The particular type of linked list implemented in Listing 7.10.1 is called a *singly linked list*. This is the most basic type of linked list. One obvious modification to this is to add a reference in each node to the preceding node. This creates a list type known as a *doubly linked list* and allows the list to be traversed in either direction. Other data structures that can be built directly from linked lists are stacks and queues. In short, linked lists are excellent structures for data that is linearly ordered.

7.11 Binary Tree Implementation

The binary tree and its many variants are the mainstays of the data structure family. A variation of the linked list, binary trees excel in situations where data must be quickly organized and quickly retrieved in a particular order. As shown in Figure 7.11.1, a binary tree consists of nodes with pointers to up to two other nodes. The nodes pointed to are generally referred to as the *children nodes*, generically called *left* and *right*, and the current node is the *parent node* of the children.

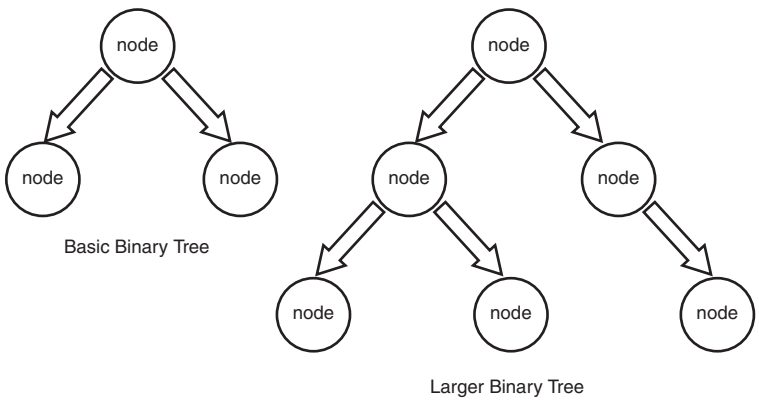

Figure 7.11.1 Binary tree.

Let's construct a class that implements the basic binary tree node. The class consists of three variables: the data value associated with the current node and the two children nodes. Because this class simply deals with the node concept, the only methods a node can have are to read out the values of the node and all its children. Listing 7.11.1 demonstrates the standard practice of traversing a tree: Get all the left children before all the right children. The one option remaining is where to put the value of the current node: before all its children, after all its children, or between the left and right children. These three options are called, respectively, preorder, post-order, and in-order traversal. To illustrate, Figure 7.11.2 shows a simple tree.

Using a preorder traversal, the values from this tree are retrieved as "a", "b", "c". For post-order traversal, the order of retrieval is "b", "c", "a". Finally, in-order traversal produces the list "b", "a", "c".

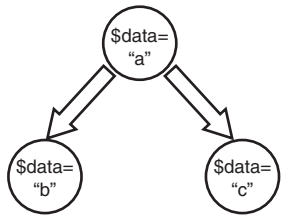

Figure 7.11.2 Binary tree example.

Listing 7.11.1 **Binary Tree Implementation**

```
<?php
// Define a class to implement a binary tree
class Binary_Tree_Node {
    // Define the variable to hold our data:
    public $data;
    // And a variable to hold the left and right objects:
    public $left;
    public $right;

    // A constructor method that allows for data to be passed in
    public function __construct($d = NULL) {
        $this->data = $d;
    }

    // Traverse the tree, left to right, in pre-order, returning an array
    // Preorder means that each node's value preceeds its children.
    public function traversePreorder() {
        // Prep some variables.
        $l = array();
        $r = array();

        // Read in the left and right children appropriately traversed:
        if ($this->left) { $l = $this->left->traversePreorder(); }
        if ($this->right) { $r = $this->right->traversePreorder(); }

        // Return a merged array of the current value, left, and right:
        return array_merge(array($this->data), $l, $r);
    }

    // Traverse the tree, left to right, in postorder, returning an array
    // Postorder means that each node's value follows its children.
    public function traversePostorder() {
        // Prep some variables.
```

Listing 7.11.1 **Continued**

```
        $l = array();
        $r = array();

        // Read in the left and right children appropriately traversed:
        if ($this->left) { $l = $this->left->traversePostorder(); }
        if ($this->right) { $r = $this->right->traversePostorder(); }

        // Return a merged array of the current value, left, and right:
        return array_merge($l, $r, array($this->data));
    }

    // Traverse the tree, left to right, in-order, returning an array.
    // In-order means that values are ordered as left children, then the
    //  node value, then the right children.
    public function traverseInorder() {
        // Prep some variables.
        $l = array();
        $r = array();

        // Read in the left and right children appropriately traversed:
        if ($this->left) { $l = $this->left->traverseInorder(); }
        if ($this->right) { $r = $this->right->traverseInorder(); }

        // Return a merged array of the current value, left, and right:
        return array_merge($l, array($this->data), $r);
    }
}

// Let's create a binary tree that will equal the following:    3
//                                                             / \
//                                                            h   9
//                                                               / \
// Create the tree:                                             6   a
$tree = new Binary_Tree_Node(3);
$tree->left = new Binary_Tree_Node('h');
$tree->right = new Binary_Tree_Node(9);
$tree->right->left = new Binary_Tree_Node(6);
$tree->right->right = new Binary_Tree_Node('a');

// Now traverse this tree in all possible orders and display the results:

// Pre-order: 3, h, 9, 6, a
echo '<p>', implode(', ', $tree->traversePreorder()), '</p>';
```

Listing 7.11.1 **Continued**

```
// Post-order: h, 6, a, 9, 3
echo '<p>', implode(', ', $tree->traversePostorder()), '</p>';

// In-order: h, 3, 6, 9, a
echo '<p>', implode(', ', $tree->traverseInorder()), '</p>';
?>
```

When used in conjunction with ordering logic between nodes, this structure provides a wonderful organization method. At any node, all the elements that satisfy a certain condition are located at the left node and its children, and all the elements that satisfy the converse of the condition are located at the right node and its children. When inserting a new element, all you need to do is start at the first, or "root" node, of the tree, discover which side the element belongs on, and then descend that side until its proper location is found. Similarly, to read back out the information, it is a simple matter of traversing the tree using the in-order traversal method.

To demonstrate, Listing 7.11.2 takes the simplest case of ordering: less than, greater than, and equal to. For each node, any values that are less than or equal to the value of the current node are placed in the left child and its children; any value greater than the value of the current node are placed in the right child and its children. This concept is encapsulated in the class `Sorting_Tree`. The one member, `$tree`, contains the root `Binary_Tree_Node` of the tree. There are two methods: one to insert a new value and one to return an array of the ordered values.

Listing 7.11.2 **Self-Sorting Binary Tree**

```
// Define a class to implement self sorting binary tree
class Sorting_Tree {
    // Define the variable to hold our tree:
    public $tree;

    // We need a method that will allow for inserts that automatically place
    // themselves in the proper order in the tree
    public function insert($val) {
        // Handle the first case:
        if (!(isset($this->tree))) {
            $this->tree = new Binary_Tree_Node($val);
        } else {
            // In all other cases:
            // Start a pointer that looks at the current tree top:
            $pointer = $this->tree;
            // Iteratively search the tree for the right place:
            for(;;) {
                // If this value is less than, or equal to the current data:
                if ($val <= $pointer->data) {
```

Listing 7.11.2 **Continued**

```
                    // We are looking to the left ... If the child exists:
                    if ($pointer->left) {
                        // Traverse deeper:
                        $pointer = $pointer->left;
                    } else {
                        // Found the new spot: insert the new element here:
                        $pointer->left = new Binary_Tree_Node($val);
                        break;
                    }
                } else {
                    // We are looking to the right ... If the child exists:
                    if ($pointer->right) {
                        // Traverse deeper:
                        $pointer = $pointer->right;
                    } else {
                        // Found the new spot: insert the new element here:
                        $pointer->right = new Binary_Tree_Node($val);
                        break;
                    }
                }
            }
        }
    }

    // Now create a method to return the sorted values of this tree.
    // All it entails is using the in-order traversal, which will now
    // give us the proper sorted order.
    public function returnSorted() {
        return $this->tree->traverseInorder();
    }
}

// Declare a new sorting tree:
$sort_as_you_go = new Sorting_Tree();

// Let's randomly create 20 numbers, inserting them as we go:
for ($i = 0; $i < 20; $i++) {
    $sort_as_you_go->insert(rand(1,100));
}

// Now echo the tree out, using in-order traversal, and it will be sorted:
// Example: 1, 2, 11, 18, 22, 26, 32, 32, 34, 43, 46, 47, 47, 53, 60, 71,
//   75, 84, 86, 90
echo '<p>', implode(', ', $sort_as_you_go->returnSorted()), '</p>';
?>
```

The value of using a binary tree should now be apparent. The structure orders itself in a highly efficient manner; while inserting a new value, at every existing node, you get halfway closer to the value's final location. This method is as efficient as the QuickSort algorithm already mentioned for arrays. Though there are more efficient methods for specific applications, a sorted binary tree works much better than arrays for most problems. Also, at any point during processing, the data already stored will be sorted and ready for whatever the next step is. No post-processing, cleanup, or re-sorting when new data is added is required.

Two aspects about the code itself should be pointed out. First, two distinct classes are used to implement the concept of the binary tree. For this example, you could easily have defined a single class that contains both the node concept and the tree concept. Another approach would be to create the `Sorting_Tree` class as an extension of the `Binary_Tree_Node` class. In general, any of these approaches is perfectly valid. The choice used here is simply a convenience to create self-contained examples that can easily be used independently. Many different guidelines and philosophies exist that can assist you in defining classes and their relationships. The common rule of most any approach; however, is to keep it simple.

The second aspect is the choice to use an iterative algorithm to traverse the tree to insert new values. Both from a logical and code-conciseness viewpoint, a recursive solution would have been more appropriate. The iterative approach was chosen because the potential exists that the tree will be large. Using a recursive solution, memory resources would be consumed for each level of recursion that takes places, ultimately leading to an exhaustion of resources. The iterative solution takes no more resources than that used by the tree already.

The general topic of trees themselves can take many chapters of a data structure book. The next obvious extension is to allow a node to have more than two children, called an *n-ary tree*. Using such concepts, structures such as Self-Balancing (Btree) trees, which attempt to keep the number of levels deep a tree becomes to a minimum, can be developed that greatly enhance search efficiency and data storage. Also, n-ary trees allow for complicated hierarchical data to be organized and stored. N-ary trees form the basis upon which the XHTML Document Object Model (DOM) and XML itself are built.

Finally, Listing 7.11.2 demonstrates the power of object-oriented design. All the trees mentioned share common characteristics, such as inserting new elements and retrieving them. Because the actual details of such functions are encapsulated into the individual classes, a larger application using such objects does not need to know exactly whether the object is a binary tree, n-ary tree, or some other type of tree. It only needs to know that the object has the methods needed to get the task completed. An application will simply have a statement such as `$anobject->insert($data);`. Whether the object is of any particular instance is inconsequential to the application. The application code will be more concise; focusing on solving the bigger problem while the details of the data structures themselves can be optimized separately without affecting the larger application.

This page intentionally left blank