INF421, Lecture 5
Balanced Trees

Leo Liberti

LIX, École Polytechnique, France

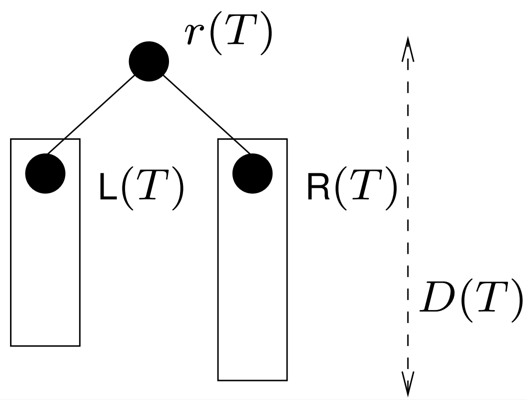
Course

- **Objective**: teach notions AND develop intelligence
- **Evaluation**: TP noté en salle info, Contrôle à la fin. Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization**: fri 31/8, 7/9, 14/9, 21/9, 28/9, 5/10, 12/10, 19/10, 26/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI:30-34)
- **Books**:
  1. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
  2. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
  3. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
  4. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2006
- **Website**: `www.enseignement.polytechnique.fr/informatique/INF421`
- **Blog**: `inf421.wordpress.com`
- **Contact**: `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

Lecture summary

- Binary search trees
- AVL trees
- Heaps and priority queues
- Tries

Notation

Tree T	node v	root node $r(T)$
L(T): left subtree of $r(T)$	R(T): right subtree of $r(T)$	depth $D(T)$
L(v): left subnode of v	R(v): right subnode of v	
L(T) = R(T) = $\varnothing$: leaf	$T = \langle \mathsf{L}(T), r(T), \mathsf{R}(T) \rangle$	$P(v)$: parent of v
$p(v)$: unique path $r(T) \to v$	path length: $\sum_v \|p(v)\|$	$D(T) = \max_v \|p(v)\|$

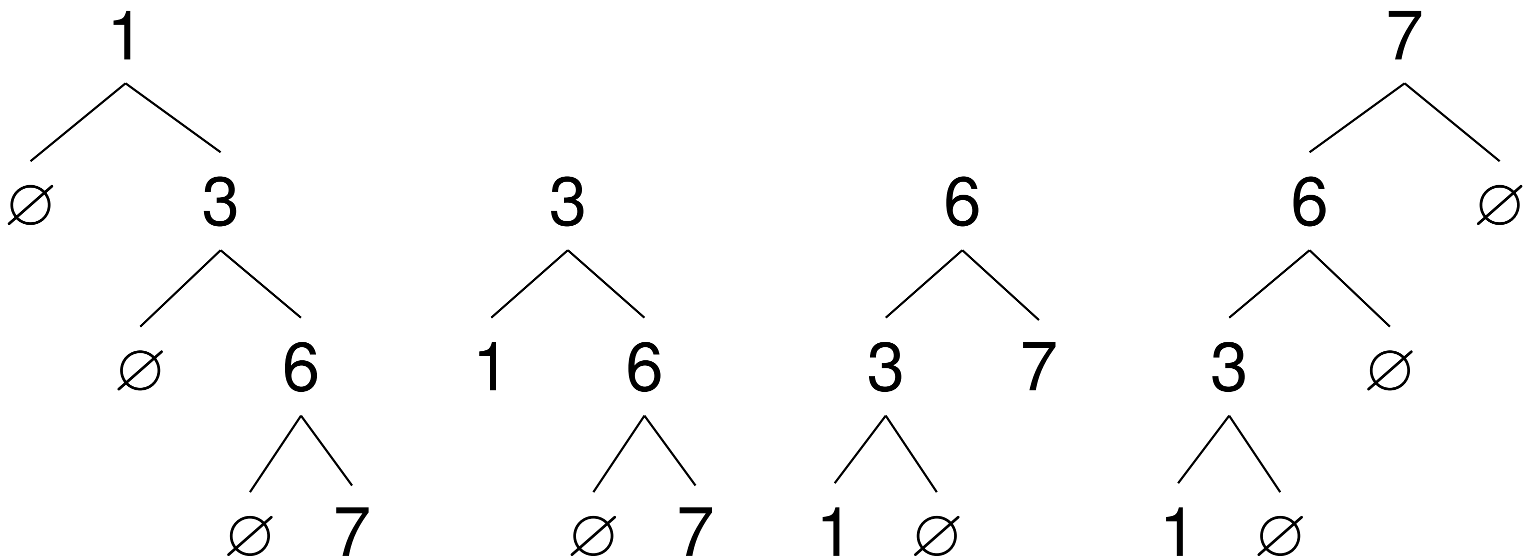
Binary search trees (BST)

Sorted sequences

- Store a set V as a **sorted sequence**
- Answer the question $v \in V$ efficiently
- Invariant :

$$\boxed{\mathsf{L}(v) < v < \mathsf{R}(v)} \qquad (*)$$

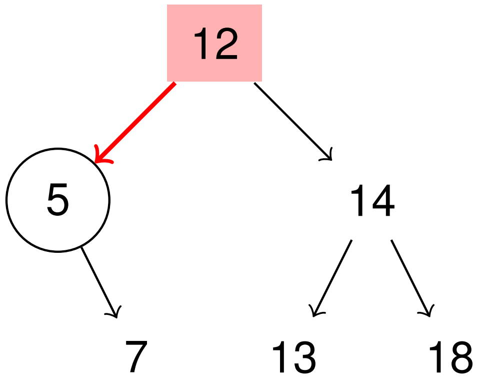
- Example: $V = \{1, 3, 6, 7\}$



BST `min/max`

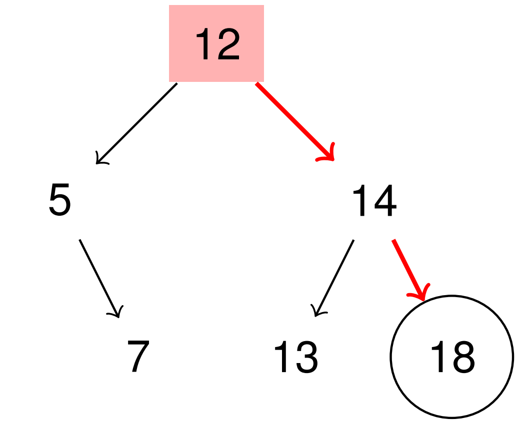
- $\mathtt{min}(v)$:

  1: **if** $\mathsf{L}(v) = \varnothing$ **then**
  2: **return** v;
  3: **else**
  4: **return** $\min(\mathsf{L}(v))$;
  5: **end if**



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

  1: **if** $\mathsf{R}(v) = \varnothing$ **then**
  2: **return** v;
  3: **else**
  4: **return** $\max(\mathsf{R}(v))$;
  5: **end if**



Base cases for recursion

All other BST functions $\mathtt{f}(k, v)$:

$\mathtt{f}(k, \varnothing)$ returns without doing anything

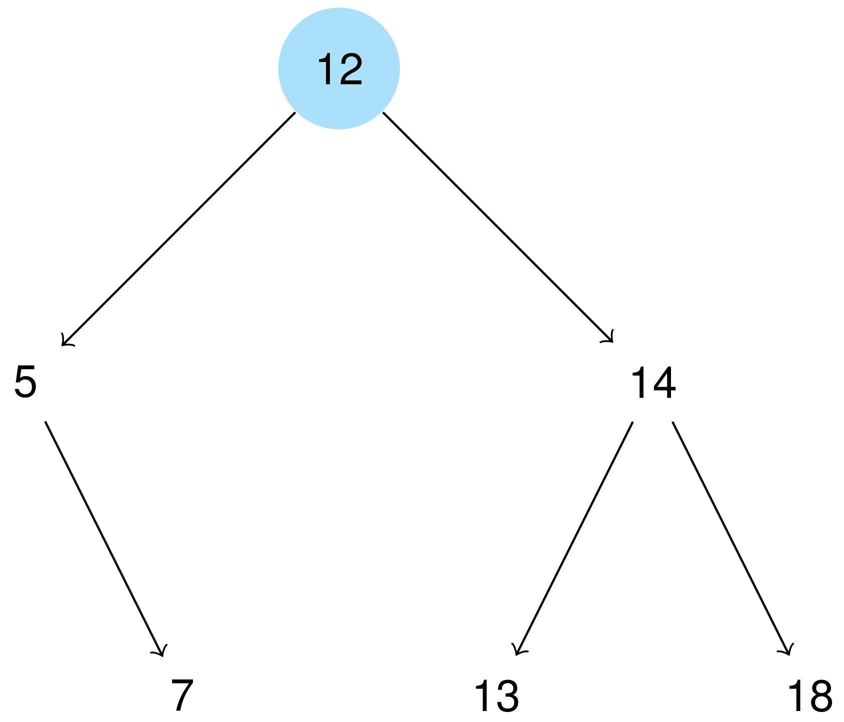
BST `find`

- `find`(k,v):

  `ret` = **not_found**;
  if $v = k$ ($\Rightarrow$"v stores 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

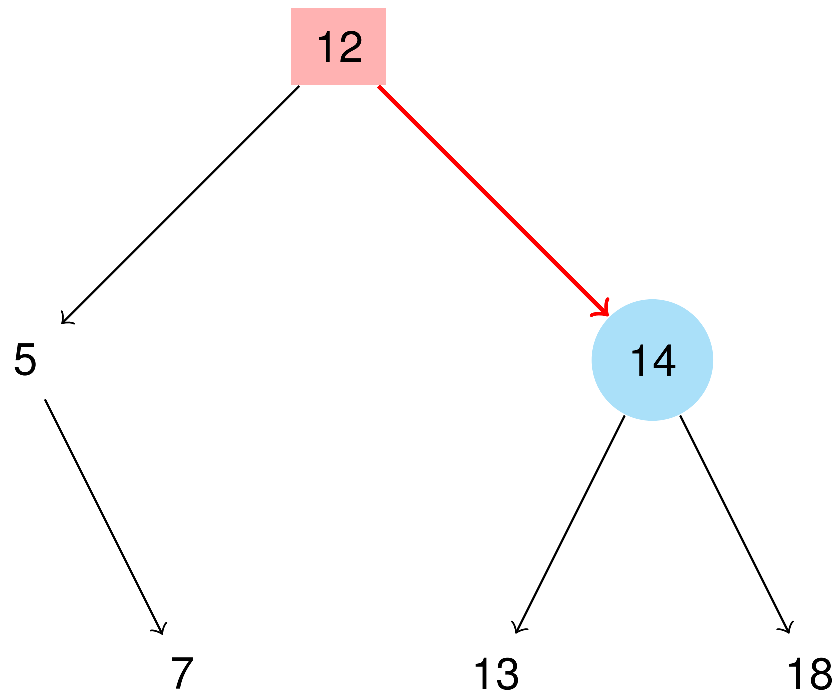
$$\texttt{find}(13, r(T))$$



$13 > 12$, take right branch

Successful find

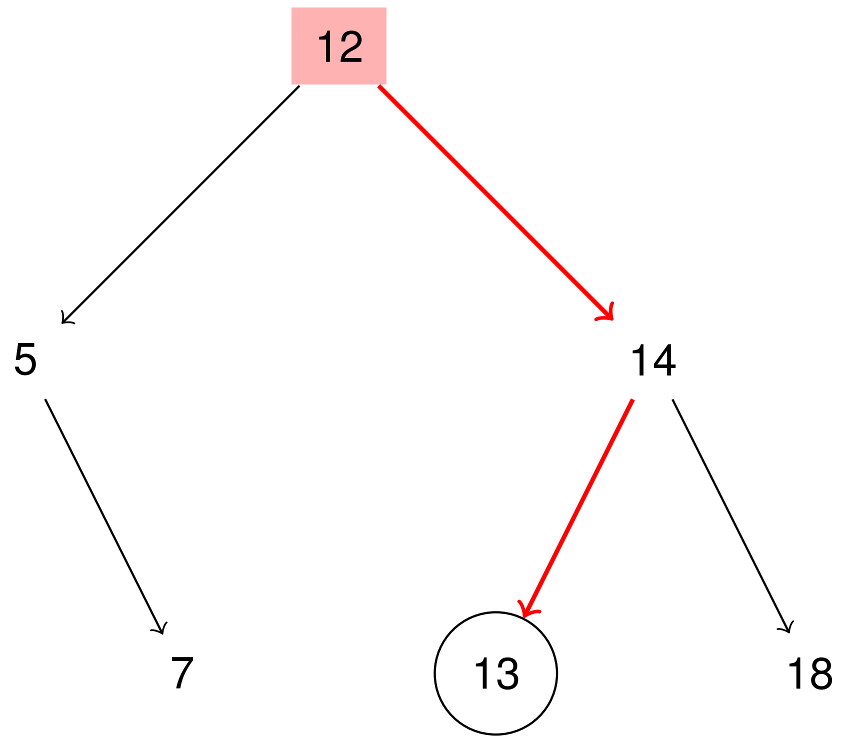
$$\texttt{find}(13, r(T))$$



$13 < 14$, take left branch

Successful find

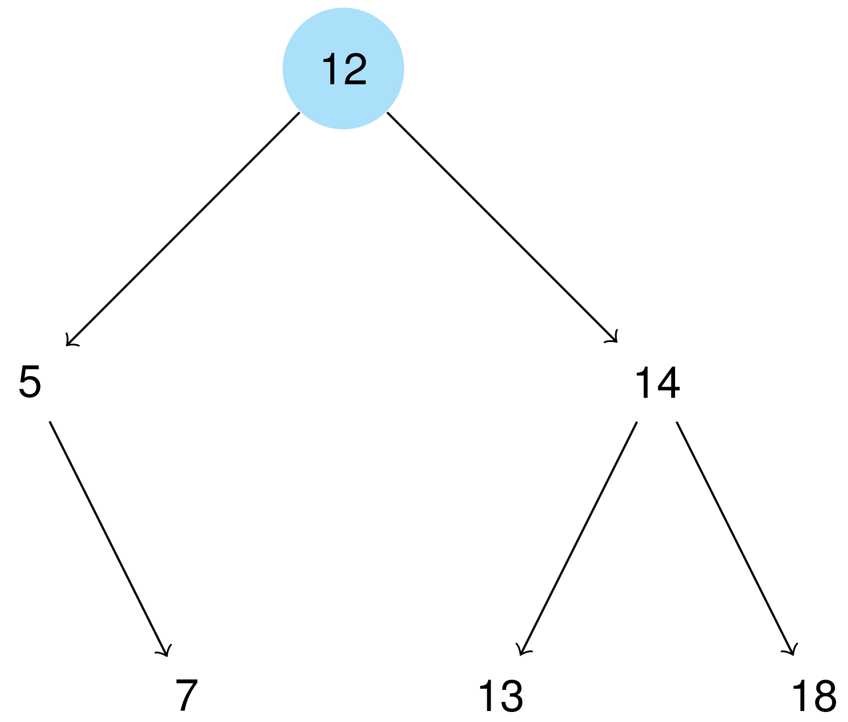
$$\texttt{find}(13, r(T))$$



found 13

Unsuccessful find

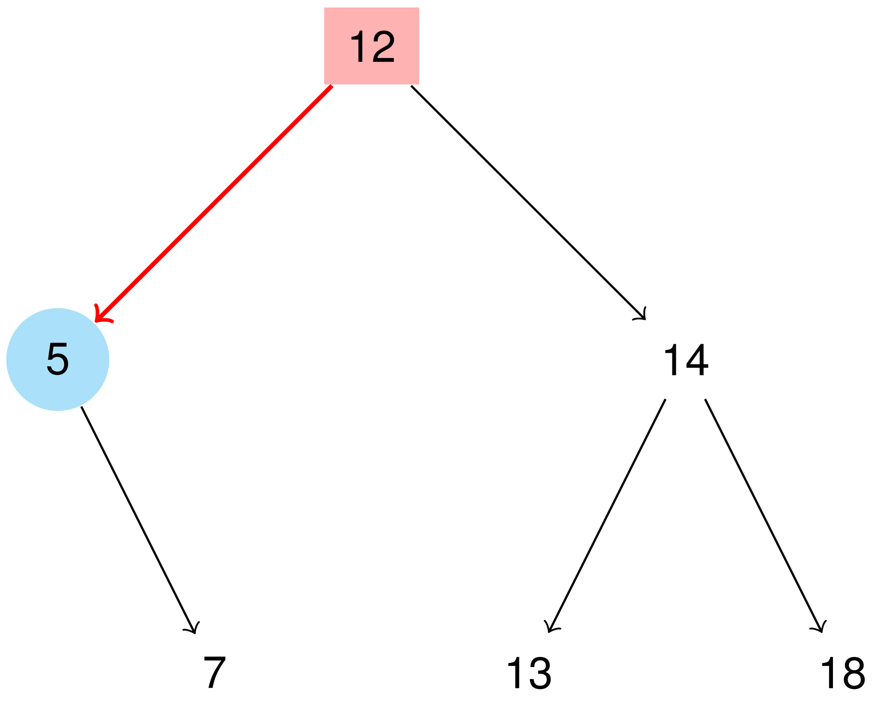
$\texttt{find}(1, r(T))$



$1 < 12$, take left branch

Unsuccessful find

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



$1 < 5$, should take left branch but $\mathsf{L}(5) = \varnothing$, not found

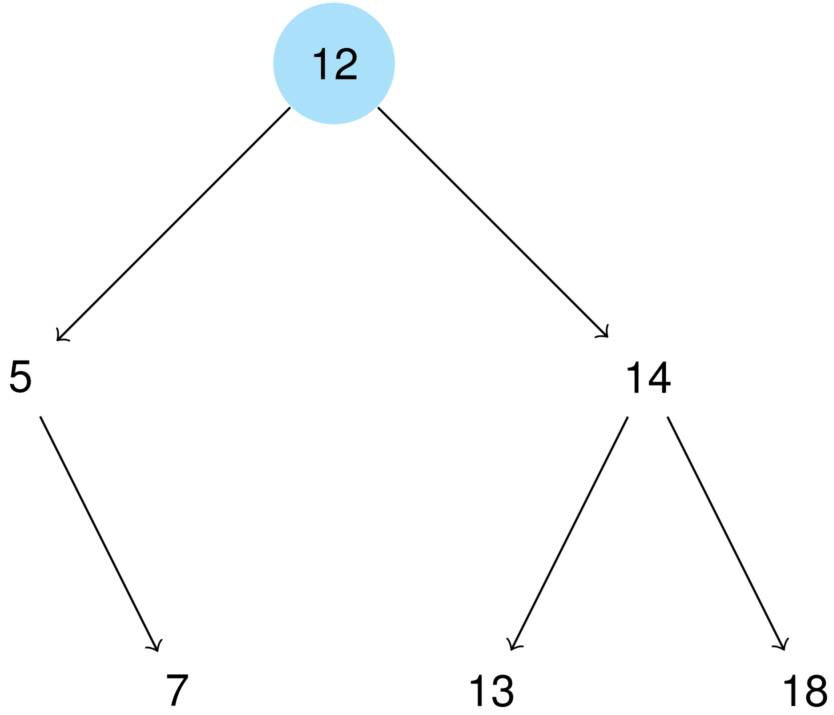
BST `insert`

- `insert`(k, v):

```
if k = v then
  return already_in_set;
else if k < v then
  if L(v) = ∅ then
    L(v) = k;          // store k in L(v)
  else
    insert(k, v);
  end if
else
  if R(v) = ∅ then
    R(v) = k;          // store k in R(v)
  else
    insert(k, R(v));
  end if
end if
```

Insert example

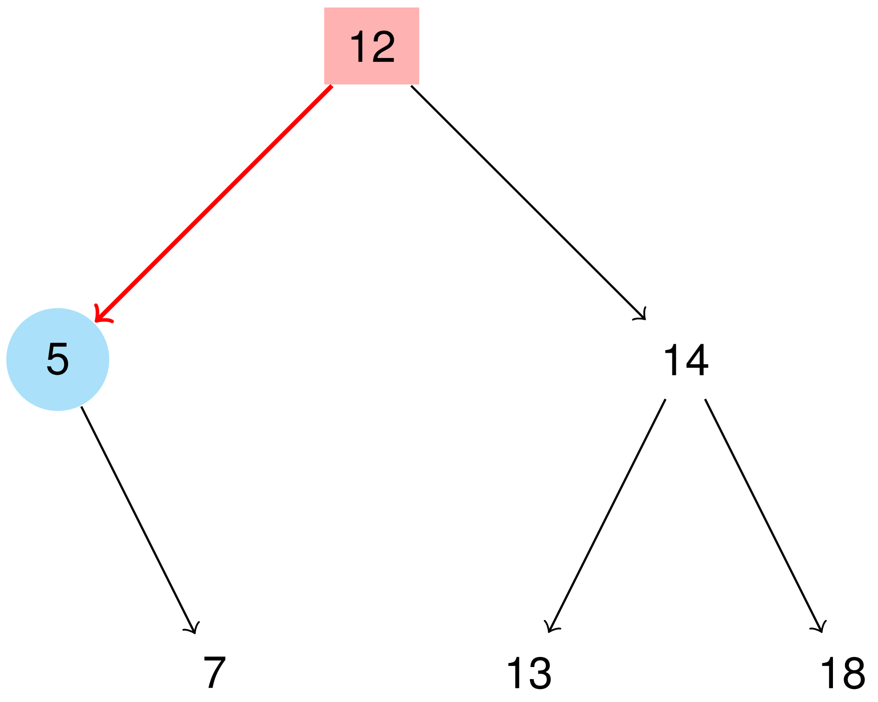
$\texttt{insert}(1, r(T))$



$1 < 12$, take left branch

Insert example

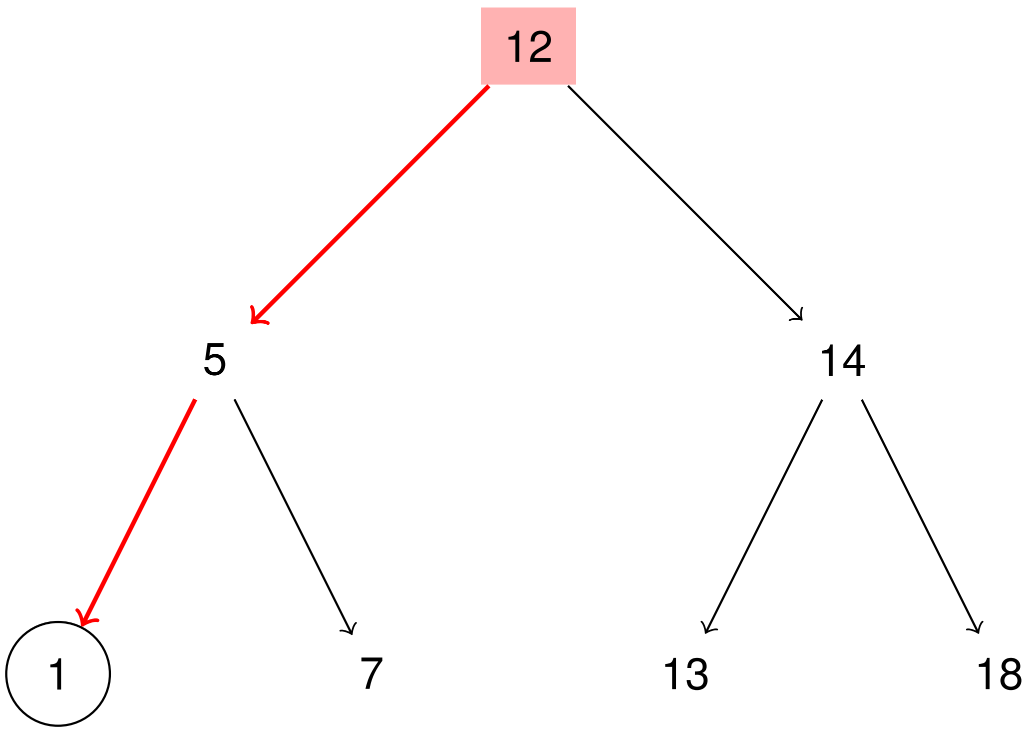
$\texttt{insert}(1, r(T))$



$1 < 5$, should take left branch but $\mathsf{L}(5) = \varnothing$

Insert example

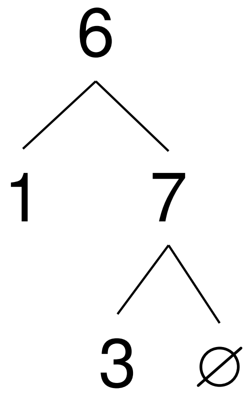
$\texttt{insert}(1, r(T))$



Add $k = 1$ as $\mathsf{L}(5)$

A global invariant

- $\mathsf{L}(v) \leq v \leq \mathsf{R}(v)$ only involves direct subnodes of v
- $\Rightarrow$ it is <u>local</u>
- Is this tree possible?

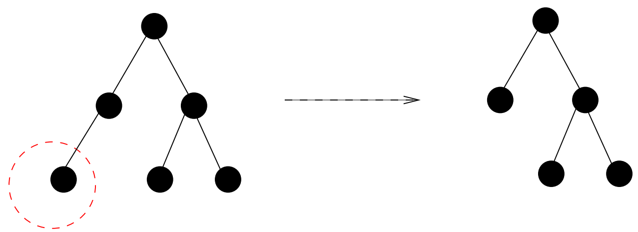


- It satisfies invariant
- By `insert`, 3 would be stored in $\mathsf{R}(1)$
- $\Rightarrow$ Invariant is global:

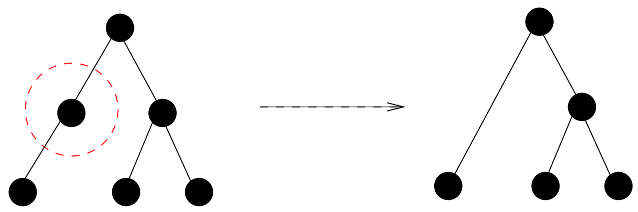
$$\forall u \in \mathsf{tree}(\mathsf{L}(v)), w \in \mathsf{tree}(\mathsf{R}(v)) \quad u < v < w$$

Deletion

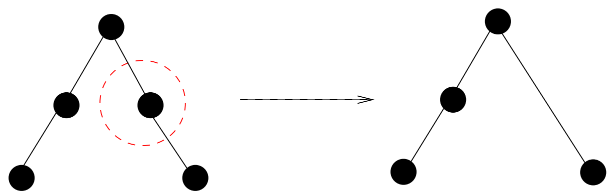
- If node v to delete is a leaf, easy: "cut" it (`unlink`)



- If $\mathsf{R}(v) = \varnothing$ and $\mathsf{L}(v) \neq \varnothing$, replace with $\mathsf{L}(v)$



- If $\mathsf{L}(v) = \varnothing$ and $\mathsf{R}(v) \neq \varnothing$, replace with $\mathsf{R}(v)$



- If v has both subtrees, nontrivial

Replacing a node

Replace link $\{P(v), v\}$ with $\{P(v), u\}$, then unlink v

- `replace`(v, u) `// replace` v `with` u

  if $\mathsf{R}(P(v)) = v$ (i.e. u is a right subnode) **then**
      $\mathsf{R}(P(v)) \leftarrow u$;
  else
      $\mathsf{L}(P(v)) \leftarrow u$;
  end if
  if $u \neq \varnothing$ **then**
      $P(u) \leftarrow P(v)$;
  end if
  unlink v;

- `unlink`: set $\mathsf{L}(v) = \mathsf{R}(v) = P(v) = \varnothing$

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

Idea: swap v with $u = \min \mathsf{R}(v)$ then delete it

Thm.

Invariant $\mathsf{L}(v) \leq v < \mathsf{R}(v)$ holds after swap

- Min of a BST: **leftmost node without left subtree**
- $\Rightarrow$ Can delete u (case $\mathsf{L}(\cdot) = \varnothing$ above)
- After swap (u, v), $v = \min(\mathsf{R}(v))$; hence $v < \mathsf{R}(v)$
- Before swap $u \in \mathsf{tree}(\mathsf{R}(v)) \Rightarrow$ after swap $v > \mathsf{L}(v)$
- $\Rightarrow$ Thm. holds

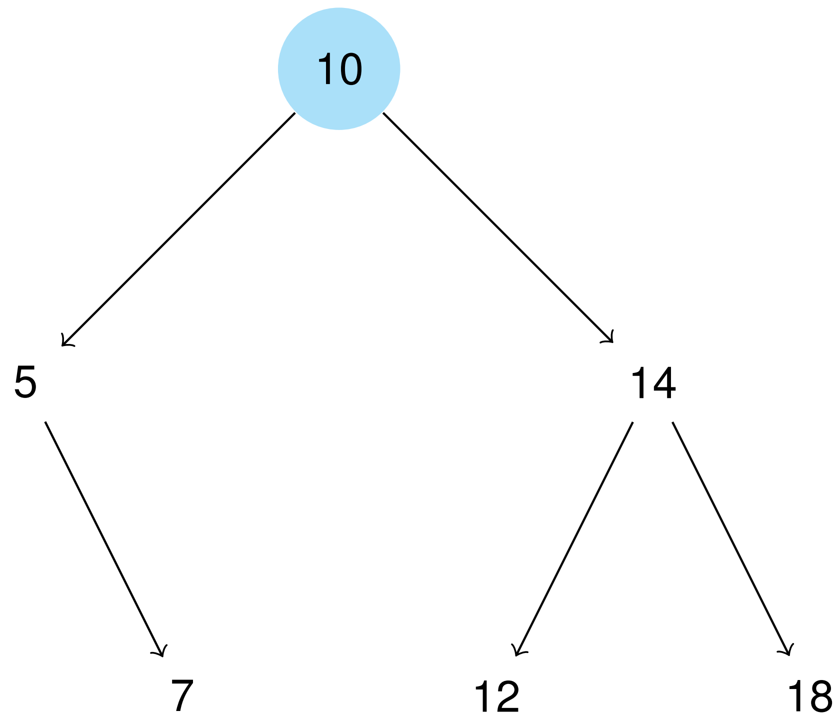
BST `delete`

- `delete`(k, v):

if $k < v$ **then**
    `delete`$(k, \mathsf{L}(v))$;
else if $k > v$ **then**
    `delete`$(k, \mathsf{R}(v))$;
else
    if $\mathsf{L}(v) = \varnothing \vee \mathsf{R}(v) = \varnothing$ **then**
        delete v; `// one of the easy cases`
    else
        $u = \mathtt{min}(\mathsf{R}(v))$;
        `swap_values`(u, v);
        delete u; `// easy case: L(u)=∅`
    end if
end if

Delete example

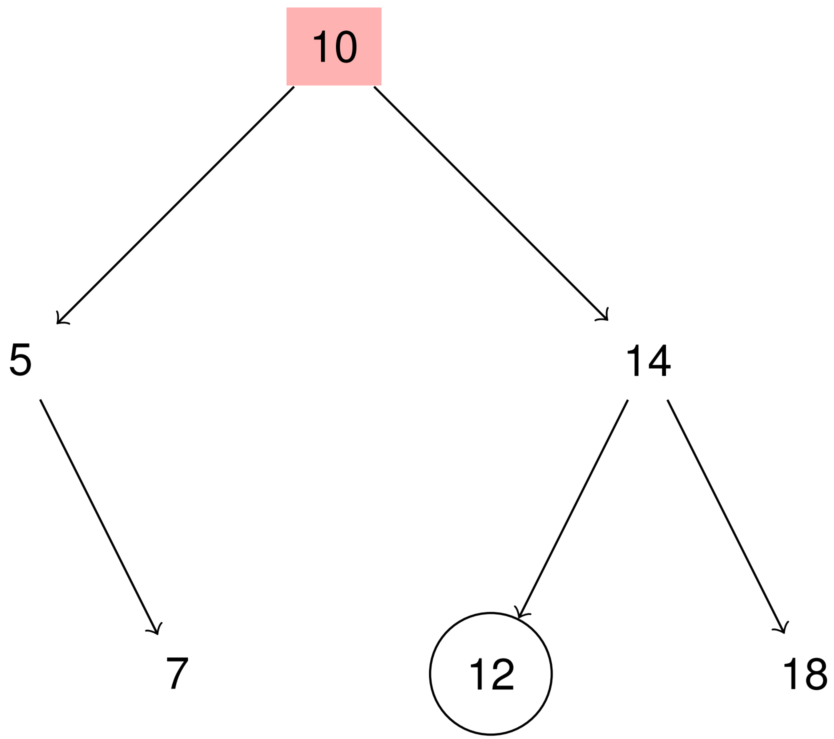
$\texttt{delete}(10, r(T))$



$$v = 10$$

Delete example

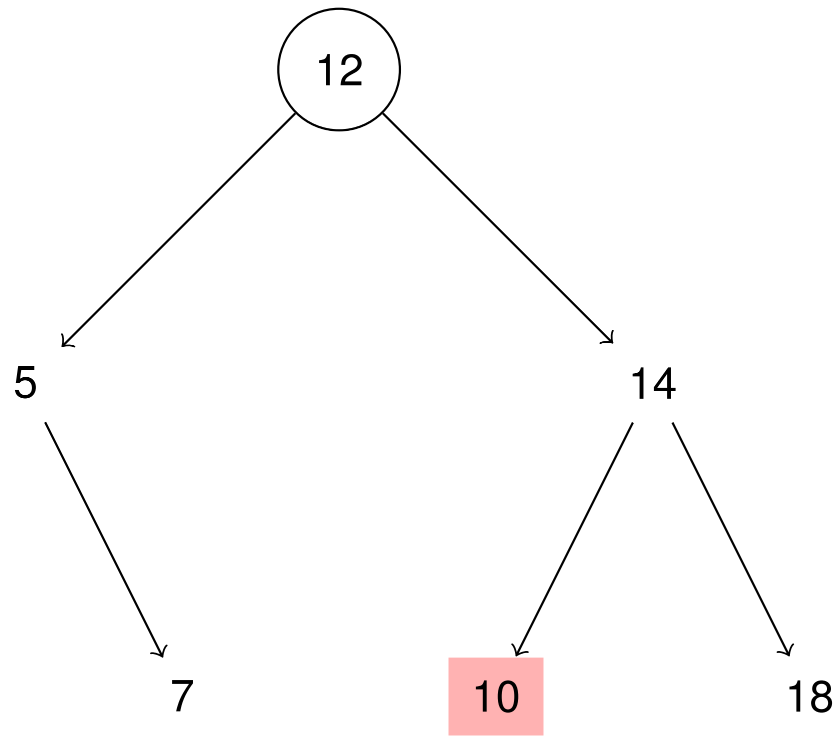
$\texttt{delete}(10, r(T))$



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

Delete example

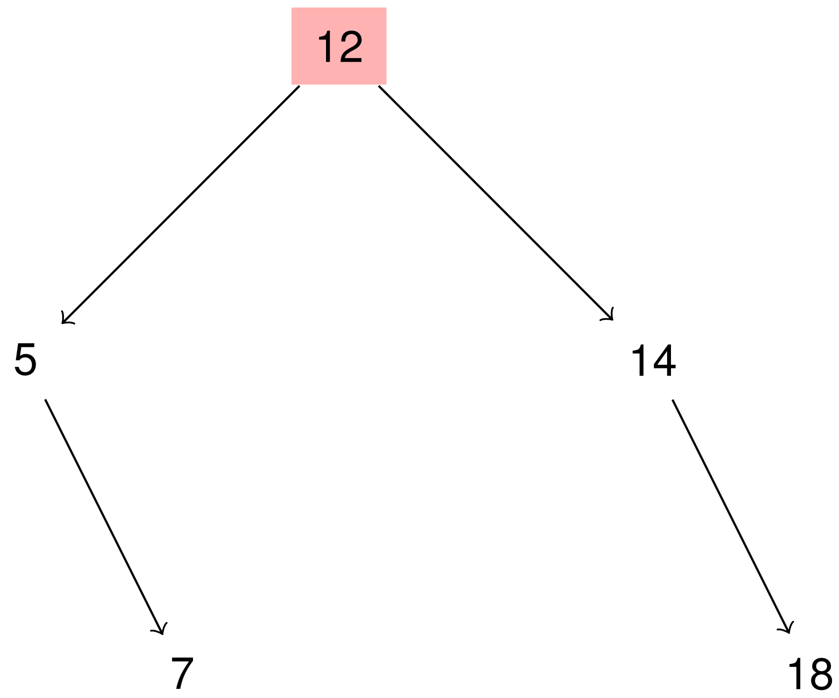
$\texttt{delete}(10, r(T))$



swap values of 10 and 12

Delete example

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



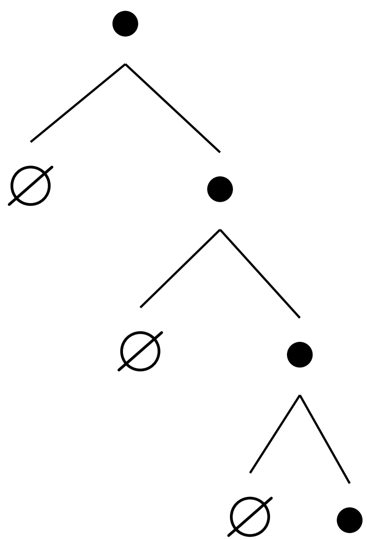
delete 10

Tree balance

- Balance: $B(T) = D(\mathsf{L}(T)) - D(\mathsf{R}(T))$
- Tree is **balanced**: $B(T) \in \{-1, 0, 1\}$
- In a balanced tree, $D(T)$ is $O(\log n)$
- Intuition : if a BST has $n = 2^k$ nodes at level k, then $k = \log n$
- Intuitively, balance $\approx$ all leaves have same depth
- **Not actually true**, but close enough
- If T is balanced, $D(T) < \log_\phi(n+2) - 1$ with ϕ golden ratio

Complexity

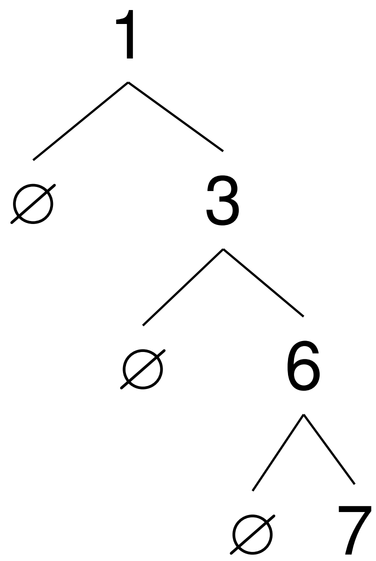
- Every call involves at most one recursion
- $\Rightarrow$ Recurse along <u>one path only</u>, no backtracking
- Worst-case complexity proportional to depth $D(T)$
- Tree balanced: $D(T)$ is $O(\log n)$
- Otherwise: $D(T)$ is $O(n)$



Adelson-Velskii & Landis (AVL) trees

AVL Trees

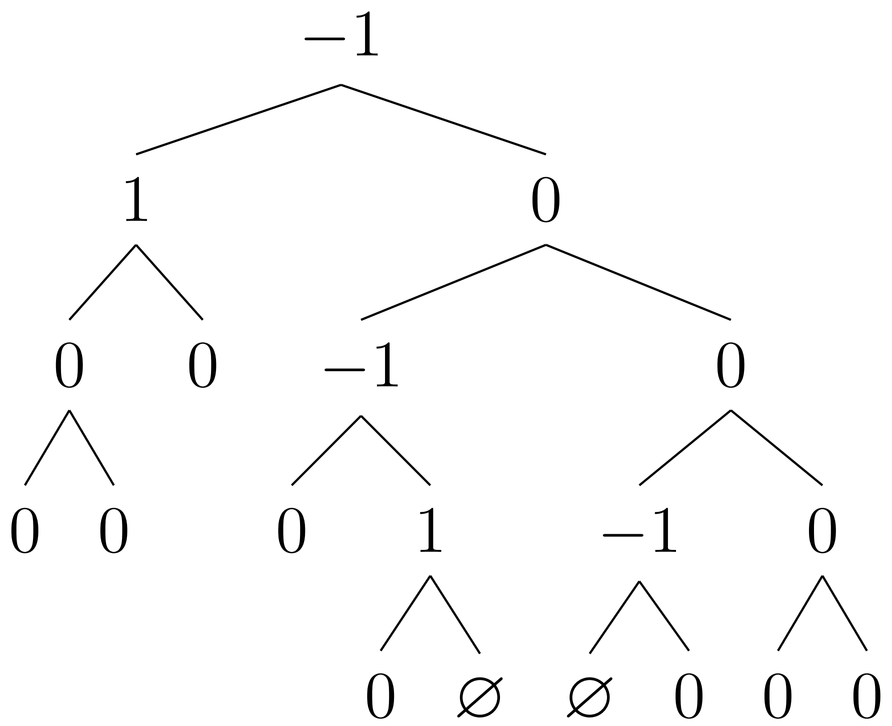
- Try inserting $1, 3, 6, 7$ in this order: get **unbalanced tree**



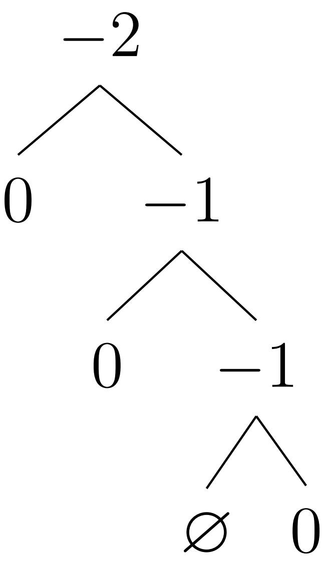
- Worst case find (i.e., find the key 7) is $O(n)$
- Need to *rebalance* the tree to be more efficient
- **AVL trees invariant**: $B(T) \in \{-1, 0, 1\}$

Examples

AVL tree:



Non-AVL tree:



Nodes indicate $B(\text{tree}(v))$

Insertion example

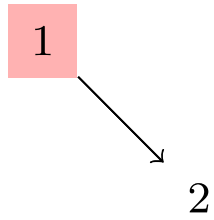
insert 1

$\boxed{1}$

$v_1 = 1;$
$r(T) = v_1;$

Insertion example

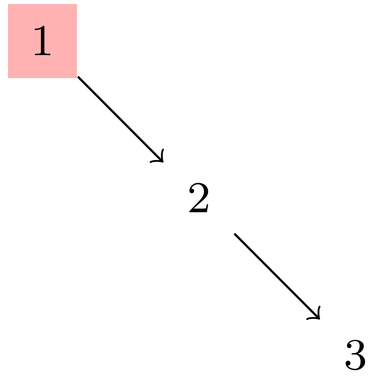
insert 2



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

Insertion example

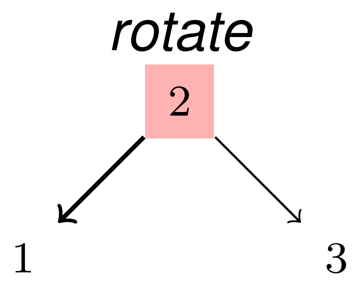
insert 3



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

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

Insertion example



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

Insertion example

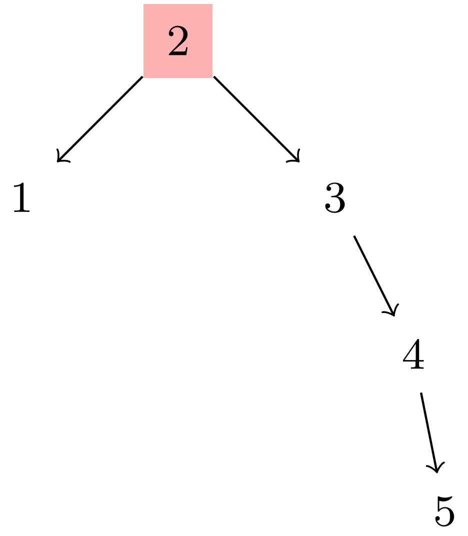
insert 4

$$\begin{tikzcd}[column sep=small] & \boxed{2} \arrow[dl] \arrow[dr] & & \\ 1 & & 3 \arrow[dr] & \\ & & & 4 \end{tikzcd}$$

$v_4 = 4$;
$\mathsf{R}(v_3) = v_4$;
$P(v_4) = v_3$;

Insertion example

insert 5

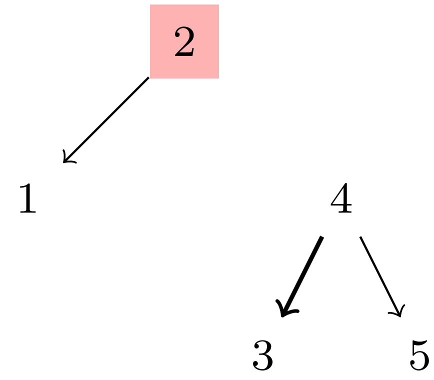


$v_5 = 5$;
$\mathsf{R}(v_4) = v_5$;
$P(v_5) = v_4$;

$H(\mathsf{L}(v_3)) = 0$,
$H(\mathsf{R}(v_3)) = 2$:
$B(T) = -2$: out of balance

Insertion example

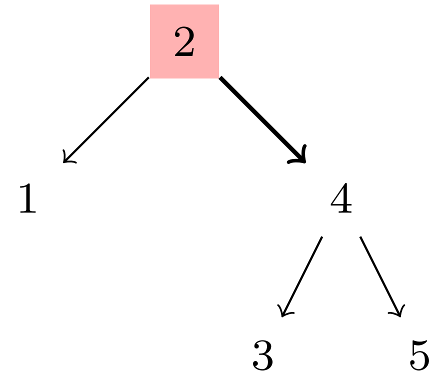
rotate 1/2



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

Insertion example

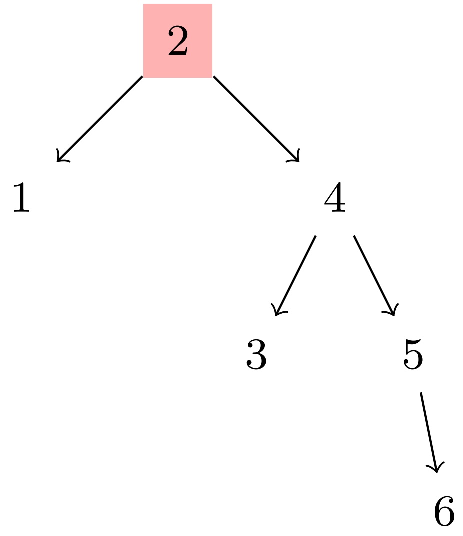
rotate 2/2



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

Insertion example

insert 6

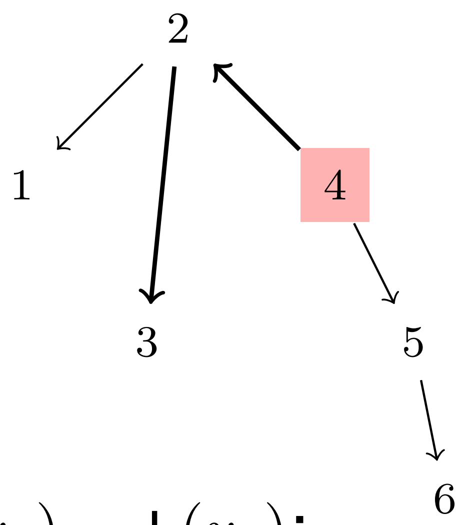


$v_6 = 6$;
$\mathsf{R}(v_5) = v_6$;
$P(v_6) = v_5$;

$H(\mathsf{L}(v_2)) = 1$,
$H(\mathsf{R}(v_2)) = 3$:
$B(T) = -2$: out of balance

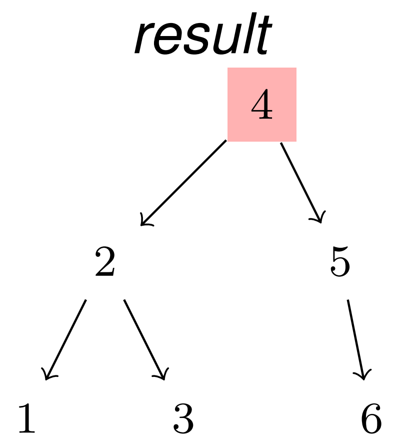
Insertion example

rotate



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

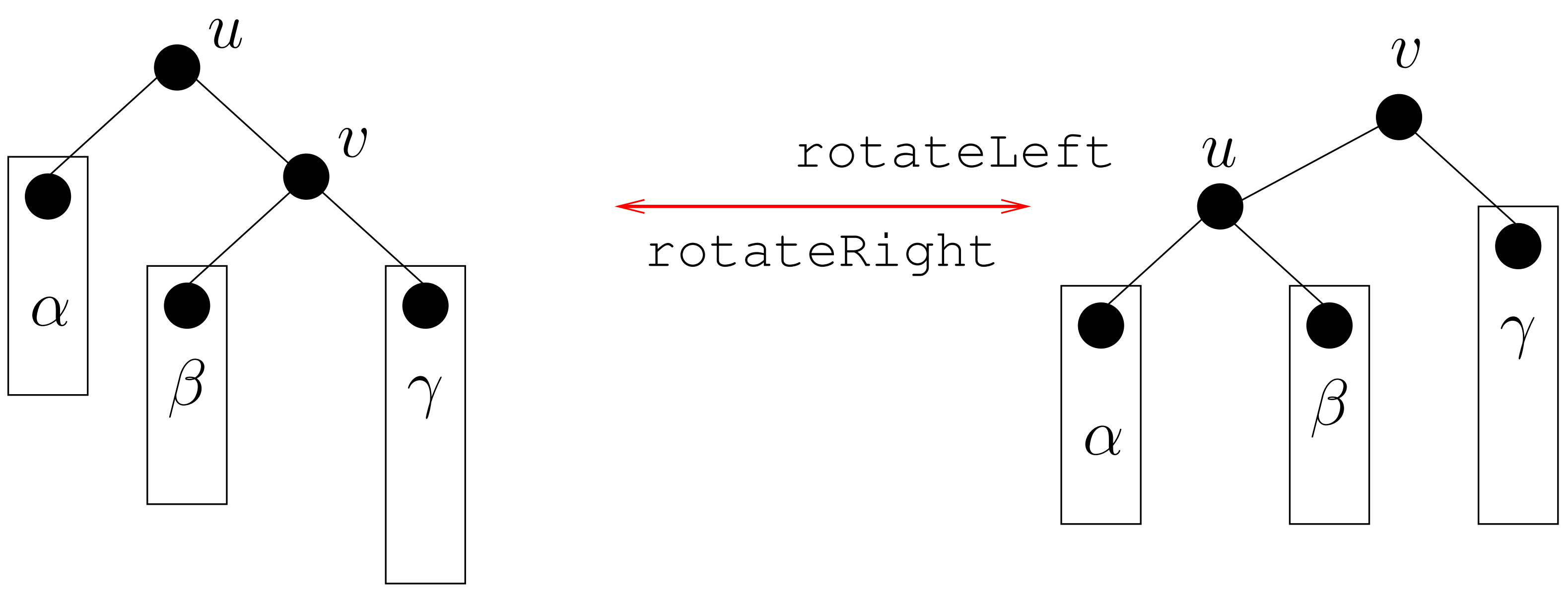
Insertion example



In general

- Decompose balanced trees operations into:
  - the operation itself
  - some *rebalancing operations* called **rotations**
- `min/max, find`: as in BSTs
- Unbalancing can occur on insertion and deletion
- Insert/delete one node at a time $\Rightarrow$ unbalance offset ≤ 1
- I.e., $B(T) \in \{-2, -1, 0, 1, 2\}$
- `insert, delete`: as in BST with rotations

Left and right rotation



Algebraic interpretation

- Let α, β, γ be trees, u, v be nodes not in α, β, γ
- Define:
  - $\texttt{rotateLeft}(\langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle) = \langle \langle \alpha, u, \beta \rangle, v, \gamma \rangle$
  - $\texttt{rotateRight}(\langle \langle \alpha, u, \beta \rangle, v, \gamma \rangle) = \langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle$
- A sort of "associativity of trees"
- Remark: $\texttt{rotateLeft}, \texttt{rotateRight}$ are inverses

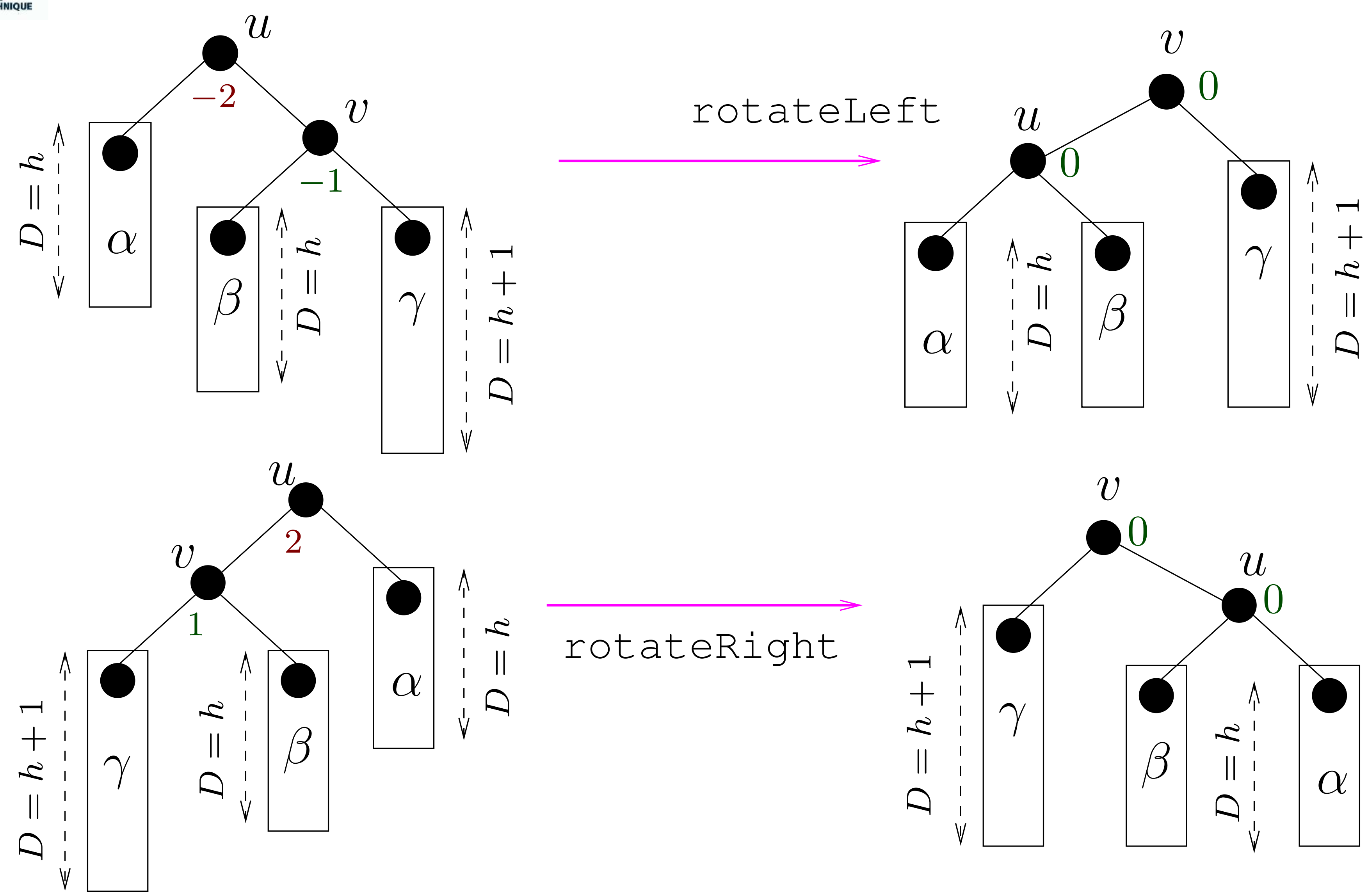
Thm.

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

Proof

Directly from the definition

Rotating and rebalancing



Properties of rotation

Thm.

$\forall T$, $\texttt{rotateLeft}(T)$, $\texttt{rotateRight}(T')$ are BSTs

Proof

(Sketch): The tree order only changes locally for u, v. In T, $\mathsf{tree}(v) = \mathsf{R}(u) \Rightarrow u < v$. In $\texttt{rotateLeft}(T)$, $\mathsf{tree}(u) = \mathsf{L}(v)$, which is consistent with $u < v$. Similarly for T'.

- Suppose $D(\alpha) = D(\beta) = h$ and $D(\gamma) = h + 1$
- Let $T = \langle \alpha, u, \langle \beta, v, \gamma \rangle \rangle$: then $B(T) = -2$
- Let $T' = \langle \langle \gamma, u, \beta \rangle, v, \alpha \rangle$: then $B(T') = 2$

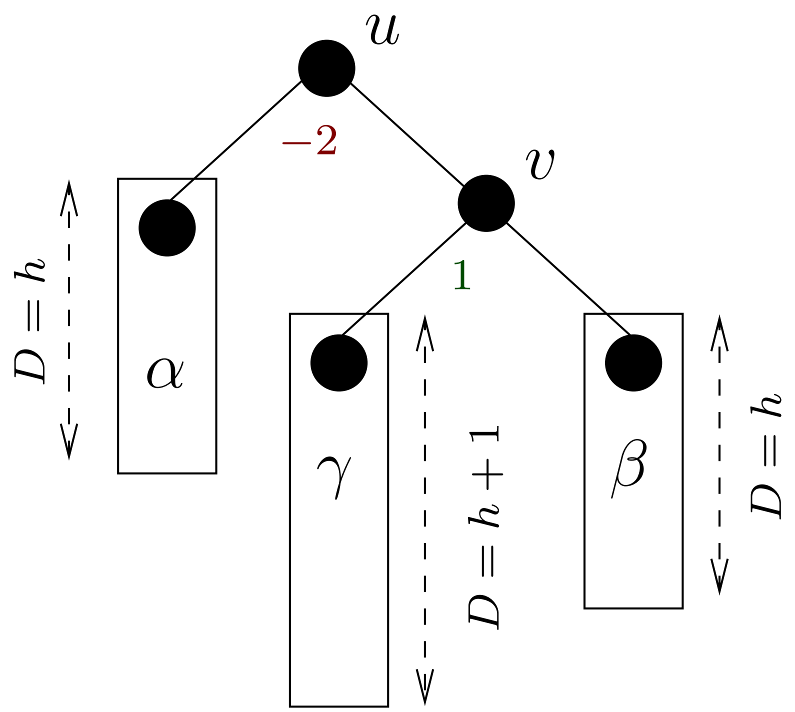
Thm.

T, T' as above $\Rightarrow B(\texttt{rotateLeft}(T)) = 0$, $B(\texttt{rotateRight}(T')) = 0$

Proof

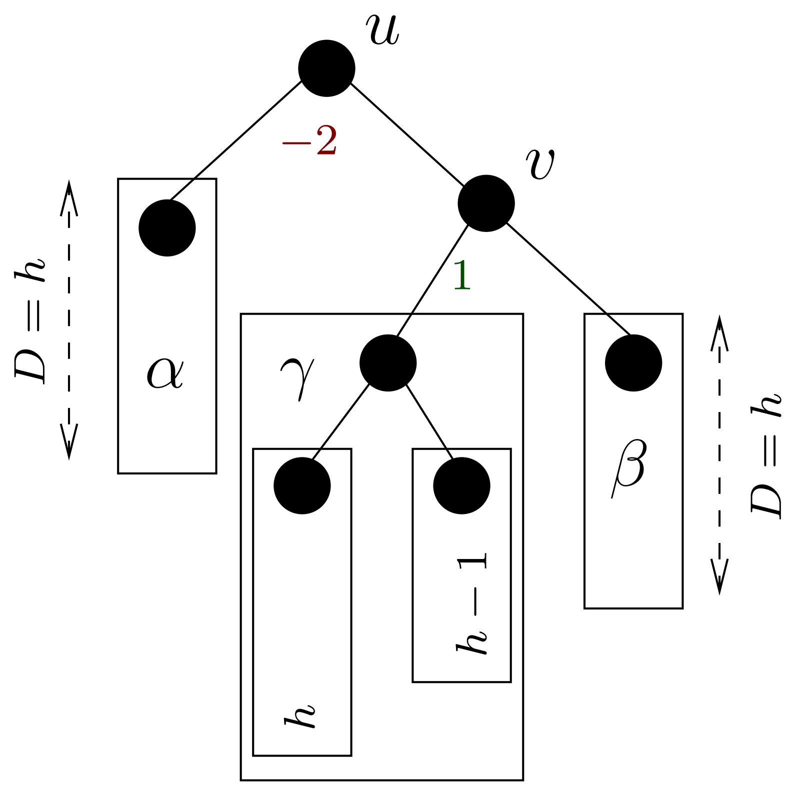
(Sketch): since subtrees α, γ are swapped, tree depth is $D = h$ for all subtrees

Is this enough?



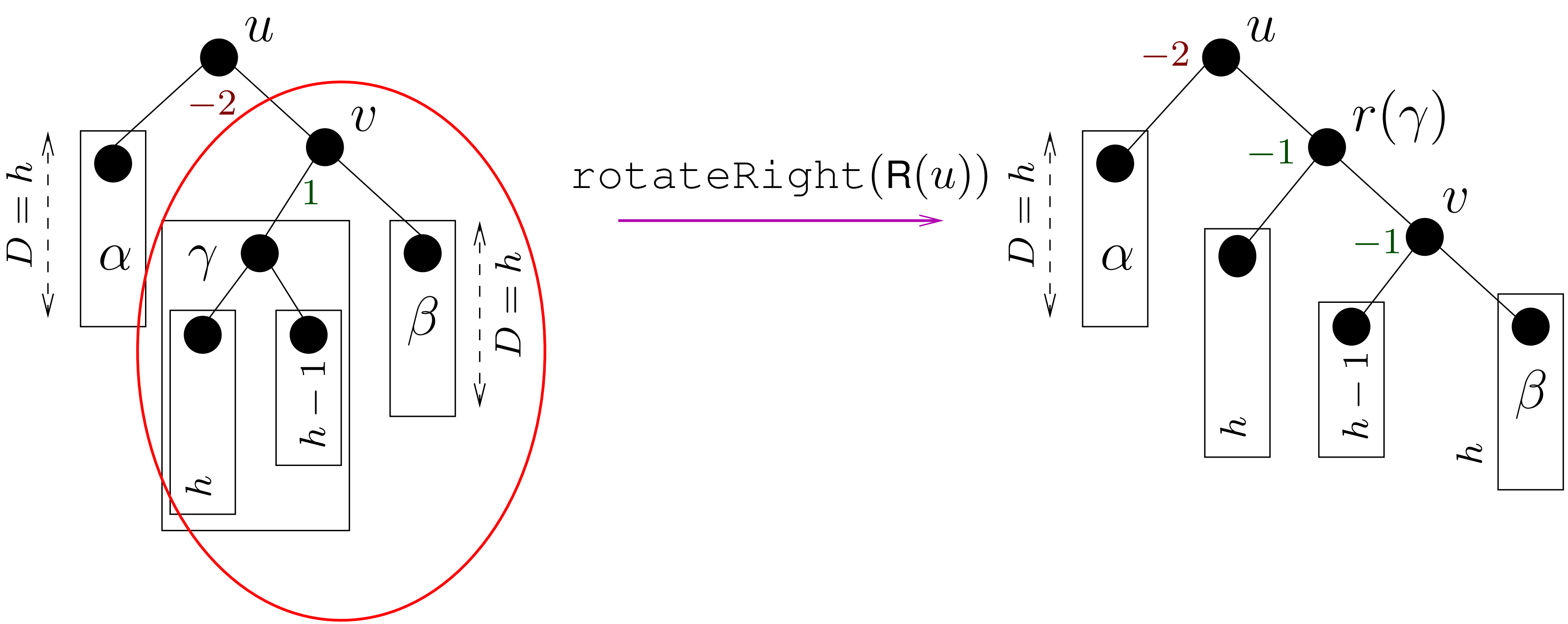
Rotating leaves γ at its place, doesn't work

Break γ up into subtrees



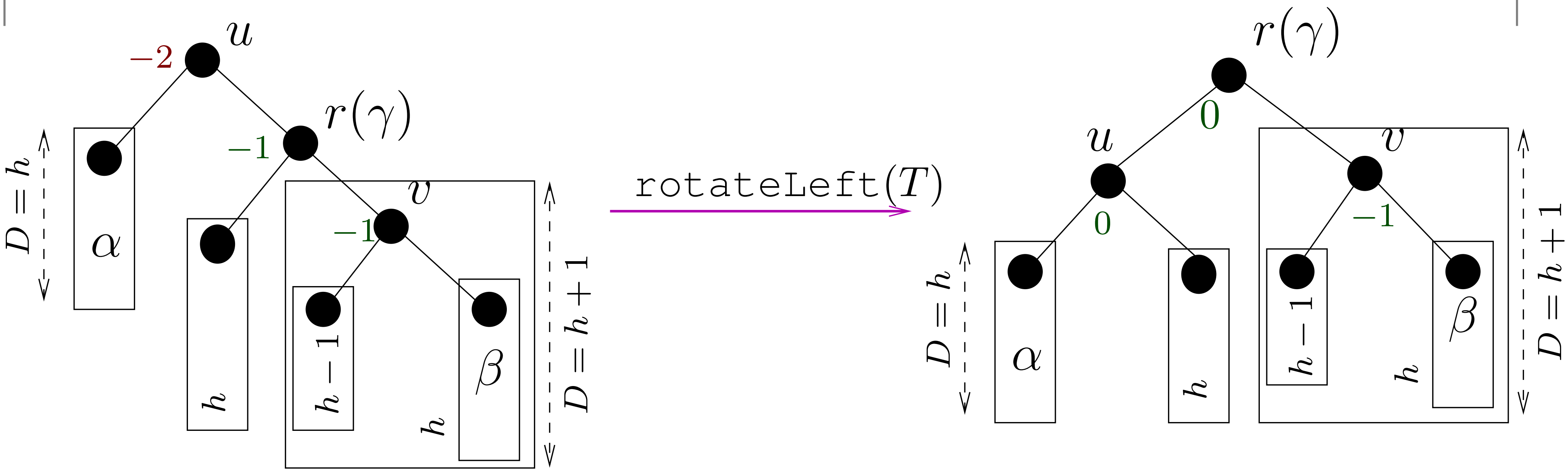
Now we can rotate tree$(v) = \mathsf{R}(u)$

Rotate a subtree right



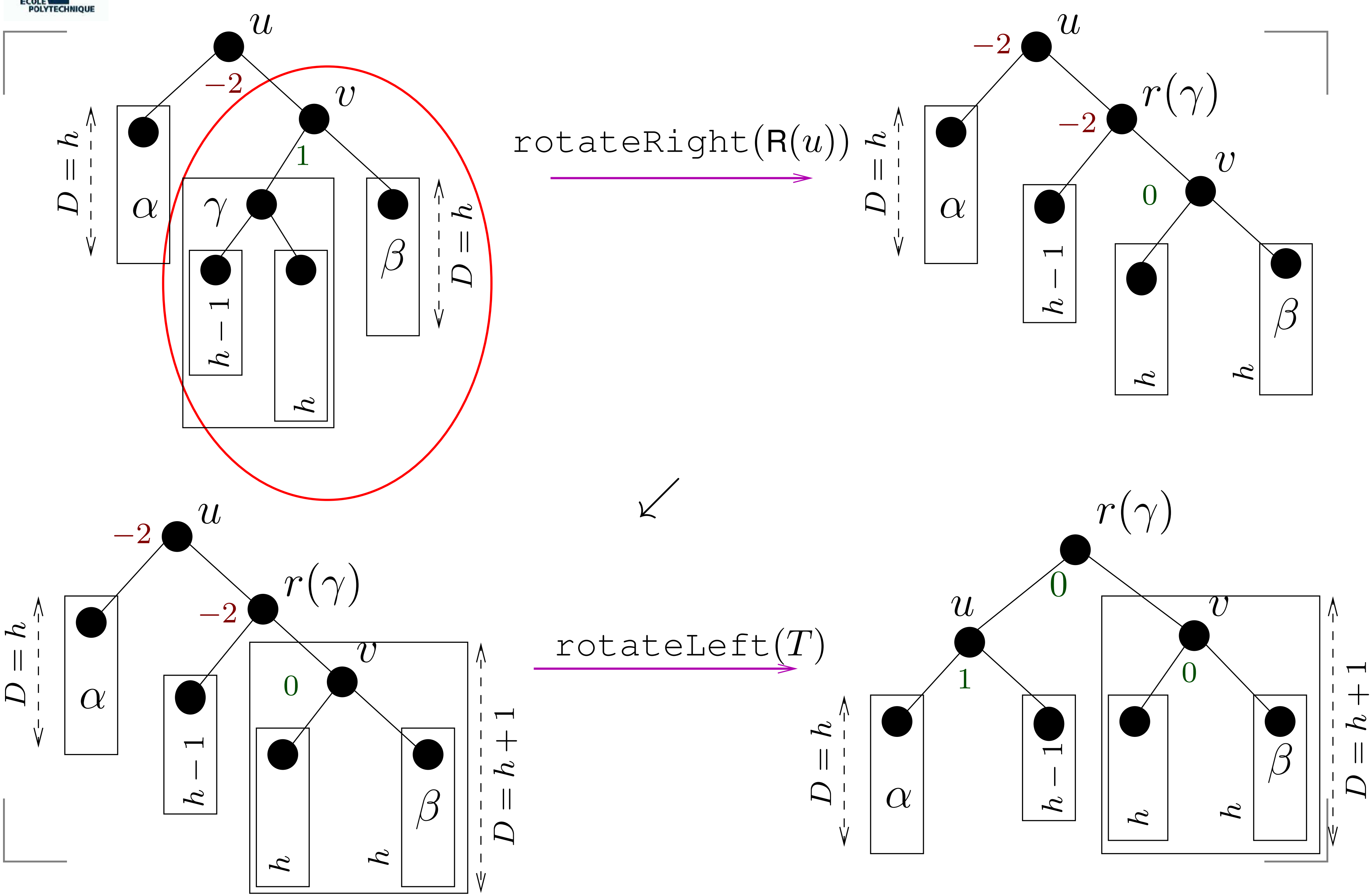
Rotate R(u) right

Finally, rotate left

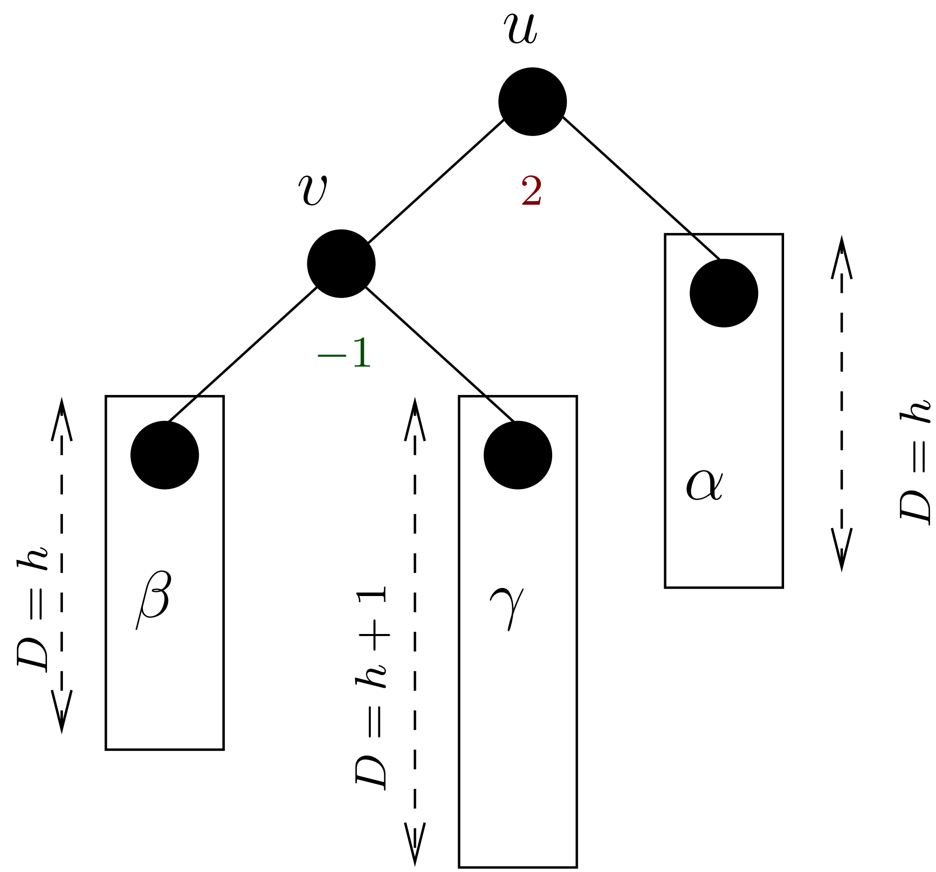


Rotate T left

Symmetric cases I



Symmetric cases II



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

Rotations vs. optimism

- Get rid of rotations, and trust chance: probability that random BST is balanced?
- Given a sequence $\sigma \in \{1, \ldots, n\}^n$, we insert it in a BST T
- Assume $|\mathsf{L}(T)| = K$ and $|\mathsf{R}(T)| = n - 1 - K$
- Assume uniform distribution on K i.e. $\forall k \le n \; P(K = k) = \frac{1}{n}$

σ	(1,2,3)	(1,3,2)	(2,1,3)	(2,3,1)	(3,1,2)	(3,2,1)
T	1 $\searrow$ 2 $\searrow$ 3	1 $\searrow$ 3 $\swarrow$ 2	2 $\swarrow \; \searrow$ $1 \quad 3$	2 $\swarrow \; \searrow$ $1 \quad 3$	3 $\swarrow$ 1 $\searrow$ 2	2 $\swarrow$ 1 $\swarrow$ 3
type	A	B	C	C	D	E

Type C (balanced) twice as likely as any other type!

The average BST balance

- Average depth for BSTs: $O(\log n)$ [Devroye, 1986]
- Average path length for BSTs: $O(n \log n)$ [Vitter & Flajolet, 1990]
- BSTs are well balanced even without rotations!

Heaps and priority queues

Queues reminder

- Queue operations:
  - `pushBack`(v): insert v at the end
  - `popFront`$()$: return and remove element at the beginning
- Used in BFS (compute paths with fewest arcs, see Lecture 2)
- If arcs are prioritized (e.g. travelling times for route segments): want queue to return *element with highest priority*

This may not be at the beginning of the queue

Priority queues

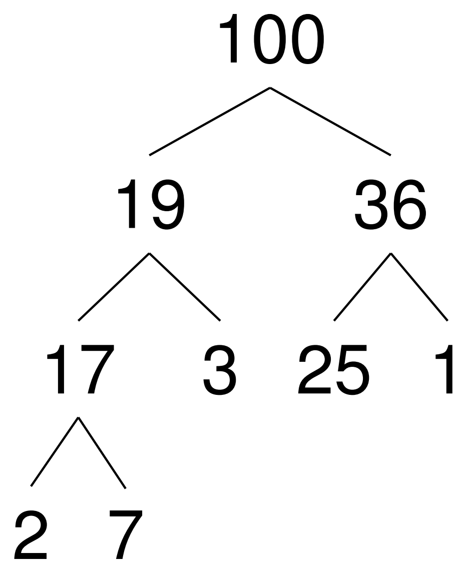
- V: a set; $(S, <)$: a totally ordered set
- **Priority queue** on V, S: set Q of pairs (v, p_v) s.t. $v \in V$ and $p_v \in S$
- Usually, p_v is a number
- E.g., if p_v is the rank of entrance of v in Q, then Q is a standard queue
- Supports three main operations:
  - `insert`(v, p_v): inserts v in Q with priority p_v
  - `max`$()$: returns the element of Q with maximum priority
  - `popMax`$()$: returns and removes `max`$()$
- Implemented as **heaps**

Heap

- A (binary) **heap** is an abstract, tree-like data structure which offers:
  - $O(\log |Q|)$ `insert`
  - $O(1)$ `max`
  - $O(\log |Q|)$ `popMax`
- `max` in $O(1)$: store max. priority element at BST root
- Invariants:
  - *shape property* : all levels except perhaps the last are fully filled; the last level is filled left-to-right
  - *heap property* : every node stores an element of higher priority than its subnodes

Example

Let $V = \mathbb{N}$, and for all $v \in V$ we let $p_v = v$



A balanced tree

Thm.

If Q is a binary heap, $B(Q) \in \{0, 1\}$

Proof

Follows trivially from the shape property. Since all levels are filled completely apart perhaps from the last, $B(Q) \in \{-1, 0, 1\}$. Since the last is filled left-to-right, $B(Q) \neq -1$

Cor.

A binary heap is a balanced binary tree

Warning: *NOT a BST/AVL*: **heap property not compatible with BST invariant** L$(v) \leq V$ R(v)

Keep the heap balanced: need $O(\log |Q|)$ work to insert/remove

Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

$$\varnothing$$

Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

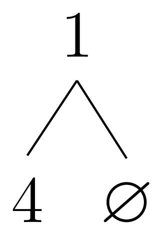
insert 1 $\qquad\qquad$ 1

Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

insert 4

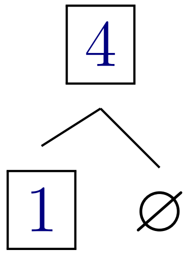


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

$1 < 4$, *swap*

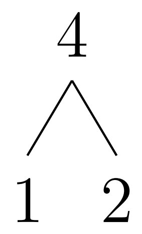


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

insert 2

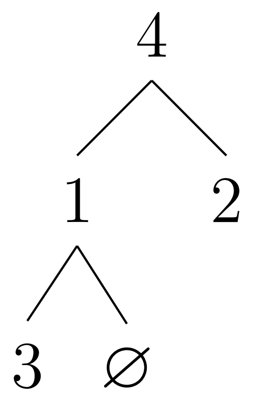


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

insert 3

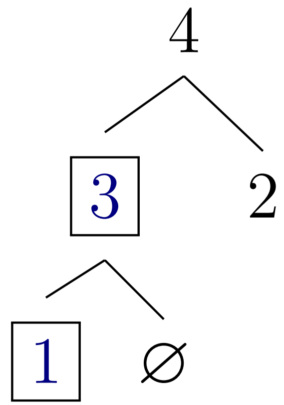


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free “slot”)
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v’s positions in the heap
- Repeat comparison/swap until heap property is attained

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

$1 < 3$, *swap*

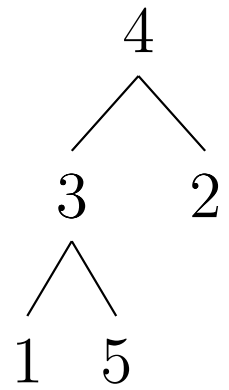


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

insert 5

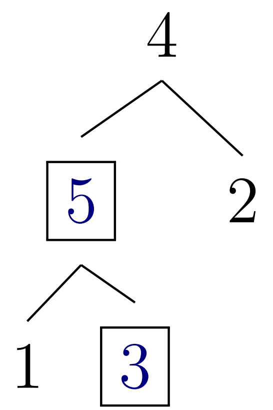


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

$3 < 5$, *swap*

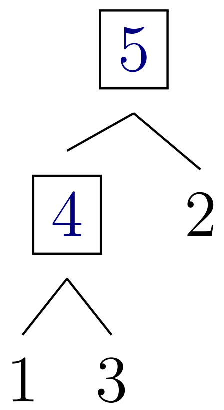


Insert

- Add new element (v, p_v) at the bottom of the heap (last level, leftmost free "slot")
- Compare with its (unique) parent (u, p_u); if $p_u < p_v$, swap u and v's positions in the heap
- Repeat comparison/swap until heap property is attained

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

$4 < 5$, *swap*



Insertion maintains the heap

- Worst case: `insert` takes time proportional to tree depth: $O(\log n)$
- The shape property is maintained:
  - when adding a new element at last level in leftmost free slot
  - when swapping node values along a path to the root
- The heap property is not maintained after adding a new element
- However, it is restored after the sequence of swaps

Thm.

The insertion operation maintains the heap

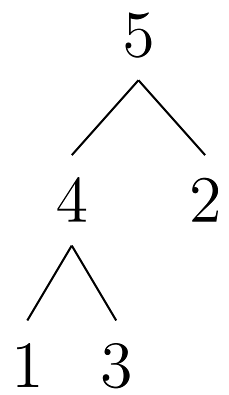
Max

- *Easy*: return the root of the heap tree
- Evidently $O(1)$

Removal of max

- Let $\texttt{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\texttt{last}(Q)$ to the root $r(Q)$
- Compare v with its children u, w: if $p_v \geq p_u, p_v \geq p_w$, heap is in correct order
- Otherwise, swap v with $\max_p(u, v)$ *(use* $\min_p$ *if min-heap)* and repeat comparison/swap until termination

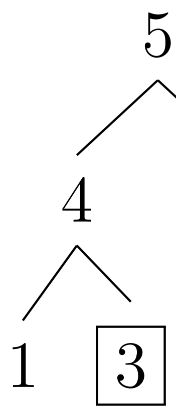
original tree



Removal of max

- Let $\texttt{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\texttt{last}(Q)$ to the root $r(Q)$
- Compare v with its children u, w: if $p_v \geq p_u, p_v \geq p_w$, heap is in correct order
- Otherwise, swap v with $\max_p(u, v)$ *(use* $\min_p$ *if min-heap)* and repeat comparison/swap until termination

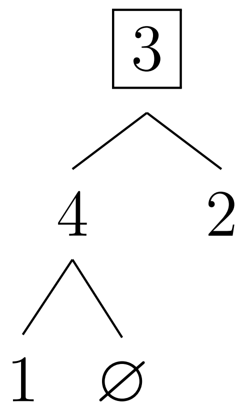
$\texttt{last}(Q) = 3$



Removal of max

- Let $\texttt{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\texttt{last}(Q)$ to the root $r(Q)$
- Compare v with its children u, w: if $p_v \geq p_u, p_v \geq p_w$, heap is in correct order
- Otherwise, swap v with $\max_p(u, v)$ *(use $\min_p$ if min-heap)* and repeat comparison/swap until termination

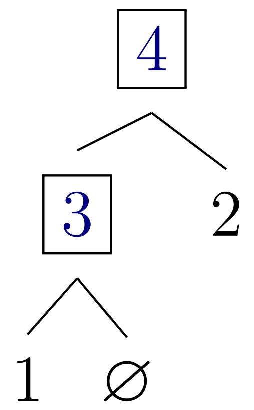
move to root



Removal of max

- Let $\texttt{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\texttt{last}(Q)$ to the root $r(Q)$
- Compare v with its children u, w: if $p_v \geq p_u, p_v \geq p_w$, heap is in correct order
- Otherwise, swap v with $\max_p(u, v)$ *(use $\min_p$ if min-heap)* and repeat comparison/swap until termination

$3 < 4$, *swap*



Efficient construction

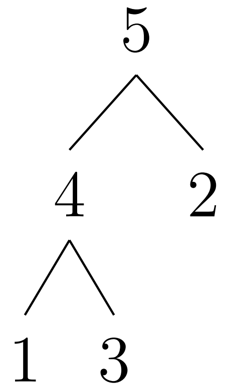
- Insert n elements of V in an empty heap
- Trivially: each insert takes $O(\log n)$, get $O(n \log n)$ to construct the whole heap
- Instead:
  1. arbitrarily put the element in a binary tree with the shape property (can do this in $O(n)$)
  2. lower level first, move nodes down using the same swapping procedure as for `popMax`
- At level ℓ, moving a node down costs $O(\ell)$ (worst-case)
- There's $\leq \lceil \frac{n}{2^{\ell+1}} \rceil$ nodes at level ℓ and $O(\log n)$ possible levels

$$\sum_{\ell=0}^{\lceil \log n \rceil} \frac{n}{2^{\ell+1}} O(\ell) = O(n \sum_{\ell=0}^{\lceil \log n \rceil} \frac{1}{2^\ell}) \leq O(n \sum_{\ell=0}^{\infty} \frac{1}{2^\ell}) = O(2n) = O(n)$$

Implementation

- A priority queue is implemented as a heap
- A heap can be implemented as a tree
- But it needn't be!

Binary trees in arrays



Node	5	4	2	1	3
Index	0	1	2	3	4
		i		$2i+1$	$2i+2$

- Heap Q of n elements stored in an array q of length n
- $q_0 = r(Q)$
- **Subnodes**
  If $q_i = v$, then $q_{2i+1} = \mathsf{L}(v))$ and $q_{2i+2} = \mathsf{R}(v)$ (whenever $2i+1, 2i+2 < n$)
- **Parent**
  If $v \neq q_0$, $P(v) = q_j$, where $j = \lfloor \frac{i-1}{2} \rfloor$

We now have all the elements: start implementing!

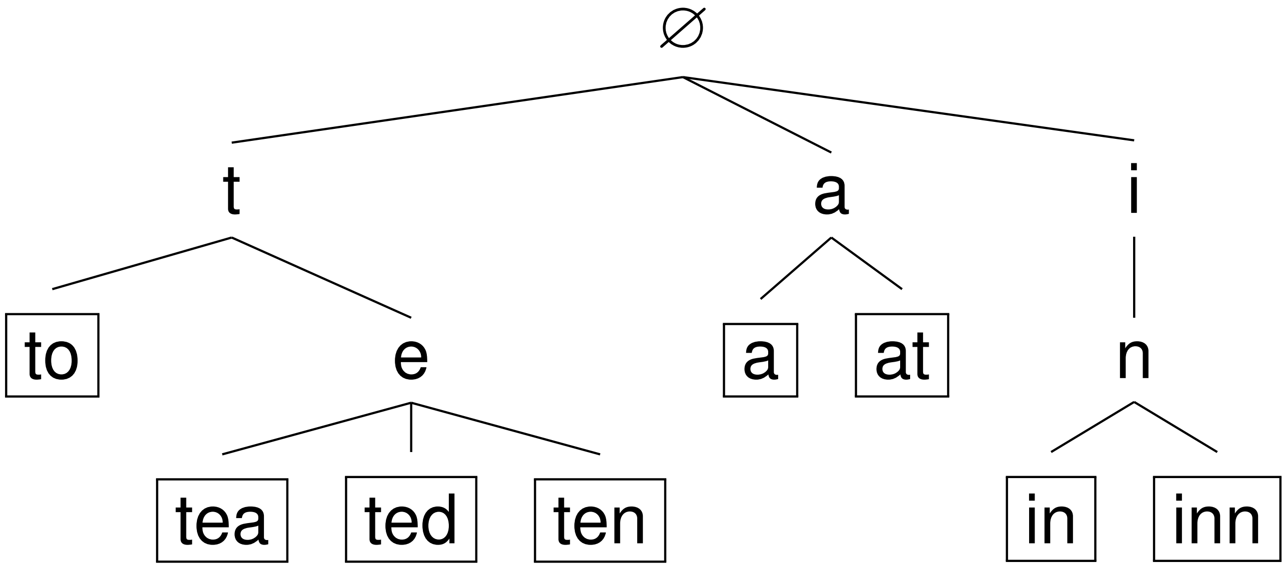
k-ary Search Trees

Tries

search	hashing
look for a key	use key to find its position
use a total order	each key defines a path to a leaf

Trie example

$$V = \{\text{a,at,to,tea,ted,ten,in,inn}\}$$



- Each key is stored at a leaf node ℓ
- Each non-leaf node v contains a prefix of all keys stored in the tree rooted at v
- The trie root node is $\varnothing$, the empty string

Trie properties

- Path on trie corresponding to key k: given by key itself

  Compare with hash functions: hash value specified by key

- If max length key is m, path length $O(m)$
- `find`, `insert` and `delete` take worst-case $O(m)$
- If m constant w.r.t. $n = |V|$, then methods are $O(1)$
- *Comparison to hash functions*:
  - With respect to hashing, tries support "ordered iteration"
  - Hash tables need re-hashing (expensive) as they become full; tries adjust to size gracefully
  - No need to construct good hash functions

Warning: there are several trie variants

End of Lecture 7