CGTM #35
January 2, 1968
J.R. Ehrman

Garbage Collection in Hash-Addressing List Processors

In a list-processing system without reference counts such as LISP,[1] there is typically a "master list" containing as sublists all currently active information. When an element on this master list must be found, a sequential search must be made for the argument; once found, the corresponding value may afterward be referenced by its address. This simplifies the process of garbage collection, since the master list and its substructures can be traversed and marked, with unused and inactive cells then being returned to free storage.[2]

The use of hash-coding techniques to reduce access time for associatively-addressed data is well known. Typically, a suitably randomized "bucket address" or "home address" is generated from the argument, and a sequential search begun from that address. When entering data, the argument and its value are placed in the first available cell, with occupied cells having been passed over. Thus the average access time for a given element can be considerably reduced, and the argument once located may subsequently be referenced by its address, at least from those substructures of the master list which know that address. A problem arises after garbage collection is complete: if cells

are vacated between the actual location of some element and its home address, any later search for that element will find a vacant cell first and determine therefore that the element is not present. It is not sufficient to simply move the element back to the available position closest to its home address, since other lists which refer to the element by its address will now point to the incorrect location.

The situation just described is due to the lack of an inverse of the hash function: that is, there is no way to tell from a value what the hashed address corresponding to the argument should be. We present below two solutions to this garbage collection problem, in which all available free space will be reclaimed, and the addresses of active elements will still refer to one another correctly. The key to each method is that provision is made to determine the bucket to which a cell belongs.

I. Hashing Directly into Free Space

In this scheme, assume that a known and fixed area of memory has been designated for storage of list structures, and contains cells with the following components in addition to the usual fields for type, successor addresses, etc.:

A a flag used during garbage collection to indicate that the cell contains an active element;

F a flag which indicates whether the cell is free (if a cell is free it is inactive);

D a field containing the displacement of the cell from its home address.

The maximum allowable value of D will be called M. We will also assume that there are L+M (initially free) cells in memory, of which the first L are

addressed by the hashing function. The method of entering elements into the table requires that the desired bucket or cell be addressed from the hash function; if the cell is empty, data is entered into it, F is set off, and D is set to zero. If the cell at the home address is occupied, a sequential search is made in the direction of increasing addresses until (1) an empty cell is found, in which case the cell is filled and D is set to the difference between the actual address and home address, or (2) no empty cell is encountered before M cells have been scanned, in which case a garbage collection must be initiated.

First, all cells are marked inactive, and then (by any standard method of list traversal) the active cells on the master list are flagged. The actual garbage collection then starts at the lowest-addressed cell, and tries to move each active cell downwards to be as close as possible to its home address (see the flowchart in Figure 1).

1. Scan forward from the home address, remembering the lowest-addressed inactive cell encountered, looking for an active cell which belongs at the home address.
2. If such a cell is found and no free cell has been encountered yet, continue scanning until M cells have been scanned.
3. If there is an available inactive cell below the active cell, move the active cell down from the "old" address to the "new" address and adjust its displacement to reflect the change. Mark the old cell inactive. Then scan all active cells for addresses referring to the "old" address and change them to the "new" address.
4. Resume the forward scan for inactive cells at the cell following the most recently-filled "new" location until the M cells have been scanned.

5. When L home addresses have been examined, mark all inactive cells free.

Aside from the evident slowness of garbage collection, there is a possibly severe limitation to this method: if after a garbage collection there is still no free cell within M cells of the home address, the computation must be abandoned. A large value for M can relieve some of this difficulty, but then more space must be allotted in each cell for D, and a larger M-cell area must be attached at the end of the L-cell area addressable by the hash function. Hence the method is best adapted to applications where the list structures will be fairly sparse in the free-cell space; its major advantage is that the range L of the hash function can be quite large, giving very rapid access to a desired element.

It is possible to avoid the extra block of M cells and the possible inability to store data (due to excessive hashing to the same address) by (1) arranging the cell storage in circular fashion, and (2) expanding the field for D so that it can contain a number as large as L. The garbage collection would then be unnecessary until all cells were filled; it would still be relatively lengthy, however, since a single pass around the "circle" of home addresses is not sufficient to assure that all cells are as close as possible to their home addresses, with no intervening gaps. The number of iterations required could be reduced somewhat by requiring that a garbage collection be initiated when some predetermined number much smaller than L of filled cells has been passed over in the search for an empty space.

A modification can be made to this method[3] which (at the cost of increased search times) simplifies the garbage collection problem, and eliminates the need for the field D. As before, the garbage collection begins by marking all (non-free) cells inactive, and then the active cells are marked by traversing

the master list. At this point, the garbage collection is complete. If an entry is now sought, its home address is generated by applying the hashing function to the argument, and a search is begun. If an inactive but non-free cell is encountered before the desired element, it must be passed over and ignored until (1) a free cell is found, in which case the element is known not to be in the table, or (2) the element is found, or (3) the end of the table is reached. In cases (1) and (3) the element is placed in the first inactive cell encountered in the scan which began at the home address.

It can be seen that garbage collection is as fast as for any standard scheme of marking the active elements of a list. The speed of the processor itself depends on the fraction of the cell storage area which has been used: for sparsely-distributed lists there is relatively little searching to be done, whereas heavy use of the cell storage area will result in average search times of the same order of magnitude as required for an ordinary sequential search. Processing speeds will decrease as the density of free cells decreases.

II. Hashing into a Directory

In this method there are two distinct but intertwined structures, one being that appropriate to the list-processing system. The other is used for storage management purposes only, and identifies the bucket to which a cell belongs: thus each bucket will contain not a single element but a <u>list</u> of elements.

The storage space is divided into two areas, one fixed area with D entries for the directory, and the other for cell storage, which need not be a single unfragmented piece of memory. The cells in the cell area contain two components <u>in addition to</u> the usual fields for type, successor addresses, etc.:

A an activity flag as described above;

C an address field used for chaining the bucket and free storage lists (a zero address will indicate the end of a list).

Initially all directory entries are zero, and all cells in the various cell storage areas are chained in a standard free storage list (see Figure 2a). To store or search for an element, the hashing function is used to generate a directory index k in the range from 1 to D. The k-th directory entry is then a pointer to the head element of a list of all elements with the same directory index k. This list is then scanned for the desired argument, using the chain address C to obtain successive elements. If an element is to be added to the bucket list, a cell is obtained from the free storage list and chained to the bucket list, and any necessary data is enterd into the cell (see Figure 2b).

When the free storage list is exhausted (Figure 2c), the garbage collection begins with the activity flag A in all cells being set off: all cells are marked inactive. A standard traversal of the master list is then made, and all active elements are flagged. The freed cells are then returned to the free storage list: rather than scanning the cell space sequentially, we instead begin at each directory entry and work down the chain of addresses of the bucket list, returning inactive cells to the free storage list and shortening the chains of the active elements. If all the elements in the bucket list are inactive, the directory entry is simply reset to zero (Figure 2d). When all the directory entries and their associated bucket lists have been examined, the garbage collection is complete.

It can be seen that garbage collection using this method will require an amount of time which is only a constant multiple of time required for a standard LISP-like system in which free cells are reclaimed in a sequential pass

over the cell space; each cell is still interrogated only once, but more pointer manipulation is required in rearranging the bucket lists. The average number of such elements in a bucket list is essentially proportional to the reciprocal of the directory size. Thus, so long as the directory occupies a relatively small fraction of the total memory space available, doubling the directory size will roughly halve the average access time of an element. In particular, this scheme clearly represents a considerable improvement over the sequential search of a single list. The disadvantage of this scheme is that space for an extra address is required for each cell, which in current systems could reduce the number of cells available by as much as a factor of two. Correspondingly, the fact that larger cells are required might allow a simpler treatment of constants, which sometimes must be placed in a separate area of memory and be garbage-collected in some different manner from that used for ordinary cells.

References

1. LISP 1.5 Programmer's Manual, MIT Press (1962).

2. See, for example, H. Schorr and W.M. Waite "An Efficient Machine-Independent Procedure for Garbage Collection in Various List Structures", Comm. ACM 10, 8(1967)501, and the references given there.

3. Suggested by A.E. Gromme (private communication).

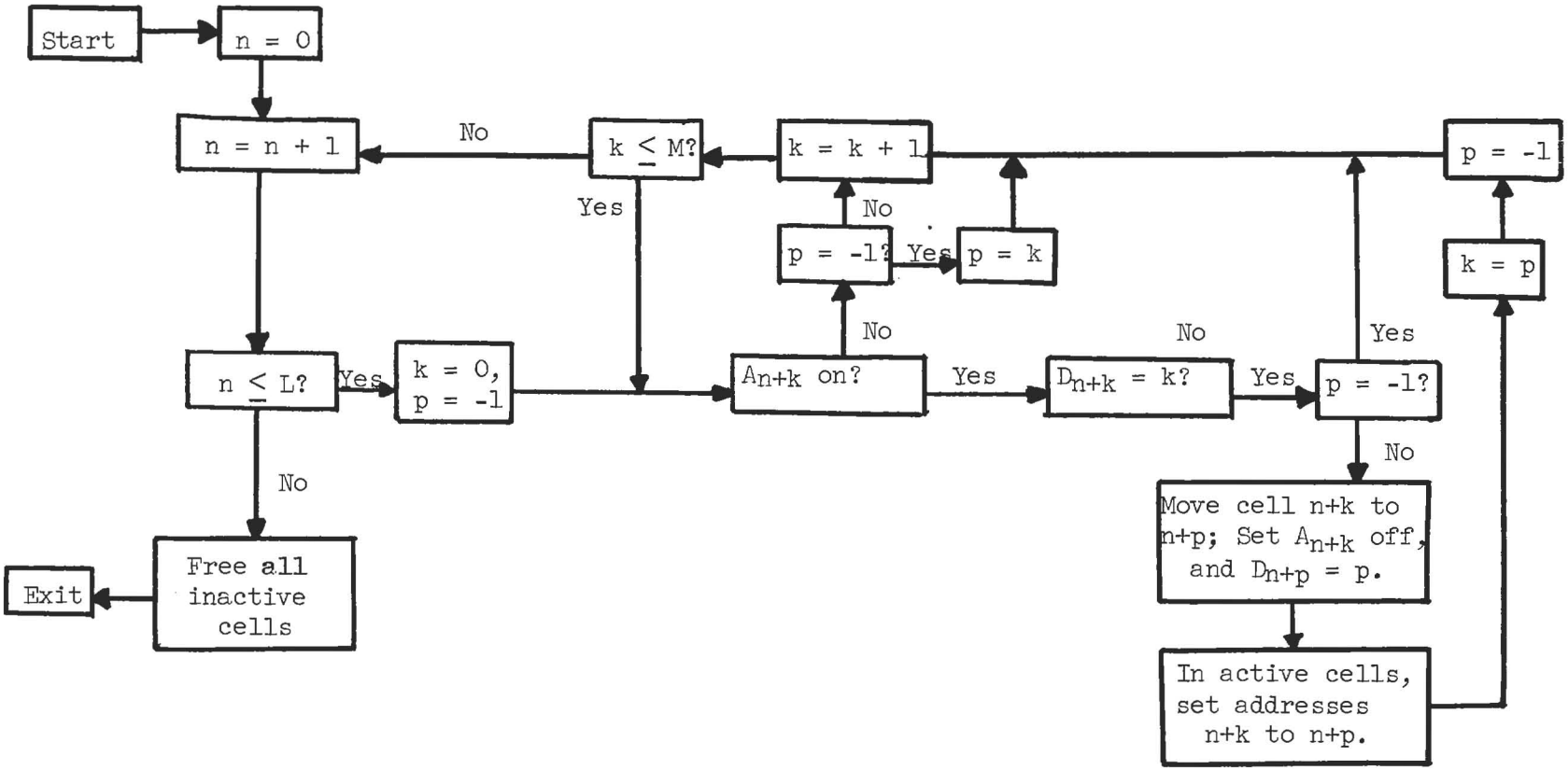

Figure 1. Garbage Collector for Method I.

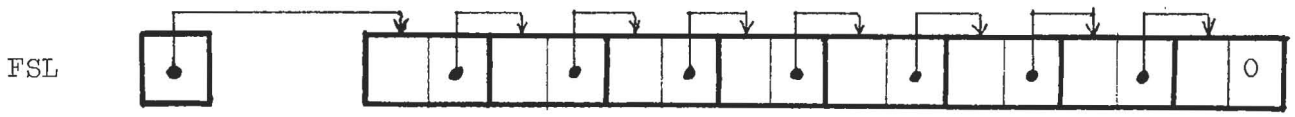

Figure 2a: Initial Configuration of Directory and Cell Storage Area; FSL points to the head of the Free Storage List.

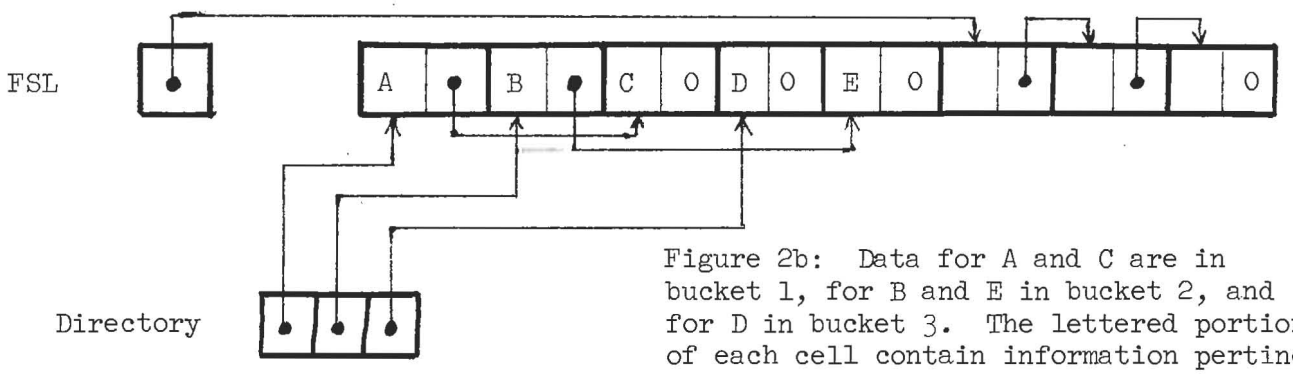

Figure 2b: Data for A and C are in bucket 1, for B and E in bucket 2, and for D in bucket 3. The lettered portions of each cell contain information pertinent to the type of list processing.

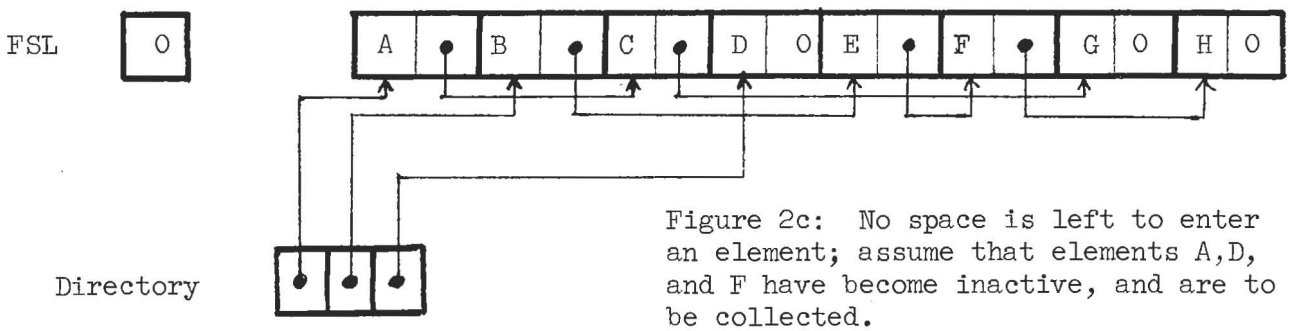

Figure 2c: No space is left to enter an element; assume that elements A,D, and F have become inactive, and are to be collected.

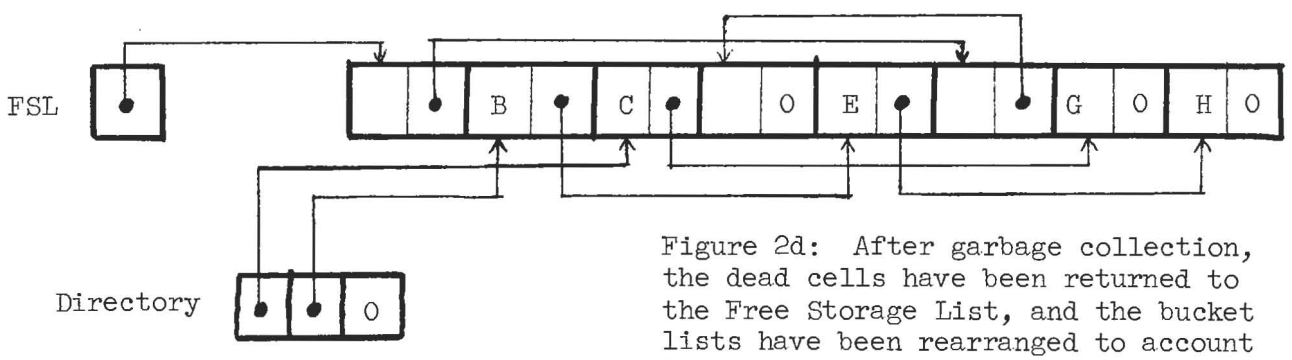

Figure 2d: After garbage collection, the dead cells have been returned to the Free Storage List, and the bucket lists have been rearranged to account for the deletions.