10.5.2 C++ Implementation of a Red-Black Tree

In this section, we discuss a C++ implementation of the dictionary ADT by means of a red-black tree. It is interesting to note that the C++ Standard Template Library uses a red-black tree in its implementation of its classes map and multimap. The difference between the two is similar to the difference between our map and dictionary ADTs. The STL map class does not allow entries with duplicate keys, whereas the STL multimap does. There is a significant difference, however, in the behavior of the map's insert(k,x) function and our map's put(k,x) function. If the key k is not present, both functions insert the new entry (k,x) in the map. If the key is already present, the STL map simply ignores the request, and the current entry is unchanged. In contrast, our put function replaces the existing value with the new value x. The implementation presented in this section allows for multiple keys.

We present the major portions of the implementation in this section. To keep the presentation concise, we have omitted the implementations of a number of simpler utility functions.

We begin by presenting the enhanced entry class, called RBEntry. It is derived from the entry class of Code Fragment 10.3. It inherits the key and value members, and it defines a member variable *col*, which stores the color of the node. The color is either RED or BLACK. It provides member functions for accessing and setting this value. These functions have been protected, so a user cannot access them, but RBTree can.

```
enum Color {RED, BLACK};                          // node colors

template <typename E>
class RBEntry : public E {                        // a red-black entry
private:
  Color col;                                      // node color
protected:                                        // local types
  typedef typename E::Key K;                      // key type
  typedef typename E::Value V;                    // value type
  Color color() const { return col; }             // get color
  bool isRed() const { return col == RED; }
  bool isBlack() const { return col == BLACK; }
  void setColor(Color c) { col = c; }
public:                                           // public functions
  RBEntry(const K& k = K(), const V& v = V())    // constructor
    : E(k,v), col(BLACK) { }
  friend class RBTree<E>;                         // allow RBTree access
};
```

Code Fragment 10.19: A key-value entry for class RBTree, containing the associated node's color.

In Code Fragment 10.20, we present the class definition for RBTree. The declaration is almost entirely analogous to that of AVLTree, except that the utility functions used to maintain the structure are different. We have chosen to present only the two most interesting utility functions, remedyDoubleRed and remedyDoubleBlack. The meanings of most of the omitted utilities are easy to infer. (For example hasTwoExternalChildren(v) determines whether a node v has two external children.)

```
template <typename E>                                   // a red-black tree
class RBTree : public SearchTree< RBEntry<E> > {
public:                                                 // public types
  typedef RBEntry<E> RBEntry;                           // an entry
  typedef typename SearchTree<RBEntry>::Iterator Iterator; // an iterator
protected:                                              // local types
  typedef typename RBEntry::Key K;                      // a key
  typedef typename RBEntry::Value V;                    // a value
  typedef SearchTree<RBEntry> ST;                       // a search tree
  typedef typename ST::TPos TPos;                       // a tree position
public:                                                 // public functions
  RBTree();                                             // constructor
  Iterator insert(const K& k, const V& x);              // insert (k,x)
  void erase(const K& k) throw(NonexistentElement);  // remove key k entry
  void erase(const Iterator& p);                        // remove entry at p
protected:                                              // utility functions
  void remedyDoubleRed(const TPos& z);                  // fix double-red z
  void remedyDoubleBlack(const TPos& r);                // fix double-black r
  // ...(other utilities omitted)
};
```

Code Fragment 10.20: Class RBTree, which implements a dictionary ADT using a red-black tree.

We first discuss the implementation of the function insert(k,x), which is given in Code Fragment 10.21. We invoke the inserter utility function of SearchTree, which returns the position of the inserted node. If this node is the root of the search tree, we set its color to black. Otherwise, we set its color to red and check whether restructuring is needed by invoking remedyDoubleRed.

This latter utility performs the necessary checks and restructuring presented in the discussion of insertion in Section 10.5.1. Let z denote the location of the newly inserted node. If both z and its parent are red, we need to remedy the situation. To do so, we consider two cases. Let v denote z's parent and let w be v's sibling. If w is black, we fall under Case 1 of the insertion update procedure. We apply restructuring at z. The top vertex of the resulting subtree, denoted by v, is set to black, and its two children are set to red.

On the other hand, if w is red, then we fall under Case 2 of the update procedure.

We resolve the situation by coloring both v and its sibling w black. If their common parent is not the root, we set its color to red. This may induce another double-red problem at v's parent u, so we invoke the function recursively on u.

```
/* RBTree⟨E⟩ :: */                                    // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                          // insert in base tree
    if (v == ST::root())
      setBlack(v);                                    // root is always black
    else {
      setRed(v);
      remedyDoubleRed(v);                             // rebalance if needed
    }
    return Iterator(v);
  }

/* RBTree⟨E⟩ :: */                                    // fix double-red z
  void remedyDoubleRed(const TPos& z) {
    TPos v = z.parent();                              // v is z's parent
    if (v == ST::root() || v->isBlack()) return;      // v is black, all ok
                                                      // z, v are double-red
    if (sibling(v)->isBlack()) {                      // Case 1: restructuring
      v = restructure(z);
      setBlack(v);                                    // top vertex now black
      setRed(v.left()); setRed(v.right());            // set children red
    }
    else {                                            // Case 2: recoloring
      setBlack(v); setBlack(sibling(v));              // set v and sibling black
      TPos u = v.parent();                            // u is v's parent
      if (u == ST::root()) return;
      setRed(u);                                      // make u red
      remedyDoubleRed(u);                             // may need to fix u now
    }
  }
```

Code Fragment 10.21: The functions related to insertion for class RBTree. The function insert invokes the inserter utility function, which was given in Code Fragment 10.10.

Finally, in Code Fragment 10.22, we present the implementation of the removal function for the red-black tree. (We have omitted the simpler iterator-based erase function.) The removal follows the process discussed in Section 10.5.1. We first search for the key to be removed, and generate an exception if it is not found. Otherwise, we invoke the eraser utility of class SearchTree, which returns the position of the node r that replaced the deleted node. If either r or its former parent was red, we color r black and we are done. Otherwise, we face a potential double-black problem. We handle this by invoking the function remedyDoubleBlack.

```
/* RBTree⟨E⟩ :: */                                  // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos u = finder(k, ST::root());                 // find the node
    if (Iterator(u) == ST::end())
      throw NonexistentElement("Erase of nonexistent");
    TPos r = eraser(u);                             // remove u
    if (r == ST::root() || r->isRed() || wasParentRed(r))
      setBlack(r);                                  // fix by color change
    else                                            // r, parent both black
      remedyDoubleBlack(r);                         // fix double-black r
  }

/* RBTree⟨E⟩ :: */                                  // fix double-black r
  void remedyDoubleBlack(const TPos& r) {
    TPos x = r.parent();                            // r's parent
    TPos y = sibling(r);                            // r's sibling
    if (y->isBlack()) {
      if (y.left()->isRed() || y.right()->isRed()) {    // Case 1: restructuring
                                                    // z is y's red child
        TPos z = (y.left()->isRed() ? y.left() : y.right());
        Color topColor = x->color();                // save top vertex color
        z = restructure(z);                         // restructure x,y,z
        setColor(z, topColor);                      // give z saved color
        setBlack(r);                                // set r black
        setBlack(z.left()); setBlack(z.right());    // set z's children black
      }
      else {                                        // Case 2: recoloring
        setBlack(r); setRed(y);                     // r=black, y=red
        if (x->isBlack() && !(x == ST::root()))
        remedyDoubleBlack(x);                       // fix double-black x
        setBlack(x);
      }
    }
    else {                                          // Case 3: adjustment
      TPos z = (y == x.right() ? y.right() : y.left());  // grandchild on y's side
      restructure(z);                               // restructure x,y,z
      setBlack(y); setRed(x);                       // y=black, x=red
      remedyDoubleBlack(r);                         // fix r by Case 1 or 2
    }
  }
```

Code Fragment 10.22: The functions related to removal for class RBTree. The function erase invokes the eraser utility function, which was given in Code Fragment 10.11.