many data moves) and wastes memory, so it is more efficient to leave NewSub$ ( ) unsorted, and create Index ( ) instead. Now a new name, Bull, has to be added to the file, so the arrays look like this:

NewSub$ ( )	Index ( )
(1) Jones	(2)
(2) Atkins	(7)
(3) Carter	(3)
(4) Rogers	(6)
(5) Smith	(1)
(6) Drake	(4)
(7) Bull	(5)

Notice that the contents of Index ( ) above the new insertion are unchanged, and its contents below the insertion are in the same order as previously, but have all been moved one place down in the array. Insertion to an index therefore requires: finding the position of the new element, moving every element between there and the end of the index down one, and writing in the new entry. This is preferable to doing the same thing with the actual data, NewSub$, but is still relatively slow, if the index is large.

Suppose, now, that we structure the data in a different way. Leave NewSub$ ( ) unsorted because manipulating it is slow and expensive, and establish a parallel array called LookUp ( ), whose contents are simply numbers referring to positions in NewSub$ ( ).

ListHead (2)

NewSub$ ( )	LookUp ( )	Index ( )
(1) Jones	(4)	(2)
(2) Atkins	(3)	(3)
(3) Carter	(6)	(6)
(4) Rogers	(5)	(1)
(5) Smith	(0)	(4)
(6) Drake	(1)	(5)

The first difference is that a simple variable called ListHead is needed: it points to NewSub$ (2) which is alphabetically the first element of NewSub$ ( ). The next difference is that the number (0) has been used in LookUp (5): this indicates that NewSub$ (5) is alphabetically the last element of the array.

The next difference is the contents of Index ( ) and LookUp ( ). Index ( ) has to be read: 'the first element is in NewSub$ (2), the second is in NewSub$ (3), the third is in NewSub$ (6)'...etc. while ListHead ( ) is read: 'the first element is in NewSub$ (2); Then LookUp (2) says that the next element is in NewSub$ (3); LookUp (3) says that the next element is in NewSub$ (6); and so on. LookUp (5) says that NewSub$ (5) is the last element.

Index ( ) gives an absolute position for elements of the file, while LookUp ( ) gives only relative positions — any item in LookUp ( ) tells you only where to find the next element, and says nothing about absolute position. The number in Index (4) points to the fourth item in the alphabetically ordered file, whereas the number in LookUp (4) points only to the item that comes after NewSub$ (4) in the ordered file. LookUp ( ) implements the data structure called a 'Linked List'. Reading a Linked List is like following a treasure hunt: at the start you're told your first destination; when you get there you find a clue which points you to your next destination, and so on. Reading an Indexed Array is like being on a car rally: at the start you're told all your destinations and the order in which to visit them.

The great advantage of the List structure is its flexibility. Consider the List after insertion of the new element, Bull:

ListHead (2)

NewSub$ ( )	LookUp ( )
(1) Jones	(4)
(2) Atkins	(7)
(3) Carter	(6)
(4) Rogers	(5)
(5) Smith	(0)
(6) Drake	(1)
(7) Bull	(3)

The array LookUp ( ) has changed in only two places:

i) LookUp (2), which formerly pointed to NewSub$ (3) as containing the next alphabetic element after NewSub$ (2), now points to NewSub$ (7) since it is now the next alphabetic element after NewSub$ (2)

ii) LookUp (7), which was unused, now points to NewSub$(3) as the next item after NewSub$ (7) in the alphabetic ordering.

This illustrates the general process of insertion to a Linked List: find the element of the list which should come just before the new element, and make that element point to the new element; then make the new element point to the element that it has displaced. These simple operations will be all that is required for insertion to a Linked List, and only the first of these is affected by the size of the List. Inserting an element to a List is like inserting a new link into a chain — decide where to put the link, break the chain, join the preceding link to the new one, and the new link to the succeeding link. Linked Lists are sometimes called Chained Lists. The numbers in LookUp ( ) — the links — are sometimes called Pointers.

A striking feature of Lists is their strong seriality; it is impossible to find an element in a List except by starting at the beginning and inspecting every element until the target is found. The List is implemented here by using arrays, which are designed to be Direct Access structures, but the List has effectively turned them into Sequential Files. In other languages, such as LISP and PASCAL, the List facility is built-in.

Lists are useful structures for handling dynamic data (data that regularly changes), and can be powerful tools when dealing with either natural language (as in speech recognition) or artificial language (when compiling programs), where the data itself naturally forms a list of elements.