Lecture 4: Stacks/Queues

- We've talked a lot about ArrayIntList, but now I want to discuss some other structures
  - We define these structures as "Abstract Data Types" or ADTs
  - ADT
    - Specifies some collection of data, how to interact with it
    - WHAT it does, not HOW it does it (BEHAVIORS)
    - Ex. a LIST is an ordered sequence of elements, can add/remove, indices
      - But can be implemented in different ways, such as with an array
      - Another implementation next week
  - In Java, we usually refer to the generalization (ex. LIST not ARRAYLIST) by using something called an INTERFACE
    - I'll talk about how interfaces work later, but for now, think of it as a contract
      - Ex. a certification - "I am a certified public accountant"
    - Code is more flexible when you use the interface type
  - Example: The generalization of ArrayList<String> is List<String>
    - Right hand side stays the same, but all other references to the type of the list change
    - We can switch out the implementation later with 1 line of code!
    - Whenever you CAN use an interface, you SHOULD use an interface (points!)
  - We'll look at 2 more ADTs today - STACK and QUEUE
    - These are very simple types
    - We don't need to understand how they are implemented, just how they work
    - (because ABSTRACT data types) - see more in future CS classes
- Overview of stacks and queues
  - Compared with the list we did last week, they seem very SIMPLE
  - Analogy: drawers and shelves
    - Very simple, and therefore kind of boring
    - But we see uses for them everywhere
  - Much less powerful than the list, but sometimes problems don't need complication
    - "Minimal data structure"
    - Think of simplest possible solution to a problem
  - Like List, stores ORDERED sequence of VALUES
  - So what minimal behaviors do we need?
    - ADD
    - REMOVE
    - IS ANYTHING LEFT?
    - Can add more, but this is the essence of the structures (can also have "size")
  - LIFO (last-in-first-out - stack) vs. FIFO (first-in-first-out - queue)
    - LIFO - like trays at the cafeteria (take from the top, place clean ones on top)
      - Tends to reverse things
    - FIFO - like a line at the grocery store

- Stack
    - Behaviors
        - PUSH (like add) - push down clean cafeteria trays
        - POP (like remove) - take a cafeteria tray
        - ISEMPTY
        - SIZE
    - Generics - like ArrayList, can store any type in the stack - String, Integer...
    - Notice - no idea of “indices”
    - Examples in CS?
        - “Undo” in a word processor
    - Implementation type: same
        - Java messed up
        - There is no interface type
- Queue
    - Behaviors
        - ADD - adds to the end
        - REMOVE - removes from the front
        - ISEMPTY
        - SIZE
    - Same generics, also no indices
    - Examples in CS?
        - Print jobs sent to the printer
    - Implementation type: LinkedList
        - Java did the right thing here
- S/Q have other operations in Java, but for this course, we almost always restrict you to these 4 operations
    - NO ITERATORS (cheating!)
- Simple example
    - (EXPLAIN HOW IT WORKS)
    - Notice WHILE instead of FOR - because there is no get() operation
    - Use debugger on toString() mode
        - Watch elements disappear
    - Notice how stack values come out in reverse order

- Let's write some code to interact with stacks and queues of integers
  - Create queue
    - Use interface type/implementation type pair
  - I want to fill the queue with random elements
    - 10 elements between 0 and 30

```
Queue<Integer> q = new LinkedList<Integer>();
Random r = new Random();
for (int i = 0; i < 10; i++) {
    q.add(r.nextInt(30));
}
```

  - Move into its own method - take size as a parameter
    - Return type is Queue<Integer>, not LinkedList
- Let's write a method to transfer values from a queue to a stack
  - Very common operation in the problems you will be doing

```
public static void queueToStack(Queue<Integer> q, Stack<Integer> s) {
    while (!q.isEmpty()) {
        int n = q.remove();
        s.push(n);
    }
}
```

  - Just like with iterator, we only want ONE remove call per iteration of the loop
- Let's create a random queue, then move the things into the stack
  - Then print the queue and the stack
  - We see that everything moves!
- Let's find the sum of things in the queue
  - A first attempt:

```
while (!q.isEmpty()) {
    sum += q.remove();
}
return sum;
```

  - Print queue afterwards
  - Doesn't work - destroys the queue!
  - Have to restore the queue to its original state - how can we do it? Use queue as storage

```
int sum = 0;
for (int i = 0; i < q.size(); i++) {
    int value = q.remove();
    sum += value;
    q.add(value);
}
return sum;
```

- Same for stack

```
Stack<Integer> nums = new Stack<Integer>();
Random r = new Random();
for (int i = 0; i < size; i++) {
    nums.push(r.nextInt(30));
}
return nums;
```

- Another operation: stack to queue!

```
public static void stackToQueue(Stack<Integer> s, Queue<Integer> q) {
     while (!s.isEmpty()) {
         int n = s.pop();
         q.add(n);
     }
 }
```

- OK, then let's do sum on a stack

```
int sum = 0;
for (int i = 0; i < s.size(); i++) {
    int n = s.pop();
    sum += n;
    s.push(n);
}
return sum;
```

  - But this doesn't work! You push and pop the same thing
  - What can we do?
  - We CAN'T do it unless we have another structure

```
int sum = 0;
Queue<Integer> q = new LinkedList<Integer>();
for (int i = 0; i < s.size(); i++) {
    int n = s.pop();
    sum += n;
    q.add(n);
}
queueToStack(q, s);
return sum;
```

  - Can use queueToStack method to go back.
  - But still not correct - reversed!
    - Also only half are in reverse order - the for loop only goes halfway b/c we're taking things out and not returning them
    - Remember, using a stack tends to reverse things, s->q->s reverses

- So we have to reverse again! And store the size before the loop

```
public static int sum(Stack<Integer> s) {
     int sum = 0;
     Queue<Integer> q = new LinkedList<Integer>();
     while (!s.isEmpty()) {
         int n = s.pop();
         sum += n;
         q.add(n);
     }
     queueToStack(q, s);
     stackToQueue(s, q);
     queueToStack(q, s);
     return sum;
 }
```

- Other: max of a stack

```
int max = nums.peek();
Stack<Integer> backup = new Stack<Integer>();

while (!nums.isEmpty()) {
    int value = nums.pop();
   backup.push(value); // save
    if (value > max) {
        max = value;
    }
}

// restore the stack
while (!backup.isEmpty()) {
    nums.push(backup.pop());
}
return max;
```

  - How would you do it with a Queue as extra storage?
- Other: max of a queue
- Other: mirror a stack using a queue as auxiliary structure
- Tomorrow: all about stacks/queues
  - Next homework also about s/q