Algorithm Analysis
Method removeAll

Consider the following implementation of method `removeAll`:

```
void removeAll(E item)
{
    while( removeItem(item) == true ) {
        // nothing to do cycle again
    }
}
```

The main question is to decide:

> What arrangement of items in the list, i.e. what configuration, will make the method work the most. This is (*worst case scenario*).

It is also useful to consider what arrangement of items in the list, i.e. what configuration, will make the method work the least (*best case scenario*).

Best Case

The best case scenario is a bit easier. After some reflection it becomes clear that to remove all occurrences of some item at the very least we need to inspect each box to make sure no item was missed.

So, the best case configuration is when the item to delete is not present in the list at all.

In this case method `removeAll` will call method `removeItem` only one time and method `removeItem` will return `false` since the item was not found.

The next question now is how long it took method `removeItem` to decide that the given item was not in the list.

For each "box" method `removeItem` needs to:

- (in list?) check if the box exists
- (item found?) check if the box has the item to delete
- move on to the next box

This means that `removeItem` executes 3 instructions per box and since there are n boxes in the list, the total time/instructions to discover that the given item is not in the list is $3n$ or $O(n)$.

Possible Better Best Case

The following may seem like a better best case configuration, but it turns out that this is not the case.

Suppose that the item to delete occurs only one time in the list and happens to be in the front. Specifically, suppose that the goal is to remove 7 and the list looks like this:

```
7 5 5 5 .... 5     (one 7 and n-1 5s)
```

This time method `removeItem` will be called twice:

1. the first time `removeItem` will discover the item to remove, 7, is at the front of the list and will remove it returning `true` to indicate success

2. since `removeItem` returned `true` it will be called again to attempt to delete another 7; since there are no more 7s, `removeItem` will return `false`

The total time/work is the sum of the work for the two steps:

1. for step 1 the total time is the 3 instructions for the box (in list? item found? report success) plus the work to actually do the removal, say 6 instructions (adjust 4 links, adjust tail)

   in summary 9 instructions for the first call to `removeItem`

2. the second call to `removeItem` will examine $n-1$ boxes looking for 7; since there is no 7 the total work will be $3(n-1)$, i.e. $n-1$ boxes, 3 instructions per box (in list? item found? move on)

   in summary $3(n-1)$ instructions for the second call to `removeItem`

The total is: $9+3(n-1)=3n+6$, which is a bit more than the earlier analysis of *best case*, but it is still $O(n)$.

Worst Case

Here is a configuration that exhibits *worst-case* performance. Specifically, suppose that the goal is to remove 7 and the list looks like this:

```
5 5 5 ... 5 5 5 7 7 7 ... 7 7 7     (n/2 5s and n/2 7s)
```

Method `removeItem` will be called $\frac{n}{2}$ times, one time for each 7 to delete. For each 7 `removeItem` will return `true`. After the last 7 was deleted, `removeItem` will be called one final time and it will return `false` since there are no more 7s.

This is what the list looks like along the way:

```
5 5 5 ... 5 5 5 7 7 7 ... 7 7 7     original
5 5 5 ... 5 5 5 7 7 7 ... 7 7       1st 7 deleted
5 5 5 ... 5 5 5 7 7 7 ... 7         2nd 7 deleted
...
5 5 5 ... 5 5 5 7 7                 most 7s deleted
5 5 5 ... 5 5 5 7                   most 7s deleted
5 5 5 ... 5 5 5                     last 7 deleted
```

The total work depends on the work done by `removeItem` for each 7:

- for the first 7 `removeItem` will have to skip over $\frac{n}{2}$ 5s; for each 5 the work is 3 instructions (in list? item found? move on) and then 8 instructions to delete the 7 (in list? item found? delete)

  work for 1st 7: $3\frac{n}{2}+8$

- for the second 7 `removeItem` will again have to skip over $\frac{n}{2}$ 5s which again takes 3 instructions per 5 and then 8 instructions to delete the 7

  work for 2nd 7: $3\frac{n}{2}+8$

- for the third 7 `removeItem` will again have to skip over $\frac{n}{2}$ 5s which again takes 3 instructions per 5 and then 8 instructions to delete the 7

  work for 3rd 7: $3\frac{n}{2}+8$

- for the last 7 `removeItem` will again have to skip over $\frac{n}{2}$ 5s which again takes 3 instructions per 5 and then 8 instructions to delete the 7

  work for last 7: $3\frac{n}{2}+8$

In summary, it turns out that for each 7 the amount of work is $3\frac{n}{2} + 8$ and since there are $\frac{n}{2}$ 7s the total work just for deleting the 7s is:

$$num7s * workPer7 = \frac{n}{2} * (3\frac{n}{2} + 8) = \frac{3}{4}n^2 + 4n$$

Also, need to add the work for the very last call to `removeItem`, i.e. the call that finds that there are no 7s and returns `false`. On the final call, `removeItem` is called on a list of $\frac{n}{2}$ 5s, and it was already established that the total work would be $3\frac{n}{2}$ (3 instructions per 5)

Finally, the total work is:

$$delete7s + checkNo77 = \frac{3}{4}n^2 + 4n + 3\frac{n}{2} = 3\frac{n^2}{4} + \frac{11}{2}n \qquad O(n^2)$$

Another Bad Case

Here is another configuration that exhibits *worst-case* performance. Specifically, suppose that the goal is to remove 7 and the list looks like this:

```
5 7 5 7 5 7 .... 5 7 5 7 5 7     (n/2 5s and n/2 7s)
```

Method `removeItem` will be called $\frac{n}{2}$ times, one time for each 7 to delete. For each 7 `removeItem` will return `true`. After the last 7 was deleted, `removeItem` will be called one final time and it will return `false` since there are no more 7s.

This is what the list looks like along the way:

```
5 7 5 7 5 7 .... 5 7 5 7 5 7 5     original
5 5 7 5 7 .... 5 7 5 7 5 7 5       1st 7 deleted
5 5 5 7 .... 5 7 5 7 5 7 5         2nd 7 deleted
...
5 5 5 .... 5 5 7 5 7 5             most 7s deleted
5 5 5 .... 5 5 5 7 5               most 7s deleted
5 5 5 .... 5 5 5 5                 last 7 deleted
```

The total work depends on the work done by `removeItem` for each 7. This time the number of 5s to skip varies:

Since the work to skip a single 5 is 3 instructions (in list? item found? move on) and the work to delete a single 7 is 8 instructions, the total work per 7 is:

- for 1st 7 need to skip 1 5s, then delete 7 1*3 + 8
- for 2nd 7 need to skip 2 5s; then delete 7 2*3 + 8
- for 3rd 7 need to skip 3 5s; then delete 7 3*3 + 8
- ...
- for kth 7 need to skip k 5s; then delete 7 k*3 + 8
- ...
- for $\frac{n}{2}$th 7 need to skip $\frac{n}{2}$ 5s; then delete 7 $\frac{n}{2}$*3 + 8

The total work just for deleting the 7s is the sum of all above:

$$(1 * 3 + 8) + (2 * 3 + 8) + (3 * 3 + 8) + ... + (\frac{n}{2} * 3 + 8)$$

and after collecting similar terms:

$$(1 * 3 + 2 * 3 + 3 * 3 + ... + \frac{n}{2} * 3) + (8 + 8 + 8 + ... + 8)$$

After taking common factor and knowing that there are $\frac{n}{2}$ 8s since there are only $\frac{n}{2}$ 7s:

$$3 * (1 + 2 + 3 + ... + \frac{n}{2}) + 8\frac{n}{2}$$

Applying the summation formula for consecutive integers starting at 1:

$$3 * [\frac{n}{2}(\frac{n}{2} + 1)]/2 + 8\frac{n}{2}$$

$$\frac{3}{8}n^2 + \frac{3}{4}n + 4n$$

$$\frac{3}{8}n^2 + \frac{19}{4}n$$

Also, need to add the work for the very last call to `removeItem`, i.e. the call that finds that there are no 7s and returns `false`. On the final call, `removeItem` is called on a list of $\frac{n}{2}$ 5s, and it was already established that the total work would be $3\frac{n}{2}$ (3 instructions per 5)

Finally, the total work is:

$$delete7s + checkNo7 = \frac{3}{8}n^2 + \frac{19}{4}n + 3\frac{n}{2} = \frac{3}{8}n^2 + \frac{25}{4}n \qquad O(n^2)$$

Exercise

What is the complexity, i.e. amount of total work for the following configuration. Specifically, suppose that the goal is to remove 7 and the list looks like this:

```
7 7 7 ... 7 7 7 5 5 5 ... 5 5 5     (n/2 7s and n/2 5s)
```