Computer Science 2530
March 27, 2020

Happy Friday, March 27.

Remember that there is an exam on Monday. Make sure that you understand the answers to the practice exam (available on the course web page), and be prepared to do similar problems on the exam.

Looping over a linked list using a while-loop

Today we look at how to write a function of a list that uses a loop instead of recursion.

We have seen how to plan a while-loop by doing a pre-simulation. The same thing works with lists. Here are two examples.

Adding up the numbers in a list

You can use a scan algorithm to add up the numbers in a linked list. It is similar to adding up the numbers in an array, but is actually more direct. When working with an array, you use an array index or length as a proxy for the values in the array. With a linked list, there is no need for a proxy. The sequence is right there to look at.

Let's have two variables, r and s, where s accumulates the sum and r is the part of the list that has not yet been looked at. Here is a pre-simulation.

r	s
[2, 4, 6, 8, 10]	0
[4, 6, 8, 10]	2
[6, 8, 10]	6
[8, 10]	12
[10]	20
[ ]	30

Notice that, at each line, s is the sum of all of the numbers that have been removed from the original list [2, 4, 6, 8, 10].

The last line of the pre-simulation tells you when to end the loop. Clearly, when r is an empty list, you should stop. Updating variables r and s is simple. Check that the following always works.

```
  s = s + head(r);
  r = tail(r);
```

Here is the completed function definition.

```
  int sum(ListCell* L)
  {
    ListCell* r = L;
    int       s = 0;

    while(!isEmpty(p))
    {
      s += head(p);
      p  = tail(p);
    }
    return s;
  }
```

Getting the reversal of a list

Suppose reverse(L) is supposed to return the reversal of list L. For example, reverse([2, 4, 6, 8, 10]) should return [10, 8, 6, 4, 2]. Here is a pre-simulation, where variable **hand** is the part of list L that has not yet been looked at and **table** is the reversal of the part that has been removed from L.

hand	**table**
[2, 4, 6, 8, 10]	[ ]
[4, 6, 8, 10]	[2]
[6, 8, 10]	[4, 2]
[8, 10]	[6, 4, 2]
[10]	[8, 6, 4, 2]
[ ]	[10, 8, 6, 4, 2]

The last line tells you to stop the loop when **hand** is an empty list. The following works to update variables **hand** and **table**. Check that it works.

```
  table = cons(head(hand), table);
  hand  = tail(hand);
```

Remember that cons(h, t) is equivalent to $h : t$. It is the list whose head is h and whose tail is t. For example, cons(6, [4, 2]) = [6, 4, 2].

Here is a completed definition of reverse(L) using a while-loop.

```
ListCell* reverse(ListCell* L)
{
  ListCell* hand  = L;
  ListCell* table = emptyList;

  while(!isEmpty(hand))
  {
    table = cons(head(hand), table);
    hand  = tail(hand);
  }
  return table;
}
```

Looping over a linked list using a for-loop

You can also use a for-loop to loop over a linked list. Remember that we treat a for-loop differently from a while-loop. Think of the for-loop as doing something for each value in the list.

Here is a boilerplate loop that looks at each member x in list L.

```
for(ListCell* p = L; !isEmpty(p); p = tail(p))
{
   int x = head(p);
   ...
}
```

All you need to do is decide what to do for each value x in list L. Here is a definition of sum(L) using a for-loop. The idea is to add each member of the list into s.

```
int sum(ListCell* L)
{
  int s = 0;

  for(ListCell* p = L; !isEmpty(p); p = tail(p))
  {
    int x = head(p);
    s = s + x;
  }
  return s;
}
```

Here is a definition of reverse(L) using a for-loop. Each value x in list L is added to the front of the result list.

```
ListCell* reverse(ListCell* L)
{
  ListCell* result = emptyList;

  for(ListCell* p = L; !isEmpty(p); p = tail(p))
  {
    int x = head(p);
    result = cons(x, result);
  }
  return result;
}
```

Const lists

Neither sum(L) nor reverse(L) changes L. So it makes sense to say that the parameter is a **const** parameter. You can do that, but any variable that refers to a const list or a part of a const list must itself be marked const. Here is a definition of sum(L with a **const** parameter. Notice that p has type **const ListCell***.

```
int sum(const ListCell* L)
{
  int s = 0;

  for(const ListCell* p = L; !isEmpty(p); p = tail(p))
  {
    int x = head(p);
    s = s + x;
  }
  return s;
}
```

Here is a definition of reverse(L) with a **const** parameter. Notice that **result** is not const, since it does not refer to any part of list L.

```
ListCell* reverse(const ListCell* L)
{
  ListCell* result = emptyList;

  for(const ListCell* p = L; !isEmpty(p); p = tail(p))
  {
    int x = head(p);
    result = cons(x, result);
  }
  return result;
}
```

Using C++ notation

When writing in C++, you can feel free to use C++ notation directly. The following table shows equivalent things, one written in conceptual notation and the other in C++ notation.

Conceptual	**C++**
emptyList	NULL
isEmpty(L)	L == NULL
head(L)	L->head
tail(L)	L->tail
$h:t$	new ListCell(h, t) or cons(h, t)

Here is the definition of reverse(L) converted to C++ notation.

```
ListCell* reverse(const ListCell* L)
{
  ListCell* result = NULL;

  for(const ListCell* p = L; p != NULL; p = p->tail)
  {
    result = new ListCell(p->head, result);
  }
  return result;
}
```

You can use any mixture of conceptual and C++ notations that you find convenient.

Exercises

Read lectures **33A** to **33D**. Do the two exercises at the bottom of page **33A**.