Pace University
DigitalCommons@Pace

CSIS Technical Reports
Ivan G. Seidenberg School of Computer Science and Information Systems

7-1-1996

A general implementation of sequence.

C.T. Zahn

Follow this and additional works at: https://digitalcommons.pace.edu/csis_tech_reports

Recommended Citation
Zahn, C.T., "A general implementation of sequence." (1996). *CSIS Technical Reports*. 126.
https://digitalcommons.pace.edu/csis_tech_reports/126

This Thesis is brought to you for free and open access by the Ivan G. Seidenberg School of Computer Science and Information Systems at DigitalCommons@Pace. It has been accepted for inclusion in CSIS Technical Reports by an authorized administrator of DigitalCommons@Pace. For more information, please contact nmcguire@pace.edu.

SCHOOL OF COMPUTER SCIENCE
AND INFORMATION SYSTEMS

TECHNICAL REPORT
Number 98, Early July 1996

A General Implementation of Sequence

C. T. Zahn

Department of Computer Science
Pace University
One Martine Avenue
White Plains, NY 10606-1909

E-Mail: Zahn@PaceVM.dac.Pace.edu

PACE
UNIVERSITY

The current manuscript was presented in May, 1989, at the International Conference on Computing and Information (ICCI' 89) held in Toronto, Ontario, and printed in the *Proceedings of the 1989 International Conference on Computing and Information* published by the Canadian Scholars' Press.

Charles T. Zahn, Professor of Computer Science at Pace University, was the last Chairperson of Pace-Westchester's CS Department. Among his publications, which number over fifteen, is C Notes, which was one of the earliest guides to programming in C.

His academic background includes eleven years at Stanford University as a researcher in computer science specializing in pattern recognition, graph theory, programming methodology, and language design. While there he was also a lecturer in the Computer Science Department. In addition, Professor Zahn spent two years as a visiting scientist at the CERN Laboratory in Geneva.

His industrial background includes work for the General Electric Company, Yourdon Inc, the Mobil Corporation, and Advanced Computer Techniques as well as consulting for corporations and public agencies too numerous to name.

A GENERAL IMPLEMENTATION OF SEQUENCE

C. T. Zahn

Computer Science
Pace University
1 Martine Ave.
White Plains, NY 10606
(914) 681-4191

Abstract

We propose an abstract view of sequences of objects which features contexts and allows an easy description of most manipulations of sequences, such as insertions, deletions and context movements. Contexts refer to positions between sequence elements and are thus more powerful than pointers. An implementation of sequences is presented which is faithful to this abstraction and whose symmetry and elegance lead to simple and efficient algorithms. Language features suitable for inclusion in a high level language are discussed.

0. Introduction

Many algorithms manipulate sequences of data objects in a dynamic fashion, including deletions and/or insertions at various positions in the sequence. Text files are sequences of lines which are themselves sequences of characters, so most text processing operations fall into this category. Linked allocation methods using pointers going forward or both ways (doubly-linked) are typically used for this kind of sequence manipulation.

Unfortunately, one-way linked lists present problems since backing up is not possible, and "search and insert before" requires an extra pointer or unnatural looking logic. Doubly-linked lists solve this problem but require twice as much pointer space. In addition, the traditional algorithms require ugly special-case analysis for modifications at the ends of a sequence and careful handling of the empty sequence. Insertions are not made at an element of the sequence, but rather at a position between two elements, before the first, or after the last.

These considerations suggest the need for a general implementation of sequence with context variables pointing to positions, not to elements. There is an elegant method of unknown origin which uses the bitwise exclusive or to achieve two-way linked lists with a single link field. Nothing comes for free and in this case the extra cost is that pointers into the sequence must record the addresses of two adjacent elements. But this is exactly what we have called a context!

In section 1. we present an abstract view of sequences and in section 2. an implementation via Doubly-linked Circular Lists with Headers, DCLH, using the exclusive or trick. In section 3. we review the important algebraic properties of exclusive or and give the crucial invariant property governing DCLH sequences. A simple theorem is proved which illustrates the power of DCLH symmetry --- reversing a sequence costs one swap!

Sections 4. and 5. present the basic operations and several second-level operations based on the earlier ones. A few basic operations are illustrated by pseudocode and some access restrictions are suggested to protect the DCLH data structure from improper modifications. The normal notions of modularity and information hiding provide guidance here. Dijkstra's two-headed stack operations appear in our second level as well as linear search with a sentinel.

We suggest some Pascal-like language extensions in section 6. They include sequence and context declarations along with syntax for the various operations. Finally, section 7. summarizes what we think are the advantages of the DCLH implementation of sequence.

1. Abstract View of a Sequence

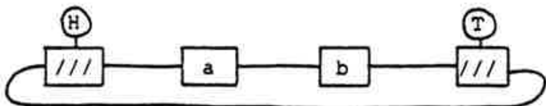
The sequence of three characters, a b c, can be thought of as a seven element series, .a.b.c. , in which even numbered positions, 0,2,4 ... , hold periods indicating a position between elements of the sequence. Odd numbered positions, 1,3,5 ... , contain the successive characters of the sequence. One can also visualize two boundary symbols, H and T (head and tail), so that each period is between two symbols. A single letter, a, would appear as H.a.T and the empty sequence as H.T .

To record a particular position in a sequence we replace a period by a caret symbol, "^", and call the caret a context in the sequence. For example, H.a.b^c.T represents the character sequence, a b c , with the position between b and c singled out. Moving a context forward or backward is well-defined except when inhibited by a boundary element, H or T.

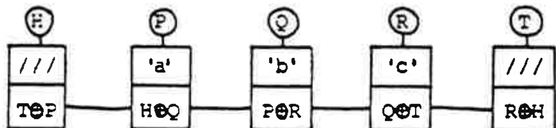
All conceivable operations on sequences can be naturally described in terms of such sequences and contexts. Any insertion is at a context and any contiguous subsequence lies between two contexts. In the following sections we describe an implementation that is faithful to this view of general sequences. Note that most text files can be viewed as sequences of sequences of characters.

2. Doubly-linked Circular Lists with Headers

In the DCLH implementation of a sequence there is a small node (e.g. Pascal record) to represent each symbol of the sequence including the head and tail boundary symbols. These nodes are formed into a doubly-linked circular list structure so H.a.b.T is represented by



The first field of each node is for data and the second field holds a link value allowing forward and backward traversals of the sequence via one link field. This is done by storing in the link field the exclusive or of the addresses of the previous and following nodes. This technique is part of the lore of programming [1,2]. Each reference to a position in the sequence must now be represented by the addresses of its left and right neighboring elements, and a context variable holds a two-field position value corresponding to a period. For example, H.a.b^c.T is a chain



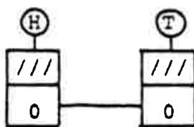
and the context, b^c , is represented by (left,right) with values (Q,R). The context immediately following this one is c^T which can be calculated from b^c by

$$\text{left}(c\hat{}T) = \text{right}(b\hat{}c) = R$$
$$\text{right}(c\hat{}T) = Q \oplus \text{link}(\text{right}(b\hat{}c)) = T$$

since $\text{link}(\text{right}(b\hat{}c)) = \text{link}(R) = Q \oplus T$
and $Q \oplus (Q \oplus T) = (Q \oplus Q) \oplus T = T$ as shown in the next section.

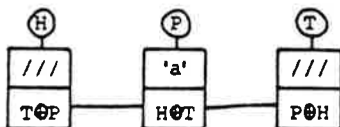
The algebraic properties of the exclusive or operator, $\oplus$, assure that, given the addresses of any two adjacent nodes, the previous or following node address can be recovered by an exclusive or of one of the addresses with the link field of the node referenced by the second address.

The empty sequence is represented as



since each has the other as both its left and right neighbor and $H \oplus H = T \oplus T = 0$.

The singleton sequence looks like



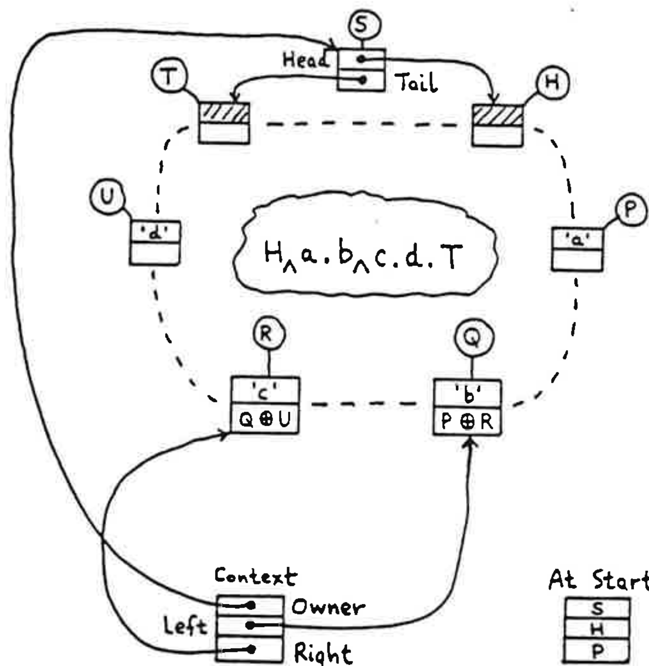
and, since there are more than two nodes in the cycle, no links are zero! ($A \oplus B = 0$ if and only if $A = B$)

Without the double headers (head and tail) one cannot distinguish empty from singleton (we tried it and failed). With the proposed structure, a zero link in the head or tail node implies empty and otherwise (T,H) represents a context from which the first or last element can be found by

$$\text{first} = T \oplus \text{link}(H) = T \oplus T \oplus P = P$$
$$\text{last} = H \oplus \text{link}(T) = H \oplus P \oplus H = P$$

To complete the structure each sequence needs a record containing the tail and head pointers (tantamount to a standard context). Each context will contain a third field referring to the owner sequence, so that context operations can check for incorrect situations at the boundaries. For example, it is illegal to advance past the context at the end of a sequence, c^T.

The following depicts a sequence with two contexts:



3. Invariant Properties of DCLH

The following properties of exclusive or ($\oplus$) are crucial for the DCLH implementation of sequences:

$$A \oplus (B \oplus C) = (A \oplus B) \oplus C$$
$$A \oplus B = B \oplus A$$
$$A \oplus 0 = A$$
$$A \oplus A = 0$$
$$A \oplus \text{new} = (A \oplus \text{old}) \oplus (\text{old} \oplus \text{new})$$

The last law of "substitution" says that to substitute new for old in the expression, $A \oplus$ old, one must simply cancel the old and incorporate the new by an exclusive or with (old $\oplus$ new).

If $H.a_1.a_2. \ldots a_k.T$ is a general sequence and we denote $H = a_0$ and $T = a_{k+1}$, then

link(a_j) = $a_{j-1} \oplus a_{j+1}$ where the subscript is modulo k+2. This holds for k=0 when, of course, we have link(H) = link(T) = 0.

As a corollary we have

$$a_{j+2} = a_j \oplus \text{link}(a_{j+1})$$
$$a_j = a_{j+2} \oplus \text{link}(a_{j+1})$$

In particular, we obtain the first and last elements of a sequence by

$$\text{first}(S) = T \oplus \text{link}(H)$$
$$\text{last}(S) = H \oplus \text{link}(T)$$

whenever link(H) or link(T) are non-zero.

We also can prove the following fairly amazing

Theorem:
Swapping the head and tail pointers in a sequence reverses the order of the sequence.

Proof:
From the previous remark that first(S) = T ⊕ link(H) and last(S) = H ⊕ link(T) we can see that the swap creates a new sequence, S', such that H' = T and T' = H. This implies

$$\begin{aligned}\text{first}(S') &= T' \oplus \text{link}(H') \\ &= H \oplus \text{link}(T) \\ &= \text{last}(S)\end{aligned}$$

and similarly

$$\text{last}(S') = \text{first}(S).$$

The symmetry of the exclusive or operator and the fact that no links have been changed assures that forward traversal of S' will mirror a backward traversal of S. Note that swapping the left and right field of a context has the effect of interchanging forward and backward, but the boundary tests are not quite correct.

4. Basic Operations

The basic low-level operations on DCLH sequences are

NewSeq --- to create a new empty sequence.
AtStart, AtFinish --- to determine if a context is at a boundary.
Advance, Retreat --- to move a context.
Insert --- to add a node at a context.
DelAft, DelBef --- to delete the node after or before a given context.
SetStart, SetFinish --- to initialize a context.
Clear --- to empty a sequence.
Empty --- to determine if a sequence is empty.

The following pseudocode is representative of the basic operations:

```
DelAft( C, P ):
   { C a context, P a node pointer }
   with C do
      if Right = Tail(Owner) then
         SequenceError
      else
         P, Link(Left), Link(Nbr), Right ←
            Right, Link(Left) ⊕ Right ⊕ Nbr,
            Link(Nbr) ⊕ Right ⊕ Left, Nbr
            where Nbr = Left ⊕ Link(Right)
```

```
SetStart( S, C ):
   { S a sequence, C a context }
   with C do
      Owner, Left, Right ←
         S, Head(S), Tail(S) ⊕ Link(Head(S))

Insert( P, C ):
   { P a node pointer, C a context }
   with C do
      Link(Left), Link(Right),
      Link(P), Right ←
         Link(Left) ⊕ Right ⊕ P,
         Link(Right) ⊕ Left ⊕ P,
         Left ⊕ Right, P
```

The user of sequences implemented as DCLH should be allowed the following access privileges:

1. Use Left and Right of a context.
2. Use and modify Data field of a node.
3. Copy from one context to another.

but with the following restrictions:

1. No access to Head or Tail of a sequence.
2. No access to Link field of a node.
3. No access to Data field in Head or Tail.

The user is expected to move contexts in a sequence and interrogate as well as manipulate sequence information by expressions of the form

```
Data(Right( C ))
```

when C is not at the end of its sequence.

In a modular language like Ada or Modula-2 these twelve procedures would be packaged into a module while hiding the actual DCLH implementation details. The user should declare sequence and context variables whose inner structure is private to the module. To implement a strict version of these hiding rules would require the addition of three new procedures:

NewVal --- to create a new node with given value.
ValAft, ValBef --- to deliver values in nodes.

5. Second Level Operations

At the next level we add the operations:

FindFirst, FindLast --- to find the first, last element of a sequence that is equal to a given value. Both use a sentinel search employing the Data field of headers.
Append --- to concatenate two sequences.
Split --- to separate a sequence into two sequences at a context.

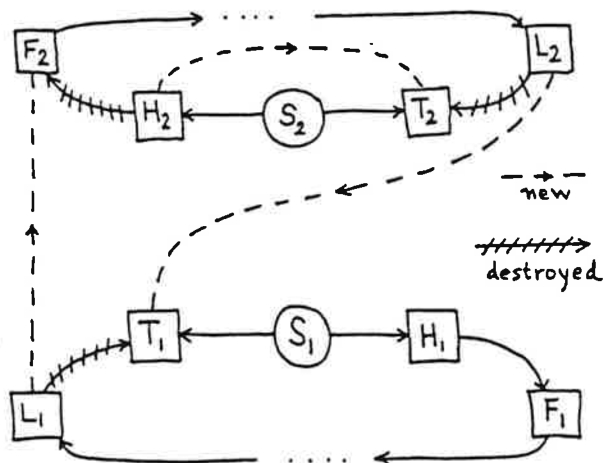
and the operations suggested by Dijkstra [3]:

LoPush, LoPop --- stack operations at head.
HiPush, HiPop --- stack operations at tail.

Other operations we have coded are:

PrintSeq --- to print the data in a sequence.
Strip --- to delete from a sequence all elements of a given value.

It is instructive to examine Append in detail since it is very efficient with the DCLH scheme, but there are some subtleties. The following diagram shows the before and after situations associated with Append(S1, S2):



The pseudocode is:

```
      Append( S1, S2 ):
        { S2 to be concatenated to S1 }
        if not Empty( S2 ) then
           Link(Last1), Link(First2),
           Link(Last2), Link(Tail(S1))  ⊕ ←
              Tail(S1) ⊕ First2,
              Head(S2) ⊕ Last1,
              Tail(S2) ⊕ Tail(S1),
              Last1 ⊕ Last2
           where Last1 = Head(S1)⊕Link(Tail(S1)),
Tail ———→     First2 = Head(S2) ⊕ Link(Head(S2)),
              Last2 = Head(S2) ⊕ Link(Tail(S2))
           Link(Head(S2)),
           Link(Tail(S2)) ← nil, nil
```

Referring to the diagram, we must perform surgery in two places. The last element of S1 must be followed by the first element of S2 and the last element of S2 must be followed by the tail element of S1. Each of these requires the alteration of two link fields by use of the substitution law of exclusive or. Finally, to keep things clean, the sequence S2 should be made empty with no dangerous links to its old contents.

6. Language Features using DCLH Sequences

It would be convenient if this elegant data structure were made available in higher level languages. We propose one set of language features that might be appropriate for a language similar to Pascal.

The programmer would declare variables:

```
var  S : sequence of ItemType;
     C : context in S;
     I : ItemType;
```

and could then manipulate sequences as follows:

```
S := null;    (* to clear S *)
C at ^S;      (* to set C to start of S *)
C at S^;      (*  ...  to end of S *)
C := I;       (* to insert I at C in S *)
I from C^;    (* move item after C to I *)
C ++;         (* advance C *)
C --;         (* retreat C *)
if C = ^S     (* is C at start of S? *)
```

Sequential file operations can be neatly described with such linguistic features as was demonstrated by the axiomatization of Pascal files by Hoare and Wirth [4].

7. Summary of Advantages

We feel that the DCLH implementation of sequences has several important advantages:

Elimination of Special Case Analyses --- Most techniques for linked lists require special checks at the beginning or at the end of a list, as well as extra checking to make sure the list is not empty. So-called header nodes are known to help in this regard and have been employed here to good effect.

Double Links at Single Cost --- The use of the properties of exclusive or to allow a single link to provide traversals in both directions is in many cases a good trade-off since nodes (where we save) are more frequent than contexts (where we must pay extra).

Between-Element Markers --- We suspect that many errors in text processing and the use of text editors may be caused by the absence of these contexts. Insertions are made, not at elements of a sequence, but between them, at the beginning, or at the end. Furthermore, any contiguous subsequence can be defined by two contexts. This view even allows the replacement of an empty subsequence by another sequence without special case analysis!

Immediate Reversal --- The incredibly efficient swap to effect sequence reversal may be important in some applications, although the ease of traversal in either direction may make reversal unnecessary.

Sentinel Searches --- The exploitation of otherwise unused data fields in the head and tail nodes to simplify linear searches using the sentinel technique is an extra benefit from the use of headers.

Symmetry --- The symmetry of the DCLH data structure is undoubtedly the main force behind the elegance and simplicity of our algorithms.

References

[1] Knuth, D. E., The Art of Computer Programming, Vol. 1, Fundamental Algorithms, section 2.2.4, exercise 18, Addison-Wesley, 1968.

[2] Standish, T. A., Data Structure Techniques, p. 197, Addison-Wesley, 1980.

[3] Dijkstra, E. W., A Discipline of Programming Prentice-Hall, 1976.

[4] Hoare, C. A. R. and Wirth, N., "An Axiomatic Definition of the Programming Language Pascal", Acta Informatica, 2, pp. 335-355, 1973.

≈ *Supplement* ≈

Pascal Implementation of DCLH Sequences

```
{$B-,D+,F-,I+,L+,N-,R+,S+,T+,V+}
program DCLH;
(* Doubly linked Circular Lists with Headers *)
   uses CRT;
   var Screen : text;

   type NodeRef = ^NodeType;
        SeqRef = ^SeqType;
        NodeType = record
                      Data : char;
                      Link : NodeRef
                   end;
        SeqType = record
                     Head, Tail : NodeRef
                  end;
        Context = record
                     Owner : SeqRef;
                     Left, Right : NodeRef
                  end;

   procedure SeqErr( N : integer );
   begin
      writeln( '****** Sequence Error #', N )
   end;

   function ExclOr( Q, R : NodeRef ) : NodeRef;
   (* required by Intel segments, far pointers *)
   begin
      ExclOr := Ptr( Seg(Q^) xor Seg(R^),
                     Ofs(Q^) xor Ofs(R^) )
   end;  (* ExclOr *)

   procedure NewSeq( var S : SeqRef );
   begin  (* should be done only once *)
      new(S);
      with S^ do
         begin
            new(Head); Head^.Link := nil;
            new(Tail); Tail^.Link := nil
         end
   end;  (* NewSeq *)

   function AtStart( C : Context ) : boolean;
   begin
      AtStart := ( C.Left = C.Owner^.Head )
   end;  (* AtStart *)

   function AtFinish( C : Context ) : boolean;
   begin
      AtFinish := ( C.Right = C.Owner^.Tail )
   end;  (* AtFinish *)
```

```
function Empty( S : SeqRef ) : boolean;
begin
   Empty := ( S^.Head^.Link = nil )
end;  (* Empty *)

procedure Advance( var C : Context );
   var temp : NodeRef;
begin
   if not AtFinish(C) then with C do
      begin
         temp := Left;
         Left := Right;
         Right := ExclOr(temp, Right^.Link)
      end
end;  (* Advance *)

procedure Retreat( var C : Context );
   var temp : NodeRef;
begin
   if not AtStart(C) then with C do
      begin
         temp := Left;
         Left := ExclOr( Right, Left^.Link );
         Right := temp
      end
end;  (* Retreat *)

procedure Insert( P : NodeRef; var C : Context );
begin
   with C do
      begin
         Left^.Link := ExclOr( Left^.Link,
                              ExclOr( Right, P ) );
         Right^.Link := ExclOr( Right^.Link,
                               ExclOr( Left, P ) );
         P^.Link := ExclOr( Left, Right );
         Right := P
      end
end;  (* Insert *)

procedure DelAft( var C : Context;
                  var P : NodeRef );
   var Nbr : NodeRef;
begin
   if C.Right = C.Owner^.Tail then
      SeqErr(2)
   else
      with C do
         begin
            Nbr := ExclOr( Left, Right^.Link );
            P := Right;
            Left^.Link := ExclOr( Left^.Link,
                                 ExclOr( Right, Nbr ) );
            Nbr^.Link := ExclOr( Nbr^.Link,
                                ExclOr( Right, Left ) );
            Right := Nbr
         end
end;  (* DelAft *)
```

```
procedure DelBef( var C : Context;
                  var P : NodeRef );
   var Nbr : NodeRef;
begin
   if C.Left = C.Owner^.Head then
      SeqErr(3)
   else
      with C do
         begin
            Nbr := ExclOr( Right, Left^.Link );
            P := Left;
            Right^.Link := ExclOr( Right^.Link,
                                   ExclOr( Left, Nbr ) );
            Nbr^.Link := ExclOr( Nbr^.Link,
                                 ExclOr( Left, Right ) );
            Left := Nbr
         end
end;  (* DelBef *)

procedure SetStart( S : SeqRef; var C : Context );
begin
   with C, S^ do
      begin
         Owner := S;
         Left := Head;
         Right := ExclOr( Tail, Head^.Link )
      end
end;  (* SetStart *)

procedure SetFinish( S : SeqRef;
                     var C : Context );
begin
   with C, S^ do
      begin
         Owner := S;
         Left := ExclOr( Head, Tail^.Link );
         Right := Tail
      end
end;  (* SetFinish *)

procedure Clear( S : SeqRef );
   var C : Context;  P : NodeRef;
begin
   SetStart(S,C);
   while not Empty(S) do
      begin DelAft(C,P);  dispose(P) end
end;  (* Clear *)

procedure FindFirst( S : SeqRef; X : char;
                     var C : Context );
begin
   S^.Tail^.Data := X;   (* sentinel for search *)
   SetStart(S,C);
   while C.Right^.Data <> X do
      Advance(C)
end;  (* FindFirst *)

procedure FindLast( S : SeqRef; X : char;
                    var C : Context );
begin
   S^.Head^.Data := X;   (* sentinel for search *)
   SetFinish(S,C);
   while C.Left^.Data <> X do Retreat(C)
end;  (* FindLast *)
```

```
procedure Append( var S1, S2 : SeqRef );
   var Last1, First2, Last2 : NodeRef;
begin
   if not Empty(S2) then
      begin
         Last1 := ExclOr( S1^.Head,
                          S1^.Tail^.Link );
         First2 := ExclOr( S2^.Tail,
                           S2^.Head^.Link );
         Last2 := ExclOr( S2^.Head,
                          S2^.Tail^.Link );
         Last1^.Link := ExclOr( Last1^.Link,
                        ExclOr( S1^.Tail, First2 ));
         First2^.Link := ExclOr( First2^.Link,
                         ExclOr( S2^.Head, Last1 ));
         Last2^.Link := ExclOr( Last2^.Link,
                        ExclOr( S2^.Tail, S1^.Tail ));
         S1^.Tail^.Link := ExclOr( S1^.Tail^.Link,
                           ExclOr( Last1, Last2 ));
         with S2^ do
            begin
               Head^.Link := nil;
               Tail^.Link := nil
            end
      end
end;  (* Append *)

procedure Split( C : Context;
                 var S1, S2 : SeqRef );
   var First1, Last1, First2, Last2 : NodeRef;
begin
   (* assumes S1, S2 have Head, Tail
    * and are Empty
    *)
   if (S1 = nil) or (S2 = nil) then
      SeqErr(4)
   else if not Empty(S1) or not Empty(S2) then
      SeqErr(5)
   else  (* okay to proceed with split *)
      begin
         with C, Owner^ do
            begin
               First1 := ExclOr( Tail,
                                 Head^.Link );
               Last1 := Left;
               First2 := Right;
               Last2 := ExclOr( Head,
                                Tail^.Link );
            end;
         with S1^ do
            begin
               Head^.Link := ExclOr( Tail,
                                     First1 );
               First1^.Link :=
                  ExclOr( First1^.Link,
                  ExclOr( C.Owner^.Head, Head ));
               Tail^.Link :=
                  ExclOr( Head, Last1 );
               Last1^.Link :=
                  ExclOr( Last1^.Link,
                  ExclOr( C.Right, Tail ))
            end;
```

```
          with S2^ do
             begin
                Head^.Link :=
                   ExclOr( Tail, First2 );
                First2^.Link :=
                   ExclOr( First2^.Link,
                   ExclOr( C.Left, Head ));
                Tail^.Link :=
                   ExclOr( Head, Last2 );
                Last2^.Link :=
                   ExclOr( Last2^.Link,
                   ExclOr( C.Owner^.Tail, Tail ))
             end;
          with C.Owner^ do
             begin
                Head^.Link := nil;
                Tail^.Link := nil
             end
       end
end;  (* Split *)

procedure LoPush( var S : SeqRef; X : char );
   var C : Context;  P : NodeRef;
begin
   SetStart(S,C);
   new(P);  P^.Data := X;
   Insert(P,C)
end;  (* LoPush *)

procedure LoPop( var S : SeqRef; var X : char );
   var C : Context;  P : NodeRef;
begin
   if Empty(S) then SeqErr(1)
   else
      begin
         SetStart(S,C);
         DelAft(C,P);  X := P^.Data;
         dispose(P)
      end
end;  (* LoPop *)

procedure HiPush( var S : SeqRef; X : char );
   var C : Context;  P : NodeRef;
begin
   SetFinish(S,C);
   new(P);  P^.Data := X;
   Insert(P,C)
end;  (* HiPush *)

procedure HiPop( var S : SeqRef; var X : char );
   var C : Context;  P : NodeRef;
begin
   if Empty(S) then SeqErr(1)
   else
      begin
         SetFinish(S,C);
         DelBef(C,P);  X := P^.Data;
         dispose(P)
      end
end;  (* HiPop *)
```

```
procedure PrintSeq( S : SeqRef );
   var C : Context;  X : char;
begin
   SetStart(S,C);
   while not AtFinish(C) do
      begin
         write(C.Right^.Data);
         Advance(C)
      end;
   writeln
end;  (* PrintSeq *)

procedure Strip( var S : SeqRef; X : char );
   var C : Context;  P : NodeRef;
begin
   SetStart(S,C);
   while not AtFinish(C) do
      if C.Right^.Data = X then
         begin
            DelAft(C,P);
            dispose(P)
         end
      else
         Advance(C)
end;  (* Strip *)

procedure Test;
   var S, S1, S2, T : SeqRef;
       C : Context;
       P : NodeRef;
       Ch : char;
begin
   NewSeq(S);
   for Ch := 'A' to 'Z' do HiPush(S,Ch);
   SetStart(S,C);
   while C.Right^.Data < 'M' do
      begin
         write( C.Right^.Data );
         C.Right^.Data := 'A';
         Advance(C)
      end;
   writeln;
   (* Data = 'M' *)
   FindLast(S, 'A', C);
   (* C.Left^.Data is last 'A' *)

   NewSeq(S1);  NewSeq(S2);
   Split(C,S1,S2);
   SetStart(S1,C);
   while not AtFinish(C) do
      begin
         write(C.Right^.Data);
         Advance(C)
      end;
   writeln;
   SetStart(S2,C);
   while not AtFinish(C) do
      begin
         write(C.Right^.Data);
         Advance(C)
      end;
```

```
      writeln;

      SetFinish(S2,C);
         while not AtStart(C) do
            begin
               write(C.Left^.Data);
               DelBef(C,P);  dispose(P)
            end;
         write( ' ':10 );
      SetStart(S1,C);
      while not AtFinish(C) do
         begin
            write(C.Right^.Data);
            DelAft(C,P);  dispose(P)
         end;
      writeln;

      NewSeq(T);
      for Ch := 'A' to 'Z' do LoPush(T,Ch);
      SetFinish(T,C);
      while C.Left^.Data < 'N' do
         begin
            C.Left^.Data := '.';
            Retreat(C)
         end;
      FindFirst(T, '.', C);
      Split(C,S1,S2);
      Clear(S);
      while not Empty(S1) do
         begin
            LoPop(S1,Ch);  HiPush(S,Ch);
            LoPop(S2,Ch);  HiPush(S,Ch)
         end;
      PrintSeq(S);
      Strip( S, '.' );
      PrintSeq(S);
      writeln( 'Got to the end.' )
   end;  (* Test *)

begin  (* main code just calls Test *)
   AssignCrt( Screen );  rewrite( Screen );
   Assign( input, '' );  reset( input );
   Assign( output, '' );  rewrite( output );

   Test;

   Close( Screen );
end.
```

program output from Test

```
ABCDEFGHIJKL
AAAAAAAAAAAA
MNOPQRSTUVWXYZ
ZYXWVUTSRQPONM          AAAAAAAAAAAA
Z.Y.X.W.V.U.T.S.R.Q.P.O.N.
ZYXWVUTSRQPON
Got to the end.
```

From: Programming on Purpose, Volume II (pages 12-13)
by P. J. Plauger
PTR Prentice-Hall, 1993

A much more clever use of the exclusive-OR is storing two pointers in a storage cell large enough to hold only one. I believe this is one of the exercises in Knuth's *The Art of Computer Programming* (**Knu68**). You say you can't do that? Watch.

Let's say you have a list of data elements that can be very long, and that you need to scan either backwards or forwards. The usual technique is to declare each data element as a structure that contains both backward and forward pointers. So if **p** points to the current element (again speaking C), **p->left** designates the element to the left, and **p->right** designates the element to the right.

If you feel you can't afford to set aside space for two pointers within the structure, what you do instead is set aside a single integer large enough to hold all the bits of a pointer. (Yes, I know there are implementations of C that may require two or more long integers to represent a pointer. And I know that converting between integer and pointer representations can cause a change of representation. If you want maximum portability, you should write all this stuff with macros so you can localize the machine-dependent parts.) What you store in the integer is the exclusive-OR of the pointers to the left and right elements. Let's call that integer cell **link**, and assume it has some defined integer type **INT** that can represent all values of the type **PTR**, which is a pointer to a list element.

Instead of a pointer to a single list element (such as **p** above), you must now maintain pointers to two adjacent list elements. Let's say **pleft** points to the left element and **pright** points to its neighbor to the right. Then you can move your two-element window on the list to the left by writing:

```
ptemp = (PTR)(pleft->link ^ (INT)pright);
pright = pleft;
pleft = ptemp;
```

And you can move your two-element window to the right by writing:

```
ptemp = (PTR)(pright->link ^ (INT)pleft);
pleft = pright;
pright = ptemp;
```

It is a fun exercise to write full blown versions of these functions. You need to make them safe for lists with zero and one elements. You need to ensure that stepping left or right will not take you off the end of the list. And you need to add functions for adding and deleting elements. Try it.

You can extend this ingenious trick to two dimensions. Say you have to represent the grid points within an arbitrarily large contiguous blob on a plane. Again, the usual solution requires that each element have four pointers, for the neighbors you reach by going up, down, left, and right from the current element. You can replace these four pointers in each element by two integers, one for each axis. To walk the list, you must maintain four pointers, to adjacent elements that form a square. You advance in any direction by sliding the square about the plane.

To span three dimensions, you need to store only three integers within each data element, instead of six pointers. But you must maintain *eight* pointers to walk the list in any direction. As you go to higher dimensions, you can see the classic tradeoff between storing information and recomputing it as needed. The exclusive-OR trick lets you squeeze considerable redundancy out of your stored data, but at a cost in computation.