2. The List

We will deal with a general list of the form $a_1, a_2, a_3, \ldots, a_n$. We say that the size of this list is $n$. We will call the special list of size 0 a null list.

For any list except the null list, we say that $a_i$ follows (or succeeds) $a_i (i < n)$ and that $a_{i-1}$ precedes $a_i (i > 1)$. The first element of the list is $a_1$, and the last element is $a_n$. We will not define the predecessor of $a_1$ or the successor of $a_n$. The position of element $a_i$ in a list is $i$. Throughout this discussion, we will assume, to simplify matters, that the elements in the list are integers, but in general, arbitrarily complex elements are allowed.

Associated with these "definitions" is a set of operations that we would like to perform on the list ADT. Some popular operations are print_list and make_null, which do the obvious things; find, which returns the position of the first occurrence of a key; insert and delete, which generally insert and delete some key from some position in the list; and find_kth, which returns the element in some position (specified as an argument). If the list is 34, 12, 52, 16, 12, then find(52) might return 3; insert(x,3) might make the list into 34, 12, 52, x, 16, 12 (if we insert after the position given); and delete(3) might turn that list into 34, 12, x, 16, 12.

Of course, the interpretation of what is appropriate for a function is entirely up to the programmer, as is the handling of special cases (for example, what does find(1) return above?). We could also add operations such as next and previous, which would take a position as argument and return the position of the successor and predecessor, respectively.

2.1. Simple Array Implementation of Lists

Obviously all of these instructions can be implemented just by using an array. Even if the array is dynamically allocated, an estimate of the maximum size of the list is required. Usually this requires a high over-estimate, which wastes considerable space. This could be a serious limitation, especially if there are many lists of unknown size.

An array implementation allows print_list and find to be carried out in linear time, which is as good as can be expected, and the find_kth operation takes constant time. However, insertion and deletion are expensive. For example, inserting at position 0 (which amounts to making a new first element) requires first pushing the entire array down one spot to make room, whereas deleting the first element requires shifting all the elements in the list up one, so the worst case of these operations is $O(n)$. On average, half the list needs to be moved for either operation, so linear time is still required. Merely building a list by $n$ successive inserts would require quadratic time.
Because the running time for insertions and deletions is so slow and the list size must be known in advance, simple arrays are generally not used to implement lists.

2.2. Linked Lists

In order to avoid the linear cost of insertion and deletion, we need to ensure that the list is not stored contiguously, since otherwise entire parts of the list will need to be moved. Figure 3.1 shows the general idea of a linked list.

The linked list consists of a series of structures, which are not necessarily adjacent in memory. Each structure contains the element and a pointer to a structure containing its successor. We call this the next pointer. The last cell's next pointer points to ; this value is defined by C and cannot be confused with another pointer. ANSI C specifies that is zero.

Recall that a pointer variable is just a variable that contains the address where some other data is stored. Thus, if \( p \) is declared to be a pointer to a structure, then the value stored in \( p \) is interpreted as the location, in main memory, where a structure can be found. A field of that structure can be accessed by \( p \) \( \text{field\_name} \), where \( \text{field\_name} \) is the name of the field we wish to examine. Figure 3.2 shows the actual representation of the list in Figure 3.1. The list contains five structures, which happen to reside in memory locations 1000, 800, 712, 992, and 692 respectively. The next pointer in the first structure has the value 800, which provides the indication of where the second structure is. The other structures each have a pointer that serves a similar purpose. Of course, in order to access this list, we need to know where the first cell can be found. A pointer variable can be used for this purpose. It is important to remember that a pointer is just a number. For the rest of this chapter, we will draw pointers with arrows, because they are more illustrative.

![Figure 2.1 A linked list](image)

![Figure 2.2 Linked list with actual pointer values](image)
To execute print_list(L) or find(L,key), we merely pass a pointer to the first element in the list and then traverse the list by following the next pointers. This operation is clearly linear-time, although the constant is likely to be larger than if an array implementation were used. The find_kth operation is no longer quite as efficient as an array implementation; find_kth(L,i) takes $O(i)$ time and works by traversing down the list in the obvious manner. In practice, this bound is pessimistic, because frequently the calls to find_kth are in sorted order (by i). As an example, find_kth(L,2), find_kth(L,3), find_kth(L,4), find_kth(L,6) can all be executed in one scan down the list.

The delete command can be executed in one pointer change. Figure 3.3 shows the result of deleting the third element in the original list.

The insert command requires obtaining a new cell from the system by using an malloc call (more on this later) and then executing two pointer maneuvers. The general idea is shown in Figure 3.4. The dashed line represents the old pointer.