Stack and Queue ADTs

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston	Ethan Knutson	Nathan Lipiarski
Amanda Park	Farrell Fileas	Sam Long
Anish Velagapudi	Howard Xiao	Yifan Bai
Brian Chan	Jade Watkins	Yuma Tou
Elena Spasova	Lea Quan	

Announcements: 1 of 2

- If you’re enrolled and don’t have your Gitlab/Piazza/Gradescope accounts yet, email cse373-staff@cs
- I don’t have add codes ☹
  - Keep trying! You can petition the CSE advising office next week
    - Drop into any section tomorrow
  - Email cse373-staff@cs to get added to Piazza/Gradescope so you can do the reading quizzes and QCs
    - We’re still struggling with Gitlab (for homeworks)

Announcements: 2 of 2

- Homework 1 is released!
- Reading Quizzes + QCs are “80% is 100%”, and are primarily graded on participation.
- Extra Drop-in Times being scheduled for Saturday; check Piazza/website later this week for more details

Lecture Outline

- **ADTs and Interfaces; Data Structures and Subtypes**
- Introduction to Runtime Analysis
- Stack ADT
- Queue ADT
- ArrayList and LinkedList as implementations of Lists, Stacks, and Queues

Questions from Reading Quiz: 1 of 2

- Who or what is the implementor? The client?
- What is a Representation Invariant? Why does it matter?
- ADTs, (concrete) Data Structures, interfaces, and subtypes all feel like the same thing. See next slide!
- Does a representation invariant apply to *Abstract* Data Types (ADTs) or *Concrete* Data Structures?
- Does a representation invariant apply to the client or the implementor?

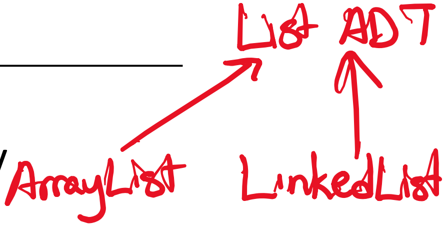
Questions from Reading Quiz: 2 of 2

- So how does `ArrayList.removeFront` actually work?
  - Demo: “nullifying” in `removeFront`
  - Demo: shifting element in `removeFront`

Generic Term	Java Term
ADT	Interface
Data Structure	Subtype

List ADT; ArrayList and LinkedList Data Structures

- **List**: An ADT representing an ordered sequence of elements.
  - Each element is accessible by a zero-based index.
  - Elements can be added to the front, back, or any index in the list.
  - Elements can be removed from the front, back, or any index



- `ArrayList`: A dynamically-resizing array
- `LinkedList`: A dynamically-allocated linear collection of nodes

Lecture Outline

- ADTs and Interfaces; Data Structures and Subtypes
- **Introduction to Runtime Analysis**
- Stack ADT
- Queue ADT
- ArrayList and LinkedList as implementations of Lists, Stacks, and Queues

Poll Everywhere

pollev.com/uwcse373

- Which **List** ADT implementation has a faster implementation for `removeFront()`?

A. Resizable array
[✓ B.] Linked nodes
C. Both are about the same
D. I'm not sure …

What is Runtime Analysis?

- What does it mean for a data structure to be “slow” or “fast”?

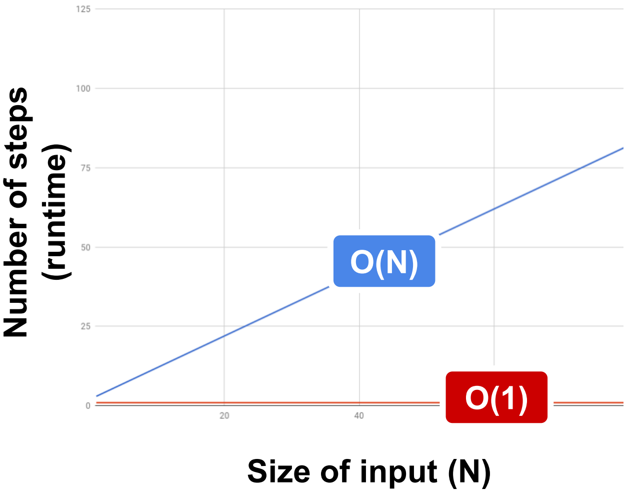
- Let’s run it and measure the (wallclock) time! Oh wait …
  - Input can affect runtime
  - Hardware 〃 〃 〃
  - Other programs

- Count how many steps a program takes to execute on an input of size N

Runtime Analysis, Intuitively

Suppose our list has N items.

- A method that takes a **constant** number of steps (e.g. 23) is in **O(1)**.
- A method that takes a **linear** number of steps (e.g. 4N + 3) is in **O(N)**.



- What is the runtime for `get`() and `removeFront`(), for each possible implementation of our **List** ADT?

	get	removeFront
ArrayList	constant	linear
LinkedList	linear	constant

Discuss: ArrayList vs. LinkedList

1. Which **List** implementation should we use to store a list of songs in a playlist?
2. Which **List** implementation should we use to store the history of a bank customer's transactions?
3. Which **List** implementation should we use to store the order of students waiting to speak to a TA at a tutoring center?

Lecture Outline

- ADTs and Interfaces; Data Structures and Subtypes
- Introduction to Runtime Analysis
- **Stack ADT**
- Queue ADT
- ArrayList and LinkedList as implementations of Lists, Stacks, and Queues

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"
  - Two methods:
    - `void push(Item i)`
    - `Item pop()`
    - *(notably, there is no `get()` method)*

Poll Everywhere

pollev.com/uwcse373

- Which **Stack** ADT implementation is faster overall? Recall that Stacks only have two operations: `push()` and `pop()`.

A. Resizable array
B. Linked nodes
C. Both are about the same
D. I'm not sure …

	push	pop
array	constant ☆	constant
linked list	constant	constant

☆ constant most invocations. But occasionally linear when need to resize

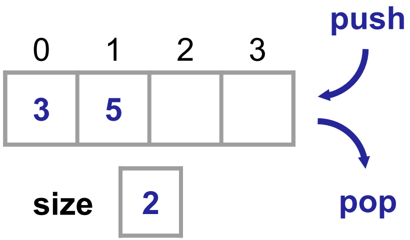
ArrayStack

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

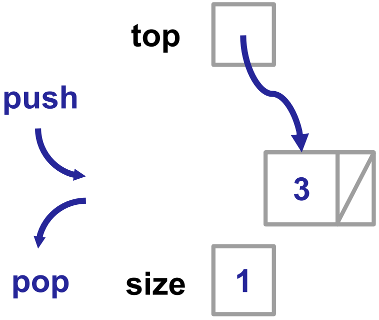


Notice how ArrayStack is a "rebranded ArrayList"!

LinkedStack

- *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(3);
push(4);
pop();
```



Notice how LinkedStack is a "rebranded LinkedList"!

Lecture Outline

- ADTs and Interfaces; Data Structures and Subtypes
- Introduction to Runtime Analysis
- Stack ADT
- **Queue ADT**
- ArrayList and LinkedList as implementations of Lists, Stacks, and Queues

Review: ArrayList vs. LinkedList

1. *Which List implementation should we use to store a list of songs in a playlist?*

2. *Which List implementation should we use to store the history of a bank customer's transactions?*

3. Which List implementation should we use to store the order of students waiting to speak to a TA at a tutoring center?

This can be a Queue ADT!

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()`
    - *(notably, there is no* `get()` *method)*

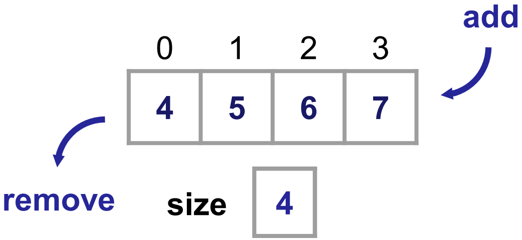
ArrayQueue (v1)

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

- *State*

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

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



- Runtime?
  - `enqueue()` constant ☆
  - `dequeue()` linear

most invocations are constant but occasionally linear when underlying array is resized

Notice how ArrayQueue is a "rebranded ArrayList"!

Poll Everywhere

pollev.com/uwcse373

- What are the runtimes for ArrayQueue: Design 1's `enqueue()` and `dequeue()` methods?

A. Linear / Linear
B. Linear / Constant
[✓ C. Constant / Linear]
D. Constant / Constant
E. I'm not sure …

Discuss: Consider Data Structure Invariants

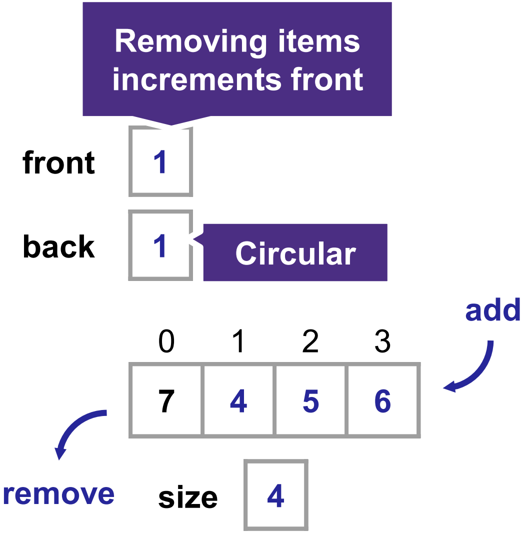
- ArrayQueue (v1) is basically an `ArrayList`.
- Recall the representation invariant for the `data` array in an `ArrayList`:
  - `data` is an array of items, never `null`
  - The `i`-th item in the list is always stored in `data[i]`
    - *This invariant affects the runtimes for `enqueue()` and `dequeue()`!*

ArrayQueue (v2)

	enqueue	dequeue
ArrayQueue (v1)	constant *	linear
ArrayQueue (v2)	constant *	constant

- If we relax the second invariant, 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);
```



Removing items increments front

front 1

back 1

Circular

add

0	1	2	3
7	4	5	6

remove

size 4

Poll Everywhere

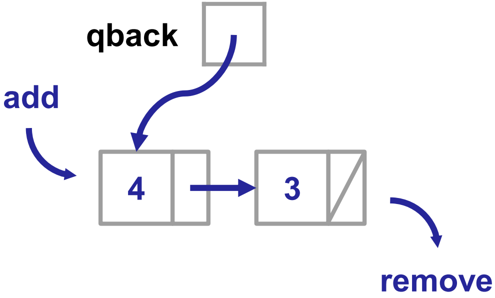
pollev.com/uwcse373

- Give an invariant that describes ArrayQueue (v2) in your own words.

LinkedQueue (v1)

- *State*

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

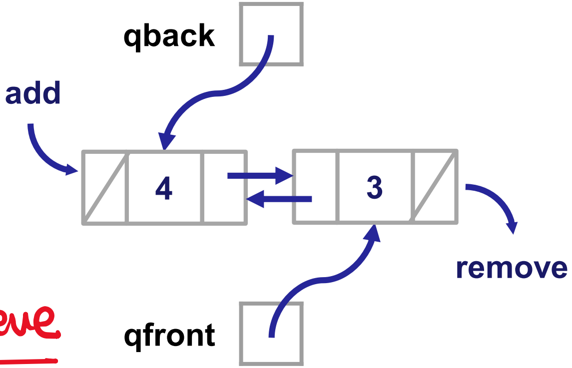


- *Behavior*
  - `enqueue()`
    - `LinkedList.addLast()`
  - `dequeue()`
    - `LinkedList.removeFront()`
- Runtime?
  - `enqueue()` linear
  - `dequeue()` constant

Notice how ArrayQueue is a "rebranded ArrayList"!

LinkedQueue (v2)

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



	enqueue	dequeue
LinkedQueue (v1)	linear	constant
LinkedQueue (v2)	constant	constant

Lecture Outline

- ADTs and Interfaces; Data Structures and Subtypes
- Introduction to Runtime Analysis
- Stack ADT
- Queue ADT
- **ArrayList and LinkedList as implementations of Lists, Stacks, and Queues**

Comparing ADT Implementations: List

	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

Comparing ADT Implementations: Stacks and Queues

- Stack (LIFO):

	ArrayStack	LinkedStack
push	constant*	constant
pop	constant	constant

* constant for most invocations

- Queue (FIFO):

	Array Queue (v2)	LinkedQueue (v2)
enqueue	constant*	constant
dequeue	constant	constant

* constant for most invocations

tl;dr

- More than one concrete data structure can implement an ADT
  - Eg: `ArrayList` and `LinkedList` both implement **List** ADT

- More than one ADT can be implemented by a concrete data structure
  - Eg: `ArrayList` implements both the **List** ADT and the **Stack** ADT

- Looking critically at representation invariants helps us design efficient data structures
  - Eg: we sped up our **Queue**-implementing data structures by removing (`ArrayQueue`) or adding (`LinkedQueue`) a representation invariant