its red children are absorbed? What can you say about the depths of the leaves of the resulting tree?

***13.1-5***
Show that the longest simple path from a node $x$ in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node $x$ to a descendant leaf.

***13.1-6***
What is the largest possible number of internal nodes in a red-black tree with black-height $k$? What is the smallest possible number?

***13.1-7***
Describe a red-black tree on $n$ keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

## 13.2 Rotations

The search-tree operations TREE-INSERT and TREE-DELETE, when run on a red-black tree with $n$ keys, take $O(\lg n)$ time. Because they modify the tree, the result may violate the red-black properties enumerated in Section 13.1. To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.

We change the pointer structure through ***rotation***, which is a local operation in a search tree that preserves the binary-search-tree property. Figure 13.2 shows the two kinds of rotations: left rotations and right rotations. When we do a left rotation on a node $x$, we assume that its right child $y$ is not $T.nil$; $x$ may be any node in the tree whose right child is not $T.nil$. The left rotation "pivots" around the link from $x$ to $y$. It makes $y$ the new root of the subtree, with $x$ as $y$'s left child and $y$'s left child as $x$'s right child.

The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$.

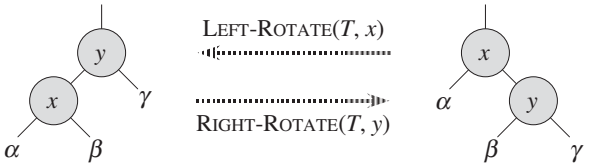


**Figure 13.2** The rotation operations on a binary search tree. The operation LEFT-ROTATE$(T, x)$ transforms the configuration of the two nodes on the right into the configuration on the left by changing a constant number of pointers. The inverse operation RIGHT-ROTATE$(T, y)$ transforms the configuration on the left into the configuration on the right. The letters $\alpha$, $\beta$, and $\gamma$ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in $\alpha$ precede $x.key$, which precedes the keys in $\beta$, which precede $y.key$, which precedes the keys in $\gamma$.

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

Figure 13.3 shows an example of how LEFT-ROTATE modifies a binary search tree. The code for RIGHT-ROTATE is symmetric. Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time. Only pointers are changed by a rotation; all other attributes in a node remain the same.

## Exercises

### *13.2-1*

Write pseudocode for RIGHT-ROTATE.

### *13.2-2*

Argue that in every $n$-node binary search tree, there are exactly $n - 1$ possible rotations.