Garbage Collecting a Heap which includes a Scatter Table

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 34

GARBAGE COLLECTING A HEAP WHICH INCLUDES A SCATTER TABLE

DANIEL P. FRIEDMAN

DAVID S. WISE

REVISED: AUGUST, 1976

Garbage Collecting a Heap which includes a Scatter Table

Daniel P. Friedman

David S. Wise

Computer Science Department

Indiana University

Bloomington, Indiana 47401

Abstract: A new algorithm is introduced for garbage collecting a heap which contains shared data structures accessed from a scatter table. The scheme provides for the purging of useless entries from the scatter table with no traversals beyond the two required by classic collection schemes. For languages which use scatter tables to sustain unique existence of complex structures, like natural variables of SNOBOL4, it indirectly allows liberal use of a single scatter table by ensuring efficient deletion of useless entries. Since the scatter table is completely restructured during the course of execution, the hashing scheme itself is easily altered during garbage collection whenever skewed loading of the scatter table warrants abandonment of the old hashing. This procedure is applicable to the maintenance of dynamic structures such as those in information retrieval schemes or in languages like LISP or SNOBOL4.

Keywords and phrases: hash table, bucket, key, inverted tree, oriented tree, chaining, rehashing, SNOBOL4, LISP.

CR categories: 3.74, 4.1, 4.34.

Research reported herein was supported (in part) by the National Science Foundation under grant no. DCR75-06678 and no. MCS75-08145.

Garbage collection [§2.3.5]* is generally a two-pass scheme for recovering unused nodes from a heap, a region of computer memory divided into nodes accessed by the user only via references. During the first pass all useful structures represented within the heap are traversed and a mark bit is set within every node encountered. Then the heap is traversed or swept completely and every unmarked node is added to a list of available space. Additional traversals may provide for repacking the structures into adjacent nodes at one end of the heap [§2.3.6-9] and for such repacking when nodes are of variable size [2]. In this paper we assume that all nodes in the heap are of equal size and that no repacking is necessary; if these features are needed then the scheme presented here can be easily extended.

A scatter table [§6.4] is a dynamic structure often used for efficiently associating ALGOL 60 style identifiers with values. It is a common part of the input phase of a computer program used to associate new occurrences of an identifier with information established earlier; the symbol table in an interpreter or compiler is an example of its application. Another use is within interpreters like that of SNOBOL4, which maintains unique instances of data structures and clash every newly created structure against extant structures through a scatter table [2, Chapter 8]. The common structure of such a table is a fixed size sequential array of references to linearly linked lists [1]. The array is accessed by a pseudo-function, the hash function, mapping identifiers to array indices hopefully scattering the identifiers uniformly. When an

---

*All § section references in this paper are to Knuth [3].

identifier or key is encountered in input it is hashed to a subscript for the array which selects a linearly linked list called a bucket. That bucket is scanned for an occurrence of the newly encountered key; if no occurrence is found a new entry may be added. Because bucket size and therefore table size is dynamic, a bucket scan usually requires a linear traversal. The net effect of the scattering strategy is reduction of the length of that traversal by a factor of the scatter table size over the search required if there were only a single bucket, that is, if all identifier associations were maintained in a single list.

When a scatter table is used in a system with a heap it is natural to maintain the table in the heap itself since its structure is similar to that of other structures represented there. If this is done then the garbage collection algorithm must carefully preserve all associations represented within the scatter table; their safety is guaranteed by marking all nodes representing these associations during a traversal of the scatter table in the first phase of garbage collection. Not every entry in the scatter table need be marked during this traversal since entries without associated values do not contribute to the information content of the scatter table. Such useless entries may occur if an identifier has occurred but has not yet been associated or if a prior association has been cancelled.

Define an element to be a node in the heap which represents the association between an identifier and its value. Since the

elementary items in a data structure (sometimes called atoms) are usually identifiers of some sort we shall assume that they are also represented as elements. Elements are therefore distinguished nodes; every node in the heap has a Boolean ATOM field which is set true only in elements (Figure 1). An element contains an INFRASTRUCTURE, composed of its key and its value and a reference field NEXTINBUCKET for structuring the scatter table. If a particular element is not in the scatter table and therefore free from being associated with future uses of its key in input (e.g. REMOBed or GENSYMed atoms in LISP) then its NEXTINBUCKET field contains the reference NIL. Finally, like all nodes in the heap, an element has a MARK bit for use by the garbage collector.

Since the garbage collector can distinguish elements from other kinds of nodes, it should be able to recover space used to represent them taking into account that some may still be useful entries in the scatter table. An element is useless if it is not accessible through the structure associated with another element in the scatter table and either it is not in the scatter table or it has no value association yet established. A useless element may be purged from the system without affecting accessible information; if its key is encountered on input sometime after garbage collection then the element may be regenerated with no information lost during its absence. While a useless element is easily detectable after the marking phase of garbage collection, it still must be carefully removed from its bucket lest the bucket's structure be destroyed by the loss of the purged elements NEXTINBUCKET field.

A naive algorithm for purging useless elements during garbage collection requires that the buckets be maintained as doubly-linked lists to allow deletion of a useless element from its bucket when it is discovered during the sweep phase. An alternative allows the single-linked buckets which we have assumed but requires a traversal of every bucket between the mark phase and the sweep phase. During this pass unmarked elements are removed from their buckets [2, Chapter 8].

We present a garbage collection scheme below which provides for the purging of useless elements from singly-linked buckets during a simple two-pass garbage collection. During the first pass each bucket is completely but cheaply restructured; the second pass restores the linear structure including only the non-useless elements in perhaps a different order. The restructuring is easily understood if we view each bucket as a single-level oriented tree [§2.3] with the bucket's entry in the scatter array (the bucket header) as the root and each element in the bucket as its son (Figure 2). Usually the tree is represented as its naturally corresponding binary tree [§2.3.2] based on an undefined ordering of the oriented tree. This structure amounts to the linear list representation of a bucket which we have assumed (Figure 3a), because no node is both a father and a son. Between phases of garbage collection a bucket is instead represented as an inverted tree (Figure 3b) with each son (element) referencing its father and the root nodes (bucket headers) referencing NIL. During the sweep phase the bucket is restored to its linearly linked form (Figure 3c).

The marking phase of the algorithm [§2.3.5] traverses the scatter table inverting all buckets, marking all associated elements, and traversing all of their infrastructures--including those of any elements encountered. In this way useful elements not in the scatter table are marked. The two effects of this phase on a useful element in the scatter table do not necessarily occur at once; an element is marked as soon as it is encountered in traversing some useful structure but an element has its NEXTINBUCKET field reset to reference its bucket header only as it is encountered by the scatter table traversal. The marking algorithm below may be preceeded by marking traversals of structures rooted in system structures disjoint from the scatter table.

```
ELT := FIRSTINBUCKET(B);
FIRSTINBUCKET(B) := NIL;
WHILE ELT ≠ NIL DO
    IF ELT is yet unmarked and has a value THEN
        traverse and mark ELT and its infrastructure;
    TEMP := NEXTINBUCKET(ELT);
    NEXTINBUCKET(ELT) := B;
    ELT := TEMP.
```

After this algorithm the former order of each bucket (Figure 3a) is lost. The inverted tree representation which results (Figure 3b) still fits the definition of a bucket as an oriented tree. An effect of the sweep phase below is to reconstitute each bucket in

its linear form (Figure 3c) restoring only non-useless elements to a structure ordered by their machine addresses in the reverse order of the sweep through memory. It is therefore adventageous to locate all frequently-used, system-defined elements at the end of the heap to be swept last and restored at the head of their respective buckets.

```
Sweep phase:  FOR all nodes, N, in heap DO
              {  IF N is unmarked THEN AVAIL ⇐ N [§2.2.3] ELSE
                 {  Clear N's mark;
                    IF N is an element THEN
                    {  B := NEXTINBUCKET(N);
                       IF B ≠ NIL THEN
                       {  NEXTINBUCKET(N) := FIRSTINBUCKET(B);
                          FIRSTINBUCKET(B) := N.
```

Two features of classic hashing/garbage-collection schemes are not possible using these algorithms. First, no bucket-search scheme can depend upon an ordering within each bucket (other than by location), so secondary hash schemes like SNOBOL4's [2, Chapter 9] must be abandoned. Secondly, the garbage collection must proceed to completion before the scatter table may be used again; Minsky's proposal of a parallel process for garbage collection [§2.3.5-12] therefore cannot be adopted.

This garbage collector has been introduced as one which allows efficient recovery of useless nodes including those

representing useless elements in the scatter table. Because the scatter table is completely reconstructed during the course of garbage collection and there is a particular point (between the mark phase and the sweep phase) at which the scatter table appears to be empty, it provides a fine opportunity to completely revise the hashing scheme. Statistics gathered during the traversal of the scatter table in the mark phase might indicate that the current hashing scheme is inadequate. (Either the hash function might be generating a skewed distribution of keys among the buckets or the buckets might be uniformly overfilled indicating a need for more buckets [1].) In that event a new hashing scheme can be chosen before the sweep phase and then during the sweep, instead of returning a marked element to its former bucket, its key can be rehashed under the new scheme and the element placed into the appropriate bucket of a fresh scatter table. With such a provision garbage collection might result in a correction of a breakdown in the hashing scheme, as well as in recovery of unused space.

References

1. D. Gries. Compiler Construction for Digital Computers, Wiley, New York (1971), 216-224.

2. Ralph E. Griswold. The Macro Implementation of SNOBOL4. W. H. Freeman, San Francisco (1972), Chapters 8-9.

3. Donald E. Knuth. The Art of Computer Programming 1 & 3, Addison Wesley, Reading, MA (1975 & 1973), Sections 2.3, 2.3.5, and 6.4.

Figure 1. Fields in a node representing an element.

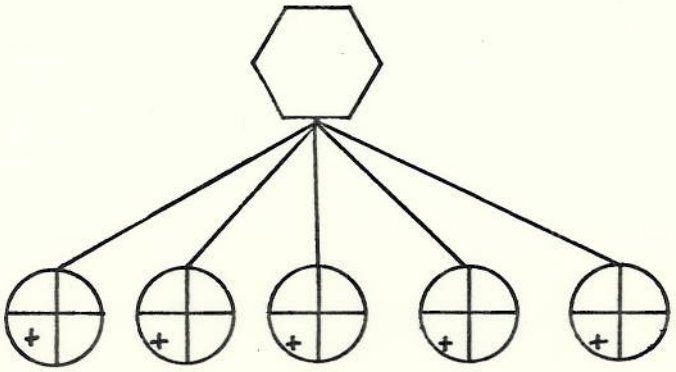
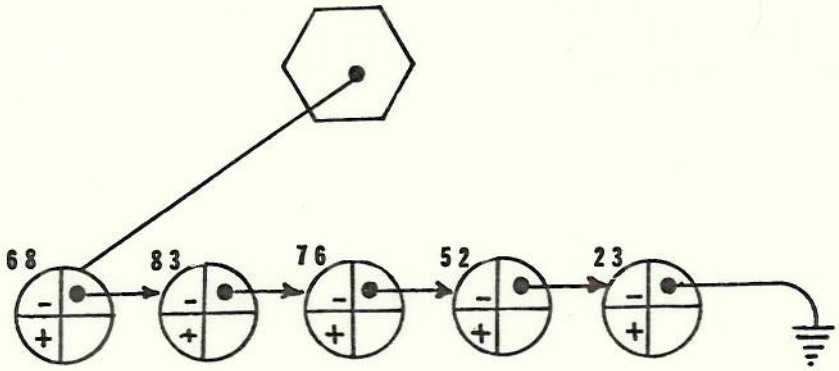
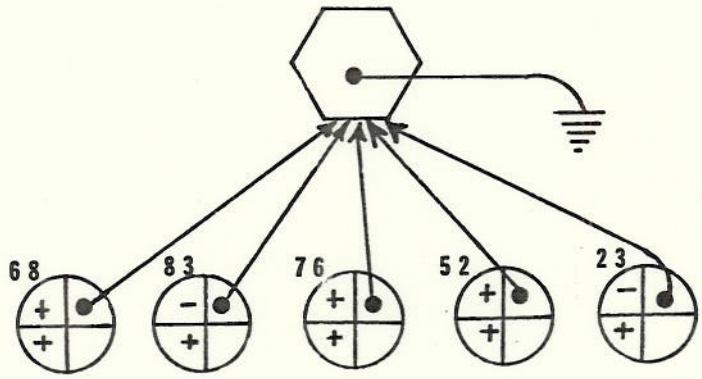


Figure 2. A bucket of five elements.





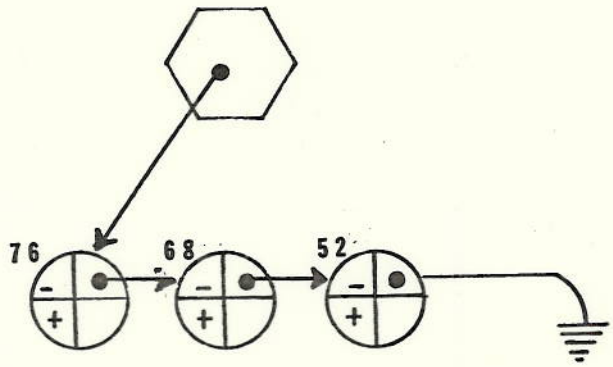


Figure 3a. The representation of the bucket before garbage collection.

Figure 3b. The representation of the bucket between the phases of garbage collection.

Figure 3c. The representation of the bucket after garbage collection without useless elements assuming a sweep in order of incresing address.