Data Structures & Their Algorithms

Harry R. Lewis
Harvard University

Larry Denenberg
Harvard University

HarperCollins*Publishers*

procedure *BinaryTreeDelete*(**key K, locative P**):
{**K** is the key value of the item to be deleted}
{P is a locative that points to the foot of the tree}
    while $P \neq$ **A and** Key(P) $\neq K$ **do**
        if $K <$ Key(P) **then** $P \leftarrow$ LC(P) **else** $P \leftarrow$ RC(P)
    if $P =$ **A then return** {Key K is not in the tree}
    if RC(P) = **A then** $P \Leftarrow$ LC(P)
    else if LC(P) = **A then** $P \Leftarrow$ RC(P)
    else {Locative P points to the node to be deleted}
        {Find the inorder successor. Q is a locative}
        Q $\leftarrow$ RC(P)
        while LC(Q) $\neq$ **A do** Q $\leftarrow$ LC(Q)
        {Replace the node P to be deleted by its inorder successor Q}

$$\begin{pmatrix} P \\ Q \\ \text{LC}(Q) \\ \text{RC}(Q) \end{pmatrix} \Leftarrow \begin{pmatrix} Q \\ \text{RC}(Q) \\ \text{LC}(P) \\ \text{RC}(P) \end{pmatrix}$$

Algorithm 6.9 Deletion of an item from a binary search tree.

6.5 STATIC BINARY SEARCH TREES

The assumption that *LookUps* are uniformly distributed across keys is likely to be inaccurate in many applications, so it is worth considering strategies that lessen the search time to find the more frequently accessed keys. On page 177 we considered such a strategy for organizing a list implementation of a dictionary, and concluded that the best possible ordering keeps the keys in order by frequency of access. The analogous line of thought in the case of binary trees suggests that **more frequently accessed keys ought to be kept closer to the root.** This is a plausible principle; it is the essential idea behind Huffman codes (§5.4). However, this idea cannot be put into effect naïvely, since the inorder traversal of the nodes must be maintained (unlike in the case of Huffman coding). Conflicting objectives can come into play, since the dictionary ordering of the keys can be at odds with their frequency ordering.

Consider, for example, the keys **A, B,** and C, and assume that their frequencies are **0.35,** 0.3, and **0.35,** respectively. There are five possible binary search trees on these three nodes (Figure 6.5). The symmetric tree (Figure 6.5(c)), which would clearly minimize the expected search time if all keys were equiprobable, has the low-frequency key at the foot. On the other hand, if a higher-frequency key is moved to the root then the height of the tree is increased. As it turns out in this particular case, the symmetric tree is the best; the expected number of comparisons is

$$2 \cdot 0.35 + 1 \cdot 0.3 + 2 \cdot 0.35 = 1.7.$$

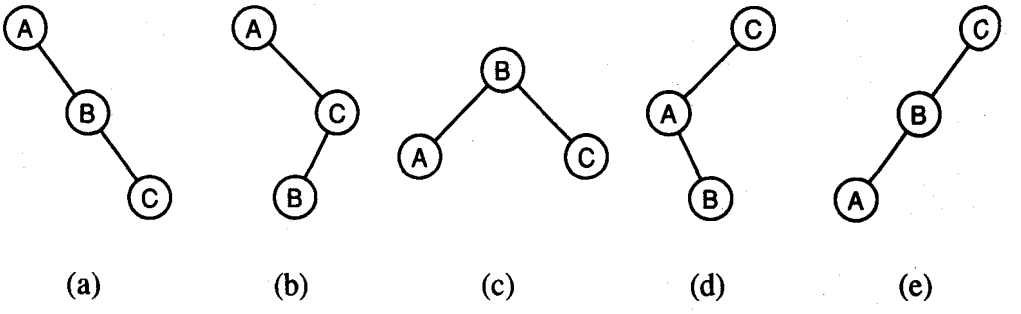

Figure 6.5 The five binary search trees on three keys.

But with other probability distributions the advantage of having a high-frequency key at the root outweighs the disadvantage of increasing the depth of some nodes of the tree. For example, if **A, B,** and C have frequencies **0.45,** 0.1, and 0.45, respectively, then Figure 6.5(b) and (d) are superior to Figure 6.5(c).

Optimal **Trees**

An **optimal binary search tree** is one that minimizes the expected search time. How can we find an optimal tree, given the access frequency of each key? (There can be more than one optimal tree; for example, in the example just given, the trees of Figure 6.5(b) and (d) are both optimal.) A brute-force approach that checks each of the possible binary search trees is impractical, because there are far too many trees to check. (The number of binary trees on n nodes turns out to be $\binom{2n}{n}/(n+1)$, where $\binom{m}{n} = m!/n!(m-n)!$. The number of binary trees is therefore in $\Theta(4^n n^{-3/2})$.) A little planning cuts down the work considerably, however.

Let the keys be $K_1 < K_2 < \cdots < K_n$ in their dictionary order, and let p_i be the probability of accessing K_i. Thus $\sum_{i=1}^{n} p_i = 1$. (We omit consideration of unsuccessful searches; these techniques can be extended to optimize the tree when the external nodes also have known probabilities, that is, when the probability of searching for a key between K_i and K_{i+1} is known for each i.)

Now let $1 \le j \le k \le n$, and let T be any tree constructed from the keys $K_j, \ldots, K_k$. As on page 148, we define $Depth_T(K_i)$, where $j \le i \le k$, to be the depth in T of the node where K_i is stored, and define the **cost** of T to be

$$C(T) = \sum_{i=j}^{k} p_i(Depth_T(K_i) + 1).$$

If $j = 1$ and $k = n$ then the cost is the expected number of comparisons to find a key in the tree; if T holds only a subset of the keys then $C(T)$ represents the cost of searching within the tree for only those keys, with searches for other keys regarded as free. We extend our previous terminology by saying that any tree T is **optimal** if its cost is as small as the cost of any other tree with the same keys.

The expression for the cost of a tree is very similar to that on page 148 for the cost of a Huffman tree. There are two significant differences. First, in the present case all nodes contribute to the sum since all nodes represent keys (in a Huffman tree only the leaves represent character codes). Second, the frequency is multiplied by the depth plus 1, not the depth itself, since even testing the root requires one comparison (in weighing Huffman trees path length from the root measures the number of bits, while here the number of nodes encountered measures the number of comparisons performed).

Thus our objective is to find that tree T on all n keys that minimizes $C(T)$. The crucial observation in reducing the number of trees to be considered is that ***every subtree of an optimal tree is itself optimal.*** That is, if T is an optimal tree for Kj, ..., K_k and its root is K_l, then its left subtree must be an optimal tree for K_j, ..., K_{l-1}, and its right subtree must be an optimal tree for K_{l+1}, ..., K_k. For if the left and right subtrees of T are T_L and T_R, then the depth of each node of T_L or T_R increases by one when it is viewed as a node of T; for example,

$$Depth_T(K_i) = 1 + Depth_{T_L}(K_i)$$

for any i such that K_i is in T_L. So it follows that if K_l is at the root of T then

$$\begin{aligned} C(T) &= p_l + \sum_{i=j}^{l-1} p_i(Depth_T(K_i) + 1) + \sum_{i=l+1}^{k} p_i(Depth_T(K_i) + 1) \\ &= p_l + \sum_{i=j}^{l-1} p_i + C(T_L) + \sum_{i=l+1}^{k} p_i + C(T_R) \\ &= \sum_{i=j}^{k} p_i + C(T_L) + C(T_R). \end{aligned} \tag{2}$$

Therefore replacing T_L or T_R by any tree on the same nodes with lower cost would result in a tree of lower cost than T.

If $d \geq 0$ and we know an optimal tree for each set of nodes $K_{j'}$, ..., $K_{k'}$, where $k' - j' < d$, then for any $j \leq n - d$ we can find an optimal tree for K_j, ..., K_{j+d} by evaluating the cost (2) for each l such that $j \leq l \leq j + d$ and choosing the trees T_L and T_R to be optimal for the keys K_j, ..., K_{l-1} and K_{l+1}, ..., K_k, respectively. This approach suggests a recursive procedure for finding optimal subtrees, but implementing this approach directly would lead to a great deal of repeated computation. Instead the computation can be organized as a dynamic programming algorithm, so that each optimal subtree is determined only once.

Let $T(j, k)$ denote an optimal subtree for the keys K_j, ..., K_k, where $k \geq j - 1$. There are $\Theta(n^2)$ of these subtrees $T(j, k)$ in all, and they can be found by induction on $k - j$. When $k - j = -1$, the tree $T(j, j - 1)$ contains

```
procedure OptimalBinarySearchTree(p_1, ..., p_n):
{Construct optimal search tree}
{Here p(j,k) = p_j + ... + p_k}
    for i from 1 to n do
        r[i,i] ← i
        C[i,i-1] ← 0
    for d from 0 to n - 1 do
        for j from 1 to n - d do
            k ← j + d
            r[j,k] ← MinIndex(C,j,k)
            C[j,k] ← p(j,k) + C[j,r[j,k] - 1] + C[r[j,k] + 1,k]
```

Algorithm 6.10 Computation of optimal binary search tree on $K_1, \ldots, K_n$. The input to the algorithm is the sequence of probabilities $p_1, \ldots, p_n$ with $0 \leq p_i \leq 1$ for each i and $\sum_{i=1}^{n} p_i = 1$; the arrays r and C are filled in by the algorithm as explained in the text. The function *MinIndex*(C, j, k) returns an index l such that $j \leq l \leq k$ and $C[j, l-1] + C[l+1, k]$ is minimized; the order in which calls on *MinIndex* occur in this algorithm ensures that the necessary entries of C have already been calculated when they are needed.

no keys and therefore must be A; and when $k - j = 0$, the tree $T(j, j)$ consists of the single node with key K_j. For $k - j > 0$, $T(j,k)$ is, for some l such that $j \leq l \leq k$, a tree with K_l at the root, $T(j, l-1)$ as the left subtree, and $T(l+1, k)$ as the right subtree; and these subtrees have been determined already, since $(l-1) - j < k - j$ and $k - (l+1) < k - j$. When $k - j = n - 1$ there is only one tree to be determined, namely, $T(1, n)$, *the* optimal tree for the entire set of keys.

To be specific, Algorithm 6.10 computes $C[j,k] = C(T(j,k))$, the cost of any optimal tree for $K_j, \ldots, K_k$, and $r[j,k]$, the root of $T(j,k)$. (There may be several choices for $r[j,k]$ since there may be several for $T(j,k)$; it does not matter which one the algorithm selects.) The tree can be recovered from the $r[j,k]$, since its root is $r[1,n]$, the root of its left subtree is $r[1, r[1,n] - 1]$, etc. In the algorithm, we let $p(j,k) = \sum_{i=j}^{k} p_i$; these values can be computed iteratively in the algorithm's doubly nested loop (Problem 42).

Algorithm 6.10 has much better performance than the brute-force method, but it is still not fast enough to be useful for large n. If function *MinIndex*(j,k), which finds an index l between j and k minimizing $C[j, l-1] + C[l+1, k]$, is implemented simply by searching through all the possibilities $j, j+1, \ldots, k$, then the total number of different triples j, k, l that are considered in the

minimizing step is

$$\sum_{d=0}^{n-1}\sum_{j=1}^{n-d}(d+1) = \sum_{d=0}^{n-1}(n-d)(d+1)$$

$$= n\cdot 1 + (n-1)\cdot 2 + \cdots + 2\cdot(n-1) + 1\cdot n.$$

This sum can be rewritten as

$$\begin{array}{ccccccc} n & + & (n-1) & + & \cdots & + & 1 \\ & + & (n-1) & + & \cdots & + & 1 \\ & & & + & \cdots & + & 1 \\ & & & & & & \vdots \\ & & & & & + & 1 \end{array}$$

$$= \sum_{j=1}^{n}\sum_{k=1}^{j} k = \sum_{j=1}^{n}\frac{j(j+1)}{2},$$

which involves the sum of the first n squares and is therefore in $\Theta(n^3)$ as follows from the Sum of Successive k^{th} Powers Theorem (page 24). Actually, the algorithm can be sped up by a factor of n quite easily; it can be shown (Problem 43) that an l that minimizes the cost falls in the range $r[j, k-1] \le r[j,k] \le r[j+1,k]$, and that restricting the search for $r[j,k]$ to this range reduces the running time to $\Theta(n^2)$.

Probability-Balanced Trees

For larger numbers of nodes, two alternatives can be suggested for the construction of static binary search trees. The first is a **balancing heuristic:** in the notation used earlier, it directs that the key at the root K_l be chosen so that

$$p(1, l-1) \approx p(l+1, n),$$

and that the roots of successive subtrees be chosen to equalize the probabilities of access to their subtrees in the same way. Let us call a tree constructed in this way **probability-balanced.** Probability-balanced trees are a natural generalization of optimal search trees for a uniform distribution; there the tree is constructed so as to have approximately equal numbers of keys in each subtree, and here, with the access probabilities known, we equalize instead the access probabilities to the subtrees. This heuristic works well in practice, and typically yields trees whose expected search times are within a few percent of the optimum; it yields even better trees if the keys one or two away from the one that balances the access probabilities are also tried as possible roots. (This refinement attempts to take advantage of placing a high-frequency key at the root, in case the next or previous key is a low-frequency key that happens to be the one around which the probabilities balance.)