CSE332: Data Abstractions

Lecture 1: Introduction; ADTs; Stacks/Queues

Dan Grossman

Spring 2012

Welcome!

We have 10 weeks to learn *fundamental data structures and algorithms for organizing and processing information*

  - "Classic" data structures / algorithms and how to analyze rigorously their efficiency and when to use them
  - Queues, dictionaries, graphs, sorting, etc.
  - Parallelism and concurrency (!)

Today in class:

- Course mechanics
- What this course is about
  - And how it fits into the CSE curriculum
- Start (finish?) ADTs, stacks, and queues
  - Largely review

Concise to-do list

In next 24-48 hours:

- Adjust class email-list settings
- Email homework 0 (worth 0 points) to me
- Read all course policies
- Read/skim Chapters 1 and 3 of Weiss book
  - Relevant to Project 1, due next week
  - Will start Chapter 2 on Wednesday

Possibly:

- Set up your Eclipse / Java environment for Project 1
  - Thursday’s section will help

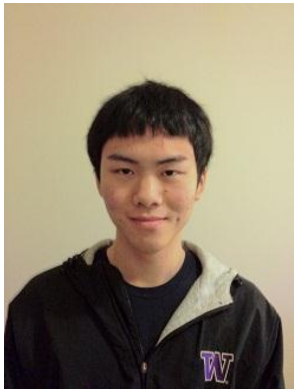
http://www.cs.washington.edu/education/courses/cse332/12sp/

Course staff

Dan Grossman

Tyler Robison

Stanley Wang



Dan: Faculty, “341 guy”, loves 332 too, did parallelism/concurrency part

Tyler: Grad student, TAed 332 many times, taught it Summer 2010

Stanley: Took 332 last quarter

Office hours, email, etc. on course web-page

Communication

- Course email list: **`cse332a_sp12@u`**
  - Students and staff already subscribed
  - You must get announcements sent there
  - Fairly low traffic
- Course staff: **`cse332-staff@cs`** plus individual emails
- Discussion board
  - For appropriate discussions; TAs will monitor
  - Optional, won't use for important announcements
- Anonymous feedback link
  - For good and bad: if you don't tell me, I don't know

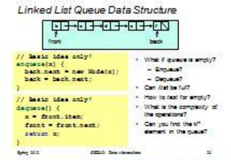
Course meetings

- Lecture (Dan)
  - Materials posted (sometimes afterwards), but take notes
  - Ask questions, focus on key ideas (rarely coding details)

- Section (Tyler)
  - Often focus on software (Java features, tools, project issues)
  - Reinforce key issues from lecture
  - Answer homework questions, etc.
  - An important part of the course (not optional)

- Office hours
  - Use them: *please visit me*
  - Ideally not *just* for homework questions (but that's great too)

Course materials



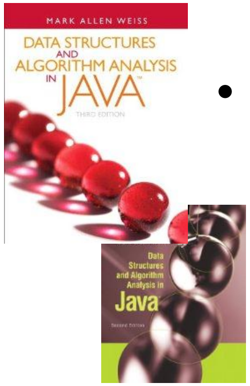
Linked List Queue Data Structure

```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

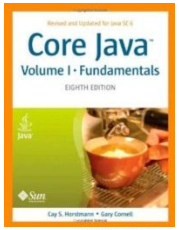
```
// Basic idea only!
dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

- What if queue is empty?
  - Enqueue?
  - Dequeue?
- Can *list* be full?
- How to test for empty?
- What is the complexity of the operations?
- Can you find the k^{th} element in the queue?

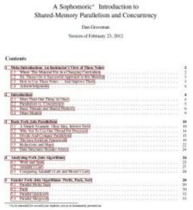
- All lecture and section materials will be posted
  - But they are visual aids, not always a complete description!
  - If you have to miss, find out what you missed



- Textbook: Weiss 3rd Edition in Java
  - Good read, but only responsible for lecture/section/hw topics
  - Will assign homework problems from it
  - 3rd edition improves on 2nd, but we'll support the 2nd



- Core Java book: A good Java reference (there may be others)
  - Don't struggle Googling for features you don't understand
  - Same book recommended for CSE331



A Sophomoric* Introduction to
Shared-Memory Parallelism and Concurrency

Dan Grossman

Version of February 23, 2012

Contents

- Parallelism / concurrency units in separate free resources designed for 332

Course Work

- 8 written/typed homeworks (25%)
  - Due at beginning of class each Friday (not this week)
  - No late homeworks accepted
  - Often covers through Monday before it's due

- 3 programming projects (with phases) (25%)
  - First phase of Project 1 due in 9 days
  - Use Java and Eclipse (see this week's section)
  - One 24-hour late-day for the quarter
  - Projects 2 and 3 will allow partners
  - Most of the grade is code design and write-up questions

- Midterm Friday April 27 (20%)

- Final Tuesday June 5 (25%)

Collaboration and Academic Integrity

- Read the course policy very carefully
  - Explains quite clearly how you can and cannot get/provide help on homework and projects

- Always explain any unconventional action on your part
  - When it happens, when you submit, not when asked

- I have promoted and enforced academic integrity since I was a freshman
  - I offer great trust but with little sympathy for violations
  - Honest work is the most important feature of a university

Unsolicited advice

- Get to class on time!
  - Instructor pet peeve (I will start and end promptly)
  - First 2 minutes are *much* more important than last 2!
  - April 27 will prove beyond any doubt you are capable

- Learn this stuff
  - You need it for so many later classes/jobs anyway
  - Falling behind only makes more work for you

- Have fun
  - So much easier to be motivated and learn

Today in Class

- Course mechanics: Did I forget anything?

- What this course is about
  - And how it fits into the CSE curriculum

- Start (finish?) ADTs, stacks, and queues
  - Largely review

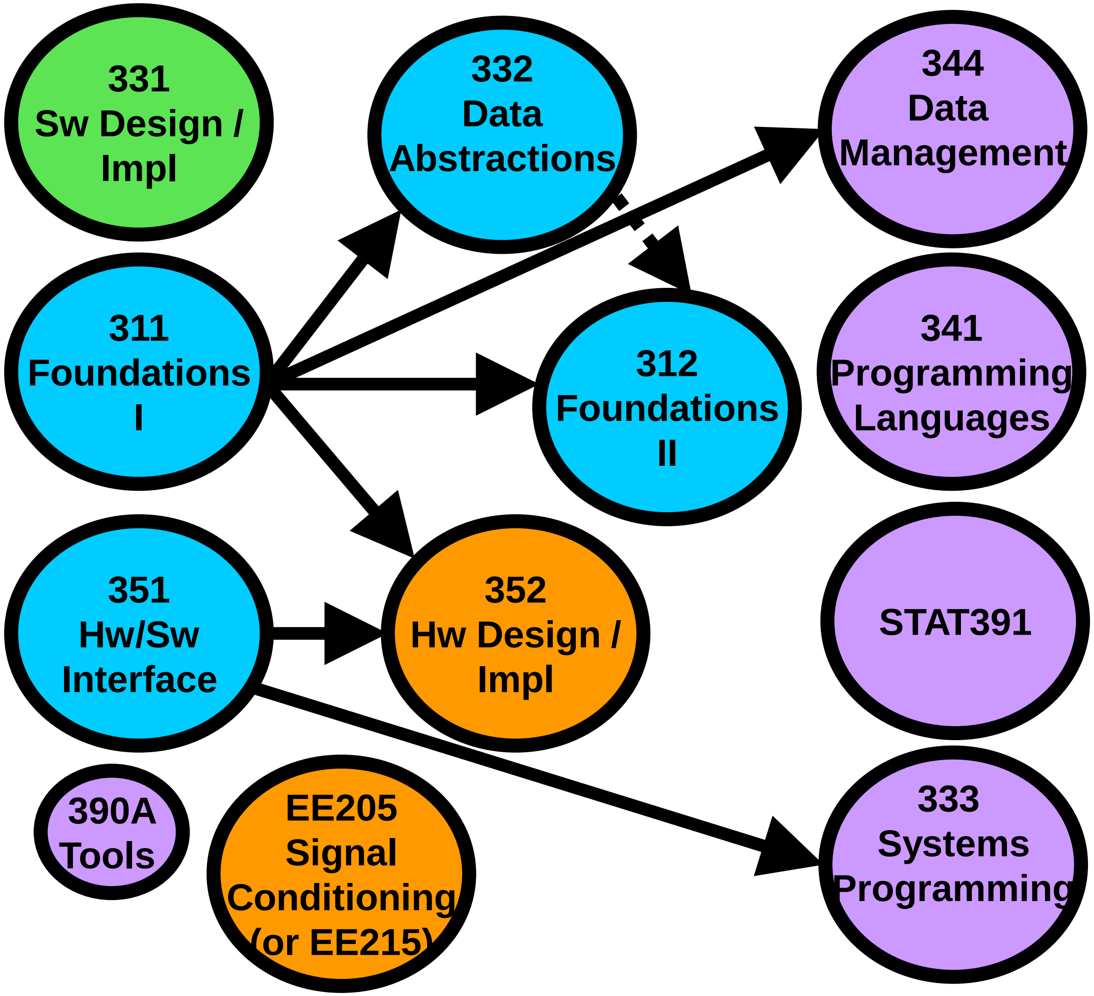
Data Structures + Threads

- About 70% of the course is a “classic data-structures course”
  - Timeless, essential stuff
  - Core data structures and algorithms that underlie most software
  - How to analyze algorithms

- Plus a serious first treatment of programming with *multiple threads*
  - For *parallelism*: Use multiple processors to finish sooner
  - For *concurrency*: Correct access to shared resources
  - Will make many connections to the classic material

Where 332 fits

- 🟦 required
- 🟩 CS required
- 🟧 CompE required
- 🟪 not required
- ➡ pre-req
- ⇢ co-req or pre-req



331 Sw Design / Impl

332 Data Abstractions

344 Data Management

311 Foundations I

312 Foundations II

341 Programming Languages

351 Hw/Sw Interface

352 Hw Design / Impl

STAT391

390A Tools

EE205 Signal Conditioning (or EE215)

333 Systems Programming

- Also the most common pre-req among 400-level courses
  - And essential stuff for many internships

What is 332 is about

- Deeply understand the basic structures used in all software
  - Understand the data structures and their trade-offs
  - Rigorously analyze the algorithms that use them (math!)
  - Learn how to pick "the right thing for the job"

- Experience the purposes and headaches of multithreading

- Practice design, analysis, and implementation
  - The elegant interplay of "theory" and "engineering" at the core of computer science

Goals

- Be able to make good design choices as a developer, project manager, etc.
  - Reason in terms of the general abstractions that come up in all non-trivial software (and many non-software) systems
- Be able to justify and communicate your design decisions

Dan’s take:

3 years from now this course will seem like it was a waste of your time because you can’t imagine not “just knowing” every main concept in it

- Key abstractions computer scientists and engineers use almost every day
- A big piece of what separates us from others

Data structures

(Often highly *non-obvious*) ways to organize information to enable *efficient* computation over that information

- Key goal over the next week is introducing asymptotic analysis to *precisely* and *generally* describe efficient use of time and space

A data structure supports certain *operations*, each with a:

- Meaning: what does the operation do/return
- Performance: how efficient is the operation

Examples:

- ***List*** with operations **`insert`** and **`delete`**
- ***Stack*** with operations **`push`** and **`pop`**

Trade-offs

A data structure strives to provide many useful, efficient operations

But there are unavoidable trade-offs:

- Time vs. space
- One operation more efficient if another less efficient
- Generality vs. simplicity vs. performance

That is why there are many data structures and educated CSEers internalize their main trade-offs and techniques

- And recognize logarithmic < linear < quadratic < exponential

Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations
- Algorithm
  - A high level, language-independent description of a step-by-step process
- Data structure
  - A specific family of algorithms for implementing an ADT
- Implementation of a data structure
  - A specific implementation in a specific language

Example: Stacks

- The ***Stack*** ADT supports operations:
  - **`isEmpty`**: have there been same number of pops as pushes
  - **`push`**: takes an item
  - **`pop`**: raises an error if isEmpty, else returns most-recently pushed item not yet returned by a pop
  - … (possibly more operations)

- A Stack data structure could use a linked-list or an array or something else, and associated algorithms for the operations

- One implementation is in the library **`java.util.Stack`**

Why useful

The Stack ADT is a useful abstraction because:

- It arises all the time in programming (e.g., see Weiss 3.6.3)
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion (see text)

- We can code up a reusable library

- We can communicate in high-level terms
  - “Use a stack and push numbers, popping for operators…”
  - Rather than, “create a linked list and add a node when…”

The Queue ADT

- Operations

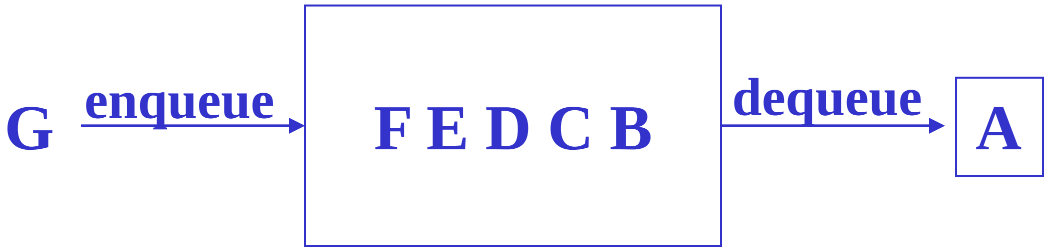
  `create`

  `destroy`

  `enqueue`

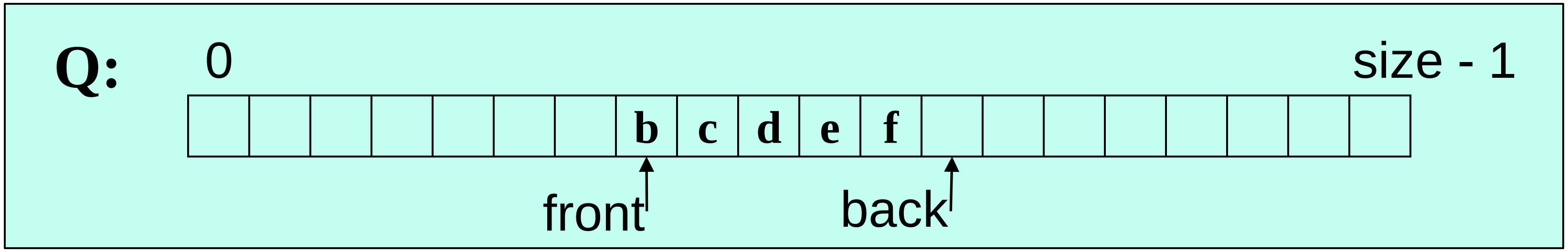
  `dequeue`

  `is_empty`



- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)
- Just as useful and ubiquitous

Circular Array Queue Data Structure

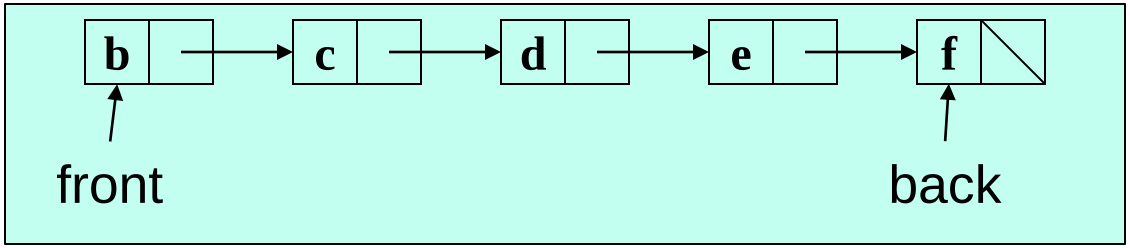


```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// Basic idea only!
dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- What if ***array*** is full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the kth element in the queue?

Linked List Queue Data Structure



```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// Basic idea only!
dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- Can ***list*** be full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?

Circular Array vs. Linked List

Array:

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast
- Constant-time access to k^{th} element

- For operation insertAtPosition, must shift all later elements
  - Not in Queue ADT

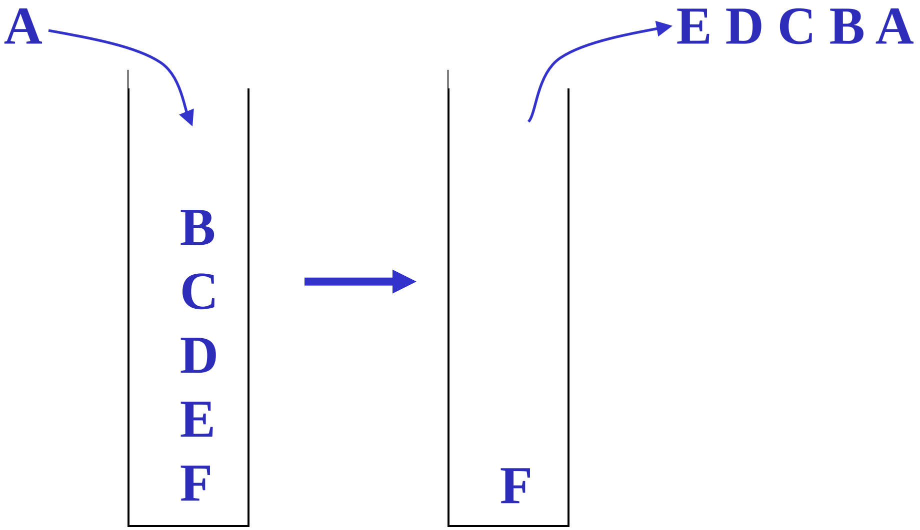
List:

- Always just enough space
- But more space per element
- Operations very simple / fast
- No constant-time access to k^{th} element

- For operation insertAtPosition must traverse all earlier elements
  - Not in Queue ADT

This is something every trained computer scientist knows in his/her sleep – it's like knowing how to do arithmetic

The Stack ADT



Operations:

- **`create`**
- **`destroy`**
- **`push`**
- **`pop`**
- **`top`**
- **`is_empty`**

Can also be implemented with an array or a linked list

- This is Project 1!
- Like queues, type of elements is irrelevant
  - Ideal for Java’s generic types (section and Project 1B)