ACHIEVING A UNIFORM INTERFACE FOR BINARY TREE IMPLEMENTATIONS

Chaya Gurwitz
Brooklyn College of the City Universtiy of New York

ABSTRACT

One of the basic principles taught in a data structures course is that an application program should be independent of the implementation of any particular data structure it uses. This policy breaks down when binary trees are introduced, because the interfaces for the various representations of binary trees are not uniform. In particular, implementing a binary tree by using an implicit array generally requires the array itself to be passed as a parameter to any function that manipulates the tree. In this paper we present an approach for defining the implicit array representation of a binary tree. Our definition makes the underlying array transparent to the user. This allows us to describe a uniform interface for a binary tree module that can be used by an application program regardless of the particular implementation of the tree.

INTRODUCTION

In a data structures course, students are introduced to basic data structures, their properties and implementations. One of the fundamental principles introduced in such a course is the separation of the specification of a data type from its implementation. The concept of an *abstract data type (ADT)* allows us to define a data type in terms of a specification of the values of the objects and the operations that can be performed on the objects. In this way a data type is viewed as a mathematical entity that is not tied to any particular implementation. The principle of *information hiding* provides that the specification of an ADT should be *implementation independent* and make no reference to the way elements of that data type are represented internally [1,2,3,4].

In teaching a data structures course, we emphasize these points:

- The implementation of a data structure should be transparent to the user.
- The use of a data structure should not depend on its implementation.
- Code that manipulates a data structure should not be changed if the implementation is changed.

These guidelines are followed through most of the course. For example, code that depends on a stack module, such as an infix-to-postfix conversion function, is independent of the actual implementation of the stack. The same conversion function can be used whether the stack is implemented as an array, or as a list using dynamic memory allocation, or as a list using a pool of available nodes. (In fact, a typical programming assignment involves running three versions of the program - one for each of the stack implementations - with the proviso that the only changes allowed between runs are in the coding of the stack module.)

However, when *binary trees* are introduced, the information-hiding principle breaks down. Typically, we teach three implementations of binary trees: the *dynamic node representation*, in which nodes of the tree are allocated dynamically; the *linked array representation*, in which nodes of the tree are allocated from an available pool of nodes; and the *implicit array implementation*, in which the tree is stored in a contiguous memory block [3].

For the first two implementations, a *tree* is represented by a *pointer* to some memory location at which the root of the tree is stored. Similarly, a subtree is represented by a pointer to the memory location containing the root of the subtree. The difference between the two implementations is whether this pointer is a true memory address (for the dynamic node representation) or an index into a statically allocated array (for the linked array representation). In either case, many trees can make use of the available memory.

A tree implemented using an implicit array is represented by the array itself, with the tacit understanding that the root of the tree is the first location of the array. A subtree is referenced slightly differently, in that an index into the array is needed along with the array itself. If more than one tree is needed, a separate array must be allocated for each tree.

Consider a basic tree function, *Tree_data(p),* that returns the data stored at node *p,* (which can be characterized as the root of a subtree). In a linked representation, the parameter *p* provides the location of the node. However, for a tree implemented using an implicit array, the function must be provided with the location of the underlying array as well as the location of the node. Therefore, either the underlying array must be known globally, or the function is rewritten as *Tree_data(p, tree_array)* (see, for example, [3, p.251] in which the array is used explicitly).

SIGCSE '95 3/95 Nashville, TN USA
© 1995 ACM 0-89791-693-x/95/0003....$3.50

Neither solution is satisfactory in terms of the goals of the data structures course. If the array is global, then the program is limited to using only one tree. If the parameter list is changed, then the calling sequence in the application program varies for the different tree implementations, and the implementation is no longer transparent.

One might argue that the various tree implementations are not equally applicable to all applications, and that therefore, the user *should* indeed be aware of which implementation is being used. In particular, the implicit array implementation is, often, an extremely inefficient implementation. This representation should be used only when the tree is known to be close to a complete binary tree, such as in heapsort. However, we feel that for the sake of uniformity, this representation should be presented in a manner that preserves the principles of abstract data typing and information hiding.

DEFINITION OF A TREE USING THE IMPLICIT ARRAY REPRESENTATION

In the following we present an approach for defining the implicit array representation of a tree in a manner that makes the underlying array transparent to the user. The tree must contain a reference both to the underlying array and to the location of the root node. (Although the root of the tree is understood to be in the first location of the array, a subtree can be rooted at any location within the array.) In addition, we need to know the location of the "last" node in the tree, so that we can determine when a location in the array is beyond the bounds of the tree. This leads us to the following definition of a tree, which can be used both for the root of a tree or for the root of a subtree:

```
struct Tree {
   Type *Tree_array;
   int root;
   int *n; };

typedef struct Tree Tree;
```

Note that the tree contains *references* to both the underlying array, *Tree_array,* and the last index, *n.* In this manner we ensure that there is only one copy of the underlying array, and its associated index, *n.* The storage for *Tree_array* and *n* is allocated when a new tree is created. This scheme allows multiple trees to be created. When a particular subtree (or node), *p,* is accessed, the *Tree_array* field of the record associated with *p* provides access to the appropriate array.

A UNIFORM INTERFACE

The file below describes a typical interface for a tree module. It contains function prototypes for basic operations defined on trees. We then show that this interface can be used regardless of whether the tree is implemented using a linked representation or using the implicit array representation.

```
#include "treedef.h"

/*tree.h: function prototypes for basic
binary tree functions */

/* Tree_new and Tree_free are functions to
construct and destruct trees */

Tree *Tree_new();
void Tree_free(Tree *t);

int Tree_empty(Tree t);

/* Tree_data returns the data stored at the
root of tree t. Tree_left returns the left
subtree of tree t. Tree_right returns the
right subtree of tree t.*/

Type Tree_data(Tree t);
Tree Tree_left(Tree t);
Tree Tree_right(Tree t);

/* Tree_make_root inserts the data, x, at
the root of tree t.Tree_set_data stores the
data, x, in node t. Tree_insert_left
inserts the data, x, at the root of the
left subtree of tree t. Tree_insert_right
inserts the data, x,at the root of the
right subtree of tree t */

void Tree_make_root(Tree *t, Type x);
void Tree_set_data(Tree *t, Type x);
void Tree_insert_left(Tree *t, Type x);
void Tree_insert_right(Tree *t, Type x);
```

Figure 1: file tree.h

The file *treedef.h* contains the definition of the tree, which can be implemented using any one of the three representations described earlier.

IMPLEMENTATIONS USING THE UNIFORM INTERFACE

In the following, we present two different implementations of trees that can be used with the interface given in the file *tree.h.* In each case, we specify the files *treedef.h* and *tree.c.* The file *treedef.h* contains the definition of a tree; the file *tree.c* provides the code to implement the functions whose prototypes appear in *tree.h.* The first implementation, presented in Figures 3 and 5, uses the implicit array representation described earlier. The second implementation, presented in Figures 4 and 6, uses a linked representation in which a *tree* is defined as a pointer to a node. The nodes themselves may be either dynamically or statically allocated. This example provides another instance of information hiding, in that the actual implementation of the nodes is hidden in the files *tnode.h* and *tnode.c.* It is not necessary to know the implementation of the nodes in order to define a tree; it is sufficient to provide access to the functions which are used to allocate and free nodes, and the functions which set and retrieve the values stored in the nodes. Figure 2 illustrates the different combinations of files that can be used in writing a program that manipulates binary trees.

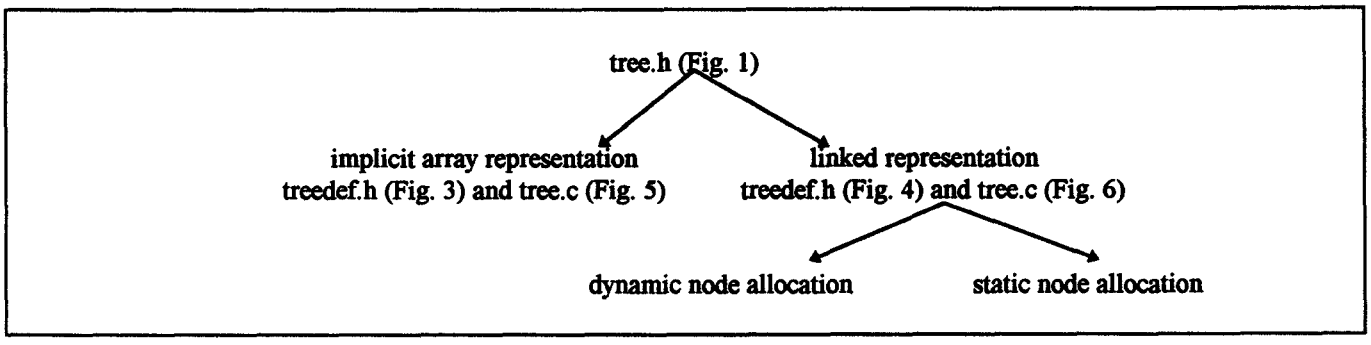

Figure 2: various implementations of binary trees defined with a uniform interface

```
#ifndef TREEDEF_H
#define TREEDEF_H

typedef int Type;
#define MaxTreeNodes 100
#define NULLVAL -99

struct Tree {
   Type *Tree_array;
   int root;
   int *n; };
typedef struct Tree Tree;
#endif
```

Figure 3: treedef.h (implicit array)

```
#ifndef TREEDEF_H
#define TREEDEF_H

#include "tnode.h"
typedef Tnodeptr Tree;

   /*Tnodeptr is a pointer to a
   Tree_node, defined in tnode.h.
   The pointer may be an address
   or an array index, depending
   on whether dynamic or static
   allocation is used */
#endif
```

Figure 4: file treedef.h (linked nodes)

```
#include <stdlib.h>
#include <assert.h>
#include "treedef.h"

/* code for basic tree functions using the
implicit array representation of a tree */

Tree *Tree_new()
{

   Tree *t;
   int i;

   t = (Tree *) malloc(sizeof(Tree));
   t -> Tree_array=(Type *)malloc
       (MaxTreeNodes*sizeof(Type));
   for (i=0; i<MaxTreeNodes; i++)
      (t->Tree_array)[i]= NULLVAL;
   t -> root = 1;
   t -> n = (int *) malloc (sizeof (int));
   *(t -> n) = 0;
   return (t);
}
```

Figure 5: file tree.c (implicit array)

```
#include <stdlib.h>
#include <assert.h>
#include "treedef.h"

/* code for basic tree functions using the
linked node representation of a tree */

Tree *Tree_new()
{
   Tree *t;

   t = (Tree *)malloc(sizeof (Tree) );
   assert(t);
   *t = NULLPTR;
   return (t);
}
```

Figure 6: file tree.c (linked nodes)

Figure 5 continued: tree.c (implicit array)

```
void Tree_free(Tree *t)
{
   free(t -> Tree_array);
   free(t -> n);
   free (t);
}

int Tree_empty(Tree t)
{
   return  (t.Tree_array[t.root]==NULLVAL)
                ||(t.root>*(t.n));
}

Type Tree_data(Tree t)
{
   assert (!Tree_empty(t));
   return (t.Tree_array)[t.root];
}

Tree Tree_left(Tree t)
{
   Tree q;
   q.root = 2*t.root;
   q.n = t.n;
   q.Tree_array = t.Tree_array;
   return (q);
}
Tree Tree_right(Tree t)
{
   Tree q;
   q.root = 2*t.root+1;
   q.n = t.n;
   q.Tree_array = t.Tree_array;
   return (q);
}

void Tree_make_root(Tree *t, Type x)
{
   Tree_set_data(t,x);
}

void Tree_set_data(Tree *t, Type x)
{
   Type *arr;

   arr = t->Tree_array;
   arr[t->root] = x;
   if (t->root > *(t->n))*(t->n) = t->root;
}



void Tree_insert_left(Tree *t, Type x)
{
   Type *arr;
   int left;

   assert(t->root <= *(t->n));
   arr = t->Tree_array;
   left = t->root*2;
   assert(left < MaxTreeNodes);
   arr[left] = x;
   if (left > *(t->n)) *(t->n) = left;
}
```

Figure 6 continued: tree.c (linked nodes)

```
void Tree_free(Tree *t)
{
   free (t);
}



int Tree_empty(Tree t)
{
   return (t == NULLPTR);
}


Type Tree_data(Tree t)
{
   assert(t != NULLPTR);
   return(Tnode_data(t));
}

Tree Tree_left(Tree t)
{
   assert(t != NULLPTR);
   return(Tnode_left(t));
}



Tree Tree_right(Tree t)
{
   assert(t != NULLPTR);
   return(Tnode_right(t));
}




void Tree_make_root(Tree *t, Type x)
{
   Tnodeptr p;

   assert(t);
   p = Tnode_new();
   Tnode_set_data(p,x);
   *t = p;
}

void Tree_set_data(Tree *t, Type x)
{
   assert(t);
   Tnode_set_data(*t,x);
}

void Tree_insert_left(Tree *t, Type x)
{
   Tnodeptr p;

   assert(*t != NULLPTR);
   p = Tnode_new();
   Tnode_set_data(p,x);
   Tnode_set_left(*t,p);

}
```

Figure 5 continued: tree.c (implicit array)

```
void Tree_insert_right(Tree *t, Type x)
{
   Type *arr;
   int right;

   assert(t->root <= *(t->n));
   arr = t->Tree_array;
   right = t->root*2 + 1;
   assert(right < MaxTreeNodes);
   arr[right] = x;
   if (right > *(t->n))*(t->n) = right;
}
```

Figure 6 continued: tree.c (linked nodes)

```
void Tree_insert_right(Tree *t, Type x)
{
   Tnodeptr p;

   assert(*t != NULLPTR);
   p = Tnode_new();
   Tnode_set_data(p,x);
   Tnode_set_right(*t,p);
}
```

OTHER TREE FUNCTIONS

The interface *tree.h*, defined above, provides some basic tree functions. More general tree functions can be defined, using the functions defined in *tree.h* as building blocks. Since we have shown that the interface *tree.h* can be used in a uniform manner, regardless of the actual representation of a tree, such general functions can also be used uniformly by application programs that use different tree implementations.

As an example, consider a procedure that reads in a list of numbers and prints out a list of duplicates. This example is based on an algorithm described in [3, p.238]. In that textbook, the program is presented in two different versions [3, p.247 and p.251], corresponding to different implementations of binary trees. Using the interface *tree.h*, defined in Figure 1, we are able to use the code below, which is totally independent of the actual tree implementation that is used.

```
#include <stdio.h>
#include "tree.h"

void print_dup()
{
   /* print_dup -- prints out duplicates in
an input list.  As the numbers are entered,
they are inserted into a binary search
tree.  If a duplicate entry is found in the
tree, it is printed out. */

   Tree *tree;
   Tree p,q;
   int number;

   tree = Tree_new();
   scanf("%d", &number);

   Tree_make_root(tree,number);

   while ( scanf("%d", &number) > 0 ) {
      p = *tree;
      q = *tree;

      while (number != Tree_data(p) &&
                !Tree_empty(q)) {
         p = q;
         if ( number < Tree_data(p))
            q = Tree_left(p);
         else
            q = Tree_right(p);
      }

      if (number == Tree_data(p) )
         printf ("%d %s\n", number, "is a
            duplicate");
      else if (number < Tree_data(p))
            Tree_insert_left(&p, number);
      else
            Tree_insert_right(&p, number);
   }
}
```

CONCLUSION

Most data structures textbooks discuss various implementations of binary trees. The implicit array representation is generally introduced in a manner that is not consistent with the presentation of the linked representation. This inconsistency detracts from the goals of the data structures course. Our approach for defining the implicit array representation of a binary tree hides the details of the implementation. This allows us to describe a uniform interface for a binary tree module that is implementation independent.

REFERENCES

[1] Horowitz, E., Sahni, S. and Anderson-Freed, S., **Fundamentals of Data Structures in C,** Computer Science Press, New York, 1993.

[2] Kingston, J.H., **Algorithms and Data Structures: Design, Correctness, Analysis,** Addison-Wesley, Reading, Mass., 1990.

[3] Tenenbaum, A.M., Langsam, Y. and Augenstein, M.J., **Data Structures Using C,** Prentice Hall, Englewood Cliffs, N.J., 1990.

[4] Weiss, M.A., **Data Structures and Algorithm Analysis in C,** Benjamin Cummings, Menlo Park, Ca., 1993.