CHAPTER 13

LINKED INFORMATION

We have managed to get along fairly well to this point in the book by storing data directly in sequential arrays. Even in the most complex information structures we have used -- multi-dimensional arrays, stacks, and queues -- all a program needs to do in order to access the next element in the sequence is to increment an index. In many applications, however, it is simply not practical to maintain units of information in physically contiguous order. Sometimes the information might be constantly in flux, with new units being inserted and older units being deleted. Other times a programmer might desire to organize information according to several perspectives, only one of which may be reflected physically.

The present chapter examines structures in which each unit of information includes one or more elements called links which direct the program to one or more related units of information (note 1). Links are particularly useful because they allow a program to manipulate structures without fussing over the physical location of each unit. Linked information structures

have been in the repertory of computer programming techniques since the 1950's when Alan Newell, J.C. Shaw, and Hebert Simon conducted their groundbreaking researches into heuristic programming (reference?). They have acquired a mystique over the years due to the introduction of interpreters with implicit link-handling features (note 2) and to the accompanying myth that compiled languages such as FORTRAN are unsuited to linked structures. The truth is that such structures are quite simple to implement explicitly and what many implicit implementations impose restrictions which are unnecessary and can even ultimately hinder a programmer.

The basic unit of information in a linked structure is called a node or record. Each node contains some number of elements called fields, at least one of which serves as a link. Links are also called pointers, references, and addresses (note 3).

13.1 APPLICATION: A SIMPLE LINKED-LIST EDITOR

As an illustration of how linked data structures are manipulated, we shall show how to implement an editor for linked lists. A linked list is one of the simplest information

structures to use links. It has the following properties:

1. each node has exactly one link, and

2. no two links point to the same node.

The linked lists manipulated by our simple editor are characterized by two fields: a link and a value. There are five commands, each designated by a single upper case letter:

1. E - End editing session.

2. L - List the values of each node in order from the smallest to the largest.

3. I - Insert a new node. Entering this command evokes a prompt for a positive value; the value of a node in turn determines the node's position in the list.

4. D - Delete a current node. Entering this command also evokes a prompt for a value; the node in the list which has this value is then deleted.

5. P - Print diagnostics. This command prints out all of

the links and values as they occur in memory. It allows users to see how the program physically stores information.

Internally, the program actually manipulates two lists: a 'used' list gives those nodes whose values have been directly specified by the user, while a 'free' list keeps tabs on all the nodes not presently occuring in the "used" lists. Inserting a new node into the used list causes a node to be deleted from the free list, and vice versa. In addition to the links and values alloted to individual nodes, the editor also keep tracks of the "heads" of both lists, so that it may distinguish between the two.

13.1.1 Dramatization

Figure 13-1: Interacting with a linked-list editor - Responses to the "Command:" prompt were entered by the author.

Figure 13-1 dramatizes an editing session. Initially, the

```
Command: P
Head of used list:  0
Head of free list: 10
Index:  1   Free node      Link:  0
Index:  2   Free node      Link:  1
Index:  3   Free node      Link:  2
Index:  4   Free node      Link:  3
Index:  5   Free node      Link:  4
Index:  6   Free node      Link:  5
Index:  7   Free node      Link:  6
Index:  8   Free node      Link:  7
Index:  9   Free node      Link:  8
Index: 10   Free node      Link:  9

Command: I
Value:  3.10

Command: I
Value:  4.20

Command: I
Value:  5.90

Command: I
Value:  2.80

Command: I
Value:  9.00

Command: I
Value:  4.30

Command: I
Value:  8.20

Command: L
  2.80
  3.10
  4.20
  4.30
  5.90
  8.20
  9.00

Command: P
Head of used list:  7
Head of free list:  3
Index:  1   Free node      Link:  0
Index:  2   Free node      Link:  1
Index:  3   Free node      Link:  2
Index:  4   Value: 8.20    Link:  6
Index:  5   Value: 4.30    Link:  8
Index:  6   Value: 9.00    Link:  0
Index:  7   Value: 2.80    Link: 10
Index:  8   Value: 5.90    Link:  4
Index:  9   Value: 4.20    Link:  5
Index: 10   Value: 3.10    Link:  9

Command: D
Value:  5.90

Command: D
Value:  4.30

Command: D
Value:  8.20
```

```
Command: L
  2.80
  3.10
  4.20
  9.00

Command: P
Head of used list:  7
Head of free list:  4
Index:  1   Free node      Link:  0
Index:  2   Free node      Link:  1
Index:  3   Free node      Link:  2
Index:  4   Free node      Link:  5
Index:  5   Free node      Link:  8
Index:  6   Value: 9.00    Link:  0
Index:  7   Value: 2.80    Link: 10
Index:  8   Free node      Link:  3
Index:  9   Value: 4.20    Link:  6
Index: 10   Value: 3.10    Link:  9

Command: I
Value:  9.10

Command: I
Value:  6.70

Command: I
Value:  2.20

Command: I
Value:   .10

Command: L
   .10
  2.20
  2.80
  3.10
  4.20
  6.70
  9.00
  9.10

Command: P
Head of used list:  3
Head of free list:  2
Index:  1   Free node      Link:  0
Index:  2   Free node      Link:  1
Index:  3   Value:  .10    Link:  8
Index:  4   Value: 9.10    Link:  0
Index:  5   Value: 6.70    Link:  6
Index:  6   Value: 9.00    Link:  4
Index:  7   Value: 2.80    Link: 10
Index:  8   Value: 2.20    Link:  7
Index:  9   Value: 4.20    Link:  5
Index: 10   Value: 3.10    Link:  9

Command: E
```

Fig 13-1

used list is empty, so the head of this list is designated by the "null link", 0. By contrast, the free list embraces all available nodes; the head of the free list is designated as node 10, which provides a link to mode 9. Node 9 in turn provides a link to node 8, and so on down to node 1, whose null link proclaims the tail of the free list.

The next seven I commands serve to insert seven new nodes, after which an L command causes the editor to list the seven corresponding seven values in order. Typing a P command reveals that each insertion has caused a node to be transferred from the head of the free list to some position in the used list. Node 7 is now situated at the head of the used list; node 7 provides a link to node 10, which in turn provides a link to node 9, and in this manner the program proceeds through nodes 5, 8, 4, and 6, whose null link indicates the tail of the used list. Only three nodes remain in the free list: 3, 2, and 1.

The next three D commands serve to transfer three used nodes to the head of the free list. The used nodes now proceed from node 7 through nodes 10 and 9 to node 6, while the free nodes proceed from node 4 through nodes 5, 8, 3, and 2 to node 1.

13.1.2 Implementation

-- Programming example 13-1: program EDIT (2 pages) --

Program EDIT and its ancillary subroutines SEARCH, INSERT, and DELETE illustrate how a programmer might implement a linked-list editor in FORTRAN '77. EDIT stores values and links for each node in two parallel arrays, the real array VALUE and the integer array LINK, respectively. A single index suffices to access both of the fields for any given node. The parameter MLST determines the maximum number of nodes, while the two integer varibles USED and FREE keep track of the "heads" of the used and free lists, respectively.

The basic structure of EDIT is a large loop (lines 18-68) which begins each iteration by prompting the user for a command and then, through a sequence of conditional tests, interprets this command. These conditional blocks break down as follows:

1. E - (line 22) EDIT terminates the editing session.

```
      program EDIT
C
C     Simple linked-list editor
C
      parameter (MLST=10)
      real VALUE(MLST),V
      integer LINK(MLST),USED,FREE,OLD,NODE,NOW,NEXT,NEW
      character*1 CMD
      logical SUCCES
      data USED/0/
C
C     Initialize free list
      do (I=1,MLST)
        LINK(I) = I - 1
      repeat
      FREE = MLST
C
      do
        type 'Command: '
        read *,CMD
        if (CMD.eq.'E') stop
        if (CMD.eq.'L') then
          I = USED
          do
            if (I.eq.0) exit
            print *, VALUE(I)
            I = LINK(I)
          repeat
        else if (CMD.eq.'I') then
          type 'Value: '
          read *,V
          call SEARCH(VALUE,LINK,USED,NOW,NEXT,V,SUCCES)
          if (SUCCES) then
            print *, 'Value already present in list.'
          else
            call INSERT(LINK,USED,FREE,NOW,NEXT,NEW)
            VALUE(NEW) = V
          end if
        else if (CMD.eq.'D') then
          type 'Value: '
          read *,V
          call SEARCH(VALUE,LINK,USED,OLD,NODE,V,SUCCES)
          if (SUCCES) then
            VALUE(NODE) = 0.0
            call DELETE(LINK,USED,FREE,OLD,NODE)
          else
            print *, 'Value not present in list.'
          end if
        else if (CMD.eq.'P') then
          print *, 'Top of used list:',USED
          print *, 'Top of free list:',FREE
          do (I=1,MLST)
            if (VALUE(I).gt.0) then
              print *, 'Index:',I,'   Value:',VALUE(I),'   Link:',LINK(I)
            else
              print *, 'Index:',I,'   Free node     Link:',LINK(I)
            end if
          repeat
        else
          print *, 'Illegal command.'
        end if
      repeat
      end
```

```
      subroutine SEARCH(VALUE,LINK,USED,NOW,NEXT,ARG,SUCCES)
      real VALUE(1),ARG
      integer LINK(1),USED,NOW,NEXT
      logical SUCCES
      SUCCES = .false.
C     Check for empty list
      if (USED.eq.0) return
C     Examine each node until VALUE exceeds ARG
      NOW = 0
      NEXT = USED
      do
        V = VALUE(NEXT)
        if (ARG.eq.V) SUCCES = .true.
        if (ARG.le.V) return
        NOW = NEXT
        NEXT = LINK(NOW)
        if (NEXT.eq.0) return
      repeat
      end
```

Ex 13-1 (page 1 of 2)

```
      subroutine INSERT(LINK,USED,FREE,NOW,NEXT,NEW)
      integer LINK(1),USED,FREE,NOW,NEXT,NEW
C
C     Test for list overflow
      if (FREE.eq.0) then
        print *, 'List overflow.'
        return
      end if
C     Obtain node from free list
      NEW = FREE
      FREE = LINK(FREE)
C     Insert node into used list
      if (NOW.eq.0) then
        USED = NEW
      else
        LINK(NOW) = NEW
      end if
      LINK(NEW) = NEXT
      return
      end

      subroutine DELETE(LINK,USED,FREE,OLD,NODE)
      integer LINK(1),USED,FREE,OLD,NODE
C
C     Delete node from used list
      if (OLD.eq.0) then
        USED = LINK(NODE)
      else
        LINK(OLD) = LINK(NODE)
      end if
C     Append node to free list
      LINK(NODE) = FREE
      FREE = NODE
      return
      end
```

Ex 13-1 (page 2 of 2)

2. L - (lines 24-29) EDIT steps through the used nodes, printing each value.

3. I - (lines 31-40) EDIT prompts the user for a value V, then requests subroutine SEARCH to determine two consective nodes in the used list, NOW and NEXT, such that V lies between VALUE(NOW) and VALUE(NEXT). EDIT then asks subroutine INSERT to detach a record NEW from the free list and insert NEW into the used list between NOW and NEXT. Finally, EDIT stores V in VALUE(NEW).

4. D - (lines 45-53) EDIT prompts the user for a value V, then requests subroutine SEARCH to locate a node NODE in the used list, such that VALUE(NODE) matches V. SEARCH also returns the node immediately preceding NODE in the used list as the integer variable OLD. EDIT then asks subroutine DELETE to delete NODE from the used list and append it to the free list.

5. P - (lines 55-64) EDIT simply prints USED, FREE, and the elements of VALUE and LINK.

The implementation chosen for EDIT and its subroutines is by no means the only implementation conceivable. In

assembly-language programming it is conventional to store the fields of one record in consecutive locations of memory. Programmers can implement such storage in FORTRAN by declaring arrays with several consecutive elements per node (one element per field). This approach requires nodal indices to proceed in increments greater than one. When the number of fields in each node is constant, it will usually be convenient to declare one array for each field and to use EQUIVALENCE statements to specify relative offsets between arrays (note 4).

13.2 LINKED LISTS IN AUTOMATED COMPOSITION

Herbert Brun has developed a utility called SAWDUST (implemented by 1976; described in Blum, 1979) for manipulating aggregates constructed from the "smallest parts of waveforms". These "smallest parts", or "elements" are each described by two values: a magnitude and a number of samples. Elements are pieced together as step functions to create waveforms; waveforms are pieced together to create notes, notes are pieced into musical phrases, and so on until an entire work has been described.

Each aggregate is represented as a linked list, which at the

most elementary level (a single node) consist of simple elements. SAWDUST provides four operations:

1. LINK splices two items together into a larger item. At the most elementary level, LINK can be used to splice elements into waveforms; the same procedure suffices to splice notes and phrases.

2. MINGLE repeats an item a specified number of times by linking together the appropriate number of copies. This operation provides the basic mechanism for creating sustained notes from waveforms.

3. MERGE accepts two items and collates them element-by-element to derive a third item. For example, if if A=[e1,e2,e3] and B=[e4,e5,e6], then MERGE A,B creates a new item of the form [e1,e4,e2,e5,e3,e6].

4. VARY transmutes a waveform over a specified duration. In addition to waveform and duration, the user specifies 1) initial and final values for either amplitude or period of the waveform and 2) a number N. SAWDUST uses this number to construct a polynomial curve passing through these initial and final values at the

appropriate times (for example, if N=1 then SAWDUST will construct a straight line; if N=2, the curve will be parabolic, and so on). It then repeats the link over and over, altering amplitude or period in accordance with this curve.

Brun has employed SAWDUST to create a series of works, including Dust (1976), More Dust (1977), Dustiny (1978), A Mere Ripple (1979), U-Turn-To (1980), and I Told You So (1981).

Petr Kotik's utility composing with Markov chains (heading 6.1.4) had two components, a Markov-matrix editor and a chain generating program. The chain generating program closely resembled program CHAIN2 and subroutine MARKOV (heading 6.2) except that it accepted matricies with an arbitrary number of states, read in from a disk file. The editor allowed Kotik to describe and modify matricies, which he tested through the chain generator. Kotik tended to specify a large number of states; however, since he also tended to restrict the number of destinations available to individual states, it was possible to reclaim much of the memory by treating each N-state matrix as a set of N linked lists. Each node in a list therefore consisted of a link, a destination (the source being implicit), and a probabilistic weight.

13.3 MORE ELABORATE STRUCTURES

The simple linked lists described up to this point are easily organized into stacks and queues (heading 10.2); indeed, the free list employed by program EDIT functions precisely as a stack. Programmers can also generalize the notion of linking by allowing each node to have two or more links.

13.3.1 Multiply-Linked Lists

A fairly straightforward generalization addresses the problem that in the linked lists we have seen thus far, it is only possible to step through a list in one direction. If each node had a second link referring "backwards" to the node's predecessor, then it would be possible to step through a list in both directions. Such a list is called a doubly linked list.

Another application was mentioned at the opening of this chapter: organizing information according to several perspectives, only one of which may be reflected physically.

This problem may easily be solved by providing each item of information with one link for each criterion of organization. For example, a program might need to manipulate a collection of notes organized physically by starting times but characterized also 1) by three instrumental choirs: strings, horns, double reeds, and 2) by five registers: soprano, alto, tenor, bass, and double bass. This program could then employ two links per note: 1) an instrumental link would provide independent access to sequences of notes played by each of the different instrumental choirs (with three pointers giving the head of each list); 2) a registral link would access sequences of notes played in each register (with five pointers).

13.3.2 Trees

Trees are hierarchic data structures in which each node has the following properties:

1. each node has two or more links, and

2. no two links point to the same node.

however, trees do not seem really necessary for the process Fry describes.

13.5 RECOMMENDED READING

Knuth, Donald. "Information structures", chapter 2 of Fundamental Algorithms (Reading: Addison-Wesley, 1963).

Trees are an extremely elegant way of structuring information, but as yet have had only limited application in automated composition (note 5).

13.4 NOTES

1. The primary reference for linked information structure is Knuth's Fundamental Algorithms (1963).

2. LISP is the most prominent example.

3. The term "address" is confusing because programmers typically refer to the location of a node in memory as its "address", e.g., "The address field of a node contains the address of the next node."

4. Unfortunately, this practice cannot be used with many FORTRAN '77 compilers, which unlike earlier implementations of FORTRAN do not allow programmers to declare equivalences with subroutine parameters.

5. The structure described by Fry (1980) is instructive;