CMPSCI 187: Programming With Data Structures

Lecture #27: Expression Trees
18 November 2011

Expression Trees

- L&C's `BinaryTreeADT` Interface
- Some Uses of Binary Trees
- Review of `LinkedBinaryTree` and `ArrayBinaryTree` Classes
- Expression Trees and the `ExpressionTree` Class
- Evaluating an Expression Tree
- The Postfix Evaluator Revisited

L&C's `BinaryTreeADT` Interface

- Just as with lists, we have an ADT defining operations that are common to all binary trees. For each type of binary tree, we can then extend this interface to add the appropriate new operations.

- Note that this interface has no operations to change the tree, because each extension will have its own methods to add or remove nodes.

```
public interface BinaryTreeADT<T> {
   public T getRoot ( );
   public boolean isEmpty ( );
   public int size ( );
   public boolean contains (T target);
   public T find (T target);
   public String toString ( );
   public Iterator<T> interatorInOrder ( );
   public Iterator<T> interatorPreOrder ( );
   public Iterator<T> interatorPostOrder ( );
   public Iterator<T> interatorLevelOrder ( );}
```

Some Uses Of Binary Trees

- A binary tree can be used to implement an ordered list more efficiently than an array. Next week we will see **binary search trees**, binary trees where every node contains an element from some comparable class. If x is the element at a node n, we make sure that all the nodes in the subtree under n's left child are less than n, and all the nodes in the right subtree are greater than n. We can then insert or delete an element by going down the tree to find the right place for the action, and the edit to the tree does not require shifting.

- If the tree is balanced, this will be O(log n) time instead of O(n) time for the corresponding array operation.

- After Thanksgiving we will look at **heaps**, complete (and therefore balanced) binary trees where the elements are "somewhat sorted" -- the element at a node is less than the elements at its children, so the lowest element is at the top. Heaps will let us implement an O(log n)-time priority queue.

`LinkedBinaryTree` and `ArrayBinaryTree`

- The `LinkedBinaryTree` class uses `BinaryTreeNode` objects as its elements -- each node has a pointer to its left and right children, and these pointers are `null` if that child does not exist. We used recursion to find whether a value is stored in the tree -- it is in a subtree if it is stored at the root of the subtree, in the left subtree, or in the right subtree. We had a similar recursive method to count the descendants of a node.

- The `ArrayBinaryTree` class stores its elements in an array, using arithmetic definitions of “left child”, “right child”, and “parent” to impose a tree structure on the linear sequence. We implemented the `find` operation as a simple linear search, and made an in-order iterator by recursively copying the nodes to a list and then making an iterator for the list.

- Note that in an unbalanced tree, the array could have lots of `null` entries and thus take up much memory than just storing the elements would need.

Expression Trees: The `ExpressionTree` Class

- Remember that in Chapter 3 we used a stack to evaluate **postfix expressions**. These were made from numbers and the arithmetic operators +, -, *, and /, with each operator coming after its two operands.

- A recursive definition of such an expression would be “a number, or two expressions followed by an operator”. It is natural to represent such an expression by an **expression tree**, a binary tree with numbers at the leaves and operators at the internal nodes. The postfix expression corresponding to an expression tree is given by the postfix traversal of the tree.

- In Section 9.5 L&C define an `ExpressionTree` class whose objects represent arithmetic expressions in this way. The element at each node is an `ExpressionTreeObject`, which can hold either a number or an operator and knows which it is holding.

Some Code for the class `ExpressionTree`

- The basic constructor makes a tree from a node and two subtrees.

```
public class ExpressionTree extends
       LinkedBinaryTree<ExpressionTreeObj> {
   public ExpressionTree ( ) {super( );}
   public ExpressionTree (ExpressionTreeObj element,
     ExpressionTree leftSubtree, ExpressionTree rightSubtree) {
      root = new BinaryTreeNode<ExpressionTreeObj> (element);
      // attach left, right subtrees, compute “count” recursively
   }

public class ExpressionTreeObj {
   private int termType;
   private char operator;
   private int value;
   public ExpressionTreeObj (int type, char op, int val)
      {termTyp = type; operator = op; value = val;}
   public boolean isOperator ( ) {return (termType == 1);}
   public char getOperator ( ) {return operator;}
   public int getValue ( ) {return value;}}
```

Evaluating an `ExpressionTree`

- The basic recursive strategy is obvious -- to evaluate a node, we evaluate its left and right children (if they exist), then apply the operator to those two values. The base of the recursion is a leaf node, where we just return the value of the node.

```
public int evaluateTree ( ) {return evaluateNode (root);}

public int evaluateNode (BinaryTreeNode<ExpressionTreeObj> root) {
   int result, op1, op2;
   ExpresssionTreeObj temp;
   if (root == null) result = 0;
   else {temp = root.element; // L&C, confused, have a cast here
      if (temp.isOperator ( )) {
         op1 = evaluateNode (root.left); // BTN field protected
         op2 = evaluateNode (root.right); // but in package, ok
         result = computeTerm (temp.getOperator( ), op1, op2);}
      else result = temp.getValue ( );
   return result;}
```

The Postfix Evaluator Revisited

- When we evaluated postfix expressions before, we had a stack of `int` values to keep the arguments that were waiting for their operators.
- Now if we can build an expression tree from our postfix expression, we already have code to evaluate the tree and thus evaluate the expression.
- We divide the input text into tokens and keep a stack of *expression trees* instead of `int` values. When we get a number, we push a one-node tree onto the stack. When we get an operator, we pop two expression trees off of the stack, make them the left and right subtrees of a new expression tree with the operator at the root, then push the new tree onto the stack.
- At the end we should have exactly one tree on the stack (L&C don't check this, of course). We evaluate this tree and we have our result.
- Note that with the trees as arrays it would be much harder to splice them.