**BINARY SEARCH TREE PERFORMANCE**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Best Time</th>
<th>Average Time</th>
<th>Worst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find</td>
<td>O(lg n)</td>
<td>??</td>
<td>O(n)</td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td>O(lg n)</td>
<td></td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td>??</td>
<td></td>
</tr>
</tbody>
</table>

**Fastest Running Time**

The find, insert and delete algorithms start at the tree root and a follow path down to, at worst case, the leaf at the very lowest level. The total number of steps of these algorithms is, therefore, the largest level of the tree, which is called the *depth* of the tree.

The best (fastest) running time occurs when the binary search tree is *full* – in which case the search processes used by the algorithms perform like a binary search.

Let's verify this. A *full binary tree* is one in which nodes completely fill every level. For example, here are the unique full binary trees of which have tree depths of 0, 1 and 2:

As you can verify by looking at the examples, a full tree has 1 node at level 0, 2 nodes at level 1, 4 nodes at level 2 and so on. Thus, it will have $2^d$ nodes at level $d$. Adding these quantities, the total number of nodes $n$ for a full binary tree with depth $d$ is:

$$n = 2^0 + 2^1 + 2^2 + \ldots + 2^d = 2^{d+1} - 1$$

For example, the full binary tree of depth 2 above has $2^3 - 1 = 7$ nodes. The binary tree below is a full tree of depth 3 and has $2^4 - 1 = 15$ nodes.
Now, considering the formula above for the number of nodes in a full binary search tree:

\[ n = 2^{d+1} - 1 \]

Solving for \( d \), we get:

\[
\begin{align*}
    n + 1 &= 2^{d+1} \\
    \log(n + 1) &= \log(2^{d+1}) \\
    \log(n + 1) &= (d + 1) \log(2) \\
    \log(n + 1) &= d + 1 \\
    d &= \lfloor \log(n) \rfloor - 1
\end{align*}
\]

For example, the depth of a full binary search tree with 15 nodes is 3.

In other words, the depth of a binary search tree with \( n \) nodes can be no less than \( \log(n) \) and so the running time of the find, insert and delete algorithms can be no less than \( \log(n) \).

A full binary search tree is said to be balanced because every node's proper descendants are divided evenly between its left and right subtrees. Thus, the search algorithm employed by the find, insert and delete operations perform like a binary search.

**Slowest Running Time**

As a binary search tree becomes more and more unbalanced, the performance of the find, insert and delete algorithms degrades until reaching the worst case of \( O(n) \), where \( n \) is the number of nodes in the tree. For example, the number of comparisons needed to find the node marked \( A \) in the binary search trees below is 3, which is equal to the number of nodes.
Binary search trees, such as those above, in which the nodes are in order so that all links are to right children (or all are to left children), are called *skewed trees*.

**Average Running Time**
The average running time of the binary search tree operations is difficult to establish because it is not clear that all binary search trees are equally likely.\(^1\) It has been proven that when a binary search tree is constructed through a random sequence of insertions then the average depth of any node is \(O(\lg n)\). For example, Figure 4.26\(^2\) shows a 500-node randomly generated tree whose average node depth (calculated over all nodes) is 9.98.

![Figure 4.26](image)

A randomly generated binary search tree

The problem arises when considering the effect of deletions since the deletion algorithm replaces a deleted node with one from its right subtree, thus favoring left imbalance. For example, Figure 4.27\(^3\) shows the result of applying 250,000 random insert/remove operations to the tree of Figure 4.26. The tree still has 500 nodes but now has an average node depth of 12.51.

---


\(^2\) Ibid, page 117.

\(^3\) Ibid, page 117.
Extreme left imbalance after deletion, however, does not always happen nor does it happen with all trees, so its effect on the average execution times is not yet understood. In short, the average execution time of the binary search tree operations appears (but cannot be proven) to be $O(\log n)$.\textsuperscript{4}

\textsuperscript{4} For a list of references on this topic, see Ibid, page 152.
**Balanced versus Unbalanced Trees**

The time complexities of operations find, insert and delete on a binary search tree is:

- At best $O(lg\ n)$, which occur when the tree is full
- At worst $O(n)$ which occur when the tree is skewed
- Thought to be on average $O(lg\ n)$

A balanced binary search tree is close to being full, although not necessarily completely full. It has, for each node, about the same number of nodes in its left subtree as in its right subtree. Thus, the find, insert and delete operations on a balanced tree give close to $O(lg\ n)$ performance. The more unbalanced the tree becomes, the longer the search time grows until, at worst, it is $O(n)$, occurring when the tree is skewed.

If you can live with running times that average $O(lg\ n)$ but may degenerate to $O(n)$ in the worst case, then the binary search tree is sufficient for your purposes. If, however, you must have $O(lg\ n)$ as your worst-case running time, then you must use a balanced binary search tree.

**Balancing Schemes**

On pages 458 and 459, your author mentions other schemes for keeping a binary search tree balanced. Here is a summary:

- The first to be invented was the **AVL tree**, named for Adelson-Velskii and Landis who invented it in 1962. The workings of the AVL scheme are visualized in David Galles' tool. For an explanation of this scheme, see Weiss.\(^5\)
- The scheme your book covers in Chapter 9 is known as **red-black trees**. Its workings are visualized in your author's RB Tree Workshop applet.
- Your author covers another scheme in Chapter 10 called **2-3-4 trees**.
- A scheme called **B-trees** is used for searching large external files (i.e. files stored on disk). These are also covered in Chapter 10.
- Other schemes include AA trees and 2-3 trees.

An alternative to balancing is called **splay trees**, which guarantees that any $m$ consecutive tree operations has a worst-case time complexity of $O(m\ lg\ n)$ without guaranteeing that each single operation has $O(lg\ n)$ time complexity. Weiss covers this scheme.\(^6\)

---


\(^6\) Ibid, pages 118-138.