CSE373: Data Structures and Algorithms

Introduction; ADTs; Stacks/Queues

Steve Tanimoto
Autumn 2016

This lecture material represents the work of multiple instructors at the University of Washington, most recently, Prof. L. Shaprio. Thank you to all who have contributed!

Registration

- We have 175 students registered and more waiting!
- If you're thinking of dropping the course please decide *soon*!

Waitlisted students

- If you don't absolutely have to take the course this quarter, it's unlikely you'll get in.
- If you think you absolutely have to take the course this quarter, fill out the course overload application online at **http://tinyurl.com/zlarys2**
- Make a note of the code that I give out in class. Do not make this public or share it with those who have not attended class today. The CSE undergraduate advisors will decide who gets added to the course.

I will not make individual decisions about registration!

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:
- Adjust class email-list and Catalyst GoPost settings
- Read all course policies
- Set up your Java environment for Assignment 1
- Answer the background survey questions. (Participation credit is available on this through Friday only.)
- Read Chapters 3.1 (lists), 3.6 (stacks) and 3.7 (queues) of the Weiss book; it's relevant to Assignment 1, due next week
- Bookmark our course web page.

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

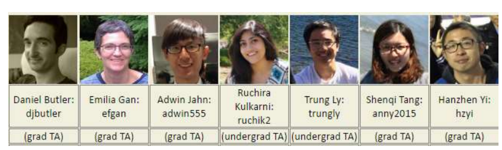
Course instructor

Steve Tanimoto
UW CSE faculty member. My research is on the design of tools to support collaborative problem solving. My interests also include livecoding, visual programming, image processing, AI, and computers in music and education.

My early research was on data structures and algorithms applied to graphical information and images (e.g., pyramids, octrees, region-adjacency graphs, transforms).

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

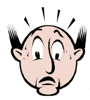
Teaching Assistants



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

Communication

- Course email list: `cse373b_au16@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
- Instructor feedback link
  - For good and bad: but please be gentle.



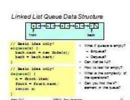
Course meetings

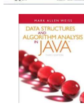
- Lecture (Steve)
  - Materials posted, but take notes
  - Ask questions, focus on key ideas (rarely coding details)
- Sections on Thursdays
  - Tentative agenda available on the calendar
  - Help on programming/tool background
  - Example problems
  - Occasional quizzes
- Office hours
  - Use them: *please visit me*
  - Ideally not *just* for homework questions (but that's great too)

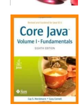
Roles of Java and Pseudocode

- Java: Programming assignments. A few lecture illustrations.
- Pseudocode : Lecture examples of algorithm descriptions. Quizzes and exams.

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: Mark Allen Weiss: Data Structures and Algorithms in Java, 3rd ed. Online readings will be used to supplement lecture material.
- 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.

Computing Facilities

- College of Arts & Sciences Instructional Computing Lab
  - http://depts.washington.edu/aslab/
  - Or your own machine
- We'll use Java 8 for the programming assignments.
- Eclipse (Neon release) is our recommended programming environment

Coursework and Assessment

- 6 Assignments (45%)
  - Most involve programming, but some have written questions
  - Higher-level concepts than "just code it up"
  - First programming assignment due Friday, October 7.
- Participation (15%)
  - Worksheets
  - Questionnaires
  - Quizzes
  - Section activities.
- Midterm Monday November 7, in class (15%)
- Final exam: Tuesday December 13, 2:30-4:20PM (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
- The CSE department and I take academic integrity extremely seriously
  - I offer great trust but with little sympathy for violations
  - Honest work is a vital feature of a university
- 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.

Advice on how to succeed in 373

- Get to class on time!
  - I will start and end promptly.
  - First 2 minutes are *much* more important than last 2!
- Learn this stuff
  - It is at the absolute core of computing and software.
  - Falling behind only makes more work for you.
- Do the work and try hard.
- This stuff is powerful and fascinating, so have fun with it!

Today in Class

- Course mechanics: Did I forget anything?
- What this course is about
- Start *abstract data types* (ADTs), *stacks*, and *queues*
  - Largely review

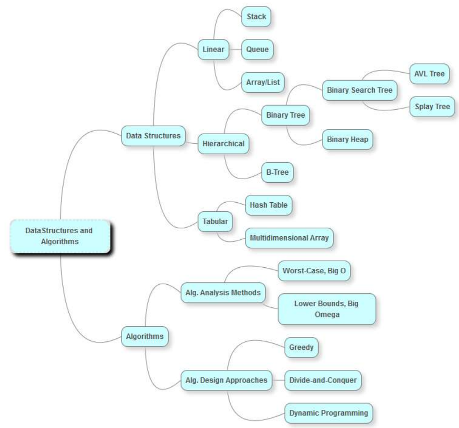
Let's see some art …

Data structures and the architecture of buildings are analogous.

What this course will cover

- Introduction to Algorithm Analysis
- Lists, Stacks, Queues
- Trees, Hashing, Dictionaries
- Heaps, Priority Queues
- Sorting
- Disjoint Sets
- Graph Algorithms
- Algorithm Paradigms and NP-Completeness
- Introduction to Parallelism and Concurrency (Time Permitting)

Let's see some more art…



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

Goals

- 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 mix 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

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

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

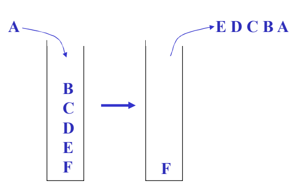
- Abstract Data Type (ADT)
  - Mathematical description of some possible groups of data items, with a set of operations on these groups.
  - Not concerned with implementation details
- Algorithm
  - A high level, language-independent description of a step-by-step process for working with information
- 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 programming 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`**

The Stack ADT

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



Can be implemented with an array or a linked list
- Like queues, type of elements is irrelevant

Why useful

The Stack ADT is a useful abstraction because:
- It arises frequently in programming
  - 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…”

The Queue ADT

- Operations
  `create`
  `destroy`
  `enqueue`
  `dequeue`
  `is_empty`

G enqueue → F E D C B dequeue → A

Back Front

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

Stacks vs. Queues

Stack

Queue

Circular Array Queue Data Structure

Q: 0 size - 1

| b | c | d | e | f | x |

front back next

```
// 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)

Q: 0 OUT ← IN size - 1

| b | c | d | e | f | g |

front back front

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
[ ]

0 0
['a' |]

4 0
['a' | 'b' | 'c' | 'd' | 'e']

back front
4 4
['a' | 'b' | 'c' | 'd' | 'e']

0 4
['f' | 'b' | 'c' | 'd' | 'e']

Complexity of Circular Queue Operations

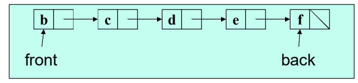
```
// Basic idea only!
enqueue(x) {
  next = (back + 1) % size
  Q[next] = x;
  back = next
}
```

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

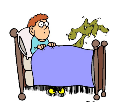
Array:
- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast
- Constant-time access to k^{th} element

- 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.