CS 228: Introduction to Data Structures
Lecture 1
Monday, January 12, 2015

Administrative Stuff

All you need to know about & do for this class is on Blackboard Learn:

https://bb.its.iastate.edu/

This includes:

- Syllabus
- Assignments and assignment submission
- Exam info
- Clarifications
- Discussion thread
- Announcements

What follows is just a summary.

You are responsible for checking the Bb Learn announcements at least every other day.

Lecture Sections

Section	Instructor	Time	Place
A	David F-B	MWF 11-11:50am	Ross 0124
B	Yan-bin J.	MWF 12:10-1pm	MolBio 1414

The two sections will cover the same topics at the same pace. They will have the same exams and assignments and will share TAs.

Recitation Sections

These are one-hour sessions led by members of our crack team of teaching assistants. They offer you an opportunity to ask questions and engage in discussions in a smaller group. We frequently use the recitations to present examples in more detail than is possible during lecture and to review especially important topics.

Section	Time	Place
1	R 10am-10:50am	Pearson 1105
2	R 2:10pm-3pm	Gilman 2109
3	R 1:10pm-2pm	Gilman 2109
4	R 4:10pm-5pm	Sweeney 1126
5	R 3:10pm-4pm	Gilman 2109
6	T 9am-9:50am	Gilman 1810
7	T 2:10pm-3pm	Gilman 2109

Prerequisite

The prerequisite for this course is CS 227. This is a ***hard*** prerequisite. That is, from the beginning, we really expect you to have solid knowledge of what you learned in 227. The first two weeks will be mostly review, but some of you may find you have to swim quickly at first.

As a practical matter, we cannot allow you to take CS 228 without fulfilling the CS 227 prerequisite. This is both for your sake and ours, as this is required to preserve ABET accreditation.

Exams

Midterm exams will be at night:

Exam	Date	Time	Place
1	R 19 Feb	6:45-7:45pm	TBD
2	R 26 Mar	6:45-7:45pm	TBD

The date, time, and place for final will be announced later.

Textbook

The main reference for the course will be the class notes and the supplementary material posted on Bb.

The following text is ***optional***.

Simon Gray, *Data Structures in Java*, Addison-Wesley 2007. (Same as in Spring 2012.)

The book covers all the material seen in class. It also has detailed examples with code. You may be assigned readings from the text, but not problems from it.

Grades

	Weight
Exam 1	17%
Exam 2	17%
Final Exam	26%
Assignments	40%
Total	**100%**

- Weights are approximate, and may be adjusted later.
- Roughly speaking: median grade = B-
- Assignments:
  - Approximately 5
  - Code must compile; zero otherwise
  - Documentation is essential
- Exams
  - Bring your university ID
  - If you cannot attend an exam, you must notify your instructor at least one week prior to the exam to make other arrangements.
- Class and recitation attendance are not mandatory, but strongly recommended

For your reference, here's a table that gives a rough idea of what your letter grade will be, depending on your total score. The final grading scale may be different, but it will not be harsher. (So the table effectively gives you lower bounds.)

Score	**Letter Grade**
at least 88	A
at least 83 but less than 88	A-
at least 78 but less than 83	B+
at least 70 but less than 78	B
at least 65 but less than 70	B-
at least 60 but less than 65	C+
at least 55 but less than 60	C
at least 50 but less than 55	C-
at least 42 but less than 50	D
at least 39 but less than 42	D-
below 39	F

Cheating

1. "No Code Rule": Never have a copy of someone else's program in your possession and never give your program to someone else.

2. Discussing an assignment without sharing any code is generally okay. Helping someone to interpret a compiler error message is an example of permissible collaboration. However, if you get a significant idea from someone, acknowledge them in your assignment.
3. These rules apply to homework and projects. No discussion whatsoever in exams, of course.
4. In group projects, you share code freely within your team, but not between teams.

Now, on to the course . . .

Overview

This course is known as "CS2" in the IEEE curriculum guidelines. The topics are:

- ***Abstract Data Types*** for collections (we'll define what this means later this week)
- ***Data structures*** for implementing ADTs. This will include things called *linked lists, stacks, queues, trees, heaps, and graphs*
- ***Algorithms*** for manipulating data structures, and the analysis of those algorithms
- Continued practice of object-oriented programming principles
- Longer and more difficult programming projects
- Some gory details of Java and the Java Collections Framework

The first three bullets are a set of elegant ideas formulated in the 1970s and 1980s that have become a crucial part of the repertoire of every software developer. These ideas are fundamental and essentially independent of the programming language you use.

The importance of these ideas is underscored by the fact that they are built into their libraries of most modern programming languages. In fact, the ***Java Collections Framework*** contains implementations of many fundamental data structures and algorithms.

Java Collections are an important part of this course. Using them enables us to build on a set of very well-designed Java interfaces. ***Code reuse*** is an essential part of programming.

On the other hand, as computer scientists or engineers, we need to learn ***how*** collections are designed and implemented. Doing this may sometimes give us the odd sense of reinventing the wheel.

An Example

Here is an example to illustrate what this course is about.

A ***whitelist*** is a list of individuals that are being provided a particular privilege, service, or access. Those on the list will be accepted or approved. For instance, credit card companies maintain customer account numbers in a whitelist — transactions involving those accounts would be considered valid; the others may be considered suspicious. As another example, spam filters often have whitelists of senders and keywords to look for in emails. Mail from the listed email addresses, domains, and/or IP address will always be allowed.

Suppose, for simplicity, that our whitelist is a collection of integers (e.g., bank accounts). To be useful, this collection should support at least the following operations.

- Add a new integer to the collection.
- Determine if a given integer is contained in the collection.
- Return the number of integers in the collection.

Notice that we have defined

- a ***collection of things*** (integers) and
- the ***behavior*** of the operations on the collection.

We have not specified ***how*** the collection is going to be implemented. This is an example of an ***abstract data type*** (ADT).

Java, along with other programming languages, offers developers a way to specify an ADT without constraining its implementation. In Java, this is usually done via an ***interface***.

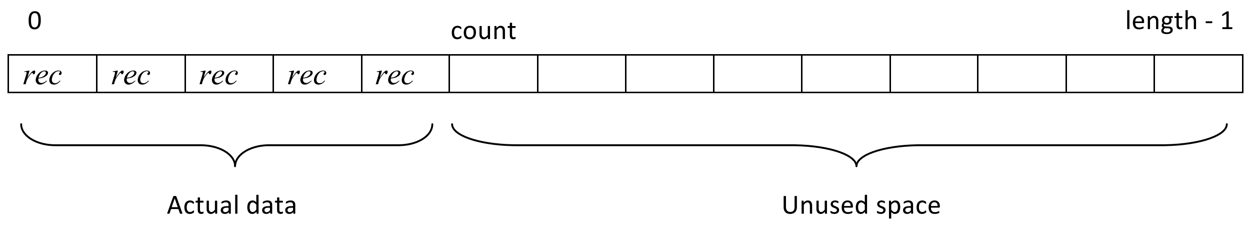
```
public interface IntCollection
{
  void add(int k);
   // adds a new integer to the set

 boolean contains(int k);
   // returns true if k is in collection;
   // returns false otherwise.

  int size();
   // returns the number of integers in
   // the collection.
}
```

Assuming we have the specification for `IntCollection`, we can write ***client*** code that uses it. For instance, suppose `whiteList` is an object of type `IntCollection` and suppose that `inputList` is a list of integers Then, the following (pseudo) code prints all elements of `inputList` that are not in `whiteList` — this implements what is known as ***whitelist filtering***.

```
for each x in inputList
   if !whiteList.contains(x)
      print x
```

Now, at some point you have to provide actual code to ***implement*** the `IntCollection` interface. The concrete method for storing `IntCollection` in a computer is called a ***data structure***. A simple data structure for storing collections of items of the same type is an array. An ArrayList, which is just an expandable array, serves a similar purpose. You have seen arrays and ArrayLists in CS 227.

An array representation of an `IntCollection` would look like this at any given time:



There are still implementation decisions to make. Significantly, we need to choose the ***algorithms*** to implement the operations. An algorithm is just a step-by-step procedure for doing something; e.g., determining whether a number is contained in a collection.

Here are two possible approaches:

Approach 1: Unsorted array + sequential search

`add(k):` put k in the next available array cell, then increment count.

`contains(k):` Use ***sequential search***: Scan through the array from left to right; stop when you either find k or there are no more items to examine.

Approach 2: Sorted array + binary search.

`add(k):` find the first integer greater priority, shift everything right, insert new integer.

`contains(k):` Use ***binary search***. We'll explain this method later in this course. The idea, though, resembles what you might do if you're looking for a specific name in a sorted list of names. First, go to the middle of the list. If the name you're looking for is there, stop and return true. Otherwise, repeat the process with the upper or lower half of the list.

Which approach is better? More fundamentally: what do we mean by "better"?

- Faster?
- Uses less memory?
- Easier to develop?

We need some criteria and techniques for ***analyzing*** the algorithms for accessing data, in order to make intelligent design decisions.

And what if we also want to delete elements? I.e., suppose we want to have the following method.

```
  boolean remove(int k);
   // removes k from collection; returns
   // false if k is not present.
```

In this case, there are other data structures that are faster than arrays. One of these is are ***binary search trees***; we will study binary search trees in the second half of this course

Summing up

The three ideas we just saw,

1. defining an ***abstract data type*** using an ***interface***,
2. coming up with a ***data structure*** to implement it, and
3. ***analyzing*** the implementation

will be the recurring themes in this course.