

INF421, Lecture 7

Balanced Trees

Leo Liberti

LIX, École Polytechnique, France

Course

- **Objective:** to teach you some data structures and associated algorithms
- **Evaluation:** TP noté en salle info le 16 septembre, Contrôle à la fin.
Note: $\max(CC, \frac{3}{4}CC + \frac{1}{4}TP)$
- **Organization:** fri 26/8, 2/9, 9/9, 16/9, 23/9, 30/9, 7/10, 14/10, 21/10, amphi 1030-12 (Arago), TD 1330-1530, 1545-1745 (SI31,32,33,34)
- **Books:**
 1. Ph. Baptiste & L. Maranget, *Programmation et Algorithmique*, Ecole Polytechnique (Polycopié), 2009
 2. G. Dowek, *Les principes des langages de programmation*, Editions de l'X, 2008
 3. D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1997
 4. K. Mehlhorn & P. Sanders, *Algorithms and Data Structures*, Springer, 2008
- **Website:** `www.enseignement.polytechnique.fr/informatique/INF421`
- **Contact:** `liberti@lix.polytechnique.fr` (e-mail subject: INF421)

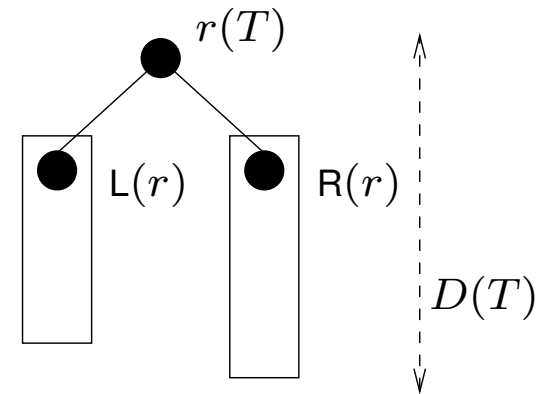
Lecture summary



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

Notation

- Tree T
- Node set of T : $V(T)$ (with $|V(T)| = n$)
- Root: $r(T)$
- Tree rooted at v : $T(v)$
- Node: $v \in V(T)$
- Root node of left subtree of v : $L(v)$
- Root node of right subtree of v : $R(v)$
- If $L(v) = R(v) = \emptyset$, v is a **leaf node**
- Parent node of v : $P(v)$
- $\Rightarrow T = \langle L(r(T)), r(T), R(r(T)) \rangle$
- For all $v \in V(T)$: $p(v)$ = unique path $r(T) \rightarrow v$
- Path length: $\lambda(T) = \sum_{v \in V(T)} |p(v)|$
- Depth (or height): $D(T) = \max_{v \in V(T)} |p(v)|$



The minimal knowledge

- Let $(V, <)$ be a totally ordered set
- V stored as a binary tree T :

$$L(v) = u \Rightarrow u \leq v \quad R(v) = u \Rightarrow u > v \quad (\dagger)$$

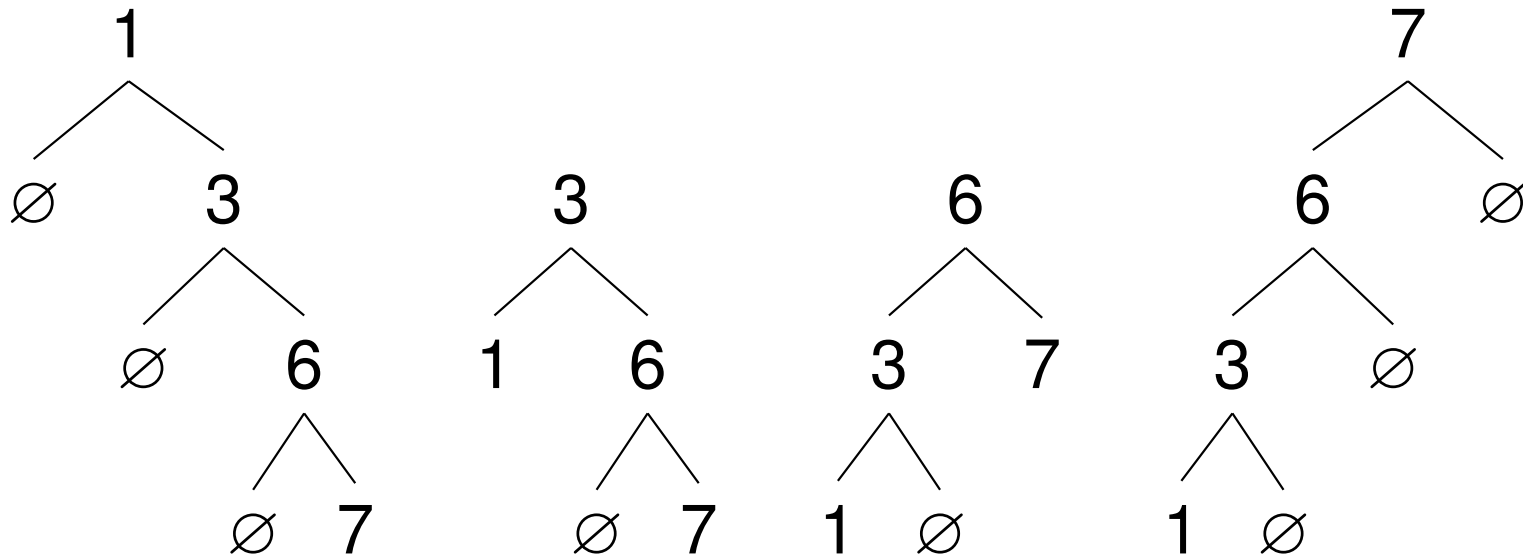
- find, insert, delete, min, max:
 $O(\log n)$ on average, $O(n)$ worst case
- AVL trees: balance**
$$B(T) = D(\mathsf{T}(L(r(T)))) - D(\mathsf{T}(R(r(T)))) \in \{-1, 0, 1\}$$
- If an operation unbalances, use a **rebalancing** operation
- \Rightarrow all operations are $O(\log n)$ in the worst case
- Can use a special balanced tree (a **heap**) to implement a **priority queue** (min/max, insert, delete)
- Tries** are k -ary trees that encode words prefix-wise

Binary search trees (BST)

Sorted sequences



- Used to store a set V as a **sorted sequences**
- Makes it efficient to answer the question $v \in V$
- Each node v in the tree is such that $L(v) \leq v < R(v)$
- Example: $V = \{1, 3, 6, 7\}$



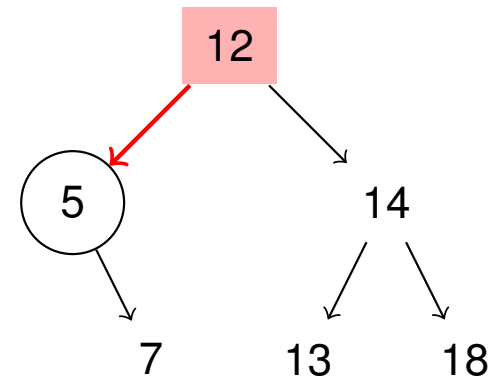
- Several possibilities

BST min/max

● min(v):

```

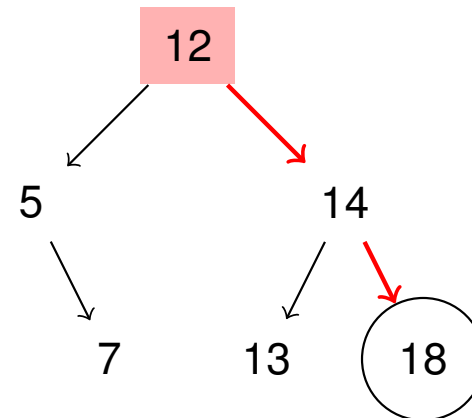
1: if  $L(v) = \emptyset$  then
2:   return  $v$ ;
3: else
4:   return min( $L(v)$ );
5: end if
  
```



● max(v):

```

1: if  $R(v) = \emptyset$  then
2:   return  $v$ ;
3: else
4:   return max( $R(v)$ );
5: end if
  
```



Base cases for recursion



All other BST functions $\mathfrak{f}(k, v)$ are assumed to be implemented so that $\mathfrak{f}(k, \emptyset)$ returns without doing anything (base case of recursion)

BST find

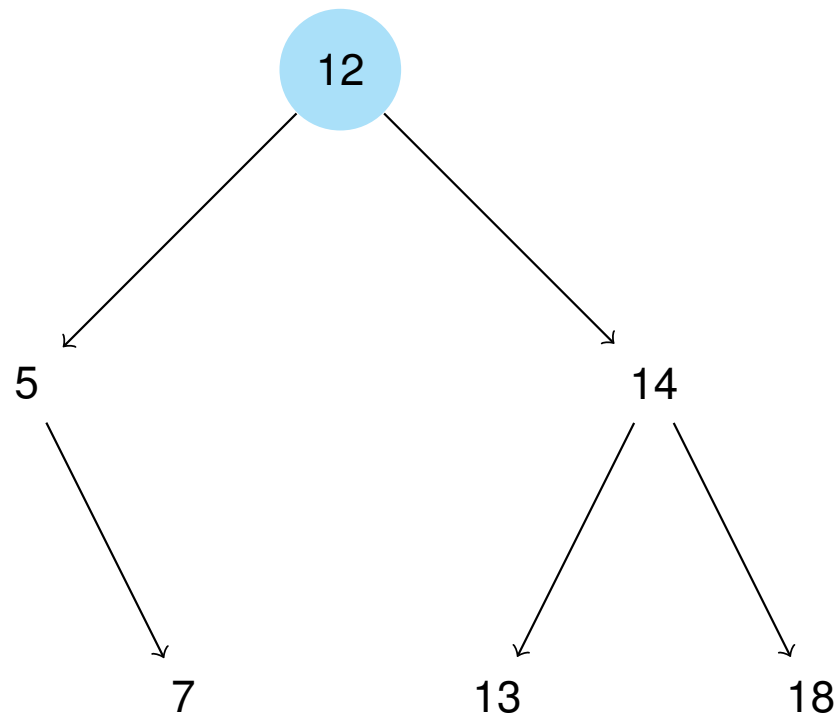


find(k, v):

```
1: ret = not_found;  
2: if  $v = k$  then  
3:   ret =  $v$ ;  
4: else if  $k < v$  then  
5:   ret = find( $k, L(v)$ );  
6: else  
7:   ret = find( $k, R(v)$ );  
8: end if  
9: return ret;
```

Successful find

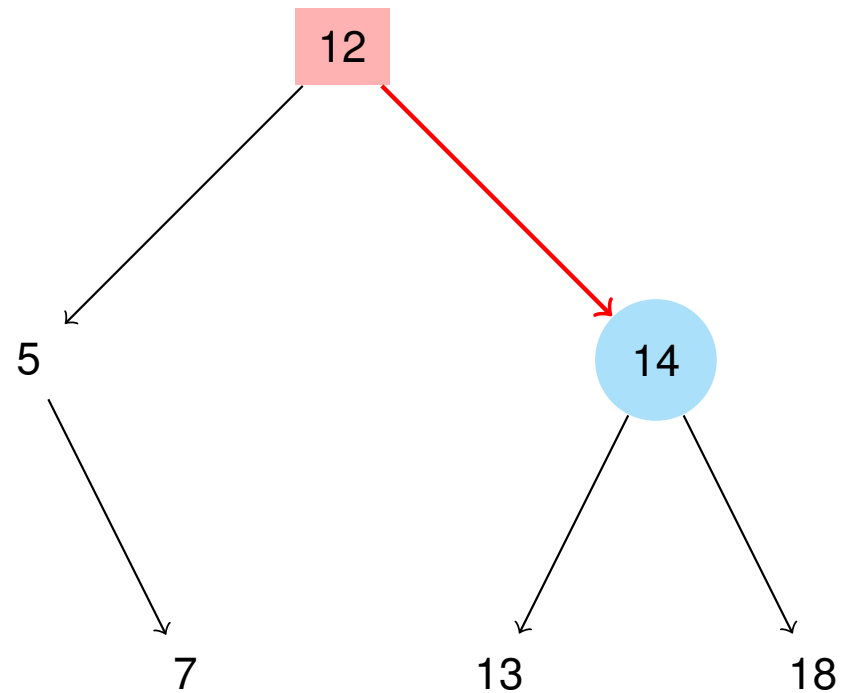
$\text{find}(13, r(T))$



$13 > 12$, take right branch

Successful find

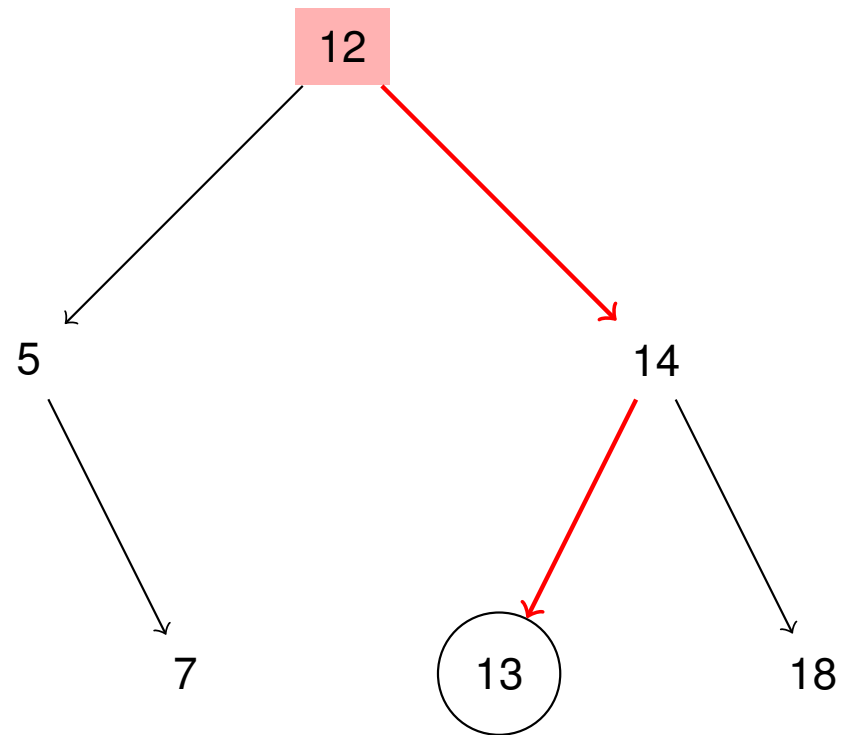
$\text{find}(13, r(T))$



$13 < 14$, take left branch

Successful find

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

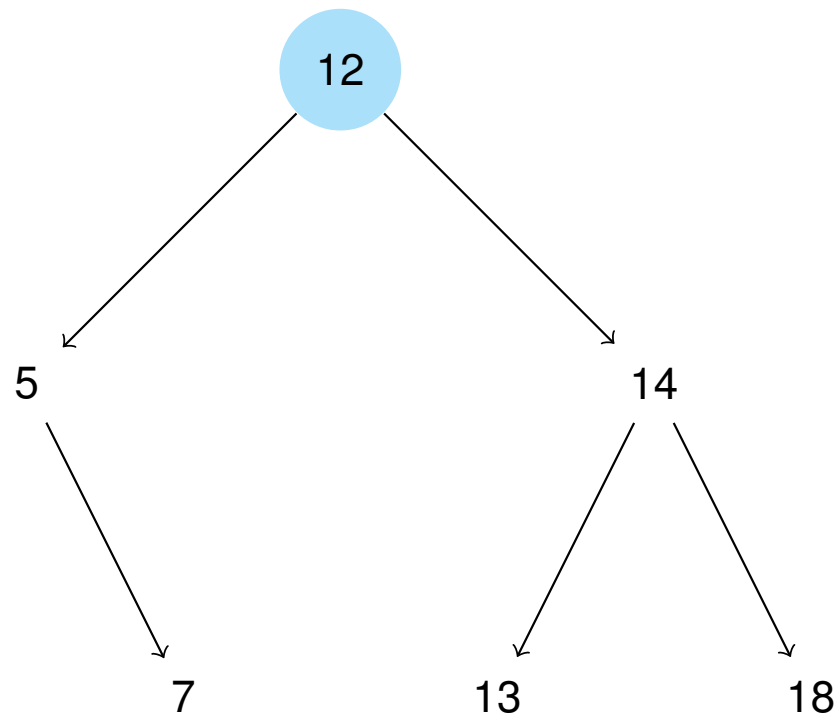


found 13

Unsuccessful find



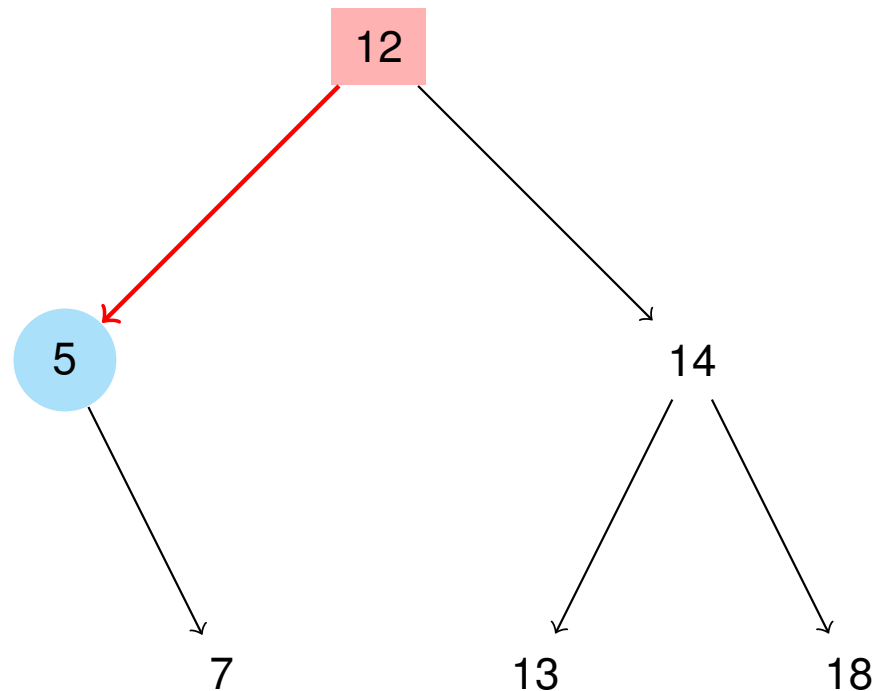
$\text{find}(1, r(T))$



$1 < 12$, take left branch

Unsuccessful find

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



$1 < 5$, should take left branch but $L(5) = \emptyset$, not found

BST insert

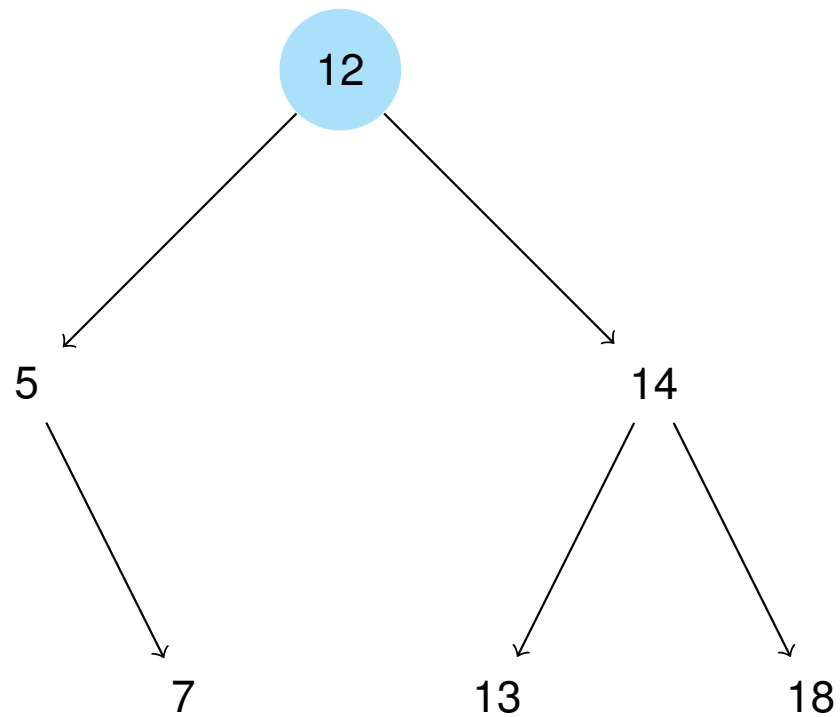


insert(k, v):

```
1: if  $k = v$  then
2:   return already_in_set; // if multiset: add new node
3: else if  $k < v$  then
4:   if  $L(v) = \emptyset$  then
5:      $L(v) = k$ ;
6:   else
7:     insert( $k, L(v)$ );
8:   end if
9: else
10:  if  $R(v) = \emptyset$  then
11:     $R(v) = k$ ;
12:  else
13:    insert( $k, R(v)$ );
14:  end if
15: end if
```


Insert example

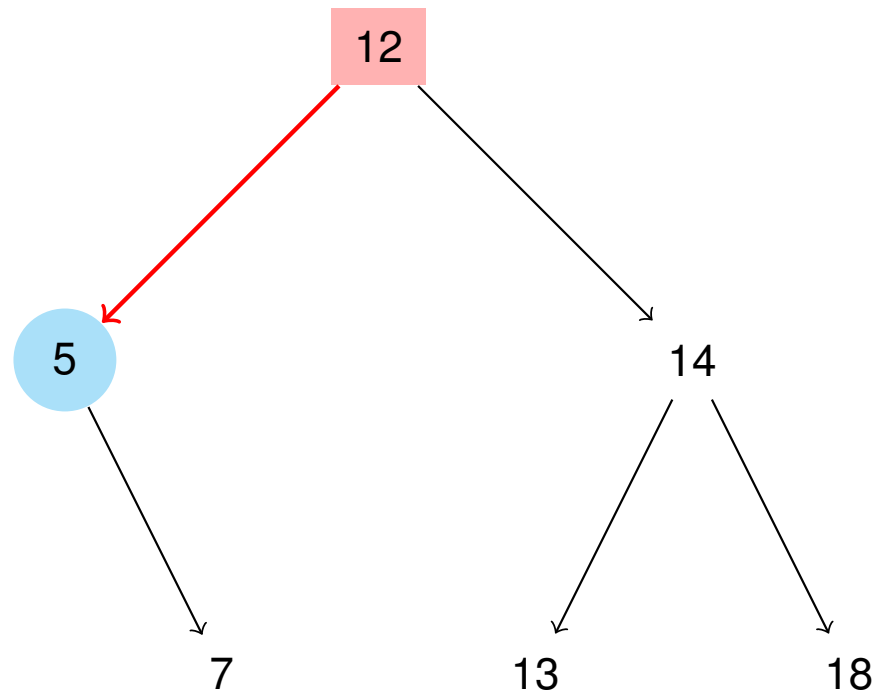
$\text{insert}(1, r(T))$



$1 < 12$, take left branch

Insert example

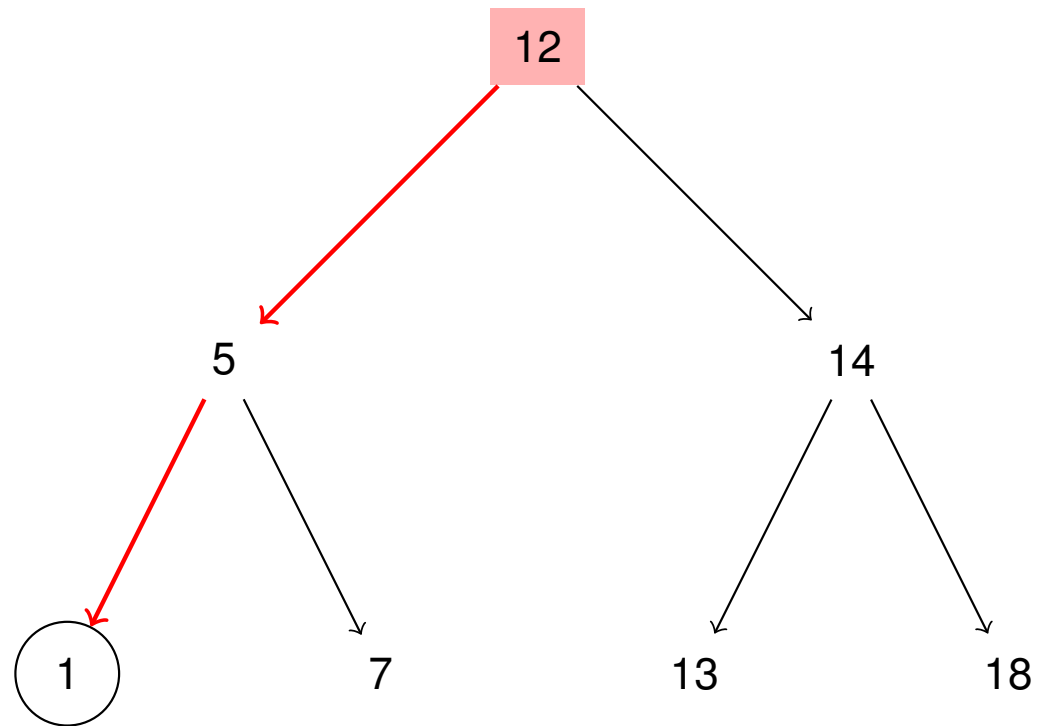
$\text{insert}(1, r(T))$



$1 < 5$, should take left branch but $L(5) = \emptyset$

Insert example

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

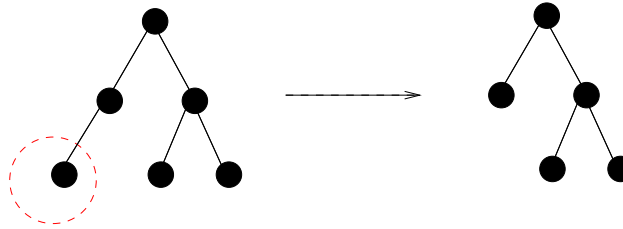


Add $k = 1$ as $L(5)$

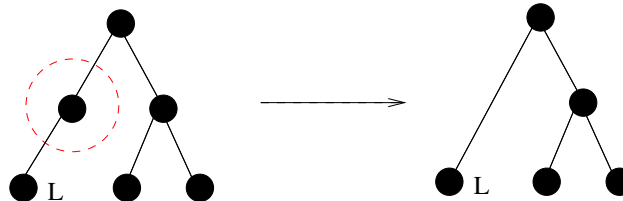
Deletion is not so easy



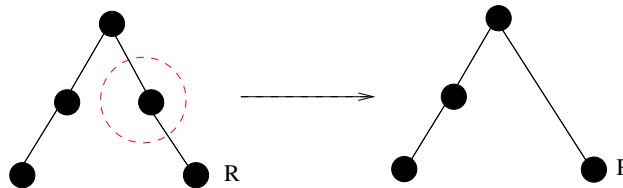
- If node v to delete is a leaf, easy: “cut” it (`unlink`)



- If $R(v) = \emptyset$ and $L(v) \neq \emptyset$, replace with $L(v)$



- If $L(v) = \emptyset$ and $R(v) \neq \emptyset$, replace with $R(v)$



- If v has both subtrees, not evident

Replacing a node



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

```
replace(v, u) // replace v with u
1: if R(P(v)) = v then
2:   R(P(v)) ← u; // u is a right subnode
3: else
4:   L(P(v)) ← u; // u is a left subnode
5: end if
6: if u ≠ ∅ then
7:   P(u) ← P(v);
8: end if
9: unlink v;
```

```
unlink: set L(v) = R(v) = P(v) = ∅
```

Deleting $v : L(v) \neq \emptyset \wedge R(v) \neq \emptyset$

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

- The minimum u of a BST is always the leftmost node without a left subtree
- Hence we know how to delete u (case $L(\cdot) = \emptyset$ in previous slide)
- We replace the value of v by that of u then delete u
- Because $u = \min T(R(v))$, we have $u < w$ for all $w \in T(R(v))$
- Since the value of v is now the value of u , v is now the minimum over all nodes in $T(R(v))$; hence $v < r(R(v))$
- Moreover, since the value of v used to be u , a node in $R(v)$, we have $v > r(L(v))$, satisfying the BST defn. (†)

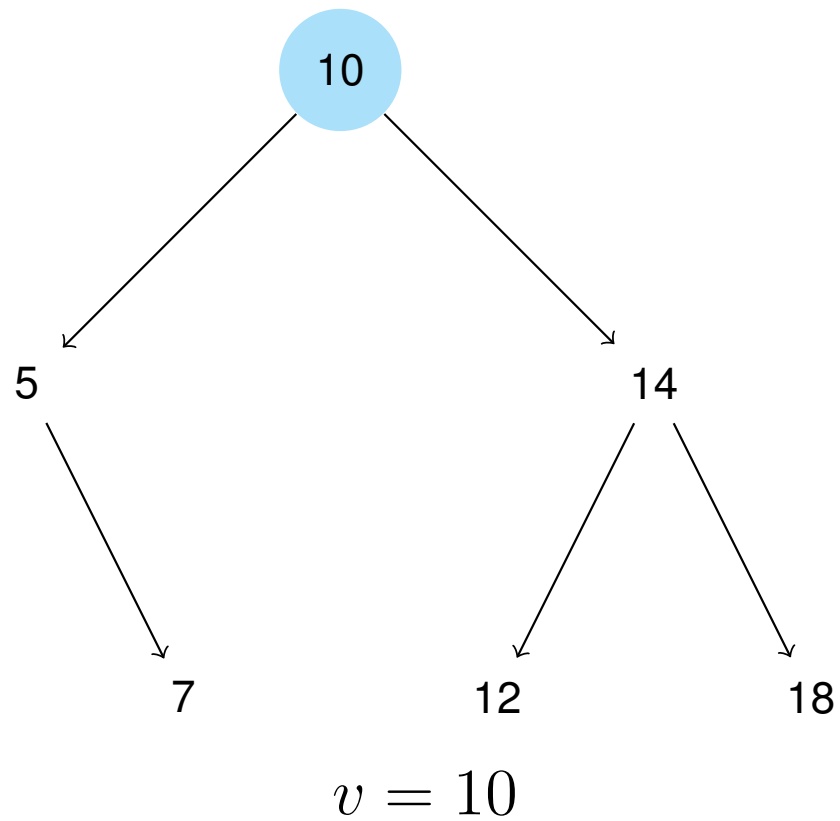
BST delete

● delete(k, v):

- 1: **if** $k < v$ **then**
- 2: delete($k, L(v)$);
- 3: **else if** $k > v$ **then**
- 4: delete($k, R(v)$);
- 5: **else**
- 6: **if** $L(v) = \emptyset \vee R(v) = \emptyset$ **then**
- 7: delete v ; // one of the easy cases
- 8: **else**
- 9: $u = \min(R(v))$;
- 10: swap_values(u, v);
- 11: delete u ; // an easy case, as $L(u) = \text{null}$
- 12: **end if**
- 13: **end if**

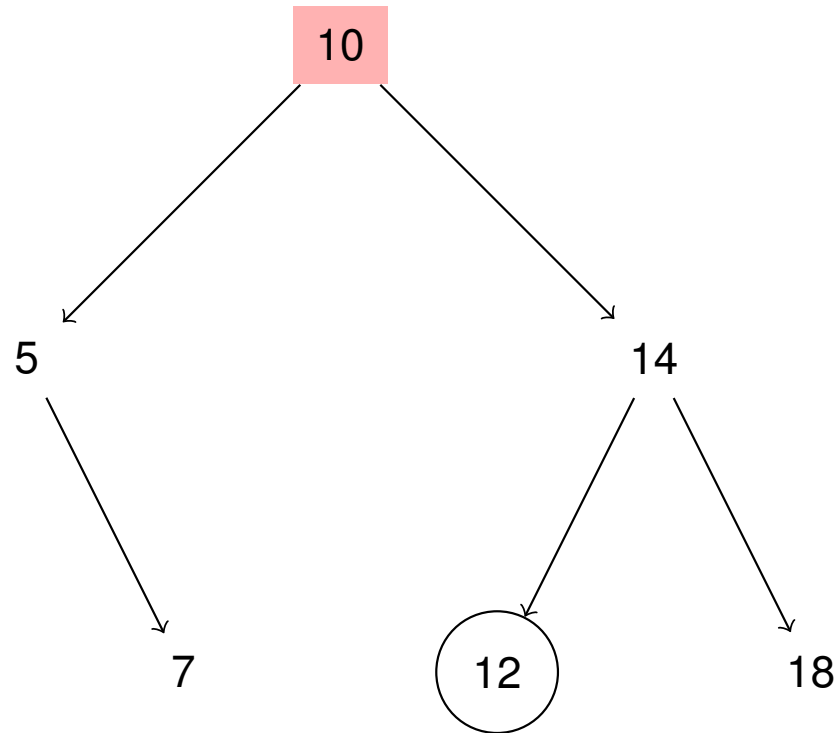
Delete example

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



Delete example

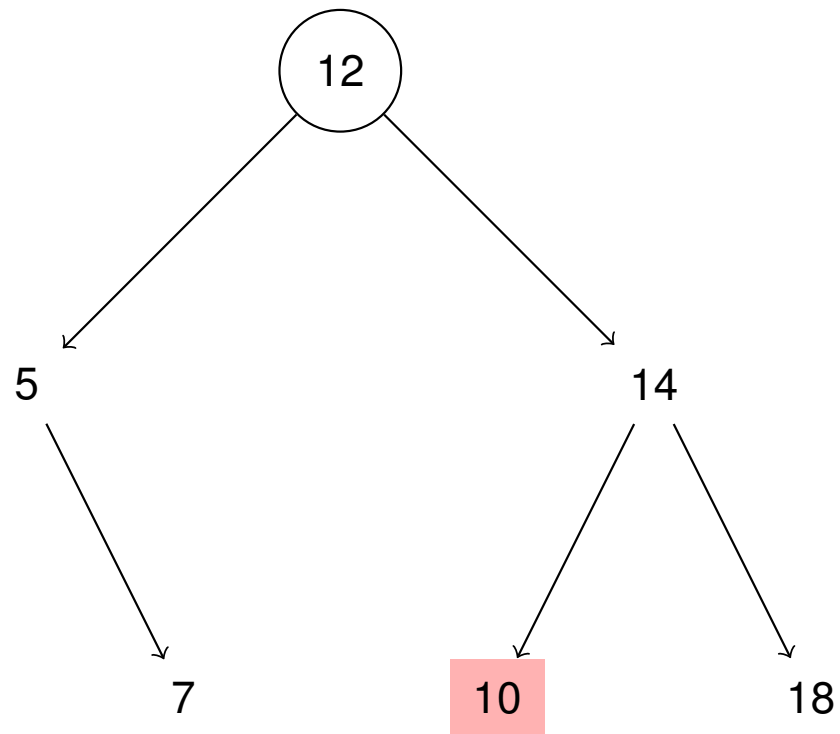
$\text{delete}(10, r(T))$



$$u = \min T(14) = 12$$

Delete example

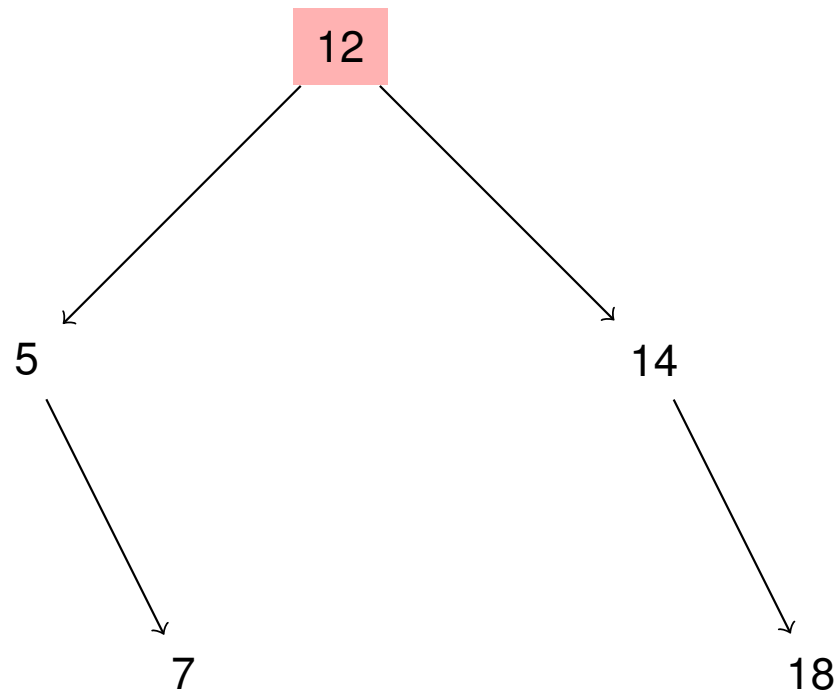
$\text{delete}(10, r(T))$



swap values of 10 and 12

Delete example

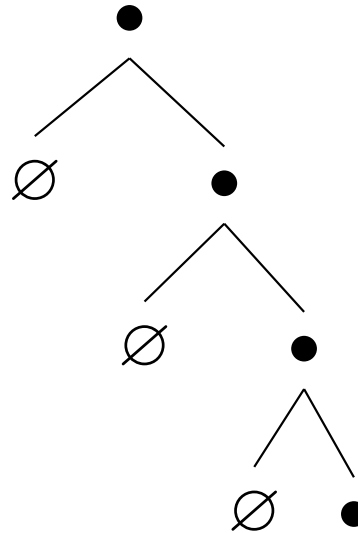
$\text{delete}(10, r(T))$



delete 10

Complexity

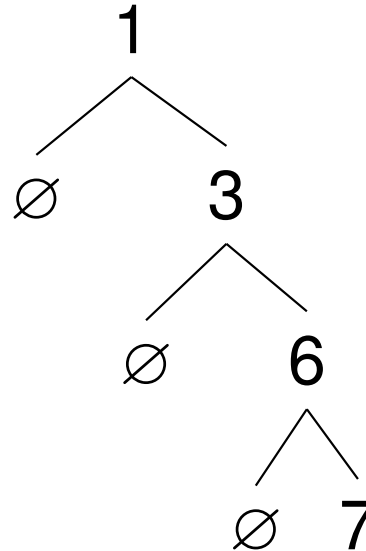
- Each IF case involves at most one recursive call
- Recurse along one branch only
- Worst-case complexity proportional to depth $D(T)$
- If tree is balanced, $D(T)$ is $O(\log n)$ (see INF311)
- In the worst case, $D(T)$ is $O(n)$



Adelson-Velskii & Landis (AVL) trees

AVL Trees

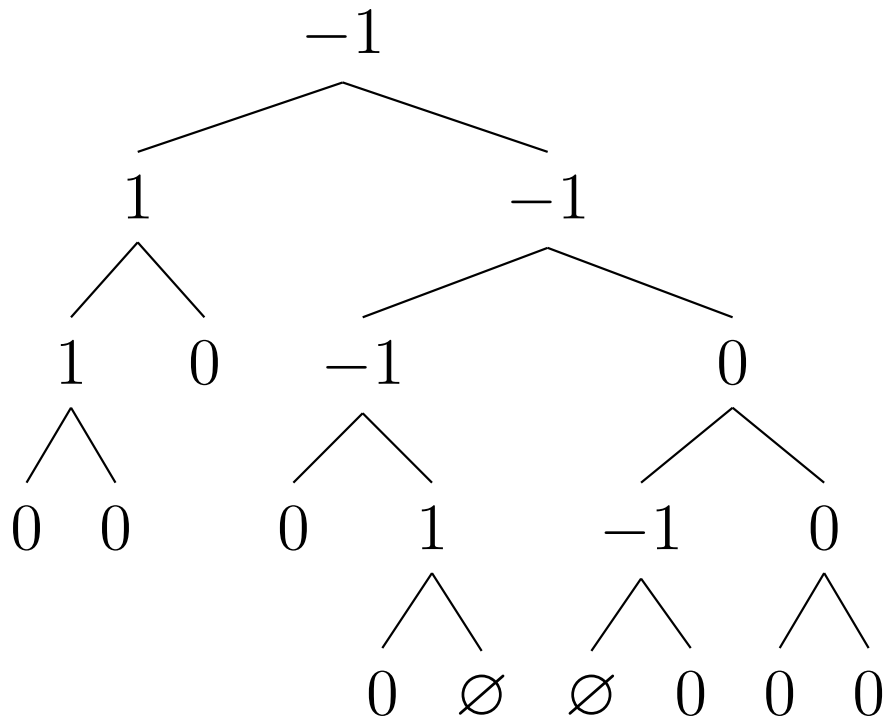
- Try inserting 1, 3, 6, 7 in this order: get **unbalanced tree**



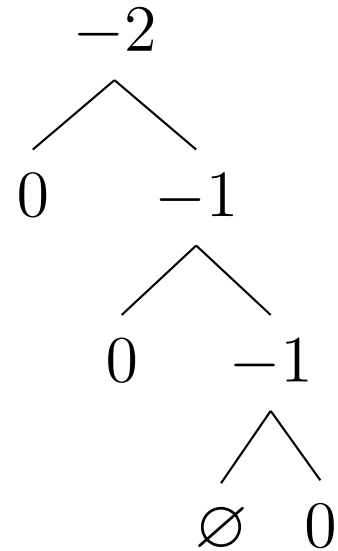
- Worst case find (i.e., find the key 7) is $O(n)$
- Need to *rebalance* the tree to be more efficient
- AVL trees:** at any node, $B(T)$ = depth difference between left and right subtrees $\in \{-1, 0, 1\}$

Examples

AVL tree:



Non-AVL tree:



Nodes indicate $B(\mathsf{T}(v))$

Insertion example



insert 1

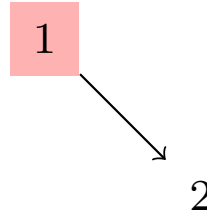
1

$$v_1 = 1;$$

$$r(T) = v_1;$$

Insertion example

insert 2

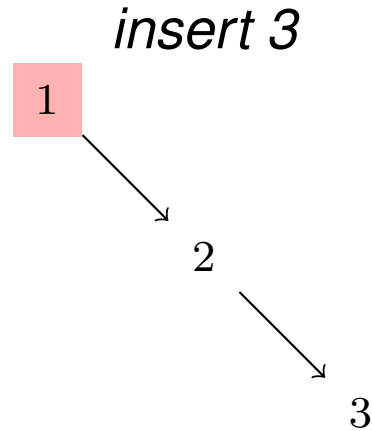


$v_2 = 2;$

$R(v_1) = v_2;$

$P(v_2) = v_1;$

Insertion example



$$v_3 = 3;$$

$$R(v_2) = v_3;$$

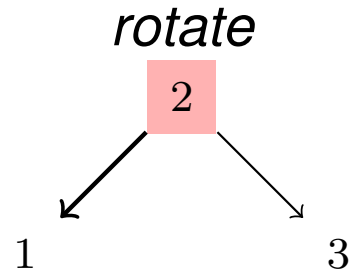
$$P(v_3) = v_2;$$

$$D(L(v_1)) = 0,$$

$$D(R(v_1)) = 2:$$

$$B(T) = -2: \text{ out of balance}$$

Insertion example

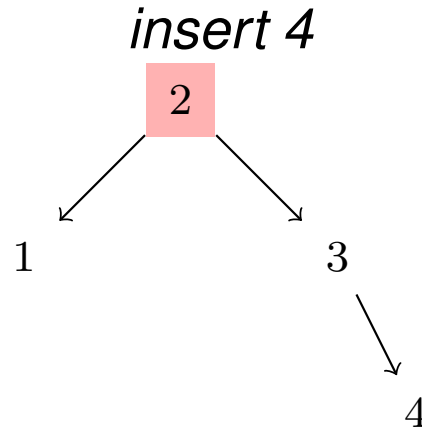


$$r(T) = v_2;$$

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

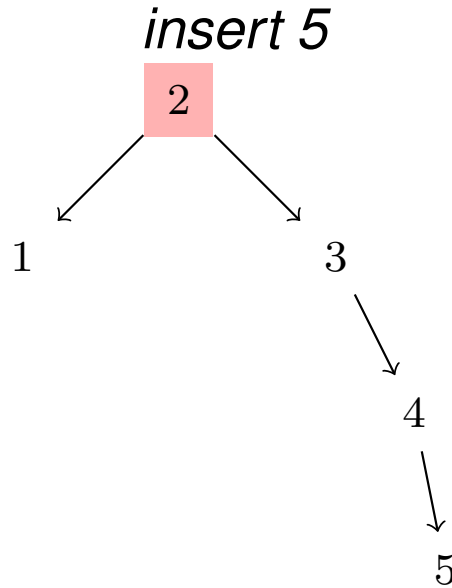
$$P(v_1) = v_2;$$

Insertion example



$v_4 = 4;$
 $R(v_3) = v_4;$
 $P(v_4) = v_3;$

Insertion example



$$v_5 = 5;$$

$$R(v_4) = v_5;$$

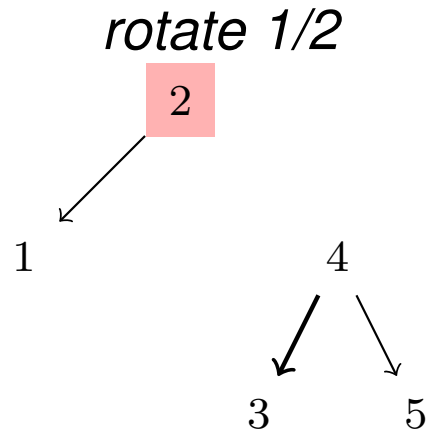
$$P(v_5) = v_4;$$

$$H(L(v_3)) = 0,$$

$$H(R(v_3)) = 2:$$

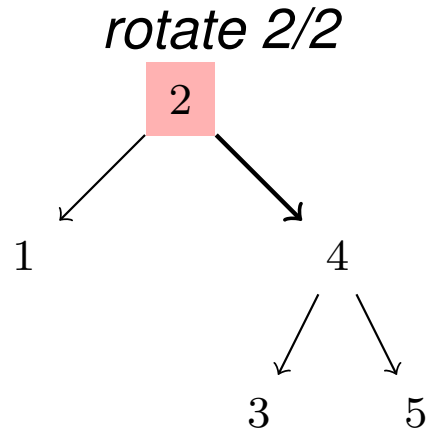
$$B(T) = -2: \text{ out of balance}$$

Insertion example



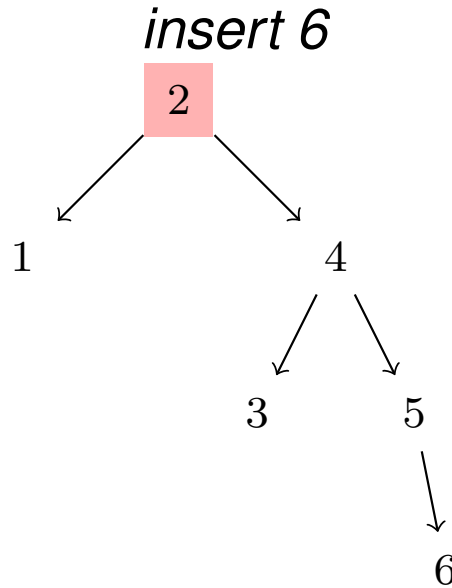
$$L(v_4) = v_3;$$
$$P(v_3) = v_4;$$

Insertion example



$$R(v_2) = v_4;$$
$$P(v_4) = v_2;$$

Insertion example



$$v_6 = 6;$$

$$R(v_5) = v_6;$$

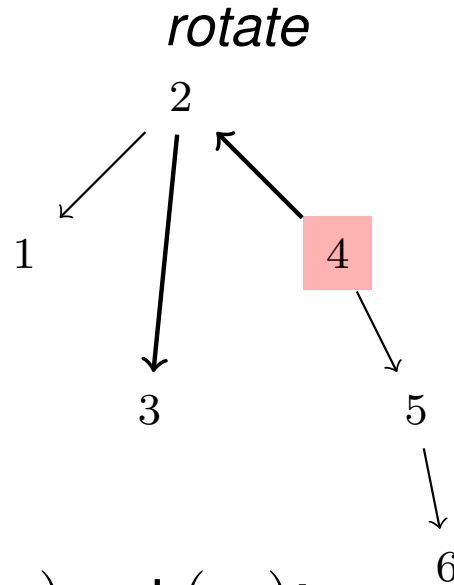
$$P(v_6) = v_5;$$

$$H(L(v_2)) = 1,$$

$$H(R(v_2)) = 3:$$

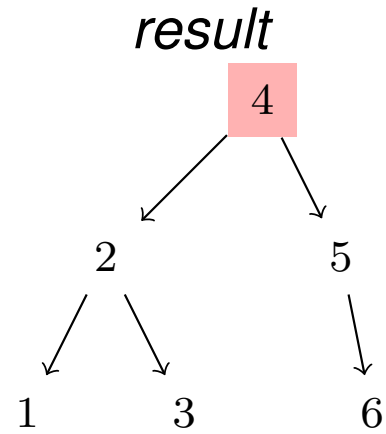
$$B(T) = -2: \text{ out of balance}$$

Insertion example



$$\begin{aligned}
 R(v_2) &= L(v_4); \\
 P(L(v_4)) &= v_2; \\
 L(v_4) &= v_2; \\
 P(v_2) &= v_4; \\
 r(T) &= v_4;
 \end{aligned}$$

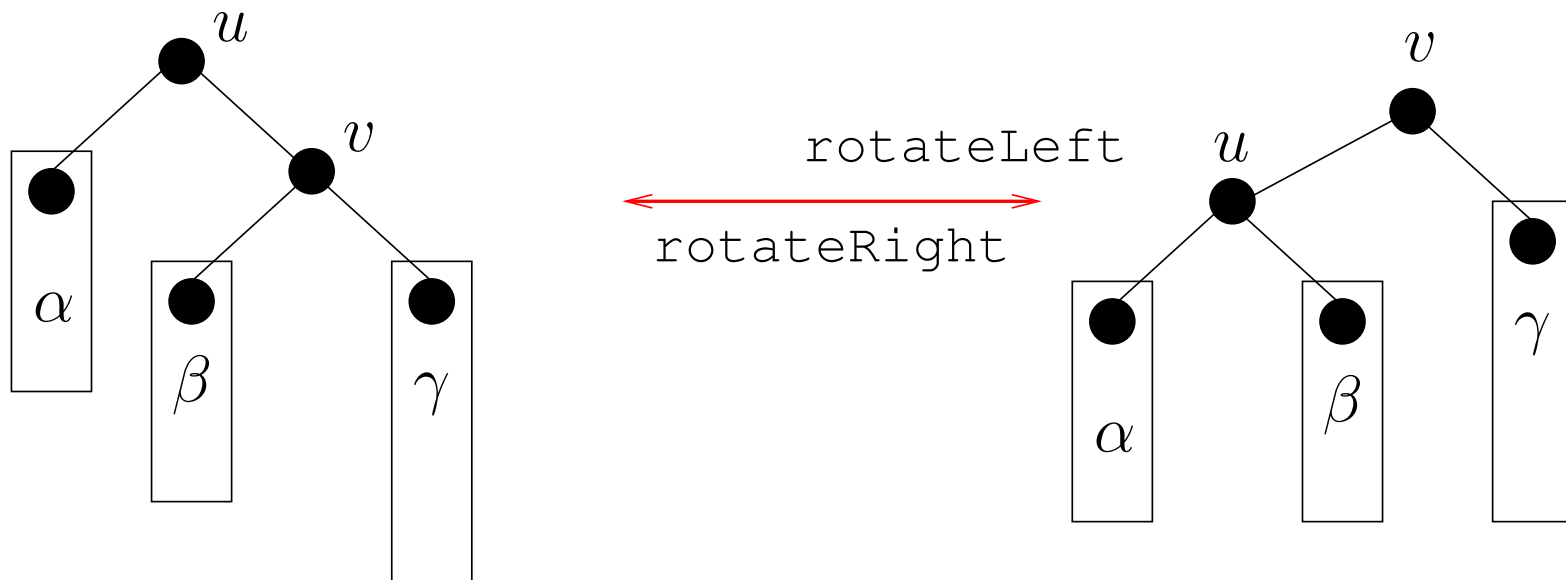
Insertion example



In general

- We can decompose balanced trees operations into:
 - the operation itself
 - a sequence of *rebalancing operations* (when required), called **rotations**
- The operations `min/max`, `find`, `insert`, `delete` are as in BST (with one simple modification)
- Unbalancing can occur on insertion and deletion
- Since we insert/delete only one node at a time, unbalance offset is at most 1 unit
- I.e., $B(T)$ = depth difference between left and right subtrees, could be $\{-2, 2\}$

Left and right rotation



Algebraic interpretation

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

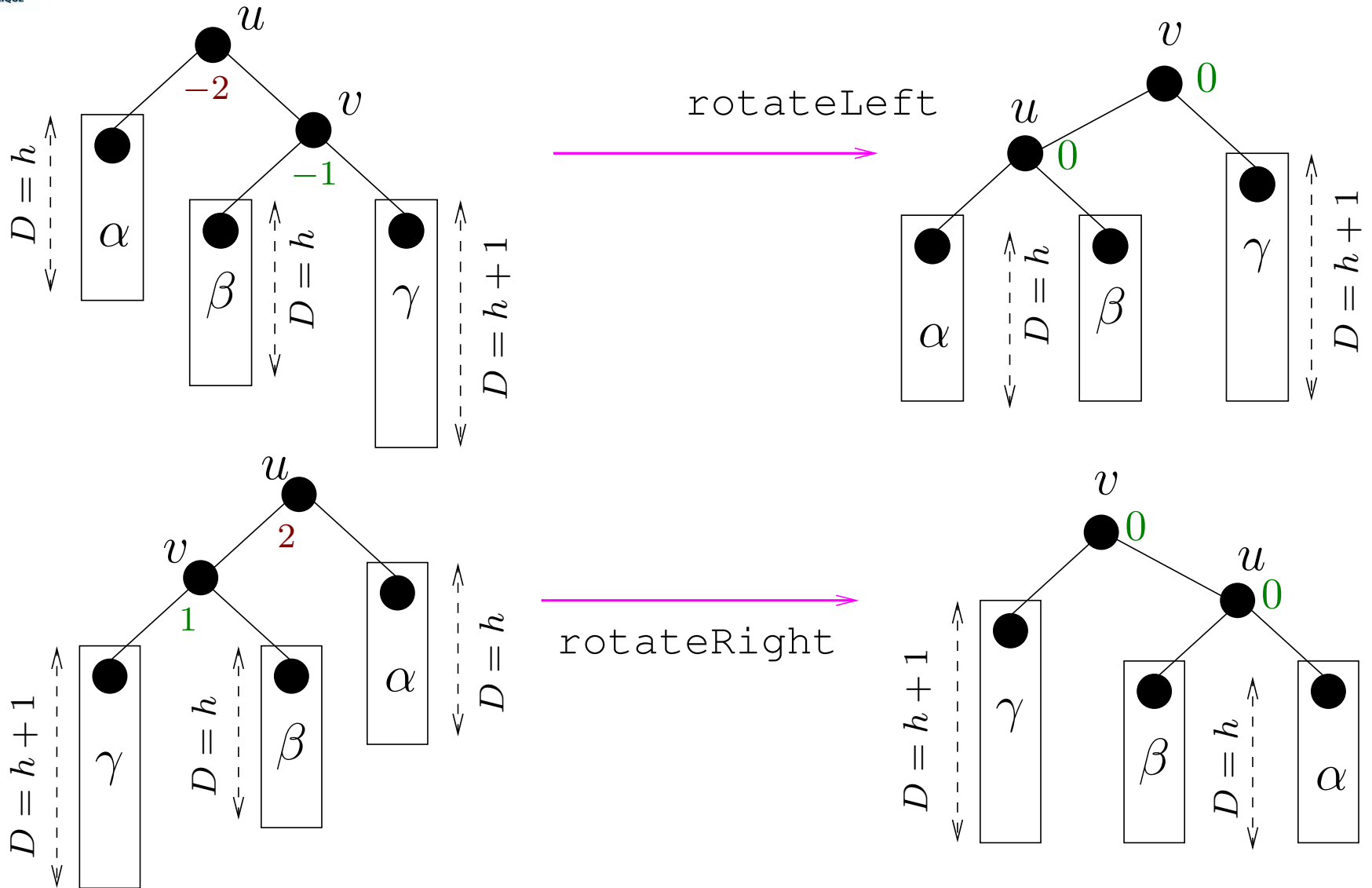
Thm.

$$\begin{aligned}\text{rotateRight}(\text{rotateLeft}(T)) &= \\ \text{rotateLeft}(\text{rotateRight}(T)) &= T\end{aligned}$$

Proof

Directly from the definition

Rotating and rebalancing



Properties of rotation

Thm.

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

Proof

(Sketch): The tree order only changes locally for u, v . In T , $T(v) = R(u)$, which implies $u < v$. In $\text{rotateLeft}(T)$, $T(u) = L(v)$, 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 $B(T') = 2$

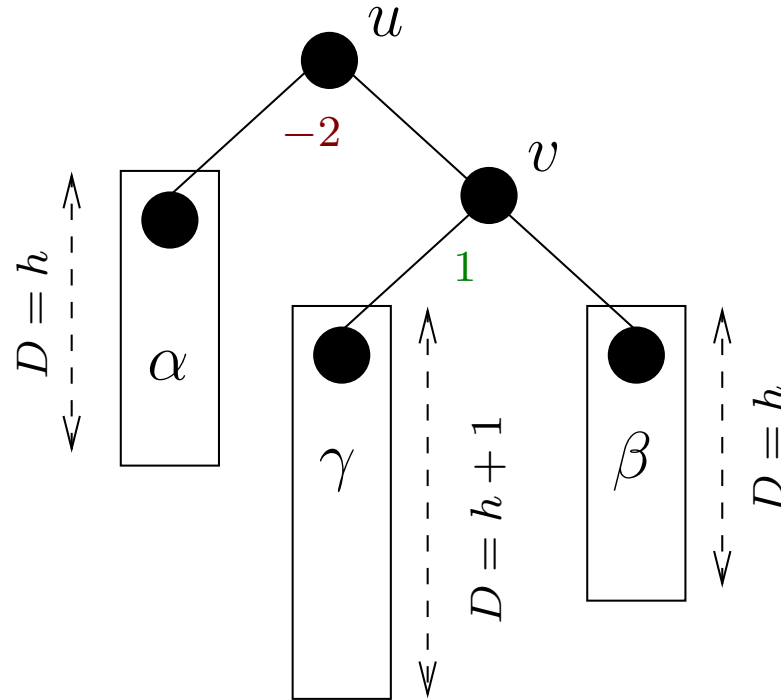
Thm.

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

Proof

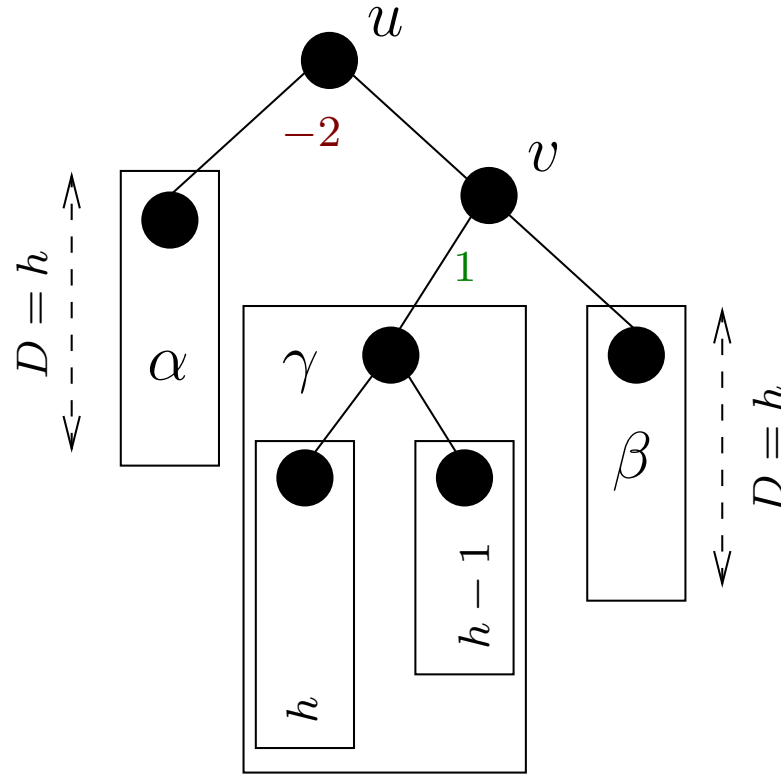
(Sketch): since subtrees α, γ are swapped, tree depth is $D = h$ for all subtrees

Is this enough?



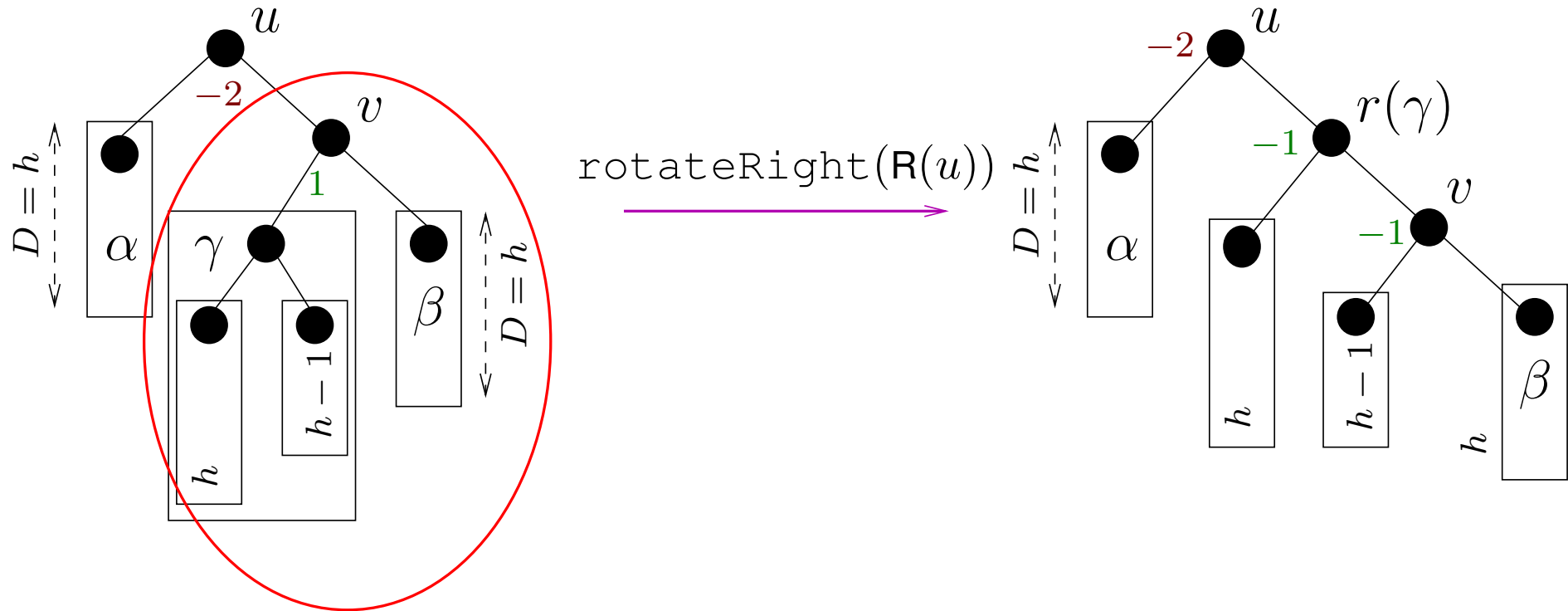
Rotating leaves γ at its place, doesn't work

Break γ up into subtrees



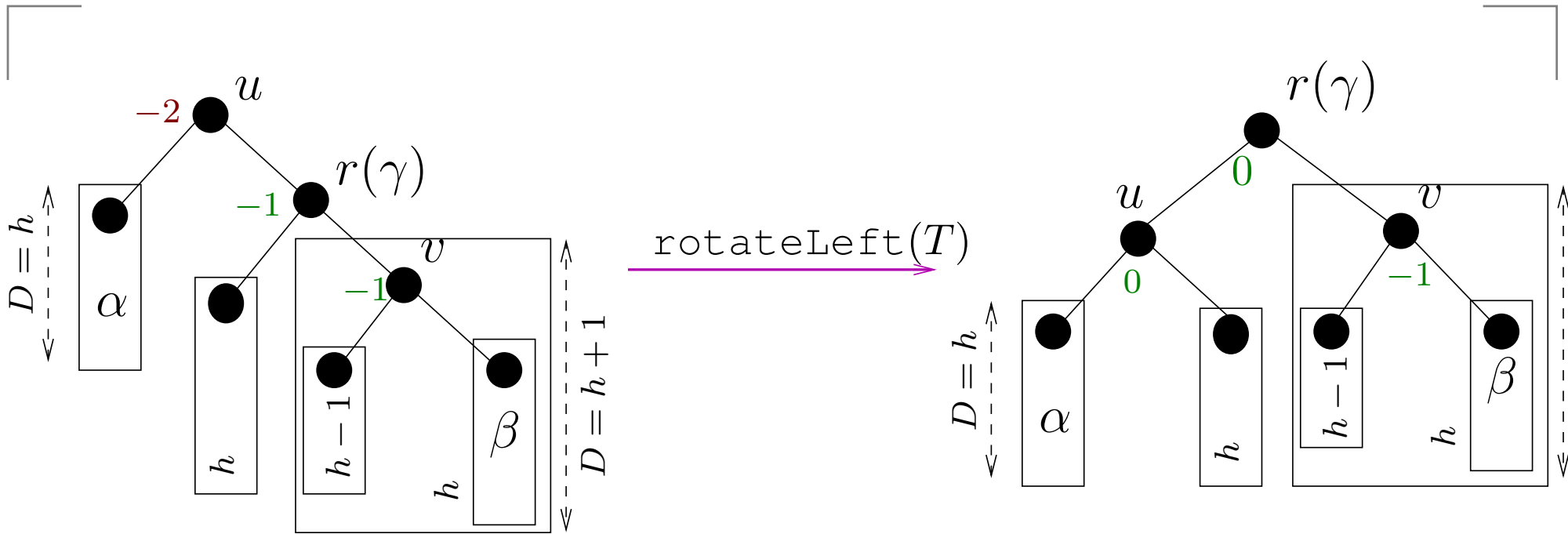
Now we can rotate $T(v) = R(u)$

Rotate a subtree right



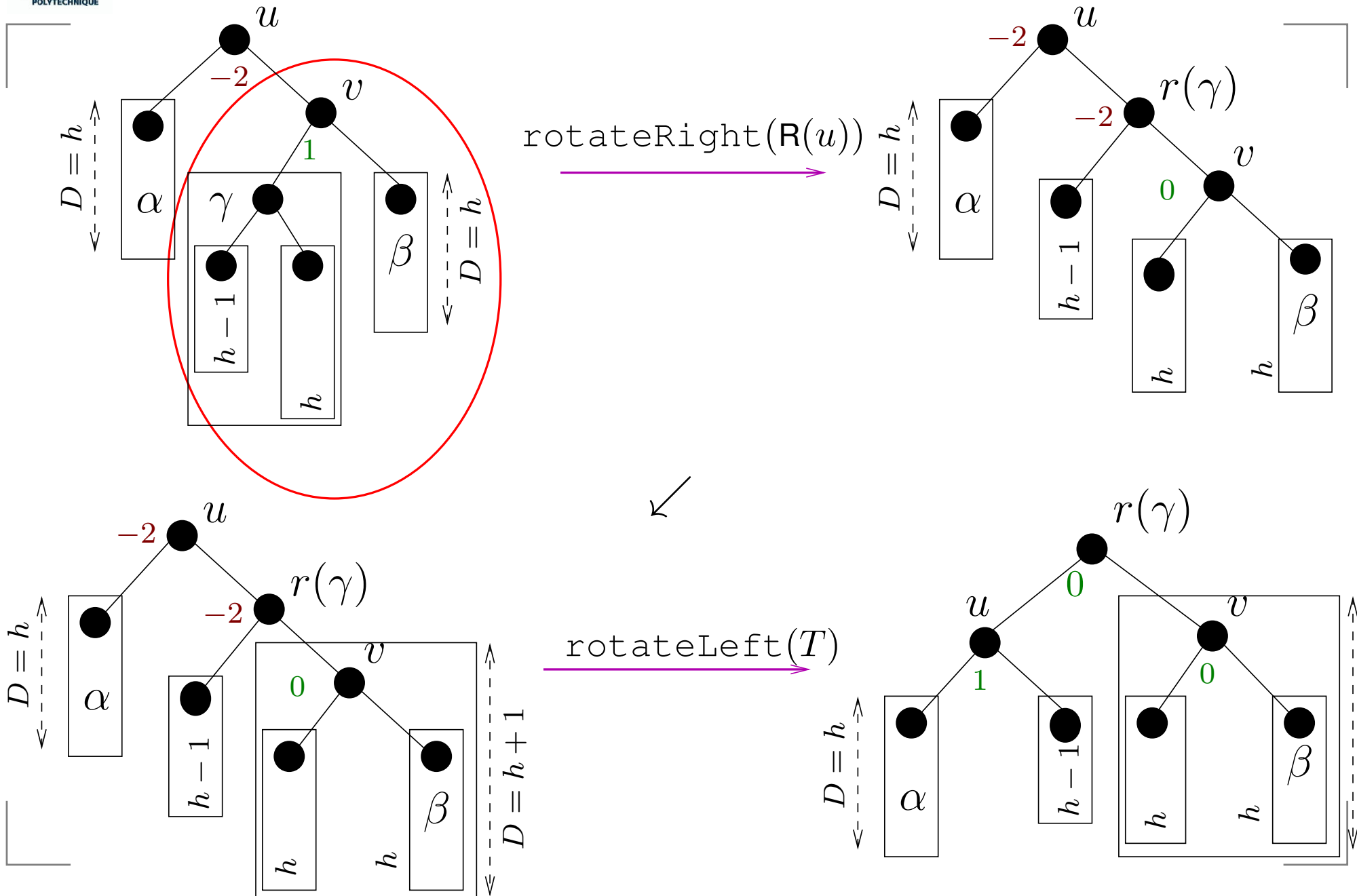
Rotate $\mathbf{R}(u)$ right

Finally, rotate left

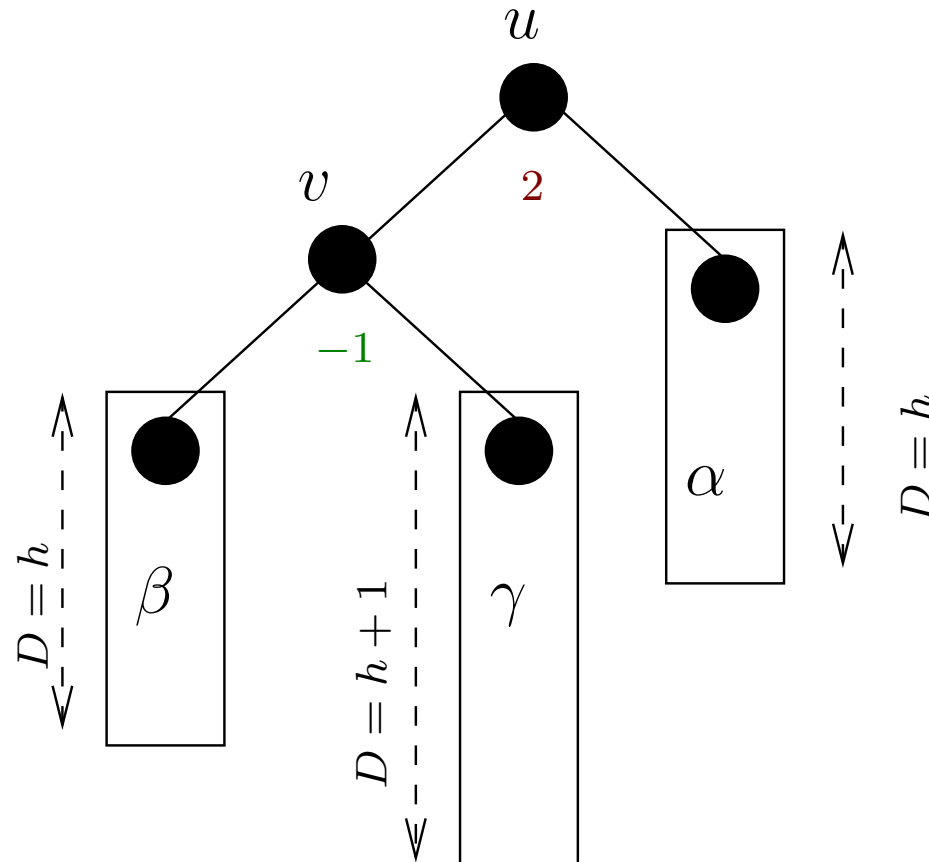


Rotate T left

Symmetric cases I



Symmetric cases II



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

Implementation of AVL trees

It took me TEN bloody hours to code a decent Java implementation!

● **Definition of “decent implementation”:**

In view of my coding odyssey, on the above choices I give you:

● **Advice:**

Implementation of AVL trees

It took me TEN bloody hours to code a decent Java implementation!

- **Definition of “decent implementation”:**
- Recursive implementation for didactical value

In view of my coding odyssey, on the above choices I give you:

- **Advice:**
- Consider iterative implementations using stacks or three threading

Implementation of AVL trees

It took me TEN bloody hours to code a decent Java implementation!

- **Definition of “decent implementation”:**
- Recursive implementation for didactical value
- Methods act on `this` node, for consistency with other lectures

In view of my coding odyssey, on the above choices I give you:

- **Advice:**
- Consider iterative implementations using stacks or three threading
- Declare static methods and pass the relevant nodes as arguments

this frees you from several constraints, e.g. you can't set `this` to `null`

Implementation of AVL trees

It took me TEN bloody hours to code a decent Java implementation!

- **Definition of “decent implementation”:**
- Recursive implementation for didactical value
- Methods act on `this` node, for consistency with other lectures
- Efficient update of $B(v)$ after insertions and rotations

In view of my coding odyssey, on the above choices I give you:

- **Advice:**
- Consider iterative implementations using stacks or three threading
- Declare static methods and pass the relevant nodes as arguments
this frees you from several constraints, e.g. you can't set `this` to `null`
- If you have trouble keeping balances updated in an efficient manner, you can always re-compute them recursively at each node, using depth
yields a slower code but worst-case complexity is the same

Implementation of AVL trees

It took me TEN bloody hours to code a decent Java implementation!

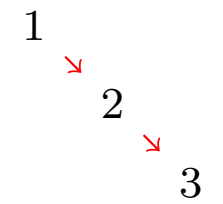
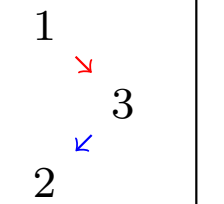
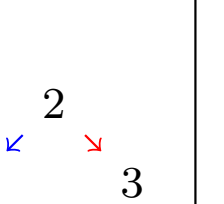
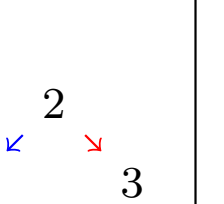
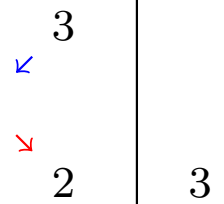
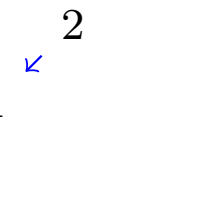
- **Definition of “decent implementation”:**
- Recursive implementation for didactical value
- Methods act on `this` node, for consistency with other lectures
- Efficient update of $B(v)$ after insertions and rotations

In view of my coding odyssey, on the above choices I give you:

- **Advice:**
- Consider iterative implementations using stacks or three threading
- Declare static methods and pass the relevant nodes as arguments
this frees you from several constraints, e.g. you can't set `this` to `null`
- If you have trouble keeping balances updated in an efficient manner, you can always re-compute them recursively at each node, using depth
yields a slower code but worst-case complexity is the same
- Look at my (online) code and INF421 Polycopié's

Balanced vs. random BST

- Balanced binary trees have $O(\log n)$ insert, delete, query ops
- What about an average (not necessarily balanced) BST?
- Given a sequence $\sigma \in \{1, \dots, n\}^n$, we insert it in a BST T
- Nodes to the left of $r(T)$ are $\leq r(T)$, nodes to the right of are $> r(T)$
- Let K be the number of nodes in $L(T)$, so that $|R(T)| = n - 1 - K$
- Uniform distribution on K i.e. $P(K = k) = \frac{1}{n}$ for all $k \in \{0, \dots, n - 1\}$

σ	(1,2,3)	(1,3,2)	(2,1,3)	(2,3,1)	(3,1,2)	(3,2,1)
T						
type	A	B	C	C	D	E

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

Average depth and path length

- Average depth for BFSs: $O(\log n)$ [Devroye, 1986]
- Average path length for BFSs: $O(n \log n)$ [Vitter & Flajolet, 1990]
- This shows that BFSs are pretty balanced on average

Heaps and priority queues

Queues reminder

- A **queue** is a data structure with main operations:
 - `pushBack(v)`: inserts v at the end of the queue
 - `popFront()`: returns and removes an element at the beginning of the queue
- Queues implement the Last-In-First-Out principle
- Definitions in Lecture 2
- Used by BFS to compute paths with fewest arcs
- If arcs are prioritized (e.g. travelling times for route segments), we want the queue to return the element of highest priority

This may not be at the beginning of the queue

Priority queues

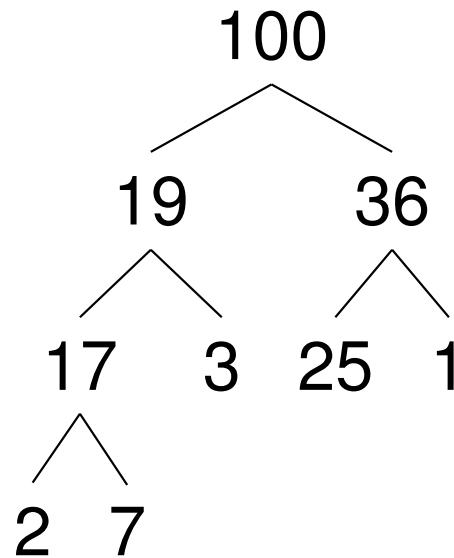
- Let V be a set and $(S, <)$ be a totally ordered set
- A **priority queue** on V, S is a set Q of pairs (v, p_v) s.t. $v \in V$ and $p_v \in S$
- Usually, p_v is a number
- E.g., if p_v is the rank of entrance of v in Q , then Q is a standard queue
- Supports three main operations:
 - `insert(v, p_v)`: 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
- The $O(1)$ is obtained by storing the maximum priority element as the root of a binary tree
- Distinguishing properties
 - *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

This 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 definition $L(v) \leq VR(v)$

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)

∅

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

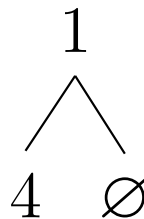
1

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

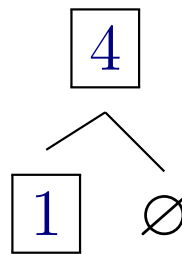


Insert

- 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)

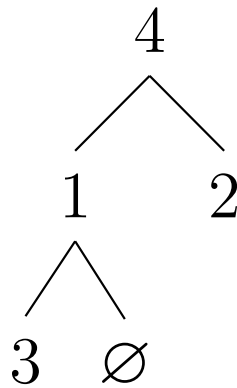


Insert

- 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)

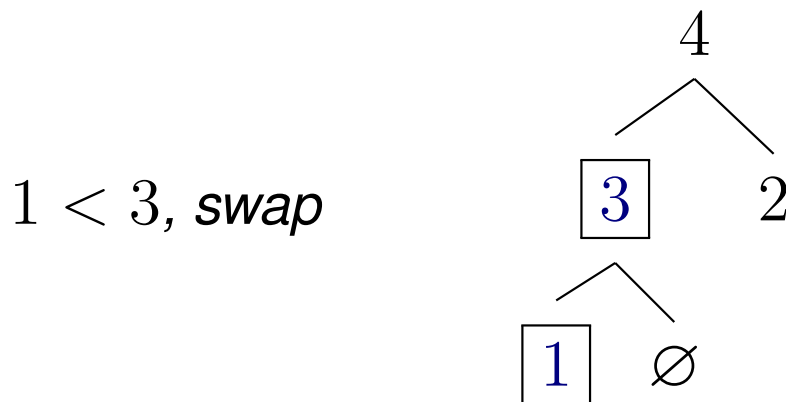
insert 3



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)

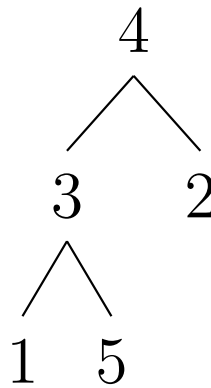


Insert

- 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)

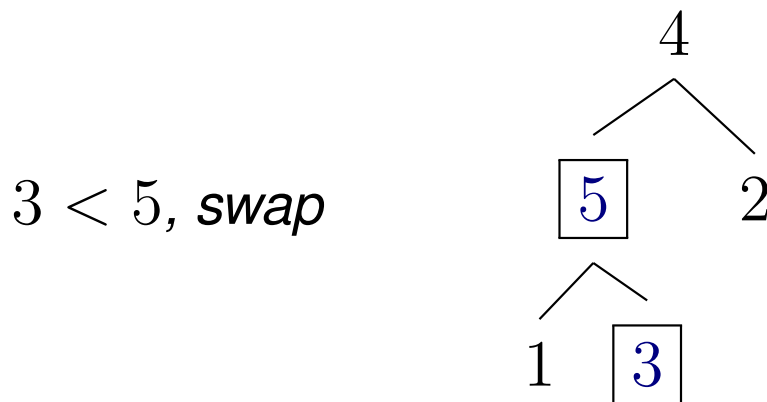
insert 5



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

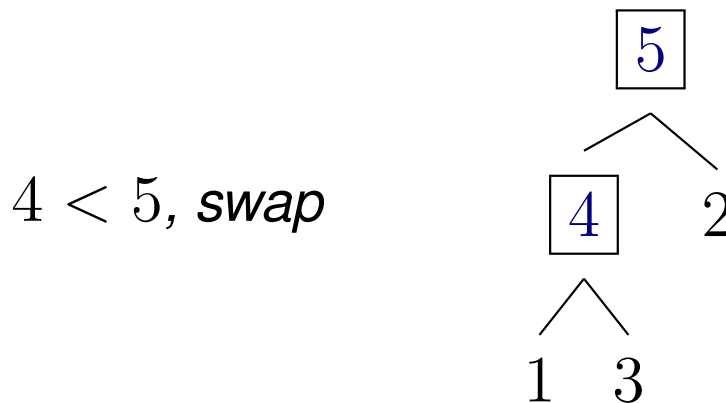
Example: insert (1, 4, 2, 3, 5)



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)



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

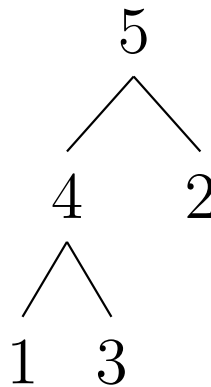
Max

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

Removal of max

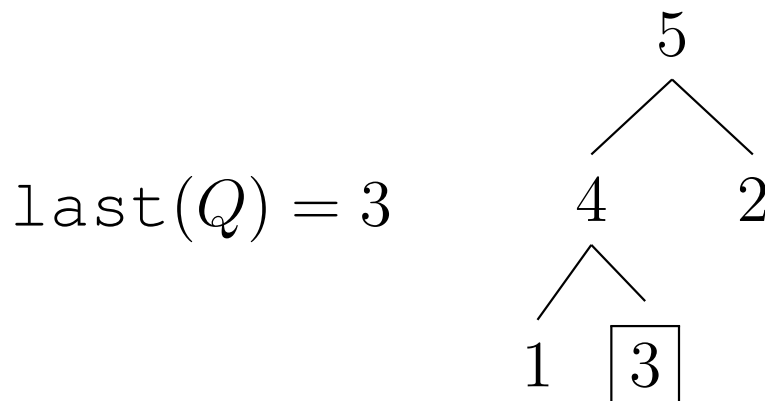
- Let $\text{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\text{last}(Q)$ to the root $r(Q)$
- Compare v with its children u, w : if $p_v \geq p_u, p_v \geq p_w$, heap is in correct order
- Otherwise, swap v with $\max_p(u, v)$ (use \min_p if min-heap) and repeat comparison/swap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

original tree



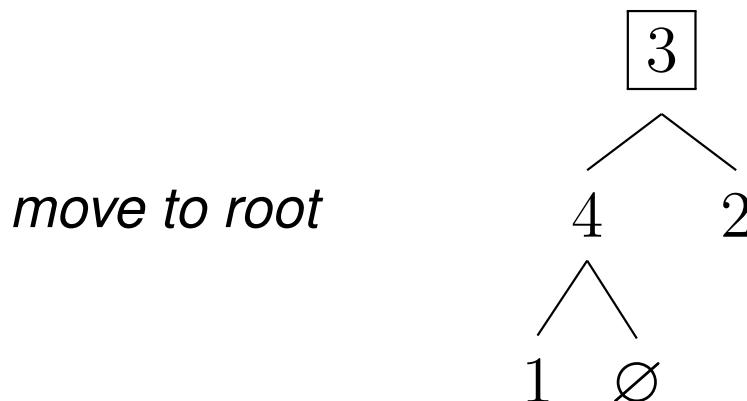
Removal of max

- Let $\text{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\text{last}(Q)$ 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



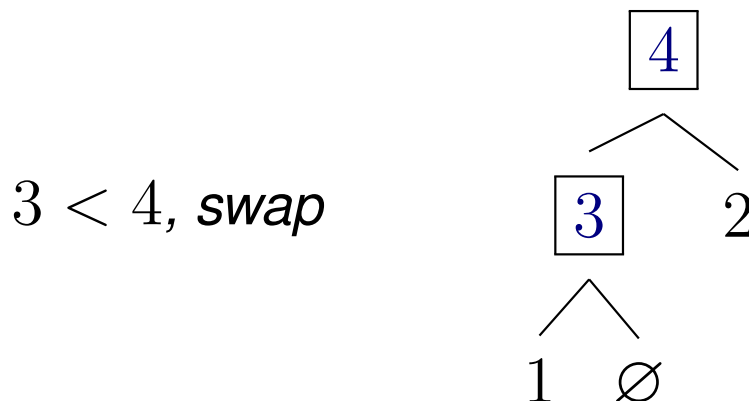
Removal of max

- Let $\text{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\text{last}(Q)$ 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



Removal of max

- Let $\text{last}(Q)$ be the rightmost non-empty element of the last heap level
- Move node $\text{last}(Q)$ 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



Efficient construction

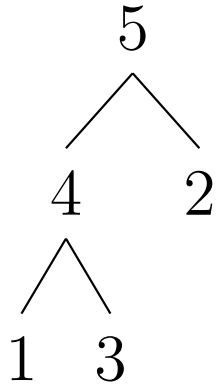
- Suppose we have n elements of V to insert in an empty heap
- Trivially: each insert takes $O(\log n)$, get $O(n \log n)$ to construct the whole heap
- Instead:
 1. arbitrarily put the element in a binary tree with the shape property (can do this in $O(n)$)
 2. lower level first, move nodes down using 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

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

Implementation

- A priority queue is implemented as a heap
- But we didn't say how a heap is to be implemented
- It *behaves* like a tree
- We're going to use an array instead (practically very efficient)

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 $q_{2i+1} = r(L(v))$ and $q_{2i+2} = r(R(v))$
(whenever $2i + 1, 2i + 2 < n$)

- **Parent**

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

We now have all the elements: start implementing!

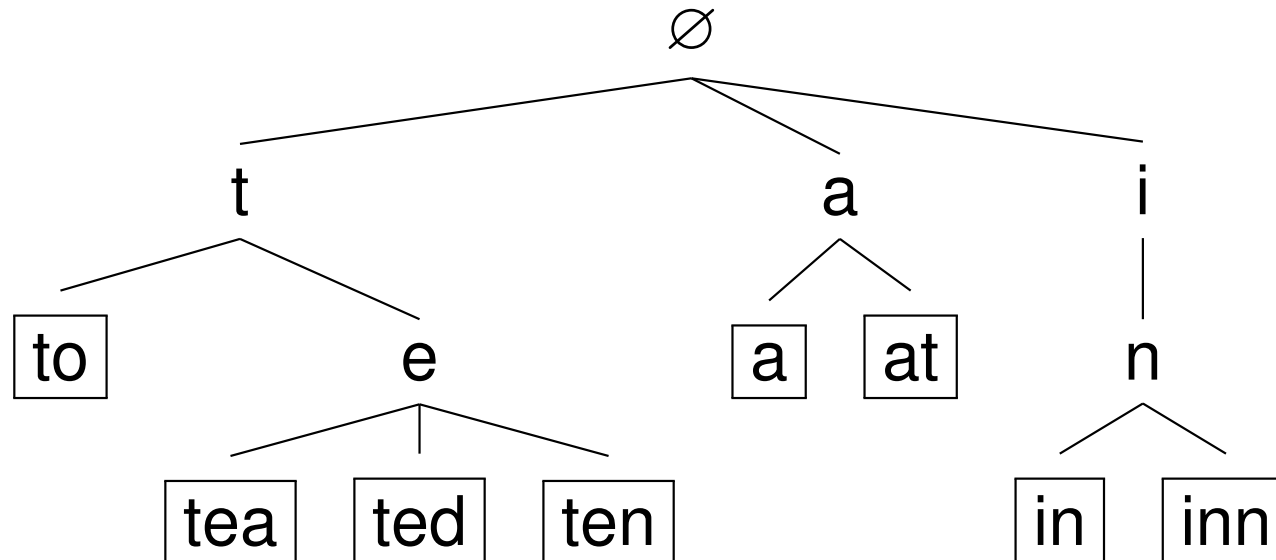
k -ary Search Trees

Tries

- Recall SEARCH problem: given a set V and a key v , determine whether $v \in V$
- Hash functions: $O(1)$ in the average case
- Let V be a set of words from same alphabet L
- We can organize keys in a k -ary tree for answering SEARCH
- In a k -ary tree, each node has at most k subnodes

Trie example

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



- 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 \emptyset , the empty string

Trie properties

- The path of the trie corresponding to a key k is given by the key itself
Compare with hash functions: the hash value is specified by the key
- This path has length m or $m + 1$ (when a key is also a prefix), in any case $O(m)$
- `find`, `insert` and `delete` take worst-case $O(m)$
- If $m, |L|$ are bounded by a constant w.r.t. $n = |V|$, then methods are $O(1)$ in the worst case (w.r.t. set size)
- 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