By now, you should have a solid understanding of the basic data structures, especially linked lists. You should also be fairly comfortable with the concept of recursion. If you're not entirely confident of these two areas, go back and read Chapter 6, "Linked Lists," and Chapter 10, "Recursion," because much of what I show you in this chapter builds upon the concepts in those chapters.

You will probably not find any of the structures discussed from this point on in a professional container library, such as STL. The reason is that these structures are now becoming very specific; they are meant for a special purpose and cannot easily be applied to general problems.

This chapter is about *trees* in the most general sense. Sometimes they are called *general trees,* but I just refer to them as trees.

In this chapter, you will learn

- What a tree is
- How trees are recursive
- How to build a tree
- How to move around a tree
- How to build a tree class
- How to build a tree iterator class
- How to traverse a tree using a recursive function
- How to use trees to store plotline information in games

What Is a Tree?

Go outside and look at a tree. In case that is not possible, I provide you with a nice diagram of a tree in Figure 11.1.

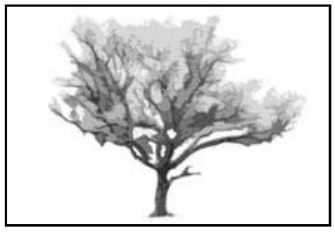


Figure 11.1

This is a tree.

The tree has several major components. The largest is the trunk, or *root*, at the bottom. *Branches* come off of the trunk, and they spread out into *twigs*, which have *leaves*. The general structure of the tree spreads out from the root.

If you think about it, a branch is really nothing but a smaller root, right? So a twig is nothing but a smaller branch as well. By looking at a tree in this manner, it is easy to see how it is considered a recursive structure. Essentially, each level is a smaller version of the level before it (except for the leaves).

Figure 11.2 shows how a tree container class looks. Instead of being drawn with the root at the bottom, though, it is drawn with the root on top. It usually makes more sense to draw them this way. Before I go any further, I'll introduce some terminology. Table 11.1 shows the common names for nodes in a tree; it refers to Figure 11.2.

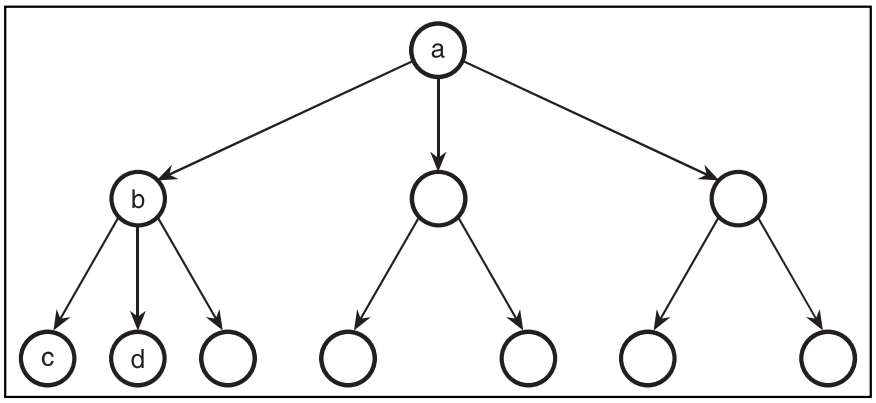


Figure 11.2

This is a tree when it's represented inside a computer.

Table 11.1 Tree Terminology

Term	Description	Example (Figure 11.2)
Root	Topmost node in a tree	Node *a* is the root.
Child	A node below another in a branch	*b* is a child of *a*.
Parent	A node above another in a branch	*b* is the parent of *c*.
Sibling	A node on the same level as another	*c* is a sibling of *d*.
Leaf	A node with no children	*c* and *d* are leaves.
Level	Describes the height of a node	*a* is at level 0, *b* is at level 1, and *c* and *d* are at level 2.
Subtree	A tree contained within another tree	*b* is the root of a subtree of *a*.

Now that you know the terminology of a tree, I can go into a little more detail. A tree, like a linked list, is a *node-based structure*. The nodes point to the next node in the structure. However, a linked list points to only one node, whereas a tree node can point to any number of children.

The Recursive Nature of Trees

Trees are considered a recursive data structure because trees are said to contain themselves. The last entry in Table 11.1 gives you a brief glimpse into this nature: Every child of a tree is a tree on its own. Figure 11.3 shows an example.

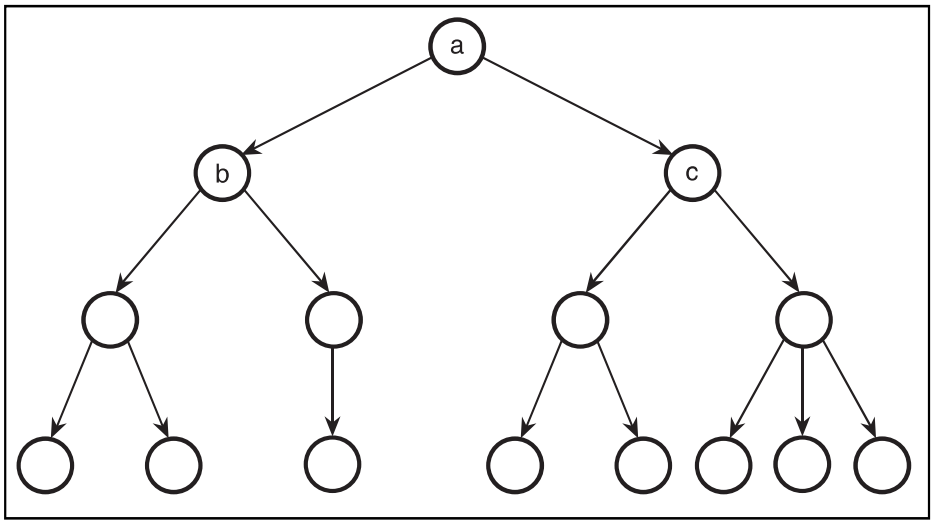


Figure 11.3

Here is a tree that demonstrates the recursiveness of trees; nodes b and c are trees themselves.

There are 3 nodes labeled in this tree: *a*, *b*, and *c*. Node *a* is the root of the entire tree. However, if you look a little lower, you can see that node *b* is the root of a smaller tree, and so is node *c*.

So you can easily say that a tree is a structure that holds trees!

Common Structure of Trees

The tree structure is very similar to a linked list, as I've said before. A tree is node-based, so each node needs a way to point to its children. In a general tree, each node can have any number of children.

So what data structure that you know of can easily be expanded to hold any number of items? That's right, linked lists!

So each node will have a linked list where each node in the list points to another tree node. Whew, what a mouthful! Figure 11.4 can better illustrate what is going on in a tree node.

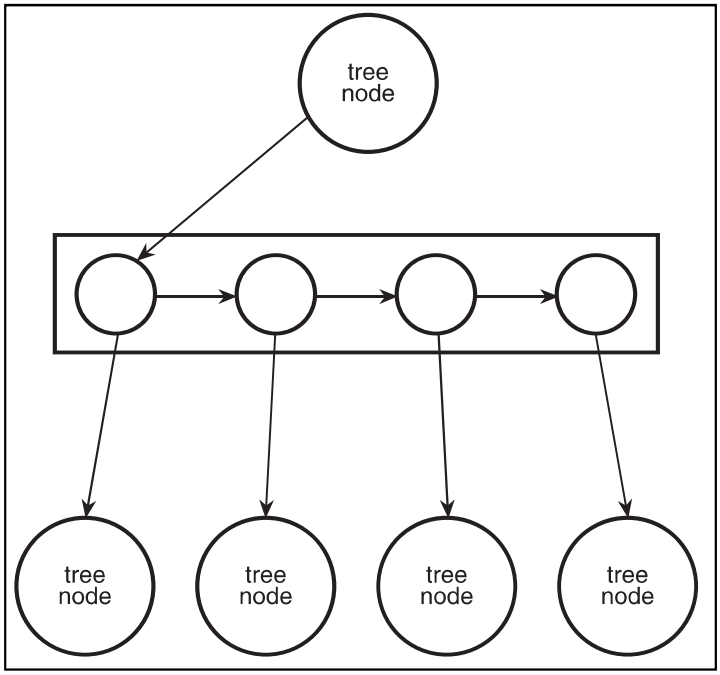


Figure 11.4

This is the internal representation of a tree node. The structure in the box is a linked list, where each node points to a tree node.

The figure shows a tree node that has four children. The top node has a linked list, which is shown inside the box. The linked list has four nodes, each of which holds a tree node pointer.

TIP

As you can see, the tree structure is built using linked list concepts, and it actually uses a linked list inside. The rest of the data structures in this book are primarily built upon the data structures that I cover in Part I. Therefore, it is very important that you understand everything in Part I before you continue.

Graphical Demonstration: Trees

This is Graphical Demonstration 11-1, which you can find on the CD in the directory \demonstrations\ch11\Demo01 - Trees\.

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 is the most complex demonstration in the book so far, so I need to do a lot of explaining before you can just jump into the demo and start playing around. Figure 11.5 shows a screenshot of the demo.

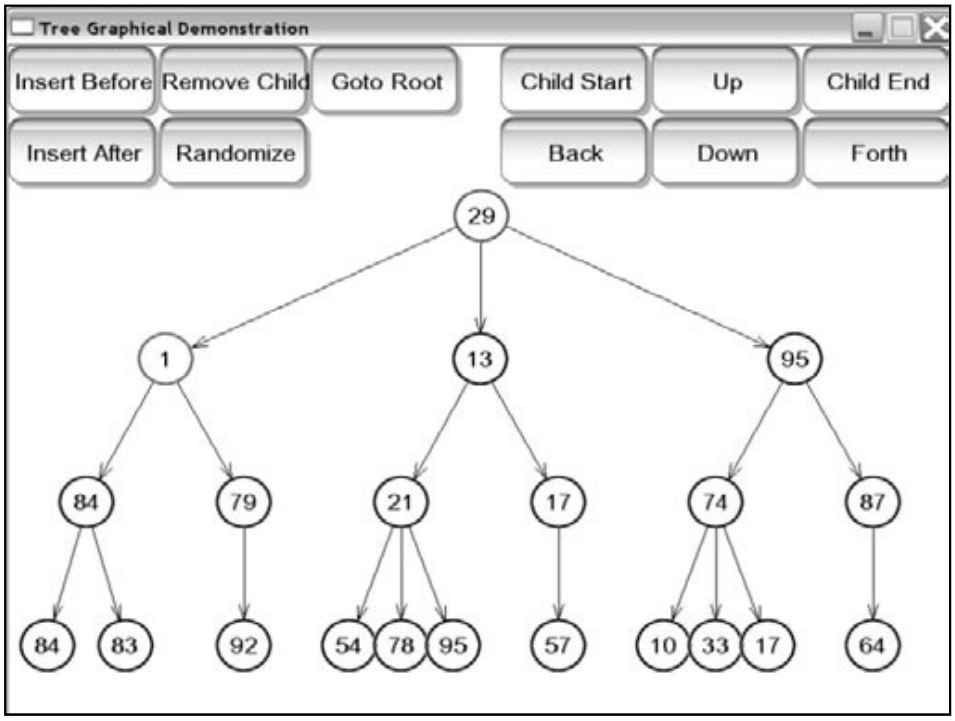


Figure 11.5

Here is a screenshot from Graphical Demonstration 11-1.

The first thing you should notice is that there are more buttons than there were in previous demos. This is because trees are the most complex structures in the book so far.

One thing that you won't see in the screenshot is that two of the nodes are colored differently. If you start up the demo, the root node will be colored red (node 29 in Figure 11.5), and the root's first child will be colored blue (node 1 in Figure 11.5).

When you iterate through a tree, you really need two iterators. If you are unfamiliar with what an iterator is, please read Chapter 6.

The first iterator, which is represented in red in the demo, keeps track of the current tree node. The second iterator, which is blue in the demo, keeps track of the current child of the current node.

Because the children in a tree are stored in a linked list, the blue iterator is just a normal `DListIterator`. Table 11.2 shows a listing of the commands in the demo and their functions.

Table 11.2 Graphical Demonstration 11-1 Commands

Command	Function
Insert Before	Inserts a new node to the left of the blue node
Insert After	Inserts a new node to the right of the blue node
Remove Child	Removes the blue node from the tree
Randomize	Creates a new random tree
Goto Root	Moves the red iterator to the root of the tree
Child Start	Moves the blue iterator to the first child of the red node
Back	Moves the blue iterator to the previous child of the red node
Up	Moves the red iterator to the parent of the current red iterator
Down	Moves the red iterator to the current child node (blue)
Child End	Moves the blue iterator to the last child of the red node
Forth	Moves the blue iterator to the next child of the red node

There are a lot of commands, and it's okay if you don't understand how they work just yet. I'm going to take you through a little tutorial with the demo.

TIP

If you want to see a cool application of recursion when dealing with trees, take a look at the source code for this demo. The `DrawSubTree` algorithm is recursive, and it is very simple, too. Essentially, the function draws a node and then calls itself to draw every child node.

Tutorial

First, run the program. You should end up with a random tree like the one in Figure 11.5. Now, click the Remove Child button. This should remove the blue node from the tree. Click that button until there is only one node left in the tree. You should only need to click it two or three times.

Now that you have an empty tree, here's how to build one.

Step 1: Build a Basic Tree

In this step, I want you to add three children to the root node. To do so, you must complete these commands:

1. Goto Root.
2. Click Insert After three times.

After you do this, you should have a tree that looks like Figure 11.6. The numbers in your nodes will be different, but just pay attention to the structure for now.

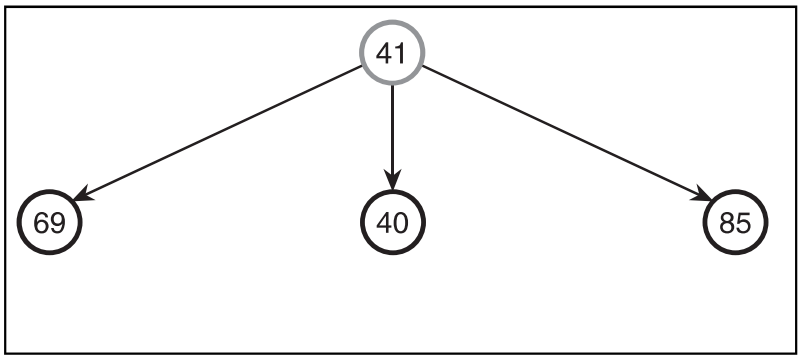


Figure 11.6

Here is the tree after Step 1.

Step 2: Traverse the Tree

Now that you have built a basic tree, I want you to traverse the tree by using the buttons on the right side of the screen.

1. Click Child Start.
2. Click Forth.
3. Click Back.
4. Click Child End.
5. Click Back.
6. Click Down.

Your red node should now be the middle node on the second level of the tree. During Steps 1–5, you should have seen the blue node moving back and forth, as if you were traversing a doubly linked list.

Step 3: Build a More Complex Tree

Now I want you to build a more complex tree. Your red node should still be on the middle node in the second level.

1. Click Insert After twice.
2. Click Child Start.
3. Click Down.
4. Click Insert After twice.
5. Click Up.
6. Click Forth.
7. Repeat Steps 3, 4, and 5.

After Step 7, you should have a tree that looks similar to Figure 11.7.

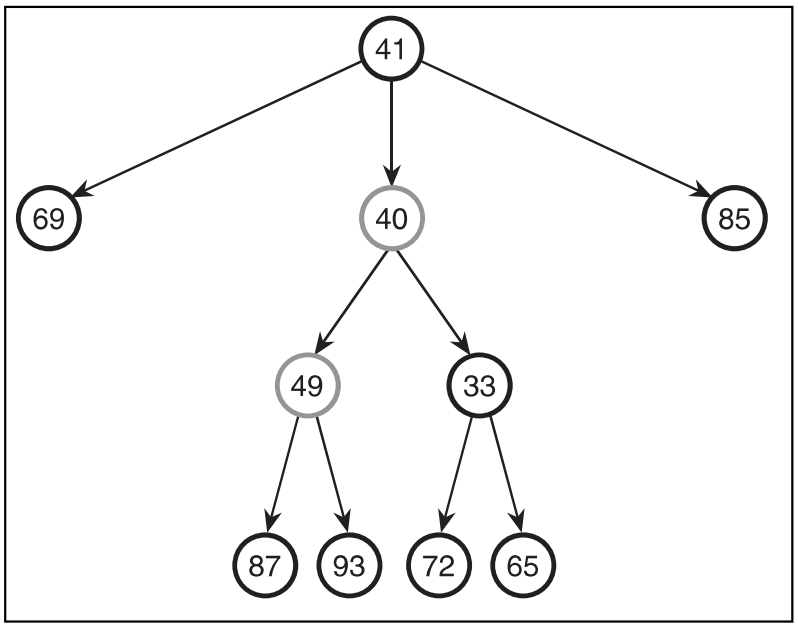


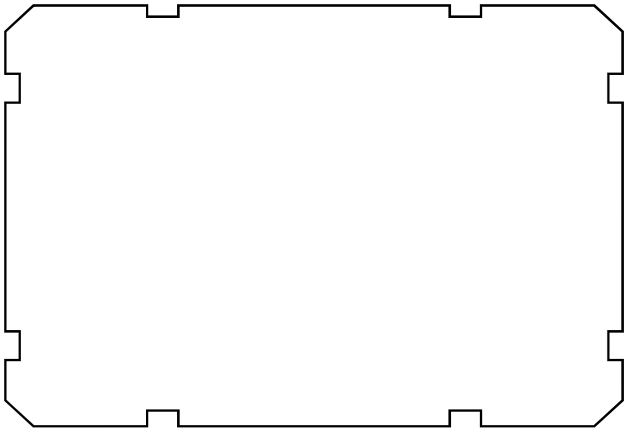
Figure 11.7

This is the tree after Step 3.

Step 4: Play Around

Now that you've created a neat-looking tree, I want you to play around with the commands—see what you can come up with. The program doesn't have any limits on the number of nodes you can add, but adding too many might make the program run slowly.

I want you to get acquainted with the manner in which you build and manipulate trees.



Building the Tree Class

The file containing all the tree classes can be found on the CD in the \structures\ directory. It's named Tree.h.

Now you should have some idea of how trees are structured. Although the actual structures themselves are not very complicated, working with the trees can get difficult.

I'll be perfectly honest with you: The classes that you are about to see are on their third major revision. When I first started working on the source code, I wanted to

build the tree so that it was nicely contained within a single class and easy to work with, like all of the classes I've used previously in the book. This method ended up being more work than it was worth, and it was more complicated to use!

So I decided to use an ultra-simplistic approach and create just the node class with very few functions. This approach didn't work, either. When I was writing Graphical Demonstration 11-1, I realized that I would end up re-writing *all* of the iterator functions if I ever wanted to use the tree class in another program.

My third and final revision of the class uses a mixture of these two approaches, as you'll see in this section. I ended up creating an iterator class so that you don't have to constantly rewrite the most-used functions.

The Structure

I already explained how general trees are structured, so I'll just post the code to show you how it looks.

```
1: template<class DataType>
2: class Tree
3: {
4: public:
5:      typedef Tree<DataType> Node;
6:      DataType m_data;
7:      Node* m_parent;
8:      DLinkedList<Node*> m_children;
9: };
```

As usual, I'm using a templated class so that you can store any type of data you want into the tree.

The first thing to note is on line 5. On that line, I used a `typedef` so that using the tree class is easier to do. Now, instead of saying `Tree<DataType>` whenever you want to use a node, you can just type `Node` instead. `Typedefs` make life so much easier.

On line 6, I define the holder for the data, just like the linked list classes.

On line 7, I put a pointer to the parent node. I like having a parent pointer in my trees because it allows me to easily backtrack, but having a pointer to the parent node is not necessary.

On line 8, I define the linked list of nodes that will store the pointers to the node's children.

All in all, this is not a difficult structure to visualize on a node-per-node basis.

The Constructor

Here is the code for the constructor:

```
Tree()
{
    m_parent = 0;
}
```

I want you to pay attention to the fact that the parent is cleared to 0. Whenever I deal with tree nodes, the node is considered a root node if the parent pointer is 0.

Also, note the other cool thing: Because the `DLinkedList` class already has a constructor, the `m_children` list is automatically initialized and holds 0 nodes.

The Destructor

Whenever a tree node is deleted, you need to make sure that it is properly cleared from memory. However, this process is not like deleting a linked list node. For example, take a look at Figure 11.8.

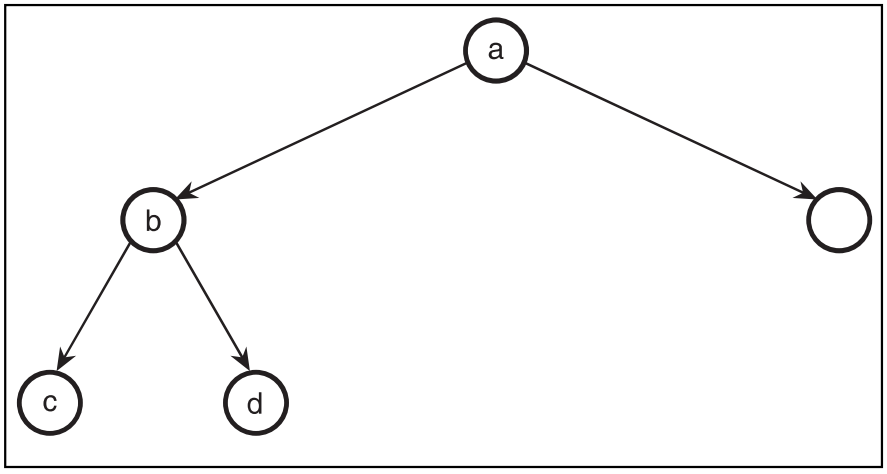


Figure 11.8

This is a tree that is used in the destruction example.

You determine that you no longer want node *b* in the tree in Figure 11.8. How do you go about removing the node from the tree?

Perhaps the easiest thing to do would be to take nodes *c* and *d* and add them as children to node *a*. Or maybe you want to make some other arrangement, and you can move *c* and *d* around to different places in the tree.

But how do you determine where those nodes go? You really can't do that with a general-purpose tree. The previous method is usually only used in specific tree

types, such as the *binary search tree* (see Chapter 13, "Binary Search Trees") and *heaps* (see Chapter 14, "Priority Queues and Heaps").

However, most of the time, you will find that the children of any given node are *directly* related to its parent. If you remove the parent node, you should also remove the children nodes as well as all of their children, and so on. So if you want to remove node *b*, you need to remove node *c* and *d* as well. If *c* or *d* had any children, you should remove them, continuing this way down to the bottom of the tree. To do this, you call the trees' `Destroy` function:

```
// destructor
~Tree()
{
    Destroy();
}
```

The Destroy Function

The `Destroy` function is called whenever a node is destructed or whenever you want to delete all of a tree's children. Note that it uses recursion; if you aren't familiar with recursion yet, please go back and read Chapter 10. The function is very simple if you think about it recursively:

```
1: void Destroy()
2: {
3:      DListIterator<Node*> itr = m_children.GetIterator();
4:      Node* node = 0;
5:      itr.Start();
6:      while( itr.Valid() )
7:      {
8:           node = itr.Item();
9:           m_children.Remove( itr );
10:           delete node;
11:      }
12: }
```

The function starts off by creating an iterator to the list of children. Then, for each child in the list, the function removes the pointer from the child list (line 9) and then deletes the node (line 10).

So how is this function recursive? Well, the destructor of each child node is called whenever a node is deleted, and the destructor calls the `Destroy` function. So, in effect, the function is recursive.

Look back to Figure 11.8. If you were to delete node *b* from that tree, this function would first loop through and remove *c* and *d* from its child list and then delete *c* and *d*. But the act of deleting *c* and *d* calls their destructors, which in turn calls `Destroy` again! If those nodes had any children, they would be deleted, too! This function is one large chain reaction that deletes every single node in a subtree. Isn't recursion neat?

The Count Function

The `Tree` class has one more function: `Count`. This function counts the number of nodes in a subtree and returns the result.

```
int Count()
{
    int c = 1;
    DListIterator<Node*> itr = m_children.GetIterator();
    for( itr.Start(); itr.Valid(); itr.Forth() )
        c += itr.Item()->Count();
    return c;
}
```

Note that this function is also recursive. (Are you noticing a trend?) The function creates an integer variable, `c`, and sets it to 1. That 1 represents the current node.

Then, an iterator to the child list is retrieved, and the function loops through every child and adds the count of the child to `c`. Finally, `c` is returned.

In effect, this algorithm says: The count of any subtree is equal to 1 plus the count of each child.

The Tree Iterator

I mentioned before that this class has undergone several variations. The first method I tried used an iterator, and the second didn't use any at all. This is the third version, in which I made the iterator into one class that is easily managed. This way, you won't have to rewrite every iterator function on your own when working with trees.

The Structure

The TreeIterator structure is simple; it only has two variables:

```
template<class DataType>
class TreeIterator
{
public:
    Node* m_node;
    DListIterator<Node*> m_childitr;
};
```

The first variable is a pointer to the current node, and the second is a DListIterator, which points to the current child in the current node. If you don't understand why there are two iterators, please go back and play around with Graphical Demonstration 11-1. That demo gives you a good idea of why two iterators are needed.

This iterator class neatly encapsulates both iterators into one class so that they are easier to use.

The Basic Iterator Functions

These are the basic iterator functions, which allow you to create an iterator and set it up to point to a node.

The Constructor

The tree iterator is different from the linked list iterators you've used before. Instead of getting an iterator from a general list object, you pass a tree node into the iterator's constructor.

```
// constructor
TreeIterator( Node* p_node = 0 )
{
    *this = p_node;
}
```

This function calls the assignment operator, which I go over next. Before that, however, I want to clarify how an iterator is used.

NOTE

If you do not pass a node into the constructor, this function still works. The = 0 in the parameter list says, "If the user didn't pass anything in as a parameter, then use the value 0 instead."

Say you have a tree node that you want to get an iterator for, like this:

```
Tree<int>* node = new Tree<int>;
```

This node is just a plain integer tree node with no children and no parents. It doesn't really matter, though; this same method works with any tree node pointer. Now you want to get an iterator to that node, which you can do in one of two ways:

```
TreeIterator<int> itr( node );
itr = node;
```

The first line of code uses the constructor to make the iterator point to the node. The second method uses the assignment operator to do the same thing.

The Assignment Operator

The assignment operator for the iterator class is somewhat confusing at first, but you'll understand it after you work with trees a little more.

```
void operator= ( Node* p_node )
{
    m_node = p_node;
    ResetIterator();
}
```

The first strange thing is that the operator takes a tree node pointer as a parameter, which is something you don't usually see when dealing with assignment operators. However, when you're working with nodes, you usually are dealing with node pointers, right? The function takes pointers as a parameter so it is easier to work with.

So the function takes a node pointer as a parameter. It then sets the `m_node` pointer to point to the tree node. After that, the function calls the `ResetIterator` helper function, which makes the `m_childitr` iterator point to the first child of `p_node`. That particular sequence of code is called often in the iterator, and rather than copying the code 20 times, I placed it into a function of its own.

The ResetIterator Function

This resets the child iterator, and it is meant to be called whenever the `m_node` pointer is changed. It is really just a helper function and is not meant to be called outside of the iterator class.

```
void ResetIterator()
{
     if( m_node != 0 )
     {
          m_childitr = m_node->m_children.GetIterator();
     }
     else
     {
          m_childitr.m_list = 0;
          m_childitr.m_node = 0;
     }
}
```

The first part checks to see whether the node is 0. If not, then it resets `m_childitr` to point to the child list of `m_node`. If the node is 0, then it is invalid, so you need to make the child iterator invalid, too. If not, then the child iterator might be pointing to the child list of a different node.

The Vertical Iterator Functions

The following functions are the so-called *vertical* iterator functions because they deal with moving the iterator up and down through the tree.

The Root Function

This simple function moves the iterator to the root of the tree. Notice how this function would not be possible if the tree node class didn't point to its parent.

The code is pretty simple, so I'm not going to bother pasting it here. The basic premise is this: While the current node's parent is not 0, move the iterator up one level.

NOTE

Note that if you invalidate the iterator somehow, you can't call this function to move back to the root because the iterator has no idea of where the root node actually is. You need to manually reset the iterator using the assignment operator.

The Up Function

The `Up` function is very similar to the `Root` function, except that it moves the iterator up only one level, and it might actually go past the root node. Because of that, this function could possibly invalidate the iterator if you make it go past the root.

```
void Up()
{
     if( m_node != 0 )
     {
          m_node = m_node->m_parent;
     }
     ResetIterator();
}
```

This function makes sure that the node is valid before it does anything. If so, then it moves the iterator up to the previous node.

The Down Function

This function is the opposite of `Up`; it moves the iterator downward to the current child iterator. However, if the child iterator isn't valid, this function doesn't do anything.

```
void Down()
{
     if( m_childitr.Valid() )
     {
          m_node = m_childitr.Item();
          ResetIterator();
     }
}
```

The Horizontal Iterator Functions

The *horizontal* functions of a tree iterator are called so because they allow you to move the current child iterator back and forth, like a linked list iterator. They are `ChildForth`, `ChildBack`, `ChildStart`, and `ChildEnd`.

However, there really is no point in pasting the code here; these functions are all one line long and directly call the `DListIterator` version of the same function.

The Other Functions

The iterator class has several other functions that make it easier to add and remove nodes to the tree and to access their contents.

The functions are `AppendChild`, `PrependChild`, `InsertChildBefore`, `InsertChildAfter`, `RemoveChild`, `ChildValid`, and `ChildItem`. Notice something about all of these? These functions all correspond to functions within the linked list classes!

Because all of these functions directly call linked list iterator functions, there is no need for me to paste them here, either.

Building a Tree

There are two common methods of building trees: top-down and bottom-up. They are used in different situations, depending on what you want a tree to do.

Top Down

I've already shown you how to build a tree one way, which is called *top-down* construction. I used this method in Graphical Demonstration 11-1. In this method you create the root node of the tree first, and then add children from there.

Bottom Up

There is another way to build trees, however, and it is very different from top-down. This method is called *bottom-up* construction, mainly because the tree is built with the bottom nodes first (the leaves) and then expanded upward. This method of tree construction is not used as often as top-down, but there are several uses for it. For example, when building *Huffman trees*, bottom-up construction is used. You'll see what Huffman trees are in Chapter 21, "Data Compression," when I show you different methods of compressing data.

Traversing a Tree

You can traverse the nodes in a tree in many ways. Using the tree iterator is one of them, but that method is sometimes too difficult to use if you just want to perform a function on every single node in the tree.

You can use two different simple methods when you want to traverse a tree, and they are both recursive.

The Preorder Traversal

The first method I show you is the *preorder* traversal. You'll see why it is called so when you look at the pseudo-code:

```
Preorder( node )
    Process( node )
    For each child
        Preorder( child )
    end For
end Preorder
```

This algorithm accepts a node as a parameter and uses a function named `Process`. You shouldn't care what `Process` does; it is just a function that does something to the node.

The function first processes the node that is passed into the function. It then loops through each child and calls `Preorder` on each child node.

Let me show you how this is run through on a simple tree. Figure 11.9 shows the tree.

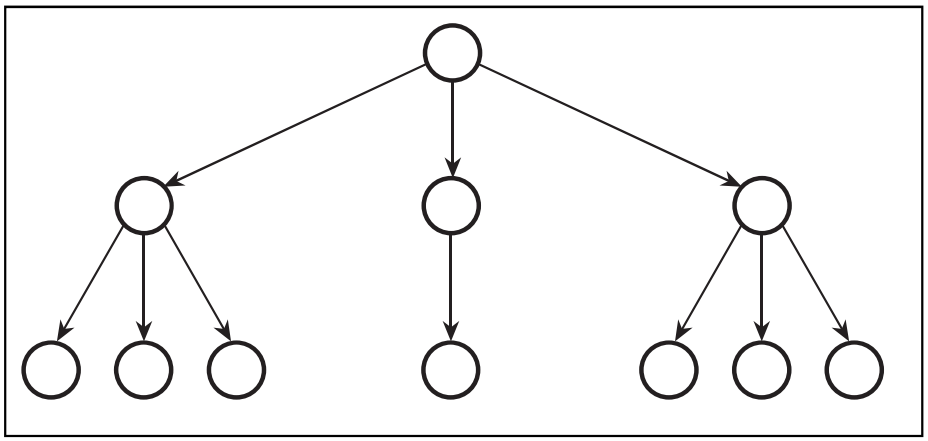


Figure 11.9

This is the sample tree that is used in the traversal examples.

Now, you call `Preorder` on the root node, so that gets processed first. The root has three children, so the function calls `Preorder` on each child. The function calls `Preorder` on the leftmost child first, which means that the leftmost child gets processed second. Now the function loops through all of the children of the leftmost child of the root, so the three children are processed third, fourth, and fifth. Now the function jumps back up to the second child of the root and repeats the process.

The order in which the nodes are processed is shown in Figure 11.10.

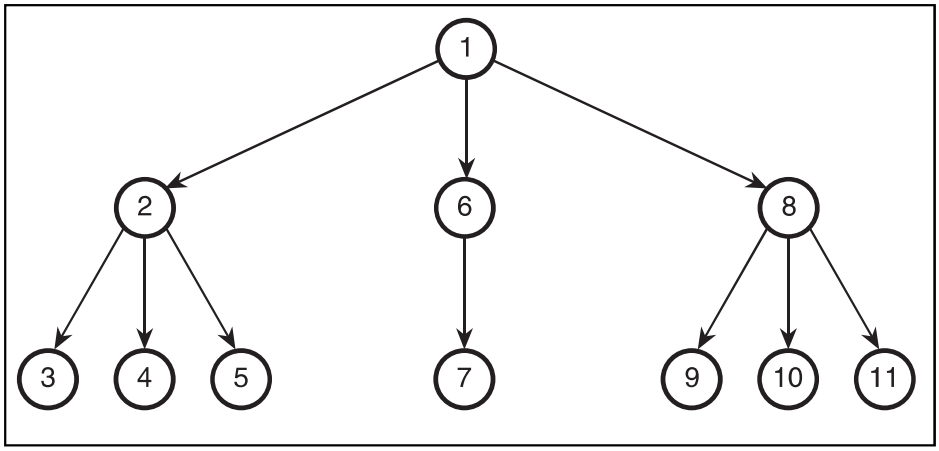


Figure 11.10
This is the order in which nodes are processed with `Preorder`.

In a preorder traversal, each subtree is processed before the next subtree is processed. You can see why it is called a preorder search from the algorithm; the current node is processed *before* the children.

Coding the Preorder Function

Now you need to actually put the algorithm into code. The method I used for the `Preorder` function is very flexible, and the code looks very ugly because of it.

The `Preorder` function takes a function pointer as a parameter. You've seen function pointers before in Chapter 8, "Hash Tables," but in case you aren't familiar with them, you can read more about them in Appendix A, "A C++ Primer."

```
1: template <class DataType>
2: void Preorder( Tree<DataType>* p_node, void (*p_process)(Tree<DataType>*) )
3: {
4:      p_process( p_node );
5:      DListIterator<Tree<DataType>*> itr = p_node->m_children.GetIterator();
6:      for( itr.Start(); itr.Valid(); itr.Forth() )
7:          Preorder( itr.Item(), p_process );
8: }
```

First off, the function is a template function. This allows `Preorder` to work on any type of tree easily.

On line 2, the function takes a node pointer and a function pointer as parameters. The function that is passed into `Preorder` is a simple function, which takes a `Tree` pointer as a parameter and doesn't return anything. I show you how to use this shortly.

On line 4, the p_process function pointer is called on the node.

On line 5, an iterator to the child list of the node is retrieved, and the function uses this iterator to loop through each child node and call Preorder on them in lines 6 and 7.

Using the Function Pointer

Say you have a tree of integers already built, and its name is g_tree. Now, you want to add together every number in the tree, but you don't want to bother using an iterator to do this.

So, you create a function called sum, which sums together the contents of tree nodes and puts them into a global integer named g_sum:

```
void sum( Tree<int>* p_node )
{
    g_sum += p_node->m_data;
}
```

Now, all you need to do to sum together the values of all the nodes in g_tree is to call these two lines of code:

```
g_sum = 0;
Preorder( g_tree, sum );
```

The first line clears the sum, and the second line traverses the tree, calling sum on each node.

The Postorder Traversal

The other major traversal type for trees is called the *postorder* traversal. If the pre-order traversal was called *pre* order because it processed the current node *before* the child nodes, what do you think the *post* order traversal does?

That's right—it processes the current node *after* the child nodes.

```
Postorder( node )
    For each child
        Postorder( child )
    end For
    Process( node )
end Postorder
```

If you were to postorder traverse the tree from Figure 11.9, the nodes would be processed in the order shown in Figure 11.11.

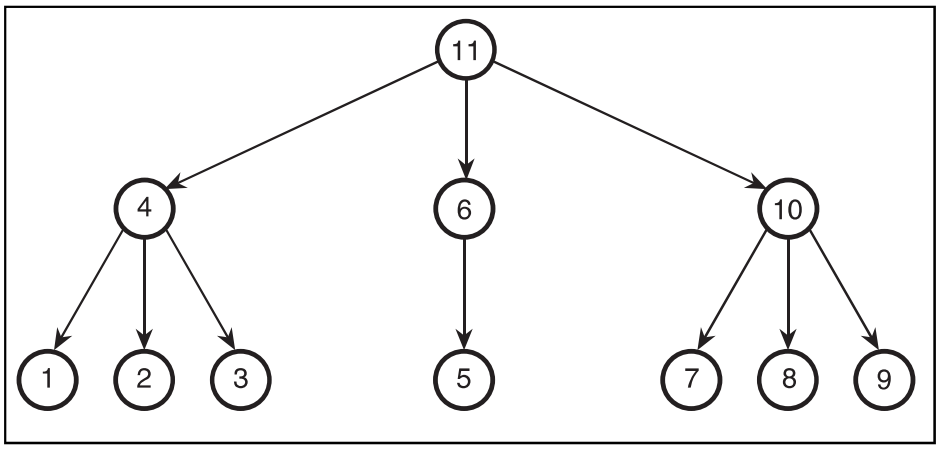


Figure 11.11

This is the order in which the nodes are processed using the postorder traversal.

This time, `Postorder` is called on the first child of the root, then the first child of that node, so that the first node to be processed is a leaf node. You can see from the figure that every child node is processed before its parent node.

I won't bother to paste the code for the actual `Postorder` function because it is so similar to the `Preorder` function. If you want to see it, it is in the tree.h file in the \structures\ directory on the CD.

Graphical Demonstration: Tree Traversals

This is Graphical Demonstration 11-2, which can be found on the CD in the directory \demonstrations\ch11\Demo02 - Tree Traversal\.

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.

This demonstration is very simple; it only has three buttons, as shown in Figure 11.12.

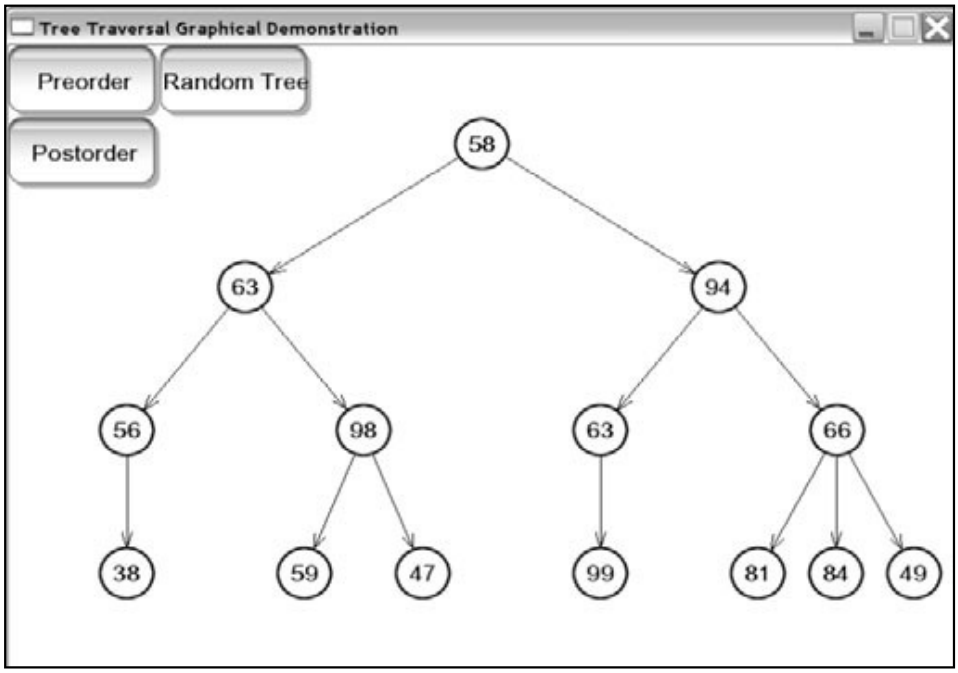


Figure 11.12

Here is a screenshot from Graphical Demonstration 11-2.

The Random Tree button generates a new random tree, as in the previous demo.

The other two buttons, Preorder and Postorder, make the demo go into an animation. The demo highlights the nodes using the preorder or postorder algorithms at 750 millisecond intervals. If you clicked Preorder, for example, Node 58 would be highlighted first, and then 63, and then 38, and then 98, and so on.

Game Demo 11-1: Plotlines

This is Game Demonstration 11-1. It is on the CD in the directory \demonstrations\ch11\Game01 - Plotlines\.

Compiling the Demo

This demonstration uses the SDLHelpers 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.

For years and years, games have been *linear* with their stories and plots. What does this mean? You start the game and you play around, progressing from level to level, until you beat the game. Figure 11.13 shows how the levels progress throughout the game. Notice that it is a straight line, which is where the term *linear* comes from.

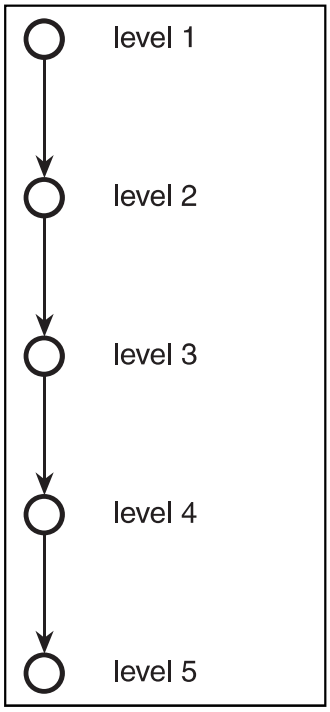


Figure 11.13

This is a linear level progression. Each level leads to the next.

Well, these types of games can be fun the first time through, but they tend to get boring. If you play the same levels over and over again, the game could get boring really quickly.

Now, imagine a game where the actions you take in the game directly affect the plot of the game. Say that at one point in the game, you are required to make a choice that will cause the game to branch out, and everything that happens during the rest of the game happens as a result from your choice.

For example, at level 2, you're required to make a choice; from that point, the game is different, depending on the choice you made. Figure 11.14 shows this.

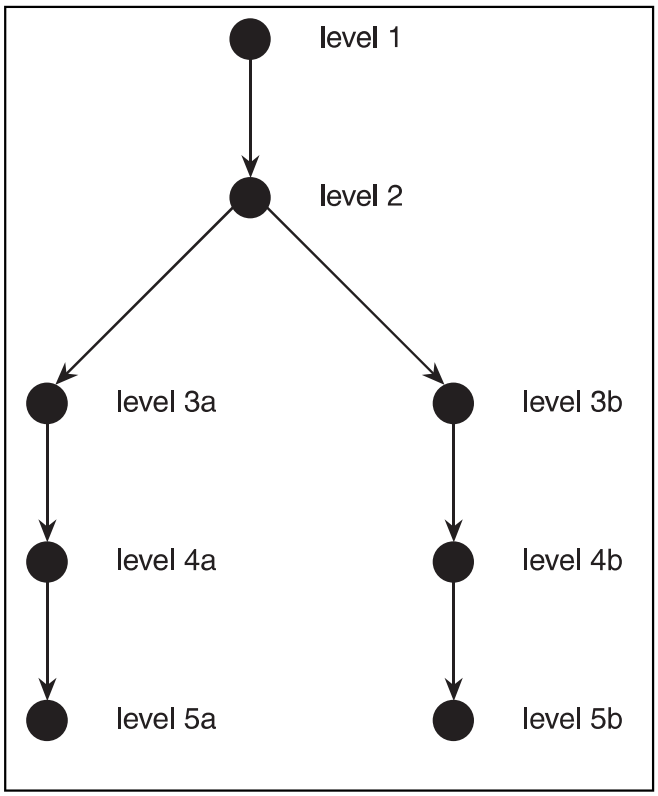


Figure 11.14

This is a branching level progression. You can choose which path to take at level 2.

Now, whenever you play through the game, you can go with branch *a* the first time and branch *b* the second time!

It turns out that trees are the ideal structure to store game data like this. You can see how the level progression from Figure 11.14 looks like a tree, albeit a basic one.

Using Trees to Store Plotlines

Obviously, creating a branching plotline takes a lot of work, and because the purpose of the demo is to show you how the data structure works, this demo doesn't really have a plot. Instead, I'll call the different story branches *a, b, c,* and so on. If you want, you can make up a plotline for yourself in your head—just don't make it too bad; I hate games with bad plots!

The first thing you need to do is create a storyline. In this little demo, the storyline will look like Figure 11.15.

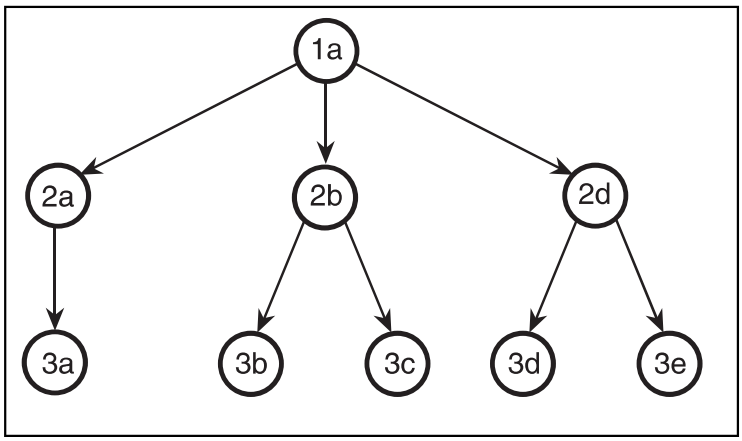


Figure 11.15

Here is the branching plotline for the demo.

Declaring the Tree

The premise of the demo is simple; when a level is completed, the player has a choice of which level to go to next. This information is stored in a tree:

```
Tree<int>* g_tree;
TreeIterator<int> g_itr;
```

The `g_tree` pointer will always point to the root of the tree, and the `g_itr` pointer will point to the current level in the tree.

The tree only stores integers, which represent which tile the level is made up of. Each level in the game demo has a different tile.

Initializing the Tree

The root of the tree is then initialized:

```
g_tree = new Tree<int>;
g_tree->m_data = 0;
g_itr = g_tree;
```

The root node is created with the value 0, which means that the player starts out on level 0. On line 3, the global iterator is assigned to point to the root node, which means that the player starts out on the root level.

After the root is initialized, a temporary iterator named `itr` is created so that I can build the tree with it. Using this iterator, I build the tree using the iterator functions:

```
TreeIterator<int> itr;
Tree<int>* node;
itr = g_tree;
// add the '2a' branch
```

```
node = new Tree<int>;
node->m_data = 1;
itr.AppendChild( node );
```

This code shows the addition of level 2a to the tree. The other seven levels of the tree are added in the same fashion; the iterator is moved around and child nodes are appended to the tree to give you the tree in Figure 11.15.

Changing Levels

Whenever the player "wins" a level in the demo, the demo switches to a state where it selects the next level.

The screen that draws the levels that are available for choosing uses the child iterator of `g_itr` to loop through each child and draw it.

```
for( g_itr.ChildStart(); g_itr.ChildValid(); g_itr.ChildForth() )
{
    // draw the level that the current child contains
}
```

When the user selects a level, the child iterator is moved to the correct level. The integer `x` will contain the number of the child which the player selected.

```
g_itr.ChildStart();
while( x > 0 )
{
    g_itr.ChildForth();
    x--;
}
g_itr.Down();
```

When the child iterator is in the correct place, the `Down` function is called, moving the iterator to the next level.

Playing the Game

The game starts off with a little dude standing on some weird alien world at the top left corner of the screen. Your mission? You are to use the arrow keys on the keyboard to successfully walk him off the edge of the screen to the right. It might be difficult and you might not succeed, but you'll make me proud by trying!

Okay, you really can't lose. There are no enemies or obstacles. Figure 11.16 shows the opening screen.

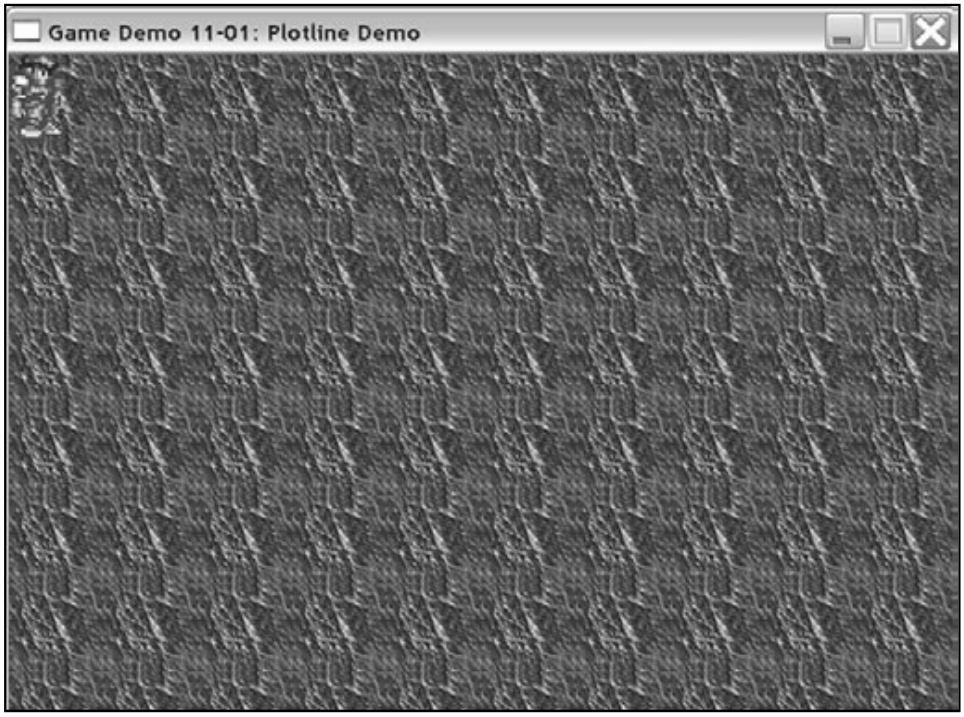


Figure 11.16
This is a screenshot of Level 0.

After you have successfully moved your little dude across the screen to the right, the level selection screen appears, as shown in Figure 11.17.

Figure 11.17
This is the level selection screen.

You use the mouse to click on one of the tiles to select the next level. That's pretty much all there is to the demo.

Conclusion

One thing you should realize about trees is that they are complex structures. They are obviously not suitable for storing any types of data, like arrays and linked lists are, so that makes trees a more specialized structure.

Only certain types of data can be stored in trees, but which kind? It turns out that hierarchical data fits nicely into trees, but that's not all. I only went into one use of trees; there are many.

For example, you could store AI decision paths into a tree. Imagine the AI process of a character within a shoot-'em-up game, as shown in Figure 11.18.

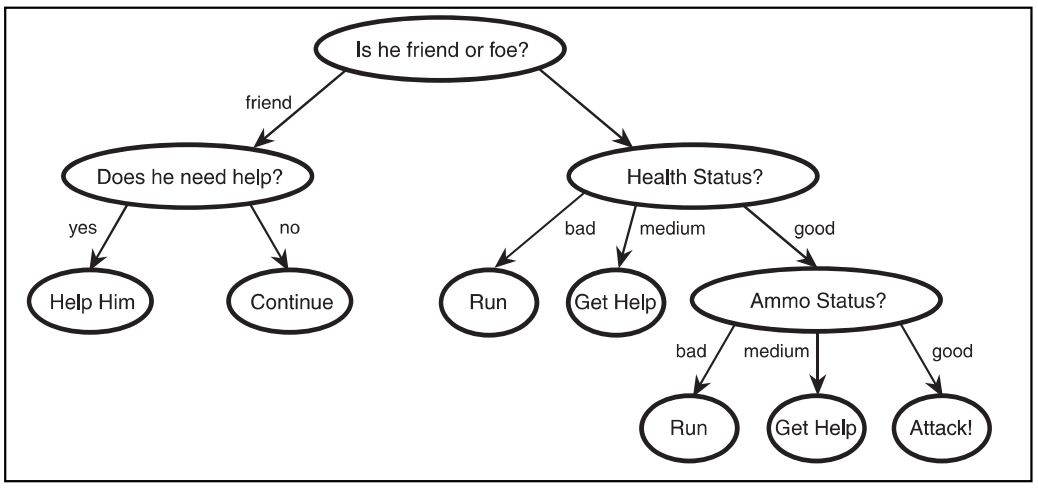


Figure 11.18
This is an AI decision tree showing the thought process of a character when he sees another character in the game.

So, as you can see, there are tons of uses for trees. The main purpose of this chapter was to introduce you to the concepts of trees and practice your recursion skills.

The next few chapters go over some more specialized trees and their uses.