Lecture 1: Welcome!

CSE 373: Data Structures and Algorithms

Agenda

- Introductions
- Syllabus
- Dust off data structure cob webs
- Meet the ADT
- What is “complexity”?

Waitlist/ Overloads

- There are no overloads
- I have no control over these things :/
- Email cse373@cs.washington.edu for all registration questions
- Many students move around, likely a spot will open
- Keep coming to lecture!

Hello!

I am Kasey Champion

Software Engineer @ Karat

High School Teacher @ Franklin High

champk@cs.washington.edu

Office in CSE 218

Office Hours: Wednesdays 9:30-11:30, Fridays 2:30-4:30

@techie4good

Class Style

Kasey has to go to her “real job” after this

- The internets
- Your TAs
- Each other

Please come to lecture (yes, there will be panoptos)

- Warm Ups -> Extra Credit
- Collaboration
- Demos
- Ask questions! Point out mistakes!

Sections

- TAs = heroes
- Exam Practice problems
- Sections start this week

Course Administration

Course Page

- All course content/announcements posted here
- Pay attention for updates!

Canvas

- Grades will be posted here

Office Hours

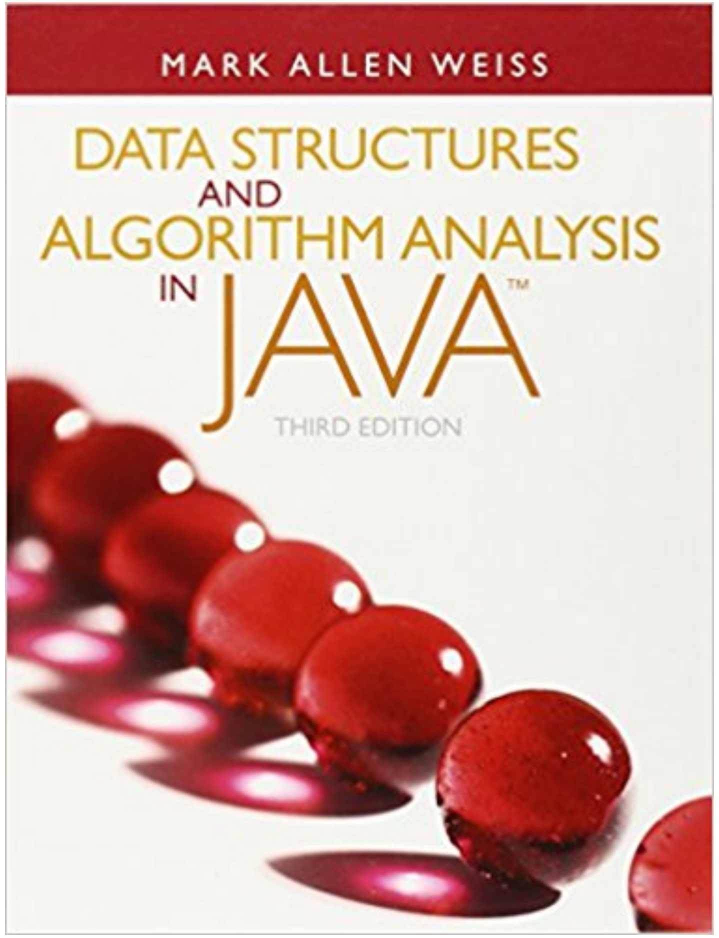
- Will be posted on Course Page
- Will start next week

Piazza

- Great place to discuss questions with other students
- Will be monitored by course staff
- No posting of project code!

Textbook

- Optional
- Data Structures and Algorithm Analysis in Java by Mark Allen Weiss



Grade Break Down

Homework (55%)

- 4 Partner Projects (40%)
  - Partners **GREATLY** encouraged
  - Graded automatically
  - Regrades available on some parts
- 3 Individual Assignments (15%)
  - Must be individual
  - Graded by TAs

Exams (45%)

- Midterm Exam – Friday May 4th in class (20%)
- Final Exam – Tuesday June 11th 8:30-10:30 here! (25%)

Syllabus

Homework Policies

- 3 late days
  - Both partners must use one
  - When you run out you will forfeit 20% each 24 hour period an assignment is late
  - No assignment will be accepted more than 2 days late

Project Regrades

- Get back half your missed points for part 1 when you turn in part 2
- Fill out form if you think your grade is incorrect

Exams

- Allowed 8.5”x11” note page
- NO MAKE UPS!
  - Let Kasey know ASAP if you cannot attend an exam

Academic Integrity

- No posting code on discussion board or ANYWHERE online
- We do run MOSS
- No directly sharing code with one another (except for partners)

Extra Credit

- Available for attending lecture
- Worth up to 0.05 GPA bump

Questions?

Clarification on syllabus, General complaining/moaning

What is this class about?

CSE 143 – OBJECT ORIENTED PROGRAMMING

- Classes and Interfaces
- Methods, variables and conditionals
- Loops and recursion
- Linked lists and binary trees
- Sorting and Searching
- O(n) analysis
- Generics

CSE 373 – DATA STRUCTURES AND ALGORITHMS

- Design decisions
- Design analysis
- Implementations of data structures
- Debugging and testing
- Abstract Data Types
- Code Modeling
- Complexity Analysis
- Software Engineering Practices

Data Structures and Algorithms

What are they anyway?

Basic Definitions

Data Structure

- A way of organizing and storing related data points
- Examples from CSE 14X: arrays, linked lists, stacks, queues, trees

Algorithm

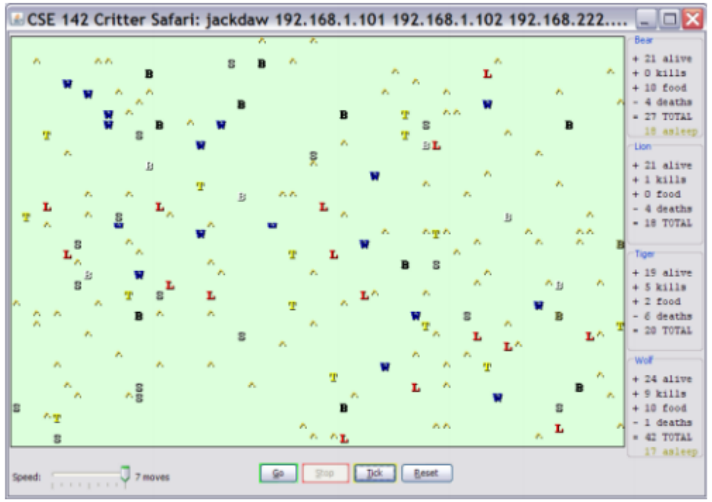
- A series of precise instructions used to perform a task
- Examples from CSE 14X: binary search, merge sort, recursive backtracking

Review: Clients vs Objects

CLIENT CLASSES

A class that is executable, in Java this means it contains a Main method

```
public static void main(String[] args)
```



OBJECT CLASSES

A coded structure that contains data and behavior

Start with the data you want to hold, organize the things you want to enable users to do with that data

1. Ant

constructor	`public Ant(boolean walkSouth)`
color	red
eating behavior	always returns `true`
fighting behavior	always scratch
movement	if the Ant was constructed with a `walkSouth` value of `true`, then alternates between south and east in a zigzag (S, E, S, E, ...); otherwise, if the Ant was constructed with a `walkSouth` value of `false`, then alternates between north and east in a zigzag (N, E, N, E, ...)
toString	`"%"` (percent)

Abstract Data Types (ADT)

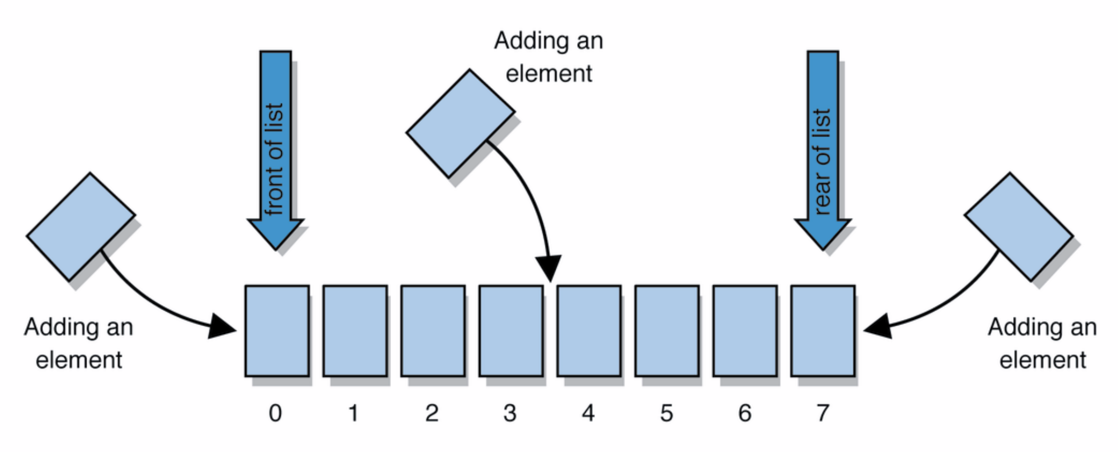
Abstract Data types

- A definition for expected operations and behavior

Start with the operations you want to do then define how those operations will play out on whatever data is being stored

Review: List - a collection storing an ordered sequence of elements

- each element is accessible by a 0-based index
- a list has a size (number of elements that have been added)
- elements can be added to the front, back, or elsewhere
- in Java, a list can be represented as an ArrayList object



Review: Interfaces

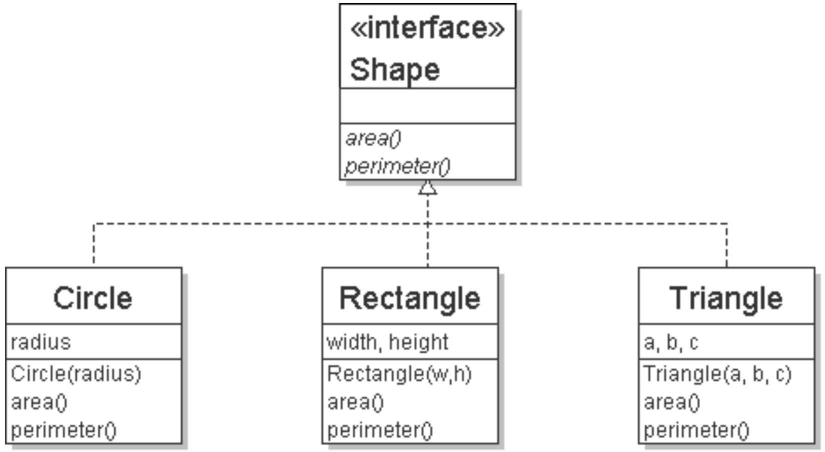
interface: A list of methods that a class promises to implement.

- Interfaces give you an is-a relationship *without* code sharing.
  - A `Rectangle` object can be treated as a `Shape` but inherits no code.
- Analogous to non-programming idea of roles or certifications:
  - "I'm certified as a CPA accountant.
    This assures you I know how to do taxes, audits, and consulting."
  - "I'm 'certified' as a Shape, because I implement the Shape interface.
    This assures you I know how to compute my area and perimeter."

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

Example

```
// Describes features common to all
// shapes.
public interface Shape {
        public double area();
        public double perimeter();
}
```



Review: Java Collections

Java provides some implementations of ADTs for you!

You used:

Lists `List<Integer> a = new ArrayList<Integer>();`

Stacks `Stack<Character> c = new Stack<Character>();`

Queues `Queue<String> b = new LinkedList<String>();`

Maps `Map<String, String> d = new TreeMap<String, String>();`

But some data structures you made from scratch… why?

Linked Lists - LinkedIntList was a collection of ListNode

Binary Search Trees – SearchTree was a collection of SearchTreeNodes

Full Definitions

Abstract Data Type (ADT)

- *A definition for expected operations and behavior*
- A mathematical description of a collection with a set of supported operations and how they should behave when called upon
- Describes what a collection does, not how it does it
- Can be expressed as an interface
- Examples: List, Map, Set

Data Structure

- *A way of organizing and storing related data points*
- An object that implements the functionality of a specified ADT
- Describes exactly how the collection will perform the required operations
- Examples: LinkedIntList, ArrayIntList

ADTs we'll discuss this quarter

- List
- Set
- Map
- Stack
- Queue
- Priority Queue
- Graph
- Disjoint Set

Case Study: The List ADT

list: stores an ordered sequence of information.

- Each item is accessible by an index.
- Lists have a variable size as items can be added and remove

List ADT
state Set of ordered items Count of items **behavior** get(index) return item at index set(item, index) replace item at index append(item) add item to end of list insert(item, index) add item at index delete(index) delete item at index size() count of items

supported operations:

- **get(index):** returns the item at the given index
- **set(value, index):** sets the item at the given index to the given value
- **append(value):** adds the given item to the end of the list
- **insert(value, index):** insert the given item at the given index maintaining order
- **delete(index):** removes the item at the given index maintaining order
- **size():** returns the number of elements in the list

Case Study: List Implementations

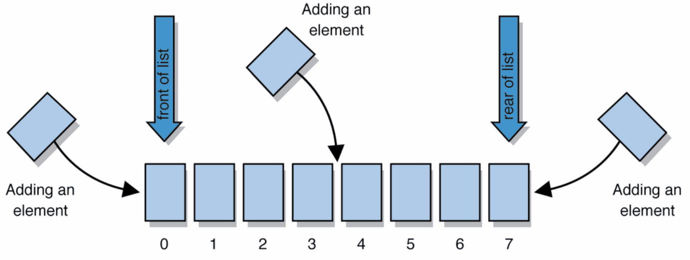
List ADT

state

Set of ordered items
Count of items

behavior

<u>get(index)</u> return item at index
<u>set(item, index)</u> replace item at index
<u>append(item)</u> add item to end of list
<u>insert(item, index)</u> add item at index
<u>delete(index)</u> delete item at index
<u>size()</u> count of items



ArrayList
uses an Array as underlying storage

ArrayList<E>

state

```
data[]
size
```

behavior

```
get return data[index]
set data[index] = value
append data[size] =
value, if out of space
grow data
insert shift values to
make hole at index,
data[index] = value, if
out of space grow data
delete shift following
values forward
size return size
```

0	1	2	3	4
88.6	26.1	94.4	0	0

list   free space

LinkedList
uses nodes as underlying storage

LinkedList<E>

state

```
Node front
size
```

behavior

```
get loop until index,
return node's value
set loop until index,
update node's value
append create new node,
update next of last node
insert create new node,
loop until index, update
next fields
delete loop until index,
skip node
size return size
```

Implementing ArrayList

ArrayList<E>

state

data[]
size

behavior

get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size

`insert(element, index)` with shifting

`insert(10, 0)`

0	1	2	3
10	4	5	

`numberOfItems = 4`

`delete(index)` with shifting

`delete(0)`

0	1	2	3
10	3	4	5

`numberOfItems = 3`

Implementing ArrayList

ArrayList<E>

state
data[]
size

behavior
get return data[index]
set data[index] = value
append data[size] = value, if out of space grow data
insert shift values to make hole at index, data[index] = value, if out of space grow data
delete shift following values forward
size return size

`append(element)` with growth

`append(2)`

0	1	2	3
10	3	4	5

`numberOfItems = ` 5

0	1	2	3	4	5	6	7
				2			

Design Decisions

For every ADT there are lots of different ways to implement them

Based on your situation you should consider:

- Memory vs Speed
- Generic/Reusability vs Specific/Specialized
- One Function vs Another
- Robustness vs Performance

This class is all about implementing ADTs based on making the right design tradeoffs!

> A common topic in interview questions