Linked list code

We're going to take a little diversion into recursive data structures for practice with recursion and pointers. A linked list is a nifty data structure that using pointers to manage a flexible, dynamic collection. Pointers are discussed in section 2.2-2.3 and linked lists are introduced in section 9.5 We will spend a lecture playing around with them to explore the idea of dynamic structures and recursive data and how you can write recursive algorithms to process these lists.

I'll be presenting some code already written, 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 node structure:

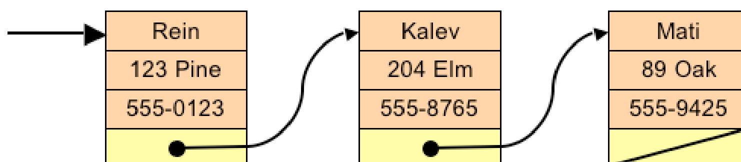
```
struct Entry {
   string name, address, phone;
   Entry *next;
};
```

Each list node contains a pointer to another node, making for a recursive definition. A linked list consists of a node, followed by another, shorter linked list. Each node is dynamically allocated in the heap, using the **new** operator. The organization of the list is determined by how the pointers are wired up between nodes. Inserting or removing a node is merely a matter of re-writing a few pointers, which makes for a very flexible data structure.

Here are the basic operations of creating and printing a node:

```
Entry *GetNewEntry()
{
   cout << "Enter name (ENTER to quit):";
   string name = GetLine();
   if (name == "") return NULL;
   Entry *newOne = new Entry; // allocate in heap
   newOne->name = name;
   cout << "Enter address: ";
   newOne->address = GetLine();
   cout << "Enter phone: ";
   newOne->phone = GetLine();
   newOne->next = NULL; // no one follows
   return newOne;
}

void PrintEntry(Entry *person)
{
   cout << person->name << endl
        << person->address << endl
        << person->phone << endl;
}
```



Let's start building the linked address list. At first, we won't keep the entries into any order. We prepend each additional entry onto the head of the list, since that's the easiest approach.

```
Entry *BuildAddressBook()
{
   Entry *listHead = NULL;

   while (true)  {
      Entry *newOne = GetNewEntry();
      if (newOne == NULL) break;
      newOne->next = listHead;      // attach rest of list to node
      listHead = newOne;            // node becomes head of list
   }
   return listHead;
}
```

Note that in deallocating the memory used for the list we have to be careful to not access the current node after we have freed it.

```
void DeallocateList(Entry *list)
{
   while (list != NULL) {
      Entry *next = list->next;     // save next ptr before deallocation
      delete list;
      list = next;
   }
}
```

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

```
void PrintList(Entry *list)
{
  for (Entry *cur = list; cur!= NULL; cur = cur->next)
      PrintEntry(cur);
}
```

Let's try exploiting the recursive structure here. One way to conceptualize a list is to think of it as a node, followed by another list. We can write algorithms to process such a list by using a recursive strategy:

```
void RecPrintList(Entry *list)
{
  if (list != NULL) {
     PrintEntry(list);
     RecPrintList(list->next);
  }
}
```

What if you wanted to print the list in reverse order? What changes would it take to the recursive version? What about the original iterative version above? Which is the easier one to modify? There are lots of algorithms that can be written recursively that exploit the recursive structure of the list (searching, inserting, reversing, and so on).

Now let's re-consider our decision about ordering and decide instead to maintain the list in sorted order. The simple add-in-front approach won't work for this, we need to write a routine to splice the node into the middle of the list at the correct position relative to its neighbors.

We can start with an iterative approach to find the appropriate position for the new node. The basic idea is to traverse from the beginning until we find the place for the new node. However note we have to go one past to find that position. After the loop finishes, **cur** points to the node 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 node previous to this one. Thus we will maintain two pointers when walking down the list, this additional pointer **prev** tracks the node right behind cur. After the loop stops, we need to splice **newOne** right in between the **prev** and **cur**. This means attaching **cur** to follow the new node and attaching the new node to follow **prev**. It is possible that the previous node is NULL (when the new node 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 — very tricky!

```
void InsertSorted(Entry * & list, Entry *newOne)
{
   Entry *cur, *prev = NULL;  // first node has no previous

   for (cur = list; cur != NULL; cur = cur->next) {
      if (newOne->name == cur->name) return;      // ignore dup
      if (newOne->name < cur->name) break;
      prev = cur;
   }
   // now, "prev" is one before newEntry, "next" is 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 node 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?

The above code works, but is pretty messy. Should we try it recursively and see how that works out? There are basically three cases to consider: we are at the end of the list (the base case for the recursion), the new cell needs to be added to the front of the list (because it precedes the first cell), or it belongs somewhere farther back in the list (because it belongs after the first cell). The first two cases are handled with the same code— splice the cell onto the front of the existing list. In the last case, we just recursively insert into the rest of the list. This is a dense piece of code — pointers, pass by reference, linked lists, and recursion all packed into 5 lines. I recommend tracing the operation of this function, in particular to get comfortable with how the recursive call eventually splices the cell into the correct place and how the list wiring is updated.

```
void RecInsertSorted(Entry * & list, Entry * newOne)
{
     if (list == NULL || newOne->name < list->name) {
        newOne->next = list;
        list = newOne;
    } else {
        RecInsertSorted(list->next, newOne);
    }
}
```

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

```
Entry *BuildSortedBook()
{
   Entry *listHead = NULL;

   while (true) {
      Entry *newOne = GetNewEntry();
      if (newOne == NULL) break;
      InsertSorted(listHead, newOne); // list by reference
   }
   return listHead;
}
```

Deleting an entry is also requires some careful thinking. We need to find the node to delete and then carefully splice it out of the list and free its memory. Here's a function that will find a particular name and remove it from the list. Again, we have the special case of deleting the first node in the list that requires passing the list by reference. Because recursion is working out so nicely, we take a recursive approach here.

```
void DeleteFromList(Entry * & list, string name)
{
    if (list != NULL) {
        if (list->name == name) {
            Entry *toDelete = list;
            list = list->next;
            delete toDelete;
      } else
             DeleteFromList(list->next, name);
    }
}
```

Is recursion an aid or a distraction when implementing the functions on a linked list? As an exercise for yourself, try writing the delete function using an iterative strategy instead of a recursion one. Which is easier to conceptualize? Easier to write? Easier to debug? 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 node would be at. It could return by reference both the previous and the current (you can easily get to the next from the previous 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 node 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 node, called a "dummy node" at the head of the list, so that the list, even when empty, always has that node and that all "real" nodes always have a previous node and thus the need for the special case is removed. This does mean elsewhere we have to skip over that dummy node (printing, finding, etc.) I'm not a big dummy node fan, but the book does adopt this technique in some places, so I thought I would at least mention it.

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

Source: http://see.stanford.edu/materials/icspacs106b/H21-LinkedListCode.pdf