On Finding the Height of a Binary Search Tree

M. A. Weiss

School of Computer Science, Florida International University, University Park, Miami, FL 33199, USA

The preorder, inorder and postorder traversals of binary trees are standard topics in Data Structures courses. A common example (or test question) which uses postorder traversal is finding the height of a randomly formed binary search tree (to verify experimentally that it is indeed \( O(\log n) \)). The natural postorder implementation gives a linear algorithm, but frequently students manage, sometimes unintentionally, to add a third recursive call to the computation, resulting in an apparently drastic increase in running time. It is obvious that the worst case running time increases from linear to exponential, but the results on the average case running time are not as immediate. We show that the average running time increases from linear to cubic.

Received February 1991

1. INTRODUCTION

We consider the problem of finding the height of a binary search tree formed by repeated random insertions. The C code in Figure 1 shows the standard postorder traversal used to solve this problem.

This function arises in many Data Structures courses as an example (or test question) involving a postorder procedure which is not easily implemented without recursion. McCracken [1] has suggested a programming assignment which requires students to empirically determine the average search time in a randomly formed binary search tree. Computing the height and other tree properties, such as the number of leaves, is a natural extension of this project.

It is well-known that the running time for the standard height computation is linear. A surprisingly common error for beginning C programmers, which is magnified by the C preprocessor, is to use a macro definition for the max function.

```c
#define max(x, y) ((x) > (y) ? (x) : (y))
```

Because the macro definition merely expands its arguments, the effect is to create the function in Figure 2 which has three recursive calls per function.

Although this error is more likely to occur by unsuspecting users of the C preprocessor, it is also common for students programming in other languages, such as Pascal and Ada, to not write a `max` function, but merely repeat the recursive call.

```c
height(tree T)
{
    if(T == NULL)
        return(-1);
    else
        return(1 + max(height(T->left), height(T->right)));
}
```

**FIGURE 1.** Standard height computation for a binary tree.

height(tree T)
{
    if(T == NULL)
        return(-1);
    else
        if(height(T->left) > height(T->right))
            return(1 + height(T->left));
        else
            return(1 + height(T->right));
}

**FIGURE 2.** Inefficient computation of height.

The worst case running time of this program is clearly exponential. This occurs for a degenerate binary search tree in which every node has only a left child. We show that the average running time is \( O(n^3) \). Empirical evidence suggests that our bound is tight to within a factor of less than two, although we cannot confirm this analytically.

2. UPPER BOUND

For the rest of this paper, we let \( T(n) \) be the average running time of the program in Figure 2. Our complexity measure will be the number of nodes touched, not counting external (NULL) nodes. The program in Figure 3 is the same as the one in Figure 2 except that an extra recursive call has been added. The effect is that in all cases, two recursive calls are made to each subtree.

Clearly the running time of the program in Figure 3 is an over-estimate of the running time of the program in Figure 2.

The analysis we perform is similar to the analysis of quicksort [2] and the computation of the average time for a binary search tree operation [3]. Let \( T_i(n) \) be the average running time of the program in Figure 3. \( T_i(0) = 0 \). If the number of nodes in the left subtree is \( i \), and the number in the right subtree is \( n - i - 1 \), then because
height(tree T)
{
    if(T == NULL)
        return(-1);
    else
        if(height(T->left) > height(T->right))
        {
            height(T->right); /* wasted call */
            return(1 + height(T->left));
        }
        else
        {
            height(T->left); /* wasted call */
            return(1 + height(T->right));
        }
}

FIGURE 3. Truly inefficient computation of height.

TABLE 1. Observed node visits versus over-estimate

<table>
<thead>
<tr>
<th>n</th>
<th>Observed</th>
<th>((n^3 + 6n^2 + 11n)/18)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>444.01</td>
<td>590.00</td>
</tr>
<tr>
<td>40</td>
<td>2832.73</td>
<td>4113.33</td>
</tr>
<tr>
<td>60</td>
<td>8619.25</td>
<td>13236.67</td>
</tr>
<tr>
<td>80</td>
<td>19455.20</td>
<td>30626.67</td>
</tr>
<tr>
<td>100</td>
<td>36563.77</td>
<td>58950.00</td>
</tr>
<tr>
<td>120</td>
<td>60557.14</td>
<td>100873.33</td>
</tr>
<tr>
<td>140</td>
<td>97710.49</td>
<td>159063.33</td>
</tr>
<tr>
<td>160</td>
<td>134656.44</td>
<td>236176.67</td>
</tr>
<tr>
<td>180</td>
<td>200185.68</td>
<td>334910.00</td>
</tr>
<tr>
<td>200</td>
<td>263195.87</td>
<td>457900.00</td>
</tr>
</tbody>
</table>

the tree is formed by random insertions, \(i\) is equally likely to be any value between 0 and \(n-1\). Thus, \(T_i(n)\) satisfies

\[
T_i(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T_i(i) + T_i(n-i-1)) + T_i(n-i-1)
\]

Since \(\sum_{i=0}^{n-1} T_i(i) = \sum_{i=0}^{n-1} T_i(n-i-1)\), we obtain

\[
T_i(n) = 1 + \frac{4}{n} \sum_{i=0}^{n-1} T_i(i)
\]

Multiplying through by \(n\), we obtain

\[
nT_i(n) = n + 4 \sum_{i=0}^{n-1} T_i(i)
\]

Applying this equation for \(n-1\),

\[
(n-1)T_i(n-1) = n - 1 + 4 \sum_{i=0}^{n-2} T_i(i)
\]

Subtracting and rearranging terms, we obtain

\[
nT_i(n) = 1 + (n+3)T_i(n-1)
\]

This equation can be solved by dividing through by \(n(n+1)(n+2)(n+3)\) to obtain

\[
\frac{T_i(n)}{n(n+1)(n+2)(n+3)} = \frac{1}{n(n+1)(n+2)(n+3)} + \frac{T_i(n-1)}{n(n+1)(n+2)}
\]

From this equation, it is readily seen that

\[
T_i(n) = (n+1)(n+2)(n+3) \sum_{i=0}^{n-1} \frac{1}{i(i+1)(i+2)(i+3)}
\]

A partial fraction decomposition of the sum yields

\[
\sum_{i=0}^{n-1} \frac{1}{i(i+1)(i+2)(i+3)} = \sum_{i=0}^{n-1} \left( \frac{1}{6i} - \frac{1}{6(i+1)} + \frac{1}{2(i+2)} - \frac{1}{2(i+1)} \right)
\]

It is then straightforward to solve for \(T_i(n)\), obtaining

\[
T_i(n) = \frac{n^3 + 6n^2 + 11n}{18}
\]

Since \(T_i(n) > T(n)\), it follows that the average running time of the function in Figure 2 is \(O(n^3)\).

We have thus far been unable to analytically establish that our bound is tight. Doing so would require solving the following recurrence:

\[
T(n) = 1 + \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i-1) + p_{i,n} T(i)) + (1 - p_{i,n}) T(n-i-1)
\]

where \(p_{i,n}\) is the probability that an \(i\) node subtree of an \(n\) node tree is deeper than the other subtree. If we set \(p_{i,n} = 0.5\) for all cases, then the equation underestimates the true value of \(T(n)\). It is easy to show that in this case the solution is quadratic, which implies that \(T(n) = \Omega(n^2)\). On the other hand, the empirical evidence strongly suggests that the running time is \(\Theta(n^3)\) and that our bound is within a factor of less than two. Some of the results are summarized in Table 1. Each observed entry is based on an average of 10,000 runs.

REFERENCES