# Linked list code

I've decided to take a little diversion into recursive data structures before we move on to abstract data types. Linked lists are introduced in section 9.5 of the text and their recursive structure is further explored in Chapter 12. We will spend a day or two playing around with them to get further practice with recursion and pointers and then get back to our real work.

I'll be presenting a lot of code for the lists, and I thought it might help for you to have your own copy 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 _cell {
   int data;
   struct _cell *next;
} Cell;
```

Here are the basic operations dealing with creating and printing a cell:

```
static Cell *GetNewCell(void)
{
   int num;
   Cell *newOne;

   printf("Enter num (or -1 to quit): ");
   num = GetInteger();
   if (num == -1) return NULL;
   newOne = New(Cell *);
   newOne->data = num;
   newOne->next = NULL;       // initialize field to show no one follows
   return newOne;
}

static void PrintCell(Cell *c)
{
   printf("%d\n", c->data);
}
```

Let's start with building up a linked list from the numbers entered by the user. In creating the list, we prepend each additional entry onto the head of the list, since that's the easiest approach.

```
static Cell *BuildList(void)
{
   Cell *newOne, *head = NULL;

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

```
}
```

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

```
static void PrintList(Cell *head)
{
   Cell *cur;

   for (cur = head; cur != NULL; cur = cur->next)
      PrintCell(cur);
}
```

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

```
static void PrintList(Cell *head)
{
   if (head != NULL) {
      PrintCell(head);
      PrintList(head->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?

How about freeing the list? Each individual cell needs to be freed, and we need to be careful to free the following node before freeing the current one, recursion again provides an elegant strategy:

```
static void FreeList(Cell *head)
{
   if (head != NULL) {
      FreeList(head->next);
      FreeBlock(head);
   }
}
```

How about a recursive traversal to sum all the numbers?

```
static int SumList(Cell *head)
{
   if (head == NULL)
      return 0;
   else
      return head->data + SumList(head->next);
}
```

There are lots of algorithms that can be written recursively that exploit the recursive structure of the list.

Let's re-consider our decision to keep the list unordered and decide instead to maintain the list in sorted order. Let's just right to 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 at the correct position.

Again, let's keep working at this recursively to practice our recursion skills. At a given position of the list, we determine whether the new element precedes it, and if so we splice it into the list in front, otherwise we recursively insert at the end. If we get to the end and all elements are less than the new value, we place it at the end.

Here is our first attempt at the code (and this code has a serious bug, so watch it)!

```
static void InsertSorted(Cell *head, Cell *newOne)
{
    if (head == NULL) {
        head = newOne;
    } else if (newOne->data < head->data) {
        newOne->next = head;
        head = newOne;
    } else {
        InsertSorted(head->next, newOne);
    }
}

static Cell *BuildSortedList(void)
{
   Cell *newOne, *head = NULL;

   while (TRUE) {
      newOne = GetNewCell();
      if (newOne == NULL) break;
      InsertSorted(head, newOne);
   }
   return head;
}
```

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? If we try some cases out, we're going to discover the serious bug I was alluding to earlier: when we are trying to insert at the beginning of the list, we need to truly change the head pointer to point to a new location and this means we need to pass the pointer by reference, necessitating the Cell **!

```
static void InsertSorted(Cell **head, Cell *newOne)
{
    if (*head == NULL) {
        *head = newOne;
    } else if (newOne->data < (*head)->data) {
        newOne->next = *head;
        *head = newOne;
    } else {
        InsertSorted(&(*head)->next, newOne);
    }
}
```

```
static Cell *BuildSortedList(void)
{
   Cell *newOne, *head = NULL;

   while (TRUE) {
      newOne = GetNewCell();
      if (newOne == NULL) break;
      InsertSorted(&head, newOne);
   }
   return head;
}
```

Wow! Trace that one carefully until it all the pointers are clear!

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 we have the special case of deleting the first cell in the list. That requires passing head by reference:

```
static void Delete(Cell **head, int num)
{
    Cell *toDelete;

    if (*head != NULL) {
        if ((*head)->data == num) {
            toDelete = *head;
            *head = (*head)->next;
             FreeBlock(toDelete);
             Delete(head, num);
       } else
           Delete(&(*head)->next, num);
    }
}
```

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