9

Red-Black Trees

IN THIS CHAPTER

- Our Approach to the Discussion
- Balanced and Unbalanced Trees
- Experimenting with the Workshop Applet
- Rotations
- Inserting a New Node
- Deletion
- The Efficiency of Red-Black Trees
- Red-Black Tree Implementation
- Other Balanced Trees

As you learned in Chapter 8, "Binary Trees," ordinary binary search trees offer important advantages as data storage devices: You can quickly search for an item with a given key, and you can also quickly insert or delete an item. Other data storage structures, such as arrays, sorted arrays, and linked lists, perform one or the other of these activities slowly. Thus, binary search trees might appear to be the ideal data storage structure.

Unfortunately, ordinary binary search trees suffer from a troublesome problem. They work well if the data is inserted into the tree in random order. However, they become much slower if data is inserted in already-sorted order (17, 21, 28, 36,…) or inversely sorted order (36, 28, 21, 17,…). When the values to be inserted are already ordered, a binary tree becomes unbalanced. With an unbalanced tree, the ability to quickly find (or insert or delete) a given element is lost.

This chapter explores one way to solve the problem of unbalanced trees: the red-black tree, which is a binary search tree with some added features.

There are other ways to ensure that trees are balanced. We'll mention some at the end of this chapter, and examine several, 2-3-4 trees and 2-3 trees, in Chapter 10, "2-3-4 Trees and External Storage." In fact, as we'll see in that chapter, operations on a 2-3-4 tree correspond in a surprising way to operations on a red-black tree.

Our Approach to the Discussion

We'll explain insertion into red-black trees a little differently than we have explained insertion into other data structures. Red-black trees are not trivial to understand. Because of this and also because of a multiplicity of

symmetrical cases (for left or right children, inside or outside grandchildren, and so on), the actual code is more lengthy and complex than one might expect. It's therefore hard to learn about the algorithm by examining code. Accordingly, there are no listings in this chapter. You can create similar functionality using a 2-3-4 tree with the code shown in Chapter 10. However, the concepts you learn about here will aid your understanding of 2-3-4 trees and are themselves quite interesting.

Conceptual

For our conceptual understanding of red-black trees, we will be aided by the RBTree Workshop applet. We'll describe how you can work in partnership with the applet to insert new nodes into a tree. Including a human into the insertion routine certainly slows it down but also makes it easier for the human to understand how the process works.

Searching works the same way in a red-black tree as it does in an ordinary binary tree. On the other hand, insertion and deletion, while based on the algorithms in an ordinary tree, are extensively modified. Accordingly, in this chapter we'll be concentrating on the insertion process.

Top-Down Insertion

The approach to insertion that we'll discuss is called *top-down* insertion. This means that some structural changes may be made to the tree as the search routine descends the tree looking for the place to insert the node.

Another approach is *bottom-up* insertion. This involves finding the place to insert the node and then working back up through the tree making structural changes. Bottom-up insertion is less efficient because two passes must be made through the tree.

Balanced and Unbalanced Trees

Before we begin our investigation of red-black trees, let's review how trees become unbalanced. Fire up the Binary Tree Workshop applet from Chapter 8 (not this chapter's RBTree applet). Use the Fill button to create a tree with only one node. Then insert a series of nodes whose keys are in either ascending or descending order. The result will be something like that in Figure 9.1.

The nodes arrange themselves in a line with no branches. Because each node is larger than the previously inserted one, every node is a right child, so all the nodes are on one side of the root. The tree is maximally unbalanced. If you inserted items in descending order, every node would be the left child of its parent, and the tree would be unbalanced on the other side.

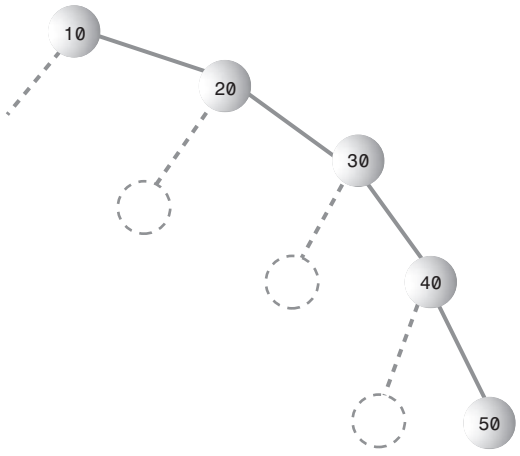

FIGURE 9.1 Items inserted in ascending order.

Degenerates to O(N)

When there are no branches, the tree becomes, in effect, a linked list. The arrangement of data is one-dimensional instead of two-dimensional. Unfortunately, as with a linked list, you must now search through (on the average) half the items to find the one you're looking for. In this situation the speed of searching is reduced to O(N), instead of O(logN) as it is for a balanced tree. Searching through 10,000 items in such an unbalanced tree would require an average of 5,000 comparisons, whereas for a balanced tree with random insertions it requires only 14. For presorted data you might just as well use a linked list in the first place.

Data that's only partly sorted will generate trees that are only partly unbalanced. If you use the Binary Tree Workshop applet from Chapter 8 to attempt to generate trees with 31 nodes, you'll see that some of them are more unbalanced than others, as shown in Figure 9.2.

Although not as bad as a maximally unbalanced tree, this situation is not optimal for searching times.

In the Binary Tree Workshop applet, trees can become partially unbalanced, even with randomly generated data, because the amount of data is so small that even a short run of ordered numbers will have a big effect on the tree. Also, a very small or very large key value can cause an unbalanced tree by not allowing the insertion of many nodes on one side or the other of its node. A root of 3, for example, allows only two more nodes to be inserted to its left.

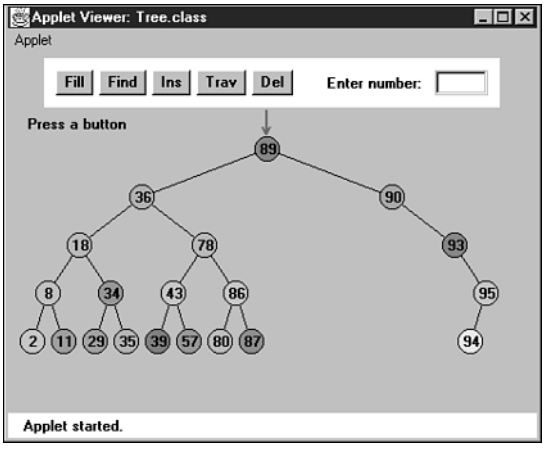

FIGURE 9.2 A partially unbalanced tree.

With a realistic amount of random data, it's not likely a tree would become seriously unbalanced. However, there may be runs of sorted data that will partially unbalance a tree. Searching partially unbalanced trees will take time somewhere between O(N) and O(logN), depending on how badly the tree is unbalanced.

Balance to the Rescue

To guarantee the quick O(log N) search times a tree is capable of, we need to ensure that our tree is always balanced (or at least almost balanced). This means that each node in a tree needs to have roughly the same number of descendents on its left side as it has on its right.

In a red-black tree, balance is achieved during insertion (and also deletion, but we'll ignore that for the moment). As an item is being inserted, the insertion routine checks that certain characteristics of the tree are not violated. If they are, it takes corrective action, restructuring the tree as necessary. By maintaining these characteristics, the tree is kept balanced.

Red-Black Tree Characteristics

What are these mysterious tree characteristics? There are two, one simple and one more complicated:

- The nodes are colored.
- During insertion and deletion, rules are followed that preserve various arrangements of these colors.

Colored Nodes

In a red-black tree, every node is either black or red. These are arbitrary colors; blue and yellow would do just as well. In fact, the whole concept of saying that nodes have "colors" is somewhat arbitrary. Some other analogy could have been used instead: We could say that every node is either heavy or light, or yin or yang. However, colors are convenient labels. A data field, which can be boolean (`isRed`, for example), is added to the node class to embody this color information.

In the RBTree Workshop applet, the red-black characteristic of a node is shown by its border color. The center color, as it was in the Binary Tree Workshop applet in the preceding chapter, is simply a randomly generated data field of the node.

When we speak of a node's color in this chapter, we'll almost always be referring to its red-black border color. In the figures (except the screenshot of Figure 9.3) we'll show black nodes with a solid black border and red nodes with a white border. (Nodes are sometimes shown with no border to indicate that it doesn't matter whether they're black or red.)

Red-Black Rules

When inserting (or deleting) a new node, certain rules, which we call the *red-black rules*, must be followed. If they're followed, the tree will be balanced. Let's look briefly at these rules:

1. Every node is either red or black.
2. The root is always black.
3. If a node is red, its children must be black (although the converse isn't necessarily true).
4. Every path from the root to a leaf, or to a null child, must contain the same number of black nodes.

The "null child" referred to in Rule 4 is a place where a child could be attached to a non-leaf node. In other words, it's the potential left child of a node with a right child, or the potential right child of a node with a left child. This will make more sense as we go along.

The number of black nodes on a path from root to leaf is called the *black height*. Another way to state Rule 4 is that the black height must be the same for all paths from the root to a leaf.

These rules probably seem completely mysterious. It's not obvious how they will lead to a balanced tree, but they do; some very clever people invented them. Copy them onto a sticky note, and keep it on your computer. You'll need to refer to them often in the course of this chapter.

You can see how the rules work by using the RBTree Workshop applet. We'll do some experiments with the applet in a moment, but first you should understand what actions you can take to fix things if one of the red-black rules is broken.

Duplicate Keys

What happens if there's more than one data item with the same key? This presents a slight problem in red-black trees. It's important that nodes with the same key are distributed on both sides of other nodes with the same key. That is, if keys arrive in the order 50, 50, 50, you want the second 50 to go to the right of the first one, and the third 50 to go to the left of the first one. Otherwise, the tree becomes unbalanced.

Distributing nodes with equal keys could be handled by some kind of randomizing process in the insertion algorithm. However, the search process then becomes more complicated if all items with the same key must be found.

It's simpler to outlaw items with the same key. In this discussion we'll assume duplicates aren't allowed.

Fixing Rule Violations

Suppose you see (or are told by the applet) that the color rules are violated. How can you fix things so your tree is in compliance? There are two, and only two, possible actions you can take:

- You can change the colors of nodes.
- You can perform rotations.

In the applet, changing the color of a node means changing its red-black border color (not the center color). A rotation is a rearrangement of the nodes that, one hopes, leaves the tree more balanced.

At this point such concepts probably seem very abstract, so let's become familiar with the RBTree Workshop applet, which can help to clarify things.

Using the RBTree Workshop Applet

Figure 9.3 shows what the RBTree Workshop applet looks like after some nodes have been inserted. (It may be hard to tell the difference between red and black node borders in the figure, but they should be clear on a color monitor.)

There are quite a few buttons in the RBTree applet. We'll briefly review what they do, although at this point some of the descriptions may be a bit puzzling.