CSE 250
Data Structures

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

Lec 17: Stacks and Queues

Announcements

- WA3 due Sunday @ 11:59PM

New ADT: Stacks

Represents a stack of objects on top of one another

```java
public class Stack<E> {

  public void push(E value); // Add value to the "top" of the stack

  public E pop();   // Remove and return the top of the stack

  public E peek();  // Return the top of the stack

}
```

Stacks

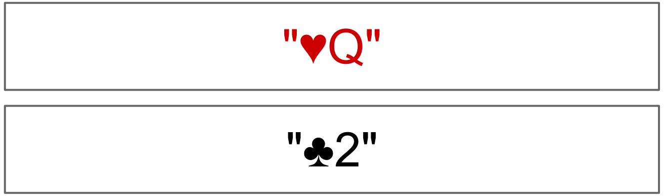
```
s.push("♣2")
```

"♣2"

Stacks

```
s.push("♣2")

s.push("❤Q")
```



Stacks

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

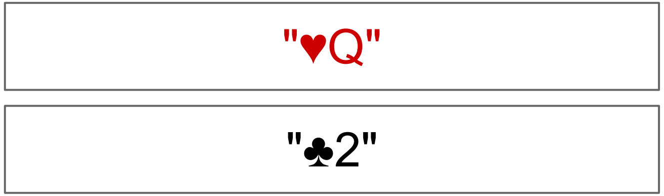
"♠A"

"♥Q"

"♣2"

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")
```

"♦5"

"♥Q"

"♣2"

Stacks in Practice

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

Stacks - LinkedList Implementation

```
public class ListStack<E> extends Stack<E> {
  private LinkedList<E> data;

  public void push(E value) {
    /* ?? */
  }
  public E pop() {
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - LinkedList Implementation

```
public class ListStack<E> extends Stack<E> {
  private LinkedList<E> data;

  public void push(E value) {
    data.add(0, value);
  }
  public E pop() {
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - LinkedList Implementation

```java
public class ListStack<E> extends Stack<E> {
  private LinkedList<E> data;

  public void push(E value) {
    data.add(0, value);
  }
  public E pop() {
    return data.remove(0);
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - LinkedList Implementation

```java
public class ListStack<E> extends Stack<E> {
  private LinkedList<E> data;

  public void push(E value) {
    data.add(0, value);
  }
  public E pop() {
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Stacks - LinkedList Implementation

```
public class ListStack<E> extends Stack<E> {
  private LinkedList<E> data;

  public void push(E value) {
    data.add(0, value);
  }
  public E pop() {
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Θ(1) complexity in all cases

…and only requires a singly linked list

Stacks - ArrayList Implementation

```java
public class ArrayStack<E> extends Stack<E> {
  private ArrayList<E> data;

  public void push(E value) {
    /* ?? */
  }
  public E pop() {
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - ArrayList Implementation

```java
public class ArrayStack<E> extends Stack<E> {
  private ArrayList<E> data;

  public void push(E value) {
    data.add(value);
  }
  public E pop() {
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - ArrayList Implementation

```java
public class ArrayStack<E> extends Stack<E> {
  private ArrayList<E> data;

  public void push(E value) {
    data.add(value);
  }
  public E pop() {
    return data.remove(data.size() - 1);
  }
  public E peek() {
    /* ?? */
  }
}
```

Stacks - ArrayList Implementation

```java
public class ArrayStack<E> extends Stack<E> {
  private ArrayList<E> data;

  public void push(E value) {
    data.add(value);
  }
  public E pop() {
    return data.remove(data.size() - 1);
  }
  public E peek() {
    return data.get(data.size() - 1);
  }
}
```

Stacks - ArrayList Implementation

```java
public class ArrayStack<E> extends Stack<E> {
  private ArrayList<E> data;

  public void push(E value) {
    data.add(value);
  }
  public E pop() {
    return data.remove(data.size() - 1);
  }
  public E peek() {
    return data.get(data.size() - 1);
  }
}
```

$O(n)$, Amortized $\Theta(1)$

$\Theta(1)$

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 $O(n) \leftarrow n$ 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 $\boldsymbol{O(n)}$, amortized $\boldsymbol{O(n)}$ ← $\boldsymbol{n}$ calls to push take $\boldsymbol{O(n^2)}$

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

A New ADT: Queues

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

```
public class Queue<E> {

  public void add(E value); // Add value to the "back" of the queue

  public E remove();    // Remove and return the front of the queue

  public E peek();  // Return the front of the queue

}
```

A New ADT: Queues

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

```java
public class Queue<E> {

  public void add(E value); //

  public E remove();   // Remo

  public E peek();  // Return the front of the queue

}
```

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

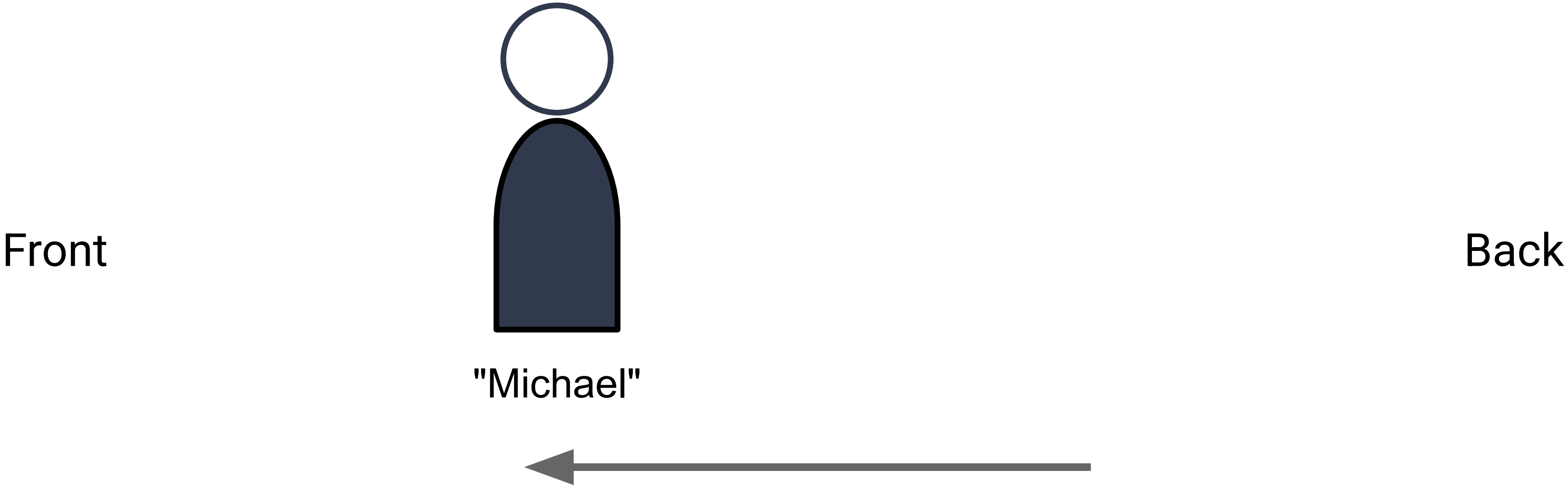
Queues

Front

Back

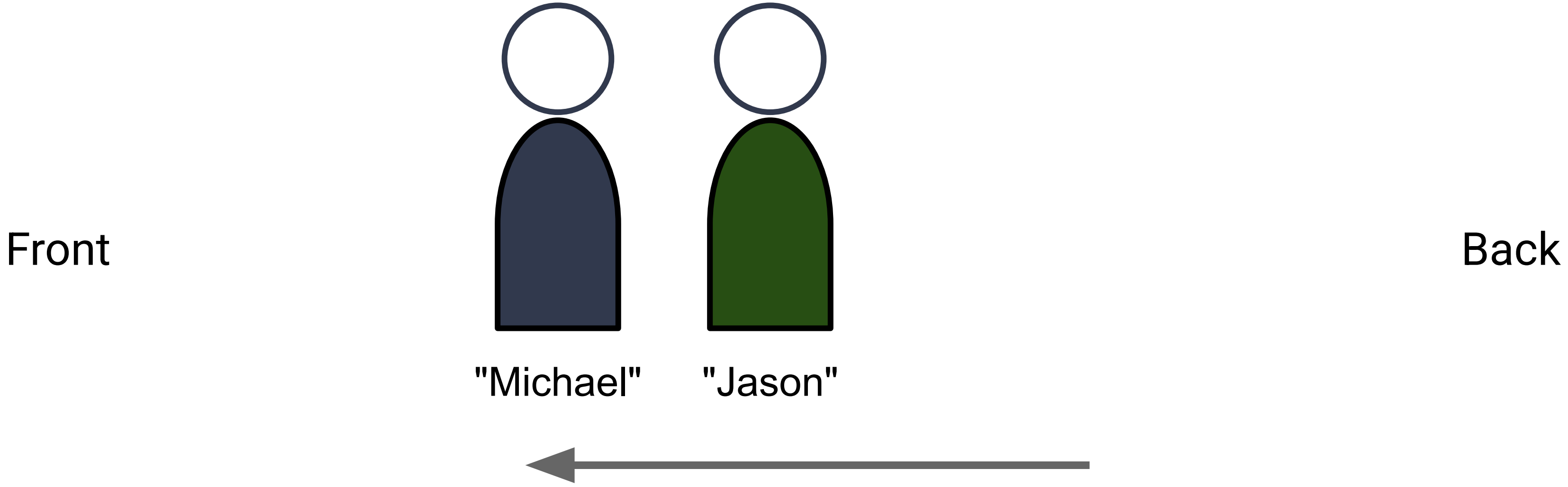
Queues

`enqueue("Michael")`



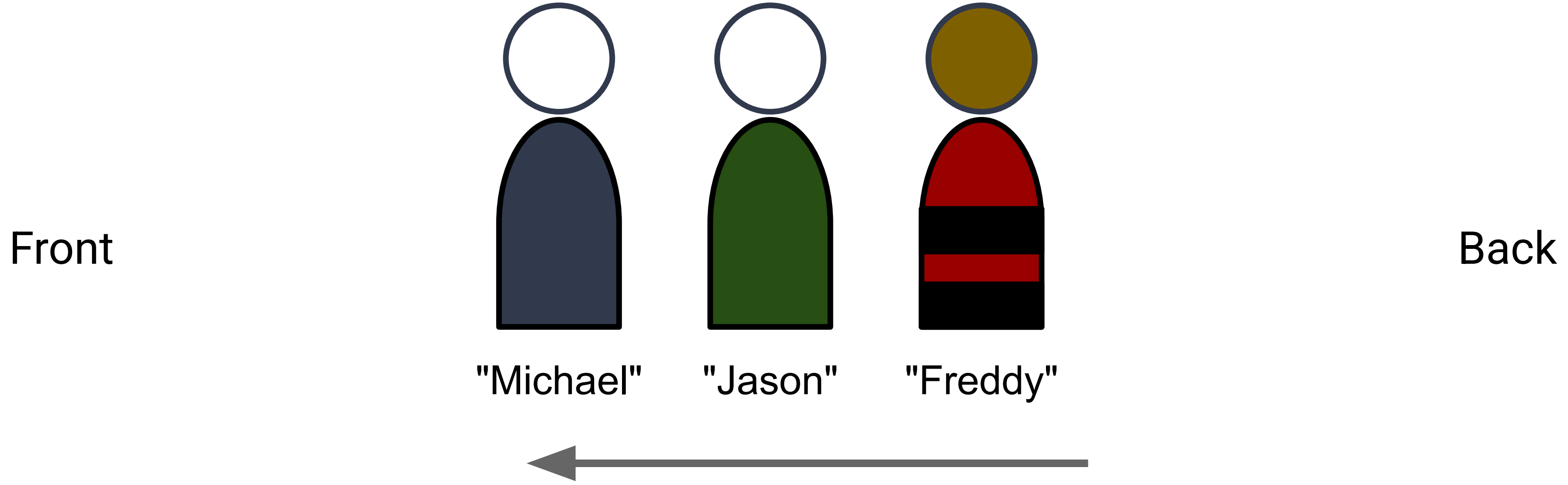
Queues

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



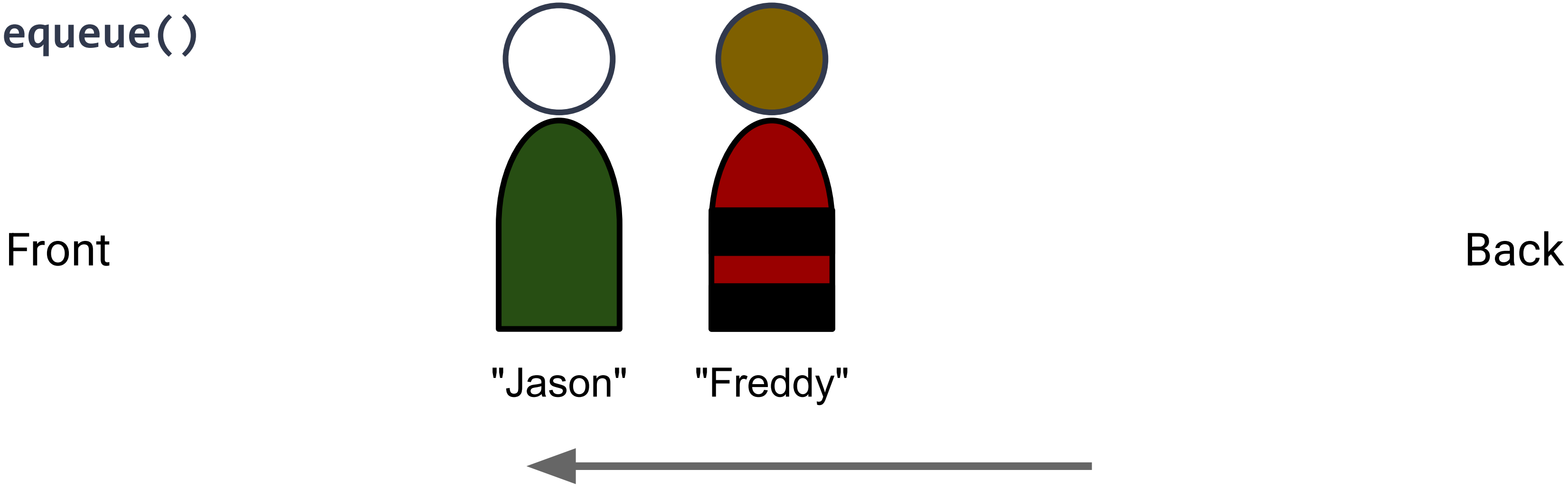
Queues

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



Queues

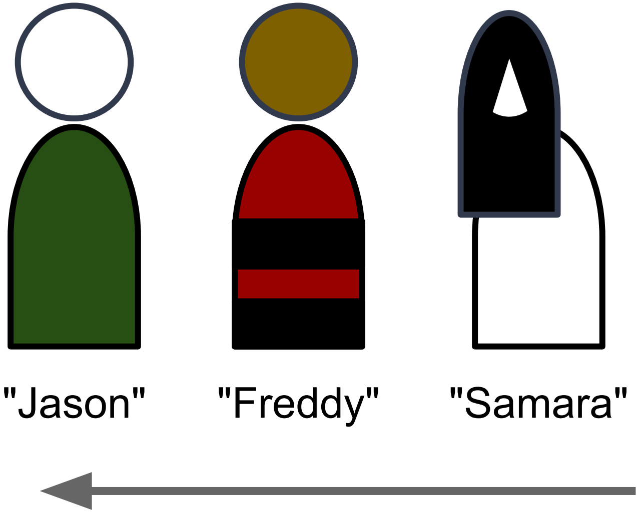
```
enqueue("Michael")
enqueue("Jason")
enqueue("Freddy")
dequeue()
```



Queues

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

Front



Back

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

```java
public class ListQueue<E> extends Queue<E> {
  private LinkedList<E> data;

  public void add(E value) { // enqueue
    /* ?? */
  }
  public E remove() { // dequeue
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Queues - LinkedList Implementation

```java
public class ListQueue<E> extends Queue<E> {
  private LinkedList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    /* ?? */
  }
  public E peek() {
    /* ?? */
  }
}
```

Queues - LinkedList Implementation

```java
public class ListQueue<E> extends Queue<E> {
  private LinkedList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    /* ?? */
  }
}
```

Queues - LinkedList Implementation

```java
public class ListQueue<E> extends Queue<E> {
  private LinkedList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Queues - LinkedList Implementation

```java
public class ListQueue<E> extends Queue<E> {
  private LinkedList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

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

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Queues - ArrayList Implementation

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(value);
  }
  public E remove() { // dequeue
    return data.remove(0);
  }
  public E peek() {
    return data.get(0);
  }
}
```

Amortized $\Theta(1)$

$\Theta(n)$:(

$\Theta(1)$

Queues - ArrayList Implementation

```java
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(0, value);
  }
  public E remove() { // dequeue
    return data.remove(data.size() - 1);
  }
  public E peek() {
    return data.get(data.size() - 1);
  }
}
```

Queues - ArrayList Implementation

```
public class ArrayQueue<E> extends Queue<E> {
  private ArrayList<E> data;

  public void add(E value) { // enqueue
    data.add(0, value);                          Θ(n) :(
  }
  public E remove() { // dequeue
    return data.remove(data.size() - 1);         Θ(1)
  }
  public E peek() {
    return data.get(data.size() - 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?

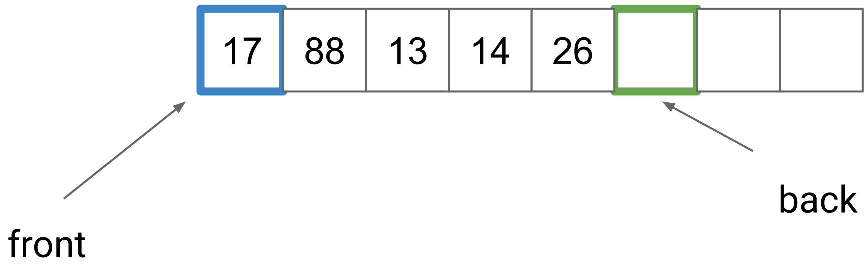
Queues

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

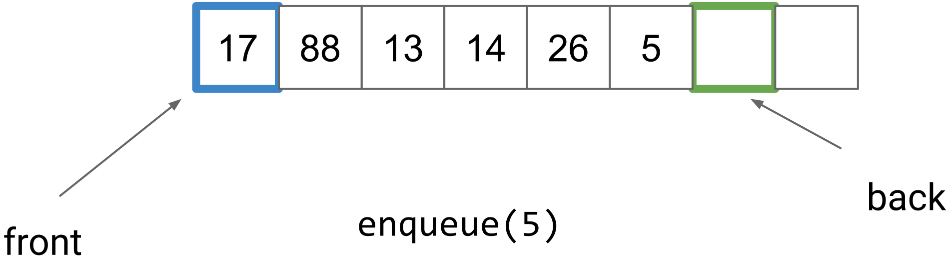
Why didn't we have to pay that cost with a list?

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

Queues



Queues

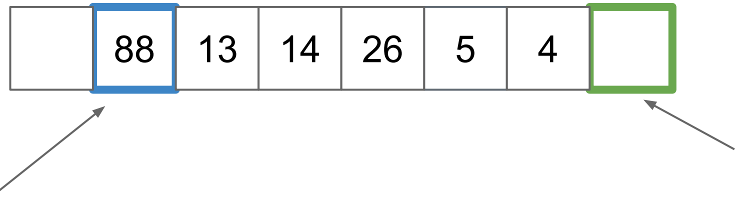


`enqueue(5)`

Queues


```
enqueue(5)
enqueue(4)
```

Queues




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

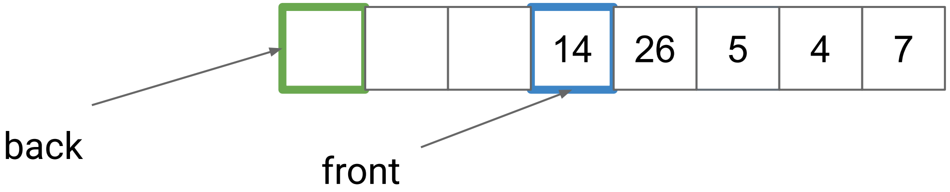
Queues


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

Queues

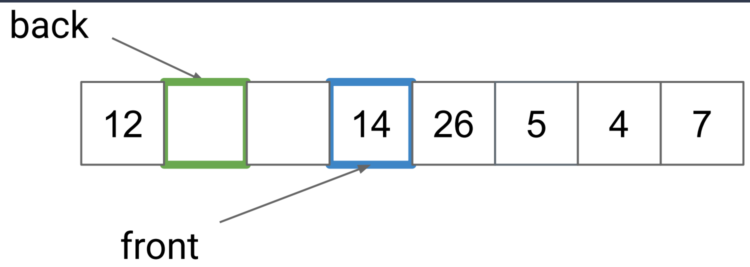

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
```

Queues



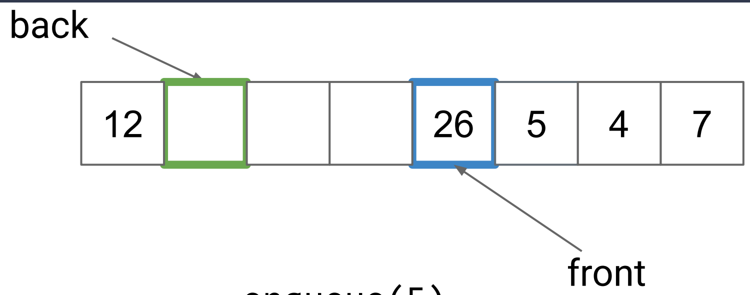

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
```

Queues



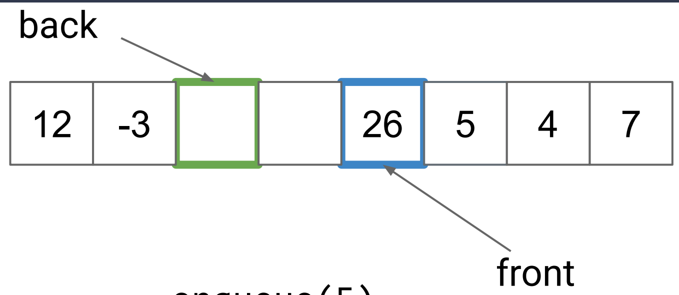

```
enqueue(5)
enqueue(4)
dequeue()
dequeue()
dequeue()
enqueue(7)
enqueue(12)
```

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 $\Theta(1)$

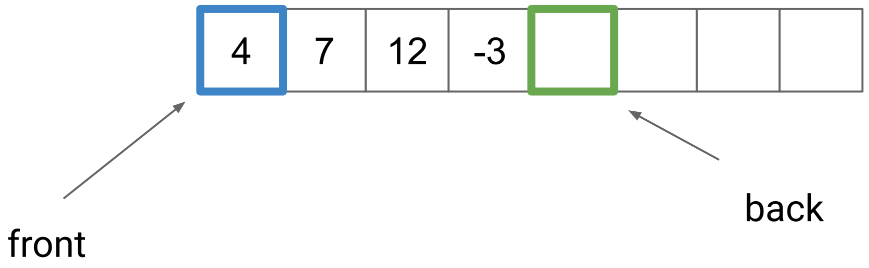
1. Resize buffer if needed
2. Add new element at buffer[end]
3. Advance end pointer (wrap to front as needed)

Dequeue $\Theta(1)$

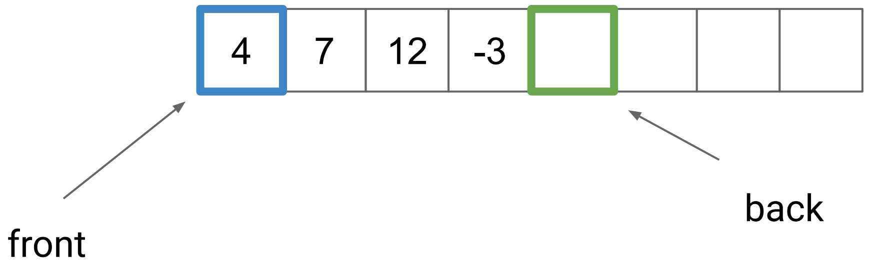
1. Remove element at buffer[start]
2. Advance start pointer (wrap to front as needed)

What is the complexity?

Why Ring Buffer?

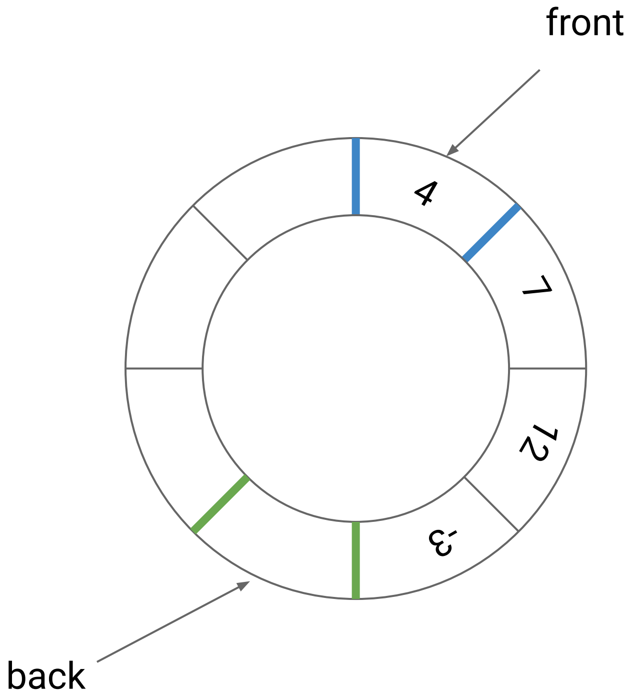


Why Ring Buffer?

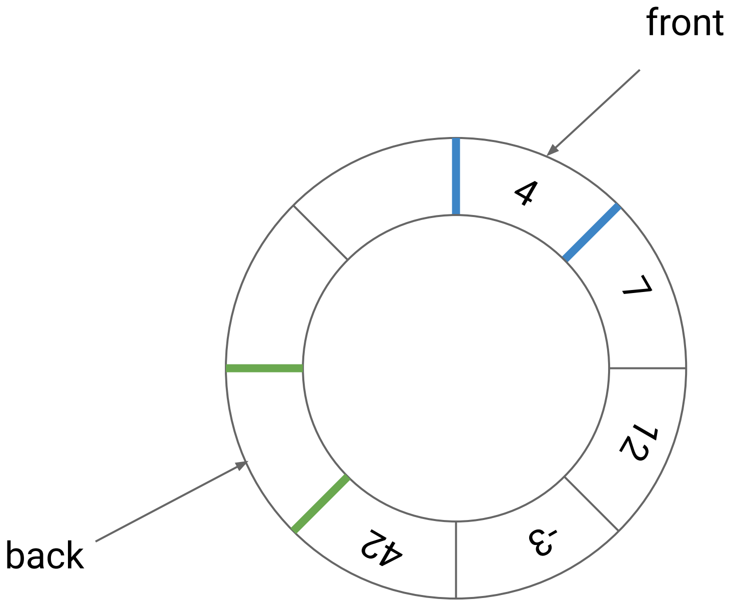


Conceptually, we can think of this as a ring…

Why Ring Buffer?

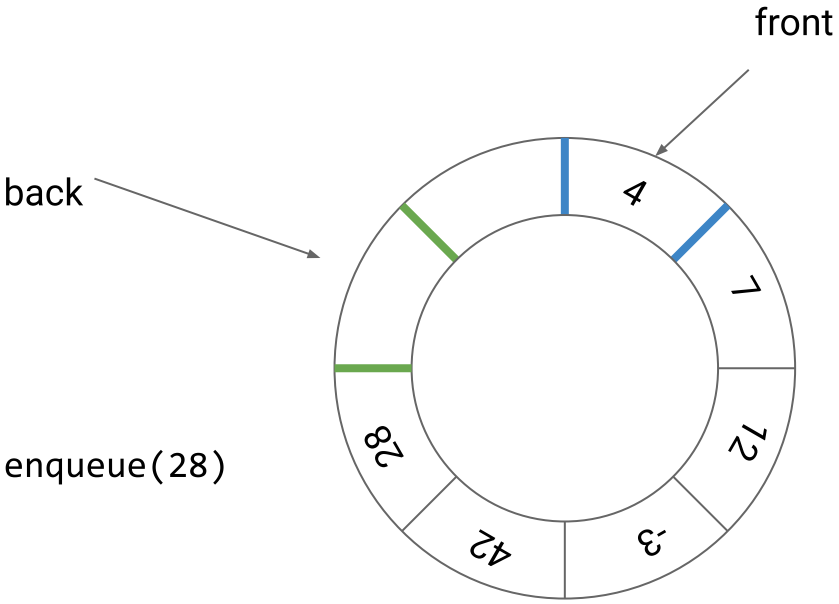


Why Ring Buffer?



`enqueue(42)`

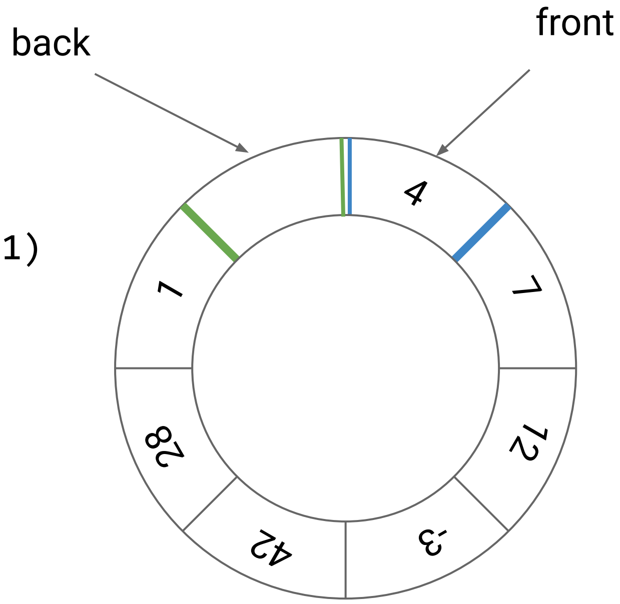
Why Ring Buffer?



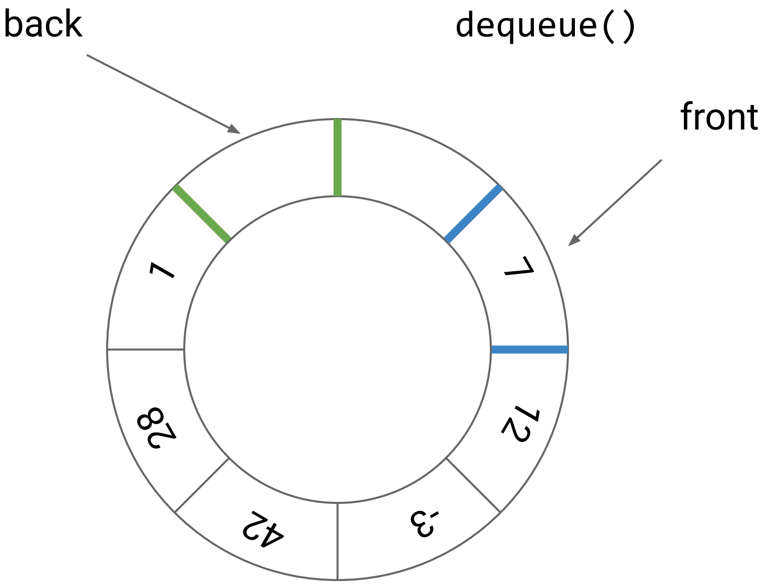
enqueue(28)

Why Ring Buffer?

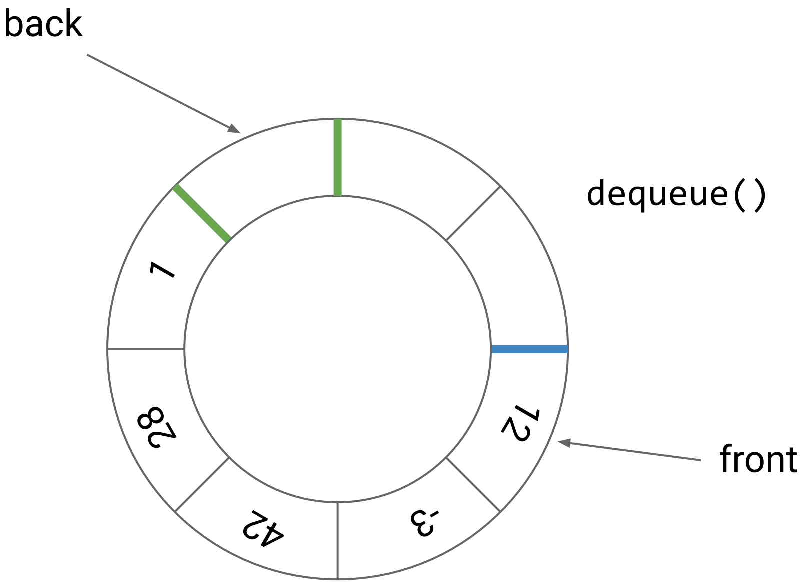
enqueue(1)



Why Ring Buffer?



Why Ring Buffer?



Applications of Stacks and Queue

Stack: Checking for balanced parentheses/braces

Queue: Scheduling packets for delivery

Both: Searching mazes

Network Packets

Router: 1gb/s internal network, 100mb/s external

- 1 gb/s sent to the router, but only 100mb/s can leave.
- How do we handle this?

Queues

- Enqueue data packets in the order they are received.
- When there is available outgoing bandwidth, dequeue and send.

Avoiding Queueing Delays

- Limit size of queue; Packets that don't fit are dropped

TCP: blocked packets are retried

UDP: application deals with dropped packets

Balanced Parentheses

What does it mean for parentheses to be balanced?

1. Every opening parenthesis is matched by a closing parenthesis

()(())   (()   ())

Balanced Parentheses

What does it mean for parentheses to be balanced?

1. Every opening parenthesis is matched by a closing parenthesis

()(())   (()   ())

✓   ✗   ✗

Balanced Parentheses

What does it mean for parentheses to be balanced?

1. Every opening parenthesis is matched by a closing parenthesis

()(()) ✓

(() ✗

()) ✗

How can we check a string for balanced parentheses?

Idea #1

Idea: Count the number of unmatched open parenthesis.

Increment counter on `(`, decrement on `)`

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

✓

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`   `{())`   `())`

✓   ✗

Balanced Parentheses/Braces

What does it mean for parentheses/braces to be balanced?

1. Every opening symbol is matched by a closing symbol
2. No nesting overlaps (ie `{(})` is not ok).

`{()({})}`	`{())`	`())`
✓	✗	✗

Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on `(` or `{`, decrement on `)` or `}`

Idea #1

Idea: Count the number of unmatched open parens/braces.

Increment counter on `(` or `{`, decrement on `)` or `}`

Problem: allows for `{(})`

Idea #2

Idea: Track nesting on a stack!

On `(` or `{`, push the symbol on the stack.

On `)` or `}`, pop the stack and check for a match.

Demo:

[https://odin.cse.buffalo.edu/teaching/cse-250/2022fa/slide/14b-QueueStackApps.html#/13]

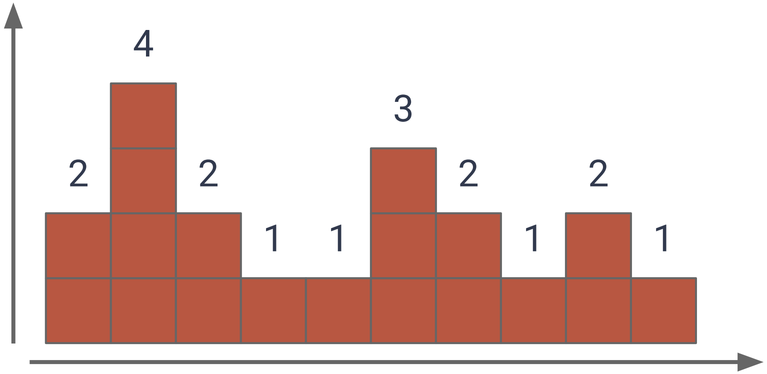
Takeaway

The stack in this solution let us "remember" what we had seen previously:

Trickier Example (Interview Question)

Imagine a bar graph as a 2D landscape that can be filled with water

Develop an algorithm to determine how much water a bar graph holds

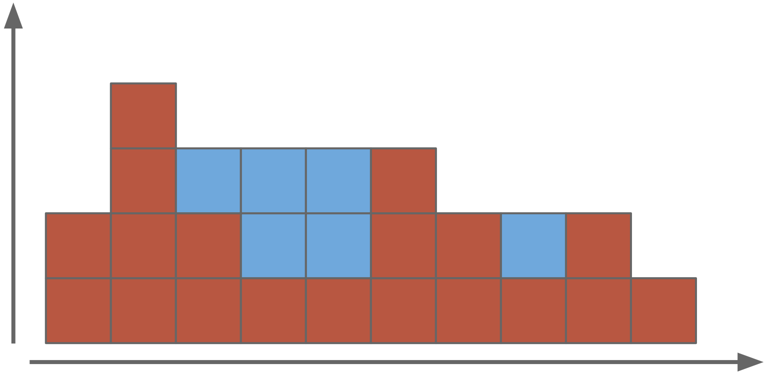


`[2,4,2,1,1,3,2,1,2,1]`

Trickier Example (Interview Question)

Imagine a bar graph as a 2D landscape that can be filled with water

Develop an algorithm to determine how much water a bar graph holds



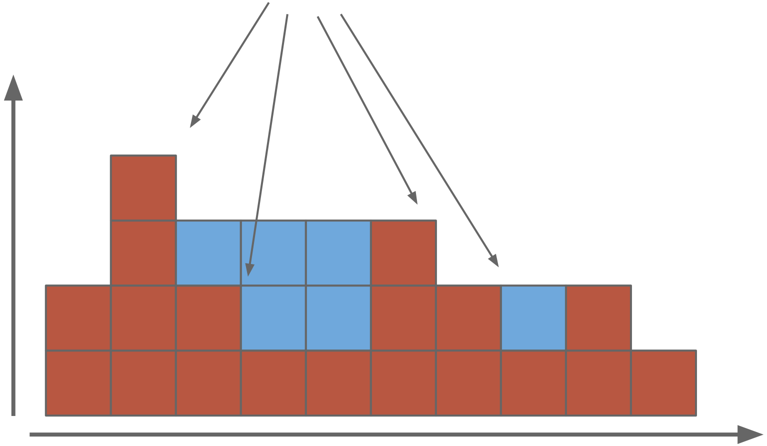
[2,4,2,1,1,3,2,1,2,1]

6 units of water

Trickier Example (Interview Question)

Basic Idea: Think of this like the brace matching problem

- Scan from left to right
- When you go **down** in height, think of that as an opening parenthesis



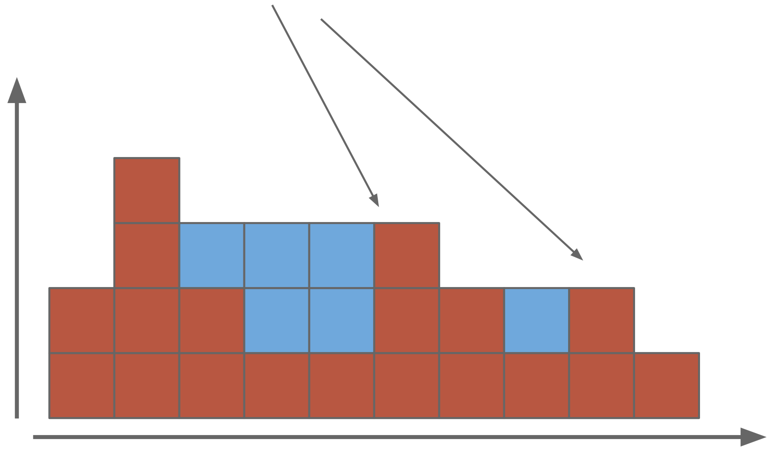
[2,4,2,1,1,3,2,1,2,1]

6 units of water

Trickier Example (Interview Question)

Basic Idea: Think of this like the brace matching problem

- Scan from left to right
- When you go **up** in height, think of that as a closing parenthesis



[2,4,2,1,1,3,2,1,2,1]

6 units of water