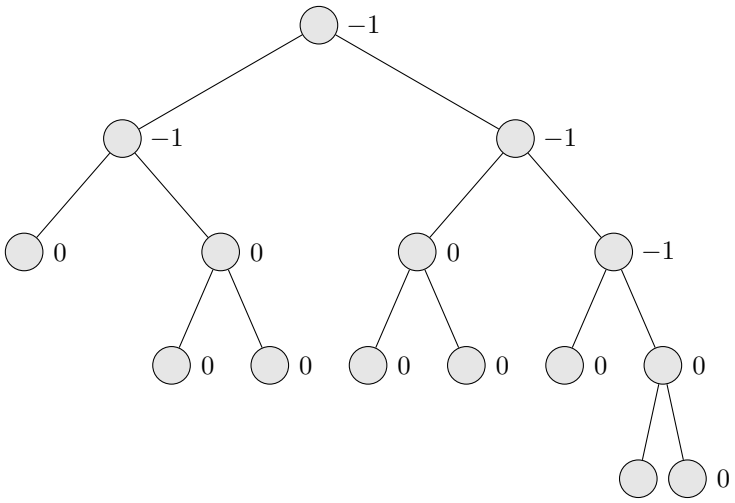
AVL Trees: a Class of Self-Balancing Binary Search Trees

CS-217 Lecture Notes

AVL trees were invented by Adelson-Velskii and Landis [1]; their idea was to use rotations to re-balance trees once they become unbalanced.

Definition 1. *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* $\mathrm{bal}(v) := h(T_\ell) - h(T_r)$ *is the balance factor of v, h denoting the height of a tree. In an AVL tree hence for every node v we have* $\mathrm{bal}(v) \in \{-1, 0, +1\}$.

This local property implies that AVL trees are nicely balanced overall. However, it does *not* mean that all leaves come to lie on two levels. AVL trees can be moderately lopsided, as shown in the following AVL tree, with balance factors listed next to each node:



How to estimate the height of an AVL tree? It's not obvious how to find a direct argument. Instead, we will argue (like we did when estimating the height of a heap in HeapSort) that an AVL tree with a certain height has many nodes in it. This will then give the desired relation between height and size.

Theorem 2. *The height of an AVL tree with n nodes is at most*

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

Proof. Let us denote by $A(h)$ the minimum number of nodes in any AVL tree of height h. An AVL tree with height 0 consists of a root only, hence $A(0) = 1$. The smallest AVL tree of height 1 has 2 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 the minimum number of nodes is

$$A(h) = 1 + A(h-1) + A(h-2). \tag{1}$$

This is reminiscent of the Fibonacci numbers, defined as $\text{Fib}(0) = \text{Fib}(1) = 1$ and $\text{Fib}(h) = \text{Fib}(h-1) + \text{Fib}(h-2)$ for $h \geq 2$.

We prove by induction that

$$A(h) = \text{Fib}(h+2) - 1$$

For the base case, $A(0) = 1 = 2 - 1 = \text{Fib}(2) - 1$ and $A(1) = 2 = 3 - 1 = \text{Fib}(3) - 1$. Assume that the claim holds for all values $0 \leq x < h$. In the induction step we now show that then the claim holds for h.

$$\begin{aligned}
A(h) &= 1 + A(h-1) + A(h-2) && \text{(by Equation (1))} \\
&= 1 + \text{Fib}(h+1) - 1 + \text{Fib}(h) - 1 && \text{(applying the induction hypothesis to } A(h-1) \text{ and } A(h-2)) \\
&= \text{Fib}(h+1) + \text{Fib}(h) - 1 \\
&= \text{Fib}(h+2) - 1 && \text{(by definition of Fib}(h+2)).
\end{aligned}$$

We will use the following inequality for $\text{Fib}(k)$:

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

So, for an AVL tree with n nodes and height h we have $n \geq A(h) \geq \text{Fib}(h+2) - 1$, hence

$$\frac{1}{\sqrt{5}} \left(\left(\frac{\sqrt{5}+1}{2} \right)^{h+3} - 1 \right) \leq n+1 \Leftrightarrow \left(\frac{\sqrt{5}+1}{2} \right)^{h+3} \leq \sqrt{5}n + \sqrt{5} + 1$$

Taking the logarithm of base $(\sqrt{5}+1)/2$ on both sides gives

$$h + 3 \leq \log_{(\sqrt{5}+1)/2}(\sqrt{5}n + \sqrt{5} + 1).$$

Along with $\log_{(\sqrt{5}+1)/2}(\sqrt{5}n + \sqrt{5} + 1) \leq \log_{(\sqrt{5}+1)/2}(n) + 3$ this implies the claim. □

Let's now look into operations on AVL trees.

Search(z)

Search works exactly like in ordinary binary search trees. The same holds for Insert and Delete, but we may need to rebalance the tree if the balance factors exceed their permitted values.

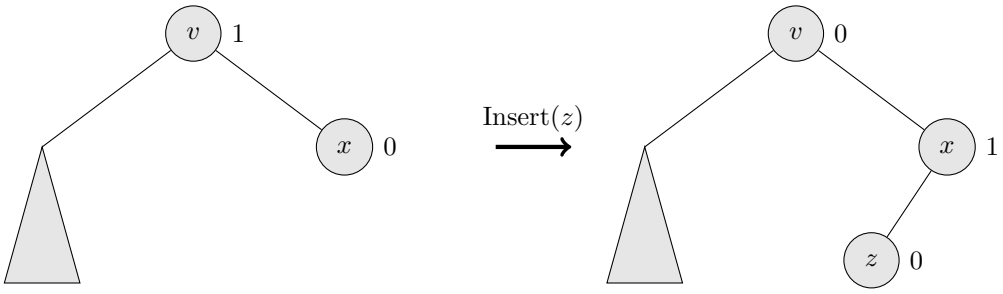
Insert(z)

When inserting element z, we record the search path from the root to z. Then we retrace the search path from z back up, rebalancing the tree where necessary. We call v the current node on the path; v starts with z.parent.

Without loss of generality, we assume that the *right* child of v is on the search path and denote this child x (the case where the *left* child of v is on the search path is symmetrical).

We distinguish the following cases based on the balance degree of v.

Case 1: bal(v) = 1. In this case, the left subtree of v was higher than the right subtree before inserting z. After inserting z, the right subtree has increased its height, hence we set bal(v) = 0 as the subtree rooted in v is now balanced. The height of said subtree has not changed, hence we terminate the rebalance procedure here. The following figure illustrates this case.

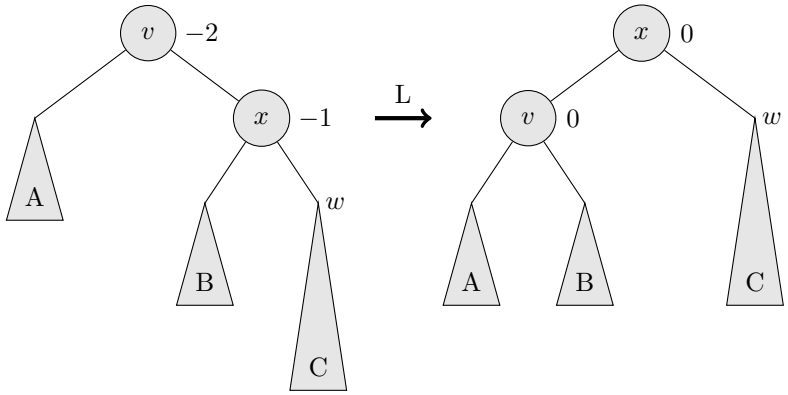


Case 2: bal(v) = 0. Here both subtrees of v had the same height before insertion. Now the height of the right subtree has increased, hence v's height has increased, and we need to rebalance the parent of v. We set bal(v) = -1 and then iterate the procedure, replacing $v = v$.parent or stopping if v is the root.

Case 3: bal(v) = -1. Here the right subtree was higher than the left, and inserting z has unbalanced the tree even more (currently bal(v) = -2 holds).

In particular, x is the root of a subtree of height at least 1. Let w be the child of x on the search path. Since the rebalance procedure has gone up to v, we know that both subtrees rooted at x and w have increased in height. We distinguish two sub-cases, depending on whether w is the right child or the left child of x.

Sub-case 3.1: w is the right child of x. Then the tree is lopsided because of an "outside" problem. We now *rotate* the tree to the left, see the figure below. In this "L"-rotation, x becomes the parent of v, and x's left subtree B becomes a subtree of v. The following figure illustrates a left rotation.

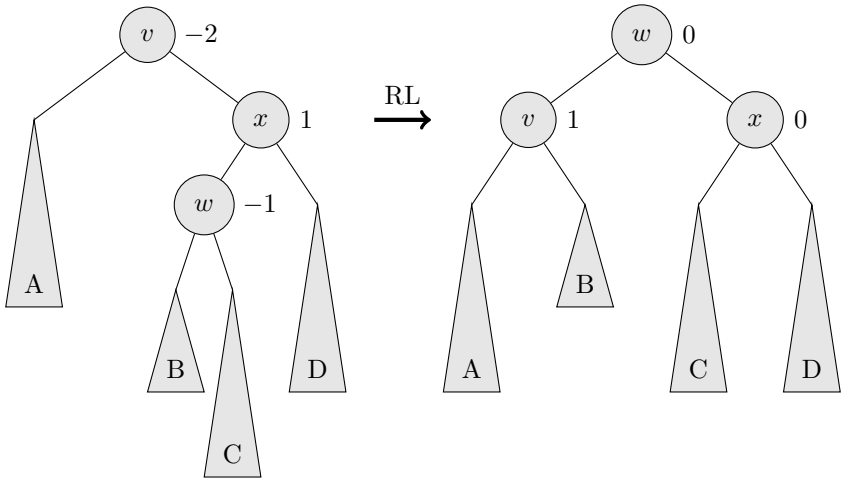


The left rotation balances the tree, such that A moves down, B remains on the same level, and C moves up. Then the deepest leaves in A, B, and C are on the same level. Hence we set bal(x) = 0 and bal(v) = 0. Since the height of the overall subtree has not grown after inserting z and rebalancing up to here, we stop the rebalance procedure.

Sub-case 3.2: w is the left child of x. Then the tree is unbalanced because of an "inside" problem: the left subtree of x is higher than its right subtree. Note that in this case a simple left rotation would not balance the tree, as it would move x's right subtree up, while keeping its left subtree on the same level.

We instead perform a *double rotation* to rebalance the tree. We can imagine this as first performing a right rotation at x, followed by an immediate left rotation at v. We refer to this as a *right-left rotation (RL).*

The first rotation turns the "inside" problem into an "outside" problem, and the second rotation proceeds as in the the previous subcase. The following figure illustrates a right-left rotation.

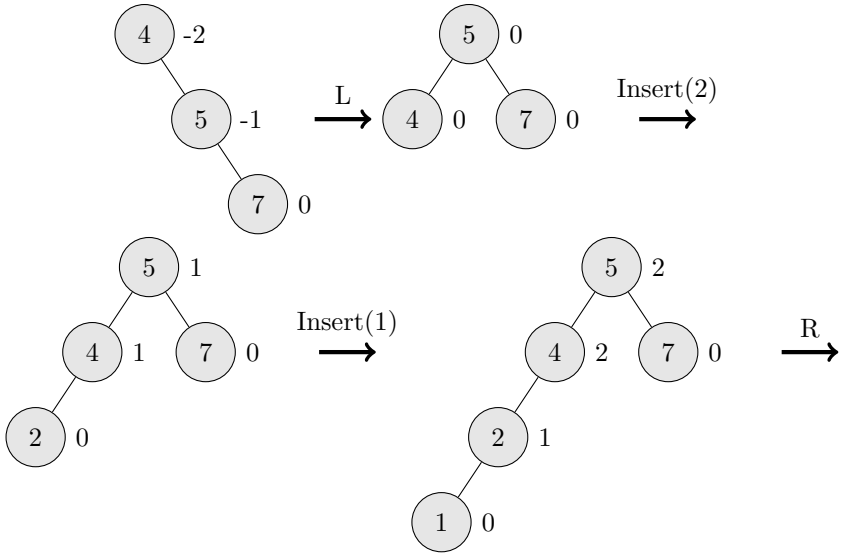


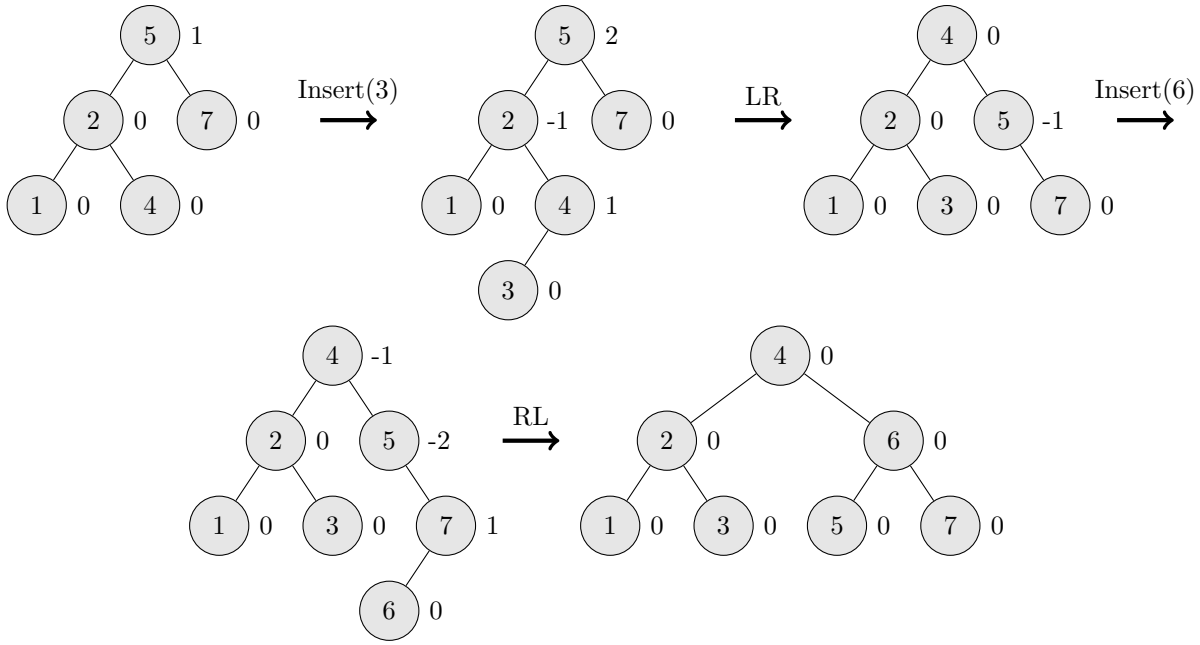
The figure assumes that the height of C is 1 larger than the height of B. It might also be that the height of B is 1 larger than the height of C, leading to a symmetric solution. Finally, there is a special case in which $w = z$ is the inserted node. Then all subtrees A, B, C, D are empty. In any case, after the double rotation the rebalancing is complete.

In all the above cases, the first rotation or double rotation completes the rebalancing procedure. It might be necessary to go up to the root, so the runtime of Insert is bounded by $O(h)$, h the height of the tree.

Example for Insertions

The following example shows how to insert the elements $4, 5, 7, 2, 1, 3, 6$ into an empty AVL tree. The example showcases all four types or rotations (L, R, RL, LR).





Delete(x)

The operation Delete is similar to Insert. We delete an element as in an ordinary binary tree, and then rebalance the tree if necessary, walking up the search path until the tree is rebalanced or the root is reached.

Consider again a node v on the search path and assume without loss of generality that an element was deleted in the *left* subtree of v, decreasing the height of that subtree (again, the other case of a *right* subtree is symmetric).

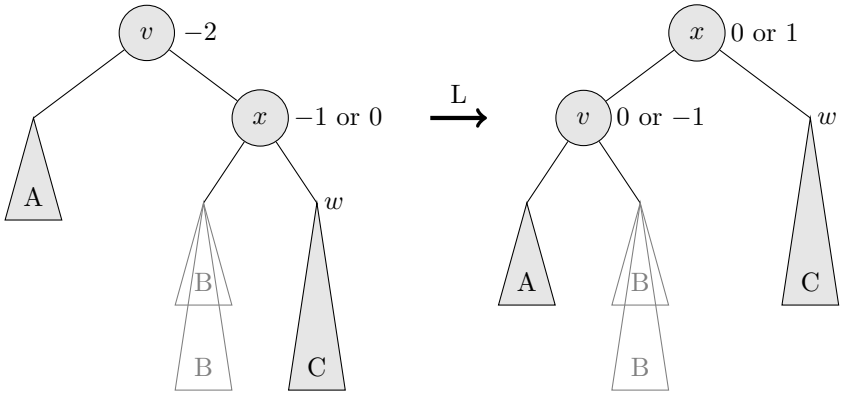
Case 1: bal(v) = 1. Here the left subtree was higher before deletion, and deletion decreased its height by 1. We set $\text{bal}(v) = 0$ as both subtrees are now balanced. However, the height of the subtree rooted in v has decreased, so we need to iterate the rebalance procedure with v's parent (unless v is the root, in which case we're done).

Case 2: bal(v) = 0. In this case both subtrees were balanced before deletion, and now the height of the left subtree has decreased. We set $\text{bal}(v) = -1$ and note that the height of v has *not* decreased, so the rebalancing is complete.

Case 3: bal(v) = −1. Here the left subtree was already shallower than the right subtree, such that the tree is now unbalanced at v (currently $\text{bal}(v) = -2$). Let x be the right child of v.

We distinguish two cases, based on whether the right subtree of x ends two levels deeper than the left subtree of v (an "outside" problem) or not (an "inside" problem).

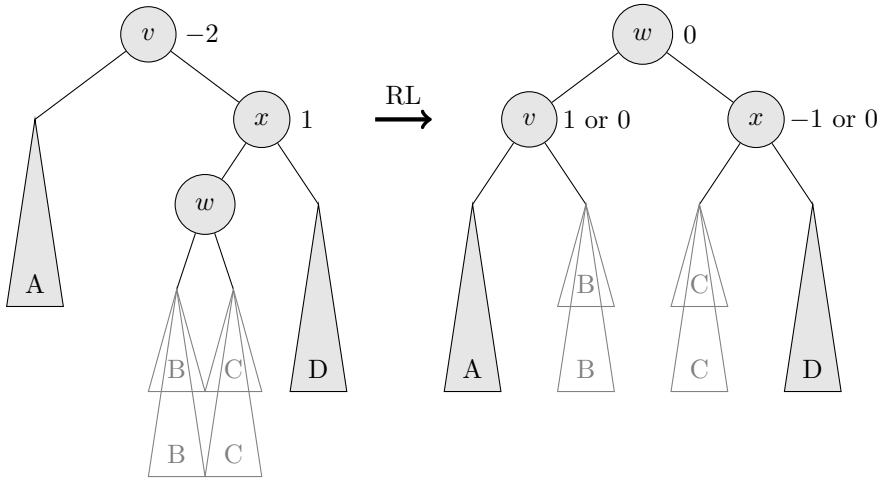
Sub-case 3.1: bal(x) ∈ {−1, 0}. Then the tree is lopsided because of an "outside" problem. We now rotate the tree to the left, see the figure below. The figure shows in gray two options for B: its height can be the same as C's ($\text{bal}(x) = 0$), or one smaller ($\text{bal}(x) = -1$).



In both cases for B, the tree is balanced. If B is shallower than C, all subtrees A, B, C now end on the same level. Then the height of the whole subtree has decreased, and we need to continue rebalancing with the parent of x. Otherwise, if B has the same height as C, the height of the whole subtree has remained the same and we are done.

Sub-case 3.2: bal(x) = 1. Then the tree is lopsided because of an "inside" problem: the left subtree of x is higher than its right subtree. We again need a double rotation to rebalance the tree.

The following figure illustrates a right-left rotation. It shows two subtrees B, C within x's left subtree: each of them may end on the same level as D, or one level deeper; however, at least one of them must end deeper than D (as bal(x) = 1).



Regardless of the height of B and C, in all cases the height of the whole subtree has decreased, hence we need to continue rebalancing with the parent of w.

In contrast to Insert, for Delete we may need several rotations or double rotations. However, the runtime is still bounded by $O(h)$, h the height of the tree. Overall, considering that the height of an n-node AVL tree is always bounded by $O(\log n)$ by Theorem 2, we proved the following theorem:

Theorem 3. *In any n-node AVL tree, the operations Search, Insert, and Delete run in time $O(\log n)$.*

References

[1] G. M. Adelson-Velskii and E. M. Landis. An information organization algorithm. *Doklady Akademia Nauk SSSR*, 146:263–266, 1962.