REVERSING A LINKED LIST IN PLACE

ANDREW FOOTE

The other day I finally realized how you reverse a linked list in place.

I had attempted to tackle this problem several times before, and had always ended up looking up the answer online, copying and pasting some code given in the answer and seeing that it worked, but not really understanding how it worked, as evidenced by my failing to remember how to do it the next time. Now, however, I'm confident that I can remember it for the long term.

It's remarkable how unhelpful the online resources were here. Consider this description of the algorithm from GeeksforGeeks.com, which was the first result Google gave me for "reversing a linked list" as I was writing this post:

(1) Initialize three pointers prev as NULL, curr as head and next as NULL.
(2) Iterate trough the linked list. In loop, do following.

```
// Before changing next of current,
// store next node
next = curr->next

// Now change next of current
// This is where actual reversing happens
curr->next = prev

// Move prev and curr one step forward
prev = curr
curr = next
```

This is a particularly bad explanation—it's not actually complete, and even the spelling and grammar is poor—but in terms of how it describes the solution it's not substantially different from the others I saw. It just says you have to use three pointers and do a particular sequence of reassignments until one of them is null. There is little explanation as to why this particular sequence of reassignments works. For the reader, it's just a magic sequence that has to be memorized.

In order to understand anything complex, you have to use abstractions, to break the complexity into manageable chunks, and analogies, to allow you to apply your existing knowledge about other things. For me, the key to understanding how to reverse a linked list was realizing that what you do to the list in the loop above can be understood as simply carrying out the two operations below in succession:

(1) Remove the first item from the list.
(2) Attach this item to the front of a new list.

The processes of attachment and removal can be readily visualised, so they allow us to draw on our spatial intuition to reason about data structures. It's obvious that if we keep removing items from the original list and attaching them to the

new list until there are no items left to remove from the original list, then the new linked list will end up having the same items the original list originally had, but in reverse order. The new list can then be assigned to the variable holding the original list.

It's really an incredibly straightforward algorithm. I think the reason it took me so long to figure it out was that I was disregarding all ideas for solutions that on a conceptual level involved moving items into a new list and then reassigning the variable. This was a case where spatial intuition was leading me astray: such solutions are *conceptually* not "in-place", so I assumed any such solution would have be *actually* not in-place; an amount of space proportional to the size of the list would need to be used in order to store the new list. Of course, I was forgetting that you can remove items from the original list as you attach them to the new list, so that no additional space needs to be used that wasn't already being used to store the original list.

Here's the code I ended up writing (in C):

```c
#include <stdlib.h>

struct node {
    int value;
    struct node *next;
};

void list_push(struct node **list, struct node *node) {
    node->next = *list;
    *list = node;
}

struct node *list_pop(struct node **list) {
    struct node *node = *list;
    *list = node->next;
    return node;
}

void list_reverse(struct node **list) {
    struct node *new_list = NULL;

    while (*list)
        list_push(&new_list, list_pop(list));

    *list = new_list;
}
```

Note that if you inline the `list_push` and `list_pop` functions in `list_reverse`, you get what is essentially the loop with three pointers that all the online resources were talking about:

```c
void list_reverse(struct list **list) {
    struct list *new_list = NULL;

    while (*list) {
```

```
        struct list *node = *list;
        *list = node->next;
        node->next = new_list;
        new_list = node;
    }

    *list = new_list;
}
```

Only the variable names are different: `*list`, `node` and `new_list` correspond to the variables `next`, `curr` and `prev` respectively in the GeeksForGeeks.com description.

There are of course different ways of conceptualizing what happens during the loop. The conceptualization described above, where we move the items into a new list by attaching to the front, is just the one that came most naturally to me. From the variable names the writer of that GeeksForGeeks.com article used, it's evident that their conceptualization was different—`prev` doesn't make a lot of sense as a name for a new container we're placing items into.

Now that I look at it again, I see that the GeeksforGeeks.com article includes an animation which illustrates the conceptualization they were probably using. You can think of the linked list as a bunch of items connected by arrows, where the arrows represent pointers. To reverse the list, you just flip the arrows!

Once you conceptualize the problem in this way, I think it's fairly straightforward to work out how to do it. You have to iterate over the list, keeping both the current node and the previous node in memory so that you can point the current node to the previous one. The previous one either will have been already set to point to one before it, or is the first node in the list, in which case it needs to be pointed to `NULL`. We'll also need make temporary copies of what the current node was originally pointing to on each iteration, so we can move on to the next node even after we change what the current node is pointing to. So with this conceptualization, we can easily see that we need three pointers, and that `prev`, `curr` and `next` are appropriate names for them.

```
void list_reverse(struct list **list) {
    struct list *prev = NULL;
    struct list *curr = *list;

    while (curr) {
        struct list *next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
}
```

This makes it more clear where the GeeksforGeeks.com article was coming from. To its writer, the sequence of assignments in the loop probably seemed like a simple atomic operation—"make the current item point to the previous one". But the writer failed to explicitly point out that they were making use of this conceptualization, and the clarity of their explanation for a beginner suffered because of it.

It's likely that they weren't particularly aware that they were relying on this particular conceptualization, or that there were alternatives. I think this is a general problem with trying to explain things to other people. People tend to stick with the first conceptualization of a problem that proves helpful for them, and think only in those terms. Even if they are aware of alternative conceptualizations, they may think of those conceptualizations as misunderstandings of the problem, when it's really just a case of different conceptualizations having their own advantages and disadvantages and suiting different thinking styles. Even if some conceptualizations are less helpful than others, it may be best to allow a beginner to work with the conceptualization they find most natural as a starting point.

Anyway, all this has got me curious—if there is anybody still reading who has ever tried to reverse a linked list, how do you prefer to conceptualize the linked list reversal problem? Moving items, or flipping pointers, or something else? Leave a comment—I'd be interested to hear your answer.