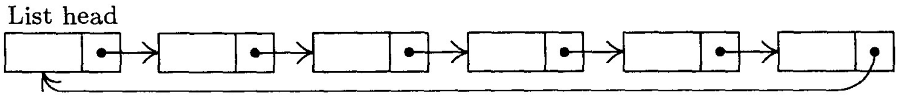
are operating on an entire list, moving from one node to the next, we should stop when we get back to our starting place (assuming, of course, that the starting place is still present in the list).

An alternative solution to the problem just posed is to put a special, recognizable node into each circular list, as a convenient stopping place. This special node is called the *list head*, and in applications we often find it is quite convenient to insist that every circular list have exactly one node that is its list head. One advantage is that the circular list will then never be empty. With a list head, diagram (1) becomes



(4)

References to lists like (4) are usually made via the list head, which is often in a fixed memory location. The disadvantage of list heads is that don't have a pointer to the right end, so we must sacrifice operation (b) stated above.

Diagram (4) may be compared with 2.2.3–(1) at the beginning of the previous section, in which the link associated with "item 5" now points to LOC(FIRST) instead of to Λ; the variable FIRST is now thought of as a link within a node, namely the link that is in NODE(LOC(FIRST)). The principal difference between (4) and 2.2.3–(1) is that (4) makes it possible (though not necessarily efficient) to get to any point of the list from any other point.

As an example of the use of circular lists, we will discuss *arithmetic on polynomials* in the variables x, y, and z, with integer coefficients. There are many problems in which a scientist wants to manipulate polynomials instead of just numbers; we are thinking of operations like the multiplication of

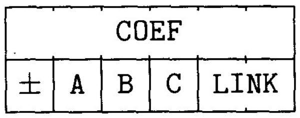
$$(x^4 + 2x^3y + 3x^2y^2 + 4xy^3 + 5y^4) \qquad \text{by} \qquad (x^2 - 2xy + y^2)$$

to get

$$(x^6 - 6xy^5 + 5y^6).$$

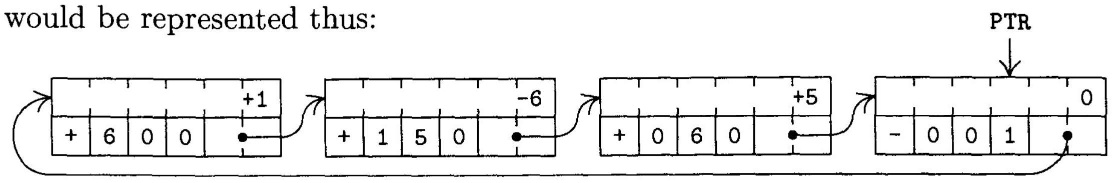
Linked allocation is a natural tool for this purpose, since polynomials can grow to unpredictable sizes and we may want to represent many polynomials in memory at the same time.

We will consider here the two operations of addition and multiplication. Let us suppose that a polynomial is represented as a list in which each node stands for one nonzero term, and has the two-word form



. (5)

Here COEF is the coefficient of the term in $x^{\mathrm{A}}y^{\mathrm{B}}z^{\mathrm{C}}$. We will assume that the coefficients and exponents will always lie in the range allowed by this format, and that it is not necessary to check the ranges during our calculations. The notation ABC will be used to stand for the ± A B C fields of the node (5), treated as a single unit. The sign of ABC, namely the sign of the second word in (5), will always be plus, except that there is a *special node* at the end of every polynomial that has ABC $= -1$ and COEF $= 0$. This special node is a great convenience, analogous to our discussion of a list head above, because it provides a convenient sentinel and it avoids the problem of an empty list (corresponding to the polynomial 0). The nodes of the list always appear in *decreasing order* of the ABC field, if we follow the direction of the links, except that the special node (which has ABC $= -1$) links to the largest value of ABC. For example, the polynomial $x^6 - 6xy^5 + 5y^6$ would be represented thus:



Algorithm A (*Addition of polynomials*). This algorithm adds polynomial(P) to polynomial(Q), assuming that P and Q are pointer variables pointing to polynomials having the form above. The list P will be unchanged; the list Q will retain the sum. Pointer variables P and Q return to their starting points at the conclusion of this algorithm; auxiliary pointer variables Q1 and Q2 are also used.

A1. [Initialize.] Set P $\leftarrow$ LINK(P), Q1 $\leftarrow$ Q, Q $\leftarrow$ LINK(Q). (Now both P and Q point to the leading terms of their polynomials. Throughout most of this algorithm the variable Q1 will be one step behind Q, in the sense that Q $=$ LINK(Q1).)

A2. [ABC(P):ABC(Q).] If ABC(P) $<$ ABC(Q), set Q1 $\leftarrow$ Q and Q $\leftarrow$ LINK(Q) and repeat this step. If ABC(P) $=$ ABC(Q), go to step A3. If ABC(P) $>$ ABC(Q), go to step A5.

A3. [Add coefficients.] (We've found terms with equal exponents.) If ABC(P) < 0, the algorithm terminates. Otherwise set COEF(Q) $\leftarrow$ COEF(Q) $+$ COEF(P). Now if COEF(Q) $= 0$, go to A4; otherwise, set P $\leftarrow$ LINK(P), Q1 $\leftarrow$ Q, Q $\leftarrow$ LINK(Q), and go to A2. (Curiously the latter operations are identical to step A1.)

A4. [Delete zero term.] Set Q2 $\leftarrow$ Q, LINK(Q1) $\leftarrow$ Q $\leftarrow$ LINK(Q), and AVAIL $\Leftarrow$ Q2. (A zero term created in step A3 has been removed from polynomial(Q).) Set P $\leftarrow$ LINK(P) and go back to A2.

A5. [Insert new term.] (Polynomial(P) contains a term that is not present in polynomial(Q), so we insert it in polynomial(Q).) Set Q2 $\Leftarrow$ AVAIL, COEF(Q2) $\leftarrow$ COEF(P), ABC(Q2) $\leftarrow$ ABC(P), LINK(Q2) $\leftarrow$ Q, LINK(Q1) $\leftarrow$ Q2, Q1 $\leftarrow$ Q2, P $\leftarrow$ LINK(P), and return to step A2. ▌

One of the most noteworthy features of Algorithm A is the manner in which the pointer variable Q1 follows the pointer Q around the list. This is very typical of list processing algorithms, and we will see a dozen more algorithms with the same characteristic. Can the reader see why this idea was used in Algorithm A?

A reader who has little prior experience with linked lists will find it very instructive to study Algorithm A carefully; as a test case, try adding $x + y + z$ to $x^2 - 2y - z$.

Given Algorithm A, the multiplication operation is surprisingly easy:

Algorithm M (*Multiplication of polynomials*). This algorithm, analogous to Algorithm A, replaces polynomial(Q) by

$$\text{polynomial(Q)} + \text{polynomial(M)} \times \text{polynomial(P)}.$$

M1. [Next multiplier.] Set `M` $\leftarrow$ `LINK(M)`. If `ABC(M)` < 0, the algorithm terminates.

M2. [Multiply cycle.] Perform Algorithm A, except that wherever the notation "`ABC(P)`" appears in that algorithm, replace it by "if `ABC(P)` < 0 then -1, otherwise `ABC(P)+ABC(M)`"; wherever "`COEF(P)`" appears in that algorithm replace it by "`COEF(P)` $\times$ `COEF(M)`". Then go back to step M1. ▌

The programming of Algorithm A in `MIX` language shows again the ease with which linked lists are manipulated in a computer. In the following code we assume that `OVERFLOW` is a subroutine that either terminates the program (due to lack of memory space) or finds further available space and exits to rJ $-$ 2.

Program A (*Addition of polynomials*). This is a subroutine written so that it can be used in conjunction with a multiplication subroutine (see exercise 15).

Calling sequence: `JMP ADD`
Entry conditions: rI1 = `P`, rI2 = `Q`.
Exit conditions: polynomial(Q) has been replaced by polynomial(Q) + polynomial(P); rI1 and rI2 are unchanged; all other registers have undefined contents.

In the coding below, `P` $\equiv$ rI1, `Q` $\equiv$ rI2, `Q1` $\equiv$ rI3, and `Q2` $\equiv$ rI6, in the notation of Algorithm A.

```
LINK EQU  4:5                    Definition of LINK field
ABC  EQU  0:3                    Definition of ABC field
ADD  STJ  3F               1     Entrance to subroutine
1H   ENT3 0,2          1+m''     A1. Initialize. Set Q1 ← Q.
     LD2  1,3(LINK)    1+m''     Q ← LINK(Q1).
0H   LD1  1,1(LINK)      1+p     P ← LINK(P).
SW1  LDA  1,1            1+p     rA(0:3) ← ABC(P).
2H   CMPA 1,2(ABC)         x     A2. ABC(P):ABC(Q).
     JE   3F               x     If equal, go to A3.
     JG   5F           p'+q'     If greater, go to A5.
     ENT3 0,2             q'     If less, set Q1 ← Q.
     LD2  1,3(LINK)       q'     Q ← LINK(Q1).
     JMP  2B              q'     Repeat.
3H   JAN  *              m+1     A3. Add coefficients.
SW2  LDA  0,1              m     COEF(P)
     ADD  0,2              m        + COEF(Q)
     STA  0,2              m           → COEF(Q).
     JANZ 1B               m     Jump if nonzero.
```