4.2. POINTERS OR REFERENCES

The characteristic property of recursive structures which clearly distinguishes them from the fundamental structures (arrays, records, sets) is their ability to vary in size. Hence, it is impossible to assign a fixed amount of storage to a recursively defined structure, and as a consequence a compiler cannot associate specific addresses to the components of such variables. The technique most commonly used to master this problem involves a *dynamic allocation* of storage, i.e., allocation of store to individual components at the time when they come into existence during program execution, instead of at translation time. The compiler then allocates a fixed amount of storage to hold the *address* of the dynamically allocated component instead of the component itself. For instance, the pedigree illustrated in Fig. 4.2 would be represented by individual—quite possibly non-contiguous—records, one for each person. These persons are then linked by their addresses assigned to the respective "father" and "mother" fields. Graphically, this situation is best expressed by the use of arrows or pointers (see Fig. 4.3).

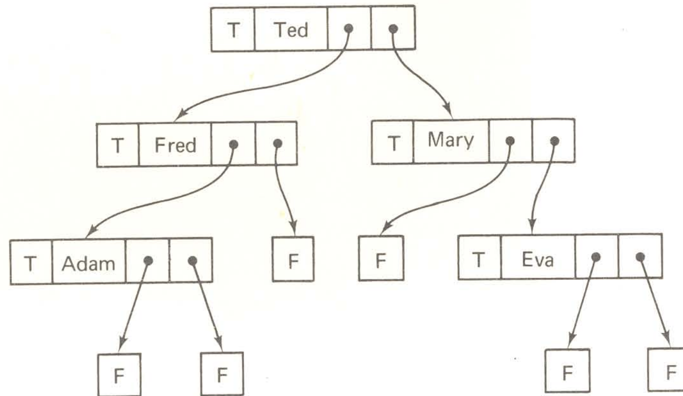

Fig. 4.3 Structure linked by pointers.

It must be emphasized that the use of pointers to implement recursive structures is merely a technique. The programmer need not be aware of their existence. Storage may be allocated automatically the first time a new component is referenced. However, if the technique of using references or pointers is made explicit, more general data structures can be constructed than those definable by purely recursive data definition. In particular, it is then possible to define "infinite" or circular structures and to dictate that certain structures are *shared*. It has therefore become common in advanced programming languages to make possible the explicit manipulation of references to data in addition to the data themselves. This implies that a clear notational distinction must exist between data and references to data and that

consequently data types must be introduced whose values are pointers (references) to other data. The notation we use for this purpose is the following:

$$\textbf{type } T_p = \uparrow T \tag{4.4}$$

The type declaration (4.4) expresses that values of type T_p are pointers to data of type T. Thus, the arrow in (4.4) is verbalized as "pointer to." It is fundamentally important that the type of elements pointed to is evident from the declaration of T_p. We say that T_p is *bound* to T. This binding distinguishes pointers in higher-level languages from addresses in assembly codes, and it is a most important facility to increase security in programming through redundancy of the underlying notation.

Values of pointer types are generated whenever a data item is dynamically allocated. We will adhere to the convention that such an occasion be explicitly mentioned at all times. This is in contrast to the situation in which the first time that an item is mentioned it is automatically (assumed to be) allocated. For this purpose, we introduce the intrinsic procedure *new*. Given a pointer variable p of type T_p, the statement

$$new(p) \tag{4.5}$$

effectively allocates a variable of type T, generates a pointer of type T_p referencing this new variable, and assigns this pointer to the variable p (see Fig. 4.4). The pointer value itself can now be referred to as p (i.e., as the value of the pointer variable p). In contrast, the variable which is referenced by p is denoted by $p\uparrow$. This is the dynamically allocated variable of type T.

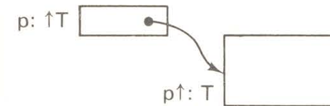

Fig. 4.4 Dynamic allocation of variable $p\uparrow$.

It was mentioned above that a variant component is essential in every recursive type to ensure finite cardinality. The example of the family predigree is of a pattern that exhibits a most frequently occurring constellation [see (4.3)], namely, the case in which the tagfield is two-valued (Boolean) and in which its value being *false* implies the absence of any further components. This is expressed by the *declaration schema* (4.6).

$$\textbf{type } T = \textbf{record if } p \textbf{ then } S(T) \textbf{ end} \tag{4.6}$$

$S(T)$ denotes a sequence of field definitions which includes one or more fields of type T, thereby ensuring recursivity. All structures of a type patterned after (4.6) will exhibit a tree (or list) structure similar to that shown in Fig. 4.3. Its peculiar property is that it contains pointers to data components with a tagfield only, i.e., without further relevant information. The implementation technique using pointers suggests an easy way of saving storage space by

letting the tagfield information be included in the value of the pointer itself. The common solution is to *extend* the range of values of a type T_p by a single value that is pointing to no element at all. We denote this value by the special symbol **nil**, and we understand that **nil** is automatically an element of all pointer types declared. This extension of the range of pointer values explains why finite structures may be generated *without* the explicit presence of variants (conditions) in their (recursive) declaration.

The new formulations of the data types declared in (4.1) and (4.3)—based on explicit pointers—are given in (4.7) and (4.8), respectively. Note that in the latter case (which originally corresponded to the schema (4.6)) the variant record component has vanished, since $p\uparrow.known = false$ is now expressed as $p =$ **nil**. The renaming of the type *ped* to *person* reflects the difference in the viewpoint brought about by the introduction of explicit pointer values. Instead of first considering the given structure in its entirety and then investigating its substructure and its components, attention is focused on the components in the first place, and their interrelationship (represented by pointers) is not evident from any fixed declaration.

```
type expression = record op: operator;
                    opd1, opd2: ↑term
                  end;
type term       = record                                  (4.7)
                    if t then (id: alfa)
                         else (sub: ↑expression)
                  end

type person     = record name: alfa;
                    father, mother: ↑person               (4.8)
                  end
```

The data structure representing the pedigree shown in Figs. 4.2 and 4.3 is again shown in Fig. 4.5 in which pointers to unknown persons are denoted by **nil**. The ensuing improvement in storage economy is obvious.

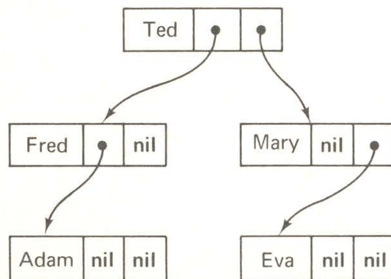

Fig. 4.5 Structure with **nil** pointers.

Again referring to Fig. 4.5, assume that Fred and Mary are siblings, i.e., have the same father and mother. This situation is easily expressed by replacing the two **nil** values in the respective fields of the two records. An implementation that hides the concept of pointers or uses a different technique of storage handling would force the programmer to represent the records of Adam and Eva twice each. Although in accessing their data for inspection it does not matter whether the two fathers (and the two mothers) are duplicated or represented by a single record, the difference is *essential when selective updating* is permitted. Treating pointers as explicit data items instead of as hidden implementation aids allows the programmer to express clearly where *storage sharing* is intended.

A further consequence of the explicitness of pointers is that it is possible to define and manipulate cyclic data structures. This additional flexibility yields, of course, not only increased power but also requires increased care by the programmer because the manipulation of cyclic data structures may easily lead to non-terminating processes.

This phenomenon of power and flexibility being intimately coupled with the danger of misuse is well-known in programming, and it particularly recalls the **goto** statement. Indeed, if the analogy between program structures and data structures is to be extended, the purely recursive data structure could well be placed at the level corresponding with the procedure, whereas the introduction of pointers is comparable to the use of **goto** statements. For, as the **goto** statement allows the construction of any kind of program pattern (including loops), so do pointers allow for the composition of any kind of data structure (including cycles). The parallel development of corresponding program and data structures is shown in concise form in Table 4.1.

In Chap. 3 it was shown that iteration is a special case of recursion and

Construction Pattern	Program Statement	Data Type
Atomic element	Assignment	Scalar type
Enumeration	Compound statement	Record type
Repetition by a known factor	For statement	Array type
Choice	Conditional statement	Variant record, type union
Repetition by an unknown factor	While or repeat statement	Sequence or file type
Recursion	Procedure statement	Recursive data type
General "graph"	Go to statement	Structure linked by pointers

Table 4.1 Correspondences of Program and Data Structures.

that a call of a recursive procedure P defined according to schema (4.9)

$$\begin{array}{l}\textbf{procedure } P;\\ \textbf{begin}\\ \quad \textbf{if } B \textbf{ then begin } P_0;\ P \textbf{ end}\\ \textbf{end}\end{array} \tag{4.9}$$

where P_0 is a statement not involving P, is equivalent to and replaceable by the iterative statement

$$\textbf{while } B \textbf{ do } P_0$$

The analogies outlined in Table 4.1 reveal that a similar relationship holds between recursive data types and the sequence. In fact, a recursive type defined according to the schema

$$\begin{array}{l}\textbf{type } T = \textbf{record}\\ \qquad \textbf{if } B \textbf{ then } (t_0\colon T_0;\ t\colon T)\\ \quad \textbf{end}\end{array} \tag{4.10}$$

where T_0 is a type not involving T, is equivalent and replaceable by the sequential data type

$$\textbf{file of } T_0$$

This shows that recursion can be replaced by iteration in program *and* data definitions if (and only if) the procedure or type name occurs recursively only once at the end (or the beginning) of its definition.

The remainder of this chapter is devoted to the generation and manipulation of data structures whose components are linked by explicit pointers. Structures with specific simple patterns are emphasized in particular; recipes for handling more complex structures may be derived from those for manipulating basic formations. These are the linear list or chained sequence—the most simple case—and trees. Our preoccupation with these "building blocks" of data structuring does not imply that more involved structures do not occur in practice. In fact, the following story which appeared in a Zürich newspaper in July 1922 is a proof that irregularity may even occur in cases which usually serve as examples for regular structures, such as (family) trees. The story tells of a man who describes the misery of his life in the following words:

> I married a widow who had a grown-up daughter. My father, who visited us quite often, fell in love with my step-daughter and married her. Hence, my father became my son-in-law, and my step-daughter became my mother. Some months later, my wife gave birth to a son, who became the brother-in-law of my father as well as my uncle. The wife of my father, that is my step-daughter, also had a son. Thereby, I got a brother and at the same time a grandson. My wife is my grandmother, since she is my mother's mother. Hence, I am my wife's husband and at the same time her step-grandson; in other words, I am my own grandfather.

4.3. LINEAR LISTS

4.3.1. Basic Operations

The simplest way to interrelate or link a set of elements is to line them up in a single *list* or *queue*. For, in this case, only a single link is needed for each element to refer to its successor.

Assume that a type T is defined as shown in (4.11). Every variable of this type consists of three components, namely, an identifying key, the pointer to its successor, and possibly further associated information omitted in (4.11).

$$\begin{array}{l}\textbf{type } T = \textbf{record } key\colon integer;\\ \qquad\qquad\qquad next\colon \uparrow T;\\ \qquad\qquad\qquad \ldots\ldots\\ \qquad\qquad \textbf{end}\end{array} \tag{4.11}$$

A list of T's, with a pointer to its first component being assigned to a variable p, is illustrated in Fig. 4.6. Probably the simplest operation to be performed with a list as shown in Fig. 4.6 is the *insertion of an element at its head*. First, an element of type T is allocated, its reference (pointer) being assigned to an auxiliary pointer variable, say q. Thereafter, a simple reassignment of pointers completes the operation, which is programmed in (4.12).

$$new(q);\ q\uparrow.next := p;\ p := q \tag{4.12}$$

Note that the order of these three statements is essential.

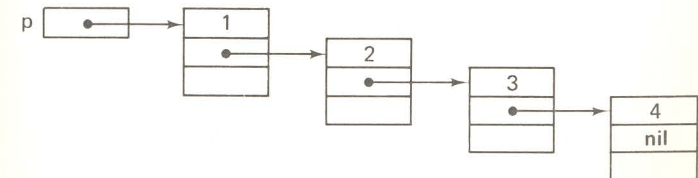

Fig. 4.6 Example of a list.

The operation of inserting an element at the head of a list immediately suggests how such a list can be generated: starting with the empty list, a heading element is added repeatedly. The process of *list generation* is expressed in (4.13); here the number of elements to be linked is n.

$$\begin{array}{l}p := \textbf{nil};\ \ \{start\ with\ empty\ list\}\\ \textbf{while } n > 0 \textbf{ do}\\ \quad \textbf{begin } new(q);\ q\uparrow.next := p;\ p := q;\\ \qquad q\uparrow.key := n;\ n := n-1\\ \quad \textbf{end}\end{array} \tag{4.13}$$

This is the simplest way of forming a list. However, the resulting order of elements is the inverse of the order of their "arrival." In some applications this is undesirable; consequently, new elements must be appended at the *end* of the list. Although the end can easily be determined by a scan of the list, this naive approach involves an effort that may as well be saved by using a second pointer, say q, always designating the last element. This method is, for example, applied in Program 4.4 which generates cross-references to a given text. Its disadvantage is that the first element inserted has to be treated differently from all later ones.

The explicit availability of pointers makes certain operations very simple which are otherwise cumbersome; among the elementary list operations are those of inserting and deleting elements (selective updating of a list), and, of course, the traversal of a list. We first investigate *list insertion.*

Assume that an element designated by a pointer (variable) q is to be inserted in a list *after* the element designated by the pointer p. The necessary pointer assignments are expressed in (4.14), and their effect is visualized by Fig. 4.7.

$$q\uparrow.next := p\uparrow.next;\ p\uparrow.next := q \tag{4.14}$$

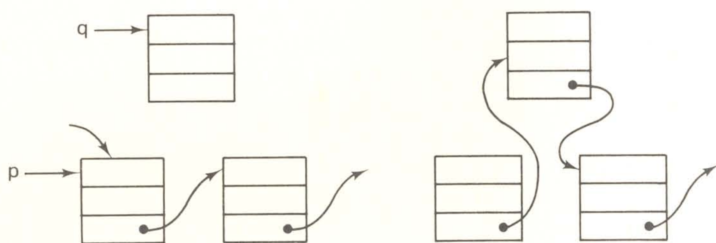

Fig. 4.7 List insertion after $p\uparrow$.

If insertion *before* instead of after the designated element $p\uparrow$ is desired, the one-directional link chain seems to cause a problem because it does not provide any kind of path to an element's predecessors. However, a simple "trick" solves our dilemma: it is expressed in (4.15) and illustrated in Fig. 4.8. Assume that the key of the new element is $k = 8$.

$$\begin{aligned} &new(q);\ q\uparrow := p\uparrow; \\ &p\uparrow.key := k;\ p\uparrow.next := q \end{aligned} \tag{4.15}$$

The "trick" evidently consists of actually inserting a new component *after* $p\uparrow$, but then interchanging the values of the new element and of $p\uparrow$.

Next, we consider the process of *list deletion.* Deleting the successor of a $p\uparrow$ is straightforward. In (4.16) it is shown in combination with the re-insertion of the deleted element at the head of another list (designated by q). r is an auxiliary variable of type $\uparrow T$.

$$\begin{aligned} &r := p\uparrow.next;\ p\uparrow.next := r\uparrow.next; \\ &r\uparrow.next := q;\ q := r \end{aligned} \tag{4.16}$$

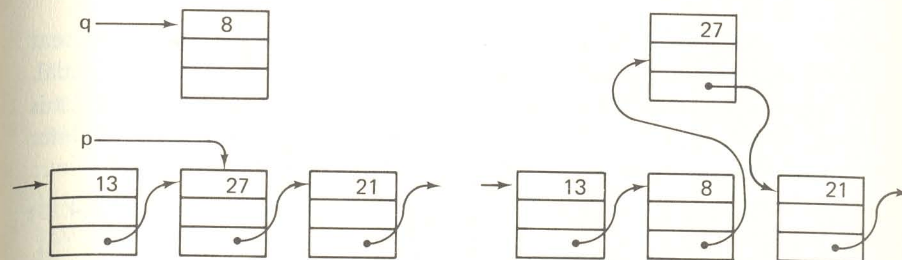

Fig. 4.8 List insertion before $p\uparrow$.

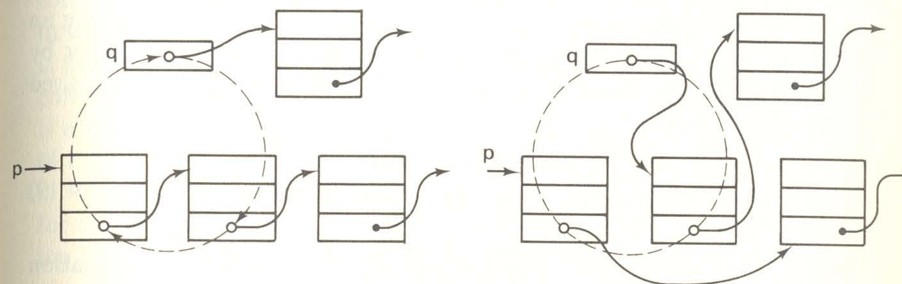

Fig. 4.9 List deletion and re-insertion.

Figure 4.9 illustrates process (4.16) and shows that it consists of a cyclic exchange of three pointers.

More difficult is the removal of a designated element itself (instead of its successor), since we encounter the same problem as with insertion in front of a $p\uparrow$: backtracking to the denoted element's predecessor is impossible. But deleting the successor after moving its value forward is a relatively obvious and simple solution. It can be applied whenever $p\uparrow$ has a successor, i.e., is not the last element on the list.

We now turn to the fundamental operation of *list traversal.* Let us assume that an operation $P(x)$ has to be performed for every element of the list whose first element is $p\uparrow$. This task is expressible as follows:

while *list designated by p is not empty* **do**
begin *perform operation P;*
    proceed to the successor
end

In detail, this operation is described by statement (4.17).

while $p \neq$ **nil do**
    begin $P(p\uparrow);\ p := p\uparrow.next$ (4.17)
    end

It follows from the definitions of the **while** statement and of the linking structure that P is applied to all elements of the list and to no other ones.

A very frequent operation performed is *list searching* for an element with a given key x. As with file structures, the search is purely sequential. The search terminates either if an element is found or if the end of the list is reached. Again, we assume that the head of the list is designated by a pointer p. A first attempt to formulate this simple search results in the following:

```
while (p ≠ nil) ∧ (p↑.key ≠ x) do p := p↑.next        (4.18)
```

However, it must be noticed that $p = \textbf{nil}$ implies that $p\uparrow$ does not exist. Hence, evaluation of the termination condition may imply access to a non-existing variable (in contrast to a variable with undefined value) and may cause failure of program execution. This can be remedied either by using an explicit break of the repetition expressed by a **goto** statement (4.19) or by introducing an auxiliary Boolean variable to record whether or not a desired key was found (4.20).

```
while p ≠ nil do
    if p↑.key = x then goto Found                      (4.19)
                  else p := p↑.next
```

The use of the **goto** statement requires the presence of a destination label at some place; note that its incompatibility with the **while** statement is evidenced by the fact that the while clause becomes misleading: the controlled statement is *not* necessarily executed as long as $p \neq \textbf{nil}$.

```
b := true;
while (p ≠ nil) ∧ b do
    if p↑.key = x then b := false                      (4.20)
                  else p := p↑.next
{(p=nil) ∨ ¬b}
```

4.3.2. Ordered Lists and Re-organizing Lists

Algorithm (4.20) strongly recalls the search routines for scanning an array or a file. In fact, a file is nothing but a linear list in which the technique of linkage to the successor is left unspecified or implicit. Since the primitive file operators do not allow insertion of new elements (except at the end) or deletion (except removal of *all* elements), the choice of representation is left wide open to the implementor, and he may well use sequential allocation, leaving successive components in contiguous storage areas. Linear lists with explicit pointers provide *more flexibility*, and therefore they should be used whenever this additional flexibility is needed.

To exemplify, we will now consider a problem that will re-occur throughout this chapter in order to illustrate alternative solutions and techniques. It is the problem of reading a text, collecting all its words, and counting the frequency of their occurrence. It is called the construction of a *concordance*.

An obvious solution is to construct a *list* of words found in the text. The list is scanned for each word. If the word is found, its frequency count is incremented; otherwise the word is added to the list. We shall simply call this process *search*, although it may apparently also include an *insertion*.

In order to be able to concentrate our attention on the essential part of list handling, we assume that the words have already been extracted from the text under investigation, have been encoded as integers, and are available in the form of an input file.

The formulation of the procedure called *search* follows in a straightforward manner from (4.20). The variable *root* refers to the head of the list in which new words are inserted according to (4.12). The complete algorithm is listed as Program 4.1; it includes a routine for tabulating the constructed concordance list. The tabulation process is an example in which an action is executed once for each element of the list, as shown in schematic form in (4.17).

The linear scan algorithm of Program 4.1 recalls the search procedure for arrays and files, and in particular the simple technique used to simplify the loop termination condition: the use of a *sentinel*. A sentinel may as well

Program 4.1 Straight List Insertion.

```
program list (input,output);
{straight list insertion}
  type ref = ↑word;
    word = record key: integer;
                  count: integer;
                  next: ref
           end ;
  var k: integer; root: ref;

  procedure search (x: integer; var root: ref);
    var w: ref; b: boolean;
  begin w := root; b := true;
    while (w≠nil) ∧ b do
      if w↑.key = x then b := false else w := w↑.next;
    if b then
    begin {new entry} w := root; new (root);
      with root↑ do
      begin key := x; count := 1; next := w
      end
    end else
    w↑.count := w↑.count + 1
  end {search} ;
```

```
    procedure printlist (w: ref);
    begin while w ≠ nil do
          begin writeln (w↑.key, w↑.count);
                w := w↑.next
          end
    end {printlist} ;
begin root := nil; read(k);
    while k ≠ 0 do
       begin search (k, root); read(k)
       end ;
    printlist(root)
end .
```

Program 4.1 (Continued)

be used in list search; it is represented by a dummy element at the end of the list. The new procedure is (4.21), which replaces the search procedure of Program 4.1, provided that a global variable *sentinel* is added and that the initialization of *root* is replaced by the statements

$$new(sentinel);\; root := sentinel;$$

which generate the element to be used as sentinel.

```
procedure search(x: integer; var root: ref);
   var w: ref;
begin w := root; sentinel↑.key := x;
   while w↑.key ≠ x do w := w↑.next;
   if w ≠ sentinel then w↑.count := w↑.count + 1 else
      begin {new entry} w := root; new(root);                (4.21)
         with root↑ do
         begin key := x; count := 1; next := w
         end
      end
end {search}
```

Obviously, the power and flexibility of the linked list are ill used in this example, and the linear scan of the entire list can only be accepted in cases in which the number of elements is limited. An easy improvement, however, is readily at hand: the *ordered list search*. If the list is ordered (say by increasing keys), then the search may be terminated at the latest upon encountering the first key which is larger than the new one. Ordering of the list is achieved by inserting new elements at the appropriate place instead of at the head. In effect, ordering is practically obtained free of charge. This is because of the ease by which insertion in a linked list is achieved, i.e., by making full use of its flexibility. It is a possibility not provided by the array and file structures.

(Note, however, that even in ordered lists no equivalent to the binary search of arrays is available.)

Ordered list search is a typical example of the situation described in (4.15) in which an element must be inserted *ahead* of a given item, namely, in front of the first one whose key is too large. The technique shown here, however, differs from the one used in (4.15). Instead of copying values, two pointers are carried along in the list traversal; *w*2 lags one step behind *w*1, and it thus identifies the proper insertion place when *w*1 has found too large a key. The general insertion step is shown in Fig. 4.10. Before proceeding we

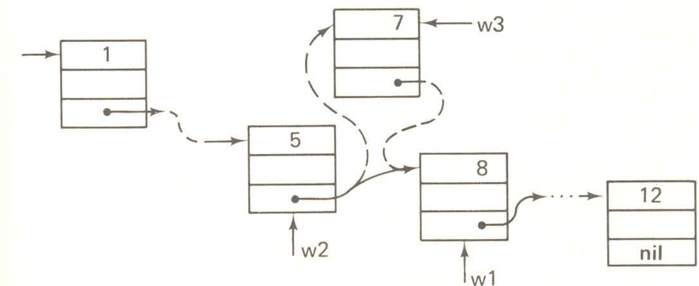

Fig. 4.10 Ordered list insertion.

must consider two circumstances:

1. The pointer to the new element (*w*3) is to be assigned to *w*2↑*.next*, except when the list is still empty. For reasons of simplicity and effectiveness, we prefer not to make this distinction by using a conditional statement. The only way to avoid this is to introduce a dummy element at the list head.
2. The scan with two pointers descending down the list one step apart requires that the list contain at least one element (in addition to the dummy). This implies that insertion of the first element be treated differently from the rest.

A proposal that follows these guidelines and hints is expressed in (4.23). It uses an auxiliary procedure *insert*, to be declared local to *search*. It generates and initializes the new element *w* and is shown in (4.22).

```
procedure insert(w: ref);
   var w3: ref;
begin new(w3);
   with w3↑ do
      begin key := x; count := 1; next := w              (4.22)
      end ;
   w2↑.next := w3
end {insert}
```

The initializing statement "*root* := **nil**" in Program 4.1 is accordingly replaced by

new(*root*); *root*↑.*next* := **nil**

Referring to Fig. 4.10, we determine the condition under which the scan continues to proceed to the next element; it consists of two factors, namely,

$$(w1\uparrow.key < x) \wedge (w1\uparrow.next \neq \textbf{nil})$$

The resulting search procedure is shown in (4.23).

```
procedure search(x: integer; var root: ref);
    var w1,w2: ref;
begin w2 := root; w1 := w2↑.next;
    if w1 = nil then insert (nil) else
    begin
        while (w1↑.key < x) ∧ (w1↑.next ≠ nil) do
            begin w2 := w1; w1 := w2↑.next                      (4.23)
            end ;
        if w1↑.key = x then w1↑.count := w1↑.count + 1 else
            insert(w1)
    end
end {search} ;
```

Unfortunately, this proposal contains a logical flaw. In spite of our care, a "bug" has crept in! The reader is urged to try to identify the oversight before proceeding. For those who choose to skip this detective's assignment, it may suffice to say that (4.23) will always push the element first inserted to the tail of the list. The flaw is corrected by taking into account that if the scan terminates because of the second factor, the new element must be inserted *after* *w*1↑ instead of *before* it. Hence, the statement "*insert*(*w*1)" is replaced by

```
begin if w1↑.next = nil then
            begin w2 := w1; w1 := nil
            end;                                                 (4.24)
        insert(w1)
end
```

Maliciously, the trustful reader has been fooled once more, for (4.24) is still incorrect. To recognize the mistake, assume that the new key lies between the last and the second last keys. This will result in both factors of the continuation condition being false when the scan reaches the end of the list, and consequently the insertion being made behind the tail element. If the same key occurs again later on, it will be inserted correctly and thus appear twice in

the tabulation. The remedy lies in replacing the condition

$$w1\uparrow.next = \textbf{nil}$$

in (4.24) by

$$w1\uparrow.key < x$$

In order to speed up the search, the continuation condition of the **while** statement can once again be simplified by using a sentinel. This requires the initial presence of a *dummy header* as well as a sentinel at the tail. Hence, the list must be initialized by the following statements

new(*root*); *new*(*sentinel*); *root*↑.*next* := *sentinel*;

and the search procedure becomes noticeably simpler as evidenced by (4.25).

```
procedure search(x: integer; var root: ref);
    var w1,w2,w3: ref;
begin w2 := root; w1 := w2↑.next; sentinel↑.key := x;
    while w1↑.key < x do
        begin w2 := w1; w1 := w2↑.next
        end ;
    if (w1↑.key = x) ∧ (w1 ≠ sentinel) then
        w1↑.count := w1↑.count + 1 else                          (4.25)
    begin new(w3); {insert w3 between w1 and w2}
        with w3↑ do
            begin key := x; count := 1; next := w1
            end ;
        w2↑.next := w3
    end
end {search}
```

It is now high time to ask what gain can be expected from ordered list search. Remembering that the additional complexity incurred is small, one should not expect an overwhelming improvement.

Assume that all words in the text occur with equal frequency. In this case the gain through lexicographical ordering is indeed also nil, once all words are listed, for the position of a word does not matter if only the *total* of all access steps is significant and if all words have the same frequency of occurrence. However, a gain is obtained whenever a new word is to be inserted. Instead of first scanning the entire list, on the average only half the list is to be scanned. Hence, ordered list insertion pays off only if a concordance is to be generated with many distinct words compared to their frequency of occurrence. The preceding examples are therefore suitable primarily as programming exercises rather than for practical applications.

The arrangement of data in a linked list is recommended when the number of elements is relatively small (say < 100), varies, and, moreover, when no information is given about their frequencies of access. A typical example is the symbol table in compilers of programming languages. Each declaration causes the addition of a new symbol, and upon exit from its scope of validity, it is deleted from the list. The use of simple linked lists is appropriate for applications with relatively short programs. Even in this case a considerable improvement in access method can be achieved by a very simple technique which is mentioned here again primarily because it constitutes a pretty example for demonstrating the flexibilities of the linked list structure.

A characteristic property of programs is that occurrences of the same identifier are very often clustered, that is, one occurrence is often followed by one or more re-occurrences of the same word. This information is an invitation to re-organize the list after each access by moving the word that was found to the top of the list, thereby minimizing the length of the search path the next time it is sought. This method of access is called *list search with re-ordering*, or—somewhat pompously—*self-organizing list search*. In presenting the corresponding algorithm in the form of a procedure which may be substituted in Program 4.1, we take advantage of our experience made so far and introduce a sentinel right from the start. In fact, a sentinel not only speeds up the search, but in this case it also simplifies the program. The list is, however, not initially empty, but contains the sentinel element already. The initial statements are

new (*sentinel*); *root* := *sentinel*;

Note that the main difference between the new algorithm and the straight list search (4.21) is the action of re-ordering when an element has been found. It is then detached or deleted from its old position and re-inserted at the top. This deletion again requires the use of two chasing pointers, such that the predecessor $w2\uparrow$ of an identified element $w1\uparrow$ is still locatable. This, in turn, calls for the special treatment of the first element (i.e., the empty list). To conceive the re-linking process, we refer to Fig. 4.11. It shows the two pointers

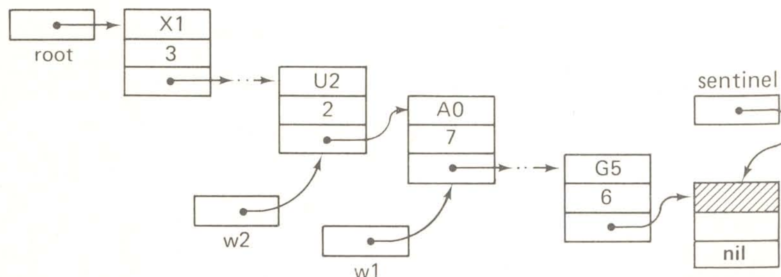

Fig. 4.11 List before re-ordering.

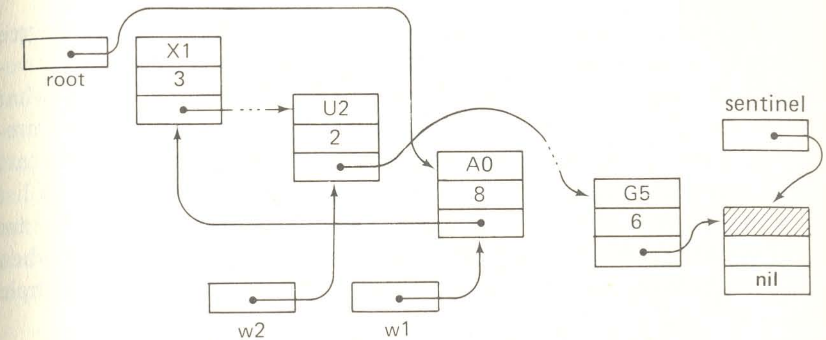

Fig. 4.12 List after re-ordering.

when $w1\uparrow$ was identified as the desired element. The configuration after correct re-ordering is represented in Fig. 4.12, and the complete new search procedure is shown in (4.26).

```
procedure search(x: integer; var root: ref);
   var w1,w2: ref;
begin w1 := root; sentinel↑.key := x;
   if w1 = sentinel then
   begin {first element} new(root);
      with root↑ do
         begin key := x; count := 1; next := sentinel
         end
   end else
   if w1↑.key = x then w1↑.count := w1↑.count + 1 else
   begin {search}
      repeat w2 := w1; w1 := w2↑.next
      until w1↑.key = x;                                        (4.26)
      if w1 = sentinel then
      begin {insert}
         w2 := root; new(root);
         with root↑ do
            begin key := x; count := 1; next := w2
            end
      end else
      begin {found, now reorder}
         w1↑.count := w1↑.count + 1;
         w2↑.next := w1↑.next; w1↑.next := root; root := w1
      end
   end
end {search}
```