Linked Lists

CS 2110

Software Development Methods

Spring 2020

Prof. Nada Basit

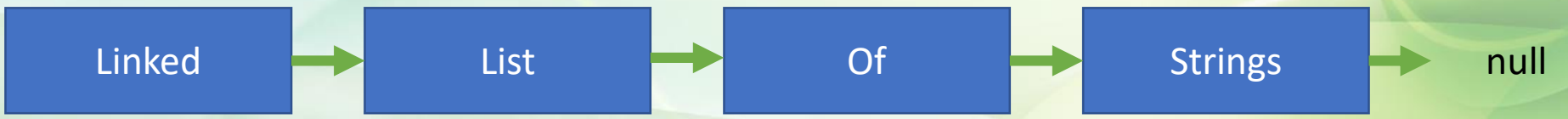
Motivation

- We've discussed how **ArrayLists** implement the List interface

- **LinkedLists** are another implementation of the List interface
  - Implemented using two classes: **Node** class and **List** class

Linked List Implementations (Stack vs Queue)

- A Linked List can be implemented in a number of ways, we will see implementations of:
  - **A stack**
  - **A queue**



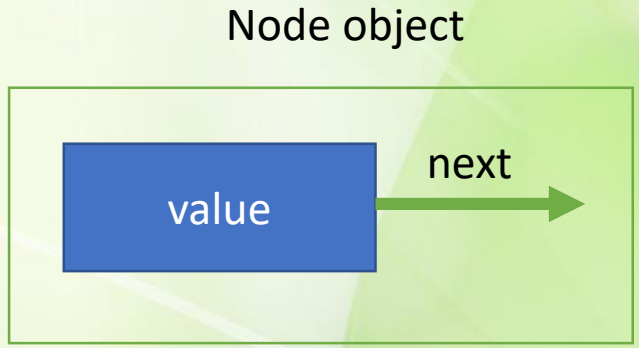
Review files…

LinkedIntStack.java

IntNode.java

Node Class

- Look at the **Node Class** below
  - `value` stores one piece of information
  - `next` says where to find the next piece of information
    - `null` if there is no more information

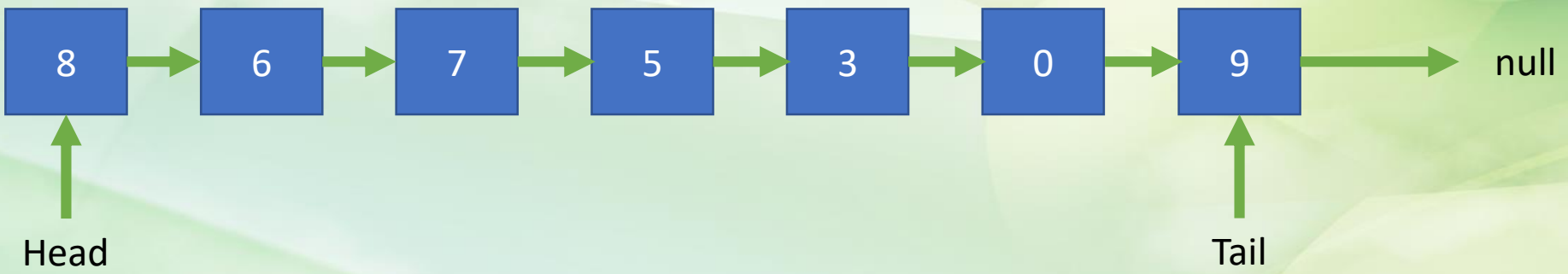



```
public class IntNode {

    public int value; //the value stored at the node
    public IntNode next; //where to find more data

    public IntNode(int value) {
        this.value = value;
        this.next = null; //this means there's nothing after that value
    }

}
```

***LinkedList** is a recursive data structure; note the data type of "next" (same as class)*

List Class

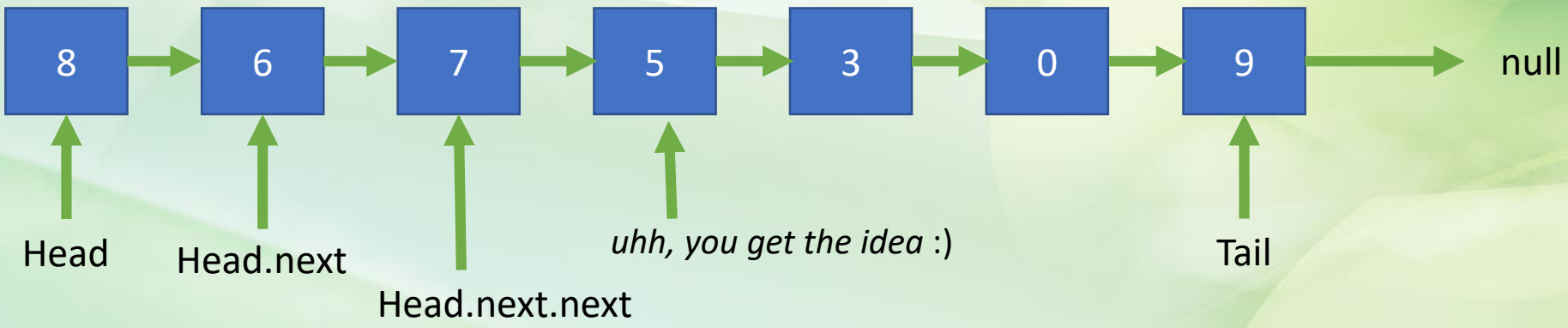
- **List** combines a **series of Nodes** to form a **list**
  - We must keep track at least of the **“first” node** (often called “`head`”)
  - Think of it like a train, the engine is the head, and we can reach every other car by moving backwards from the engine
- It may be useful to keep track of the **end** (“`tail`”) as well
  - *Not required*



List Class

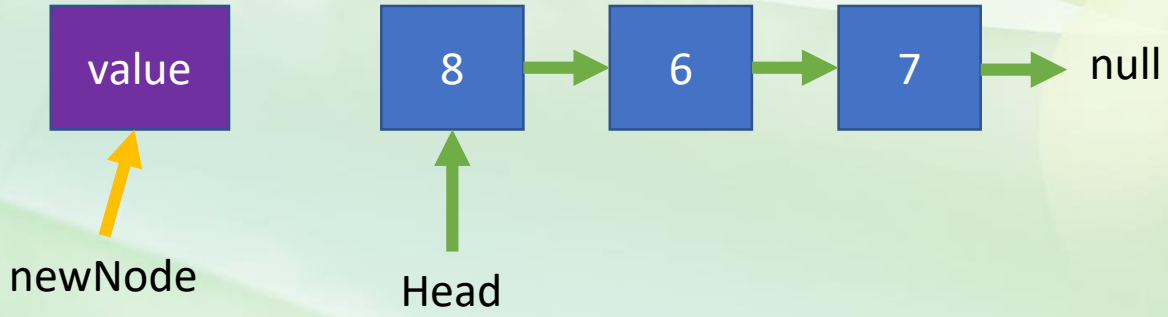
- The **List class** doesn't directly keep track of every node
  - We access every node indirectly through the `head`

- For example, `head.next.next.next.value = 5`
  - Always remember: a node isn't a value, it's a **value** AND a **next**
    - So you need to use the **dot operator** to access the value or next separately



LinkedIntStack – Push function

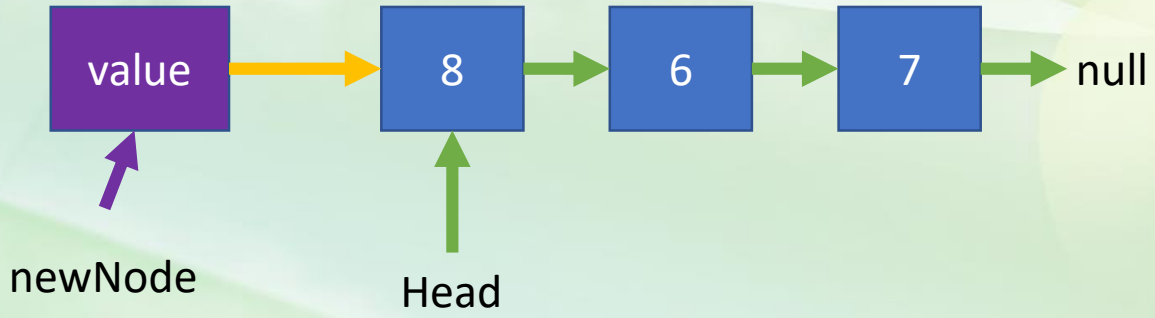
- To implement the `push` function, do the following
  1. **Create a new node containing the value we are pushing**
  2. Add the node before the head node
  3. Update the head pointer




```java
/**
 * Push - add the value at the head of the list
 */
public void push(int value) {
    //create a new Node object
    IntNode newNode = new IntNode(value);

    //put the node in front of head
    newNode.setNext(head);

    //update the head pointer to the new first node
    head = newNode;
}
```

LinkedIntStack – Push function

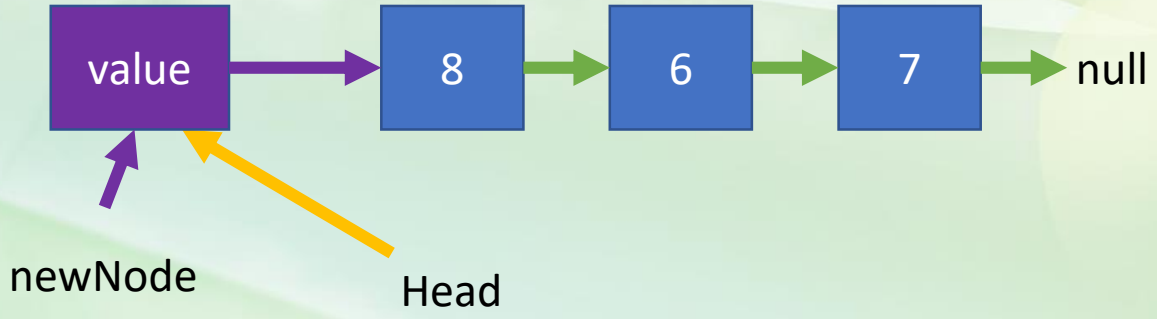
- To implement the `push` function, do the following
  1. Create a new node containing the value we are pushing
  2. **Add the node before the head node**
  3. Update the head pointer




```java
/**
 * Push - add the value at the head of the list
 */
public void push(int value) {
    //create a new Node object
    IntNode newNode = new IntNode(value);

    //put the node in front of head
    newNode.setNext(head);

    //update the head pointer to the new first node
    head = newNode;
}
```

LinkedIntStack – Push function

- To implement the `push` function, do the following
  1. Create a new node containing the value we are pushing
  2. Add the node before the head node
  3. **Update the head pointer**




```java
/**
 * Push - add the value at the head of the list
 */
public void push(int value) {
    //create a new Node object
    IntNode newNode = new IntNode(value);

    //put the node in front of head
    newNode.setNext(head);

    //update the head pointer to the new first node
    head = newNode;
}
```

LinkedIntStack - isEmpty

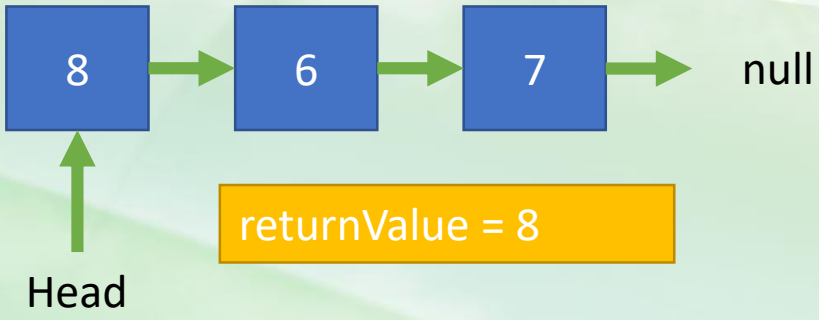
- How can we determine if a Linked List is empty?
- Which List below is empty?
  - Obviously List 2

```
/**
 * Determine if stack is empty
 */
public boolean isEmpty() {
    return head == null;
}
```

LinkedIntStack – Pop() function

- To remove a value
  1. **Store the value to be removed in a variable**
  2. Update head pointer
  3. Return the value

```
public int pop() {
    //we cannot pop if the list is empty
    if (isEmpty()) {
        throw new IllegalStateException("Error: cannot pop from empty list");
    }
    //store the value at the front of the list
    int returnValue = head.getValue();

    //remove the node at the front of the list
    head = head.getNext();

    //return the stored value
    return returnValue;
}
```

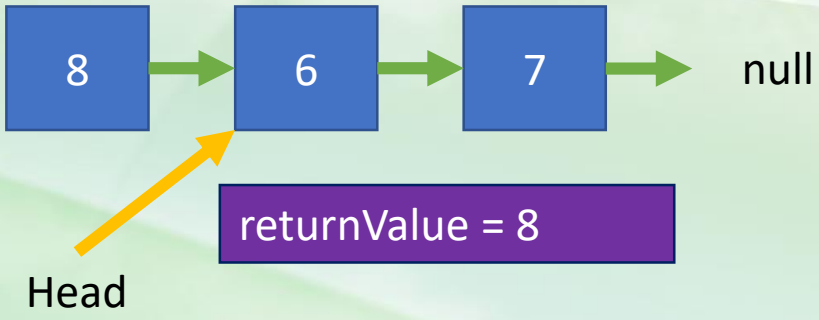


LinkedIntStack – Pop() function

- To remove a value
  1. Store the value to be removed in a variable
  2. **Update head pointer**
  3. Return the value

```
public int pop() {
    //we cannot pop if the list is empty
    if (isEmpty()) {
        throw new IllegalStateException("Error: cannot pop from empty list");
    }
    //store the value at the front of the list
    int returnValue = head.getValue();

    //remove the node at the front of the list
    head = head.getNext();

    //return the stored value
    return returnValue;
}
```

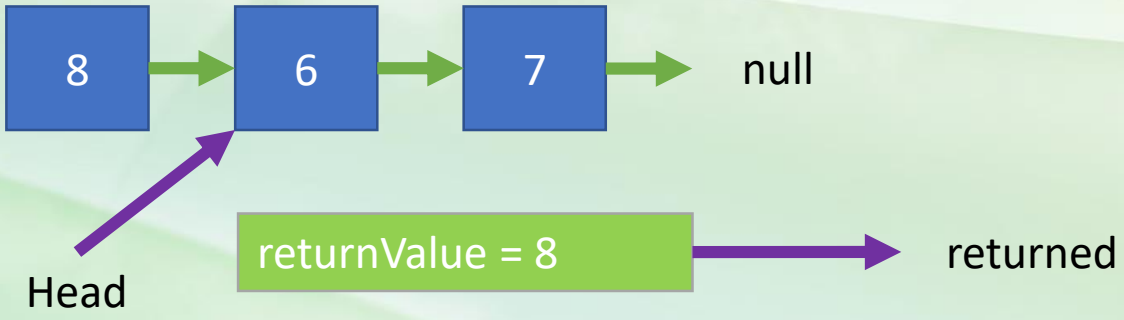
"head" now becomes what it's next pointer was pointing to (node 6 in this case)



LinkedIntStack – Pop() function

- To remove a value
  1. Store the value to be removed in a variable
  2. Update head pointer
  3. **Return the value**

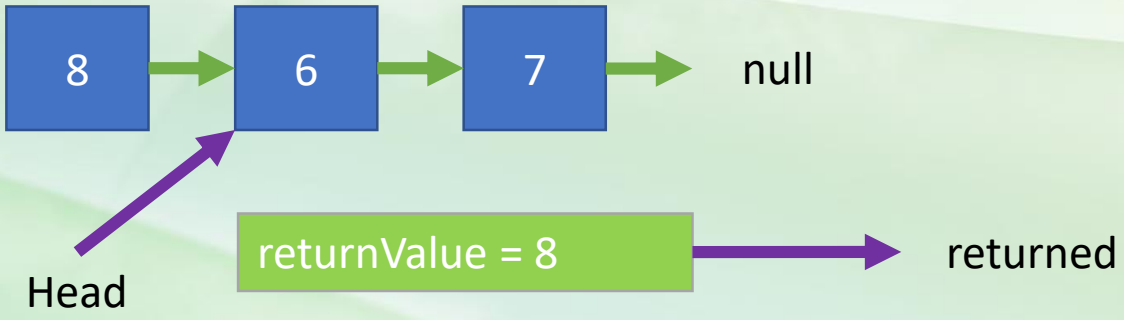
```
public int pop() {
    //we cannot pop if the list is empty
    if (isEmpty()) {
        throw new IllegalStateException("Error: cannot pop from empty list");
    }
    //store the value at the front of the list
    int returnValue = head.getValue();

    //remove the node at the front of the list
    head = head.getNext();

    //return the stored value
    return returnValue;
}
```



LinkedIntStack – Pop() function

- *Wait*, what happens to the node containing 8?

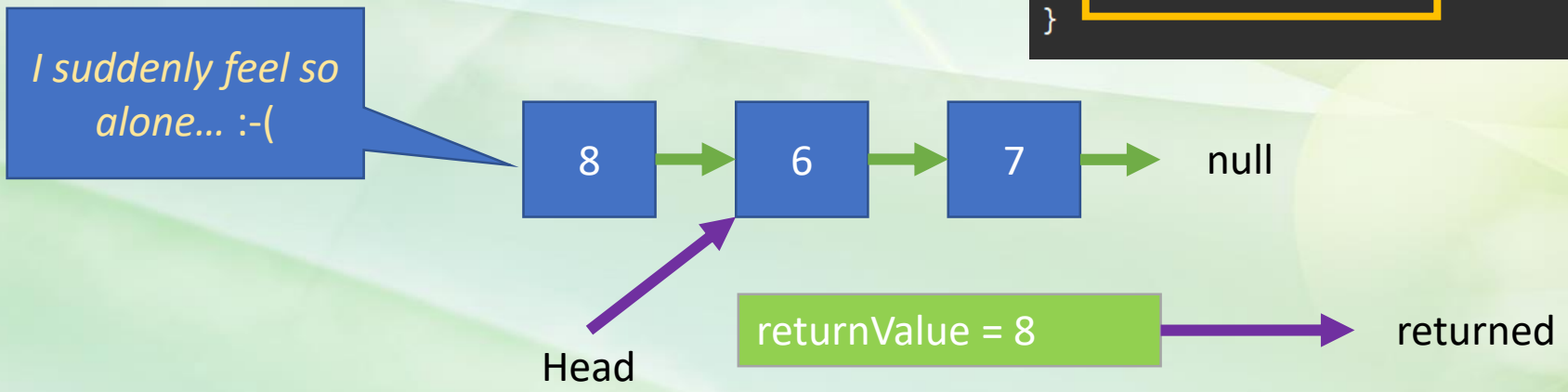
```
public int pop() {
    //we cannot pop if the list is empty
    if (isEmpty()) {
        throw new IllegalStateException("Error: cannot pop from empty list");
    }
    //store the value at the front of the list
    int returnValue = head.getValue();

    //remove the node at the front of the list
    head = head.getNext();

    //return the stored value
    return returnValue;
}
```



LinkedIntStack – Pop() function

- *Wait*, what happens to the node containing 8?
  - **No** variable connects to it either *directly* or *indirectly* (**No** inbound arrows)
- **Java garbage collection**

```
public int pop() {
    //we cannot pop if the list is empty
    if (isEmpty()) {
        throw new IllegalStateException("Error: cannot pop from empty list");
    }
    //store the value at the front of the list
    int returnValue = head.getValue();

    //remove the node at the front of the list
    head = head.getNext();

    //return the stored value
    return returnValue;
}
```



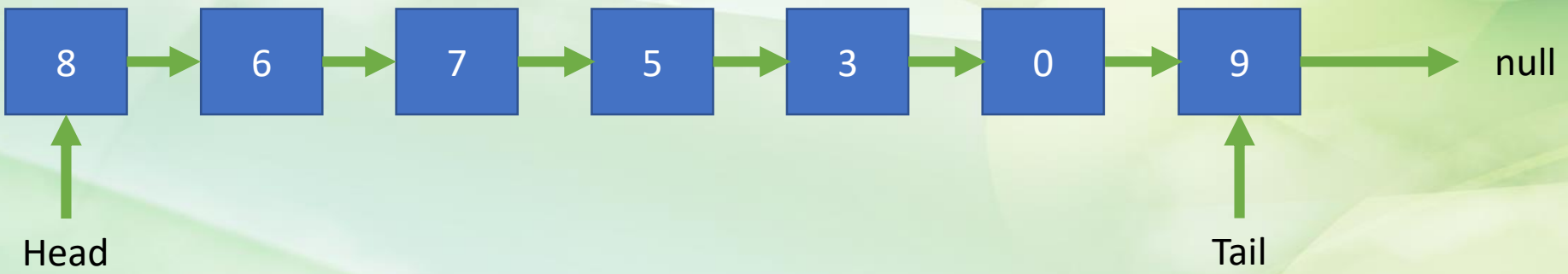
Review file...

StringQueue.java

Implementing a Queue

- When we implement a **queue**, we have two choices
  1. **Add** values *before the head*, **remove** values *from the tail*
  2. **Add** values *after the tail*, **remove** values *from the head*

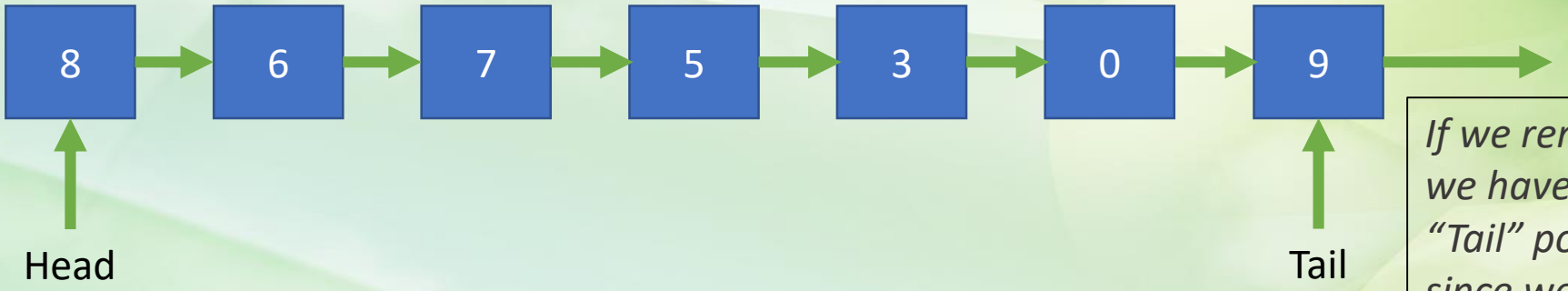
- Given what we have access to, which is **easier** to implement?



Implementing a Queue

- When we implement a **queue**, we have two choices
  1. Add values before the head, remove values from the tail
  2. **Add values *after the tail*, remove values *from the head***

- Given what we have access to, which is **easier** to implement?
  - **Option 2**, because it's hard to find the value before the tail
    - We only have a next point, not previous pointer



If we remove values from the tail, we have no way to reassign the "Tail" pointer to the previous node since we do not have a previous pointer, only a next pointer

StringQueue

- Notice that the **StringNode class** is implemented **inside** of **StringQueue**
  - This is ideal, since classes outside StringQueue should only use the **public interface** of StringQueue, and **shouldn't** even know nodes exist

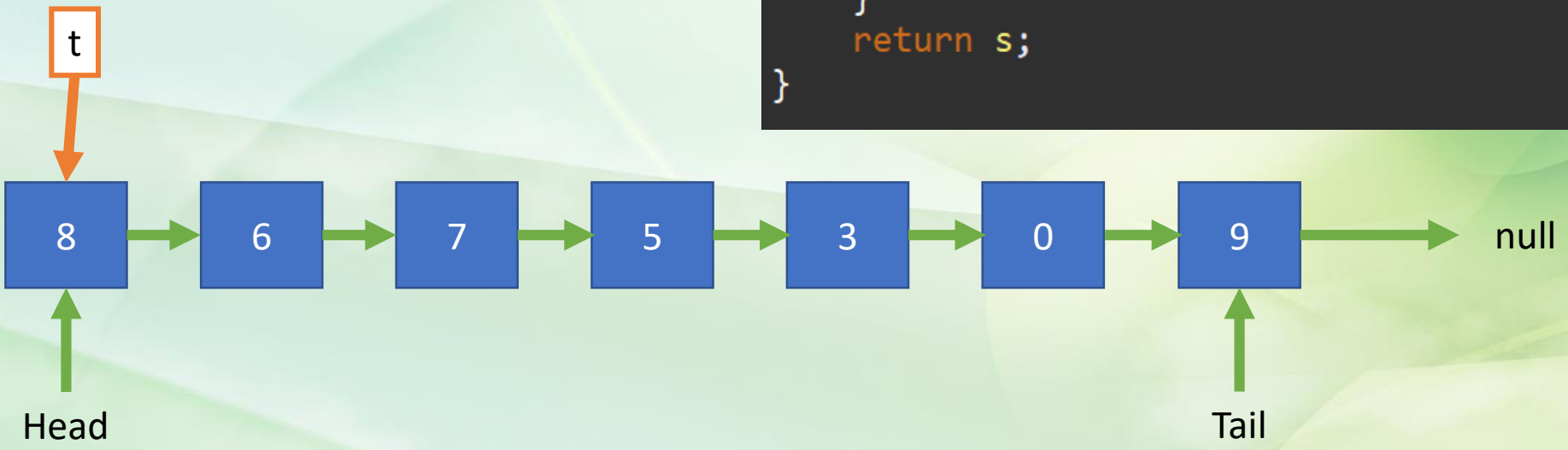
- Pay special attention to the *empty conditions* of
  - Enqueue
  - Dequeue (specifically, empty if you just removed the *last* element)

```
public class StringQueue {
    // StringNode Class --------------------
    private class StringNode {
        /* StringNode fields */
        String value;
        StringNode next;

        public StringNode(String value) {
            this.value = value;
            this.next = null;
        }
    }
    //-----------------------------------
```

Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

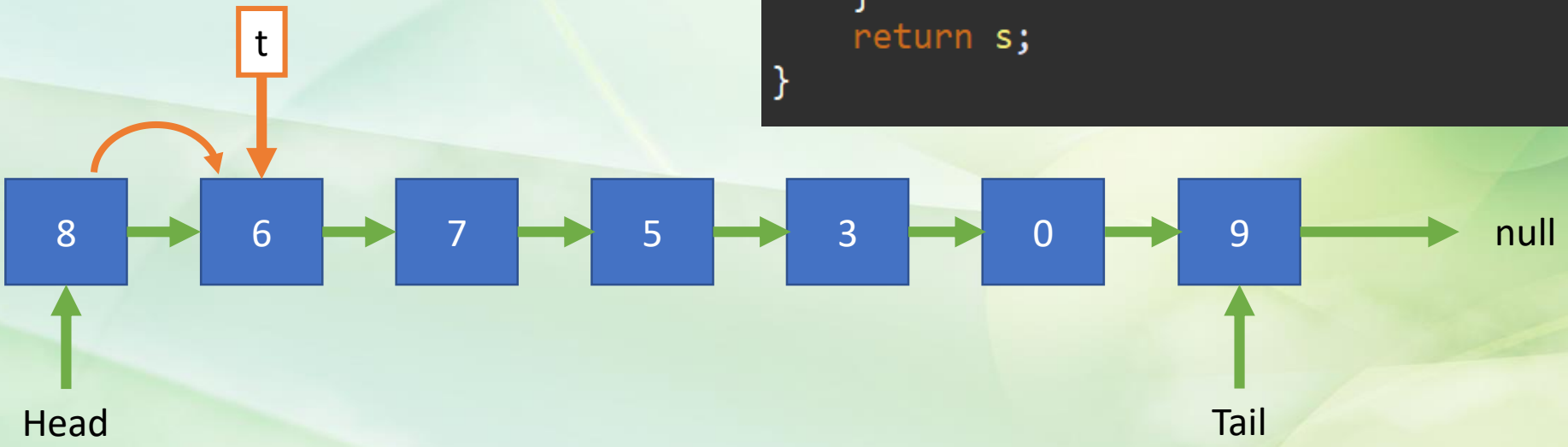
```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```



Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

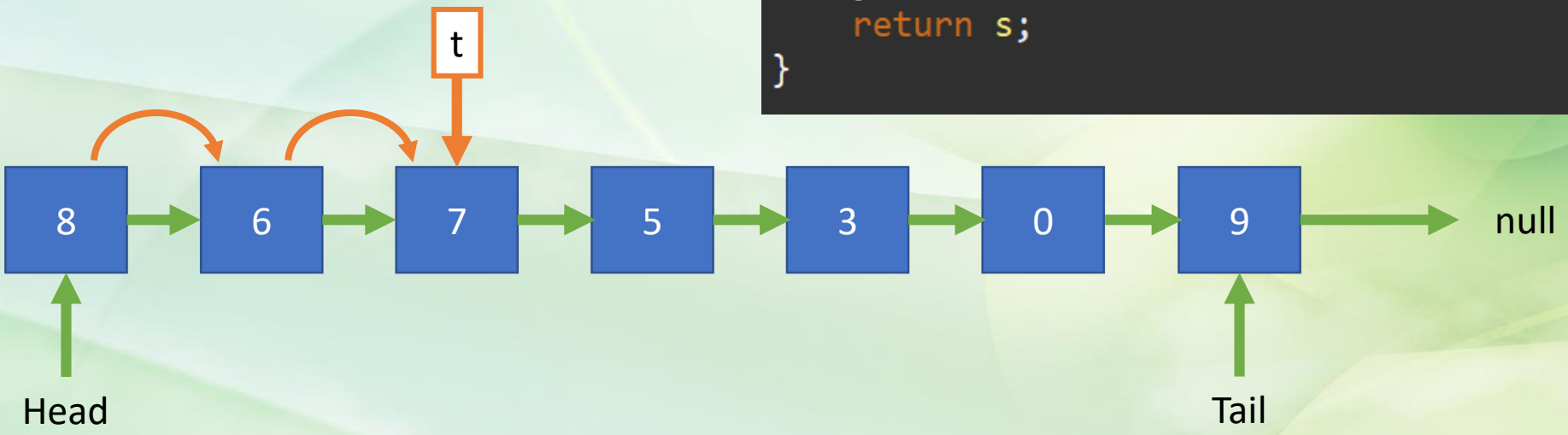
```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```



Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

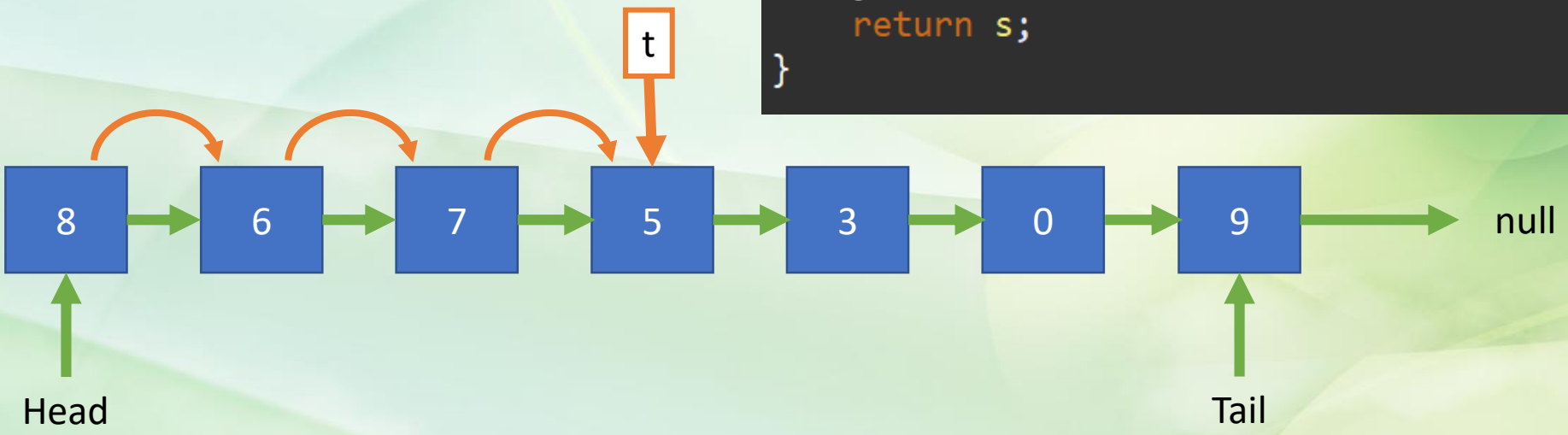
```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```



Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```

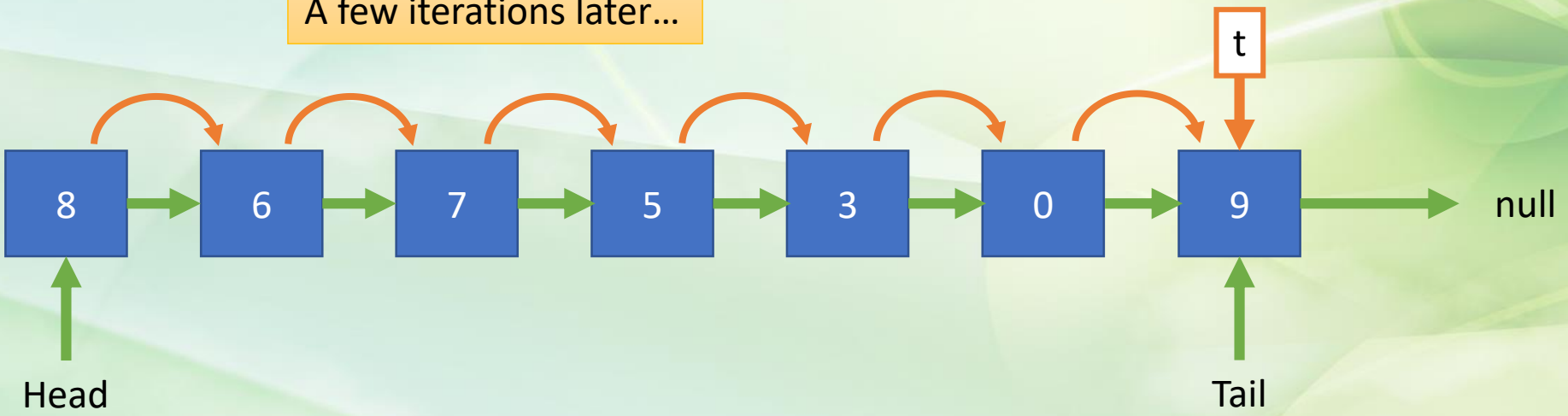


Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```

A few iterations later…

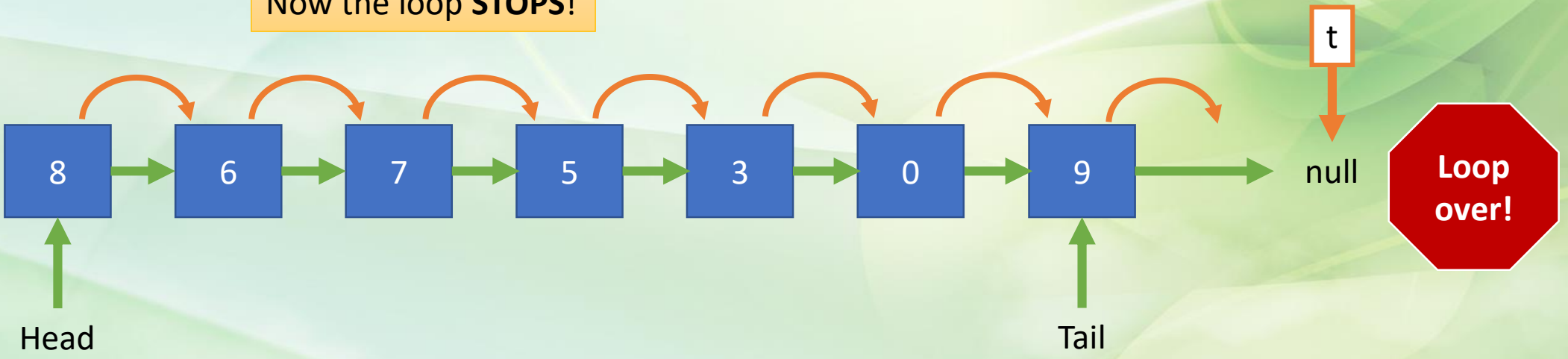


Looping through a Linked List

- Let's look at StringQueue's `toString()` function
  - **t** is a travelling node
    - It visits every node in the list, stopping when it reaches **null**

```
public String toString() {
    String s = "Head: ";
    if (isEmpty()) {
        s += "null";
        return s;
    }
    for (StringNode t = head; t != null; t = t.next) {
        s += " -> " + t.value;
    }
    return s;
}
```

Now the loop **STOPS**!



Adding to the tail example : enqueue() method

- Let's look at how adding to the tail of the linked list (queue implementation) would work

- **enqueue**() method

- [See Eclipse – StringQueue.java]

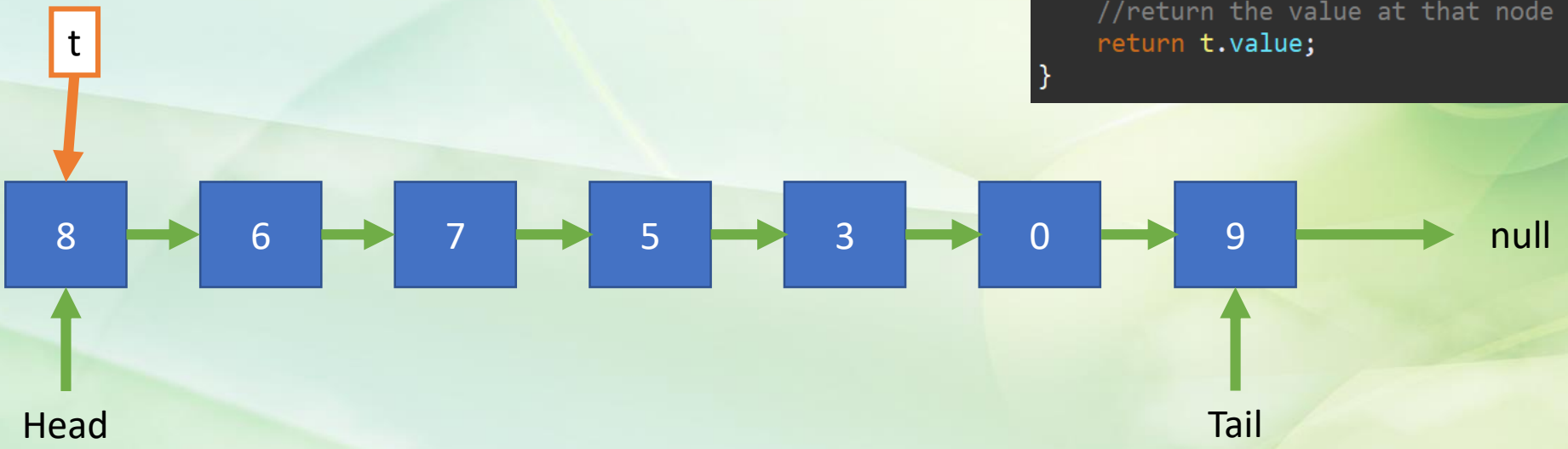
Review file…

MyList.java

MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

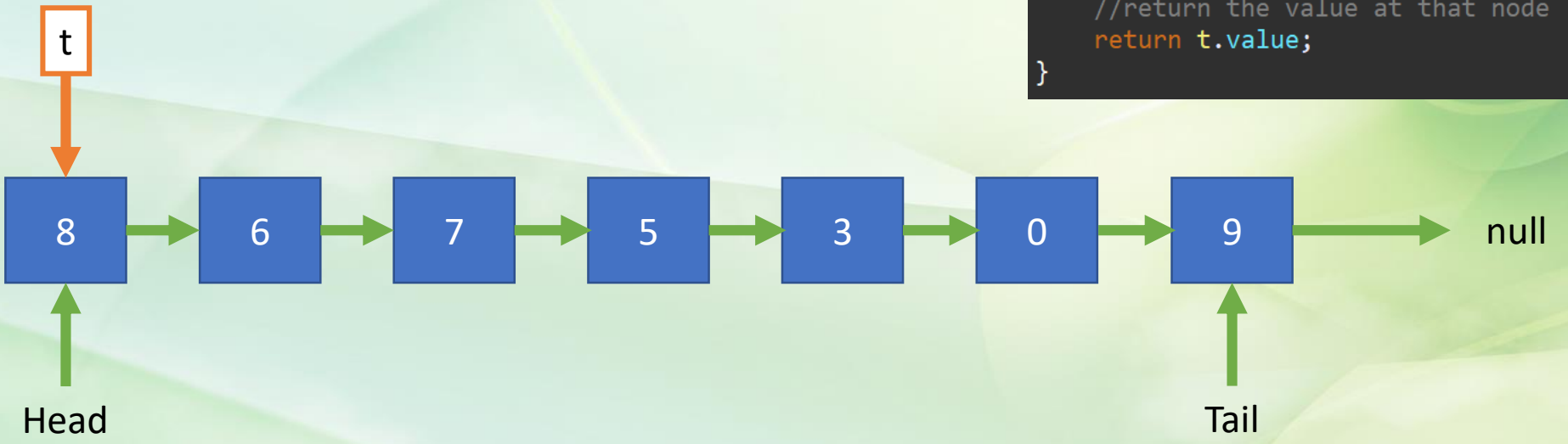


MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

index = 4

```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

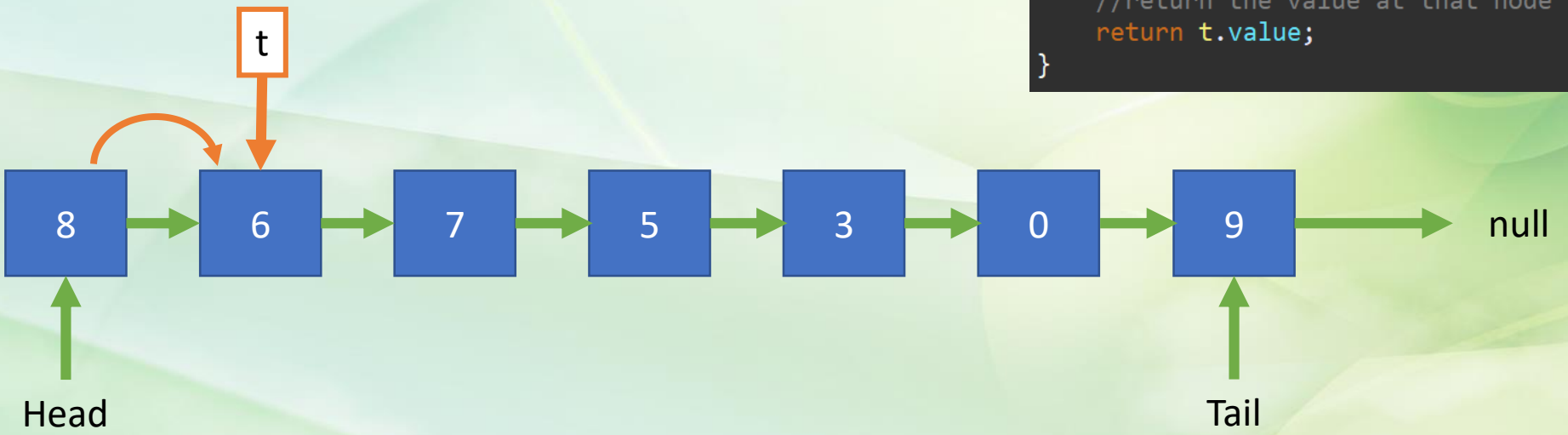


MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

index = 3

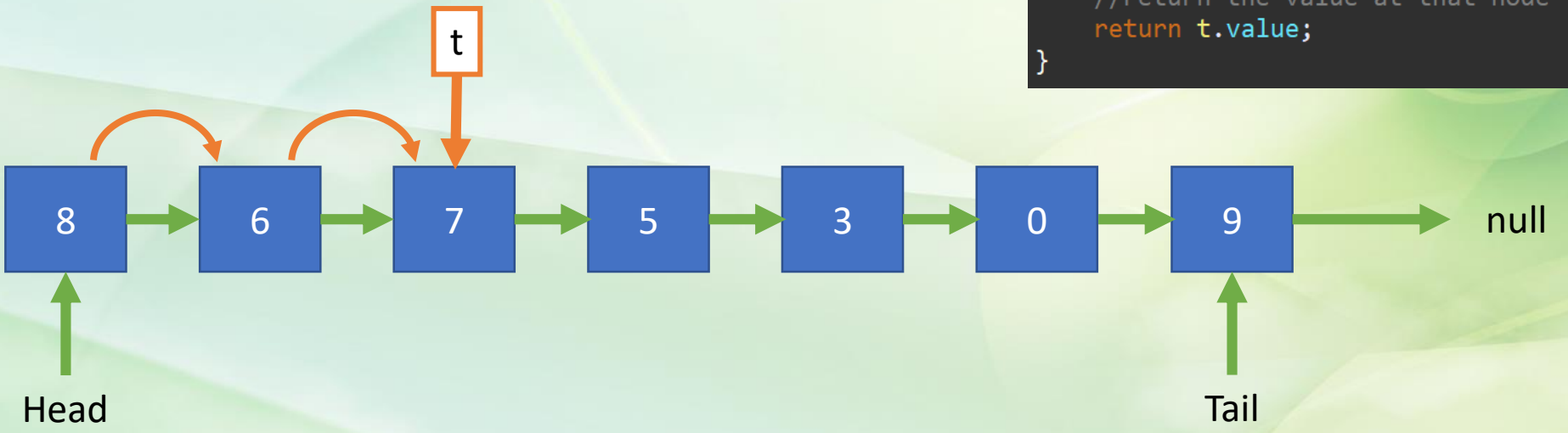


MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

index = 2

```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

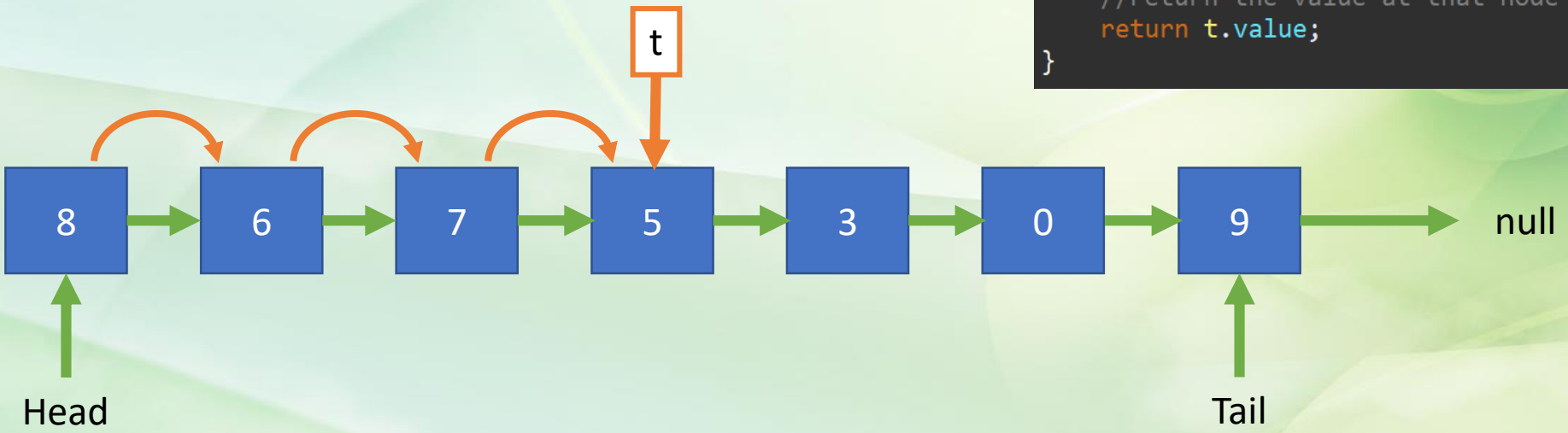


MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

index = 1

```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

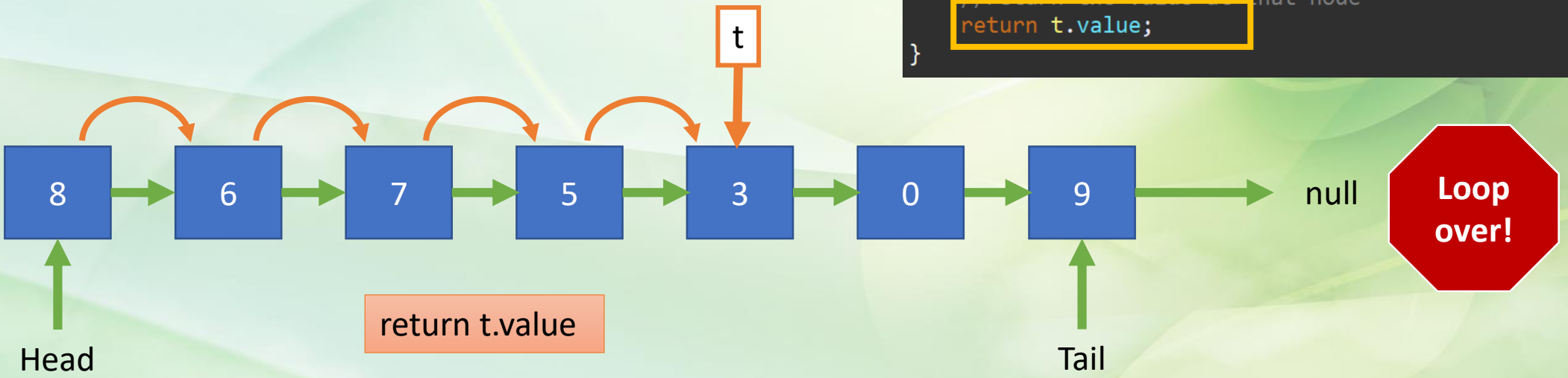


MyList – get(int index)

- Let's step through:
  - `list.get(4)`
- Use the list below

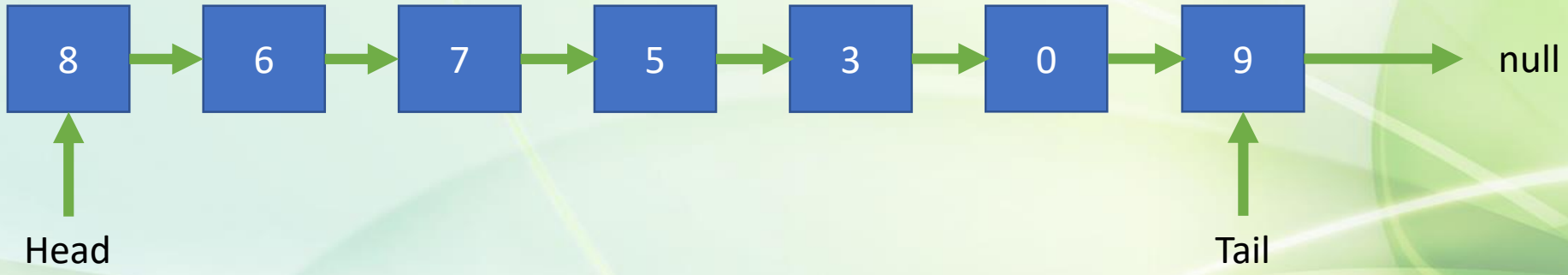
```java
public int get(int index) {
    //make sure index is in bounds
    if ((index < 0) || index >= size()) {
        //if it's not, throw an exception
        throw new IllegalArgumentException("Error: " +
    }
    //create our traveling node
    Node t = head;

    //loop to the index-th value
    while (index > 0) {
        index--;
        t = t.next;
    }

    //return the value at that node
    return t.value;
}
```

index = 0

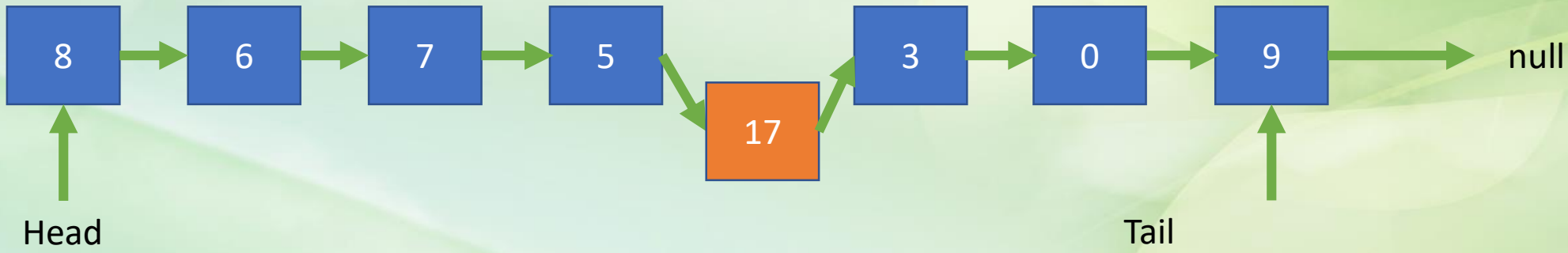


Add(int index, int value)

- Say we call "`add(4, 17)`"
  - First, let's start with this list

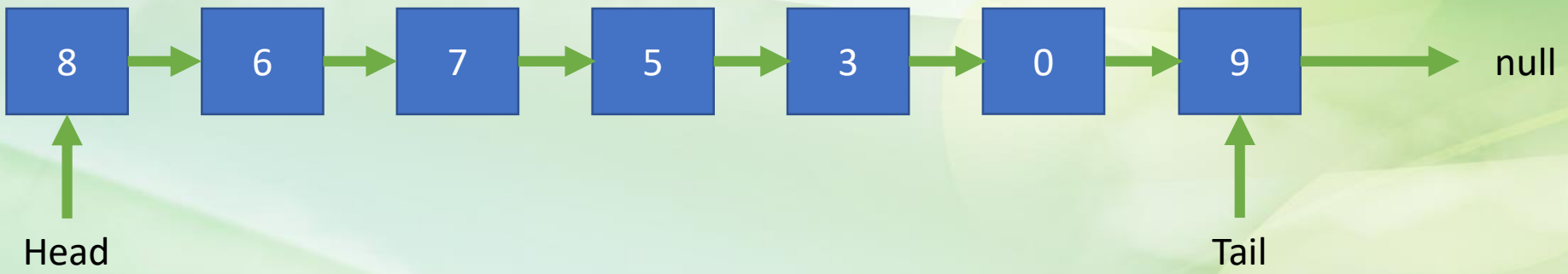


  - Our goal is to get to this list:



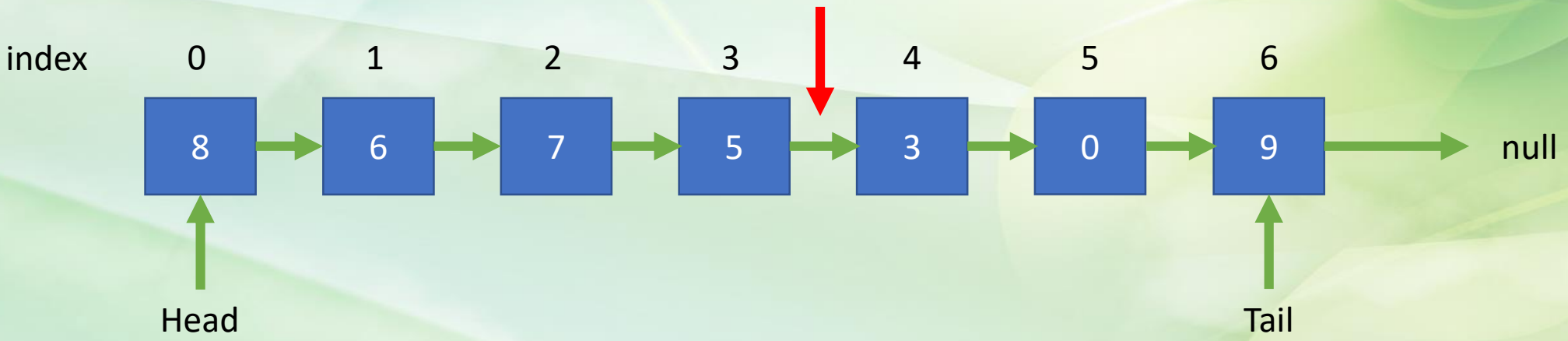
We can add at the end…
How do we add in the middle of the list?

- We've already talked about adding to the **end** and adding to the **front** of a list
  - `add(0, value)` – add at the **beginning** *(index position 0)*
  - `add(n, value)` – where **n** is the size() of the list, add at the **end**
- Let's focus on how we add in the **middle**



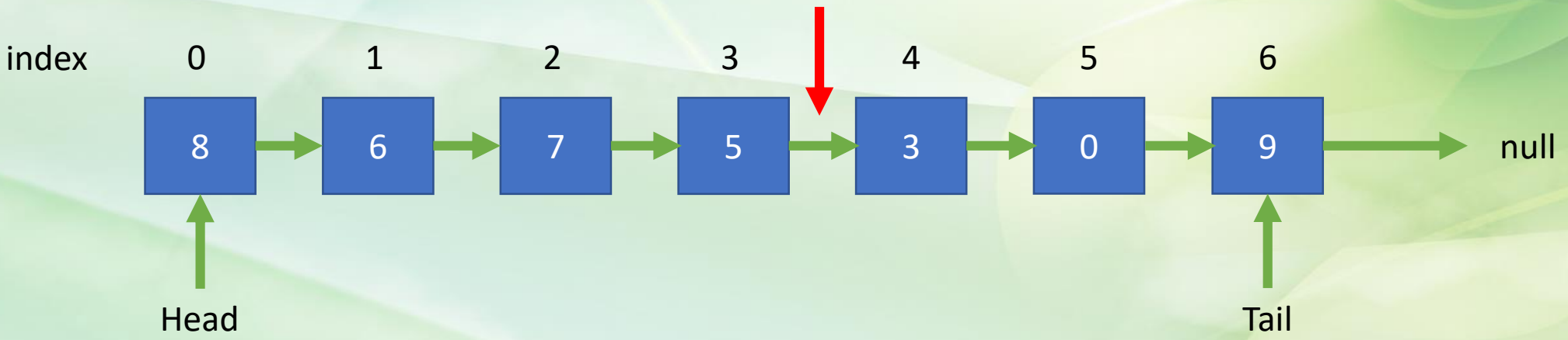
add(4, 17)

- If we are adding at index 4, then that means we are adding a node **AFTER index 3**



add(4, 17)

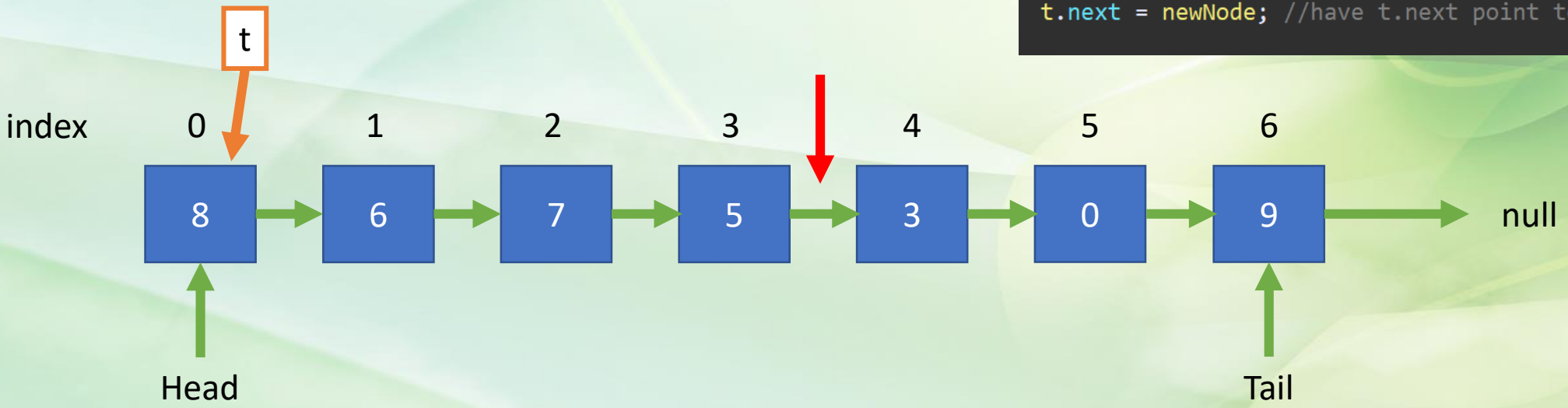
- If we are adding at index 4, then that means we are adding a node AFTER index 3
- Creating our **traveling** node and loop to the node "`index -1`" where `index` is where we are inserting



add(4, 17)

```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```

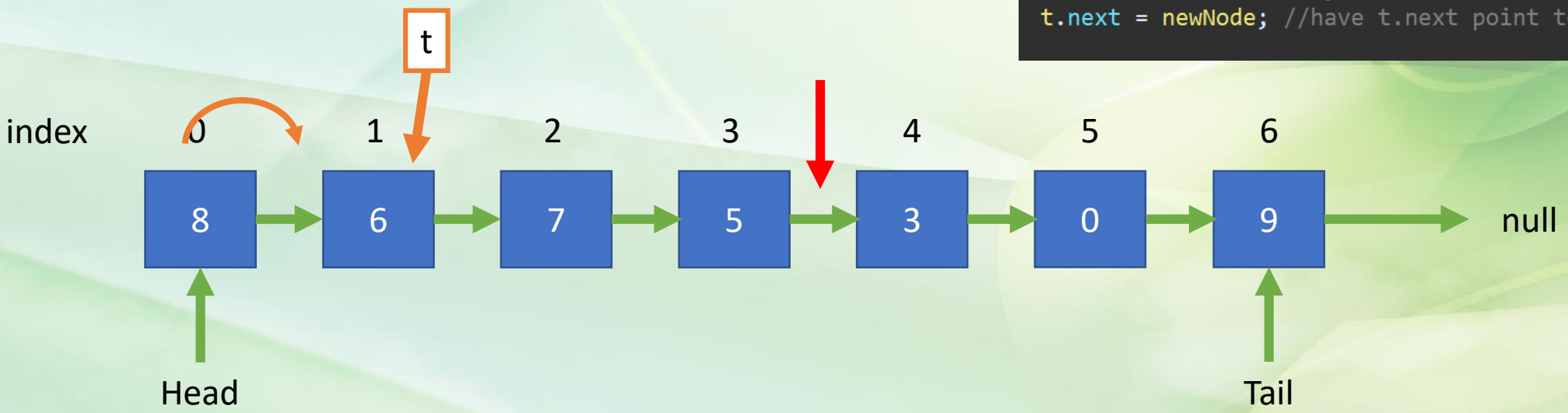
index = 5



add(4, 17)

```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```

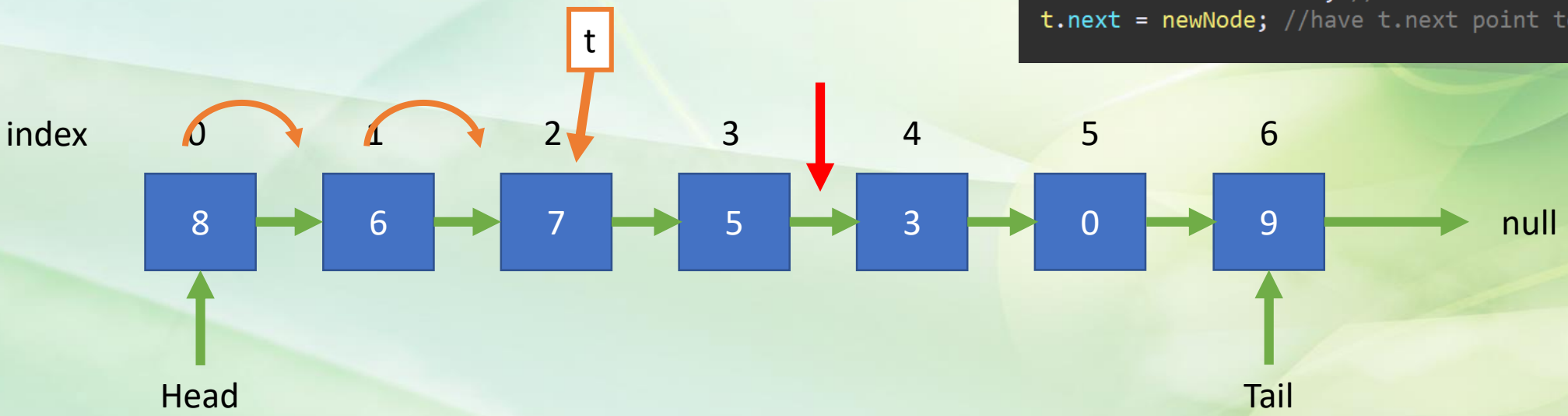
index = 3



add(4, 17)

```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```

index = 2

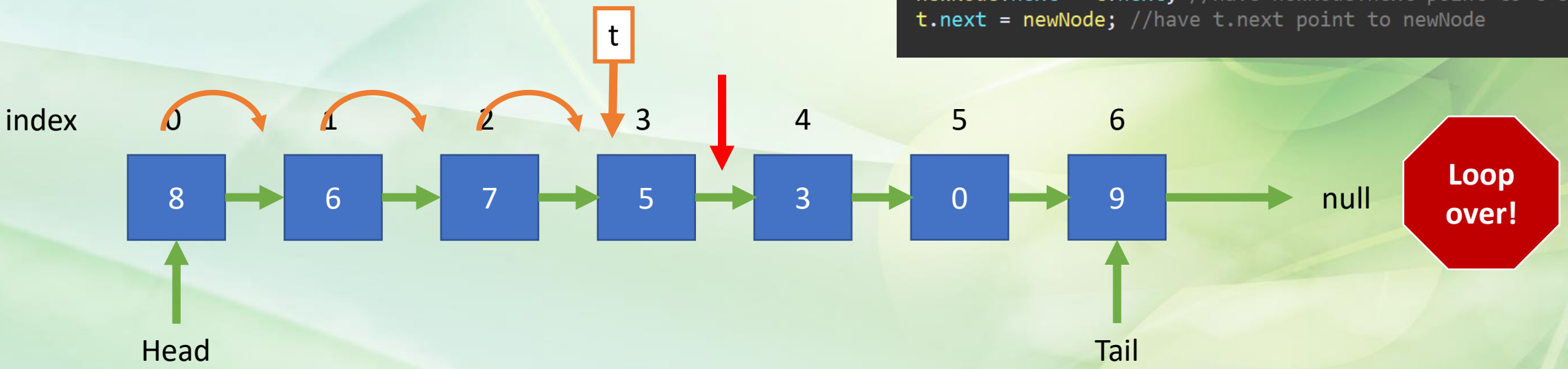


add(4, 17)

- We've reached the end of our loop!

```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```

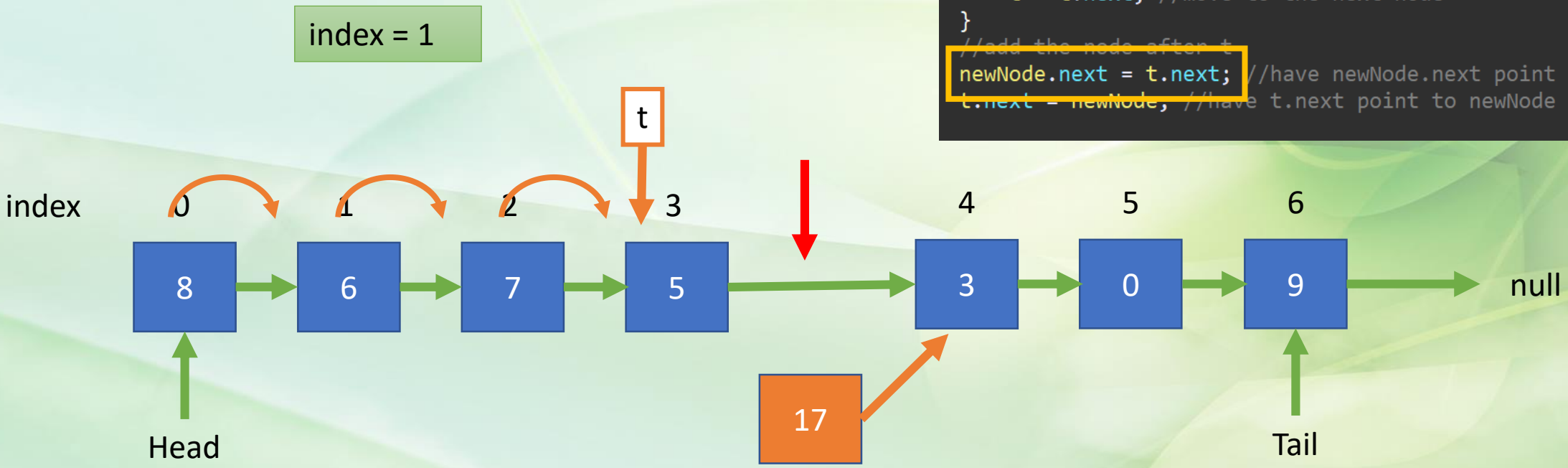
index = 1



add(4, 17)

- Now we link the “new node” in:
  - First `newNode.next = t.next`

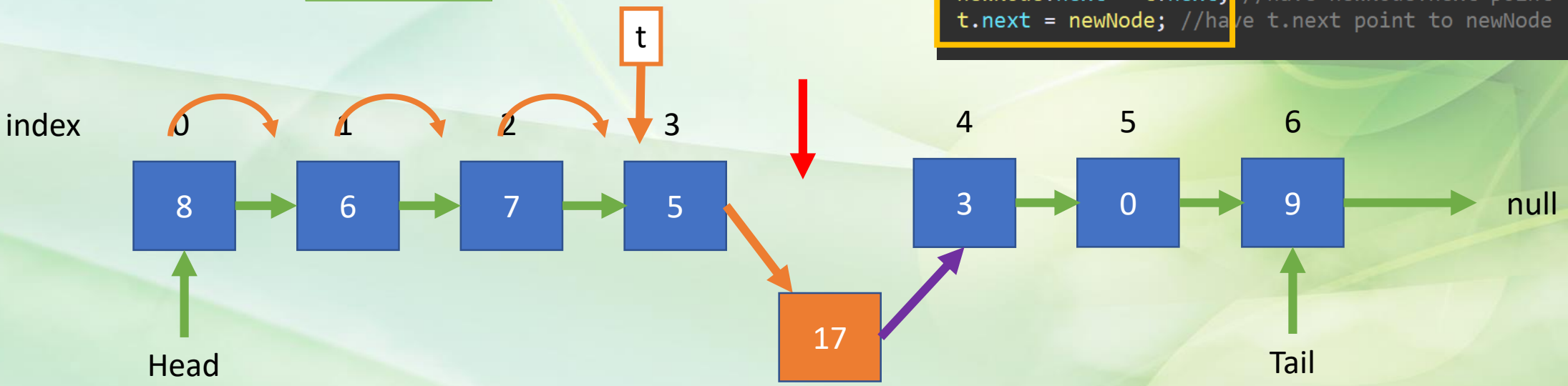
```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```



add(4, 17)

- Next we link the previous node into our new node
  - `t.next = newNode`
  - Done!!!! ☺

```
//create the traveling node
Node t = head;

//get the the node before index
while (index > 1) {
    index--; //decrement index to signal getting closer
    t = t.next; //move to the next node
}
//add the node after t
newNode.next = t.next; //have newNode.next point to t's next
t.next = newNode; //have t.next point to newNode
```



Linked List In-Class Activity

- Look at the method `contains(int target)` in the class MyList.java
  - This will help significantly with the in-class activity

- Write an implement **`find`**`(int target)`
  - Return the ***index*** of the first node containing the value "**target**"
  - Ex: List: 5 -> 3 -> 2 -> 3 -> 7
    - `find(3)` returns 1
  - Return **-1** if the value "target" is not in the list