CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Lec 15: Stacks and Queues

Announcements

- WA2 due Sunday @ 11:59PM
- Midterm #1 next week, more details Monday. Review on Wednesday.

Recap

QuickSort

- Divide and Conquer sorting algorithm like MergeSort
  - All of the work for Merge Sort happened during the combine step
  - QuickSort attempts to move the work to the divide step
- **Divide:** Move small elements to the left, and big elements to the right
- **Conquer:** Recursively call QuickSort on left and right halves
- **Combine:** …nothing

Recap

QuickSort

- Divide and Conquer sorting algorithm like MergeSort
  - All of the work for Merge Sort happened during the combine step
  - QuickSort attempts to move the work to the divide step
- **Divide:** Move small elements to the left, and big elements to the right
- **Conquer:** Recursively call QuickSort on left and right halves
- **Combine:** …nothing

QuickSort Review

Divide: Move *small* elements to the left and *big* elements to the right

How do we define what is *big* and what is *small*?

QuickSort Review

Divide: Move *small* elements to the left and *big* elements to the right

How do we define what is *big* and what is *small*?

Pick a pivot value

QuickSort Review

Divide: Move *small* elements to the left and *big* elements to the right

How do we define what is *big* and what is *small*?

Pick a pivot value

[smaller than pivot], pivot, [larger than pivot]

QuickSort Review

Divide: Move *small* elements to the left and *big* elements to the right

How do we define what is *big* and what is *small*?

Pick a pivot value

[smaller than pivot], pivot, [larger than pivot]

How do we pick a pivot?

QuickSort Review

If our pivot was the median value, then our list would be split in half by the divide step, resulting in the same structure as MergeSort...

…but finding the median value is expensive…(it costs ***n*log(*n*)**).

So what if we pick one randomly instead?

QuickSort Review

Picking a pivot value randomly from the n elements of our sequence is the same as rolling an n-sided die.

There is a $1/n$ probability in any particular value being selected.

$X = k$ means that X is the kth largest value, and the expected value of X corresponds to the median value.

QuickSort Review

$$T(n) = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ T(0) + T(n-1) + \Theta(n) & \textbf{if } n > 1 \wedge X = 1 \\ T(1) + T(n-2) + \Theta(n) & \textbf{if } n > 1 \wedge X = 2 \\ T(2) + T(n-3) + \Theta(n) & \textbf{if } n > 1 \wedge X = 3 \\ .. & \\ T(n-2) + T(1) + \Theta(n) & \textbf{if } n > 1 \wedge X = n-1 \\ T(n-1) + T(0) + \Theta(n) & \textbf{if } n > 1 \wedge X = n \end{cases}$$

QuickSort Review

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ E[T(X-1) + T(n-X)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

QuickSort Review

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ 2E[T(X-1)] + \Theta(n) & \textbf{otherwise} \end{cases}$$

QuickSort Review

$$E[T(n)] = \begin{cases} \Theta(1) & \textbf{if } n \leq 1 \\ \frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + \Theta(n) & \textbf{otherwise} \end{cases}$$

Back to Induction

Hypothesis: $E[T(n)] \in O(n \log(n))$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$\cancel{2} \cdot (T(0)/\cancel{2} + T(1)/\cancel{2}) + 2c_1 \leq 2c$$

$$T(0) + T(1) + 2c_1 \leq 2c$$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

$$T(0) + T(1) + 2c_1 \leq 2c$$

$$2c_0 + 2c_1 \leq 2c$$

Base Case

Base Case: $E[T(2)] \leq c\ (2 \log(2))$

$$2 \cdot E_i[T(i - 1)] + 2c_1 \leq 2c$$

$$2 \cdot (T(0)/2 + T(1)/2) + 2c_1 \leq 2c$$

$$T(0) + T(1) + 2c_1 \leq 2c$$

$$2c_0 + 2c_1 \leq 2c$$

True for any $c \geq c_0 + c_1$

Inductive Case

Assume: $E[T(n')] \leq c\ (n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\ (n \log(n))$

Inductive Case

Assume: $E[T(n')] \leq c\ (n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\ (n \log(n))$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

Inductive Case

Assume: $E[T(n')] \leq c\ (n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\ (n \log(n))$

Our *i* here is always less than *n*, so we can use our assumption to substitute

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} ci\log(i)\right) + c_1 \leq cn\log(n)$$

Inductive Case

Assume: $E[T(n')] \leq c\ (n' \log(n'))$ for **all** $n' < n$

Show: $E[T(n)] \leq c\ (n \log(n))$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} E[T(i)]\right) + c_1 \leq cn\log(n)$$

$$\frac{2}{n}\left(\sum_{i=0}^{n-1} ci\log(i)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

$$cn\log(n) - c\log(n) + c_1 \leq cn\log(n)$$

Inductive Case

$$c\frac{2}{n}\left(\sum_{i=0}^{n-1} i\log(n)\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\sum_{i=0}^{n-1} i\right) + c_1 \leq cn\log(n)$$

$$c\frac{2\log(n)}{n}\left(\frac{(n-1)(n-1+1)}{2}\right) + c_1 \leq cn\log(n)$$

$$c\frac{\log(n)}{n}\left(n^2 - n\right) + c_1 \leq cn\log(n)$$

$$cn\log(n) - c\log(n) + c_1 \leq cn\log(n)$$

$$c_1 \leq c\log(n)$$

QuickSort

So…is QuickSort $O(n \log(n))$...?

No!

What guarantees do you get?

If *f*(*n*) is a Tight Bound

The algorithm always runs in $cf(n)$ steps

If *f*(*n*) is a Worst-Case Bound

The algorithm always runs in at most $cf(n)$

If *f*(*n*) is an Amortized Worst-Case Bound

n invocations of the algorithm **always** run in $cnf(n)$ steps

If *f*(*n*) is an Average Bound

…we don't have any guarantees

Stacks

Represents a stack of objects on top of one another

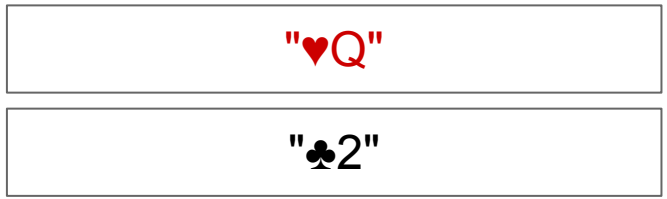
```
1  public class Stack<E> {
2
3    public void push(E value); // Add value to the "top" of the stack
4
5    public E pop();   // Remove and return the top of the stack
6
7    public E peek();  // Return the top of the stack
8
9  }
```

Stacks

```
s.push("♣2")
```

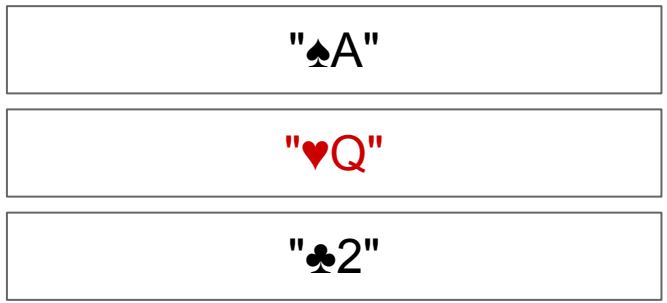
Stacks

```
s.push("♣2")
s.push("❤️Q")
```



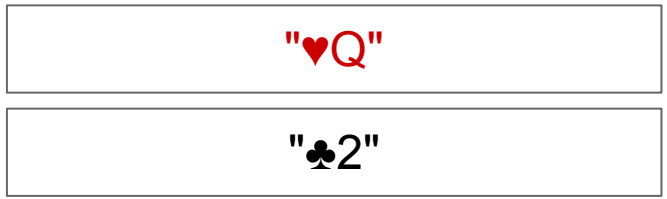
Stacks

```
s.push("♣2")
s.push("❤️Q")
s.push("♠A")
```



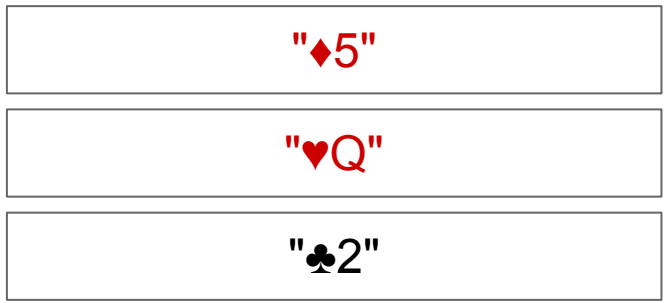
Stacks

```
s.push("♣2")

s.push("❤️Q")

s.push("♠A")

card = s.pop() // Removes "♠A"
```



Stacks

```
s.push("♣2")
s.push("❤Q")
s.push("♠A")
card = s.pop() // Removes "♠A"
s.push("♦5")
```



Stacks in Practice

- Storing function variables in a "call stack"
- Certain types of parsers ("context free")
- Backtracking search
- Reversing Sequences

Stacks - LinkedList Implementation

```
1  public class ListStack<E> extends Stack<E> {
2    private LinkedList<E> data;
3
4    public void push(E value) {
5      /* ?? */
6    }
7    public E pop() {
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - LinkedList Implementation

```
1  public class ListStack<E> extends Stack<E> {
2    private LinkedList<E> data;
3
4    public void push(E value) {
5      data.add(0, value);
6    }
7    public E pop() {
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - LinkedList Implementation

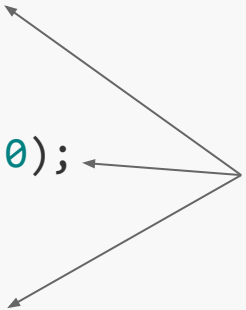
```
1  public class ListStack<E> extends Stack<E> {
2    private LinkedList<E> data;
3
4    public void push(E value) {
5      data.add(0, value);
6    }
7    public E pop() {
8      return data.remove(0);
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - LinkedList Implementation

```
1  public class ListStack<E> extends Stack<E> {
2    private LinkedList<E> data;
3
4    public void push(E value) {
5      data.add(0, value);
6    }
7    public E pop() {
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```

Stacks - LinkedList Implementation

```
1  public class ListStack<E> extends Stack<E> {
2    private LinkedList<E> data;
3
4    public void push(E value) {
5      data.add(0, value);
6    }
7    public E pop() {
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```



Θ(1) complexity in all cases

…and only requires a singly linked list

Stacks - ArrayList Implementation

```
1  public class ArrayStack<E> extends Stack<E> {
2    private ArrayList<E> data;
3
4    public void push(E value) {
5      /* ?? */
6    }
7    public E pop() {
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - ArrayList Implementation

```
1  public class ArrayStack<E> extends Stack<E> {
2    private ArrayList<E> data;
3
4    public void push(E value) {
5      data.add(value);
6    }
7    public E pop() {
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - ArrayList Implementation

```
1  public class ArrayStack<E> extends Stack<E> {
2    private ArrayList<E> data;
3
4    public void push(E value) {
5      data.add(value);
6    }
7    public E pop() {
8      return data.remove(data.size() - 1);
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Stacks - ArrayList Implementation

```
1  public class ArrayStack<E> extends Stack<E> {
2    private ArrayList<E> data;
3
4    public void push(E value) {
5      data.add(value);
6    }
7    public E pop() {
8      return data.remove(data.size() - 1);
9    }
10   public E peek() {
11     return data.get(data.size() - 1);
12   }
13 }
```

Stacks - ArrayList Implementation

```
1  public class ArrayStack<E> extends Stack<E> {
2    private ArrayList<E> data;
3
4    public void push(E value) {
5      data.add(value);
6    }
7    public E pop() {
8      return data.remove(data.size() - 1);
9    }
10   public E peek() {
11     return data.get(data.size() - 1);
12   }
13 }
```

***O*(*n*)**, Amortized **Θ(1)**

Θ(1)

Stacks in Java

Java's **`Stack`** implementation is based on **`Vector`**

(Like **`ArrayList`** but increases capacity by a constant, rather than doubling)

What does this mean for runtime of **`push`** in Java specifically?

Stacks in Java

Java's **`Stack`** implementation is based on **`Vector`**

(Like **`ArrayList`** but increases capacity by a constant, rather than doubling)

What does this mean for runtime of **`push`** in Java specifically?

`push` has a runtime of $O(n)$, amortized $O(n)$ ← n calls to push take $O(n^2)$

Stacks in Java

Java's **`Stack`** implementation is based on **`Vector`**

(Like **`ArrayList`** but increases capacity by a constant, rather than doubling)

What does this mean for runtime of **`push`** in Java specifically?

`push` has a runtime of $O(n)$, amortized $O(n)$ ← n calls to push take $O(n^2)$

A common assumption for Stacks (and Queues) is they will have a limited size. The contiguous nature of array memory usage has some benefits…

Queues

Outside of the US, "queueing" is lining up, ie at Starbucks

```
1  public class Queue<E> {
2
3    public void add(E value); // Add value to the "back" of the queue
4
5    public E remove();   // Remove and return the front of the queue
6
7    public E peek();  // Return the front of the stack
8
9  }
```

Queues

Outside of the US, "queueing" is lining up, ie at Starbucks

```
1  public class Queue<E> {
2
3    public void add(E value); //                              ue
4
5    public E remove();    // Remo                            e
6
7    public E peek();  // Return the front of the stack
8
9  }
```

In context of queues we will often refer to add/remove as enqueue/dequeue

Queues

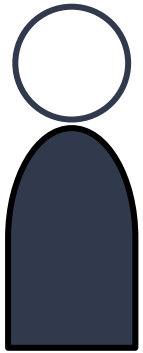
Front

Back

Queues

```
enqueue("Michael")
```

Front

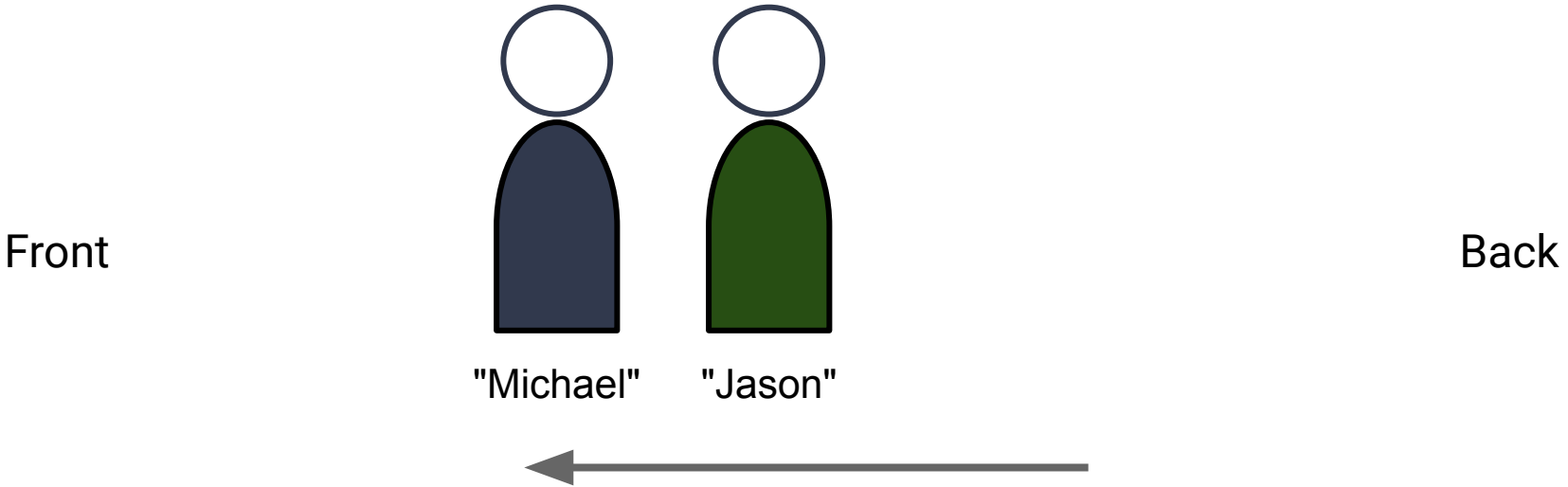


"Michael"

Back

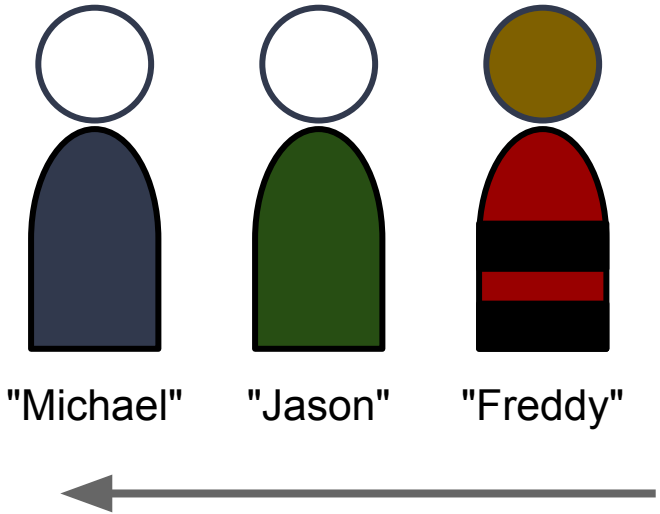
Queues

```
enqueue("Michael")
enqueue("Jason")
```



Queues

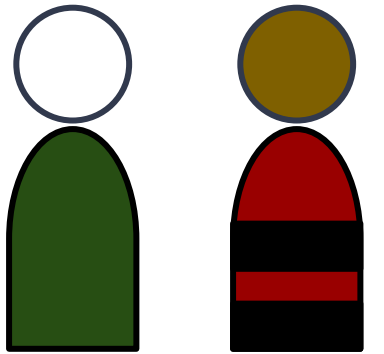
```
enqueue("Michael")
enqueue("Jason")
enqueue("Freddy")
```



Queues

```
enqueue("Michael")
enqueue("Jason")
enqueue("Freddy")
dequeue()
```

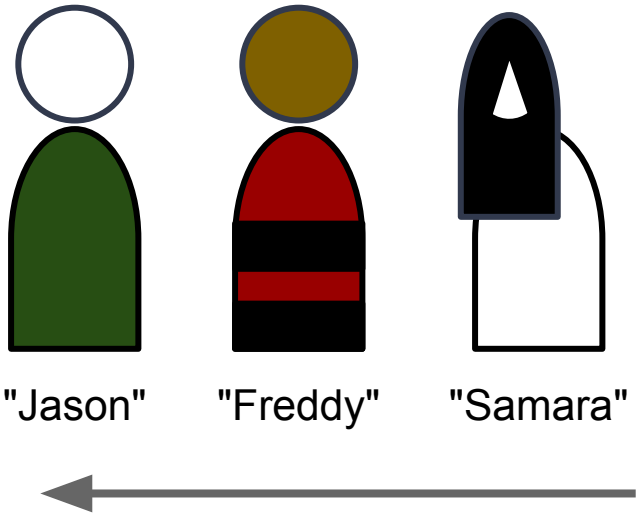
Front



Back

Queues

```
enqueue("Michael")
enqueue("Jason")
enqueue("Freddy")
dequeue()
enqueue("Samara")
```



Queues vs Stacks

Queue First in, First Out (FIFO)

Stacks Last in, First Out (LIFO / FILO)

Queues in Practice

- Delivering network packets, emails, twitter/tiktok/instagram
- Scheduling CPU cycles
- Deferring long-running tasks

Queues - LinkedList Implementation

```
1  public class ListQueue<E> extends Queue<E> {
2    private LinkedList<E> data;
3
4    public void add(E value) { // enqueue
5      /* ?? */
6    }
7    public E remove() { // dequeue
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Queues - LinkedList Implementation

```
1  public class ListQueue<E> extends Queue<E> {
2    private LinkedList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      /* ?? */
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Queues - LinkedList Implementation

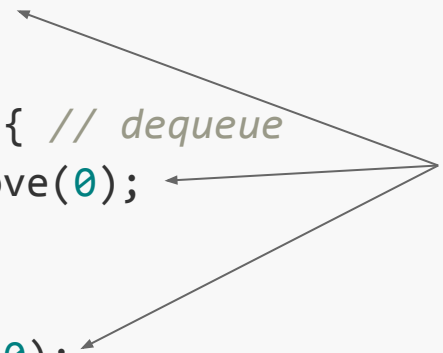
```
1  public class ListQueue<E> extends Queue<E> {
2    private LinkedList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      return data.remove(0);
9    }
10   public E peek() {
11     /* ?? */
12   }
13 }
```

Queues - LinkedList Implementation

```
1  public class ListQueue<E> extends Queue<E> {
2    private LinkedList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```

Queues - LinkedList Implementation

```
1  public class ListQueue<E> extends Queue<E> {
2    private LinkedList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```



Θ(1) complexity in all cases as long as we have a reference to the last node in the list

Queues

Thought Experiment: How can we use an array to build a queue?

Queues - ArrayList Implementation

```
1  public class ArrayQueue<E> extends Queue<E> {
2    private ArrayList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```

Queues - ArrayList Implementation

```
1  public class ArrayQueue<E> extends Queue<E> {
2    private ArrayList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(value);
6    }
7    public E remove() { // dequeue
8      return data.remove(0);
9    }
10   public E peek() {
11     return data.get(0);
12   }
13 }
```

Amortized $\Theta(1)$

$\Theta(n)$:(

$\Theta(1)$

Queues - ArrayList Implementation

```
1  public class ArrayQueue<E> extends Queue<E> {
2    private ArrayList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(0, value);
6    }
7    public E remove() { // dequeue
8      return data.remove(data.size() - 1);
9    }
10   public E peek() {
11     return data.get(data.size() - 1);
12   }
13 }
```

Queues - ArrayList Implementation

```
1  public class ArrayQueue<E> extends Queue<E> {
2    private ArrayList<E> data;
3
4    public void add(E value) { // enqueue
5      data.add(0, value);
6    }
7    public E remove() { // dequeue
8      return data.remove(data.size() - 1);
9    }
10   public E peek() {
11     return data.get(data.size() - 1);
12   }
13 }
```

Θ(*n*) :(

Θ(1)

Θ(1)

Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

Why didn't we have to pay that cost with a list?

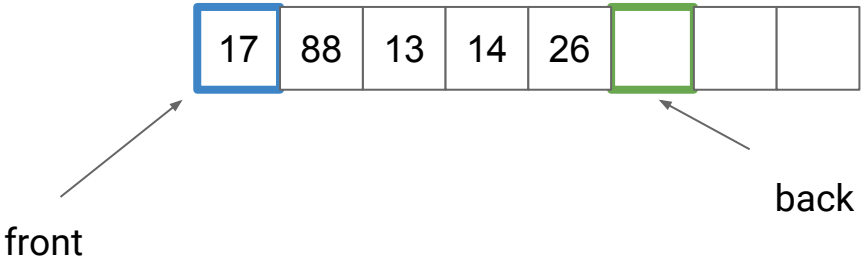
Queues

Can we avoid the cost of moving all of the elements forward or backward each time we add or remove?

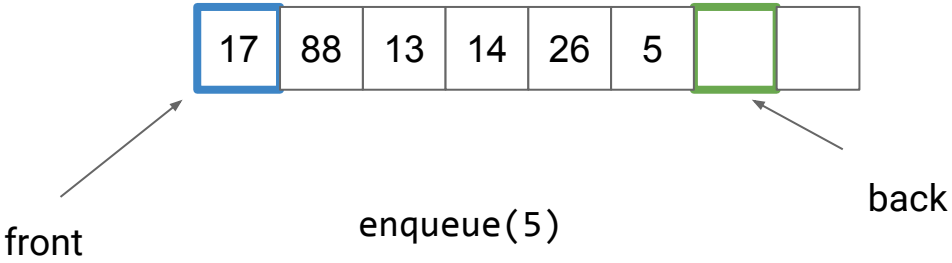
Why didn't we have to pay that cost with a list?

Update our values of "first" and "last"!

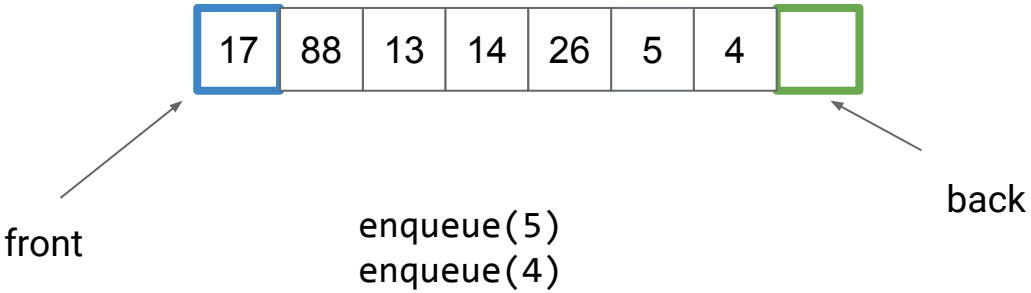
Queues



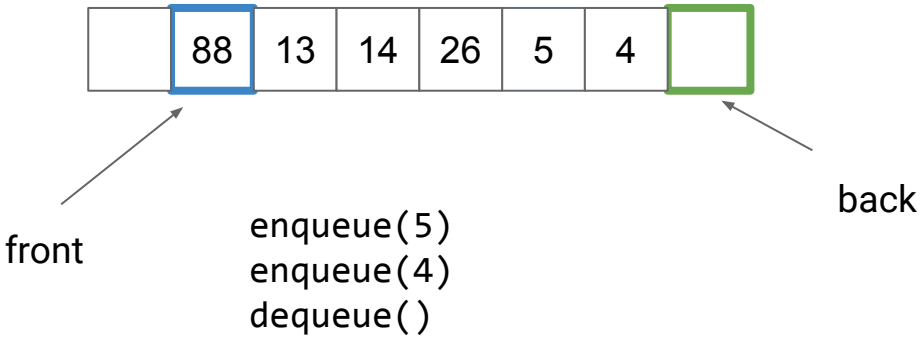
Queues



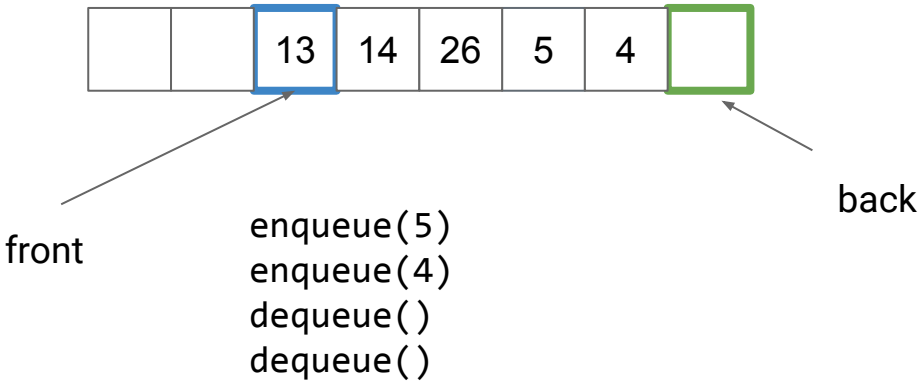
Queues



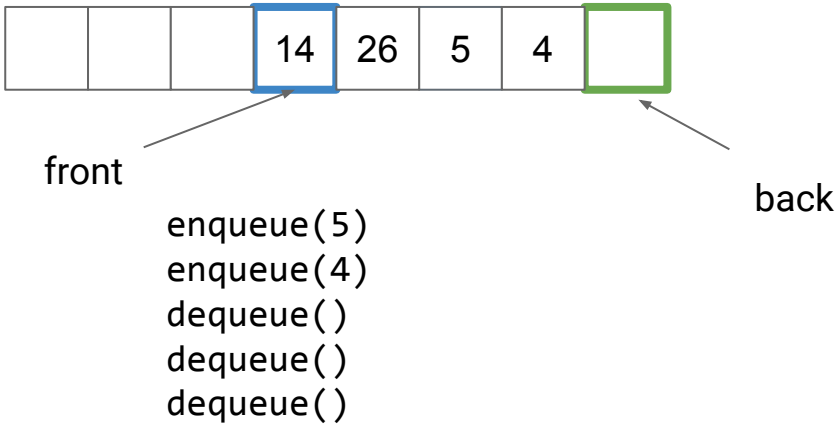
Queues



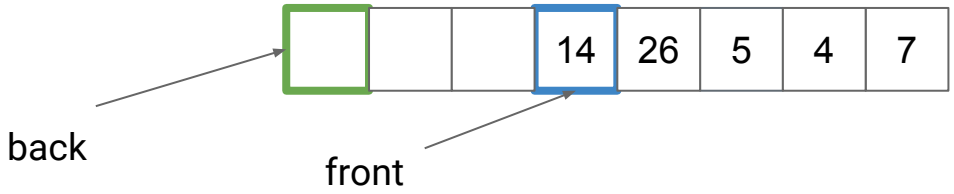
Queues



Queues

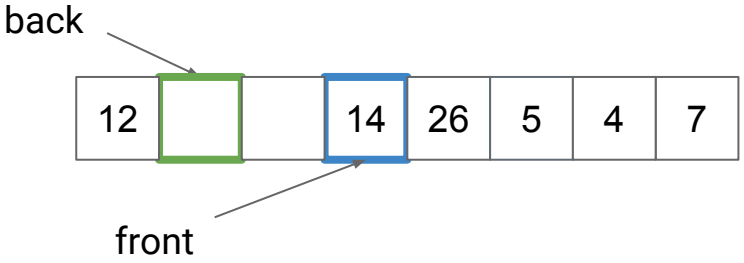

14
26
5
4
front
back
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()

Queues



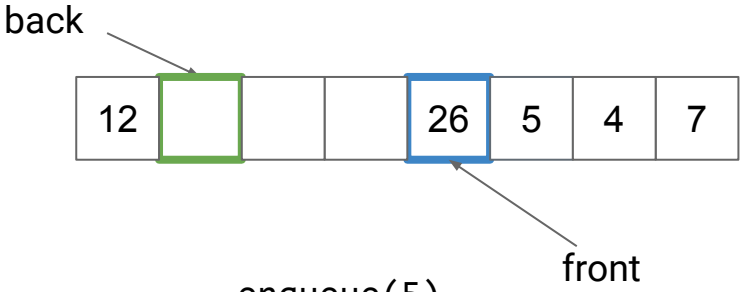

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
```

Queues



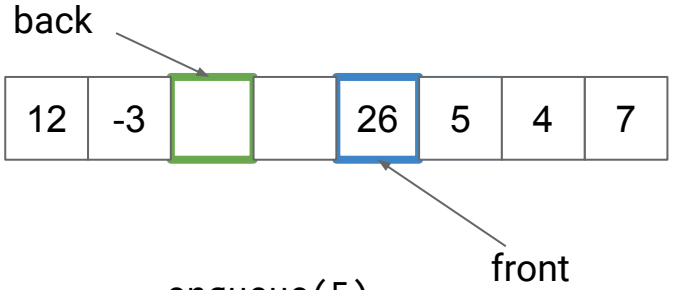

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
```

Queues




```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
```

Queues




```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
dequeue()
enqueue(-3)
```

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue

1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue

1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue

1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue

1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

What is the complexity?

ArrayDeque (Resizable Ring Buffer)

Active Array = [start, end)

Enqueue Amortized $\Theta(1)$

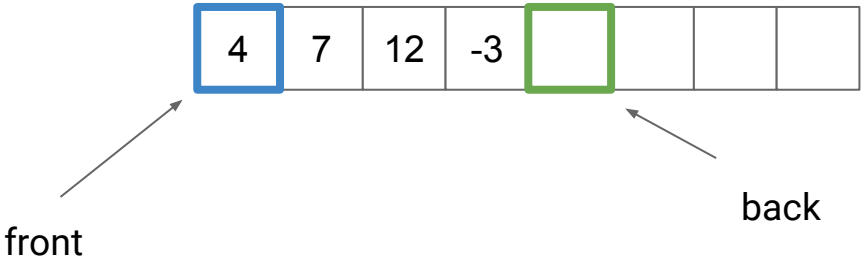
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue $\Theta(1)$

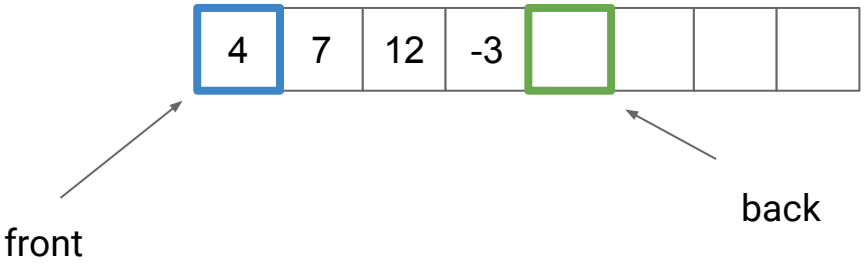
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

What is the complexity?

Why Ring Buffer?

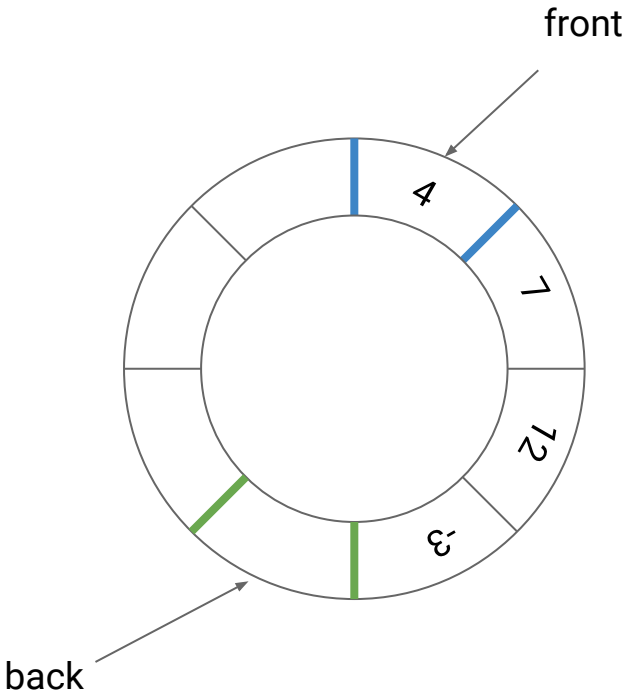


Why Ring Buffer?

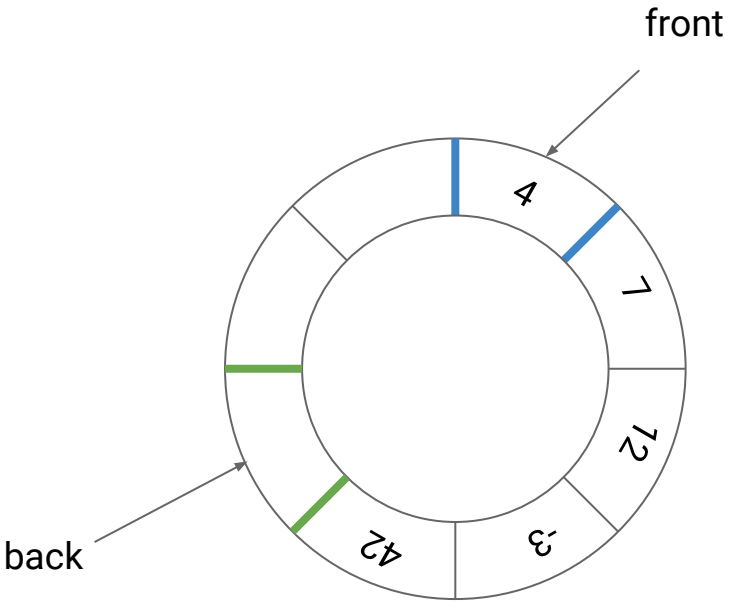


Conceptually, we can think of this as a ring…

Why Ring Buffer?

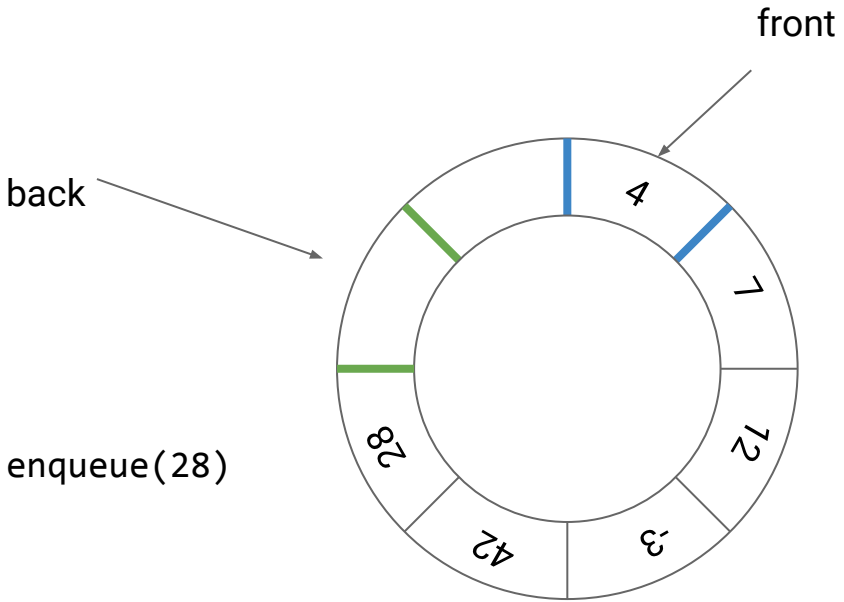


Why Ring Buffer?

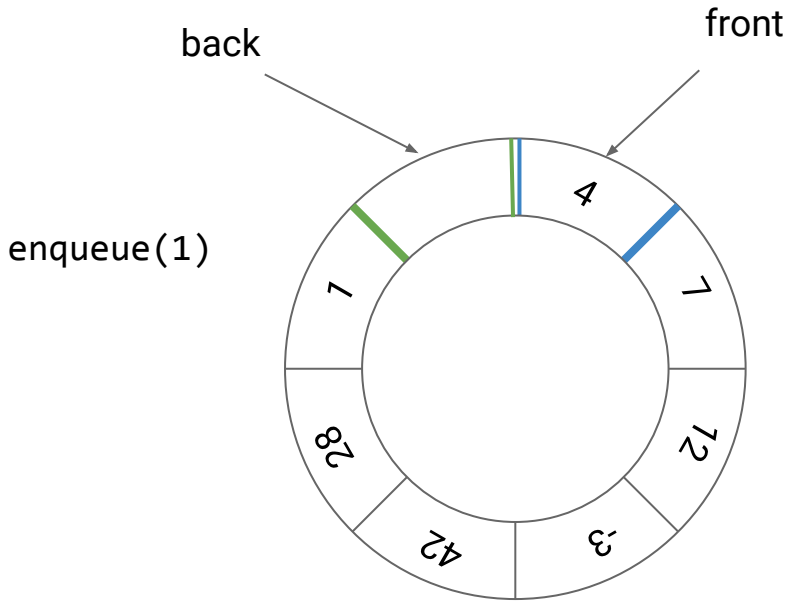


enqueue(42)

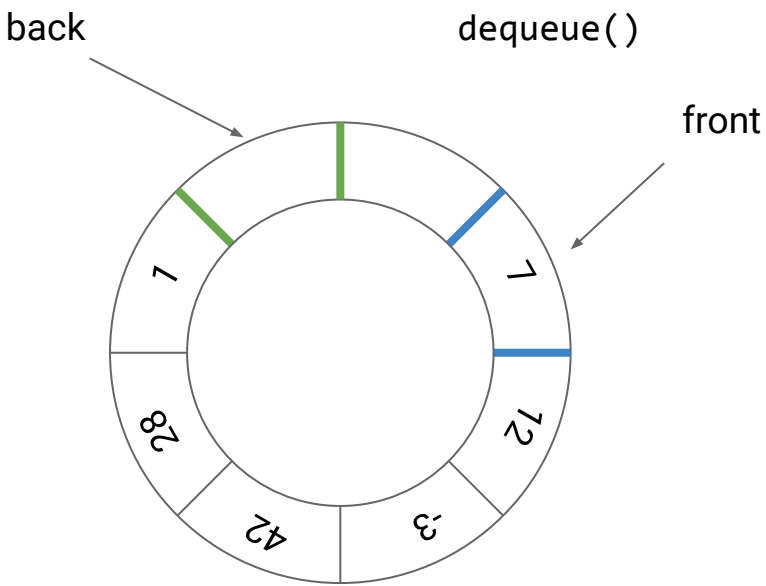
Why Ring Buffer?



Why Ring Buffer?



Why Ring Buffer?



Why Ring Buffer?