Review

- Lists (including arrays) are very common data structures
  - Multiple versions, with pros and cons
  - So common, in fact, that other data structures are build on Lists

- This lecture focuses on two such important data structures:
  - Stacks and Queues
  - Two of the most widely used data structures in the field
  - Both are more restrictive Lists
  - Chap 4 in the book
Outline

- Queues
  - Priority Queues
    - Implementation
    - Using Java's PriorityQueue
    - Comparator
  - Stacks
    - Application of Stack: ParenChecker
    - Implementation
- Conclusion
Queues

- A queue is a first-in-first-out (FIFO) List
  - One of the most widely-used data structures
  - e.g., modern operating systems
  - e.g., simulations

- Operations
  - The item dequeued (or polled) is the oldest item in the queue
  - The item enqueued (or offered) is the youngest item in the queue
A queue is a first-in-first-out (FIFO) List!

- **head of queue**
- **tail of queue**

Diagram:
- `poll()` function
- `offer(..)` function
Java's Queue<E> Interface

- Java has a Queue<E> Interface
  - This tells you there are lots of types of queues in practice
  - [https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html](https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>Inserts the specified element in the tail of the queue, returning true if successful, and false otherwise.</td>
</tr>
<tr>
<td>public E remove()</td>
<td>Removes item at the head of the queue, and returns it. Throws java.util.NoSuchElementException if queue is empty.</td>
</tr>
<tr>
<td>public E poll()</td>
<td>Removes item at the head of the queue, and returns it. Returns null if queue is empty.</td>
</tr>
<tr>
<td>public E peek()</td>
<td>Returns item at head of the queue without removing it. Returns null if queue is empty.</td>
</tr>
<tr>
<td>public E element()</td>
<td>Returns item at head of the queue without removing it. Throws java.util.NoSuchElementException if queue is empty.</td>
</tr>
</tbody>
</table>
Usage Example

- Fun fact:
  - Java's LinkedList\(<E>\) class implements the Queue\(<E>\) interface!

- Import the class first

```java
import java.util.LinkedList;
```

- Sample usage in code pad:

```java
Queue<String> names = new LinkedList<>();
names.offer("Rich");
names.offer("Mary");
names.offer("Brian");
names.peek()
> "Rich" (String)

names.poll()
> "Rich" (String)

names.size()
> 2 (int)
```
Implementing Queue\langle E\rangle

- How can we implement queues?
  - Well, they \textit{are} just lists
  - Use one of the two we know...

- Options:
  - Arrays and ArrayList?
    - Bad idea (\textit{why?})
  - SinglyLinkedList?
    - Better idea
  - DoublyLinkedList?
    - No different than SinglyLinkedList (\textit{know why!})
Implement the QueueInt Interface

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>Inserts the specified element in the tail of the queue, returning <strong>true</strong> if successful, and <strong>false</strong> otherwise.</td>
</tr>
<tr>
<td>public E remove()</td>
<td>Removes item at the head of the queue, and returns it. Throws <strong>java.util.NoSuchElementException</strong> if queue is empty.</td>
</tr>
<tr>
<td>public E poll()</td>
<td>Removes item at the head of the queue, and returns it. Returns <strong>null</strong> if queue is empty.</td>
</tr>
<tr>
<td>public E peek()</td>
<td>Returns item at head of the queue without removing it. Returns <strong>null</strong> if queue is empty.</td>
</tr>
<tr>
<td>public E element()</td>
<td>Returns item at head of the queue without removing it. Throws <strong>java.util.NoSuchElementException</strong> if queue is empty.</td>
</tr>
</tbody>
</table>

```java
public interface QueueInt<E> {
    boolean offer(E item);
    E remove();
    E poll();
    E peek();
    E element();
}
```
If we used an ArrayList to implement a Queue

Assuming head of ArrayList is head of queue

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public boolean offer(E item)</code></td>
<td>$O(1)$ amortized</td>
</tr>
<tr>
<td><code>public E remove()</code></td>
<td>$O(n)$ -- why?</td>
</tr>
<tr>
<td><code>public E poll()</code></td>
<td>$O(n)$ -- why?</td>
</tr>
<tr>
<td><code>public E peek()</code></td>
<td>$O(1)$</td>
</tr>
<tr>
<td><code>public E element()</code></td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
If we used a Linked List to implement a Queue

- Assuming head of LinkedList is head of queue

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>public E remove()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>public E poll()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>public E peek()</td>
<td>( O(1) )</td>
</tr>
<tr>
<td>public E element()</td>
<td>( O(1) )</td>
</tr>
</tbody>
</table>

Shouldn't offer() be \( O(n) \) on a LinkedList?

Not if LinkedList has a tail reference in addition to head! (Lab!)
Application of Queues: Simulation

- Airport
  - One runway to handle all takeoffs
    - Takeoffs take 15 minutes to complete
  - Each minute, 4.7% chance a takeoff request is made
    - If request can't be honored, add it to a takeoff queue

- Simulation
  - Simulation "time" starts at minute 0
    - When a request is made, it should be timestamped
  - Each time a takeoff is performed,
    - Print a timestamped message, include wait time \((\text{takeoff\_time} - \text{request\_time})\)
  - Run for 120 minutes and after it finishes,
    - Print number of takeoffs completed
Application: Airport Simulation (Cont.)

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
<th>Request Time</th>
<th>Wait Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Takeoff completed, req_time=9, wait_time=15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>53</td>
<td>Takeoff completed, req_time=38, wait_time=15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>54</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>Takeoff completed, req_time=54, wait_time=15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>76</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>Takeoff completed, req_time=67, wait_time=17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>99</td>
<td>Takeoff completed, req_time=76, wait_time=23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>102</td>
<td>Takeoff requested</td>
<td></td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>Takeoff completed, req_time=102, wait_time=15</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Summary of Simulation

Simulated time: 120
Number of takeoffs requested: 6
Number of takeoffs serviced: 6 (100.0%)
Unserviced takeoffs: 0

AirportSim code available on course page.

Take a look before tackling Hwk 4
Outline

- Queues

- Priority Queues
  - Implementation
  - Using Java's PriorityQueue
  - Comparator

- Stacks
  - Application of Stack: ParenChecker
  - Implementation

- Conclusion
Priority Queues

- Recall: Queues are FIFO structures
- Sometimes, the arrival order shouldn't determine queue ordering
  - Next student to register for courses should be determined by standing
    - Sort by descending order of class standing
  - Next takeoff to perform should be the flight with most passengers
    - Sort by passenger size

- A *priority queue* always reorganizes contents based on some "natural ordering" of its objects
  - Sounds like Comparable
Java's PriorityQueue<E> Class

- Full API here:
  - [https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html)

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>Inserts the specified element into the queue, returning true if successful, and false otherwise.</td>
</tr>
<tr>
<td>public void clear()</td>
<td>Clears the priority queue.</td>
</tr>
<tr>
<td>public E poll()</td>
<td>Removes item at the head of the queue, and returns it. Returns null if queue is empty.</td>
</tr>
<tr>
<td>public E peek()</td>
<td>Returns item at head of the queue without removing it. Returns null if queue is empty.</td>
</tr>
<tr>
<td>public int size()</td>
<td>Returns the size of this priority queue.</td>
</tr>
</tbody>
</table>

(Wait, this API is just like Queue<E>... how is a different ordering preserved?)
Giving a Natural Ordering to Objects

- Seems like a familiar concept...
  - Hey we've had to do this before to sort objects (Shapes)
  - Before, we ordered Shapes by their areas

- Recall the Comparable<T> interface?

```java
public interface Comparable<T> {

    /**
     * Compares this object with the specified object for order.
     * @return a negative integer, zero, or a positive integer as this object
     * is less than, equal to, or greater than the specified object.
     */
    int compareTo(T o);
}
```
An Airplane Class

- Let's design Airplanes to be compared using the number of passengers
  - Want planes with more passengers to be prioritized

```java
public class Airplane implements Comparable<Airplane> {
    protected List<Passenger> passengers;

    /** @return the number of passengers onboard */
    public int size() {
        return passengers.size();
    }

    @Override
    public int compareTo(Airplane other) {
        return (other.size() - passengers.size());
    }
}
```
Let's design Airplanes to be compared using the number of passengers

```java
Airplane a1 = new Airplane();  // eventually filled with 50 passengers
Airplane a2 = new Airplane();  // eventually filled with 29 passengers
Airplane a3 = new Airplane();  // eventually filled with 90 passengers

// .. code omitted to add passengers ..

PriorityQueue<Airplane> runway_q = new PriorityQueue<Airplane>();
runway_q.offer(a1);
runway_q.offer(a2);
runway_q.offer(a3);
runway_q.poll();  //a3 dequeued
runway_q.poll();  //a1 dequeued
runway_q.poll();  //a2 dequeued
```
Takeaway: Java's PriorityQueue<E>

- Used when FIFO is an insufficient policy for determining order
  - Very, very common in CS

- Objects the priority queue holds should implement Comparable<T>
  - And override int compareTo(..)

- When comparable objects are enqueued, the PriorityQueue "magically" orders the queue based on return value of compareTo(..)
Thoughts on Implementation?

- Let's assume that a `LinkedList<E>` is used to store the queue

- One change: When `offer(E item)` is called:
  - May need to walk entire list to find place to insert item: $O(n)$
  - Then link up the new item: $O(1)$
  - Total: $O(n)$

- No changes to `poll(..)`, `peek(..)`, etc.
Analysis: PriorityQueue

- If we used a Linked List to implement a Queue
  - Assuming head of Linked List is head of queue

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>O(n)</td>
</tr>
<tr>
<td>public E remove()</td>
<td>O(1)</td>
</tr>
<tr>
<td>public E poll()</td>
<td>O(1)</td>
</tr>
<tr>
<td>public E peek()</td>
<td>O(1)</td>
</tr>
<tr>
<td>public E element()</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Can We Do Better?

- In CS 361, you'll learn how to use "heaps" to implement PriorityQueues
  - Heaps are just arrays that are ordered in a certain way
  - Heap implementation will give us:

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean offer(E item)</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>public E remove()</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>public E poll()</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>public E peek()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E element()</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Dequeuing gets slower for faster enqueuing

Therefore, use a heap-based implementation when you expect lots of enqueues!
Outline

- Queues
- Priority Queues
  - Implementation
  - Using Java's PriorityQueue
  - Comparator
- Stacks
  - Application of Stack: ParenChecker
- Conclusion
Stacks

- A stack is a last-in-first-out (LIFO) List!
  - We have many real-world examples
  - Notice how the last thing pushed on the stack is the first one popped off
Stacks (Cont.)

- A *stack* is a last-in-first-out (LIFO) List!
A *stack* is a last-in-first-out (LIFO) List!
Java has a Stack\langle E\rangle class that can be used.

• Here's its API

• Feel free to use this for assignments
  - But what's the fun without implementing our own?

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean isEmpty()</td>
<td>Returns true if the stack is empty, and false otherwise</td>
</tr>
<tr>
<td>public E peek()</td>
<td>Returns a reference to the top element \textbf{without} removing it. \textbf{Throws} java.util.EmptyStackException if stack was empty prior to</td>
</tr>
<tr>
<td>public E pop()</td>
<td>Remove the top element and return it. \textbf{Throws} java.util.EmptyStackException if stack was empty prior to popping</td>
</tr>
<tr>
<td>public E push(E item)</td>
<td>Pushes the item on top of the stack, and return a reference to it</td>
</tr>
<tr>
<td>public int search(Object o)</td>
<td>Searches for the object and returns the 1-based position if found, and -1 if not found.</td>
</tr>
</tbody>
</table>
Usage Example

- Import the class first

```java
import java.util.Stack;
```

- Sample usage in code pad:

```java
Stack<String> names = new Stack<String>();
names.push("Rich");
names.push("Mary");
names.push("Brian");
names.push("Jarrod");
names.peek()  // > "Jarrod" (String)
names.pop()   // > "Jarrod" (String)

names.push("Ally");
names.peek()  // > "Ally" (String)
```
Programming Exercise (From Book)

- Write a main function that creates three stacks of Integer objects. Store the numbers \(-1, 15, 23, 44, 4, 99\) in the first two stacks. The top of these two stacks should store 99.

- Write a loop to get each number from the first stack and store it into the third stack.

- Write a second loop to remove a value from the second and third stacks and display each pair of values on a separate output line. Continue until the stacks are empty. Show the output.
Outline

- Queues
- Priority Queues
  - Implementation
  - Using Java's PriorityQueue
  - Comparator
- Stacks
  - Application of Stack: ParenChecker
- Conclusion
A More Practical Application: ParenChecker

Languages and calculators must accept arbitrary expressions

- One thing you'd need to do would be to ensure the parentheses match!
  - Or else, it's an illegal expression!

Legal expressions:

- \( 4 + 5 - 1 \)
- \( \text{Math.sqrt(Math.pow(a,2) + Math.pow(b,2));} \)
- \( 1 + ((a / b) - 5 * c) \)

Illegal expressions:

- \( ) \)
- \( ((4 + 5) - 1 \)
- \( \text{Math.sqrt(Math.pow(a,2) + Math.pow(b,2));} \)
ParenChecker Idea

- Input: An expression is a String
- Run through each character in the String
  - If it's anything besides '(' or ')' then ignore it
  - If it's an open paren '('
    - Push it on the Stack
  - If it's a close-paren ')'
    - Pop an open-paren off the stack if possible
    - If not possible, then there was one fewer open-paren up to now -- return false
- After you've seen all characters in the String,
  - Return true if stack is empty, false otherwise
Example: Legal Expression

1 + ((a / b) - 5 * c)
Example: Illegal Expression

\[ 1 + \frac{(a / b) - 5 * c}{a + b} \]
Outline

- **Stacks**
  - Application of Stack: ParenChecker
    - Implementation

- **_queues**

- **Priority Queues**
  - Implementation
  - Using Java's PriorityQueue
    - Comparator

- **Conclusion**
What Good Is a Stack?

Lots of uses:

- Call stacks in language implementation
- Palindrome checker
- Matching parentheses checker (as we saw)
- Calculators: evaluating mathematical expressions

Since it's a more restrictive list, why not just use a List?

- You *could*, but...
- Cleaner API => cleaner code
- Better thought process for solving the problem
Let's Write Our Own Stack

First, we'll define our own Stack interface, called StackInt<E>

<table>
<thead>
<tr>
<th>Signature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean isEmpty()</td>
<td>Returns true if the stack is empty, and false otherwise</td>
</tr>
<tr>
<td>public E peek()</td>
<td>Returns a reference to the top element without removing it. Throws java.util.EmptyStackException if stack was empty prior to popping</td>
</tr>
<tr>
<td>public E pop()</td>
<td>Remove the top element and return it. Throws java.util.EmptyStackException if stack was empty prior to popping</td>
</tr>
<tr>
<td>public E push(E item)</td>
<td>Pushes the item on top of the stack, and return a reference to it</td>
</tr>
</tbody>
</table>

Here's the interface:

```java
public interface StackInt<E> {
    E push(E item);
    E pop();
    E peek();
    boolean isEmpty();
}
```
Stack Implementation

- We know we ought to use a List as the underlying structure, but which?
- Let's recall the behavior of stacks:
  - Pop, push, peek all done to the head of the list
  - No methods that access middle, back of list
  - LinkedLists it is!
    - Though, it should be noted that we can also implement using ArrayLists with the same performance *(how?)*
- Remember to implement StackInt<E> and *Override* the contracted methods.
### Analysis: Stack

- If we used a Linked List to implement a Stack

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E peek()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E pop()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E push(E item)</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
If we used an ArrayList to implement a Stack

- Requires a little ingenuity, but can still get the same performance

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>public boolean isEmpty()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E peek()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E pop()</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>public E push(E item)</td>
<td>$O(1)$ amortized</td>
</tr>
</tbody>
</table>
Outline

- **Stacks**
  - Application of Stack: ParenChecker
  - Implementation

- **Queues**

- **Priority Queues**
  - Implementation
  - Using Java's PriorityQueue
  - Comparator

- **Conclusion**
Conclusion

- Stacks and queues are two of the most commonly used data structures
  - Lots of problems can be simplified with their usage
  - Their usage will occur over and over again in this class and beyond

- Deep down, both are just Lists (FIFO vs. LIFO)
  - In fact less general
  - You can only operate on one end (Stack) or both ends (Queue)
  - Priority queues offer another way ordering items
What You Need to Know

- Their interfaces and implementation choices
  - How different choices of underlying List affects runtime of various operations

- How to get a PriorityQueue to automatically organize your objects
  - Comparable<T> interface

- Applications of stack and queues
  - Know when to use them when given a problem to solve
  - Look over the airport simulation code!