CSE373: Data Structures and Algorithms

Lecture 1: Introduction; ADTs; Stacks/Queues

Linda Shapiro

Spring 2016

Registration

- We have 180 students registered and others who want to get in.
- If you're thinking of dropping the course please decide *soon!*

Waiting students

- Please sign up on the paper waiting list after class, so I know who you are.
- If you need the class to graduate this June, put that down, too.
- The CSE advisors and I will decide by end of Friday who gets in.

Welcome!

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

- "Classic" data structures / algorithms
- How to rigorously analyze their efficiency
- How to decide 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

To-do list

In next 24-48 hours:

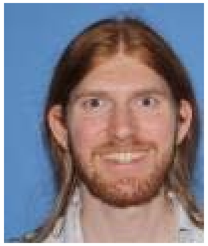
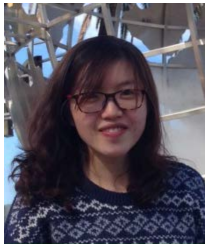
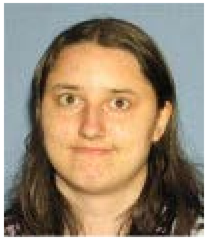
- Read the web page
- Read all course policies
- Read Chapters 3.1 (lists), 3.6 (stacks) and 3.7 (queues) of the Weiss book
  - Relevant to Homework 1, due next week
- Set up your Java environment for Homework 1

http://courses.cs.washington.edu/courses/cse373/16sp/

Course staff

Linda Shapiro
CSE Professor with research in computer vision.
Have taught CS&E for 40 years.
First course I ever taught was Data Structures.



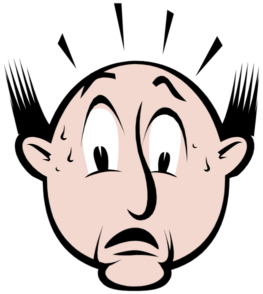




Ezgi Mercan Mert Saglam Ben Jones Shenqi Tang Bran Hagger Chloe Lathe Lysia Li

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

Communication

- Course email list: cse373a_sp16@**u.washington.edu**
  - Students and staff already subscribed
  - You must get announcements sent there
  - Fairly low traffic; You don't post there
- 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, but please be gentle.



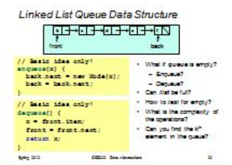
Course meetings

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

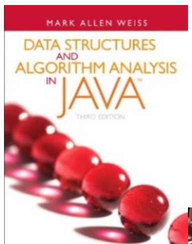
- Optional help sessions
  - Help on programming/tool background
  - Helpful math review and example problems
  - Again, optional but helpful
  - **Fill out our online poll for best times**

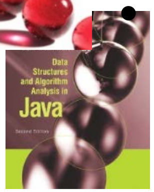
- Office hours
  - Use them: *please visit me for talking about course concepts or just CSE in general.*
  - *See the TAs for Java programming questions.*

Course materials



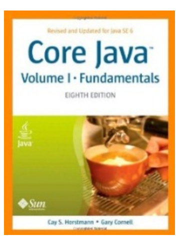
- All lecture will be posted
  - But they are visual aids, not always a complete description!
  - If you have to miss, find out what you missed





- Textbook: Weiss 3rd Edition in Java

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



- Constantly skipping class is not good for your grade.

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

- 5-6 homeworks (50%)
  - Most involve programming, but also written questions
  - Higher-level concepts than “just code it up”
  - First programming assignment due next week

- Midterm Week of May 2, in class (20%)
- Final exam: Tuesday June 7, 2:30-4:20PM (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

- The CSE Department and I take academic integrity extremely seriously.

- IF YOU'RE NOT SURE, THEN ASK!

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

What this course will cover

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms
- Advanced Data Structures and Applications

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

You will learn the key abstractions used almost every day in just about anything related to computing and software.

- This is not a course about Java! We use Java as a tool, but the data structures you learn about can be implemented in any language.

Let's start!

Data structures

A data structure is a (often *non-obvious*) way 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 (and reuse), 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
  - Not concerned with implementation details
- 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`**: adds an item to the top of the stack
  - **`pop`**: raises an error if empty, else removes and returns most-recently pushed item not yet returned by a pop
  - What else? **top (java peek)**

- A Stack data structure could use a linked-list or an array 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:

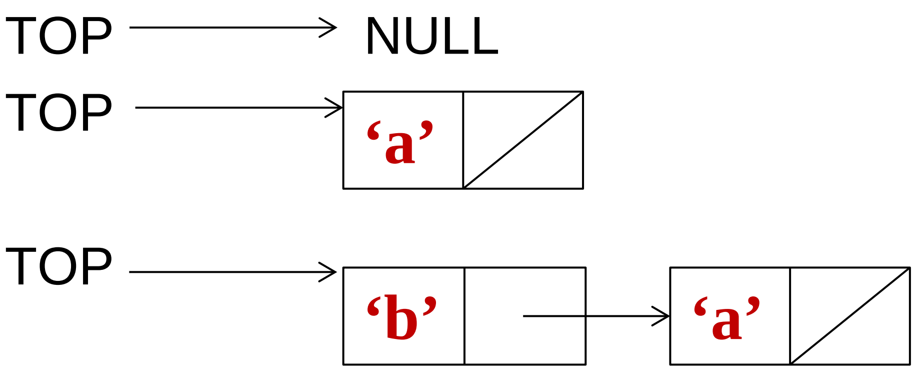
- It arises all the time in programming (e.g., see Weiss 3.6.3)
  - Recursive function calls
  - Balancing symbols in programming (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion (see text)

- We can code up a reusable library

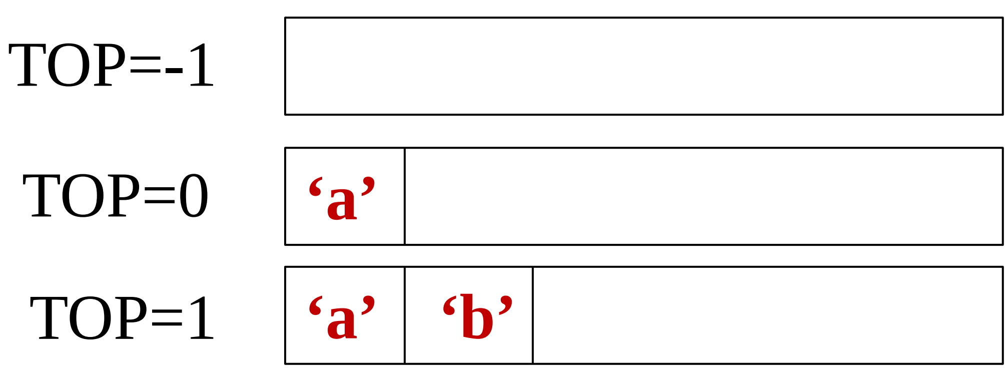
- We can communicate in high-level terms
  - "Use a stack and push numbers, popping for operators…"
  - Rather than, "create an array and keep indices to the…"

Stack Implementations

- stack as a linked list



- stack as an array



The Queue ADT

- Operations

  `create`

  `destroy`

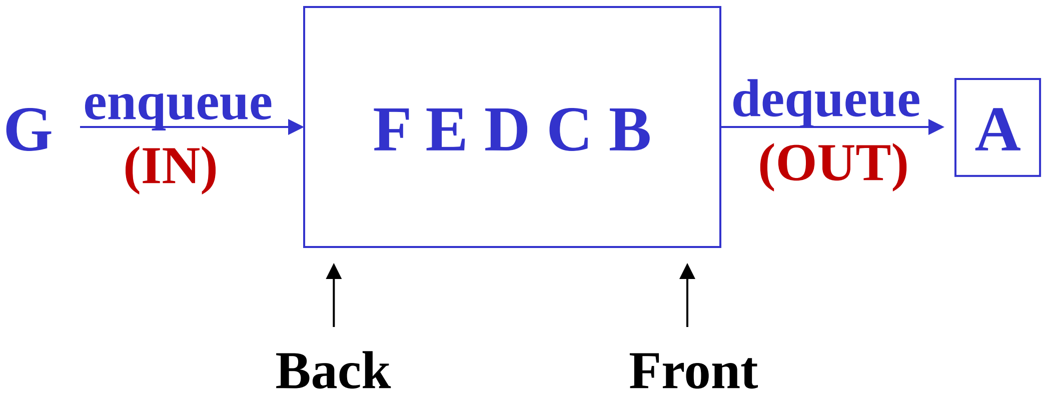
  `enqueue`

  `dequeue`

  `is_empty`

`What else?`

  front



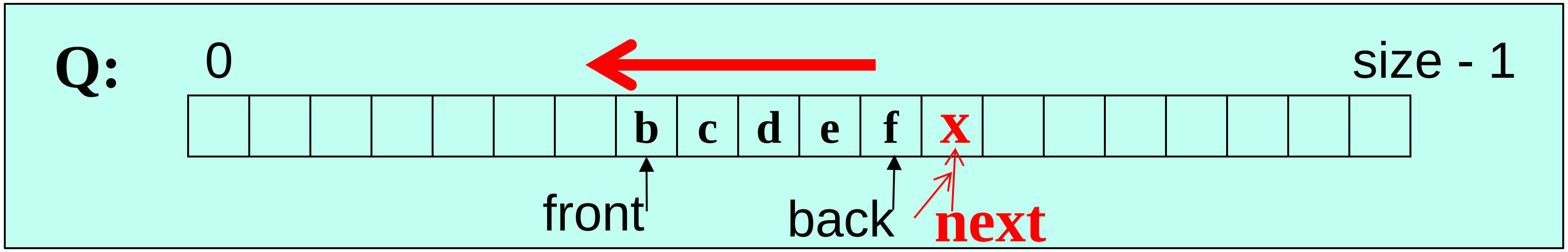
- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)

Stacks vs. Queues

Stack

Queue

Circular Array Queue Data Structure

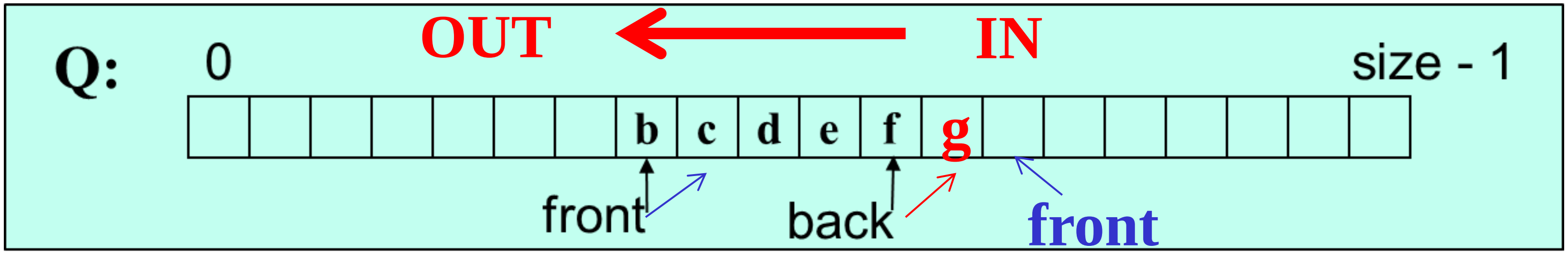



```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

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

- What if ***queue*** is empty?
  - Enqueue?
    yes
  - Dequeue?
    no
- What if ***array*** is full?
  - Enqueue?
    no
  - Dequeue
    yes

Circular Array Example *(text p 94 has another one)*



enqueue('g')

o1 = dequeue( )
b
o2 = dequeue( )
c
o3 = dequeue( )
d

o4 = dequeue( )
e
o5 = dequeue( )
f
o6 = dequeue( )
g

Now where are back and front?

Now front = back+1!

Empty Queue

- Will front = back + 1 always be true for an empty queue?

back front

-1 0

0 0

‘a’	

4 0

‘a’	‘b’	‘c’	‘d’	‘e’

back front

4 4

‘a’	‘b’	‘c’	‘d’	‘e’

4 0

‘a’	‘b’	‘c’	‘d’	‘e’

front = (back + 1) % arraysize

0 = 5 % 5

Circular Queue

- When we add an ‘f’ to the queue that has only the ‘e’, back will go around to position zero. back=(4+1)%5

back	front
-1	0

back	front
0	0

‘a’				

back	front
4	0

‘a’	‘b’	‘c’	‘d’	‘e’

back	front
4	4

‘a’	‘b’	‘c’	‘d’	‘e’

back	front
0	4

‘f’	‘b’	‘c’	‘d’	‘e’

Complexity of Circular Queue Operations

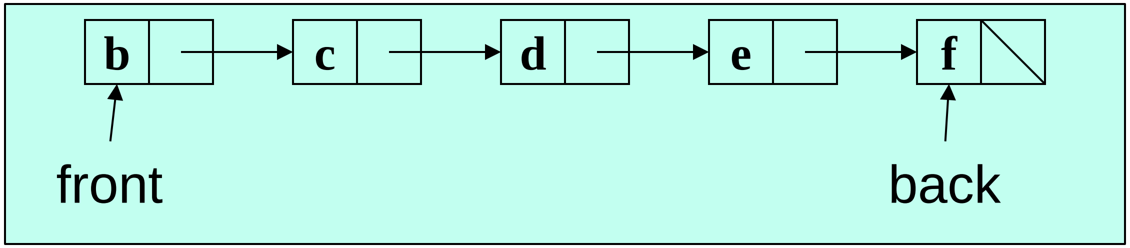
```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

O(1)
constant

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

O(1)
constant

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? **yes**
  - Dequeue? **no**
- Can ***list*** be full? **no**
- How to *test* for empty?
- front=back=null
- What is the *complexity* of the operations?
- O(1)

Circular Array vs. Linked List for Queues

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

Conclusion

- Abstract data structures allow us to define a new data type and its operations.
- Each abstraction will have one or more implementations.
- Which implementation to use depends on the application, the expected operations, the memory and time requirements.
- Both stacks and queues have array and linked implementations.
- We'll look at other ordered-queue implementations later.