Before we start

Fill out the 373 Start-of-Quarter survey! Max 3 min:

tinyurl.com/20SuStart
Announcements

• Project 0 released!
  - Instructions on course website (calendar or under projects)
  - Due **Wednesday July 1 before 11:59pm PST**
  - Goals
    - Refresh 143 concepts
    - Set up IntelliJ and tools we’ll use in this class (Java, GitLab, Git, Checkstyle)
    - Learn about JUnit and unit testing

• Sections start tomorrow!
  - Canvas events will be created today

• Missing permissions?
  - We’ll post a form to fill out
Announcements

• Office Hours
  - This quarter, we’ll run queue via Discord
    - A popular instant messaging + voice/video chat service
    - All-in-one location for OH queue, community building, getting help from peers

#oh-queue

@TA On Duty quick question about the definition of an ADT @dubs

Sure! Let’s all discuss in this Zoom meeting:

ping all TAs currently “on duty” in OH

briefly summarize your question

@ your project partner or anyone else you’re working with
Why Discord?

• Build community!
  - Survey results: COVID has made us distant 😞
  - Social channels for hanging out, meeting people, finding partners
    - These are your space! Not managed by course staff

• Seamless Queueing
  - Students reported Zoom waiting rooms and spreadsheets were clunky
  - Hang around, get a notification when a TA is ready

• Be In the Room Where It Happens
  - Chat while you’re waiting!
    - Public channels: ask if anyone has a similar issue, see who else is on the queue and reach out
Using Discord

• Two ways to participate:

1. Create Discord Account
   • Enter your email
   • Stay logged in for the quarter
   • Easier to meet people and build community

2. Join Anonymously
   • Temporary display name, no other info
   • Account disappears when you close window
   • Use Discord as simple, anonymous queue service; get helped over Zoom

• Discord is a 3rd-Party App
  - You do NOT need to enter any personal information to participate in OH
  - But you are welcome to make an account or use an existing one
  - Have fun, but be respectful and welcoming
Announcements

• Office Hours
  - Discord server invite will be posted later today
  - Office Hours will start this Friday, June 26th
  - Note: TA continually monitor Piazza, only monitor Discord during OH

• Instructor meeting link added to Staff Page
  - Schedule a 1:1 for anything! Course concerns, taking these concepts beyond 373, interviews/job advice, meaning of life, etc.

• Survey results: Anxious about 143 material?
  - Don’t worry! 😊 P0 is all about helping you get back up to speed!
  - We’ll publish an additional review guide today
Lecture Outline

• Runtime Analysis
• The List ADT
• Design Decisions
Learning Objectives
After this lecture, you should be able to...

• *(143 Review)* Determine whether simple code belongs to the constant, linear, or quadratic complexity classes

• Distinguish the List ADT from ArrayList and LinkedList implementations

• Compare the runtime of certain operations on ArrayList and LinkedList, based on how they’re implemented

• Describe the process of making design decisions
Runtime Analysis

• What does it mean for a data structure to be "slow" or "fast"?

• We could just run and measure the (wallclock) time!
  - Why won’t that work?
    - Different hardware could affect speed
    - What other programs are running?
    - Speed affected by the input given

• Our general approach:
  - Count how many “steps” a program takes to execute on an input of size N
143 Review “Big Oh”

• **Efficiency**: measure of computing resources used by code
  - Could be time (most common), space/memory taken up, etc.

• We measure runtime in proportion to the input data size, N
  - **Growth Rate**: change in runtime as N gets bigger

• Assume:
  - Every Java statement takes the same amount of time to run
  - Method call runtime: total of statements in its body
  - Loop runtime: (number of repetitions) x (total of its body)
143 Review  “Big Oh”

```
int a = b + 1;
for (int i = 0; i < N; i++)
    data[i] = a;
```

- Runs 1 statement
  - Constant

```
for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
        data1[i] = a;
        data2[i] = b;
        data3[i] = c;
```

- Runs N statements
  - Linear
  - Runs 3N^2 statements
  - Quadratic

- We ignore constants like 3 because they are tiny next to N or N^2
- We say that this algorithm runs "on the order of" N^2
- or \( O(N^2) \) for short ("Big-Oh of N squared")
143 Review Complexity Class

- **Complexity Class**: a category of algorithm efficiency based on the algorithm’s relationship to the input size $N$

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Big-O</th>
<th>Runtime if you double $N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>$O(1)$</td>
<td>unchanged</td>
</tr>
<tr>
<td>logarithmic</td>
<td>$O(\log_2 N)$</td>
<td>increases slightly</td>
</tr>
<tr>
<td>linear</td>
<td>$O(N)$</td>
<td>doubles</td>
</tr>
<tr>
<td>log-linear</td>
<td>$O(N \log_2 N)$</td>
<td>slightly more than doubles</td>
</tr>
<tr>
<td>quadratic</td>
<td>$O(N^2)$</td>
<td>quadruples</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
</tr>
</tbody>
</table>
Lecture Outline

• Runtime Analysis

• The List ADT

• Design Decisions
Review  ADTs: Abstract Data Types

• An **abstract data type** is a data type that does not specify any one implementation.
  - Think of this as an **agreement**: about what is provided, but not how.

• **Data structures** implement ADTs.
  • **Resizable array** can implement List, Stack, Queue, Deque, PQ, etc.
  • **Linked nodes** can implement List, Stack, Queue, Deque, PQ, etc.
Case Study: The List ADT

**List**: a collection storing an ordered sequence of elements.
- Each item is accessible by an index.
- A list has a variable size defined as the number of elements in the list
- Elements can be added to or removed from any position in the list

Relation to code/mental image of a list:

```java
List<String> names = new ArrayList<>(); // []
names.size(); // evaluates to 0
names.add("Timothy"); // [“Timothy”]
names.add("Siddharth"); // [“Timothy, Siddharth”]
names.insert("Leona", 0); // [“Leona”, “Timothy”, “Siddharth”]
names.size(); // evaluates to 3
```
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
  - `add(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<E>**

- **State**
  - `data[]`
  - `size`

- **Behavior**
  - `get` return `data[index]`
  - `set` `data[index] = value`
  - `add` `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<E>**

- **State**
  - `Node front;`
  - `size`

- **Behavior**
  - `get` loop until index, return node’s value
  - `set` loop until index, update node’s value
  - `add` 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`

```
[88.6, 26.1, 94.4]
```

![Diagram showing list implementation](image)
Case Study: Let’s **zoom** In On ArrayList

- How do Java / other programming languages implement ArrayList to achieve all the List behavior?

- On the inside:
  - stores the elements **inside an array** (which has a fixed capacity) that typically has more space than currently used (For example when there is only 1 element in the actual list, the array might have 10 spaces for data),
  - stores all of these elements at the front of the array and keeps track of how many **there are** (the size) so that the implementation doesn’t get confused enough to look at the empty space. This means that sometimes we will have to do a lot of work to shift the elements around.

```
List View
[“Leona”, “Timothy”, “Siddharth”]

ArrayList View
[“Leona”, “Timothy”, “Siddharth”, null, null, null]
```
Implementing ArrayList

ArrayList\<E\>

State
- data[]
- size

Behavior
- get return data[index]
- set data[index] = value
- add 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

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>d</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

insert(“d”, 0)

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>b</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

delete(index) with shifting

<table>
<thead>
<tr>
<th>index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>a</td>
<td>d</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

delete(0)
Should we overwrite index 3 with null?

Briefly explain why or why not.

**ArrayList&lt;E&gt;**

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[]</td>
<td>get return data[index]</td>
</tr>
<tr>
<td>size</td>
<td>set data[index] = value</td>
</tr>
<tr>
<td></td>
<td>add data[size] = value, if out of space grow data</td>
</tr>
<tr>
<td></td>
<td>insert shift values to make hole at index, data[index] = value, if out of space grow data</td>
</tr>
<tr>
<td></td>
<td>delete shift following values forward</td>
</tr>
<tr>
<td></td>
<td>size return size</td>
</tr>
</tbody>
</table>

**insert(element, index) with shifting**

```plaintext
insert("d", 0) [d, a, b, c] size = 4
```

**delete(index) with shifting**

```plaintext
delete(0) [a, b, c, c] size = 3
```

Briefly explain why or why not.
Implementing ArrayList

ArrayList<E>

State
data[]
size

Behavior
get return data[index]
set data[index] = value
add 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(2)

append(\text{element}) \text{ 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 = \boxed{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>
Which operations will be much faster for LinkedList than ArrayList? Briefly explain why.

### LinkedList<E>

**State**
- Node front;
- size

**Behavior**
- get loop until index, return node’s value
- set loop until index, update node’s value
- add 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<E>

**State**
- data[]
- size

**Behavior**
- get return data[index]
- set data[index] = value
- add 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

### LIST ADT

**State**
- Set of ordered items
- Count of items

**Behavior**
- get(index) return item at index
- set(item, index) replace item at index
- add(item) add item to end of list
- insert(item, index) add item at index
- delete(index) delete item at index
- size() count of items
Lecture Outline

• Runtime Analysis

• The List ADT

• Design Decisions
Design Decisions

• For every ADT, many ways to implement

• Based on your situation you should consider:
  - Speed vs Memory Usage
  - 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
Design Decisions

• Dub Street Burgers is implementing a new system to manage orders

• When an order comes in, it’s placed at the end of the set of orders

• Food is prepared in approximately the same order it was requested, but sometimes orders are fulfilled out of order

• Let’s represent tickets using the List ADT. **What implementation should we use? Why?**
What implementation should we use? Why?

• **ArrayList**
  - Creating a new order is very fast (as long as we don’t have to resize)
  - Cooks can see any given order easily

• **LinkedList**
  - Creating an order is slower (have to iterate through whole list)
  - We’ll mostly be removing from the front of the list, which is fast because it requires no shifting
Comparing ADT Implementations: List

<table>
<thead>
<tr>
<th></th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (front)</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>remove (front)</td>
<td>linear</td>
<td>constant</td>
</tr>
<tr>
<td>add (back)</td>
<td>(usually) constant</td>
<td>linear</td>
</tr>
<tr>
<td>remove (back)</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>get</td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td>put</td>
<td>linear</td>
<td>linear</td>
</tr>
</tbody>
</table>

- Important to be able to come up with this, and understand why
- But only half the story: to be able to make a design decision, need the context to understand which of these we should prioritize
Design Decisions

• Both ArrayList and LinkedList have pros and cons, neither is strictly better than the other

• The Design Decision process:
  - Evaluate pros and cons
  - Decide on a design
  - Defend your design decision

• This is a major objective of the course!