Computer Science 2530
April 1, 2020

Happy Wednesday, April 1.

Destructive functions on linked lists

So far we have only used and written functions that do not change existing lists. Now we look at functions that make changes to existing linked lists.

We cannot stay within the conceptual notation that we have used because it provides no functions that modify a list. So we must use C++ notation and operations directly. We will use the following notation.

NULL	empty list
L->head	head of list L
L->tail	tail of list L
new ListCell(h, t)	a list whose head is h and whose tail is t.

Call by reference

Recall that, when we pass a variable by reference, we are passing the variable itself, not the current value of the variable. (You indicate call by reference by writing & after the type of the parameter.)

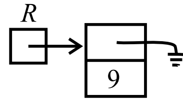
The variable that is passed can belong to a structure. For example, Suppose increment is defined as follows.

```
void increment(int& x)
{
  x = x + 1;
}
```

Then

```
ListCell* R = new ListCell(9, NULL);
increment(R->head);
```

first creates R as follows.



The call to increment passes the head *variable* of *R to increment. That variable has its value increased by 1; after increment returns, R->head = 10.

Example: add 1 to each value in a list

Let's write a function incAll(L) that adds 1 to each number in linked list L. For example, if L is list [2, 4, 6], then, after incAll(L), L will be [3, 5, 7]. We have already seen a boilerplate loop for looping over a linked list. Let's use it.

```
void incAll(ListCell* L)
{
  for(ListCell* p = L; p != NULL; p = p->tail)
  {
    (p->head)++;
  }
}
```

You can also write incAll using recursion.

```
void incAll(ListCell* L)
{
  if(L != NULL)
  {
    (L->head)++;
    incAll(L->tail);
}
```

Notice that it was not necessary to pass list L by reference, since incAll(L) does not change L. It changes L->head. The next function needs a list L passed by reference because it will (sometimes) change L.

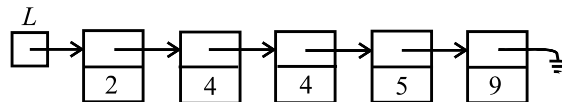
Read lecture notes page **34D** and do the exercise at the end of that page.

Insert a value into a sorted list

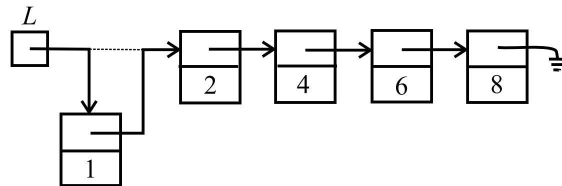
Page **34A** shows how to insert a value into a sorted linked list. Read that page. Here is the function that is derived (with 'cons' replaced by 'new ListCell').

```
void insert(int x, ListCell*& L)
{
  if(L == NULL || x <= L->head)
  {
    L = new ListCell(x, L);
  }
  else
  {
    insert(x, L->tail);
  }
}
```

Notice that, in the first case, the value of L is changed. If L is:



then a call to insert(1, L) should end with L as follows.



It is essential that L is passed by reference to insert.

Another example

Lecture notes page **34C** derives a definition of remove(x, L), which removes the first occurrence of x from L (if there is one). Read that page.

Exercises

Do the exercises at the bottom of lecture notes page **34A**. The last time I asked for students to submit a function definition, hardly anybody did it. So I won't bother to ask again. But if you want clarification about whether your definition works, feel free to send it to me and ask.

Memory sharing

We can define a nondestructive function cat(A, B) that builds a list that contains all of the members of list A followed by all of the members list B. For example, using conceptual notation, cat([2,4,6], [3,5,7]) should return [2,4,6,3,5,7]. Since cat is nondestructive, we can derive equations for it.

$$\begin{aligned} \text{cat}([\,], B) &= B \\ \text{cat}(A, B) &= \text{head}(A)\colon \text{cat}(\text{tail}(A), B) \end{aligned}$$

converting that to C++ gives the following definition of cat.

```
ListCell* cat(ListCell* A, ListCell* B)
{
  if(isEmpty(A))
  {
    return B;
  }
  else
  {
    return cons(head(A), cat(tail(A), B));
  }
}
```

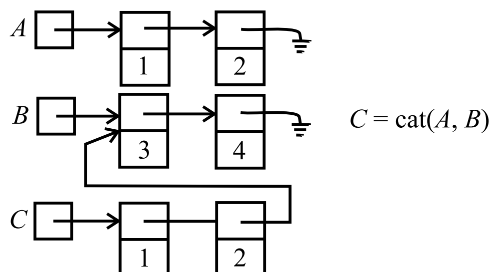
Written in direct C++, it looks like this.

```
ListCell* cat(ListCell* A, ListCell* B)
{
  if(A == NULL)
  {
    return B;
  }
  else
  {
    return new ListCell(A->head, cat(A->tail, B));
  }
}
```

This definition of cat brings up an important issue. Suppose that we do the following. Remember that List is identical to ListCell*.

```
List A = cons(1, cons(2, NULL));   // A = [1, 2]
List B = cons(3, cons(4, NULL));   // B = [3, 4]
List C = cat(A, B);                // C = [1, 2, 3, 4]
```

The result, shown as linked list diagrams, is a follows.



The list cells that contain 3 and 4 are part of two different lists, B and C. That is called *memory sharing.*

If you use memory sharing, it is important to be aware of it. Usually, you avoid changing lists when you use memory sharing. Why? Suppose that you do

```
incAll(C);
```

Now you have changed two lists, B and C, not just list C.

If you want to be able to use destructive functions, it is a good idea to avoid memory sharing. You can make cat avoid memory sharing by making a copy of list B.

```
ListCell* cat(ListCell* A, ListCell* B)
{
  if(A == NULL)
  {
    return copyList(B);
  }
  else
  {
    return new ListCell(A->head, cat(A->tail, B));
  }
}
```

Exercise

Write a definition of copyList(L), which returns a copy of L, using new ListCells. Use recursion. That makes it easy.