6.2. SEARCHING BY COMPARISON OF KEYS

IN THIS SECTION we shall discuss search methods that are based on a linear ordering of the keys, such as alphabetic order or numeric order. After comparing the given argument K to a key K_i in the table, the search continues in three different ways, depending on whether $K < K_i$, $K = K_i$, or $K > K_i$. The sequential search methods of Section 6.1 were essentially limited to a two-way decision ($K = K_i$ versus $K \neq K_i$), but if we free ourselves from the restriction of sequential access we are able to make effective use of an order relation.

6.2.1. Searching an Ordered Table

What would you do if someone handed you a large telephone directory and told you to find the name of the person whose number is 795-6841? There is no better way to tackle this problem than to use the sequential methods of Section 6.1. (Well, you might try to dial the number and talk to the person who answers; or you might know how to obtain a special directory that is sorted by number instead of by name.) The point is that it is much easier to find an entry by the party's name, instead of by number, although the telephone directory contains all the information necessary in both cases. When a large file must be searched, sequential scanning is almost out of the question, but an ordering relation simplifies the job enormously.

With so many sorting methods at our disposal (Chapter 5), we will have little difficulty rearranging a file into order so that it may be searched conveniently. Of course, if we need to search the table only once, a sequential search would be faster than to do a complete sort of the file; but if we need to make repeated searches in the same file, we are better off having it in order. Therefore in this section we shall concentrate on methods that are appropriate for searching a table whose keys satisfy

$$K_1 < K_2 < \cdots < K_N,$$

assuming that we can easily access the key in any given position. After comparing K to K_i in such a table, we have either

	• $K < K_i$	[$R_i, R_{i+1}, \ldots, R_N$ are eliminated from consideration];
or	• $K = K_i$	[the search is done];
or	• $K > K_i$	[$R_1, R_2, \ldots, R_i$ are eliminated from consideration].

In each of these three cases, substantial progress has been made, unless i is near one of the ends of the table; this is why the ordering leads to an efficient algorithm.

Binary search. Perhaps the first such method that suggests itself is to start by comparing K to the middle key in the table; the result of this probe tells which half of the table should be searched next, and the same procedure can be used again, comparing K to the middle key of the selected half, etc. After at most about $\lg N$ comparisons, we will have found the key or we will have established

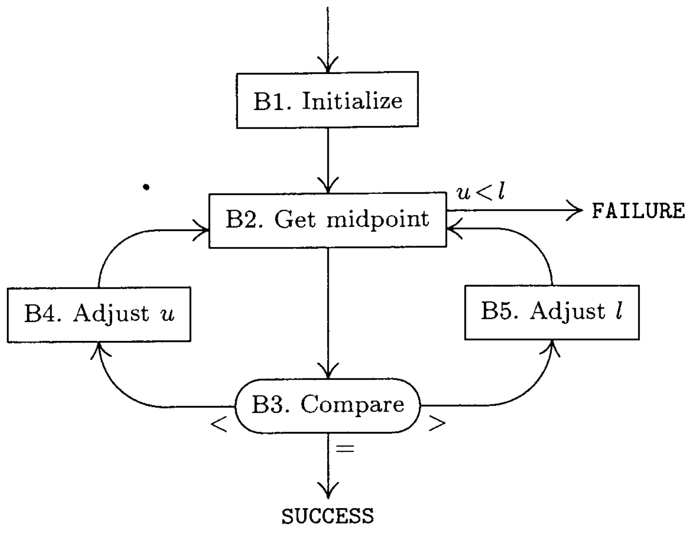


Fig. 3. Binary search.

that it is not present. This procedure is sometimes known as "logarithmic search" or "bisection," but it is most commonly called *binary search.*

Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky, and many good programmers have done it wrong the first few times they tried. One of the most popular correct forms of the algorithm makes use of two pointers, l and u, that indicate the current lower and upper limits for the search, as follows:

Algorithm B (*Binary search*). Given a table of records $R_1 R_2 \ldots R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K.

B1. [Initialize.] Set $l \leftarrow 1$, $u \leftarrow N$.

B2. [Get midpoint.] (At this point we know that if K is in the table, it satisfies $K_l \le K \le K_u$. A more precise statement of the situation appears in exercise 1 below.) If $u < l$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow \lfloor (l+u)/2 \rfloor$, the approximate midpoint of the relevant table area.

B3. [Compare.] If $K < K_i$, go to B4; if $K > K_i$, go to B5; and if $K = K_i$, the algorithm terminates successfully.

B4. [Adjust u.] Set $u \leftarrow i - 1$ and return to B2.

B5. [Adjust l.] Set $l \leftarrow i + 1$ and return to B2. ▌

Figure 4 illustrates two cases of this binary search algorithm: first to search for the argument 653, which is present in the table, and then to search for 400, which is absent. The brackets indicate l and u, and the underlined key represents K_i. In both examples the search terminates after making four comparisons.

a) Searching for 653:

[061 087 154 170 275 426 503 $\underline{509}$ 512 612 653 677 703 765 897 908]
061 087 154 170 275 426 503 509 [512 612 653 $\underline{677}$ 703 765 897 908]
061 087 154 170 275 426 503 509 [512 $\underline{612}$ 653] 677 703 765 897 908
061 087 154 170 275 426 503 509 512 612 [$\underline{653}$] 677 703 765 897 908

b) Searching for 400:

[061 087 154 170 275 426 503 $\underline{509}$ 512 612 653 677 703 765 897 908]
[061 087 154 $\underline{170}$ 275 426 503] 509 512 612 653 677 703 765 897 908
061 087 154 170 [275 $\underline{426}$ 503] 509 512 612 653 677 703 765 897 908
061 087 154 170 [$\underline{275}$] 426 503 509 512 612 653 677 703 765 897 908
061 087 154 170 275] [426 503 509 512 612 653 677 703 765 897 908

Fig. 4. Examples of binary search.

Program B (*Binary search*). As in the programs of Section 6.1, we assume here that K_i is a full-word key appearing in location KEY $+\,i$. The following code uses rI1 $\equiv l$, rI2 $\equiv u$, rI3 $\equiv i$.

```
01 START ENT1 1          1        B1. Initialize. l ← 1.
02       ENT2 N          1        u ← N.
03       JMP  2F         1        To B2.
04 5H    JE   SUCCESS    C1       Jump if K = K_i.
05       ENT1 1,3        C1 − S   B5. Adjust l. l ← i + 1.
06 2H    ENTA 0,1        C + 1 − S  B2. Get midpoint.
07       INCA 0,2        C + 1 − S  rA ← l + u.
08       SRB  1          C + 1 − S  rA ← ⌊rA/2⌋. (rX changes too.)
09       STA  TEMP       C + 1 − S
10       CMP1 TEMP       C + 1 − S
11       JG   FAILURE    C + 1 − S  Jump if u < l.
12       LD3  TEMP       C        i ← midpoint.
13 3H    LDA  K          C        B3. Compare.
14       CMPA KEY,3      C
15       JGE  5B         C        Jump if K ≥ K_i.
16       ENT2 -1,3       C2       B4. Adjust u. u ← i − 1.
17       JMP  2B         C2       To B2. ▌
```

This procedure doesn't blend with MIX quite as smoothly as the other algorithms we have seen, because MIX does not allow much arithmetic in index registers. The running time is $(18C - 10S + 12)u$, where $C = C1 + C2$ is the number of comparisons made (the number of times step B3 is performed), and $S = [\text{outcome is successful}]$. The operation on line 08 of this program is "shift right binary 1," which is legitimate only on binary versions of MIX; for general byte size, this instruction should be replaced by "MUL =1//2+1=", increasing the running time to $(26C - 18S + 20)u$.

A tree representation. In order to really understand what is happening in Algorithm B, our best bet is to think of the procedure as a binary decision tree, as shown in Fig. 5 for the case $N = 16$.

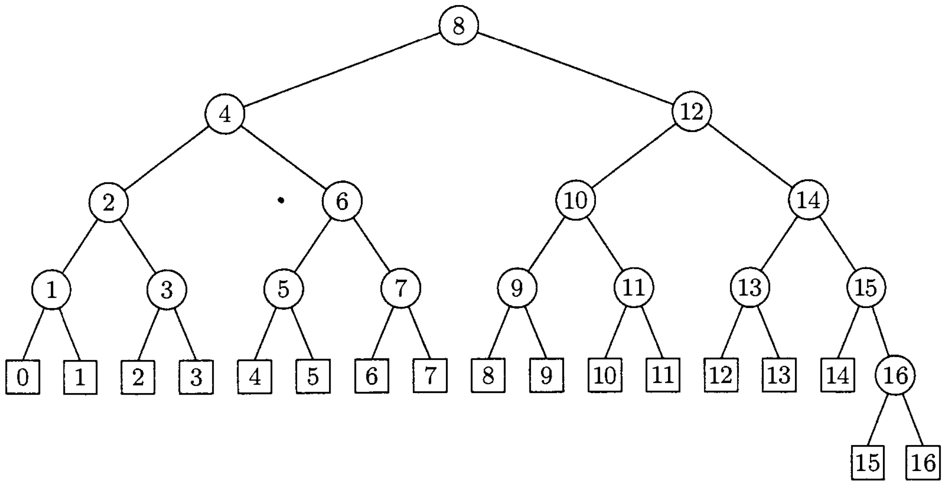


Fig. 5. A comparison tree that corresponds to binary search when $N = 16$.

When N is 16, the first comparison made by the algorithm is $K : K_8$; this is represented by the root node ⑧ in the figure. Then if $K < K_8$, the algorithm follows the left subtree, comparing K to K_4; similarly if $K > K_8$, the right subtree is used. An unsuccessful search will lead to one of the external square nodes numbered $\boxed{0}$ through $\boxed{N}$; for example, we reach node $\boxed{6}$ if and only if $K_6 < K < K_7$.

The binary tree corresponding to a binary search on N records can be constructed as follows: If $N = 0$, the tree is simply $\boxed{0}$. Otherwise the root node is

$$(\lceil N/2 \rceil),$$

the left subtree is the corresponding binary tree with $\lceil N/2 \rceil - 1$ nodes, and the right subtree is the corresponding binary tree with $\lfloor N/2 \rfloor$ nodes and with all node numbers increased by $\lceil N/2 \rceil$.

In an analogous fashion, *any* algorithm for searching an ordered table of length N by means of comparisons can be represented as an N-node binary tree in which the nodes are labeled with the numbers 1 to N (unless the algorithm makes redundant comparisons). Conversely, any binary tree corresponds to a valid method for searching an ordered table; we simply label the nodes

$$\boxed{0} \quad ① \quad \boxed{1} \quad ② \quad \boxed{2} \quad \cdots \quad \boxed{N{-}1} \quad (N) \quad \boxed{N} \qquad (1)$$

in symmetric order, from left to right.

If the search argument input to Algorithm B is K_{10}, the algorithm makes the comparisons $K > K_8$, $K < K_{12}$, $K = K_{10}$. This corresponds to the path from the root to ⑩ in Fig. 5. Similarly, the behavior of Algorithm B on other keys corresponds to the other paths leading from the root of the tree. The method of constructing the binary trees corresponding to Algorithm B therefore makes it easy to prove the following result by induction on N:

Theorem B. *If* $2^{k-1} \le N < 2^k$, *a successful search using Algorithm B requires* (min 1, max k) *comparisons. If* $N = 2^k - 1$, *an unsuccessful search requires*

k comparisons; and if $2^{k-1} \le N < 2^k - 1$, an unsuccessful search requires either $k-1$ or k comparisons. ▌

Further analysis of binary search. (Nonmathematical readers should skip to Eq. (4).) The tree representation shows us also how to compute the *average* number of comparisons in a simple way. Let C_N be the average number of comparisons in a successful search, assuming that each of the N keys is an equally likely argument; and let C'_N be the average number of comparisons in an *un*successful search, assuming that each of the $N+1$ intervals between and outside the extreme values of the keys is equally likely. Then we have

$$C_N = 1 + \frac{\text{internal path length of tree}}{N}, \qquad C'_N = \frac{\text{external path length of tree}}{N+1},$$

by the definition of internal and external path length. We saw in Eq. 2.3.4.5–(3) that the external path length is always $2N$ more than the internal path length. Hence there is a rather unexpected relationship between C_N and C'_N:

$$C_N = \left(1 + \frac{1}{N}\right)C'_N - 1. \tag{2}$$

This formula, which is due to T. N. Hibbard [*JACM* **9** (1962), 16–17], holds for all search methods that correspond to binary trees; in other words, it holds for all methods that are based on nonredundant comparisons. The variance of successful-search comparisons can also be expressed in terms of the corresponding variance for unsuccessful searches (see exercise 25).

From the formulas above we can see that the "best" way to search by comparisons is one whose tree has minimum external path length, over all binary trees with N internal nodes. Fortunately it can be proved that *Algorithm B is optimum* in this sense, for all N; for we have seen (exercise 5.3.1–20) that a binary tree has minimum path length if and only if its external nodes all occur on at most two adjacent levels. It follows that the external path length of the tree corresponding to Algorithm B is

$$(N+1)\big(\lfloor \lg N \rfloor + 2\big) - 2^{\lfloor \lg N \rfloor + 1}. \tag{3}$$

(See Eq. 5.3.1–(34).) From this formula and (2) we can compute the exact average number of comparisons, assuming that all search arguments are equally probable.

$N =$ 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$C_N =$ 1	$1\frac{1}{2}$	$1\frac{2}{3}$	2	$2\frac{1}{5}$	$2\frac{2}{6}$	$2\frac{3}{7}$	$2\frac{5}{8}$	$2\frac{7}{9}$	$2\frac{9}{10}$	3	$3\frac{1}{12}$	$3\frac{2}{13}$	$3\frac{3}{14}$	$3\frac{4}{15}$	$3\frac{6}{16}$
$C'_N =$ 1	$1\frac{2}{3}$	2	$2\frac{2}{5}$	$2\frac{4}{6}$	$2\frac{6}{7}$	3	$3\frac{2}{9}$	$3\frac{4}{10}$	$3\frac{6}{11}$	$3\frac{8}{12}$	$3\frac{10}{13}$	$3\frac{12}{14}$	$3\frac{14}{15}$	4	$4\frac{2}{17}$

In general, if $k = \lfloor \lg N \rfloor$, we have

$$\begin{aligned} C_N &= k + 1 - (2^{k+1} - k - 2)/N &&= \lg N - 1 + \epsilon + (k+2)/N, \\ C'_N &= k + 2 - 2^{k+1}/(N+1) &&= \lg(N+1) + \epsilon' \end{aligned} \tag{4}$$

where $0 \le \epsilon, \epsilon' < 0.0861$; see Eq. 5.3.1–(35).

To summarize: Algorithm B never makes more than $\lfloor \lg N \rfloor + 1$ comparisons, and it makes about $\lg N - 1$ comparisons in an average successful search. No search method based on comparisons can do better than this. The average running time of Program B is approximately

$$\begin{array}{ll} (18\lg N - 16)u & \text{for a successful search,} \\ (18\lg N + 12)u & \text{for an unsuccessful search,} \end{array} \qquad (5)$$

if we assume that all outcomes of the search are equally likely.

An important variation. Instead of using three pointers l, i and u in the search, it is tempting to use only two, namely the current position i and its rate of change, δ; after each unequal comparison, we could then set $i \leftarrow i \pm \delta$ and $\delta \leftarrow \delta/2$ (approximately). It is possible to do this, but only if extreme care is paid to the details, as in the following algorithm. Simpler approaches are doomed to failure!

Algorithm U (*Uniform binary search*). Given a table of records $R_1, R_2, \ldots, R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K. If N is even, the algorithm will sometimes refer to a dummy key K_0 that should be set to $-\infty$ (or any value less than K). We assume that $N \geq 1$.

U1. [Initialize.] Set $i \leftarrow \lceil N/2 \rceil$, $m \leftarrow \lfloor N/2 \rfloor$.

U2. [Compare.] If $K < K_i$, go to U3; if $K > K_i$, go to U4; and if $K = K_i$, the algorithm terminates successfully.

U3. [Decrease i.] (We have pinpointed the search to an interval that contains either m or $m-1$ records; i points just to the right of this interval.) If $m = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i - \lceil m/2 \rceil$; then set $m \leftarrow \lfloor m/2 \rfloor$ and return to U2.

U4. [Increase i.] (We have pinpointed the search to an interval that contains either m or $m-1$ records; i points just to the left of this interval.) If $m = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i + \lceil m/2 \rceil$; then set $m \leftarrow \lfloor m/2 \rfloor$ and return to U2. ▌

Figure 6 shows the corresponding binary tree for the search, when $N = 10$. In an unsuccessful search, the algorithm may make a redundant comparison just before termination; those nodes are shaded in the figure. We may call the search process *uniform* because the difference between the number of a node on level l and the number of its ancestor on level $l - 1$ has a constant value δ for all nodes on level l.

The theory underlying Algorithm U can be understood as follows: Suppose that we have an interval of length $n - 1$ to search; a comparison with the middle element (for n even) or with one of the two middle elements (for n odd) leaves us with two intervals of lengths $\lfloor n/2 \rfloor - 1$ and $\lceil n/2 \rceil - 1$. After repeating this process k times, we obtain 2^k intervals, of which the smallest has length $\lfloor n/2^k \rfloor - 1$ and the largest has length $\lceil n/2^k \rceil - 1$. Hence the lengths of two intervals at the same

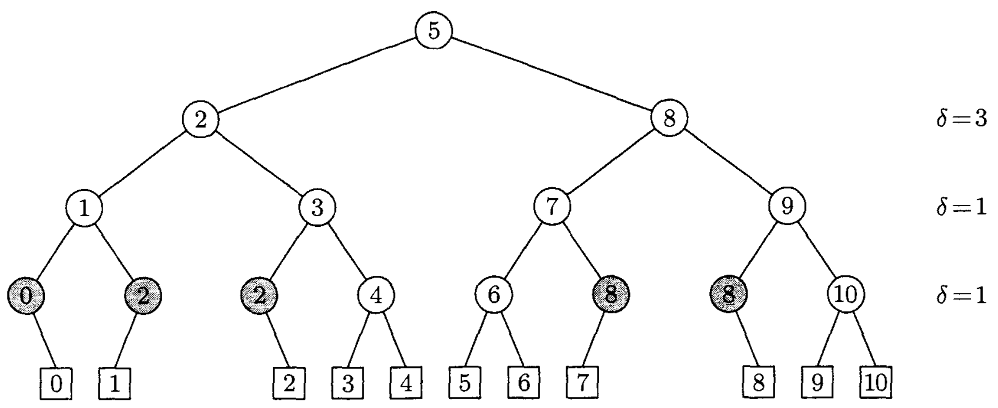


Fig. 6. The comparison tree for a "uniform" binary search, when $N = 10$.

level differ by at most unity; this makes it possible to choose an appropriate "middle" element, without keeping track of the exact lengths.

The principal advantage of Algorithm U is that we need not maintain the value of m at all; we need only refer to a short table of the various δ to use at each level of the tree. Thus the algorithm reduces to the following procedure, which is equally good on binary or decimal computers:

Algorithm C (*Uniform binary search*). This algorithm is just like Algorithm U, but it uses an auxiliary table in place of the calculations involving m. The table entries are

$$\texttt{DELTA}[j] = \left\lfloor \frac{N + 2^{j-1}}{2^j} \right\rfloor, \qquad \text{for } 1 \le j \le \lfloor \lg N \rfloor + 2. \tag{6}$$

C1. [Initialize.] Set $i \leftarrow \texttt{DELTA[1]}$, $j \leftarrow 2$.

C2. [Compare.] If $K < K_i$, go to C3; if $K > K_i$, go to C4; and if $K = K_i$, the algorithm terminates successfully.

C3. [Decrease i.] If $\texttt{DELTA}[j] = 0$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow i - \texttt{DELTA}[j]$, $j \leftarrow j + 1$, and go to C2.

C4. [Increase i.] If $\texttt{DELTA}[j] = 0$, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow i + \texttt{DELTA}[j]$, $j \leftarrow j + 1$, and go to C2. ▌

Exercise 8 proves that this algorithm refers to the artificial key $K_0 = -\infty$ only when N is even.

Program C (*Uniform binary search*). This program does the same job as Program B, using Algorithm C with $\text{rA} \equiv K$, $\text{rI1} \equiv i$, $\text{rI2} \equiv j$, $\text{rI3} \equiv \texttt{DELTA}[j]$.

```
START  ENT1 N+1/2          1          C1. Initialize. i ← ⌊(N + 1)/2⌋.
       ENT2 2              1          j ← 2.
       LDA  K              1
       JMP  2F             1
3H     JE   SUCCESS        C1         Jump if K = K_i.
       J3Z  FAILURE        C1 − S     Jump if DELTA[j] = 0.
       DEC1 0,3            C1 − S − A C3. Decrease i.
```

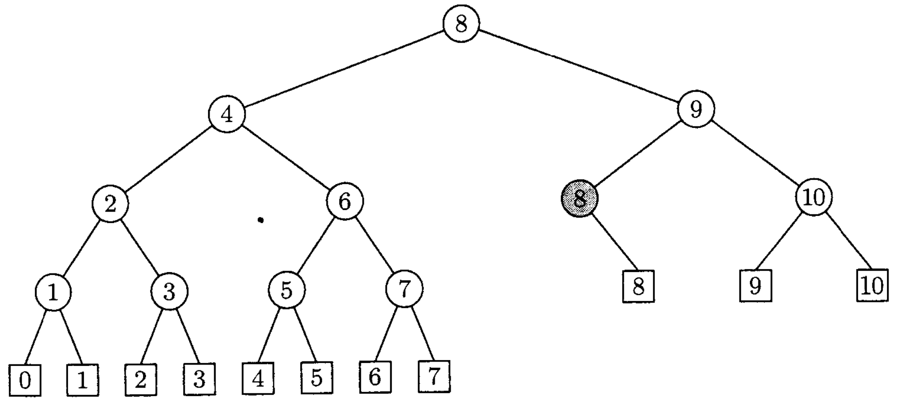


Fig. 7. The comparison tree for Shar's almost uniform search, when $N = 10$.

```
5H      INC2 1         C-1    j ← j+1.
2H      LD3  DELTA,2   C      C2. Compare.
        CMPA KEY,1     C
        JLE  3B        C      Jump if K ≤ K_i.
        INC1 0,3       C2     C4. Increase i.
        J3NZ 5B        C2     Jump if DELTA[j] ≠ 0.
FAILURE EQU  *         1-S    Exit if not in table. ▌
```

In a successful search, this algorithm corresponds to a binary tree with the same internal path length as the tree of Algorithm B, so the average number of comparisons C_N is the same as before. In an unsuccessful search, Algorithm C always makes exactly $\lfloor \lg N \rfloor + 1$ comparisons. The total running time of Program C is not quite symmetrical between left and right branches, since C1 is weighted more heavily than C2, but exercise 11 shows that we have $K < K_i$ roughly as often as $K > K_i$; hence Program C takes approximately

$$\begin{aligned} &(8.5 \lg N - 6)u && \text{for a successful search,} \\ &(8.5 \lfloor \lg N \rfloor + 12)u && \text{for an unsuccessful search.} \end{aligned} \tag{7}$$

This is more than twice as fast as Program B, without using any special properties of binary computers, even though the running times (5) for Program B assume that MIX has a "shift right binary" instruction.

Another modification of binary search, suggested in 1971 by L. E. Shar, will be still faster on some computers, because it is uniform after the first step, and it requires no table. The first step is to compare K with K_i, where $i = 2^k$, $k = \lfloor \lg N \rfloor$. If $K < K_i$, we use a uniform search with the δ's equal to 2^{k-1}, 2^{k-2}, ..., 1, 0. On the other hand, if $K > K_i$ we reset i to $i' = N + 1 - 2^l$, where $l = \lceil \lg(N - 2^k + 1) \rceil$, and pretend that the first comparison was actually $K > K_{i'}$, using a uniform search with the δ's equal to 2^{l-1}, 2^{l-2}, ..., 1, 0.

Shar's method is illustrated for $N = 10$ in Fig. 7. Like the previous algorithms, it never makes more than $\lfloor \lg N \rfloor + 1$ comparisons; hence it makes at most one more than the minimum possible average number of comparisons, in spite of the fact that it occasionally goes through several redundant steps in succession (see exercise 12).

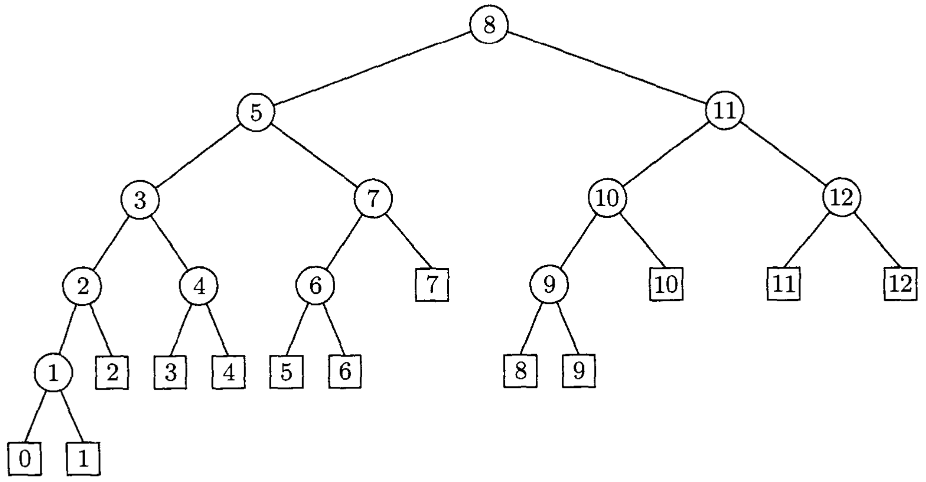


Fig. 8. The Fibonacci tree of order 6.

Still another modification of binary search, which increases the speed of *all* the methods above when N is extremely large, is discussed in exercise 23. See also exercise 24, for a method that is faster yet.

***Fibonaccian search.** In the polyphase merge we have seen that the Fibonacci numbers can play a role analogous to the powers of 2. A similar phenomenon occurs in searching, where Fibonacci numbers provide us with an alternative to binary search. The resulting method is preferable on some computers, because it involves only addition and subtraction, not division by 2. The procedure we are about to discuss should be distinguished from an important numerical procedure called "Fibonacci search," which is used to locate the maximum of a unimodal function [see *Fibonacci Quarterly* **4** (1966), 265–269]; the similarity of names has led to some confusion.

The Fibonaccian search technique looks very mysterious at first glance, if we simply take the program and try to explain what is happening; it seems to work by magic. But the mystery disappears as soon as the corresponding search tree is displayed. Therefore we shall begin our study of the method by looking at *Fibonacci trees.*

Figure 8 shows the Fibonacci tree of order 6. It looks somewhat more like a real-life shrub than the other trees we have been considering, perhaps because many natural processes satisfy a Fibonacci law. In general, the Fibonacci tree of order k has $F_{k+1}-1$ internal (circular) nodes and F_{k+1} external (square) nodes, and it is constructed as follows:

If $k = 0$ or $k = 1$, the tree is simply $\boxed{0}$.

If $k \geq 2$, the root is F_k; the left subtree is the Fibonacci tree of order $k-1$; and the right subtree is the Fibonacci tree of order $k-2$ with all numbers increased by F_k.

Except for the external nodes, the numbers on the two children of each internal node differ from their parent's number by the same amount, and this amount is a Fibonacci number. For example, $5 = 8 - F_4$ and $11 = 8 + F_4$ in Fig. 8. When the difference is F_j, the corresponding Fibonacci difference for the next branch on the left is F_{j-1}, while on the right it skips down to F_{j-2}. For example, $3 = 5 - F_3$ while $10 = 11 - F_2$.

If we combine these observations with an appropriate mechanism for recognizing the external nodes, we arrive at the following method:

Algorithm F (*Fibonaccian search*). Given a table of records $R_1 R_2 \ldots R_N$ whose keys are in increasing order $K_1 < K_2 < \cdots < K_N$, this algorithm searches for a given argument K.

For convenience in description, we assume that $N + 1$ is a perfect Fibonacci number, F_{k+1}. It is not difficult to make the method work for arbitrary N, if a suitable initialization is provided (see exercise 14).

F1. [Initialize.] Set $i \leftarrow F_k$, $p \leftarrow F_{k-1}$, $q \leftarrow F_{k-2}$. (Throughout the algorithm, p and q will be consecutive Fibonacci numbers.)

F2. [Compare.] If $K < K_i$, go to step F3; if $K > K_i$, go to F4; and if $K = K_i$, the algorithm terminates successfully.

F3. [Decrease i.] If $q = 0$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i - q$, and set $(p, q) \leftarrow (q, p-q)$; then return to F2.

F4. [Increase i.] If $p = 1$, the algorithm terminates unsuccessfully. Otherwise set $i \leftarrow i + q$, $p \leftarrow p - q$, then $q \leftarrow q - p$, and return to F2. ▌

The following `MIX` implementation gains speed by making two copies of the inner loop, one in which p is in rI2 and q in rI3, and one in which the registers are reversed; this simplifies step F3. In fact, the program actually keeps $p - 1$ and $q - 1$ in the registers, instead of p and q, in order to simplify the test "$p = 1$?" in step F4.

Program F (*Fibonaccian search*). We follow the previous conventions, with rA $\equiv K$, rI1 $\equiv i$, (rI2 or rI3) $\equiv p - 1$, (rI3 or rI2) $\equiv q - 1$.

```
START LDA  K                1          F1. Initialize.
      ENT1 $F_k$            1          $i \leftarrow F_k$.
      ENT2 $F_{k-1}$-1      1          $p \leftarrow F_{k-1}$.
      ENT3 $F_{k-2}$-1      1          $q \leftarrow F_{k-2}$.
      JMP  F2A              1          To step F2.
F4A   INC1 1,3        $C2 - S - A$     F4. Increase i. $i \leftarrow i + q$.
      DEC2 1,3        $C2 - S - A$     $p \leftarrow p - q$.
      DEC3 1,2        $C2 - S - A$     $q \leftarrow q - p$.
F2A   CMPA KEY,1           $C$         F2. Compare.
      JL   F3A             $C$         To F3 if $K < K_i$.
      JE   SUCCESS         $C2$        Exit if $K = K_i$.
      J2NZ F4A           $C2 - S$      To F4 if $p \neq 1$.
      JMP  FAILURE         $A$         Exit if not in table.
F3A   DEC1 1,3             $C1$        F3. Decrease i. $i \leftarrow i - q$.
      DEC2 1,3             $C1$        $p \leftarrow p - q$.
      J3NN F2B             $C1$        Swap registers if $q > 0$.
      JMP  FAILURE     $1 - S - A$     Exit if not in table.
```

```
18  F4B   INC1 1,2            (Lines 18–29 are parallel to 06–17.)
19        DEC3 1,2
20        DEC2 1,3
21  F2B   CMPA KEY,1
22        JL   F3B
23        JE   SUCCESS
24        J3NZ F4B
25        JMP  FAILURE
26  F3B   DEC1 1,2
27        DEC3 1,2
28        J2NN F2A
29        JMP  FAILURE        ▌
```

The running time of this program is analyzed in exercise 18. Figure 8 shows, and the analysis proves, that a left branch is taken somewhat more often than a right branch. Let C, $C1$, and $(C2-S)$ be the respective number of times steps F2, F3, and F4 are performed. Then we have

$$\begin{aligned} C &= (\text{ave}\quad \phi k/\sqrt{5}+O(1),\quad \max\ k-1),\\ C1 &= (\text{ave}\quad k/\sqrt{5}+O(1),\quad \max\ k-1),\\ C2-S &= (\text{ave}\ \phi^{-1}k/\sqrt{5}+O(1),\quad \max\ \lfloor k/2\rfloor). \end{aligned} \tag{8}$$

Thus the left branch is taken about $\phi \approx 1.618$ times as often as the right branch (a fact that we might have guessed, since each probe divides the remaining interval into two parts, with the left part about ϕ times as large as the right). The total average running time of Program F therefore comes to approximately

$$\tfrac{1}{5}\bigl((18+4\phi)k+31-26\phi\bigr)u \approx (7.050\lg N+1.08)u \tag{9}$$

for a successful search, plus $(9-3\phi)u \approx 4.15u$ for an unsuccessful search. This is faster than Program C, although the worst case running time (roughly $8.6\lg N$) is slightly slower.

Interpolation search. Let's forget computers for a moment, and consider how people actually carry out a search. Sometimes everyday life provides us with clues that lead to good algorithms.

Imagine yourself looking up a word in a dictionary. You probably *don't* begin by looking first at the middle page, then looking at the 1/4 or 3/4 point, etc., as in a binary search. It's even less likely that you use a Fibonaccian search!

If the word you want starts with the letter A, you probably begin near the front of the dictionary. In fact, many dictionaries have thumb indexes that show the starting page or the middle page for the words beginning with a fixed letter. This thumb-index technique can readily be adapted to computers, and it will speed up the search; such algorithms are explored in Section 6.3.

Yet even after the initial point of search has been found, your actions still are not much like the methods we have discussed. If you notice that the desired word is alphabetically much greater than the words on the page being examined, you will turn over a fairly large chunk of pages before making the next reference.

This is quite different from the algorithms above, which make no distinction between "much greater" and "slightly greater."

Such considerations suggest an algorithm that might be called *interpolation search:* When we know that K lies between K_l and K_u, we can choose the next probe to be about $(K - K_l)/(K_u - K_l)$ of the way between l and u, assuming that the keys are numeric and that they increase in a roughly constant manner throughout the interval.

Interpolation search is asymptotically superior to binary search. One step of binary search essentially reduces the amount of uncertainty from n to $\frac{1}{2}n$, while one step of interpolation search essentially reduces it to $\sqrt{n}$, when the keys in the table are randomly distributed. Hence interpolation search takes about $\lg\lg N$ steps, on the average, to reduce the uncertainty from N to 2. (See exercise 22.)

However, computer simulation experiments show that interpolation search does not decrease the number of comparisons enough to compensate for the extra computing time involved, unless the table is rather large. Typical files aren't sufficiently random, and the difference between $\lg\lg N$ and $\lg N$ is not substantial unless N exceeds, say, $2^{16} = 65{,}536$. Interpolation is most successful in the early stages of searching a large possibly external file; after the range has been narrowed down, binary search finishes things off more quickly. (Note that dictionary lookup by hand is essentially an external, not an internal, search. We shall discuss external searching later.)

History and bibliography. The earliest known example of a long list of items that was sorted into order to facilitate searching is the remarkable Babylonian reciprocal table of Inakibit-Anu, dating from about 200 B.C. This clay tablet contains more than 100 pairs of values, which appear to be the beginning of a list of approximately 500 multiple-precision sexagesimal numbers and their reciprocals, sorted into lexicographic order. For example, the list included the following sequence of entries:

01 13 09 34 29 08 08 53 20	49 12 27
01 13 14 31 52 30	49 09 07 12
01 13 43 40 48	48 49 41 15
01 13 48 40 30	48 46 22 59 25 25 55 33 20
01 14 04 26 40	48 36

The task of sorting 500 entries like this, given the technology available at that time, must have been phenomenal. [See D. E. Knuth, *Selected Papers on Computer Science* (Cambridge Univ. Press, 1996), Chapter 11, for further details.]

It is fairly natural to sort numerical values into order, but an order relation between letters or words does not suggest itself so readily. Yet a collating sequence for individual letters was present already in the most ancient alphabets. For example, many of the Biblical psalms have verses that follow a strict alphabetic sequence, the first verse starting with aleph, the second with beth, etc.; this was an aid to memory. Eventually the standard sequence of letters was used by Semitic and Greek peoples to denote numerals; for example, α, β, γ stood for 1, 2, 3, respectively.

The use of alphabetic order for entire words seems to be a much later invention; it is something we might think is obvious, yet it has to be taught to children, and at some point in history it was necessary to teach it to adults. Several lists from about 300 B.C. have been found on the Aegean Islands, giving the names of people in certain religious cults; these lists have been alphabetized, but only by the first letter, thus representing only the first pass of a left-to-right radix sort. Some Greek papyri from the years A.D. 134–135 contain fragments of ledgers that show the names of taxpayers alphabetized by the first two letters. Apollonius Sophista used alphabetic order on the first two letters, and often on subsequent letters, in his lengthy concordance of Homer's poetry (first century A.D.). A few examples of more perfect alphabetization are known, notably Galen's *Hippocratic Glosses* (c. 200), but they are very rare. Words were arranged by their first letter only in the *Etymologiarum* of St. Isidorus (c. 630, Book x); and the *Corpus Glossary* (c. 725) used only the first two letters of each word. The latter two works were perhaps the largest nonnumerical files of data to be compiled during the Middle Ages.

It is not until Giovanni di Genoa's *Catholicon* (1286) that we find a specific description of true alphabetical order. In his preface, Giovanni explained that

amo	precedes	*bibo*
abeo	precedes	*adeo*
amatus	precedes	*amor*
imprudens	precedes	*impudens*
iusticia	precedes	*iustus*
polisintheton	precedes	*polissenus*

(thereby giving examples of situations in which the ordering is determined by the 1st, 2nd, ..., 6th letters), "and so in like manner." He remarked that strenuous effort was required to devise these rules. "I beg of you, therefore, good reader, do not scorn this great labor of mine and this order as something worthless."

A detailed study of the development of alphabetic order, up to the time printing was invented, has been made by Lloyd W. Daly [*Collection Latomus* **90** (1967), 100 pages]. He found some interesting old manuscripts that were evidently used as worksheets while sorting words by their first letters (see pages 89–90 of his monograph).

The first dictionary of English, Robert Cawdrey's *Table Alphabeticall* (London, 1604), contains the following instructions:

> Nowe if the word, which thou art desirous to finde, beginne with (a) then looke in the beginning of this Table, but if with (v) looke towards the end. Againe, if thy word beginne with (ca) looke in the beginning of the letter (c) but if with (cu) then looke toward the end of that letter. And so of all the rest. &c.

Cawdrey seems to have been teaching *himself* how to alphabetize as he prepared his dictionary; numerous misplaced words appear on the first few pages, but the alphabetic order in the last part is not as bad.

Binary search was first mentioned by John Mauchly, in what was perhaps the first published discussion of nonnumerical programming methods [*Theory and Techniques for the Design of Electronic Digital Computers*, edited by G. W. Patterson, **1** (1946), 9.7–9.8; **3** (1946), 22.8–22.9]. The method became well known to programmers, but nobody seems to have worked out the details of what should be done when N does not have the special form $2^n - 1$. [See A. D. Booth, *Nature* **176** (1955), 565; A. I. Dumey, *Computers and Automation* **5** (December 1956), 7, where binary search is called "Twenty Questions"; Daniel D. McCracken, *Digital Computer Programming* (Wiley, 1957), 201–203; and M. Halpern, *CACM* **1**, 1 (February 1958), 1–3.]

D. H. Lehmer [*Proc. Symp. Appl. Math.* **10** (1960), 180–181] was apparently the first to publish a binary search algorithm that works for all N. The next step was taken by H. Bottenbruch [*JACM* **9** (1962), 214], who presented an interesting variation of Algorithm B that avoids a separate test for equality until the very end: Using

$$i \leftarrow \lceil (l+u)/2 \rceil$$

instead of $i \leftarrow \lfloor (l+u)/2 \rfloor$ in step B2, he set $l \leftarrow i$ whenever $K \geq K_i$; then $u - l$ decreases at every step. Eventually, when $l = u$, we have $K_l \leq K < K_{l+1}$, and we can test whether or not the search was successful by making one more comparison. (He assumed that $K \geq K_1$ initially.) This idea speeds up the inner loop slightly on many computers, and the same principle can be used with all of the algorithms we have discussed in this section; but a successful search will require about one more iteration, on the average, because of (2). Since the inner loop is performed only about $\lg N$ times, this tradeoff between an extra iteration and a faster loop does not save time unless n is extremely large. (See exercise 23.) On the other hand Bottenbruch's algorithm will find the rightmost occurrence of a given key when the table contains duplicates, and this property is occasionally important.

K. E. Iverson [*A Programming Language* (Wiley, 1962), 141] gave the procedure of Algorithm B, but without considering the possibility of an unsuccessful search. D. E. Knuth [*CACM* **6** (1963), 556–558] presented Algorithm B as an example used with an automated flowcharting system. The uniform binary search, Algorithm C, was suggested to the author by A. K. Chandra of Stanford University in 1971.

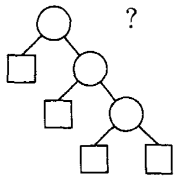
Fibonaccian searching was invented by David E. Ferguson [*CACM* **3** (1960), 648]. Binary trees similar to Fibonacci trees appeared in the pioneering work of the Norwegian mathematician Axel Thue as early as 1910 (see exercise 28). A Fibonacci tree without labels was also exhibited as a curiosity in the first edition of Hugo Steinhaus's popular book *Mathematical Snapshots* (New York: Stechert, 1938), page 28; he drew it upside down and made it look like a real tree, with right branches twice as long as left branches so that all the leaves would occur at the same level.

Interpolation searching was suggested by W. W. Peterson [*IBM J. Res. & Devel.* **1** (1957), 131–132]. A correct analysis of its average behavior was not discovered until many years later (see exercise 22).

EXERCISES

▸ **1.** [*21*] Prove that if $u < l$ in step B2 of the binary search, we have $u = l - 1$ and $K_u < K < K_l$. (Assume by convention that $K_0 = -\infty$ and $K_{N+1} = +\infty$, although these artificial keys are never really used by the algorithm so they need not be present in the actual table.)

▸ **2.** [*22*] Would Algorithm B still work properly when K is present in the table if we (a) changed step B5 to "$l \leftarrow i$" instead of "$l \leftarrow i+1$"? (b) changed step B4 to "$u \leftarrow i$" instead of "$u \leftarrow i - 1$"? (c) made both of these changes?

3. [*15*] What searching method corresponds to the tree  ?

What is the average number of comparisons made in a successful search? in an unsuccessful search?

4. [*20*] If a search using Program 6.1S (sequential search) takes exactly 638 units of time, how long does it take with Program B (binary search)?

5. [*M24*] For what values of N is Program B actually *slower* than a sequential search (Program 6.1Q$'$) on the average, assuming that the search is successful?

6. [*28*] (K. E. Iverson.) Exercise 5 suggests that it would be best to have a hybrid method, changing from binary search to sequential search when the remaining interval has length less than some judiciously chosen value. Write an efficient MIX program for such a search and determine the best changeover value.

▸ **7.** [*M22*] Would Algorithm U still work properly if we changed step U1 so that

a) both i and m are set equal to $\lfloor N/2 \rfloor$?

b) both i and m are set equal to $\lceil N/2 \rceil$?

[*Hint:* Suppose the first step were "Set $i \leftarrow 0$, $m \leftarrow N$ (or $N+1$), go to U4."]

8. [*M20*] Let $\delta_j =$ DELTA[j] be the jth increment in Algorithm C, as defined in (6).

a) What is the sum $\sum_{j=0}^{\lfloor \lg N \rfloor + 2} \delta_j$?

b) What are the minimum and maximum values of i that can occur in step C2?

9. [*20*] Is there any value of $N > 1$ for which Algorithm B and C are exactly equivalent, in the sense that they will both perform the same sequence of comparisons for all search arguments?

10. [*21*] Explain how to write a MIX program for Algorithm C containing approximately $7 \lg N$ instructions and having a running time of about $4.5 \lg N$ units.

11. [*M26*] Find exact formulas for the average values of $C1$, $C2$, and A in the frequency analysis of Program C, as a function of N and S.

12. [*20*] Draw the binary search tree corresponding to Shar's method when $N = 12$.

13. [*M24*] Tabulate the average number of comparisons made by Shar's method, for $1 \le N \le 16$, considering both successful and unsuccessful searches.

14. [*21*] Explain how to extend Algorithm F so that it will apply for all $N \ge 1$.

15. [*M19*] For what values of k does the Fibonacci tree of order k define an optimal search procedure, in the sense that the fewest comparisons are made on the average?

16. [*21*] Figure 9 shows the lineal chart of the rabbits in Fibonacci's original rabbit problem (see Section 1.2.8). Is there a simple relationship between this and the Fibonacci tree discussed in the text?

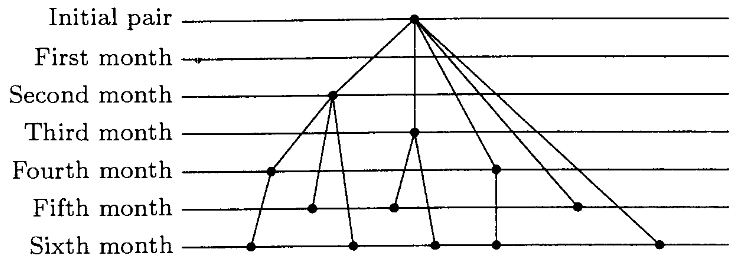


Fig. 9. Pairs of rabbits breeding by Fibonacci's rule.

17. [*M21*] From exercise 1.2.8–34 (or exercise 5.4.2–10) we know that every positive integer n has a unique representation as a sum of Fibonacci numbers

$$n = F_{a_1} + F_{a_2} + \cdots + F_{a_r},$$

where $r \geq 1$, $a_j \geq a_{j+1} + 2$ for $1 \leq j < r$, and $a_r \geq 2$. Prove that in the Fibonacci tree of order k, the path from the root to node $\textcircled{n}$ has length $k + 1 - r - a_r$.

18. [*M30*] Find exact formulas for the average values of $C1$, $C2$, and A in the frequency analysis of Program F, as a function of k, F_k, F_{k+1}, and S.

19. [*M42*] Carry out a detailed analysis of the average running time of the algorithm suggested in exercise 14.

20. [*M22*] The number of comparisons required in a binary search is approximately $\log_2 N$, and in the Fibonaccian search it is roughly $(\phi/\sqrt{5})\log_\phi N$. The purpose of this exercise is to show that these formulas are special cases of a more general result.

Let p and q be positive numbers with $p+q=1$. Consider a search algorithm that, given a table of N numbers in increasing order, starts by comparing the argument with the (pN)th key, and iterates this procedure on the smaller blocks. (The binary search has $p = q = 1/2$; the Fibonacci search has $p = 1/\phi$, $q = 1/\phi^2$.)

If $C(N)$ denotes the average number of comparisons required to search a table of size N, it approximately satisfies the relations

$$C(1) = 0; \qquad C(N) = 1 + pC(pN) + qC(qN) \quad \text{for } N > 1.$$

This happens because there is probability p (roughly) that the search reduces to a pN-element search, and probability q that it reduces to a qN-element search, after the first comparison. When N is large, we may ignore the small-order effect caused by the fact that pN and qN aren't exactly integers.

a) Show that $C(N) = \log_b N$ satisfies these relations exactly, for a certain choice of b. For binary and Fibonaccian search, this value of b agrees with the formulas derived earlier.

b) Consider the following argument: "With probability p, the size of the interval being scanned in this algorithm is divided by $1/p$; with probability q, the interval size is divided by $1/q$. Therefore the interval is divided by $p \cdot (1/p) + q \cdot (1/q) = 2$ on the average, so the algorithm is exactly as good as the binary search, regardless of p and q." Is there anything wrong with this reasoning?

21. [*20*] Draw the binary tree corresponding to interpolation search when $N = 10$.

22. [*M41*] (A. C. Yao and F. F. Yao.) Show that an appropriate formulation of interpolation search requires asymptotically $\lg\lg N$ comparisons, on the average, when applied to N independent uniform random keys that have been sorted. Furthermore *all* search algorithms on such tables must make asymptotically $\lg\lg N$ comparisons, on the average.

▸ **23.** [*25*] The binary search algorithm of H. Bottenbruch, mentioned at the close of this section, avoids testing for equality until the very end of the search. (During the algorithm we know that $K_l \le K < K_{u+1}$, and the case of equality is not examined until $l = u$.) Such a trick would make Program B run a little bit faster for large N, since the "JE" instruction could be removed from the inner loop. (However, the idea wouldn't really be practical since $\lg N$ is always rather small; we would need $N > 2^{66}$ in order to compensate for the extra work necessary on a successful search, because the running time $(18\lg N - 16)u$ of (5) is "decreased" to $(17.5\lg N + 17)u$!)

Show that *every* search algorithm corresponding to a binary tree can be adapted to a search algorithm that uses two-way branching ($<$ versus $\ge$) at the internal nodes of the tree, in place of the three-way branching ($<$, $=$, or $>$) used in the text's discussion. In particular, show how to modify Algorithm C in this way.

▸ **24.** [*23*] We have seen in Sections 2.3.4.5 and 5.2.3 that the complete binary tree is a convenient way to represent a minimum-path-length tree in consecutive locations. Devise an efficient search method based on this representation. [*Hint:* Is it possible to use multiplication by 2 instead of division by 2 in a binary search?]

▸ **25.** [*M25*] Suppose that a binary tree has a_k internal nodes and b_k external nodes on level k, for $k = 0, 1, \ldots$. (The root is at level zero.) Thus in Fig. 8 we have $(a_0, a_1, \ldots, a_5) = (1, 2, 4, 4, 1, 0)$ and $(b_0, b_1, \ldots, b_5) = (0, 0, 0, 4, 7, 2)$.

a) Show that a simple algebraic relationship holds between the generating functions $A(z) = \sum_k a_k z^k$ and $B(z) = \sum_k b_k z^k$.

b) The probability distribution for a successful search in a binary tree has the generating function $g(z) = zA(z)/N$, and for an unsuccessful search the generating function is $h(z) = B(z)/(N+1)$. (Thus in the text's notation we have $C_N = \text{mean}(g)$, $C'_N = \text{mean}(h)$, and Eq. (2) gives a relation between these quantities.) Find a relation between $\text{var}(g)$ and $\text{var}(h)$.

26. [*22*] Show that Fibonacci trees are related to polyphase merge sorting on three tapes.

27. [*M30*] (H. S. Stone and John Linn.) Consider a search process that uses k processors simultaneously and that is based solely on comparisons of keys. Thus at every step of the search, k indices $i_1, \ldots, i_k$ are specified, and we perform k simultaneous comparisons; if $K = K_{i_j}$ for some j, the search terminates successfully, otherwise the search proceeds to the next step based on the 2^k possible outcomes $K < K_{i_j}$ or $K > K_{i_j}$, for $1 \le j \le k$.

Prove that such a process must always take at least approximately $\log_{k+1} N$ steps on the average, as $N \to \infty$, assuming that each key of the table is equally likely as a search argument. (Hence the potential increase in speed over 1-processor binary search is only a factor of $\lg(k+1)$, not the factor of k we might expect. In this sense it is more efficient to assign each processor to a different, independent search problem, instead of making them cooperate on a single search.)

28. [*M23*] Define *Thue trees* T_n by means of algebraic expressions in a binary operator $*$ as follows: $T_0(x) = x * x$, $T_1(x) = x$, $T_{n+2}(x) = T_{n+1}(x) * T_n(x)$.

a) The number of leaves of T_n is the number of occurrences of x when $T_n(x)$ is written out in full. Express this number in terms of Fibonacci numbers.

b) Prove that if the binary operator $*$ satisfies the axiom

$$\bullet\,((x * x) * x) * ((x * x) * x) = x\,,$$

then $T_m(T_n(x)) = T_{m+n-1}(x)$ for all $m \geq 0$ and $n \geq 1$.

▸ **29.** [*22*] (Paul Feldman, 1975.) Instead of assuming that $K_1 < K_2 < \cdots < K_N$, assume only that $K_{p(1)} < K_{p(2)} < \cdots < K_{p(N)}$ where the permutation $p(1)p(2)\ldots p(N)$ is an involution, and $p(j) = j$ for all even values of j. Show that we can locate any given key K, or determine that K is not present, by making at most $2\lfloor \lg N \rfloor + 1$ comparisons.

30. [*27*] (*Involution coding.*) Using the idea of the previous exercise, find a way to arrange N distinct keys in such a way that their relative order implicitly encodes an arbitrarily given array of t-bit numbers $x_1, x_2, \ldots, x_m$, when $m \leq N/4 + 1 - 2^t$. With your arrangement it should be possible to determine the leading k bits of x_j by making only k comparisons, for any given j, as well as to look up an arbitrary key with $\leq 2\lfloor \lg N \rfloor + 1$ comparisons. (This result is used in theoretical studies of data structures that are asymptotically efficient in both time and space.)

6.2.2. Binary Tree Searching

In the preceding section, we learned that an implicit binary tree structure makes the behavior of binary search and Fibonaccian search easier to understand. For a given value of N, the tree corresponding to binary search achieves the theoretical minimum number of comparisons that are necessary to search a table by means of key comparisons. But the methods of the preceding section are appropriate mainly for fixed-size tables, since the sequential allocation of records makes insertions and deletions rather expensive. If the table is changing dynamically, we might spend more time maintaining it than we save in binary-searching it.

The use of an *explicit* binary tree structure makes it possible to insert and delete records quickly, as well as to search the table efficiently. As a result, we essentially have a method that is useful both for searching and for sorting. This gain in flexibility is achieved by adding two link fields to each record of the table.

Techniques for searching a growing table are often called *symbol table algorithms*, because assemblers and compilers and other system routines generally use such methods to keep track of user-defined symbols. For example, the key of each record within a compiler might be a symbolic identifier denoting a variable in some FORTRAN or C program, and the rest of the record might contain information about the type of that variable and its storage allocation. Or the key might be a symbol in a `MIXAL` program, with the rest of the record containing the equivalent of that symbol. The tree search and insertion routines to be described in this section are quite efficient for use as symbol table algorithms, especially in applications where it is desirable to print out a list of the symbols in alphabetic order. Other symbol table algorithms are described in Sections 6.3 and 6.4.

Figure 10 shows a binary search tree containing the names of eleven signs of the zodiac. If we now search for the twelfth name, `SAGITTARIUS`, starting at the

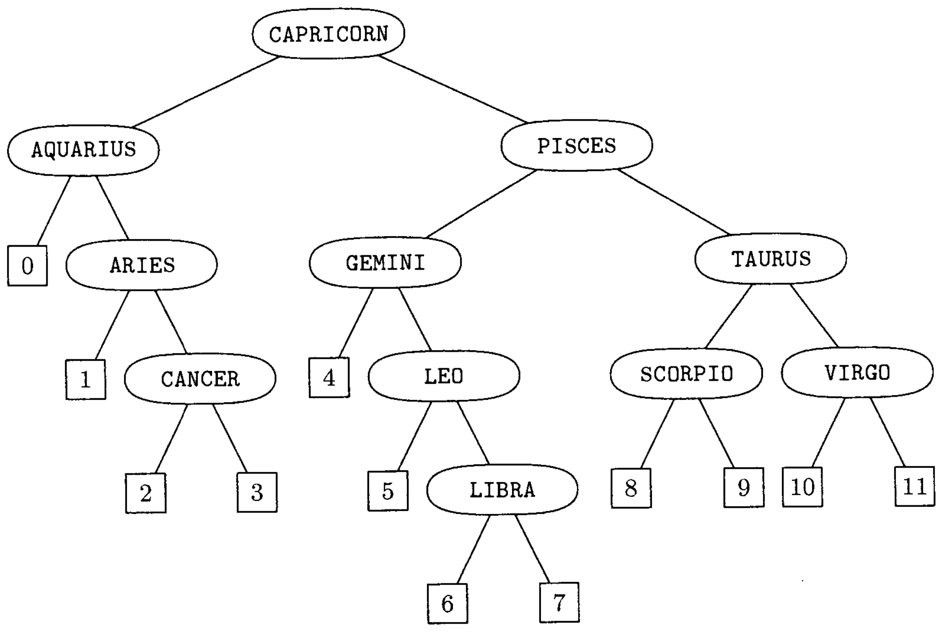


Fig. 10. A binary search tree.

root or apex of the tree, we find it is greater than CAPRICORN, so we move to the right; it is greater than PISCES, so we move right again; it is less than TAURUS, so we move left; and it is less than SCORPIO, so we arrive at external node $\boxed{8}$. The search was unsuccessful; we can now *insert* SAGITTARIUS at the place the search ended, by linking it into the tree in place of the external node $\boxed{8}$. In this way the table can grow without the necessity of moving any of the existing records. Figure 10 was formed by starting with an empty tree and successively inserting the keys CAPRICORN, AQUARIUS, PISCES, ARIES, TAURUS, GEMINI, CANCER, LEO, VIRGO, LIBRA, SCORPIO, in this order.

All of the keys in the left subtree of the root in Fig. 10 are alphabetically less than CAPRICORN, and all keys in the right subtree are alphabetically greater. A similar statement holds for the left and right subtrees of every node. It follows that the keys appear in strict alphabetic sequence from left to right,

AQUARIUS, ARIES, CANCER, CAPRICORN, GEMINI, LEO, ..., VIRGO

if we traverse the tree in *symmetric order* (see Section 2.3.1), since symmetric order is based on traversing the left subtree of each node just before that node, then traversing the right subtree.

The following algorithm spells out the searching and insertion processes in detail.

Algorithm T (*Tree search and insertion*). Given a table of records that form a binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place.

The nodes of the tree are assumed to contain at least the following fields:

$$\begin{aligned}\texttt{KEY(P)} &= \text{key stored in } \texttt{NODE(P)};\\ \texttt{LLINK(P)} &= \text{pointer to left subtree of } \texttt{NODE(P)};\\ \texttt{RLINK(P)} &= \text{pointer to right subtree of } \texttt{NODE(P)}.\end{aligned}$$

Null subtrees (the external nodes in Fig. 10) are represented by the null pointer Λ. The variable ROOT points to the root of the tree. For convenience, we assume that the tree is not empty (that is, ROOT $\neq \Lambda$), since the necessary operations are trivial when ROOT $= \Lambda$.

T1. [Initialize.] Set P $\leftarrow$ ROOT. (The pointer variable P will move down the tree.)

T2. [Compare.] If $K <$ KEY(P), go to T3; if $K >$ KEY(P), go to T4; and if $K =$ KEY(P), the search terminates successfully.

T3. [Move left.] If LLINK(P) $\neq \Lambda$, set P $\leftarrow$ LLINK(P) and go back to T2. Otherwise go to T5.

T4. [Move right.] If RLINK(P) $\neq \Lambda$, set P $\leftarrow$ RLINK(P) and go back to T2.

T5. [Insert into tree.] (The search is unsuccessful; we will now put K into the tree.) Set Q $\Leftarrow$ AVAIL, the address of a new node. Set KEY(Q) $\leftarrow K$, LLINK(Q) $\leftarrow$ RLINK(Q) $\leftarrow \Lambda$. (In practice, other fields of the new node should also be initialized.) If K was less than KEY(P), set LLINK(P) $\leftarrow$ Q, otherwise set RLINK(P) $\leftarrow$ Q. (At this point we could set P $\leftarrow$ Q and terminate the algorithm successfully.) ▌

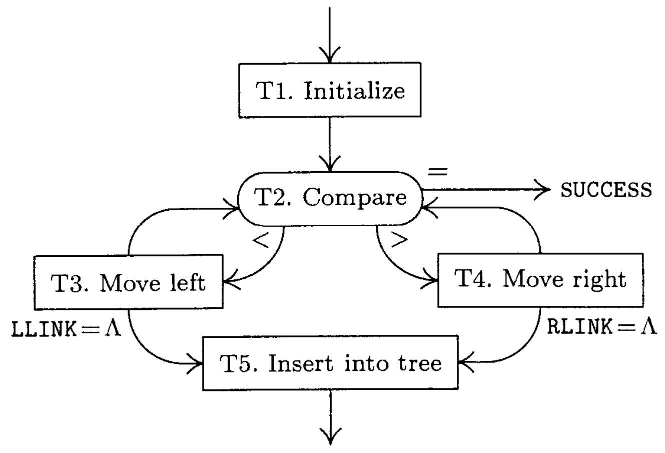


Fig. 11. Tree search and insertion.

This algorithm lends itself to a convenient machine language implementation. We may assume, for example, that the tree nodes have the form

+	0	LLINK	RLINK
KEY			

(1)

followed perhaps by additional words of INFO. Using an AVAIL list for the free storage pool, as in Chapter 2, we can write the following MIX program:

Program T (*Tree search and insertion*). rA $\equiv K$, rI1 $\equiv$ P, rI2 $\equiv$ Q.

```
01  LLINK EQU   2:3
02  RLINK EQU   4:5
03  START LDA   K              1        T1. Initialize.
04        LD1   ROOT           1        P ← ROOT.
05        JMP   2F             1
06  4H    LD2   0,1(RLINK)     C2       T4. Move right. Q ← RLINK(P).
07        J2Z   5F             C2       To T5 if Q = Λ.
08  1H    ENT1  0,2            C − 1    P ← Q.
09  2H    CMPA  1,1            C        T2. Compare.
10        JG    4B             C        To T4 if K > KEY(P).
11        JE    SUCCESS        C1       Exit if K = KEY(P).
12        LD2   0,1(LLINK)     C1 − S   T3. Move left. Q ← LLINK(P).
13        J2NZ  1B             C1 − S   To T2 if Q ≠ Λ.
14  5H    LD2   AVAIL          1 − S    T5. Insert into tree.
15        J2Z   OVERFLOW       1 − S
16        LDX   0,2(RLINK)     1 − S
17        STX   AVAIL          1 − S    Q ⇐ AVAIL.
18        STA   1,2            1 − S    KEY(Q) ← K.
19        STZ   0,2            1 − S    LLINK(Q) ← RLINK(Q) ← Λ.
20        JL    1F             1 − S    Was K < KEY(P)?
21        ST2   0,1(RLINK)     A        RLINK(P) ← Q.
22        JMP   *+2            A
23  1H    ST2   0,1(LLINK)     1 − S − A LLINK(P) ← Q.
24  DONE  EQU   *              1 − S    Exit after insertion. ▌
```

The first 13 lines of this program do the search; the last 11 lines do the insertion. The running time for the searching phase is $(7C + C1 - 3S + 4)u$, where

$$C = \text{number of comparisons made;}$$
$$C1 = \text{number of times } K \le \texttt{KEY(P)};$$
$$C2 = \text{number of times } K > \texttt{KEY(P)};$$
$$S = [\text{search is successful}].$$

On the average we have $C1 = \frac{1}{2}(C + S)$, since $C1 + C2 = C$ and $C1 - S$ has the same probability distribution as $C2$; so the running time is about $(7.5C - 2.5S + 4)u$. This compares favorably with the binary search algorithms that use an implicit tree (see Program 6.2.1C). By duplicating the code as in Program 6.2.1F we could effectively eliminate line 08 of Program T, reducing the running time to $(6.5C - 2.5S + 5)u$. If the search is unsuccessful, the insertion phase of the program costs an extra $14u$ or $15u$.

Algorithm T can conveniently be adapted to *variable-length keys* and variable-length records. For example, if we allocate the available space sequentially, in a last-in-first-out manner, we can easily create nodes of varying size; the first word of (1) could indicate the size. Since this is an efficient use of storage, symbol table algorithms based on trees are often especially attractive for use in compilers, assemblers, and loaders.

But what about the worst case? Programmers are often skeptical of Algorithm T when they first see it. If the keys of Fig. 10 had been entered into the tree in alphabetic order AQUARIUS, ..., VIRGO instead of the calendar order CAPRICORN, ..., SCORPIO, the algorithm would have built a degenerate tree that essentially specifies a *sequential* search. All LLINKs would be null. Similarly, if the keys come in the uncommon order

AQUARIUS, VIRGO, ARIES, TAURUS, CANCER, SCORPIO,
CAPRICORN, PISCES, GEMINI, LIBRA, LEO

we obtain a "zigzag" tree that is just as bad. (Try it!)

On the other hand, the particular tree in Fig. 10 requires only $3\frac{2}{11}$ comparisons, on the average, for a successful search; this is just a little higher than the minimum possible average number of comparisons, 3, achievable in the best possible binary tree.

When we have a fairly balanced tree, the search time is roughly proportional to $\log N$, but when we have a degenerate tree, the search time is roughly proportional to N. Exercise 2.3.4.5–5 proves that the average search time would be roughly proportional to $\sqrt{N}$ if we considered each N-node binary tree to be equally likely. What behavior can we really expect from Algorithm T?

Fortunately, it turns out that tree search will require only about $2\ln N \approx 1.386\lg N$ comparisons, if the keys are inserted into the tree in random order; well-balanced trees are common, and degenerate trees are very rare.

There is a surprisingly simple proof of this fact. Let us assume that each of the $N!$ possible orderings of the N keys is an equally likely sequence of insertions for building the tree. The number of comparisons needed to find a key is exactly one more than the number of comparisons that were needed when that key was entered into the tree. Therefore if C_N is the average number of comparisons involved in a successful search and C'_N is the average number in an unsuccessful search, we have

$$C_N = 1 + \frac{C'_0 + C'_1 + \cdots + C'_{N-1}}{N}. \tag{2}$$

But the relation between internal and external path length tells us that

$$C_N = \left(1 + \frac{1}{N}\right) C'_N - 1; \tag{3}$$

this is Eq. 6.2.1–(2). Putting (3) together with (2) yields

$$(N+1)C'_N = 2N + C'_0 + C'_1 + \cdots + C'_{N-1}. \tag{4}$$

This recurrence is easy to solve. Subtracting the equation

$$NC'_{N-1} = 2(N-1) + C'_0 + C'_1 + \cdots + C'_{N-2},$$

we obtain

$$(N+1)C'_N - NC'_{N-1} = 2 + C'_{N-1}, \qquad \text{hence} \qquad C'_N = C'_{N-1} + 2/(N+1).$$

Since $C'_0 = 0$, this means that

$$C'_N = 2H_{N+1} - 2. \tag{5}$$

Applying (3) and simplifying yields the desired result

$$C_N = 2\left(1 + \frac{1}{N}\right)H_N - 3. \tag{6}$$

Exercises 6, 7, and 8 below give more detailed information; it is possible to compute the exact probability distribution of C_N and C'_N, not merely the average values.

Tree insertion sorting. Algorithm T was developed for searching, but it can also be used as the basis of an internal *sorting* algorithm; in fact, we can view it as a natural generalization of list insertion, Algorithm 5.2.1L. When properly programmed, its average running time will be only a little slower than some of the best algorithms we discussed in Chapter 5. After the tree has been constructed for all keys, a symmetric tree traversal (Algorithm 2.3.1T) will visit the records in sorted order.

A few precautions are necessary, however. Something different needs to be done if $K =$ `KEY(P)` in step T2, since we are sorting instead of searching. One solution is to treat $K =$ `KEY(P)` exactly as if $K >$ `KEY(P)`; this leads to a stable sorting method. (Equal keys will not necessarily be adjacent in the tree; they will only be adjacent in symmetric order.) But if many duplicate keys are present, this method will cause the tree to get badly unbalanced, and the sorting will slow down. Another idea is to keep a list, for each node, of all records having the same key; this requires another link field, but it will make the sorting faster when a lot of equal keys occur.

Thus if we are interested only in sorting, not in searching, Algorithm T isn't the best, but it isn't bad. And if we have an application that combines searching with sorting, the tree method can be warmly recommended.

It is interesting to note that there is a strong relation between the analysis of tree insertion sorting and the analysis of quicksort, although the methods are superficially dissimilar. If we successively insert N keys into an initially empty tree, we make the same average number of comparisons between keys as Algorithm 5.2.2Q does, with minor exceptions. For example, in tree insertion every key gets compared with K_1, and then every key less than K_1 gets compared with the first key less than K_1, etc.; in quicksort, every key gets compared to the first partitioning element K and then every key less than K gets compared to a particular element less than K, etc. The average number of comparisons needed in both cases is $NC_N - N$. (However, Algorithm 5.2.2Q actually makes a few more comparisons, in order to speed up the inner loops.)

Deletions. Sometimes we want to make the computer forget one of the table entries it knows. We can easily delete a node in which either `LLINK` or `RLINK` $= \Lambda$; but when both subtrees are nonempty, we have to do something special, since we can't point two ways at once.

For example, consider Fig. 10 again; how could we delete the root node, `CAPRICORN`? One solution is to delete the alphabetically *next* node, which always has a null `LLINK`, then reinsert it in place of the node we really wanted to delete. For example, in Fig. 10 we could delete `GEMINI`, then replace `CAPRICORN` by `GEMINI`. This operation preserves the essential left-to-right order of the table entries. The following algorithm gives a detailed description of such a deletion process.

Algorithm D (*Tree deletion*). Let `Q` be a variable that points to a node of a binary search tree represented as in Algorithm T. This algorithm deletes that node, leaving a binary search tree. (In practice, we will have either `Q` $\equiv$ `ROOT` or `Q` $\equiv$ `LLINK(P)` or `RLINK(P)` in some node of the tree. This algorithm resets the value of `Q` in memory, to reflect the deletion.)

D1. [Is `RLINK` null?] Set `T` $\leftarrow$ `Q`. If `RLINK(T)` $= \Lambda$, set `Q` $\leftarrow$ `LLINK(T)` and go to D4. (For example, if `Q` $\equiv$ `RLINK(P)` for some `P`, we would set `RLINK(P)` $\leftarrow$ `LLINK(T)`.)

D2. [Find successor.] Set `R` $\leftarrow$ `RLINK(T)`. If `LLINK(R)` $= \Lambda$, set `LLINK(R)` $\leftarrow$ `LLINK(T)`, `Q` $\leftarrow$ `R`, and go to D4.

D3. [Find null `LLINK`.] Set `S` $\leftarrow$ `LLINK(R)`. Then if `LLINK(S)` $\neq \Lambda$, set `R` $\leftarrow$ `S` and repeat this step until `LLINK(S)` $= \Lambda$. (At this point `S` will be equal to `Q$`, the symmetric successor of `Q`.) Finally, set `LLINK(S)` $\leftarrow$ `LLINK(T)`, `LLINK(R)` $\leftarrow$ `RLINK(S)`, `RLINK(S)` $\leftarrow$ `RLINK(T)`, `Q` $\leftarrow$ `S`.

D4. [Free the node.] Set `AVAIL` $\Leftarrow$ `T`, thus returning the deleted node to the free storage pool. ▌

The reader may wish to try this algorithm by deleting `AQUARIUS`, `CANCER`, and `CAPRICORN` from Fig. 10; each case is slightly different. An alert reader may have noticed that no special test has been made for the case `RLINK(T)` $\neq \Lambda$, `LLINK(T)` $= \Lambda$; we will defer the discussion of this case until later, since the algorithm as it stands has some very interesting properties.

Since Algorithm D is quite unsymmetrical between left and right, it stands to reason that a sequence of deletions will make the tree get way out of balance, so that the efficiency estimates we have made will be invalid. But deletions don't actually make the trees degenerate at all!

Theorem H (T. N. Hibbard, 1962). *After a random element is deleted from a random tree by Algorithm D, the resulting tree is still random.*

[Nonmathematical readers, please skip to (10).] This statement of the theorem is admittedly quite vague. We can summarize the situation more precisely as follows: Let T be a tree of n elements, and let $P(T)$ be the probability that T occurs if its keys are inserted in random order by Algorithm T. Some trees are more probable than others. Let $Q(T)$ be the probability that T will occur if $n+1$ elements are inserted in random order by Algorithm T and then one of these elements is chosen at random and deleted by Algorithm D. In calculating $P(T)$, we assume that the $n!$ permutations of the keys are equally likely; in calculating

$Q(T)$, we assume that the $(n+1)!\,(n+1)$ permutations of keys and selections of the doomed key are equally likely. The theorem states that $P(T) = Q(T)$ for all T.

Proof. We are faced with the fact that permutations are equally probable, not trees, and therefore we shall prove the result by considering *permutations* as the random objects. We shall define a deletion from a permutation, and then we will prove that "a random element deleted from a random permutation leaves a random permutation."

Let $a_1\,a_2 \ldots a_{n+1}$ be a permutation of $\{1, 2, \ldots, n{+}1\}$; we want to define the operation of deleting a_i, so as to obtain a permutation $b_1\,b_2 \ldots b_n$ of $\{1, 2, \ldots, n\}$. This operation should correspond to Algorithms T and D, so that if we start with the tree constructed from the sequence of insertions $a_1, a_2, \ldots, a_{n+1}$ and delete a_i, renumbering the keys from 1 to n, we obtain the tree constructed from $b_1\,b_2 \ldots b_n$.

It is not hard to define such a deletion operation. There are two cases:

Case 1: $a_i = n + 1$, or $a_i + 1 = a_j$ for some $j < i$. (This is essentially the condition "$\mathtt{RLINK}(a_i) = \Lambda$.") Remove a_i from the sequence, and subtract unity from each element greater than a_i.

Case 2: $a_i + 1 = a_j$ for some $j > i$. Replace a_i by a_j, remove a_j from its original place, and subtract unity from each element greater than a_i.

For example, suppose we have the permutation 4 6 1 3 5 2. If we circle the element to be deleted, we have

$$\begin{array}{llll}
\textcircled{4}\;6\;1\;3\;5\;2 = 4\;5\;1\;3\;2 & & 4\;6\;1\;\textcircled{3}\;5\;2 = 3\;5\;1\;4\;2 \\
4\;\textcircled{6}\;1\;3\;5\;2 = 4\;1\;3\;5\;2 & & 4\;6\;1\;3\;\textcircled{5}\;2 = 4\;5\;1\;3\;2 \\
4\;6\;\textcircled{1}\;3\;5\;2 = 3\;5\;1\;2\;4 & & 4\;6\;1\;3\;5\;\textcircled{2} = 3\;5\;1\;2\;4
\end{array}$$

Since there are $(n+1)!\,(n+1)$ possible deletion operations, the theorem will be established if we can show that every permutation of $\{1, 2, \ldots, n\}$ is the result of exactly $(n+1)^2$ deletions.

Let $b_1\,b_2 \ldots b_n$ be a permutation of $\{1, 2, \ldots, n\}$. We shall define $(n+1)^2$ deletions, one for each pair i, j with $1 \le i, j \le n+1$, as follows:

If $i < j$, the deletion is

$$b'_1 \ldots b'_{i-1}\,\textcircled{b_i}\,b'_{i+1} \ldots b'_{j-1}\,(b_i{+}1)\,b'_j \ldots b'_n. \tag{7}$$

Here, as below, b'_k stands for either b_k or $b_k + 1$, depending on whether or not b_k is less than the circled element. This deletion corresponds to Case 2.

If $i > j$, the deletion is

$$b'_1 \ldots b'_{i-1}\,\textcircled{b_j}\,b'_i \ldots b'_n; \tag{8}$$

this deletion fits the definition of Case 1.

Finally, if $i = j$, we have another Case 1 deletion, namely

$$b'_1 \ldots b'_{i-1}\,\textcircled{$n{+}1$}\,b'_i \ldots b'_n. \tag{9}$$

As an example, let $n = 4$ and consider the 25 deletions that map into 3 1 4 2:

	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$
$j=1$	⑤3 1 4 2	4③1 5 2	4 1③5 2	4 1 5③2	4 1 5 2③
$j=2$	③4 1 5 2	3⑤1 4 2	4 2①5 3	4 2 5①3	4 2 5 3①
$j=3$	③1 4 5 2	4①2 5 3	3 1⑤4 2	3 1 5④2	3 1 5 2④
$j=4$	③1 5 4 2	4①5 2 3	3 1④5 2	3 1 4⑤2	4 1 5 3②
$j=5$	③1 5 2 4	4①5 3 2	3 1④2 5	4 1 5②3	3 1 4 2⑤

The circled element is always in position i, and for fixed i we have constructed $n+1$ different deletions, one for each j; hence $(n+1)^2$ different deletions have been constructed for each permutation $b_1 b_2 \ldots b_n$. Since only $(n+1)^2 n!$ deletions are possible, we must have found all of them. ∎

The proof of Theorem H not only tells us about the result of deletions, it also helps us analyze the running time in an average deletion. Exercise 12 shows that we can expect to execute step D2 slightly less than half the time, on the average, when deleting a random element from a random table.

Let us now consider how often the loop in step D3 needs to be performed: Suppose that we are deleting a node on level l, and that the *external* node immediately following in symmetric order is on level k. For example, if we are deleting CAPRICORN from Fig. 10, we have $l = 0$ and $k = 3$ since node $\boxed{4}$ is on level 3. If $k = l + 1$, we have RLINK(T) $= \Lambda$ in step D1; and if $k > l + 1$, we will set S $\leftarrow$ LLINK(R) exactly $k - l - 2$ times in step D3. The average value of l is (internal path length)$/N$; the average value of k is

$$(\text{external path length} - \text{distance to leftmost external node})/N.$$

The distance to the leftmost external node is the number of left-to-right minima in the insertion sequence, so it has the average value H_N by the analysis of Section 1.2.10. Since external path length minus internal path length is $2N$, the average value of $k - l - 2$ is $-H_N/N$. Adding to this the average number of times that $k - l - 2$ is -1, we see that *the operation S $\leftarrow$ LLINK(R) in step D3 is performed only*

$$\tfrac{1}{2} + \bigl(\tfrac{1}{2} - H_N\bigr)/N \qquad (10)$$

times, on the average, in a random deletion. This is reassuring, since the worst case can be pretty slow (see exercise 11).

Although Theorem H is rigorously true, in the precise form we have stated it, it *cannot* be applied, as we might expect, to a sequence of deletions followed by insertions. The shape of the tree is random after deletions, but the relative distribution of values in a given tree shape may change, and it turns out that the first random insertion after deletion actually *destroys* the randomness property on the shapes. This startling fact, first observed by Gary Knott in 1972, must be seen to be believed (see exercise 15). Even more startling is the empirical evidence gathered by J. L. Eppinger [*CACM* **26** (1983), 663–669, **27** (1984), 235], who found that the path length decreases slightly when a few random deletions and insertions are made, but then it *increases* until reaching a steady state after about n^2 deletion/insertion operations have been performed. This steady state is *worse* than the behavior of a random tree, when N is greater than about 150. Further study by Culberson and Munro [*Comp. J.* **32** (1989), 68–75; *Algorithmica* **5** (1990), 295–311] has led to a plausible conjecture that the average search time in the steady state is asymptotically $\sqrt{2N/9\pi}$. However, Eppinger also devised a simple modification that alternates between Algorithm D and a left-right reflection of the same algorithm; he found that this leads to an excellent steady state in which the path length is reduced to about 88% of its normal value for random trees. A theoretical explanation for this behavior is still lacking.

As mentioned above, Algorithm D does not test for the case LLINK(T) $= \Lambda$, although this is one of the easy cases for deletion. We could add a new step between D1 and D2, namely,

D1$\frac{1}{2}$. [Is LLINK null?] If LLINK(T) $= \Lambda$, set Q $\leftarrow$ RLINK(T) and go to D4.

Exercise 14 shows that Algorithm D with this extra step always leaves a tree that is at least as good as the original Algorithm D, in the path-length sense, and sometimes the result is even better. When this idea is combined with Eppinger's symmetric deletion strategy, the steady-state path length for repeated random deletion/insertion operations decreases to about 86% of its insertion-only value.

Frequency of access. So far we have assumed that each key was equally likely as a search argument. In a more general situation, let p_k be the probability that we will search for the kth element inserted, where $p_1 + \cdots + p_N = 1$. Then a straightforward modification of Eq. (2), if we retain the assumption of random order so that the shape of the tree stays random and Eq. (5) holds, shows that the average number of comparisons in a successful search will be

$$1 + \sum_{k=1}^{N} p_k(2H_k - 2) = 2\sum_{k=1}^{N} p_k H_k - 1. \qquad (11)$$

For example, if the probabilities obey Zipf's law, Eq. 6.1–(8), the average number of comparisons reduces to

$$H_N - 1 + H_N^{(2)}/H_N \qquad (12)$$

if we insert the keys in decreasing order of importance. (See exercise 18.) This is about half as many comparisons as predicted by the equal-frequency analysis, and it is fewer than we would make using binary search.

Fig. 12 shows the tree that results when the most common 31 words of English are entered in decreasing order of frequency. The relative frequency is shown with each word, using statistics from *Cryptanalysis* by H. F. Gaines (New York: Dover, 1956), 226. The average number of comparisons for a successful search in this tree is 4.042; the corresponding binary search, using Algorithm 6.2.1B or 6.2.1C, would require 4.393 comparisons.

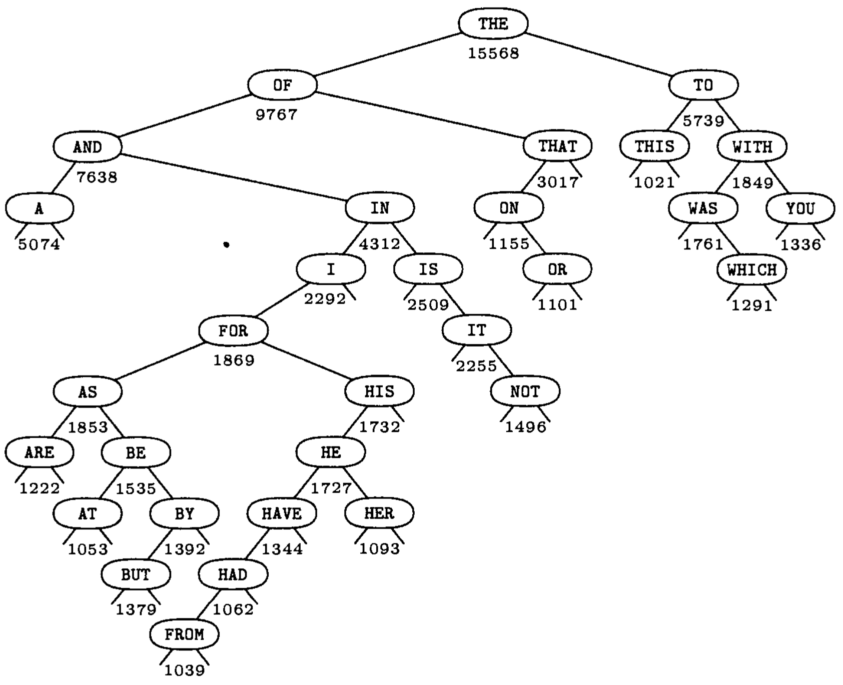
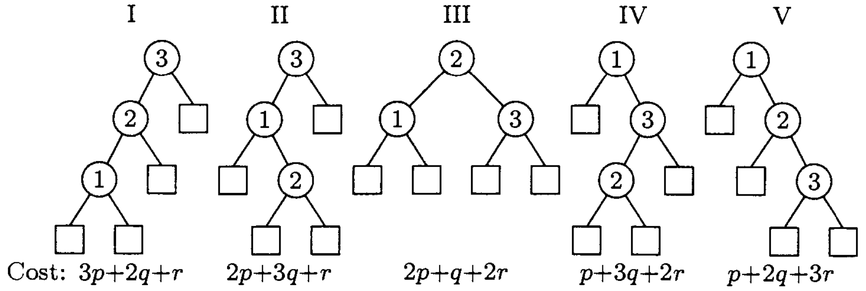


Fig. 12. The 31 most common English words, inserted in decreasing order of frequency.

Optimum binary search trees. These considerations make it natural to ask about the best possible tree for searching a table of keys with given frequencies. For example, the optimum tree for the 31 most common English words is shown in Fig. 13; it requires only 3.437 comparisons for an average successful search.

Let us now explore the problem of finding the optimum tree. When $N = 3$, for example, let us assume that the keys $K_1 < K_2 < K_3$ have respective probabilities p, q, r. There are five possible trees:



(13)

Figure 14 shows the ranges of p, q, r for which each tree is optimum; the balanced tree is best about 45 percent of the time, if we choose p, q, r at random (see exercise 21).

Unfortunately, when N is large there are

$$\binom{2N}{N}\bigg/(N+1) \approx 4^N/(\sqrt{\pi}\,N^{3/2})$$

binary trees, so we can't just try them all and see which is best. Let us therefore study the properties of optimum binary search trees more closely, in order to discover a better way to find them.

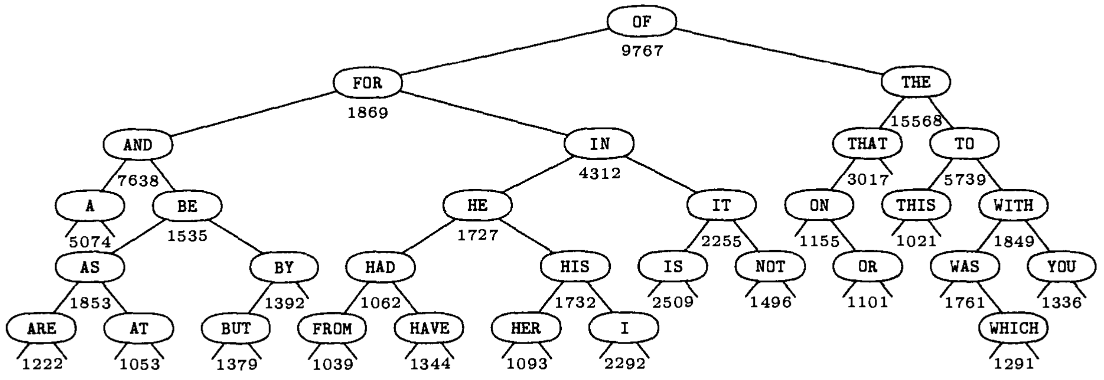


Fig. 13. An optimum search tree for the 31 most common English words.

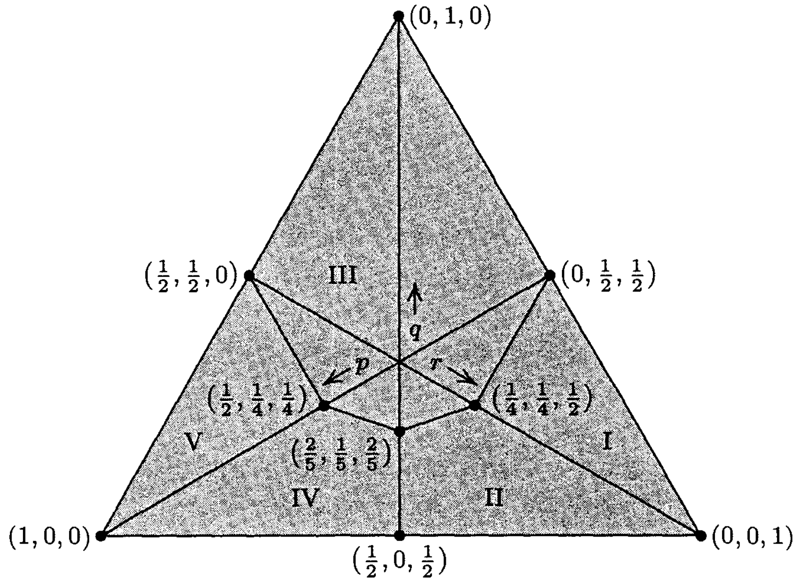


Fig. 14. If the relative frequencies of (K_1, K_2, K_3) are (p, q, r), this graph shows which of the five trees in (13) is best. The fact that $p + q + r = 1$ makes the graph two-dimensional although there are three coordinates.

So far we have considered only the probabilities for a successful search; in practice, the unsuccessful case must usually be considered as well. For example, the 31 words in Fig. 13 account for only about 36 percent of typical English text; the other 64 percent will certainly influence the structure of the optimum search tree.

Therefore let us set the problem up in the following way: We are given $2n+1$ probabilities $p_1, p_2, \ldots, p_n$ and $q_0, q_1, \ldots, q_n$, where

$$p_i = \text{probability that } K_i \text{ is the search argument;}$$

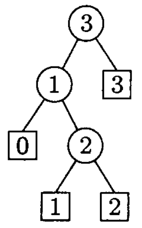
$$q_i = \text{probability that the search argument lies between } K_i \text{ and } K_{i+1}.$$

(By convention, q_0 is the probability that the search argument is less than K_1, and q_n is the probability that the search argument is greater than K_n.) Thus,

$p_1 + p_2 + \cdots + p_n + q_0 + q_1 + \cdots + q_n = 1$, and we want to find a binary tree that minimizes the expected number of comparisons in the search, namely

$$\sum_{j=1}^{n} p_j\bigl(\text{level}(\text{Ⓙ}) + 1\bigr) + \sum_{k=0}^{n} q_k \,\text{level}(\boxed{k}), \tag{14}$$

where Ⓙ is the jth internal node in symmetric order and $\boxed{k}$ is the $(k+1)$st external node, and where the root has level zero. Thus the expected number of comparisons for the binary tree


(15)

is $2q_0 + 2p_1 + 3q_1 + 3p_2 + 3q_2 + p_3 + q_3$. Let us call this the *cost* of the tree; and let us say that a minimum-cost tree is *optimum.* In this definition there is no need to require that the p's and q's sum to unity; we can ask for a minimum-cost tree with any given sequence of "weights" $(p_1, \ldots, p_n;\ q_0, \ldots, q_n)$.

We have studied Huffman's procedure for constructing trees with minimum weighted path length, in Section 2.3.4.5; but that method requires all the p's to be zero, and the tree it produces will usually not have the external node weights $(q_0, \ldots, q_n)$ in the proper symmetric order from left to right. Therefore we need another approach.

What saves us is that *all subtrees of an optimum tree are optimum.* For example, if (15) is an optimum tree for the weights $(p_1, p_2, p_3;\ q_0, q_1, q_2, q_3)$, then the left subtree of the root must be optimum for $(p_1, p_2;\ q_0, q_1, q_2)$; any improvement to a subtree leads to an improvement in the whole tree.

This principle suggests a computation procedure that systematically finds larger and larger optimum subtrees. We have used much the same idea in Section 5.4.9 to construct optimum merge patterns; the general approach is known as "dynamic programming," and we shall consider it further in Section 7.7.

Let $c(i,j)$ be the cost of an optimum subtree with weights $(p_{i+1}, \ldots, p_j;\ q_i, \ldots, q_j)$; and let $w(i,j) = p_{i+1} + \cdots + p_j + q_i + \cdots + q_j$ be the sum of all those weights; thus $c(i,j)$ and $w(i,j)$ are defined for $0 \le i \le j \le n$. It follows that

$$\begin{aligned} &c(i,i) = 0,\\ &c(i,j) = w(i,j) + \min_{i<k\le j}\bigl(c(i,k{-}1) + c(k,j)\bigr), \qquad \text{for } i < j, \end{aligned} \tag{16}$$

since the minimum possible cost of a tree with root Ⓚ is $w(i,j) + c(i,k{-}1) + c(k,j)$. When $i < j$, let $R(i,j)$ be the set of all k for which the minimum is achieved in (16); this set specifies the possible roots of the optimum trees.

Equation (16) makes it possible to evaluate $c(i,j)$ for $j - i = 1, 2, \ldots, n$; there are about $\frac{1}{2}n^2$ such values, and the minimization operation is carried out for about $\frac{1}{6}n^3$ values of k. This means we can determine an optimum tree in $O(n^3)$ units of time, using $O(n^2)$ cells of memory.

A factor of n can actually be removed from the running time if we make use of a monotonicity property. Let $r(i,j)$ denote an element of $R(i,j)$; we need not compute the entire set $R(i,j)$, a single representative is sufficient. Once we have found $r(i,j-1)$ and $r(i+1,j)$, the result of exercise 27 proves that we may always assume that

$$r(i,j-1) \le r(i,j) \le r(i+1,j) \tag{17}$$

when the weights are nonnegative. This limits the search for the minimum, since only $r(i+1,j) - r(i,j-1) + 1$ values of k need to be examined in (16) instead of $j-i$. The total amount of work when $j-i = d$ is now bounded by the telescoping series

$$\sum_{\substack{d\le j\le n\\ i=j-d}} \bigl(r(i+1,j) - r(i,j-1) + 1\bigr) = r(n-d+1,n) - r(0,d-1) + n - d + 1 < 2n;$$

hence the total running time is reduced to $O(n^2)$.

The following algorithm describes this procedure in detail.

Algorithm K (*Find optimum binary search trees*). Given $2n+1$ nonnegative weights $(p_1,\ldots,p_n;\ q_0,\ldots,q_n)$, this algorithm constructs binary trees $t(i,j)$ that have minimum cost for the weights $(p_{i+1},\ldots,p_j;\ q_i,\ldots,q_j)$ in the sense defined above. Three arrays are computed, namely

$c[i,j]$,	for $0 \le i \le j \le n$,	the cost of $t(i,j)$;
$r[i,j]$,	for $0 \le i < j \le n$,	the root of $t(i,j)$;
$w[i,j]$,	for $0 \le i \le j \le n$,	the total weight of $t(i,j)$.

The results of the algorithm are specified by the r array: If $i=j$, $t(i,j)$ is null; otherwise its left subtree is $t(i,\, r[i,j]-1)$ and its right subtree is $t(r[i,j],\, j)$.

K1. [Initialize.] For $0 \le i \le n$, set $c[i,i] \leftarrow 0$ and $w[i,i] \leftarrow q_i$ and $w[i,j] \leftarrow w[i,j-1] + p_j + q_j$ for $j = i+1,\ldots,n$. Then for $1 \le j \le n$ set $c[j-1,j] \leftarrow w[j-1,j]$ and $r[j-1,j] \leftarrow j$. (This determines all the 1-node optimum trees.)

K2. [Loop on d.] Do step K3 for $d = 2,3,\ldots,n$, then terminate the algorithm.

K3. [Loop on j.] (We have already determined the optimum trees of fewer than d nodes. This step determines all the d-node optimum trees.) Do step K4 for $j = d,\ d+1,\ \ldots,\ n$.

K4. [Find $c[i,j]$, $r[i,j]$.] Set $i \leftarrow j-d$. Then set

$$c[i,j] \leftarrow w[i,j] + \min\nolimits_{r[i,j-1]\le k\le r[i+1,j]}\bigl(c[i,k-1] + c[k,j]\bigr),$$

and set $r[i,j]$ to a value of k for which the minimum occurs. (Exercise 22 proves that $r[i,j-1] \le r[i+1,j]$.) ▌

As an example of Algorithm K, consider Fig. 15, which is based on a "key-word-in-context" (KWIC) indexing application. The titles of all articles in the first ten volumes of the *Journal of the ACM* were sorted to prepare a concordance in which there was one line for every word of every title. However, certain words like "THE" and "EQUATION" were felt to be sufficiently uninformative that they were left out of the index. These special words and their frequency of occurrence are shown in the internal nodes of Fig. 15. Notice that a title such as "On the solution of an equation for a certain new problem" would be so uninformative, it wouldn't appear in the index at all! The idea of KWIC indexing is due to H. P. Luhn, *Amer. Documentation* **11** (1960), 288–295. (See W. W. Youden, *JACM* **10** (1963), 583–646, where the full KWIC index appears.)

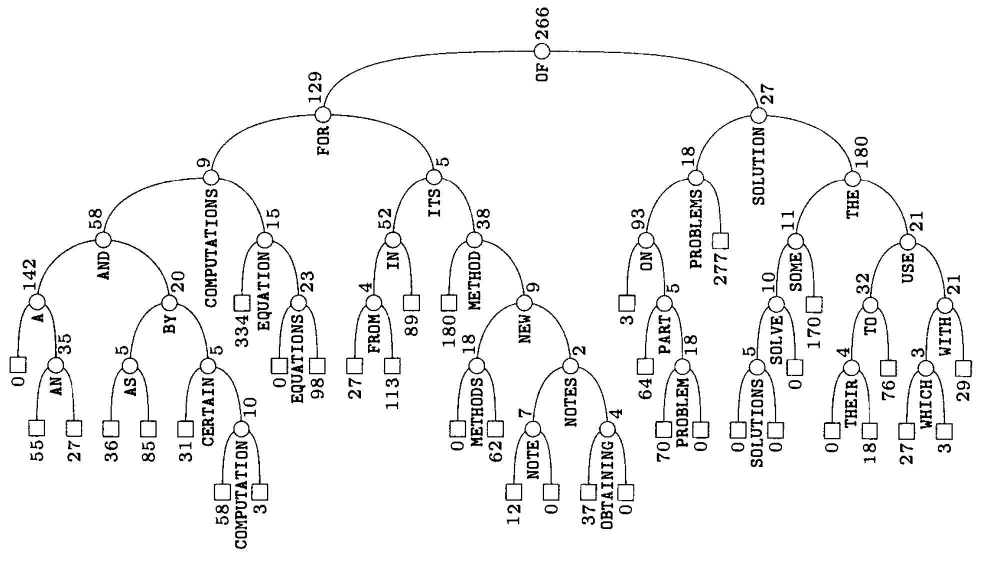


Fig. 15. An optimum binary search tree for a KWIC indexing application.

When preparing a KWIC index file for sorting, we might want to use a binary search tree in order to test whether or not each particular word is to be indexed. The other words fall between two of the unindexed words, with the frequencies shown in the external nodes of Fig. 15; thus, exactly 277 words that are alphabetically between "PROBLEMS" and "SOLUTION" appeared in the *JACM* titles during 1954–1963.

Figure 15 shows the optimum tree obtained by Algorithm K, with $n = 35$. The computed values of $r[0,j]$ for $j = 1, 2, \ldots, 35$ are $(1,1,2,3,3,3,3,8,8,8,$ $8,8,8,11,11,\ldots,11,21,21,21,21,21,21)$; the values of $r[i,35]$ for $i = 0,1,\ldots,34$ are $(21,21,\ldots,21,25,25,25,25,25,25,26,26,26,30,30,30,30,30,30,30,33,33,$ $33,35,35)$.

The "betweenness frequencies" q_j have a noticeable effect on the optimum tree structure; Fig. 16(a) shows the optimum tree that would have been obtained with the q_j set to zero. Similarly, the internal frequencies p_i are important; Fig. 16(b) shows the optimum tree when the p_i are set to zero. Considering the full set of frequencies, the tree of Fig. 15 requires only 4.15 comparisons, on the average, while the trees of Fig. 16 require, respectively, 4.69 and 4.55.

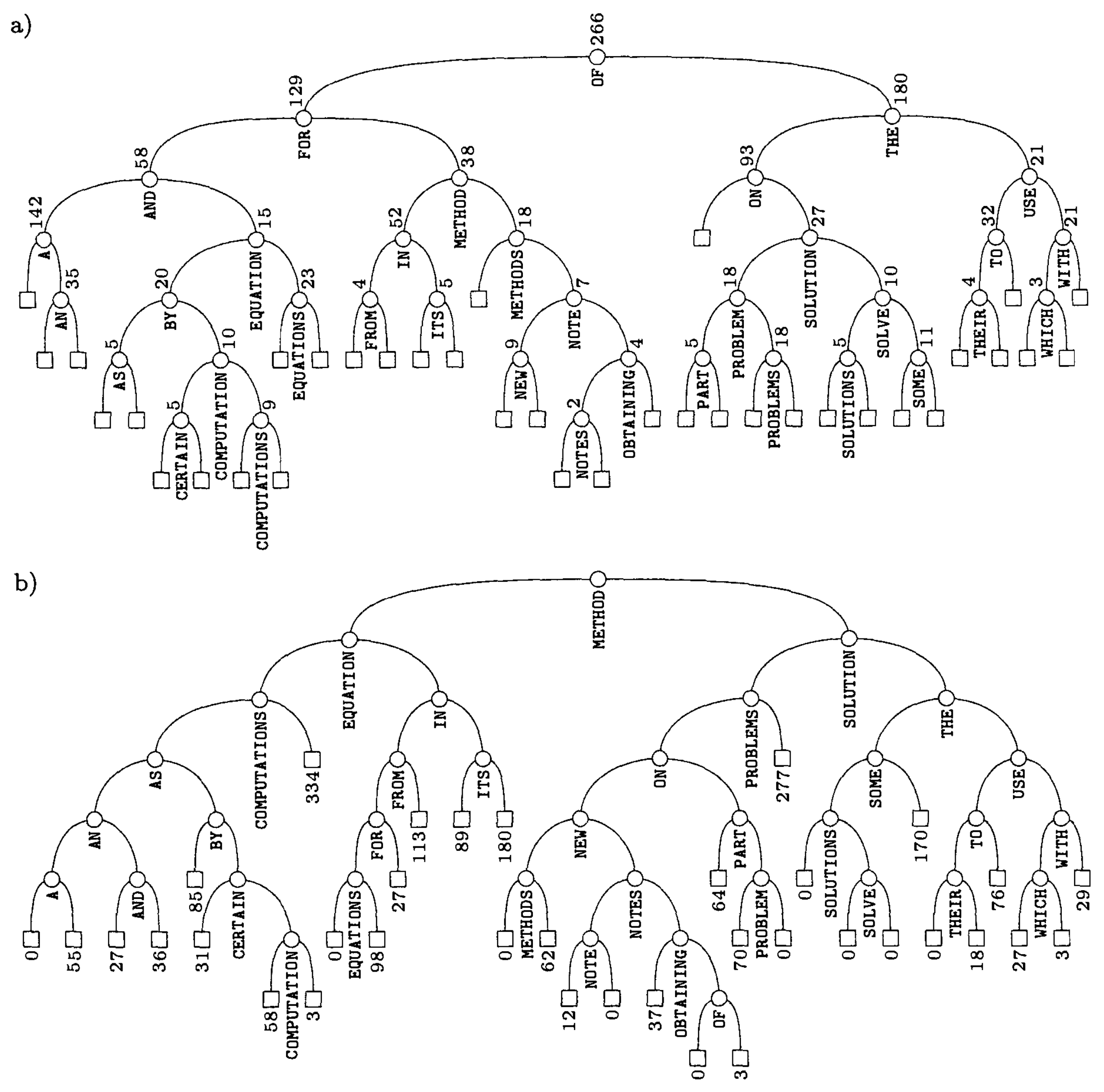


Fig. 16. Optimum binary search trees based on half of the data of Fig. 15: (a) external frequencies suppressed; (b) internal frequencies suppressed.

Since Algorithm K requires time and space proportional to n^2, it becomes impractical when n is very large. Of course we may not really want to use binary search trees for large n, in view of the other search techniques to be discussed later in this chapter; but let's assume anyway that we want to find an optimum or nearly optimum tree when n is large.

We have seen that the idea of inserting the keys in order of decreasing frequency can tend to make a fairly good tree, on the average; but it can also be very bad (see exercise 20), and it is not usually very near the optimum, since it makes no use of the q_j weights. Another approach is to choose the root k so that the resulting maximum subtree weight, $\max\bigl(w(0,k-1),\ w(k,n)\bigr)$, is as small as possible. This approach can also be fairly poor, because it may choose a node with very small p_k to be the root; however, Theorem M below shows that the resulting tree will not be extremely far from the optimum.

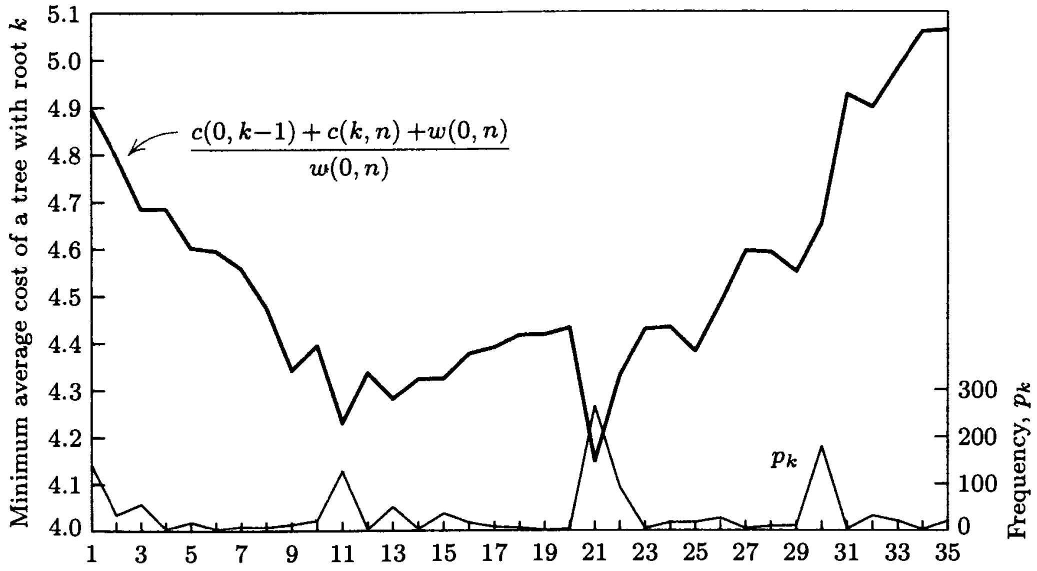


Fig. 17. Behavior of the cost as a function of the root, k.

A more satisfactory procedure can be obtained by combining these two methods, as suggested by W. A. Walker and C. C. Gotlieb [*Graph Theory and Computing* (Academic Press, 1972), 303–323]: Try to equalize the left-hand and right-hand weights, but be prepared to move the root a few steps to the left or right to find a node with relatively large p_k. Figure 17 shows why this method is reasonable: If we plot $c(0,k-1)+c(k,n)$ as a function of k, for the KWIC data of Fig. 15, we see that the result is quite sensitive to the magnitude of p_k.

A top-down method such as this can be used for large n to choose the root and then to work on the left and the right subtrees. When we get down to a sufficiently small subtree we can apply Algorithm K. The resulting method yields fairly good trees (reportedly within 2 or 3 percent of the optimum), and it requires only $O(n)$ units of space, $O(n\log n)$ units of time. In fact, M. Fredman has shown that $O(n)$ units of time suffice, if suitable data structures are used [*STOC* **7** (1975), 240–244]; see K. Mehlhorn, *Data Structures and Algorithms* **1** (Springer, 1984), Section 4.2.

Optimum trees and entropy. The minimum cost is closely related to a mathematical concept called *entropy*, which was introduced by Claude Shannon in his seminal work on information theory [*Bell System Tech. J.* **27** (1948), 379–423, 623–656]. If $p_1, p_2, \ldots, p_n$ are probabilities with $p_1+p_2+\cdots+p_n=1$, we define the entropy $H(p_1,p_2,\ldots,p_n)$ by the formula

$$H(p_1,p_2,\ldots,p_n) = \sum_{k=1}^{n} p_k \lg\frac{1}{p_k}. \tag{18}$$

Intuitively, if n events are possible and the kth event occurs with probability p_k, we can imagine that we have received $\lg(1/p_k)$ bits of information when the kth event has occurred. (An event of probability $\frac{1}{32}$ gives 5 bits of information, etc.) Then $H(p_1,p_2,\ldots,p_n)$ is the expected number of bits of information in a random event. If $p_k = 0$, we define $p_k \lg(1/p_k) = 0$, because

$$\lim_{\epsilon\to 0+} \epsilon \lg \frac{1}{\epsilon} = \lim_{m\to\infty} \frac{1}{m} \lg m = 0.$$

This convention allows us to use (18) when some of the probabilities are zero.

The function $x\lg(1/x)$ is concave; that is, its second derivative, $-1/(x\ln 2)$, is negative. Therefore the maximum value of $H(p_1,p_2,\ldots,p_n)$ occurs when $p_1 = p_2 = \cdots = p_n = 1/n$, namely

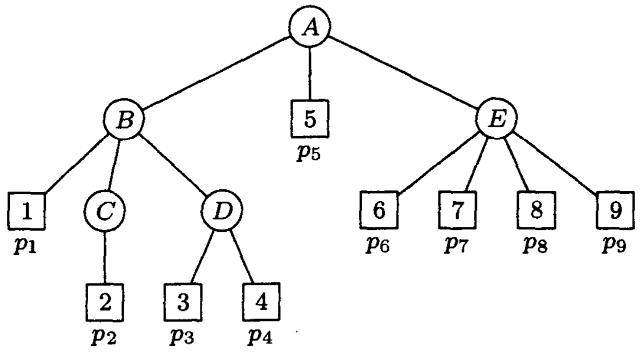
$$H\Bigl(\frac{1}{n},\frac{1}{n},\ldots,\frac{1}{n}\Bigr) = \lg n. \tag{19}$$

In general, if we specify p_1, …, p_{n-k} but allow the other probabilities p_{n-k+1}, …, p_n to vary, we have

$$\begin{aligned} H(p_1,\ldots,p_{n-k},p_{n-k+1},\ldots,p_n) &\le H\Bigl(p_1,\ldots,p_{n-k},\frac{q}{k},\ldots,\frac{q}{k}\Bigr)\\ &= H(p_1,\ldots,p_{n-k},q) + q\lg k, \qquad (20)\\ H(p_1,\ldots,p_{n-k},p_{n-k+1},\ldots,p_n) &\ge H(p_1,\ldots,p_{n-k},q,0,\ldots,0)\\ &= H(p_1,\ldots,p_{n-k},q), \qquad (21)\end{aligned}$$

where $q = 1 - (p_1 + \cdots + p_{n-k})$.

Consider any not-necessarily-binary tree in which probabilities have been assigned to the leaves, say



(22)

Here p_k represents the probability that a search procedure will end at leaf $\boxed{k}$. Then the branching at each internal (nonleaf) node corresponds to a local probability distribution based on the sums of leaf probabilities below each branch. For example, at node Ⓐ the first, second, and third branches are taken with the respective probabilities

$$(p_1 + p_2 + p_3 + p_4,\ p_5,\ p_6 + p_7 + p_8 + p_9),$$

and at node Ⓑ the probabilities are

$$(p_1, p_2, p_3 + p_4)/(p_1 + p_2 + p_3 + p_4).$$

Let us say that each internal node has the entropy of its local probability distribution; thus

$$\begin{aligned}
H(A) &= (p_1+p_2+p_3+p_4)\lg\frac{1}{p_1+p_2+p_3+p_4}\\
&\qquad + p_5\lg\frac{1}{p_5} + (p_6+p_7+p_8+p_9)\lg\frac{1}{p_6+p_7+p_8+p_9},\\
H(B) &= \frac{p_1}{p_1+p_2+p_3+p_4}\lg\frac{p_1+p_2+p_3+p_4}{p_1} + \frac{p_2}{p_1+p_2+p_3+p_4}\lg\frac{p_1+p_2+p_3+p_4}{p_2}\\
&\qquad + \frac{p_3+p_4}{p_1+p_2+p_3+p_4}\lg\frac{p_1+p_2+p_3+p_4}{p_3+p_4},\\
H(C) &= \frac{p_2}{p_2}\lg\frac{p_2}{p_2},\\
H(D) &= \frac{p_3}{p_3+p_4}\lg\frac{p_3+p_4}{p_3} + \frac{p_4}{p_3+p_4}\lg\frac{p_3+p_4}{p_4},\\
H(E) &= \frac{p_6}{p_6+p_7+p_8+p_9}\lg\frac{p_6+p_7+p_8+p_9}{p_6} + \frac{p_7}{p_6+p_7+p_8+p_9}\lg\frac{p_6+p_7+p_8+p_9}{p_7}\\
&\qquad + \frac{p_8}{p_6+p_7+p_8+p_9}\lg\frac{p_6+p_7+p_8+p_9}{p_8} + \frac{p_9}{p_6+p_7+p_8+p_9}\lg\frac{p_6+p_7+p_8+p_9}{p_9}.
\end{aligned}$$

Lemma E. *The sum of $p(\alpha)H(\alpha)$ over all internal nodes α of a tree, where $p(\alpha)$ is the probability of reaching node α and $H(\alpha)$ is the entropy of α, equals the entropy of the probability distribution on the leaves.*

Proof. It is easy to establish this identity by induction from bottom to top. For example, we have

$$\begin{aligned}
H(A)+(p_1+p_2+p_3+p_4)H(B)+p_2H(C)+(p_3+p_4)H(D)+(p_6+p_7+p_8+p_9)H(E)\\
= p_1\lg\frac{1}{p_1}+p_2\lg\frac{1}{p_2}+\cdots+p_9\lg\frac{1}{p_9}
\end{aligned}$$

with respect to the formulas above; all terms involving $\lg(p_1+p_2+p_3+p_4)$, $\lg(p_3+p_4)$, and $\lg(p_6+p_7+p_8+p_9)$ cancel out. ∎

As a consequence of Lemma E, we can use entropy to establish a convenient lower bound on the cost of any binary tree.

Theorem B. *Let $(p_1,\ldots,p_n;q_0,\ldots,q_n)$ be nonnegative weights as in Algorithm K, normalized so that $p_1+\cdots+p_n+q_0+\cdots+q_n=1$, and let $P=p_1+\cdots+p_n$ be the probability of a successful search. Let*

$$H = H(p_1,\ldots,p_n,q_0,\ldots,q_n)$$

be the entropy of the corresponding probability distribution, and let C be the minimum cost, (14). Then if $H \ge 2P/e$ we have

$$C \ge H - P\lg\frac{eH}{2P}. \qquad (23)$$

Proof. Take a binary tree of cost C and assign the probabilities q_k to its leaves. Also add a middle branch below each internal node, leading to a new leaf that has probability p_k. Then $C = \sum p(\alpha)$, summed over the internal nodes α of the resulting ternary tree, and $H = \sum p(\alpha)H(\alpha)$ by Lemma E.

The entropy $H(\alpha)$ corresponds to a three-way distribution, where one of the probabilities is $p_j/p(\alpha)$ if α is internal node $\textcircled{j}$. Exercise 35 proves that

$$H(p,q,r) \le p\lg x + 1 + \lg\Bigl(1 + \frac{1}{2x}\Bigr) \tag{24}$$

for all $x > 0$, whenever $p + q + r = 1$. Therefore we have the inequality

$$H = \sum_{\alpha} p(\alpha)H(\alpha) \;\le\; \sum_{j=1}^{n} p_j \lg x \;+\; \Bigl(1 + \lg\Bigl(1 + \frac{1}{2x}\Bigr)\Bigr)C$$

for all positive x. Choosing $2x = H/P$ now leads to the desired result, since

$$\begin{aligned} C &\ge \frac{1}{1+\lg(1+P/H)}\Bigl(H - P\lg\frac{H}{2P}\Bigr) \\ &= \frac{1}{1+\lg(1+P/H)}(H + P\lg e) - \frac{P}{1+\lg(1+P/H)}\lg\frac{eH}{2P} \\ &\ge H - P\lg\frac{eH}{2P}, \end{aligned}$$

because $\lg(1+y) \le y\lg e$ for all $y > 0$. ▌

Eq. (23) does not necessarily hold when the entropy is extremely low. But the restriction to cases where $H \ge 2P/e$ is not severe, since the value of H is usually near $\lg n$; see exercise 37. Notice that the proof doesn't actually use the left-to-right order of the nodes; the lower bound (23) holds for any binary search tree that has internal node probabilities p_j and external node probabilities q_k in any order.

Entropy calculations also yield an upper bound that is not too far from (23), even when we do stick to the left-to-right order:

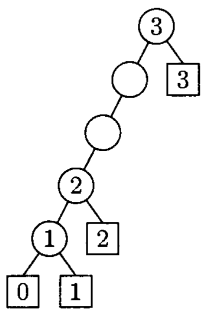
Theorem M. *Under the assumptions of Theorem B, we also have*

$$C \;<\; H + 2 - P. \tag{25}$$

Proof. Form the $n+1$ sums $s_0 = \frac{1}{2}q_0$, $s_1 = q_0+p_1+\frac{1}{2}q_1$, $s_2 = q_0+p_1+q_1+p_2+\frac{1}{2}q_2$, …, $s_n = q_0+p_1+\cdots+q_{n-1}+p_n+\frac{1}{2}q_n$; we may assume that $s_0 < s_1 < \cdots < s_n$ (see exercise 38). Express each s_k as a binary fraction, writing $s_n = (.111\ldots)_2$ if $s_n = 1$. Then let the string σ_k be the leading bits of s_k, retaining just enough bits to distinguish s_k from s_j for $j \ne k$. For example, we might have $n = 3$ and

$s_0 = (.0000001)_2$	$\sigma_0 = 00000$
$s_1 = (.0000101)_2$	$\sigma_1 = 00001$
$s_2 = (.0001011)_2$	$\sigma_2 = 0001$
$s_3 = (.1100000)_2$	$\sigma_3 = 1$

Construct a binary tree with $n+1$ leaves, in such a way that σ_k corresponds to the path from the root to $\boxed{k}$ for $0 \le k \le n$, where '0' denotes a left branch and '1' denotes a right branch. Also, if σ_{k-1} has the form $\alpha_k 0 \beta_k$ and σ_k has the form $\alpha_k 1 \gamma_k$ for some α_k, β_k, and γ_k, let the internal node ⓚ correspond to the path α_k. Thus we would have



in the example above. There may be some internal nodes that are still nameless; replace each of them by their one and only child. The cost of the resulting tree is at most $\sum_{k=1}^{n} p_k(|\alpha_k|+1) + \sum_{k=0}^{n} q_k|\sigma_k|$.

We have

$$p_k \le \tfrac{1}{2}q_{k-1} + p_k + \tfrac{1}{2}q_k = s_k - s_{k-1} \le 2^{-|\alpha_k|}, \tag{26}$$

because $s_k \le (.\alpha_k)_2 + 2^{-|\alpha_k|}$ and $s_{k-1} \ge (.\alpha_k)_2$. Furthermore, if $q_k \ge 2^{-t}$ we have $s_k \ge s_{k-1} + 2^{-t-1}$ and $s_{k+1} \ge s_k + 2^{-t-1}$, hence $|\sigma_k| \le t+1$. It follows that $q_k < 2^{-|\sigma_k|+2}$, and we have constructed a binary tree of cost

$$\le \sum_{k=1}^{n} p_k(1+|\alpha_k|) + \sum_{k=0}^{n} q_k|\sigma_k| \le \sum_{k=1}^{n} p_k\Bigl(1+\lg\frac{1}{p_k}\Bigr) + \sum_{k=0}^{n} q_k\Bigl(2+\lg\frac{1}{q_k}\Bigr)$$
$$= P + 2(1-P) + H = H + 2 - P. \quad ▌$$

In the KWIC indexing application of Fig. 15, we have $P = 1304/3288 \approx 0.39659$, and $H(p_1,\ldots,p_{35},q_0,\ldots,q_{35}) \approx 5.00635$. Therefore Theorem B tells us that $C \ge 3.3800$, and Theorem M tells us that $C < 6.6098$.

***The Garsia–Wachs algorithm.** An amazing improvement on Algorithm K is possible in the special case that $p_1 = \cdots = p_n = 0$. This case, in which only the leaf probabilities $(q_0, q_1, \ldots, q_n)$ are relevant, is especially important because it arises in a several significant applications. Let us therefore assume in the remainder of this section that the probabilities p_j are zero. Notice that Theorems B and M reduce to the inequalities

$$H(q_0,q_1,\ldots,q_n) \le C(q_0,q_1,\ldots,q_n) < H(q_0,q_1,\ldots,q_n) + 2 \tag{27}$$

in this case; and the cost function (14) simplifies to

$$C = \sum_{k=0}^{n} q_k l_k, \qquad l_k = \text{the level of } \boxed{k}. \tag{28}$$

The key property that makes a simpler algorithm possible is the following observation:

Lemma W. *If $q_{k-1} > q_{k+1}$ then $l_k \le l_{k+1}$ in every optimum tree. If $q_{k-1} = q_{k+1}$ then $l_k \le l_{k+1}$ in some optimum tree.*

Proof. Suppose $q_{k-1} \ge q_{k+1}$ and consider a tree in which $l_k > l_{k+1}$. Then $\boxed{k}$ must be a right child, and its left sibling L is a subtree of cost $c \ge q_{k-1}$. Replace the parent of $\boxed{k}$ by L; replace $\boxed{k{+}1}$ by a node whose children are $\boxed{k}$ and $\boxed{k{+}1}$. This changes the overall cost by $-c - q_k(l_k - l_{k+1} - 1) + q_{k+1} \le q_{k+1} - q_{k-1}$. So the given tree was not optimum if $q_{k-1} > q_{k+1}$, and it has been transformed into another optimum tree if $q_{k-1} = q_{k+1}$. In the latter case a sequence of such transformations will make $l_k \le l_{k+1}$. ▌

A deeper analysis of the structure tells us considerably more.

Lemma X. *Suppose j and k are indices such that $j < k$ and we have*

i) $q_{i-1} > q_{i+1}$ *for* $1 \le i < k$;
ii) $q_{k-1} \le q_{k+1}$;
iii) $q_i < q_{k-1} + q_k$ *for* $j \le i < k-1$; *and*
iv) $q_{j-1} \ge q_{k-1} + q_k$.

Then there is an optimum tree in which $l_{k-1} = l_k$ and either

a) $l_j = l_k - 1$, *or*
b) $l_j = l_k$ *and* $\boxed{j}$ *is a left child.*

Proof. By reversing left and right in Lemma W, we see that (ii) implies the existence of an optimum tree in which $l_{k-1} \ge l_k$. But Lemma W and (i) also imply that $l_1 \le l_2 \le \cdots \le l_k$. Therefore $l_{k-1} = l_k$.

Suppose $l_s < l_k - 1 \le l_{s+1}$ for some s with $j \le s < k$. Let t be the smallest index $< k$ such that $l_t = l_k$. Then $l_{s+1} = \cdots = l_{t-1} = l_k - 1$, and $\boxed{s{+}1}$ is a left child; hence $t-s$ is odd, and node $\boxed{i}$ is a left child for $i = s+1, s+3, \ldots, t$. Replace the parent of $\boxed{t}$ by $\boxed{t{+}1}$; replace $\boxed{i}$ by $\boxed{i{+}1}$ for $s < i < t$; and replace the external node $\boxed{s}$ by an internal node whose children are $\boxed{s}$ and $\boxed{s{+}1}$. This changes the cost by $\le q_s - q_t - q_{t+1} \le q_s - q_{k-1} - q_k$, so it is an improvement if $q_s < q_{k-1} + q_k$. Therefore, by (iii), $l_j \ge l_k - 1$.

We still have not used hypothesis (iv). If $l_j = l_k$ and $\boxed{j}$ is not a left child, $\boxed{j}$ must be the right sibling of $\boxed{j{+}1}$. Replace their parent by $\boxed{j{+}1}$; then replace leaf $\boxed{i}$ by $\boxed{i{+}1}$ for $j < i < k$; and replace the external node $\boxed{k}$ by an internal node whose children are $\boxed{k{+}1}$ and $\boxed{k}$. This changes the cost by $-q_{j-1} + q_{k-1} + q_k \le 0$, so we obtain an optimum tree satisfying (b). ▌

Lemma Y. *Let j and k be as in Lemma X, and consider the modified probabilities $(q'_0, \ldots, q'_{n-1}) = (q_0, \ldots, q_{j-1}, q_{k-1} + q_k, q_j, \ldots, q_{k-2}, q_{k+1}, \ldots, q_n)$ obtained by removing q_{k-1} and q_k and inserting $q_{k-1} + q_k$ after q_{j-1}. Then*

$$C(q'_0, \ldots, q'_{n-1}) \le (q_{k-1} + q_k) + C(q_0, \ldots, q_n). \tag{29}$$

Proof. It suffices to show that any optimum tree for $(q_0, \ldots, q_n)$ can be transformed into a tree of the same cost in which $\boxed{k{-}1}$ and $\boxed{k}$ are siblings and the leaves appear in permuted order

$$\boxed{0} \quad \boxed{j{-}1} \quad \boxed{k{-}1} \quad \boxed{k} \quad \boxed{j} \quad \ldots \quad \boxed{k{-}2} \quad \boxed{k{+}1} \quad \ldots \quad \boxed{n}. \tag{30}$$

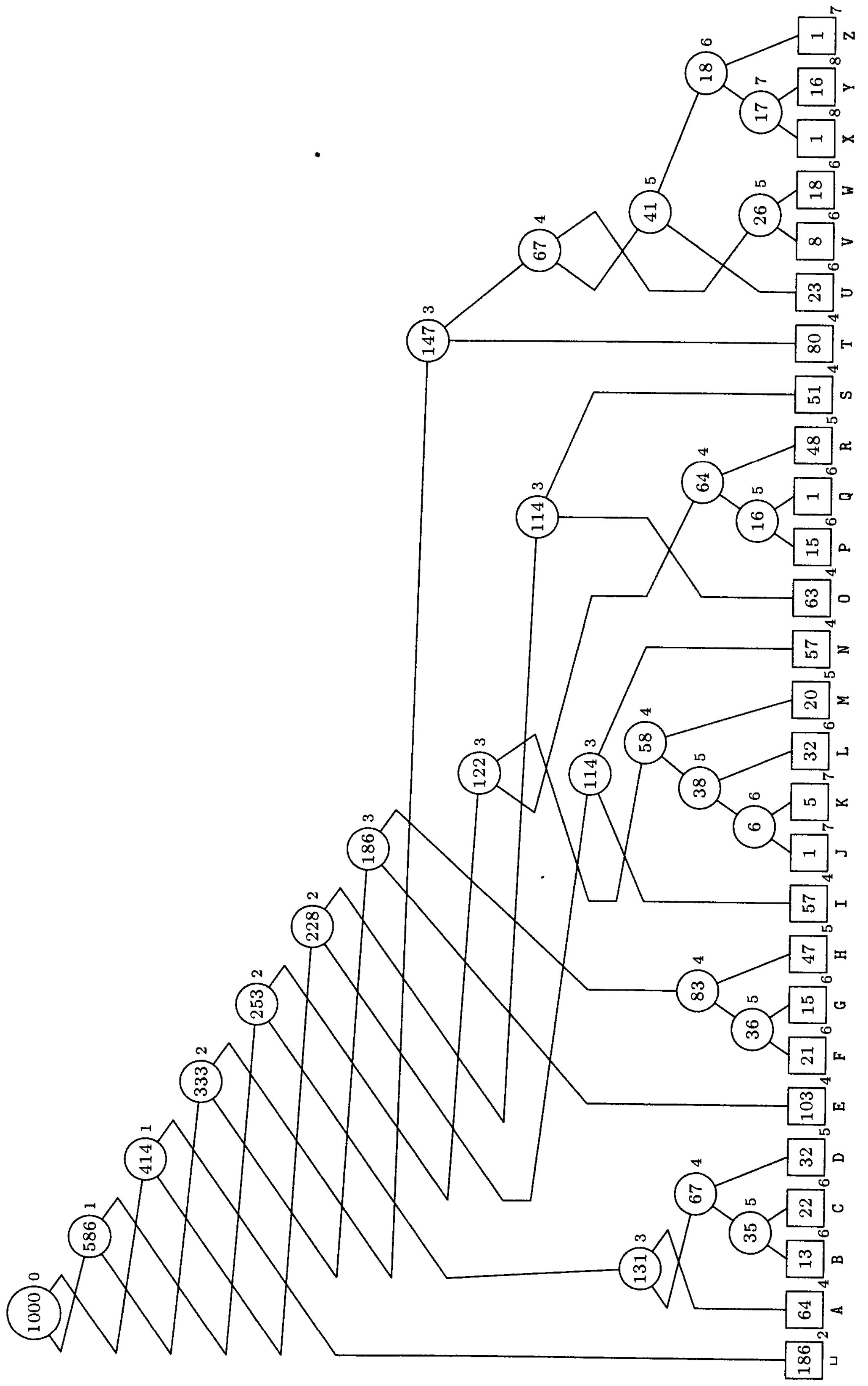


Fig. 18. The Garsia–Wachs algorithm applied to alphabetic frequency data: Phases 1 and 2.

We start with the tree constructed in Lemma X. If it is of type (b), we simply rename the leaves, sliding $\boxed{k-1}$ and $\boxed{k}$ to the left by $k-1-j$ places. If it is of type (a), suppose $l_{s-1} = l_k - 1$ and $l_s = l_k$; we proceed as follows: First slide $\boxed{k-1}$ and $\boxed{k}$ left by $k-1-s$ places; then replace their (new) parent by $\boxed{k-2}$; finally replace $\boxed{j}$ by a node whose children are $\boxed{k-1}$ and $\boxed{k}$, and replace node $\boxed{i}$ by $\boxed{i-1}$ for $j < i < k-1$. ▌

Lemma Z. *Under the hypotheses of Lemma Y, equality holds in* (29).

Proof. Every tree for $(q'_0, \ldots, q'_{n-1})$ corresponds to a tree with leaves (30) in which the two out-of-order leaf nodes $\boxed{k-1}$ and $\boxed{k}$ are siblings. Let internal node $\textcircled{x}$ be their parent. We want to show that any optimum tree of that type can be converted to a tree of the same cost in which the leaves appear in normal order $\boxed{0}$ ... $\boxed{n}$.

There is nothing to prove if $j = k-1$. Otherwise we have $q'_{i-1} > q'_{i+1}$ for $j \le i < k-1$, because $q_{j-1} \ge q_{k-1} + q_k > q_j$. Therefore by Lemma W we have $l_x \le l_j \le \cdots \le l_{k-2}$, where l_x is the level of $\textcircled{x}$ and l_i is the level of $\boxed{i}$ for $j \le i < k-1$. If $l_x = l_{k-2}$, we simply slide node $\textcircled{x}$ to the right, replacing the sequence $\textcircled{x}$ $\boxed{j}$ ... $\boxed{k-2}$ by $\boxed{j}$ ... $\boxed{k-2}$ $\textcircled{x}$; this straightens out the leaves as desired.

Otherwise suppose $l_s = l_x$ and $l_{s+1} > l_x$. We first replace $\textcircled{x}$ $\boxed{j}$ ... $\boxed{s}$ by $\boxed{j}$ ... $\boxed{s}$ $\textcircled{x}$; this makes $l \le l_{s+1} \le \cdots \le l_{k-2}$, where $l = l_x + 1$ is the common level of nodes $\boxed{k-1}$ and $\boxed{k}$. Finally replace nodes

$$\boxed{k-1}\ \boxed{k}\ \boxed{s+1}\ \ldots\ \boxed{k-2}$$

by the cyclically shifted sequence

$$\boxed{s+1}\ \ldots\ \boxed{k-2}\ \boxed{k-1}\ \boxed{k}.$$

Exercise 40 proves that this decreases the cost, unless $l_{k-2} = l$. But the cost cannot decrease, because of Lemma Y. Therefore $l_{k-2} = l$, and the proof is complete. ▌

These lemmas show that the problem for $n+1$ weights $q_0, q_1, \ldots, q_n$ can be reduced to an n-weight problem: We first find the smallest index k with $q_{k-1} \le q_{k+1}$; then we find the largest $j < k$ with $q_{j-1} \ge q_{k-1} + q_k$; then we remove q_{k-1} and q_k from the list, and insert the sum $q_{k-1} + q_k$ just after q_{j-1}. In the special cases $j = 0$ or $k = n$, the proofs show that we should proceed as if infinite weights q_{-1} and q_{n+1} were present at the left and right. The proofs also show that any optimum tree T' that is obtained from the new weights $(q'_0, \ldots, q'_{n-1})$ can be rearranged into a tree T that has the original weights $(q_0, \ldots, q_n)$ in the correct left-to-right order; moreover, each weight will appear at the same level in both T and T'.

For example, Fig. 18 illustrates the construction when the weights q_k are the relative frequencies of the characters ␣, `A`, `B`, ..., `Z` in English text. The first few weights are

$$186,\ 64,\ 13,\ 22,\ 32,\ 103,\ \ldots$$

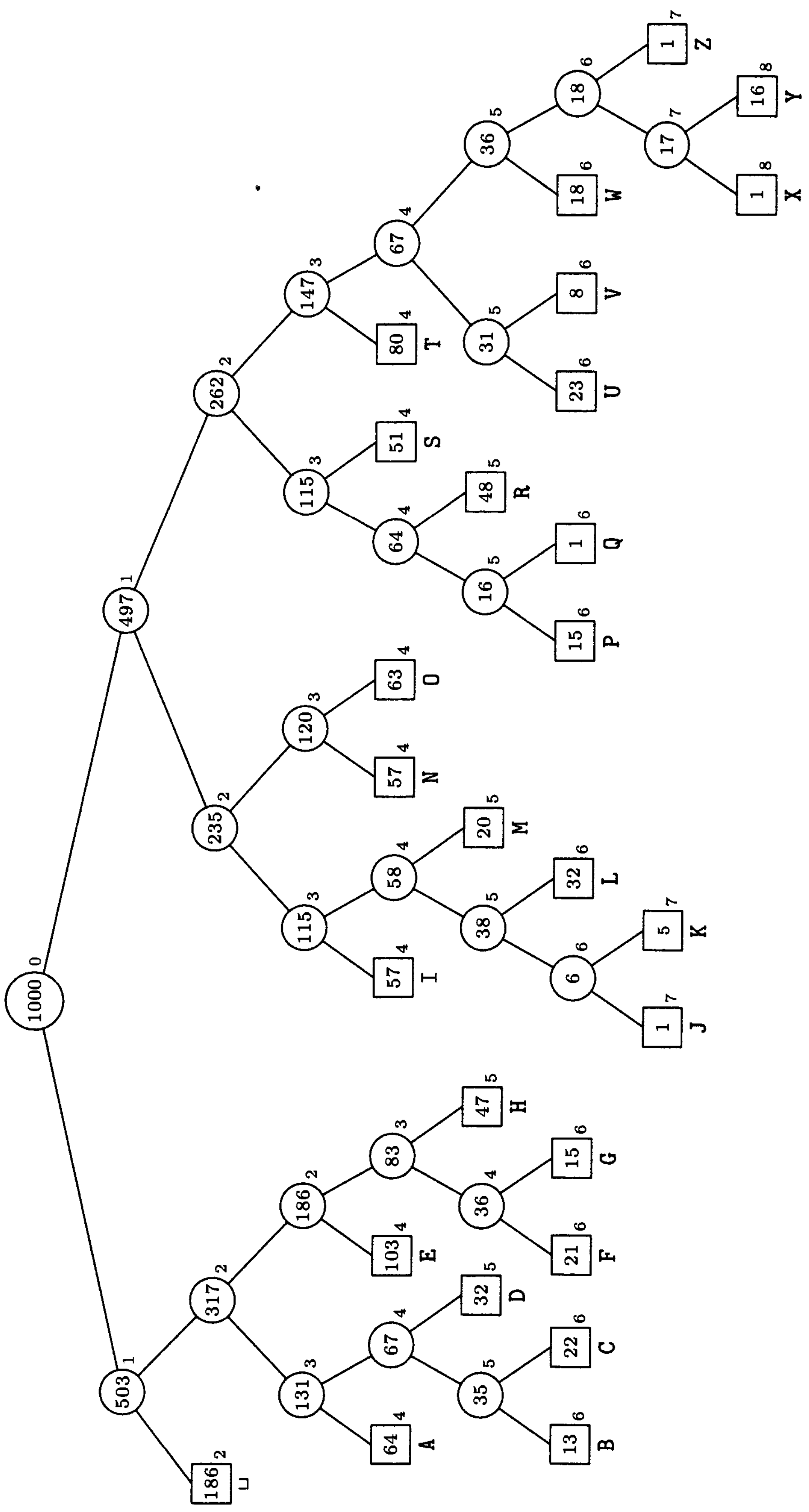


Fig. 19. The Garsia–Wachs algorithm applied to alphabetic frequency data: Phase 3.

and we have $186 > 13$, $64 > 22$, $13 \le 32$; therefore we replace "13, 22" by 35. In the new sequence

$$186,\ 64,\ 35,\ 32,\ 103,\ \ldots$$

we replace "35, 32" by 67 and slide 67 to the left of 64, obtaining

$$186,\ 67,\ 64,\ 103,\ \ldots.$$

Then "67, 64" becomes 131, and we begin to examine the weights that follow 103. After the 27 original weights have been combined into the single weight 1000, the history of successive combinations specifies a binary tree whose weighted path length is the solution to the original problem.

But the leaves of the tree in Fig. 18 are not at all in the correct order, because they get tangled up when we slide $q_{k-1}+q_k$ to the left (see exercise 41). Still, the proof of Lemma Z guarantees that there is a tree whose leaves are in the correct order and on exactly the same levels as in the tangled tree. This untangled tree, Fig. 19, is therefore optimum; it is the binary tree output by the Garsia–Wachs algorithm.

Algorithm G (*Garsia–Wachs algorithm for optimum binary trees*). Given a sequence of nonnegative weights $w_0, w_1, \ldots, w_n$, this algorithm constructs a binary tree with n internal nodes for which $\sum_{k=0}^{n} w_k l_k$ is minimum, where l_k is the distance of external node $\boxed{k}$ from the root. It uses an array of $2n+2$ nodes whose addresses are $\mathtt{X}_k$ for $0 \le k \le 2n+1$; each node has four fields called `WT`, `LLINK`, `RLINK`, and `LEVEL`. The leaves of the constructed tree will be nodes $\mathtt{X}_0 \ldots \mathtt{X}_n$; the internal nodes will be $\mathtt{X}_{n+1} \ldots \mathtt{X}_{2n}$; the root will be $\mathtt{X}_{2n}$; and $\mathtt{X}_{2n+1}$ is used as a temporary sentinel. The algorithm also maintains a working array of pointers $\mathtt{P}_0, \mathtt{P}_1, \ldots, \mathtt{P}_t$, where $t \le n+1$.

G1. [Begin phase 1.] Set $\mathtt{WT}(\mathtt{X}_k) \leftarrow w_k$ and $\mathtt{LLINK}(\mathtt{X}_k) \leftarrow \mathtt{RLINK}(\mathtt{X}_k) \leftarrow \Lambda$ for $0 \le k \le n$. Also set $\mathtt{P}_0 \leftarrow \mathtt{X}_{2n+1}$, $\mathtt{WT}(\mathtt{P}_0) \leftarrow \infty$, $\mathtt{P}_1 \leftarrow \mathtt{X}_0$, $t \leftarrow 1$, $m \leftarrow n$. Then perform step G2 for $j = 1, 2, \ldots, n$, and go to G3.

G2. [Absorb w_j.] (At this point we have the basic condition

$$\mathtt{WT}(\mathtt{P}_{i-1}) > \mathtt{WT}(\mathtt{P}_{i+1}) \qquad \text{for } 1 \le i < t; \tag{31}$$

in other words, the weights in the working array are "2-descending.") Perform Subroutine C below, zero or more times, until $\mathtt{WT}(\mathtt{P}_{t-1}) > w_j$. Then set $t \leftarrow t+1$ and $\mathtt{P}_t \leftarrow \mathtt{X}_j$.

G3. [Finish phase 1.] Perform Subroutine C zero or more times, until $t = 1$.

G4. [Do phase 2.] (Now $\mathtt{P}_1 = \mathtt{X}_{2n}$ is the root of a binary tree, and $\mathtt{WT}(\mathtt{P}_1) = w_0+\cdots+w_n$.) Set l_k to the distance of node $\mathtt{X}_k$ from node $\mathtt{P}_1$, for $0 \le k \le n$. (See exercise 43. An example is shown in Fig. 18, where level numbers appear at the right of each node.)

G5. [Do phase 3.] By changing the links of $\mathtt{X}_{n+1}, \ldots, \mathtt{X}_{2n}$, construct a new binary tree having the same level numbers l_k, but with the leaf nodes in symmetric order $\mathtt{X}_0, \ldots, \mathtt{X}_n$. (See exercise 44; an example appears in Fig. 19.) ▌

Subroutine C (*Combination*). This subroutine is the heart of the Garsia–Wachs algorithm. It combines two weights, shifts them left as appropriate, and maintains the 2-descending condition (31).

C1. [Initialize.] Set $k \leftarrow t$.

C2. [Create a new node.] .(At this point we have $k \geq 2$.) Set $m \leftarrow m+1$, $\mathtt{LLINK(X}_m\mathtt{)} \leftarrow \mathtt{P}_{k-1}$, $\mathtt{RLINK(X}_m\mathtt{)} \leftarrow \mathtt{P}_k$, $\mathtt{WT(X}_m\mathtt{)} \leftarrow \mathtt{WT(P}_{k-1}\mathtt{)} + \mathtt{WT(P}_k\mathtt{)}$.

C3. [Shift the following nodes left.] Set $t \leftarrow t-1$, then $\mathtt{P}_{j+1} \leftarrow \mathtt{P}_j$ for $k \leq j \leq t$.

C4. [Shift the preceding nodes right.] Set $j \leftarrow k-2$; then while $\mathtt{WT(P}_j\mathtt{)} < \mathtt{WT(X}_m\mathtt{)}$ set $\mathtt{P}_{j+1} \leftarrow \mathtt{P}_j$ and $j \leftarrow j-1$.

C5. [Insert the new node.] Set $\mathtt{P}_{j+1} \leftarrow \mathtt{X}_m$.

C6. [Done?] If $j > 0$ and $\mathtt{WT(P}_{j-1}\mathtt{)} \leq \mathtt{WT(X}_m\mathtt{)}$, set $k \leftarrow j$ and return to C2. ▌

As stated, Subroutine C might need $\Omega(n)$ steps to create and insert a new node, because it uses sequential memory instead of linked lists. Therefore the total running time of Algorithm G might be $\Omega(n^2)$. But more elaborate data structures can be used to guarantee that phase 1 will require at most $O(n \log n)$ steps (see exercise 45). Phases 2 and 3 need only $O(n)$ steps.

Kleitman and Saks [*SIAM J. Algeb. Discr. Methods* **2** (1981), 142–146] proved that the optimum weighted path length never exceeds the value of the optimum weighted path length that occurs when the q's have been rearranged in "sawtooth order":

$$q_0 \leq q_2 \leq q_4 \leq \cdots \leq q_{2\lfloor n/2 \rfloor} \leq q_{2\lceil n/2 \rceil - 1} \leq \cdots \leq q_3 \leq q_1 . \qquad (32)$$

(This is the inverse of the organ-pipe order discussed in exercise 6.1–18.) In the latter case the Garsia–Wachs algorithm essentially reduces to Huffman's algorithm on the weights q_0+q_1, q_2+q_3, ..., because the weights in the working array will actually be nonincreasing (not merely "2-descending" as in (31)). Therefore we can improve the upper bound of Theorem M without knowing the order of the weights.

The optimum binary tree in Fig. 19 has an important application to coding theory as well as to searching: Using 0 to stand for a left branch in the tree and 1 to stand for a right branch, we obtain the following variable-length codewords:

␣	00	I	1000	R	11001	
A	0100	J	1001000	S	1101	
B	010100	K	1001001	T	1110	
C	010101	L	100101	U	111100	
D	01011	M	10011	V	111101	(33)
E	0110	N	1010	W	111110	
F	011100	O	1011	X	11111100	
G	011101	P	110000	Y	11111101	
H	01111	Q	110001	Z	1111111	

Thus a message like "`RIGHT ON`" would be encoded by the string

$$1100110000111010111111100010111010.$$

Decoding from left to right is easy, in spite of the variable length of the codewords, because the tree structure tells us when one codeword ends and another begins. This method of coding preserves the alphabetical order of messages, and it uses an average of about 4.2 bits per letter. Thus the code could be used to compress data files, without destroying lexicographic order of alphabetic information. (The figure of 4.2 bits per letter is minimum over all binary tree codes, although it could be reduced to 4.1 bits per letter if we disregarded the alphabetic ordering constraint. A further reduction, preserving alphabetic order, could be achieved if pairs of letters instead of single letters were encoded.)

History and bibliography. The tree search methods of this section were discovered independently by several people during the 1950s. In an unpublished memorandum dated August 1952, A. I. Dumey described a primitive form of tree insertion in the following way:

> Consider a drum with 2^n item storages in it, each having a binary address.
>
> Follow this program:
>
> 1. Read in the first item and store it in address 2^{n-1}, i.e., at the halfway storage place.
> 2. Read in the next item. Compare it with the first.
> 3. If it is larger, put it in address $2^{n-1} + 2^{n-2}$. It it is smaller, put it at 2^{n-2}. ...

Another early form of tree insertion was introduced by D. J. Wheeler, who actually allowed multiway branching similar to what we shall discuss in Section 6.2.4; and a binary tree insertion technique was devised by C. M. Berners-Lee [see *Comp. J.* **2** (1959), 5].

The first published descriptions of tree insertion were by P. F. Windley [*Comp. J.* **3** (1960), 84–88], A. D. Booth and A. J. T. Colin [*Information and Control* **3** (1960), 327–334], and Thomas N. Hibbard [*JACM* **9** (1962), 13–28]. Each of these authors seems to have developed the method independently of the others, and each paper derived the average number of comparisons (6) in a different way. The individual authors also went on to treat different aspects of the algorithm: Windley gave a detailed discussion of tree insertion sorting; Booth and Colin discussed the effect of preconditioning by making the first 2^n-1 elements form a perfectly balanced tree (see exercise 4); Hibbard introduced the idea of deletion and showed the connection between the analysis of tree insertion and the analysis of quicksort.

The idea of *optimum* binary search trees was first developed for the special case $p_1 = \cdots = p_n = 0$, in the context of alphabetic binary encodings like (33). A very interesting paper by E. N. Gilbert and E. F. Moore [*Bell System Tech. J.* **38** (1959), 933–968] discussed this problem and its relation to other coding problems. Gilbert and Moore proved Theorem M in the special case $P = 0$, and observed that an optimum tree could be constructed in $O(n^3)$ steps, using a method like Algorithm K but without making use of the monotonicity relation (17). K. E. Iverson [*A Programming Language* (Wiley, 1962), 142–144] independently considered the *other* case, when all the q's are zero. He suggested that an optimum tree would be obtained if the root is chosen so as to equalize the left and right subtree probabilities as much as possible; unfortunately we have seen that this idea doesn't work. D. E. Knuth [*Acta Informatica* **1** (1971), 14–25, 270] subsequently considered the case of general p and q weights and proved that the algorithm could be reduced to $O(n^2)$ steps; he also presented an example from a compiler application, where the keys in the tree are "reserved words" in an ALGOL-like language. T. C. Hu had been studying his own algorithm for the case $p_j = 0$ for several years; a rigorous proof of the validity of that algorithm was difficult to find because of the complexity of the problem, but he eventually obtained a proof jointly with A. C. Tucker [*SIAM J. Applied Math.* **21** (1971), 514–532]. Simplifications leading to Algorithm G were found several years later by A. M. Garsia and M. L. Wachs, *SICOMP* **6** (1977), 622–642, although their proof was still rather complicated. Lemmas W, X, Y, and Z above are due to J. H. Kingston, *J. Algorithms* **9** (1988), 129–136. See also the paper by Hu, Kleitman, and Tamaki, *SIAM J. Applied Math.* **37** (1979), 246–256, for an elementary proof of the Hu–Tucker algorithm and some generalizations to other cost functions.

Theorem B is due to Paul J. Bayer, report MIT/LCS/TM-69 (Mass. Inst. of Tech., 1975), who also proved a slightly weaker form of Theorem M. The stronger form above is due to K. Mehlhorn, *SICOMP* **6** (1977), 235–239.

EXERCISES

1. [*15*] Algorithm T has been stated only for nonempty trees. What changes should be made so that it works properly for the empty tree too?

2. [*20*] Modify Algorithm T so that it works with *right-threaded* trees. (See Section 2.3.1; symmetric traversal is easier in such trees.)

▶ **3.** [*20*] In Section 6.1 we found that a slight change to the sequential search Algorithm 6.1S made it faster (Algorithm 6.1Q). Can a similar trick be used to speed up Algorithm T?

4. [*M24*] (A. D. Booth and A. J. T. Colin.) Given N keys in random order, suppose that we use the first $2^n - 1$ to construct a perfectly balanced tree, placing 2^k keys on level k for $0 \le k < n$; then we use Algorithm T to insert the remaining keys. What is the average number of comparisons in a successful search? [*Hint:* Modify Eq. (2).]

▶ **5.** [*M25*] There are $11! = 39{,}916{,}800$ different orders in which the names `CAPRICORN`, `AQUARIUS`, etc. could have been inserted into a binary search tree.

a) How many of these arrangements will produce Fig. 10?

b) How many of these arrangements will produce a *degenerate* tree, in which `LLINK` or `RLINK` is Λ in each node?

6. [*M26*] Let P_{nk} be the number of permutations $a_1 a_2 \ldots a_n$ of $\{1, 2, \ldots, n\}$ such that, if Algorithm T is used to insert $a_1, a_2, \ldots, a_n$ successively into an initially empty tree, exactly k comparisons are made when a_n is inserted. (In this problem, we will ignore the comparisons made when $a_1, \ldots, a_{n-1}$ were inserted. In the notation of the text, we have $C'_{n-1} = (\sum_k kP_{nk})/n!$, since this is the average number of comparisons made in an unsuccessful search of a tree containing $n-1$ elements.)

a) Prove that $P_{(n+1)k} = 2P_{n(k-1)} + (n-1)P_{nk}$. [*Hint:* Consider whether or not a_{n+1} falls below a_n in the tree.]

b) Find a simple formula for the generating function $G_n(z) = \sum_k P_{nk}z^k$, and use your formula to express P_{nk} in terms of Stirling numbers.

c) What is the *variance* of this distribution?

7. [*M25*] (S. R. Arora and W. T. Dent.) After n elements have been inserted into an initially empty tree, in random order, what is the average number of comparisons needed by Algorithm T to find the mth largest element, given the key of that element?

8. [*M38*] Let $p(n, k)$ be the probability that k is the total internal path length of a tree built by Algorithm T from n randomly ordered keys. (The internal path length is the number of comparisons made by tree insertion sorting as the tree is being built.)

a) Find a recurrence relation that defines the corresponding generating function.

b) Compute the variance of this distribution. [Several of the exercises in Section 1.2.7 may be helpful here.]

9. [*41*] We have proved that tree search and insertion requires only about $2 \ln N$ comparisons when the keys are inserted in random order; but in practice, the order may not be random. Make empirical studies to see how suitable tree insertion really is for symbol tables within a compiler and/or assembler. Do the identifiers used in typical large programs lead to fairly well-balanced binary search trees?

▶ **10.** [*22*] (R. W. Floyd.) Perhaps we are not interested in the sorting property of Algorithm T, but we expect that the input will come in nonrandom order. Devise a way to keep tree search efficient, by making the input "appear to be" in random order.

11. [*20*] What is the maximum number of times the assignment `S ← LLINK(R)` might be performed in step D3, when deleting a node from a tree of size N? empirical data

12. [*M22*] When making a random deletion from a random tree of N items, how often does step D1 go to D4, on the average? (See the proof of Theorem H.)

▶ **13.** [*M23*] If the root of a random tree is deleted by Algorithm D, is the resulting tree still random?

▶ **14.** [*22*] Prove that the path length of the tree produced by Algorithm D with step D$1\frac{1}{2}$ added is never more than the path length of the tree produced without that step. Find a case where step D$1\frac{1}{2}$ actually decreases the path length.

15. [*23*] Let $a_1\,a_2\,a_3\,a_4$ be a permutation of $\{1, 2, 3, 4\}$, and let $j = 1$, 2, or 3. Take the one-element tree with key a_1 and insert a_2, a_3 using Algorithm T; then delete a_j using Algorithm D; then insert a_4 using Algorithm T. How many of the $4! \times 3$ possibilities produce trees of shape I, II, III, IV, V, respectively, in (13)?

▶ **16.** [*25*] Is the deletion operation *commutative*? That is, if Algorithm D is used to delete X and then Y, is the resulting tree the same as if Algorithm D is used to delete Y and then X?

17. [*25*] Show that if the roles of left and right are completely reversed in Algorithm D, it is easy to extend the algorithm so that it deletes a given node from a *right-threaded* tree, preserving the necessary threads. (See exercise 2.)

18. [*M21*] Show that Zipf's law yields (12).

19. [*M23*] What is the approximate average number of comparisons, (11), when the input probabilities satisfy the 80-20 law defined in Eq. 6.1–(11)?

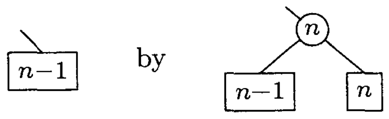
20. [*M20*] Suppose we have inserted keys into a tree in order of decreasing frequency $p_1 \geq p_2 \geq \cdots \geq p_n$. Can this tree be substantially worse than the optimum search tree?

21. [*M20*] If p, q, r are probabilities chosen at random, subject to the condition that $p+q+r=1$, what are the probabilities that trees I, II, III, IV, V of (13) are optimal, respectively? (Consider the relative areas of the regions in Fig. 14.)

22. [*M20*] Prove that $r[i,j-1]$ is never greater than $r[i+1,j]$ when step K4 of Algorithm K is performed.

▸ **23.** [*M23*] Find an optimum binary search tree for the case $N = 40$, with weights $p_1 = 9$, $p_2 = p_3 = \cdots = p_{40} = 1$, $q_0 = q_1 = \cdots = q_{40} = 0$. (Don't use a computer.)

24. [*M25*] Given that $p_n = q_n = 0$ and that the other weights are nonnegative, prove that an optimum tree for $(p_1, \ldots, p_n;\ q_0, \ldots, q_n)$ may be obtained by replacing



in any optimum tree for $(p_1, \ldots, p_{n-1};\ q_0, \ldots, q_{n-1})$.

25. [*M20*] Let A and B be nonempty sets of real numbers, and define $A \leq B$ if the following property holds:

$$(a \in A,\ b \in B, \text{ and } b < a) \qquad \text{implies} \qquad (a \in B \text{ and } b \in A).$$

a) Prove that this relation is transitive on nonempty sets.

b) Prove or disprove: $A \leq B$ if and only if $A \leq A \cup B \leq B$.

26. [*M22*] Let $(p_1, \ldots, p_n;\ q_0, \ldots, q_n)$ be nonnegative weights, where $p_n + q_n = x$. Prove that as x varies from 0 to ∞, while $(p_1, \ldots, p_{n-1};\ q_0, \ldots, q_{n-1})$ are held constant, the cost $c(0,n)$ of an optimum binary search tree is a concave, continuous, piecewise linear function of x with integer slopes. In other words, prove that there exist positive integers $l_0 > l_1 > \cdots > l_m$ and real constants $0 = x_0 < x_1 < \cdots < x_m < x_{m+1} = \infty$ and $y_0 < y_1 \cdots < y_m$ such that $c(0,n) = y_h + l_h x$ when $x_h \leq x \leq x_{h+1}$, for $0 \leq h \leq m$.

27. [*M33*] The object of this exercise is to prove that the sets of roots $R(i,j)$ of optimum binary search trees satisfy

$$R(i,j-1) \leq R(i,j) \leq R(i+1,j), \qquad \text{for } j-i \geq 2,$$

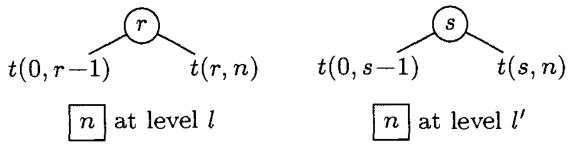
in terms of the relation defined in exercise 25, when the weights $(p_1, \ldots, p_n;\ q_0, \ldots, q_n)$ are nonnegative. The proof is by induction on $j-i$; our task is to prove that $R(0,n-1) \leq R(0,n)$, assuming that $n \geq 2$ and that the stated relation holds for $j-i < n$. [By left-right symmetry it follows that $R(0,n) \leq R(1,n)$.]

a) Prove that $R(0,n-1) \leq R(0,n)$ if $p_n = q_n = 0$. (See exercise 24.)

b) Let $p_n + q_n = x$. In the notation of exercise 26, let R_h be the set $R(0,n)$ of optimum roots when $x_h < x < x_{h+1}$, and let R'_h be the set of optimum roots when $x = x_h$. Prove that

$$R'_0 \leq R_0 \leq R'_1 \leq R_1 \leq \cdots \leq R'_m \leq R_m.$$

Hence by part (a) and exercise 25 we have $R(0,n-1) \le R(0,n)$ for all x. [*Hint:* Consider the case $x = x_h$, and assume that both the trees



are optimum, with $s < r$ and $l \ge l'$. Use the induction hypothesis to prove that there is an optimum tree with root $\textcircled{r}$ such that $\boxed{n}$ is at level l', and an optimum tree with root $\textcircled{s}$ such that $\boxed{n}$ is at level l.]

28. [*24*] Use some macro language to define an "optimum binary search" macro, whose parameter is a nested specification of an optimum binary tree.

29. [*40*] What is the *worst* possible binary search tree for the 31 most common English words, using the frequency data of Fig. 12?

30. [*M34*] Prove that the costs of optimum binary search trees satisfy the "quadrangle inequality" $c(i,j) - c(i,j-1) \ge c(i+1,j) - c(i+1,j-1)$ when $j \ge i+2$.

31. [*M35*] (K. C. Tan.) Prove that, among all possible sets of probabilities $(p_1,\ldots,p_n;$ $q_0,\ldots,q_n)$ with $p_1 + \cdots + p_n + q_0 + \cdots + q_n = 1$, the most expensive minimum-cost tree occurs when $p_i = 0$ for all i, $q_j = 0$ for all even j, and $q_j = 1/\lceil n/2\rceil$ for all odd j.

▸ **32.** [*M25*] Let $n+1 = 2^m + k$, where $0 \le k \le 2^m$. There are exactly $\binom{2^m}{k}$ binary trees in which all external nodes appear on levels m and $m+1$. Show that, among all these trees, we obtain one with the minimum cost for the weights $(p_1,\ldots,p_n;q_0,\ldots,q_n)$ if we apply Algorithm K to the weights $(p_1,\ldots,p_n;M+q_0,\ldots,M+q_n)$ for sufficiently large M.

33. [*M41*] In order to find the binary search tree that minimizes the running time of Program T, we should minimize the quantity $7C + C1$ instead of simply minimizing the number of comparisons C. Develop an algorithm that finds optimum binary search trees when different costs are associated with left and right branches in the tree. (Incidentally, when the right cost is twice the left cost, and the node frequencies are all equal, the Fibonacci trees turn out to be optimum; see L. E. Stanfel, *JACM* **17** (1970), 508–517. On machines that cannot make three-way comparisons at once, a program for Algorithm T will have to make two comparisons in step T2, one for equality and one for less-than; B. Sheil and V. R. Pratt have observed that these comparisons need not involve the same key, and it may well be best to have a binary tree whose internal nodes specify either an equality test *or* a less-than test but not both. This situation would be interesting to explore as an alternative to the stated problem.)

34. [*HM21*] Show that the asymptotic value of the multinomial coefficient

$$\binom{N}{p_1N,\ p_2N,\ \ldots,\ p_nN}$$

as $N \to \infty$ is related to the entropy $H(p_1,p_2,\ldots,p_n)$.

35. [*HM22*] Complete the proof of Theorem B by establishing the inequality (24).

▸ **36.** [*HM25*] (Claude Shannon.) Let X and Y be random variables with finite ranges $\{x_1,\ldots,x_m\}$ and $\{y_1,\ldots,y_n\}$, and let $p_i = \Pr(X = x_i)$, $q_j = \Pr(Y = y_j)$, $r_{ij} = \Pr(X = x_i \text{ and } Y = y_j)$. Let $H(X) = H(p_1,\ldots,p_m)$ and $H(Y) = H(q_1,\ldots,q_n)$ be the respective entropies of the variables singly, and let $H(XY) = H(r_{11}, \ldots, r_{mn})$ be the entropy of their joint distribution. Prove that

$$H(X) \le H(XY) \le H(X) + H(Y).$$

[*Hint:* If f is any concave function, we have $\mathrm{E}\, f(X) \le f(\mathrm{E}\, X)$.]

37. [*HM26*] (P. J. Bayer, 1975.) Suppose $(P_1, \ldots, P_n)$ is a random probability distribution, namely a random point in the $(n-1)$-dimensional simplex defined by $P_k \ge 0$ for $1 \le k \le n$ and $P_1 + \cdots + P_n = 1$. (Equivalently, $(P_1, \ldots, P_n)$ is a set of random *spacings*, in the sense of exercise 3.3.2–26.) What is the expected value of the entropy $H(P_1, \ldots, P_n)$?

38. [*M20*] Explain why Theorem M holds in general, although we have only proved it in the case $s_0 < s_1 < s_2 < \cdots < s_n$.

▶ **39.** [*M25*] Let $w_1, \ldots, w_n$ be nonnegative weights with $w_1 + \cdots + w_n = 1$. Prove that the weighted path length of the Huffman tree constructed in Section 2.3.4.5 is less than $H(w_1, \ldots, w_n) + 1$. *Hint:* See the proof of Theorem M.

40. [*M26*] Complete the proof of Lemma Z.

41. [*21*] Fig. 18 shows the construction of a tangled binary tree. List its leaves in left-to-right order.

42. [*23*] Explain why Subroutine C preserves the 2-descending condition (31).

43. [*20*] Explain how to implement phase 2 of the Garsia–Wachs algorithm efficiently.

▶ **44.** [*25*] Explain how to implement phase 3 of the Garsia–Wachs algorithm efficiently: Construct a binary tree, given the levels $l_0, l_1, \ldots, l_n$ of its leaves in symmetric order.

▶ **45.** [*30*] Explain how to implement Subroutine C so that the total running time of the Garsia–Wachs algorithm is at most $O(n \log n)$.

46. [*M30*] (C. K. Wong and Shi-Kuo Chang.) Consider a scheme whereby a binary search tree is constructed by Algorithm T, except that whenever the number of nodes reaches a number of the form $2^n - 1$ the tree is reorganized into a perfectly balanced uniform tree, with 2^k nodes on level k for $0 \le k < n$. Prove that the total number of comparisons made while constructing such a tree is $N \lg N + O(N)$ on the average. (It is not difficult to show that the amount of time needed for the reorganizations is $O(N)$.)

47. [*M40*] Generalize Theorems B and M from binary trees to t-ary trees. If possible, also allow the branching costs to be nonuniform as in exercise 33.

48. [*M47*] Carry out a rigorous analysis of the steady state of a binary search tree subjected to random insertions and deletions.

49. [*HM42*] Analyze the average height of a random binary search tree.

6.2.3. Balanced Trees

The tree insertion algorithm we have just learned will produce good search trees, when the input data is random, but there is still the annoying possibility that a degenerate tree will occur. Perhaps we could devise an algorithm that keeps the tree optimum at all times; but unfortunately that seems to be very difficult. Another idea is to keep track of the total path length, and to completely reorganize the tree whenever its path length exceeds $5N \lg N$, say. But such an approach might require about $\sqrt{N/2}$ reorganizations as the tree is being built.

A very pretty solution to the problem of maintaining a good search tree was discovered in 1962 by two Russian mathematicians, G. M. Adelson-Velsky and E. M. Landis [*Doklady Akademiia̐ Nauk SSSR* **146** (1962), 263–266; English translation in *Soviet Math.* **3**, 1259–1263]. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search the tree or to insert an item. In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary *linear lists* of length N, so that each of the following operations can be done in only $O(\log N)$ units of time:

i) Find an item having a given key.

ii) Find the kth item, given k.

iii) Insert an item at a specified place.

iv) Delete a specified item.

If we use sequential allocation for linear lists, operations (i) and (ii) are efficient but operations (iii) and (iv) take order N steps; on the other hand, if we use linked allocation, operations (iii) and (iv) are efficient but operations (i) and (ii) take order N steps. A tree representation of linear lists can do *all four* operations in $O(\log N)$ steps. And it is also possible to do other standard operations with comparable efficiency, so that, for example, we can concatenate a list of M elements with a list of N elements in $O\bigl(\log(M+N)\bigr)$ steps.

The method for achieving all this involves what we shall call *balanced trees*. (Many authors also call them *AVL trees*, where the AV stands for Adelson-Velsky and the L stands for Landis.) The preceding paragraph is an advertisement for balanced trees, which makes them sound like a universal panacea that makes all other forms of data representation obsolete; but of course we ought to have a balanced attitude about balanced trees! In applications that do not involve all four of the operations above, we may be able to get by with substantially less overhead and simpler programming. Furthermore, there is no advantage to balanced trees unless N is reasonably large; thus if we have an efficient method that takes $64\lg N$ units of time and an inefficient method that takes $2N$ units of time, we should use the inefficient method unless N is greater than 256. On the other hand, N shouldn't be too large, either; balanced trees are appropriate chiefly for *internal* storage of data, and we shall study better methods for external direct-access files in Section 6.2.4. Since internal memories seem to be getting larger and larger as time goes by, balanced trees are becoming more and more important.

The *height* of a tree is defined to be its maximum level, the length of the longest path from the root to an external node. A binary tree is called *balanced* if the height of the left subtree of every node never differs by more than ± 1 from the height of its right subtree. Figure 20 shows a balanced tree with 17 internal nodes and height 5; the *balance factor* within each node is shown as $+$, $\bullet$, or $-$ according as the right subtree height minus the left subtree height is $+1$, 0, or -1. The Fibonacci tree in Fig. 8 (Section 6.2.1) is another balanced binary tree of height 5, having only 12 internal nodes; most of the balance factors in that tree

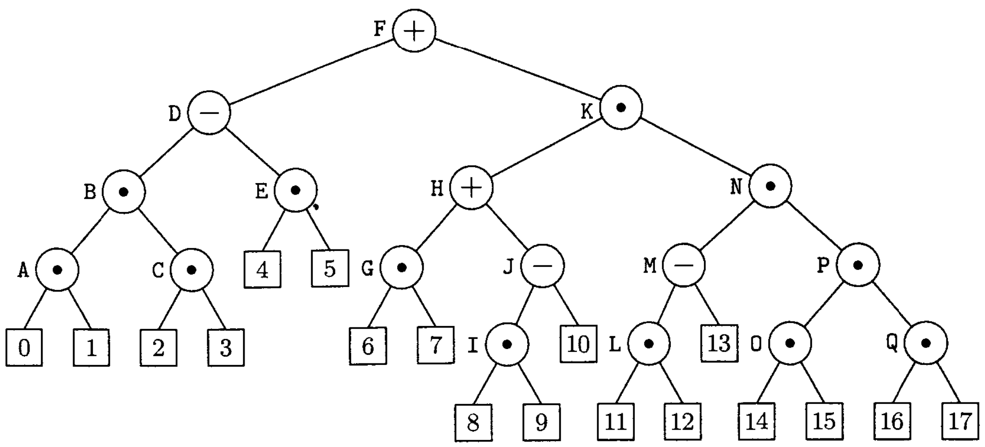


Fig. 20. A balanced binary tree.

are −1. The zodiac tree in Fig. 10 (Section 6.2.2) is *not* balanced, because the height restriction on subtrees fails at both the AQUARIUS and GEMINI nodes.

This definition of balance represents a compromise between *optimum* binary trees (with all external nodes required to be on two adjacent levels) and *arbitrary* binary trees (unrestricted). It is therefore natural to ask how far from optimum a balanced tree can be. The answer is that its search paths will never be more than 45 percent longer than the optimum:

Theorem A (Adelson-Velsky and Landis). *The height of a balanced tree with N internal nodes always lies between* $\lg(N+1)$ *and* $1.4404\lg(N+2)-0.3277$.

Proof. A binary tree of height h obviously cannot have more than 2^h external nodes; so $N+1 \le 2^h$, that is, $h \ge \lceil \lg(N+1)\rceil$ in any binary tree.

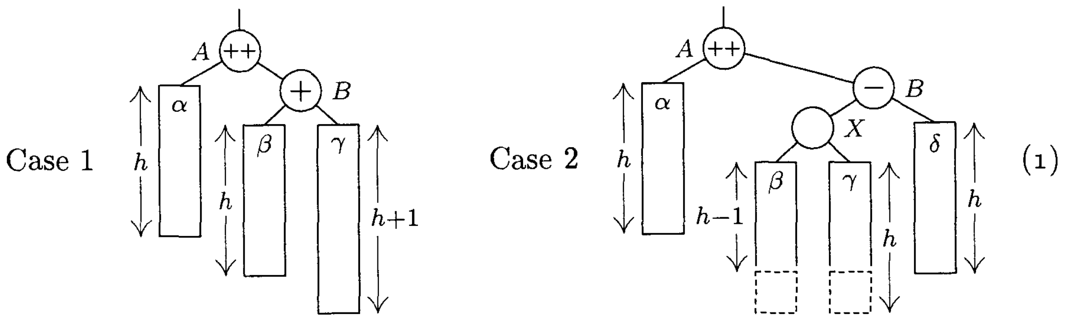
In order to find the maximum value of h, let us turn the problem around and ask for the minimum number of nodes possible in a balanced tree of height h. Let T_h be such a tree with fewest possible nodes; then one of the subtrees of the root, say the left subtree, has height $h-1$, and the other subtree has height $h-1$ or $h-2$. Since we want T_h to have the minimum number of nodes, we may assume that the left subtree of the root is T_{h-1}, and that the right subtree is T_{h-2}. This argument shows that the *Fibonacci tree* of order $h+1$ has the fewest possible nodes among all possible balanced trees of height h. (See the definition of Fibonacci trees in Section 6.2.1.) Thus

$$N \ge F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

and the stated result follows as in the corollary to Theorem 4.5.3F. ▌

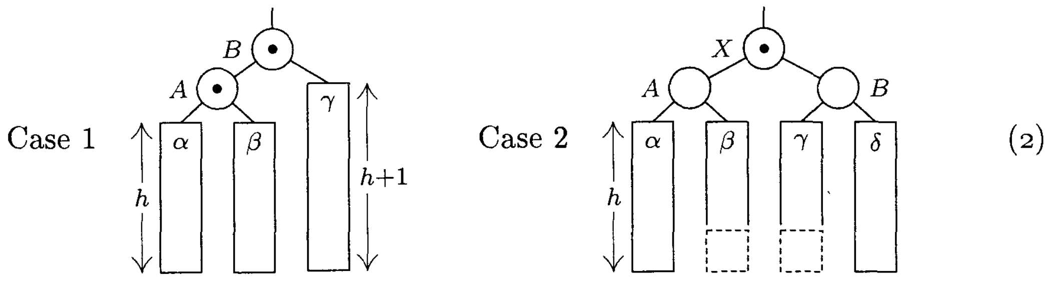
The proof of this theorem shows that a search in a balanced tree will require more than 25 comparisons only if the tree contains at least $F_{28} - 1 = 317{,}810$ nodes.

Consider now what happens when a new node is inserted into a balanced tree using tree insertion (Algorithm 6.2.2T). In Fig. 20, the tree will still be balanced if the new node takes the place of $\boxed{4}$, $\boxed{5}$, $\boxed{6}$, $\boxed{7}$, $\boxed{10}$, or $\boxed{13}$, but some adjustment will be needed if the new node falls elsewhere. The problem arises when we have a node with a balance factor of +1 whose right subtree got higher after the insertion; or, dually, if the balance factor is −1 and the left subtree got higher. It is not difficult to see that trouble arises only in two cases:



(Two other essentially identical cases occur if we reflect these diagrams, interchanging left and right.) In these diagrams the large rectangles α, β, γ, δ represent subtrees having the respective heights shown. Case 1 occurs when a new element has just increased the height of node B's right subtree from h to $h+1$, and Case 2 occurs when the new element has increased the height of B's left subtree. In the second case, we have either $h = 0$ (so that X itself was the new node), or else node X has two subtrees of respective heights $(h-1, h)$ or $(h, h-1)$.

Simple transformations will restore balance in both of these cases, while preserving the symmetric order of the tree nodes:



In Case 1 we simply "rotate" the tree to the left, attaching β to A instead of B. This transformation is like applying the associative law to an algebraic formula, replacing $\alpha(\beta\gamma)$ by $(\alpha\beta)\gamma$. In Case 2 we use a double rotation, first rotating (X, B) right, then (A, X) left. In both cases only a few links of the tree need to be changed. Furthermore, the new trees have height $h+2$, which is exactly the height that was present before the insertion; hence the rest of the tree (if any) that was originally above node A always remains balanced.

For example, if we insert a new node into position $\boxed{17}$ of Fig. 20 we obtain the balanced tree shown in Fig. 21, after a single rotation (Case 1). Notice that several of the balance factors have changed.

The details of this insertion procedure can be worked out in several ways. At first glance an auxiliary stack seems to be necessary, in order to keep track of which nodes will be affected, but the following algorithm gains some speed by

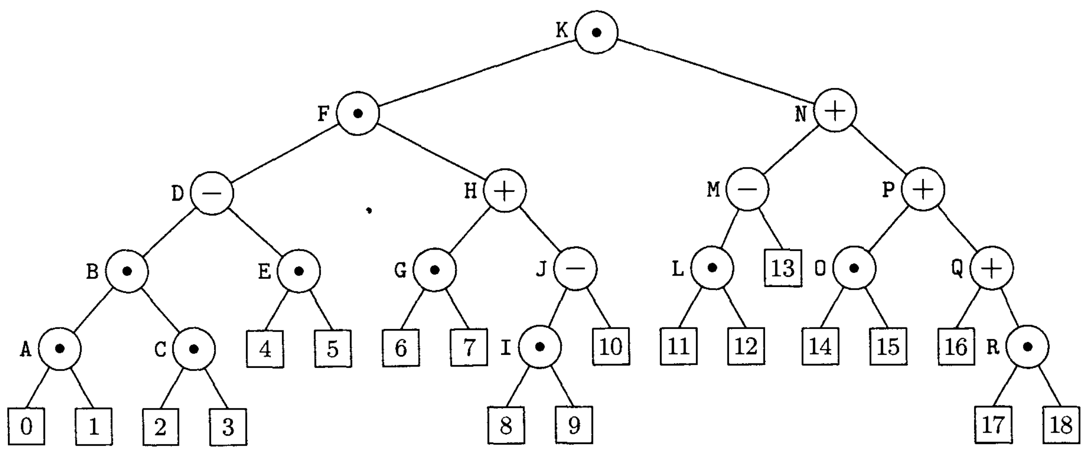


Fig. 21. The tree of Fig. 20, rebalanced after a new key R has been inserted.

exploiting the fact that the balance factor of node B in (1) was zero before the insertion.

Algorithm A (*Balanced tree search and insertion*). Given a table of records that form a balanced binary tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place and the tree is rebalanced if necessary.

The nodes of the tree are assumed to contain KEY, LLINK, and RLINK fields as in Algorithm 6.2.2T. We also have a new field

$$\texttt{B(P)} = \text{balance factor of } \texttt{NODE(P)},$$

the height of the right subtree minus the height of the left subtree; this field always contains either $+1$, 0, or -1. A special header node also appears at the top of the tree, in location HEAD; the value of RLINK(HEAD) is a pointer to the root of the tree, and LLINK(HEAD) is used to keep track of the overall height of the tree. (Knowledge of the height is not really necessary for this algorithm, but it is useful in the concatenation procedure discussed below.) We assume that the tree is *nonempty*, namely that RLINK(HEAD) $\neq \Lambda$.

For convenience in description, the algorithm uses the notation LINK(a,P) as a synonym for LLINK(P) if $a = -1$, and for RLINK(P) if $a = +1$.

A1. [Initialize.] Set T $\leftarrow$ HEAD, S $\leftarrow$ P $\leftarrow$ RLINK(HEAD). (The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary, and T always points to the parent of S.)

A2. [Compare.] If $K <$ KEY(P), go to A3; if $K >$ KEY(P), go to A4; and if $K =$ KEY(P), the search terminates successfully.

A3. [Move left.] Set Q $\leftarrow$ LLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and LLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set P $\leftarrow$ Q and return to step A2.

A4. [Move right.] Set Q $\leftarrow$ RLINK(P). If Q $= \Lambda$, set Q $\Leftarrow$ AVAIL and RLINK(P) $\leftarrow$ Q and go to step A5. Otherwise if B(Q) $\neq 0$, set T $\leftarrow$ P and S $\leftarrow$ Q. Finally set

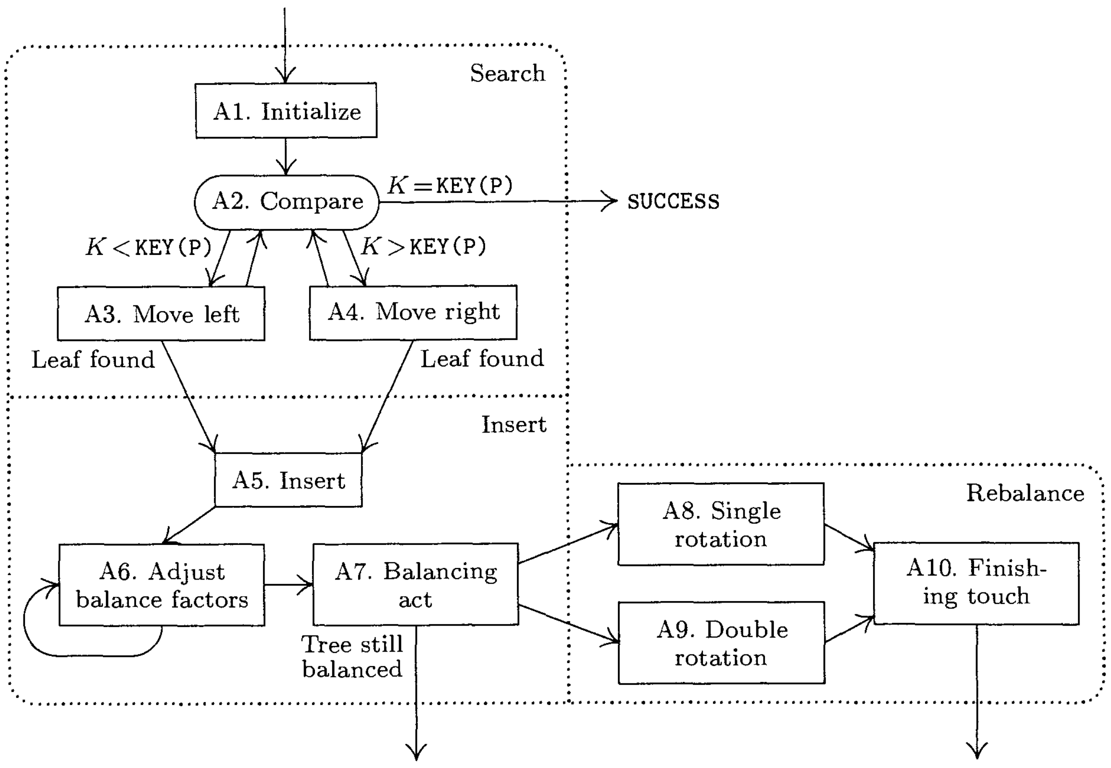


Fig. 22. Balanced tree search and insertion.

P ← Q and return to step A2. (The last part of this step may be combined with the last part of step A3.)

A5. [Insert.] (We have just linked a new node, `NODE(Q)`, into the tree, and its fields need to be initialized.) Set `KEY(Q)` ← K, `LLINK(Q)` ← `RLINK(Q)` ← Λ, and `B(Q)` ← 0.

A6. [Adjust balance factors.] (Now the balance factors on nodes between S and Q need to be changed from zero to ±1.) If $K <$ `KEY(S)` set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Then set R ← P ← `LINK(`a`,S)`, and repeatedly do the following operations zero or more times until P = Q: If $K <$ `KEY(P)` set `B(P)` ← -1 and P ← `LLINK(P)`; if $K >$ `KEY(P)`, set `B(P)` ← $+1$ and P ← `RLINK(P)`. (If $K =$ `KEY(P)`, then P = Q and we proceed to the next step.)

A7. [Balancing act.] Several cases now arise:

i) If `B(S)` = 0 (the tree has grown higher), set `B(S)` ← a, `LLINK(HEAD)` ← `LLINK(HEAD)` + 1, and terminate the algorithm.

ii) If `B(S)` = $-a$ (the tree has gotten more balanced), set `B(S)` ← 0 and terminate the algorithm.

iii) If `B(S)` = a (the tree has gotten out of balance), go to step A8 if `B(R)` = a, to A9 if `B(R)` = $-a$.

(Case (iii) corresponds to the situations depicted in (1) when $a = +1$; S and R point, respectively, to nodes A and B, and `LINK(`$-a$`,S)` points to α, etc.)

A8. [Single rotation.] Set `P` $\leftarrow$ `R`, `LINK(`a`,S)` $\leftarrow$ `LINK(`$-a$`,R)`, `LINK(`$-a$`,R)` $\leftarrow$ `S`, `B(S)` $\leftarrow$ `B(R)` $\leftarrow 0$. Go to A10.

A9. [Double rotation.] Set `P` $\leftarrow$ `LINK(`$-a$`,R)`, `LINK(`$-a$`,R)` $\leftarrow$ `LINK(`a`,P)`, `LINK(`a`,P)` $\leftarrow$ `R`, `LINK(`a`,S)` $\leftarrow$ `LINK(`$-a$`,P)`, `LINK(`$-a$`,P)` $\leftarrow$ `S`. Now set

$$(\texttt{B(S)},\texttt{B(R)}) \leftarrow \begin{cases} (-a,0), & \text{if } \texttt{B(P)} = a; \\ (\ \ 0,0), & \text{if } \texttt{B(P)} = 0; \\ (\ \ 0,a), & \text{if } \texttt{B(P)} = -a; \end{cases} \tag{3}$$

and then set `B(P)` $\leftarrow 0$.

A10. [Finishing touch.] (We have completed the rebalancing transformation, taking (1) to (2), with `P` pointing to the new subtree root and `T` pointing to the parent of the old subtree root `S`.) If `S` = `RLINK(T)` then set `RLINK(T)` $\leftarrow$ `P`, otherwise set `LLINK(T)` $\leftarrow$ `P`. ▌

This algorithm is rather long, but it divides into three simple parts: Steps A1–A4 do the search, steps A5–A7 insert a new node, and steps A8–A10 rebalance the tree if necessary. Essentially the same method can be used if the tree is *threaded* (see exercise 6.2.2–2), since the balancing act never needs to make difficult changes to thread links.

We know that the algorithm takes about $C \log N$ units of time, for some C, but it is important to know the approximate value of C so that we can tell how large N should be in order to make balanced trees worth all the trouble. The following `MIX` implementation gives some insight into this question.

Program A (*Balanced tree search and insertion*). This program for Algorithm A uses tree nodes having the form

B	LLINK	RLINK
KEY		

; (4)

rA $\equiv K$, rI1 $\equiv$ `P`, rI2 $\equiv$ `Q`, rI3 $\equiv$ `R`, rI4 $\equiv$ `S`, rI5 $\equiv$ `T`. The code for steps A7–A9 is duplicated so that the value of a appears implicitly (not explicitly) in the program.

```
B     EQU  0:1
LLINK EQU  2:3
RLINK EQU  4:5
START LDA  K              1      A1. Initialize.
      ENT5 HEAD           1      T ← HEAD.
      LD2  0,5(RLINK)     1      Q ← RLINK(HEAD).
      JMP  2F             1      To A2 with S ← P ← Q.
4H    LD2  0,1(RLINK)    C2      A4. Move right. Q ← RLINK(P).
      J2Z  5F            C2      To A5 if Q = Λ.
1H    LDX  0,2(B)       C − 1    rX ← B(Q).
      JXZ  *+3          C − 1    Jump if B(Q) = 0.
      ENT5 0,1          D − 1    T ← P.
```

```
2H    ENT4 0,2              D          S ← Q.
      ENT1 0,2              C          P ← Q.
      CMPA 1,1              C          A2. Compare.
      JG   4B               C          To A4 if K > KEY(P).
      JE   SUCCESS          C1         Exit if K = KEY(P).
      LD2  0,1(LLINK)       C1 − S     A3. Move left. Q ← LLINK(P).
      J2NZ 1B               C1 − S     Jump if Q ≠ Λ.
5H    LD2  AVAIL            1 − S      A5. Insert.
      J2Z  OVERFLOW         1 − S
      LDX  0,2(RLINK)       1 − S
      STX  AVAIL            1 − S      Q ⇐ AVAIL.
      STA  1,2              1 − S      KEY(Q) ← K.
      STZ  0,2              1 − S      LLINK(Q) ← RLINK(Q) ← Λ.
      JL   1F               1 − S      Was K < KEY(P)?
      ST2  0,1(RLINK)       A          RLINK(P) ← Q.
      JMP  *+2              A
1H    ST2  0,1(LLINK)       1 − S − A  LLINK(P) ← Q.
6H    CMPA 1,4              1 − S      A6. Adjust balance factors.
      JL   *+3              1 − S      Jump if K < KEY(S).
      LD3  0,4(RLINK)       E          R ← RLINK(S).
      JMP  *+2              E
      LD3  0,4(LLINK)       1 − S − E  R ← LLINK(S).
      ENT1 0,3              1 − S      P ← R.
      ENTX -1               1 − S      rX ← −1.
      JMP  1F               1 − S      To comparison loop.
4H    JE   7F               F2 + 1 − S To A7 if K = KEY(P).
      STX  0,1(1:1)         F2         B(P) ← +1 (it was +0).
      LD1  0,1(RLINK)       F2         P ← RLINK(P).
1H    CMPA 1,1              F + 1 − S
      JGE  4B               F + 1 − S  Jump if K ≥ KEY(P).
      STX  0,1(B)           F1         B(P) ← −1.
      LD1  0,1(LLINK)       F1         P ← LLINK(P).
      JMP  1B               F1         To comparison loop.
7H    LD2  0,4(B)           1 − S      A7. Balancing act. rI2 ← B(S).
      STZ  0,4(B)           1 − S      B(S) ← 0.
      CMPA 1,4              1 − S
      JG   A7R              1 − S      To a = +1 routine if K > KEY(S).
A7L   J2P  DONE             U1         Exit if rI2 = −a.
      J2Z  7F               G1 + J1    Jump if B(S) was zero.
      ENT1 0,3              G1         P ← R.
      LD2  0,3(B)           G1         rI2 ← B(R).
      J2N  A8L              G1         To A8 if rI2 = a.
A9L   LD1  0,3(RLINK)       H1         A9. Double rotation.
      LDX  0,1(LLINK)       H1         LINK(a,P ← LINK(−a,R))
      STX  0,3(RLINK)       H1            → LINK(−a,R).
      ST3  0,1(LLINK)       H1         LINK(a,P) ← R.
      LD2  0,1(B)           H1         rI2 ← B(P).
      LDX  T1,2             H1         −a, 0 or 0
      STX  0,4(B)           H1            → B(S).
```

```
       LDX  T2,2              H1      0, 0, or a
       STX  0,3(B)            H1        → B(R).
A8L    LDX  0,1(RLINK)        G1      A8. Single rotation.
       STX  0,4(LLINK)        G1      LINK(a,S) ← LINK(−a,P).
       ST4  0,1(RLINK)        G1      LINK(−a,P) ← S.
       JMP  8F                G1      Join up with the other branch.
A7R    J2N  DONE              U2      Exit if rI2 = −a.
       J2Z  6F             G2 + J2    Jump if B(S) was zero.
       ENT1 0,3               G2      P ← R.
       LD2  0,3(B)            G2      rI2 ← B(R).
       J2P  A8R               G2      To A8 if rI2 = a.
A9R    LD1  0,3(LLINK)        H2      A9. Double rotation.
       LDX  0,1(RLINK)        H2      LINK(a,P ← LINK(−a,R))
       STX  0,3(LLINK)        H2        → LINK(−a,R).
       ST3  0,1(RLINK)        H2      LINK(a,P) ← R.
       LD2  0,1(B)            H2      rI2 ← B(P).
       LDX  T2,2              H2      −a, 0 or 0
       STX  0,4(B)            H2        → B(S).
       LDX  T1,2              H2      0, 0, or a
       STX  0,3(B)            H2        → B(R).
A8R    LDX  0,1(LLINK)        G2      A8. Single rotation.
       STX  0,4(RLINK)        G2      LINK(a,S) ← LINK(−a,P).
       ST4  0,1(LLINK)        G2      LINK(−a,P) ← S.
8H     STZ  0,1(B)            G       B(P) ← 0.
A10    CMP4 0,5(RLINK)        G       A10. Finishing touch.
       JNE  *+3               G       Jump if RLINK(T) ≠ S.
       ST1  0,5(RLINK)        G3      RLINK(T) ← P.
       JMP  DONE              G3      Exit.
       ST1  0,5(LLINK)        G4      LLINK(T) ← P.
       JMP  DONE              G4      Exit.
       CON  +1
T1     CON  0                         Table for (3).
T2     CON  0
       CON  -1
6H     ENTX +1                J2      rX ← +1.
7H     STX  0,4(B)            J       B(S) ← a.
       LDX  HEAD(LLINK)       J       LLINK(HEAD)
       INCX 1                 J         + 1
       STX  HEAD(LLINK)       J           → LLINK(HEAD).
DONE   EQU  *               1 − S     Insertion is complete. ▌
```

Analysis of balanced tree insertion. [Nonmathematical readers, please skip to (10).] In order to figure out the running time of Algorithm A, we would like to know the answers to the following questions:

- How many comparisons are made during the search?
- How far apart will nodes S and Q be? (In other words, how much adjustment is needed in step A6?)
- How often do we need to do a single or double rotation?

It is not difficult to derive upper bounds on the worst case running time, using Theorem A, but of course in practice we want to know the average behavior. No theoretical determination of the average behavior has been successfully completed as yet, since the algorithm appears to be quite complicated, but several interesting theoretical and empirical results have been obtained.

In the first place we can ask about the number B_{nh} of balanced binary trees with n internal nodes and height h. It is not difficult to compute the generating function $B_h(z) = \sum_{n\ge 0} B_{nh} z^n$ for small h, from the relations

$$B_0(z) = 1, \qquad B_1(z) = z, \qquad B_{h+1}(z) = zB_h(z)\big(B_h(z) + 2B_{h-1}(z)\big). \tag{5}$$

(See exercise 6.) Thus

$$\begin{aligned} B_2(z) &= 2z^2 + z^3, \\ B_3(z) &= \qquad 4z^4 + 6z^5 + 4z^6 + z^7, \\ B_4(z) &= \qquad\qquad 16z^7 + 32z^8 + 44z^9 + \cdots + 8z^{14} + z^{15}, \end{aligned}$$

and in general $B_h(z)$ has the form

$$2^{F_{h+1}-1} z^{F_{h+2}-1} + 2^{F_{h+1}-2} L_{h-1} z^{F_{h+2}} + \text{complicated terms} + 2^{h-1} z^{2^h-2} + z^{2^h-1} \tag{6}$$

for $h \ge 3$, where $L_k = F_{k+1} + F_{k-1}$. (This formula generalizes Theorem A.) The total number of balanced trees with height h is $B_h = B_h(1)$, which satisfies the recurrence

$$B_0 = B_1 = 1, \qquad B_{h+1} = B_h^2 + 2B_hB_{h-1}, \tag{7}$$

so that $B_2 = 3$, $B_3 = 3\cdot 5$, $B_4 = 3^2\cdot 5\cdot 7$, $B_5 = 3^3\cdot 5^2\cdot 7\cdot 23$; and, in general,

$$B_h = A_0^{F_h} A_1^{F_{h-1}} \ldots A_{h-1}^{F_1} A_h^{F_0}, \tag{8}$$

where $A_0 = 1$, $A_1 = 3$, $A_2 = 5$, $A_3 = 7$, $A_4 = 23$, $A_5 = 347$, $\ldots$, $A_h = A_{h-1}B_{h-2} + 2$. The sequences B_h and A_h grow very rapidly; in fact, they are *doubly exponential*: Exercise 7 shows that there is a real number $\theta \approx 1.43687$ such that

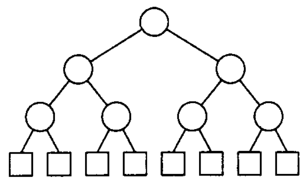
$$B_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor. \tag{9}$$

If we consider each of the B_h trees to be equally likely, exercise 8 shows that the average number of nodes in a tree of height h is

$$B_h'(1)/B_h(1) \approx (0.70118)2^h - 1. \tag{10}$$

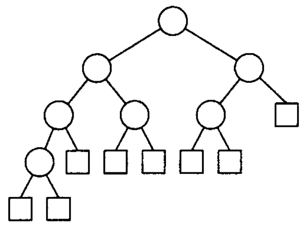
This indicates that the height of a balanced tree with N nodes is usually much closer to $\log_2 N$ than to $\log_\phi N$.

Unfortunately, these results don't really have much to do with Algorithm A, since the mechanism of that algorithm makes some trees significantly more probable than others. For example, consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys can be inserted, and the perfectly balanced "complete" tree



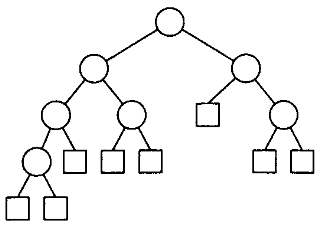
(11)

is obtained 2160 times. By contrast, the Fibonacci tree



(12)

occurs only 144 times, and the similar tree



(13)

occurs 216 times. Replacing the left subtrees of (12) and (13) by arbitrary four-node balanced trees, and then reflecting left and right, yields 16 different trees; the eight generated from (12) each occur 144 times, and those generated from (13) each occur 216 times. It is surprising that (13) is more common than (12).

The fact that the perfectly balanced tree is obtained with such high probability — together with (10), which corresponds to the case of equal probabilities — makes it plausible that the average search time for a balanced tree should be about $\lg N + c$ comparisons for some small constant c. But R. W. Floyd has observed that the coefficient of $\lg N$ is unlikely to be exactly 1, because the root of the tree would then be near the median, and the roots of its two subtrees would be near the quartiles; then single and double rotation could not easily keep the root near the median. Empirical tests indicate that the true average number of comparisons needed to insert the Nth item is approximately $1.01 \lg N + 0.1$, except when N is small.

In order to study the behavior of the insertion and rebalancing phases of Algorithm A, we can classify the external nodes of balanced trees as shown in Fig. 23. The path leading up from an external node can be specified by a sequence of +'s and -'s (+ for a right link, - for a left link); we write down the link specifications until reaching the first node with a nonzero balance factor, or until reaching the root, if there is no such node. Then we write `A` or `B` according as the new tree will be balanced or unbalanced when an internal node is inserted in the given place. Thus the path up from $\boxed{3}$ is `++-B`, meaning "right link, right link, left link, unbalance." A specification ending in `A` requires

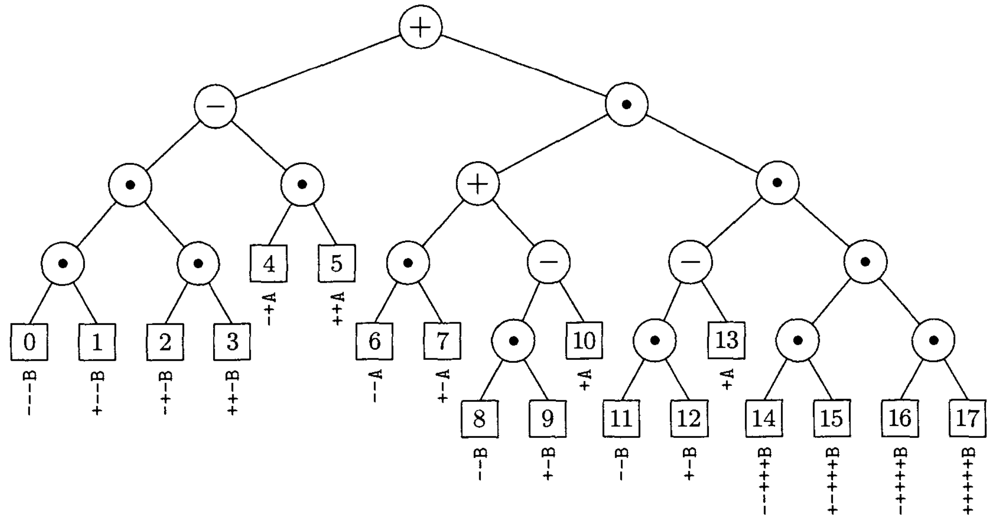


Fig. 23. Classification codes that specify the behavior of Algorithm A after insertion.

no rebalancing after insertion of a new node; a specification ending in ++B or --B requires a single rotation; and a specification ending in +-B or -+B requires a double rotation. When k links appear in the specification, step A6 has to adjust exactly $k-1$ balance factors. Thus the specifications give the essential facts that govern the running time of steps A6 to A10.

Empirical tests on random numbers for $100 \le N \le 2000$ gave the approximate probabilities shown in Table 1 for paths of various types; apparently these probabilities rapidly approach limiting values as $N \to \infty$. Table 2 gives the exact probabilities corresponding to Table 1 when $N = 10$, considering the 10! permutations of the input as equally probable. (The probabilities that show up as .143 in Table 1 are actually equal to 1/7, for all $N \ge 7$; see exercise 11. Single and double rotations are equally likely when $N \le 15$, but double rotations occur slightly less often when $N \ge 16$.)

Table 1

APPROXIMATE PROBABILITIES FOR INSERTING THE NTH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	.143	.000	.000
2	.152	.143	.143
3	.092	.048	.048
4	.060	.024	.024
5	.036	.010	.010
> 5	.051	.009	.008
ave 2.78	total .534	.233	.232

From Table 1 we can see that k is ≤ 2 with probability about $.143 + .153 + .143 + .143 = .582$; thus, step A6 is quite simple almost 60 percent of the time. The average number of balance factors changed from 0 to ± 1 in that step is

Table 2

EXACT PROBABILITIES FOR INSERTING THE 10TH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	1/7	0	0
2	6/35	1/7	1/7
3	4/21	2/35	2/35
4	0	1/21	1/21
ave 247/105	53/105	26/105	26/105

about 1.8. The average number of balanced factors changed from ± 1 to 0 in steps A7 through A10 is approximately $.534+2(.233+.232) \approx 1.5$; thus, inserting one new node adds about $1.8-1.5=0.3$ unbalanced nodes, on the average. This agrees with the fact that about 68 percent of all nodes were found to be balanced in random trees built by Algorithm A.

An approximate model of the behavior of Algorithm A has been proposed by C. C. Foster [*Proc. ACM Nat. Conf.* **20** (1965), 192–205.] This model is not rigorously accurate, but it is close enough to the truth to give some insight. Let us assume that p is the probability that the balance factor of a given node in a large tree built by Algorithm A is 0; then the balance factor is $+1$ with probability $\frac{1}{2}(1-p)$, and it is -1 with the same probability $\frac{1}{2}(1-p)$. Let us assume further (without justification) that the balance factors of all nodes are independent. Then the probability that step A6 sets exactly $k-1$ balance factors nonzero is $p^{k-1}(1-p)$, so the average value of k is $1/(1-p)$. The probability that we need to rotate part of the tree is $q \approx \frac{1}{2}$. Inserting a new node should increase the number of balanced nodes by p, on the average; this number is actually increased by 1 in step A5, by $-p/(1-p)$ in step A6, by q in step A7, and by $2q$ in step A8 or A9, so we should have

$$p = 1 - p/(1-p) + 3q \approx 5/2 - p/(1-p).$$

Solving for p yields fair agreement with Table 1:

$$p \approx \frac{9-\sqrt{41}}{4} \approx 0.649; \qquad 1/(1-p) \approx 2.851. \tag{14}$$

The running time of the search phase of Program A (lines 01–19) is

$$10C + C1 + 2D + 2 - 3S, \tag{15}$$

where C, $C1$, S are the same as in previous algorithms of this chapter and D is the number of unbalanced nodes encountered on the search path. Empirical tests show that we may take $D \approx \frac{1}{3}C$, $C1 \approx \frac{1}{2}(C+S)$, $C+S \approx 1.01 \lg N + 0.1$, so the average search time is approximately $11.3 \lg N + 3.3 - 13.7S$ units. (If searching is done much more often than insertion, we could of course use a separate, faster program for searching, since it would be unnecessary to look at the balance factors; the average running time for a successful search would then be only about $(6.6 \lg N - 3.4)u$, and the worst case running time would in fact be better than the average running time obtained with Program 6.2.2T.)

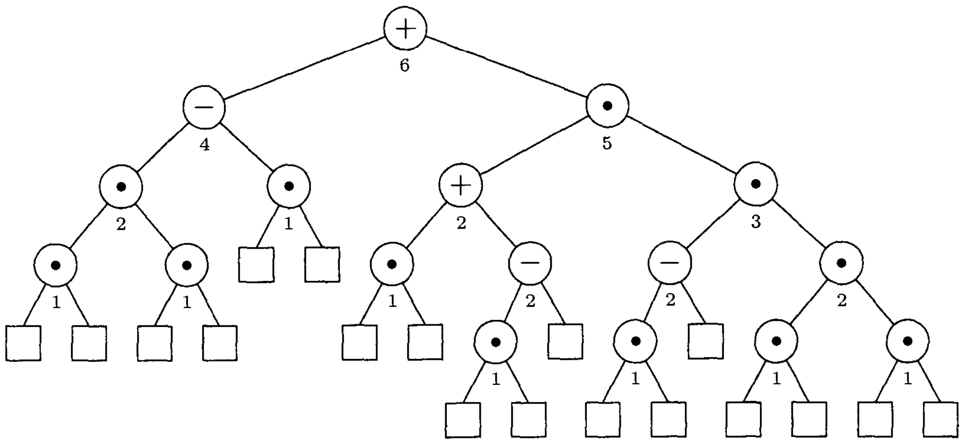


Fig. 24. RANK fields, used for searching by position.

The running time of the insertion phase of Program A (lines 20–45) is $8F + 26 + (0,\ 1,\ \text{or}\ 2)$ units, when the search is unsuccessful. The data of Table 1 indicate that $F \approx 1.8$ on the average. The rebalancing phase (lines 46–101) takes either 16.5, 8, 27.5, or 45.5 (± 0.5) units, depending on whether we increase the total height, or simply exit without rebalancing, or do a single or double rotation. The first case almost never occurs, and the others occur with the approximate probabilities .534, .233, .232, so the average running time of the combined insertion-rebalancing portion of Program A is about $63u$.

These figures indicate that maintenance of a balanced tree in memory is reasonably fast, even though the program is rather lengthy. If the input data are random, the simple tree insertion algorithm of Section 5.2.2 is roughly $50u$ faster per insertion; but the balanced tree algorithm is guaranteed to be reliable even with nonrandom input data.

One way to compare Program A with Program 6.2.2T is to consider the worst case of the latter. If we study the amount of time necessary to insert N keys in increasing order into an initially empty tree, it turns out that Program A is slower for $N \leq 26$ and faster for $N \geq 27$.

Linear list representation. Now let us return to the claim made at the beginning of this section, that balanced trees can be used to represent linear lists in such a way that we can insert items rapidly (overcoming the difficulty of sequential allocation), yet we can also perform random accesses to list items (overcoming the difficulty of linked allocation).

The idea is to introduce a new field in each node, called the RANK field. The field indicates the relative position of that node in its subtree, namely one plus the number of nodes in its left subtree. Figure 24 shows the RANK values for the binary tree of Fig. 23. We can eliminate the KEY field entirely; or, if desired, we can have both KEY and RANK fields, so that it is possible to retrieve items either by their key value or by their relative position in the list.

Using such a RANK field, retrieval by position is a straightforward modification of the search algorithms we have been studying.

Algorithm B (*Tree search by position*). Given a linear list represented as a binary tree, this algorithm finds the kth element of the list (the kth node of the tree in symmetric order), given k. The binary tree is assumed to have LLINK and RLINK fields and a header as in Algorithm A, plus a RANK field as described above.

B1. [Initialize.] Set $\mathtt{M} \leftarrow k$, $\mathtt{P} \leftarrow \mathtt{RLINK(HEAD)}$.

B2. [Compare.] If $\mathtt{P} = \Lambda$, the algorithm terminates unsuccessfully. (This can happen only if k was greater than the number of nodes in the tree, or $k \le 0$.) Otherwise if $\mathtt{M} < \mathtt{RANK(P)}$, go to B3; if $\mathtt{M} > \mathtt{RANK(P)}$, go to B4; and if $\mathtt{M} = \mathtt{RANK(P)}$, the algorithm terminates successfully (P points to the kth node).

B3. [Move left.] Set $\mathtt{P} \leftarrow \mathtt{LLINK(P)}$ and return to B2.

B4. [Move right.] Set $\mathtt{M} \leftarrow \mathtt{M} - \mathtt{RANK(P)}$ and $\mathtt{P} \leftarrow \mathtt{RLINK(P)}$ and return to B2. ▌

The only new point of interest in this algorithm is the manipulation of M in step B4. We can modify the insertion procedure in a similar way, although the details are somewhat trickier:

Algorithm C (*Balanced tree insertion by position*). Given a linear list represented as a balanced binary tree, this algorithm inserts a new node just before the kth element of the list, given k and a pointer Q to the new node. If $k = N+1$, the new node is inserted just after the last element of the list.

The binary tree is assumed to be nonempty and to have LLINK, RLINK and B fields and a header, as in Algorithm A, plus a RANK field as described above. This algorithm is merely a transcription of Algorithm A; the difference is that it uses and updates the RANK fields instead of the KEY fields.

C1. [Initialize.] Set $\mathtt{T} \leftarrow \mathtt{HEAD}$, $\mathtt{S} \leftarrow \mathtt{P} \leftarrow \mathtt{RLINK(HEAD)}$, $\mathtt{U} \leftarrow \mathtt{M} \leftarrow k$.

C2. [Compare.] If $\mathtt{M} \le \mathtt{RANK(P)}$, go to C3, otherwise go to C4.

C3. [Move left.] Set $\mathtt{RANK(P)} \leftarrow \mathtt{RANK(P)} + 1$ (we will be inserting a new node to the left of P). Set $\mathtt{R} \leftarrow \mathtt{LLINK(P)}$. If $\mathtt{R} = \Lambda$, set $\mathtt{LLINK(P)} \leftarrow \mathtt{Q}$ and go to C5. Otherwise if $\mathtt{B(R)} \ne 0$ set $\mathtt{T} \leftarrow \mathtt{P}$, $\mathtt{S} \leftarrow \mathtt{R}$, and $\mathtt{U} \leftarrow \mathtt{M}$. Finally set $\mathtt{P} \leftarrow \mathtt{R}$ and return to C2.

C4. [Move right.] Set $\mathtt{M} \leftarrow \mathtt{M} - \mathtt{RANK(P)}$, and $\mathtt{R} \leftarrow \mathtt{RLINK(P)}$. If $\mathtt{R} = \Lambda$, set $\mathtt{RLINK(P)} \leftarrow \mathtt{Q}$ and go to C5. Otherwise if $\mathtt{B(R)} \ne 0$ set $\mathtt{T} \leftarrow \mathtt{P}$, $\mathtt{S} \leftarrow \mathtt{R}$, and $\mathtt{U} \leftarrow \mathtt{M}$. Finally set $\mathtt{P} \leftarrow \mathtt{R}$ and return to C2.

C5. [Insert.] Set $\mathtt{RANK(Q)} \leftarrow 1$, $\mathtt{LLINK(Q)} \leftarrow \mathtt{RLINK(Q)} \leftarrow \Lambda$, $\mathtt{B(Q)} \leftarrow 0$.

C6. [Adjust balance factors.] Set $\mathtt{M} \leftarrow \mathtt{U}$. (This restores the former value of M when P was S; all RANK fields are now properly set.) If $\mathtt{M} < \mathtt{RANK(S)}$, set $\mathtt{R} \leftarrow \mathtt{P} \leftarrow \mathtt{LLINK(S)}$ and $a \leftarrow -1$; otherwise set $\mathtt{R} \leftarrow \mathtt{P} \leftarrow \mathtt{RLINK(S)}$, $a \leftarrow +1$, and $\mathtt{M} \leftarrow \mathtt{M} - \mathtt{RANK(S)}$. Then repeatedly do the following operations until $\mathtt{P} = \mathtt{Q}$: If $\mathtt{M} < \mathtt{RANK(P)}$, set $\mathtt{B(P)} \leftarrow -1$ and $\mathtt{P} \leftarrow \mathtt{LLINK(P)}$; if $\mathtt{M} > \mathtt{RANK(P)}$, set $\mathtt{B(P)} \leftarrow +1$ and $\mathtt{M} \leftarrow \mathtt{M} - \mathtt{RANK(P)}$ and $\mathtt{P} \leftarrow \mathtt{RLINK(P)}$. (If $\mathtt{M} = \mathtt{RANK(P)}$, then $\mathtt{P} = \mathtt{Q}$ and we proceed to the next step.)

C7. [Balancing act.] Several cases now arise.

i) If `B(S)` $= 0$, set `B(S)` $\leftarrow a$, `LLINK(HEAD)` $\leftarrow$ `LLINK(HEAD)` $+ 1$, and terminate the algorithm.

ii) If `B(S)` $= -a$, set `B(S)` $\leftarrow 0$ and terminate the algorithm.

iii) If `B(S)` $= a$, go to step C8 if `B(R)` $= a$, to C9 if `B(R)` $= -a$.

C8. [Single rotation.] Set `P` = `R`, `LINK(`a`,S)` $\leftarrow$ `LINK(`$-a$`,R)`, `LINK(`$-a$`,R)` $\leftarrow$ `S`, `B(S)` $\leftarrow$ `B(R)` $\leftarrow 0$. If $a = +1$, set `RANK(R)` $\leftarrow$ `RANK(R)` + `RANK(S)`; if $a = -1$, set `RANK(S)` $\leftarrow$ `RANK(S)` − `RANK(R)`. Go to C10.

C9. [Double rotation.] Do all the operations of step A9 (Algorithm A). Then if $a = +1$, set `RANK(R)` $\leftarrow$ `RANK(R)` − `RANK(P)`, `RANK(P)` $\leftarrow$ `RANK(P)` + `RANK(S)`; if $a = -1$, set `RANK(P)` $\leftarrow$ `RANK(P)` + `RANK(R)`, then `RANK(S)` $\leftarrow$ `RANK(S)` − `RANK(P)`.

C10. [Finishing touch.] If `S` = `RLINK(T)` then set `RLINK(T)` $\leftarrow$ `P`, otherwise set `LLINK(T)` $\leftarrow$ `P`. ▌

***Deletion, concatenation, etc.** It is possible to do many other things to balanced trees and maintain the balance, but the algorithms are sufficiently lengthy that the details are beyond the scope of this book. We shall discuss the general ideas here, and an interested reader will be able to fill in the details without much difficulty.

The problem of deletion can be solved in $O(\log N)$ steps if we approach it correctly [C. C. Foster, "A Study of AVL Trees," Goodyear Aerospace Corp. report GER-12158 (April 1965)]. In the first place we can reduce deletion of an arbitrary node to the simple deletion of a node `P` for which `LLINK(P)` or `RLINK(P)` is Λ, as in Algorithm 6.2.2D. The algorithm should also be modified so that it constructs a list of pointers that specify the path to node `P`, namely

$$(P_0, a_0), \qquad (P_1, a_1), \qquad \ldots, \qquad (P_l, a_l), \tag{16}$$

where P_0 = `HEAD`, $a_0 = +1$; `LINK(`a_i`,`P_i`)` $= P_{i+1}$, for $0 \le i < l$; P_l = `P`; and `LINK(`a_l`,`P_l`)` $= \Lambda$. This list can be placed on an auxiliary stack as we search down the tree. The process of deleting node `P` sets `LINK(`a_{l-1}`,`P_{l-1}`)` $\leftarrow$ `LINK(`$-a_l$`,`P_l`)`, and we must adjust the balance factor at node P_{l-1}. Suppose that we need to adjust the balance factor at node P_k, because the a_k subtree of this node has just decreased in height; the following adjustment procedure should be used: If $k = 0$, set `LLINK(HEAD)` $\leftarrow$ `LLINK(HEAD)` $- 1$ and terminate the algorithm, since the whole tree has decreased in height. Otherwise look at the balance factor `B(`P_k`)`; there are three cases:

i) `B(`P_k`)` $= a_k$. Set `B(`P_k`)` $\leftarrow 0$, decrease k by 1, and repeat the adjustment procedure for this new value of k.

ii) `B(`P_k`)` $= 0$. Set `B(`P_k`)` to $-a_k$ and terminate the deletion algorithm.

iii) `B(`P_k`)` $= -a_k$. Rebalancing is required!

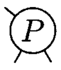
The situations that require rebalancing are almost the same as we met in the insertion algorithm; referring again to (1), A is node P_k, and B is the node `LINK(`$-a_k$`,`P_k`)`, on the *opposite* branch from where the deletion has occurred. The only new feature is that node B might be balanced; this leads to a new

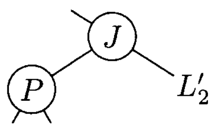
Case 3, which is like Case 1 except that β has height $h + 1$. In Cases 1 and 2, rebalancing as in (2) means that we decrease the height, so we set `LINK(`a_{k-1}`,`P_{k-1}`)` to the root of (2), decrease k by 1, and restart the adjustment procedure for this new value of k. In Case 3 we do a single rotation, and this leaves the balance factors of both A and B nonzero without changing the overall height; after making `LINK(`a_{k-1}`,`P_{k-1}`)` point to node B, we therefore terminate the algorithm.

The important difference between deletion and insertion is that deletion might require up to $\log N$ rotations, while insertion never needs more than one. The reason for this becomes clear if we try to delete the rightmost node of a Fibonacci tree (see Fig. 8 in Section 6.2.1). But empirical tests show that only about 0.21 rotations per deletion are actually needed, on the average.

The use of balanced trees for linear list representation suggests also the need for a *concatenation* algorithm, where we want to insert an entire tree L_2 to the right of tree L_1, without destroying the balance. An elegant algorithm for concatenation was first devised by Clark A. Crane: Assume that height$(L_1) \geq$ height(L_2); the other case is similar. Delete the first node of L_2, calling it the *juncture node* J, and let L_2' be the new tree for $L_2 \setminus \{J\}$. Now go down the right links of L_1 until reaching a node P such that

$$\text{height}(P) - \text{height}(L_2') = 0 \text{ or } 1;$$

this is always possible, since the height changes by 1 or 2 each time we go down one level. Then replace  by



and proceed to adjust L_1 as if the new node J had just been inserted by Algorithm A.

Crane also solved the more difficult inverse problem, to *split* a list into two parts whose concatenation would be the original list. Consider, for example, the problem of splitting the list in Fig. 20 to obtain two lists, one containing $\{$`A`$, \ldots,$ `I`$\}$ and the other containing $\{$`J`$, \ldots,$ `Q`$\}$; a major reassembly of the subtrees is required. In general, when we want to split a tree at some given node P, the path to P will be something like that in Fig. 25. We wish to construct a left tree that contains the nodes of $\alpha_1, P_1, \alpha_4, P_4, \alpha_6, P_6, \alpha_7, P_7, \alpha, P$ in symmetric order, and a right tree that contains $\beta, P_8, \beta_8, P_5, \beta_5, P_3, \beta_3, P_2, \beta_2$. This can be done by a sequence of concatenations: First insert P at the right of α, then concatenate β with β_8 using P_8 as juncture node, concatenate α_7 with αP using P_7 as juncture node, α_6 with $\alpha_7 P_7 \alpha P$ using P_6, $\beta P_8 \beta_8$ with β_5 using P_5, etc.; the nodes $P_8, P_7, \ldots, P_1$ on the path to P are used as juncture nodes. Crane proved that this splitting algorithm takes only $O(\log N)$ units of time, when the original tree contains N nodes; the essential reason is that concatenation using a given juncture node takes $O(k)$ steps, where k is the difference in heights between the

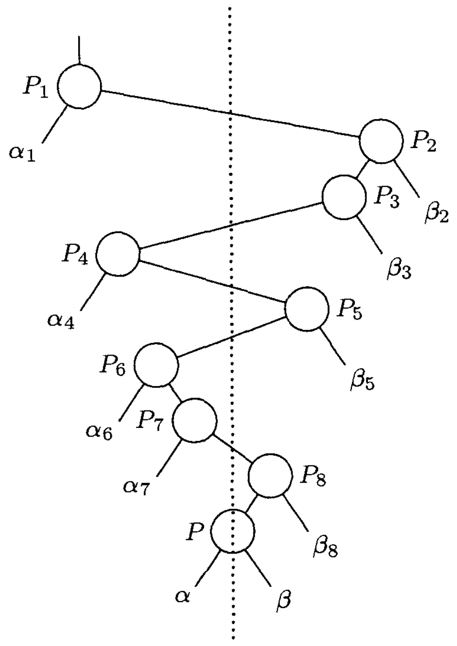


Fig. 25. The problem of splitting a list.

trees being concatenated, and the values of k that must be summed essentially form a telescoping series for both the left and right trees being constructed.

All of these algorithms can be used with either KEY or RANK fields or both (although in the case of concatenation the keys of L_2 must all be greater than the keys of L_1. For general purposes it is often preferable to use a *triply linked tree*, with UP links as well as LLINKs and RLINKs, together with a new one-bit field that specifies whether a node is the left or right child of its parent. The triply linked tree representation simplifies the algorithms slightly, and allows us to specify nodes in the tree without explicitly tracing the path to that node; we can write a subroutine to delete NODE(P), given P, or to delete the node that follows NODE(P) in symmetric order, or to find the list containing NODE(P), etc. In the deletion algorithm for triply linked trees it is unnecessary to construct the list (16), since the UP links provide the information we need. Of course, a triply linked tree requires us to change a few more links when insertions, deletions, and rotations are being performed. The use of a triply linked tree instead of a doubly linked tree is analogous to the use of two-way linking instead of one-way: We can start at any point and go either forward or backward. A complete description of list algorithms based on triply linked balanced trees appears in Clark A. Crane's Ph.D. thesis (Stanford University, 1972).

Alternatives to AVL trees. Many other ways have been proposed to organize trees so that logarithmic accessing time is guaranteed. For example, C. C. Foster [*CACM* **16** (1973), 513–517] considered the binary trees that arise when we allow the height difference of subtrees to be at most k. Such structures have been called HB(k) (meaning "height-balanced"), so that ordinary balanced trees represent the special case HB(1).

The interesting concept of *weight-balanced trees* has been studied by J. Nievergelt, E. Reingold, and C. K. Wong. Instead of considering the height of trees, they stipulate that the subtrees of all nodes must satisfy

$$\sqrt{2}-1 < \frac{\text{left weight}}{\text{right weight}} < \sqrt{2}+1, \tag{17}$$

where the left and right weights count the number of *external* nodes in the left and right subtrees, respectively. It is possible to show that weight balance can be maintained under insertion, using only single and double rotations for rebalancing as in Algorithm A (see exercise 25). However, it may be necessary to do many rebalancings during a single insertion. It is possible to relax the conditions of (17), decreasing the amount of rebalancing at the expense of increased search time.

Weight-balanced trees may seem at first glance to require more memory than plain balanced trees, but in fact they sometimes require slightly less! If we already have a RANK field in each node, for the linear list representation, this is precisely the left weight, and it is possible to keep track of the corresponding right weights as we move down the tree. However, it appears that the bookkeeping required for maintaining weight balance takes more time than Algorithm A, and the elimination of two bits per node is probably not worth the trouble.

Why don't you pair 'em up in threes?
— attributed to YOGI BERRA (c. 1970)

Another interesting alternative to AVL trees, called "2-3 trees," was introduced by John Hopcroft in 1970 [see Aho, Hopcroft, and Ullman, *The Design and Analysis of Computer Algorithms* (Reading, Mass.: Addison–Wesley, 1974), Chapter 4]. The idea is to have either 2-way or 3-way branching at each node, and to stipulate that all external nodes appear on the same level. Every internal node contains either one or two keys, as shown in Fig. 26.

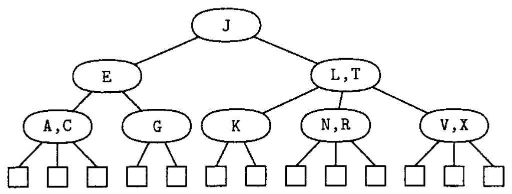


Fig. 26. A 2-3 tree.

Insertion into a 2-3 tree is somewhat easier to explain than insertion into an AVL tree: If we want to put a new key into a node that contains just one key, we simply insert it as the second key. On the other hand, if the node already contains two keys, we divide it into two one-key nodes, and insert the middle key into the parent node. This may cause the parent node to be divided in a similar way, if it already contains two keys. Figure 27 shows the process of inserting a new key into the 2-3 tree of Fig. 26.

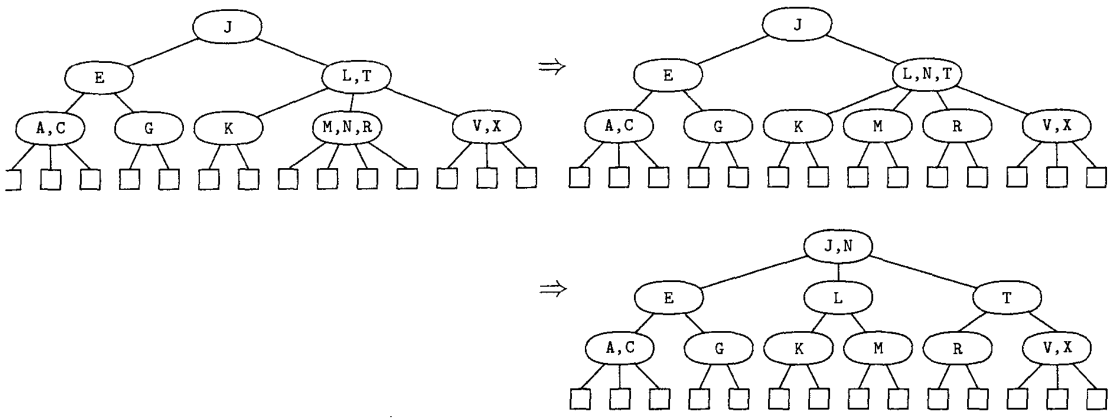


Fig. 27. Inserting the new key "M" into the 2-3 tree of Fig. 26.

Hopcroft observed that deletion, concatenation, and splitting can all be done with 2-3 trees, in a reasonably straightforward manner analogous to the corresponding operations with AVL trees.

R. Bayer [*Proc. ACM–SIGFIDET Workshop* (1971), 219–235] proposed an interesting binary tree representation for 2-3 trees. See Fig. 28, which shows the binary tree representation of Fig. 26; one bit in each node is used to distinguish "horizontal" RLINKs from "vertical" ones. Note that the keys of the tree appear from left to right in symmetric order, just as in any binary search tree. It turns out that the transformations we need to perform on such a binary tree, while inserting a new key as in Fig. 27, are precisely the single and double rotations used while inserting a new key into an AVL tree, although we need just one version of each rotation, not the left-right reflections needed by Algorithms A and C.

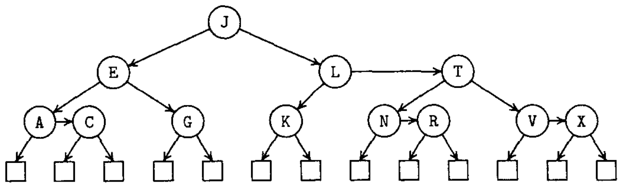


Fig. 28. The 2-3 tree of Fig. 26 represented as a binary search tree.

Elaboration of these ideas has led to many additional flavors of balanced trees, most notably the *red-black trees*, also called symmetric binary B-trees or half-balanced trees [R. Bayer, *Acta Informatica* **1** (1972), 290–306; L. Guibas and R. Sedgewick, *FOCS* **19** (1978), 8–21; H. J. Olivié, *RAIRO Informatique Théorique* **16** (1982), 51–71; R. E. Tarjan, *Inf. Proc. Letters* **16** (1983), 253–257; T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms* (MIT Press, 1989), Chapter 14; R. Sedgewick, *Algorithms in C* (Addison–Wesley, 1997), §13.4]. There is also a strongly related family called hysterical B-trees or (a, b)-trees, notably $(2, 4)$-trees [D. Maier and S. C. Salveter, *Inf. Proc. Letters* **12** (1981), 199–202; S. Huddleston and K. Mehlhorn, *Acta Informatica* **17** (1982), 157–184].

When some keys are accessed much more frequently than others, we want the important ones to be relatively close to the root, as in the optimum binary search trees of Section 6.2.2. Dynamic trees that make it possible to maintain weighted balance within a constant factor of the optimum, called *biased trees*, have been developed by S. W. Bent, D. D. Sleator, and R. E. Tarjan, *SICOMP* **14** (1985), 545–568; J. Feigenbaum and R. E. Tarjan, *Bell System Tech. J.* **62** (1983), 3139–3158. The algorithms are, however, quite complicated.

A much simpler self-adjusting data structure called a *splay tree* was developed subsequently by D. D. Sleator and R. E. Tarjan [*JACM* **32** (1985), 652–686], based on ideas like the move-to-front and transposition heuristics discussed in Section 6.1; similar techniques had previously been explored by B. Allen and I. Munro [*JACM* **25** (1978), 526–535] and by J. Bitner [*SICOMP* **8** (1979), 82–110]. Splay trees, like the other kinds of balanced trees already mentioned, support the operations of concatenation and splitting as well as insertion and deletion, and in a particularly simple way. Moreover, the time needed to access data in a splay tree is known to be at most a small constant multiple of the access time of a statically optimum tree, when amortized over any series of operations. Indeed, Sleator and Tarjan conjectured that the total splay tree access time is at most a constant multiple of the optimum time to access data and to perform rotations dynamically by any binary tree algorithm whatsoever.

Randomization leads to methods that appear to be even simpler and faster than splay trees. Jean Vuillemin [*CACM* **23** (1980), 229–239] introduced *Cartesian trees*, in which every node has two keys (x, y). The x parts are ordered from left to right as in binary search trees; the y parts are ordered from top to bottom as in the priority queue trees of Section 5.2.3. C. R. Aragon and R. G. Seidel gave this data structure the more colorful name *treap*, because it neatly combines the notions of trees and heaps. Exactly one treap can be formed with n given key pairs (x_1, y_1), ..., (x_n, y_n), if the x's and y's are distinct. One way to obtain it is to insert the x's by Algorithm 6.2.2T according to the order of the y's; but there is also a simple algorithm that inserts any new key pair directly into any treap. Aragon and Seidel observed [*FOCS* **30** (1989), 540–546] that if the x's are ordinary keys while the y's are chosen at random, we can be sure that the treap has the shape of a random binary search tree. In particular, a treap with random y values will always be reasonably well balanced, except with exponentially small probability (see exercise 5.2.2–42). Aragon and Seidel also showed that treaps can readily be biased so that, for example, a key x with relative frequency f will appear suitably near the root when it is associated with $y = U^{1/f}$, where U is a random number between 0 and 1. Treaps performed consistently better than splay trees in some experiments conducted by D. E. Knuth relating to the calculation of convex hulls [*Lecture Notes in Comp. Sci.* **606** (1992), 53–55].

A new Section 6.2.5 devoted to randomized data structures is planned for the next edition of the present book. It will discuss "skip lists" [W. Pugh, CACM **33** *(1990), 668–676] and "randomized binary search trees" [S. Roura and C. Martínez, Lecture Notes in Comp. Sci.* **1136** *(1996), 91–106] as well as treaps.*

EXERCISES

1. [*01*] In Case 2 of (1), why isn't it a good idea to restore the balance by simply interchanging the left subtrees of A and B?

2. [*16*] Explain why the tree has gotten one level higher if we reach step A7 with $\texttt{B(S)} = 0$.

▶ **3.** [*M25*] Prove that a balanced tree with N internal nodes never contains more than $(\phi - 1)N \approx 0.61803N$ nodes whose balance factor is nonzero.

4. [*M22*] Prove or disprove: Among all balanced trees with $F_{h+1} - 1$ internal nodes, the Fibonacci tree of order h has the greatest internal path length.

▶ **5.** [*M25*] Prove or disprove: If Algorithm A is used to insert the keys $K_2, \ldots, K_N$ successively in increasing order into a tree that initially contains only the single key K_1, where $K_1 < K_2 < \cdots < K_N$, then the tree produced is always *optimum* (that is, it has minimum internal path length over all N-node binary trees).

6. [*M21*] Prove that Eq. (5) defines the generating function for balanced trees of height h.

7. [*M27*] (N. J. A. Sloane and A. V. Aho.) Prove the remarkable formula (9) for the number of balanced trees of height h. [*Hint:* Let $C_n = B_n + B_{n-1}$, and use the fact that $\log(C_{n+1}/C_n^2)$ is exceedingly small for large n.]

8. [*M24*] (L. A. Khizder.) Show that there is a constant β such that $B_h'(1)/B_h(1) = 2^h\beta - 1 + O(2^h/B_{h-1})$ as $h \to \infty$.

9. [*HM44*] What is the asymptotic number of balanced binary trees with n internal nodes, $\sum_{h\geq 0} B_{nh}$? What is the asymptotic average height, $\sum_{h\geq 0} hB_{nh}/\sum_{h\geq 0} B_{nh}$?

▶ **10.** [*27*] (R. C. Richards.) Show that the shape of a balanced tree can be constructed uniquely from the list of its balance factors $\texttt{B(1)B(2)}\ldots\texttt{B(}N\texttt{)}$ in symmetric order.

11. [*M24*] (Mark R. Brown.) Prove that when $n \geq 6$ the average number of external nodes of each of the types `+A`, `-A`, `++B`, `+-B`, `-+B`, `--B` is exactly $(n+1)/14$, in a random balanced tree of n internal nodes constructed by Algorithm A.

▶ **12.** [*24*] What is the maximum possible running time of Program A when the eighth node is inserted into a balanced tree? What is the minimum possible running time for this insertion?

13. [*05*] Why is it better to use `RANK` fields as defined in the text, instead of simply to store the index of each node as its key (calling the first node "1", the second node "2", and so on)?

14. [*11*] Could Algorithms 6.2.2T and 6.2.2D be adapted to work with linear lists, using a `RANK` field, just as the balanced tree algorithms of this section have been so adapted?

15. [*18*] (C. A. Crane.) Suppose that an ordered linear list is being represented as a binary tree, with both `KEY` and `RANK` fields in each node. Design an algorithm that searches the tree for a given key, K, and determines the position of K in the list; that is, it finds the number m such that K is the mth smallest key.

▶ **16.** [*20*] Draw the balanced tree that is obtained after node `E` and the root node `F` are deleted from Fig. 20, using the deletion algorithm suggested in the text.

▶ **17.** [*21*] Draw the balanced trees that are obtained after the Fibonacci tree (12) is concatenated (a) to the right, (b) to the left, of the tree in Fig. 20, using the concatenation algorithm suggested in the text.

18. [*22*] Draw the balanced trees that are obtained after Fig. 20 is split into two parts $\{\mathtt{A}, \ldots, \mathtt{I}\}$ and $\{\mathtt{J}, \ldots, \mathtt{Q}\}$, using the splitting algorithm suggested in the text.

▶ **19.** [*26*] Find a way to transform a given balanced tree so that the balance factor at the root is not -1. Your transformation should preserve the symmetric order of the nodes; and it should produce another balanced tree in $O(1)$ units of time, regardless of the size of the original tree.

20. [*40*] Explore the idea of using the restricted class of balanced trees whose nodes all have balance factors of 0 or $+1$. (Then the length of the B field can be reduced to one bit.) Is there a reasonably efficient insertion procedure for such trees?

▶ **21.** [*30*] (*Perfect balancing.*) Design an algorithm to construct N-node binary trees that are optimum in the sense of exercise 5. Your algorithm should use $O(N)$ steps and it should be "online," in the sense that it inputs the nodes one by one in increasing order and builds partial trees as it goes, without knowing the final value of N in advance. (It would be appropriate to use such an algorithm when restructuring a badly balanced tree, or when merging the keys of two trees into a single tree.)

22. [*M20*] What is the analog of Theorem A, for weight-balanced trees?

23. [*M20*] (E. Reingold.) Demonstrate that there is no simple relation between height-balanced trees and weight-balanced trees:

a) Prove that there exist height-balanced trees that have an arbitrarily small ratio (left weight)/(right weight) in the sense of (17).

b) Prove that there exist weight-balanced trees that have an arbitrarily large difference between left and right subtree heights.

24. [*M22*] (E. Reingold.) Prove that if we strengthen condition (17) to

$$\frac{1}{2} < \frac{\text{left weight}}{\text{right weight}} < 2,$$

the only binary trees that satisfy this condition are perfectly balanced trees with $2^n - 1$ internal nodes. (In such trees, the left and right weights are exactly equal at all nodes.)

25. [*27*] (J. Nievergelt, E. Reingold, C. Wong.) Show that it is possible to design an insertion algorithm for weight-balanced trees so that condition (17) is preserved, making at most $O(\log N)$ rotations per insertion.

26. [*40*] Explore the properties of balanced t-ary trees, for $t > 2$.

▶ **27.** [*M23*] Estimate the maximum number of comparisons needed to search in a 2-3 tree with N internal nodes.

28. [*41*] Prepare efficient implementations of 2-3 tree algorithms.

29. [*M47*] Analyze the average behavior of 2-3 trees under random insertions.

30. [*26*] (E. McCreight.) Section 2.5 discusses several strategies for dynamic storage allocation, including best-fit (choosing an available area as small as possible from among all those that fulfill the request) and first-fit (choosing the available area with lowest address among all those that fulfill the request). Show that if the available space is linked together as a balanced tree in an appropriate way, it is possible to do (a) best-fit (b) first-fit allocation in only $O(\log n)$ units of time, where n is the number of available areas. (The algorithms given for those methods in Section 2.5 take order n steps.)

31. [*34*] (M. L. Fredman, 1975.) Invent a representation of linear lists with the property that insertion of a new item between positions $m - 1$ and m, given m, takes $O(\log m)$ units of time.

32. [*M27*] Given two n-node binary trees, T and T', let us say that $T \preceq T'$ if T' can be obtained from T by a sequence of zero or more rotations to the right. Prove that $T \preceq T'$ if and only if $r_k \le r'_k$ for $1 \le k \le n$, where r_k and r'_k denote the respective sizes of the right subtrees of the kth nodes of T and T' in symmetric order.

▶ **33.** [*25*] (A. L. Buchsbaum.) Explain how to encode the balance factors of an AVL tree implicitly, thus saving two bits per node, at the expense of additional work when the tree is accessed.

Samuel considered the nation of Israel, tribe by tribe,
and the tribe of Benjamin was picked by lot.
Then he considered the tribe of Benjamin, family by family,
and the family of Matri was picked by lot.
Then he considered the family of Matri, man by man,
and Saul son of Kish was picked by lot.
But when they looked for Saul he could not be found.
— *1 Samuel 10:20–21*

6.2.4. Multiway Trees

The tree search methods we have been discussing were developed primarily for internal searching, when we want to look at a table that is contained entirely within a computer's high-speed internal memory. Let's now consider the problem of *external* searching, when we want to retrieve information from a very large file that appears on direct access storage units such as disks or drums. (An introduction to disks and drums appears in Section 5.4.9.)

Tree structures lend themselves nicely to external searching, if we choose an appropriate way to represent the tree. Consider the large binary search tree shown in Fig. 29, and imagine that it has been stored in a disk file. (The LLINKs and RLINKs of the tree are now disk addresses instead of internal memory addresses.) If we search this tree in a naïve manner, simply applying the algorithms we have learned for internal tree searching, we will have to make about $\lg N$ disk accesses before our search is complete. When N is a million, this means we will need 20 or so seeks. But suppose we divide the table into 7-node "pages," as shown by the dotted lines in Fig. 29; if we access one page at a time, we need only about one third as many seeks, so the search goes about three times as fast!

Grouping the nodes into pages in this way essentially changes the tree from a binary tree to an octonary tree, with 8-way branching at each page-node. If we let the pages be still larger, with 128-way branching after each disk access, we can find any desired key in a million-entry table after looking at only three pages. We can keep the root page in the internal memory at all times, so that only two references to the disk are required even though the internal memory never needs to hold more than 254 keys at any time.

Of course we don't want to make the pages arbitrarily large, since the internal memory size is limited and also since it takes a long time to read a large page. For example, suppose that it takes $72.5 + 0.05m$ milliseconds to read a page that allows m-way branching. The internal processing time per page will

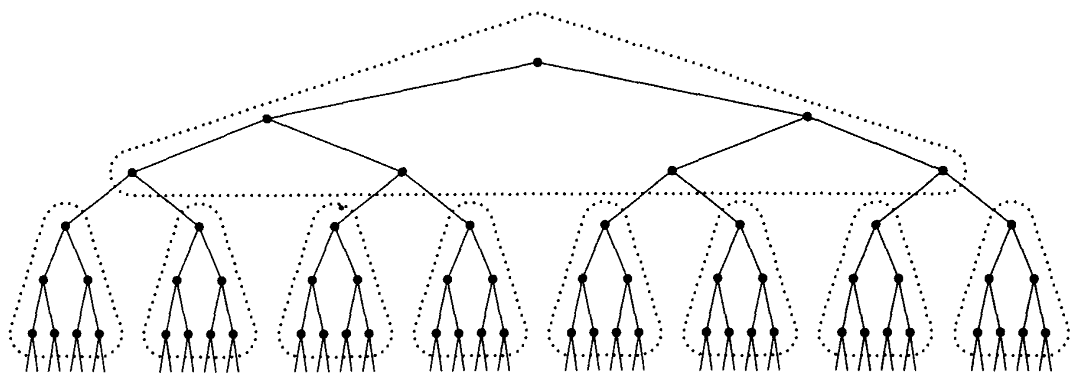


Fig. 29. A large binary search tree can be divided into "pages."

be about $a + b \lg m$, where a is small compared to 72.5 ms, so the total amount of time needed for searching a large table is approximately proportional to $\lg N$ times

$$(72.5 + 0.05m)/\lg m + b.$$

This quantity achieves a minimum when $m \approx 307$; actually the minimum is very "broad" — a nearly optimum value is achieved for all m between 200 and 500. In practice there will be a similar range of good values for m, based on the characteristics of particular external memory devices and on the length of the records in the table.

W. I. Landauer [*IEEE Trans.* **EC-12** (1963), 863–871] suggested building an m-ary tree by requiring level l to become nearly full before anything is allowed to appear on level $l + 1$. This scheme requires a rather complicated rotation method, since we may have to make major changes throughout the tree just to insert a single new item; Landauer was assuming that we need to search for items in the tree much more often than we need to insert or delete them.

When a file is stored on disk, and is subject to comparatively few insertions and deletions, a three-level tree is appropriate, where the first level of branching determines what cylinder is to be used, the second level of branching determines the appropriate track on that cylinder, and the third level contains the records themselves. This method is called *indexed-sequential* file organization [see *JACM* **16** (1969), 569–571].

R. Muntz and R. Uzgalis [*Proc. Princeton Conf. on Inf. Sciences and Systems* **4** (1970), 345–349] suggested modifying the tree search and insertion method, Algorithm 6.2.2T, so that all insertions go onto nodes belonging to the same page as their parent node, whenever possible; if that page is full, a new page is started, whenever possible. If the number of pages is unlimited, and if the data arrives in random order, it can be shown that the average number of page accesses is approximately $H_N/(H_m - 1)$, only slightly more than we would obtain in the best possible m-ary tree. (See exercise 8.)

B-trees. A new approach to external searching by means of multiway tree branching was discovered in 1970 by R. Bayer and E. McCreight [*Acta Informa-*

tica **1** (1972), 173–189], and independently at about the same time by M. Kaufman [unpublished]. Their idea, based on a versatile new kind of data structure called a *B-tree*, makes it possible both to search and to update a large file with guaranteed efficiency, in the worst case, using comparatively simple algorithms.

A *B-tree of order m* is a tree that satisfies the following properties:

i) Every node has at most m children.

ii) Every node, except for the root and the leaves, has at least $m/2$ children.

iii) The root has at least 2 children (unless it is a leaf).

iv) All leaves appear on the same level, and carry no information.

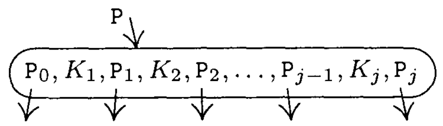
v) A nonleaf node with k children contains $k-1$ keys.

(As usual, a "leaf" is a terminal node, one with no children. Since the leaves carry no information, we may regard them as external nodes that aren't really in the tree, so that Λ is a pointer to a leaf.)

Figure 30 shows a B-tree of order 7. Each node (except for the root and the leaves) has between $\lceil 7/2 \rceil$ and 7 children, so it contains 3, 4, 5, or 6 keys. The root node is allowed to contain from 1 to 6 keys; in this case it has 2. All of the leaves are at level 3. Notice that (a) the keys appear in increasing order from left to right, using a natural extension of the concept of symmetric order; and (b) the number of leaves is exactly one greater than the number of keys.

B-trees of order 1 or 2 are obviously uninteresting, so we will consider only the case $m \geq 3$. The 2-3 trees defined at the close of Section 6.2.3 are equivalent to B-trees of order 3. (Bayer and McCreight considered only the case that m is odd; some authors consider a B-tree of order m to be what we are calling a B-tree of order $2m+1$.)

A node that contains j keys and $j+1$ pointers can be represented as


P
$\mathrm{P}_0, K_1, \mathrm{P}_1, K_2, \mathrm{P}_2, \ldots, \mathrm{P}_{j-1}, K_j, \mathrm{P}_j$

(1)

where $K_1 < K_2 < \cdots < K_j$ and P_i points to the subtree for keys between K_i and K_{i+1}. Therefore searching in a B-tree is quite straightforward: After node (1) has been fetched into the internal memory, we search for the given argument among the keys $K_1, K_2, \ldots, K_j$. (When j is large, we probably do a binary search; but when j is smallish, a sequential search is best.) If the search is successful, we have found the desired key; but if the search is unsuccessful because the argument lies between K_i and K_{i+1}, we fetch the node indicated by P_i and continue the process. The pointer P_0 is used if the argument is less than K_1, and P_j is used if the argument is greater than K_j. If $\mathrm{P}_i = \Lambda$, the search is unsuccessful.

The nice thing about B-trees is that insertion is also quite simple. Consider Fig. 30, for example; every leaf corresponds to a place where a new insertion might happen. If we want to insert the new key 337, we simply change the

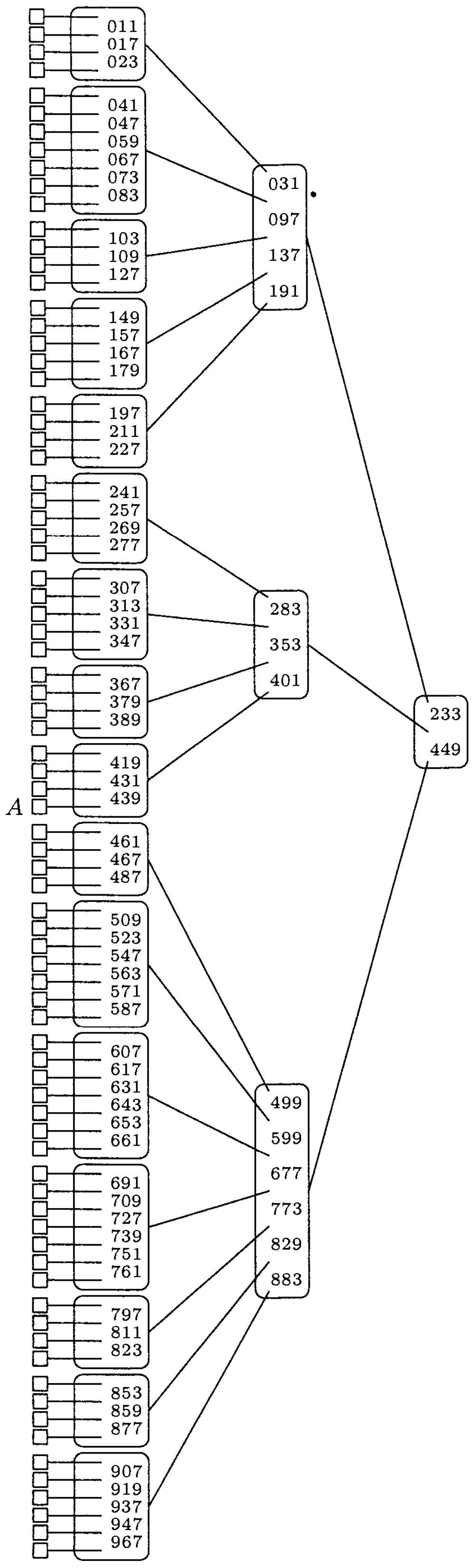
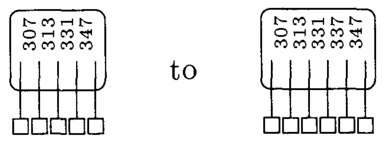


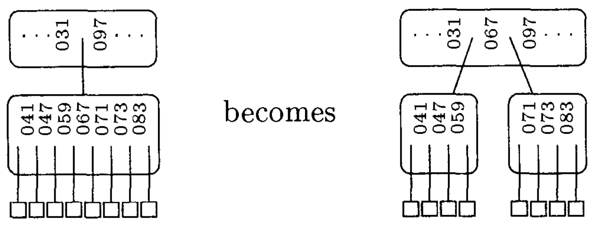
Fig. 30. A B-tree of order 7, with all leaves on level 3. Every node contains 3, 4, 5, or 6 keys. The leaf that precedes key 449 has been marked A; see (8).

appropriate node from



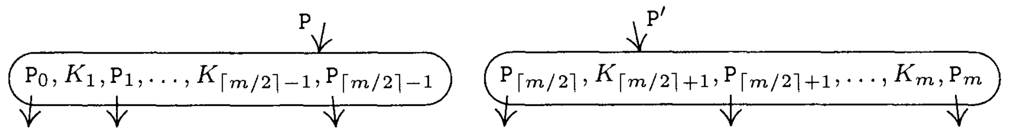
. (2)

On the other hand, if we want to insert the new key 071, there is no room since the corresponding node on level 2 is already "full." This case can be handled by splitting the node into two parts, with three keys in each part, and passing the middle key up to level 1:



. (3)

In general, if we want to insert a new item into a B-tree of order m, when all the leaves are at level l, we insert the new key into the appropriate node on level $l-1$. If that node now contains m keys, so that it has the form (1) with $j = m$, we split it into two nodes


P
P′
$P_0, K_1, P_1, \ldots, K_{\lceil m/2\rceil-1}, P_{\lceil m/2\rceil-1}$
$P_{\lceil m/2\rceil}, K_{\lceil m/2\rceil+1}, P_{\lceil m/2\rceil+1}, \ldots, K_m, P_m$
(4)

and insert the key $K_{\lceil m/2\rceil}$ into the parent of the original node. (Thus the pointer P in the parent node is replaced by the sequence P, $K_{\lceil m/2\rceil}$, P′.) This insertion may cause the parent node to contain m keys, and if so, it should be split in the same way. (Fig. 27 in the previous section illustrates the case $m = 3$.) If we need to split the root node, which has no parent, we simply create a new root node containing the single key $K_{\lceil m/2\rceil}$; the tree gets one level taller in this case.

This insertion procedure neatly preserves all of the B-tree properties; in order to appreciate the full beauty of the idea, the reader should work exercise 1. The tree essentially grows up from the top, instead of down from the bottom, since it gains in height only when the root splits.

Deletion from B-trees is only slightly more complicated than insertion (see exercise 6).

Upper bounds on the running time. Let us now see how many nodes have to be accessed in the worst case, while searching in a B-tree of order m. Suppose that there are N keys, and that the $N+1$ leaves appear on level l. Then the number of nodes on levels $1, 2, 3, \ldots$ is at least 2, $2\lceil m/2\rceil$, $2\lceil m/2\rceil^2$, $\ldots$; hence

$$N+1 \geq 2\lceil m/2\rceil^{l-1}. \qquad (5)$$

In other words,

$$l \leq 1 + \log_{\lceil m/2\rceil}\left(\frac{N+1}{2}\right); \qquad (6)$$

this means, for example, that if $N = 1{,}999{,}998$ and $m = 199$, then l is at most 3. Since we need to access at most l nodes during a search, this formula guarantees that the running time is quite small.

When a new key is being inserted, we may have to split as many as l nodes. However, the average number of nodes that need to be split is much less, since the total number of splittings that occur while the entire tree is being constructed is just the total number of internal nodes in the tree, minus l. If there are p internal nodes, there are at least $1 + (\lceil m/2\rceil - 1)(p - 1)$ keys; hence

$$p \le 1 + \frac{N-1}{\lceil m/2\rceil - 1}. \tag{7}$$

It follows that the average number of times we need to split a node while building a tree of N keys is less than $1/(\lceil m/2\rceil - 1)$ split per insertion.

Refinements and variations. There are several ways to improve upon the basic B-tree structure defined above, by breaking the rules a little.

In the first place, we note that all of the pointers in the level $l - 1$ nodes are Λ, and none of the pointers in the other levels are Λ. This often represents a significant amount of wasted space, so we can save both time and space by eliminating all the Λ's and using a different value of m for all of the "bottom" nodes. This use of two different m's does not foul up the insertion algorithm, since both halves of a node that is being split remain on the same level as the original node. We could in fact define a generalized B-tree of orders $m_1, m_2, m_3, \ldots$ by requiring all nonroot nodes on level $l - k$ to have between $m_k/2$ and m_k children; such a B-tree has different m's on each level, yet the insertion algorithm still works essentially as before.

To carry the idea in the preceding paragraph even further, we might use a completely different node format in each level of the tree, and we might also store information in the leaves. Sometimes the keys form only a small part of the records in a file, and in such cases it is a mistake to store the entire records in the branch nodes near the root of the tree; this would make m too small for efficient multiway branching.

We can therefore reconsider Fig. 30, imagining that all the records of the file are now stored in the leaves, and that only a few of the keys have been duplicated in the branch nodes. Under this interpretation, the leftmost leaf contains all records whose key is ≤ 011; the leaf marked A contains all records whose key satisfies

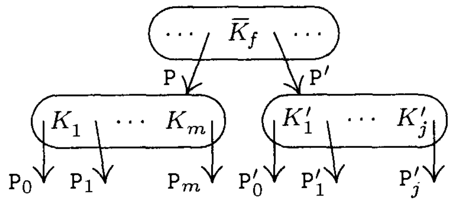
$$439 < K \le 449; \tag{8}$$

and so on. Under this interpretation the leaf nodes grow and split just as the branch nodes do, except that a record is never passed up from a leaf to the next level. Thus the leaves are always at least half filled to capacity. A new key enters the nonleaf part of the tree whenever a leaf splits. If each leaf is linked to its successor in symmetric order, we gain the ability to traverse the file both sequentially and randomly in an efficient and convenient manner. This variant has become known as a B^+-tree.

Some calculations by S. P. Ghosh and M. E. Senko [*JACM* **16** (1969), 569–579] suggest that it might be a good idea to make the leaves fairly large, say up to about 10 consecutive pages long. By linear interpolation in the known range of keys for each leaf, we can guess which of the 10 pages probably contains a given search argument. If our guess is wrong, we lose time, but experiments indicate that this loss might be less than the time we save by decreasing the size of the tree.

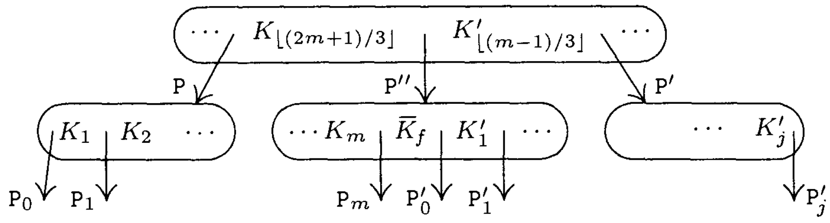
T. H. Martin [unpublished] has pointed out that the idea underlying B-trees can be used also for *variable-length* keys. We need not put bounds $[m/2\,..\,m]$ on the number of children of each node; instead we can say merely that each node should be at least about half full of data. The insertion and splitting mechanism still works fine, even though the exact number of keys per node depends on whether the keys are long or short. However, the keys shouldn't be allowed to get extremely long, or they can mess things up. (See exercise 5.)

Another important modification to the basic B-tree scheme is the idea of *overflow* introduced by Bayer and McCreight. The idea is to improve the insertion algorithm by resisting its temptation to split nodes so often; a local rotation is used instead. Suppose we have a node that is over-full because it contain m keys and $m+1$ pointers; instead of splitting it, we can look first at its sibling node on the right, which has say j keys and $j+1$ pointers. In the parent node there is a key $\overline{K}_f$ that separates the keys of the two siblings; schematically,



(9)

If $j < m-1$, a simple rearrangement makes splitting unnecessary: We leave $\lfloor (m+j)/2 \rfloor$ keys in the left node, we replace $\overline{K}_f$ by $K_{\lfloor (m+j)/2\rfloor+1}$ in the parent node, and we put the $\lceil (m+j)/2 \rceil$ remaining keys (including $\overline{K}_f$) and the corresponding pointers into the right node. Thus the full node "flows over" into its sibling node. On the other hand, if the sibling node is already full ($j = m-1$), we can split *both* of the nodes, making three nodes each about two-thirds full, containing, respectively, $\lfloor (2m-2)/3 \rfloor$, $\lfloor (2m-1)/3 \rfloor$, and $\lfloor 2m/3 \rfloor$ keys:



(10)

If the original node has no right sibling, we can look at its left sibling in essentially the same way. (If the original node has both a right and a left sibling, we could even refrain from splitting off a new node unless *both* left and right siblings are full.) Finally if the original node to be split has no siblings at all, it must be the root; we can change the definition of B-tree, allowing the root to contain as many as $2\lfloor(2m-2)/3\rfloor$ keys, so that when the root splits it produces two nodes of $\lfloor(2m-2)/3\rfloor$ keys each.

The effect of all the technicalities in the preceding paragraph is to produce a superior breed of tree, say a B^*-tree of order m, which can be defined as follows:

i) Every node except the root has at most m children.

ii) Every node, except for the root and the leaves, has at least $(2m-1)/3$ children.

iii) The root has at least 2 and at most $2\lfloor(2m-2)/3\rfloor+1$ children.

iv) All leaves appear on the same level.

v) A nonleaf node with k children contains $k-1$ keys.

The important change is condition (ii), which asserts that we utilize at least two-thirds of the available space in every node. This change not only uses space more efficiently, it also makes the search process faster, since we may replace $\lceil m/2\rceil$ by $\lceil(2m-1)/3\rceil$ in (6) and (7). However, the insertion process gets slower, because nodes tend to need more attention as they fill up; see B. Zhang and M. Hsu, *Acta Informatica* **26** (1989), 421–438, for an approximate analysis of the tradeoffs involved.

At the other extreme, it is sometimes better to let nodes become less than half full in a tree that changes quite frequently, especially if insertions tend to outnumber deletions. This situation has been analyzed by T. Johnson and D. Shasha, *J. Comput. Syst. Sci.* **47** (1993), 45–76.

Perhaps the reader has been skeptical of B-trees because the degree of the root can be as low as 2. Why should we waste a whole disk access on merely a 2-way decision?! A simple buffering scheme, called *least-recently-used page replacement*, overcomes this objection; we can keep several bufferloads of information in the internal memory, so that input commands can be avoided when the corresponding page is already present. Under this scheme, the algorithms for searching or insertion issue "virtual read" commands that are translated into actual input instructions only when the necessary page is not in memory; a subsequent "release" command is issued when the buffer has been read and possibly modified by the algorithm. When an actual read is required, the buffer that has least recently been released is chosen; we write out that buffer, if its contents have changed since they were read in, then we read the desired page into the chosen buffer.

Since the number of levels in the tree is generally small compared to the number of buffers, this paging scheme will ensure that the root page is always present in memory; and if the root has only 2 or 3 children, the first-level pages will almost surely stay there too. Any pages that might need to be split during an insertion are automatically present in memory when they are needed, because they will be remembered from the immediately preceding search.

Experiments by E. McCreight have shown that this policy is quite successful. For example, he found that with 10 buffers and $m=121$, the process of inserting

100,000 keys in ascending order required only 22 actual read commands, and only 857 actual write commands; thus most of the activity took place in the internal memory. Furthermore the tree contained only 835 nodes, just one higher than the minimum possible value $\lceil 100000/(m-1)\rceil = 834$; thus the storage utilization was nearly 100 percent. For this experiment he used the overflow technique, but with only 2-way node splitting as in (4), not 3-way splitting as in (10). (See exercise 3.)

In another experiment, again with 10 buffers and $m = 121$ and the overflow technique, he inserted 5000 keys into an initially empty tree, in *random* order; this produced a 2-level tree with 48 nodes (87 percent storage utilization), after making 2762 actual reads and 2739 actual writes. Then 1000 random searches required 786 actual reads. The same experiment *without* the overflow feature produced a 2-level tree with 62 nodes (67 percent storage utilization), after making 2743 actual reads and 2800 actual writes; 1000 subsequent random searches required 836 actual reads. This shows not only that the paging scheme is effective but also that it is wise to handle overflows locally before deciding to split a node.

Andrew Yao has proved that the average number of nodes after random insertions without the overflow feature will be

$$N/(m\ln 2) + O(N/m^2),$$

for large N and m, so the storage utilization will be approximately $\ln 2 = 69.3$ percent [*Acta Informatica* **9** (1978), 159–170]. See also the more detailed analyses by B. Eisenbarth, N. Ziviani, G. H. Gonnet, K. Mehlhorn, and D. Wood, *Information and Control* **55** (1982), 125–174; R. A. Baeza-Yates, *Acta Informatica* **26** (1989), 439–471.

B-trees became popular soon after they were invented. See, for example, the article by Douglas Comer in *Computing Surveys* **11** (1979), 121–138, 412, which discusses early developments and describes a widely used system called VSAM (Virtual Storage Access Method) developed by IBM Corporation. One of the innovations of VSAM was to replicate blocks on a disk track so that latency time was minimized.

Two of the most interesting developments of the basic B-tree strategy have unfortunately been given almost identical names: "SB-trees" and "SB-trees." The SB-tree of P. E. O'Neil [*Acta Inf.* **29** (1992), 241–265] is designed to minimize disk I/O time by allocating nearby records to the same track or cylinder, maintaining efficiency in applications where many consecutive records need to be accessed at the same time; in this case "SB" is in italic type and the S connotes "sequential." The SB-tree of P. Ferragina and R. Grossi [*STOC* **27** (1995), 693–702; *SODA* **7** (1996), 373–382] is an elegant combination of B-tree structure with the Patricia trees that we will consider in Section 6.3; in this case "SB" is in roman type and the S connotes "string." SB-trees have many applications to large-scale text processing, and they provide a basis for efficient sorting of variable-length strings on disk [see Arge, Ferragina, Grossi, and Vitter, *STOC* **29** (1997), 540–548].

EXERCISES

1. [*10*] What B-tree of order 7 is obtained after the key 613 is inserted into Fig. 30? (Do not use the overflow technique.)

2. [*15*] Work exercise 1, but use the overflow technique, with 3-way splitting as in (10).

▸ **3.** [*23*] Suppose we insert the keys 1, 2, 3, ... in ascending order into an initially empty B-tree of order 101. Which key causes the leaves to be on level 4 for the first time

a) when we use no overflow?

b) when we use overflow and only 2-way splitting as in (4)?

c) when we use a B^*-tree of order 101, with overflow and 3-way splitting as in (10)?

4. [*21*] (Bayer and McCreight.) Explain how to handle insertions into a generalized B-tree so that all nodes except the root and leaves will be guaranteed to have at least $\frac{3}{4}m - \frac{1}{2}$ children.

▸ **5.** [*21*] Suppose that a node represents 1000 character positions of external memory. If each pointer occupies 5 characters, and if the keys are variable in length, between 5 and 50 characters long but always a multiple of 5 characters, what is the minimum number of character positions occupied in a node after it splits during an insertion? (Consider only a simple splitting procedure analogous to that described in the text for fixed-length-key B-trees, without overflowing; move up the key that makes the remaining two parts most nearly equal in size.)

6. [*23*] Design a deletion algorithm for B-trees.

7. [*28*] Design a concatenation algorithm for B-trees (see Section 6.2.3).

▸ **8.** [*HM37*] Consider the generalization of tree insertion suggested by Muntz and Uzgalis, where each page can hold M keys. After N random items have been inserted into such a tree, so that there are $N+1$ external nodes, let $b_{Nk}^{(j)}$ be the probability that an unsuccessful search requires k page accesses and that it ends at an external node whose parent node belongs to a page containing j keys. If $B_N^{(j)}(z) = \sum b_{Nk}^{(j)} z^k$ is the corresponding generating function, prove that we have $B_1^{(j)}(z) = \delta_{j1} z$ and

$$B_N^{(j)}(z) = \frac{N-j-1}{N+1} B_{N-1}^{(j)}(z) + \frac{j+1}{N+1} B_{N-1}^{(j-1)}(z), \qquad \text{for } 1 < j < M;$$

$$B_N^{(1)}(z) = \frac{N-2}{N+1} B_{N-1}^{(1)}(z) + \frac{2z}{N+1} B_{N-1}^{(M)}(z);$$

$$B_N^{(M)}(z) = \frac{N-1}{N+1} B_{N-1}^{(M)}(z) + \frac{M+1}{N+1} B_{N-1}^{(M-1)}(z).$$

Find the asymptotic behavior of $C_N' = \sum_{j=1}^{M} B_N^{(j)\prime}(1)$, the average number of page accesses per unsuccessful search. [*Hint:* Express the recurrence in terms of the matrix

$$W(z) = \begin{pmatrix} -3 & 0 & \ldots & 0 & 2z \\ 3 & -4 & \ldots & 0 & 0 \\ 0 & 4 & \ldots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \ldots & -M-1 & 0 \\ 0 & 0 & \ldots & M+1 & -2 \end{pmatrix},$$

and relate C_N' to an Nth degree polynomial in $W(1)$.]

9. [*22*] Can the B-tree idea be used to retrieve items of a linear list by position instead of by key value? (See Algorithm 6.2.3B.)

▸ **10.** [*35*] Discuss how a large file, organized as a B-tree, can be used for concurrent accessing and updating by a large number of simultaneous users, in such a way that users of different pages rarely interfere with each other.

Little is known, even for otherwise equivalent algorithms,
about the optimization of storage allocation,
minimization of the number of required operations,
and so on. This area of investigation
must draw upon the most powerful resources
of both pure and applied mathematics
for further progress.

— ANTHONY G. OETTINGER (1961)