Optimal Binary Search Trees Meet Object–Oriented Programming

Stuart Hansen and Lester I. McCann
Computer Science Department
University of Wisconsin — Parkside
Kenosha, WI 53141
{hansen,mccann}@cs.uwp.edu

August 14, 2002

Abstract

This paper presents an object–oriented approach to the problem of finding optimal binary search trees. We show that object–oriented techniques produce a solution that is an improvement over those created procedurally and that is well within the reach of our undergraduate students. Combining the study of optimality with that of object–oriented design helps the students gain a deeper appreciation of both.

1 Introduction

Finding optimal solutions to problems and applying object–oriented design techniques are both standard topics in a data structures and algorithms course. Too frequently they receive separate treatments, while in reality they have profound influences on each other. An object–oriented design typically distributes the responsibility for managing a data structure across its components. This allows the programmer to not only optimize the overall data structure, but to optimize the components as well. Students gain insight into both optimality and object–oriented design by examining the interplay between them. This interaction is beginning to be explored by other researchers as well [3].

This paper takes a dynamic programming problem, that of finding an optimal binary search tree, and shows how the standard solution may be improved by applying object–oriented techniques to the design of its components. These optimizations extend the solution found in most textbooks. The authors have used this example in several sections of data structures and algorithms. We have found the material well within the reach of our students. Because it integrates optimization and object–oriented concepts, we have found that it helps students better understand both.

In Sections 2 and 3, we briefly visit object–oriented binary search trees and optimal binary search trees. Section 4 introduces our ideas for creating optimal object–oriented binary search trees, and the paper's conclusion can be found in Section 5.

2 Review of Object–Oriented Binary Search Trees

We assume the reader is familiar with binary trees and binary search trees. They are covered in depth in many standard texts such as [1, 4, 5, 7, 8]. The purpose of this section is to review an object–oriented implementation of binary search trees.

Definition: A **binary tree** T is a structure defined on a finite set of nodes that either contains no nodes, or is composed of three disjoint sets of nodes: a root node, a binary tree called its left subtree, and a binary tree called its right subtree [5].

Definition: A **binary search tree** (BST) is a binary tree whose nodes are organized according to the binary search tree property: keys in the left subtree are all less than the key at the root; keys in the right subtree are all greater than the key at the root; and both subtrees are themselves BSTs. (Note: We are assuming unique keys.)

A variety of possible implementations of object–oriented BSTs are possible, with variations due to design philosophies and language capabilities [2]. Our implementation uses two node classes, *Node* and *NullNode*.

Both are derived from the abstract class *AbstNode*. The *Node* class represents internal nodes; each *Node* object contains a key value and two subtrees. The *NullNode* class represents "external" or "empty" nodes. A *NullNode* object contains no key value and has no subtrees.

With this approach, there is never a need to check if a reference is null. Dispatching works correctly because left and right subtrees exist for all internal nodes and are never referenced by a *NullNode* object. We cannot completely eliminate comparisons, however. Insertion and search keys must still be compared to the tree's keys.

Our object–oriented approach to searching a BST is to distribute the responsibility to each node. We illustrate the approach by presenting Java code fragments for searching. Other basic tree operations follow similar principles.

Our Java instance method for searching a *Node*: [1]

```
boolean search (Key search_key) {
    if (search_key.lt(key))
        return left.search(search_key);
    else if (search_key.gt(key))
        return right.search(search_key);
    else
        return true;
}
```

If a *NullNode* node is reached, the search has failed. Our Java instance method for searching a *NullNode* is trivial:

```
boolean search (Key searchkey) {
    return false;
}
```

This approach to BSTs gives slightly more efficient code than a procedural approach, because there is no check for null pointers/references. The trade–off is that the program makes calls to *NullNode*s that exist for no reason other than to terminate recursion.

3 Optimal Binary Search Trees Revisited

Optimal Binary Search Trees are covered in many algorithms texts [1, 5, 7, 8]. Our treatment closely parallels that found in [5].

For any set of keys, there are many different binary search trees. The time required to seek a given key can vary from tree to tree depending on the depth of the node where the key is found, or the length of the branch searched if the key is not present.

3.1 Basic Definitions

When we know the probability of the search key equaling each key and the probability of the search key falling between each pair of adjacent keys, we may calculate the *average search time* (AST) for the tree.

Let $K = < key_1, key_2, \ldots, key_n >$ be a sequence of keys in sorted order. The keys are associated with the internal nodes of a BST.

Let $D = < d_0, d_1, \ldots, d_n >$ be a sequence of ranges representing the intervals between two keys. d_0 represents all possible keys less than key_1. d_k represents all possible keys between key_k and key_{k+1}. d_n represents all possible keys greater than key_n.

Conceptually, the ranges are associated with the external nodes that terminate each branch of a BST. The *NullNode* class presented in Section 2 does not model them. In Section 4 we implement this idea in the form of a *RangeNode* class.

Let p_i be the probability of the search key equaling key_i, and let q_k be the probability of the search key falling within d_k's interval. Of course, $\sum p_i + \sum q_k = 1$. Let $AST(i, j)$ be the mean value of the number of nodes visited during a search of a tree containing keys i through j. Then:

[1] Interested readers may find Java implementations of the classes used in this paper at `www.cs.uwp.edu/staff/hansen`.

$$AST(i,j) = \sum_{k=i}^{j} c_k p_k + \sum_{l=i-1}^{j} b_l q_l \qquad \text{for } 1 \le i \le j \le n$$

where $c_k = 1 + depth(key_k)$ and $b_l = 1 + depth(d_l)$. $AST(1,n)$ is the average search time for the entire tree.

The average search times vary significantly among the BSTs containing the set of keys, K. An optimal BST is one that gives the minimum AST for K.

3.2 An Example

An example will clarify these definitions and concepts. Consider the following sequences of keys and probabilities:

```
K  =      <   2    ,     5    ,     7   ,   8  >
p  =      < 0.15   ,   0.10   ,   0.10 , 0.05 >
q  = < 0.10   ,   0.05   ,   0.25 ,   0.15   , 0.05 >
```

The spacing of the p and q probabilities reflects their significance. For example, the probability of searching for 2 is 0.15, and the probability of searching for a value between 5 and 7 is 0.25.

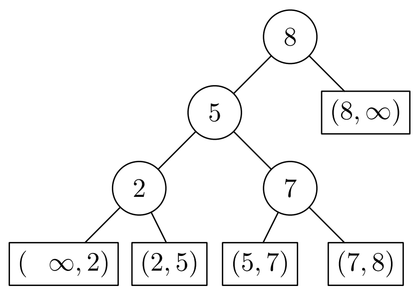

Figure 1: *One possible BST with* $AST = 3.30$

The AST of the tree in Figure 1 is calculated as shown in the following table:

$$\begin{array}{lclcr}
c_1 p_1 & = & 3 \cdot 0.15 & = & 0.45 \\
c_2 p_2 & = & 2 \cdot 0.10 & = & 0.20 \\
c_3 p_3 & = & 3 \cdot 0.10 & = & 0.30 \\
c_4 p_4 & = & 1 \cdot 0.05 & = & 0.05 \\
b_0 q_0 & = & 4 \cdot 0.10 & = & 0.40 \\
b_1 q_1 & = & 4 \cdot 0.05 & = & 0.20 \\
b_2 q_2 & = & 4 \cdot 0.25 & = & 1.00 \\
b_3 q_3 & = & 4 \cdot 0.15 & = & 0.60 \\
b_4 q_4 & = & 2 \cdot 0.05 & = & \underline{0.10} \\
 & & & & 3.30
\end{array}$$

The AST of the tree in Figure 2 is calculated in a similar fashion. The tree in Figure 2 is optimal for the given p and q probabilities.

3.3 Finding the Optimal BST

To find an optimal BST, we first note that each subtree must also be optimal. If it weren't, we could find a better overall BST by replacing the subtree with an optimal one. This insight leads us to the recurrence for the AST of the optimal BST. $AST(i,j)$ for a subtree α is the sum of the ASTs of α's subtrees plus $w(i,j)$, which is the cost contributed by searching the root of α as we perform searches that terminate at nodes within α.

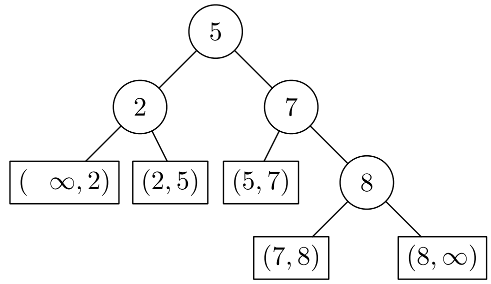

Figure 2: *A second possible BST with $AST = 2.75$*

$$w(i,j) = \sum_{m=i}^{j} p_m + \sum_{r=i-1}^{j} q_r$$

$$AST(i,j) = \begin{cases} \min_{i \le k \le j}\big(AST(i,k-1) + AST(k+1,j)\big) + w(i,j) & \text{for } 1 \le i \le j \le n \\ q_{i-1} & \text{for } j = i-1 \end{cases}$$

Note that we use $j = i - 1$ for the empty tree that lies between $node_{i-1}$ and $node_i$.

Solving this recurrence gives the AST and provides the structure of the optimal BST. The recurrence contains many overlapping subproblems, making it an ideal candidate for evaluation by dynamic programming techniques.

4 Further Optimizing the BST

It is important for students to explore the assumptions made when developing the solution in the previous section. There were two assumptions that should be reviewed:

1. We assumed that the optimal tree will be created from the two types of nodes presented in Section 2, namely *Node* and *NullNode*.
2. We also assumed that the only important metric for measuring optimality is AST.

In this section we challenge each of those assumptions. We show that modifying the assumptions may improve the optimal BST further.

4.1 An Expanded Node Hierarchy

In Section 2 an "external" or "empty" node represented an empty tree. In Section 3 we associated a range with each external node. The external nodes in Figures 1 and 2 are labeled with the interval of values they represent. The intervals are determined by the sequence of keys, not the tree. Both of the sample trees show the same intervals, just arranged differently.

The binary search tree property applies to both the keys in internal nodes and the intervals in external nodes. Every value in the interval of an external node satisfies the property.

Modeling the external nodes with intervals rather than *NullNode*s brings our implementation model closer to our conceptual model. There are four types of intervals: `(left_key, right_key)`, `(`$-\infty$`, right_key)`, `(left_key, `∞`)` and `(`$-\infty$`, `∞`)`. A separate class is developed for each because each uses a different search technique. In the discussion that follows, we use only the first of these for illustrative purposes; the concepts extend to the others.

A *RangeNode* represents the interval of possible values falling between two keys. As such, it represents the first of our four interval types. A *RangeNode* must contain the two bounding key values, `left_key` and `right_key`, in order to test whether or not a search key lies within its interval. Because the binary search tree property applies to range nodes, such nodes need not be relegated to the leaves. Range nodes and ordinary nodes may each appear high in the tree or at the leaves. Thus, *RangeNode*s also must contain `left` and `right` subtrees. Either variety of node may be the root of a subtree.

The advantage of being able to place range nodes anywhere in the tree is that doing so may improve the AST. If there is a high probability of the search key matching an interval, raising the interval's *RangeNode* high in the tree can significantly improve the AST.

A *RangeNode* delegates its search to its left subtree or right subtree as appropriate. It reports failure if the search key lies between `left_key` and `right_key`. The instance method for searching a *RangeNode*:

```
boolean search (Key search_key) {
    if (search_key.le(left_key))
        return left.search(search_key);
    else if (search_key.ge(right_key))
        return right.search (search_key);
    else
        return false;
}
```

A path from root to leaf passes through both ordinary nodes and range nodes. As before, a search must terminate somewhere along a path. Before the end of a branch is reached, the search key is found, or a range node is encountered whose interval contains the search key.

No changes are needed for internal nodes. Polymorphic dispatching allows internal nodes to reference other internal nodes and range nodes, as needed.

The optimal BST recurrence may be adapted to handle range nodes. In this adaptation, *Nodes* and *RangeNodes* are treated uniformly.

Let $R =< r_1 \ldots r_{2n+1} > = < q_0, p_1, q_1, p_2, \ldots p_n, q_n >$ The recurrence is now:

$$AST(i,j) = \begin{cases} \min_{i \le k \le j} \big(AST(i,k-1) + AST(k+1,j) \big) + \sum_{m=i}^{j} r_m & \text{for } 1 \le i \le j \le 2n+1 \\ 0 & \text{otherwise} \end{cases}$$

Because there are $2n+1$ elements in R, the entire tree's AST is found by solving $AST(1, 2n+1)$.

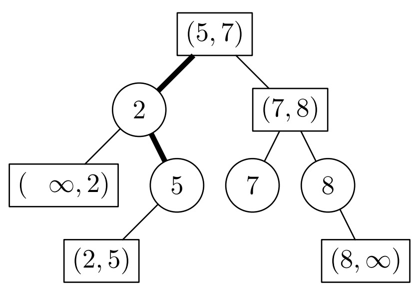

Figure 3: *The optimal BST when range nodes may be distributed throughout the tree.* $AST = 2.30$

Figure 3 shows the improved optimal BST generated for the data provided in Section 3. Comparing this tree with that of Figure 2, we notice that some of the range nodes have risen high in the tree. Computing AST for this tree shows that the average search time has been reduced to 2.30 from 2.75. This is a significant improvement even in this very small tree.

The search path for 5 is highlighted in Figure 3. 5 is less than the range in the root, and we move to its left subtree. 5 is greater than 2, and we move right to discover the match with 5. Now consider a search for 6. This search fails at the root because 6 falls within the range of the root node, indicating that it is not stored within the tree.

4.2 Reducing the Number of Key Comparisons

At this point of our classroom presentation, our better students are complaining that AST is an unfair metric. Their objection is legitimate. Recall that AST is defined as the mean of the number of nodes visited during a search. We have been able to reduce AST only by adding $2n$ more `if` statements to the tree. If, instead of AST, we count the total number of key comparisons executed, different results are obtained. Let:

$$AST'(i,j) = \sum_{k=i}^{j} c_k^{'} p_k + \sum_{l=i-1}^{j} b_l^{'} q_l \qquad \text{for } 1 \leq i \leq j \leq n$$

where $c_k^{'}$ is the number of key comparisons executed when searching for key_k, and $b_l^{'}$ is the number of key comparisons executed when searching in the range (key_{l-1}, key_l). When applied to the trees from Figures 2 and 3, $AST' = 3.50$ and 3.80, respectively. This metric suggests that the original tree is better choice than the modified tree. This is not surprising. When using AST', searching a *NullNode* is free, because it contains no key comparisons. Searching a *RangeNode* is not free. Following the left branch of a *RangeNode* incurs a cost of one key comparison, while following the right branch or having the search terminate at the node requires two key comparisons.

We can reduce AST' in either BST by applying a search strategy [6]. Different search strategies are used at different nodes throughout the tree. The strategies are based on the number of children the node has, and, when there are two children, evaluating the condition most likely to be true before considering the others.

If a *RangeNode* is at a leaf, a search of the node can report failure without checking any conditions. Any search that reaches this node will already have reduced the possible values of the search key to those within the *RangeNode*'s interval. None of these values appear in the tree. In this case the search method returns `false`. In fact, when a *RangeNode* appears at a leaf, we use a *NullNode* in its place.

Similarly, if a *Node* is at a leaf, any search of the node always returns `true`. A search that reaches this node will have already reduced the possible values of the search key to one; namely, the key stored in the *Node*. We can simplify the search method to return the constant `true`. We use a *LeafNode* class for this situation.

Looking again at Figure 3, we see that some nodes have only one child. We develop four more classes for these cases. A *Node* may have only a left child or only a right child. Similarly, a *RangeNode* may have only a left child or only a right child. These classes each contain a single `if` in their search methods and only one child reference.

When a node has two children, the search method will require two conditions. The key to optimizing these nodes is to check the most probable condition first. Only when that condition fails do we check the less likely condition.

For *Nodes*, there are three search strategies (conditions that could be checked first):

(1) `searchKey` $<$ `key`
(2) `searchKey` $==$ `key`
(3) `searchKey` $>$ `key`

The appropriate strategy for each node is chosen when the node is first added to the tree. In the dynamic programming solution the trees are built from the bottom up. Left and right subtrees already exist and are joined by adding the root node. The sum of the probabilities for the left and right subtrees are easy to calculate. Choosing a search strategy for $node_i$ involves finding the maximum of these three values: p_i, the total probability for $node_i$'s left subtree, and the total probability for $node_i$'s right subtree.

For *RangeNodes* there are only two strategies, as testing for key equality is not an option because there is no key:

(1) `searchKey` $<$ `leftKey`
(2) `searchKey` $>$ `rightKey`

Applying the search strategy to the tree in Figure 3 reduces AST' from 3.80 to 3.10. As stated at the beginning of this section, AST' for the tree in Figure 2 was 3.50. The improved tree using our search strategy performs fewer comparisons and visits fewer nodes.

4.3 The Trade–offs

Both AST and AST' affect the execution time of a sequence of searches. Students should understand how each of the optimization techniques discussed influences AST and AST'.

Using range nodes improves AST while usually increasing AST'. Range nodes give the biggest improvement when there are high probabilities of the search key falling between two keys.

Using the search strategy will improve AST' while not changing AST. The search strategy always improves the efficiency of searching leaves and nodes with only one child. For nodes with two children it gives the largest improvement when individual keys have high probabilities or when there is a heavy right branch. In both cases, the simple search incurs a cost of two comparisons, while the improved search strategy only incurs a cost of one comparison.

The two approaches may be combined. The resulting tree will generally show a marked improvement in both AST and AST'.

An interesting student exercise asks them to compare execution times of BSTs that use the different optimization techniques. Carefully designed data sets lead students to a deeper appreciation of both optimization approaches.

5 Conclusion

In this paper we have presented new perspectives on merging optimal binary search trees with object–oriented programming. This merger draws material from dynamic programming and object–oriented design, and is approachable by undergraduate students. We have found that presenting this problem in a data structures and algorithms course helps students integrate ideas from a variety of topics.

We are continuing to explore optimal BSTs, mining the problem for further ways in which object–oriented design may be used. Java source code for optimal BSTs as presented in this paper may be downloaded from: `www.cs.uwp.edu/staff/hansen`.

References

[1] Baase, S., and Gelder, A. *Computer Algorithms: Introduction to Design and Analysis.* Addison–Wesley, 2000.

[2] Berman, A. M., and Duvall, R. C. Thinking about binary trees in an object–oriented world. *SIGCSE Bulletin 28* (1996), 185–189.

[3] Bucci, P., Heym, W., Long, T., and Weide, B. Algorithms and object–oriented programming: Bridging the gap. *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education* (2002), 302–306.

[4] Budd, T. *Classic Data Structures in Java.* Addison–Wesley, 2001.

[5] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, 2nd ed.* MIT Press, 2001.

[6] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object–Oriented Software.* Addison–Wesley, 1995.

[7] Goodrich, M., and Tamassia, R. *Data Structures and Algorithms in Java.* Wiley, 1998.

[8] Sedgewick, R. *Algorithms, 2nd ed.* Addison–Wesley, 1988.