Balanced Binary Trees

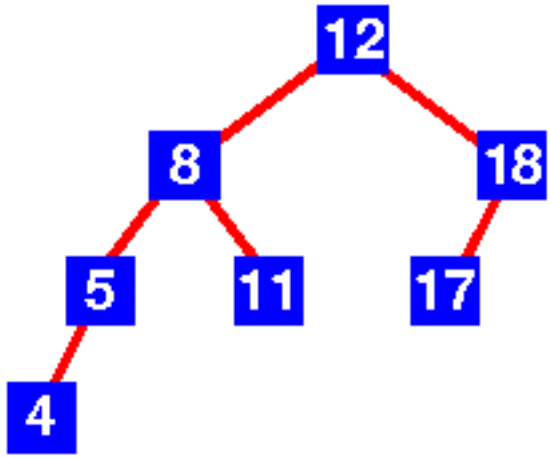
With pictures by John Morris (ciips.ee.uwa.edu.au/~morris)

Reminders about Trees

A *binary tree* is a tree with exactly two sub-trees for each node, called the *left* and *right* sub-trees.

A *binary search tree* is a binary tree where, for each node *m*,
the left sub-tree only has nodes with keys smaller than (according to some total order) the key of *m*,
while the right sub-tree only has nodes with keys larger than the key of *m*.

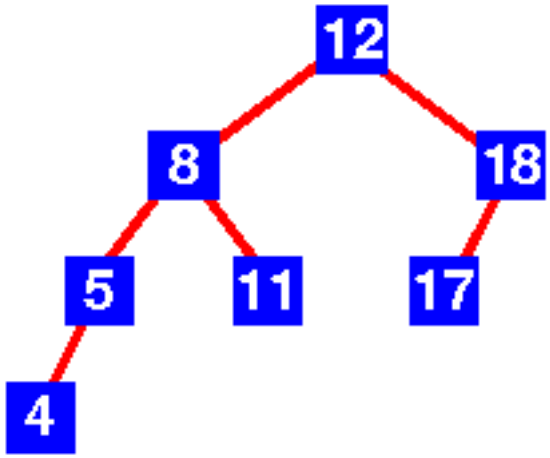
The *height* of a tree is the number of nodes on its longest branch (a path from the root to a leaf).



Observations

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

A pathological case: The *n* nodes were inserted by increasing order on the keys,
yielding something like a linear list (but with a worse space consumption), with O(*n*) search time.



Node search, insertion, deletion, … *all* take time proportional to the height of the binary search tree.
The height *h* of a binary tree of *n* nodes is such that $\log_2 n < h \leq n$.
The height of a *randomly* built binary search tree of *n* nodes is $O(\log_2 n)$.

Observations (continued)

In practice, one can however *not* always guarantee that binary search trees are built randomly. Binary search trees are thus only interesting if 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 good, that is $O(\log_2 n)$ time.

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 various balancing schemes give actual definitions for "not more than a certain distance" and require different efforts to keep the trees balanced:

- AVL trees
- Red-black trees.

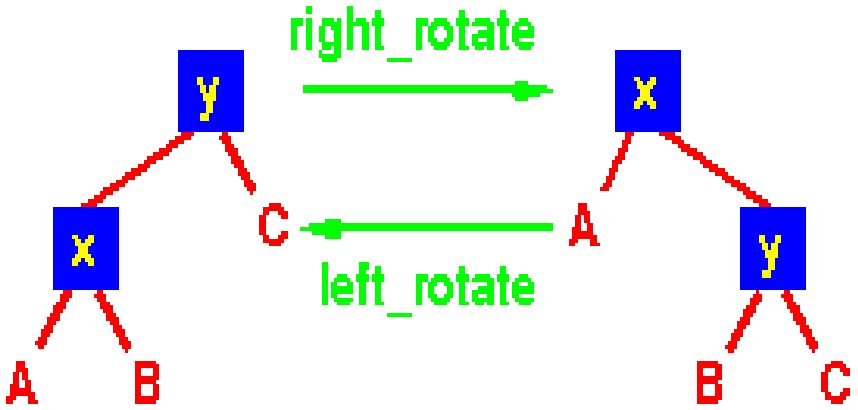
Inserting into, and deleting from, a balanced binary search tree involves transforming the tree if its balancing property — which is to be kept *invariant* — is violated.
These re-balancing transformations should *also* take $O(\log_2 n)$ time, so that the effort is worth it.
These transformations are built from operators that are *independent* from the balancing scheme.

Operations on Binary Trees

A *walk* of a tree is a way of visiting each of its nodes exactly once. For binary trees, we distinguish *preorder walk* (visit the *root*, then the left sub-tree, and last the right sub-tree), *inorder walk* (left, *root*, right), and *postorder walk* (left, right, *root*).

Remark: The inorder walk of a binary search trees gives its nodes in *sorted* (increasing) key order.

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

postorder walk = A B x C y

preorder walk = x A y B C

postorder walk = A B C y x

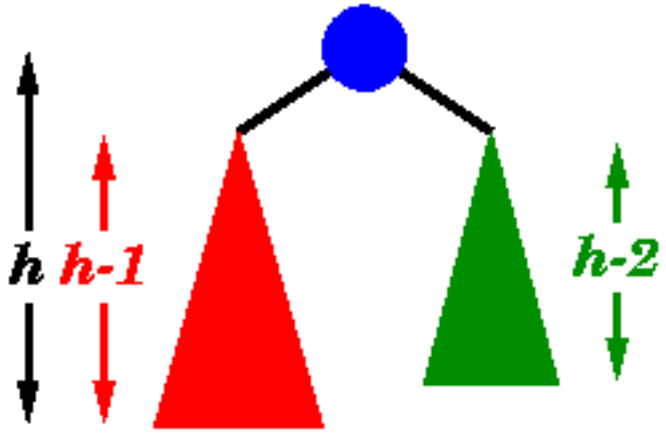
AVL Trees

Definition

Named after their inventors — G.M. Adel'son-Velskii and E.M. Landis (USSR) —
AVL trees were the first dynamically balanced trees to be proposed, namely in 1962.

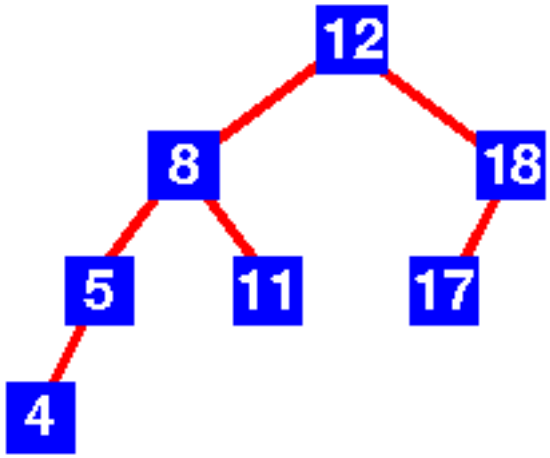
They are not perfectly balanced, but maintain $O(\log_2 n)$ search, insertion, and deletion times,
when n is the number of nodes in the tree.

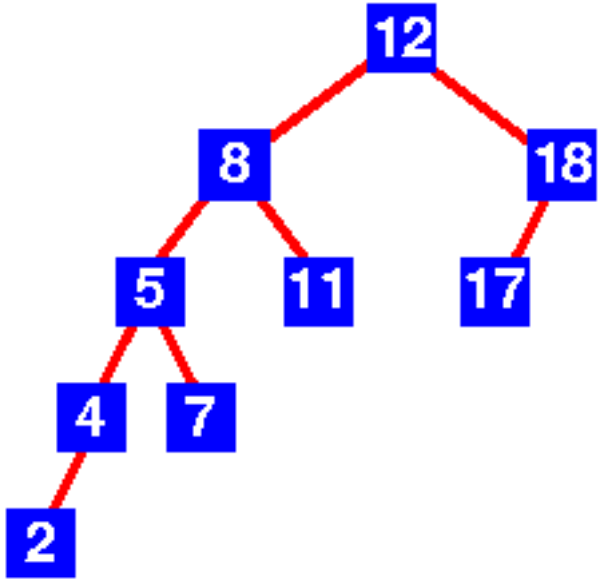
An *AVL tree* is a binary search tree where the sub-trees of every node differ in height by at most 1.



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

Two Examples and One Counter-Example





Let us annotate each node with a *balance factor*:

- • 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 Performance Can We Expect from AVL Trees?

Key question: What is the maximal height $hmax(n)$ of an AVL tree with n nodes?

Conversely: What is the minimal number of nodes $nmin(h)$ of an AVL tree of height h?

The latter question is easier to answer!

$$nmin(0) = 0$$
$$nmin(1) = 1$$
$$nmin(h) = 1 + nmin(h-1) + nmin(h-2), \quad \text{when } h > 1$$

Compare this 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	…

Hence: $nmin(h) = fib(h+2) - 1$. (Homework: Prove this!)

Conversely, the maximal height $hmax(n)$ of an AVL tree with n nodes is the largest h such that:

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

which simplifies into $hmax(n) \le 1.44 \log_2(n+1) - 1.33$
so that search in an AVL tree indeed takes $O(\log_2 n)$ time!

Insertion into AVL Trees

How to insert — in $O(\log_2 n)$ time — a node into an AVL tree such that it remains an AVL tree?

After locating the insertion place and performing the insertion, there are three cases:

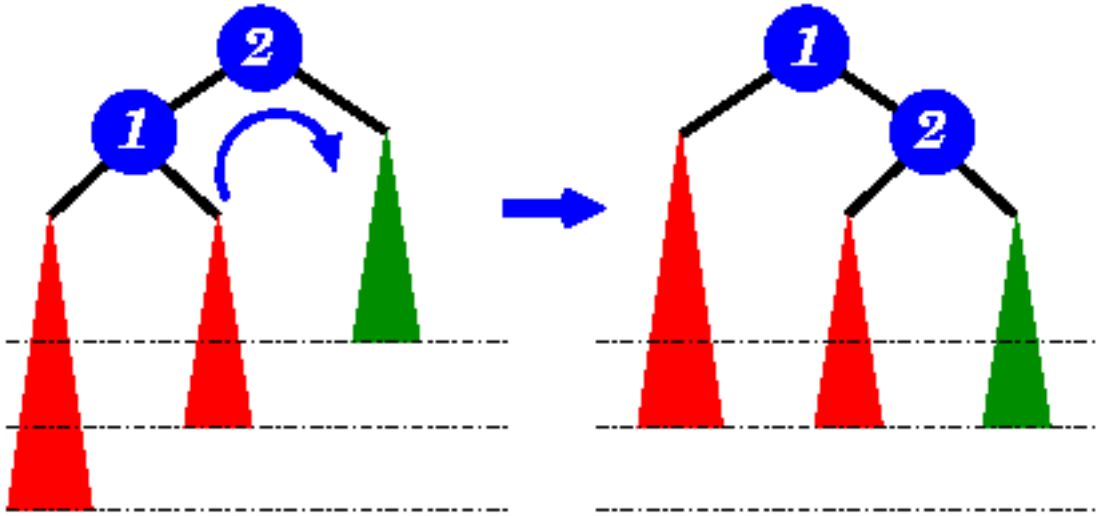
(1) The tree remains balanced (•, −, or +): do nothing.

(2) A tree was left-heavy (–) and became left-unbalanced (−−): see below.

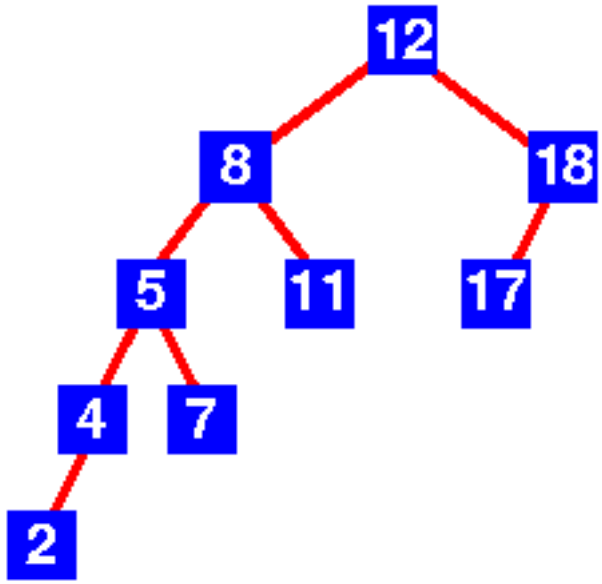
(3) A tree was right-heavy (+) and became right-unbalanced (++): symmetric to the second case.

Regarding case (2), there are two sub-cases:

- The left sub-tree became *left*-heavy: right-rotate the left sub-tree to the root.



Example:



- The left sub-tree became *right*-heavy: *first* left-rotate the right sub-tree of the left sub-tree to its parent, and *then* right-rotate the left sub-tree to the root.

Property: An insertion requires re-balancing $\rightarrow$ It does *not* modify the height of the tree.

Insertion requires at most two walks of the path from the root to the added node,
hence indeed takes $O(\log_2 n)$ time.

Conclusion

AVL trees are interesting when:

- the number n of nodes 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 cost of re-balancing.