Welcome to CSE 332!
Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Lecture Outline

- About This Course
  - **Learning Objectives**
  - People
  - Policies
- Abstract and Concrete Data Types
- ADTs & Data Structures you've already learned

Learning Objectives

- Learn fundamental, "classic", *data structures* and *algorithms*
- Learn thought processes/patterns for *organizing* and *processing information*
  - Understand how to analyze a program's efficiency
  - Learn how to analyze tradeoffs and pick "the right tool for the job"
  - Learn about how programs work in *parallel* and the related concurrency issues
- Learn to *communicate* about these ideas
  - Explaining your reasoning to others
  - Working with a partner on code
- Learn to *read* and *understand* code you didn't write
- This isn't a "how to program" or "software engineering" class!
  - We will *practice* design, analysis, and implementation
  - Witness elegant interplay of "theory" and "engineering" at the core of computer science
- **Crush** your technical interviews this fall!

Course Content

- What do we mean by “Data Structures and Parallelism”?

- About 70% of the course is a “classic data-structures course”
  - Timeless, essential stuff
  - Core data structures and algorithms that underlie most software
  - How to analyze algorithms

- About 30% is programming with *multiple executors*
  - *Parallelism*: Use multiple executors to finish sooner
  - *Concurrency*: Correct access to shared resources
  - Will make many connections to the classic data structures material

In Other Words …

- This is the class where you begin to think like a computer scientist
  - You stop thinking in Java code
  - You start thinking that this is a hashtable problem, a stack problem, a sorting problem, etc.
  - You recognize tradeoffs
    - Time vs. space
    - One operation more efficient if another less efficient
    - Generality vs. simplicity vs. performance
- We are filling your “toolbox” with tools (data structures and algorithms) and a methodology for selecting the right one
  - Eg, logarithmic < linear < quadratic < exponential

Why take this course?

- Macro:
  - Want to revolutionize some part of the tech industry?
  - Self-driving cars
  - Fake news detection

- Micro:
  - Get everyone on the same page
  - Get internships!
  - Be prepared for industry

Lecture Outline

- About This Course
  - Learning Objectives
  - **People**
  - Policies
- Abstract and Concrete Data Types
- ADTs & Data Structures you've already learned

Introductions: Me

- Kristofer Wong (he/him)
  - Graduated UW CSE last week
  - First time instructor, but 7x TA
- Interests
  - Music
  - Swimming, IMA sports w friends
  - Boba
  - Please dear god no Loki spoilers
- No computer science in high school
  - No experience is ok! (That's why we're here)
- Brain damage in sophomore year: CSE 332???
  - I promise: grades don't matter.
- Industry experience
- Future plans
  - More school :')

Introductions: TA's

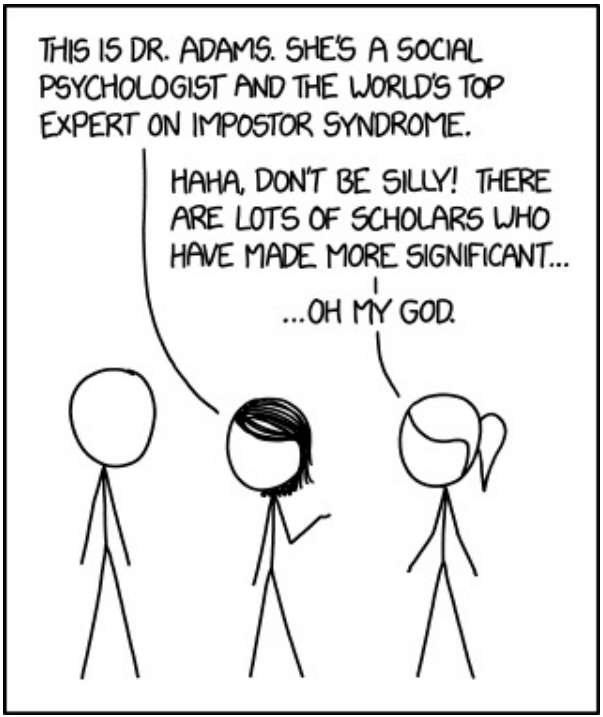
- TAs:
  - Alena, Arya, Finn, Joon, Kimi, Peyton, Rahul, Winston
  - Available in section, office hours, Ed, and 1-on-1's
  - An invaluable source of information and help (!!)

- **Please** get to know us
  - We are excited to help you succeed!
  - Schedule time for a virtual one-on-one to discuss anything

- A couple promises:
  - To do our best to be as inclusive of every student as possible
  - To

Introduction: You

- ~55 students registered
  - All different experience levels
- You are **never** alone
- Toward the end {don't jinx it} of the pandemic, but it's still affecting us.
- "Nearly 70% of individuals will experience signs and symptoms of impostor phenomenon at least once in their life."
  - https://en.wikipedia.org/wiki/Impostor_syndrome



https://xkcd.com/1954

Lecture Outline

- About This Course
  - Learning Objectives
  - People
  - **Policies**
- Abstract and Concrete Data Types
- ADTs & Data Structures you've already learned

Communication

- Website: http://cs.uw.edu/332
  - Schedule, policies, materials, assignments, etc.
- Discussion: https://edstem.org/us/courses/6530/discussion/
  - Announcements made here
  - Ask and answer questions – staff will monitor and contribute
- Office hours: spread throughout the week
  - Can e-mail or private Ed post to make individual appointments
- Feedback:
  - Anonymous feedback goes to Kris, but he can't respond directly
  - cse332-staff@cs goes to the entire staff

Course Components

- Lectures
  - Introduces the concepts (but rarely covers coding details)
  - Try to stay engaged!
    - Why??
  - Slides posted after class
  - Lectures **are** recorded

- Sections
  - Practice problems and concept application
  - Review materials (occasionally introduces new materials)
  - Answer Java/project/homework questions

- Office Hours
  - Come to these, even if you don't know what you're confused about

Materials

- Textbook:
  - *Data Structures & Algorithm Analysis in Java*, Mark Allen Weiss
  - 3rd edition, 2012 (but 2nd edition ok)

- Parallelism/concurrency units in separate free resources specifically designed for 332

- Readings are not required, but heavily encouraged.
  - I didn't do them
  - I wish I did them

Evaluation & Grading

- 14 total homework exercises + 1 EC exercise (35%)
  - 9 individual (+1 EC)
  - 5 communication / group based (look out for an announcement with further explaination)
- 3 *partner-based* multi-phase programming projects (35%)
  - Use Java 11, IntelliJ, Gitlab
  - Partner programming
- 2 assessments (20%)
  - No traditional assessments
  - 1 midterm, self graded, out for a whole week
  - 1 five minute oral ”final”
- Participation (10%)
  - 5%: In class activities
    - due 11:59 PM before the next class
  - 5%: course engagement
    - Asking / Responding on Ed
    - Participation in quiz sections
    - Office Hours
    - Providing course feedback (Private Ed posts, google docs comments, ~~anonymous feedback~~)
    - Generally helping your peers succeed

Deadlines and Student Conduct

- Late policies
  - Exercises & Assessments: No late submissions accepted
  - Projects: 4 late days for the entire quarter, max 2 per assignment
    - If you have extenuating circumstances, reach out to the course staff and we'll try to accommodate.

- Academic Conduct (**read** the full policy in the syllabus)
  - In short: don't attempt to gain credit for something you didn't do and don't help others do so either
  - This does ***not*** mean suffer in silence!
    - Attempt a problem on your own first, but then…
    - Learn from the course staff and peers, talk, share ideas; *but* don't share or copy work that is supposed to be yours
    - Collaboration is **strongly** encouraged! Discuss confusing points with each other, because organizing your thoughts is the best way to learn!

Lecture Outline

- About This Course
  - Learning Objectives
  - People
  - **Policies**
- **Abstract and Concrete Data Types**
- ADTs & Data Structures you've already learned

Terminology: Data Structures vs Algorithms

- **Data Structures**:
  - A way of organizing, storing, accessing, and updating a set of data
  - *Examples from 14X*: arrays, linked lists, trees

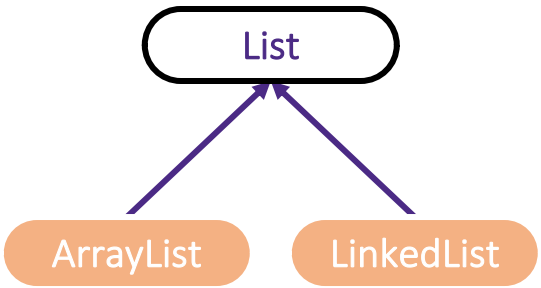
- **Algorithms**:
  - A series of precise instructions guaranteed to produce a certain answer
  - *Examples from 14X*: binary search, merge sort, recursive backtracking

Terminology: ADTs vs Concrete Data Structures

- **Abstract Data Types (ADTs):**
  - Mathematical description of a "thing" and its set of operations
- **Data Structures**:
  - A way of organizing, storing, accessing, and updating a set of data
- **Implementations:**
  - An implementation of an ADT is a data structure
  - An implementation of a data structure are the collection of methods and variables in a specific language

Analogy from 143

- In Java, an **interface** is a data type that specifies what to do but not how to do it
  - **List**: an ordered sequence of elements.
- A **subtype** implements all methods required by the interface
  - **ArrayList**: Resizable array implementation of the List interface
  - **LinkedList**: Doubly-linked implementation of the List interface



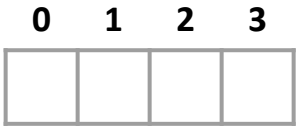
A Java interface is to a Java subtype, as an ADT is to a data structure!

Lecture Outline

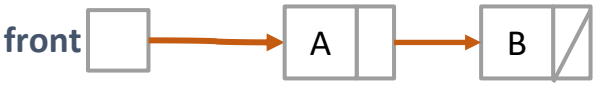
- About This Course
  - Learning Objectives
  - People
  - Policies
- Abstract and Concrete Data Types
- **ADTs & Data Structures you've already learned**

Data Structures from 143

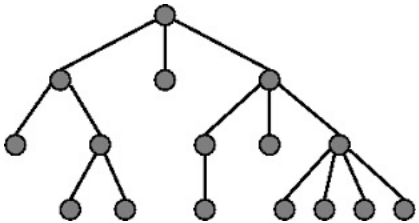
- Arrays



- Linked Lists



- Trees



List Functionality

List ADT. A collection storing an ordered sequence of elements.

- Each element is accessible by a zero-based index
- A list has a size defined as the number of elements in the list
- Elements can be added to the front, back, *or any index in the list*
- Optionally, elements can be removed from the front, back, *or any index in the list*

- Possible Implementations:
  - ArrayList
  - LinkedList

List Performance Tradeoffs

	ArrayList	LinkedList
addFront	linear	constant
removeFront	linear	constant
addBack	constant*	linear
removeBack	constant	linear
get(idx)	const	linear
put(idx)	linear	linear

* constant for most invocations

Stack and Queue ADTs

Stack ADT. A collection storing an ordered sequence of elements.

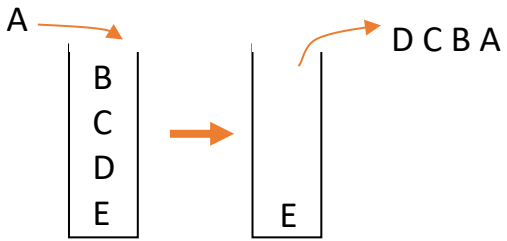
- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top (“LIFO”)

Queue ADT. A collection storing an ordered sequence of elements.

- A queue has a size defined as the number of elements in the queue
- Elements can only be added to one end and removed from the other (“FIFO”)

Stack ADT

- **Stack**: an ADT representing an ordered sequence of elements whose elements can only be added/removed from one end.
  - Corollary: has "last in, first out" semantics (LIFO)
  - The end of the stack that we operate on is called the "top"
  - Operations:
    - `void push(Item i)`
    - `Item pop()`
    - `Item top()/peek()`
    - `boolean isEmpty()`
    - *(notably, there is no generic `get()` method)*



Terminology Example: Stack

- The Stack **ADT** has the following operations:
  - **`push`**: adds an item
  - **`pop`**: raises an error if `isEmpty()`, else **removes** and **returns** *most-recently pushed item* not yet returned by a `pop()`
  - **`top`** or **`peek`**: same as pop, but doesn't remove the item
  - **`isEmpty`**: initially true, later true if there have been same number of `pop()`'s as `push()`es

- A Stack **data structure** could use a linked-list or an array or something else.
  - There are associated **algorithms** for each operation

- One **implementation** is in the library **`java.util.Stack`**

Why care about ADTs?

- We can **communicate** in shorthand and high-level terms
  - “Use a stack and push numbers”
  - Rather than: “create a linked list and add a node when you see a …”

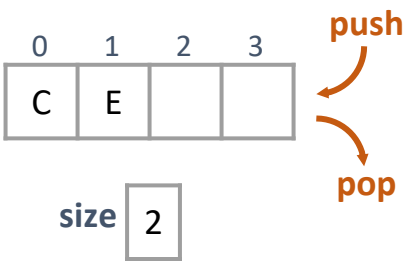
Stack Data Structure: Array

- *State*

```
Item[] data;
int size;
```

- *Behavior*
  - `push()`
    - Resize `data` array if necessary
    - Assign `data[size] = item`
    - Increment `size`
    - *Note: this is `ArrayList.addBack()`*
  - `pop()`
    - Return `data[size]`
    - Decrement `size`
    - *Note: this is `ArrayList.removeBack()`*

```
push('C');
push('D');
pop();  // 'D'
push('E');
```



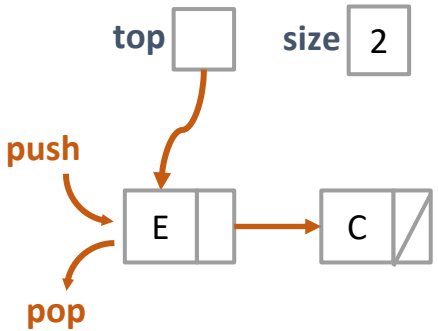
Stack Data Structure: Linked List

- *State*

```
Node top;
```

- *Behavior*
  - `push()`
    - Create a new node linked to `top`'s current value
    - Update `top` to new node
    - Increment `size`
    - *Note: this is `LinkedList.addBack()`*
  - `pop()`
    - Return `top`'s item
    - Update `top`
    - Decrement `size`
    - *Note: this is `LinkedList.removeBack()`*

```
push('C');
push('D');
pop();  // 'D'
push('E');
```



Queue ADT

- **Queue**: an ADT representing an ordered sequence of elements, whose elements can only be added to one end and removed from the other end.
  - Corollary: has "first in, first out" semantics (FIFO)
  - Two methods:
    - `void enqueue(Item i)`
    - `Item dequeue()`
    - `boolean isEmpty()`
    - *(notably, there is no generic `get()` method)*

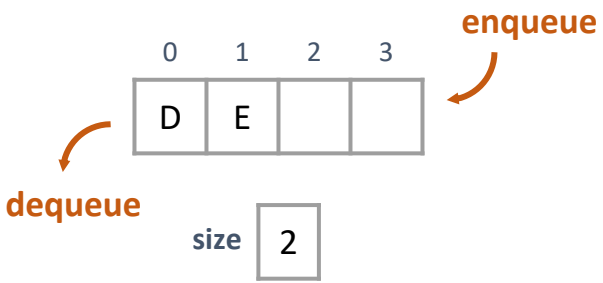
Queue Data Structure: Simple Array

- *State*

```
Item[] data;
int size;
```

- *Behavior*
  - `enqueue()`
    - `ArrayList.addBack()`
  - `dequeue()`
    - `ArrayList.removeFront()`

```
enqueue('C');
enqueue('D');
dequeue();  // 'C'
enqueue('E');
```

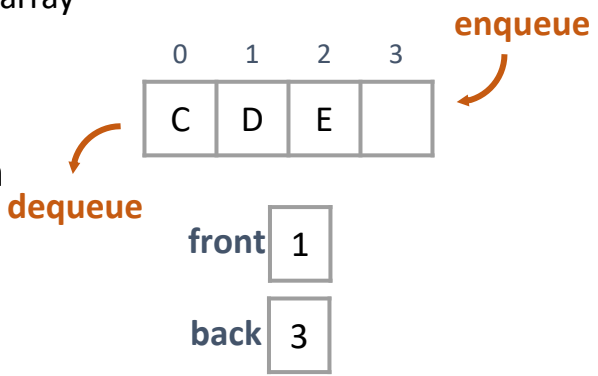


Queue Data Structure: Circular Array

- The front of the queue does not need to be the front of the array!
  - This data structure is also known as a **circular array**
  - Removing items increments `front`
  - Adding items increments `back`
  - `back` "wraps around" to the front of the array if there's capacity

- No longer need to shift elements down during `dequeue()`s

```
enqueue('C');
enqueue('D');
dequeue();  // 'C'
enqueue('E');
```



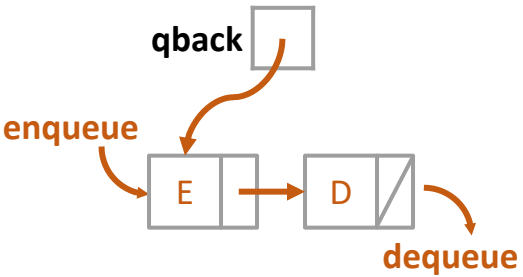
Queue Data Structure: (Singly) Linked List

- *State*

```
Node qback;   // front of
              // list is the
              // logical back
              // of the queue
```

```
enqueue('C');
enqueue('D');
dequeue();  // 'C'
enqueue('E');
```

- *Behavior*
  - `enqueue()`
    - `LinkedList.addLast()`
  - `dequeue()`
    - `LinkedList.removeFront()`



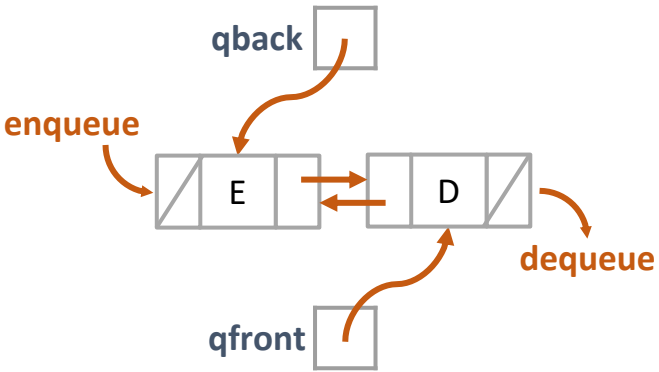
How does our linked list know where the last element is?

Queue Data Structure: Doubly Linked List

- What if we:
  - made the list doubly-linked
  - added a pointer representing the **front** of the queue

- How do I decide which structure to back my queue??
  - Time constraints
  - Space constraints
  - Potential need for operations not in the ADT?

```
enqueue('C');
enqueue('D');
dequeue();  // 'C'
enqueue('E');
```



Dictionary ADT

Dictionary ADT. A collection keys, each associated with a value

- A dictionary has a size defined by the number of elements in the dictionary (key/value pairs)
- You can add and remove key/value pairs, but the keys must be unique
- Each value is accessible by its key via a "find" or "contains" operation

Terminology: a dictionary maps *keys* to *values;* an item or data refers to the key/value pair.

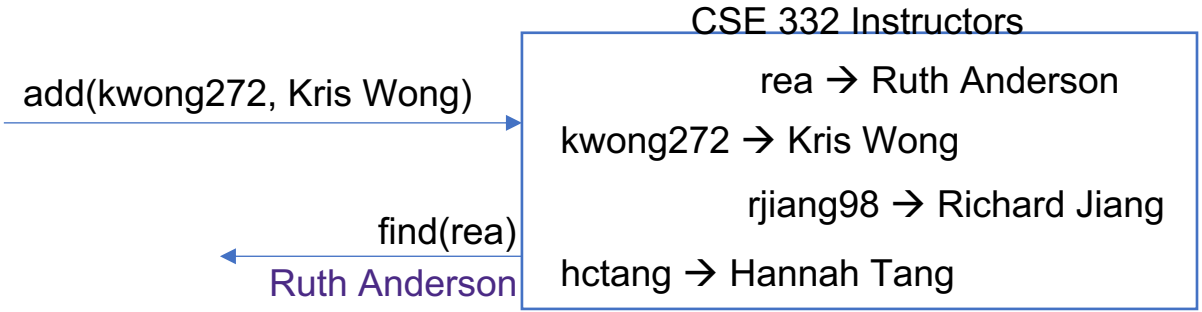
- Also known as: "Map ADT"
  - add(k,v)
  - contains(k,v)
  - find(k)
  - remove(k)
- Naïve implementation: a list of key/value pairs:

```
class KVPair<Key, Value> {
    Key k;
    Value v;
}

LinkedList<KVPair> dict;
```

Dictionary ADT

- We tend to emphasize keys in this class, but don't forget about the associated values!
- Quick example using add and find:



- Dictionaries are **everywhere**
  - Any time you want to store information according to some key and retrieve it efficiently, you want a dictionary!
  - In upper level CS: Networks, OS, Compilers, Databases
  - In the real world: UW NetIDs, Google's indexing, Biology Genome mapping

Set ADT

Set ADT. A collection keys.

- A set has a size defined by the number of elements in the set
- You can add and remove keys, but the keys must be unique
- Each value is accessible by its key via a “find” or ”contains” operation

Look familiar…?

- Operations
  - add(v)
  - contains(v)
  - remove(k)
- Naïve implementation: a list of key/value pairs:

```
class Item<Key> {
   Key k;
}

LinkedList<Item> set;
```

Homework for TODAY!!

- By 11:59 TOMORROW:
  - P1 Partner Matching Survey (Google Forms)
  - Pre-Course Check-In Survey
  - Previous Experience & Course Related Survey
- Due Friday, 11:59:
  - Exercise 1 (Warmup / Java review)

Everything linked on the website!