3-1-1983

# Garbage Collection of Linked Data Structures: An Example in a Network Oriented Database Management System

Delphine T H Kung

Rochester Institute of Technology
School of Computer Science and Technology

# GARBAGE COLLECTION OF LINKED DATA STRUCTURES: AN EXAMPLE IN A NETWORK ORIENTED DATABASE MANAGEMENT SYSTEM

by

DELPHINE T.H. KUNG

A thesis submitted to
The Faculty of the School of Computer Science and Techanology
in partial fulfillment of the requirements for the
degree of Master of Science in Computer Science

MARCH, 1983

Approved by:

| | | |
|---|---|---|
| Thesis Advisor | Mrs. Mary Ann Dvonch | 5/11/84 Date |
| Committee Member | DOCTOR Peter H. Lutz | 5/7/84 Date |
| Committee Member | DOCTOR Rayno D. Niemi | 5/7/84 Date |

G-929169

Statement for granting or denying permission to
reproduce an RIT thesis

Title of Thesis "Garbage collection of linked data structure:
an example in a network orierted database management system"

I Delphine Kung hereby deny permission to the Wallace Memorial
Library, of R.I.T., to reproduce my thesis in whole or in part.
Any reproduction will not be for commercial use or profit.

Signature: _____ May 3, 1984
date

ACKNOWLEDGEMENT

I would like to thank my thesis adviser, Mary Ann Dvonch, for her conducit guidence and discussion in directing my effort on this project. I would like to express my thanks to Dr. Rayno Niemi and Dr. Peter Lutz for the ideas they stimulated and the time they have generously spent during the evolution of this project. Lastly, I would like to place the most gratitute to my parents for their encouragement and support letting me concentrate on my study and research in accomplishing this work.

ABSTRACT

A unified view of the numerous existing algorithms for performing garbage collection of linked data structure has been presented. An implementation of a garbage collection tool in a network oriented database management system has been described

Table of Content

# 1. INTRODUCTION

Computer programs and languages having a need for the dynamic creation of data objects must return the data objects' space that is no longer used to a pool of free storage. Hence the objects' unused storage may be reused in the future for the creation of new data objects. Garbage collection is the method that has been traditionally used to do this bookkeeping. The process of garbage collection is one of reclaiming unused storage space which can be accomplished by various algorithms. Experience with large LISP programs indicates that substantial excution time - 10 to 30 percent - is spent in garabge collection [STEELE 75, WADLER 76]. Hence the efficiency of programs depends directly on the availability of fast methods for garbage collection.

Garbage collection has become important. Every data structure book has devoted sections to discuss this topic [KNUTH 73, HOROWITZ & SAHNI 76. PFALTZ 77, GOTLIEB 78, AUGENSTEIN 79, STANDISH 80], Knuth's book, section 2.3.5, is the most comprehensive. It contains detailed descriptions and analyses of some of the garbage collection algorithms that appeared prior to 1968, and, despite its age, it remains a standard reference for algorithms. Although numerous papers have been published since the late sixties, very little has been written to summarize and

classify current work. The purpose of this thesis is to investigate a unified view of the many existing algorithms for performing garbage collection of linked data structures. This thesis is divided into two parts: research and implementation. The objectives of the research portions are:

(1) to review the classical algorithms for collecting linked data structures, and to discuss their marking and collecting phases, with and without compacting;

(2) to provide a unified description of recent garbage collection algorithms and to explain how they relate to the classical algorithms;

(3) to survey the related topics of parallel and real-time garbage collections, the use of reference counters, the use of secondary and virtual storage, and language features which influence the design of collectors.

(4) to present a comprehensive bibliography on the subject for the interested reader.

Although storage allocation and garbage collection are interrelated, the emphasis of this research is on garbage collection proper, that is, on reclaiming storage. Buddy systems [KNUTH 73] and related work are not covered.

The implementation portion of the thesis explores the use of a gartage collection tool in a network oriented Database Management System [DBMS] which uses three major data structures: B-trees, binary trees and doubly linked list.

## 2. THEORETICAL AND CONCEPTUAL DEVELOPMENT

A cell is a number of contiguous computer words that can be made available to a user. Cells are requested by the user's program from the storage allocator. Since the number of available cells is finite, a time may come when no single cell is available.

Experience indicates that some of the cells are inaccessible and unused but still occupy space in the memory. A cell becomes "garbage", when it can no longer be accessed through the pointers of any reachable cell. It is the garbage collector's task to reclaim this wasted storage space, and return it to the allocator.

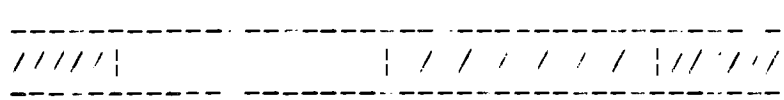
Typically, a garbage collector is invoked automatically when the free storage list is near exhaustion. Users never know, unless they become suspicious because the computer seems to drop dead suddenly, but temporarily. This is a static garbage collection. At this time, processing is stopped and the system takes over, going through an exhaustive and often time-consuming process of determining exactly what list nodes are still in use, and which ones can be returned to free storage. All these free nodes are returned, and then user processing resumes with a replenished supply of free storage [see KNUTH 73]. However, some DBMS such as SEED [see SEED 81] are equipped

with a tool that has to be called to identify the percentage of uncovered freespace. The Data Base Administrator [DBA] uses this utility to monitor areas that are recognized as highly volatile to avoid last minute scrambles for space. The DBA then calls the garbage collector to reclaim freespace if the percentage of uncovered freespace is high enough to warrant garbage collection. (SEED is a DBMS based on the DBMS specification given in the April 1971 CODASYL DBTG report.)

Garbage collection is usually called automatically either when the allocator runs out of space or shortly before. Higher level languages often contain primitives for requesting groups of words from the allocator. The garbage collector may be turned on when one of these primitives is excuted. For example, in LISP, some newer garbage collectors do not use the mark-and-sweep idea. Instead they do a little garbage collection with each "cons". They never stop for a complete collect. The function "cons" also calls the garbage collector [see WINSTON 81, pp.106-110].

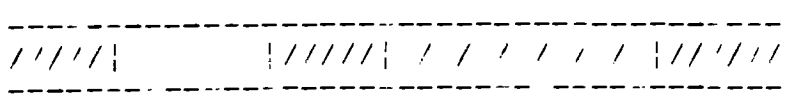
The difficulty with the static garbage-collection technique is that it requires an execution time proportional to the number of nodes in use. It tends to be slow when the memory is nearly full. And in such cases the number of free storage cells found by the reclamation

process is not worth the effort. This unattractive performance under pressure is a major drawback of garbage collection [KNUTE 73]. A further problem is that an allocation may be made from the middle of a physically contiguous area of storage (from the end of the first of two contiguous blocks, the second of which is not yet known to be free). Thus unnecessary fragmentation can result. In diagram (1), the heavily shaded areas at the extreme left and right are unavailable. Suppose that there are 2 adjacent area of memory, both of which are available, but because of the garbage-collection philosophy, one of them (shown shaded) is not in the available space list.



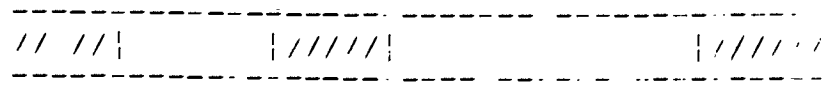
(1)

We may now reserve the areas known to be available (shown on diagram (2)):



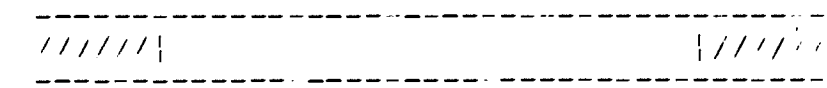
(2)

If garbage collection occurs at this point, we have two seperate free areas (shown on diagram (3)).



(3)

And as time goes on the situation gets worse. This phenomenon causes the static garbage collection technique to leave memory more broken up than it should be. If we had used a philosophy of returning blocks to the available list as soon as they become free (real-time dynamic garbage collectio), and collapsing adjacent available areas together, we have collapsed diagram (1) into diagram (4).



(4)

and we would reserve the areas known to be availble on diagram (5).

(5)

In the case discussed above, the situation portrayed by Diagram (5) is much better than the situation protrayed by Diagram (3).

To remove this difficulty of storage fragmentation, we use marking together with the process of compacting memory, i.e., moving all the reserved blocks into the consecutive location; so that all available blocks will come together whenever garbage collection is done. The allocation now becomes completely trivial, since there is only one available block. Even though this technique takes time to recopy all the location that are in use, and to change the value of the link fields therein, it can be applied efficiently with a disciplined use of pointers. However, garbage collection is unsuitable for real-time applications, because even if the garbage collector goes into action infrequently, it requires large amounts of computer time on these occasions.

After my investigation into the matter, I have chosen to catagorize garbage collection into two seperate phases as follows:

(1) the identification of the storage space that may be reclaimed.

(2) the collection of this reclaimed space and its return to the memory area available to the user.

Two principal methods have been suggested for phase one:

(a) keep counters that indicate the number of times cells

have been referenced, and identify the inaccessible cell count as zero. This identification method is called the reference counter.

(b) keep a list of accessible cells and trace their links and mark every accessible cell. This identification method is usually called marking.

Phase 2 generally can be classified into 2 approaches:

(a) incorporate all available cells that are linked by pointers into a free list.

(b) compact all used cells in one end of memory. All freed cells are made available to the allocator in the other end.

There are various ways to do the compaction according to the relative positions of the cells left after compaction. It can be classified into the following three types:

(b-1) Arbitrary: Cells which originally point to one another do not necessarily occupy contiguous positions in the memory after compaction.

(b-2) Linearizing: Cells which originally point to one another become adjacent in the memory after compaction.

(b-3) Sliding: Cells are moved toward one end of the storage space without changing their linear order.

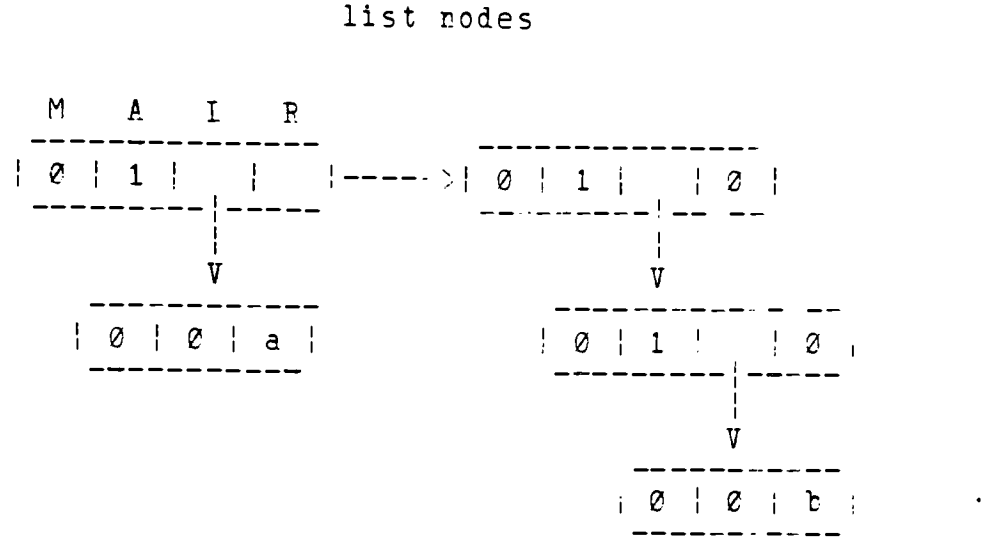
It is also convenient to classify garbage collection according to the type of cells that are reclaimed. The early methods were applicable only to programs in which all the cells were of the same sizes. With the introduction of records (or similar structures) into programming languages, it became important to do garbage collection in programs involving cells of different sizes.

## 2.1. Collecting Single-sized Cells

### 2.1.1. Marking

LISP cells illustrate the problems involved in marking single-size cells. Each LISP cell has two fields: the left(or car) and the right (or cdr). These fields contain pointers either to other cells or to atoms (special kinds of cells containing no pointers). Each cell also contains two Boolean fields: one to help differentiate between atomic and nonatomic cells, and the other to be used in marking. With this node structure, the list(a,(b)) is represented in Figure 2-1 [referenced from HOROWITZ 76]:

The algorithm shown in Figure 2-2 is a recursive procedure generally used for marking LISP lists (including




M: Mark bit  M = Ø (unmarked)  M = 1 (marked)

A: Atom bit  A = Ø (atomic node)  A = 1 (list node)

I: Left pointer

R: Right pointer

Figure 2-1

atoms). It uses three auxiliary procedures:

nonatomic(p): Boolean function that tests whether the cell pointed to by p is nonatomic;

unmarked(p): Boolean function that tests whether the cell pointed to by p is unmarked;

markcode(p): Procedure that marks the cell pointed to by p that turned on its marking bit (marking bits are initially

turned off).

Note the similarity of the marking algorithm in Figure 2-2 with the classical preorder tree-traversal algorithm [see KNUTH 73]. The one in Figure 2-2, however, can handle general lists, including circular ones.

The following is perhaps the simplest and oldest marking algorithm in use. A 1-bit field MARK (initially Ø) for marking is associated with each node. A stack S (initially empty) is used by the algorithm to hold references to sublists in use but not yet marked. The algorithm marks an unmarked list by marking each element in turn, then marking each unmarked sublist of the list. This algorithm is quite simple and straight forward [see KNUTH 73]. A nonrecursive version of the algorithm in

```
procedure mark(p)    /* p is pointer that is called
                        by value */
begin
     if unmarked(p)
        then
            marknode(p)
            if nonatomic(p)
               then
                   mark(left(p));
                   mark(right(p))
end.
```

Figure 2-2

Figure 2-2 could be also designed with this stack technique. It has, however, one major drawback. As I have indicated, garbage collection is being performed because storage is essentially exhausted, and yet the collector requires, for its execution, storage for a stack of indeterminate size. For example, if the storage area consists of n LISP cells, the maximum depth required for the stack is then n. To reserve this much additional storage initiallly is uneconomical.

A fixed-size stack can of course be reserved for the routine, with the space taken away from that available for list storage. But if that stack is exhausted during garbage collection, there is no recourse. Partial reclamation does not work because nodes not yet processed are all apparently not in use - their MARK bits are 0. This strategy fails if the stack overflows. Several algorithms have been proposed to eliminate this difficulty; all of them involve reducing the required storage by trading it for longer time needed in which to do the marking. Many interesting algorithms have been defined to effect marking [referenced from KNUTH'73, pp. 414-419].

The first of these algorithms (see Algorithm C in KNUTH 73, p.415) uses a stack of fixed length h, where h is smaller than n. (Suppose that the storage area consists of n LISP cells.) The pointers to the cells being

marked are stacked using mod h as the stack index. In other words, the stack can be thought of as being "circular", and when its index exceeds h, the additional information is written over previously stored information. The stack therefore only "recalls" the most recently stored h items and "forgets" the other ones. First, the immediately accessible cells (i.e., the cells pointed to by certain fixed locations in the main program which are used as a source for all memory accesses.) are marked. Marking then proceeds as in the algorithm in Figure 2-2. However, since some cells that should have been "remembered" have been "forgotten", the stack will become empty before the task is complete. When this happens, the memory is scanned from the lowest address, looking for any marked cell whose contents point to unmarked cells. If such a cell is found, marking resumes as before and continues until the stack becomes empty again. Eventually, a scan will find no marked cells referring to unmarked cells, and marking is complete. Actually, the scanning need not start from the beginning of the memory each time. The next scan may begin just after the recorded last address of the previous scan.

An elegant alternative to the use of a seperate stack is the temporary use of the link fields of the lists themselves, to contain the recovery information held in the

stack. These fields are modified during the marking process to allow the list traversals, then reset to their proper values as the marking of each list is completed. This strategy is known as back-tracing which is an elegant marking method developed independently by Peter Deutsch and by Herbert Schorr and W.M. Waite in 1965. As the algorithm moves to the right and down through sublists, the link fields are temporarily set in such a way that the algorithm can later retrace its steps back along the list. This garbage collection method is very generally applicable even though it is not fast. Its execution time is still proportional to the number of nodes marked. This technique requires one additional bit per cell. The additional bit per cell (called a tag bit) indicates the direction in which the restoration of reversed links should proceed (i.e., whether to follow the left or the right pointer). Knuth suggests a method for avoiding using a tag bit by instead using the bit (atom bit) already necessary for testing whether a cell is atomic. [HOROWITZ 76] contains clear descriptions of this method.

Veillon [see VEILLON 76] has shown that it is possible to transform the classical recursive algorithm in Figure 2-2 into the Deutsch/Schorr-Waite algorithm. First, the parameter of the recursive procedure "mark" is eliminated by introducing the link reversal feature. The two

recursive calls of the resulting parameterless procedure, needed to mark the left and right fields, are eliminated by introducing the tag bits to differentiate between the returns from the two calls.

Knuth proves by induction (from Algorithm A to Algorithm E) the correctness of the link-reversal marking algorithm of Deutsch/Schorr-Waite. An alterrate proof may be obtained by noting that the transformations suggested by Veillon preserve correctness. Wegbreit [WEGBREIT 72] proposes a change of the Deutsch/Schorr-Waite algorithm that uses a bit stack instead of a tag bit per cell. In the light of Veillon's program transformation, one sees that Wegbreit's stack simply implements the returns from the parameterless recursive procedure derived from Figure 2-2.

Schorr and Waite then proposed using a hybrid algorithm which combines a fixed-size stack with their link-reversal technique. It consists of using the stack algorithm whenever possible. If stack overflow occurs, the tracing and marking proceed by the method of link reversal [see KNUTH 73, p. 592].

We have seen that, at most, three bits per cell are necessary to perform LISP's garbage collection. The first two are used in recognizing atoms and in marking; the

third one is used as a tag bit, if needed by the algorithm. It should be pointed out that these three bits need not be located within or near their corresponding cells. special areas of the memory (bit maps or tables) may be allocated for this purpose. Whether or not this should be done is machine dependent.

The algorithms described in this section can be generalized to cover cells of a single-size m, with m > 2. The generalized version of the algorithm in Figure 2-2 would involve recursively marking each of the m fields of the cell.

### 2.1.2. Reclaiming Marked Cells

The simplest method for reclaiming the marked cells consists of linearly sweeping the entire memory. After turning off their mark bit, unmarked cells are incorporated into the free list administered by the storage allocator.

If compacting is preferred, it can be performed by scanning the memory twice. In the first scan, two pointers are used, one starting at the bottom of the memory (higher address), the other at the top. The top one is incremented until it points to an unmarked cell;

the bottom pointer is then decremented until it points to a marked cell. The contents of the marked cell are thereupon moved to the unmarked cell. A pointer to the new cell is placed in the old cell so that the new cell can be found during the second scan. The mark bit is also turned off during this scan. By the time the two pointers meet, all marked cells have been compacted in the upper part (lower address) of the memory.

. The second scan is needed for readjusting the pointers since some cells have been moved. It is essential to update any pointers to obsolete cell locations. This scan sweeps only the compacted area. Since we placed the pointer to the new address in the obsolete cell. Pointers to the obselete cell locations are readjusted whenever they point to cells whose contents have been moved from the liberated area to the compacted area of the memory. In other words, each of these pointers is replaced by the new pointer. According to Knuth, this method was first proposed by D. Edwards [KNUTH 73, pp. 421]. LISP and ALGOL 60 programs describing in detail this method of compacting have appeared in [COHEN 67]. Note that the two-pointer compactor is of the arbitrary type; after compaction, cells that originally point to one another do not necessarily occupy contiguous positions of the memory.

## 2.1.3. Moving Collectors

An obvious algorithm for garbge collection would be to output all useful (i.e., reachable) data to the secondary storage area and then to read them back to the main memory. This, however, has several drawbacks [see MINSKY 63]:

(1) It may require additional storage equally as large as the main memory.

(2) The time overhead for transferring between memories is usually considerable.

(3) Unless special precautions are taken, shared cells would be output more than once, in which case the main memory may not be sufficiently large for reading back the information (this situation becomes critical when the main memory contains loops of pointers).

Minsky [MINSKY 63] proposes an algorithm that eliminates the difficulties described in (3). His algorithm does not use a stack, but requires one marking bit per LISP cell. Each unmarked cell is traced and marked. Triplets (the new address of a cell and the content of its left and right fields) are computed and output to the secondary storage. The new address is also placed in the marked cell, and whenever a pointer to that cell is

encountered, the pointer is adjusted to reflect the move. When the triplets are subsequently read back into the main memory, the contents of the fields are stored in the specified new address. Minsky's algorithm has the advantage of compacting the useful information into one area of the main memory. After compaction, list elements that are linked are positioned next to each other, making Minsky's algorithm a linearizing compactor. These two properties are very important when virtual memory is used, as will be discussed in Chapter 3.

In Minsky's algorithm, fields of the original list are used to store information about the output list which is in the secondary storage. Consequently, the original list is destroyed. In this respect, it is convenient to distinguish between the terms moving and copying. The former implies a possible destruction of the original structure, whereas the latter does not. Minsky's algorithm can be used to move lists in contexts other than garbage collection. Since its appearance, several other algorithms have been proposed to do moving or copying. They can be used for garbage collection purposes as well. Most are designed to move or to copy lists without resorting to mark bits or to a stack. As in Minsky's algorithm, (1) a forwarding address is usually left in the old cell, and pointers referring to that cell are readjusted accord-

ingly, and (2) the moved lists are compacted in contiguous positions of the memory.

A few algorithms have been proposed for coping lists without using a stack or mark bits. They differ from the moving algorithms in which the altered contents of old lists are later restored to their original values. Clark [CLARK 75, CLARK 78] and Fisher [FISHER 75] discuss this kind of coping of trees and general lists.

Fenichel and Yochelson [FENICHEL 69] suggest a variant of Minsky's collector which uses an implicit stack but does not require mark bits. They divide the available memory into two areas called semispaces. At a given time, only one area is used by the allocator. When its space is exhausted, the reachable lists are moved to the other space in a linearized compacted form. The algorithm is intended for use in a paging environment.

Cheney's algorithm [CHENEY 70, WAIDEN 72], Reingold's algorithm [REINGOLD 73], and Clark's algorithm [CLARK 76, GOTLIEB 78] all represent improvements over the previous algorithm: they require neither a stack nor mark bits. Cheney's algorithm is done by moving the list to a contiguous area. Reingold's algorithm is achieved by using the Deutsch/Schorr-Waite link-reversal technique mentioned in Section 2.1.1.. And Clark's algorithm moves a list

into a contiguous area of the memory with the stack implicit in the list being moved. Clark shows that his algorithm is in most cases more efficient than both Cheney's and Reingold's.

## 2.2. Collecting Varisized Cells

### 2.2.1. Marking

As I have indicated, garbge collection in programs involving varisized cells became important with the introduction of records (or similar structures) into programming languages. Figure 2-3 shows a marking algorithm similar to that of Figure 2-2, but applicable to varisized cells. Two additional auxiliary procedures are used:

number(p): an integer function yielding the number of contiguous words (items) in the cell to which p points (this information is stored in the cell itself); and

field(p,i): a function yielding the th item of the cell pointed to by p.

It is assumed that p always points to the first item of the cell. The algorithm in Figure 2-3, like that in Figure 2-2, requires stack storage space when none may be available. If the memory contains n cells of various

```
procedure mark(p)     * p is a pointer that is called
                        by value */

integer i;

begin
     if unmarked(p)
        then
            marknode(p);
            if nonatomic(p)
               then
                  for i=1 until number(p) do
                      mark(field(p,i))
end.
```

Figure 2-3

sizes, the maximum depth required for the stack is n. Two additional fields are reserved per cell for distributing stack storage among the cells. Essentially, these fields contain the quantities p and i needed to implement the recursive calls of the procedure in Figure 2-3. A description of an algorithm of this kind appears in [THORELLI 72].

The marking algorithms of Section 2.1.1. which use a fixed-length stack can also be adapted to process variisized cells. They may then use a fixed-length stack of height h with "stack index = mod h" as before, but each stack position will contain information corresponding to p and i in the algorithm of Figure 2-3.

Variants of the Deutsch/Schorr-Waite link-reversal algorithm applicable to varisized cells are described in [MARSHALL 71, WOODCN 71, THORELLI 72, THORELLI 76 and HANSON 77].

## 2.2.2. Reclaiming Marked Cells

The method of compacting described in Section 2.1.2 is not applicable to varisized cells, since marked and unmarked cells cannot be swapped if they are of different sizes.

A possible storge configuration made up of varisized cells is shown in Figure 2-4 [referenced from WAITE 73, pp. 178-183] The 'title" is used to reserve the first word of every cell, it contains the length of cell and its own address [detail in WAITE 73, pp. 178-183]. To move all cells to a compact block at the bottom of memory. several algorithms have been proposed. One of the earliest is that of Haddon and Waite [HADDON 67, WAITE 73]. This compactor is of the sliding type and performs two scans of the entire memory. The objective of the first scan is to perform the compaction and to build a "break table" [see Figure 2-5], which the second scan uses to readjust the pointers.

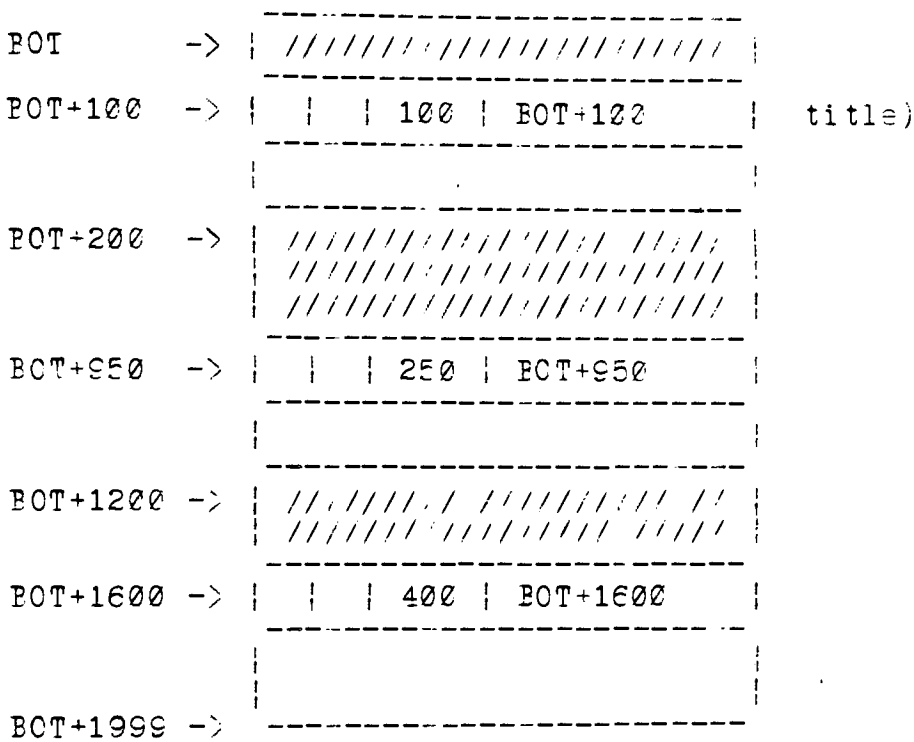



Figure 2-4: A possible configuration of a 2000-word store (The hatched areas represent free storage)

This algorithm updates all pointers after the elements have been moved. As the elements are moved to a compact group at the lower end of the storage area, a table is built for use in relocation. For each set of consecutive blocks which are not free, this break table has one entry giving the address of the first word of the set and the number of free words which were found below the set in the uncompacted store. At the end of the first

scan, the break table occupies the liberated part of the memory. It is then sorted on the address value to speed up the pointer readjutment done by the second scan. Readjustment consists of examining each pointer, consulting the table (using binary search) to determine the new position of the cell it used to point to, and changing the pointer accordingly [HADDON 67].

Figure 2-5 shows the break table corresponding to the storage of Figure 2-4. After all blocks have been moved, the relocatable address are updated by looking them up in the break table and subtracting the number of free words associated with that particular address. If an 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 table entry is unchanged. Using Figure 2-5, for example, the relocatable address POT+950 would be altered to contain POT+100 (= POT+950-850) and POT+990 would become POT+140 (= POT+990-850). This procedure may require more processing time (depending upon the layout of the free blocks in memory). An interesting feature of Haddon and Waite's algorithm is that no additional storage is needed to build the break table since it can be proved that the space available in the free sotrage suffices to store the table.

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

Figure 2-5: The break table for Figure 2-4

Other compacting algorithms for varisized cells have been proposed. The IISP 2 garbage collection algorithm described in [KNUTH 73, p. 602-602], and those presented in [WEGBREIT 72] and [TECREILI 76] have the following features in common.

Three linear scans are used. In the first scan the holes (inaccessible cells) are linked to form a free list. Two fields are reserved in each hole to store its size and a pointer to the next hole. (A subsequent scan may combine adjacent holes into a single larger hole.) The second scan consists of recognizing pointers and using the information contained in the free list to adjust them. Once the pointers have been readjusted, a third scan takes care of moving the accessible cells to the compacted area. This compactor is therefore of the sliding type.

The second scan, which readjusts pointers, is the most time consuming of the three scans. Wegbreit [WEGBREIT 72] proposes variants of this algorithm which make this scan more efficient. One variant consists of constructing a break table (called directory) which summarizes the information contained in the free list of holes. However, storage for the directory may be unavailable. Wegbreit suggests trying to use the largest hole for this purpose. When this is possible, binary search can speed up pointer readjustment. Many interesting variants for collecting varisized cells are described in [LANG 72, KNUTH 73, ZAVE 75, TERASHIMA 78, MORRIS 78 and MORRIS 79].

Marking, pointer readjustment, and compacting can be made simpler if the list processing "preserves address ordering". This means that nodes are allocated sequentially and placed into a contiguous section of memory. Under these conditions, marking can be performed in a single scan through the entire memory without using a stack. This scan also finds the number of active cells, which the second scan then uses for readjusting the pointers. The third and final scan performs the compaction which is of the sliding type. This method is applicable to fixed or varable size cells. Details are given in [FISHER 74].

## 2.2.3. Moving Collectors

Some of the moving algorithms mentioned in Section 2.1.3. may be adapted to handle varisized cells. A representative of this class of algorithms is described in the section 3.1 since it is particularly suitable for operation in virtual memory. An algorithm for copying varisized cells is described in [STANDISH 80]. It requires cells to have an additional field large enough to store an address. A first pass consists of linking all used cells via the additional field [see THOFELD 72]. A second pass copies each cell $c_i$ in the linkage and inserts the copy, $c_i'$, as the successor of $c_i$. The successor of $c_i'$ becomes the cell $c_{i+1}$. After this copying, the odd-numbered elements of the new linkage contain the original cells and the even-numbered ones contain the copies. Finally, a third pass readjusts the pointers in each copied element and separates the copy from the original. Fisher [FISHER 75] and Robson [ROBSON 77] also describe algorithms for copying LISP cells that can be generalized for copying varisized cells.

# 3. SPECIAL TOPICS ON GARBAGE COLLECTION

## 3.1. Collecting in Virtual Memory

The ratio of the size of secondary memory to the size of main memory is an important factor in designing collectors that operate in virtual memory. When this ratio is small, some of the algorithms described in the previous sections may be used. The methods described in this section are suitable when the ratio is large.

The use of secondary storage through paging [COHEN 67] changes the design considerations for implementing garbage collection algorithms in important ways. First, it is no longer necessary to try to avoid using additional storage for a stack, since the size of the available virtual memory in current systems is considerable. Avoiding page faults and thrashing (caused by having structures whose cells are scattered in many pages) becomes a critical factor in improving the efficiency of garbage collection. Compaction is for this reason important when collecting in this environment. Cohen and Trilling show that garbage collection with compaction brings about significant time gains in performance of LISP programs. They also find that a direct transcription of the classical garbage collection algorithms to a virtual memory

environment can lead to unbearably slow collection times. [CLARK 79] contains additional useful information about the performance of compacting collectors operating in virtual memory. Compaction of cells in virtual memory should not only eliminate unused holes but should also construct the compacted area so that pointers refer, if possible, to neighboring cells. As mentioned in Section 2.1.3., Minsky's algorithm [see MINSKY 63] satisfies this requirement.

Measurements in actual LISP programs show that about 97 percent of list cells have just one reference to them [CLARK 77]. This property is important when designing garbage collection algorithms which operate in virtual memory.

Bobrow and Murphy [BOBROW 67] show that the use of a selective "cons" can improve the efficiency of subsequent processing and garbage collection. Basically, they advocate keeping one free-list per page. A new cell requested by a call of cons[x,y] is taken from the free area of a page according to the following strategy:

(1) If possible, take from the page containing the cell pointed to by y; otherwise,

(2) take from the page containing the cell pointed to by x; otherwise,

(3) take from the page containing the most recently created cell; otherwise,

(4) take from any page containing a fair number (say, 16 of free cells.

The purpose is to minimize page faults in manipulating linked lists. Additional information on garbage collection using virtual memory can be found in [ECERCA 67,68].

An important design consideration for implementing garbage collection algorithms in a paging environment is deciding when collection should be invoked. Since very large memories are currently available, it seems reasonable to collect whenever page faults render the program processing unbearably slow.

A class of algorithms suitable for use in virtual memory is the one described by Baker [BAKER 78]. It is based on the copying collector proposed by Fenichel-Yochelson and by Cheney that was briefly described in Section 2.1.3.. What follows is a more detailed presentation of this type of algorithm. Although it is applicable in collecting varisized cells, this presentation applies only to IISP cells.

The available memory is divided into two areas called semispaces. At a given time, only one is used by the allocator. During garbage collection, the reachable lists are moved to the other space in a compacted form.

Bishop proposed an approach for designing collectors that operate in a very large virtual memory (of the order of 10K12 bits). Ever using the real-time approaches which will be discussed in Section 3.3 of this paper, it is impractical to garbage collect the entire memory at one stretch since large portions of memory may remain unchanged during program execution. Bishop suggests collecting only in parts of the address space rather than in the entire space. (A similar apprcach is used in Ross' AED system [ROSS 67].) The memory is divided into areas that can be collected independently, and a variant of the Fenichel and Yochelson collector is used. This collector increases the locality of reference, an important factor in a paging environment.

Tracing and copying are performed only within a given area. The system keeps lists of all area references, both incoming and outgoing. Incoming references are modified to point to the area's new copy; they define the immediately accessible cells from which collection starts. Before discarding the old copy of an area, its useless outgoing references are removed from the corresponding

memory. The disadvantages of this approach are:

(1) the extra space needed for the counters,

(2) the overhead required to update the counters, and

(3) the inability to reclaim general cyclic structures.

The principal drawback of the reference counter method is the fact that it does not always free all the cells which are available. It cannot work for the circular list. For example, recursive lists will never be returned to storage by the reference counter technique. Their counts will be nonzero (since they refer to themselves) even when no other list accessible to the running program points to them [detail in HOROWITZ 76, pp. 165-169].

Deutsch, Knuth, and Weizenbaum suggest combining the reference counter technique with classical garbage collection in order to overcome the problems of each [see DEUTSCH 76, KNUTH 73, WEIZENBAUM 69]. The former would be used during most of the processing time; the latter, being more expensive, would be done as a last resort. This allows the use of small reference counter (thus reducing the storage requirements) because counters that reach their maximum value remain unmodified. Classical garbage collection, called when the free list is exhausted, starts

lists of incoming references.

Bishop developed a method for maintaining the lists of areas references and indicated that this can be done automatically without incurring substantial run-time overhead. He advocates altering the virtual memory mechanism to cause traps when interarea references are stored into cells, and shows how virtual memory hardware can be constructed to perform this extra service efficiently.

## 3.2. Reference Counter

The reference counter approach toward storage reclamation provides for constant monitoring of list linkages, with lists returned to free storage as soon as they become available. This is a dynamic garbage collection.

The use of reference counters [advocated by COLLINS 60 and WEIZENBAUM 63] has recently attracted renewed interest. An extra field, called refcount, is required for each cell to indicate the number of times the cell is referenced. This field has to be updated each time a pointer to the cell is created or destroyed. When the refcount becomes equal to zero, the cell is inactive and can be collected. At least, theoretically, the refcount must be large enough to hold the number of cells in the

of a program still active, and those which are truly unreferenced and can be reclaimed. It follows from (1) and (2) that if a cell's address is not in the MRT or the ZCT, its refcount is one.

(3) The variable reference table (VRT) contains the addresses of cells referred to by program variables (including the temporary variables in the recursion stack).

Deutsch and Bobrow note that a transaction file is built up consisting of three sorts of transactions, that may affect the accessibility of data. They are (1) allocation of a new cell from free space, (2) creation of a pointer to a cell, and (3) destruction of a pointer to a cell. Instead of updating the hash tables as the transactions occur, Deutsch and Bobrow propose storing them in a sequential file. The transactions are examined at suitable time intervals and then the tables are updated.

When a new cell is allocated, its address should be placed in the ZCT. Since this is usually followed by the creation of a pointer to the newly allocated cell (that implies removal from the ZCT), the pair of transactions can be ignored.

According to the hybrid approach, when a pointer is created, it is examined before its insertion into a cell

or pointer variable. Three cases are possible:

(a) The pointer refers to a cell in the MRT. The corresponding refcount value is then increased by one if it has not already reached its maximum; otherwise, the refcount value is left unchanged.

(b) The pointer refers to a cell in the ZCT. The cell is then removed from that table, since its count becomes one.

(c) If tests (a) and (b) fail, the pointer refers to a cell having a reference count of one. It must then be placed in the MRT with a refcount of two.

When a pointer is destroyed (removed from a cell), two cases are possible:

(a) the pointer refers to a cell in the MRT. The cell's refcount value is decreased by one, except when it has reached its maximum, in which case it is left unchanged. If the new value of reference count is one, the cell is removed from the MRT.

(b) The pointer does not refer to a cell in the MRT. Its count is one by default, and should be reduced to zero. The cell is therefore entered in the ZCT.

The VRT is used when incorporating new cells into the free list. Since the stack is constantly being updated,

the VRT is only computed periodically. A cell is reclaimed when its address is listed in the ZCT but not in the VRT. The ZCT is updated by eliminating the entries of the reclaimed cells that are not pointed to by the program variables.

For the classical collection, Deutsch and Bobrow advocate using a variant of the two-semispace collector of Fenichel and Yochelson [FENICHEL 69]. They also point out that an auxiliary processor could speed up the collection. Its task would be to scan the ZCT and VRT tables to determine which cells could be incorporated into the free list.

Barth [BARTH 77] considers reference counters in relation to shifting garbage collection overhead to compile time. He shows that savings in collection time are sometimes possible by carefully studying, at compile time, the program's assignments. For example, if r <- right(r), the cell originally pointed to by r may be incorporated into the free list if it is known that it will not be refererced by other pointers.

## 3.3. Parallel and Real-Time Collections

Two proposals have been made to circumvent the time lost by garbage collecting interruptions. The first is to

allow garbage collection to proceed simultaneously with program execution by using two parallel processors: one is responsible for collection, the other for program execution. When collection actually takes place, it is bound by a known, tolerable, maximum time.

Minsky is credited by Knuth with initiating the development of algorithms for time-sharing garbage collection and list-processing tasks [KNUTH 73]. If two processors are available, these tasks can be performed in parallel, with one of these processors, the collector, responsible for actual garbage collection, and the other performing the list processing and providing the storage requested by a user's program. Dijkstra calls this latter processor the mutator. The collector performs the basic tasks of marking and incorporating unmarked cells to a free list, during which time the mutator is active. The mutator may not request cells until the collector makes them available.

France, Gries, and Muller provide detailed descriptions of Dijkstra's algorithm, but their main concern is to prove correctness [FRANCE 78, GRIES 77, MULLER 76]. An extension of Dijkstra's algorithm with multiple mutators is considered in [IAMPORT 76].

Steele has independently developed a method for parallel garbage collection based on the Minsky-Knuth suggestion. He was one of the first to propose actual algorithms for collecting in parallel. Steel's collector makes exclusive use of semaphores and requires two bits per cell, which are used not only for marking but also for compacting and for readjusting pointers. Compaction is done using the two-pointer technique.

Comparing Dijkstra's to Steele's algorithm is difficult because these authors had different objectives. The former wanted to assure the correctness of his algorithm (regardless of its efficiency), and the latter had in mind an implementation using special hardware, possibly microcoded.

Kung and Song [KUNG 77] propose a variant of Dijkstra's method that uses four colors for marking and that does not need to trace the free list. The authors prove the correctness of the algorithm and show that it is more efficient than Dijkstra's. To these authors' knowledge, none of the parallel garbage collection algorithms has been implemented, nor are any detailed results from simulation yet available.

An alternative to using two processors is to have one processor time-share the duties of the mutator and collec-

tor. Wadler [WADLER 76] shows that algorithms for performing garbage collection with time-sharing demand a greater percentage of the processing time than does classical sequential garbage collection. This is because the collection effort must proceed even when there is no demand for it.

A second approach for avoiding substantial program interruptions due to garbage collection has been proposed by Baker [see BAKER 78]. His method is an interesting modification of the collector. Baker's modification is such that each time a cell is requested a fixed number of cells, k, are moved from one semispace to the other. This implies that the two semispaces are simultaneously active. In a paging environment, the extra memory required is of less significance than the possible increase in the size of the average working set. Since the moved lists are compacted, page faults are likely to be minimized.

The moving of k cells during a "cons" corresponds to the tracing of many cells in classical garbage collection. By distributing some of the garbage collection tasks during list processing. Baker's method guarantees that actual garbage collection cannot last more than a fixed (tolerable) amount of time: the time to flip the semispaces and to readjust a fixed number of pointers declared in the user's program. Thus his algorithm may be

used in real-time applications

A characteristic of Baker's real-time algorithm is that the size of the semispaces may have to be increased, depending on the value of k and the type of list processing done by the program. In other words, the choice of k expresses the trade-off between the time to execute a cons and the total storage required. For example, for k = 1/3, a cell is moved every third time a cons is called. This would speed up the computation but increase the amount of storage required.

In his paper Baker offers an informal proof of his algorithm's correctness and shows how it can be modified to handle varisized cells and arrays of pointers. He also presents analyses of storage requirements of the algorithm and how it compares with those of other garbage collection methods. A IISP machine built at M.I.T used Baker's approach [BAWDEN 77]; its memory is subdivided into areas, and a list of outgoing references is kept for each. Those areas which do not change during program execution are not copied: tracing starts from their coresponding list of outgoing references. This approach, which has been further developed by Bishop [BISHOP 77], is a possible alternative for real-time collection. Another alternative is the use of an auxiliary processor as suggested by Deutsch and Botrow [DEUTSCH 76] in their incremental

garbage collection technique mentioned in the previous section.

## 3.4. Language Implementation

Recursion is frequently utilized in programs that manipulate linked-list structures. A stack is indispensable for executing these programs. Therefore, separate regions for the allocated cells and for the stack must coexist in the memory. It is true that stacks can be "simulated" by linked lists, so that the memory stores only list structures. However, this is both space and time consuming because an extra field is required to link together the data in the stack and more complex operations are needed for pushing and poping.. It is, therefore, easier and more space efficient to implement the stack in contiguous positions of memory.

It has become current practice to divide the available memory into two areas that are allowed to grow from opposite ends. One of these is reserved for a stack using contiguous locations. The other, called the heap, is available to the allocator for providing new cells, also from contiguous locations. With this arrangement, a simple test can be used to trigger garbage collection. When the pointer to the next free stack position meets the

pointer to the next available position in the heap, collection with compaction is invoked to retrieve space for new cells or for stacking. Therefore, the functions "push" and "new" may trigger garbage collection.

In the case of LISP programs, the function "new" corresponds to a "cons", and "push" is used internally by the compiler or interpreter. Collection can be started either by a "cons" or by a stack overflow caused by situations such as great recursion depth or reading long atoms [COHEN 72].

Since the stack is used in implementing recursion, it usually contains pointers to active, useful cells. The marking algorithms of Section 2.1.1. are used to mark not only the structures referred to by the pointer variables of a program but also those structures that are referred to by pointers on the stack. Therefore, means must be provided to recognize whether a stacked quantity is a pointer (tag bits may be used for this purpose) [Deutsch/Schorr-Waite list-traversal technique].

LISP processors sometimes allow a user to invoke the collector. This is useful when he has an idea of the most propitious time for triggering the collection. Also, the function return may be made available to the user. When the free list is used, the returned cells can be immedi-

ately incorporated into the list.

However, when compaction is required (with the heap and stack arrangement), the returned cells may not be available to the allocator until after the next collection. Another problem with the function return is that a cell may be explicitly returned even though there is still a pointer to it. This is sometimes called the dangling reference problem. Thus, care must be taken not to reuse the cell until there are no pointers to it.

Processors for languages like PL/I and PASCAL allow a user to call the function "new" and provide messages when storage is exhausted. The use of the functions "return" or "collect" is implementation dependent. This author is unaware of PASCAL run-time systems that perform fully automatic garbage collection. It is the user's responsibility to keep free lists of unused cells and to check whether a new cell may be obtained from the allocator.

Arnborg [ARNBORG 73] described the implementation of a SIMULA compiler designed to operate in a virtual memory environment. SIMULA is a language with block structure: variables declared in a block or procedure exist only when the block or procedure is activated. Although it seems at first sight that one could collect the structures referred to by pointer variables upon exiting from the block in

which they are declared, this is not the case. SIMULA also allows variables of such types as classes, arrays, and tests that may have longer life spans than their originating blocks; if these variables share linked structures with local block variables, collection cannot be done when exiting from a block.

Since Arnborg's proposed implementation operated in a paging environment, one of the objectives of the collection is to reduce the number of page faults. To perform the collection, Arnborg uses a variant of the method proposed by Fenichel and Yochelson [FENICHEL 69] described in Section 3.1.. The variant can handle varisized cells rather than only simple LISP cells.

A SNOBOL implementation proposed by Hanson [HANSON 77] uses a variation of the garbage collection techniques for collecting varisized cells described in section 2.2.. It is assumed that additional space for the stack can be requested from the operating system, although such requests should be kept to a minimum. An effort is made to reduce collection time by avoiding marking cells which are known to be used throughout the program's execution. For this purpose the heap is subdivided into two areas of consecutive locations: heap 1 and heap 2. The first contains information which is constantly active and never needs to be marked; the second may contain inactive cells

ately incorporated into the list.

However, when compaction is required (with the heap and stack arrangement), the returned cells may not be available to the allocator until after the next collection. Another problem with the function return is that a cell may be explicitly returned even though there is still a pointer to it. This is sometimes called the dangling reference problem. Thus, care must be taken not to reuse the cell until there are no pointers to it.

Processors for languages like PL/I and PASCAL allow a user to call the function "new" and provide messages when storage is exhausted. The use of the functions "return" or "collect" is implementation dependent. This author is unaware of PASCAL run-time systems that perform fully automatic garbage collection. It is the user's responsibility to keep free lists of unused cells and to check whether a new cell may be obtained from the allocator.

Arnborg [ARNBORG 73] described the implementation of a SIMULA compiler designed to operate in a virtual memory environment. SIMULA is a language with block structure: variables declared in a block or procedure exist only when the block or procedure is activated. Although it seems at first sight that one could collect the structures referred to by pointer variables upon exiting from the block in

## 3.5. Final Remarks

Tables 1-5 summarize the characteristics of the main algorithms described in the corresponding Sections above. The number of references presented in the bibliography bears witness to the importance and interest in garbage collection. In spite of this activity, many facets of garbage collection remain to be investigated. Specifically, very little comparison has been made of the relative efficiencies of many of the algorithms described above

New developments in hardware are likely to play an important role in speeding up collection. It has already been suggested that new machines should contain extra bits per word to be used for marking, tagging, or counting references. Machines with special hardware for segmentation and list processing have been constructed [FAWDEN 77] and are in experimental operation.

There has been a trend toward designing and implementing collectors for varisized cells stored in large virtual memories. No explicit guidance based on experimental evidence is yet available on how to do this collection efficiently or in real time. Two promising directions, discussed in Section 3.3., involve either using parallel processors or distributing some of the garbage

collection tasks during the actual processing. It is hoped that this will allow the collection to be done within a known, tolerable, maximum time.

Collection in very large virtual memories is another subject that will become increasingly important. The suggested approaches for these collections deserve further study [see PISHCP 77].

Table 1: Collecting single-sized cells

| Algorithm | Main references | Auxiliary storage | Mark(M) or tag(T) bit | Comments |
|---|---|---|---|---|
| **Marking single-sized cells [1] :** | | | | |
| Classical | Knuth 73 | Stack | M | Stack may be as large as n [2] |
| Bounded workspace | Knuth 73 (pp. 415) | Limited-size stack | M | Requres more time than the classical algorithm |
| Link reversal (Deutsch/ Schorr-Waite) | Schorr 67 Knuth 73 | No stack is needed | M&T | As above (the tag bit may be replaced by the atom bit |
| Link reversal with bit stack | Wegbreit 72 | Bit stack | M | Similar to the above algorithm |
| Hybrid | Knuth 73 (pp. 592) | Limited-size stack | M&T | uses a comtination of stack and link reversal |
| **Compaction of single-sized cells:** | | | | |
| Two pointer | Knuth 73 | None | None | Only applicable to single-sized cells. Compaction of arbitrary type |
| **Moving collectors of single-sized cells:** | | | | |
| Minsky | Minsky 63 | None | M | Compaction of linearizing type |
| Fenichel-Yochelson | Fenichel 69 | Stack | None | Uses two semispaces. Compactionof linearizing type |

[1] These marking algorithms may be extended to mark varisized cells.
[2] r is the total number of accessible cells

Table 2: Collecting varisized cells

| Algorithm | Main references | Auxiliary storage | Mark(M) or tag(T) bit | Comments |
|---|---|---|---|---|
| Marking varisized cells: | | | | |
| Classical | Knuth 73<br>Thorelli 72 | Stack | M | Stack can be stored using an additional field of each cell |
| link reversal | Thorelli 76 | No stack is | M&T | Variant of Deutsch/schorr-waite link-reversal technique |
| Compacting varisized cells: All these compactions are of the sliding type | | | | |
| Break table | Haddon 67 | None | None | Break table has to be sorted |
| IISP 2 | Knuth 73<br>(pp. 602) | Additional word per cell | None | Requires three or more scans |
| Moving (Copying) collectors of varisized cells: | | | | |
| Standish | Standish 80 | One field | None | Copies using three passes |

Table 3: Collecting in virtual memory

| Algorithm | Main references | Auxiliary storage | Comments |
|---|---|---|---|
| Baker | Baker 78 | None | Uses two semispaces |
| Bishop | Bishop 77 | Space for keeping interarea lists | Designed for use in very large virtual memories |

Table 4: Reference counter (m is the number of available cells

| Algorithm | Main references | Storage needed | Comments |
|---|---|---|---|
| Classical | Collin 60 | An extra field (of size m) per cell | Cannot handle general circular lists |
| Hybrid | Knuth 73<br>Deutsch 76 | An extra field (of size h < m) per cell | Combines reference counters with classical compacting garbage collection |

Table 5: Parallel and Real-Time Collection

| Algorithm | Main reference | Storage needed | Comments |
|---|---|---|---|
| Parallel (Dijkstra) | Dijkstra 78 | No stack & two bits per cell | Main objective is to prove correctness |
| Parallel (Steele) | Steele 75 | Stack, Two bits per cell and several semaphores | Designed to be microcoded. Does compacting |
| Baker | Baker 78 | Two semispaces whose sizes vary at execution time | Moing of accessible cells is done when a new cell is requested |

## 4. An Implementation Of Garbage Collector In A Network DBMS

When dealing with performance problems, what does the Database Administrator[DBA] do when jobs are aborting because the database is full or out of space? What should the DBA do when complaints start pouring in about response time being slow or jobs running behind schedule?

Space and performance problems in a volatile database are sometimes caused by a high percentage of deleted, but unremoved, records in the database [SEED 81]. The problem of unremoved, deleted records can be solved by garbage collection mechanism.

## 4.1. Problem Statement

When a delete of a record is enacted under the DBMS, the record is logically deleted from the database, that is, any reference to the deleted item will not be satisfied. However, in some DBMS, the actual physical file record is not deleted. Over time, the DBMS will cease to operate efficiently because of these undeleted records. The problems that develop when deleted records remain unremoved are a shortage of freespace, and a loss of efficiency. Hence, storage will be wasted on inaccessible and useless data. Time will be wasted since the system will be required to do increased I/O. The space allocated will carry the defunct information, and degrade performance.

## 4.2. Implementation Part Approach

In M.A. Dvonch's system [DVONCH 82], when the deletion of a set owner/member or the deletion of a record is enacted under the DBMS, the set occurrence or the record is logically deleted from the database. However, the actual physical file record is not deleted. Over time, the database will become cluttered with useless information. In answer to this problem, we plan to implement a garbage collection utility to remove the deleted records and to compact this storage.

This enhancement is based on M.A. Dvonch's DBMS in conjunction with W. Stratton's B-Tree program (the logic is original from Wirth and Niklaus B-tree algorithm [see WIRTH 76]), and is written in the C programming language under the UNIX operating system.

In the following sections, we will first of all take a look at the structure of the database files in the implemented DBMS and the functions of Database deletion that give the background of the garbage collector.

### 4.2.1. Database Files

Figure 4-1 depicts the implemented system on which this project is based [ DVONCH 82]. The Database Management system presently implemented provides (1) definition directives, which allow the creation of record and set types, and (2) manipulation commands or primitives, which allow the manipulation of the data stored in the database.

This model uses two different form types to represent the information contained in the database. A record type is the format of the record. A set type is the definition of a relationship between two record types. For a set type, one record type is the owner type and another record type is the member type.

Definition directives are accepted by the DBMS and appropriate record or set types are defined. The DBMS does not allow the user to delete a record type or set type once it has been defined. The record type and set type definition are maintained in a binary tree in memory while the DBMS is in use. When a new record or set type is defined, a node is created for it in the appropriate tree. At the conclusion of the database update, pertinent information from the trees is stored in a file named dbm_file which is the data base management file as shown in Figure 4-1. The first field in the dbm_file is the

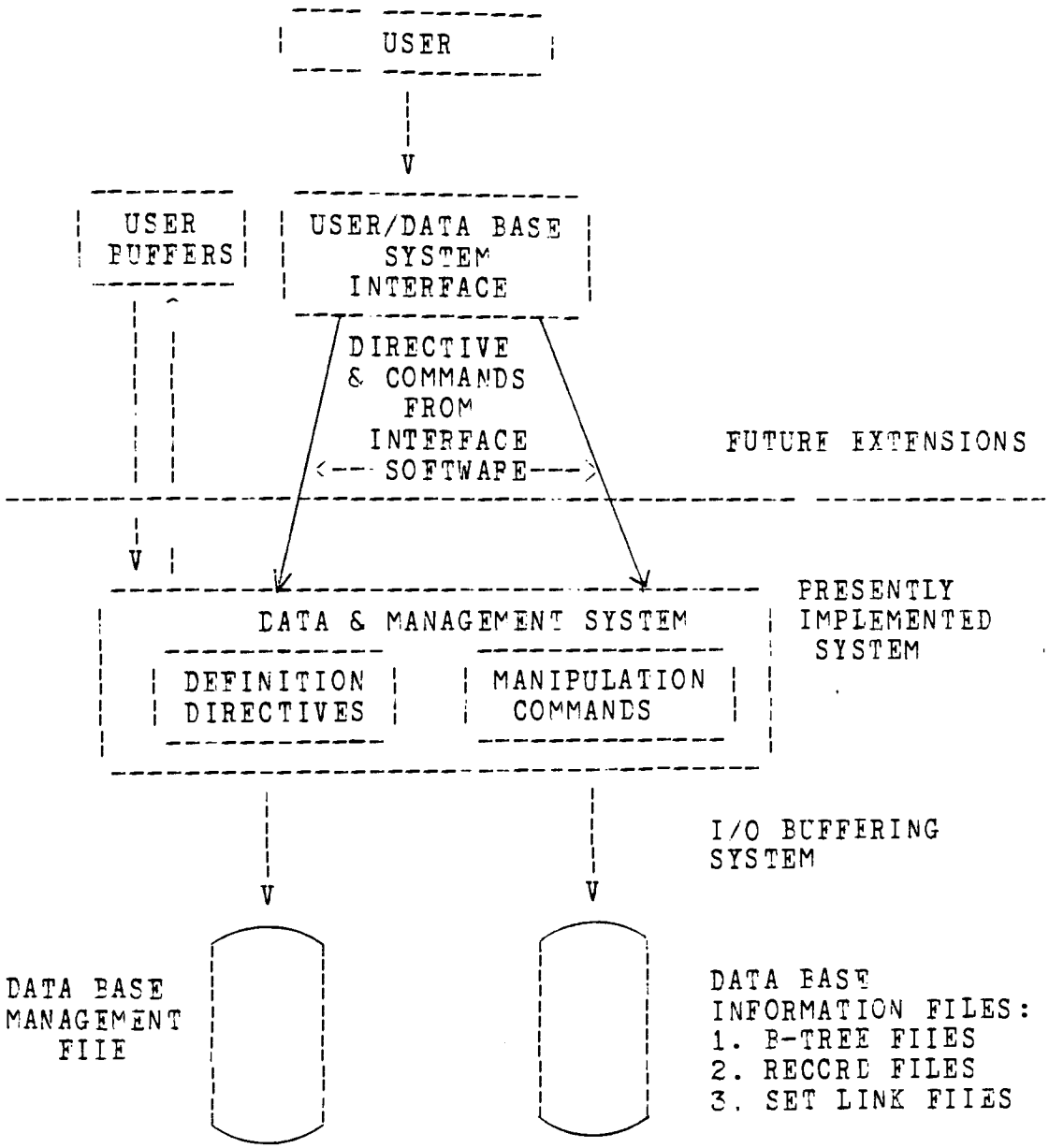



FIGURE 4-1
SYSTEM ON WHICH THE PROJECT IS BASED

number of record types defined within the database, the second is the number of set types defined. After these two fields come the record type information and then the set type information. When the database management system is run, the system builds the new binary type trees from the information stored on the dbm_file. The dbm_file is a normal UNIX file that can be examined by and used with other UNIX tools. If no record types or set types have been defined, the system will print a message informing the user of this fact.

#### 4.2.1.1. Record Types

The system creates the record type by inserting the record type name with ".rf" appended to it into the binary tree containing record type definitions. When the user then issues an "add record" command for a particular record type, the database system fills an input/output buffer with the desired record, checks the field count on the record for an error, and builds the record's key. The key is also error checked for duplicity. The record is then passed to the input/output buffering system where it is stored at the end of a file whose name is the record type name with the suffix ".rf . As the record is passed to the buffering system, the location of the record within

the file is determined. The location of a record is the block offset from the beginning of the file and then an offset to the position within the block. Once the address of the record within the file is known, the address information and the record key is stored in the record type B-tree named "dbm.rec.bt". If the record type participates as an owner in any set type, the new record is added as an owner occurrence for each such set. These records are stored in regular UNIX file. Again, this allows the stored database files to be used with any other UNIX tools. For example, a record file could be printed directly. This separation of record files from database information is one of the strengths of the system, for it is in keeping with the UNIX idea of software tools [M.A. Dvonch's thesis 82]. However, deleted records still remain on the record file until garbage collection is done.

#### 4.2.1.2. Set Types

The system creates a set type by inserting the set type name with ".sf" appended to it into the set type binary tree containing set type definition.

After the set type has been defined, owner occurrences are automatically created whenever a record of

the owner record type is added to the database. When an owner record occurrence is added to the database, a link-record for the owner is built. This link-record contains pointers with the forward pointer first and backward pointer second. The front pointer of any owner or member in a set occurence contains the key of the next owner or member in the set occurrence. The back pointer of any member contains the key of the prior member or the owner. The owner's back pointer is alwasys null. This link-record is then given to the I/O buffer system to be added to a file named by the set type name with ".sf" appended to it. As the record is passed to the I/O buffer system, information is gathered on the location of the pointer record within the set type file. This location information and the owner key are stored in the set type B-tree. The link-records are stored as regular UNIX files and, therefore, can be used as arguments for other UNIX tools.

If the user wishes to add a member to a set occurrence, he must give an explicit "add member" command. The user specifies the set type and owner key of the appropriate set occurrence. Inserting the new member into a set occurrence is accomplished by building a link-record for the new member. These link-records make up a chain that connect the set occurrences, and can be traversed to obtain all the members of a set occurrence. The new

member occurrence is added as the immediate successor to the owner. Order of retrieval is therefore LIFO (Figure 1-2). Once the link-record is stored, the file offset information along with the new member key are stored in the set type B-tree.

## 4.2.2. Database Deletion

The database management system's deletion tools are recursive. Since deleting a record implies possible deletion of set occurrences and vice versa, recursion is required. Each deletion tool was designed to do its own

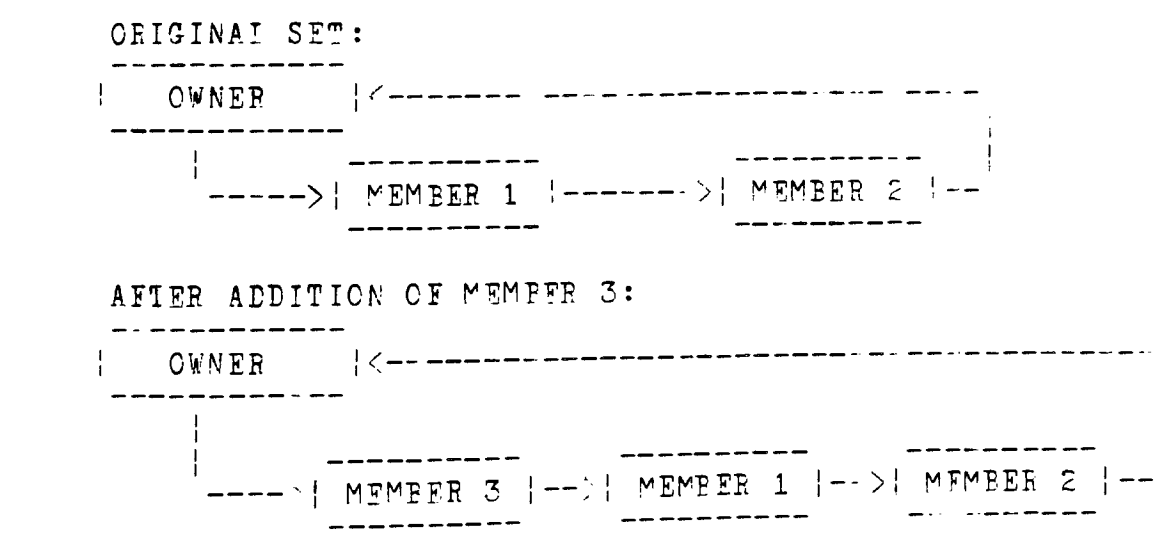



FIGURE 4-2
ADDITION OF NEW MEMBER TO SET OCCURRENCE

particular job and then call a friend to do the rest" [Dvonch 82].

#### 4.2.2.1. Delete Record

When a record is deleted from the database, the record key and file offset are removed from the record type B-tree. Since the offsets for the actual record can no longer be referenced, the record is logically deleted from the database. However, the actual physical record will still remain on the record type file. This record then becomes the province of the garbage collection system.

Once the record is logically deleted, delete record recursively searches the set type binary tree looking for sets in which this record type participates as an owner or member. If a set is found, delete owner or delete member is called to remove the discarded record key from the set.

#### 4.2.2.2. Delete Member

The delete member tool causes a member's key and link-record offset to be removed from the set type B-tree. The pointers in the link-records of prior and post set occurrence members are reset to exclude the deleted

member. The actual link-record is left on the set type file, requiring garbage collection at a later time. However, since this link-record can no longer be referenced, it is effectively deleted from the database.

#### 4.2.2.3. Delete Owner

When the delete owner command is given, the owner's key and link-record file offset are removed from the set type B-tree. Next, "delete member" is repeatedly called until all of this owner's members are deleted. Finally, "delete record" is called to delete the owner's record from the record type file. That is, when an owner is deleted from a set, each of his members is also deleted.

### 4.2.3. Prototype

Consider a simple prototype database which consists of four record types and three set types with certain relationships expressed through set occurrences (see Figure 4-3). And using the same test data files run by M.A. Dvorch's DBMS as the test data files of this garbage collector's implemention project for consistency.

Four record type descriptions and contents:

(1)

Record Type Name: faculty
Field Delimiter: *
Number of Fields: 5
Number of Key Fields: 1
Position of Key Fields: 2

Contents:

Peter*A1*10*A196*25
Bill*A2*10*2132*57
Roy*3A*10*A285*72
Jack*4A*10*1116*13

(2)

Record Type Name: student
Field Delimiter: :
Number of Fields: 4
Number of Key Fields: 1
Position of Key Fields: 3

Contents:

Mary:CAST:B1:Comp Scie
Leslie:CAST:B2:Comp Scie
John:SP:3B:PPPD
Tom:CAST:4B:Syst Soft
Mary:SP:5B:PPPD

(3)

Record Type Name: housing
Field Delimiter: *
Number of Fields: 3
Number of Key Fields: 1
Position of Key Fields: 1

Contents:

405*Billings*25
215*Watson*1105

(4)

Record Type Name: courses
Field Delimiter: *
Number of Fields: 8
Number of Key Fields: 4
Position of Key Fields: 5 1 3 4

Contents:

B1*0601*81*1*875*1*D*r
B2*0601*81*2*875*1*A*nr
B1*0601*81*1*720*2*B*nr
3B*0532*81*2*875*1*A*nr
B2*0532*81*2*859*1*B*r
B1*0601*81*2*875*1*B*r
5B*0601*80*2*875*1*D*r
5B*0601*81*3*875*1*B*r

Three set type descriptions and contents:

(1)

Set Type Name: fs
Owner Record Type: faculty
Member Record Type: student
Relationship: the faculty member is the advisor of the student

Contents:

A1 owns
B1
3B

A2 owns
B2
4B

3A owns
5B

4A has no members

(2)

Set Type Name: sc
Owner Record Type: student
Member Record Type: courses
Relationship: the student has taken the course

Contents:

B1 owns
875B1E11
720B1E11
875B1E12

B2 owns
875B2E12
850b2912

3B owns
8753BE12

4P has no members

5P owns
8755B802
8755B813

(3)

Set Type Name: hs
Owner Record Type: housing
Member Record Type: student
Relationship: the student is a resident of the housing complex

Contents:

405 owns
P1
5B

216 owns
3B

To delete student P1's record in this database, we use the delete record command ("dr student P1"). (see Figure 4-4).

The "dr student P1" command would remove the record key P1 and file offset from the record type B-tree. Then P1 would be removed from the faculty-student set (file: fs.sf) and the housing-student set (file: hs.sf) as a member (call "dm fs P1" and "dm hs P1" respectively) and the student-courses set (file: sc.sf) as an owner (call "do sc P1").

The "do sc B1" command would remove B1 from the sc set. Next, courses 875B1811, 720B1811, and 875B1812 would be removed from the sc set (call "dm sc 875B1811", "dm sc 720B1811", "dm sc 875B1812") and the courses record file (call "dr courses 875B1811","dr courses 720B1811", "dr courses 875B1812") and then P1 would be removed from the student record file.

Figure 4-4 diplcts the Database files when the DBMS is not being used after executing the delete command "dr student P1". As we can see, it needs garbage collection to reorgnize the database information files, ie record files and set link files which contains the useless records after deletion.

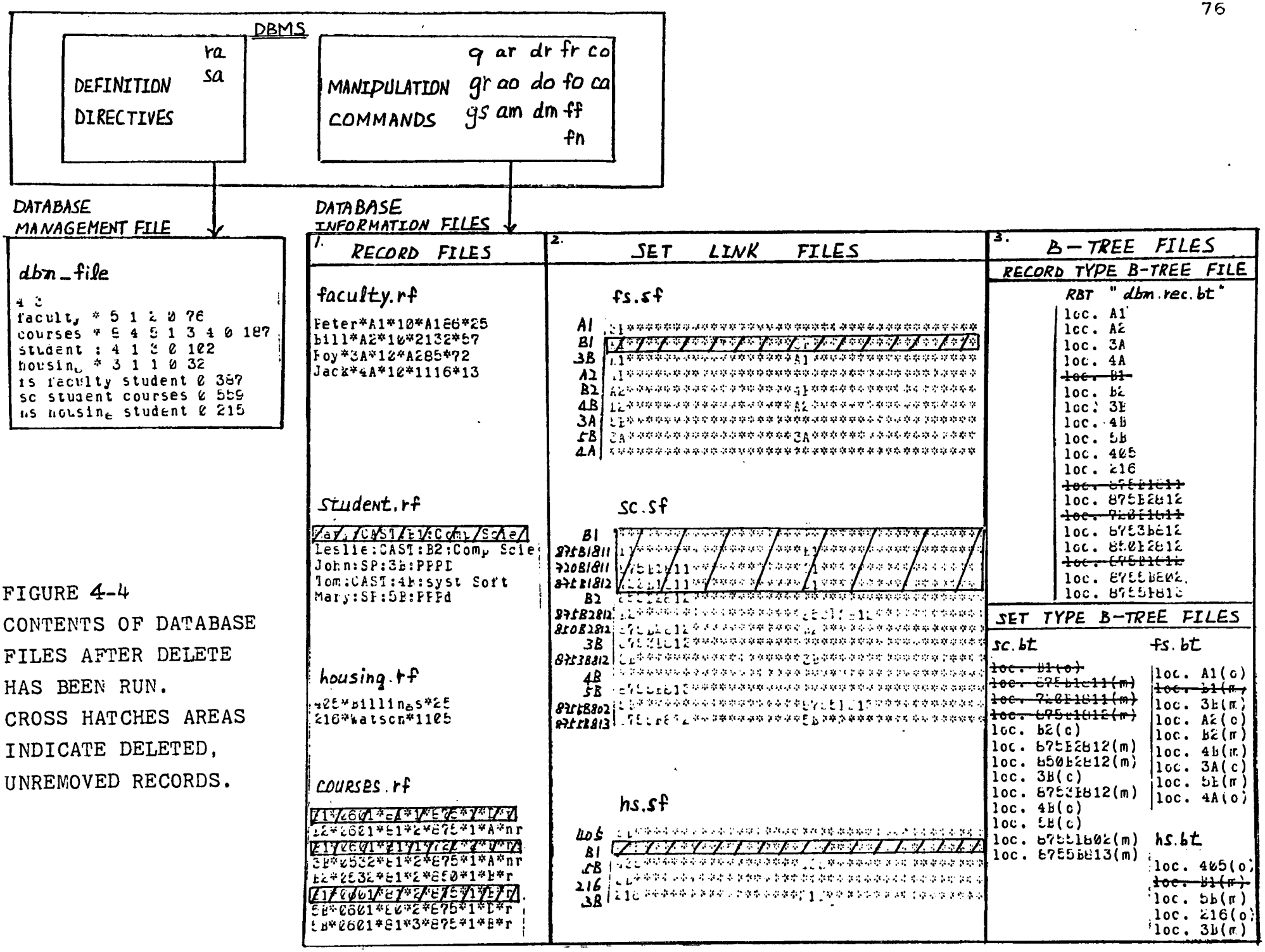
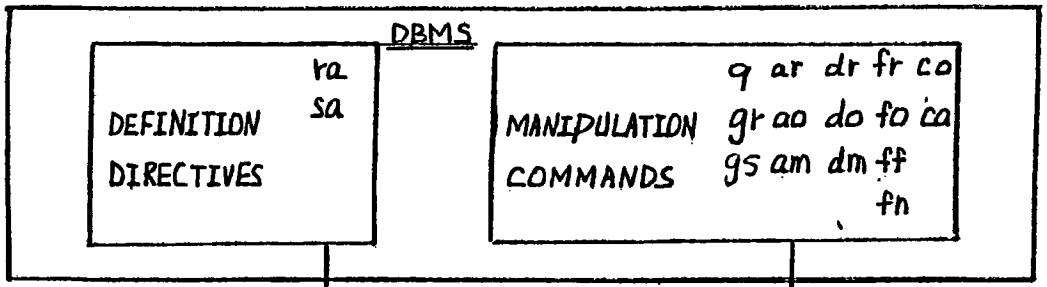



FIGURE 4-4
CONTENTS OF DATABASE FILES AFTER DELETE HAS BEEN RUN. CROSS HATCHES AREAS INDICATE DELETED, UNREMOVED RECORDS.




DATABASE MANAGEMENT FILE

**dbm_file**

```
4 3
faculty * 5 1 2 0 76
courses * 8 4 5 1 3 4 0 187
student : 4 1 3 0 102
housing * 3 1 1 0 32
fs faculty student 0 387
sc student courses 0 559
hs housing student 0 215
```

DATABASE INFORMATION FILES

1. RECORD FILES

**faculty.rf**

```
Peter*A1*10*A186*25
B111*A2*10*2132*57
Roy*3A*10*A285*72
Jack*4A*10*1116*13
```

**student.rf**

```
Mary:CAST:B1:Comp Scie
Leslie:CAST:B2:Comp Scie
John:SP:3B:PPPD
Tom:CAST:4B:syst Soft
Mary:SP:5B:PFPd
```

**housing.rf**

```
405*Billings*25
216*Watson*1105
```

**courses.rf**

```
B1*0601*B1*1*875*1*D*r
B2*0601*B1*2*875*1*A*nr
B1*0601*B1*1*720*2*B*nr
3B*0532*B1*2*875*1*A*nr
B2*0532*B1*2*850*1*B*r
B1*0601*B1*2*875*1*B*r
5B*0601*B0*2*875*1*D*r
5B*0601*B1*3*875*1*B*r
```

2. SET LINK FILES

**fs.sf**

```
A1 | 3B**********
B1 | A1**********3B**********
3B | B1**********A1**********
A2 | 4B**********
B2 | A2**********4B**********
4B | B2**********A2**********
3A | 5B**********
5B | 3A**********3A**********
4A | **********
```

**sc.sf**

```
B1       | 875B1812**********
875B1811 | B1**********720B1811**********
720B1811 | 875B1811**********875B1812**********
875B1812 | 720B1811**********B1**********
B2       | 850B2812**********
850B2812 | B2**********850B2812**********
875B2812 | 875B2812**********B2**********
3B       | 8753B812**********
8753B812 | 3B**********3B**********
4B       | **********
5B       | 8755B813**********
8755B802 | 5B**********8755B813**********
8755B813 | 8755B802**********5B**********
```

**hs.sf**

```
405 | 5B**********
B1  | 405**********5B**********
5B  | B1**********405**********
216 | 3B**********
3B  | 216**********216**********
```

3. B-TREE FILES

RECORD TYPE B-TREE FILE

**RBT "dbm.rec.bt"**

```
loc. A1
loc. A2
loc. 3A
loc. 4A
loc. B1
loc. B2
loc. 3B
loc. 4B
loc. 5B
loc. 405
loc. 216
loc. 875B1811
loc. 875B2812
loc. 720B1811
loc. 8753B812
loc. 850B2B12
loc. 875B1812
loc. 8755B802
loc. 8755B813
```

SET TYPE B-TREE FILES

**sc.bt**

```
loc. B1(o)
loc. 875B1811(m)
loc. 720B1811(m)
loc. 875B1812(m)
loc. B2(o)
loc. 875B2812(m)
loc. 850B2B12(m)
loc. 3B(c)
loc. 8753B812(m)
loc. 4B(o)
loc. 5B(o)
loc. 8755B802(m)
loc. 8755B813(m)
```

**fs.bt**

```
loc. A1(o)
loc. B1(m)
loc. 3B(m)
loc. A2(o)
loc. B2(m)
loc. 4B(m)
loc. 3A(o)
loc. 5B(m)
loc. 4A(o)
```

**hs.bt**

```
loc. 405(o)
loc. B1(m)
loc. 5B(m)
loc. 216(o)
loc. 3B(m)
```

FIGURE 4-3
DATABASE FILES
AFTER THE DBMS
HAS BEEN RUN.

### 4.2.4. Algorithms

For the system involved, the approach to the garbage collection problem for record type files is the inspect and slide function. Armed with the record type definition, the garbage collector moves down the record file, inspecting each record. The key for the record is recreated using the record type definition Then the B-tree is searched for that key. If the key is present in the B-tree, the record slides as far forward on the file as possible, and then the offsets of this record in the B-tree reset to reflect the new location. If the key was not in the B-tree, the record overlaid when a valid record from farther down the file was slid over it. This method works well for the record type files; however, it can not work for the set type files that contain link pointers information.

The method used for the set type files is a B-tree walk and move procedure. A tool is built that would visit each node in the B-tree, the set link record is moved to a new file and the offsets readjusted to fit the link record's new placement. Once the new "clean file is built, the old file space is returned to the system.

Here, two utilities: "gr" (record garbage collector) and "gs" (set garbage collector) are implemented. There

are many ways to collect wasted file space. The designer of the garbage collection tool will have to weigh the merits of each possibility, and implement the most reasonable approach.

# BIBLIOGRAPHY

Arnborg, S., Optimal memory management in a system with garbage collection, BIT 14, 4(1974), pp. 375-381

Baker, H.G., List-processing inreal time on a serial computer CACM 21, 4(april 1978), pp. 280-294

Barth, J.M., Shifting garbage collection overhead to compile time CACM 20, 7(July 1977), pp. 513-518

Bawden, A., Greenblatt, R., Holloway, J., Knight, T., Moon, D., and Weinreb, D., Lisp machine progress report, Memo 4444, A.I. Lab. M.I.T., Cambridge, Mass., Aug. 1977

Bishop, P.B., Computer systems with a very large address space and garbage collection Lab. for Computer Science, MIT Rep., TR-178, M.I.T., Cambridge, Mass., May 1977

Bobrow, D.G., and Murphy, D.L. structure of a LISP system using two-level storage CACM 10, 3(March 1967), pp. 155-159

Bobrow, D.G., and Murphy, D.L., A note on the efficiency of a LISP computation in a paged machine, CACM 11, 8(Aug. 1968), pp. 558-560

Bobrow, D.G., A note on hash linking CACM 18, 7(July 1975), pp. 413-415

Cherey, C.J., A nonrecursive list compacting algorithm CACM 13, 11(Nov. 1970), pp. 677-678

Clark, D.W., A fast algorithm for copying binary trees, Inf. Process. Lett. 5, 3(Dec. 1975), pp. 62-63

Clark, D.W., An efficient list moving algorithm using constant workspace, CACM 19, 6(June 1976), pp. 352-354

Clark, D.W., and Green, C.C., An empirical study of list structure in Lisp, CACM 20, 2(Feb. 1977), pp. 78-86

Clark, D.W., A fast algorithm for copying list structures, CACM 21, 5(May 1978), pp. 351-357

Cohen, J., and Trilling, L., Remarks on Garbage Collection using a two level storage, BIT 7, 1(Jan. 1967), pp. 22-30

Cohen, J., Use of fast and slow memories in list-processing languages CACM 10, 2(Feb. 1967), pp. 82-86

Cohen, J., and Zuckerman, C., Evalquote in simple Fortran: A tutorial on interpreting Lisp, BIT 12, 3(1972), pp. 299-317

Collin, B.E., A method for overlapping and erasure of lists, CACAM 3, 12(Dec. 1960), pp. 655-657

Date, C. J., An Introduction To Database Systems, Addison-Wasley Publishing Company,1977

Deitel, H. M., An Introduction to Operation System Addison-Wasley Publishing Company, 1983

Deutsch. L.P., An efficient incremental automatic garbage coll_ctor CACM 19. 9(Sept. 1976), pp. 522-526

Dijkstra, E.W., Lamport, I., Martin, A.J., Scholten, C.S., and Steffens, E.F.M., On-the-fly garbage collection: An exercise in cooperation, CACM 21, 11(Nov 1978), pp. 966-975

Dvorch, M.A., The Implementation Of A Network Oriented Database Management System Designed To Run Under The UNIX Operation System RIT thesis, Jan,1982.

Fenichel, R.R. and Yochelson, J.C., A LISP Garbage-Collector for Virtual-Memory Computer Systems, CACM 12,11(Nov,1969) pp. 611-612

Fisher, D.A., Boounded workspace garbage collection in an address order preserving list processing environment, Inf. Process. Lett., 3, 1(July 1974). pp. 29-32

Fisher, D.A., Copying cyclic list structure in linear time using bounded workspace, CACM 18,5(May 1975), pp. 251-252

Fitch, J.P., Norman, A.C. a note on compacting garbage coll_ction, Comput. J. 21. 1(Feb. 1978), pp. 31-34

France, N., An application of a method for analysis of

cyclic programs, IEEE Trans. Softw. Eng. 4. 5(Sept. 1978), pp. 371-377

Fredman, D.P., and Wise, D.S., Garbage collecting a heap which included a scatter table, Inf. Process Lett. 5, 6(Dec. 1976), pp. 161-164

Gotlieb, C.C., and Gotlieb, L.R., Data types and structures, Prentice-Hall, Englewood Cliffs, N.J., 1978

Gries, D., An exercise in proving parallel programs correct, CACM 20, 12(Dec. 1977), pp. 921-930

Haddon, B.K., and Waite, W.M., A compaction procedure for variable length storage element. Comput. J., 10(Aug, 1967), pp. 162-165

Hansen, W.J., Compact List Representation: Definition,Garbage Collection, And System Implementation CACM 12,9(Sep,1969), pp. 499-507

Hanson, D.R., Storage management for an implementation of Snobol 4, Software: Practice and Experience 7, 2(1977), pp. 179-192

Horowitz, E., Sahni, S., Fundamentals Of Data Structures, Computer Science Press, Inc. 1976

Kernighan and Ritchie, The C Programming Language, Prentice-Hall,INC.,1978

Kernighan and Ritchie, "UNIX Programming - Second Edition"

Knuth, D. E., The Art of Computer Programming, Vol.1, Fundamental Algorithms Addison-Wasley Publishing Company,1973

Lamport, L., Garbage collection with multiple processes: An exercise in parallelism, Proc. IEEE conf. Parallel Processing, Aug. 1976

Lang, P., and Wegbreit, B., Fast compactification, Rep. 25-72, Harvard Univ., Cambridge, Mass., Nov. 1972

Martin J., Computer Data-Base Organization Prentice-Hall,INC. 1977

Marshall, S., An Algol-68 garbage collector, in Algol 68 implementation J.E.I. peck (Ed.), North-Holland, Amsterdam, 1971, pp. 239-243

Minsky, M.L., A Lisp garbage collector algorithm using seri_l secondary storage Memo 58 (rev.), Project MAC, M.I.T., Cambridge, Mass, Dec. 1963

Moon, D.A., MACLisp reference manual, Project MAC, M.I.T., Cambridge, Mass, April 1974

Morris, F.l., A time- and space- efficient garbage collection algorithm, CACM 21, 8(Aug. 1978), pp. 662-665

Morris, F.L., On a comparison of garbage collection tech-

niques, technical correspondence, CACM 22, 10(Oct. 1979), pp. 571

Muller, K.G., On the feasibility of concurrent garbage coll_ction Ph.D thesis, Tech. Hogeschool Delft, March 1976

Pfaltz, J.L., Computer Date Structures McGraw-Hill,Inc. 1977

Reingold, E.M., A nonrecursive list moving algorithm, CACM 16, 5(May 1973), pp. 305-307

Robson, J.M., A bounded storage algorithm for copying cyclic structures, CACM 20, 6(June 1977), pp. 431-433

Ross. D.T., The AED free storage package CACM 10, 8(Aug. 1967), pp. 481-492

Schorr, H., Waite, M., ,ul 2 A Efficient Machine-Independent Procedure For Garbage Collection Various list Structures, CACM 10,8(Aug,1967), pp. 501-506

Schorr, H., and Yochelson, J.C., An Efficient Machine-Independent Procedure For Garbage Collection In Various List Structures, CACM 10,8(Aug,1967), pp. 501-506

SEED---a course for new users International DP system Inc., 1981.

Standish, T.A , Date structures techniques Addison-wesley,

Reading, Mass, 1980

Steele, G.L., Multiprocessing Compactifying Garbage Collection, CACM 19, 9(SEPT. 1976), pp. 495-508

Terashima, M., and Goto, E., Genetic order and compactifying garbage collectors, Inf. Process. Lett. 7, 1(Jan. 1978), pp. 27-32

Thorelli, L.E., A fast compactifying garbage collector, BIT 16, 4(1976), pp. 426-441

Wadler, P.L., Analysis Of An Altorithm For Real Time Garbge Collection, CACM 19,9(Sept. 1976), pp. 491-500

Waite, W.M., Implementing software for non-numeric applications, Prentice-Hall, englewood cliffs, N.J., 1973

Walden, D.C., A note on Cheney's nonrecursive list-compacting algorithm, CACM 15, 4(April 1972), pp. 275

Wegbreit, B., A Generalized Compactifying Garbage Collector, Comput. J., 15, 3(Aug 1972), pp. 204-208

Weissman, C., Lisp 1.5 primer, Dickenson Publ., Belmont, Calif., 1967

Weizenbaum, J., Symmetric list processor, ACM 6, 9(Sept. 1963), pp. 524-544

Weizenbaum, J., Recovery Of Reentrant List Structures In

SLIP, CACM 12,7(July,1969) pp. 370-372

Winston P.H., Horn B.K.P. LISP, Addison-wesley, Mass, 1981

Wirth, Niklaus, Algorithms + data structures = Programs, Englewood Cliffs, N.J., Prentice Hall, 1976

Wise, D.S., Morris garbage compaction algorithm restores reference counts, ACM Trans. Programm. Lang. Syst. 1, 1(July 1979), pp. 115-120

Wodon, P.L., Methods of garbage collection for Algol 68. in Algol 68 implementation, J. E. L. Peck (Ed.), North-Holland, Amsterdam, 1971, pp. 245-262

Zave, D.A., A fast compacting garbage collector, Inf. Process. Lett. 3, 6(July 1975), pp. 167-169

by resetting all counters to zero. The counters of the accessible cells are restored during the marking phase of the collection by incrementing a cell's counter every time the cell is visited. The collection reclaims inactive circular list structures and cells with maximum refcount which have become unreachable. The point of this hybrid scheme is to keep track of very small refcounts between necessary invocation of garbage collection so that nodes which are allocated and quickly abandoned can be returned to available space, delaying necessity for garbage collection.

The hybrid approach of garbage collection and reference counting suggested by Deutsch and Bobrow is particularly applicable to LISP [see DEUTSCH 76]. According to Clark [CLARK 77], it is based on statistical evidence that in most LISP programs, most refcounts (about 97 percent) are one. They propose three hash tables [BOBROW 75]:

(1) The multiple reference table (MRT): Its key is a cell address and the associated value is the cell's refcount. Only cells whose refcounts are two or greater are listed in the MRT.

(2) the zero count table (ZCT) contains the addresses of cells whose refcount is zero. These cells may be of two types, those which are referred to only by the variables