Lab 5

16 February 2017

In this lab, you'll work a little with linked lists, and see how to use the unit testing framework with pointer-based data. First, though...

Command line FOTD: `grep`

The `grep` command is a general search tool that lets you find occurrences of some pattern in a whole batch of files. For instance, if you type

```
grep Node Node.h
```

you'll get a listing of every line on which the word `Node` shows up in the file `Node.h` (which is a lot!).

But `grep` is more powerful than that. Its first argument is what's called a "regular expression" or "regex", and lets you search for some pretty complicated things. You can get more information on this on your own, but a few quick tricks:

- By enclosing the pattern in (single) quotes, you can search for strings with spaces in them:

  ```
  grep 'void set' Node.h
  ```

- To match any amount of any text, use the "period asterisk" wildcard (note that for filenames on the command line we use asterisk by itself, but within a regular expression we need the period plus the asterisk):

  ```
  grep 'ItemType.*item' Node.h
  ```

- To match the beginning or end of the line, use caret and dollar-sign respectively.

  ```
  grep '^Node' Node.h
  ```

  will give just those lines that *start* with `Node`.

Vim FOTD: searching (and replacing)

From command mode, if you hit the forward slash key,[1] it's a little bit like colon mode: the cursor moves to the bottom of the screen and awaits further input. But what it's waiting for now is a regular expression to search for.

Having just explained regexes in the context of `grep`, there's not much more to explain here; they work essentially the same way. After the initial slash, you type a regex and hit enter, and Vim will find the next place in the file that matches that regex, or if there are none it will tell you that.

Also inside command mode, the `n` command will repeat the previous search. So pressing `n` repeatedly will cycle through all matches in a file. Using `N` instead goes through matches in reverse order.

The `n` command together with the period command (which repeats the previous command) is a workhorse combination: first, search for a pattern and do something; then alternate `n.n.n.` until you've done your action every place that pattern occurs. Try it in `Node.h`: Let's say that instead of `ItemType` as the stand-in name of the type this object contains, we prefer `Thing`. Press `/` and type `ItemType` (and hit enter) to find some occurrence of that word. Then, type the command `cw` to "change word" and type `Thing` (and hit escape) to complete the change. Now press `n` to go to the next occurrence, and press period to make the same change. Keep pressing `n` and `.` until you've made that change throughout the file.

You can either save and quit (if you prefer `Thing`), or save without quitting (using `:q!`), but other than that change, you should not need to further modify the `Node.h` file for this lab.

Nodes and lists

First, create files `NodeType.h` and `ItemType.h` in your directory for this lab. Nodes are described in section 3.4 in the book, and Items in section 3.2. Make your items contain characters.

Then, create a file `test_NodeFunctions.u` that, for now, just has a fixture that declares and builds at least three linked list structures (*not* `UnsortedTypes`)—

[1] Having trouble remembering which is forward slash and which is backslash? If you imagine them walking across the page from left to right, the forward slash is leaning forward: `/` And the backslash is leaning backward: `\`

one should have just a single element in it, and at least one of them should be three elements or longer.

Remember as you're doing this that the top part of the fixture should have decl-and-init statements that look like

typename varname = initvalue;

and any additional setup goes in the `setup` block.

In the rest of this lab, you'll be writing functions that operate on nodes—again, *not* part of `UnsortedType`. Put them in a file called `NodeFunctions.cpp` (and `NodeFunctions.h`), and test them in the `.u` file you've already made.

Implementing `removeAllLessThan`

In class, we worked out pseudocode for a hypothetical method of `UnsortedType` called `removeAllLessThan`. Here we will implement it as a function, which means we need to make a slight change to its header:

```
NodeType* removeAllLessThan (NodeType* start, ItemType value)
```

The pointer to the starting node has to be given explicitly, and the head of the resulting list has to be returned explicitly (since we don't have any instance variables to work with).

Look at your notes from yesterday and the board photos to remind yourself of our discussions about how to implement this.

Before typing in its implementation, though, make sure to declare it (in the `.h` and test it in the .u file. Your test cases will probably include lines that look something like this:

```
NodeType* result = removeAllLessThan (myExample, ItemType{'J'});
check (result) expect != nullptr;
check (result->info.value) expect == 'K';
check (result->next) expect != nullptr;
check (result->next->info.value) expect == 'X';
check (result->next->next) expect == nullptr;
```

or whatever, depending very much on how you declared `ItemType` and what your example data looks like.

countMatch

Now try writing another one, again making sure to test it, that behaves as follows:

> `countMatch` counts how many elements in the given linked list are exactly equal to the value represented by the given item.

Plan your way through the function design—header and test cases, and you may wish to create an additional example in your fixture to test this one.

When it comes to actually writing the function, there will again be repetition, and in that repetition there are three things we need to account for:

- We don't find it
  (the current node is `nullptr`)
- We find it
  (the current node has it as its value)
- We have to keep looking
  (continue on to the current node's `next`)

As you write your function, keep all three in mind and be sure you don't forget to handle one!

You should now have at least three examples: one that represents no node at all; one with a dead end node that contains just a single element; and one that (indirectly) contains at least three elements (that is, it contains one and points to further `Node`s with the other two elements).

Compile and run your test suite to make sure you haven't made any typoes or other mistakes. Of course, since we still only have stub functions for `linkSearchRec` and `linkSearchIter`, some of the tests should report failures (if not, you need more or better test cases!).

Handing in

This is feeding into class Friday and an upcoming homework; do as much as you can and make note of any questions you have, and bring those with you to class, but no need to run `handin` this week.