Welcome!

Welcome to CSE 332

You're here early!

Grab a handout from the front, or from the webpage: cs.uw.edu/332

You can find the slides there too (see the calendar)

Welcome to CSE 332

CSE 332 Spring 25
Lecture 1

Outline

Course Mechanics

Start of content

- Review of queues and stacks

Your TODO list

- Make sure you're on Ed.
- Get started on Exercise 0 (and update your IDE/java installs while you're at it).

Staff

Instructor: Robbie Weber

Ph.D. from UW CSE in theory
Fifth year as teaching faculty

Office: CSE2 311
Email: rtweber2@cs.washington.edu

TAs

Jacklyn Cui
Charles Hamilton-Eppler
Anthony He
Aaron Honjaya
Yafqa Khan
Alysa Meng
Cindy Ni
Juliette Park
Hana Smahi
Samarth Ramya Venkatesh
Iris Zhao
Rubee Zhao
Jolie Zhou

What's in this course?

Data Structures and Parallelism

Data structures and Algorithms (about 70% of the course)

- Starting to really think like a computer scientist.
  - Make design decisions, think about trade-offs.
- Core data structures and algorithms (timeless, classic material).
- Mathematically analyze those structures and algorithms.
- **Implement them**

Parallelism

- First serious treatment of programming with multiple threads

Goals

By the end of the quarter you will

- Understand the most common tools for storing and processing common data types.
- Consider tradeoffs between tools you could use.

So that you can

- Make good design decisions on your code
- Justify and communicate those decisions

Logistics

Textbook:

Weiss, Data Structures and Algorithm Analysis in Java

OPTIONAL (useful if you want more info, or an alternative presentation)

Ed (message board).

Gradescope.

You were invited to Ed and Gradescope already (if you were registered by last Friday)

Logistics – Exercises

Assigned throughout the quarter, about 15 in total.

We'll usually have two per week.

- Usually one due on Monday; one due on Friday (except at the end of the quarter)
- About half programming, half theory (at most one of each type per week)

Starting earlier is better (they can take some thought and/or some debugging).

We'll count the 12 highest scores, drop the others.

Logistics – Late Days

You have 6 late days to use during the quarter; a late day lets you submit 24 hours later than you would have otherwise.

You can use at most 2 late days per exercise.

Logistics – Exams

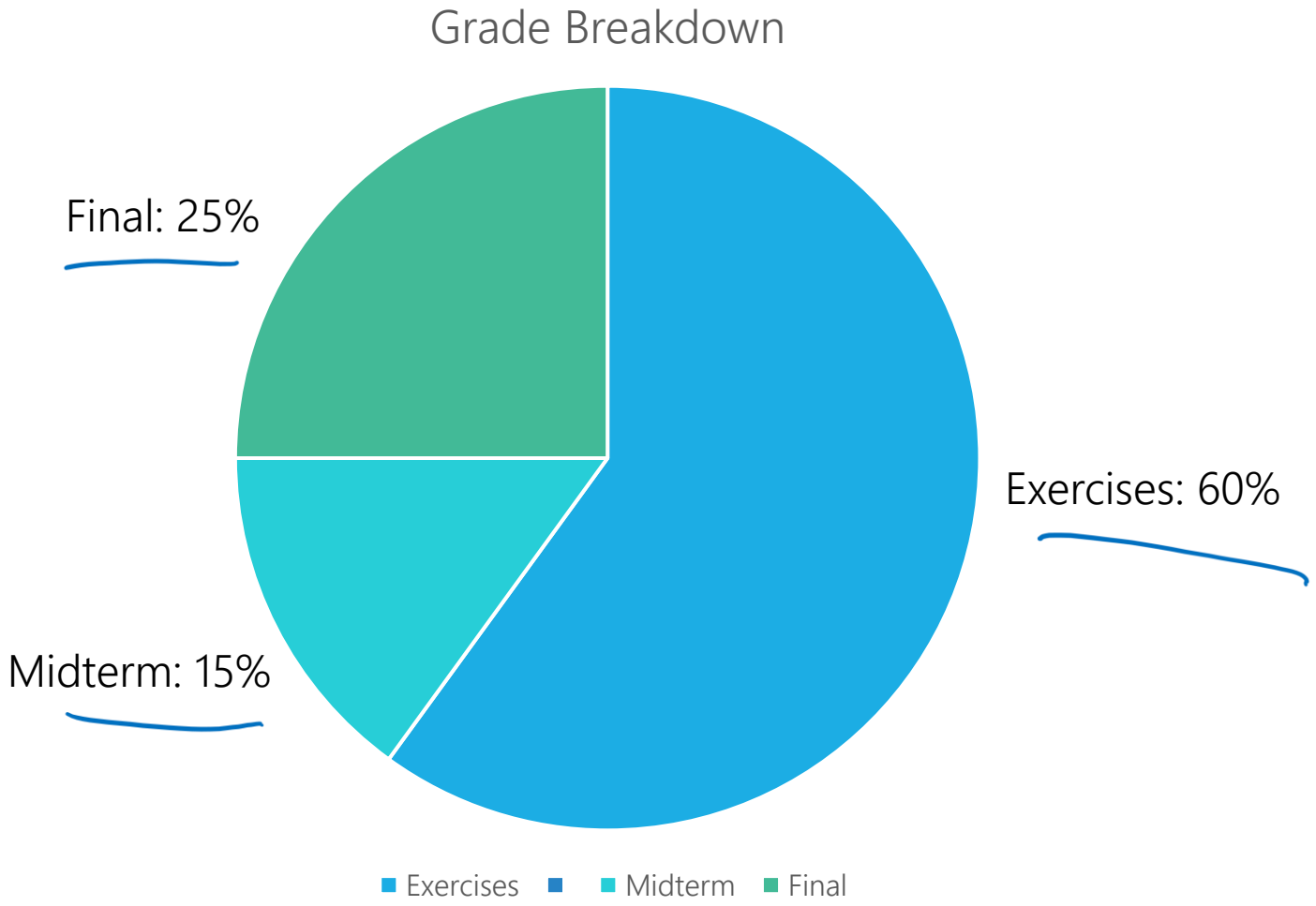
Midterm Wed Apr 30, 6-7:30 PM.

Final Thursday June 12th 12:30-2:20

Both are "combined" exams (same time regardless of which section you're in).

Please read the conflict policy in the syllabus if you might not be able to make those times.

Logistics – Work



Logistics – Section

Sections start this week

- Chance to practice problems about what we learned in lecture.
- Occasionally learn new material

We won't record sections.

Please strongly consider attending section in-person – the ability to discuss and ask questions in the smaller setting is hard to replace by just looking at the handout.

- But participation/attendance is **not** tracked

Academic Honesty

High level discussion is fine.

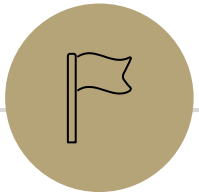
High level means you won't look at someone else's code, nor at their half-finished (or fully finished) writeup for the theory exercises

But you must:

- Not take any written notes away from your discussion.
- List everyone you collaborated with on your assignment.
- Take a 30-minute break between your discussion and writing/coding up your assignment.

Goal is for you to learn the material.

More details, including examples, on the webpage later this week.



Abstract Data Types

Abstract Data Types

An Abstract Data Type (ADT)

Mathematical description of a *thing* and a set of operations to interact with that *thing*.

Mathematical, think "like 311" --- precise enough you could do a proof on it (not necessarily "a bunch of numbers")

Thing think "organized data"

Abstract Data Type

An **Abstract Data Type (ADT)** is a mathematical definition of an object with operations to interact with that object.

Queue ADT

state

Set of elements

behavior

enqueue(element) – add a new element to the collection.

dequeue() – returns the element that has been in the collection the longest, and removes it.

peek() – find, but do not remove the element that has been in the collection the longest.

FIFO

Stack ADT

state

Set of elements

behavior

push(element) – add a new element to the collection.

pop() – returns the element that has been in the collection the shortest, and removes it.

peek() – find, but do not remove the element that has been in the collection the shortest.

LIFO

Why Define an ADT?

ADTs let us:

Identify patterns more easily: When you say "hey, I need an object that gives me back the thing I put in most recently" can remember what does that if it has a name.

Stacks show up in: the call stack (managing recursion), algorithms for handling context-free grammars, depth-first search (we'll see it later this quarter), and more…

Communicate more easily: Telling someone "keep track of this data on a stack" is easier than "make a linked list, and when you insert or remove update the head, not the tail, and keep track of the size, and…"

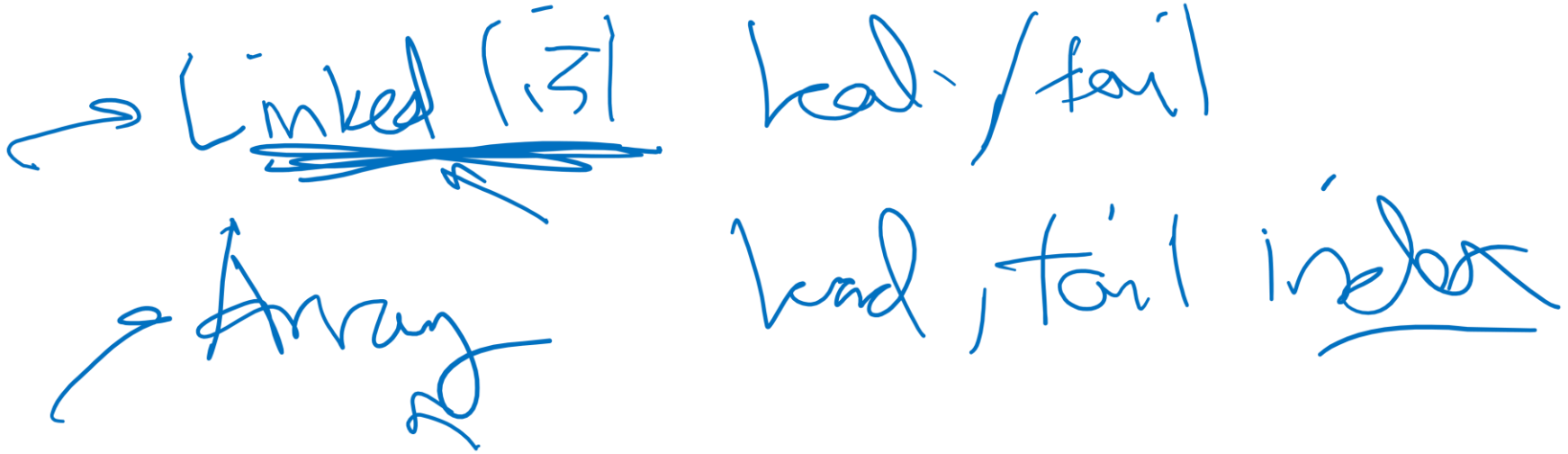
Reuse code!: Implement stacks once (or maybe a handful of versions) and reuse

What Data Structures for a Queue?

A data structure is a specific organization of data (and associated algorithms) to implement an abstract data type.

How would you implement a queue?

I.e., what data structure would you use?



Data Structure

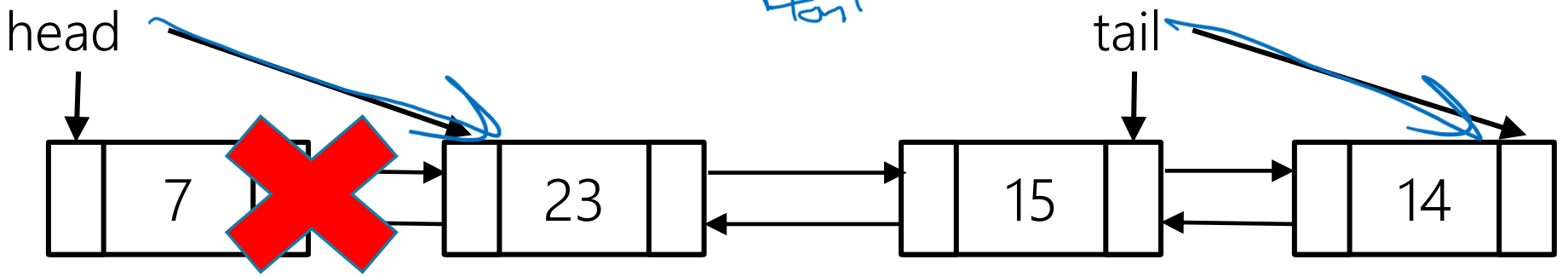
A clever way of organizing data points

- A data structure is an implementation of an ADT.

Ways to implement a queue

Array

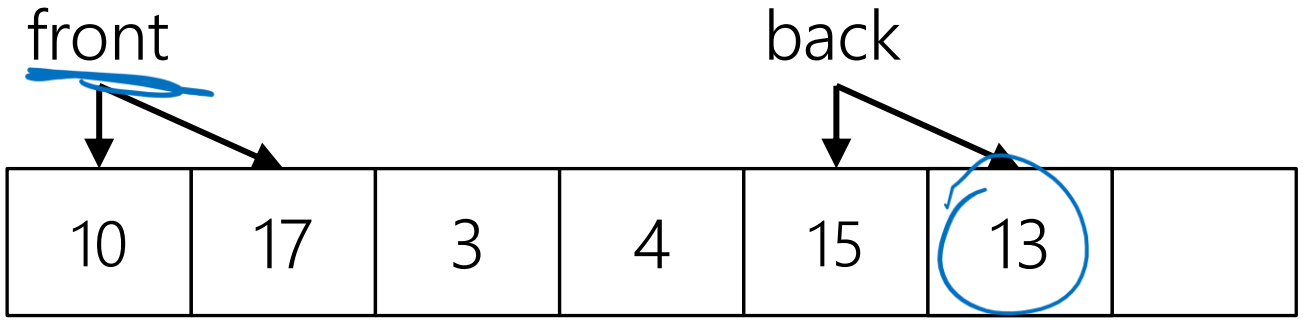
LinkedList



"Circular" Array

A different queue implementation

Removing elements is expensive. What if we just remember where the next element is?



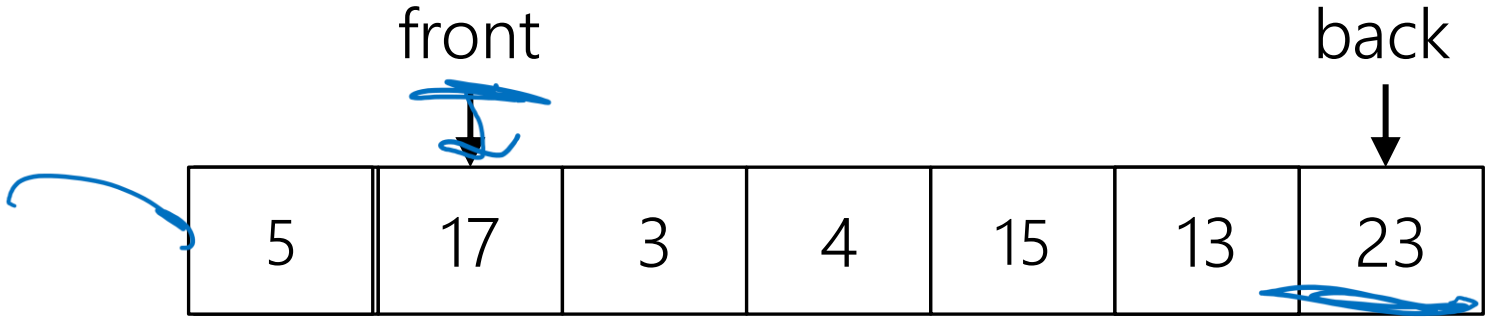

```
//sketch only
Enqueue(item) {
 Q[back]=item;
 back = (back+1) % size;
}
```

```
//sketch only
Dequeue() {
 item = Q[front];
 front = (front+1) % size;
 return item;
}
```

"Circular" Array

What about insertions?

We can "wrap around" (At least until the array is completely full.)



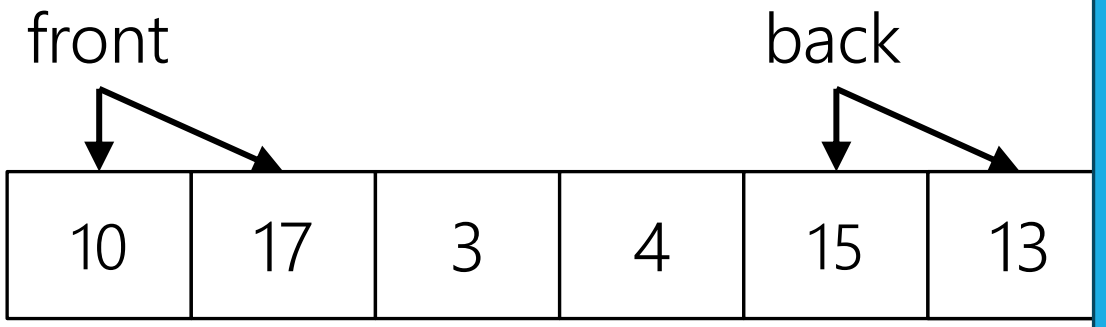

```
//sketch only
Enqueue(item) {
 Q[back]=item;
 back = (back+1) % size;
}
```

```
//sketch only
Dequeue() {
 item = Q[front];
 front = (front+1) % size;
 return item;
}
```

"Circular" Array

A different queue implementation

Removing elements is expensive. What if we just re
element is?



Sketches don't handle:

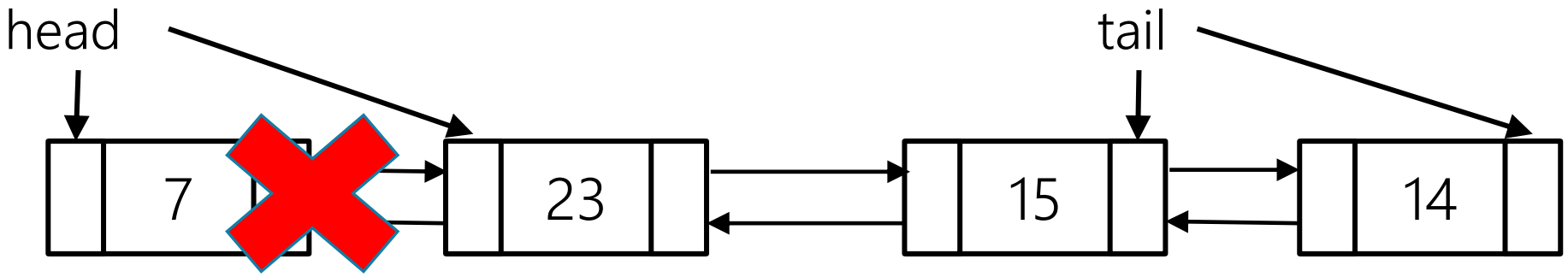
- What if queue is empty (for either enqueue or dequeue)
  - How do you test if it's empty
- What if array is full?
- Keeping `size` up to date
- Etc.

```
//sketch only
Enqueue(item) {
 Q[back]=item;
 back = (back+1) % size;
}
```

```
//sketch only
Dequeue() {
 item = Q[front];
 front = (front+1) % size;
 return item;
}
```

Linked List

What do `Enqueue` and `Dequeue` look like? What are their time complexities?



Linked List

What do `Enqueue` and `Dequeue` look like? What are their time complexities?

```
//sketch only
Enqueue(item) {
 tail.next=new Node(item);
 tail=tail.next;
}
```

```
//sketch only
Dequeue() {
 item = head.data;
 head = head.next;
 return item;
}
```

Still just sketches!
Ask the same questions from the circular array.
Notice that sometimes the answer is “don’t worry about it”!

Tradeoffs

What makes the circular queue implementation different from the linked list implementation? In what ways is one more desirable than the other?

Tradeoffs

LINKED LIST

$O(1)$ enqueue and dequeue

Pointers lead to slower constants and worse cache behavior (see CSE351)

More space used per element (pointers take space)

Not in Queue ADT, but if you want to insert at position k, must traverse k elements first.

CIRCULAR ARRAY

$O(1)$ enqueue and dequeue unless the array is already full.

Faster constants, but when you resize suddenly a **very** slow insert.

Less space when full, but potentially **a lot** of wasted space if queue gets huge then tiny.

Not in Queue ADT, but if you want to insert at position k, must shift last n-k elements.

Tradeoffs

This class is built on **tradeoffs**

If data structure A beats data structure B in every way, you'll choose A.

Usually, data structure A is better in some ways, and B is better in others.

Knowing the tradeoffs will help you frame the choice.

If you know the size in advance, you might choose the array (no resize worries)

If any one insertion being slow would be very bad, you might choose the linked list

Common Tradeoffs

Often there are multiple unavoidable tradeoffs

- Time vs. space
- One operation vs. another being as fast as possible
- Generality vs. simplicity vs. performance

These tradeoffs are why there are many data structures.

And well-trained computer scientists have internalized those tradeoffs to pick.

Your TODO list

Make sure you're on Ed.

Get started on Exercise 0 (and update your IDE/java installs while you're at it).