# CSE332: Data Abstractions

## Lecture 1: Introduction; Stacks/Queues

Tyler Robison

Summer 2010

# Welcome to 332!

What we're going to be doing this quarter:

- Study many common data structures & algorithms that underlie most computer systems, for instance:
  - Btrees -> Databases
  - Queues -> Printer queue
  - Stacks -> Program call-stack
  - Hashtables, sorting algorithms, graphs, etc.
- Learn to rigorously analyze them and think carefully about what to use when: Uses, limitations, efficiency, etc.
  - Asymptotic analysis -> shows up everywhere in CS
- Study the increasingly important areas of parallelism and concurrency, and relevance to algorithms/data-structures

# Today in class:

- Course mechanics
- What this course is about
  - How it differs from 326
- Abstract Data Types
- Start (finish?) stacks and queues (largely review)

# About us

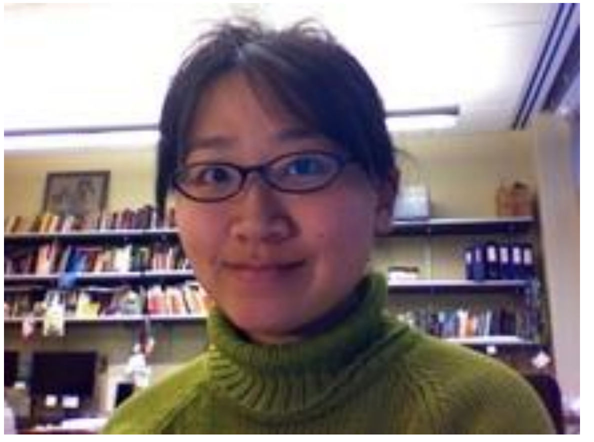
Course Staff:

Tyler Robison

Office hours:

Wednesday 2:00-3:00 &
by appointment
Room: CSE 212

Sandra Fan



Office hours:

Thursday 12:00-1:00
Room: CSE 218

# To-do

Your to-do:

- Make sure you get mail sent to

  cse332a_su10 at u.washington.edu
- Read all course policies
- Read/skim Chapters 1 and 3 of Weiss book
  - Relevant to Project 1, due next week (don't worry; it's not too bad)
  - Relevant to Hw 1, due next week
  - Will start Chapter 2 on Wednesday
- Possibly set up your Eclipse / Java environment for the first project
  - Thursday's section will help
- Check out the website:

http://www.cs.washington.edu/education/courses/cse332/10su/

# Staying in touch

- Course email list: cse332a_su10@u
  - Students and staff already subscribed (in theory – let me know)
  - Used for announcements
  - Fairly low traffic
- Course staff: cse332-staff@cs to send to both Sandra & myself
  - Questions, comments, etc.
- Message Board
  - Posing questions, discussing material
  - Sandra & I will try to check it on a regular basis
- Anonymous feedback link on webpage
  - For good and bad: if you don't tell me, I don't know

# Course materials

- Lectures:
  - First exposure to material
  - Presentation of algorithms, proofs, etc.
  - Provide examples, asides
- Section:
  - Programming details (Eclipse, generics, junit, ForkJoin framework)
  - Practice with algorithms: Given the stuff we're going to cover, practice is definitely important
- Main Textbook: Weiss 2nd Edition in Java
- Optional Textbook: Core Java book: A good Java reference (there may be others)
- Parallelism/Concurrency material not in either book (or any appropriate one)
  - However, Dan Grossman wrote up excellent slides and notes for those topics

# Course Work

- 7 to 8 written/typed homeworks (25%)
  - Due at beginning of class each Friday (but not this week)
  - No late homework, please
    - Even if you don't have time to do it all, turn in something – some credit is better than no credit
- 3 programming projects (some with phases) (25%)
  - Use Java and Eclipse (see this week's section)
  - You've got one 24-hour late-day for the quarter
  - First project due next week (rather lighter than the others)
  - Projects 2 and 3 will allow partners; use of SVN encouraged
- Midterm: July 19th (20%)
- Final: August 20th (25%)
- 5% to your strongest above

# Collaboration and Academic Integrity

- Working together is fine – even encouraged – but keep discussions at a high level, and always prepare your own solutions
- Read the course policy (on the website)
  - Explains how you can and cannot get/provide help on homework and projects

# How 332 differs from 326

- 332 is about *70%* of the material from 326
  - Covers the same general topics, and the important algorithms/data-structures
  - Cuts out some of the alternative data-structures, and some less important ones
    - You can probably live a full & meaningful life without knowing what a binomial queue is
- Biggest new topic: a serious treatment of programming with *multiple threads*
  - For *parallelism*: To use multiple processors to finish sooner
  - For *concurrency*: Allow properly synchronized access to shared resources

# Data structures

(Often highly *non-obvious*) ways to organize information in order 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
  - 'Big Oh' notation used frequently in CS: O(n), O(logn), O(1), etc.

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

- Purpose: 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

Often no clear-cut 'best': there are usually 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

- Recognize the right tool for the job
- And recognize logarithmic < linear < quadratic < exponential

# Terminology

- Abstract Data Type (ADT)
  - Mathematical description of a “thing” with set of operations on that “thing”; doesn’t specify the details of how it’s done
    - Ex, Stack: You push stuff and you pop stuff
      - Could use an array, could use a linked list
- Algorithm
  - A high level, language-independent description of a step-by-step process
    - Ex: Binary search
- Data structure
  - A specific family of algorithms & data for implementing an ADT
    - Ex: Linked list stack
- Implementation of a data structure
  - A specific implementation in a specific language

# Example: Stacks

- The ***Stack*** ADT supports operations:
  - **`isEmpty`**: initially true, later 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
  - … (Often some 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 ADT is a useful abstraction

The Stack ADT is a useful abstraction because:

- It arises all the time in programming (see text for more)
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
- Common ideas; 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…"
- We as humans think in abstractions

# The Queue ADT

- Operations

  **`enqueue`**

  **`dequeue`**

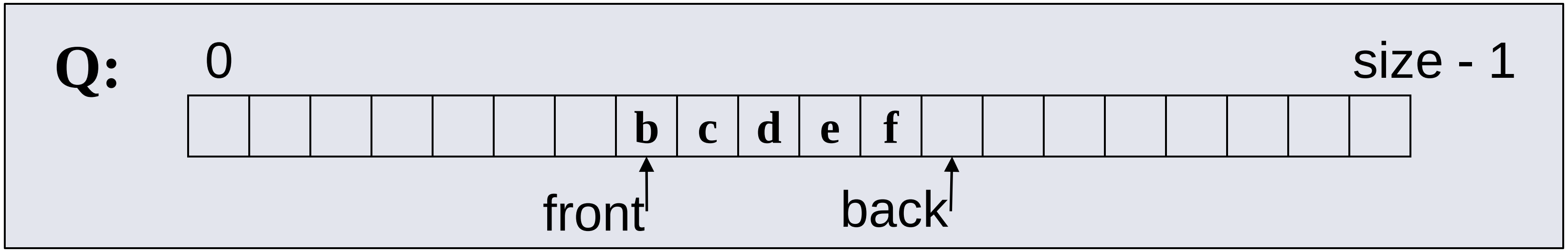
  **`is_empty`**

  **`create`**

  **`destroy`**


- 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

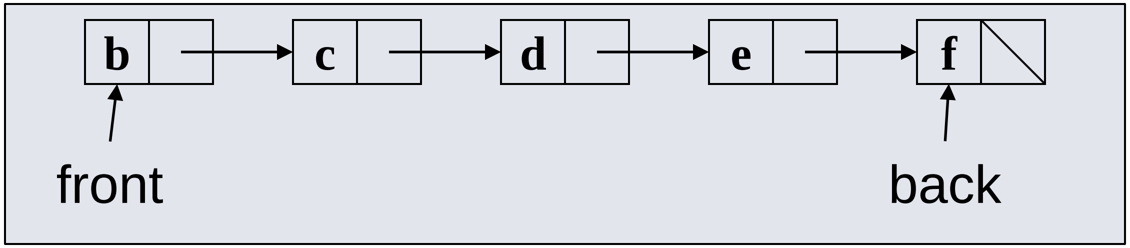



```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// 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 $k^{th}$ 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:

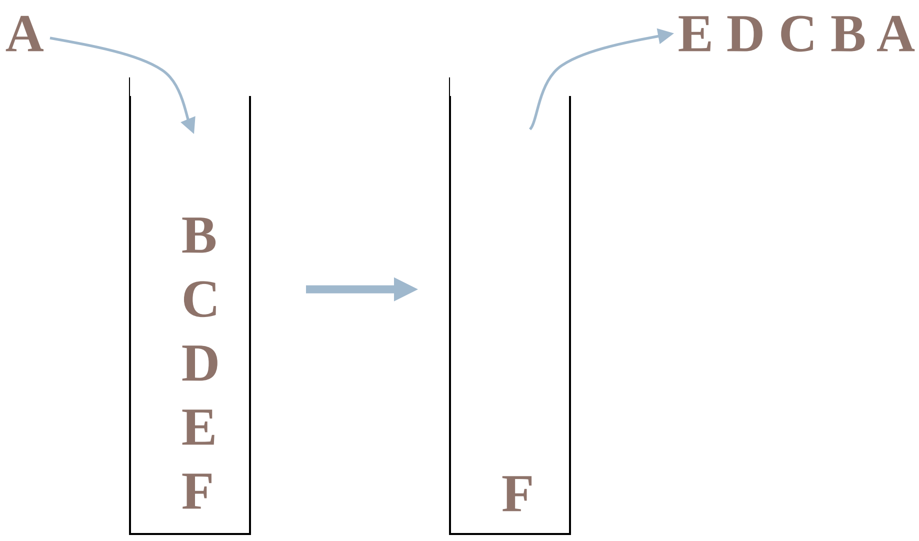
- Always just enough space
- But more space per element
- Operations very simple / fast
- No constant-time access to $k^{th}$ element

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

# The Stack ADT

- Operations

```
 create
destroy
push
pop
top (also 'peek')
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 (covered in section; important for project 1)