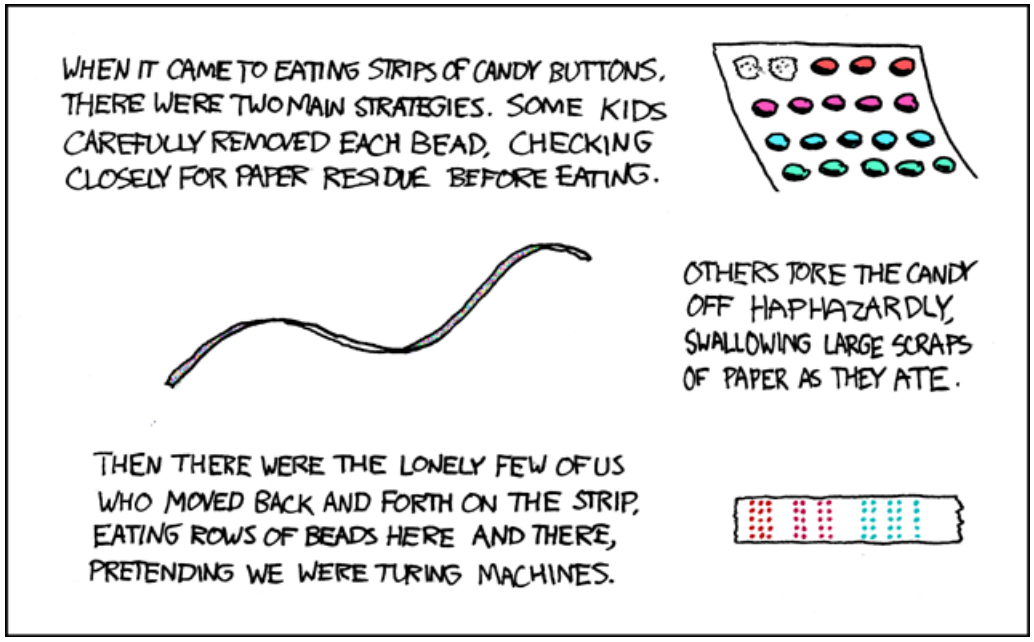
CS201 - Assignment 7
Due: Wednesday April 16, at the beginning of class



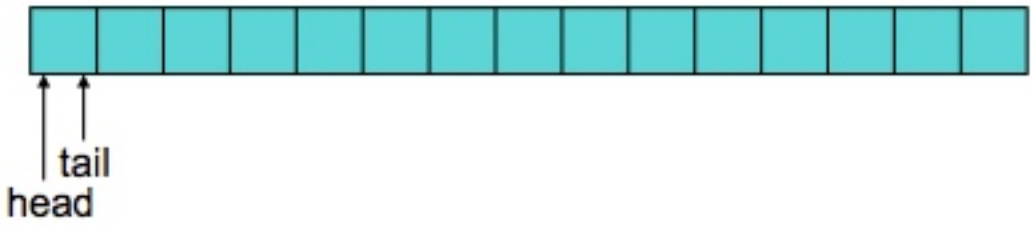
http://xkcd.com/205/

For this assignment, we're going to be playing a bit with stacks and queues. **Make sure you read through the entire assignment before starting since I give hints at the end.**

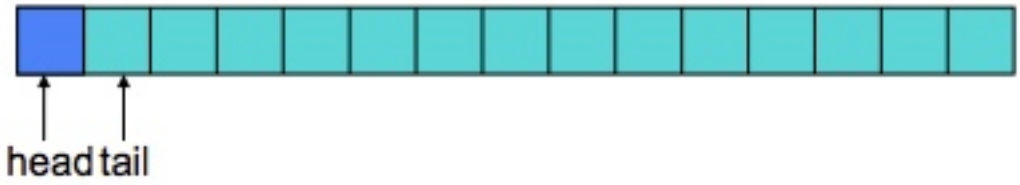
1 One more Q

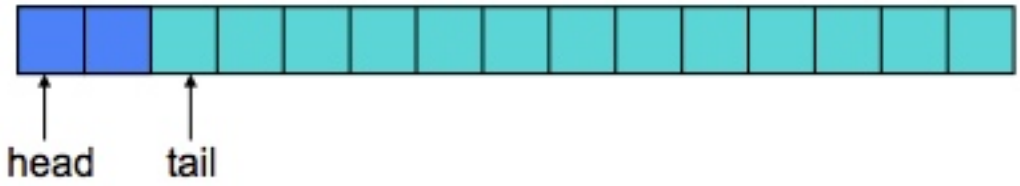
In class, we saw that you can implement a queue with both an `ArrayList` or a `LinkedList` (*Which is better?*). In some situations, you know the maximum number of things that you will ever have to keep track of in the queue. In these situations, there is a third implementation option: an array!

Like the linked list implementation, you keep track of the head and tail of the queue. However, instead of begin references, the head and tail are just indices into a fixed size array. Elements are added (`enqueued`) at the tail and elements are removed (`dequeued`) from the head. For example, below is a depiction of an empty array queue and the head and tail references (note, these will actually be indices not references in code):



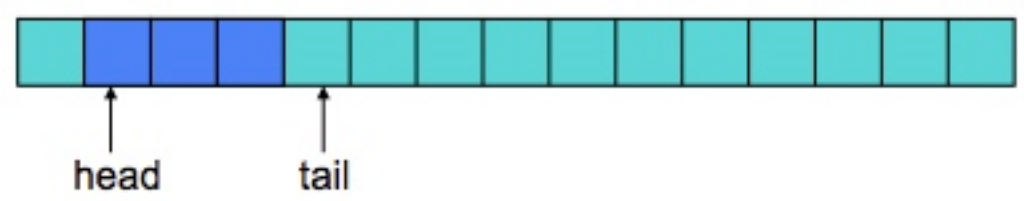
As elements are added, the elements are added at the end of of the queue where the tail is:



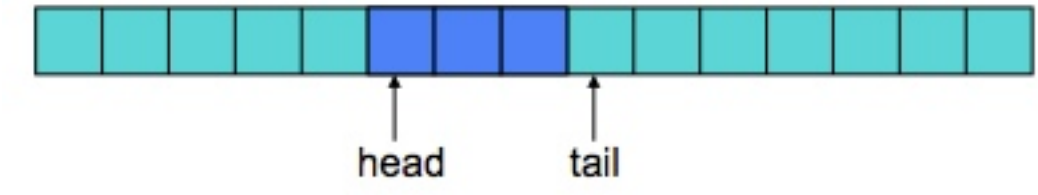


When an element is removed, it is removed from the head:

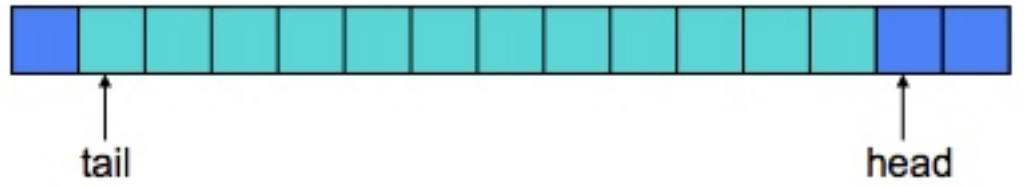
This process can happen (adding at the tail and/or removing from the head) regardless of where the head and tail are in the array. For example, we can do another add:



If we continue doing adds and removes, the location of the data inside the array shifts:



The one thing that you need to be careful of is that eventually the tail (then the head) may reach the end array. When this happens, we can wrap around do the beginning again:



As long as the number of elements added is less than the capacity, we can continue to do this indefinitely.

2 Test-driven development

For the first part of this assignment, you're going to implement an array-based queue that implements the `Queue` and `Linear` interfaces we saw in class. One approach to development is to *first* write unit tests to test the various functions. Once you've written tests, you then write the different methods. Each time you complete a method you can then run the tests and make sure it works. This is called test-driven development.

For this assignment, I'm going to have you try this out. At:

`http://www.cs.middlebury.edu/~dkauchak/classes/cs201/assignments/assign7/`

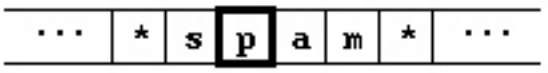
I've provided starter code for this class. Create a new Java project for this assignment and copy in the starter code. As part of this code is a skeleton for the `ArrayQueue` class.

Before you attempt to fill in the details for this class, create a new JUnit test class called `ArrayQueueTest`. Write tests that test all four of the `Queue` methods using `ArrayQueue`. You will be graded based on how thorough your tests are.

Once you have a good collection of JUnit tests, start writing the `ArrayQueue` class. Ideally, you implement one (or sometimes a couple) method and then run the tests. Sometimes, like in the case of `enqueue` and `dequeue`, you may have to implement both to really get good testing.

3 Turing Machines

To study the what computers can and cannot do, it is often useful to come up with a theoretical model of computation. *Turing machines* are one such model. There are many variations of Turing machines, however, the basic idea is that you have a linear tape that is infinitely long in both directions. The tape has cells that can hold symbols (e.g. characters). The machine has a tape head that hovers over the "current" cell being processed on the table. For example, here is an example tape:



For our machine, we will use the character '*' to represent an empty cell in the tape. This tape has 4 non-empty cells and the tape head is over the cell that contains the letter 'p'.

Turing machines "compute" by moving the tape head left or write, examining the current character and then making decisions based on the current character.[1]

4 `Tape`

For the second part of this assignment, we're going to implement a class that models the tape in a Turing machine. Specifically, a class called `Tape` that has the following `public` methods:

- A constructor that creates an empty tape. An empty tape will have a single empty cell (i.e. cell with '*').
- `store(char c)`: Write this character on the tape at the current tape head position.
- `read()`: Read the character on the tape at the current tape head position.
- `left()`: Move the tape head to the left. The tape can move infinitely to the left, so if there is not a character on the tape to the left, it will move over an empty cell.

[1] It can be shown that a real computer can compute can also be computed by a Turing machine!

- `right()`: Move the tape head to the right. The tape can move infinitely to the right, so if there is not a character on the tape to the right, it will move over an empty cell.
- `toString()`: Generates a `String` representing the tape from left to right of *any* cell that the tape head has ever visited. Notice that this may include some "empty" cells that may have had the tape head move over. The position of the tape should be indicated by a pair of square brackets surrounding the contents of the cell under the tape head. Each cell position should be separated by a space.

Implementation requirements

To make things interesting, we are going to impose the following restrictions on the `Tape` class:

1. You may only have three instance variables `left`, `right` and `current`.
2. The type of `left` and `right` *must* be either a queue or a stack, that is our `Queue` or `Stack` interface.
3. The type of `current` must be `char` (a character).
4. You may not use any data structures in the `Tape` class besides queues or stacks (e.g. no arrays, `ArrayList`s or linked lists).

A few example runs

Consider the following sequence of calls to the tape:

```
Tape t = new Tape();

t.store('a');
t.left();
t.left();
t.right();
t.store('c');
t.right();
t.right();
t.store('t');
t.right();
t.left();
t.left();
```

If we were to put a `System.out.println(t);` after each line we would see the following displayed:

```
[*]
```

```
[a]
[*] a
[*] * a
* [*] a
* [c] a
* c [a]
* c a [*]
* c a [t]
* c a t [*]
* c a [t] *
* c [a] t *
```

5 Writing simple programs on our Tape

Having to write out calls to all of the methods of the tape class to write simple programs gets old very quickly. To make life a bit easier, write a `static` method in the `Tape` class called `execute`. This methods should take a `String` as a parameter. Each character in the string will represent an instruction for the tape, specifically:

character	method
$>$	`right()`
$<$	`left()`
?	print out the tape using `System.out.println()`
c	`store(`c`)`

The method will generate a new `Tape` and read through the input string a character at a time, executing the appropriate methods on the tape.

For example, if we ran the string "`?a<<>c>>t><<?`" through `execute`, we would see printed out:

```
[*]
* c [a] t *
```

If you want to try another one for fun, try "`<<e<v<<<<<I>>l>o>>>>>>><S<C<<<<<<<?`"

6 A few hints

- If you need an example of how to create an array that stores generics for `ArrayQueue`, look at our implementation of `ArrayList`.
- There are many ways to handle the wrap-around problem in the array implementation of queues. The *cool* way (and, honestly, the best way) is to use the mod operator ('%') to help you out. See the Wikipedia page on "Modular arithmetic":

`http://en.wikipedia.org/wiki/Modular_arithmetic`

- Think about which data structures are appropriate for this problem. Make sure you think through all of the use cases. You will be graded on efficiency!
- Most of the methods in the `Tape` class can be written *very* simply. The only one that should require any major amount of code is `toString`.
- Since stacks and queues do not provide any way of iterating over the elements in them you will have to get a little creative in the `toString` method. You will very likely need a temporary data structure to help you out here (but, remember, it will have to be a stack of a queue!).
- To store characters in a stack or queue, you will need to use the `Character` class. For example, if you decided to use a `Queue` for `left` variable, it would be declared like:

  ```
  Queue<Character> left;
  ```

  and you could assign to it like:

  ```
  left = new LinkedQueue<Character>();
  ```

- The `String` class has a method called `charAt` that gets the character at a particular index in the `String`.
- Characters are indicated with single quotes. For example, the character:

  ```
  char empty = '*';
  ```

  would create a new variably of type `char` called `empty` with the asterisk character stored in it.

7 When You're Done

Make sure you have comments at the top of any file/class that you wrote with with your name and the assignment number.

JavaDoc

All of your methods MUST have JavaDoc style headers. I've included it for many of the methods, but make sure you include it for any additional ones you write.

To group these files into a single file, you need to export your project. To do this:

1. Right-click on the project (on Mac, ctrl+click) and select `Export`.
2. You'll see a number of options. Open up the `Java` folder and select `JAR file` and click `Next`.

3. You should just see your project selected. Below, make sure **only** the following two options are checked:

   - "Export Java source files and resources"
   - "Compress the contents of the JAR File"

4. Click on the `Browse...` button and pick a location to save the output file. Give the file a name like "kauchak1.jar", where "kauchak" is your last name and "1" is the assignment number.

5. Click `Finish`.

If all went well this will generate a single `.jar` file wherever you picked to save it.

Submit this JAR file as assignment number "5.2" via the online submission mechanism on the course web page.