COS226 Precept 2 **Fall** ’23

Topics

- Review of Lectures 3 and 4:
    - Stacks and Queues
    - Advanced Java
- Stacks: resizing arrays + call stack
- Iterables and Iterators

Relevant Material

- Book chapter: 1.3

A. RECAP: Stacks & Queues + Advanced Java

This section contains a number of code snippets that your preceptor will help you analyze. Stay tuned!

- **Stacks and Queues**

```java
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());
```

```java
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());
```

- **The call stack**

```java
public class Pythagoras {
   private static double square(double side)
      { return side * side; }

   public static double hypotenuse(double a, double b) {
      return Math.sqrt(square(a) + square(b));
   }

   public static void main(String[] args) {
      // Unit testing
   }
}
```

- **Iterators and Iterables**

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

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

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

```java
Stack<String> stack = new Stack<String>();
// initialize stack
for (String s : stack) {
      // do something with s
}
```

B. EXERCISES: Stacks

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

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

(Optional) For those with $\Theta(1)$ amortized cost, express the cost of performing n `push()` operations in tilde notation (i.e., calculate the constant). Note that the calculation performed in lecture assumes $n = 2^i$, but we do not do this here – a useful warm-up is redoing that example with $n = 2^i + 1$.

You may assume any initial array size (e.g., 2, 3 or 4) for each case if it helps your reasoning; this will not change the constants.

2) The call stack

In this problem, you will use the call stack to work through the execution of nested methods. This is quite a good tool to have in your toolbox if you're ever confused by recursion. (And who isn't?)

Write the output of the execution of the following snippets (part of the code is omitted for simplicity).

```
int multiply(int a, int b) {
   return a * b;
}

      int sum(int a, int b) {
   return a + b;
}


StdOut.println(sum(multiply(2,
3), multiply(4, 5)));
```

```
void methodOne(int n) {
   if (n <= 0) return;
   StdOut.println("Calling methodTwo(%d)", n-1);
   methodTwo(n - 1);
}

void methodTwo(int n) {
  if (n <= 0) return;
  StdOut.println("Calling methodOne(%d)", n-1);
  methodOne(n - 1);
}

methodOne(3);
```

```
int collatz(int n) {
   StdOut.println(n);
   if (n == 1) return 1;
   if (n % 2)
      return collatz(n / 2);
   else
      return collatz(3 * n + 1);
}

collatz(5);
```

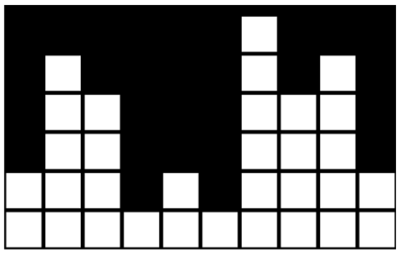
```
void methodOne(int n) {
   if (n <= 0) return;
   methodTwo(n - 1);
   StdOut.printf("Called methodTwo(%d)", n-1);
}

void methodTwo(int n) {
  if (n <= 0) return;
  StdOut.println("Calling methodOne(%d)", n-1);
  methodOne(n - 1);
}

methodOne(3);
```

3) Challenge Problem (<u>optional</u>): 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 an $k \times n$ grid where the i^{th} 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. EXERCISES: Iterators and Iterables

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

D. ASSIGNMENT OVERVIEW: Queues

Your preceptor will briefly go over the assignment due next week. If you haven’t yet and have some time to spare, take a look at the assignment page. Feel free to ask questions, but please leave implementation/debugging details for the lab TAs and office hours.