Queue

Last lecture we looked at a abstract data type called a ”stack”. I introduced the idea of a stack by saying that it was a kind of list, but with a restricted set of operations, `push` and `pop`. Today we will consider another kinds of abstract data type called a ”queue”. A queue can also be thought of a list. However, a queue is again a restricted type of list since it has a limited set of operations.

You are familiar with queues in daily life. You know that when you have a single resource such as a cashier in the cafeteria, you need to “join the end of the line” and the person at the front of the line is the one that gets served next. There are many examples of queues in a computer system. When you type on your keyboard, the key values enter a queue (or 'buffer'). Normally don't notice the queue because each keystroke value gets read and removed from the queue before the next one is entered. But sometimes the computer is busy doing something else, and you do notice the queue. There is a pause where nothing you type gets echoed to your screen (e.g. to your text editor), and then suddenly some sequence of characters you typed gets processed very quickly. Other examples of queues are printer jobs, CPU processes, client requests to a web server, etc.

The fundamental property of a queue is that, among those things currently in the queue, the one that is removed next is the one that first entered the queue, i.e. the one that was least recently added. This is different from a stack, where the one that is removed next is the newest one, that is, the most recently added. We say that queues implement “first come, first served” policy (also called FIFO, first in, first out), whereas stacks implement a LIFO policy, namely last in, first out.

The queue abstract data type (ADT) has two basic operations associated with it: `enqueue(e)` which adds an element to the queue, and `dequeue()` which removes an element from the queue. We could also have operations `isEmpty()` which checks if there is an element in the queue, and `peek()` which returns the first element in the queue (but does not remove it), and `size()` which returns the number of items in the queue. But these are not necessary for a core queue.

Example

Suppose we add (and remove) items `a,b,c,d,e,f,g` in the following order, shown on left. On the right is show the corresponding state of the queue *after* the operation.

```
OPERATION                 STATE AFTER OPERATION
                          (initially empty)
enqueue(a)                a
enqueue(b)                ab
dequeue()                 b
enqueue(c)                bc
enqueue(d)                bcd
enqueue(e)                bcde
dequeue()                 cde
enqueue(f)                cdef
enqueue(g)                cdefg
```

Data structures for implementing a queue

Singly linked list

One way to implement a queue is with a singly linked list. Just as you join a line at the back, when you add an element to a singly linked list queue, you manipulate the `tail` reference. Similarly, just as you serve the person at the front the queue, when you remove an item from a singly linked list queue, you manipulate the `head` reference. The `enqueue(E)` and `dequeue()` operations are implemented with `addLast(E)` and `removeFirst()` operations, respectively, when a singly linked list is used.

Array list

What if we implement a queue using an array list? In this case, `enqueue(element)` would be `addLast(element)` and `dequeue()` would be `removeFirst()`. The `enqueue()` with array list is fine, and the only issue to note is that we need to use a larger underlying array in the case that the array is full. The problem is the `dequeue()` since `removeFirst()` is inefficient for array lists since we have to shift all the remaining elements.

Expanding Array

A better way to use an array to implement a queue would be to relax the requirement that the front of the queue is at position 0. Instead of shifting when we dequeue, we keep track of an index `head` which is the index of the next item to be removed.[1] We also keep track of `size`. Then, `tail = head + size - 1`. Note that when the queue is initialized, `tail == -1`. Also note that we need to expand the array when it is full. Finally, note that, with this approach, both `add` and `remove` require only a few operations (independent of the length of the array). Below is the state of the array queue for the same example as above.

```
                 -------------------------------------------
                 01234567              head    tail   size

                 ----                    0      -1      0
enqueue(a)       a---                    0       0      1
enqueue(b)       ab--                    0       1      2
remove()         -b--                    1       1      1
enqueue(c)       -bc-                    1       2      2
enqueue(d)       -bcd                    1       3      3
enqueue(e)       -bcde---                1       4      4
remove()         --cde---                2       4      3
enqueue(f)       --cdef--                2       5      4
enqueue(g)       --cdefg-                2       6      5
                 --------------------------------------------
```

The problem with this approach is that, when we remove an element from the array, we never use that array position again. This is an inefficient usage of space.

[1]In the context of linked lists, `head` was a reference variable. For arrays, we can treat `head` as an integer index.

Circular array

A better approach is treat the array as *circular*, so that the last array position (`length-1`) is 'followed' by position 0. The relationship between indices becomes

$$\texttt{tail} = (\texttt{head} + \texttt{size} - 1)\ \%\ \texttt{length}.$$

Note that in the initial state, we have `size == 0` and `head == 0`, and so the formula implies that `tail` has value `length - 1`. See example below.

The algorithm for dequeueing is as follows. Note that it does not change `tail`.

```
dequeue(){          //  check that size > 0  (omitted)
   element = queue[head]
   head = (head + 1) % length
   size = size - 1
   return element
}
```

One subtlety here is that if there is just one element in the queue, then `head == tail` before we remove that element. So if remove this single element, then we advance the `head` index which will leave `head` with a *bigger* value than `tail` (unless `head` and `tail` were `length-1` in which case `head` becomes 0).

What about enqueueing? The next available position is `(head + size) % length`, but only if `size < length` since if `size == length` then the array is full. In the case that the array is full, the length of the array needs to be increased before we can add the element.

Take the above example and suppose that the array has `length = 4`. As always, we have

$$\texttt{tail} = (\texttt{head} + \texttt{size} - 1)\ \%\ \texttt{length}.$$

	0123	head	tail	size	
	----	0	3	0	tail == -1 % 4 == 3
enqueue(a)	a---	0	0	1	
enqueue(b)	ab--	0	1	2	
remove()	-b--	1	1	1	
enqueue(c)	-bc-	1	2	2	
enqueue(d)	-bcd	1	3	3	
enqueue(e)	ebcd	1	0	4	
dequeue()	e-cd	2	0	3	
enqueue(f)	efcd	2	1	4	

The next instruction is `enqueue(g)`. If we were simply to double the length of the array

```
efcd----
```

then it would be awkward to add the `g` since it is supposed to go after `f`. Instead, we copy the elements to the larger array such the head is at position 0 in the array.

```
cdef----
```

so that `head == 0, tail == 3`.

So the algorithm for enqueueing would go like this:

```
 enqueue( element ){
     if ( size == length)  {    //  increase length of array
        create a bigger array tmp[ ]    //  e.g. 2*length
        for i = 0  to length  - 1       //  'length' of queue (not of tmp)
            tmp[i] = queue[ (head + i) % length ]
        head = 0
        queue = tmp     //   'length' of queue is now twice as big as before
     }
     queue[size] = element
     queue.size = queue.size + 1
}
```

There was some discussion in class about whether this is the best way to copy the elements to the bigger array. Other suggestions were made. I am not claiming that the method above is the only way to do it. Rather, I am claiming that you need to copy all the elements of the queue from the full small array to the bigger array anyhow, so there no gain in efficiency is doing it any other way than what I suggested above. (I also claimed that other ways of doing it that seem right, in fact have subtle problems with it.)

ADT's, the Java API's, and interfaces

I finished the lecture by discussing some terminology and three related concepts.

ADT's

I have discussed three ADTs: lists, stacks, and queues. ADTs are not formal mathematical things, and they are not programming language specific. Rather, they are abstract things that exist in our heads only. They are a tool for thinking about algorithms and writing them out in pseudocode. This is helpful because algorithms can be complicated and subtle, even at a high level and having to think about coding them correctly in some particular programming language just makes things more complicated. If the implementation details don't matter because we are thinking and working at a highl level, then it is better to leave them aside and work with ADTs. Note that you are free to include whatever detail you like in an ADT. Certain operations might be $O(N)$ with one implementation of an ADT but $O(1)$ with another implementation and these details might be part of the description of the ADT too. The point is that the ADT gives the 'user' the information that they need, but doesn't burden the user with implementation details that they don't need.

Java API

By this point in the course, you should have checked out the Java API for various classes, such as the `ArrayList` or `LinkedList` classes. The Java API for a class is similar to an ADT in that that the Java API specifies the methods for the class, and what the methods do, but it doesn't specify the details on the implementation. The "I" in API stands for interface (application program interface) and the meaning is that the API for a class contains all the information that the user gets to know about that class.

Java `interface`

In Java, the word `interface` is also a reserved word (or key word) and the meaning of interface is quite different there. An `interface` is like a class in that it contains a set of method definitions. Specifically a method is defined formally by a return type, and by a method *signature* (a method name, and method argument types in a particular order). An `interface` does not and cannot include an implementation of the methods, however. Rather, in Java one needs to define a class that `implements` the interface where `implements` is another reserved word in Java.

For example, there is a `List` interface (`https://docs.oracle.com/javase/7/docs/api/java/util/List.html`) which is implemented by the `ArrayList` and `LinkedList` classes. For example, the definition of the `ArrayList` class starts out:

```
class  ArrayList < T >   implements  List < T >
```

and similarly for the `LinkedList` class.

For each method defined in the `List` interface, there is an implemented method in the `ArrayList` and `LinkedList` class. The `List<T>` interface includes familiar method signatures such as:

- `void add(T o)`
- `void add(int index, T element)`
- `boolean isEmpty()`
- `T get(int index)`
- `T remove(int index)`
- `int size()`

Since `ArrayList<T>` and `LinkedList<T>` implement this interface, they must implement all the methods in this interface. These two classes have other methods as well, which are not defined in the `List` interface.

Why is the `List` interface useful? Sometimes you may wish to write a program that uses either an `ArrayList<T>` or a `LinkedList<T>` but you may not care which. You want to be flexible, so that the code will work regardless of which type is used. In this case, you can use the generic Java interface `List<T>`. For example,

```
void myMethod( List<String> list ){
   :
   list.add("hello");
   :
}
```

Java allows you to do this. The compiler will see the `List` type in the parameter, and it will infer that the argument passed to this method will be an object of some class that implements the `List` interface. As long as `list` only invokes methods from the `List` interface, the compiler will not complain. I will say more about this works at the end of the course when we discuss object oriented design in Java.

Java Stack and Queue

Java has a `Stack` class `https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html` It has the usual stack methods (push, pop) as well as common ones that make it more useful (isEmpty and peek). It also has one which is not stacklike, namely `search`. This method checks the stack for a particular object and if it is there then it returns the position in the stack. (Frankly I don't know why the people who invented the Java language chose to include this non-stack method, but they must have had their good reason.)

Java does not have a `Queue` class. Rather, `Queue` is an interface `https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html`

which is implemented by several different queue classes. It has methods for enqueueing and dequeueing but these are given other names than *enqueue* and `dequeue`. In the lecture I used the terms *enqueue* and `dequeue` since they are traditional in CS.

I mention the Java `Stack` class and `Queue` interface for your curiosity. We will not be discussing them further in the course.