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`
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
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?

<table>
<thead>
<tr>
<th>Method</th>
<th><code>ArrayList</code></th>
<th><code>LinkedList</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get()</code></td>
<td>constant</td>
<td>linear</td>
</tr>
<tr>
<td><code>removeFront()</code></td>
<td>linear</td>
<td>constant</td>
</tr>
</tbody>
</table>
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)*
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 ...

\[\begin{array}{c|c|c}
\text{array} & \text{push} & \text{pop} \\
\hline
\text{constant} & \text{constant} \\
\hline
\text{linked list} & \text{constant} & \text{constant}
\end{array}\]

*Constant most invocations. But occasionally linear when need to reallocate*
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()

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()

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)

- **State**
  
  Item[] data;
  
  int size;

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

- **Runtime?**
  
  - enqueue() constant
  
  - dequeue() linear

Notice how ArrayQueue is a “rebranded ArrayList”!
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)

- 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

```java
enqueue(3);
enqueue(4);
dequeue();
enqueue(5);
enqueue(6);
enqueue(7);
```
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?
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 ADT
- ArrayList
  - addFront \( L \)
  - removeFront \( L \)
  - addBack \( C \)
  - removeBack \( C \)
  - get \( C \)
- LinkedList
  - addFront \( C \)
  - removeFront \( C \)
  - addBack \( L \)
  - removeBack \( L \)
  - get \( L \)

❖ Stack ADT (LIFO)
- ArrayStack (aka ArrayList)
  - push \( C \)
  - pop \( C \)
- LinkedStack (aka LinkedList)
  - push \( C \)
  - pop \( C \)

❖ Queue ADT (FIFO)
- ArrayQueue (v2)
  - enqueue \( C \)
  - dequeue \( C \)
- LinkedQueue (v2)
  - enqueue \( C \)
  - dequeue \( C \)

\( C \) = constant

\( L \) = linear

\( \ast \) = constant for most invocations
Conclusions

❖ 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