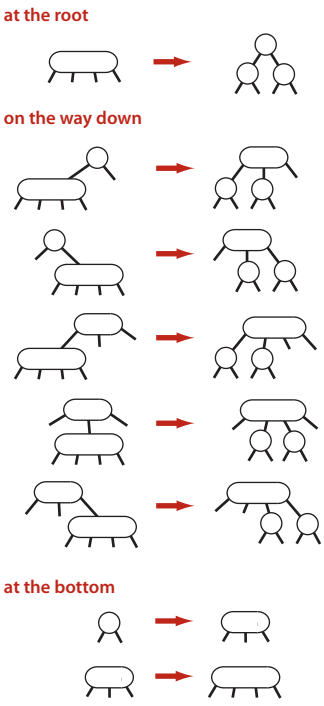
Deletion Since `put()` in ALGORITHM 3.4 is already one of the most intricate methods that we consider in this book, and the implementations of `deleteMin()`, `deleteMax()`, and `delete()` for red-black BSTs are a bit more complicated, we defer their full implementations to exercises. Still, the basic approach is worthy of study. To describe it, we begin by returning to 2-3 trees. As with insertion, we can define a sequence of local transformations that allow us to delete a node while still maintaining perfect balance. The process is somewhat more complicated than for insertion, because we do the transformations both on the way down the search path, when we introduce temporary 4-nodes (to allow for a node to be deleted), and also on the way up the search path, where we split any leftover 4-nodes (in the same manner as for insertion).



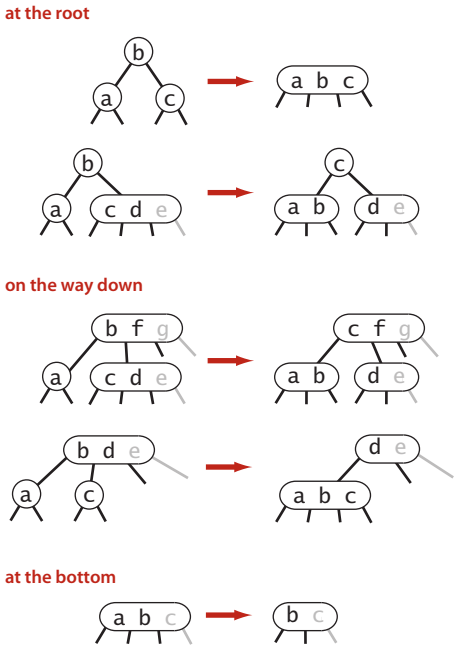
Transformations for insert in top-down 2-3-4 trees

Top-down 2-3-4 trees. As a first warmup for deletion, we consider a simpler algorithm that does transformations both on the way down the path and on the way up the path: an insertion algorithm for *2-3-4 trees*, where the temporary 4-nodes that we saw in 2-3 trees can persist in the tree. The insertion algorithm is based on doing transformations on the way down the path to maintain the invariant that the current node is not a 4-node (so we are assured that there will be room to insert the new key at the bottom) and transformations on the way up the path to balance any 4-nodes that may have been created. The transformations on the way down are precisely the same transformations that we used for splitting 4-nodes in 2-3 trees. If the root is a 4-node, we split it into three 2-nodes, increasing the height of the tree by 1. On the way down the tree, if we encounter a 4-node with a 2-node as parent, we split the 4-node into two 2-nodes and pass the middle key to the parent, making it a 3-node; if we encounter a 4-node with a 3-node as parent, we split the 4-node into two 2-nodes and pass the middle key to the parent, making it a 4-node. We do not need to worry about encountering a 4-node with a 4-node as parent by virtue of the invariant. At the bottom, we have, again by virtue of the invariant, a 2-node or a 3-node, so we have room to insert the new key. To implement this algorithm with red-black BSTs, we

- Represent 4-nodes as a balanced subtree of three 2-nodes, with both the left and right child connected to the parent with a red link
- Split 4-nodes on the way down the tree with color flips
- Balance 4-nodes on the way up the tree with rotations, as for insertion

Remarkably, you can implement top-down 2-3-4 trees by moving one line of code in `put()` in ALGORITHM 3.4: move the `colorFlip()` call (and accompanying test) to before the recursive calls (between the test for null and the comparison). This algorithm has some advantages over 2-3 trees in applications where multiple processes have access to the same tree, because it always is operating within a link or two of the current node. The deletion algorithms that we describe next are based on a similar scheme and are effective for these trees as well as for 2-3 trees.

Delete the minimum. As a second warmup for deletion, we consider the operation of deleting the minimum from a 2-3 tree. The basic idea is based on the observation that we can easily delete a key from a 3-node at the bottom of the tree, but not from a 2-node. Deleting the key from a 2-node leaves a node with no keys; the natural thing to do would be to replace the node with a null link, but that operation would violate the perfect balance condition. So, we adopt the following approach: to ensure that we do not end up on a 2-node, we perform appropriate transformations on the way down the tree to preserve the invariant that the current node is not a 2-node (it might be a 3-node or a temporary 4-node). First, at the root, there are two possibilities: if the root is a 2-node and both children are 2-nodes, we can just convert the three nodes to a 4-node; otherwise we can borrow from the right sibling if necessary to ensure that the left child of the root is not a 2-node. Then, on the way down the tree, one of the following cases must hold:



Transformations for delete the minimum

- If the left child of the current node is not a 2-node, there is nothing to do.
- If the left child is a 2-node and its immediate sibling is not a 2-node, move a key from the sibling to the left child.
- If the left child and its immediate sibling are 2-nodes, then combine them with the smallest key in the parent to make a 4-node, changing the parent from a 3-node to a 2-node or from a 4-node to a 3-node.

Following this process as we traverse left links to the bottom, we wind up on a 3-node or a 4-node with the smallest key, so we can just remove it, converting the 3-node to a

2-node or the 4-node to a 3-node. Then, on the way up the tree, we split any unused temporary 4-nodes.

Delete. The same transformations along the search path just described for deleting the minimum are effective to ensure that the current node is not a 2-node during a search for any key. If the search key is at the bottom, we can just remove it. If the key is not at the bottom, then we have to exchange it with its successor as in regular BSTs. Then, since the current node is not a 2-node, we have reduced the problem to deleting the minimum in a subtree whose root is not a 2-node, and we can use the procedure just described for that subtree. After the deletion, as usual, we split any remaining 4-nodes on the search path on the way up the tree.

SEVERAL OF THE EXERCISES at the end of this section are devoted to examples and implementations related to these deletion algorithms. People with an interest in developing or understanding implementations need to master the details covered in these exercises. People with a general interest in the study of algorithms need to recognize that these methods are important because they represent the first symbol-table implementation that we have seen where *search*, insert, and delete are all guaranteed to be efficient, as we will establish next.