

CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #10

►AVL Trees: a class of self-balancing trees

Prof. Pietro S. Oliveto

Department of Computer Science and Engineering
Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn

<https://faculty.sustech.edu.cn/olivetop>

Reading: Lecture notes

► Aims of this lecture

- To see a class of **self-balancing trees** guaranteeing operations in time $O(\log n)$.
- To show that the depth of AVL trees is $O(\log n)$.
- To show how to perform insertions and deletions, **rebalancing the tree through rotations** whenever it becomes unbalanced.

► Self-balancing trees

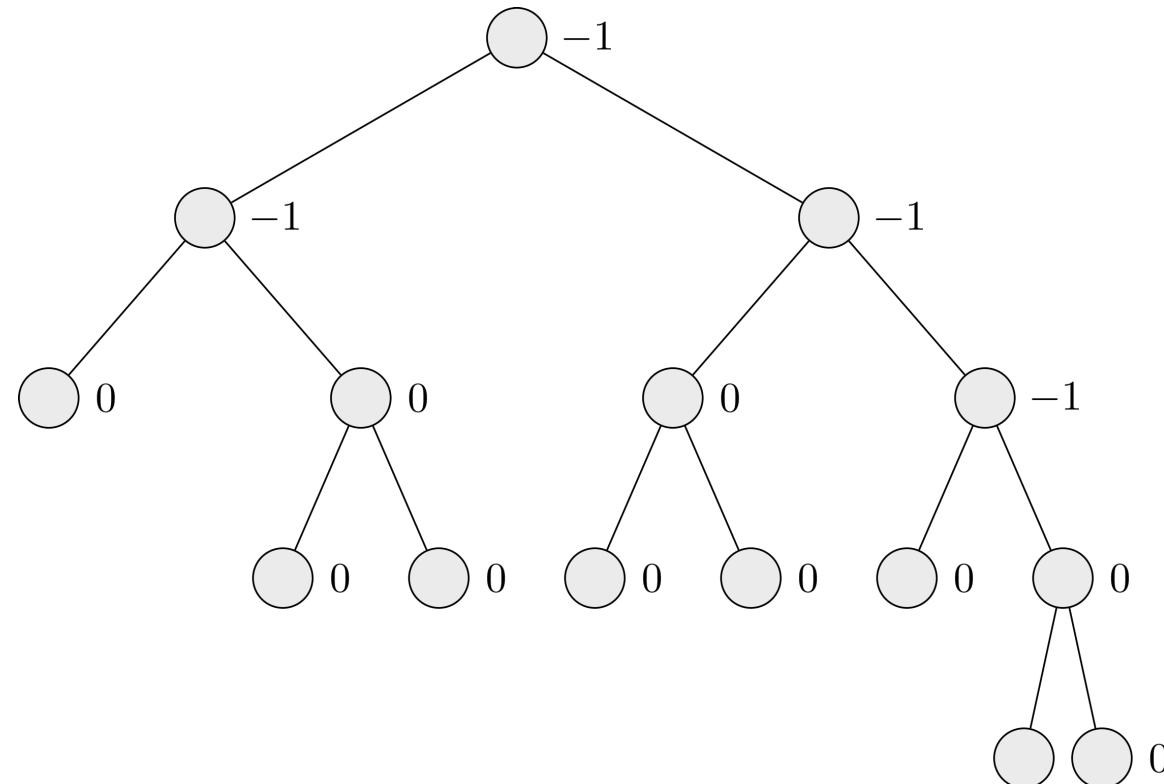
- There are various types of binary search trees that are guaranteed to have depth $O(\log n)$.
 - AVL Trees
 - 2-3 Trees
 - B-Trees
 - Red-black Trees
 - Splay Trees
 - Van Emde Boas Trees
 - ...

► AVL Trees

- Invented by and named after Adelson-Velskii and Landis.
- **Invariant:** all nodes are **locally balanced**.
- A binary tree is called AVL tree if for every node the following holds: the height of the left subtree and the height of the right subtree only differ by at most 1.
- Let v be a node and T_l, T_r be its left and right subtrees, respectively. Then $bal(v) := h(T_l) - h(T_r)$ is the **balance factor** of v , $h()$ denoting the height of a tree.
- In an AVL tree hence for every node v we have $bal(v) \in \{-1, 0, +1\}$.

► Balance properties

- The local property does **not** mean that all leaves are on two levels. AVL trees can be lopsided, see this example:



- However, overall the tree is still pretty balanced.

► Estimating the depth of an AVL tree

Theorem: the height of an AVL tree with n nodes is at most

$$h \leq \frac{1}{\log((\sqrt{5} + 1)/2)} \log n \approx 1.44 \log n.$$

- This is only up to 44% deeper than a perfectly balanced tree.

Proof outline:

- Consider the minimum number of nodes in any AVL tree of height h and call it $A(h)$.
 - This means that any AVL tree of height h will have $n \geq A(h)$ nodes.
- Show that $A(h)$ (and thus n) is exponentially large in h .
 - Will show that $A(h)$ is similar to Fibonacci numbers.
- Take logarithms (+maths) to get the claimed bound.

► Minimum number of nodes in an AVL tree

- Let $A(h)$ be the minimum number of nodes in any AVL tree of height h .
 - An AVL tree with height 0 consists of the root only, hence $A(0) = 1$.
 - The smallest AVL tree of height 1 has two nodes, hence $A(1) = 2$.
 - An AVL tree of height h has to have a root with one subtree of height $h - 1$, and the other subtree of height at least $h - 2$.
Hence $A(h) = 1 + A(h - 1) + A(h - 2)$.
- This is similar to the Fibonacci numbers (bar the “1 +”):
 - $Fib(0) = Fib(1) = 1$ and
 - $Fib(h) = Fib(h - 1) + Fib(h - 2)$.
 - Handy closed form:

$$Fib(k) \geq \frac{1}{\sqrt{5}} \left[\left(\frac{\sqrt{5} + 1}{2} \right)^{k+1} - 1 \right]$$

► Link to Fibonacci numbers

- We prove by induction that $A(h) = Fib(h + 2) - 1$.
- Base case: $A(0) = 1 = 2 - 1 = Fib(2) - 1$
and $A(1) = 2 = 3 - 1 = Fib(3) - 1$.
- Assume that the claim holds for $A(h - 1)$ and $A(h - 2)$, then

$$\begin{aligned} A(h) &= 1 + A(h - 1) + A(h - 2) && \text{(by recurrence)} \\ &= 1 + Fib(h + 1) - 1 + Fib(h) - 1 && \text{(2x induction hypothesis)} \\ &= Fib(h + 1) + Fib(h) - 1 \\ &= Fib(h + 2) - 1 && \text{(by definition of } Fib(h + 2)\text{).} \end{aligned}$$

- Every AVL tree with n nodes and height h has
 $n \geq A(h) \geq Fib(h + 2) - 1$.
- Plugging in closed form for Fib gives $\left(\frac{\sqrt{5}+1}{2}\right)^{h+3} \leq \sqrt{5}n + \sqrt{5} + 1$
- Taking logarithm of base $\frac{\sqrt{5}+1}{2}$:
$$h + 3 \leq \log_{(\sqrt{5}+1)/2}(\sqrt{5}n + \sqrt{5} + 1)$$

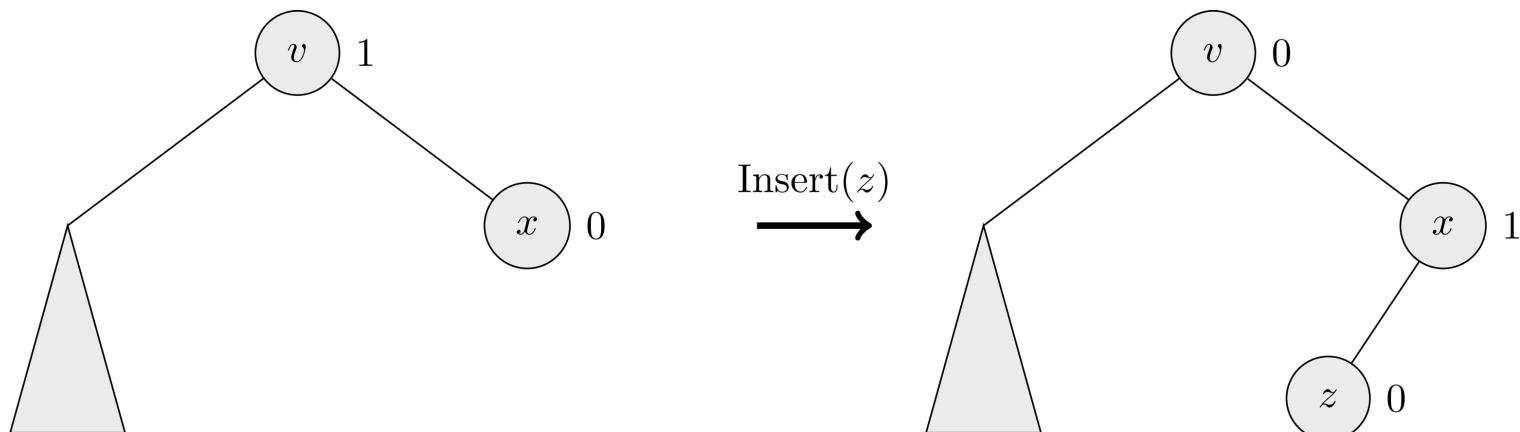
$$\Rightarrow h \leq \log_{(\sqrt{5}+1)/2}(n)$$
- Converting to \log_2 completes proof.

► **Search in an AVL Tree**

- Works like in an ordinary binary search tree.

▶ Inserting in an AVL Tree

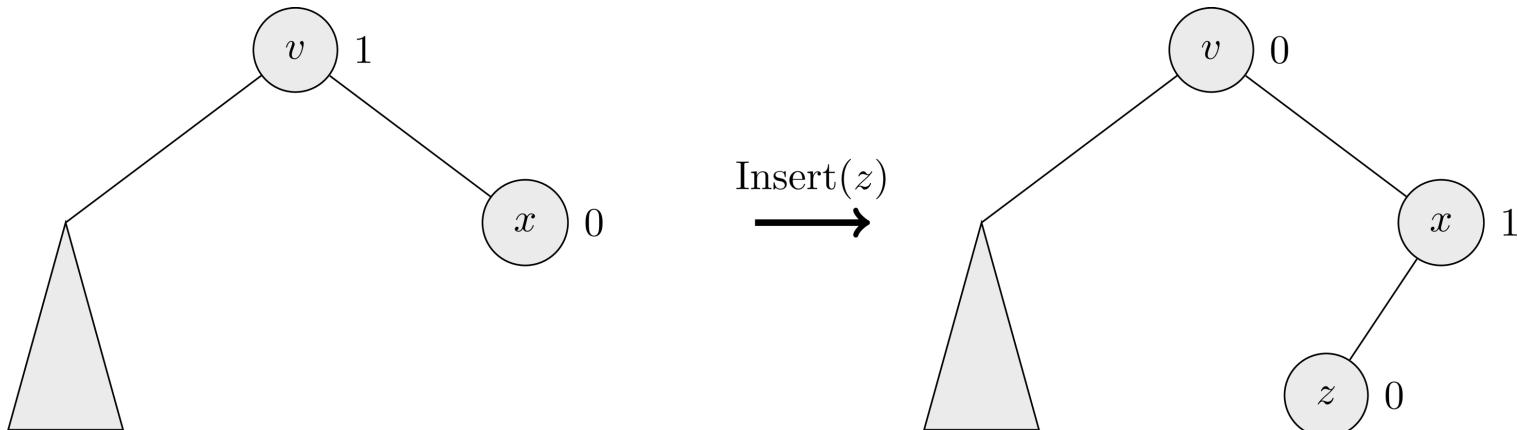
- Works like in an ordinary binary search tree.
- But the tree may become unbalanced, hence we need to **rebalance**. (We focus on ideas here: code is lab exercise)
- We record the **search path** to new element z , and then **work back up the search path** to rebalance **so long as the height of the current subtree has increased**.
- Let v be the current node and its **right child x** be on the search path (left child is symmetric) -> start at $v = z.parent$



► Insert (1)

Case 1: $bal(v) = 1$.

- Left subtree of v was higher than right subtree before insertion.
- After inserting z , the right subtree has increased its height, hence the subtree at v is now balanced. We set $bal(v) = 0$
- The height of v has not changed, hence rebalancing is **done**.



► Insert (2)

Case 2: $\text{bal}(v) = 0$.

- Both subtrees of v were balanced before insertion.
- After inserting z , the right subtree has increased its height, hence now $\text{bal}(v) = -1$.
- The height of the subtree at v has **increased** (we cannot stop), hence we need to continue rebalancing at v 's parent to check for imbalances further up the tree.
- **If** v was the root, we stop: **done**

► Insert (3)

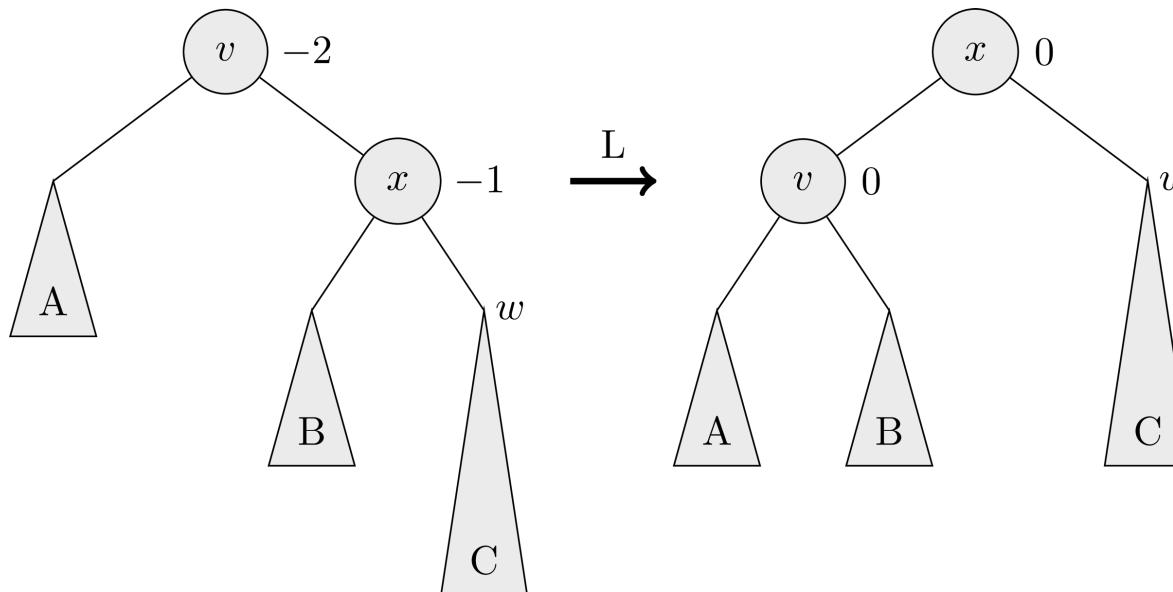
Case 3: $bal(v) = -1$.

- After insertion, the tree has become unbalanced: $bal(v) = -2 \rightarrow$ we need to fix this!
- Search path contains nodes v, x, w whose subtrees **increased in height**.
- We distinguish two sub-cases, depending on whether w is the right child or the left child of x .

► Insert (4)

Sub-case 3-1: w is the right child of x.

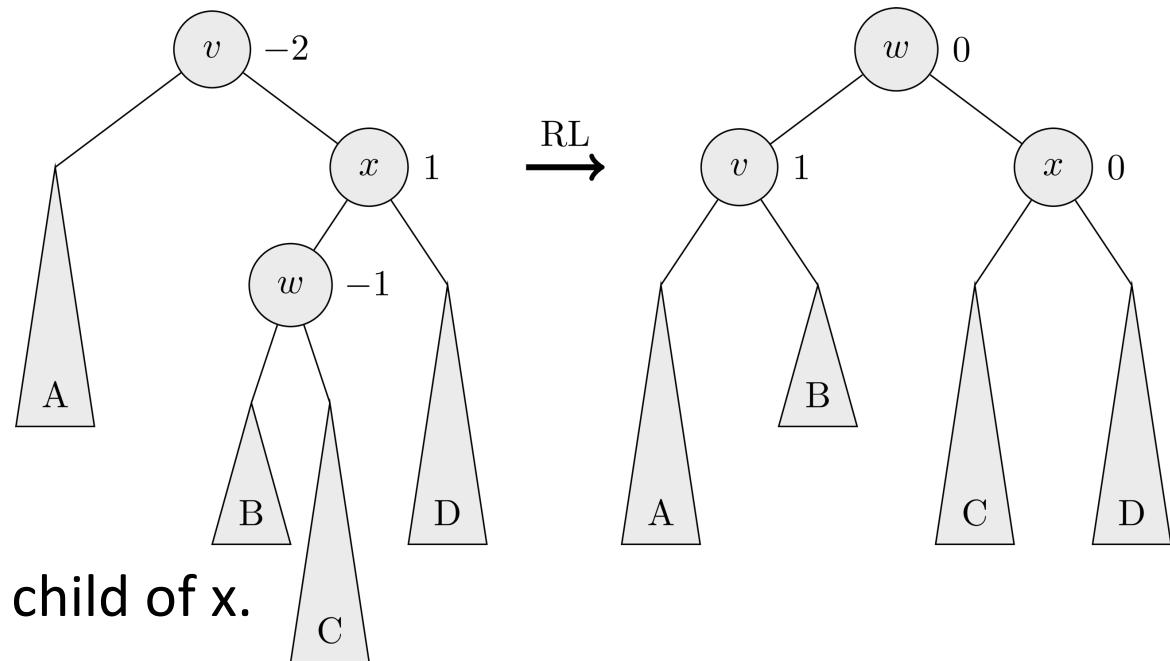
- The tree is lopsided because of an “outside” problem.
- Now **rotate** the tree to the left: x becomes the parent of v, and x's left subtree B becomes a subtree of v. -> $\text{bal}(x) = \text{bal}(v) = 0$



- Height of whole subtree is the same as before insert. **Done.**

► Insert (5)

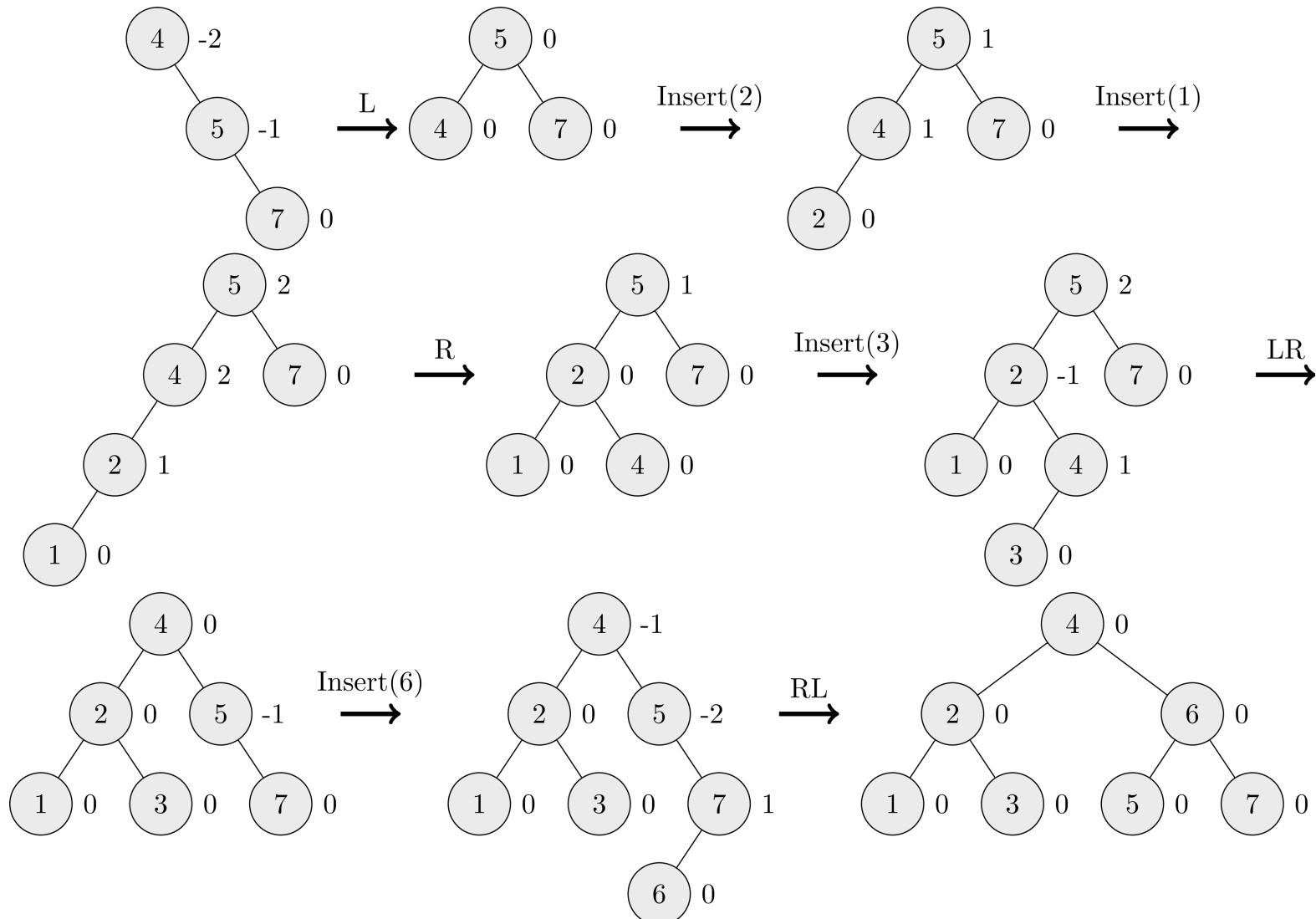
NB: heights of B and C could be the other way round.



Sub-case 3-2: w is the left child of x.

- The tree is lopsided because of an “inside” problem.
- Now need a **double rotation** to rebalance the tree: a right rotation at x, followed by an immediate left rotation at v.
- $bal(w) = 0$** ; If after the insertion:
 - $bal(w)$ was $-1 \Rightarrow bal(v) = 1, bal(x) = 0;$
 - $bal(w)$ was $1 \Rightarrow bal(v)=0, bal(x) = -1;$
 - $bal(w)$ was $0 (w = z) \Rightarrow bal(v)=0, bal(x) = 0;$
(last case: A,B,C,D all empty)

► Insert: Example



► Insert Rebalancing: Summary

- We go from $v=z.parent$ up the tree until we find a node v with $\text{bal}(v)=1$ coming from right child (-1 coming from left): **Stop** or $\text{bal}(v)=-1$ coming from right (+1 coming from left): **Rotate and Stop**
- If $\text{bal}(v)=0$ **set to -1** if coming from right child (to +1 if coming from left), and **iterate** unless v is root.
- Rotation:
 - If $x=v.right \& w=x.right$ then **L Rotation** ($x=v.left \& w=x.left \Rightarrow R$ rot)
 - If $x=v.right \& w=x.left$ the **RL Rotation** ($x=v.left \& w=x.right \Rightarrow LR$ rot)

► Runtime of Insert

- Inserting an element takes time $O(h)$.
- Rebalancing:
 - finishes with the first rotation/double rotation.
 - All rotations (L/R/LR/RL) take time $O(1)$.
 - Backing up the search path takes time $O(1)$ for each node on the search path, hence time $O(h)$ overall.
 - This includes the time to update balance factors.
- Total runtime of Insert: $O(h) = O(\log n)$.

► Deleting in an AVL Tree

- Like for Insert, we work backwards up the search path to rebalance so long as the height of the current subtree has decreased.
- Assume without loss of generality that delete decreased the height of the *left* subtree.
- **Case 1:** $\text{bal}(v) = 1$. Here deletion decreased the height of the higher subtree, leading to $\text{bal}(v) = 0$. However, the height of v has decreased, so we need to **iterate** the rebalance procedure with v 's parent.
- **Case 2:** $\text{bal}(v) = 0$. Then we update $\text{bal}(v) = -1$ and note that the height of v 's subtree has **not** decreased, so the rebalancing is complete.

► Delete (2)

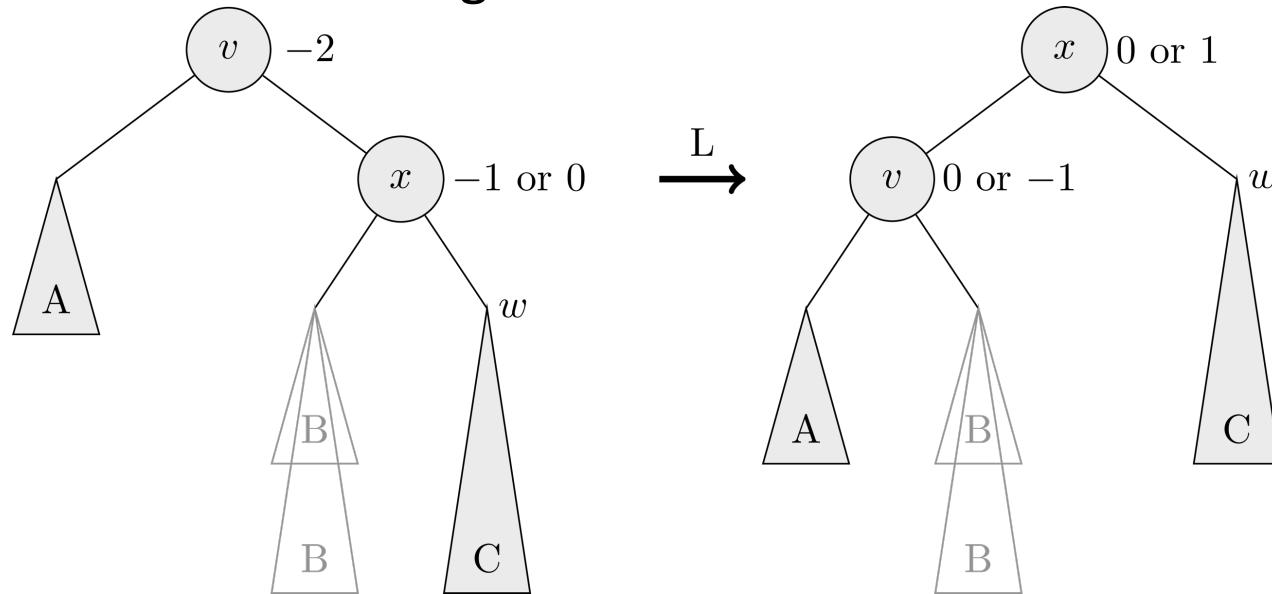
Case 3: $bal(v) = -1$.

- After deletion, the shallower subtree has become even more shallow: $bal(v) = -2$.
- Consider path of nodes v, x, w whose subtrees are now too high.
- We distinguish two sub-cases, depending on whether w is the right child or the left child of x .

► Delete (3)

Sub-case 3-1: $\text{bal}(x) \in \{-1, 0\}$.

- The tree is lopsided because of an “outside” problem.
- Now **rotate** the tree to the left.
- Two possibilities for the height of B.

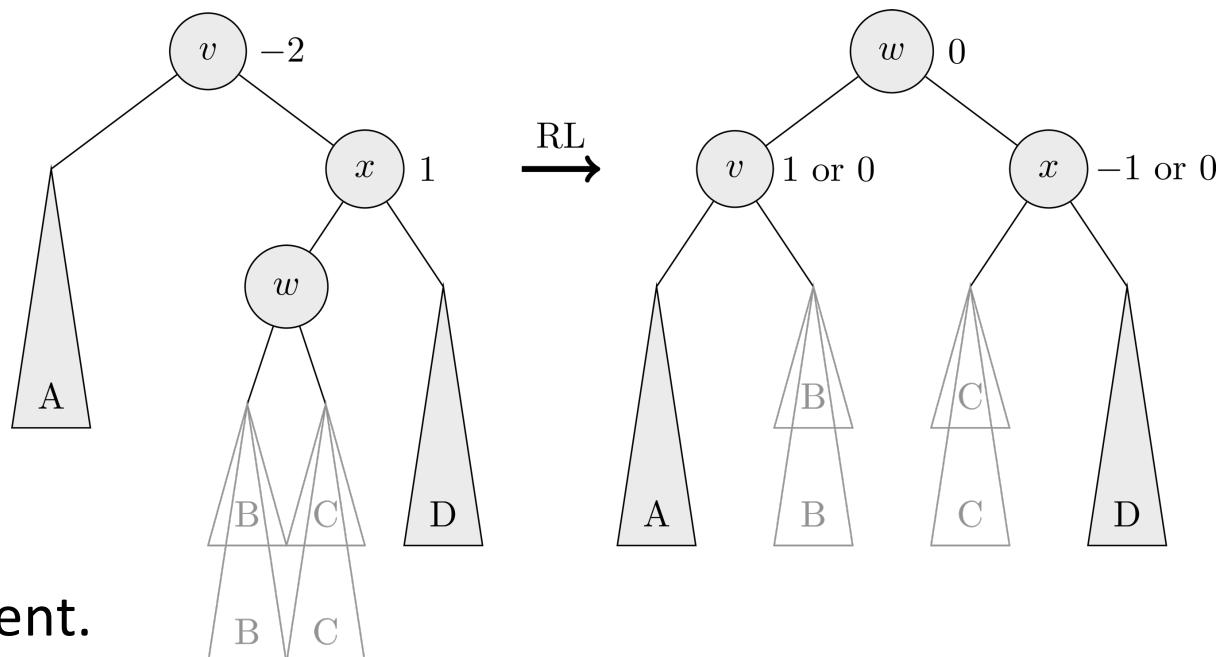


- If B was high ($\text{bal}(x)$ was 0), $\Rightarrow \text{bal}(v) = -1$, $\text{bal}(x) = 0$ & we’re **done**.
- Otherwise, height of the subtree **decreased**, ($\text{bal}(x)$ was -1),
 $\Rightarrow \text{bal}(v) = 0$, $\text{bal}(x) = 0$ & **iterate** at x ’s parent.

► Delete (4)

Sub-case 3-2: $bal(x) = 1$.

- The tree is lopsided because of an “inside” problem.
- Again need a **double rotation**.
- B and C can have one of two heights; one must be high.



- Continue at parent.

► Runtime of Delete

- Delete may not finish with the first rotation/double rotation.
- Still, the time spent at each node on the search path is $O(1)$, so we still get a time of $O(h) = O(\log n)$.

► Summary

- AVL trees with n elements have height $O(\log n)$.
- AVL trees with n nodes execute the following operations in time $O(\log n)$
 - **Searching, Minimum, Maximum, Successor**
 - Follows since AVL trees are binary search trees whose height is always $h = O(\log n)$.
 - **Insertion**
 - **Deletion**
- Greater efficiency from a simple idea: rotating nodes.