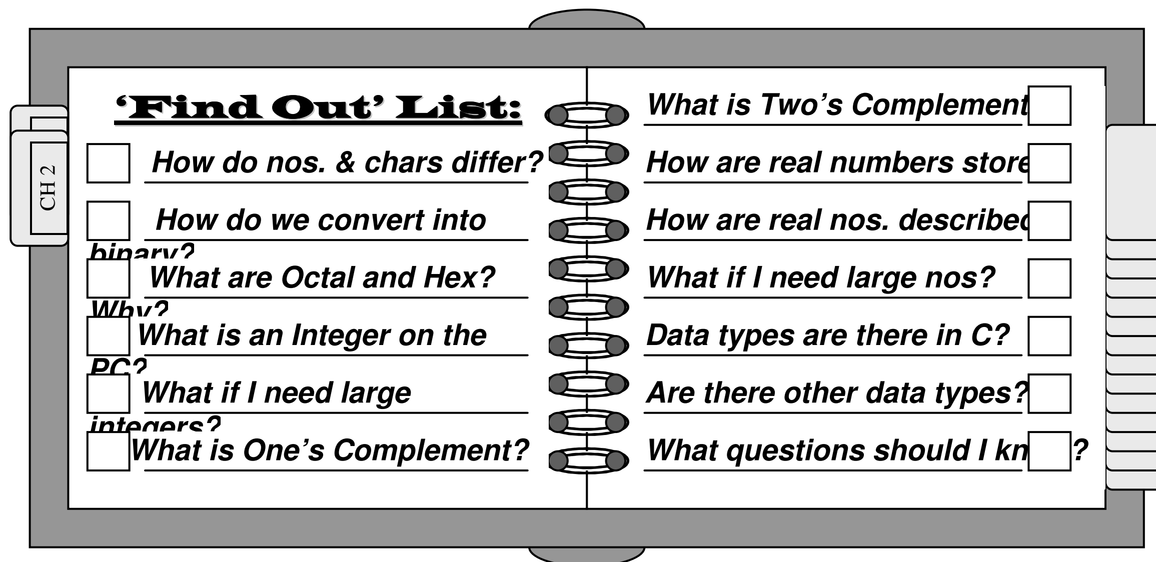
CHAPTER 10:
LINKED LISTS

"Simplicity of life, even the barest, is not a misery,
but the very foundation of refinement"
William Morris (1834–96)



Introduction

In our chapter on searching and sorting, we considered some of the trade-offs between the two techniques. These considerations take on greater significance when we consider how they affect databases: We frequently wish to view the records in different orders. For example, consider our student database (from Chapter 7, which we have duplicated as table 10.1.). Sometimes, we might wish to list students by *SSN* or *name* or *class* or *gpa* or *hrs* or *balance* or by *state* then *city* then *zip*. Essentially, we might wish a listing by any field, or combination of fields.

? ***What are we to do??? Sort the database each time we wish a new listing???***

We know that this could take a considerable amount of time, especially if it is a large database.

Table 10.1.

Field	SSN	name	street	city	state	zip	class	gpa	hrs	balance
Datatype	**char**[10]	**char**[31]	**char**[41]	**char**[26]	**char**[3]	**char**[6]	**char**	**float**	**int**	**float**
Bytes	10	31	41	26	3	6	1	4	2	4

One of the main advantages of linked lists is that we order, and quickly display, the contents of a list in various order without having to physically sort the list. As we will see, there are trade-offs involved, but then again, isn't everything a trade-off?

Another problem encountered with arrays, and especially with records, is that we don't always know exactly how much storage we will require in advance. Compounding that is the fact that we can't always get contiguous bytes of storage, even though we might have enough total storage available to us. Dynamic memory allocation (which we will discuss in the next chapter) will allow to relax our constraint that we must have a *fixed number of contiguous addresses* allocated before we begin our program.

First, However, we will see how we can order lists without having to physically sort them.

Singly Linked Lists

As we noted above, we often have a list which may (or may not) be sorted on some field, but which we frequently wish to display on some other field. Let's assume that we have a database which contains information on eight novelists (who just happen to be taking a course in data structures because they find it fascinating). The list is sorted by ID (Social Security Number, or *ssn*), but also contains the Writer's Name (let's call the field *name*), and their scores on quiz 1 (let's call it *q1*), and on quiz 2 (let's call this field *q2*). In tabular form, this might appear as:

Table 10.2.

Offset\Field	*ssn*	*name*	*q1*	*q2*
0	123-45-6789	Christie, Agatha	67	78
1	234-56-7890	Tolkien, J.R.R.	100	72
2	345-67-8901	Eliot, T.S.	94	97
3	456-78-9012	Carroll, Lewis	88	62
4	567-89-0123	Joyce, James	59	74
5	678-90-1234	Albee, Edward	79	87
6	789-01-2345	Hesse, Herman	93	88
7	890-12-3456	Beckett, Samuel	60	64

If we wished to merely store the data in our program, our structure template, and the variable which we could associate with the template, might appear as:

C/C++ Code 10.1.

```
struct writers
{    char    ssn[10], name[31];
     int     q1, q2;
};
int main()
{ struct writers novelists[8] = { . . .};
```

Each record, of course, would require 10 + 31 + 2 + 2 = 45-bytes of contiguous storage. Since we wish to store eight (8) records, we would require a total of 45 * 8 = 360 contiguous bytes of storage.

As it stands right now, the database is sorted by Social Security Number (*ssn*). What if we wished to print out the table alphabetically by name, however? We could, of course sort the list, it which case it would be stored as:

Table 10.3.

Record/Offset	*ssn*	*name*	*q1*	*q2*
0	678-90-1234	Albee, Edward	79	87
1	890-12-3456	Beckett, Samuel	60	64
2	456-78-9012	Carroll, Lewis	88	62
3	123-45-6789	Christie, Agatha	67	78
4	345-67-8901	Eliot, T.S.	94	97
5	789-01-2345	Hesse, Herman	93	88
6	567-89-0123	Joyce, James	59	74
7	234-56-7890	Tolkien, J.R.R.	100	72

? ***What's the problem???***

Nothing, really. It's just that that sometimes we might wish to display the names by ID, sometimes by name. If we must sort (and resort) each time we wish to display by a different key, well, we have already seen how much effort that can be, especially with long lists. We can, however, display the list on either key, without having to sort, through the use of pointers.

? ***How???***

Since we know that each record in the table has a base address, we could add one additional field which would be the pointer to the next record in the table. A simplistic way of looking at this might be:

Table 10.4.

Record	*ssn*	*name*	*q1*	*q2*	*nextname*

0	123-45-6789	Christie, Agatha	67	78	Eliot, T.S.
1	234-56-7890	Tolkien, J.R.R.	100	72	** No One **
2	345-67-8901	Eliot, T.S.	94	97	Hesse, Herman
3	456-78-9012	Carroll, Lewis	88	62	Christie, Agatha
4	567-89-0123	Joyce, James	59	74	Tolkien, J.R.R.
5	678-90-1234	Albee, Edward	79	87	Beckett, Samuel
6	789-01-2345	Hesse, Herman	93	88	Joyce, J.
7	890-12-3456	Beckett, Samuel	60	64	Carroll, Lewis

Now, if we wished to display the names in this list, we would start with the 1st person on the list ("Albee, E."), display the record information, and then look at the name in the *nextname* field ("Beckett, Samuel"), go to that record, and display the information contained in it. Next, looking at the name in Beckett's *nextname* field ("Carroll, Lewis"), we would go to that record, and display the information contained in it. We would continue the process until we got to Tolkien's record. Since Tolkien's *nextname* field points to no one, we know we are the end of the list.

We would have to keep track of that information by storing the base address of the first record (i.e., the address at which Albee's information begins) in some separate location. Let's look at the situation in more applicable terms. Assume that the base address of the database were 12450. In other words, the list might appear as given in Table 10.5.

Notice we can readily calculate the base address of any record in the database given the base address of the database (12450) since we know that each record requires 45 bytes of storage. The address of the 5th record (offset 4) must, for example, be:

$$12450 + 4 * 45 = 12450 + 180 = 12630$$

which, as we can see, it is. Now, if instead of adding a field which gives the name of the next individual on the list, we add a pointer field which contains the address of the next person on the list, we might see something like Figure 10.1.

Table 10.5.

Address	Record	*ssn*	*name*	*q1*	*q2*
12450	**0**	123-45-6789	Christie, Agatha	67	78
12495	**1**	234-56-7890	Tolkien, J.R.R.	100	72
12540	**2**	345-67-8901	Eliot, T.S.	94	97
12585	**3**	456-78-9012	Carroll, Lewis	88	62
12630	**4**	567-89-0123	Joyce, James	59	74
12675	**5**	678-90-1234	Albee, Edward	79	87
12720	**6**	789-01-2345	Hesse, Herman	93	88
12765	**7**	890-12-3456	Beckett, Samuel	60	64

Figure 10.1.

Addres s	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12744
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499
12695	**5**	678-90-1234	Albee, Edward	79	87	12793
12744	**6**	789-01-2345	Hesse, Herman	93	88	12646
12793	**7**	890-12-3456	Beckett, Samuel	60	64	12597

? ***Wait!!! The base addresses of each of the records have changed!!!***

Yes, this is true. Each record now contains an extra field (a pointer field). It would be necessary to modify our structure template so that it would appear as:

C/C++ Code 10.2.

```
struct writers
{    char    ssn[10], name[31];
     int     q1, q2;
     struct writers * next;
};
```

Which means that each record now requires an additional 4 bytes (the size of a pointer) of contiguous storage in RAM, for a total of 49-bytes per record, and 8 * 49 = 392 bytes for the array.

? ***How can we define a structured data object which has a pointer to itself???***

Why not? A pointer is nothing more than a location in memory that contains an address and points to a predefined data type. In this case, if we go to the address contained in field *next*, we expect to find that the 1st 10 bytes contain a string, the 2nd 31 bytes also contain a string, the next 2 bytes contain an integer, the following 2 bytes contain another integer, and the remaining 4 bytes contain an address.

In the above example, how did we know that the first name (alphabetically) on the list was Albee???

We need to store this information in a separate location (variable). In this case, we would store the base address of the 1st record on the list (according to how we wish to order it). The variable, of course, would be a pointer of data type **struct writers**. The declaration (in function main) might appear as (we will eventually store the address of our 1st record in variable first):

C Code 10.3.

```
int main()
{ struct writers novelists[8] = { {“123456789”,”Christie, Agatha”, 67,78,NULL},
          {“234567890”,“Tolkien, J.R.R.”,100,72,},{“345678901”,”Eliot, T.S.”,94,97,},
          {“456789012”,”Carroll, Lewis”,88,62,},{“567890123”,”Joyce, James”,59,74,},
          {“678901234”,“Albee, Edward”,79,87,},{“789012345”,”Hesse, Herman”,93,88,},
          {“890123456”,”Beckett, Samuel”,60,64,} },
* first;
```

Notice that we do not initialize the pointer field next, since we do not know in advance where they will be stored in RAM. If we were to look into RAM (again, assuming the base address is 12450), we might see (Table 10.6):

But the 1st record on the list has the next field initialized to NULL. What is NULL, anyway???

We initialized the 1st record because of manner in which we will set up the linked list (stay tuned, we will go over the procedure shortly). NULL means null pointer value (i.e., nothing). It is defined in a number of header files, one of which we must include. We will use it to indicate that the record does not point to any other records (it is the end of the list), as we did in Figure 10.1. when we had Tolkien pointing to NULL.

Table 10.6.

Offset\Field	*ssn*	*name*	*q1*	*q2*	*next*
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
0	"123456789\0"	"Christie, Agatha\0"	67	78	NULL
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
1	"234567890\0"	"Tolkien, J.R.R.\0"	100	72	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
2	"345678901\0"	"Eliot, T.S.\0"	94	97	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
3	"456789012\0"	"Carroll, Lewis\0"	88	62	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
4	"567890123\0"	"Joyce, James\0"	59	74	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
5	"678901234\0"	"Albee, Edward\0"	79	87	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
6	"789012345\0"	"Hesse, Herman\0"	93	88	----
	12450-12459	12460-12490	12491-12492	12493-12494	12495-12498
7	"890123456\0"	"Beckett, Samuel\0"	60	64	----

? ***How do we go about setting up the list so we can display it in alphabetical order???***

Establishing a Linked List

Let's go over general procedure needed. Suppose that we already had the beginnings of an ordered list. Suppose that we moved the first record from the unordered list and made it the sole record on the ordered list. We now have two lists:

Ordered List

Figure 10.2.

First
12450

→

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	NULL

Unordered List

Table 10.7.

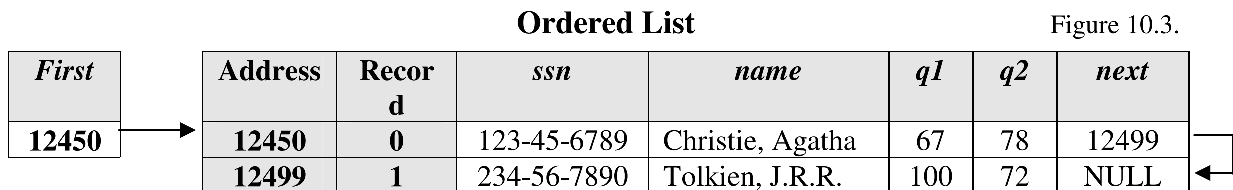
Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	-----
12548	**2**	345-67-8901	Eliot, T.S.	94	97	-----
12597	**3**	456-78-9012	Carroll, Lewis	88	62	-----
12646	**4**	567-89-0123	Joyce, James	59	74	-----
12695	**5**	678-90-1234	Albee, Edward	79	87	-----
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

Notice that we keep track of the first record on the ordered list by placing its address in pointer variable *first*. Since the record is also the only record in our ordered list, we know that it must also be the last record on the list, and therefore it points to no one (i.e., the address stored in pointer field next is NULL).

Now, the question becomes, *How do we add records to the list? If we wished to add the next record* ("Tolkien, J.R.R.") *from the unordered list onto the ordered list, where should it go?*

When we compare the name "Tolkien, J.R.R." with "Christie, Agatha", we know that it should follow it. We can readily add the name to the list by having "Christie, Agatha" point to "Tolkien, J.R.R.", and having "Tolkien, J.R.R." point to no one (NULL) since it is now the last record on the list. Our lists would now appear as:

Ordered List Figure 10.3.



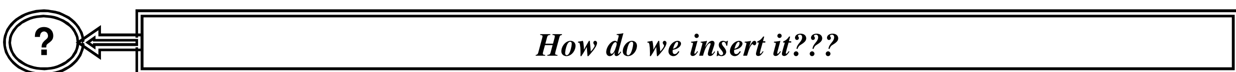
First
12450

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12499
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL

Unordered List Table 10.8.

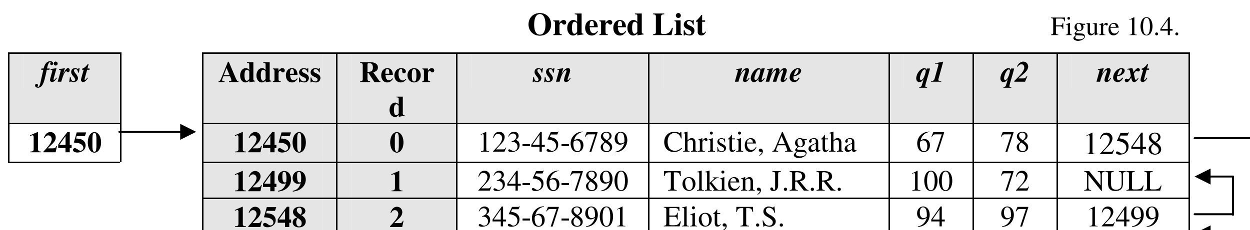
Address	Record	*Ssn*	*name*	*q1*	*q2*	*next*
12548	**2**	345-67-8901	Eliot, T.S.	94	97	-----
12597	**3**	456-78-9012	Carroll, Lewis	88	62	-----
12646	**4**	567-89-0123	Joyce, James	59	74	-----
12695	**5**	678-90-1234	Albee, Edward	79	87	-----
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

Taking the next record from the unordered list, the question becomes *Where does "Eliot, T.S." belong on the list?* Looking at the ordered list, we see that it should be between "Christie, Agatha" and "Tolkien, J.R.R.".



Simply have "Christie, Agatha" point to "Eliot, T.S." and have "Eliot, T.S." point to "Tolkien, J.R.R". The respective lists would now appear as:

Ordered List Figure 10.4.



first
12450

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12499

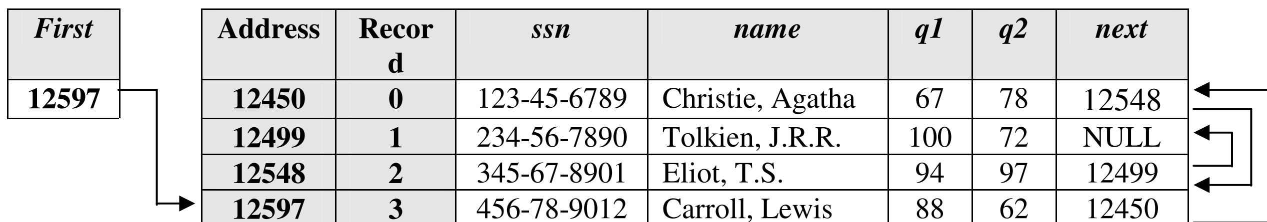
Unordered List Table 10.9.

Address	Record	*Ssn*	*name*	*q1*	*q2*	*next*
12597	**3**	456-78-9012	Carroll, Lewis	88	62	-----
12646	**4**	567-89-0123	Joyce, James	59	74	-----
12695	**5**	678-90-1234	Albee, Edward	79	87	-----
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

Continuing on, we now need to add the fourth record from the unordered list as the fourth record on the ordered list. A quick inspection shows that the name "Carroll, Lewis" should go before "Christie, Agatha", making it the 1st name on the ordered list. Before we add it, however, notice that we need to perform one other action:

We MUST place the base address of the new 1st record (i.e., "Carroll, Lewis") in the pointer variable *first*.

Ordered List Figure 10.5.



Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12499
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450

Unordered List Table 10.10.

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12646	**4**	567-89-0123	Joyce, James	59	74	-----
12695	**5**	678-90-1234	Albee, Edward	79	87	-----
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

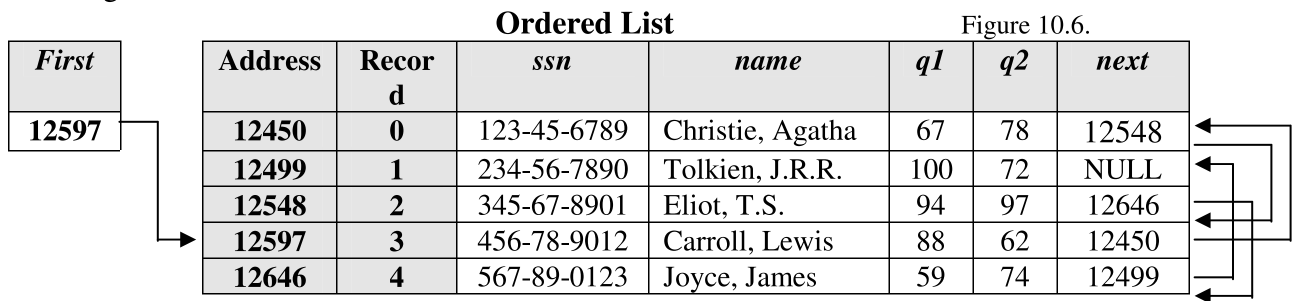
This last addition to the ordered list illustrates the last of three possibilities when adding records:

1. A record can be placed at the beginning of the list, in which case we must have the variable pointer first point to it, and have its next pointer point to the old first record
2. A record can be inserted between two existing records, in which case we have the record which goes before it point to the new record, and have the new record point to the record which the one preceding it previously pointed to.
3. A record can go at the end of the list, in which case we have previously last record point to the new record, and have the new record point to no one (NULL).

Continuing on with the ordered list insertions:

Adding the 5th record:

Ordered List Figure 10.6.



First
12597

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12646
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499

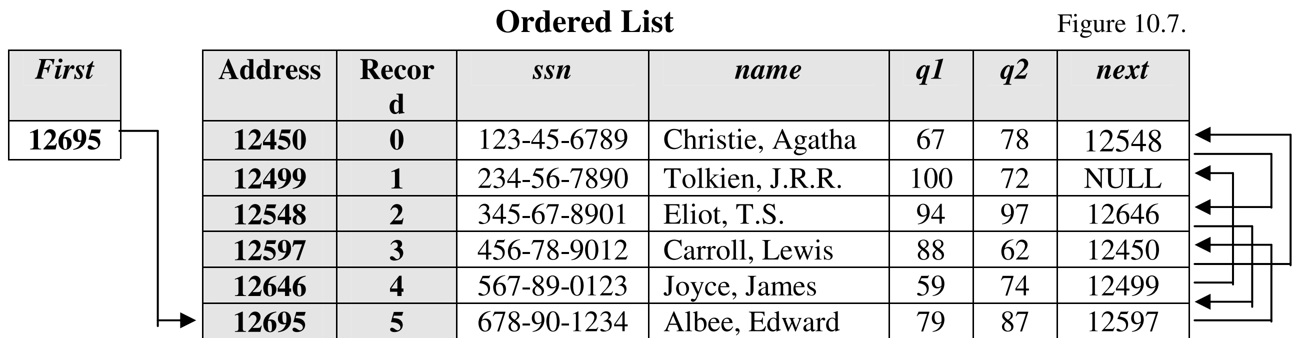
Unordered List Table 10.11.

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12695	**5**	678-90-1234	Albee, Edward	79	87	-----
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

Note the *next* address change for “Eliot, T.S.”

Adding the 6th record:

Ordered List Figure 10.7.



First
12695

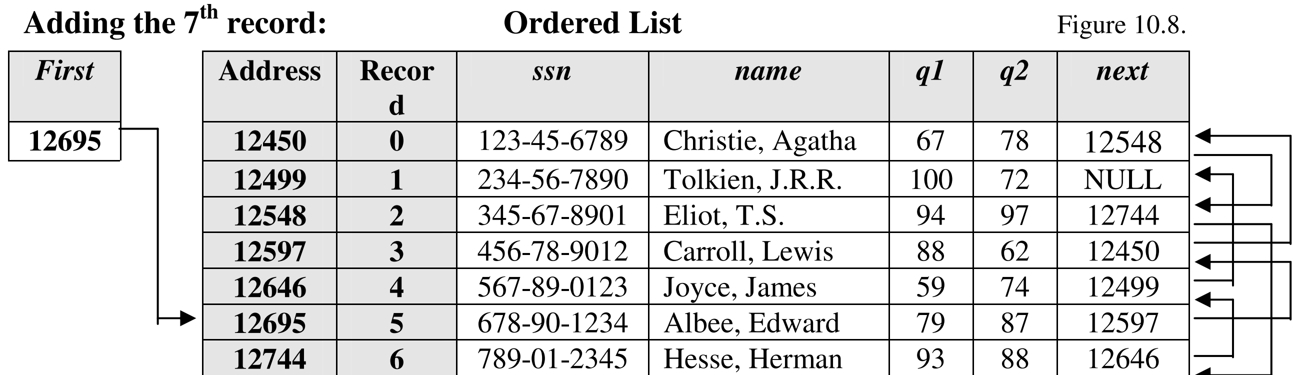
Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12646
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499
12695	**5**	678-90-1234	Albee, Edward	79	87	12597

Unordered List Table 10.12.

Address	Record	*Ssn*	*name*	*q1*	*q2*	*next*
12744	**6**	789-01-2345	Hesse, Herman	93	88	-----
12793	**7**	890-12-3456	Beckett, Samuel	60	64	-----

Note the change in variable *first*.

Adding the 7th record: **Ordered List** Figure 10.8.



First
12695

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12744
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499
12695	**5**	678-90-1234	Albee, Edward	79	87	12597
12744	**6**	789-01-2345	Hesse, Herman	93	88	12646

Unordered List Table 10.14.

Address	Record	*Ssn*	*name*	*q1*	*q2*	*Next*
12793	7	890-12-3456	Beckett, Samuel	60	64	-----

And adding the final record, we end up with the same ordered list that we saw in Figure 10.1.

Adding the final record: **Ordered List** Figure 10.9.

first
12695

Address	Record	*Ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12744
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499
12695	**5**	678-90-1234	Albee, Edward	79	87	12793
12744	**6**	789-01-2345	Hesse, Herman	93	88	12646
12793	**7**	890-12-3456	Beckett, Samuel	60	64	12597

What about the C Code necessary to link the list???

See C Code 10.4.

Let's follow the code line-by-line (after initialization of the array, which we already discussed):
The line after the array initialization: **struct writers** **first* = *novelists*, **present*, **previous*;

Declares three pointer variables:

1. *first*, which we already know will point to the 1st record on the ordered list. As we stated in our algorithm description, since we initially add the 1st record from the unordered list onto the ordered list, we initialize the variable first with the base address of our array (*first* = *novelists*).
2. *present*, which will hold the address of the record on the ordered list against which we are comparing the name from the unordered list.
3. *previous*, which will hold the address of the previous record on the ordered list (i.e., the record which points to the record against which we are comparing the name from the unordered list).

C Code 10.4.

```
#include <string.h>                                // for strcmp
struct writers                                     // Our structure template
{    char   ssn[10], name[31];
     int    q1, q2;
    struct writers * next;   };

 int main()
{ struct writers novelists[8] = { {"123456789","Christie, Agatha", 67,78,NULL},
        {"234567890","Tolkien, J.R.R.",100,72,},{"345678901","Eliot, T.S.",94,97,},
        {"456789012","Carroll, Lewis",88,62,},{"567890123","Joyce, James",59,74,},
        {"678901234","Albee, Edward",79,87,},{"789012345","Hesse, Herman",93,88,},
        {"890123456","Beckett, Samuel",60,64,} };
  struct writers *first = novelists, *present, *previous;
  int recno;
  for (recno = 1; recno < 8; recno++)
  {   present = first;                             // start with 1st record on list
// Is the new record name larger AND are there additional records?
      while ((strcmp(novelists[recno].name, present->name)>0) && (present->next != NULL))
         {   previous = present;                   // old record now previous
             present = present->next;   }          // Then get the next record
// Why are we out of the loop? Was the new < old record?
       if (strcmp(novelists[recno].name, present->name) < 0)
       {   if (present == first)
                first = &novelists[recno];         // reset first pointer
           else
               previous->next = &novelists[recno]; // prev -> new
           novelists[recno].next = present;        // new -> old
       }
       else                                        // new  > old
      {   if (present->next == NULL)
               novelists[recno].next = NULL;       // new -> NULL
          else
               novelists[recno].next = present->next; // new-> old
          present->next = &novelists[recno];       // old -> new
      }
    }
  return 0;
 }
```

There is also one integer variable: **int** *recno*; which will hold the offset of the unordered array. In other words, it will indicate the record in the unordered array which we are trying to place into the ordered array.

Starting with our loop:

for (*recno* = 1; *recno* < 8; *recno*++)
{ *present* = *first*; // place the address of the first record in pointer *present*.

Note that we start with the 2nd record on the unordered list (recno = 1). As we saw in Figure 10.2. and Table 10.7., We already have 1 record ("Christie, Agatha") on the ordered list. The for loop will take each unlinked record add it to the list.

Our inner (while) loop will be used for inserting the new records in their correct position on the loop:

while ((strcmp(*novelists*[*recno*].*name*, *present*->*name*)>0) && (present->next != NULL))

We will compare the unlinked record with every record on the linked list until:

1. The *name* field of the unlinked record is greater than the *name* field of the record in the linked list we are comparing the record with, and
2. We are not at the end of the linked list.

If BOTH conditions are met:

{ *previous* = *present*; // Store the address of the record we were comparing
present = *present*->*next*; } // Then get the next record

If we have to insert a record between two records we now have the addresses of both the records.

If the name field in the unlinked list is greater than the name on the linked list that we are comparing it to: **if** (strcmp(*novelists*[*recno*].*name*, *present*->*name*) < 0)

That means that we must insert the new (unlinked) before the record on the linked list that we are comparing it to. There are two possibilities:

1. The record on the linked list was the first record, meaning we must make the new record the first record on the linked list:

   { **if** (*present* == *first*)
   first = &*novelists*[*recno*]; // reset *first* pointer

2. The (linked) record which previously pointed to the (linked) record must be reset so that it points to the new (unlinked) record:

else
previous->*next* = &*novelists*[*recno*]; // *previous* contains the new record's address

regardless of which of the two possibilities occurred, we must next set the new record's *next* pointer field to point to the record we were comparing it with:

novelists[*recno*].*next* = *present*; }

If the new (unlinked) record has a name field which is greater than the name field of the record on the linked, then there are again two possibilities:

1. The record on the linked list was the last record, which means that the new record must become the last record on the list:

   if (*present*->*next* == NULL)
   novelists[*recno*].*next* = NULL; // the new record points to NULL

2. The new record must point to the record which the record on the linked list was pointing to:

   else
   novelists[*recno*].*next* = *present*->*next*;

Again, regardless of possibility, the linked record must point to the new record:

present->*next* = &*novelists*[*recno*];

The procedure continues until all of the unlinked records have been placed on the linked list.

Maintaining a Linked List

Actually, it is considerably less. We do not have to physically move any of the records. Maintaining the list is also considerably easier. To add a record to into the linked list, we simply follow the procedure above. To remove a record from the list, we simply remove the pointers to it (making sure that whatever it points to can still be accessed). For example, suppose we wished to remove "Eliot, T.S." from our list. The list would appear as:

Figure 10.10.

first
12695

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12548
12499	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL
12548	**2**	345-67-8901	Eliot, T.S.	94	97	12744
12597	**3**	456-78-9012	Carroll, Lewis	88	62	12450
12646	**4**	567-89-0123	Joyce, James	59	74	12499
12695	**5**	678-90-1234	Albee, Edward	79	87	12793
12744	**6**	789-01-2345	Hesse, Herman	93	88	12646
12793	**7**	890-12-3456	Beckett, Samuel	60	64	12597

Notice that there are no links to "Eliot, T.S.". "Christie, Agatha" now points to "Hesse, Herman".

? ***But all of the information about "Eliot, T.S." is still there!!!***

Yes, but we can not access it through our linked list (we still can access it through our offset, however). If we had a new record which we wished to add onto the list, we could write over Eliot's information (although chances are that the list would no longer be sorted by *ssn*). This problem is something we will overcome when we discuss dynamic memory allocation.

There is another advantage to linked lists, namely, we can order the list on multiple keys (without sorting). For example, if we also wished to link the list on quiz scores, as well as alphabetically, we might set up our record structure as:

C/C++ Code 10.5.

```
struct writers
{    char     ssn[10], name[31];
     int      q1, q2;
     struct writers * next, * quiz1, * quiz2;
};
```

And the ordered list (in ascending order) would appear as:

Table 10.11.

Address	Record	*ssn*	*name*	*q1*	*q2*	*next*	*quiz1*	*quiz2*
12450	**0**	123-45-6789	Christie, Agatha	67	78	12564	12735	12735
12507	**1**	234-56-7890	Tolkien, J.R.R.	100	72	NULL	NULL	12678
12564	**2**	345-67-8901	Eliot, T.S.	94	97	12792	12507	NULL
12621	**3**	456-78-9012	Carroll, Lewis	88	62	12450	12792	12849
12678	**4**	567-89-0123	Joyce, James	59	74	12507	12849	12450
12735	**5**	678-90-1234	Albee, Edward	79	87	12849	12621	12792
12792	**6**	789-01-2345	Hesse, Herman	93	88	12678	12564	12564
12849	**7**	890-12-3456	Beckett, Samuel	60	64	12621	12450	12507

Notice that since we have added two additional pointer fields, each record now contains:

10 + 31 + 2 + 2 + 4 + 4 + 4 = 57 bytes.

Notice also that we would have to store the 1st record addresses in separate variables, as we did when we stored the address 12685 (the base address for "Albee, Edward") in variable *first*. Our declaration might appear as:

C/C++ Code 10.6.

```
int main()
{ struct writers novelists[8] = { {"123456789","Christie, Agatha", 67,78,NULL,NULL,NULL},
        {"234567890","Tolkien, J.R.R.",100,72,,,},{"345678901","Eliot, T.S.",94,97,,,},
        {"456789012","Carroll, Lewis",88,62,,,},{"567890123","Joyce, James",59,74,,,},
        {"678901234","Albee, Edward",79,87,,,},{"789012345","Hesse, Herman",93,88,,,},
        {"890123456","Beckett, Samuel",60,64,,,} },
* first; * firstquiz1, * firstquiz2;
```

Where, after the linkages were made, pointer *firstquiz1* would contain the address 12678 and pointer *firstquiz2* would contain the address 12621. Notice also that our initialization allows for the additional pointer fields (of which only the first record has been initialized).

How would we go about finding a record on the linked list???

Let's try and find the name "Joyce, James" on the list. Starting with the first record on the list ("Albee, Edward", whose address is store in our pointer variable first), we would check to see if the record we were pointing to contained "Joyce, James" in the *name* field. If it did, we found it. If it didn't, AND the name we were searching for was greater than the name found at that address, we would get the address contained in the *next* field and repeat the process. If, for example, we were looking for the "Dostoyevski, Feodor", and we were comparing it with "Eliot, T.S.", we would know that it wasn't on the list because we should have found it by then. If we haven't found a match, and the address in the *next* field is NULL, we also know that the name is not on the list.

The relevant section of c code necessary (allowing the user to enter in the name to search for) might appear as:

C/C++ Code 10.7.

```
struct writers * present  =  first;
char searchname[31];
gets(searchname);
while ((strcmp(searchname, present->name) < 0) && (present->next != NULL))
      present = present->next;
if (strcmp(searchname, present->name) == 0)
      printf("The name %s was found at address %p\n", searchname, present);
else
      printf("The name is not on the list\n");
```

Notice that one advantage of searching a linked list, as opposed to searching a list which is not ordered in any fashion, is that we do not have to go to the end of the list before finding out that a record is not on the list. Because the list is ordered, we know that we have gone 'too far' if we have passed the point where we should have found an element.

? ⇐ ***What are the disadvantages of linked lists???***

There are a few, especially given the manner in which we set up our linked list here (i.e., as a fixed number of contiguous bytes of RAM).

1. Additional RAM is required for the pointer fields and for the pointers to the beginning of the list.
2. The list can only be searched in a sequential fashion, unless we make modifications, or add other structures.

? ⇐ ***How can we improve the search??? Can we do a binary search???***

No, we can't do a binary search, not directly at any rate, although we can come close using binary trees, as we shall see in a later chapter. We can, however, speed up the process.

? ⇐ ***How???***

Doubly Linked Lists

A doubly linked list implies that the list is linked in both directions, or in ascending order as well as in descending order.

? ⇐ ***How is that an advantage???***

Suppose we were looking for the name "Albee, Edward" on our list. We would expect to find it near the beginning of the list. Suppose however, that we were looking for the name "Zola, Emile" (Which is not on the list, but could be). *Where would we expect to find the record?* The way we presently have the list set up, we would have to start at the beginning of the list and search until the end.

Consider the following structure template and declaration (Code 10.8):

C/C++ Code 10.8.

```
struct writers
{    char    ssn[10], name[31];
     int     q1, q2;
     struct writers * next, * previous;
};
int main()
{ struct writers novelists[8] = { • • • }, * first, * last;
```

The doubly linked list might appear as:

Figure 10.12.

next *previous*

Address	*ssn*	*name*	*q1*	*q2*	*next*	*previous*
12450	123-45-6789	Christie, Agatha	67	78	12556	12609
12503	234-56-7890	Tolkien, J.R.R.	100	72	NULL	12662
12556	345-67-8901	Eliot, T.S.	94	97	12768	12450
12609	456-78-9012	Carroll, Lewis	88	62	12450	12821
12662	567-89-0123	Joyce, James	59	74	12503	12768
12715	678-90-1234	Albee, Edward	79	87	12821	NULL
12768	789-01-2345	Hesse, Herman	93	88	12662	12556
12821	890-12-3456	Beckett, Samuel	60	64	12609	12715

Where each record requires 10 + 31 + 2 + 2 + 4 + 4 = 53 bytes of storage, and variable first will contain the address 12715 and variable last will contain the address 12503.

With our doubly linked list, we might search for records whose *name* field starts with a character less than ‘M’ starting at the beginning of the list. We might begin our search for records whose *name* field begins with the letter ‘M’ or greater by starting at the end of the list.

This may seem like a great deal of work for eight records, but imagine a list with 100,000 records. While we do require 4 additional bytes of storage per record, and initial list establishment requires some effort, it is considerably less effort than sorting, relatively easy to maintain, and considerably faster to find a record.

Still, if we do have a list of 100,000 records, and we are searching for “Maugham, Somerset”, for example, we might still require 50,000 comparisons.

? ***Can we refine the search???***

Leveled Lists

The idea behind leveled list follows the simple concept we followed for searching and sorting:

The shorter the list, the faster it is to sort and search it.

If we had a list of 1,000 elements on a linked list, we know that on average, it would take 500 comparisons to find an element (assuming the element was on the list). What if we broke up the entire list into 5 (ordered) sublists, each list containing 100 elements. If we knew which sublist an element was on, we would only have to search that sublist, meaning that, on average, we could find that element with only 50 comparisons (10% the number of comparisons needed for the full list).

That is the idea behind a leveled list. If we have a linked list of 1,000 elements, we might determine where the 100th element (on the ordered list) is, the 200th, the 300th, and so forth. We would then store the addresses of those elements (in order) in a different array, and begin our search by first going through the array until we had a fairly good idea of where we should begin our search on the linked list.

Let's assume that our (unsorted) array (*novelists*) of structured objects (**struct writers**) contained 1,000 records, and that we have already linked the list. The list ranges (alphabetically) from "Abelard, Peter" through "Zola, Emile". Let's also assume that we have created an additional array of structured data objects called *index*. The structure template (call it **struct writerindex**) and the declaration for the array might appear as:

C Code 10.9.

```
struct writerindex
{    char    name[31];
     struct writerindex * address;
 };
int main()
{ struct writerindex index[10];
```

Where each record in the array index will require 31 + 4 = 35 bytes of contiguous storage, or a total of 350 bytes of contiguous storage.

If we were to go through our linked list (in array *novelists*), we might find that the 100th name (alphabetically, not physically) was "Burns, Robert", the 200th was "Dickenson, E.", the 300th was "Fielding, Henry", and so forth, all the way to the last name on the list (the 1,000th), "Zola, Emile". Let's take each of these names, along with the addresses at which they can be found, and store them in our array *index*. Each name in the *index* array is the last name on our sublist of 100 names. In other words, there are 100 names after "Oates, Carol" and between (and including) "Rilke, Marie".

index Table 10.12.

Record	*name*	*address*
0	Burns Robert	40233
1	Dickenson, E.	55227
2	Fielding, Henry	33030
3	Hardy, Thomas	27101
4	Keats, John	61058
5	Millay, Edna	16958
6	Oates, Carol	17595
7	Rilke, Marie	34304
8	Twain, Mark	13381
9	Zola, Emile	42928

Suppose that we were now looking for "Shakespeare, William". We would start by comparing "Shakespeare, William" with every name on the *index* list until we either found it (in which case we can go directly to it), or came across a name in the index which was greater than the name "Shakespeare, William". In this case, we would when we encountered the name "Twain, Mark". We now know that if Shakespeare is on the list, he must be found (on the linked list) after "Rilke, Marie" (and before "Twain, Mark").

Now all we have to do is go to the address at which we will find "Rilke, Marie" (34304) and work our way through the linked list until we either find "Shakespeare, William", or run across a name which is greater (in which case we know he is not on the list).

What if we were looking for a name that came before "Burns, Robert", on the list, say, "Aristophanes"Can we refine the search???

The procedure is still basically the same. As soon as we came across a name greater than "Aristophanes", we would start with the record before it. However, since there is no record before "Burns, Robert", we would go to the first record on the linked list (whose address we have stored in our variable *first*).

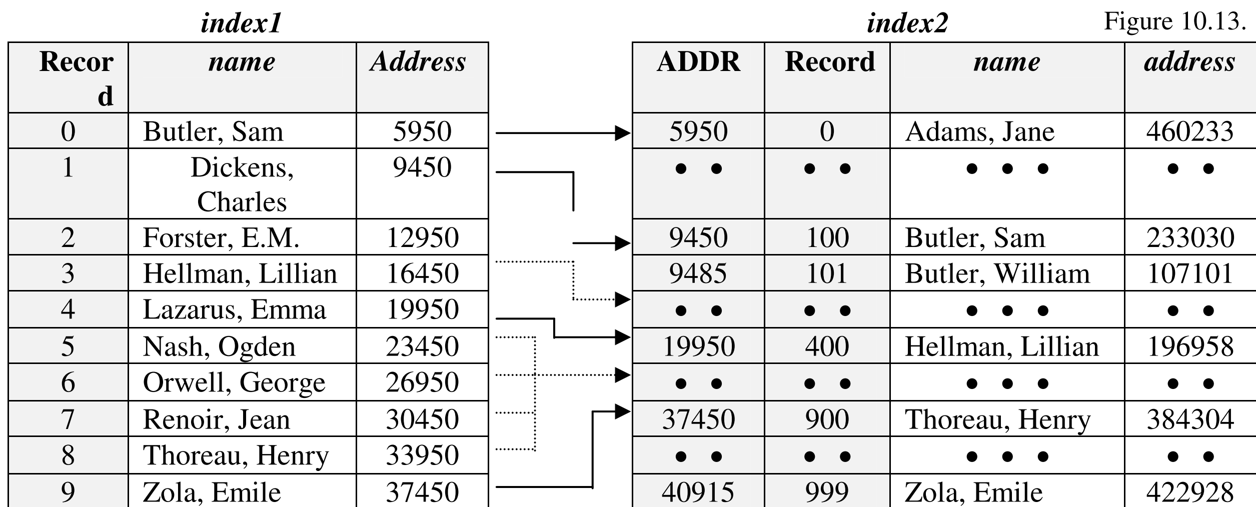
Using this 'leveled list' approach as we have laid it out, it would take, on average, 5 comparisons in *index*, and 50 comparisons in the linked list, for a total of 55 comparisons (assuming the name were on the list). This is a considerable savings over the 500 comparisons it would take if we didn't have our first list.

Since the array index is sorted, couldn't we perform a binary search on it???

Yes and no. A binary search assumes that the element we are looking for is on a list, or it doesn't exist at all. In our example, only 10 of the 1,000 names available are on the 1st list. We could modify our binary search to account for manner in which the list is set up, but the code then becomes more complex. Aside from that, remember that we wish to keep the list short, so that searching it is quick.

? ***But, if there were 100,000 names on the list, then the index array we used would contain 1,000 names, and we would have to make an average of 500 comparisons just on that list!!!***

Yes, it would take an average of 500 comparisons, plus additional 50 comparisons in the linked list, for 550 total comparisons. This is obviously unacceptable. We could adjust the interval between the names represented on the linked list, but that probably wouldn't help much. What we could do, however, is apply the same procedure and break-up our *index* array into sublists. We would now have two indices (call them *index1* and *index2*). To find a novelist, we would first start with *index1*, then go to *index2*, and then go to our linked list.



index1

Record	*name*	*Address*
0	Butler, Sam	5950
1	Dickens, Charles	9450
2	Forster, E.M.	12950
3	Hellman, Lillian	16450
4	Lazarus, Emma	19950
5	Nash, Ogden	23450
6	Orwell, George	26950
7	Renoir, Jean	30450
8	Thoreau, Henry	33950
9	Zola, Emile	37450

index2

ADDR	Record	*name*	*address*
5950	0	Adams, Jane	460233
• •	• •	• • •	• •
9450	100	Butler, Sam	233030
9485	101	Butler, William	107101
• •	• •	• • •	• •
19950	400	Hellman, Lillian	196958
• •	• •	• • •	• •
37450	900	Thoreau, Henry	384304
• •	• •	• • •	• •
40915	999	Zola, Emile	422928

Figure 10.13.

This approach is called a 2-leveled list.

This time, let's assume that we are trying to find the name "Kipling, Rudyard" on the linked list. We would start with *index1* and continue through the list until we encountered a name which was greater (in terms of the ASCII characters). When we do get to "Lazarus, Emma", we go to the address associated with it (19950). Starting with the name we find there ("Hellman, Lilian"), we continue until we again find a name greater than "Kipling, Rudyard" (let's assume that we encounter the name "Lamb, Charles"). We then go to the address (on the complete linked list) associated with "Lamb, Charles" in *index2*, where we might find the name "Kant, Emmanuel". We then continue comparing names until we

either find “Kipling, Rudyard”, or find a name alphabetically greater (in which case, “Kipling, Rudyard” is not on the list).

Suppose we were looking for “Orwell, George”. Since he is in index1, couldn’t we go to him directly?

Yes, but we would have to include an additional pointer field. Since the chances of an individual being on index1 are only 10/100,000 = 1 in 10,000, it doesn’t seem worth it. Similarly, the chances of an individual being in index2 are 1,000/100,000 or 1 in 100, it doesn’t seem worth the extra RAM required.

? ***How much of improvement is this 2-leveled list approach???***

We know that if we only had a linked list of 100,000 names, it would take an average of 50,000 comparisons. It we had the linked list and only *index2* (which contained 1,000 names), it would take an average of 550 comparisons. With the two indices, it would take an average of 5 + 50 + 50 = 105 comparisons.

We could probably improve the average number of comparisons necessary by increasing the number of names on index1 (for example, to 33) which would decrease the number of names in an the interval on index2 (to about 30), which means that the average number of comparisons needed would be about 16 + 15 + 50 = 81. The optimal number of names which we should include in both index1 and index2 can be mathematically determined, although we won’t discuss that here.

We could also have 3-leveled lists (or 4-leveled, 5-leveled, etc.) which might improve the number of comparisons (or might not; again we could mathematically determine the number of lists we should use for a given number of elements).

? ***Are there any disadvantages???***

Yes, there are always trade-offs involved. Adding leveled lists increases the amount of RAM required. The searching programs required (which we will not reproduce here) become more complex and time consuming. Further, both of these problems are compounded when we start increasing the number of leveled lists involved.

Summary

Linked lists offer a number of advantages over simple lists:

1. We can order elements without having to physically sort them
2. We can order on any field, or any number of fields
3. We can establish and maintain linked lists without extreme effort
4. We can develop search procedures on linked lists which can significantly reduce the number of comparisons needed to find an element.

As with all of the new data structures we have introduced, there are trade-offs involved. Linked lists do increase storage requirements because we must add pointer fields. Additional storage, and more complex programming, is required if we wish to refine our search procedures through the use of leveled lists. Nonetheless, the advantages of using linked lists far outweigh the disadvantages.

We will continue using the concepts underlying structured data objects and linking them through the use of pointers. Essentially, they will fill the foundations of all of the other data structures we will look at through the rest of this text. First, however, we must loosen two major constraints we have been working under. Namely, we must develop data structures which are NOT of a fixed size (such as an array) and do NOT require contiguous blocks of memory. We will dispose of these constraints in the following chapter on dynamic memory allocation.

Chapter Terminology: Be able to fully describe these terms

2-Level lists
3-Level Lists
Advantages of Linked Lists
Deleting Records from a List
Disadvantages of Linked Lists
Doubly Linked Lists
Inserting Records into a list
Leveled Lists
NULL Pointers
Pointer Fields
Pointers to the first record
Pointers to the last record
Searching a linked List
Singly Linked Lists
Sublists

Review Questions

1. List all the advantages and disadvantages of singly linked list. Of doubly linked lists.

2. I have five books which I have listed in a spreadsheet as follows:

Title	Author	Year Written	Value
Anna Karenina	Tolstoy	1876	125
Great Gadsby, The	Fitgerald, F.S.	1932	50
Guliver's Travels	Swift	1726	250
Rhetoric	Aristotle	329 BC	1500
Vanity Fair	Thackeray	1847	750

   a. Set up the needed structure template (call it **struct booktemplate**) to link the list alphabetically by author (make up string lengths as you see fit).

   b. Assign the variable *book* to the structure template. Show the declarations necessary.

   c. Assume that the base address of the variable *book* is 9000. Set up a table which would show all of the linkages.

   d. Now link the list by year written and value (maintaining the linkages established above). Show the modified **struct booktemplate**. Assuming that the base address of *books* remains 9000, show (in tabular form) how the linkages would appear.

3. Given the following structured object and declarations:

```
struct studentinfo { char ssn[11], name[28], address[30];
                     int age, IQ;
                     float gpa, balance;
                     struct studentinfo *next, *previous; };

void main()
{  struct studentinfo student[5] = {{"123457689", "Gates, W.","99 Microsoft
          Ln",38,108,3.87,4500,NULL,NULL},{"234567890", "Ford, H.","17 Michigan
          Ave", 125,115,3.10,250.00,,}, {"345678901","Trump, D.","1 Trump
          Towers",55,110,1.86,500,,},{ "456789012", "Astor, J.J.", "20 Wall
Street",76,
          100,2.76, 1000,,},{ "567890123","Carnegie, A.","100 Steel St", 81, 79,
2.25,0,,};

   struct studentinfo  * first, * last;
```

Using a linked list, I have ordered the names alphabetically (in ascending order). Field *next* will hold the address of the next person (alphabetically) on the list. Field *previous* will hold the address of the previous person (alphabetically) on the list. Variable *first* will point to the first person (alphabetically) on the list, and variable *last* will point to the last person (alphabetically) on the list.

When I entered the command: printf (“%lu\n”, &*student*[2]);
I received the output: **7828**

For each of the following printf statements, show the output which would be obtained:

a. printf(“%lu\n”,*first*);
b. printf(“%lu\n”,*last*);
c. printf(“%s\n”, *first*->*address*);
d. printf(“%lu\n”,&*student*[1].*balance*);
e. printf(“%f5.2\n”,*student*[3].*next*->*balance*);
f. printf(“%lu\n”, ++*first*);
g. printf(“%d\n”,*last*->*previous*->*age*);

4. Discuss all of the advantages and disadvantages of using leveled lists.

Review Question Answers (NOTE: checking the answers before you have tried to answer the questions doesn’t help you at all)

1. List all the advantages and disadvantages of singly linked list. Of doubly linked lists.

Singly Linked List Advantages	Singly Linked List Diadvantages
Allow for display in order w/o sorting	Additional RAM due to pointer field
Can order on any field	Additional programming needed
Easy to establish and maintain	
Need not search until list end if an Element is not on the list	

Doubly Linked List Advantages	Doubly Linked List Diadvantages
In addition to the above:	Additional RAM due to extra pointer field
Can search & display from either end	Additional programming needed

2a. The structure template would appear as:

struct booktemplate { **char** *title*[30], *author*[25];
int *yrwritten, value;*
struct booktemplate * *next*; };

Note: *yrwritten* could be stored as a character string. I have chosen to store it as a (signed) integer (where B.C. dates will be stored as negative values). The field *value* could also be stored as a real (float).

The manner in which I have set up the template requires:

30 + 25 + 2 + 2 + 4 = 63 bytes of storage per record, or 5 * 63 = 315 bytes of contiguous RAM allocation

2.b. The declaration assigning it to variable *book* would be:

void main()
{ **struct booktemplate** *book*[5] ={{"Anna Karenina","Tolstoy",1876,125,NULL},
Great Gadsby, The", Fitzgerald, F.S.",1932,50,}, {"Guliver's Travels", "Swift",
726,250,},{"Rhetoric","Aristotle",-329,1500,},{"Vanity Fair","Thackeray",
1847,750,}}, * *first*;

Note: Pointer variable first (which holds the address to the first name on the list) should be declared.

2.c. The lay-out would appear as:

first
9189

Rec.	Addr.	*title*	*author*	*yrwritten*	*value*	*next*
0	9000	Anna Karenina	Tolstoy	1876	125	NULL
1	9063	Great Gadsby, The	Fitgerald, F.S.	1932	50	9126
2	9126	Guliver's Travels	Swift	1726	250	9252
→ 3	9189	Rhetoric	Aristotle	-329	1500	9063
4	9252	Vanity Fair	Thackeray	1847	750	9000

2.d. The new template and declarations would be:

struct booktemplate { **char** *title*[30], *author*[25];
int *yrwritten, value;*
struct booktemplate * *nextname*, * *nextyrwritten*, *
nextvalue};
void main()
{ **struct booktemplate** *book*[5] ={{}}, * *firstname*, * *firstyrwritten*, * *firstvalue*;

and each record would require 8 more bytes of storage, or 71 bytes per record. The layout might appear as:

Rec.	Addr.	*title*	*author*	*yrwritten*	*value*	*nextn*	*nexty*	*nextv*
0	9000	Anna Karenina	Tolstoy	1876	125	NULL	9071	9142
1	9071	Great Gadsby, The	Fitgerald, F.S.	1932	50	9126	NULL	9000
2	9142	Guliver's Travels	Swift	1726	250	9252	9284	9284
3	9213	Rhetoric	Aristotle	-329	1500	9063	9142	NULL
4	9284	Vanity Fair	Thackeray	1847	750	9000	9000	9213

And the contents of each of the first pointers would be:

firstname	*firstyrwritten*	*firstvalue*
9213	9213	9071

3. Given the record structure and that &*student*[2] is 7828:

Each record requires: 11 + 28 + 30 + 2 + 2 + 4 + 4 + 4 + 4 = 89 bytes of storage.
The base address of the array would be: 7828 – 2 * 89 = 7828 – 178 = 7650

The linked list would thus appear as:

Addr	*ssn*	*name*	*address*	*age*	*IQ*	*gpa*	*balance*	*next*	*prev*
7650	123456789	Gates, W.	99 Microsoft Ln	38	108	3.87	4500.00	7828	7739
7739	234567890	Ford, H.	17 Michigan Ave	125	115	3.10	250.00	7650	8006
7828	345678901	Trump, D,	1 Trump Tower	55	110	1.86	500.00	NULL	7650
7917	456789012	Astor, J.J.	20 Wall Street	76	100	2.76	1000.00	8006	NULL
8006	567890123	Carnegie, A.	100 Steel St.	81	79	2.25	0.00	7739	7917

And the contents of the variables first and last are:

first	*last*
7917	7828

Therefore:

	Yields:
a. printf("%lu\n",*first*);	**7917**
b. printf("%lu\n",*last*);	**7828**
c. printf("%s\n", *first->address*);	**20 Wall St.**
d. printf("%lu\n",&*student*[1].*balance*);	7739 + 11 + 28 + 30 + 2 + 2 + 4 = **7816**
e. printf("%f5.2\n",*student*[3].*next->balance*);	8006 -> **0.00** (Carnegie's Balance)
f. printf("%lu\n", ++*first*);	Prefix notation: 7917 + 89 = **8006**
g. printf("%d\n",*last->previous->age*);	7828 -> 7650 -> **38** (Gate's age)

4. Discuss all of the advantages and disadvantages of using leveled lists.

Advantages:

- Search times decrease significantly
- As the number of leveled lists is increased, search times decrease accordingly (up to a point).

Disadvantages:

- The amount of RAM required increases
- The searching programs required are more complex
- Both problems are compounded when additional levels are added