Application Developers Training Company is: *Learn from the experts.* We did! The lectures and demonstrations took us from database design fundamentals and simple table, query, form, and report design and creation to advanced topics in coding *Access* macros, *Access* Basic (a programming language which is included with Microsoft *Access*), and event-driven programming. The Advanced Microsoft *Access* Programming class included an introduction to security issues, Data Access Objects, OLE/DDE, and *Access* Add-ins (used to extend the capabilities of the *Access* development environment). Although the details of these most advanced topics are beyond the scope of our second Microcomputer Applications course, I expect to incorporate these concepts in the separate, and more general Database Management course which I will also be teaching in the spring semester.

Throughout the week I was very impressed with the power and sophistication that Microsoft *Access* places at our fingertips. I anticipate that my Microcomputer Applications II students will produce usable applications far more powerful than my COBOL students could hope to create in a single semester. Of course, since many of my COBOL students will also be taking Microcomputer Applications II, our CIS graduates should be well equipped to handle projects in a traditional MIS setting as well as utilize today's powerful office application software.

For more information about Application Developers Training Company, you may call (800) 295-1883.

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

DOUBLY-LINKED OPPORTUNITIES

by Jim Carraway
Manatee Community College
Bradenton, FL 34207
carrawj@firnvx.firn.edu

Does the expression, Opportunity always knocks at the least opportune moment, mean anything to you? Well, it seems to control my life. Recently, while teaching a C++ class, the topic of sorting doubly-linked lists came up. I thought this was a subject worth pursuing. so I told my class to consider the problem and that we would cover the matter during the next meeting. Now, it was time for me to do my homework. My usual approach to a situation like this is to review all the textbooks that fill my book shelves and cover my floor. No success. This left me with an opportunity (remember the expression) at a time when I had at least a hundred other things on my *To Do* list. The result of my efforts are presented below.

At the next class meeting I presented the students with the discussion topic: *Which sorting algorithm is best suited for doubly-linked lists?* Due to the chained nature of linked lists it was the consensus of the class that a sort of the sequential variety would be the most appropriate. Of course the grand-daddy of sorts was mentioned -- the Bubble Sort. At this time I unveiled my masterpiece -- the Sediment Sort (Listing 1).

Additionally, the Swap function (Listing 2) was covered. The description of the process using the diagrams of a doubly-linked list (Figures 1- 3) and an example (Table 1) was one of those occasions when a picture was worth a thousand words. (Besides that it gave me an excuse to use the overhead projector that was gathering dust in the corner.) The students gave the appearance of being impressed.

At my next opportunity I plan to convert this algorithm for use with an array so I can do a comparative study with the Bubble Sort and the Shell Sort against 2000, 4000, and 8000 items. Then, I think I'll compute the Big O for the Sediment Sort. Does the expression, *If a study not worth doing at all, it is not worth doing well* mean anything to you?
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Listing 1

```
// The purpose of this function is to sort a doubly linked list. As each item in the original list settles to its
// layer in the sorted list it becomes the new bottom. Just as sedimentation is completed when everything
// has settled, the sort is completed when all the items have settled.

void sediment_sort(void)
    {
        struct list_type *sort_ptr, *new_tail;
        int swapped, end;

// new_tail controls the end of the list for sorting purposes.
        new_tail = tail;
        while (new_tail != head) {
// At this time, if new_tail = head either the list is empty (both are NULL) or there is
// only one item in the list (they both point to the same item). Later, if they are equal it
// indicates all the items in the list have settled to their proper level.
// Each pass starts at the head of the list.
                sort_ptr = head;
// If swapped is still zero at the end of a pass the sort is complete.
                swapped = 0;
// end is used to control the inside loop; it ends the loop when the next item to be
// compared is also one of the terminating items (tail or new_tail).
                end = 0;

                do {
// Compare an item to the next one in the list, if it is greater then the two items will swap
// positions in the list, otherwise use the next item as the comparison item.
                        if (stricmp(sort_ptr->name, sort_ptr->next->name) > 0) {
                                sort_ptr = swap(sort_ptr, sort_ptr->next);
                                swapped = 1;
                        }
                        else
                            sort_ptr = sort_ptr->next;

// If the new item to be compared is either the last item in the list (tail) or the item that
// controls the end of the list for sorting purposes (new_tail) establish a new end of the
// list for sorting purposes (new_tail) and set the end indicator - what this means is that
// there is no need to continue comparing items on this pass.
                        if ((sort_ptr == tail) || (sort_ptr == new_tail)) {
// Under these conditions if an item was swapped on this pass we need to establish a
// new end of the list for sorting purposes (new_tail) so the outside loop can be
// controlled - what this means is that if a swap has taken place the sediment has
// created a new bottom to the list and if a swap did not take place the list is completely
// sorted at this time.
                                if (swapped)
                                    new_tail = sort_ptr->prev;
                                else
                                    new_tail = head;
// Set the end indicator to terminate the inside loop.
                                end = 1;
                        }
                } while (!end);
        }
    }
```

Listing 1 - This is the listing of the Sediment Sort. Due to their nature linked lists do not offer themselves to a variety of sorts. The Sediment Sort is one of the fastest and most efficient sorts for linked lists.

Listing 2

```
// This structure contains only a name and two pointers.  It is a minimal structure used
// to illustrate the Sediment Sort.
struct list_type
    {
        char name[20];
// The following point to the previous entry in the list and the next entry in the list, respectively.
        list_type *prev;
        list_type *next;
    };

// head points to the front of the list and is initialized to NULL.
list_type *head = NULL;

// tail points to the end of the list and is initialized to NULL.
list_type *tail = NULL;

// This function swaps the order of the two items sent to it.
struct list_type* swap(struct list_type *one, struct list_type *two)
    {
// 1.
      if (one->prev == NULL)
            head = two;
         else
            one->prev->next = two;
// 2.
      if (two->next == NULL)
            tail = one;
         else
             two->next->prev = one;
// 3.
      two->prev = one->prev;
// 4.
      one->next = two->next;
// 5.
      one->prev = two;
// 6.
      two->next = one;

         return one;
    }
```

Listing 2 - The Swap function is one of the most efficient for this purpose because of the use of resources (no additional variables) and the limited number of compares and the minimum of swapping. The numbered statements correspond to the swapping in Figure 2.

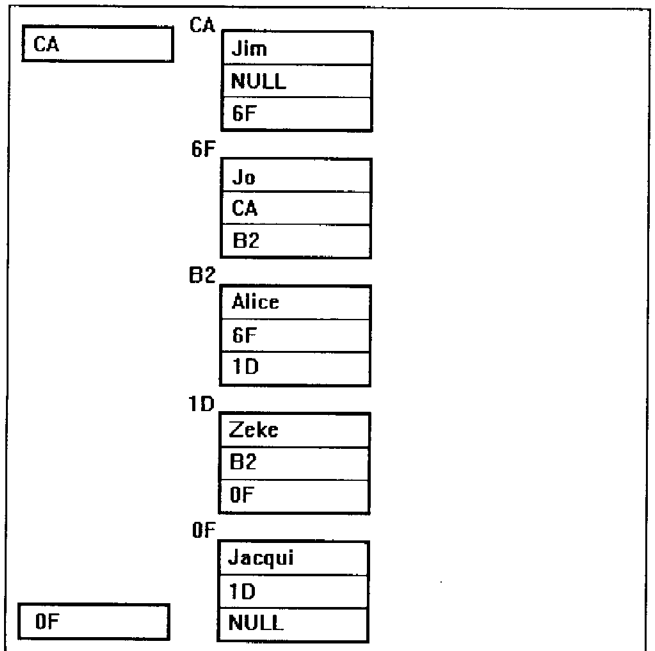

Figure 1 - This is the order of the list before the first swap in the first iteration of the sort.

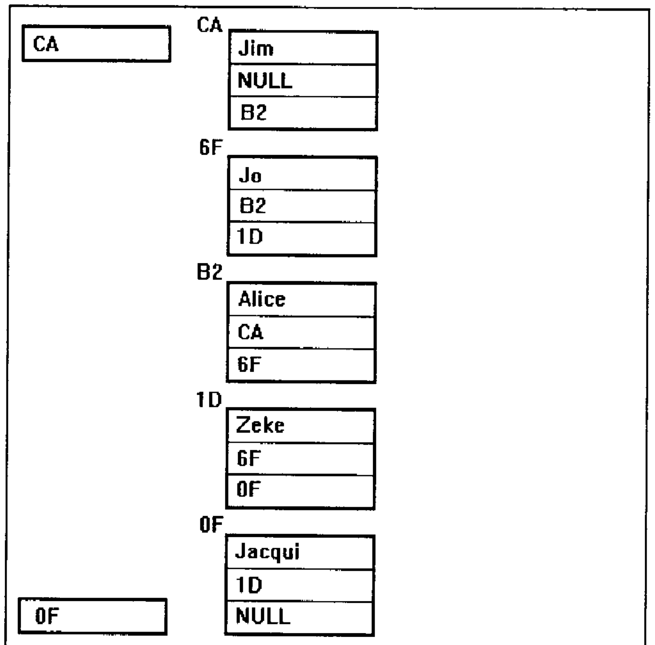

Figure 3 - This is the order of the list after the first swap in the first iteration of the sort.

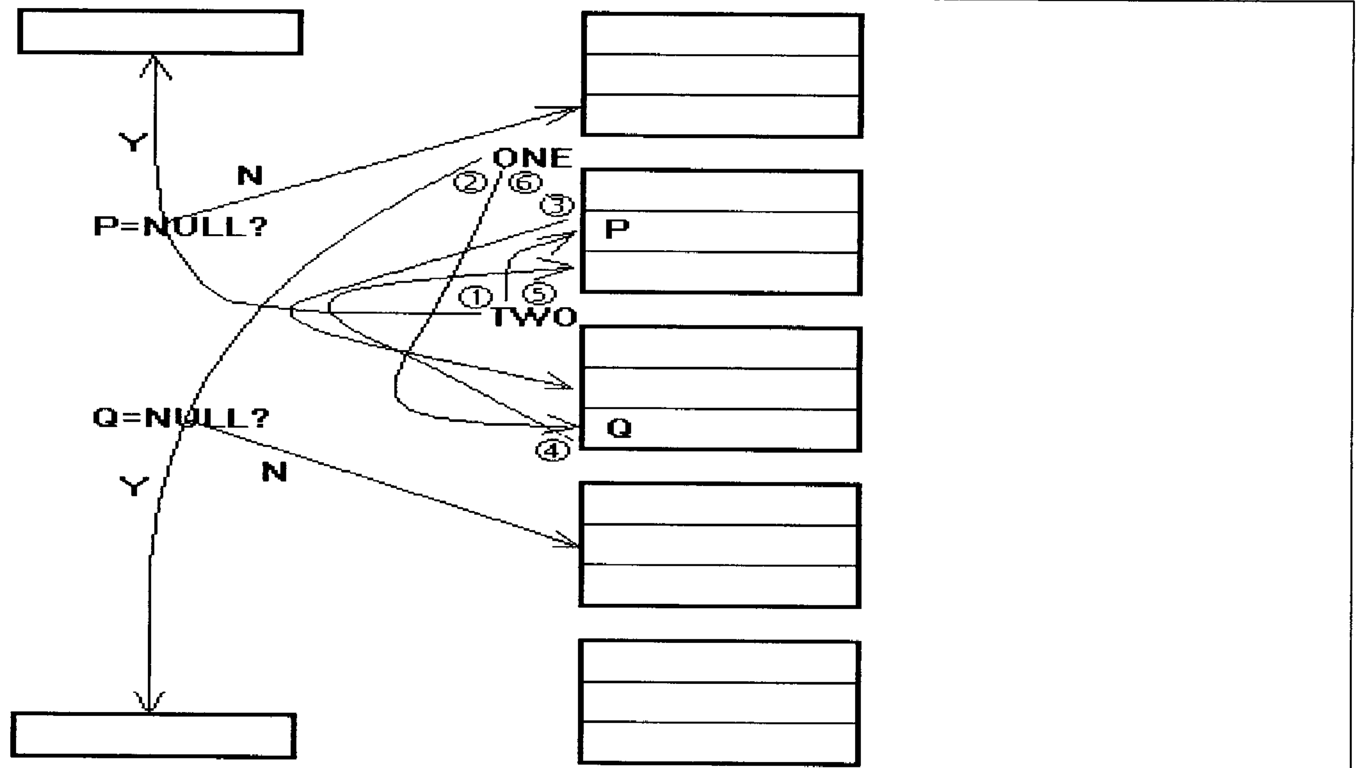

Figure 2 - This is a depiction of the swapping of addresses in the doubly linked list. The numbers correspond to the numbered commands in Listing 2.

Table 1

The configuration of the table below is: the first grouping is the settings of the variables and pointers as the inside loop is entered; the next grouping is the order of the items in the list at the completion of the inside loop (italicized items are removed from consideration because **new_tail** sets the end of the list; and the last grouping is the settings of the variables and pointers at the completion of the inside loop. The columns represent the executions of the inside loop.

1.	2.	3.	4.
head = CA	head = CA	head = B2	head = B2
tail = 0F	tail = 1D	tail = 1D	tail = 1D
new_tail = 0F	new_tail = 0F	new_tail = 0F	new_tail = 0F
sort_ptr = CA	sort_ptr = CA	sort_ptr = B2	sort_ptr = B2
swapped = 0	swapped = 0	swapped = 0	swapped = 0
end = 0	end = 0	end = 0	end = 0
Jim	Alice	Alice	Alice
Alice	Jim	Jacqui	Jacqui
Jo	Jacqui	Jim	*Jim*
Jacqui	Jo	*Jo*	*Jo*
Zeke	*Zeke*	*Zeke*	*Zeke*
head = CA	head = B2	head = B2	head = B2
tail = 1D	tail = 1D	tail = 1D	tail = 1D
new_tail = 0F	new_tail = 0F	new_tail = 0F	new_tail = B2
sort_ptr = 1D	sort_ptr = 6F	sort_ptr = CA	sort_ptr = 0F
swapped = 1	swapped = 1	swapped = 1	swapped = 0
end = 1	end = 1	end = 1	end = 1

1. The initial execution of the inside loop begins with the following order of the items in the list.
Jim Jo Alice Zeke Jacqui

The entire doubly linked list is shown in Figure 1 along with the **head** and **tail** variables. The address of each item is represented by the two characters by the struct (i.e., CA).

During this execution there are four comparisons made and there are two swaps. The first swap is shown in Figure 2. The result of the swap is show in Figure 3.

2. During this execution there are three comparisons made and there are two swaps. Zeke is not an item of comparison because the **new_tail** is set to 0F effectively making Jacqui the last item in the list.

3. During this execution there are two comparisons made and there is only one swap. Zeke and Jo are not items of comparison because the **new_tail** is set to 0F effectively making Jacqui the last item in the list.

4. During this execution there is only one comparison and no swaps. After the first comparison the **sort_ptr** and the **new_tail** were equal. Because no swap took place **new_tail** was set to **head**. When the loop control mechanism for the outside loop was tested it failed and the sort was complete.