CS151 Intro to Data Structures

Queues

Lists

Announcements

- HW02 was due Friday

- HW03 (Stacks & Queues) released – due Friday 10/10
  - shorter than hw2
  - Lab5 also due Friday

Outline

- Stacks & Queues review
- Dequeues
- ADTs
- Iterators

Stacks Review - FILO

- First In Last Out
- *stack* of plates in the dining hall

- Big O of the following operations - designed to be O(1)
  - push
  - pop
  - peek

- How did we do this with LL?
- How did we do this with an array?

Queues - First-in First-out

The first item in, is the first item out (grocery line)

`Queue` Interface

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

- `null` is returned from `dequeue()` and `first()` when queue is empty

Implementing a Queue with an Array

Let’s code it.

- enqueue? O(1)
- dequeue? O(1)
  - O(n) if we don’t have the markers for first and last elem
- isEmpty? O(1)
- first? O(1)

Implementing a Queue with a LinkedList

- Goal: O(1) operations for enqueue and dequeue
- enqueue: add to tail
- dequeue: remove from front
- How can we make these O(1) ?

Implementing a Queue with a LinkedList

- enqueue?
  - runtime complexity?
    - O(1)
- dequeue?
  - runtime complexity?
    - O(1)
- first?
  - runtime complexity?
    - O(1)
- isEmpty?
  - runtime complexity?
    - O(1)

Performance and Limitations

- Performance
  - let n be the number of objects in the queue
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

- Limitations
  - Depending on the implementation,
    - max size is limited and can not be changed
    - Or need to grow the array when out of room

Deques / “Decks”

Queues (FIFO)

- Restaurant Waitlist
  - FIFO make sense!
  - What if we pop off a party in the waitlist but then a table wasn’t ready... we need to add them back in... push will put them at the end!

- Solution? Deques
  - gives us more flexibility

```java
public interface Deque<E> {

    //returns number of elements in the deque
    int size();

    //returns true if the deque is empty, false otherwise
    boolean isEmpty();

    //returns top element in the deque (or null if empty)
    E first();

    //returns top element in the deque (or null if empty)
    E last();

    //inserts the element e to the beginning
    void addFirst(E e);

    //adds the element e to the end
    void addLast(E e);

    //returns the first element in the dequeu (or null if empty)
    E removeFirst();

    //returns the last element in the dequeu (or null if empty)
    E removeLast();
}
```

Deques

- Implemented as array with O(1) operations:
  - max capacity (no expansion)
  - front marker
  - rear maker

Front and Back Markers

addFirst:

```
    f = (f - 1 + n) % n;
```

addLast:

```
    r = (r + 1) % n;
```

removeFirst:

```
    f = (f + 1) % n;
```

removeLast:

```
    r = (r - 1 + n) % n;
```

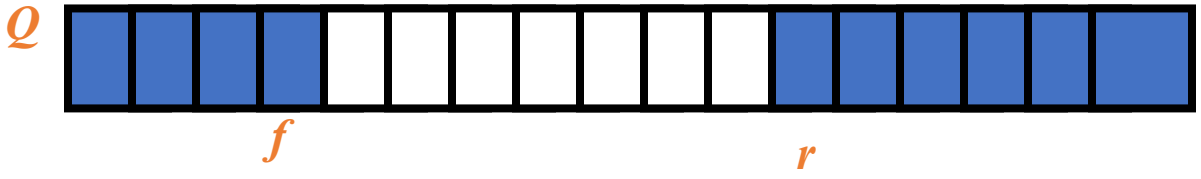
where n is <u>size of the array</u>

Circular Queue

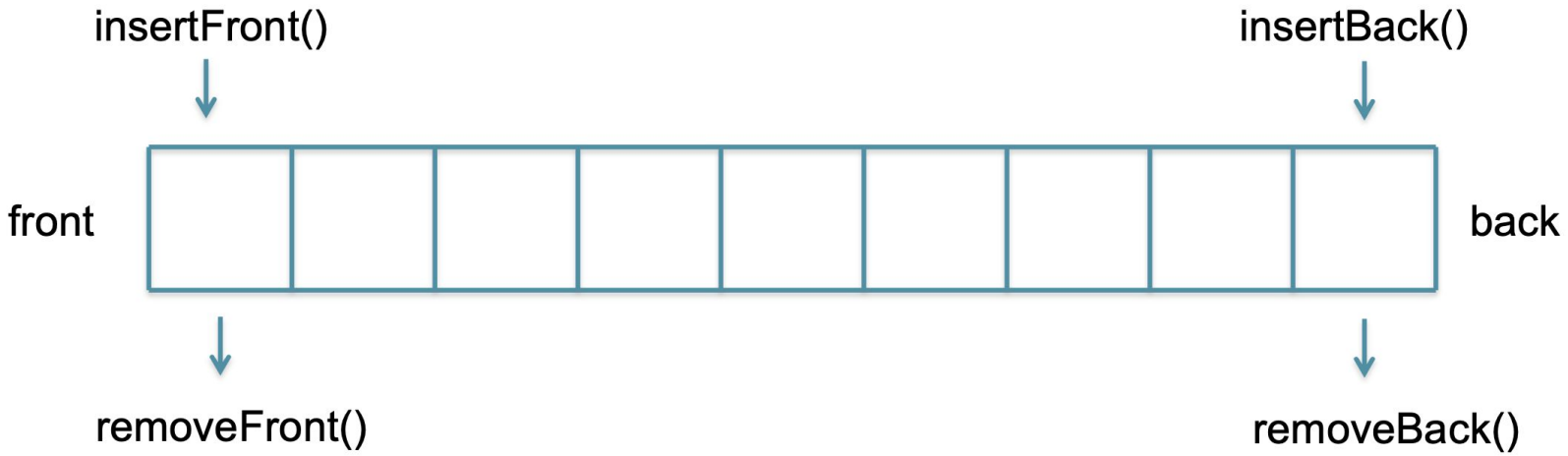
`r = (r+1)%n` is the first empty slot past the rear of the queue

where `n =` length of the array

wrapped-around configuration



Doubled-ended Queue (aka Deques aka “Decks”)



Dynamic Data Structure used for storing sequences of data

- Insert/Remove at either end in O(1)
- If you exclusively add/remove at one end,
  - then ***it becomes a stack***
- If you exclusive add to one end and remove from other,
  - then ***it becomes a queue***

HW3 Discussion

Part one: TwoStacksQueue

Part two: Deque implemented with an Array

JUnit test cases required!

ADTs

Abstract Data Types

- high-level description of a set of operations that can be performed on a data structure
- It defines the behavior of a data type independently of its implementation
- Cannot instantiate
- What does this remind you of that we've learned so far?

Queue ADT

https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html

Look at the “Implementing Classes”

Abstract Data Types

- There are multiple ways to implement a data structure each with different trade offs
  - Ex. stack can be implemented with an array or a linked list

- **List ADT:**
  - supports a *linear sequence of elements*

Lists

`java.util.List` ADT

size( ): Returns the number of elements in the list.

isEmpty( ): Returns a boolean indicating whether the list is empty.

get(i): Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

set(i, e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.

remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

Example

Method	Return Value	List Contents
add(0, A)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)		
set(2, C)		
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

List ADT

https://docs.oracle.com/javase/8/docs/api/java/util/List.html

Look at the “all known implementing classes”

We’re going to focus on ArrayList today

List ADT

Reminder of our methods:

size( ): Returns the number of elements in the list.

isEmpty( ): Returns a boolean indicating whether the list is empty.

get(i): Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

set(i, e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.

remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

ArrayList

Big-O memory?

- O(n)

Indexing / random access?

- O(1)

Add / remove?

- O(n)

Iterators

Iterators

- represents a sequence of elements and provides a way to iterate, or traverse, through those elements one at a time

Iterators

- Abstracts the process of scanning through a sequence of elements (traversal)
- provides a way to iterate, or traverse, through elements one at a time

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

- Combination of these two methods allow a generic traversal structure

```
while(iter.hasNext()) {
  iter.next();
}
```

Iterators

- **code**

- Can an iterator go backwards? NO. Only can do `next()`

`Iterable` Interface

- What can i use an `iterator` on? Anything that implements the `iterable` interface.

- Each call to `iterator()` returns a new iterator instance, thereby allowing traversals of a collection

- `List` interface extends `Iterable` and `ArrayList` implements `List`

`Iterable` Interface

An interface with a single method:

- `iterator()`: returns an iterator of the elements in the collection

`Iterator` Interface

`Iterator` Interface

Another interface that supports iteration

- `boolean hasNext()`
- `E next()`
- `void remove()`

- `Scanner` implements `Iterator<String>`
- `ArrayList` inner class `ArrayListIterator` implements `Iterator`

Let’s make ExpandableArray iterable

`Iterable` versus `Iterator`?

- `Iterable`
  - `java.lang`
  - override `iterator()`
  - Doesn't store the iteration state
  - Removing elements during iteration isn't allowed
- `Iterator`
  - `java.util`
  - Override `hasNext()`, `next()`
  - Optional `remove()`
  - Stores iteration state (list cursor)
  - Removing elements during iteration supported

Summary

- Stacks and Queues are limited data structures with O(1) operations
    - surprisingly useful in practice
- Dequeues
    - O(1) data structure where you can add to front or back
    - cannot access middle!
- ADTs
    - Abstract, implementation independent DS
    - Queue is an ADT that can be implemented with LL or Array
- Iterators
    - Abstracts the process of traversal