CSCI 261
Computer Science II
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
- 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!
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>
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<\textit{E}> 

- 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 <code>java.util.NoSuchElementException</code> 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 <code>java.util.NoSuchElementException</code> 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();
}
```
Analysis: Queue (Cont.)

- 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>public boolean offer(E item)</td>
<td>$O(1)$ amortized</td>
</tr>
<tr>
<td>public E remove()</td>
<td>$O(n)$ -- why?</td>
</tr>
<tr>
<td>public E poll()</td>
<td>$O(n)$ -- why?</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>
Analysis: Queue

- 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!)*
Applications of Queues

- Breadth-first search of graphs
  - Graph "nodes" are visited and adjacent nodes are stored in a queue

- Computer simulation
  - Example:
    - If it takes a toll booth operator 10 seconds to service a car, and there are 30 cars a minute, how much time would adding a toll booth delay traffic?
    - How many toll booths would we need to install to ensure traffic is only delayed by X seconds? Cost vs. profits?
    - Use queues to model a line of cars
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>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>24</td>
<td>takeoff completed, req_time=9, wait_time=15</td>
</tr>
<tr>
<td>38</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>53</td>
<td>takeoff completed, req_time=38, wait_time=15</td>
</tr>
<tr>
<td>54</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>67</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>69</td>
<td>takeoff completed, req_time=54, wait_time=15</td>
</tr>
<tr>
<td>76</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>84</td>
<td>takeoff completed, req_time=67, wait_time=17</td>
</tr>
<tr>
<td>99</td>
<td>takeoff completed, req_time=76, wait_time=23</td>
</tr>
<tr>
<td>102</td>
<td>takeoff requested</td>
</tr>
<tr>
<td>117</td>
<td>takeoff completed, req_time=102, wait_time=15</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
- 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

<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>

PriorityQueue<E>... implements the Queue<E> interface
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
class 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);
}
```
A College Class

- Want: colleges with more students to be prioritized in our queue
  - Write a `compareTo()` in such a way that a college is ordered before another college if it has more students

```java
public class College implements Comparable<College> {
    protected List<Student> students;

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

    @Override
    public int compareTo(College other) {
        return (this.size() - other.size());
    }
}
```
PriorityQueue Usage

College c1 = new College(2900);  // filled with 2900 passengers
College c2 = new College(500);   // filled with 500 passengers
College c3 = new College(60000); // filled with 60000 passengers

PriorityQueue<College> queue = new PriorityQueue<>();
queue.offer(c1);
queue.offer(c2);
queue.offer(c3);
queue.poll();  // c2 dequeued
queue.poll();  // c1 dequeued
queue.poll();  // c3 dequeued
Thoughts on PriorityQueue Implementation?

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

- No changes to poll(..), peek(..), etc.

- 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: $O(n)$
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>
In 361, you'll learn how to use a "heap" to implement PriorityQueues

- Heaps are just arrays ordered (not sorted, per se) in a certain way
- Heap-based PriorityQueue<E> 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>

Tradeoff:
Dequeuing gets slower for faster enqueuing

Therefore, use a heap-based implementation when you expect lots of enqueues!
Takeaway: Java's PriorityQueue<E>

- Used when FIFO is the wrong policy for determining order in queue
  - Very common, it turns out

- Know this:
  - Java's PriorityQueue<E> is heap-based, not list-based
  - Objects the PriorityQueue holds must implement Comparable<T>
Outline

- Queues
- Priority Queues
  - Implementation
- Stacks
  - Application of Stack: ParenChecker
- Conclusion
Stacks

- A *stack* is a last-in-first-out (LIFO) List!
  - We have many real-world examples
  - Note: the last thing *pushed* on the stack is the first thing *popped* off
A **stack** is a last-in-first-out (LIFO) List!
Java has a Stack<E> 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 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>
Usage Example

- Import the class first

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

- Sample usage in code pad:

```java
Stack<String> names = new Stack<>();
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)

1. 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.

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

3. 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
- Stacks
  - Implementation
  - Application of Stack: ParenChecker
- 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><code>public boolean isEmpty()</code></td>
<td>Returns true if the stack is empty, and false otherwise</td>
</tr>
<tr>
<td><code>public E peek()</code></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><code>public E pop()</code></td>
<td>Remove the top element and return it. Throws java.util.EmptyStackException if stack was empty prior to popping</td>
</tr>
<tr>
<td><code>public E push(E item)</code></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.
If we used a Linked List to implement a Stack

- Know why the following are true

<table>
<thead>
<tr>
<th>Signature</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>public boolean isEmpty()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>public E peek()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>public E pop()</code></td>
<td>O(1)</td>
</tr>
<tr>
<td><code>public E push(E item)</code></td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Analysis: Stack (Using ArrayList)

- If we used an ArrayList to implement a Stack
  - Use the 0th element as top-of-stack too?
    - No! Terrible for 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(n)</td>
</tr>
<tr>
<td>public E push(E item)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
If we used an ArrayList to implement a Stack

- Solution: Use the tail of the ArrayList to represent top of stack instead!

<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

- Queues
- Priority Queues
  - Implementation
- Stacks
  - Application of Stack: ParenChecker
- Conclusion
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:

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

Illegal expressions:

\[
) \\
((4 + 5) - 1 \\
Math.sqrt(Math.pow(a,2) + Math.pow(b,2));
\]
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

\[ 1 + ((a / b) - 5 \times c) \]
Example: Illegal Expression

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

- Queues
- Priority Queues
  - Implementation
  - Using Java's PriorityQueue
  - Comparator
- Stacks
  - Application of Stack: ParenChecker
  - Application of Stack: Infix Evaluation
  - Implementation
- Conclusion
Another Application: Infix Expression Solver

- In practice algebraic expressions usually follow *infix notation*

- As we know, parentheses explicitly determine precedence, but are often implied:
  - This expression $7 + 9 \times 10$ is equivalent to $(7 + (9 \times 10))$
  - This expression $5 \times 3 + 4 \div 2$ is equivalent to $(5 \times 3 + (4 \div 2))$

- Type the following into BlueJ:

```
( 7 + ( 9 * 10 ) )
> 97 (int)

( ( 5 * 3 ) + ( 4 / 2 ) )
> 17 (int)
```
How Does a Machine Solve Infix Expressions?

- We'll deal with the explicit infix version \(( ( 5 * 3 ) + ( 4 / 2 ) )\)
  - (Parentheses thrown around every two operands and their operator)

- Dijkstra's Two-Stack Algorithm:
  - Read each symbol from left to right
    - If "(" , ignore
    - If operand (any number), push on value stack
    - If operator (+,-,*,/), push on operator stack
    - If ")" , pop operator stack, and pop two values. Apply operator on two values, and push result on value stack.
Interestingly... (RPN)

- Dijkstra's 2-Stack algorithm works even if you rearrange the operators to come \textit{after} the two associated operands:
  - This expression $\left( 7 + \left( 9 \times 10 \right) \right)$ is equivalent to $\left( 7 \left( 9 \times 10 \right) + \right)$
  - This expression $\left( \left( 5 \times 3 \right) + \left( 4 / 2 \right) \right)$ is equivalent to $\left( \left( 5 \times 3 \right) \left( 4 / 2 \right) + \right)$

- Remove parentheses to obtain \textit{postfix (Reverse Polish) notation}

\begin{align*}
7 &\ 9 \ 10 \ \times \ + \\
5 &\ 3 \ \times \ 4 \ 2 \ / \ +
\end{align*}

\textit{Only need a value stack for processing postfix expressions! (Why?)}

\textit{Due to efficiency, used everywhere: calculator programs on all major operating systems (mac, win, linux, android, iOS)
Outline

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