Chapter 6

Trees

In a hash table, the items are not stored in any particular order in the table. This is fine for implementing Sets and Maps, since for those abstract data types, the only thing that matters is the presence or absence of an item, and hash tables are designed to check for this very efficiently. Often, however, the elements of a Set or the keys of a Map have a natural ordering and there are times when it is important to be able to access the items in order. For example, we might want to list the elements of a set of strings in alphabetical order. Or, for a map in which the keys are times and the values are events, we might want to list all the events that occur in a given time period. Such operations are not supported by hash tables.

The abstract data types *SortedSet* and *SortedMap* are designed to fix these problems.[1] In a SortedSet, the elements can be accessed in increasing order. In a SortedMap, the key/value pairs can be accessed in order of increasing key.[2]

For accessing items in order, we will need something like the *position* types that were associated with the *List* ADT in Chapter 2. In this case, we will use the term *Enumerator*.[3] An Enumerator is a kind of generalized pointer that points to an item in a set or map. The Enumerator type has an operation that advances the Enumerator from one item to the next item in sorted order. In analogy to a null pointer, we will say that a *null Enumerator* is one that does not point to any item. This can happen, for example, if an item is advanced past the last item in a set. (However, you should remember that a null Enumerator is not necessarily implemented as an actual null pointer.)

Definition 6.1. Let *BaseType* be a data type whose values are ordered by the $<$ operator. The abstract data type **SortedSet of BaseType** is an extension of the *Set Of BaseType* ADT, with an associated *Enumerator* type and the following new operations:

> **begin():** Returns a value of type *Enumerator* that points to the smallest element of the set (in the ordering given by $<$). If the set is empty, then a null Enumerator is returned.

[1] Note that the *set* and *map* template classes in the C++ standard template library are actually implementations of what I am calling *SortedSet* and *SortedMap*. In particular, *set* and *map* are not implemented as hash tables.

[2] We will assume in this chapter that the ordering is defined by the $<$ operator, as it is for numbers and for C++ strings.

[3] Both *position* and *Enumerator* are similar in concept to C++ iterators, although the details are quite different.

seek(x): Returns a value of type *Enumerator* that points to the first element of the set that is greater than or equal to x, where x is of type *BaseType*. If no such item exists, then a null Enumerator is returned.

The *Enumerator* type has the following operations:

more(): Returns a boolean value that is true if the Enumerator points to an item in the set, and is false if it is a null Enumerator.

item(): Returns the item that the Enumerator points to. It is an error to call this if the Enumerator is null.

next(): Advances the Enumerator to point to the next item in increasing order. It is an error to call this if the Enumerator is null.

erase(): Removes the item to which this Enumerator points from the set, and advances the Enumerator to point to the next item in the set. Has no effect if the Enumerator is null.

The operations of the *Enumerator* type make it easy to access the elements of a set in order. The *begin* and *seek* operators make it possible to choose any starting point for this access. For example, if *set* is a SortedSet of integers, we could print out all the elements of *set* with:

```
iterator p = set.begin();
while (p.more()) {
   cout << p.item() << endl;
   p.next();
}
```

Or to delete all integers in the range 100 to 200 from the set:

```
Enumerator p = set.seek(100);
while (p.more() && p.item() <= 200) {
   p.erase();
}
```

A *SortedMap of (KeyType,ValueType)* would be defined in almost exactly the same way. The *seek* operation would take a parameter of type *KeyType*. An *Enumerator* would point to a (key,value) pair. And the *item*() operation in the *Enumerator* class would be replaced by two operations, *key*() and *value*(), that return the key and the value from the current pair.

A simple implementation of SortedSet or SortedMap would store the items in a sorted list. In such an implementation, however, some operations would have run time $\Theta(n)$. We can do better than that. There are implementations in which all operations have worst case run times that are at most $\Theta(\log(n))$. These efficient implementations use *balanced trees*. In this chapter, we will see how SortedSet and SortedMap can be implemented as binary trees. The trees we use will *not* necessarily be balanced, and so they do not guarantee that operations on them are efficient. Later in the chapter, we will look at what it means for a tree to be balanced, and why it is important. And in the next chapter, we will look at a type of balanced tree that does guarantee efficiency of operations.

6.1 Binary Sort Trees

Recall that a binary tree is a concrete data structure that consists of zero or more *nodes*, where each node contains a *left* and *right* pointer (as well as other fields). These pointers, when non-null, point to the left and right *child nodes* of the node. A node is said to be the *parent* of its children.

An *empty tree* contains no nodes. If a tree is non-empty, then it contains exactly one node that has no parent node. This node is called the *root* node of the tree. Every other node in the tree is a descendent of the root node, and for any given node, there is a unique path in the tree from the root to that node. The length of this path (that is, the number of pointers that must be followed to get from the root to the node) is called the *height* of the node, and the maximal height of any node is called the *height* of the tree.

If we consider any node in a binary tree, the left child of that node together with all the descendents of the left child form a tree that is called the *left subtree* of the node. Similarly, the *right subtree* of the node consists of its right child and all descendents of its right child. Either or both subtrees can be empty.

To be more definite, we will concentrate on using binary trees to implement SortedMaps. So, we assume that, in addition to the left and right pointers, each node contains a key and a value. SortedSets would be similar, except that the node would contain an element of the set instead of a (key,value) pair. Furthermore, to make certain operations more efficient, we will assume that each node contains a pointer to its parent node. (For the root node, the parent pointer is null.) So, we can define the type:

```
struct Node {
   KeyType key;
   ValueType value;
   Node *left;
   Node *right;
   Node *parent;
   Node(KeyType k, ValueType v) { // A constructor, for convenience
      key = k;
      value = v;
      left = right = parent = NULL;
   }
};
```

We will use these nodes to build *binary sort trees*. A binary sort tree is a binary tree that satisfies the binary sort tree property: For each node n in the tree, the key in node n is $\geq$ every key in the left subtree of n and is $\leq$ every key in the right subtree of n. Since we are working with maps, which allow no duplicate keys, we actually have that the keys in the left subtree of a node are strictly less than the key in that node, and the keys in the right subtree are strictly greater. In a binary sort tree, an *inorder traversal* of the tree will access the nodes in order of increasing key:

```
void inorder_traversal( Node* node ) {
```

```
        // Perform an inorder traversal of the tree whose
        // root is the given node.
     if (node != NULL) {
        inorder_traversal(node->left);  // Traverse left subtree.
        process(key,value);  // Process this node in some way.
        inorder_traversal(node->right); // Traverse right subtree.
     }
  }
```

To traverse an entire tree, this recursive procedure should be called with a pointer to the root of the tree as its parameter. However, recursive tree traversals are not exactly what we need to implement SortedMaps, so let's turn to the design of an implementation class for the SortedMap ADT. This class will use the definition of *Node*, as given above, but note that *Node* is not part of the public interface of the SortedMap class. The class will include a private instance variable named *root* that is a pointer to the root of the binary tree. There is also an integer instance variable named *count* that is always equal to the number of nodes in the tree.

We need to implement all the operations of the SortedMap ADT. Let's start with the *containsKey*(k) operation, which is very straightforward:

```
bool containsKey(KeyType k) {
   Node *runner = root;
   while (runner && runner->key != k) {
      if (k < runner->key)
         runner = runner->left;
      else
         runner = runner->right;
   }
   return (runner != NULL);
}
```

This function starts at the root of the tree and follows a path down the tree to the location of k in the tree. The while loop ends when k is found or when the bottom of the tree is reached. At that point, *runner* is non-null if k has been found and is null if the bottom of the tree has been reached without finding k. Inside the while loop, if *runner* is pointing to a node that does not contain k, the binary sort tree property makes it possible to decide which subtree of the node should contain k. If k is less than the key in the current node, then k must be in the left subtree, so *runner* is advanced to the left. If k is greater than the key in the node, then k must be in the right subtree, so *runner* is advanced to the right.

The implementation of *get*(k) is very similar, except that the value associated to the key k is returned, and an error occurs if k is not found in the tree. The implementation of *set*(k,v) is more interesting. If k is found in the tree, the value associated with k is simply changed to v. When k is not in the tree, a new node containing k and v must be added to the tree; this is a little tricky. Furthermore, the first node added to the tree is a special case, since the value of *root* must be modified:

```
void set(KeyType k, ValueType v) {
   if (!root) {
         // Create a root node containing k and v.
      root = new Node(k,v);
      count++;
      return;
   }
   Node *runner = root;  // Start at the root, which is non-null.
   while (true) {
      if (k == runner->key) {
            // k is already in the tree; set associated value to v.
         runner->value = v;
         return;
      }
      if (k < runner->key) {
            // k belongs in the left subtree of the current node.
         if (runner->left) {
               // Left subtree is non-empty; move into that subtree.
            runner = runner->left;
         }
         else {
               // Left subtree is empty, so k can't be in the tree;
               // Create a new node containing k and v as the
               // left child of runner.
            runner->left = new Node(k,v);
            runner->left->parent = runner;
            count++;
            return;
         }
      }
      else {
            // k belongs in the right subtree of the current node.
         if (runner->right) {
            runner = runner->right;
         }
         else {
            runner->right = new Node(k,v);
            runner->right->parent = runner;
            count++;
            return;
         }
      }
```

```
   }
}
```

The *removeKey*(k) operation is much more difficult than this. The idea is to find the node that contains k, if there is one, and remove that node from the tree. However, we must modify the tree in a way that will preserve both its binary tree structure and the binary sort tree property. Of course, if the node has no children, we can simply remove it from the tree. Although it is not quite as obvious, if the node has exactly one child, we can remove the node from the tree and replace it by its child. Here is a function that performs this operation; this would be a private member of our class:

```
void spliceOut(Node *node) {
      // Precondition:  node has zero or one child.
      // Remove the node from the tree.  If it has a
      // child, substitute that child for node.
   if (node == root) {
         // We are splicing out the root node.  This is a special
         // case because the instance variable root has to change.
      if (node->left)
         root = node->left;  // Replace root with left child.
      else
         root = node->right; // Replace with right child (or NULL).
      if (root)
         root->parent = 0;   // Update parent pointer in root.
   }
   else if (node == node->parent->left) {
         // The node is the left child of its parent.  The parent's
         // left child pointer must be set to point to node's child.
      if (node->left) {
            // Node has a left child.  Attach it to the parent node.
         node->parent->left = node->left;
         node->left->parent = node->parent;
      }
      else if (node->right) {
            // Node has a right child.  Attach it to the parent node.
         node->parent->left = node->right;
         node->right->parent = node->parent;
      }
      else {
            // Node has no children; parent's left child becomes NULL.
         node->parent->left = 0;
      }
   }
```

```
   else { // Must have node == node->parent->right
         // The node is the right child of its parent.  The parent's
         // right child pointer must be set to point to node's child.
      if (node->left) {
         node->parent->right = node->left;
         node->left->parent = node->parent;
      }
      else if (node->right) {
         node->parent->right = node->right;
         node->right->parent = node->parent;
      }
      else
         node->parent->right = 0;
   }
}
```

We can use this function to delete a node that has no children or only one child. If a node has two children, we can proceed as follows: Find the node that contains the next largest key in the tree. This is called the *successor*. Remove the successor node from the tree, and replace the node with its successor node. This does not disturb the binary sort tree property. The successor of a node with a right child is the left-most node in the right subtree of the node. It can be found by following the right pointer from the node and then following left pointers until the bottom of the tree is reached. Note that this successor node does not have a left child, and so we can remove it from the tree by applying the *spliceOut* operation to it.

The operation of finding a successor node is useful in other contexts, and we also need to be able to find the successor of a node that does *not* have a right child. In this case, the successor of the node (if any) is one of the ancestors of the node. In fact, if we climb up the tree by following parent pointers, the successor is the first node we come to in which the key is bigger than the key in the node. Here is a function that implements this idea:

```
Node *successor(Node *node) {
      // Find the successor node of a non-null node.
      // If node is the maximal node, the return value is NULL.
   if (node->right) {
         // Node has a right child; the successor is the
         // leftmost node in the right subtree.
      node = node->right;  // Move to the right subtree.
      while (node->left)   // Find the leftmost node.
         node = node->left;
   }
   else {
        // There is no right subtree, so the successor, if one
        // exists, the first ancestor with a larger key.
```

```
      KeyType k = node->key;  // The key in this node;
      do {  // Climb the tree until successor is found
            // or until node becomes NULL.
         node = node->parent;
      } while (node && node->key < k);
   }
   return node;
}
```

We can now write a function for removing a node from the tree. The only hard part is adjusting all the pointers in the tree when we replace a node with its successor. The function is written to return a pointer to the successor of the node, since that pointer will be needed when we implement the Enumerator's *erase* operation:

```
Node* removeNode(Node *node) {

   Node *next = successor(node);  // Find the successor of this node.

   if (!node->left || !node->right) {
         // The node has at most one child; just splice it out.
      spliceOut(node);
   }
   else {

         // The node has two children.  Replace it with its successor.

      spliceOut(next); // Splice out the successor node.

          // Now, we have to replace the node with its successor
          // node.  This means changing all the pointers in next
          // and changing any pointers that currently point to
          // node so that they point to next instead.

      next->left = node->left;      // Copy pointers from node to next.
      next->right = node->right;
      next->parent = node->parent;

      if (next->left)
         next->left->parent = next;  // Update pointer in left child.

      if (next->right)
         next->right->parent = next; // Update pointer in right child.
```

```
            if (next->parent) {
                  // The node had a parent, so we must update the
                  // child pointer in that parent so that it points
                  // to next.  We have to check whether node was the
                  // left or right child of the parent.
               if (next->parent->left == node)
                  next->parent->left = next;
               else  // Must have next->parent->right == node
                  next->parent->right = next;
            }
            else {
                 // The node that we deleted was the root node, so
                 // next becomes the root of the tree.
               root = next;
            }
         }
         delete node;  // Recover the memory occupied by the deleted node.
         count--;      // Update the number of nodes in the tree.
         return next;  // The return value is the successor node.
      }
```

And finally, we can look at the *Enumerator* class. An Enumerator is always associated with a particular map. It is created specifically to access items in that map. As I've implemented it here, an Enumerator contains a pointer to a node as well as a pointer to the map to which it is associated. The constructor for the enumerator class is private. However, the TreeMap class, which is my SortedMap implementation, is declared to be a friend of the Enumerator class. This means that Enumerator objects can be created within the TreeMap class, but nowhere outside that class. An Enumerator is null if its Node pointer is null. An exception is thrown when any attempt is made to access an item through a null Enumerator.

```
      class Enumerator {
           Node* node;
           TreeMap* map;
           Enumerator(Node *n, TreeMap *t) { node = n; map = t; }
           friend class TreeMap;
         public:
           bool more() {   // Tests whether this Enumerator is non-null.
              return (node != 0);
           }
           KeyType key() {  // Return the key of the current item.
              if (!node)
                 throw NO_MORE_ITEMS;
              else
```

```
            return node->key;
        }
        ValueType value() {  // Return the value of the current item.
            if (!node)
                throw NO_MORE_ITEMS;
            else
                return node->value;
            }
        void next() {  // Advance current item to next item in the tree.
            if (!node)
                throw NO_MORE_ITEMS;
            else
                node = map->successor(node);
        }
        void erase() {  // Delete the current item.
            if (node)
                node = map->removeNode(node);
        }
    };
```

This class is actually a public nested class inside the TreeMap class. The SortedMap *begin*() and *seek*(k) operations return objects of type Enumerator. Their definitions are left as an exercise.

6.2 Balanced Binary Trees

Recall that the height of a binary tree is the length of the longest path from the root to any leaf node in the tree. If we look at any of the binary tree operations discussed in the previous section, we see that the algorithm starts at the root and traverses a path down the tree, doing a constant amount of work at each step along the path. (For the delete operation, it is sometimes necessary to follow this path back up the tree to find the successor of a node, but again, this only adds a constant amount of work at each node along the path.) After traversing the path, the algorithm might do another constant amount of work, such as inserting or deleting a node. It is easy to see, then, that the worst-case run time is $\Theta(h)$, where h is the height of the tree. However, we would like to know the run time in terms of n, the total number of nodes in the tree.

What is the height of a binary tree with n nodes? In fact, it can be as large as n. Suppose, for example, a sequence of n strings are inserted in alphabetical order into an initially empty binary sort tree. Each string that is inserted is greater than all the strings already in the tree, so as it moves down the tree it always moves to the right. The result is a "tree" that is really a linked list of nodes descending to the right, connected by right-child pointers. The height of this tree is n. Binary tree operations on such a tree have worst-case run time $\Theta(n)$, the same as for a linked list.

However, if items are inserted into the tree in a random order, it seems that the *average* path length for all the nodes in the tree will be much less than n. It can be shown that when items are

inserted into a binary sort tree in random order, the height of the resulting tree will be, on average, $\Theta(\log(n))$. This does not mean that a tree with much larger height is an impossible outcome, merely that such trees are very unlikely.

The "average" tree created by inserting random items into a binary sort tree is *balanced*. A binary tree is balanced if each node has approximately the same number of descendents in its left subtree as in its right subtree. Suppose that you start at the root of such a tree and follow a path down the tree to a leaf node. Consider the number of nodes in the subtree rooted at your current position on the path. At the root of the tree, this number is n, the total number of nodes in the tree. At a leaf node, the number is zero. Since the tree is balanced, moving one step down the tree will cut the number of nodes in the subtree approximately in half (since the left and right subtrees at any position contain about the same number of nodes.) It follows that the number of steps along the path is approximately $\log(n)$. From this, if follows that the both the worst-case and average-case run times of binary sort tree operations on a balanced tree are $\Theta(\log(n))$.

It would nice if we could **ensure** that the trees we use are balanced, whether or not the items are inserted in a random order. This would ensure a worst-case run time of $\Theta(\log(n))$ for the SortedSet and SortedMap operations.

One approach that has been developed is to do insertions and deletions much as we have described, but, when an operation causes the tree to become unbalanced, to adjust the structure of the tree to restore its balance. This will involving moving some nodes within the tree. This must be done in a way that preserves the binary sort tree property. And it must be efficient, with run time $\mathcal{O}(\log(n))$, or it will not be worth doing. These requirements are not easy to meet. A **red-black tree**[4] is a well-known data structure that does meet the requirements. A red-black tree is a binary sort tree that is kept approximately in balance by adjusting the structure of the tree after each insertion or deletion to preserve an approximate balance. The height of a red-black tree with n nodes is guaranteed to be less than or equal to $2 * \log_2(n)$. Red-black trees are used to implement Java's Set and Map classes, and they are a likely candidate for the set and map classes in the C++ standard template library. However, we will not study red-black trees, since the details are not very illuminating.

Another approach is to abandon binary trees and to use nodes that can have a larger, variable number of children. In these trees, all leaf nodes are on the bottom level of the tree, so that the tree is always balanced. These trees are known as *BTrees*, and we will study them in the next chapter.

6.3 Multisets and Multimaps

A set is defined to be a collection of items with no duplicates. If we allow duplicate items, we get the Abstract Data Type known as a *Multiset* or *Bag*. For a multiset, instead of asking whether an item is contained in the data structure, we can ask how many copies of the item are contained. Adding an item to a multiset increments the count for that item. Removing an item decrements the count, unless the count is already zero. In fact, a multiset as a way of having a counter associated

[4] *Introduction to Algorithms*, by Cormen, Leiserson, and Rivest, Chapter 14

to each item in a set. A multiset can easily be implemented as a hash table or as a binary sort tree simply by adding a counter to each node in the data structure used to represent sets.

We can extend maps in a similar way. A map does not allow duplicate keys. That is, any given key has at most one associated value. In a *Multimap*, duplicate keys are allowed so that a given key can have several associated values. This is actually very common. For example, if a value consists of a name-and-address record and the key is the name, there can be many values with the same key. So, if we want to support look up of address by name, we need a multimap rather than a map.

Multimaps are not difficult to implement. In fact, the binary sort tree implementation of maps works with only minor modification for multimaps. Only the code for finding the successor of a node has to be modified (because the successor of a node can contain the same key as that node).