12.2 Red-Black Trees

A historically popular alternative to the AVL tree is the **red-black tree**. Operations on red-black trees take $O(\log N)$ time in the worst case, and, as we will see, a careful nonrecursive implementation (for insertion) can be done relatively effortlessly (compared with AVL trees).

A red-black tree is a binary search tree with the following coloring properties:

1. Every node is colored either red or black.
2. The root is black.
3. If a node is red, its children must be black.
4. Every path from a node to a `null` reference must contain the same number of black nodes.

A consequence of the coloring rules is that the height of a red-black tree is at most $2\log(N + 1)$. Consequently, searching is guaranteed to be a logarithmic operation. Figure 12.9 shows a red-black tree. Red nodes are shown with double circles.

The difficulty, as usual, is inserting a new item into the tree. The new item, as usual, is placed as a leaf in the tree. If we color this item black, then we are certain to violate condition 4, because we will create a longer path of black nodes. Thus the item must be colored red. If the parent is black, we are done. If the parent is already red, then we will violate condition 3 by having consecutive red nodes. In this case, we have to adjust the tree to ensure that condition 3 is enforced (without introducing a violation of condition 4). The basic operations that are used to do this are color changes and tree rotations.

12.2.1 Bottom-Up Insertion

As we have already mentioned, if the parent of the newly inserted item is black, we are done. Thus insertion of 25 into the tree in Figure 12.9 is trivial.

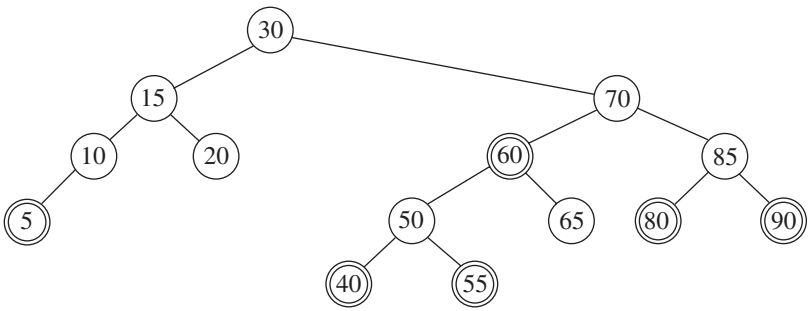

Figure 12.9 Example of a red-black tree (insertion sequence is: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55)

There are several cases (each with a mirror image symmetry) to consider if the parent is red. First, suppose that the sibling of the parent is black (we adopt the convention that `null` nodes are black). This would apply for an insertion of 3 or 8, but not for the insertion of 99. Let *X* be the newly added leaf, *P* be its parent, *S* be the sibling of the parent (if it exists), and *G* be the grandparent. Only *X* and *P* are red in this case; *G* is black, because otherwise there would be two consecutive red nodes *prior* to the insertion, in violation of red-black rules. Adopting the splay tree terminology, *X*, *P*, and *G* can form either a *zig-zig* chain or a *zig-zag* chain (in either of two directions). Figure 12.10 shows how we can rotate the tree for the case where *P* is a left child (note there is a symmetric case). Even though *X* is a leaf, we have drawn a more general case that allows *X* to be in the middle of the tree. We will use this more general rotation later.

The first case corresponds to a single rotation between *P* and *G*, and the second case corresponds to a double rotation, first between *X* and *P* and then between *X* and *G*. When we write the code, we have to keep track of the parent, the grandparent, and, for reattachment purposes, the great-grandparent.

In both cases, the subtree's new root is colored black, and so even if the original great-grandparent was red, we removed the possibility of two consecutive red nodes. Equally important, the number of black nodes on the paths into *A*, *B*, and *C* has remained unchanged as a result of the rotations.

So far so good. But what happens if *S* is red, as is the case when we attempt to insert 79 in the tree in Figure 12.9? In that case, initially there is one black node on the path from the subtree's root to *C*. After the rotation, there must still be only one black node. But in both cases, there are three nodes (the new root, *G*, and *S*) on the path to *C*. Since only one may be black, and since we cannot have consecutive red nodes, it follows that we'd have to color both *S* and the subtree's new root red, and *G* (and our fourth node) black. That's great, but what happens if the great-grandparent is also red? In that case, we

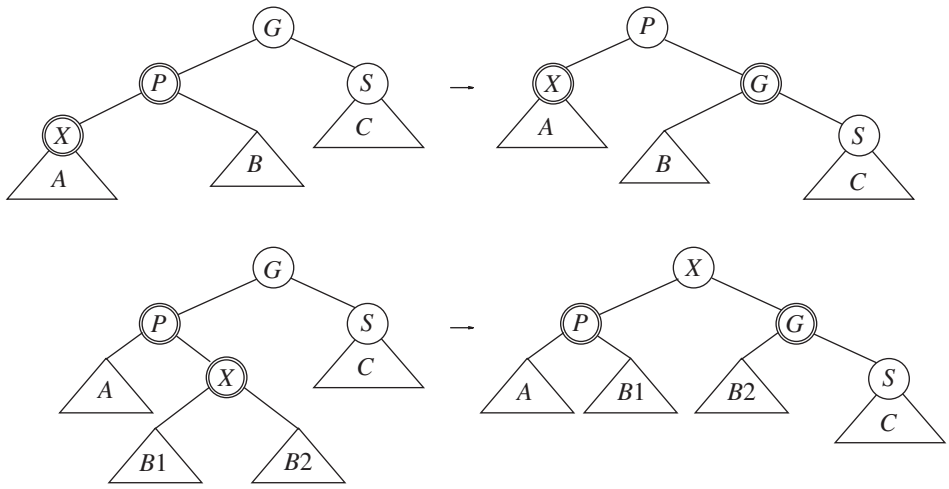

Figure 12.10 *Zig* rotation and *zig-zag* rotation work if *S* is black

could percolate this procedure up toward the root as is done for B-trees and binary heaps, until we no longer have two consecutive red nodes, or we reach the root (which will be recolored black).

12.2.2 Top-Down Red-Black Trees

Implementing the percolation would require maintaining the path using a stack or parent links. We saw that splay trees are more efficient if we use a top-down procedure, and it turns out that we can apply a top-down procedure to red-black trees that guarantees that *S* won't be red.

The procedure is conceptually easy. On the way down, when we see a node *X* that has two red children, we make *X* red and the two children black. (If *X* is the root, after the color flip it will be red but can be made black immediately to restore property 2.) Figure 12.11 shows this color flip. This will induce a red-black violation only if *X*'s parent *P* is also red. But in that case, we can apply the appropriate rotations in Figure 12.10. What if *X*'s parent's sibling is red? This possibility has been removed by our actions on the way down, and so *X*'s parent's sibling can't be red! Specifically, if on the way down the tree we see a node *Y* that has two red children, we know that *Y*'s grandchildren must be black, and that since *Y*'s children are made black too, even after the rotation that may occur, we won't see another red node for two levels. Thus when we see *X*, if *X*'s parent is red, it is not possible for *X*'s parent's sibling to be red also.

As an example, suppose we want to insert 45 into the tree in Figure 12.9. On the way down the tree, we see node 50, which has two red children. Thus, we perform a color flip, making 50 red, and 40 and 55 black. Now 50 and 60 are both red. We perform the single rotation between 60 and 70, making 60 the black root of 30's right subtree, and 70 and 50 both red. We then continue, performing an identical action if we see other nodes on the path that contain two red children. When we get to the leaf, we insert 45 as a red node, and since the parent is black, we are done. The resulting tree is shown in Figure 12.12.

As Figure 12.12 shows, the red-black tree that results is frequently very well balanced. Experiments suggest that the average red-black tree is about as deep as an average AVL tree and that, consequently, the searching times are typically near optimal. The advantage of red-black trees is the relatively low overhead required to perform insertion, and the fact that in practice rotations occur relatively infrequently.

An actual implementation is complicated not only by the host of possible rotations, but also by the possibility that some subtrees (such as 10's right subtree) might be empty, and the special case of dealing with the root (which among other things, has no parent). Thus, we use two sentinel nodes: one for the root, and `nullNode`, which indicates a `null` reference,

Figure 12.11 Color flip: Only if *X*'s parent is red do we continue with a rotation

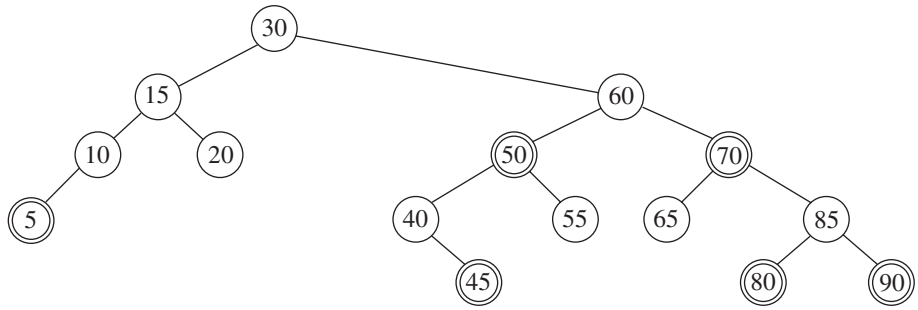

Figure 12.12 Insertion of 45 into Figure 12.9

as it did for splay trees. The root sentinel will store the key $-\infty$ and a right link to the real root. Because of this, the searching and printing procedures need to be adjusted. The recursive routines are trickiest. Figure 12.13 shows how the inorder traversal is rewritten.

Figure 12.14 shows the `RedBlackTree` skeleton (omitting the methods), along with the constructors. Next, Figure 12.15 shows the routine to perform a single rotation. Because

```
1       /**
2        * Print the tree contents in sorted order.
3        */
4       public void printTree( )
5       {
6           if( isEmpty( ) )
7               System.out.println( "Empty tree" );
8           else
9               printTree( header.right );
10      }
11
12      /**
13       * Internal method to print a subtree in sorted order.
14       * @param t the node that roots the subtree.
15       */
16      private void printTree( RedBlackNode<AnyType> t )
17      {
18          if( t != nullNode )
19          {
20              printTree( t.left );
21              System.out.println( t.element );
22              printTree( t.right );
23          }
24      }
```

Figure 12.13 Inorder traversal for tree and two sentinels

```
1  public class RedBlackTree<AnyType extends Comparable<? super AnyType>>
2  {
3      /**
4       * Construct the tree.
5       */
6      public RedBlackTree( )
7      {
8          nullNode = new RedBlackNode<>( null );
9          nullNode.left = nullNode.right = nullNode;
10         header      = new RedBlackNode<>( null );
11         header.left = header.right = nullNode;
12     }
13
14     private static class RedBlackNode<AnyType>
15     {
16             // Constructors
17         RedBlackNode( AnyType theElement )
18           { this( theElement, null, null ); }
19
20         RedBlackNode( AnyType theElement, RedBlackNode<AnyType> lt, RedBlackNode<AnyType> rt )
21           { element = theElement; left = lt; right = rt; color = RedBlackTree.BLACK; }
22
23         AnyType               element;    // The data in the node
24         RedBlackNode<AnyType> left;       // Left child
25         RedBlackNode<AnyType> right;      // Right child
26         int                   color;      // Color
27     }
28
29     private RedBlackNode<AnyType> header;
30     private RedBlackNode<AnyType> nullNode;
31
32     private static final int BLACK = 1;    // BLACK must be 1
33     private static final int RED   = 0;
34 }
```

Figure 12.14 Class skeleton and initialization routines

the resultant tree must be attached to a parent, `rotate` takes the parent node as a parameter. Rather than keeping track of the type of rotation as we descend the tree, we pass `item` as a parameter. Since we expect very few rotations during the insertion procedure, it turns out that it is not only simpler, but actually faster, to do it this way. `rotate` simply returns the result of performing an appropriate single rotation.

Finally, we provide the insertion procedure in Figure 12.16. The routine `handleReorient` is called when we encounter a node with two red children, and also when we insert a leaf.

```
1       /**
2        * Internal routine that performs a single or double rotation.
3        * Because the result is attached to the parent, there are four cases.
4        * Called by handleReorient.
5        * @param item the item in handleReorient.
6        * @param parent the parent of the root of the rotated subtree.
7        * @return the root of the rotated subtree.
8        */
9       private RedBlackNode<AnyType> rotate( AnyType item, RedBlackNode<AnyType> parent )
10      {
11          if( compare( item, parent ) < 0 )
12              return parent.left = compare( item, parent.left ) < 0 ?
13                  rotateWithLeftChild( parent.left )  :  // LL
14                  rotateWithRightChild( parent.left ) ;  // LR
15          else
16              return parent.right = compare( item, parent.right ) < 0 ?
17                  rotateWithLeftChild( parent.right ) :  // RL
18                  rotateWithRightChild( parent.right );  // RR
19      }
20
21      /**
22       * Compare item and t.element, using compareTo, with
23       * caveat that if t is header, then item is always larger.
24       * This routine is called if it is possible that t is header.
25       * If it is not possible for t to be header, use compareTo directly.
26       */
27      private final int compare( AnyType item, RedBlackNode<AnyType> t )
28      {
29          if( t == header )
30              return 1;
31          else
32              return item.compareTo( t.element );
33      }
```

Figure 12.15 `rotate` method

The most tricky part is the observation that a double rotation is really two single rotations, and is done only when branching to *X* (represented in the `insert` method by `current`) takes opposite directions. As we mentioned in the earlier discussion, `insert` must keep track of the parent, grandparent, and great-grandparent as the tree is descended. Since these are shared with `handleReorient`, we make these class members. Note that after a rotation, the values stored in the grandparent and great-grandparent are no longer correct. However, we are assured that they will be restored by the time they are next needed.

```
1           // Used in insert routine and its helpers
2       private RedBlackNode<AnyType> current;
3       private RedBlackNode<AnyType> parent;
4       private RedBlackNode<AnyType> grand;
5       private RedBlackNode<AnyType> great;
6
7       /**
8        * Internal routine that is called during an insertion
9        * if a node has two red children. Performs flip and rotations.
10       * @param item the item being inserted.
11       */
12      private void handleReorient( AnyType item )
13      {
14              // Do the color flip
15          current.color = RED;
16          current.left.color = BLACK;
17          current.right.color = BLACK;
18
19          if( parent.color == RED )   // Have to rotate
20          {
21              grand.color = RED;
22              if( ( compare( item, grand ) < 0 ) !=
23                  ( compare( item, parent ) < 0 ) )
24                  parent = rotate( item, grand );  // Start dbl rotate
25              current = rotate( item, great );
26              current.color = BLACK;
27          }
28          header.right.color = BLACK; // Make root black
29      }
30
31      /**
32       * Insert into the tree.
33       * @param item the item to insert.
34       */
35      public void insert( AnyType item )
36      {
37          current = parent = grand = header;
38          nullNode.element = item;
39
40          while( compare( item, current ) != 0 )
41          {
42              great = grand; grand = parent; parent = current;
43              current = compare( item, current ) < 0 ? current.left : current.right;
44
```

Figure 12.16 Insertion procedure

```
45                  // Check if two red children; fix if so
46              if( current.left.color == RED && current.right.color == RED )
47                   handleReorient( item );
48          }
49
50              // Insertion fails if already present
51          if( current != nullNode )
52              return;
53          current = new RedBlackNode<>( item, nullNode, nullNode );
54
55              // Attach to parent
56          if( compare( item, parent ) < 0 )
57              parent.left = current;
58          else
59              parent.right = current;
60          handleReorient( item );
61      }
```

Figure 12.16 *(continued)*

12.2.3 Top-Down Deletion

Deletion in red-black trees can also be performed top-down. Everything boils down to being able to delete a leaf. This is because to delete a node that has two children, we replace it with the smallest node in the right subtree; that node, which must have at most one child, is then deleted. Nodes with only a right child can be deleted in the same manner, while nodes with only a left child can be deleted by replacement with the largest node in the left subtree, and subsequent deletion of that node. Note that for red-black trees, we don't want to use the strategy of bypassing for the case of a node with one child because that may connect two red nodes in the middle of the tree, making enforcement of the red-black condition difficult.

Deletion of a red leaf is, of course, trivial. If a leaf is black, however, the deletion is more complicated because removal of a black node will violate condition 4. The solution is to ensure during the top-down pass that the leaf is red.

Throughout this discussion, let X be the current node, T be its sibling, and P be their parent. We begin by coloring the root sentinel red. As we traverse down the tree, we attempt to ensure that X is red. When we arrive at a new node, we are certain that P is red (inductively, by the invariant we are trying to maintain), and that X and T are black (because we can't have two consecutive red nodes). There are two main cases.

First, suppose X has two black children. Then there are three subcases, which are shown in Figure 12.17. If T also has two black children, we can flip the colors of X, T, and P to maintain the invariant. Otherwise, one of T's children is red. Depending on which

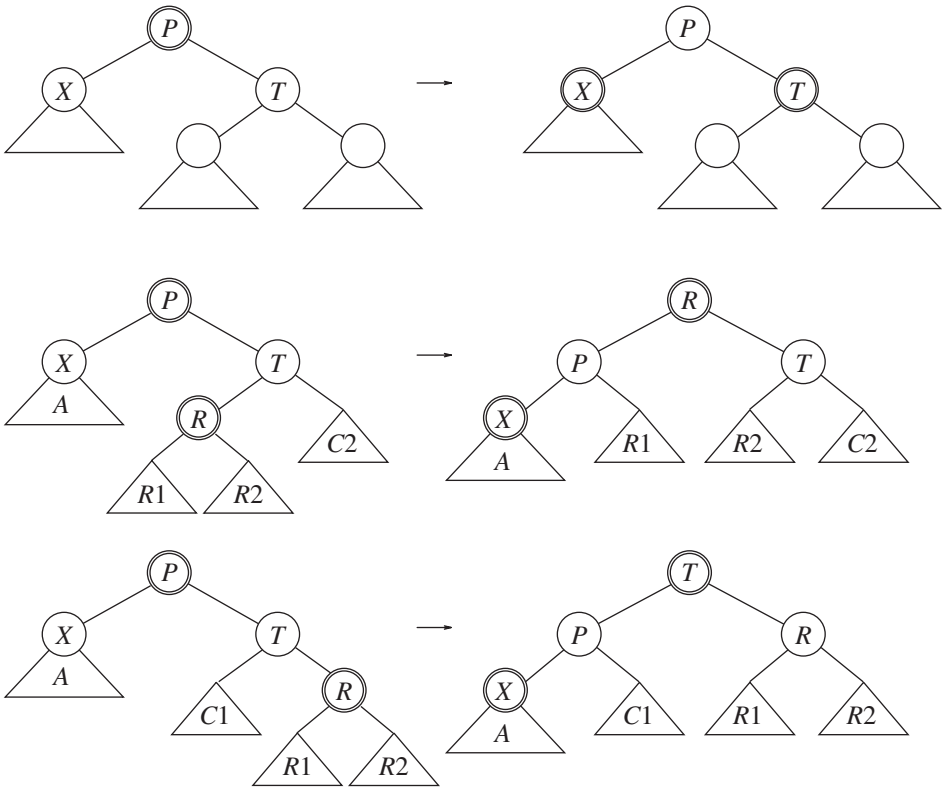

Figure 12.17 Three cases when *X* is a left child and has two black children

one it is,[3] we can apply the rotation shown in the second and third cases of Figure 12.17. Note carefully that this case will apply for the leaf, because `nullNode` is considered to be black.

Otherwise, one of *X*'s children is red. In this case, we fall through to the next level, obtaining new *X*, *T*, and *P*. If we're lucky, *X* will land on the red child, and we can continue onward. If not, we know that *T* will be red, and *X* and *P* will be black. We can rotate *T* and *P*, making *X*'s new parent red; *X* and its grandparent will, of course, be black. At this point we can go back to the first main case.

[3] If both children are red, we can apply either rotation. As usual, there are symmetric rotations for the case when *X* is a right child that are not shown.