10.2.2 C++ Implementation of an AVL Tree

Let us now turn to the implementation details and analysis of using an AVL tree T with n internal nodes to implement an ordered dictionary of n entries. The insertion and removal algorithms for T require that we are able to perform trinode restructurings and determine the difference between the heights of two sibling nodes. Regarding restructurings, we now need to make sure our underlying implementation of a binary search tree includes the method restructure(x), which performs a trinode restructuring operation (Code Fragment 10.12). (We do not provide an implementation of this function, but it is a straightforward addition to the linked binary tree class given in Section 7.3.4.) It is easy to see that a restructure operation can be performed in $O(1)$ time if T is implemented with a linked structure. We assume that the SearchTree class includes this function.

Regarding height information, we have chosen to store the height of each internal node, v, explicitly in each node. Alternatively, we could have stored the ***balance factor*** of v at v, which is defined as the height of the left child of v minus the height of the right child of v. Thus, the balance factor of v is always equal to -1, 0, or 1, except during an insertion or removal, when it may become ***temporarily*** equal to -2 or $+2$. During the execution of an insertion or removal, the heights and balance factors of $O(\log n)$ nodes are affected and can be maintained in $O(\log n)$ time.

In order to store the height information, we derive a subclass, called AVLEntry, from the standard entry class given earlier in Code Fragment 10.3. It is templated with the base entry type, from which it inherits the key and value members. It defines a member variable ht, which stores the height of the subtree rooted at the associated node. It provides member functions for accessing and setting this value. These functions are protected, so that a user cannot access them, but AVLTree can.

```
template <typename E>
class AVLEntry : public E {                          // an AVL entry
private:
  int ht;                                            // node height
protected:                                           // local types
  typedef typename E::Key K;                         // key type
  typedef typename E::Value V;                       // value type
  int height() const { return ht; }                  // get height
  void setHeight(int h) { ht = h; }                  // set height
public:                                              // public functions
  AVLEntry(const K& k = K(), const V& v = V())  // constructor
      : E(k,v), ht(0) { }
  friend class AVLTree<E>;                           // allow AVLTree access
};
```

Code Fragment 10.13: An enhanced key-value entry for class AVLTree, containing the height of the associated node.

In Code Fragment 10.14, we present the class definition for AVLTree. This class is derived from the class SearchTree, but using our enhanced AVLEntry in order to maintain height information for the nodes of the tree. The class defines a number of typedef shortcuts for referring to entities such as keys, values, and tree positions. The class declares all the standard dictionary public member functions. At the end, it also defines a number of protected utility functions, which are used in maintaining the AVL tree balance properties.

```
template <typename E>                                    // an AVL tree
class AVLTree : public SearchTree< AVLEntry<E> > {
public:                                                  // public types
  typedef AVLEntry<E> AVLEntry;                          // an entry
  typedef typename SearchTree<AVLEntry>::Iterator Iterator; // an iterator
protected:                                               // local types
  typedef typename AVLEntry::Key K;                      // a key
  typedef typename AVLEntry::Value V;                    // a value
  typedef SearchTree<AVLEntry> ST;                       // a search tree
  typedef typename ST::TPos TPos;                        // a tree position
public:                                                  // public functions
  AVLTree();                                             // 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
  int height(const TPos& v) const;                       // node height utility
  void setHeight(TPos v);                                // set height utility
  bool isBalanced(const TPos& v) const;                  // is v balanced?
  TPos tallGrandchild(const TPos& v) const;              // get tallest grandchild
  void rebalance(const TPos& v);                         // rebalance utility
};
```

Code Fragment 10.14: Class AVLTree, an AVL tree implementation of a dictionary.

Next, in Code Fragment 10.15, we present the constructor and height utility function. The constructor simply invokes the constructor for the binary search tree, which creates a tree having no entries. The function height returns the height of a node, by extracting the height information from the AVLEntry. We employ the condensed function notation that we introduced in Section 9.2.7.

```
/* AVLTree⟨E⟩ :: */                                      // constructor
  AVLTree() : ST() { }

/* AVLTree⟨E⟩ :: */                                      // node height utility
  int height(const TPos& v) const
    { return (v.isExternal() ? 0 : v−>height()); }
```

Code Fragment 10.15: The constructor for class AVLTree and a utility for extracting heights.

In Code Fragment 10.16, we present a few utility functions needed for maintaining the tree's balance. The function setHeight sets the height information for a node as one more than the maximum of the heights of its two children. The function isBalanced determines whether a node satisfies the AVL balance condition, by checking that the height difference between its children is at most 1. Finally, the function tallGrandchild determines the tallest grandchild of a node. Recall that this procedure is needed by the removal operation to determine the node to which the restructuring operation will be applied.

```
/* AVLTree<E> :: */                                // set height utility
  void setHeight(TPos v) {
    int hl = height(v.left());
    int hr = height(v.right());
    v->setHeight(1 + std::max(hl, hr));             // max of left & right
  }

/* AVLTree<E> :: */                                // is v balanced?
  bool isBalanced(const TPos& v) const {
    int bal = height(v.left()) − height(v.right());
    return ((−1 <= bal) && (bal <= 1));
  }

/* AVLTree<E> :: */                                // get tallest grandchild
  TPos tallGrandchild(const TPos& z) const {
    TPos zl = z.left();
    TPos zr = z.right();
    if (height(zl) >= height(zr))                   // left child taller
      if (height(zl.left()) >= height(zl.right()))
        return zl.left();
      else
        return zl.right();
    else                                            // right child taller
      if (height(zr.right()) >= height(zr.left()))
        return zr.right();
      else
        return zr.left();
  }
```

Code Fragment 10.16: Some utility functions used for maintaining balance in the AVL tree.

Next, we present the principal function for rebalancing the AVL tree after an insertion or removal. The procedure starts at the node v affected by the operation. It then walks up the tree to the root level. On visiting each node z, it updates z's height information (which may have changed due to the update operation) and checks whether z is balanced. If not, it finds z's tallest grandchild, and applies the restructuring operation to this node. Since heights may have changed as a result, it updates the height information for z's children and itself.

```
/* AVLTree⟨E⟩ :: */                                  // rebalancing utility
  void rebalance(const TPos& v) {
    TPos z = v;
    while (!(z == ST::root())) {                     // rebalance up to root
      z = z.parent();
      setHeight(z);                                  // compute new height
      if (!isBalanced(z)) {                          // restructuring needed
        TPos x = tallGrandchild(z);
        z = restructure(x);                          // trinode restructure
        setHeight(z.left());                         // update heights
        setHeight(z.right());
        setHeight(z);
      }
    }
  }
```

Code Fragment 10.17: Rebalancing the tree after an update operation.

Finally, in Code Fragment 10.18, we present the functions for inserting and erasing keys. (We have omitted the iterator-based erase function, since it is very simple.) Each invokes the associated utility function (inserter or eraser, respectively) from the base class SearchTree. Each then invokes rebalance to restore balance to the tree.

```
/* AVLTree⟨E⟩ :: */                                  // insert (k,x)
  Iterator insert(const K& k, const V& x) {
    TPos v = inserter(k, x);                         // insert in base tree
    setHeight(v);                                    // compute its height
    rebalance(v);                                    // rebalance if needed
    return Iterator(v);
  }

/* AVLTree⟨E⟩ :: */                                  // remove key k entry
  void erase(const K& k) throw(NonexistentElement) {
    TPos v = finder(k, ST::root());                  // find in base tree
    if (Iterator(v) == ST::end())                    // not found?
      throw NonexistentElement("Erase of nonexistent");
    TPos w = eraser(v);                              // remove it
    rebalance(w);                                    // rebalance if needed
  }
```

Code Fragment 10.18: The insertion and erasure functions.