

MTH 4300, Lecture 20

Adding Elements;
Linked Lists and Recursion;
Reversing Linked Lists;
Templates

E Fink

April 23, 2025

1. Adding Elements

Let's take the original list, and add in an element, placing it in its natural place in alphabetical order. This is somewhat similar to before – basically, you just switch some `next` pointers around – but there is the additional challenge of finding the location.

L20x1_add.cpp

It will again be helpful to have two pointers, `current` and `prev`, because you only know when you've found the correct position for your new element when you have gone one entry past it.

Additionally, extra care is required if the element should be added at the front or the end of the list.

Now that we've pointed out the benefits of linked lists with regard to insertion and deletion, we should mention a drawback: **accessing element `#i` in the list is slow**. You need a loop with a counter!

2. Linked Lists and Recursion

Linked Lists tend to be processed quite well using recursive functions. If we think of a Linked List as being represented by a pointer to its leading Node, then `->next` for that Node is, basically, just another smaller Linked List!

For example, consider the print function

```
void print_list(Node *listptr) {  
    if (listptr != nullptr) {  
        cout << listptr->data << " ";  
        print_list(listptr->next);  
    } else {  
        cout << endl;  
    }  
}
```

With the data from before, if you called `print_list(head)`, first `head->data` would print (Alice), and then `print_list(n1.next)` would be called; that would start by print `n1.next->data = n2.data`, Bob; and so forth.

L20x2_recur.cpp

Linked Lists and Recursion

Write a recursive `int size(Node*)` function. If `head` is a pointer to the first `Node` of a list, then `size(head)` should return the number of `Nodes` in that list.

We can also write a recursive function which takes in a pointer to the head of a list, as well as a `string` named `entry`, and then appends a new `Node` to the end of the list with `entry` as the data.

The signature line of this function is tricky:

```
void append(Node* &listptr, string entry)
```

The first argument should be a pointer-to-`Node`, which explains the `Node*` data type. The function should return nothing – that's why the return type is `void`.

But after the function is done executing, the list should be changed, and so `listptr` needs to be passed by **reference**. Note the order: `*` first, `&` second.

Linked Lists and Recursion

Once you get past the signature, the base case is the trickiest part. The argument `listptr` could be pointing to `nullptr`. In that case, either the list is completely empty, or we have somehow arrived at the end of the list. Either way, we should create a new `Node` for the entry, and have `listptr` point to that.

The created `Node` shouldn't be a local variable – we want it to continue existing after the function is done running. So we should allocate it on the heap.

Finally, for the recursive step: if we are not at the end of our list, append the entry to the end of `listptr->next`.

3. Reversing Linked Lists

A famous interview problem: given a linked list, how can one *reverse* it?
To clarify the problem:

- We wish to *not* create any new Nodes; instead, we simply want to edit the linking of existing Nodes.
- At the end, we want the original head Node to point to nullptr, and we want head to now point to the original tail Node.

Let's do this with a function, NON-recursively (although you can do it recursively as well!).

We'll use THREE pointers: `prev`, `current` and `following`. `current` will be the Node that is currently being "fixed."

We keep `prev` around because, when we edit a Node, its next pointer should be directed to `prev`.

And we keep `following` around since after we fix `current`, we need to know where to go after – and `current->next` will now point backwards.

4. Templates

The core data in our Nodes are strings. But most of the interesting operations that we perform on the data have code that almost entirely does not rely on the fact that our data fields happen to be strings. In fact, you could take every string in the code, and replace it with, say, double, and it ought to work in approximately the same way.

But performing tons of replacements is tedious and error-prone; and moreover, what if we want to work with several different linked lists containing different types of data? Creating several photo-copy classes seems silly.

There is a simple way to overcome this problem: templates! And in fact, you already know a bit about templates, since you know how to declare, for example, vectors, e.g.

```
std::vector<int> x;
```

The data type in the angle brackets is a *type parameter*.

Templates

Before discussing class templates, we will apply templates to functions. These allow for the writing of functions where parameters, return types, and local variable types that would ordinarily have to be fixed to be replaced by type variables.

To create a function template, simply immediately precede your function's declaration or definition with

```
template<typename T>
```

(which is customarily placed on the line above the ordinary signature line). The symbol T can be replaced with any other variable name. It will represent the name of a type that will be utilized within the function body, and which will be different each time the function is called.

You can then call the function by following the function name with <int>, <double>, <string>, or whatever type you want T to be replaced with in any particular call.

L20x3_fn.cpp

(In fact, if the specification of the type is omitted, the compiler will deduce the type from the arguments, if it can.)

Templates

Now, we'll use templates in our struct/class definitions. This is most appropriate for *container* classes: e.g., vectors or linked lists, where the data structure is primarily intended to hold collections of data of a particular type.

To allow your classes to have type parameters, simply begin their declaration with

```
template<typename T>  
struct MyType { ...
```

Then, create objects with a declaration of the form

```
MyType<actual_type> varname;
```

L20x4_struct.cpp

(Notice: when you refer to Node in the declaration of the class, you simply call the class Node. However, outside of the class declaration, you refer to the data type as, e.g., Node<int>.)