Advanced BASIC Part 2 — Chaining

Phil Cohen

Following the series on Back Door into BASIC, ETI has begun a series on Advanced BASIC for those who want to know where to go now. The first article of the series, 'Sorting', appeared in the June 1981 issue of ETI; this month Phil Cohen explains and illustrates 'chaining'. The language used is 8K Microsoft BASIC, and minor alterations will allow the included examples to run on almost any medium-sized personal computer system.

LET'S SAY you've got a list of items with a definite sequence — words in a word processor, or names in alphabetical order, for example. What's the best way to organise their storage in a program?

Easy, you say. Put them into an array.

Okay, but what if you want to add one item to the list, but *in the middle of the list*. What happens? One approach is to find out where the item goes and move all the items after it down one position (assuming, of course, there's room in the array). This becomes very slow if you have a lot of items in the list.

Another way of doing it is to have a separate list of 'amendments' to the existing list and to check it as the main list is read out. This is okay if the number of amendments is very small.

One solution to the problem is to use what is known as 'chaining'. Each item in the list 'points' to the next item.

Perhaps the easiest way to explain this is with an example. The list we want to represent is:

1.1, 2.2, 3.3, 4.4

This is stored in the array A(5,2). The 5 gives us room to add one more item to the list and the 2 allows the storage of 1) the data, and 2) the location of the next item:

A(1,1) = 1.1
A(2,1) = 2.2
A(3,1) = 3.3
A(4,1) = 4.4

The item after 1.1 is 2.2, and so

A(1,2) = 2

Similarly,

A(2,2) = 3
A(3,2) = 4

Now 4.4 is at the end of the list, so we'll mark this by a negative number:

A(4,2) = −1

This couldn't possibly be the next location, since the array subscripts start at 0, and so the program will be able to recognise the end of the list. We also need a 'pointer' to the start of the list:

S = 1

The following program will print the list defined above (see also Figure 1):

10 I = S

I is the current item. Point it to the start.

20 PRINT A(I,1)

prints the current item.

30 I = A(I,2)

sets I to the *next* element's location.

40 IF I › 0 THEN 20

repeats the process until all the items have been printed. When 4.4 has been printed, line 30 will set I = A(4,2), which is −1, and line 40 will 'trap' this as the end of the list.

Fine, we can read the list out — what about adding items?

This is where it gets a bit cleverer — we don't have to move any of the data. All we have to do is to find a place for the data, point one of the list's pointers at it (*which* pointer depends on where in the list the new item is to be added) and then point the new item's pointer back into the list where it was 'broken'.

First of all, though, we have to find out if there's room. Remember, we have no way of knowing (yet) which locations hold data and which are unused (without going through the list each time, that is). What if we set the 'next item' part of the unused locations to 0. Then we'll be able to tell the locations from data, or from the 'end of list' marker (which is −1). The unused location in the above example was location 5, so

A(5,2) = 0. ▶

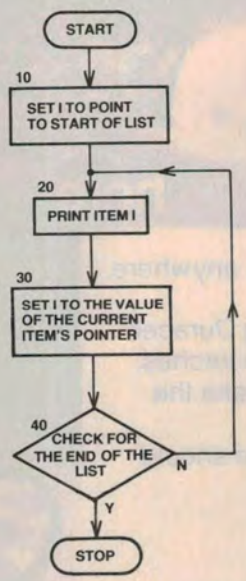
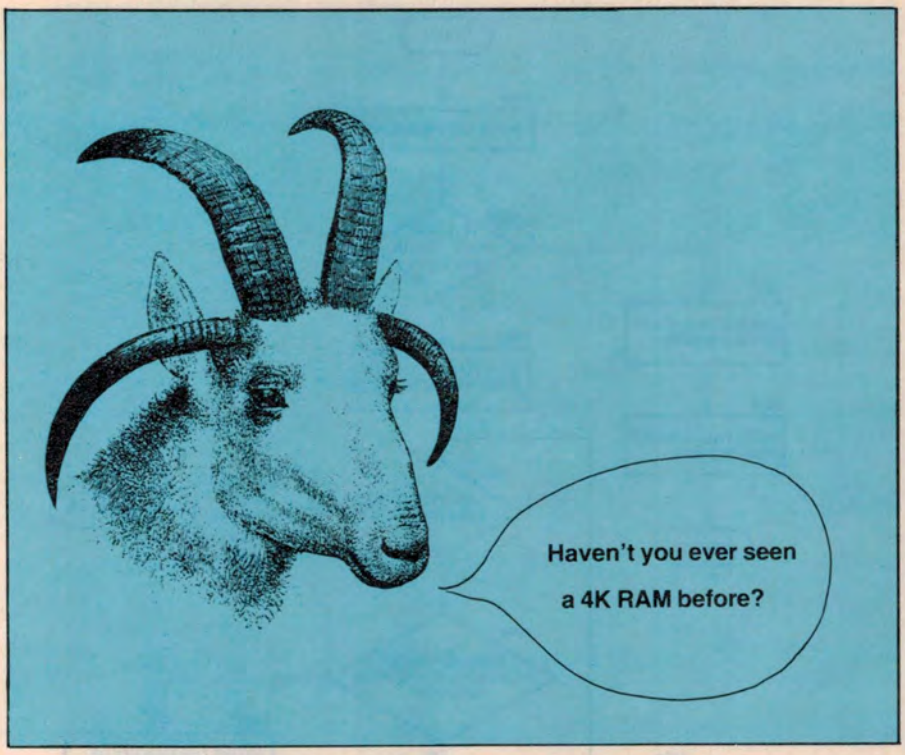


Figure 1. **A program which allows the list to be printed in order.**



What we have to do to add an item, then, is

a) Look for an empty location (and STOP if there are none!)

b) Change the pointer of the item which comes before the one we want to add to point to the new item.

c) Set the pointer of the new item to point to the item which will come after it.

Say we want to change the list to:

1.1, 2.2, 99, 3.3, 4.4

We set

A(2,2) = 5

A(5,1) = 99

A(5,2) = 3

The following program segment will add an item with value V after item L (see flowchart, Figure 2).

```
10 FOR I = 1 TO 5
20 IF A(I,2) = 0 THEN 50
30 NEXT I
40 PRINT "NO ROOM LEFT" : STOP
```

checks for empty locations (which have a pointer value of 0) and stops if there are none.

```
50 A(I,1) = V
```

I should be the number of the first empty location.

```
60 A(I,2) = A(L,2)
```

sets the pointer of the new item to where the pointer of the item was before it *was* pointing (whew!).

```
70 A(L,2) = I
```

completes the addition.

The following segment will remove item R. This is rather more tricky than the last program, as we have to find out which item is pointing *at* item R. It won't remove R if it's the first member of the list, by the way — this involves changing the value of S. It's easy enough to add a small section of program to check for this, though (see the full-sized example later in the article).

```
10 I = S
```

the start of the list;

```
20 IF A(I,2) = R THEN 60
```

if the item I *points at* R, jump.

```
30 IF A(I,2) < 0 THEN PRINT
   "NOT FOUND" : STOP
```

If R hasn't been found by the end of the list then *something's* wrong!

```
40 I = A(I,2)
50 GOTO 20
```

try the next one:

```
60 A(I,2) = A(R,2)
```

This takes care of the pointers in the list. All we have to do now is to mark R as being empty:

```
70 A(R,2) = 0
```

Figure 3 shows the flowchart of the above program. ▶

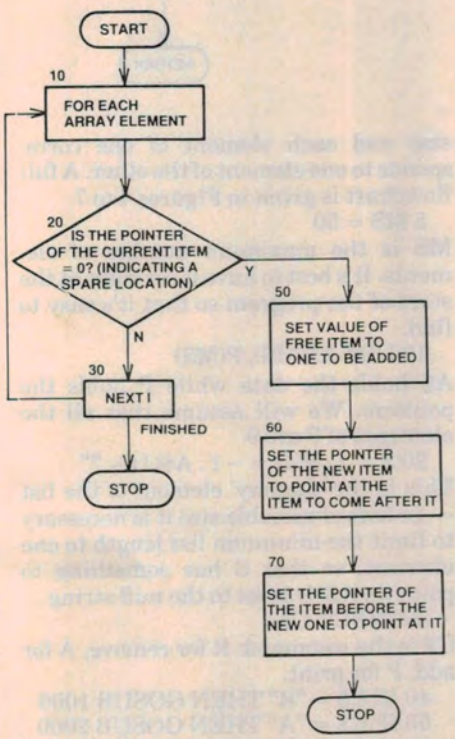


Figure 2. **This segment adds one item to a specified position in the list.**

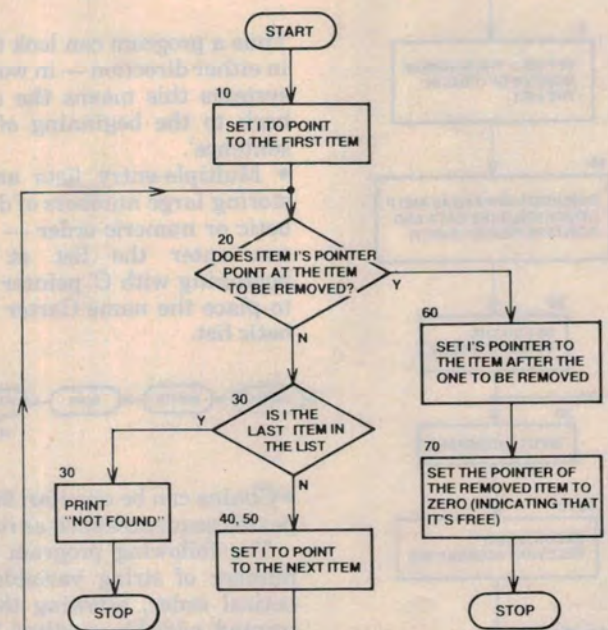
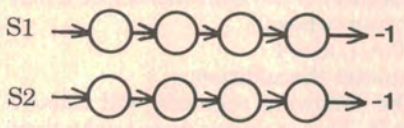


Figure 3. **Removes a specified item — without the need to 'shuffle' data.**

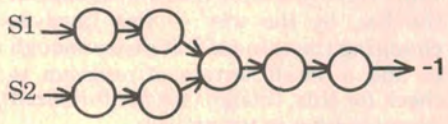
Other points to note about chaining are:

• There's no reason why there should only be one list per array. All that's required is to have several 'start of chain' pointers (like S in the above example). Remember — one array takes up less room than two of the same total size.

• Several lists can share the same 'tail'. If this is a representation of two lists:



then there's no reason why they shouldn't be arranged like this



This sort of thing is useful for mailing list systems where some addresses would appear on several lists — members of a club who are also committee members, for instance. Their addresses would only be recorded once.

• List pointers can go in both directions:

A(1,1) = 1.1, A(2,1) = 2.2, A(3,1) = 3.3
A(1,2) = 2, A(2,2) = 3, A(3,2) = −1
A(1,3) = −1, A(2,2) = 1, A(3,3) = 2

Using the A(I,2) pointers would cause the list to be read as

1.1, 2.2, 3.3

whereas using the A(1,3) pointers would cause it to be read as

3.3, 2.2, 1.1

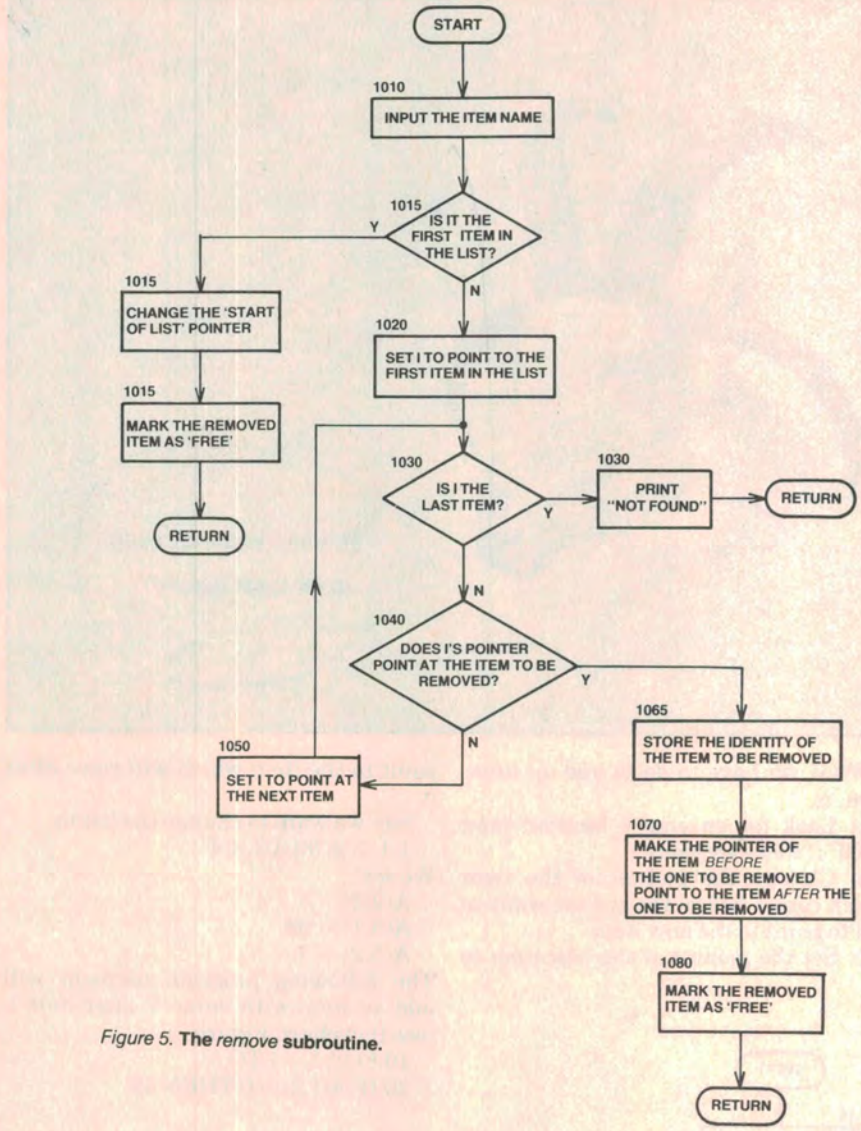


Figure 5. **The *remove* subroutine.**

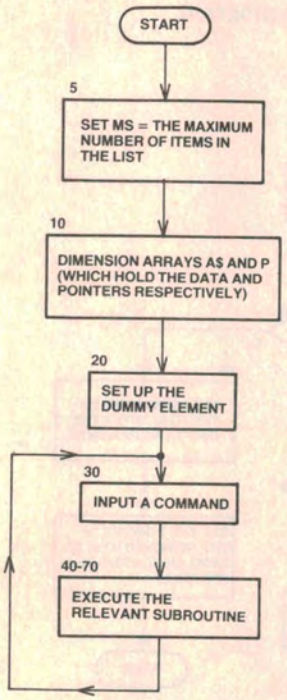


Figure 4. **The main program. Flowcharts for the subroutines are given in Figures 5 to 7.**

Thus a program can look through a list in either direction — in word processing systems this means the ability to 'go back to the beginning of the current sentence'.

• Multiple-entry lists are useful for storing large numbers of data in alphabetic or numeric order — the program can enter the list at the 'words beginning with C' pointer when trying to place the name Carter in an alphabetic list.

• Chains can be circular, for storing repeated sequences such as rotas.

The following program will store a number of string variables in alphabetical order, allowing the list to be printed, added to or edited. The pointers are stored in a separate array to the data but the two arrays are the same size and each element of one corresponds to one element of the other. A full flowchart is given in Figures 4 to 7.

```
5 MS = 50
```

MS is the maximum number of elements. It's best to have it set right at the start of the program so that it's easy to find.

```
10 DIM A$(MS), P(MS)
```

A$ holds the data while P holds the pointers. We will assume that all the elements of P are 0.

```
20 S = 1 : P(1) = −1 : A$(1) = ""
```

This is the 'dummy' element of the list — in lists of variable size it is necessary to limit the minimum list length to one element, so that S has something to point to. A$(1) is set to the null string.

```
30 INPUT C$
```

C$ is the command: R for remove, A for add, P for print.

```
40 IF C$ = "R" THEN GOSUB 1000
50 IF C$ = "A" THEN GOSUB 2000
60 IF C$ = "P" THEN GOSUB 3000
70 GOTO 30
```

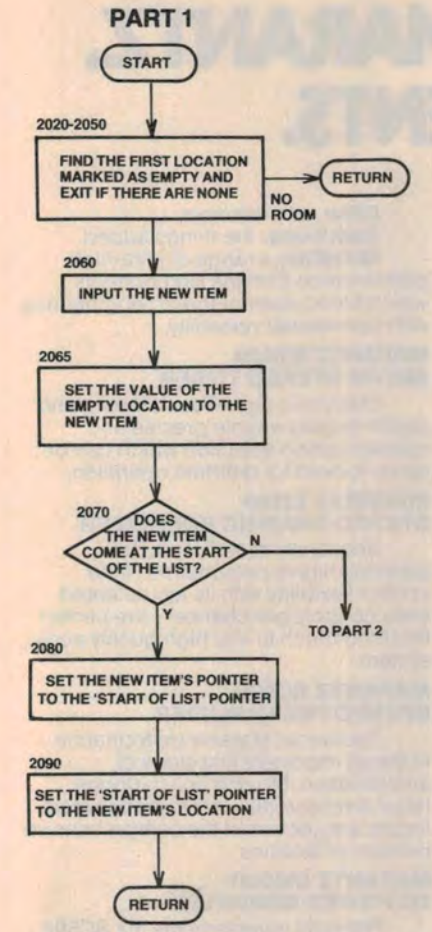
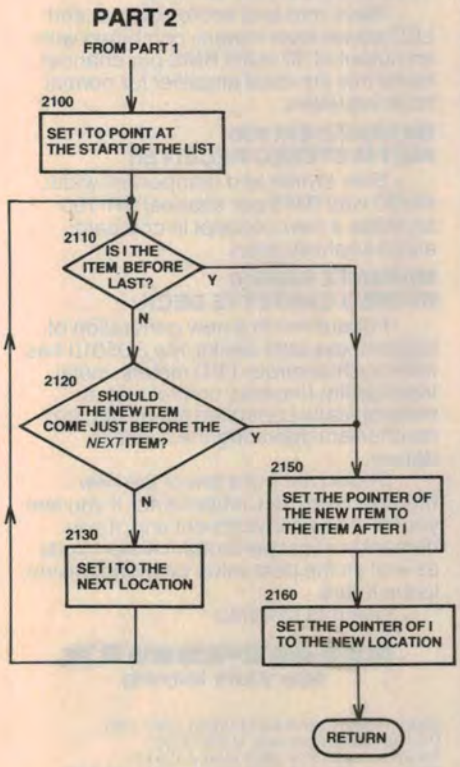


Figure 6. Add **subroutine — this also checks to see if there's room.**



executes the relevant subroutine and returns for the next command.

```
1000  REM REMOVAL
      SUBROUTINE
1010  INPUT "WHICH ITEM "; W$
```

finds out which item to remove.

```
1015  IF A$(S) = W$ THEN
      SS=S : S = P(S) : P(SS)=0
      RETURN
```

checks to see if the first member of the list is the one to be removed. As the algorithm used in the subroutine can't handle this directly, this line grabs it when it occurs. SS is a temporary store for S — see line 1080 later.

```
1020  I = S
1030  IF P(I) = -1 THEN PRINT
      "NOT FOUND" : RETURN
```

Since we're looking one item ahead — so that we can change the pointer which points *at* W$ — we look for the end of the list at the item *after* item I.

```
1040  IF A$(P(I)) = W$ THEN 1065
```

found W$ in the next item? — then jump out of the loop. If not,

```
1050  I = P(I)
1060  GOTO 1030
```

. . . try the next one. If W$ *has* been found, these next two lines take care of the pointers:

```
1065  P = P(I)
1070  P(I) = P(P(I))
```

Now mark item I as being empty:

```
1080  P(P) = 0
```

P was a temporary store for the value of P(I), as we need this value at line 1080 but it has been changed by line 1070. Location I is now empty. The dummy location (see line 20) will not be removed by this subroutine, as line 1010 will not accept a null string and the routine cannot, in any case, remove the last item in the list — line 1030 would prevent it.

```
1090  RETURN
2000  REM ADD AN ITEM
```

First, find out if there's room:

```
2020  FOR F = 1 TO MS
2030  IF P(F) = 0 THEN 2060
2040  NEXT F
2050  PRINT
      "NO ROOM" : RETURN
```

F is now the location of the first free element of the array — if there are any.

```
2060  INPUT "ITEM "; W$
2065  A$(F) = W$
```

This routine uses an alternative approach to the problem of dealing with the beginning and end of the list — the end causes problems for the REMOVE subroutine and this is why the dummy element was put in at the end. The beginning of the list will cause problems for this routine if we're not careful but we won't put a dummy element in the beginning as well — just to show the alternative approach. Instead:

```
2070  IF A$(S) ‹› "" THEN
      IF A$(S) ‹ W$ THEN 2100
```

If the dummy element isn't the only one in the list and W$ doesn't come first in the list, jump to 2100. The ‹ can be used to compare strings alphabetically in most comprehensive BASICs. If the version you're using doesn't have this feature you will have to write a short subroutine to do it. The IF . . . THEN IF . . . THEN construction causes the program to ignore the result of IF "" › W$. If W$ got past line 2080 then it must come just at the start of the list, so:

```
2080  P(F) = S : S = F
2090  RETURN
```

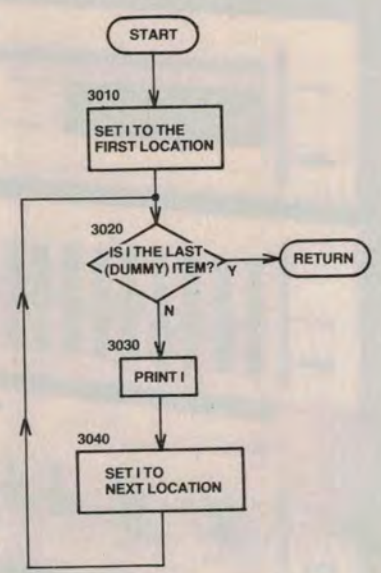


Figure 7. **The *list* routine — the simplest of the three. Note how the addition of a dummy element makes this routine different from Figure 1.**

However, if W$ comes further down the list,

```
2100  I = S
2110  IF P(P(I)) = -1 THEN 2150
```

If the search reaches the end of the list, W$ must come last in the list.

```
2120  IF A$(P(I)) › W$ THEN 2150
```

If the next item is further down the alphabet than W$, it must go in just after I (and before A$(P(I))). If not, try the next one:

```
2130  I = P(I)
2140  GOTO 2110
```

Now add location F in the right place:

```
2150  P(F) = P(I)
2160  P(I) = F
2170  RETURN
```

Now for the simplest of the three routines:

```
3000  REM PRINT
3010  I = S
3020  IF P(I) = -1 THEN RETURN
```

This prevents the dummy element from being printed, and ends the subroutine's execution.

```
3030  PRINT A$(I)
3040  I= P(I)
3050  GOTO 3020
```

●