DCA0204, Módulo 7
Árvores balanceadas

Daniel Aloise

baseado em slides do prof. Leo Liberti, École Polytechnique, França

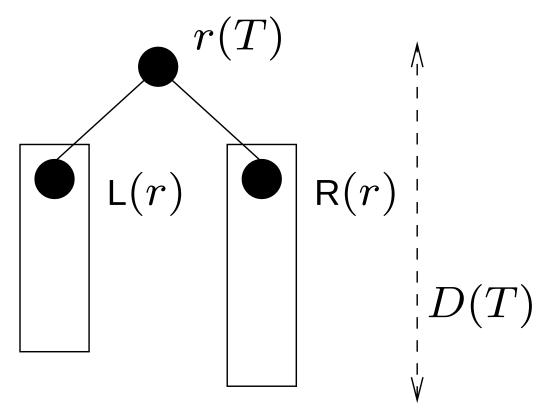
DCA, UFRN

Sumário

- Árvores binárias de busca (*Binary Search Trees(BST)*)
- Árvores AVL
- Heaps e filas de prioridade

Notação

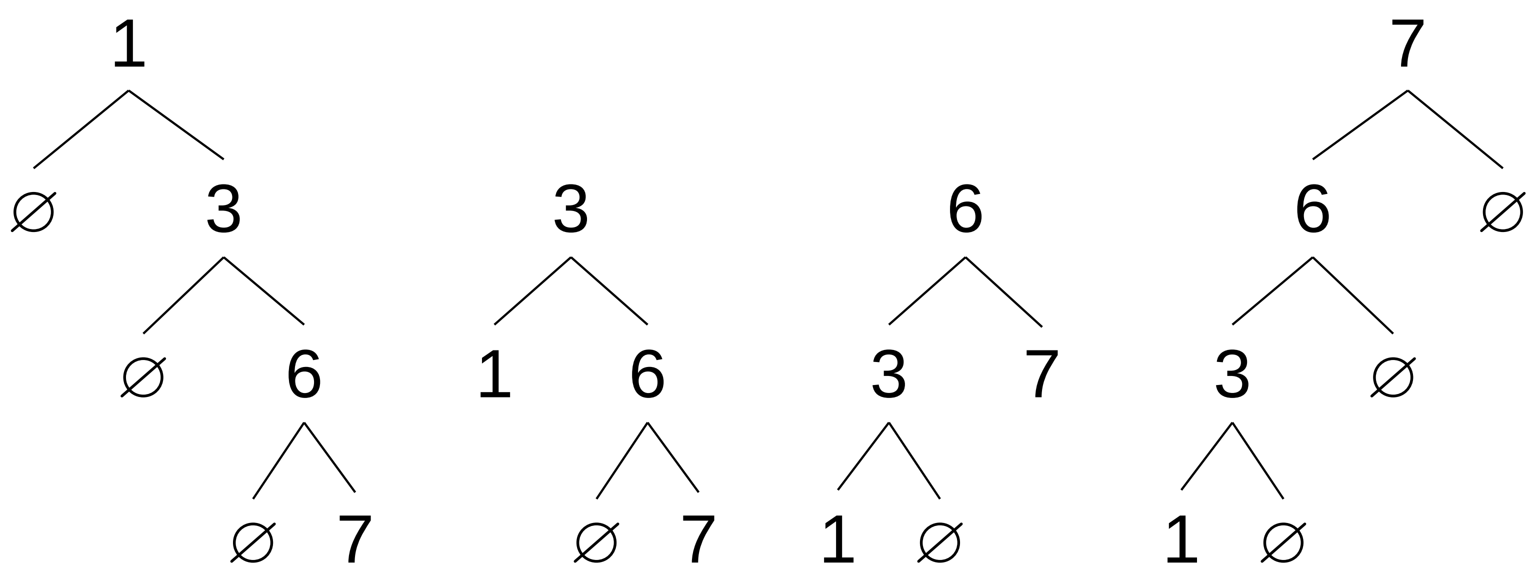
- Árvore T
- Conjunto de nós de T: $\mathsf{V}(T)$ (com $|\mathsf{V}(T)| = n$)
- Raiz: $r(T)$
- Árvore rotacionada em v: $\mathsf{T}(v)$
- Nó: $v \in \mathsf{V}(T)$
- Raiz da subárvore esquerda de v: $\mathsf{L}(v)$
- Raiz da subárvore direita de v: $\mathsf{R}(v)$
- Se $\mathsf{L}(v) = \mathsf{R}(v) = \emptyset$, v é uma **folha**
- Nó pai de v: $P(v)$
- $\Rightarrow T = \langle \mathsf{L}(r(T)), r(T), \mathsf{R}(r(T)) \rangle$
- Para todo $v \in \mathsf{V}(T)$: $p(v) =$ caminho único $r(T) \rightarrow v$
- Comprimento: $\lambda(T) = \sum_{v \in \mathsf{V}(T)} |p(v)|$
- Profundidade (ou altura): $D(T) = \max_{v \in \mathsf{V}(T)} |p(v)|$



Árvores Binárias de Busca (BST)

Sequências ordenadas

- Usada para armazenar um conjunto V como uma **sequência ordenada**
- Torna eficiente responder a questão $v \in V$
- Cada nó v na árvore é tal que $\mathsf{L}(v) < v \leq \mathsf{R}(v)$
- Exemplo: $V = \{1, 3, 6, 7\}$

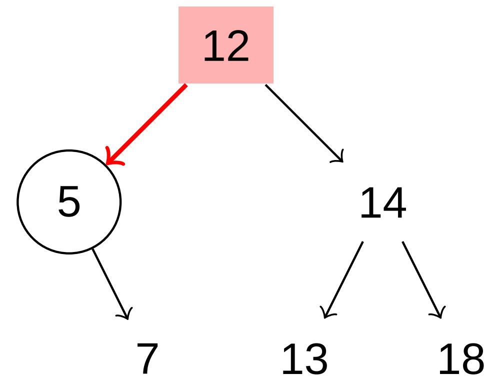


- Várias possibilidades

BST `min/max`

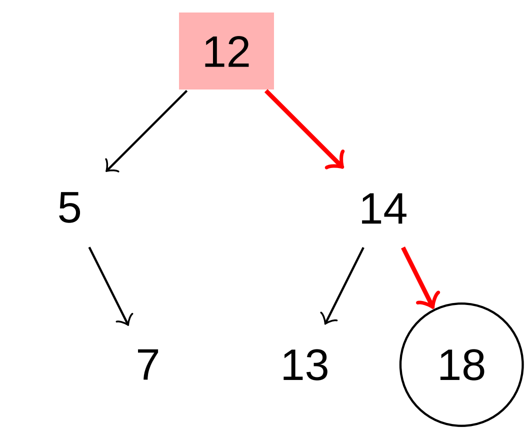
- $\texttt{min}(v)$:

if $\mathsf{L}(v) = \varnothing$ **then**
  return v;
else
  return $\min(\mathsf{L}(v))$;
end if



- $\texttt{max}(v)$:

if $\mathsf{R}(v) = \varnothing$ **then**
  return v;
else
  return $\max(\mathsf{R}(v))$;
end if



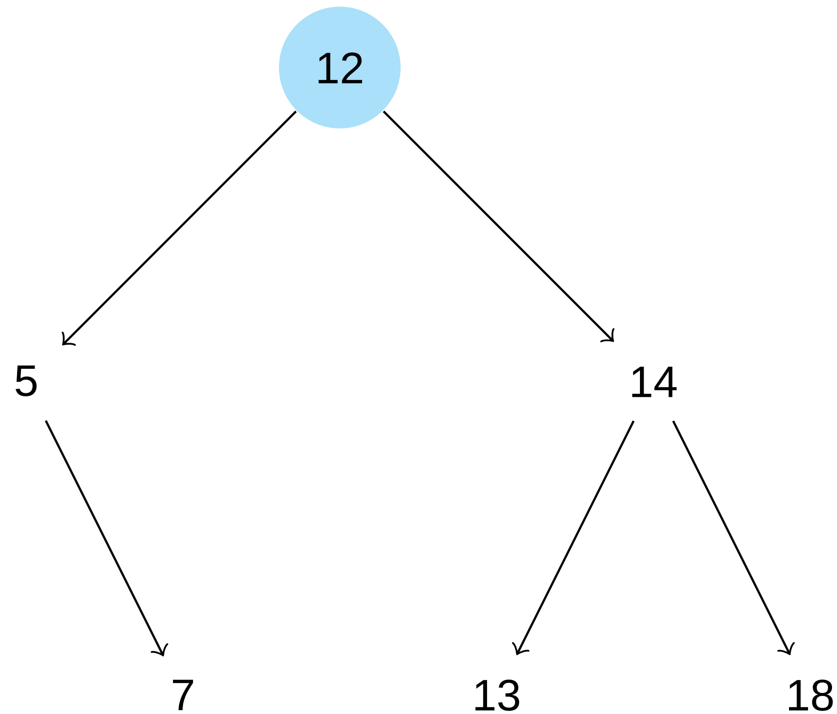
BST `find`

- `find`(k, v):

```
ret = not_found;
if v = k  (i.e. v armazena k) then
  ret = v;
else if k < v then
  ret = find(k, L(v));
else
  ret = find(k, R(v));
end if
return ret;
```

Successful find

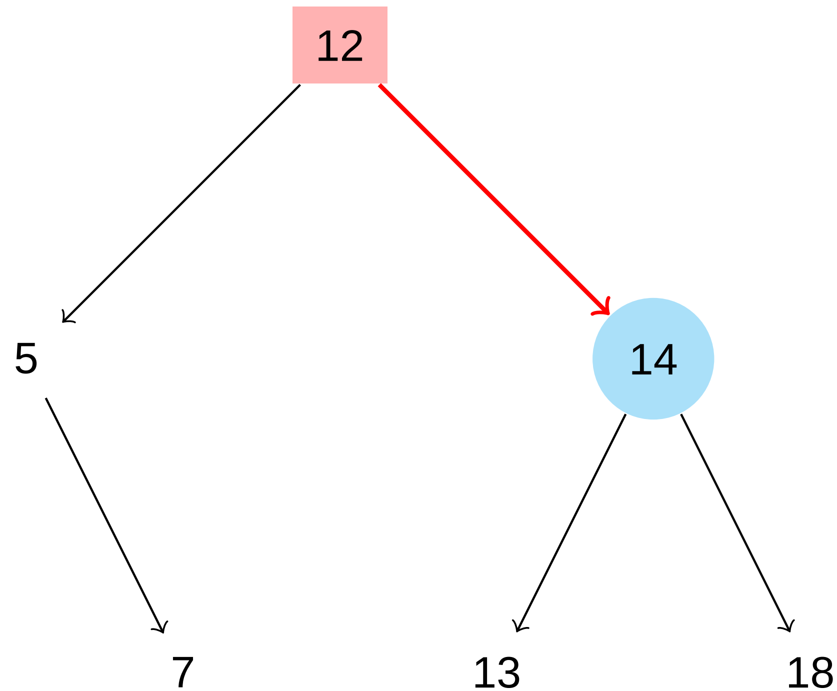
$\texttt{find}(13, r(T))$



$13 > 12$, vá para o branch direito

Successful find

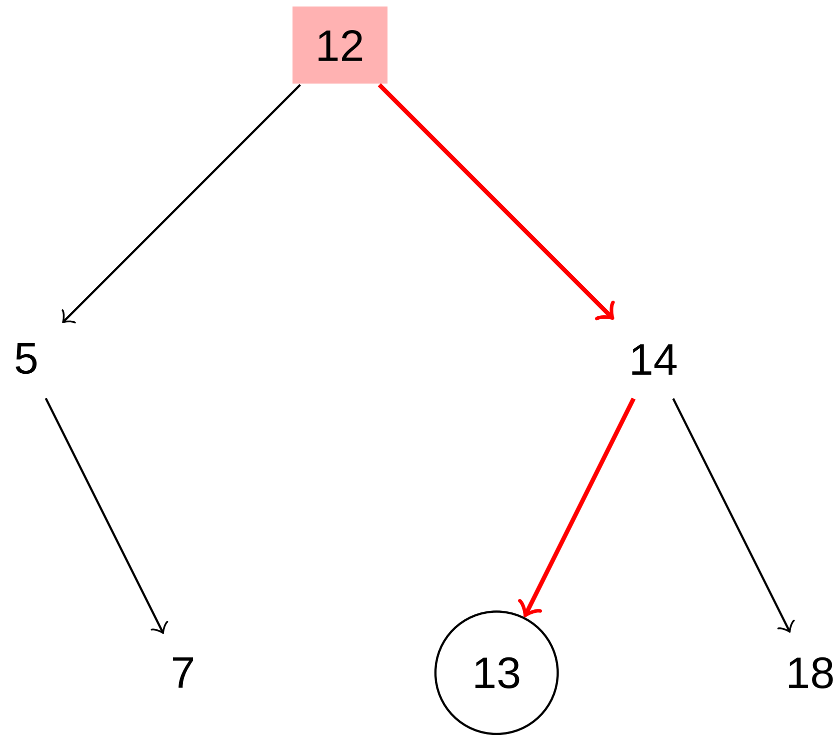
$\texttt{find}(13, r(T))$



$13 < 14$, vá para o branch esquerdo

Successful find

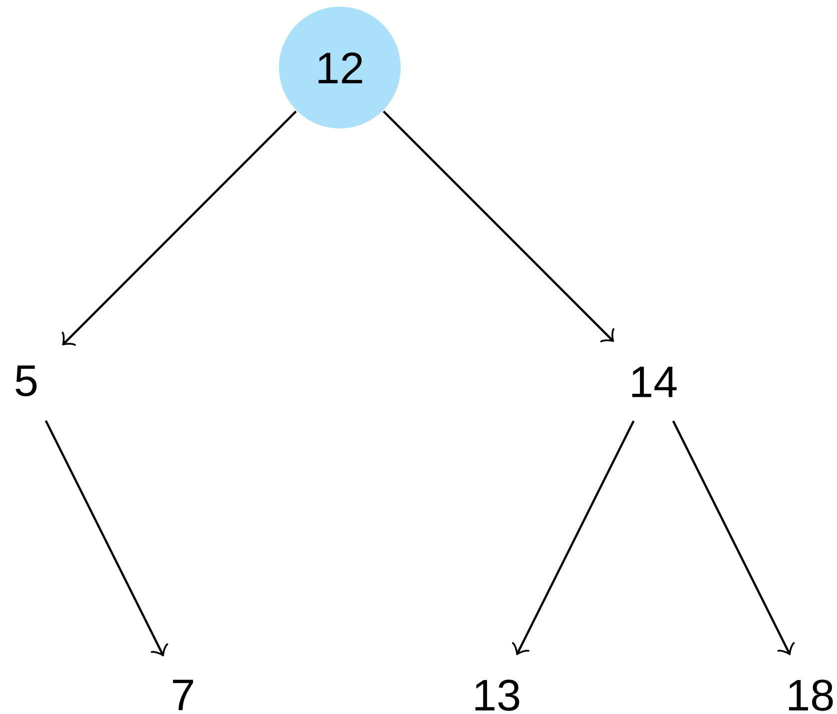
$\texttt{find}(13, r(T))$



found 13

Unsuccessful find

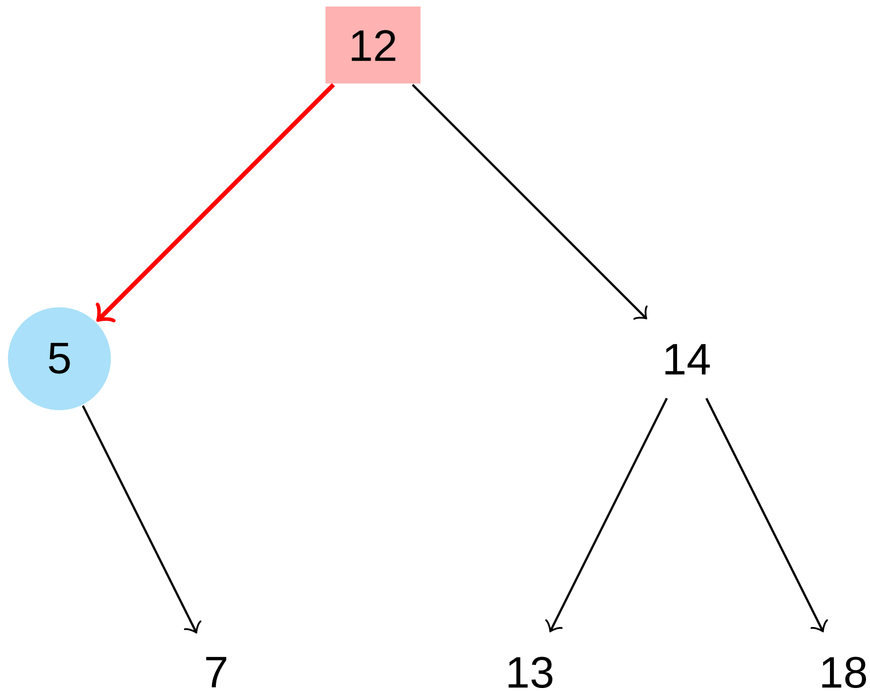
$\texttt{find}(1, r(T))$



$1 < 12$, vá para o branch esquerdo

Unsuccessful find

$\texttt{find}(1, r(T))$



$1 < 5$, deve ir para o branch esquerdo porém $\texttt{L}(5) = \varnothing$, not found

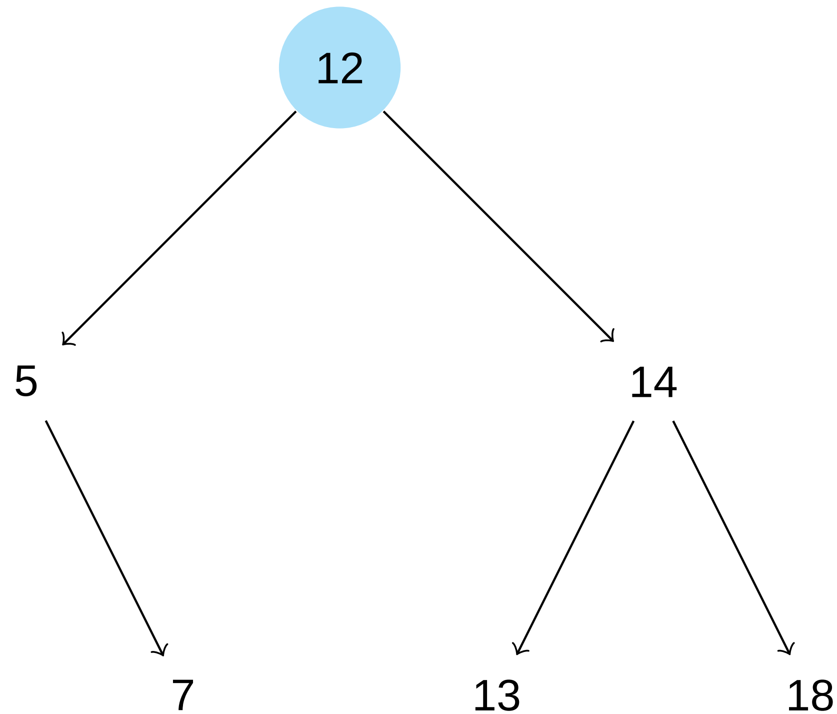
BST insert

- insert(k, v):

if $k = v$ **then**
  return **already_in_set**; // se multiconjunto: adicione novo n
else if $k < v$ **then**
  if L$(v) = \varnothing$ **then**
    L$(v) = k$;
  else
    insert$(k,$ L$(v))$;
  end if
else
  if R$(v) = \varnothing$ **then**
    R$(v) = k$;
  else
    insert$(k,$ R$(v))$;
  end if
end if

Exemplo Insert

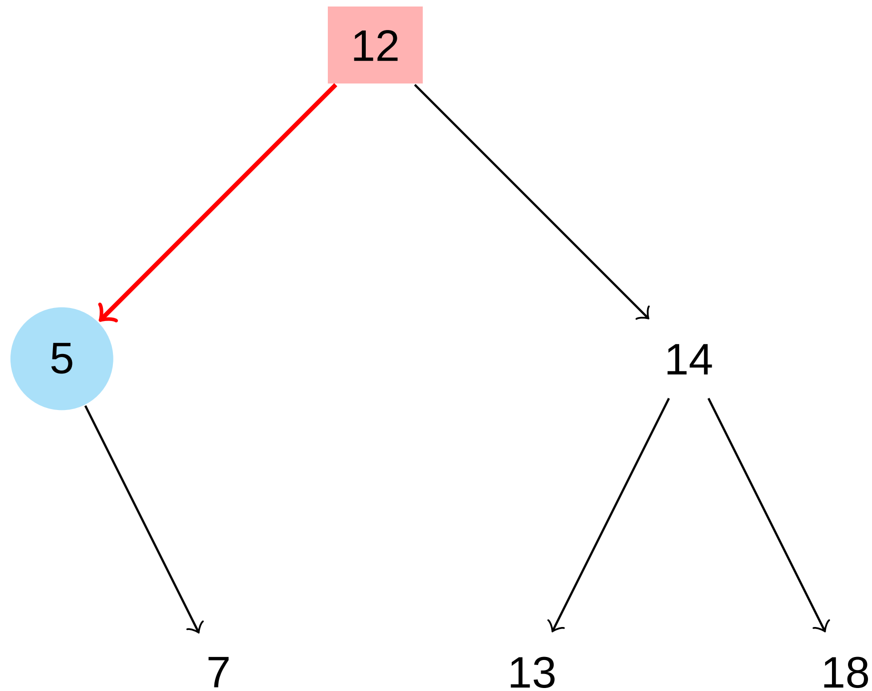
$\texttt{insert}(1, r(T))$



$1 < 12$, vá para o branch esquerdo

Exemplo Insert

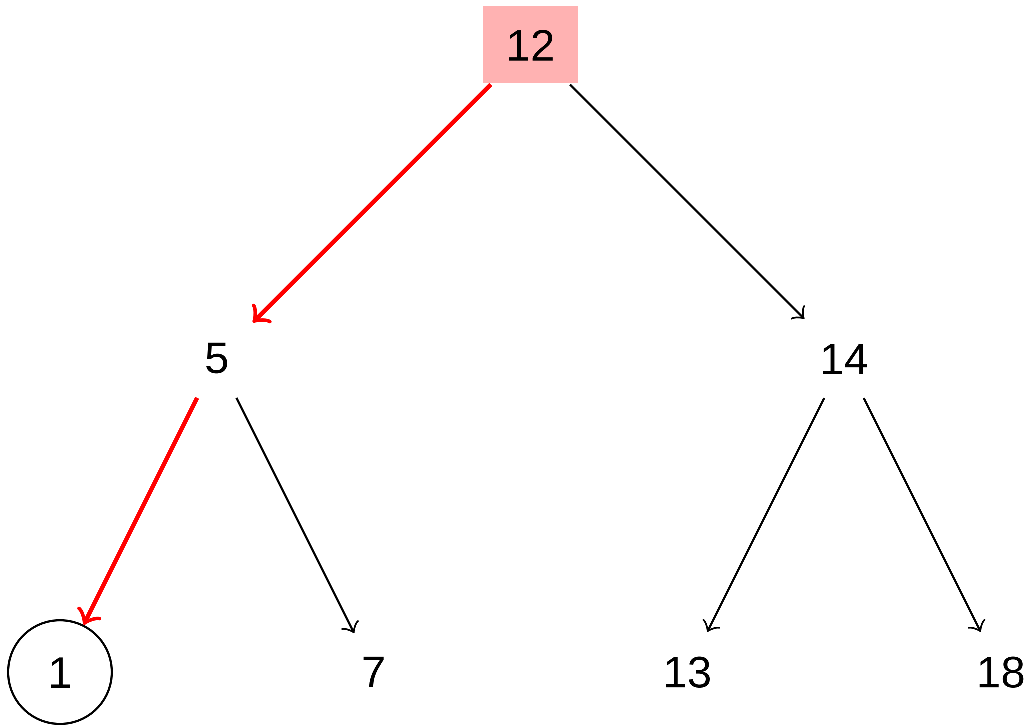
$\texttt{insert}(1, r(T))$



$1 < 5$, deve ir para o branch esquerdo porém $\texttt{L}(5) = \emptyset$

Exemplo Insert

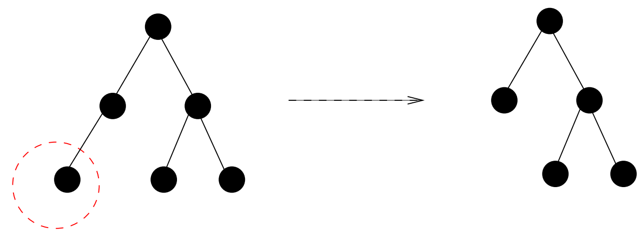
$\texttt{insert}(1, r(T))$



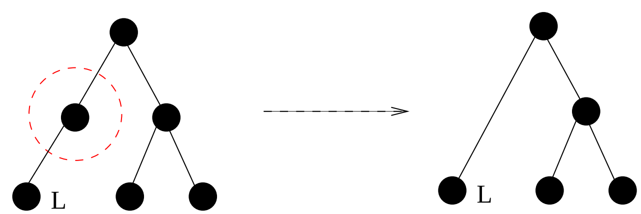
Adicione $k = 1$ como $\mathsf{L}(5)$

Remoção não é tão fácil

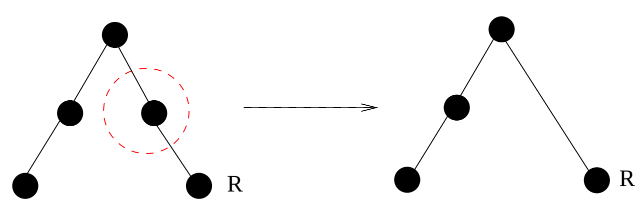
- Se nó v removido é uma folha, fácil: "corte" ele (`unlink`)



- Se $\mathsf{R}(v) = \varnothing$ e $\mathsf{L}(v) \neq \varnothing$, troque-o por $\mathsf{L}(v)$



- Se $\mathsf{L}(v) = \varnothing$ e $\mathsf{R}(v) \neq \varnothing$, troque-o por $\mathsf{R}(v)$



- Se v tem as duas subárvores, não é evidente

Troca de um nó

Troca link $\{P(v), v\}$ por $\{P(v), u\}$, então desconecta v

- `replace`(v, u) `// troca` v `por` u

if $\mathtt{R}(P(v)) = v$ **then**
  $\mathtt{R}(P(v)) \leftarrow u$; `// ` u ` é um subnó direito`
else
  $\mathtt{L}(P(v)) \leftarrow u$; `// ` u ` é um subnó esquerdo`
end if
if $u \neq \emptyset$ **then**
  $P(u) \leftarrow P(v)$;
end if
`unlink` v;

Remove v : $\mathsf{L}(v) \neq \varnothing \wedge \mathsf{R}(v) \neq \varnothing$

Ideia: troca v por $u = \mathtt{min}\mathsf{T}(\mathsf{R}(v))$ e então remove v

- O mínimo u de uma BST é sempre a folha mais a esquerda da árvore
- Portanto, sabemos como remover u (caso $\mathsf{L}(\cdot) = \varnothing$ do slide anterior)
- Trocamos o valor de v pelo valor de u e então removemos u
- Uma vez que $u = \min \mathsf{T}(\mathsf{R}(v))$, temos $u < w$ para todo $w \in \mathsf{T}(\mathsf{R}(v))$
- Uma vez que o valor de v é agora igual ao valor de u, v é agora o mínimo entre todos os nós em $\mathsf{T}(\mathsf{R}(v))$; portanto $v < r(\mathsf{R}(v))$
- Além disso, uma vez que o valor de v era u, um nó em $\mathsf{R}(v)$, temos $v > r(\mathsf{L}(v))$, satisfazendo a definição da BST. (†)

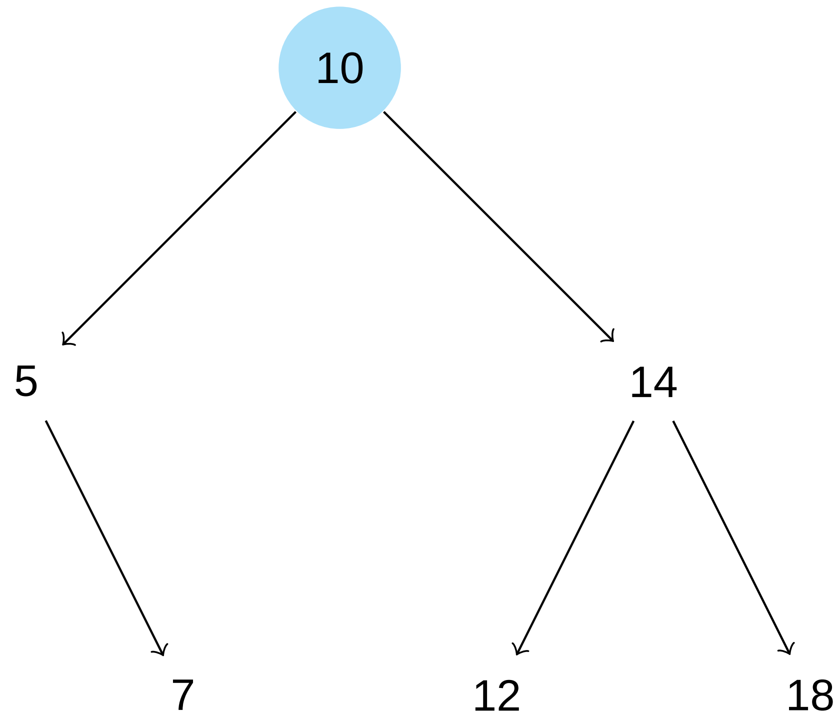
Remoção na BST

- `delete`(k, v):

```
if k < v then
  delete(k, L(v));
else if k > v then
  delete(k, R(v));
else
  if L(v) = ∅ ∨ R(v) = ∅ then
    unlink v; // um dos casos fáceis
  else
    u = min(R(v));
    replace(u, v);
    unlink u; // uma caso fácil, como L(u)=null
  end if
end if
```

Exemplo de remoção

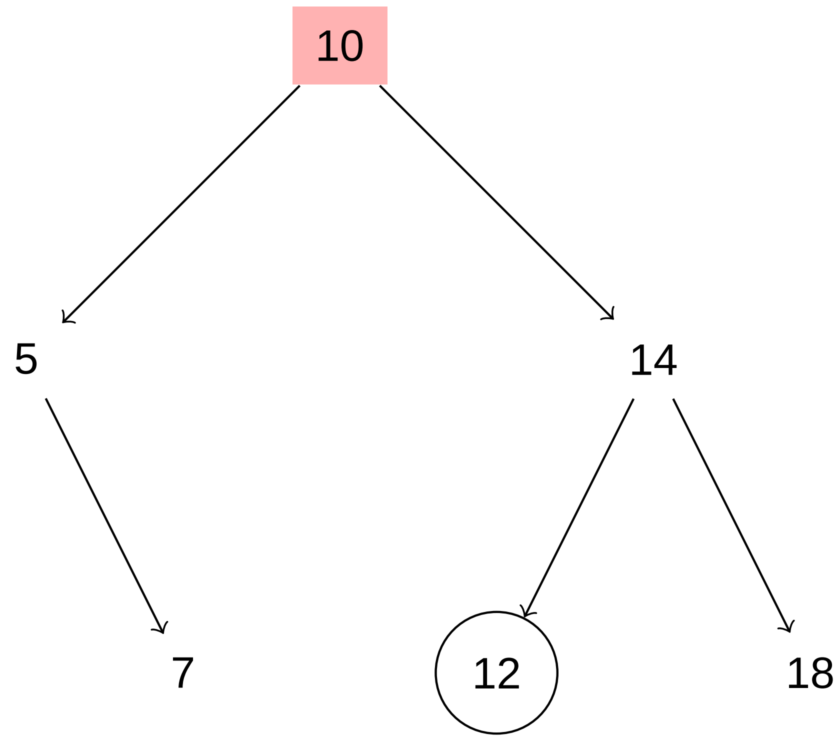
$\texttt{delete}(10, r(T))$



$$v = 10$$

Exemplo de remoção

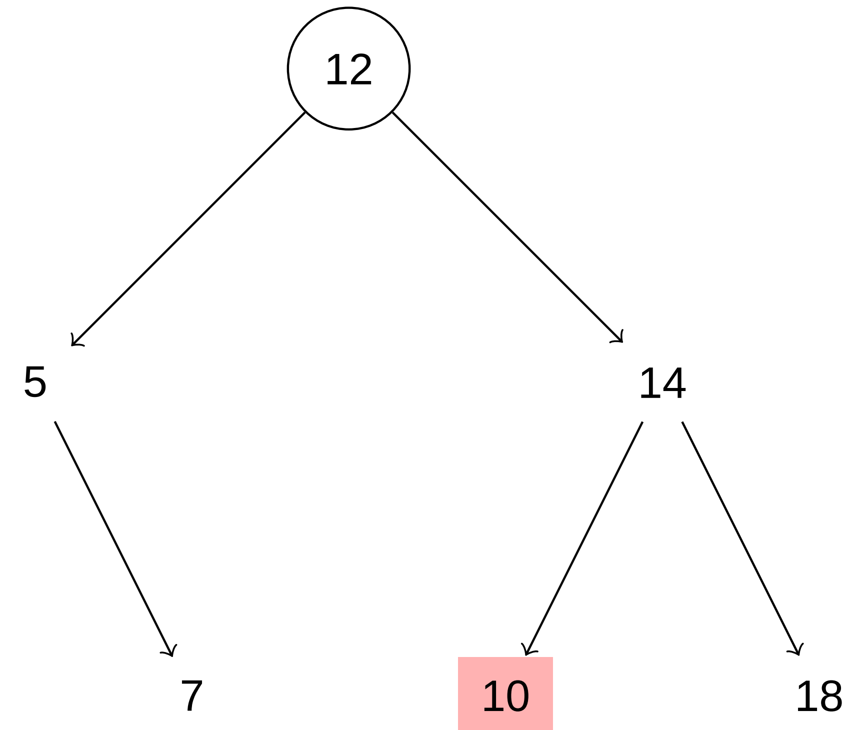
$$\texttt{delete}(10, r(T))$$



$$u = \min \mathsf{T}(14) = 12$$

Exemplo de remoção

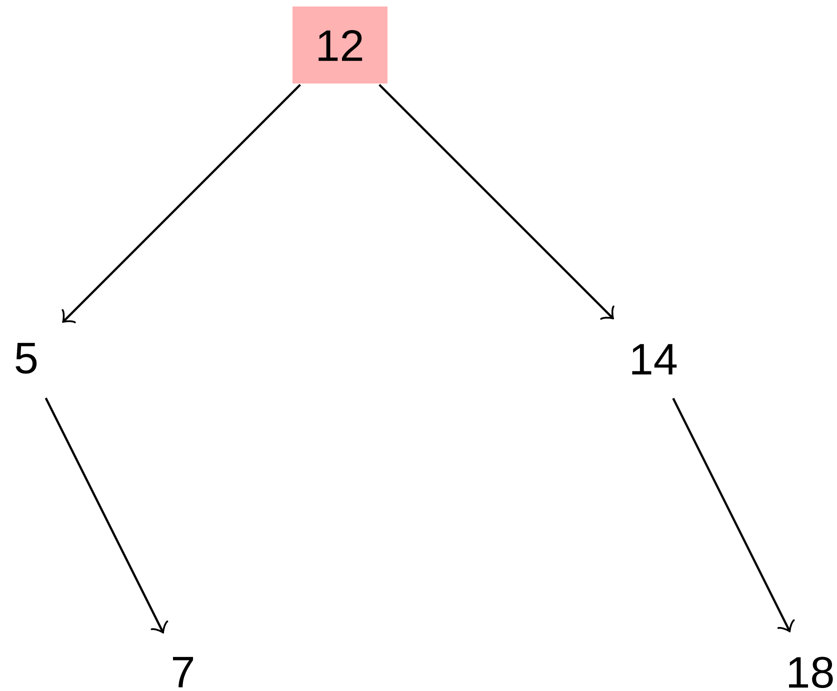
$\texttt{delete}(10, r(T))$



troca valores de 10 e 12

Exemplo de remoção

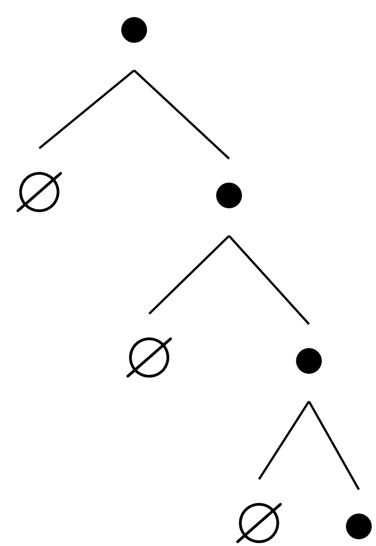
$\texttt{delete}(10, r(T))$



$\texttt{unlink}\ 10$

Complexidade

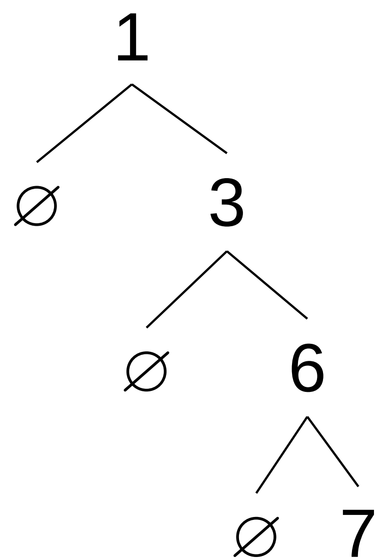
- Cada IF involve no máximo uma chamada recursiva
- Faz a recursão ao longo de um único branch
- Complexidade de pior caso proporcional à profundidade $D(T)$
- No pior caso, $D(T)$ é $O(n)$



Árvores de Adelson-Velskii & Landis (AVL)

Árvores AVL

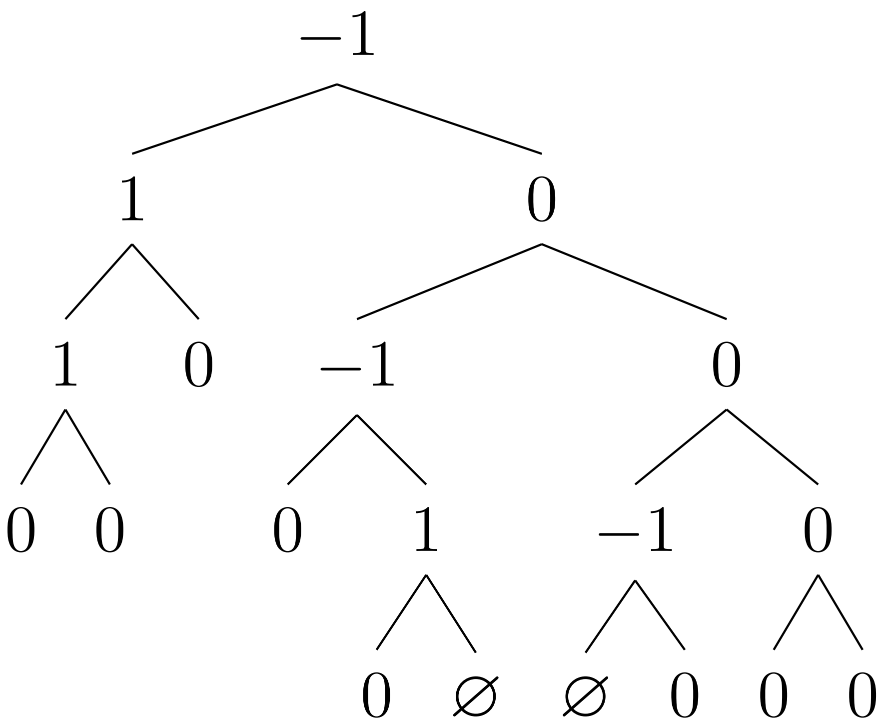
- Tente inserir $1, 3, 6, 7$ nesta ordem: obtem-se uma **árvore desbalanceada**



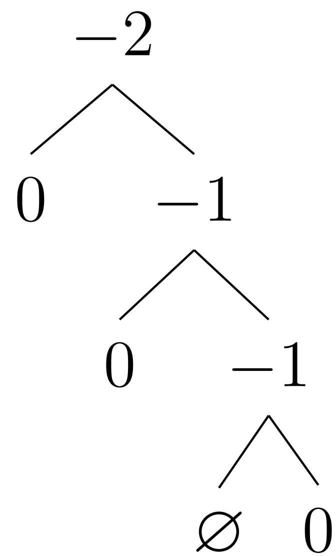
- Pior caso para find (i.e., procure a chave 7) é $O(n)$
- Necessário *rebalancear* a árvore para ser mais eficiente
- **árvores AVL**: em qualquer nó, $B(T) =$dif. de profunidade entre as subárvores esquerda e direita $\in \{-1, 0, 1\}$

Exemplos

árvore AVL:



árvore não-AVL:



Nós indicam $B(\mathsf{T}(v))$

Exemplo inserção

insert 1

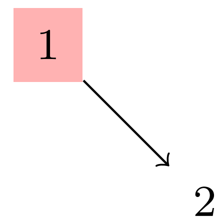
1

$v_1 = 1;$

$r(T) = v_1;$

Exemplo inserção

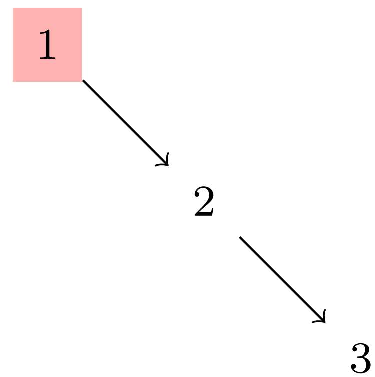
insert 2



$v_2 = 2$;
$\mathsf{R}(v_1) = v_2$;
$P(v_2) = v_1$;

Exemplo inserção

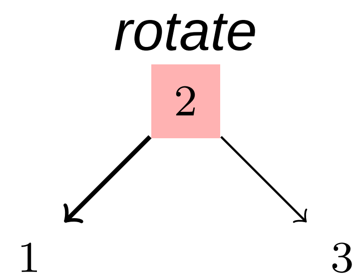
insert 3



$v_3 = 3$;
$\mathsf{R}(v_2) = v_3$;
$P(v_3) = v_2$;

$D(\mathsf{L}(v_1)) = 0$,
$D(\mathsf{R}(v_1)) = 2$:
$B(T) = -2$: desbalanceada

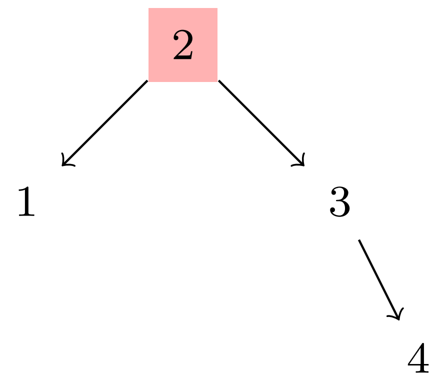
Exemplo inserção



$r(T) = v_2$;
$\mathsf{L}(v_2) = v_1$;
$P(v_1) = v_2$;

Exemplo inserção

insert 4



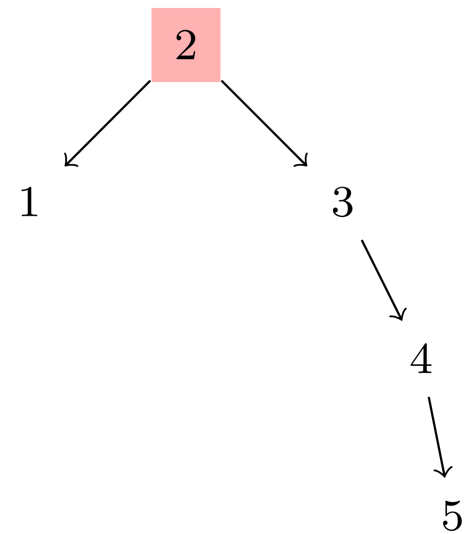
$v_4 = 4$;

$\mathsf{R}(v_3) = v_4$;

$P(v_4) = v_3$;

Exemplo inserção

insert 5

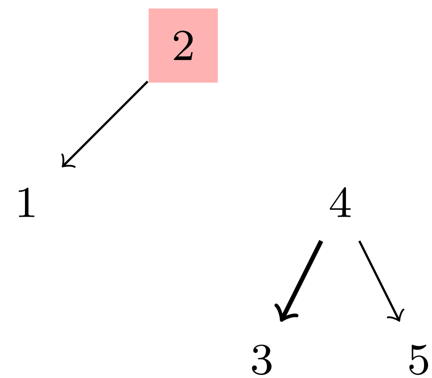


$v_5 = 5$;
$\mathsf{R}(v_4) = v_5$;
$P(v_5) = v_4$;

$D(\mathsf{L}(v_3)) = 0$,
$D(\mathsf{R}(v_3)) = 2$:
$B(T) = -2$: desbalanceada

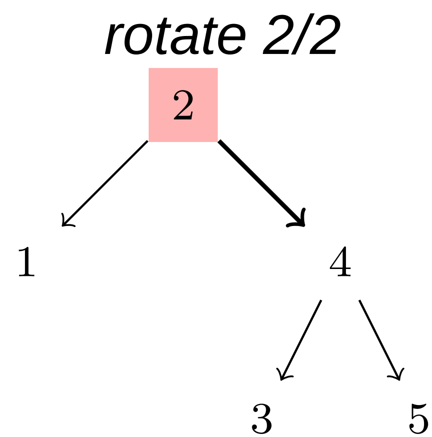
Exemplo inserção

rotate 1/2



$\mathsf{L}(v_4) = v_3$;
$P(v_3) = v_4$;

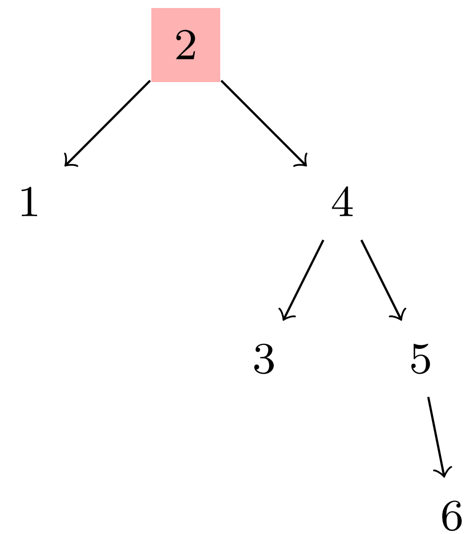
Exemplo inserção



$\mathsf{R}(v_2) = v_4$;
$P(v_4) = v_2$;

Exemplo inserção

insert 6



$v_6 = 6$;

$\mathsf{R}(v_5) = v_6$;

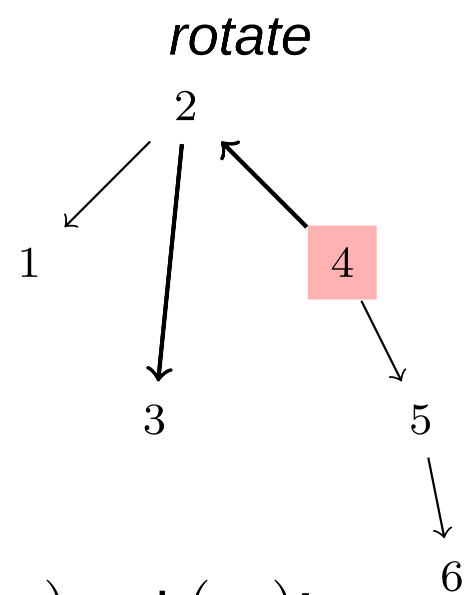
$P(v_6) = v_5$;

$D(\mathsf{L}(v_2)) = 1$,

$D(\mathsf{R}(v_2)) = 3$:

$B(T) = -2$: desbalanceada

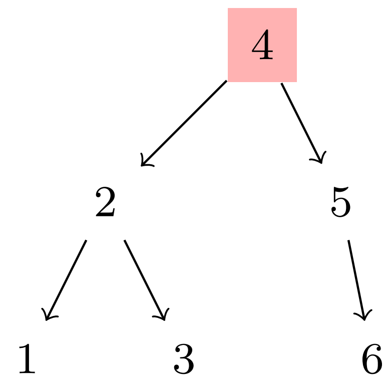
Exemplo inserção



$\mathsf{R}(v_2) = \mathsf{L}(v_4)$;
$P(\mathsf{L}(v_4)) = v_2$;
$\mathsf{L}(v_4) = v_2$;
$P(v_2) = v_4$;
$r(T) = v_4$;

Exemplo inserção

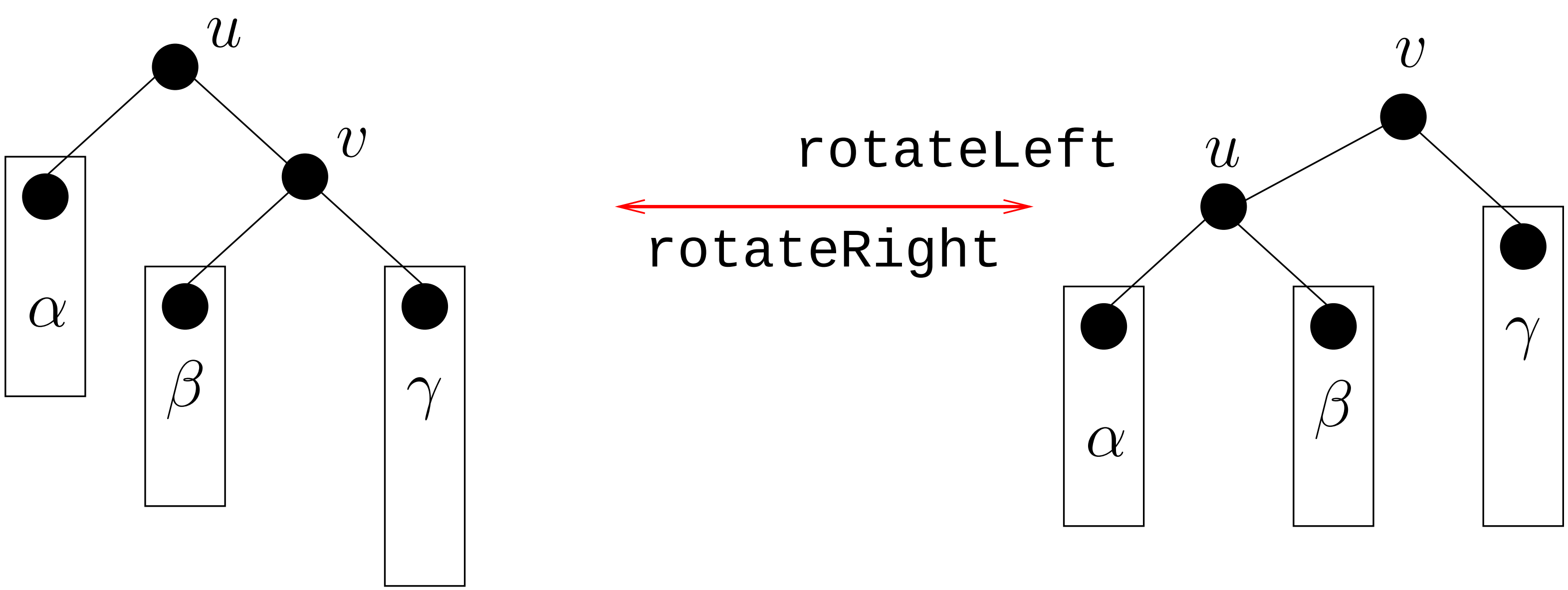
resultado



Em geral

- Podemos decompor as operações de uma árvore balanceada:
  - na operação em si
  - em uma sequência de *operações de rebalanceamento* (quando necessário), denominadas **rotações**
- As operações `min/max, find, insert, delete` são iguais as da BST.
- Desbalanceamento pode ocorrer na inserção e remoção
- Uma vez que inserimos/removemos um nó por vez, o desbalanceamento é de no max. 1 unidade
- I.e., $B(T) \in \{-2, 2\}$

Rotação à esquerda e à direita



Interpretação algébrica

- Sejam α, β, γ árvores, u, v são nós fora de α, β, γ
- Defina:
  - $\texttt{rotateLeft}(\langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle) = \langle \langle \alpha, u, \beta \rangle, v, \gamma \rangle$
  - $\texttt{rotateRight}(\langle \langle \alpha, u, \beta \rangle, v, \gamma \rangle) = \langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle$
- Uma espécie de “associatividade de árvores”
- Obs: `rotateLeft`, `rotateRight` são operações inversas

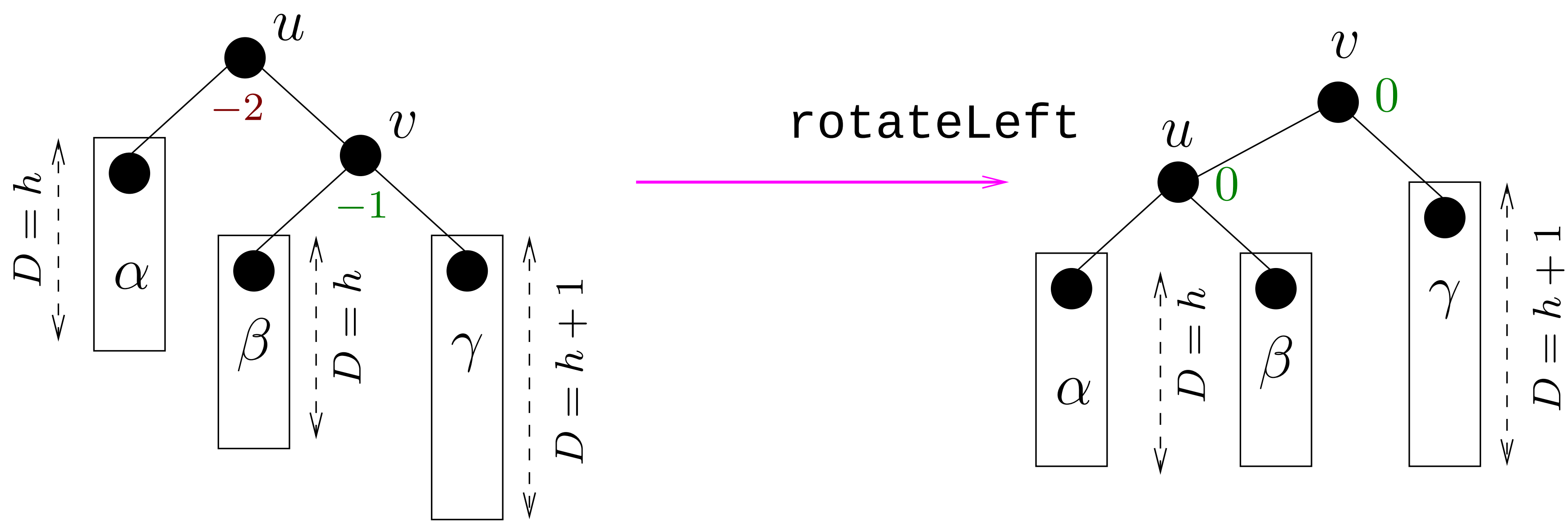
Thm.

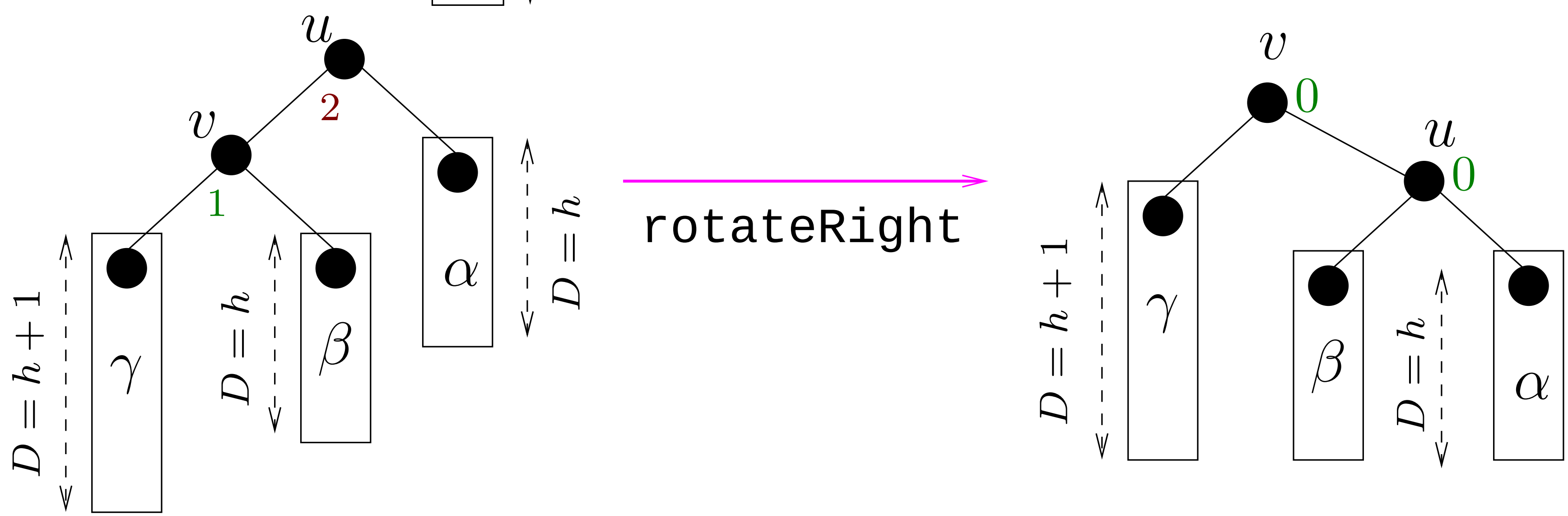
$\texttt{rotateRight}(\texttt{rotateLeft}(T)) =$
$\texttt{rotateLeft}(\texttt{rotateRight}(T)) = T$

Proof

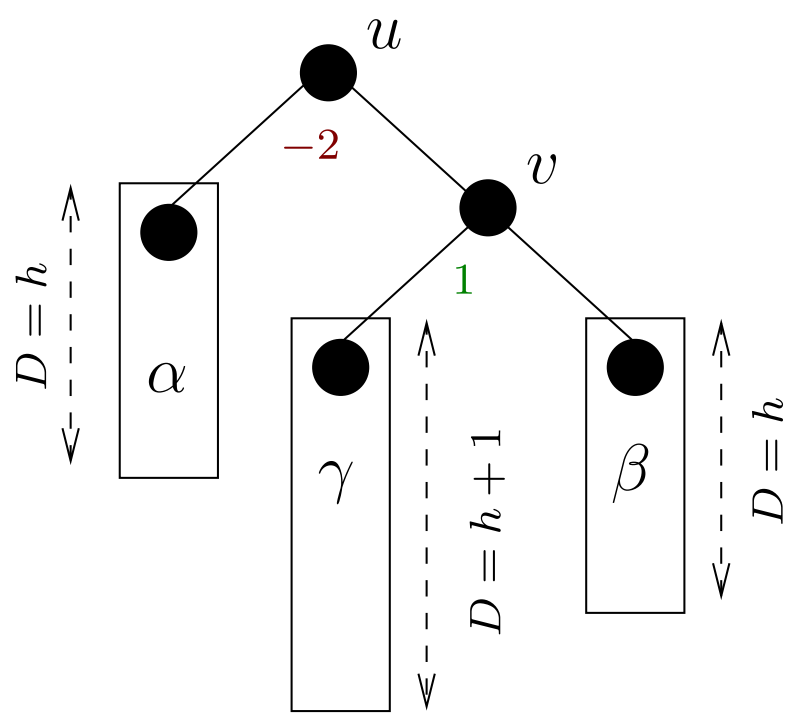
Diretamente da definição

Rotacionando e rebalanceando



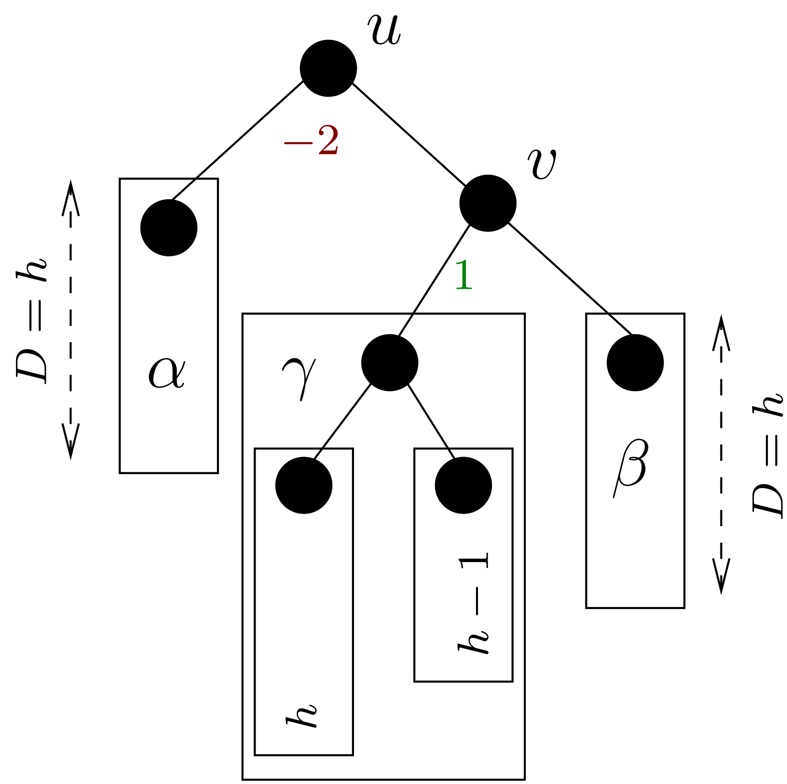


Isto é suficiente?



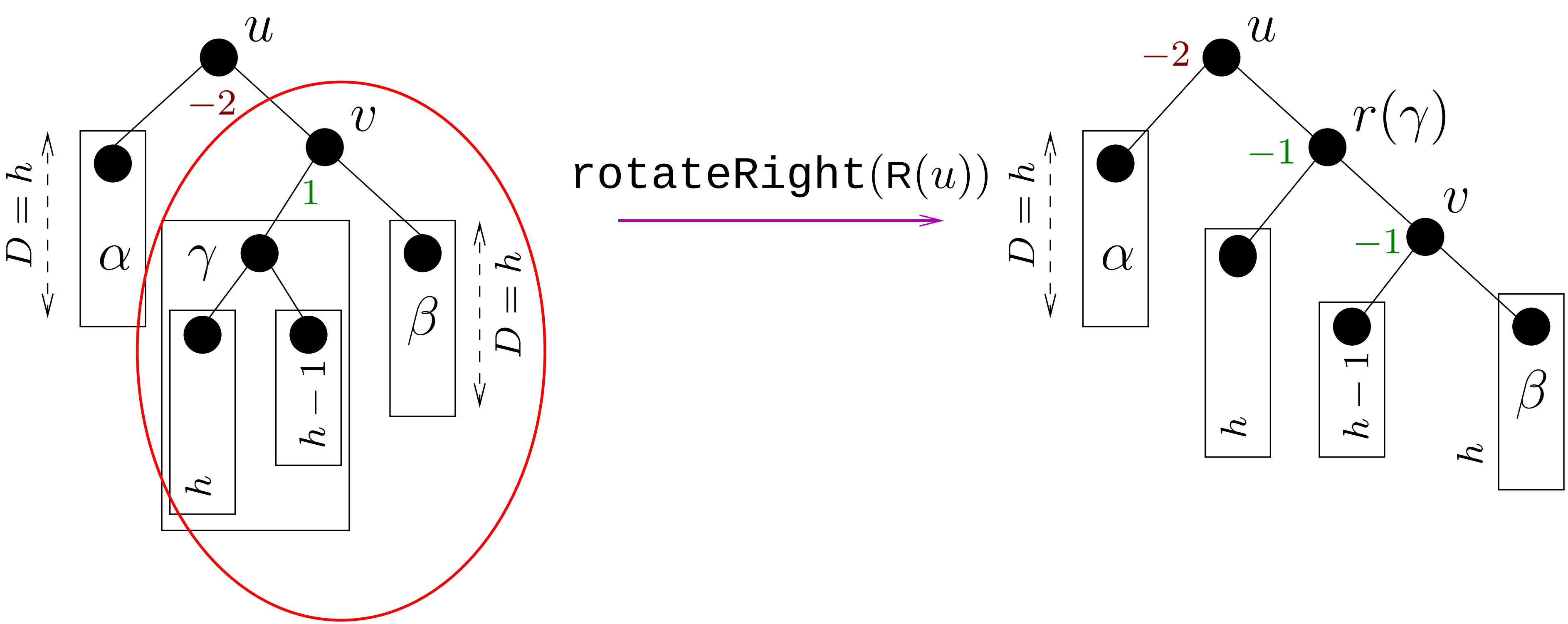
A rotação deixa γ no seu local, não funciona

Quebrar γ em subárvores



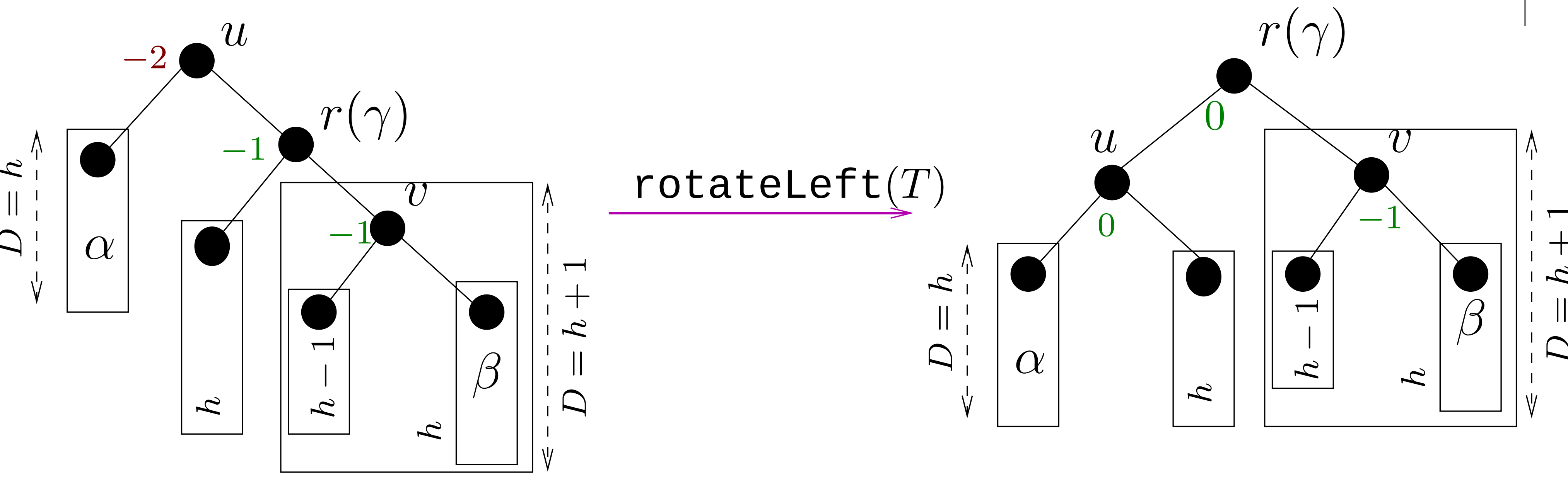
Agora podemos rotacionar $\mathsf{T}(v) = \mathsf{R}(u)$

Rotacione a subárvore direita



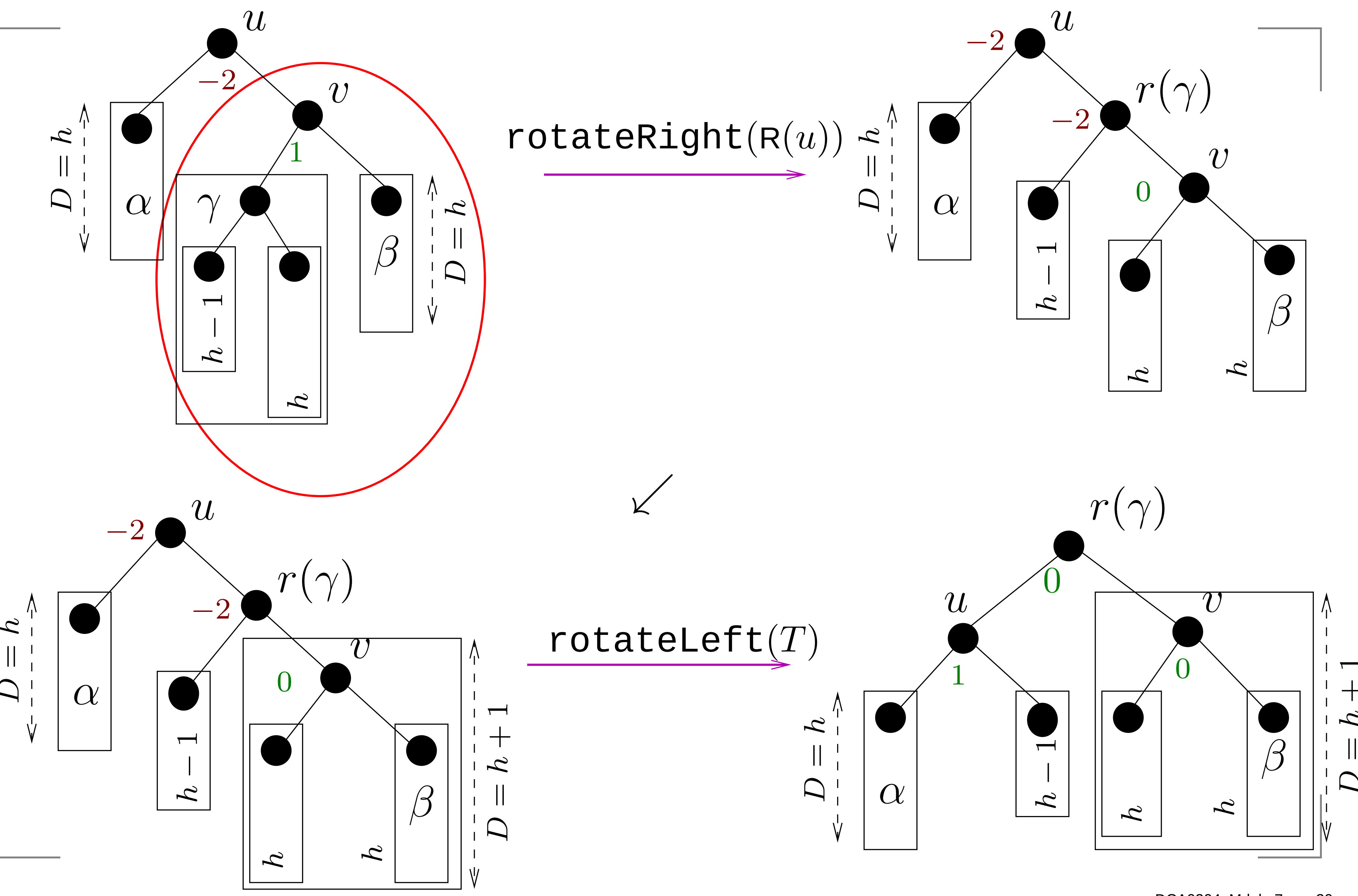
Rotacione R(u) à direita

Finalmente, rotacione à esquerda

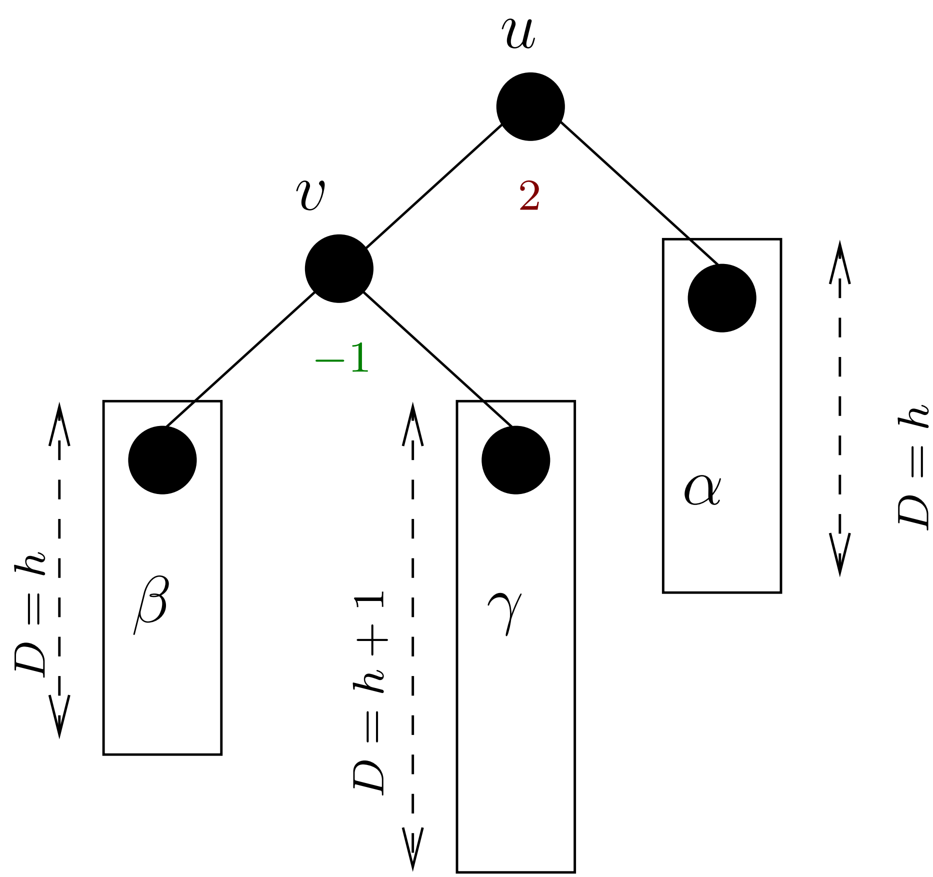


Rotacione T à esquerda

Casos simétricos I



Casos simétricos II



Rebalanceamento: `rotateLeft`$($`L`$(u))$, `rotateRight`(T)

Implementação de árvores AVL

Trabalhosa pra caramba!

- **Implementação em Java do prof. Leo Liberti disponível no SIGAA**

Balanceada vs. random BST

- Árvores binárias de busca balanceadas tem ops $O(\log n)$
- O que podemos dizer sobre as BST na média (não necessariamente balanceadas)?
- Dada uma sequência $\sigma \in \{1, \ldots, n\}^n$, inserimos ela em uma BST T
- Os nós à esquerda de $r(T)$ são $\leq r(T)$, nós à direita são $> r(T)$
- Seja K o número de nós em $\mathsf{L}(T)$, de modo que $|\mathsf{R}(T)| = n - 1 - K$
- Distribuição uniforme em K i.e. $P(K = k) = \frac{1}{n}$ para todo $k \in \{0, \ldots, n-1\}$

σ	(1,2,3)	(1,3,2)	(2,1,3)	(2,3,1)	(3,1,2)	(3,2,1)
T	$\begin{array}{ccccc}1&&&&\\&\searrow&&&\\&&2&&\\&&&\searrow&\\&&&&3\end{array}$	$\begin{array}{ccc}1&&\\&\searrow&\\&&3\\&\swarrow&\\2&&\end{array}$	$\begin{array}{ccccc}&&2&&\\&\swarrow&&\searrow&\\1&&&&3\end{array}$	$\begin{array}{ccccc}&&2&&\\&\swarrow&&\searrow&\\1&&&&3\end{array}$	$\begin{array}{ccc}&&3\\&\swarrow&\\1&&\\&\searrow&\\&&2\end{array}$	$\begin{array}{ccccc}&&&&2\\&&&\swarrow&\\&&1&&\\&\swarrow&&&\\3&&&&\end{array}$
tipo	A	B	C	C	D	E

Tipo C (balanceado) tem prob. 2x maior do que qualquer outro tipo!

Profundidade média

- Profundidade média para BSTs: $O(\log n)$ [Devroye, 1986]
- Isto mostra que BSTs são bem balanceadas na média

Heaps e filas de prioridade

Lembrete de filas

- Uma **fila** é um estrutura de dados com operações principais:
  - `pushBack`(v): insere v no fim da fila
  - `popFront`$()$: retorna e remove um elemento do começo da fila
- Filas implementam o princípio First-In-First-Out
- Usado por BFS (ver Módulo 2)
- Se arcos são priorizados (ex. através de tempos de percurso), queremos que a fila retorne o <u>*elemento de mais alta prioridade*</u>

  Ele pode não estar no começo da fila

Filas de prioridade

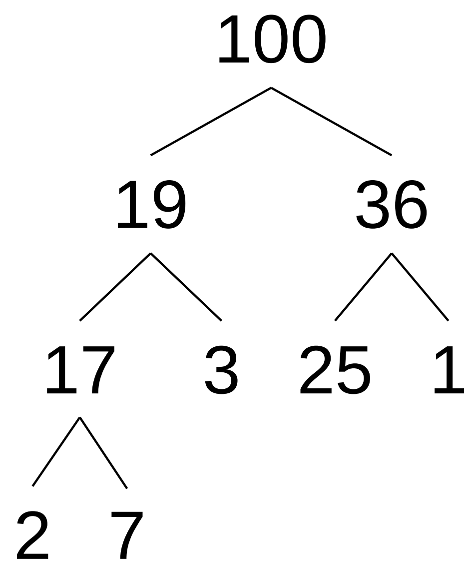
- Seja V um conjunto e $(S, <)$ um conjunto totalmente ordenado
- **fila de prioridade** em V, S: conjunto Q de pares (v, p_v) tal que $v \in V$ e $p_v \in S$
- Usualmente, p_v é um número
- ex. se p_v é o rank de entrada de v em Q, então Q é uma fila padrão
- Suporta três operações principais:
  - `insert`(v, p_v): insere v em Q com prioridade p_v
  - `max`$()$: retorna o elemento de Q com prioridade máxima
  - `popMax`$()$: retorna e remove `max`$()$
- Implementado como **heaps**

Heap

- Uma **heap** binária é uma estrutura de dados abstrata (pensada como uma árvore) que oferece:
  - $O(\log |Q|)$ `insert`
  - $O(1)$ `max`
  - $O(\log |Q|)$ `popMax`
- $O(1)$ é obtido armazenando-se o elemento de prioridade máxima como raiz da árvore binária
- Propriedades principais
  - *shape property*: todos os níveis exceto talvez o último são completamente preenchidos; o último nível é preenchido da esquerda para direita.
  - *heap property*: todo nó armazena um elemento de maior prioridade do que seus subnós.

Exemplo

Seja $V = \mathbb{N}$, e para todo $v \in V$ seja $p_v = v$



Uma árvore balanceada

Thm.

Se Q é uma heap binária, $B(Q) \in \{0, 1\}$

Proof

Isto segue trivialmente da shape property. Uma vez que todos os níveis são completamente preenchidos (com exceção talvez do último), $B(Q) \in \{-1, 0, 1\}$. Uma vez que o último é preenchido da esquerda-para-direita, $B(Q) \neq -1$

Cor.

Uma heap binária é uma árvore binária balanceada

Atenção: *NÃO uma BST/AVL*: **heap property não compatível com definição da BST** $\mathsf{L}(v) \leq v \leq \mathsf{R}(v)$

Manter a heap balanceada: $O(\log |Q|)$ para inserção/remoção

Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

$\varnothing$

Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no "slot" mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v's posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

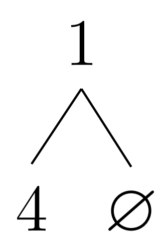
insert 1 $\quad\quad$ 1

Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

insert 4

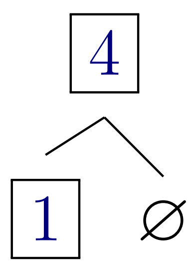


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

$1 < 4$, *swap*

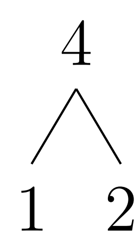


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

insert 2

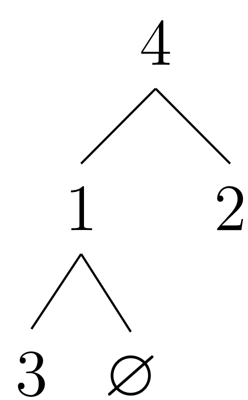


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

insert 3

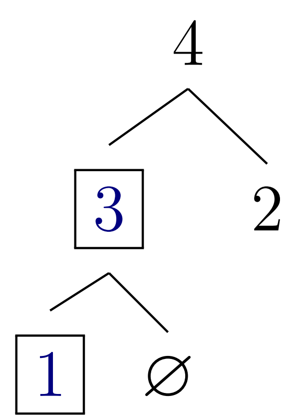


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

$1 < 3$, *swap*

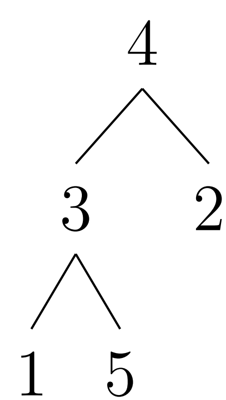


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no “slot” mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v’s posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

insert 5

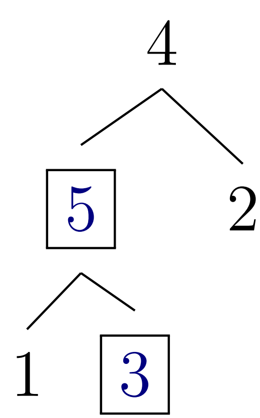


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no "slot" mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v's posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

$3 < 5$, *swap*

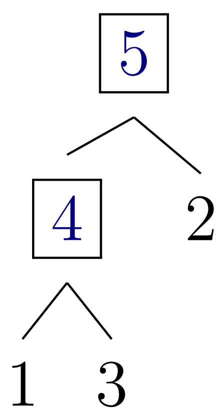


Inserção

- Adiciona novo elemento (v, p_v) na base da heap (último nível, no "slot" mais à esquerda livre)
- Compara com o seu (único) pai (u, p_u); se $p_u < p_v$, swap u e v's posições na heap
- Repita comparação/swap até que a heap property se estabeleça

Exemplo: insert $(1, 4, 2, 3, 5)$

$4 < 5$, *swap*



Inserção mantem a heap

- Pior caso: `insert` leva tempo proporcional à altura da árvore: $O(\log n)$
- A shape property é mantida:
  - quando adiciona-se um novo elemento no último nível no slot livre mais à esquerda
- A propriedade da heap não é mantida depois de adicionar um novo elemento
- Entretanto, ela é restabelecida depois de uma sequência de swaps

Thm.

A operação de inserção mantem a heap

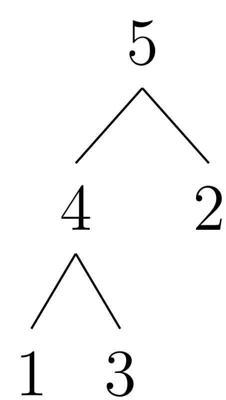
Max

- *Fácil*: retorna a raiz da heap
- Evidentemente $O(1)$

Remoção do max

- Seja $\texttt{last}(Q)$ o elemento mais à direita da heap em seu último nível
- Mova nó $\texttt{last}(Q)$ para a raiz $r(Q)$
- Compare v com seus filhos u, w: se $p_v \geq p_u, p_v \geq p_w$, heap está na ordem correta
- C.c., swap v com $\max_p(u, v)$ (*use* $\min_p$ *se min-heap)* e repita comparação/swap até terminação

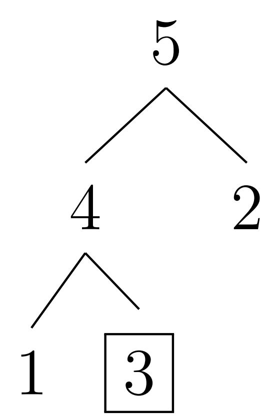
árvore original



Remoção do max

- Seja $\texttt{last}(Q)$ o elemento mais à direita da heap em seu último nível
- Mova nó $\texttt{last}(Q)$ para a raiz $r(Q)$
- Compare v com seus filhos u, w: se $p_v \geq p_u, p_v \geq p_w$, heap está na ordem correta
- C.c., swap v com $\max_p(u, v)$ (*use* $\min_p$ *se min-heap)* e repita comparação/swap até terminação

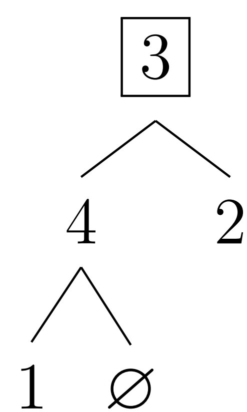
$\texttt{last}(Q) = 3$



Remoção do max

- Seja $\texttt{last}(Q)$ o elemento mais à direita da heap em seu último nível
- Mova nó $\texttt{last}(Q)$ para a raiz $r(Q)$
- Compare v com seus filhos u, w: se $p_v \geq p_u, p_v \geq p_w$, heap está na ordem correta
- C.c., swap v com $\max_p(u, v)$ (*use* $\min_p$ *se min-heap)* e repita comparação/swap até terminação

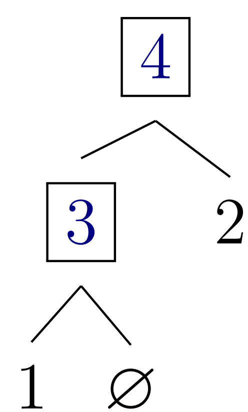
move to root



Remoção do max

- Seja `last`(Q) o elemento mais à direita da heap em seu último nível
- Mova nó `last`(Q) para a raiz $r(Q)$
- Compare v com seus filhos u, w: se $p_v \geq p_u, p_v \geq p_w$, heap está na ordem correta
- C.c., swap v com $\max_p(u, v)$ (*use* $\min_p$ *se min-heap*) e repita comparação/swap até terminação

$3 < 4$, *swap*



Construção eficiente

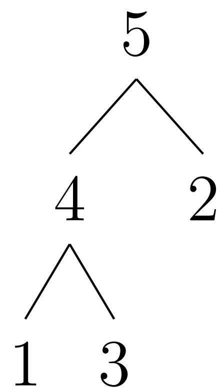
- Suponha que temos n elementos de V para inserir em uma heap vazia
- Trivialmente: cada insert leva $O(\log n)$, levamos $O(n \log n)$ para construir a heap inteira
- DÃą para fazer melhor:
    1. coloque arbitrariamente os elementos em uma árvore binária mantendo a shape property (pode ser feito em $O(n)$)
    2. a partir dos níveis mais baixos, mova os nós para baixo usando o mesmo procedimento swap usado em `popMax`
- No nível ℓ, mover um nó para baixo custa $O(\log n - \ell)$. Denotaremos $k = \log n$
- Existem no máximo 2^ℓ nós no nível ℓ.

$$O(\sum_{0 \leq \ell < k} 2^\ell (k - \ell)) = O(2^k \sum_{0 \leq \ell < k} \frac{k - \ell}{2^{k-\ell}}) = O(n \sum_{j \geq 1} \frac{j}{2^j}) = O(n)$$

Implementação

- Uma fila de prioridade é implementada como uma heap
- Mas não dissemos como a heap deve ser implementada
- Ela *se comporta* como uma árvore
- Vamos usar um array (muito mais eficiente na prática)

Árvores binárias em arrays



Nó	5	4	2	1	3
Índice	0	1	2	3	4
		i		$2i+1$	$2i+2$

- Heap Q de n elementos armazenados em um array q de tamanho n
- $q_0 = r(Q)$
- **Subnós**

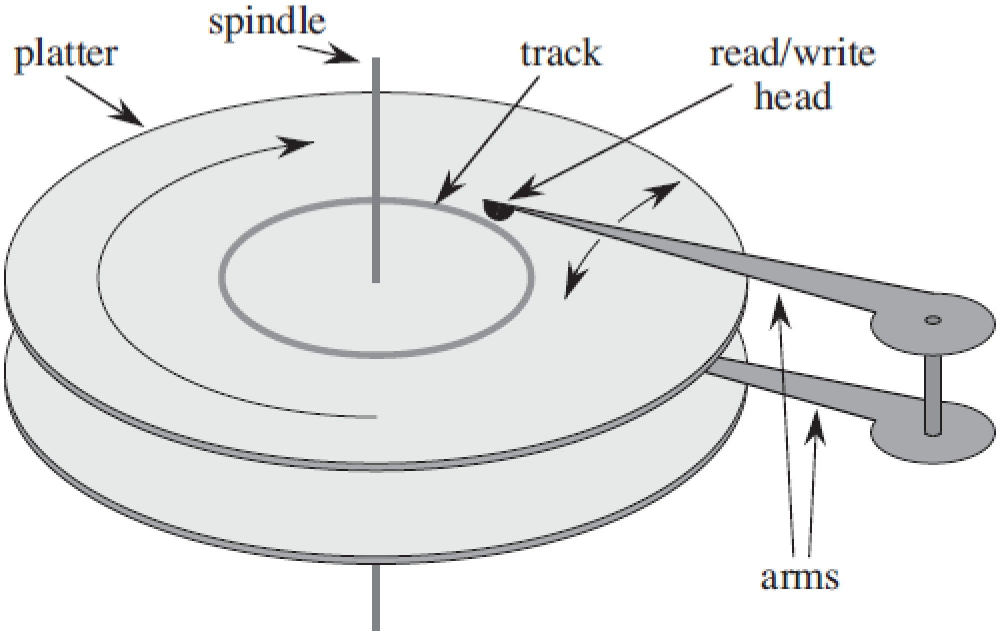
  Se $q_i = v$, então $q_{2i+1} = r(\mathsf{L}(v))$ e $q_{2i+2} = r(\mathsf{R}(v))$ (sempre que $2i+1, 2i+2 < n$)
- **Pai**

  Se $q_i = v \neq r(Q)$, $q_j = P(v)$ onde $j = \lfloor \frac{i-1}{2} \rfloor$

Temos agora todos os elementos para a implementação!

Árvores B

- Árvores B são árvores balanceadas de busca => altura $O(\log n)$
- Originalmente desenvolvidas para trabalhar com memória secundária



Árvores B

- Memória secundária é:
  - mais barata do que memória primária
  - tem maior capacidade
  - porém seu acesso é mais lento (braços mecânicos)

- As árvores B tentam minimizar o número de acessos à memória secundária obtendo o máximo de informação possível em cada acesso

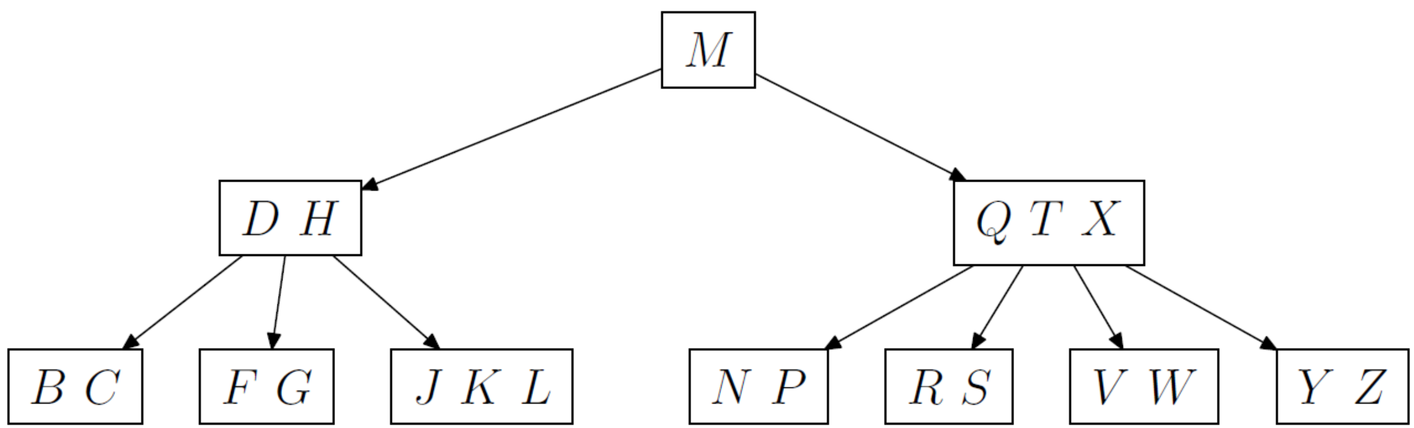
Definição

- Propriedades:
  - Todo nó *x* tem quatro campos
    1. O número de chaves atualmente armazenado no nó *x*, *n[x]*
    2. As *n[x]* chaves em si, armazenadas em ordem crescente
       $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$
    3. *n[x]* ponteiros para os dados na memória secundária
    4. *n[x]+1* ponteiros, $c_1[x]$, $c_2[x]$, ..., $c_{n[x]+1}[x]$ para seus filhos
  - O nó é também chamado página

Definição

- Propriedades:
  - As chaves *key*$_i$ [x] separam os intervalos de chaves armazenados em cada subárvore

$$k_1 \leq \mathrm{key}_1[x] \leq k_2 \leq \mathrm{key}_2[x] \leq \ldots \leq \mathrm{key}_{n[x]} \leq k_{n[x]+1}$$



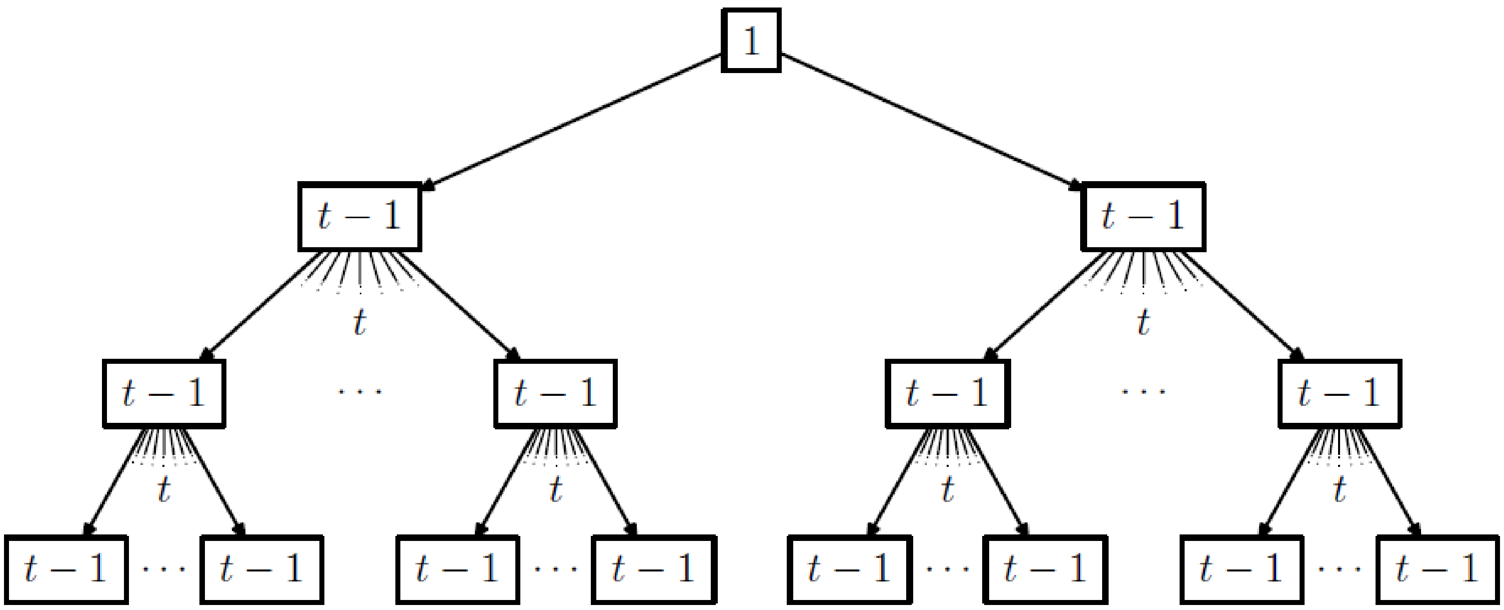
  - Todas as folhas estão na mesma profundidade

Definição

- Propriedades:
  - Todo nó com *n[x]* chaves tem *n[x]*+1 filhos
  - Existem limites inferiores e superiores para o número de chaves em um nó com base em um parâmetro t = grau da árvore B
    - Todo nó, com exceção da raíz, tem que ter no mínimo $t - 1$ chaves
    - Todo nó tem que ter no máximo $2t - 1$ chaves

Exemplo do pior caso

- Uma árvore-B de profundidade 3 contendo o mínimo possível de nós



prof.	# nós
0	1
1	2
2	$2t$
3	$2t^2$

Exemplo do pior caso

- O número *n* de chaves satisfaz portanto:

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$
$$= 2t^h - 1$$

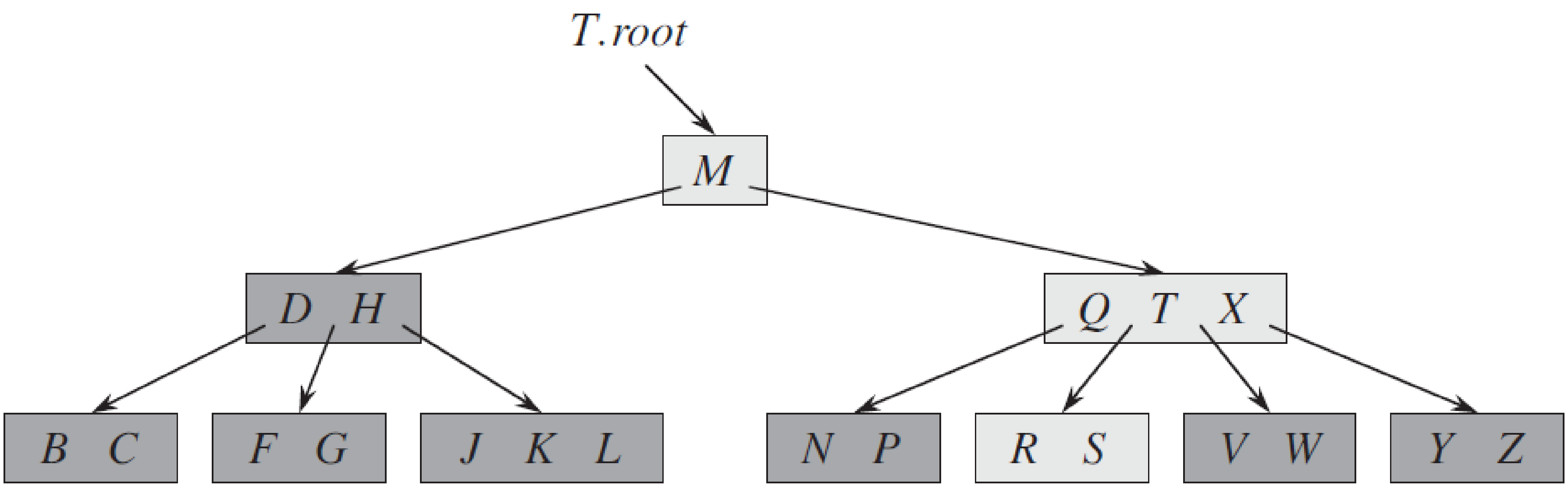
- Logo a pior profundidade possível para uma árvore B é $O(\log_t n)$
- O número de acessos ao disco é proporcional à profundidade
- É por esta razão que os nós das árvores B possuem mais de uma chave

Busca

- Semelhante à busca em uma BST
- Acrescenta-se testes relativos às chaves existentes em cada nó
- Pesquisa sequencial ou binária dentro do nó

Busca

- Exemplo: busca pela chave *R*

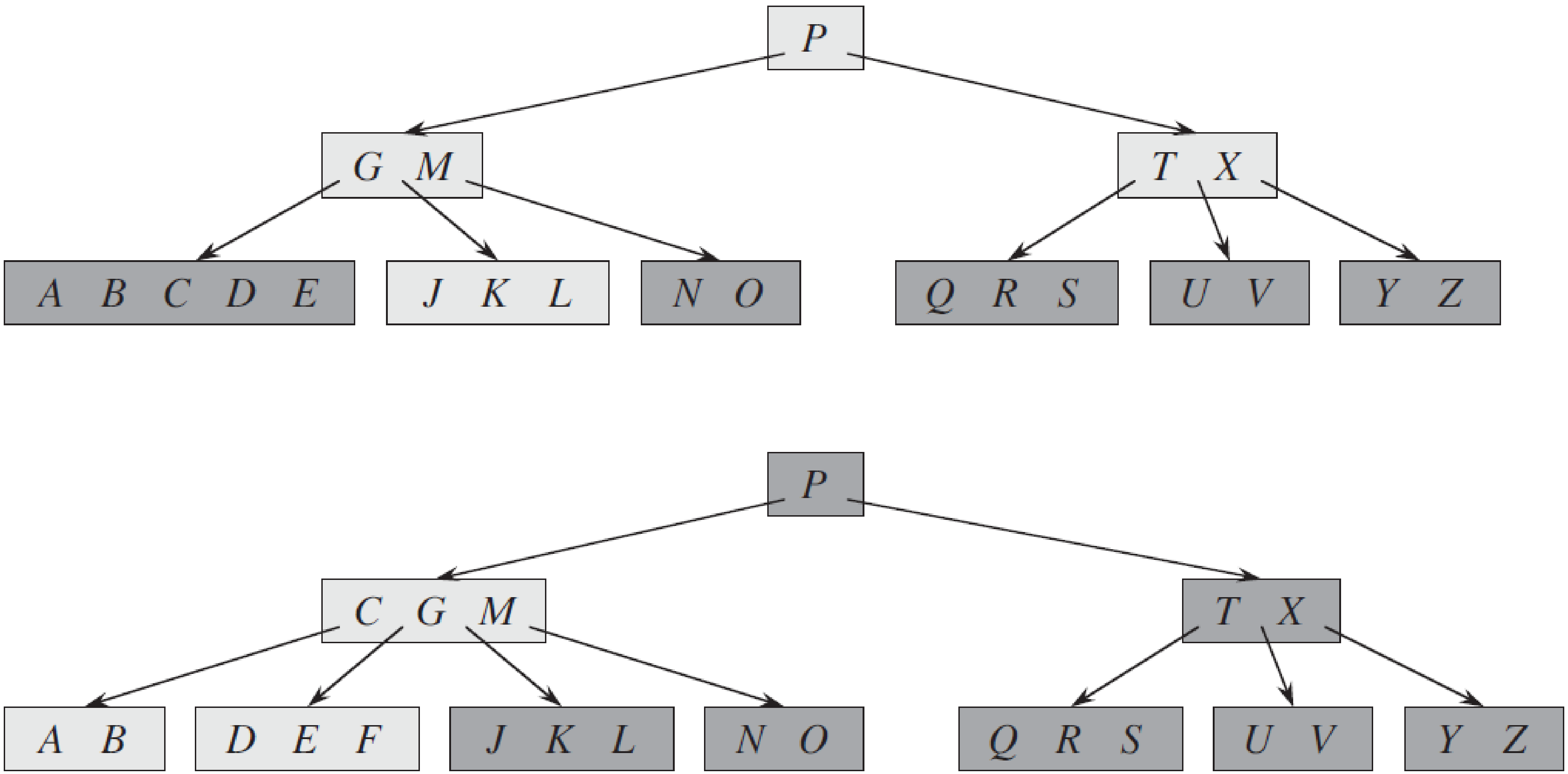


Inserção

- Primeiro é feita uma busca a fim de encontrar a folha de inserção
- Caso exista espaço na folha, basta adicionar a chave de maneira a preservar a ordenação
- Caso não exista espaço na folha, i.e., a folha já tenha $2t$-1 chaves:
  - Dividimos a folha
  - A chave correspondente à mediana vai para o nó pai
  - Se não houver espaço no nó pai, o processo é repetido.

Inserção

- Exemplo: inserção de *F* (t = 3)



Inserção

- No pior caso, o processo de divisão propaga-se até a raiz da árvore B.
- Neste caso, a árvore aumenta sua profundidade de um nível
- Uma árvore B somente aumenta sua profundidade com a divisão da raiz

Remoção

- Diferentes casos

- Simular em

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