REFERENCING LISTS BY AN EDGE

David S. Wise

Computer Science Department
Indiana University
Bloomington, Indiana 47401

TECHNICAL REPORT NO. 9

REFERENCING LISTS BY AN EDGE

DAVID S. WISE

REVISED: JUNE, 1975

To appear in
Communications of the ACM
June or July, 1976

REFERENCING LISTS BY AN EDGE

David S. Wise
Computer Science Department
Indiana University
Bloomington, Indiana

Abstract

An edge reference into a list structure is a pair of pointers to adjacent nodes. Such a reference often requires little additional space, but its use can yield efficient algorithms. For instance, a circular link between the ends of a list is redundant if the list is always referenced by that edge, and list traversal is easier when that link is null. Edge references also allow threading of non-recursive lists, can replace some header cells, and enhance the famous exclusive-or trick to double-link lists.

Key words and phrases: list processing, circular, doubly-linked, overlapping sublist, header cell, pointer, cursor.

CR categories: 4.22, 4.10, 3.73.

Introduction and Definition

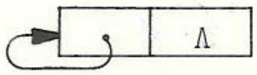
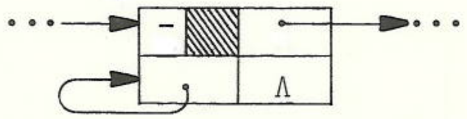
The purpose of this communication is to indicate some natural advantages of referring to and into lists by pointing to an edge (i.e., to two adjacent nodes). These appear as space savings for data structures, and often as time savings for algorithms which depend upon edge references.

In the following sections the list structures under consideration are defined, and the technique of referring to these structures by the edge between first and final nodes is demonstrated. The sample algorithms, not surprisingly, involve traversals. However, edge references are shown to yield faster list traversal algorithms when used in place of circular links and they allow a natural threading of non-recursive lists. Finally, an application of edge pointers into a list (to any pair of adjacent nodes) is presented. Such a reference (or cursor) enables the famous "exclusive-or" double linking technique to be used in many situations where double linking is needed, but space is tight.

Knuth's terminology [1] on list structures is used here with the definition of a list slightly modified. A list is an ordered sequence of zero or more elementary elements or extant lists. This definition is weaker than Knuth's in that sublists are allowed but stronger than his definition of a List because a list cannot contain itself. Recursive lists are therefore prohibited. Sublists need not be disjoint so we are considering lists which can overlap or which are shared with other structures. The act of adding a sublist which is already part of another self-sufficient structure, thereby establishing an overlap, is sometimes called borrowing. The resulting structures describe those used in LISP [2], SAC-1 [3], and SLIP [4]. (As in the former langauges, we shall accept restrictions on the ways a shared sublist can be changed.)

The INFORMATION field provided in a fixed size node often plays a dual role in a list environment. It is large enough to carry the elementary item which is required by the host hardware (e.g., a floating point number) but is often only used to point to a sublist. With an entire word (e.g., 36-48 bits) available for a pointer on many machines we can maintain two (18-24 bit) pointers to each substructure with no increase in node size. These pointers will be used here to point to the first (H or HEAD) and last (T or TAIL) node of every sublist, and every list pointer will be so substructured.

Figure 1 illustrates a typical list. Each node carries a bit indicating the TYPE of the INFORMATION field, elementary or sublist, and a pointer to the NEXT node in the list. The double-pointer (T,H) refers to the edge which would connect the first and last nodes of a list and allows direct access to their contents. The use of the link or NEXT field of node T is distinguished because it affects the design of all algorithms which operate on the list and therefore it characterizes the structure. Very often, as in Figure 2b, it is simply null; it can also be a circular link back to H as in Figure 2b or a thread which is an "up" link to a higher level in the list structure and is often tagged by an extra bit to distinguish it from "across" links (Figure 2c). Direct access to this distinguished field via the edge reference allows its use to be changed each time the list is accessed afresh.

For a reference to an empty list we specify $H = \Lambda = T$. Other representations of the empty list are possible but this convention provides a symmetry desirable in some algorithms below. In singly linked systems where elementary items are represented by references to atomic structures and the empty list is itself a reference to the atom Λ [2], the H = Λ = T convention is a special case of double-referencing every atomic item, A , by T = Λ and H = A . That is, T = Λ indicates that the double pointer refers to the atom H ; the TYPE field is therefore redundant. The convention of referring to every list by its HEAD and TAIL effects the standard linking structure for a queue (actually an output-restricted deque). An extension of that scheme [1:p.257] suggests that an empty list might alternatively be represented with HEAD null and with TAIL pointing to the word (<u>not</u> the node) containing the list pointer, where the HEAD and NEXT fields are the same relative positions within a word. Then a pointer to an empty list would be  and a node pointing to an empty sublist might appear as  This possibility influenced the ordering of double pointers as (T,H) instead of (H,T) .

Operations

The main benefit of direct access to the last node of a list is easy alteration of its link field. Some well-known operations thereby become simple: adding to the tail of a queue, list concatenation (NCONC, as opposed to APPEND, in LISP), and returning a list to available space without traversing it. An edge reference to the list allows the previous available space list to be concatenated at its tail [1:2.2.3-29b] or an edge reference to available space allows the list being returned to be spliced on at the end of the available space list. Weizenbaum used the latter operation to delay the erasing of sublists of an erased list until all space previously available had been exhausted [4 and 1: p.413].

Inspection of Figure 1 convinces one that it would be trivial to insert the circular link into any edge referenced list: $\text{NEXT}(T) \leftarrow H$. The redundant circular linking can often be abandoned with double pointers. That is, the structure of Figure 2a is profitably converted into that of Figure 2b when the structure is in fast memory and available to only one user. Whenever a list is being traversed it is more efficient to detect the last link because it is null (Figure 2b) or tagged as a thread (Figure 2c) than to test it equal to a starting address stored as a variable. If circular linking is needed later, say for the rapid rolling of a round-robin queue, it can still be restored. On the assumption that such operations are restricted to more static structures, the schemes of Figures 2b and 2c will be used here.

Algorithm 1 is offered as an example of the power of changing from circular to linear linking. It takes as input a circular list similar to that of Figure 2a and reverses all links at the top level. The initial pointer is (T,H) . (Actually the initial value of T is not used.) In studying the algorithm, note carefully the effect of the first execution of the loop. For the empty list nothing is done. For a list of $n > 0$ nodes the first iteration changes the structure from that of Figure 2a to a slightly reordered list of the type shown in Figure 2b, and n more iterations remain to reverse the list.* Comparison of this algorithm to Knuth's solution for a restricted problem [1:2.2.4-5 on p.547], demonstrates that this is faster for all but very short lists.

*This elegant loop is due to Benna Kay Young.

Algorithm 1: Reverse a circular list. A is an auxiliary pointer.

```
A ← Λ .
while H ≠ Λ do
    ⎧T ← A ;
    ⎨A ← H ;
    ⎪H ← NEXT(H) ;
    ⎩NEXT(A) ← T .
H ← A ;
T ↔ H . ■
```

The last step, exchanging T and H , is not essential to Knuth's problem but is included to restore the conventional pointer.

Algorithm 1 does not initially require a tail pointer to operate properly, but its operation is more easily understood from the viewpoint of paired pointers. Algorithm 2 is offered as an algorithm for which this paired referencing is essential. It traverses a list in depth-first order (if a tree then in preorder) as might be done during the marking phase of garbage collection. Threads are placed as in Figure 2c only while traversing a particular sublist (newly answering [1:2.3.5-2]). Since sublists need not be disjoint, threads may change during the course of execution.

Algorithm 2: Traverse the non-empty structure of the isolated node P in preorder assuming NEXT(P) is initially null.

```
while P ≠ Λ do
    ⎧Visit node P;
    ⎪while TYPE(P) is sublist & HEAD(P) ≠ Λ do
    ⎨       ⎧NEXT(TAIL(P)) ← NEXT(P);
    ⎪       ⎨P ← HEAD(P);
    ⎪       ⎩Visit the son-node P;
    ⎩P ← NEXT(P) . ■
```

Another dynamic threading scheme for lists would thread the end of a sublist to the father: NEXT(TAIL(P)) ← P . In Figure 2c the thread would point to N . Using such a scheme one can write efficient traversal algorithms for lists which are sensitive to level changes because a thread from P is then distinguished from an ordinary NEXT link by TAIL(NEXT(P)) = P . Such algorithms are necessary to copy or compare lists.

Off with his Header

The reader familiar with some uses of header cells [1,4] might suggest that the usual contents of such a cell has been simply moved into the sublist reference (T,H) into NEXT(T) , and into node H . To do so would indicate confusion between two concepts which are often used together. Some header cells [1:p.409] do not refer to edges, and some edge references may point into structures with no sublists (i.e. linear lists) and therefore with no need of header cells. Space may indeed be saved by using an edge reference rather than a header cell when the only purpose of the header would be to access the TAIL or to mark the limit of circular linking.

Without header cells some restrictions on the way in which list structures can be changed arise because lists can overlap. Under the philosophy of Lists in [1:2.3.5] where any change in a sublist affects <u>all</u> references to that list, the first and last nodes of any sublist may not be changed. In that case, the extremal nodes (perhaps coinciding) play the role of a list header but changes are allowed between them.

Another philosophy [3] allows lists to be borrowed, and alteration of a borrowed list has only local impact. This means that alterations of shared singly linked lists, like Figure 1, must not occur anywhere but at the left end, as in Figure 3. When a change is made elsewhere the shared list must be copied up through the change, borrowing the suffix from the original list. (This technique is essential to LISP [2].)

<u>$\oplus$-Lists</u>

Doubly linked sublists might be implemented by using the shaded fields of Figure 1 as reverse pointers, but the technique described below might also be used to implement doubly linked lists with no additional space. Let B's NEXT field take the value of the "exclusive or", denoted $\oplus$, of references to the left and right neighbors, A and C, of node B. Knuth points out [1: 2.2.4-18] that any invertable operation might do for $\oplus$, but we make this choice because $\oplus$ is its own inverse and is commutative, associative, and (usually) cheap to implement. This structure is called an <u>$\oplus$-list</u> and is illustrated in Figure 4. Such a structure may be circular if needed. Siklóssy uses a similar scheme for binary trees [5].

Just as we represented an edge reference to a list, we shall now assume that a reference <u>into</u> a list is represented by an edge (two adjacent nodes). This convention distinguishes between a reference to a node, from which only its information can be retrieved, and a reference to a position in a list from which the rest of that list may be traversed. Calling the latter a <u>cursor</u>, we see that a cursor can refer to a node in the list, but a pointer to a node is not sufficient to determine a cursor at its position.

The list references discussed earlier in this paper can be taken as cursors to the edge linking the ends of a ⊕-list. Similarly, if (T,H) is taken to be an arbitrary cursor, then T refers to a node on the left of the cursor, and H points to the node to the right. Operations to the right of a cursor correspond to operations at the head of a list when the cursor is taken as a list reference.

Several operations on a cursor into a ⊕-list are indicated below. It is important here that Λ be 0 , the identity, and that the empty list (or null cursor) is indicated by $H = T = \Lambda$. Although the ⊕-list belongs to folklore, these operations are worth repeating here to demonstrate that they are only a little more expensive than comparable modifications on paired-pointers as list references. Edge references into dynamic structures require more bothersome maintenance.

The extra time needed for this scheme is offset by space savings. For a doubly-linked, multi-levelled structure we can use a short NEXT field for double-linking and a long sublist reference (or cursor) rather than vice-versa. This means that the NEXT field, in every node, is short, and the long field is available for INFORMATION when the node represents an elementary item.

<u>Algorithm 3</u>: Move cursor (T,H) to the right in a ⊕-list. A is an auxiliary variable.

```
if  $H \neq 0$  then
    { $A \leftarrow H$ ;
      $H \leftarrow T \oplus \mathrm{NEXT}(H)$ ;
      $T \leftarrow A$ . ■
```

Algorithm 4: Insert node P into a circular ⊕-list to the right of a cursor (T,H) .

```
NEXT(P) ← H ⊕ T
if  $H = 0$  then  $T \leftarrow P$  else
    { NEXT(T) ← NEXT(T) ⊕ H ⊕ P ;
      NEXT(H) ← NEXT(H) ⊕ T ⊕ P .
H ← P . ■
```

Algorithm 5: Delete the node to the right of a cursor in a circular ⊕-list. TEM and C are auxiliary variables.

```
TEM ← H ⊕ T .
if TEM = 0 then
    { if  H = 0  then  Underflow
      else  C ← T ← 0
else
    { C ← T ⊕ NEXT(H) ;
      NEXT(C) ← NEXT(C) ⊕ TEM;
      NEXT(T) ← NEXT(T) ⊕ H ⊕ C .
Return H to Available space.
H ← C . ■
```

Application

Although the space required for the NEXT-field of an ⊕-list is half that required for standard double linking (using two pointer fields), the requirement for a cursor reference to such a list is doubled. Moreover, shared references are volatile; a change on the structure at one cursor may render another meaningless. Such a problem occurs when Algorithm 4 or 5 is performed on one of two initially identical cursors. These problems, aggravated by the increased time needed to alter a reference, have stigmatized the ⊕-list as a curiosity of computer lore, useless for practical applications.

That brand is not entirely appropriate. The ⊕-list is used as the basic data structure in the VW text editor [6] which is remarkably versatile for its size. Text and user commands are represented in VW as lists of ANSI-coded characters, one character per node. The text list is only a "window" on the file being edited, with the remainder on input and output files linked through extremal nodes distinguished by ANSI (communication control) characters barred in user text. Because there are no sublists, the single NEXT field is the only overhead in each node besides the seven-bit character.

Every cursor points to an edge <u>between</u> two nodes, and all editing operations can be run forwards or backwards from the initial cursor. The bi-directionality of all commands yields much of the power of VW and, therefore, the ease with which that feature is implemented through ⊕-lists was most surprising. When an operation is to be run in a reversed direction, the initial cursor need only be momentarily reversed!

<u>Algorithm 6</u>: Reverse the ⊕-list with reference (T,H) .

```
        T ↔ H  . ■
```

If double pointers were used for the double linking then a direction toggle would be tested each time a move was made selecting which pointer to use. With ⊕-lists, however, the commutativity of the exclusive-or operation allows the order of the initial cursor to control the direction of each move and implies a similar order for the derived cursor. Therefore, the time and code overhead required for computing succeeding cursors for ⊕-lists is offset by the required test of the direction toggle under standard double linking.

The multiple reference problem is resolved by checking all endangered cursors whenever the data structure is altered. There are only seven cursors in the system, of which at most two may be side-effected by an edge change, so these are always checked after critical operations.

The ⊕-list is a successful data structure for VW because of the space savings achieved. There are 2^{11} nodes but only about 2^3 cursors for a single user. The space allotment could increase more than sixty per cent (depending on hardware) in providing standard double linking.

Therefore, ⊕-lists can be practical when there are many nodes but only a few cursors in a dynamic data structure. Checking all side-effectable cursors after any change in the structure guarantees its integrity, and the limiting of double-pointers assures the reduction of space.

One other data structure for which the ⊕ trick can be useful is the ordered tree. Each ⊕-list might be restricted to have only one cursor, namely the list reference, so that changes to that list would side-effect no other cursors. With overlaps thereby prohibited the resulting structure is an ordered tree, with a father node pointing to an edge between two of its sons which can be changed safely.

Conclusion

Edge references are more powerful than node references into a data structure, and they cost surprisingly little. As references to lists they can often replace header cells or circular links, yielding better traversal algorithms because of simpler termination tests or new possibilities for threading. As references into lists they form the foundation for $\oplus$-lists, which can be an efficient scheme of double-linking if space is critical for a structure which has few references, is static, or is an ordered tree.

Acknowledgement

A referee is responsible for some of the code-efficiency of Algorithm 2.

References

1. Knuth, D.E. The Art of Computer Programming, Vol. 1 (2nd edition), Addison Wesley, Reading, Mass., 1973.

2. McCarthy, J., et al. LISP 1.5 Programmer's Manual, MIT Press, Cambridge, Mass., 1962.

3. Collins, G.E. PM, a system for polynomial manipulation. Comm. ACM 9, 8 (August, 1966), 578-589.

4. Weizenbaum, J. Symmetric list processor. Comm. ACM 6, 9 (September, 1963), 524-544.

5. Siklóssy, L. Fast read-only algorithms for traversing trees without an auxiliary stack. Information Processing Letters 1, 4 (June, 1972), 149-152.

6. Vitulli, N. VW: a small but potent machine-independent text editor. M.S. dissertation, Indiana University, in preparation.

Typed by Christopher Charles

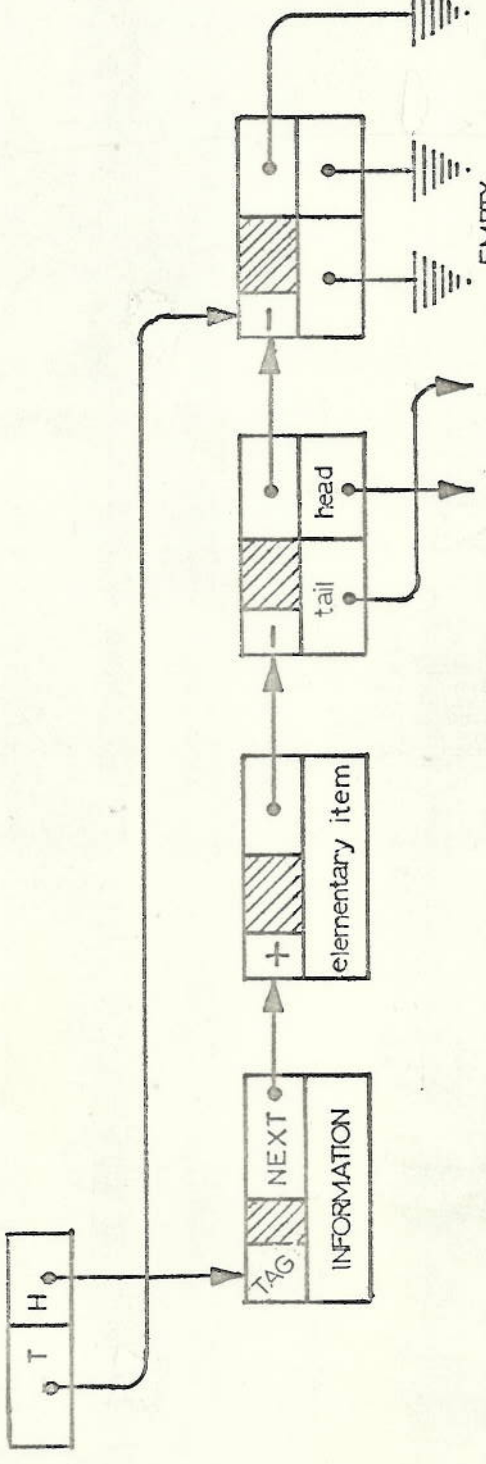


Figure 1. A pointer to a list.

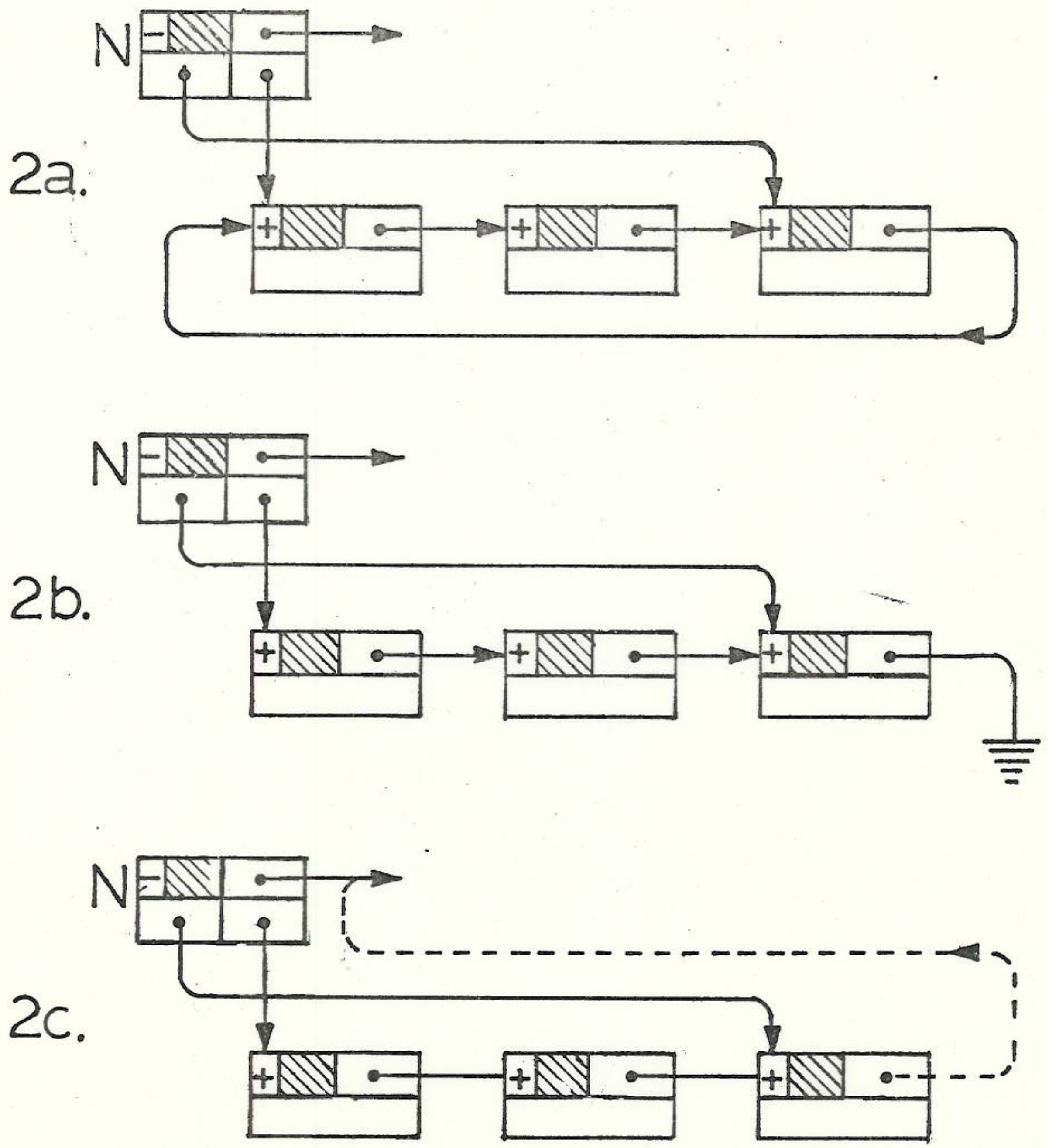


Figure 2. The circular sublist from node N .

a.) Standard.

b.) Circular link broken but restorable.

c.) Temporary threading as part of a traversal algorithm.

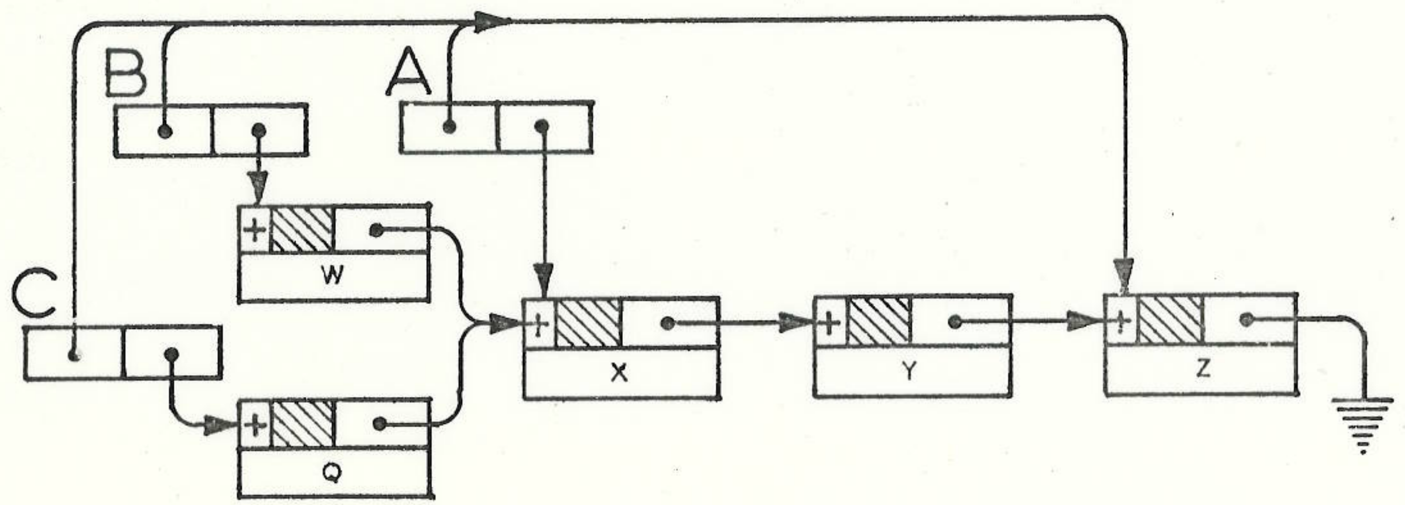


Figure 3. The lists referenced by B and C share the list referenced by A as a suffix.

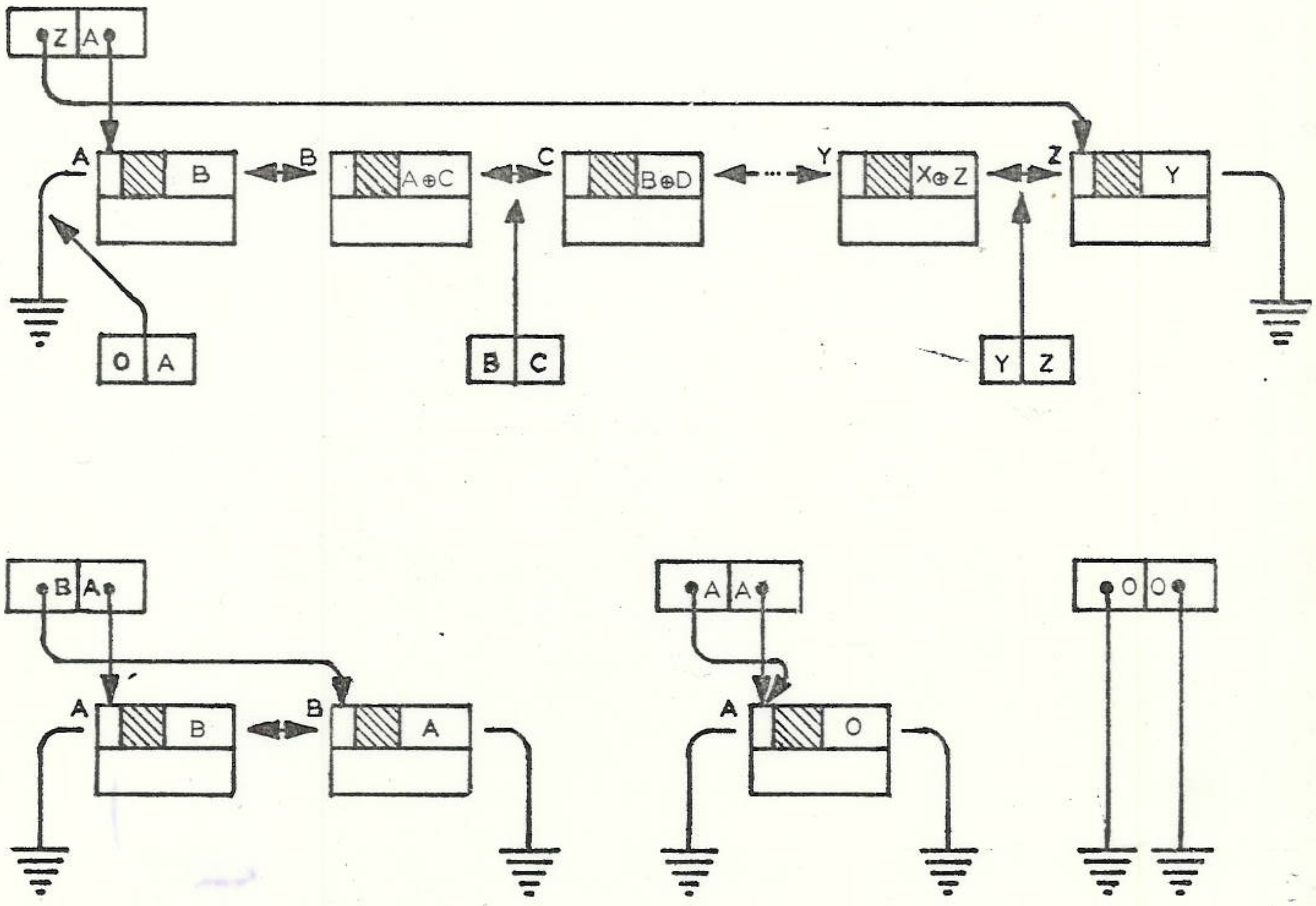


Figure 4. References to two-way ⊕-lists of 26 (with cursors), 2, 1, and 0 elements. End links are null, as in Figure 2b. Algorithms 4 and 5, stated for circular lists, can be modified to handle these structures, or these can easily be made circular.