CHAPTER 13

Binary Search Trees

Previously, you learned about recursion, general trees, and binary trees. This chapter deals with a variant of the binary tree called a *Binary Search Tree (BST)*. The BST is a structure where recursion is more important in determining how the data is stored rather than how the data is accessed. You'll see what I mean by this later in the chapter.

In this chapter, you will learn

- What a BST is
- How to insert data into a BST
- How to find data in a BST
- How to code a BST class
- How to use a BST to search for resources in a game

What Is a BST?

Imagine that you have to sort a group of people by height so that you can easily search for someone by their height later on. How would you go about doing this?

Figure 13.1 shows six people that you need to sort.

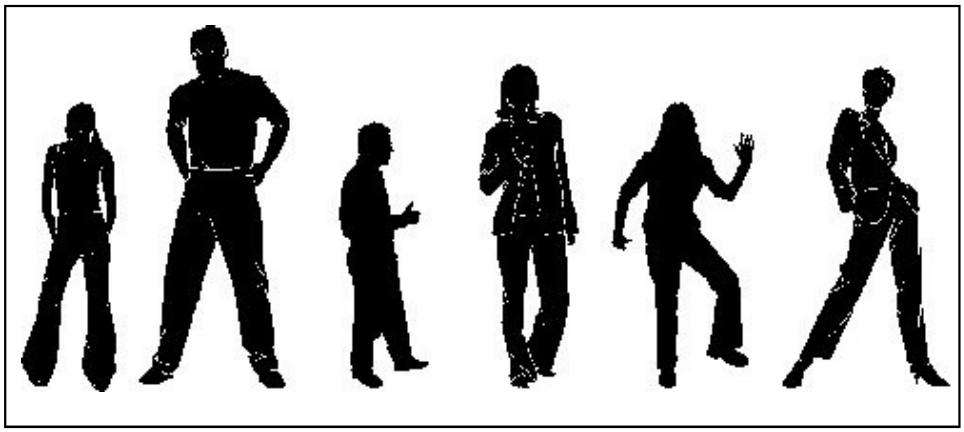


Figure 13.1
Don't hate them because they're beautiful.

The easiest way to sort them is to find the shortest person and put him/her first, and then find and place the next shortest, and so on. This method of sorting on a computer is slow, though. You can stand back and immediately see the shorter

people in the line of people waiting to be sorted; the computer can't do that. The computer would need to look at every person in line to find out who is the shortest.

Instead, why don't you do something clever? Pick a midpoint (say, 5 feet, 6 inches) and look at the first person in line. If he/she is below that height, you move him/her to the left. If he/she is above that height, you move him/her to the right. Now, whenever you want to search for someone of a particular height, all you need to do is determine which half of the line that height would be in and search only that half of the line!

For example, if you wanted to find someone with a height of 6 feet, you would look in the right half of the line because no one who is 6 feet tall would be in the left half.

Figure 13.2 shows the group of people *partitioned* in half.

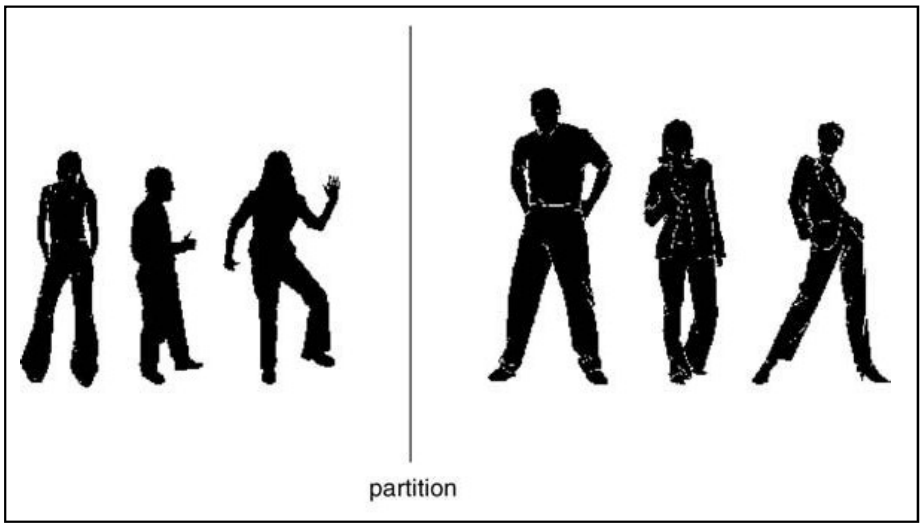


Figure 13.2

The perfume models are now partitioned into two groups, the tallest on one side, and the shortest on the other.

This sorting method is employed by the Binary Search Tree data structure. It attempts to split data in half to make searching easier.

Inserting Data into a BST

Say you have a queue of data that you want to search through. You take the first item off the queue and put it as the root of the tree. Then, you take the next item off the queue and compare it with the root. If it is less than the root, then you make it the left child of the root. If it is more than the root, then you make it the right child of the root.

Now, repeat the process. Take another item off the queue and do the same thing. If a node already exists on the left or the right children, then you go down another level and compare the items again.

For example, say you have a queue containing this data: 4, 2, 6, 5, 1, 3, 7. The first step is to take off the 4 and insert it as the root node in a BST. Then you take off the 2 and compare it with the 4. Because 2 is less than 4, you insert 2 as the left child of the root. Then you take off 6, which is placed as the right child of the root because it is more than 4. Figure 13.3 shows the first three steps.

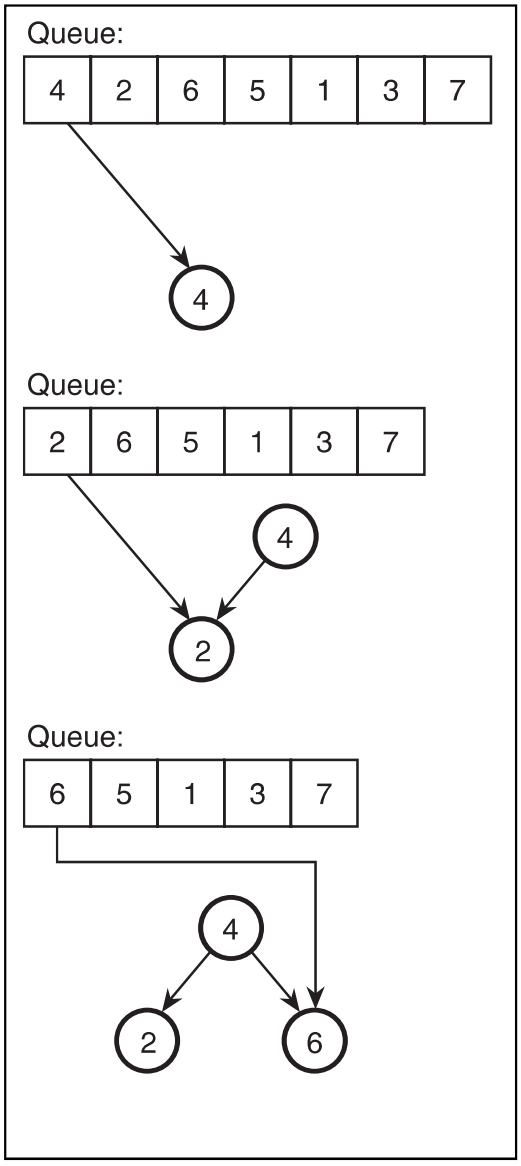


Figure 13.3

This is how you insert the first three nodes into the BST.

After you have completed that step, you want to insert 5 into the tree. First, you compare it with 4 at the root, and because it is larger than 4, you try to insert it to the right. However, there is already a node to the right! So you compare the 5 with the 6 in the right node; because 5 is less than 6, you insert the 5 as the left child of

the 6. Likewise, the 1 is compared to the 4 and then the 2 and then inserted as the left child of the 2. Figure 13.4 shows these two steps.

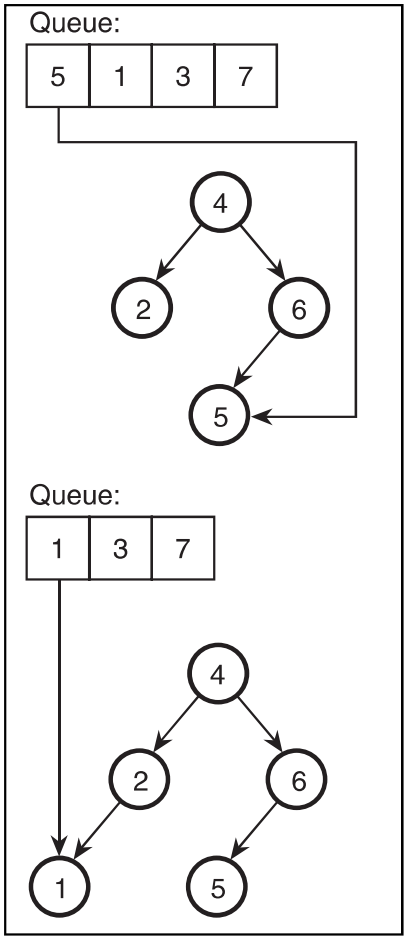


Figure 13.4

This is how you insert the next two nodes.

See if you can figure out where the 3 and the 7 go. Figure 13.5 shows where they are inserted if you're stumped.

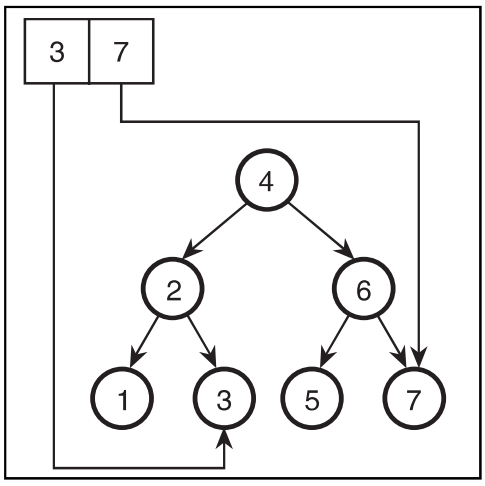


Figure 13.5

Finally, this is how you insert the last two nodes.

So, now that you have the final BST in Figure 13.5, see if you can figure out why I've partitioned the data like this.

Finding Data in a BST

Now that the data has been inserted into the tree, how do you search for the data quickly? By using the same algorithm, of course! If you want to search for 3, you compare it with 4, go left, compare it with 2, and go right, and you've found it! That was nice and easy, wasn't it? In fact, the most comparisons you can make when searching for something within this tree is 3, and there are 7 items within the tree. If the tree was one level larger, it could hold 15 items, but the most comparisons you could make would be 4!

In Chapter 1, "Basic Algorithm Analysis," I introduced you to the logarithm function. The base-2 logarithm of 8 is 3 (because $2^3 = 8$ and the logarithm is the inverse of the power function), and the base-2 logarithm of 16 is 4 ($2^4 = 16$). You can see that the BST search algorithm is roughly $O(\log_2 n)$. However, this is the best-case scenario; you will see why in a bit.

Removing Data from a BST

There is a BST node removal algorithm, but I don't cover it here. The algorithm is long and messy, and because I consider BSTs to be of only marginal importance to general game programming, I refer you to an article I've included on the CD in the \goodies\articles\ directory entitled *Trees Part II: Binary Trees*. It has the complete algorithm for removing nodes from a BST.

The BST Rules

You must always follow two rules for every node in a BST:

1. Every node in the left subtree must be less than the current node.
2. Every node in the right subtree must be greater than the current node.

You can see that this is a recursive definition; it applies to *every* node in the tree. You can also see that these rules effectively (in an optimal tree) split the amount of data you need to search through by half for every level you search in the tree.

Sub-Optimal Trees

I admit it: The first BST example I gave you was doctored. I fixed the data so that the tree ends up being full. However, data is usually not organized like that, and it usually produces BSTs that are not optimal.

First, let me show you the absolute worst case for inserting data into a BST. Say you have a queue of this data: 1, 2, 3, 4, 5. Inserting this data into a BST creates the tree shown in Figure 13.6.

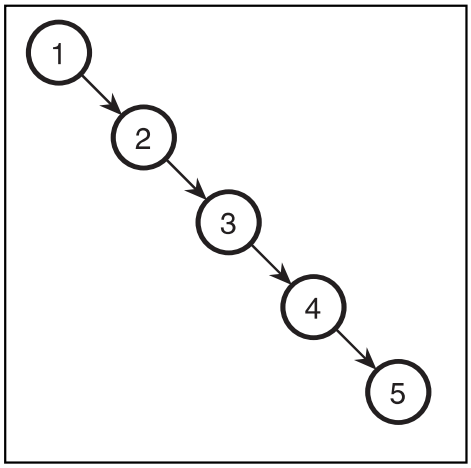


Figure 13.6

This is a worst-case BST; it looks just like a linked list.

The 1 is inserted as the root, the 2 as the right child of 1, the 3 as the right child of 2, and so on. What does this resulting tree look like? A linked list, of course. There is no branching done at all in this tree, and if you want to search for data within it, you're stuck doing a linear search, $O(n)$, which is considerably slower than $O(\log_2 n)$. This is rather unfortunate, and there are ways around this, but they are beyond the scope of the book. *AVL trees, splay trees,* and *red-black trees* are all special forms of BSTs that perform *rotations* on the nodes when they are inserted so that the tree ends up more balanced.

As long as the data you are inserting is somewhat random, you will end up with decent trees. However, if data is sorted already or has some statistical correlation, you might end up with less than optimal trees.

Graphical Demonstration: BSTs

This is Graphical Demonstration 13-1, which you can find on the CD in the directory \demonstrations\ch13\Demo01 - BSTs\.

Compiling the Demo

This demonstration uses the SDLGUI library that I have developed for the book. For more information about this library, see Appendix B, "The Memory Layout of a Computer Program."

To compile this demo, either open up the workspace file in the directory or create your own project using the settings described in Appendix B. If you create your own project, all of the files you need to include are in the directory.

This demonstration is fairly simple because the BST structure is fairly simple to use. Figure 13.7 shows a screenshot from the demo in action.

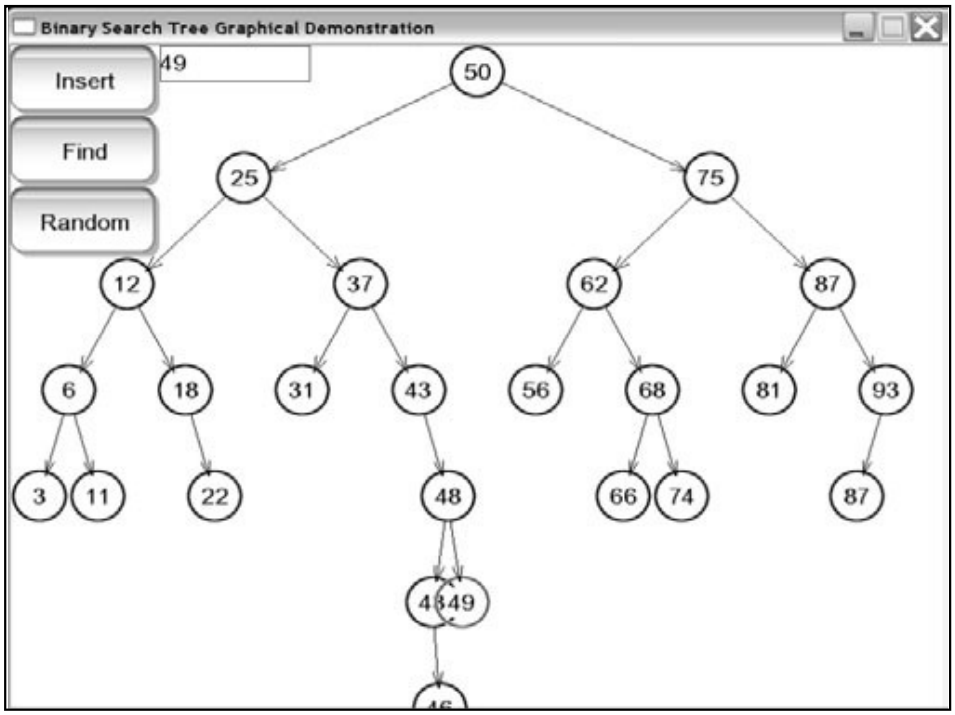


Figure 13.7

Here is a screenshot from the BST demo.

As you can see from the screenshot, the demo has three buttons and a text box. You can type any number from 0–99 in the text box, or you can click the Random button to insert a random number into the text box.

After you have a number in the text box, you can do two things with it: You can either insert that number into the BST or search for that number in the BST.

Clicking either button makes the demo follow a path down the tree, either trying to insert a node or just finding a node.

Play around with it and get to know how BSTs work a little better.

Coding a BST

The code for the Binary Search Tree is located on the CD in the file \structures\BinarySearchTree.h.

The Structure

The binary search tree uses a binary tree as its underlying structure, but the actual class is just a container; it has a pointer to the root node and a comparison function.

```
template <class DataType>
class BinarySearchTree
{
public:
    typedef BinaryTree<DataType> Node;
    Node* m_root;
    int (*m_compare)(DataType, DataType);
};
```

Comparison Functions

You've seen function pointers a few times already in this book; the hash functions for hash tables (see Chapter 8, "Hash Tables") and the process functions for the tree traversals (see Chapters 11, "Trees," and 12, "Binary Trees") come to mind. This time, I introduce you to the idea of comparison functions.

The idea here is that you are probably going to be storing complex structures in the BST, right? So how, exactly, does one determine if one class is "larger" or "smaller" than another? Sure, it's easy with integers, but what about other classes, say, a complex game player class?

Using a custom comparison function allows you to customize how data is stored in the BST. For example, you may want to store characters in a BST based on how much life they have left and search based on that. Then, sometime down the road, you might want to make a different BST that stores characters, but this time you

want to search based on another attribute—perhaps how strong the character is. By using a comparison function, this change is easy; you can make a new function that compares the strength of two characters instead of the health.

The definition of the comparison function is simple: It takes two parameters of type `DataType` and returns an integer. The integer return value can have three meanings. If the number is negative, then the left parameter is less than the right. If the number is 0, then the two parameters are equal. If the number is positive, then the left parameter is more than the right.

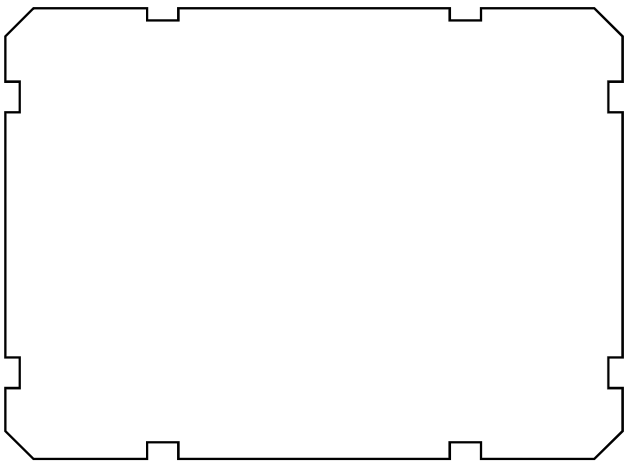
For example, you can create a simple comparison function for integers, like this:

```
int CompareInts( int left, int right )
{
    return left - right;
}
```

If the `left` is less than the `right`, then the result is negative. If they are equal, then the result is 0. If `left` is larger than `right`, then the result is positive.

The Constructor

The constructor function basically takes a comparison function as a parameter and sets the root to null.



```
BinarySearchTree( int
(*p_compare)(DataType, DataType) )
{
    m_root = 0;
    m_compare = p_compare;
};
```

The Destructor

The destructor should simply delete the root node. Remember from Chapter 12 that the `BinaryTree` destructor recursively destroys every node in the tree. That makes this function really simple:

```
~BinarySearchTree()
{
    if( m_root != 0 )
        delete m_root;
}
```

The Insert Function

Now comes the `Insert` function. There are two ways you can insert the node into the binary tree; one is recursive, and the other is iterative. The recursive function in this case is pointless because this isn't really a recursive algorithm. So instead of recursion, I use the iterative algorithm. I split this up into a few segments so that it is easier to understand.

```
void Insert( DataType p_data )
{
    Node* current = m_root;
    if( m_root == 0 )
        m_root = new Node( p_data );
```

This first segment takes a piece of data as a parameter and creates an iterator named `current`, which points to the root of the tree. If the root is empty, the function creates a new root node.

If not, the function continues:

```
    else
    {
        while( current != 0 )
        {
```

This segment starts the `while` loop. The function travels down the tree while the iterator is valid, and as soon as the function inserts a node into the tree, it sets the iterator to 0 so that the loop will exit.

```
            if( m_compare( p_data, current->m_data ) < 0 )
            {
                if( current->m_left == 0 )
                {
                    current->m_left = new Node( p_data );
                    current->m_left->m_parent = current;
                    current = 0;
                }
                else
                    current = current->m_left;
            }
```

The previous segment of code does a few things. It first compares the data in the current node with the data that you want to insert into the tree. If the result of the `m_compare` function is less than 0, you want to insert it into the left child. The next

step is to check if the left child exists. If not, create a new left child and set `current` to 0. If it does, then move the current pointer to the left.

This next code segment does the same thing, but to the right this time:

```
            else
            {
                if( current->m_right == 0 )
                {
                    current->m_right = new Node( p_data );
                    current->m_right->m_parent = current;
                    current = 0;
                }
                else
                    current = current->m_right;
            }
        }
    }
}
```

And that's the function.

> **CAUTION**
>
> **This function does *not* check for duplicated data. Typically, BSTs do not allow for duplicated data to be entered into the tree, but sometimes they do. Because this BST class doesn't support node removal, you're just wasting space if you insert duplicated data into the tree—the `Find` function will never find it.**

The Find Function

This function is almost the same as the `Insert` function except that it just returns a pointer to the node if it finds the data in the tree.

```
Node* Find( DataType p_data )
{
    Node* current = m_root;
    int temp;
    while( current != 0 )
    {
        temp = m_compare( p_data, current->m_data );
        if( temp == 0 )
            return current;
        if( temp < 0 )
            current = current->m_left;
        else
```

```
            current = current->m_right;
    }
    return 0;
}
```

If the data isn't found in the tree, this function returns 0.

Example 13-1: Using the BST Class

This is Example 13-1, which demonstrates how to use the `BinarySearchTree` class with integers. The source code for this example is on the CD in the directory \examples\ch13\01 - Binary Search Trees\.

The example uses the `CompareInts` function I showed you earlier to store integers in a BST:

```
void main()
{
    BinarySearchTree<int> tree( CompareInts );
    BinaryTree<int>* node;
    // insert data
    tree.Insert( 8 );
    tree.Insert( 4 );
    tree.Insert( 12 );
    tree.Insert( 2 );
    tree.Insert( 6 );
    tree.Insert( 10 );
    tree.Insert( 14 );
    // these searches are successful
    node = tree.Find( 8 );
    node = tree.Find( 2 );
    node = tree.Find( 14 );
    node = tree.Find( 10 );
    // these searches return 0
    node = tree.Find( 1 );
    node = tree.Find( 3 );
    node = tree.Find( 5 );
    node = tree.Find( 7 );
}
```

Application: Storing Resources, Revisited

This is Game Demonstration 13-1, and you can locate it on the CD in the directory \demonstrations\ch13\Game01 - Resources Revisited\.

Compiling the Demo

This demonstration uses the SDLGUI library that I have developed for the book. For more information about this library, see Appendix B.

To compile this demo, either open up the workspace file in the directory or create your own project using the settings described in Appendix B. If you create your own project, all of the files you need to include are in the directory.

When you think about them, binary search trees are nothing more than a different version of the hash tables from Chapter 8. They are designed for storing data so that you can retrieve it again quickly by using a key.

Because of this, I want to go back to Game Demonstration 8-1 and rewrite it so that it uses Binary Search Trees instead.

The Resource Class

You may have noticed that using a BST is slightly different than using a hash table; whereas a hash table used a key/value pair to store and retrieve data, my BST class doesn't do that. Instead, it just stores the data right in the tree. This particular quirk of my implementation causes me to code the demo a little differently.

First of all, I create a `Resource` class, which will have two things, a string and an `SDL_Surface` pointer:

```
class Resource
{
public:
    char m_string[64];
    SDL_Surface* m_surface;
};
```

The Comparison Function

The next thing that I need to do is to create the comparison function. Because you want to search the tree for string matches, you'll use the standard C `strcmp` function to compare the strings.

```
int ResourceCompare( Resource p_left, Resource p_right )
{
    return strcmp( p_left.m_string, p_right.m_string );
}
```

Luckily, the `strcmp` function returns a negative number if the left string is less than the right string, 0 if they are equal, and a positive number if the left is greater than the right!

So this function compares resources based on name only, not based on the actual bitmap that the `Resource` class contains. This is important when you search for something in the tree.

Inserting Resources

Inserting resources into the tree is similar to inserting them into a hash table except that instead of inserting a string/surface pair into the tree, you create a resource structure first.

```
Resource res;
res.m_surface = SDL_LoadBMP( "sky.bmp" );
strcpy( res.m_string, "sky" );
g_tree.Insert( res );
```

The `strcpy` function copies the string into the resource's name. This step is repeated for every resource in the demo.

Finding Resources

To search for a resource, you need to set up a *dummy* resource, which doesn't contain a surface, but only a string:

```
Resource res;
strcpy( res.m_string, g_name );
```

The `g_name` variable is a string that contains the name of the resource you are searching for. The `m_surface` variable of `res` is left blank.

After that, you declare a binary tree node pointer, which will hold the node that is returned from the BST's `Find` function:

```
BinaryTree<Resource>* node = 0;
node = g_tree.Find( res );
```

Now the BST will compare the dummy resource's name with the name of the resources in the BST, and if it finds a match, it will return the node that contains the resource. When the node is returned, all you need to do is determine whether it is valid and then use it:

```
if( node != 0 )
    g_resource = node->m_data.m_surface;
else
    g_resource = 0;
```

Playing the Demo

The demo plays exactly like Game Demo 8-1. Figure 13.8 shows a screenshot of the program in action.

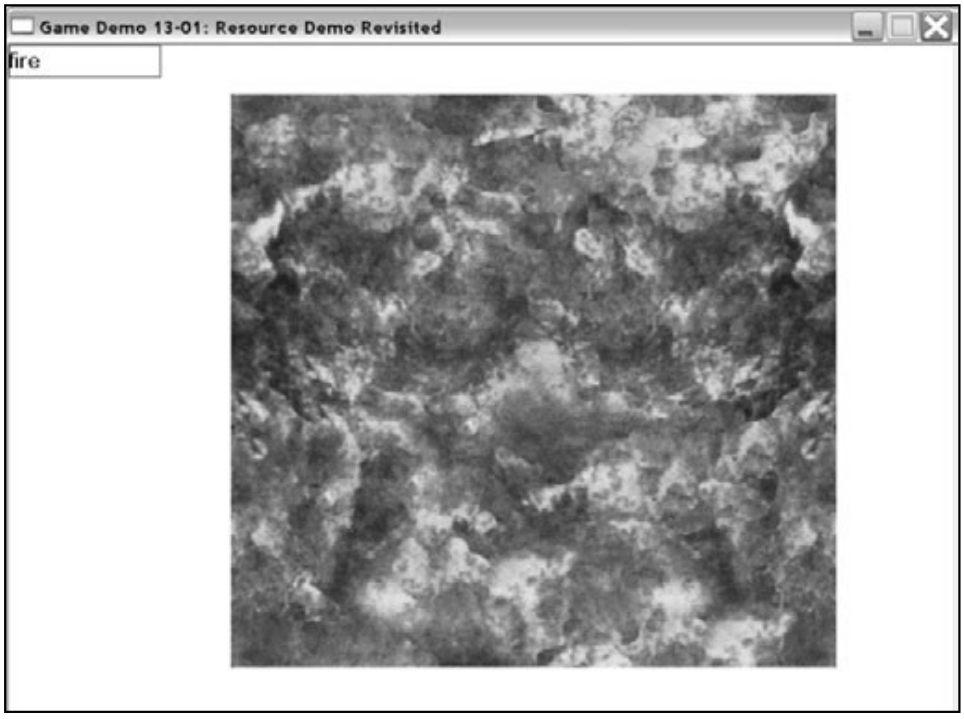


Figure 13.8

Here is a screenshot from the demo.

As before, you enter the name of the resource you want to load into the text box, and it loads the resource for you automatically. The valid resource names are sky, water, water2, snow, fire, vortex, and stone.

Conclusion

I'm going to be honest with you: Binary Search Trees don't really do much that a hash table doesn't do better. Whereas a hash table's search time runs close to $O(c)$, the best-case search time for a BST is still higher than that, at $O(\log_2 n)$. So why did I even bother to teach you BSTs?

Well, BSTs introduce you to the concept of recursively storing data. This concept becomes very important when you get into the more advanced trees used in game programming, such as *Binary Space Partition* (*BSP*) trees. BSPs are a really neat form of tree that splits polygons in a 3D (or even 2D—John Carmack used them in *DOOM*) world so that you can easily determine which polygons in a scene are visible. The concepts used in BSP trees are remarkably similar to the concepts of BSTs.

All in all, I hope you're getting a feel of how recursive tricks are used to split up large amounts of work into smaller problems.

This page intentionally left blank