

## Ex. 3.2

After pushing two numbers on the stack and specifying three pops, I got the following output:

```
dmitristanchevici@Dmitris-iMac module3 % java MyStackExample2
Exception in thread "main" java.util.EmptyStackException
    at java.base/java.util.Stack.peek(Stack.java:103)
    at java.base/java.util.Stack.pop(Stack.java:85)
    at MyStackExample2.main(MyStackExample2.java:16)
```

## Ex. 3.5

We need the second condition in

```
if ( (unbalanced) || ( ! stack.isEmpty() ) )
```

because it is this condition that ensures that the number of the opening parens '(', which are pushed on the stack, matches the number of the closing parens ')'. If the stack is not empty after the for loop, it means that the opening parens still remaining on the stack do not have matching closing parens and that the string has unbalanced parens. This situation applies to this case:

```
s = "((( )))";
```

## Ex. 3.8

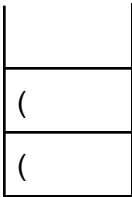
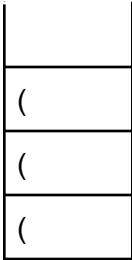
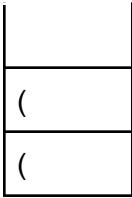
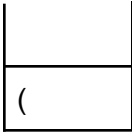
Example "[ ( ) ]" (Balanced)

i	stack
0	<div><div></div><div>(</div></div>

1	<div> <div></div> <div>[</div> <div>(</div> </div>
2	<div> <div></div> <div></div> <div>[</div> <div>(</div> </div>
3	<div> <div></div> <div>[</div> <div>(</div> </div>
4	<div> <div></div> <div>(</div> </div>
5	<div> <div></div> </div>

Example “((( )))” (Unbalanced)

i	stack
0	<div> <div></div> <div>(</div> </div>

1	
2	
3	
4	

## Ex. 3.12

After commanding more pops than there were pushes, the underflow error is

```
dmitristanchevici@Dmitris-iMac module3 % java OurStackExample
```

```
c
```

```
b
```

```
a
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of bounds
for length 100
```

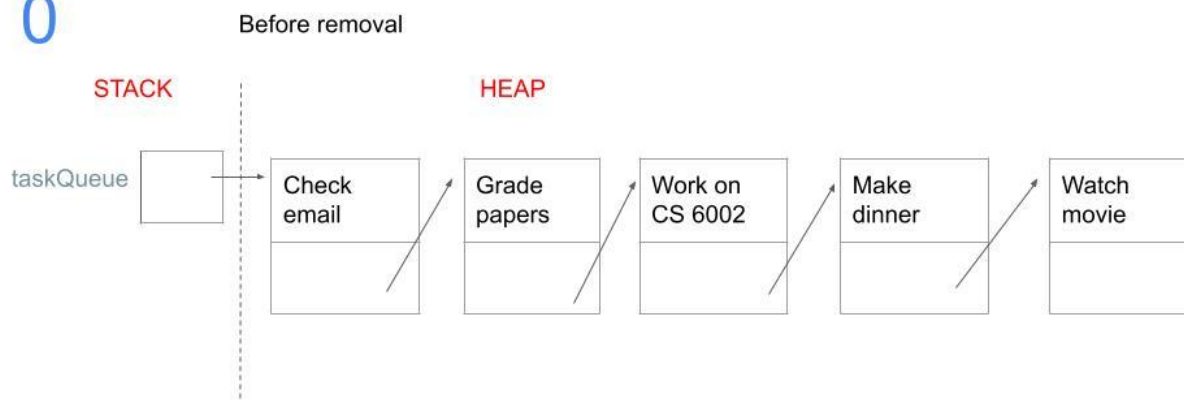
```
at OurStack.pop(OurStack.java:22)
at OurStackExample.main(OurStackExample.java:15)
```

The index in an array, which we use to implement a stack, cannot be negative, where a number of pops exceeding the number of pushes moves the index in our implementation into a negative range.

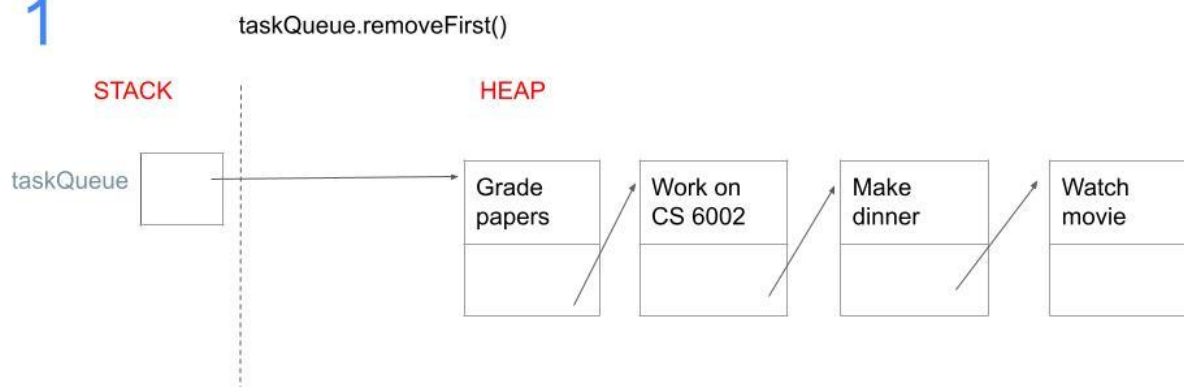
## Ex. 3.17

State of memory after each removal in QueueExample.

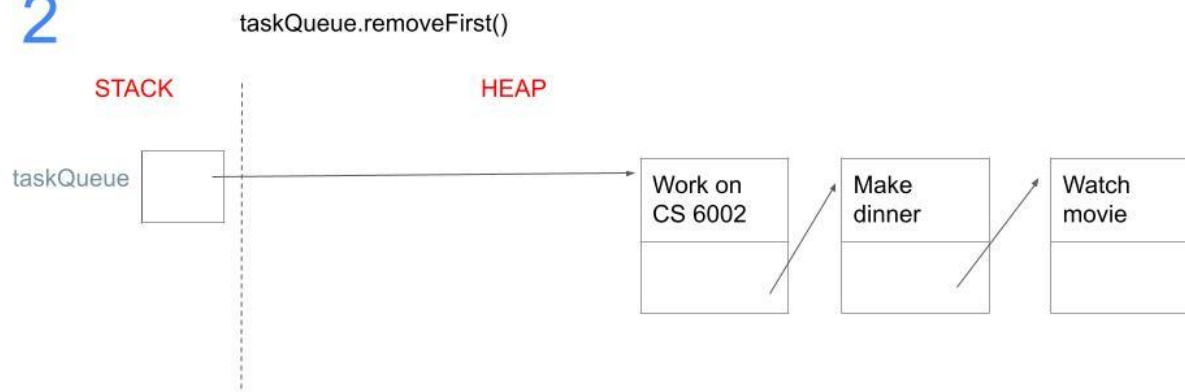
0



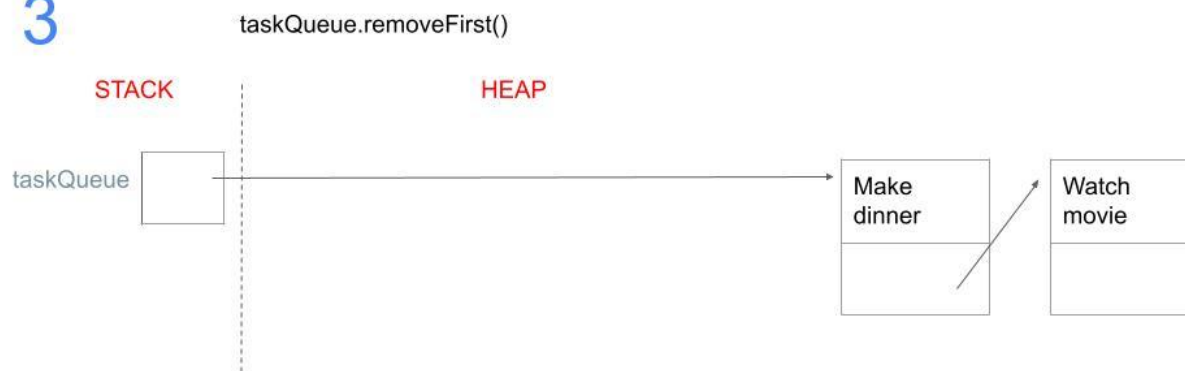
1



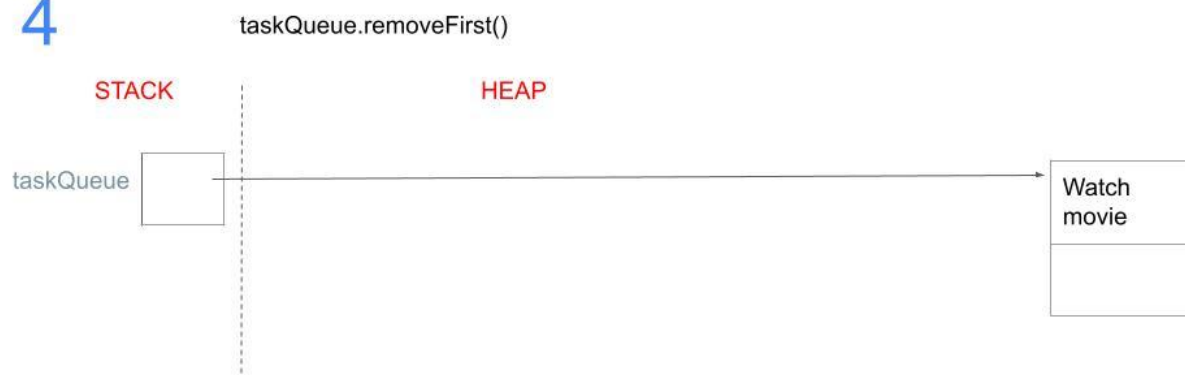
2



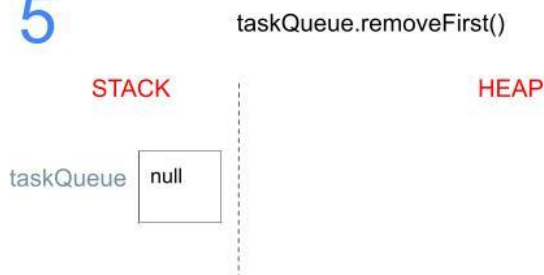
3



4



5



## Ex. 3.25

The code in the client programs, AntEater6 and AntEater7, is almost identical with the exception of the line declaring and initializing the queue, correspondingly:

```
MyQueue antQueue = new MyQueue ();
```

and

```
MyArrayQueue antQueue = new MyArrayQueue ();
```

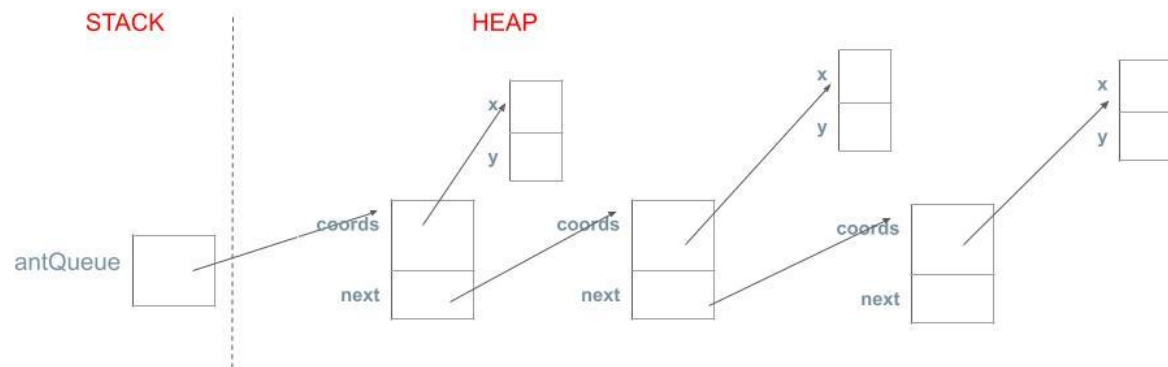
The calls to methods in antQueue are the same in both programs: antQueue.add(), antQueue.removeFirst(), etc.

The differences occur in the ways these methods are implemented, as a linked list in MyQueue and as an array in MyArrayQueue. The linked list adds and removes items (objects) dynamically. The array keeps the same reserved space of 100 items at all times, with only the values in its boxes depending on whether items are added or removed.

Interestingly, both implementations require a variable (in my case numItems) to track the number of “ants” in the queue. In the linked list this number reflects the number of actual objects with sequential pointers. In the array, this variable stands for the index of the last item + 1.

The methods in the array are easier to implement because the linkages are inherent to an array, containing values in sequentially numbered cells. The linked list is more complex because it requires creating, deleting, and reassigning links. However, the linked list has the advantage of growing dynamically, depending on the program’s needs. There is not as much chance of this list to overflow as there is in a rigidly sized array.

Memory state after adding three ants in MyQueue (based on a linked list)



Memory state after adding three ants in MyArrayQueue (based on an array)

