Recursive Implementation of Recursive Data Structures

Jonathan Yackel
Computer Science Department
University of Wisconsin Oshkosh
Oshkosh, WI 54901
yackel@uwosh.edu

Abstract

The elegant recursive definitions of data structures such as lists and trees suggest that recursion can be used to develop software involving these structures. We show how the careful design and implementation of list and binary tree classes in Java and C++ can make the most of recursion both in client code and in the internal implementation of the classes. We show how our approach provides an effective alternative to the traditional iterative treatment of lists and pointer-based treatment of trees in computer science curricula, and that it promotes recursive thinking in a context where recursion is natural.

1 Introduction

Computer science students exposed to a traditional treatments of lists and trees typically learn "pointer chasing" techniques to search or modify the structures. Object-oriented approaches may emphasize the use of iterators rather than direct manipulation of pointers (or references). Iteration and the notion of "getting inside" the lists or trees are the norm. Since lists and trees have elegant recursive definitions, an alternative approach emphasizing recursion has the dual advantages of avoiding involvement in the internals of the structures, and promoting recursive thinking. Recursive treatment of lists and trees is, of course, not a new idea. Various recursive approaches have been advocated by Berman and Duvall [3], Beidler *et al.* [2], Felleisen and Friedman [5], Bloch [4], and Adams [1]. Their work focused on using object-oriented techniques to implement recursive data structures. Our primary interest however, is in providing the client programmer with classes whose public interfaces naturally encourage recursion. The various class implementation strategies investigated by the aforementioned authors are relatively unimportant from our point of view.

2 Recursive definitions

The notion of a list can be elegantly defined via recursion [7]. Let us use the following version:

Definition 1 *A list is either*

i. empty, or

ii. consists of a data item called the head and another list called the tail.

Similarly, the notion of a binary tree can be recursively defined [9]. Here is our version:

Definition 2 *A binary tree is either*

i. empty, or

ii. consists of a data item called the root and two other binary trees called the left subtree and the right subtree.

These definitions contrast with non-recursive definitions found in many data structures texts that generally define lists and trees as collections of nodes connected by edges in certain ways [6, 8, 10, 11].

3 Public interface

The public interfaces for our list and binary tree classes are based directly on the recursive definitions of the data structures. We should be able to test whether list and tree objects are empty. Given a non-empty list we should be able to access and modify the head and tail; for a non-empty tree we want similar capabilities for the root, left subtree, and right subtree.

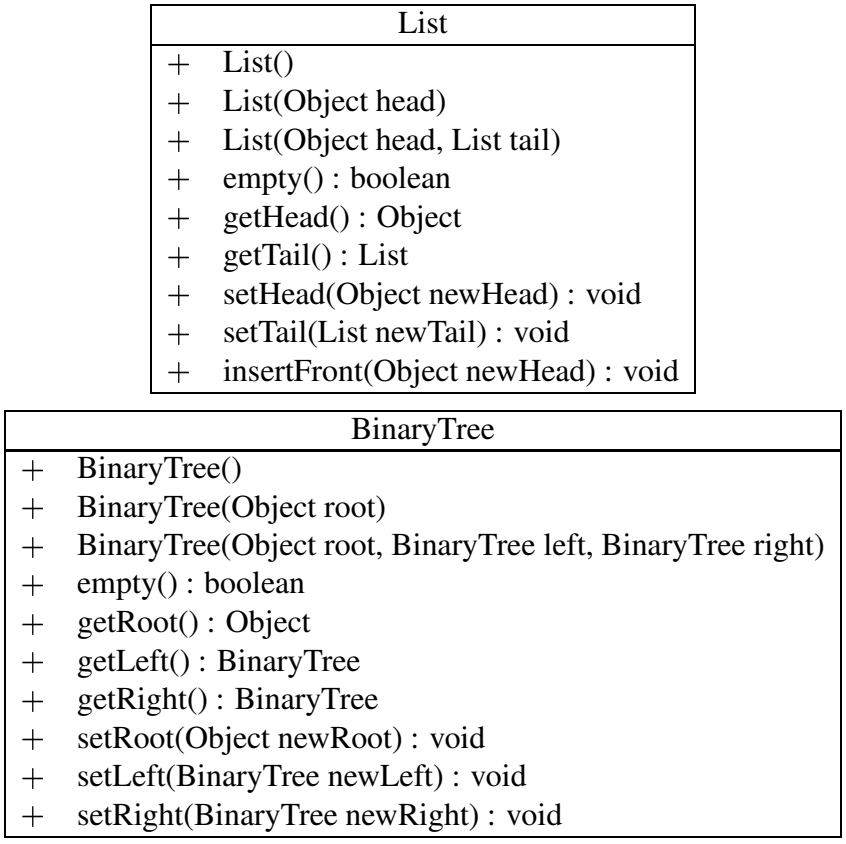

Figure 1: Class diagrams for List and BinaryTree. Only the public methods are indicated.

We also expect that both empty and non-empty objects can be instantiated. Finally, it is natural to grow a list by adding data to the front. The public methods described here have proven sufficient to support the applications considered by the author. Nothing prevents the addition of further public methods. In fact one effective way to use these classes in programming assignments is to require students to write additional methods. See section 5 for an example of such an assignment.

The class diagrams in figure 1 summarize the public interfaces for our List and BinaryTree classes. Of primary importance are the return types of getTail(), getLeft(), and getRight(). Notice that each of these methods returns a List or BinaryTree object, not a node (or pointer to a node). Two important advantages over traditional linked data structures emerge from this feature. First, clients of List and BinaryTree are shielded from dealing with the guts of the data structure, through nodes or (in C++) pointers to nodes. The existence of Node objects is entirely hidden from the client. Our definitions of list and binary tree, after all, do not involve the notion of a "node". Iterators provide similar shielding in the traditional approach to linked data structures. Our approach, however, provides the same effect without involving an auxiliary iterator class. Also, access to the sub-structure of List and BinaryTrees as fully functional Lists and BinaryTrees makes recursive processing on these data structures natural and uncomplicated as will be demonstrated in section 5 of this paper.

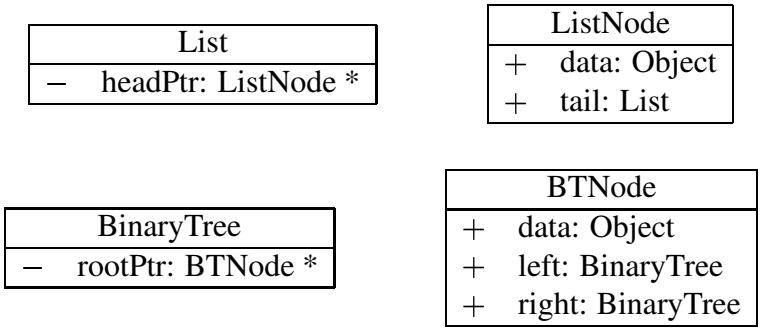

Figure 2: Class diagrams for List and BinaryTree classes, along with their supporting ListNode and BTNode classes. Only the instance variables are shown. The instance variables for the node classes are public since these classes are only visible within the List and BinaryTree classes respectively.

4 Internal implementation

4.1 Instance variables

The instance variables used to implement our List and BinaryTree classes, like the public methods described in the previous section, are motivated by the recursive definitions of the data structures. Our implementations of both List and BinaryTree are based on the use of inner Node classes. It is possible to implement lists and trees without node classes, however, such schemes result in unappealing representations of empty lists and trees [2], or introduce empty and non-empty variants which are inconvenient for stateful (non-functional) programmming [3, 1, 4].

For both List and BinaryTree classes, the single instance variable is a pointer (or reference) to a node of the appropriate variety, *i.e.*, ListNode or BTNode. An empty list or tree is represented by a null pointer. A non-empty List object contains a pointer to a node, which contains a data item (the head) and another list object (the tail). Similarly, a non-empty BinaryTree object contains a pointer to a node, which contains a data item (the root) and two more BinaryTree objects (the left and right subtrees). Figure 2 illustrates this implementation. It is of critical importance that our node classes do not contain pointers (or references) to other node objects as is traditional, but rather fully functional List or BinaryTree objects. This design decision makes implementation of our public methods trivial.

4.2 Methods

The public methods described in section 3 are easy to implement given the choice of instance variables in figure 2. The methods that involve dynamic memory management (the clone method in Java; and the copy constructor, overloaded assignment operator, and destructor in C++), which were not explicitly identified in section 3 but are nevertheless mandatory for a well-behaved class involving dynamic memory, are non-trivial since they involve a traversal of the list or tree. The traditional approach to these methods would use

```
/**
 * Returns a copy of the list.
 * Throws CloneNotSupportedException if cloning of data
 * at a node fails.
 *
 * @return   A copy of the list
 */
public Object clone()throws CloneNotSupportedException{
 if(empty()) return new List<T>();
 else return new List<T>(getHead(),
                         (List<T>)(getTail().clone()));
}//clone
```

Figure 3: Java code for the List clone method. Notice the recursion.

```
// BinaryTree copy constructor
template<class T>
BinaryTree<T>::BinaryTree(const BinaryTree<T>& t){
 if(t.empty()) this->rootPtr = NULL;
 else this->rootPtr = new BTNode<T>(*(t.rootPtr));
}

// BTNode copy constructor
template <class T>
BTNode<T>::BTNode(const BTNode<T>& that):
 data(that.data), left(that.left), right(that.right)
{}
```

Figure 4: C++ code for the BinaryTree copy constructor. Notice the co-recursion between the BinaryTree and BTNode copy constructors.

iterative techniques. In our recursive framework, the copy constructor and clone methods are naturally implemented using recursion. For example, in the Java implementation of List, the clone method is directly recursive, while in the C++ implementation of BinaryTree, the BinaryTree and BTNode copy constructors are co-recursive (see figures 3 and 4). The source code for Java and C++ versions of our List and BinaryTree classes can be found on the world wide web at http://www.uwosh.edu/faculty_staff/yackel/recursion/.

5 Applications

The List and BinaryTree classes described in this paper have been used successfully at the University of Wisconsin Oshkosh in second- and third-semester computer science courses. Programming assignments utilizing the classes fall into two categories: assignments to

write additional methods for the classes and assignments to write "client programs" that use List or BinaryTree objects. Brief descriptions of assignments from both categories are outlined below. The complete assignments are available on the world wide web at http://www.uwosh.edu/faculty_staff/yackel/recursion/.

5.1 Merge sort

In this assignment, students in the second-semester course "Object-oriented programming in C++" were given the C++ source code for the List class. They were required to implement the merge sort algorithm by adding appropriate methods to the class. Recursion was encouraged wherever possible. The intended solution involved writing new methods mergeSort, merge, and length, as well as methods to split a list into two halves. All of these could be done recursively.

5.2 Length and Insert

In this lab assignment (designed to be completed in a sixty-minute class period), students in "Object-oriented programming in C++" were required to write iterative and recursive versions of length and insert methods to be added to the List class. The objective for this assignment was to impress upon students the elegance and power of recursion for list processing.

5.3 Huffman codes

In this assignment, students in the third-semester course "Data Structures" are given the C++ source code for the BinaryTree class and asked to write a HuffmanCode class. A HuffmanCode object would contain, among other things, a BinaryTree object. No modification of the BinaryTree class is expected. This assignment uses the BinaryTree class similarly to STL classes, *i.e.*, the programmer can use binary trees without having to implement them. Our BinaryTree class is useful here since the STL does not have a tree class. This assignment replaces a previous Huffman code assignment which required a special-purpose binary tree application. By providing students with a working binary tree class, the focus of the assignment shifts from tree internals to tree-building and tree-traversal techniques.

5.4 Binary search trees

In this lab assignment, students in "Data Structures" were given the C++ BinaryTree class, and wrote a short client program to create a binary search tree. As in the Huffman code assignment, the BinaryTree class allowed them to focus on the tree-building task and avoid the internal details.

5.5 Decision trees

In this "Data Structures" programming assignment, students were required to design an interactive program that builds up a binary decision tree. The tree becomes larger with every interaction from the user. The program was also required to read trees from data files and write trees to data files. Again, students were provided with the C++ code for the BinaryTree class and wrote a client program.

6 Conclusions and future work

The object-oriented use of list and tree classes with API's that emphasize recursion and discourage iteration allow computer science students to use recursion in natural ways to solve interesting computational problems. The author's C++ implementation of a binary tree class was particularly useful since the standard template library does not have a binary tree.
In the future, a re-design of the author's list and binary tree code using inheritance and design patterns will result in more properly object-oriented implementations and will provide a vehicle for teaching topics in object-oriented design and design patterns.

References

[1] J.C. Adams, *Knowing your roots: object-oriented binary search trees revisited*, SIGCSE Bulletin **28** (1996), no. 4, 36–40.

[2] J. Beidler, Y. Bi, and R. McCloskey, *A recursive list paradigm with java and c++ implementations*, JCSC **17** (2002), no. 6, 197–205.

[3] A.M. Berman and R.C. Duvall, *Thinking about binary trees in an object-oriented world*, 27th SIGCSE Technical Symposium on Computer Science Education, vol. 27, February 1996, pp. 185–189.

[4] S. Bloch, *Teaching linked lists and recursion without conditionals or null*, JCSC **18** (2003), no. 5, 96–108.

[5] M. Felleisen and D.P. Friedman, *A little java, a few patterns*, MIT press, 1998.

[6] W. Ford and W. Topp, *Data structures with c++ using stl*, second ed., Prentice Hall, 2002.

[7] D. Friedman, M. Wand, and C. Haynes, *Essentials of programming languages*, pp. 32–33, MIT Press, 1992.

[8] M. Goodrich and R. Tamassia, *Data structures and algorithms in java*, second ed., John Wiley & Sons, 2001.

[9] D. Knuth, *The art of computer programming: Fundamental algorithms*, second ed., vol. 1, p. 305, Addison Wesley, 1973.

[10] M. Weiss, *Data structures & problem solving using java*, Addison Wesley, 1998.

[11] _____, *Data structures & algorithm analysis in c++*, second ed., Addison Wesley, 1999.