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

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());
```

```
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 can't resize down at 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

In each of the following questions, assume that each linked-list is singly linked-list, i.e. each node has some item (of generic type) and a reference to the next node in the list. Further assume that each linked-list is given to you through its `first` node. You can create extra nodes or linked-lists, but you shouldn’t modify any linked-list given as input. To answer each question feel free to write code or pseudocode to describe the requested algorithm.

All the operations should run in linear time in the size of the linked-lists given in the input.

Reverse: Suppose you have one linked-list `l`, that contains any number of elements. Describe an algorithm that produces a new linked-list that contains the same elements of `l` but in reverse order.

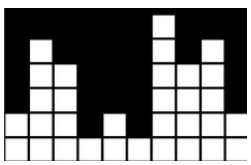
Merging: Suppose you have two linked-lists l1 and l2 that can contain any number of elements. Describe an algorithm that produces a new linked-list that merges l1 and l2, i.e. it contains all of the elements of l1 in the same original order, followed by all the elements of l2.

Unique: Suppose you have one linked-list l **integers** that contains any number of elements in sorted order (non-decreasing). Describe an algorithm that produces a new linked-list that contains the unique elements of l, i.e. it contains the same elements as l but no duplicates.

Part 3: Challenge Problem (optional)

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.



C. Iterators and Iterables

Solve all the exercises in the "Iterators" Ed lesson (including the "Stack Iterator" part).

D. 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.
- Determine the theoretical memory usage of each of the data structures, and report it is using tilde notation. This will be done in `readme.txt`.