BEFORE WE START

Let us know in the chat:
What custom emotes should we add to the 373 Discord server?
Announcements

• Office Hours started Wednesday!
  - View office hours schedule on left panel of course website
  - Queue is run on Discord, two ways to join (separate invite links!):
    1. **Create Discord Account**
       - Enter your email
       - Stay logged in for the quarter
       - Easier to meet people and build community
    2. **Join Anonymously**
       - Temporary display name, no other info
       - Account disappears when you close window
       - Use Discord as simple, anonymous queue service; get helped over Zoom
  - Use a message to enter the queue:
    
    @TA On Duty quick question about the definition of an ADT @dubs
  - Reach out to other students while waiting!
Announcements

• Other reasons to join Discord:
  - #search-for-partners: find project partners, high success rate!
  - #career-prep: links & discussion for technical interviews, careers!
  - More? Let us know your ideas

• Project 0 (CSE 143 Review) due next Wednesday 10/07 11:59pm

• Project 1 (Deques) comes out that same day
  - Three options for projects:
    - Choose a partner – someone you know or meet in the class (#search-for-partners or Ed)
    - Join the partner pool – we’ll assign you a partner
      - Will send info about this early next week!
    - Opt to work alone – not recommended, but available
Survey: What are you currently thinking of for partner projects?

This doesn’t mean you have to commit to your answer, but we are trying to get a sense of where people are. Select which of these options best describes your thoughts.

• You already have a partner decided.
• You want to find a partner in the class on your own.
• You want to join the partner pool and have us assign you a partner.
• You want to work alone.
• Not sure yet!
Lecture Outline

• The Stack ADT

• The Queue ADT

• Design Decisions

• The Map ADT
Learning Objectives

After this lecture, you should be able to...

1. **(143 Review)** Describe the state and behavior for the Stack, Queue, and Map ADTs

2. Describe how a resizable array or linked nodes could be used to implement Stack, Queue, or Map

3. Compare the runtime of Stack, Queue, and Map operations on a resizable array vs. linked nodes, based on how they’re implemented

4. Identify invariants for the data structures we’ve seen so far
143 Review  The Stack ADT

- **Stack**: an ADT representing an ordered sequence of elements whose elements can only be added & removed from one end.
  - Last-In, First-Out (LIFO)
  - Elements stored in order of insertion
    - We don’t think of them as having indices
  - Clients can only add/remove/examine the “top”

### STACK ADT

**State**
- Collection of ordered items
- Count of items

**Behavior**
- `push(index)` add item to top
- `pop()` return & remove item at top
- `peek()` return item at top
- `size()` count of items
- `isEmpty()` is count 0?

```
<table>
<thead>
<tr>
<th>push</th>
<th>pop, peek</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
```
# Implementing a Stack with Linked Nodes

## STACK ADT

**State**
- Collection of ordered items
- Count of items

**Behavior**
- `push(index)` add item to top
- `pop()` return & remove item at top
- `peek()` return item at top
- `size()` count of items
- `isEmpty()` is count 0?

## LinkedStack\(<E>\)

**State**
- Node top
- size

**Behavior**
- `push` add new node at top
- `pop` return & remove node at top
- `peek` return node at top
- `size` return size
- `isEmpty` return size == 0

### Big-Oh Analysis

- `pop()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `push()` O(1) Constant

---

```
push(3)
push(4)
pop()
```

```
size = 2
```
STACK ADT

State
- Collection of ordered items
- Count of items

Behavior
- `push(index)` add item to top
- `pop()` return & remove item at top
- `peek()` return item at top
- `size()` count of items
- `isEmpty()` is count 0?

LinkedStack<E>

State
- Node top
- Size

Behavior
- `push()` add new node at top
- `pop()` return & remove node at top
- `peek()` return node at top
- `size()` return size
- `isEmpty()` return size == 0

Big-Oh Analysis

- `pop()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `push()` O(1) Constant

What do you think the worst possible runtime of `push()` could be?

`push(3)`
`push(4)`
`pop()`
Implementing a Stack with an Array

STACK ADT

State
Collection of ordered items
Count of items

Behavior
- push(index) add item to top
- pop() return & remove item at top
- peek() return item at top
- size() count of items
- isEmpty() is count 0?

ArrayStack<E>

State
- data[]
- size

Behavior
- push data[size] = value, if out of room grow data
- pop return data[size - 1], size -= 1
- peek return data[size - 1]
- size return size
- isEmpty return size == 0

Big-Oh Analysis
- pop() O(1) Constant
- peek() O(1) Constant
- size() O(1) Constant
- isEmpty() O(1) Constant
- push() O(1) Constant

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
</tbody>
</table>

push(3)
push(4)
pop()
push(5)
What do you think the worst possible runtime of `push()` could be?
Preview Why Not Decide on One?

• Big-Oh analysis of push(): **O(n) linear** if you have to resize, **O(1) constant** otherwise

• Two insights to keep in mind:
  1. Behavior is *completely* different in these two cases. Almost better not to try and analyze them both together.
  2. Big-Oh is a *tool* to describe runtime. Having to decide just one or the other would make it a less useful tool – not a complete description.
Lecture Outline

• The Stack ADT

• The Queue ADT

• Design Decisions

• The Map ADT
143 Review The Queue ADT

- **Queue**: an ADT representing an ordered sequence of elements whose elements can only be added from one end and removed from the other.
  - First-In, First-Out (FIFO)
  - Elements stored in order of insertion
    - We don’t think of them as having indices
  - Clients can only add to the “end”, and can only examine/remove at the “front”

**QUEUE ADT**

<table>
<thead>
<tr>
<th>State</th>
<th>Collection of ordered items</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Count of items</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Behavior</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>add(item)</td>
<td>add item to back</td>
</tr>
<tr>
<td>remove()</td>
<td>remove and return</td>
</tr>
<tr>
<td>peek()</td>
<td>return item at front</td>
</tr>
<tr>
<td>size()</td>
<td>return item at front</td>
</tr>
<tr>
<td>isEmpty()</td>
<td>count of items</td>
</tr>
<tr>
<td></td>
<td>count is 0?</td>
</tr>
</tbody>
</table>

```plaintext
front
```

```
back
```

add

to

remove, peek

1 2 3
## Implementing a Queue with Linked Nodes

### QUEUE ADT

**State**
- Collection of ordered items
- Count of items

**Behavior**
- `add(item)` add item to back
- `remove()` remove and return item at front
- `peek()` return item at front
- `size()` count of items
- `isEmpty()` count is 0?

### LinkedQueue<E>

**State**
- Node front
- Node back
- Size

**Behavior**
- `add()` add node to back
- `remove()` remove and return node at front
- `peek()` return node at front
- `size()` return size
- `isEmpty()` return size == 0

### Big-Oh Analysis

- `remove()` O(1) Constant
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `add()` O(1) Constant

---

### Example

- `add(5)`
- `add(8)`
- `remove()`

**Diagram:**
- Front: 5
- Back: 8
- Size: 2
Implementing a Queue with an Array (v1)

### QUEUE ADT

**State**
- Collection of ordered items
- Count of items

**Behavior**
- `add(item)` add item to back
- `remove()` remove and return item at front
- `peek()` return item at front
- `size()` count of items
- `isEmpty()` count is 0?

### ArrayQueueV1<E>

**State**
- `data[]`
- `size`

**Behavior**
- `add` - `data[size] = value`
- if out of room grow
- `remove` - return/remove at 0, shift everything
- `peek` - return node at 0
- `size` - return size
- `isEmpty` - return size == 0

**Big-Oh Analysis**
- `peek()` O(1) Constant
- `size()` O(1) Constant
- `isEmpty()` O(1) Constant
- `add()`
- `remove()`

```
add(5)
add(8)
add(9)
remove()
```
**Queue ADT**

**State**
- Collection of ordered items
- Count of items

**Behavior**
- `add(item)`: add item to back
- `remove()`: remove and return item at front
- `peek()`: return item at front
- `size()`: count of items
- `isEmpty()`: count is 0?

**ArrayQueueV1<E>**

**State**
- `data[]`
- `size`

**Behavior**
- `add(value)`: data[size] = value, if out of room grow
- `remove()`: return/remove at 0, shift everything
- `peek()`: return node at 0
- `size()`: return size
- `isEmpty()`: return size == 0

**Big-Oh Analysis**

- `peek()`: O(1) Constant
- `size()`: O(1) Constant
- `isEmpty()`: O(1) Constant
- `add()`: O(n) Linear if you have to resize, O(1) otherwise
- `remove()`: O(n) Linear

**What do you think the worst possible runtime of add() & remove() could be?**
Consider Data Structure Invariants

• **Invariant**: a property of a data structure that is always true between operations
  - true when finishing any operation, so it can be counted on to be true when starting an operation.

• ArrayQueueV1 is basically an `ArrayList`. What invariants does ArrayList have for its data array?
  - The i-th item in the list is stored in data[i]
    - Notice: serving this invariant is what slows down the operation. Could we choose a different invariant?
Implementing a Queue with an Array

Wrapping Around with “front” and “back” indices

add(7)
add(4)
add(1)
remove()
### Implementing a Queue with an Array (v2)

#### ArrayQueueV2<E>

**State**
- `data[]`, `front`, `size`, `back`

**Behavior**
- **add(value)**: `data[back] = value`, `back++`, `size++`, if out of room grow
- **remove()**: `return data[front]`, `size--`, `front++`
- **peek()**: `return data[front]`
- **size()**: `return size`
- **isEmpty()**: `return size == 0`

**Big-Oh Analysis**
- **peek()**: O(1) Constant
- **size()**: O(1) Constant
- **isEmpty()**: O(1) Constant
- **add()**: O(n) Linear if you have to resize, O(1) otherwise
- **remove()**: O(1) Constant
Lecture Outline

• The Stack ADT
• The Queue ADT
• Design Decisions
• The Map ADT
ADTs & Data Structures

• We’ve now seen that just like an ADT can be implemented by multiple data structures, a data structure can implement multiple ADTs

• But the ADT decides how it can be used
  - An ArrayList used as a List should support `get()`, but when used as a Stack should not
The Map ADT

- **Map**: an ADT representing a set of distinct keys and a collection of values, where each key is associated with one value.
  - Also known as a **dictionary**
  - If a key is already associated with something, calling `put(key, value)` replaces the old value

- A programmer’s best friend 😊
  - It’s hard to work on a big project without needing one sooner or later
  - CSE 143 introduced:
    - `Map<String, Integer> map1 = new HashMap<>();`
    - `Map<String, String> map2 = new TreeMap<>();`
Abstract Representations of Maps

• Plenty of different ways you might think about the Map ADT:

```java
{  
    "AA": 930,  
    "AF": 1530,  
    "AI": 1530  
}
```

• Be careful: remember these are still abstract! No assumption of how duplicates are actually stored
  - Doesn’t matter: implementation must match behavior of Map ADT, regardless of how it stores
Implementing a Map with an Array

**MAP ADT**

**State**
- Set of keys, Collection of values
- Count of keys

**Behavior**
- **put(key, value)** add value to collection, associated with key
- **get(key)** return value associated with key
- **containsKey(key)** return if key is associated
- **remove(key)** remove key and associated value
- **size()** return count

**ArrayMap<K, V>**

**State**
- Pair<K, V>[] data
- size

**Behavior**
- **put** find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary
- **get** scan all pairs looking for given key, return associated item if found
- **containsKey** scan all pairs, return if key is found
- **remove** scan all pairs, replace pair to be removed with last pair in collection
- **size** return count of items in dictionary

**Big-Oh Analysis – (if key is the last one looked at / not in the dictionary)**

- **put()** O(n) linear
- **get()** O(n) linear
- **containsKey()** O(n) linear
- **remove()** O(n) linear
- **size()** O(1) constant

**Big-Oh Analysis – (if the key is the first one looked at)**

- **put()** O(1) constant
- **get()** O(1) constant
- **containsKey()** O(1) constant
- **remove()** O(1) constant
- **size()** O(1) constant

```
put('b', 97)
put('e', 20)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>('a', 1)</td>
<td>('b', 97)</td>
<td>('c', 3)</td>
<td>('d', 4)</td>
<td>('e', 20)</td>
</tr>
</tbody>
</table>
```
Implementing a Map with Linked Nodes

**MAP ADT**

- **State**
  - Set of keys, Collection of values
  - Count of keys

- **Behavior**
  - `put(key, value)` add value to collection, associated with key
  - `get(key)` return value associated with key
  - `containsKey(key)` return if key is associated
  - `remove(key)` remove key and associated value
  - `size()` return count

```plaintext
containsKey('c')
get('d')
put('b', 20)
```

**LinkedMap<K, V>**

- **State**
  - `front`
  - `size`

- **Behavior**
  - `put()` if key is unused, create new with pair, add to front of list, else replace with new value
  - `get()` scan all pairs looking for given key, return associated item if found
  - `containsKey()` scan all pairs, return if key is found
  - `remove()` scan all pairs, skip pair to be removed
  - `size()` return count of items in dictionary

```plaintext
front

'a' 1 -> 'b' 20 -> 'c' 9 -> 'd' 4
```

**Big O Analysis – (if key is the last one looked at / not in the dictionary)**

- `put()` O(n) linear
- `get()` O(n) linear
- `containsKey()` O(n) linear
- `remove()` O(n) linear
- `size()` O(n) linear

**Big O Analysis – (if the key is the first one looked at)**

- `put()` O(1) constant
- `get()` O(1) constant
- `containsKey()` O(1) constant
- `remove()` O(1) constant
- `size()` O(1) constant
Consider: what if we delete size?

MAP ADT

State
- Set of keys, Collection of values
- Count of keys

Behavior
- put(key, value) add value to collection, associated with key
- get(key) return value associated with key
- containsKey(key) return if key is associated
- remove(key) remove key and associated value
- size() return count

LinkedMap<K, V>

State
- front
- size

Behavior
- put if key is unused, create new with pair, add to front of list, else replace with new value
- get scan all pairs looking for given key, return associated item if found
- containsKey scan all pairs, return if key is found
- remove scan all pairs, skip pair to be removed
- size return count of items in dictionary

Big O Analysis – (if key is the last one looked at / not in the dictionary)
- put() O(n) linear
- get() O(n) linear
- containsKey() O(n) linear
- remove() O(n) linear
- size() O(n) linear

Big O Analysis – (if the key is the first one looked at)
- put() O(1) constant
- get() O(1) constant
- containsKey() O(1) constant
- remove() O(1) constant
- size() O(n) linear

1. Is this okay? What about “Count of keys” in the ADT?
   Yes! The abstract state is still stored – just as # of nodes, not an int field

2. Would you ever do this? It only increases runtime.
   Possibly, if you care much more about storage space than runtime
Takeaways

• We’ve seen how different implementations can make a huge runtime difference on the same ADT
  - E.g. implementing Queue with a resizable array

• These ADTs & data structures may be review for you
  - Either way, the skills of determining & comparing these runtimes are the real goals! 😊

• Starting to see that analyzing runtimes isn’t as simple as 143 made it seem
  - E.g. one operation can have multiple Big-Oh complexity classes

• Hard to go further without a more thorough understanding of this Big-Oh tool
  - Next up: Algorithmic Analysis (Wednesday)!