WS 2007/2008

Fundamental Algorithms

Dmytro Chibisov, Jens Ernst

Fakultät für Informatik
TU München

http://www14.in.tum.de/lehre/2007WS/fa-cse/

Fall Semester 2007

1. AVL Trees

As we saw in the previous section, the efficiency of standard operations on binary search trees depends on the maximum tree height. Using height balancing, we ensure that trees cannot degenerate linearly but instead have logarithmic height.

Definition 1

Let v be a vertex in a binary search tree. Then v is height balanced if and only if the heights of v's subtrees differ by at most one.
A binary search tree in which every vertex is height balanced is an AVL Tree (named after its inventors Adelson, Velskii, Landis).

1. AVL Trees

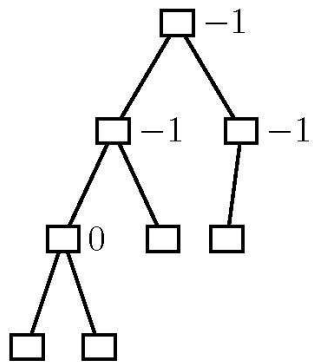
As we saw in the previous section, the efficiency of standard operations on binary search trees depends on the maximum tree height. Using height balancing, we ensure that trees cannot degenerate linearly but instead have logarithmic height.

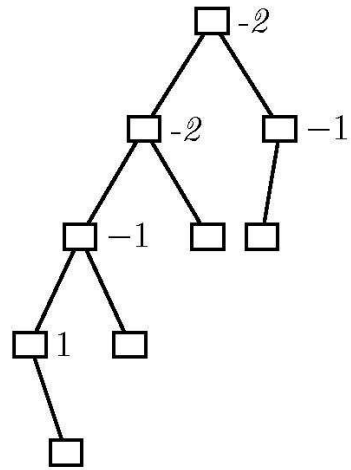
Definition 1

Let v be a vertex in a binary search tree. Then v is height balanced if and only if the heights of v's subtrees differ by at most one.
A binary search tree in which every vertex is height balanced is an AVL Tree (named after its inventors Adelson, Velskii, Landis).

Example 2

The internal vertices in the trees below are annotated with *balance values*, defined as the difference between the heights of the right and the left subtree. These values have to range between -1 and 1. Thus, the first example is an AVL tree, whereas the second is not.





Lemma 3

An AVL tree of height h has at least $fib(h+3)-1$ vertices (where $fib(k) := f_k$ is the k-th Fibonacci number) and at most $2^{h+1}-1$ vertices.

Proof

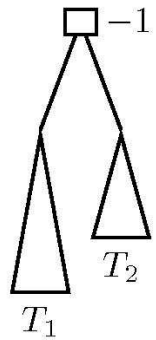
a. The upper bound is immediate: A tree of height h has $h+1$ levels, and on each level i (counting from 0), the number of vertices is at most 2^i. Thus,

$$n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1.$$

b. The lower bound can be shown by induction on h. Base case: For $h = 0, 1$, the statement is obviously true.

Proof (cont.)

Inductive step: Assuming that the statement is correct for AVL trees of height $< h$, we can conclude the following for height h:
A minimal AVL tree of height h consists of an AVL tree of height $h-1$ and an AVL tree of height $h-2$, plus one common root:



Using the induction hypothesis, we obtain
$n \geq (fib(h+1)-1)+(fib(h+2)-1)+1 = fib(h+3)-1.$ □

Corollary 4

The height of an AVL tree containing n vertices is $\Theta(\log n)$.

Proof.

This follows directly from an equality seen in the first chapter:

$$fib(k) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

implying

$$fib(k) \geq 2^{\lfloor (k-1)/2 \rfloor}.$$

□

Note that this also implies logarithmic complexity for the is_element operation. In the following subsections we shall see details on AVL tree operations.

Corollary 4

The height of an AVL tree containing n vertices is $\Theta(\log n)$.

Proof.

This follows directly from an equality seen in the first chapter:

$$fib(k) = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right)$$

implying

$$fib(k) \geq 2^{\lfloor (k-1)/2 \rfloor}.$$

□

Note that this also implies logarithmic complexity for the is_element operation. In the following subsections we shall see details on AVL tree operations.

1.1 Operations on AVL Trees

1.1.1 is_element

This operation has to be implemented precisely as specified for general binary search trees. The only difference is that, in the case of AVL trees, its logarithmic complexity is guaranteed even in the worst case.

1.1.2 insert

The first two steps of the general procedure resemble the insert operation in unbalanced binary search trees:

1.1 Operations on AVL Trees

1.1.1 is_element

This operation has to be implemented precisely as specified for general binary search trees. The only difference is that, in the case of AVL trees, its logarithmic complexity is guaranteed even in the worst case.

1.1.2 insert

The first two steps of the general procedure resemble the insert operation in unbalanced binary search trees:

1.1 Operations on AVL Trees

1.1.1 is_element

This operation has to be implemented precisely as specified for general binary search trees. The only difference is that, in the case of AVL trees, its logarithmic complexity is guaranteed even in the worst case.

1.1.2 insert

The first two steps of the general procedure resemble the insert operation in unbalanced binary search trees:

(i) The is_element operation leads to a vertex where a new leaf can be added

(ii) Attach the leaf containing the new key

(iii) Restore the height balancing, if necessary

1.1 Operations on AVL Trees

1.1.1 is_element

This operation has to be implemented precisely as specified for general binary search trees. The only difference is that, in the case of AVL trees, its logarithmic complexity is guaranteed even in the worst case.

1.1.2 insert

The first two steps of the general procedure resemble the insert operation in unbalanced binary search trees:

(i) The is_element operation leads to a vertex where a new leaf can be added

(ii) Attach the leaf containing the new key

(iii) Restore the height balancing, if necessary

1.1 Operations on AVL Trees

1.1.1 is_element

This operation has to be implemented precisely as specified for general binary search trees. The only difference is that, in the case of AVL trees, its logarithmic complexity is guaranteed even in the worst case.

1.1.2 insert

The first two steps of the general procedure resemble the insert operation in unbalanced binary search trees:

(i) The is_element operation leads to a vertex where a new leaf can be added

(ii) Attach the leaf containing the new key

(iii) Restore the height balancing, if necessary

Rebalancing AVL Trees After an Insertion:

There are multiple cases to be distinguished:

1. The newly attached leaf already has a left or right brother: In this case no subtree changes in height as a result of step (ii). No rebalancing is needed.
2. The newly attached leaf is an only child of its father v. In this case, v and all ancestors of v have a subtree whose height changes as a result of step (ii). This may lead to height balance violations.

Let us examine the potential problem cases:

Rebalancing AVL Trees After an Insertion:

There are multiple cases to be distinguished:

1. The newly attached leaf already has a left or right brother: In this case no subtree changes in height as a result of step (ii). No rebalancing is needed.
2. The newly attached leaf is an only child of its father v. In this case, v and all ancestors of v have a subtree whose height changes as a result of step (ii). This may lead to height balance violations.

Let us examine the potential problem cases:

Rebalancing AVL Trees After an Insertion:

There are multiple cases to be distinguished:

1. The newly attached leaf already has a left or right brother: In this case no subtree changes in height as a result of step (ii). No rebalancing is needed.
2. The newly attached leaf is an only child of its father v. In this case, v and all ancestors of v have a subtree whose height changes as a result of step (ii). This may lead to height balance violations.

Let us examine the potential problem cases:

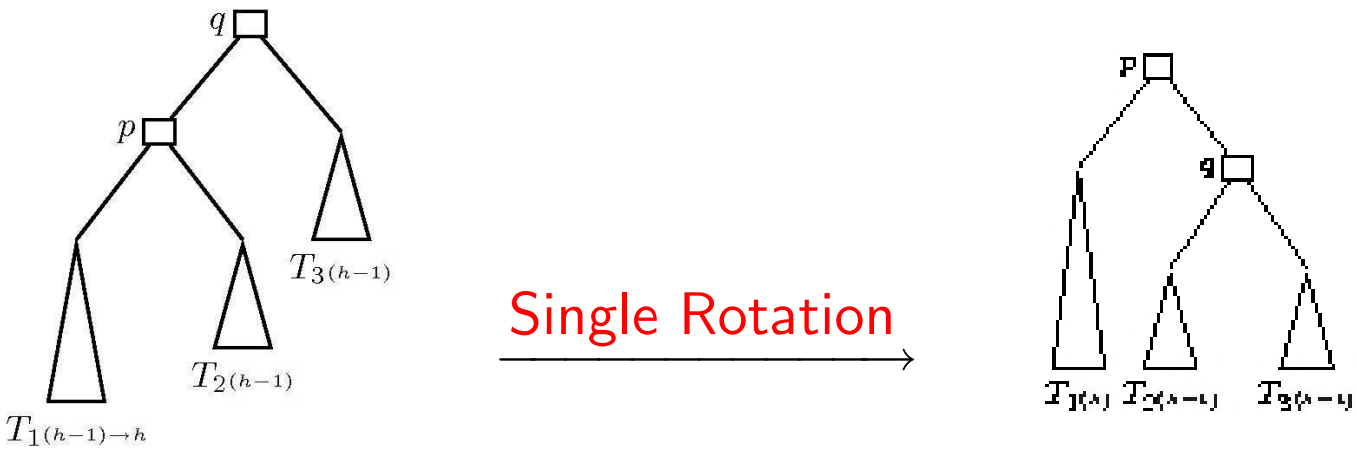
Rebalancing AVL Trees After an Insertion:

There are multiple cases to be distinguished:

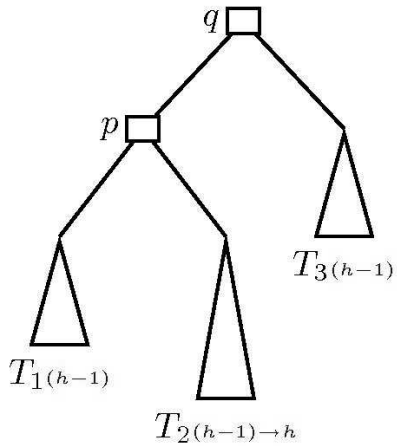
1. The newly attached leaf already has a left or right brother: In this case no subtree changes in height as a result of step (ii). No rebalancing is needed.
2. The newly attached leaf is an only child of its father v. In this case, v and all ancestors of v have a subtree whose height changes as a result of step (ii). This may lead to height balance violations.

Let us examine the potential problem cases:

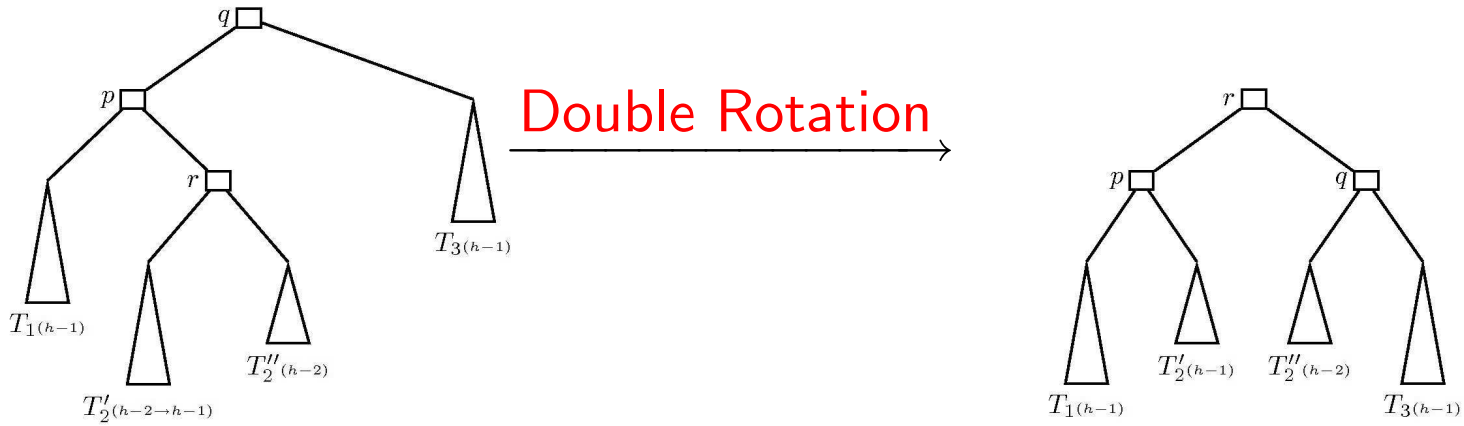
Case 1: The outer subtree below p increases its height from $h-1$ to h, pushing q's balance from -1 to -2. Note that the single rotation preserves the subtree ordering required of search trees (sorting condition) and, at the same time, repairs the balance violation at the root of the considered subtree.



Case 2: The inner subtree below p increases its height from $h-1$ to h, pushing q's balance from -1 to -2.



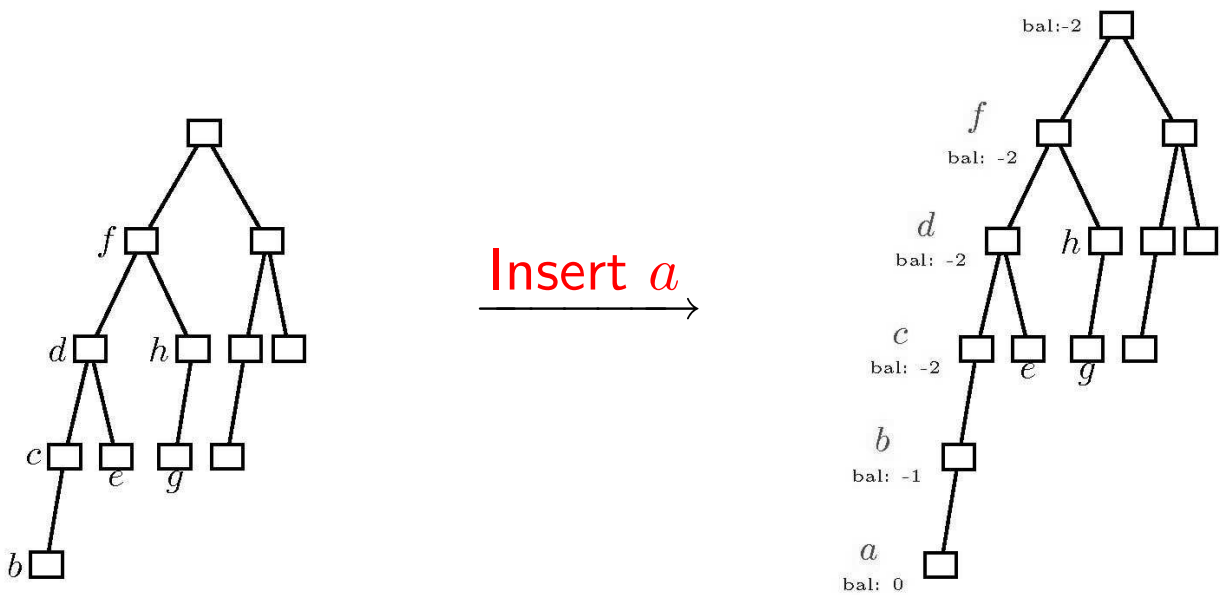
Try what happens if a similar approach is taken as in Case 1. — You will see that this doen't help yet. It turns out that this case requires a closer look into the subtree of p



Note that the double rotation preserves the subtree ordering required of search trees (sorting condition) and, at the same time, repairs the balance violation at the root of the considered subtree.

Also note that a single insertion of a new leaf can cause balance violations on the entire path from that leaf up to the root.

Example 5



Important observation: After the rebalancing procedure, applied to vertex v, the entire subtree rooted at v regains the same height as before the insertion of the new leaf (key a). This means that legal balance values are re-established at vertex v and all its ancestors. Therefore, the rebalancing has to be performed at the deepest vertex with illegal balance. In the example (rebalancing at node c):