A compaction procedure for variable-length storage elements

By B. K. Haddon* and W. M. Waite†

When a dynamic storage allocation scheme requires variable-length elements, an element of a given length may be requested when no free element large enough is available. This can happen even though the total free space is more than adequate to fulfil the request. In such a situation the program will fail unless some method is at hand for forming the available space into a single element large enough to satisfy the request. We present a procedure for compacting the store such that all of the free space forms a single element.

Introduction

One of the most important problems in the construction of a dynamic storage allocation scheme is that of recovering storage space which is no longer being used by the program. The mechanics of detecting the existence of such "garbage" have been adequately described (Newell *et al.* (1964); McCarthy, 1960; Collins, 1960; Schorr and Waite, 1965). In the case where the elements of storage required by the program are of variable size (Comfort, 1964; Ross, 1961; Knowlton, 1965) it may happen that an element of a certain size is needed and, although sufficient free space is available, no free element which is large enough can be found. For example, suppose that 20,000 words of memory were allotted as a storage area to contain one- and two-word elements. If the first, third, . . ., 19999th words were all occupied by one-word elements, it would not be possible to obtain a two-word element even though 10,000 words were free. What should be done in this case is to "compact" the store by moving all one-word elements to one end, leaving a single 10,000 word block of free space.

To our knowledge, none of the papers describing variable-length storage elements deals with this problem. Perhaps, as has been intimated in discussion of the subject, it is felt that a procedure to compact the store is too expensive to include in the storage allocation scheme. Here we emphatically disagree! Compaction is only used when the program would otherwise fail, and in such circumstances no expedient is too expensive. The method which we shall propose requires no I/O devices, and only a small, fixed amount of temporary storage. (We assume that the smallest storage element contains at least enough space for two addresses and a flag.)

One of the characteristics of the storage elements under consideration is that they are linked by address pointers which must be relocated when compaction takes place. We assume that the relative positions of these relocatable fields in a given element can be determined, possibly by relocation bits in the element or by some type code which specifies a format.

The procedure

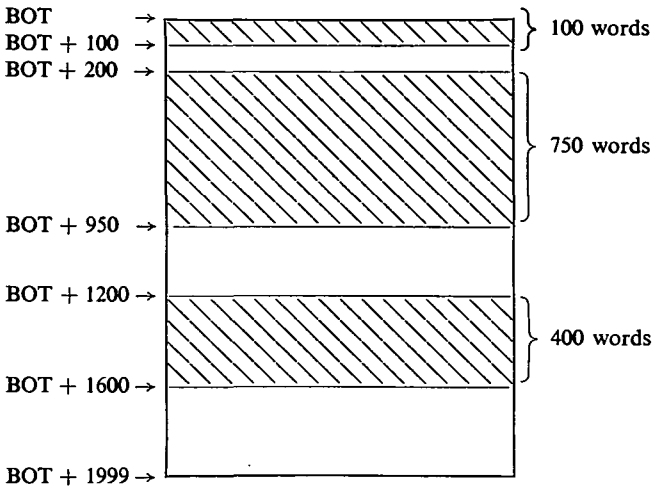
Suppose that, during the course of program execution, we require a storage element which is larger than any available on the free list. The first step is to perform a normal garbage collection, forming a new free list with elements as large as possible (Schorr and Waite, 1965). As the free list is being formed, a count of the total number of free words should be kept. If there is no free element which is large enough to satisfy the requirement, and yet there are enough free words, then the compaction procedure is entered.

By consulting the free list, we can mark all free storage. Using this information we move all non-free elements to a compact block at the lower end of the storage area, and at the same time build up a table for use in relocation. For each block of consecutive non-free elements, this "break table" has one entry giving the address of the first word of the block and the number of free words below the block. **Fig. 1** shows an example of the break table corresponding to a particular state of the storage area: If the hatched elements in Fig. 1(*a*) have been marked as being free, then the break table will be that shown in Fig. 1(*b*).

After the non-free elements have been moved and the break table constructed, all relocatable fields are updated by looking up their contents in the break table and subtracting the number of free words below that address. If a relocatable address is not equal to any table entry, the number of free words associated with the next lower table entry is used; any address lower than the lowest entry is unchanged. In Fig. 1, for example, a relocatable field which contained the number BOT + 950 would be altered to contain BOT + 100 (= BOT + 950 − 850); a field containing BOT + 990 would be changed to BOT + 140.

In the introduction we asserted that the compaction procedure did not use I/O devices, and required only a small, fixed amount of temporary storage. A simple inductive argument can be used to demonstrate the truth of that assertion: First note that the situation depicted in Fig. 1 always holds for the uncompacted store. That is to say, the first element of the uncom-

**Basser Computing Department, University of Sydney, Sydney, N.S.W., Australia.*
†*Department of Electrical Engineering, University of Colorado, Boulder, Colorado, U.S.A.*

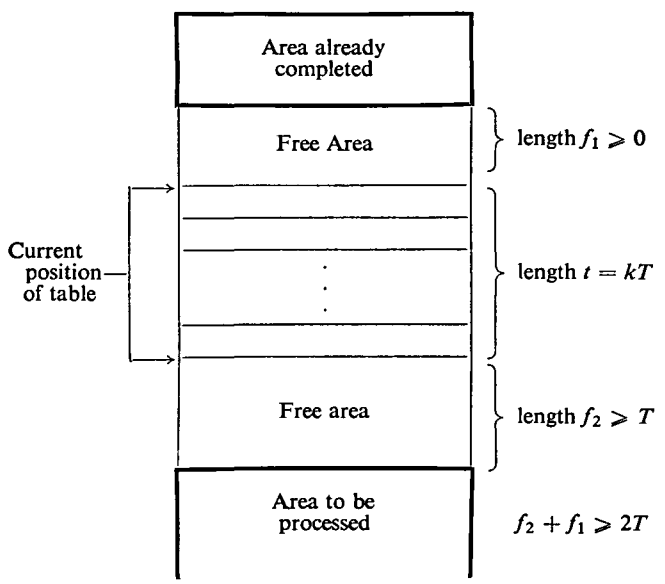


(*a*) A state of a 2,000-word store

Address	Free words
BOT + 100	100
BOT + 950	850
BOT + 1600	1250

(*b*) Break table corresponding to (*a*)

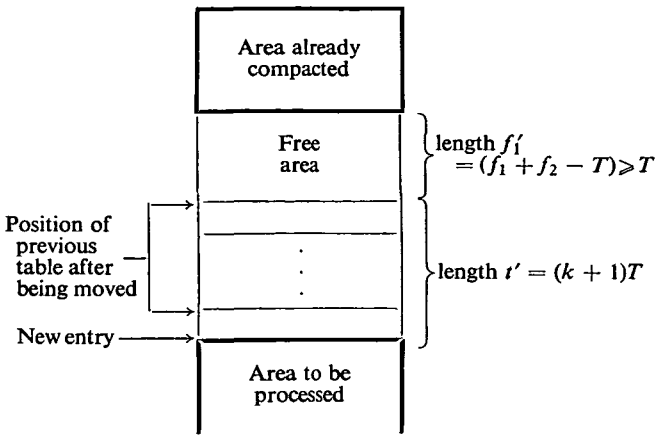
Fig. 1.—Example of a break table



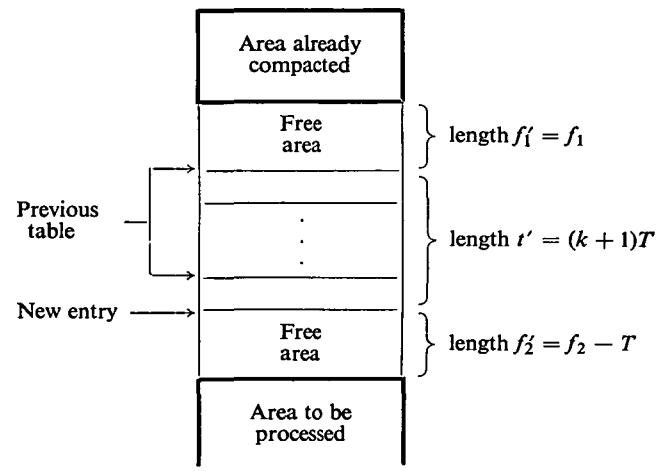
T = length of one table element

Fig. 2.—State of the store after moving a non-free block

pacted store will always be free. (For if it were not free there would be no need to move it, and hence we could begin the compaction procedure with the next element.) The temporary storage area is capable of holding one table entry, and the compaction process begins by finding the first non-free block and placing



(*a*) State of store after moving the table (case 1)



(*b*) State of store after adding new table entry (case 2).

Fig. 3.—Adding the next table entry

its table entry in this area. The block is moved down, filling the free space below it. The store is then in the state shown by **Fig. 2,** with $f_1 = k = 0$. If f_2 is the length of the next free space, then $f_2 \geqslant 2T$ because it is made up of two continguous free areas, each of which must be at least large enough to hold one table entry. (Recall that a table entry is no larger than two addresses, and the smallest storage element has space for two addresses and a flag.)

We now complete the induction by assuming that at some stage the store has the form of Fig. 2. Two cases may be distinguished: (1) $f_1 = 0, f_2 \geqslant 2T$, and (2) $f_1 > 0$. In case (1), we place the table entry for the next non-free block immediately below the element which it describes, and move the rest of the table up to it. The state of the store is then given by **Fig. 3***a*. Case (2) does not require that the table be moved, and the entry describing the next non-free block is placed immediately above the existing table as shown in **Fig. 3***b*. In either case $f_1' + f_2' \geqslant T$ because $f_1 + f_2 \geqslant 2T$. The next step is to move the non-free block down into the space

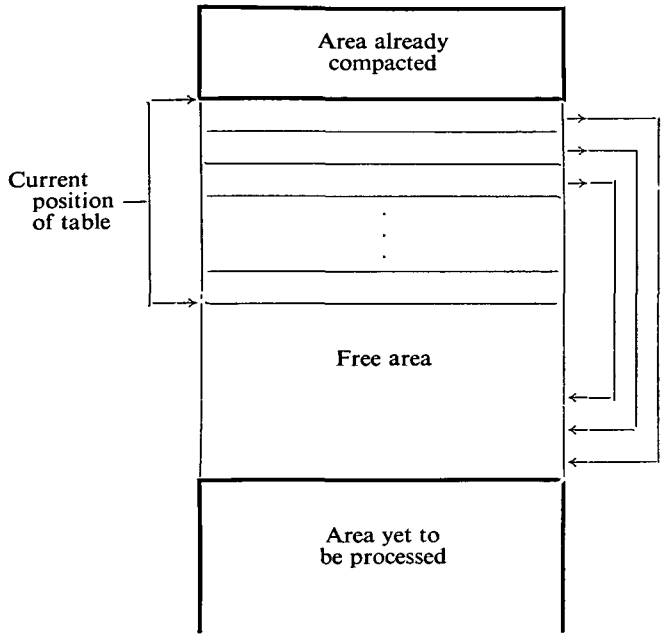


Fig. 4.—"Rolling" the table

Table 1

Compaction times for various element lengths

LENGTH OF ELEMENT	TIME TO COMPACT AND SORT	
1 word	2·87 sec	0·478 × 10^6 cycles
2	1·26	0·210
3	0·94	0·157
4	0·69	0·115
5	0·57	0·095
6	0·52	0·087
7	0·44	0·073
8	0·42	0·070
9	0·39	0·065
10	0·34	0·057
20	0·24	0·040
40	0·19	0·031
80	0·16	0·027
100	0·15	0·025
1000	0·10	0·017

below the table. If the block length is less than f_1' there is no problem; if it is greater, then when the space f_1' is full there will be a contiguous free area above the table which is at least as large as one table entry (because $f_1' + f_2' \geqslant T$). We therefore move the table up as far as possible before continuing to move the non-free block down. When the non-free block has been completely moved, the store will be in the state of Fig. 2. To see this, recall that before the move, we had a free space $f_1' + f_2' \geqslant T$ below the non-free block. After the move, this free area and the table are adjacent to the next free block, which is at least of length T. Thus the total free space adjoining the table is $2T$, as shown in Fig. 2.

The inductive argument given above shows that we will always have room to add another entry to the table, as long as we uncover a new free block. Suppose, however, that the last block of memory is not free. We are able to compact this block, and move the table if necessary, but we will not get another free block. This is not serious, however, because we need not add any more entries to the table. The free space adjoining the table has length greater than T, so that we can recall the first table entry from temporary storage.

The most crucial point in a practical realization of the algorithm is to minimize the number of moves involving the table entries. If the table is "rolled" (see **Fig. 4**: entries are moved from the front to the highest available space, continuing until either the whole table is moved or the free space above the table is filled), one can guarantee that only one table entry will be moved *into* a given position. An upper bound on the number of moves during compaction is thus (size of store)/T. Additional movement of table entries occurs when the table is sorted just before relocation.

Results

The compaction procedure described in this paper has been successfully run on the English Electric KDF 9, in conjunction with an extension of the WISP (Wilkes, 1964; Orgass *et al.*, 1965) list processor. The program occupies 58 words, and takes 2·87 seconds to compact a 10,000-word store in the worst case (single word elements, alternate elements free).

In order to determine the effect of element size, we tried compaction with alternate elements free and element sizes between 1 and 1000 words. In each case the length of a table entry was one word, and the length of the storage area was 10,000 words. The results are shown in **Table 1**; all of these times include the sorting of the table. A rough estimate of the time required on other computers may be obtained by multiplying the number of cycles given in Table 1 by the main store cycle time of the machine.

We wish to emphasize once more that compaction is only used when the program could not otherwise continue. The procedure is *not* invoked with each garbage collection. The results summarized above indicate that this procedure is a cheap form of protection to include in any dynamic allocation scheme which uses variable-length elements.

References

NEWELL, *et al.* (1964). *Information Processing Language—V Manual*, 2nd Edition, Prentice-Hall, Englewood Cliffs, N.J.

McCarthy, J. (1960). "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I", *Comm. ACM*, Vol. 3, p. 184.
Collins, G. E. (1960). "A Method for Overlapping and Erasure of Lists", *Comm. ACM*, Vol. 3, p. 655.
Schorr, H., and Waite, W. M. (1965). *An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures*, Research Rept. RC-1450, International Business Machines Corp.
Comfort, W. T. (1964). "Multiword List Items", *Comm. ACM*, Vol. 7, p. 357.
Ross, D. T. (1961). "A Generalized Technique for Symbol Manipulation and Numerical Calculation", *Comm. ACM*, Vol. 4, p. 147.
Wilkes, M. V. (1964). "An Experiment with a Self-Compiling Compiler for a Simple List Processing Language", *Annual Review in Automatic Programming*, Vol. 4, p. 1.
Orgass, *et al.* (1965). *WISP—A Self Compiling List Processing Language*, Technical Rept. 36, Basser Computing Department, University of Sydney, Sydney, Australia.

Data compression and automatic programming

By A. G. Fraser*

Data compression is defined as the reduction of the volume of a data file without loss of information. Methods of obtaining this effect are considered and the implications for automatic programming systems are discussed.

In certain commercial undertakings it is necessary to process data which is not always of fixed format or size. Commonly, names and addresses of exceptional length are held, although the majority are of only modest proportions. Also a data record may contain the occasional but bulky item of additional information which, for the normal case, is totally irrelevant. Variations such as these do not normally present serious problems in a manual data processing system but they must be viewed with some concern when mechanization is introduced.

Operating costs for an automatic data processing system depend heavily upon the volume of data held and the extent of the computing activity which is required per event. It is usually possible, however, to reduce the volume of data but only at the cost of increased complexity in the computing process, and there are a number of ways in which this is commonly done.

(*a*) *Redundancies*

It may be possible to reduce the volume of redundant information carried by the system, but it may then be necessary to re-compute certain values when they are required.

(*b*) *Coding*

By the use of special codes it may be possible to reduce the size of a data item without loss of information. This method may be worthwhile where the number of distinct values that can be taken by an N-bit item is considerably less than 2^n but its use will probably involve table accesses during processing.

(*c*) *Special cases*

By the use of special representations for special values it may be possible to take advantage of known patterns in the data. One example is the special treatment of zero items and "not applicable" groups. Another example is the storage of only the significant characters in an alphanumeric string or the storage of only the significant members of a multi-occurrence group.

The user must strike a balance between data volume and procedural complexity in order to get maximum return from his data processing machinery. The choice of method is one that can only be made in the light of a full understanding of the nature of the job and the demands to be met. Such a choice can only be made by the user and cannot be made automatically on his behalf without risk of severe inefficiency. However, automatic programming methods could well assist the user once the basic decision has been taken, since the application of these techniques for data handling is a well defined operation.

Compiler source language

Methods of data compression vary but for many there are two clearly defined stages in their application. First, the method is defined by a series of rules which must be obeyed rigorously if the method is to work. Secondly, each and every data reference must be handled in accordance with the rules applicable to the item concerned. An automatic programming system can be of service to its user by rigorously applying such rules. Indeed the mechanical application of rules of this type is almost

* *University Mathematical Laboratory, Corn Exchange St., Cambridge, England.*