

Figure 94 shows Hollerith's original battery-driven apparatus; of chief interest to us is the "sorting box" at the right, which has been opened to show half of the 26 inner compartments. The operator would insert a $6\frac{5}{8}'' \times 3\frac{1}{4}''$ punched card into the "press" and lower the handle; this caused spring-actuated pins in the upper plate to make contact with pools of mercury in the lower plate, wherever a hole was punched in the card. The corresponding completed circuits would cause associated dials on the panel to advance by one unit; and furthermore, one of the 26 lids of the sorting box would pop open. At this point the operator would reopen the press, put the card into the open compartment, and close the lid. One man reportedly ran 19071 cards through this machine in a single $6\frac{1}{2}$ -hour working day, an average of about 49 cards per minute! (A typical operator would work at about one-third this speed.)

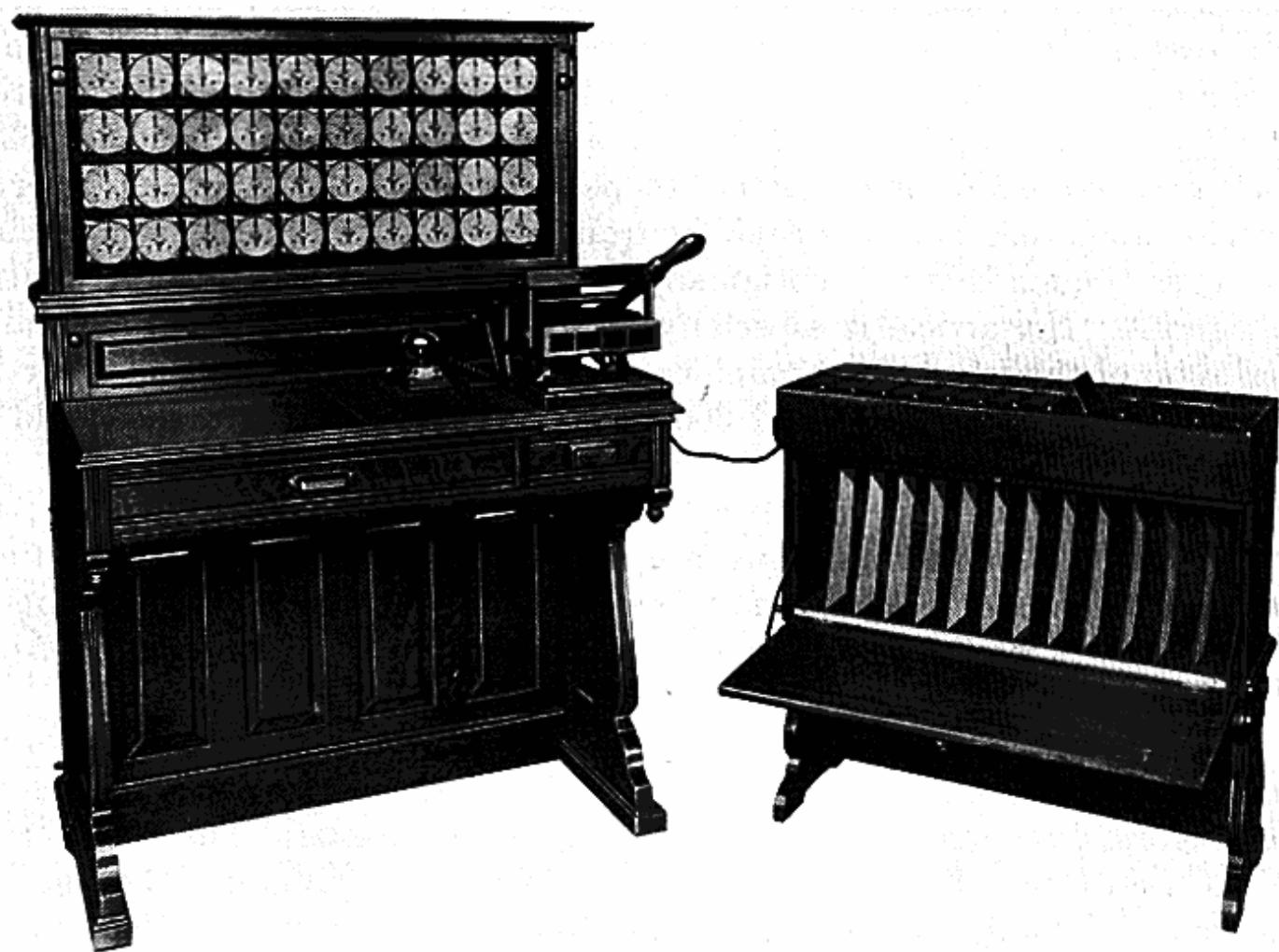


Fig. 94. Hollerith's original tabulating and sorting machine. (Photo courtesy of IBM Archives.)

Population continued its inexorable growth pattern, and the original tabulator-sorters were not fast enough to handle the 1900 census; so Hollerith devised another machine to stave off another data processing crisis. His new device (patented in 1901 and 1904) had an automatic card feed, and in fact it looked essentially like modern card sorters. The story of Hollerith's early machines has been told in interesting detail by Leon E. Truesdell, *The Development of Punch Card Tabulation* (Washington: U.S. Bureau of the Census, 1965); see also the contemporary accounts in *Columbia College School of Mines Quarterly* 10 (1889), 238-255; *J. Franklin Inst.* 129 (1890), 300-306; *The Electrical Engi-*

neer 12 (Nov. 11, 1891), 521–530; *J. Amer. Statistical Assn.* 2 (1891), 330–341, 4 (1895), 365; *J. Royal Statistical Soc.* 55 (1892), 326–327; *Allgemeines Statistisches Archiv* 2 (1892), 78–126; *J. Soc. Statistique de Paris* 33 (1892), 87–96; *U.S. Patents* 395781 (1889), 685608 (1901), 777209 (1904). Hollerith and another former Census Bureau employee, James Powers, went on to found rival companies which eventually became part of IBM and Remington Rand corporations, respectively.

Hollerith's sorting machine is, of course, the basis for radix sorting methods now used in digital computers. His patent mentions that two-column numerical items are to be sorted "separately for each column," but he didn't say whether or not the units or the tens columns should be considered first. The nonobvious trick of using the units column first was presumably discovered by some anonymous machine operator and passed on to others (cf. Section 5.2.5); it appears in the earliest extant IBM sorter manual (1936). The first known mention of this right-to-left technique is an incidental remark which appears in an article by L. J. Comrie, *Trans. of the Office Machinery Users' Assoc.* (London, 1930), 25–37. Incidentally, Comrie was the first person to make the important observation that tabulating machines could be fruitfully employed in scientific calculations, even though they were originally designed for statistical and accounting applications. His article is especially interesting because it gives a detailed description of the tabulating equipment available in England in 1930. Sorting machines at that time processed 360 to 400 cards per minute, and could be rented for £9 per month.

The idea of merging goes back to another card-walloping machine, the *collator*, which was a much later invention (1938). With its two feeding stations, it could merge two sorted decks of cards into one, in only one pass; the technique for doing this was clearly explained in the first IBM collator manual (April, 1939). [Cf. James W. Bryce, *U.S. Patent 2189024* (1940).]

Then computers arrived on the scene, and sorting was intimately involved in this development; in fact, there is evidence that a sorting routine was the first program ever written for a stored program computer. The designers of EDVAC were especially interested in sorting, because it epitomized the potential non-numerical applications of computers; they realized that a satisfactory order code should not only be capable of expressing programs for the solution of difference equations, it must also have enough flexibility to handle the combinatorial "decision-making" aspects of algorithms. John von Neumann therefore prepared programs for internal merge sorting in 1945, in order to test the adequacy of some instruction codes he was proposing for the EDVAC computer; the existence of efficient special-purpose sorting machines provided a natural standard by which the merits of his proposed computer organization could be evaluated. Details of this interesting development have been described in an article by D. E. Knuth, *Computing Surveys* 2 (1970), 247–260; see also von Neumann's *Collected Works* 5 (New York: Macmillan, 1963), 196–214, for the final "polished" form of his original sorting programs.

The limited internal memory size planned for early computers made it

natural to think of external sorting as well as internal sorting, and a "Progress Report on the EDVAC" prepared by J. P. Eckert and J. W. Mauchly of the Moore School of Electrical Engineering (September 30, 1945) pointed out that a computer augmented with magnetic wire or tape devices could simulate the operations of card equipment, achieving a faster sorting speed. This progress report described balanced two-way radix sorting, and balanced two-way merging (called "collating"), using four magnetic wire or tape units, reading or writing "at least 5000 pulses per second."

John Mauchly lectured on "Sorting and Collating" at the special session on computing presented at the Moore School in 1946, and the notes of his lecture constitute the first published discussion of computer sorting [*Theory and techniques for the design of electronic digital computers*, ed. by G. W. Patterson, 3 (1946), 22.1-22.20]. Mauchly began his presentation with an interesting remark: "To ask that a single machine combine the abilities to compute and to sort might seem like asking that a single device be able to perform both as a can opener and a fountain pen." Then he observed that machines capable of carrying out sophisticated mathematical procedures must also have the ability to sort and classify data, and he showed that sorting may even be useful in connection with numerical calculations. He described straight insertion and binary insertion, observing that the former method uses about $N^2/4$ comparisons on the average, while the latter never needs more than about $N \log_2 N$. Yet binary insertion requires a rather complex data structure, and he went on to show that two-way merging achieves the same low number of comparisons using only sequential accessing of lists. The last half of his lecture notes were devoted to a discussion of partial-pass radix sorting methods which simulate digital card sorting on four tapes, using less than four passes per digit (cf. Section 5.4.7).

Shortly afterwards, Eckert and Mauchly started a company which produced some of the earliest electronic computers, the BINAC (for military applications) and the UNIVAC (for commercial applications). Again the U.S. Census Bureau played a part in this development, receiving the first UNIVAC. At this time it was not at all clear that computers would be economically profitable; computing machines could sort faster than card equipment, but they cost more. Therefore the UNIVAC programmers, led by Frances E. Holberton, put considerable effort into the design of high-speed external sorting routines, and their preliminary programs also influenced the hardware design. According to their estimates, 100 million 10-word records could be sorted on UNIVAC in 9000 hours (i.e., 375 days).

UNIVAC I, officially dedicated in July, 1951, had an internal memory of 1000 12-character (72-bit) words. It was designed to read and write 60-word blocks on tapes, at a rate of 500 words per second; reading could be either forward or backward, and simultaneous reading/writing/computing was possible. In 1948, Mrs. Holberton devised an interesting way to do two-way merging with perfect overlap of reading, writing, and computing, using six input buffers: Let there be one "current buffer" and two "auxiliary buffers" for each input file; it is possible to merge in such a way that, whenever it is time

to output one block, the two current input buffers contain a total of exactly one block's worth of unprocessed records. Therefore exactly one input buffer becomes empty while each output block is being formed, and we can arrange to have three of the four auxiliary buffers full at all times while we are reading into the other. This method is slightly faster than the forecasting method of Algorithm 5.4.6F, since it is not necessary to inspect the result of one input before initiating the next. [Cf. "Collation Methods for the UNIVAC System," (Eckert-Mauchly Computer Corp., 1950), 2 vols.]

The culmination of this work was a sort generator program, which was the first major "software" routine ever developed for automatic programming. The user would specify the record size, the positions of up to five keys in partial fields of each record, and the "sentinel" keys which mark file's end, and the sort generator would produce a copyrighted sorting program for one-reel files. The first pass of this program was an internal sort of 60-word blocks, using comparison counting (Algorithm 5.2C); then came a number of balanced two-way merge passes, reading backwards and avoiding tape interlock as described above. [Cf. "Master Generating Routine for 2-way Sorting" (Eckert-Mauchly Div. of Remington Rand, 1952); the first draft of this report was entitled "Master Prefabrication Routine for 2-way Collation"! See also F. E. Holberton, *Symposium on Automatic Programming* (Office of Naval Research, 1954), 34-39.]

By 1952, many approaches to internal sorting were well known in the programming folklore, but comparatively little theory had been developed. Daniel Goldenberg ["Time analyses of various methods of sorting data," Digital Computer Laboratory memo M-1680 (Mass. Inst. of Tech., October 17, 1952)] coded five different methods for the Whirlwind computer, and made best-case and worst-case analyses of each program. When sorting one hundred 15-bit words on an 8-bit key, he found that the fastest method was to use a 256-word table, storing each record into a unique position corresponding to its key, then compressing the table. But this technique had an obvious disadvantage, since it would eliminate a record whenever a subsequent one had the same key. The other four methods he analyzed were ranked as follows: straight two-way merging beat radix-2 sorting beat straight selection beat bubble sort.

These results were extended by Harold H. Seward in his 1954 Master's thesis ["Information sorting in the application of electronic digital computers to business operations," Digital Computer Lab. report R-232 (Mass. Inst. of Tech., May 24, 1954; 60 pp.)]. Seward introduced the ideas of distribution counting and replacement selection; he showed that the first run in a random permutation has an average length of $e - 1$; and he analyzed external sorting as well as internal sorting, on various types of bulk memories as well as tapes.

An even more noteworthy thesis—a Ph.D. thesis in fact—was written by Howard B. Demuth in 1956 ["Electronic Data Sorting" (Stanford University, October, 1956), 92 pp.]. This work helped to lay the foundations of computational complexity theory. It considered three abstract models of the sorting

problem, using cyclic, linear, and random-access memories; and optimal or near-optimal methods were developed for each model. (Cf. exercise 5.3.4–6.) Although no practical consequences flowed immediately from Demuth's thesis, it established important ideas about how to link theory with practice.

Thus the history of sorting has been closely associated with many "firsts" in computing: The first data-processing machines, the first stored programs, the first software, the first buffering methods, the first work on algorithmic analysis and computational complexity.

None of the computer-related documents mentioned so far actually appeared in the "open literature"; in fact, most of the early history of computing appears in comparatively inaccessible reports, because comparatively few people were involved with computers at the time. Literature about sorting finally broke into print in 1955–1956, in the form of three major survey articles.

The first paper was prepared by J. C. Hosken [*Proc. Eastern Joint Computer Conference* 8 (1955), 39–55]. He began with an astute observation: "To lower costs per unit of output, people usually increase the size of their operations. But under these conditions, the unit cost of sorting, instead of falling, rises." Hosken surveyed all the available special-purpose equipment then being marketed, as well as the methods of sorting on computers. His bibliography of 54 items is mostly based on manufacturer's brochures.

The comprehensive paper "Sorting on Electronic Computer Systems" by E. H. Friend [*JACM* 3 (1956), 134–168] was a major milestone in the development of sorting. Although numerous techniques have been developed since 1956, this paper is still remarkably up-to-date in many respects. Friend gave careful descriptions of quite a few internal and external sorting algorithms, and he paid special attention to buffering techniques and the characteristics of magnetic tape units. He introduced some new methods (e.g., tree selection, amphibiaen sorting, and forecasting), and developed some of the mathematical properties of the older methods.

The third survey of sorting to appear about this time was prepared by D. W. Davies [*Proc. Inst. Elect. Engineers* 103B, Supplement 1 (1956), 87–93]. In the following years several other notable surveys were published, by D. A. Bell [*Comp. J.* 1 (1958), 71–77]; A. S. Douglas [*Comp. J.* 2 (1959), 1–9]; D. D. McCracken, H. Weiss, and T. Lee [*Programming Business Computers* (New York: Wiley, 1959), Chapter 15, pp. 298–332]; I. Flores [*JACM* 8 (1961), 41–80]; K. E. Iverson [*A Programming Language* (New York: Wiley, 1962), Chapter 6, 176–245]; C. C. Gotlieb [*CACM* 6 (1963), 194–201]; T. N. Hibbard [*CACM* 6 (1963), 206–213]; M. A. Goetz [*Digital Computer User's Handbook*, ed. by M. Klerer and G. A. Korn (New York: McGraw-Hill, 1967), Chapter 1.10, pp. 1.292–1.320]. A symposium on sorting was sponsored by ACM in November, 1962; most of the papers presented at that symposium were published in the May, 1963, issue of *CACM*, and they constitute a good representation of the state of the art at that time. C. C. Gotlieb's survey of contemporary sort generators, T. N. Hibbard's survey of minimal storage internal sorting, and

G. U. Hubbard's early exploration of disk file sorting are particularly noteworthy articles in this collection.

New sorting methods were being discovered throughout this period: Address calculation (1956), merge insertion (1959), radix exchange (1959), cascade merge (1959), Shell's diminishing increment sort (1959), polyphase merge (1960), tree insertion (1960), oscillating sort (1962), Hoare's quicksort (1962), Williams's heapsort (1964), Batcher's merge exchange (1964). The history of each individual algorithm has been traced in the particular section of this chapter where that method is described. The late 1960's saw an intensive development of the corresponding theory.

A complete bibliography of all papers on sorting examined by the author as this chapter was being written appears in *Computing Reviews* 13 (1972), 283-289.

EXERCISES

1. [05] Summarize the content of this chapter by stating a generalization of Theorem 5.4.6A.
2. [20] Based on the information in Table 1, what is the best list-sorting method for six-digit keys, for use on the MIX computer?
3. [47] (*Stable sorting in minimum storage.*) A sorting algorithm is said to require *minimum storage* if it uses only $O((\log N)^2)$ bits of memory space for its variables besides the space needed to store the N records. The algorithm must be general in the sense that it works for all N , not just for a particular value of N , assuming that a sufficient amount of random access memory has been made available whenever the algorithm is actually called upon to sort.
Many of the sorting methods we have studied violate this minimum-storage requirement; in particular, the use of N link fields is forbidden. Quicksort (Algorithm 5.2.2Q) satisfies the minimum-storage requirement, but its worst case running time is proportional to N^2 . Heapsort (Algorithm 5.2.3H) is the only $O(N \log N)$ algorithm we have studied which uses minimum storage, although another such algorithm could be formulated using the idea of exercise 5.2.4–18.
The fastest general algorithm we have considered which sorts keys in a *stable* manner is the list merge sort (Algorithm 5.2.4L), but it does not use minimum storage. In fact, the only stable minimum-storage sorting algorithms we have seen are $O(N^2)$ methods (straight insertion, bubble sorting, and a variant of straight selection).
Is there a stable minimum-storage sorting algorithm which requires less than $O(N^2)$ units of time in its worst case, and/or on the average?

*I shall have accomplished my purpose if I have sorted and put in logical order
the gist of the great volume of material which has been generated about sorting
over the past few years.*

—J. C. HOSKEN (1955)

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 $\text{TABLE} + i$, the associated data (or a pointer to it) might be in location $\text{TABLE} + i + 1$, or in $\text{DATA} + i$, 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 $\{1, 2, \dots, N\}$; then the record containing K can simply be placed in location $\text{TABLE} + K$. 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, $A = \{a_1, a_2, \dots, a_m\}$ and $B = \{b_1, b_2, \dots, b_n\}$, 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 $c_1 mn$ 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 → 1	l → 4
c, g, j, k, q, s, x, z → 2	m, n → 5
d, t → 3	r → 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 sports-writer 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, \dots, R_N , whose respective keys are K_1, K_2, \dots, 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. ■

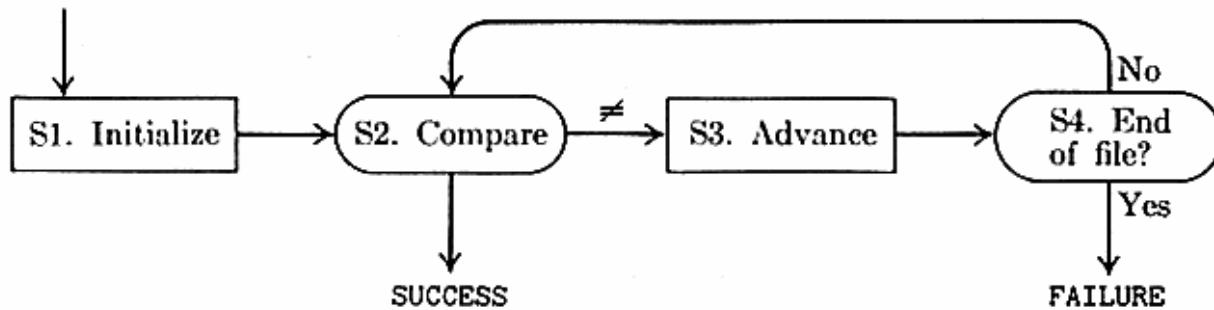


Fig. 1. Sequential search.

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 $\text{KEY} + i$, and that the remainder of record R_i appears in location $\text{INFO} + i$. The following program uses $rA \equiv K$, $rI1 \equiv i - N$.

01	START	LDA	K	1	<u>S1. Initialize.</u>
02		ENT1	1-N	1	$i \leftarrow 1$.
03	2H	CMPA	KEY+N,1	C	<u>S2. Compare.</u>
04		JE	SUCCESS	C	Exit if $K = K_i$.
05		INC1	1	C - S	<u>S3. Advance.</u>
06		J1NP	2B	C - S	<u>S4. End of file?</u>
07	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,

$$\begin{aligned} C &= \text{the number of key comparisons;} \\ S &= 1 \text{ if successful, } 0 \text{ if unsuccessful.} \end{aligned} \quad (1)$$

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

$$\frac{1 + 2 + \dots + N}{N} = \frac{N + 1}{2}; \quad (2)$$

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 $i \leq N$, the algorithm terminates successfully; otherwise it terminates unsuccessfully ($i = N + 1$). ■

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

01	BEGIN	LDA	K	1	<u>Q1. Initialize.</u>
02		STA	KEY+N+1	1	$K_{N+1} \leftarrow K$.
03		ENT1	-N	1	$i \leftarrow 0$.
04		INC1	1	$C + 1 - S$	<u>Q3. Advance.</u>
05		CMPA	KEY+N,1	$C + 1 - S$	<u>Q2. Compare.</u>
06		JNE	*-2	$C + 1 - S$	To Q3 if $K_i \neq K$.
07		J1NP	SUCCESS	1	<u>Q4. End of file?</u>
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 $C \geq 6$ in a successful search, and it is an improvement whenever $N \geq 8$ 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	BEGIN	LDA	K	1	<u>Q1. Initialize.</u>
02		STA	KEY+N+1	1	$K_{N+1} \leftarrow K$.
03		ENT1	-1-N	1	$i \leftarrow -1$.
04	3H	INC1	2	$\lfloor (C - S + 2)/2 \rfloor$	<u>Q3. Advance.</u> (twice)
05		CMPA	KEY+N,1	$\lfloor (C - S + 2)/2 \rfloor$	<u>Q2. Compare.</u>
06		JE	4F	$\lfloor (C - S + 2)/2 \rfloor$	To Q4 if $K = K_i$.
07		CMPA	KEY+N+1,1	$\lfloor (C - S + 1)/2 \rfloor$	<u>Q2. Compare.</u> (next)
08		JNE	3B	$\lfloor (C - S + 1)/2 \rfloor$	To Q3 if $K \neq K_{i+1}$.
09		INC1	1	$(C - S) \bmod 2$	Advance i .
10	4H	J1NP	SUCCESS	1	<u>Q4. End of file?</u>
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

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

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, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, 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 $p_1 + p_2 + \dots + p_N = 1$. The time required to do a successful search is essentially proportional to the number of comparisons, C , which now has the average value

$$\bar{C}_N = p_1 + 2p_2 + \dots + Np_N. \quad (3)$$

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

$$p_1 \geq p_2 \geq \dots \geq p_N, \quad (4)$$

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 $p_1 = p_2 = \dots = p_N = 1/N$, formula (3) reduces to $\bar{C}_N = (N + 1)/2$; we have already derived this in Eq. (2). Suppose, on the other hand, that

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

Then by exercise 7, $\bar{C}_N = 2 - 2^{1-N}$; 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

$$p_1 = Nc, \quad p_2 = (N-1)c, \quad \dots, \quad p_N = c,$$

where

$$c = 2/N(N+1). \quad (6)$$

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

$$\bar{C}_N = c \sum_{1 \leq k \leq N} k \cdot (N+1-k) = \frac{N+2}{3}; \quad (7)$$

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,”

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

This distribution was formulated by G. K. Zipf, who observed that the n th 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

$$\bar{C}_N = N/H_N; \quad (9)$$

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,

$$\frac{p_1 + p_2 + \dots + p_{.20n}}{p_1 + p_2 + p_3 + \dots + p_n} \approx .80, \quad \text{for all } n. \quad (10)$$

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

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

where

$$c = 1/N^\theta, \quad \theta = \frac{\log .80}{\log .20} = 0.1386, \quad (12)$$

since $p_1 + p_2 + \dots + p_n = cn^\theta$ 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

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

Here $\theta = \log .80/\log .20$ as before, and $H_N^{(s)}$ is the N th harmonic number of order s , namely $1^{-s} + 2^{-s} + \dots + N^{-s}$. 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

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

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

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

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

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

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

A “self-organizing” file. The above calculations with probabilities are very nice, but in most cases we don't know 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

auxiliary count fields: Whenever a record has been successfully located, it is moved to the beginning of the table. This procedure is readily implemented if the table is a linked linear list, especially because the record being moved to the beginning often has to be substantially updated anyway.

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

$$\bar{C}_N = 1 + 2 \sum_{1 \leq i < j \leq N} \frac{p_i p_j}{p_i + p_j} = \frac{1}{2} + \sum_{i,j} \frac{p_i p_j}{p_i + p_j}. \quad (17)$$

(See exercise 11.) For example, if $p_i = 1/N$ for $1 \leq i \leq N$, the self-organizing table is always in completely random order, and this formula reduces to the familiar expression $(N + 1)/2$ derived above.

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

$$\begin{aligned} \bar{C}_N &= \frac{1}{2} + \sum_{1 \leq i,j \leq N} \frac{(c/i)(c/j)}{c/i + c/j} = \frac{1}{2} + c \sum_{1 \leq i,j \leq N} \frac{1}{i+j} \\ &= \frac{1}{2} + c \sum_{1 \leq i \leq N} (H_{N+i} - H_i) = \frac{1}{2} + c \sum_{1 \leq i \leq 2N} H_i - 2c \sum_{1 \leq i \leq N} H_i \\ &= \frac{1}{2} + c((2N+1)H_{2N} - 2N - 2(N+1)H_N + 2N) \\ &= \frac{1}{2} + c(N \ln 4 - \ln N + O(1)) \\ &\approx 2N/\log_2 N, \end{aligned} \quad (18)$$

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).

A somewhat similar self-organizing scheme has been studied by G. Schay, Jr. and F. W. Dauer, *SIAM J. Appl. Math.* **15** (1967), 874–888.

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

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

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 ~~$\frac{2}{3} \cdot 1 + \frac{1}{3} \cdot 4 = \frac{11}{3}$~~ .

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

$$p_1/L_1 \geq p_2/L_2 \geq \cdots \geq p_N/L_N. \quad (20)$$

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 $a_1 a_2 \dots a_N$ of $\{1, 2, \dots, N\}$, 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 $R_1 R_2 \dots R_N$ by using a sequence of interchanges of adjacent records. Each of these interchanges replaces $\dots R_j R_i \dots$ by $\dots R_i R_j \dots$ for some $i < j$, so it decreases the search time by the non-negative amount $p_i L_j - p_j L_i$. Therefore the order $R_1 R_2 \dots R_N$ must have minimum search time.

Second proof. Replace each probability p_i by

$$p_i(\epsilon) = p_i + \epsilon^i - (\epsilon^1 + \epsilon^2 + \dots + \epsilon^N)/N, \quad (21)$$

where ϵ is an extremely small positive number. When ϵ is sufficiently small, we will never have $x_1 p_1(\epsilon) + \dots + x_N p_N(\epsilon) = y_1 p_1(\epsilon) + \dots + y_N p_N(\epsilon)$ unless $x_1 = y_1, \dots, x_N = y_N$; 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 $(\text{gap size})/2$; 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	52	19609
2	3	72	31397
4	7	86	155921
6	23	96	360653
8	89	112	370261
14	113	114	492113
18	523	118	1349533
20	887	132	1357201
22	1129	148	2010733
34	1327	154	4652353
36	9551	180	17051707
44	15683	210	20831323

This table lists $p_{k+1} - p_k$ whenever it exceeds $p_{j+1} - p_j$ for all $j < k$. For further information, see "Statistics on the first six million prime numbers" by F. Gruenberger and G. Armerding, RAND Corp. report P-2460 (October, 1961).

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 **A**.)
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 $n \rightarrow \infty$, 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 $f_m(x_1, x_2, \dots, x_m)$ be the infinite sum of all distinct ordered products $x_{i_1}x_{i_2}\dots x_{i_k}$ such that $1 \leq i_1, \dots, i_k \leq m$, where each of x_1, x_2, \dots, x_m appears in every term. For example,

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

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

$$p_{nm} = \sum_{1 \leq j_1 < \dots < j_m \leq n} f_m(x_{j_1}, \dots, x_{j_m}); \quad Q_{nm} = \sum_{1 \leq j_1 < \dots < j_m \leq n} \frac{1}{1-x_{j_1}-\dots-x_{j_m}}.$$

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 m th item from the front with limiting probability $p_i P_{N-1,m-1}$, where $X = \{p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_N\}$.
- b) By summing the result of (a) for $m = 1, 2, \dots$, we obtain the identity

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

Prove that, consequently,

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

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

- c) Compute the limiting average distance $d_i = \sum_{m \geq 1} m p_i P_{N-1,m-1}$ of R_i from the front of the list; then evaluate $\bar{C}_N = \sum_{1 \leq i \leq n} p_i d_i$.

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 \bar{C}_N for the probability distribution (6).

14. [M21] Given two sequences $\langle x_1, x_2, \dots, x_n \rangle$ and $\langle y_1, y_2, \dots, y_n \rangle$ of real numbers, what permutation $a_1 a_2 \dots a_n$ of the subscripts will make $\sum x_i y_{a_i}$ 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 $(1 - P_1)(1 - P_2) \dots (1 - P_N)$; and the probability that the search will end just after loading the i th subroutine is $P_i(1 - P_{i+1}) \dots (1 - P_N)$. If L_i is the length of subroutine i , the average search time will therefore be essentially proportional to

$$L_1 P_1 (1 - P_2) \dots (1 - P_N) + (L_1 + L_2) P_2 (1 - P_3) \dots (1 - P_N) \\ + \dots + (L_1 + \dots + L_N) P_N.$$

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 i th job takes T_i units of time, and it has a *deadline* D_i . In other words, the i th job is supposed to be finished after at most D_i units of time have elapsed. What schedule for processing the jobs will minimize the *maximum tardiness*, i.e.,

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

18. [M30] (*Catenated search*.) Suppose N records are located in a linear array $R_1 \dots R_N$, 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 $\sum_{1 \leq i, j \leq N} p_i p_j d(i, j)$, 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 $d_1 < d_2 < \dots < d_{N-1}$. (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 $p_1 \leq p_N \leq p_2 \leq p_{N-1} \leq \dots \leq p_{\lfloor N/2 \rfloor + 1}$ or $p_N \leq p_1 \leq p_{N-1} \leq p_2 \leq \dots \leq p_{\lfloor N/2 \rfloor}$. (Thus, an “organ pipe” arrangement of probabilities is best, as shown in Fig. 2.) *Hint:* Consider any arrangement where the respective probabilities are $q_1 q_2 \dots q_k s r_k \dots r_2 r_1 t_1 \dots t_m$, for some $m \geq 0$ and $k > 0; N = 2k + m + 1$. Show that the rearrangement $q'_1 q'_2 \dots q'_k s r'_k \dots r'_2 r'_1 t_1 \dots t_m$ 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 $N = 2k + m$.

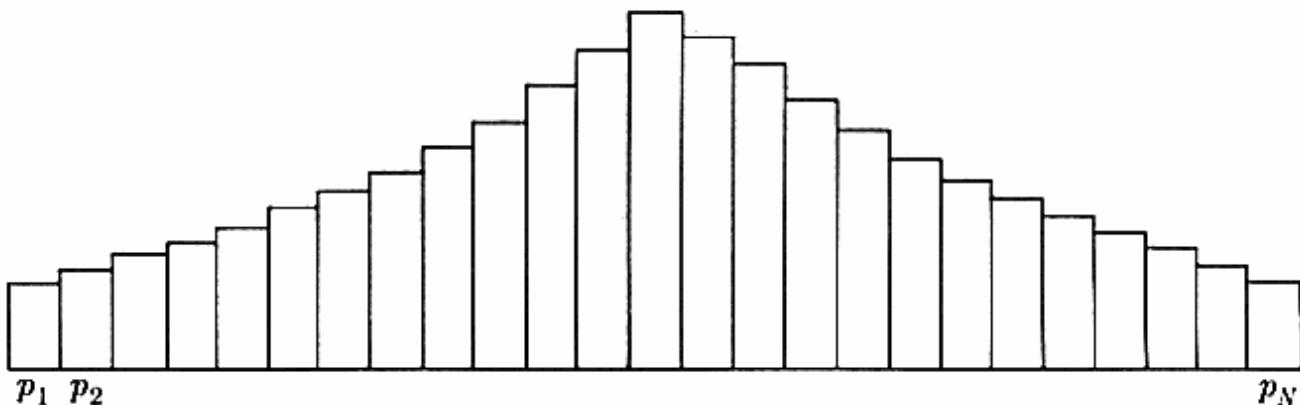


Fig. 2. An “organ-pipe arrangement” of probabilities minimizes the average seek time in a catenated search.

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, $d(i, j) = a + b(L_{i+1} + \dots + L_j)$ and $d(j, i) = a + b(L_{j+1} + \dots + L_N) + r + b(L_1 + \dots + L_i)$, 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 $d_1 < d_2 < \dots$? [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, \dots, 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 $x_0 \leq x_1 \leq \dots \leq x_{2^n-1}$ 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?

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}, \dots, R_N are eliminated from consideration];
- or • $K = K_i$ [the search is done];
- or • $K > K_i$ [R_1, R_2, \dots, 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

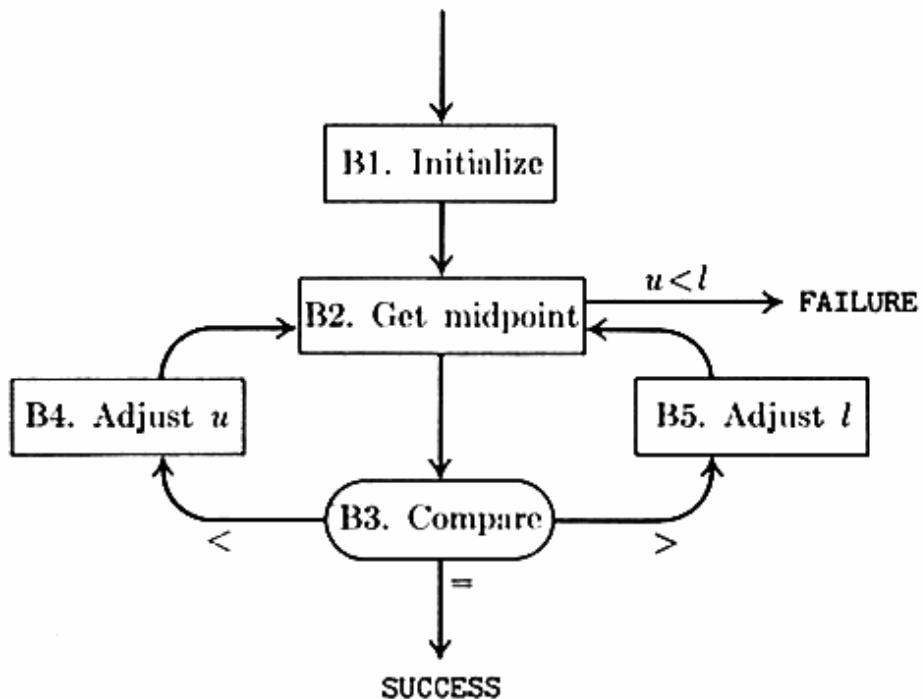


Fig. 3. Binary search.

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, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, 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 $K_l \leq K \leq K_u$. 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 :

```
[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
 061 087 154 170 275 426 503 509[512 612 653 677 703 765 897 908]
 061 087 154 170 275 426 503 509[512 612 653] 677 703 765 897 908
 061 087 154 170 275 426 503 509 512 612[653] 677 703 765 897 908
```

b) Searching for 400 :

```
[061 087 154 170 275 426 503 509 512 612 653 677 703 765 897 908]
[061 087 154 170 275 426 503] 509 512 612 653 677 703 765 897 908
 061 087 154 170 [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
 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 $\text{KEY} + i$. The following code uses $rI1 \equiv l$, $rI2 \equiv u$, $rI3 \equiv i$.

01	START	ENT1	1	1	<u>B1. Initialize.</u> $l \leftarrow 1$.
02		ENT2	N	1	$u \leftarrow N$.
03		JMP	2F	1	To B2.
04	5H	JE	SUCCESS	C1	Jump if $K = K_i$.
05		ENT1	1,3	C1 - S	<u>B5. Adjust l.</u> $l \leftarrow i + 1$.
06	2H	ENTA	0,1	C + 1 - S	<u>B2. Get midpoint.</u>
07		INCA	0,2	C + 1 - S	$rA \leftarrow l + u$.
08		SRB	1	C + 1 - S	$rA \leftarrow \lfloor rA/2 \rfloor$.
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	$i \leftarrow \text{midpoint}$.
13	3H	LDA	K	C	<u>B3. Compare.</u>
14		CMPA	KEY,3	C	
15		JGE	5B	C	Jump if $K \geq K_i$.
16		ENT2	-1,3	C2	<u>B4. Adjust u.</u> $u \leftarrow i - 1$.
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 $(26C - 18S + 20)u$.

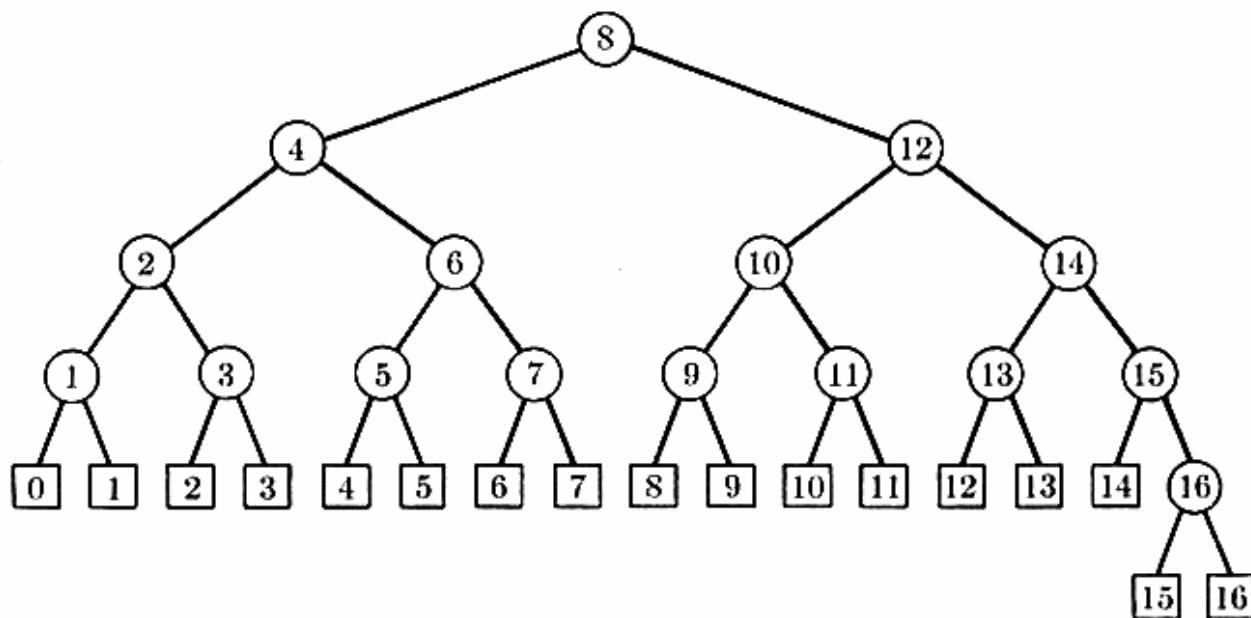


Fig. 5. A binary tree which corresponds to binary search when $N = 16$.

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 0 through 16; for example, we reach node 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 0. Otherwise the root node is

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

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

In an analogous fashion, *any* algorithm for searching an ordered table of length N by means of comparisons can be represented as 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

$$0 \quad 1 \quad 1 \quad 2 \quad 2 \quad \dots \quad N-1 \quad N \quad N \quad (1)$$

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 $2^{k-1} \leq N < 2^k$, 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 $2^{k-1} \leq N < 2^k - 1$, 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 *unsuccessful* 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 :

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

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

$$(N + 1)(\lfloor \log_2 N \rfloor + 2) - 2^{\lfloor \log_2 N \rfloor + 1}. \quad (3)$$

(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 $k = \lfloor \log_2 N \rfloor$, we have (cf. Eq. 5.3.1–34)

$$\begin{aligned} C_N &= k + 1 - (2^{k+1} - k - 2)/N = \log_2 N - 1 + \epsilon + (k + 2)/N, \\ C'_N &= k + 2 - 2^{k+1}/(N + 1) = \log_2 N + \epsilon', \end{aligned} \quad (4)$$

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

To summarize: Algorithm B never makes more than $\lfloor \log_2 N \rfloor + 1$ comparisons, and it makes about $\log_2 N - 1$ 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

$$\begin{aligned} (18 \log_2 N - 15)u &\quad \text{for a successful search,} \\ (18 \log_2 N + 13)u &\quad \text{for an unsuccessful search,} \end{aligned} \quad (5)$$

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, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, 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 $N \geq 1$.

- U1. [Initialize.]** Set $i \leftarrow \lceil N/2 \rceil$, $m \leftarrow \lfloor N/2 \rfloor$.
- U2. [Compare.]** If $K < K_i$, go to U3; if $K > K_i$, go to U4; and if $K = K_i$, the algorithm terminates successfully.
- U3. [Decrease i .]** (We have pinpointed the search to an interval 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. ■

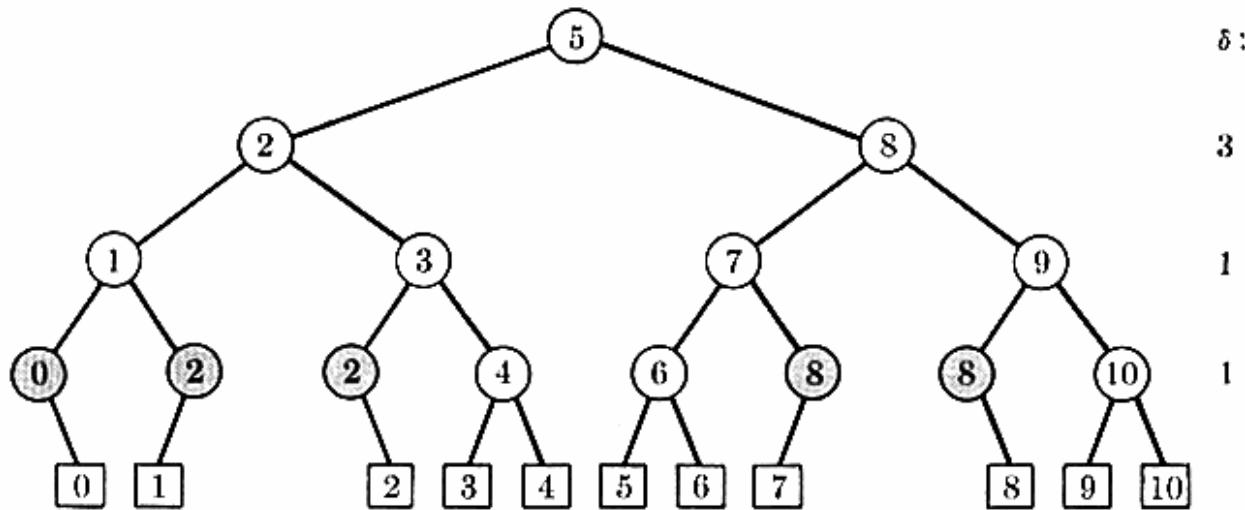


Fig. 6. The binary tree for a “uniform” binary search, when $N = 10$.

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

$$\text{DELTA}[j] = \left\lfloor \frac{N + 2^{j-1}}{2^j} \right\rfloor = \left(\frac{N}{2^j} \right) \text{rounded}, \quad \text{for } 1 \leq j \leq \lfloor \log_2 N \rfloor + 2. \quad (6)$$

- 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 $\text{DELTA}[j] = 0$, 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 $\text{rA} \equiv K$, $\text{rI1} \equiv i$, $\text{rI2} \equiv j$, $\text{rI3} \equiv \text{DELTA}[j]$.

01	START	ENT1	$N+1/2$	1	<u>C1. Initialize.</u>
02		ENT2	2	1	$j \leftarrow 2$.
03		LDA	K	1	
04		JMP	2F	1	
05	3H	JE	SUCCESS	C1	Jump if $K = K_i$.
06		J3Z	FAILURE	C1 - S	Jump if $\text{DELTA}[j] = 0$.
07		DEC1	0,3	C1 - S - A	<u>C3. Decrease i.</u>
08	5H	INC2	1	C - 1	$j \leftarrow j + 1$.
09	2H	LD3	DELTA,2	C	<u>C2. Compare.</u>
10		CMPA	KEY,1	C	
11		JLE	3B	C	Jump if $K \leq K_i$.
12		INC1	0,3	C2	<u>C4. Increase i.</u>
13		J3NZ	5B	C2	Jump if $\text{DELTA}[j] \neq 0$.
14	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 $\lfloor \log_2 N \rfloor + 1$ 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

$$(8.5 \log_2 N - 6)u \quad \text{for a successful search,} \tag{7}$$

$$(8.5 \lfloor \log_2 N \rfloor + 12)u \quad \text{for an unsuccessful search.}$$

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 $i = 2^k$, $k = \lfloor \log_2 N \rfloor$. If $K < K_i$, we use a uniform search with the δ 's equal to 2^{k-1} , 2^{k-2} , ..., 1, 0. On the other hand, if $K > K_i$ and $N > 2^k$, we reset i to $i' = N + 1 - 2^\ell$, where $\ell = \lfloor \log_2 (N - 2^k) \rfloor + 1$, and pretend that the first comparison was actually $K > K_{i'}$, using a uniform search with the δ 's equal to $2^{\ell-1}$, $2^{\ell-2}$, ..., 1, 0.

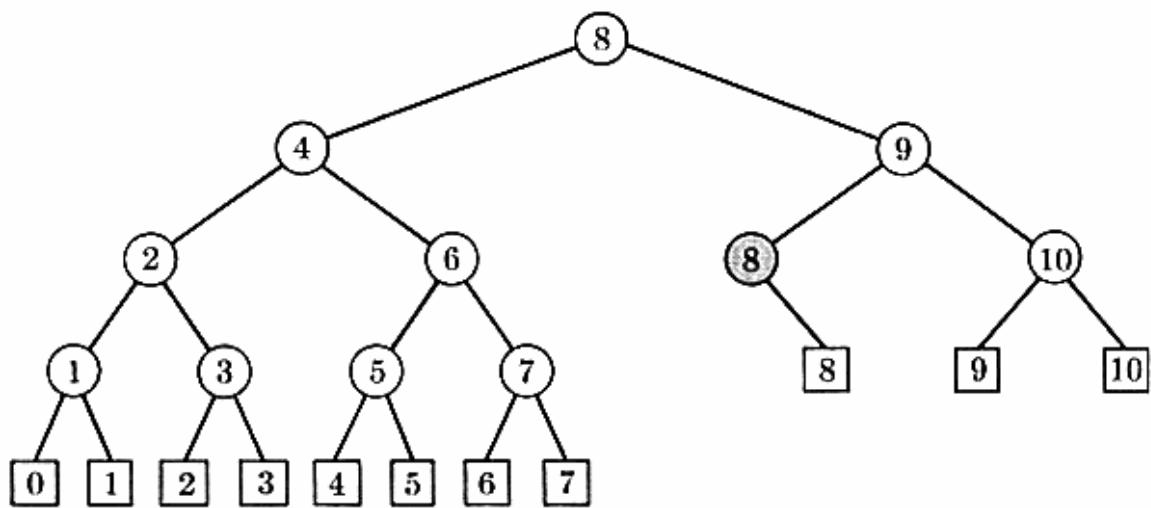


Fig. 7. The binary tree for Shar's almost uniform search, when $N = 10$.

Shar's method is illustrated for $N = 10$ in Fig. 7. Like the previous algorithms, it never makes more than $\lfloor \log_2 N \rfloor + 1$ 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 .

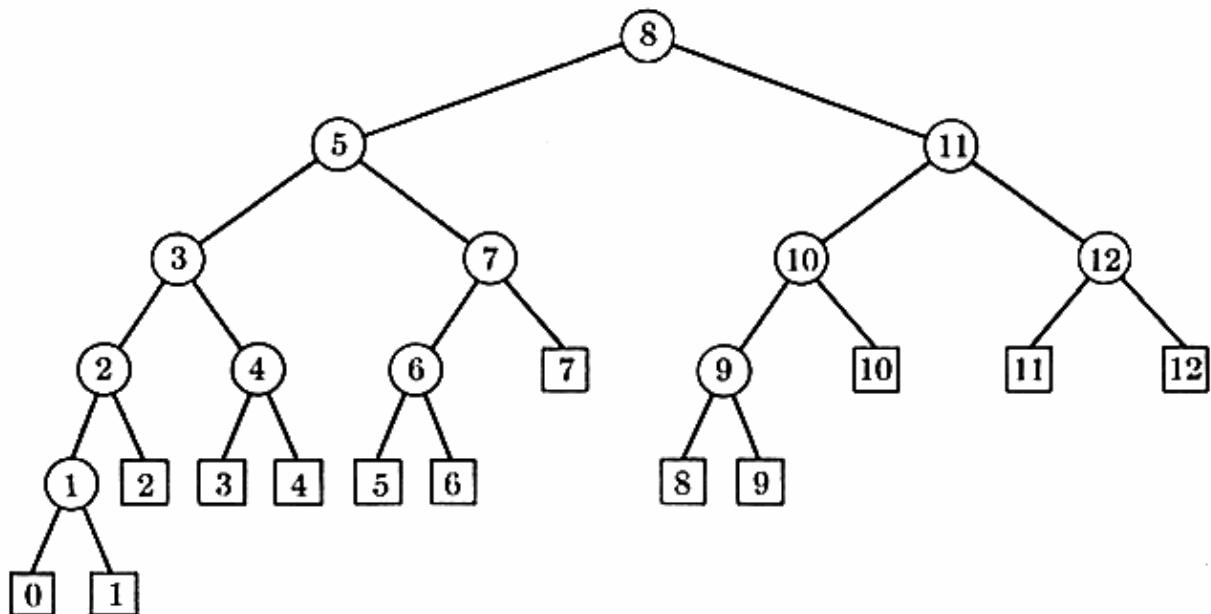


Fig. 8. The Fibonacci tree of order 6.

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 (Fibonacci search). Given a table of records R_1, R_2, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, 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.

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

	01	START	LDA	K		1	<u>F1. Initialize.</u>	
	02		ENT1	F_k		1	$i \leftarrow F_k$.	
	03		ENT2	$F_{k-1}-1$		1	$p \leftarrow F_{k-1}$.	
	04		ENT3	$F_{k-2}-1$		1	$q \leftarrow F_{k-2}$.	
	05		JMP	F2A		1	To step F2.	
06	F4A	INC1	1,3		18	F4B	INC1 1,2	<u>F4. Increase i.</u> $i \leftarrow i + q$.
07		DEC2	1,3		19	DEC3	1,2	$p \leftarrow p - q$.
08		DEC3	1,2		20	DEC2	1,3	$q \leftarrow q - p$.
09	F2A	CMPA	KEY,1		21	F2B	CMPA KEY,1	<u>F2. Compare.</u>
10		JL	F3A		22	JL	F3B	To F3 if $K < K_i$.
11		JE	SUCCESS		23	JE	SUCCESS	Exit if $K = K_i$.
12		J2NZ	F4A		24	J3NZ	F4B	To F4 if $p \neq 1$.
13		JMP	FAILURE		25	JMP	FAILURE	Exit if not in table.
14	F3A	DEC1	1,3		26	F3B	DEC1 1,2	<u>F3. Decrease i.</u> $i \leftarrow i - q$.
15		DEC2	1,3		27	DEC3	1,2	$p \leftarrow p - q$.
16		J3NN	F2B		28	J2NN	F2A	Swap registers if $q > 0$.
17		JMP	FAILURE		29	JMP	FAILURE	Exit if not in table. ■
						1 - S - A		

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 , C_1 , and $(C_2 - S)$ be the respective number of times steps F2, F3, and F4 are performed. Then we have

$$\begin{aligned} C &= (\text{ave } \phi k / \sqrt{5} + O(1), \quad \max k - 1), \\ C_1 &= (\text{ave } k / \sqrt{5} + O(1), \quad \max k - 1), \\ C_2 - S &= (\text{ave } \phi^{-1} k / \sqrt{5} + O(1), \quad \max \lfloor k/2 \rfloor). \end{aligned} \quad (8)$$

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

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

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

<i>amo</i>	precedes	<i>bibo</i>
<i>abeo</i>	precedes	<i>adeo</i>
<i>amatus</i>	precedes	<i>amor</i>
<i>imprudens</i>	precedes	<i>impudens</i>
<i>iusticia</i>	precedes	<i>iustus</i>
<i>polisintheton</i>	precedes	<i>polissenus</i>

(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

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, 3 (1946), 22.8–22.9]. The method became “well known” during the 50’s, 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.

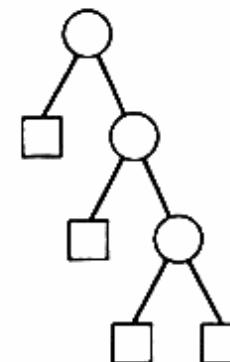
EXERCISES

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

$K_u < K < K_l$. (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 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 $\sum_{0 \leq j \leq \lfloor \log_2 N \rfloor + 2} \text{DELTA}[j]$ 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 $7 \log_2 N$ instructions and having a running time of about $4.5 \log_2 N$ 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 $1 \leq N \leq 16$, considering both successful and unsuccessful searches.

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

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?

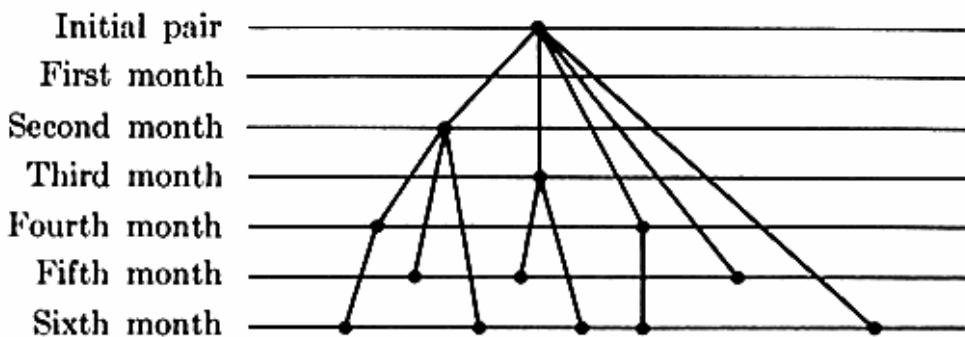


Fig. 9. Pairs of rabbits breeding by Fibonacci's rule.

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 $n = F_{a_1} + F_{a_2} + \dots + F_{a_r}$, 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 (n) has length $k + 1 - r - a_r$.
18. [M30] Find exact formulas for the average values of C_1 , C_2 , 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 Fibonacci 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

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

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

- a) Show that $C(N) = \log_b N$ satisfies these relations exactly, for a certain choice of b . For binary and Fibonacci 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 $\log_2 N$ 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 $k = 0, 1, \dots$. (The root is at level zero.) Thus in Fig. 8 we have $(a_0, a_1, \dots, a_6) = (1, 2, 4, 4, 1, 0)$ and $(b_0, b_1, \dots, b_6) = (0, 0, 0, 4, 7, 2)$. (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, \dots, 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 $N \rightarrow \infty$, 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 $\log_2(k+1)$, 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

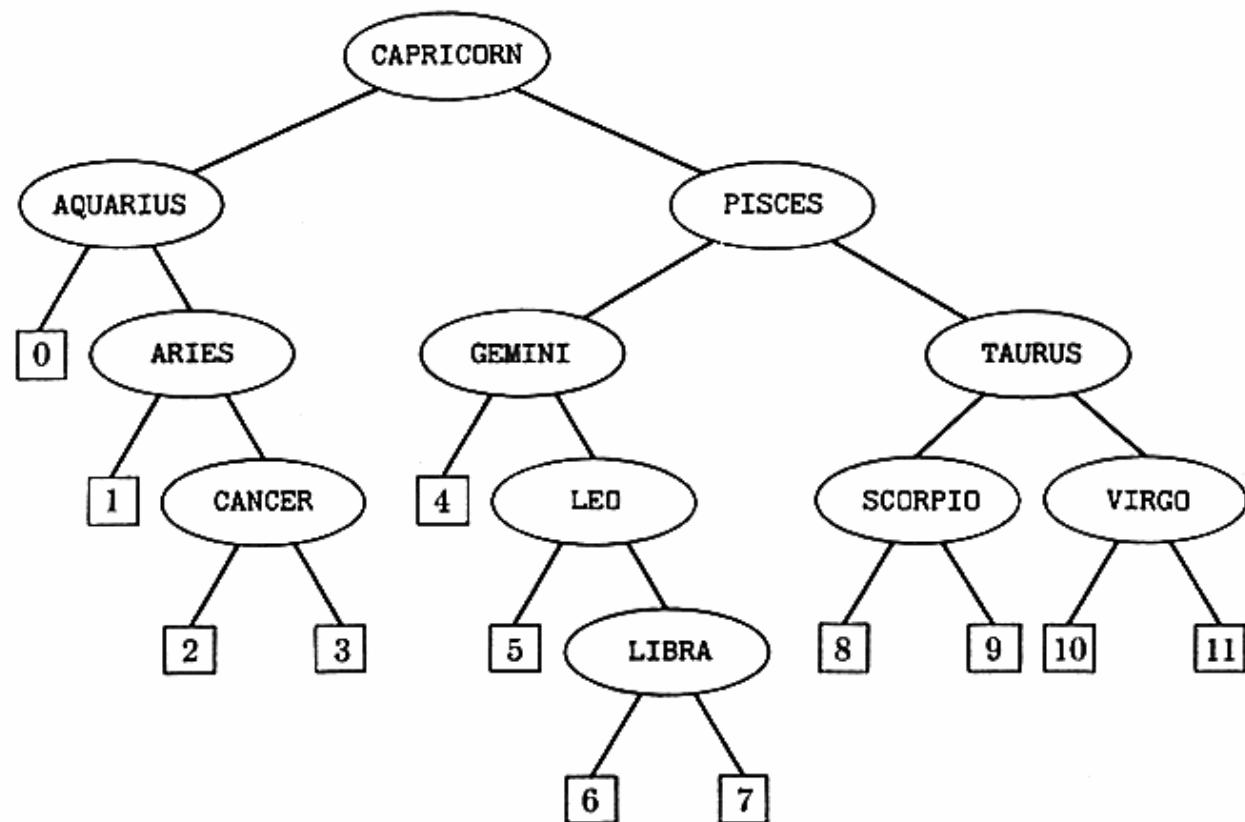


Fig. 10. A binary search tree.

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 [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 [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:

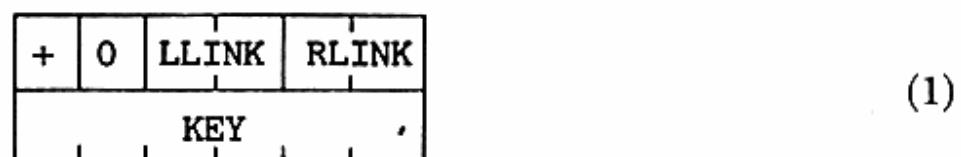
KEY(P) = key stored in NODE(P);
 LLINK(P) = pointer to left subtree of NODE(P);
 RLINK(P) = pointer to right subtree of NODE(P).

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., $\text{ROOT} \neq \Lambda$).

- T1. [Initialize.] Set $P \leftarrow \text{ROOT}$. (The pointer variable P will move down the tree.)
- T2. [Compare.] If $K < \text{KEY}(P)$, go to T3; if $K > \text{KEY}(P)$, go to T4; and if $K = \text{KEY}(P)$, the search terminates successfully.
- T3. [Move left.] If $\text{LLINK}(P) \neq \Lambda$, set $P \leftarrow \text{LLINK}(P)$ and go back to T2. Otherwise go to T5.
- T4. [Move right.] If $\text{RLINK}(P) \neq \Lambda$, set $P \leftarrow \text{RLINK}(P)$ and go back to T2.
- T5. [Insert into tree.] (The search is unsuccessful; we will now put K into the

tree.) Set $Q \leftarrow \text{AVAIL}$, the address of a new node. Set $\text{KEY}(Q) \leftarrow K$, $\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$. (In practice, other fields of the new node should also be initialized.) If K was less than $\text{KEY}(P)$, set $\text{LLINK}(P) \leftarrow Q$, otherwise set $\text{RLINK}(P) \leftarrow Q$. (At this point we could set $P \leftarrow Q$ 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



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

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

01	LLINK	EQU	2:3		
02	RLINK	EQU	4:5		
03	START	LDA	K	1	<u>T1. Initialize.</u>
04		LD1	ROOT	1	$P \leftarrow \text{ROOT}$.
05		JMP	2F	1	
06	4H	LD2	0,1(RLINK)	C2	<u>T4. Move right.</u> $Q \leftarrow \text{RLINK}(P)$.
07		J2Z	5F	C2	To T5 if $Q = \Lambda$.
08	1H	ENT1	0,2	C — 1	<u>T2. Compare.</u>
09	2H	CMPA	1,1	C	$P \leftarrow Q$.
10		JG	4B	C	To T4 if $K > \text{KEY}(P)$.
11		JE	SUCCESS	C1	Exit if $K = \text{KEY}(P)$.
12		LD2	0,1(LLINK)	C1 — S	<u>T3. Move left.</u> $Q \leftarrow \text{LLINK}(P)$.
13		J2NZ	1B	C1 — S	To T2 if $Q \neq \Lambda$.
14	5H	LD2	AVAIL	1 — S	<u>T5. Insert into tree.</u>
15		J2Z	OVERFLOW	1 — S	
16		LDX	0,2(RLINK)	1 — S	
17		STX	AVAIL	1 — S	$Q \leftarrow \text{AVAIL}$.
18		STA	1,2	1 — S	$\text{KEY}(Q) \leftarrow K$.
19		STZ	0,2	1 — S	$\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$.
20		JL	1F	1 — S	Was $K < \text{KEY}(P)$?
21		ST2	0,1(RLINK)	A	$\text{RLINK}(P) \leftarrow Q$.
22		JMP	*+2	A	
23	1H	ST2	0,1(LLINK)	1 — S — A	$\text{LLINK}(P) \leftarrow Q$.
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 $K \leq \text{KEY}(P)$;

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

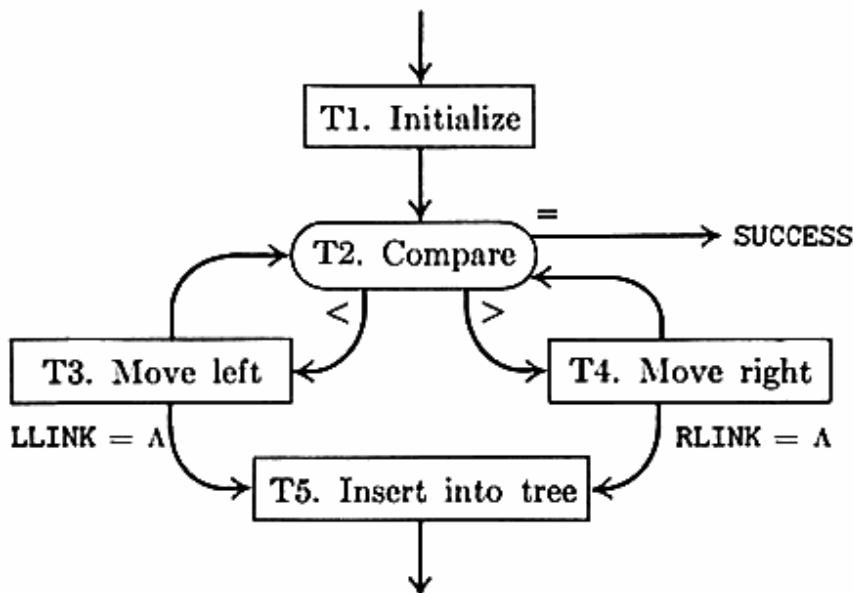


Fig. 11. Tree search and insertion.

On the average we have $C_1 = \frac{1}{2}(C + S)$, since $C_1 + C_2 = C$ and $C_1 - S \approx C_2$; 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 $2 \ln N \approx 1.386 \log_2 N$ 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

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

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

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

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

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

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

$$\begin{aligned} (N + 1)C'_N - NC'_{N-1} &= 2 + C'_{N-1}, \\ C'_N &= C'_{N-1} + 2/(N + 1). \end{aligned}$$

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

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

Applying (3) and simplifying yields the desired result

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

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 $K = \text{KEY}(P)$ in step T2, since we are sorting instead of searching. One solution is to treat $K = \text{KEY}(P)$ exactly as if $K > \text{KEY}(P)$; 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 made in both cases is NC_N . (However, Algorithm 5.2.2Q actually makes a few more comparisons, in order to speed up the inner loop.)

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** = Λ ; 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.

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 $Q \equiv \text{ROOT}$ or $Q \equiv \text{LLINK}(P)$ or $\text{RLINK}(P)$ in some node of the tree. This algorithm resets the value of Q in memory, to reflect the deletion.)

- D1. [Is RLINK null?] Set $T \leftarrow Q$. If $\text{RLINK}(T) = \Lambda$, set $Q \leftarrow \text{LLINK}(T)$ and go to D4.
- D2. [Find successor.] Set $R \leftarrow \text{RLINK}(T)$. If $\text{LLINK}(R) = \Lambda$, set $\text{LLINK}(R) \leftarrow \text{LLINK}(T)$, $Q \leftarrow R$, and go to D4.
- D3. [Find null LLINK .] Set $S \leftarrow \text{LLINK}(R)$. Then if $\text{LLINK}(S) \neq \Lambda$, set $R \leftarrow S$ and repeat this step until $\text{LLINK}(S) = \Lambda$. (At this point S will be equal to $Q\$$, the symmetric successor of Q .) Finally, set $\text{LLINK}(S) \leftarrow \text{LLINK}(T)$, $\text{LLINK}(R) \leftarrow \text{RLINK}(S)$, $\text{RLINK}(S) \leftarrow \text{RLINK}(T)$, $Q \leftarrow S$.
- D4. [Free the node.] Set $\text{AVAIL} \leftarrow T$ (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 $\text{RLINK}(T) \neq \Lambda$, $\text{LLINK}(T) = \Lambda$; 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 long sequence of random deletions and insertions will make the tree get way out of balance, so that the efficiency estimates we have made will be invalid. But actually the trees do not 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{J} be a tree of n elements, and let $P(\mathfrak{J})$ be the probability that \mathfrak{J} occurs if its keys are inserted in random order by Algorithm T. Some trees are more probable than others. Let $Q(\mathfrak{J})$ be the probability that \mathfrak{J} 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 $P(\mathfrak{J})$, we assume that the $n!$ permutations of the keys are equally likely; in calculating $Q(\mathfrak{J})$, 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 $P(\mathfrak{J}) = Q(\mathfrak{J})$ for all \mathfrak{J} .

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

Let $a_1 a_2 \dots a_{n+1}$ be a permutation of $\{1, 2, \dots, n+1\}$; we want to define the operation of deleting a_i , so as to obtain a permutation $b_1 b_2 \dots b_n$ of $\{1, 2, \dots, n\}$. 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, \dots, a_{n+1} and delete a_i , renumbering the keys from 1 to n , we obtain the tree constructed from $b_1 b_2 \dots b_n$.

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 “RLINK(a_i) = A.”) 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

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

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

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

If $i < j$, the deletion is

$$b'_1 \dots b'_{i-1} (\textcircled{b}_i) b'_{i+1} \dots b'_{j-1} (b_i + 1) b'_j \dots b'_n. \quad (7)$$

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

$$b'_1 \dots b'_{i-1} (\textcircled{b}_j) b'_i \dots b'_n; \quad (8)$$

this deletion fits the definition of Case 1.

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

$$b'_1 \dots b'_{i-1} (\textcircled{n+1}) b'_i \dots b'_n. \quad (9)$$

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
$j = 1$	⑤ 3 1 4 2	4 ③ 1 5 2	4 1 ③ 5 2	4 1 5 ③ 2	4 1 5 2 ③
$j = 2$	③ 4 1 5 2	3 ⑤ 1 4 2	4 2 ① 5 3	4 2 5 ① 3	4 2 5 3 ①
$j = 3$	③ 1 4 5 2	4 ① 2 5 3	3 1 ⑤ 4 2	3 1 5 ④ 2	3 1 5 2 ④
$j = 4$	③ 1 5 4 2	4 ① 5 2 3	3 1 ④ 5 2	3 1 4 ⑤ 2	4 1 5 3 ②
$j = 5$	③ 1 5 2 4	4 ① 5 3 2	3 1 ④ 2 5	4 1 5 ② 3	3 1 4 2 ⑤

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 $b_1 b_2 \dots b_n$. Since only $(n + 1)^2 n!$ deletions are possible, we must have found all of them. ■

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

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

$$\frac{1}{2} + \frac{\frac{1}{2} - H_N}{N} \quad (10)$$

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

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

D1½. [Is $\text{LLINK}(T) = \Lambda$?] If $\text{LLINK}(T) = \Lambda$, set $Q \leftarrow \text{RLINK}(T)$ 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. Thus, a sequence of insertions and deletions using this modification of Algorithm D will result in trees which are actually *better* than the theory of random trees would predict: the average computation time for search and insertion will tend to decrease as time goes on.

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 k th element inserted, where $p_1 + \dots + p_N = 1$. 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

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

(Cf. Eq. (5).)

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

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

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.

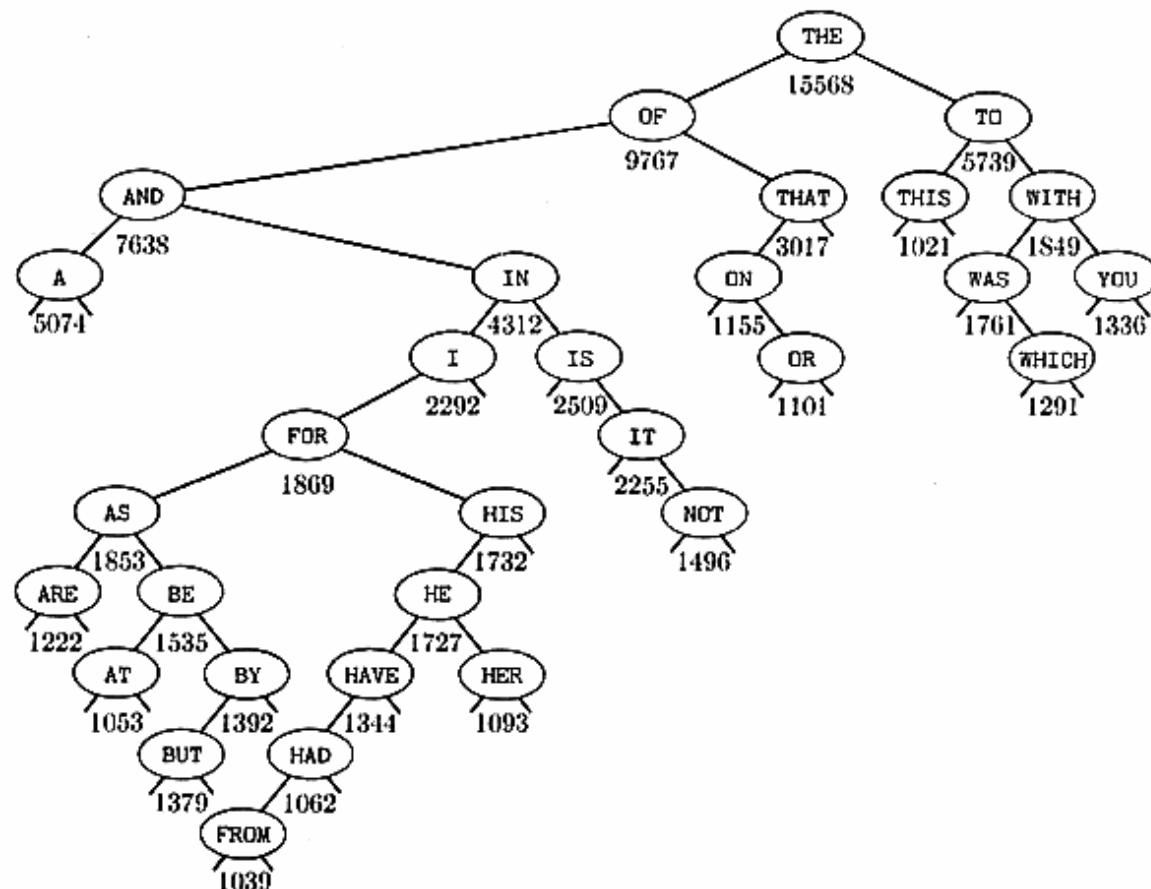


Fig. 12. The 31 most common English words, inserted in decreasing order of frequency.

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.

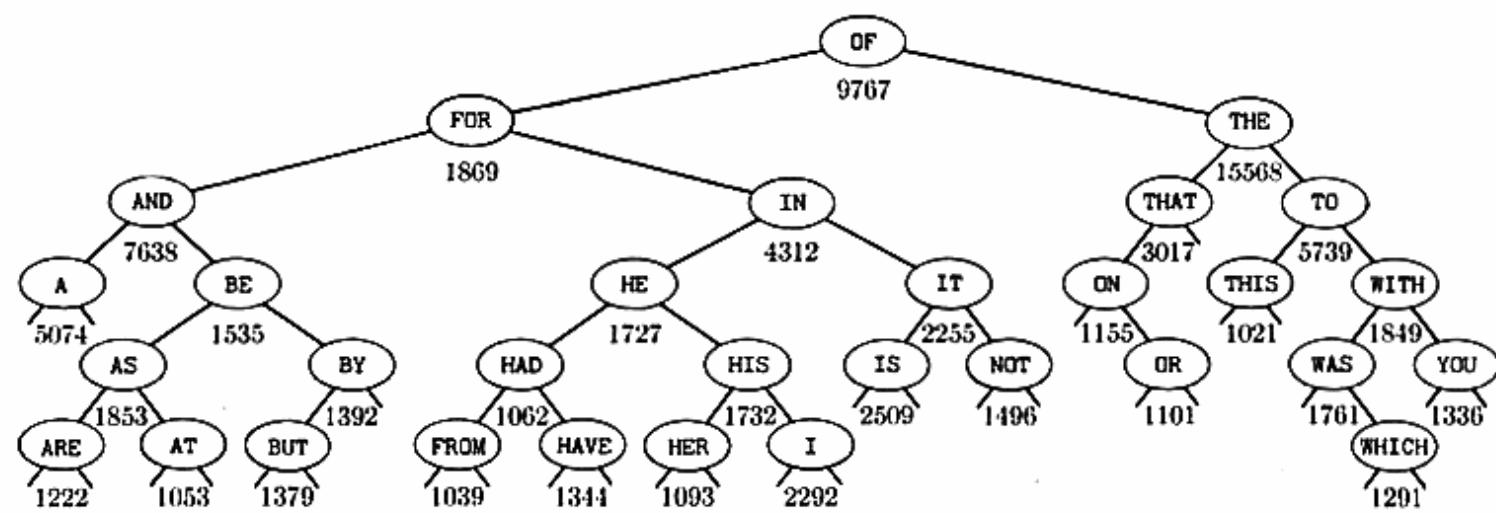


Fig. 13. Optimum search tree for the 31 most common English words.

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:

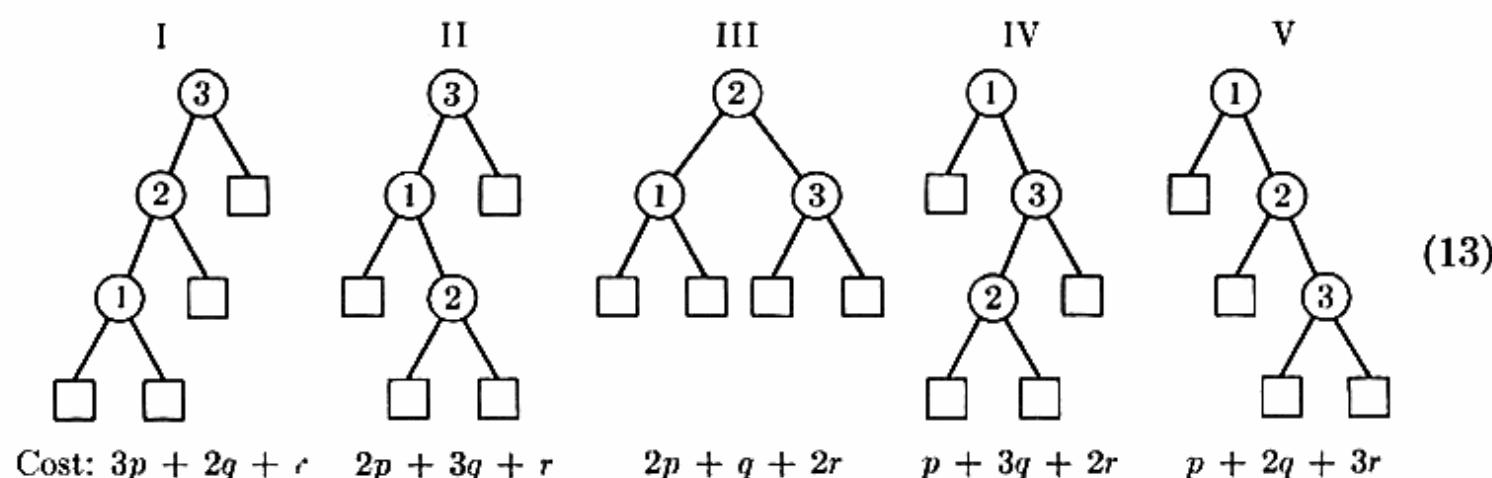


Figure 14 shows the ranges of p, q, r for which each tree is optimum; the balanced tree is best about 45 percent of the time, if we choose p, q, r at random (see exercise 21).

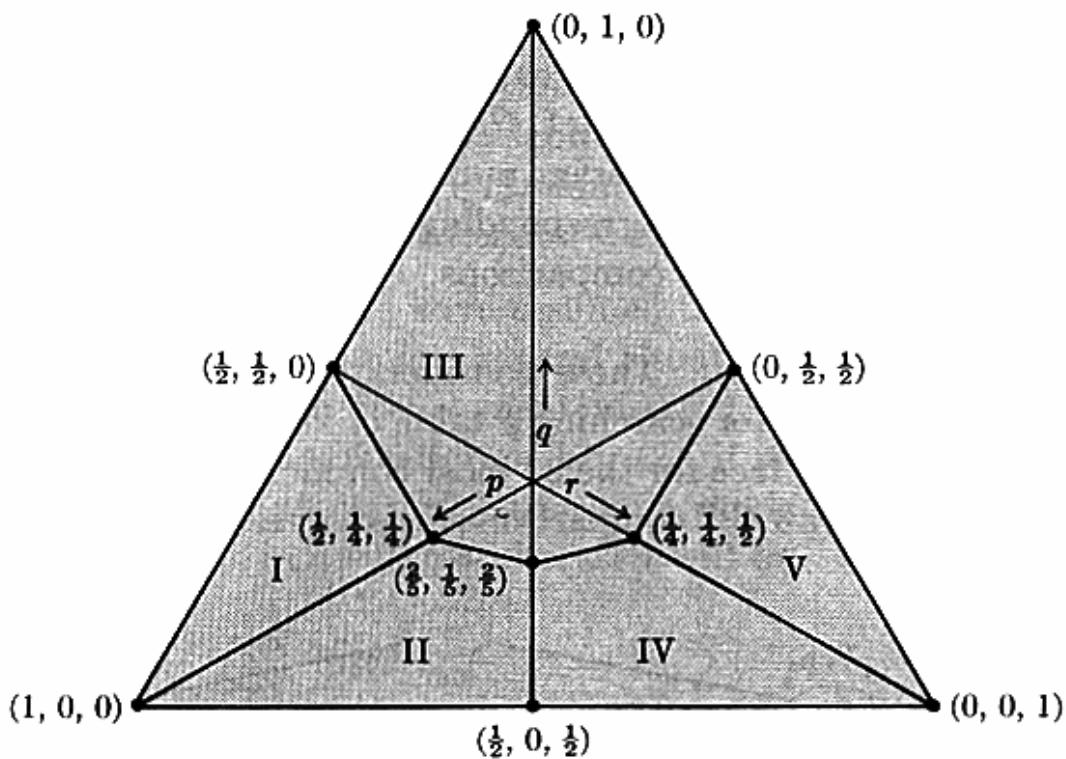


Fig. 14. If the relative frequencies of (K_1, K_2, K_3) are (p, q, r) , this graph shows which of the five trees in (13) is best. The fact that $p + q + r = 1$ makes the graph two-dimensional although there are three coordinates.

Unfortunately, when N is large there are

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

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, \dots, p_n and q_0, q_1, \dots, q_n , where

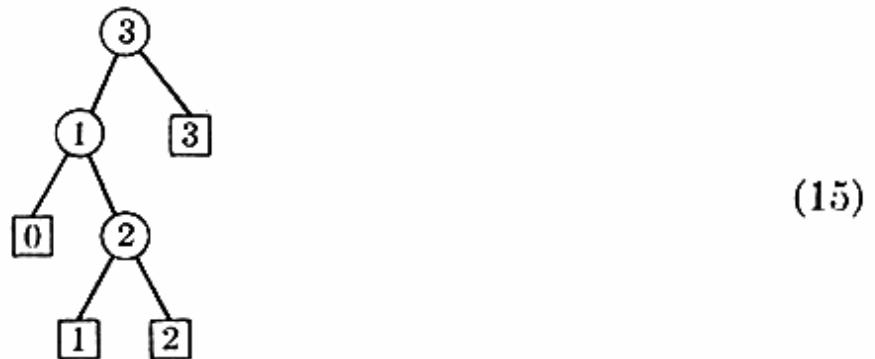
p_i = probability that K_i is the search argument;

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

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

where \circled{j} is the j th 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



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" $(p_1, \dots, p_n; q_0, \dots, q_n)$.

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, \dots, 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 $(p_{i+1}, \dots, p_j; q_i, \dots, q_j)$; and let $w(i, j) = p_{i+1} + \dots + p_j + q_i + \dots + q_j$ be the sum of all those weights; $c(i, j)$ and $w(i, j)$ are defined for $0 \leq i \leq j \leq n$. It follows that

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

since the minimum possible cost of a tree with root \circled{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 $j - i = 1, 2, 3, \dots, n$; 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

$$r(i, j - 1) \leq r(i, j) \leq r(i + 1, j) \quad (17)$$

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

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

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 $(p_1, \dots, p_n; q_0, \dots, q_n)$, this algorithm constructs binary trees $t(i, j)$ which have minimum cost for the weights $(p_{i+1}, \dots, p_j; q_i, \dots, q_j)$ in the sense defined above. Three arrays are computed, namely

$$\begin{aligned} c[i, j], &\quad \text{for } 0 \leq i \leq j \leq n, && \text{the cost of } t(i, j); \\ r[i, j], &\quad \text{for } 0 \leq i \leq j \leq n, && \text{the root of } t(i, j); \\ w[i, j], &\quad \text{for } 0 \leq i \leq j \leq n, && \text{the total weight of } t(i, j). \end{aligned}$$

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 $0 \leq i \leq n$, 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 $j = i + 1, \dots, n$. Then for $1 \leq j \leq n$ 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 $d = 2, 3, \dots, n$, 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 $j = d, d + 1, \dots, n$.

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

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

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

As an example of Algorithm K, consider Fig. 15, which is based on a “key-word-in-context” (KWIC) indexing application. The titles of all articles in the first ten volumes of the 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.)

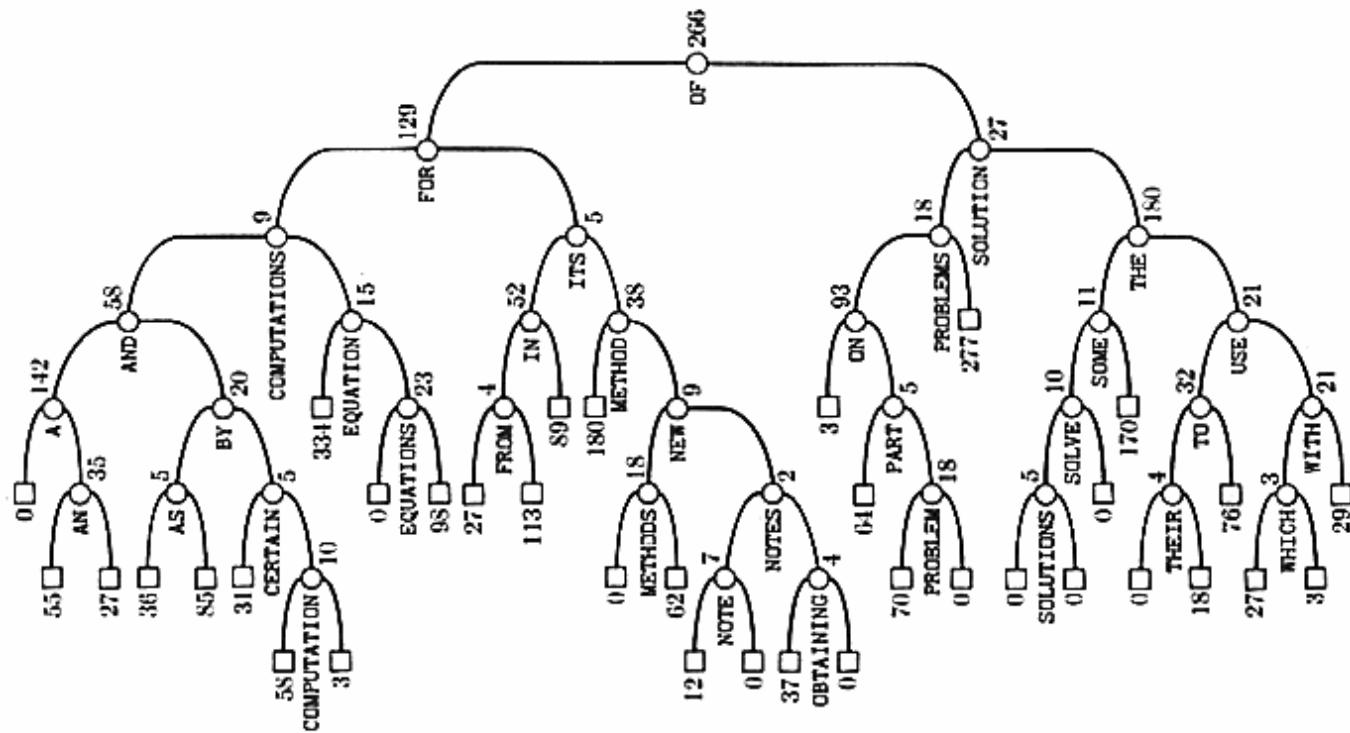


Fig. 15. An optimum binary search tree for a KWIC indexing application.

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 $j = 1, 2, \dots, 35$ are $(1, 1, 2, 3, 3, 3, 3, 8, 8, 8, 8, 8, 8, 11, 11, \dots, 11, 21, 21, 21, 21, 21)$; the values of $r[i, 35]$ for $i = 0, 1, \dots, 34$ are $(21, 21, \dots, 21, 25, 25, 25, 25, 25, 26, 26, 26, 30, 30, 30, 30, 30, 30, 33, 33, 35)$.

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

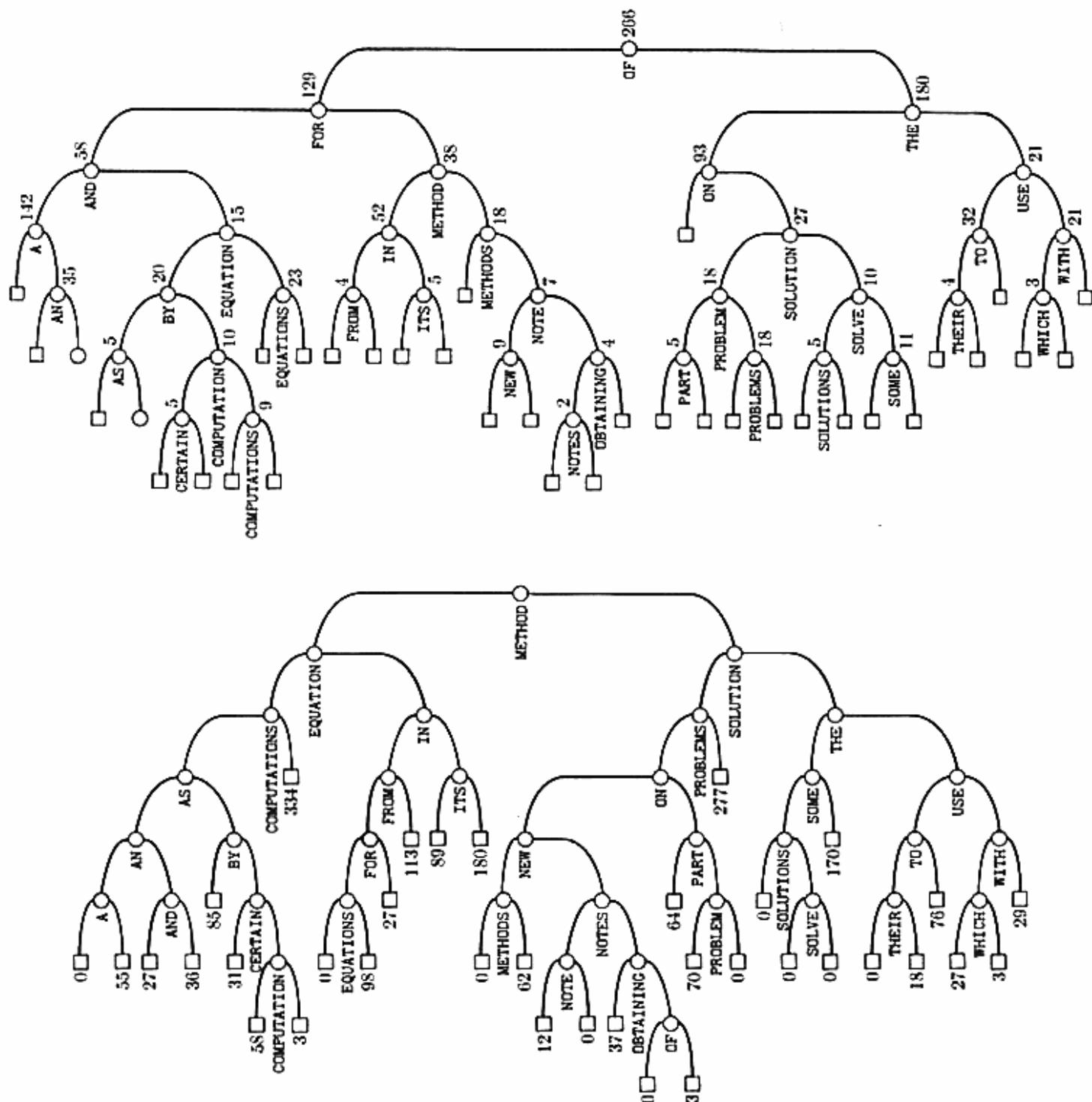


Fig. 16. Optimum binary search trees based on half of the data of Fig. 15; (a) external frequencies suppressed, (b) internal frequencies suppressed.

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)]: 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 $c(0, k - 1) + c(k, n) + w(0, n)$ 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 .

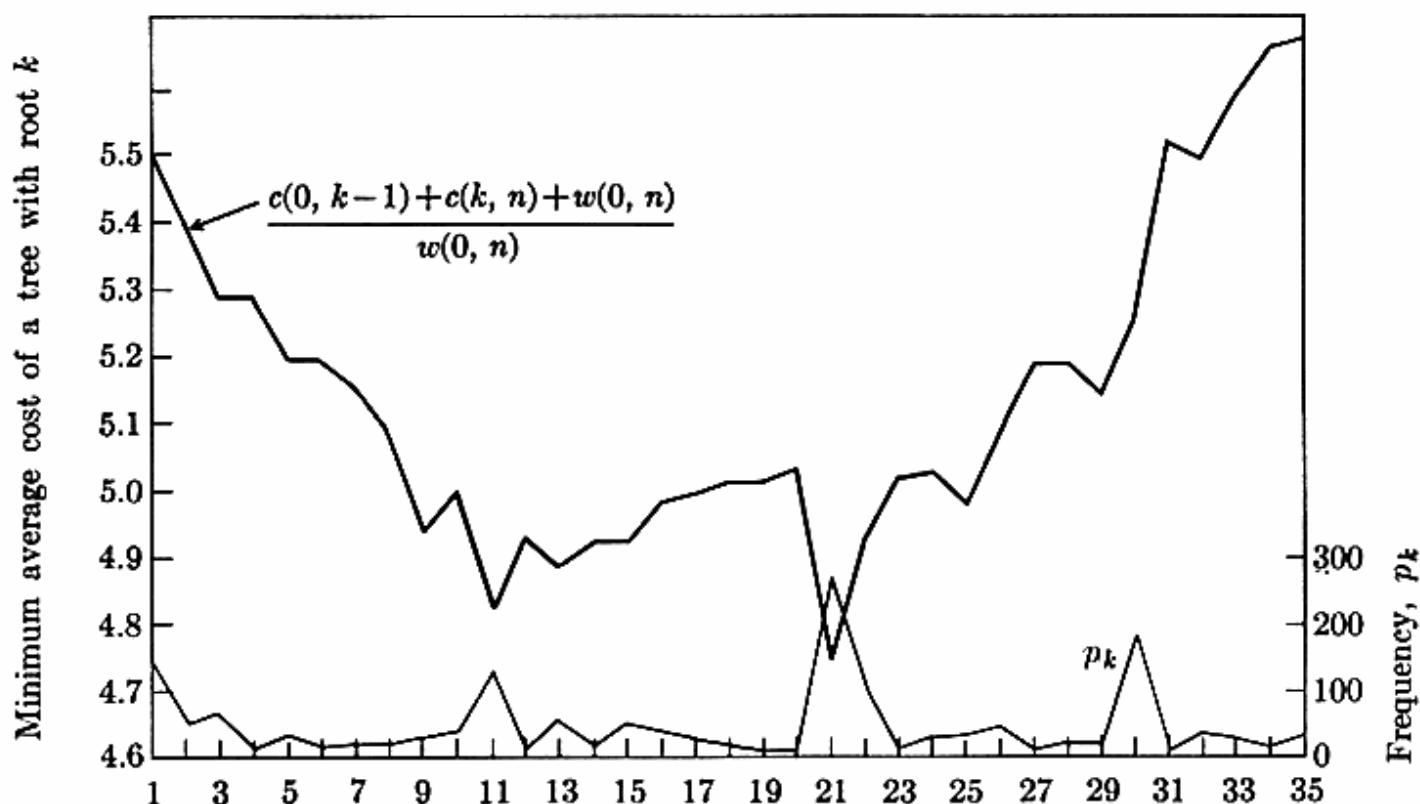


Fig. 17. Behavior of the cost as a function of the root, k .

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,

$$\boxed{q_0} \quad \boxed{q_1} \quad \boxed{q_2} \quad \dots \quad \boxed{q_n}. \quad (18)$$

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

$$\textcircled{q_i + q_j}. \quad (19)$$

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

$$\begin{array}{ccccccc} \boxed{q_0} & & \boxed{q_1} & & \boxed{q_2} & & \dots & \boxed{q_n} \\ l_1 & & l_2 & & l_3 & & & l_n \end{array}.$$

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, \dots, 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 $\text{U}, \text{A}, \text{B}, \dots, \text{Z}$ in English text. During Phase 1, the

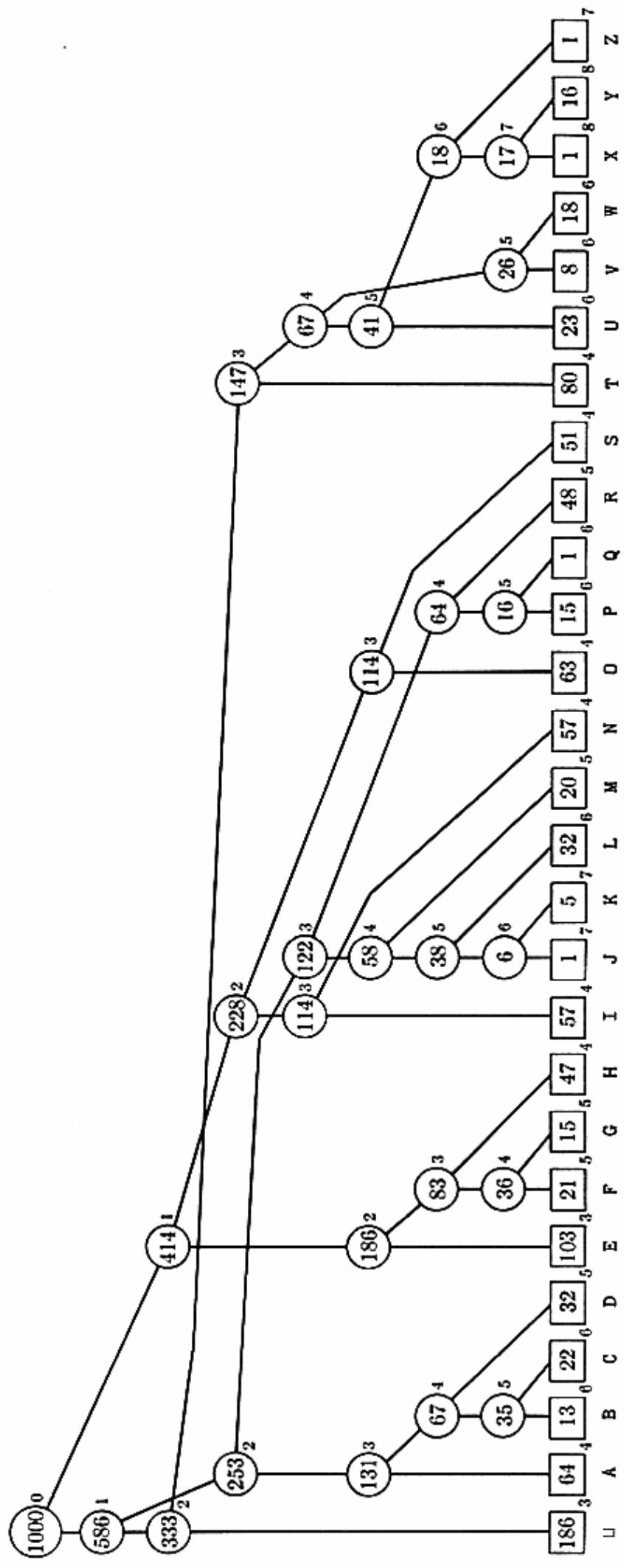


Fig. 18. The Hu-Tucker algorithm applied to alphabetic frequency data: Phases 1 and 2.

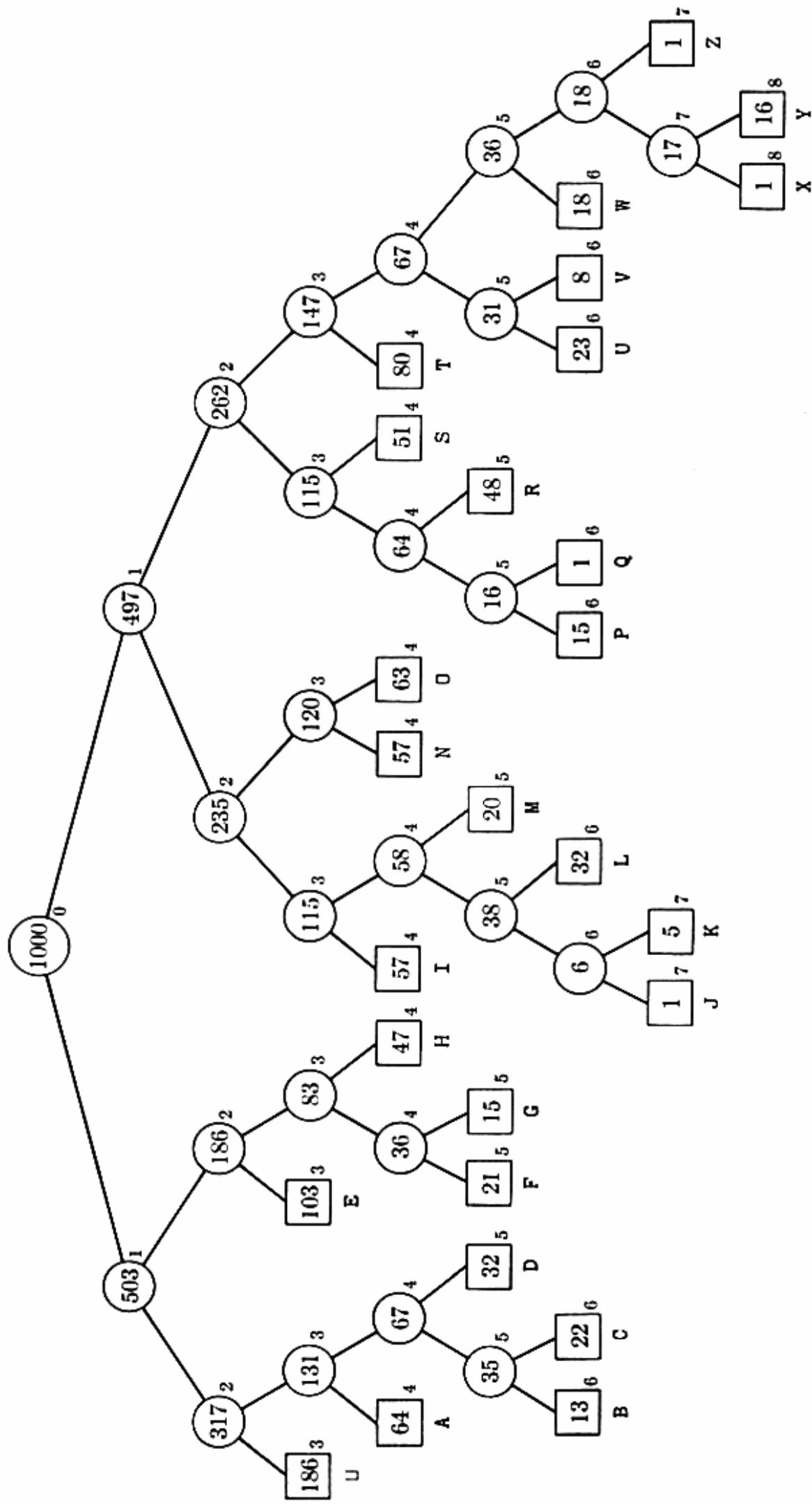


Fig. 19. The Hu-Tucker algorithm applied to alphabetic frequency data: Phase 3.

first node formed is ⑥, combining the J and K frequencies; then the node ⑯ is formed (combining P and Q), then

⑯, ⑰, ⑱, ⑲, ⑳, ㉑, ㉒, ㉓, ㉔, ㉕, ㉖, ㉗, ㉘, ㉙, ㉚;

at this point we have the working sequence

186 64 67 103 83 57 58 57 63 64 51 80 67. (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.

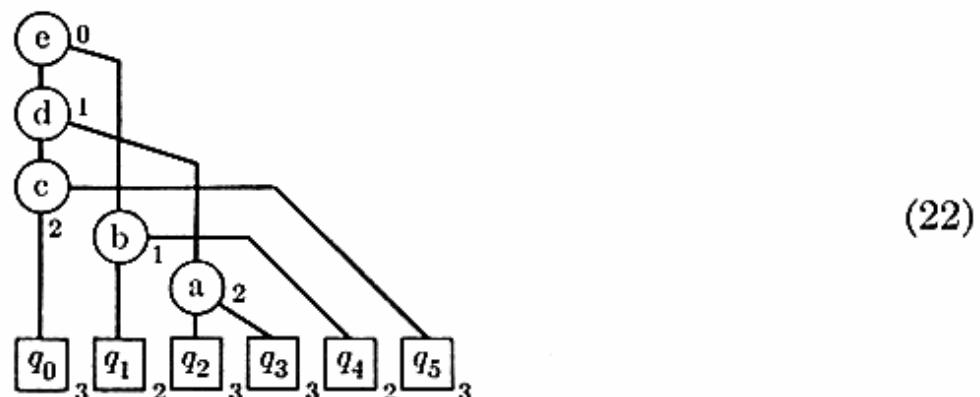
The level numbers assigned during Phase 2 appear at the right of each node in Fig. 18. The recombination during Phase 3 now yields the tree shown in Fig. 19; note that things must be associated differently in this tree than in Fig. 18, because Fig. 18 does not preserve the left-to-right ordering. But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



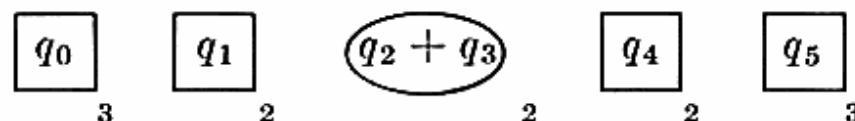
This example shows that the two smallest weights, 2 and 3, should *not* always be combined in an optimum tree, even when they are adjacent; some recombination phase is needed.

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



by combining nodes (a), (b), (c), (d), (e) in this order; this accords with rule

(i). Then Phase 3 will get stuck after forming



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,

64	64	57	57	57	51	51	67	
186	67	83	57	63	63	80	80	(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 $n \rightarrow \infty$. 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:

U	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
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

110011000011010111111000010111010.

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 $p_1 = p_2 = \dots = p_n = 0$, the weighted path length of an optimum binary search tree lies between*

$$\sum_{0 \leq i \leq n} q_i \log_2 (Q/q_i) \quad \text{and} \quad 2Q + \sum_{0 \leq i \leq n} q_i \log_2 (Q/q_i),$$

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

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

$$Q + \sum_{0 \leq i < k} q_i \log_2 (Q_1/q_i) + \sum_{k \leq i \leq n} q_i \log_2 ((Q - Q_1)/q_i) \geq \sum_{0 \leq i \leq n} q_i \log_2 (Q/q_i) + f(Q_1),$$

for some k , where

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

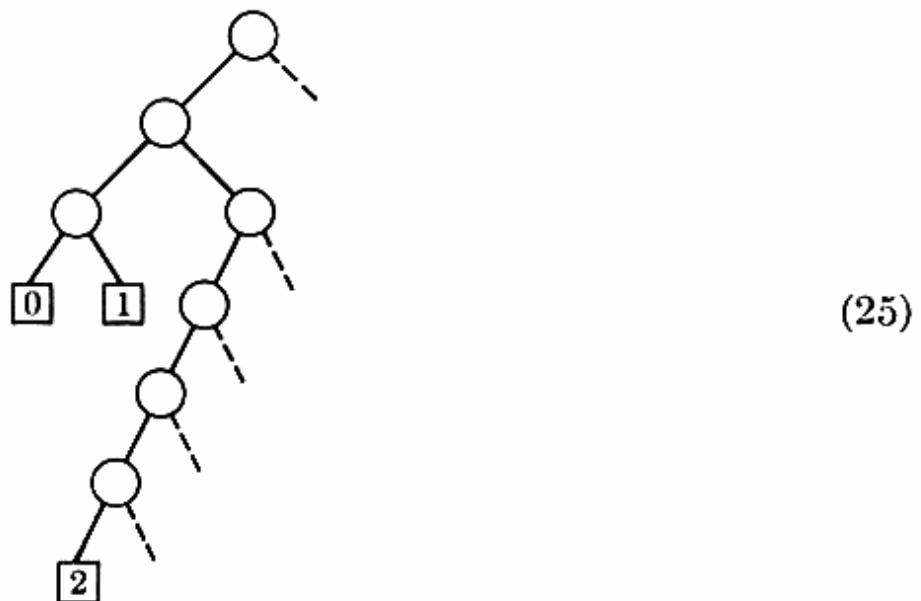
and

$$f(Q_1) = Q + Q_1 \log_2 Q_1 + (Q - Q_1) \log_2 (Q - Q_1) - Q \log_2 Q.$$

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, \dots, e_n be integers such that $2^{-e_i} \leq q_i < 2^{1-e_i}$, for $0 \leq i \leq n$. Construct codewords C_i of 0's and 1's, by using the most significant $e_i + 1$ binary digits of the fraction $\sum_{0 \leq k < i} q_k + \frac{1}{2}q_i$, expressed in binary notation. Exercise 35 proves that C_i is never an initial substring of C_j when $i \neq j$; 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



(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

$$\leq \sum_{0 \leq i \leq n} (e_i + 1)q_i < \sum_{0 \leq i \leq n} q_i(2 + \log(1/q_i)). \blacksquare$$

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 $\sum_{0 \leq i \leq n} q_i \log_2(Q/q_i)$, 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].

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 [*J ACM* **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 $p_1 = \dots = p_n = 0$, 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. [2] 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 $0 \leq k < n$; then we use Algorithm T to insert the remaining keys. What is the average number of comparisons in a successful search? [Hint: Modify Eq. (2).]

► 5. [M25] There are $11! = 39,916,800$ different orders in which the names CAPRICORN, AQUARIUS, etc. could have been inserted into a binary search tree. (a) How many of these arrangements will produce Fig. 10? (b) How many of these arrangements will produce a degenerate tree, in which LLINK or RLINK is Λ in each node?

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

- 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.]
- 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.
- 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 m th 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 $S \leftarrow \text{LLINK}(R)$ 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. [M47] When the new step D1 $\frac{1}{2}$ added to Algorithm D, exactly how much is the average running time of Algorithm T affected after a long sequence of insertions and deletions?

- 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 $p_1 \geq p_2 \geq \dots \geq p_N$. 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 $p_1 = 5, p_2 = p_3 = \dots = p_{40} = 1, q_0 = q_1 = \dots = q_{40} = 0$. (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 $(p_1, \dots, p_n; q_0, \dots, q_n)$ may be obtained by replacing
- by
- in any optimum tree for $(p_1, \dots, p_{n-1}; q_0, \dots, q_{n-1})$.
25. [M20] Let A and B be nonempty sets of real numbers, and define $A \leq B$ if the following property holds:
- $$(a \in A, b \in B, \text{ and } b < a) \quad \text{implies} \quad (a \in B \text{ and } b \in A).$$
- (a) Prove that this relation is transitive on nonempty sets. (b) Prove or disprove: $A \leq B$ if and only if $A \leq A \cup B \leq B$.
26. [M22] Let $(p_1, \dots, p_n; q_0, \dots, q_n)$ be nonnegative weights, where $p_n + q_n = x$. Prove that as x varies from 0 to ∞ , while $(p_1, \dots, p_{n-1}; q_0, \dots, q_{n-1})$ 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 $l_0 > l_1 > \dots > l_m$ and real constants $0 = x_0 < x_1 < \dots < x_m < x_{m+1} = \infty$ and $y_0 < y_1 < \dots < y_m$ 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

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

in terms of the relation defined in exercise 25, whenever the weights $(p_1, \dots, p_n; q_0, \dots, q_n)$ are nonnegative. The proof is by induction on $j - i$; our task is to prove

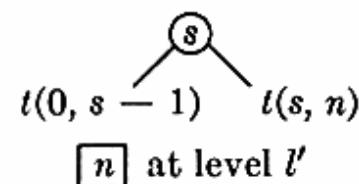
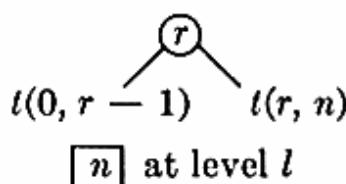
that $R(0, n - 1) \leq R(0, n)$, assuming that $n \geq 2$ and that the above relation holds for $j - i < n$. [By left-right symmetry it follows that $R(0, n) \leq R(1, n)$.]

- Prove that $R(0, n - 1) \leq R(0, n)$ if $p_n = q_n = 0$. (See exercise 24.)
- Let $p_n + q_n = x$. In the notation of exercise 26, let R_h be the set $R(0, n)$ of optimum roots when $x_h < x < x_{h+1}$, and let R'_h be the set of optimum roots when $x = x_h$. Prove that

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

Hence by part (a) and exercise 25 we have $R(0, n - 1) \leq R(0, n)$ for all x .

[Hint: Consider the case $x = x_h$, and assume that both the trees



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

28. [24] Use some macro-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 $c(i, j) + c(i + 1, j - 1) \geq c(i, j - 1) + c(i + 1, j)$.

31. [M20] (a) If the weights (q_0, \dots, 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 $i \neq j$.

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

$$2S + q_0 \log_2 (S/q_0) + \sum_{1 \leq i \leq n} (p_i + q_i) \log_2 (S/(p_i + q_i)),$$

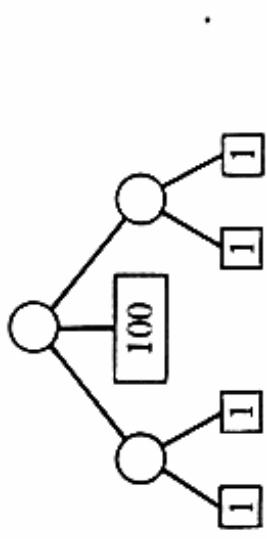
where $S = q_0 + \sum_{1 \leq i \leq n} (p_i + q_i)$ is the total weight.

► 37. [M25] (T. C. Hu and K. C. Tan.) Let $n + 1 = 2^m + k$, where $0 \leq k \leq 2^m$. 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, \dots, q_n) if we apply the Hu-Tucker algorithm to the weights $(M + q_0, \dots, M + q_n)$ for sufficiently large M .

38. [M35] (K. C. Tan.) Prove that, among all sets of probabilities $(p_1, \dots, p_n; q_0, \dots, q_n)$ with $p_1 + \dots + p_n + q_0 + \dots + q_n = 1$, the most expensive minimum-cost tree occurs when $p_i = 0$ for all even i , $q_j = 0$ for all even j , and $q_j = 1/\lceil n/2 \rceil$ for all odd j . [Hint: Given arbitrary probabilities $(p_1, \dots, p_n; q_0, \dots, q_n)$, let $c_0 = q_0$, $c_i = p_i + q_i$ for $1 \leq i \leq n$, and $S(0) = \emptyset$; and for $1 \leq r \leq \lceil n/2 \rceil$ 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 $q = \lceil \log_2 n \rceil$. Prove that the cost of this tree is $\leq f(n)$, 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 $0 \leq k < n$. Prove that the number of comparisons made while constructing such a tree is $N \log_2 N + O(N)$ 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 $5N \log_2 N$, 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'skii and E. M. Landis [*Doklady Akademii Nauk SSSR* **146** (1962), 263–266; English translation in *Soviet Math. J.* **3**, 1259–1263]. Their method requires only two extra bits per node, and it never uses more than $O(\log N)$ operations to search

the tree or to insert an item. In fact, we shall see that their approach also leads to a general technique that is good for representing arbitrary *linear lists* of length N , so that each of the following operations can be done in only $O(\log N)$ units of time:

- i) Find an item having a given key.
- ii) Find the k th 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 $20 \log_2 N$ units of time and an inefficient method that takes $2N$ units of time, we should use the inefficient method unless N is greater than 1024. 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 +, -, 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*

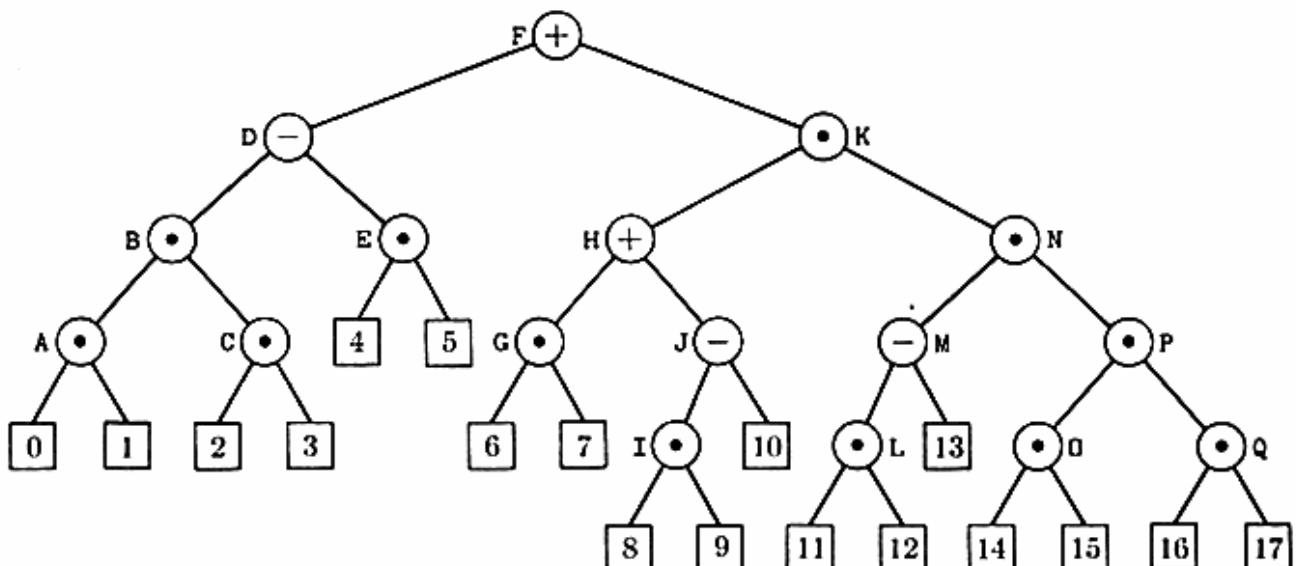


Fig. 20. A balanced binary tree.

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 $\log_2(N + 1)$ and $1.4404 \log_2(N + 2) - 0.328$.*

Proof. A binary tree of height h obviously cannot have more than 2^h external nodes; so $N + 1 \leq 2^h$, that is, $h \geq \lceil \log_2(N + 1) \rceil$ 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

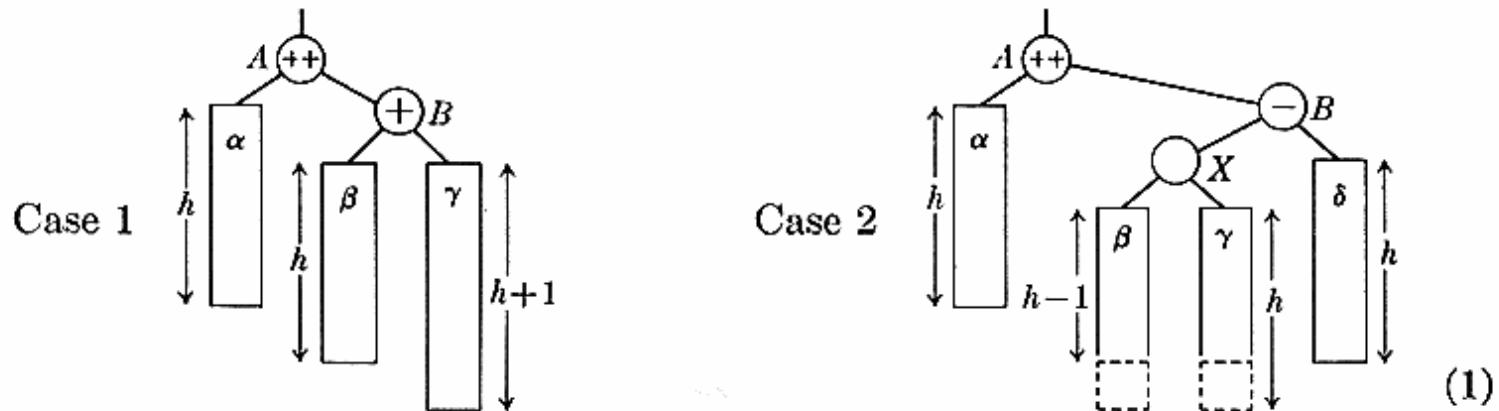
$$N \geq F_{h+2} - 1 > \phi^{h+2}/\sqrt{5} - 2,$$

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.

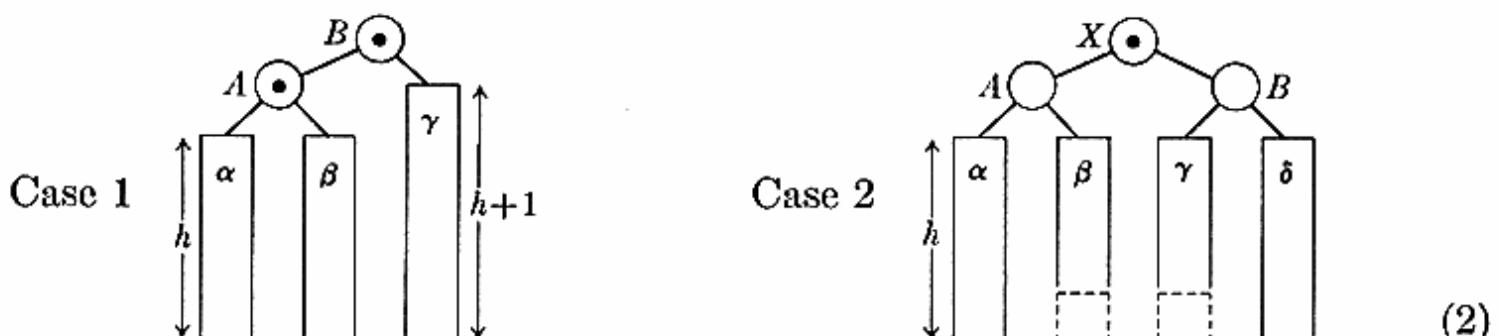
Consider now what happens when a new node is inserted into a balanced tree using tree insertion (Algorithm 6.2.2T). In Fig. 20, the tree will still be balanced if the new node takes the place of 4, 5, 6, 7, 10, or 13, but some adjustment will be needed if the new node falls elsewhere. The problem arises when we have a node with a balance factor of +1 whose right subtree got higher after the insertion; or, dually, if the balance factor is -1 and the left

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



(Two other essentially identical cases occur if we reflect these diagrams, interchanging left and right.) In these diagrams the large rectangles $\alpha, \beta, \gamma, \delta$ 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:



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 [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.

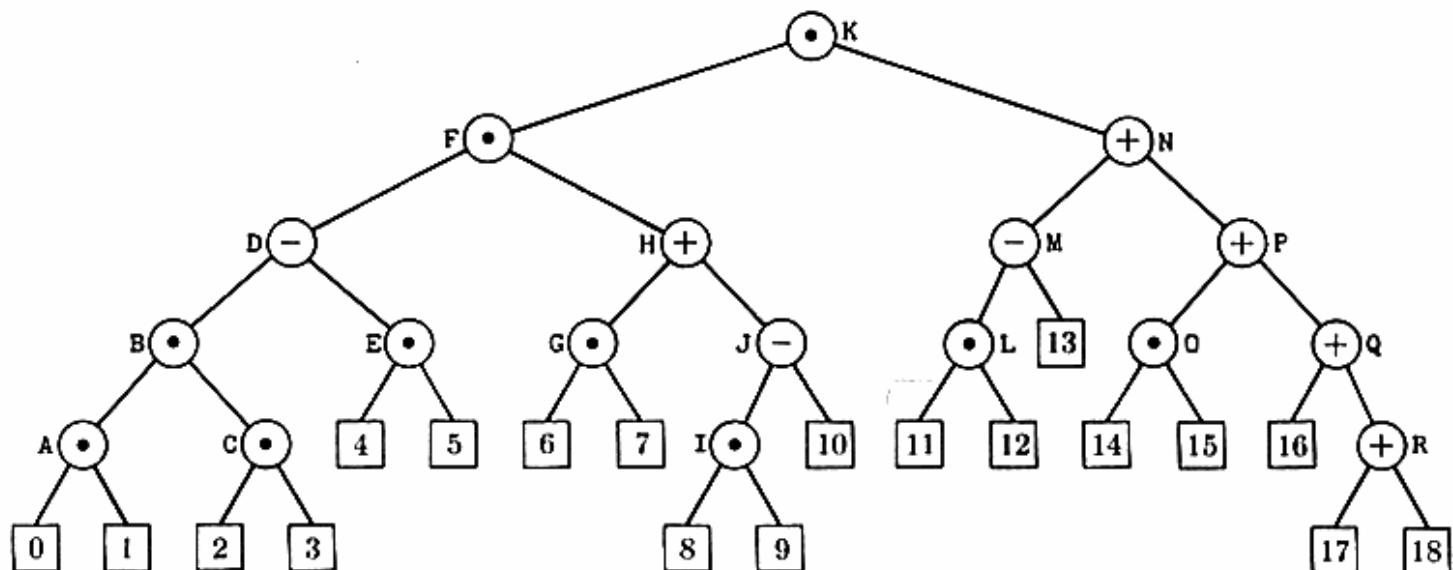


Fig. 21. The tree of Fig. 20, rebalanced after a new key R has been inserted.

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

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

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 $\text{LINK}(a, P)$ as a synonym for $\text{LLINK}(P)$ if $a = -1$, and for $\text{RLINK}(P)$ if $a = +1$.

- A1. [Initialize.] Set $T \leftarrow \text{HEAD}$, $S \leftarrow P \leftarrow \text{RLINK}(\text{HEAD})$. (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 $K < \text{KEY}(P)$, go to A3; if $K > \text{KEY}(P)$, go to A4; and if $K = \text{KEY}(P)$, the search terminates successfully.
- A3. [Move left.] Set $Q \leftarrow \text{LLINK}(P)$. If $Q = \Lambda$, set $Q \leftarrow \text{AVAIL}$ and $\text{LLINK}(P) \leftarrow Q$ 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 $Q \leftarrow \text{RLINK}(P)$. If $Q = \Lambda$, set $Q \leftarrow \text{AVAIL}$ and $\text{RLINK}(P) \leftarrow Q$ 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.)

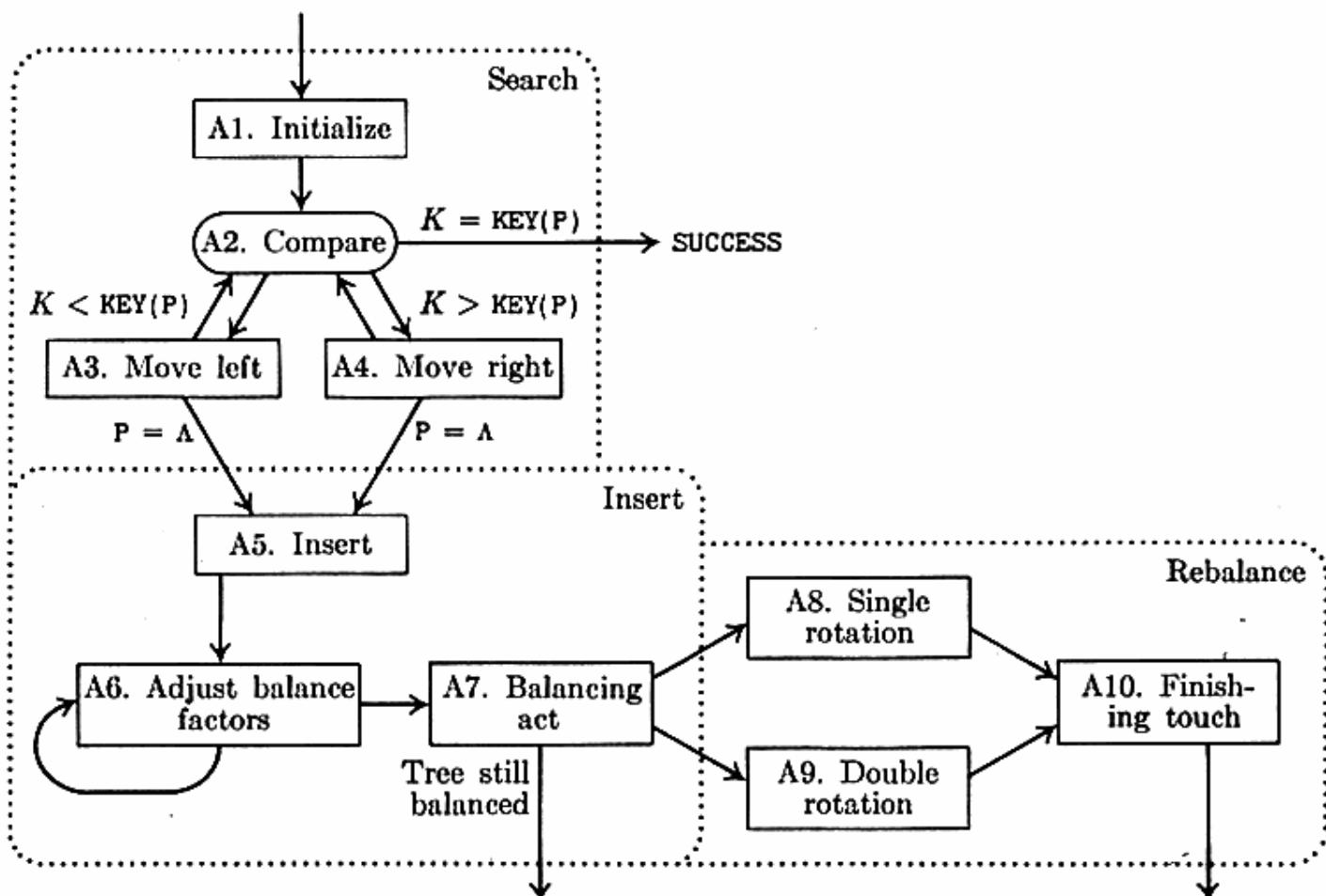


Fig. 22. Balanced tree search and insertion.

- A5. [Insert.] (We have just linked a new node, $\text{NODE}(Q)$, into the tree, and its fields need to be initialized.) Set $\text{KEY}(Q) \leftarrow K$, $\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$, $B(Q) \leftarrow 0$.
- A6. [Adjust balance factors.] (Now the balance factors on nodes between S and Q need to be changed from zero to ± 1 .) If $K < \text{KEY}(S)$, set $R \leftarrow P \leftarrow \text{LLINK}(S)$, otherwise set $R \leftarrow P \leftarrow \text{RLINK}(S)$. Then repeatedly do the following operation zero or more times until $P = Q$: If $K < \text{KEY}(P)$ set $B(P) \leftarrow -1$ and $P \leftarrow \text{LLINK}(P)$; if $K > \text{KEY}(P)$, set $B(P) \leftarrow +1$ and $P \leftarrow \text{RLINK}(P)$. (If $K = \text{KEY}(P)$, then $P = Q$ and we may go on to the next step.)
- A7. [Balancing act.] If $K < \text{KEY}(S)$ set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Several cases now arise:
- If $B(S) = 0$ (the tree has grown higher), set $B(S) \leftarrow a$, $\text{LLINK}(\text{HEAD}) \leftarrow \text{LLINK}(\text{HEAD}) + 1$, and terminate the algorithm.
 - If $B(S) = -a$ (the tree has gotten more balanced), set $B(S) \leftarrow 0$ and terminate the algorithm.
 - 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 $\text{LINK}(-a, S)$ points to α , etc.)

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

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

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

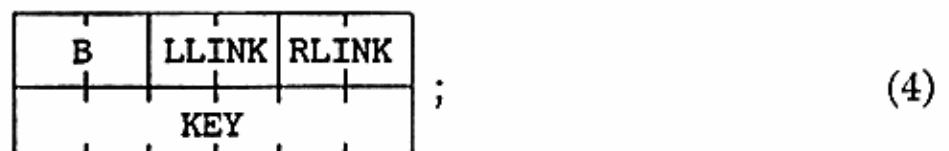
and then set $B(P) \leftarrow 0$.

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 $S = \text{RLINK}(T)$ then set $\text{RLINK}(T) \leftarrow P$, otherwise set $\text{LLINK}(T) \leftarrow P$. ■

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



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

01	B	EQU	0:1	
02	LLINK	EQU	2:3	
03	RLINK	EQU	4:5	
04	START	LDA	K	A1. Initialize. $T \leftarrow \text{HEAD}$.
05		ENT5	HEAD	
06		LD2	0.5(RLINK)	
07		JMP	2F	
08	4H	LD2	0.1(RLINK)	A4. Move right. $Q \leftarrow \text{RLINK}(P)$.
09		J2Z	5F	
10	1H	LDX	0.2(B)	To A5 if $Q = \Lambda$. $rX \leftarrow B(Q)$.
11		JXZ	*+3	
12		ENT5	0.1	Jump if $B(Q) = 0$. $T \leftarrow P$.
13	2H	ENT4	0.2	
14		ENT1	0.2	$S \leftarrow Q$. $P \leftarrow Q$.
15		CMPA	1,1	A2. Compare. To A4 if $K > \text{KEY}(P)$.
16		JG	4B	
17		JE	SUCCESS	Exit if $K = \text{KEY}(P)$. A3. Move left. $Q \leftarrow \text{LLINK}(P)$.
18		LD2	0.1(LLINK)	
19		J2NZ	1B	Jump if $Q \neq \Lambda$. A5. Insert.
20-29		5H	(copy here lines 14–23 of Program 6.2.2T)	

30	6H	CMPA	1,4		1 - S	A6. Adjust balance factors.
31		JL	*+3		1 - S	Jump if $K < \text{KEY}(S)$.
32		LD3	0,4(RLINK)		E	$R \leftarrow \text{RLINK}(S)$.
33		JMP	*+2		E	
34		LD3	0,4(LLINK)		1 - S - E	$R \leftarrow \text{LLINK}(S)$.
35		ENT1	0,3		1 - S	$P \leftarrow R$.
36		ENTX	-1		1 - S	$rX \leftarrow -1$.
37		JMP	1F		1 - S	To comparison loop.
38	4H	JE	7F		F2 + 1 - S	To A7 if $K = \text{KEY}(P)$.
39		STX	0,1(1:1)		F2	$B(P) \leftarrow +1$ (it was +0).
40		LD1	0,1(RLINK)		F2	$P \leftarrow \text{RLINK}(P)$.
41	1H	CMPA	1,1		F + 1 - S	
42		JGE	4B		F + 1 - S	Jump if $K \geq \text{KEY}(P)$.
43		STX	0,1(B)		F1	$B(P) \leftarrow -1$.
44		LD1	0,1(LLINK)		F1	$P \leftarrow \text{LLINK}(P)$.
45		JMP	1B		F1	To comparison loop.
46	7H	LD2	0,4(B)		1 - S	A7. Balancing act. $rI2 \leftarrow B(S)$.
47		STZ	0,4(B)		1 - S	$B(S) \leftarrow 0$.
48		CMPA	1,4		1 - S	
49		JG	A7R		1 - S	To $a = +1$ routine if $K > \text{KEY}(S)$.
50	A7L	J2P	DONE	68	A7R	J2N DONE
51		J2Z	7F	69		J2Z 6F
52		ENT1	0,3	70		ENT1 0,3
53		LD2	0,3(B)	71		LD2 0,3(B)
54		J2N	A8L	72		J2P A8R
55	A9L	LD1	0,3(RLINK)	73	A9R	LD1 0,3(LLINK)
56		LDX	0,1(LLINK)	74		LDX 0,1(RLINK)
57		STX	0,3(RLINK)	75		STX 0,3(LLINK)
58		ST3	0,1(LLINK)	76		ST3 0,1(RLINK)
59		LD2	0,1(B)	77		LD2 0,1(B)
60		LDX	T1,2	78		LDX T2,2
61		STX	0,4(B)	79		STX 0,4(B)
62		LDX	T2,2	80		LDX T1,2
63		STX	0,3(B)	81		STX 0,3(B)
64	A8L	LDX	0,1(RLINK)	82	A8R	LDX 0,1(LLINK)
65		STX	0,4(LLINK)	83		STX 0,4(RLINK)
66		ST4	0,1(RLINK)	84		ST4 0,1(LLINK)
67		JMP	A8R1	85	A8R1	STZ 0,1(B)
86	A10	CMP4	0,5(RLINK)		G	A10. Finishing touch.
87		JNE	*+3		G	Jump if $\text{RLINK}(T) \neq S$.
88		ST1	0,5(RLINK)		G2	$\text{RLINK}(T) \leftarrow P$.
89		JMP	DONE		G2	Exit.
90		ST1	0,5(LLINK)		G1	$\text{LLINK}(T) \leftarrow P$.
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	$rX \leftarrow +1$.
97	7H	STX	0,4(B)		J	$B(S) \leftarrow a$.
98		LDX	HEAD(LLINK)		J	$\text{LLINK}(\text{HEAD})$
99		INCX	1		J	+1
100		STX	HEAD(LLINK)		J	$\rightarrow \text{LLINK}(\text{HEAD})$.
101	DONE	EQU	*		1 - S	Insertion is complete. ■

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

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

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

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

(See exercise 6.) Thus

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

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

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

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

$$B_0 = B_1 = 1, \quad B_{h+1} = B_h^2 + 2B_h B_{h-1}, \quad (7)$$

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,

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

where $A_0 = 1, A_1 = 3, A_2 = 5, A_3 = 7, A_4 = 23, A_5 = 347, \dots, A_h = A_{h-1}B_{h-2} + 2$. 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

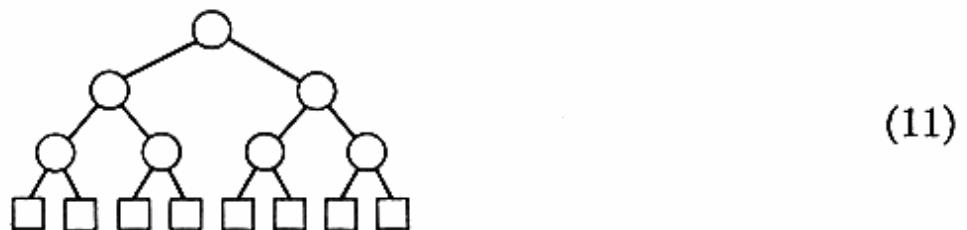
$$B_h = \lfloor \theta^{2^h} \rfloor - \lfloor \theta^{2^{h-1}} \rfloor + \lfloor \theta^{2^{h-2}} \rfloor - \cdots + (-1)^h \lfloor \theta^{2^0} \rfloor. \quad (9)$$

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

$$B'_h(1)/B_h(1) \approx (0.70118)2^h. \quad (10)$$

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

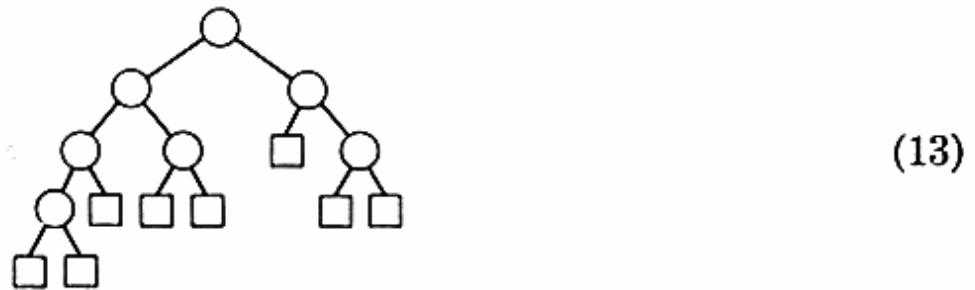
Unfortunately, none of these results have much to do with Algorithm A, since the mechanism of that algorithm makes some trees much more probable than others. For example, consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys can be inserted, and the perfectly balanced “complete” tree,



is obtained 2160 times. By contrast, the Fibonacci tree,



occurs only 144 times, and the similar tree,



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 $\log_2 N + c$ comparisons for some small constant c . Empirical tests support this conjecture: The average number of comparisons needed to insert the N th item seems to be approximately $\log_2 N + 0.25$ 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

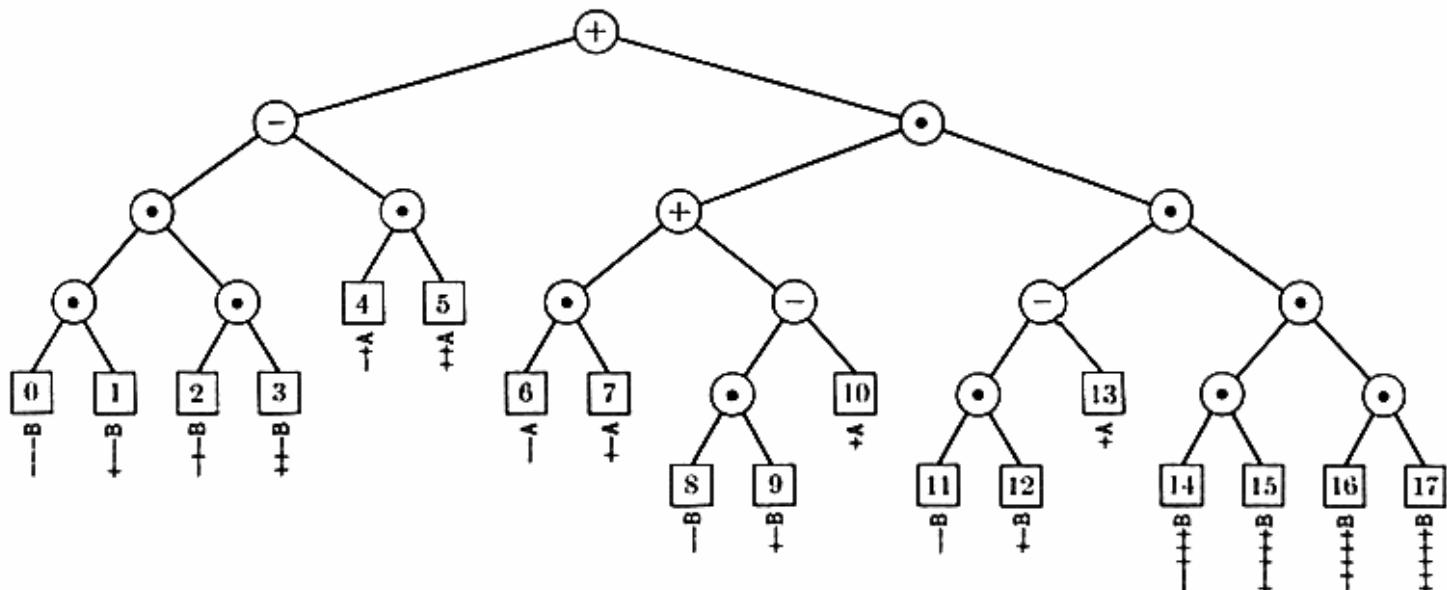


Fig. 23. Classification codes which specify the behavior of Algorithm A after insertion.

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 [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 $100 \leq N \leq 2000$ gave the approximate probabilities shown in Table 1 for paths of various types; apparently these probabilities rapidly approach limiting values as $N \rightarrow \infty$. 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 N TH 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

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

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

Solving for p yields fair agreement with Table 1:

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

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

$$10C + C1 + 2D + 2 - 3S, \quad (15)$$

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 $D \approx \frac{1}{3}C$, $C1 \approx \frac{1}{2}(C + S)$, $C + S \approx \log_2 N + 0.25$, so the average search time is approximately $11.17 \log_2 N + 4.8 - 13.7S$ 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 $(6.5 \log_2 N + 4.1)u$, 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 $SF + 26 + (0, 1, \text{ or } 2)$ units, when the search is unsuccessful. The data of Table 1 indicate that $F \approx 1.8$ on the average. The rebalancing phase (lines 46–101) takes either 16.5, 8, 27.5, or 45.5 (± 0.5) units, depending on whether we increase the total height, or simply exit without rebalancing, or do a single or double rotation. The first case almost never occurs, and the others occur with the approximate probabilities .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

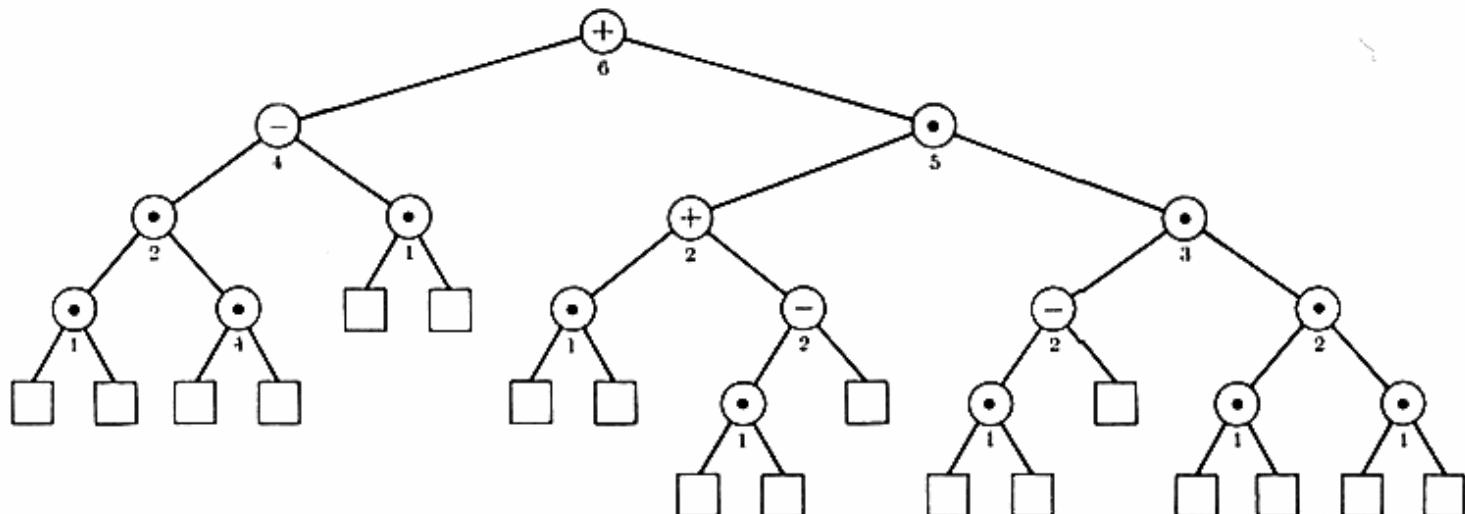


Fig. 24. RANK fields, used for searching by position.

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 k th element of the list (the k th 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 $M \leftarrow k$, $P \leftarrow \text{RLINK}(\text{HEAD})$.
- B2. [Compare.] If $P = \Lambda$, the algorithm terminates unsuccessfully. (This can happen only if k was greater than the number of nodes in the tree, or $k \leq 0$.) Otherwise if $M < \text{RANK}(P)$, go to B3; if $M > \text{RANK}(P)$, go to B4; and if $M = \text{RANK}(P)$, the algorithm terminates successfully (P points to the k th node).
- B3. [Move left.] Set $P \leftarrow \text{LLINK}(P)$ and return to B2.
- B4. [Move right.] Set $M \leftarrow M - \text{RANK}(P)$ and $P \leftarrow \text{RLINK}(P)$ 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 k th 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 $T \leftarrow \text{HEAD}$, $S \leftarrow P \leftarrow \text{RLINK}(\text{HEAD})$, $U \leftarrow M \leftarrow k$.
- C2. [Compare.] If $M \leq \text{RANK}(P)$, go to C3, otherwise go to C4.
- C3. [Move left.] Set $\text{RANK}(P) \leftarrow \text{RANK}(P) + 1$ (we will be inserting a new node to the left of P). Set $R \leftarrow \text{LLINK}(P)$. If $R = \Lambda$, set $\text{LLINK}(P) \leftarrow Q$ and go to C5. Otherwise if $B(R) \neq 0$ set $T \leftarrow P$, $S \leftarrow R$, and $U \leftarrow M$. Finally set $P \leftarrow R$ and return to C2.
- C4. [Move right.] Set $M \leftarrow M - \text{RANK}(P)$, and $R \leftarrow \text{RLINK}(P)$. If $R = \Lambda$, set $\text{RLINK}(P) \leftarrow Q$ and go to C5. Otherwise if $B(R) \neq 0$ set $T \leftarrow P$, $S \leftarrow R$, and $U \leftarrow M$. Finally set $P \leftarrow R$ and return to C2.
- C5. [Insert.] Set $\text{RANK}(Q) \leftarrow 1$, $\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$, $B(Q) \leftarrow 0$.

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 $M < \text{RANK}(S)$, set $R \leftarrow P \leftarrow \text{LLINK}(S)$, otherwise set $R \leftarrow P \leftarrow \text{RLINK}(S)$ and $M \leftarrow M - \text{RANK}(S)$. Then repeatedly do the following operation until $P = Q$: If $M < \text{RANK}(P)$, set $B(P) \leftarrow -1$ and $P \leftarrow \text{LLINK}(P)$; if $M > \text{RANK}(P)$, set $B(P) \leftarrow +1$ and $M \leftarrow M - \text{RANK}(P)$ and $P \leftarrow \text{RLINK}(P)$. (If $M = \text{RANK}(P)$, then $P = Q$ and we may go on to the next step.)

C7. [Balancing act.] If $U < \text{RANK}(S)$, set $a \leftarrow -1$, otherwise set $a \leftarrow +1$. Several cases now arise:

- i) If $B(S) = 0$, set $B(S) \leftarrow a$, $\text{LLINK}(\text{HEAD}) \leftarrow \text{LLINK}(\text{HEAD}) + 1$, 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 $P \leftarrow R$, $\text{LINK}(a, S) \leftarrow \text{LINK}(-a, R)$, $\text{LINK}(-a, R) \leftarrow S$, $B(S) \leftarrow B(R) \leftarrow 0$. If $a = +1$, set $\text{RANK}(R) \leftarrow \text{RANK}(R) + \text{RANK}(S)$; if $a = -1$, set $\text{RANK}(S) \leftarrow \text{RANK}(S) - \text{RANK}(R)$.

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

C10. [Finishing touch.] If $S = \text{RLINK}(T)$ then set $\text{RLINK}(T) \leftarrow P$, otherwise set $\text{LLINK}(T) \leftarrow P$. ■

***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 $\text{LLINK}(P)$ or $\text{RLINK}(P)$ 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

$$(P_0, a_0), \quad (P_1, a_1), \quad \dots, \quad (P_l, a_l), \quad (16)$$

where $P_0 = \text{HEAD}$, $a_0 = +1$; $\text{LINK}(a_i, P_i) = P_{i+1}$, for $0 \leq i < l$; $P_l = P$; and $\text{LINK}(a_l, P_l) = \Lambda$. This list can be placed on an auxiliary stack as we search down the tree. The process of deleting node P sets $\text{LINK}(a_{l-1}, P_{l-1}) \leftarrow \text{LINK}(-a_l, P_l)$, 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 $\text{LLINK}(\text{HEAD}) \leftarrow \text{LLINK}(\text{HEAD}) - 1$ 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!

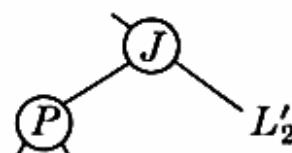
The situations requiring rebalancing are almost the same as we met in the insertion algorithm; referring again to (1), A is node P_k , and B is node $\text{LINK}(-a_k, P_k)$, the *opposite* branch from where the deletion has occurred. The only new feature is that node B might be balanced; this leads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

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 $\text{height}(L_1) \geq \text{height}(L_2)$; the other case is similar. Delete the first node of L_2 , calling it the “juncture node” J , and let L'_2 be the new tree for $L_2 \setminus \{J\}$. Now go down the right links of L_1 until reaching a node P such that

$$\text{height}(P) - \text{height}(L'_2) = 0 \text{ or } 1;$$

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



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

Crane 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 $\{A, \dots, I\}$ and the other containing $\{J, \dots, Q\}$; 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-

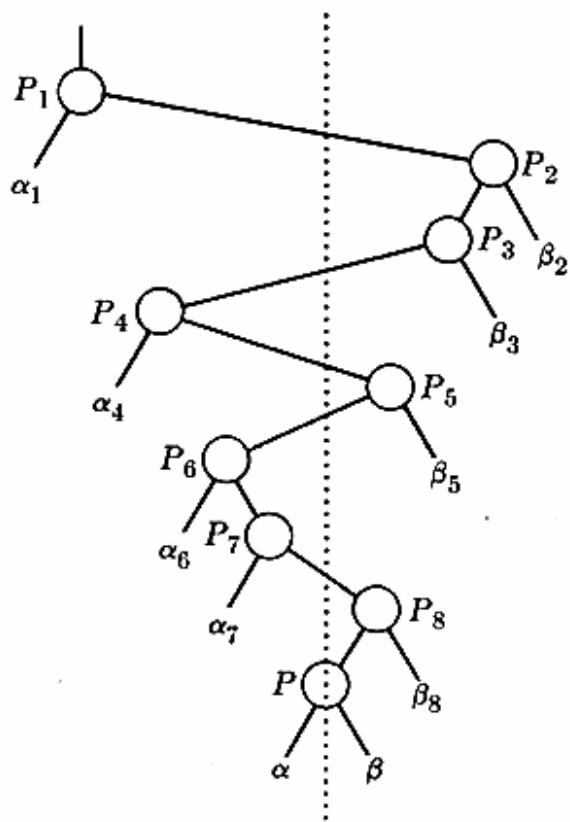


Fig. 25. The problem of splitting a list.

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, \dots, 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-

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.

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

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

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 book-keeping 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 "3-2 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.

Insertion into a 3-2 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

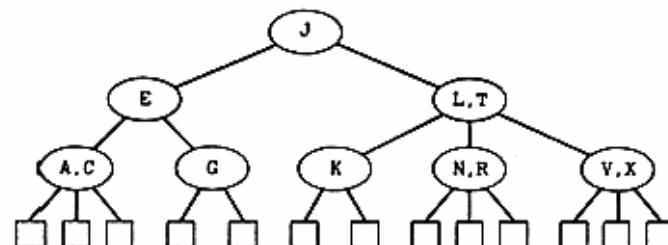


Fig. 26. A 3-2 tree.

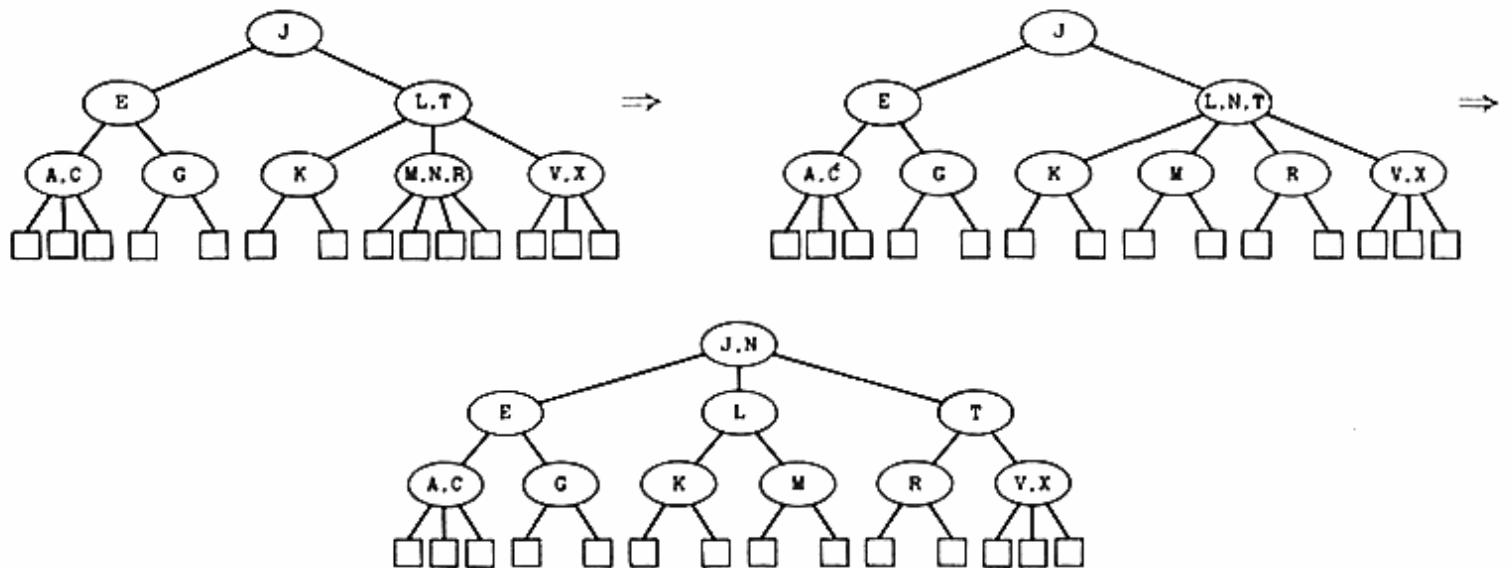


Fig. 27. Inserting the new key “M” into the 3-2 tree of Fig. 26.

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 3-2 tree of Fig. 26.

Hopcroft has observed that deletion, concatenation, and splitting can all be done with 3-2 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 3-2 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).

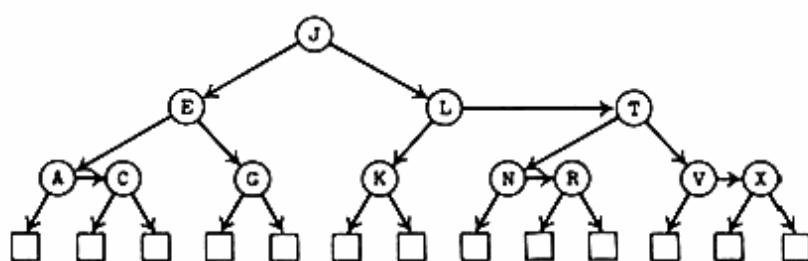


Fig. 28. The 3-2 tree of Fig. 26 represented as a binary search tree.

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 N keys into an initially empty tree, and if these keys arrive in increasing order, 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 $B'_h(1)/B_h(1) - 2^h\beta \rightarrow 1$ as $h \rightarrow \infty$.
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, $\sum_{h \geq 0} h B_{nh} / \sum_{h \geq 0} B_{nh}$?
10. [M48] Does Algorithm A make an average of $\sim \log_2 N + c$ comparisons to insert the N th 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 M th 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 $\{A, \dots, I\}$ and $\{J, \dots, Q\}$, using the splitting algorithm suggested in the text.

- 19. [26] Find a way to transform a given balanced tree so that the balance factor at the root is not -1 . Your transformation should preserve the symmetric order of the nodes; and it should produce another balanced tree in $O(1)$ units of time, regardless of the size of the original tree.
20. [40] Explore the idea of using the restricted class of balanced trees whose nodes all have balance factors of 0 or $+1$. (Then the length of the B field can be reduced to one bit.) Is there a reasonably efficient insertion procedure for such trees?
- 21. [30] 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. [M28] Estimate the maximum number of comparisons needed to search in a 3-2 tree with N internal nodes.
28. [41] Prepare efficient implementations of 3-2 tree algorithms.
29. [M47] Analyze the average behavior of 3-2 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-

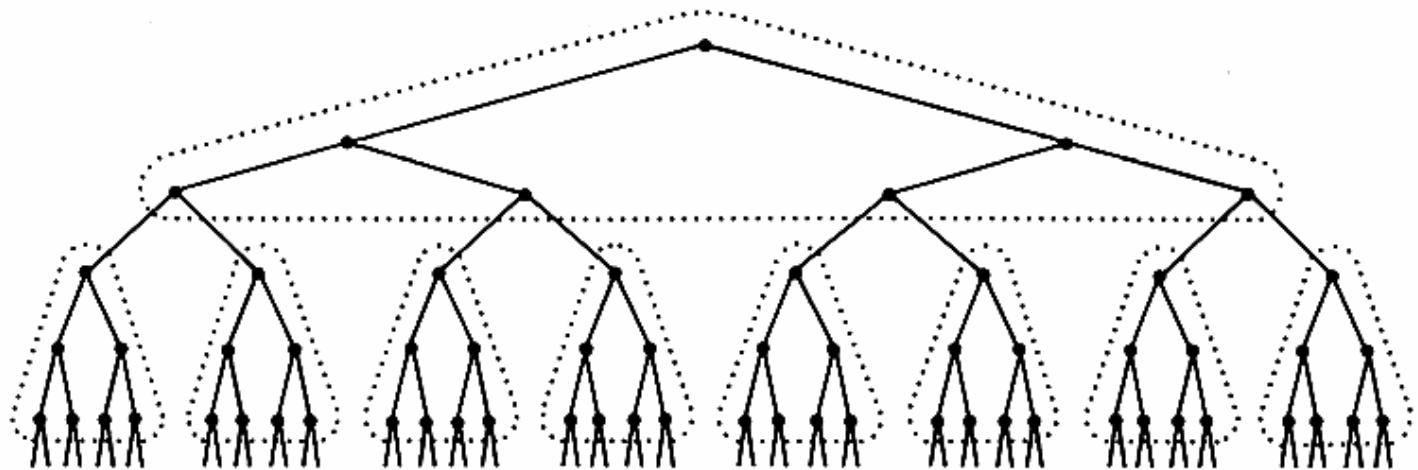


Fig. 29. A large binary search tree can be divided into “pages.”

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 naïve 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

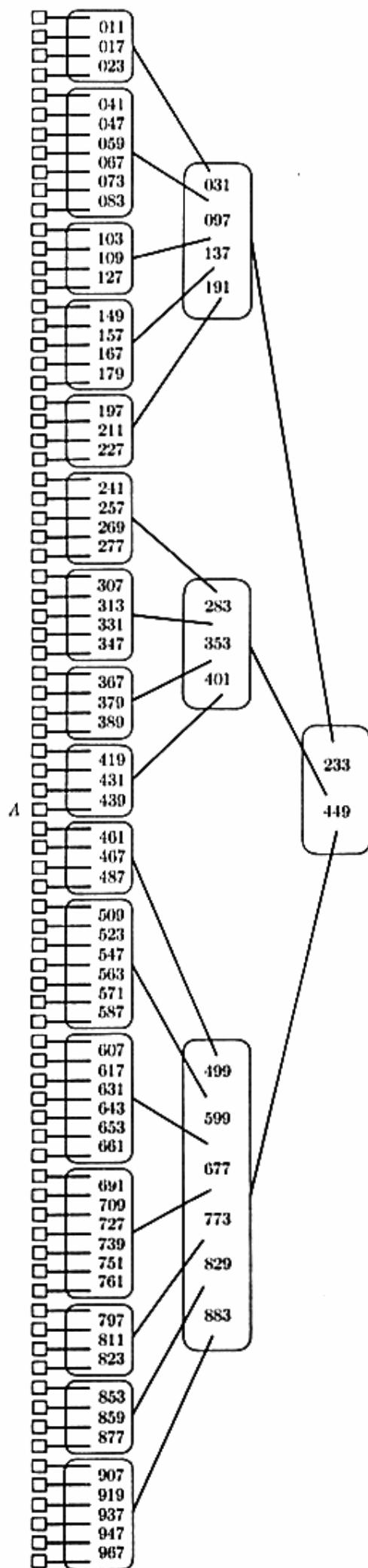
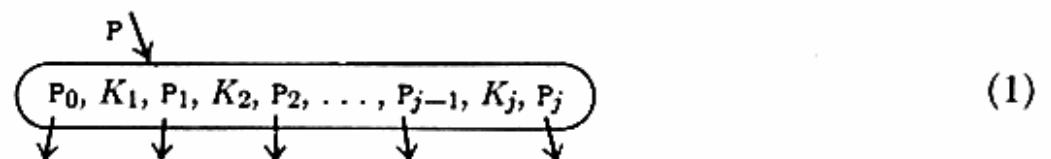


Fig. 30. A *B*-tree of order 7, with all leaves on level 3.

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.

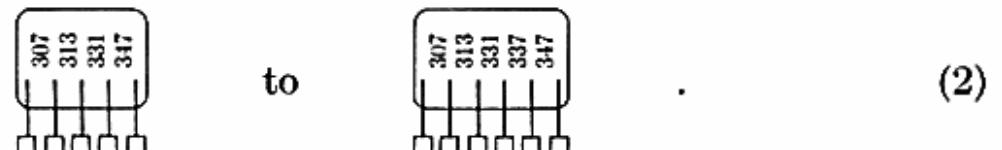
B-trees of order 1 or 2 are obviously uninteresting, so we will consider only the case $m \geq 3$. Note that “3-2 trees,” defined at the close of Section 6.2.3, are *B*-trees of order 3; and conversely, a *B*-tree of order 3 is a 3-2 tree.

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

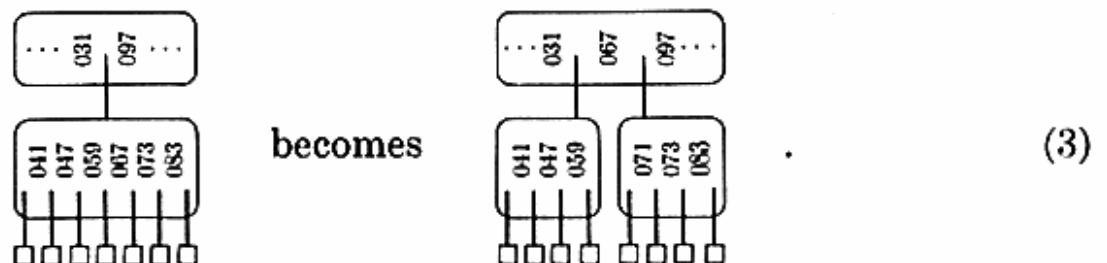


where $K_1 < K_2 < \dots < K_j$ and P_i points to the subtree for keys between K_i and K_{i+1} . Therefore searching in a *B*-tree is quite straightforward: After node (1) has been fetched into the internal memory, we search for the given argument among the keys K_1, K_2, \dots, K_j . (When j is large,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

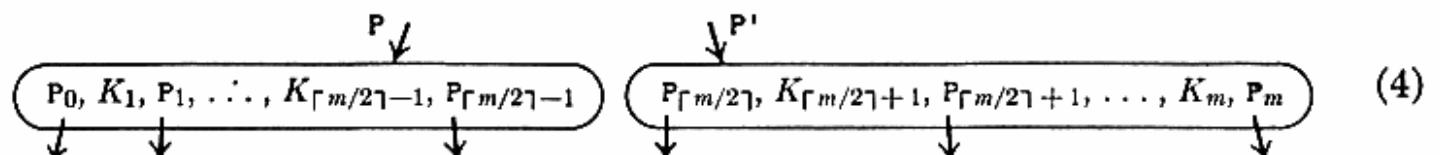
The nice thing about *B*-trees is that insertion is also quite simple. Consider Fig. 30, for example; every leaf corresponds to a place where a new insertion might happen. If we want to insert the new key 337, we simply change the appropriate node from



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:



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



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 $2, 2\lceil m/2 \rceil, 2\lceil m/2 \rceil^2, \dots$; hence

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

In other words,

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

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 one 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

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

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"

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, \dots 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.

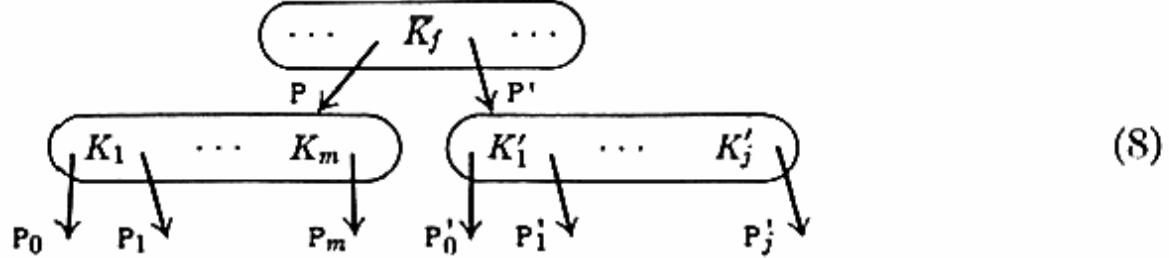
We can therefore reconsider Fig. 30, imagining that all the records of the file are now stored in the leaves, and that only a few of the keys have been duplicated in the branch nodes. Under this interpretation, the leftmost leaf contains all records whose key is ≤ 011 ; the leaf marked A contains all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

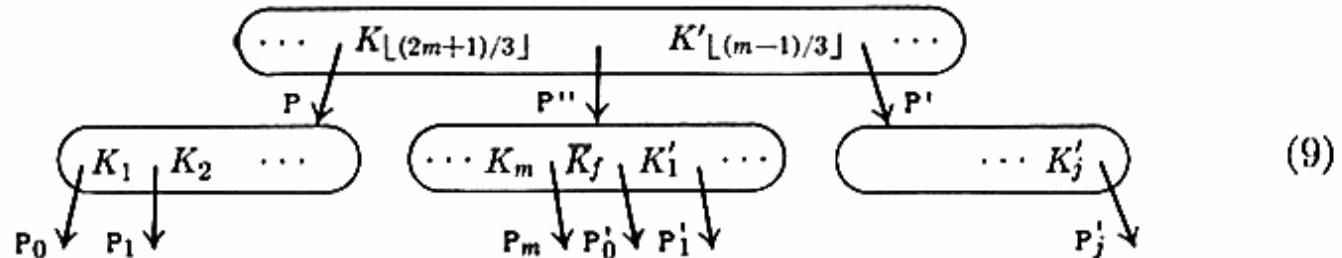
T. H. Martin [unpublished] has pointed out that the idea underlying B -trees can be used also for *variable-length* keys. We need not put bounds $[m/2, m]$ on the number of sons of each node, instead we can say merely that each node should be at least about half full of data. The insertion and splitting 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

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



If $j < m - 1$, a simple rearrangement makes splitting unnecessary: we leave $\lfloor(m+j)/2\rfloor$ keys in the left node, we replace \bar{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 \bar{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:



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 $B_1^{(j)}(z) = \delta_{j1}z$;

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

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

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

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

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

and relate C'_N to an N th 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 retrieval. 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 $0 \leq k < M$.) Let X be table entry number k in $\text{NODE}(P)$. If X is a link, go to T3; but if X is a key, go to T4.
- T3. [Advance.] If $X \neq \Lambda$, set $P \leftarrow X$ and return to step T2; otherwise the algorithm terminates unsuccessfully.
- T4. [Compare.] If $K = X$, the algorithm terminates successfully; otherwise it terminates unsuccessfully. ■

Table 1

A 'TRIE' FOR THE 31 MOST COMMON ENGLISH WORDS

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)
U	—	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

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. $rI1 \equiv P$, $rX \equiv$ unscanned part of K .

01	START	LDX	K	1	<u>T1. Initialize.</u>
02		ENT1	ROOT	1	$P \leftarrow$ pointer to root of trie.
03	2H	SLAX	1	C	<u>T2. Branch.</u>
04		STA	*+1(2:2)	C	Extract next character, k .
05		ENT2	0,1	C	$Q \leftarrow P + k$.
06		LD1N	0,2(0:2)	C	$P \leftarrow \text{LINK}(Q)$.
07		J1P	2B	C	<u>T3. Advance.</u> To T2 if P is a link $\neq \Lambda$.
08		LDA	0,2	1	<u>T4. Compare.</u> $rA \leftarrow \text{KEY}(Q)$.
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

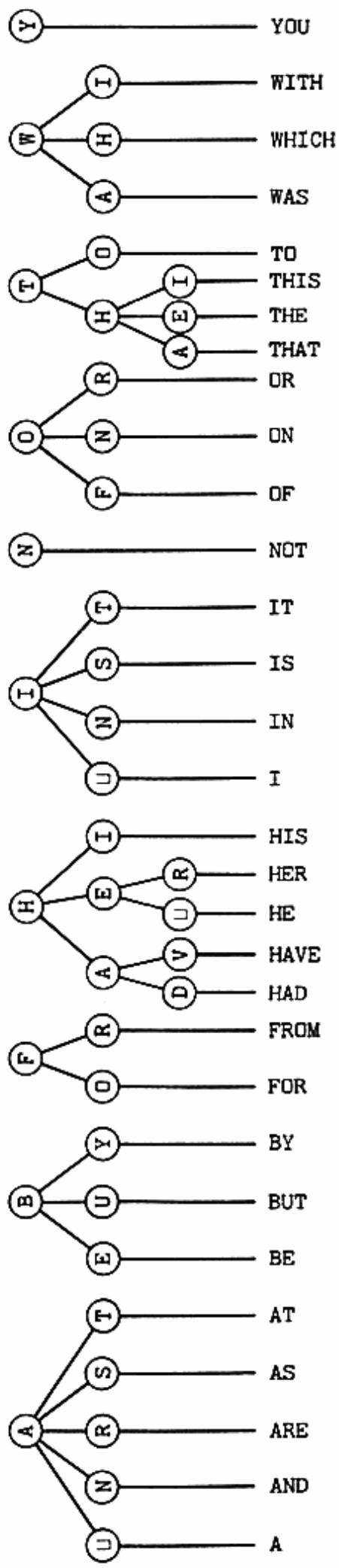
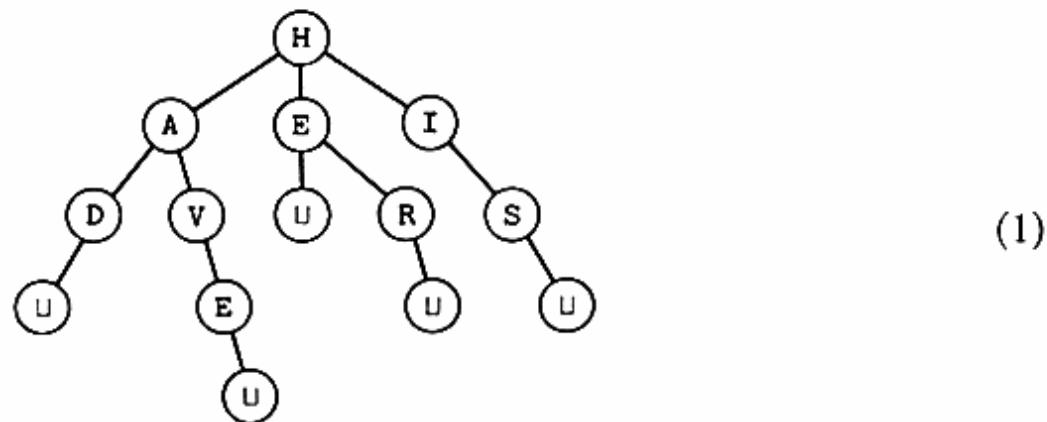


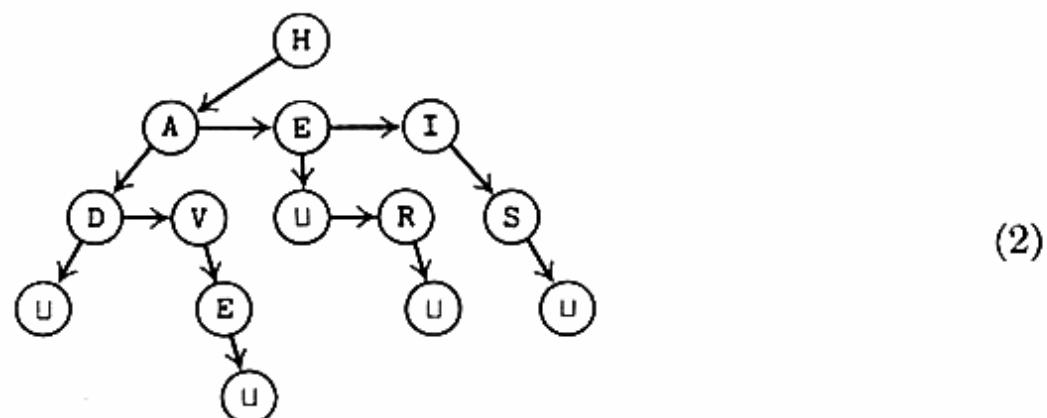
Fig. 31. The trie of Table 1, converted into a "forest."

would actually have used



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



(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 $\log_M N = \log_2 N / \log_2 M$ 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,

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. "computation"; 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

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 $\alpha, \alpha + 1, \dots, \alpha + n - 1$. 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*

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.

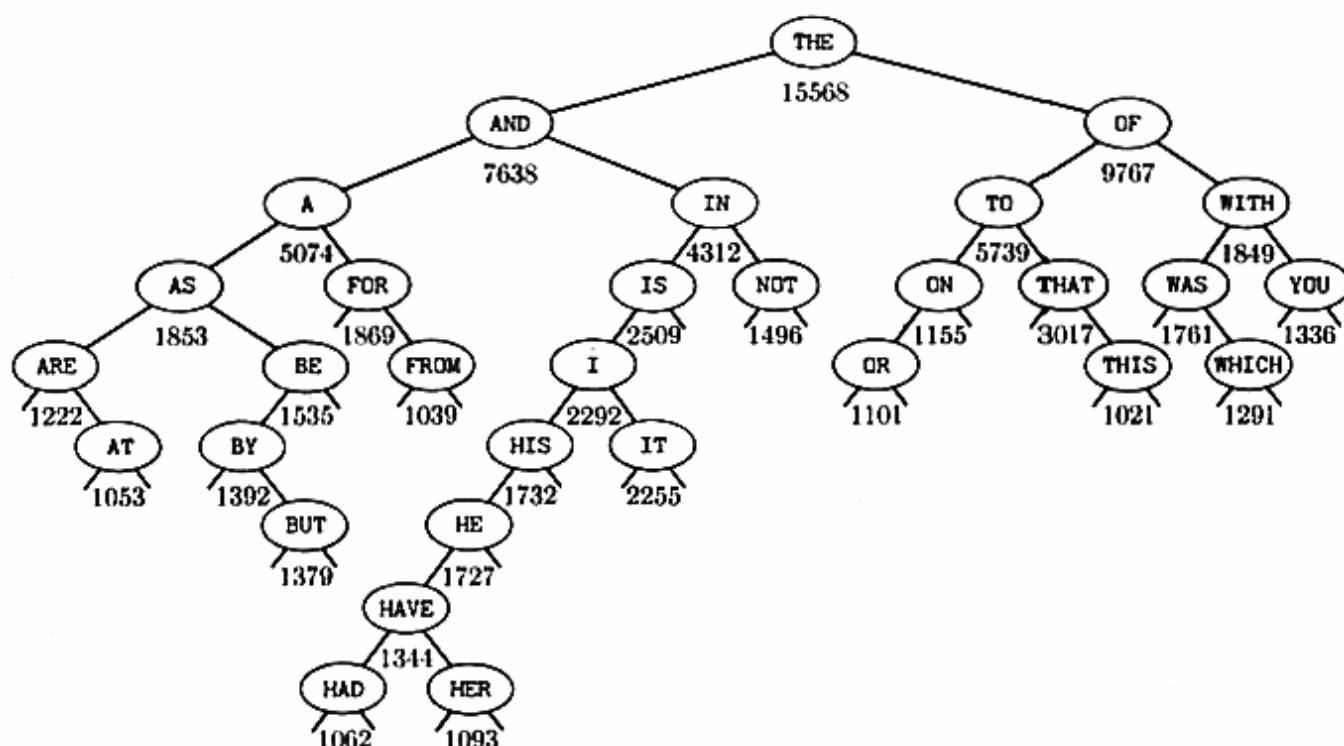


Fig. 32. A digital search tree for the 31 most common English words, inserted in decreasing order of frequency.

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 $P \leftarrow \text{ROOT}$, and $K1 \leftarrow K$.
- D2.** [Compare.] If $K = \text{KEY}(P)$, 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 $\text{LLINK}(P) \neq \Lambda$, set $P \leftarrow \text{LLINK}(P)$ and go back to D2. Otherwise go to D5.
- D4.** [Move right.] If $\text{RLINK}(P) \neq \Lambda$, set $P \leftarrow \text{RLINK}(P)$ and go back to D2.
- D5.** [Insert into tree.] Set $Q \leftarrow \text{AVAIL}$, $\text{KEY}(Q) \leftarrow K$, $\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$. If $b = 0$ set $\text{LLINK}(P) \leftarrow Q$, otherwise set $\text{RLINK}(P) \leftarrow Q$. ■

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.

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 we end the text with a unique end-of-text code (in this case “.”) that appears

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: THIS IS THE HOUSE THAT JACK BUILT.

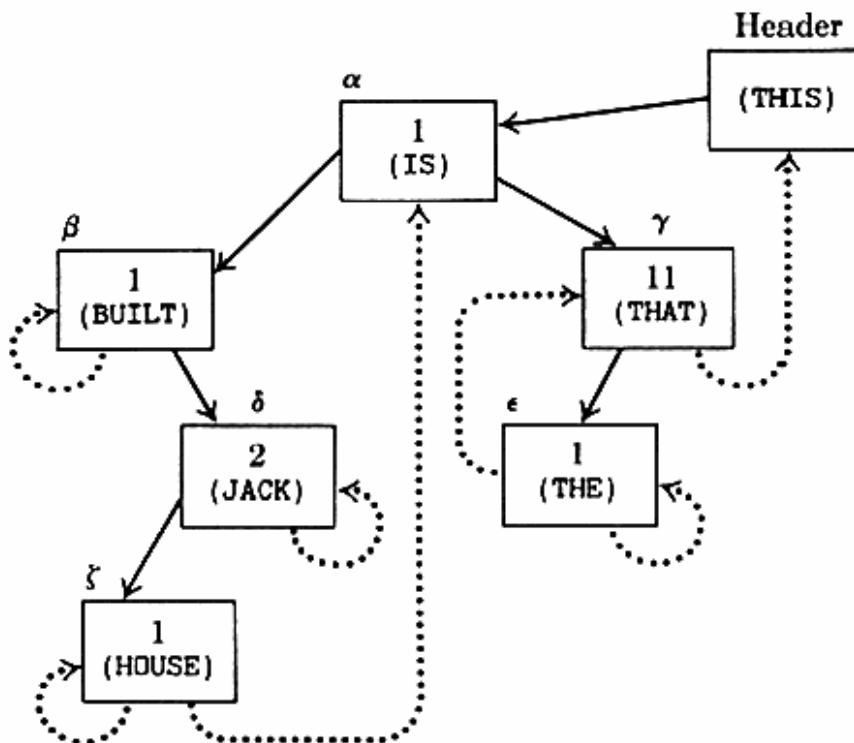


Fig. 33. An example of Patricia's tree and TEXT.

nowhere else. The same restriction was implicit in the trie scheme of Algorithm T, where “ \sqcup ” 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 xxxx 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 $P \leftarrow \text{HEAD}$ 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 $P \leftarrow \text{LLINK}(Q)$. If $\text{LTAG}(Q) = 1$, 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 $\text{KEY}(P)$.) Set $j \leftarrow j + \text{SKIP}(P)$. 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 $\text{KEY}(P)$.) If the j th bit of K is 0, go to P2, otherwise go to P5.
- P5. [Move right.] Set $Q \leftarrow P$ and $P \leftarrow \text{RLINK}(Q)$. If $\text{RTAG}(Q) = 0$, go to P3.
- P6. [Compare.] (At this point we know that if K matches any key, it matches the key starting at $\text{KEY}(P)$.) Compare K to the key starting at position $\text{KEY}(P)$ 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.

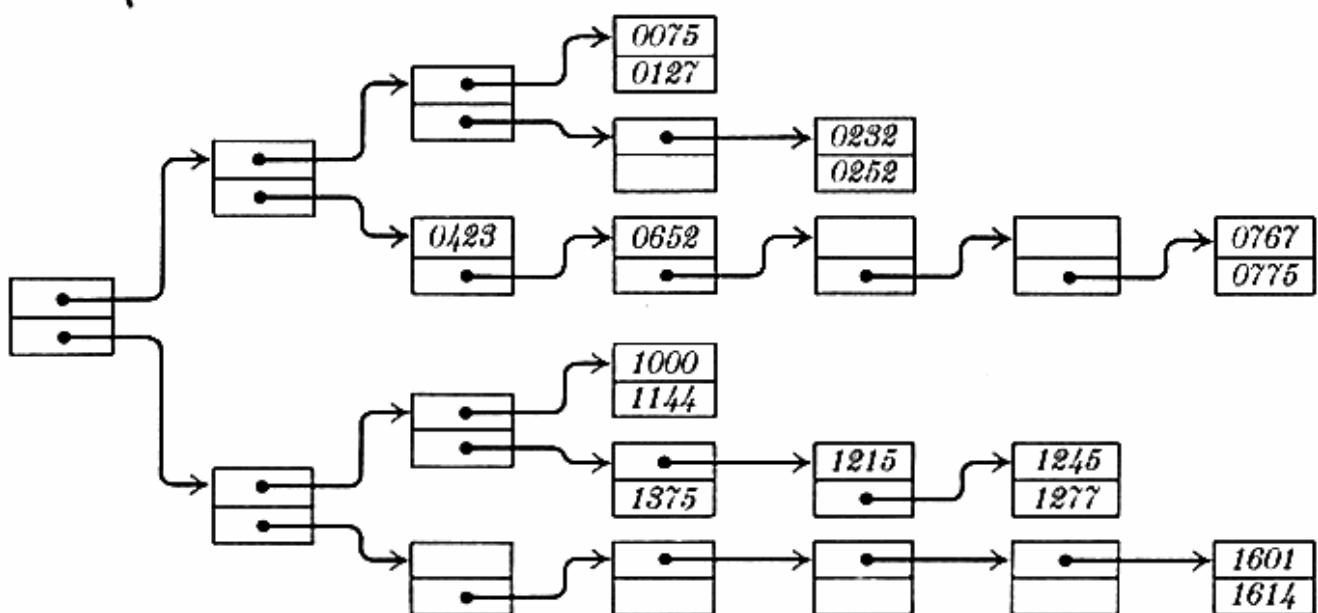


Fig. 34. Example of a random binary trie.

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 p th 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 subtrees; 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.* ■

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

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

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

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

by using symmetry and then summing over k_2, \dots, 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

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

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 \bar{C}_N denotes the average total number of bit inspections made when looking up all N keys in a binary digital search tree, it is not difficult to deduce as above that $\bar{C}_0 = \bar{C}_1 = 0$, and

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

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.)

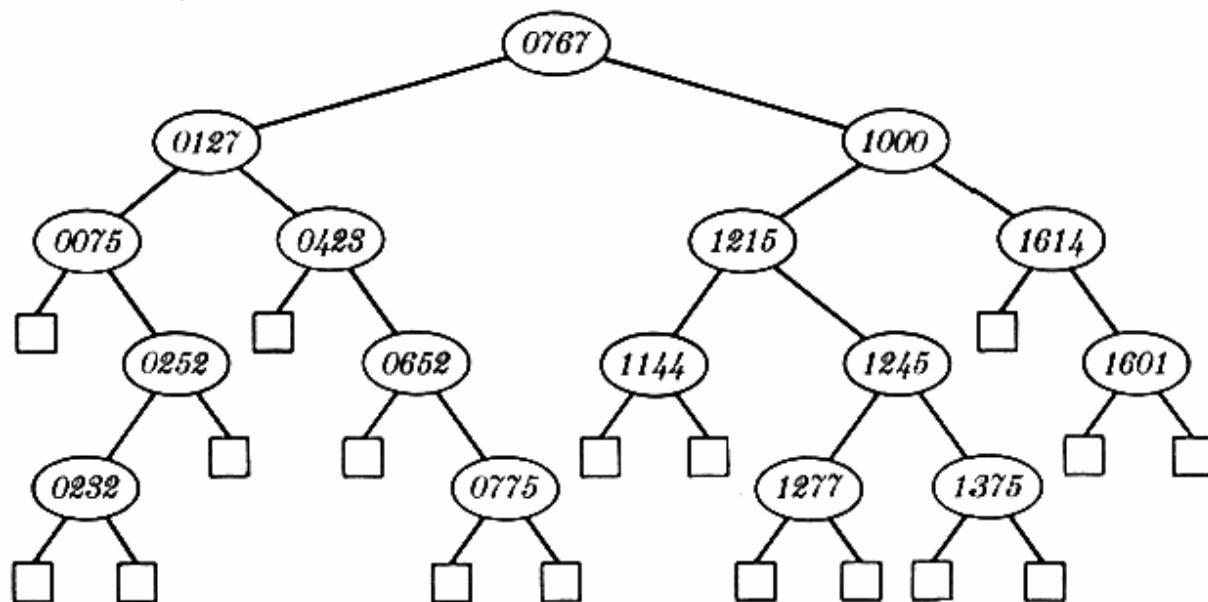


Fig. 35. A random digital search tree constructed by Algorithm D.

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

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

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 $\sum lb_i 2^{-l} = \frac{1}{2} b'(\frac{1}{2}) = a(\frac{1}{2})$, 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

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

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

$$h_{N+1}(z) = h_N(z) + (2z - 1)h_N(\frac{1}{2}z); \quad h_0(z) = 1. \quad (9)$$

Applying this rule repeatedly, we find that

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

and so on, so that eventually we have

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

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

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:

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

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

It is not at all obvious that this formula for \bar{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 \bar{C}_N can be obtained by applying the partition identities of exercise 5.1.1–16. We have

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

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* (to appear).]

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

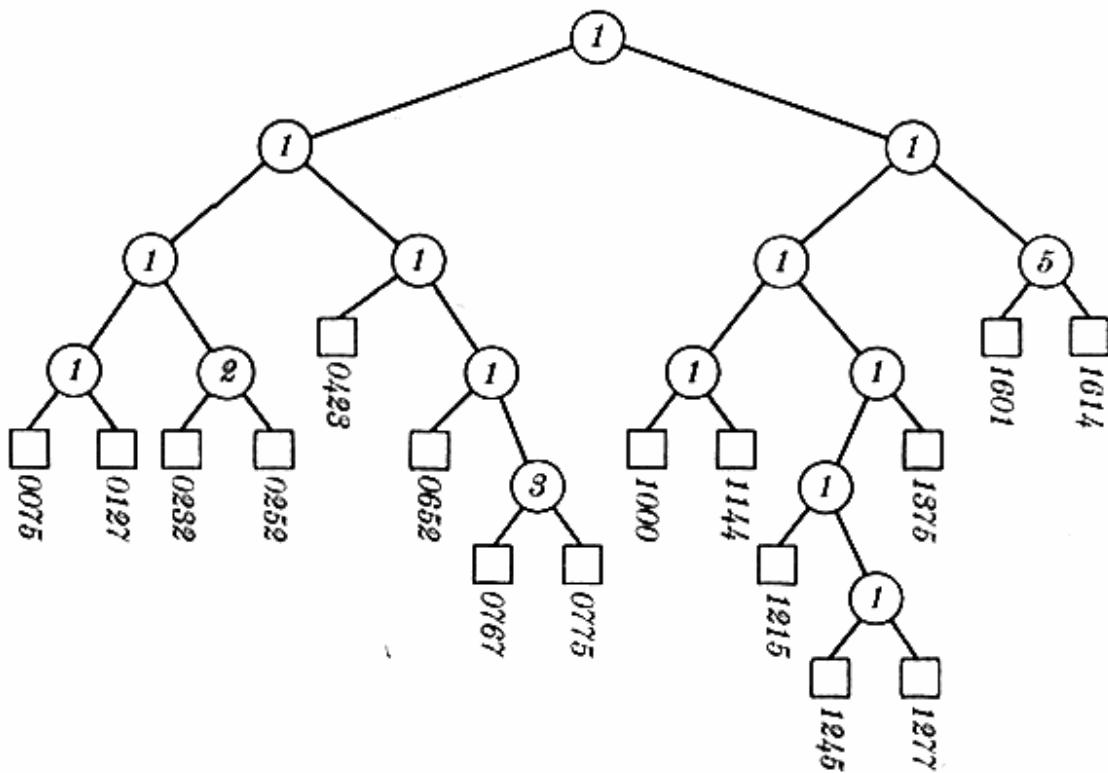


Fig. 36. Patricia constructs this tree instead of Fig. 34.

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

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

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

$$\begin{aligned} h'_n(1) &= 2^{1-n} \sum_k \binom{n}{k} h'_k(1) + 2^{1-n} \sum_k \binom{n}{k} k(1 - \delta_{kn}) \\ &= n - 2^{n-1}n + 2^{1-n} \sum_k \binom{n}{k} h'_k(1). \end{aligned} \quad (16)$$

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

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 unsuccessful search by Patricia, we obtain the recurrence

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

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:

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

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	$\log_2 N + 1.33275$	$\log_2 N - 0.10995$
Digital tree search	$\log_2 N - 1.71665$	$\log_2 N - 0.27395$
Patricia	$\log_2 N + 0.33275$	$\log_2 N - 0.31875$

(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	28
Entry	A	—	THAT	(26)	—	WAS	THE	(15)	(2)	I	THIS	HIS	WHICH	WITH	HE	AND	TO	OF	BE	ARE	(1)	AS	IN	AT	DN	(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 $K = b_1b_2\dots b_n$, suppose we want to find the maximum value of k such that there is a key in the file beginning with the bit pattern $b_1b_2\dots b_k$. How can we

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

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

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 $n \rightarrow \infty$, 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 $N \rightarrow \infty$, 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 "L" 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 $l = 0, 1, 2, \dots$?
23. [M26] How many digit inspections are made on the average during an unsuccessful 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 " $P \leftarrow \text{RLINK}(P)$ " 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 $A_N \leq M(N - 1)/(M - 1)$ for $N \geq 1$. (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

$$(1 - \frac{1}{2})(1 - \frac{1}{4})(1 - \frac{1}{8})(1 - \frac{1}{16}) \dots$$

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

27. [HM31] What is the asymptotic value of \bar{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 $M \geq 2$. 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

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

[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 $A(z) = \sum_{n \geq 0} a_n z^n / n!$.]

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

(a) Prove that

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

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

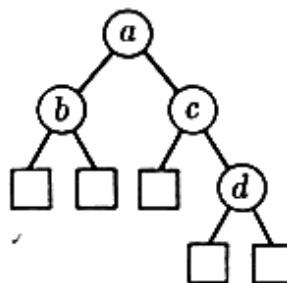
$$\frac{1}{e^x - 1} - \frac{1}{x} + \frac{1}{2} = \frac{1}{2\pi i} \int_{-\frac{1}{2}-i\infty}^{-\frac{1}{2}+i\infty} \xi(z)\Gamma(z)x^{-z} dz, \quad \text{for real } x > 0.$$

(d) Therefore the sum equals

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

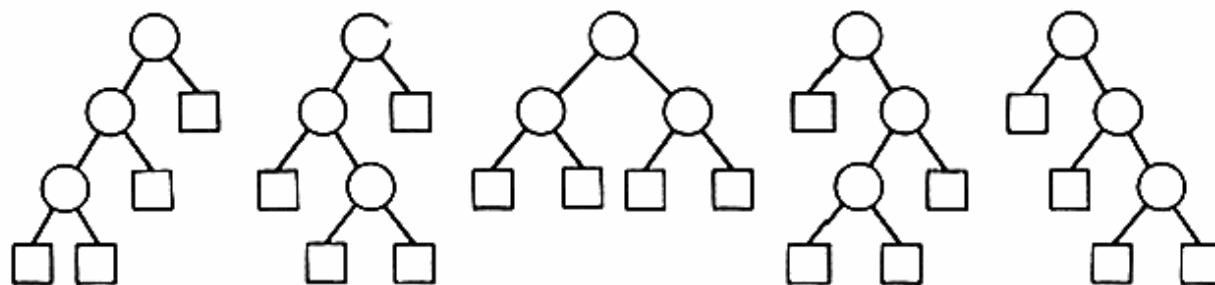
evaluate this integral.

► 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:

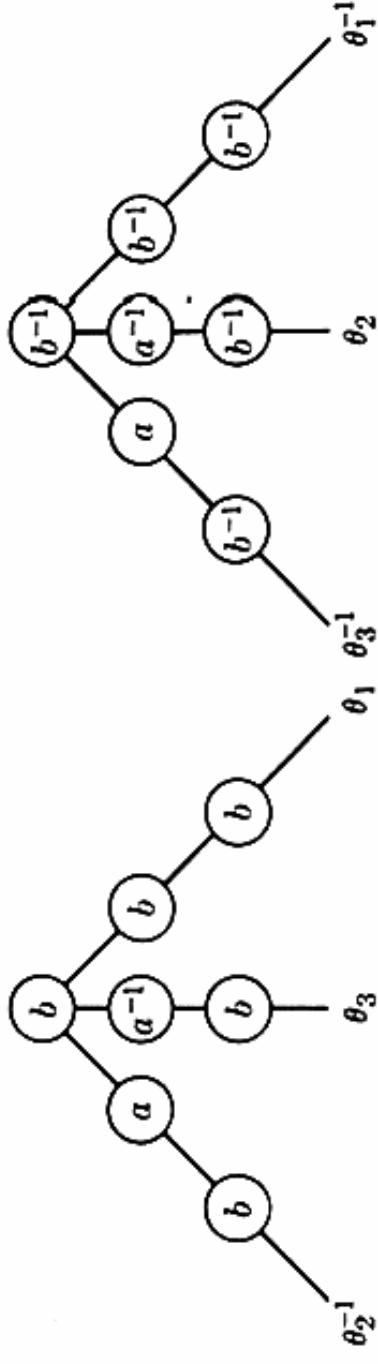


Tree search (Algorithm 6.2.2T)	$\frac{1}{6}$	$\frac{1}{6}$	$\frac{1}{3}$	$\frac{1}{6}$	$\frac{1}{6}$
Digital tree search (Algorithm D)	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{2}$	$\frac{1}{8}$	$\frac{1}{8}$
Patricia (Algorithm P)	$\frac{1}{7}$	$\frac{1}{7}$	$\frac{3}{7}$	$\frac{1}{7}$	$\frac{1}{7}$

(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 $\alpha \cdot (\text{internal path length}) + \beta \cdot (\text{modified external path length})$, 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 $a_0a_1a_2\dots$ 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 $b_0b_1\dots b_{n-1}$, 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 $\{a_1, \dots, a_n\}$, i.e., the set of all strings $\alpha = b_1 \dots b_r$, where each b_i is one of the a_j or a_j^{-1} and no adjacent pair $a_j a_j^{-1}$ or $a_j^{-1} a_j$ occurs. The inverse of α is $b_r^{-1} \dots b_1^{-1}$, 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 $\{\beta_1, \dots, \beta_p\}$, 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 $\theta_1, \dots, \theta_m$ 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 $\theta_j^e \theta_i \cdot e = \pm 1$, 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 $\theta_1, \dots, \theta_m, \theta_1^{-1}, \dots, \theta_m^{-1}$ in a character-oriented search tree, using the $2n$ letters $a_1, \dots, a_n, a_1^{-1}, \dots, a_n^{-1}$. Let $\alpha = b_1 \dots b_r$ be a given element of G ; if $r = 0$, α is obviously in H . Otherwise look up α , finding the longest prefix $b_1 \dots b_k$ that matches a key. If there is more than one key beginning with $b_1 \dots b_k$, α is not in H ; otherwise let the unique such key be $b_1 \dots b_k c_1 \dots c_l = \theta_i^e$, and replace α by $\theta_i^{-e} \alpha = c_l^{-1} \dots c_1^{-1} b_{k+1} \dots b_r$. 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 $41 \cdot 40 \cdot \dots \cdot 11 = 41!/10! \approx 10^{43}$ 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

	A	AND	ARE	AS	AT	BE	BUT	BY	FOR	FROM	HAD	HAVE	HE	HER
Instruction														
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 originated in work of R. von Mises, *İstanbul Üniversitesi Fen Fakültesi Mecmuası* 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. Grieniewski 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

HIS	H	N	S	T	NOT	OF	ON	OR	THAT	THE	THIS	TO	WAS	WHICH	WITH	YOU	
Contents of r11 after executing the instruction, given a particular key K																	
-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	-23	-15	-33	-26	-25	-20
-7	-17	-2	5	6	-7	-18	-9	-5	-23	-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

$$0 \leq h(K) < M, \quad (1)$$

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 :

$$h(K) = K \bmod M. \quad (2)$$

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 $r^k \not\equiv \pm a$ (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	$rX \leftarrow K$.
ENTA	0	$rA \leftarrow 0$.
DIV	=1009=	$rX \leftarrow K \bmod 1009$.

(3)

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

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

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	$rA \leftarrow K$.
MUL	A	$rAX \leftarrow AK$.
ENTA	0	$rAX \leftarrow AK \bmod w$.
SLB	m	Shift rAX m bits to the left.

(5)

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 would 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

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

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 $\{K, K + d, K + 2d, \dots, K + td\}$ 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 $h(K), h(K + d), h(K + 2d), \dots$ 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 $h(K), h(K + 1), h(K + 2), \dots$ can be studied by considering the behavior of the successive values $h(0), h(1), h(2), \dots$. This suggests the following experiment: Starting with the line segment $[0, 1]$, we successively mark off the points $\{\phi^{-1}\}, \{2\phi^{-1}\}, \{3\phi^{-1}\}, \dots$, 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

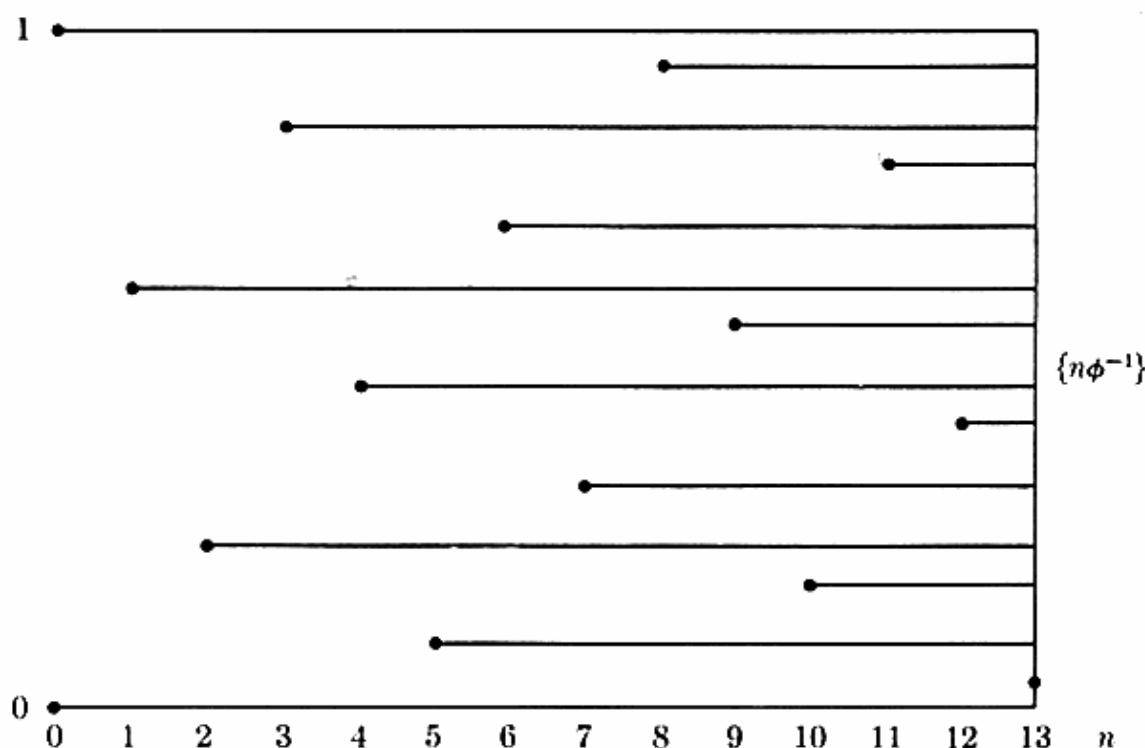


Fig. 37. Fibonacci hashing.

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 $\{\theta\}$, $\{2\theta\}$, \dots , $\{n\theta\}$ 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 $\{\theta\}$, $\{2\theta\}$, \dots , $\{n\theta\}$ 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

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

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

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

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-

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

$$\frac{1}{4} < \theta < \frac{3}{10}, \quad \frac{1}{3} < \theta < \frac{3}{7}, \quad \frac{4}{7} < \theta < \frac{2}{3}, \quad \frac{7}{10} < \theta < \frac{3}{4}.$$

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.,

$$A = \boxed{+ \ 61 \ 25 \ 42 \ 33 \ 71} . \quad (9)$$

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-or-ing.

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 m th degree polynomial $P(x) = x^m + p_{m-1}x^{m-1} + \cdots + p_0$. An n -digit binary key $K = (k_{n-1} \dots k_1 k_0)_2$ 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 $h(K) = (h_{m-1} \dots h_1 h_0)_2$. 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

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

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

$$K = \text{EN, TO, TRE, FIRE, FEM, SEKS, SYV} \quad (11)$$

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

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

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

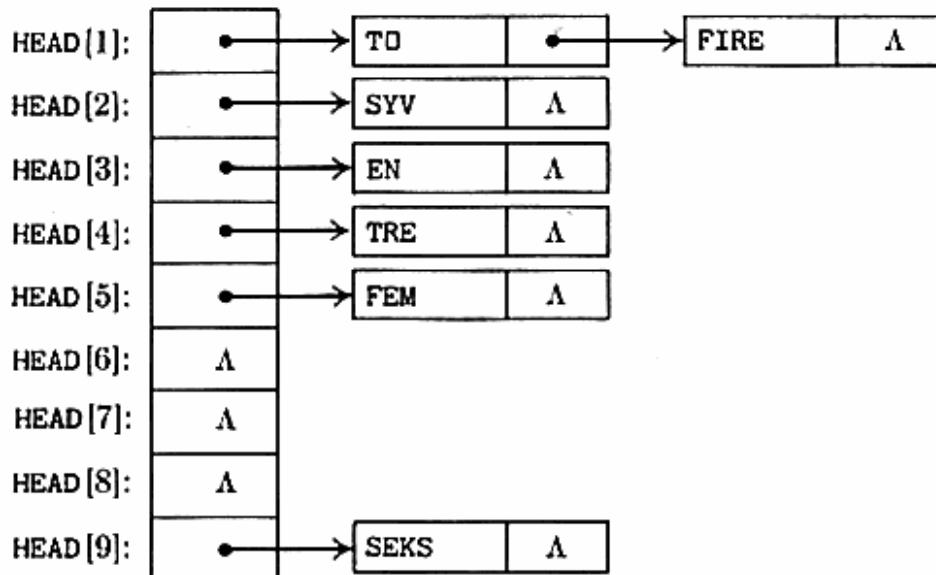


Fig. 38. Separate chaining.

to make the lists ascending, the **TO** and **FIRE** nodes of Fig. 38 would be interchanged, and all the **A** 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 $\text{TABLE}[i]$, for $0 \leq i \leq M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key field $\text{KEY}[i]$, a link field $\text{LINK}[i]$, 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 $\text{TABLE}[j]$ is occupied for all j in the range $R \leq j \leq M$. By convention, $\text{TABLE}[0]$ will always be empty.

- C1. [Hash.] Set $i \leftarrow h(K) + 1$. (Now $1 \leq i \leq M$.)
- C2. [Is there a list?] If $\text{TABLE}[i]$ is empty, go to C6. (Otherwise $\text{TABLE}[i]$ is occupied; we will look at the list of occupied nodes which starts here.)
- C3. [Compare.] If $K = \text{KEY}[i]$, the algorithm terminates successfully.
- C4. [Advance to next.] If $\text{LINK}[i] \neq 0$, set $i \leftarrow \text{LINK}[i]$ and go back to step C3.

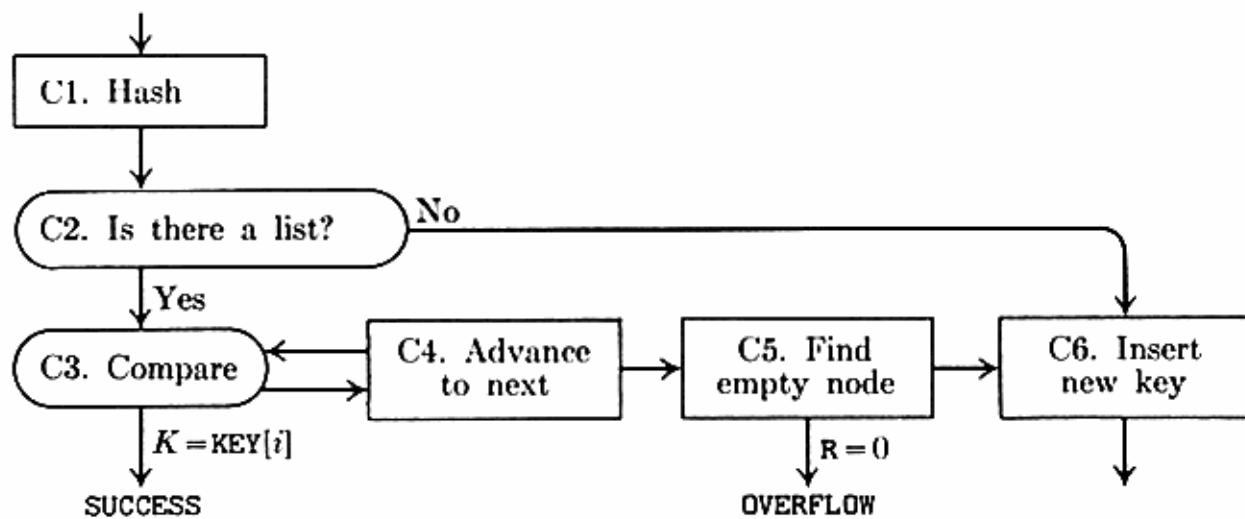


Fig. 39. Chained scatter table search and insertion.

- 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 $\text{TABLE}[R]$ is empty. If $R = 0$, the algorithm terminates with overflow (there are no empty nodes left); otherwise set $\text{LINK}[i] \leftarrow R$, $i \leftarrow R$.
- C6. [Insert new key.] Mark $\text{TABLE}[i]$ as an occupied node, with $\text{KEY}[i] \leftarrow K$ and $\text{LINK}[i] \leftarrow 0$. ■

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.

TABLE [1]:	TO	•
TABLE [2]:	SYV	Λ
TABLE [3]:	EN	Λ
TABLE [4]:	TRE	Λ
TABLE [5]:	FEM	Λ
TABLE [6]:		
TABLE [7]:		
TABLE [8]:	SEKS	Λ
TABLE [9]:	FIRE	•

Fig. 40. Coalesced chaining.

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 $\equiv i$, rA $\equiv K$.

01	KEY	EQU	3:5			
02	LINK	EQU	0:2			
03	START	LDX	K	1	<u>C1. Hash.</u>	
04		ENTA	0	1		
05		DIV	=M=	1		
06		STX	*+1(0:2)	1		
07		ENT1	*	1	$i \leftarrow h(k)$	
08		INC1	1	1	+ 1.	
09		LDA	K	1		
10		LD2	TABLE,1(LINK)	1	<u>C2. Is there a list?</u>	
11		J2N	6F	1	To C6 if TABLE[i] empty.	
12		CMPA	TABLE,1(KEY)	A	<u>C3. Compare.</u>	
13		JE	SUCCESS	A	Exit if $K = KEY[i]$.	
14		J2Z	5F	A — S1	To C5 if LINK[i] = 0.	
15	4H	ENT1	0,2	C — 1	<u>C4. Advance to next.</u>	
16		CMPA	TABLE,1(KEY)	C — 1	<u>C3. Compare.</u>	
17		JE	SUCCESS	C — 1	Exit if $K = KEY[i]$.	
18		LD2	TABLE,1(LINK)	C — 1 — S2		
19		J2NZ	4B	C — 1 — S2	Advance if $LINK[i] \neq 0$.	
20	5H	LD2	R	A — S	<u>C5. Find empty node.</u>	
21		DEC2	1	T	$R \leftarrow R - 1$.	
22		LDX	TABLE,2	T		
23		JXNN	*-2	T	Repeat until TABLE[R] empty.	
24		J2Z	OVERFLOW	A — S	Exit if no empty nodes left.	
25		ST2	TABLE,1(LINK)	A — S	$LINK[i] \leftarrow R$.	
26		ENT1	0,2	A — S	$i \leftarrow R$.	
27		ST2	R	A — S	Update R in memory.	
28	6H	STZ	TABLE,1(LINK)	1 — S	<u>C6. Insert new key.</u>	
29		STA	TABLE,1(KEY)	1 — S	$KEY[i] \leftarrow K$. ■	

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 $S = S_1 + S_2$, where $S_1 = 1$ if successful on the first try. The total running time for the searching phase of Program C is $(7C + 4A + 17 - 3S + 2S_1)u$, 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

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

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

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

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

$$\begin{aligned} C_N = \frac{1}{N} \sum_{0 \leq k < N} \left(C'_k + \frac{k}{M} \right) &= 1 + \frac{1}{8} \frac{M}{N} \left(\left(1 + \frac{2}{M} \right)^N - 1 - \frac{2N}{M} \right) + \frac{1}{4} \frac{N-1}{M} \\ &\approx 1 + \frac{1}{8\alpha} (e^{2\alpha} - 1 - 2\alpha) + \frac{1}{4}\alpha \end{aligned} \quad (16)$$

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 S_1 turns out to be

$$S_{1N} = 1 - \frac{1}{2}((N-1)/M) \approx 1 - \frac{1}{2}\alpha. \quad (17)$$

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,

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

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

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

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

$$h(K), h(K) - 1, \dots, 0, M - 1, M - 2, \dots, h(K) + 1 \quad (20)$$

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 $\text{TABLE}[i]$, for $0 \leq i < M$, and they are of two distinguishable types, *empty* and *occupied*. An occupied node contains a key, called $\text{KEY}[i]$, 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 $\text{TABLE}[i]$ is empty, go to L4. Otherwise if $\text{KEY}[i] = K$, the algorithm terminates successfully.
- 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 $\text{TABLE}[i]$ occupied, and set $\text{KEY}[i] \leftarrow K$. ■

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 $h(K)$.

0	FEM
1	TRE
2	EN
3	
4	
5	SYV
6	SEKS
7	TO
8	FIRE

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 non-negative, letting empty positions contain -1 .) The table size M is assumed to be prime, and $\text{TABLE}[i]$ is stored in location $\text{TABLE} + i$ for $0 \leq i < M$. For speed in the inner loop, location $\text{TABLE} - 1$ is assumed to contain 0. Location VACANCIES is assumed to contain the value $M - 1 - N$; and $\text{rA} \equiv K$, $\text{rI1} \equiv i$.

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 $9u$.

01	START	LDX	K	1	<u>L1. Hash.</u>
02		ENTA	0	1	
03		DIV	=M=	1	
04		STX	*+1(0:2)	1	
05		ENT1	*	1	$i \leftarrow h(K)$.
06		LDA	K	1	
07		JMP	2F	1	
08	8H	INC1	M+1	E	<u>L2. Advance to next.</u>
09	3H	DEC1	1	C + E - 1	$i \leftarrow i - 1$.
10	2H	CMPA	TABLE,1	C + E	<u>L2. Compare.</u>
11		JE	SUCCESS	C + E	Exit if $K = \text{KEY}[i]$.
12		LDX	TABLE,1	C + E - S	
13		JXNZ	3B	C + E - S	To L3 if TABLE[i] empty.
14		J1N	8B	E + 1 - S	To L3 with $i \leftarrow M$ if $i = -1$.
15	4H	LDX	VACANCIES	1 - S	<u>L4. Insert.</u>
16		JXZ	OVERFLOW	1 - S	Exit with overflow if $N = M - 1$.
17		DECX	1	1 - S	
18		STX	VACANCIES	1 - S	Increase N by 1.
19		STA	TABLE,1	1 - S	TABLE[i] $\leftarrow K$. ■

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 $M = 19$, $N = 9$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
		■	■	■	■	■						■	■	■	■	■	■	

(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 $11 \leq h(K) \leq 15$, 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

$$C'_N \approx \frac{1}{2} \left(1 + \left(\frac{1}{1 - \alpha} \right)^2 \right) \quad (\text{unsuccessful search}); \quad (22)$$

$$C_N \approx \frac{1}{2} \left(1 + \frac{1}{1 - \alpha} \right) \quad (\text{successful search}), \quad (23)$$

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 $\{K, K + 1, K + 2, \dots\}$ 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 $\{K, K + 1, K + 2, \dots\}$ 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 Balbinc [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 \leftarrow rA \vee 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

$$h_2(K) = \begin{cases} 1, & \text{if } h_1(K) = 0; \\ M - h_1(K), & \text{if } h_1(K) > 0. \end{cases} \quad (25)$$

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. $rI2 \equiv c - 1$.

01	START	LDX	K	1	08	JE	SUCCESS	1
02		ENTA	O	1	09	JXZ	4F	1 — S1
03		DIV	=M=	1	10	SRAX	5	A — S1
04		STX	*+1(0:2)	1	11	DIV	=M-2=	A — S1
05		ENT1	*	1	12	STX	*+1(0:2)	A — S1
06		LDX	TABLE,1	1	13	ENT2	*	A — S1
07		CMPX	K	1	14	LDA	K	A — S1
		15	3H	DEC1	1,2		C — 1	
		16		J1NN	*+2		C — 1	
		17		INCL	M		B	
		18		CMPA	TABLE,1		C — 1	
		19		JE	SUCCESS		C — 1	
		20		LDX	TABLE,1	C — 1	— S2	
		21		JXNZ	3B	C — 1	— S2	
		22	4H	LDX	VACANCIES	1	— S	
		23		JXZ	OVERFLOW	1	— S	
		24		DECX	1	1	— S	
		25		STX	VACANCIES	1	— S	
		26		LDA	K	1	— S	
		27		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, $1 \leq h_2(K) \leq \frac{1}{2}M$, 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

$$C'_N = \frac{M + 1}{M + 1 - N} \approx (1 - \alpha)^{-1} \quad (\text{unsuccessful search}), \quad (26)$$

$$C_N = \frac{M + 1}{N} (H_{M+1} - H_{M+1-N}) \approx -\alpha^{-1} \ln(1 - \alpha) \quad (\text{successful search}), \quad (27)$$

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

$$C'_N = \frac{M + 1}{M + 1 - N} - \frac{N}{M + 1} + H_{M+1} - H_{M+1-N} + O(M^{-1}) \approx (1 - \alpha)^{-1} - \alpha - \ln(1 - \alpha); \quad (28)$$

$$C_N = 1 + H_{M+1} - H_{M+1-N} - \frac{N}{2(M + 1)} - (H_{M+1} - H_{M+1-N})/N + O(N^{-1}) \approx 1 - \ln(1 - \alpha) - \frac{1}{2}\alpha. \quad (29)$$

(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	$c \leftarrow M - i.$
J1NZ *+2	(30)
ENT2 0	If $i = 0$, $c \leftarrow 1$.

In this case, secondary clustering is preferable to independent double hashing.

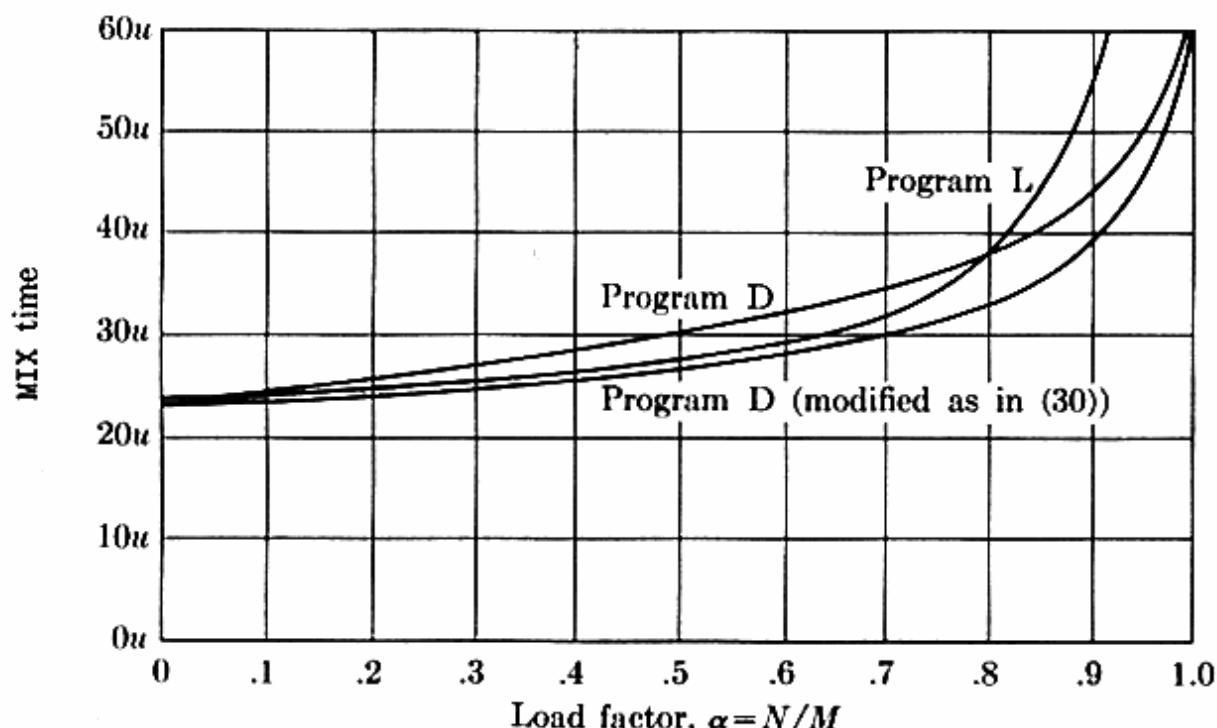


Fig. 42. The running time for successful searching by three open addressing schemes.

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=	$rA \leftarrow rA \bmod 512.$
STA *+1(0:2)	(31)
ENT2 *	$c \leftarrow rA + 1.$

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.

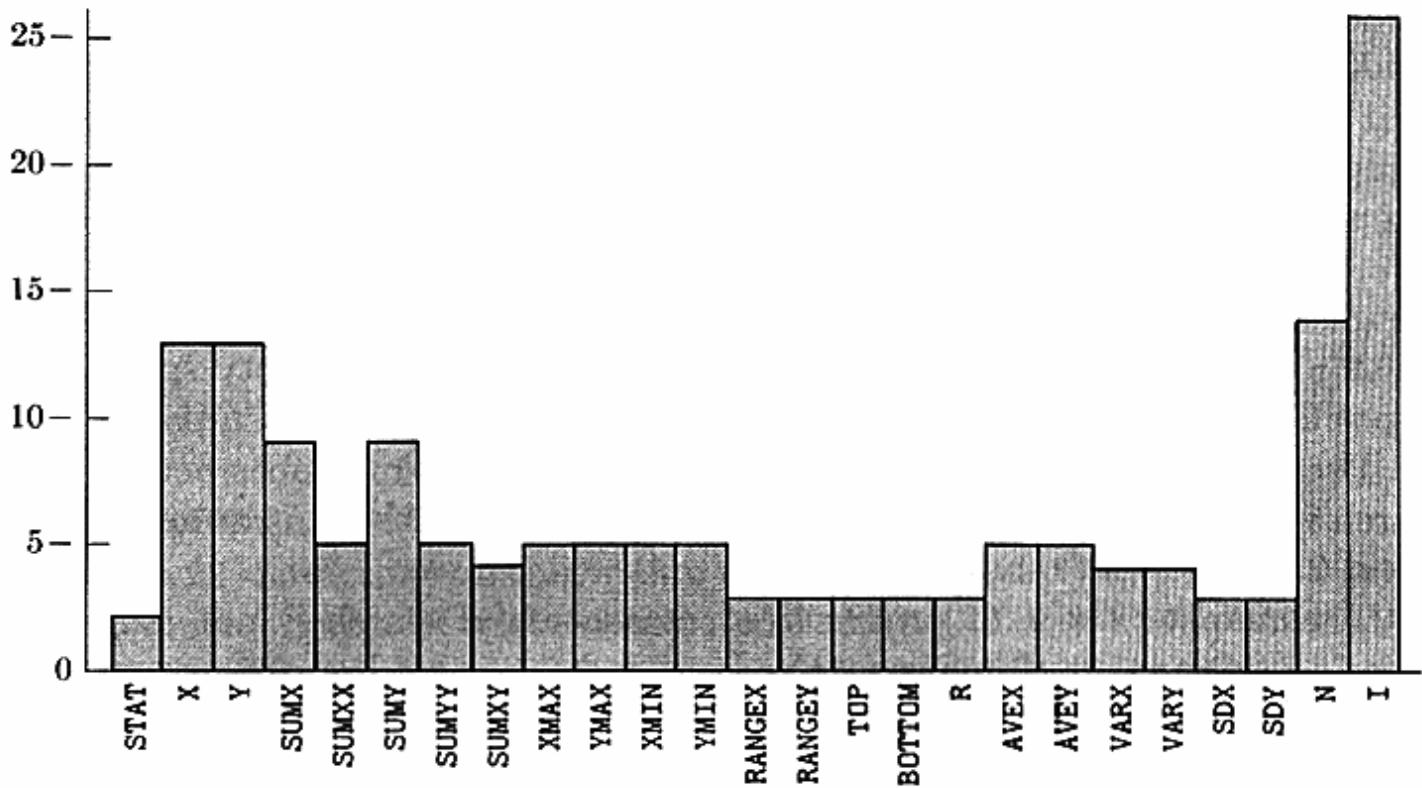


Fig. 43. The number of times a compiler typically searches for variable names. The names are listed from left to right in order of their first appearance.

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 15 (1972), to appear.]

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

$$p_0, p_1, \dots, p_{t-1}, p_t,$$

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

$\text{TABLE}[p_0]$ 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 $\text{TABLE}[(p_0 - c_0) \bmod M]$ is occupied and $t \geq 3$, we try $\text{TABLE}[(p_0 - 2c_0) \bmod M]$; if that is full too, we compute $c_1 = h_2(\text{KEY}[p_1])$ and try $\text{TABLE}[(p_1 - c_1) \bmod M]$; etc. In general, let $c_j = h_2(\text{KEY}[p_j])$ and $p_{j,k} = (p_j - kc_j) \bmod M$; if we have found $\text{TABLE}[p_{j,k}]$ occupied for all indices j, k such that $j + k < r$, and if $t \geq r + 1$, we look at $\text{TABLE}[p_{0,r}], \text{TABLE}[p_{1,r-1}], \dots, \text{TABLE}[p_{r-1,1}]$. If the first empty space occurs at position $p_{j,r-j}$ we set $\text{TABLE}[p_{j,r-j}] \leftarrow \text{TABLE}[p_j]$ 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 $\alpha^2 + \alpha^5 + \frac{1}{3}\alpha^6 + \dots$, 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 $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^6 + \dots$.

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 $\text{TABLE}[i]$.

- R1. [Empty a cell.] Mark $\text{TABLE}[i]$ 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 $\text{TABLE}[i]$.] If $\text{TABLE}[i]$ 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 $i \leq r < j$ or if $r < j < i$ or $j < i \leq r$ (in other words, if r lies cyclically between i and j), go back to R2.
- R4. [Move a record.] Set $\text{TABLE}[j] \leftarrow \text{TABLE}[i]$, 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

$$P_r = \binom{M-r}{N-r+1} / \binom{M}{N};$$

therefore the average number of probes for uniform hashing is

$$\begin{aligned} C'_N &= \sum_{1 \leq r \leq M} r P_r = M + 1 - \sum_{1 \leq r \leq M} (M + 1 - r) P_r \\ &= M + 1 - \sum_{1 \leq r \leq M} (M + 1 - r) \binom{M-r}{M-N-1} / \binom{M}{N} \end{aligned}$$

$$\begin{aligned}
&= M + 1 - \sum_{1 \leq r \leq M} (M - N) \binom{M+1-r}{M-N} / \binom{M}{N} \\
&= M + 1 - (M - N) \binom{M+1}{M-N+1} / \binom{M}{N} \\
&= M + 1 - (M - N) \frac{M+1}{M-N+1} = \frac{M+1}{M-N+1}, \quad \text{for } 1 \leq N < M.
\end{aligned} \tag{32}$$

(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

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots, \tag{33}$$

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

$$\begin{aligned}
C_N &= \frac{1}{N} \sum_{0 \leq k < N} C'_k = \frac{M+1}{N} \left(\frac{1}{M+1} + \frac{1}{M} + \dots + \frac{1}{M-N+2} \right) \\
&= \frac{M+1}{N} (H_{M+1} - H_{M-N+1}) \approx \frac{1}{\alpha} \log \frac{1}{1-\alpha}.
\end{aligned} \tag{34}$$

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”

$$a_1 a_2 \dots a_N, \quad 0 \leq a_j < M, \tag{35}$$

is equally likely, where a_j denotes the initial hash address of the j th 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 k th key, averaged over $1 \leq k \leq N$ 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 N th 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,

$$C_N = \frac{1}{N} \sum_{0 \leq k < N} C'_k, \tag{36}$$

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

$$f(M, N) = \left(1 - \frac{N}{M}\right) M^N. \quad (37)$$

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

$$g(M, N, k) = \binom{N}{k} f(k+1, k) f(M-k-1, N-k), \quad (38)$$

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

$$P_k = M^{-N} (g(M, N, k) + g(M, N, k+1) + \cdots + g(M, N, N)). \quad (39)$$

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*

$$C_N = \frac{1}{2}(1 + Q_0(M, N - 1)) \quad (\text{successful search}), \quad (40)$$

$$C'_N = \frac{1}{2}(1 + Q_1(M, N)) \quad (\text{unsuccessful search}), \quad (41)$$

where

$$\begin{aligned} Q_r(M, N) &= \binom{r}{0} + \binom{r+1}{1} \frac{N}{M} + \binom{r+2}{2} \frac{N(N-1)}{M^2} + \dots \\ &= \sum_{k \geq 0} \binom{r+k}{k} \frac{N}{M} \frac{N-1}{M} \dots \frac{N-k+1}{M}. \end{aligned} \quad (42)$$

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

$$N^k - \binom{k}{2} N^{k-1} \leq N(N-1) \dots (N-k+1) \leq N^k;$$

hence if $N/M = \alpha$,

$$\begin{aligned} \sum_{k \geq 0} \binom{r+k}{k} \left(N^k - \binom{k}{2} N^{k-1} \right) / M^k &\leq Q_r(M, N) \leq \sum_{k \geq 0} \binom{r+k}{k} N^k / M^k, \\ \sum_{k \geq 0} \binom{r+k}{k} \alpha^k - \frac{\alpha}{M} \sum_{k \geq 0} \binom{r+k}{k} \binom{k}{2} \alpha^{k-2} &\leq Q_r(M, \alpha M) \leq \sum_{k \geq 0} \binom{r+k}{k} \alpha^k, \end{aligned}$$

i.e.,

$$\frac{1}{(1-\alpha)^{r+1}} - \frac{1}{M} \binom{r+2}{2} \frac{\alpha}{(1-\alpha)^{r+3}} \leq Q_r(M, \alpha M) \leq \frac{1}{(1-\alpha)^{r+1}}. \quad (43)$$

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.*

In other words, any rearrangement of a hash sequence $a_1 a_2 \dots a_N$ 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 $a_1 a_2 \dots a_N$ is the same as the total number needed for $a_1 \dots a_{i-1} a_i a_{i+1} a_{i+2} \dots a_N$, $1 \leq i < N$. There is clearly no difference unless the $(i + 1)$ st key in the second sequence falls into the position occupied by the i th in the first sequence. But then the i th 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 i th is increased. ■

Theorem P tells us that the average search length for a hash sequence $a_1 a_2 \dots a_N$ can be determined from the numbers $b_0 b_1 \dots b_{M-1}$, where b_j is the number of a 's that equal j . From this sequence we can determine the "carry sequence" $c_0 c_1 \dots c_{M-1}$, 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

$$c_j = \begin{cases} 0, & \text{if } b_j = c_{(j+1)\bmod M} = 0; \\ b_j + c_{(j+1)\bmod M} - 1, & \text{otherwise.} \end{cases} \quad (44)$$

For example, let $M = 10$, $N = 8$, and $b_0 \dots b_9 = 0\ 3\ 2\ 0\ 1\ 0\ 0\ 0\ 2$; then $c_0 \dots c_9 = 2\ 3\ 1\ 0\ 0\ 0\ 0\ 1\ 2\ 3$, 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 $b_0 + b_1 + \dots + b_{M-1} = N$, and the average number of probes needed for retrieval of the N keys is

$$1 + (c_0 + c_1 + \dots + c_{M-1})/N. \quad (45)$$

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

$$\begin{aligned} q_0 &= p_0 q_0 + p_1 q_0 + p_0 q_1, \\ q_1 &= p_2 q_0 + p_1 q_1 + p_0 q_2, \\ q_2 &= p_3 q_0 + p_2 q_1 + p_1 q_2 + p_0 q_3, \end{aligned} \quad (46)$$

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

$$B(z)C(z) = p_0 q_0 + (q_0 - p_0 q_0)z + q_1 z^2 + \dots = p_0 q_0(1 - z) + zC(z).$$

Since $B(1) = 1$, we may write $B(z) = 1 + (z - 1)D(z)$, and it follows that

$$C(z) = \frac{p_0 q_0}{1 - D(z)} = \frac{1 - D(1)}{1 - D(z)}, \quad (47)$$

since $C(1) = 1$. The average number of probes needed for retrieval, according to (45), will therefore be

$$1 + \frac{M}{N} C'(1) = 1 + \frac{M}{N} \frac{D'(1)}{1 - D(1)} = 1 + \frac{M}{2N} \frac{B''(1)}{1 - B'(1)}. \quad (48)$$

Since we are assuming that each hash sequence $a_1 \dots a_N$ is equally likely we, have

$$\begin{aligned} p_k &= \Pr(\text{exactly } k \text{ of the } a_i \text{ are equal to } j, \text{ for fixed } j) \\ &= \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k}; \end{aligned} \quad (49)$$

hence

$$B(z) = \left(1 + \frac{z - 1}{M}\right)^N, \quad B'(1) = \frac{N}{M}, \quad B''(1) = \frac{N(N - 1)}{M^2}, \quad (50)$$

and the average number of probes according to (48) will be

$$C_N = \frac{1}{2} \left(1 + \frac{M - 1}{M - N}\right). \quad (51)$$

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 $\{0, 1, \dots, M - 1\}$, 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	$(1 - \epsilon)/6$
3 0 1 2	$(1 - \epsilon)/6$	3 1 0 2	$(1 - \epsilon)/6$

(52)

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
2 3 0 1	1/6	2 0 3 1	1/12
3 0 1 2	1/6	3 1 0 2	1/12

(53)

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 $A \subseteq \{0, 1, \dots, M-1\}$, and let $\Pi(A)$ 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 $P(A) = P_1(A) + P_2(A) + \dots$. The proof is by induction on $N \geq 1$, assuming that

$$P(A) = S(A) = 1 / \binom{M}{n}$$

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 k th 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

$$\binom{N}{k} / \left(\binom{M}{N-1} \binom{M}{k} \right), \quad \text{if } k < N;$$

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

$$\alpha = N/Mb, \quad (54)$$

for fixed α as $M, N \rightarrow \infty$. 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 size, b	Load factor, α									
	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 size, b	Load factor, α									
	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 size, b	Load factor, α									
	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

$$R(\alpha, n) = \frac{n}{n+1} + \frac{n^2\alpha}{(n+1)(n+2)} + \frac{n^3\alpha^2}{(n+1)(n+2)(n+3)} + \dots, \quad (55)$$

and

$$\begin{aligned} t_n(\alpha) &= e^{-n\alpha} \left(\frac{(\alpha n)^n}{(n+1)!} + 2 \frac{(\alpha n)^{n+1}}{(n+2)!} + 3 \frac{(\alpha n)^{n+2}}{(n+3)!} + \dots \right) \\ &= \frac{e^{-n\alpha} n^n \alpha^n}{n!} (1 - (1-\alpha) R(\alpha, n)). \end{aligned} \quad (56)$$

In terms of these functions, the average number of accesses made by the chaining method (A) in an unsuccessful search is

$$C'_N = 1 + \alpha b t_b(\alpha) + O\left(\frac{1}{M}\right) \quad (57)$$

as $M, N \rightarrow \infty$, and the corresponding number in a successful search is

$$C_N = 1 + (1 - \frac{1}{2}b(1-\alpha)) t_b(\alpha) + \frac{e^{-b\alpha} b^b \alpha^b}{2b!} R(\alpha, b) + O\left(\frac{1}{M}\right). \quad (58)$$

These 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 $M(C'_N - 1) = N t_b(\alpha)$, 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 $M \rightarrow \infty$.

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

$$\max C'_N = 1 + \frac{e^{-b} b^{b+1}}{b!} = \sqrt{\frac{b}{2\pi}} + 1 + O(b^{-1/2}), \quad (59)$$

$$\max C_N = 1 + \frac{e^{-b} b^b}{2b!} (R(b) + 1) = \frac{5}{4} + \sqrt{\frac{2}{9\pi b}} + O(b^{-1}), \quad (60)$$

as $b \rightarrow \infty$, 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

$$C_N \approx 1 + t_b(\alpha) + t_{2b}(\alpha) + t_{3b}(\alpha) + \dots, \quad (61)$$

which can be understood as follows: The average total number of accesses to look up all N keys is NC_N , and this is $N + T_1 + T_2 + \dots$, 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

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.

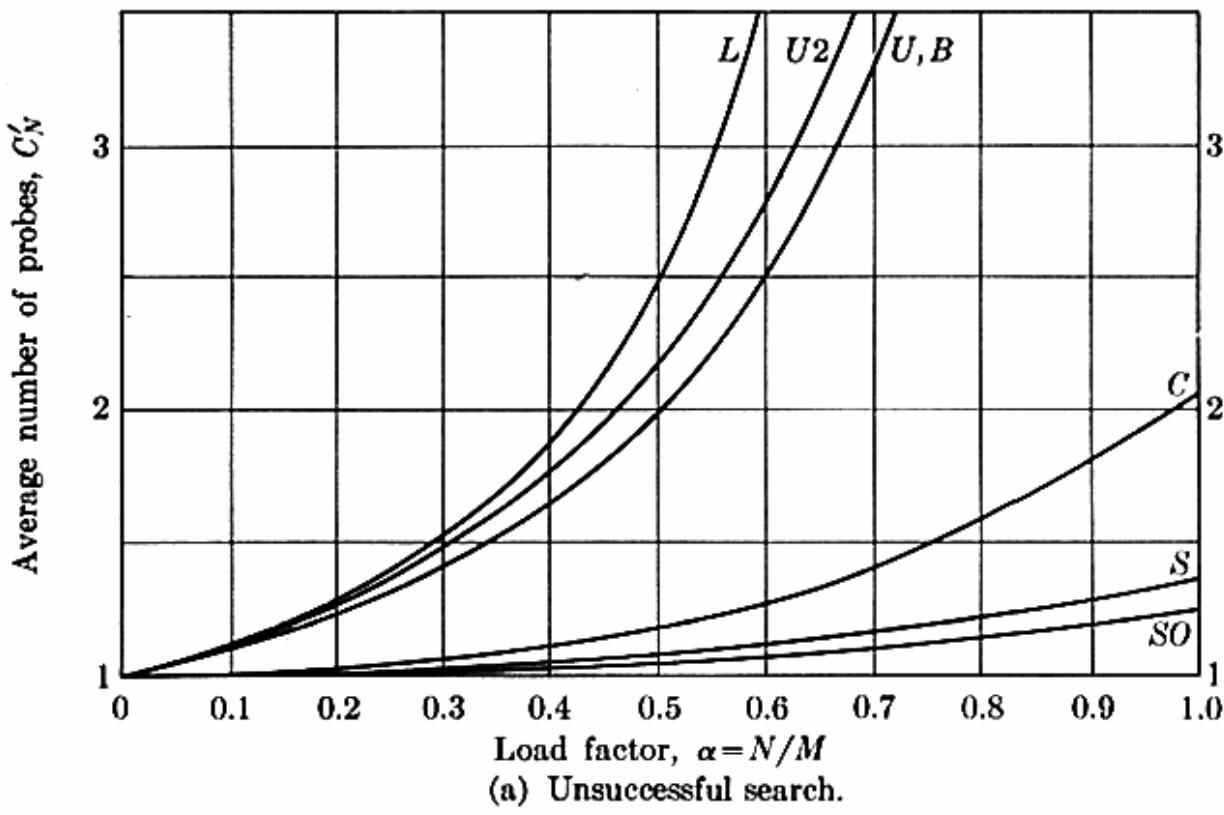
An excellent discussion of practical considerations involved in the design of external scatter tables has been given by Charles A. Olson, *Proc. ACM Nat'l Conf.* 24 (1969), 539–549. He includes several worked examples and points out that the number of overflow records will increase substantially if the 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.

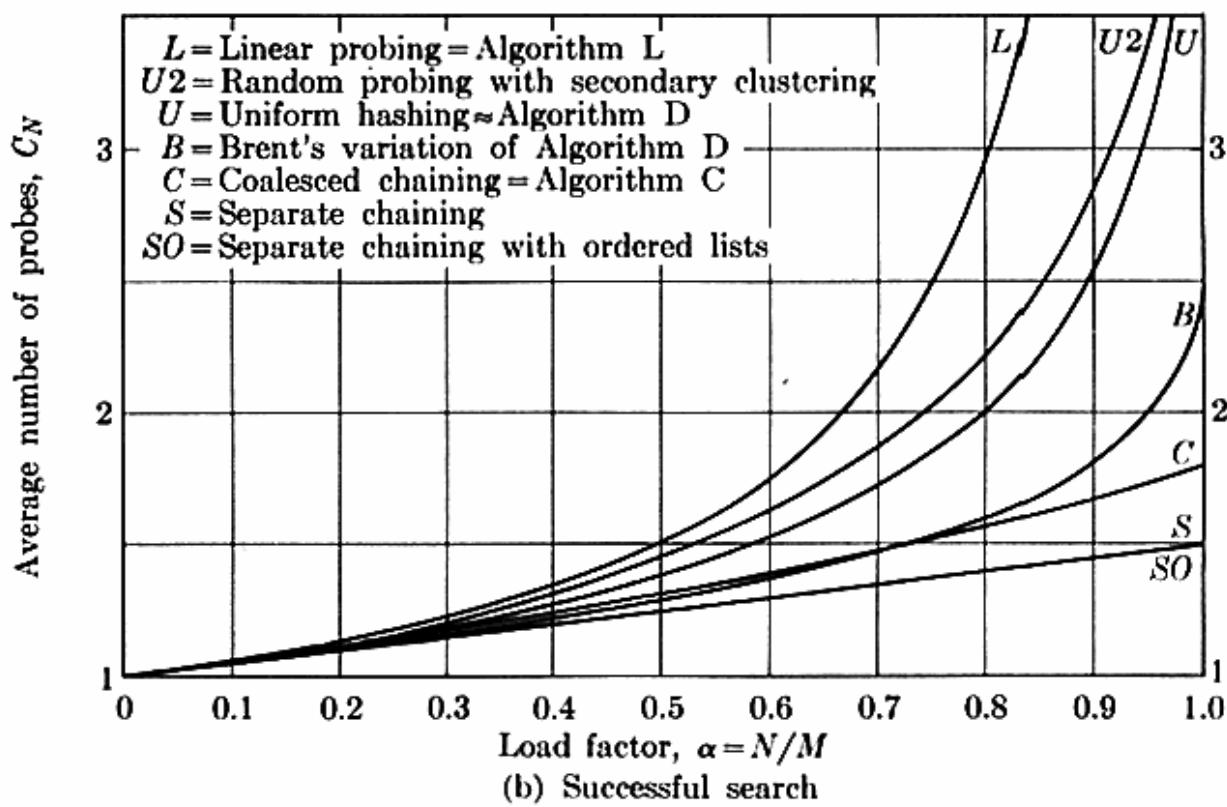
Figure 44 summarizes the analyses of this section, showing that the various methods for collision resolution lead to different numbers of probes. But this does not tell the whole story, since the time per probe varies in different methods, and the latter variation has a noticeable effect on the running 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 $N \rightarrow \infty$ 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.



(a) Unsuccessful search.



(b) Successful search

Fig. 44. Comparison of collision resolution methods: Limiting values of the average number of probes as $M \rightarrow \infty$.

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 output 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 $r11$ 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. [28] 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 $h(K) = \text{leftmost byte of } K$. 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 $\{1, 2, \dots, t\} \subseteq S$, 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 $R(x) = r_{n-1}x^{n-1} + \dots + r_1x + r_0$ 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 $\theta = /a_1, a_2, a_3, \dots/$. Let $q_0 = 0, p_0 = 1, q_1 = 1, p_1 = 0$, and $q_{k+1} = a_k q_k + q_{k-1}$, $p_{k+1} = a_k p_k + p_{k-1}$, 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 $\{\theta\}, \{2\theta\}, \{3\theta\}, \dots$ 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, ..., $s-1$) of length $\{(-1)^k(rq_k + q_{k-1})\theta\}$;
- the first $n - q_k$ intervals (numbered 0, ..., $n - q_k - 1$) of length $\{(-1)^k q_k \theta\}$;
- the last $q_k - s$ intervals (numbered $s, \dots, q_k - 1$) 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 $\{\theta\}, \{2\theta\}, \dots$ 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 $\theta, \alpha_1, \dots, \alpha_d$ are real numbers with $\alpha_1 = 0$, and if n_1, \dots, 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 $n_1 + \dots + n_d$ (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:

$\text{TAG}(P) = 0$ indicates a word in the list of available space; $\text{LINK}(P)$ points to the next available word.

$\text{TAG}(P) = 1$ indicates any word in use that is not part of the scatter table; the other fields of this word may have any desired format.

$\text{TAG}(P) = 2$ indicates a word in the scatter table; $\text{LINK}(P)$ points to another word. Whenever we are processing a list that is not part of the scatter table and we access a word with $\text{TAG}(P) = 2$, we are supposed to set $P \leftarrow \text{LINK}(P)$ until reaching a word with $\text{TAG}(P) \leq 1$. (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 $\text{TAG}(P) = 2$ also have another link field $\text{AUX}(P)$.

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 , S_1 , 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 $r = 1, 2, 3, \dots, 10$; (b) $1 \leq h_2(K) \leq \rho M$ for $\rho = \frac{1}{10}, \frac{2}{10}, \dots, \frac{9}{10}$.

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 $h_1(K), (h_1(K) - 1) \bmod M, \dots, (h_1(K) - \binom{M}{2}) \bmod M$ will be a permutation of $\{0, 1, \dots, M - 1\}$. 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 $\text{KEY}[i]$ 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.

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 $h(K_1) = 1, h(K_2) = 2, \dots, h(K_7) = 1$), as a function of M and P ?

25. [M19] Explain why Eq. (39) is true.

26. [M20] How many hash sequences $a_1 a_2 \dots a_9$ yield the pattern of occupied cells (21), using linear probing?

27. [M27] Complete the proof of Theorem K. [Hint: Let

$$s(n, x, y) = \sum_{k \geq 0} \binom{n}{k} (x+k)^{k+1} (y-k)^{n-k-1} (y-n);$$

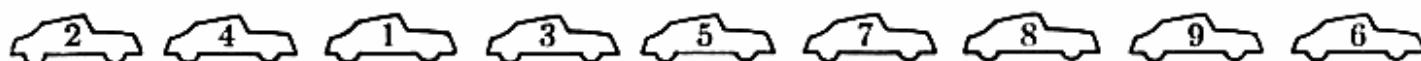
use Abel's binomial theorem, Eq. 1.2.6–16, to prove that $s(n, x, y) = x(x+y)^n + ns(n-1, x+1, y-1)$.]

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 $M \rightarrow \infty$.

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 $k, k+1, \dots, m$ are all full), he expresses his regrets and drives on.

Suppose, in fact, that this happens for n different cars, where the j th wife wakes up just in time to park at space a_j . In how many of the sequences $a_1 \dots a_n$ 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 $a_1 \dots a_9 = 3 1 4 1 5 9 2 6 5$, 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 $p_1 p_2 \dots p_n$ of $\{1, 2, \dots, n\}$ 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 $(c_0, c_1, \dots, c_{M-1})$, whenever b_0, b_1, \dots, 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 $c(k_1, k_2, k_3, \dots)$ 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 $k_1 + 2k_2 + 3k_3 + \dots = N$. Find a recurrence relation which defines these numbers $c(k_1, k_2, k_3, \dots)$, and use it to determine a simple formula for the sum

$$S_N = \sum_{\substack{j \geq 1 \\ k_1 + 2k_2 + \dots = N}} \binom{j}{2} k_j c(k_1, k_2, \dots).$$

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

$$h(K), (h(K) + p_1) \bmod M, (h(K) + p_2) \bmod M, \dots, (h(K) + p_{M-1}) \bmod M,$$

where $p_1 p_2 \dots p_{M-1}$ is a randomly chosen permutation of $\{1, 2, \dots, M - 1\}$ 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 leftmost, 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

$$\frac{7}{192}qqq + \frac{1}{6}pqq + \frac{1}{6}qpq + \frac{11}{64}qqp + \frac{1}{3}ppq + \frac{1}{4}pqp + \frac{1}{4}qpp.$$

(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, ... 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 $m \rightarrow \infty$.

45. [M48] Solve the analog of exercise 44 with *tertiary clustering*, when the probe sequence begins $h_1(K)$, $((h_1(K) + h_2(K)) \bmod M)$, 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

$$h(K), 0, 1, \dots, h(K) - 1, h(K) + 1, \dots, M - 1.$$

47. [M25] Find the average number of probes needed by open addressing when the probe sequence is

$$h(K), h(K) - 1, h(K) + 1, h(K) - 2, h(K) + 2, \dots$$

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 $\min(1, k - b + 1)$ probes in an unsuccessful search. Instead of using the exact probability P_{Nk} as in exercise 34, use the *Poisson approximation*

$$\begin{aligned} \binom{N}{k} \left(\frac{1}{M}\right)^k \left(1 - \frac{1}{M}\right)^{N-k} &= \frac{N}{M} \frac{N-1}{M} \dots \frac{N-k+1}{M} \left(1 - \frac{1}{M}\right)^N \left(1 - \frac{1}{M}\right)^{-k} \frac{1}{k!} \\ &= e^{-\rho} \rho^k / k! + O(M^{-1}), \end{aligned}$$

which is valid for $N = \rho M$ as $M \rightarrow \infty$.

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 $n \rightarrow \infty$, 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

$$1 + \frac{M}{N} C'(1) = 1 + \frac{1}{b} \left(\frac{1}{1 - q_1} + \dots + \frac{1}{1 - q_{b-1}} - \frac{1}{2} \frac{B''(1) - b(b-1)}{B'(1) - b} \right),$$

where q_1, \dots, 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 $p_1 p_2 \dots p_{M-1}$ 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?

6.5. RETRIEVAL ON SECONDARY KEYS

We have now completed our study of searching for "primary keys," i.e., for keys which uniquely specify a record in a file. But it is sometimes necessary to conduct a search based on the values of other fields in the records besides the primary key; these other fields are often called "secondary keys" or "attributes" of the record. For example, in an enrollment file which contains information about the students at a university, it may be desirable to search for all sophomores from Ohio who are not majoring in mathematics or statistics; or to search for all unmarried French-speaking graduate student women; etc.

In general, we assume that each record contains several attributes, and we want to search for all records that have certain values of certain attributes. The specification of which records are desired is called a *query*. Queries are usually restricted to at most the following three types:

- a) A *simple query* which gives a specific value of a specific attribute; e.g.,
MAJOR = MATHEMATICS; or RESIDENCE.STATE = OHIO.
- b) A *range query* which gives a specific range of values for a specific attribute;
e.g., COST < \$18.00; or 21 ≤ AGE ≤ 23.
- c) A *Boolean query* which consists of the previous types combined with the operations AND, OR, NOT; e.g.,

(CLASS = SOPHOMORE) AND (RESIDENCE.STATE = OHIO)
AND NOT ((MAJOR = MATHEMATICS) OR (MAJOR = STATISTICS)).

The problem of discovering efficient search techniques for these three types of queries is already quite difficult, and therefore queries of more complicated types are usually not considered. For example, a railroad company might have a file giving the current status of all its freight cars; a query such as "find all empty refrigerator cars within 500 miles of Seattle" would not be explicitly allowed, unless "distance from Seattle" were an attribute stored within each record instead of a complicated function to be deduced from other attributes. And the use of logical quantifiers, in addition to AND, OR, and NOT, would introduce further complications, limited only by the imagination of the query-poser; given a file of baseball statistics, for example, we might ask for the longest consecutive hitting streak in night games. These examples are complicated, but they can still be handled by taking one pass through a suitably-arranged file; other queries are even more difficult, e.g. to find all pairs of records that have the same values on five or more attributes (without specifying which attributes must match). Such queries may be regarded as general programming tasks which are beyond the scope of this discussion, although they can often be broken down into subproblems of the kind considered here.

Before we begin to study the various techniques for secondary key retrieval, it is important to put the subject in a proper economic context. Although a vast number of applications fit into the general framework of the three types of queries outlined above, not many of these applications are really suited to the

sophisticated techniques we shall be studying, and some of them are better done by hand than by machine! Computers have increased the speed of scientific calculations by a factor of 10^7 or 10^8 , but they have provided nowhere near this gain in efficiency with respect to problems of information handling. When large quantities of data are involved, today's computers are still constrained to working at mechanical (instead of electronic) speeds, so there is no dramatic increase in performance per unit cost when a computer system replaces a manual system. We shouldn't expect too much of a computer just because it performs certain other tasks so well.

People climb Mt. Everest "because it is there" and because tools have been developed which make the climb possible; similarly, when faced with a mountain of data, people are tempted to use a computer to find the answer to the most difficult queries they can dream up, in an "on-line, real-time" environment, without properly balancing the cost. The desired calculations are possible, but they're not right for everyone's application.

For example, consider the following simple approach to secondary key retrieval: After *batching* a number of queries, we can do a sequential search through the entire file, retrieving all the relevant records. ("Batching" means that we accumulate a number of queries before doing anything about them.) This method is quite satisfactory if the file isn't too large and if the queries don't have to be handled immediately. It can be used even with tape files, and it only ties up the computer at odd intervals, so it will tend to be very economical in terms of equipment costs. Moreover, it will even handle computational queries of the "distance to Seattle" type discussed above.

Another simple way to facilitate secondary key retrieval is to let *people* do part of the work, by providing them with suitable printed indexes to the information. This method is often the most reasonable and economical way to proceed (provided, of course, that the old paper is recycled whenever a new index is printed).

The applications which are not satisfactorily handled by the above simple schemes involve very large files for which quick responses to queries are important. Such a situation would occur, for example, if the file were continuously being queried by a number of simultaneous users, or if the queries were being generated by machine instead of by people. Our goal in this section will be to see how well we can do secondary key retrieval with conventional computers, under various assumptions about the file structure.

A lot of good ideas have been developed for dealing with the problem, but (as the reader will have guessed from all these precautionary remarks) the algorithms are by no means as good as those available for primary key retrieval. Because of the wide variety of files and applications, we will not be able to give a complete discussion of all the possibilities that have been considered, or to analyze the behavior of each algorithm in typical environments. The remainder of this section presents the basic approaches which have been proposed, and it is left to the reader's imagination to decide what combination of techniques is most appropriate in each particular case.

Inverted files. The first important class of techniques for secondary key retrieval is based on the idea of an "inverted file." This does not mean that the file is turned upside down, it means that the roles of records and attributes are reversed. Instead of listing the attributes of a given record, we list the records having a given attribute.

We encounter inverted files (under other names) quite often in our daily lives. For example, the inverted file corresponding to a Russian-English dictionary is an English-Russian dictionary. The inverted file corresponding to this book is the index which appears at the close of the book. Accountants traditionally use "double-entry bookkeeping," where all transactions are entered both in a cash account and in a customer account, so that the current cash position and customer liability are both readily accessible.

In general, an inverted file usually doesn't stand by itself, it is to be used together with the original uninverted file; it provides duplicate, redundant information in order to speed up secondary key retrieval. The components of an inverted file are called *inverted lists*, namely the lists of all records having a given value of some attribute.

Like all lists, the inverted lists can be represented in many ways within a computer, and different modes of representation are appropriate at different times. Some secondary key fields have only two values (e.g., a 'sex' attribute), and the corresponding inverted lists are quite long; but other fields typically have a great many values with few duplications (e.g., a 'telephone number' attribute).

Imagine, for example, that we want to store the information in a telephone directory so that all entries can be retrieved on the basis of either name, phone number, or residence address. One solution is simply to make three separate files, oriented to retrieval on each type of key. Another idea is to combine these files, for example by making three hash tables which serve as the list heads for the chaining method. In the latter scheme, each record of the file would be an element of three lists, and it would therefore contain three link fields; this is the so-called *Multilist* method which is discussed further below. A third possibility is to combine the three files into one super file, by analogy with library card catalogues in which author cards, title cards, and subject cards are all alphabetized together.

A consideration of the format used in the index to this book leads to further ideas on inverted list representation. For secondary key fields in which there are typically five or so entries per attribute value, we can simply make a short sequential list of the record locations (analogous to page locations in a book index), following the key value. If related records tend to be clustered consecutively, a "range specification" code (e.g., pages 200 through 214) could be useful. If the records in the file tend to be reallocated frequently, it may be better to use primary keys instead of record locations in the inverted files, so that no updating needs to be done when the locations change; for example, references to Bible passages are always given by chapter and verse, and the index to some books is based on paragraph numbers instead of page numbers.

None of these ideas is especially appropriate for the case of a two-valued attribute like 'SEX'. In such a case only one inverted list is needed, of course, since the non-males will be female and conversely. If each value relates to about half the items of the file, the inverted list will be horribly long, but we can solve the problem rather nicely on a binary computer by using a bit string representation, with each bit specifying the value of a particular record. Thus the bit string 01001011101 . . . might mean that the first record in the file refers to a male, the second female, the next two male, etc. (Cf. also the discussion of prime number representation at the end of Section 6.1.)

The above methods suffice to handle simple queries about specific attribute values. A slight extension makes it possible to treat range queries, except that some comparison-based search scheme (Section 6.2) must be used instead of hashing.

For Boolean queries like "(MAJOR = MATHEMATICS) AND (RESIDENCE.STATE = OHIO)", we need to intersect two inverted lists. This can be done in several ways; for example, if both lists are ordered, one pass through each one will pick out all common entries. Alternatively, we could select the *shortest* list and look up each of its records, checking the other attributes; but this method works only for AND's, not for OR's, and it is unattractive on external files because it requires many accesses to records that will not satisfy the query.

The same considerations show that a Multilist organization as described above is inefficient for Boolean queries, on an external file, since it implies many unnecessary accesses. For example, imagine what would happen if the index to this book were organized in a Multilist manner. This would mean that each entry of the index would refer only to the last page on which its particular subject was mentioned; then on every page there would be a further reference, for each subject on that page, to the previous occurrence of that subject. In order to find all pages relevant to "[Analysis of algorithms] and [(External sorting) or (External searching)]", it would be necessary to turn many pages. On the other hand, the same query can be resolved by looking at only two pages of the real index as it actually appears, doing simple operations on the inverted lists in order to find the small subset of pages which satisfy the query.

When an inverted list is represented as a bit string, Boolean combinations of simple queries are, of course, rather easily performed, because computers can manipulate bit strings at relatively high speed. For mixed queries in which some attributes are represented as sequential lists of record numbers while other attributes are represented as bit strings, it is not difficult to convert the sequential lists into bit strings, then to perform the Boolean operations on these bit strings.

A quantitative example of a hypothetical application may be helpful at this point. Assume that we have 1,000,000 records of 40 characters each, and that our file is stored on MIXTEC disks, as described in Section 5.4.9. The file itself therefore fills two disk packs, and the inverted lists will probably fill several more. Each track contains 5000 characters = 30,000 bits, so an inverted list for a particular attribute will take up at most 34 tracks. (This maximum

number of tracks occurs when the bitstring representation is the shortest possible one.) Suppose that we have a rather involved query that refers to a Boolean combination of 10 inverted lists; in the worst case we will have to read 340 tracks of information from the inverted file, for a total read time of $340 \times 25 \text{ ms} = 8.5 \text{ sec}$. The average latency delay will be about one half of this, but by careful programming it may be possible to eliminate the latency. By storing the first track of each bitstring list in one cylinder, and the second track of each list in the next, etc., most of the seek time will be eliminated, so we can estimate the seek time as at most about $34 \times 26 \text{ ms} \approx 0.9 \text{ sec}$ (or twice this if two different disk packs are involved). Finally, if k records satisfy the query, it will take about $k \times (60 \text{ ms (seek)} + 12.5 \text{ ms (latency)} + 0.2 \text{ ms (read)})$ extra time to fetch each one for subsequent processing. Thus an optimistic estimate of the total expected time to process this rather complicated query is $<(10 + .073k)$ seconds. This may be contrasted with about 210 seconds to read through the entire file at top speed under the same assumptions without using any inverted lists.

This example shows that space optimization is closely related to time optimization in a disk memory; the time to process the inverted lists is roughly the time needed to seek and to read them.

The above discussion has more or less assumed that the file is not growing or shrinking as we query it; what should we do if frequent updates are necessary? In many applications it is sufficient to batch a number of requests for updates, and to take care of them in dull moments when no queries need to be answered. Alternatively, if updating the file has high priority, the method of *B*-trees (Section 6.2.4) is attractive. The entire collection of inverted lists could be made into one huge *B*-tree, with special conventions for the leaves so that the branch nodes contain key values while the leaves contain both keys and lists of pointers of records.

We have glossed over another point in the above discussion, namely the difficult subject of Boolean combinations of range queries. For example, suppose that the records of the file refer to North American cities, and that the query asks for all cities with

$$(21.49^\circ \leq \text{LATITUDE} \leq 37.41^\circ) \text{ AND } (70.34^\circ \leq \text{LONGITUDE} \leq 75.72^\circ).$$

No really nice data structures seem to be available for such "orthogonal range queries." (Reference to a map will show that many cities satisfy this LATITUDE range, and many satisfy the LONGITUDE range, but hardly any cities lie in both ranges.) Perhaps the best approach is to partition the set of all possible LATITUDE and LONGITUDE values rather coarsely, with only a few classes per attribute (e.g., by truncating to the next lower multiple of 5°), then to have one inverted list for each combined (LATITUDE, LONGITUDE) class. This is like having maps

with one page for each local region. Using 5° intervals, the above query would refer to eight pages, namely $(20^\circ, 70^\circ)$, $(25^\circ, 70^\circ)$, \dots , $(35^\circ, 75^\circ)$. The range query needs to be processed for each of these pages, either by going to a finer partition within the page or by direct reference to the records themselves, depending on the number of records corresponding to that page. In a sense this is a tree structure with two-dimensional branching at each internal node.

Another tree-structured approach to orthogonal range queries has been proposed by Bruce McNutt. Suppose, for example, that we wish to handle queries like "What is the nearest city to point x ?", given the value of x . Each node of McNutt's proposed binary search tree corresponds to a city y and a "test radius" r ; the left subtree of this node corresponds to all cities z entered subsequently into this part of the tree such that the distance from y to z is $\leq r + \delta$, and the right subtree similarly is for distances $\geq r - \delta$. Here δ is a given tolerance; cities between $r - \delta$ and $r + \delta$ away from y must be entered in *both* subtrees. Searching in such a "post-office tree" makes it possible to locate all cities within distance δ of a given point. (See Fig. 45.)

Several experiments based on this idea were conducted in 1972 by McNutt and Edward Pring, using the 231 most populous cities in the continental United States, in random order, as an example data base. They let the test radii shrink in a regular manner, replacing r by $0.67r$ when going to the left, and by $0.57r$ when going to the right, except that r was left unchanged when taking the second of two consecutive right branches. The result was that 610 nodes were required in the tree for $\delta = 20$ miles, and 1600 nodes were required for $\delta = 35$ miles. The top levels of their smaller tree are shown in Fig. 45. (In the remaining levels of this tree, Orlando FL appeared below both Jacksonville and Miami. Some cities occurred quite often; e.g., 17 of the nodes were for Brockton MA!)

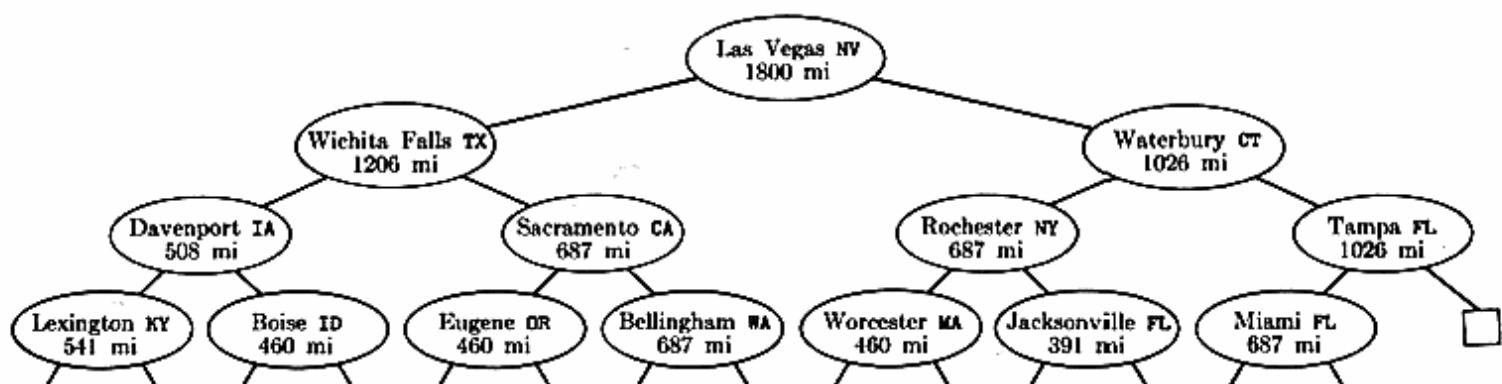


Fig. 45. The top levels of an example "post-office tree." To search for all cities near a given point x , start at the root: If x is within 1800 miles of Las Vegas, go left, otherwise go to the right; then repeat the process until encountering a terminal node. The method of tree construction ensures that all cities within 20 miles of x will be encountered during this search.

Table 1 A FILE WITH BINARY ATTRIBUTES

	Special ingredients	Vanilla extract	Sugar, powdered	Sugar, granulated	Sugar, brown	Milk	Molasses	Nutmeg	Nuts	Oatmeal	Raisins	Salt	Salt	Sugar, granulated	Sugar, powdered	Vanilla extract	Special ingredients
Almond Lace Wafers	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Applesauce-Spice Squares	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	Applesauce
Banana-Oatmeal Cookies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Bananas
Chocolate Chip Cookies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Coconut Macaroons	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Cream cheese
Cream-Cheese Cookies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Oranges, prunes
Delicious Prune Bars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Double-Chocolate Drops	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Dream Bars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Filled Turnovers	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Finska Kakor	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Glazed Gingersnaps	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Hermits	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Jewel Cookies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Jumbles	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Kris Kringles	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Lebkuchen Rounds	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Meringues	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Moravian Spice Cookies	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Oatmeal-Date Bars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Old-Fashioned Sugar Cookies	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Sour cream
Peanut-Butter Pinwheels	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Peanut butter
Petticoat Tails	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	—
Pfeffernusse	1	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	Citron, mace, pepper
Scotch Oatmeal Shortbread	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Shortbread Stars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	—
Springerle	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	—
Spritz Cookies	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	—
Swedish Kringle	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	—
Swiss-Cinnamon Crisps	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	—
Toffee Bars	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	—
Vanilla-Nut Icebox Cookies	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	—

Reference: McCall's Cook Book (New York: Random House, 1963), Chapter 9.

Compound attributes. It is possible to combine two or more attributes into one super attribute. For example, a “(CLASS, MAJOR) attribute” could be created by combining the CLASS and MAJOR fields of a university enrollment file. In this way queries can often be satisfied by taking the union of disjoint, short lists instead of the intersection of longer lists.

The idea of attribute combination has been further developed by V. Y. Lum [CACM 13 (1970), 660–665], who suggests ordering the inverted lists of combined attributes lexicographically from left to right, and making multiple copies, with the individual attributes permuted in a clever way. For example, suppose that we have three attributes A, B, C; we can form three combined attributes

$$(A, B, C), \quad (B, C, A), \quad (C, A, B) \quad (1)$$

and construct ordered inverted lists for each of these. (Thus in the first list, the records occur in order of their A values, with all records of the same A value in order by B and then by C.) This organization makes it possible to satisfy queries based on any combination of the three attributes; e.g., all records having specified values for A and C will appear contiguously in the third list.

Similarly, from four attributes A, B, C, D, we can form the six combined attributes

$$(A, B, C, D), \quad (B, C, D, A), \quad (B, D, A, C), \quad (C, A, D, B), \quad (C, D, A, B), \quad (D, A, B, C), \quad (2)$$

which suffice to answer all combinations of simple queries relating to the simultaneous values of one, two, three, or four of the attributes. There is a general procedure for constructing $\binom{n}{k}$ combined attributes from n attributes, where $k \leq \frac{1}{2}n$, such that all records having specified combinations of $\leq k$ or $\geq n - k$ of the attribute values will appear consecutively in one of the combined attribute lists (see exercise 1). Alternatively, we can get by with fewer combinations when some attributes have a limited number of values. For example if D is simply a two-valued attribute, the three combinations

$$(D, A, B, C), \quad (D, B, C, A), \quad (D, C, A, B), \quad (3)$$

obtained by placing D in front of (1), will be almost as good as (2) with only half the redundancy, since queries that do not depend on D can be treated by looking in just two places in one of the lists.

Binary attributes. It is instructive to consider the special case in which all attributes are two-valued. In a sense this is the *opposite* of combining attributes, since we can represent any value as a binary number and regard the individual bits of that number as separate attributes. Table 1 shows a typical file involving “yes-no” attributes; in this case the records stand for selected cookie recipes, and the attributes specify which ingredients are used. For example, Almond Lace Wafers are made from butter, flour, milk, nuts, and granulated sugar.

If we think of Table 1 as a matrix of zeros and ones, the transpose of the matrix is the "inverted" file, in bitstring form.

The right-hand column of Table 1 is used to indicate special items that occur only rarely. These could be coded in a more efficient way than to devote an entire column to each one; the "Cornstarch" column could be similarly treated. Dually, we could find a more efficient way to encode the "Flour" column, since flour occurs in everything except Meringues. For the present, however, let us sidestep these considerations and simply ignore the "Special ingredients" column.

Let us define a *basic query* in a binary attribute file as a request for all records having 0's in certain columns, 1's in other columns, and arbitrary values in the remaining columns. Using "*" to stand for an arbitrary value, we can represent any basic query as a sequence of 0's, 1's, and *'s. For example, consider a man who is in the mood for some coconut cookies, but he is allergic to chocolate, hates anise, and has run out of vanilla extract; he can formulate the query

* 0 * * * * 0 * * 1 * 0. (4)

Table 1 now says that Dream Bars are just the thing.

Before we consider the general problem of organizing a file for basic queries, it is important to look at the special case where no 0's are specified, only 1's and *'s. This may be called an *inclusive query*, because it asks for all records which include a certain set of attributes, if we assume that 1's denote attributes that are present and that 0's denote attributes that are absent. For example, the recipes in Table 1 which call for both baking powder and baking soda are Glazed Gingersnaps and Old-Fashioned Sugar Cookies.

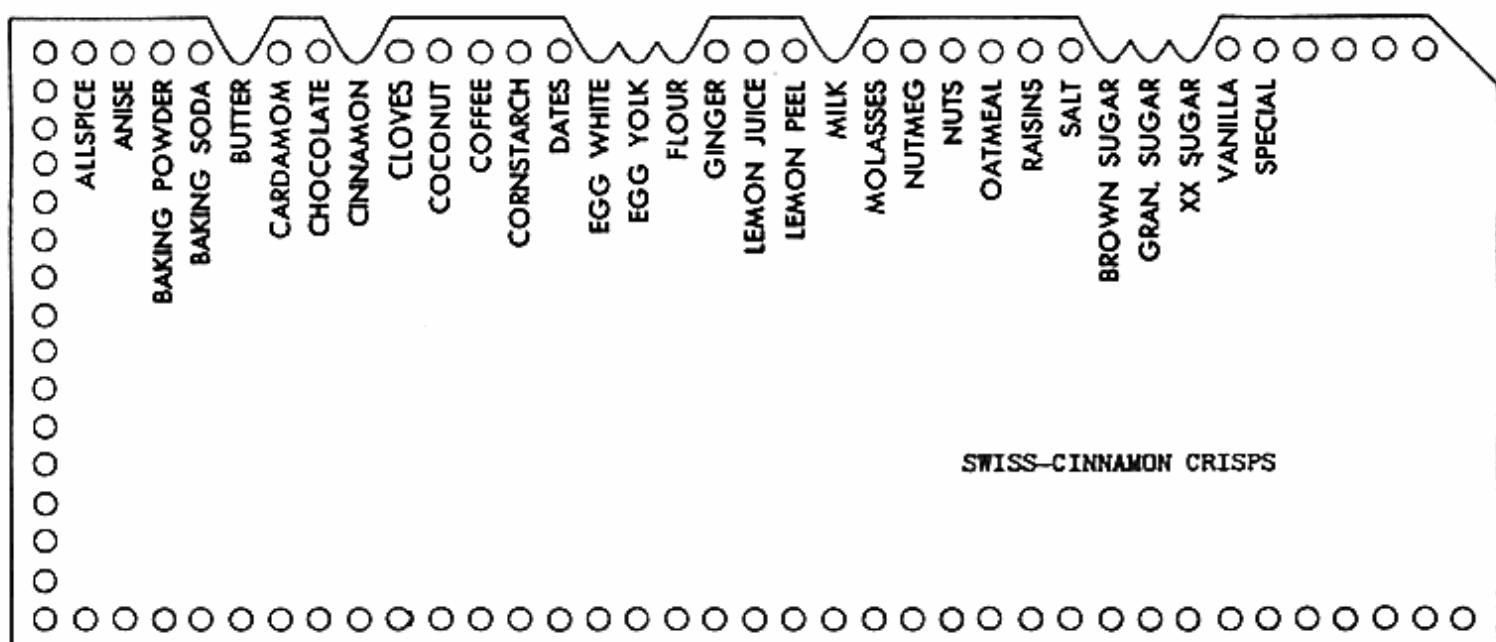


Fig. 46. An edge-notched card.

In some applications it is sufficient to provide for the special case of inclusive queries. This occurs, for example, in the case of many manual card-filing systems, such as "edge-notched cards" or "feature cards." An edge-notched card system corresponding to Table 1 would have one card for every recipe, with holes cut out for each ingredient (see Fig. 46). In order to process an inclusive query, the file of cards is arranged into a neat deck and needles are put in each column position corresponding to an attribute that is to be included. After raising the needles, all cards having the appropriate attributes will drop out.

A feature-card system works on the inverse file in a similar way. In this case there is one card for every attribute, and holes are punched in designated positions on the surface of the card for every record possessing that attribute. An ordinary 80-column card can therefore be used to tell which of $12 \times 80 = 960$ records have a given attribute. To process an inclusive query, the feature cards for the specified attributes are selected and put together; then light will shine through all positions corresponding to the desired records. This operation is analogous to the treatment of Boolean queries by intersecting inverted bit strings as explained above.

Superimposed coding. The reason these manual card systems are of special interest to us is that ingenious schemes have been devised to save space on edge-notched cards; the same principles can be applied in the representation of computer files. Superimposed coding is a technique similar to hashing, and it was actually invented several years before hashing itself was discovered. The idea is to map attributes into random k -bit codes in an n -bit field, and to superimpose the codes for each attribute that is present in a record. An inclusive query for some set of attributes can be converted into an inclusive query for the corresponding superimposed bit codes. A few extra records may satisfy this query, but the number of such "false drops" can be statistically controlled. [Cf. Calvin N. Mooers, *Amer. Chem. Soc. Meeting* 112 (September, 1947), 14E-15E; *American Documentation* 2 (1951), 20-32.]

As an example of superimposed coding, let's consider Table 1 again, but only the flavorings instead of the basic ingredients like baking powder, shortening, eggs, and flour. Table 2 shows what happens if we assign random 2-bit codes in a 10-bit field to each of the flavoring attributes and superimpose the coding. For example, the entry for Chocolate Chip Cookies is obtained by superimposing the codes for chocolate, nuts, and vanilla:

$$0010001000 \vee 0000100100 \vee 0000001001 = 0010101101.$$

The superimposition of these codes also yields some spurious attributes, in this case allspice, candied cherries, currant jelly, peanut butter, and pepper; these will cause false drops to occur on certain queries (and they also suggest the creation of a new recipe called False Drop Cookies!).

Table 2
AN EXAMPLE OF SUPERIMPOSED CODING

Codes for individual flavorings			
Almond extract	0100000001	Dates	1000000100
Allspice	0000100001	Ginger	0000110000
Anise seed	0000011000	Honey	0000000011
Applesauce	0010010000	Lemon juice	1000100000
Apricots	1000010000	Lemon peel	0011000000
Bananas	0000100010	Mace	0000010100
Candied cherries	0000101000	Molasses	1001000000
Cardamom	1000000001	Nutmeg	0000010010
Chocolate	0010001000	Nuts	0000100100
Cinnamon	1000000010	Oranges	0100000100
Citron	0100000010	Peanut butter	0000000101
Cloves	0001100000	Pepper	0010000100
Coconut	0001010000	Prunes	0010000010
Coffee	0001000100	Raisins	0101000000
Currant jelly	0010000001	Vanilla extract	0000001001
Superimposed codes			
Almond Lace Wafers	0000100100	Lebkuchen Rounds	1011110111
Applesauce-Spice Squares	1111111111	Meringues	1000101100
Banana-Oatmeal Cookies	1000111111	Moravian Spice Cookies	1001110011
Chocolate Chip Cookies	0010101101	Oatmeal-Date Bars	1000100100
Coconut Macaroons	0001111101	Old-Fashioned Sugar Cookies	0000011011
Cream-Cheese Cookies	0010001001	Peanut-Butter Pinwheels	0010001101
Delicious Prune Bars	0111110110	Petticoat Tails	0000001001
Double-Chocolate Drops	0010101100	Pfeffernuesse	1111111111
Dream Bars	0001111101	Scotch Oatmeal Shortbread	0000001001
Filled Turnovers	1011101101	Shortbread Stars	0000000000
Finska Kakor	0100100101	Springerle	0011011000
Glazed Gingersnaps	1001110010	Spritz Cookies	0000001001
Hermits	1101010110	Swedish Kringler	0000000000
Jewel Cookies	0010101101	Swiss-Cinnamon Crisps	1000000010
Jumbles	1000001011	Toffee Bars	0010111101
Kris Kringles	1011100101	Vanilla-Nut Icebox Cookies	0000101101

Superimposed coding actually doesn't work very well in Table 2, because that table is a small example with lots of attributes present. In fact, Applesauce-Spice Squares will drop out for *every* query, since it was obtained by superimposing seven codes that cover all ten positions; and Pfefferneusse is even worse, obtained by superimposing twelve codes! On the other hand Table 2 works surprisingly well in some respects; e.g., for the query "Vanilla extract", only the record for Pfefferneusse comes out as a false drop.

A more appropriate example of superimposed coding occurs if we have, say, a 32-bit field and a set of $\binom{32}{3} = 4960$ different attributes, where each record is allowed to possess up to six attributes and each attribute is encoded by specifying 3 of the 32 bits. In this situation, if we assume that each record has six randomly selected attributes, the probability of a false drop in an inclusive query

on one attribute is	.0806933;
on two attributes is	.0070878;
on three attributes is	.0006709;
on four attributes is	.0000679;
on five attributes is	.0000073;
on six attributes is	.0000008.

(5)

Thus if there are M records which do not actually satisfy a two-attribute query, about $.007M$ will have a superimposed code that spuriously matches all code bits of the two specified attributes. (These probabilities are computed in exercise 4.) The total number of bits needed in the inverted file is only 32 times the number of records, which is less than half the number of bits needed to specify the attributes themselves in the original file.

Malcolm C. Harrison [CACM 14 (1971), 777–779] has observed that superimposed coding can be used to speed up *text searching*. Assume that we want to locate all occurrences of a particular string of characters in a long body of text, without building an extensive table as in Algorithm 6.3P; and assume, for example, that the text is divided into individual lines $c_1c_2 \dots c_{50}$ of 50 characters each. Harrison suggests encoding each of the 49 pairs $c_1c_2, c_2c_3, \dots, c_{49}c_{50}$ by hashing each of them into a number between 0 and 127, say; then the “signature” of the line $c_1c_2 \dots c_{50}$ is the string of 128 bits $b_0b_1 \dots b_{127}$, where $b_i = 1$ if and only if $h(c_jc_{j+1}) = i$ for some j .

If now we want to search for all occurrences of the word NEEDLE in a large text file called HAYSTACK, we simply look for all lines whose signature contains 1-bits in positions $h(\text{NE}), h(\text{EE}), h(\text{ED}), h(\text{DL}),$ and $h(\text{LE})$. Assuming that the hash function is random, the probability that a random line contains all these bits in its signature is only 0.00341 (cf. exercise 4); hence the intersection of five inverted-list bit strings will rapidly identify all the lines containing NEEDLE, together with a few false drops.

The assumption of randomness is not really justified in this application, since typical text has so much redundancy; the distribution of bigrams, i.e. of two-letter combinations in words, is highly biased. For example, it will probably be very helpful to discard all pairs c_jc_{j+1} containing a blank character, since blanks are usually much more common than any other symbol.

Another interesting application of superimposed coding to search problems has been suggested by Burton H. Bloom [CACM 13 (1970), 422–426]; his method actually applies to *primary* key retrieval, although it is most appropriate for us to discuss it in this section. Imagine a search application with a large data base

in which no calculation needs to be done if the search was unsuccessful. For example, we might want to check somebody's credit rating or passport number, etc., and if no record for him appears in the file we don't have to investigate further. Similarly in an application to computerized typesetting, we might have a simple algorithm that hyphenates most words correctly, but it fails on some 50,000 exceptional words; if we don't find the word in the exception file we are free to use the simple algorithm.

In such situations it is possible to maintain a bit table in internal memory so that most keys not in the file can be recognized as absent without making any references to the external memory. Here's how: Let the internal bit table be $b_0 b_1 \dots b_{M-1}$, where M is rather large. For each key K_j in the file, compute k independent hash functions $h_1(K_j), \dots, h_k(K_j)$, and set the corresponding k b 's equal to 1. (These k values need not be distinct.) Thus $b_i = 1$ if and only if $h_l(K_j) = i$ for some j and l . Now to determine if a search argument K is in the external file, first test whether or not $b_{h_l(K)} = 1$ for $1 \leq l \leq k$; if not, there is no need to access the external memory, but if so, a conventional search will probably find K if k and M have been chosen properly. The chance of a false drop when there are N records in the file is approximately $(1 - e^{-kN/M})^k$. In a sense, Bloom's method treats the entire file as one record, with the primary keys as the attributes which are present, and with superimposed coding in a huge M -bit field.

Still another variation of superimposed coding has been developed by Richard A. Gustafson [Ph.D. thesis (Univ. South Carolina, 1969)]. Suppose that we have N records and that each record possesses six attributes chosen from a set of 10,000 possible attributes. The records may, for example, stand for technical articles and the attributes may be keywords describing the article. Let h be a hash function which maps each attribute into a number between 0 and 15. If a record has attributes a_1, a_2, \dots, a_6 , Gustafson suggests mapping the record into the 16-bit number $b_0 b_1 \dots b_{15}$, where $b_i = 1$ if and only if $h(a_j) = i$ for some j ; and furthermore if this method results in only k of the b 's equal to 1, for $k < 6$, another $6 - k$ 1's are supplied by some random method (not necessarily depending on the record itself). There are $\binom{16}{6} = 8008$ sixteen-bit codes in which exactly six 1 bits are present, and with luck about $N/8008$ records will be mapped into each value. We can keep 8008 lists of records, directly calculating the address corresponding to $b_0 b_1 \dots b_{15}$ using a suitable formula. In fact, if the 1's occur in positions $0 \leq p_1 < p_2 < \dots < p_6$, the function

$$\binom{p_1}{1} + \binom{p_2}{2} + \dots + \binom{p_6}{6}$$

will convert each string $b_0 b_1 \dots b_{15}$ into a unique number between 0 and 8007, as we have seen in exercises 1.2.6–56, 2.2.6–7.

Now if we want to find all records having three particular attributes A_1, A_2, A_3 , we compute $h(A_1), h(A_2), h(A_3)$; assuming that these three values are distinct, we need only look at the records stored in the $\binom{13}{3} = 286$ lists whose

bit code $b_0 b_1 \dots b_{15}$ contains 1's in those three positions. In other words, only $286/8008 \approx 3.5$ percent of the records need to be examined in the search.

Combinatorial hashing. The idea underlying Gustafson's method just described is to find some way to map the records into memory locations so that comparatively few locations are relevant to a particular query. But his method applies only to inclusive queries when the individual records possess few attributes. Another type of mapping, designed to handle arbitrary basic queries like (4) consisting of 0's, 1's, and *'s, was discovered by Ronald L. Rivest in 1971.

Suppose first that we have 1,000,000 records each containing 10 secondary keys, where each secondary key has a fairly large number of possible values. We can map the records whose secondary keys are $(K_1, K_2, \dots, K_{10})$ into the 20-bit number

$$h(K_1)h(K_2) \dots h(K_{10}), \quad (6)$$

where h is a hash function taking each secondary key into a 2-bit value, and (6) stands for the juxtaposition of these ten pairs of bits. This scheme maps 1,000,000 records into $2^{20} = 1,048,576$ possible values, and we can consider the total mapping as a hash function with $M = 2^{20}$; chaining can be used to resolve collisions. If we want to retrieve all records having specified values of any five secondary keys, we need to look at only 2^{10} lists [corresponding to the five unspecified bit pairs in (6)], so only about $1000 = \sqrt{N}$ records need to be examined on the average. (A similar approach has been suggested by M. Arisawa, *J. Inf. Proc. Soc. Japan* 12 (1971), 163–167.)

Rivest has developed this idea further so that in many cases we have the following situation. Assume that there are $N \approx 2^n$ records, each having m secondary keys. Each record is mapped into an n -bit hash address, in such a way that a query which leaves the values of k keys unspecified corresponds to approximately $N^{k/m}$ hash addresses. All the other methods we have discussed in this section (except Gustafson's) require order N steps for retrieval, although the constant of proportionality is small; for large enough N , Rivest's method will be faster, and it requires no inverted files.

But we have to define an appropriate mapping, before we can apply this technique. Here is an example with small parameters, when $m = 4$ and $n = 3$ and when all secondary keys are binary-valued; we can map 4-bit records into eight addresses as follows:

* 0 0 0 → 1	0 1 * 0 → 5
* 1 1 1 → 2	1 0 * 1 → 6
0 * 0 1 → 3	0 0 1 * → 7
1 * 1 0 → 4	1 1 0 * → 8

(7)

An examination of this table reveals that all records corresponding to the query $0 * * *$ are mapped into locations 1, 2, 3, 5, and 7; and similarly *any* basic query with three *'s corresponds to exactly five locations. The basic queries with two *'s each correspond to three locations; and the basic queries with one * cor-

respond to either one or two locations, $(8 \times 1 + 24 \times 2)/32 = 1.75$ on the average. Thus we have

Number of unspecified bits in the query	Number of locations to search	
4	$8 = 8^{4/4}$	
3	$5 \approx 8^{3/4}$	
2	$3 \approx 8^{2/4}$	(8)
1	$1.75 \approx 8^{1/4}$	
0	$1 = 8^{0/4}$	

Of course this is such a small example, we could handle it more easily by brute force. But it leads to nontrivial applications, since we can use it also when $m = 4r$ and $n = 3r$, mapping $4r$ -bit records into $2^{3r} \approx N$ locations by dividing the secondary keys into r groups of 4 bits each and applying (7) in each group. The resulting mapping has the desired property: *A query that leaves k of the m bits unspecified will correspond to approximately $N^{k/m}$ locations.* (See exercise 6.)

Some further mappings designed by Rivest appear in exercise 7; they can be used in combination with (7) to produce combinatorial hash functions for cases with $1 \leq m/n \leq 2$. In practice, buckets of size b would be used, and we would take $N \approx 2^nb$; the case $b = 1$ has been used in the above discussion for simplicity in exposition.

***Balanced filing schemes.** Another combinatorial approach to information retrieval, based on "balanced incomplete block designs," has been the subject of considerable investigation. Although the subject is very interesting from a mathematical point of view, it has unfortunately not yet proved to be very useful by comparison with the other methods described above. A brief introduction to the theory will be presented here in order to indicate the flavor of the results, in hopes that some reader might think of a good way to put the ideas to practical use.

A *Steiner triple system* is an arrangement of v objects into unordered triples in such a way that every pair of objects occurs in exactly one triple. For example, when $v = 7$ there is essentially only one Steiner triple system, namely

Triple	Pairs included	
$\{1, 2, 4\}$	$\{1, 2\}, \{1, 4\}, \{2, 4\}$	
$\{2, 3, 5\}$	$\{2, 3\}, \{2, 5\}, \{3, 5\}$	
$\{3, 4, 6\}$	$\{3, 4\}, \{3, 6\}, \{4, 6\}$	
$\{4, 5, 0\}$	$\{0, 4\}, \{0, 5\}, \{4, 5\}$	(9)
$\{5, 6, 1\}$	$\{1, 5\}, \{1, 6\}, \{5, 6\}$	
$\{6, 0, 2\}$	$\{0, 2\}, \{0, 6\}, \{2, 6\}$	
$\{0, 1, 3\}$	$\{0, 1\}, \{0, 3\}, \{1, 3\}$	

Since there are $\frac{1}{2}v(v - 1)$ pairs of objects and three pairs per triple, there must be $\frac{1}{6}v(v - 1)$ triples in all; and since each object must be paired with $v - 1$

others, each object must appear in exactly $\frac{1}{2}(v - 1)$ triples. These conditions imply that a Steiner triple system can't exist unless $\frac{1}{6}v(v - 1)$ and $\frac{1}{2}(v - 1)$ are integers, and this is equivalent to saying that v is odd and not congruent to 2 modulo 3, i.e.,

$$v \bmod 6 = 1 \text{ or } 3. \quad (10)$$

Conversely, T. P. Kirkman proved in 1847 that Steiner triple systems do exist for all $v \geq 1$ such that (10) holds. His interesting construction is given in exercise 10.

Steiner triple systems can be used to reduce the redundancy of combined inverted file indexes. For example, consider again the cookie recipe file of Table 1, and convert the rightmost column into a 31st attribute which is 1 if any special ingredients are necessary, 0 otherwise. Assume that we want to answer all inclusive queries on pairs of attributes, e.g., "What recipes use both coconut and raisins?" We could make up an inverted list for each of these $\binom{31}{2} = 465$ possible queries. But it would turn out that this takes a lot of space since Pfeffernuesse (for example) would appear in $\binom{17}{2} = 136$ of the lists, and a record with all 31 attributes would appear in every list! A Steiner triple system can be used to make a slight improvement in this situation. There is a Steiner triple system on 31 objects, with 155 triples and each pair of objects occurring in exactly one of the triples. We can associate four lists with each triple $\{a, b, c\}$, one list for all records having attributes a, b, \bar{c} (i.e., not c); another for a, \bar{b}, c ; another for \bar{a}, b, c ; and another for records having all three attributes a, b, c . This guarantees that no record will be included in more than 155 of the inverted lists, and it saves space whenever a record has three attributes corresponding to a triple of the system.

Triple systems are special cases of block designs which have blocks of three or more objects. For example, there is a way to arrange 31 objects into sextuples so that every pair of objects appears in exactly one sextuple:

{1, 5, 17, 22, 23, 25}	{7, 11, 23, 28, 29, 0}	{13, 17, 29, 3, 4, 6}	{20, 24, 5, 10, 11, 13}	{26, 30, 11, 16, 17, 19}
{2, 6, 18, 23, 24, 26}	{8, 12, 24, 29, 30, 1}	{14, 18, 30, 4, 5, 7}	{21, 25, 6, 11, 12, 14}	{27, 0, 12, 17, 18, 20}
{3, 7, 19, 24, 25, 27}	{9, 13, 25, 30, 0, 2}	{15, 19, 0, 5, 6, 8}	{22, 26, 7, 12, 13, 15}	{28, 1, 13, 18, 19, 21}
{4, 8, 20, 25, 26, 28}	{10, 14, 26, 0, 1, 3}	{16, 20, 1, 6, 7, 9}	{23, 27, 8, 13, 14, 16}	{29, 2, 14, 19, 20, 22}
{5, 9, 21, 26, 27, 29}	{11, 15, 27, 1, 2, 4}	{17, 21, 2, 7, 8, 10}	{24, 28, 9, 14, 15, 17}	{30, 3, 15, 20, 21, 23}
{6, 10, 22, 27, 28, 30}	{12, 16, 28, 2, 3, 5}	{18, 22, 3, 8, 9, 11}	{25, 29, 10, 15, 16, 18}	{0, 4, 16, 21, 22, 24}
		{19, 23, 4, 9, 10, 12}		

(11)

(This design is formed from the first block by addition mod 31. To verify that it has the stated property, note that the 30 values $(a_i - a_j) \bmod 31$, for $i \neq j$, are distinct, where $(a_1, a_2, \dots, a_6) = (1, 5, 17, 22, 23, 25)$. To find the sextuple containing a given pair (x, y) , choose i and j such that $a_i - a_j \equiv x - y$ (modulo 31); now if $k = (x - a_i) \bmod 31$, we have $(a_i + k) \bmod 31 = x$ and $(a_j + k) \bmod 31 = y$.)

We can use the above design to store the inverted lists in such a way that no record can appear more than 31 times. Each sextuple $\{a, b, c, d, e, f\}$ is associated with 57 lists, for the various possibilities of records having two or

more of the attributes a, b, c, d, e, f , namely $(a, b, \bar{c}, \bar{d}, \bar{e}, \bar{f})$, $(a, \bar{b}, c, \bar{d}, \bar{e}, \bar{f})$, . . . , (a, b, c, d, e, f) ; and the answer to each inclusive 2-attribute query is the disjoint union of 16 appropriate lists in the appropriate sextuple. For this design, Pflefferneuse would be stored in 29 of the 31 blocks, since that record has two of the six attributes in all but blocks $\{19, 23, 4, 9, 10, 12\}$ and $\{13, 17, 29, 3, 4, 6\}$ if we number the columns from 0 to 30.

A similar idea can be used to cut down redundancy of compound inverted lists when we wish to process basic queries instead of inclusive queries. For example, suppose that we have records with five secondary keys K_1, K_2, K_3, K_4, K_5 , each of which has four possible values $\{0, 1, 2, 3\}$. In order to answer queries about records that have $K_i = a$ and $K_j = b$, given a and b and $i \neq j$, we could form inverted lists for all 160 such queries; each record would then appear in ten of these inverted lists. An alternative is to make use of the following configuration of ordered quintuples based on a combinatorial pattern known as "mutually orthogonal latin squares":

$(0, 0, 0, 0, 0)$	$(1, 0, 1, 2, 3)$	$(2, 0, 2, 3, 1)$	$(3, 0, 3, 1, 2)$
$(0, 1, 1, 1, 1)$	$(1, 1, 0, 3, 2)$	$(2, 1, 3, 2, 0)$	$(3, 1, 2, 0, 3)$
$(0, 2, 2, 2, 2)$	$(1, 2, 3, 0, 1)$	$(2, 2, 0, 1, 3)$	$(3, 2, 1, 3, 0)$
$(0, 3, 3, 3, 3)$	$(1, 3, 2, 1, 0)$	$(2, 3, 1, 0, 2)$	$(3, 3, 0, 2, 1)$

(12)

Here if we look at any two of the five components, we will see all 16 possible ordered pairs of values occurring exactly once in those components. We can associate with block (a, b, c, d, e) of this configuration the records which satisfy at least two of the conditions $K_1 = a, K_2 = b, \dots, K_5 = e$. In this way each of the 16 blocks will be associated with 376 of the 1024 possible records, so the average redundancy is lowered from 10 to $16 \times 376/1024 = 57/8$.

The theory of block designs, latin squares, etc. is developed in detail in Marshall Hall, Jr.'s book *Combinatorial Theory* (Waltham, Mass.: Blaisdell, 1967). Although these combinatorial configurations are very beautiful, their only real application to information retrieval so far has been to decrease the redundancy incurred when compound inverted lists are being used; and David K. Chow [*Information and Control* 15 (1969), 377–396] has observed that this type of decrease can be obtained even without using combinatorial designs.

A short history and bibliography. The first published article dealing with a technique for secondary key retrieval was by L. R. Johnson in *CACM* 4 (1961), 218–222. The Multilist system was independently developed by Noah S. Prywes, H. J. Gray, W. I. Landauer, D. Lefkowitz and S. Litwin at about the same time; cf. *IEEE Trans. on Communication and Electronics* 68 (1963), 488–492. Another rather early publication which influenced later work was by D. R. Davis and A. D. Lin, *CACM* 8 (1965), 243–246.

Since then a large literature on the subject has grown up, but most of it deals with the user interface and with programming language considerations, which are not within the scope of this book. In addition to the papers already

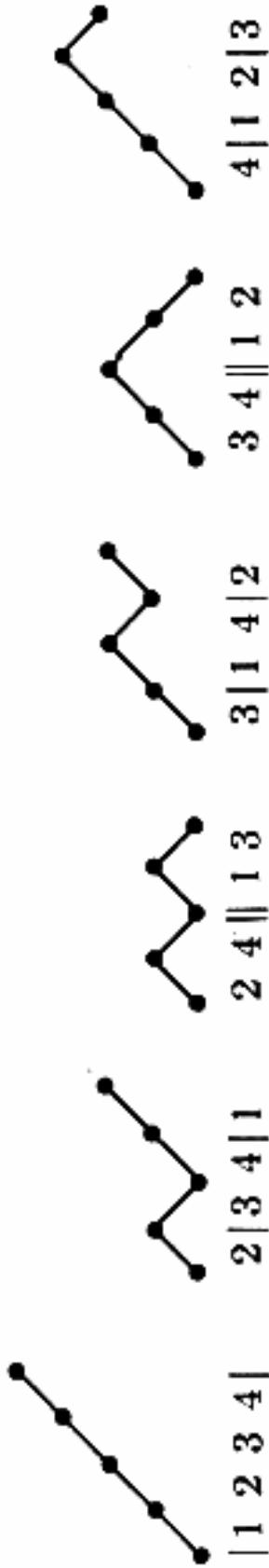
cited, the following published articles were found to be most helpful to the author as this section was being written: Jack Minker and Jerome Sable, *Ann. Rev. of Information Science and Technology* **2** (1967), 123–160; Robert E. Bleier, *Proc. ACM Nat. Conf.* **22** (1967), 41–49; Jerome A. Feldman and Paul D. Rovner, *CACM* **12** (1969), 439–449; Burton H. Bloom, *Proc. ACM Nat. Conf.* **24** (1969), 83–95; H. S. Heaps and L. H. Thiel, *Information Storage and Retrieval* **6** (1970), 137–153; Vincent Y. Lum and Huei Ling, *Proc. ACM Nat. Conf.* **26** (1971), 349–356. A good survey of manual card-filing systems appears in *Methods of Information Handling* by C. P. Bourne (New York: Wiley, 1963), Chapter 5. “Balanced filing schemes” were originally developed by C. T. Abraham, S. P. Ghosh and D. K. Ray-Chaudhuri in 1965; perhaps the best summary of this work and its subsequent extensions appears in an article by R. C. Bose and Gary G. Koch, *SIAM J. Appl. Math.* **17** (1969), 1203–1214.

In this section we have discussed a fairly large number of interesting ideas that are helpful in quite different situations. Since many of these techniques were introduced shortly before this section was written, further advances will probably be forthcoming soon.

It is interesting to note that the human brain is much better at secondary key retrieval than computers are; in fact, it is rather easy for us to recognize faces, or melodies, etc., from only fragmentary information, while computers have barely been able to do this at all. Therefore it is not unlikely that a completely new approach to machine design will someday be discovered, which solves the problem of secondary key retrieval once and for all, making this entire section obsolete.

EXERCISES

- 1. [M27] Let $0 \leq k \leq \frac{1}{2}n$. Prove that the following construction produces $\binom{n}{k}$ permutations of $\{1, 2, \dots, n\}$, such that every t -element subset of $\{1, 2, \dots, n\}$ appears as the first t elements of at least one of the permutations, for $t \leq k$ or $t \geq n - k$: Consider a path in the plane from $(0, 0)$ to (n, r) where $r \geq n - 2k$, in which the i th step is from $(i-1, j)$ to $(i, j+1)$ or to $(i, j-1)$; the latter possibility is allowed only if $j \geq 1$, so that the path never goes below the x -axis. There are exactly $\binom{n}{k}$ such paths. For each path of this kind, a permutation is constructed as follows, using three lists that are initially empty: For $i = 1, 2, \dots, n$, if the i th step of the path goes up, put the number i into list B ; if the step goes down, put i into list A and move the currently largest element of list B into list C . The resulting permutation is equal to the final contents of list A , then list B , then list C , each list in increasing order.
- For example, when $n = 4$ and $k = 2$, the six paths and permutations defined by this procedure are



(Vertical lines show the division between lists A, B, and C. These six permutations correspond to the compound attributes in (2).)

Hint: Represent each t -element subset S by a path that goes from $(0, 0)$ to $(n, n - 2t)$, whose i th step runs from $(i - 1, j)$ to $(i, j + 1)$ if $i \notin S$ and to $(i, j - 1)$ if $i \in S$. Convert every such path into an appropriate path having the special form stated above.

2. [M25] (Sakti P. Ghosh.) Find the minimum possible length l of a list $r_1 r_2 \dots r_l$ of references to records, such that the set of all responses to any of the inclusive queries $**1, *1*, 1**$, $*11, 1*1, 11*$, 111 on three binary-valued secondary keys will appear in consecutive locations $r_i \dots r_j$.

3. [19] In Table 2, what inclusive queries will cause (a) Old-Fashioned Sugar Cookies, (b) Oatmeal-Date Bars, to be obtained among the false drops?

4. [M30] Find exact formulas for the probabilities in (5), assuming that each record has r distinct attributes chosen randomly from among the $\binom{n}{k}$ k -bit codes in an n -bit field and that the query involves q distinct but otherwise random attributes. (Don't be alarmed if the formulas do not simplify.)

5. [40] Experiment with various ways to avoid the redundancy of text when using Harrison's technique for substring searching.

► 6. [M20] The total number of m -bit basic queries with k bits specified is $s = \binom{m}{k} 2^k$. If a combinatorial hashing function like that in (7) converts these queries into l_1, l_2, \dots, l_s locations, respectively, $L(k) = (l_1 + l_2 + \dots + l_s)/s$ is the average number of locations per query. [For example, in (7) we have $L(3) = 1.75$.]

Consider now a composite hash function on an $(m_1 + m_2)$ -bit field, formed by mapping the first m_1 bits with one hash function and the remaining m_2 with another, where $L_1(k)$ and $L_2(k)$ are the corresponding average numbers of locations per query. Find a formula that expresses $L(k)$, for the composite function, in terms of L_1 and L_2 .

7. [M24] (R. L. Rivest.) Find the functions $L(k)$, as defined in the previous exercise, for the following combinatorial hash functions:

$$(a) \quad m = 3, n = 2$$

$$\begin{array}{ll} 0\ 0\ * \rightarrow 1 \\ 1\ *\ 0 \rightarrow 2 \\ * \ 1\ 1 \rightarrow 3 \\ 1\ 0\ 1 \rightarrow 4 \\ 0\ 1\ 0 \rightarrow 4 \end{array}$$

$$(b) \quad m = 4, n = 2$$

$$\begin{array}{ll} 0\ 0\ * \ * \rightarrow 1 \\ * \ 1\ * \ 0 \rightarrow 2 \\ * \ 1\ 1\ 1 \rightarrow 3 \\ 1\ 0\ 1\ * \rightarrow 3 \\ * \ 1\ 0\ 1 \rightarrow 4 \\ 1\cdot 0\ 0\ * \rightarrow 4 \end{array}$$

8. [M47] Develop further useful classes of combinatorial hash functions like (7).

9. [M20] Prove that when $v = 3^n$, the set of all triples of the form

$$\{(a_1 \dots a_{k-1} 0 b_1 \dots b_{n-k})_3, (a_1 \dots a_{k-1} 1 c_1 \dots c_{n-k})_3, \\ (a_1 \dots a_{k-1} 2 d_1 \dots d_{n-k})_3\}, \quad 1 \leq k \leq n,$$

forms a Steiner triple system, where the a 's, b 's, c 's, and d 's range over all combinations of 0's, 1's, and 2's such that $b_i + c_i + d_i \equiv 0 \pmod{3}$.

10. [M32] (Thomas P. Kirkman, *Cambridge and Dublin Math. Journal* 2 (1847), 191–204.) Let us say that a Kirkman triple system of order v is an arrangement of

$v + 1$ objects $\{x_0, x_1, \dots, x_v\}$ into triples such that every pair $\{x_i, x_j\}$ for $i \neq j$ occurs in exactly one triple, except that the v pairs $\{x_i, x_{(i+1) \bmod v}\}$ do not ever occur in the same triple, for $0 \leq i < v$. For example,

$$\{x_0, x_2, x_4\}, \{x_1, x_3, x_4\}$$

is a Kirkman triple system of order 4.

- a) Prove that a Kirkman triple system can exist only when $v \bmod 6 = 0$ or 4.
- b) Given a Steiner triple system S on v objects $\{x_1, \dots, x_v\}$, prove that the following construction yields another one S' on $2v + 1$ objects and a Kirkman triple system K' of order $2v - 2$: The triples of S' are those of S plus
 - i) $\{x_i, y_j, y_k\}$ where $j + k \equiv i$ (modulo v) and $j < k$, $1 \leq i, j, k \leq v$;
 - ii) $\{x_i, y_j, z\}$ where $2j \equiv i$ (modulo v), $1 \leq i, j \leq v$.
 The triples of K' are those of S' minus all those containing y_1 and/or y_v .
- c) Given a Kirkman triple system K on $\{x_0, x_1, \dots, x_v\}$, where $v = 2u$, prove that the following construction yields a Steiner triple system S' on $2v + 1$ objects and a Kirkman triple system K' of order $2v - 2$: The triples of S' are those of K plus
 - i) $\{x_i, x_{(i+1) \bmod v}, y_{i+1}\}$, $0 \leq i < v$;
 - ii) $\{x_i, y_j, y_k\}$, $j + k \equiv 2i + 1$ (modulo $v - 1$), $1 \leq j < k - 1 \leq v - 2$, $1 \leq i \leq v - 2$;
 - iii) $\{x_i, y_j, y_v\}$, $2j \equiv 2i + 1$ (modulo $v - 1$), $1 \leq j \leq v - 1$, $1 \leq i \leq v - 2$;
 - iv) $\{x_0, y_{2j}, y_{2j+1}\}$, $\{x_{v-1}, y_{2j-1}, y_{2j}\}$, $\{x_v, y_j, y_{v-j}\}$, for $1 \leq j < u$;
 - v) $\{x_v, y_u, y_v\}$.
 The triples of K' are those of S' minus all those containing y_1 and/or y_{v-1} .

- d) Use the preceding results to prove that Kirkman triple systems of order v exist for all $v \geq 0$ of the form $6k$ or $6k + 4$, and Steiner triple systems of v objects exist for all $v \geq 1$ of the form $6k + 1$ or $6k + 3$.

11. [M25] The text describes the use of Steiner triple systems in connection with inclusive queries; in order to extend this to all basic queries it is natural to define the following concept. A *complemented triple system* of order v is an arrangement of $2v$ objects $\{x_1, \dots, x_v, \bar{x}_1, \dots, \bar{x}_v\}$ into triples such that every pair of objects occurs together in exactly one triple, except that complementary pairs $\{x_i, \bar{x}_i\}$ never occur together. For example,

$$\{x_1, x_2, x_3\}, \{x_1, \bar{x}_2, \bar{x}_3\}, \{\bar{x}_1, x_2, \bar{x}_3\}, \{\bar{x}_1, \bar{x}_2, x_3\}$$

is a complemented triple system of order three.

Prove that complemented triple systems of order v exist for all $v \geq 0$ not of the form $3k + 2$.

12. [M23] Continuing exercise 11, construct a complemented quadruple system of order 7.

13. [M25] Construct quadruple systems with $v = 4^n$ elements, analogous to the triple systems of exercise 9.

14. [25] Discuss the problem of deleting nodes from post-office trees like Fig. 45.

ANSWERS TO EXERCISES

*"I have answered three questions, and that is enough,"
Said his father. "Don't give yourself airs!
Do you think I can listen all day to such stuff?
Be off, or I'll kick you down-stairs!"*

— LEWIS CARROLL (*Alice's Adventures in Wonderland*, Chapter 5)

NOTES ON THE EXERCISES

1. An average problem for a mathematically inclined reader.
3. See W. J. LeVeque, *Topics in Number Theory 2* (Reading, Mass.: Addison-Wesley, 1956), Chapter 3. (*Note:* One of the men who read a preliminary draft of the manuscript for this book reported that he had discovered a truly remarkable proof, which the margin of his copy was too small to contain.)

CHAPTER 5

1. Let $p(1) \dots p(n)$ and $q(1) \dots q(n)$ be different permutations satisfying the conditions, and let i be minimal with $p(i) \neq q(i)$. Then $p(i) = q(j)$ and $q(i) = p(k)$ for some j, k, i . Since $K_{p(i)} \leq K_{p(k)} = K_{q(i)} \leq K_{q(j)} = K_{p(i)}$ we have $K_{p(i)} = K_{q(i)}$; hence by stability $p(i) < p(k) = q(i) < q(j) = p(i)$, a contradiction.
2. Yes, if the sorting operations were all stable. (If they were not stable we cannot say.) Mr. A and Mr. C certainly have the same result; and so does Mr. B, since the stability shows that equal major keys in his result are accompanied by minor keys in nondecreasing order.

Formally, assume that Mr. B obtains the result $R_{p(1)}, \dots, R_{p(N)} = R'_1, \dots, R'_N$ after sorting the minor keys, then $R'_{q(1)}, \dots, R'_{q(N)} = R_{p(q(1))}, \dots, R_{p(q(N))}$ after sorting the major keys; we want to show that

$$(K_{p(q(i))}, k_{p(q(i))}) \leq (K_{p(q(i+1))}, k_{p(q(i+1))})$$

for $1 \leq i < N$. If $K_{p(q(i))} \neq K_{p(q(i+1))}$, we have $K_{p(q(i))} < K_{p(q(i+1))}$; and if $K_{p(q(i))} = K_{p(q(i+1))}$, then $K'_{q(i)} = K'_{q(i+1)}$, hence $q(i) < q(i+1)$, hence $k'_{q(i)} \leq k'_{q(i+1)}$, i.e., $k_{p(q(i))} \leq k_{p(q(i+1))}$.

3. We can always bring all records with equal keys together, preserving their relative order, treating these groups of records as a unit in further operations; hence we may assume that all keys are distinct. Let $a < b < c < d$; then we can arrange things so that the first three keys are abc , bca , or cab . Now if $N - 1$ distinct keys can be sorted, so can N ; for if $K_1 < \dots < K_{N-1} > K_N$ we always have either $K_{i-1} < K_N < K_i$ for some i , or $K_N < K_1$.

4. Overflow is possible, and it can lead to a false equality indication. He should have written, e.g., LDA A; CMPA B and tested the comparison indicator. (The inability to make full-word comparisons by subtraction is a problem on many computers; it is the chief reason for including CMPA, . . . , CMPX in MIX's repertoire.)

5.

	COMPARE	STJ	9F
	1H	LDX	A,1
		CMPX	B,1
		JNE	9F
		DEC1	1
		J1P	1B
9H		JMP	*
			■

6. Solution 1, based on the identity $\min(a, b) = \frac{1}{2}(a + b - |a - b|)$:

LDA	A	SRAX	1
SRA	5	ADD	AB1
DIV	=2=	ENTX	1
STA	A1 $a = 2a_1 + a_2$	SLAX	5
STX	A2 $ a_2 \leq 1$	MUL	AB2
LDA	B	STX	AB3 $(a_2 - b_2)$ sign $(a - b)$
SRA	5	LDA	A2
DIV	=2=	ADD	B2
STA	B1 $b = 2b_1 + b_2$	SUB	AB3
STX	B2 $ b_2 \leq 1$	SRAX	5
LDA	A1	DIV	=2=
SUB	B1 no overflow possible	ADD	A1
STA	AB1 $a_1 - b_1$	ADD	B1 no overflow possible
LDA	A2	SUB	AB1(1:5)
SUB	B2	STA	C ■
STA	AB2 $a_2 - b_2$		

Solution 2, based on the fact that indexing can cause interchanges in a tricky way:

LDA	A
STA	C
STA	TA
LDA	B
STA	TB

Now duplicate the following code k times, where $2^k > 10^{10}$:

LDA	TA	LDA	TB	INC1	0,2
SRA	5	SRA	5	INC1	0,2
DIV	=2=	DIV	=2=	INC1	0,2
STX	TEMP	STX	TEMP	LD3	TMIN,1
LD1	TEMP	LD2	TEMP	LDA	0,3
STA	TA	STA	TB	STA	C

(This scans the binary representations of a and b from right to left.) Finally, the table

		HLT	
	CON C	-1	-1
	CON B	0	-1
	CON B	+1	-1
	CON A	-1	0
TMIN	CON C	0	0
	CON B	1	0
	CON A	-1	1
	CON A	0	1
	CON C	1	1

7. $\sum_j \binom{r+j-1}{j} (-1)^j \binom{N}{r+j} x^r + j$, by the method of inclusion and exclusion (exercise 1.3.3-26). This is $r \binom{N}{r} \int_0^x t^{r-1} (1-t)^{N-r} dt$, a "beta distribution."

8. Sort it, then count. (Some sorting methods make it convenient to drop records whose keys are duplicated elsewhere in the file.)

9. Assign each person an identification number, which must appear on all forms concerning him. Sort the information forms and the tax forms separately, with this identification number as the key. Denote the sorted tax forms by R_1, \dots, R_N , with keys $K_1 < \dots < K_N$. (There should be no two tax forms with equal keys.) Add a new $(N+1)$ st record whose key is ∞ , and set $i \leftarrow 1$. Then, for each record in the information file, check if it has been reported, as follows: Let K denote the key on the information form being processed.

- a) If $K > K_i$, increase i by 1 and repeat this step.
- b) If $K < K_i$, or if $K = K_i$ and the information is not reflected on tax form R_i , signal an error.

Try to do all this processing without wasting the taxpayers' money.

10. One way is to attach the key (j, i) to the entry $a_{i,j}$ and to sort using lexicographic order, then omit the keys. (A similar idea can be used to obtain any desired reordering of information, when a simple formula for the reordering can be given.)

In the special case considered in this problem, the method of "balanced two-way merge sorting" treats the keys in such a simple manner that it is unnecessary to actually write them explicitly on the tapes. Given an $n \times n$ matrix, we may proceed as follows: First put odd-numbered rows on tape 1, even-numbered rows on tape 2, etc., obtaining

Tape 1: $a_{11} a_{12} \dots a_{1n} a_{31} a_{32} \dots a_{3n} a_{51} a_{52} \dots a_{5n} \dots$

Tape 2: $a_{21} a_{22} \dots a_{2n} a_{41} a_{42} \dots a_{4n} a_{61} a_{62} \dots a_{6n} \dots$

Then rewind these tapes, and process them synchronously, to obtain

Tape 3: $a_{11} a_{21} a_{12} a_{22} \dots a_{1n} a_{2n} a_{51} a_{61} a_{52} a_{62} \dots a_{5n} a_{6n} \dots$

Tape 4: $a_{31} a_{41} a_{32} a_{42} \dots a_{3n} a_{4n} a_{71} a_{81} a_{72} a_{82} \dots a_{7n} a_{8n} \dots$

Rewind these tapes, and process them synchronously, to obtain

Tape 1: $a_{11} a_{21} a_{31} a_{41} a_{12} \dots a_{42} \dots a_{4n} a_{9,1} \dots$

Tape 2: $a_{51} a_{61} a_{71} a_{81} a_{52} \dots a_{82} \dots a_{8n} a_{13,1} \dots$

And so on, until the desired transpose is obtained after $\lceil \log_2 n \rceil$ passes.

11. One way is to attach random distinct key values, sort on these keys, then discard the keys. (Cf. exercise 10; a similar method for obtaining a random *sample* was discussed in Section 3.4.2.) Another technique, involving about the same amount of work but apparently not straining the accuracy of the random number generator as much, is to attach a random integer in the range $0 \leq K_i \leq N - i$ to R_i , then rearrange using the technique of exercise 5.1.1-5.

12. For example, prepare 84-character records $a_1 a_2 \dots a_{84}$ as follows. For each committee card, $c_1 \dots c_{80}$, set $a_1 a_2 a_3 \leftarrow c_7 c_8 c_9 c_{80}$, $a_4 \leftarrow$ blank, $a_5 \dots a_{84} \leftarrow c_1 \dots c_{80}$. For each faculty card $f_1 \dots f_{80}$, and for each nonblank field $f_{21+3k} f_{22+3k} f_{23+3k}$ on this card for $0 \leq k < 20$, prepare a record with $a_1 a_2 a_3 \leftarrow f_{21+3k} f_{22+3k} f_{23+3k}$, $a_4 \dots a_{22} \leftarrow f_2 \dots f_{20}$, $a_{23} \dots a_{21+j} \leftarrow f_1 \dots f_j$, $a_{22+j} \leftarrow \text{“},”$, $a_{24+j} \leftarrow f_{19}$, $a_{25+j} \leftarrow \text{“}.,”$, $a_{27+j} \leftarrow f_{20}$, and (if f_{20} is nonblank) $a_{28+j} \leftarrow \text{“}.”$; other positions blank. Here j is the largest integer ≤ 18 with f_j nonblank. (If blanks are not lowest in the collating sequence, a_4 on the committee records and all blank characters among $a_4 \dots a_{22}$ on the faculty records should be changed to the character lowest in the collating sequence.)

Now sort these 84-character records alphabetically. After sorting, process each 84-character record in turn as follows: If a_4 is blank, start a new page and print $a_4 \dots a_{84}$. If a_4 is nonblank, print $a_{23} \dots a_{84}$ preceded by 19 blanks. (There are obvious refinements to check for committee numbers without names, and faculty members not on enough committees. This exercise illustrates a fairly general method for creating listings, by sorting on a key which is not printed as part of the listing itself. It would be possible to keep the committee records separate from the faculty records, sorting the two files separately and listing them by processing them in a synchronous manner; but the separation probably doesn't save enough time to be worth the extra effort.)

13. With a character-conversion table, you can design a lexicographic comparison routine which simulates the order used on the other machine. Alternatively, you could create artificial keys, different from the actual characters but giving the desired ordering. The latter method has the advantage that it needs to be done only once; but it takes more space and requires conversion of the entire key. The former method can often determine the result of a comparison by converting only one or two letters of the keys; during later stages of sorting, the comparison will be between nearly equal keys, and it will perhaps be advantageous in the former method to check for equality of letters before converting them.

14. For this problem, just run through the file once keeping 50 or so individual counts. But if "city" were substituted for "state," and if the total number of cities were quite large, it would be a good idea to sort on the city name.

15. As in exercise 14, it depends on the size of the problem. If the total number of cross-reference entries fits into high-speed memory, the best approach is probably to use a symbol table algorithm (Chapter 6) with each identifier associated with the head of a list of references. For larger problems, create a file of records, one record for each cross-reference citation to be put in the index, and sort it.

16. Carry along with each card a key which, sorted lexicographically in the usual simple way, will define the desired ordering. This key is to be supplied by library personnel and attached to the card data when it first enters the system. A possible

key, using MIX character code to define the collating sequence, uses the following two-letter codes to separate words from each other:

UU	end of key
U.	end of cross-reference
U,	end of surname
U(hyphen of multiple surname
U)	end of author name
U+	end of place name
U-	end of subject heading
U*	end of book title
U/	space between words.

The given example would then come out as follows (showing only the first 25 characters):

ACCADEMIAL/NAZIONALEU/DEI
ACHTZEHNHUNDERTU/ZWOLFU/E
BIBLIOTHEQUEU/DU/HISTOIRE
BIBLIOTHEQUEU/DESU/CURIOS
BROWNU, JU/CROSBYU)UU
BROWNU, JOHNU)UU
BROWNU, JOHNU)MATHEMATICIA
BROWNU, JOHNU)OFU/BOSTONUU
BROWNU, JOHNU)1715UU
BROWNU, JOHNU)1715U-UU
BROWNU, JOHNU)1761UU
BROWNU, JOHNU)1810UU
BROWNU(WILLIAMSU, REGINALD
BROWNU/AMERICAU*UU
BROWNU/ANDU/DALLISONSU/NE
BROWNJOHNU, ALANU)UU
DENU, VLADIMIRU/EDUARDOVIC
DENU*UU
DENU/LIEBENU/SUSSENU/MADE
DIXU, MORGANU)1827UU
DIXU/HUITU/CENTU/DOUZELU/O
DIXU/NEUVIEMELU/SIECLEU/FR
EIGHTEENU/FORTYU/SEVENU/I
EIGHTEENU/TWELVEU/OVERTUR
IU/AMLU/AU/MATHEMATICIANU*
IU/BU/MLU/JOURNALU/OFU/RES

IU/HALU/EHADU*UU
IAU/AU/LOVELU/STORYU*UU
INTERNATIONALU/BUSINESSU/
KHUWARIZMIU, MUHAMMADU/IBN
LABORU*UU
LABORU/RESEARCHLI/ASSOCIAT
LABOURU.UU
MACCALLSU/COOKBOOKU*U
MACCARTHYU, JOHNU)1927UU
MACHINEU/INDEPENDENTU/COM
MACMAHONU, PERCYU/ALEXANDE
MISTRESSU/DALLOWAYU*UU
MISTRESSU/OFU/MISTRESSESU
ROYALU/SOCIETYU/OFU/LONDO
SAINTU/PETERSBURGERU/ZEIT
SAINTU/SAENSU, CAMILLEU)18
SAINTELU/ANNEU/DESU/MONTSU
SEMINUMERICALU/ALGORITHMS
UNCLEU/TOMSU/CABINU*UU
UNITEDU/STATESU/BUREAUU/O
VANDERMONDELU, ALEXANDERU/T
VANVALKENBURGU, MACU/ELWYN
VONNEUMANNU, JOHNU)1903UU
WHOLEU/ARTU/OFU/LEGERDEMA
WHOSU/AFRAIDLU/OFU/VIRGINI
WIJNGAARDENU, ADRIAANU/VAN

This auxiliary key should be followed by the card data, so that unequal cards having the same auxiliary key (e.g., Sir John = John) are distinguished properly.

17. Prepare two files, one containing $a^{mn} \bmod p$ and the other containing $(ba^{-n}) \bmod p$ for $0 \leq n < m$. Sort these files and look for a common entry.

Note: This reduces the work from $O(p)$ to $O(\sqrt{p} \log p)$. Is a further reduction possible? It is not difficult to determine if n is even or odd, in $\log p$ steps, by testing

whether $b^{(p-1)/2} \bmod p = 1$ or $(p - 1)$. So we may reduce the problem to the case n even; and $a^{2n} \equiv b$ is equivalent to $a^n \equiv \pm\sqrt{b}$. Unfortunately, there is no apparent way to select the appropriate \sqrt{b} which has $n \leq (p - 1)/2$.

18. For example, we can make two files containing values of $(u^6 + v^6 + w^6) \bmod W$ and $(z^6 - x^6 - y^6) \bmod W$ for $u \leq v \leq w, x \leq y \leq z$, where W is the word size of our computer. Sort these and look for duplicates, then subject the duplicates to further tests. (Some congruences modulo small primes might also be used to place further restrictions on u, v, w, x, y, z .)

19. In general, to find all pairs of numbers (x_i, x_j) with $x_i + x_j = c$, where c is given: Sort the file so that $x_1 < x_2 < \dots < x_N$. Set $i \leftarrow 1, j \leftarrow N$, and then repeat the following operation until $j \leq i$:

If $x_i + x_j = c$, output (x_i, x_j) and (x_j, x_i) , set $i \leftarrow i + 1, j \leftarrow j - 1$;
If $x_i + x_j < c$, set $i \leftarrow i + 1$;
If $x_i + x_j > c$, set $j \leftarrow j - 1$.

Finally if $j = i$ and $2x_i = c$, output (x_i, x_i) . This process is like those of exercises 16 and 17: We are essentially making two sorted files, one containing x_1, \dots, x_N and the other containing $c - x_N, \dots, c - x_1$, and checking for duplicates. But the second file doesn't need to be explicitly formed in this case. Another approach when c is odd is to sort on a key such as $(x \text{ odd} \Rightarrow x, x \text{ even} \Rightarrow c - x)$.

20. Some of the alternatives are: (a) For each of the 499,500 pairs i, j with $1 \leq i < j \leq 1000$, set $y_1 \leftarrow x_i \oplus x_j, y_2 \leftarrow y_1 \wedge (y_1 - 1), y_3 \leftarrow y_2 \wedge (y_2 - 1)$; then print (x_i, x_j) if and only if $y_3 = 0$. Here \oplus denotes "exclusive or" and \wedge denotes "and". (b) Create a file with 31000 entries, forming 31 entries from each original word x_i by including x_i and the 30 words that differ from x_i in one position. Sort this file and look for duplicates. (c) Do a test analogous to (a) on

- i) all pairs of words which agree in their first 10 bits;
- ii) all pairs of words which agree in their middle 10 bits (but not the first 10);
- iii) all pairs of words which agree in their last 10 bits (but neither the first nor middle 10).

This involves three sorts of the data, using a specified 10-bit key each time. The expected number of pairs in each of the three cases is less than 500, if the original words are randomly distributed.

21. First prepare a file containing all five-letter English words. (Be sure to consider adding suffixes such as -ED, -ER, -ERS, -S to shorter words.) Now take each five-letter word α and sort its letters into ascending order, obtaining the sorted five-letter sequence α' . Finally sort all pairs (α', α) to bring all anagrams together.

Experiments by Kim D. Gibson in 1967 indicate that the second longest set of anagrams is LEAST, SLATE, STALE, STEAL, TAELS, TALES, TEALS. [The use of a larger dictionary would have enlarged this set by adding ASTEL, a splinter; LEATS, water junctions; TESLA, a unit of magnetic induction. And we might also count Madame de Staël! Cf. H. E. Dudeney, *300 Best Word Puzzles*, ed. by Martin Gardner (N. Y.: Chas. Scribner's Sons, 1968), puzzle 194.]

The first and last sets found were ALBAS, BALAS, BALSA, BASAL and STRUT, STURT, TRUST, respectively. The unexpected sets ADDER, DARED, DREAD and ALGOR, ARGOL, GORAL, LARGO are of possible interest to computer scientists.

A faster way to proceed is to compute $f(\alpha) = (x + a_1)(x + a_2)(x + a_3)(x + a_4)(x + a_5)$ mod m , where a_1, \dots, a_5 are numerical codes for the individual letters in α , and where m is the computer word size. Here x is any fixed value which may be chosen "at random" before starting the computation. Sorting the file $(f(\alpha), \alpha)$ will bring anagrams together; afterwards when $f(\alpha) = f(\beta)$ we must make sure that we have a true anagram with $\alpha' = \beta'$. The value $f(\alpha)$ can be calculated more rapidly than α' , and this method avoids the determination of α' for most of the words α in the file.

Note: A similar technique can be used when we want to bring together all sets of records which have equal n -word keys (a_1, \dots, a_n) . Suppose that we don't care about the order of the file, except that records with equal keys are to be brought together; it is sometimes faster to sort on the one-word key $(a_1x^{n-1} + a_2x^{n-2} + \dots + a_n)$ mod m instead of the original n -word key.

22. Find isomorphic invariants of the graphs (i.e., functions which take equal values on isomorphic directed graphs) and sort on these, to separate "obviously nonisomorphic" graphs from each other. Examples of isomorphic invariants: (a) Represent vertex v_i by (a_i, b_i) , where a_i is its in-degree and b_i is its out-degree; then sort the pairs (a_i, b_i) into lexicographic order. The resulting file is an isomorphic invariant. (b) Represent an arc from v_i to v_j by (a_i, b_i, a_j, b_j) , and sort these quadruples into lexicographic order. (c) Separate the directed graph into connected components (cf. Algorithm 2.3.3E), determine invariants of each component, and sort the components into order of their invariants in some way. See also the discussion in exercise 21.

After sorting the directed graphs on their invariants, it will still in general be necessary to make secondary tests to see whether directed graphs with identical invariants are in fact isomorphic. The invariants are helpful for these tests too. In the case of trees it is possible to find "characteristic" or "canonical" invariants which completely characterize the tree, so that secondary testing is unnecessary [see H. I. Scovs, *Machine Intelligence 3* (1969), 43-60].

23. One way is to form a file containing all three-person cliques, then transform it into a file containing all four-person cliques, etc.; if there are no large cliques this method will be quite satisfactory. (On the other hand, if there is a clique of size n , there are at least $\binom{n}{k}$ cliques of size k , so this method can blow up even when n is only 20 or so.)

Given a file which lists all $(k - 1)$ -person cliques, in the form (a_1, \dots, a_{k-1}) where $a_1 < \dots < a_{k-1}$, we can find the k -person cliques by (i) creating a new file containing the entries $(b, c, a_1, \dots, a_{k-2})$ for each pair of $(k - 1)$ -person cliques of the respective forms (a_1, \dots, a_{k-2}, b) , (a_1, \dots, a_{k-2}, c) with $b < c$; (ii) sorting this file on its first two components; (iii) for each entry $(b, c, a_1, \dots, a_{k-2})$ in this new file which matches a pair (b, c) of acquaintances in the originally given file, output the k -person clique $(a_1, \dots, a_{k-2}, b, c)$.

24. Make another copy of the input file; sort one copy on the first components and the other on the second. Passing over these files in sequence now allows us to create a new file containing all pairs (x_i, x_{i+2}) for $1 \leq i \leq N - 2$, and to identify (x_{N-1}, x_N) . The pairs $(N - 1, x_{N-1})$ and (N, x_N) should be written on still another file.

The process continues inductively. Assume that file F contains all pairs (x_i, x_{i+t}) for $1 \leq i \leq N - t$, in random order, and that file G contains all pairs (i, x_i) for $N - t < i \leq N$ in order of the second components. Let H be a copy of file F , and sort H by first components, F by second. Now go through F , G , and H , creating two

new files F' and G' , as follows. If the current records of files F , G , H are, respectively, (x, x') , (y, y') , (z, z') , then:

- i) If $x' = z$, output (x, z') to F' and advance files F and H .
- ii) If $x' = y'$, output $(y - t, x)$ to G' and advance files F and G .
- iii) If $x' > y'$, advance file G .
- iv) If $x' > z$, advance file H .

When file F is exhausted, sort G' by second components and merge G with it; then replace t by $2t$, F by F' , G by G' .

Thus t takes the values $2, 4, 8, \dots$; for fixed t we do $O(\log n)$ passes over the data to sort it; hence the total number of passes is $O((\log n)^2)$. Eventually $t \geq N$, so F is empty; then we simply sort G on its *first* components.

This ingenious solution is due to Norman Hardy. One phase could be saved at the expense of some complication, by recognizing x_{N+1-i} on the same pass as x_i is presently identified.

SECTION 5.1.1

1. $2 \ 0 \ 5 \ 2 \ 2 \ 3 \ 0 \ 0 \ 0; \ 2 \ 7 \ 3 \ 5 \ 4 \ 1 \ 8 \ 6.$
2. $b_1 = (m - 1) \text{ mod } n; b_{j+1} = (b_j + m - 1) \text{ mod } (n - j).$
3. $\bar{a}_j = a_{n+1-j}$ (the “reflected” permutation). This idea was used by O. Terquem [*Journ. de Math.* (1) 3 (1838), 559–560] to prove that the average number of inversions in a random permutation is $\frac{1}{2}\binom{n}{2}$.
4. C1. Set $x_0 \leftarrow 0$. (It is possible to let x_j share memory with b_j in what follows, for $1 \leq j \leq n$.)
- C2. For $k = n, n - 1, \dots, 1$ (in this order) do the following: Set $j \leftarrow 0$; then set $j \leftarrow x_j$ exactly b_k times; then set $x_k \leftarrow x_j$ and $x_j \leftarrow k$.
- C3. Set $j \leftarrow 0$.
- C4. For $k = 1, 2, \dots, n$ (in this order), do the following: Set $a_k \leftarrow x_j$; then set $j \leftarrow x_j$. ■

To save memory space, see exercise 5.2–12.

5. Let α be a string $[m_1, n_1] \cdots [m_k, n_k]$ of ordered pairs of nonnegative integers; we write $|\alpha| = k$, the length of α . Let ϵ denote the empty (length 0) string. Consider the binary operation defined recursively on pairs of such strings as follows:

$$\epsilon \circ \alpha = \alpha \circ \epsilon = \alpha;$$

$$([m, n]\alpha) \circ ([m', n']\beta) = \begin{cases} [m, n](\alpha \circ ([m' - m, n']\beta)), & \text{if } m \leq m', \\ [m', n']([m - m' - 1, n]\alpha) \circ \beta), & \text{if } m > m'. \end{cases}$$

It follows that the computation time required to evaluate $\alpha \circ \beta$ is proportional to $|\alpha \circ \beta| = |\alpha| + |\beta|$. Furthermore, we can prove that \circ is associative and that $[b_1, 1] \circ [b_2, 2] \circ \dots \circ [b_n, n] = [0, a_1][0, a_2] \dots [0, a_n]$. The expression on the left can be evaluated in $\lceil \log_2 n \rceil$ passes, each pass combining pairs of strings, for a total of $O(n \log n)$ steps.

Example: Starting from (2), we want to evaluate $[2, 1] \circ [3, 2] \circ [6, 3] \circ [4, 4] \circ [0, 5] \circ [2, 6] \circ [2, 7] \circ [1, 8] \circ [0, 9]$. The first pass reduces this to $[2, 1][1, 2] \circ [4, 4][1, 3] \circ [0, 5][2, 6] \circ [1, 8][0, 7] \circ [0, 9]$. The second pass reduces it to $[2, 1][1, 2][1, 4][1, 3] \circ [0, 5][1, 8][0, 6][0, 7] \circ [0, 9]$. The third pass yields

$$[0, 5][1, 1][0, 8][0, 2][0, 6][0, 4][0, 7][0, 3] \circ [0, 9].$$

The fourth pass yields (1).

Motivation: A string such as $[4, 4][1, 3]$ stands for “ $\square\square\square 4 \square 3$ ”, where “ \square ” denotes a large number to be filled in later. Note that, together with exercise 2, we obtain an algorithm for the Josephus problem which is $O(n \log n)$ instead of $O(mn)$, partially answering a question raised in exercise 1.3.2–22.

Another $O(n \log n)$ solution to this problem, using a random-access memory, follows from the use of balanced trees in a straightforward manner.

6. Start with $b_1 = b_2 = \dots = b_n = 0$. For $k = \lfloor \log_2 n \rfloor, \lfloor \log_2 n \rfloor - 1, \dots, 0$ do the following: Set $x_s \leftarrow 0$ for $0 \leq s \leq n/2^{k+1}$; then for $j = 1, 2, \dots, n$ do the following: Set $r \leftarrow \lfloor a_j/2^k \rfloor \bmod 2$, $s \leftarrow \lfloor a_j/2^{k+1} \rfloor$ (these are essentially bit extractions); if $r = 0$, set $b_{a_j} \leftarrow b_{a_j} + x_s$, and if $r = 1$ set $x_s \leftarrow x_s + 1$.

Another solution appears in exercise 5.2.4–21.

7. $B_j < j$ and $C_j \leq n - j$ since a_j has less than j elements to its left and $n - j$ elements to its right. To reconstruct $a_1 a_2 \dots a_n$ from $B_1 B_2 \dots B_n$, start with the element 1; then for $k = 2, \dots, n$ add one to each element $> B_k$ and append $(B_k + 1)$ at the right. (Cf. Method 2 in Section 1.2.5.) A similar procedure works for the C 's. Alternatively, we could use the result of the following exercise.

8. $b' = C$, $c' = B$, $B' = c$, $C' = b$, since each inversion (a_i, a_j) of $a_1 \dots a_n$ corresponds to the inversion (j, i) of $a'_1 \dots a'_n$. Some further relations: (a) $c_j = j - 1$ iff $(b_i > b_j \text{ for all } i < j)$; (b) $b_j = n - j$ iff $(c_i > c_j \text{ for all } i > j)$; (c) $b_j = 0$ iff $(c_i - i < c_j - j \text{ for all } i > j)$; (d) $c_j = 0$ iff $(b_i + i < b_j + j \text{ for all } i < j)$; (e) $b_i \leq b_{i+1}$ iff $c_i \geq c_{i+1}$; (f) $a_j = j + C_j - B_j$; $a'_j = j + b_j - c_j$.

9. $b = C = b'$ is equivalent to $a = a'$.

10. $\sqrt{10}$. (One way to coordinatize the truncated octahedron lets the respective vectors $(1, 0, 0)$, $(0, 1, 0)$, $\frac{1}{2}(1, 1, \sqrt{2})$, $\frac{1}{2}(1, -1, \sqrt{2})$, $\frac{1}{2}(-1, 1, \sqrt{2})$, $\frac{1}{2}(-1, -1, \sqrt{2})$ stand for adjacent interchanges of the respective pairs 21, 43, 41, 31, 42, 32. The sum of these vectors gives $(1, 1, 2\sqrt{2})$ as the difference between vertices 4321 and 1234.)

A more symmetric solution is to represent vertex π in four dimensions by

$$\sum \{\mathbf{e}_u - \mathbf{e}_v \mid (u, v) \text{ is an inversion of } \pi\},$$

where $\mathbf{e}_1 = (1, 0, 0, 0)$, $\mathbf{e}_2 = (0, 1, 0, 0)$, $\mathbf{e}_3 = (0, 0, 1, 0)$, $\mathbf{e}_4 = (0, 0, 0, 1)$. Thus, $1 2 3 4 \leftrightarrow (0, 0, 0, 0)$; $1 2 4 3 \leftrightarrow (0, 0, -1, 1)$; \dots ; $4 3 2 1 \leftrightarrow (-3, -1, 1, 3)$. All points lie on the three-dimensional subspace $\{(w, x, y, z) \mid w + x + y + z = 0\}$; the distance between adjacent vertices is $\sqrt{2}$. Equivalently (cf. exercise 8(f)) we may represent $\pi = a_1 a_2 a_3 a_4$ by the vector (a'_1, a'_2, a'_3, a'_4) , where $a'_1 a'_2 a'_3 a'_4$ is the inverse permutation. (This 4-D representation of the truncated octahedron with permutations as coordinates, and the n -dimensional generalization, was discussed by C. Howard Hinton in *The Fourth Dimension* (London, 1904), Chapter 10.)

Replicas of the truncated octahedron will fill three-dimensional space in the “simplest” possible way [see H. Steinhaus, *Mathematical Snapshots* (Oxford, 1960),

200–203; C. S. Smith, *Scientific American* 190 (January, 1954), 58–64]. For illustrations of the 14 “Archimedean” solids (i.e., nonprism polyhedra whose faces are regular polygons), see W. W. Rouse Ball, *Mathematical Recreations and Essays*, rev. by H. S. M. Coxeter (Macmillan, 1939), 129–140; H. Martyn Cundy and A. P. Rollett, *Mathematical Models* (Oxford, 1952), 94–109.

11. (a) Obvious. (b) Construct a directed graph with vertices $(1, 2, \dots, n)$, arcs from $x \rightarrow y$ if $x > y$ and $(x, y) \in E$ or $x < y$ and $(y, x) \in \bar{E}$. If there are no oriented cycles, this directed graph can be topologically sorted, and the resulting linear order is the desired permutation. If there is an oriented cycle, the shortest has length three, since there are none of length 1 or 2 and since a longer cycle $a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow a_4 \rightarrow \dots \rightarrow a_1$ can be shortened (either $a_1 \rightarrow a_3$ or $a_3 \rightarrow a_1$). But an oriented cycle of length 3 contains two arcs of either E or \bar{E} , and proves that E or \bar{E} is not transitive after all.

12. [C. Berge, *Principes de Combinatoire* (Paris, 1968), 114–117.] Suppose that $(a, b) \in \bar{E}$, $(b, c) \in \bar{E}$, $(a, c) \notin \bar{E}$. Then for some $k \geq 1$ we have $a = x_0 > x_1 > \dots > x_k = c$, where $(x_i, x_{i+1}) \in E(\pi_1) \cup E(\pi_2)$ for $0 \leq i < k$. Consider a counterexample of this type where k is minimal. Since $(a, b) \notin E(\pi_1)$ and $(b, c) \notin E(\pi_1)$, we have $(a, c) \notin E(\pi_1)$, and similarly $(a, c) \notin E(\pi_2)$; hence $k > 1$. But if $x_1 > b$, then $(x_1, b) \in \bar{E}$ contradicts the minimality of k , while $(x_1, b) \in E$ implies that $(a, b) \in E$. Similarly if $x_1 < b$ we find that both $(b, x_1) \in \bar{E}$ and $(b, x_1) \in E$ are impossible.

13. For $0 \leq t < m$, the quantity $\sum_{k \bmod m=t} I_n(k)$ is the coefficient of z^{mN+t} in $G_n(z)/(1-z^m)$ for all large N . The latter function is $1/(1-z)$ times the polynomial

$$(1+z)\dots(1+z+\dots+z^{n-1})/(1+z+\dots+z^{m-1}),$$

and the desired quantity is the sum of the coefficients of the latter polynomial.

14. The hinted construction takes pairs of distinct-part partitions into each other, except in the two cases $j = k = p_k$ and $j = k = p_k - 1$. In the exceptional cases, n is $(2j-1) + \dots + j = (3j^2-j)/2$ and $(2j) + \dots + (j+1) = (3j^2+j)/2$, respectively, and there is a unique unpaired partition with j parts. [Euler's original proof, in *Novi Comment. acad. sc. Pet.* 5 (1754), 75–83, was also very interesting. He showed by simple manipulations that the infinite product equals s_1 , where $s_n = 1 - z^{2n-1} - z^{3n-1}s_{n+1}$ for $n \geq 1$.]

15. Transpose the dot diagram, to go from the p 's to the P 's. The generating function for the P 's is easily obtained, since we first choose any number of 1's (generating function $1/(1-z)$), then independently choose any number of 2's (generating function $(1-z^2)$), ..., finally any number of n 's.

16. The coefficient of $z^n q^m$ in the first identity is the number of partitions of m into at most n parts. In the second identity it is the number of partitions of m into n distinct nonnegative parts, i.e., $m = p_1 + p_2 + \dots + p_n$, where $p_1 > p_2 > \dots > p_n \geq 0$. This is the same as $m - \binom{n}{2} = q_1 + q_2 + \dots + q_n$, where $q_1 \geq q_2 \geq \dots \geq q_n \geq 0$, under the correspondence $q_i = p_i - n + i$. [*Commentarii academiæ scientiarum Petropolitanæ* 13 (1741), 64–93.]

17.	0 0 0 0	0 0 1 0	0 1 0 0	0 0 0 1
	1 0 1 1	1 0 2 1	1 2 0 1	2 1 0 1
	0 1 0 1	0 1 1 0	0 2 1 0	2 0 1 0
	1 1 0 1	1 1 1 0	1 2 1 0	2 1 1 0
	1 0 0 1	1 0 1 0	1 1 0 0	2 1 0 0
	2 1 0 2	2 1 2 0	2 2 1 0	3 2 1 0

18. Let $q = 1 - p$. The sum $\sum \Pr(\alpha)$ over all instances α of inversions may be evaluated by summing on k , where $0 \leq k < n$ is the exact number of leftmost bit positions in which there is equality between i and j as well as between X_i and X_j , in an inversion $X_i \oplus i > X_j \oplus j$ for $i < j$. In this way we obtain the formula $\sum_{0 \leq k < n} 2^k (p^2 + q^2)^k (p^{22^n-k-1} 2^{n-k-1} (2^{n-k-1} - 1))$; summing and simplifying yields $2^{n-1} (p(2-p)(2^n - (p^2 + q^2)^n)/(2 - p^2 - q^2) + (p^2 + q^2)^n - 1)$.
19. [Proc. Amer. Math. Soc. 19 (1968), 236–240.] Assume that $n > 1$, and that a permutation $a_1 \dots a_n$ is given. By induction we may assume that $a_1 \dots a_{n-1}$ corresponds to $b_1 \dots b_{n-1}$, where $\text{index}(a_1 \dots a_{n-1}) = \text{inversions}(b_1 \dots b_{n-1})$ and $a_{n-1} = b_{n-1}$. *Case 1*, $a_{n-1} < a_n$. Let $b_1 \dots b_{n-1} = \alpha_1 x_1 \dots \alpha_r x_r$, where $\alpha_1, \dots, \alpha_r$ are (possibly empty) strings of elements $> a_n$, and where x_1, \dots, x_r are the elements $< a_n$. Then we take $a_1 \dots a_n$ into $x_1 \alpha_1 \dots x_r \alpha_r a_n$. *Case 2*, $a_{n-1} > a_n$. Same as Case 1, with “ $<$ ” and “ $>$ ” interchanged. This transformation is clearly reversible, since we can distinguish Case 1 from Case 2 by comparing the first and last elements. *Example:* 2 7 1 8 3 6 4 is taken into 7 2 8 1 6 3 4; hence 2 7 1 8 3 6 4 5 is left unchanged.
20. See E. M. Wright, *J. London Math. Soc.* 40 (1965), 55–57; and J. Zolnowsky (to appear).

SECTION 5.1.2

1. False (because of a reasonably important technicality). If you said “true,” you probably didn’t know the definition of $M_1 \cup M_2$ given in Section 4.6.3, which has the property that $M_1 \cup M_2$ is a set whenever M_1 and M_2 are sets. Actually, $\alpha \uparrow \beta$ is a permutation of $M_1 \cup M_2$.

2. $b\ c\ a\ d\ d\ a\ d\ b$.

3. Certainly not, since we may have $\alpha = \beta$. (The unique factorization theorem shows that there aren’t too many possibilities, however.)

4. $(d) \uparrow (b\ c\ d) \uparrow (b\ b\ c\ a\ d) \uparrow (b\ a\ b\ c\ d) \uparrow (d)$.

5. The number of occurrences of the pair $\dots xx\dots$ is always either equal to or one less than the number of \sharp columns. When x is the smallest element, the numbers of occurrences are equal iff x is not first in the permutation.

6. Counting the associated number of two-line arrays is easy: $\binom{m}{k} \binom{n}{l}$.

7. Using part (a) of Theorem B, a derivation like that of (20) gives

$$\begin{aligned} & \binom{A-1}{A-k-m-1} \binom{B}{m} \binom{C}{k} \binom{B+k}{B-l} \binom{C-k}{l}; \\ & \binom{A-1}{A-k-m} \binom{B}{m} \binom{C}{k} \binom{B+k-1}{B-l-1} \binom{C-k}{l}; \\ & \binom{A-1}{A-k-m} \binom{B}{m} \binom{C}{k} \binom{B+k-1}{B-l} \binom{C-k}{l}. \end{aligned}$$

8. The complete factorization into primes is $(d) \uparrow (b\ c\ d) \uparrow (b) \uparrow (a\ d\ b\ c) \uparrow (a\ b) \uparrow (b\ c\ d) \uparrow (d)$, which is unique since no adjacent pairs commute. So there are eight solutions, with $\alpha = \epsilon, (d), (d) \uparrow (b\ c\ d), \dots$.

10. False, but true in interesting cases. Given any linear ordering of the primes, there is at least one factorization of the stated form, since whenever the condition is

violated we can make an interchange which reduces the number of "inversions" in the factorization. So the condition fails only because some permutations have more than one such factorization.

Let $\rho \sim \sigma$ mean that ρ commutes with σ . The following condition is necessary and sufficient for the uniqueness of the factorization as stated:

$$\rho \sim \sigma \sim \tau \quad \text{and} \quad \rho \prec \sigma \prec \tau \quad \text{implies} \quad \rho \sim \tau.$$

Proof. If $\rho \sim \sigma \sim \tau$ and $\rho \prec \sigma \prec \tau$ and $\rho \not\sim \tau$, we would have two factorizations $\sigma \uparrow \tau \uparrow \rho = \tau \uparrow \rho \uparrow \sigma$; hence the condition is necessary. Conversely, to show that it is sufficient for uniqueness, let $\rho_1 \uparrow \cdots \uparrow \rho_n = \sigma_1 \uparrow \cdots \uparrow \sigma_n$ be two distinct factorizations satisfying the condition. We may assume that $\sigma_1 \prec \rho_1$, and hence $\sigma_1 = \rho_k$ for some $k > 1$; furthermore $\sigma_1 \sim \rho_j$ for $1 \leq j < k$. Since $\rho_{k-1} \sim \sigma_1 = \rho_k$, we have $\rho_{k-1} \prec \sigma_1$; hence $k > 2$. Let j be such that $\sigma_1 \prec \rho_j$ and $\rho_i \prec \sigma_1$ for $j < i < k$. Then $\rho_{j+1} \sim \sigma_1 \sim \rho_j$ and $\rho_{j+1} \prec \sigma_1 \prec \rho_j$ implies that $\rho_{j+1} \sim \rho_j$; hence $\rho_j \prec \rho_{j+1}$, a contradiction.

Therefore if we are given an ordering relation on a set S of primes, satisfying the above condition, and if we know that all prime factors of a permutation π belong to S , we can conclude that π has a unique factorization of the stated type. Such a condition holds, for example, when S is the set of cycles in (29).

But the set of *all* primes cannot be so ordered; for if we have, say, $(a \ b) \prec (d \ e)$, then we are forced to define

$$(a \ b) \prec (d \ e) \succ (b \ c) \prec (e \ a) \succ (c \ d) \prec (a \ b) \succ (d \ e),$$

a contradiction. (See also the following exercise.)

11. We wish to show that, if $p(1) \dots p(t)$ is a permutation of $(1, \dots, t)$, the permutation $x_{p(1)} \dots x_{p(t)}$ is topologically sorted iff $\sigma_{p(1)} \uparrow \cdots \uparrow \sigma_{p(t)} = \sigma_1 \uparrow \cdots \uparrow \sigma_t$; and that if $x_{p(1)} \dots x_{p(t)}$ and $x_{q(1)} \dots x_{q(t)}$ are distinct topological sortings, we have $\sigma_{p(j)} \neq \sigma_{q(j)}$ for some j . The first property follows by observing that $x_{p(1)}$ can be first in a topological sort iff $\sigma_{p(1)}$ commutes with (yet is distinct from) $\sigma_{p(1)-1}, \dots, \sigma_1$; and this condition implies that $\sigma_{p(2)} \uparrow \cdots \uparrow \sigma_{p(t)} = \sigma_1 \uparrow \cdots \uparrow \sigma_{p(1)-1} \uparrow \sigma_{p(1)+1} \uparrow \cdots \uparrow \sigma_t$, so induction can be used. The second property follows because if j is minimal with $p(j) \neq q(j)$, we have, say, $p(j) < q(j)$ and $x_{p(j)} \not\prec x_{q(j)}$ by definition of topological sorting; hence $\sigma_{p(j)}$ has no letters in common with $\sigma_{q(j)}$.

To get an arbitrary partial ordering, let the cycle σ_k consist of all ordered pairs (i, j) such that $x_i \prec x_j$ and either $i = k$ or $j = k$; these ordered pairs are to appear in some arbitrary order as individual elements of the cycle. Thus the cycles for the partial ordering $x_1 \prec x_3, x_2 \prec x_4, x_1 \prec x_4$ would be $\sigma_1 = ((1, 3)(1, 4)), \sigma_2 = ((2, 4)), \sigma_3 = ((1, 3)), \sigma_4 = ((2, 4)(1, 4))$.

12. No other cycles can be formed, since, for example, the original permutation contains no $\frac{a}{c}$ columns. If $(a \ b \ c \ d)$ occurs s times, then $(a \ b)$ must occur $A - r - s$ times, since there are $A - r$ columns $\frac{a}{b}$, and only two kinds of cycles contribute to such columns.

13. In the two-line notation, first place $A - t$ columns of the form $\frac{d}{a}$, then put the other t a 's in the second line, then place the b 's, and finally the remaining letters.

14. Since the elements below any given letter in the two-line notation for π^{-1} are

in nondecreasing order, we do not always have $(\pi^{-1})^{-1} = \pi$; but it is true that $((\pi^{-1})^{-1})^{-1} = \pi^{-1}$. In fact, the identity

$$(\alpha \uparrow \beta)^{-1} = ((\alpha^{-1} \uparrow \beta^{-1})^{-1})^{-1}$$

holds for all α, β .

Given a multiset whose distinct letters are $x_1 < \dots < x_m$, we can characterize its self-inverse permutations by observing that they each have a unique prime factorization of the form $\beta_1 \uparrow \dots \uparrow \beta_m$, where β_j has zero or more prime factors $(x_j) \uparrow \dots \uparrow (x_j) \uparrow (x_j x_{k_1}) \uparrow \dots \uparrow (x_j x_{k_t})$, $j < k_1 \leq \dots \leq k_t$. For example, $(a) \uparrow (a b) \uparrow (a b) \uparrow (b c) \uparrow (c)$ is a self-inverse permutation. The number of self-inverse permutations of $(m \cdot a, n \cdot b)$ is therefore $\min(m, n) + 1$; of $(\ell \cdot a, m \cdot b, n \cdot c)$ it is the number of solutions of the inequalities $x + y \leq \ell$, $y + z \leq m$, $z + x \leq n$ in nonnegative integers x, y, z . The number of self-inverse permutations of a set is considered in Section 5.1.4.

The number of permutations of $(n_1 \cdot x_1, \dots, n_m \cdot x_m)$ having n_{ij} occurrences of x_i over x_j in their two-line notation is $\prod_i n_i! / \prod_{i,j} n_{ij}!$, the same as the number having n_{ij} occurrences of x_i under x_j in the two-line notation. Hence there ought to be another way to define the inverse of a multiset permutation, by giving an appropriate one-to-one correspondence.

15. See Theorem 2.3.4.2D. Removing one arc of the directed graph must leave an oriented tree.

16. If the multiset is $(n_1 \cdot x_1, n_2 \cdot x_2, \dots)$, with $x_1 < x_2 < \dots$, the inversion table entries for the x_j 's must have the form $b_{j1} \leq \dots \leq b_{jn_j}$ where b_{jn_j} (the number of inversions of the rightmost x_j) is at most $n_{j+1} + n_{j+2} + \dots$. So the generating function for the j th part of the inversion table is the generating function for partitions into at most n_j parts, no part exceeding $n_{j+1} + n_{j+2} + \dots$. The generating function for partitions into at most m parts, no part exceeding n , is the z -binomial coefficient $\binom{m+n}{m}_z$; this is readily proved by induction, and it can also be proved by means of an ingenious construction due to Sylvester [*Amer. J. Math.* 5 (1882), 268–269], or another ingenious construction due to Pólya [*Elemente der Mathematik* 26 (1971), 102–109]. Multiplying the generating functions for $j = 1, 2, \dots$ gives the desired result.

17. Let $h_n(z) = (n!z)/n!(1-z)^n$; the desired generating function is $g(z) = h_n(z)/h_{n_1}(z)h_{n_2}(z)\dots$. The mean of $h_n(z)$ is $\frac{1}{2}\binom{n}{2}$, by Eq. 5.1.1–12, so the mean of g is

$$\frac{1}{2} \left(\binom{n}{2} - \binom{n_1}{2} - \binom{n_2}{2} - \dots \right) = \frac{1}{4} \left(n^2 - n_1^2 - n_2^2 - \dots \right) = \frac{1}{2} \sum_{i < j} n_i n_j.$$

The variance is, similarly,

$$\begin{aligned} \frac{1}{72} (n(n-1)(2n+5) - n_1(n_1-1)(2n_1+5) - \dots) \\ = \frac{1}{36} (n^3 - n_1^3 - n_2^3 - \dots) + \frac{1}{24} (n^2 - n_1^2 - n_2^2 - \dots). \end{aligned}$$

18. Yes. The construction of exercise 5.1.1–17 can be extended in a straightforward way. Alternatively we can generalize the proof given in Section 5.2.1, constructing a one-to-one correspondence between m -tuples (q_1, \dots, q_m) where q_j is a multiset containing n_j nonnegative integers, on the one hand, and ordered pairs of n -tuples $((a_1, \dots, a_n), (p_1, \dots, p_n))$ on the other hand, where $a_1 \dots a_n$ is a permutation of

$(n_1 \cdot 1, \dots, n_m \cdot m)$, and $p_1 \geq \dots \geq p_n \geq 0$. This correspondence is defined as before, inserting the elements of q_j in nonincreasing order; it satisfies the condition

$$\left(\sum_{q_1}\right) + \dots + \left(\sum_{q_m}\right) = J(a_1 \dots a_n) + (p_1 + \dots + p_n)$$

where $\sum q_j$ is the sum of the elements of q_j . [For a further generalization of the technique used in this proof (and in the derivation of Eq. (8) in Section 5.1.3) see D. E. Knuth, *Math. Comp.* **24** (1970), 955–961. See also the comprehensive treatment by Richard P. Stanley, *Memoirs Amer. Math. Soc.* **119** (1972), 104 pp.]

19. (a) Let $S = \{\sigma | \sigma$ is prime, σ is a left factor of $\pi\}$. If S has k elements, the left factors λ of π such that $\mu(\lambda) \neq 0$ are precisely the 2^k intercalations of the subsets of S (see the proof of Theorem C); hence $\sum_{\lambda} \mu(\lambda) = \prod_{\sigma \in S} (1 + \mu(\sigma)) = 0$, since $\mu(\sigma) = -1$ and S is nonempty. (b) Clearly $\epsilon(i_1 \dots i_n) = 0$ if $i_j = i_k$ for some $j \neq k$. Otherwise $\epsilon(i_1 \dots i_n) = (-1)^r$ where $i_1 \dots i_n$ has r inversions; this is $(-1)^s$, where $i_1 \dots i_n$ has s even cycles; and this is $(-1)^{n+t}$ where $i_1 \dots i_n$ has t cycles.

20. (a) Obvious, by definition of intercalation. (b) By definition,

$$\det(b_{ij}) = \sum_{1 \leq i_1, \dots, i_m \leq m} \epsilon(i_1 \dots i_m) b_{1i_1} \dots b_{mi_m}.$$

Setting $b_{ij} = \delta_{ij} - a_{ij}x_j$, this becomes

$$\sum_{n \geq 0} \sum_{1 \leq i_1, \dots, i_n < m} (-1)^n x_{i_1} \dots x_{i_n} \mu(x_{i_1} \dots x_{i_n}) \nu(x_{i_1} \dots x_{i_n}),$$

since $\mu(\pi)$ is usually zero.

(c) Use exercise 19(a) to show that $D \mathbf{T} G = 1$ when we regard the products of x 's as permutations of noncommutative variables, using the natural algebraic convention $(\alpha + \beta) \mathbf{T} \pi = \alpha \mathbf{T} \pi + \beta \mathbf{T} \pi$.

SECTION 5.1.3

- 1.** We must only show that this value makes (11) valid for $x = k$, when $k \geq 1$.
 The formula becomes

$$\begin{aligned} k^n &= \sum_{0 \leq j \leq r \leq k} (-1)^j (r-j)^n \binom{n+1}{j} \binom{n+k-r}{n} \\ &= \sum_{0 \leq s \leq k} s^n \sum_{0 \leq j \leq k-s} (-1)^j \binom{n+1}{j} \binom{n+k-s-j}{n} \end{aligned}$$

For $s < k$, the sum on j can be extended to the range $0 \leq j \leq n+1$, and it is zero (the $(n+1)$ st difference of an n th degree polynomial in j).

- 2.** (a) The number of sequences $a_1 a_2 \dots a_n$ containing each of the elements $(1, 2, \dots, q)$ at least once is $\binom{n}{q} q!$ (cf. exercise 1.2.6-64); the number of such sequences satisfying the analog of (10), for $m = q$, is $\binom{n-k}{n-q}$, since we must choose $n - q$ of the possible = signs. (b) Add the results of (a) for $m = n - q$ and $n - q - 1$.

3. By (20),

$$\begin{aligned} \sum \frac{x^n}{n!} \sum \binom{n}{k} (-1)^k &= \sum g_n(-1)x^n = -2/(e^{-2x} + 1) \\ &= -\frac{1}{x} \left(\frac{(-2x)}{e^{-2x} - 1} - \frac{(-4x)}{e^{-4x} - 1} \right) \\ &= -\frac{1}{x} \sum_{n \geq 0} \frac{B_n x^n}{n!} ((-2)^n - (-4)^n); \end{aligned}$$

hence the result is $(-1)^n B_{n+1} 2^{n+1} (2^{n+1} - 1)/(n+1)$. Alternatively, since $-2/(e^{-2x} + 1) = -(1 + \tanh x)$, we can express the answer as $(-1)^{(n+1)/2} T_n$, when n is odd, where T_n denotes the tangent number defined by the formula

$$\tan z = T_1 z + T_3 z^3/3! + T_5 z^5/5! + \dots$$

When $n > 0$ is even, the sum obviously vanishes, by (7).

Note that, by (18), we get the curious identity

$$\sum_k \binom{n}{k} k! (-\frac{1}{2})^k = 2B_{n+1}(1 - 2^{n+1})/(n+1).$$

4. $(-1)^{n+m} \binom{n}{m+1}$. (Consider the coefficient of z^{m+1} in (18).)

5. First we find that $\binom{p-1}{k} \pmod{p} = 1$, for $1 \leq k < p$, by formula (13). Hence by the defining recurrence (2), $\binom{p}{k} \pmod{p} = 1$ for $1 \leq k \leq p$.

6. Summing first on k is not allowed, because the terms are nonzero for arbitrarily large j and k , and the sum of the absolute values is infinite.

For a simpler example of the fallacy, let $a_{jk} = 0$ when $|j - k| \neq 1$, and let $a_{j(j+1)} = +1$, $a_{(j+1)j} = -1$. Then $\sum_{j \geq 0} (\sum_{k \geq 0} a_{jk}) = \sum_{j \geq 0} (\delta_{j0}) = +1$, while $\sum_{k \geq 0} (\sum_{j \geq 0} a_{jk}) = \sum_{k \geq 0} (-\delta_{k0}) = -1$.

7. Yes. [David and Barton, *Combinatorial Chance* (1962), 150–154.]

8. [*Combinatory Analysis* 1 (1915), 190.] By inclusion and exclusion. For example, $1/(l_1 + l_2)! l_3! (l_4 + l_5 + l_6)!$ is the probability that $x_1 < \dots < x_{l_1+l_2}$, $x_{l_1+l_2+1} < \dots < x_{l_1+l_2+l_3}$, and $x_{l_1+l_2+l_3+1} < \dots < x_{\Sigma l}$.

9. $p_{km} = q_{km} - q_{k(m+1)}$ in (23). Since $\sum_{k,m} q_{km} z^m x^k = g(x, z)/(1 - x)$, we have $h(z, x) = \sum h_k(z) x^k = g(x, z)/(1 - x) - (g(x, z) - g(x, 0))/z(1 - x) = (z - 1)x/(ze^{(x-1)z} - x) + x/z(1 - x)$. Thus $h_1(z) = e^z - (e^z - 1)/z$; $h_2(z) = (e^{2z} - ze^z) + e^z - (e^{2z} - 1)/z$.

10. Let $M_n = L_1 + \dots + L_n$ be the mean; then $\sum M_n x^n = h'(1, x)$, where the derivative is taken with respect to z , and this is $x/(e^{x-1} - x) - x/(1 - x) = M(x)$, say. By the residue theorem,

$$\frac{1}{2\pi i} \oint M(z) z^{-n-1} dz = M_n - 2(n + \frac{1}{3}) + 1 + \frac{z_1^{-n}}{z_1 - 1} + \frac{\bar{z}_1^{-n}}{\bar{z}_1 - 1},$$

if we integrate around a circle of radius r where $|z_1| < r < |z_2|$. (Note the double pole at $z = 1$.) Furthermore, the absolute value of this integral is $< \oint |M(z)| r^{-n-1} dz = O(r^{-n})$. Integrating over larger and larger circles gives the convergent series $M_n = 2n - \frac{1}{3} + \sum_{k \geq 1} 2 \Re (1/z_k^n (1 - z_k))$.

To determine the variance, we have $h''(1, x) = -2h'(1, x) + 2x(x-1)e^{x-1}/(e^{x-1} - x)^2$. An argument similar to that used for the mean, this time with a triple pole, shows that the coefficients of $h''(1, x)$ are asymptotically $4n^2 + \frac{4}{3}n - 2M_n$ plus smaller terms; this leads to the asymptotic formula $\frac{2}{3}n + \frac{2}{9}$ (plus exponentially smaller terms) for the variance.

11. $P_{kn} = \sum_{t_1 \geq 1, \dots, t_k \geq 1} D(t_1, \dots, t_{k-1}, n, 1)$, where $D(l_1, l_2, \dots, l_k)$ is MacMahon's determinant of exercise 8. Evaluating this determinant by its first row, we find $P_{kn} = c_0 P_{(k-1)n} + c_1 P_{(k-2)n} + \dots + c_{k-2} P_{1n} - E_k(n)$, where c_j and E_k are defined as follows:

$$c_j = (-1)^j \sum_{t_1, \dots, t_{j+1} \geq 1} \frac{1}{(t_1 + \dots + t_{j+1})!} = (-1)^j \sum_{m \geq 0} \binom{m}{j} \frac{1}{(m+1)!}$$

$$= (-1)^j \sum_{r, m \geq 0} \binom{-1}{j-r} \binom{m+1}{r} \frac{1}{(m+1)!} = -1 + e \left(\frac{1}{0!} - \frac{1}{1!} + \dots + (-1)^k \frac{1}{k!} \right).$$

$$E_1(n) = -1/n! + 1/(n+1)!; \quad E_2(n) = 1/(n+1)!;$$

$$E_k(n) = (-1)^k \sum_{m \geq 0} \binom{m}{k-3} \frac{1}{(n+2+m)!}, \quad k \geq 3.$$

Let $P_{0n} = 0$, $C(z) = \sum c_j z^j = (e^{1-z} - 1)/(1-z)$, and let

$$E(z, x) = \sum_{n, k} E_{k+1}(n) z^n x^k$$

$$= 1 - e^z + \frac{(e^{1-z} - 1)x^2}{(1-x)^2} - \frac{x^2(e^{1-z} - e^z)}{(1-x)(1-x-z)} + \frac{e^z - 1 - z}{z(1-x)}.$$

The recurrence relation we have derived is equivalent to the formula $C(x)H(z, x) = H(z, x)/x + E(z, x)$; hence $H(z, x) = E(z, x)x(1-x)/(xe^{1-z} - 1)$. Expanding this power series gives $H_1(z) = h_1(z)$ (see exercise 9); $H_2(z) = eh_1(z) + 1 - e^z$.

[Note: The generating functions for the first three runs were derived by Knuth, *CACM* 6 (1963), 685–688. Barton and Mallows, *Ann. Math. Statistics* 36 (1965), 249, stated the formula $1 - H_{n+1}(z) = (1 - H_n(z))/(1-z) - L_n h_1(z)$, together with (25). Another way to attack this problem is illustrated in exercise 23. Because adjacent runs are not independent, there is no simple relation between the problem solved here and the simpler (probably more useful) result of exercise 9.]

12. [Combinatory Analysis 1 (1915), 209–211.] The number of ways to put the multi-set into t distinguishable boxes is

$$N_t = \binom{t+n_1-1}{n_1} \binom{t+n_2-1}{n_2} \dots \binom{t+n_m-1}{n_m},$$

since there are $\binom{t+n_1-1}{n_1}$ ways to place the 1's, etc. If we require that no box be empty, the method of inclusion and exclusion tells us that the number of ways is

$$M_t = N_t - \binom{t}{1} N_{t-1} + \binom{t}{2} N_{t-2} - \dots$$

Let P_k be the number of permutations having k runs; if we put $k - 1$ vertical lines between the runs, and $t - k$ additional vertical lines in any of the $n - k$ remaining places, we get one of the M_t ways to divide the multiset into t nonempty distinguishable parts. Hence

$$M_t = P_t + \binom{n-t+1}{1} P_{t-1} + \binom{n-t+2}{2} P_{t-2} + \dots.$$

Equating the two values of M_t allows us to determine P_1, P_2, \dots successively in terms of N_1, N_2, \dots . (A more direct proof would be desirable.)

13. $1 + \frac{1}{2} 13 \times 3 = 20.5$.

14. By Foata's correspondence the given permutation corresponds to

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 3 & 1 & 1 & 2 & 3 & 4 & 3 & 2 & 1 & 1 & 3 & 4 & 2 & 2 & 4 & 4 \end{pmatrix};$$

by (33) this corresponds to

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 2 & 2 & 2 & 2 & 3 & 3 & 3 & 3 & 4 & 4 & 4 & 4 \\ 2 & 4 & 4 & 3 & 3 & 3 & 1 & 1 & 4 & 4 & 2 & 1 & 2 & 1 & 2 & 3 \end{pmatrix};$$

and this corresponds to $2 \ 3 \ 4 \ 2 \ 3 \ 4 \ 1 \ 4 \ 2 \ 1 \ 4 \ 3 \ 2 \ 1 \ 3 \ 1$ with 9 runs.

15. The number of alternating runs is the number of j such that $1 < j < n$ and either $a_{j-1} < a_j > a_{j+1}$ or $a_{j-1} > a_j < a_{j+1}$. For fixed j , the probability is $\frac{2}{3}$; hence the average, for $n \geq 2$, is $1 + \frac{2}{3}(n - 2)$.

16. Each permutation on $\{1, 2, \dots, n - 1\}$, having k alternating runs, yields k permutations with k such runs, 2 with $k + 1$, and $n - k - 2$ with $k + 2$, when the new element n is inserted in all possible places. Hence

$$\langle\langle \frac{n}{k} \rangle\rangle = k \langle\langle \frac{n-1}{k} \rangle\rangle + 2 \langle\langle \frac{n-1}{k-1} \rangle\rangle + (n-k) \langle\langle \frac{n-1}{k-2} \rangle\rangle.$$

It is convenient to let $\langle\langle \frac{1}{k} \rangle\rangle = \delta_{k0}$, $G_1(z) = 1$. Then

$$G_n(z) = \frac{z}{n} ((1 - z^2) G'_{n-1}(z) + (2 + (n-2)z) G_{n-1}(z)).$$

Differentiation leads to the recurrence

$$x_n = \frac{1}{n} (x_{n-1}(n-2) + 2n - 2)$$

for $x_n = G'_n(1)$, and this has the solution $x_n = \frac{2}{3}n - \frac{1}{3}$ for $n \geq 2$. Another differentiation leads to the recurrence

$$y_n = \frac{1}{n} (y_{n-1}(n-4) + \frac{8}{3}n^2 - \frac{26}{3}n + 6)$$

for $y_n = G''_n(1)$. Set $y_n = \alpha n^2 + \beta n + \gamma$ and solve for α, β, γ to get $y_n = \frac{4}{9}n^2 - \frac{14}{15}n + \frac{11}{90}$ for $n \geq 4$. Hence $\text{var}(g_n) = \frac{1}{90}(16n - 29)$, $n \geq 4$.

These formulas for the mean and variance are due to J. Bienaymé, who stated them without proof [*Bull. Soc. Math. de France* 2 (1874), 153–154; *Comptes Rendus* 81 (Acad. Sciences, Paris, 1875), 417–423, see also Bertrand's remarks on p. 458]. The recurrence relation for $\langle \langle \frac{n}{k} \rangle \rangle$ is due to D. André [*Comptes Rendus* 97 (Acad. Sciences, Paris, 1883), 1356–1358; *Annales scientifiques de l'Ecole normale supérieure* (3) 1 (Paris, 1884), 121–134]. André noted that $g_n(-1) = 0$ for $n \geq 4$; i.e., the number of permutations with an even number of alternating runs is $n!/2$. He also proved the formula for the mean, and determined the number of permutations which have the maximum number of alternating runs (see exercise 5.1.4–22). It can be shown that

$$G_n(z) = \left(\frac{1+z}{2}\right)^{n-1} (1+w)^{n+1} g_n\left(\frac{1-w}{1+w}\right), \quad w = \sqrt{\frac{1-z}{1+z}}, \quad n \geq 2,$$

where $g_n(z)$ is the generating function (18) for ascending runs. [See David and Barton, *Combinatorial Chance* (London: Griffin, 1962), 157–162.]

17. $\binom{n+1}{2k-1}$; $\binom{n}{2k-2}$ end with 0, $\binom{n}{2k-1}$ end with 1.

18. Let the given sequence be $C_1 C_2 \dots C_n$, an inversion table such as we have considered in exercise 5.1.1–7. If there are k runs in this sequence, there are k in the corresponding permutation; hence the answer is $\binom{n}{k}$.

19. (a) $\langle \langle \frac{n}{k+1} \rangle \rangle$, by the correspondence of Theorem 5.1.2B. (b) There are $(n-k)!$ ways to put $n-k$ further nonattacking rooks on the entire board; hence the answer is $1/(n-k)!$ times $\sum_{j \geq 0} a_{nj} \binom{j}{k}$, where $a_{nj} = \langle \langle \frac{n}{j+1} \rangle \rangle$ is the solution for part (a) with $j = k$. This comes to $\{ \frac{n}{n-k} \}$, by exercise 2.

A direct proof of this result, due to E. A. Bender, associates each partition of $\{1, 2, \dots, n\}$ into k nonempty disjoint subsets with an arrangement of $(n-k)$ rooks: Let the partition be

$$\{1, 2, \dots, n\} = \{a_{11}, a_{12}, \dots, a_{1n_1}\} \cup \dots \cup \{a_{k1}, \dots, a_{kn_k}\},$$

where $a_{ij} < a_{i(j+1)}$ for $1 \leq j < n_i$, $1 \leq i \leq k$. The corresponding arrangement puts rooks in column a_{ij} of row $a_{i(j+1)}$, for $1 \leq j < n_i$, $1 \leq i \leq k$. For example, the configuration illustrated in Fig. 4 corresponds to the partition $\{1, 3, 8\} \cup \{2\} \cup \{4, 6\} \cup \{5\} \cup \{7\}$.

20. The number of readings is the number of runs in the inverse permutation. The first run corresponds to the first reading, etc.

21. It has $n+1-k$ runs and requires $n+1-j$ readings.

22. [J. Combinatorial Theory 1 (1966), 350–374.] If $rs < n$, some reading will pick up $t > r$ elements, $a_{i_1} = j+1, \dots, a_{i_t} = j+t$, where $i_1 < \dots < i_t$. We cannot have $a_m > a_{m+1}$ for all m in the range $i_k \leq m < i_{k+1}$, so the permutation contains at least $t-1$ places with $a_m < a_{m+1}$; it therefore has at most $n-t+1$ runs.

But if $rs \geq n$, consider the permutation

$$(tr) \dots (2r) (r) \dots (3) ((t-1)r+2) \dots (r+2) (2) ((t-1)r+1) \dots (r+1) (1)$$

where $\lceil n/r \rceil = t \leq s$, and where elements $> n$ are to be omitted. It has $n+1-r$ runs and it requires $n+1-r$ readings. By rearranging the groups $\{kr+r, \dots, kr+1\}$ independently among themselves, it is possible to adjust the number of readings to any desired lesser value $s \geq t$.

- 23.** [SIAM Review 3 (1967), 121–122.] Assume that the infinite permutation consists of independent samples from the uniform distribution. Let $f_k(x) dx$ be the probability that the k th long run begins with x ; and let $g(u, x) dx$ be the probability that a long run begins with x , when the preceding long run begins with u . Then $f_1(x) = 1$, $f_{k+1}(x) = \int_0^1 f_k(u) g(u, x) du$. We have $g(u, x) = \sum_{m \geq 1} \theta_m(u, x)$, where $\theta_m(u, x) = \Pr(u < X_1 < \dots < X_m > x \text{ or } u > X_1 > \dots > X_m < x) = \Pr(u < X_1 < \dots < X_m) + \Pr(u > X_1 > \dots > X_m) - \Pr(u < X_1 < \dots < X_m < x) - \Pr(u > X_1 > \dots > X_m > x) = (u^m + (1-u)^m + |u-x|^m)/m!$; hence $g(u, x) = e^u + e^{1-u} - 1 - e^{|u-x|}$. Consequently, $f_2(x) = 2e - 1 - e^x - e^{1-x}$. It can be shown that $f_k(x)$ approaches the limiting value $(2 \cos(x - \frac{\pi}{2}) - \sin \frac{\pi}{2} - \cos \frac{\pi}{2})/(3 \sin \frac{\pi}{2} - \cos \frac{\pi}{2})$. The average length of a run starting with x is $e^x + e^{1-x} - 1$ hence the length LL_k of the k th long run is $\int_0^1 f_k(x) (e^x + e^{1-x} - 1) dx$; $LL_1 = 2e - 3 \approx 2.4365$; $LL_2 = 3e^2 - 8e + 2 \approx 2.4209$. See Section 5.4.1 for similar results.
- 24.** Arguing as before, the result is

$$1 + \sum_{0 \leq k < n} 2^k (p^2 + q^2)^k (p^2 + 2pq(2^{n-k-1} - 1) + q^2((2pq)^{n-k-1} - 1)/(2pq - 1)));$$

carrying out the sum and simplifying yields

$$\begin{aligned} 2^n (p^2 + q^2)^n (p(p - q)/(p^2 + q^2 - pq) - \frac{1}{2}) + (2pq)^n pq^3 / (p^2 + q^2)(p^2 + q^2 - pq) \\ + q^2 / (p^2 + q^2) + 2^{n-1}. \end{aligned}$$

SECTION 5.1.4

1.	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>2</td><td>3</td><td>8</td></tr> <tr><td>4</td><td>5</td><td>7</td><td></td></tr> <tr><td>6</td><td>9</td><td></td><td></td></tr> </table>	1	2	3	8	4	5	7		6	9			<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>3</td><td>5</td><td>8</td></tr> <tr><td>2</td><td>4</td><td>9</td><td></td></tr> <tr><td>6</td><td>7</td><td></td><td></td></tr> </table>	1	3	5	8	2	4	9		6	7			;
1	2	3	8																								
4	5	7																									
6	9																										
1	3	5	8																								
2	4	9																									
6	7																										

1.	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>3</td><td>5</td><td>8</td></tr> <tr><td>2</td><td>4</td><td>9</td><td></td></tr> <tr><td>6</td><td>7</td><td></td><td></td></tr> </table>	1	3	5	8	2	4	9		6	7		
1	3	5	8										
2	4	9											
6	7												

$$1. \quad \begin{pmatrix} 1 & 3 & 5 & 7 & 8 \\ 2 & 4 & 9 & & \\ 6 & 7 & & & \end{pmatrix}.$$

2. When p_i is inserted into column t , let the element in column $t - 1$ be p_j . Then (q_j, p_j) is in class $t - 1$, $q_j < q_i$, and $p_j < p_i$; so, by induction, indices i_1, \dots, i_t exist with the property. Conversely, if $q_j < q_i$ and $p_j < p_i$ and if (q_j, p_j) is in class $(t - 1)$, then column $t - 1$ contains an element $< p_i$ when p_i is inserted, so (q_i, p_i) is in class t .
3. The columns are the “bumping sequences” (9) when p_i is inserted. Lines 1 and 2 reflect the operations on row 1, cf. (14). If we remove columns in which line 2 has ∞ entries, lines 0 and 2 constitute the “bumped” array, as in (15). The stated method for going from line k to line $k + 1$ is just the class-determination algorithm of the text.
4. Let there be t classes in all; exactly k of these have an odd number of elements, since the elements of a class have the form

$$(p_{i_k}, p_{i_1}), \quad (p_{i_{k-1}}, p_{i_2}), \quad \dots, \quad (p_{i_1}, p_{i_k}).$$

(See (18) and (22).) The “bumped” two-line array has exactly $t - k$ fixed points, because of the way it is constructed; hence by induction the tableau minus its first

row has $t - k$ columns of odd length. So the t elements in the first row lead to k odd-length columns in the whole tableau.

5. (a) Use a case analysis, by induction on the size of the tableau, considering first the effect on row 1 and then the effect on the sequence of elements bumped from row 1.
 (b) Admissible interchanges can simulate the operations of Algorithm I, with the tableau represented as a canonical permutation before and after the algorithm. For example, we can transform

$$17 \ 11 \ 4 \ 13 \ 14 \ 2 \ 6 \ 10 \ 15 \ 1 \ 3 \ 5 \ 9 \ 12 \ 16 \ 8$$

into

$$17 \ 11 \ 13 \ 4 \ 10 \ 14 \ 2 \ 6 \ 9 \ 15 \ 1 \ 3 \ 5 \ 8 \ 12 \ 16$$

by a sequence of admissible interchanges (cf. (4) and (5)).

6. Admissible interchanges are symmetrical between left and right, and the canonical permutation for P obviously goes into P^T when the insertion order is reversed.

7. The number of columns, namely the length of row 1, is the number of classes (exercise 2). The number of rows is the number of columns of P^T , so exercise 6 (or Theorem D) completes the proof.

8. With more than n^2 elements, the corresponding P tableau must either have more than n rows or more than n columns. But there are $n \times n$ tableaux. [This result was originally proved in *Compositio Math.* 2 (1935), 463–470.]

9. Such permutations are in 1–1 correspondence with pairs of tableaux of shape (n, n, \dots, n) ; so by (34) the answer is

$$\left(\frac{n^2! \Delta(2n-1, 2n-2, \dots, n)}{(2n-1)!(2n-2)! \dots n!} \right)^2 = \left(\frac{n^2!}{(2n-1)(2n-2)^2 \dots n^n(n-1)^{n-1} \dots 1^1} \right)^2$$

The existence of such a simple formula for this problem is truly amazing. We can also count the number of permutations of $\{1, 2, \dots, mn\}$ with no increasing subsequences longer than m , no decreasing subsequences longer than n .

10. We prove inductively that, at step S3, $P_{(r-1)s}$ and $P_{r(s-1)}$ are both less than $P_{(r+1)s}$ and $P_{r(s+1)}$.

11. We also need to know, of course, the element which was originally P_{11} . Then it is possible to restore things using an algorithm remarkably similar to Algorithm S.

$$12. \binom{n_1+1}{2} + \binom{n_2+2}{2} + \dots + \binom{n_m+m}{2} - \binom{m+1}{3}.$$

The minimum is the sum of the first n terms of the sequence 1, 2, 2, 3, 3, 3, 4, 4, 4, ... of exercise 1.2.4–41; this sum is approximately $\sqrt{8/9} n^{3/2}$. (Since the majority of tableaux on n elements come reasonably near this lower bound, according to the tables cited in exercise 28, the average number of times is almost surely $O(n^{3/2})$, but this has not been proved.)

13. Assume that the elements permuted are $\{1, 2, \dots, n\}$ so that $a_i = 1$; and assume that $a_j = 2$. *Case 1*, $j < i$. Then 1 bumps 2, so row 1 of the tableau corresponding to $a_1 \dots a_{i-1} a_{i+1} \dots a_n$ is row 1 of P^S ; and the “bumped” permutation is the former bumped permutation except for its smallest element, 2, so we may use

induction on n . Case 2, $j > i$. Apply Case 1 to P^T , in view of exercise 6 and the fact $(P^T)^S = (P^S)^T$.

15. As in (37), the example permutation corresponds to the tableau

1	2	5	9	11
3	6	7		
4	8	10		

;

hence the number is $f(l, m, n) = (l + m + n)!/(l - m + 1)(l - n + 2)(m - n + 1)/(l + 2)!(m + 1)!(n)!$, provided, of course, that $l \geq m \geq n$.

16. By Theorem H, 80080.

17. Since g is antisymmetric in the x 's, it is zero when $x_i = x_j$, so it is divisible by $x_i - x_j$ for all $i < j$. Hence $g(x_1, \dots, x_n; y) = h(x_1, \dots, x_n; y)\Delta(x_1, \dots, x_n)$. Here h must be homogeneous in x_1, \dots, x_n, y , of total degree 1, and symmetric in x_1, \dots, x_n ; so $h(x_1, \dots, x_n; y) = a(x_1 + \dots + x_n) + by$ for some a, b depending only on n . We can evaluate a by setting $y = 0$; we can evaluate b by taking the partial derivative with respect to y and then setting $y = 0$. We have

$$\begin{aligned} \tilde{\frac{\partial}{\partial y}} \Delta(x_1, \dots, x_i + y, \dots, x_n)|_{y=0} &= \frac{\partial}{\partial x_i} \Delta(x_1, \dots, x_n) \\ &= \Delta(x_1, \dots, x_n) \sum_{j \neq i} \frac{1}{x_i - x_j}. \end{aligned}$$

Finally,

$$\sum_i \sum_{j \neq i} (x_i/(x_i - x_j)) = \sum_i \sum_{j < i} (x_i/(x_i - x_j) + x_j/(x_j - x_i)) = \binom{n}{2}.$$

18. It must be $\Delta(x_1, \dots, x_n) \cdot (b_0 + b_1y + \dots + b_my^m)$, where each b_k is a homogeneous symmetric polynomial of degree $m - k$ in the x 's. We have

$$\frac{\partial^k}{k! \partial y^k} \Delta(x_1, \dots, x_i + y, \dots, x_n)|_{y=0} = \Delta(x_1, \dots, x_n) \sum 1/\prod_l (x_i - x_{j_l})$$

summed over all $\binom{n-1}{k}$ choices of distinct indices $j_1, \dots, j_k \neq i$. Now in $b_k = \sum x_i^m / \prod_l (x_i - x_{j_l})$ we may combine those groups of $k+1$ terms having a given set of indices $\{i, j_1, \dots, j_k\}$; for example, when $k = 2$, we group sets of three terms of the form $a^m/(a-b)(a-c) + b^m/(b-a)(b-c) + c^m/(c-a)(c-b)$. The sum of each such group is evaluated as in exercise 1.2.3-33; it is the coefficient of z^{m-k} in $1/(1-x_iz)(1-x_{j_1}z) \cdots (1-x_{j_k}z)$. We find therefore that

$$b_k = \sum_j \binom{n-j}{k+1-j} \sum s(p_1, \dots, p_j),$$

where $s(p_1, \dots, p_j)$ is the monomial symmetric function consisting of all distinct terms having the form $x_{i_1}^{p_1} \cdots x_{i_j}^{p_j}$, for distinct indices $i_1, \dots, i_j \in \{1, \dots, n\}$; and the second sum is over all partitions of $m - k$ into exactly j parts, namely $p_1 \geq \cdots \geq p_j \geq 1$, $p_1 + \cdots + p_j = m - k$. (This result was obtained jointly with E. A. Bender.)

When $m = 2$ the answer is $(s(2) + (n-1)s(1)y + \binom{n}{3}y^2)\Delta(x_1, \dots, x_n)$; for $m = 3$ we get $(s(3) + ((n-1)s(2) + s(1, 1))y + \binom{n-1}{2}s(1)y^2 + \binom{n}{4}y^3)\Delta(x_1, \dots, x_n)$; etc.

Another expression gives b_k as the coefficient of z^m in

$$\left(\binom{n}{k+1} z^k - \binom{n-1}{k+1} a_1 z^{k+1} + \binom{n-2}{k+1} a_2 z^{k+2} - \dots \right) / (1 - a_1 z + a_2 z^2 - \dots),$$

where $a_i = \sum_{1 \leq i_1 < \dots < i_l \leq n} x_{i_1} \dots x_{i_l}$ is an “elementary symmetric function.” Multiplying by y^k and summing on k gives the answer as the coefficient of z^m in

$$\frac{1}{yz} \left(\frac{(1+z(y-x_1)) \dots (1+z(y-x_n))}{(1-zx_1) \dots (1-zx_n)} - 1 \right) \Delta(x_1, \dots, x_n).$$

19. Let the shape of the transposed tableau be $(n'_1, n'_2, \dots, n'_r)$; the answer is $\frac{1}{2}f(n_1, n_2, \dots, n_m)$ times $(\sum(n_i^2) - \sum(n_j'^2))/n(n-1) + 1$, where $n = \sum n_i = \sum n'_j$. (This formula was obtained after a rather elaborate computation, using the sum

$$\begin{aligned} \sum_{i < j} x_i x_j \Delta(x_1, \dots, x_i + y, \dots, x_j + y, \dots, x_n) \\ = \left(s(1, 1) + s(1) \binom{n-1}{2} y + \left\{ \begin{array}{c} n \\ n-2 \end{array} \right\} y^2 \right) \Delta(x_1, \dots, x_n) \end{aligned}$$

at one point; cf. exercise 18, and note that

$$\left\{ \begin{array}{c} n \\ n-2 \end{array} \right\} = \binom{n}{3} + 3 \binom{n}{4}.$$

The answer can be expressed in a less symmetrical form using the relation $\sum in_i = n + \frac{1}{2}\sum n_j'^2$.

20. The fallacious argument in the discussion following Theorem H is actually valid for this case (the corresponding probabilities *are* independent).

21. [Michigan Math. J. 1 (1952), 81–88.] Let $g(n_1, \dots, n_m) = (n_1 + \dots + n_m)! \Delta(n_1, \dots, n_m)/n_1! \dots n_m! \sigma(n_1, \dots, n_m)$, where $\sigma(x_1, \dots, x_m) = \prod_{1 \leq i < j \leq m} (x_i + x_j)$. To prove that $g(n_1, \dots, n_m)$ is the number of ways to fill the shifting tableau, we must prove that $g(n_1, \dots, n_m) = g(n_1 - 1, \dots, n_m) + \dots + g(n_1, \dots, n_m - 1)$. The identity corresponding to exercise 17 is $x_1 \Delta(x_1 + y, \dots, x_n)/\sigma(x_1 + y, \dots, x_n) + \dots + x_n \Delta(x_1, \dots, x_n + y)/\sigma(x_1, \dots, x_n + y) = (x_1 + \dots + x_n) \Delta(x_1, \dots, x_n)/\sigma(x_1, \dots, x_n)$, independent of y ; for if we calculate the derivative as in exercise 17, we find that $2x_i x_j/(x_j^2 - x_i^2) + 2x_j x_i/(x_i^2 - x_j^2) = 0$.

22. [Journ. de Math. (3) 7 (1881), 167–184.] (This is a special case of exercise 5.1.3–8, with all runs, except perhaps the last, of length 2.) When $n > 0$, element n must appear in one of the rightmost positions of a row; once it has been placed in the rightmost box on row k , we have $\binom{n-1}{2k-1} A_{2k-1} A_{n-2k}$ ways to complete the job. Let

$$h(z) = \sum_{n \geq 1} A_{2n-1} z^{2n-1} / (2n-1)! = \frac{1}{2}(g(z) - g(-z));$$

then

$$h(z)g(z) = \sum_{k,n \geq 1} \binom{n}{2k-1} A_{2k-1} A_{n-2k} z^n / n! = \left(\sum_{n \geq 1} A_{n+1} z^n / n! \right) - 1 = g'(z) - 1.$$

Replace z by $-z$ and add, obtaining $h(z)^2 = h'(z) - 1$; hence $h(z) = \tan z$. Setting $k(z) = g(z) - h(z)$, we have $h(z)k(z) = k'(z)$; hence $k(z) = \sec z$ and $g(z) = \sec z + \tan z = \tan(\frac{1}{2}z + \frac{1}{4}\pi)$. The coefficients A_{2n} are therefore the Euler numbers E_{2n} ; the coefficients A_{2n+1} are the tangent numbers T_{2n+1} . Tables of these numbers appear in *Math. Comp.* 21 (1967), 663–688.

23. Assume that $m = N$, by adding 0's to the shape if necessary; if $m > N$ and $n_m > 0$, the number of ways is clearly zero. When $m = N$ the answer is

$$\det \begin{pmatrix} \binom{n_1+m-1}{m-1} & \binom{n_2+m-2}{m-1} & \cdots & \binom{n_m}{m-1} \\ \vdots & & & \vdots \\ \binom{n_1+m-1}{0} & \binom{n_2+m-1}{0} & \cdots & \binom{n_m}{0} \end{pmatrix}.$$

Proof: We may assume that $n_m = 0$, for if $n_m > 0$, the first n_m columns of the array must be filled with i in row i , and we may consider the remaining shape $(n_1 - n_m, \dots, n_m - n_m)$. By induction on m , the number of ways is

$$\sum_{\substack{n_2 \leq k_1 \leq n_1 \\ \vdots \\ n_m \leq k_{m-1} \leq n_{m-1}}} \det \begin{pmatrix} \binom{k_1+m-2}{m-2} & \binom{k_2+m-2}{m-2} & \cdots & \binom{k_m}{m-2} \\ \vdots & & & \vdots \\ \binom{k_1+m-2}{0} & \binom{k_2+m-2}{0} & \cdots & \binom{k_m}{0} \end{pmatrix},$$

where $n_j - k_j$ represents the number of m 's in row j . The sum on each k_j may be carried out independently, giving

$$\det \begin{pmatrix} \binom{n_1+m-1}{m-1} - \binom{n_2+m-2}{m-1} & \binom{n_2+m-2}{m-1} - \binom{n_3+m-3}{m-1} & \cdots & \binom{n_{m-1}+1}{m-1} - \binom{n_m}{m-1} \\ \vdots & & & \vdots \\ \binom{n_1+m-1}{1} - \binom{n_2+m-2}{1} & \binom{n_2+m-2}{1} - \binom{n_3+m-3}{1} & \cdots & \binom{n_{m-1}+1}{1} - \binom{n_m}{1} \end{pmatrix}.$$

which is the desired answer since $n_m = 0$. The answer can be converted into a Vandermonde determinant by row operations, giving the formula $\Delta(n_1 + m - 1, n_2 + m - 2, \dots, n_m)/(m - 1)!(m - 2)! \dots 0!$. [The answer to this exercise, in connection with an equivalent problem in group theory, appears in D. E. Littlewood's *Theory of Group Characters* (Oxford, 1940), 189.]

25. In general, if u_{nk} is the number of permutations on $\{1, 2, \dots, n\}$, having no cycles of length $> k$, $\sum u_{nk} z^n/n! = \exp(z + z^2/2 + \dots + z^k/k)$; this is proved by multiplying $\exp(z) \times \dots \times \exp(z^k/k)$, obtaining

$$\sum_n z^n/n! \left(\sum_{j_1+2j_2+\dots+kj_k=n} 1/1^{j_1} j_1! 2^{j_2} j_2! \dots \right);$$

cf. exercise 1.3.3–21. Similarly, $\exp(\sum_{s \in S} z^s/s)$ is the corresponding generating function for permutations whose cycle lengths are all members of a given set S .

26. The integral from 0 to ∞ is $n^{(t+1)/4} \Gamma((t+1)/2)/2^{(t+3)/2}$, by the gamma function integral (exercise 1.2.5–20, $t = 2x^2/\sqrt{n}$). So, from $-\infty$ to ∞ , we get 0 when t is odd, otherwise $n^{(t+1)/4} \sqrt{\pi} t!/2^{(3t+1)/2}(t/2)!$.

27. (a) If $r_i < r_{i+1}$ and $c_i < c_{i+1}$, $i < Q_{r_i c_{i+1}} < i+1$ is impossible. If $r_i \geq r_{i+1}$ and $c_i < c_{i+1}$, clearly $r_i \neq r_{i+1}$ so a similar contradiction is obtained. (b) Prove, by induction on the number of rows in the tableau for $a_1 \dots a_i$, that $a_i < a_{i+1}$ implies $c_i < c_{i+1}$, and $a_i > a_{i+1}$ implies $c_i \geq c_{i+1}$. (Consider row 1 and the "bumped" sequences.) (c) This follows from Theorem D(c).

30. [*Discrete Math.* **2** (1972), 73–94.]

31. $x_n = a_{\lfloor n/2 \rfloor}$, where $a_0 = 1$, $a_1 = 2$, $a_n = 2a_{n-1} + (2n-2)a_{n-2}$; $\sum a_n z^n / n! = \exp(2z + z^2) = (\sum t_n z^n / n!)^2$; $x_n \sim \exp(\frac{1}{4}n \ln n - \frac{1}{4}n + \sqrt{n} - \frac{1}{2} - \frac{1}{2} \ln 2)$.

SECTION 5.2

1. Yes, i and j may run through the set of values $1 \leq j < i \leq N$ in any order. This shows that counting can take place simultaneously as records are being read in.
2. The sorting is "stable" as defined at the beginning of the chapter; for the algorithm is essentially sorting by lexicographic order on the *distinct* key-pairs $(K_1, 1), (K_2, 2), \dots, (K_N, N)$. (If we think of each key as extended on the right by its location in the file, no equal keys are present, and the sorting is stable.)
3. It would sort, but not in a "stable" manner; if $K_j = K_i$ and $j < i$, R_j will come after R_i in the final ordering. This change would also make Program C run more slowly.

```

4. ENT1  N          1
        LD2  COUNT,1      N
        LDA  INPUT,1      N
        STA  OUTPUT+1,2    N
        DEC1  1            N
        J1P  *-4           N  ■

```

5. The running time is changed by $N - 1 - A + B$ units, and this is almost always an improvement.

6. $u = 0, v = 9$.

After D1,	COUNT =	0 0 0 0 0 0 0 0 0 0
After D2,	COUNT =	2 2 1 0 1 3 3 2 1 1
After D4,	COUNT =	2 4 5 5 6 9 12 14 15 16
During D5,	COUNT =	2 3 5 5 5 8 9 12 15 16
		$j = 8$
	OUTPUT =	— — — 1G — 4A — — 5L 6A 6T 6I 70 7N — —
	After D5,	OUTPUT = 0C 00 1N 1G 2R 4A 5T 5U 5L 6A 6T 6I 70 7N 8S 9.

7. Yes (note that $\text{COUNT}[K_j]$ is decreased in step D6, and j decreases).
8. It would sort, but not in a "stable" manner (cf. exercise 7).
9. Let $M = v - u$; $\text{LOC}(R_j) \equiv \text{INPUT} + j$; $\text{LOC}(\text{COUNT}[j]) \equiv \text{COUNT} + j$; $\text{LOC}(S_j) \equiv \text{OUTPUT} + j$; $rI1 \equiv i$; $rI2 \equiv j$; $rI3 \equiv i - v, K_j$. Assume that $|u|, |v|$ fit in two bytes.

M	EQU	V-U	
KEY	EQU	0:2	(Satellite information in bytes 3:5.)
1H	ENN3	M	1
	STZ	COUNT+V,3	$M + 1$
	INC3	1	$M + 1$
	J3NP	*-2	$M + 1$
2H	ENT2	N	1
			<u>D1. Clear COUNTs.</u>
			COUNT[v - k] $\leftarrow 0$.
			<u>$u \leq i \leq v$.</u>
			<u>D2. Loop on j.</u>

3H	LD3	INPUT, 2(KEY)	N	<u>D3. Increase COUNT[K_j].</u>
	LDA	COUNT, 3	N	
	INCA	1	N	
	STA	COUNT, 3	N	
	DEC2	1	N	
	J2P	3B	N	$N \geq j > 0.$
	ENN3	M-1	1	<u>D4. Accumulate.</u>
	LDA	COUNT+U	1	$rA \leftarrow COUNT[i - 1].$
4H	ADD	COUNT+V, 3	M	$COUNT[i - 1] + COUNT[i]$
	STA	COUNT+V, 3	M	$\rightarrow COUNT[i].$
	INC3	1	M	
	J3NP	4B	M	$u \leq i \leq v.$
5H	ENT2	N	1	<u>D5. Loop on j.</u>
6H	LD3	INPUT, 2(KEY)	N	<u>D6. Output R_j.</u>
	LD1	COUNT, 3	N	$i \leftarrow COUNT[K_j].$
	LDA	INPUT, 2	N	$rA \leftarrow R_j.$
	STA	OUTPUT, 1	N	$S_i \leftarrow rA.$
	DEC1	1	N	
	ST1	COUNT, 3	N	$COUNT[K_j] \leftarrow i - 1.$
	DEC2	1	N	
	J2P	6B	N	$N \geq j > 0.$ ■

The running time is $(10M + 22N + 10)u$.

10. In order to avoid using N extra "tag" bits [see Section 1.3.3 and *Cybernetics 1* (1965), 95], yet keep the running time essentially proportional to N , we may use the following algorithm based on the cycle structure of the permutation:

- P1. [Loop on i .] Do step P2 for $1 \leq i \leq N$; then terminate the algorithm.
- P2. [Is $p(i) = i$?] Do steps P3 through P5, if $p(i) \neq i$.
- P3. [Begin cycle.] Set $t \leftarrow R_i$, $j \leftarrow i$.
- P4. [Fix R_j .] Set $k \leftarrow p(j)$, $R_j \leftarrow R_k$, $p(j) \leftarrow j$, $j \leftarrow k$. If $p(j) \neq i$, repeat this step.
- P5. [End cycle.] Set $R_j \leftarrow t$, $p(j) \leftarrow j$. ■

This algorithm changes $p(i)$, since the sorting application lets us assume that $p(i)$ is stored in memory. On the other hand, there are applications such as matrix transposition where $p(i)$ is a function of i which is to be computed (not tabulated) in order to save memory space. In such a case we can use the following method, performing steps B1 through B3 for $1 \leq i \leq N$:

- B1. Set $k \leftarrow p(i)$.
- B2. If $k > i$, set $k \leftarrow p(k)$ and repeat this step.
- B3. If $k < i$, do nothing; but if $k = i$ (this means that i is smallest in its cycle), we permute the cycle containing i as follows: Set $t \leftarrow R_i$; then while $p(k) \neq i$ repeatedly set $R_k \leftarrow R_{p(k)}$ and $k \leftarrow p(k)$; finally set $R_k \leftarrow t$. ■

This algorithm is similar to the procedure of J. Boothroyd [*Comp. J.* 10 (1967), 310], but requires less data movement; some refinements have been suggested by I. D. G. MacLeod [*Australian Comp. J.* 2 (1970), 16-19]. For random permutations the analysis in exercise 1.3.3-14 shows that step B2 is performed $(N + 1)H_N - N$ steps on the average. Cf. D. E. Knuth [*Proc. IFIP Congress* 1971, I, 19-27]. Similar algorithms

can be designed to replace $(R_{p(1)}, \dots, R_{p(N)})$ by (R_1, \dots, R_N) , e.g. if the rearrangement in exercise 4 were to be done with **OUTPUT** = **INPUT**.

11. Let $rI1 \equiv i$; $rI2 \equiv j$; $rI3 \equiv k$; $rX \equiv t$.

1H	ENT1	N	1	<u>P1. Loop on i.</u>
2H	CMP1	P,1	N	<u>P2. Is $p(i) = i$?</u>
	JE	8F	N	Jump if $p(i) = i$.
3H	LDX	INPUT,1	A - B	<u>P3. Begin cycle.</u> $t \leftarrow R_i$.
	ENT2	0,1	A - B	$j \leftarrow i$.
4H	LD3	P,2	N - A	<u>P4. Fix R_j.</u> $k \leftarrow p(j)$.
	LDA	INPUT,3	N - A	
	STA	INPUT,2	N - A	$R_j \leftarrow R_k$.
	ST2	P,2	N - A	$p(j) \leftarrow j$.
	ENT2	0,3	N - A	$j \leftarrow k$.
	CMP1	P,2	N - A	
	JNE	4B	N - A	Repeat if $p(j) \neq i$.
5H	STX	INPUT,2	A - B	<u>P5. End cycle.</u> $R_j \leftarrow t$.
	ST2	P,2	A - B	$p(j) \leftarrow j$.
8H	DEC1	1	N	
	J1P	2B	N	$N \geq i \geq 1$. ■

The running time is $(17N - 5A - 7B + 1)u$, where A is the number of cycles in the permutation $p(1) \dots p(N)$, and B is the number of fixed points (1-cycles). By Eqs. 1.3.3–21 and 28, $A = (\min 1, \text{ave } H_N, \max N, \text{dev } \sqrt{H_N - H_N^{(2)}})$; $B = (\min 0, \text{ave } 1, \max N, \text{dev } 1)$, for $N \geq 2$.

12. The following “direct” method is due to M. D. MacLaren. (Assume for convenience that $0 \leq \text{LINK}(P) \leq N$, for $1 \leq P \leq N$, where $\Lambda \equiv 0$.)

- M1. [Initialize.] Set $P \leftarrow \text{HEAD}$, $k \leftarrow 1$.
- M2. [Done?] If $P = \Lambda$ (or equivalently if $k = N + 1$), the algorithm terminates.
- M3. [Ensure $P \geq k$.] If $P < k$, set $P \leftarrow \text{LINK}(P)$ and repeat this step.
- M4. [Exchange.] Interchange R_k and $R[P]$. (Assume that $\text{LINK}(k)$ and $\text{LINK}(P)$ are also interchanged in this process.) Then set $Q \leftarrow \text{LINK}(k)$, $\text{LINK}(k) \leftarrow P$, $P \leftarrow Q$, $k \leftarrow k + 1$, and return to step M2. ■

A proof that MacLaren’s method is valid can be based on an inductive verification of the following property which holds at the beginning of step M2: The entries which are $\geq k$ in the sequence $P, \text{LINK}(P), \text{LINK}(\text{LINK}(P)), \dots, \Lambda$ are $a_1, a_2, \dots, a_{N+1-k}$, where $R_1 \leq \dots \leq R_{k-1} \leq R_{a_1} \leq \dots \leq R_{a_{N+1-k}}$ is the desired final order of the records. Furthermore $\text{LINK}(j) \geq j$ for $1 \leq j < k$, so that $\text{LINK}(j) = \Lambda$ implies $j \geq k$.

It is quite interesting to analyze the above algorithm; one of its remarkable properties is that it can be run backwards, reconstructing the original set of links from the final values of $\text{LINK}(1) \dots \text{LINK}(N)$. Each of the $N!$ possible output configurations with $j \leq \text{LINK}(j) \leq N$ corresponds to exactly one of the $N!$ possible input configurations. If A is the number of times $P \leftarrow \text{LINK}(P)$ in step M3, then $N - A$ is the number of j such that $\text{LINK}(j) = j$ at the conclusion of the algorithm; this occurs if and only if j was largest in its cycle; hence $N - A$ is the number of cycles in the permutation, and $A = (\min 0, \text{ave } N - H_N, \max N - 1)$.

13. D5'. Set $r \leftarrow N$.

- D6'.** If $r = 0$, stop. Otherwise if $\text{COUNT}[K_r] < r$ set $r \leftarrow r - 1$ and repeat this step; if $\text{COUNT}[K_r] = r$, decrease both $\text{COUNT}[K_r]$ and r by 1 and repeat this step. Otherwise set $R \leftarrow R_r$, $j \leftarrow \text{COUNT}[K_r]$, $\text{COUNT}[K_r] \leftarrow j - 1$.
- D7'.** Set $S \leftarrow R_j$, $k \leftarrow \text{COUNT}[K_j]$, $\text{COUNT}[K_j] \leftarrow k - 1$, $R_j \leftarrow R$, $R \leftarrow S$, $j \leftarrow k$. Then if $j \neq r$ repeat this step; if $j = r$ set $R_j \leftarrow R$, $r \leftarrow r - 1$, and go back to D6'. ■

To prove that this is valid, observe that at the beginning of step D6' all records R_j such that $j > r$ which are not in their final resting place must move to the left; when $r = 0$ there can't be any such records since *somebody* must move right. The algorithm is elegant but not "stable" for equal keys; it is intimately related to Foata's construction in Theorem 5.1.2B.

SECTION 5.2.1

1. Yes, equal elements are never moved across each other.
2. Yes, but the running time would be slower when equal elements are present and the sorting would be just the opposite of stable.
3. The following eight-liner is conjectured to be the shortest MIX sorting routine, although it is not recommended for speed. We assume that the numbers appear in locations 1, ..., N (i.e. INPUT EQU 0; otherwise another line of code is necessary).

2H	LDA	0,1	B
	CMPA	1,1	B
	JLE	1F	B
	MOVE	1,1	A
	STA	0,1	A
START	ENT1	N	$A + 1$
1H	DEC1	1	$B + 1$
J1P	2B		$B + 1 \blacksquare$

Note: To estimate the running time of this program, note that A is the number of inversions. B is a reasonably simple function of the inversion table, and (assuming distinct inputs in random order) it has the generating function

$$z^{N-1}(1+z)(1+z^2+z^{2+1}) \\ (1+z^3+z^{3+2}+z^{3+2+1}) \dots \left(1+z^{N-1}+z^{2N-3}+\dots+z^{\binom{N}{2}}\right) / N!.$$

The mean value of B is $N - 1 + \sum_{1 \leq k \leq N} (k - 1)(2k - 1)/6 = (N - 1)(4N^2 + N + 36)/36$; hence the average running time of this program is roughly $\frac{7}{9}N^3u$.

4. Consider the inversion table $B_1 \dots B_N$ of the given input permutation, in the sense of exercise 5.1.1-7. A is one less than the number of B_j 's which are equal to $j - 1$, and B is the sum of the B_j 's. Hence both $B - A$ and B are maximized when the input permutation is $N \dots 2 1$; they both are minimized when the input is $1 2 \dots N$. Hence the minimum achievable time occurs for $A = 0$, $B = 0$, namely $(10N - 9)u$; the maximum occurs for $A = N - 1$, $B = \binom{N}{2}$, namely $4.5N^2 + 2.5N - 6$.

5. The generating function is z^{10N-9} times the generating function for $9B - 3A$. By considering the inversion table as in the previous exercise, remembering that individual entries of the inversion table are independent of each other, the desired

generating function is $z^{10N-9} \prod_{1 < j \leq N} ((1 + z^9 + \dots + z^{9j-18} + z^{9j-12})/j)$. The variance comes to $2.25N^3 + 3.375N^2 - 32.625N + 36H_N - 9H_N^{(2)}$.

6. Treat the input area as a circular list, with position N adjacent to position 1. Take new elements to be inserted from either the left or the right of the current segment of unsorted elements, according as the previously inserted element fell to the right or left of the center of the sorted elements, respectively. Afterwards it will usually be necessary to "rotate" the area, moving each record k places around the circle for some fixed k ; this can be done in $\gcd(N, k)$ passes (cyclically permute $R_i, R_{i+k}, \dots, R_{N+i-k}$ for $1 \leq i \leq \gcd(N, k)$; cf. W. Fletcher and R. Silver, *CACM* 9 (1966), 326).

7. The average value of $|a_j - j|$ is

$$\frac{1}{n}(|1-j| + |2-j| + \dots + |n-j|) = \frac{1}{n} \left(\binom{j}{2} + \binom{n-j+1}{2} \right);$$

summing on j gives

$$\frac{1}{n} \left(\binom{n+1}{3} + \binom{n+1}{3} \right) = \frac{1}{3}(n^2 - 1).$$

8. No; for example, consider the keys 2 1 1 1 1 1 1 1 1 1 1 1.

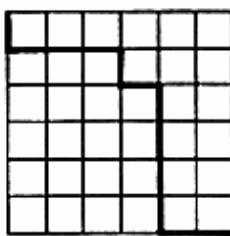
9. For Table 3, $A = 3 + 0 + 2 + 1 = 6$, $B = 3 + 1 + 4 + 21 = 29$; in Table 4, $A = 4 + 2 + 2 + 0 = 8$, $B = 4 + 3 + 8 + 10 = 25$; hence the running time of Program D comes to $786u$ and $734u$, respectively. Although the number of moves has been cut from 41 to 25, the running time is not competitive with Program S since the bookkeeping time for four passes is wasted when $N = 16$. When sorting 16 items we will be better off using only two passes; a two-pass Program D begins to beat Program S at about $N = 13$, although they are fairly equal for awhile (and for such small N the length of the program is perhaps significant).

10. Insert "INC1 INPUT; ST1 3F(0:2)" between lines 07 and 08, and change lines 10-17 to:

3H	CMPA	INPUT+N-H,1	$NT - S$
	JGE	8F	$NT - S$
4H	ENT2	N-H,1	$NT - S - C$
5H	LDX	INPUT,2	B
6H	STX	INPUT+H,2	B
	DEC2	0,4	B
	J2NP	7F	B
	CMPA	INPUT,2	$B - A$
	JL	5B	$B - A$
7H	STA	INPUT+H,2	$NT - S - C$

For a net increase of four instructions, this saves $3(C - T)$ units of time, where C is the number of times $K_j \geq K_{j-h}$. In Tables 3 and 4 the time savings is approximately 87 and 88, respectively; empirically the value of $C/(NT - S)$ seems to be about 0.4 when $h_{s+1}/h_s \approx 2$ and about 0.3 when $h_{s+1}/h_s \approx 3$, so the improvement is worth while. (On the other hand, the analogous change to Program S is not desirable, since the savings in that case is only proportional to $\log N$ unless the input is known to be pretty well ordered.)

11.



12. When the path includes the line segment from (i, j) to $(i + 1, j)$, we have $a_{2i+1} = i + j + 1$ in the corresponding permutation $a_1 \dots a_n$. If $i \leq j$, there are $j - i$ elements to the right of a_{2i+1} which are less than it, but all elements to its left are less; a similar argument for the case $i \geq j$ shows that a_{2i+1} contributes to exactly $|i - j|$ inversions. This completes the proof, since each inversion involves one element with an odd subscript.

13. Put the weight $|i - j|$ on the segment from $(i, j - 1)$ to (i, j) . A path using this segment has $a_{2j} = i + j$, so we may argue as in exercise 12.

14. (a) Interchange i, j in the sum for A_{2n} and add these two sums. (b) Taking half of this result, we see that

$$A_{2n} = \sum_{0 \leq i \leq j} (j - i) \binom{i+j}{i} \binom{2n-i-j}{n-j};$$

hence $\sum A_{2n} z^n = z/(1 - 4z)^2$ by two applications of the stated identity.

The above proof was suggested to the author by Leonard Carlitz. Another proof can be based on interplay between horizontal and vertical weights (cf. exercise 13); but no simple combinatorial derivation of the formula $A_n = \lfloor n/2 \rfloor 2^{n-2}$ is apparent.

15. For $n > 0$,

$$\begin{aligned} \bar{g}_n(z) &= z^{n-1} g_{n-1}(z); \bar{h}_n(z) = \bar{g}_n(z) + z^{-n} \bar{g}_n(z); \\ g_n(z) &= \sum_{1 \leq k \leq n} \bar{g}_k(z) g_{n-k}(z); h_n(z) = \sum_{1 \leq k \leq n} \bar{h}_k(z) h_{n-k}(z). \end{aligned}$$

Letting $G(w, z) = \sum_n g_n(z) w^n$, we find that $wzG(w, z)G(wz, z) = G(w, z) - 1$. From this representation we can deduce that, if $t = \sqrt{1 - 4w} = 1 - 2w - 2w^2 - 4w^3 - \dots$, we have $G(w, 1) = (1 - t)/(2w)$; $G_z(w, 1) = 1/(wt) - (1 - t)/(2w^2)$; $G'(w, 1) = 1/(2t^2) - 1/(2t)$; $G_{zz}(w, 1) = 2/(wt^3) - 2/(w^2t) + (1 - t)/w^3$; $G'(w, 1) = 2/t^4 - 1/t^3$; and $G''(w, 1) = 1/t^3 - (1 - 2w)/t^4 + 10w^2/t^5$. Here lower primes denote differentiation with respect to the first parameter, and upper primes denote differentiation with respect to the second parameter. Similarly from the formula

$$w(zG(wz, z) + G(w, z))H(w, z) = H(w, z) - 1$$

we deduce that

$$H'(w, 1) = w/t^4, \quad H''(w, 1) = -w/t^3 - w/t^4 + 2w/t^5 + (2w^2 + 20w^3)/t^7.$$

The formula manipulation summarized here was done by hand, but it should have been done by computer. In principle all moments of the distribution are obtainable in this way.

16. For $h = 2$ the maximum clearly occurs for the path that goes through the upper right corner of the lattice diagram, namely

$$\binom{\lfloor n/2 \rfloor + 1}{2}.$$

For general h the corresponding number is

$$\tilde{f}(n, h) = \binom{h}{2} \binom{q+1}{2} + \binom{r}{2} (q+1),$$

where q and r are defined in Theorem H; for the permutation with $a_{i+jh} = (n+1-i(q+1)+j, \text{ if } 1 \leq i \leq r; n+1-r-iq+j, \text{ if } r < i \leq h)$ maximizes the number of inversions between each of the $\binom{h}{2}$ pairs of sorted subsequences. The maximum number of moves is obtained if we replace f by \tilde{f} in (6).

17. The only two-ordered permutation of $\{1, 2, \dots, 2n\}$ which has as many as $\binom{n+1}{2}$ inversions is $(n+1) 1 (n+2) 2 \dots (2n) n$. Using this idea recursively, we obtain the permutation defined by adding unity to each element of the sequence $(2^t - 1)^R \dots 1^R 0^R$, where R denotes the operation of writing an integer as a t -bit binary number, then reversing the left-to-right order of the digits!

18. Take out a common factor and let $h_{t+1} = 4N/\pi$; we want to minimize $\sum_{t \geq s \geq 1} h_{s+1}^{1/2}/h_s$, when $h_1 = 1$. Differentiation yields $h_s^3 = 4h_{s+1}^2 h_{s-1}$, which has the solution $h_s = 2^{s-2+s/(2^s-1)} h_{s+1}^{(2^{s-1}-1)/(2^s-1)}$. The minimum value of the originally-given estimate comes to $(1 - 2^{-t}) \pi^{(2^{t-1}-1)/(2^t-1)} N^{1+(2^{t-1}+1)/(2^t-1)} / 2^{1+(t-1)/(2^t-1)}$ which rapidly approaches $N\sqrt{\pi N}/2$ as $t \rightarrow \infty$.

Typical examples of "optimum" h 's when $N = 1000$ (see also Table 6):

$$h_3 = 57.64, \quad h_2 = 6.13, \quad h_1 = 1;$$

$$h_5 = 284.5, \quad h_4 = 67.23, \quad h_3 = 16.34, \quad h_2 = 4.028, \quad h_1 = 1;$$

$$h_{10} = 9165, \quad h_9 = 12294, \quad h_8 = 7120, \quad h_7 = 2709, \\ h_6 = 835.5, \quad \dots, \quad h_2 = 3.97, \quad h_1 = 1.$$

19. Let $g(n, h) = H_r - 1 + \sum_{r < j \leq h} q/(qj+r)$, where q and r are defined in Theorem H; then replace f by g in (6).

20. (This is much harder to write down than to understand.) Assume that a k -ordered file R_1, \dots, R_N has been h -sorted, and let $1 \leq i \leq N-k$; we want to show that $K_i \leq K_{i+k}$. Find u, v such that $1 \leq u, v \leq h$; $i \equiv u$ and $i+k \equiv v$ (modulo h); and apply Lemma L with $m = (u-v+k)/h$, $r = \lfloor (N-k+h-u)/h \rfloor$, $n+r = \lfloor (N+h-u)/h \rfloor$, $x_j = K_{v+(j-1)h}$, $y_j = K_{u+(j-1)h}$.

21. If $ah + bk = hk - h - k$, then $a \bmod k = k-1$ and $b \bmod h = h-1$; hence $ah + bk \geq (k-1)h + (h-1)k > hk - h - k$, a contradiction. Conversely, if $n \geq (h-1)(k-1)$, choose a so that $0 \leq a < k$ and $ah \equiv n$ (modulo k). Then $(n-ah)/k$ is a nonnegative integer, b ; hence n is representable.

A slight strengthening of this argument shows that exactly $\frac{1}{2}(h-1)(k-1)$ positive integers are unrepresentable in the stated form. (See R. Z. Norman, *AMM* 67 (1960), 594.)

22. To avoid cumbersome notation, consider $h = 4$, which is representative of the general case. Let n_k be the smallest number representable in the form $15a_0 + 31a_1 + \dots$ which is congruent to k (modulo 15); then we find easily that

$$k = 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10 \ 11 \ 12 \ 13 \ 14$$

$$n_k = 15 \ 31 \ 62 \ 63 \ 94 \ 125 \ 126 \ 127 \ 158 \ 189 \ 190 \ 221 \ 252 \ 253 \ 254.$$

Hence $239 = 2^4(2^4 - 1) - 1$ is the largest unrepresentable number, and the total number of unrepresentables is

$$\begin{aligned}x_4 &= (n_1 - 1 + n_2 - 2 + \cdots + n_{14} - 14)/15 \\&= (2 + 4 + 4 + 6 + 8 + 8) + 8 + (10 + 12 + 12 + 14 + 16 + 16) + 16 \\&= 2x_3 + 8 \cdot 9;\end{aligned}$$

in general, $x_h = 2x_{h-1} + 2^{h-1}(2^{h-1} + 1)$.

For the other problem the answers are $2^{2h} + 2^h + 2$ and $2^{h-1}(2^h + h - 1) + 2$, respectively.

23. Each of the N numbers has at most $\lceil (h_s+2 - 1)(h_s+1 - 1)/h_s \rceil$ inversions in its subfile.

24. (Solution obtained jointly with V. Pratt.) Construct the " h -recidivous permutation" of $\{1, 2, \dots, N\}$ as follows. Start with $a_1 \dots a_N$ blank; then for $j = 2, 3, 4, \dots$ do Step j : Fill in all blank positions a_i from left to right, using the smallest number that has not yet appeared in the permutation, whenever $(2^h - 1)j - i$ is a positive integer representable as in exercise 22. Continue until all positions are filled. Thus the 2-recidivous permutation for $N = 20$ is

6 2 1 9 4 3 12 7 5 15 10 8 17 13 11 19 16 14 20 18.

The h -recidivous permutation is $(2^k - 1)$ -ordered for all $k \geq h$. When $2^h < j \leq N/(2^h - 1)$, exactly $2^h - 1$ positions are filled during step j ; the $(k + 1)$ st of these adds at least $2^{h-1} - 2k$ to the number of moves required to $(2^{h-1} - 1)$ -sort the permutation. Hence the number of moves to sort the h -recidivous permutation with increments $h_s = 2^s - 1$ when $N = 2^{h+1}(2^h - 1)$ is $> 2^{3h-4} > \frac{1}{128}N^{3/2}$. Pratt has generalized this construction to a large family of similar sequences, including (8), in his Ph.D. thesis (Stanford University, 1972).

25. F_{n+1} [this result is due to H. B. Mann, *Econometrica* 13 (1945), 256]; for the permutation must begin either 1 ... or 2 1 There are at most $\lfloor n/2 \rfloor$ inversions; and the total number of inversions is

$$\frac{n-1}{5} F_n + \frac{2n}{5} F_{n-1}.$$

(See exercise 1.2.8-12.) Note that the F_{n+1} permutations can conveniently be represented by "Morse code" sequences of dots and dashes, where a dash corresponds to an inversion; thus when $n = 4$ the permutations

1 2 3 4, 1 2 4 3, 1 3 2 4, 2 1 3 4, 2 1 4 3

correspond respectively to the Morse code sequences

..., ..., --., -.., --

of length 4. Hence we have found the total number of dashes among all Morse code sequences of length n .

Since 3-sorting followed by 2-sorting doesn't leave a "random" 3- and 2-ordered permutation, the total number of inversions does not tell us anything precise about the average number obtained in Algorithm D.

26. Yes; a shortest example is 4 1 3 7 2 6 8 5, which has nine inversions. In general, the construction $a_{3k+s} = 3k + 4s$ for $-1 \leq s \leq 1$ yields files which are 3-, 5-, and 7-ordered, having approximately $\frac{4}{3}N$ inversions. When $N \bmod 3 = 2$ this construction is best possible.

28. 255 63 15 7 3 1. But better sequences are possible, see exercise 29.

29. Experiments by C. Tribolet in 1971 resulted in the choices 373 137 53 19 7 3 1 ($B_{\text{ave}} \approx 7100$) and 317 101 31 11 3 1 ($B_{\text{ave}} \approx 8300$). [The first of these resulted in a sorting time of $126457u$, compared to $131002u$ when the same data were sorted using increments (8).] In general, Tribolet suggests letting h_s be the nearest prime number to $N^{(s-1)/t}$. Experiments by Shelby Siegel in 1972 indicate that the best number of increments in such a method, for $N \leq 10000$, is $t \approx \frac{4}{3} \ln(N/5.75)$.

Another good sequence, found by Robert L. Tomlinson, Jr., is 199 79 31 11 5 1 ($B_{\text{ave}} \approx 7600$).

The best three-increment sequence, according to extensive tests by Carole M. McNamee, appears to be 45 7 1 ($B_{\text{ave}} \approx 17600$). For four increments, 91 23 7 1 was the winner in her tests ($B_{\text{ave}} \approx 11500$), but a rather broad range of increments gave roughly the same performance.

30. The number of integer points in the triangular region

$$\{x \ln 2 + y \ln 3 < \ln N, x \geq 0, y \geq 0\} \quad \text{is} \quad \frac{1}{2}(\log_2 N)(\log_3 N) + O(\log N).$$

While we are h -sorting, the file is already $2h$ -ordered and $3h$ -ordered, by Theorem K; hence exercise 25 applies.

31.	01	START	ENT3	T	1
	02	1H	LD4	H,3	T
	03		ENN2	INPUT-N,4	T
	04		ST2	6F(0:2)	T
	05		ST2	7F(0:2)	T
	06		ST2	4F(0:2)	T
	07		ENT2	0,4	T
	08		JMP	9F	T
	09	2H	LDA	INPUT+N,1	$NT - S - B + A$
	10	4H	CMPA	INPUT+N-H,1	$NT - S - B + A$
	11		JGE	8F	$NT - S - B + A$
	12	6H	LDX	INPUT+N-H,1	B
	13		STX	INPUT+N,1	B
	14	7H	STA	INPUT+N-H,1	B
	15		INC1	0,4	B
	16	8H	INC1	0,4	$NT - B + A$
	17		J1NP	2B	$NT - B + A$
	18		DEC2	1	S
	19	9H	ENT1	-N,2	$T + S$
	20		J2P	8B	$T + S$
	21		DEC3	1	T
	22		J3P	1B	T

Here A is the number of right-to-left maxima, in the sense that A in Program D is the number of left-to-right minima; both quantities have the same statistical behavior.

The simplifications in the inner loop have cut the running time to $7NT + 7A - 2S + 1 + 15T$ units, curiously independent of B !

When $N = 8$ the increments are 6, 4, 3, 2, 1, and we have $A_{\text{ave}} = 3.892$, $B_{\text{ave}} = 6.762$; the average total running time is $276.24u$. (Cf. Table 5.) Both A and B are maximized in the permutation 7 3 8 4 5 1 6 2. When $N = 1000$ there are 40 increments, 972, 864, 768, 729, . . . , 8, 6, 4, 3, 2, 1; empirical tests like those in Table 6 give $A \approx 875$, $B \approx 4250$, and a total time of about $268000u$ (about twice as long as Program D with the increments of exercise 28).

Instead of storing the increments in an auxiliary table, it is convenient to generate them as follows on a binary machine:

- P1. Set $m \leftarrow 2^{\lceil \log_2 N \rceil - 1}$, the largest power of 2 less than N .
- P2. Set $h \leftarrow m$.
- P3. Use h as the increment for one sorting pass.
- P4. If h is even, set $h \leftarrow h + h/2$; then if $h < N$, return to P3.
- P5. Set $m \leftarrow \lfloor m/2 \rfloor$ and if $m \geq 1$ return to P2.

Although the increments are not being generated in descending order, the order specified here is sufficient to make the sorting algorithm valid.

32. 4 12 11 13 2 0 8 5 10 14 1 6 3 9 16 7 15.

33. Two separate types of improvements can be made. First, by assuming that the artificial key K_0 is ∞ , we can omit testing whether or not $p > 0$. (This idea has been used, for example, in Algorithm 2.2.4A.) Secondly, a standard “optimization” technique: We can make two copies of the inner loop with the register assignments for p and q interchanged; this avoids the assignment $q \leftarrow p$. (This idea has been used in exercise 1.1-3.)

Thus we assume that location INPUT contains the largest possible value in its (0:3) field, and we replace lines 07 and following of Program L by:

07	8H	LD3	INPUT, 2(LINK)	B'	$p \leftarrow L_q$. (Here $p \equiv rI2$, $q \equiv rI3$.)
08		CMPA	INPUT, 3(KEY)	B'	
09		JG	4F	B'	To L4 if $K > K_p$.
10	7H	ST1	INPUT, 2(LINK)	N'	$L_q \leftarrow j$.
11		ST3	INPUT, 1(LINK)	N'	$L_j \leftarrow p$.
12		JMP	6F	N'	Go to decrease j .
13	4H	LD2	INPUT, 3(LINK)	B''	$p \leftarrow L_q$. (Here $p \equiv rI2$, $q \equiv rI3$.)
14		CMPA	INPUT, 2(KEY)	B''	
15		JG	8B	B''	To L4 if $K > K_p$.
16	5H	ST1	INPUT, 3(LINK)	N''	$L_q \leftarrow j$.
17		ST2	INPUT, 1(LINK)	N''	$L_j \leftarrow p$.
18	6H	DEC1	1	N	$j \leftarrow j - 1$.
19		ENT3	0	N	$q \leftarrow 0$.
20		LDA	INPUT, 1	N	$K \leftarrow K_j$.
21		J1P	4B	N	$N > j \geq 1$. ■

Here $B' + B'' = B + N - 1$, $N' + N'' = N - 1$, so the total running time is $5B + 14N + N' - 3$ units. (N' is the number of elements with an odd number of

lesser elements to their right; so it has the statistics (min 0, ave $\frac{1}{2}N + \frac{1}{4}H_{\lfloor N/2 \rfloor} - \frac{1}{2}H_N$, max $N - 1$.)

The trick of setting $K_0 \leftarrow -\infty$ makes a similar saving in the running time of Program S, reducing it to $(8B + 11N - 10)u$.

34. There are $\binom{N}{n}$ sequences of N choices in which the given list is chosen n times; each such sequence has probability $(1/M)^n(1 - 1/M)^{N-n}$ of occurring, since the given list is chosen with probability $1/M$.

35. Program L: $A = 3$, $B = 41$, $N = 16$, time = $496u$. Program M: $A = 2 + 1 + 1 + 3 = 7$, $B = 2 + 0 + 3 + 3 = 8$, $N = 16$, time = $549u$. (Because of the 16 multiplications. But the improved Program L in exercise 33 takes only $358u$.)

36. To minimize $AC/M + BM$ we need $M = \sqrt{AC/B}$, so M is one of the integers just above or below this quantity. (In the case of Program M we would choose M proportional to N .)

37. The stated identity is equivalent to

$$g_{NM}(z) = M^{-N} \sum_{n_1 + \dots + n_M = N} \left(\frac{N!}{n_1! \dots n_M!} \right) g_{n_1}(z) \dots g_{n_M}(z),$$

which is proved as in exercise 34. It may be of interest to tabulate some of these generating functions, to indicate the trend for increasing M :

$$\begin{aligned} g_{41}(z) &= (216 + 648z + 1080z^2 + 1296z^3 + 1080z^4 + 648z^5 + 216z^6)/5184, \\ g_{42}(z) &= (945 + 1917z + 1485z^2 + 594z^3 + 135z^4 + 81z^5 + 27z^6)/5184, \\ g_{43}(z) &= (1704 + 2264z + 840z^2 + 304z^3 + 40z^4 + 24z^5 + 8z^6)/5184. \end{aligned}$$

Differentiating the stated identity twice with respect to z and then setting $z = 1$ yields

$$\begin{aligned} G''_{NM}(1) &= M^{-N} \left(M(M-1) \sum_{m,n} \binom{N}{m, n, N-m-n} \right. \\ &\quad \times (M-2)^{N-m-n} g'_m(1) g'_n(1) + M \sum_n \binom{N}{n} (M-1)^{N-n} g''_n(1) \left. \right). \end{aligned}$$

Using the formula

$$g''_n(1) = \frac{3}{2} \binom{n}{4} + \frac{5}{3} \binom{n}{3},$$

these sums present no special difficulty, and we find that

$$G''_{NM}(1) = \left(\frac{3}{2} \binom{N}{4} + \frac{5}{3} \binom{N}{3} \right) / M^2.$$

Finally since

$$G'_{NM}(1) = \frac{1}{2} \binom{N}{2} / M, \quad G'_{NM}(1)^2 = \left(\frac{3}{2} \binom{N}{4} + \frac{3}{2} \binom{N}{3} + \frac{1}{4} \binom{N}{2} \right) / M^2,$$

the variance is

$$\left(\frac{1}{6} \binom{N}{3} + \frac{(2M-1)}{4} \binom{N}{2} \right) / M^2.$$

38. $\sum_{j,n} \binom{N}{n} p_j^n (1-p_j)^{N-n} \binom{n}{2} = \binom{N}{2} \sum_j p_j^2$; setting $p_j = F(j/M) - F((j-1)/M)$, and $F'(x) = f(x)$, this converges to $\binom{N}{2}/M$ times $\int_0^1 f(x)^2 dx$ when F is reasonably well behaved.

39. (Solution by R. W. Floyd.) A deletion-insertion operation essentially moves only a_i . In a sequence of such operations, unmoved elements retain their relative order. Therefore if π can be sorted with k deletion-insertions, it has an increasing subsequence of length $n-k$; and conversely. Hence $\text{dis } (\pi) = n - (\text{length of longest increasing subsequence of } \pi)$. (A longest increasing subsequence can be found in $O(n \log n)$ steps; cf. Section 5.1.4.)

SECTION 5.2.2

1. No, it has $2m + 1$ less inversions, where $m \geq 0$ is the number of elements a_k such that $i < k < j$ and $a_i > a_k > a_j$. (Hence all exchange-sorting methods will eventually converge to a sorted permutation.)
2. (a) 6. (b) [A. Cayley, *Philos. Mag.* 34 (1849), 527–529.] Consider the cycle representation of π : Any exchange of elements in the same cycle increases the number of cycles by 1; any exchange of elements in different cycles decreases the number by 1. (This is essentially the content of exercise 2.2.4–3.) A completely sorted permutation is characterized by having n cycles. Hence $xch(\pi)$ is n minus the number of cycles in π .
3. Yes, equal elements are never moved across each other.
4. The probability that $b_1 > b_2, \dots, b_n$ in the inversion table, namely

$$\left(\sum_{1 \leq k < n} k! \cdot k^{n-k-1} \right) / n! = \sqrt{\pi/2n} + O(n^{-1}) = \text{negligible.}$$

5. We may assume that $r > 0$. Let $b'_i = (b_i \geq r \Rightarrow b_i - r + 1; 0)$ be the inversion table after $r - 1$ passes. If $b'_i > 0$, element i is preceded by b'_i larger elements, the largest of which will bubble up at least to position $b'_i + i$ (in view of the i elements $\leq i$). Furthermore if element j is the rightmost to be exchanged, we have $b'_j > 0$ and $\text{BOUND} = b'_j + j - 1$ after the r th pass.

6. Solution 1: An element displaced farthest to the right of its final position moves one step left on each pass except the last. Solution 2 (higher level): By exercise 5.1.1–8, answer (f), $a'_i - i = b_i - c_i$, for $1 \leq i \leq n$, where $c_1 c_2 \dots c_n$ is the dual inversion table. If $b_j = \max(b_1, \dots, b_n)$ then $c_j = 0$.

$$7. (2(n+1)(1+P(n)) - P(n+1)) - P(n) - P(n)^2)^{1/2} = \sqrt{(2 - \pi/2)n} + O(1).$$

8. For $i < k + 2$ there are $j + k - i + 1$ choices for b_i ; for $k + 2 \leq i < n - j + 2$ there are $j - 1$ choices; and for $i \geq n - j + 2$ there are $n - i + 1$.

10. (a) If $i = 2k - 1$, from $(k - 1, a_i - k)$ to $(k, a_i - k)$. If $i = 2k$, from $(a_i - k, k - 1)$ to $(a_i - k, k)$. (b) Step a_{2k-1} is above the diagonal iff $k \leq a_{2k-1} - k$ iff $a_{2k-1} \geq 2k$ iff $a_{2k-1} > a_{2k}$ iff $a_{2k} \leq 2k - 1$ iff $a_{2k} - k \leq k - 1$ iff step a_{2k} is above the diagonal. Exchanging them interchanges horizontal and vertical steps. (c) Step a_{2k+d} is at least m below the diagonal iff $k + m - 1 \geq a_{2k+d} - (k + m) + m$ iff

$a_{2k+d} < 2k + m$ iff $a_{2k} \geq 2k + m$ iff $a_{2k} - k \geq k + m$ iff step a_{2k} is at least m below the diagonal. (If $a_{2k+d} < 2k + m$ and $a_{2k} < 2k + m$, there are at least $(k + m) + k$ elements less than $2k + m$; that's impossible. If $a_{2k+d} \geq 2k + m$ and $a_{2k} \geq 2k + m$, one of the \geq must be $>$; but we can't fit all of the elements $\leq 2k + m$ into less than $(k + m) + k$ positions. Hence $a_{2k+d} < a_{2k}$ iff $a_{2k+d} < 2k + m$ iff $2k + m \leq a_{2k}$. A rather unexpected result!)

11. 16 10 13 5 14 6 9 2 15 8 11 3 12 4 7 1 (61 exchanges), by considering the lattice diagram. In general, when $N = 2^t$ the set $\{K_2, K_4, \dots\}$ should be $\{1, 2, \dots, \lfloor N/3 \rfloor + 1, \lfloor N/3 \rfloor + 3, \lfloor N/3 \rfloor + 5, \dots\}$, permuted so as to maximize the exchanges for $N/2$ elements.

12. In this program, $TT \equiv 2^{t-1}$, $p \equiv rI1$, $r \equiv rI2$, $i \equiv rI3$, $i + d - N \equiv rI4$; assume that $N \geq 2$.

01		START	ENT1	TT	1	<u>M1. Initialize p.</u> $p \leftarrow 2^{t-1}$.
02			ST1	P	1	
03	2H		ENT2	TT	T	<u>M2. Initialize q, r, d.</u>
04			ST2	Q(1:2)	T	$q \leftarrow 2^{t-1}$.
05			ENT2	0	T	$r \leftarrow 0$.
06			ENT4	0,1	T	$rI4 \leftarrow p$.
07	3H		DEC4	N	A	<u>M3. Loop on i.</u>
08			ENT3	0	A	$i \leftarrow 0$.
09	8H		ENTA	0,3	C + E	(At this point $rI4 < 0$.)
10			AND	P	C + E	$rA \leftarrow i \wedge p$.
11			DECA	0,2	C + E	
12			JAZ	4F	C + E	Jump if $i \wedge p = r$.
13			INC3	0,1	D	Otherwise $i \leftarrow i + p$.
14			INC4	0,1	D	
15			J4NN	5F	D	Exit loop if $i + d \geq N$.
16	4H		LDA	INPUT+1,3	C	<u>M4. Compare/exchange</u>
17			CMPA	INPUT+N+1,4	C	<u>$R_{i+1}:R_{i+d+1}$</u> .
18			JLE	*+4	C	
19			LDX	INPUT+N+1,4	B	
20			STX	INPUT+1,3	B	
21			STA	INPUT+N+1,4	B	
22			INC3	1	C	$i \leftarrow i + 1$.
23			INC4	1	C	
24			J4N	8B	C	Repeat loop if $i + d < N$.
25	5H		ENT2	0,1	A	<u>M5. Loop on q.</u> $r \leftarrow p$.
26	Q		ENT4	*	A	$rI4 \leftarrow q$.
27			ENTA	0,4	A	
28			SRB	1	A	
29			STA	Q(1:2)	A	$q' \leftarrow q/2$.
30			DEC4	0,1	A	$rI4 \leftarrow q - p$.
31			J4P	3B	A	To M3 if $q \neq p$.
32	6H		ENTA	0,1	T	<u>M6. Loop on p.</u>
33			SRB	1	T	
34			STA	P	T	
35			LD1	P	T	$p \leftarrow \lfloor p/2 \rfloor$.
36			JANZ	2B	T	To M2 if $p \neq 0$. ■

The running time depends on six quantities, only one of which depends on the input data (the remaining five are functions of N alone): $T = t$, the number of "major cycles"; $A = t(t+1)/2$, the number of passes or "minor cycles"; B = the (variable) number of exchanges; C = the number of comparisons; D = the number of times $i \wedge p \neq r$ in step M2; and E = the number of times $i \wedge p \neq r$ and $i + p + d \geq N$ in step M2. When $N = 2^t$, it can be shown that $D = 2^t(t-2) + 2 + E$, and $E = \binom{t}{2}$. For Table 1, $T = 4$, $A = 10$, $B = 3 + 0 + 1 + 4 + 0 + 0 + 8 + 0 + 4 + 5 = 25$, $C = 63$, $D = 40$, $E = 6$, so the total running time is $11A + 6B + 13C + 3D + 5E + 13T + 3 = 1284u$.

13. No, nor are Algorithms Q, R.

14. (a) When $p = 1$ we do $(2^{t-1} - 0) + (2^{t-1} - 1) + (2^{t-1} - 2) + (2^{t-1} - 4) + \cdots + (2^{t-1} - 2^{t-2}) = (t-1)2^{t-1} + 1$ comparisons for the final merge. (b) $x_t = x_{t-1} + \frac{1}{2}(t-1) + 2^{-t} = \cdots = x_0 + \sum_{0 \leq k < t} (\frac{1}{2}k + 2^{-k-1}) = \frac{1}{2}\binom{t}{2} + 1 - 2^{-t}$. Hence $c(2^t) = 2^{t-2}(t^2 - t + 4) - 1$.

15. (a) Consider the number of comparisons such that $i + d = N$; then use induction on r . (b) If $b(n) = c(n+1)$, we have $b(2n) = a(1) + \cdots + a(2n) = a(0) + a(1) + a(1) + \cdots + a(n-1) + a(n) + x(1) + x(2) + \cdots + x(2n) = 2b(n) + y(2n) - a(n)$; similarly $b(2n+1) = 2b(n) + y(2n+1)$. (c) Cf. exercise 1.2.4-42. (d) A rather laborious calculation of $(z(N) + 2z(\lfloor N/2 \rfloor) + \cdots) - a(N)$, using formulas such as

$$\sum_{0 \leq k \leq n} 2^k(n-k) = 2^{n+1} - n - 2, \quad \sum_{0 \leq k \leq n} 2^k \binom{n-k}{2} = 2^{n+1} - \binom{n+2}{2} - 1,$$

leads to the result

$$c(N) = N \left(\frac{1}{2} \binom{e_1}{2} + 2e_1 - 1 \right) - 2^{e_1}(e_1 - 1) - 1 \\ + \sum_{1 \leq j \leq r} 2^{e_j} \left(e_1 + \cdots + e_{j-1} - j(e_1 - 1) + \frac{1}{2} \binom{e_1 - e_j}{2} \right).$$

16. Consider the $\binom{2n}{n}$ lattice paths from $(0, 0)$ to (n, n) . For $k \geq 0$, the number of times a line $(i, i+k) - (i, i+k+1)$ is traversed for some i , summed over all paths, is

$$\sum_i \binom{k+2i}{i} \binom{2n-1-k-2i}{n-i},$$

the coefficient of x^{2n} in $x^{k+1}y^{2k+1}/(1-4x)$, where $y = (1 - \sqrt{1-4x})/2x$. (Cf. exercise 5.2.1-14.) It can now be shown that the number of exchanges during the final merge when $N = 2n$, summed over all paths, is the coefficient of x^N in

$$\frac{x}{(1-4x)^{3/2}} \frac{1}{1-z} \left(1 + \frac{2z}{1+z^2} + \frac{2z^2}{1+z^4} + \frac{2z^4}{1+z^8} + \cdots \right), \quad z = xy^2 = y - 1.$$

The asymptotic value of this coefficient is desired.

17. K_{N+1} is inspected when we are sorting a subfile with $r = N$ and K_l the largest key. K_0 is inspected during a straight-insertion sorting phase with $l = 1$, when left-to-right minima sink to position R_1 .

18. Each comparison is followed by a transfer; the partitioning process for $R_l \dots R_r$ ends with $i = \lceil (l+r)/2 \rceil$ in step Q7, bisecting the subfile as perfectly as possible. (Quantitatively speaking, we replace Eqs. (17) by $A = 1$, $B = \lfloor (N-1)/2 \rfloor$, $C = N$, $X = 1 - (N \bmod 2)$; this puts us essentially in the *best* case of the algorithm, as in exercise 27 below, except that $B \approx \frac{1}{2}C$.) If the " $<$ " signs in steps Q3 and Q5 are changed, we need " $<$ " signs in (13); and the algorithm behaves terribly, letting j run from N down to 0 each time and making the slowest conceivable progress. Thus the extra transfers are well worth making. [Another way to handle equal keys is suggested in exercise 30.]

19. Yes, the other subfiles may be processed in any order (without increasing the size of the storage area); but a stack is easiest to work with.

20. $\lfloor \log_2 (N+1)/(M+2) \rfloor$. (The worst case occurs when $N = 2^k(M+1) - 1$ and all subfiles are perfectly bisected when they are partitioned.)

21. If $s > 1$, the first occurrence of step Q4 removes s from the keys in memory, and no comparison in steps Q3 and Q5 will come out $=$. Exactly t records are transferred from the area $R_2 \dots R_s$ to $R_{s+1} \dots R_N$; and $t+h$ are transferred from $R_s \dots R_N$ to $R_1 \dots R_{s-1}$. If $h = 1$, the last comparison is $K_i : K$ with $i = s+1$ and $j = s$; if $h = 0$ it is $K : K_j$ with $j = s-1$ and $i = s$. So we obtain (17); in fact, $C' = N + 2 - s - h - \delta_{s1}$ and $C'' = s - 1 + h$.

22. The stated relations for $A_N(z)$ follow easily because $A_{s-1}(z)A_{N-s}(z)$ is the generating function for the value of A after independently sorting randomly ordered files of sizes $s-1$ and $N-s$. Similarly, we obtain the relations

$$B_N(z) = \sum_{1 \leq s \leq N} \sum_{0 \leq t \leq s} b_{stN} z^t B_{s-1}(z) B_{N-s}(z),$$

$$C_N(z) = \frac{1}{N} \sum_{1 \leq s \leq N} z^{N+1-\delta_{s1}} C_{s-1}(z) C_{N-s}(z),$$

$$D_N(z) = \frac{1}{N} \sum_{1 \leq s \leq N} D_{s-1}(z) D_{N-s}(z),$$

$$E_N(z) = \frac{1}{N} \sum_{1 \leq s \leq N} E_{s-1}(z) E_{N-s}(z),$$

$$L_N(z) = \frac{1}{N} \sum_{1 \leq s \leq N} L_{s-1}(z) L_{N-s}(z),$$

$$X_N(z) = \frac{1}{N} \sum_{1 \leq s \leq N} (1 + (z-1)h_{sN}) X_{s-1}(z) X_{N-s}(z),$$

for $N > M$. Here b_{stN} is the probability that s and t have given values in a file of length N , namely

$$\binom{s-1}{t} \binom{N-s}{t} / N \binom{N-1}{s-1},$$

which is $(1/N!)$ times the $(s-1)!$ ways to permute $\{1, \dots, s-1\}$ times the $(N-s)!$

ways to permute $\{s+1, \dots, N\}$ times the $\binom{s-1}{t} \binom{N-s}{t}$ patterns with t displaced elements on each side; and h_{sN} is the probability that $h = 1$ when N and s are given, namely $(s-1)/(N-1)$. For $0 \leq N \leq M$, we have $B_N(z) = C_N(z) = X_N(z) = 1$; $L_N(z) = 1$, except $L_0(z) = L_1(z) = z$; $D_N(z) = z^{N-1}$, except $D_0(z) = 1$; and $E_N(z)$ is $\prod_{1 \leq k \leq N} ((1+z+\dots+z^{k-1})/k)$.

23. When $N > M$, $A_N = 1 + (2/N) \sum_{0 \leq k < N} A_k$; $B_N = \sum_{0 \leq t < s \leq N} b_{stN}(t + B_{s-1} + B_{N-s}) = (1/N) \sum_{1 \leq s \leq N} ((s-1)(N-s)/(N-1) + B_{s-1} + B_{N-s}) = (N-2)/6 + (2/N) \sum_{0 \leq k < N} B_k$ [cf. exercise 22]; $D_N = (2/N) \sum_{0 \leq k < N} D_k$; E_N and L_N are similar; $X_N = (1/N) \sum_{1 \leq s \leq N} (h_{sN} + X_{s-1} + X_{N-s}) = \frac{1}{2} + (2/N) \sum_{0 \leq k < N} X_k = \frac{1}{2} A_N$. Each of these recurrences has the form (20) for some function f_n .

24. The recurrence $C_N = N - 1 + (2/N) \sum_{0 \leq k < N} C_k$, $N > M$, has the solution $(N+1)(2H_{N+1} - 2H_{M+2} + 1 - 4/(M+2) + 2/(N+1))$, $N > M$. (So we save about $4N/M$ comparisons, but each comparison takes longer.)

25. (Use (17) repeatedly with $s = 1$.) $A = N - M$, $B = 0$, $C = \binom{N+1}{2} - \binom{M+1}{2}$, $D = M - 1$, $E = 0$, $L = N - M + \delta_{M1}$, $X = 0$.

26. $N M (M-1) \dots 1 (M+1) (M+2) \dots (N-1)$. Using (17) repeatedly with $s = N$, we have $A = N - M$, $B = 0$, $C = \binom{N+2}{2} - \binom{M+2}{2}$, $D = M - 1$, $E = \binom{M}{2}$, $L = N - M + \delta_{M1}$, $X = N - M$. (Since $B = 0$ this may not be the worst possible case.)

27. Eq. (17) with $A = 1$, $B = 0$, $C = N + 1$, $X = 1$, and $s = \lfloor (N+1)/2 \rfloor$ is best for the partitioning; and $E = 0$ is best for straight insertion. When $N = (M+1)2^k - 1$, the totals are $A = X = 2^k - 1$, $B = E = 0$, $C = k(M+1)2^k$, $D = 2^k(M-1)$; when $N = (M+2)2^k - 1$, $A = X = 2^{k+1} - 1$, $C = (k+1)(M+2)2^k$, $D = 2^{k+1}\lfloor (M-1)/2 \rfloor$. In general, the approximate values are $A = X \approx$ between N/M and $2N/M$; $B = E = 0$; $C \approx N \log_2(N/M)$; $D \approx N$.

28. The recurrence

$$C_n = n + 1 + \frac{2}{\binom{n}{3}} \sum_{1 \leq k \leq n} (k-1)(n-k)C_{k-1}$$

can be transformed into

$$\binom{n}{3} C_n - 2 \binom{n-1}{3} C_{n-1} + \binom{n-2}{3} C_{n-2} = 2(n-1)(n-2) + 2(n-2)C_{n-2}.$$

29. In general, consider the recurrence

$$C_n = n + 1 + \frac{2}{\binom{n}{2t+1}} \sum_{1 \leq k \leq n} \binom{k-1}{t} \binom{n-k}{t} C_{k-1},$$

which arises when the median of $2t+1$ elements governs the partitioning. Letting $C(z) = \sum_n C_n z^n$, the recurrence can be transformed to $(1-z)^{t+1} C^{(2t+1)}(z)/(2t+2)! = 1/(1-z)^{t+2} + C^{(t)}(z)/(t+1)!$. Let $f(x) = C^{(t)}(1-x)$; then $p_t(\vartheta)f(x) = (2t+2)!/(x)^{t+2}$, where ϑ denotes the operator $x(d/dx)$, and $p_t(x) = (t-x)^{\underline{t+1}} - (2t+2)^{\underline{t+1}}$. The general solution to $(\vartheta - \alpha)g(x) = x^\beta$ is $g(x) = x^\beta/(\beta - \alpha) + Cx^\alpha$, for $\alpha \neq \beta$; $g(x) = x^\beta(\ln x + C)$ for $\alpha = \beta$. We have $p_t(-t-2) = 0$; so the general solution to our differential equation is

$$C^{(t)}(z) = (2t+2)! \ln(1-z)/p_t'(-t-2) (1-z)^{t+2} + \sum_{0 \leq j \leq t} c_j (1-z)^{\alpha_j}$$

where $\alpha_0, \dots, \alpha_t$ are the roots of $p_t(x) = 0$, and the constants c_i depend on the initial values C_t, \dots, C_{2t} . The handy identity

$$\frac{1}{(1-z)^{m+1}} \ln\left(\frac{1}{1-z}\right) = \sum_{n \geq 0} (H_{n+m} - H_m) \binom{n+m}{m} z^n, \quad m \geq 0,$$

now leads to the surprisingly simple *closed form solution*

$$C_n = (H_{n+1} - H_{t+1})(n+1)/(H_{2t+2} - H_{t+1}) + \left(\sum_{0 \leq j \leq t} c_j (-\alpha_j)^{\frac{n-t}{\alpha_j}} \right) / n!,$$

from which the asymptotic formula is easily deduced. (The leading term $n \ln n / (H_{2t+2} - H_{t+1})$ was discovered by M. H. van Emden [CACM 13 (1970), 563–567] using an information-theoretic approach. In fact, suppose we wish to analyze any partitioning process such that the left subfile contains at most xN elements with asymptotic probability $\int_0^x f(x) dx$, as $N \rightarrow \infty$, for $0 \leq x \leq 1$; van Emden has proved that the average number of comparisons required to sort the file completely is $\sim n \ln n / \alpha$, where $\alpha = -1/\int_0^1 (f(x) + f(1-x))x \ln x dx$. This formula applies to radix exchange as well as quicksort and various other methods. See also H. Hurwitz, CACM 14 (1971), 99–102.)

30. Each subfile may be identified by four quantities (l, r, k, X) , where l and r are the boundaries (as presently), k indicates the number of words of the keys which are known to be equal throughout the subfile, and X is a lower bound for the $(k+1)$ st words of the keys. Assuming nonnegative keys, we have $(l, r, k, X) = (1, N, 0, 0)$ initially. When partitioning a file, we let K be the $(k+1)$ st word of the test key K_q . If $K > X$, partitioning takes place with all keys $\geq K$ at the right and all keys $< K$ at the left (looking only at the $(k+1)$ st word of the key each time; the partitioned subfiles get the respective identifications $(l, i-1, k, X)$ and (i, r, k, K)). But if $K = X$, partitioning takes place with all keys $> K$ at the right and all keys $\leq K$ [actually $= K$] at the left; the partitioned subfiles get the respective identifications $(l, i, k+1, 0)$ and $(i+1, r, k, K)$. In both cases we are unsure that R_i is in its final position since we haven't looked at the $(k+2)$ nd words. Obvious further changes are made to handle boundary conditions properly. By adding a fifth "upper bound" component, the method could be made symmetrical between left and right.

31. Go through a normal partitioning process, with R_1 finally falling into position R_k . If $k = m$, stop; if $k > m$, use the same technique to find the $(m-k)$ th smallest element of the right-hand subfile; and if $k < m$ find the m th smallest element of the left-hand subfile. [CACM 4 (1961), 321–322.]

32. The recurrence is $C_{nm} = n - 1 + (A_{nm} + B_{nm})/n$, where

$$A_{nm} = \sum_{1 \leq k < m} C_{n-k, m-k} \quad \text{and} \quad B_{nm} = \sum_{m \leq k < n} C_{km},$$

for $1 \leq m \leq n$. Since $A_{n+1, m+1} = A_{nm} + C_{nm}$ and $B_{n+1, m} = B_{nm} + C_{nm}$, we can first find a formula for $(n+1)C_{n+1, m+1} - nC_{nm}$, then sum this to obtain the answer $2((n+1)H_n - (n+3-m)H_{n+1-m} - (m+2)H_m + n+3)$. When $n = 2m-1$,

it becomes $4m(H_{2m-1} - H_m) + 4m - 8H_m + 4 = (4 + 4 \ln 2)m - 8 \ln m - 8\gamma + 1 + O(m^{-1}) \approx 3.39n$.

33. Proceed as in the first stage of radix exchange, using the sign instead of bit 1.
34. We can avoid testing whether or not $i \leq j$, as soon as we have found at least one 0 bit and at least one 1 bit in each stage, i.e., after making the first exchange in each stage. This saves approximately $2C$ units of time in Program R.

35. $A = N - 1$, $B = (\min 0, \text{ave } \frac{1}{4}N \log_2 N, \max \frac{1}{2}N \log_2 N)$, $C = N \log_2 N$, $G = \frac{1}{2}N$, $K = L = R = 0$, $S = \frac{1}{2}N - 1$, $X = (\min 0, \text{ave } \frac{1}{2}(N - 1), \max N - 1)$. [It is interesting to note that, in general, the quantities A, C, G, K, L, R , and S depend only on the set of keys in the file, not on their initial order; only B and X are influenced by the initial order of the keys.]

36. (a) $\sum \binom{n}{k} \binom{k}{j} (-1)^{k+j} a_j = \sum \binom{n}{j} \binom{n-j}{k-j} (-1)^{k-j} a_j = \sum \binom{n}{j} \delta_{nj} a_j = a_n$. (b) $\langle \delta_{n0} \rangle$; $\langle -\delta_{n1} \rangle$; $\langle (-1)^n \delta_{nm} \rangle$; $\langle (1-a)^n \rangle$; $\langle \binom{n}{m} (-a)^m (1-a)^{n-m} \rangle$. (c) Writing the relations to be proved as $x_n = y_n = a_n + z_n$, we have $y_n = a_n + z_n$ by part (a); also $2^{1-n} \sum_{k \geq 2} \binom{n}{k} y_k = z_n$, so y_n satisfies the same recurrence as x_n . [See exercises 53 and 6.3-17 for some generalizations of this result. It does not appear to be easy to prove directly that $\hat{x}_n = a_n 2^{n-1} / (2^{n-1} - 1)$.]

37. $\langle \sum_m c_m \binom{n}{2m} 2^{-n} \rangle$ for an arbitrary sequence of constants c_0, c_1, c_2, \dots . [This answer, although correct, does not reveal immediately that $\langle 1/(n+1) \rangle$ and $\langle n - \delta_{n1} \rangle$ are such sequences! Sequences having the form $\langle a_n + \hat{a}_n \rangle$ are always self-dual. Note that, in terms of the generating function $A(z) = \sum a_n z^n / n!$, we have $\hat{A}(z) = e^z A(-z)$; hence $A = \hat{A}$ is equivalent to saying that $A(z)e^{-z/2}$ is an even function.]

38. A partitioning stage which yields a left subfile of size s and a right subfile of size $N - s$ makes the following contributions to the total running time:

$$A = 1, \quad B = t, \quad C = N, \quad K = \delta_{s1}, \quad L = \delta_{s0}, \quad R = \delta_{sN}, \quad X = h,$$

where t is the number of keys K_1, \dots, K_s with bit b equal to 1, and h is bit b of K_{s+1} ; if $s = N$, then $h = 0$. (Cf. (17).) This leads to recurrence equations such as

$$\begin{aligned} B_N &= 2^{-N} \sum_{0 \leq t \leq s \leq N} \binom{s}{t} \binom{N-s}{t} (t + B_s + B_{N-s}) \\ &= \frac{1}{4}(N-1) + 2^{1-N} \sum_{s \geq 2} \binom{N}{s} B_s, \quad \text{for } N \geq 2; \quad B_0 = B_1 = 0. \end{aligned}$$

(Cf. exercise 23.) Solving these recurrences by the method of exercise 36 yields the formulas $A_N = V_N - U_N + 1$, $B_N = \frac{1}{4}(U_N + N - 1)$, $C_N = V_N + N$, $K_N = N/2$, $L_N = R_N = \frac{1}{2}(V_N - U_N - N) + 1$, $X_N = \frac{1}{2}(A_N - L_N)$. Clearly $G_N = 0$.

39. Each stage of quicksort puts at least one element into its final position, but this need not happen during radix exchange (cf. Table 3).

40. If we switch to straight insertion whenever $r - l < M$ in step R2, the problem doesn't arise unless more than M equal elements occur. If the latter is a likely prospect, we can test whether or not $K_l = \dots = K_r$ whenever $j < l$ or $j = r$ in step R8.

43. As $a \rightarrow 0+$, $\int_0^1 y^{a-1} (e^{-y} - 1) dy + \int_1^\infty y^{a-1} e^{-y} dy = \Gamma(a) - 1/a = (\Gamma(a+1) - \Gamma(1))/a \rightarrow \Gamma'(1) = -\gamma$, by exercise 1.2.7-24.

44. For $k \geq 0$, we have $r_k(m) \sim \frac{1}{2}(2m)^{(k+1)/2}\Gamma((k+1)/2) - \sum_{j \geq 0} (-1)^j B_{k+2j+1}/((k+2j+1)j!(2m)^j)$. When $k = -1$, the contributions from $f_k^{(j-1)}(m)$ in (36) cancel with similar terms in the expansion of H_{m-1} , and we have $r_{-1}(m) = H_{m-1} + (1/\sqrt{2m})\sum_{t \geq 0} f_{-1}(t) \sim \frac{1}{2}(\ln(2m) + \gamma) - \sum_{j \geq 1} (-1)^j B_{2j}/(2j)j!(2m)^j$. Therefore the contribution to W_{m-1} from the term N^t/t of (33) is obtained from $m\sum_{t \geq 1} t^{-1} \exp(-t^2/2m)(1 - t^3/3m^2 + t^6/18m^4)(1 - t^4/4m^3)(1 - t/2m - t^2/8m^2) + O(m^{-1/2}) = \frac{1}{2}m \ln m + \frac{1}{2}(\ln 2 + \gamma)m - \frac{5}{12}\sqrt{2\pi m} + \frac{4}{9} + O(m^{-1/2})$. The term $-\frac{1}{2}N^{t-1}$ contributes $-\frac{1}{2}\sum_{t \geq 1} \exp(-t^2/2m)(1 - t^3/3m^2)(1 - t/2m)(1 + t/m) + O(m^{-1/2}) = -\frac{1}{4}\sqrt{2\pi m} - \frac{1}{6}$. The term $\frac{1}{2}\delta_{t1}$ yields $\frac{1}{2}$. And finally the term $\frac{1}{2}(t-1)B_2 N^{t-2}$ contributes $\frac{1}{12}m^{-1}\sum_{t \geq 1} t \exp(-t^2/2m) + O(m^{-1/2}) = \frac{1}{12} + O(m^{-1/2})$.

45. The argument used to derive (42) is also valid for (43), except that we leave out the residues at $z = -1$ and $z = 0$.

46. Proceeding as we did with (45), we obtain $(s-1)!/\ln 2 + f_s(n)$, where

$$f_s(n) = \frac{2}{\ln 2} \sum_{k \geq 1} \Re(\Gamma(s - 2\pi ik/\ln 2) \exp(2\pi ik \log_2 n)).$$

[Note that $|\Gamma(s+it)|^2 = (\prod_{0 \leq k < s} (k^2 + t^2))\pi/(t \sinh \pi t)$, for integer $s \geq 0$, so we can bound $f_s(n)$.]

47. In fact, $\sum_{j \geq 1} e^{-n/2^j}(n/2^j)^s$ equals the integral in exercise 46, for all $s > 0$.

48. Making use of the intermediate identity

$$1 - e^{-x} = \frac{-1}{2\pi i} \int_{-1/2-i\infty}^{-1/2+i\infty} \Gamma(z)x^{-z} dz,$$

we proceed as in the text, with $1 - e^{-x}$ playing the role of $e^{-x} - 1 + x$; $V_{n+1}/(n+1) = (-1/2\pi i) \int_{-1/2-i\infty}^{-1/2+i\infty} \Gamma(z)n^{-z} dz/(2^{-z} - 1) + O(n^{-1})$, and the integral equals $\log_2 n + \gamma/(\ln 2) - \frac{1}{2} - f_0(n)$ in the notation of exercise 46. [Thus A_N in exercise 38 is $N(1/\ln 2) - f_0(N-1) - f_{-1}(N) + O(1)$.]

49. The right-hand side of Eq. (40) can be improved to $e^{-x}(n/x + \frac{1}{2}x + x^3O(n^{-1}))$. The effect is to subtract $\frac{1}{2}$ times the sum in exercise 47, replacing $O(1)$ in (47) by $2 - \frac{1}{2}(1/\ln 2) + f_1(n) + O(n^{-1})$. (The “2” comes from the “ $2/n$ ” in (45).)

50. $U_{mn} = n \log_m n + n((\gamma - 1)/(\ln m) - \frac{1}{2} + f_{-1}(n)) + m/(m-1) - 1/(2 \ln m) - \frac{1}{2}f_1(n) + O(n^{-1})$, where $f_s(n)$ is defined as in exercise 46 but with $\ln 2$ and \log_2 replaced by $\ln m$ and \log_m . [Note: For $m = 2, 3, 4, 5, 10, 100, 1000, 10^6$ we have $f_{-1}(n) < .0000001725, .00041227, .000296, .00085, .00627, .068, .153, .341$, respectively.]

51. Let $N = 2m$. We may extend the sum (35) over all $t \geq 1$, when it equals

$$\sum_{t \geq 1} (1/2\pi i) \int_{a-i\infty}^{a+i\infty} \Gamma(z)(t^2/N)^{-z} t^k dz = (1/2\pi i) \int_{a-i\infty}^{a+i\infty} \Gamma(z)N^z \zeta(2z-k) dz,$$

provided that $a > (k+1)/2$. So we need to know properties of the zeta function. When $\Re(w) \geq -q$, $\zeta(w) = O(|w|^{q+1})$ as $|w| \rightarrow \infty$; hence we can shift the line of integration to the left as far as we please if we only take the residues into account. The factor $\Gamma(z)$ has poles at $0, -1, -2, \dots$, and $\zeta(2z-k)$ has a pole only at $z = (k+1)/2$. The residue at $z = -j$ is $N^{-j}(-1)^j \zeta(-2j-k)/j!$, and $\zeta(-n) =$

$(-1)^{n+1}B_{n+1}/(n+1)$. The residue at $z = (k+1)/2$ is $\frac{1}{2}\Gamma((k+1)/2)N^{(k+1)/2}$. But when $k = -1$ there is a double pole at $z = 0$; $\zeta(z) = 1/(z-1) + \gamma + O(|z-1|)$, so the residue at 0 in this case is $\gamma + \frac{1}{2}\ln N - \frac{1}{2}\gamma$. We therefore obtain the asymptotic series mentioned in the answer to exercise 44.

52. Set $x = t/n$; then

$$\binom{2n}{n+t} / \binom{2n}{n} = \exp(-2n(x^2/1 \cdot 2 + x^4/3 \cdot 4 + \dots) + (x^2/2 + x^4/4 + \dots) - (1/6n)(x^2 + x^4 + \dots) + \dots);$$

the desired sum can now be expressed in terms of $\sum_{t \geq 1} t^k d(t) e^{-t^2/n}$, for various k . Proceeding as in exercise 51, since $\zeta(z)^2 = \sum_{t \geq 1} d(t)t^{-z}$, we wish to evaluate the residues of $\Gamma(z)n^z\zeta(2z-k)^2$ when $k \geq 0$. At $z = -j$ the residue is $n^{-j}(-1)^j(B_{2j+k+1}/(2j+k+1))^2/j!$, and at $z = (k+1)/2$ the residue is $n^{(k+1)/2}\Gamma((k+1)/2)(\gamma + \frac{1}{4}\ln n + \frac{1}{4}\psi((k+1)/2))$, where $\psi(z) = \Gamma'(z)/\Gamma(z) = H_{z-1} - \gamma$; thus, for example, when $k = 0$, $\sum_{t \geq 1} e^{-t^2/n} d(t) = \frac{1}{4}\sqrt{\pi n} \ln n + (\frac{3}{4}\gamma - \frac{1}{2}\ln 2)\sqrt{\pi n} + \frac{1}{4} + O(n^{-M})$ for all M . For $S_n/(2^n)$, add $(\frac{1}{32}\ln n + \frac{3}{32}\gamma + \frac{1}{24} - \frac{1}{16}\ln 2)\sqrt{\pi/n} + O(n^{-1})$ to this quantity. (Cf. exercise 1.2.7-23, 1.2.9-19.)

53. Let $q = 1 - p$. Generalizing exercise 36(c), if

$$x_n = a_n + \sum_{k \geq 2} \binom{n}{k} (p^k q^{n-k} + q^k p^{n-k}) x_k,$$

then

$$x_n = a_n + \sum_{k \geq 2} \binom{n}{k} (-1)^k a_k (p^k + q^k) / (1 - p^k - q^k).$$

We can therefore find B_N and C_N as before (the factor $\frac{1}{4}$ in B_N should be replaced by pq). The asymptotic examination of U_N proceeds essentially as in the text, with

$$\begin{aligned} T_n &= \sum_{r \geq 1, s \geq 0} \binom{r}{s} (e^{-nr^s q^{r-s}} - 1 + np^s q^{r-s}) \\ &= (1/2\pi i) \int_{-3/2-i\infty}^{-3/2+i\infty} \Gamma(z) n^{-z} (p^{-z} + q^{-z}) dz / (1 - p^{-z} - q^{-z}) \\ &= (n/h_p) (\ln n + \gamma - 1 + h_p^{(2)}/2h_p - h_p + f(n)) + O(1), \end{aligned}$$

where

$$h_p = -(p \ln p + q \ln q), \quad h_p^{(2)} = p(\ln p)^2 + q(\ln q)^2,$$

and

$$f(n) = \sum \Gamma(z) n^{-1-z} / h_p$$

summed over all complex $z \neq -1$ such that $p^{-z} + q^{-z} = 1$. The latter set of points seems to be difficult to analyze in general; but when $p = \phi^{-1}$, $q = \phi^{-2}$, the solutions are $z = (-1)^{k+1} + k\pi i/(\ln \phi)$. The dominant term, $(n \ln n)/h_p$, could also have been obtained from van Emden's general formula quoted in the answer to exercise 29. For $p = \phi^{-1}$ we have $1/h_p = 1.521$, compared to $1/h_{1/2} = 1.4427$.

54. Let C be a circle of radius $(M + \frac{1}{2})b$, so that the integral vanishes on C as $M \rightarrow \infty$. (The asymptotic form of U_n can now be derived in a new way, expanding $\Gamma(n+1)/\Gamma(n+ibm)$. The method of this exercise applies to all sums of the form $\sum_k \binom{n}{k} (-1)^k f(k)$, when f is reasonably well behaved!)

55. Analysis shows that it is actually a little better to use $\lceil(l+r)/2\rceil$ in place of $\lfloor(l+r)/2\rfloor$, because of the way Program Q is written. Thus, we replace lines 09 and 10 of Program Q by

ENTA	0,3	LDA	INPUT,5	JGE	Line13
INCA	1,4	LDX	INPUT,3	LDA	INPUT,5
SRB	1	CMPA	INPUT,4	LDX	INPUT,3
STA	TEMP	JL	5F	CMPA	INPUT,4
LD5	TEMP	STX	INPUT,5	JG	5F
LDA	INPUT,3	JMP	Line19	STX	INPUT,5
CMPA	INPUT,4	5H	LDA	JMP	Line13
JL	1F	STX	INPUT,4	5H	LDA
CMPA	INPUT,5	JMP	Line13	STX	INPUT,4
JLE	Line19	1H	CMPA	JMP	Line21

The first three of these lines should be replaced by "ENTX 0,3; INCX 1,4; ENTA 0; DIV =2=" if binary shifting is not available. In this code, "Line19" refers to line 19 of Program Q, etc.

56. We may solve the recurrence $\binom{n}{3}x_n = b_n + 2\sum_{1 \leq k \leq n} (k-1)(n-k)x_{k-1}$, $n > m$, by letting $y_n = nx_n$, $u_n = ny_{n+1} - (n+2)y_n$, $v_n = nu_{n+1} - (n-5)u_n$; it follows that $v_n = 6(b_{n+2} - 2b_{n+1} + b_n)$, for $n > m$. Example: Let $x_n = \delta_{n1}$ for $n \leq m$, and let $b_n = 0$. Then $v_n = 0$ for all $n > m$, hence $n^5 u_{n+1} = m^5 u_{m+1}$. Since $y_{m+1} = 12/m$, $y_{m+2} = 12/(m+1)$, we ultimately find $x_n = \frac{48}{7}(n+1)/m(m+1)(m+2) + \frac{36}{7}(m-1)^4/n^6$ for $n > m$. In general, when b_n is identically zero, let $f_n = (12/(n-1)(n-2))\sum_{1 \leq k \leq n} (k-1)(n-k)x_{k-1}$; the solution for $n > m$ is $x_n = (n+1)((m+1)f_{m+2} - (m-4)f_{m+1})/7(m+1)(m+2) + ((m+1)f_{m+2} - (m+3)f_{m+1})m^5/7n^6$. When $b_n = \binom{n}{3}/n^2$ and $x_n = 0$ for $n \leq m$, the solution is

$$x_n/(n+1) = (p-3)(p-2)/(p-6)(p+1)(n+1)^{\frac{p+1}{2}} + 12/7(p+1)(m+2)^{\frac{p+1}{2}} - 12(m+1-p)^{\frac{6-p}{2}}/7(p-6)(n+1)^{\frac{7}{2}},$$

for $n > m$; except that when $p = -1$, $x_n/(n+1) = \frac{12}{7}(H_{n+1} - H_{m+2}) + \frac{37}{49} + \frac{12}{49}(m+2)^{\frac{7}{2}}/(n+1)^{\frac{7}{2}}$, and when $p = 6$, $x_n/(n+1) = -\frac{12}{7}(H_{n-6} - H_{m-5})/(n+1)^{\frac{7}{2}} + \frac{12}{49}(m+2)^{\frac{7}{2}} + \frac{37}{49}/(n+1)^{\frac{7}{2}}$.

The above recurrence arises in the following ways in this problem, if we use $\lceil(l+r)/2\rceil$ in place of $\lfloor(l+r)/2\rfloor$ (cf. exercise 55):

Quantity	Value for $n \leq m$	$b_n/\binom{n}{3}$ for $n > m$	Solution for $n > m$.
A_n	0	1	$(n+1)(\frac{12}{7}/(m+2)) - 1$
B_n	0	$\frac{2n-3}{10} + \frac{3}{32} \frac{n(n-2)}{(n-1)(n-3)}$	$(n+1)(\frac{12}{7}(H_{n+1} - H_{m+2}) + \frac{37}{49} - \frac{3}{8}/(m+2) + \dots)$
C_n	0	$n+1$	$(n+1)(\frac{12}{7}(H_{n+1} - H_{m+2}) + \frac{37}{49})$
D_n	$n-1$	0	$(n+1)(1 - \frac{24}{7}/(m+2))$
E_n	$n(n-1)/4$	0	$(n+1)(\frac{48}{35}m - \frac{17}{5} + \frac{6}{7}/(m+2))$
L_n	δ_{n1}	0	$(n+1)(\frac{48}{7}/(m+2)(m+1)m)$
X_n	0	$\frac{1}{2} - \frac{3}{4} \frac{n+1-n}{(n-1)(n-3)}$	$(n+1)(\frac{6}{7}/(m+2) - \dots) - \frac{1}{2}$

Here $\bar{n} = 2\lceil n/2 \rceil$, and the solutions involve further terms of $O(n^{-6})$. The total average running time of the program in exercise 55 is $52\frac{1}{6}A_N + 14B_N + 4C_N + 12D_N + 8E_N - L_N + 8X_N + 15$; $M = 9$ and $M = 10$ are about equally good. With DIV instead of SRB, add $13A_N$ to the running time and take $M = 10$.

57. The asymptotic series for

$$\sum_{n>N} n^{-1} (1 - \alpha/N)^{n-N} = -N^{-1} + \sum_{k \geq 0} (N+k)^{-1} (1 - \alpha/N)^k$$

can be obtained by restricting k to $O(N^{1+\epsilon})$, expanding $(1 - \alpha/N)^k$ as $e^{-\alpha k/N}$ times $(1 - k\alpha^2/2N^2 + \dots)$, and using Euler's summation formula; it comes to $e^\alpha E_1(\alpha) (1 + \alpha^2/2N) - (1 + \alpha)/2N + O(N^{-2})$. Hence the asymptotic value of 5.2.1-11 is $N(\ln \alpha + \gamma + E_1(\alpha))/\alpha + (1 - e^{-\alpha}(1 + \alpha))/2\alpha + O(N^{-1})$. [The coefficient of N is 0.7966, 0.6596, 0.2880, respectively, for $\alpha = 1, 2, 10$.] Note that $\ln \alpha + \gamma + E_1(\alpha) = \int_0^\alpha (1 - e^{-t}) t^{-1} dt$.

SECTION 5.2.3

1. No; but the method using ∞ (described just before Algorithm S) is stable.
2. Traversing a linear list stored sequentially in memory is often slightly faster if we scan the list from higher indices to lower, since it is usually easier to test if an index is zero than to test if it exceeds N . (For the same reason, the search in step S2 runs from j down to 1; but see exercise 8!)
3. (a) The permutation $a_1 \dots a_{N-1} N$ occurs for inputs $N a_2 \dots a_{N-1} a_1, a_1 N a_3 \dots a_{N-1} a_2, \dots, a_1 a_2 \dots a_{N-2} N a_{N-1}, a_1 \dots a_{N-1} N$. (b) The average number of times the maximum is changed during the first iteration of step S2 is $H_N - 1$, as shown in Section 1.2.10. [Hence B_N can be found from Eq. 1.2.7-8.]
4. If the input is a permutation of $\{1, 2, \dots, N\}$, the number of times $i = j$ in step S3 is exactly one less than the number of cycles in the permutation. (Indeed, it is not hard to show that steps S2 and S3 simply remove element j from its cycle; hence S3 is inactive only when j was the smallest element in its cycle.) By Eq. 1.3.3-21 we could save $H_N - 1$ of the $N - 1$ executions of step S3, on the average. Thus it is inefficient to insert an extra test “ $i = j?$ ” before step S3. Instead of testing i vs. j , however, we could lengthen the program for S2 slightly, duplicating part of the code, so that S3 never is encountered if the initial guess K_j is not changed during the search for the maximum; this would make Program S a wee bit faster.
5. $(N - 1) + (N - 3) + \dots = \lfloor N^2/4 \rfloor$.
6. (a) If $i \neq j$ in step S3, that step decreases the number of inversions by $2m - 1$, where m is one more than the number of keys in $K_{i+1} \dots K_{j-1}$ lying between K_i and K_j ; clearly m is not less than the contribution to B on the previous step S2. Now apply the observation of exercise 4, connecting cycles to the condition $i = j$. (b) Every permutation can be obtained from $N \dots 2 1$ by successive interchanges that sort elements that are out of order. (Apply, in reverse sequence, the interchanges that sort the permutation into decreasing order.) Each such operation decreases I by one and changes C by ± 1 . Hence no permutation has a value of $I - C$ which exceeds the corresponding value for $N \dots 2 1$. [By exercise 5 the inequality $B \leq \lfloor N^2/4 \rfloor$ is best possible.]

7. [When $N = 5$ the generating function is $\frac{1}{1-2z}(1 + 4z + 15z^2 + 30z^3 + 37z^4 + 26z^5 + 7z^6)$.]

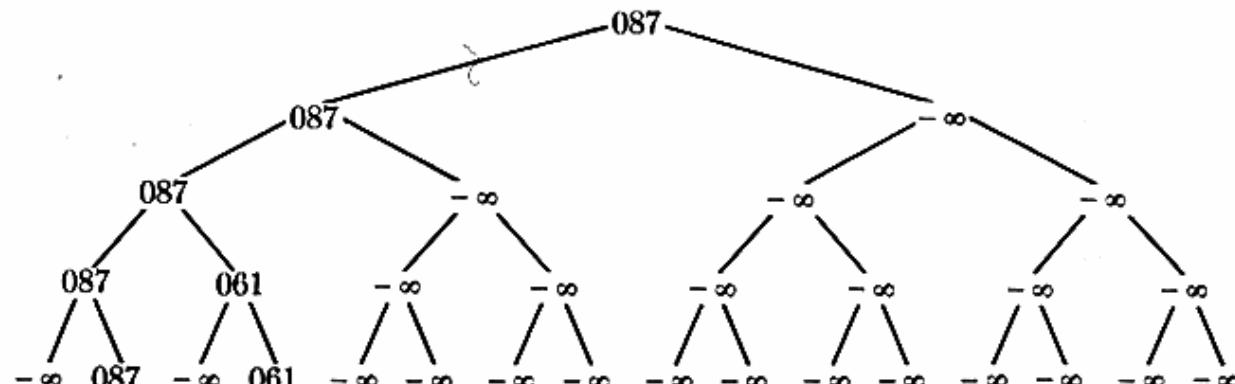
8. We can start the next iteration of step S2 at position K_i , provided that we have remembered $\max(K_1, \dots, K_{i-1})$. One way to keep all of this auxiliary information is to use a link table $L_1 \dots L_N$ such that K_{L_k} is the previous boldface element whenever K_k is boldface; $L_1 = 0$. [We could also use less auxiliary storage, at the expense of some redundant comparisons.]

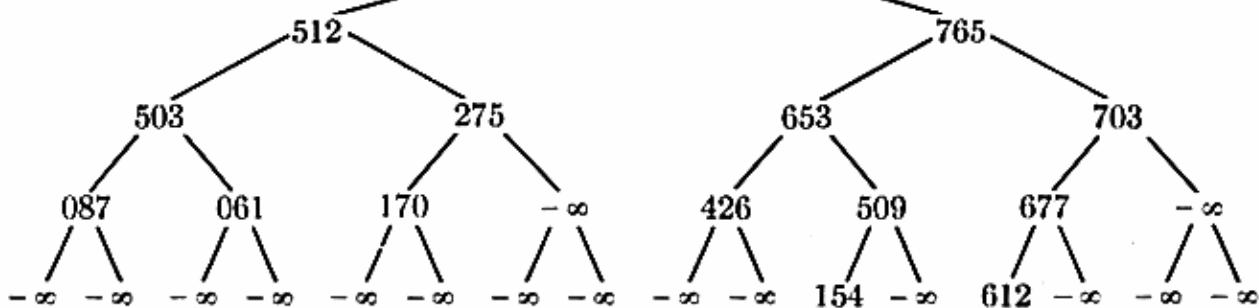
The following MIX program uses address modification so that the inner loop is fast. $rI1 \equiv j$, $rI2 \equiv k - j$, $rI3 \equiv i$, $rA \equiv K_i$.

01	START	ENT1	N	1	$j \leftarrow N$.
02		STZ	LINK+1	1	
03		JMP	9F	1	
04	1H	ST1	6F(0:2)	$N - D$	Modify addresses in loop.
05		ENT4	INPUT,1	$N - D$	
06		ST4	7F(0:2)	$N - D$	
07		ENT4	LINK,1	$N - D$	
08		ST4	8F(0:2)	$N - D$	
09	7H	CMPA	INPUT+J,2	A	[Address modified]
10		JGE	*+4	A	Jump if $K_i \geq K_k$.
11	8H	ST3	LINK+J,2	$N + 1 - C$	Otherwise $L_k \leftarrow i$, [Address modified]
12	6H	ENT3	J,2	$N + 1 - C$	$i \leftarrow k$. [Address modified]
13		LDA	INPUT,3	$N + 1 - C$	
14		INC2	1	A	$k \leftarrow k + 1$
15		J2NP	7B	A	Jump if $k \leq j$.
16	4H	LDX	INPUT,1	N	
17		STX	INPUT,3	N	$R_i \leftarrow R_j$.
18		STA	INPUT,1	N	$R_j \leftarrow \text{former } R_i$.
19		DEC1	1	N	$j \leftarrow j - 1$.
20		ENT2	0,3	N	$rI2 \leftarrow i$.
21		LD3	LINK,3	N	$i \leftarrow L_i$.
22		J3NZ	5F	N	If $i > 0$, k will start at i .
23	9H	ENT3	1	C	Otherwise $i \leftarrow 1$;
24		ENT2	2	C	k will start at 2.
25	5H	DEC2	0,1	$N + 1$	
26		LDA	INPUT,3	$N + 1$	$rA \leftarrow K_i$.
27		J2NP	1B	$N + 1$	Jump if $k \leq j$.
28		J1P	4B	$D + 1$	Jump if $j > 0$. ■

9. $N - 1 + \sum_{N \geq k \geq 2} ((k-1)/2 - 1/k) = \frac{1}{2}\binom{N}{2} + N - H_N$. [The average values of C and D are, respectively, $H_N + 1$ and $H_N - \frac{1}{2}$; hence the average running time of the program is $(1.25N^2 + 31.75N - 15H_N + 14.5)u$.] Program H is much better.

10.





12. $2^n - 1$, once for each “ $-\infty$ ” in a branch node.

13. If $K \geq K_{r+1}$, then step H4 may go to step H5 if $j = r$. (Step H5 is inactive unless $K_r < K_{r+1}$, when step H6 will go to H8 anyway.) To ensure that $K \geq K_{r+1}$ throughout the algorithm, we may start with $K_{N+1} \leq \min(K_1, \dots, K_N)$; instead of setting $R_r \leftarrow R_1$ in step H2, set $R_{r+1} \leftarrow R_{N+1}$ and $R_{N+1} \leftarrow R_1$; also set $R_2 \leftarrow R_{N+1}$ after $r = 1$. (This trick does not speed up the algorithm nor does it make Program H any shorter.)

14. When inserting an element, give it a key which is less (or greater) than all previously assigned keys, to get the effect of a simple queue (or stack, respectively).

15. For efficiency, the following solution is a little bit tricky, avoiding all multiples of 3 [CACM 10 (1967), 570].

a) Set $p[1] \leftarrow 2$, $p[2] \leftarrow 3$, $k \leftarrow 2$, $n \leftarrow 5$, $d \leftarrow 2$, $r \leftarrow 1$, $t \leftarrow 25$, and place (25, 10, 30) in the priority queue. (In this algorithm, $p[i]$ = i th prime; k = number of primes found so far; n = prime candidate; d = distance to next candidate; r = number of elements in the queue; $t = p[r+2]^2$, the next n for which we should increase r . The queue entries have the form $(u, v, 6p)$, where p is the smallest prime divisor of u , $v = 2p$ or $4p$, and $u + v$ is not a multiple of 3.)

b) Let (q, q', q'') be the queue element having the smallest first component. If $n \neq q$, set $k \leftarrow k + 1$, $p[k] \leftarrow n$, $n \leftarrow n + d$, $d \leftarrow 6 - d$, and repeat this step until either $n > N$ or $n = q$.

c) If $n = t$, set $r \leftarrow r + 1$, $u \leftarrow p[r+2]$, $t \leftarrow u^2$ and insert $(t, 2u, 6u)$ or $(t, 4u, 6u)$ into the queue according as $u \bmod 3 = 2$ or $u \bmod 3 = 1$.

d) Replace (q, q', q'') in the queue by $(q + q', q'' - q', q'')$. Set $n \leftarrow n + d$, $d \leftarrow 6 - d$, and return to (b).

Thus the computation begins as follows:

Queue contents	Primes found
(25, 10, 30)	5, 7, 11, 13, 17, 19, 23
(35, 20, 30)(49, 28, 42)	29, 31
(49, 28, 42)(55, 10, 30)	37, 41, 43, 47
(55, 10, 30)(77, 14, 42)(121, 22, 66)	53

Note that the queue entries always have distinct first components. If the queue is maintained as a heap, we can find all primes $\leq N$ in $O(N \log N)$ steps; the length of the heap is at most the number of primes $\leq \sqrt{N}$. The sieve of Eratosthenes (exercise 4.5.4-8) is a $O(N)$ method requiring considerably more random-access storage.

16. Step 1. Set $K \leftarrow$ key to be inserted; $j \leftarrow n + 1$.

Step 2. Set $i \leftarrow \lfloor j/2 \rfloor$.

Step 3. If $i = 0$ or $K_i \geq K$, set $K_j \leftarrow K$ and terminate the algorithm.

Step 4. Set $K_j \leftarrow K_i$, $j \leftarrow i$, and return to step 2.

17. The file 1 2 3 goes into the heap 3 2 1 with Algorithm H, but into 3 1 2 with exercise 16. (*Note:* The latter method of heap creation has a worst case of order $N \log N$; but empirical tests have shown that the number of iterations of step 2 during the creation of a heap is only about $2.27N - 32$, for random input.)

18. (For convenience we assume the presence of an artificial key $K_0 \geq \max(K_1, \dots, K_N)$.) Delete step H6, and replace H8 by:

H8'. [Move back up.] Set $j \leftarrow i$, $i \leftarrow \lfloor j/2 \rfloor$.

H9'. [Does K fit?] If $K \leq K_i$, set $K_j \leftarrow K$ and return to H2. Otherwise set $K_j \leftarrow K_i$ and return to H8'. ■

The method is essentially the same as in exercise 16, but with a different starting place in the heap. The net change to the file is the same as in Algorithm H. Empirical tests on this method show that the number of times $K_j \leftarrow K_i$ occurs per siftup during the selection phase is (0, 1, 2) with respective probabilities (.848, .135, .016). This method makes Program H somewhat longer but improves its asymptotic speed to $13N \log_2 N + O(N)$. An instruction to halve the value of an index register would be desirable.

19. Proceed as in the revised sift-up algorithm of exercise 18, with $K = K_N$, $l = 1$, $r = N - 1$, starting with a given value of j in step H3.

20. For $0 \leq k \leq n$, the number of positive integers $\leq N$ whose binary representation has the form $(b_n \dots b_k a_1 \dots a_q)_2$ for some $q \geq 0$ is clearly $\sum_{0 \leq q < k} 2^q + (b_{k-1} \dots b_0)_2 + 1 = (1b_{k-1} \dots b_0)_2$.

21. Let $j = (c_r \dots c_0)_2$ be in the range $\lfloor N/2^{k+1} \rfloor = (b_n \dots b_{k+1})_2 < j < (b_n \dots b_k)_2 = \lfloor N/2^k \rfloor$. Then s_j is the number of positive integers $\leq N$ whose binary representation has the form $(c_r \dots c_0 a_1 \dots a_q)_2$ for some $q \geq 0$, namely $\sum_{0 \leq q < k} 2^q = 2^{k+1} - 1$. Hence the number of nonspecial subtrees of size $2^{k+1} - 1$ is

$$\lfloor N/2^k \rfloor - \lfloor N/2^{k+1} \rfloor - 1 = \lfloor (N - 2^k)/2^{k+1} \rfloor.$$

[For the latter identity, use the replicative law in exercise 1.2.4-38 with $n = 2$ and $x = N/2^{k+1}$.]

22. Before $l = 1$ the five possibilities are 5 3 4 1 2, 3 5 4 1 2, 4 3 5 1 2, 1 5 4 3 2, 2 5 4 1 3. Each of these possibilities $a_1 a_2 a_3 a_4 a_5$ leads to three possible permutations $a_1 a_2 a_3 a_4 a_5$, $a_1 a_4 a_3 a_2 a_5$, $a_1 a_5 a_3 a_4 a_2$ before $l = 2$.

23. (a) After B iterations, $j \geq 2^{Bl}$; hence $2^{Bl} \leq r$. (b) $\sum_{1 \leq l \leq N} \lfloor \log_2 (N/l) \rfloor = (\lfloor N/2 \rfloor - \lfloor N/4 \rfloor) + 2(\lfloor N/4 \rfloor - \lfloor N/8 \rfloor) + 3(\lfloor N/8 \rfloor - \lfloor N/16 \rfloor) + \dots = \lfloor N/2 \rfloor + \lfloor N/4 \rfloor + \lfloor N/8 \rfloor + \dots = N - \nu(N)$, where $\nu(N)$ is the number of ones in the binary representation of N . Also by exercise 1.2.4-42 we have $\sum_{1 \leq r < N} \lfloor \log_2 r \rfloor = N \lfloor \log_2 N \rfloor - 2^{\lfloor \log_2 N \rfloor + 1} + 2$. We know by Theorem H that this upper bound on B is best possible during the heap creation phase. Furthermore it is interesting to note that there is a unique heap containing the keys $\{1, 2, \dots, N\}$ such that K is identically equal to 1 throughout the selection phase of Algorithm H. (For example, when $N = 7$ that heap is 7 5 6 2 4 3 1; it is not difficult to pass from N to $N + 1$.) This heap gives the maximum value of B (as well as the maximum value $\lfloor N/2 \rfloor$ of D) for the selection phase of heapsort, so the best possible upper bound on B for the entire sort is $N - \nu(N) + N \lfloor \log_2 N \rfloor - 2^{\lfloor \log_2 N \rfloor + 1} + 2$.

24. $\sum_{1 \leq k \leq N} \lfloor \log_2 k \rfloor^2 = (N + 1 - 2^n)n^2 + \sum_{0 \leq k < n} k^2 2^k = (N + 1)n^2 - 6 -$

$(2n - 3)2^{n+1}$, where $n = \lfloor \log_2 N \rfloor$ (cf. exercise 4.5.2-23); hence the variance of the last sift-up is $\beta_N = ((N + 1)n^2 - (2n - 3)2^{n+1} - 6)/N = ((N + 1)n + 2 - 2^{n+1})^2/N^2 = O(1)$. The standard deviation of B'_N is $(\sum_{s \in M_N} \beta_s)^{1/2} = O(\sqrt{N})$.

25. The sift-up is “uniform,” and each comparison $K_j:K_{j+1}$ has probability $\frac{1}{2}$ of coming out $<$. The average contribution to C in this case is just one-half the sum of the average contributions to A and B , namely $((2n - 3)2^{n-1} + \frac{1}{2})/(2^{n+1} - 1)$.

26. (a) $(\frac{10}{25} + \frac{1}{2} + 1\frac{3}{9} + \frac{1}{2} + 1\frac{1}{2} + 1\frac{2}{5} + 2\frac{1}{2} + \frac{1}{2} + 1\frac{1}{2} + 2\frac{1}{2} + 1\frac{1}{2} + 2 + 2 + 3 + 0 + 1 + 1 + 2 + 1 + 2 + 2 + 3 + 1 + 2 + 2)/26$. (b) $(\sum_{1 \leq k \leq N} \nu(k) - N + \frac{1}{2}\lfloor N/2 \rfloor - \frac{1}{2}n + \sum_{1 \leq k < n} \min(\alpha_{k-1}, \alpha_k - \alpha_{k-1} - 1)/(\alpha_k - 1))/N$, where $\nu(k)$ is the number of one bits in the binary representation of k , and $\alpha_k = (1b_k \dots b_0)_2$. If $N = 2^{e_1} + 2^{e_2} + \dots + 2^{e_t}$, with $e_1 > e_2 > \dots > e_t \geq 0$, it can be shown that $\sum_{0 \leq k \leq N} \nu(k) = \frac{1}{2}((e_1 + 2)2^{e_1} + (e_2 + 4)2^{e_2} + \dots + (e_t + 2t)2^{e_t}) + t - N$.

27. In general, the Lambert series $\sum_{n \geq 1} a_n x^n / (1 - x^n) = \sum_{N \geq 1} (\sum_{d \mid n} a_d) x^N = \sum_{m \geq 1} x^{m^2} (a_m + \sum_{k \geq 1} (a_m + a_{m+k}) x^{km})$.

[Note that when $a_n = n$ and $x = \frac{1}{2}$ we obtain the relation

$$\beta = \sum_{n \geq 1} \frac{1}{2^n - 1} = \sum_{n \geq 1} 2^{-n^2} \left(n \left(\frac{2^n + 1}{2^n - 1} \right) + \frac{2^n}{(2^n - 1)^2} \right) = 2.74403 38887 59488 \dots ;$$

this constant arises in (20), where we have $B'_N \sim (\beta - 2)N$ and $C'_N \sim (\frac{1}{2}\beta - \frac{1}{4}\alpha - \frac{1}{2})N$.]

28. The sons of node k are nodes $3k - 1$, $3k$, and $3k + 1$. A MIX program analogous to Program H will take asymptotically $21\frac{2}{3}N \log_3 N \approx 13.7N \log_2 N$ units of time. Using the idea of exercise 18 lowers this to $18\frac{2}{3}N \log_3 N \approx 11.8N \log_2 N$, although the division by 3 will add a large $O(N)$ term.

31. Let A , B be arrays of n elements each such that $A[\lfloor i/2 \rfloor] \leq A[i]$ and $B[i] \leq B[\lfloor i/2 \rfloor]$ for $1 < i \leq n$; furthermore we require that $A[i] \leq B[i]$ for $1 \leq i \leq n$. (The latter condition holds for all i iff it holds for $n/2 < i \leq n$, because of the heap structure.) This “twin-heap” contains $2n$ elements; to handle an odd number of elements, we simply keep one element off to the side. Appropriate modifications of the other algorithms in this section can be used to maintain twin heaps, and it is interesting to work out the details.

32. Let P , Q point to the given priority queues; for brevity, the following algorithm uses the convention $\text{DIST}(\Lambda) = 0$.

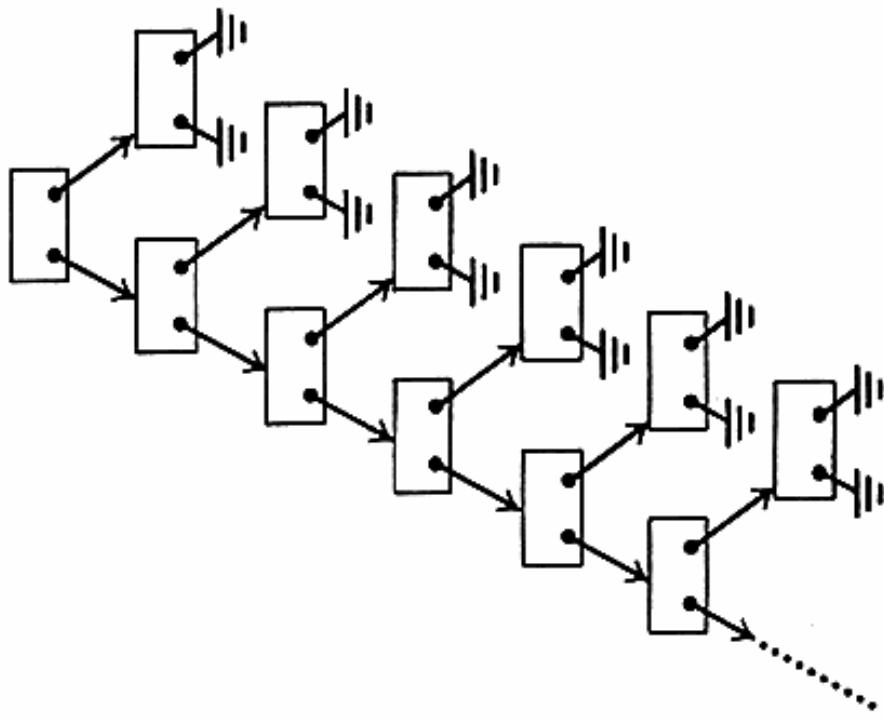
M1. [Initialize.] Set $R \leftarrow \Lambda$.

M2. [List merge.] If $Q = \Lambda$, set $D \leftarrow \text{DIST}(P)$ and go to M3. If $P = \Lambda$, set $P \leftarrow Q$, $D \leftarrow \text{DIST}(P)$, and go to M3. Otherwise if $\text{KEY}(P) \geq \text{KEY}(Q)$, set $T \leftarrow \text{RIGHT}(P)$, $\text{RIGHT}(P) \leftarrow R$, $R \leftarrow P$, $P \leftarrow T$ and repeat step M2. If $\text{KEY}(P) > \text{KEY}(Q)$, set $T \leftarrow \text{RIGHT}(Q)$, $\text{RIGHT}(Q) \leftarrow R$, $R \leftarrow Q$, $Q \leftarrow T$ and repeat step M2. (This step essentially merges the two “right lists” of the given trees, temporarily inserting upward pointers into the **RIGHT** fields.)

M3. [Done?] If $R = \Lambda$, terminate the algorithm; P points to the answer.

M4. [Fix DISTs.] Set $Q \leftarrow \text{RIGHT}(R)$. If $\text{DIST}(\text{LEFT}(R)) < D$, then set $D \leftarrow \text{DIST}(\text{LEFT}(R)) + 1$, $\text{RIGHT}(R) \leftarrow \text{LEFT}(R)$, $\text{LEFT}(R) \leftarrow P$; otherwise set $D \leftarrow D + 1$, $\text{RIGHT}(R) \leftarrow P$. Finally set $\text{DIST}(R) \leftarrow D$, $R \leftarrow Q$, $P \leftarrow R$, and return to M3. ■

33. We might have to promote about half the nodes in a lop-sided tree such as



34. Let the number be L_{N+1} . It is not difficult to prove that $L_{m+n} \leq L_m L_n$, hence (taking logarithms) $\lim L_n^{1/n}$ exists. (This suggestion is due to D. A. Klarner. Some calculations by E. Logg suggest that $L_{n+1}/L_n \geq L_n/L_{n-1}$ may hold for all n .)

35. Let the DIST field of the deleted node be d_0 , and let the DIST field of the merged subtrees be d_1 . If $d_0 = d_1$, we need not go up at all. If $d_0 > d_1$, and if we go up n levels, the new DIST fields of the ancestors of P must be, respectively, $d_1 + 1, d_1 + 2, \dots, d_1 + n$. If $d_0 < d_1$, the upward path must go only leftwards.

36. Instead of a general priority queue, it is simplest to use a doubly-linked list; when “using” a node, move it to one end of the list, and delete nodes from the other end. [See the discussion of “self-organizing files” in Section 6.1.]

SECTION 5.2.4

1. Start with $i_1 = \dots = i_k = 1$, $j = 1$. Repeatedly find $\min(x_{1i_1}, \dots, x_{ki_k}) = x_{ri_r}$ and set $z_j \leftarrow x_{ri_r}$, $j \leftarrow j + 1$, $i_r \leftarrow i_r + 1$. (In this case the use of $x_{i(m_i+1)} = \infty$ is a decided convenience.)

When k is moderately large, it is desirable to keep the keys $x_{1i_1}, \dots, x_{ki_k}$ in a tree structure suited to repeated selection, as discussed in Section 5.2.3, so that only $\lfloor \log_2 k \rfloor$ comparisons are needed to find the new minimum each time after the first. Indeed this is a typical application of the principle of “smallest in, first out” in a priority queue. The keys can be maintained as a heap, and ∞ can be avoided entirely. See the further discussion in Section 5.4.1.

2. Let C be the number of comparisons; we have $C = m + n - S$, where S is the number of elements transmitted in step M4 or M6. The probability that $S \geq s$ is easily seen to be

$$q_s = \left(\binom{m+n-s}{m} + \binom{m+n-s}{n} \right) / \binom{m+n}{m}$$

for $1 \leq s \leq m+n$; $q_s = 0$ for $s > m+n$. Hence the mean of S is $\mu_{mn} = q_1 + q_2 + \dots = m/(n+1) + n/(m+1)$ [cf. exercises 3.4.2–5, 6], and the variance is $\sigma_{mn}^2 = (q_1 + 3q_2 + 5q_3 + \dots) - \mu_{mn}^2 = m(2m+n)/(n+1)(n+2) + (m+2n)n/(m+1)(m+2) - \mu_{mn}^2$. Thus

$$C = (\min \min(m, n), \quad \text{ave } m+n - \mu_{mn}, \quad \max m+n - 1, \quad \text{dev } \sigma_{mn}).$$

When $m = n$ the average was first computed by H. Nagler, *CACM* 3(1960), 618–620; it is asymptotically $2n - 2 + O(n^{-1})$, with a standard deviation of $\sqrt{2} + O(n^{-1})$. Thus C hovers close to its maximum value.

3. M2'. If $K_i < K'_j$, go to M3'; if $K_i = K'_j$, go to M7'; if $K_i > K'_j$, go to M5'.

M7'. Set $K''_k \leftarrow K'_j$, $k \leftarrow k+1$, $i \leftarrow i+1$, $j \leftarrow j+1$. If $i > M$, go to M4'; otherwise if $j > N$, go to M6'; otherwise return to M2'. ■

(Appropriate modifications are made to other steps of Algorithm M. Again many special cases disappear if we insert artificial keys $K_{M+1} = K'_{N+1} = \infty$ at the end of the files.)

4. The sequence of elements which appears at a fixed internal node of the selection tree, as time passes, is obtained by merging the sequences of elements which appear at the sons of that node. (The discussion in Section 5.2.3 is based on selecting the *largest* element, but it could equally well have reversed the order.) So the operations involved in tree selection are essentially the same as those involved in merging, but they are performed in a different sequence and using different data structures.

Another relation between merging and tree selection is indicated in exercise 1. Note that an N -way merge of one-element files is a selection sort; compare also four-way merging of (A, B, C, D) to two-way merging of (A, B) , (C, D) , then (AB, CD) .

5. In step N6 we always have $K_i < K_{i-1} \leq K_j$; in N10, $K_j < K_{j+1} < K_i$.

6. 2 6 4 10 8 14 12 16 15 11 13 7 9 3 5 1. After one pass we have 1 2 5 6 7 8 13 14 16 15 12 11 10 9 4 3 (two of the expected stepdowns disappear). This possibility was noted by D. A. Bell, *Comp. J.* 1 (1958), 74. Quirks like this make it almost hopeless to carry out a precise analysis of Algorithm N.

7. $\lceil \log_2 N \rceil$, if $N > 1$. (Consider how many times p must be doubled until it is $\geq N$.)

8. If N is not a multiple of $2p$, there is one short run on the pass, and it is always near the middle; letting its length be t , we have $0 \leq t < p$, and the situation is essentially $x_1 \leq x_2 \leq \dots \leq x_p \mid y_t \geq \dots \geq y_1$. If $x_p \leq y_t$, the left-hand run is exhausted first, and step S6 will take us to S13 after x_p has been transmitted. On the other hand, if $t = 0$ or $x_p > y_t$, the right-hand side will be artificially exhausted, but $K_j = x_p$ will never be $< K_i$ in step S3! Thus S6 will eventually take us to S13 in all cases.

10. For example, Algorithm M can merge elements $x_{j+1} \dots x_{j+m}$ with $x_{j+m+1} \dots x_{j+m+n}$ into positions $x_1 \dots x_{m+n}$ of an array without conflict, if $j \geq n$. With care we can exploit this idea so that $N + 2^{\lceil \log_2 N \rceil - 1}$ locations are required for an entire sort. But the program seems to be rather complicated compared to Algorithm S. [*Comp. J.* 1 (1958), 75; see also L. S. Lozinskii, *Kibernetika* 1, 3 (1965), 58–62.]

11. Yes. This can be seen, for example, by considering the relation to tree selection mentioned in exercise 4. But Algorithms N and S are obviously not stable.

12. Set $L_0 \leftarrow 1$, $t \leftarrow N + 1$; then for $p = 1, 2, \dots, N - 1$, do the following:

If $K_p \leq K_{p+1}$ set $L_p \leftarrow p + 1$; otherwise set $L_t \leftarrow -(p + 1)$, $t \leftarrow p$.

Finally set $L_t \leftarrow 0$, $L_N \leftarrow 0$, $L_{N+1} \leftarrow |L_{N+1}|$.

(Stability is preserved. The number of passes is $\lceil \log_2 r \rceil$, where r is the number of ascending runs in the input; the exact distribution of r is analyzed in Section 5.1.3. We may conclude that “natural” merging is preferable to “straight” merging when linked allocation is being used, although it was inferior for sequential allocation.)

13. The running time for $N \geq 3$ is $(11A + 6B + 3B' + 9C + 2C'' + 4D + 5N + 9)u$, where A is the number of passes; $B = B' + B''$ is the number of subfile-merge operations performed, where B' is the number of such merges in which the p subfile was exhausted first; $C = C' + C''$ is the number of comparisons performed, where C' is the number of such comparisons with $K_p \leq K_q$; $D = D' + D''$ is the number of elements remaining in subfiles when the other subfile has been exhausted, where D' is the number of such elements belonging to the q subfile.

Algorithm L does a sequence of merges on subfiles whose sizes (m, n) can be determined as follows: Let $N - 1 = (b_k \dots b_1 b_0)_2$ in binary notation. There are $(b_k \dots b_{j+1})_2$ “ordinary” merges with $(m, n) = (2^j, 2^j)$, for $0 \leq j < k$; and there are “special” merges with $(m, n) = (2^j, 1 + (b_{j-1} \dots b_0)_2)$ whenever $b_j = 1$, for $0 \leq j \leq k$. For example when $N = 14$, there are six ordinary $(1, 1)$ merges, three ordinary $(2, 2)$ merges, one ordinary $(4, 4)$ merge, and the special merges deal with subfiles of sizes $(1, 1)$, $(4, 2)$, $(8, 6)$.

It follows that, regardless of the input distribution, we have $A = \lceil \log_2 N \rceil$, $B = N - 1$, $C' + D'' = \sum_{0 \leq j \leq k} b_j 2^j (1 + \frac{1}{2}j)$, $C'' + D' = \sum_{0 \leq j \leq k} b_j (1 + 2^j (\frac{1}{2}j + b_{j+1} + \dots + b_k))$; hence only B' , C' , D' need to be analyzed further.

If the input to Algorithm L is random, each of the merging operations satisfies the conditions of exercise 2, and is independent of the behavior of the other merges; so the distribution of B' , C' , D' is the convolution of their individual distributions for each subfile merge. The average values for such a merge are $B' = n/(m+n)$, $C' = mn/(n+1)$, $D' = n/(m+1)$. Sum these over all relevant (m, n) to get the exact average values.

When $N = 2^k$ we have, of course, the simplest situation; $B'_{\text{ave}} = \frac{1}{2}B$, $C'_{\text{ave}} = \frac{1}{2}C_{\text{ave}}$, $C + D = kN$, and $D_{\text{ave}} = \sum_{1 \leq j \leq k} (2^{k-j} \cdot 2^j / (2^{j-1} + 1)) = \alpha'N + O(1)$, where $\alpha' = \sum_{n \geq 0} 1/(2^n + 1) = \alpha + \frac{1}{2} - 2 \sum_{n \geq 1} 1/(4^n - 1) = 1.2644997803484442091913197472554984825577$ —can be evaluated to high precision as in exercise 5.2.3–27. This special case was first analyzed by A. Gleason [unpublished, 1956] and H. Nagler [CACM 3 (1960), 618–620].

In Table 3 we have $A = 4$, $B' = 6$, $B'' = 9$, $C' = 22$, $C'' = 22$, $D' = 10$, $D'' = 10$, total time = $761u$. (The comparable Program 5.2.1L takes only $433u$, when improved as in exercise 5.2.1–33, so we can see that merging isn’t especially efficient when N is small.)

14. Set $D = B$ in exercise 13 to maximize C .

15. Make extra copies of steps L3, L4, L6 for the cases that L_s is known to equal p or q . [A further improvement can also be made, removing the assignment $s \leftarrow p$ (or $s \leftarrow q$) from the inner loop, by simply renaming the registers! For example, change lines 20, 21 to “LD3 INPUT,1(L)” and continue with p in rI3, s in rI1 and L_s known

to equal p . With twelve copies of the inner loop, corresponding to the different permutations of (p, q, r) with respect to $(rI1, rI2, rI3)$, and the different knowledge about L_s , we can cut the average running time to $8N \log_2 N + O(N)$.)

16. (The result will be slightly faster than Algorithm L, cf. exercise 5.2.3-28.)
17. Consider the new record as a subfile of length 1. Repeatedly merge the smallest two subfiles if they have the same length. (The resulting sorting algorithm is essentially the same as Algorithm L, but the subfiles are merged at different relative times.)
18. Yes, but it seems to be a complicated job. The simplest known method uses the following ingenious construction [*Dokladi Akad. Nauk SSSR* 186 (1969), 1256-1258]: Let n be $\approx \sqrt{N}$. Divide the file into $m+2$ "zones" $Z_1 \dots Z_m Z_{m+1} Z_{m+2}$, where Z_{m+2} contains $(N \bmod n)$ records while each other zone contains exactly n records. Interchange the records of Z_{m+1} with the zone containing R_M ; the file now takes the form $Z_1 \dots Z_m A$, where each of $Z_1 \dots Z_m$ contains exactly n records in order and where A is an auxiliary area containing s records, for some s , where $n \leq s < 2n$.

Find the zone with smallest leading element, and exchange that entire zone with Z_1 . (This takes $O(m+n)$ operations.) Then find the zone with next smallest leading element, and exchange it with Z_2 , etc. Finally in $O(m(m+n)) = O(N)$ operations we will have rearranged the m zones so that their leading elements are in order. Furthermore, because of our original assumptions about the file, each of the keys in $Z_1 \dots Z_m$ now has less than n inversions.

We can merge Z_1 with Z_2 , using the following trick: Interchange Z_1 with the first n elements A' of A ; then merge Z_2 with A' in the usual way but exchanging elements with the elements of $Z_1 Z_2$ as they are output. For example, if $n = 3$ and $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$, we have

	Zone 1	Zone 2	Auxiliary
Initial contents:	$x_1 \ x_2 \ x_3 \ y_1 \ y_2 \ y_3$		$a_1 \ a_2 \ a_3$
Exchange Z_1 :	$a_1 \ a_2 \ a_3 \ y_1 \ y_2 \ y_3$	$x_1 \ x_2 \ x_3$	
Exchange x_1 :	$x_1 \ a_2 \ a_3 \ y_1 \ y_2 \ y_3$	$a_1 \ x_2 \ x_3$	
Exchange y_1 :	$x_1 \ y_1 \ a_3 \ a_2 \ y_2 \ y_3$	$a_1 \ x_2 \ x_3$	
Exchange x_2 :	$x_1 \ y_1 \ x_2 \ a_2 \ y_2 \ y_3$	$a_1 \ a_3 \ x_3$	
Exchange y_2 :	$x_1 \ y_1 \ x_2 \ y_2 \ a_2 \ y_3$	$a_1 \ a_3 \ x_3$	
Exchange x_3 :	$x_1 \ y_1 \ x_2 \ y_2 \ x_3 \ y_3$	$a_1 \ a_3 \ a_2$	

(The merge is always complete when the n th element of the auxiliary area has been exchanged; this method generally permutes the auxiliary records.)

The above trick is used to merge Z_1 with Z_2 , then Z_2 with Z_3, \dots, Z_{m-1} with Z_m , requiring a total of $O(mn) = O(N)$ operations. Since no element has more than n inversions, the $Z_1 \dots Z_m$ portion of the file has been completely sorted.

For the final "cleanup," we sort $R_{N+1-2s} \dots R_N$ by insertion, in $O(s^2) = O(N)$ steps; this brings the s largest elements into area A . Then we merge $R_1 \dots R_{N-2s}$ with $R_{N+1-2s} \dots R_{N-s}$, using the above trick with auxiliary storage area A (but interchanging the roles of right and left, less and greater, throughout). Finally, we sort $R_{N+1-s} \dots R_N$ by insertion.

19. We may number the input cars so that the final permutation has them in order, $1 \ 2 \ \dots \ 2^n$; so this is essentially a sorting problem. First move the first 2^{n-1} cars through $n - 1$ stacks, putting them in decreasing order, and transfer them to the n th stack so that the smallest is on top. Then move the other 2^{n-1} cars through $n - 1$ stacks, putting them into increasing order and leaving them positioned just before the n th stack. Finally, merge the two sequences together in the obvious way.
20. For further information, see R. E. Tarjan, *J ACM* 19 (1972), 341–346.

SECTION 5.2.5

1. No, because radix sorting doesn't work at all unless the distribution sorting is stable, after the first pass. (But the suggested distribution sort *could* be used in a most-significant-digit-first radix sorting method, generalizing radix exchange, as suggested in the last paragraph of the text.)
2. It is "anti-stable," just the opposite; elements with equal keys appear in reverse order, since the first pass goes through the records from R_N to R_1 . (This proves to be convenient because of lines 28 and 20 of Program R , equating Λ with 0, but of course it is not necessary to make the first pass go backwards.)
3. If pile 0 is not empty, $\text{BOTM}[0]$ already points to the first element; if it is empty, we set $P \leftarrow \text{LOC}(\text{BOTM}[0])$ and later make $\text{LINK}(P)$ point to the bottom of the first nonempty pile.
4. When there are an even number of passes remaining, take pile 0 first (top to bottom), followed by pile 1, ..., pile $(M - 1)$; the result will be in order with respect to the digits examined so far. When there are an odd number of passes remaining, take pile $(M - 1)$ first, then pile $(M - 2)$, ..., pile 0; the result will be in *reverse* order with respect to the digits examined so far. (This rule was apparently first published by E. H. Friend [*JACM* 3 (1956), 156, 165–166]. See also an article by Donald W. Johnson [*Library Resources and Technical Services* 3 (1959), 300–310], who proposed radix sorting as a manual aid for putting library cards in order. Johnson says the University of California library at Berkeley found that radix sorting made it possible to alphabetize 2000 cards by hand in only 4.8 hours instead of 8.5 hours by previous methods.)
5. (a) There are k empty piles after the $(N + 1)$ st element is placed iff (i) the $(N + 1)$ st element falls into a previously empty pile, with probability $((k + 1)/M) \times p_{MN(k+1)}$; or (ii) it falls into a nonempty pile, with probability $((M - k)/M)p_{MNk}$. The recurrence

$$p_{M,N+1,k} = \frac{k+1}{M} p_{M,N,k+1} + \frac{M-k}{M} p_{M,N,k}$$

is equivalent to the stated formula. (b) The n th derivative satisfies $g_{M,N+1}^{(n)}(z) = (1 - n/M)g_{M,N}^{(n)}(z) + ((1 - z)/M)g_{M,N}^{(n+1)}(z)$, by induction on n . Setting $z = 1$, we find $g_{M,N}^{(n)}(1) = (1 - n/M)^NM^n$, since $g_{M0}(z) = z^M$. Hence $\text{mean}(g_{MN}) = (1 - 1/M)^NM$, $\text{var}(g_{MN}) = (1 - 2/M)^NM(M - 1) + (1 - 1/M)^NM - (1 - 1/M)^{2N}M^2$. (Note that the generating function for E in Program R is $(g_{MN}(z))^p$.)

6. Change line 04 to "ENT3 7", and change the R3SW and R5SW tables to:

R3SW	LD2	KEY,1(1:1)
	LD2	KEY,1(2:2)
	LD2	KEY,1(3:3)
	LD2	KEY,1(4:4)
	LD2	KEY,1(5:5)
	LD2	INPUT,1(1:1)
	LD2	INPUT,1(2:2)
	LD2	INPUT,1(3:3)
R5SW	LD1	INPUT,1(LINK)
	...	(repeat previous line six more times)
	DEC1	1

The new running time is found by changing "3" to "8" everywhere; it amounts to $(11p - 1)N + 16pM + 12p - 4E + 2$, for $p = 8$.

7. Let R = radix sort, RX = radix exchange. Some of the important similarities and differences: RX goes from most significant digit to least significant, while R goes the other way. Both methods sort by digit inspections, without making comparisons of keys. RX always has $M = 2$ (but see exercise 1). The running time for R is almost unvarying, while RX is sensitive to the distribution of the digits. In both cases the running time is $O(N \log K)$, where K is the range of keys, but the constant of proportionality is higher for RX; on the other hand, when the keys are uniformly distributed in their leading digits, RX has an average running time of $O(N \log N)$ regardless of the size of K . R requires link fields while RX runs in "minimal storage." The inner loop of R is more suited to "pipeline" computers.

8. On the final pass, the piles should be hooked together in another order; for example, if $M = 256$, pile $(10000000)_2$ comes first, then pile $(10000001)_2, \dots$, pile $(11111111)_2$, pile $(00000000)_2$, pile $(00000001)_2, \dots$, pile $(01111111)_2$. This change in hooking order can be done easily by modifying Algorithm H, or (in Table 1) by changing the storage allocation strategy, on the last pass.

9. We could first separate the negative keys from the positive keys, as in exercise 5.2.2-33; or we could change the keys to complement notation on the first pass. Alternatively, after the last pass we could separate the positive keys from the negative ones, reversing the order of the latter, although the method of exercise 5.2.2-33 no longer applies.

11. Without the first pass the method would still sort perfectly, because (by coincidence) 503 already precedes 509. Without the first two passes, the number of inversions would be $1 + 1 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 = 5$.

12. After exchanging R_k with $R[P]$ in step M4 (exercise 5.2-12), we can compare K_k to K_{k-1} . If K_k is less, we compare it to K_{k-2}, K_{k-3}, \dots , until finding $K_k \geq K_j$. Then set $(R_{j+1}, \dots, R_{k-1}, R_k) \leftarrow (R_k, R_{j+1}, \dots, R_{k-1})$, without changing the LINK fields. It is convenient to place an artificial key K_0 , which is \leq all other keys, at the left of the file.

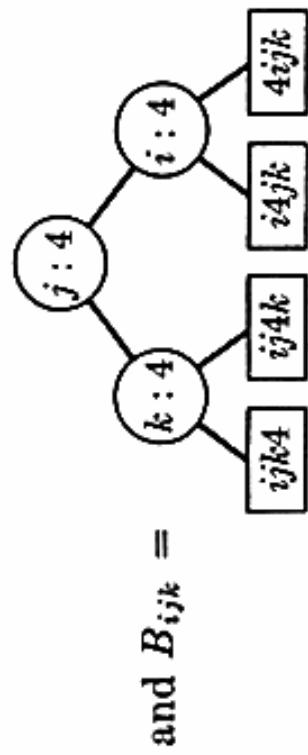
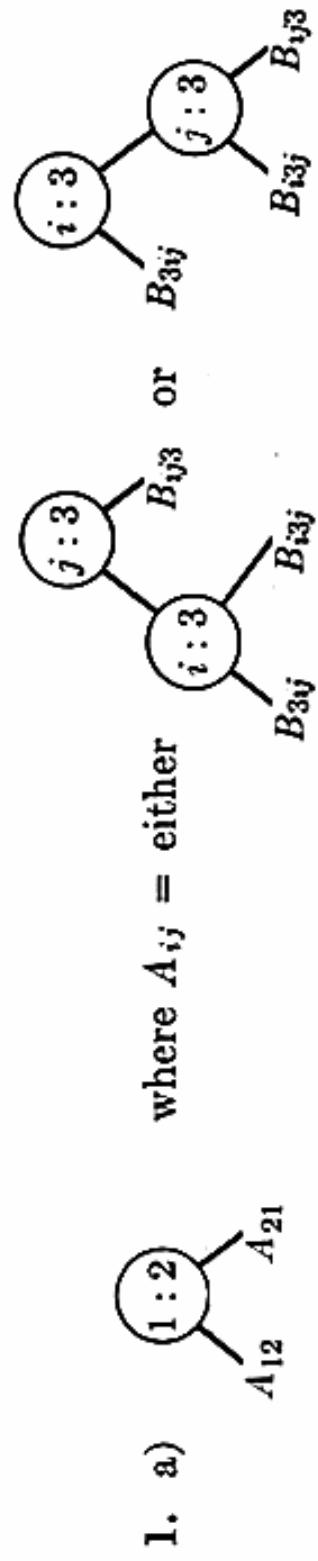
14. If the original permutation of the cards requires k readings, in the sense of exercise 5.1.3-20, and if we use m piles per pass, we must make at least $\lceil \log_m k \rceil$ passes. (Consider going back from a sorted deck to the original one; the number of readings increases

by at most a factor of m on each pass.) The given permutation requires 4 increasing readings, 10 decreasing readings, so decreasing order requires 4 passes with two piles or 3 passes with three piles.

Conversely, this optimum number of passes can be achieved: Number the cards from 0 to $k - 1$ according to which reading it belongs to, and use a radix sort (least significant digit first in radix m). [Cf. *Martin Gardner's Sixth Book of Mathematical Games* (San Francisco: W. H. Freeman, 1971), 111–112.]

15. Let there be k readings and m piles. The order is reversed on each pass; if there are k readings in one order, the number of readings in the opposite order is $n + 1 - k$. The minimum number of passes is either the smallest even number greater than or equal to $\log_m k$ or the smallest odd number greater than or equal to $\log_m (n + 1 - k)$. (Going backwards, there are at most m decreasing readings after one pass, m^2 increasing readings after two passes, etc.) The example can be sorted into increasing order in $\min(2, 5) = 2$ passes, into decreasing order in $\min(3, 4) = 3$ passes, using only two piles.

SECTION 5.3.1



External path length is 112 (optimum).



External path length is 112 (optimum).

2. In the notation of exercise 5.2.4-14, $L(n) - B(n) = \sum_{1 \leq k \leq t} ((e_k + k - 1)2^{ek} - (e_1 + 1)2^{ek}) + 2^{e_1+1} - 2^{e_1} = 2^{e_1} - 2^{e_1} - \sum_{2 \leq k \leq t} (e_1 - e_k + 2 - k)2^{ek} \geq 2^{e_1} -$

$(2^{e_1-1} + \cdots + 2^{e_1-i+1} + 2^{e_i}) \geq 0$, with equality iff $n = 2^k - 2^i$ for some $k > j \geq 0$.

3. When $n > 0$, the number of outcomes such that the smallest key appears exactly k times is $\binom{n}{k} P_{n-k}$. Thus $2P_n = \sum_k \binom{n}{k} P_{n-k}$, for $n > 0$, and we have $2P(z) = e^z P(z) + 1$. (Cf. Eq. 1.2.9–10.)

Another proof comes from the fact that $P_n = \sum_{k \geq 0} \{ \binom{n}{k} k! \}$, since $\{ \binom{n}{k} \}$ is the number of ways to partition n elements into k nonempty parts and these parts can be permuted in $k!$ ways. Thus $\sum_{n \geq 0} P_n z^n / n! = \sum_{k \geq 0} (e^z - 1)^k = 1/(2 - e^z)$ by Eq. 1.2.9–23.

Still another proof, perhaps the most interesting, arises if we arrange the elements in sequence in a “stable” manner, so that K_i precedes K_j if and only if $K_i < K_j$ or ($K_i = K_j$ and $i < j$). Among all P_n outcomes, a given arrangement $K_{a_1} \dots K_{a_n}$ now occurs exactly 2^{k-1} times if the permutation $a_n \dots a_1$ contains k runs; hence P_n can be expressed in terms of the Eulerian numbers, $P_n = \sum_k \langle \binom{n}{k} \rangle 2^{k-1}$. Eq. 5.1.3–20 with $z = 2$ now establishes the desired result.

This generating function was obtained by A. Cayley [*Phil. Mag.* 18 (1859), 374–378] in connection with the enumeration of an imprecisely defined class of trees. See also J. Touchard, *Ann. Soc. Sci. Bruxelles* 53 (1933), 21–31. A table of P_1, \dots, P_{14} appears in O. A. Gross, *AMM* 69 (1962), 4–8; Gross gives the interesting formula $P_n = \sum_{k \geq 1} k^n / 2^{1+k}$, $n \geq 1$.

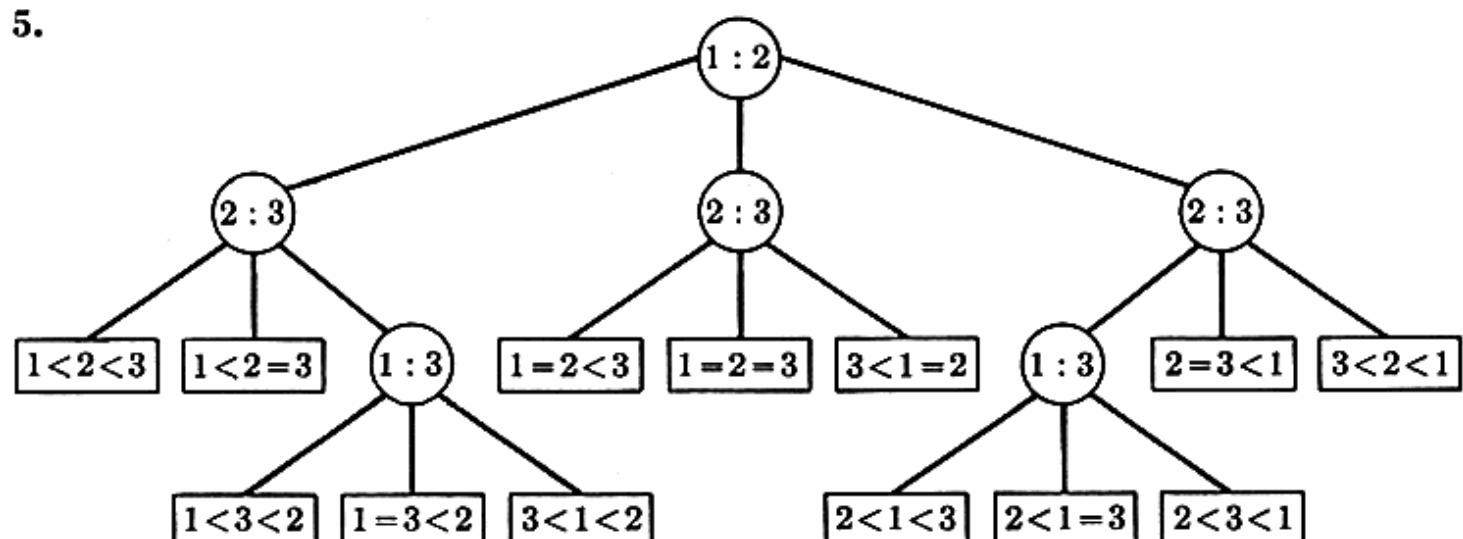
4. The representation

$$2P(z) = \frac{1}{2}(1 - i \cot(i(z - \ln 2)/2))$$

$$= \frac{1}{2} - \frac{1}{z - \ln 2} - \sum_{k \geq 1} \left(\frac{1}{z - \ln 2 - 2\pi ik} + \frac{1}{z - \ln 2 + 2\pi ik} \right)$$

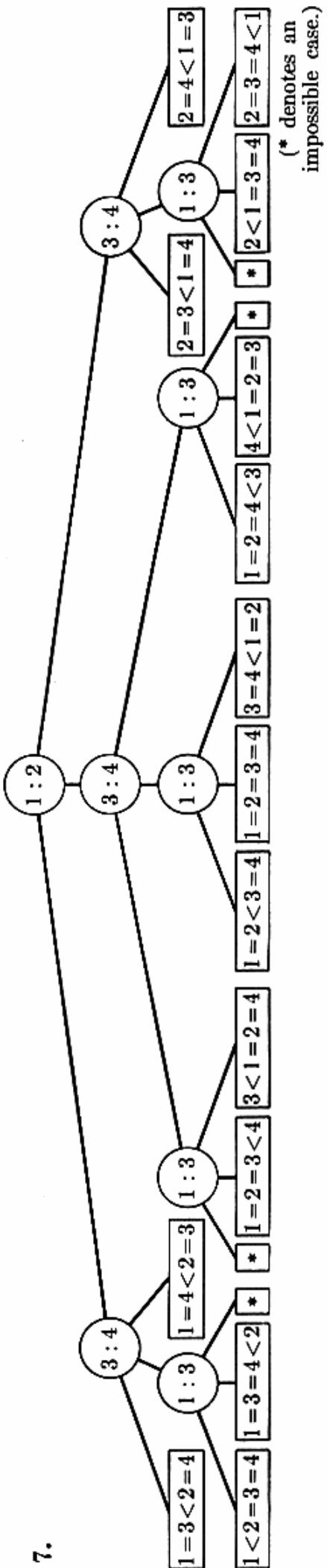
yields a convergent series $P_n/n! = \frac{1}{2}(\ln 2)^{-n-1} + \sum_{k \geq 1} \Re((\ln 2 + 2\pi ik)^{-n-1})$.

5.



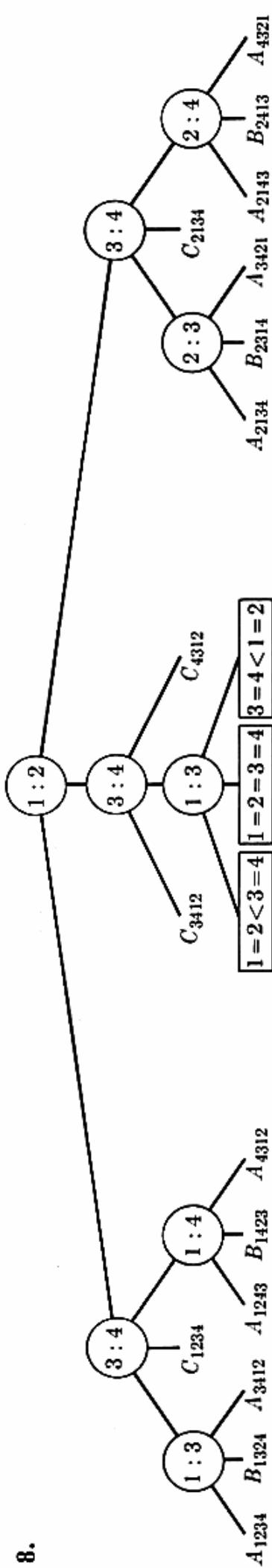
6. $S'(n) \geq S(n)$, since the keys might all be distinct; thus we must show that $S'(n) \leq S(n)$. Given a sorting algorithm which takes $S(n)$ steps on distinct keys, we can construct a sorting algorithm for the general case by defining the $=$ branch to be identical to the $<$ branch, removing redundancies. When an external node appears, we know all of the equality relations, since we have $K_{a_1} \leq K_{a_2} \leq \dots \leq K_{a_n}$ and an explicit comparison $K_{a_i}:K_{a_{i+1}}$ has been made for $1 \leq i < n$.

1

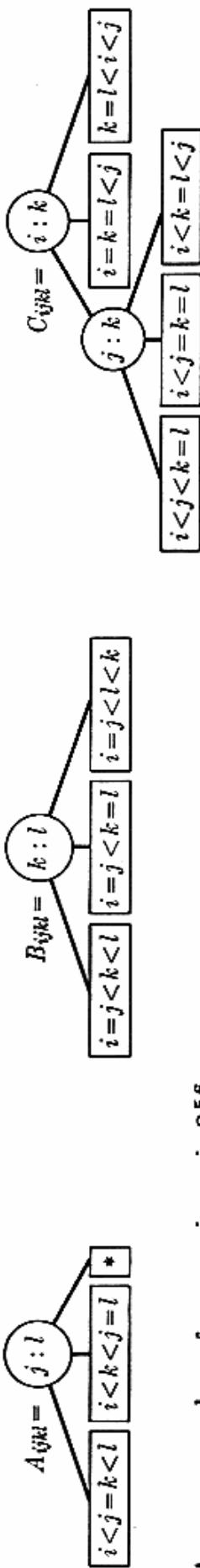


The average number of comparisons is $(2 + 3 + 3 + 2 + 3 + 3 + 3 + 3 + 6 + 3 + 3 + 3 + 3 + 2 + 3 + 3 + 2 + 3)/16 = 2\frac{3}{4}$.

8



where



Average number of comparisons is $3\frac{56}{81}$.

9. We need at least $n - 1$ comparisons to discover that all keys are equal, if they are. Conversely, $n - 1$ always suffices, since we can always deduce the final ordering after comparing K_1 with all of the other keys!

10. Let $f(n)$ be the desired function, and let $g(n)$ be the minimum average number of comparisons needed to sort $n + k$ elements when $k > 0$ and exactly k of the elements have known values (0 or 1). Then $f(0) = f(1) = g(0) = 0$, $g(1) = 1$; $f(n) = 1 + \frac{1}{2}f(n-1) + \frac{1}{2}g(n-2)$, $g(n) = 1 + \min(g(n-1), \frac{1}{2}g(n-1) + \frac{1}{2}g(n-2)) = 1 + \frac{1}{2}g(n-1) + \frac{1}{2}g(n-2)$, for $n \geq 2$. (Thus the best strategy is to compare two unknown elements whenever possible.) It follows that $f(n) - g(n) = \frac{1}{2}(f(n-1) - g(n-1))$ for $n \geq 2$, and $g(n) = \frac{2}{3}(n + \frac{1}{3}(1 - (-\frac{1}{2})^n))$ for $n \geq 0$. Hence the answer is $\frac{2}{3}n + \frac{2}{9} - \frac{2}{9}(-\frac{1}{2})^n - (\frac{1}{2})^{n-1}$, for $n \geq 1$. (This exact formula may be compared with the information-theoretic lower bound, $\log_3(2^n - 1) \approx 0.6309n$.)

11. Binary insertion proves that $S_m(n) \leq B(m) + (n-m)\lceil \log_2(m+1) \rceil$, for $n \geq m$. On the other hand $S_m(n) \geq \lceil \log_2 \sum_{1 \leq k \leq m} \binom{n}{k} k! \rceil$ and this is asymptotically $n \log_2 m$ (cf. Eq. 1.2.6-49).

12. (a) If there are no redundant comparisons, we can arbitrarily assign an order to keys which are actually equal, when they are first compared, since no order can be deduced from previously made comparisons. (b) Assume that the tree strongly sorts every sequence of zeros and ones; we shall prove that it strongly sorts every permutation of $\{1, 2, \dots, n\}$. Suppose it doesn't, so that there is a permutation for which it claims that $K_{a_1} \leq K_{a_2} \leq \dots \leq K_{a_n}$ whereas in fact $K_{a_i} > K_{a_{i+1}}$ for some i . Replace all elements $< K_{a_i}$ by 0 and all elements $\geq K_{a_i}$ by 1; by assumption the method will now sort when we take the path which leads to $K_{a_1} \leq K_{a_2} \leq \dots \leq K_{a_n}$, a contradiction.

13. If n is even, $F(n) - F(n-1) = 1 + F(\lfloor n/2 \rfloor) - F(\lfloor n/2 \rfloor - 1)$ so we must prove that $w_{k-1} < \lfloor n/2 \rfloor \leq w_k$; this is obvious since $w_{k-1} = \lfloor w_k/2 \rfloor$. If n is odd, $F(n) - F(n-1) = G(\lceil n/2 \rceil) - G(\lfloor n/2 \rfloor)$, so we must prove that $t_{k-1} < \lceil n/2 \rceil \leq t_k$; this is obvious since $t_{k-1} = \lceil w_k/2 \rceil$.

14. By exercise 1.2.4-42, the sum is $n\lceil \log_2(\frac{3}{4}n) \rceil - (w_1 + \dots + w_j)$ where $w_j < n \leq w_{j+1}$. The latter sum is $w_{j+1} - \lfloor j/2 \rfloor - 1$. $F(n)$ can therefore be written $n\lceil \log_2(\frac{3}{4}n) \rceil - \lfloor 2^{\lceil \log_2(6n) \rceil}/3 \rfloor + \lfloor \frac{1}{2}\log_2(6n) \rfloor$ (and in many other ways).

15. If $\lceil \log_2(\frac{3}{4}n) \rceil = \log_2(\frac{3}{4}n) + \theta$, $F(n) = n \log_2 n - (3 - \log_2 3)n + n(\theta + 1 - 2^\theta) + O(\log n)$. If $\lceil \log_2 n \rceil = \log_2 n + \theta$, $B(n) = n \log_2 n - n + n(\theta + 1 - 2^\theta) + O(\log n)$. [Note that $\log_2 n! = n \log_2 n - n/(\ln 2) + O(\log n)$; $1/(\ln 2) \approx 1.443$; $3 - \log_2 3 \approx 1.415$.]

17. The number of cases with $b_k < a_p < b_{k+1}$ is

$$\binom{m-p+n-k}{m-p} \binom{p-1+k}{p-1},$$

and the number of cases with $a_j < b_q < a_{j+1}$ is

$$\binom{n-q+m-j}{n-q} \binom{q-1+j}{q-1}.$$

18. No, since we are considering only the less efficient branch of the tree below each comparison. One of the more efficient branches might turn out to be harder to handle.

20. Let L be the maximum level on which an external node appears, and let l be the minimum such level. If $L \geq l + 2$, we can remove two nodes from level L and place them below a node at level l ; this decreases the external path length by $l + 2L - (L - 1 + 2(l + 1)) = L - l - 1 \geq 1$. Conversely, if $L \leq l + 1$, let there be k external nodes on level $l - 1$ and $N - k$ on level l , where $0 < k \leq N$. By exercise 2.3.4.5–3, $k2^{-l+1} + (N - k)2^{-l} = 1$; hence $N + k = 2^l$. The inequalities $N < 2^l \leq 2N$ now show that $l = \lfloor \log_2 N \rfloor + 1$; this defines k and yields the external path length (33).

21. Let $r(x)$ be the root of x 's right subtree. We need $\lceil \log_2 t(l(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$ and $\lceil \log_2 t(r(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$. The first condition is equivalent to $2t(l(x)) - t(x) \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$, and the second condition is equivalent to $t(x) - 2t(l(x)) \leq 2^{\lceil \log_2 t(x) \rceil} - t(x)$.

22. By exercise 20, the conditions $\lfloor \log_2 t(l(x)) \rfloor, \lfloor \log_2 t(r(x)) \rfloor \geq \lfloor \log_2 t(x) \rfloor - 1$ and $\lceil \log_2 t(l(x)) \rceil, \lceil \log_2 t(r(x)) \rceil \leq \lceil \log_2 t(x) \rceil - 1$ are necessary and sufficient. Arguing as in exercise 21, we can prove them equivalent to the stated conditions. [Cf. Martin Sandelius, *AMM* 68 (1961), 133–134.] See exercise 33 for a generalization.

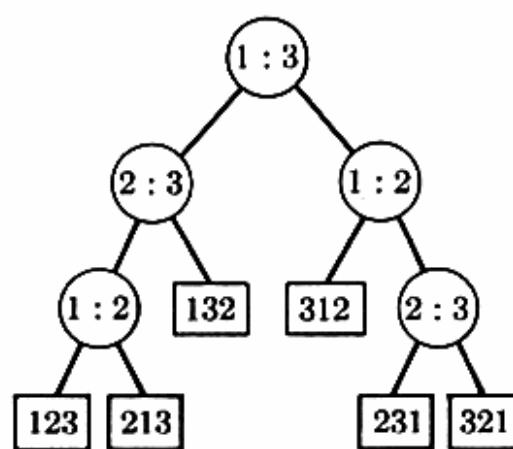
23. Multiple list insertion assumes that the keys are uniformly distributed in a known range, so it isn't a “pure comparison” method satisfying the restrictions considered in this section.

24. First proceed as if sorting five elements, until after five comparisons we reach one of the configurations in (6). In the first three cases, complete sorting the five elements in two more comparisons, then insert the sixth element f . In the latter case, first compare $f:b$, insert f into the main chain, then insert c . [*Théorie des Questionnaires*, p. 116.]

25. Since $N = 7! = 5040$ and $q = 13$, there would be $8192 - 5040 = 3152$ external nodes on level 12 and $5040 - 3152 = 1888$ on level 13.

26. (The best methods known to date have an external path length of 62416.)

27.



is the *only* way to recognize the two most frequent permutations with two comparisons, even though the first comparison produces a .27–.73 split!

28. Lun Kwan has constructed an 873-line program whose average running time is $38.925u$. Its maximum running time is $43u$; the latter appears to be optimal since it is the time for 7 compares, 7 tests, 6 loads, 5 stores.

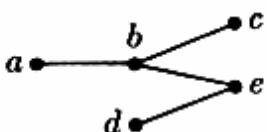
29. We must make at least $S(n)$ comparisons, because it is impossible to know whether a permutation is even or odd unless we have made enough comparisons to determine it uniquely. For we can assume that enough comparisons have been made to narrow things down to two possibilities that depend on whether or not a_i is less than a_j , for some i and j ; one of the two possibilities is even, the other is odd. [On the other hand

there is an $O(n)$ algorithm for this problem, which simply counts the number of cycles and uses no comparisons at all; see exercise 5.2.2–2.]

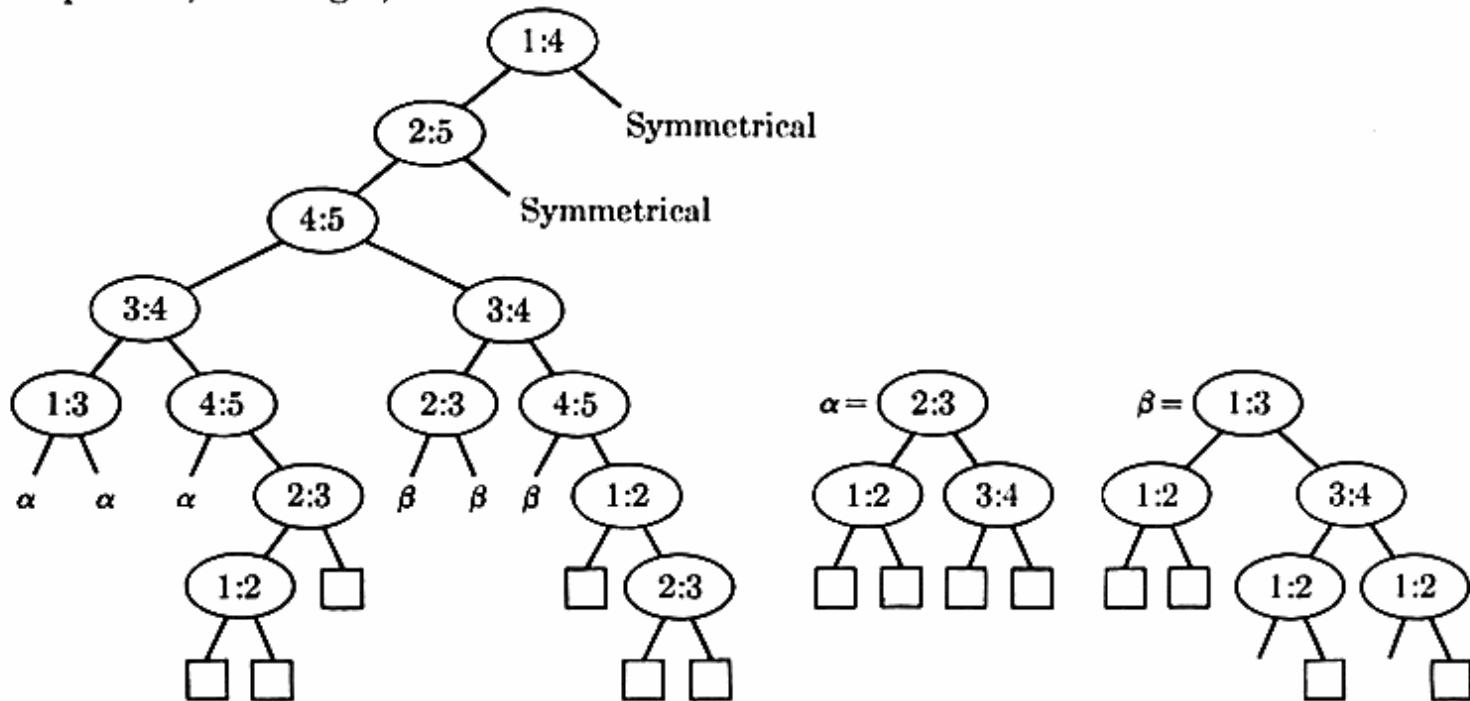
30. Start with an optimal comparison tree of height $S(n)$; repeatedly interchange $i \leftrightarrow j$ in the right subtree of a node labeled $(i:j)$, from top to bottom. Interpreting the result as a comparison-exchange tree, every terminal node defines a unique permutation which can be sorted by at most $n - 1$ more comparison-exchanges (by exercise 5.2.2–2).

[The idea of a comparison-exchange tree is due to T. N. Hibbard.]

31. At least 8 are required, since every tree of height 7 will produce the configuration



(or its dual) in some branch after 4 steps, with $a \neq 1$. This configuration cannot be sorted in 3 more comparison/exchange operations. On the other hand the following tree achieves the desired bound (and perhaps also the minimum *average* number of comparison/exchanges):



33. Simple operations applied to any tree of order x and resolution 1 can be applied to yield another whose weighted path length is no greater and such that (a) all external nodes lie on levels k and $k - 1$ for some k ; (b) at most one external node is non-integer, and if it is present it lies on level k . The weighted path length of any such tree has the stated value, so this must be minimal. Conversely if (iv) and (v) hold in any real-valued search tree it is possible to show by induction that the weighted path length has the stated value, since there is a simple formula for the weighted path length of a tree in terms of the weighted path lengths of the two subtrees of the root.

SECTION 5.3.2

1. $S(m + n) \leq S(m) + S(n) + M(m, n)$.
2. The internal node which is k th in symmetric order corresponds to the comparison $A_1 : B_k$.
3. Strategy $B(1, l)$ is no better than strategy $A(1, l+1)$, and strategy $B'(1, l)$ no

better than $A'(1, l - 1)$; hence we must solve the recurrence

$M_*(1, n)$

$$= \min_{1 \leq i \leq n} \max \left(\max_{1 \leq l \leq i} (1 + .M.(1, l - 1)), \max_{j \leq l \leq n} (1 + .M.(1, n - l)) \right), \quad n \geq 1;$$

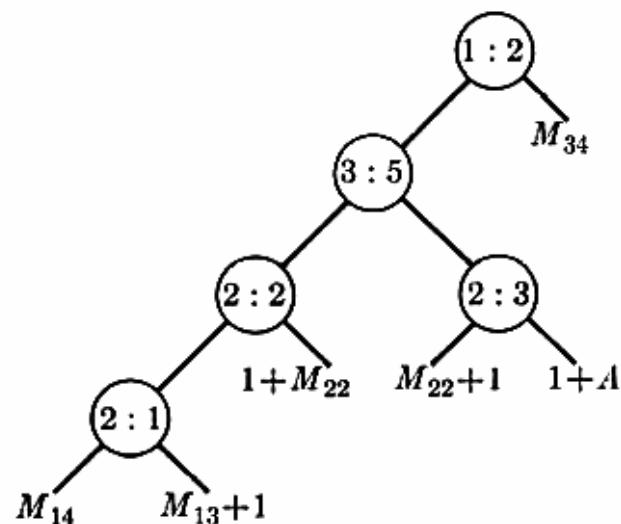
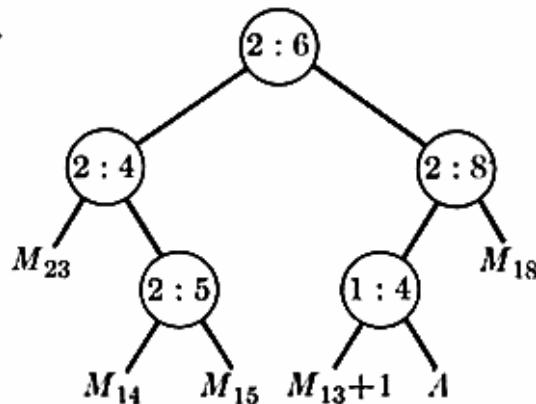
$M(1, 0) = 0$. It is not difficult to verify that $\lceil \log_2(n+1) \rceil$ satisfies this recurrence.

6. Strategy $A'(i, i+1)$ can be used when $j = i+1$, except when $i \leq 2$. And we can use strategy $A(i, i+2)$ when $j \geq i+2$.

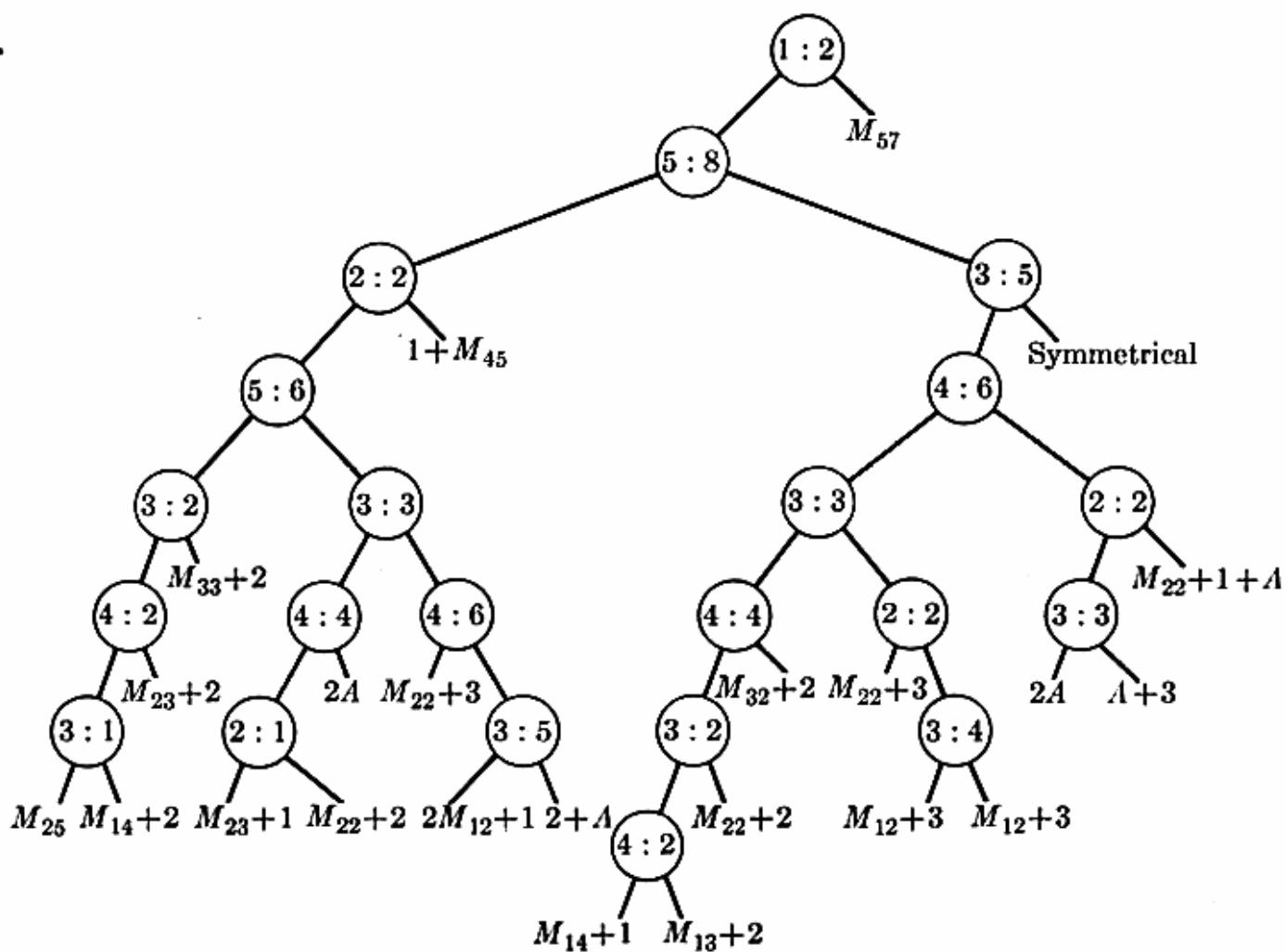
7. To insert $k + m$ elements among n others, independently insert k elements and m elements. (When k and m are large, an improved procedure is possible, see exercise 19.)

8. In the following diagrams, $i:j$ denotes the comparison $A_i:B_j$, $M_{i,j}$ denotes merging i elements with j in $M(i, j)$ steps, and A denotes sorting the pattern  or  in three steps.

9.



10.



11. Let $n = g_t$ as in the hint. We may assume that $t \geq 6$. Without loss of generality let $A_2:B_j$ be the first comparison. If $j > g_{t-1}$, the outcome $A_2 < B_j$ will require $\geq t$ more steps. If $j \leq g_{t-1}$, the outcome $A_2 < B_j$ would be no problem, so only the case $A_2 > B_j$ needs study, and we get the most information when $j = g_{t-1}$. If $t = 2k+1$, we might have to merge A_2 with the $g_t - g_{t-1} = 2^{k-1}$ elements $> B_{g_{t-1}}$, and merge A_1 with the g_{t-1} others, but this requires $k + (k+1) = t$ further steps. On the other hand if $n = g_t - 1$, we could merge A_2 with $2^{k-1} - 1$ elements, then A_1 with n elements, in $(k-1) + (k+1)$ further steps, hence $M(2, g_t - 1) \leq t$.

The case $t = 2k$ is considerably more difficult; note that $g_t - g_{t-1} \geq 2^{k-2}$. After $A_2 > B_{g_{t-1}}$, suppose we compare $A_1:B_j$. If $j > 2^{k-1}$ the outcome $A_1 < B_j$ requires $k + (k-1)$ further comparisons (too many). If $j \leq 2^{k-1}$, we can argue as before that $j = 2^{k-1}$ gives most information. After $A_1 > B_{2^{k-1}}$, the next comparisons with A_1 might as well be with $B_{2^{k-1}+2^{k-2}}$, then $B_{2^{k-1}+2^{k-2}+2^{k-3}}$; since $2^{k-1} + 2^{k-2} + 2^{k-3} > g_{t-1}$, we are left with merging $\{A_1, A_2\}$ with $n - (2^{k-1} + 2^{k-2} + 2^{k-3})$ elements. Of course we needn't make any comparisons with A_1 right away; we could instead compare $A_2:B_{n+1-j}$. If $j \leq 2^{k-3}$, we consider the case $A_2 < B_{n+1-j}$, while if $j > 2^{k-3}$ we consider $A_2 > B_{n+1-j}$. The latter case requires at least $(k-2) + (k+1)$ more steps. Continuing, we find that the *only* potentially fruitful line is $A_2 > B_{g_{t-1}}, A_2 < B_{n+1-2^{k-3}}, A_1 > B_{2^{k-1}}, A_1 > B_{2^{k-1}+2^{k-2}}, A_1 > B_{2^{k-1}+2^{k-2}+2^{k-3}}$, but then we have exactly g_{t-5} elements left! Conversely, if $n = g_t - 1$, this line works. [*Acta Informatica* 1 (1971), 145–158.]

12. The first comparison must be either $\alpha:X_k$ for $1 \leq k \leq i$, or (symmetrically) $\beta:X_{n-k}$ for $1 \leq k \leq j$. In the former case the response $\alpha < X_k$ leaves us with $R_n(k-1, j)$ more comparisons to make; the response $\alpha > X_k$ leaves us with the problem of sorting $\alpha < \beta, Y_1 < \dots < Y_{n-k}, \alpha < Y_{i-k+1}, \beta > Y_{n-k-j}$, where $Y_r = X_{r-k}$.

13. [*Computers in Number Theory* (New York: Academic Press, 1971), 397–404.]

15. Double m until it exceeds n . This involves $\lfloor \log_2(n/m) \rfloor + 1$ doublings.

16. All except $(m, n) = (2, 8), (3, 6), (3, 8), (3, 10), (4, 8), (4, 10), (5, 9), (5, 10)$, when it's one over.

17. Assume that $m \leq n$ and let $t = \log_2(n/m) - \theta$. Then $\log_2(\frac{m+n}{m}) > \log_2 n^m - \log_2 m! \geq m \log_2 n - (m \log_2 m - m + 1) = m(t + \theta) + m - 1 = H(m, n) + \theta m - \lfloor 2^\theta m \rfloor \geq H(m, n) + \theta m - 2^\theta m \geq H(m, n) - m$. (The inequality $m! \leq m^m 2^{1-m}$ is a consequence of the fact that $k(m-k) \leq (m/2)^2$ for $1 \leq k < m$.)

19. First merge $\{A_1, \dots, A_m\}$ with $\{B_2, B_4, \dots, B_{2\lfloor n/2 \rfloor}\}$. Then we must insert the odd elements B_{2i-1} among a_i of the A 's for $1 \leq i \leq \lceil n/2 \rceil$, where $a_1 + a_2 + \dots + a_{\lceil n/2 \rceil} \leq m$. The latter operation requires $\leq a_i$ operations for each i , so at most m more comparisons will finish the job.

20. Apply (12).

23. The oracle keeps an $n \times n$ matrix X whose entries x_{ij} are initially all 1. When the algorithm asks if $A_i = B_j$, the oracle sets x_{ij} to 0. The answer is “No,” unless the permanent of X has just become zero. In the latter case, the oracle answers “Yes,” (as it must, lest the algorithm terminate immediately!), and deletes row i and column j from X ; the resulting $(n-1) \times (n-1)$ matrix will have a nonzero permanent. The oracle continues in this way until only a 0×0 matrix is left.

One can show (for example by using Philip Hall's theorem, cf. Chapter 7) that at least n zeroes are deleted when the oracle first answers “Yes,” and $n-1$ the second

time, etc. The algorithm will terminate only after receiving n “Yes” answers to non-redundant questions, and after asking at least $n + (n - 1) + \dots + 1$ questions.

SECTION 5.3.3

1. Player 11 lost to 05, so 13 was known to be worse than 05, 11, 12.
2. Let x be the t th largest, and let S be the set of all elements y such that the comparisons made are insufficient to prove either that $x < y$ or $y < x$. There are permutations, consistent with all the comparisons made, in which all elements of S are less than x ; for we can stipulate that all elements of S are less than x and embed the resulting partial ordering in a linear ordering. Similarly there are consistent permutations in which all elements of S are greater than x . Hence we don't know the rank of x unless S is empty.
3. An oracle may regard the loser of the first comparison as the worst player of all. [R. W. Floyd has used a similar approach to show that the values in Table 1 are exact for all $n \leq 7$.]
4. There are at least n^t outcomes.
5. In fact, $W_t(n) \leq V_t(n) + S(t - 1)$, by exercise 2.
6. Let $g(l_1, l_2, \dots, l_m) = m - 2 + \log_2(2^{l_1} + 2^{l_2} + \dots + 2^{l_m})$, and assume that $f = g$ whenever $l_1 + l_2 + \dots + l_m + 2m < N$. We shall prove that $f = g$ when $l_1 + l_2 + \dots + l_m + 2m = N$. We may assume that $l_1 \geq l_2 \geq \dots \geq l_m$. There are only a few possible ways to make the first comparison:

Strategy A(j, k), for $j < k$. Compare the largest element of group j with the largest of group k . This gives the relation

$$\begin{aligned} f(l_1, \dots, l_m) &\leq 1 + g(l_1, \dots, l_{j-1}, l_j + 1, l_{j+1}, \dots, l_{k-1}, l_k + 1, \dots, l_m) \\ &= g(l_1, \dots, l_{j-1}, l_j, l_{j+1}, \dots, l_{k-1}, l_j, l_{k+1}, \dots, l_m) \geq g(l_1, \dots, l_m). \end{aligned}$$

Strategy B(j, k), for $l_k > 0$. Compare the largest element of group j with one of the small elements of group k . This gives the relation

$$f(l_1, \dots, l_m) \leq 1 + \max(\alpha, \beta) = 1 + \beta,$$

where

$$\begin{aligned} \alpha &= g(l_1, \dots, l_{j-1}, l_{j+1}, \dots, l_m) \leq g(l_1, \dots, l_m) - 1, \\ \beta &= g(l_1, \dots, l_{k-1}, l_k - 1, l_{k+1}, \dots, l_m) \geq g(l_1, \dots, l_m) - 1. \end{aligned}$$

Strategy C(j, k), for $j \leq k, l_j > 0, l_k > 0$. Compare a small element from group j with a small element from group k . The corresponding relation is

$$f(l_1, \dots, l_m) \leq 1 + g(l_1, \dots, l_{k-1}, l_k - 1, l_{k+1}, \dots, l_m) \geq g(l_1, \dots, l_m).$$

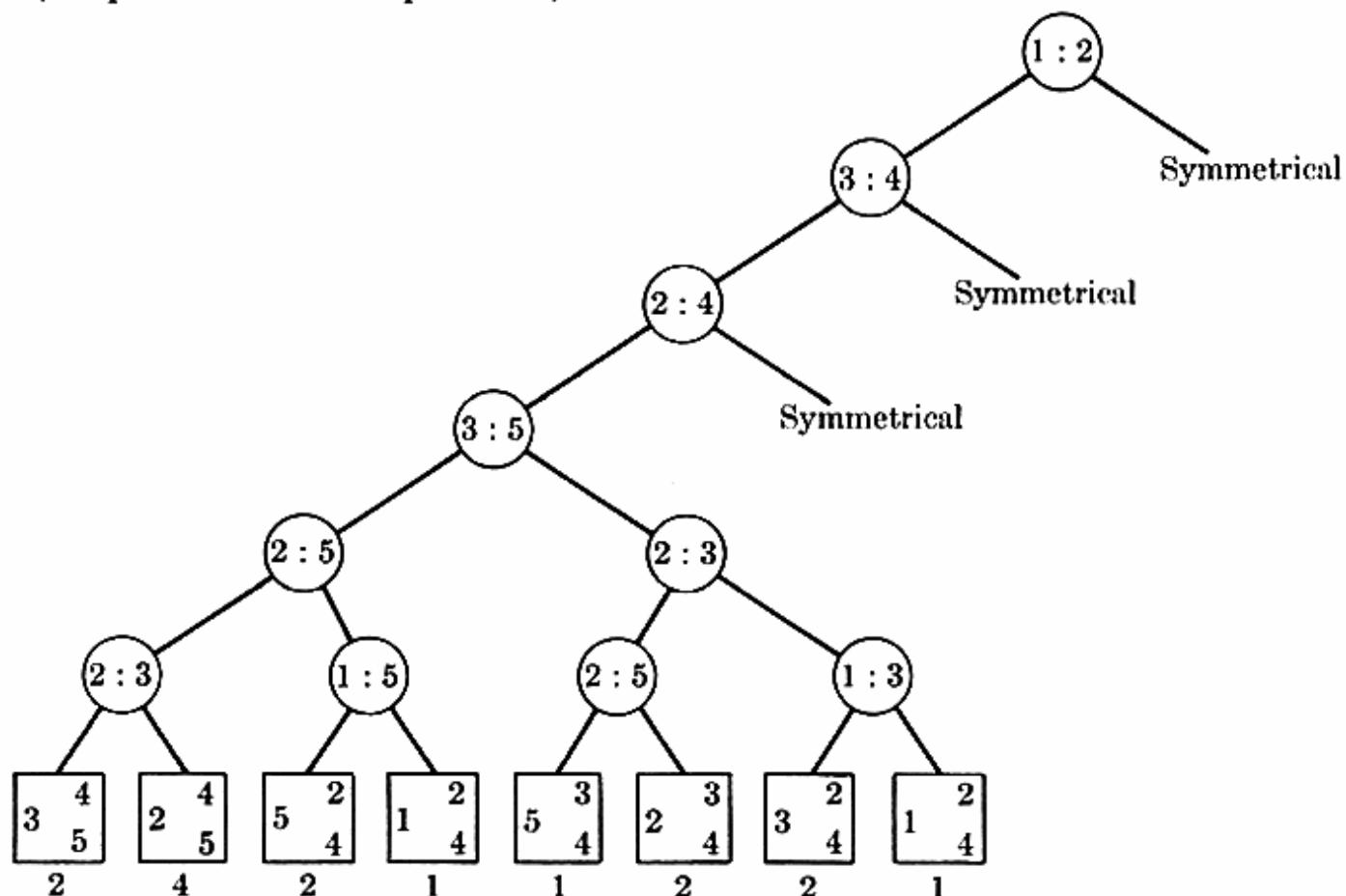
The value of $f(l_1, \dots, l_m)$ is found by taking the minimum right-hand side over all these strategies; hence $f(l_1, \dots, l_m) \geq g(l_1, \dots, l_m)$. When $m > 1$, Strategy A($m - 1, m$) shows that $f(l_1, \dots, l_m) \leq g(l_1, \dots, l_m)$, since $g(l_1, \dots, l_{m-1}, l_m) = g(l_1, \dots, l_{m-1}, l_{m-1})$ when $l_1 \geq \dots \geq l_m$. (*Proof:* $\lceil \log_2(M + 2^a) \rceil = \lceil \log_2(M + 2^b) \rceil$ for $0 \leq a \leq b$, when M is a positive multiple of 2^b .) When $m = 1$, use Strategy C(1, 1).

[S. S. Kislytsyn's paper determined the optimum strategy $A(m - 1, m)$ and evaluated $f(l, l, \dots, l)$ in closed form; the general formula for f and this simplified proof were discovered by Floyd in 1970.]

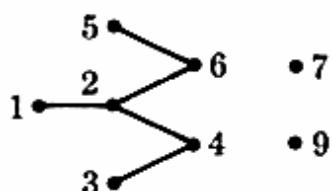
7. For $j > 1$, if $j + 1$ is in α' , c_j is 1 plus the number of comparisons needed to select the next largest element of α' . Similarly if $j + 1$ is in α'' ; and c_1 is always 0, since the tree always looks the same at the end.

8. In other words, is there an extended binary tree with n external nodes such that the sum of the distances to the $t - 1$ farthest internal nodes from the root is less than the corresponding sum for the complete binary tree? The answer is no, since it is not hard to show that the k th largest element of $\mu(\alpha)$ is $\geq \lfloor \log_2(n - k) \rfloor$ for all α .

9. (All paths use six comparisons.)



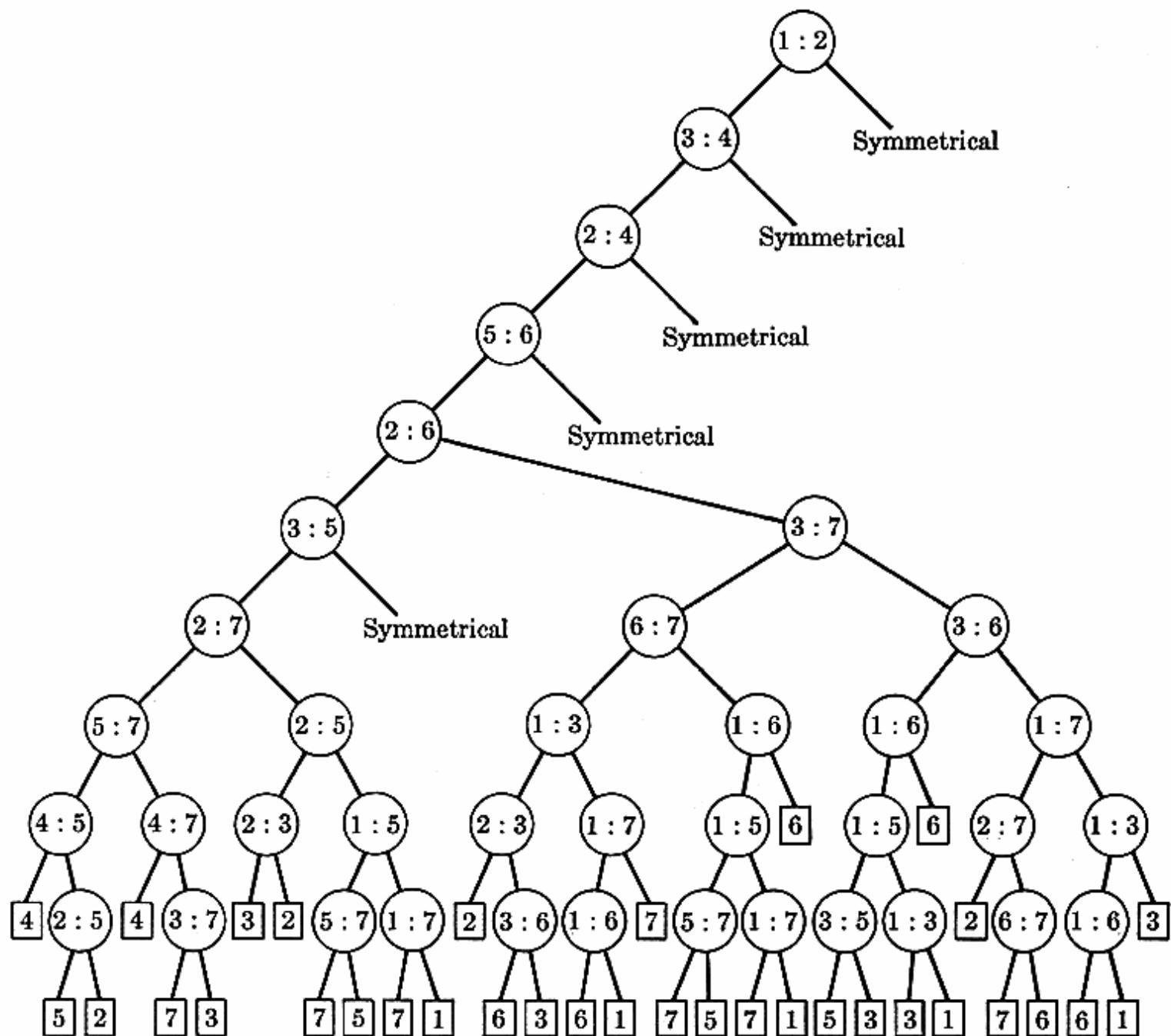
10. After the first seven comparisons in Doren's method, we may discard X_8 , and so we must find the fourth largest of



This is more information than we knew after four steps of Doren's method, so at most $12 - 4 = 8$ more comparisons are required.

11. After discarding the smallest of $\{X_1, X_2, X_3, X_4\}$, we have the configuration plus $n - 3$ isolated elements; the third largest of these can be found in $V_3(n - 1) - 1$ further steps.

12. (Found manually by trial and error, using exercise 6 to help find fruitful lines.)



13. After finding the median of the first $f(n)$ elements, say X_j , compare it to each of the others; this splits the elements into approximately $n/2 - k$ less than X_j and $n/2 + k$ greater than X_j , for some k . It remains to find the $|k|$ th largest or smallest element of the bigger set, which requires $n/2 + O(|k| \log n)$ further comparisons. The average value of $|k|$ (consider points uniformly distributed in $[0, 1]$) is $O(1/\sqrt{n}) + O(n/\sqrt{f(n)})$. Let $T(n)$ be the average number of comparisons when $f(n) = n^{2/3}$; $T(n) - n = T(n^{2/3}) - n^{2/3} + n/2 + O(n^{2/3})$, and the result follows.

It is interesting to note that when $n = 5$, this method requires only $5\frac{1}{15}$ comparisons on the average, slightly better than the tree of exercise 9.

14. In general, the t largest can be found in $V_t(n - 1) + 1$ comparisons, by finding the t th largest of $\{X_1, \dots, X_{n-1}\}$ and comparing it with X_n , because of exercise 2.

15. $\min(t, n + 1 - t)$. Assuming that $t \leq n + 1 - t$, if we don't save each of the first t words when they are first read in, we may have forgotten the t th largest, depending on the subsequent values still unknown to us. Conversely, t locations are sufficient, since we can compare a newly input item with the previous t th largest, storing the register iff it is greater.

- 16.** The algorithm starts with $(a, b, c, d) = (n, 0, 0, 0)$ and ends with $(0, 1, 1, n - 2)$. If the oracle avoids “surprising” outcomes, the only transitions possible after each comparison are from (a, b, c, d) to itself or to

$$\begin{aligned} & (a - 2, b + 1, c + 1, d), \quad \text{if } a \geq 2; \\ & (a - 1, b, c + 1, d) \quad \text{or} \quad (a - 1, b + 1, c, d), \quad \text{if } a \geq 1; \\ & (a, b - 1, c, d + 1), \quad \text{if } b \geq 2; \\ & (a, b, c - 1, d + 1), \quad \text{if } c \geq 2. \end{aligned}$$

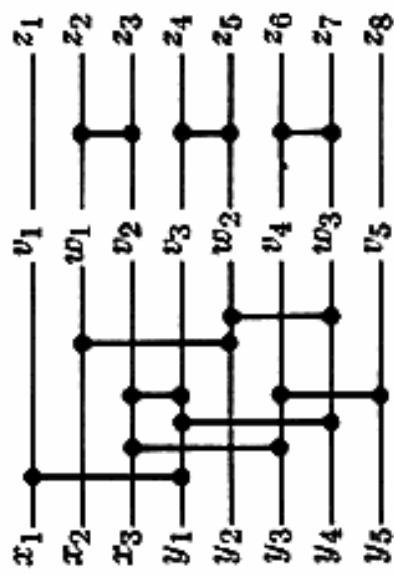
It follows that $\lceil \frac{3}{2}a \rceil + b + c - 2$ comparisons are needed to get from (a, b, c, d) to $(0, 1, 1, a + b + c + d - 2)$. [*CACM* 15 (1972), 462–464.]

- 17.** Use (6) first for the largest, then for the smallest, noting that $\lfloor n/2 \rfloor$ of the comparisons are common to both.
- 18.** $V_t(n) \leq 18n - 151$, for all sufficiently large n .
- 21.** Let the elements be $\{x_1, \dots, x_n, y, z\}$, where $n = 2^k$. Find the two largest x 's, say $b < a$, in $2^k + k - 2$ comparisons, using Kislitsyn's method; and compare $y:z$ (say $y < z$). Now if $z < b$, continue Kislitsyn's method, finding the third largest of the x 's in $k - 1$ comparisons, and compare it to z . On the other hand, if $z > b$, it is easy to finish in at most three more steps.

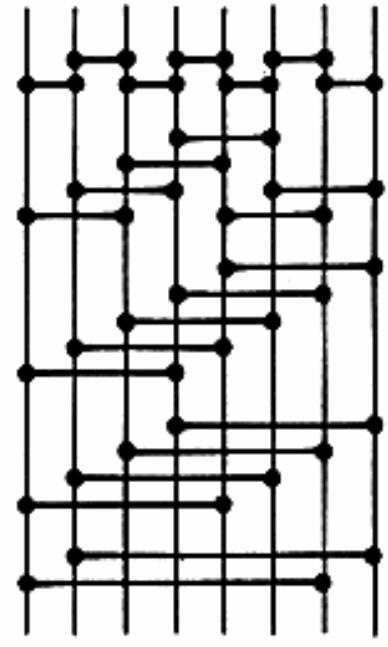
SECTION 5.3.4

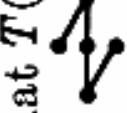
1. (When m is odd it is best to have v_m followed by w_{m+1}, w_{m+2}, \dots instead of by $w_{m+1}, v_{m+1}, w_{m+2}, \dots$ in the diagram.)

(3, 5) odd-even merge



Pratt eight-sort



2. See the above diagram for $n = 8$.
3. $C(m, m - 1) = C(m, m) - 1$, for $m \geq 1$.
4. If $\hat{T}(6) = 4$, there would be three comparators acting at each time, since $\hat{S}(6) = 12$. But then removing the bottom line and its four comparators would give $\hat{S}(5) \leq 8$, a contradiction. [The same argument yields $\hat{T}(7) = \hat{T}(8) = 6$.]
5. Let $f(n) = f(\lceil n/2 \rceil) + 1 + \lceil \log_2 \lceil n/2 \rceil \rceil$, if $n \geq 2$; by exercises 1.2.4–34, 35 , $f(n) = (1 + \lceil \log_2 \frac{1}{4}n \rceil) + (1 + \lceil \log_2 \frac{1}{4}n \rceil) + \dots$.
6. We may assume that each stage makes $\lfloor n/2 \rfloor$ comparisons (extra comparisons can't hurt). Since $\hat{T}(6) = 5$, it suffices to show that $T(5) = 5$. After two stages when $n = 5$, we cannot avoid the partial orderings  or  which cannot be sorted in two more stages.
7. Assume that the input keys are $\{1, 2, \dots, 10\}$. The key fact is that after the

first 16 comparators, lines 2–5 cannot contain 8 or 9, nor can they contain both 6 and 7.

8. Straightforward generalization of Theorem F.

9. $\hat{M}(3, 3) \geq \hat{S}(6) - 2\hat{S}(3)$; $\hat{M}(4, 4) \geq \hat{S}(8) - 2\hat{S}(4)$; $\hat{M}(5, 5) \geq 2\hat{M}(2, 3) + 3$ by exercise 8; and $\hat{M}(2, 3) \geq \hat{S}(5) - \hat{S}(2) - \hat{S}(3)$.

10. The hint follows by the method of proof in Theorem Z. Then show that the number of 0's in the even subsequence minus the number of 0's in the odd subsequence is ± 1 .

11. (Solution by M. W. Green.) The network is symmetric in the sense that, whenever z_i is compared to z_j , there is a corresponding comparison of z_{2^p-1-j} : z_{2^p-1-i} . Any symmetric network capable of sorting a sequence $\langle z_0, \dots, z_{2^p-1} \rangle$ will also sort the sequence $\langle -z_{2^p-1}, \dots, -z_0 \rangle$.

Batcher has observed that the network will actually sort any cyclic shift $\langle z_j, z_{j+1}, \dots, z_{2^p-1}, z_0, \dots, z_{j-1} \rangle$ of a bitonic sequence. This is a consequence of the 0-1 principle.

12. $x \vee y$ is (consider 0-1 sequences), but not $x \vee y$ (consider $\langle 3, 1, 4, 5 \rangle \wedge \langle 6, 7, 8, 2 \rangle$).

13. A perfect shuffle has the effect of replacing z_i by z_j , where the binary representation of j is that of i rotated cyclically to the right one place (cf. exercise 3.4.2–13). Consider shuffling the comparators instead of the lines; then the first column of comparators acts on the pairs $z[i]$ and $z[i \oplus 2^{r-1}]$, the next column on $z[i]$ and $z[i \oplus 2^{r-2}]$, \dots , the t th column on $z[i]$ and $z[i \oplus 1]$, the $(t+1)$ st column on $z[i]$ and $z[i \oplus 2^{r-1}]$ again, etc. Here “ \oplus ” denotes exclusive-or on the binary representation. This shows that Fig. 57 is equivalent to Fig. 56; after s stages we have groups of 2^s elements which are alternately sorted and reverse-sorted.

14. After l levels the input x_1 can be in at most 2^l different places. After merging is complete, x_1 can be in $n+1$ different places.

15. $[1:4][3:2][1:3][2:4][2:3]$.

16. The process clearly terminates. Each execution of step T2 has the effect of interchanging the i_q th and j_q th outputs, so the result of the algorithm is to permute the output lines in some way. Since the resulting (standard) network makes no change to the input $\langle 1, 2, \dots, n \rangle$, the output lines must have been returned to their original position.

17. Make the network standard by the algorithm of exercise 16; then by considering the input sequence $\langle 1, 2, \dots, n \rangle$, we see that standard selection networks must take the t largest elements into the t highest-numbered lines; and a $\hat{V}_t(n)$ network must take the t th largest into line $n+1-t$. Apply the zero-one principle.

18. The proof in Theorem A shows that $\hat{V}_t(n) \geq (n-t)\lceil \log_2(t+1) \rceil + \lceil \log_2 t \rceil$.

19. The network $[1:n][2:n] \dots [1:3][2:3]$ selects the smallest two elements with $2n-4$ comparators; add $[1:2]$ for $\hat{V}_2(n)$. The lower bound comes from Theorem A (cf. exercise 18).

20. First note that $\hat{V}_3(n) \geq \hat{V}_3(n-1) + 2$ when $n \geq 4$; by symmetry the first comparator may be assumed to be $[1:n]$; after this must come a network to select the third largest of $\langle x_2, x_3, \dots, x_n \rangle$, and another comparator touching line 1. On the other hand, $\hat{V}_3(5) \leq 7$, since four comparators find the min and max of $\{x_1, x_2, x_3, x_4\}$ and it remains to sort three elements.

21. By induction on the length of α , since $x_i \leq y_i$ and $x_j \leq y_j$ implies that $x_i \wedge x_j \leq y_i \wedge y_j$ and $x_i \vee x_j \leq y_i \vee y_j$.
22. By induction on the length of α , since $(x_i \wedge x_j)(y_i \wedge y_j) + (x_i \vee x_j)(y_i \vee y_j) \geq x_i y_i + x_j y_j$. [Consequently $\nu(x \wedge y) \leq \nu(x\alpha \wedge y\alpha)$, an observation due to W. Shockley.]
23. Let $x_k = 1$ iff $p_k \geq j$, $y_k = 1$ iff $p_k > j$; then $(x\alpha)_k = 1$ iff $(p\alpha)_k \geq j$, etc.
24. The formula for l'_t is obvious and for l'_j take $z = x \wedge y$ as in the hint and observe that $(z\alpha)_i = (z\alpha)_j = 0$ by exercise 21. Adding additional 1's to z shows the existence of a permutation p with $(p\alpha')_j \leq \zeta(z)$, by exercise 23. The relations for u'_t , u'_j follow by reversing the order.
25. Let $f(x) = (x\alpha)_k$; let z be a vector of 0's and 1's with $f(z) = 1$, having minimum $\nu(z)$ over all such vectors. (It follows that $\nu(z) = n + 1 - u_k$.) Let p in P_n be a permutation such that $f(p) = l_k$, and let p correspond to vectors x and y as in exercise 23. The following algorithm transforms x and y in such a way that the corresponding vectors p have $f(p)$ taking all the values from l_k to u_k , inclusive; each step of the algorithm makes x have at least one more component in common with z , while changing $\nu(x)$ by 0 or ± 1 . (Note that if $z \not\leq x$, there is some r with $z_r > x_r$ and some s with $x_s < z_s$ because of the minimality of $\nu(z)$.)

If the following hold:

Replace (x, y) by:

$z \not\leq x$, $z_r > x_r$, $z_s < x_s$, and

$f(y \vee e^{(r)}) = 0$	$(x \vee e^{(r)}, y \vee e^{(r)})$
$f(y \vee e^{(r)}) = 1$, $f(x \wedge \bar{e}^{(s)}) = 1$	$(x \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)})$
$f(y \vee e^{(r)}) = 1$, $f((x \vee e^{(r)}) \wedge \bar{e}^{(s)}) = 0$	$(x \vee e^{(r)}, (x \vee e^{(r)}) \wedge \bar{e}^{(s)})$
$f((x \vee e^{(r)}) \wedge \bar{e}^{(s)}) = 1$, $f((y \vee e^{(r)}) \wedge \bar{e}^{(s)}) = 0$	$((x \vee e^{(r)}) \wedge \bar{e}^{(s)}, (y \vee e^{(r)}) \wedge \bar{e}^{(s)})$
$f((y \vee e^{(r)}) \wedge \bar{e}^{(s)}) = 1$	$((y \vee e^{(r)}) \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)})$
$z \leq x$, $z_s < x_s$	$(x \wedge \bar{e}^{(s)}, y \wedge \bar{e}^{(s)}).$

[Is there a simpler solution?]

26. There is a one-to-one correspondence which takes the element $\langle p_1, \dots, p_n \rangle$ of $P_n\alpha$ into the “covering sequence” $x^{(0)}$ covers $x^{(1)}$ covers \dots covers $x^{(n)}$, where the $x^{(i)}$ are in $D_n\alpha$; in this correspondence, $x^{(i-1)} = x^{(i)} \vee e^{(i)}$ iff $p_j = i$. For example, $\langle 3, 1, 4, 2 \rangle$ corresponds to the sequence $\langle 1, 1, 1, 1 \rangle$ covers $\langle 1, 0, 1, 1 \rangle$ covers $\langle 1, 0, 1, 0 \rangle$ covers $\langle 0, 0, 1, 0 \rangle$ covers $\langle 0, 0, 0, 0 \rangle$.

27. If x and y denote different columns of a matrix whose rows are sorted, so that $x_i \leq y_i$ for all i , and if $x\alpha$ and $y\alpha$ denote the result of sorting the columns, the standard principle shows that $(x\alpha)_i \leq (y\alpha)_i$ for all i , since we can choose i elements of x in the same rows as any i given elements of y . [This principle was used in the text, to prove the invariance property of Shell’s sort, Theorem 5.2.1K. Further exploitation of the idea appears in an interesting paper by David Gale and R. M. Karp, *J. Computer and System Sciences* 6 (1972), 103–115. The fact that column sorting does not mess up sorted rows was apparently first observed in connection with the manipulation of tableaux; cf. Hermann Boerner, *Darstellung von Gruppen* (Springer, 1955), 137.]

28. If $\{x_{i_1}, \dots, x_{i_t}\}$ are the t largest elements, then $x_{i_1} \wedge \dots \wedge x_{i_t}$ is the t th largest. If $\{x_{i_1}, \dots, x_{i_t}\}$ are not the t largest, then $x_{i_1} \wedge \dots \wedge x_{i_t}$ is less than the t th largest.
29. $\langle x_1 \wedge y_1, (x_2 \wedge y_1) \vee (x_1 \wedge y_2), (x_3 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_1 \wedge y_3), y_1 \vee (x_3 \wedge y_2) \vee$

$(x_2 \wedge y_3) \vee (x_1 \wedge y_4), y_2 \vee (x_3 \wedge y_3) \vee (x_2 \wedge y_4) \vee (x_1 \wedge y_5), y_3 \vee (x_3 \wedge y_4) \vee (x_2 \wedge y_5) \vee x_1, y_4 \vee (x_3 \wedge y_5) \vee x_2, y_5 \vee x_3$.

30. Applying the distributive and associative laws reduces any formula to \vee 's of \wedge 's; then the commutative, idempotent, and absorption laws lead to canonical form. The S_i are precisely those sets S such that the formula is 1 when $(x_j = 1 \text{ iff } j \in S)$ while the formula is 0 when $(x_j = 1 \text{ iff } j \in S')$ for any proper subset S' of S .

31. $\delta_4 = 166$. R. Church [Duke Math. J. 6 (1940), 732–734] found $\delta_5 = 7579$, M. Ward [Bull. Amer. Math. Soc. 52 (1946), 423] found $\delta_6 = 7828352$, and W. F. Lunnon [unpublished, 1969] found $\delta_7 = 2208061288136$; the latter value has not yet been checked independently. No simple formula for δ_n is apparent; D. Kleitman [Proc. Amer. Math. Soc. 21 (1969), 677–682] proved that

$$\log_2 \delta_n \sim \binom{n}{\lfloor n/2 \rfloor},$$

using an extremely complicated argument.

32. G_{n+1} is also the set of all strings $\theta\psi$ where θ and ψ are in G_n and $\theta \leq \psi$ as vectors of 0's and 1's. It follows that G_n is the set of all strings $z_0 \dots z_{2^n-1}$ of 0's and 1's, where $z_i \leq z_j$ whenever the binary representation of i is " \leq " the binary representation of j in the 0-1 vector sense. Each element $z_0 \dots z_{2^n-1}$ of G_n , except 00...0 and 11...1, represents a $\wedge\vee$ function $f(x_1, \dots, x_n)$ from D_n into $\{0, 1\}$, under the correspondence $f(x_1, \dots, x_n) = z[(x_1 \dots x_n)_2]$.

33. If such a network existed we would have $(x_1 \wedge x_2) \vee (x_2 \wedge x_3) \vee (x_3 \wedge x_4) = f(x_1 \wedge x_2, x_1 \vee x_2, x_3, x_4)$ or $f(x_1 \wedge x_3, x_2, x_1 \vee x_3, x_4)$ or ... or $f(x_1, x_2, x_3 \wedge x_4, x_3 \vee x_4)$ for some function f . The choices $\langle x_1, x_2, x_3, x_4 \rangle = \langle x, \bar{x}, 1, 0 \rangle, \langle x, 0, \bar{x}, 1 \rangle, \langle x, 1, 0, \bar{x} \rangle, \langle 1, x, \bar{x}, 0 \rangle, \langle 1, x, 0, \bar{x} \rangle, \langle 0, 1, x, \bar{x} \rangle$ show that no such f exists.

34. Yes; after proving this, you are ready to tackle the $n = 16$ network in Fig. 49.

35. Otherwise the permutation in which only i and $i + 1$ are misplaced would never be sorted.

36. (a) Each adjacent comparator reduces the number of inversions by 0 or 1, and $\langle n, n-1, \dots, 1 \rangle$ has $\binom{n}{2}$ inversions. (b) Let $\alpha = \beta[p:p+1]$, and argue by induction on the length of α . If $p = i$, then $j > p+1$, and $(x\beta)_p > (x\beta)_j, (x\beta)_{p+1} > (x\beta)_j$; hence $(y\beta)_p > (y\beta)_j$ and $(y\beta)_{p+1} > (y\beta)_j$. If $p = i-1$, then either $(x\beta)_p$ or $(x\beta)_{p+1}$ is $> (x\beta)_j$; hence either $(y\beta)_p$ or $(y\beta)_{p+1}$ is $> (y\beta)_j$. If $p = j-1$ or j , the arguments are similar. For other p the argument is trivial. [An amusing consequence of this exercise is that if α is sorting network consisting of $\binom{n}{2}$ adjacent comparators, so is α^R (the comparators in reverse order).]

37. It suffices to show that if each comparator is replaced by an *interchange* operation we obtain a "reflection network" which transforms $\langle x_1, \dots, x_n \rangle$ into $\langle x_n, \dots, x_1 \rangle$. But in this interpretation it is not difficult to trace the route of x_k . (Note that the permutation $\pi = (1, 2)(3, 4) \dots (2n-1, 2n)(2, 3)(4, 5) \dots (2n-2, 2n-1) = (1, 3, 5, \dots, 2n-1, 2n, 2n-2, \dots, 2)$ satisfies $\pi^{n/2} = (1, 2n)(2, 2n-1) \dots (n-1, n)$.) The odd-even transposition sort was mentioned briefly by H. Seward in 1954; it has been discussed by A. Grasselli [IRE Trans. EC-11 (1962), 483] and by Kautz et al. [IEEE Trans. C-17 (1968), 443–451].

38. Let u be the smallest element of $(x\alpha)_j$, and let $y^{(0)}$ be any vector in D_n such that

$(y^{(0)})_k = 0$ implies $(x\alpha)_k$ contains an element $\leq u$, $(y^{(0)})_k = 1$ implies $(x\alpha)_k$ contains an element $> u$. If $\alpha = \beta[p:q]$, it is possible to find a vector $y^{(1)}$ satisfying the same conditions but with α replaced by β , and such that $y^{(1)}[p:q] = y^{(0)}$. Starting with $(y^{(0)})_i = 1$, $(y^{(0)})_j = 0$, we eventually have a vector $y = y^{(r)}$ satisfying the desired condition.

39. Both $(x \vee y) \vee z$ and $x \vee (y \vee z)$ represent the largest m elements of the multiset $x \sqcup y \sqcup z$; $(x \wedge y) \wedge z$ and $x \wedge (y \wedge z)$ represent the smallest m . If $x = y = z = \{0, 1\}$, $(x \wedge z) \vee (y \wedge z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) = \{0, 0\}$ while (middle elements of $\{0, 0, 0, 1, 1, 1\}$) = $(x \vee y) \wedge z = \{0, 1\}$. Sorting networks for three elements and the result of exercise 38 imply that the middle elements of $x \sqcup y \sqcup z$ may be expressed either as $((x \vee y) \wedge z) \vee (x \wedge y)$ or $((x \wedge y) \vee z) \wedge (x \vee y)$ or any other formula obtained by permuting x, y, z in these expressions. (There seems to be no "symmetrical" formula for the middle elements.)

40. Let $\alpha' = \alpha[i:j]$, and let k be an index $\neq i, j$. If $(x\alpha)_i \leq (x\alpha)_k$ for all x , then $(x\alpha')_i \leq (x\alpha')_k$; if $(x\alpha)_k \leq (x\alpha)_i$ and $(x\alpha)_k \leq (x\alpha)_j$ for all x , the same holds when α is replaced by α' ; if $(x\alpha)_k \leq (x\alpha)_i$ for all x , then $(x\alpha')_k \leq (x\alpha')_i$. In this way we see that α' has at least as many known relations as α , plus one more if $[i:j]$ isn't redundant. [Bell System Tech. J. 49 (1970), 1627–1644.]

41. (a) Let $y_{i_s} = x_{j_s}$, $y_{j_s} = x_{i_s}$, $y_k = x_k$ for $i_s \neq k \neq j_s$; then $y\alpha^s = x\alpha$. (b) This is obvious unless the set $\{i_s, j_s, i_t, j_t\}$ has only three distinct elements; suppose that $i_s = i_t$. Then if $s < t$ the first $s - 1$ comparators have (i_s, j_s, j_t) replaced, respectively, by (j_s, j_t, i_s) in both $(\alpha^s)^t$ and $(\alpha^t)^s$. (c) $(\alpha^s)^s = \alpha$, and $\alpha^1 = \alpha$, so we can assume that $s_1 > s_2 > \dots > s_k > 1$. (d) Let $\beta = \alpha[i:j]$; then $g_\beta(x_1, \dots, x_n) = (\bar{x}_i \vee x_j) \wedge (g_\alpha(x_1, \dots, x_i, \dots, x_j, \dots, x_n) \vee g_\alpha(x_1, \dots, x_j, \dots, x_i, \dots, x_n))$. Iterating this identity yields the result. (e) $f_\alpha(x) = 1$ iff no path in G_α goes from i to j where $x_i > x_j$. If α is a sorting network, the conjugates of α are also; and $f_\alpha(x) = 0$ for all x with $x_i > x_{i+1}$. Take $x = e^{(i)}$; this shows that G has an arc from i to k_1 for some $k_1 \neq i$. If $k_1 \neq i + 1$, $x = e^{(i)} \vee e^{(k_1)}$ shows that G has an arc from i or k_1 to k_2 for some $k_2 \notin \{i, k_1\}$. If $k_2 \neq i + 1$, continue in the same way until finding a path in G from i to $i + 1$. Conversely if α is not a sorting network, let x be a vector with $x_i > x_{i+1}$ and $g_\alpha(x) = 1$. Some conjugate α' has $f_{\alpha'}(x) = 1$, so $G_{\alpha'}$ can have no path from i to $i + 1$. [In general, $(x\alpha)_i \leq (x\alpha)_j$ for all x iff G_α has an oriented path from i to j for all α' conjugate to α .]

42. There must exist a path of length $\lceil \log_2 n \rceil$ or more, from some input to the largest output (consider m_n in Theorem A); when that input is set to ∞ , the comparators on this path have a predetermined behavior, and the remaining network must be an $(n - 1)$ -sorter. [IEEE Trans. on Computers C-21 (1972), 612–613.]

43.



If $P(n)$ denotes the minimum number of switches needed in a permutation network, it is clear that $P(n) \geq \lceil \log_2 n! \rceil$. A. Waksman and M. Green have proved that $P(n) \leq B(n)$ for all n , where $B(n)$ is the binary insertion function of Eq. 5.3.1–3; cf. IEEE Trans. C-17 (1968), 447; Green also has proved that $P(5) = 8$.

52. If $h[k + 1] = h[k] + 1$ and the file is not in order, something must happen to it

on the next pass; this decreases the number of inversions, by exercise 5.2.2-1, hence the file will eventually become sorted. But if $h[k+1] \geq h[k] + 2$ for $1 \leq k < m$, the smallest key will never move into its proper place if it is initially in R_2 .

53. We use the hint, and also regard $K_{N+1} = K_{N+2} = \dots = 1$. If $K_{h[1]+j} = \dots = K_{h[m]+j} = 1$ at step j , and if $K_i = 0$ for some $i > h[1] + j$, we must have $i < h[m] + j$ since there are less than n 1's. Suppose k and i are minimal such that $h[k] + j < i < h[k+1] + j$ and $K_i = 0$. Let $s = h[k+1] + j - i$; we have $s < h[k+1] - h[k] \leq k$. At step $j - s$, at least $k + 1$ 0's must have been under the heads, since $K_i = K_{h[k+1]+j-s}$ was set to zero at that step; s steps later, there are at least $k + 1 - s \geq 2$ 0's remaining between $K_{h[1]+j}$ and K_i , inclusive, contradicting the minimality of i .

The second pass gets the next $n - 1$ elements into place, etc. If we start with the permutation $N (N-1) \dots 2 1$, the first pass changes it to $(N+1-n) (N-n) \dots 1 (N+2-n) \dots (N-1) N$, since $K_{h[1]+j} \leftarrow K_{h[m]+j}$ whenever $1 \leq h[1] + j$ and $h[m] + j \leq N$; therefore the bound is best possible.

54. Suppose that $h[k+1] - s > h[k]$ and $h[k] \leq s$; the smallest key ends in position R_i for $i > 1$ if it starts in R_{n-s} . Therefore $h[k+1] \leq 2h[k]$ is necessary; it is also sufficient, by the special case $t = 0$ of the

Theorem. *If $n = N$ and if $K_1 \dots K_N$ is a permutation of $\{1, 2, \dots, n\}$, a single sorting pass will set $K_i = i$ for $1 \leq i \leq t+1$, if $h[k+1] \leq h[k] + h[k-i] + i$ for $1 \leq k < m$ and $0 \leq i \leq t$. (By convention, let $h[k] = k$ when $k \leq 0$.)*

Proof. By induction on t ; if step t does not find the key $t+1$ under the heads, we may assume that it appears in position $R_{h[k+1]+t-s}$ for some $s > 0$, where $h[k+1] - s < h[k]$; hence $h[k-t] + t - s > 0$. But this is impossible if we consider step $t-s$, which presumably placed the element $t+1$ into position $R_{h[k+1]+t-s}$ although there were at least $t+1$ lower heads active. ■

(The condition is necessary for $t = 0, 1$, but not for $t = 2$.)

55. If the numbers $\{1, \dots, 23\}$ are being sorted, the theorem in the previous exercise shows that $\{1, 2, 3, 4\}$ find their true destination. When 0's and 1's are being sorted it is possible to verify that it is impossible to have all heads reading 0 while all positions not under the heads contain 1's, at steps $-2, -1$, and 0; hence the proof in the previous exercise can be extended to show that $\{5, 6, 7\}$ find their true destination. Finally $\{8, \dots, 23\}$ must be sorted, by the argument in exercise 53.

57. When $r \leq m-2$, the heads take the string $0^r 1^1 0 1^3 0 1^7 0 \dots 0 1^{2^r-1} 0 1^q$ into $0^{r+1} 1^1 0 1^3 0 1^7 0 \dots 0 1^{2^r-1-1} 0 1^{2^r-1+q}$; hence $m-2$ passes are necessary. [When the heads are at positions $1, 2, 3, 5, \dots, 1+2^{m-2}$, Pratt has discovered a similar result: the string $0^r 1^a 0 1^{2^b-1} 0 1^{2^b+1-1} 0 \dots 0 1^{2^r-1} 0 1^q$, $1 \leq a \leq 2^{b-1}$, goes into $0^{r+1} 1^{a-1} 0 1^{2^b-1} 0 1^{2^b+1-1} 0 \dots 0 1^{2^r-1-1} 0 1^{2^r+q}$, hence at least $m - \lceil \log_2 m \rceil - 1$ passes are necessary in the worst case for this sequence of heads. The latter head sequence is of special interest since it has been used as the basis of a very ingenious sorting device invented by P. N. Armstrong [cf. U.S. Patent 3399883]. Pratt conjectures that these input sequences provide the true worst case, over all inputs.]

58. During quicksort, each key K_2, \dots, K_N is compared with K_1 ; let $A = \{i | K_i < K_1\}$, $B = \{j | K_j > K_1\}$. Subsequent operations quicksort A and B independently; all comparisons $K_i : K_j$ for i in A and j in B are suppressed, by both quicksort and the

restricted uniform algorithm, and no other comparisons are suppressed by the unrestricted uniform algorithm.

In this case we could restrict the algorithm even further, omitting Cases 1 and 2 so that arcs are added to G only when comparisons are explicitly made, yet considering only paths of length 2 when testing for redundancy. Another way to solve this problem is to consider the equivalent tree insertion sorting algorithm of Section 6.2.2, which makes precisely the same comparisons as the uniform algorithm in the same order.

59. (a) The probability that K_{a_i} is compared with K_{b_i} is the probability that c_i other specified keys do not lie between K_{a_i} and K_{b_i} ; this is the probability that two numbers chosen at random from $\{1, 2, \dots, c_i + 2\}$ are consecutive, namely

$$(c_i + 1) / \binom{c_i + 2}{2}.$$

- (b) The first $n - 1$ values of c_i are zero, then come $(n - 2)$ 1's, $(n - 3)$ 2's, etc.; hence the average is $2 \sum_{1 \leq k \leq n} (n - k)/(k + 1) = 2 \sum_{1 \leq k \leq n} ((n + 1)/(k + 1) - 1) = 2(n + 1)(H_{n+1} - 1) - 2n$. (c) The "bipartite" nature of merging shows that the restricted uniform algorithm is the same as the uniform algorithm for this sequence. The pairs involving vertex N have c 's equal to $0, 1, \dots, N - 2$, respectively; so the average number of comparisons is exactly the same as quicksort.

60. No; when $N = 5$ no pair sequence ending in $(1, 5)(1, 2)(2, 3)(3, 4)(4, 5)$ will require 10 comparisons. [An interesting research problem: For all N , find a (restricted) uniform sorting method whose worst case is as good as possible.]

62. An item can lose at most one inversion per pass, so the minimum number of passes is at least the maximum number of inversions of any item in the input permutation. The bubble sort strategy achieves this bound, since each pass decreases the inversion count of every inverted item by one (cf. 5.2.2-1). An additional pass may be needed to determine whether or not sorting is complete, but the wording of this exercise allows us to overlook such considerations.

It is perhaps unfortunate that the first result in the study of computational complexity via automata established the "optimality" of a sorting method which is so poor from a programming standpoint! The situation is analogous to the history of random number generation, which took several backward steps when generators that are "optimum" from one particular point of view were recommended for general use. The moral is that optimality results are often heavily dependent on the abstract model; although the results are very interesting, they must be applied wisely in practice.

[Demuth went on to consider a generalization to an r -register machine (saving a factor of r), and to a Turing-like machine in which the direction of scan could oscillate between left-right and right-left at will. He observed that the latter type of machine can do the straight insertion and the cocktail shaker sorts; but any such 1-register machine must go through at least $\frac{1}{4}N^2$ steps on the average, since each step reduces the total number of inversions by at most one. Finally he considered r -register random-access machines and the question of minimum-comparison sorting.]

SECTION 5.4

1. We could omit the internal sorting phase, but that would generally be much

slower since it would increase the number of times each piece of data is read and written on the external memory.

2. The runs are distributed as in (1), then Tape 3 is set to $R_1 \dots R_{2000}; R_{2001} \dots R_{4000}; R_{4001} \dots R_{5000}$. After all tapes are rewound, a “one-way merge” sets T_1 and T_2 to the respective contents of T_3 and T_4 in (2). Then T_1 and T_2 are merged to T_3 , and the information is copied back and merged once again, for a total of five passes. In general, the procedure is like the four-tape balanced merge, but with copy passes between each of the merge passes, so one less than twice as many passes are performed.
3. (a) $\lceil \log_P S \rceil$. (b) $\log_B S$, where $B = \sqrt{P(T - P)}$ is called the “effective power of the merge.” When $T = 2P$ the effective power is P ; when $T = 2P - 1$ the effective power is $\sqrt{P(P - 1)} = P - \frac{1}{2} - \frac{1}{8}P^{-1} + O(P^{-2})$, slightly less than $\frac{1}{2}T$.
4. $\frac{1}{2}T$. If T is odd and P must be an integer, both $\lceil T/2 \rceil$ and $\lfloor T/2 \rfloor$ give the same maximum value. It is best to have $P \geq T - P$, according to exercise 3, so we should choose $P = \lceil T/2 \rceil$ for balanced merging.

SECTION 5.4.1

1.
$$\begin{array}{l} \{ 503 \infty \\ 908 \infty \\ 426 653 \infty \\ 612 \infty \end{array}$$
2. The path $\textcircled{061} - \textcircled{154} - \textcircled{087} - \textcircled{512} - \boxed{061}$ would be changed to $\textcircled{087} - \textcircled{154} - \textcircled{512} - \boxed{612}$. (We are essentially doing a “bubble sort” from bottom to top along the path!)
3. and fourscore our seven years/ ago brought fathers forth on this/ a conceived continent in liberty nation new the to/ and dedicated men proposition that/ all are created equal.
4. (The problem is slightly ambiguous; in this interpretation we do not clear the internal memory until the reservoir is about to overflow.)
and fourscore on our seven this years/ ago brought continent fathers forth in liberty nation new to/ a and conceived dedicated men proposition that the/ all are created equal.
5. False, the complete binary tree with P external nodes is defined for all $P \geq 1$.
6. Insert “If $T = \text{LOC}(X[0])$ then go to R2, otherwise” at the beginning of step R6, and delete the similar clause from step R7.
7. There is no output, and RMAX stays equal to 0.
8. If the first actual key were ∞ the record would be lost. To avoid ∞ , we could set a switch, initially avoiding the test involving LASTKEY in step R4. Then when $RQ \neq 0$ in step R3 for the first time, the switch is changed so that R4 tests LASTKEY and R3 no longer tests RQ.
9. Assume, for example, that the current run is ascending, while the next should be

descending. Then the steps of Algorithm R will work properly except for one change: In step R6 if $\text{RN}(T) = \text{RQ} > \text{RC}$, reverse the test on $\text{KEY}(\text{LOSER}(T))$ vs. $\text{KEY}(Q)$.

When RC changes, the key tests of steps R4 and R6 should change appropriately.

10. Let $\cdot j \equiv \text{LOC}(X[j])$. The mechanism of Algorithm R ensures that the following conditions are true whenever we reach step R3, if we set first $\text{LOSER}(\cdot 0) \leftarrow Q$ and $\text{RN}(\cdot 0) \leftarrow \text{RQ}$: The values of $\text{LOSER}(\cdot 0), \dots, \text{LOSER}(\cdot(P-1))$ are a permutation of $\{\cdot 0, \cdot 1, \dots, \cdot(P-1)\}$; and there exists a permutation of the pointers $\{\text{LOSER}(\cdot j) | \text{RN}(\cdot j) = 0\}$ which corresponds to an actual tournament. In other words when $\text{RN}(\cdot j) = 0$, the value of $\text{KEY}(\text{LOSER}(\cdot j))$, and therefore also of $\text{LOSER}(\cdot j)$, is irrelevant; we may permute such “Losers” among themselves. After P steps all $\text{RN}(\cdot j)$ will be nonzero, so the entire tree will be consistent. (The answer to the hint is “yes.”)

11. True. (Both keys are from the same subsequence in the proof of Theorem K.)

13. The keys left in memory when the first run has ended tend to be smaller than average, since they didn’t make it into the first run. Thus the second run can output more of the smaller keys.

14. Assume that the snow suddenly stops when the snowplow is at a random point u , $0 \leq u < 1$, after it has reached its “steady state.” Then the second-last run contains $(1 + 2u - u^2)P$ records, and the last run contains u^2P . Integrating this times du yields an average time of $(2 - \frac{1}{3})P$ records in the penultimate run, $\frac{1}{3}P$ in the last.

15. False, the last run can be arbitrarily long; but only in the comparatively rare circumstance that all records in memory belong to the same run when the input is exhausted.

16. Iff each element has less than P inversions. (Cf. Sections 5.1.1, 5.4.8.) The probability is 1 when $N \leq P$, $P^{N-P}P!/N!$ when $N \geq P$, by considering inversion tables. (In actual practice, however, a one-pass sort is not too uncommon, since people tend to sort a file even when they suspect it might be in order, as a precautionary measure.)

17. Exactly $\lceil N/P \rceil$ runs, all but the last of length P . (The “worst case.”)

18. Nothing changes on the second pass, since it is possible to show that the k th record of a run is less than at least $P + 1 - k$ records of the preceding run, for $1 \leq k \leq P$. (However, there seems to be no simple way to characterize the result of P -way replacement selection followed by P' -way replacement selection when $P' > P$.)

19. Argue as in the derivation of (2) that $h(x, t) dx = KL dt$, where this time $h(x, t) = I + Kt$ for all x , and $P = IL$. This implies $x(t) = L \ln((I + Kt)/I)$, so that when $x(T) = L$ we have $KT = (e - 1)I$. The amount of snowfall since $t = 0$ is therefore $(e - 1)IL = (e - 1)P$.

20. As in exercise 19, we have $(I + Kt) dx = K(L - x) dt$; hence $x(t) = LKt/(I + Kt)$. The reservoir contents is $IL = P = P' = \int_0^T x(t)K dt = L(KT - I \ln((I + KT)/I))$, hence $KT = \alpha I$, where $\alpha \approx 2.1461$ is the root of $1 + \alpha = e^{\alpha-1}$. The run length is the total amount of snowfall during $0 \leq t \leq T$, namely $KLT = \alpha P$.

21. Proceed as in the text, but after each run wait for $P - P'$ snowflakes to fall before the plow starts out again. This means that $h(x(t), t)$ is now KT_1 , instead of KT , where $T_1 - T$ is the amount of time taken by the extra snowfall. The run length is LKT_1 , $x(t) = L(1 - e^{-t/T_1})$, $P = LKT_1 e^{-T/T_1}$, and $P' = \int_0^T x(t)K dt = P + LK(T - T_1)$. In other words, a run length of $e^\theta P$ is obtained when $P' = (1 - (1 - \theta)e^\theta)P$, for $0 \leq \theta \leq 1$.

22. For $0 \leq t \leq (\kappa - 1)T$, $dx \cdot h = K dt(x(t+T) - x(t))$, and for $(\kappa - 1)T \leq t \leq T$, $dx \cdot h = K dt(L - x(t))$, where h is seen to be constantly equal to KT at the position of the plows. It follows that for $0 \leq j \leq k$, $0 \leq u \leq 1$, and $t = (\kappa - j - u)T$, we have $x(t) = L(1 - e^{u-\theta} F_j(u)/F(\kappa))$. The run length is KTL , the amount of snowfall between the times that consecutive snowplows leave point 0 in the steady state; P is the amount cleared during each snowplow's last burst of speed, namely $KT(L - x(\kappa T)) = KTLe^{-\theta}/F(\kappa)$; and $P' = \int_0^{\kappa T} x(t)K dt$ can be shown to have the stated form.

[Notes: It turns out that the stated formulas are valid also for $k = 0$. When $k \geq 1$ the number of elements per run which go into the reservoir *twice* is $P'' = \int_0^{(\kappa-1)T} x(t)K dt$, and it is easy to show that (run length) $- P' + P'' = (e - 1)P$, a phenomenon noticed by Frazer and Wong. Is it a coincidence that the generating function for $F_k(\theta)$ is so similar to that in exercise 5.1.3-11?]

23. Let $P = pP'$ and $q = 1 - p$. For the first T_1 units of time the snowfall comes from the qP' elements remaining in the reservoir after the first pP' have been initially removed in random order; and when the old reservoir is empty, uniform snow begins to fall again. We choose T_1 so that $KLT_1 = qP'$. For $0 \leq t \leq T_1$, $h(x, t) = (p + qt/T_1)g(x)$, where $g(x)$ is the height of snow put into the reservoir from position x ; for $T_1 \leq t \leq T$, $h(x, t) = g(x) + (t - T_1)K$. For $0 \leq t \leq T_1$, $g(x(t))$ is $(q(T_1 - t)/T_1)g(x(t)) + (T - T_1)K$; and for $T_1 \leq t \leq T$, $g(x(t)) = (T - t)K$. Hence $h(x(t), t) = (T - T_1)K$ for $0 \leq t \leq T$, and $x(t) = L(1 - \exp(-t/(T - T_1)))$. The total run length is $(T - T_1)KL$; the total amount "recycled" from the reservoir back again (cf. exercise 22) is T_1KL ; and the total amount cleared after time T is $P = KT(L - x(T))$.

So the assumptions of this exercise give runs of length $(e^s/s)P$ when the reservoir size is $(1 + (s - 1)e^s/s)P$. This is considerably worse than the results of exercise 22, since the reservoir contents are being used in a more advantageous order in that case.

(The fact that $h(x(t), t)$ is constant in so many of these problems is not surprising, since it is equivalent to saying that the elements of each run obtained during a steady state of the system are uniformly distributed.)

24. (a) Essentially the same proof works, each of the subsequences has runs in the same direction as the output runs. (b) The stated probability is the probability that the run has length $n + 1$ and is followed by y ; it equals $(1 - x)^n/n!$ when $x > y$, and it is $(1 - x)^n/n! - (y - x)^n/n!$ when $x \leq y$. (c) Induction. For example if the n th run is ascending, the $(n - 1)$ st was descending with probability p , so the first integral applies. (d) We find that $f'(x) = f(x) - c - pf(1 - x) - qf(x)$, then $f''(x) = -2pc$, which ultimately leads to $f(x) = c(1 - qx - px^2)$, $c = 6/(3 + p)$. (e) If $p > eq$ then $pe^x + qe^{1-x}$ is monotone increasing for $0 \leq x \leq 1$, and $\int_0^1 |pe^x + qe^{1-x} - e^{1/2}| dx = (p - q)(e^{1/2} - 1)^2 < 0.43$. If $q \leq p < eq$ then $pe^x + qe^{1-x}$ lies between $2\sqrt{pqe}$ and $p + qe$, so $\int_0^1 |pe^x + qe^{1-x} - \frac{1}{2}(p + qe + 2\sqrt{pqe})| dx \leq \frac{1}{2}(\sqrt{p} - \sqrt{qe})^2 < 0.4$; and if $p < q$ we may use a symmetrical argument. Thus for all p and q there is a constant C such that $\int_0^1 |pe^x + qe^{1-x} - C| dx < 0.43$. Let $\delta_n(x) = f_n(x) - f(x)$. Then $\delta_{n+1}(y) = (1 - e^{y-1}) \int_0^1 (pe^x + qe^{1-x} - C)\delta_n(x) dx + p \int_0^{1-y} e^{y-1+x}\delta_n(x) dx + q \int_y^1 e^{y-x}\delta_n(x) dx$; hence if $\delta_n(y) \leq \alpha_n$, $|\delta_{n+1}(y)| \leq (1 - e^{y-1})(1.43)\alpha_n < 0.91\alpha_n$. (f) For all $n \geq 0$, $(1 - x)^n/n!$ is the probability that the run length is $> n$. (g) $\int_0^1 (pe^x + qe^{1-x})f(x) dx = 6/(3 + p)$.

26. (a) Consider the number of permutations with $n + r + 1$ elements and n left-to-right minima, where the rightmost element is not the smallest. (b) Use the fact that

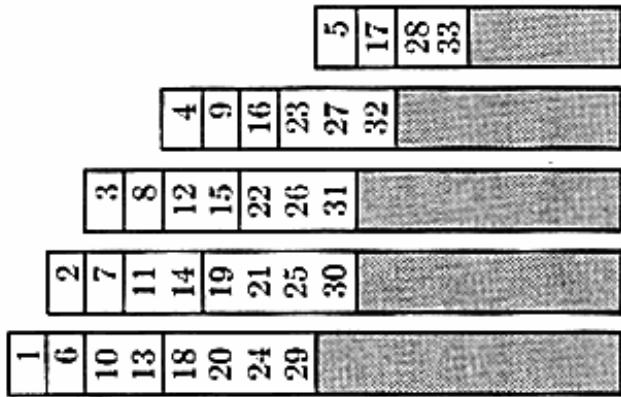
$$\sum_{1 \leq k < n} \left[\begin{matrix} k \\ k-r \end{matrix} \right] k = \left[\begin{matrix} n \\ n-r-1 \end{matrix} \right],$$

by the definition of Stirling numbers in the Index to Notations. (c) Add $r+1$ to the mean, using the fact that $\sum_{n \geq 0} \left[\begin{matrix} n+r \\ n \end{matrix} \right] (n+r)/(n+r+1)! = 1$, to get $\sum_{n \geq 0} \left[\begin{matrix} n+r \\ n \end{matrix} \right] / (n+r-1)!.$

27. For multiway merging there is comparatively little problem, since P stays constant and records are processed sequentially on each file; but when forming initial runs, we would like to vary the number of records in memory depending on their lengths. We could keep a heap of as many records as will fit in memory, using dynamic storage allocation as described in Section 2.5. M. A. Goetz [Proc. AFIPS Joint Computer Conf. 25 (1964), 602–604] has suggested another approach, breaking each record into fixed-size parts which are linked together; they occupy space at the leaves of the tree, but only the leading part participates in the tournament.

SECTION 5.4.2

1.



2. After the first merge phase, all remaining dummies are on tape T , and there are at most $a_n - a_{n-1} \leq a_{n-1}$ of them. Therefore they all disappear during the second merge phase.

3. We have $(D[1], D[2], \dots, D[T]) = (a_n - a_{n-p}, a_n - a_{n-p+1}, \dots, a_n - a_n)$, so the condition follows from the fact that the a 's are nondecreasing. The condition is important to the validity of the algorithm, since steps D2 and D3 never decrease $D[j+1]$ more often than $D[j]$.

4. $(1 - z - \dots - z^5)a(z) = 1$ because of (3). Also, $t(z) = \sum_{n \geq 1} (a_n + b_n + c_n + d_n + e_n)z^n = (z + \dots + z^5)a(z) + (z + \dots + z^4)a(z) + \dots + za(z) = (5z + 4z^2 + 3z^3 + 2z^4 + z^5)a(z)$. [Cf. (16).]

5. Let $g_p(z) = (z - 1)f_p(z) = z^{p+1} - 2z^p + 1$, and let $h_p(z) = z^{p+1} - 2z^p$. Let $\phi^{-1} > \epsilon > 0$; Rouché's theorem tells us that $h_p(z)$ and $g_p(z)$ have the same number of roots inside the circle $|z| = 1 + \epsilon$, provided $|h_p(z) - g_p(z)| > |h_p(z) - g_p(z)| = 1$ on the circle. We have $|h_p(z)| \geq (1 + \epsilon)^p(1 - \epsilon) > (1 + \phi^{-1})^2(1 - \phi^{-1}) = 1$. Hence g_p has p roots of magnitude ≤ 1 . They are distinct, since $\gcd(g_p(z), g'_p(z)) = \gcd(g_p(z), g'_p(z)) = 1$. [AMM 67 (1960), 745–752.]

6. Let $c_0 = -\alpha p(\alpha^{-1})/q'(\alpha^{-1})$. Then $p(z)/q(z) - c_0/(1 - \alpha z)$ is analytic in $|z| \leq R$ for some $R > |\alpha|^{-1}$; hence the coefficient of z^n in $p(z)/q(z)$ is $c_0\alpha^n + O(R^{-n})$. Thus, $\ln S = n \ln \alpha + \ln c_0 + O((\alpha R)^{-n})$; and $n = (\ln S/\ln \alpha) + O(1)$ implies that $O((\alpha R)^{-n}) = O(S^{-\epsilon})$. Similarly, let $c_1 = \alpha^2 p(\alpha^{-1})/q'(\alpha^{-1})^2$, $c_2 = -\alpha p'(\alpha^{-1})/q'(\alpha^{-1})^2 + \alpha p(\alpha^{-1})q''(\alpha^{-1})/q'(\alpha^{-1})^3$, and consider $p(z)/q(z)^2 - c_1/(1 - \alpha z)^2 - c_2/(1 - \alpha z)$.

7. Let $\alpha_p = 2(1 - t)$, and $z = 2^{-p-1}$. Then $z = t(1 - t)^p$, and reversion of this series (Algorithm 4.7R) gives a power series for t in terms of z , yielding the asymptotic expansion $\alpha_p = 2 - 2^{-p} - p2^{-2p-1} + O(p^2 2^{-3p})$.

Note: It follows that the quantity ρ in exercise 6 becomes approximately $\log_4 S$ as p increases. Similarly, for both Table 5 and Table 6, the coefficient c approaches $1/((\phi + 2) \ln \phi)$ on a large number of tapes.

8. Evidently $N_0^{(p)} = 1$, $N_m^{(p)} = 0$ for $m < 0$, and by considering the different possibilities for the first summand we have $N_m^{(p)} = N_{m-1}^{(p)} + \dots + N_{m-p}^{(p)}$ when $m > 0$. Hence $N_m^{(p)} = F_{m+p-1}^{(p)}$. [Lehrbuch der Combinatorik (Leipzig: Teubner, 1901), 136–137.]

9. Consider the position of the leftmost 0, if there is one; we find $K_m^{(p)} = F_{m+p}^{(p)}$. *Note:* There is a simple one-to-one correspondence between such sequences of 0's and 1's and the representations of $m+1$ considered in exercise 8: Place a 0 at the right end of the sequence, and look at the positions of all the 0's.

10. *Lemma:* If $n = F_{j_1}^{(p)} + \dots + F_{j_m}^{(p)}$ is such a representation, with $j_1 > \dots > j_m \geq p$, we have $n < F_{j_1+1}^{(p)}$. *Proof:* The result is obvious if $m < p$; otherwise let k be minimal with $j_k > j_{k+1} + 1$; we have $k < p$, and by induction $F_{j_{k+1}}^{(p)} + \dots + F_{j_m}^{(p)} < F_{j_{k+1}-1}^{(p)}$, hence $n < F_{j_1}^{(p)} + \dots + F_{j_1-p+1}^{(p)} \leq F_{j_1+1}^{(p)}$.

The stated result can now be proved, by induction on n . If $n > 0$ let j be maximal such that $F_j^{(p)} \leq n$. The lemma shows that each representation of n must consist of $F_j^{(p)}$ plus a representation of $n - F_j^{(p)}$. By induction, $n - F_j^{(p)}$ has a unique representation of the desired form, and this representation does not include all of the numbers $F_{j-1}^{(p)}, \dots, F_{j-p+1}^{(p)}$ because j is maximal.

Notes: The case $p = 2$, which is due to E. Zeckendorf [cf. *Simon Stevin* 29 (1952), 190–195], has been considered in exercise 1.2.8–34. There is a simple algorithm to go from the representation of n to that of $n+1$, working on the sequence $c_j \dots c_1 c_0$ of 0's and 1's such that $n = \sum c_j F_{j+p}^{(p)}$: For example, if $p = 3$, we look at the rightmost digits, changing $\dots 0$ to $\dots 1$, $\dots 01$ to $\dots 10$, $\dots 011$ to $\dots 100$; then we “carry” to the left if necessary, replacing “ $\dots 0111 \dots$ ” by “ $\dots 1000 \dots$ ”. See the sequences of 0's and 1's in exercise 9, in the order listed. A similar number system has been studied by W. C. Lynch [*Fibonacci Quarterly* 8 (1970), 6–22], who found a very interesting way to make it govern both the distribution and merge phases of a polyphase sort.

12. The k th power contains the perfect distributions for levels $k-4$ through k , on successive rows, with the largest elements to the right.

13. By induction on the level.

14. (a) $n(1) = 1$, so assume that $k > 1$. The law $T_{nk} = T_{n-1,k-1} + \dots + T_{n-p,k-1}$ shows that $T_{nk} \leq T_{n+1,k}$ iff $T_{n-p,k-1} \leq T_{n,k-1}$. Let r be any positive integer, and let n' be minimal such that $T_{n'-r,k-1} > T_{n',k-1}$; then $T_{n-r,k-1} \geq T_{n,k-1}$ for all $n \geq n'$, since it is trivial for $n \geq n(k-1) + r$ and otherwise $T_{n-r,k-1} \geq T_{n'-r,k-1} \geq T_{n',k-1} \geq T_{n,k-1}$. (b) The same argument with $r = n - n'$ shows that $T_{n',k'} < T_{nk'}$.

implies $T_{n'-j,k'} \leq T_{n-j,k'}$ for all $j \geq 0$; hence the recurrence implies that $T_{n'-j,k} \leq T_{n-j,k}$ for all $j \geq 0$ and $k \geq k'$. (c) Let $\ell(S)$ be the least n such that $\Sigma_n(S)$ assumes its minimum value. The sequence M_n exists as desired iff $\ell(S) \leq \ell(S+1)$ for all S . Suppose $n = \ell(S) > \ell(S+1) = n'$, so that $\Sigma_n(S) < \Sigma_{n'}(S)$ and $\Sigma_n(S+1) \geq \Sigma_{n'}(S+1)$. There is some smallest S' such that $\Sigma_n(S') < \Sigma_{n'}(S')$, and we have $m = \Sigma_n(S') - \Sigma_n(S' - 1) < \Sigma_{n'}(S') - \Sigma_{n'}(S' - 1) = m'$. Then $\sum_{1 \leq k \leq m} T_{n'k} < S' \leq \sum_{1 \leq k \leq m} T_{nk}$; hence there is some $k' \leq m$ such that $T_{n'k'} < T_{nk'}$. Similarly we have $l = \Sigma_n(S+1) - \Sigma_n(S) > \Sigma_{n'}(S+1) - \Sigma_{n'}(S) = l'$; hence $\sum_{1 \leq k \leq l'} T_{n'k} \geq S+1 > \sum_{1 \leq k \leq l'} T_{nk}$. Since $l' \geq m' > m$, there is some $k > m$ such that $T_{n'k} > T_{nk}$. But this contradicts part (b).

15. This theorem has been proved by D. A. Zave, whose article was cited in the text.

16. D. A. Zave has shown that the number of records input (and output) is $S \log_{T-1} S + O(S\sqrt{\log S})$.

17. Let $T = 3$; $A_{11}(x) = 6x^6 + 35x^7 + 56x^8 + \dots$, $B_{11}(x) = x^6 + 15x^7 + 35x^8 + \dots$, $T_{11}(x) = 7x^6 + 50x^7 + 91x^8 + 64x^9 + 19x^{10} + 2x^{11}$. The optimum distribution for $S = 144$ requires 55 runs on T2, and this forces a nonoptimum distribution for $S = 145$. D. A. Zave has studied near-optimum procedures of this kind.

18. Let $S = 9$, $T = 3$, and consider the following two patterns.

Optimum Polyphase:				Alternative:			
T1	T2	T3	Cost	T1	T2	T3	Cost
0^21^6	0^21^3	—		0^11^6	0^11^3	—	
1^3	—	0^22^3	6	1^3	—	0^12^3	6
—	1^23^1	2^2	5	—	1^13^2	2^1	7
3^2	3^1	—	6	3^1	3^2	—	3
3^1	—	6^1	6	—	3^1	6^1	6
—	9^1	—	9	9^1	—	—	9
			<u>32</u>				<u>31</u>

(Still another way to improve on "optimum" polyphase is to reconsider where dummy runs appear on the output tape of every merge phase. For example, the result of merging 0^21^3 with 0^21^3 might be regarded as $2^10^12^10^12^1$ instead of 0^22^3 . Thus, many unresolved questions of optimality remain.)

19.	Level	T1	T2	T3	T4	Total	Final output on
0	1	0	0	0	0	1	T1
1	0	1	1	1	1	3	T6
2	1	1	1	0	0	3	T5
3	1	2	1	1	1	5	T4
4	2	2	2	1	1	7	T3
5	2	4	3	2	2	11	T2
6	4	5	4	2	2	15	T1
7	5	8	6	4	4	25	T6
...
n	a_n	b_n	c_n	d_n	t_n		$T(k)$
$n+1$	b_n	$c_n + a_n$	$d_n + a_n$	a_n	$t_n + 2a_n$		$T(k-1)$

20. $a(z) = 1/(1 - z^2 - z^3 - z^4)$, $t(z) = (3z + 3z^2 + 2z^3 + z^4)/(1 - z^2 - z^3 - z^4)$, $\sum_{n \geq 1} T_n(x)z^n = x(3z + 3z^2 + 2z^3 + z^4)/(1 - x(z^2 + z^3 + z^4))$. $D_n = A_{n-1} + 1$, $C_n = A_{n-1}A_{n-2} + 1$, $B_n = A_{n-1}A_{n-2}A_{n-3} + 1$, $A_n = A_{n-2}A_{n-3}A_{n-4} + 1$.

21. 333343333332322 3333433333323 333343333333 33334333 333323 T5

22. $t_n - t_{n-1} - t_{n-2} = 2$ when $n \bmod 3 = 1$, and it is -1 otherwise. (This Fibonacci-like relation follows from the fact that $1 - z^2 - 2z^3 - z^4 = (1 - \phi z)(1 - \hat{\phi} z)(1 - \omega z)(1 - \bar{\omega} z)$, where $\omega^3 = 1$.)

23. In place of (25), the run lengths during the first half of the n th merge phase are s_n , and on the second half they are t_n , where

$$s_n = t_{n-2} + t_{n-3} + s_{n-3} + s_{n-4}, \quad t_n = t_{n-2} + s_{n-2} + s_{n-3} + s_{n-4}.$$

Here we regard $s_n = t_n = 1$ for $n \leq 0$. [In general, if v_{n+1} is the sum of the first $2r$ terms of $u_{n-1} + \dots + v_{n-P}$, we have $s_n = t_n = t_{n-2} + \dots + t_{n-r} + 2t_{n-r-1} + t_{n-r-2} + \dots + t_{n-P}$; if v_{n+1} is the sum of the first $2r-1$, we have $s_n = t_{n-2} + \dots + t_{n-r-1} + s_{n-r-1} + \dots + s_{n-P}$, $t_n = t_{n-2} + \dots + t_{n-r} + s_{n-r} + \dots + s_{n-P}$.]

In place of (27) and (28), $A_n = (U_{n-1}V_{n-1}U_{n-2}V_{n-2}U_{n-3}V_{n-3}U_{n-4}V_{n-4}) + 1$, \dots , $D_n = (U_{n-1}V_{n-1}) + 1$, $E_n = (U_{n-2}V_{n-2}U_{n-3}) + 1$; $V_{n+1} = (U_{n-1}V_{n-1}U_{n-2}) + 1$, $U_n = (V_{n-2}U_{n-3}V_{n-3}U_{n-4}V_{n-4}) + 1$.

25.	1^{16}	1^8	—	1^8
	1^{12}	1^4	R	1^82^4
	1^8	—	2^4	R

	R	8^116^1	8^1	8^0
	16^0	R	8^1	—
	16^1	16^1	8^0	R
	R	16^1	—	24^0
	16^1	16^1	R	24^032^0
	16^0	16^0	32^1	(R)

26. When 2^n are sorted, $n \cdot 2^n$ initial runs are processed while merging; each half phase (with few exceptions) merges 2^{n-2} and rewinds 2^{n-1} . When $2^n + 2^{n-1}$ are sorted, $n \cdot 2^n + (n-1) \cdot 2^{n-1}$ initial runs are processed while merging; each half phase (with few exceptions) merges 2^{n-2} or 2^{n-1} and rewinds $2^{n-1} + 2^{n-2}$.

27. It works if and only if the gcd of the distribution numbers is 1. For example, let there be six tapes; if we distribute (a, b, c, d, e) to T1 through T5, where $a \geq b \geq c \geq d \geq e > 0$, the first phase leaves a distribution $(a - e, b - e, c - e, d - e, e)$, and $\gcd(a - e, b - e, c - e, d - e, e) = \gcd(a, b, c, d, e)$. (Any common divisor of one set of numbers divides the others too.) The process decreases the number of runs at each phase until $\gcd(a, b, c, d, e)$ runs are left on a single tape.

[Note that these nonpolyphase distributions sometimes turn out to be superior to polyphase under certain configurations of dummy runs, as shown in exercise 18. This property was first observed by B. Sackman about 1963.]

28. We get any such (a, b, c, d, e) by starting with $(1, 0, 0, 0, 0)$ and doing the following operation exactly n times: Choose x in $\{a, b, c, d, e\}$, and add x to each of the other four elements of (a, b, c, d, e) .

To show that $a + b + c + d + e \leq t_n$, we shall prove by induction that if $a \geq b \geq c \geq d \geq e$ we always have $a \leq a_n$, $b \leq b_n$, $c \leq c_n$, $d \leq d_n$, $e \leq e_n$. Assuming that this holds for level n , it must hold also for level $n+1$, since the level $n+1$

distributions are $(b+a, c+a, d+a, e+a, a)$, $(a+b, c+b, d+b, e+b, b)$,
 $(a+c, b+c, d+c, e+c, c)$, $(a+d, b+d, c+d, e+d, d)$, $(a+e, b+e, c+e, d+e, e)$.

30. The following table has been computed by J. A. Mortenson.

Level	$T = 5$	$T = 6$	$T = 7$	$T = 8$	$T = 9$	$T = 10$	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}
1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	4	5	6	7	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9
3	4	5	6	7	8	8	9	9	9	9	9	9	9	9	9	9	9	9	9	9
4	8	8	10	12	14	16	17	18	19	20	21	22	23	24	25	26	27	28	29	29
5	10	14	18	21	26	32	46	47	56	56	56	56	56	56	56	56	56	56	56	56
6	18	20	26	32	46	53	74	82	92	92	92	92	92	92	92	92	92	92	92	92
7	26	32	46	53	74	92	122	122	122	122	122	122	122	122	122	122	122	122	122	122
8	44	53	74	92	122	122	122	122	122	122	122	122	122	122	122	122	122	122	122	122
9	68	83	100	122	144	171	206	234	264	294	324	354	384	414	444	474	504	534	564	594
10	112	134	156	178	200	227	254	281	308	335	362	389	416	443	470	497	524	551	578	605
11	178	197	216	235	254	273	292	311	330	349	368	387	406	425	444	463	482	501	520	539
12	290	350	410	488	566	640	714	788	852	917	981	1045	1109	1173	1237	1291	1345	1399	1453	1507
13	466	566	640	733	833	933	1033	1133	1233	1333	1433	1533	1633	1733	1833	1933	2033	2133	2233	2333
14	756	917	1371	1769	2137	2505	2873	3241	3609	3977	4345	4713	5081	5449	5817	6185	6553	6921	7289	7657

SECTION 5.4.3

1. Comparing the average number of times each record is processed, in Table 5.4.2-6, the tape-splitting polyphase is superior when there are 6, 7, or 8 tapes.
2. The methods are essentially identical when the number of initial runs is a Fibonacci number; but the manner of distributing dummy runs in other cases is better with polyphase. The cascade algorithm puts 1 on T1, then 1 on T2, 1 on T1, 2 on T2, 3 on T1, 5 on T2, etc. and step D8 never finds $D[p - 1] = M[p - 1]$ when $p = 2$. In effect, all dummies are on one tape, and this is less efficient than the method of Algorithm 5.4.2D.
3. (Distribution stops after putting 12 runs on T3 during Step (3, 3).)

	T1	T2	T3	T4	T5	T6
1 ²⁶	1 ²¹	1 ²⁴	1 ¹⁴	1 ¹⁵	—	—
1 ⁵	—	1 ¹²	1 ²² 7	1 ¹⁵	2 ²⁴ 12	—
8 ⁴	6 ² 9 ³	5 ²	6 ³	1 ¹	—	—
—	9 ¹	23 ¹	17 ¹	25 ¹	26 ¹	—
100 ¹	—	—	—	—	—	—

4. Induction. (Cf. exercise 5.4.2-28.)

5. When there are a_n initial runs, the k th pass outputs a_{n-k} runs of length a_k , then b_{n-k} of length b_k , etc.

6. / 1 1 1 1 1 \ / 1 1 1 1 0 \ / 1 1 1 0 0 \ / 1 1 0 0 0 \ / 1 0 0 0 0 \

7. $e_2e_{n-2} + e_3e_{n-3} + \dots + e_ne_0$ initial run lengths (cf. exercise 5), which may also be written $a_1a_{n-3} + a_2a_{n-4} + \dots + a_{n-2}a_0$; it is the coefficient of z^{n-2} in $A(z)^2 - A(z)$.

8. The denominator of $A(z)$ has distinct roots and greater degree than the numerator, hence $A(z) = \sum q_3(\rho)/(1 - \rho z)\rho(1 - q'_4(\rho))$ summed over all roots ρ of $q_4(\rho) = \rho$. The special form of ρ is helpful in evaluating $q_3(\rho)$ and $q'_4(\rho)$.

9. The formulas hold for all large n , by (8) and (12), in view of the value of $q_m(2 \sin \theta_k)$. To show that they hold for all n we need to know that $q_{m-1}(z)$ is the quotient when $q_{r-1}(z)q_m(z)$ is divided by $q_r(z) - z$, for $0 \leq m < r$. This can be proved either by (a) using (10) and noting that cancellations bring down the degree of $q_{r-1}(z)q_m(z) - q_r(z)q_{m-1}(z)$; or (b) the result of exercise 5 implies that $A(z)^2 + B(z)^2 + \dots + E(z)^2 \rightarrow 0$ as $z \rightarrow \infty$; or (c) it is possible to find an explicit formula for the numerators of $B(z)$, $C(z)$, etc.

10. $E(z) = r_1(z)A(z)$; $D(z) = r_2(z)A(z) - r_1(z)$; $C(z) = r_3(z)A(z) - r_2(z)$; $B(z) = r_4(z)A(z) - r_3(z)$; $A(z) = r_5(z)A(z) + 1 - r_4(z)$. Thus $A(z) = (1 - r_4(z))/(1 - r_5(z))$. [Note that $r_m(2 \sin \theta) = \sin(2m\theta)/\cos \theta$; $r_m(z)$ is the Chebyshev polynomial $(-1)^{m+1}U_{2m-1}(z/2)$.]

11. Prove that $f_m(z) = q_{[m/2]}(z) - r_{[m/2]}(z)$ and that $f_m(z)f_{m-1}(z) = 1 - r_m(z)$. Then use the result of exercise 10.

SECTION 5.4.4

1. When writing an ascending run, *first* write a sentinel record containing $-\infty$ before outputting the run. (And a $+\infty$ sentinel should be written at the end of the run as well, if the output is ever going to be read forward, e.g., on the final pass.) For descending runs, interchange the roles of $-\infty$ and $+\infty$.
2. The smallest number on level $n + 1$ is equal to the largest on level n ; hence the columns are nondecreasing, regardless of the way we permute the numbers in any particular row.

3. In fact, during the merge process the first run on T2–T6 will always be descending, and the first on T1 will always be ascending. (By induction.)

4. It requires several “copy” operations on the second and third phases; the approximate extra cost is $(\log 2)/(\log \rho)$ passes, where ρ is the “growth ratio” in Table 5.4.2–1.
5. If α is a string, let α^R denote its left-right reversal.

Level	T1	T2	T3	T4	T5	
0	0	—	—	—	—	2
1	1	1	1	1	1	3 2
2	12	12	12	12	2	3 4 3 2
3	1232	1232	1232	232	32	4 5 4 3 2
4	12323432	12323432	2323432	323432	3432	5 4 4 3 2 4
...
n	A_n	B_n	C_n	D_n	E_n	F_n
$n+1$	$B_n(A_n^R + 1)$	$C_n(A_n^R + 1)$	$D_n(A_n^R + 1)$	$E_n(A_n^R + 1)$	$A_n^R + 1$	3 3 3 3 3 3

$$\begin{aligned}
E_n &= A_{n-1}^R + 1, \\
D_n &= A_{n-2}^R A_{n-1}^R + 1, \\
C_n &= A_{n-3}^R A_{n-2}^R A_{n-1}^R + 1, \\
B_n &= A_{n-4}^R A_{n-3}^R A_{n-2}^R A_{n-1}^R + 1, \\
A_n &= A_{n-5}^R A_{n-4}^R A_{n-3}^R A_{n-2}^R A_{n-1}^R + 1 = n - Q_n,
\end{aligned}$$

where

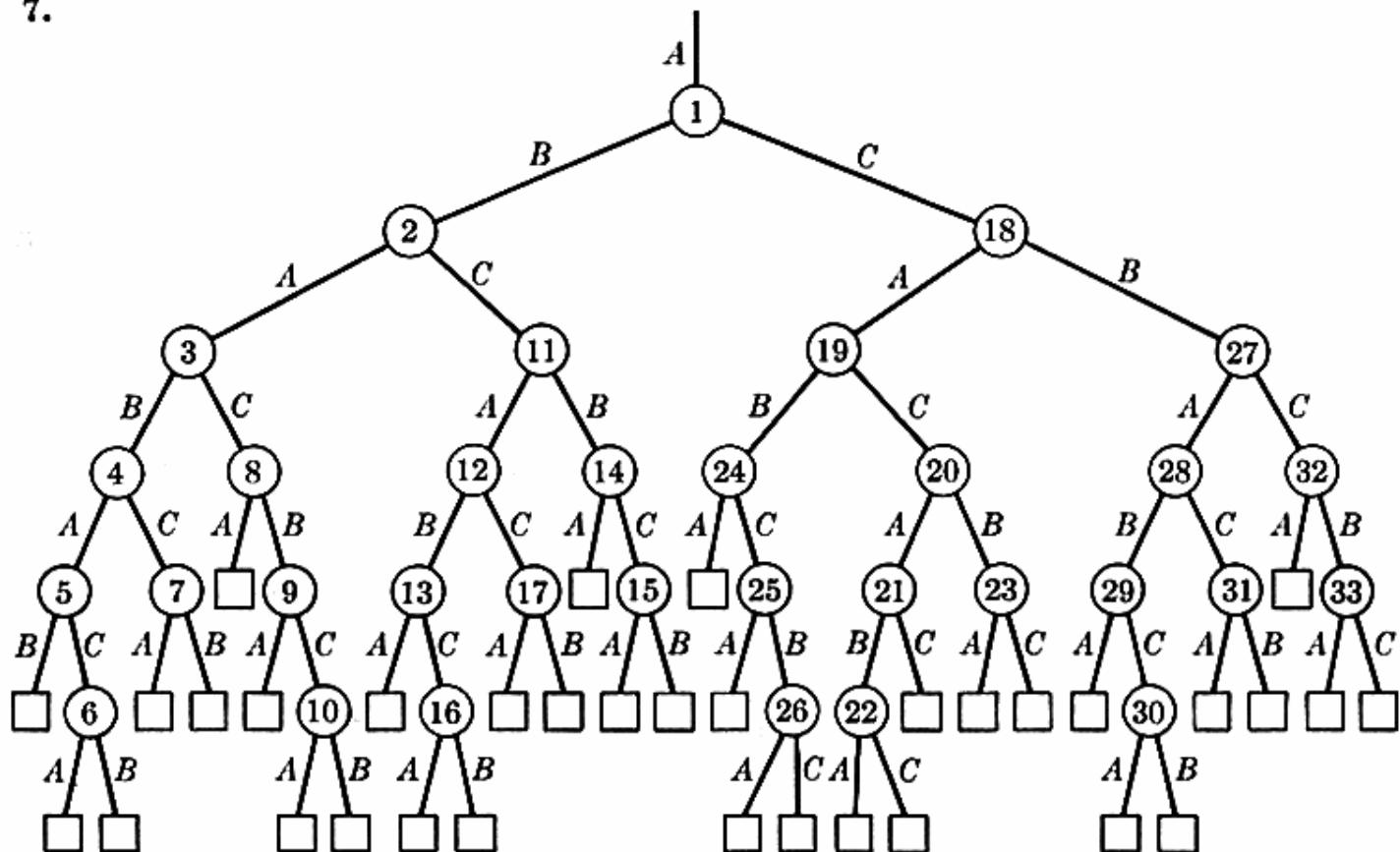
$$Q_n^R = Q_{n-1}(Q_{n-2} + 1)(Q_{n-3} + 2)(Q_{n-4} + 3)(Q_{n-5} + 4), \quad n \geq 1.$$

$Q_0 = '0'$, and Q_n is null for $n < 0$. In place of 5.4.2-12 we could consider a doubly infinite string which contains $Q_1^R, Q_2, Q_3^R, Q_4, Q_5^R$, etc. near its center.

These strings A_n, B_n , etc. contain the same entries as the corresponding strings in Section 5.4.2, but in another order. Note that adjacent merge numbers always differ by 1. An initial run must be A iff its merge number is even, D iff odd. Simple distribution schemes such as Algorithm 5.4.2D are not quite as effective at placing dummies into high-merge-number positions; therefore it is probably advantageous to compute Q_n between phases 1 and 2, in order to help control dummy run placement (assuming S is small enough that Q_n will fit comfortably in memory).

6. $y^{(4)} = (+1, +1, -1, +1)$
- $y^{(3)} = (+1, 0, -1, 0)$
- $y^{(2)} = (+1, -1, +1, +1)$
- $y^{(1)} = (-1, +1, +1, +1)$
- $y^{(0)} = (-1, 0, 0, 0)$

7.



Incidentally, 34 is apparently the smallest Fibonacci number F_n for which polyphase doesn't produce the optimum read-backwards merge for F_n initial runs on three tapes. This tree has external path length 178, which beats polyphase by two.

8. For $T = 4$, the tree with external path length 13 is not T -lifo, and every tree with external path length 14 includes a one-way merge.

9. We may consider a complete $(T - 1)$ -ary tree, by the result of exercise 2.3.4.5-6; the degree of the “last” internal node is between 2 and $T - 1$. When there are $(T - 1)^q - m$ external nodes, $\lfloor m/(T - 2) \rfloor$ of them are on level $q - 1$, and the rest are on level q .

11. True by induction on the number of initial runs. If there is a valid distribution with S runs and two adjacent runs in the same direction, then there is one with $< S$ runs; but there is none when $S = 1$.

12. Conditions (a), (b) are obvious. If either configuration in (4) is present, for some tape name A and some $i < j < k$, node j must be in a subtree below node i and to the left of node k , by the definition of preorder. Hence the “ $j - l$ ” case can’t be present, and A must be the “special” name since it appears on an external branch. But this contradicts the fact that the special name is supposed to be on the leftmost branch below node i .

13. Nodes now numbered 4, 7, 11, 13 could be external instead of one-way merges. (This gives an external path length one higher than the polyphase tree.)

15. Let the tape names be A , B , and C . We shall construct several species of trees, botanically identified by their root and leaf (external node) structure:

Type $r(A)$. Root A .

Type $s(A, C)$. Root A , no C leaves.

Type $t(A)$. Root A , no A leaves.

Type $u(A, C)$. Root A , no C leaves, no compound B leaves.

Type $v(A, C)$. Root A , no C leaves, no compound A leaves.

Type $w(A, C)$. Root A , no A leaves, no compound C leaves.

A “compound leaf” is a leaf whose brother is not a leaf. We can grow a 3-lifo type $r(A)$ tree by first growing its left subtree as a type $s(B, C)$, then growing the right subtree as type $r(C)$. Similarly, type $s(A, C)$ comes from a type $s(B, C)$ then $t(C)$; type $u(A, C)$ from $v(B, C)$ and $w(C, B)$; type $v(A, C)$ from $u(B, C)$ and $w(C, A)$. We can grow a 3-lifo type $t(A)$ tree whose left subtree is type $u(B, A)$ and whose right subtree is type $s(C, A)$, by first letting the left subtree grow except for its (non-compound) C leaves and its right subtree; at this point the left subtree has only A and B leaves, so we can grow the right subtree of the whole tree, then grow off the A leaves of the left left subtree, and finally grow the left right subtree. Similarly, a type $w(A, C)$ tree can be fabricated from a $u(B, A)$ and a $v(C, A)$. [The tree of exercise 7 is an $r(A)$ tree constructed in this manner.]

Let $r(n), \dots, w(n)$ denote the minimum external path length over all n -leaf trees of the relevant type, when they are constructed by such a procedure. We have $r(1) = s(1) = u(1) = 0$, $r(2) = t(2) = w(2) = 2$, $t(1) = v(1) = w(1) = s(2) = u(2) = v(2) = \infty$; and for $n \geq 3$,

$$\begin{aligned} r(n) &= n + \min_k (s(k) + r(n - k)), & u(n) &= n + \min_k (v(k) + w(n - k)), \\ s(n) &= n + \min_k (s(k) + t(n - k)), & v(n) &= n + \min_k (u(k) + w(n - k)), \\ t(n) &= n + \min_k (u(k) + s(n - k)), & w(n) &= n + \min_k (u(k) + v(n - k)). \end{aligned}$$

It follows that $r(n) \leq s(n) \leq u(n)$, $s(n) \leq v(n)$, and $r(n) \leq t(n) \leq w(n)$ for all n ; furthermore $s(2n) = t(2n+1) = \infty$. (The latter is evident *a priori*.)

Let $A(n)$ be the function defined by the laws $A(1) = 0$, $A(2n) = 2n + 2A(n)$, $A(2n+1) = 2n+1+A(n)+A(n+1)$; then $A(2n) = 2n+A(n-1)+A(n+1)-1$ or 0 for all $n \geq 2$. Let C be a constant such that, for $4 \leq n \leq 8$,

- i) n even implies that $w(n) \leq A(n) + Cn - 1$.
- ii) n odd implies that $u(n)$ and $v(n)$ are $\leq A(n) + Cn - 1$.

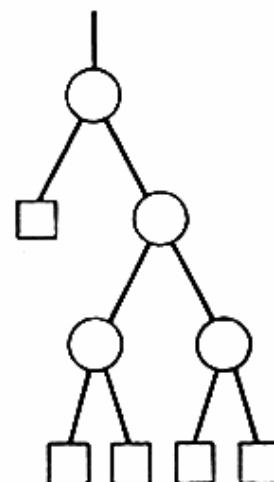
(This actually works for all $C \geq \frac{5}{6}$.) Then an inductive argument, choosing k to be $\lfloor n/2 \rfloor \pm 1$ as appropriate, shows that the relations are valid for all $n \geq 4$. But $A(n)$ is the lower bound in (9) when $T = 3$, and $r(n) \leq \min(u(n), v(n), w(n))$, hence we have proved that $A(n) \leq K_3(n) \leq r(n) \leq A(n) + \frac{5}{6}n - 1$. [The constant $\frac{5}{6}$ can be lowered here.]

17.

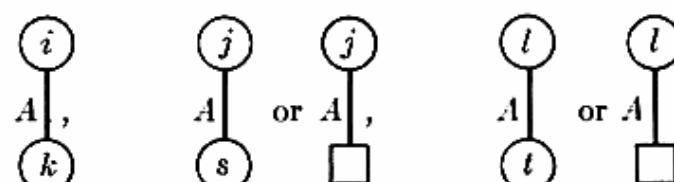
Level	T1	T2	T3	T4	T5
0	1	0	0	0	0
1	5	4	3	2	1
2	55	50	41	29	15
...
n	a_n	b_n	c_n	d_n	e_n
$n+1$	$5a_n + 4b_n + 3c_n + 2d_n + e_n$	$4a_n + 4b_n + 3c_n + 2d_n + e_n$	$3a_n + 3b_n + 3c_n + 2d_n + e_n$	$2a_n + 2b_n + 2c_n + 2d_n + e_n$	$a_n + b_n + c_n + d_n + e_n$

To get from level n to level $n+1$ during the initial distribution, insert k_1 "sublevels" with $(4, 4, 3, 2, 1)$ runs added respectively to tapes (T1, T2, ..., T5), k_2 "sublevels" with $(4, 3, 3, 2, 1)$ runs added, k_3 with $(3, 3, 2, 2, 1)$, k_4 with $(2, 2, 2, 1, 1)$, k_5 with $(1, 1, 1, 1, 0)$, where $k_1 \leq a_n$, $k_2 \leq b_n$, $k_3 \leq c_n$, $k_4 \leq d_n$, $k_5 \leq e_n$. [If $(k_1, \dots, k_5) = (a_n, \dots, e_n)$ we have reached level $n+1$.] Add dummy runs if necessary to fill out a sublevel. Then merge $k_1 + k_2 + k_3 + k_4 + k_5$ runs from (T1, ..., T5) to T6, merge $k_1 + \dots + k_4$ from (T1, ..., T4) to T5, ..., merge k_1 from T1 to T2; and merge k_1 from (T2, ..., T6) to T1, k_2 from (T3, ..., T6) to T2, ..., k_5 from T6 to T5. [This method was used in the UNIVAC III sort program, about 1962, and presented at the ACM Sort Symposium that year.]

19.

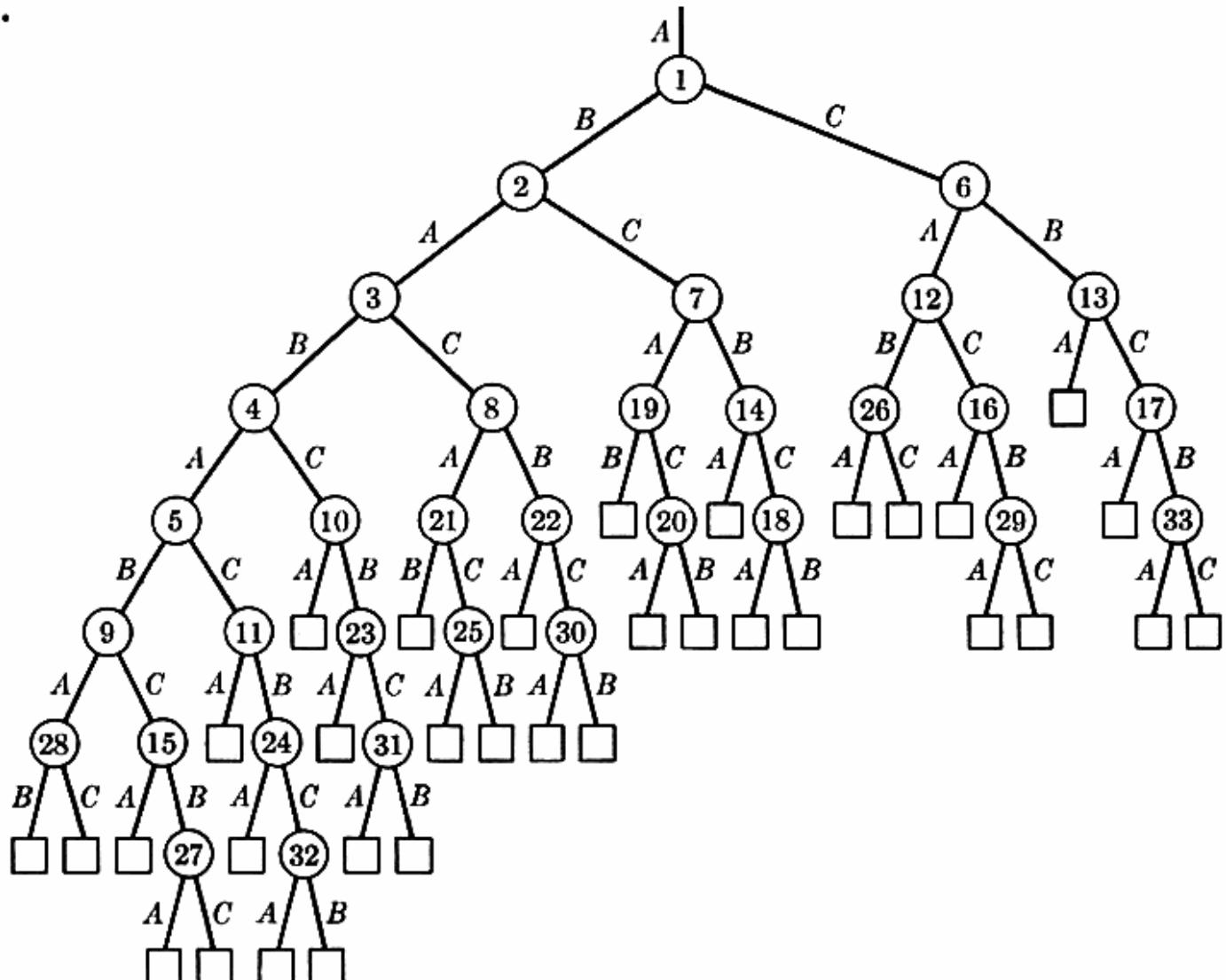


20. A strongly T -fifo tree has a T -fifo labeling in which there are no three branches having the respective forms



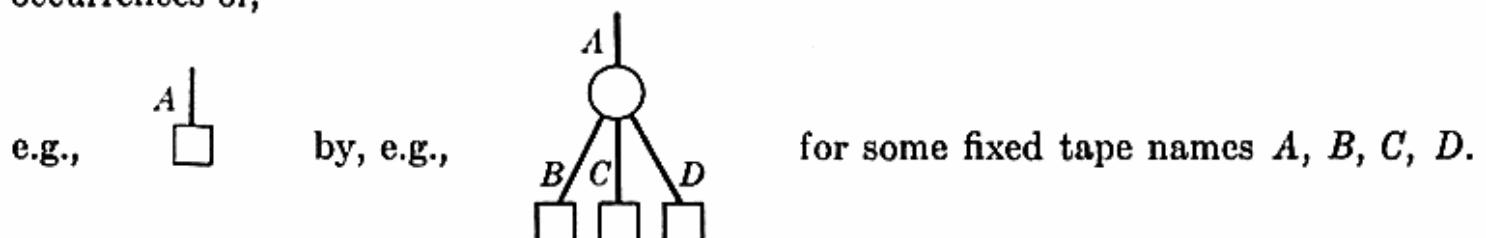
for some tape name A and some $i < j < k < l < s$. Informally, when we "grow on" an A , we must grow on all other A 's before creating any new A 's.

21.



which is very weakly fifo.

22. This occurs for any tree representations formed by successively replacing all occurrences of,



Since all occurrences are replaced by the same pattern, the lifo or fifo order makes no difference in the structure of the tree.

Stating the condition in terms of the vector model: Whenever $(y^{(k+1)} \neq y^{(k)}$ or $k = m$) and $y_j^{(k)} = -1$, we have $y_j^{(k)} + \dots + y_j^{(1)} + y_j^{(0)} = 0$.

23. (a) Assume that $v_1 \leq v_2 \leq \dots \leq v_T$; the "cascade" stage $(1, \dots, 1, -1)^{v_T} (1, \dots, 1, -1, 0)^{v_{T-1}} \dots (1, -1, 0, \dots, 0)^{v_2}$ takes $C(v)$ into v . (b) Immediate, since $C(v)_k \leq C(w)_k$ for all k . (c) If v is obtained in q stages, we have $u \rightarrow u^{(1)} \rightarrow \dots \rightarrow u^{(q)} = v$ for some unit vector u , and some other vectors $u^{(1)}, \dots$. Hence $u^{(1)} \leq C(u)$, $u^{(2)} \leq C(C(u))$, \dots , $v \leq C^q(u)$. Hence $v_1 + \dots + v_T$ is less than or equal to the sum of the elements of $C^q(u)$; and the latter is obtained in cascade merge. [This theorem generalizes the result of exercise 5.4.3-4. Unfortunately the concept of "stage" as defined here doesn't seem to have any practical significance.]

24. Let $y^{(m)} \dots y^{(l+1)}$ be a stage which reduces w to v . If $y_j^{(0)} = -1$, $y_j^{(t-1)} = 0, \dots, y_j^{(k+1)} = 0$, and $y_j^{(k)} = -1$, for some $k < i - 1$, we can insert $y^{(k)}$ between $y^{(i)}$ and $y^{(i-1)}$. Repeat this operation until all (-1) 's in each column are adjacent. Then if $y_j^{(0)} = 0$ and $y_j^{(t-1)} \neq 0$ it is possible to set $y_j^{(t)} \leftarrow 1$; ultimately each column consists of $+1$'s followed by -1 's followed by 0 's, and we have constructed a stage which reduces w' to v for some $w' \geq w$. Permuting the columns, this stage takes the form $(1, \dots, 1, -1)^{a_T} \dots (1, -1, 0, \dots, 0)^{a_2} (-1, 0, \dots, 0)^{a_1}$. The sequence of $T - 1$ relations

$$\begin{aligned} (x_1, \dots, x_T) &\leq (x_1 + x_T, \dots, x_{T-1} + x_T, 0) \\ &\leq (x_1 + x_{T-1} + x_T, \dots, x_{T-2} + x_{T-1} + x_T, x_T, 0) \\ &\leq (x_1 + x_{T-2} + x_{T-1} + x_T, \dots, x_{T-3} + x_{T-2} + x_{T-1} + x_T, \\ &\quad x_{T-1} + x_T, x_T, 0) \leq \dots \\ &\leq (x_1 + x_2 + x_3 + \dots + x_T, x_3 + \dots + x_T, \dots, x_{T-1} + x_T, x_T, 0) \end{aligned}$$

now shows that the best choice of the a 's is $a_T = v_T, a_{T-1} = v_{T-1}, \dots, a_2 = v_2, a_1 = 0$. And the result is best if we permute columns so that $v_1 \leq \dots \leq v_T$.

25. (a) Assume that $v_{T-k+1} \geq \dots \geq v_T \geq v_1 \geq \dots \geq v_{T-k}$ and use $(1, \dots, 1, -1, 0, \dots, 0)^{v_{T-k+1}} \dots (1, \dots, 1, 0, \dots, 0, -1)^{v_T}$. (b) The sum of the l largest elements of $D_k(v)$ is $(l - 1)s_k + s_{k+l}$, for $1 \leq l \leq T - k$. (c) If $v \Rightarrow w$ in a phase that uses k output tapes, we may obviously assume that the phase has the form $(1, \dots, 1, -1, 0, \dots, 0)^{a_1} \dots (1, \dots, 1, 0, \dots, 0, -1)^{a_k}$, with each of the other $T - k$ tapes used as input in each operation. Choosing $a_1 = v_{T-k+1}, \dots, a_k = v_T$ is best. (d) See exercise 22(c). We always have $k_1 = 1$; and $k = T - 2$ always beats $k = T - 1$ since we assume that at least one component of v is zero. Hence for $T = 3$ we have $k_1 \dots k_q = 1^q$ and the initial distribution $(F_{q+1}, F_q, 0)$. For $T = 4$ the undominated strategies and their corresponding distributions are found to be

$q = 2$	$12 (3, 2, 0, 0)$
$q = 3$	$121 (5, 3, 3, 0); 122 (5, 5, 0, 0)$
$q = 4$	$1211 (8, 8, 5, 0); 1222 (10, 10, 0, 0); 1212 (11, 8, 0, 0)$
$q = 5$	$12121 (19, 11, 11, 0); 12222 (20, 20, 0, 0); 12112 (21, 16, 0, 0)$
$q = 6$	$122222 (40, 40, 0, 0); 121212 (41, 30, 0, 0)$
$q \geq 7$	$12^{q-1} (5 \cdot 2^{q-3}, 5 \cdot 2^{q-3}, 0, 0)$

So for $T = 4$ and $q \geq 6$, the minimum-phase merge is like balanced merge, with a slight twist at the very end (going from $(3, 2, 0, 0)$ to $(1, 0, 1, 1)$ instead of to $(0, 0, 2, 1)$).

When $T = 5$ the undominated strategies are $1(32)^{n-1}2, 1(32)^{n-1}3$ for $q = 2n \geq 2$; $1(32)^{n-1}32, 1(32)^{n-1}22, 1(32)^{n-1}23$ for $q = 2n + 1 \geq 3$. (The first strategy listed has most runs in its distribution.) On six tapes they are 13 or 14, 142 or 132 or 133, 1333 or 1423, then 13^{q-1} for $q \geq 5$.

SECTION 5.4.5

1. The following algorithm is controlled by a table $A[L - 1] \dots A[1]A[0]$ that essentially represents a number in radix P notation. As we repeatedly add unity to this number, the “carries” tell us when to merge. Tapes are numbered from 0 to P .

- O1.** [Initialize.] Set $(A[L-1], \dots, A[0]) \leftarrow (0, \dots, 0)$ and $q \leftarrow 0$. (During this algorithm, q will equal $(A[L-1] + \dots + A[0]) \bmod T$.)
- O2.** [Distribute.] Write an initial run on tape q , in ascending order. Set $l \leftarrow 0$.
- O3.** [Add one.] If $l = L$, stop; the output is on tape $(-L) \bmod T$, in ascending order iff L is even. Otherwise set $A[l] \leftarrow A[l] + 1$, $q \leftarrow (q + 1) \bmod T$.
- O4.** [Carry?] If $A[l] < P$, return to O2. Otherwise merge to tape $(q - l) \bmod T$, increase l by 1, set $A[l] \leftarrow 0$ and $q \leftarrow (q + 1) \bmod T$, and return to O3. ■

2. Keep track of how many runs are on each tape. When the input is exhausted, add dummy runs if necessary until reaching a situation with at most one run on each tape and at least one tape empty. Then finish the sort in one more merge, rewinding some tapes first if necessary. (It is possible to deduce the orientation of the runs from the **A** table.)

3.	Op	T0	T1	T2	Op	T0	T1	T2
	Dist	—	A_1	$A_1 A_1$	Dist	$D_2 A_1$	A_1	A_4
	Merge	D_2	—	A_1	Merge	D_2	—	$A_4 D_2$
	Dist	$D_2 A_1$	—	A_1	Merge	—	A_4	A_4
	Merge	D_2	D_2	—	Dist	—	A_4	$A_4 A_1$
	Dist	D_2	$D_2 A_1$	A_1	Copy	—	$A_4 D_1$	A_4
	Merge	$D_2 D_2$	D_2	—	Copy	—	A_4	$A_4 A_1$
	Merge	D_2	—	A_4	Merge	D_5	—	A_4

At this point T2 would be rewound and a final merge would complete the sort. To avoid the useless copy operations, replace step B4 by:

- B4'.** [Ready to merge?] If $A[l-1, q] = s$, go to step B5. Otherwise if the input is exhausted and $A[l-1, q] - s$ is a positive even number and $A[l-1, (2r-q) \bmod T] \neq A[l-1, q] - 1$, set $s \leftarrow A[l-1, q]$ and go to B5. Otherwise go back to step B3.

[This change avoids situations in which the runs are simply shifted back and forth. The first parenthesized comment in step B5 is no longer strictly correct, but the algorithm still works properly because the **A** table entries on level l will not be looked at. The smallest S for which this change makes any difference is $P^3 + 1$. When P is large, the change hardly ever makes much difference, but it does keep the computer from looking too foolish in some circumstances. The algorithm should also be changed to handle the case $S = 1$ more efficiently.]

4. We can, in fact, omit setting $A[0, 0]$ in step B1, $A[l, q]$ in steps B3 and B5. [But $A[l, r]$ must be set in step B3.]

5. $P^{2k} - (P-1)P^{2k-2} < S \leq P^{2k}$ for some $k > 0$.

SECTION 5.4.6

- 1.** $\lfloor 23000480/(n + 480) \rfloor n.$
- 2.** At the instant shown, all the records in that buffer have been moved to the output. Step F2 insists that the test “Is output buffer full?” precede the test “Is input buffer empty?” while merging, otherwise we would have trouble (unless the changes of exercise 4 were made).

3. No, e.g., we might reach a state with P buffers $1/P$ full and $P - 1$ buffers full, if file i contains the keys $i, i + P, i + 2P, \dots$ for $1 \leq i \leq P$. This example shows that $2P$ input buffers are necessary for continuous output even if we allowed simultaneous reading, unless we reallocated memory for partial buffers or used a “scatter read” in some way. [Actually, if blocks contain less than $P - 1$ records, we need less than $2P$ buffers, but this is unlikely. Older computers would read a block of data into a special hardware buffer separate from the computer memory, and computation would stop while the contents of this buffer were transferred to memory. In such a situation, $2P - 1$ “software” buffers in memory would be sufficient.]

4. Set up S sooner (in steps F1 and F4 instead of F3).

5. If, for example, all keys of all files were equal, we couldn't simply make arbitrary decisions while forecasting; the forecast must be compatible with decisions made by the merging process. One safe way is to find the smallest possible m in steps F1 and F4, i.e. to consider a record from file $C[i]$ to be less than all records having the same key on file $C[j]$ whenever $i < j$. (In essence, the file number is appended to the key.)

6. In step C1 also set $\text{TAPE}[T + 1] \leftarrow T + 1$. In step C8 the merge should be to $\text{TAPE}[p + 2]$ instead of $\text{TAPE}[p + 1]$. In step C9, set $(\text{TAPE}[1], \dots, \text{TAPE}[T + 1]) \leftarrow (\text{TAPE}[T + 1], \dots, \text{TAPE}[1])$.

7. The method used in Chart A is $(A_1 D_1)^4 A_0 D_0 (A_1 D_1)^2 A_0 D_0 (A_1 D_1)^3 A_0, D_1 (A_1 D_1)^4 A_0 D_0 (A_1 D_1)^3 A_0 D_0 \alpha A_0 D_0 A_0, D_1 A_0 D_0 (A_1 D_1)^3 A_0 D_0 \alpha A_1 D_1 A_0, D_1 A_1 D_1 \alpha A_1 D_1 A_0$, where $\alpha = (A_0 D_0)^2 A_1 D_1 A_0 D_0 (A_1 D_1)^2 (A_0 D_0)^7 A_1 D_1 (A_0 D_0)^3 A_1 D_1 A_0 D_0$. The first merge phase writes $D_0 A_3 D_3 A_1 D_1 A_4 D_4 A_0 D_0 A_1 D_1 A_1 D_1 A_4 D_4 A_0 D_0 A_1 D_1 A_0 D_0 (A_1 D_1)^4$ on tape 5; the next writes $A_4 D_4 A_4 D_4 A_1 D_1 A_4 D_4 A_0 D_0 A_1 D_1 A_1 D_1 A_7$ on tape 1; the next, $D_{13} A_4 D_4 A_0 D_0 A_{10}$ on tape 4. The final phases are

$$\begin{array}{ccccc}
 A_4 D_4 A_4 & - & D_{19} A_3 D_3 A_{12} & D_{13} A_4 D_4 A_4 & D_0 A_3 \\
 A_4 & D_{23} A_{11} & D_{19} A_3 & D_{13} A_4 & - \\
 - & D_{23} & D_{19} & D_{13} & D_{22} \\
 A_{77} & - & - & - & -
 \end{array}$$

8. No, since at most S stop/start are saved, and since the speed of the input tape (not the output tapes) tends to govern the initial distribution time anyway. The other advantages of the distribution schemes used in Chart A outweigh this minuscule disadvantage.

9. $P = 5, B = 8300, B' = 734, S = \lceil(3 + 1/P)N/6P'\rceil + 1 = 74, \omega = 1.094, \alpha = 0.795, \beta = -1.136, \alpha' = \beta' = 0$; (9) = 855 seconds, to which we add the time for initial rewind, for a total of 958 seconds. The savings of about one minute in the merging does not compensate for the loss of time due to the initial rewinding and tape changing (unless perhaps we are in a multiprogramming environment).

10. The rewinds during standard polyphase merge are over about 54 percent of the file (the “pass/phase” column in Table 5.4.2-1), and the longest rewinds during standard cascade merge cover approximately $a_k a_{n-k} / a_n \approx (4/(2T - 1)) \cos^2(\pi/(4T - 2)) < \frac{4}{11}$ of the file, by exercise 5.4.3-5 and Eq. 5.4.3-13.

11. Only initial and final rewinds get to make use of the “high-speed” feature, since the reel is only a little more than $10/23$ full when it contains the whole example file.

Using $\pi = \lceil .946 \ln S - 1.204 \rceil$, $\pi' = 1/8$ in example 8, we get the following estimated totals for examples 1–9, respectively:

$$1115, 1296, 1241, 1008, 1014, 967, 891, 969, 856.$$

12. (a) An obvious solution with $4P + 4$ buffers simply reads and writes simultaneously from paired tapes. But note that three output buffers are sufficient (at a given moment we are performing the second half of a write from one, the first half of a write from another, and outputting into the third), and this suggests a corresponding improvement in the input buffer situation. It turns out that $3P$ input buffers and 3 output buffers are necessary and sufficient, using a slightly weakened “forecasting” technique. A simpler and superior approach, suggested by J. Sue, adds a “lookahead key” to each block, specifying the final key of the subsequent block. Sue’s method requires $2P + 1$ input buffers and 4 output buffers, and it is a straightforward modification of Algorithm F.

(b) In this case the high value of α means that we must do between five and six passes over the data, which wipes out the advantage of double-quick merging. The idea works out much better on eight or nine tapes.

13. No, consider for example the situation just before $A_{16} A_{16} A_{16} A_{16}$. But two reelfulls can be handled.

$$14. \det \begin{pmatrix} 0 & -p_0z & 0 & z-1 \\ 0 & 1-p_1z & -p_0z & z-1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} / \det \begin{pmatrix} 1-p_{\geq 1}z & -p_0z & 0 & z-1 \\ -p_{\geq 2}z & 1-p_1z & -p_0z & z-1 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

15. The A matrix has the form

$$A = \begin{pmatrix} B_{10}z & B_{11}z & \dots & B_{1n}z & 1-z \\ \vdots & & & & \vdots \\ B_{n0}z & B_{n1}z & \dots & B_{nn}z & 1-z \\ 0 \dots 0 & 1 & 0 & 0 & 0 \\ 0 \dots 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \quad \begin{array}{c} B_{10} + B_{11} + \dots + B_{1n} = 1 \\ \vdots \\ B_{n0} + B_{n1} + \dots + B_{nn} = 1 \end{array} \quad (11)$$

Therefore

$$\det(I - A) = \det \begin{pmatrix} 1 - B_{10}z & -B_{11}z & \dots & -B_{1,n-1}z & -B_{1n}z \\ \vdots & & & & \vdots \\ -B_{n0}z & -B_{n1}z & \dots & 1 - B_{n,n-1}z & -B_{nn}z \\ 0 & 0 & & -1 & 1 \end{pmatrix}$$

and we can add all columns to the first column, then factor out $(1 - z)$. Consequently $g_Q(z)$ has the form $h_Q(z)/(1 - z)$, and $\alpha^{(Q)} = h_Q(1)$ because $h_Q(1) \neq 0$ and $\det(I - A) \neq 0$ for $|z| < 1$.

SECTION 5.4.7

1. Sort from least significant digit to most significant digit in the number system whose radices are alternately P and $T - P$. (If pairs of digits are grouped, we have

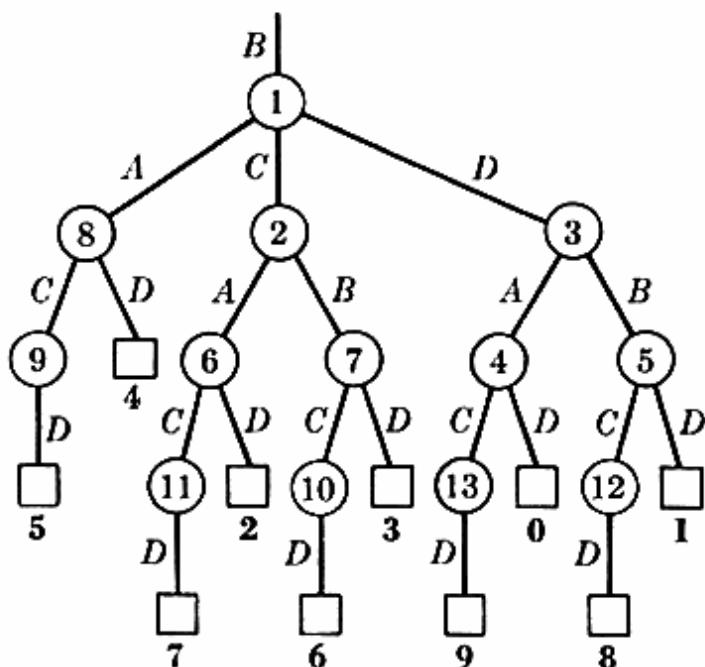
essentially a pure radix of $P \cdot (T - P)$. Thus, if $P = 2$ and $T = 7$, the number system is "biquinary," related to decimal notation in a simple way.)

2. If K is a key between 0 and $F_n - 1$, let the Fibonacci representation of $F_n - 1 - K$ be $a_{n-2}F_{n-1} + \dots + a_1F_2$, where the a_j are 0 or 1, and no two consecutive 1's appear. After phase j , tape $j + 1$ (modulo 3) contains those keys with $a_j = 0$ and tape $j - 1$ (modulo 3) contains those with $a_j = 1$, in decreasing order of $a_{j-1} \dots a_1$.

[Imagine a card sorter with two pockets "0" and "1," and consider the procedure of sorting F_n cards that have been punched with the keys $a_{n-2} \dots a_1$ in $n - 2$ columns. The conventional procedure for sorting these into decreasing order, starting at the least significant digit, can be simplified since we know that everything in the "1" pocket at the end of one pass will go into the "0" pocket on the following pass.]

4. If there were an external node on level 2 we could not construct such a good tree. Otherwise there are at most three external nodes on level 3, six on level 4, since each external node is supposed to appear on the same tape.

5.



6. 09, 08, ..., 00, 19, ..., 10, 29, ..., 20, 39, ..., 30, 40, 41, ..., 49, 59, ..., 50, 60, 61, ..., 99.

7. Yes; first distribute the records into smaller and smaller subfiles until obtaining one-reel files which can be individually sorted. This is dual to the process of sorting one-reel files and then merging them into larger and larger multireel files.

SECTION 5.4.8

1. Yes. The ordering of the file and the selection tree alternates between ascending and descending, and we have in effect an order- P cocktail shaker sort; cf. exercise 9.
2. Let $Z_N = Y_N - X_N$, and solve the recurrence for Z_N by noting that $(N+1)NZ_{N+1} = N(N-1)Z_N + N^2 + N$; hence

$$Z_N = \frac{1}{3}(N+1) + \binom{M+2}{3} / N(N-1), \quad \text{for } N > M.$$

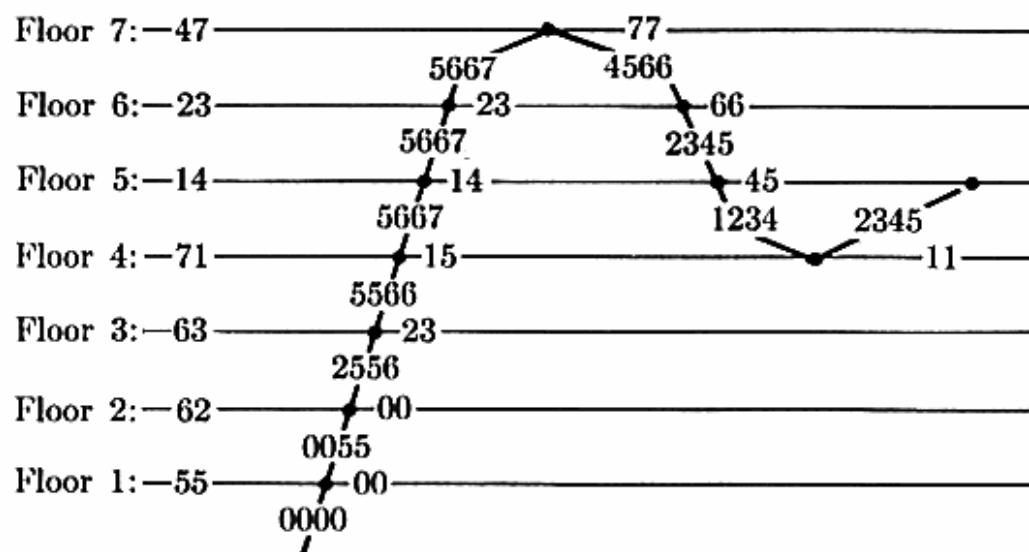
Now eliminate Y_N and obtain

$$\begin{aligned} \frac{X_N}{N+1} &= 6\frac{2}{3}(H_{N+1} - H_{M+2}) + 2\left(\frac{1}{N+1} - \frac{1}{M+2}\right) \\ &\quad - \frac{2}{3}\binom{M+2}{3}\left(\frac{1}{(N+1)N(N-1)} - \frac{1}{(M+2)(M+1)M}\right) + \frac{3M+4}{M+2}, \quad N > M. \end{aligned}$$

3. Yes; find a median element in $O(N)$ steps, using a construction like that of Theorem 5.3.3, and use it to partition the file. Another interesting approach, due to R. W. Floyd and A. J. Smith, is to merge two runs of N items in $O(N)$ units of time as follows: Spread the items out on the tapes, with spaces between them, then successively fill each space with a number specifying the final position of the item just preceding that space.

4. It is possible to piece together a schedule for floors $1, \dots, p+1$ with a schedule for floors q, \dots, n : When the former schedule first reaches floor $p+1$, go up to floor q and carry out the latter schedule (using the current elevator contents as if they were the extra "0" men in the algorithm of Theorem K). After finishing that schedule, go back to floor $p+1$ and resume the previous schedule.

5. Consider $b = 4, c = 2$ and the following behavior of the algorithm:



Now 2 (in the elevator) is less than 3 (on floor 3).

[After constructing an example such as this, the reader should be able to see how to demonstrate the weaker property required in the proof of Theorem K.]

6. Let i, j be minimal with $c_i < c'_i$ and $c_j > c'_j$. Introduce a new man who wants to go from i to j . This doesn't increase $\max(u_k, d_{k+1}, 1)$ or $\max(c_k, c'_k)$ for any k . Continue this until $c_j = c'_j$ for all j . Now observe that the algorithm in the text works also with c replaced by c_k in steps K1 and K3.

8. Let the number be P_n , and let Q_n be the number of permutations such that $u_k = 1$ for $1 \leq k < n$. Then $P_n = Q_1 P_{n-1} + Q_2 P_{n-2} + \dots + Q_n P_0$, $P_0 = 1$. It can be shown that $Q_n = 3^{n-2}$ for $n \geq 2$ (see below), hence a generating function argument yields

$$\sum P_n z^n = (1 - 3z)/(1 - 4z + 2z^2) = 1 + z + 2z^2 + 6z^3 + 20z^4 + \dots;$$

$$2P_n = (2 + \sqrt{2})^{n-1} + (2 - \sqrt{2})^{n-1}.$$

To prove that $Q_n = 3^{n-2}$, consider a ternary sequence $x_1x_2 \dots x_n$ such that $x_1 = 2$, $x_n = 0$, and $0 \leq x_k \leq 2$ for $1 < k < n$. The following rule defines a one-to-one correspondence between such sequences and the desired permutations $a_1a_2 \dots a_n$:

$$a_k = \begin{cases} \max \{j \mid j < k \text{ and } x_j = 0\} \text{ or } j = 1, & \text{if } x_k = 0; \\ k, & \text{if } x_k = 1; \\ \min \{j \mid j > k \text{ and } x_j = 2\} \text{ or } j = n, & \text{if } x_k = 2. \end{cases}$$

(This correspondence was obtained by the author jointly with E. A. Bender.)

9. The number of passes of the cocktail shaker sort is $2 \max(u_1, \dots, u_n) - (0 \text{ or } 1)$, since each pair of passes (left-right-left) reduces each of the nonzero u 's by 1.

10. Begin with a distribution method (quicksort or radix exchange) until one-reel files are obtained. And be patient.

SECTION 5.4.9

1. $\frac{1}{4} - (x \bmod \frac{1}{2})^2$ revolutions.

2. The probability that $k = a_{iq}$ and $k+1 = a_{i'q}$, for fixed k, q, r , and $i \neq i'$ is $f(q, r, k)LL!(PL-2L)!/(PL)!$, where

$$\begin{aligned} f(q, r, k) &= \binom{k-1}{q-1} \binom{k-q}{r-1} \binom{PL-k-1}{L-q} \binom{PL-k-1-L+q}{L-r} \\ &= \binom{k-1}{q+r-2} \binom{q+r-2}{q-1} \binom{PL-k-1}{2L-q-r} \binom{2L-q-r}{L-q}; \end{aligned}$$

and

$$\begin{aligned} \sum_{\substack{1 \leq k \leq PL \\ 1 \leq q, r \leq L}} |q-r| f(q, r, k) &= \sum_{q,r} |q-r| \binom{PL-1}{2L-1} \binom{q+r-2}{q-1} \binom{2L-q-r}{L-q} \\ &= \binom{PL-1}{2L-1} A_{2L-1}. \end{aligned}$$

The probability that $k = a_{iq}$ and $k+1 = a_{i,q+1}$ for fixed k, q , and i is

$$g(k, q) / \binom{PL}{L}, \quad \text{where } g(k, q) = \binom{k-1}{q-1} \binom{PL-k-1}{L-q-1};$$

and

$$\sum_{\substack{1 \leq k \leq PL \\ 1 \leq q \leq L \\ 1 \leq r \leq L}} g(k, q) = \sum_{1 \leq q \leq L} \binom{PL-1}{L-1} = (L-1) \binom{PL-1}{L-1}.$$

[SIAM J. Computing 1 (1972), 161-166.]

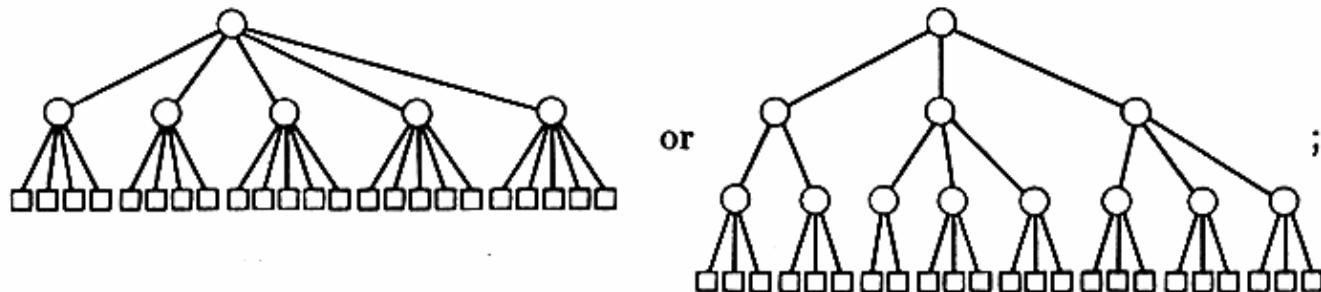
3. Take the minimum in (5) over the range $2 \leq m \leq \min(9, n)$.

4. (a) $(0.000725(\sqrt{P}+1)^2 + 0.014)L$. (b) Change “ $\alpha mn + \beta n$ ” in formula (5) to “ $(0.000725(\sqrt{m}+1)^2 + 0.014)n$.” [Computer experiments show that the nature of the optimal trees defined by this new recurrence is very much the same as those

defined by Theorem K with $\alpha = 0.00145$, $\beta = 0.01545$; in fact, trees exist which are optimal for both recurrences, when $30 \leq n \leq 100$. The change suggested in this exercise saves about 10 percent of the merging time, when $n = 64$ or 100 as in the text's example. This style of buffer allocation was considered already in 1954 by H. Seward, who found that four-way merging minimizes the seek time.]

5. Let $A_m(n)$, $B_m(n)$ represent sets of m trees all of whose leaves are at (even, odd) levels, respectively. Then $A_1(1) = 0$, $B_1(1) = \alpha + \beta$; $A_m(n)$ and $B_m(n)$ are defined as in (4) when $m \geq 2$; $A_1(n) = \min_{1 \leq m \leq n} (\alpha mn + \beta n + B_m(n))$, $B_1(n) = \min_{1 \leq m \leq n} (\alpha mn + \beta n + A_m(n))$. The latter equations are well-defined in spite of the fact that $A_1(n)$ and $B_1(n)$ are defined in terms of each other!

6.



$A_1(23) = B_1(23) = 268$. [Curiously, $n = 23$ is the only value ≤ 50 for which no equal-parity tree with n leaves is optimal in the unrestricted-parity case. Perhaps it is the *only* such value, when $\alpha = \beta$.]

7. Consider the quantities $\alpha d_1 + \beta e_1, \dots, \alpha d_n + \beta e_n$ in any tree, where (d_j, e_j) are the (degree sum, path length) for the j th leaf. An optimum tree for weights $w_1 \leq \dots \leq w_n$ will have $\alpha d_1 + \beta e_1 \geq \dots \geq \alpha d_n + \beta e_n$. It is always possible to reorder these so that $\alpha d_1 + \beta e_1 = \dots = \alpha d_k + \beta e_k$ where the first k leaves are merged together.

11. Using exercise 1.2.4-38, $f_m(n) = 3qn + 2(n - 3^q)m$, when $2 \cdot 3^{q-1} \leq n/m \leq 3^q$; $3qn + 4(n - 3^q)m$, when $3^q \leq n/m \leq 2 \cdot 3^q$. Thus $f_2(n) + 2n \geq f(n)$, with equality iff $4 \cdot 3^{q-1} \leq n \leq 2 \cdot 3^q$; $f_3(n) + 3n = f(n)$; $f_4(n) + 4n \geq f(n)$, with equality iff $n = 4 \cdot 3^q$; and $f_m(n) + mn > f(n)$ for all $m \geq 5$.

12. Use the specifications $-$, $1:1$, $1:1:1$, $1:1:1:1$ or $2:2$, $2:3$, $2:2:2$, $\dots, \lfloor n/3 \rfloor : \lfloor (n+1)/3 \rfloor : \lfloor (n+2)/3 \rfloor, \dots$; this gives trees with all leaves at level $q+2$, for $4 \cdot 3^q \leq n \leq 4 \cdot 3^{q+1}$. (When $n = 4 \cdot 3^q$, two such trees are formed.)

14. The following tree specifications were found for $n = 1, 2, 3, \dots$, by exhaustively examining all partitions of n . $-$, $1:1$, $1:1:1$, $1:1:1:1$, $1:1:1:1:1$, $1:1:1:1:1:1$, $1:1:1:1:3$, $1:1:3:3$, $3:3:3$, $1:3:3:3$, $3:4:4$, $3:3:3:3$, $3:3:3:4$, $3:3:4:4$, $3:4:4:4$, $4:4:4:4$, \dots , $5:6:6:6:12$, $6:6:6:6:12$, $6:6:6:6:13$, \dots . (The degrees seem to be always ≤ 6 , but such a result appears to be quite difficult to prove.)

15. If a people initially got on the elevator, the togetherness rating increases by at most $a + c$ at the first stop. When it next stops at the initial floor, the rating increases by at most $b + c - a$. Hence the rating increases at most $(k-1)b + kc$ after k stops.

16. $123456 \rightarrow$ floor 2; $112334 \rightarrow$ floor 3; $224456 \rightarrow$ floor 4; $135566 \rightarrow$ floor 5; $112334 \rightarrow$ floor 6; $224456 \rightarrow$ floor 2; $222444 \rightarrow$ floor 4; $222222 \rightarrow$ floor 2; $555666 \rightarrow$ floor 5; $666666 \rightarrow$ floor 6; $111333 \rightarrow$ floor 3; $111111 \rightarrow$ floor 1. [Is there a shorter solution? Exercise 15 implies that at least nine stops are needed.]

17. Consider doing step (g) backwards, distributing the records into bin 1, then bin 2, bin 3. This operation is precisely what step (d) is simulating on the key file.

18. The internal sort must be carefully chosen, with paging in mind; methods such as

Shell's diminishing increment sort, address calculation, heapsort, and list sorting would be disastrous, since they require a large "working set" of pages. Quicksort, radix exchange, and sequentially-allocated merge or radix sorting are much better suited to a paging environment.

Some things the designer of an external sort can do, which are "virtually impossible" to include in an automatically paged method, are: (a) Forecasting the input file which should be read next, so that the data is available when it is required; (b) choosing the buffer sizes and the order of merge according to hardware and data characteristics. On the other hand a virtual machine is considerably easier to program, and it can give results that aren't bad, if the programmer is careful and knows the properties of the underlying actual machine. See the study made by Brawn, Gustavson, and Mankin [*ACM* 13 (1970), 483–494].

SECTION 5.5

1. *It is difficult to decide which sorting algorithm is best in a given situation.* ■
2. For small N , list insertion; for medium N , e.g. $N = 64$, list merge; for large N , radix list sort.

SECTION 6.1

1. $\sqrt{(N^2 - 1)/12}$.
2. **S1'.** [Initialize.] Set P \leftarrow FIRST.
3. **S2'.** [Compare.] If K = KEY(P), the algorithm terminates successfully.
4. **S3'.** [Advance.] Set P \leftarrow LINK(P).
5. **S4'.** [End of file?] If P \neq A, go back to S2'. Otherwise the algorithm terminates unsuccessfully. ■
6. Replace line 08 by

```
KEY EQU 3:5
LINK EQU 1:2
START LDA K
        LD1 FIRST
        1
2H    CMPA 0,1(KEY)
        C
        JE SUCCESS
        C
        LD1 0,1(LINK)
        C - S
        J1NZ 2B
        C - S
        1 - S ■ (6C - 3S + 4)u.
```

4. Yes, if we have a way to set "KEY(Λ)" equal to K. [But the technique of loop duplication used in Program Q' has no effect in this case.]
5. No, Program Q always does at least as many operations as Program Q'.
6. Replace line 08 by

```
JE *+4
CMPA KEY+N+2,1
JNE 3B
INC1 1
```

and change lines 03–04 to ENT1 –2–N; 3H INC1 3.

7. Note that $\bar{C}_N = \frac{1}{2}\bar{C}_{N-1} + 1$.

8. Euler's summation formula gives

$$H_n^{(x)} \sim \text{const} + n^{1-x}/(1-x) + \frac{1}{2}n^{-x} - B_2 xn^{-1-x}/2! + B_3 x(x+1)n^{-2-x}/3! - \dots$$

[The constant is $\zeta(x)$; complex variable theory tells us that

$$\zeta(x) = 2^x \pi^{x-1} \sin(\frac{1}{2}\pi x) \Gamma(1-x) \zeta(1-x),$$

a formula which is particularly useful when $x < 0$.]

9. Yes; in fact, $\bar{C}_N = N - N^{-\theta} H_{N-1}^{(-\theta)} = \theta N/(1+\theta) + \frac{1}{2} + O(N^{-\theta})$.

10. $p_1 \leq \dots \leq p_N$; maximum $\bar{C}_N = (N+1) - (\text{minimum } \bar{C}_N)$. [Similarly in the unequal-length case, the maximum average search time is $L_1(1+p_1) + \dots + L_N(1+p_N)$ minus the minimum average search time.]

11. (a) The terms of $f_{m-1}(x_{i_1}, \dots, x_{i_{m-1}})p_i$ are just the probabilities of the possible sequences of requests that could have preceded, leaving R_i in position m . (b) The second identity comes from summing $\binom{n}{m}$ cases of the first, noting the number of times each P_{nk} occurs. The third identity is a consequence of the second, by inversion. [Alternatively, the principle of inclusion and exclusion could be used.]

(c) $\sum_{m \geq 0} m P_{nm} = n Q_{nn} - Q_{n,n-1}$; hence

$$\begin{aligned} d_i &= 1 + (N-1) - p_i \sum_{j \neq i} 1/(p_i + p_j); \\ \sum p_i d_i &= N - \sum_{i < j} (p_i^2 + p_j^2)/(p_i + p_j) \\ &= N - \sum_{i < j} (p_i + p_j - 2p_i p_j/(p_i + p_j)) = (17). \end{aligned}$$

12. $\bar{C}_N = 2^{1-N} + 2 \sum_{0 \leq n \leq N-2} 1/(2^n + 1)$, which converges rapidly to $2\alpha' \approx 2.5290$; exercise 5.2.4-13 gives α' to 40 decimal places.

13. After evaluating the rather tedious sum

$$\sum_{1 \leq i \leq n} i^2 H_{n+i} = \frac{1}{6}n(n+1)(2n+1)(2H_{2n} - H_n) - \frac{1}{36}n(10n^2 + 9n - 1),$$

we obtain the answer

$$\bar{C}_N = \frac{4}{3}N - \frac{2}{3}(2N+1)(H_{2N} - H_N) + \frac{5}{6} - \frac{1}{3}(N+1)^{-1} \approx .409N.$$

[It would be very interesting to know the maximum factor by which (17) can exceed the optimum result (3).]

14. We may assume that $x_1 \leq x_2 \leq \dots \leq x_n$; then the maximum value occurs when $y_{a_1} \leq y_{a_2} \leq \dots \leq y_{a_n}$, and the minimum when $y_{a_1} \geq \dots \geq y_{a_n}$, by an argument like that of Theorem S.

15. Arguing as in Theorem S, the arrangement $R_1 R_2 \dots R_N$ is optimum iff

$$P_1/L_1(1-P_1) \geq \dots \geq P_N/L_N(1-P_N).$$

16. The expected time $T_1 + p_1 T_2 + p_1 p_2 T_3 + \dots + p_1 p_2 \dots p_{N-1} T_N$ is minimized iff $T_1/(1-p_1) \leq \dots \leq T_N/(1-p_N)$. [BIT 3 (1963), 255-256.]

17. Do the jobs in order of increasing deadlines, independent of the respective times T_i ! [Of course in practice some jobs are more important than others, and we may want to minimize the maximum *weighted* tardiness. Or we may wish to minimize the sum $\sum_{1 \leq i \leq n} \max(T_{a_1} + \dots + T_{a_i} - D_{a_i}, 0)$. Neither of these problems appears to have a simple solution.]

18. Let $h = 1$ if s is present, 0 if s is absent. Let $A = \{i \mid q_i < r_i\}$, $B = \{i \mid q_i = r_i\}$, $C = \{i \mid q_i > r_i\}$, $D = \{j \mid t_j > 0\}$; then the sum $\sum_{1 \leq i,j \leq N} p_i p_j d_{|i-j|}$ for the (q, r) arrangement minus the corresponding sum for the (q', r') arrangement is equal to

$$2 \sum_{i \in A, j \in C} (q_i - r_i)(q_j - r_j)(d_{|i-j|} - d_{h+1+2k-i-j}) \\ + 2 \sum_{i \in C, j \in D} (q_i - r_i)t_j(d_{h+2k-i+j} - d_{i-1+j}).$$

This is positive unless $C = \emptyset$ or $A \cup D = \emptyset$. The desired result now follows because the organ pipe arrangements are the only permutations which are not improved by this construction and its left-right dual when $m = 0, 1$.

[This result is essentially due to G. H. Hardy, J. E. Littlewood, and G. Pólya, *Proc. London Math. Soc.* (2), 25 (1926), 265–282, who showed, in fact, that the minimum of $\sum p_i q_j d_{|i-j|}$ is achieved, under all independent rearrangements of the p 's and q 's, when both p 's and q 's are in a consistent organ-pipe order. For further commentary and generalizations, see their book *Inequalities* (Cambridge University Press, 1934), Chapter 10.)]

19. All arrangements are equally good. We have

$$\sum p_i p_j d(i, j) = \frac{1}{2} \sum p_i p_j (d(i, j) + d(j, i)) = \frac{1}{2} c.$$

[The special case $d(i, j) = 1 + (j - i) \bmod N$ is due to K. E. Iverson, *A Programming Language* (New York: Wiley, 1962), 138. R. L. Baber, *JACM* 10 (1963), 478–486, has studied some other problems associated with tape searching when a tape can read forward, rewind, or backspace k blocks without reading. W. D. Frazer observes that it would be possible to make significant reductions in the search time if we were allowed to *replicate* some of the information in the file; cf. E. B. Eichelberger et. al., *IBM J. Research & Development* 12 (1968), 130–139, for an empirical solution to a similar problem.]

20. Going from (q, r) to (q', r') as in exercise 18, with $m = 0$ or $m = h = 1$, gives a net change of

$$\sum_{i \in A, j \in C} (q_i - r_i)(q_j - r_j)(d_{|i-j|} - \min(d_{h+1+2k-i-j}, d_{i+j-1})),$$

which is positive unless A or C is \emptyset . By circular symmetry it follows that the only optimal arrangements are cyclic shifts of the organ pipe configurations. [For a different problem with the same answer, see Alfred Lehman, *Israel J. Math* 1 (1963), 22–28.]

21. This problem was essentially first solved by L. H. Harper, *J. SIAM* 12 (1964), 131–135. For generalizations and references to other work, see *J. Applied Probability* 4 (1967), 397–401.

22. A priority queue of size 1000 (represented as, say, a heap, see Section 5.2.3).

Enter the first 1000 records into this queue, with the element of *greatest* $d(K_i, K)$ at the front. For each subsequent K_i with $d(K_i, K) < d(\text{front of queue}, K)$, replace the front element by R_i .

SECTION 6.2.1

1. Prove inductively that $K_{l-1} < K < K_{u+1}$ whenever we reach step B2; and that $l \leq i \leq u$ whenever we reach B3.
2. (a, c) No, it loops if $l = u$ and $K > K_u$. (b) Yes, it would!
3. This is Algorithm 6.1T with $N = 3$. In a successful search, that algorithm makes $(N + 1)/2$ comparisons, on the average; in an unsuccessful search it makes $N/2 + 1 - 1/(N + 1)$.
4. It must be an unsuccessful search with $N = 127$; hence by Theorem B the answer is $138u$.
5. Program 6.1Q' has an average running time of $1.75N + 8.5 - (N \bmod 2)/4N$; this beats Program B iff $N \leq 44$. [It beats Program C only for $N \leq 11$.]
7. (a) Certainly not. (b) The parenthesized remarks in Algorithm U will hold true, so it will work, but only if $K_0 = -\infty$ and $K_{N+1} = +\infty$ are both present when N is odd.
8. (a) N . It is interesting to prove this by induction, observing that exactly one of the DELTA's increases if we replace N by $N + 1$. [See *AMM* 77 (1970), 884 for a generalization.] (b) Maximum = $\sum \text{DELTA}[j] = N$; minimum = $2\text{DELTA}[1] - \sum \text{DELTA}[j] = N \bmod 2$.
9. Consider the corresponding tree (e.g. Fig. 6): When N is odd, the left subtree of the root is a mirror image of the right subtree, so $K < K_i$ occurs just as often as $K > K_i$; on the average, $C1 = \frac{1}{2}(C + S)$ and $C2 = \frac{1}{2}(C - S)$, $A = \frac{1}{2}(1 - S)$. When N is even, the tree is the same as the tree for $N + 1$ with all labels decreased by 1, except that ① becomes redundant; on the average,

$$\begin{aligned} C1 &= \frac{1}{2}(C + 1) - \frac{1}{2}\lfloor \log_2 N \rfloor / N, \\ C2 &= \frac{1}{2}(C - 1) + \frac{1}{2}\lfloor \log_2 N \rfloor / N, \quad A = 0 \quad \text{for } S = 1; \\ C1 &= \frac{1}{2} \frac{N}{N+1} (\lfloor \log_2 N \rfloor + 1), \\ C2 &= \frac{1}{2} \frac{N+2}{N+1} (\lfloor \log_2 N \rfloor + 1), \quad A = \frac{1}{2} \frac{N}{N+1} \quad \text{for } S = 0. \end{aligned}$$

(The average value of C is stated in the text.)

10. If $N = 2^k - 1$.
11. Use a "macro-expanded" program with the DELTA's included; e.g., for $N = 10$:

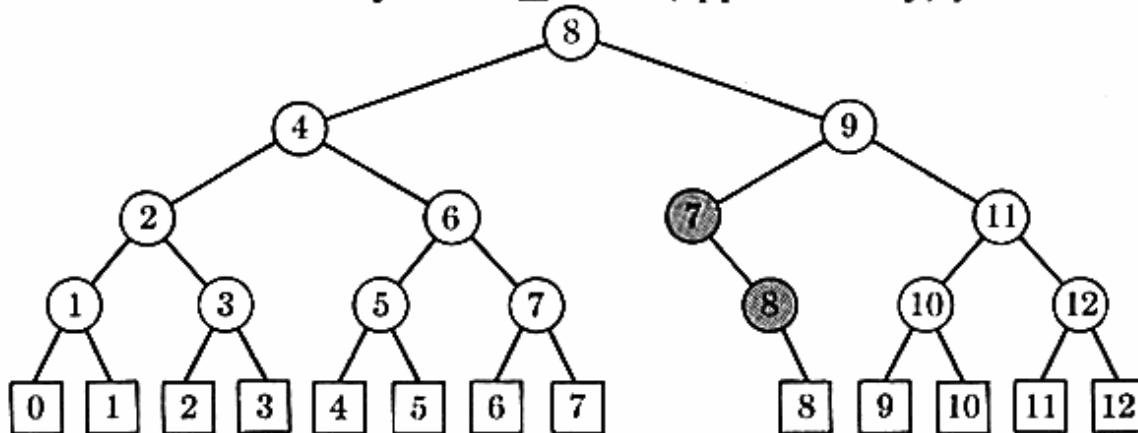
```

START ENT1 5
      LDA   K
      CMPA KEY,1
      JL    C3A
  
```

C4A	JE	SUCCESS	C3A	EQU	*
	INC1	3		DEC1	3
	CMPA	KEY,1		CMPA	KEY,1
	JL	C3B		JGE	C4B
C4B	JE	SUCCESS	C3B	EQU	*
	INC1	1		DEC1	1
	CMPA	KEY,1		CMPA	KEY,1
	JL	C3C		JGE	C4C
C4C	JE	SUCCESS	C3C	EQU	*
	INC1	1		DEC1	1
	CMPA	KEY,1		CMPA	KEY,1
	JE	SUCCESS		JE	SUCCESS
	JMP	FAILURE		JMP	FAILURE

[Exercise 23 shows that most of the “JE” instructions may be eliminated, yielding a program about $6 \log_2 N$ lines long which takes only about $4 \log_2 N$ units of time; but that program will be faster only for $N \geq 1000$ (approximately).]

12.



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

14. One idea is to find the least $M \geq 0$ such that $N + M$ has the form $F_{k+1} - 1$, then to start with $i \leftarrow F_k - M$ in step F1, and to insert “If $i \leq 0$, go to F4” at the beginning of F2. A better solution would be to adapt Shar’s idea to the Fibonaccian case: If the result of the very first comparison is $K > K_{F_k}$, set $i \leftarrow i - M$ and go to F4 (proceeding normally from then on). This avoids extra time in the inner loop.

15. The Fibonacci tree of order k , with left and right reversed, is the binary tree corresponding to the lineal chart up to the k th month, under the “natural correspondence” of Section 2.3.2, if we remove the topmost node of the lineal chart.

16. The external nodes appear on levels $\lfloor k/2 \rfloor$ through $k - 1$; the difference between these levels is greater than unity except when $k = 0, 1, 2, 3, 4$.

17. Let the path length be $k - A(n)$; then $A(F_i) = j$ and $A(F_i + m) = 1 + A(m)$ when $0 < m < F_{j-1}$.

18. Successful search: $A_k = 0$, $C_k = (3kF_{k+1} + (k-4)F_k)/5(F_{k+1} - 1) - 1$, $C'_{1k} = C_{k-1}$. Unsuccessful search: $A'_k = F_k/F_{k+1}$, $C'_k = (3kF_{k+1} + (k-4)F_k)/5F_{k+1}$, $C'_{1k} = C'_{k-1} + F_{k-1}/F_{k+1}$. $C2 = C - C1$. (Cf. exercise 1.2.8-12.)

20. (a) $p^{-p}q^{-q}$. (b) There are at least two errors. The first blunder is that division

is not a linear function, so it can't be simply "averaged over." Actually with probability p we get pN elements remaining, and with probability q we get qN , so we can expect to get $(p^2 + q^2)N$; thus the average "reduction factor" is really $1/(p^2 + q^2)$. Now the reduction factor after k iterations is $1/(p^2 + q^2)^k$, but we cannot conclude that $b = 1/(p^2 + q^2)$ since the number of iterations needed to locate some of the items is much more than to locate others. This is a second fallacy. [It is very easy to make plausible but fallacious probability arguments, and we must always be on our guard against such pitfalls!]

21. It's impossible, since the method depends on the key values.

23. Go to the right on \geq , to the left on $<$; when reaching node i it follows from (1) that $K_i \leq K < K_{i+1}$, so a final test for equality will distinguish between success or failure. (The key $K_0 = -\infty$ should always be present.)

Algorithm C would be changed to go to C4 if $K = K_i$ in step C2. In C3 if $\text{DELTA}[j] = 0$, set $i \leftarrow i - 1$ and go to C5. In C4 if $\text{DELTA}[j] = 0$, go directly to C5. Add a new step C5: "If $K = K_i$, the algorithm terminates successfully, otherwise it terminates unsuccessfully." [This would not speed up Program C unless N were extremely large.]

24. The keys can be arranged so that we first set $i \leftarrow 1$, then $i \leftarrow 2i$ or $2i + 1$ according as $K < K_i$ or $K > K_i$; the search is unsuccessful when $i > N$. For example when $N = 12$ the necessary key arrangement is

$$K_8 < K_4 < K_9 < K_2 < K_{10} < K_5 < K_{11} < K_1 < K_{12} < K_6 < K_3 < K_7.$$

When programmed for MIX this method will take only about $6 \log_2 N$ units of time, so it is faster than Program C. The only disadvantage is that it is a little tricky to set up the table in the first place.

25. (a) Since $a_0 = 1 - b_0$, $a_1 = 2a_0 - b_1$, $a_2 = 2a_1 - b_2$, etc., we have $A(z) + B(z) = 1 + 2zA(z)$. Several of the formulas derived in Section 2.3.4.5 follow immediately from this relation by considering $A(1)$, $B(1)$, $B(\frac{1}{2})$, $A'(1)$, $B'(1)$. If we use two variables to distinguish left and right steps of a path we obtain the more general result $A(x, y) + B(x, y) = 1 + (x + y)A(x, y)$, a special case of a formula which holds in t -ary trees [cf. R. M. Karp, *IRE Trans. IT-7* (1961), 27–38]. (b) $\text{var}(g) = ((N+1)/N) \text{ var}(h) - ((N+1)/N^2) \text{ mean}(h)^2 + 2$.

26. The merge tree for the three-tape polyphase merge with a perfect level k distribution is the Fibonacci tree of order $k + 1$ if we permute left and right appropriately. (Redraw the Polyphase $T = 3$, $S = 13$ picture of Section 5.4.4, with the left and right subtrees of A, C reversed, obtaining Fig. 8.)

27. At most $k + 1$ of the 2^k outcomes will ever occur, since we may order the indices such that $K_{i_1} < K_{i_2} < \dots < K_{i_k}$. Thus the search can be described by a tree with at most $(k + 1)$ -way branching at each node. The number of items which can be found on the m th step is at most $k(k + 1)^{m-1}$; hence the average number of comparisons is at least N^{-1} times the sum of the smallest N elements of the multiset $\{k \cdot 1, k(k + 1) \cdot 2, k(k + 1)^2 \cdot 3, \dots\}$. When $N \geq (k + 1)^n - 1$, the average number of comparisons is $\geq N^{-1} \sum_{1 \leq m \leq n} k(k + 1)^{m-1} m > n - 1/k$.

SECTION 6.2.2

1. Use a header node, with say $\text{ROOT} \equiv \text{RLINK}(\text{HEAD})$; start the algorithm at step T4 with $P \leftarrow \text{HEAD}$. Step T5 is to act as if $K > \text{KEY}(\text{HEAD})$. [Thus change lines 04 and 05 of Program T to “ENT1 ROOT; CMPA K”.]

2. In step T5, set $\text{RTAG} \leftarrow -$. Also, when inserting to the left, set $\text{RLINK}(Q) \leftarrow P$; when inserting to the right, set $\text{RLINK}(Q) \leftarrow \text{RLINK}(P)$. [If nodes are inserted into successively increasing locations Q, and if all deletions are last-in-first-out, the RTAG fields can be eliminated since $\text{RTAG}(P)$ will be $-$ if and only if $\text{RLINK}(P) < P$.]

3. We could replace Λ by a valid address, and set $\text{KEY}(\Lambda) \leftarrow K$ at the beginning of the algorithm; then the tests for LLINK or $\text{RLINK} = \Lambda$ could be removed from the inner loop. However, in order to do a proper insertion we need to introduce another pointer variable which follows P ; this can be done without losing the stated speed advantage, by duplicating the code as in Program 6.2.1F. Thus the MIX time would be reduced to about $5.5C$.

4. $C_N = 1 + (0 \cdot 1 + 1 \cdot 2 + \dots + (n-1)2^{n-1} + C'_2 + \dots + C'_{N-1})/N = (1 + 1/N)C'_N - 1$, for $N \geq 2^n - 1$. The solution to these equations is $C'_N = 2(H_{N+1} - H_{2^n}) + n$ for $N \geq 2^n - 1$, a savings of $2H_{2^n} - n - 2 \approx n(\ln 4 - 1)$ comparisons. The actual improvement for $n = 1, 2, 3, 4$ is, respectively, $0, \frac{1}{6}, \frac{61}{140}, \frac{274399}{360360}$; thus comparatively little is gained for small fixed n . [See Frazer and McKellar, *JACM* 17 (1970), 502, for a more detailed derivation related to an equivalent sorting problem.]

5. (a) The first element must be CAPRICORN; then we multiply the number of ways to produce the left subtree by the number of ways to produce the right subtree, times $\binom{10}{3}$, the number of ways to shuffle those two sequences together. Thus the answer comes to

$$\binom{10}{3} \binom{2}{0} \binom{1}{0} \binom{0}{0} \binom{6}{3} \binom{2}{0} \binom{1}{0} \binom{0}{0} \binom{2}{1} \binom{0}{0} \binom{0}{0} = 4800.$$

[In general, the answer is the product, over all nodes, of $\binom{l+r}{r}$, where l and r stand for the sizes of the left and right subtrees of the node. This is equal to $N!$ divided by the product of the subtree sizes. It is same formula as in exercise 5.1.4-20; indeed, there is an obvious one-to-one correspondence between the permutations which yield a particular search tree and the “topological” permutations counted in that exercise, if we replace a_k in the search tree by k (using the notation of exercise 6).] (b) $2^{N-1} = 1024$; at each step but the last, insert either the smallest or largest remaining key.

6. (a) For each of the P_{nk} permutations $a_1 \dots a_{n-1} a_n$ whose “score” is k , construct $n+1$ permutations $a'_1 \dots a'_{n-1} m a'_n$, where $a'_j = a_j$ or $a_j + 1$, according as $a_j < m$ or $a_j \geq m$. [Cf. Section 1.2.5, Method 2.] If $m = a_n$ or $a_n + 1$, this permutation has a “score” of $k+1$, otherwise it has a score of k . (b) $G_n(z) = (2z + n - 2)(2z + n - 3) \dots (2z)$. Hence

$$P_{nk} = \binom{n-1}{k} 2^k.$$

This generating function was, in essence, obtained by W. C. Lynch, *Comp. J.* 7 (1965), 299–302. (c) The generating function for probabilities is $g_n(z) = G_n(z)/n!$.

This is a product of simple probability generating functions, so

$$\begin{aligned}\text{var}(g_n) &= \sum_{0 \leq k \leq n-2} \text{var}((2z+k)/(2+k)) \\ &= \sum_{0 \leq k \leq n-2} (2/(k+2) - 4/(k+2)^2) = 2H_n - 4H_n^{(2)} + 2.\end{aligned}$$

[From exercise 6.2.1-25b we can therefore compute the variance of C_n , which is $(2+10/n)H_n - 4(1+1/n)(H_n^{(2)}/n + H_n^2) + 2$; this formula is due to G. D. Knott (unpublished).]

7. Set up a recurrence like that in the answer to exercise 5.2.2-32, but with $C_{nm} = 1 + (A_{nm} + B_{nm})/n$. The solution is $C_{nm} = H_m + H_{n+1-m} - 1$. [CACM 12 (1969), 77-80].

8. (a) $g_n(z) = z^{n-1} \sum_{1 \leq i \leq n} g_{i-1}(z)g_{n-i}(z)/n$, $g_0(z) = 1$. (b) $7n^2 - 4(n+1)^2H_n^{(2)} - 2(n+1)H_n + 13n$. [P. F. Windley, Comp. J. 3 (1960), 86, gave recurrence relations from which this variance could be computed numerically, but he did not obtain the solution. Note that this result is *not* simply related to the variance of C_n stated in the answer to exercise 6.]

10. For example, each word x of the key could be replaced by $(ax) \bmod m$, where m is the computer word size and a is a random multiplier relatively prime to m . A value near to $(\phi - 1)m$ can be recommended, see Section 6.4. The flexible storage allocation of tree methods may make them more attractive than other hash coding schemes.

11. $N - 2$, but this occurs with probability $1/(N \cdot N!)$, only in the deletion (1) $N(N - 1) \dots 2$.

12. $\frac{1}{2}(n+1)(n+2)$ of the deletions in the proof of Theorem H belong to Case 1, so the answer is $(N+1)/2N$.

13. Yes, in fact the proof of Theorem H shows that if we delete the i th element inserted, for any fixed i , the result is random. [Conjecture: After an arbitrary sequence of random insertions and *first-in-first-out* deletions, the result is random.]

14. Let $\text{NODE}(T)$ be on level k , and let $\text{LLINK}(T) = \Lambda$, $\text{RLINK}(T) = R_1$, $\text{LLINK}(R_1) = R_2, \dots, \text{LLINK}(R_d) = \Lambda$, where $R_d \neq \Lambda$ and $d \geq 1$. Let $\text{NODE}(R_i)$ have n_i nodes in its right subtree, for $1 \leq i \leq d$. With step $D1\frac{1}{2}$ the internal path length decreases by $k + d + n_1 + \dots + n_d$; without that step it decreases by $k + d + n_d$.

16. Yes; even the deletion operation on permutations, as defined in the proof of Theorem H, is commutative (if we omit the renumbering aspect). If there is an element between X and Y , deletion is obviously commutative since the operation is affected only by the relative positions of X , Y , and their successors and there is no interaction between the deletion of X and the deletion of Y . On the other hand, if Y is the successor of X , and Y is the largest element, both orders of deletion have the effect of simply removing X and Y . If Y is the successor of X and Z the successor of Y , both orders of deletion have the effect of replacing the *first* occurrence of X , Y , or Z by Z and deleting the second and third occurrences of these elements within the permutation.

18. Use exercise 1.2.7-14.

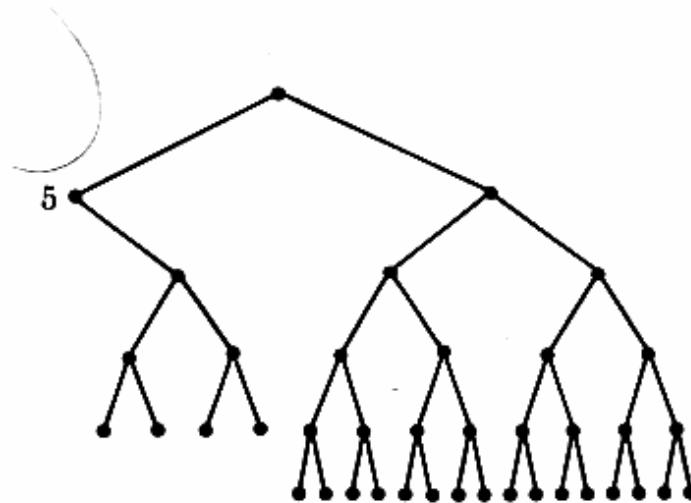
19. $2H_N - 1 - 2\sum_{1 \leq k \leq N} (k-1)^\theta/kN^\theta = 2H_N - 1 - 2/\theta + O(N^{-\theta})$.

20. Yes indeed. Assume that $K_1 < \dots < K_N$, so that the tree built by Algorithm T is degenerate; if, say, $p_k = ((N+1)/2 - k)\epsilon/N$, the average number of comparisons is $(N+1)/2 - (N^2-1)\epsilon/12$, while the optimum tree requires less than $\lceil \log_2 N \rceil$ comparisons.

21. $\frac{1}{8}, \frac{3}{20}, \frac{9}{20}, \frac{3}{20}, \frac{1}{8}$. (Most of the angles are 30° , 60° , or 90° .)

22. This is obvious when $d = 2$, and for $d > 2$ we had $r[i, j-1] \leq r[i+1, j-1] \leq r[i+1, j]$.

23.



[Increasing the weight of the first node will eventually make it move to the root position; this suggests that dynamically maintaining a perfectly optimum tree is hard.]

24. Let c be the cost of a tree obtained by deleting the n th node of an optimum tree. Then $c(0, n-1) \leq c \leq c(0, n) - q_{n-1}$, since the deletion operation always moves $n-1$ up one level. Also $c(0, n) \leq c(0, n-1) + q_{n-1}$, since the stated replacement yields a tree of the latter cost. It follows that $c(0, n-1) = c = c(0, n) - q_{n-1}$.

25. (a) Assume that $A \leq B$ and $B \leq C$, and let $a \in A$, $b \in B$, $c \in C$, $c < a$. If $c \leq b$ then $c \in B$; hence $c \in A$ and $a \in B$; hence $a \in C$. If $c > b$, then $a \in B$; hence $a \in C$ and $c \in B$; hence $c \in A$. (b) Not hard to prove.

26. The cost of every tree has the form $y + lx$ for some real $y \geq 0$ and integer $l > 0$. The minimum of a finite number of such functions (taken over all trees) always has the form described.

27. (a) The answer to exercise 24 (especially the fact that $c = c(0, n-1)$) implies that $R(0, n-1) = R(0, n) \setminus \{n\}$. (b) If $l = l'$ the result in the hint is trivial. Otherwise let the respective paths to n be

$$(r_1), (r_2), \dots, [r_l] \quad \text{and} \quad (s_1), (s_2), \dots, [s_{l'}].$$

Since $r = r_1 > s_1 = s$ and $r_{l'} < s_{l'} = n$, we can find a level $k \geq 1$ such that $r_k > s_k$ and $r_{k+1} \leq s_{k+1}$. We have $r_{k+1} \in R(r_k, n)$, $s_{k+1} \in R(s_k, n)$, and $R(s_k, n) \subseteq R(r_k, n)$ by induction, hence $r_{k+1} \in R(s_k, n)$ and $s_{k+1} \in R(r_k, n)$; the right subtrees of (r_k) and (s_k) can be interchanged, and the result in the hint follows.

Now to show that $R'_h \leq R_h$, let $r \in R'_h$, $s \in R_h$, $s < r$, and consider the optimum trees shown when $x = x_h$; we must have $l \geq l_h$ and we may assume that $l' = l_h$. To show that $R_h \leq R'_{h+1}$, let $r \in R_h$, $s \in R'_{h+1}$, $s < r$, and consider the optimum trees shown when $x = x_{h+1}$; we must have $l' \leq l_h$ and we may assume that $l = l_h$.

29. It is a degenerate tree with YOU at the top, THE at the bottom, needing 19.158 comparisons on the average.

Douglas A. Hamilton has proved that some degenerate tree is always worst. Therefore an $O(n^2)$ algorithm exists to find pessimal binary search trees.

31. (a) 30; (b) 31. Thus there are trees built up according to rule (i) which have less cost than trees for which Phase 3 will succeed.

32. Suppose that we form $q_i + q_j$ at one step, and $q_{i'} + q_{j'} < q_i + q_j$ at the next step. Then the combination $q_{i'} + q_{j'}$ must have been blocked at the first step by having external node $\boxed{q_i}$ or $\boxed{q_j}$ (or both) between $q_{i'}$ and $q_{j'}$. If both, we must have $q_{i'} > q_i$, $q_{j'} \geq q_j$; otherwise we must have $q_{i'} > q_j$, $q_{j'} > q_i$.

35. Let $i < j$; C_i is the leading $e_i + 1$ bits of x_i and C_j is the leading $e_j + 1$ bits of $x_j \geq x_i + 2^{-e_i-1} + 2^{-e_j-1}$. Therefore if $e_i \leq e_j$, C_i cannot be a prefix of C_j ; if $e_i > e_j$, C_j cannot be a prefix of C_i .

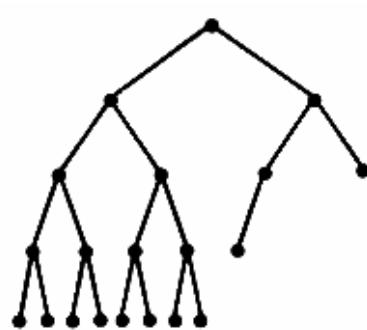
36. In fact it is less than $2S + \sum_{0 \leq i \leq n} (q_i + x_i + y_i) \log_2 (S/(q_i + x_i + y_i))$, where the x_i and y_i are any nonnegative values such that $p_i = x_i + y_{i-1}$ and $x_0 = y_n = 0$. For if \textcircled{i} appears at level l_i and \boxed{j} at level l'_j , we have $l'_j > l_i$ and $l'_{i-1} > l_i$ in any tree. Hence, constructing the tree of Theorem G with weights $q_i + x_i + y_i$, we have $\sum p_i(l_i + 1) + \sum q_j l'_j \leq \sum (q_j + x_j + y_j) l'_j$. [The best bound is obtained when the $q_j + x_j + y_j$ are farthest from uniformity.]

38. Let $a_r = \sum_{i \in S(r)} c_i$; then $a_{r+1} - a_r \geq a_r - a_{r-1}$ for $1 \leq r < \lceil n/2 \rceil$. (Cf. exercise 32.) Hence $a_r \leq r/\lceil n/2 \rceil$, for all r ; this follows since the relations $a_r > r/\lceil n/2 \rceil$ and $a_{r-1} \leq (r-1)/\lceil n/2 \rceil$ imply $1 \geq a_{\lceil n/2 \rceil} = a_r + \sum_{r \leq k < \lceil n/2 \rceil} (a_{k+1} - a_k) \geq a_r + (\lceil n/2 \rceil - r)(a_r - a_{r-1}) > r/\lceil n/2 \rceil + (\lceil n/2 \rceil - r)/\lceil n/2 \rceil = 1$. The proof is completed by using the Hu-Tucker algorithm on the given "worst" probabilities: the resulting tree has $n+1-2^q$ pairs of external nodes on level $q+1$, and all other external nodes on level q , so $f(n) = q(n+1-2^q)/\lceil n/2 \rceil + (q-1)(1-(n+1-2^q)/\lceil n/2 \rceil)$. This is at least as great as the cost of T , which has a weight of $a_{n+1-2^q} \leq (n+1-2^q)/\lceil n/2 \rceil$ on level q . [*Acta Informatica 2* (1972), to appear.]

SECTION 6.2.3

1. The symmetric order of nodes must be preserved by the transformation, otherwise we wouldn't have a binary search tree.
2. $B(S) = 0$ can happen only when S points to the root of the tree (it has never been changed in steps A3 or A4), and all nodes from S to the point of insertion were balanced.
3. Let ρ_h be the largest possible ratio of unbalanced nodes in balanced trees of height h . Thus $\rho_1 = 0$, $\rho_2 = \frac{1}{2}$, $\rho_3 = \frac{1}{2}$. We will prove that $\rho_h = (F_{h+1} - 1)/(F_{h+2} - 1)$. Let T_h be a tree which maximizes ρ_h ; then we may assume that its left subtree has height $h - 1$ and its right subtree has height $h - 2$, for if both subtrees had height $h - 1$ the ratio would be less than ρ_{h-1} . Thus the ratio for T_h is at most $(\rho_{h-1}N_l + \rho_{h-2}N_r + 1)/(N_l + N_r + 1)$, where there are (N_l, N_r) nodes in the (left, right) subtree. This formula takes its maximum value when (N_l, N_r) take their minimum values; hence we may assume that T_h is a Fibonacci tree.

4. When $n = 7$,



has greater path length. [Note: C. C. Foster, *Proc. ACM Nat. Conf.* 20 (1965), 197–198, gave an incorrect procedure for constructing N -node balanced trees of maximum path length; Edward Logg has observed that Foster's Fig. 3 gives a nonoptimal result after 24 steps (node number 22 can be removed in favor of number 25).]

5. This can be proved by induction; if T_N denotes the tree constructed, we have

$$T_N = \begin{cases} \text{Diagram of a single node} & , \quad \text{if } 2^n \leq N < 2^n + 2^{n-1}; \\ T_{2^{n-1}-1} \quad T_{N-2^{n-1}} & \\ \text{Diagram of a single node} & , \quad \text{if } 2^n + 2^{n-1} \leq N < 2^{n+1}. \\ T_{2^n-1} \quad T_{N-2^n} & \end{cases}$$

6. The coefficient of z^n in $zB_i(z)B_j(z)$ is the number of n -node binary trees whose left subtree is a balanced binary tree of height i and whose right subtree is a balanced binary tree of height j .

7. $C_{n+1} = C_n^2 + 2B_{n-1}B_{n-2}$; hence if we let $\alpha_0 = \ln 2$, $\alpha_1 = 0$, and $\alpha_{n+2} = \ln(1 + 2B_{n+1}B_n/C_{n+2}^2) = O(1/B_nC_{n+2})$, and $\theta = \exp(\alpha_0/2 + \alpha_1/4 + \alpha_2/8 + \dots)$, we find that $0 < \theta^{2^n} - C_n = C_n(\exp(\alpha_n/2 + \alpha_{n+1}/4 + \dots) - 1) < 1$, i.e., $C_n = \lfloor \theta^{2^n} \rfloor$.

8. Let $b_h = B'_h(1)/B_h(1) + 1$, and let ϵ_h be the very small quantity $2B_hB_{h-1}(b_h - b_{h-1})/B_{h+1}$. Then $b_h = 2b_{h-1} - \epsilon_h$; hence $b_h = 2^h(1 - \epsilon_1/2 - \epsilon_2/4 - \dots) + r_h$, where $r_h = \epsilon_{h+1}/2 + \epsilon_{h+2}/4 + \dots$ is extremely small for large h . [*Zh. vychisl. matem. i matem. fiziki* 6,2 (1966), 389–394.]

11. Good reason: A balanced tree with more than one internal node has precisely as many $-A$ external nodes as it has $+B$ and $++B$, and as many $+A$'s as $+B$'s and $--B$'s.

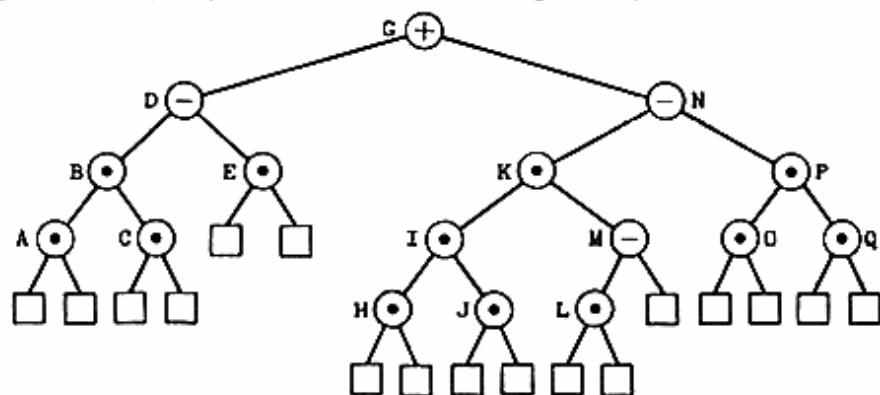
12. The maximum occurs when inserting into the second external node of (12); $C = 4$, $C1 = 3$, $D = 3$, $A = 1$, $F = G = H = 1$, $E = J = 0$, and the JMP in line 67 is taken, for a total time of $132u$. The minimum occurs when inserting into the third-last external node of (13); $C = 2$, $C1 = 1$, $D = 2$, $A = E = F = G = H = J = 0$, for a total time of $61u$. [The corresponding figures for Program 6.2.2T are $74u$ and $26u$.]

13. When the tree changes, only $O(\log N)$ RANK values need to be updated; the "simple" system might require very extensive changes.

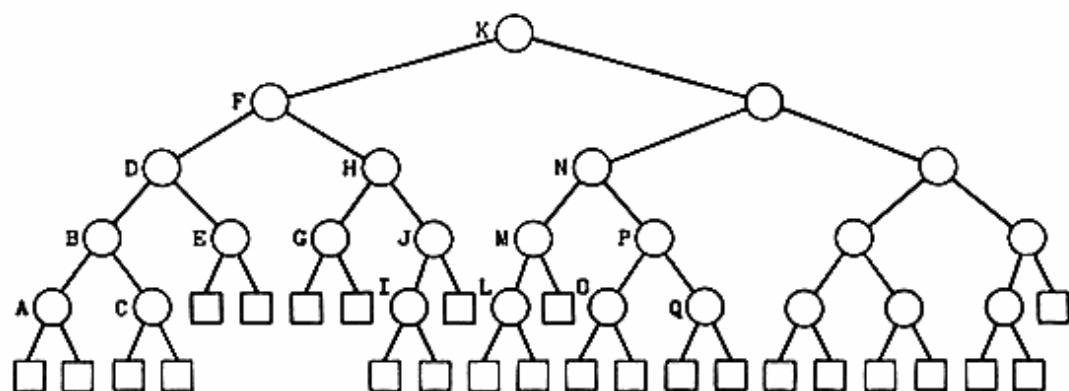
14. Yes. (But typical operations on lists are sufficiently nonrandom that degenerate trees would probably occur.)

15. Use Algorithm 6.2.2T with M set to zero in step T1, and $M \leftarrow M + \text{RANK}(P)$ whenever $K \geq \text{KEY}(P)$ in step T2.

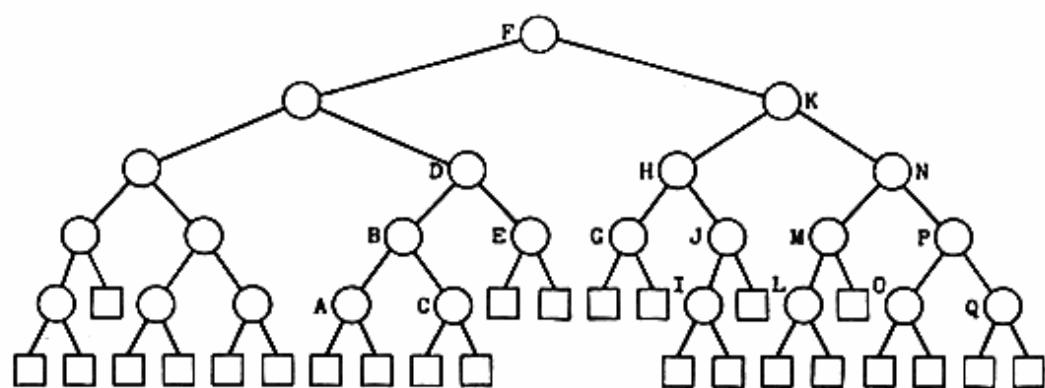
16. Delete G; replace F by G; Case 2 rebalancing at H; Case 3 rebalancing at K.



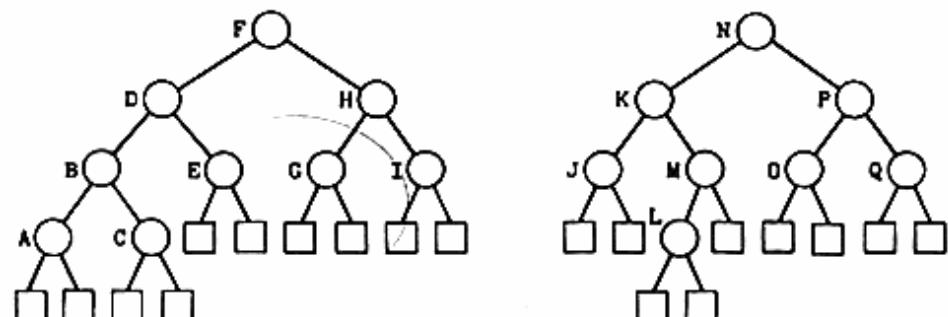
17. (a)



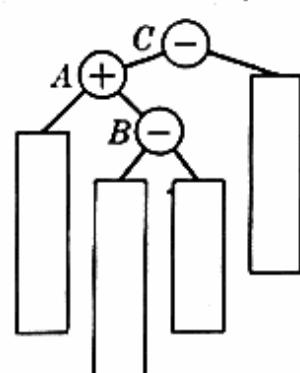
(b)



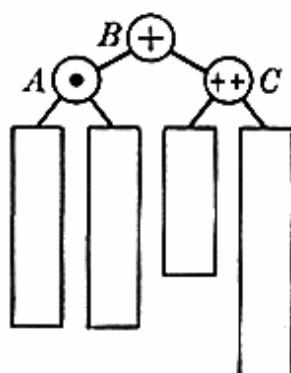
18.



19. (Solution by Clark Crane.) There is one case which can't be handled by a single or double rotation at the root, namely



. Change it to



and then resolve the imbalance by applying a single or double rotation at C.