Knowing Your Roots:
Object-Oriented Binary Search Trees Revisited

Joel C. Adams
Department of Mathematics and Computer Science
Calvin College
Grand Rapids, MI 49546
adams@calvin.edu

Abstract

By applying object-oriented design to the definition of a binary search tree, Berman and Duvall [1] designed a data structure comprised of three classes: (i) an `EmptyBST` class to model empty binary search trees, (ii) a `NonEmptyBST` class to model non-empty binary search trees, and (iii) a `BST` base class for common attributes of `EmptyBST` and `NonEmptyBST` objects. That paper noted the problem of inserting new values into such a structure: since insertions occur at an `EmptyBST` object, an `EmptyBST` would have to "turn into" a `NonEmptyBST`; a behavior beyond the capabilities of the classes in most languages.

This paper presents three C++ solutions to the insertion problem in their order of development. The first solution uses a procedural programming technique, with the second and third solutions shifting to a more object-oriented approach. This chronology illustrates the author's ongoing battle to shift from procedural to object-oriented thinking.

Introduction

The phrase, "You *can't teach an old dog new tricks*" is a staple of American folk wisdom. One "new trick" — object-oriented design (OOD) — promises to produce non-redundant, easier to maintain code, and revolutionize the software industry.

In a simplistic version of the "Booch method" of OOD [2], the objects in a problem are equated with the nouns in the problem description. If two or more such objects have common attributes, an abstract base class is designed to house those attributes. Classes for the original objects are then derived from such base classes.

In a good O-O design, the resulting set of objects are *autonomous*, meaning that each object contains all of the methods it needs. This is sometimes referred to as the "I can do it myself" principle of OOD — objects should perform their own operations, rather than be operated upon.

As Berman and Duvall have noted, when OOD is applied to the binary search tree (BST), the result is very different from the traditional implementation [1]. To illustrate, here is an informal binary search tree definition:

A ***binary search tree*** is either an empty binary search tree or it is a non-empty binary search tree. A non-empty binary search tree consists of a *value* and two binary search tree children *left* and *right*, such that *value* is greater than all values in its *left* child, and is less than all values in its *right* child.

Figure 1 shows the BST of a typical CS2 textbook:

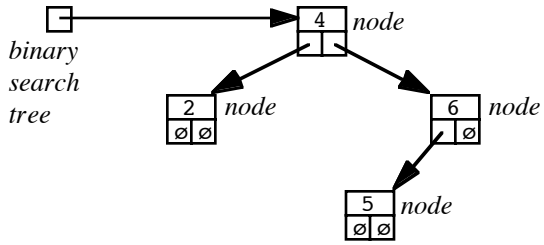

Figure 1: Traditional BST Implementation

consisting of a pointer to a *node*, a structure containing a value and two pointers to nodes. This is inconsistent with the definition, which defines a *binary search tree* recursively, as opposed to defining a recursive *node*, and then defining the binary search tree as a pointer to a node.

By contrast, when OOD is applied to this definition, the objects *empty binary search tree* and *non-empty binary search tree* are identified. To store common attributes, a `BST` base class can be defined, from which `EmptyBST` and `NonEmptyBST` classes can be derived, as shown in Figure 2:

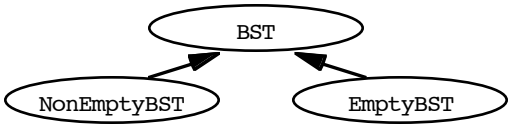

Figure 2: OOD BST Inheritance Hierarchy

Figure 3 shows the C++ syntax to define these classes:

```
class BST {
 public:
  BST() {}
 virtual ~BST() {}
 virtual bool contains(const Value &) const = 0;
 // ... other BST method prototypes
};

class EmptyBST : public BST {
 public:
  EmptyBST() {}
  ~EmptyBST() {}
  bool contains (const Value &) const;
  // ... other EmptyBST method prototypes
};

class NonEmptyBST : public BST {
 public:
  NonEmptyBST(const Value &, BST *, BST *);
  ~NonEmptyBST();
  bool contains(const Value &) const;
  // ... other NonEmptyBST method prototypes
 private:
  Value itsValue;
  BST * itsLeftChild,
      * itsRightChild;
};
```

Figure 3: C++ Inheritance Hierarchy

As Berman and Duvall note, OOD can greatly simply the binary search tree operations, because the functionality is partitioned between two different objects. To illustrate, Figure 4 shows the implementations of the `contains()` methods for the two derived classes:

```
bool EmptyBST::contains(const Value & val) const
{
   return false;
}

bool NonEmptyBST::contains(const Value & val) const
{
   if (val == itsValue)
      return true;
   else if (val < itsValue)
      return itsLeftChild->contains(val);
   else
      return itsRightChild->contains(val);
}
```

Figure 4: `contains()` Methods

For polymorphic methods like `contains()`, the functionality is divided across the `EmptyBST` and `NonEmptyBST` classes. As a result, the special tests for `NULL` pointers that characterize the traditional approach disappear in the object-oriented binary search tree.

The Problem

Unfortunately, the `insert()` and `delete()` operations are somewhat more difficult, as Berman and Duvall note. To illustrate, suppose that a program has executed

```
BST * myBST = new EmptyBST;
```

producing the situation shown in Figure 5:

Figure 5: An Empty BST

Suppose the program then wants to insert some `newItem` into the tree using a polymorphic `insert()` method:

```
myBST->insert(newItem);
```

Because `myBST` is pointing at an `EmptyBST` object, the method `EmptyBST::insert()` is being invoked. To accomplish its task, this method must somehow replace its `EmptyBST` with a new `NonEmptyBST` containing `newItem`, as shown in Figure 6:

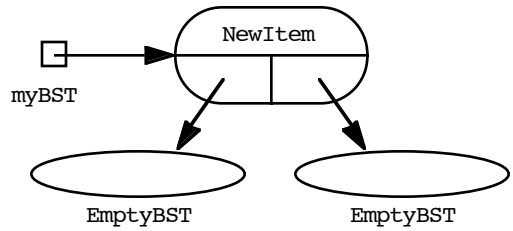

Figure 6: After Inserting `NewItem`

Berman and Duvall note that this could be accomplished if one object were able to transform itself into another object (i.e., an `EmptyBST` into a `NonEmptyBST`). Since C++ classes do not provide this capability, they present two alternative solutions.

One alternative they present is to consolidate the two classes into one, with a boolean flag that when set, treats the class as a `NonEmptyBST`, and when cleared treats the class as an `EmptyBST`. By eliminating the polymorphism, this approach requires its methods to test the flag to distinguish which behavior to perform. It is thus a step away from the O-O approach.

Their second alternative is a technique adapted from Object-Pascal, which maintains the O-O design, but modifies the `insert()` method to return the address of the new `NonEmptyBST`. This address can then be assigned to the pointer making the call, as in:

```
myBST = myBST->insert(NewItem);
```

As Berman and Duvall note, this approach seems unnatural and requires some getting used to.

As it turns out, neither the two alternatives nor the exotic capability of class transformation is needed to solve this problem. In a nutshell, what is needed is this: *the method* `EmptyBST::insert()` *must be able to alter the pointer by which it was called*.

In the remainder of this paper, we present three different approaches that use this observation, each an improvement on its predecessor.

Approach 1: Passing the Pointer.

Our first approach began with this thought: *If* `insert()` *needs to be able to alter the pointer by which it was called,, why not pass that pointer to* `insert()` *via a reference parameter*? Figure 7 shows the resulting pair of methods:

```
void EmptyBST::insert(const Value & val,
                      BST * & viaPtr)
{
 BST * tmpPtr = new EmptyBST;
 viaPtr = new NonEmptyBST(val, viaPtr, tmpPtr);
}

void NonEmptyBST::insert(const Value & val,
                         BST * & viaPtr)
{
  if (val < itsValue)
    itsLeftChild->insert(val, itsLeftChild);
  else if (it > itsItem)
    itsRightChild->insert(val, itsRightChild);
  else
    cerr << val << " is already in the tree!";
}
```

Figure 7: Insert by Passing the Pointer

Applying this idea retains the benefits of polymorphism for the `insert()` method, since calls like:

```
myBST->insert(4, myBST);
myBST->insert(2, myBST);
myBST->insert(6, myBST);
myBST->insert(5, myBST);
```

can be used to build an O-O binary search tree equivalent to that shown in Figure 1. Note that the first insertion is a call to `EmptyBST::insert()`, but the other calls are to `NonEmptyBST::insert()`, which recursively descends the tree to the appropriate `EmptyBST`, where a (polymorphic) call to `EmptyBST::insert()` performs the insertion.

This approach works, but it seems ungainly, especially since the passed pointer is used only in the very first call. Aesthetically, this approach seems only marginally better than the "insert using an assignment" approach.

Worse yet, this approach reflects a *procedural mindset*: When a procedure needs a piece of data, that data is passed to the procedure via a parameter. By contrast, in the O-O approach, such data should be *stored within the object*. Recognition of this led to our second approach.

Approach 2: "I Can Do It Myself"

Dissatisfaction with our first approach led us to re-evaluate our class hierarchy's design. The "I can do it myself" world of OOD suggests that instead of passing the `EmptyBST` methods a parameter containing the pointer by which it is attached to the tree, an `EmptyBST` should store that information. Figure 8 illustrates this idea:

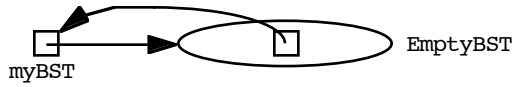

Figure 8: Storing the Pointer Address

We call the pointer by which an `EmptyBST` or a `NonEmptyBST` is attached to the tree its *root pointer*, so in Figure 8, `myBST` is the root pointer of the `EmptyBST`. If the value 4 is inserted, then `myBST` will be the root pointer of the `NonEmptyBST` containing 4, and `myBST->itsLeftChild` and `myBST->itsRightChild` will be the root pointers of two `EmptyBST`s, as shown in Figure 9:

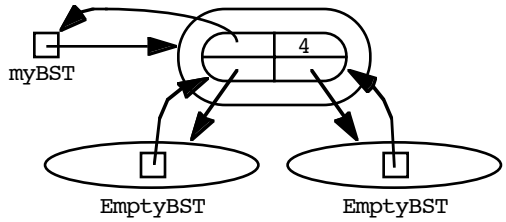

Figure 9: Storing Root Pointers

As it happens, both `EmptyBST` and `NonEmptyBST` methods must access their root pointers. Since methods in both classes require this information, the root pointer should be stored in and inherited from the base class, `BST`.

To provide this capability, the `BST` base class stores the *address* of the root pointer, and then provides methods for using it (two for the data member storing its address, and two for the root pointer itself), as shown in Figure 10:

```
class BST {
 public:
  BST(BST * *)
 virtual ~BST() {}
 virtual bool contains(const Value &) const = 0;
  // ... other BST method prototypes
 protected:
  void SetRootPtrAddress(BST * *);
  BST * GetRootPtrAddress() const;
  void SetRootPtr(BST *);
  BST * GetRootPtr() const;
 private:
  BST * * itsRootPtrAddress;
};
```

Figure 10: The Modified BST Base Class

The definitions of the class constructors must then be modified to initialize the root pointer, as Figure 11 shows:

```
BST::BST(BST * * rootPtrAddress)
{
 itsRootPtrAddress = rootPtrAddress;
}

EmptyBST::EmptyBST(BST * * rootPtrAddress)
 : BST(rootPtrAddress)
{}

// ... NonEmptyBST constructor omitted
```

Figure 11: Constructor Modifications

Given access to the root pointer, the `insert()` methods can be encoded without pointer parameters, as Figure 12 shows:

```
void EmptyBST::insert(const Value & val)
{
  BST * * rPtrAddr = GetRootPtrAddress();
  NonEmptyBST * tmpPtr =
    new NonEmptyBST(val, this, NULL, rPtrAddr);
  SetRootPtr(tmpPtr);
  SetRootPtrAddress(&(tmpPtr->itsLeftChild));
  tmpPtr->itsRightChild =
    new EmptyBST(&(tmpPtr->itsRightChild));
}

void NonEmptyBST::insert(const Value & val)
{
  if (val < itsValue)
    itsLeftChild->insert(val);
  else if (it > itsValue)
    itsRightChild->insert(val);
  else
    cerr << val << " is already in the tree!";
}
```

Figure 12: Insert — I Can Do It Myself

With these methods, a program can now create the empty binary search tree of Figure 8 by executing:

```
myBST = new EmptyBST(&myBST);
```

and then perform a series of insertions quite cleanly:

```
myBST->insert(4);
myBST->insert(2);
myBST->insert(6);
myBST->insert(5);
```

Thus, this approach requires passing the address of the root pointer *once*, when the tree is created. Since methods require this information, the information must be passed at some point, and doing so once, when the object is created, seems like a minimal imposition. Once created, an object "knows" its root pointer, and its methods can make use of this "knowledge".

The primary drawback of this approach is that the user must remember to pass *the address* of the root pointer, which is both error-prone and inconvenient. Thinking about how to eliminate this drawback led to our third approach, which is a simple refinement of approach 2.

Approach 3: Using References

Our final approach resulted after some research into the capabilities of C++ references. In terms of its behavior, a C++ reference variable is essentially a constant pointer variable that is automatically dereferenced. Moreover, when the address-of operator (&) is applied to a reference variable, the result is the address of the object being pointed at [3]. For example, if a function `F()` has a reference parameter `p`:

```
void F(Type & p);
```

and is called with argument `a`:

```
F(a);
```

then within the definition of `F()`, the expression

```
&p
```

yields the address of argument `a`. This allows us to eliminate the drawback of our second approach. Figure 13 presents the constructors using this approach:

```
EmptyBST::EmptyBST(BST * & rootPtr)
 : BST (& rootPtr)
{}

NonEmptyBST::NonEmptyBST(const Value & val,
              BST * left, BST * right,
              BST * & rootPtr)
   : BST(& rootPtr)
{
   itsValue = val;
   itsLeftChild = left;
   itsRightChild = right;
}
```

Figure 13: Constructing with References

None of the other methods need to be modified — we simply add these constructors, after which a statement like

```
myBST = new EmptyBST(myBST);
```

can be used to build the empty binary search tree shown in Figure 8, eliminating the requirement that the caller supply the address-of operator. Following this, the insertions

```
myBST->insert(4);
myBST->insert(2);
```

produce a binary search tree like that in Figure 14:

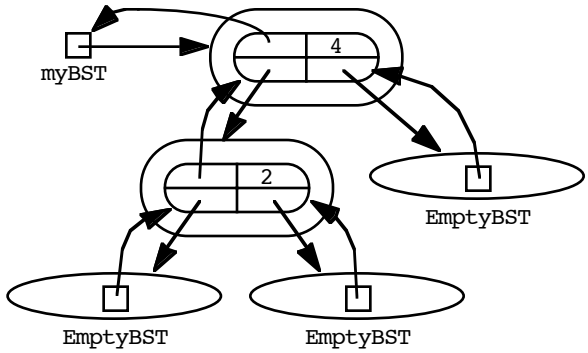

Figure 14: An O-O BST

By passing a reference to the root pointer when the `BST` is created, the resulting implementation retains the "I can do it myself" flavor of our second approach, while eliminating its drawback. The result is a binary search tree implementation that is consistent with the traditional definition, and that contains no redundant code.

Miscellany

To keep this paper short and (relatively) simple, we have ignored the `delete()` operation in our discussion. Because all of the work must be done by `NonEmptyBST::delete()`, this operation is more complicated than `insert()`. However, it is not a difficult method to implement, and doing so is a useful exercise.

We have also ignored the issue of the relative efficiency of the O-O implementation compared to the traditional implementation, since that was discussed in the original (Berman and Duvall) paper.

The complete source code (including the `delete()` methods) for each of the three approaches is available on the World-Wide-Web at

`http://www.calvin.edu/~adams/OO-BST`

CS2 instructors in search of projects may freely download any of the three approaches and delete whatever portions they wish their students to implement. Providing students with a working class and having them extend its functionality has avoids overwhelming students, and it provides experience maintaining code written by others, reinforcing the importance of thorough documentation.

A simple honors project is to provide students with a copy of [3] and the source code to approach 2, and tell them to make whatever modifications are needed in order for approach 3 to work. Rather than having to write lots of code, this project requires students to have an in-depth understanding of pointers and references (in greater depth than is available in the typical textbook), and then use that understanding to write a few lines of code. More importantly, the project provides a simple vehicle to teach students to find information for themselves and learn on their own, a worthy goal for any honors project.

In Closing

We have presented three approaches that solve the problem of inserting a value into an O-O binary search tree. The approaches range from a clumsy initial effort to a final effort that more fully embodies O-O principles.

The chronology of our three approaches provides an interesting insight into the difficulty of making the procedural-to-O-O paradigm shift. After learning software design in the traditional top-down modular approach, virtually all of the author's programming for the past 5 years has been O-O programming in C++. Yet when faced with a new problem, the first solution that occurred to him reflected a procedural approach (thankfully, one whose deficiencies led him to the O-O approach). This leaves the author feeling like "an old dog", and wondering

How long does it take to learn this "new trick"?

References

1. A. Michael Berman and Robert C. Duvall, *Thinking About Binary Trees In An Object-Oriented World*, Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, 27(1): 185-189, February, 1996.

2. Grady Booch, *Object Oriented Design*, Benjamin/ Cummings, 1991.

3. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1992, pp. 540.