Lecture 1: Welcome!

CSE 373: Data Structures and Algorithms
Agenda

- Introductions
- Syllabus
- Dust off cob webs
- Meet the ADT
Course Staff

Instructor: Robbie Weber
rtweber2@cs.washington.edu
Office: CSE 330
Office Hours: M,W 2:20-3:30 and by appointment

Ph.D. student in CSE
Research in algorithm design
2nd time as instructor
TAed 15 times
(3 times for this course)

Amazing Teaching Assistants:

Brian Chan  Oscar Sprumont  Blarry Wang
Zach Chun    Matthew Taing     Xin Yang
Kevin Son    Howard Xiao       Velocity Yu
Class Style

Please come to lecture (yes, there will be panoptos)
- Warm Ups -> Extra Credit
- Discuss with other students
- Ask questions! Point out mistakes!

Sections
- TAs = heroes
- Practice problems
- Different Explanations
- Sections start this week
Class Style

Summer Quarter is weird.
We have fewer class meetings
But they’re all 60 minutes, not 50 minutes.

We’ve rearranged (and cut) things to account for the schedule
But things are more cramped.

Please start on assignments early and ask for help early
TAs are amazing – we can help A LOT 3 days before an assignment is due.
There’s only so much we can do 3 hours before it’s due.
Course Administration

Course Page [cs.washington.edu/373](http://cs.washington.edu/373)
- Course content posted here
- Pay attention for updates!

Canvas
- Grades will be posted to Canvas at end of quarter

Office Hours
- Will be posted on Course Page
- Will start next week (at the latest)

Piazza
- If you were signed up for the course before this morning, you were added already.
- If not, you can find an add code on Canvas (or ask a member of staff)
- Announcements made via 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%)
- Programming Projects (35%)
- 1 Individual Project (review of 14X)
- 4 Partner Projects
  - Partners GREATLY encouraged
  - Graded automatically
  - Regrades available on some parts
  - Some projects are two connected one-week assignments; others are just one week.
- Written Assignments (20%)
  - Written assignments graded by TAs

Exams (45%)
- Midterm Exam – Friday July 26th in class (15%)
- Final Exam (30%)
  - Two one-hour parts:
    - In place of final section Thursday August 22
    - In place of final lecture Friday August 23

If you have a conflict with any of those dates, email Robbie as soon as possible.
Homework Policies
- 4 late days
  - Both partners must use one for pair projects
  - 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

Regrades
- For assignments with two parts: automatically get back half missed points for part 1 when you turn in part 2
- If you think we made a mistake grading (or the autograder did something unreasonable):
  - Fill out a regrade request on gradescope for written work
  - Fill out the google form for programming projects

Academic Integrity
- Discussing concepts and high-level ideas for problems is strongly encouraged, but submitted work must be your own.
- Read policy on the webpage.
- 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, by filling out PollEverywhere questions
- Based on completion not correctness.
- Worth up to 0.05 GPA bump
Questions?

Clarification on syllabus, General complaining/moaning
What is this class about?

DATA STRUCTURES

How do we organize our data most effectively?
And how do we justify those decisions
Actually implement data structures
Not just use them

ALGORITHMS

Some classic, fundamental algorithms
How they work
How do we get them use them to solve problems

TOOLS TO MAKE AND ANALYZE DS&ALGS

Basic Software Engineering Practices
Debugging
Testing
Version Control

Basic Theoretical Computer Science
Modelling Code/Big-O
Analyzing Recurrences
Data Structures and Algorithms

What are they anyway?
Basic Definitions

Data Structure
- A way of organizing, storing, accessing, and updating a set of data
- Examples from CSE 14X: arrays, linked lists, binary search trees

Algorithm
- A series of precise instructions guaranteed to produce a certain answer
- Examples from CSE 14X: binary search, merge sort
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."

```java
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**

```java
// 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>();`
  ```
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 removed

**List ADT**

<table>
<thead>
<tr>
<th>state</th>
<th>behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of ordered items</td>
<td>get(index) return item at index</td>
</tr>
<tr>
<td>Count of items</td>
<td>set(item, index) replace item at index</td>
</tr>
<tr>
<td></td>
<td>append(item) add item to end of list</td>
</tr>
<tr>
<td></td>
<td>insert(item, index) add item at index</td>
</tr>
<tr>
<td></td>
<td>delete(index) delete item at index</td>
</tr>
<tr>
<td></td>
<td>size() count of items</td>
</tr>
</tbody>
</table>

**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
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

ArrayList
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

LinkedList
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

ArrayList
uses an Array as underlying storage

LinkedList
uses nodes as underlying storage

Table:
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>88.6</td>
<td>26.1</td>
<td>94.4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

ArrayList
uses an Array as underlying storage

LinkedList
uses nodes as underlying storage

Node front
size

Linked List
<table>
<thead>
<tr>
<th>88.6</th>
<th>26.1</th>
<th>94.4</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>free space</td>
<td></td>
</tr>
</tbody>
</table>
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
insert  shift following
values forward
data[size] = return

size  return numberOfItems

insert(element, index) with shifting

insert(10, 0)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

numberOfItems = 4

delete(index) with shifting

delete(0)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

numberOfItems = 3

Take 2 Minutes

Should we overwrite index 3 with null?
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

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

numberOfItems = 5

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
Case Study: List Implementations

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

**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

ArrayList uses an Array as underlying storage

```java
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
```

Linked List uses nodes as underlying storage

```java
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
```

*Take 2 Minutes*

What method will be much faster for LinkedList than for ArrayList?

**ArrayList**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>88.6</td>
<td>26.1</td>
<td>94.4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**LinkedList**

<table>
<thead>
<tr>
<th>88.6</th>
<th>26.1</th>
<th>94.4</th>
</tr>
</thead>
</table>

list free space
Design Decisions

Dub Street Burgers is implementing a new system for ticket (i.e. food order) management. When a new ticket comes in, it is placed at the end of the set of tickets. Food is prepared in approximately the same order it was requested, but sometimes tickets are fulfilled out of order.

Let’s represent tickets as a list. Which of our ADT implementations should we use? Why?
Design Decisions

Let’s represent tickets as a list. Which of our ADT implementations should we use?

Why?

ArrayList

Creating a new ticket is very fast (as long as we don’t resize), and I want the cooks to be able to see all the orders right away.

LinkedList

We’ll mostly be removing from the front of the list, which is much faster for the linkedlist (no shifting), and I want finished orders to be removed fast so they aren’t distracting.
Design Decisions

Both ArrayList and LinkedList implementations have pros and cons. Neither is strictly better than the other.

Some major objectives of this course:

**Evaluating** pros and cons

**Deciding** on a design

**Defending** that design decision

Especially when there’s more than one possible answer.