CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 17 Feb 22, 2013

Writing Test Code

Lecture Goals

1. Announcements
2. Writing Test Code

Upcoming Homework Assignment

HW #4 **Due: Friday, Feb 22**

B-Tree Design Document

Details are up, and you should be nearly done by now anyway. Be absolutely sure to send the email correctly as detailed in the HW directory, or we might miss it.

Final Test Date Set

Central Command has issued a memo.

The final for CS 2270:

May 8th
1:30pm
RAMY C250 (our regular classroom)

Bachelor of Arts in CS is Open

I sent an email last night about this. The new Bachelor of Arts CS major is now open for enrollment.

For what it's worth, if I had this opportunity back in the day I might have gone BA rather than BS. That's just me.

See the email for details.

Today = Writing Tests

Writing code is good. Writing code that tests your code is great. Writing code that tests your code before or while you write your code is excellent.

Three main strategies:

1] **Interactive**

2] **Custom Driver**

3] **Unit tests**.

Interactive Testing

You type things and it prints stuff out, like a command line. This can be pretty involved, so we'll not do this live in class. It can help you find weird problems. Might look like:

```
> insert 10
Key 10 inserted into B-Tree
> print tree
 [2 3 8 10]
```

Custom Driver

We'll do some of this. It is a main function that you can edit to test various things.

Pro: This is convenient to write and narrow in on vexing problems.

Con: It tends to get stale, or it prints out lots of junk that is hard to read.

Unit Tests

Write code that simulates actual use and forces you to think about edge cases. Each test looks for one thing and is decoupled from functions that aren't the test's focus. (As decoupled as possible.)

Pro. They are the closest thing to ground truth you will get. They are clean, easy to run and interpret.
Con. They can be hard or time-consuming to write. At least, they seem that way, even if they end up saving you time.

Example with Deque

Double-ended Queue is a linked-list like abstract data type whose implementation has these operations:

- insert at end of list: `void push (int val)`
- insert at front of list: `void push_front (int val)`
- remove last element: `int pop ( )`
- remove first element: `int pop_front ( )`
- look at last element: `int peek ( )`
- look at first element: `int peek_front ( )`
- `int size ( )`
- `boolean contains (int val)`

Where to get stuff

Alec's unit testing framework is on the course GitHub:
cs2270 / homeworks / Resources / UTFramework

The Makefile can be adapted from any of the homework assignments (I basically never write a Makefile from scratch).

The Deque code is on GitHub here:
cs2270 / code / deque

My General Strategy

Here's how I do it. Your process can be totally different, that's cool. This is just for example.

I start by gathering the header, the stub of an implementation, the Makefile, and whatever other things are required (like the unit testing framework) and put them into a directory.

First step is to get 'hello world' to work. Really, start there. Don't even start your *real* code yet.

Sanity Check

```
$ make all
<< lots of warnings >>
$ ./deque_driver
Word up, CSCI 2270.
$ ./deque_unittest
```

```
Suite: Deque
|
|    Test: Sanity Check
|    |    - Sane?
|    |    |    I'm insane!
|    Failed!
|
Failed!
```