Middlebury

CSCI 201: Data Structures

Spring 2025

Lecture 6M: Linked Lists

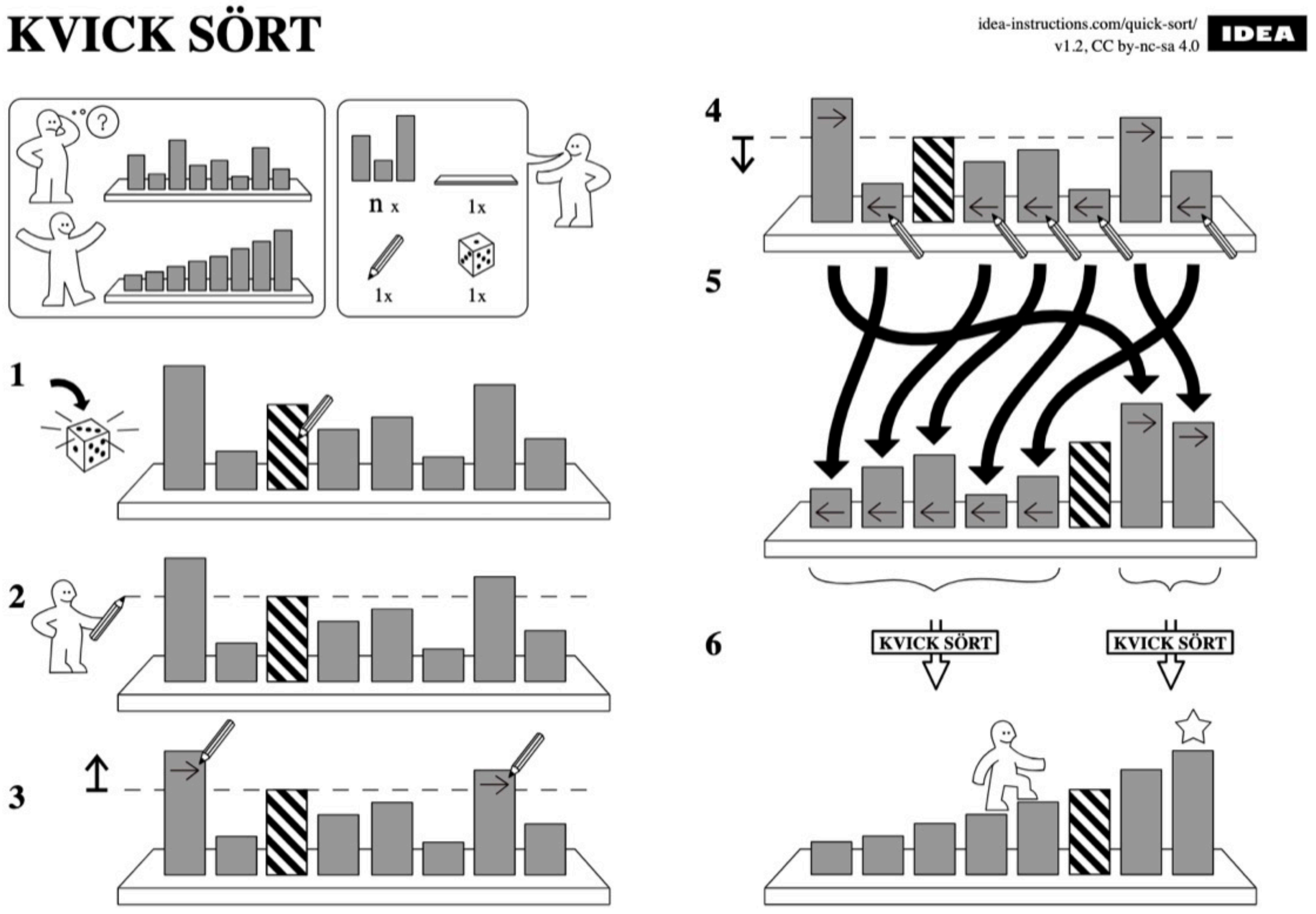
Goals for today:

- Analyze the runtime complexity of `QuickSort`.
- Motivate the use of the singly-`LinkedList` data structure.
- **Insert** a node into a `LinkedList` right after a given node.
- **Remove** a node from a `LinkedList` right after a given node.
- Reinforce the concept of **references**, which can be used to link nodes.

and a few others

Sorting algorithm #6: QuickSort (from last class).

1. Pick a pivot.
2. Partition items so than any item < pivot is in left subarray and any item > pivot is in the right subarray.
3. Call QuickSort on each subarray.



Sorting algorithm #6: QuickSort (from last class).

1. Pick a pivot. last item in array
2. Partition items so than any item < pivot is in left subarray and any item > pivot is in the right subarray.
3. Call `QuickSort` on each subarray.

```java
public static void sort(int[] items) {
  quicksortHelper(items, 0, items.length - 1);
}

private static void quicksortHelper(int[] items,
                                     int left, int right) {
  if (left >= right) return;

  // pick a pivot and partition items to the left/right
  int p = partition(items, left, right);

  // call quicksort on left and right subarrays
  quicksortHelper(items, left, p - 1);
  quicksortHelper(items, p + 1, right);
}
```

8	5	1	3	6	2	7	4

worst case: array already sorted

quicksort

empty

comparisons

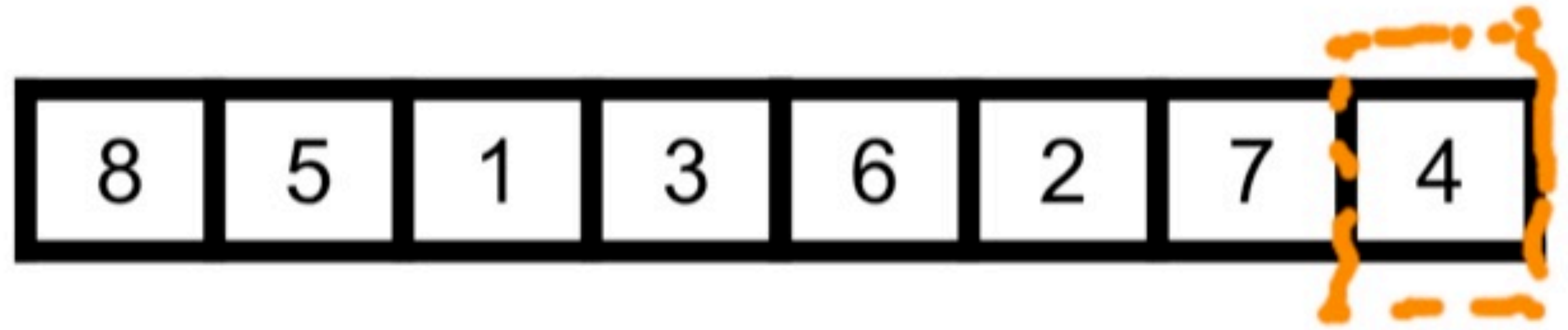
$(n-1) + (n-2) + \dots + 1$

$= \frac{n(n-1)}{2} = \boxed{O(n^2)}$ worst case

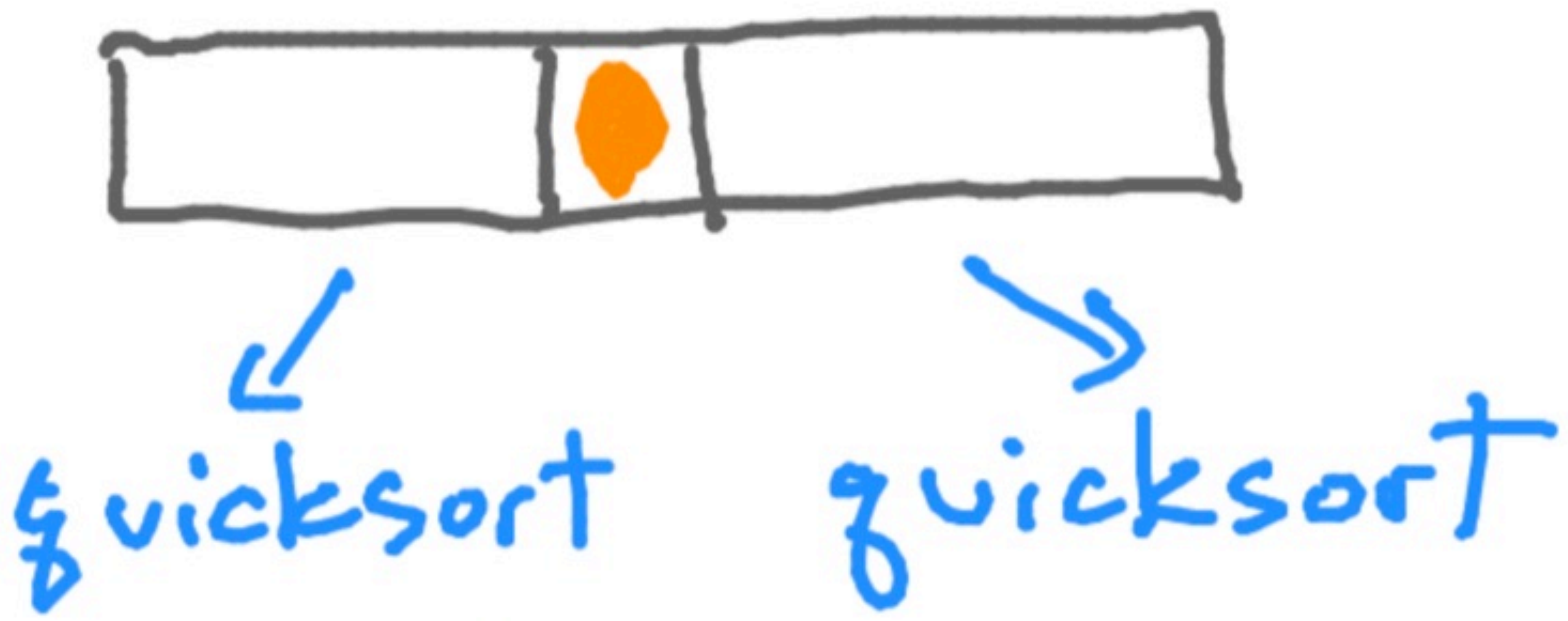
Sorting algorithm #6: QuickSort (from last class).

1. Pick a pivot.
2. Partition items so than any item < pivot is in left subarray and any item > pivot is in the right subarray.
3. Call QuickSort on each subarray.

```java
public static void sort(int[] items) {
  quicksortHelper(items, 0, items.length - 1);
}

private static void quicksortHelper(int[] items,
                                    int left, int right) {
  if (left >= right) return;

  // pick a pivot and partition items to the left/right
  int p = partition(items, left, right);

  // call quicksort on left and right subarrays
  quicksortHelper(items, left, p - 1);
  quicksortHelper(items, p + 1, right);
}
```



Best/average case: pivot splits list evenly in 2 parts



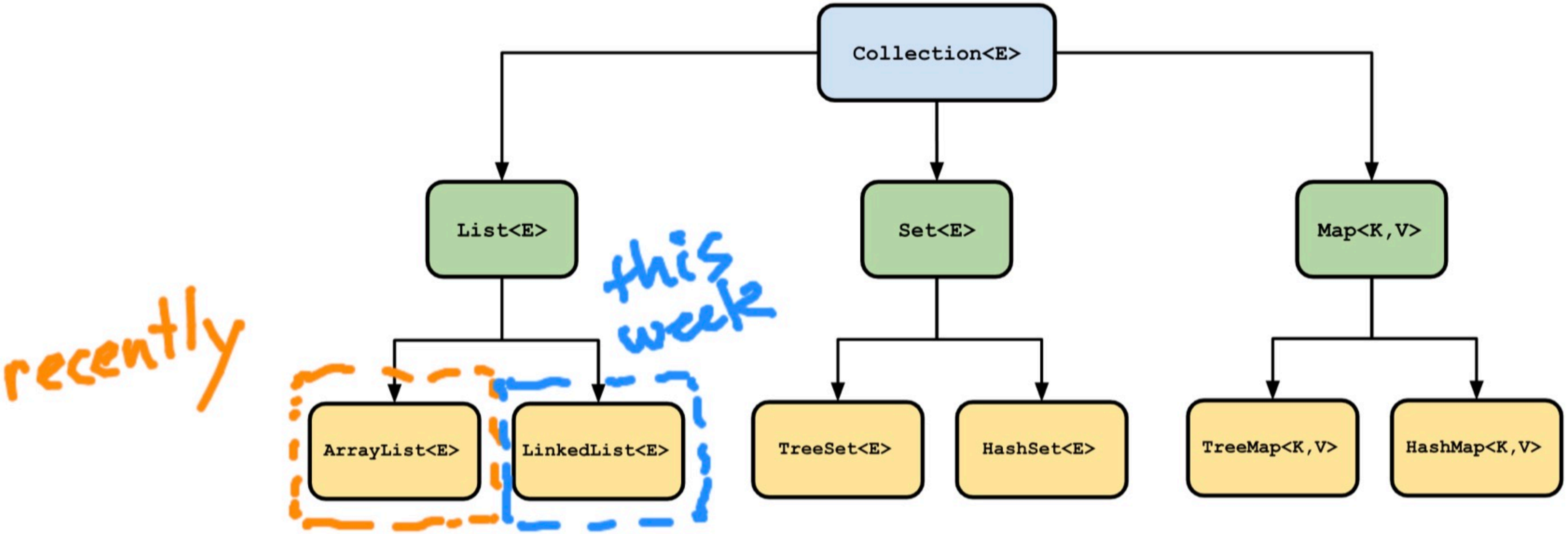
≈ n comparisons per level

total work: $\boxed{O(n \log n)}$

best/average case

levels: log n

Now, let's talk about the `List` *interface* again!



`ArrayList` was good for adding to the end, but what if we want to add to the **front**?

Let's experiment and see what happens... (open up `ListAddFrontTest.java`).

```
n = 10, time = 6.5137E-6 sec.
n = 100, time = 1.36825E-6 sec.
n = 1000, time = 1.4225509999999999E-6 sec.
n = 10000, time = 1.0787066E-6 sec.
n = 100000, time = 8.00874796E-6 sec.
```

1 million is very slow!

Is there a way to achieve *constant* time addition to the front?

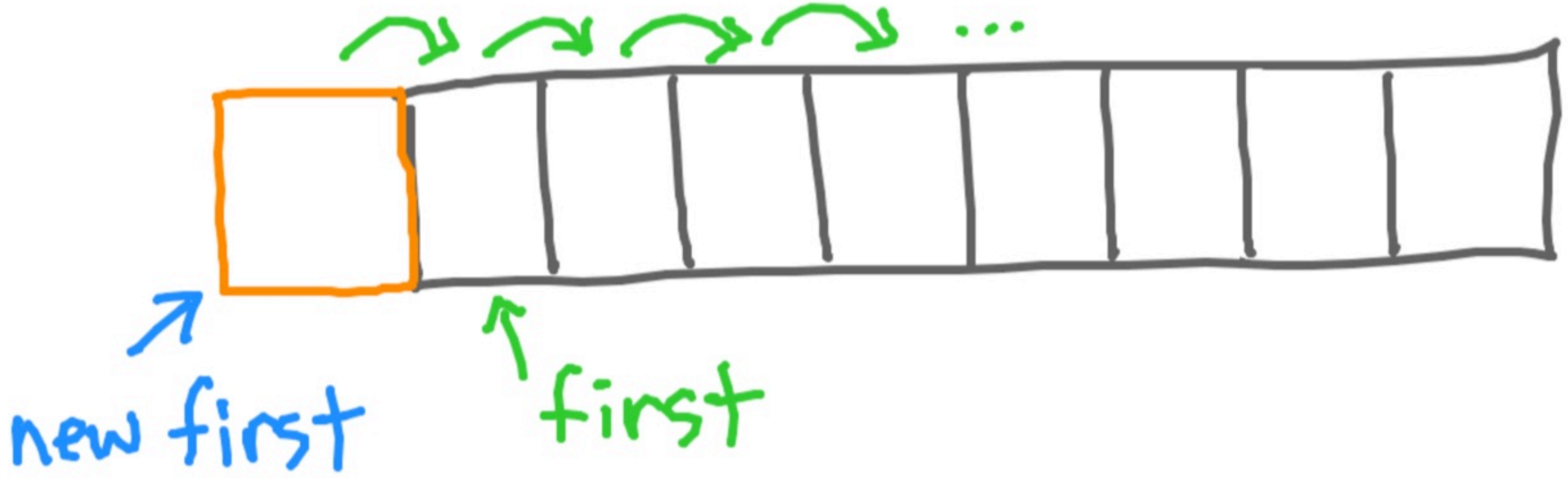
Recall with ArrayLists:

addLast() has O(1) amortized cost

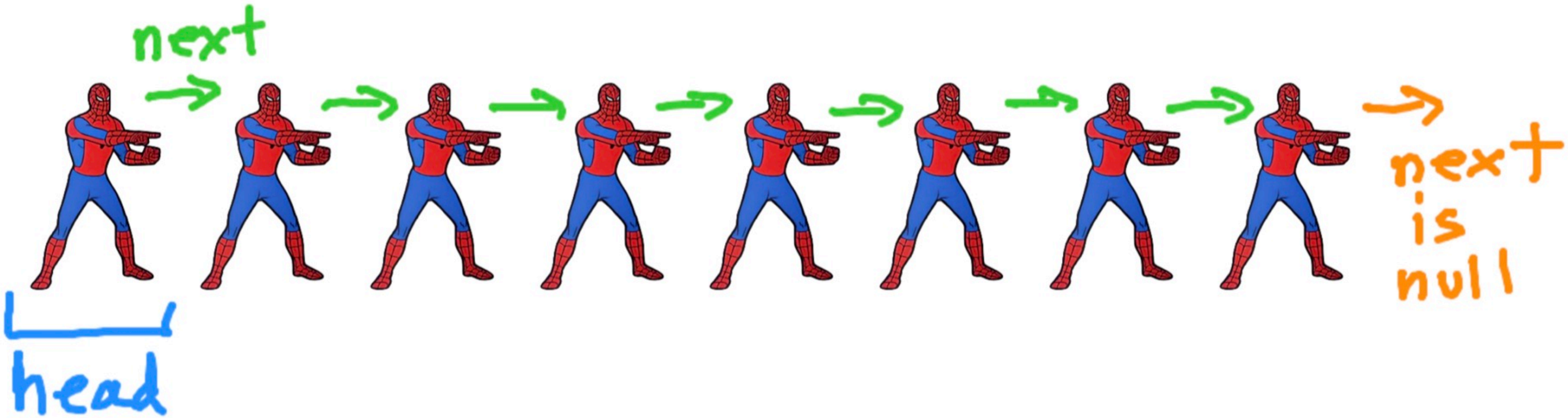
addFirst() has O(n) cost

because all elements need to shift

Idea: what if we kept track of which elements of the list are next to each other?



This is the main idea of linked lists



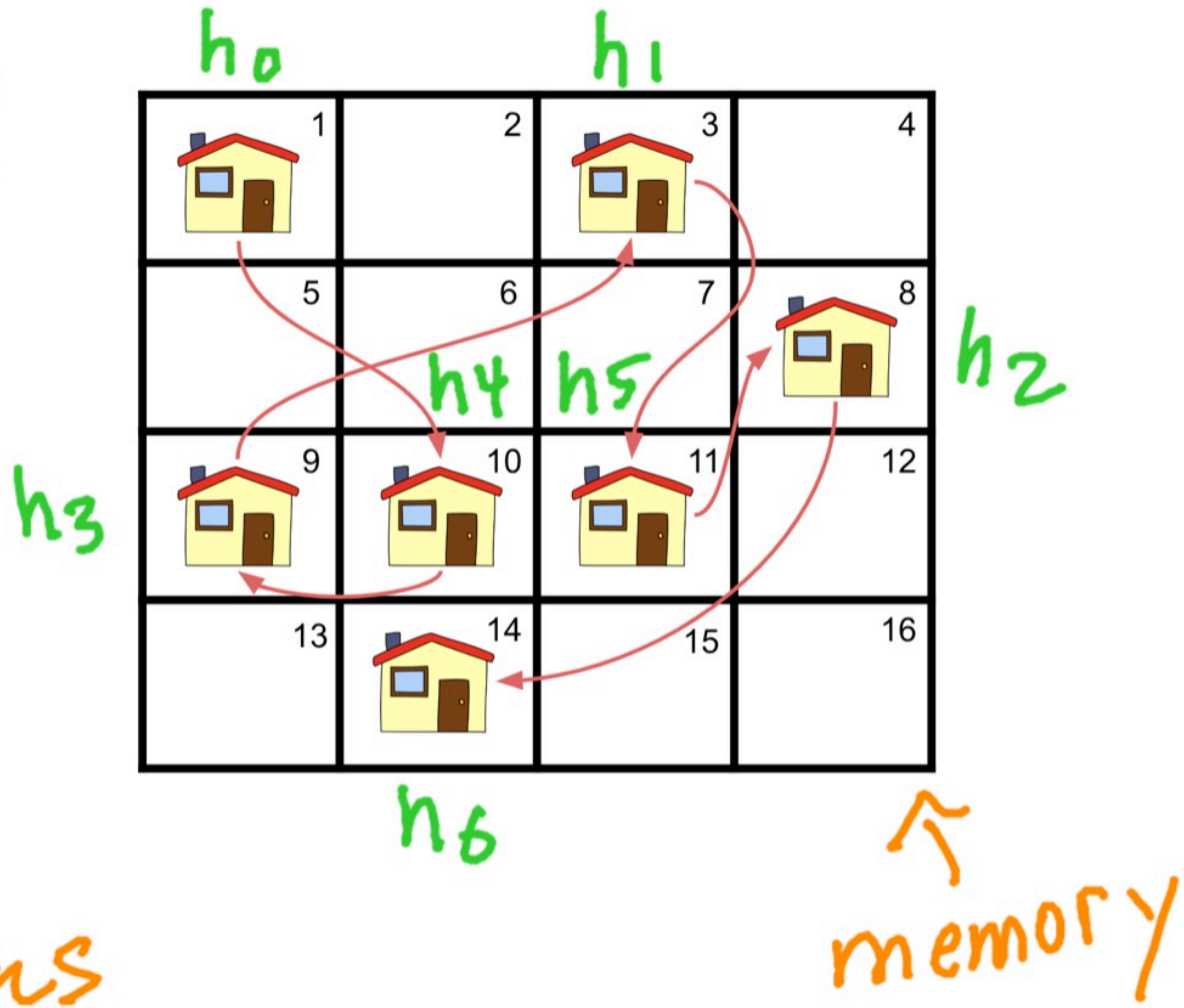
How do we achieve this idea of "pointing" to other nodes?

[h_0 h_1 h_2 h_3 h_4 h_5 h_6]

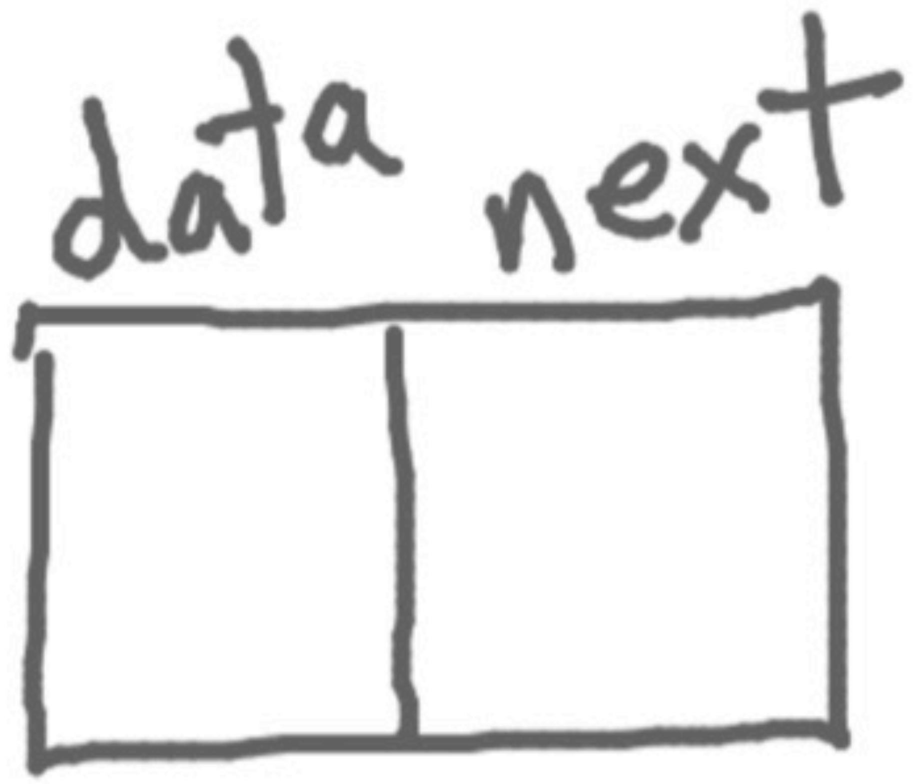
h_4 h_5 h_6 h_1 h_3 h_2 null

$\underbrace{10 \quad 11 \quad 14 \quad 3 \quad 9 \quad 8}$

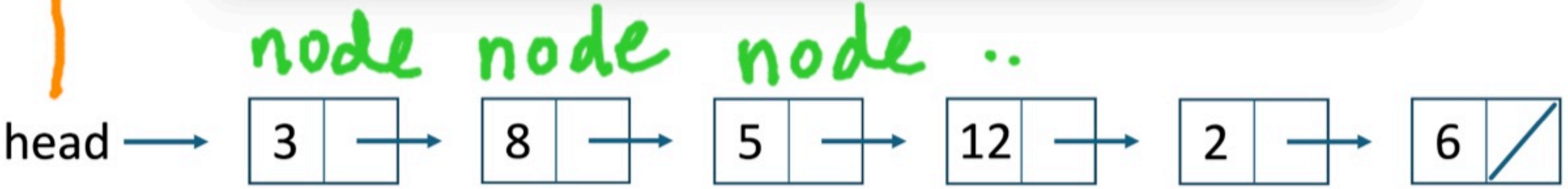
references
to
memory locations



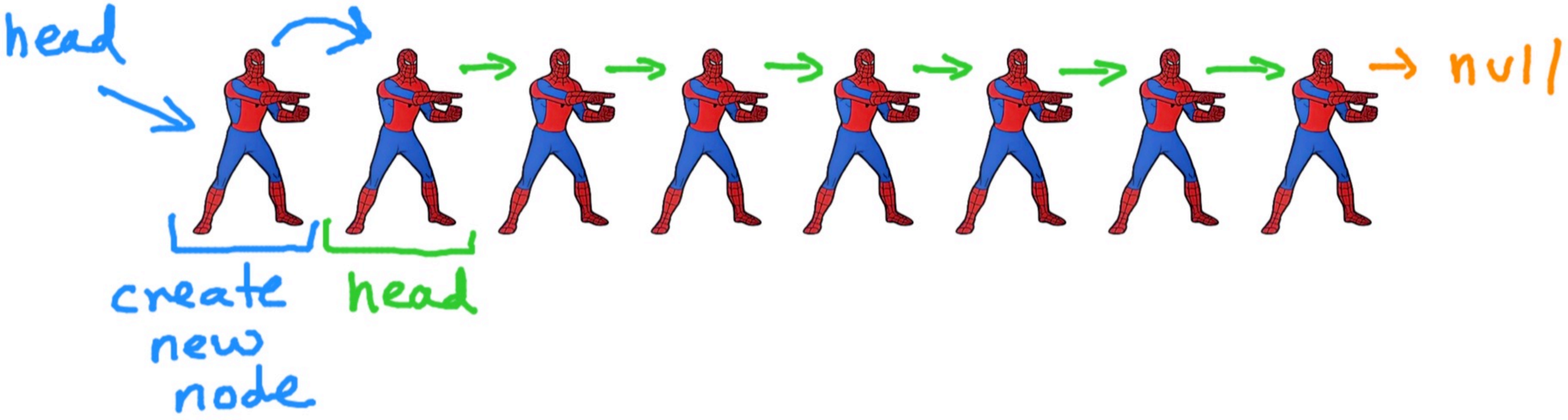
Designing our own LinkedList (for integers)!




```
class ListNodeInt {
  public ListNodeInt next;
  public int data;

  public ListNodeInt(int data) {
    this.data = data;
    next = null;
  }
}

public class LinkedListInt {
  private ListNodeInt head;

  public LinkedListInt() {
    head = null;
  }
}
```



Inserting a node at the beginning of the list.




```java
public void addFront(int data) {
  ListNodeInt node = new ListNodeInt(data);
  node.next = head;
  head = node;
}
```

Okay, let's see how fast this is!

```java
public void add(int index, int data) {
  if (index == 0) {
    addFront(data);
  } else {
    // we'll be able to do this soon...
  }
}
```

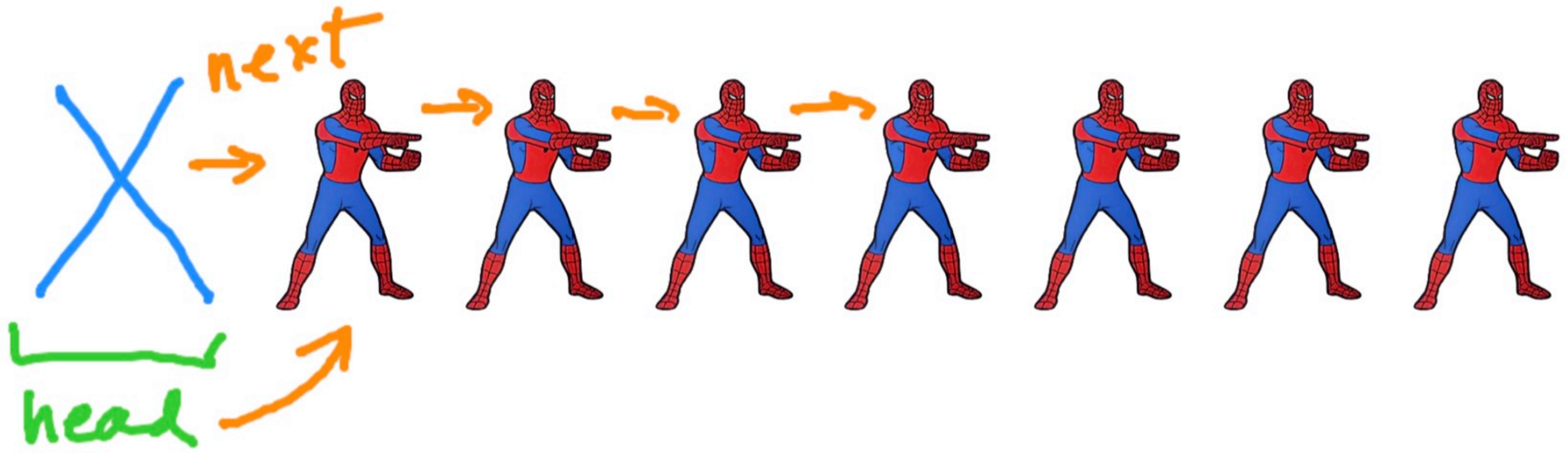
Then go back to `ListAddFrontTest.java` and change the following line:

```java
//ArrayList<Integer> list = new ArrayList<>();
LinkedListInt list = new LinkedListInt();
```

```
n = 10, time = 4.204513E-4 sec.
n = 100, time = 1.16407E-6 sec.
n = 1000, time = 6.24369E-7 sec.
n = 10000, time = 1.239002E-7 sec.
n = 100000, time = 7.398624E-8 sec.
n = 1000000, time = 9.1930419E-8 sec.
```

1 million is very fast!

Removing a node from the beginning of the list.



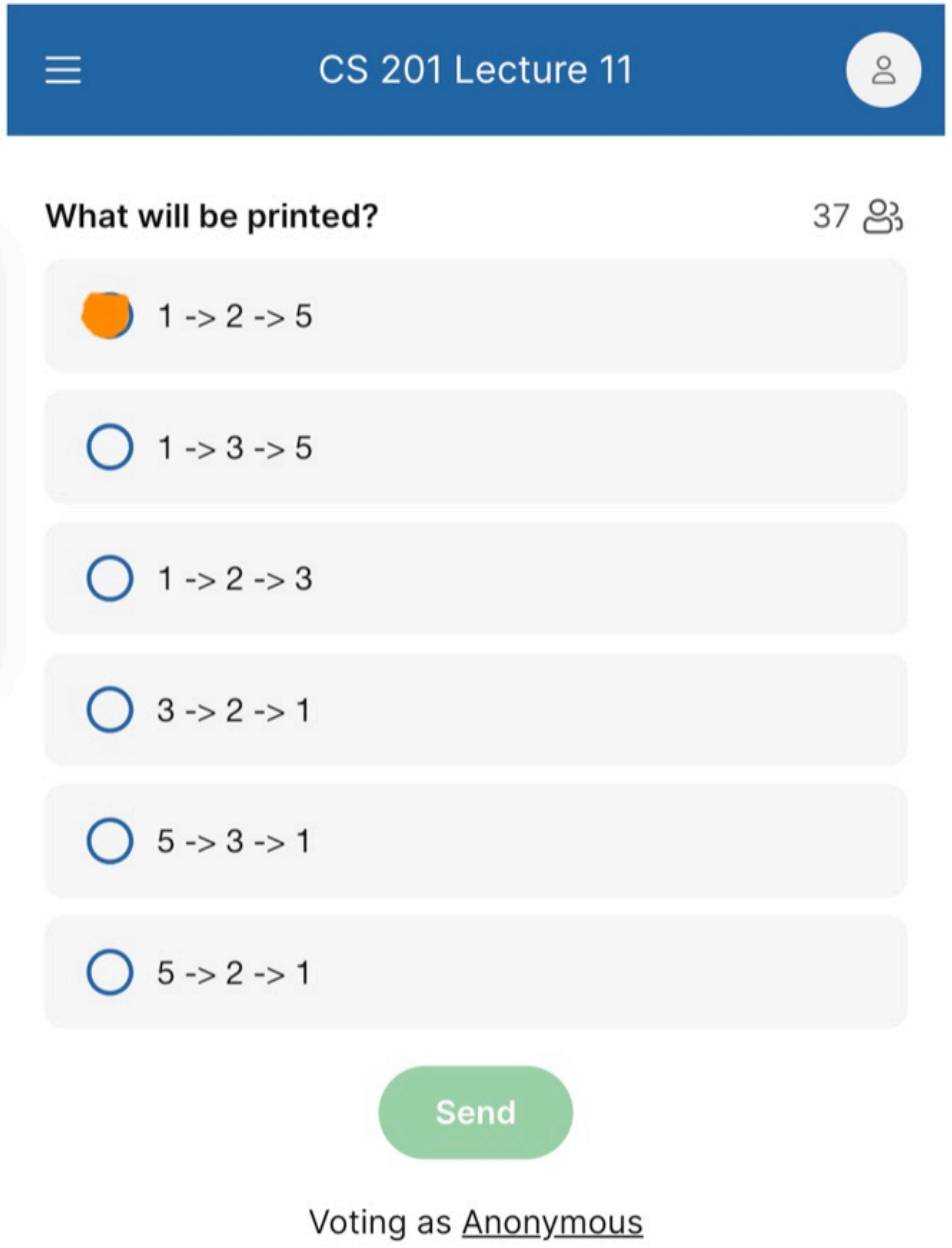
Let's write a method to print out the linked list.

```java
public String toString() {
  String result = "";
  ListNodeInt node = head;
  while (node != null) {
    result += node.data;
    if (node.next != null) result += " -> ";
    node = node.next;
  }
  return result;
}

// then in a PSVM
System.out.println(list);
```

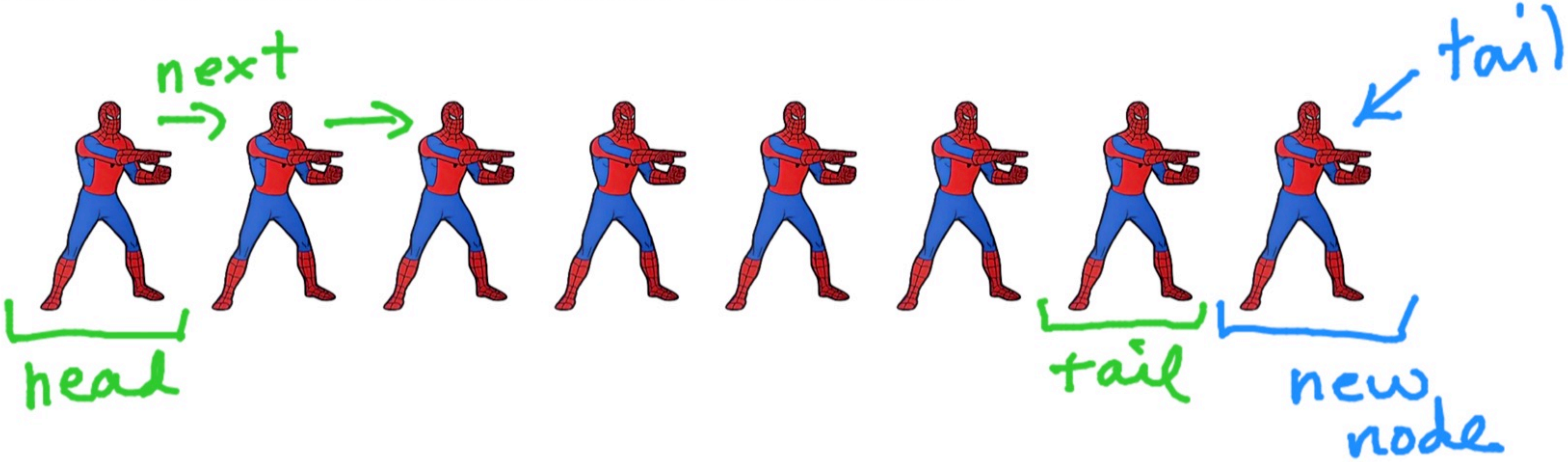
What will be printed here? See slido.com # 3174317

```
LinkedListInt list = new LinkedListInt();
list.addFront(5);
list.addFront(3);
list.removeFront();
list.addFront(2);
list.addFront(1);
System.out.println(list);
```

1 → 2 ~~3~~ 5



What if we want to add a node to the end?




```java
public class LinkedListInt {
  ListNodeInt head;
  ListNodeInt tail;

  void addLast(int data) {
    ListNodeInt node = new ListNodeInt(data);
    if (tail != null) tail.next = node;
    tail = node;
  }
}
```

And finally, what if we wanted to design our own text editor?

```
1 Hello CS 201!
```

Lab 5!

We will use <u>doubly-linked lists</u> (next class) that have <u>next</u> and <u>prev</u> pointers

See you on Wednesday!

- We'll extend these ideas to implement a **doubly-linked list**.
- We'll also **generic**-ify our linked list implementation.
- Midterm Study Guide is posted.
- Reminder that Noah (go/noah) and Smith (go/smith) have office hours throughout the week and the 201 Course Assistants have drop-in hours in the late afternoons/evenings (go/cshelp).