# Debugging, Big O, Lists

*Quote of the week: "If we are indeed in as bad a state as I take us to be, pessimism will turn out to be one more cultural luxury that we shall have to dispense with in order to survive in these hard times."*

# You have an exam coming up

- Friday, 7 - 9pm
  - Sections 101 - 103: Go to 145 Dwinelle
  - Sections 104 - 109: Go to 155 Dwinelle
- It will cover all the labs, readings, and lectures that happen before it
  - With the exception of runtime analysis. No runtime analysis will be on the exam
- You can bring a *one-sided* 8.5″ x 11″ cheat sheet

# Did you enjoy project 1?

- Debugging it may have been frustrating.
- I'd like to explicitly go over techniques for debugging.
- Why not, right?

# How code can be broken

1. Your code won't even compile — Easiest to debug
2. Your code compiles, but crashes at runtime (throws an exception) — Medium to debug
3. Your code runs, but returns the wrong answer — Hardest to debug

# 1. Your code won't compile

- Solution: Hover over the red underline in Eclipse, and it will tell you exactly what's wrong, often with suggested fixes
  - If you don't know what the error message means, look it up

# 1. Your code won’t compile

- Coding recommendation: ***<u>Always</u>*** make sure that your code correctly compiles

- If you write a line and it generates a compile-time error, *don’t move on until you fix the line*

- If you treat compile-time errors as soon as they occur, they should never be a major contributor of debugging time

# 2. Your code crashes at runtime

- Java gives you the name of the Exception, as well as a *stack trace*
- First, ensure you know what the Exception means. If you don't, look it up
- Next, follow the stack trace until you find the problematic line, and fix it

# Example: a stack trace

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 2
   at WordCounter.indexOf(WordCounter.java:51)
   at WordCounter.getCounts(WordCounter.java:40)
   at WordCounter.main(WordCounter.java:82)
```

- What does this mean?
- An `ArrayIndexOutOfBoundsException` occurred. Where?
- Inside the `indexOf` method, at line 51
- At the time of error, `indexOf` was being called from the `getCounts` method, line 40
- And at the time of error, `getCounts` was being called from `main`, at line 82

# Example: `NullPointerException`

- Everyone's favorite error is the `NullPointerException`
- A `NullPointerException` means one thing, and one thing only
- We have a null expression that we are trying to call a method or get an instance variable from

# Spot the `NullPointerException`

- Suppose a stack trace tells us we got a null pointer exception on this line

```
if (this.pangolin().wug() == capybara) {
```

- Which of the following could be the null that caused the exception?

A. this
B. the return of this.pangolin()
C. the return of this.pangolin().wug()
D. capybara
E. some variable inside pangolin() or wug()

# 3. Your code runs, but returns the wrong answer

- This is by far the hardest to debug, because you have no indication *where* the error occurs.
- In the previous two cases, Java tells you exactly where the problem was. Resolving these is simply a matter of knowing what the error means

# 3. Your code runs, but returns the wrong answer

- The first step: Search your code for *where* the bug occurs

# Search, huh?

- You're searching for a bug in your code.
- Do you know any good ways of searching?
- How about, well, *binary search*?

# Binary search?

- *I thought this was used for checking if a sorted array contained a particular number.*
- Well yes. But a similar idea applies for looking for bugs.
- *Really?*
- Yes. Maybe an example will clarify.

# Binary search demo

# Runtime analysis

- We're interested in knowing how fast our code is
- How to figure out? Timing? But it may run with different speeds on different computers, under different amounts of traffic, etc.

# Runtime analysis

- **The big idea**: Let's count the number of statements our program has to execute
- This will (roughly) approximate the runtime
- NOT the full story (see 61C, future classes). But it's a start

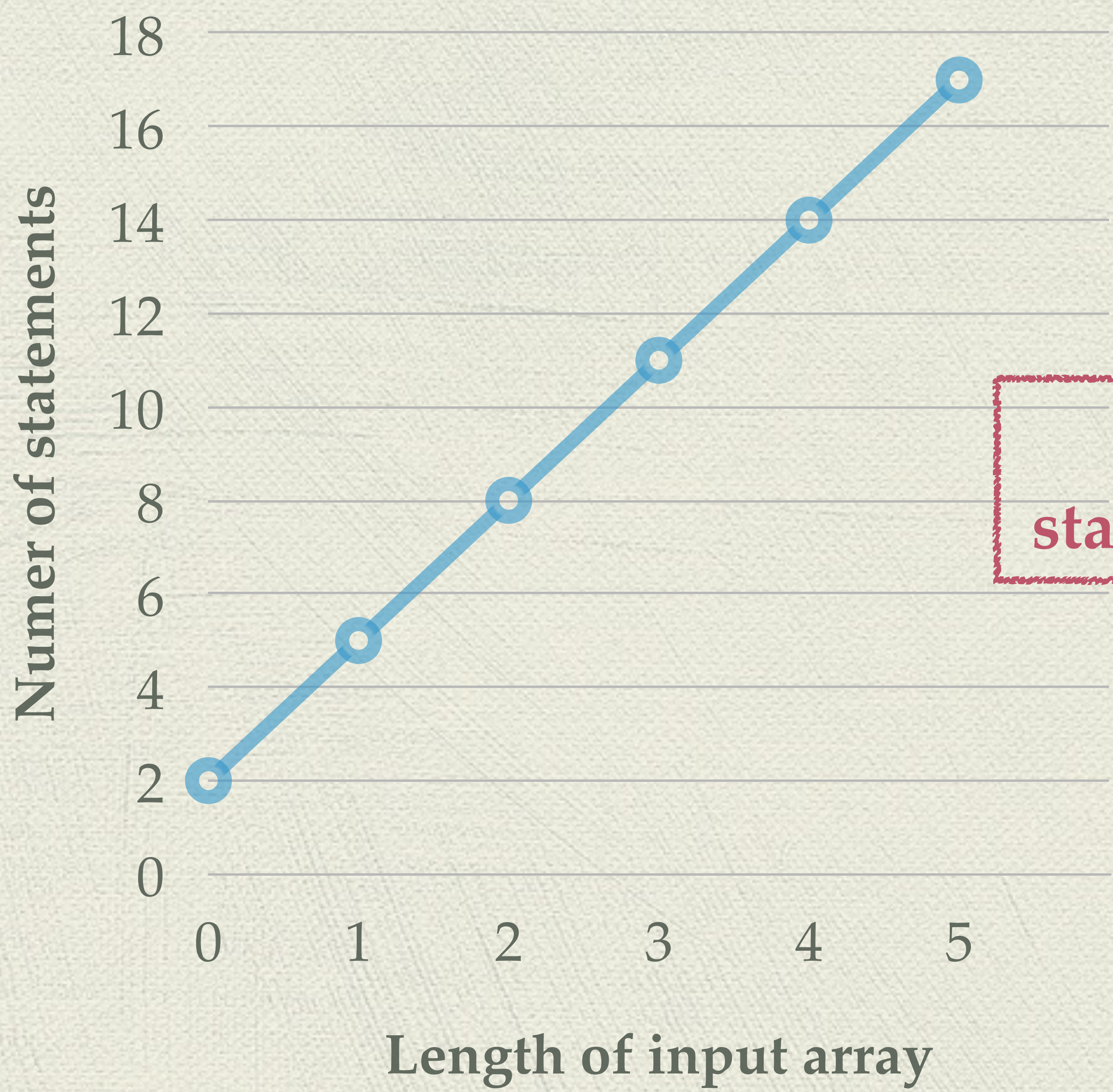
# Some observations

- **Observation 1**: The number of statements our program has to execute *varies* based on the size of the input

```
public static double min(double[] arr) {
  double minSoFar = Double.POSITIVE_INFINITY;
  for (double item : arr) {
    if (item < minSoFar) {
      minSoFar = item;
    }
  }
  return minSoFar;
}
```

Takes longer depending on how many items `arr` has in it!

- So, runtime is *a function of the input size*

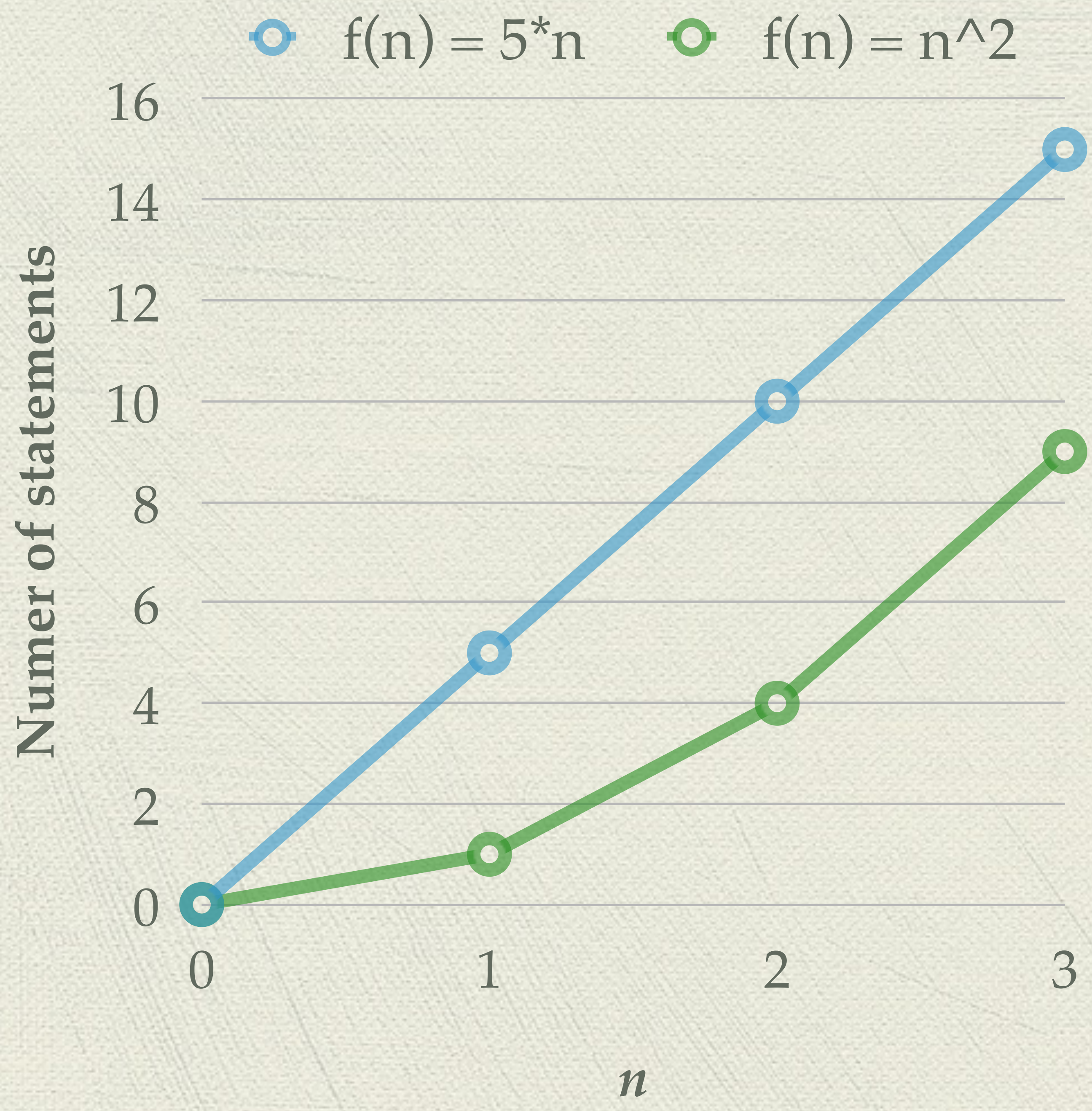



**Graph of the function**
**statements(*length*) = 3 * *length* + 2**

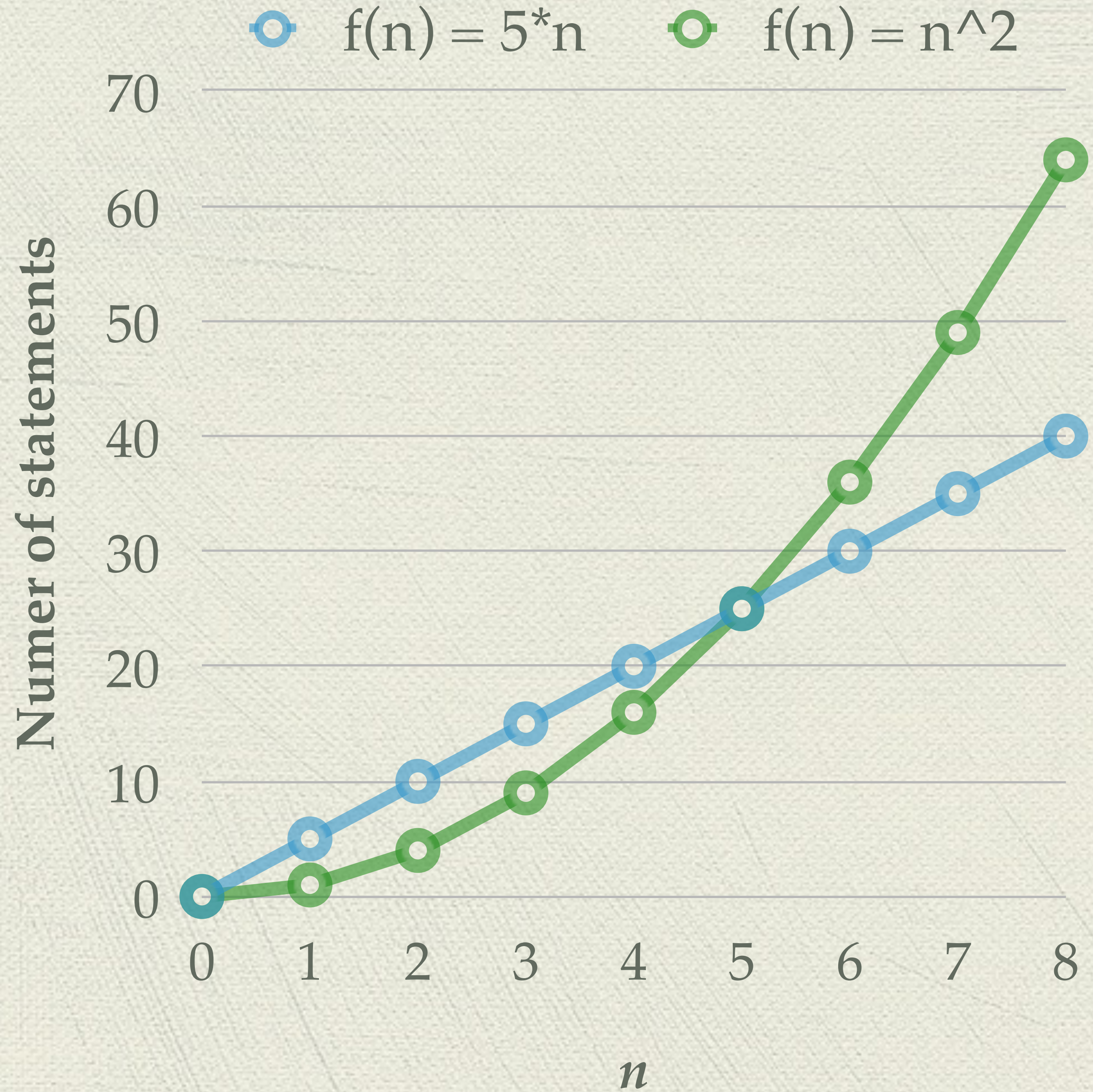
# Some observations

- **Observation 2**: It doesn't matter how fast our program runs for small input
- The only reason computer science is useful is to solve large problems
- We could have just solved it by hand, otherwise…

- Which one is algorithm is better?
- Remember, more statements is *worse*



- Trick question!
- Remember, what matters is what happens when *n* is BIG



# Some observations

- **Observation 3**: It's kind of annoying to count *every single statement* in the program

```
public static double min(double[] arr) {
   double minSoFar = Double.POSITIVE_INFINITY;
   for (double item : arr) {
      if (item < minSoFar) {
         minSoFar = item;
      }
   }
   return minSoFar;
}
```

- Do I really have to count the first line? It barely takes any time at all. Where the real work is done is in the loop

- It'd be better to *approximate* the number of statements

# Some observations

- **Observation 1**: Runtime is a function of the input size
- **Observation 2**: What matters is how this function behaves when input gets really big
- **Observation 3**: We don't need an exact function. Only an approximate function

# Our solution: Big Θ notation

- Say you have a function called $g(n)$. Probably represents the runtime of a program based on the input size, $n$
- We have a notation $\Theta(g(n))$
- This thing is a set of functions that grow similarly to $g(n)$

# Our solution: Big Θ notation

- Suppose in reality, the runtime of our program can be represented by $f(n) = 2n + 3$
- We might claim this grows similarly to the simpler function, $g(n) = n$
- The notation to claim this is
  - $f(n)$ is in $\Theta(g(n))$
  - $2n + 3$ is in $\Theta(n)$

# The big picture

- We have a program like

```
public static double min(double[] arr) {
   double minSoFar = Double.POSITIVE_INFINITY;
   for (double item : arr) {
      if (item < minSoFar) {
         minSoFar = item;
      }
   }
   return minSoFar;
}
```

- And we'll say, the runtime of this program is in $\Theta(n)$
- This expresses the approximate behavior of the program as the input grows large, which is what we really care about

# Some formalism

- So what does it *really* mean to claim that some f(n) is in $\Theta(g(n))$?
- I said $g(n)$ is an approximation of $f(n)$. But what is the exact nature of the approximation?
- First: the approximation only holds when $n$ is large

# Some formalism

- f(n) is in Θ(*g(n))* if and only if
- The limit of *f(n)* and *g(n)* as *n* goes to infinity is similar, or…

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$$

- Where *c* is a **positive constant**

# What's the alternative?

- $2n + 3$ is **NOT** in $\Theta(n^2)$ because

$$\lim_{n \to \infty} \frac{2n+3}{n^2} = 0$$

- And 0 is not a **positive constant.** The point is that $2n+3$ is NOT similar to $n^2$. $n^2$ is **much bigger** as $n$ gets big

# What's the alternative?

- Similarly, $n^2$ is **NOT** in $\Theta(2n + 3)$ because

$$\lim_{n \to \infty} \frac{n^2}{2n+3} \to \infty$$

- So the limit does not equal a **positive constant.** The point is that $2n + 3$ is NOT similar to $n^2$. $n^2$ is **much bigger** as $n$ gets big

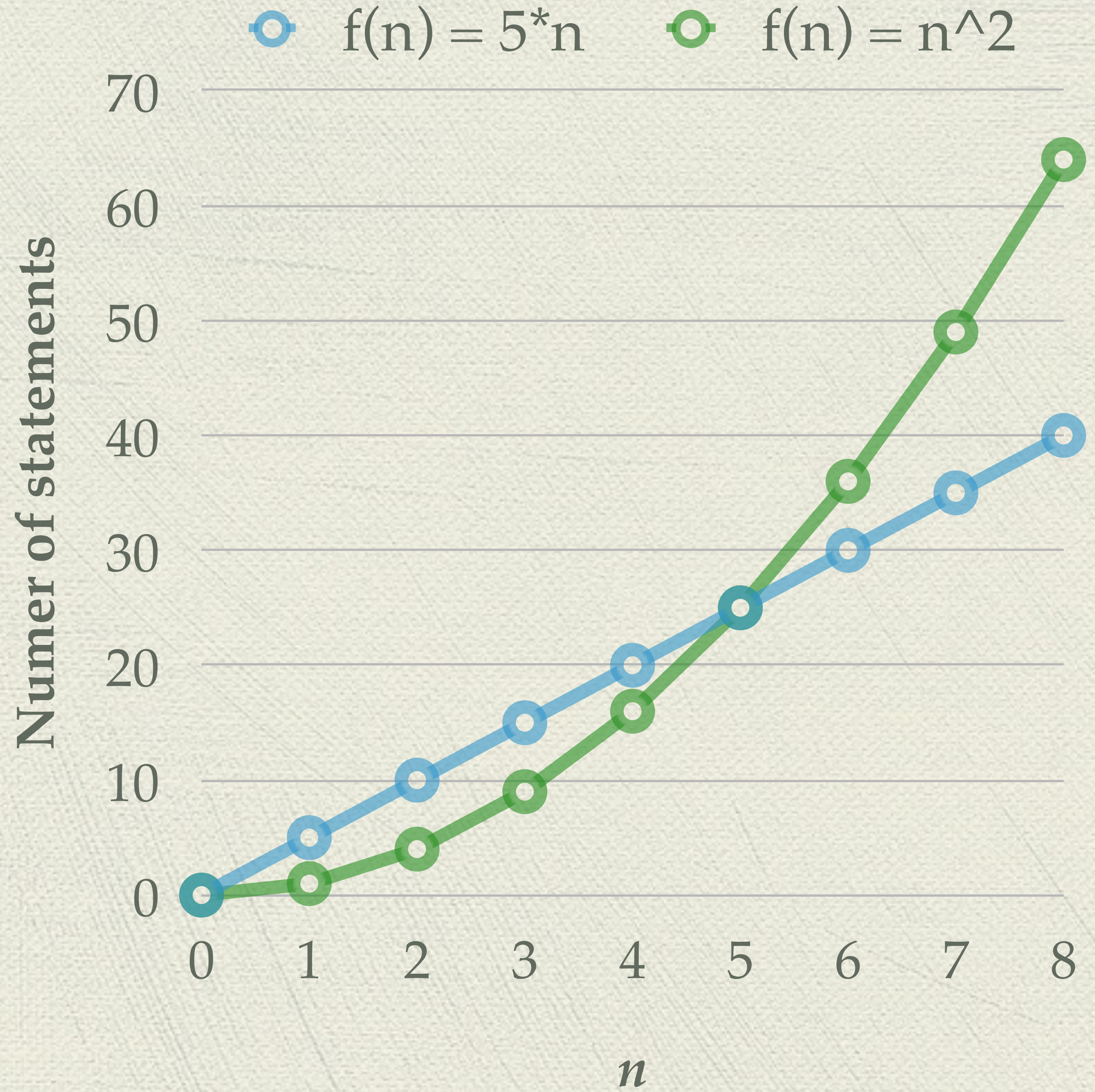
# Shortcuts

- You can always check big Θ membership using limits
- Luckily, there are a couple of shortcuts we notice

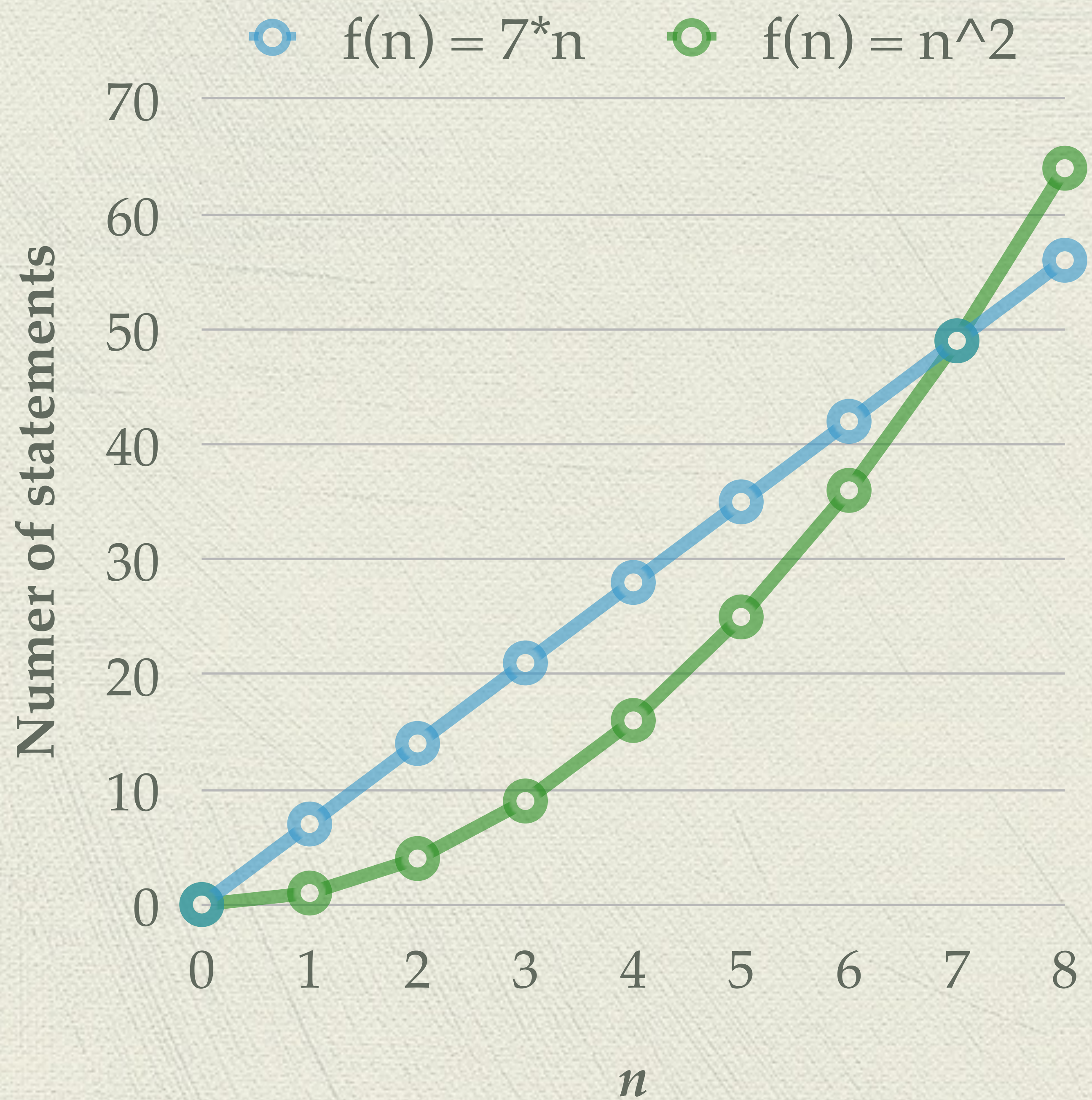
# Shortcuts

- **Shortcut 1:** Constant multiplied factors don't matter at all.
  - Ex: 100000*$n$ is in $\Theta(n)$
- **Shortcut 2:** When you have a *sum* of terms, only the term with the highest power matters
  - Ex: $n^5 + n^3 + n + 1$ is in $\Theta(n^5)$

- Only the high term matters…
- A function with $n^2$ will **always** overtake a function with only $n$



- Only the high term matters…
- A function with $n^2$ will **always** overtake a function with only $n$



# Shortcuts

- There are other kinds of terms than polynomials that matter. Here are some common ones:
  - Logs: $log(n)$
  - Exponentials: $2^n$, $3^n$, $4^n$, …
  - Factorials: $n!$

# Shortcuts

- Logs are always smaller than polynomials
- Polynomials are always smaller than exponentials
  - So $2^n + n^{10000} + 10log(n)$ is in $\Theta(2^n)$

# Shortcuts

- Multiplying non-constant terms *does* make a difference
    - $2^n * n^{10000} + n$ is **NOT** in $\Theta(2^n)$
    - It's in $\Theta(2^n * n^{10000})$
- When in doubt, check by taking limits

# By the way

- Algorithms that run in…
  - $\Theta(1)$ are called *constant time* algorithms
  - $\Theta(n)$ are called *linear* algorithms
  - $\Theta(n^2)$ are called *quadratic* algorithms
  - $\Theta(log(n))$ are called *logarithmic* algorithms
  - $\Theta(2^n)$, $\Theta(3^n)$, etc. are called *exponential* algorithms

# I’m not so sure about this Big Θ thing…

- You’re really saying that if one program has to execute *n* statements, and another has to execute 2*n* statements, they’re roughly the same?
- Even though one runs twice as fast the other other?
- *Yes, that’s what I’m saying!*

# I’m not so sure about this Big Θ thing…

- It’s true that constant factors do matter. If you can cut your program’s runtime by 2, good for you!
- But it really doesn’t matter quite as much as other orders
- $n$ and $2n$ might be the difference between waiting 1 second and 2 seconds. But $n$ and $n^2$ could be the difference between waiting 1 second and 1 hour, or worse

# I’m not so sure about this Big Θ thing…

- In other words…
- Constant factors can make the difference between a fast and slow program
- But different Θ orders can make the difference between runnable and *completely un-runnable*

# A brief demonstration

- $n$ and $n^2$ demo

# Big O and Big $\Omega$

- $f(n)$ is in $\Theta(g(n))$ if it grows similarly to $g(n)$
- $f(n)$ is in $\mathbf{O}(g(n))$ if it grows similarly to $g(n)$, or grows more slowly
    - We say $g(n)$ is an **upper bound** on $f(n)$
- $f(n)$ is in $\mathbf{\Omega}(g(n))$ if it grows similarly to $g(n)$, or grows faster
    - We say $g(n)$ is a **lower bound** on $f(n)$

# Potentially confusing point

- If $f(n)$ grows more slowly than $g(n)$, that means $f(n)$ represents a faster program!

# Oh, and why “O”?

- “O” is for <u>O</u>rder of growth
- So you might hear this term as well

# BREAK!!!

- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!
- Break!

# Let's play a game

- Guess the runtime from the code!

```java
public static void awesomeMethod(int[] arr) {

  int n = arr.length;

  for (int i = 0; i < n; i++) {
    System.out.println(arr[i]);
    for (int j = 0; j < n; j++) {
      System.out.println(arr[j]);
    }
  }
}
```

**Options:**

**A.** O($n$)

**B.** O(2$n$)

**C.** O($n^2$)

**D.** None of the above

```java
public static void wayCoolMethod(int[] arr) {

  int n = arr.length;

  for (int i = 0; i < n; i++) {
    System.out.println(arr[i]);
  }

  for (int j = 0; j < n; j++) {
    System.out.println(arr[j]);
  }
}
```

**Options:**

A. $O(n)$

B. $O(2n)$

C. $O(n^2)$

D. None of the above

```java
  public static void outrageousMethod(int[] arr1,
int[] arr2) {

    int n = arr1.length;
    int m = arr2.length;

    for (int i = 0; i < n; i++) {
      System.out.println(arr1[i]);
      for (int j = 0; j < n; j++) {
        System.out.println(arr1[j]);
      }
    }

    for (int k = 0; k < m; k++) {
      System.out.println(arr2[k]);
    }
  }
```

**Options:**

A. $O(n^2)$

B. $O(n^2 + m)$

C. $O(n^{2*} m)$

D. None of the above

```java
public static void groovyMethod(int[] arr1,
int[] arr2) {

    int n = arr1.length;
    int m = arr2.length;

    if (arr1 != arr2) {
      for (int i = 0; i < n; i++) {
        System.out.println(arr1[n]);
      }
    } else {
      System.out.println("Aren't you special?");
    }
}
```

**Options:**

**A.** $\Theta(n)$

**B.** $\Theta(n + m)$

**C.** $\Theta(n^2 + m)$

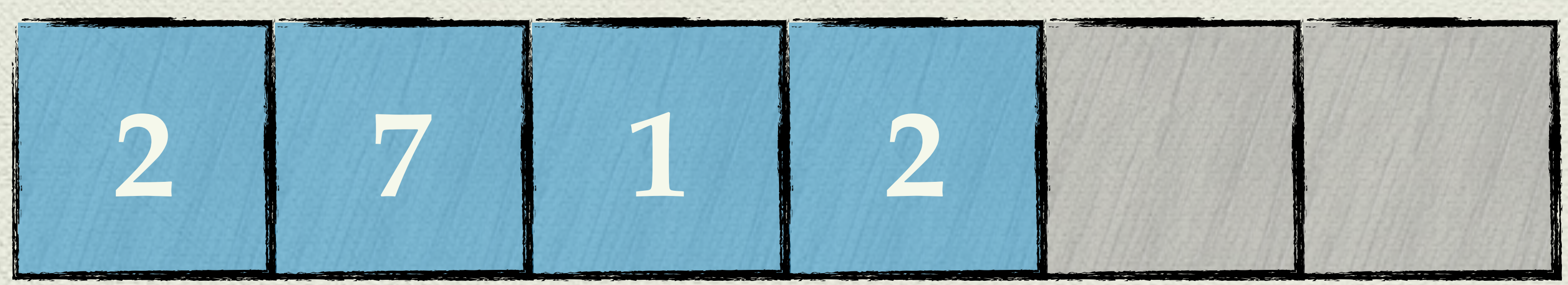
**D.** None of the above

# Lists

- We'd like a data structure to represent *sequential* data
- The array worked kinda
- The problem was its fixed size. We fixed this with our `ResizableIntSequence` class, but…

# Inserting into an array

- Here's an array that might be a part of an `IntList`. What if we want to append an item to the front? Say we want to put 8 in front.



# Inserting into an array— a sad story

- First, move everything over to make room…

# Inserting into an array— a sad story

- First, move everything over to make room…

# Inserting into an array— a sad story

- First, move everything over to make room…

# Inserting into an array— a sad story

- First, move everything over to make room…

# Inserting into an array— a sad story

- First, move everything over to make room…

# Inserting into an array— a sad story

- Phew! Finally, put the new item in the first position

# Inserting into an array — a sad story

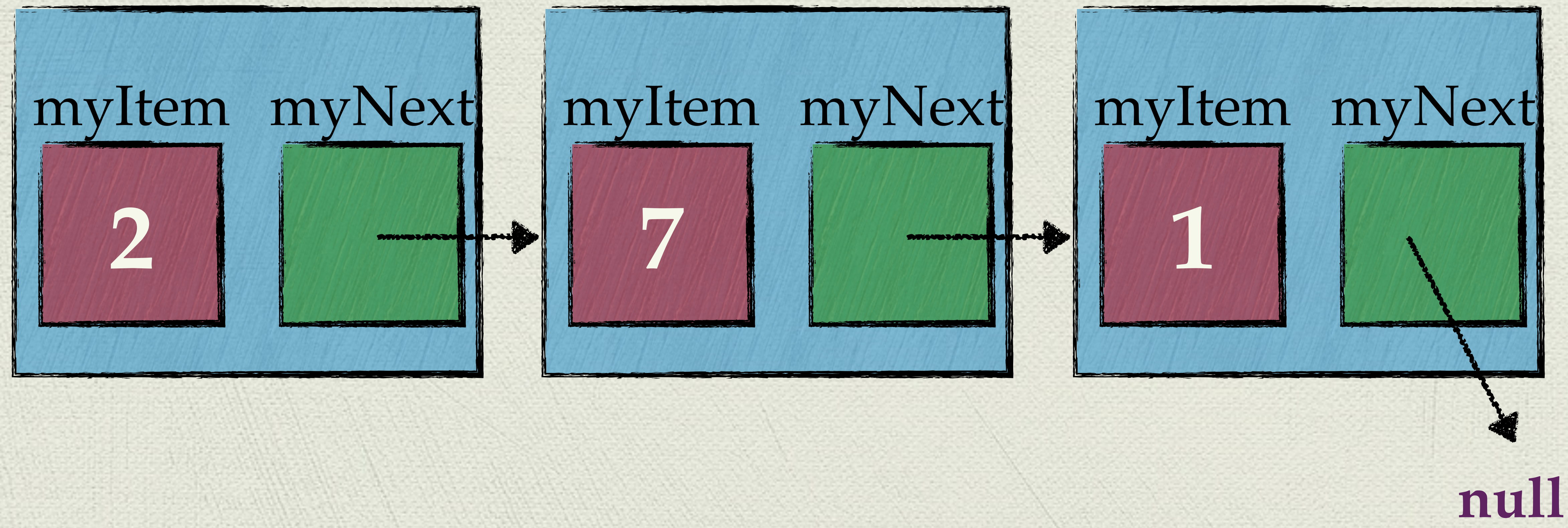
- To insert in the front, we have to move *every single other item in the array*
- Inserting a single item into a sequence of length $n$ takes worst case $O(n)$ time.
- This seems more trouble than it should be.
  - We only wanted to add one item!

# Introducing the linked list

- An alternative way to represent sequential data is using a *node-based* list, aka a *linked list*
- Each item in the list is stored in a little object, called a node
- This node also contains a reference to the next item in the list

# Introducing the linked list

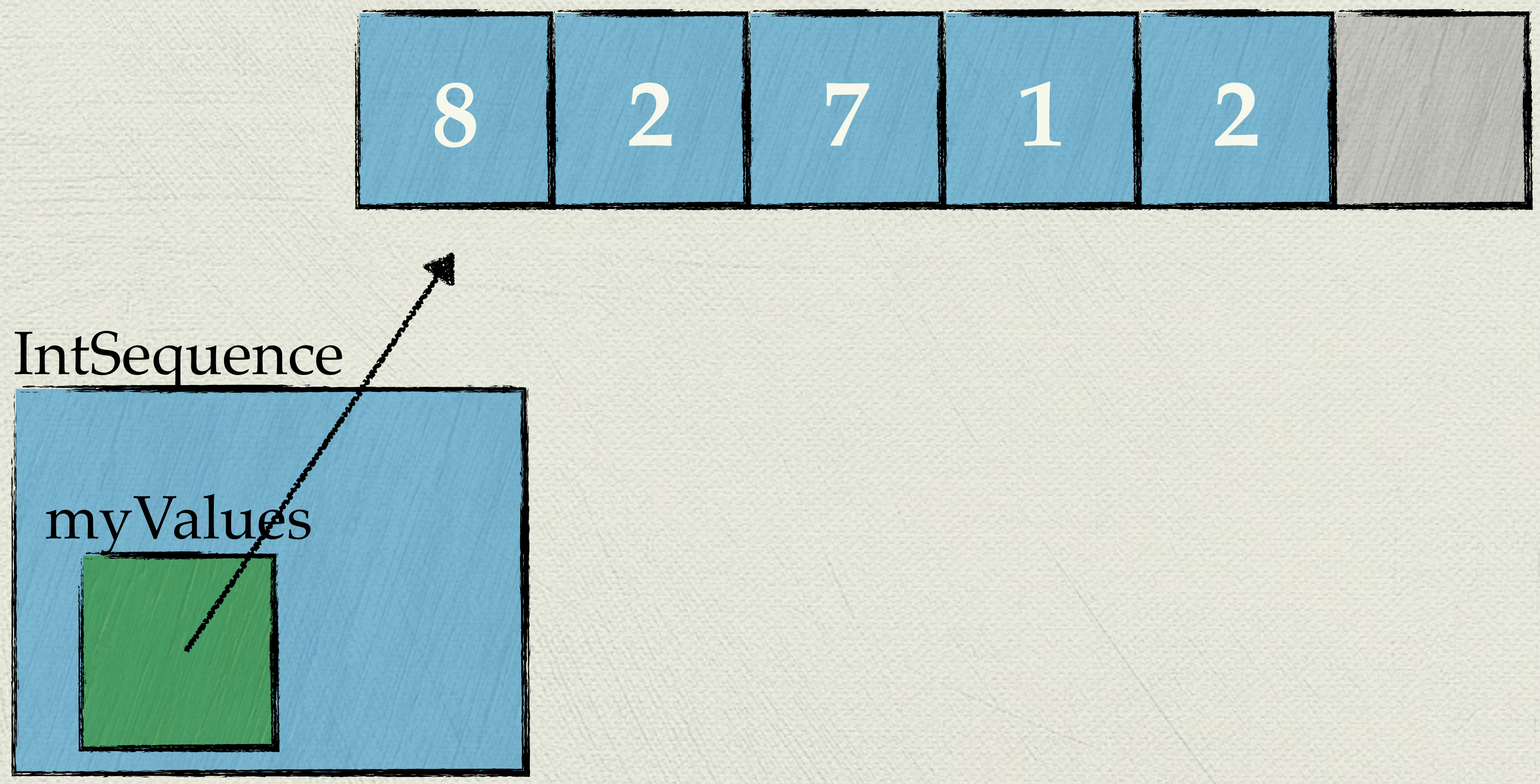
- Here's a picture of the a similar sequence



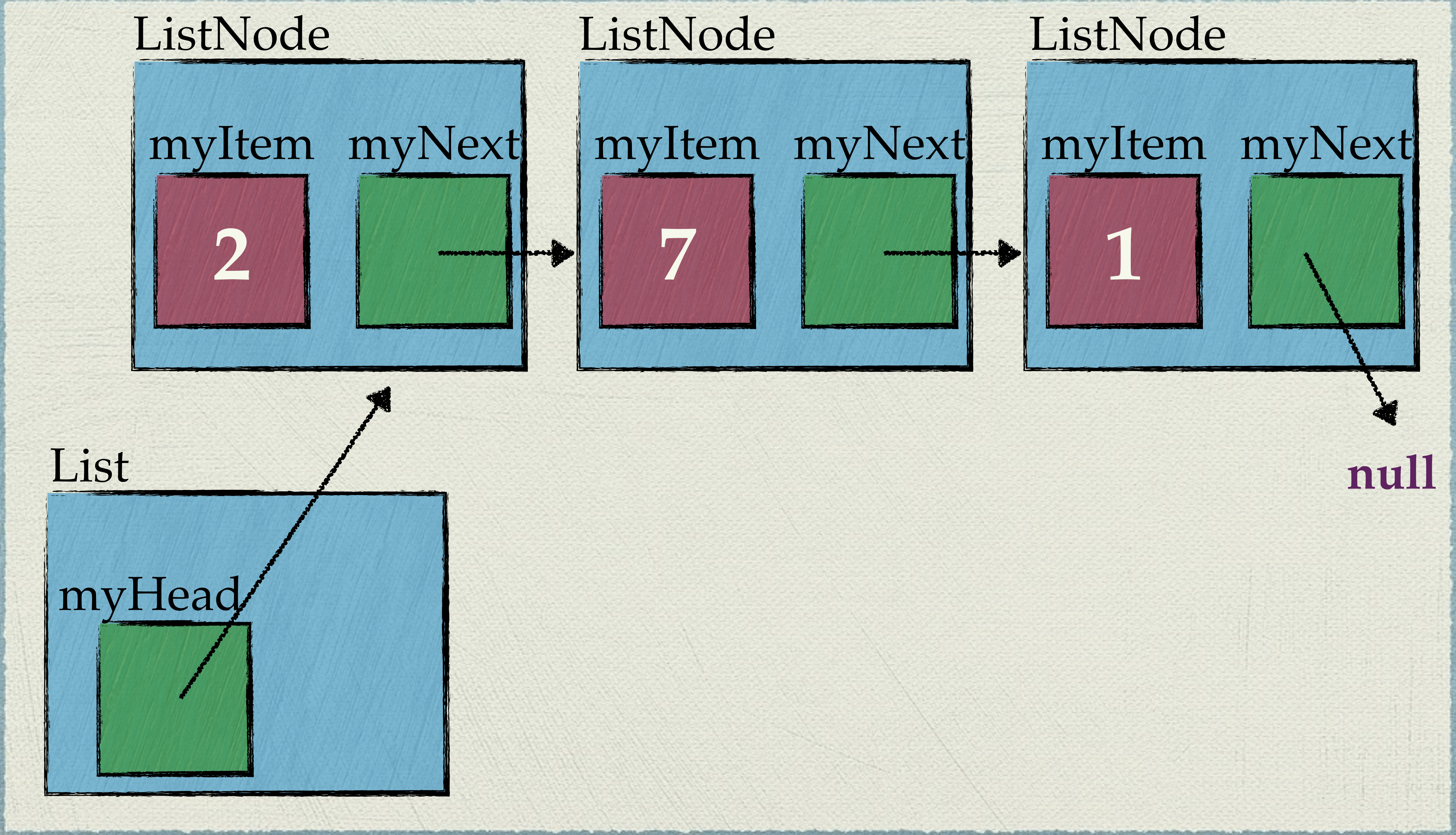
# Remember IntSequence?

- With IntSequence, the array was just a private instance variable inside the IntSequence class
- Then we could add other methods to the IntSequence class to do fancy things to that array
- Same with the linked list. The nodes will just be the instance variables of another class

# IntSequence and array



# List and ListNode class



# Inserting into a linked list

- Inserting into the front of a linked list is easy! O(1) time.
- Just reassign `myHead` to a new `ListNode`

```
public void insertFront(int item) {
  ListNode oldHead = myHead;
  myHead = new ListNode(item, oldHead);
}
```

# List and ListNode class

- The List class only contains a reference to the first node in the list.
- This is sufficient to iterate through all nodes, since all nodes can be found from the first one

# Iterating through a linked list

- In 61A, you may have processed these using recursion
- In 61BL, we introduce an iterative style

# Iterating through a linked list

- This method is in the List class, not the ListNode class
- It starts at the first node in the list, then prints out the items one-by-one

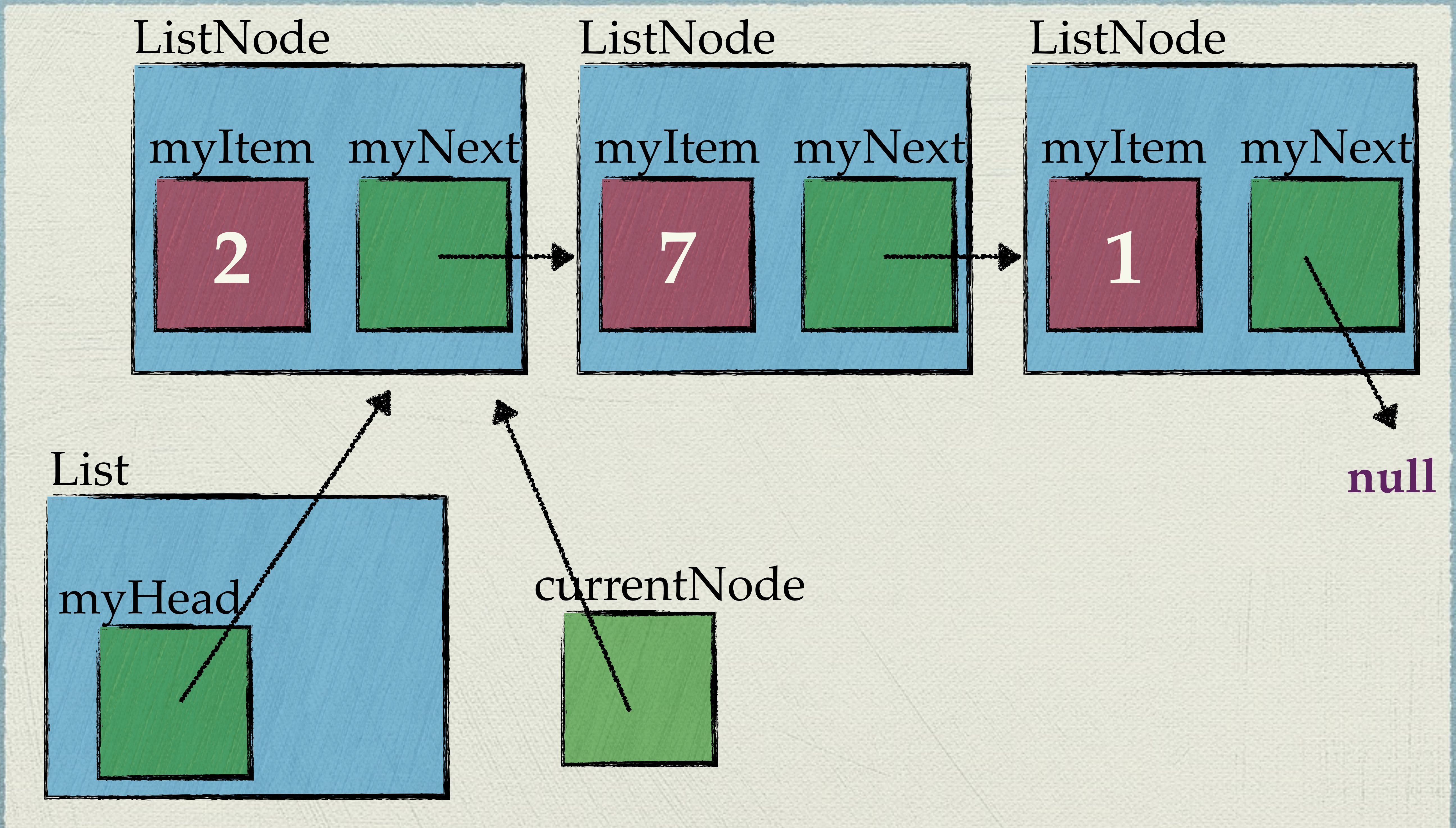
```
public void printAll() {
  ListNode currentNode = myHead;
  while (currentNode != null) {
    System.out.println(currentNode.myItem);
    currentNode = currentNode.myNext;
  }
}
```

# Iterating through a linked list

- This method is in the List class, not the ListNode class
- It starts at the first node in the list, then prints out the items one-by-one

```
public void printAll() {
  ListNode currentNode = myHead;
  while (currentNode != null) {
    System.out.println(currentNode.myItem);
    currentNode = currentNode.myNext;
  }
}
```

# List and ListNode class

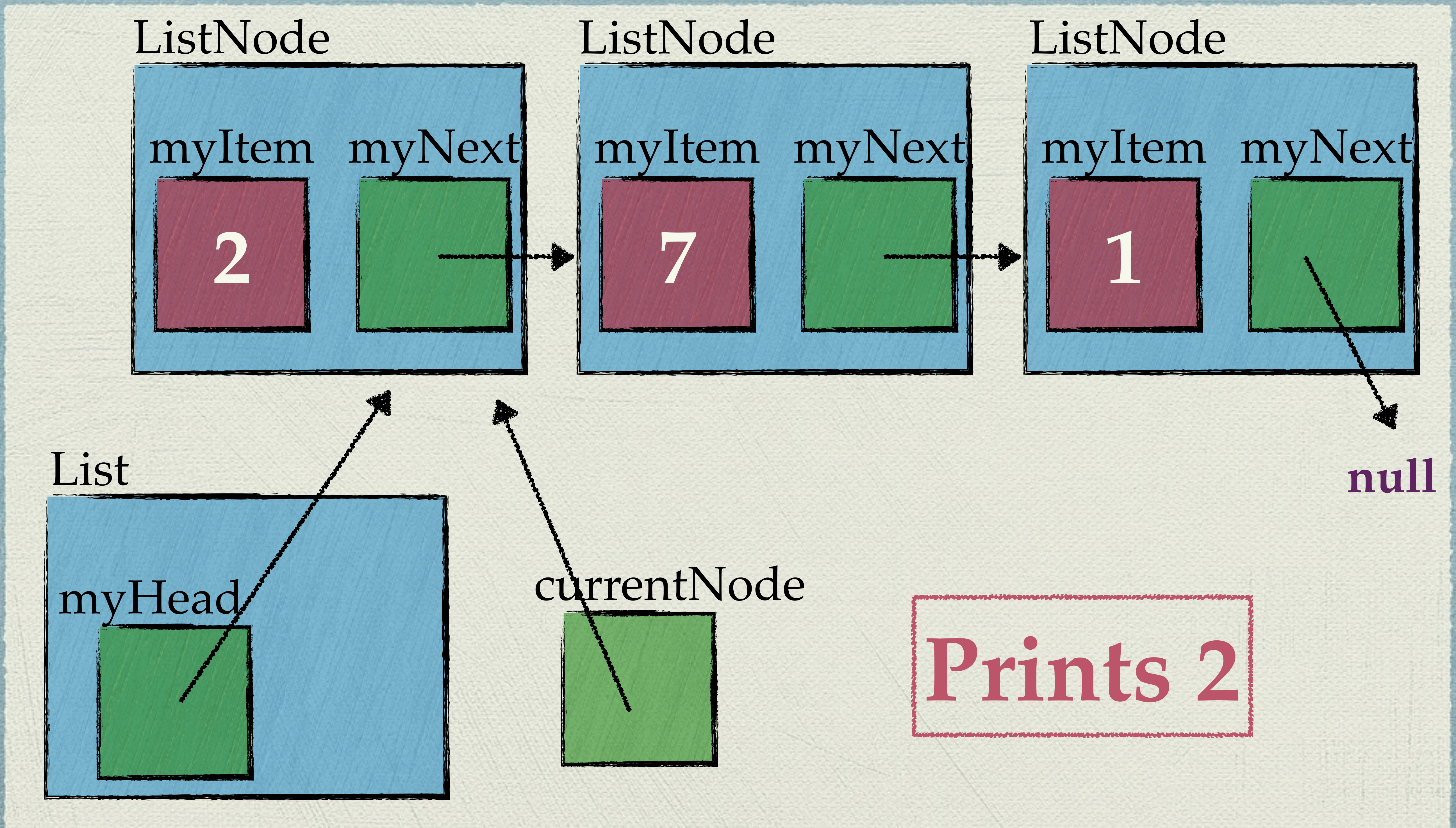


# Iterating through a linked list

- This method is in the List class, not the ListNode class
- It starts at the first node in the list, then prints out the items one-by-one

```
public void printAll() {
  ListNode currentNode = myHead;
  while (currentNode != null) {
    System.out.println(currentNode.myItem);
    currentNode = currentNode.myNext;
  }
}
```

# List and ListNode class

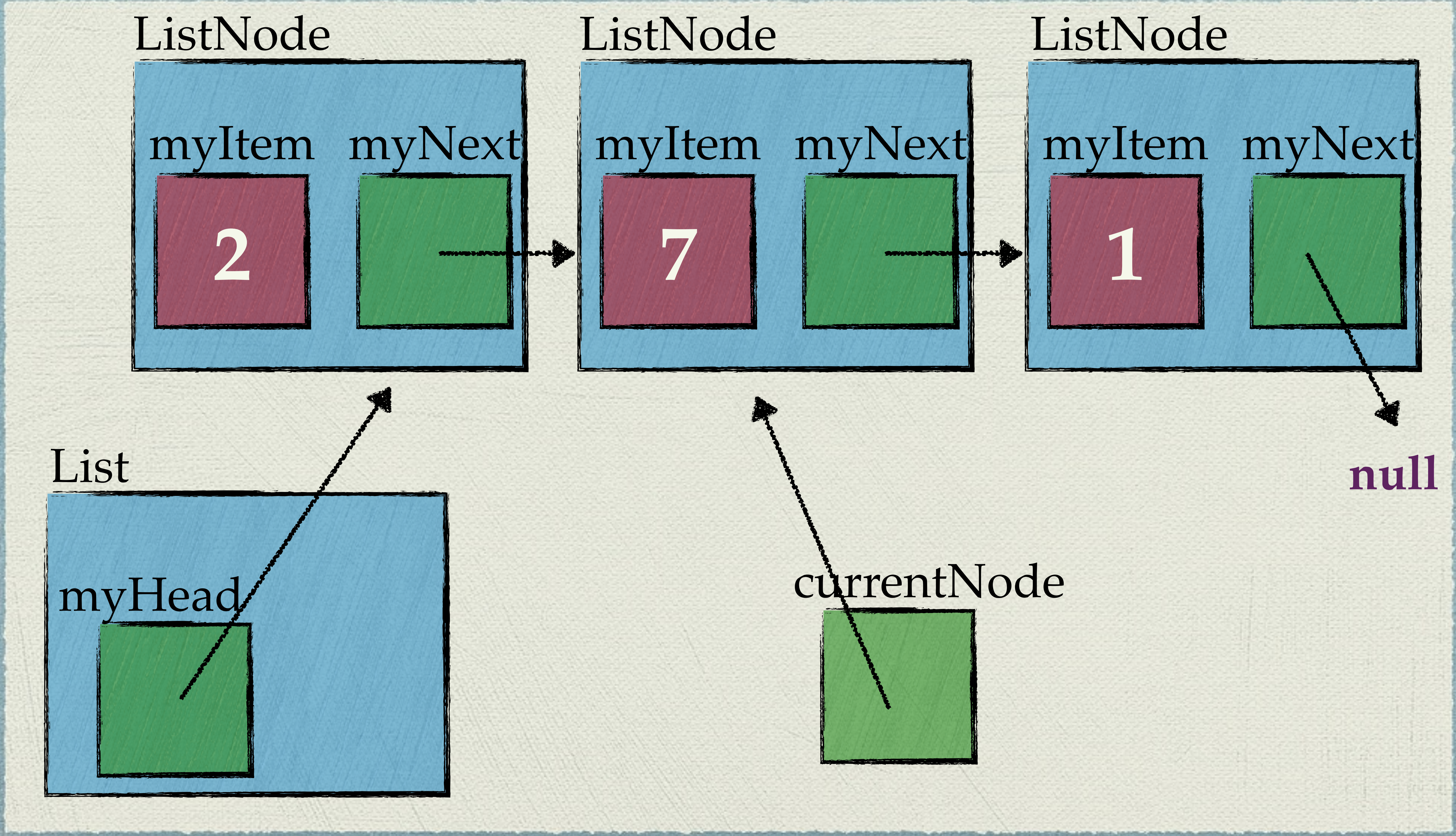


# Iterating through a linked list

- This method is in the List class, not the ListNode class
- It starts at the first node in the list, then prints out the items one-by-one

```
public void printAll() {
  ListNode currentNode = myHead;
  while (currentNode != null) {
    System.out.println(currentNode.myItem);
    currentNode = currentNode.myNext;
  }
}
```

# List and ListNode class

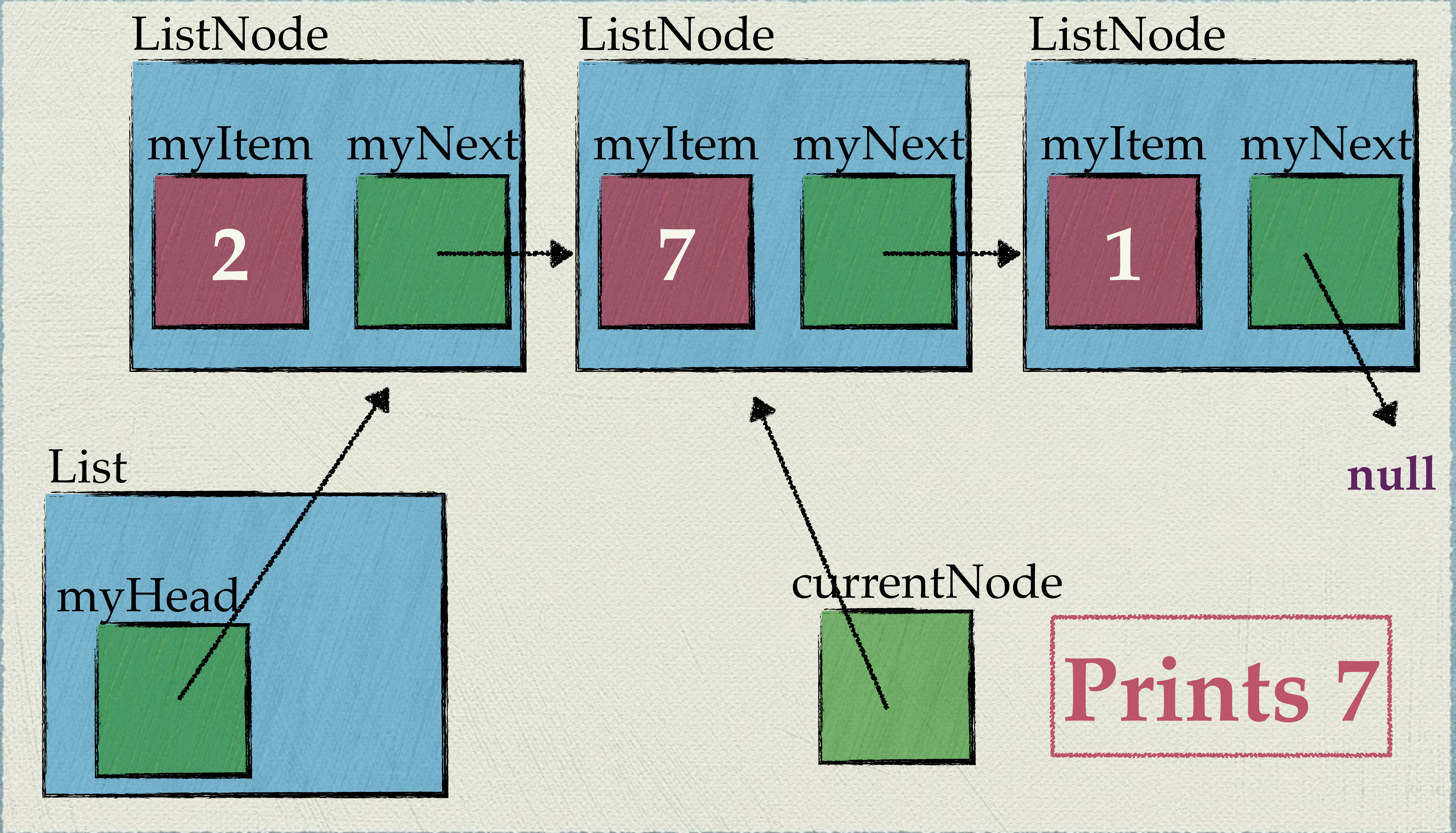


# Iterating through a linked list

- This method is in the List class, not the ListNode class
- It starts at the first node in the list, then prints out the items one-by-one

```
public void printAll() {
  ListNode currentNode = myHead;
  while (currentNode != null) {
    System.out.println(currentNode.myItem);
    currentNode = currentNode.myNext;
  }
}
```

# List and ListNode class



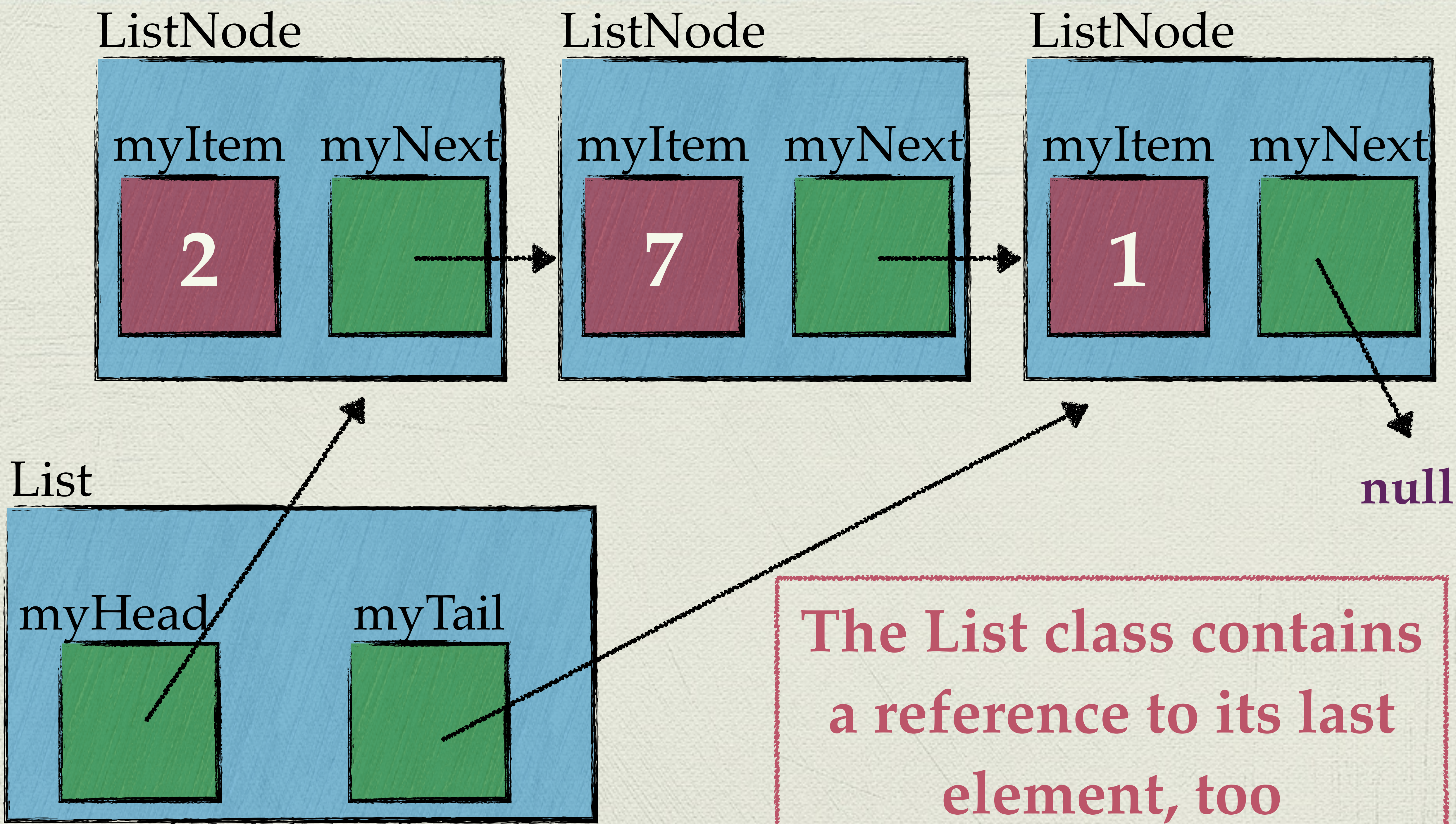
# Iterating through a linked list

- What is this…?!
- You may have seen this before
- We introduced it during the first quiz!

# Speaking of quizzes

- The ultimate showdown!! Linked list vs. arrays!!
- Who is the better data structure for representing a sequence?
- For each of the following, write the runtime of completing the operation for both linked lists and arrays
    - Use Big O notation! If you need to distinguish best-case and worst-case times, please do so
- Fyi, indexing into an array takes O(1) time, regardless where in the array

# One more thing




**The List class contains a reference to its last element, too**

# Linked lists vs. arrays!

A. Append an item to the *end* of the sequence

B. Append a sequence of length $n$ to the end of another sequence of length $m$

C. Return the $k$th item of a sequence of length $n$

D. Append $k$ items to the end of a sequence, in a row. Assume the sequence starts with 0 items

E. Remove the $k$th item from a sequence with $n$ elements

# Solutions

A. Append an item to the *end* of the sequence of length $n$
   **linked: O(1), array: O(1) best, O(n) worst**

B. Append a sequence of length $n$ to the end of another sequence of length $m$

C. Return the $k$th item of a sequence of length $n$

D. Append $k$ items to the end of a sequence, in a row. Assume the sequence starts with 0 items

E. Remove the $k$th item from a sequence with $n$ elements

# Solutions

A. Append an item to the *end* of the sequence of length *n* **linked: O(1), array: O(1) best, O(n) worst**

B. Append a sequence of length *n* to the end of another sequence of length *m* **linked: O(1), array: best O(n), worst O(m + n)**

C. Return the *k*th item of a sequence of length *n*

D. Append *k* items to the end of a sequence, in a row. Assume the sequence starts with 0 items

E. Remove the *k*th item from a sequence with *n* elements

# Solutions

A. Append an item to the *end* of the sequence of length *n* **linked: O(1), array: O(1) best, O(n) worst**

B. Append a sequence of length *n* to the end of another sequence of length *m* **linked: O(1), array: best O(n), worst O(m + n)**

C. Return the *k*th item of a sequence of length *n* **linked: O(k), array: O(1)**

D. Append *k* items to the end of a sequence, in a row. Assume the sequence starts with 0 items

E. Remove the *k*th item from a sequence with *n* elements

# Solutions

A. Append an item to the *end* of the sequence of length *n* **linked: O(1), array: O(1) best, O(n) worst**

B. Append a sequence of length *n* to the end of another sequence of length *m* **linked: O(1), array: best O(n), worst O(m + n)**

C. Return the *k*th item of a sequence of length *n* **linked: O(k), array: O(1)**

D. Append *k* items to the end of a sequence, in a row. Assume the sequence starts with 0 items **linked: O(k), array: O(k) — really??**

E. Remove the *k*th item from a sequence with *n* elements

# Solutions

A. Append an item to the *end* of the sequence of length *n* **linked: O(1), array: O(1) best, O(n) worst**

B. Append a sequence of length *n* to the end of another sequence of length *m* **linked: O(1), array: best O(n), worst O(m + n)**

C. Return the *k*th item of a sequence of length *n* **linked: O(k), array: O(1)**

D. Append *k* items to the end of a sequence, in a row. Assume the sequence starts with 0 items **linked: O(k), array: O(k) — really??**

E. Remove the *k*th item from a sequence with *n* elements **linked: O(k), array: O(n - k)**

# The score board

A. Append an item to the *end* of the sequence **Depends**

B. Append a sequence of length $n$ to the end of another sequence of length $m$ **Linked list**

C. Return the $k$th item of a sequence of length $n$ **Array**

D. Append $k$ items to the end of a sequence, in a row. Assume the sequence starts with 0 items **Tie**

E. Remove the $k$th item from a sequence with $n$ elements **Depends**

# Sooo… linked lists…

- You're not really impressing me right now
- Linked lists are useful in specific cases
  - We'll see examples later
- But your default choice should probably be an array (ArrayList)