Search times using hash tables for records with non-unique keys

W. B. Samson* and R. H. Davis†

Recent research in hash coding (Knott, 1975; Maurer and Lewis, 1975; Severance, 1974) has concentrated on unique keys, or uniform distributions of keys. This paper is intended to clarify the effect of non-unique keys with various distributions on search times in the hash table thus enabling recommendations to be made to those who must deal with hash tables of this type. It is found that extreme rank-order frequency distribution of keys, such as the Zipf distribution, result in much higher access times than more uniform distributions, but it is possible to reduce these to some extent by loading records with common keys on to the hash table first.

(Received December 1976)

1. Introduction

1.1 *Hash tables*

A hash table consists of a number of addressable buckets, each of which is capable of holding one or more records. Each record is located in the table by a transformation (or hash) of its key into an address within the range of bucket addresses. For example, for a reasonably uniform distribution of key values it may be possible to divide the key by the number of buckets and use the remainder as the bucket address (the range of key values is, of course, much greater than the range of bucket addresses). This system may be used in any addressable storage medium—the most common application areas being symbol tables for compilers and direct access files.

Clearly, problems occur when the number of records directed to a particular bucket is greater than the bucket capacity. Some method of dealing with bucket overflows becomes necessary. When a bucket overflows, the overflowing records may be dealt with in one of several ways. The quadratic hash overflow method of Maurer (1968) is relatively widely used. This method is efficient and has largely replaced alternative approaches.

1.2 *Non-unique search keys*

Up to the present time, all published work on hash tables assumes either that each record has a unique search key or that non-unique search keys follow a uniform distribution (Bookstein, 1974). The problem of a series of non-unique keys is encountered in many distributions of 'naturally' occurring keys. For example, coded fingerprints form a distribution that is far from uniform, as do surnames and words in running speech; Zipf (1949) found that the frequency of the nth most commonly used word is equal to $\frac{1}{n}$ of the frequency of the most common word

$$\text{i.e. } f_n = \frac{f_1}{n} \qquad (1)$$

where f_i is the frequency of the ith most commonly used word.

In this paper, other frequency distributions of keys are investigated, the need for further research on the subject of non-unique keys having been highlighted elsewhere (Maurer and Lewis, 1975). Some small scale experiments were carried out by Samson (1976b) to examine the need for further research and these confirmed that the access time for a set of records whose keys follow the distribution of equation (1) is likely to be considerably higher than that for a set of records with unique keys. Furthermore it is apparent that if records with non-unique keys are to be loaded on to a hash table, the best search times are achieved by loading the records on to the table in descending order of key frequency. In the light of these simple experiments it was decided to investigate the loading of various frequency distributions of records on to a file of standard size for various loading factors; loading factor being defined as the percentage of the table which is occupied. The objective of the investigation is to determine the conditions under which the strategy of loading the records in descending order of frequency is significantly superior to the random loading of records.

Experiments are described for the loading and searching of a hash table not only for a number of theoretical distributions but also for a number of real distributions of records with non-unique keys.

1.3 *Frequency distributions of non-unique keys*

The theoretical distributions chosen were of the form

$$f_n = f_1/n^\theta \qquad (2)$$

where f_i is the frequency of the ith most frequent key and θ is the negative slope in the log-log plane. This is a special case of the Mandlebrot distribution (1954) which refines the Zipf distribution. θ was chosen to take values 1, ·9, ·8, . . . , ·1 in order to span realistic distributions, three of which were considered, viz.

1. The number of occurrences of surnames in the Isle of Man telephone directory (referred to as the Manx distribution for short).
2. The number of occurrences of keywords in the KWIC index of a set of Building Science Abstracts (KWIC).
3. The number of files kept by users at Heriot-Watt University (H-W).

In order to make the three real distributions and the theoretical ones directly comparable they were each scaled to hold 1000 records.

2. Simulation experiments for theoretical distributions of records with non-unique keys

The table parameters chosen for the simulation experiments were as follows:

Number of records:	1000
Overflow method:	quadratic hash
Loading factors:	10%, 20%, . . . , 95%
Blocking factor:	10 records per bucket
Keys used:	the natural numbers 1, 2, 3, . . . , etc. in descending order of frequency

*Department of Mathematics and Computer Studies, Dundee College of Technology.
†Computer Science Department, Heriot-Watt University, 79 Grassmarket, Edinburgh EH1 2HJ.

Hashing algorithm: bucket address = $(k*p) \bmod n$ where k is the key, p is a prime number and n is the table size.

These choices are justified in the following sections.

2.1 *Number of records*

The number of records used in the simulation experiments was chosen as 1000 for the following reasons:

(*a*) when the mean number of table accesses per search is calculated the standard error of this mean is small compared to the fluctuations from other sources and so it may be ignored

(*b*) 1000 records represents a realistic table size in many practical situations

(*c*) if the number of records chosen was much more than 1000, the tables used in the simulation could not have been fitted into the core of the machine used for the simulation. The use of backing store would then have made simulation run times prohibitively long

(*d*) 1000 is a round figure and so eases the analysis of the results.

2.2 *Overflow method*

The overflow method chosen for the simulation experiments is the quadratic hash method. The particular case chosen is that for which the ith bucket address in the sequence of overflows is

$$(j + \tfrac{1}{2}i^2 - \tfrac{1}{2}i) \bmod n$$

where j is the initial calculated address and n is the table size. This method was chosen because it is simple, efficient, easy to check manually and widely used in practice.

There are, of course, simpler overflow methods, for example linear probing, but these are much less efficient. There are also more efficient methods, for example direct chaining, but these are much more complicated and less widely used in practice.

2.3 *Loading factor*

The loading factor, or density, of a hash table is defined to be the ratio of the number of records held in the table to the overall capacity of the table and is usually expressed as a percentage.

High loading factors normally mean long access times. This effect is felt most strongly for loading factors in excess of 50%.

Critical loading factors occur at different values for the different distribution of records. It is therefore useful to observe access times over a wide range of possible loading factors. The range of values chosen is

10%, 20%, 30%, 40%, 50%, 55%, 60%, 65%, 70%, 75%, 80%, 85%, 90%, 95%

It will be noticed that in the critical range over 50%, the loading factor goes up in increments of 5%.

A subset of this range of values for the loading factor was chosen by Lum, Yuen and Dodd (1971) when they investigated the effect of blocking factors on access times.

2.4 *Blocking factor*

The blocking factor of a table is defined as the number of records a bucket will hold when it is full.

The blocking factor chosen for these simulations is 10 records per bucket. This value was chosen for several practical reasons:

(*a*) the table is represented by an array in the main store of the computer, one array element representing one bucket. With 1000 records, a loading factor of 10% and a blocking factor of 10 the table size is 1000 buckets which fits nicely in the main store as an integer array. If, now, a blocking factor of 1 was chosen, the table size would be 10,000 buckets. This table along with other storage requirements for the simulation would not fit in the main store

(*b*) small blocking factors encourage overflows due to the random collisions of records, even those with rare keys. These overflows tend to mask the effect of collisions of records with the commoner keys, which is the essence of the present study

(*c*) blocking factors in practical direct access files may be of the order of 10

(*d*) the effect on access times of various blocking factors was determined in a series of simulation experiments by Lum, Yuen and Dodd (1971) and it is not difficult to apply these results to the present experiments to determine the effect of using other blocking factors.

2.5 *Keys*

The keys used for the groups of records are the natural numbers 1, 2, 3, 4, . . . , etc. in descending order of frequency. Thus, the most common set of records has key = 1, the second most common has key = 2, and so on. This set of keys is, of course, an ideal one when it comes to spreading them evenly across the table. The reason for choosing such keys was that there would be no 'clumping' for the initially calculated addresses. Such clumping is, of course, hard to predict and its effect is to increase search times by an unpredictable amount. Fluctuations of this kind would merely obscure the results which these experiments are designed to determine.

The effect of using a non-uniform set of keys is independent of the present study but is nevertheless a topic worthy of further research in its own right.

2.6 *The hashing algorithm*

The hashing algorithm used in the simulation is of the form

$$j = (k*p) \bmod n$$

where j is the calculated bucket address (before overflow, if any), k is the key of the record being loaded or searched for, p is a prime number and n is the number of buckets in the table.

This algorithm spreads the hashed keys as evenly as possible across the table. Since n and p are relatively prime, the first n keys are directed to different buckets. When the number of keys exceeds n, the cycle of addresses repeats itself. This process ensures that no two of the n most frequent keys are directed to the same address, thus relieving the problem of collisions between two sets of records with common keys. Such collisions would, again, only serve to obscure the results being sought.

2.7 *Table size*

The choice of table size is crucial when quadratic hash overflow is used. The danger in choosing a bad table size is that the sequence of overflow addresses only reaches a small fraction of the buckets in the table before it starts to repeat itself.

Hopgood and Davenport (1972) have shown that when the table size is a power of 2, all the buckets will be visited before the overflow sequence starts to repeat itself. This holds true for one particular case of the quadratic hash algorithm—the case used in the present work. Ecker (1975) has shown that for any table size, a quadratic hash algorithm may be found which allows all buckets in the table to be visited.

The size of the table determines the loading factor for a given number of records. For each of the loading factors indicated in Section 2.3, the corresponding table size was calculated. If the calculated table size did not have a long overflow sequence, then the closest table size with a reasonable overflow sequence was chosen, using a method described by Samson (1976a).

2.8 *The simulation*

Three programs were used for the simulation of the behaviour of the theoretical distributions of records in a hash table. The first simulates table behaviour when the records are loaded in random order, the second when records are loaded in descending order of frequency and the last when records are loaded in ascending order of frequency. The programs only differ in the order in which the records are blocked, see Samson (1976b).

The output from each simulation run includes the following:

(*a*) a histogram showing the distribution of records in the loaded table

(*b*) the mean number of buckets to be accessed during the searches

(*c*) the loading factor.

If an endless overflow cycle was entered during the loading of the table, the message ENDLESS BLOCKRUN was printed. If loading was completed successfully but an endless overflow cycle was entered during searching, the message ENDLESS SEARCH was printed.

Simulation experiments were performed for the ten theoretical distributions for each of the fourteen loading factors. The set of experiments was performed seven times for various values of the prime number p in the hashing algorithm (Section 2.6). There are thus seven results for each experiment. The median value and interquartile range of each result may be determined from these results.

The analysis of the results is dealt with in Section 4.1.

3. Simulation experiments for three natural distributions of records with non-unique keys

The three natural distributions referred to in Section 1.3 were scaled so that each distribution contains, as nearly as possible 1000 records, thus making simulation results directly comparable with those from the theoretical distributions.

The hash table used for these simulation experiments is identical in all respects with that described in Section 2 for the simulations of theoretical distributions, that is, hashing algorithm $j = (k*p) \bmod n$, blocking factor = 10, and loading factors covering the range 10% to 95%. Quadratic hash overflow of the form ith address $= j + \frac{1}{2}i^2 - \frac{1}{2}i$ is chosen. Justification of the choice of table parameters is set out in Section 2.

The programs used for the simulation of the behaviour of the natural distributions of records are similar to those used for the theoretical distributions, and the output format is identical.

In particular, the mean number of table accesses per search is given for each loading factor.

One simulation run was performed for each of the natural distributions over the range of loading factors used. It was not found necessary to perform a series of simulation runs for different hashing algorithms because the results from a set of runs are sufficient to indicate that the natural distributions behave in an exactly similar way to the theoretical ones. In all cases, the duration of a simulation run was 30 minutes. Such a run covered the full range of 14 loading factors for a given input distribution of records.

4. Simulation results

4.1 *Analysis of theoretical distributions*

The results of the simulation experiments described in Sections 2 and 3 were punched on to cards and a set of procedures written to analyse these results and to plot graphs of them. **Figs. 1, 2** and **3** show isometric projections of the three dimensional graphs of the results of the simulation experiments for the theoretical distributions. The three axes are:

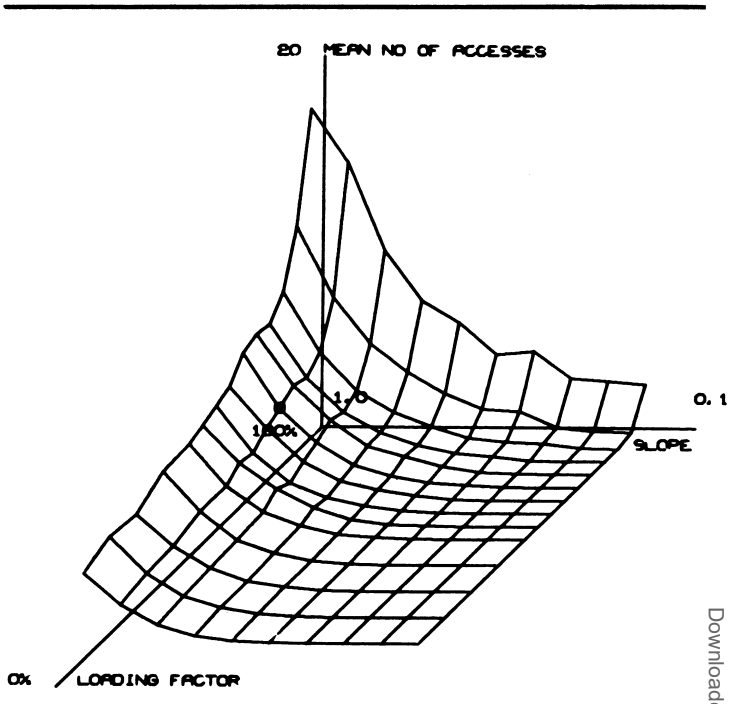

Fig. 1 Random order of blocking

(*a*) the negative slope of the frequency distribution of records, in the log-log plane, ranging from 0·1 to 1·0

(*b*) the loading factor, ranging from 10% to 95%

(*c*) the mean number of accesses per search (median value over seven experiments).

Fig. 1 shows results when records are loaded in random order, Fig. 2 when records are loaded in ascending order of frequency and Fig. 3 when records are loaded in descending order of frequency.

It can be clearly seen that at high values of slope and high loading factors there is a clear-cut separation between the mean access times for the three orders of loading the table.

Loading records on to the file in descending order of key frequency gives much shorter access times than loading them in random order, for combinations of high slope and high loading factor.

Loading records on to the file in ascending order of key frequency gives correspondingly worse results.

Since the table parameters used in these experiments are typical of those encountered in practice, it is of interest to determine when the strategy of loading records in descending order of key frequency is significantly better then the random order strategy.

It should be noticed that the mean number of accesses per search does not tend asymptotically to 1·0 as the loading factor is reduced. This is because the larger groups of records sharing the same key will always fill several buckets. For example if key k_i is shared by f_i records, the number of buckets occupied will always be greater than or equal to $\frac{fi}{b}$ where b is the blocking factor. In the limit, as loading factor tends to zero, the mean number of buckets accessed in a search will be

$$\frac{\sum_{i=1}^{n} f_i \left(\text{entier}\left(\frac{f_i}{b}\right) + 1\right)}{\sum_{i=1}^{n} f_i}$$

where n is the number of distinct keys.

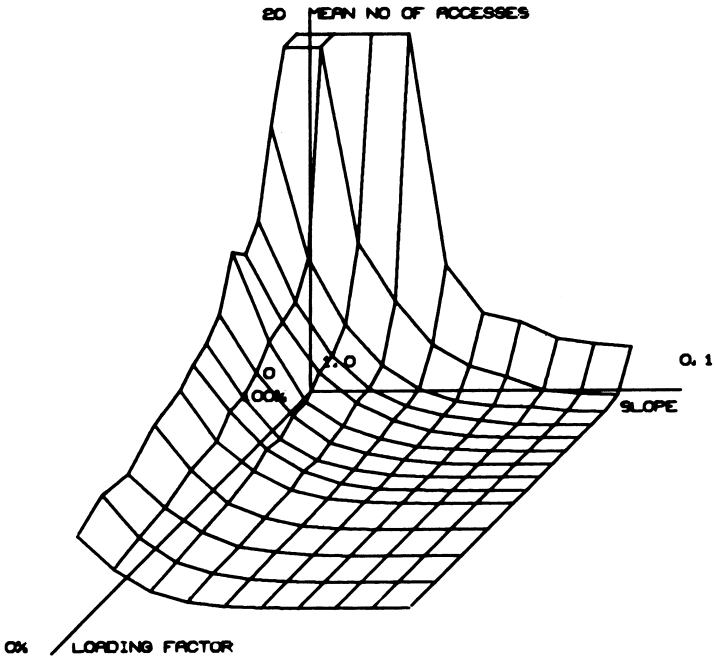

Fig. 2 Rare keys blocked first

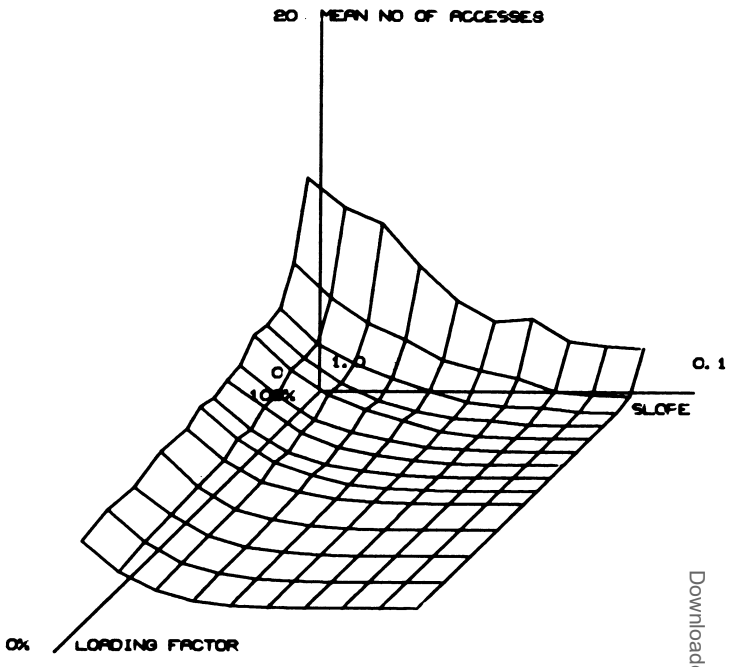

Fig. 3 Frequent keys blocked first

4.2 *The median test*

In Section 2 an account was given of the experiments, using different hashing algorithms, performed for a range of table parameters. The central tendency and dispersion of the mean number of accesses made during search were measured using medians and interquartile ranges respectively. The median test was used to determine whether the strategy of loading records in descending order of frequency is significantly better than random loading.

Fig. 4 shows that region of the load-slope plane and the points where a significant improvement was found. It is of interest that some points which might be expected to show a significant improvement (e.g. slope = 0·8, loading = 90%) do not, although the medians found do indicate an improvement. This is due to the high interquartile range of the values found and of course to the small number of experiments.

Despite this, there are sufficient points with this degree of significance to show that the effect is a real one.

In general, when the slope is 0·7 or greater, there is a good chance that a significant improvement in table performance may be obtained by loading records on to the table in descending order of frequency.

In order to assess the benefits of using the best loading strategy the amount of improvement in median access time is indicated in Fig. 4 by specific symbols within the circled points. '*' represents <30% improvement, '+' represents 20%-30% improvement, '×' 10-20% improvement and no symbol represents a point where less than 10% improvement was found.

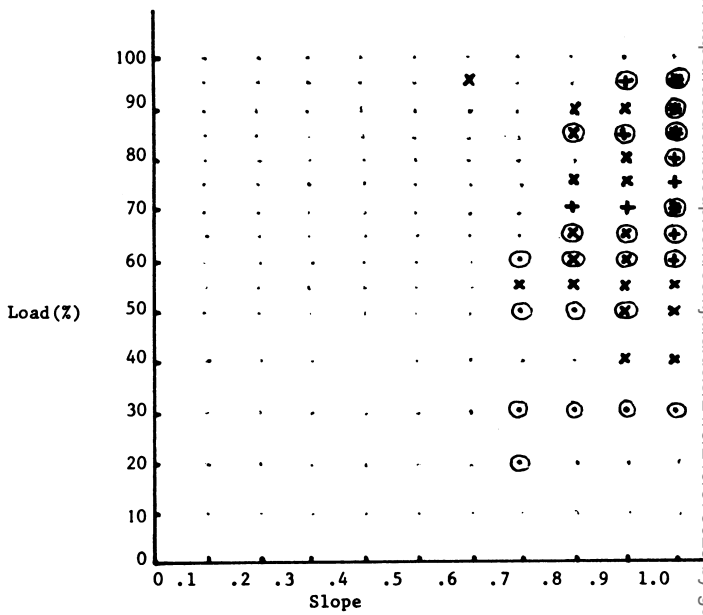

Fig. 4 Showing points (circled) where 'frequent first' loading strategy is significantly better than random loading

4.3 *Comparison with natural distributions*

Results for the three natural distributions referred to in Section 1.3 can be compared on the log-log plane to the results for the theoretical distributions. The theoretical distribution results with negative slope of ·6 gave a good fit to both the Manx and KWIC distribution results, while the theoretical distribution results with negative slope ·85 fitted the H-W distribution results.

Given any natural distribution, therefore, it would be easy to predict its behaviour simply by fitting a straight line in the log-log plane and measuring its slope. The natural distribution of records would behave in a similar way to the theoretical distribution with this slope.

5. Conclusions

The results of the simulation experiments described show that for a hash table containing records with non-unique keys a significant improvement in performance can be achieved by loading records in descending order of frequency of their keys, provided the negative slope of the distribution in the log-log plane is high. If, on the other hand, the negative slope of the distribution is low, there is no significant improvement in performance and the expense of loading records in descending order of frequency would not be justified. The area in which a significant improvement in performance might be expected is shown in Fig. 4.

It is possible to make the following recommendations to systems analysts and programmers who must deal with hash tables involving records with non-unique keys:

1. determine the shape of the rank order frequency distribution of records with the same key, by examining a random sample of the records.

2. if the negative slope of this distribution in the log-log plane is greater than, say, 0·6 then perform a simulation as described below.
3. The simulation should involve the same number of records, loading factor, blocking factor and overflow method as the hash table itself. The distribution of records found in (1) above should be scaled appropriately.
4. The simulation should be performed several times for various hashing algorithms and the median test applied to determine whether the performance is significantly better when records are loaded in descending order of frequency.
5. If a significant difference is found in step (4) above then provision should be made for loading the table in descending order of frequency. If the table is a dynamic one, e.g. a random access file to which records are constantly being added and deleted, a housekeeping operation should be performed periodically to reload the current set of records in descending order of frequency.
6. If the negative slope of the distribution in the log-log plane is less than, say, 0·6 or no significant improvement in performance was found in step (4) then the records may be loaded in random order, without fear of a significantly degraded performance.

6. Acknowledgements

The authors wish to thank Professor Stocker, School of Computer Studies, University of East Anglia, for his helpful comments, Mr. G. Rutherford, of Department of Mathematics and Computer Studies, Dundee College of Technology, for his advice in the use of statistical tests, Mr. J. R. Lowe of the Home Office, Scientific Advisory Branch who first brought the problem of non-unique keys and ways of handling them to our attention, the Board of Governors of Dundee College of Technology for financial help and the Director and staff of Dundee College of Technology, Computer Centre for their help in preparing and running the programs.

References

BOOKSTEIN, A. (1974). Hash Coding with a Non-Unique Search Key, *J. Amer. Soc. Inf. Sc.*, July and Aug., pp. 232-236.
ECKER, A. (1975). The Period of Search for the Quadratic and Related Hash Methods, *The Computer Journal*, Vol. 17, No. 4, pp. 340-343.
HOPGOOD, F. R. A. and DAVENPORT, J. (1972). The Quadratic Hash Method When the Table Size is a Power of 2, *The Computer Journal*, Vol. 15, No. 4, pp. 314-315.
KNOTT, G. D. (1975). Hashing Functions, *The Computer Journal*, Vol. 18, No. 3, pp. 265-278.
LUM, V. Y , YUEN. P. and DODD, M. (1971). Key-to-address Transform Techniques: A Fundamental Performance Study on Large, Existing Formatted Files, *CACM*, Vol. 14, No. 4, pp. 228-239.
MANDELBROT, B. (1954). Structure Formelle des Textes et Communication, *Word*, Vol. 10, pp. 1-27.
MAURER, W. D. (1968). An Improved Hash-code for Scatter Storage, *CACM*, Vol. 11, No. 1, pp. 35-38.
MAURER, W. B. and LEWIS, T. G. (1975). Hash Table Methods, *Computing Surveys*, Vol. 7, No. 1, pp. 5-19.
SAMSON, W. B. (1976a). Testing Overflow Algorithms for a Table of Variable Size, *The Computer Journal*, Vol. 19, No. 1, p. 92.
SAMSON, W. B. (1976b). M.Sc. Thesis, Heriot-Watt University, Edinburgh.
SEVERANCE, D. G. (1974). Identifier Search Mechanisms, *Computing Surveys*, Vol. 6, No. 3, pp. 175-194.
ZIPF, G. K. (1949). *Human Behaviour and the Principle of Least Effort*, Addison-Wesley.

Book reviews

Fundamentals of Data Structures, by E. Horowitz and S. Sahni, 1977; 564 pages. (*Pitman*, £5·95).

This text is intended for use in a data structures course for students with some previous programming experience. Its scope is standard and may be summarised by the chapter headings—after an introduction, chapters follow on arrays, stacks and queues, linked lists, trees, graphs, internal sorting, external sorting, symbol tables and files.

This is a comprehensive treatment of standard topics with the overall approach kept so informal that parts are not sufficiently rigorous. Many algorithms are given, with some accompanying correctness proofs and performance analyses that might usefully be more formal. The algorithms are written in Sparks, the authors' own ALGOL-like programming language—Sparks is not completely defined, and attempts to run some of the algorithms given in the text using the Sparks to FORTRAN translator discussed in Appendix A would seem fraught with difficulties. The authors do pay some attention to good structured programming practices, and also to the idea that data structures should be defined in terms of the operations applicable to them—this important point may be lost on some students since the notation used is only briefly discussed. Numerous exercises are included with no indications of difficulty or suggested solutions.

The book contains much useful material, but the overall effect is frequently spoilt by use of a deliberately informal writing style. This is a useful additional source of ideas for anyone planning a data structures course, but seems of limited value as recommended reading for students.

PETER WALLIS (Bath)

A Concurrent PASCAL Compiler for Minicomputers, by A. C Hartmann, 1977; 119 pages. (*Springer-Verlag*, Lecture Notes in Computer Science, 50, DM 18,— about £4·50)

This monograph is a description of a seven pass compiler for Per Brinch Hansen's Concurrent PASCAL. It is a postgraduate text which will be of interest to those concerned with the implementation of PASCAL and languages derived from PASCAL.

The overall structure of each pass is presented: lexical analysis, syntax analysis, name analysis, declaration analysis, body analysis, code selection and code assembly. The final code is interpreted mainly because of the synchronisation primitives. Such interpretation aids portability at some cost in processor usage. The detail in the description is rather uneven, giving a complete syntax for the input and output of each pass but no indication of the semantics of the final machine code. Concurrent PASCAL is not a superset of PASCAL, it contains no nested procedures or variant records. The compiler itself is written in Sequential PASCAL which *does* contain variant records (which are used heavily in the implementation). The divergence between the two PASCALS is unfortunate as the author notes with the lack of **classes** in Sequential PASCAL. An interesting statistic from the implementation is that 28% of the initial errors in the compiler were detected by the interpreter as **variant** access errors, i.e. accessing a field of a variant when the tag field is not set correctly. Since ordinary PASCAL implementations do not check for this, does this mean that PASCAL is not the reliable language it is supposed to be?

This book will appeal to a small but appreciative audience.

B. A. WICHMANN (Woking)