Program 12.12 Rotations in BSTs

These twin routines perform the *rotation* operation on a BST. A *right rotation* makes the old root the right subtree of the new root (the old left subtree of the root); a *left rotation* makes the old root the left subtree of the new root (the old right subtree of the root). For implementations where a count field is maintained in the nodes (for example, to support *select*, as we will see in Section 14.9), we need also to exchange the count fields in the nodes involved in the rotation (see Exercise 12.75).

```
void rotR(link& h)
  { link x = h->l; h->l = x->r; x->r = h; h = x; }
void rotL(link& h)
  { link x = h->r; h->r = x->l; x->l = h; h = x; }
```

we implement *remove*, *join*, and other ADT operations with rotations; in Chapter 13 we use them to help us build trees that afford near-optimal performance.

The rotation operations provide a straightforward recursive implementation of root insertion: Recursively insert the new item into the appropriate subtree (leaving it, when the recursive operation is complete, at the root of that tree), then rotate to make it the root of the main tree. Figure 12.14 depicts an example, and Program 12.13 is a direct implementation of this method. This program is a persuasive example of the power of recursion—any reader not so persuaded is encouraged to try Exercise 12.76.

Figures 12.15 and 12.16 show how we construct a BST by inserting a sequence of keys into an initially empty tree, using the root insertion method. If the key sequence is random, a BST built in this way has precisely the same stochastic properties as does a BST built by the standard method. For example, Properties 12.6 and 12.7 hold for BSTs built by root insertion.

In practice, an advantage of the root insertion method is that recently inserted keys are near the top. The cost for search hits on recently inserted keys therefore is likely to be lower than that for the standard method. This property is significant, because many applications have precisely this kind of dynamic mix among their *search* and *insert* operations. A symbol table might contain a great many items, but a large fraction of the searches might refer to the items that

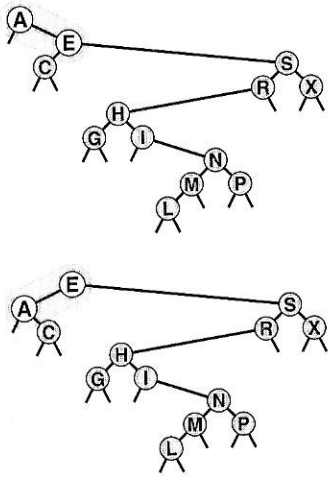

Figure 12.13
Left rotation in a BST

This diagram shows the result (bottom) *of a left rotation at* A *in an example BST* (top). *The node containing* A *moves down in the tree, becoming the left child of its former right child.*

We accomplish the rotation by getting the link to the new root E *from the right link of* A, *setting the right link of* A *by copying the left link of* E, *setting the left link of* E *to* A, *and setting the link to* A *(the head link of the tree) to point to* E *instead.*

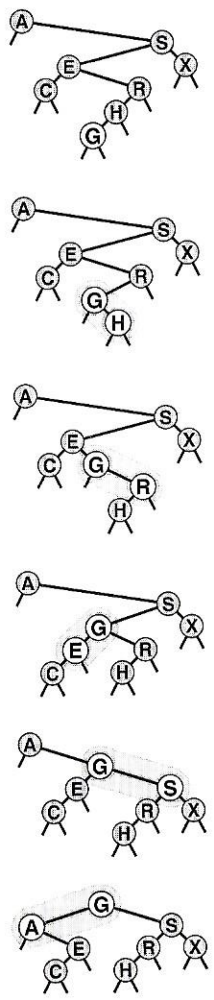

Figure 12.14
BST root insertion

This sequence depicts the result of inserting G *into the BST at the top, with (recursive) rotation after insertion to bring the newly inserted node* G *to the root. The process is equivalent to inserting* G*, then performing a sequence of rotations to bring it to the root.*

Program 12.13 Root insertion in BSTs

With the rotation functions in Program 12.12, a recursive function that inserts a new node at the root of a BST is immediate: Insert the new item at the root in the appropriate subtree, then perform the appropriate rotation to bring it to the root of the main tree.

```
private:
  void insertT(link& h, Item x)
    { if (h == 0) { h = new node(x); return; }
      if (x.key() <  h->item.key())
           { insertT(h->l, x); rotR(h); }
      else { insertT(h->r, x); rotL(h); }
    }
public:
  void insert(Item item)
    { insertT(head, item); }
```

were most recently inserted. For example, in a commercial transaction processing system, active transactions could remain near the top and be processed quickly, without access to old transactions being lost. The root insertion method gives the data structure this and similar properties automatically.

If we also change the *search* function to bring the node found to the root when we have a search hit, then we have a self-organizing search method (see Exercise 12.28) that keeps frequently accessed nodes near the top of the tree. In Chapter 13, we shall see a systematic application of this idea to provide a symbol-table implementation that has guaranteed fast performance characteristics.

As is true of several other methods that we have mentioned in this chapter, it is difficult to make precise statements about the performance of the root insertion method versus the standard insertion method for practical applications, because the performance depends on the mixture of symbol-table operations in a way that is difficult to characterize analytically. Our inability to analyze the algorithm should not necessarily dissuade us from using root insertion when we know that the preponderance of searches are for recently inserted data, but we always seek precise performance guarantees—our main