CSE373: Data Structures and Algorithms

Lecture 1: Introduction; ADTs; Stacks/Queues

Kevin Quinn

Fall 2015

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.

Today in class:

- Introductions and course mechanics
- What this course is about
- Start *abstract data types* (ADTs), *stacks*, and *queues*
  - Largely review

Concise to-do list

In next 24-48 hours:

- Verify that you have received an email from me!
- Take homework 0 (worth 0 points) as Catalyst quiz
- Read all course policies
- Read/skim Chapters 1 and 3 of Weiss book
  - Relevant to Homework 1, due next week
  - Will start Chapter 2 fairly soon

Possibly:

- Set up your Java environment for Homework 1

http://courses.cs.washington.edu/courses/cse373/15au/

Course staff

Instructor: Kevin Quinn, kchq@cs.washington.edu

TA: Eden Ghirmai ghirme@cs.washington.edu

TA: Megan Hopp hoppm@cs.washington.edu

TA: Andy Li boruil@cs.washington.edu

TA: Rahul Nadkarni rahuln@cs.washington.edu

TA: Hunter Schafer hschafer@cs.washington.edu

TA: Rocne Scribner rocnes@cs.washington.edu

TA: Yunyi Song bessieyy@cs.washington.edu

TA: Mauricio Vinagre Hernandez mvh92@cs.washington.edu

TA: Hunter Zahn hzahn93@cs.washington.edu

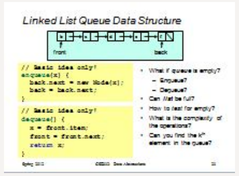
Communication

- Course email list: **`cse373a_au15@`**`u.washington.edu`
  - Students and staff already subscribed
  - You must get announcements sent there
  - Fairly low traffic
- Course staff: **`cse373-staff@`**`cs.washington.edu` plus individual emails
- Discussion board
  - For appropriate discussions; TAs will monitor
  - Encouraged, but won't use for important announcements
- Anonymous feedback link
  - For good and bad: if you don't tell me, I don't know

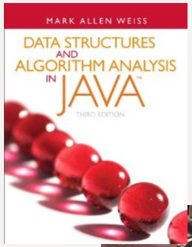
Course meetings

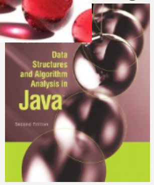
- Lecture
  - Materials posted, but take notes
  - Ask questions, focus on key ideas (rarely coding details)
- Optional meetings on Tuesday/Thursday afternoons
  - Will post rough agenda roughly a day or more in advance
  - Help on programming/tool background
  - Helpful math review and example problems
  - Again, optional but helpful
  - May cancel some later in course (experimental)
- Office hours
  - Use them: *please visit me*
  - Ideally not *just* for homework questions (but that's OK too)

Course materials

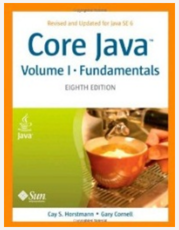


- All lecture and section materials will be posted
  - But they are visual aids, not always a complete description!
  - If you have to miss, find out what you missed





- Textbook: Weiss 3^{rd} Edition in Java
  - Good read, but only responsible for lecture/hw topics
  - 3^{rd} edition improves on 2^{nd}, but we'll support the 2^{nd}



- A good Java reference of your choosing?
  - Don't struggle Googling for features you don't understand?

Computer Lab

- College of Arts & Sciences Instructional Computing Lab
  - http://depts.washington.edu/aslab/
  - Or your own machine

- Will use Java for the programming assignments

- Eclipse is recommended programming environment

Course Work

- 6 homeworks (50%)
  - Most involve programming, but also written questions
  - Higher-level concepts than "just code it up"
  - First programming assignment due week from Friday
- Midterm(s) (20%): TBD. Will announce more about these in the coming week.
- Final exam: Tuesday December 15, 2:30 – 4:20 KNE 120 (30%)

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

Some details

- You are expected to do your own work
  – Exceptions (group work), if any, will be clearly announced
- Sharing solutions, doing work for, or accepting work from others is cheating
- Referring to solutions from this or other courses from previous quarters is cheating
- But you can learn from each other: see the policy

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!
  - Midterms will prove beyond any doubt you are capable

- Learn this stuff
  - It is at the absolute core of computing and software
  - 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

- Start *abstract data types* (ADTs), *stacks*, and *queues*
  - Largely review

Data Structures

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms
- *May have time for other brief exposure to topics, maybe parallelism*

Assumed background

- Prerequisite is CSE143

- Topics you should have a basic understanding of:
  - Variables, conditionals, loops, methods, fundamentals of defining classes and inheritance, arrays, single linked lists, simple binary trees, recursion, some sorting and searching algorithms, basic algorithm analysis (e.g., $O(n)$ vs $O(n^2)$ and similar things)

- We can fill in gaps as needed, but if any topics are new, plan on some extra studying

What 373 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"
  - More thorough and rigorous take on topics introduced in CSE143 (plus more new topics)

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

- More programming experience (as a way to learn)

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:

- Key abstractions used almost every day in just about anything related to computing and software
- It is a vocabulary you are likely to internalize permanently

Data structures

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

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

We ask ourselves questions like:

- Does this support the operations I need efficiently?
- Will it be easy to use, implement, and debug?
- What assumptions am I making about how my software will be used? (E.g., more lookups or more inserts?)

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 organization of data and 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 empty, 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:

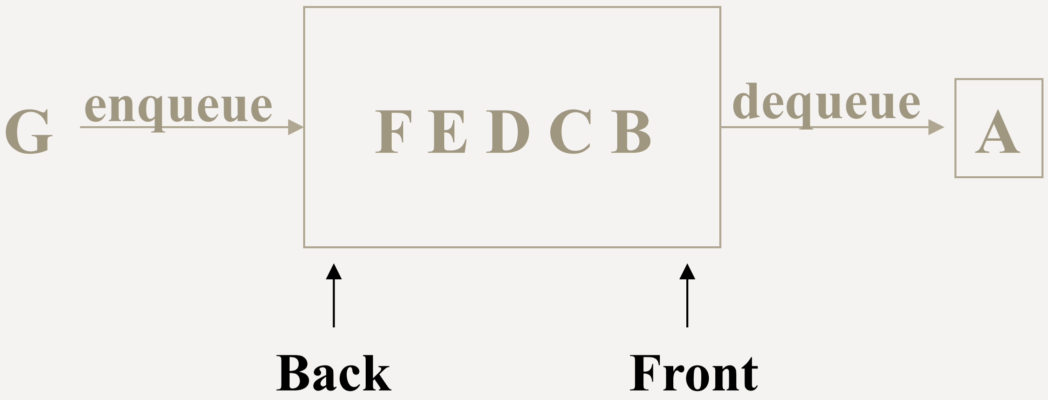
- It arises all the time in programming (e.g., see Weiss 3.6.3)
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion (see text)

- 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…"

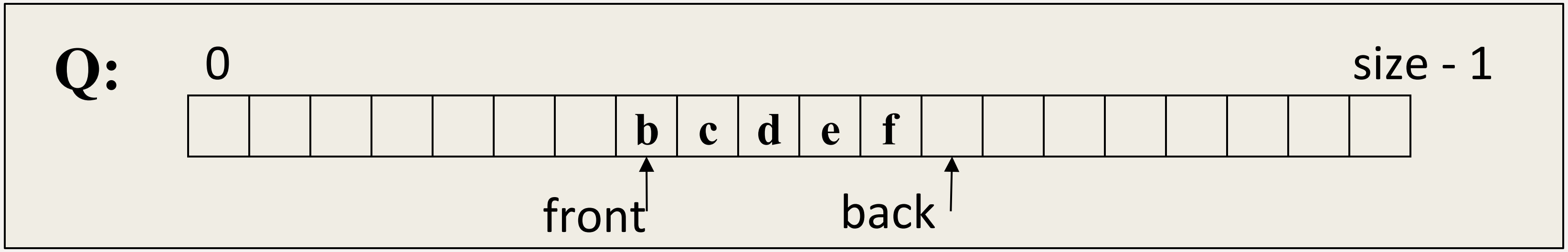
The Queue ADT

- 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

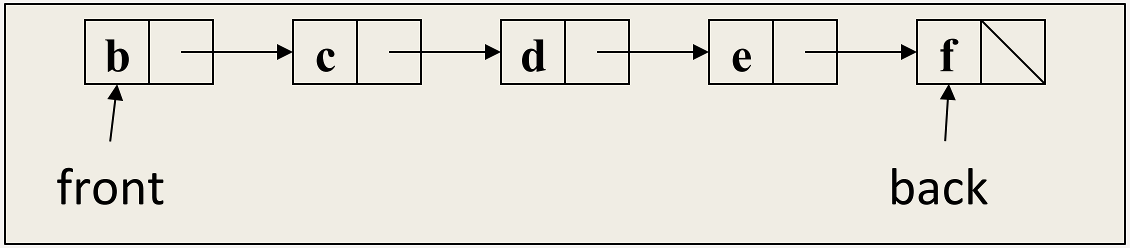



```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// 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 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?

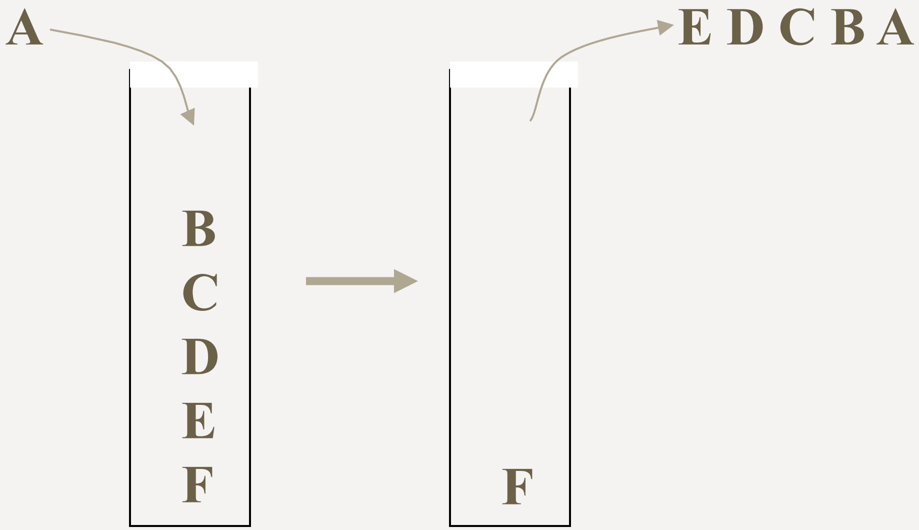
Circular Array vs. Linked List

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

This is stuff you should know after being awakened in the dark

The Stack ADT



Operations:

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

Can also be implemented with an array or a linked list

- This is Homework 1!
- Like queues, type of elements is irrelevant