LIST PROCESSING

Lists are very flexible data structures that are suitable for a large number of applications. The main advantage of lists is that they organize computer memory into an elastic object that can be allocated on demand in various amounts and shapes during the running of a program. Lists can be lengthened by the insertion of new elements or by adjoining a new list to a given list. Also, one list could be embedded inside another list, creating a two-dimensional list. Embedding can be performed repeatedly, producing structures of unrestricted depth. Thus, lists can grow both in length and depth by nesting. Lists can also shrink. Elements or embedded lists can be deleted and lists can be broken into constituent parts.

Lists have interesting growth, combining, and decay properties that permit them to change size and shape dynamically under programmed control. By contrast, other methods of storage allocation must allocate storage in a fixed size and shape before a program is run, and during program execution, size and shape either cannot vary or are costly to vary. This happens, for instance, with the allocation of tables, arrays, and record structures in some programming languages. Often, in order to use these structures advantageously, one must be able to predict demand for storage before a program is run so that one can preallocate a sufficient amount of space.

SAMPLE APPLICATIONS OF LISTS

With lists, one need not preallocate the size or shape of storage structures. This property makes lists ideal for applications whose natural information requirements grow and shrink unpredictably, and whose parts change shape and combine with each other in ways that cannot be forecast easily.

For example, in symbolic formula manipulation, subexpressions may be nested within expressions to an unpredictable depth, and the number of terms of a formula may grow without limit. Therefore, lists are natural to use. Also, lists can absorb overflows in a table of fixed size, since lists can grow to meet unforeseen demand for table space. This could be done by making the last element of a table a pointer to the overflow list. Lists may also be useful in devising efficient algorithms in which they can be used to keep track of internal information at intermediate stages in the execution of a process. For reasons such as these, lists and list structures are an important topic in the study of data structure.

FORMAL DEFINITION OF LISTS

A list is a finite ordered sequence of items $(x_1, x_2, \ldots, x_n)$ where $n \geq 0$. The list ( ) of no items occurs as a special case where $n = 0$, and is called the *empty list.* The empty list is denoted by the symbol Λ. The items x_i $(1 \leq i \leq n)$ in a list can be arbitrary in nature. In particular, it is possible for a given list to be an item in another list. For example, let L be the list $[(x_1, x_2, (y_1, y_2, y_3), x_4).]$ Then, the third item of L is the list (y_1, y_2, y_3). In this case we say (y_1, y_2, y_3) is a *sublist of L.* If a list L has one or more sublists, we say that L is a *list structure.* If a list has no sublists, we call it either a *linear list* or a *chain.*

TYPES OF LISTS

As might be expected, there are a number of different possible underlying representations for lists, each with particular advantages and disadvantages. Three broad classes are sequentially allocated lists, linked lists, and associate lists. We devote the most attention to linked lists, because they are the richest in terms of variety and they possess a number of special cases such as one-way linked lists, symmetrically linked lists, and circular lists.

To support linked-list representations, memory is organized into cells, and unused cells are linked together into a list of available (or unallocated) cells. As demands for storage arise, cells are removed from the list of available space and are added to the structures in use. It is also possible for various list cells to become disconnected from the set currently in use by a program, and such cells may be reclaimed and used again.

Sequentially Allocated Lists

Let $L = (x_1, x_2, \ldots, x_n)$ be a linear list with elements x_i $(1 \leq i \leq n)$, where each element requires one word to be represented in memory. In sequential allocation, the representations of the items x_i are stored consecutively in memory beginning at a certain address α, as shown in Fig. 1.

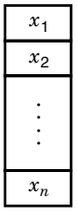

Figure 1. A sequentially allocated list.

In general, we can store x_i in location $\alpha + i - 1$ $(1 \leq i \leq n)$. As immediate generalizations, the items x_i might each take k words, and we could store L either in ascending or in descending order of addresses, so that item x_i would be stored in the k words beginning at address $\alpha + k(i - 1)$ for ascending order and $\alpha - k(i - 1)$ for descending order.

If the items x_i have nonuniform sizes, we may store them contiguously, as long as we provide some means for recognizing the boundaries between adjacent elements. For example, we could mark the first word of each item specially, or we could store the number of words per item in a special field in each item. However, such storage policies entail loss of ability to perform direct arithmetic indexing of list elements.

Sequential representations of lists save space at the expense of element access time and increased cost of growth and decay operations. Further, managing the growth of several sequentially allocated lists at or near saturation of the available memory is quite costly.

Linked Allocation for Lists

Linked allocation of list structures provides a natural way of allocating storage for lists that conveniently accommodates growth and decay properties, as well as certain natural traversals of the elements. The cost of such representations is borne in increased expense for access to arbitrary elements and in a reduction in storage utilization efficiency because of the extra space needed to hold links.

Unidirectional Linked Allocation. Let $L = (x_1, x_2, \ldots, x_n)$ be a list. Let α_i $(1 \leq i \leq n)$ be a set of distinct addresses of memory cells. We consider only cells of uniform size. The *link* fields of a particular cell are assumed to be fields that hold addresses of other cells. Thus, relationships such as contiguity of elements in a list, or sublists of a list, can be represented by explicit links stored in particular fields of cells. This permits contiguous or nested listed elements to be represented in nonadjacent cells in memory.

Since the possibilities for such representation schemes are numerous, we give various illustrations in the hope that the reader will be able to generalize to a set of linked representations themes of general utility that can be adapted to the peculiarities of the many machine environments and many representation requirements encountered in practice.

The simplest form is illustrated in Fig. 2. Each cell has two fields, an INFO field containing an item x_i in L and a LINK field containing an address of another cell. The LINK field of the last cell α_n contains a null address. This representation is usually drawn as shown in Fig. 3.

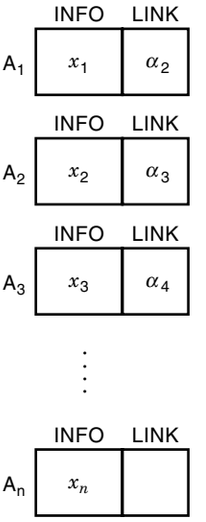

Figure 2. A linear linked list.

To represent the more general case of list structures, as opposed to linear lists, we must enlarge upon the idea illustrated in Figs. 2 and 3. For example, suppose we are fortunate enough to have representations of atoms and pointers that take identical amounts of space. Then, to represent list structures with items that are either sublists or atoms x_i, we can store either an atom x_i or the address of a sublist in the INFO field of a given cell. However, we now need a systematic way to tell whether the INFO field contains the address of a sublist or an atom directly. For example, in Fig. 4, a TAG field containing "+" is used to indicate that the content of the INFO field is an atom x_i, and a TAG field containing "−" is used to indicate that the INFO field contains the address of a sublist of the list.

Figure 4 shows how the list structure $[x_1, x_2, (y_1, y_2, y_3), x_4]$ is represented using these conventions. In many cases, all bits in a cell are required to contain efficient machine representations of such atoms as integers, floating point numbers and so forth; and there is no space left for a tag bit. Under these circumstances, we can have space for both an atom field and a sublink field (SUBLINK), and only one can be used at a time. We would need an additional tag field for this representation.

Symmetrically Linked Allocation. Consider the diagram in Fig. 5. In this diagram, each cell contains links to its left and right neighbors (except for the first cell which has no left neighbor, and the last cell, which has no right neighbor). Each cell has an INFO field, which contains an item x_i, and two address fields, LEFT LINK and RIGHT LINK. Such a structure is called a symmetrically linked list.

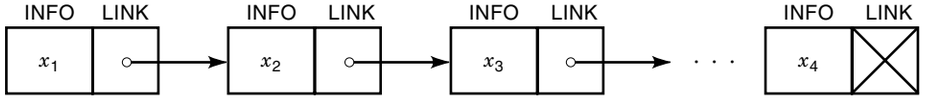

Figure 3. Graphical representation of a linear linked list.

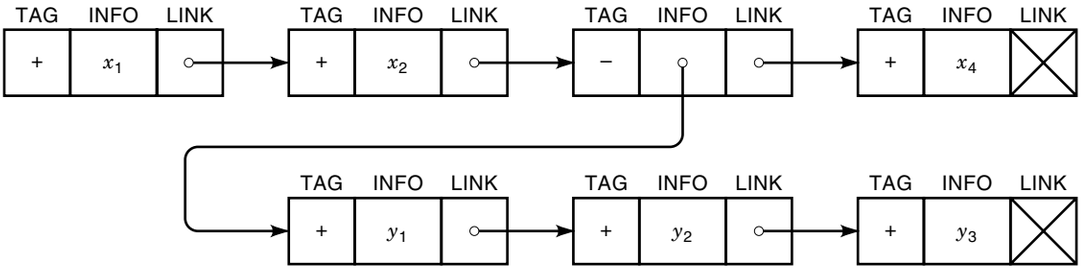

Figure 4. Linked representation of a list structure.

It is easy to traverse a symmetric list in either direction (forwards or backwards), starting from the address of any cell. By contrast, given the address of an arbitrary cell in a one-way list, traversal is possible only in the forward direction. Similarly, if a is the address of an arbitrary cell on a symmetric list S, one can delete cell a of S, or one can insert a new cell before or after cell a of S easily without traversing the list. Fig. 6 shows how to delete cell C from S.
To delete a cell C from a symmetric list S:

1. If LEFT LINK(C) <>A then RIGHT LINK[LEFT LINK(C)] ← RIGHT LINK(C)
2. If RIGHT LINK(C) <>A then LEFT LINK[RIGHT LINK(C)] ← LEFT LINK(C)
3. Clean up: LEFT LINK(C) ← RIGHT LINK(C) ← A

Again, by contrast, if α is the address of an arbitrary cell on a one-way linked list L, it is not possible to delete cell α from L, or insert a new cell before cell α unless we have the address of the first cell in the list (the header). As shown above, one must pay extra space for the extra flexibility of symmetric lists, since each cell of a symmetric cell has two address fields instead of one.

To make list structures composed from symmetrically linked cells, it is convenient to use special header cells that point to the left and right ends of a symmetrically linked chain. An example of a symmetric list structure [x_1, (x_2, x_3), x_4, x_5] using header cells is given in Fig. 7.

Each header cell links to the leftmost and rightmost cells of a symmetrically linked chain and the leftmost and rightmost cells of the chain each link back to the header cell. A list item that points to a sublist points to the header for the sublist. The INFO field of a list header frequently can be used to contain storage management information. For example, one policy for storage reclamation is to keep a cell count in the header cell for each list. Such a reference count is an integer equal to the total number of nodes in the list. Each time a new cell is added, the cell count increases by one, and each time a cell is removed, the cell count is decremented. Whenever the cell count reaches zero, the header cell itself can be removed.

Circular Lists. Circular lists are formed by linking the last cell of a chain to the head of the chain. As illustrated in Fig. 8. Circular lists have the property that all elements can be accessed starting from any cell on the list, without incurring the overhead of two pointers per cell.

OPERATIONS ON LISTS

In this section we describe a few operations on lists. We use a unidirectional (linear) linked list because it is the most common structure.

We now introduce some notation for use in the operations. If p is the address of a node (i.e., pointer), *node*(p) refers to the node pointed to by p, *info*(p) refers to the information pointer of that node, and *link*(p) refers to the LINK field and is therefore a pointer. Thus, if *link*(p) is not *nil*, *info*[*next*(p)] refers to the information portion of the node that follows *node*(p) in the list.

Inserting Nodes in the Beginning of a List

Suppose that we are given a list of integers, as illustrated in Fig. 9(a), and we desire to add the integer 6 to the front of the list. That is, we wish to change the list so that it appears as in Fig. 9(f). The first step is to obtain a node to house the new integer. If a list is to grow and shrink, there must be some mechanism for obtaining new nodes to add. Let us assume the existence of a mechanism for obtaining empty nodes. The operation

p := getnode

obtains an empty node and sets the contents of a variable named p to that address. This means that p is a pointer to this newly allocated node, as illustrated in Fig. 9(b).

The next step is to insert the integer 6 into the INFO field of the newly created node p. This is done by the operation

info(p) = 6

The result of this operation is shown in Fig. 9(c).

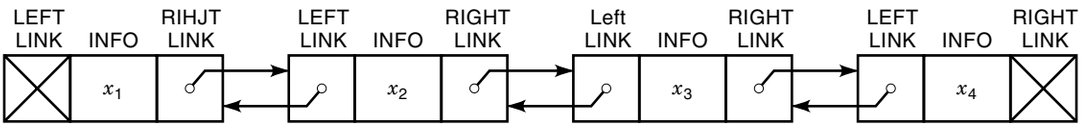

Figure 5. A symmetrically linked list.

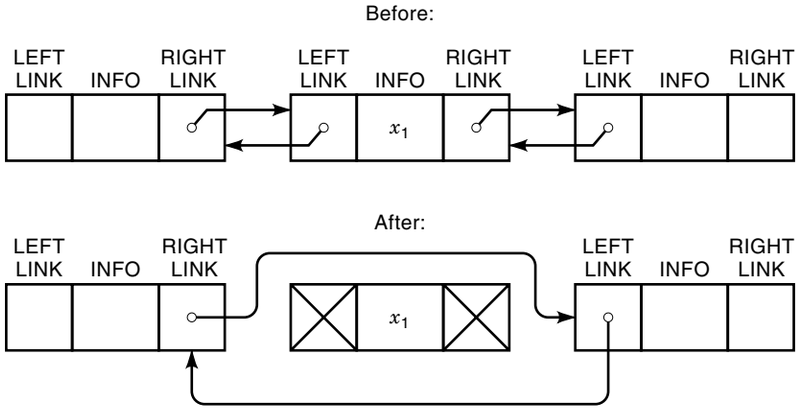

Figure 6. Deleting a node from a symmetrically linked list.

After setting the INFO portion of *node*(p), it is necessary to set the LINK portion. Since *node*(p) is to be inserted at the front of the list, the node that follows should be the current first node of the list. Since the variable L contains the address of that first node, *node*(p) can be added to the list by performing the operation

$$link(p) = L$$

This operation places the value of L (which is the address of the first node on the list) in the link field of *node*(p). Figure 9(d) illustrates the result of this operation.

At this point, p points to the list with the additional item included. However, since L is the "official" external pointer to the list, its value must be modified to the address of the new first node of the list. This can be done by performing the operation

$$L = p$$

which changes the value of L to the value of p. Figure 9(e) illustrates the results of this operation. Note that Figs. 9(e) and 9(f) are identical except that the value of p is not shown in Fig. 9(f). This is because p is used as an auxiliary variable during the process of modifying the list, but its value is irrelevant to the status of the list before and after the process.

Putting all the steps together, we have an algorithm for adding the integer 6 to the front of the list L:

p = getnode
$info(p) = 6$
$link(p) = L$
$L = p$

Deleting the First Node of a List

Figure 10 illustrates the process of removing the first node of a nonempty list and storing the value of its info field into a variable x. The initial configuration is in Fig. 10(a) and the

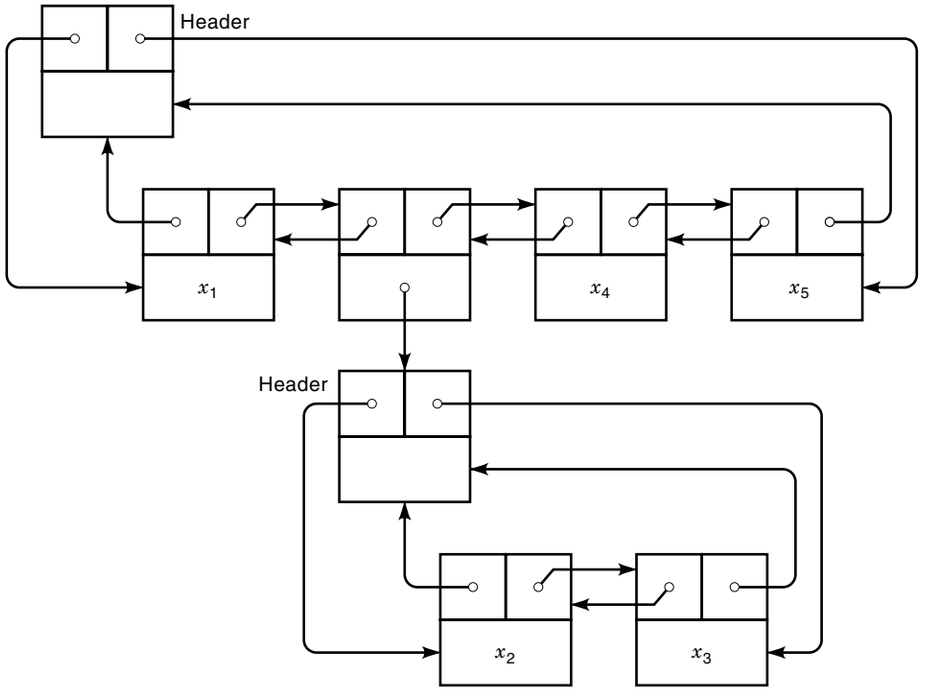

Figure 7. A symmetrical list structure.

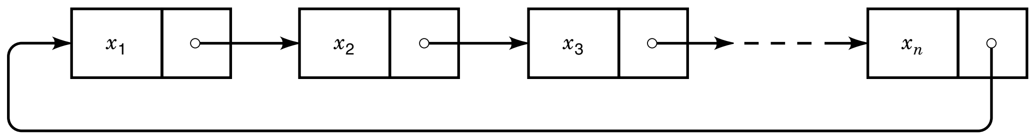

Figure 8. A circular linked list.

final configuration is in Fig. 10(e). The process itself is almost the exact opposite of the process to add a node to the front of a list. The algorithm is as follows:

$p = L$ [Fig. 10(b)]
$L = link(p)$ [Fig. 10(c)]
$x = info(p)$ [Fig. 10(d)]
freenode(p) [Fig. 10(e)]

The operation freenode(p) will make node p available for reuse by adding it to the empty cells list. Once this operation has been performed, it is illegal to reference *node*(p), since the

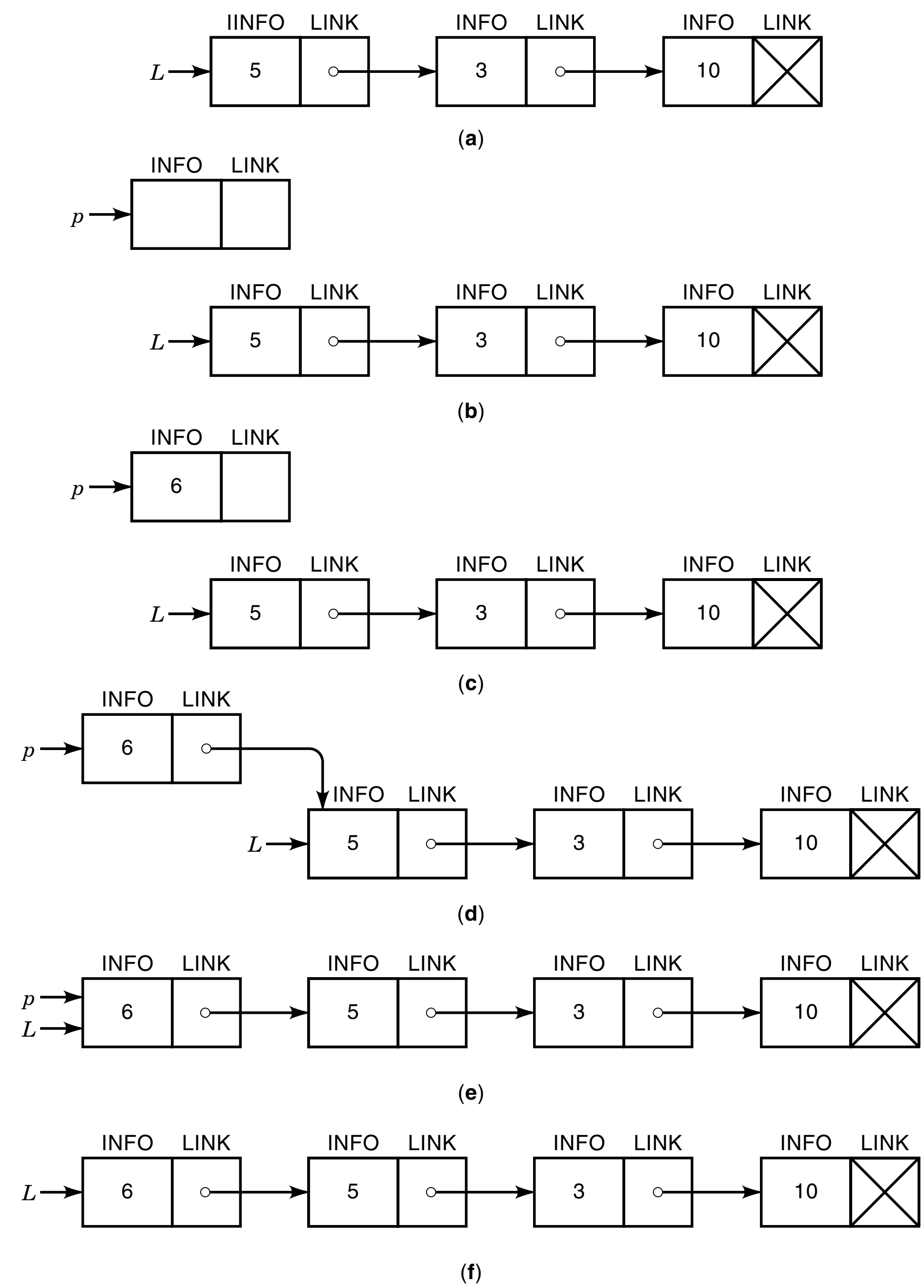

Figure 9. Adding a node to the beginning of a list.

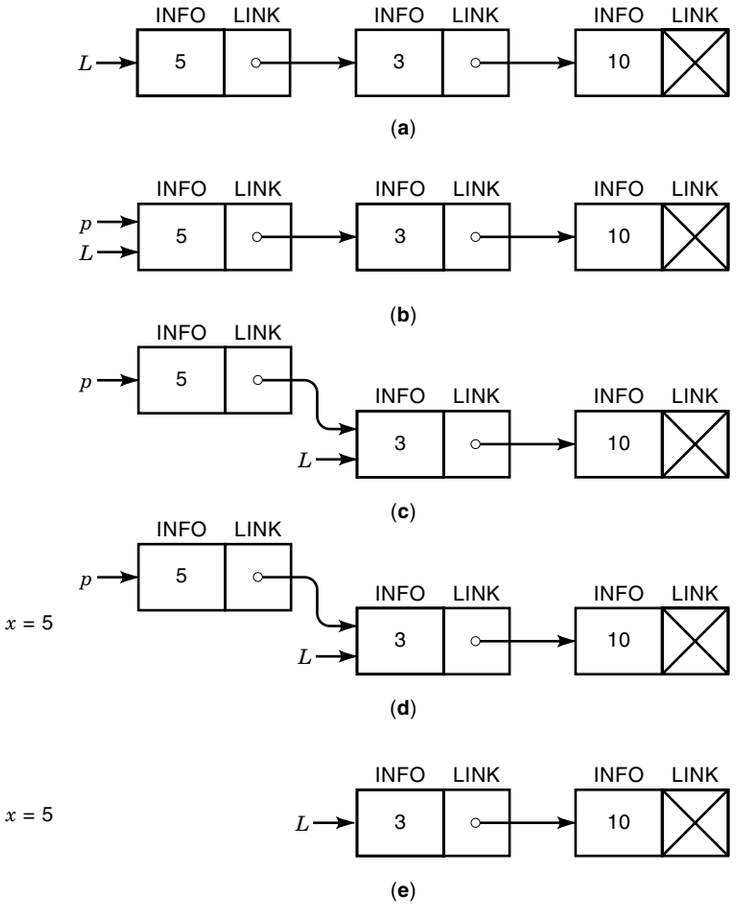

Figure 10. Removing the first node of a list.

node is no longer allocated. Since the value of p is a pointer to a node that has been freed, any reference to that value is also illegal.

Adding a Node to the Middle of a List

The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements. For example, suppose that we wished to insert an item x between the third and fourth elements in an array of size 10 that currently contains 7 items. Items 7 through 4 must first be moved one slot and the new element inserted in the newly available position 4. In this case, insertion of one item involves moving four items in addition to the insertion itself. If the array contained 500 or 1000 elements, a correspondingly larger number of elements would have to be moved. Similarly, to delete an element from an array, all the elements past the element deleted must be moved one position.

On the other hand, if the items are stored in a list, then if p is a pointer to a given element of the list, inserting a new element after *node*(p) involves allocating a node, inserting the information, and adjusting two pointers. The amount of work required is independent of the size of the list. This is illustrated in Fig. 11.

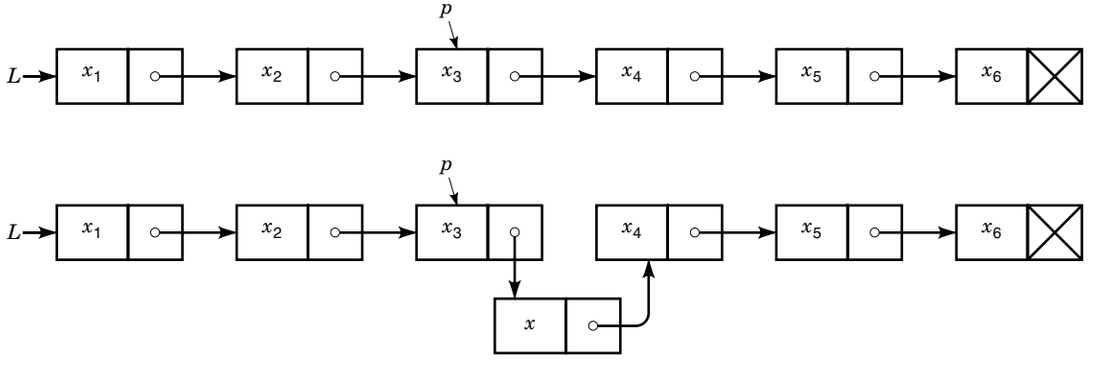

Figure 11. Adding a node in the middle of a list.

Let *insafter*(*p*,*x*) denote the operation of inserting an item *x* into a list after a node pointed to by *p*, This operation may be implemented as follows:

```
q = getnode
info(q) = x
link(q) = link(p)
link(p) = q
```

Before inserting a new node, we will need to traverse the list to reach node *p*. List traversal is a very common operation. For example, suppose we want to insert the new node after the first node we find with an INFO value of 100 if it exists. Therefore *p* should point to that node or be *nil* if the list is empty or that node is not found. The operation goes as follows:

```
p = L
/* traverse list until a node with info = 100 is found */
while (p <> nil) and (info(p) <> 100) do
  p = link(p)
/* insert new node after p */
if p nil
  then insafter(p,x)
```

List Traversal

This is the simplest of operations, we just need to start at the list header and follow the LINK field to the end of the list. This example counts the number of nodes in the list and keeps that value in an integer variable *count:*

```
p = L
count = 0
while (p <> nil)
  begin
    count = count + 1
    p = link(p)
  end
```

We start by initializing the counter to 0, and setting *p* to the first node in the list. Then we start traversing and incrementing the counter with each step. The operation

```
p = link(p)
```

is the key operation here. It sets the pointer *p* to the following node in the list using the LINK field.

Erasing a List

This operation is an extension of the process of deleting the first node in a list, as explained in a previous section. We delete the first node in the list repeatedly until there are no more nodes:

```
while (L <> nil)
  begin
    p = L
    L = link(L)
    freenode(p)
  end
```

Reading List

A. Aho, J. Hopcroft, and J. Ullman, *Data Structures and Algorithms,* Reading, MA: Addison-Wesley, 1983.

G. Gonnet, *Handbook of Algorithms and Data Structures,* Reading, MA: Addison-Wesley, 1984.

D. Knuth, *Fundamental Algorithms,* volume 1 of *The Art of Computer Programming,* Reading, MA: Addison-Wesley, 1968.

D. Knuth, *Sorting & Searching,* volume 3 of *The Art of Computer Programming,* Reading, MA: Addison-Wesley, 1973.

SAMAH A. SENBEL
Old Dominion University