```
 1 // Tree interface; details are similar to List
 2
 3 public interface Tree
 4 {
 5     boolean isEmpty( );
 6     void    makeEmpty( );
 7 }
 8
 9
10 // TreeItr interface; maintains "current position"
11
12 public interface TreeItr
13 {
14     void    insert( Object x ) throws DuplicateItem;
15     boolean find( Object x );
16     void    remove( Object x ) throws ItemNotFound;
17
18     void    gotoRoot( );
19     void    firstChild( );
20     void    nextSibling( );
21     boolean isValid( );
22     Object  retrieve( );
23 }
```

Figure 6.15 Interfaces for the abstract tree class and iterator

With this in mind we can see what kinds of operations a tree supports. We want to add, search, and remove from the tree. Much like the linked list, we want to define an iterator that allows us access to do this by maintaining the notion of a current internal node. We need to provide methods that allow us to traverse the tree in an orderly manner. Thus, in Figure 6.15, we define the tree and iterator interfaces using the same method as for linked lists. The operations `gotoRoot`, `firstChild`, and `nextSibling` are sufficient to allow us to traverse the tree in one of several orders. In Chapter 17, we describe a hierarchy of classes that supports traversal of trees using only the (logical) methods `firstChild`, `retrieve`, `isValid`, and `nextSibling`. This includes the classes `InOrder`, `PreOrder`, `PostOrder`, and `LevelOrder`.

Expression trees are used in parsing. In an expression tree, the value of a node is the result of applying the operand at the node, using the children as operands.

A second application of trees, called the *expression tree*, is shown in Figure 6.16. In an expression tree, the value of a node is the result of applying the operator at the node, using the children as operands. Leaves evaluate to themselves. Consequently, the expression tree shown in Figure 6.16 evaluates to `(a+b)*(c-d)`. Expression trees and the related parse tree are crucial data structures in the parsing and code-generation stages of the compiler. More details are in Section 11.2.

A *binary tree* has at most two children per node.

The expression tree in Figure 6.16 is a *binary tree* because the number of children is limited to at most two per node. An important use of the binary tree is examined in the next section.