Precept Outline

- Review of Lectures 3 and 4:
  - Stacks and Queues I
  - Stacks and Queues II
- Problem Solving

Relevant Book Sections

- 1.3 (Stacks and Queues)

A. Review: Linked Lists, Resizable Arrays, 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:

```
public class SinglyLinkedList<Item> {
    private Node first = null;
    private class Node {
        Item item;
        Node next;
    }

    public void push(Item item) {
        Node oldFirst = first;
        first = new Node();
        first.item = item;
        first.next = oldFirst;
    }

    public Item pop() {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

```
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

```

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

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

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

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

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

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

for (String s : stack) {
    // do something with s
}
```

B. Stacks and Queues

Part 1: Resizable arrays

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

(Warm-up) Recall what goes wrong if we resize down at 50%: give an example of a sequence of m `push()` and `pop()` operations with $\Theta(m)$ amortized cost (per operation). 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 $\Theta(1)$ amortized running time per operation.

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 given a linked list it *reverses the order* of the elements. So, for example, a list of integers containing $1 \to 2 \to 3 \to 4$ would become $4 \to 3 \to 2 \to 1$.

Assume that you are implementing a public instance method in the `SinglyLinkedList` implementation from the review section (so `public Node reverse()` would be the function signature). You can modify the original input list, but you **can't** create extra nodes or linked lists. 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.
    StringBuilder newStr = new StringBuilder();
    while (!stack.isEmpty()) {
        newStr.append(stack.pop()); // Append characters (in reverse order)
    }

    return newStr.reverse().toString();
}
```

Recall that appending a character to a `String` takes linear time in the length of the string, so we use the `StringBuilder` class instead to append characters in constant time. Also, note that when removing characters from a stack they come in reverse order (since the last character is popped first), so we need to reverse the string at the end.

(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.

( ) $\sim \frac{1}{4} \log_2 n$.

( ) $\sim \log_2 n$.

(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.