Welcome to CSE 332!

CSE 332 Summer 2020

Instructor: Richard Jiang

Teaching Assistants:

Hamsa Shankar	Kristin Li	Winston Jodjana
Maggie Jiang	Hans Zhang	Michael Duan
Jeffery Tian	Annie Mao	

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-22A

Lecture Outline

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

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-22A

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"
- 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 threads*
  - *Parallelism*: Use multiple processors to finish sooner
  - *Concurrency*: Correct access to shared resources
  - Will make many connections to the classic 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
- That is why we cover many data structures; educated CSEers internalize their tradeoffs and techniques
  - And recognize logarithmic < linear < quadratic < exponential

Lecture Outline

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

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-22A

Introductions: Course Staff

- Richard Jiang
  - UW CSE graduate, many time TA for 332
- TAs:
  - Hans Zhang, Hamsa Shankar, Maggie Jiang, Michael Duan, Jeffery Tian, Kristin Li, Winston Jodjana
    - Consulting TA: Annie Mao
  - Available in section, office hours, Piazza, and 1-on-1s
  - 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

Lecture Outline

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

Lecture Q&A: pollev.com/332summer

Lecture clarifications: tinyurl.com/332-06-22A

Communication

- Website: http://cs.uw.edu/332
  - Schedule, policies, materials, assignments, etc.
- Discussion: http://piazza.com/washington/summer2020/cse332
  - Announcements made here
  - Ask and answer questions – staff will monitor and contribute
- Office hours: spread throughout the week
  - Can e-mail/private Piazza post to make individual appointments
- Feedback:
  - Anonymous feedback goes to Richard, 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)
  - Take notes!!! They will help you stay engaged. Slides posted after class
  - (Hopefully) recorded
- Sections
  - Practice problems and concept application
  - Review materials (occasionally introduces new materials)
  - Answer Java/project/homework questions
- Office Hours
  - Use them to work with other students and TAs

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

- ~16 *individual* homework exercises (25%)
- 3 *partner-based* multi-phase programming projects (35%)
  - Use Java 11 and IntelliJ, Gitlab
  - Done in partners
- No midterm or final exam!!! (40%)
  - Instead, we will have 5 bi-weekly quizzes
  - Released on Thursday, due on Friday
  - Open book, small-group collaboration encouraged. But no staff support (eg, Piazza or office hours questions)

Deadlines and Student Conduct

- Late policies
  - Exercises & Quizzes: no late submissions accepted
  - Projects: 4 "tokens" for the entire quarter
    - Every token is worth 1 late day

- 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**
- 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, binary trees

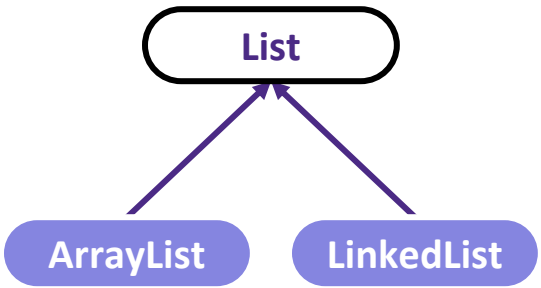
- **Algorithms**:
  - A series of precise instructions guaranteed to produce a certain answer
  - *Examples from 14X*: binary search, merge sort, recursive backtracking

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

A Helpful analogy?

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

Lecture questions: pollev.com/cse332summer

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)	constant	linear
put(idx)	linear	linear

* constant for most invocations

Stack and Queue ADTs

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

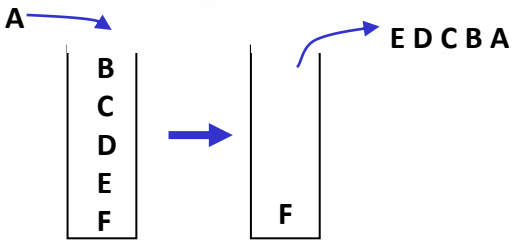
- A stack has a size defined as the number of elements in the stack
- Elements can only be added and removed from the top ("LIFO")

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: Stacks

- 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 talk about ADTs?

- The **ADT** is a useful abstraction because 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 …"

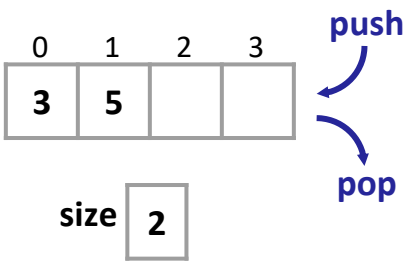
Stack Data Structure: Array

- *State*

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

- *Behavior*
  - `push()`
    - Resize `data` array if necessary
    - Assign `data[size] = item`
    - Increment `size`
  - `pop()`
    - Return `data[size]`
    - Decrement `size`

```
push(3);
push(4);
pop();
push(5);
```



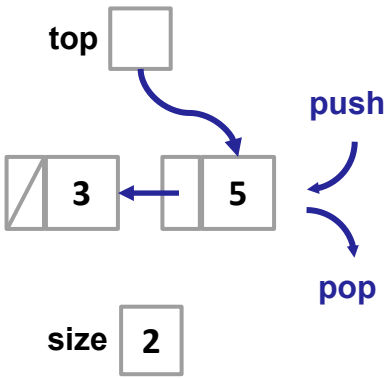
Stack Data Structure: Linked List

- *State*

  `Node top;`

- *Behavior*
  - `push()`
    - Create a new node linked to `top`'s current value
    - Update `top` to new node
    - Increment `size`
  - `pop()`
    - Return `top`'s item
    - Update `top`
    - Decrement `size`

```
push(3);
push(4);
pop();
push(5);
```



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)*

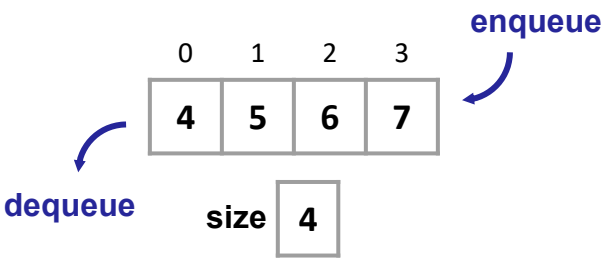
Queue Data Structure: Array

```
enqueue(3);
enqueue(4);
dequeue();
enqueue(5);
enqueue(6);
enqueue(7);
```

- *State*

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

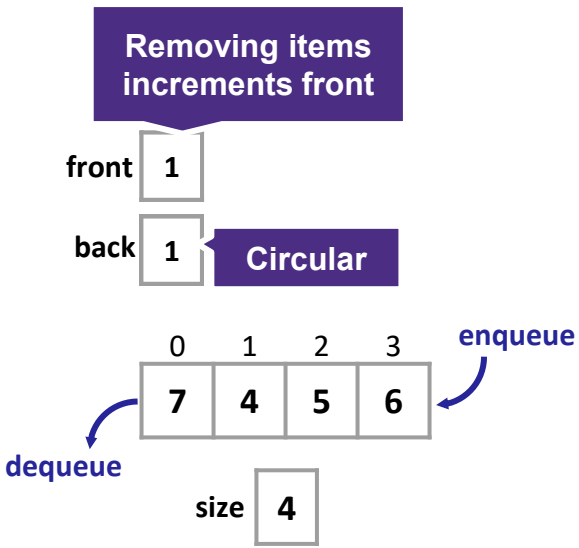
- *Behavior*
  - `enqueue()`
    - `ArrayList.addBack()`
  - `dequeue()`
    - `ArrayList.removeFront()`



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**

```
enqueue(3);
enqueue(4);
dequeue();
enqueue(5);
enqueue(6);
enqueue(7);
```

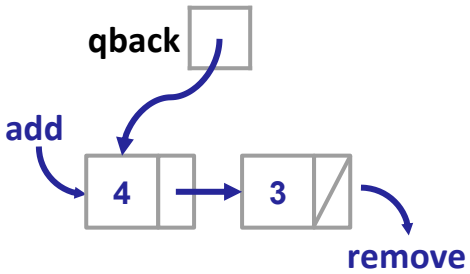


Queue Data Structure: Linked List

- *State*

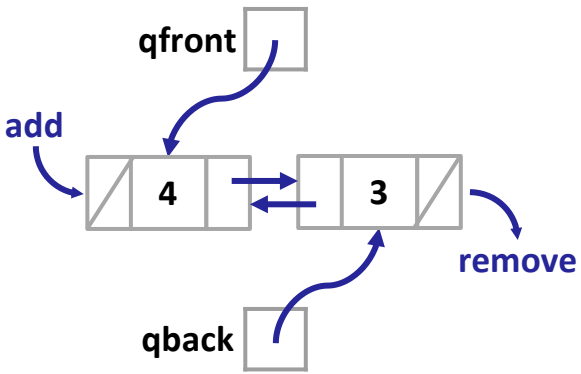
```
Node qback;   // front of list
              // is back of
              // queue
```

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



Queue Data Structure: Doubly Linked List

- What if we made the list doubly-linked and added a **front** pointer?



Circular Array vs Doubly Linked List

- May waste unneeded space or run out of space
- Space per element excellent
- Operations very simple / fast

- Operations not in Queue ADT, but also:
  - Constant-time "access to kth element"
  - For operation "insertAtPosition", must shift all later elements

- Always just enough space
- But more space per element
- Operations very simple / fast

- Operations not in Queue ADT, but also:
  - No constant-time "access to kth element"
  - For operation "insertAtPosition" must traverse all earlier elements

Homework for Today!!

- Project #1:
  - Fill out partner request survey(s) by 6pm PDT TOMORROW
  - Review Java & install IntelliJ
- Exercise #1
  - Due FRIDAY at 11:59pm
- Section #1
  - Fill out breakout room request survey by 6pm PDT TOMORROW

All of these will be linked on the website!

Come see us in Office Hours!

Dictionary ADT (1 of 2)

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

- A dictionary has a size defined as the number of elements in the dictionary
- You can add and remove (key, value) pairs , but the keys are 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

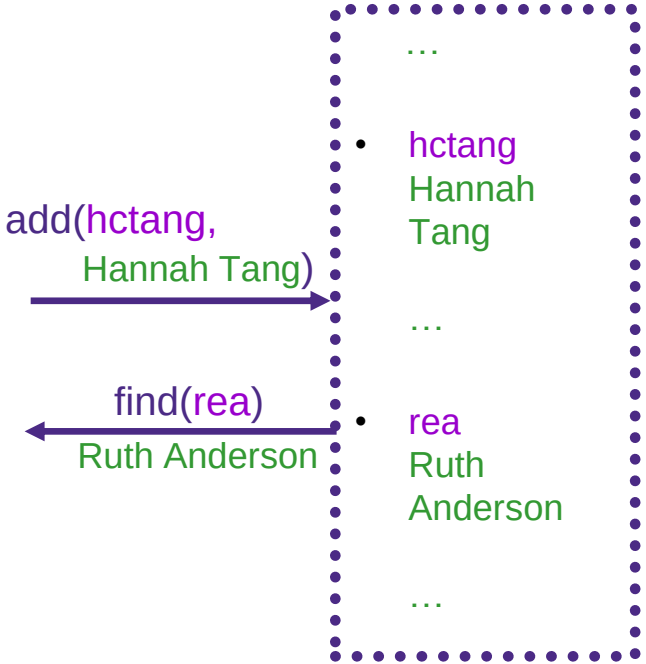
- Also known as "**Map ADT**"
  - add(k, v)
  - contains(k)
  - find(k)
  - remove(k)
- Naïve implementation: a list of (key, value) pairs

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

LinkedList<KVPair> dict;
```

Dictionary ADT (2 of 2)

- Operations:
  - **`add(k, v)`:**
    - places (k,v) in dictionary
    - if key already present, typically overwrites existing entry
  - **`find(k)`:**
    - Returns v associated with k
  - **`contains(k)`:**
    - Returns true if k is in the dictionary
  - **`remove(k)`:**
    - …



We will tend to emphasize the keys, but don't forget about the stored values!

A Modest Few Uses for Dictionaries

- Any time you want to store information according to some key and be able to retrieve it efficiently – a **dictionary** is the ADT to use!
  - Lots of programs do that!

Networks	Router tables
Operating systems	Page tables
Compilers	Symbol tables
Databases	Dictionaries with other nice properties
Search	Inverted indices, phone directories, …
Biology	Genome maps

Set ADT

Set ADT. A collection of keys.

- A set has a size defined as the number of elements in the set
- You can add and remove keys, but the contained values are unique
- Each key is accessible via a "contains" operation

- Operations:
  - add(v)
  - contains(v)
  - remove(v)

- Naïve implementation: a dictionary where we ignore the "value" portion of the (key, value) pair

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

LinkedList<Item> set;
```

Comparison: Set ADT vs. Dictionary ADT

- The *Set* ADT is like a Dictionary without any values
  - A key is *present* or not (no repeats)

- For **`contains`**, **`add`**, **`remove`**, there is little difference
  - In dictionary, values are "just along for the ride"
  - So *same data-structure ideas* work for dictionaries and sets
    - Java HashSet implemented using a HashMap, for instance

- Set ADT may have other important operations
  - **`union`**, **`intersection`**, **`isSubset`**, etc.
  - Notice these are binary operators on sets
  - We will want different data structures to implement these operators

Summary (1 of 2)

- **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
- **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

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")