Stacks and Queues

Contents

- Introduction
  - Stacks
  - Queues
- Implementing Stacks
  - Array Implementation
    - Test Yourself #1
    - Test Yourself #2
    - Test Yourself #3
  - Linked-list Implementation
    - Test Yourself #4
    - Test Yourself #5
- Implementing Queues
  - Array Implementation
  - Linked-list Implementation
- Comparison of Array and Linked-List Implementations
- Applications of Stacks and Queues
- Test Yourself #6

Introduction

Both Stacks and Queues are like Lists (ordered collections of items), but with more restricted operations. They can both be implemented either using an array or using a linked list to hold the actual items.

Stacks

The conceptual picture of a Stack ADT is something like this:

```
          --------      ----------
         /       \      /         \
values in       \/    /          \/
                | -----  |     values out
                |        |
                | -----  |
                |        |
                | -----  |
                |        |
                |        |
                ----------
```

Think of a stack of newspapers or trays in a cafeteria. The only item that can be taken out (or even seen) is the ***most recently added*** (or **top**) item; a Stack is a **Last-In-First-Out** (**LIFO**) abstract data type.

Here are the Stack ADT operations:

Operation	Description
boolean isEmpty()	return true iff the Stack is empty
void push(E ob)	add ob to the top of the Stack
E pop()	remove and return the item from the top of the Stack (error if the Stack is empty)
E peek()	return the item that is on the top of the Stack, but do not remove it (error if the Stack is empty)

In Java we create the `StackADT` interface as:

```
public interface StackADT<E> {
    boolean isEmpty();
    void push(E ob);
    E pop() throws EmptyStackException;
    E peek() throws EmptyStackException;
}
```

Queues

The conceptual picture of a Queue ADT is something like this:

```
                  ------------------
values in ---->   items in the queue   ----> values out
                  ------------------
                  ^                 ^
                  |                 |
     this is the rear of           this is the front of
     the queue                     the queue
```

Think of people standing in line. A Queue is a **First-In-First-Out** (**FIFO**) abstract data type. Items can only be added at the **rear** of the queue and the only item that can be removed is the one at the **front** of the queue.

Here are the Queue ADT operations:

Operation	Description
boolean isEmpty()	return true iff the Queue is empty
void enqueue(E ob)	add ob to the rear of the Queue

E dequeue()	remove and return the item from the front of the Queue (error if the Queue is empty)

In Java we create the `QueueADT` interface as:

```
public interface QueueADT<E> {
    boolean isEmpty();
    void enqueue(E ob);
    E dequeue() throws EmptyQueueException;
}
```

Implementing Stacks

The Stack ADT is very similar to the List ADT; therefore, their implementations are also quite similar.

Array Implementation

Below is the definition of the `ArrayStack` class, using an array to store the items in the stack; note that we include a static final variable `INITSIZE`, to be used by the `ArrayStack` constructor as the initial size of the array (the same thing was done for the `ArrayList` class).

```
public class ArrayStack<E> implements StackADT<E> {
  // *** fields ***
    private static final int INITSIZE = 10;  // initial array size
    private E[] items; // the items in the stack
    private int numItems;   // the number of items in the stack


  // *** constructor ***
    public ArrayStack() { ... }

  // *** required StackADT methods ***

    // add items
    public void push(E ob) { ... }

    // remove items
    public E pop() throws EmptyStackException { ... }

    // other methods
    public E peek() throws EmptyStackException { ... }
    public boolean isEmpty() { ... }
}
```

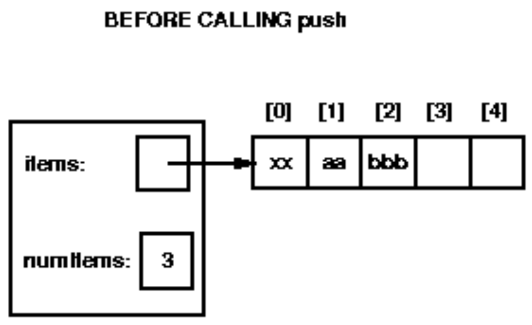
TEST YOURSELF #1

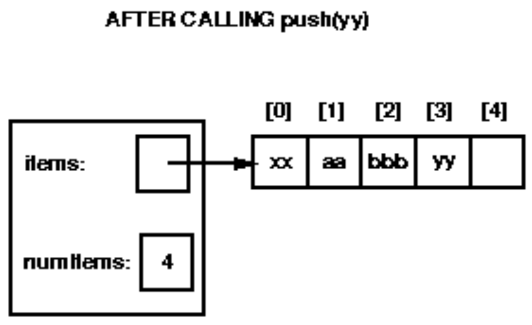
Write the `ArrayStack` constructor.

solution

The push method is like the version of the List add method that adds an object to the end of the list (because items are always pushed onto the ***top*** of the stack). Note that it is up to us as the designers of the ArrayStack class to decide which end of the array corresponds to the top of the stack. We could choose always to add items at the beginning of the array or always to add items at the end of the array. However, it is clearly not a good idea to add items at the beginning of the array since that requires moving all existing items; i.e., that choice would make push be O(N) (where N is the number of items in the stack). If we add items at the end of the array, then the time for push depends on how we handle expanding the array. The naive implementation makes push O(1) when the array is not full, O(N) when it is full, and O(1) on average. If we use the "shadow array" trick, then push is always O(1).

Here are before and after pictures, illustrating the effects of a call to push:

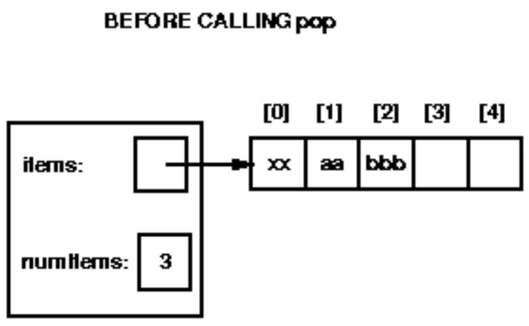


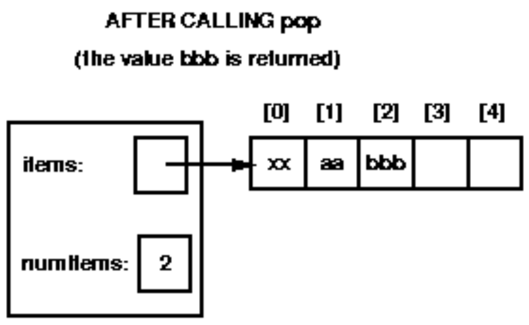


And here's the code for the push method:

```
public void push(E ob) {
    if (items.length == numItems) {
        expandArray();
    }
    items[numItems] = ob;
    numItems++;
}
```

The pop method needs to remove the top-of-stack item and return it, as illustrated below.





Note that, in the picture, the value "bbb" is still in items[2]; however, that value is no longer in the stack because numItems is 2 (which means that items[1] is the last item in the stack).

TEST YOURSELF #2

Complete the pop method, using the following header

```
public E pop() throws EmptyStackException {

}
```

solution

The peek method is very similar to the pop method, except that it only returns the top-of-stack value without changing the stack. The isEmpty method simply returns true iff numItems is zero.

TEST YOURSELF #3

Fill in the following table, using Big-O notation to give the worst and average-case times for each of the `ArrayStack` methods for a stack of size N.

Operation	Worst-case Time	Average-case Time
constructor		
isEmpty		
push		
pop		
peek		

solution

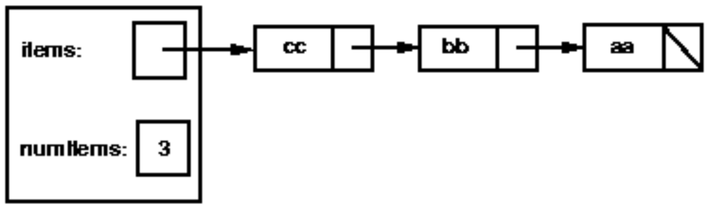
Linked-list Implementation

To implement a stack using a linked list, we must first define the `Listnode` class. The `Listnode` definition is the same one we used for the linked-list implementation of the `LinkedList` class.

The signatures of the methods of the `StackADT` interface are independent of whether the stack is implemented using an array or using a linked list; to implement the `StackADT` using a linked list, we'll change the name of the class implementing the stack and the type of the `items` field:

```
public class LLStack<E> implements StackADT<E> {
    private Listnode<E> items;  // pointer to the linked list of items in the stack
    ....
```

As discussed above, an important property of stacks is that items are only pushed and popped at one end (the top of the stack). If we implement a stack using a linked list, we can choose which end of the list corresponds to the top of the stack. It is easiest and most efficient to add and remove items at the front of a linked list, therefore, we will choose the front of the list as the top of the stack (i.e., the `items` field will be a pointer to the node that contains the top-of-stack item). Below is a picture of a stack represented using a linked list; in this case, items have been pushed in alphabetical order, so "cc" is at the top of the stack:



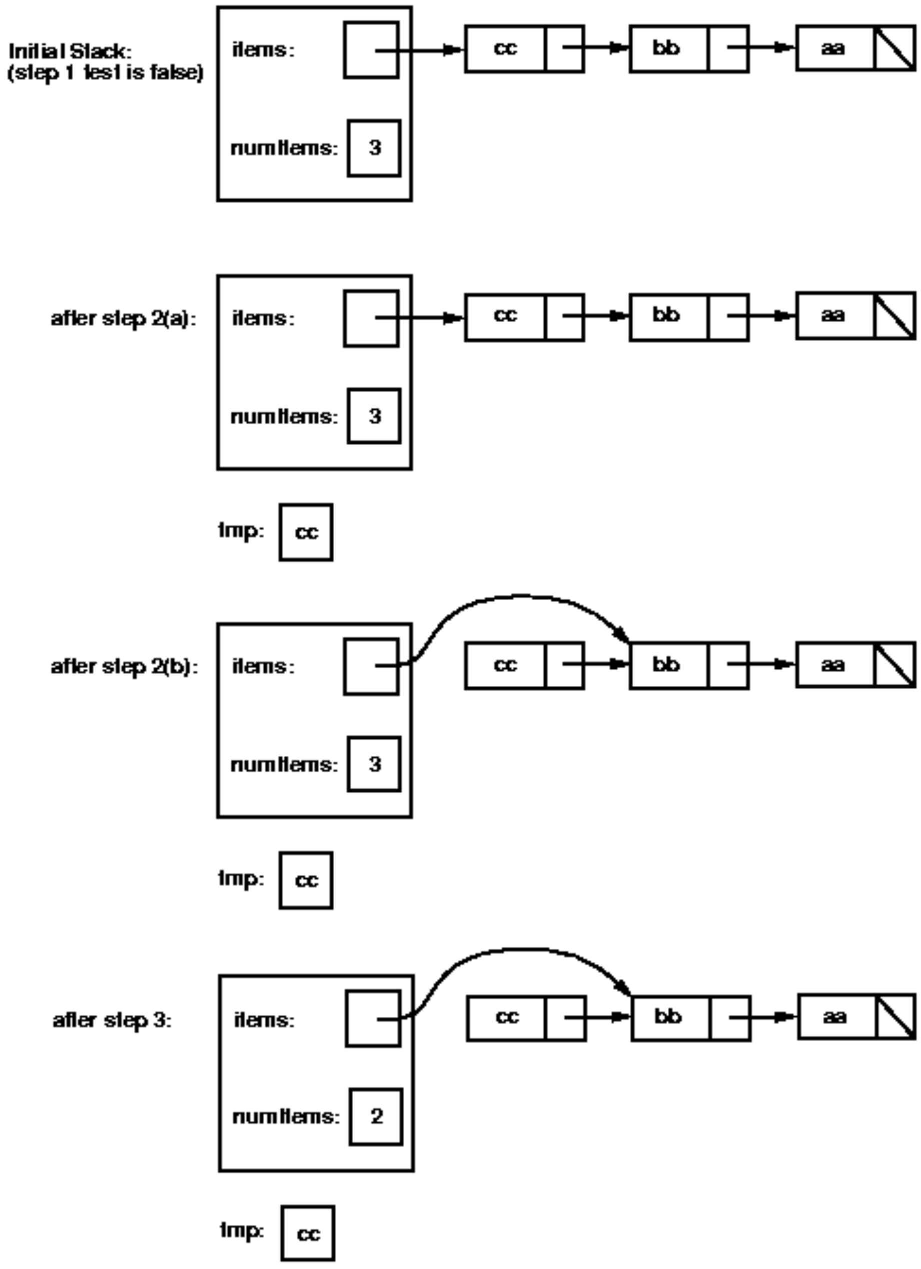
Notice that, in the picture, the top of stack is to the left (at the front of the list), while for the array implementation, the top of stack was to the right (at the end of the array).

Let's consider how to write the pop method. It will need to perform the following steps:

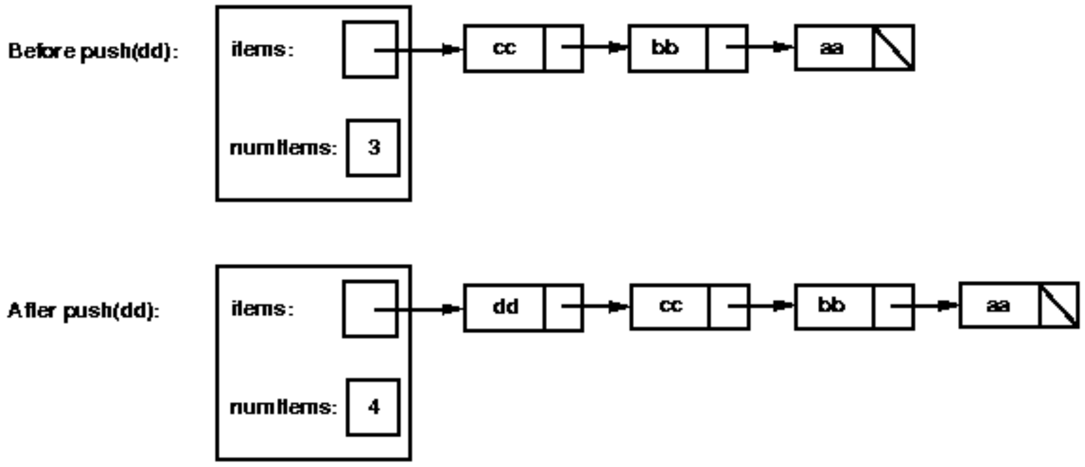
1. Check whether the stack is empty; if so, throw an `EmptyStackException`.
2. Remove the first node from the linked list by setting `items = items.getNext()`.
3. Decrement `numItems`.
4. Return the value that was in the first node in the list.

Note that by the time we get to the last step (returning the top-of-stack value), the first node has already been removed from the list, so we need to save its value in order to return it (we'll call that step 2(a)). Here's the code and an illustration of what happens when `pop` is called for a stack containing "cc", "bb", "aa" (with "cc" at the top).

```
public E pop() throws EmptyStackException {
    if (isEmpty())
        throw new EmptyStackException();  // step 1
    E tmp = items.getData();              // step 2(a)
    items = items.getNext();              // step 2(b)
    numItems--;                           // step 3
    return tmp;                           // step 4
}
```



Now let's consider the `push` method. Here are before and after pictures, illustrating the effect of a call to `push` when the stack is implemented using a linked list:



The steps that need to be performed are:

1. Create a new node whose data field contains the object to be pushed and whose next field contains a pointer to the first node in the list (or null if the list is empty). Note that the value for the next field of the new node is the value in the `LLStack`'s items field.
2. Change `items` to point to the new node.
3. Increment `numItems`.

TEST YOURSELF #4

Complete the `push` method, using the following header.

```
public void push(E ob) {

}
```

solution

The remaining methods (the constructor, peek, and empty) are quite straightforward. You should be able to implement them without any major problems.

TEST YOURSELF #5

Fill in the following table, using Big-O notation to give the worst-case times for each of the `LLStack` methods for a stack of size N, assuming a linked-list implementation. Look back at the table you filled in for the array implementation. How do the times compare? What are the advantages and disadvantages of using an array vs using a linked list to implement the Stack ADT?

Operation	Worst-case Time
constructor	
isEmpty	
push	
pop	
peek	

solution

Implementing Queues

The main difference between a stack and a queue is that a stack is only accessed from the top, while a queue is accessed from both ends (from the rear for adding items, and from the front for removing items). This makes both the array and the linked-list implementation of a queue more complicated than the corresponding stack implementations.

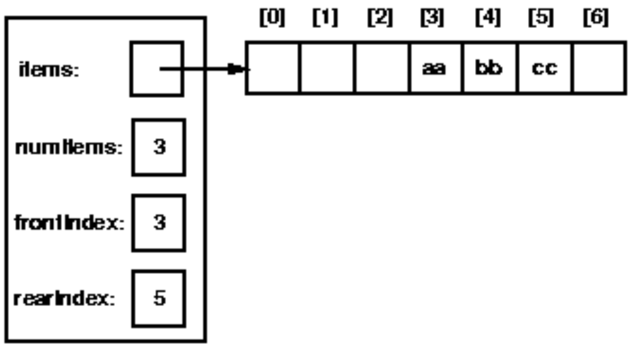
Array Implementation

Let's first consider a Queue implementation that is very similar to our (array-based) List implementation. Here's the class definition:

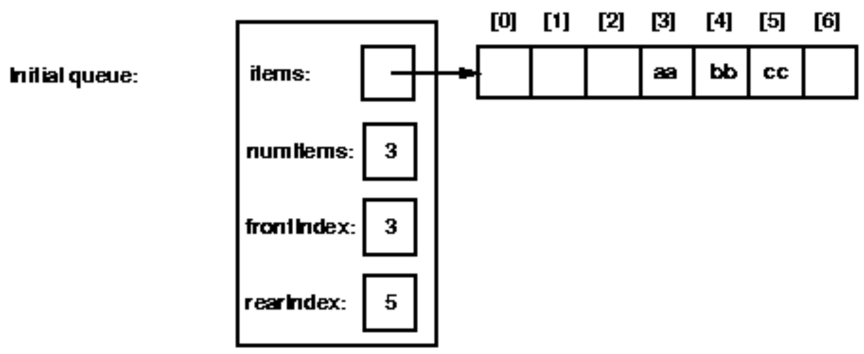
```
public class ArrayQueue<E> implements QueueADT<E> {
  // *** fields ***
    private static final int INITSIZE = 10;  // initial array size
    private E[] items; // the items in the queue
    private int numItems;   // the number of items in the queue


  //*** constructor ***
    public ArrayQueue() { ... }

  //*** required QueueADT methods ***

    // add items
    public void enqueue(E ob) { ... }

    // remove items
    public E dequeue() throws EmptyQueueException { ... }

  // other methods
    public boolean isEmpty() { ... }
}
```

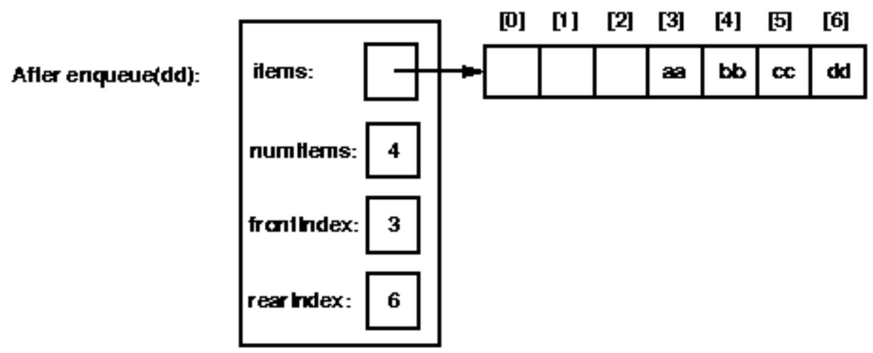
We could implement enqueue by adding the new item at the end of the array and implement dequeue by saving the first item in the array, moving all other items one place to the left, and returning the saved value. The problem with this approach is that, although the enqueue operation is efficient, the dequeue operation is not -- it requires time proportional to the number of items in the queue.

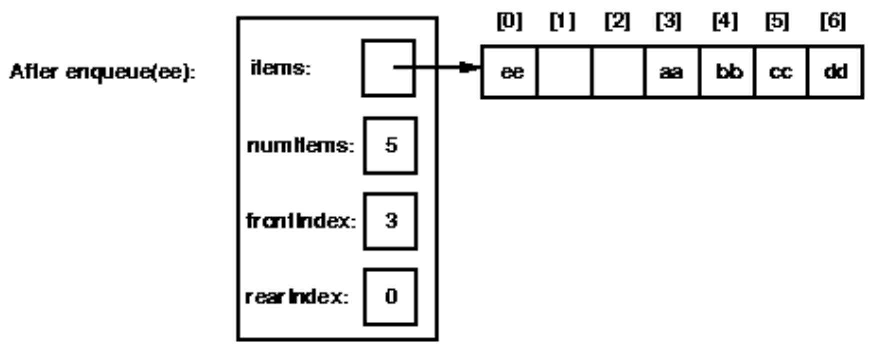
To make both enqueue and dequeue efficient, we need the following insight: There is no reason to force the front of the queue always to be in `items[0]`, we can let it "move up" as items are dequeued. To do this, we need to keep track of the indexes of the items at the front and rear of the queue (so we need to add two new fields to the `ArrayQueue` class, `frontIndex` and `rearIndex`, both of type `int`). To illustrate this idea, here is a picture of a queue after some `enqueue` and `dequeue` operations have been performed:



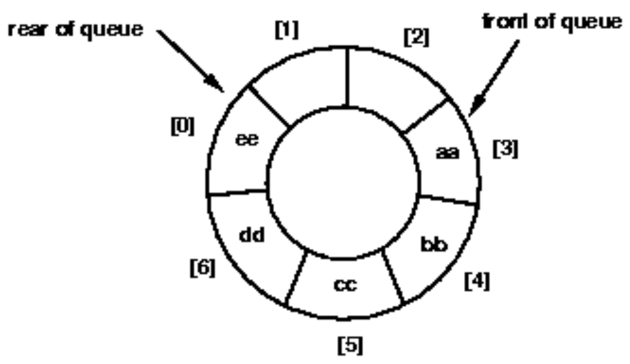
Now think about what should happen to this queue if we enqueue two more items: "dd" and "ee". Clearly "dd" should be stored in `items[6]`. Then what? We could increase the size of the array and put "ee" in `items[7]`, but that would lead to wasted space -- we would never reuse `items[0]`, `items[1]`, or `items[2]`. In general, the items in the queue would keep "sliding" to the right in the array, causing more and more wasted space at the beginning of the array. A better approach is to let the rear index "wrap around" (in this case, from 6 to 0) as long as there is empty space in the front of the array. Similarly, if after enqueuing "dd" and "ee" we dequeue four items (so that only "ee" is left in the queue), the front index will have to wrap around from 6 to 0. Here's a picture of what happens when we enqueue "dd" and "ee":







Conceptually, the array is a **circular** array. It may be easier to visualize it as a circle. For example, the array for the final queue shown above could be thought of as:

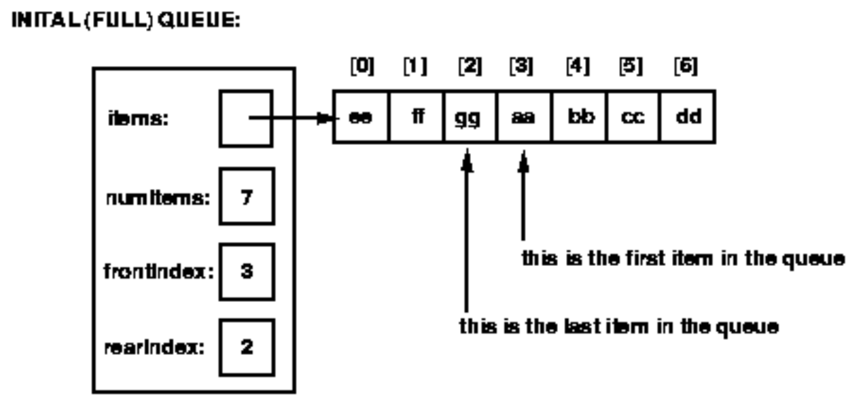


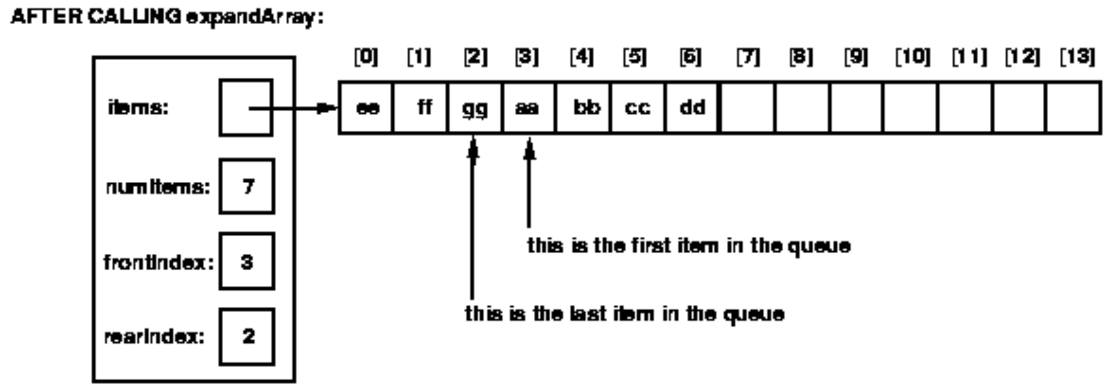
We still need to think about what should happen if the array is full; we'll consider that case in a minute. Here's the code for the `enqueue` method, with the "full array" case still to be filled in:

```
public void enqueue(E ob) {
    // check for full array and expand if necessary
    if (items.length == numItems) {
       // code missing here
    }

    // use auxiliary method to increment rear index with wraparound
    rearIndex = incrementIndex(rearIndex);

    // insert new item at rear of queue
    items[rearIndex] = ob;
    numItems++;
}

private int incrementIndex(int index) {
    if (index == items.length-1)
        return 0;
    else
        return index + 1;
}
```

Note that instead of using `incrementIndex` we could use the mod operator (`%`), and write: `rearIndex = (rearIndex + 1) % items.length`. However, the mod operator is quite slow and it is easy to get that expression wrong, so we will use the auxiliary method (with a check for the "wrap-around" case) instead.

To see why we can't simply use `expandArray` when the array is full, consider the picture shown below.



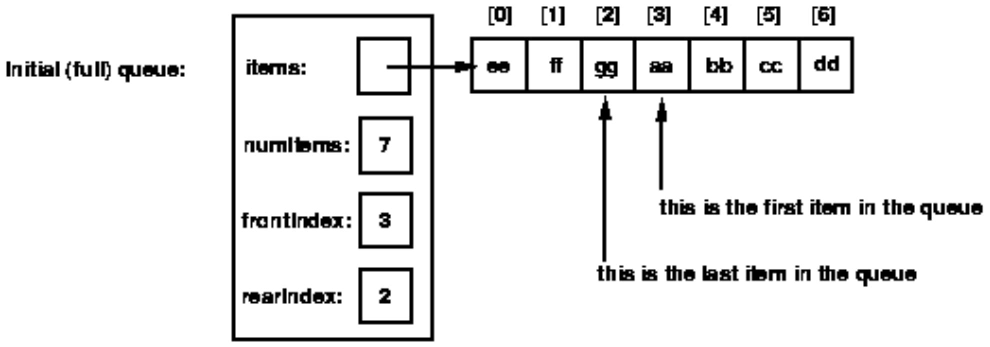


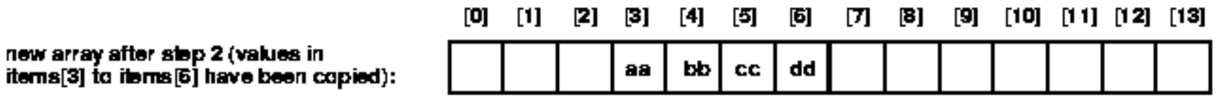
After calling `expandArray`, the last item in the queue is still right before the first item -- there is still no place to put the new item (and there is a big gap in the middle of the queue, from `items[7]` to `items[13]`). The problem is that `expandArray` copies the values in the old array into the ***same*** positions in the new array. This does not work for the queue implementation; we need to move the "wrapped-around" values to come after the non-wrapped-around values in the new array.

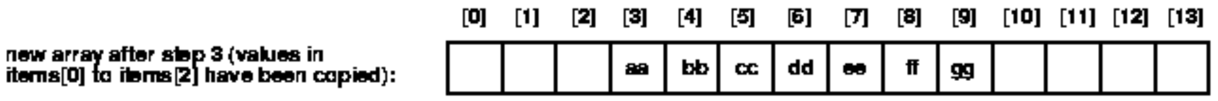
The steps that need to be carried out when the array is full are:

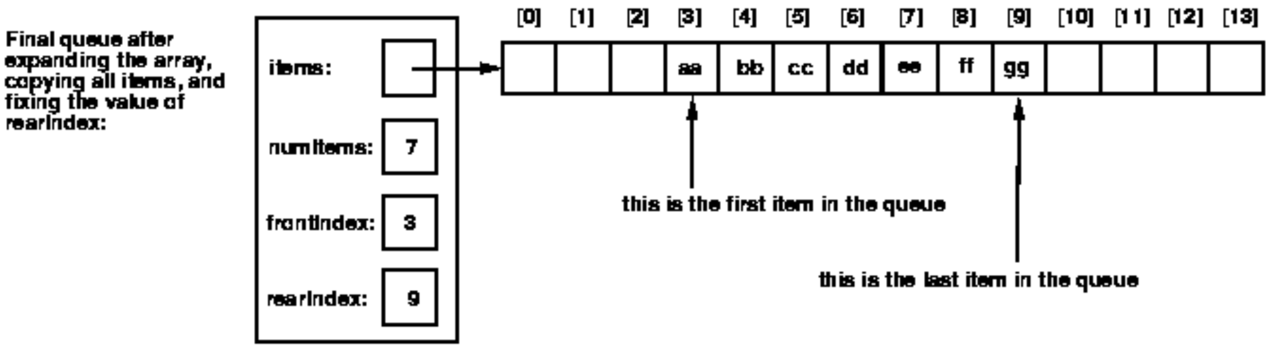
1. Allocate a new array of twice the size.
2. Copy the values in the range `items[frontIndex]` to `items[items.length-1]` into the new array (starting at position `frontIndex` in the new array).
3. Copy the values in the range `items[0]` to `items[rearIndex]` into the new array (starting at position `items.length` in the new array). Note: if the front of the queue was in `items[0]`, then all of the values were copied by step 2, so this step is not needed.
4. Set `items` to point to the new array.
5. Fix the value of `rearIndex`.

Here's an illustration:




new array after step 2 (values in items[3] to items[6] have been copied):
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]
aa bb cc dd




Final queue after expanding the array, copying all items, and fixing the value of rearIndex:
items:
numItems: 7
frontIndex: 3
rearIndex: 9
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]
aa bb cc dd ee ff gg
this is the first item in the queue
this is the last item in the queue

And here's the final code for `enqueue`:

```
public void enqueue(E ob) {
    // check for full array and expand if necessary
    if (items.length == numItems) {
        E[] tmp = (E[])(new Object[items.length*2]);
        System.arraycopy(items, frontIndex, tmp, frontIndex,
                         items.length-frontIndex);
        if (frontIndex != 0) {
            System.arraycopy(items, 0, tmp, items.length, frontIndex);
        }
        items = tmp;
            rearIndex = frontIndex + numItems - 1;
    }

    // use auxiliary method to increment rear index with wraparound
    rearIndex = incrementIndex(rearIndex);
```

```
        // insert new item at rear of queue
        items[rearIndex] = ob;
        numItems++;
    }
```

The `dequeue` method will also use method `incrementIndex` to add one to `frontIndex` (with wrap-around) before returning the value that was at the front of the queue.

The other `ArrayQueue` method, `isEmpty`, is the same as for the `ArrayStack` class -- it just uses the value of the `numItems` field.

Linked-list Implementation

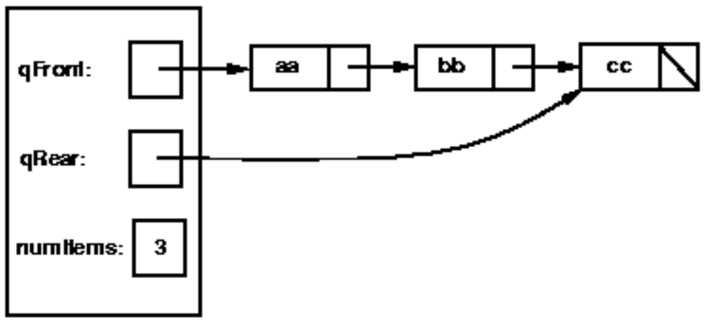
The first decision in planning the linked-list implementation of the `Queue` ADT is which end of the list will correspond to the front of the queue. Recall that items need to be added to the rear of the queue and removed from the front of the queue. Therefore, we should make our choice based on whether it is easier to add/remove a node from the front/end of a linked list.

If we keep pointers to both the first and last nodes of the list, we can add a node at either end in constant time. However, while we can remove the first node in the list in constant time, removing the last node requires first locating the ***previous*** node, which takes time proportional to the length of the list. Therefore, we should choose to make the end of the list be the rear of the queue and the front of the list be the front of the queue.

The class definition is the similar to the array implementation,:

```
public class LLQueue<E> implements QueueADT<E> {
  // *** fields ***
    private Listnode<E> qFront; // pointer to the front of the queue
                                // (the first node in the list)
    private Listnode<E> qRear;  // pointer to the rear of the queue
                                // (the last node in the list)
    private int numItems;       // the number of items in the queue
```

Here's a picture of a queue with three items, aa, bb, cc, with aa at the front of the queue:



You should be able to write all of the `LLQueue` methods using the code you wrote for the linked-list implementation of the List ADT as a guide.

Comparison of Array and Linked-List Implementations

The advantages and disadvantages of the two implementations are essentially the same as the advantages and disadvantages in the case of the List ADT:

- In the linked-list implementation, one pointer must be stored for every item in the stack/queue, while the array stores only the items themselves.
- On the other hand, the space used for a linked list is always proportional to the number of items in the list. This is not necessarily true for the array implementation as described: if a lot of items are added to a stack/queue and then removed, the size of the array can be arbitrarily greater than the number of items in the stack/queue. However, we could fix this problem by modifying the pop/dequeue operations to shrink the array when it becomes too empty.
- For the array implementation, the worst-case times for the push and enqueue methods are O(N) for the naive implementation, for a stack/queue with N items (to allocate a new array and copy the values); using the "shadow array" trick, those two operations are O(1). For the linked-list implementation, push and enqueue are always O(1).

Applications of Stacks and Queues

Stacks are used to manage methods at runtime (when a method is called, its parameters and local variables are pushed onto a stack; when the method returns, the values are popped from the stack). Many parsing algorithms (used by compilers to determine whether a program is syntactically correct) involve the use of stacks. Stacks can be used to evaluate arithmetic expressions (e.g., by a simple calculator program) and they are also useful for some operations on **graphs**, a data structure we will learn about later in the semester.

Queues are useful for many simulations and are also used for some operations on graphs and trees.

TEST YOURSELF #6

Complete method `reverseQ`, whose header is given below. Method `reverseQ` should use a Stack to reverse the order of the items in its Queue parameter.

```
public static<E> void reverseQ(QueueADT<E> q) {
// precondition: q contains x1 x2 ... xN (with x1 at the front)
// postcondition: q contains xN ... x2 X1 (with xN at the front)
}
```

solution

Back

Next: Continue with Trees (intro)