EXHIBIT D

THE ART OF COMPUTER PROGRAMMING

DONALD E. KNUTH *Stanford University*

ADDISON-WESLEY PUBLISHING COMPANY

GN 005227

Volume 3 / **Sorting and Searching**

THE ART OF COMPUTER PROGRAMMING

Reading, Massachusetts
Menlo Park, California · London · Amsterdam · Don Mills, Ontario · Sydney

This book is in the
**ADDISON-WESLEY SERIES IN
COMPUTER SCIENCE AND INFORMATION PROCESSING**

Consulting Editors
RICHARD S. VARGA and MICHAEL A. HARRISON

ISBN 0-201-03803-X
HIJKLMNOPQ-MA-89876543210

CHAPTER SIX

SEARCHING

Let's look at the record.
—AL SMITH (1928)

This chapter might have been given the more pretentious title, "Storage and Retrieval of Information"; on the other hand, it might simply have been called "Table Look-Up." We are concerned with the process of collecting information in a computer's memory, and with the subsequent recovery of that information as quickly as possible. Sometimes we are confronted with more data than we can really use, and it may be wisest to forget and to destroy most of it; but at other times it is important to retain and organize the given facts in such a way that fast retrieval is possible.

Most of this chapter is devoted to the study of a very simple search problem: how to find the data that has been stored with a given identification. For example, in a numerical application we might want to find $f(x)$, given x and a table of the values of f; in a nonnumerical application, we might want to find the English translation of a given Russian word.

In general, we shall suppose that a set of N records has been stored, and the problem is to locate the appropriate one. As in the case of sorting, we assume that each record includes a special field called its *key*, perhaps because many people spend so much time every day searching for their keys. We generally require the N keys to be distinct, so that each key uniquely identifies its record. The collection of all records is called a *table* or a *file*, where the word "table" is usually used to indicate a small file, and "file" is usually used to indicate a large table. A large file or a group of files is frequently called a *data base*.

Algorithms for searching are presented with a so-called *argument*, K, and the problem is to find which record has K as its key. After the search is complete, two possibilities can arise: Either the search was *successful*, having located the unique record containing K, or it was *unsuccessful*, having determined that K is nowhere to be found. After an unsuccessful search it is sometimes desirable to enter a new record, containing K, into the table; a method which does this is called a "search and insertion" algorithm. Some hardware devices known as "associative memories" solve the search problem automatically, in a way that resembles the functioning of a human brain; but we shall study techniques for searching on a conventional general-purpose digital computer.

Although the goal of searching is to find the information stored in the record associated with K, the algorithms in this chapter generally ignore everything but the keys themselves. In practice we can find the associated data once we have located K; for example, if K appears in location TABLE $+ i$, the associated data (or a pointer to it) might be in location TABLE $+ i + 1$, or in DATA $+ i$, etc. It is therefore convenient to gloss over the details of what should be done after K has been successfully found.

Searching is the most time-consuming part of many programs, and the substitution of a good search method for a bad one often leads to a substantial increase in speed. In fact it is often possible to arrange the data or the data structure so that searching is eliminated entirely, i.e., so that we always know just where to find the information we need. Linked memory is a common way to achieve this; for example, a doubly-linked list makes it unnecessary to search for the predecessor or successor of a given item. Another way to avoid searching occurs if we are allowed to choose the keys freely, since we might as well let them be the numbers $\{1, 2, \ldots, N\}$; then the record containing K can simply be placed in location TABLE $+ K$. Both of these techniques were used to eliminate searching from the topological sorting algorithm discussed in Section 2.2.3. However, there are many cases when a search is necessary (for example, if the objects in the topological sorting algorithm had been given symbolic names instead of numbers), so it is important to have efficient algorithms for searching.

Search methods might be classified in several ways. We might divide them into internal vs. external searching, just as we divided the sorting algorithms of Chapter 5 into internal vs. external sorting. Or we might divide search methods into static vs. dynamic searching, where "static" means that the contents of the table are essentially unchanging (so that it is important to minimize the search time without regard for the time required to set up the table), and "dynamic" means that the table is subject to frequent insertions (and perhaps also deletions). A third possible scheme is to classify search methods according to whether they are based on comparisons between keys or on digital properties of the keys, analogous to the distinction between sorting by comparison and sorting by distribution. Finally we might divide searching into those methods which use the actual keys and those which work with transformed keys.

The organization of this chapter is essentially a combination of the latter two modes of classification. Section 6.1 considers "brute force" sequential methods of search, then Section 6.2 discusses the improvements which can be made based on comparisons between keys, using alphabetic or numeric order to govern the decisions. Section 6.3 treats digital searching, and Section 6.4 discusses an important class of methods called hashing techniques, based on arithmetic transformations of the actual keys. Each of these sections treats both internal and external searching, in both the static and the dynamic case;

and each section points out the relative advantages and disadvantages of the various algorithms.

There is a certain amount of interaction between searching and sorting. For example, consider the following problem:

Given two sets of numbers, $A = \{a_1, a_2, \ldots, a_m\}$ and $B = \{b_1, b_2, \ldots, b_n\}$, determine whether or not $A \subseteq B$.

Three solutions suggest themselves, namely

1. Compare each a_i sequentially with the b_j's until finding a match.
2. Enter the b_j's in a table, then search for each of the a_i.
3. Sort the a's and b's, then make one sequential pass through both files, checking the appropriate condition.

Each of these solutions is attractive for a different range of values of m and n. Solution 1 will take roughly c_1mn units of time, for some constant c_1, and solution 3 will take about $c_2(m \log_2 m + n \log_2 n)$ units, for some (larger) constant c_2. With a suitable hashing method, solution 2 will take roughly $c_3m + c_4n$ units of time, for some (still larger) constants c_3 and c_4. It follows that solution 1 is good for very small m and n, but solution 3 soon becomes better as m and n grow larger. Eventually solution 2 becomes preferable, until n exceeds the internal memory size; then solution 3 is usually again superior until n gets much larger still. Thus we have a situation where sorting is sometimes a good substitute for searching, and searching is sometimes a good substitute for sorting.

More complicated search problems can often be reduced to the simpler case considered here. For example, suppose that the keys are words which might be slightly misspelled; we might want to find the correct record in spite of this error. If we make two copies of the file, one in which the keys are in normal alphabetic order and another in which they are ordered from right to left (as if the words were spelled backwards), a misspelled search argument will probably agree up to half or more of its length with an entry in one of these two files. The search methods of Sections 6.2 and 6.3 can therefore be adapted to find the key that was probably intended.

A related problem has received considerable attention in connection with airline reservation systems, and in other applications involving people's names when there is a good chance that the name will be misspelled due to poor handwriting or voice transmission. The goal is to transform the argument into some code that tends to bring together all variants of the same name. The following "Soundex" method, which was originally developed by Margaret K. Odell and Robert C. Russell [cf. *U.S. Patents 1261167* (1918), *1435663* (1922)], has often been used for encoding surnames:

1. Retain the first letter of the name, and drop all occurrences of a, e, h, i, o, u, w, y in other positions.
2. Assign the following numbers to the remaining letters after the first:

b, f, p, v $\rightarrow$ 1	l $\rightarrow$ 4
c, g, j, k, q, s, x, z $\rightarrow$ 2	m, n $\rightarrow$ 5
d, t $\rightarrow$ 3	r $\rightarrow$ 6

3. If two or more letters with the same code were adjacent in the original name (before step 1), omit all but the first.
4. Convert to the form "letter, digit, digit, digit" by adding trailing zeros (if there are less than three digits), or by dropping rightmost digits (if there are more than three).

For example, the names Euler, Gauss, Hilbert, Knuth, Lloyd, and Łukasiewicz have the respective codes E460, G200, H416, K530, L300, L222. Of course this system will bring together names that are somewhat different, as well as names that are similar; the same six codes would be obtained for Ellery, Ghosh, Heilbronn, Kant, Ladd, and Lissajous. And on the other hand a few related names like Rogers and Rodgers, or Sinclair and St. Clair, or Tchebysheff and Chebyshev, remain separate. But by and large the Soundex code greatly increases the chance of finding a name in one of its disguises. [For further information, cf. C. P. Bourne and D. F. Ford, *JACM* **8** (1961), 538–552; Leon Davidson, *CACM* **5** (1962), 169–171; *Federal Population Censuses* 1790–1890 (Washington, D.C.: National Archives, 1971), 90.]

When using a scheme like Soundex, we need not give up the assumption that all keys are distinct; we can make lists of all records with equivalent codes, treating each list as a unit.

Large data bases tend to make the retrieval process more complex, since people often want to consider many different fields of each record as potential keys, with the ability to locate items when only part of the key information is specified. For example, given a large file about stage performers, a producer might wish to find all unemployed actresses between 25 and 30 with dancing talent and a French accent; given a large file of baseball statistics, a sportswriter may wish to determine the total number of runs scored by the Cincinnati Redlegs in 1964, during the seventh inning of night games, against lefthanded pitchers. Given a large file of data about anything, people like to ask arbitrarily complicated questions. Indeed, we might consider an entire library as a data base, and a searcher may want to find everything that has been published about information retrieval. An introduction to the techniques for such *multi-attribute retrieval* problems appears below in Section 6.5.

Before entering into a detailed study of searching, it may be helpful to put things in historical perspective. During the pre-computer era, many books of logarithm tables, trigonometry tables, etc., were compiled, so that mathematical calculations could be replaced by searching. Eventually these tables were transferred to punched cards, and used for scientific problems in connection with collators, sorters, and duplicating punch machines. But when stored-program computers were introduced, it soon became apparent that it was now cheaper to recompute $\log x$ or $\cos x$ each time, instead of looking up the answer in a table.

Although the problem of sorting received considerable attention already in the earliest days of computers, comparatively little was done about algorithms for searching. With small internal memories, and with nothing but sequential media like tapes for storing large files, searching was either trivially easy or almost impossible.

But the development of larger and larger random-access memories during the 1950's eventually led to the recognition that searching was an interesting problem in its own right. After years of complaining about the limited amounts of space in the early machines, programmers were suddenly confronted with larger amounts of memory than they knew how to use efficiently.

The first surveys of the searching problem were published by A. I. Dumey, *Computers & Automation* **5**, 12 (December 1956), 6–9; W. W. Peterson, *IBM J. Research & Development* **1** (1957), 130–146; A. D. Booth, *Information and Control* **1** (1958), 159–164; A. S. Douglas, *Comp. J.* **2** (1959), 1–9. More extensive treatments were given later by Kenneth E. Iverson, *A Programming Language* (New York: Wiley, 1962), 133–158, and by Werner Buchholz, *IBM Systems J.* **2** (1963), 86–111.

During the early 1960's, a number of interesting new search procedures based on tree structures were introduced, as we shall see; and research about searching is still actively continuing at the present time.

6.1. SEQUENTIAL SEARCHING

"Begin at the beginning, and go on till you find the right key: then stop." This sequential procedure is the obvious way to search, and it makes a useful starting point for our discussion of searching because many of the more intricate algorithms are based on it. We shall see that sequential searching involves some very interesting ideas, in spite of its simplicity.

The algorithm might be formulated more precisely as follows:

Algorithm S (*Sequential search*). Given a table of records $R_1, R_2, \ldots, R_N$, whose respective keys are $K_1, K_2, \ldots, K_N$, this algorithm searches for a given argument K. We assume that $N \geq 1$.

S1. [Initialize.] Set $i \leftarrow 1$.

S2. [Compare.] If $K = K_i$, the algorithm terminates successfully.

S3. [Advance.] Increase i by 1.

S4. [End of file?] If $i \leq N$, go back to S2. Otherwise the algorithm terminates unsuccessfully. ▮

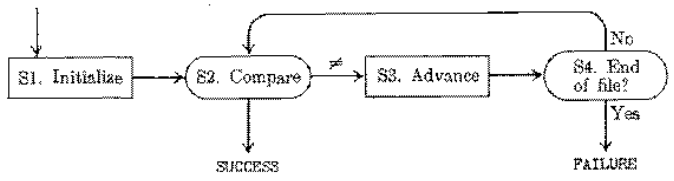


Fig. 1. Sequential search.

Note that this algorithm can terminate in two different ways, *successfully* (having located the desired key) or *unsuccessfully* (having established that the given argument is not present in the table). The same will be true of most other algorithms in this chapter.

A MIX program can be written down immediately:

Program S (*Sequential search*). Assume that K_i appears in location KEY + i, and that the remainder of record R_i appears in location INFO + i. The following program uses rA ≡ K, rI1 ≡ $i - N$.

```
START    LDA   K            1        S1. Initialize.
         ENT1  1-N          1        i ← 1.
2H       CMPA  KEY+N,1      C        S2. Compare.
         JE    SUCCESS      C        Exit if K = K_i.
         INC1  1            C − S    S3. Advance.
         J1NP  2B           C − S    S4. End of file?
FAILURE  EQU   *            1 − S    Exit if not in table.
```

At location SUCCESS, the instruction "LDA INFO+N,1" will now bring the desired information into rA. ▌

The analysis of this program is straightforward; it shows that the running time of Algorithm S depends on two things,

$$C = \text{the number of key comparisons;}\quad S = 1 \text{ if successful, } 0 \text{ if unsuccessful.} \tag{1}$$

Program S takes $5C - 2S + 3$ units of time. If the search successfully finds $K = K_i$, we have $C = i$, $S = 1$; hence the total time is $(5i + 1)u$. On the other hand if the search is unsuccessful, we have $C = N$, $S = 0$, for a total time of $(5N + 3)u$. If every input key occurs with equal probability, the average value of C in a successful search will be

$$\frac{1 + 2 + \cdots + N}{N} = \frac{N + 1}{2}; \tag{2}$$

the standard deviation is, of course, rather large, about $0.289N$ (see exercise 1).

The above algorithm is surely familiar to all programmers. But too few people know that it is *not* always the right way to do a sequential search! A straightforward change makes the algorithm faster, unless the list of records is quite short:

Algorithm Q (*Quick sequential search*). This algorithm is the same as Algorithm S, except that it assumes the presence of a "dummy" record R_{N+1} at the end of the file.

Q1. [Initialize.] Set $i \leftarrow 1$, and set $K_{N+1} \leftarrow K$.

Q2. [Compare.] If $K = K_i$, go to Q4.

Q3. [Advance.] Increase i by 1 and return to Q2.

Q4. [End of file?] If $i \le N$, the algorithm terminates successfully; otherwise it terminates unsuccessfully ($i = N + 1$). ▮

Program Q (*Quick sequential search*). $rA \equiv K$, $rI1 \equiv i - N$.

```
01   START    LDA   K            1           Q1. Initialize.
02            STA   KEY+N+1      1           K_{N+1} ← K.
03            ENT1  -N           1           i ← 0.
04            INC1  1            C + 1 − S   Q3. Advance.
05            CMPA  KEY+N,1      C + 1 − S   Q2. Compare.
06            JNE   *-2          C + 1 − S   To Q3 if K_i ≠ K.
07            J1NP  SUCCESS      1           Q4. End of file?
08   FAILURE  EQU   *            1 − S       Exit if not in table. ▮
```

In terms of the quantities C and S in the analysis of Program S, the running time has decreased to $(4C - 4S + 10)u$; this is an improvement whenever $C \ge 6$ in a successful search, and it is an improvement whenever $N \ge 8$ in an unsuccessful search.

The transition from Algorithm S to Algorithm Q makes use of an important "speed-up" principle: When an inner loop of a program tests two or more conditions, an attempt should be made to reduce it to just one condition.

Another technique will make Program Q *still* faster.

Program Q′ (*Quicker sequential search*). $rA \equiv K$, $rI1 \equiv i - N$.

```
01   START    LDA   K            1               Q1. Initialize.
02            STA   KEY+N+1      1               K_{N+1} ← K.
03            ENT1  -1-N         1               i ← −1.
04   3H       INC1  2            ⌊(C − S + 2)/2⌋  Q3. Advance. (twice)
05            CMPA  KEY+N,1      ⌊(C − S + 2)/2⌋  Q2. Compare.
06            JE    4F           ⌊(C − S + 2)/2⌋  To Q4 if K = K_i.
07            CMPA  KEY+N+1,1    ⌊(C − S + 1)/2⌋  Q2. Compare. (next)
08            JNE   3B           ⌊(C − S + 1)/2⌋  To Q3 if K ≠ K_{i+1}.
09            INC1  1            (C − S) mod 2   Advance i.
10   4H       J1NP  SUCCESS      1               Q4. End of file?
11   FAILURE  EQU   *            1 − S           Exit if not in table. ▮
```

The inner loop has been duplicated; this avoids about half of the "$i \leftarrow i + 1$" instructions, so it reduces the running time to

$$3.5C - 3.5S + 10 + \frac{(C - S) \bmod 2}{2}$$

GN 005634

units. We have saved 30 percent of the running time of Program S, when large tables are being searched; many existing programs can be improved in this way.

A slight variation of the algorithm is appropriate if we know that the keys are in increasing order:

Algorithm T (*Sequential search in ordered table*). 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 and speed, the algorithm assumes that there is a dummy record R_{N+1} whose key value is $K_{N+1} = \infty > K$.

T1. [Initialize.] Set $i \leftarrow 1$.

T2. [Compare.] If $K \leq K_i$, go to T4.

T3. [Advance.] Increase i by 1 and return to T2.

T4. [Equality?] If $K = K_i$, the algorithm terminates successfully. Otherwise it terminates unsuccessfully. ▌

If all input keys are equally likely, this algorithm takes essentially the same average time as Algorithm Q, for a successful search. But unsuccessful searches are performed about twice as fast, since the absence of a record can be established more quickly.

Each of the above algorithms uses subscripts to denote the table entries. It is convenient to describe the methods in terms of these subscripts, but the same search procedures can be used for tables which have a *linked* representation, since the data is being traversed sequentially. (See exercises 2, 3, and 4.)

Frequency of access. So far we have been assuming that every argument occurs as often as every other. This is not always a realistic assumption; in a general situation, key K_i will occur with probability p_i, where $p_1 + p_2 + \cdots + p_N = 1$. The time required to do a successful search is essentially proportional to the number of comparisons, C, which now has the average value

$$\overline{C}_N = p_1 + 2p_2 + \cdots + Np_N. \tag{3}$$

If we have the option of putting the records into the table in any desired order, this quantity $\overline{C}_N$ is smallest when

$$p_1 \geq p_2 \geq \cdots \geq p_N, \tag{4}$$

i.e., when the most frequently used records appear near the beginning.

Let's look at several probability distributions, in order to see how much of a saving is possible when the records are arranged in the "optimal" manner specified in (4). If $p_1 = p_2 = \cdots = p_N = 1/N$, formula (3) reduces to $\overline{C}_N = (N+1)/2$; we have already derived this in Eq. (2). Suppose, on the other hand, that

$$p_1 = \frac{1}{2}, \quad p_2 = \frac{1}{4}, \quad \ldots, \quad p_{N-1} = \frac{1}{2^{N-1}}, \quad p_N = \frac{1}{2^{N-1}}. \tag{5}$$

Then by exercise 7, $\overline{C}_N = 2 - 2^{1-N}$; the average number of comparisons is *less than two*, for this distribution, if the records appear in the proper order within the table.

Another probability distribution that suggests itself is

$$p_1 = Nc, \qquad p_2 = (N-1)c, \qquad \ldots, \qquad p_N = c,$$

where

$$c = 2/N(N+1). \tag{6}$$

This "wedge-shaped" distribution is not as dramatic a departure from uniformity as (5). In this case we find

$$\overline{C}_N = c \sum_{1 \le k \le N} k \cdot (N+1-k) = \frac{N+2}{3}; \tag{7}$$

the optimum arrangement saves about one-third of the search time which would have been obtained if the records had appeared in random order.

Of course the probability distributions in (5) and (6) are rather artificial, and they may never be a very good approximation to reality. A more typical distribution is "Zipf's law,"

$$p_1 = c/1, \quad p_2 = c/2, \quad \ldots, \quad p_N = c/N, \quad \text{where } c = 1/H_N. \tag{8}$$

This distribution was formulated by G. K. Zipf, who observed that the nth most common word in natural language text seems to occur with a frequency inversely proportional to n. [*Human Behavior and the Principle of Least Effort, an Introduction to Human Ecology* (Reading, Mass.: Addison-Wesley, 1949).] He observed the same phenomenon in census tables, when metropolitan areas are ranked in order of decreasing population. If Zipf's law governs the frequency of the keys in a table, we have immediately

$$\overline{C}_N = N/H_N; \tag{9}$$

searching such a file is about $\frac{1}{2} \ln N$ times as fast as searching the same file with randomly-ordered records. [Cf. A. D. Booth et al., *Mechanical Resolution of Linguistic Problems* (New York: Academic Press, 1958), 79.]

Another approximation to realistic distributions is the "80-20" rule of thumb that has been commonly observed in commercial applications [cf. W. P. Heising, *IBM Systems J.* 2 (1963), 114–115]. This rule states that 80 percent of the transactions deal with the most active 20 percent of a file; and the same applies to this 20 percent, so that 64 percent of the transactions deal with the most active 4 percent, etc. In other words,

$$\frac{p_1 + p_2 + \cdots + p_{.20n}}{p_1 + p_2 + p_3 + \cdots + p_n} \approx .80, \qquad \text{for all } n. \tag{10}$$

GN 005636

One distribution which satisfies this rule exactly whenever n is a multiple of 5 is

$$p_1 = c,\quad p_2 = (2^\theta - 1)c,\quad p_3 = (3^\theta - 2^\theta)c,\quad \ldots,\quad p_N = (N^\theta - (N-1)^\theta)c, \tag{11}$$

where

$$c = 1/N^\theta, \qquad \theta = \frac{\log .80}{\log .20} = 0.1386, \tag{12}$$

since $p_1 + p_2 + \cdots + p_n = cn^\theta$ for all n in this case. It is not especially easy to work with the probabilities in (11); we have, however, $n^\theta - (n-1)^\theta = \theta n^{\theta-1}(1 + O(1/n))$, so there is a simpler distribution which approximately fulfills the 80-20 rule, namely

$$p_1 = c/1^{1-\theta},\quad p_2 = c/2^{1-\theta},\quad \ldots,\quad p_N = c/N^{1-\theta},\quad \text{where } c = 1/H_N^{(1-\theta)}. \tag{13}$$

Here $\theta = \log .80/\log .20$ as before, and $H_N^{(s)}$ is the Nth harmonic number of order s, namely $1^{-s} + 2^{-s} + \cdots + N^{-s}$. Note that this probability distribution is very similar to that of Zipf's law (8); as θ varies from 1 to 0, the probabilities vary from a uniform distribution to a Zipfian one. (Indeed, Zipf found that $\theta \approx \frac{1}{2}$ in the distribution of personal income.). Applying (3) to (13) yields

$$\overline{C}_N = H_N^{(-\theta)}/H_N^{(1-\theta)} = \frac{\theta N}{\theta + 1} + O(N^{1-\theta}) \approx 0.122N \tag{14}$$

as the mean number of comparisons for the 80-20 law (see exercise 8).

A study of word frequencies carried out by E. S. Schwartz [see the interesting graph on p. 422 of *JACM* **10** (1963)] suggests that a more appropriate substitute for Zipf's law is

$$p_1 = c/1^{1+\theta},\ p_2 = c/2^{1+\theta},\ \ldots,\ p_N = c/N^{1+\theta},\ \text{where } c = 1/H_N^{(1+\theta)}, \tag{15}$$

for a small *positive* value of θ. [Cf. with (13); the sign of θ has been reversed.] In this case

$$\overline{C}_N = H_N^{(\theta)}/H_N^{(1+\theta)} = N^{1-\theta}/(1-\theta)\zeta(1+\theta) + O(N^{1-2\theta}), \tag{16}$$

which is substantially smaller than (9) as $N \to \infty$.

A "self-organizing" file. The above calculations with probabilities are very nice, but in most cases we don't know what the probabilities are. We could keep a count in each record of how often it has been accessed, reallocating the records on the basis of these counts; the formulas derived above suggest that this procedure would often lead to a worthwhile savings. But we probably don't want to devote so much memory space to the count fields, since we can make better use of that memory (e.g. by using nonsequential search techniques which are explained later in this chapter).

A simple scheme, which has been in use for many years although its origin is unknown, can be used to keep the records in a pretty good order without uxiliary count fields: Whenever a record has been successfully located, it is noved to the beginning of the table.

The idea behind this "self-organizing" technique is that the oft-used items vill tend to be located fairly near the beginning of the table, when we need hem. If we assume that the N keys occur with respective probabilities $p_1, p_2, \ldots, p_N\}$, with each search being completely *independent* of previous earches, it can be shown that the average number of comparisons needed to ind an item in such a self-organizing file tends to the limiting value

$$\overline{C}_N = 1 + 2 \sum_{1 \le i < j \le N} \frac{p_i p_j}{p_i + p_j} = \frac{1}{2} + \sum_{i,j} \frac{p_i p_j}{p_i + p_j}. \tag{17}$$

See exercise 11.) For example, if $p_i = 1/N$ for $1 \le i \le N$, the self-organizing able is always in completely random order, and this formula reduces to the amiliar expression $(N+1)/2$ derived above. In general, the average number 17) of comparisons is always less than twice the optimal value (3), since $_N \le 1 + 2\sum_{1 \le j \le N}(j-1)p_j < 2\sum_{1 \le j \le N} jp_j$.

Let us see how well the self-organizing procedure works when the key robabilities obey Zipf's law (8). We have

$$\begin{aligned}
\overline{C}_N &= \frac{1}{2} + \sum_{1 \le i,j \le N} \frac{(c/i)(c/j)}{c/i + c/j} = \frac{1}{2} + c \sum_{1 \le i,j \le N} \frac{1}{i+j} \\
&= \frac{1}{2} + c \sum_{1 \le i \le N} (H_{N+i} - H_i) = \frac{1}{2} + c \sum_{1 \le i \le 2N} H_i - 2c \sum_{1 \le i \le N} H_i \\
&= \tfrac{1}{2} + c\bigl((2N+1)H_{2N} - 2N - 2(N+1)H_N + 2N\bigr) \\
&= \tfrac{1}{2} + c\bigl(N \ln 4 - \ln N + O(1)\bigr) \approx 2N/\log_2 N,
\end{aligned} \tag{18}$$

by Eqs. 1.2.7–8, 3. This is substantially better than $\frac{1}{2}N$, when N is reasonably large, and it is only about $\ln 4 \approx 1.386$ times as many comparisons as would be obtained in the optimum arrangement (cf. 9).

Computational experiments involving actual compiler symbol tables indicate that the self-organizing method works even better than the above formulas predict, because successive searches are not independent (small groups of keys tend to occur in bunches).

This self-organizing scheme was first analyzed by John McCabe [*Operations Research* 13 (1965), 609–618], who established (17). See G. Schay, Jr., and F. W. Dauer, *SIAM J. Appl. Math.* 15 (1967), 874–888, for related results.

McCabe also introduced another interesting scheme, under which each successfully located key that is not already at the beginning of the table is simply *interchanged with the preceding key*, instead of being moved all the way to the front. He conjectured that the limiting average search time for this method, assuming independent searches, never exceeds (17). Ronald L. Rivest later proved in fact that the transposition method asymptotically uses strictly *fewer* comparisons than the move-to-first method, except of course when $N \le 2$ or when all the nonzero probabilities are equal. [To appear.]

GN 005638

Tape searching with unequal-length records. Now let's give the problem still another twist: Suppose the table we are searching is stored on tape, and the individual records have varying lengths. For example, in an old-fashioned operating system, the "system library tape" was such a file; standard system programs such as compilers, assemblers, loading routines, report generators, etc. were the "records" on this tape, and most user jobs would start by searching down the tape until the appropriate routine had been input. This setup makes our previous analysis of Algorithm S inapplicable, since step S3 takes a variable amount of time each time we reach it. The number of comparisons is therefore not the only criterion of interest.

Let L_i be the length of record R_i, and let p_i be the probability that this record will be sought. The running time of the search method will now be approximately proportional to

$$p_1L_1 + p_2(L_1 + L_2) + \cdots + p_N(L_1 + L_2 + L_3 + \cdots + L_N). \qquad (19)$$

When $L_1 = L_2 = \cdots = L_N = 1$, this reduces to (3), the case already studied.

It seems logical to put the most frequently-needed records at the beginning of the tape; but this is sometimes a bad idea! For example, assume that the tape contains just two programs, A and B; A is needed twice as often as B, but it is four times as long. Thus, $N = 2$, $p_A = \frac{2}{3}$, $L_A = 4$, $p_B = \frac{1}{3}$, $L_B = 1$. If we place A first on tape, according to the "logical" principle stated above, the average running time is $\frac{2}{3}\cdot 4 + \frac{1}{3}\cdot 5 = \frac{13}{3}$; but if we use an "illogical" idea, placing B first, the average running time is reduced to $\frac{1}{3}\cdot 1 + \frac{2}{3}\cdot 5 = \frac{11}{3}$.

The optimum arrangement of programs on a library tape may be determined as follows.

Theorem S. *Let L_i and p_i be as defined above. The arrangement of records in the table is optimal if and only if*

$$p_1/L_1 \ge p_2/L_2 \ge \cdots \ge p_N/L_N. \qquad (20)$$

In other words, the minimum value of

$$p_{a_1}L_{a_1} + p_{a_2}(L_{a_1} + L_{a_2}) + \cdots + p_{a_N}(L_{a_1} + \cdots + L_{a_N}),$$

over all permutations $a_1\, a_2 \ldots a_N$ of $\{1, 2, \ldots, N\}$, is equal to (19) if and only if (20) holds.

Proof. Suppose that R_i and R_{i+1} are interchanged on the tape; the cost (19) changes from

$$\cdots + p_i(L_1 + \cdots + L_{i-1} + L_i) + p_{i+1}(L_1 + \cdots + L_{i+1}) + \cdots$$

to

$$\cdots + p_{i+1}(L_1 + \cdots + L_{i-1} + L_{i+1}) + p_i(L_1 + \cdots + L_{i+1}) + \cdots,$$

a net change of $p_iL_{i+1} - p_{i+1}L_i$. Therefore if $p_i/L_i < p_{i+1}/L_{i+1}$, such an interchange will improve the running time, and the given arrangement is not optimal. It follows that (20) holds in any optimal arrangement.

Conversely, assume that (20) holds; we need to prove that the arrangement is optimal. The argument just given shows that the arrangement is "locally optimal" in the sense that adjacent interchanges make no improvement; but there may conceivably be a long, complicated sequence of interchanges which leads to a better "global optimum." We shall consider two proofs, one which uses computer science and one which uses a mathematical trick.

First proof. Assume that (20) holds. We know that any permutation of the records can be "sorted" into the order $R_1 R_2 \ldots R_N$ by using a sequence of interchanges of adjacent records. Each of these interchanges replaces $\ldots R_j R_i \ldots$ by $\ldots R_i R_j \ldots$ for some $i < j$, so it decreases the search time by the nonnegative amount $p_i L_j - p_j L_i$. Therefore the order $R_1 R_2 \ldots R_N$ must have minimum search time.

Second proof. Replace each probability p_i by

$$p_i(\epsilon) = p_i + \epsilon^i - (\epsilon^1 + \epsilon^2 + \cdots + \epsilon^N)/N, \tag{21}$$

where ϵ is an extremely small positive number. When ϵ is sufficiently small, we will never have $x_1 p_1(\epsilon) + \cdots + x_N p_N(\epsilon) = y_1 p_1(\epsilon) + \cdots + y_N p_N(\epsilon)$ unless $x_1 = y_1, \ldots, x_N = y_N$; and in particular, equality will not hold in (20). Consider now the $N!$ permutations of the records; at least one of these is optimum, and we know that it satisfies (20); but only one permutation satisfies (20) because there are no equalities. Therefore (20) uniquely characterizes the optimum arrangement of records in the table for the probabilities $p_i(\epsilon)$, whenever ϵ is sufficiently small. By continuity, the same arrangement must also be optimum when ϵ is set equal to zero. (This "tie-breaking" type of proof is often useful in connection with combinatorial optimization.) ▮

Theorem S is due to W. E. Smith, *Naval Research Logistics Quarterly* **3** (1956), 59–66. The exercises below contain further results about optimum file arrangements.

File compression. Sequential searching on tape and other external memory devices goes faster if we can pack the data so that it takes up less space; therefore it is a good idea to consider alternative ways to represent a file. We need not always store the keys explicitly.

For example, suppose that we want to have a table of all the prime numbers less than one million, for use in factoring 12-digit numbers. (Cf. Section 4.5.4.) There are 78498 such primes; so if we use 20 bits for each one, the file will be 1,569,960 bits long. This is obviously wasteful, since we could have a million-bit table, with each bit telling us whether or not the corresponding number is prime. Since all primes (except 2) are odd, the file could in fact be shortened to 500,000 bits.

Another way to cut down the size of this file is to list the sizes of *gaps* between primes, instead of the primes themselves. According to Table 1, the difference $(p_{k+1} - p_k)/2$ is less than 64 for all primes less than 1,357,201, so we can represent all primes between 3 and 1000000 by simply storing 78496 six-bit values of (gap size)/2; this makes the file about 471,000 bits long. Further compression can be achieved by using a variable-length binary code for the gaps (cf. Section 6.2.2).

Table 1

RECORD-BREAKING GAPS BETWEEN CONSECUTIVE PRIME NUMBERS

Gap ($p_{k+1} - p_k$)	p_k	Gap ($p_{k+1} - p_k$)	p_k
1	2	96	360653
2	3	112	370261
4	7	114	492113
6	23	118	1349533
8	89	132	1357201
14	113	148	2010733
18	523	154	4652353
20	887	180	17051707
22	1129	210	20831323
34	1327	220	47326693
36	9551	222	122164747
44	15683	234	189695659
52	19609	248	191912783
72	31397	250	387096133
86	155921	282	436273009

This table lists $p_{k+1} - p_k$ whenever it exceeds $p_{j+1} - p_j$ for all $j < k$. For further information, see R. P. Brent, *Math. Comp.* **27** (1973), 959–963.

EXERCISES

1. [*M20*] When all the search keys are equally probable, what is the standard deviation of the number of comparisons made in a successful sequential search through a table of N records?

2. [*16*] Restate the steps of Algorithm S, using linked-memory notation instead of subscript notation. (If `P` points to a record in the table, assume that `KEY(P)` is the key, `INFO(P)` is the associated information, and that `LINK(P)` is a pointer to the next record. Assume also that `FIRST` points to the first record, and that the last record points to Λ.)

3. [*16*] Write a `MIX` program for the algorithm of exercise 2. What is the running time of your program, in terms of the quantities C and S in (1)?

▶ **4.** [*17*] Does the idea of Algorithm Q carry over from subscript notation to linked-memory notation? (Cf. exercise 2.)

5. [*20*] Program Q′ is, of course, noticeably faster than Program Q, when C is large. But are there any small values of C and S for which Program Q′ actually takes more time than Program Q?

▶ **6.** [*20*] Add three more instructions to Program Q′, reducing its running time to about $(3.33C + \text{constant})u$.

7. [*M20*] Evaluate the average number of comparisons, (3), using the "binary" probability distribution (5).

8. [*HM22*] Find an asymptotic series for $H_n^{(x)}$ as $n \to \infty$, when $x \neq 1$.

▶ **9.** [*M23*] The text observes that the probability distributions given by (11) and (13) are roughly equivalent, and that the mean number of comparisons using (13) is $\theta N/(\theta + 1) + O(N^{1-\theta})$. Is the mean number of comparisons equal to $\theta N/(\theta + 1) + O(N^{1-\theta})$ also when the probabilities of (11) are used?

10. [*M20*] The best arrangement of records in a sequential table is specified by (4); what is the *worst* arrangement? Show that the average number of comparisons in the worst arrangement has a simple relation to the average number of comparisons in the best arrangement.

11. [*M30*] The purpose of this exercise is to analyze the behavior of the text's self-organizing file. First we need to define some rather complicated notation: Let $f_m(x_1, x_2, \ldots, x_m)$ be the infinite sum of all distinct ordered products $x_{i_1}x_{i_2}\ldots x_{i_k}$ such that $1 \le i_1, \ldots, i_k \le m$, where each of $x_1, x_2, \ldots, x_m$ appears in every term. For example,

$$f_2(x, y) = \sum_{j,k\ge 0} \left(x^{1+j}y(x+y)^k + y^{1+j}x(x+y)^k\right) = \frac{xy}{1-x-y}\left(\frac{1}{1-x} + \frac{1}{1-y}\right).$$

Given a set X of n variables $\{x_1, \ldots, x_n\}$, let

$$P_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} f_m(x_{j_1}, \ldots, x_{j_m});\quad Q_{nm} = \sum_{1\le j_1<\cdots<j_m\le n} \frac{1}{1 - x_{j_1} - \cdots - x_{j_m}}.$$

For example, $P_{32} = f_2(x_1, x_2) + f_2(x_1, x_3) + f_2(x_2, x_3)$ and $Q_{32} = 1/(1 - x_1 - x_2) + 1/(1 - x_1 - x_3) + 1/(1 - x_2 - x_3)$. By convention we set $P_{n0} = Q_{n0} = 1$.

a) Assume that the text's self-organizing file has been servicing requests for item R_i with probability p_i. After the system has been running a long time, show that R_i will be the mth item from the front with limiting probability $p_iP_{N-1,m-1}$, where $X = \{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_N\}$.

b) By summing the result of (a) for $m = 1, 2, \ldots$, we obtain the identity

$$P_{nn} + P_{n,n-1} + \cdots + P_{n0} = Q_{nn}.$$

Prove that, consequently,

$$P_{nm} + \binom{n-m+1}{1}P_{n,m-1} + \cdots + \binom{n-m+m}{m}P_{n0} = Q_{nm};$$

$$Q_{nm} - \binom{n-m+1}{1}Q_{n,m-1} + \cdots + (-1)^m\binom{n-m+m}{m}Q_{n0} = P_{nm}.$$

c) Compute the limiting average distance $d_i = \sum_{m\ge 1} mp_iP_{N-1,m-1}$ of R_i from the front of the list; then evaluate $\overline{C}_N = \sum_{1\le i\le N} p_i\,d_i$.

12. [*M23*] Use (17) to evaluate the average number of comparisons needed to search the self-organizing file when the search keys have the binary probability distribution (5).

13. [*M27*] Use (17) to evaluate $\overline{C}_N$ for the probability distribution (6).

14. [*M21*] Given two sequences $\langle x_1, x_2, \ldots, x_n\rangle$ and $\langle y_1, y_2, \ldots, y_n\rangle$ of real numbers, what permutation $a_1\,a_2\ldots a_n$ of the subscripts will make $\sum x_iy_{a_i}$ a maximum? a minimum?

▶ **15.** [*M22*] The text shows how to arrange programs optimally on a "system library tape," when only one program is being sought. But another set of assumptions is more appropriate for a *subroutine* library tape, from which we may wish to load various subroutines called for in a user's program.

For this case let us suppose that subroutine i is desired with probability P_i, independently of whether or not other subroutines are desired. Then, for example, the probability that no subroutines at all are needed is $(1 - P_1)(1 - P_2) \ldots (1 - P_N)$; and the probability that the search will end just after loading the ith subroutine is $P_i(1 - P_{i+1}) \ldots (1 - P_N)$. If L_i is the length of subroutine i, the average search time will therefore be essentially proportional to

$$L_1P_1(1 - P_2) \ldots (1 - P_N) + (L_1 + L_2)P_2(1 - P_3) \ldots (1 - P_N) + \cdots + (L_1 + \cdots + L_N)P_N.$$

What is the optimum arrangement of subroutines on the tape, under these assumptions?

16. [*M22*] (H. Riesel.) A programmer wants to test whether or not n given conditions are all simultaneously true. (For example, he may want to test whether both $x > 0$ and $y < z^2$, and it is not immediately clear which condition should be tested first.) Suppose that it costs T_i units of time to test condition i, and that the condition will be true with probability p_i, independent of the outcomes of all the other conditions. In which order should he make the tests?

17. [*M23*] (W. E. Smith.) Suppose you have to do n jobs; the ith job takes T_i units of time, and it has a *deadline* D_i. In other words, the ith job is supposed to be finished after at most D_i units of time have elapsed. What schedule $a_1 a_2 \ldots a_n$ for processing the jobs will minimize the *maximum tardiness*, i.e.,

$$\max(T_{a_1} - D_{a_1}, T_{a_1} + T_{a_2} - D_{a_2}, \ldots, T_{a_1} + T_{a_2} + \cdots + T_{a_n} - D_{a_n})?$$

18. [*M30*] (*Catenated search.*) Suppose N records are located in a linear array $R_1 \ldots R_N$, with probability p_i that record R_i will be sought. A search process is called "catenated" if each search begins where the last one left off. If consecutive searches are independent, the average time required will be $\sum_{1 \le i,j \le N} p_ip_jd(i, j)$, where $d(i, j)$ represents the amount of time to do a search that starts at position i and ends at position j. This model can be applied, for example, to disk file seek time, if $d(i, j)$ is the time needed to travel from cylinder i to cylinder j.

The object of this exercise is to characterize the optimum placement of records for catenated searches, whenever $d(i, j)$ is an increasing function of $|i - j|$, i.e., whenever we have $d(i, j) = d_{|i-j|}$ for $d_1 < d_2 < \cdots < d_{N-1}$. (The value of d_0 is irrelevant.) Prove that in this case the records are optimally placed, among all $N!$ permutations, if and only if either $p_1 \le p_N \le p_2 \le p_{N-1} \le \cdots \le p_{\lfloor N/2 \rfloor+1}$ or $p_N \le p_1 \le p_{N-1} \le p_2 \le \cdots \le p_{\lceil N/2 \rceil}$. (Thus, an "organ pipe" arrangement of probabilities is best, as shown in Fig. 2.) *Hint:* Consider any arrangement where the respective probabilities are $q_1\, q_2 \ldots q_k\, s\, r_k \ldots r_2\, r_1\, t_1 \ldots t_m$, for some $m \ge 0$ and $k > 0$; $N = 2k + m + 1$. Show that the rearrangement $q_1'\, q_2' \ldots q_k'\, s\, r_k' \ldots r_2'\, r_1'\, t_1 \ldots t_m$ is better, where $q_i' = \min(q_i, r_i)$ and $r_i' = \max(q_i, r_i)$, except when $q_i' = q_i$ and $r_i' = r_i$ for all i or when $q_i' = r_i$ and $r_i' = q_i$ and $t_j = 0$ for all i, j. The same holds when s is not present and $N = 2k + m$.

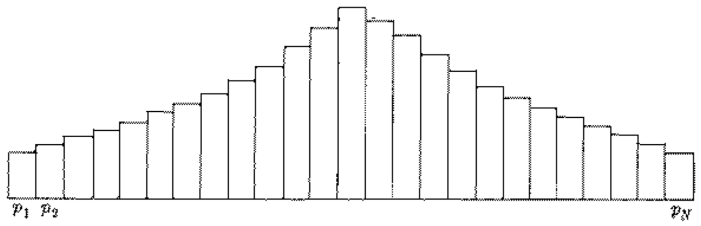


Fig. 2. An "organ-pipe arrangement" of probabilities minimizes the average seek time in a catenated search.

19. [*M20*] Continuing exercise 18, what are the optimal arrangements for catenated searches when the function $d(i, j)$ has the property that $d(i, j) + d(j, i) = c$ for all i, j? [This situation occurs, for example, on tapes without read-backwards capability, when we do not know the appropriate direction to search; for $i < j$ we have, say, $d(i, j) = a + b(L_{i+1} + \cdots + L_j)$ and $d(j, i) = a + b(L_{j+1} + \cdots + L_N) + r + b(L_1 + \cdots + L_i)$, where r is the rewind time.]

20. [*M28*] Continuing exercise 18, what are the optimal arrangements for catenated searches when the function $d(i, j) = \min(d_{|i-j|}, d_{n-|i-j|})$, for $d_1 < d_2 < \cdots$? [This situation occurs, for example, in a two-way linked circular list, or in a two-way shift-register storage device.]

21. [*M28*] Consider an n-dimensional cube whose vertices have coordinates $(d_n, \ldots, d_1)$ with $d_i = 0$ or 1; two vertices are called *adjacent* if they differ in exactly one coordinate. Suppose that a set of 2^n numbers $x_0 \le x_1 \le \cdots \le x_{2^n-1}$ is to be assigned to the 2^n vertices in such a way that $\sum |x_i - x_j|$ is minimized, where the sum is over all i and j such that x_i and x_j have been assigned to adjacent vertices. Prove that this minimum will be achieved if x_i is assigned to the vertex whose coordinates are the binary representation of i, for all i.

▸**22.** [*20*] Suppose you want to search a large file, not for equality but to find the 1000 records which are *closest* to a given key, in the sense that these 1000 records have the smallest values of $d(K_i, K)$ for some given distance function d. What data structure is most appropriate for such a sequential search?

GN 005644

6.2. SEARCHING BY COMPARISON OF KEYS

In this section we shall discuss search methods which are based on a linear ordering of the keys (e.g., 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 ($K = K_i$ vs. $K \neq K_i$), but if we free ourselves from the restriction of sequential access it becomes possible 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 man whose number is 795-6841? There is no better way to tackle this problem than to use the sequential methods of Section 6.1. (However, a clever private detective might try dialing the number and finding out who answers; or he might have a friend at the telephone company who has access to 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 only need to search the table once, it is faster to do a sequential search 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 which are appropriate for searching a table whose keys are in order,

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

making random accesses to the table entries. After comparing K to K_i in such a table, we either have

- $K < K_i$ $[R_i, R_{i+1}, \ldots, R_N$ 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 which suggests itself is to start by comparing K to the middle key in the table; the result of this probe tells

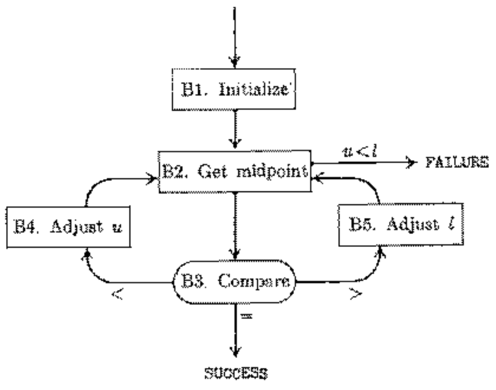


Fig. 3. Binary search.

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 $\log_2 N$ comparisons, we will have found the key or we will have established 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 somewhat 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, which 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,

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.

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.

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⌋.
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 $S = 1$ or 0 depending on whether the outcome is successful or not. Note that 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 $(26C - 18S + 20)u$.

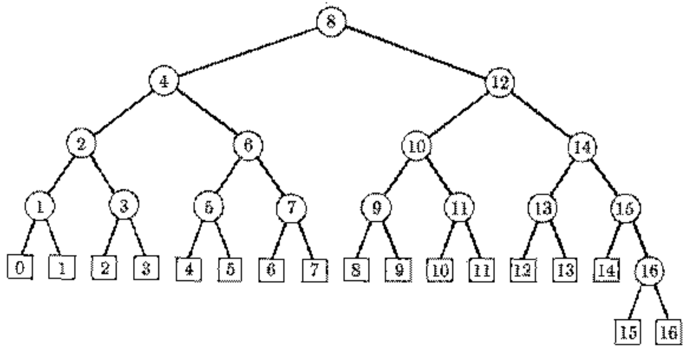


Fig. 5. A binary tree which corresponds to binary search when $N = 16$.

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

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

$$\bigcirc\!\!\!\!\!\!\!\!\!\;\;\lceil N/2 \rceil\,,$$

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 a binary tree in which the nodes are labelled 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

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

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, 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 keys is equally likely. Then we have

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

by the definition of internal and external path length. We have seen 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 Hibbard [*JACM* **9** (1962), 16–17], holds for all search methods which correspond to binary trees, i.e., for all methods which are based on nonredundant comparisons. The variance of C_N can also be expressed in terms of the variance of C'_N (see exercise 25).

From the above formulas 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 all its external nodes occur on at most two adjacent levels. It follows that the external path length of the tree corresponding to Algorithm B is

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

(See Eq. 5.3.1–33.) 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 (cf. Eq. 5.3.1–34)

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

where $0 \le \epsilon, \epsilon' < 0.0861$.

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

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

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, 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 which should be set to $-\infty$ (or any value less than K). We assume that $N \ge 1$.

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 which 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 which 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. ▌

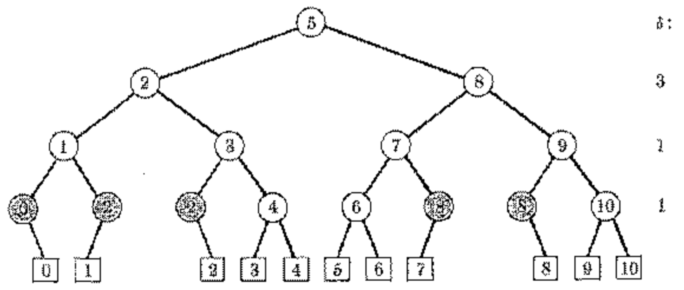


Fig. 6. The binary tree for a "uniform" binary search, when $N = 10$.

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; these 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 $\ell - 1$ 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 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

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

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

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

C4. [Increase i.] If DELTA[j] = 0, the algorithm terminates unsuccessfully. Otherwise, set $i \leftarrow i + \text{DELTA}[j]$, $j \leftarrow j + 1$, 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 rA $\equiv K$, rI1 $\equiv i$, rI2 $\equiv j$, rI3 $\equiv$ DELTA[j].

```
START    ENT1  N+1/2         1          C1. Initialize.
         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.
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 9 shows that we have $K < K_i$ roughly as often as $K > K_i$; hence Program C takes approximately

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

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}, $2^{k-2}, \ldots, 1, 0$. On the other hand, if $K > K_i$ and $N > 2^k$, we reset i to $i' = N + 1 - 2^l$, where $l = \lceil \lg (N - 2^k) \rceil + 1$, and pretend that the first comparison was actually $K > K_{i'}$, using a uniform search with the δ's equal to $2^{l-1}, 2^{l-2}, \ldots, 1, 0$.

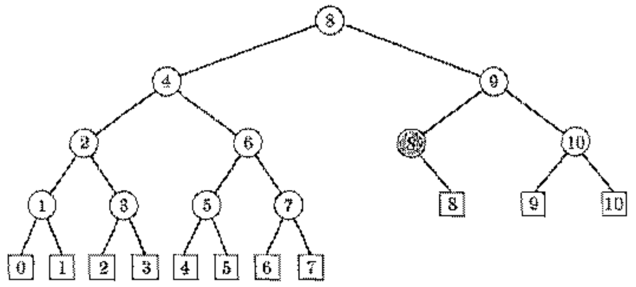


Fig. 7. The binary tree for Shar's almost uniform search, when $N = 10$.

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 is within one of the best possible average number of comparisons, in spite of the fact that it occasionally goes through several redundant steps in succession (cf. exercise 12).

Still another modification of binary search, which increases the speed of *all* the above methods when N is very 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 "Fibonacci search" procedure used to locate the maximum of a unimodal function [cf. *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. Note that 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 $\textcircled{F_k}$; 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.

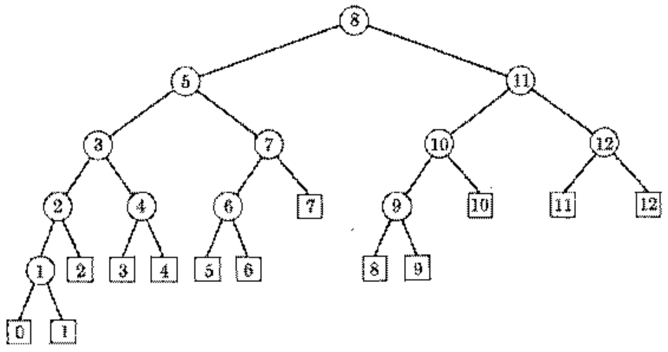


Fig. 8. The Fibonacci tree of order 6.

Note that, except for the external nodes, the numbers on the two sons of each internal node differ from their father's number by the same amount, and this amount is a Fibonacci number. Thus, $5 = 8 - F_4$ 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, this algorithm assumes 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.

GN 005654

Program F (*Fibonaccian search*). $rA \equiv K$, $rI1 \equiv i$, $(rI2, rI3) \equiv p-1$, $(rI3, rI2) \equiv q-1$.

```
      START  LDA   K                                          1          F1. Initialize.
             ENT1  F_k                                        1          i ← F_k.
             ENT2  F_{k-1}-1                                  1          p ← F_{k-1}.
             ENT3  F_{k-2}-1                                  1          q ← F_{k-2}.
             JMP   F2A                                        1          To step F2.
F4A  INC1  1,3           F4B  INC1  1,2          C2 − S − A      F4. Increase i. i ← i + q.
     DEC2  1,3                DEC3  1,2          C2 − S − A      p ← p − q.
     DEC3  1,2                DEC2  1,3          C2 − S − A      q ← q − p.
F2A  CMPA  KEY,1         F2B  CMPA  KEY,1        C               F2. Compare.
     JL    F3A                JL    F3B          C               To F3 if K < K_i.
     JE    SUCCESS            JE    SUCCESS      C2              Exit if K = K_i.
     J2NZ  F4A                J3NZ  F4B          C2 − S          To F4 if p ≠ 1.
     JMP   FAILURE            JMP   FAILURE      A               Exit if not in table.
F3A  DEC1  1,3           F3B  DEC1  1,2          C1              F3. Decrease i. i ← i − q.
     DEC2  1,3                DEC3  1,2          C1              p ← p − q.
     J3NN  F2B                J2NN  F2A          C1              Swap registers if q > 0.
     JMP   FAILURE            JMP   FAILURE      1 − S − A       Exit if not in table. ▌
```

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

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

Thus the left branch is taken about $\phi = 1.618$ times as often as the right branch (a fact which 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

$$\begin{aligned} &(6\phi k/\sqrt{5} - (2 + 22\phi)/5)\ u \approx (6.252 \log_2 N - 4.6)u \quad \text{for a successful search;} \\ &(6\phi k/\sqrt{5} + (58/(27\phi))/5)\ u \approx (6.252 \log_2 N + 5.8)u \quad \text{for an unsuccessful search.} \end{aligned} \tag{9}$$

This is slightly faster than Program C, although the worst case running time (roughly $8.6 \log_2 N$) 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" which show the starting 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 above algorithms, which make no distinction between "much greater" and "slightly greater."

These considerations suggest an algorithm which 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.

Unfortunately, computer simulation experiments show that interpolation search does not decrease the number of comparisons enough to compensate for the extra computing time involved, when searching a table stored within a high-speed memory. It has been successful only to a limited extent when applied to *external* searching in peripheral memory devices. (Note that dictionary look-up 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 is apparently the first of a series of three, which contained over 800 multiple-precision sexagesimal numbers and their reciprocals, sorted into lexicographic order. For example, the list included the following sequence of entries:

02 43 50 24	21 58 21 33 45
02 44 01 30	21 56 52 20 44 26 40
02 45 42 03 14 08	21 43 33 12 53 32 50 30 28 07 30
02 45 53 16 48	21 42 05
02 46 04 31 07 30	21 40 36 53 04 38 11 21 28 53 20

The task of sorting 800 entries like this, given the technology available at that time, must have been truly phenomenal. [See D. E. Knuth, *CACM* **15** (1972), 671–677, 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 which 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.

But 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 134–135 A.D. contain fragments of ledgers which 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 A.D.), but these were very rare. Thus, words were arranged by their first letter only, in the *Etymologiarum* of St. Isidorus (c. 630 A.D., 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 pp. He found some interesting old manuscripts that were evidently used as worksheets while sorting words by their first letters (see pp. 87–90 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.

It is interesting to note that Cawdrey was teaching *himself* how to alphabetize

GN 005657

as he prepared his dictionary; numerous misplaced words appear on the first few pages, but the last part is in nearly perfect alphabetical order!

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*, ed. by G. W. Patterson, **1** (1946), 9.7–9.8; **3** (1946), 22.8–22.9]. The method became "well known," 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** (February, 1958), 1–3.]

H. Bottenbruch [*JACM* **9** (1962), 214] was apparently the first to publish a binary search algorithm which works for all N. He presented an interesting variation of Algorithm B which avoids a separate test for equality until the very end: Using $i \leftarrow \lceil (l+u)/2 \rceil$ instead of $\lfloor (l+u)/2 \rfloor$ 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 the change is desirable only for large N (see exercise 23).

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], but his flowchart and analysis were incorrect. The Fibonacci tree (without labels) had appeared many years earlier, as a curiosity in the first edition of Hugo Steinhaus's popular book *Mathematical Snapshots* (New York: Stechert, 1938), p. 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 occur at the same level.

Interpolation searching was suggested by W. W. Peterson [*IBM J. Res. & Devel.* **1** (1957), 131–132]; he gave a theoretical estimate for the average number of comparisons needed if the keys are randomly selected from a uniform distribution, but the estimate does not seem to agree with actual simulation experiments.

EXERCISES

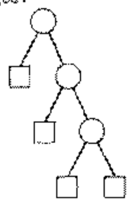
▸ 1. [*21*] Prove that if $u < l$ in step B2 of the binary search, we have $u = l - 1$ and

GN 005658

$K_u < K < K_l$. (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*] (a) What is the sum $\sum_{0 \le j \le \lfloor \lg N \rfloor + 2} \mathtt{DELTA}[j]$ of the increments in Algorithm C? (b) What are the minimum and maximum values of i which can occur in step C2?

9. [*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.

10. [*20*] Is there any value of $N > 1$ for which Algorithms B and C are exactly equivalent, in the sense that they will both perform the same sequence of comparisons for all search arguments?

11 [*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.

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

GN 005659

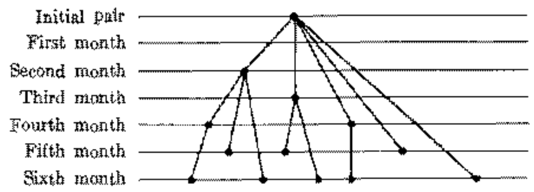


Fig. 9. Pairs of rabbits breeding by Fibonacci's rule.

16. [*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?

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 which, 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

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

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) A man argues as follows: "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 his argument?

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

22. [*M47*] Derive formulas which properly estimate the average number of iterations needed in the interpolation search applied to random data.

▸**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 $K_l \leq K < K_{u+1}$, 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 usually small; we would need $N > 2^{36}$ in order to compensate for the extra iteration necessary!)

Show that *every* search algorithm corresponding to a binary tree can be adapted to a search algorithm that uses two-way branching ($<$ vs. $\geq$) 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*] The complete binary tree is a convenient way to represent a minimum-path-length tree in consecutive locations. (Cf. Section 2.3.4.5.) 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 there is a simple algebraic relationship which connects 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 the Fibonacci tree is related to polyphase merge sorting on three tapes.

27. [*M30*] (H. S. Stone and John Linn.) Consider a search process which uses k processors simultaneously, and which 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 $K > K_{i_j}$, $1 \leq j \leq n$.

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.)

6.2.2. Binary Tree Searching

In the preceding section, we learned that an implicit binary tree structure makes it easier to understand the behavior of binary search and Fibonacci search. 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 dynamically changing, 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 which 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 ALGOL program, and the rest of that 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

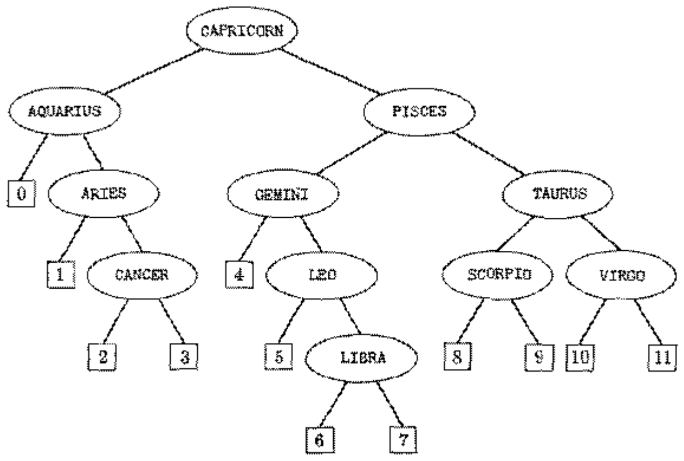


Fig. 10. A binary search tree.

of the zodiac. If we now search for the twelfth name, SAGITTARIUS, starting at the 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* (cf. 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 which 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:

KEY(P) = key stored in NODE(P);
LLINK(P) = pointer to left subtree of NODE(P);
RLINK(P) = pointer to right subtree of NODE(P).

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 (i.e., ROOT $\neq \Lambda$).

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

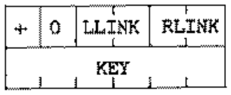
T2. [Compare.] If $K <$ KEY(P), go to T3; if $K >$ KEY(P), go to T4; and if $K =$ KEY(P), the search terminates successfully.

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` ⇐ `AVAIL`, the address of a new node. Set `KEY(Q)` ← K, `LLINK(Q)` ← `RLINK(Q)` ← Λ. (In practice, other fields of the new node should also be initialized.) If K was less than `KEY(P)`, set `LLINK(P)` ← `Q`, otherwise set `RLINK(P)` ← `Q`. (At this point we could set `P` ← `Q` and terminate the algorithm successfully.) ▮

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



(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 ≡ K, rI1 ≡ `P`, rI2 ≡ `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 = number of comparisons made;

$C1$ = number of times $K \le$ `KEY(P)`;

S = 1 if the search is successful, 0 otherwise.

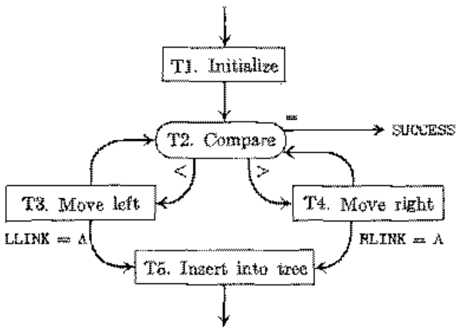


Fig. 11. Tree search and insertion.

On the average we have $C1 = \frac{1}{2}(C + S)$, since $C1 + C2 = C$ and $C1 - S \approx C2$; so the running time is about $(7.5C - 2.5S + 4)u$. This compares favorably with the binary search algorithms which use an implicit tree (cf. Program 6.2.1C). By duplicating the code as in Program 6.2.1F we could 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 be conveniently 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 which 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 which 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 well-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 can be proved 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

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

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

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

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

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

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

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

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

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

Applying (3) and simplifying yields the desired result

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

Exercises 6–8 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. Note that 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. (Note, however, that equal keys will not necessarily be adjacent in the tree, they will only be adjacent in symmetric order.) 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 L isn't bad; but there are better ways to sort. On the other hand, if we have an application that combines searching and 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 partition exchange ("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. (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. It is easy to delete a "leaf" node (one in which both subtrees are empty), or to delete a node in which either LLINK or RLINK $= \Lambda$; but when both LLINK and RLINK are non-null pointers, we have to do something special, since we can't point two ways at once.

For example, consider Fig. 10 again; how can we delete CAPRICORN? One solution is to delete the *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 one general way to do this.

GN 005667

Algorithm D (*Tree deletion*). Let Q be a variable which 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 ≡ ROOT or Q ≡ 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 ← Q. If RLINK(T) = Λ, set Q ← LLINK(T) and go
    to D4.
D2. [Find successor.] Set R ← RLINK(T). If LLINK(R) = Λ, set LLINK(R)
    ← LLINK(T), Q ← R, and go to D4.
D3. [Find null LLINK.] Set S ← LLINK(R). Then if LLINK(S) ≠ Λ, set R ← S
    and repeat this step until LLINK(S) = Λ. (At this point S will be equal to
    Q$, the symmetric successor of Q.) Finally, set LLINK(S) ← LLINK(T),
    LLINK(R) ← RLINK(S), RLINK(S) ← RLINK(T), Q ← S.
D4. [Free the node.] Set AVAIL ⇐ T (i.e., return 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, of course, very vague. We can summarize the situation more precisely as follows: Let $\mathfrak{T}$ be a tree of n elements, and let $P(\mathfrak{T})$ be the probability that $\mathfrak{T}$ occurs if its keys are inserted in random order by Algorithm T. Some trees are more probable than others. Let $Q(\mathfrak{T})$ be the probability that $\mathfrak{T}$ 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 $P(\mathfrak{T})$, we assume that the $n!$ permutations of the keys are equally likely; in calculating $Q(\mathfrak{T})$, we assume that the $(n + 1) \cdot (n + 1)!$ permutations of keys and selections of the key to delete are equally likely. The theorem states that $P(\mathfrak{T}) = Q(\mathfrak{T})$ for all $\mathfrak{T}$.

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."

GN 005668

Let $a_1a_2\ldots a_{n+1}$ 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 $b_1b_2\ldots b_n$ 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 $b_1b_2\ldots b_n$.

Fortunately 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 which is to be deleted, we have

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

Since there are $(n+1)\cdot(n+1)!$ 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 $b_1b_2\ldots b_n$ 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 which 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 clearly constructed $n + 1$ different deletions; hence $(n + 1)^2$ different deletions have been constructed for each permutation $b_1b_2 \ldots b_n$. Since only $(n + 1)^2n!$ 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 (external path length $-$ 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*

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

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 (cf. exercise 15). Empirical evidence suggests strongly that the path length tends to *decrease* after repeated deletions and insertions, so the departure from randomness seems to be in the right direction; 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,

D1½. [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.

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, shows that the average number of comparisons in a successful search will be

$$1 + \sum_{1\le k\le N} p_k(2H_k - 2) = 2\sum_{1\le k\le N} p_kH_k - 1. \tag{11}$$

(Cf. Eq. (5).)

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

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

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 less comparisons than we would make using binary search.

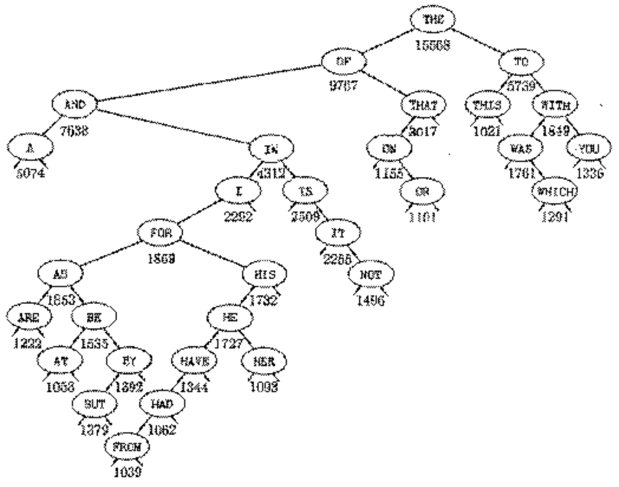


Fig. 12. The 31 most common English words, inserted in decreasing order of frequency.

For example, Fig. 12 shows the tree which results when the most common 31 words of English are entered in decreasing order of frequency. The relative frequency is shown with each word [cf. *Cryptanalysis* by H. F. Gaines (New York: Dover, 1956), p. 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 C, would require 4.393 comparisons.

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.

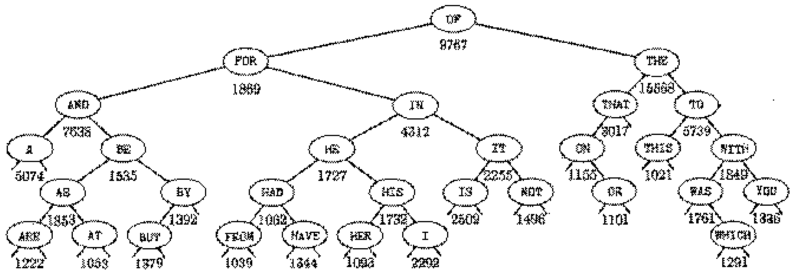
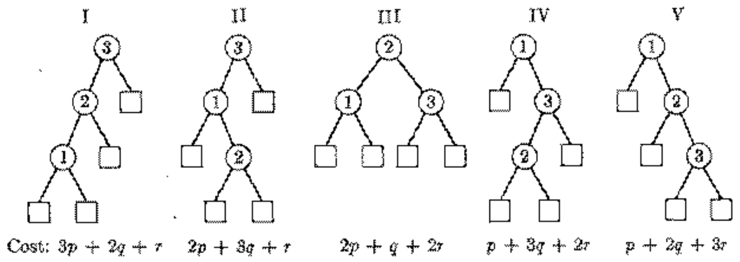


Fig. 13. Optimum search tree for the 31 most common English words.

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).

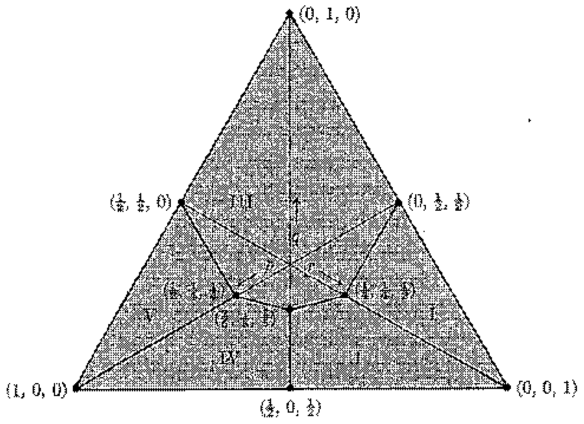


Fig. 14. If the relative frequencies of (K_1, K_2, K_3) are (p, q, r), this graph shows which of the five trees in (13) is best. The fact that $p + q + r = 1$ makes the graph two-dimensional although there are three coordinates.

Unfortunately, when N is large there are

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

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.

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

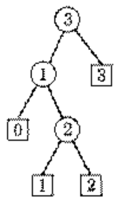
p_i = probability that K_i is the search argument;

q_i = probability that the search argument lies between K_i 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 which minimizes the expected number of comparisons in the search, namely

$$\sum_{1\le j\le n} p_j(\text{level}(\textcircled{j}) + 1) + \sum_{0\le k\le n} q_k\,\text{level}(\boxed{k}), \tag{14}$$

where $\textcircled{j}$ 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.

The principle which 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 which 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 Chapter 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; $c(i,j)$ and $w(i,j)$ are defined for $0 \le i \le j \le n$. It follows that

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

since the minimum possible cost of a tree with root $\textcircled{k}$ 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 $j - i = 1, 2, 3, \ldots, n$; 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

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

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 $t(i, j)$ which 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; else 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 less 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

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

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

GN 005675

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 ACM *Journal* were sorted to prepare a concordance in which there is 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. Note 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.)

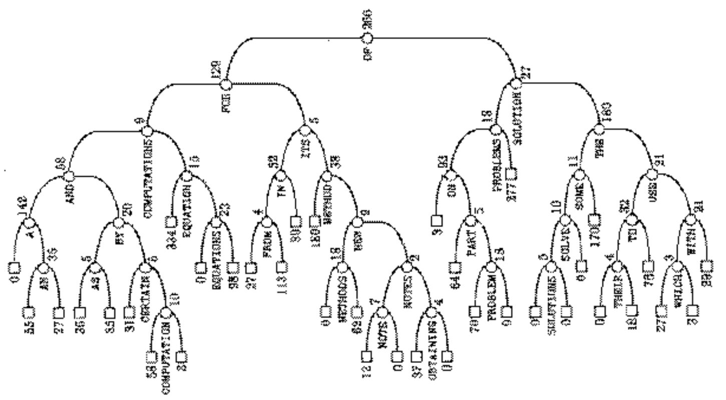


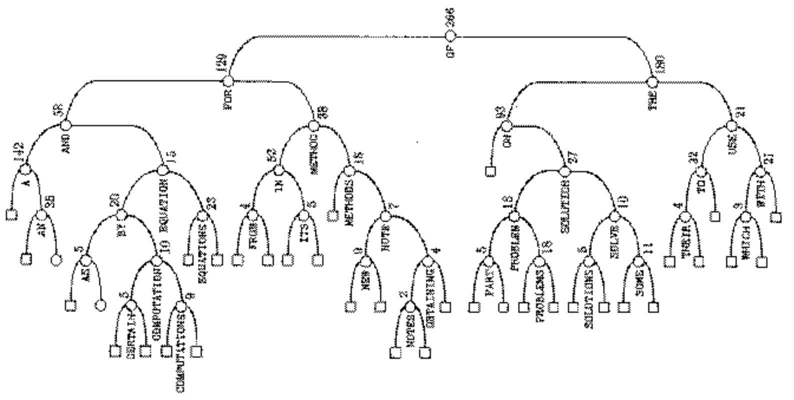
Fig. 15. An optimum binary search tree for a KWIC indexing application.

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 which 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, ..., 11, 21, 21, 21, 21, 21, 21); the values of $r[i, 35]$ for $i = 0, 1, \ldots, 34$ are (21, 21, ..., 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

GN 005676



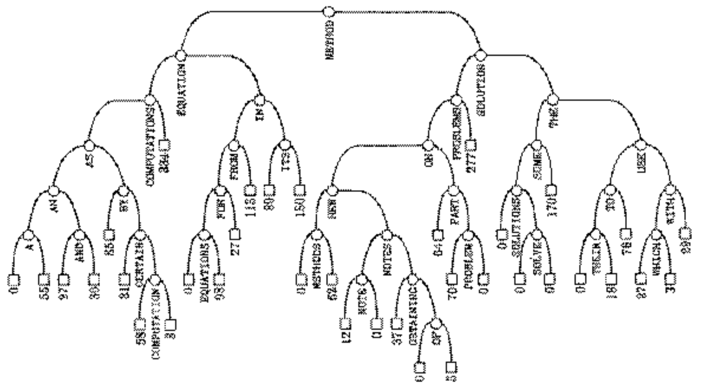


Fig. 16. Optimum binary search trees based on half of the data of Fig. 15: (a) external frequencies suppressed, (b) internal frequencies suppressed.

the full set of frequencies, the tree of Fig. 15 requires only 4.75 comparisons, on the average, while the trees of Fig. 16 require, respectively, 5.29 and 5.32. (A straight binary search would have been better than the trees of Fig. 16, in this example.)

Since Algorithm K requires time and space proportional to n^2, it becomes impractical to use it 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 weights $w(0, k-1)$ and $w(k, n)$ of the resulting subtrees are as near to being equal as possible. This approach also fails, because it may choose a node with very small p_k to be the root.

A fairly 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.

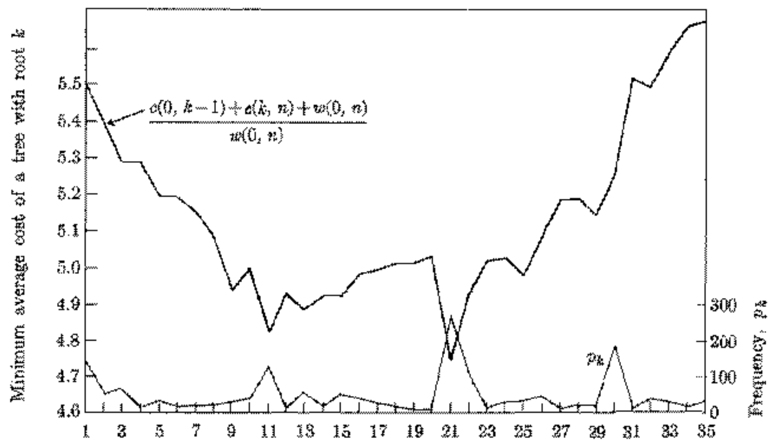


Fig. 17. Behavior of the cost as a function of the root, k.

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.

***The Hu-Tucker algorithm.** In the special case that all the p's are zero, T. C. Hu and A. C. Tucker have discovered a remarkable "bottom-up" way to construct optimum trees; if appropriate data structures are used, their method requires $O(n)$ units of space and $O(n \log n)$ units of time, and it constructs a tree which is *really* optimum (not just approximately so).

The Hu-Tucker algorithm can be described as follows.

• PHASE 1, Combination. Start with the "working sequence" of weights written inside of *external* nodes,

$$\boxed{q_0} \quad \boxed{q_1} \quad \boxed{q_2} \quad \cdots \quad \boxed{q_n}. \tag{18}$$

Then repeatedly combine two weights q_i and q_j for $i < j$ into a single weight $q_i + q_j$, deleting the node containing q_j from the working sequence and replacing the node containing q_i by the *internal* node

$$\left(q_i + q_j\right). \tag{19}$$

This combination is to be done on the unique pair of weights (q_i, q_j) satisfying the following rules:

i) No external nodes occur between q_i and q_j. (This is the most important rule which distinguishes the algorithm from Huffman's method.)

ii) The sum $q_i + q_j$ is minimum over all (q_i, q_j) satisfying rule (i).

iii) The index i is minimum over all (q_i, q_j) satisfying rules (i), (ii).

iv) The index j is minimum over all (q_i, q_j) satisfying rules (i), (ii), (iii).

• PHASE 2, Level assignment. When Phase 1 ends, there is a single node left in the working sequence. Mark it with level number 0. Then undo the steps of Phase 1 in reverse order, marking level numbers of the corresponding tree; if (19) has level l, the nodes containing q_i and q_j which formed it are marked with level $l + 1$.

• PHASE 3, Recombination. Now we have the working sequence of external nodes and levels

$$\boxed{q_0}_{l_0} \quad \boxed{q_1}_{l_1} \quad \boxed{q_2}_{l_2} \quad \cdots \quad \boxed{q_n}_{l_n}.$$

The internal nodes used in Phases 1 and 2 are now discarded, we shall create new ones by combining weights (q_i, q_j) according to the following new rules:

i′) The nodes containing q_i and q_j must be adjacent in the working sequence.

ii′) The levels l_i and l_j must both be the maximum among all remaining levels.

iii′) The index i must be minimum over all (q_i, q_j) satisfying (i′), (ii′).

The new node (19) is assigned level $l_i - 1$. The binary tree formed during this phase has minimum weighted path length over all binary trees whose external nodes are weighted $q_0, q_1, \ldots, q_n$ from left to right.

Figure 18 shows an example of this algorithm; the weights q_i are the relative frequencies of the letters ␣, A, B, ..., Z in English text. During Phase 1, the

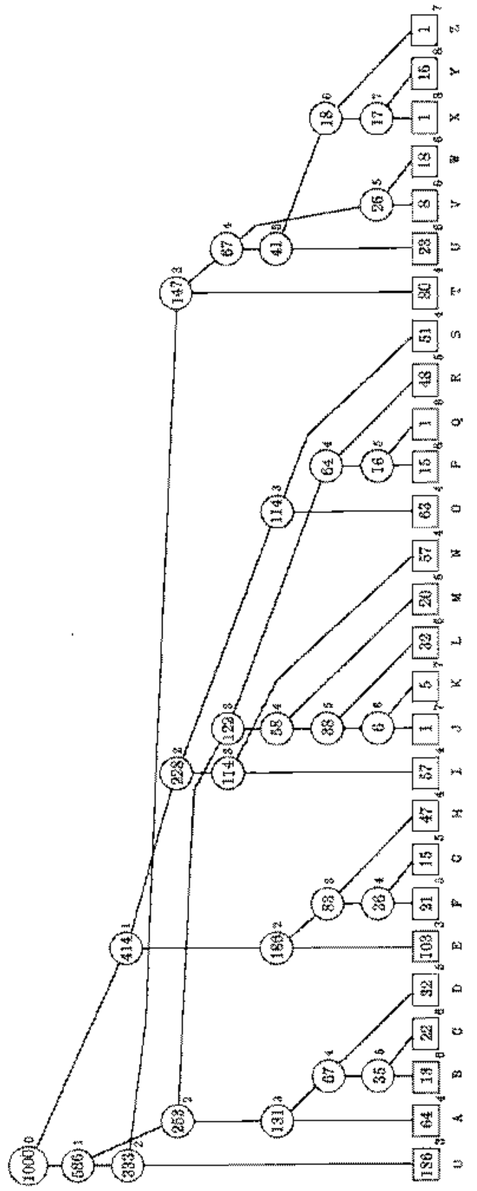


Fig. 18. The Hu-Tucker algorithm applied to alphabetic frequency data: Phases 1 and 2.

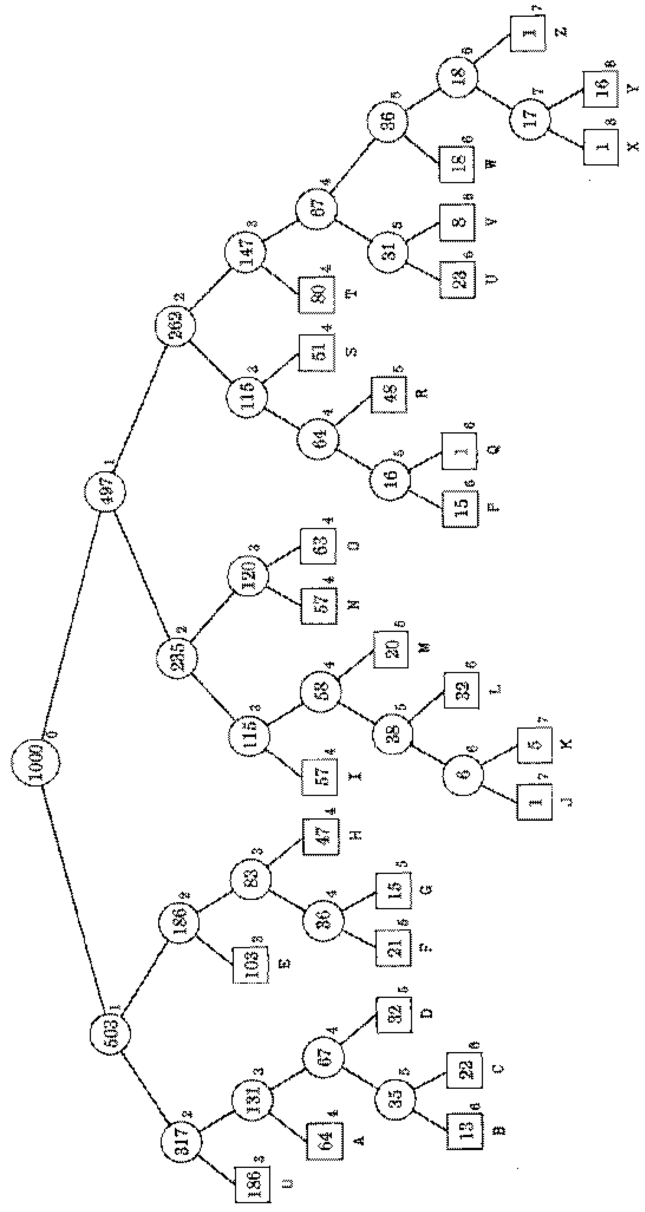


Fig. 19. The Hu-Tucker algorithm applied to alphabetic frequency data: Phase 3.

first node formed is $\textcircled{6}$, combining the J and K frequencies; then the node $\textcircled{16}$ is formed (combining P and Q), then

$$\textcircled{17},\ \textcircled{18},\ \textcircled{26},\ \textcircled{35},\ \textcircled{36},\ \textcircled{38},\ \textcircled{41},\ \textcircled{58},\ \textcircled{64},\ \textcircled{67},\ \textcircled{67},\ \textcircled{83};$$

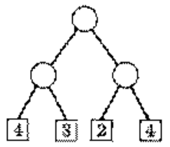
at this point we have the working sequence

$$\boxed{186}\ \boxed{64}\ \textcircled{67}\ \boxed{103}\ \textcircled{83}\ \boxed{57}\ \textcircled{58}\ \boxed{57}\ \boxed{63}\ \textcircled{64}\ \boxed{51}\ \boxed{80}\ \textcircled{67}. \qquad (20)$$

Rule (i) allows us to combine nonadjacent weights only if they are separated by internal nodes; so we can combine $57 + 57$, then $63 + 51$, then $58 + 64$, etc.

The level numbers assigned during Phase 2 appear at the right of each node in Fig. 18. The recombination during Phase 3 now yields the tree shown in Fig. 19; note that things must be associated differently in this tree than in Fig. 18, because Fig. 18 does not preserve the left-to-right ordering. But Fig. 19 has the same cost as Fig. 18, since the external nodes appear at the same levels in both trees.

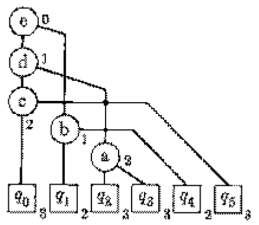
Consider a simple example where the weights are 4, 3, 2, 4; the unique optimum tree is easily shown to be



(21)

This example shows that the two smallest weights, 2 and 3, should *not* always be combined in an optimum tree, even when they are adjacent; some recombination phase is needed.

It is beyond the scope of this book to give a proof that the Hu-Tucker algorithm is valid; no simple proof is known, and it is quite possible that no simple proof will ever be found! In order to illustrate the inherent complexities of the situation, note that Phase 3 must combine all nodes into a single tree, and this is not obviously possible. For example, suppose that Phases 1 and 2 were to construct the tree



(22)

by combining nodes $\textcircled{a}$, $\textcircled{b}$, $\textcircled{c}$, $\textcircled{d}$, $\textcircled{e}$ in this order; this accords with rule (i). Then Phase 3 will get stuck after forming

$$\boxed{q_0}_3 \quad \boxed{q_1}_2 \quad \left(q_2+q_3\right)_2 \quad \boxed{q_4}_2 \quad \boxed{q_5}_3$$

because the two level-3 nodes are not adjacent! Rule (i) does not by itself guarantee that Phase 3 will be able to proceed, and it is necessary to prove that configurations like (22) will *never* be constructed during Phase 1.

When implementing the Hu-Tucker algorithm, we can maintain priority queues for the sets of node weights which are not separated by external nodes. For example, (20) could be represented by priority queues containing, respectively,

$$\begin{array}{cccccccc} \begin{array}{c}64\\186\end{array} & \begin{array}{c}64\\67\\103\end{array} & \begin{array}{c}57\\83\\103\end{array} & \begin{array}{c}57\\57\\58\end{array} & \begin{array}{c}57\\63\end{array} & \begin{array}{c}51\\63\\64\end{array} & \begin{array}{c}51\\80\end{array} & \begin{array}{c}67\\80\end{array} \end{array} \qquad (23)$$

plus information about which of these is external, and an indication of left-to-right order for breaking ties by rules (iii) and (iv). Another "master" priority queue can keep track of the sums of the two least elements in the other queues. The creation of the new node 57 + 57 causes three of the above priority queues to be merged. When priority queues are represented as leftist trees (cf. Section 5.2.3), each combination step of Phase 1 requires at most $O(\log n)$ operations; thus $O(n \log n)$ operations suffice as $n \to \infty$. Of course for small n it will be more efficient to use a comparatively straightforward $O(n^2)$ method of implementation.

The optimum binary tree in Fig. 19 has an interesting 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:

⊔	000	I	1000	R	11001	
A	0010	J	1001000	S	1101	
B	001100	K	1001001	T	1110	
C	001101	L	100101	U	111100	
D	00111	M	10011	V	111101	(24)
E	010	N	1010	W	111110	
F	01100	O	1011	X	11111100	
G	01101	P	110000	Y	11111101	
H	0111	Q	110001	Z	1111111	

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

$$1100110000110101111110000101111010.$$

Note that 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.)

An interesting asymptotic bound on the minimum weighted path length of search trees has been derived by E. N. Gilbert and E. F. Moore:

Theorem G. *If $p_1 = p_2 = \cdots = p_n = 0$, the weighted path length of an optimum binary search tree lies between*

$$\sum_{0\le i\le n} q_i \lg (Q/q_i) \qquad \textit{and} \qquad 2Q + \sum_{0\le i\le n} q_i \lg (Q/q_i),$$

where $Q = \sum_{0\le i\le n} q_i$.

Proof. To get the lower bound, we use induction on n. If $n > 0$ the weighted external path length is at least

$$Q + \sum_{0\le i<k} q_i \lg (Q_1/q_i) + \sum_{k\le i\le n} q_i \lg \bigl((Q - Q_1)/q_i\bigr) = \sum_{0\le i\le n} q_i \lg (Q/q_i) + f(Q_1),$$

for some k, where

$$Q_1 = \sum_{0\le i<k} q_i,$$

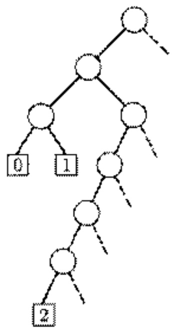
and

$$f(Q_1) = Q + Q_1 \lg Q_1 + (Q - Q_1) \lg (Q - Q_1) - Q \lg Q.$$

The function $f(Q_1)$ is nonnegative, and it takes its minimum value 0 when $Q_1 = \frac{1}{2}Q$.

To get the upper bound, we may assume that $Q = 1$. Let $e_0, \ldots, e_n$ be integers such that $2^{-e_i} \le q_i < 2^{1-e_i}$, for $0 \le i \le n$. Construct codewords C_i of 0's and 1's, by using the most significant $e_i + 1$ binary digits of the fraction $\sum_{0\le k<i} q_k + \frac{1}{2}q_i$, expressed in binary notation. Exercise 35 proves that C_i is never an initial substring of C_j when $i \ne j$; it follows that we can construct a binary search tree corresponding to these codewords. For example when the q's are the letter frequencies of Fig. 19, this construction gives $C_0 = 0001$,

$C_1 = 00110$, $C_2 = 01000001$, $C_3 = 0100011$, etc.; the tree begins



(25)

(Redundant bits at the right end of the codes can often be removed.) The weighted path length of the binary tree constructed by this general procedure is

$$\le \sum_{0\le i\le n} (e_i+1)q_i < \sum_{0\le i\le n} q_i(2+\lg(1/q_i)). \quad \blacksquare$$

The first part of the above proof is readily extended to show that the weighted path length of *every* binary tree must be at least $\sum_{0\le i\le n} q_i \lg(Q/q_i)$, whether or not the weights are required to be in order from left to right. (This fundamental result is due to Claude Shannon.) Therefore the left-to-right constraint does not raise the cost of the minimum tree by more than the cost of two extra levels, i.e., twice the total weight.

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

"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 $2^{n-1}+2^{n-2}$. If 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 also independently devised by C. M. Berners-Lee [see *Comp. J.* 2 (1959), 5].

GN 005685

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]. All three of these authors seem to have developed the method independently of one another, and all three authors gave somewhat different proofs of the average number of comparisons (6). The three 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 (24). 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 observed, among other things, that an optimum tree could be constructed in $O(n^3)$ steps, using a method like Algorithm K but without 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 $p = 0$ case 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 in 1969 [*SIAM J. Applied Math.* **21** (1971), 514–532].

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. (Cf. Section 2.3.1; symmetric traversal is easier in such trees.)

▶ 3. [*20*] In Section 6.1 we saw 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).]

GN 005686

▶ **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 $a_1a_2 \ldots a_n$ 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 $C'_{n-1} = (\sum kP_{nk})/n!$, since this is the average number of comparisons made in an unsuccessful search of a tree containing $n - 1$ elements.)

a) Prove that $P_{n+1,k} = 2P_{n,k-1} + (n-1)P_{n,k}$. [*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 C'_{n-1}?

7. [*M30*] (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 to find the mth largest 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 which 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*] Suppose that a programmer is not interested in the sorting property of this algorithm, but he expects the input will come in with nonrandom order. Discuss methods by which he can still use the tree search by making the input "appear to be" in random order.

11. [*20*] What is the maximum number of times $\text{S} \leftarrow \text{LLINK(R)}$ can be performed in step D3 when deleting a node from a tree of size N?

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 D1$\frac{1}{2}$ added is never more than the path length of the tree produced without that step. Find a case where step D1$\frac{1}{2}$ actually decreases the path length.

15. [*23*] Let $a_1a_2a_3a_4$ be a permutation of $\{1, 2, 3, 4\}$, and let $1 \leq j \leq 3$. 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. (Cf. 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 Eqs. 6.1–11, 12?

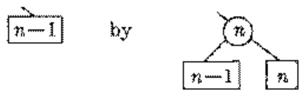
20. [*M20*] Suppose we have inserted keys into a tree in order of decreasing frequency $p_1 \geq p_2 \geq \cdots \geq p_N$. Can this tree be substantially worse than the optimum search tree?

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 $p_1 = 5$, $p_2 = p_3 = \cdots = p_{40} = 1$, $q_0 = q_1 = \cdots = q_{40} = 0$. (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:

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

(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 "convex, continuous, piece-wise 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

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

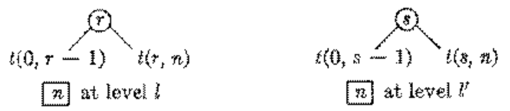
in terms of the relation defined in exercise 25, whenever the weights $(p_1, \ldots, p_n; q_0, \ldots, q_n)$ are nonnegative. The proof is by induction on $j - i$; our task is to prove that $R(0, n-1) \le R(0, n)$, assuming that $n \ge 2$ and that the above relation holds for $j - i < n$. [By left-right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

a) Prove that $R(0, n-1) \le R(0, n)$ 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

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

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 ⓡ such that $\boxed{n}$ is at level l', and an optimum tree with root ⓢ such that $\boxed{n}$ is at level l.]

28. [*24*] Use some macro-assembly language to define a "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. [*M46*] Prove or disprove that the costs of optimum binary search trees satisfy $c(i, j) + c(i+1, j-1) \ge c(i, j-1) + c(i+1, j)$.

31. [*M20*] (a) If the weights $(q_0, \ldots, q_5)$ in (22) are (2, 3, 1, 1, 3, 2), respectively, what is the weighted path length of the tree? (b) What is the weighted path length of the *optimum* binary search tree having this sequence of weights?

▸ **32.** [*M22*] (T. C. Hu and A. C. Tucker.) Prove that the new node weights $q_i + q_j$ formed during Phase 1 of the Hu-Tucker algorithm are created in nondecreasing order.

33. [*M41*] In order to find the binary search tree which 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 which 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. Cf. L. E. Stanfel, *JACM* 17 (1970), 508–517.)

34. [*41*] Write a computer program for the Hu-Tucker algorithm, using $O(n)$ units of storage and $O(n \log n)$ units of time.

35. [*M23*] Show that the codewords constructed in the proof of Theorem G have the property that C_i never begins with C_j when $i \ne j$.

36. [*M26*] Generalizing the upper bound of Theorem G, prove that the cost of any optimum binary search tree with nonnegative weights must be less than

$$2S + q_0 \lg (S/q_0) + \sum_{1 \le i \le n} (p_i + q_i) \lg \bigl(S/(p_i + q_i)\bigr),$$

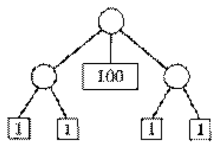
where $S = q_0 + \sum_{1 \le i \le n} (p_i + q_i)$ is the total weight.

▸ **37.** [*M25*] (T. C. Hu and K. C. Tan.) Let $n+1 = 2^m + k$, where $0 \le k \le 2^m$. There are $\binom{2^m}{k}$ 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 weighted path length for the weight sequence $(q_0, \ldots, q_n)$ if we apply the Hu-Tucker algorithm to the weights $(M+q_0, \ldots, M+q_n)$ for sufficiently large M.

38. [*M35*] (K. C. Tan.) Prove that, among all 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. [*Hint:* Given arbitrary probabilities $(p_1, \ldots, p_n; q_0, \ldots, q_n)$, let $c_0 = q_0$, $c_i = p_i + q_i$ for $1 \le i \le n$, and $S(0) = \emptyset$; and for $1 \le r \le \lceil n/2 \rceil$ let $S(r) = S(r-1) \cup \{i, j\}$, where $c_i + c_j$ is minimum over all $i < j$ such that $i, j \notin S(r-1)$ and $k \in S(r-1)$ for all $i < k < j$. Construct the binary tree T with the external nodes of $S(n+1-2^q)$ on level $q+1$ and the other external nodes on level q, where $q = \lfloor \lg n \rfloor$. Prove that the cost of this tree is $\le f(n)$, where $f(n)$ is the cost of the optimum search tree for the stated "worst" probabilities.]

39. [*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 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 $O(N)$.)

40. [*M50*] Can the Hu-Tucker algorithm be generalized to find optimum trees where each node has degree at most t? For example, when $t = 3$ and the sequence of weights is (1, 1, 100, 1, 1) the optimum tree is



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 which 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 this 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. Adel'son-Vel'skiĭ and E. M. Landis [*Doklady Akademiia 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

GN 005690

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 $O(\log(M+N))$ steps.

The method for achieving all this involves what we shall call "balanced trees." 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 which do not involve all four of the above operations, 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 $+$, $\cdot$, 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 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*

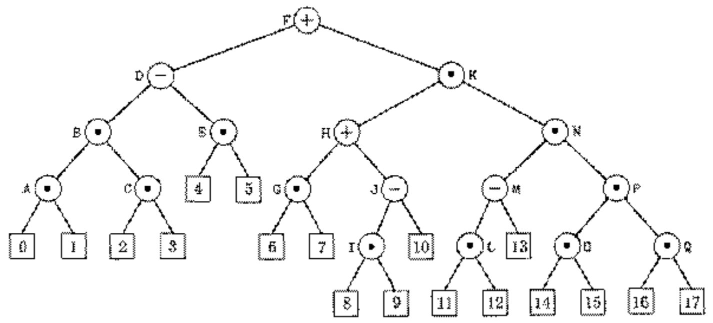


Fig. 20. A balanced binary tree.

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 (Adel'son-Vel'skiĭ and Landis). *The height of a balanced tree with N internal nodes always lies between* $\lg (N+1)$ *and* $1.4404 \lg (N+2) - 0.328$.

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

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

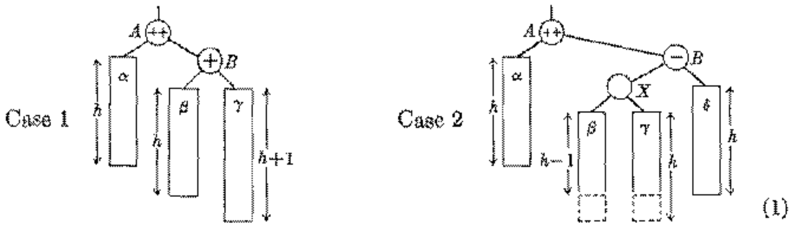
and the stated result follows as in the corollary to Theorem 4.5.3F. ▌

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 $F_{27} - 1 = 196{,}417$ 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

GN 005692

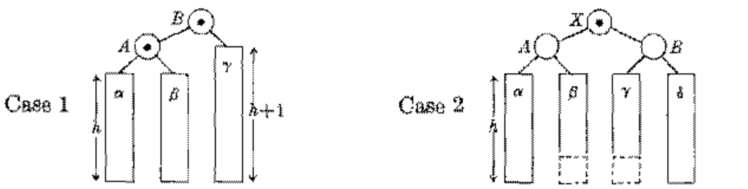
subtree got higher. It is not difficult to see that there are essentially only two cases which cause trouble:



(1)

(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 the above cases, while preserving the symmetric order of the tree nodes:



(2)

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). Note 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 avoiding the need for a stack in a slightly tricky way.

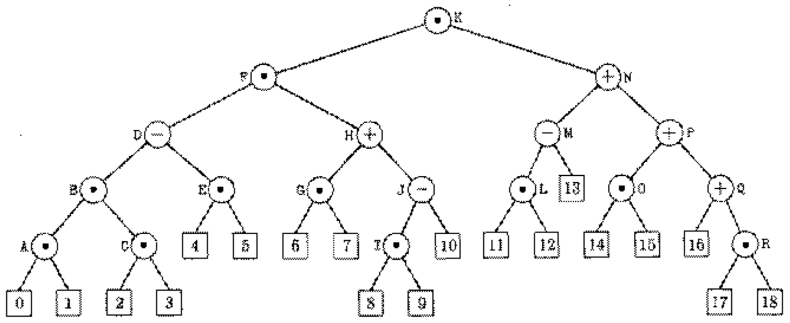


Fig. 21. The tree of Fig. 20, rebalanced after a new key R has been inserted.

Algorithm A (*Balanced tree search and insertion*). Given a table of records which 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

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

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*, i.e., 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 father 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 P $\leftarrow$ Q and return to step A2. (The last part of this step may be combined with the last part of step A3.)

GN 005694

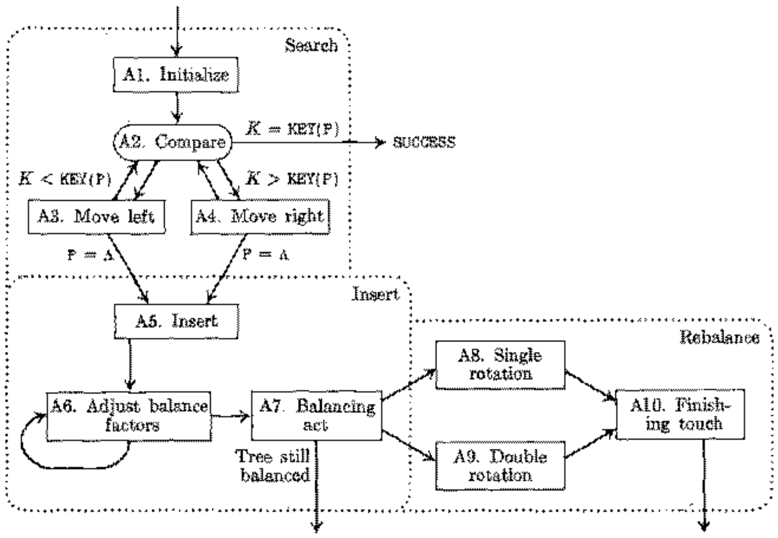


Fig. 22. Balanced tree search and insertion.

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, `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 `R` ← `P` ← `LLINK(S)`, otherwise set `R` ← `P` ← `RLINK(S)`. Then repeatedly do the following operation 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 may go on to the next step.)

A7. [Balancing act.] If $K <$ `KEY(S)` set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. 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.)

GN 005695

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

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

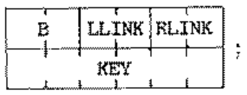
and then set $B(P) \leftarrow 0$.

A10. [Finishing touch.] (We have completed the rebalancing transformation, taking (1) to (2), with P pointing to the new root and T pointing to the father of the old root.) 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.

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



; (4)

rA ≡ K, rI1 ≡ P, rI2 ≡ Q, rI3 ≡ R, rI4 ≡ S, rI5 ≡ 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.
```

```
        20      5H      LD2    AVAIL               1 − S      A5. Insert.
        21              J2Z    OVERFLOW            1 − S
        22              LDX    0,2(RLINK)          1 − S
        23              STX    AVAIL               1 − S      Q ⇐ AVAIL.
        24              STA    1,2                 1 − S      KEY(Q) ← K.
        25              STZ    0,2                 1 − S      LLINK(Q) ← RLINK(Q) ← Λ.
        26              JL     1F                  1 − S      Was K < KEY(P)?
        27              ST2    0,1(RLINK)            A        RLINK(P) ← Q.
        28              JMP    *+2                   A
        29      1H      ST2    0,1(LLINK)        1 − S − A    LLINK(P) ← Q.
        30      6H      CMPA   1,4                 1 − S      A6. Adjust balance factors.
        31              JL     *+3                 1 − S      Jump if K < KEY(S).
        32              LD3    0,4(RLINK)            E        R ← RLINK(S).
        33              JMP    *+2                   E
        34              LD3    0,4(LLINK)        1 − S − E    R ← LLINK(S).
        35              ENT1   0,3                 1 − S      P ← R.
        36              ENTX   −1                  1 − S      rX ← −1.
        37              JMP    1F                  1 − S      To comparison loop.
        38      4H      JE     7F                F2 + 1 − S   To A7 if K = KEY(P).
        39              STX    0,1(1:1)              F2       B(P) ← +1 (it was +0).
        40              LD1    0,1(RLINK)            F2       P ← RLINK(P).
        41      1H      CMPA   1,1               F + 1 − S
        42              JGE    4B                F + 1 − S    Jump if K ≥ KEY(P).
        43              STX    0,1(B)                F1       B(P) ← −1.
        44              LD1    0,1(LLINK)            F1       P ← LLINK(P).
        45              JMP    1B                    F1       To comparison loop.
        46      7H      LD2    0,4(B)              1 − S      A7. Balancing act. rI2 ← B(S).
        47              STZ    0,4(B)              1 − S      B(S) ← 0.
        48              CMPA   1,4                 1 − S
        49              JG     A7R                 1 − S      To a = +1 routine if K > KEY(S).

50  A7L  J2P   DONE           68  A7R   J2N   DONE          1 − S    Exit if rI2 = −a.
51       J2Z   7F             69        J2Z   6F            G + J    Jump if B(S) was zero.
52       ENT1  0,3            70        ENT1  0,3             G      P ← R.
53       LD2   0,3(B)         71        LD2   0,3(B)          G      rI2 ← B(R).
54       J2N   A8L            72        J2P   A8R             G      To A8 if rI2 = a.
55  A9L  LD1   0,3(RLINK)     73  A9R   LD1   0,3(LLINK)      H      A9. Double rotation.
56       LDX   0,1(LLINK)     74        LDX   0,1(RLINK)      H      LINK(a, P ← LINK(−a, R))
57       STX   0,3(RLINK)     75        STX   0,3(LLINK)      H      → LINK(−a, R).
58       ST3   0,1(LLINK)     76        ST3   0,1(RLINK)      H      LINK(a, P) ← R.
59       LD2   0,1(B)         77        LD2   0,1(B)          H      rI2 ← B(P).
60       LDX   T1,2           78        LDX   T2,2            H      −a, 0, or 0
61       STX   0,4(B)         79        STX   0,4(B)          H      → B(S).
62       LDX   T2,2           80        LDX   T1,2            H      0, 0, or a
63       STX   0,3(B)         81        STX   0,3(B)          H      → B(R).
64  A8L  LDX   0,1(RLINK)     82  A8R   LDX   0,1(LLINK)      G      A8. Single rotation.
65       STX   0,4(LLINK)     83        STX   0,4(RLINK)      G      LINK(a, S) ← LINK(−a, P).
66       ST4   0,1(RLINK)     84        ST4   0,1(LLINK)      G      LINK(−a, P) ← S.
67       JMP   A8R1           85  A8R1  STZ   0,1(B)          G      B(P) ← 0.

        86      A10     CMP4   0,5(RLINK)            G        A10. Finishing touch.
        87              JNE    *+3                   G        Jump if RLINK(T) ≠ S.
        88              ST1    0,5(RLINK)            G2       RLINK(T) ← P.
        89              JMP    DONE                  G2       Exit.
        90              ST1    0,5(LLINK)            G1       LLINK(T) ← P.
        91              JMP    DONE                  G1       Exit.
        92              CON    +1
        93      T1      CON    0                              Table for (3)
        94      T2      CON    0
        95              CON    −1
        96      6H      ENTX   +1                    J2       rX ← +1.
        97      7H      STX    0,4(B)                J        B(S) ← a.
        98              LDX    HEAD(LLINK)           J        LLINK(HEAD)
        99              INCX   1                     J        + 1
       100              STX    HEAD(LLINK)           J        → LLINK(HEAD).
       101      DONE    EQU    *                   1 − S      Insertion is complete. ▌
```

GN 005697

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 rather complicated, but some interesting 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 $B_h(z) = \sum_{n\geq 0} B_{nh}z^n$ 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

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

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 $h \geq 3$, 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,

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

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 $\theta \approx 1.43684$ such that

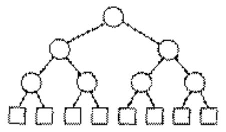
$$B_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor. \tag{9}$$

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

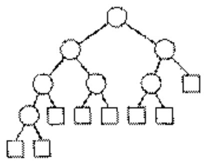
$$B_h'(1)/B_h(1) \approx (0.70118)2^h. \qquad (10)$$

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

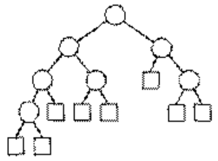
Unfortunately, none of these results have much to do with Algorithm A, since the mechanism of that algorithm makes some trees much more probable than others. For example, consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys can 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 somewhat 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 extremely plausible that the average search time for a balanced tree is about $\lg N + c$ comparisons for some small constant c. Empirical tests support this conjecture: The average number of comparisons needed to insert the Nth item seems to be approximately $\lg N + 0.25$ for large N.

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

GN 005699

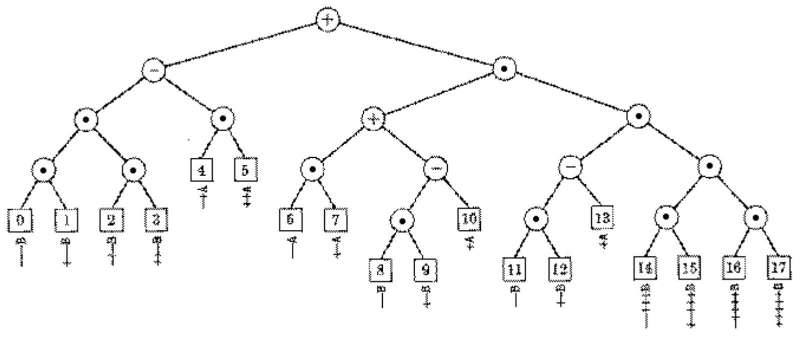


Fig. 23. Classification codes which specify the behavior of Algorithm A after insertion.

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 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 governing 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.

Table 1

APPROXIMATE PROBABILITIES FOR INSERTING THE NTH ITEM

Path length k	No rebalancing	Single rotation	Double rotation
1	.144	.000	.000
2	.153	.144	.144
3	.093	.048	.048
4	.058	.023	.023
5	.036	.010	.010
> 5	.051	.008	.007
ave 2.8	.535	.233	.232

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

From Table 1 we can see that k is ≤ 2 with probability $.144 + .153 + .144 + .144 = .585$; 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 about 1.8. The average number of balance factors changed from ± 1 to 0 in steps A7 through A10 is $.535 + 2(.233 + .232) = 1.5$; thus, inserting one new node adds about .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 $k-1$ is $p/(1-p)$. The probability that we need to rotate part of the tree is $\frac{1}{2}$. 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 $\frac{1}{2}$ in step A7, and by $\frac{1}{2}\cdot 2$ in step A8 or A9, so we should have

$$p = 1 - p/(1-p) + \tfrac{1}{2} + 1.$$

Solving for p yields fair agreement with Table 1:

$$p = \frac{9-\sqrt{41}}{4} \approx 0.649; \qquad p/(1-p) \approx 1.851. \tag{14}$$

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 $D \approx \frac{1}{3}C$, $C1 \approx \frac{1}{2}(C + S)$, $C + S \approx \lg N + 0.25$, so the average search time is approximately $11.17 \lg N + 4.8 - 13.7S$ 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 $(6.5 \lg N + 4.1)u$, and the worst case running time would in fact be better than the average running time obtained with Algorithm 6.2.2T.)

The running time of the insertion phase of Program A (lines 20–45) is $8F + 26 + (0, 1,$ 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 .535, .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 6.2.2 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. This field indicates the relative position of that node in its subtree, i.e., one plus the

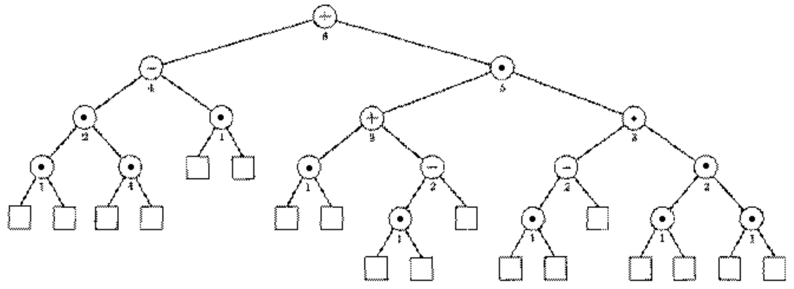


Fig. 24. RANK fields, used for searching by position.

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 $\texttt{M} \leftarrow k$, $\texttt{P} \leftarrow \texttt{RLINK(HEAD)}$.

B2. [Compare.] If $\texttt{P} = \Lambda$, 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 $\texttt{M} < \texttt{RANK(P)}$, go to B3; if $\texttt{M} > \texttt{RANK(P)}$, go to B4; and if $\texttt{M} = \texttt{RANK(P)}$, the algorithm terminates successfully (P points to the kth node).

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

B4. [Move right.] Set $\texttt{M} \leftarrow \texttt{M} - \texttt{RANK(P)}$ and $\texttt{P} \leftarrow \texttt{RLINK(P)}$ 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 $\texttt{T} \leftarrow \texttt{HEAD}$, $\texttt{S} \leftarrow \texttt{P} \leftarrow \texttt{RLINK(HEAD)}$, $\texttt{U} \leftarrow \texttt{M} \leftarrow k$.

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

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

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

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

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

C7. [Balancing act.] If $U <$ RANK(S), set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. 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 $P \leftarrow R$, 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).

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 which specifies 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!

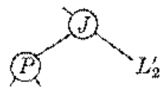
The situations requiring rebalancing are almost the same as we met in the insertion algorithm; referring again to (1), A is node P_k, and B is node LINK$(-a_k, P_k)$, the *opposite* branch from where the deletion has occurred. The only new feature is that node B might be balanced; this leads to a new 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 has been 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 $L_2\backslash\{J\}$. 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 has 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 con-

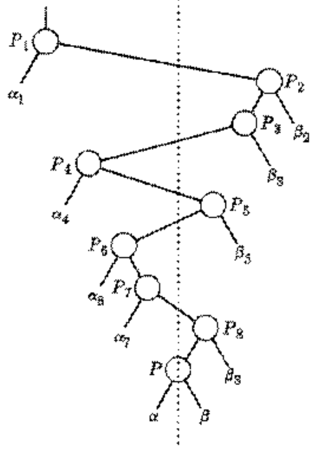


Fig. 25. The problem of splitting a list.

struct a left tree which 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 containing $\beta, P_8, \beta_8, P_5, \beta_5, P_3, \beta_3, P_2, \beta_2$. 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 has 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 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 which specifies whether a node is the left or right son of its father. The triply-linked tree representation simplifies the algorithms slightly, and makes it possible 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(P\$) which follows 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 that a few more links be changed when insertions, deletions, and rotations are being performed. The use of a triply-

GN 005706

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 balanced trees. Some other ways of organizing trees, so as to guarantee logarithmic accessing time, have recently been proposed. At the present time they have not yet been studied very thoroughly; it is possible that they may prove to be superior to balanced trees on some computers.

For example, C. C. Foster [*CACM* **16** (1973), 513–517] has studied the generalized balanced trees which arise when we allow the height difference of subtrees to be greater than one, but at most four (say).

The interesting concept of *weight-balanced tree* has been studied by J. Nievergelt, E. Reingold, and C. K. Wong. Instead of considering the height of trees, we 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 this small savings of two bits per node is probably not worth the trouble.

Another interesting alternative to balanced trees, called "2-3 trees," was introduced by John Hopcroft in 1970 (unpublished). 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.

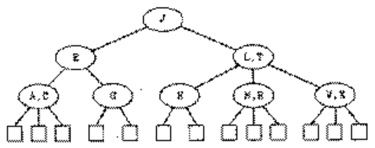


Fig. 26. A 2-3 tree.

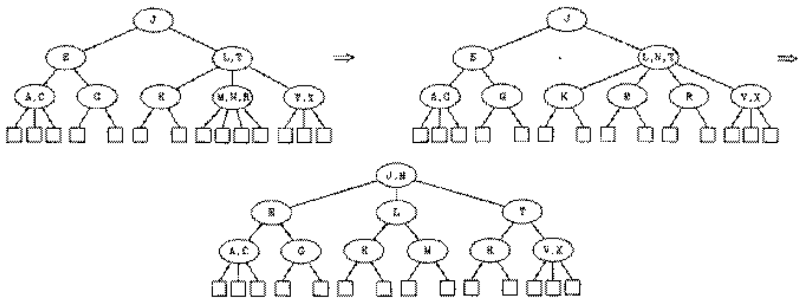


Fig. 27. Inserting the new key "M" into the 2–3 tree of Fig. 26.

Insertion into a 2-3 tree is somewhat easier to explain than insertion into a balanced 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.

Hopcroft has 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 balanced trees.

R. Bayer [*Proc. ACM-SIGFIDET Workshop* (1971), 219–235] has suggested 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 a balanced tree, although we need just one version of each rotation (not the left-right reflections as in Algorithms A and C).

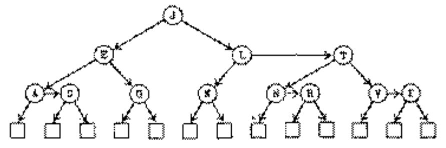


Fig. 28. The 2–3 tree of Fig. 26 represented as a binary search tree.

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 B(S) = 0.

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

4. [*M22*] Prove or disprove: Among all balanced trees with $F_{n+1} - 1$ internal nodes, the Fibonacci tree of order n 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 which initially contains only the single key K_1, where $K_1 < K_2 < \cdots < K_N$, then the tree produced is always *optimum* (i.e., 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 $B_h'(1)/B_h(1) - 2^h\beta \to 1$ as $h \to \infty$.

9. [*M47*] 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. [*M48*] Does Algorithm A make an average of $\sim\lg N + c$ comparisons to insert the Nth item, for some constant c?

▸ **11.** [*22*] The value .144 appears three times in Table 1, once for $k = 1$ and twice for $k = 2$. The value $\frac{1}{7}$ appears in the same three places in Table 2. Is it a coincidence that the same value should appear in all three places, or is there some good reason for this?

▸ **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. [*10*] 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 which searches the tree for a given key, K, and determines the position of K in the list; i.e., it finds the number M such that K is the Mth smallest key.

▸ **16.** [*20*] Draw the balanced tree that would be obtained if the root node F were deleted from Fig. 20, using the deletion algorithm suggested in the text.

▸ **17.** [*21*] Draw the balanced trees that would be obtained if the Fibonacci tree (12) were concatenated (a) to the right, (b) to the left, of the tree in Fig. 20, using the concatenation algorithm suggested in the text.

18. [*21*] Draw the balanced trees that would be obtained if Fig. 20 were split into two parts {A, ..., I} and {J, ..., Q}, using the splitting algorithm suggested in the text.

GN 005709

▸**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*] Design an algorithm which constructs optimum N-node binary trees (in the sense of exercise 5), in $O(N)$ steps. Your algorithm should be "on line," 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.) (a) Prove that there exist balanced trees whose weight balance (left weight)/(right weight) is arbitrarily small. (b) Prove that there exist weight-balanced trees having arbitrarily large differences 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 which 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 which 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 in Section 2.5 take order n steps.)

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 prob-

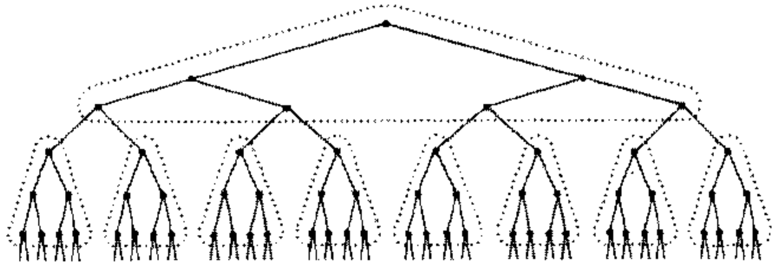


Fig. 29. A large binary search tree can be divided into "pages."

lem 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 naive manner, simply applying the algorithms we have learned for internal tree searching, we will have to make about $\log_2 N$ 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 we never have more than 254 keys in the internal memory 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 longer time to read in a larger 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 be about $a + b \log m$, 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 $\log N$ times

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

This quantity achieves a minimum when $m \approx 350$; 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 [cf. *JACM* **16** (1969), 569–571].

R. Muntz and R. Uzgalis [*Proc. Princeton Conf. on Inf. Sciences and Systems* **4** (1970), 345–349] have 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 father 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 10.)

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 Informatica* (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 which satisfies the following properties:

i) Every node has $\leq m$ sons.
ii) Every node, except for the root and the leaves, has $\geq m/2$ sons.
iii) The root has at least 2 sons (unless it is a leaf).
iv) All leaves appear on the same level, and carry no information.
v) A nonleaf node with k sons contains $k-1$ keys.

(As usual, a "leaf" is a terminal node, one with no sons. Since the leaves carry no information, we may regard them as external nodes which 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 sons, 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. Note that (a) the keys appear in increasing order from

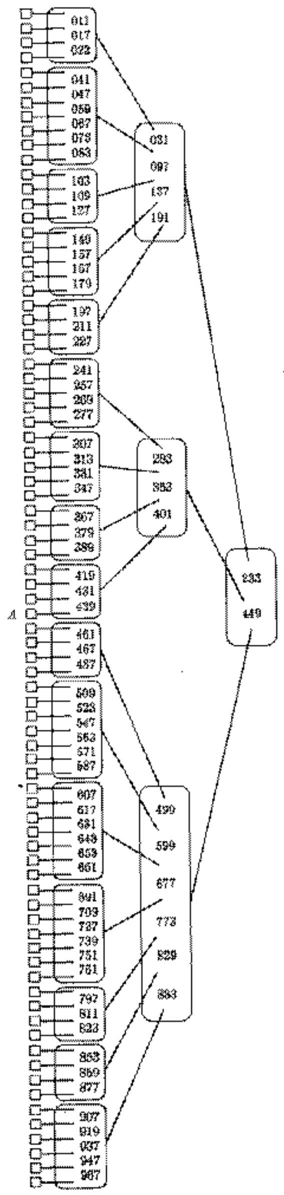
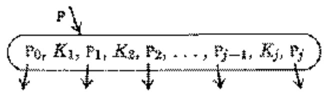


Fig. 30. A *B*-tree of order 7, with all leaves on level 3.

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 $m \geq 3$. Note that "2-3 trees," defined at the close of Section 6.2.3, are B-trees of order 3; and conversely, a B-tree of order 3 is a 2-3 tree.

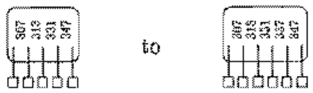
A node which contains j keys and $j + 1$ pointers can be represented as



(1)

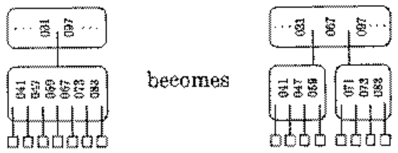
where $K_1 < K_2 < \cdots < K_j$ and P_i points to the subtree for keys between K_i and K_{i+1}. Therefore searching in a B-tree is quite straightforward: After node (1) has been fetched into the internal memory, we search for the given argument among the keys $K_1, K_2, \ldots, K_j$. (When j is large, 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 $P_i = \Lambda$, 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 appropriate node from



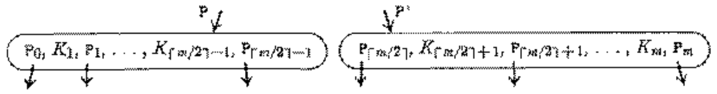
(2)

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
P0, K1, P1, . . . , K⌈m/2⌉−1, P⌈m/2⌉−1
P′
P⌈m/2⌉, K⌈m/2⌉+1, P⌈m/2⌉+1, . . . , Km, Pm

(4)

and insert the key $K_{\lceil m/2 \rceil}$ into the father of the original node. (Thus the pointer P in the father node is replaced by the sequence P, $K_{\lceil m/2 \rceil}$, P′.) This insertion may cause the father node to contain m keys, and if so, it should be split in the same way. (Cf. Fig. 27, which shows the case $m = 3$.) If we need to split the root node, which has no father, 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. Note that the tree more or less 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 7).

Upper bounds on the performance. 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, ... is at least 2, $2\lceil m/2 \rceil$, $2\lceil m/2 \rceil^2$, ...; hence

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

In other words,

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

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 node 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 l less than the total number of nodes in the tree. If there are p nodes, there are at least $1 + (\lceil m/2 \rceil - 1)(p - 1)$ keys; hence

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

It follows that the average number of times we need to split a node 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"

GN 005715

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, ... by requiring all nonroot nodes on level $l - i$ to have between $m_i/2$ and m_i sons; 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 $439 < K \leq 449$; 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.

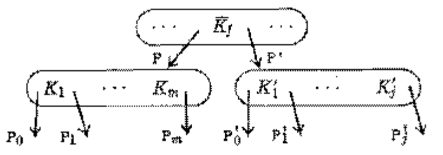
Some calculations by S. P. Ghosh and M. E. Senko [*JACM* **16** (1969), 569–579] suggest that it might be a good idea to make the leaves fairly large, say up to about 10 consecutive pages long. By linear interpolation in the known range of keys for each leaf, we can guess which of the 10 pages probably 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 sons 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 contains m keys and $m + 1$ pointers; instead of splitting it, we can look first at its brother node on the right, which has say j keys and $j + 1$ pointers. In

GN 005716

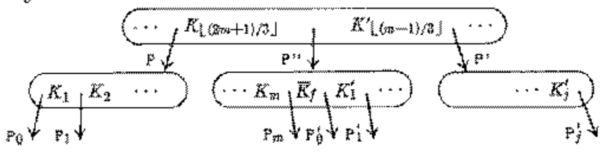
the father node there is a key $\overline{K}_f$ which separates the keys of the two brothers; schematically,



$\cdots$ $\overline{K}_f$ $\cdots$
P
P'
K_1 $\cdots$ K_m
K'_1 $\cdots$ K'_j
P_0 P_1 P_m P'_0 P'_1 P'_j

(8)

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 father 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 brother node. On the other hand, if the brother 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:



$\cdots$ $K_{\lfloor (2m+1)/3 \rfloor}$ $K'_{\lfloor (m-1)/3 \rfloor}$ $\cdots$
P
P''
P'
K_1 K_2 $\cdots$
$\cdots$ K_m $\overline{K}_f$ K'_1 $\cdots$
$\cdots$ K'_j
P_0 P_1 P_m P'_0 P'_1 P'_j

(9)

If the original node has no right brother, we can look at its left brother in essentially the same way. (If the original node has both a right and a left brother, we could even refrain from splitting off a new node unless *both* left and right brothers are full.) Finally if the original node to be split has no brothers 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 sons.
ii) Every node, except for the root and the leaves, has $\geq (2m-1)/3$ sons.
iii) The root has at least 2 and at most $2\lfloor (2m-2)/3 \rfloor + 1$ sons.
iv) All leaves appear on the same level.
v) A nonleaf node with k sons 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).

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 which 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 sons, the first level pages will probably stay there too. Some special mechanism could be incorporated to ensure that a certain minimum number of pages near the root are always present. Note that the least-recently-used scheme implies that the pages that might need to be split during an insertion are automatically in memory when they are needed.

Some experiments by E. McCreight have shown that this idea is quite successful. For example, he found that with 10 page-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 (9). (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.

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 (9).

▸ **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 (9)?

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 $\frac{3}{4}m - \frac{1}{2}$ sons.

▸ **5.** [*21*] Suppose that a node represents 1000 character positions of external memory. If each pointer takes up 5 characters, and if the keys are variable in length, between 5 and 50 characters long, 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.")

6. [*22*] Can the *B*-tree idea be used to retrieve items of a linear list by position instead of by key value? (Cf. Algorithm 6.2.3B)

7. [*23*] Design a deletion algorithm for *B*-trees.

8. [*28*] Design a concatenation algorithm for *B*-trees (cf. Section 6.2.3).

9. [*30*] Discuss how a large file, organized as a *B*-tree, can be used for multiple accessing and updating by a large number of simultaneous users, in such a way that users of different pages rarely interfere with each other.

10. [*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 father 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 $B_1^{(j)}(z) = \delta_{j1}z$;

$$B_N^{(j)}(z) = \frac{N-j-1}{N+1}\,B_{N-1}^{(j)}(z) + \frac{j+1}{N+1}\,B_{N-1}^{(j-1)}(z), \quad \text{for} \quad 1 < j < M;$$
$$B_N^{(1)}(z) = \frac{N-2}{N+1}\,B_{N-1}^{(1)}(z) + \frac{2z}{N+1}\,B_{N-1}^{(M)}(z);$$
$$B_N^{(M)}(z) = \frac{N-1}{N+1}\,B_{N-1}^{(M)}(z) + \frac{M+1}{N+1}\,B_{N-1}^{(M-1)}(z).$$

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

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

and relate C'_N to an Nth degree polynomial in $\mathbf{W}(1)$.]

6.3. DIGITAL SEARCHING

Instead of basing a search method on comparisons between keys, we can make use of their representation as a sequence of digits or alphabetic characters. Consider, for example, the "thumb-index" on a large dictionary; from the first letter of a given word, we can immediately locate the pages which contain all words beginning with that letter.

If we pursue the thumb-index idea to one of its logical conclusions, we come up with a searching scheme based on repeated "subscripting" as illustrated in Table 1. Suppose that we want to test a given search argument to see whether it is one of the 31 most common words of English (cf. Figs. 12 and 13 in Section 6.2.2). The data is represented in Table 1 as a so-called "trie" structure; this name was suggested by E. Fredkin [*CACM* **3** (1960), 490–500] because it is a part of information re*trie*val. A trie is essentially an M-ary tree, whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level l represents the set of all keys that begin with a certain sequence of l characters; the node specifies an M-way branch, depending on the $(l+1)$st character.

For example, the trie of Table 1 has 12 nodes; node (1) is the root, and we look up the first letter here. If the first letter is, say, N, the table tells us that our word must be NOT (or else it isn't in the table). On the other hand, if the first letter is W, node (1) tells us to go on to node (9), looking up the second letter in the same way; node (9) says that the second letter should be A, H, or I.

The node vectors in Table 1 are arranged according to MIX character code. This means that a trie search will be quite fast, since we are merely fetching words of an array by using the characters of our keys as subscripts. Techniques for making quick multiway decisions by subscripting have been called "Table Look-At" as opposed to "Table Look-Up" [see P. M. Sherman, *CACM* **4** (1961), 172–173, 175].

Algorithm T (*Trie search*). Given a table of records which form an M-ary trie, this algorithm searches for a given argument K. The nodes of the trie are vectors whose subscripts run from 0 to $M-1$; each component of these vectors is either a key or a link (possibly null).

T1. [Initialize.] Set the link variable P so that it points to the root of the tree.

T2. [Branch.] Set k to the next character of the input argument, K, from left to right. (If the argument has been completely scanned, we set k to a "blank" or end-of-word symbol. The character should be represented as a number in the range $0 \le k < M$.) Let X be table entry number k in NODE(P). If X is a link, go to T3; but if X is a key, go to T4.

T3. [Advance.] If $X \neq \Lambda$, set $\text{P} \leftarrow X$ and return to step T2; otherwise the algorithm terminates unsuccessfully.

T4. [Compare.] If $K = X$, the algorithm terminates successfully; otherwise it terminates unsuccessfully. ▮

GN 005720

Table 1

A 'TRIE' FOR THE 31 MOST COMMON ENGLISH WORDS

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
␣	—	A	—	—	—	I	—	—	—	—	HE	—
A	(2)	—	—	—	(10)	—	—	—	WAS	—	—	THAT
B	(3)	—	—	—	—	—	—	—	—	—	—	—
C	—	—	—	—	—	—	—	—	—	—	—	—
D	—	—	—	—	—	—	—	—	—	HAD	—	—
E	—	—	BE	—	(11)	—	—	—	—	—	—	THE
F	(4)	—	—	—	—	—	OF	—	—	—	—	—
G	—	—	—	—	—	—	—	—	—	—	—	—
H	(5)	—	—	—	—	—	—	(12)	WHICH	—	—	—
I	(6)	—	—	—	HIS	—	—	—	WITH	—	—	THIS
Θ	—	—	—	—	—	—	—	—	—	—	—	—
J	—	—	—	—	—	—	—	—	—	—	—	—
K	—	—	—	—	—	—	—	—	—	—	—	—
L	—	—	—	—	—	—	—	—	—	—	—	—
M	—	—	—	—	—	—	—	—	—	—	—	—
N	NOT	AND	—	—	—	IN	ON	—	—	—	—	—
O	(7)	—	—	FOR	—	—	—	TO	—	—	—	—
P	—	—	—	—	—	—	—	—	—	—	—	—
Q	—	—	—	—	—	—	—	—	—	—	—	—
R	—	ARE	—	FROM	—	—	OR	—	—	—	HER	—
Φ	—	—	—	—	—	—	—	—	—	—	—	—
Π	—	—	—	—	—	—	—	—	—	—	—	—
S	—	AS	—	—	—	IS	—	—	—	—	—	—
T	(8)	AT	—	—	—	IT	—	—	—	—	—	—
U	—	—	BUT	—	—	—	—	—	—	—	—	—
V	—	—	—	—	—	—	—	—	—	HAVE	—	—
W	(9)	—	—	—	—	—	—	—	—	—	—	—
X	—	—	—	—	—	—	—	—	—	—	—	—
Y	YOU	—	BY	—	—	—	—	—	—	—	—	—
Z	—	—	—	—	—	—	—	—	—	—	—	—

Note that if the search is unsuccessful, the *longest match* has been found. This property is occasionally useful in applications.

In order to compare the speed of this algorithm to the others in this chapter, we can write a short MIX program assuming that the characters are bytes and that the keys are at most five bytes long.

Program T (*Trie search*). This program assumes that all keys are represented in one MIX word, with blank spaces at the right whenever the key has less than

GN 005721

five characters. Since we use the MIX character code, each byte of the search argument is assumed to contain a number less than 30. Links are represented as negative numbers in the 0:2 field of a node word. rI1 ≡ P, rX ≡ unscanned part of K.

```
01  START    LDX   K          1   T1. Initialize.
02           ENT1  ROOT       1   P ← pointer to root of trie.
03  2H       SLAX  1          C   T2. Branch.
04           STA   *+1(2:2)   C   Extract next character, k.
05           ENT2  0,1        C   Q ← P + k.
06           LD1N  0,2(0:2)   C   P ← LINK(Q).
07           J1P   2B         C   T3. Advance. To T2 if P is a link ≠ Λ.
08           LDA   0,2        1   T4. Compare. rA ← KEY(Q).
09           CMPA  K          1
10           JE    SUCCESS    1   Exit successfully if rA = K.
11  FAILURE  EQU   *              Exit if not in the trie. ▌
```

The running time of this program is $8C + 8$ units, where C is the number of characters examined. Since $C \leq 5$, the search will never take more than 48 units of time.

If we now compare the efficiency of this program (using the trie of Table 1) to Program 6.2.2T (using the *optimum* binary search tree of Fig. 13), we can make the following observations:

1. The trie takes much more memory space; we are using 360 words just to represent 31 keys, while the binary search tree uses only 62 words of memory. (However, exercise 4 shows that, with some fiddling around, we can actually fit the trie of Table 1 into only 55 words.)

2. A successful search takes about 26 units of time for both programs. But an unsuccessful search will go faster in the trie, slower in the binary search tree. For this data the search will be unsuccessful more often than it is successful, so the trie is preferable from the standpoint of speed.

3. If we consider the KWIC indexing application of Fig. 15 instead of the 31 commonest English words, the trie loses its advantage because of the nature of the data. For example, a trie requires 12 iterations to distinguish between COMPUTATION and COMPUTATIONS. (In this case it would be better to build the trie so that words are scanned from right to left instead of from left to right.)

The idea of trie memory was first published by Rene de la Briandais [*Proc. Western Joint Computer Conf.* **15** (1959), 295–298]. He pointed out that we can save memory space at the expense of running time if we use a linked list for each node vector, since most of the entries in the vectors tend to be empty. In effect, this idea amounts to replacing the trie of Table 1 by the forest of trees shown in Fig. 31. Searching in such a forest proceeds by finding the root which matches the first character, then finding the son node of that root which matches the second character, etc.

In his article, de la Briandais did not actually stop the tree branching exactly as shown in Table 1 or Fig. 31; instead, he continued to represent each key, character by character, until reaching the end-of-word delimiter. Thus he

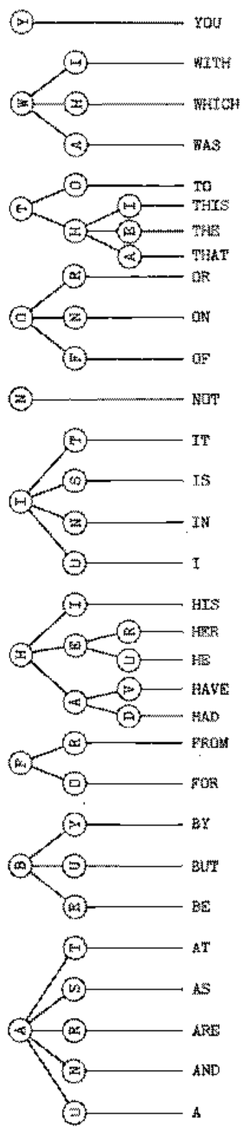
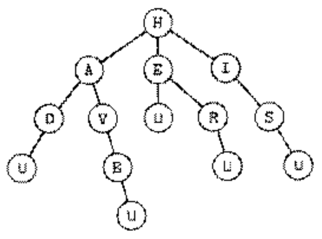


Fig. 31. The trie of Table 1, converted into a "forest."

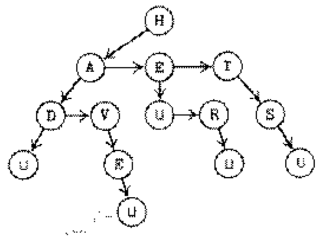
would actually have used



(1)

in place of the "H" tree in Fig. 31. This representation requires more storage, but it makes the processing of variable-length data especially easy. If we use two link fields per character, dynamic insertions and deletions can be handled in a simple manner. Furthermore there are many applications in which the search argument appears in "unpacked form," one character per word, and such a tree makes it unnecessary to pack the data before conducting the search.

If we use the normal way of representing trees as binary trees, (1) becomes the binary tree



(2)

(In the representation of the full forest, Fig. 31, we would also have a pointer leading to the right from H to its neighboring root I.) The search in this binary tree proceeds by comparing a character of the argument to the character in the tree, and following RLINKs until finding a match; then the LLINK is taken and we treat the next character of the argument in the same way.

With such a binary tree, we are more or less doing a search by comparison, with equal-unequal branching instead of less-greater branching. The elementary theory of Section 6.2.1 tells that we must make at least $\log_2 N$ comparisons, on the average, to distinguish between N keys; the average number of tests made when searching a tree like that of Fig. 31 must be at least as many as we make when doing a binary search using the techniques of Section 6.2.

On the other hand, the trie in Table 1 is capable of making an M-way branch all at once; we shall see that the average search time for large N involves only about $\log_M N = \lg N/\lg M$ iterations, if the input data is random. We shall also see that a "pure" trie scheme like that in Algorithm T requires a total of approximately $N/\ln M$ nodes to distinguish between N random inputs; hence the total amount of space is proportional to $MN/\ln M$.

From these considerations it is clear that the trie idea pays off only in the first few levels of the tree. We can get better performance by mixing two strategies, using a trie for the first few characters and then switching to some other technique. For example, E. H. Sussenguth, Jr. [*CACM* **6** (1963), 272–279] has suggested using a character-by-character scheme until we reach part of the tree where only, say, six or less keys of the file are possible, and then we can sequentially run through the short list of remaining keys. We shall see that this mixed strategy decreases the number of trie nodes by roughly a factor of six, without substantially changing the running time.

An application to English. Many variations on the basic trie and character search strategies suggest themselves. In order to get a feeling for some of these possibilities, let us consider a hypothetical large-scale application: Suppose that we want to store a fairly complete dictionary of the English language in the memory of our computer. For this purpose we will, of course, need a reasonably large internal memory, say 50,000 words. Our goal is to find a compact way to represent the dictionary, yet to keep the searching reasonably fast.

Such a project is obviously no small task; it may be expected to require a good knowledge of the contents of the dictionary as well as considerable programming ingenuity. For the moment, let us try to put ourselves in the position of someone embarking on such a major project.

A typical college dictionary contains over 100,000 words; this is somewhat larger than contemplated here, but a glance through such a dictionary will give us some idea of what to expect. If we try to apply the trie memory approach, we soon notice that important simplifications can be made. For example, suppose that we discount proper names and abbreviations. Then if the first letter of a word is b, the second letter will never be any of the characters b, c, d, f, g, j, k, m, n, p, q, s, t, v, w, x, or z (*except* for the word "bdellium," which we might choose to leave out of our dictionary!). In fact, the same 17 possibilities are excluded as second letters in words starting with c, d, f, g, h, j, k, l, m, n, p, q, r, t, v, w, x, z, except for a few words starting with ct, cz, dw, fj, gn, mn, nt, pf, pn, pt, tc, tm, ts, tz, zw and a fair number of words which begin with kn, ps, tw. One way to make use of this fact is to encode the letters (e.g., to have a 26-word table and to perform the equivalent of "`LD1 TABLE,1`"), so that the consonants b, c, d, . . . , z listed above are converted into a special representation greater than the numeric code for the remaining letters a, e, h, i, l, o, r, u, y. In this way, many nodes of a trie can be shortened to a 9-way branch, with another "escape" cell to be used for the rarely occurring exceptional letters. This will save memory space in many parts of a trie for English, not just in the second letter position.

Of the $26^2 = 676$ possible combinations of two letters, only 309 actually occur at the beginning of words in a typical college dictionary; and of these 309 pairs, 88 are the initial letters of 15 or less words. (Typical examples of these 88 rare pairs are aa, ah, aj, ak, ao, aq, ay, az, bd, bh, . . . , xr, yc, yi, yp,

GN 005725

yt, yu, yw, za, zu, zw; most people can't name more than one word from most of these categories.) When one of the rare categories occurs, we can shift from trie memory to some other scheme like a sequential search.

Another way to cut down the storage requirement for a dictionary is to make use of prefixes. For example, if we are looking up a word that begins with re-, pre-, anti-, trans-, dis-, un-, etc., we might wish to detach the prefix and look up the remainder of the word. In this way we can remove many words like reapply, recompute, redecorate, redesign, redeposit, etc.; but we still need to retain words like remainder, requirement, retain, remove, readily, etc., since their meaning is not readily deducible when the prefix "re-" is suppressed. Thus we should first look up the word and then try deleting the prefix only if the first search fails.

The use of suffixes is even more important than the use of prefixes. It would certainly be wasteful to incorporate each noun and verb twice, in both singular and plural form; and there are many other types of suffixes. For example, the following endings may be added to many verb stems, to make a family of related words:

-e	-es	-ing
-ed	-s	-ings
-edly	-able	-ingly
-er	-ible	-ion
-or	-ably	-ions
-ers	-ibly	-ional
-ors	-ability	-ionally
	-abilities	

(Many of these suffixes are themselves composed of suffixes.) If we try to apply these suffixes to the stems

comput-
calculat-
search-
suffix-
translat-
interpret-
confus-

we see that a great many words are formed; this greatly multiplies the capacity of our dictionary. Of course a lot of nonwords are formed also, e.g. "compution"; the stem computat- seems to be necessary as well as comput-. But this causes no harm since such combinations will never appear in the input anyway; and if some author chooses to coin the word "computedly," we will have a ready-made translation of it for him. Note that most people would understand the word "confusability," although it appears in few dictionaries; our dictionary

GN 005726

will give a suitable interpretation. If prefixes and suffixes are correctly handled, our dictionary may even be able to deduce the meaning of "antidisestablishmentarianism," given only the verb stem "establish."

Of course we must be careful that the meaning of each word is properly determined by its stem and suffix; if not, the exceptions should be entered into the dictionary so that they will be found before we attempt to look for a suffix. For example, the analogy between the words "socialism" and "socialist" is not at all the same as between "organism" and "organist"!

These are some of the tricks that can be used to reduce the amount of memory needed. But how shall we represent this hodge-podge of methods compactly in a single system? The answer is to think of the dictionary as a *program* written in a special machine language for a special *interpretive system* (cf. Section 1.4.3); the entries within each node of a trie can be thought of as *instructions.* For example, in Table 1 we have two kinds of "instructions," and Program T uses the sign bit as the "op-code"; a minus sign means branch to another node and advance to the next character for the next instruction, while a plus sign means that the argument is supposed to be compared to a specified key.

We might have the following types of op-codes in the interpretive language for our dictionary application:

- Test n, α, β. "If the next character of the argument has an encoded value $k \leq n$, go to location $\alpha + k$ for the next instruction; otherwise go to location β."
- Compare n, α, β. "Compare the remaining characters of the argument to the n words stored in locations α, $\alpha + 1, \ldots, \alpha + n - 1$. If a match is found in location $\alpha + k$, the search terminates successfully with 'meaning' $\beta + k$, but if no match is found, it terminates unsuccessfully."
- Split α, β. "The word scanned up to this point is a possible prefix or stem. Continue searching by going to location α for the next instruction. If that search is unsuccessful, continue searching by looking up the remaining characters of the argument as if they were a new argument. If this second search is successful, combine the 'meaning' found with the 'meaning' β."

The test operation is essentially the trie search concept, and the compare operation denotes a changeover to sequential searching. The split operation handles both prefixes and suffixes. Several other operations can be envisioned based on further idiosyncrasies of English. It would be possible to save further memory space by eliminating the parameter β from each instruction, since the memory can be arranged so that β is implied by α, n, or the location of the instruction, in each case.

For additional information about dictionary organization, see the interesting articles by Sydney M. Lamb and William H. Jacobsen, Jr., *Mechanical*

GN 005727

Translation **6** (1961), 76–107; Eugene S. Schwartz, *JACM* **10** (1963), 413–439; E. J. Galli and H. Yamada, *IBM Systems J.* **6** (1967), 192–207.

The binary case. Let us now consider the special case $M = 2$, in which we scan the search argument one bit at a time. Two interesting methods have been developed which are especially appropriate for this case.

The first method, which we shall call *digital tree search*, is due to E. G. Coffman and J. Eve [*CACM* **13** (1970), 427–432, 436]. The idea is to store full keys in the nodes just as we did in the tree search algorithm of Section 6.2.2, but to use bits of the argument (instead of results of the comparisons) to govern whether to take the left or right branch at each step. Figure 32 shows the tree constructed by this method when we insert the 31 most common English words in order of decreasing frequency. In order to provide binary data for this illustration, the words have been expressed in MIX character code which was then converted into binary numbers with 5 bits per byte. Thus, the word WHICH is represented as "11010 01000 01001 00011 01000".

To search for this word WHICH in Fig. 32, we compare it first with the word THE at the root of the tree. Since there is no match and since the first bit of WHICH is 1, we move to the right and compare with OF. Since there is no match and since the second bit of WHICH is 1, we move to the right and compare with WITH; and so on.

It is interesting to note the contrast between Fig. 32 and Fig. 12 in Section 6.2.2, since the latter tree was formed in the same way but using comparisons instead of key bits for the branching. If we consider the given frequencies, the digital search tree of Fig. 32 requires an average of 3.42 comparisons per successful search; this is somewhat better than the 4.04 comparisons needed by Fig. 12, although of course the computing time per comparison will probably be different.

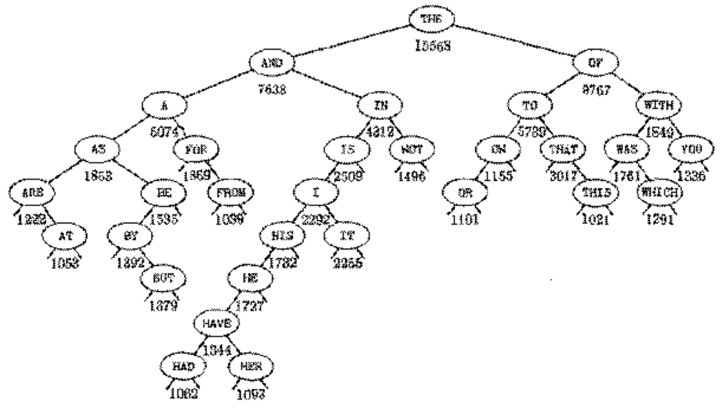


Fig. 32. A digital search tree for the 31 most common English words, inserted in decreasing order of frequency.

GN 005728

Algorithm D (*Digital tree search*). Given a table of records which 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.

This algorithm assumes that the nodes of the tree have KEY, LLINK, and RLINK fields just as in Algorithm 6.2.2T. In fact, the two algorithms are almost identical, as the reader may verify.

D1. [Initialize.] Set P $\leftarrow$ ROOT, and $K1 \leftarrow K$.

D2. [Compare.] If $K =$ KEY(P), the search terminates successfully. Otherwise set b to the leading bit of $K1$, and shift $K1$ left one place (thereby removing that bit and introducing a 0 at the right). If $b = 0$, go to D3, otherwise go to D4.

D3. [Move left.] If LLINK(P) $\neq \Lambda$, set P $\leftarrow$ LLINK(P) and go back to D2. Otherwise go to D5.

D4. [Move right.] If RLINK(P) $\neq \Lambda$, set P $\leftarrow$ RLINK(P) and go back to D2.

D5. [Insert into tree.] Set Q $\Leftarrow$ AVAIL, KEY(Q) $\leftarrow K$, LLINK(Q) $\leftarrow$ RLINK(Q) $\leftarrow \Lambda$. If $b = 0$ set LLINK(P) $\leftarrow$ Q, otherwise set RLINK(P) $\leftarrow$ Q. ▌

Although the tree search of Algorithm 6.2.2T is inherently binary, it is not difficult to see that the present algorithm could be extended to an M-ary digital search for any $M \geq 2$ (see exercise 13).

Donald R. Morrison [*JACM* **15** (1968), 514–534] has discovered a very pretty way to form N-node search trees based on the binary representation of keys, *without* storing keys in the nodes. His method, called "Patricia" (Practical Algorithm To Retrieve Information Coded In Alphanumeric), is especially suitable for dealing with extremely long, variable-length keys such as titles or phrases stored within a large bulk file. A closely related algorithm was published at almost exactly the same time in Germany by G. Gwehenberger, *Elektronische Rechenanlagen* **10** (1968), 223–226.

Patricia's basic idea is to build a binary trie, but to avoid one-way branching by including in each node the number of bits to skip over before making the next test. There are several ways to exploit this idea; perhaps the simplest to explain is illustrated in Fig. 33. We have a TEXT array of bits, which is usually quite long; it may be stored as an external direct-access file, since each search accesses TEXT only once. Each key to be stored in our table is specified by a starting place in the text, and it can be imagined to go from this starting place all the way to the end of the text. (Patricia does not search for strict equality between key and argument, rather it will determine whether or not there exists a key *beginning* with the argument.)

The situation depicted in Fig. 33 involves seven keys, one starting at each word, namely "THIS IS THE HOUSE THAT JACK BUILT." and "IS THE HOUSE THAT JACK BUILT." and . . . and "BUILT.". There is one important restriction, namely that *no one key may be a prefix of another*; this restriction can be met if

```
       1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34
TEXT:  T H I S   I S   T H E   H O U S E   T H A T   J A C K   B U I L T .
```

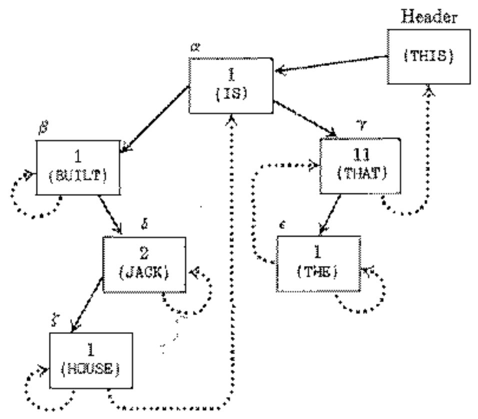


Fig. 33. An example of Patricia's tree and TEXT.

we end the text with a unique end-of-text code (in this case ".") that appears nowhere else. The same restriction was implicit in the trie scheme of Algorithm T, where "␣" was the termination code.

The tree which Patricia uses for searching should be contained in random-access memory, or it should be arranged on pages as suggested in Section 6.2.4. It consists of a header and $N - 1$ nodes, where the nodes contain several fields:

KEY, a pointer to the text. This field must be at least $\log_2 C$ bits long, if the text contains C characters. In Fig. 33 the words shown within each node would really be represented by pointers to the text, e.g. instead of "(JACK)" the node contains the number 24 (the starting place of "JACK BUILT.").

LLINK and RLINK, pointers within the tree. These fields must be at least $\log_2 N$ bits long.

LTAG and RTAG, one-bit fields which tell whether or not LLINK and RLINK, respectively, are pointers to sons or to ancestors of the node. The dotted lines in Fig. 33 correspond to pointers whose TAG bit is 1.

SKIP, a number which tells how many bits to skip when searching, as explained below. This field should be large enough to hold the largest number k such that identical k-bit substrings occur in two different keys; in practice, we may usually assume that k isn't too large, and an error indication can be given if the size of the SKIP field is exceeded. The SKIP fields are shown as numbers within each node of Fig. 33.

The header contains only KEY, LLINK, and LTAG fields.

A search in Patricia's tree is carried out as follows: Suppose we are looking up the word THAT (bit pattern 10111 01000 00001 10111). We start by looking at the SKIP field of the root node, which tells us to examine the first bit of the argument. It is 1, so we move to the right. The SKIP field in the next node tells us to look at the $1 + 11 = 12$th bit of the argument. It is 0, so we move to the left. The SKIP field of the next node tells us to look at the $(12 + 1)$st bit, which is 0; now we find LTAG $= 1$, so we go to node γ which refers us to the TEXT. The search path we have taken would occur for any argument whose bit pattern is 1xxxx xxxxx x00 . . . , and we must check to see if it matches the unique key which begins with that pattern.

Suppose, on the other hand, that we are looking for any or all keys starting with TH. The search process begins as above, but it eventually tries to look at the (nonexistent) 12th bit of the 10-bit argument. At this point we compare the argument to the TEXT at the point specified in the current node (in this case node γ); if it does not match, the argument is not the beginning of any key, but if it does match, the argument is the beginning of every key represented by dotted links in node γ and its descendants.

This process can be spelled out more precisely as follows.

Algorithm P (*Patricia*). Given a TEXT array and a tree with KEY, LLINK, RLINK, LTAG, RTAG, and SKIP fields as described above, this algorithm determines whether or not there is a key in the TEXT which begins with a specified argument K. (If r such keys exist, for $r \geq 1$, it is subsequently possible to locate them all in $O(r)$ steps; see exercise 14.) We assume that at least one key is present.

P1. [Initialize.] Set P $\leftarrow$ HEAD and $j \leftarrow 0$. (Variable P is a pointer which will move down the tree, and j is a counter which will designate bit positions of the argument.) Set $n \leftarrow$ number of bits in K.

P2. [Move left.] Set Q $\leftarrow$ P and P $\leftarrow$ LLINK(Q). If LTAG(Q) $= 1$, go to P6.

P3. [Skip bits.] (At this point we know that if the first j bits of K match any key whatsoever, they match the key which starts at KEY(P).) Set $j \leftarrow j +$ SKIP(P). If $j > n$, go to P6.

P4. [Test bit.] (At this point we know that if the first $j - 1$ bits of K match any key, they match the key starting at KEY(P).) If the jth bit of K is 0, go to P2, otherwise go to P5.

P5. [Move right.] Set Q $\leftarrow$ P and P $\leftarrow$ RLINK(Q). If RTAG(Q) $= 0$, go to P3.

P6. [Compare.] (At this point we know that if K matches any key, it matches the key starting at KEY(P).) Compare K to the key starting at position KEY(P) in the TEXT array. If they are equal (up to n bits, the length of K), the algorithm terminates successfully; if unequal, it terminates unsuccessfully. ▌

Exercise 15 shows how Patricia's tree can be built in the first place. We can also add to the text and insert new keys, provided that the new text material always ends with a unique delimiter (e.g., an end-of-text symbol followed by a serial number).

Patricia is a little tricky, and she requires careful scrutiny before all of her beauties are revealed.

Analyses of the algorithms. We shall conclude this section by making a mathematical study of tries, digital search trees, and Patricia. A summary of the main consequences of these analyses appears at the very end.

Let us consider first the case of binary tries, i.e., tries with $M = 2$. Figure 34 shows the binary trie that is formed when the sixteen keys from the sorting examples of Chapter 5 are treated as 10-bit binary numbers. (The keys are shown in octal notation, so that for example *1144* represents the 10-bit number $(1001100100)_2$.) As in Algorithm T, we use the trie to store information about the leading bits of the keys until we get to the first point where the key is uniquely identified; then the key is recorded in full.

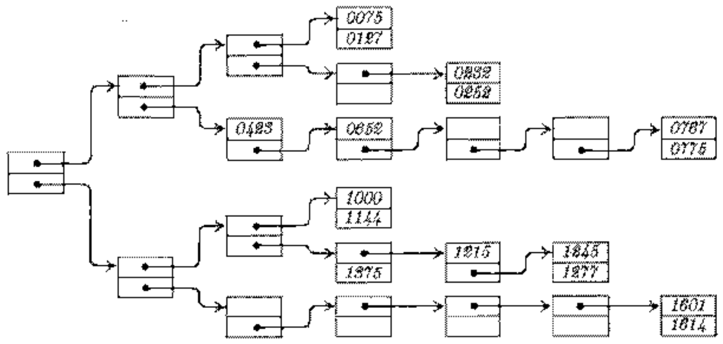


Fig. 34. Example of a random binary trie.

If Fig. 34 is compared to Table 5.2.2–3, an amazing relationship between trie memory and radix-exchange sorting is revealed. (Then again, perhaps this relationship is obvious.) The 22 nodes of Fig. 34 correspond precisely to the 22 partitioning stages in Table 5.2.2–3, with the pth node in preorder corresponding to Stage p. The number of bit inspections in a partitioning stage is equal to the number of keys within the corresponding node and its subtries; consequently we may state the following result.

Theorem T. *If N distinct binary numbers are put into a binary trie as described above, then* (i) *the number of nodes of the trie is equal to the number of partitioning stages required if these numbers are sorted by radix-exchange; and* (ii) *the average number of bit inspections required to retrieve a key by means of Algorithm T is $1/N$ times the number of bit inspections required by the radix-exchange sort.* ▌

GN 005732

Because of this theorem, we can make use of all the mathematical machinery that was developed for radix exchange in Section 5.2.2. For example, if we assume that our keys are infinite-precision random uniformly distributed real numbers between 0 and 1, the number of bit inspections needed for retrieval will be $\log_2 N + \gamma/(\ln 2) + 1/2 + f(N) + O(N^{-1})$, and the number of trie nodes will be $N/(\ln 2) + Ng(N) + O(1)$. Here $f(N)$ and $g(N)$ are complicated functions which may be neglected since their value is always less than 10^{-6} (see exercises 5.2.2–38, 48).

Of course there is still more work to be done, since we need to generalize from binary tries to M-ary tries. We shall describe only the starting point of the investigations here, leaving the instructive details as exercises.

Let A_N be the average number of nodes in a random M-ary search trie that contains N keys. Then $A_0 = A_1 = 0$, and for $N \geq 2$ we have

$$A_N = 1 + \sum_{k_1+\cdots+k_M=N} \left(\frac{N!}{k_1!\ldots k_M!}M^{-N}\right)(A_{k_1} + \cdots + A_{k_M}), \qquad (3)$$

since $N!M^{-N}/k_1!\ldots k_M!$ is the probability that k_1 of the keys are in the first subtrie, ..., k_M in the Mth. This equation can be rewritten

$$\begin{aligned} A_N &= 1 + M^{1-N} \sum_{k_1+\cdots+k_M=N} \left(\frac{N!}{k_1!\ldots k_M!}\right) A_{k_1} \\ &= 1 + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} A_k, \qquad \text{for} \qquad N \geq 2, \end{aligned} \qquad (4)$$

by using symmetry and then summing over $k_2, \ldots, k_M$. Similarly, if C_N denotes the average total number of bit inspections needed to look up all N keys in the trie, we find $C_0 = C_1 = 0$ and

$$C_N = N + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} C_k \qquad \text{for} \qquad N \geq 2. \qquad (5)$$

Exercise 17 shows how to deal with general recurrences of this type, and exercises 18–25 work out the corresponding theory of random tries. [The analysis of A_N was first approached from another point of view by L. R. Johnson and M. H. McAndrew, *IBM J. Res. and Devel.* **8** (1964), 189–193, in connection with an equivalent hardware-oriented sorting algorithm.]

If we now turn to a study of digital search trees, we find that the formulas are similar, yet different enough that it is not easy to see how to deduce the asymptotic behavior. For example, if $\overline{C}_N$ denotes the average total number of bit inspections made when looking up all N keys in an M-ary digital search tree, it is not difficult to deduce as above that $\overline{C}_0 = \overline{C}_1 = 0$, and

$$\overline{C}_{N+1} = N + M^{1-N} \sum_k \binom{N}{k} (M-1)^{N-k} \overline{C}_k \qquad \text{for} \qquad N \geq 0. \qquad (6)$$

This is almost identical to Eq. (5); but the appearance of $N+1$ instead of N on the left-hand side of this equation is enough to change the entire character of the recurrence, and so the methods we have used to study (5) are wiped out.

Let's consider the binary case first. Figure 35 shows the digital search tree corresponding to the sixteen example keys of Fig. 34, when they have been inserted in the order used in the examples of Chapter 5. If we want to determine the average number of bit inspections made in a random successful search, this is just the internal path length of the tree divided by N, since we need l bit inspections to find a node on level l. Note, however, that the average number of bit inspections made in a random *unsuccessful* search is *not* simply related to the external path length of the tree, since unsuccessful searches are more likely to occur at external nodes near the root; thus, the probability of reaching the left sub-branch of node *0075* in Fig. 35 is $\frac{1}{8}$ (assuming infinitely precise keys), and the left sub-branch of node *0232* will be encountered with probability only $\frac{1}{32}$. (For this reason, digital search trees tend to stay better balanced than the binary search trees of Algorithm 6.2.2T, when the keys are uniformly distributed.)

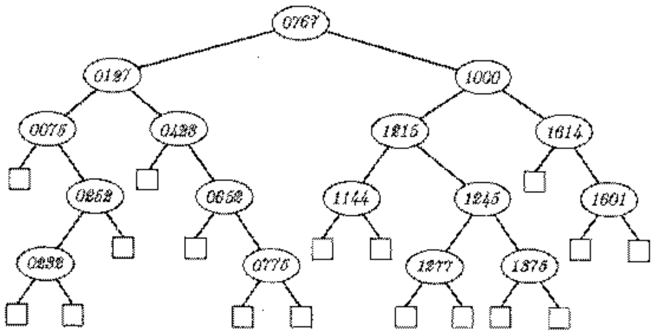


Fig. 35. A random digital search tree constructed by Algorithm D.

We can use a generating function to describe the pertinent characteristics of a digital search tree. If there are a_l internal nodes on level l, consider the generating function $a(z) = \sum a_l z^l$; for example, the generating function corresponding to Fig. 35 is $a(z) = 1 + 2z + 4z^2 + 5z^3 + 4z^4$. If there are b_l external nodes on level l, and if $b(z) = \sum b_l z^l$, we have

$$b(z) = 1 + (2z - 1)a(z) \qquad (7)$$

by exercise 6.2.1–25. For example, $1 + (2z-1)(1 + 2z + 4z^2 + 5z^3 + 4z^4) = 3z^3 + 6z^4 + 8z^5$. The average number of bit inspections made in a random successful search is $a'(1)/a(1)$, since $a'(1)$ is the internal path length of the tree and $a(1)$ is the number of internal nodes. The average number of bit inspections made in a random *unsuccessful* search is $\sum lb_l 2^{-l} = \frac{1}{2}b'(\frac{1}{2}) = a(\frac{1}{2})$, since we end up at a given external node on level l with probability 2^{-l}. The number of comparisons is the same as the number of bit inspections, plus one in a successful search. For example, in Fig. 35, a successful search will take $2\frac{9}{16}$ bit inspections and $3\frac{9}{16}$ comparisons, on the average; an unsuccessful search will take $3\frac{7}{8}$ of each.

Now let $g_N(z)$ be the "average" $a(z)$ for trees with N nodes; in other words, $g_N(z)$ is the sum $\sum p_T a_T(z)$ over all binary digital search trees T with N internal nodes, where $a_T(z)$ is the generating function for the internal nodes of T and p_T is the probability that T occurs when N random numbers are inserted using Algorithm D. Then the average number of bit inspections will be $g'_N(1)/N$ in a successful search, $g_N(\frac{1}{2})$ in an unsuccessful search.

We can compute $g_N(z)$ by mimicking the tree construction process, as follows. If $a(z)$ is the generating function for a tree of N nodes, we can form $N+1$ trees from it by making the next insertion into any one of the external node positions. The insertion goes into a given external node on level l with probability 2^{-l}; hence the sum of the generating functions for the $N+1$ new trees, multiplied by the probability of occurrence, is $a(z) + b(\frac{1}{2}z) = a(z) + 1 + (z-1)a(\frac{1}{2}z)$. Averaging over all trees for N nodes, it follows that

$$g_{N+1}(z) = g_N(z) + 1 + (z-1)g_N(\tfrac{1}{2}z); \qquad g_0(z) = 0. \tag{8}$$

The corresponding generating function for external nodes, $h_N(z) = 1 + (2z-1)g_N(z)$, is somewhat easier to work with, because (8) is equivalent to the formula

$$h_{N+1}(z) = h_N(z) + (2z-1)h_N(\tfrac{1}{2}z); \qquad h_0(z) = 1. \tag{9}$$

Applying this rule repeatedly, we find that

$$\begin{aligned} h_{N+1}(z) &= h_{N-1}(z) + 2(2z-1)h_{N-1}(\tfrac{1}{2}z) + (2z-1)(z-1)h_{N-1}(\tfrac{1}{4}z) \\ &= h_{N-2}(z) + 3(2z-1)h_{N-2}(\tfrac{1}{2}z) + 3(2z-1)(z-1)h_{N-2}(\tfrac{1}{4}z) \\ &\quad + (2z-1)(z-1)(\tfrac{1}{2}z-1)h_{N-2}(\tfrac{1}{8}z) \end{aligned}$$

and so on, so that eventually we have

$$h_N(z) = \sum_k \binom{N}{k} \prod_{0\le j<k} (2^{1-j}z - 1); \tag{10}$$

$$g_N(z) = \sum_{k\ge 0} \binom{N}{k+1} \prod_{0\le j<k} (2^{-j}z - 1). \tag{11}$$

For example, $g_4(z) = 4 + 6(z-1) + 4(z-1)(\frac{1}{2}z-1) + (z-1)(\frac{1}{2}z-1)(\frac{1}{4}z-1)$. These formulas make it possible to express the quantities we are looking for as sums of products:

$$\overline{C}_N = g'_N(1) = \sum_{k\geq 0} \binom{N}{k+2} \prod_{1\leq j\leq k} (2^{-j}-1); \tag{12}$$

$$g_N(\tfrac{1}{2}) = \sum_{k\geq 0} \binom{N}{k+1} \prod_{1\leq j\leq k} (2^{-j}-1) = \overline{C}_{N+1} - \overline{C}_N. \tag{13}$$

It is not at all obvious that this formula for $\overline{C}_N$ satisfies (6)!

Unfortunately, these expressions are not suitable for calculation or for finding an asymptotic expansion, since $2^{-j}-1$ is negative; we get large terms and a lot of cancellation. A more useful formula for $\overline{C}_N$ can be obtained by applying the partition identities of exercise 5.1.1–16. We have

$$\begin{aligned}\overline{C}_N &= \Bigl(\prod_{j\geq 1}(1-2^{-j})\Bigr) \sum_{k\geq 0} \binom{N}{k+2} (-1)^k \prod_{l\geq 0} (1-2^{-l-k-1})^{-1} \\ &= \Bigl(\prod_{j\geq 1}(1-2^{-j})\Bigr) \sum_{k\geq 0} \binom{N}{k+2} (-1)^k \sum_{m\geq 0} (2^{-k-1})^m \prod_{1\leq r\leq m} (1-2^{-r})^{-1} \\ &= \sum_{m\geq 0} 2^m \Bigl(\sum_k \binom{N}{k} (-2^{-m})^k - 1 + 2^{-m}N\Bigr) \prod_{j\geq 0} (1-2^{-j-m-1}) \\ &= \sum_{m\geq 0} 2^m((1-2^{-m})^N - 1 + 2^{-m}N) \sum_{n\geq 0} (-2^{-m-1})^n 2^{-n(n-1)/2} \\ &\qquad\qquad \times \prod_{1\leq r\leq n} (1-2^{-r})^{-1}. \end{aligned} \tag{14}$$

This may not seem at first glance to be an improvement over Eq. (12), but it has the great advantage that it converges rapidly for each fixed n. A precisely analogous situation occurred for the trie case in Eq. 5.2.2–38, 39; in fact, if we consider only the $n=0$ terms of (14), we have exactly $N-1$ plus the number of bit inspections in a binary trie. We can now proceed to get the asymptotic value in essentially the same way as before; see exercise 27. [The above derivation is largely based on an approach suggested by A. J. Konheim and D. J. Newman, *Discrete Mathematics* 4 (1973), 57–63.]

Finally let us take a mathematical look at Patricia. In her case the binary tree is like the corresponding binary trie on the same keys, but squashed together (because the SKIP fields eliminate 1-way branching), so that there are $N-1$ internal nodes and N external nodes. Figure 36 shows the Patrician tree corresponding to the sixteen keys in the trie of Fig. 34. The number shown in each branch node is the amount of SKIP; the keys are indicated with the external nodes, although the external node is not explicitly present (there is actually a tagged link to an internal node which references the TEXT, in place

GN 005736

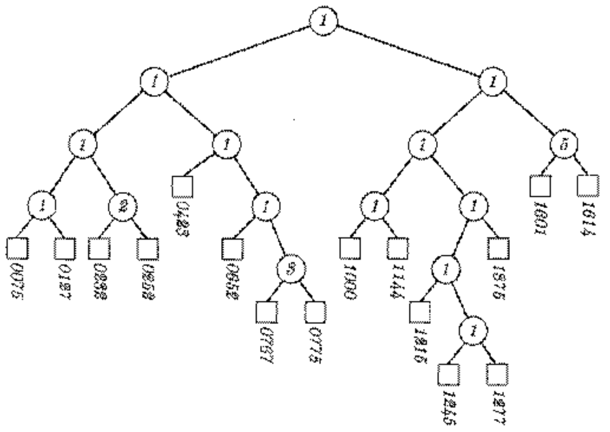


Fig. 36. Patricia constructs this tree instead of Fig. 34.

of each external node.) For the purposes of analysis, we may assume that external nodes exist as shown.

Since successful searches with Patricia end at external nodes, the average number of bit inspections made in a random successful search will be the external path length, divided by N. If we form the generating function $b(z)$ for external nodes as above, this will be $b'(1)/b(1)$. An *unsuccessful* search with Patricia also ends at an external node, but weighted with probability 2^{-l} for external nodes on level l, so the average number of bit inspections is $\frac{1}{2}b'(\frac{1}{2})$. For example, in Fig. 36 we have $b(z) = 3z^3 + 8z^4 + 3z^5 + 2z^6$; there are $4\frac{1}{4}$ bit inspections per successful search and $3\frac{25}{32}$ per unsuccessful search, on the average.

Let $h_N(z)$ be the "average" $b(z)$ for a Patrician tree constructed with N external nodes, using uniformly distributed keys. The recurrence relation

$$h_n(z) = 2^{1-n} \sum_k \binom{n}{k} h_k(z)\big(z + \delta_{kn}(1-z)\big), \quad h_0(z) = 0, \quad h_1(z) = 1 \tag{15}$$

appears to have no simple solution. But fortunately, there is a simple recurrence for the average external path length $h_n'(1)$, since

$$\begin{aligned} h_n'(1) &= 2^{1-n} \sum_k \binom{n}{k} h_k'(1) + 2^{1-n} \sum_k \binom{n}{k} k(1-\delta_{kn}) \\ &= n - 2^{1-n}n + 2^{1-n} \sum_k \binom{n}{k} h_k'(1). \end{aligned} \tag{16}$$

Since this has the form of (6), we can use the methods already developed to solve for $h_n'(1)$, which turns out to be exactly n less than the corresponding number of bit inspections in a random binary trie. Thus, the SKIP fields save us

GN 005737

about one bit inspection per successful search, on random data. (See exercise 31.) The redundancy of typical real data will lead to greater savings.

When we try to find the average number of bit inspections for a random *un*successful search by Patricia, we obtain the recurrence

$$a_n = 1 + \frac{1}{2^n - 2} \sum_{k<n} \binom{n}{k} a_k, \qquad \text{for} \qquad n \geq 2; \qquad a_0 = a_1 = 0. \qquad (17)$$

Here $a_n = \frac{1}{2}h'_n(\frac{1}{2})$. This does *not* have the form of any recurrences we have studied, nor is it easily transformed into such a recurrence. In fact, it turns out that the solution involves the Bernoulli numbers:

$$\frac{na_{n-1}}{2} - n + 2 = \sum_{2\leq k<n} \binom{n}{k} \frac{B_k}{2^{k-1} - 1}. \qquad (18)$$

This formula is probably the hardest asymptotic nut we have yet had to crack; the solution in exercise 34 is an instructive review of many things we have done before, with some slightly different twists.

Summary of the analyses. As a result of all the complicated mathematics in this section, the following facts are perhaps the most noteworthy:

(a) The number of nodes needed to store N random keys in an M-ary trie, with the trie branching terminated for subfiles of $\leq s$ keys, is approximately $N/(s \ln M)$. (This approximation is valid for large N, small s, and small M.) Since a trie node involves M link fields, we will need only about $N/\ln M$ link fields if we choose $s = M$.

(b) The number of digits or characters examined during a random search is approximately $\log_M N$ for all methods considered. When $M = 2$, the various analyses give us the following more accurate approximations to the number of bit inspections:

	Successful	Unsuccessful
Trie search	$\lg N + 1.33275$	$\lg N - 0.10995$
Digital tree search	$\lg N - 1.71665$	$\lg N - 0.27395$
Patricia	$\lg N + 0.33275$	$\lg N - 0.31875$

(These approximations can all be expressed in terms of fundamental mathematical constants, e.g. 0.31875 stands for $(\ln \pi - \gamma)/(\ln 2) - 1/2$.)

(c) "Random" data here means that the M-ary digits are uniformly distributed, as if the keys were real numbers between 0 and 1 expressed in M-ary notation. Digital search methods are insensitive to the order in which keys are entered into the file (except for Algorithm D, which is only slightly sensitive to the order); but they are very sensitive to the distribution of digits. For example, if 0 bits are much more common than 1 bits, the trees will become much more skewed than they would be for random data as considered in the analyses cited above. (Exercise 5.2.2–53 works out one example of what happens when the data is biased in this way.)

EXERCISES

1. [*00*] If a tree has leaves, what does a trie have?

2. [*20*] Design an algorithm for the insertion of a new key into an M-ary trie, using the conventions of Algorithm T.

3. [*21*] Design an algorithm for the deletion of a key from an M-ary trie, using the conventions of Algorithm T.

► 4. [*21*] Most of the 360 entries in Table 1 are blank (null links). But we can compress the table into only 55 entries, by overlapping nonblank entries with blank ones, as follows:

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
Entry	A	—	THAT	(26)	—	WAS	THE	(15)	(2)	I	THIS	HIS	WHICH	WITH	HE	AND	TO	OF	BE	ARE	(1)	(14)	AS	AT	IN	(21)	ON	(3)

Position	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55
Entry	(10)	HAD	OR	IS	IT	HER	NOT	(12)	FOR	BUT	—	FROM	—	BY	(1)	—	—	(5)	—	YOU	—	—	HAVE	—	—	—	—

(Nodes (1), (2), . . . , (12) of Table 1 begin, respectively, at positions 20, 1, 14, 21, 3, 10, 12, 1, 5, 26, 15, 2 within this compressed table.)

Show that if the compressed table is substituted for Table 1, Program T will still work, but not quite as fast.

► 5. [*M26*] (Y. N. Patt.) The trees of Fig. 31 have their letters arranged in alphabetic order within each family. This order is not necessary, and if we rearrange the order of nodes within the families before constructing binary tree representations such as (2) we may get a faster search. What rearrangement of Fig. 31 is optimum from this standpoint? (Use the frequency assumptions of Fig. 32, and find the forest which minimizes the successful search time when it has been represented as a binary tree.)

6. [*15*] What digital search tree is obtained if the fifteen 4-bit binary keys 0001, 0010, 0011, . . . , 1111 are inserted in increasing order by Algorithm D? (Start with 0001 at the root and then do fourteen insertions.)

► 7. [*M26*] If the fifteen keys of exercise 6 are inserted in a different order, we may get a different tree. Of all the 15! possible permutations of these keys, which is the *worst*, in the sense that it produces a tree with the greatest internal path length?

8. [*20*] Consider the following changes to Algorithm D, which have the effect of eliminating variable $K1$: Change "$K1$" to "K" in both places in step D2, and delete the operation "$K1 \leftarrow K$" from step D1. Will the resulting algorithm still be valid for searching and insertion?

9. [*21*] Write a MIX program for Algorithm D, and compare it to Program 6.2.2T. You may use binary operations such as SLB (shift left AX binary), JAE (jump if A even), etc.; and you may also use the idea of exercise 8 if it helps.

10. [*23*] Given a file in which all the keys are n-bit binary numbers, and given a search argument $K = b_1b_2 \ldots b_n$, suppose we want to find the maximum value of k such that there is a key in the file beginning with the bit pattern $b_1b_2 \ldots b_k$. How can we

GN 005739

do this efficiently if the file is represented as (a) a binary search tree (cf. Algorithm 6.2.2T); (b) a binary trie (cf. Algorithm T); (c) a binary digital search tree (cf. Algorithm D)?

11. [*21*] Can Algorithm 6.2.2D be used without change to delete a node from a digital search tree?

12. [*25*] After a random element is deleted from a random digital search tree constructed by Algorithm D, is the resulting tree still random? (Cf. exercise 11 and Theorem 6.2.2H.)

13. [*20*] (*M-ary digital searching.*) Explain how Algorithms T and D can be combined into a generalized algorithm that is essentially the same as Algorithm D when $M = 2$. What changes would be made to Table 1, if your algorithm is used for $M = 30$?

▸**14.** [*25*] Design an efficient algorithm that can be performed just after Algorithm P has terminated successfully, to locate *all* places where K appears in the TEXT.

15. [*28*] Design an efficient algorithm that can be used to construct the tree used by Patricia, or to insert new TEXT references into an existing tree. Your insertion algorithm should refer to the TEXT array at most twice.

16. [*22*] Why is it desirable for Patricia to make the restriction that no key is a prefix of another?

17. [*M25*] Find a way to express the solution of the recurrence

$$x_0 = x_1 = 0; \qquad x_n = a_n + m^{1-n} \sum_k \binom{n}{k} (m-1)^{n-k} x_k, \qquad n \geq 2,$$

in terms of binomial transforms, by generalizing the technique of exercise 5.2.2–36.

18. [*M21*] Use the result of exercise 17 to express the solutions to (4) and (5) in terms of functions U_n and V_n analogous to those defined in exercise 5.2.2–38.

19. [*HM23*] Find the asymptotic value of the function

$$K(n, s, m) = \sum_{k \geq 2} \binom{n}{k} \binom{k}{s} \frac{(-1)^k}{m^{k-1} - 1}$$

to $O(1)$ as $n \to \infty$, for fixed $s \geq 0$ and $m > 1$. [The case $s = 0$ has already been solved in exercise 5.2.2–50, and the case $s = 1$, $m = 2$ has been solved in exercise 5.2.2–48.]

▸**20.** [*M30*] Consider M-ary trie memory in which we use a sequential search whenever reaching a subfile of s or less keys. (Algorithm T is the special case $s = 1$.) Apply the results of the preceding exercises to analyze (a) the average number of trie nodes; (b) the average number of digit or character inspections in a successful search; and (c) the average number of comparisons made in a successful search. State your answers as asymptotic formulas as $N \to \infty$, for fixed M and s; the answer for (a) should be correct to within $O(1)$, and the answers for (b) and (c) should be correct to within $O(N^{-1})$. [When $M = 2$, this analysis applies also to the modified radix-exchange sort, in which subfiles of size $\leq s$ are sorted by insertion.]

21. [*M25*] How many of the nodes, in a random M-ary trie containing N keys, have a null pointer in table entry 0? (For example, 9 of the 12 nodes in Table 1 have a null pointer in the "⊔" position. "Random" in this exercise means as usual that the characters of the keys are uniformly distributed between 0 and $M - 1$.)

22. [*M25*] How many trie nodes are on level l of a random M-ary trie containing N keys, for $l = 0, 1, 2, \ldots$?

23. [*M26*] How many digit inspections are made on the average during an *un*successful search in an M-ary trie containing N random keys?

24. [*M30*] Consider an M-ary trie which has been represented as a forest (cf. Fig. 31). Find exact and asymptotic expressions for (a) the average number of nodes in the forest, and (b) the average number of times "P ← RLINK(P)" is performed during a random successful search.

▶ **25.** [*M24*] The mathematical derivations of asymptotic values in this section have been quite difficult, involving complex variable theory, because it is desirable to get more than just the leading term of the asymptotic behavior (and the second term is intrinsically complicated). The purpose of this exercise is to show that elementary methods are good enough to deduce some of the results in weaker form. (a) Prove by induction that the solution to (4) satisfies $A_N \le M(N-1)/(M-1)$ for $N \ge 1$. (b) Let $D_N = C_N - NH_{N-1}/(\ln M)$, where C_N is defined by (5). Prove that $D_N = O(N)$; hence $C_N = N \log_M N + O(N)$. [*Hint:* Use (a) and Theorem 1.2.7A.]

26. [*23*] Determine the value of the infinite product

$$(1-\tfrac{1}{2})(1-\tfrac{1}{4})(1-\tfrac{1}{8})(1-\tfrac{1}{16})\ldots$$

correct to five decimal places, by hand calculation. [*Hint:* Cf. exercise 5.1.1–16.]

27. [*HM31*] What is the asymptotic value of $\overline{C}_N$, as given by (14), to within $O(1)$?

28. [*HM26*] Find the asymptotic average number of digit inspections when searching in a random M-ary digital search tree, for general $M \ge 2$. Consider both successful and unsuccessful search, and give your answer to within $O(N^{-1})$.

29. [*M46*] What is the asymptotic average number of nodes, in an M-ary digital search tree, for which all M links are null? (We might save memory space by eliminating such nodes, cf. exercise 13.)

30. [*M24*] Show that the Patrician generating function $h_n(z)$ defined in (15) can be expressed in the rather horrible form

$$n\sum_{m\ge 1} z^m \left(\sum_{\substack{a_1+\cdots+a_m=n-1\\ a_1,\ldots,a_m\ge 1}} \binom{n-1}{a_1,\ldots,a_m} \frac{1}{(2^{a_1}-1)(2^{a_1+a_2}-1)\ldots(2^{a_1+\cdots+a_m}-1)} \right).$$

[Thus, if there is a simple formula for $h_n(z)$, we will be able to simplify this rather ungainly expression.]

31. [*M21*] Solve the recurrence (16).

32. [*M21*] What is the average value of the sum of all SKIP fields in a random Patrician tree with $N-1$ internal nodes?

33. [*M30*] Prove that (18) is a solution to the recurrence (17). [*Hint:* Consider the generating function $A(z) = \sum_{n\ge 0} a_n z^n/n!$.]

34. [*HM40*] The purpose of this exercise is to find the asymptotic behavior of (18). (a) Prove that

$$\frac{1}{n}\sum_{2\le k<n}\binom{n}{k}\frac{B_k}{2^{k-1}-1}=\sum_{j\ge 1}\left(\frac{1^{n-1}+2^{n-1}+\cdots+(2^j-1)^{n-1}}{2^{j(n-1)}}-\frac{2^j}{n}+\frac{1}{2}\right).$$

(b) Show that the summand in (a) can be approximated by $1/(e^x-1)-1/x+1/2$, where $x=n/2^j$; the resulting sum equals the original sum $+\,O(n^{-1})$. (c) Show that

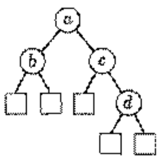
$$\frac{1}{e^x-1}-\frac{1}{x}+\frac{1}{2}=\frac{1}{2\pi i}\int_{-\frac12-i\infty}^{-\frac12+i\infty}\zeta(z)\Gamma(z)x^{-z}\,dz,\qquad \text{for real } x>0.$$

(d) Therefore the sum equals

$$\frac{1}{2\pi i}\int_{-\frac12-i\infty}^{-\frac12+i\infty}\frac{\zeta(z)\Gamma(z)n^{-z}}{2^{-z}-1}\,dz+O(n^{-1});$$

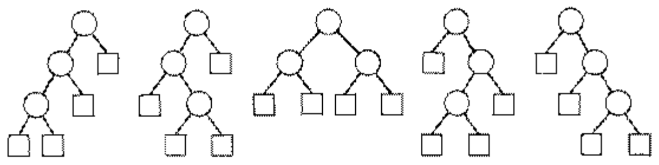
evaluate this integral.

▸**35.** [*M20*] What is the probability that Patricia's tree on five keys will be



with the SKIP fields a, b, c, d as shown? (Assume that the keys have independent random bits, and give your answer as a function of a, b, c and d.)

36. [*M25*] There are five binary trees with three internal nodes. If we consider how frequently each particular one of these occurs as the search tree in various algorithms, for random data, we find the following different probabilities:



Tree search (Algorithm 6.2.2T)	$\frac16$	$\frac16$	$\frac13$	$\frac16$	$\frac16$
Digital tree search (Algorithm D)	$\frac18$	$\frac18$	$\frac12$	$\frac18$	$\frac18$
Patricia (Algorithm P)	$\frac17$	$\frac17$	$\frac37$	$\frac17$	$\frac17$

(Note that the digital search tree tends to be balanced more often than the others.) In exercise 6.2.2–5 we found that the probability of a tree in the tree search algorithm was $\Pi(1/s(x))$, where the product is over all internal nodes x, and $s(x)$ is the number of internal nodes in the subtree rooted at x. Find similar formulas for the probability of a tree in the case of (a) Algorithm D; (b) Algorithm P.

▸**37.** [*M22*] Consider a binary tree with b_l external nodes on level l. The text observes that the running time for unsuccessful searching in digital search trees is not directly related to the external path length $\sum lb_l$, but instead it is essentially proportional to the *modified external path length* $\sum lb_l 2^{-l}$. Prove or disprove: The smallest modified external path length, over all trees with N external nodes, occurs when all of the external nodes appear on at most two adjacent levels. [Cf. exercise 5.3.1–20.]

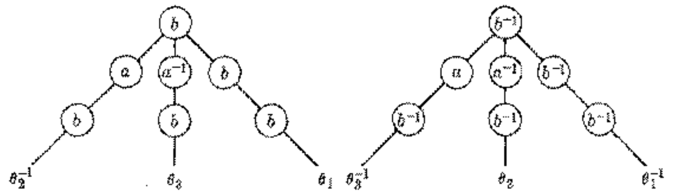
38. [*M40*] Develop an algorithm to find the n-node tree having the minimum value of $\alpha \cdot$ (internal path length) $+ \beta \cdot$ (modified external path length), given α and β. (Cf. exercise 37.)

39. [*M47*] Develop an algorithm to find optimum digital search trees, analogous to the optimum binary search trees considered in Section 6.2.2.

40. [*25*] Let $a_0a_1a_2\ldots$ be a periodic binary sequence with $a_{N+k} = a_k$ for all $k \geq 0$. Show that there is a way to represent any fixed sequence of this type in $O(N)$ memory locations, so that the following operation can be done in only $O(n)$ steps: Given any binary pattern $b_0b_1\ldots b_{n-1}$, determine how often the pattern occurs in the period (i.e., find how many values of p exist with $0 \leq p < N$ and $b_k = a_{p+k}$ for $0 \leq k < n$). (The length n of the pattern is variable as well as the pattern itself. Assume that each memory location is big enough to hold arbitrary integers between 0 and N).

41. [*HM28*] This is an application to group theory. Let G be the free group on the letters $\{a_1, \ldots, a_n\}$, i.e., the set of all strings $\alpha = b_1 \ldots b_r$ where each b_i is one of the a_j or a_j^{-1} and no adjacent pair $a_ja_j^{-1}$ or $a_j^{-1}a_j$ occurs. The inverse of α is $b_r^{-1}\ldots b_1^{-1}$, and we multiply two such strings by concatenating them and cancelling adjacent inverse pairs. Let H be the subgroup of G generated by the strings $\{\beta_1, \ldots, \beta_p\}$, i.e., the set of all elements of G which can be written as products of the β's and their inverses. It can be shown (see Marshall Hall, *The Theory of Groups* (New York: Macmillan, 1959), Chapter 7) that we may always find generators $\theta_1, \ldots, \theta_m$ of H, with $m \leq p$, satisfying the "Nielsen property," which states that the middle character of θ_i (or at least one of the two central characters of θ_i if it has even length) is never cancelled in the expressions $\theta_i\theta_j^e$ or $\theta_j^e\theta_i$, $e = \pm 1$, unless $j = i$ and $e = -1$. This property implies that there is a simple algorithm for testing whether an arbitrary element of G is in H: Record the $2m$ keys $\theta_1, \ldots, \theta_m, \theta_1^{-1}, \ldots, \theta_m^{-1}$ in a character-oriented search tree, using the $2n$ letters $a_1, \ldots, a_n, a_1^{-1}, \ldots, a_n^{-1}$. Let $\alpha = b_1 \ldots b_r$ be a given element of G; if $r = 0$, α is obviously in H. Otherwise look up α, finding the longest prefix $b_1 \ldots b_k$ that matches a key. If there is more than one key beginning with $b_1 \ldots b_k$, α is not in H; otherwise let the unique such key be $b_1 \ldots b_kc_1 \ldots c_l = \theta_i^e$, and replace α by $\theta_i^{-e}\alpha = c_l^{-1}\ldots c_1^{-1}b_{k+1}\ldots b_r$. If this new value of α is longer than the old (i.e., if $l > k$), α is not in H; otherwise repeat the process on the new value of α. The Nielsen property implies that this algorithm will always terminate. If α is eventually reduced to the null string, we can reconstruct the representation of the original α as a product of θ's.

For example, let $\{\theta_1, \theta_2, \theta_3\} = \{bbb, b^{-1}a^{-1}b^{-1}, ba^{-1}b\}$ and $\alpha = bbabaab$. The forest



can be used with the above algorithm to deduce that $\alpha = \theta_1\theta_3^{-1}\theta_1\theta_3^{-1}\theta_2^{-1}$. Implement this algorithm, given the θ's as input to your program.

42. [*23*] (*Front and rear compression.*) When a set of binary keys is being used as an index, to partition a larger file, we need not store the full keys. For example, if the sixteen keys of Fig. 34 are used, they can be truncated at the right, as soon as enough digits have been given to uniquely identify them: 0000, 0001, 00100, 00101, 010, . . . , 1110001. These truncated keys can be used to partition a file into seventeen parts, where for example the fifth part consists of all keys beginning with 0011 or 010, and the last part contains all keys beginning with 111001, 11101, or 1111. The truncated keys can be represented more compactly if we suppress all leading digits common to the previous key: 0000, ***1, **100, ****1, *10, . . . , ******1. The bit following a * is always 1, so it may be suppressed. A large file will have many *'s, and we need store only the number of *'s and the values of the following bits. (This compression technique was shown to the author by A. Heller and R. L. Johnsen.)

Show that the total number of bits in the compressed file, excluding *'s and the following 1 bits, is always equal to the number of nodes in the binary trie for the keys.

(Consequently the average total number of such bits in the entire index is about $N/(\ln 2)$, only 1.44 bits per key. Still further compression is possible, since we need only represent the trie structure; cf. Theorem 2.3.1 A.)

6.4 HASHING

So far we have considered search methods based on comparing the given argument K to the keys in the table, or using its digits to govern a branching process. A third possibility is to avoid all this rummaging around by doing some arithmetical calculation on K, computing a function $f(K)$ which is the location of K and the associated data in the table.

For example, let's consider again the set of 31 English words which we have subjected to various search strategies in Section 6.2.2 and 6.3. Table 1 shows a short MIX program which transforms each of the 31 keys into a unique number $f(K)$ between -10 and 30. If we compare this method to the MIX programs for the other methods we have considered (e.g., binary search, optimal tree search, trie memory, digital tree search), we find that it is superior from the standpoint of both space and speed, except that binary search uses slightly less space. In fact, the average time for a successful search, using the program of Table 1 with the frequency data of Fig. 12, is only about $17.8u$, and only 41 table locations are needed to store the 31 keys.

Unfortunately it isn't very easy to discover such functions $f(K)$. There are $41^{31} \approx 10^{50}$ possible functions from a 31-element set into a 41-element set, and only $41 \cdot 40 \cdot \cdots \cdot 11 = 41!/10! \approx 10^{43}$ of them will give distinct values for each argument; thus only about one of every 10 million functions will be suitable.

Functions which avoid duplicate values are surprisingly rare, even with a fairly large table. For example, the famous "birthday paradox" asserts that if 23 or more people are present in a room, chances are good that two of them

Table 1

TRANSFORMING A SET OF KEYS INTO UNIQUE ADDRESSES

	Instruction		A	AND	ARE	AS	AT	BE	BUT	BY	FOR	FROM	HAD	HAVE	HE	HER
	LD1N	K(1:1)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	LD2	K(2:2)	−1	−1	−1	−1	−1	−2	−2	−2	−6	−6	−8	−8	−8	−8
	INC1	-8,2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	J1P	*+2	−9	6	10	13	14	−5	14	18	2	5	−15	−15	−11	−11
	INC1	16,2	7	.	.	.	.	16	.	.	.	.	2	2	10	10
	LD2	K(3:3)	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	J2Z	9F	7	6	10	13	14	16	14	18	2	5	2	2	10	10
	INC1	-28,2	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	1
	J1P	9F	.	−18	−13	.	.	.	9	.	−7	−7	−22	−1	.	1
	INC1	11,2	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	LDA	K(4:4)	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	JAZ	9F	.	−3	3	.	.	.	.	.	23	20	−7	35	.	.
	DEC1	-5,2	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	J1N	9F	.	.	.	.	.	.	.	.	.	9	.	15	.	.
	INC1	10	.	.	.	.	.	.	.	.	.	19	.	25	.	.
9H	LDA	K	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	CMPA	TABLE,1	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1
	JNE	FAILURE	7	−3	3	13	14	16	9	18	23	19	−7	25	10	1

will have the same month and day of birth! In other words, if we select a random function which maps 23 keys into a table of size 365, the probability that no two keys map into the same location is only 0.4927 (less than one-half). Skeptics who doubt this result should try to find the birthday mates at the next large parties they attend. [The birthday paradox apparently originated in unpublished work of H. Davenport; cf. W. W. R. Ball, *Math. Recreations and Essays* (1939), 45. See also R. von Mises, *İstanbul Üniversitesi Fen Fakültesi Mecmuasi* 4 (1939), 145–163, and W. Feller, *An Introduction to Probability Theory* (New York: Wiley, 1950), Section 2.3.]

On the other hand, the approach used in Table 1 is fairly flexible [cf. M. Greniewski and W. Turski, *CACM* 6 (1963), 322–323], and for a medium-sized table a suitable function can be found after about a day's work. In fact it is rather amusing to solve a puzzle like this.

Of course this method has a serious flaw, since the contents of the table must be known in advance; adding one more key will probably ruin everything, making it necessary to start over almost from scratch. We can obtain a much more versatile method if we give up the idea of uniqueness, permitting different keys to yield the same value $f(K)$, and using a special method to resolve any ambiguity after $f(K)$ has been computed.

These considerations lead to a popular class of search methods commonly known as *hashing* or *scatter storage* techniques. The verb "to hash" means to chop something up or to make a mess out of it; the idea in hashing is to chop off some aspects of the key and to use this partial information as the basis for searching. We compute a *hash function* $h(K)$ and use this value as the address where the search begins.

The birthday paradox tells us that there will probably be distinct keys $K_i \neq K_j$ which hash to the same value $h(K_i) = h(K_j)$. Such an occurrence is

Contents of rI1 after executing the instruction, given a particular key K

HIS	I	IN	IS	IT	NOT	OF	ON	OR	THAT	THE	THIS	TO	WAS	WHICH	WITH	YOU
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−8	−9	−9	−9	−9	−15	−16	−16	−16	−23	−23	−23	−23	−26	−26	−26	−28
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
−7	−17	−2	5	6	−7	−18	−9	−5	−23	−23	−23	−15	−33	−26	−25	−20
18	−1	29	.	.	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
18	−1	29	5	6	25	4	22	30	1	1	1	17	−16	−2	0	12
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
12	.	.	.	.	20	.	.	.	−26	−22	−18	.	−22	−21	−5	8
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−14	−6	2	.	11	−1	29	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	−10	.	−2	.	.	−5	11	.
.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	21	.
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8
12	−1	29	5	6	20	4	22	30	−10	−6	−2	17	11	−5	21	8

called a *collision*, and several interesting approaches have been devised to handle the collision problem. In order to use a scatter table, a programmer must make two almost independent decisions: He must choose a hash function $h(K)$, and he must select a method for collision resolution. We shall now consider these two aspects of the problem in turn.

Hash functions. To make things more explicit, let us assume throughout this section that our hash function h takes on at most M different values, with

$$0 \le h(K) < M, \tag{1}$$

for all keys K. The keys in actual files that arise in practice usually have a great deal of redundancy; we must be careful to find a hash function that breaks up clusters of almost identical keys, in order to reduce the number of collisions.

It is theoretically impossible to define a hash function that creates random data from the nonrandom data in actual files. But in practice it is not difficult to produce a pretty good imitation of random data, by using simple arithmetic as we have discussed in Chapter 3. And in fact we can often do even better, by exploiting the nonrandom properties of actual data to construct a hash function that leads to fewer collisions than truly random keys would produce.

Consider, for example, the case of 10-digit keys on a decimal computer. One hash function that suggests itself is to let $M = 1000$, say, and to let $h(K)$ be three digits chosen from somewhere near the middle of the 20-digit product $K \times K$. This would seem to yield a fairly good spread of values between 000 and 999, with low probability of collisions. Experiments with actual data show, in fact, that this "middle square" method isn't bad, provided that the keys do not have a lot of leading or trailing zeros; but it turns out that there are safer and saner ways to proceed, just as we found in Chapter 3 that the middle square method is not an especially good random number generator.

Extensive tests on typical files have shown that two major types of hash functions work quite well. One of these is based on division, and the other is based on multiplication.

The division method is particularly easy; we simply use the remainder modulo M:

$$h(K) = K \bmod M. \tag{2}$$

In this case, some values of M are obviously much better than others. For example, if M is an even number, $h(K)$ will be even when K is even and odd when K is odd, and this will lead to a substantial bias in many files. It would be even worse to let M be a power of the radix of the computer, since $K \bmod M$ would then be simply the least significant digits of K (independent of the other digits). Similarly we can argue that M probably shouldn't be a multiple of 3 either; for if the keys are alphabetic, two keys which differ from each other only by permutation of letters would then differ in numeric value by a multiple of 3. (This occurs because $10^n \bmod 3 = 4^n \bmod 3 = 1$.) In general, we want to avoid values of M which divide $r^k \pm a$, where k and a are small numbers and r is the radix of the alphabetic character set (usually $r = 64$, 256, or 100), since a remainder modulo such a value of M tends to be largely a simple superposition of the key digits. Such considerations suggest that we *choose M to be a prime number* such that $r^k \equiv \pm a$ (modulo M) for small k and a. This choice has been found to be quite satisfactory in virtually all cases.

For example, on the MIX computer we could choose $M = 1009$, computing $h(K)$ by the sequence

```
LDX   K          rX ← K.
ENTA  0          rA ← 0.                (3)
DIV   =1009=     rX ← K mod 1009.
```

The multiplicative hashing scheme is equally easy to do, but it is slightly harder to describe because we must imagine ourselves working with fractions instead of with integers. Let w be the word size of the computer, so that w is usually 10^{10} or 2^{30} for MIX; we can regard an integer A as the fraction A/w if we imagine the radix point to be at the left of the word. The method is to choose some integer constant A relatively prime to w, and to let

$$h(K) = \left\lfloor M\left(\left(\frac{A}{w}K\right) \bmod 1\right)\right\rfloor. \tag{4}$$

In this case we usually let M be a power of 2 on a binary computer, so that $h(K)$ consists of the leading bits of the least significant half of the product AK.

In MIX code, if we let $M = 2^m$ and assume a binary radix, the multiplicative hash function is

```
LDA   K     rA ← K.
MUL   A     rAX ← AK.                       (5)
ENTA  0     rAX ← AK mod w.
SLB   m     Shift rAX m bits to the left.
```

Now $h(K)$ appears in register A. Since MIX has rather slow multiplication and shift instructions, this sequence takes exactly as long to compute as (3); but on many machines multiplication is significantly faster than division.

In a sense this method can be regarded as a generalization of (3), since we could for example take A to be an approximation to $w/1009$; multiplying by the reciprocal of a constant is often faster than dividing by that constant. Note that (5) is almost a "middle square" method, but there is one important difference: We shall see that multiplication by a suitable constant has demonstrably good properties.

One of the nice features of the multiplicative scheme is that no information was lost in (5); we could determine K again, given only the contents of rAX after (5) has finished. The reason is that A is relatively prime to w, so Euclid's algorithm can be used to find a constant A' with $AA' \bmod w = 1$; this implies that $K = (A'(AK \bmod w)) \bmod w$. In other words, if $f(K)$ denotes the contents of register X just before the SLB instruction in (5), then

$$K_1 \neq K_2 \qquad \text{implies} \qquad f(K_1) \neq f(K_2). \tag{6}$$

GN 005748

Of course $f(K)$ takes on values in the range 0 to $w - 1$, so it isn't any good as a hash function, but it can be very useful as a *scrambling function*, namely a function satisfying (6) and tending to randomize the keys. Such a function can be very useful in connection with the tree search algorithms of Section 6.2.2, if the order of keys is unimportant, since it removes the danger of degeneracy when keys enter the tree in increasing order. A scrambling function is also useful in connection with the digital tree search algorithm of Section 6.3, if the bits of the actual keys are biased.

Another feature of the multiplicative hash method is that it makes good use of the nonrandomness found in many files. Actual sets of keys often have a preponderance of arithmetic progressions, where $\{K, K + d, K + 2d, \ldots, K + td\}$ all appear in the file; for example, consider alphabetic names like {PART1, PART2, PART3} or {TYPEA, TYPEB, TYPEC}. The multiplicative hash method converts an arithmetic progression into an approximate arithmetic progression $h(K)$, $h(K + d)$, $h(K + 2d)$, ... of distinct hash values, reducing the number of collisions from what we would expect in a random situation. The division method has this same property.

Figure 37 illustrates this aspect of multiplicative hashing in a particularly interesting case. Suppose that A/w is approximately the golden ratio $\phi^{-1} = (\sqrt{5} - 1)/2 = 0.6180339887$; then the behavior of successive values $h(K)$, $h(K + 1)$, $h(K + 2)$, ... can be studied by considering the behavior of the successive values $h(0)$, $h(1)$, $h(2)$, ... This suggests the following experiment: Starting with the line segment [0, 1], we successively mark off the points $\{\phi^{-1}\}$, $\{2\phi^{-1}\}$, $\{3\phi^{-1}\}$, ..., where $\{x\}$ denotes the fractional part of x (namely $x - \lfloor x \rfloor = x \bmod 1$). As shown in Fig. 37, these points stay very well separated from each other; in fact, each newly added point falls into one of the largest remaining intervals, and divides it in the golden ratio! [This phenomenon was

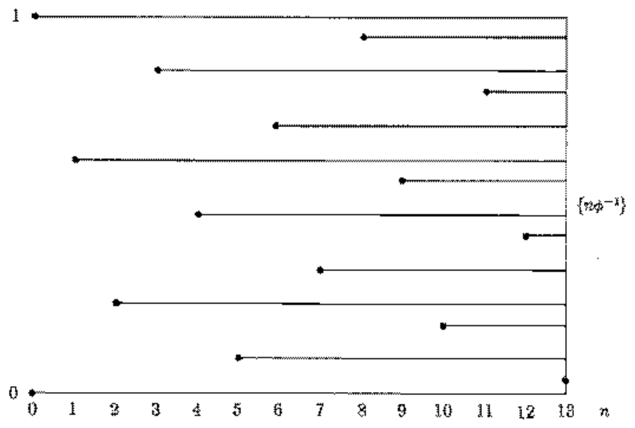


Fig. 37. Fibonacci hashing.

first conjectured by J. Oderfeld and proved by S. Świerczkowski, *Fundamenta Math.* **46** (1958), 187–189. Fibonacci numbers play an important rôle in the proof.]

This remarkable property of the golden ratio is actually just a special case of a very general result, originally conjectured by Hugo Steinhaus and first proved by Vera Turán Sós [*Acta Math. Acad. Sci. Hung.* **8** (1957), 461–471; *Ann. Univ. Sci. Budapest. Eötvös Sect. Math.* **1** (1958), 127–134]:

Theorem S. *Let θ be any irrational number. When the points $\{\theta\}$, $\{2\theta\}, \ldots,$ $\{n\theta\}$ are placed in the line segment $[0, 1]$, the $n+1$ line segments formed have at most three different lengths. Moreover, the next point $\{(n+1)\theta\}$ will fall in one of the largest existing segments.* ▌

Thus, the points $\{\theta\}$, $\{2\theta\}$, $\ldots$, $\{n\theta\}$ are spread out very evenly between 0 and 1. If θ is rational, the same theorem holds if we give a suitable interpretation to the segments of length 0 that appear when n is greater than or equal to the denominator of θ. A proof of Theorem S, together with a detailed analysis of the underlying structure of the situation, appears in exercise 8; it turns out that the segments of a given length are created and destroyed in a first-in-first-out manner. Of course, some θ's are better than others, since for example a value that is near 0 or 1 will start out with many small segments and one large segment. Exercise 9 shows that the two numbers ϕ^{-1} and $\phi^{-2} = 1 - \phi^{-1}$ lead to the "most uniformly distributed" sequences, among all numbers θ between 0 and 1.

The above theory suggests *Fibonacci hashing*, where we choose the constant A to be the nearest integer to $\phi^{-1}w$ that is relatively prime to w. For example if MIX were a decimal computer we would take

$$A = \boxed{+}\boxed{61}\boxed{80}\boxed{33}\boxed{98}\boxed{87}\,. \qquad (7)$$

This multiplier will spread out alphabetic keys like LIST1, LIST2, LIST3 very nicely. But notice what happens when we have an arithmetic series in the fourth character position, as in the keys SUM1␣, SUM2␣, SUM3␣: The effect is as if Theorem S were being used with $\theta = \{100A/w\} = .80339887$ instead of $\theta = .6180339887 = A/w$. The resulting behavior is still all right, in spite of the fact that this value of θ is not quite as good as ϕ^{-1}. On the other hand, if the progression occurs in the second character position, as in A1␣␣␣, A2␣␣␣, A3␣␣␣, the effective θ is .9887, and this is probably too close to 1.

Therefore we might do better with a multiplier like

$$A = \boxed{+}\boxed{61}\boxed{61}\boxed{61}\boxed{61}\boxed{61} \qquad (8)$$

in place of (7); such a multiplier will separate out consecutive sequences of keys differing in *any* character position. Unfortunately this choice suffers from another problem analogous to the difficulty of dividing by $r^k \pm 1$: Keys such as XY and YX will tend to hash to the same location! One way out of this diffi-

GN 005750

culty is to look more closely at the structure underlying Theorem S; for short progressions of keys, only the first few partial quotients of the continued fraction representation of θ are relevant, and small partial quotients correspond to good distribution properties. Therefore we find that the best values of θ lie in the ranges

$$\tfrac{1}{4} < \theta < \tfrac{3}{10}, \qquad \tfrac{1}{3} < \theta < \tfrac{3}{7}, \qquad \tfrac{4}{7} < \theta < \tfrac{2}{3}, \qquad \tfrac{7}{10} < \theta < \tfrac{3}{4}.$$

A value of A can be found so that each of its bytes lies in a good range and is not too close to the values of the other bytes or their complements, e.g.,

$$A = \begin{array}{|c|c|c|c|c|c|}\hline + & 61 & 25 & 42 & 33 & 71 \\ \hline\end{array}\,. \tag{9}$$

Such a multiplier can be recommended. (These ideas about multiplicative hashing are due largely to R. W. Floyd.)

A good hash function should satisfy two requirements:

a) Its computation should be very fast.

b) It should minimize collisions.

Property (a) is somewhat machine-dependent, and property (b) is data-dependent. If the keys were truly random, we could simply extract a few bits from them and use these bits for the hash function; but in practice we nearly always need to have a hash function that depends on all bits of the key in order to satisfy (b).

So far we have considered how to hash one-word keys. Multiword or variable-length keys can be handled by multiple-precision extensions of the above methods, but it is generally adequate to speed things up by combining the individual words together into a single word, then doing a single multiplication or division as above. The combination can be done by addition mod w, or by "exclusive or" on a binary computer; both of these operations have the advantage that they are invertible, i.e., that they depend on all bits of both arguments, and exclusive-or is sometimes preferable because it avoids arithmetic overflow. Note that both of these operations are commutative, so that (X, Y) and (Y, X) will hash to the same address; G. D. Knott has suggested avoiding this problem by doing a cyclic shift just before adding or exclusive-oring.

Many more methods for hashing have been suggested, but none of these has proved to be superior to the simple division and multiplication methods described above. For a survey of some other methods together with detailed statistics on their performance with actual files, see the article by V. Y. Lum, P. S. T. Yuen, and M. Dodd, *CACM* **14** (1971), 228–239.

Of all the other hash methods that have been tried, perhaps the most interesting is a technique based on algebraic coding theory; the idea is analogous to the division method above, but we divide by a polynomial modulo 2 instead of dividing by an integer. (As observed in Section 4.6, this operation is analogous to division, just as addition is analogous to exclusive-or.) For this method, M

should be a power of 2, say $M = 2^m$, and we make use of an mth degree polynomial $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_0$. An n-digit binary key $K = (k_{n-1} \ldots k_1k_0)_2$ can be regarded as the polynomial $K(x) = k_{n-1}x^{n-1} + \cdots + k_1x + k_0$, and we compute the remainder

$$K(x) \bmod P(x) = h_{m-1}x^{m-1} + \cdots + h_1x + h_0$$

using polynomial arithmetic modulo 2; then $h(K) = (h_{m-1} \ldots h_1h_0)_2$. If $P(x)$ is chosen properly, this hash function can be guaranteed to avoid collisions between nearly-equal keys. For example if $n = 15$, $m = 10$, and

$$P(x) = x^{10} + x^8 + x^5 + x^4 + x^2 + x + 1, \tag{10}$$

it can be shown that $h(K_1)$ will be unequal to $h(K_2)$ whenever K_1 and K_2 are distinct keys that differ in fewer than seven bit positions. (See exercise 7 for further information about this scheme; it is, of course, more suitable for hardware or microprogramming implementation than for software.)

It has been found convenient to use the constant hash function $h(K) = 0$ when debugging a program, since all keys will be stored together; an efficient $h(K)$ can be substituted later.

Collision resolution by "chaining." We have observed that some hash addresses will probably be burdened with more than their share of keys. Perhaps the most obvious way to solve this problem is to maintain M linked lists, one for each possible hash code. A LINK field should be included in each record, and there will also be M list heads, numbered say from 1 through M. After hashing the key, we simply do a sequential search in list number $h(K) + 1$. (Cf. exercise 6.1–2. The situation is very similar to multiple-list-insertion sorting, Program 5.2.1M.)

Figure 38 illustrates this simple chaining scheme when $M = 9$, for the sequence of seven keys

$$K = \texttt{EN, TO, TRE, FIRE, FEM, SEKS, SYV} \tag{11}$$

(i.e., the numbers 1 through 7 in Norwegian), having the respective hash codes

$$h(K) + 1 = 3,\quad 1,\quad 4,\quad 1,\quad 5,\quad 9,\quad 2. \tag{12}$$

The first list has two elements, and three of the lists are empty.

Chaining is quite fast, because the lists are short. If 365 people are gathered together in one room, there will probably be many pairs having the same birthday, but the average number of people with any given birthday will be only 1! In general, if there are N keys and M lists, the average list size is N/M; thus hashing decreases the amount of work needed for sequential searching by roughly a factor of M.

This method is a straightforward combination of techniques we have discussed before, so we do not need to formulate a detailed algorithm for chained scatter tables. It is often a good idea to keep the individual lists in order by key, so that insertions and unsuccessful searches go faster. Thus if we choose

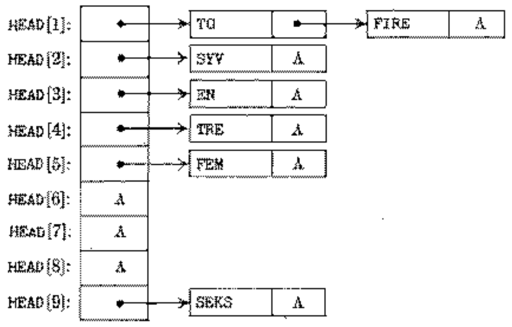


Fig. 38. Separate chaining.

to make the lists ascending, the TO and FIRE nodes of Fig. 38 would be interchanged, and all the Λ links would be replaced by pointers to a dummy record whose key is ∞. (Cf. Algorithm 6.1T.) Alternatively we can make use of the "self-organizing file" concept discussed in Section 6.1; instead of keeping the lists in order by key, they may be kept in order according to the time of most recent occurrence.

For the sake of speed we would like to make M rather large. But when M is large, many of the lists will be empty and much of the space for the M list heads will be wasted. This suggests another approach, when the records are small: We can overlap the record storage with the list heads, making room for a total of M records and M links instead of for N records and $M + N$ links. Sometimes it is possible to make one pass over all the data to find out which list heads will be used, then to make another pass inserting all the "overflow" records into the empty slots. But this is often impractical or impossible, and we would rather have a technique that processes each record only once when it first enters the system. The following algorithm, due to F. A. Williams [*CACM* 2, 6 (June 1959), 21–24], is a convenient way to solve the problem.

Algorithm C (*Chained scatter table search and insertion*). This algorithm searches an M-node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted.

The nodes of the table are denoted by TABLE[i], for $0 \le i \le M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key field KEY[i], a link field LINK[i], and possibly other fields.

The algorithm makes use of a hash function $h(K)$. An auxiliary variable R is also used, to help find empty spaces; when the table is empty, we have R $= M + 1$, and as insertions are made it will always be true that TABLE[j] is occupied for all j in the range R $\le j \le M$. By convention, TABLE[0] will always be empty.

C1. [Hash.] Set $i \leftarrow h(K) + 1$. (Now $1 \le i \le M$.)

C2. [Is there a list?] If TABLE[i] is empty, go to C6. (Otherwise TABLE[i] is occupied; we will look at the list of occupied nodes which starts here.)

C3. [Compare.] If $K =$ KEY[i], the algorithm terminates successfully.

C4. [Advance to next.] If LINK[i] $\ne 0$, set $i \leftarrow$ LINK[i] and go back to step C3.

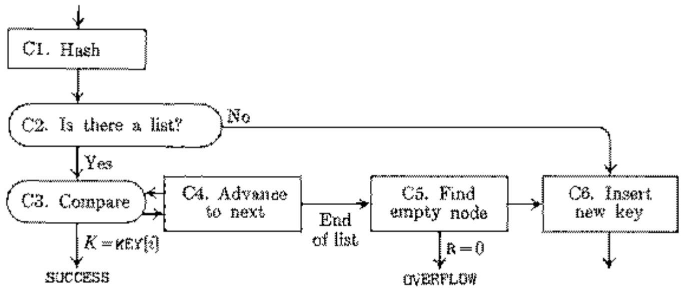


Fig. 39. Chained scatter table search and insertion.

C5. [Find empty node.] (The search was unsuccessful, and we want to find an empty position in the table.) Decrease R one or more times until finding a value such that TABLE[R] is empty. If R $= 0$, the algorithm terminates with overflow (there are no empty nodes left); otherwise set LINK[i] $\leftarrow$ R, $i \leftarrow$ R.

C6. [Insert new key.] Mark TABLE[i] as an occupied node, with KEY[i] $\leftarrow K$ and LINK[i] $\leftarrow 0$. ▮

This algorithm allows several lists to coalesce, so that records need not be moved after they have been inserted into the table. For example, see Fig. 40, where SEKS appears in the list containing TO and FIRE since the latter had already been inserted into position 9.

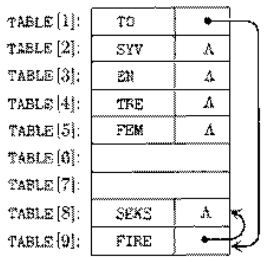


Fig. 40. Coalesced chaining.

GN 005754

In order to see how this algorithm compares with others in this chapter, we can write the following MIX program. The analysis worked out below indicates that the lists of occupied cells tend to be short, and the program has been designed with this fact in mind.

Program C (*Chained scatter table search and insertion*). For convenience, the keys are assumed to be only three bytes long, and nodes are represented as follows:

empty	−	1	0	0	0	0
occupied	+	LINK		KEY		

(13)

The table size M is assumed to be prime; TABLE[i] is stored in location TABLE + i. rI1 ≡ i, rA ≡ K.

```
KEY    EQU   3:5
LINK   EQU   0:2
START  LDX   K                    1         C1. Hash.
       ENTA  0                    1
       DIV   =M=                  1
       STX   *+1(0:2)             1
       ENT1  *                    1         i ← h(K)
       INC1  1                    1                 + 1.
       LDA   K                    1
       LD2   TABLE,1(LINK)        1         C2. Is there a list?
       J2N   6F                   1         To C6 if TABLE[i] empty.
       CMPA  TABLE,1(KEY)         A         C3. Compare.
       JE    SUCCESS              A         Exit if K = KEY[i].
       J2Z   5F                 A − S1      To C5 if LINK[i] = 0.
4H     ENT1  0,2                C − 1       C4. Advance to next.
       CMPA  TABLE,1(KEY)       C − 1       C3. Compare.
       JE    SUCCESS            C − 1       Exit if K = KEY[i].
       LD2   TABLE,1(LINK)    C − 1 − S2
       J2NZ  4B               C − 1 − S2    Advance if LINK[i] ≠ 0.
5H     LD2   R                  A − S       C5. Find empty node.
       DEC2  1                    T         R ← R − 1.
       LDX   TABLE,2              T
       JXNN  *-2                  T         Repeat until TABLE[R] empty.
       J2Z   OVERFLOW           A − S       Exit if no empty nodes left.
       ST2   TABLE,1(LINK)      A − S       LINK[i] ← R.
       ENT1  0,2                A − S       i ← R.
       ST2   R                  A − S       Update R in memory.
6H     STZ   TABLE,1(LINK)      1 − S       C6. Insert new key.
       STA   TABLE,1(KEY)       1 − S       KEY[i] ← K. ▮
```

The running time of this program depends on

C = number of table entries probed while searching;

A = 1 if the initial probe found an occupied node;

$S = 1$ if successful, 0 if unsuccessful;

$T =$ number of table entries probed while looking for an empty space.

Here $S = S1 + S2$, where $S1 = 1$ if successful on the first try. The total running time for the searching phase of Program C is $(7C + 4A + 17 - 3S + 2S1)u$, and the insertion of a new key when $S = 0$ takes an additional $(8A + 4T + 4)u$.

Suppose there are N keys in the table at the start of this program, and let

$$\alpha = N/M = \text{load factor of the table.} \tag{14}$$

Then the average value of A in an unsuccessful search is obviously α, if the hash function is random; and exercise 39 proves that the average value of C in an unsuccessful search is

$$C'_N = 1 + \frac{1}{4}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) \approx 1 + \tfrac{1}{4}(e^{2\alpha} - 1 - 2\alpha). \tag{15}$$

Thus when the table is half full, the average number of probes made in an unsuccessful search is about $\frac{1}{4}(e + 2) \approx 1.18$; and even when the table gets completely full, the average number of probes made just before inserting the final item will be only about $\frac{1}{4}(e^2 + 1) \approx 2.10$. The standard deviation is also small, as shown in exercise 40. These statistics prove that *the lists stay short even though the algorithm occasionally allows them to coalesce,* when the hash function is random. Of course C can be as high as N, if the hash function is bad or if we are extremely unlucky.

In a successful search, we always have $A = 1$. The average number of probes during a successful search may be computed by summing the quantity $C + A$ over the first N unsuccessful searches and dividing by N, if we assume that each key is equally likely. Thus we obtain

$$C_N = \frac{1}{N}\sum_{0 \le k < N}\left(C'_k + \frac{k}{M}\right) = 1 + \frac{1}{8}\,\frac{M}{N}\left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M}\right) + \frac{1}{4}\,\frac{N-1}{M}$$
$$\approx 1 + \frac{1}{8\alpha}\left(e^{2\alpha} - 1 - 2\alpha\right) + \tfrac{1}{4}\alpha \tag{16}$$

as the average number of probes in a random successful search. Even a full table will require only about 1.80 probes, on the average, to find an item! Similarly (see exercise 42), the average value of $S1$ turns out to be

$$S1_N = 1 - \tfrac{1}{2}((N - 1)/M) \approx 1 - \tfrac{1}{2}\alpha. \tag{17}$$

At first glance it may appear that step C5 is inefficient, since it has to search sequentially for an empty position. But actually the total number of table probes made in step C5 as a table is being built will never exceed the number of items in the table; so we make an average of at most one of these probes per insertion! Exercise 41 proves that T is approximately αe^{α} in a random unsuccessful search.

It would be possible to modify Algorithm C so that no two lists coalesce, but then it would become necessary to move records around. For example,

GN 005756

consider the situation in Fig. 40 just before we wanted to insert SEKS into position 9; in order to keep the lists separate, it would be necessary to move FIRE, and for this purpose it would be necessary to discover which node points to FIRE. We could solve this problem without providing two-way linkage by using circular lists, as suggested by Allen Newell in 1962, since the lists are short; but that would probably slow down the main search loop because step C4 would be more complicated. Exercise 34 shows that the average number of probes, when lists aren't coalesced, is reduced to

$$C'_N = \left(1 - \frac{1}{M}\right)^N + \frac{N}{M} \approx e^{-\alpha} + \alpha \qquad \text{(unsuccessful search);} \qquad (18)$$

$$C_N = \quad 1 + \frac{N-1}{2M} \quad \approx 1 + \tfrac{1}{2}\alpha \qquad \text{(successful search).} \qquad (19)$$

This is not enough of an improvement over (15) and (16) to warrant changing the algorithm.

On the other hand, Butler Lampson has observed that most of the space actually needed for links can actually be saved in the chaining method, if we avoid coalescing the lists. This leads to an interesting algorithm which is discussed in exercise 13.

Note that chaining can be used when $N > M$, so overflow is not a serious problem. If separate lists are used, formulas (18) and (19) are valid for $\alpha > 1$. When the lists coalesce as in Algorithm C, we can link extra items into an auxiliary storage pool; L. Guibas has proved that the average number of probes to insert the $(M + L + 1)$st item is then $(L/2M + \frac{1}{4})((1 + 2/M)^M - 1) + \frac{1}{2}$.

Collision resolution by "open addressing." Another way to resolve the problem of collisions is to do away with the links entirely, simply looking at various entries of the table one by one until either finding the key K or finding an empty position. The idea is to formulate some rule by which every key K determines a "probe sequence," namely a sequence of table positions which are to be inspected whenever K is inserted or looked up. If we encounter an open position while searching for K, using the probe sequence determined by K, we can conclude that K is not in the table, since the same sequence of probes will be made every time K is processed. This general class of methods was named *open addressing* by W. W. Peterson [*IBM J. Research & Development* 1 (1957), 130–146].

The simplest open addressing scheme, known as *linear probing*, uses the cyclic probe sequence

$$h(K),\ h(K) - 1,\ \ldots,\ 0,\ M - 1,\ M - 2,\ \ldots,\ h(K) + 1 \qquad (20)$$

as in the following algorithm.

Algorithm L (*Open scatter table search and insertion*). This algorithm searches an M-node table, looking for a given key K. If K is not in the table and the table is not full, K is inserted.

The nodes of the table are denoted by TABLE[i], for $0 \leq i < M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key, called KEY[i], and possibly other fields. An auxiliary variable N is used to keep track of how many nodes are occupied; this variable is considered to be part of the table, and it is increased by 1 whenever a new key is inserted.

This algorithm makes use of a hash function $h(K)$, and it uses the linear probing sequence (20) to address the table. Modifications of that sequence are discussed below.

L1. [Hash.] Set $i \leftarrow h(K)$. (Now $0 \leq i < M$.)

L2. [Compare.] If KEY[i] $= K$, the algorithm terminates successfully. Otherwise if TABLE[i] is empty, go to L4.

L3. [Advance to next.] Set $i \leftarrow i - 1$; if now $i < 0$, set $i \leftarrow i + M$. Go back to step L2.

L4. [Insert.] (The search was unsuccessful.) If N $= M - 1$, the algorithm terminates with overflow. (This algorithm considers the table to be full when N $= M - 1$, not when N $= M$; see exercise 15.) Otherwise set N $\leftarrow$ N $+ 1$, mark TABLE[i] occupied, and set KEY[i] $\leftarrow K$. ▌

Figure 41 shows what happens when the seven example keys (11) are inserted by Algorithm L, using the respective hash codes 2, 7, 1, 8, 2, 8, 1: The last three keys, FEM, SEKS, and SYV, have been displaced from their initial locations $h(K)$.

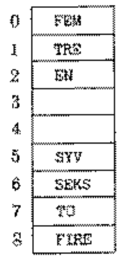


Fig. 41. Linear open addressing.

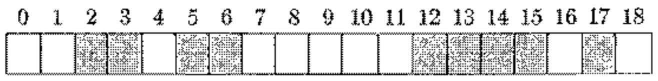
Program L (*Open scatter table search and insertion*). This program deals with full-word keys; but a key of 0 is not allowed, since 0 is used to signal an empty position in the table. (Alternatively, we could require the keys to be nonnegative, letting empty positions contain -1.) The table size M is assumed to be prime, and TABLE[i] is stored in location TABLE $+ i$ for $0 \leq i < M$. For speed in the inner loop, location TABLE $- 1$ is assumed to contain 0. Location VACANCIES is assumed to contain the value $M - 1 -$ N; and rA $\equiv K$, rI1 $\equiv i$.

In order to speed up the inner loop of this program, the test "$i < 0$" has been removed from the loop so that only the essential parts of steps L2 and L3 remain. The total running time for the searching phase comes to $(7C + 9E + 21 - 4S)u$, and the insertion after an unsuccessful search adds an extra $9u$.

```
START LDX   K                  1        L1. Hash.
      ENTA  0                  1
      DIV   =M=                1
      STX   *+1(0:2)           1
      ENT1  *                  1        i ← h(K).
      LDA   K                  1
      JMP   2F                 1
8H    INC1  M+1                E        L3. Advance to next.
3H    DEC1  1              C + E − 1    i ← i − 1.
2H    CMPA  TABLE,1          C + E      L2. Compare.
      JE    SUCCESS          C + E      Exit if K = KEY[i].
      LDX   TABLE,1        C + E − S
      JXNZ  3B             C + E − S    To L3 if TABLE[i] empty.
      J1N   8B             E + 1 − S    To L3 with i ← M if i = −1.
4H    LDX   VACANCIES        1 − S      L4. Insert.
      JXZ   OVERFLOW         1 − S      Exit with overflow if N = M − 1.
      DECX  1                1 − S
      STX   VACANCIES        1 − S      Increase N by 1.
      STA   TABLE,1          1 − S      TABLE[i] ← K. ▮
```

As in Program C, the variable C denotes the number of probes, and S tells whether or not the search was successful. We may ignore the variable E, which is 1 only if a spurious probe of TABLE[−1] has been made, since its average value is $(C-1)/M$.

Experience with linear probing shows that the algorithm works fine until the table begins to get full; but eventually the process slows down, with long drawn-out searches becoming increasingly frequent. The reason for this behavior can be understood by considering the following hypothetical scatter table with $M = 19$, $N = 9$:



(21)

Shaded squares represent occupied positions. The next key K to be inserted into the table will go into one of the ten empty spaces, but these are not equally likely; in fact, K will be inserted into position 11 if $11 \le h(K) \le 15$, while it will fall into position 8 only if $h(K) = 8$. Therefore position 11 is five times as likely as position 8; long lists tend to grow even longer.

This phenomenon isn't enough by itself to account for the relatively poor behavior of linear probing, since a similar thing occurs in Algorithm C. (A list of length 4 is four times as likely to grow in Algorithm C as a list of length 1.) The real problem occurs when a cell like 4 or 16 is filled in (21); then two separate lists are combined, while the lists in Algorithm C never grow by more than one step at a time. Consequently the performance of linear probing degrades rapidly when N approaches M.

We shall prove later in this section that the average number of probes needed by Algorithm L is approximately

$$C'_N \approx \frac{1}{2}\left(1+\left(\frac{1}{1-\alpha}\right)^2\right) \qquad \text{(unsuccessful search);} \qquad (22)$$

$$C_N \approx \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad \text{(successful search),} \qquad (23)$$

where $\alpha = N/M$ is the load factor of the table. So Program L is almost as fast as Program C, when the table is less than 75 percent full, in spite of the fact that Program C deals with unrealistically short keys. On the other hand, when α approaches 1 the best thing we can say about Program L is that it works, slowly but surely. In fact, when $N = M - 1$, there is only one vacant space in the table, so the average number of probes in an unsuccessful search is $(M+1)/2$; we shall also prove that the average number of probes in a successful search is approximately $\sqrt{\pi M/8}$ when the table is full.

The pileup phenomenon which makes linear probing costly on a nearly full table is aggravated by the use of division hashing, if consecutive key values $\{K, K+1, K+2, \ldots\}$ are likely to occur, since these keys will have consecutive hash codes. Multiplicative hashing will break up these clusters satisfactorily. Note that multiplicative hashing is slightly awkward for the chained method, since we generally would want to use at least $(m+1)$-bit link fields when $M = 2^m$ in order to distinguish rapidly between Λ and a valid link. This wasted bit position is no problem with open addressing since there are no links.

Another way to protect against the consecutive hash code problem is to set $i \leftarrow i - c$ in step L3, instead of $i \leftarrow i - 1$. Any positive value of c will do, so long as it is *relatively prime* to M, since the probe sequence will still examine every position of the table in this case. Such a change will make Program L somewhat slower. It doesn't alter the pileup phenomenon, since groups of c-apart records will still be formed; equations (22) and (23) will still apply, but the appearance of consecutive keys $\{K, K+1, K+2, \ldots\}$ will now actually be a help instead of a hindrance.

Although a fixed value of c does not reduce the pileup phenomenon, we can improve the situation nicely by letting c depend on K! This idea leads to an important modification of Algorithm L, first discovered by Guy de Balbine [Ph.D. thesis, Calif. Inst. of Technology (1968), 149–150]:

Algorithm D (*Open addressing with double hashing*). This algorithm is almost identical to Algorithm L, but it probes the table in a slightly different fashion by making use of two hash functions $h_1(K)$ and $h_2(K)$. As usual $h_1(K)$ produces a value between 0 and $M-1$, inclusive; but $h_2(K)$ must produce a value between 1 and $M-1$ that is *relatively prime* to M. (For example, if M is prime, $h_2(K)$ can be *any* value between 1 and $M-1$ inclusive; or if $M = 2^m$, $h_2(K)$ can be any *odd* value between 1 and $2^m - 1$.)

D1. [First hash.] Set $i \leftarrow h_1(K)$.

D2. [First probe.] If TABLE[i] is empty, go to D6. Otherwise if KEY[i] $= K$, the algorithm terminates successfully.

D3. [Second hash.] Set $c \leftarrow h_2(K)$.

D4. [Advance to next.] Set $i \leftarrow i - c$; if now $i < 0$, set $i \leftarrow i + M$.

D5. [Compare.] If TABLE[i] is empty, go to D6. Otherwise if KEY[i] $= K$, the algorithm terminates successfully. Otherwise go back to D4.

D6. [Insert.] If N $= M - 1$, the algorithm terminates with overflow. Otherwise set N $\leftarrow$ N $+ 1$, mark TABLE[i] occupied, and set KEY[i] $\leftarrow K$. ▮

Several possibilities have been suggested for computing $h_2(K)$. If M is prime and $h_1(K) = K \bmod M$, we might let $h_2(K) = 1 + (K \bmod (M-1))$; but since $M - 1$ is even, it would be better to let $h_2(K) = 1 + (K \bmod (M-2))$. This suggests choosing M so that M and $M - 2$ are "twin primes" like 1021 and 1019. Alternatively, we could set $h_2(K) = 1 + (\lfloor K/M \rfloor \bmod (M-2))$, since the quotient $\lfloor K/M \rfloor$ might be available in a register as a by-product of the computation of $h_1(K)$.

If $M = 2^m$ and we are using multiplicative hashing, $h_2(K)$ can be computed simply by shifting left m more bits and "oring in" a 1, so that the coding sequence (5) would be followed by

```
ENTA  0      Clear rA.
SLB   m      Shift rAX m bits left.          (24)
ORR   =1=    rA ← rA ∨ 1.
```

This is faster than the division method.

In each of the techniques suggested above, $h_1(K)$ and $h_2(K)$ are "independent," in the sense that different keys will have the same value for both h_1 and h_2 with probability $O(1/M^2)$ instead of $O(1/M)$. Empirical tests show that the behavior of Algorithm D with independent hash functions is essentially indistinguishable from the number of probes which would be required if the keys were inserted at random into the table; there is practically no "piling up" or "clustering" as in Algorithm L.

It is also possible to let $h_2(K)$ depend on $h_1(K)$, as suggested by Gary Knott in 1968; for example, if M is prime we could let

$$h_2(K) = \begin{cases} 1, & \text{if} \quad h_1(K) = 0; \\ M - h_1(K), & \text{if} \quad h_1(K) > 0. \end{cases} \tag{25}$$

This would be faster than doing another division, but we shall see that it does cause a certain amount of *secondary clustering*, requiring slightly more probes because of the increased chance that two or more keys will follow the same path. The formulas derived below can be used to determine whether the gain in hashing time outweighs the loss of probing time.

Algorithms L and D are very similar, yet there are enough differences that it is instructive to compare the running time of the corresponding MIX programs.

Program D (*Open addressing with double hashing*). Since this program is substantially like Program L, it is presented without comments. rI2 $\equiv c - 1$.

```
START LDX   K             1
      ENTA  0             1
      DIV   =M=           1
      STX   *+1(0:2)      1
      ENT1  *             1
      LDX   TABLE,1       1
      CMPX  K             1
      JE    SUCCESS       1
      JXZ   4F            1 − S1
      SRAX  5             A − S1
      DIV   =M-2=         A − S1
      STX   *+1(0:2)      A − S1
      ENT2  *             A − S1
      LDA   K             A − S1
3H    DEC1  1,2           C − 1
      J1NN  *+2           C − 1
      INC1  M             B
      CMPA  TABLE,1       C − 1
      JE    SUCCESS       C − 1
      LDX   TABLE,1       C − 1 − S2
      JXNZ  3B            C − 1 − S2
4H    LDX   VACANCIES     1 − S
      JXZ   OVERFLOW      1 − S
      DECX  1             1 − S
      STX   VACANCIES     1 − S
      LDA   K             1 − S
      STA   TABLE, 1      1 − S ▮
```

The frequency counts A, C, $S1$, $S2$ in this program have a similar interpretation to those in Program C above. The other variable B will be about $\frac{1}{2}(C-1)$ on the average. (If we restricted the range of $h_2(K)$ to, say, $1 \le h_2(K) \le \frac{1}{2}M$, B would be only about $\frac{1}{4}(C-1)$; this increase of speed will probably *not* be offset by a noticeable increase in the number of probes.) When there are $N = \alpha M$ keys in the table, the average value of A is, of course, α in an unsuccessful search, and $A = 1$ in a successful search. As in Algorithm C, the average value of $S1$ in a successful search is $1 - \frac{1}{2}((N-1)/M) \approx 1 - \frac{1}{2}\alpha$. The average number of probes is difficult to determine exactly, but empirical tests show good agreement with formulas derived below for "uniform probing," namely

$$C'_N = \frac{M+1}{M+1-N} \approx (1-\alpha)^{-1} \qquad \text{(unsuccessful search)}, \tag{26}$$

$$C_N = \frac{M+1}{N}(H_{M+1} - H_{M+1-N}) \approx -\alpha^{-1}\ln(1-\alpha) \qquad \text{(successful search)}, \tag{27}$$

when $h_1(K)$ and $h_2(K)$ are independent. When $h_2(K)$ depends on $h_1(K)$ as in (25), the secondary clustering causes these formulas to be increased to

$$C'_N = \frac{M+1}{M+1-N} - \frac{N}{M+1} + H_{M+1} - H_{M+1-N} + O(M^{-1}) \approx (1-\alpha)^{-1} - \alpha - \ln(1-\alpha); \tag{28}$$

$$C_N = 1 + H_{M+1} - H_{M+1-N} - \frac{N}{2(M+1)} - (H_{M+1} - H_{M+1-N})/N + O(N^{-1}) \approx 1 - \ln(1-\alpha) - \tfrac{1}{2}\alpha. \tag{29}$$

(See exercise 44.) Note that as the table gets full, these values of C_N approach $H_{M+1} - 1$ and $H_{M+1} - \frac{1}{2}$, respectively, when $N = M$; this is much better than we observed in Algorithm L, but not as good as in the chaining methods.

Since each probe takes slightly less time in Algorithm L, double hashing is advantageous only when the table gets full. Figure 42 compares the average running time of Program L, Program D, and a modified Program D which involves secondary clustering, replacing the rather slow calculation of $h_2(K)$ in lines 10–13 by the three instructions

```
ENN2 -M-1,1          c ← M − i.
J1NZ *+2                                    (30)
ENT2 0               If i = 0, c ← 1.
```

In this case, secondary clustering is preferable to independent double hashing.

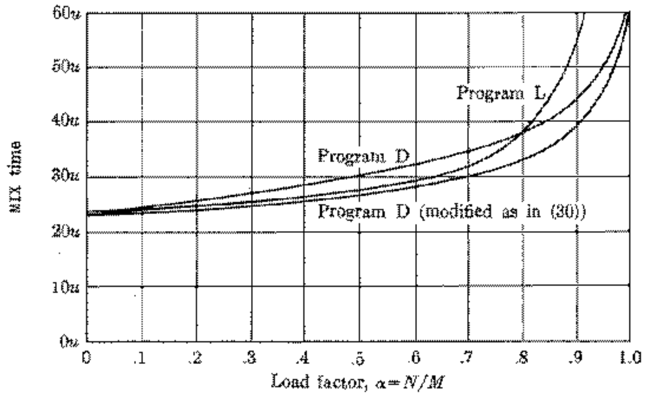


Fig. 42. The running time for successful searching by three open addressing schemes.

On a binary computer, we could speed up the computation of $h_2(K)$ in another way, replacing lines 10–13 by, say,

```
AND  =511=           rA ← rA mod 512.
STA  *+1(0:2)                               (31)
ENT2 *               c ← rA + 1.
```

if M is a prime greater than 512. This idea (suggested by Bell and Kaman, *CACM* 13 (1970), 675–677, who discovered Algorithm D independently) avoids secondary clustering without the expense of another division.

Many other probe sequences have been proposed as improvements on Algorithm L, but none seem to be superior to Algorithm D except possibly the method described in exercise 20.

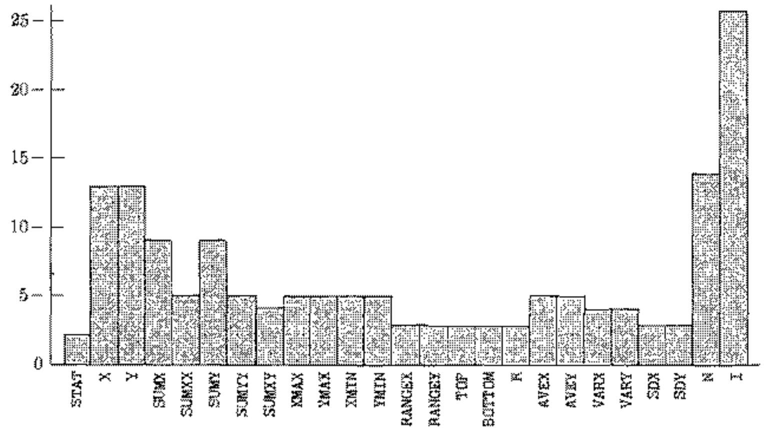


Fig. 43. The number of times a compiler typically searches for variable names. The names are listed from left to right in order of their first appearance.

Brent's Variation. Richard P. Brent has discovered a way to modify Algorithm D so that the average successful search time remains bounded as the table gets full. His method is based on the fact that successful searches are much more common than insertions, in many applications; therefore he proposes doing more work when inserting an item, moving records in order to reduce the expected retrieval time. [*CACM* **16** (1973), 105–109.]

For example, Fig. 43 shows the number of times each identifier was actually found to appear, in a typical PL/I procedure. This data indicates that a PL/I compiler which uses a hash table to keep track of variable names will be looking up many of the names five or more times but inserting them only once. Similarly, Bell and Kaman found that a COBOL compiler used its symbol table algorithm 10988 times while compiling a program, but made only 735 insertions into the table; this is an average of about 14 successful searches per unsuccessful search. Sometimes a table is actually created only once (for example, a table of symbolic opcodes in an assembler), and it is used thereafter purely for retrieval.

Brent's idea is to change the insertion process in Algorithm D as follows. Suppose an unsuccessful search has probed locations

$$p_0, p_1, \ldots, p_{t-1}, p_t,$$

where $p_j = (h_1(K) - jh_2(K)) \bmod M$ and $\text{TABLE}[p_t]$ is empty. If $t \leq 1$, we insert K in position p_t as usual; but if $t \geq 2$, we compute $c_0 = h_2(K_0)$, where $K_0 = \text{KEY}[p_0]$, and see if $\text{TABLE}[(p_0 - c_0) \bmod M]$ is empty. If it is, we set it to

GN 005764

TABLE[p_0] and then insert K in position p_0. This increases the retrieval time for K_0 by one step, but it decreases the retrieval time for K by $t \geq 2$ steps, so it results in a net improvement. Similarly, if TABLE[$(p_0 - c_0) \bmod M$] is occupied and $t \geq 3$, we try TABLE[$(p_0 - 2c_0) \bmod M$]; if that is full too, we compute $c_1 = h_2$(KEY[p_1]) and try TABLE[$(p_1 - c_1) \bmod M$]; etc. In general, let $c_j = h_2$(KEY[p_j]) and $p_{j,k} = (p_j - kc_j) \bmod M$; if we have found TABLE[$p_{j,k}$] occupied for all indices j, k such that $j + k < r$, and if $t \geq r + 1$, we look at TABLE[$p_{0,r}$], TABLE[$p_{1,r-1}$], $\ldots$, TABLE[$p_{r-1,1}$]. If the first empty space occurs at position $p_{j,r-j}$ we set TABLE[$p_{j,r-j}$] $\leftarrow$ TABLE[p_j] and insert K in position p_j.

Brent's analysis indicates that the average number of probes per successful search is reduced as shown in Fig. 44, on page 539, with a maximum value of about 2.49.

The number $t + 1$ of probes in an unsuccessful search is not reduced by Brent's variation; it remains at the level indicated by Eq. (26), approaching $\frac{1}{2}(M + 1)$ as the table gets full. The average number of times h_2 needs to be computed per insertion is $\alpha^2 + \alpha^5 + \frac{1}{3}\alpha^6 + \cdots$, according to Brent's analysis, eventually approaching the order of $\sqrt{M}$; and the number of additional table positions probed while deciding how to make the insertion is about $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \cdots$.

Deletions. Many computer programmers have great faith in algorithms, and they are surprised to find that *the obvious way to delete records from a scatter table doesn't work*. For example, if we try to delete the key EN from Fig. 41, we can't simply mark that table position empty, because another key FEM would suddenly be forgotten! (Recall that EN and FEM both hashed to the same location. When looking up FEM, we would find an empty place, indicating an unsuccessful search.) A similar problem occurs with Algorithm C, due to the coalescing of lists; imagine the deletion of both TO and FIRE from Fig. 40.

In general, we can handle deletions by putting a special code value in the corresponding cell, so that there are three kinds of table entries: empty, occupied, and deleted. When searching for a key, we should skip over deleted cells, as if they were occupied. If the search is unsuccessful, the key can be inserted in place of the first deleted or empty position that was encountered.

But this idea is workable only when deletions are very rare, because the entries of the table never become empty again once they have been occupied. After a long sequence of repeated insertions and deletions, all of the empty spaces will eventually disappear, and every unsuccessful search will take M probes! Furthermore the time per probe will be increased, since we will have to test whether i has returned to its starting value in step D4; and the number of probes in a successful search will drift upward from C_N to C'_N.

When linear probing is being used (i.e., Algorithm L), it is possible to do deletions in a way that avoids such a sorry state of affairs, if we are willing to do some extra work for the deletion.

Algorithm R (*Deletion with linear probing*). Assuming that an open scatter table has been constructed by Algorithm L, this algorithm deletes the record from a given position TABLE[i].

R1. [Empty a cell.] Mark TABLE[i] empty, and set $j \leftarrow i$.

R2. [Decrease i.] Set $i \leftarrow i - 1$, and if this makes i negative set $i \leftarrow i + M$.

R3. [Inspect TABLE[i].] If TABLE[i] is empty, the algorithm terminates. Otherwise set $r \leftarrow h(\text{KEY}[i])$, the original hash address of the key now stored at position i. If $i \le r < j$ or if $r < j < i$ or $j < i \le r$ (in other words, if r lies cyclically between i and j), go back to R2.

R4. [Move a record.] Set TABLE[j] $\leftarrow$ TABLE[i], and return to step R1. ▮

Exercise 22 shows that this algorithm causes no degradation in performance, i.e., the average number of probes predicted in Eqs. (22) and (23) remains the same. (A similar result for tree insertion was proved in Theorem 6.2.2H.) But the validity of Algorithm R depends heavily on the fact that linear probing is involved, and no analogous deletion procedure for use with Algorithm D is possible.

Of course when chaining is used with separate lists for each possible hash value, deletion causes no problems since it is simply a deletion from a linked linear list. Deletion with Algorithm C is discussed in exercise 23.

***Analysis of the algorithms.** It is especially important to know the average behavior of a hashing method, because we are committed to trusting in the laws of probability whenever we hash. The worst case of these algorithms is almost unthinkably bad, so we need to be reassured that the average behavior is very good.

Before we get into the analysis of linear probing, etc., let us consider a very approximate model of the situation, which may be called *uniform hashing* (cf. W. W. Peterson, *IBM J. Research & Development* **1** (1957), 135–136). In this model, we assume that the keys go into random locations of the table, so that each of the $\binom{M}{N}$ possible configurations of N occupied cells and $M - N$ empty cells is equally likely. This model ignores any effect of primary or secondary clustering; the occupancy of each cell in the table is essentially independent of the others. For this model the probability that exactly r probes are needed to insert the $(N+1)$st item is the number of configurations in which $r - 1$ given cells are occupied and another is empty, divided by $\binom{M}{N}$, namely

$$P_r = \binom{M-r}{N-r+1} \Big/ \binom{M}{N};$$

therefore the average number of probes for uniform hashing is

$$\begin{aligned} C'_N &= \sum_{1\le r\le M} rP_r = M + 1 - \sum_{1\le r\le M} (M+1-r)P_r \\ &= M + 1 - \sum_{1\le r\le M} (M+1-r)\binom{M-r}{M-N-1} \Big/ \binom{M}{N} \end{aligned}$$

$$\begin{aligned}&= M+1-\sum_{1\le r\le M}(M-N)\binom{M+1-r}{M-N}\Big/\binom{M}{N} \qquad (32)\\ &= M+1-(M-N)\binom{M+1}{M-N+1}\Big/\binom{M}{N}\\ &= M+1-(M-N)\frac{M+1}{M-N+1}=\frac{M+1}{M-N+1}, \quad \text{for} \quad 1\le N<M.\end{aligned}$$

(We have already solved essentially the same problem in connection with random sampling, in exercise 3.4.2–5.) Setting $\alpha = N/M$, this exact formula for C'_N is approximately equal to

$$\frac{1}{1-\alpha}=1+\alpha+\alpha^2+\alpha^3+\cdots, \qquad (33)$$

a series which has a rough intuitive interpretation: With probability α we need more than one probe, with probability α^2 we need more than two, etc. The corresponding average number of probes for a successful search is

$$\begin{aligned}C_N=\frac{1}{N}\sum_{0\le k<N}C'_k&=\frac{M+1}{N}\left(\frac{1}{M+1}+\frac{1}{M}+\cdots+\frac{1}{M-N+2}\right)\\ &=\frac{M+1}{N}(H_{M+1}-H_{M-N+1})\approx\frac{1}{\alpha}\ln\frac{1}{1-\alpha}.\end{aligned} \qquad (34)$$

As remarked above, extensive tests show that Algorithm D with two independent hash functions behaves essentially like uniform hashing, for all practical purposes.

This completes the analysis of uniform hashing. In order to study linear probing and other types of collision resolution, we need to set up the theory in a different, more realistic way. The probabilistic model we shall use for this purpose assumes that each of the M^N possible "hash sequences"

$$a_1\,a_2\ldots a_N, \qquad 0\le a_j<M, \qquad (35)$$

is equally likely, where a_j denotes the initial hash address of the jth key inserted into the table. The average number of probes in a successful search, given any particular searching algorithm, will be denoted by C_N as above; this is assumed to be the average number of probes needed to find the kth key, averaged over $1\le k\le N$ with each key equally likely, and averaged over all hash sequences (35) with each sequence equally likely. Similarly, the average number of probes needed when the Nth key is inserted, considering all sequences (35) to be equally likely, will be denoted by C'_{N-1}; this is the average number of probes in an unsuccessful search starting with $N-1$ elements in the table. When open addressing is used,

$$C_N=\frac{1}{N}\sum_{0\le k<N}C'_k, \qquad (36)$$

so that we can deduce one quantity from the other as we have done in (34).

Strictly speaking, there are two defects even in this more accurate model. In the first place, the different hash sequences aren't all equally probable, because the keys themselves are distinct. This makes the probability that $a_1 = a_2$ slightly less than $1/M$; but the difference is usually negligible since the set of all possible keys is typically very large compared to M. (See exercise 24.) Furthermore a good hash function will exploit the nonrandomness of typical data, making it even less likely that $a_1 = a_2$; as a result, our estimates for the number of probes will be pessimistic. Another inaccuracy in the model is indicated in Fig. 43: Keys that occur earlier are (with some exceptions) more likely to be looked up than keys that occur later. Therefore our estimate of C_N tends to be doubly pessimistic, and the algorithms should perform slightly better in practice than our analysis predicts.

With these precautions, we are ready to make an "exact" analysis of linear probing.* Let $f(M, N)$ be the number of hash sequences (35) such that position 0 of the table will be empty after the keys have been inserted by Algorithm L. The circular symmetry of linear probing implies that position 0 is empty just as often as any other position, so it is empty with probability $1 - N/M$; in other words

$$f(M, N) = \left(1 - \frac{N}{M}\right) M^N. \tag{37}$$

Now let $g(M, N, k)$ be the number of hash sequences (35) such that the algorithm leaves position 0 empty, positions 1 through k occupied, and position $k + 1$ empty. We have

$$g(M, N, k) = \binom{N}{k} f(k + 1, k)\, f(M - k - 1, N - k), \tag{38}$$

because all such hash sequences are composed of two subsequences, one (containing k elements $a_i \leq k$) that leaves position 0 empty and positions 1 through k occupied and one (containing $N - k$ elements $a_j \geq k + 1$) that leaves position $k + 1$ empty; there are $f(k + 1, k)$ subsequences of the former type and $f(M - k - 1, N - k)$ of the latter, and there are $\binom{N}{k}$ ways to intersperse two such subsequences. Finally let P_k be the probability that exactly $k + 1$ probes will be needed when the $(N + 1)$st key is inserted; it follows (see exercise 25) that

$$P_k = M^{-N}\bigl(g(M, N, k) + g(M, N, k + 1) + \cdots + g(M, N, N)\bigr). \tag{39}$$

Now $C'_N = \sum_{0 \leq k \leq N} (k + 1) P_k$; putting this equation together with (36)–(39) and simplifying yields the following result.

* The author cannot resist inserting a biographical note at this point: I first formulated the following derivation in 1962, shortly after beginning work on *The Art of Computer Programming*. Since this was the first nontrivial algorithm I had ever analyzed satisfactorily, it has had a strong influence on the structure of these books. Little did I know that more than ten years would go by before this derivation got into print!

Theorem K. *The average number of probes needed by Algorithm L, assuming that all M^N hash sequences* (35) *are equally likely, is*

$$C_N = \tfrac{1}{2}(1 + Q_0(M, N-1)) \qquad \text{(successful search)}, \tag{40}$$

$$C'_N = \tfrac{1}{2}(1 + Q_1(M, N)) \qquad \text{(unsuccessful search)}, \tag{41}$$

where

$$Q_r(M,N) = \binom{r}{0} + \binom{r+1}{1}\frac{N}{M} + \binom{r+2}{2}\frac{N(N-1)}{M^2} + \cdots$$
$$= \sum_{k\ge 0}\binom{r+k}{k}\frac{N}{M}\,\frac{N-1}{M}\cdots\frac{N-k+1}{M}. \tag{42}$$

Proof. Details of the calculation are worked out in exercise 27. ▮

The rather strange-looking function $Q_r(M, N)$ which appears in this theorem is really not hard to deal with. We have

$$N^k - \binom{k}{2}N^{k-1} \le N(N-1)\cdots(N-k+1) \le N^k;$$

hence if $N/M = \alpha$,

$$\sum_{k\ge 0}\binom{r+k}{k}\left(N^k - \binom{k}{2}N^{k-1}\right)\Big/ M^k \le Q_r(M,N) \le \sum_{k\ge 0}\binom{r+k}{k}N^k/M^k,$$
$$\sum_{k\ge 0}\binom{r+k}{k}\alpha^k - \frac{\alpha}{M}\sum_{k\ge 0}\binom{r+k}{k}\binom{k}{2}\alpha^{k-2} \le Q_r(M,\alpha M) \le \sum_{k\ge 0}\binom{r+k}{k}\alpha^k,$$

i.e.,

$$\frac{1}{(1-\alpha)^{r+1}} - \frac{1}{M}\binom{r+2}{2}\frac{\alpha}{(1-\alpha)^{r+3}} \le Q_r(M,\alpha M) \le \frac{1}{(1-\alpha)^{r+1}}. \tag{43}$$

This relation gives us a good estimate of $Q_r(M, N)$ when M is large and α is not too close to 1. (The lower bound is a better approximation than the upper bound.) When α approaches 1, these formulas become useless, but fortunately $Q_0(M, M-1)$ is the function $Q(M)$ whose asymptotic behavior was studied in great detail in Section 1.2.11.3; and $Q_1(M, M-1)$ is simply equal to M (see exercise 50).

Another approach to the analysis of linear probing has been taken by G. Schay, Jr. and W. G. Spruth [*CACM* **5** (1962), 459–462]. Although their method yields only an approximation to the exact formulas in Theorem K, it sheds further light on the algorithm, so we shall sketch it briefly here. First let us consider a surprising property of linear probing which was first noticed by W. W. Peterson in 1957:

Theorem P. *The average number of probes in a successful search by Algorithm L is independent of the order in which the keys were inserted; it depends only on the number of keys which hash to each address.*

GN 005769

In other words, any rearrangement of a hash sequence $a_1\, a_2 \ldots a_N$ yields a hash sequence with the same average displacement of keys from their hash addresses. (We are assuming, as stated earlier, that all keys in the table have equal importance. If some keys are more frequently accessed than others, the proof can be extended to show that an optimal arrangement occurs if we insert them in decreasing order of frequency, by the method of Theorem 6.1S.)

Proof. It suffices to show that the total number of probes needed to insert keys for the hash sequence $a_1\, a_2 \ldots a_N$ is the same as the total number needed for $a_1 \ldots a_{i-1} a_{i+1} a_i a_{i+2} \ldots a_N$, $1 \le i < N$. There is clearly no difference unless the $(i+1)$st key in the second sequence falls into the position occupied by the ith in the first sequence. But then the ith and $(i+1)$st merely exchange places, so the number of probes for the $(i+1)$st is decreased by the same amount that the number for the ith is increased. ▌

Theorem P tells us that the average search length for a hash sequence $a_1\, a_2 \ldots a_N$ can be determined from the numbers $b_0\, b_1 \ldots b_{M-1}$, where b_j is the number of a's that equal j. From this sequence we can determine the "carry sequence" $c_0\, c_1 \ldots c_{M-1}$, where c_j is the number of keys for which both locations j and $j-1$ are probed as the key is inserted. This sequence is determined by the rule

$$c_j = \begin{cases} 0, & \text{if} \quad b_j = c_{(j+1) \bmod M} = 0; \\ b_j + c_{(j+1) \bmod M} - 1, & \text{otherwise.} \end{cases} \tag{44}$$

For example, let $M = 10$, $N = 8$, and $b_0 \ldots b_9 = 0\ 3\ 2\ 0\ 1\ 0\ 0\ 0\ 0\ 2$; then $c_0 \ldots c_9 = 2\ 3\ 1\ 0\ 0\ 0\ 0\ 1\ 2\ 3$, since one key needs to be "carried over" from position 2 to position 1, three from position 1 to position 0, two of these from position 0 to position 9, etc. We have $b_0 + b_1 + \cdots + b_{M-1} = N$, and the average number of probes needed for retrieval of the N keys is

$$1 + (c_0 + c_1 + \cdots + c_{M-1})/N. \tag{45}$$

Rule (44) seems to be a circular definition of the c's in terms of themselves, but actually there is a unique solution to the stated equations whenever $N < M$ (see exercise 32).

Schay and Spruth used this idea to determine the probability q_k that $c_j = k$, in terms of the probability p_k that $b_j = k$. (These probabilities are independent of j.) Thus

$$\begin{aligned} q_0 &= p_0 q_0 + p_1 q_0 + p_0 q_1, \\ q_1 &= p_2 q_0 + p_1 q_1 + p_0 q_2, \\ q_2 &= p_3 q_0 + p_2 q_1 + p_1 q_2 + p_0 q_3, \end{aligned} \tag{46}$$

etc., since, for example, the probability that $c_j = 2$ is the probability that $b_j + c_{(j+1) \bmod M} = 3$. Let $B(z) = \sum p_k z^k$ and $C(z) = \sum q_k z^k$ be the generating functions for these probability distributions; the equations (46) are equivalent to

$$B(z)C(z) = p_0 q_0 + (q_0 - p_0 q_0)z + q_1 z^2 + \cdots = p_0 q_0 (1 - z) + zC(z).$$

GN 005770

Since $B(1) = 1$, we may write $B(z) = 1 + (z - 1)D(z)$, and it follows that

$$C(z) = \frac{p_0 q_0}{1 - D(z)} = \frac{1 - D(1)}{1 - D(z)}, \tag{47}$$

since $C(1) = 1$. The average number of probes needed for retrieval, according to (45), will therefore be

$$1 + \frac{M}{N} C'(1) = 1 + \frac{M}{N} \frac{D'(1)}{1 - D(1)} = 1 + \frac{M}{2N} \frac{B''(1)}{1 - B'(1)}. \tag{48}$$

Since we are assuming that each hash sequence $a_1 \ldots a_N$ is equally likely we, have

$$\begin{aligned} p_k &= \Pr(\text{exactly } k \text{ of the } a_i \text{ are equal to } j, \text{ for fixed } j) \\ &= \binom{N}{k}\left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}; \end{aligned} \tag{49}$$

hence

$$B(z) = \left(1 + \frac{z - 1}{M}\right)^N, \qquad B'(1) = \frac{N}{M}, \qquad B''(1) = \frac{N(N - 1)}{M^2}, \tag{50}$$

and the average number of probes according to (48) will be

$$C_N = \frac{1}{2}\left(1 + \frac{M - 1}{M - N}\right). \tag{51}$$

Can the reader see why this answer is different from the result in Theorem K? (Cf. exercise 33.)

***Optimality considerations.** We have seen several examples of probe sequences for open addressing, and it is natural to ask for one that can be proved "best possible" in some meaningful sense. This problem has been set up in the following interesting way by J. D. Ullman [*JACM* **19** (1972), 569–575]: Instead of computing a "hash address" $h(K)$, we map each key K into an entire permutation of $\{0, 1, \ldots, M - 1\}$, which represents the probe sequence to use for K. Each of the $M!$ permutations is assigned a probability, and the generalized hash function is supposed to select each permutation with that probability. The question is, "What assignment of probabilities to permutations gives the best performance, in the sense that the corresponding average number of probes C_N or C'_N is minimized?"

For example, if we assign the probability $1/M!$ to each permutation, it is easy to see that we have exactly the behavior of *uniform hashing* which we have analyzed above in (32), (34). However, Ullman has found an example with $M = 4$, $N = 2$ for which C'_N is smaller than the value $\frac{5}{3}$ obtained with uniform hashing. His construction assigns zero probability to all but the following six permutations:

Permutation	Probability	Permutation	Probability	
0 1 2 3	$(1+2\epsilon)/6$	1 0 3 2	$(1+2\epsilon)/6$	
2 0 1 3	$(1-\epsilon)/6$	2 1 0 3	$(1-\epsilon)/6$	(52)
3 0 1 2	$(1-\epsilon)/6$	3 1 0 2	$(1-\epsilon)/6$	

Roughly speaking, the first choice favors 2 and 3, but the second choice is 0 or 1. The average number of probes needed to insert the third item turns out to be $\frac{5}{3} - \frac{1}{9}\epsilon + O(\epsilon^2)$, so we can improve on uniform hashing by taking ϵ to be a small positive value.

However, the corresponding value of C'_1 for these probabilities is $\frac{23}{18} + O(\epsilon)$, which is larger than $\frac{5}{4}$ (the uniform hashing value). Ullman has proved that any assignment of probabilities such that $C'_N < (M+1)/(M+1-N)$ for some N always implies that $C'_n > (M+1)/(M+1-n)$ for some $n < N$; you can't win all the time over uniform hashing.

Actually the number of probes C_N for a *successful* search is a better measure than C'_N. The permutations in (52) do not lead to an improved value of C_N for any N, and indeed it seems reasonable to conjecture that no assignment of probabilities will be able to make C_N less than the uniform value $((M+1)/N) \times (H_{M+1} - H_{M+1-N})$.

This conjecture appears to be very difficult to prove, especially because there are many ways to assign probabilities to achieve the effect of uniform hashing; we do not need to assign $1/M!$ to each permutation. For example, the following assignment for $M = 4$ is equivalent to uniform hashing:

Permutation	Probability	Permutation	Probability	
0 1 2 3	1/6	0 2 1 3	1/12	
1 2 3 0	1/6	1 3 2 0	1/12	(53)
2 3 0 1	1/6	2 0 3 1	1/12	
3 0 1 2	1/6	3 1 0 2	1/12	

with zero probability assigned to the other 16 permutations.

The following theorem characterizes *all* assignments that produce the behavior of uniform hashing.

Theorem U. *An assignment of probabilities to permutations will make each of the $\binom{M}{N}$ configurations of empty and occupied cells equally likely after N insertions, for $0 < N < M$, if and only if the sum of probabilities assigned to all permutations whose first N elements are the members of a given N-element set is $1/\binom{M}{N}$, for all N and all N-element sets.*

For example, the sum of probabilities assigned to each of the $3!(M-3)!$ permutations beginning with the numbers $\{0, 1, 2\}$ in some order must be $1/\binom{M}{3} = 3!(M-3)!/M!$. Observe that the condition of this theorem holds in (53).

Proof. Let $A \subseteq \{0, 1, \ldots, M-1\}$, and let $\prod(A)$ be the set of all permutations whose first $\|A\|$ elements are members of A, and let $S(A)$ be the sum of the probabilities assigned to these permutations. Let $P_k(A)$ be the probability that the first $\|A\|$ insertions of the open addressing procedure occupy the locations specified by A, and that the last insertion required exactly k probes; and let $P(A) = P_1(A) + P_2(A) + \cdots$. The proof is by induction on $N \geq 1$, assuming that

$$P(A) = S(A) = 1\bigg/\binom{M}{n}$$

for all sets A with $\|A\| = n < N$. Let B be any N-element set. Then

$$P_k(B) = \sum_{\substack{A \subseteq B \\ \|A\| = k}} \sum_{\pi \in \Pi(A)} \Pr(\pi) P(B \setminus \{\pi_k\}),$$

where $\Pr(\pi)$ is the probability assigned to permutation π and π_k is its kth element. By induction

$$P_k(B) = \sum_{\substack{A \subseteq B \\ \|A\| = k}} \frac{1}{\binom{M}{N-1}} \sum_{\pi \in \Pi(A)} \Pr(\pi),$$

which equals

$$\binom{N}{k}\bigg/\binom{M}{N-1}\binom{M}{k}, \qquad \text{if} \qquad k < N;$$

hence

$$P(B) = \frac{1}{\binom{M}{N-1}} \left(S(B) + \sum_{1 \leq k < N} \frac{\binom{N}{k}}{\binom{M}{k}} \right),$$

and this can be equal to $1/\binom{M}{N}$ if and only if $S(B)$ has the correct value. ∎

External searching. Hashing techniques lend themselves well to external searching on direct-access storage devices like disks or drums. For such applications, as in Section 6.2.4, we want to minimize the number of accesses to the file, and this has two major effects on the choice of algorithms:

1) It is reasonable to spend more time computing the hash function, since the penalty for bad hashing is much greater than the cost of the extra time needed to do a careful job.
2) The records are usually grouped into *buckets*, so that several records are fetched from the external memory each time.

The file is usually divided into M buckets containing b records each. Collisions now cause no problem unless more than b keys have the same hash address. The following three approaches to collision resolution seem to be best:

A) *Chaining with separate lists.* If more than b records fall into the same bucket, a link to an "overflow" record can be inserted at the end of the first bucket. These overflow records are kept in a special overflow area. There is usually no advantage in having buckets in the overflow area, since comparatively few overflows occur; thus, the extra records are usually linked together so that the $(b+k)$th record of a list requires $1+k$ accesses. It is usually a good idea to leave some room for overflows on each "cylinder" of a disk file, so that most accesses are to the same cylinder.

Although this method of handling overflows seems inefficient, the number of overflows is statistically small enough that the average search time is very good. See Tables 2 and 3, which show the average number of accesses required as a function of the load factor

$$\alpha = N/Mb, \tag{54}$$

for fixed α as $M, N \to \infty$. Curiously when $\alpha = 1$ the asymptotic number of accesses for an unsuccessful search increases with increasing b.

Table 2

AVERAGE ACCESSES IN AN UNSUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket	Load factor, α									
size, b	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0048	1.0187	1.0408	1.0703	1.1065	1.1488	1.197	1.249	1.307	1.3
2	1.0012	1.0088	1.0269	1.0581	1.1036	1.1638	1.238	1.327	1.428	1.5
3	1.0003	1.0038	1.0162	1.0433	1.0898	1.1588	1.252	1.369	1.509	1.6
4	1.0001	1.0016	1.0095	1.0314	1.0751	1.1476	1.253	1.394	1.571	1.7
5	1.0000	1.0007	1.0056	1.0225	1.0619	1.1346	1.249	1.410	1.620	1.7
10	1.0000	1.0000	1.0004	1.0041	1.0222	1.0773	1.201	1.426	1.773	2.0
20	1.0000	1.0000	1.0000	1.0001	1.0028	1.0234	1.113	1.367	1.898	2.3
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0007	1.018	1.182	1.920	2.7

Table 3

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY SEPARATE CHAINING

Bucket	Load factor, α									
size, b	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0500	1.1000	1.1500	1.2000	1.2500	1.3000	1.350	1.400	1.450	1.5
2	1.0063	1.0242	1.0520	1.0883	1.1321	1.1823	1.238	1.299	1.364	1.4
3	1.0010	1.0071	1.0216	1.0458	1.0806	1.1259	1.181	1.246	1.319	1.4
4	1.0002	1.0023	1.0097	1.0257	1.0527	1.0922	1.145	1.211	1.290	1.3
5	1.0000	1.0008	1.0046	1.0151	1.0358	1.0699	1.119	1.186	1.286	1.3
10	1.0000	1.0000	1.0002	1.0015	1.0070	1.0226	1.056	1.115	1.206	1.3
20	1.0000	1.0000	1.0000	1.0000	1.0005	1.0038	1.018	1.059	1.150	1.2
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.015	1.083	1.2

B) *Chaining with coalescing lists.* Instead of providing a separate overflow area, we can adapt Algorithm C to external files. A doubly linked list of available space can be used which links together each bucket that is not yet full. Under this scheme, every bucket contains a count of how many record positions are empty, and the bucket is removed from the doubly linked list only when this count becomes zero. A 'roving pointer' can be used to distribute overflows (cf. exercise 2.5–6), so that different lists tend to use different overflow buckets. It may be a good idea to have separate available-space lists for the buckets on each cylinder of a disk file.

This method has not yet been analyzed, but it should prove to be quite useful.

C) *Open addressing.* We can also do without links, using an "open" method. Linear probing is probably better than random probing when we consider external searching, because the increment c can often be chosen so that it minimizes latency delays between consecutive accesses. The approximate theoretical model of linear probing which was worked out above can be generalized to account for the influence of buckets, and it shows that linear probing is indeed satisfactory unless the table has gotten very full. For example, see Table 4; when the load factor is 90 percent and the bucket size is 50, the average

Table 4

AVERAGE ACCESSES IN A SUCCESSFUL SEARCH BY LINEAR PROBING

Bucket	Load factor, α									
size, b	10%	20%	30%	40%	50%	60%	70%	80%	90%	95%
1	1.0556	1.1250	1.2143	1.3333	1.5000	1.7500	2.167	3.000	5.500	10.5
2	1.0062	1.0242	1.0553	1.1033	1.1767	1.2930	1.494	1.903	3.147	5.6
3	1.0009	1.0066	1.0201	1.0450	1.0872	1.1584	1.286	1.554	2.378	4.0
4	1.0001	1.0021	1.0085	1.0227	1.0497	1.0984	1.190	1.386	2.000	3.2
5	1.0000	1.0007	1.0039	1.0124	1.0307	1.0661	1.136	1.289	1.777	2.7
10	1.0000	1.0000	1.0001	1.0011	1.0047	1.0154	1.042	1.110	1.345	1.8
20	1.0000	1.0000	1.0000	1.0000	1.0003	1.0020	1.010	1.036	1.144	1.4
50	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.001	1.005	1.040	1.1

number of accesses in a successful search is only 1.04. This is actually *better* than the 1.08 accesses required by the chaining method (A) with the same bucket size!

The analysis of methods (A) and (C) involves some very interesting mathematics; we shall merely summarize the results here, since the details are worked out in exercises 49 and 55. The formulas involve two functions strongly related to the Q-functions of Theorem K, namely

$$R(\alpha, n) = \frac{n}{n+1} + \frac{n^2\alpha}{(n+1)(n+2)} + \frac{n^3\alpha^2}{(n+1)(n+2)(n+3)} + \cdots, \qquad (55)$$

GN 005775

and

$$t_n(\alpha) = e^{-n\alpha}\left(\frac{(\alpha n)^n}{(n+1)!} + 2\,\frac{(\alpha n)^{n+1}}{(n+2)!} + 3\,\frac{(\alpha n)^{n+2}}{(n+3)!} + \cdots\right)$$
$$= \frac{e^{-n\alpha}n^n\alpha^n}{n!}\,\bigl(1 - (1-\alpha)R(\alpha, n)\bigr). \tag{56}$$

In terms of these functions, the average number of accesses made by the chaining method (A) in an unsuccessful search is

$$C'_N = 1 + \alpha b t_b(\alpha) + O\left(\frac{1}{M}\right) \tag{57}$$

as $M, N \to \infty$, and the corresponding number in a successful search is

$$C_N = 1 + \bigl(1 - \tfrac{1}{2}b(1-\alpha)\bigr)t_b(\alpha) + \frac{e^{-b\alpha}b^b\alpha^b}{2b!}\,R(\alpha, b) + O\left(\frac{1}{M}\right). \tag{58}$$

The limiting values of these formulas are the quantities shown in Tables 2 and 3.

Since chaining method (A) requires a separate overflow area, we need to estimate how many overflows will occur. The average number of overflows will be $M(C'_N - 1) = Nt_b(\alpha)$, since $C'_N - 1$ is the average number of overflows in any given list. Therefore Table 2 can be used to deduce the amount of overflow space required. For fixed α, the standard deviation of the total number of overflows will be roughly proportional to $\sqrt{M}$ as $M \to \infty$.

Asymptotic values for C'_N and C_N appear in exercise 53, but the approximations aren't very good when b is small or α is large; fortunately the series for $R(\alpha, n)$ converges rather rapidly even when α is large, so the formulas can be evaluated exactly without much difficulty. The maximum values occur for $\alpha = 1$, when

$$\max C'_N = 1 + \frac{e^{-b}b^{b+1}}{b!} = \sqrt{\frac{b}{2\pi}} + 1 + O(b^{-1/2}), \tag{59}$$

$$\max C_N = 1 + \frac{e^{-b}b^b}{2b!}\,\bigl(R(b) + 1\bigr) = \frac{5}{4} + \sqrt{\frac{2}{9\pi b}} + O(b^{-1}), \tag{60}$$

as $b \to \infty$, by Stirling's approximation and the analysis of the function $R(n) = R(1, n) - 1$ in Section 1.2.11.3.

The average number of access in a successful external search with *linear* probing has the remarkably simple expression

$$C_N \approx 1 + t_b(\alpha) + t_{2b}(\alpha) + t_{3b}(\alpha) + \cdots, \tag{61}$$

which can be understood as follows: The average total number of accesses to look up all N keys is NC_N, and this is $N + T_1 + T_2 + \cdots$, where T_k is the average number of keys which require more than k accesses. Theorem P says that we can enter the keys in any order without affecting C_N, and it follows that T_k is the average number of overflow records that would occur in the

GN 005776

chaining method if we had M/k buckets of size kb, namely $Nt_{kb}(\alpha)$ by what we said above. Further justification of Eq. (61) appears in exercise 55.

An excellent discussion of practical considerations involved in the design of external scatter tables has been given by Charles A. Olson, *Proc. ACM Nat'l Conf.* **24** (1969), 539–549. He includes several worked examples and points out that the number of overflow records will increase substantially if the file is subject to frequent insertion/deletion activity without relocating records; and he presents an analysis of this situation which was obtained jointly with J. A. de Peyster.

Comparison of the methods. We have now studied a large number of techniques for searching; how can we select the right one for a given application? It is difficult to summarize in a few words all the relevant details of the "trade-offs" involved in the choice of a search method, but the following things seem to be of primary importance with respect to the speed of searching and the requisite storage space.

Figure 44 summarizes the analyses of this section, showing that the various methods for collision resolution lead to different numbers of probes. But this does not tell the whole story, since the time per probe varies in different methods, and the latter variation has a noticeable effect on the running time (as we have seen in Fig. 42). Linear probing accesses the table more frequently than the other methods shown in Fig. 44, but it has the advantage of simplicity. Furthermore, even linear probing isn't terribly bad: when the table is 90 percent full, Algorithm L requires an average of less than 5.5 probes to locate a random item in the table. (However, the average number of probes needed to insert a *new* item with Algorithm L is 50.5, with a 90-percent-full table.)

Figure 44 shows that the chaining methods are quite economical with respect to the number of probes, but the extra memory space needed for link fields sometimes makes open addressing more attractive for small records. For example, if we have to choose between a chained scatter table of capacity 500 and an open scatter table of capacity 1000, the latter is clearly preferable, since it allows efficient searching when 500 records are present and it is capable of absorbing twice as much data. On the other hand, sometimes the record size and format will allow space for link fields at virtually no extra cost. (Cf. exercise 65.)

How do hash methods compare with the other search strategies we have studied in this chapter? From the standpoint of speed we can argue that they are better, when the number of records is large, because the average search time for a hash method stays bounded as $N \to \infty$ if we stipulate that the table never gets too full. For example, Program L will take only about 55 units of time for a successful search when the table is 90 percent full; this beats the fastest MIX binary search routine we have seen (exercise 6.2.1–24) when N is greater than 600 or so, at the cost of only 11 percent in storage space. Moreover the binary search is suitable only for fixed tables, while a scatter table allows efficient insertions.

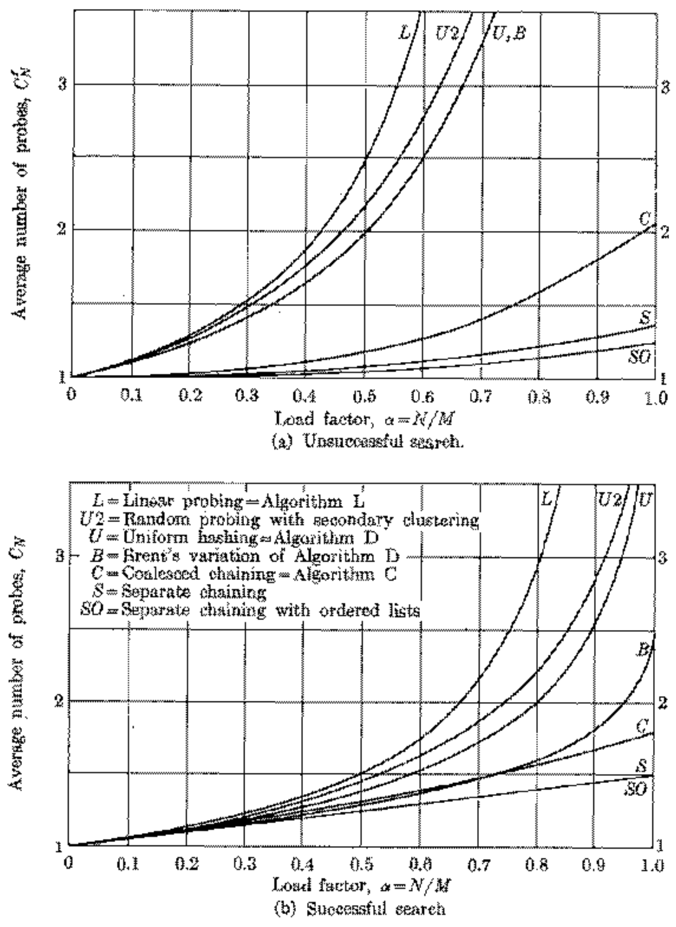


Fig. 44. Comparison of collision resolution methods: Limiting values of the average number of probes as $M \to \infty$.

We can also compare Program L to the tree-oriented search methods which allow dynamic insertions. Program L with a 90-percent-full table is faster than Program 6.2.2T when N is greater than about 90, and faster than Program 6.3D (exercise 6.3–9) when N is greater than about 75.

Only one search method in this chapter is efficient for successful searching with virtually no storage overhead, namely Brent's variation of Algorithm D. His method allows us to put N records into a table of size $M = N + 1$, and to find any record in about $2\frac{1}{2}$ probes on the average. No extra space for link fields, etc., is needed; however, an unsuccessful search will be very slow, requiring about $\frac{1}{2}N$ probes.

Thus hashing has several advantages. On the other hand, there are three important respects in which scatter table searching is inferior to other methods we have discussed:

a) After an unsuccessful search in a scatter table, we know only that the desired key is not present. Search methods based on comparisons always yield more information, making it possible to find the largest key $\leq K$ and/or the smallest key $\geq K$; this is important in many applications (e.g., for interpolation of function values from a stored table). It is also possible to use comparison-based algorithms to locate all keys which lie *between* two given values K and K'. Furthermore the tree search algorithms of Section 6.2 make it easy to traverse the contents of a table in ascending order, without sorting it separately, and this is occasionally desirable.

b) The storage allocation for scatter tables is often somewhat difficult; we have to dedicate a certain area of the memory for use as the hash table, and it may not be obvious how much space should be allotted. If we provide too much memory, we may be wasting storage at the expense of other lists or other computer users; but if we don't provide enough room, the table will overflow. When a scatter table overflows, it is probably best to "rehash" it, i.e., to allocate a larger space and to change the hash function, reinserting every record into the larger table. F. R. A. Hopgood [*Comp. Bulletin* **11** (1968), 297–300] has suggested rehashing the table when it becomes α_0 percent full, replacing M by d_0M; suitable choices of these parameters α_0 and d_0 can be made by using the analyses above and characteristics of the data, so that the critical point at which it becomes cheaper to rehash can be determined. (Note that the method of chaining does not lead to any troublesome overflows, so it requires no rehashing; but the search time is proportional to N when M is fixed and N gets large.) By contrast, the tree search and insertion algorithms require no such painful rehashing; the trees grow no larger than necessary. In a virtual memory environment we probably ought to use tree search or digital tree search, instead of creating a large scatter table that requires bringing in a new page nearly every time we hash a key.

c) Finally, we need a great deal of faith in probability theory when we use hashing methods, since they are efficient only on the average, while their worst case is terrible! As in the case of random number generators, we are never completely sure that a hash function will perform properly when it is applied to a new set of data. Therefore scatter storage would be inappropriate for certain real-time applications such as air traffic control, where people's lives are at stake; the balanced tree algorithms of Sections 6.2.3 and 6.2.4 are much safer, since they provide guaranteed upper bounds on the search time.

History. The idea of hashing appears to have been originated by H. P. Luhn, who wrote an internal IBM memorandum suggesting the use of chaining, in

January 1953; this was also among the first applications of linked linear lists. He pointed out the desirability of using buckets containing more than one element, for external searching. Shortly afterwards, A. D. Lin carried Luhn's analysis a little further, and suggested a technique for handling overflows that used "degenerative addresses"; e.g., the overflows from primary bucket 2748 are put in secondary bucket 274; overflows from that bucket go to tertiary bucket 27, etc., assuming the presence of 10000 primary buckets, 1000 secondary buckets, 100 tertiary buckets, etc. The hash functions originally suggested by Luhn were digital in character, e.g., adding adjacent pairs of key digits mod 10, so that 31415926 would be compressed to 4548.

At about the same time the idea of hashing occurred independently to another group of IBMers: Gene M. Amdahl, Elaine M. Boehme, N. Rochester, and Arthur L. Samuel, who were building an assembly program for the IBM 701. In order to handle the collision problem, Amdahl originated the idea of open addressing with linear probing.

Hash coding was first described in the open literature by Arnold I. Dumey, *Computers and Automation* 5, 12 (December 1956), 6–9. He was the first to mention the idea of dividing by a prime number and using the remainder as the hash address. Dumey's interesting article mentions chaining but not open addressing. A. P. Ershov of Russia independently discovered linear open addressing in 1957 [*Doklady Akad. Nauk SSSR* **118** (1958), 427–430]; he published empirical results about the number of probes, conjecturing correctly that the average number of probes per successful search is <2 when $N/M < 2/3$.

A classic article by W. W. Peterson, *IBM J. Research & Development* **1** (1957), 130–146, was the first major paper dealing with the problem of searching in large files. Peterson defined open addressing in general, analyzed the performance of uniform hashing, and gave numerous empirical statistics about the behavior of linear open addressing with various bucket sizes, noting the degradation in performance that occurred when items were deleted. Another comprehensive survey of the subject was published six years later by Werner Buchholz [*IBM Systems J.* **2** (1963), 86–111], who gave an especially good discussion of hash functions.

Up to this time linear probing was the only type of open addressing scheme that had appeared in the literature, but another scheme based on repeated random probing by independent hash functions had independently been developed by several people (see exercise 48). During the next few years hashing became very widely used, but hardly anything more was published about it. Then Robert Morris wrote a very influential survey of the subject [*CACM* **11** (1968), 38–44], in which he introduced the idea of random probing (with secondary clustering). Morris's paper touched off a flurry of activity which culminated in Algorithm D and its refinements.

It is interesting to note that the word "hashing" apparently never appeared in print, with its present meaning, until Morris's article was published in 1968, although it had already become common jargon in several parts of the world by that time. The only previous occurrence of the word among approximately 60 relevant documents studied by the author as this section was being written was in an unpublished memorandum written by W. W. Peterson in 1961. Somehow the verb "to hash" magically became standard terminology for key transformation during the mid-1960's, yet nobody was rash enough to use such an undignified word publicly until 1968!

EXERCISES

1. [*20*] When the instruction 9H in Table 1 is reached, how small and how large can the contents of rI1 possibly be, assuming that bytes 1, 2, 3 of K contain alphabetic character codes less than 30?

2. [*20*] Find a reasonably common English word not in Table 1 that could be added to that table without changing the program.

3. [*23*] Explain why no program beginning with the five instructions

```
          LD1  K(1:1)    or    LD1N  K(1:1)
          LD2  K(2:2)    or    LD2N  K(2:2)
                    INC1  a,2
                    LD2   K(3:3)
                    J2Z   9F
```

could be used in place of the more complicated program in Table 1, for any constant a, since unique addresses would not be produced for the given keys.

4. [*M30*] How many people should be invited to a party in order to make it likely that there are *three* with the same birthday?

5. [*15*] Mr. B. C. Dull was writing a FORTRAN compiler using a decimal MIX computer, and he needed a symbol table to keep track of the names of variables in the FORTRAN program being compiled. These names were restricted to be ten or less characters in length. He decided to use a scatter table with $M = 100$, and to use the fast hash function $h(K) =$ leftmost byte of K. Was this a good idea?

6. [*15*] Would it be wise to change the first two instructions of (3) to LDA K; ENTX 0?

7. [*HM30*] (*Polynomial hashing.*) The purpose of this exercise is to consider the construction of polynomials $P(x)$ such as (10), which convert n-bit keys into m-bit addresses, such that distinct keys differing in t or fewer bits will hash to different addresses. Given n and $t \leq n$, and given an integer k such that n divides $2^k - 1$, we shall construct a polynomial whose degree m is a function of n, t, and k. (Usually n is increased, if necessary, so that k can be chosen to be reasonably small.)

Let S be the smallest set of integers such that $\{1, 2, \ldots, t\} \subseteq S$, and $(2j) \bmod n \in S$ for all $j \in S$. For example, when $n = 15$, $k = 4$, $t = 6$, we have $S = \{1, 2, 3, 4, 5, 6, 8, 10, 12, 9\}$. We now define the polynomial $P(x) = \prod_{j \in S}(x - \alpha^j)$, where α is an element of order n in the finite field $GF(2^k)$, and where the coefficients of $P(x)$ are computed in this field. The degree m of $P(x)$ is the number of elements of S. Since α^{2j} is a root of $P(x)$ whenever α^j is a root, it follows that the coefficients p_i of $P(x)$ satisfy $p_i^2 = p_i$, so they are all 0 or 1.

Prove that if $R(x) = r_{n-1}x^{n-1} + \cdots + r_1x + r_0$ is any nonzero polynomial modulo 2, with at most t nonzero coefficients, then $R(x)$ is not a multiple of $P(x)$, modulo 2. [It follows that the corresponding hash function behaves as advertised.]

8. [*M34*] (*The three-distance theorem.*) Let θ be an irrational number between 0 and 1, whose regular continued fraction representation in the notation of Section 4.5.3 is $\theta = /\!/a_1, a_2, a_3, \ldots /\!/$. Let $q_0 = 0$, $p_0 = 1$, $q_1 = 1$, $p_1 = 0$, and $q_{k+1} = a_kq_k + q_{k-1}$, $p_{k+1} = a_kp_k + p_{k-1}$, for $k \geq 1$. Let $\{x\}$ denote $x \bmod 1 = x - \lfloor x \rfloor$, and let $\{x\}^+$ denote $x - \lceil x \rceil + 1$. As the points $\{\theta\}$, $\{2\theta\}$, $\{3\theta\}$, ... are successively inserted into the interval $[0, 1]$, let the line segments be numbered as they appear in such a way that the first segment of a given length is number 0, the next is number 1, etc. Prove that the following statements are all true: Interval number s of length $\{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \leq r < a_k$ and k is even and $0 \leq s < q_k$, has left endpoint $\{s\theta\}$ and right endpoint $\{(s+t)\theta\}^+$. Interval number s of length $1 - \{t\theta\}$, where $t = rq_k + q_{k-1}$ and $0 \leq r < a_k$ and k is odd and $0 \leq s < q_k$, has left endpoint $\{(s+t)\theta\}$ and right endpoint $\{s\theta\}^+$. Every positive integer n can be uniquely represented as $n = rq_k + q_{k-1} + s$ for some $k \geq 1$, $1 \leq r \leq a_k$, $0 \leq s < q_k$. In terms of this representation, just before the point $\{n\theta\}$ is inserted the n intervals present are

the first s intervals (numbered $0, \ldots, s-1$) of length $\{(-1)^k(rq_k + q_{k-1})\theta\}$;
the first $n - q_k$ intervals (numbered $0, \ldots, n - q_k - 1$) of length $\{(-1)^kq_k\theta\}$;
the last $q_k - s$ intervals (numbered $s, \ldots, q_k - 1$) of length
$\{(-1)^k((r-1)q_k + q_{k-1})\theta\}$.

The operation of inserting $\{n\theta\}$ removes interval number s of the latter type and converts it into interval number s of the first type, number $n - q_k$ of the second type.

9. [*M30*] When we successively insert the points $\{\theta\}$, $\{2\theta\}$, ... into the interval $[0, 1]$, Theorem S asserts that each new point always breaks up one of the largest remaining intervals. If the interval $[a, c]$ is thereby broken into two parts $[a, b]$, $[b, c]$, we may call it a *bad break* if one of these parts is more than twice as long as the other, i.e. if $b - a > 2(c - b)$ or $c - b > 2(b - a)$.

Prove that bad breaks will occur for some $\{n\theta\}$ unless $\theta \bmod 1 = \phi^{-1}$ or ϕ^{-2}; and the latter values of θ *never* produce bad breaks.

10. [*M48*] (R. L. Graham.) Prove or disprove the following *3d distance conjecture:* If $\theta, \alpha_1, \ldots, \alpha_d$ are real numbers with $\alpha_1 = 0$, and if $n_1, \ldots, n_d$ are positive integers, and if the points $\{n\theta + \alpha_i\}$ are inserted into the interval $[0, 1]$ for $0 \leq n < n_i$, $1 \leq i \leq d$, the resulting $n_1 + \cdots + n_d$ (possibly empty) intervals have at most $3d$ different lengths.

11. [*16*] Successful searches are usually more frequent than unsuccessful ones. Would it therefore be a good idea to interchange lines 12–13 of Program C with lines 10–11?

▸ **12.** [*21*] Show that Program C can be rewritten so that there is only one conditional jump instruction in the inner loop. Compare the running time of the modified program with the original.

▸ **13.** [*24*] (*Abbreviated keys.*) Let $h(K)$ be a hash function, and let $q(K)$ be a function of K such that K can be determined once $h(K)$ and $q(K)$ are given. For example, in division hashing we may let $h(K) = K \bmod M$ and $q(K) = \lfloor K/M \rfloor$; in multiplicative hashing we may let $h(K)$ be the leading bits of $(AK/w) \bmod 1$, and $q(K)$ can be the other bits.

Show that when chaining is used without overlapping lists, we need only store $q(K)$ instead of K in each record. (This almost saves the space needed for the link fields.) Modify Algorithm C so that it allows such abbreviated keys by avoiding overlapping lists, yet uses no auxiliary storage locations for "overflow" records.

14. [*24*] (E. W. Elcock.) Show that it is possible to let a large scatter table *share memory* with any number of other linked lists. Let every word of the list area have a 2-bit TAG field, with the following interpretation:

TAG(P) = 0 indicates a word in the list of available space; LINK(P) points to the next available word.

TAG(P) = 1 indicates any word in use that is not part of the scatter table; the other fields of this word may have any desired format.

TAG(P) = 2 indicates a word in the scatter table; LINK(P) points to another word. Whenever we are processing a list that is not part of the scatter table and we access a word with TAG(P) = 2, we are supposed to set P ← LINK(P) until reaching a word with TAG(P) $\leq$ 1. (For efficiency we might also then change one of the prior links so that it will not be necessary to skip over the same scatter table entries again and again.)

Show how to define suitable algorithms for inserting and retrieving keys from such a combined table, assuming that words with TAG(P) = 2 also have another link field AUX(P).

15. [*16*] Why is it a good idea for Algorithm L and Algorithm D to signal overflow when N = $M - 1$ instead of when N = M?

16. [*10*] Program L says that K should not be zero. But doesn't it actually work even when K is zero?

17. [*15*] Why not simply define $h_2(K) = h_1(K)$ in (25), when $h_1(K) \neq 0$?

▸ **18.** [*21*] Is (31) better or worse than (30), as a substitute for lines 10–13 of Program D? Give your answer on the basis of the average values of A, $S1$, and C.

19. [*40*] Empirically test the effect of restricting the range of $h_2(K)$ in Algorithm D, so that (a) $1 \leq h_2(K) \leq r$ for $r = 1, 2, 3, \ldots, 10$; (b) $1 \leq h_2(K) \leq \rho M$ for $\rho = \frac{1}{10}, \frac{2}{10}, \ldots, \frac{9}{10}$.

20. [*M25*] (R. Krutar.) Change Algorithm D as follows, avoiding the hash function $h_2(K)$: In step D3, set $c \leftarrow 0$; and at the beginning of step D4, set $c \leftarrow c + 1$. Prove that if $M = 2^m$, the corresponding probe sequence $h_1(K)$, $(h_1(K) - 1) \bmod M, \ldots, (h_1(K) - \binom{M}{2}) \bmod M$ will be a permutation of $\{0, 1, \ldots, M - 1\}$. When this method is programmed for MIX, how does it compare with the three programs considered in Fig. 42, assuming that the behavior is like random probing with secondary clustering?

▸ **21.** [*20*] Suppose that we wish to delete a record from a table constructed by Algorithm D, marking it "deleted" as suggested in the text. Should we also decrease the variable N which is used to govern Algorithm D?

22. [*27*] Prove that Algorithm R leaves the table exactly as it would have been if KEY[i] had never been inserted in the first place.

▸ **23.** [*23*] Design an algorithm analogous to Algorithm R, for deleting entries from a chained scatter table that has been constructed by Algorithm C.

GN 005783

24. [*M20*] Suppose that the set of all possible keys that can occur has MP elements, where exactly P keys hash to a given address. (In practical cases, P is very large; e.g. if the keys are arbitrary 10-digit numbers and if $M = 10^3$, we have $P = 10^7$.) Assume that $M \geq 7$ and $N = 7$. If seven distinct keys are selected at random from the set of all possible keys, what is the exact probability that the hash sequence 1 2 6 2 1 6 1 will be obtained (i.e., that $h(K_1) = 1$, $h(K_2) = 2, \ldots, h(K_7) = 1$), as a function of M and P?

25. [*M19*] Explain why Eq. (39) is true.

26. [*M20*] How many hash sequences $a_1\,a_2 \ldots a_9$ yield the pattern of occupied cells (21), using linear probing?

27. [*M27*] Complete the proof of Theorem K. (*Hint:* Let

$$s(n, x, y) = \sum_{k\geq 0} \binom{n}{k} (x+k)^{k+1}(y-k)^{n-k-1}(y-n);$$

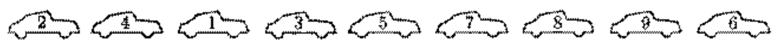
use Abel's binomial theorem, Eq. 1.2.6–16, to prove that $s(n, x, y) = x(x+y)^n + ns(n-1, x+1, y-1)$.)

28. [*M30*] In the old days when computers were much slower than they are now, it was possible to watch the lights flashing and see how fast Algorithm L was running. When the table began to fill up, some entries would be processed very quickly, while others took a great deal of time.

This experience suggests that the standard deviation of C'_N is rather high, when linear probing is used. Find a formula which expresses the variance in terms of the Q_r functions defined in Theorem K, and estimate the variance when $N = \alpha M$ as $M \to \infty$.

29. [*M21*] (*The parking problem.*) A certain one-way street has m parking spaces in a row, numbered 1 through m. A man and his dozing wife drive by, and suddenly she wakes up and orders him to park immediately. He dutifully parks at the first available space; but if there are no places left that he can get to without backing up (i.e., if his wife awoke when the car approached space k, but spaces $k, k+1, \ldots, m$ are all full), he expresses his regrets and drives on.

Suppose, in fact, that this happens for n different cars, where the jth wife wakes up just in time to park at space a_j. In how many of the sequences $a_1 \ldots a_n$ will all of the cars get safely parked, assuming that the street is initially empty and that nobody leaves after parking? For example, when $m = n = 9$ and $a_1 \ldots a_9 = 3\ 1\ 4\ 1\ 5\ 9\ 2\ 6\ 5$, the cars get parked as follows:



[*Hint:* Use the analysis of linear probing.]

30. [*M28*] (John Riordan.) When $n = m$ in the parking problem of exercise 29, show that all cars get parked if and only if there exists a permutation $p_1\,p_2 \ldots p_n$ of $\{1, 2, \ldots, n\}$ such that $a_j \leq p_j$ for all j.

31. [*M40*] When $n = m$ in the parking problem of exercise 29, the number of solutions turns out to be $(n+1)^{n-1}$; and from exercise 2.3.4.4–22 we know this is the same as the number of free trees on $n+1$ labeled vertices! Find an interesting connection between parking sequences and trees.

32. [*M26*] Prove that the system of equations (44) has a unique solution $(c_0, c_1, \ldots, c_{M-1})$, whenever $b_0, b_1, \ldots, b_{M-1}$ are nonnegative integers whose sum is less than M. Design an algorithm which finds that solution.

▸ **33.** [*M23*] Explain why (51) is only an approximation to the true average number of probes made by Algorithm L. [What was there about the derivation of (51) that wasn't rigorously exact?]

▸ **34.** [*M22*] The purpose of this exercise is to investigate the average number of probes in a chained scatter table when the lists are kept separate as in Fig. 38. (a) What is P_{Nk}, the probability that a given list has length k, when the M^N hash sequences (35) are equally likely? (b) Find the generating function $P_N(z) = \sum_{k\geq 0} P_{Nk}z^k$. (c) Express the average number of probes for successful and unsuccessful search in terms of this generating function. (Assume that an unsuccessful search in a list of length k requires $k + \delta_{k0}$ probes.)

35. [*M21*] Continuing exercise 34, what is the average number of probes in an unsuccessful search when the individual lists are kept in order by their key values?

36. [*M22*] Find the variance of (18), the number of probes in separate chaining when the search is unsuccessful.

▸ **37.** [*M29*] Find the variance of (19), the number of probes in separate chaining when the search is successful.

38. [*M32*] (*Tree hashing.*) A clever programmer might try to use binary search trees instead of linear lists in the chaining method, thereby combining Algorithm 6.2.2T with hashing. Analyze the average number of probes that would be required by this compound algorithm, for both successful and unsuccessful searches. [*Hint:* Cf. Eq. 5.2.1–11.]

39. [*M30*] The purpose of this exercise is to analyze the average number of probes in Algorithm C (chaining with coalescing lists). Let $c(k_1, k_2, k_3, \ldots)$ be the number of hash sequences (35) that cause Algorithm C to form exactly k_1 lists of length 1, k_2 of length 2, etc., when $k_1 + 2k_2 + 3k_3 + \cdots = N$. Find a recurrence relation which defines these numbers $c(k_1, k_2, k_3, \ldots)$, and use it to determine a simple formula for the sum

$$S_N = \sum_{\substack{j\geq 1 \\ k_1+2k_2+\cdots=N}} \binom{j}{2} k_j c(k_1, k_2, \ldots).$$

How is S_N related to the number of probes in an unsuccessful search by Algorithm C?

40. [*M33*] Find the variance of (15), the number of probes used by Algorithm C in an unsuccessful search.

41. [*M40*] Analyze T_N, the average number of times R is decreased by 1 when the $(N + 1)$st item is being inserted by Algorithm C.

▸ **42.** [*M20*] Derive (17).

43. [*M42*] Analyze a modification of Algorithm C that uses a table of size $M' > M$. Only the first M locations are used for hashing, so the first $M' - M$ empty nodes found in step C5 will be in the extra locations of the table. For fixed M', what choice of M in the range $1 \leq M \leq M'$ leads to the best performance?

44. [*M43*] (*Random probing with secondary clustering.*) The object of this exercise is to determine the expected number of probes in the open addressing scheme with probe sequence

$$h(K),\quad (h(K)+p_1) \bmod M,\quad (h(K)+p_2) \bmod M,\quad \ldots,\quad (h(K)+p_{M-1}) \bmod M,$$

where $p_1\,p_2\ldots p_{M-1}$ is a randomly chosen permutation of $\{1, 2, \ldots, M-1\}$ that depends on $h(K)$. In other words, all keys with the same value of $h(K)$ follow the same probe sequence, and the $(M-1)!^M$ possible choices of M probe sequences with this property are equally likely.

This situation can be accurately modeled by the following experimental procedure performed on an initially empty linear array of size m. Do the following operation n times:

> With probability p, occupy the leftmost empty position. Otherwise (i.e., with probability $q = 1 - p$), select any table position except the one at the extreme left, with each of these $m - 1$ positions equally likely. If the selected position is empty, occupy it; otherwise select *any* empty position (including the leftmost) and occupy it, considering each of the empty positions equally likely.

For example when $m = 5$ and $n = 3$, the array configuration after the above experiment will be (occupied, occupied, empty, occupied, empty) with probability

$$\tfrac{7}{192}qqq + \tfrac{1}{6}pqq + \tfrac{1}{6}qpq + \tfrac{11}{64}qqp + \tfrac{1}{3}ppq + \tfrac{1}{4}pqp + \tfrac{1}{4}qpp.$$

(This procedure corresponds to random probing with secondary clustering, when $p = 1/m$, since we can renumber the table entries so that a particular probe sequence is $0, 1, 2, \ldots$ and all the others are random.)

Find a formula for the average number of occupied positions at the left of the array (i.e., 2 in the above example). Also find the asymptotic value of this quantity when $p = 1/m$, $n = \alpha(m + 1)$, and $m \to \infty$.

45. [*M48*] Solve the analog of exercise 44 with *tertiary clustering*, when the probe sequence begins $h_1(K)$, $\bigl(h_1(K) + h_2(K)\bigr) \bmod M$, and the succeeding probes are randomly chosen depending only on $h_1(K)$ and $h_2(K)$. (Thus the $(M-2)!^{M(M-1)}$ possible choices of $M(M-1)$ probe sequences with this property are considered to be equally likely.) Is this procedure asymptotically equivalent to uniform probing?

46. [*M42*] Determine C'_N and C_N for the open addressing method which uses the probe sequence

$$h(K),\ 0,\ 1,\ \ldots,\ h(K)-1,\ h(K)+1,\ \ldots,\ M-1.$$

47. [*M25*] Find the average number of probes needed by open addressing when the probe sequence is

$$h(K),\ h(K)-1,\ h(K)+1,\ h(K)-2,\ h(K)+2,\ \ldots.$$

This probe sequence was once suggested because all the distances between consecutive probes are distinct when M is even. [*Hint:* Find the trick and this problem is easy.]

▶ **48.** [*M21*] Analyze the open addressing method that probes locations $h_1(K)$, $h_2(K)$, $h_3(K)$, …, given an infinite sequence of mutually independent random hash functions

$h_n(K)$. Note that it is possible to probe the same location twice, e.g. if $h_1(K) = h_2(K)$, but this is rather unlikely.

49. [*HM24*] Generalizing exercise 34 to the case of b records per bucket, determine the average number of probes (i.e., file accesses) C_N and C'_N, for chaining with separate lists, assuming that a list containing k elements requires $\max(1, k - b + 1)$ probes in an unsuccessful search. Instead of using the exact probability P_{Nk} as in exercise 34, use the *Poisson approximation*

$$\binom{N}{k}\left(\frac{1}{M}\right)^k\left(1-\frac{1}{M}\right)^{N-k} = \frac{N}{M}\,\frac{N-1}{M}\cdots\frac{N-k+1}{M}\left(1-\frac{1}{M}\right)^N\left(1-\frac{1}{M}\right)^{-k}\frac{1}{k!}$$
$$= \frac{e^{-\rho}\rho^k}{k!}\left(1+O(k^2/M)\right),$$

which is valid for $N = \rho M$ and $k \leq \sqrt{M}$ as $M \to \infty$.

50. [*M20*] Show that $Q_1(M, N) = M - (M - N - 1)Q_0(M, N)$, in the notation of (42). [*Hint:* Prove first that $Q_1(M, N) = (N + 1)Q_0(M, N) - NQ_0(M, N - 1)$.]

51. [*HM16*] Express the function $R(\alpha, n)$ defined in (55) in terms of the function Q_0 defined in (42).

52. [*HM20*] Prove that $Q_0(M, N) = \int_0^\infty e^{-t}(1 + t/M)^N\,dt$.

53. [*HM20*] Prove that the function $R(\alpha, n)$ can be expressed in terms of the incomplete gamma function, and use the result of exercise 1.2.11.3–9 to find the asymptotic value of $R(\alpha, n)$ to $O(n^{-2})$ as $n \to \infty$, for fixed $\alpha < 1$.

54. [*40*] Experiment with the behavior of Algorithm C when it has been adapted to external searching as described in the text.

55. [*HM43*] Generalize the Schay-Spruth model, discussed after Theorem P, to the case of M buckets of size b. Prove that $C(z)$ is equal to $Q(z)/(B(z) - z^b)$, where $Q(z)$ is polynomial of degree b and $Q(1) = 0$. Show that the average number of probes is

$$1+\frac{M}{N}C'(1) = 1+\frac{1}{b}\left(\frac{1}{1-q_1}+\cdots+\frac{1}{1-q_{b-1}}-\frac{1}{2}\,\frac{B''(1)-b(b-1)}{B'(1)-b}\right),$$

where $q_1, \ldots, q_{b-1}$ are the roots of $Q(z)/(z - 1)$. Replacing the binomial probability distribution $B(z)$ by the Poisson approximation $P(z) = e^{b\alpha(z-1)}$, where $\alpha = N/Mb$, and using Lagrange's inversion formula (cf. Eq. 2.3.4.4–9 and exercise 4.7–8), reduce your answer to Eq. (61).

56. [*M48*] Generalize Theorem K, obtaining an exact analysis of linear probing with buckets of size b.

57. [*M47*] Does the uniform assignment of probabilities to probe sequences give the minimum value of C_N, over all open addressing methods?

58. [*M21*] (S. C. Johnson.) Find ten permutations on $\{0, 1, 2, 3, 4\}$ that are equivalent to uniform hashing in the sense of Theorem U.

59. [*M25*] Prove that if an assignment of probabilities to permutations is equivalent to uniform hashing, in the sense of Theorem U, the number of permutations with nonzero probabilities exceeds M^a for any fixed exponent a, when M is sufficiently large.

60. [*M48*] Let us say that an open addressing scheme involves *single hashing* if it uses exactly M probe sequences, one beginning with each possible value of $h(K)$, each of which occurs with probability $1/M$.

Are the best single-hashing schemes (in the sense of minimum C_N) asymptotically better than the random ones analyzed in exercise 44?

61. [*M46*] Is the method analyzed in exercise 46 the worst possible single-hashing scheme? (Cf. exercise 60.)

62. [*M49*] How good can a single-hashing scheme be when the increments $p_1\,p_2 \ldots p_{M-1}$ in the notation of exercise 44 are fixed for all K? (Examples of such methods are linear probing and the sequences considered in exercises 20 and 47.)

63. [*M25*] If repeated random insertions and deletions are made in a scatter table, how many independent insertions are needed on the average before all M locations have become occupied at one time or another?

64. [*M46*] Analyze the expected behavior of Algorithm R. How many times will step R4 be performed, on the average?

▸**65.** [*20*] (*Variable-length keys.*) Many applications of scatter tables deal with keys that can be any number of characters long. In such cases we can't simply store the key in the table as in the programs of this section. What would be a good way to deal with variable-length keys in a scatter table on the MIX computer?

66. [*25*] (Ole Amble.) Is it possible to insert keys into an open hash table making use also of their numerical or alphabetic order, so that a search with Algorithm L or Algorithm D is known to be unsuccessful whenever a key *smaller* than the search argument is encountered?

HASH, x. There is no definition for this word—
nobody knows what hash is.
—AMBROSE BIERCE (*The Devil's Dictionary*, 1906)

GN 005788