Linked Lists and List Editing Methods

Tailored to Assignment #1

Diagrams taken from lecture notes

Language: Plain Old C

By: Anthony Nguyen

Table of Contents

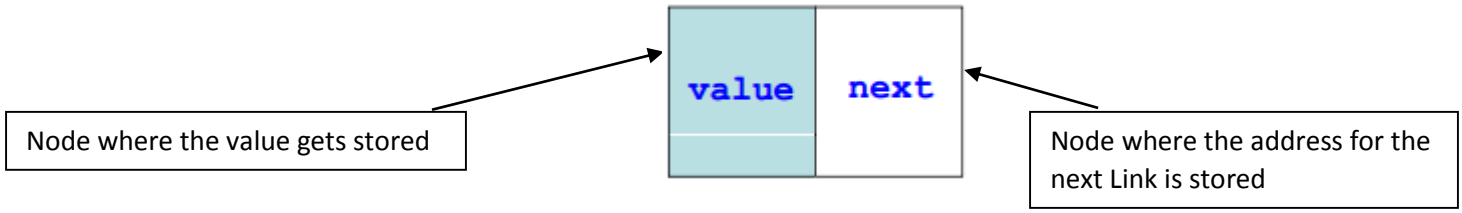
1. What is a Linked List?

A **linked list** is a method of storing data that is slightly different than arrays. Each item inside of a linked list is called a **Link**. Each link consists of two parts, or **nodes**:

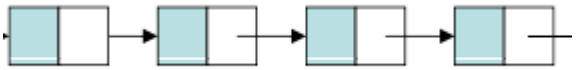
-Value, which is where the data value goes

-**Next**, which is where the address for the next Link in the linked list is stored

The 'Next' part of the link is what **links together the values in the list**, which is what makes the list a **Linked List**.



A single link inside a Linked List



An entire linked list

A linked list will contain a pointer that will point to the beginning of the linked list, called the **head**. The head does not have a value, it only contains the address that points to the first Link.

Real Life Comparison

Suppose have a five piece puzzle, but someone scattered the pieces of the puzzle around your house. At the location of each piece, there is the puzzle piece (obviously) and a note that tells you where the next piece of the puzzle is. The person also left a note on your front door that tells you where the first piece is, so you know where to start.

The note on the door is the **head pointer**, because it shows you where the first piece is.

The piece of the puzzle and the note directing you to the next piece together make up a **Link**.

The puzzle piece itself is the **Value** node**.**

The note that comes with the puzzle piece which directs you to the next puzzle piece is the **Next** node.

All five pieces, together, make up the **Linked List**.

Side Note: Arrow and Dot Operators

After declaring a data structure, which is shown in the next section, you can access members of a data structure using the arrow operator or the dot operator.

If you declare your variable as a regular variable, you use the **dot operator** (.) to access the member. If you declare your variable as a pointer, you use the **arrow operator** (->) to access the member.

2. Creating a linked list in C (with a sentinel)

Step 1: Creating the data structure types

Since a Link is not an included data structure in C, we need to declare it first.

We use **struct** to create the new data structure type itself. Add a curly brace to the end.

```
1. struct { //Creates a new data structure type
```

We then **declare the Value node**. Declaring the Value node is like declaring a variable, just use the data type you want to store in the node and then the name of the node (value).

```
2. int value ; //Declare the Value node, use a data type
```

After the Value node is declared, we need to declare the Next node where the address of the next Link will be stored. We use the syntax **Link *next**.

```
3. Link *next ; //Creates a node inside the Link for the address of the
   next Link, the Next node
```

Close off the newly created data type with a closing curly brace, and then add the name of the data type (in our case a **Link)**.

```
4. } Link ; //Call the new data type a Link
```

The entire thing looks like this:

```
1. typedef struct { //Creates a new data structure type
2. int value ; //Declare the Value node, use a data type
3. struct Link *next ; //Creates a node inside the Link for the address of
   the next Link, the Next node
4. } Link ; //Call the new data type a Link
```

Now, you have a structure in your program called a Link. It can be used like a regular data type. The structure itself contains two members: **value** and **next**.

After the Link data type is declared, we need to declare a data structure for the entire linked list itself. We can call the linked list data type anything we want but for the purposes of following lecture slides, we will use the name **LList3**. They call it that because they are implementing a variation 3 linked list.

Once again, we use **typedef struct** to create the new data structure type itself.

```
5. typedef struct { //Creates a new data structure type
```

We then need to create an integer inside the data structure to **store the size of the linked list**.

```
6. int size ; //Integer inside the data structure that stores linked list
   size
```

After we declare the size of our linked list, we need to **make a Link inside the data structure for the head pointer**. We will call this Link **head**. There is a star in front of **head** because it is a pointer. Note that because head is actually a link, **head** will also contain **value** and **next**, so which means that you can access value or next within the data structure by saying "linklist"->head->next

```
7. Link *head ; //Declare the head pointer link
```

Like before, we close off our data structure with a closing curly brace, and add the name.

```
8. } LList3;
```

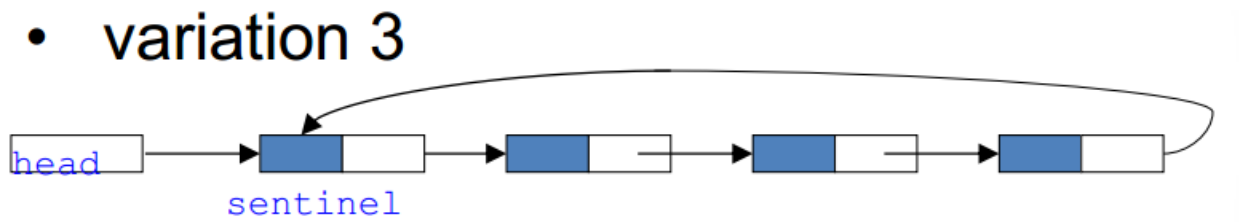
Now, we have:

```
1. typedef struct { //Creates a new data structure type
2. int value ; //Declare the Value node, use a data type
3. struct Link *next ; //Creates a node inside the Link for the address of
   the next Link, the Next node
4. } Link; //Call the new data type a Link
```

```
5. typedef struct { //Creates a new data structure type
6. int size ; //Integer inside the data structure that stores linked list
   size
7. Link *head ; //Declare the head pointer link
8. } LList3;
```

Step 2: Initializing the Linked List and Adding a Sentinel

Now that we have declared data types, we can create our linked list. The linked list can be initialized inside a function. The assignment calls the function **initList** so we will do the same. This list will contain a sentinel because the assignment asks for one, but it usually isn't needed. A sentinel is a link that makes sure the linked list is never empty. It contains no values and also acts as the head pointer. What makes this unique is that the last link actually points back to the sentinel, creating a loop. This is a **variation 3 linked list** which looks like this:



In the main task, you would create your linked list by calling the function, like this (assuming we call the linked list **llist**):

```
1. LList3 *llist = initList() ;
```

Now we can begin writing the function. First, we declare the function by using the new data structure **LList3**, the name of the function with a * because we are doing this by reference, and **void** in parenthesis because we are not taking any input variables.

```
1.  LList3 *initList(void)
2.  {
```

We then need to create a temporary linked list within the function. We need to do this because the function will eventually return the entire temporary linked list as the initialized new linked list. We declare the linked list like we would declare a pointer, using LList3 as the data type. We will go ahead and allocate memory using malloc on the same line because it is easier that way. The sizeof should be set to LList3.

```
3.  LList3 *templist = (LList3 *) malloc(sizeof(LList3));
```

We then set the head and the size of the temporary linked list. The head of the linked list should be set to NULL for now, because there are no links in the list yet. Size should be set to 0.

```
4.  templist->head = NULL;
5.  templist->size = 0;
```

We then create a sentinel for the linked list using the Link data structure.

```
6.  Link *sentinel;
```

We need to allocate some memory in the heap for the sentinel. The amount of memory allocated needs to be the size of a Link.

```
7.  sentinel = (Link *) malloc(sizeof(Link));
```

The purpose of the sentinel is simply to be there. There are no values stored inside, so we can set the value node of the sentinel to be the current value of the sentinel.

```
8.  sentinel->value = sentinel;
```

We then want the make the head pointer the sentinel, so we set them equal to each other.

```
9.  templist->head = sentinel; //making the head pointer of the list the sentinel
```

Finally, we want to pass the temporary linked list back to linked list on the main task.

```
10. return(templist);
```

The linked list is initialized with a sentinel.

Step 3: Adding Links

Method A: Prepending

Creating the new link and calling the prepend function

Prepending is adding a new link to the beginning of the linked list.

First, create a link called newLink (or whatever you want to call it) in your int main. Use the previously declared data type, and make it equal to NULL because we want empty nodes.

```
11. Link *newLink = NULL;
```

We then need to allocate memory in the heap to store the link using malloc.

```
12. newLink = (Link *) malloc(sizeof(Link));
```

The new link needs to be assigned a value before sending it off to the prepend function. You can store anything in there provided that it is the same data type as in the Link structure declaration. We'll use the number 16. Also, because the link is not pointing anywhere yet, we can set the next pointer to NULL.

```
13. newLink->value = 16;
14. newLink->next = NULL;
```

We then call our prepend function like this, using the new link and the linked list as input variables:

```
1.  prepend(newLink, llist);
```

Writing the prepend function

We will call our function **prepend** (great name choice). It is declared like any other function, and because values are not being returned, it is a void function. The input variables are the new link and the already initialized linked list. We call these input variables anything we want.

```
1.  void prepend(Link *link, LList3 *llist)
2.  {
```

We then want the "Next" node of the new link to point to wherever the head pointer of the linked list is pointing. Because we want to keep the sentinel, we want to actually place the **link next to the head pointer**. We do this by equating the new link's next node to the header pointer's next node.

```
3.  link->next = llist->head->next; //giving the link the address stored in the head
    pointer
```

Finally, we update the head pointer to point to the newly prepended link. With the two statements, the newly prepended link will point to the one after that. We also increase the size of the linked list by updating the llist->size value.

```
4. llist->head->next = link; //giving the header's next node the address of the link
5. llist->size++;
        6. }
```

And we are done writing our function.

Method B: Appending (Without Recursion)

It is probably a good idea to learn appending without recursion first so that you get how it's done.

First, you create a new link with the method described in the prepending method (Method A). The only difference is that now, instead of calling the prepend function, you would call the append function, like this:

```
append(newLink, llist);
```

The function itself will be declared like this:

```
1. void append( Link *link, LList3 *llist ) ;
2. {
```

It takes the new link and the linked list as input variables. Because we have a sentinel link, we know for sure that the list is not empty, so it is not necessary to check for links on the linked list.

Start off by trying to find the last link. You do this by creating a tester link called **lastlink** (or whatever you want to call it, doesn't matter do me) to traverse the linked list.

```
    3. Link *lastlink = (Link *) malloc(sizeof(Link));
```

You then want the tester link to point to the link **next** to the head pointer, because we want to ignore the sentinel (for now).

```
    4. lastlink = llist->head->next;
```

Now, we want to transverse the link until we reach the end of the link. To do this, we keep on pointing the last link pointer to the next link in the list until we reach the final link, which will point back to the sentinel/head pointer.

This is done in a **while loop**. You want the program to keep looping the task until the lastlink->next points to the sentinel.

```
    5. while (lastlink->next != llist->head) //do the loop while the tester link does not
       point to the head pointer/sentinel
6. {
```

To traverse the list, we point the tester link to the address stored in the 'next' node. This will essentially move the tester to the next node, and the loop will repeat. We then close off the while loop.

```
7.  lastlink = lastlink->next ; //point lastlink to the next link
```

When the while loop finishes, the tester link will point to the last link in the linkedlist. We then want to set the 'next' node of the tester link to point to the inputted link we want to append, which is simply 'link'. Doing this will append the linked list.

```
8.  lastlink->next = link ; //makes the last link point to the link being appended
```

We then update the address of the next node of the last link to point back to the head pointer. Finally, we increase the size of the linked list to accommodate for this change and close off the function.

```
9.  link->next = llist->head; //makes the appended link point back to the sentinel
10. Llist->size++ ; //increase the size of the list
11. }
```

Our append function is written. This method is not recursive, so it wouldn't be correct for the assignment, but it is good to know how to traverse and append a linked list.

Method C: Appending with Recursion

Appending with recursion involves two parts: finding the last link in a recursive function, and then appending to the last link found. The two functions will be called **findLastLink** and **append** because that's what the assignment asks for.

The Append function

It is easier to write the append function first because you know exactly what you're passing off, which will make the find last link function easier to write second. Much of the append function is the same as the previous one in Method B, but the link finding is done in a separate function. So first, you declare the function accepting the same input variables as before (the link you want to append and the linked list).

```
1.  void append( Link *link, LList3 *llist ) ;
2.  {
```

Then, we want to declare and retrieve the last link. You start off by declaring the link using the data structure type (Link) and then the pointer name, which we will call appropriately lastlink. We then equate it to the function findLastLink. The function itself will take the entire linked list as one input, and then the linked list's sentinel as the other.

```
3.  Link *lastLink = findLastLink(llist, llist->head) ;
```

After the function is called and we retrieved the last link, we can then append the input link to the linked list. We use the same method as the other append function in Method B.

```
    4.  lastLink->next = link ; // points the last link in the list to the new link
5.          link->next = llist->head ;  //points the new link to the sentinel
6.          llist->size++ ; //increases the size of the link
7.  }
```

Now that we have the append function written, we can write the find last link function.

The findLastLink Function

The find last link function will find the last link (shocker…). As inputs, it takes the linked list and a link. **The function will see if the inputted link points to the head**. The function type is Link because it returns a link.

```
1.  Link *findLastLink(LList3 *llist, Link *link)
2.  {
```

Now that the function is declared, we want to start the comparing. We create an if statement asking if the inputted link points to the sentinel on the linked list.

```
3.          if ( link->next == llist->head )//sees if the inputted link points to the sentinel
4.          {
```

If it does, then the link is the last link and the link will be returned that way. Use 'return' to return the link.

```
5.                  return( link ) ;//returns the link as the last link
6.          }
```

If the inputted link does not point to the sentinel, then it is not the last link. If that is the case, we need to run the function **again within the function** (recursion). This time, we use the link **next to the previously inputted link** as the link input variable. We run it inside the **else** condition.

```
7.          else
8.          {
9.                  return( findLastLink(llist, link->next) ) ; //repeats using the next link
10.         }
11. }
```

Eventually, when the list is traversed, and we find the last link, the function will unwind return the value for the last link. It will then return to the append function to do the append operation.

3. Modifying the List

Removing from the Front

This function removes the front link (the one next to the sentinel). It then returns the removed link. It takes the linked list as the input.

```
1. Link *removeFromFront(LList3 *llist)
2. {
```

We now need to declare the removed link. We assign it a NULL value because there is nothing in it yet.

```
3.         Link *removedLink = NULL ;
```

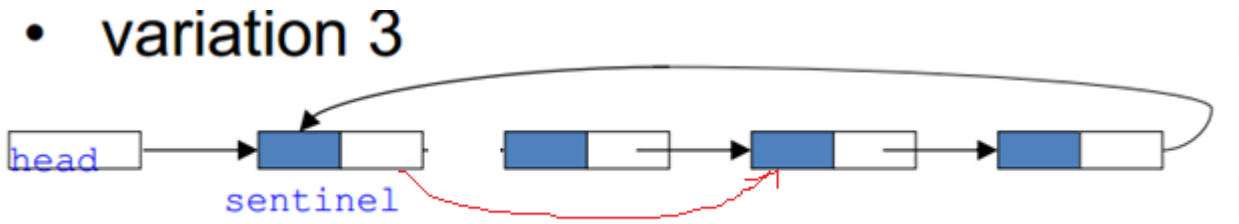
After this, we need to check if there is more than just the sentinel node in the linked list. A linked list with just a sentinel is pretty much an empty list when it comes to values, but we never want to remove the sentinel. **We do this check by checking if the sentinel's 'next' node doesn't point to itself in an if statement**.

```
4.         if ( llist->head->next != llist->head ) //making sure the sentinel doesn't point to
   itself
5.         {
```

Once we've made sure that the list isn't empty for values, we can then set the removed link equal to the link next to the sentinel. This is necessary to return the link that was removed.

```
6.                 removedLink = llist->head->next ;
```

After this, we then make the sentinel point to not the link after the sentinel, but the one after the link after the sentinel. A drawing to illustrate this would look like this:



Doing this pretty much removes the link next to the sentinel from the linked list.

```
7.                 llist->head->next = llist->head->next->next ; //making the sentinel skip the
   next link and point to the one after
```

To accommodate for the smaller linked list, we decrease the size variable by one.

```
8.                  llist->size-- ;
9.          }
```

At this point we can end the if statement and return the removed link to the main task.

```
10.         return(removedLink) ; //returns the removed link
11. }
```

Our function for removing the front is now written. It can be initialized in the task main by setting a newly declared empty link equal to the function.

Side Note or Challenge: Removing the last link (won't include the code)

You can remove the last link by traversing the linked list and **checking if the link next to the current link points to the sentinel**. Finding the actual last link itself is kind of useless (unless you have a two way linked list). Once you have the link before the last link, make the link before the last link point to the sentinel, and decrease the size of the linked list by 1.

4. Printing the List

You can print the list with a recursive function. This function will be called **printList**. **The function will print the links one at a time**. When you call the function, you want to pass the linked list and the link next to the sentinel. We pass the link next to the sentinel because we don't want to print the sentinel. Calling the function looks like this:

```
1.  printList( llist, llist->head->next ) ;
```

Now we write the function. We declare the function first taking the linked list and the link as input arguments. It is a void function because no values are being returned.

```
1.  void printList( LList3 *llist, Link *link )
2.  {
```

We then check to make sure that the link is not the sentinel. This is necessary because we will be going through the whole list after the recursion and we don't want the sentinel to be printed. This also acts as the recursion breaker.

```
3.          if ( link != llist->head )
4.          {
```

The internal function printf is then used to print the link's value. Similar to RobotC's nxtDisplayText or whatever (did I really just reference RobotC…), printf takes a few arguments. The first argument is in quotations, which basically tells printf what to say. %d tells printf that there will be an integer variable in place. After the variable, we put a -> to show a link. The variable is set as the link's value.

```
5.                  printf( "%d->", link->value ) ; //prints the link's value
```

To print more links, we call the printList function again, **but this time we use the next link as the input argument rather than the current link.**

```
6.                  printList(llist, link->next) ;
7.          }
```

After the function prints the entire list, it will eventually get to the sentinel, where the above if condition will not be satisfied. The function will then move to an else condition which is defined below.

```
8.          else
9.          {
```

We want to start a new line on the print screen, so we're going to tell printf to create a new line by putting \n in the quotations. We then close off the function.

```
10.                 printf("\n") ;
11.         }
12. }
```

The function for printing the list is now written.

5. What if there is no sentinel?

If there is no sentinel, there are a few things that change.

1. **Initialization:** We do not need to include the sentinel stuff in the initialization.
2. The header is not the sentinel. The header is just a pointer that will point to the first value but is not an actual link.
3. **Prepending:** When prepending, you want the new link to point to wherever the header is pointing, and after that is done you want the header to point to the new link.
4. **Finding the last link:** When looking for the last link, you wouldn't check to see if the last link points to the sentinel, but you would check to see if the last link points to null.
5. **Appending:** You would make your newly appended link point to null.
6. **Remove Front:** When removing the front, you would make the header point to the second link and then make the first link point to null.
7. **Remove Back:** When removing the back, when traversing you would look for the second last link by checking if the one after points to null. You would then make the second last link point to null.
8. **Printing:** You would pass off the head pointer as an input argument so that the machine can print whatever it is pointing to. Your if condition would make sure the link does not point to null. When it does point to null, in the else condition you would print the last link before starting a new line.