Balanced Binary Search Trees

(Version of 28th April 2010)

(With pictures by John Morris and from the Wikipedia)

Road-Map:

Consider various seen data structures containing n elements:

	access	search	insertion
Arrays	$\Theta(1)$	$O(n)$	$O(n)$
Sorted arrays	$\Theta(1)$	$O(\log_2 n)$	$O(n)$
?	?	$O(\log_2 n)$	$O(\log_2 n)$

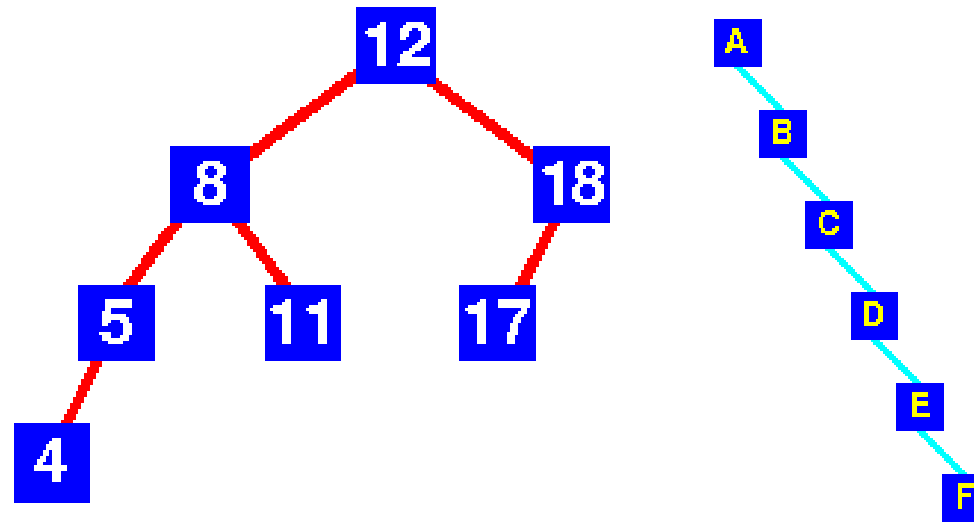
Can we also get insertion in $O(\log_2 n)$ time at worst?

Yes, at the cost of sacrificing the $\Theta(1)$ access time on a given key!

Observations on Binary Search Trees

The search time in a binary search tree depends on the shape of the tree, that is on the order in which its elements were inserted.

A pathological case: The n elements are inserted by increasing order on the keys, yielding something like a linear list (but with a worse memory consumption), with $O(n)$ search time at worst:



Element search, insertion, deletion, $\ldots$ all take worst-case time proportional to the height of the binary search tree.

The height h (here: the number of nodes on the longest branch) of a binary tree of n elements is such that $\log_2 n < h \leq n$.

The height of a randomly built (via insertions only, all keys being of equal probability) binary search tree of n elements is $O(\log_2 n)$.

In practice, one can however not always guarantee that binary search trees are built randomly. Binary search trees are thus only interesting when they are "relatively complete."

So we must look for specialisations of binary search trees whose worst-case performance on the basic tree operations can be guaranteed to be logarithmic at worst.

“Definition”: A balanced tree is a tree where every leaf is “not more than a certain distance” away from the root than any other leaf.

The balancing properties defining “not more than a certain distance” differ between various kinds of balanced trees:

- AVL trees (focus of this course)
- red-black trees
- ...

Inserting into, and deleting from, a balanced binary search tree involves transforming the tree if its balancing property — which is to be kept invariant — is violated.

These re-balancing transformations should also take $O(\log_2 n)$ time at worst, so that the effort is worth it. These transformations are built from operators (introduced on the second-next page) that are independent of the balancing property.

Walks of Binary Trees

Definition: A walk of a tree is a way of visiting each of its elements exactly once. For binary trees, we distinguish:

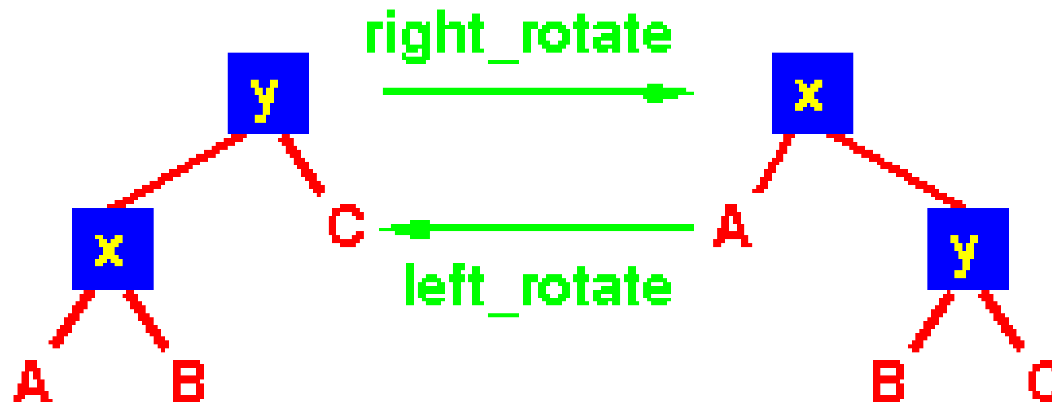
- preorder walk (first visit the *root*, then the left sub-tree, and last the right sub-tree)
- inorder walk (left, *root*, right)
- postorder walk (left, right, *root*)

Remark: The inorder walk of a binary search tree gives its elements in sorted (increasing) key order.

Question: Do we have a linear-time sorting algorithm now?!

Rotations of Binary Trees

Definition: A rotation of a binary tree transforms it so that its inorder-walk key ordering is preserved. We distinguish left rotation and right rotation:



inorder walk = A x B y C

preorder walk = y x A B C   preorder walk = x A y B C

postorder walk = A B x C y   postorder walk = A B C y x

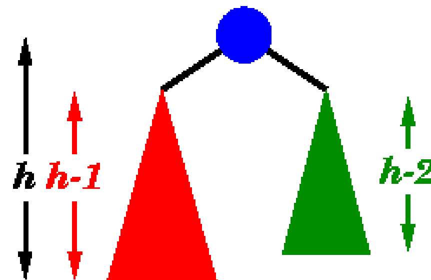
Note that both rotations can be implemented to take $\Theta(1)$ time.

AVL Trees

G. M. Adel'son-Velskii and E. M. Landis (USSR, 1962) proposed the first dynamically balanced trees.

AVL trees are not perfectly balanced, but guarantee $O(\log_2 n)$ worst-case search, insertion, and deletion times for n-element trees.

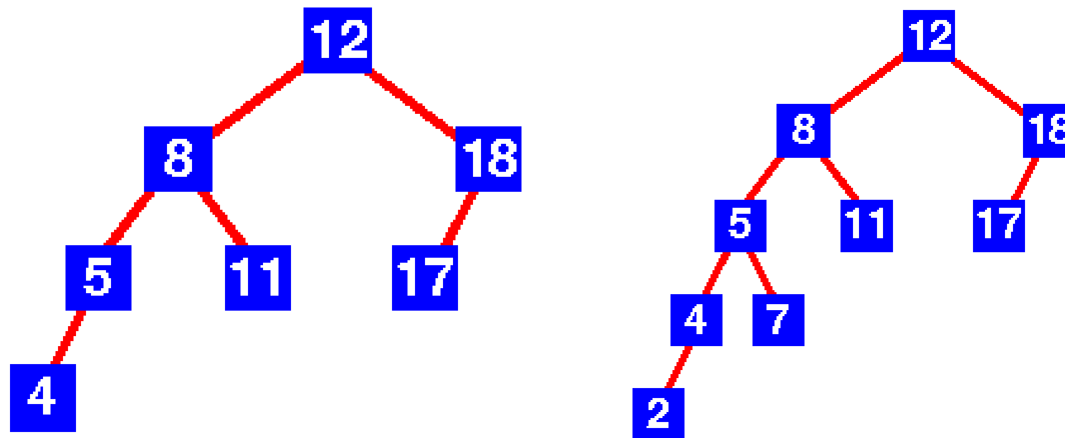
Definition: An AVL tree is a binary search tree where the sub-trees of every node differ in height by at most 1:



and conversely: when the left and right sub-trees are exchanged,
or when *both* are of height $h - 1$

© Pierre Flener, IT Department, Uppsala University, Sweden

Two Examples and One Counter-Example



Let us annotate each node with a balance factor:

- $\bullet$ when the tree rooted at this node is stable
- $-$ when the tree rooted at this node is left-heavy
- $+$ when the tree rooted at this node is right-heavy
- $--$ when the tree rooted at this node is left-unbalanced
- $++$ when the tree rooted at this node is right-unbalanced

What Can We Expect from AVL Trees?

Key Questions:

What is the maximum height $hMax(n)$ of an AVL tree with n nodes?

What is the minimum nbr $nMin(h)$ of nodes of an AVL tree of height h?

The latter question is equivalent to the former, but easier to answer!

Recurrence:

$$
\begin{aligned}
nMin(0) &= 0 \\
nMin(1) &= 1 \\
nMin(h) &= 1 + nMin(h-1) + nMin(h-2), \text{ when } h > 1
\end{aligned}
$$

Compare the $nMin$ series with the Fibonacci series:

h	0	1	2	3	4	5	6	7	8	...
$nMin(h)$	0	1	2	4	7	12	20	33	54	...
$fib(h)$	0	1	1	2	3	5	8	13	21	...

Observe: $nMin(h) = fib(h+2) - 1$. (**Exercise:** Prove this!)

Equivalently, the maximum height $hMax(n)$ of an AVL tree with n elements is the largest h such that:

$$fib(h+2) - 1 \leq n$$

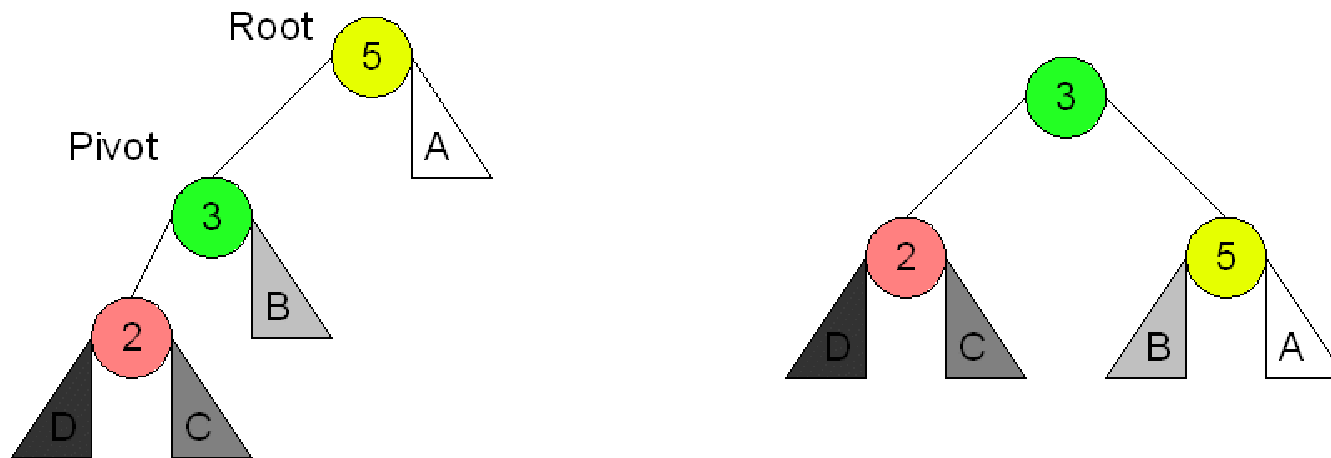
which simplifies into $hMax(n) \leq 1.44 \cdot \log_2(n+1) - 1.33$
so that search in an AVL tree indeed takes $O(\log_2 n)$ time at worst!

Insertion into AVL Trees

How to insert — in logarithmic time — an element into an AVL tree such that it remains an AVL tree? After locating the insertion place and performing a binary-search-tree insertion, there are five cases:

1. Every sub-tree remains balanced ($\bullet$, $-$, or $+$): do nothing.
2. A sub-tree was left-heavy ($-$) and became left-unbalanced ($--$):
   (a) The left sub-tree became left-heavy ($-$):
   right-rotate the left sub-tree toward the root.
   (b) The left sub-tree became right-heavy ($+$):
   first left-rotate the right sub-tree of the left sub-tree toward its parent, and then right-rotate the left sub-tree toward the root.
3. A sub-tree was right-heavy ($+$) and became right-unbalanced ($++$):
   two symmetric sub-cases 3(c) & 3(d) to 2(a) & 2(b) above.

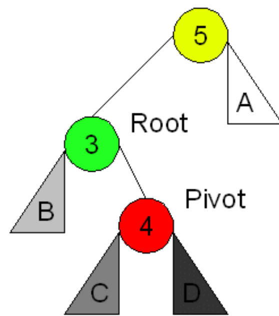
Case 2(a): Right-rotate left-heavy pivot 3 toward left-unbalanced root 5:



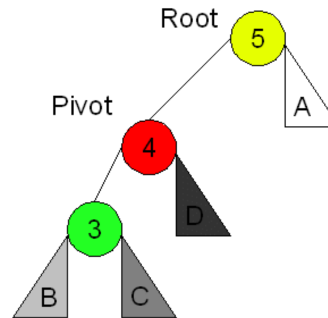
The inserted element is either 2 (if C and D are empty) or a leaf of C or D. The trees A and B have height h, while the tree rooted at 2 had height h before the insertion and has height $h + 1$ after the insertion.

Case 2(b): First left-rotate the stable pivot 4 toward the right-heavy root 3, and then right-rotate the now left-heavy pivot 4 toward the still left-unbalanced root 5:

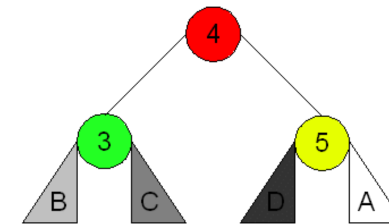
Left Right Case



Left Rotation



Right Rotation



The inserted element is either 4 (if C and D are empty) or a leaf of C or D. The trees A and B have height h, while the tree rooted at 4 had height h before the insertion and has height $h + 1$ after the insertion.

Property:

$$\text{An insertion includes re-balancing} \\ \Rightarrow \quad (\not\Leftarrow) \\ \text{that insertion does not modify the height of the tree.}$$

Complexity: Insertion requires at most two walks of the path from the root to the added element, plus at most two constant-time rotations, hence insertion indeed takes $O(\log_2 n)$ time at worst.

Conclusion: AVL trees are interesting when:

- the number n of elements is large (say $n \geq 50$), and
- the keys are (suspected of) not appearing randomly, and
- the ratio s/i of searches to insertions is large enough (say $s/i \geq 5$) to justify the costs of re-balancing.