THINKING ABOUT BINARY TREES IN AN OBJECT-ORIENTED WORLD

A. Michael Berman, Rowan College of New Jersey
berman@rowan.edu

Robert C. Duvall, Brown University
rcd@cs.brown.edu

ABSTRACT

The Binary Search Tree serves as an important example when teaching data structures. We explore new approaches to understanding the implementation of a Binary Search Tree, using concepts from Object-Oriented Programming and C++. The Binary Search Tree illustrates how adopting a new approach and a new language can lead to a new way of thinking about a familiar problem.

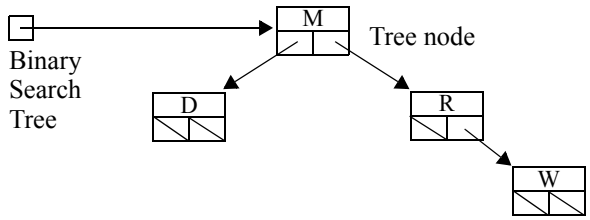
THE GOOD OL' BINARY SEARCH TREE

The Binary Search Tree has been familiar and useful to programmers and Computer Scientists since it was discovered independently by several people in the 1950's [14]. Discussion of the Binary Search Tree can be found in standard references on data structures and algorithms, such as [2] or [9], and in most textbooks intended for the course commonly known as CS2, for example [11] and [13].

Of the many equivalent definitions for the Binary Search Tree, we'll use the following:

Definition 1 A Binary Search Tree (BST) is either a Null BST, or it consists of a root value, *d*, and two children *left* and *right*, each of which is a BST, such that *left* is either Null or has root value less than *d*, and *right* is either Null or has a root value greater than or equal to *d*.

Figure 1 Typical textbook implementation of a BST



The typical textbook implementation of a BST, whether in Pascal or C++, defines a node consisting of a data value and two child pointers, as illustrated in Figure 1. A BST is defined to be a pointer to a node. The Nil pointers in the BST nodes represent the Null BST's.[1]

The traditional style of declaring a BST works, but there are three significant weaknesses to this approach:

- *Conceptual Gap:* Our abstract notion of a tree, contained within Definition 1 , describes a BST as either Null or Non-Null, and doesn't refer to pointers.
- *Special cases:* An empty tree will be represented by a Nil pointer; hence inserting the first item into the tree, or deleting the last, has to be treated as a special case.
- *Checking for Null:* Since a leaf contains Nil pointers, the code will contain many comparisons to Nil. These comparisons are not particularly intuitive, and they make the code less readable.

This paper provides ways of programming the BST that more nearly implement the definition directly into the code, and eliminate the problems with the "traditional" implementation. We believe that by creating code that closely mimics our abstract notions, we can reduce that conceptual gap between the abstract and the concrete, making it easier for our students to understand the implementation.

We begin with an Object-Oriented design for BST's that's conceptually clean, but unfortunately cannot be implemented in C++ (nor in the other most commonly used OOP languages). Then we present another approach that compromises somewhat on our original design, but is still simpler and cleaner than the "traditional" implementation, and is very easy to implement. Next, we show a design with greater conceptual clarity; this method is attractive, but does require some sophistication on the part of the student. Finally, we discuss how developments in programming languages may make it possible for us to implement directly

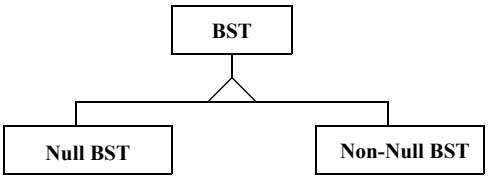
1. We're using the term "Nil Pointer" or just "Nil" for what in C++ is a zero pointer. A zero pointer is often called a "Null Pointer" but we felt that could cause the reader confusion between "Null Pointer" and "Null Tree".

the "better" designs in the future.

AN OBJECT-ORIENTED DESIGN FOR BST'S

We begin our design by identifying the *classes* inherent in Definition 1 . The definition describes a BST as actually consisting of two kinds of BST's, a Null BST and a Non-Null BST. In other words, a Null BST *is a* BST, and a Non-Null BST *is a* BST. Whenever we encounter an "*is-a*" relationship, we should consider modeling this relationship using inheritance [5, p. 93]. This leads us to the inheritance model illustrated in Figure 2.

Figure 2 BST's modeled in an inheritance hierarchy



We can, in fact, implement a static BST in C++ using this design.[1] A class definition for the inheritance hierarchy is shown in Code Example 1. The reader should compare this definition with Definition 1 .

Note that we assume that a class `btClass`, the type of data stored as the value in the Non-Null BST's, has been defined elsewhere. A more flexible implementation would use templates, but we've used a pre-defined class instead to make the code accessible to a wider audience. Template-based implementations of all the examples in the paper can be obtained by contacting the author.

Code Example 1 BST declaration with inheritance

```
// the following definition is needed if
// compiler does not implement a Boolean type
enum bool {false, true};

// BST is an abstract base class for BST's
class BST {
public:
  BST() { } // base class constructor
  virtual ~BST() { } // base class destructor
  virtual bool member(const btClass & d) = 0;
};
// A nullBST inherits its interface from BST class
class nullBST : public BST {
public:
  nullBST() { } // create a null BST
  ~nullBST() { } // class destructor
  bool member(const btClass & d);
};

// nonNullBST inherits interface from BST class,
// and also contains some private data
class nonNullBST : public BST {
public:
  // constructor for nonNullBST has to put the
  // data into the tree, since it's static
  nonNullBST(const btClass & d, BST * leftChild,
             BST * rightChild);
  ~nonNullBST();
  bool member(const btClass & d);
private:
  BST * left;
  BST * right;
  btClass data;
};
```

Given this declaration, it's incredibly simple to write, for example, the `member` function. Consider the algorithm for searching for an item *d* in a BST *b*:

- if *b* is a null BST, return false.
- else, if *d* equals the data at the root of *b*, return true
- else, if *d* is smaller than the root of *b*, search for *d* in *b*'s left subtree
- else, search for *d* in *b*'s right subtree.

Our implementation, shown in Code Example 2, tracks the algorithm exactly, except that it's split into two parts, one for null BST's and one for non-Null BST's. The compiler automatically selects the appropriate function to apply, depending on whether the object is a `nullBST` or a `nonNullBST`.

Code Example 2 *member* function with inheritance

```
bool nullBST::member(const btClass & d)
{
  return false;
}
bool nonNullBST::member(const btClass & d)
{
  if (d == data)
    return true;
  else if (d < data)
    return left->member(d);
  else // d must be > data
    return right->member(d);
}
```

Of course, one might observe that a Binary Search Tree *is-a* Binary Tree. Thus it makes sense to inherit the BST from a Binary Tree, as can be seen in several textbooks, including [7] and [11]. The first author uses this approach in his upcoming textbook [3]; however, to simplify this brief paper we've implemented the BST *de novo*.

EXTENDING THE DESIGN FOR DYNAMIC TREES

Unfortunately, if we want to implement a dynamic BST, our design cannot be implemented directly in C++. To understand the problem, consider the algorithm for an *insert* — to insert data *d* into a non-Null BST, insert into either the left or right subtree (as appropriate); to insert into a Null BST, *make the Null BST into a non-Null BST* containing *d*. C++ doesn't support a way for a member function to change itself from an object of one kind into an object of another kind, even though they are derived from the same base class. In this last section we discuss this operation from the

1. A *static* data structure does not support insertions and deletions.

perspective of other languages; for the moment, let's look at how we can handle it in C++.

One possible approach would be to use a *handle class* or *letter/envelope classes* [10, p. 133]. You wrap a second class (the *handle* or *envelope*) around the class the represents the BST (the *body* or *letter*), thus giving you greater flexibility in the operations you can perform. However, this approach requires the student to have more sophistication in coding than one usually finds at this level. The handle class implementation of a BST may well be useful if you want to teach students about creating handle classes, but we're not convinced it's a good approach to helping them understand the implementation of a BST, our goal here.

AN ALTERNATIVE DEFINITION

One way to handle the inability of an object to change classes is to represent the Null and non-Null BST's explicitly within a single class. This approach has the drawback that it abandons the inheritance hierarchy, and requires a test in each function to determine which case — Null or non-Null — applies. By explicitly representing Null and non-Null BST's, we have brought the implementation closer to the definition; the student can see that when a Null tree receives a request to add data, it actually becomes a non-Null tree.

We can distinguish between the two types of trees by adding a flag to the class that is, say, true when the object represents a Null tree and false for a non-Null tree, or by using an enumerated type. By using a slightly more subtle representation, we can reduce our space requirement by keeping a single pointer in each tree.[1] A Null tree's pointer will be Nil, while a non-Null tree contains a pointer to a structure that contains the three items associated with the root of the tree.[2] Code Example 3 illustrates the declaration of this class.

Code Example 3 BST declaration without inheritance

```
class BST {
  BST();
  ~BST();
  bool member(const btClass & d);
  void insert(const btClass & d);
  void remove(const btClass & d);
  bool isEmpty() { return r == 0; }
  btClass getData() { return data; }
private:
  struct root {
    BST * left;
    BST * right;
    btClass data;
  };
  root * r;
};
```

Our new implementation of `member` essentially pulls the two `member` functions from Code Example 2 into a single function. The identification of a BST as Null or non-Null, performed implicitly by the compiler in our first implementation, now must be performed explicitly by checking the pointer `r`. We also need an additional level of indirection to refer to the fields in a non-Null tree, since they are accessed via the `r`. Otherwise, the code in Code Example 4 is identical to Code Example 2.

Code Example 4 *member* function without inheritance

```
bool BST::member(const btClass & d)
{
  if (r == 0)// this is a Null tree
    return false;
  else { // this is a non-Null tree
    if (d == r->data)
      return true;
    else if (d < r->data)
      return r->left->member();
    else // d must be > r->data
      return r->right->member();
  }
}
```

Our new implementation makes it easy to perform an insert. In order to change a BST from a Null to non-Null BST, we simply create a new struct to store the data fields for the non-Null BST, as illustrated by Code Example 5.

Code Example 5 *insert* function

```
void BST::insert(const btClass & d)
{
  if (r == 0) { // this is a Null tree
    r = new root;
    r->left = new BST; // a new Null tree
    r->right = new BST;
    r->data = d;
  }
  // otherwise, handle non-Null BST
  else if (d < r->data)
    r->left->insert(d);
  else
    r->right->insert(d);
}
```

One the one hand, we claim that students can easily understand and apply this implementation, and that it partially closes the conceptual gap we started with. On the other hand, we recognize that what we've done is apply a procedural approximation of an object-oriented concept — we would much prefer to use polymorphism to distinguish between the Null and non-Null trees, rather than applying a explicit test. In the next section, we will return to an inheritance-based approach, but with some cost as we shall see.

DYNAMIC NODES BY ASSIGNMENT — A "FUNCTIONAL" APPROACH

An alternative developed by instructors at Brown University, including the second author, requires the use of an assignment with each insert.

The general method as implemented in Object Pascal can be found in [8]. In this approach, `insert` becomes a recursive function that returns the newly inserted node. As it recurses

1. More about space requirements below.
2. A similar design, implemented in Pascal, can be found in [7].

down the tree, it stores the identity of the node it is about to visit; when the code gets to the point of an insertion, it creates the new node and returns it as the result of the function.

You can think of a nullBST as a smarter kind of Null pointer. Often (e.g. for the `member` function) it just returns Nil (0, false) to indicate that the operation failed to produce a useful value, but that is semantically different from explicitly checking to determine if the node is empty. The nonNullBST does most of the work, but even here it's greatly simplified by not having to check for Nil and being functional in nature. The functional approach helps by making it easier to update one's children when the structure of the tree changes (since only local changes are possible). The `insert` function appears as Code Example 6.

Code Example 6 Insert with assignment

```
BST * nullBST::insert (const btClass & d)
{
  // Note you only need 1 copy of the nullBST
  return new nonNullBST(d, this, this);
}
BST * nonNullBST::insert (const btClass & d)
{
  if (d < data)
     left = left->insert(d);
  else if (d > data)
     right = right->insert(d);
  return this;
}
```

The strength of the approach is illustrated quite strongly by examining the `remove` function. Compare the relative brevity and clarity of this routine to the approach in, e.g., [6, pp. 324-325]. The function `getSuccessor()` simply finds the successor node in the tree.

Code Example 7 Remove function with assignment

```
BST * nonNullBST::remove(const btClass & d)
{
  if (d < data)
    left = left->remove(d);
  else if (d > data)
    right = right->remove(d);
  else {
    BST * replacement = left->getSuccessor();
    if (0 == replacement) {
      BST * result = right;
      // zero out to prevent children from deletion
      right = left = 0;
      delete this;
      return result;
    }
    data = replacement->getData();
    left = left->remove(data);
  }
  return this;
}
```

Unfortunately, this method is not truly generalizable and has only proven to work well for this particular problem (changing node-types in a data structure dynamically). Teaching students to handle the assignment of the value of the tree properly can be difficult at first, but they become comfortable with it after some practice. Also, there's a certain lack of naturalness in using assignment along with an insertion operation, at least in an object-oriented language. After all, you wouldn't expect to use assignment with, for instance, a push operation on a stack. However, an assignment does seem natural in a functional language, which was the inspiration for this approach. We also note that this assignment might be an implementation detail hidden by a higher-level interface.

SPACE EFFICIENCY

By explicitly representing Null BST's, both implementations use more space than the "traditional" approach in which a Null tree is just an Nil pointer. We suggest the philosophy "get it right first, then get it efficient". The advantages of conceptual consistency justify the lack of space efficiency. Consider, for example, the lack of special cases. If you represent a Null BST as a Nil pointer, then you have to provide special-case code for adding the first item to the tree, and for removing the last item from the tree. Furthermore, explicitly representing the Null BST's means that even though they "do nothing" in the standard tree, it's possible to create a subclass in which the Null BST contains data or has some additional behavior. In the first author's course, he presents the BST with explicit Null trees, and then gives the space-efficient design as an exercise for the students to develop.

OTHER LANGUAGES

The inability of C++ to support the inheritance-based approach discussed above is not a special quirk of C++. For example, Ada95 also lacks the ability to create code in which an object changes its own type [12]. Note that in most OOP languages, including C++, it's easy to change the type of an object that a pointer refers to, by deleting the old object and creating a new one; this is exactly the approach taken by the "functional" approach above.

It also appears possible to implement our design in Eiffel [15]. Eiffel contains a special INTERNAL class that allows an application to examine and modify the internal representation of an object. However, according to Bertrand Meyer, "Class INTERNAL is intended for special applications only, and should be used with great care.[15, p. 350]" Using the INTERNAL class to implement a BST could not be justified based on the grounds of greater clarity and would seem to be of no value as a tool for simplifying the teaching of data structures.

The language *SELF*, on the other hand, does provide a notable exception. It's beyond the scope of this paper to present a tutorial on SELF; the reader is referred to [17]. In SELF, objects are not organized into classes; rather, SELF is prototype-based, which means that objects themselves are templates for creating new objects, as well as being concrete and available to the program. In other words, there is no separation between the class (or type) and the instantiated

object [18]. Something like a class is created by creating two concrete objects: a prototype and a "traits". There is a slot in the prototype (and all its copies) which allows it to delegate messages to the traits object. The essence of this pattern is that the prototype stores any state necessary for each instance of the "class" and the traits object stores the common behavior.

The power of this pattern comes from making these "parent" slots just like any other slots; thus there can be more than one (which gives you multiple inheritance), and they can be assigned to. By assigning to a parent slot, a*n object can change the object it inherits from*; this is called *dynamic inheritance*. Consequently, BST's *can* be implemented in SELF so that an object which at one time inherits the behavior of a Null BST can, via dynamic inheritance, change its parent to a non-Null BST. A partial example of the implementation of BST's in SELF can be found in [1]. While it's important to recognize that the ability to implement one's favorite example can be a dangerous way to choose a language, SELF should certainly be explored as an OOP for teaching purposes. A team at Brown University has begun to look into this area [19].

Kim Bruce has developed binary trees in ML (a functional language) and Object Pascal (OO language) in his data structures course at Williams College [4]. It's instructive to compare and contrast the way in which the problem can be solved in different programming paradigms.

CONCLUSION

The most exciting thing about introducing new paradigms in the classroom is the opportunity to look at old problems in a new light. In our experience, approaching the design of binary trees from an object-oriented perspective provides new insight into the problem, both for the instructor and the student. We believe that such insight helps illustrate the power of OOP and other "alternative" paradigms.

AVAILABILITY OF THE CODE

The code from this paper, as well as additional code that is not shown here, can be obtained via anonymous ftp at cs.rowan.edu:~ftp/berman/bintree.tar(unix) or ~ftp/berman/bintree.zip (dos), or by sending email to the first author (berman@rowan.edu).

ACKNOWLEDGMENTS

Thanks to Rick Mercer for the opportunity to try out some of these ideas in his classroom.

REFERENCES

[1] Agesen, Ole, Jens Palsberg and Michael I. Schwartzbach, "Type Inference of SELF: Analysis of Objects with Dynamic and Multiple Inheritance". Proceedings of the European Conference on Object-Oriented Programming 1993, Springer-Verlag. (Available via [16].)

[2] Aho, Alfred V., John E. Hopcroft and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] Berman, A. Michael. *Data Structures via C++: Objects by Evolution*. To be published by Oxford University Press, 1997.

[4] Bruce, Kim. Private communication.

[5] Budd, Timothy. *An Introduction to Object-Oriented Programming.* Addison-Wesley, 1991.

[6] Budd, Timothy A. *Classic Data Structures in C++*. Addison-Wesley, 1994.

[7] Collins, William J. *Data Structures: An Object-Oriented Approach*. Addison-Wesley, 1992.

[8] Conner, D. Brookshire, David Niguidula and Andries van Dam. *OO Programming in Pascal: A Graphical Approach*. 1995.

[9] Cormen, Thomas H., Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1991.

[10] Coplien, James O. *Advanced C++ — Programming Styles and Idioms.* Addison-Wesley, 1992.

[11] Decker, Rick and Stuart Hirshfield. *Working Classes: Data Structures and Algorithms Using C++*. PWS Kent, 1996.

[12] Feldman, Michael. Personal communication via email.

[13] Headington, Mark R. and David D. Riley. *Data Abstraction and Structures Using C++*. D.C. Heath and Company, 1994.

[14] Knuth, Donald E. *The Art of Computer Programming: Vol. 3, Sorting and Searching.* Addison-Wesley, 1993.

[15] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[16] SELF Home Page. http://self.smli.com.

[17] Ungar, David and Randall B. Smith, "SELF: The Power of Simplicity," *Lisp and Symbolic Computation*, 4(3) 1991. (Available via [16].)

[18] Ungar, David, Chambers, Chang, and Holzle, "Organizing Programs Without Classes", *Lisp and Symbolic Computation* 4(3), 1991.

[19] van Dam, Andries, Brook Conner and Robert Duvall, "CS 196b: Object-Oriented Programming Languages in Computer Science Education," http://www.cs.brown.edu/courses/cs196b.