## 11.2 Balanced Search Trees

In the closing of the previous section, we noted that if we could assume a random series of insertions and removals, the standard binary search tree supports $O(\log n)$ expected running times for the basic map operations. However, we may only claim $O(n)$ worst-case time, because some sequences of operations may lead to an unbalanced tree with height proportional to $n$.

In the remainder of this chapter, we explore four search tree algorithms that provide stronger performance guarantees. Three of the four data structures (AVL trees, splay trees, and red-black trees) are based on augmenting a standard binary search tree with occasional operations to reshape the tree and reduce its height.

The primary operation to rebalance a binary search tree is known as a ***rotation***. During a rotation, we "rotate" a child to be above its parent, as diagrammed in Figure 11.8.

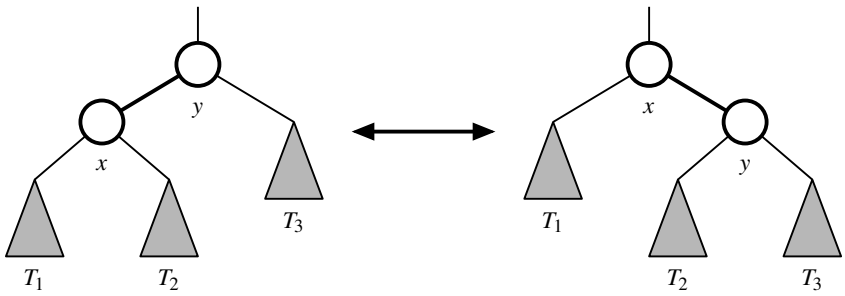


**Figure 11.8:** A rotation operation in a binary search tree. A rotation can be performed to transform the left formation into the right, or the right formation into the left. Note that all keys in subtree $T_1$ have keys less than that of position $x$, all keys in subtree $T_2$ have keys that are between those of positions $x$ and $y$, and all keys in subtree $T_3$ have keys that are greater than that of position $y$.

To maintain the binary search tree property through a rotation, we note that if position $x$ was a left child of position $y$ prior to a rotation (and therefore the key of $x$ is less than the key of $y$), then $y$ becomes the *right* child of $x$ after the rotation, and vice versa. Furthermore, we must relink the subtree of items with keys that lie between the keys of the two positions that are being rotated. For example, in Figure 11.8 the subtree labeled $T_2$ represents items with keys that are known to be greater than that of position $x$ and less than that of position $y$. In the first configuration of that figure, $T_2$ is the right subtree of position $x$; in the second configuration, it is the left subtree of position $y$.

Because a single rotation modifies a constant number of parent-child relationships, it can be implemented in $O(1)$ time with a linked binary tree representation.

In the context of a tree-balancing algorithm, a rotation allows the shape of a tree to be modified while maintaining the search tree property. If used wisely, this operation can be performed to avoid highly unbalanced tree configurations. For example, a rightward rotation from the first formation of Figure 11.8 to the second reduces the depth of each node in subtree $T_1$ by one, while increasing the depth of each node in subtree $T_3$ by one. (Note that the depth of nodes in subtree $T_2$ are unaffected by the rotation.)

One or more rotations can be combined to provide broader rebalancing within a tree. One such compound operation we consider is a ***trinode restructuring***. For this manipulation, we consider a position $x$, its parent $y$, and its grandparent $z$. The goal is to restructure the subtree rooted at $z$ in order to reduce the overall path length to $x$ and its subtrees. Pseudo-code for a restructure($x$) method is given in Code Fragment 11.9 and illustrated in Figure 11.9. In describing a trinode restructuring, we temporarily rename the positions $x$, $y$, and $z$ as $a$, $b$, and $c$, so that $a$ precedes $b$ and $b$ precedes $c$ in an inorder traversal of $T$. There are four possible orientations mapping $x$, $y$, and $z$ to $a$, $b$, and $c$, as shown in Figure 11.9, which are unified into one case by our relabeling. The trinode restructuring replaces $z$ with the node identified as $b$, makes the children of this node be $a$ and $c$, and makes the children of $a$ and $c$ be the four previous children of $x$, $y$, and $z$ (other than $x$ and $y$), while maintaining the inorder relationships of all the nodes in $T$.

**Algorithm** restructure(x):

***Input:*** A position x of a binary search tree T that has both a parent y and a grandparent z

***Output:*** Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let $(T_1, T_2, T_3, T_4)$ be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b.
3: Let a be the left child of b and let $T_1$ and $T_2$ be the left and right subtrees of a, respectively.
4: Let c be the right child of b and let $T_3$ and $T_4$ be the left and right subtrees of c, respectively.

**Code Fragment 11.9:** The trinode restructuring operation in a binary search tree.

In practice, the modification of a tree $T$ caused by a trinode restructuring operation can be implemented through case analysis either as a single rotation (as in Figure 11.9a and b) or as a double rotation (as in Figure 11.9c and d). The double rotation arises when position $x$ has the middle of the three relevant keys and is first rotated above its parent, and then above what was originally its grandparent. In any of the cases, the trinode restructuring is completed with $O(1)$ running time.

**Case 1:** ***The Sibling s of y Is Black (or None).*** (See Figure 11.33.) In this case, the double red denotes the fact that we have added the new node to a corresponding 3-node of the $(2,4)$ tree $T'$, effectively creating a malformed 4-node. This formation has one red node ($y$) that is the parent of another red node ($x$), while we want it to have the two red nodes as siblings instead. To fix this problem, we perform a ***trinode restructuring*** of $T$. The trinode restructuring is done by the operation restructure($x$), which consists of the following steps (see again Figure 11.33; this operation is also discussed in Section 11.2):

- Take node $x$, its parent $y$, and grandparent $z$, and temporarily relabel them as $a$, $b$, and $c$, in left-to-right order, so that $a$, $b$, and $c$ will be visited in this order by an inorder tree traversal.
- Replace the grandparent $z$ with the node labeled $b$, and make nodes $a$ and $c$ the children of $b$, keeping inorder relationships unchanged.

After performing the restructure($x$) operation, we color $b$ black and we color $a$ and $c$ red. Thus, the restructuring eliminates the double-red problem. Notice that the portion of any path through the restructured part of the tree is incident to exactly one black node, both before and after the trinode restructuring. Therefore, the black depth of the tree is unaffected.

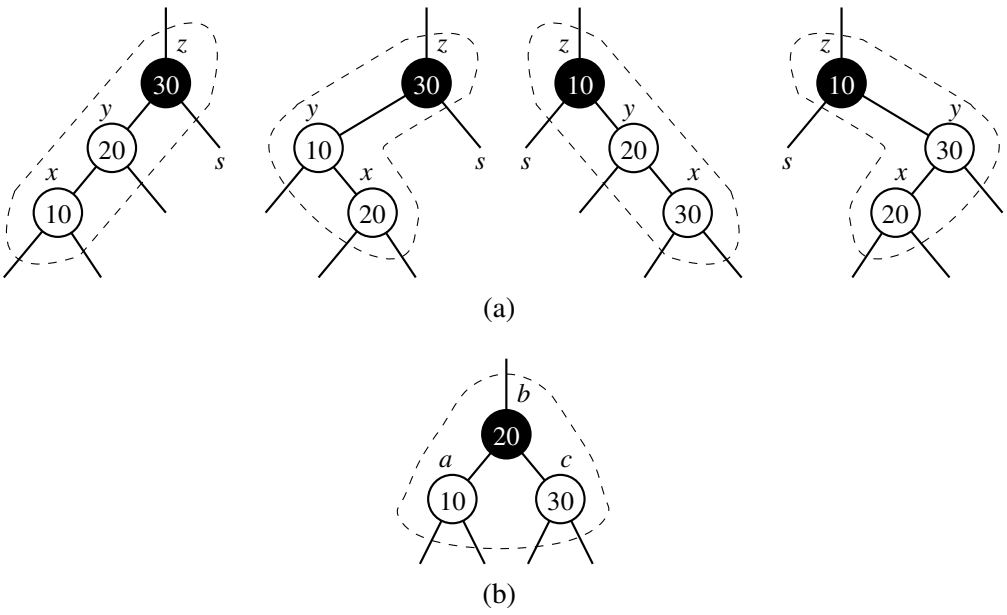


**Figure 11.33:** Restructuring a red-black tree to remedy a double red: (a) the four configurations for $x$, $y$, and $z$ before restructuring; (b) after restructuring.