Linked list code

We're going to take a little diversion into linked data structures before we move on to recursion, the first major topic of the 106B material. Linked lists are introduced in section 9.5 of the text and their recursive structure is explored in Chapter 12. The presentation in the book doesn't exactly match how we will cover it, but t may be worth skimming for background info. We will spend a day or two playing around with them as more practice with pointers and dynamic structures and then we will return to our real work. We will re-visit linked lists on several occasions in the B material.

I'll be presenting a lot of pre-written code using the overhead camera, and I thought it might help for you to have your own copy of the linked list code to follow along with. The explanatory text and comments are fairly sparse, so you'll want to keep awake in lecture.

First, here's our typedef. Note the use of the structure tag to embed a pointer to a structure of the same type within the structure:

```
typedef struct _address {
   string name, address, phone;
   struct _address *next;
} Address;
```

Here are the basic operations dealing with one Address cell, creating, freeing, and printing:

```
static Address *GetNewAddress(void)
{
   string name;
   Address *newOne;

   printf("Enter name (or return to quit): ");
   name = GetLine();
   if (StringEqual(name, "")) return NULL;
   newOne = New(Address *);
   newOne->name = name;
   printf("Enter address: ");
   newOne->address = GetLine();
   printf("Enter phone: ");
   newOne->phone = GetLine();
   newOne->next = NULL;       // initialize field to show no one follows
   return newOne;
}

static void FreeAddress(Address *person)
{
   FreeBlock(person->name);
   FreeBlock(person->address);
   FreeBlock(person->phone);
   FreeBlock(person);
}

static void PrintAddress(Address *person)
{
   printf("%s\n", person->name);
   printf("%s\n", person->address);
   printf("%s\n", person->phone);
}
```

Let's start with the first (unsorted) version of the linked address list. In creating the list, we prepend each additional entry onto the head of the list, since that's the easiest approach.

```
static Address *BuildAddressBook(void)
{
   Address *list, *newOne;

   list = NULL;
   while (TRUE) {
      newOne = GetNewAddress();
      if (newOne == NULL) break;
      newOne->next = list;    // attach rest of list to new cell
      list = newOne;          // new cell becomes the head of list
   }
   return list;
}
```

Note that in freeing the list we have to be careful to not access the current cell after we have freed it.

```
static void FreeAddressBook(Address *list)
{
   Address *next;
   while (list != NULL) {
      next = list->next;      // save next ptr before we free this cell
      FreeAddress(list);
      list = next;
   }
}
```

We can use the idiomatic `for` loop linked-list traversal to print each address cell:

```
static void PrintAddressBook(Address *list)
{
   Address *cur;

   for (cur = list; cur != NULL; cur = cur->next)
      PrintAddress(cur);
}
```

A linear search is used to look up an entry by name:

```
static Address *FindPerson(Address *list, string name)
{
   Address *cur;

   for (cur = list; cur != NULL; cur = cur->next)
      if (StringEqual(name, cur->name))
         return cur;
   return NULL;
}
```

Now, we re-consider our decision to keep the list unordered and decide instead to maintain the list in alphabetical order. With this change, we can re-write our FindPerson function to be a bit smarter.

We use StringCompare to determine if we have exactly matched on the current cell. We can also note if our name has been passed alphabetically, indicating it isn't in the list, allowing us to bail early:

```
static Address *FindPersonInSortedList(Address *list, string name)
{
   Address *cur;
   int cmp;

   for (cur = list; cur != NULL; cur = cur->next) {
      cmp = StringCompare(name, cur->name);
      if (cmp == 0) return cur;     // exact match right here
      if (cmp < 0) return NULL;     // passed position & didn't find it
   }
   return NULL;   // finished off list and didn't find it
}
```

Now, here comes the tricky part, we have to construct the functions that will build the list up in sorted order. The simple add-in-front approach won't work for this, we need to write a routine to splice the cell into the middle of the list. The loop we wrote for FindPerson can find the appropriate position for us, but note that it will go one past where we need to stop. After the loop, cur points to the cell that will follow the newOne in the list. However, given the forward-chaining properties of linked lists, we have no easy way to get to the cell previous to this one. Thus we will maintain two pointers when walking down the list, this additional pointer "prev" tracks the previous cell. After the loop, "prev" will point to the cell before the newOne, "cur" will point to the cell after. We need to splice newOne right in between the two cells. This means attaching "cur" to follow the new cell and attaching the new cell to follow "prev". It is possible that the previous cell is NULL (when newCell is inserted at the head of the list), and thus we must handle this case specially. Since this will require changing list, we need to pass the pointer by reference, necessitating the Address **!

```
static void InsertSorted(Address **list, Address *newOne)
{
   Address *cur, *prev;
   int cmp;

   prev = NULL;               // first cell has no previous cell
   for (cur = *list; cur != NULL; cur = cur->next) {
      cmp = StringCompare(newOne->name, cur->name);
      if (cmp == 0) return;   // already there, ignore new one
      if (cmp < 0) break;     // passed position and didn't find it
      prev = cur;
   }
   // now, "prev" points to one before newEntry, "next" is right after
   newOne->next = cur;
   if (prev != NULL)
      prev->next = newOne;
   else
      *list = newOne;   // note the special case!
}
```

Now's a good time to think through the special cases and make sure the code handles them correctly. What happens if the cell is being inserted at the very end of the list? What about at the very beginning? What if the list is entirely empty before we start?

Here's how we would call the insertion function to build a sorted address book:

```
static Address *BuildSortedBook(void)
{
   Address *list, *newOne;

   list = NULL;
   while (TRUE) {
      newOne = GetNewAddress();
      if (newOne == NULL) break;
      InsertSorted(&list, newOne);   // pass list by reference
   }
   return list;
}
```

Deleting is also a bit treacherous. We need to find the cell to delete and then carefully splice it out of the list and free its memory. Again, we're going to travel two pointers down the list to find the cell to delete and the cell previous to it. If we don't find the cell at all, we bail. Once we have the cell and its previous neighbor, we can wire up what the list past the deleted cell to follow the previous cell. Again, we have the special case of deleting the first cell in the list.

```
static void DeleteAddress(Address **list, string name)
{
   Address *cur, *prev;
   int cmp;

   prev = NULL;
   for (cur = *list; cur != NULL; cur = cur->next) {
      cmp = StringCompare(name, cur->name);
      if (cmp == 0) break;          // found it
      if (cmp < 0) return;          // passed position & didn't find it
      prev = cur;
   }
   if (cur == NULL) return;         // we never found it
   // now, cur points to the entry to delete, prev is one before
   if (prev != NULL)
      prev->next = cur->next;
   else
      *list = cur->next;
   FreeAddress(cur);    // free all memory from this cell
}
```

You might note that find, inserting, deleting all start with a very similar loop, and a good instinct is to want to unify them into a helper function. This function could be given a list and a name and would find the position in the list that cell would be at. It could return by reference both the prev and the current (you can easily get to the next from the prev though) which you could use to finish off insert, delete, find, etc.

One piece of extra complication for both insert and delete is handling the special case of inserting or deleting at the head of the list. In such a case, there is no previous cell that needs to be re-linked to what follows and the head needs to be re-set. One programming technique that can be used to avoid this need for a special case is to add an extra cell, called a "dummy cell" at the head of the list, so that the list, even when empty, always has that cell and that all "real" cells always have a previous cell and thus the need for the special case is removed. This does mean elsewhere we have to skip over that dummy cell (printing, finding, etc.) I'm not a big dummy cell fan, but the book does adopt this technique, so I thought I would at least mention it.

"Either that wallpaper goes or I go."
— last words of Oscar Wilde