Intro; ADTs; Lists, Stacks, and Queues

CSE 332 Spring 2021

Instructor: Hannah C. Tang

Teaching Assistants:

Aayushi Modi
Aashna Sheth
Frederick Huyan
Hamsa Shankar
Khushi Chaudhari
Kris Wong
Logan Milandin
Nachiket Karmarkar
Patrick Murphy
Richard Jiang
Winston Jodjana

gradescope

gradescope.com/courses/256241

- (Breakout rooms are hit-or-miss. But research shows that y'all learn better by:
  - Practicing with the materials
  - Forming an opinion / answering questions – even if the opinion turns out to be wrong
- So I'm not going to do group activities. But I still need you to *engage*
  - ie, write down your answers on paper or complete the ungraded Gradescope activity

- Give a (very specific) example of the following:
  1. Pie *(if your student ID is odd)*
  2. Main course *(if your student ID is even)*

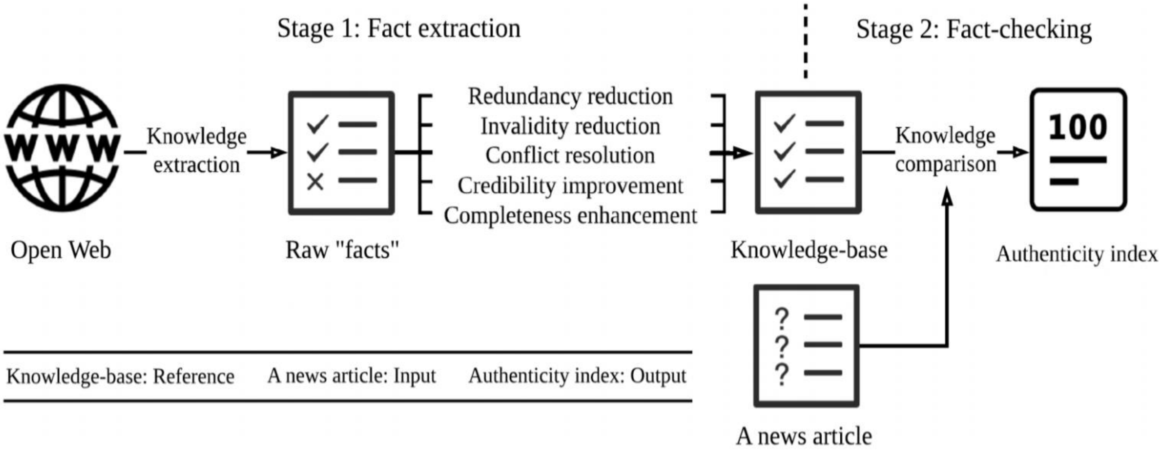
- Example:
  1. Sour cherry pie with a butter (not shortening) crust and canned cherries
  2. Fried rice (楊州炒飯 – with 叉燒, not 火腿 or 臘腸)

Lecture Outline

- **Introduction: Why This Course?**
- About This Course
  - Learning Objectives
  - People
  - Policies
- Abstract and Concrete Data Types
- List, Stack, and Queue ADTs

Why: Increase Progress (?) in Society

Why: Discover New Knowledge



Fake News: A Survey of Research, Detection Methods, and Opportunities (Xinyi Zhou, Reza Zafarani/arXiv:1812.00315)

Why: Support Daily Life

How to search the internet

About 7,470,000,000 results (0.60 seconds)

Lecture Outline

- Introduction: Why This Course?
- About This Course
  - **Learning Objectives**
  - People
  - Policies
- Abstract and Concrete Data Types
- List, Stack, and Queue ADTs

Learning Objectives

- Learn fundamental *data structures* and *algorithms*
  - "Classic" data structures and algorithms
    - Queues, dictionaries, graphs, sorting, etc.

- Learn thought processes/patterns for *organizing* and *processing information*
  - Understand how to analyze their efficiency
  - Learn how to analyze tradeoffs and pick "the right tool for the job"
  - Parallelism and concurrency (!)

- 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

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 and make informed 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

Lecture Outline

- Introduction: Why This Course?
- About This Course
  - Learning Objectives
  - **People**
  - Policies
- Abstract and Concrete Data Types
- List, Stack, and Queue ADTs

Introductions: Course Staff

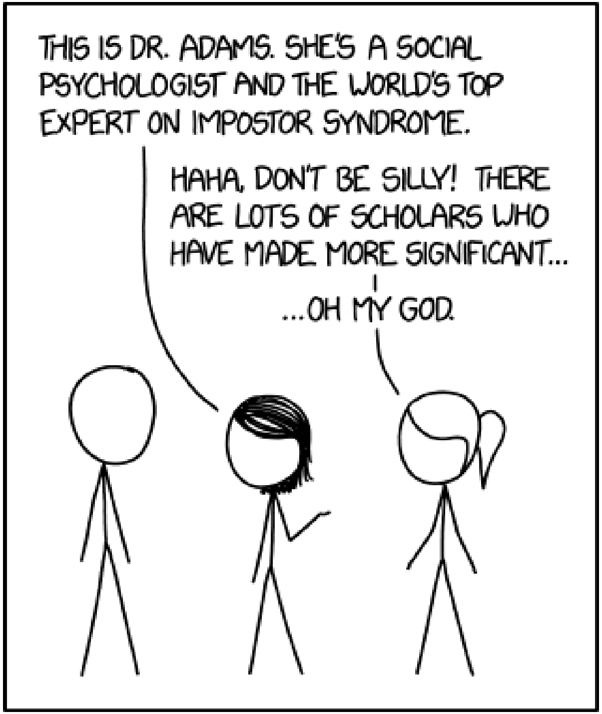
- Hannah C. Tang
  - UW CSE alumna with 17 years ~~of bugs~~ in industry
- TAs:
  - Aayushi, Aashna, Frederick, Hamsa, Khushi, Kris, Logan, Nachiket, Patrick, Richard, Winston
  - Available in section, office hours, and discussion group
  - An invaluable source of information and help (!!)
- Get to know us
  - We are excited to help you succeed!
  - Schedule time for a virtual one-on-one to discuss anything

Introduction: Students

- ~120 students registered, scattered all around the world
  - When we’re online only, it’s easy to feel lost, as if everyone is “better” than you

- “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

- Our course size can be an asset!



https://xkcd.com/1954

Lecture Outline

- Introduction: Why This Course?

- About This Course
  - Learning Objectives
  - People
  - **Policies**

- Abstract and Concrete Data Types

- List, Stack, and Queue ADTs

Communication

- Website: http://cs.uw.edu/332
  - Schedule, policies, materials, assignments, etc.
- Discussion: https://edstem.org/us/courses/4898/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 Hannah, but she can’t respond directly
  - cse332-staff@cs goes to the entire staff

Course Components

- Lectures
  - Introduces the concepts (but rarely covers coding details)
  - Please take notes!!! Slides posted after class
  - ~~(Hopefully)~~ recorded

    Defn!
- Sections
  - Practice problems and concept application
  - Review materials (occasionally introduces new materials)
  - Answer Java/project/homework questions
- Office Hours
  - Use them!

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

Evaluation

- ~3 *partner-based* multi-phase programming projects (50%)
  - Use Java, IntelliJ, Gitlab
  - Three “signature” programming projects plus a collection of smaller parallelism exercises we consider as a “half project”

- No midterm or final exam!!! (40%)
  - Instead, we will have 4 bi-weekly quizzes
  - Released on Tuesday(ish), due on Thursday morning
  - Open book, small-group collaboration allowed. But no staff support (eg, message board or office hours)

- “Participation” (10%)
  - Gradescope activities
  - “90% is 100%”

Deadlines and Student Conduct

- Late policies
  - Projects: Non-linear penalty, no submissions accepted after 48h

- Academic Conduct (**read** the full policy on the web)
  - 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!
    - 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

- Introduction: Why This Course?

- About This Course
  - Learning Objectives
  - People
  - Policies

- **Abstract and Concrete Data Types**

- List, Stack, and Queue ADTs

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, stacks, queues, trees

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

Terminology: Data Structures vs ADTs

- **Data Structures**:
  - A way of organizing, storing, accessing, and updating a set of data
- **Abstract Data Types (ADTs):**
  - Mathematical description of a “thing” and its set of operations
- **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

Intuitively: Data Structures vs ADTs (1 of 2)

- Remember Our Potluck?
  - “Give a (very specific) example of a pie or main course”

- The ADTs and data structures you’ll learn are a cookbook
  - ADTs are the chapters/categories: Soups, Salads, Cookies, Cakes, etc
    - High-level descriptions of a category of functionality
    - You don’t serve a soup when guests expect a cookie!
  - Data structures are the recipes: chocolate chip cookies, snickerdoodles, etc
    - Step-by-step, concrete descriptions of an item with specific characteristics
    - Understand your tradeoffs before replacing carrot cake with a wedding cake

- Anyone have a pie that could serve as a main course?

Intuitively: Data Structures vs ADTs (2 of 2)

- *The ADTs and data structures you'll learn are a cookbook*
  - *ADTs are the chapters/categories: Soups, Salads, Cookies, Cakes, etc*
  - *Data structures are the recipes: chocolate chip cookies,*
- When you go out into the world …
  - Determine which category is required
  - Choose the specific recipe that best fits the situation

Learn how to analyze tradeoffs and pick "the right tool for the job"

Lecture Outline

- Introduction: Why This Course?

- About This Course
  - Learning Objectives
  - People
  - Policies

- Abstract and Concrete Data Types

- **List, Stack, and Queue ADTs**

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")

gradescope

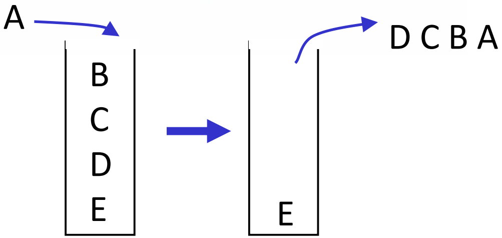
gradescope.com/courses/256241

- Just like we did with our potluck example, give real-life examples of each ADT:
  - List
  - Stack
  - Queue

- Please choose examples that are not software related
  - Eg: an escalator is an example of a Queue ADT

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)*



Stack ADT: Details

- 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'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`**

Stack ADT: Applications

- The ***Stack* ADT** is a useful abstraction because:
  - It arises **all the time** in programming (see Weiss for more)
    - Recursive function calls
    - Balancing symbols (parentheses)
    - Evaluating postfix notation: 3 4 + 5 *
    - Clever: Infix ((3+4) * 5) to postfix conversion (see Weiss)

- 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 …"

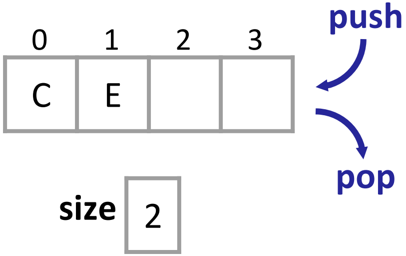
Stack Data Structure: Array

- *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');
```



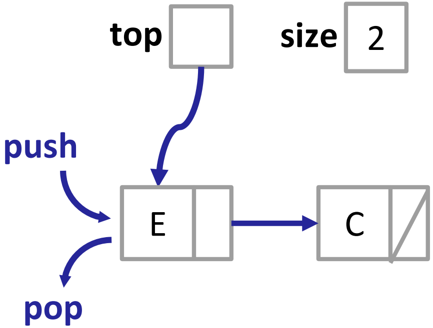
Stack Data Structure: (Singly) Linked List

- *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)*

G **enqueue** → F E D C B **dequeue** → A

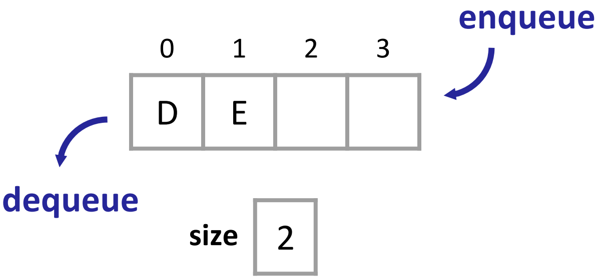
Queue Data Structure: Simple Array

- *State*

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

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

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

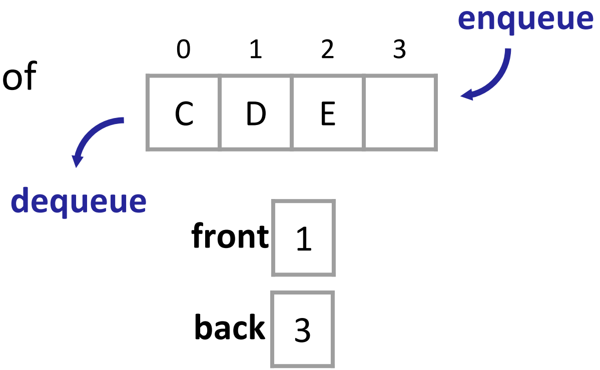


What else happens during the removal?

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');
```



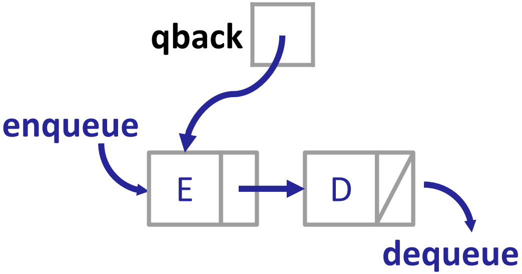
Queue Data Structure: (Singly) Linked List

- *State*

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

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

- *Behavior*
  - `enqueue()`
    - `LinkedList.addFirst()`
  - `dequeue()`
    - `LinkedList.removeLast()`

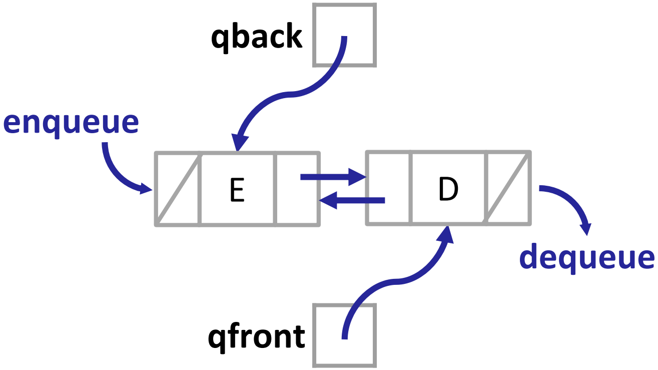


How does our linked list know where the last element is?

Queue Data Structure: Doubly Linked List

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

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



Summary (1 of 2)

- Definitions
  - **Data Structures**: A way of organizing, storing, accessing, and updating a set of data
  - **Algorithms**: A series of precise instructions guaranteed to produce a certain answer
  - **Abstract Data Types (ADTs):** Mathematical description of a “thing” and its set of operations

Summary (2 of 2)

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*

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")