COS 423
Spring, 2009

Rank-Balanced Binary Search Trees

These notes describe a relaxation of AVL trees. These trees have properties like those of red-black trees but are slightly easier to maintain. In particular, rebalancing after an insertion or a deletion takes one single or double rotation and logarithmic time worst case, O(1) time amortized. Red-black trees require up to three single rotations for a deletion, AVL trees a logarithmic number.

A *ranked binary tree* is a binary tree whose nodes have integer *ranks*, with leaves having rank zero and missing nodes having rank minus one. If x is a child, the *rank difference* of x, Δx, is the rank of its parent minus the rank of x. The *rank* of a ranked tree is the rank of its root. An AVL tree is a ranked binary tree such that every child has rank difference one or two and every node has at least one child with rank difference one. We call this the *balance condition.* In an AVL tree the rank of a node is its height; this will no longer be true when we relax the balance condition. To represent an AVL tree we store with each node pointers to its children and, if it is a child, a bit that indicates whether its rank difference is one or two; we do not need to store ranks explicitly.

An *AVL search tree* is an AVL tree whose nodes contain items from a totally ordered universe, arranged in symmetric order in the tree. Henceforth we shall omit the word "search"; all our trees are search trees. To do an insertion in an AVL tree, follow the search path for the item until reaching an external node, and replace the external node by a new leaf (of rank zero) containing the item. This will violate the balance condition if the new leaf is a child of an old leaf. In this case, increase the rank of the old leaf from zero to one and walk back up along the search path, repeating the following step until balance is restored. In general there is at most one violation: a node q has $\Delta q = 0$, and exactly one child of q has rank difference one. Let p and r be the parent and sibling of q, respectively. There are two cases (see Figure 1):

- $\Delta r = 1$: Increase the rank of p by one. This repairs the violation at q but may create a new violation at p. The children of node p now have rank differences 1 (q) and 2 (r).
- $\Delta r = 2$: Assume q is the left child of p; the other case is symmetric. Exactly one child of q has rank difference one. Let t be the right child of q. There are two subcases:
  - $\Delta t = 2$: Do a right rotation at q and decrease the rank of p by one. This repairs the violation without creating a new one.
  - $\Delta t = 1$: Do a double rotation at t, making q its left child and p its right child. Increase the rank of t by one and decrease the ranks of p and q by one. This repairs the violation without creating a new one.

One can do a deletion on an AVL tree similarly, but the rebalancing may require a logarithmic number of rotations, rather than the one or two needed for an insertion. To avoid this, we relax the balance condition by eliminating the requirement that at least one child of a node has rank difference one. We call the resulting trees *rank-balanced trees* or *rb-trees* (not to be confused with another kind of rank-balanced trees, red-black trees). We call a node a *d-node* for $d = 2$, 3, or 4 if the sum of the rank differences of its children is d. Leaves are 2-nodes by this definition. An AVL tree is just an rb-tree with no 4-nodes.

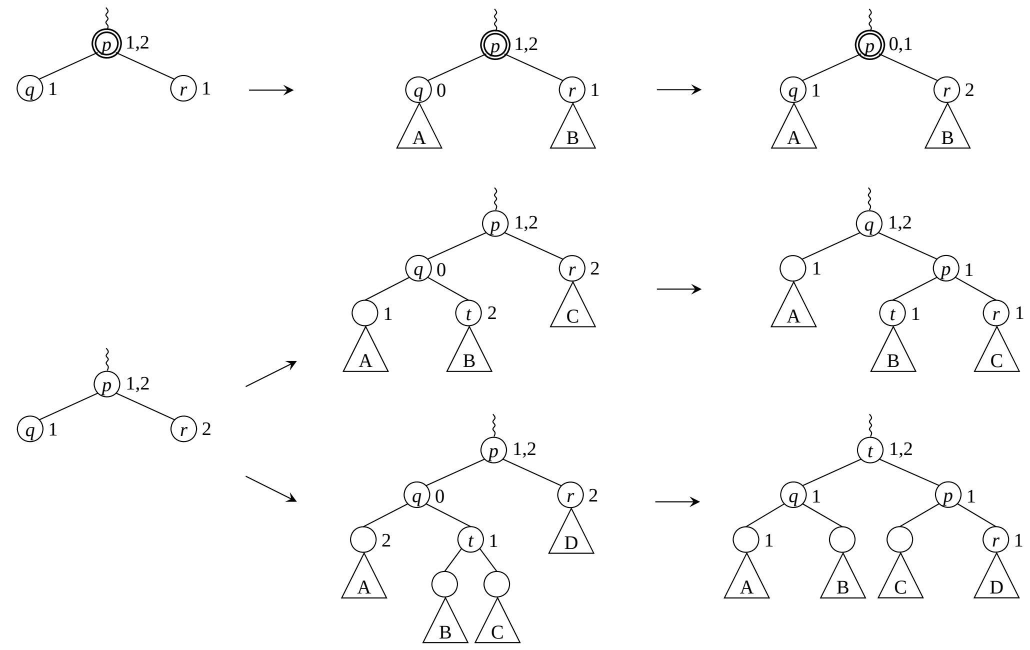

Figure 1: Rebalancing after an insertion. Numbers are rank differences. The first case may be non-terminating if $\Delta p = 0$ after the update. In the first case, node p (double-circled) loses potential.

The minimum number of nodes n_k in an rb-tree of rank k satisfies the recurrence $n_0 = 1$, $n_1 = 2$, $n_k = 2n_{k-2} + 1$ for $k \geq 2$. By induction $n_k \geq 2^{\lceil k/2 \rceil}$. Thus the rank of an n-node rb-tree is at most $2 \lg n$.

Insertion is the same on rb-trees as on AVL trees: rebalancing steps cannot create 4-nodes, only convert them into 2-nodes. To do a deletion in an rb-tree, if the item is in a node with two children swap the item with the item in the predecessor (or successor) node. Now the item is in a leaf or a node with one child. Delete this node and give its child, if any, to its parent. This may violate the balance condition, because its effect is to decrease the rank of a node by one: if a leaf is deleted, it is replaced by an external node, of rank minus one; if a node with one child (and rank one) is deleted, it is replaced by a leaf, of rank zero. To rebalance the tree, walk up the path to the root, repeating the following step until balance is restored. In general there is at most one violation: a node q has rank difference three. Let p and r be the parent and sibling of q, respectively. There are two cases (see Figure 2):

- $\Delta r = 2$: Decrease the rank of p by one. This repairs the violation at q but may create a new violation at p.
- $\Delta r = 1$: Assume q is the right child of p; the other case is symmetric. Let s and t be the left and right children of r. There are three subcases:
  - $\Delta s = \Delta t = 2$: Decrease the ranks of p and r by one. This repairs the violation at q but may create a new violation at p.

– $\Delta s = 1$: Do a right rotation at r, increase the rank of r by one, and decrease the rank of p by one. This repairs the violation without creating a new one.

– $\Delta s = 2$ and $\Delta t = 1$: Do a double rotation at t, making r its left child and p its right child. Increase the rank of t by two, decrease the rank of r by one, and decrease the rank of p by two. This repairs the violation without creating a new one.

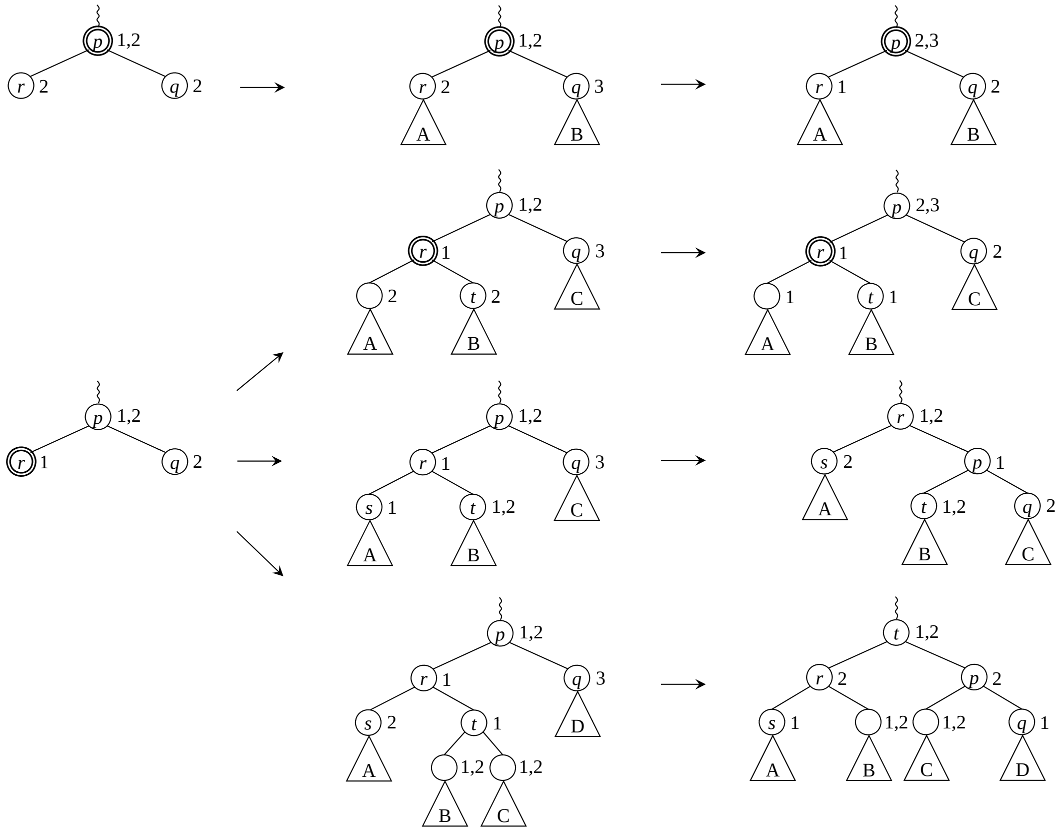

Figure 2: Rebalancing after a deletion. Numbers are rank differences. The first two cases are non-terminating if $\Delta p = 3$ after the update. Double-circled nodes lose potential.

Insertion and deletion differ mainly in that the former has only one non-terminating rebalancing case, the latter two. A search, insertion, or deletion takes $O(\log n)$ time in the worst case on an n-node tree; an insertion or deletion takes at most two rotations. A non-terminating rebalancing step after an insertion converts a 2-node into a 3-node; a non-terminating rebalancing step after a deletion converts a 4-node into a 2- or 3-node. (See the double-encircled nodes in Figures 1 and 2.) Thus if we define the potential of a tree to be the number of 2-nodes plus twice the number of 4-nodes and charge one unit of time for each non-terminating rebalancing step, each such step has amortized time zero or minus one, and the amortized time for rebalancing after an insertion or deletion is $O(1)$.

Exercise: Develop insertion and deletion algorithms for rb-trees that do the rebalancing top-down as the search for the item to be inserted or deleted proceeds. Can you make the amortized number of rotations for an insertion or deletion $O(1)$?