length. As is shown later in the chapter, it is sometimes convenient to use a sentinel to replace the `null` node.)

DEFINITION: The *external path length* of a binary search tree is the sum of the depths of the $N + 1$ `null` references. The terminating `null` node is considered a node for these purposes.

One plus the result of dividing the average external path length by $N + 1$ yields the average cost of an unsuccessful search or insertion. As with the binary search algorithm, the average cost of an unsuccessful search is only slightly more than the cost of a successful search. This follows from Theorem 18.2.

Theorem 18.2

For any tree T, let $IPL(T)$ be the internal path length of T and let $EPL(T)$ be its external path length. Then, if T has N nodes, $EPL(T) = IPL(T) + 2N$.

Proof

This theorem is proved by induction and is left as Exercise 18.8.

Random `remove` operations do not preserve the randomness of a tree. The effects are not completely understood theoretically, but it appears that they are negligible in practice.

It is tempting to say immediately that these results imply that the average running time of all operations is $O(\log N)$. This is true in practice, but it has not been established analytically because the assumption used to prove the previous results do not take into account the deletion algorithm. In fact, close examination suggests that we might be in trouble with our deletion algorithm because the `remove` always replaces a two-child deleted node with a node from the right subtree. This would seem to have the effect of eventually unbalancing the tree and tending to make it left-heavy. It has been shown that if we build a random binary search tree and then perform roughly N^2 pairs of random `insert`/`remove` combinations, then the binary search trees will have an expected depth of $O(\sqrt{N})$. However, it has never been shown that a reasonable number of random `insert` and `remove` operations (in which the order of `insert` and `remove` is also random) unbalances the tree in any observable way. In fact, for small search trees, the `remove` algorithm seems to balance the tree. Consequently, it is reasonable to assume that for random input, all operations will behave in logarithmic average time, although this has not been proved mathematically. Exercise 18.26 describes some alternative deletion strategies.

The most important problem is not the potential imbalance caused by the `remove` algorithm. Rather, it is the fact that if the input sequence is sorted, then the worst-case tree occurs. When this happens, we are in deep trouble: We have linear time per operation (for a series of N operations) rather than logarithmic cost per operation. This is analogous to passing items to quicksort but having an insertion sort executed instead. The resulting running time is completely unacceptable.