Precept Outline

- Review of Lectures 3 and 4:
  - Stacks and Queues
  - Advanced Java
- Stacks: resizing arrays + linked lists

Relevant Book Sections

- 1.3 (Queues and Stacks)

A. Review: Stacks, Queues and Iterators/Iterables

Your preceptor will briefly review key points of this week's lectures.

Here are some code snippets that your instructor might refer to as examples:

```
Stack<String> stack = new Stack<String>();

stack.push("One");
stack.push("Two");
stack.push("Three");
stack.push("Four");
stack.push("Five");

for (i = 0; i < 5; i++)
   StdOut.println(stack.pop());
```

```
Queue<String> queue = new Queue<String>();

queue.enqueue("One");
queue.enqueue("Two");
queue.enqueue("Three");
queue.enqueue("Four");
queue.enqueue("Five");

for (i = 0; i < 5; i++)
   StdOut.println(queue.dequeue());
```

```
public class YourClass<Item> implements Iterable<Item> {

    public Iterator<Item> iterator() {
        return new YourClassIterator();
    }

    private class YourClassIterator implements Iterator<Item> {
        // instance variable(s) to keep track of where iterator is

        public boolean hasNext() {
            // condition to end iteration
        }

        public Item next() {
            // returns next item and updates instance variable(s)
        }
    }
```

```
Stack<String> stack = new Stack<String>();
// initialize stack

Iterator<String> iter = stack.iterator();

while (iter.hasNext()) {
  String s = iter.next();
  // do something with s
}
```

```
Stack<String> stack = new Stack<String>();
// initialize stack



for (String s : stack) {

  // do something with s
}
```

B. Stacks and Queues

Part 1: Resizing arrays

In lecture, you saw how the *repeated doubling* strategy solves the problem of resizing arrays too often. There was a caveat, however: we resize up at 100% capacity but resize down at 25% (rather than 50%) capacity.

(Warm-up) Recall what goes wrong if we resize down at 50%: give an example of a sequence of `push()` and `pop()` operations with $\Theta(n)$ amortized cost. The cost of a sequence of operations (as in lecture) is the total number of array accesses made throughout their execution.

Consider the following "resizing policies":

1. Double at 100% capacity, halve at 25%;
2. Triple at 100% capacity, multiply by 1/3 at 1/3;
3. Triple at 100% capacity, multiply by 2/3 at 1/3;
4. Double at 75% capacity, halve at 25%.

Identify which policies have worst-case $\Theta(n)$ amortized cost for n operations and which have $\Theta(1)$.

Part 2: Linked Lists

Recall that in a singly linked list, each node stores an item (of generic type) and a reference to the next node in the list. Describe a method that produces a new linked list with the same elements of `list` but in *reverse order*.

Assume that the input list is given as its `first` node. You can create extra nodes or linked lists, but not modify the input list. Feel free to write code or pseudocode.

Part 3: Fall'22 Midterm Problem

We wish to implement a method `public static String parseUndos(String str)`, which takes as input a string that represents a series of keystrokes and interprets each occurrence of the `<` symbol as a one-character undo request. The method returns the string that is obtained after the undo requests are implemented.

For example,

```
String s = parseUndos("Princesses<<<<ton");
String t = parseUndos("COM<S217<<40<<26?<!");

StdOut.println(s);
StdOut.println(t);
```

prints the strings `Princeton` and `COS226!`.

(a) Fill in the two blanks in the following Java implementation of `parseUndos()`.

```
public static String parseUndos(String str)
{
    Stack<Character> stack = new Stack<Character>();
    for (int i = 0; i < str.length(); i++) {
        char current = str.charAt(i);
        if (current != '<')
            _______ // first blank
        else
            _______ // second blank
    }
    // copy the content of the stack to a new string.
    String newStr = "";
    while (!stack.isEmpty())
        newStr = stack.pop() + newStr;
    return newStr;
}
```

(b) Executing the statement `parseUndos("COS226!<<<<<<<<<<")` yields a(n)

( ) Stack overflow.

( ) Stack underflow.

( ) Infinite loop.

( ) Return value `"COS226!"`.

( ) Return value `"<<<<<<<<<<"`.

( ) None of the above.

(c) Assume that the Stack data type is implemented as a *resizable array*. How many times would the array *shrink* when calling `parseUndos(str)`, where `str` is a string that consists of $16n$ non-undo characters followed by $13n$ undo characters?

Assume that the stack is 100% full after `parseUndos` processes the first $16n$ non-undo characters, and recall that `pop()` resizes the array (to half of its size) when it reaches 25% capacity.

( ) 0.

( ) 1.

( ) 2.

( ) Constant strictly greater than 2.

( ) $\sim \frac{1}{4} \log n$.

( ) $\sim \log n$.

(d) When the stack is implemented as a linked list, the *worst-case* running time of `parseUndos()` on a string of length n is $\Theta(n)$.

( ) True.

( ) False.

(e) When the stack is implemented as a resizable array, the *worst-case* running time of `parseUndos()` on a string of length n is $\Theta(n)$.

( ) True.

( ) False.

C. Assignment Overview: Queues

Your preceptor will introduce and give an overview of your second assignment. Please don't hesitate to ask questions!

Summary of the assignment.

- Implement a *deque* data structure, which is a queue that supports insertion and removal of items from both ends. This will be done in `Deque.java`.
- Implement a randomized queue, which is a queue that differs from a typical queue in that items are removed uniformly at random, not based on the sequence they were added. This will be done in `RandomizedQueue.java`.
- Implement an application of the randomized queue to read a sequence of strings and print a subset of them uniformly at random. This will be done in `Permutation.java`.
- Both deque and randomized queue require iterators that support operations in constant worst-case time and use space efficiently.

D. Optional Bonus Problems

Part 1: Obstructed Skyline (Challenge)

Suppose you are given the shape of a city's skyline in the form of a length-n array $h = [h_0, h_1, \ldots, h_{n-1}]$ (where the height of building i is h_i). In other words, the skyline is a $1 \times n$ grid where the ith column/building has height $h_i \leq k$.

Design an algorithm to find the rectangle with the largest area that is blocked by the skyline. (E.g., your algorithm should output 2 when $h = [2, 1]$, 4 when $h = [2, 2]$ and 12 with the input drawn below.) It should run in $\Theta(n)$ time and space.