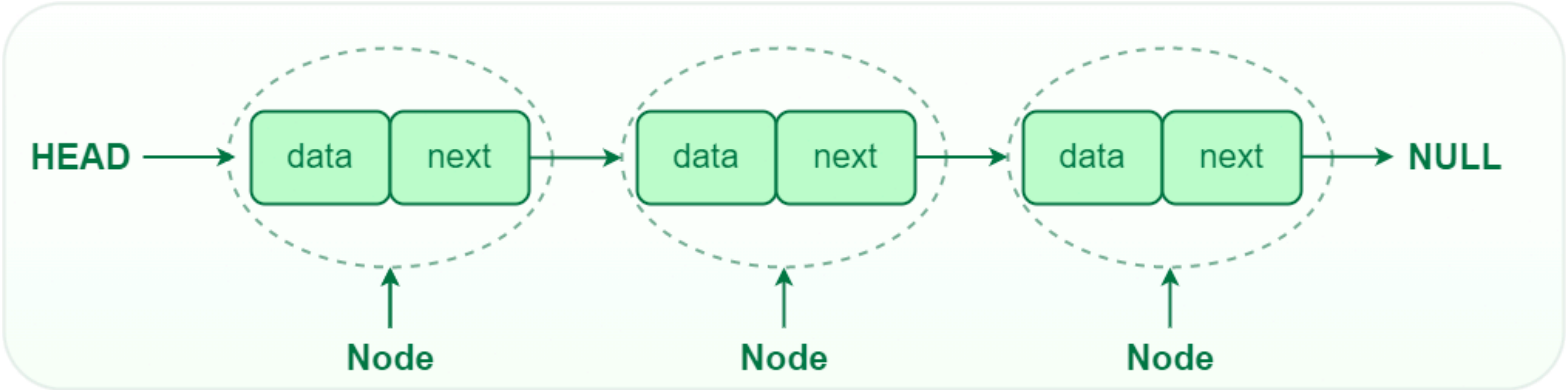
CS62 Class 6: Singly linked lists

Basic Data Structures



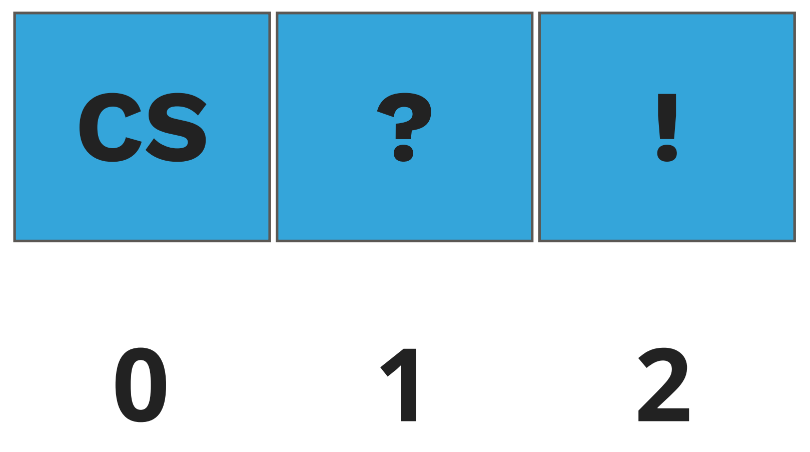
https://www.geeksforgeeks.org/introduction-to-singly-linked-list/

Linked Lists (conceptually)

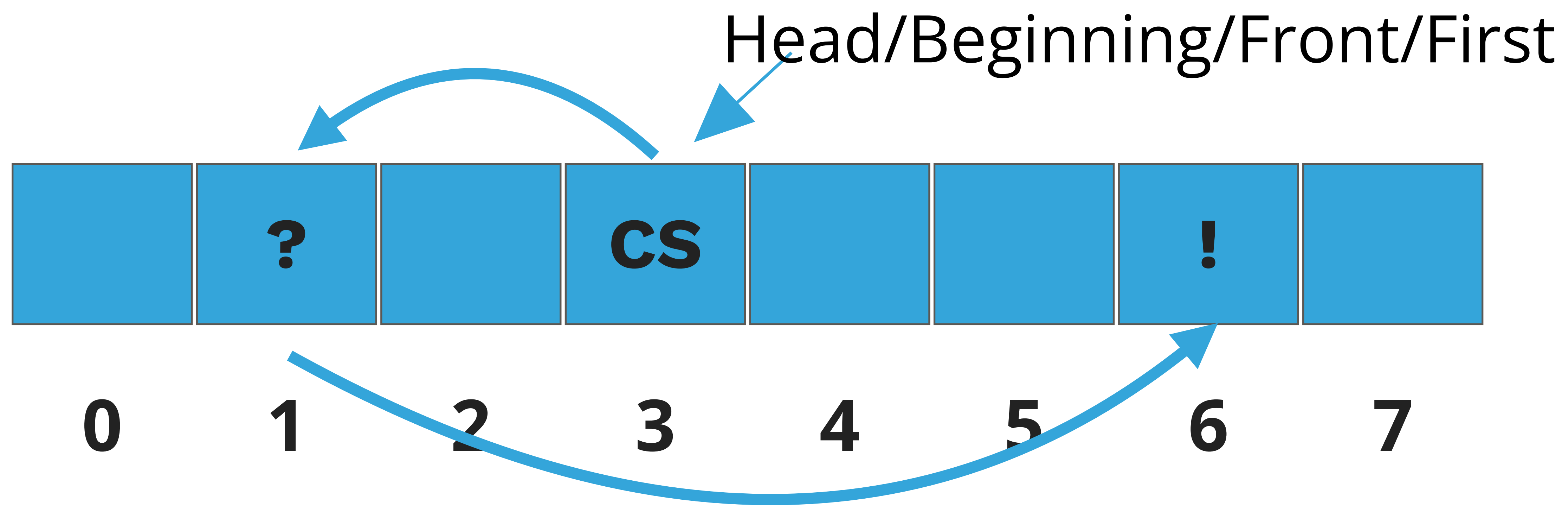
Linked Lists

- Dynamic linear data structures.
- In contrast to sequential data structures, linked data structures use pointers/links/ references from one object to another.
    - For example, the list of elements CS, ?, ! could be in very different memory locations. We just need a pointer to the head and links to subsequent elements to reconstruct it.

What the user's mental model is:

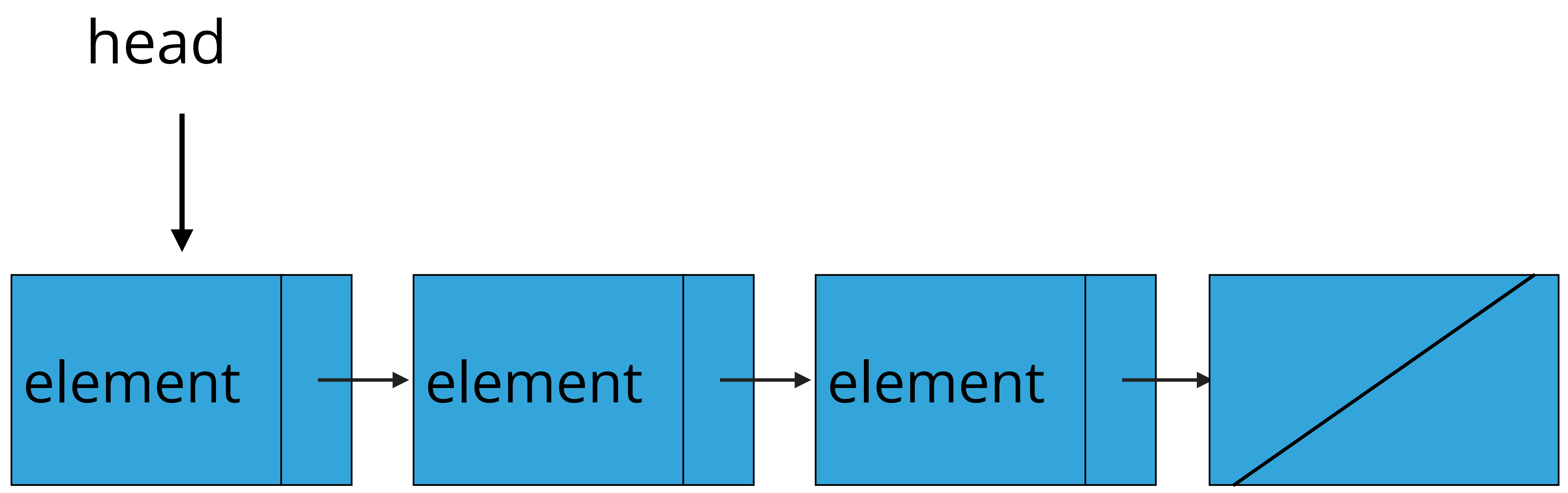


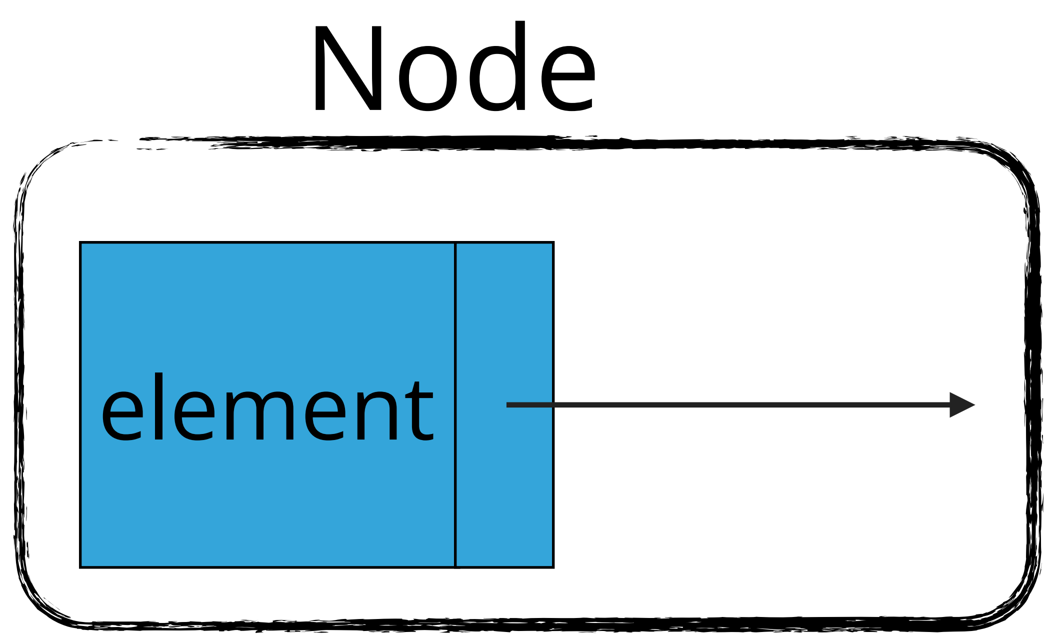
How it is in memory:



Recursive Definition of Singly Linked Lists

- A singly linked list is either empty (null) or a node having a reference to a singly linked list.
- Node: is a data type that holds elements of the same type and a reference to a next node.
- Singly linked lists only have a “next” pointer, and a head pointer
  - add() and remove() happen at the head



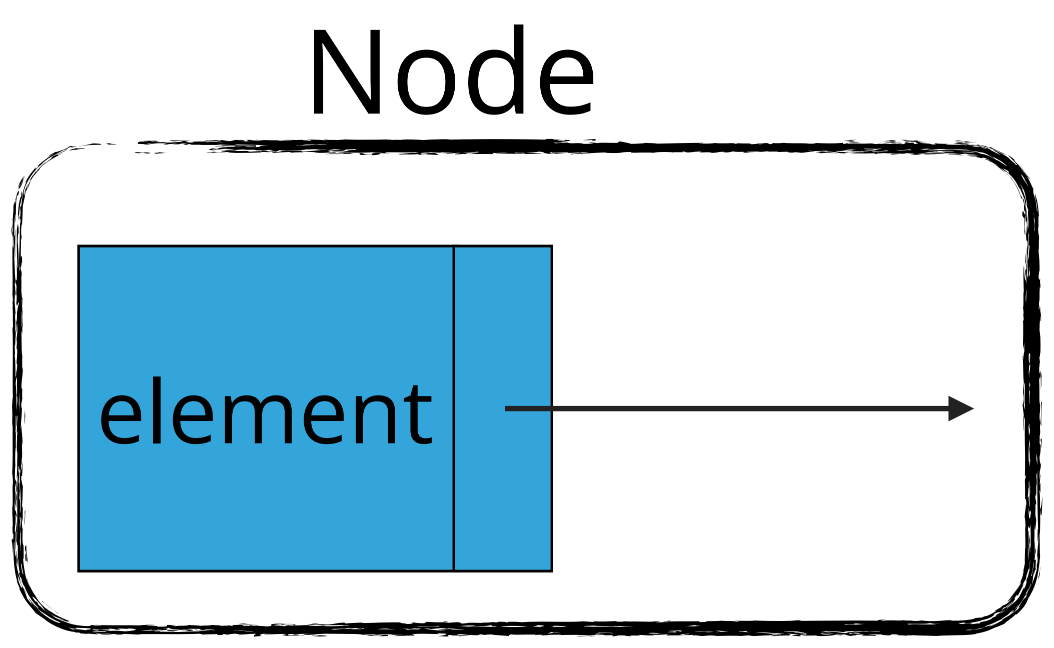


SinglyLinkedList & Node – Nested Classes

```
public class SinglyLinkedList<E> implements List<E> {

    private class Node {
        E element;
        Node next;
    }

    private Node head;
```

Nested classes are used when a class doesn't stand on its own—node may not make sense out of SLList

Instance variables, constructors, and methods of SLList typically go below nested class definition



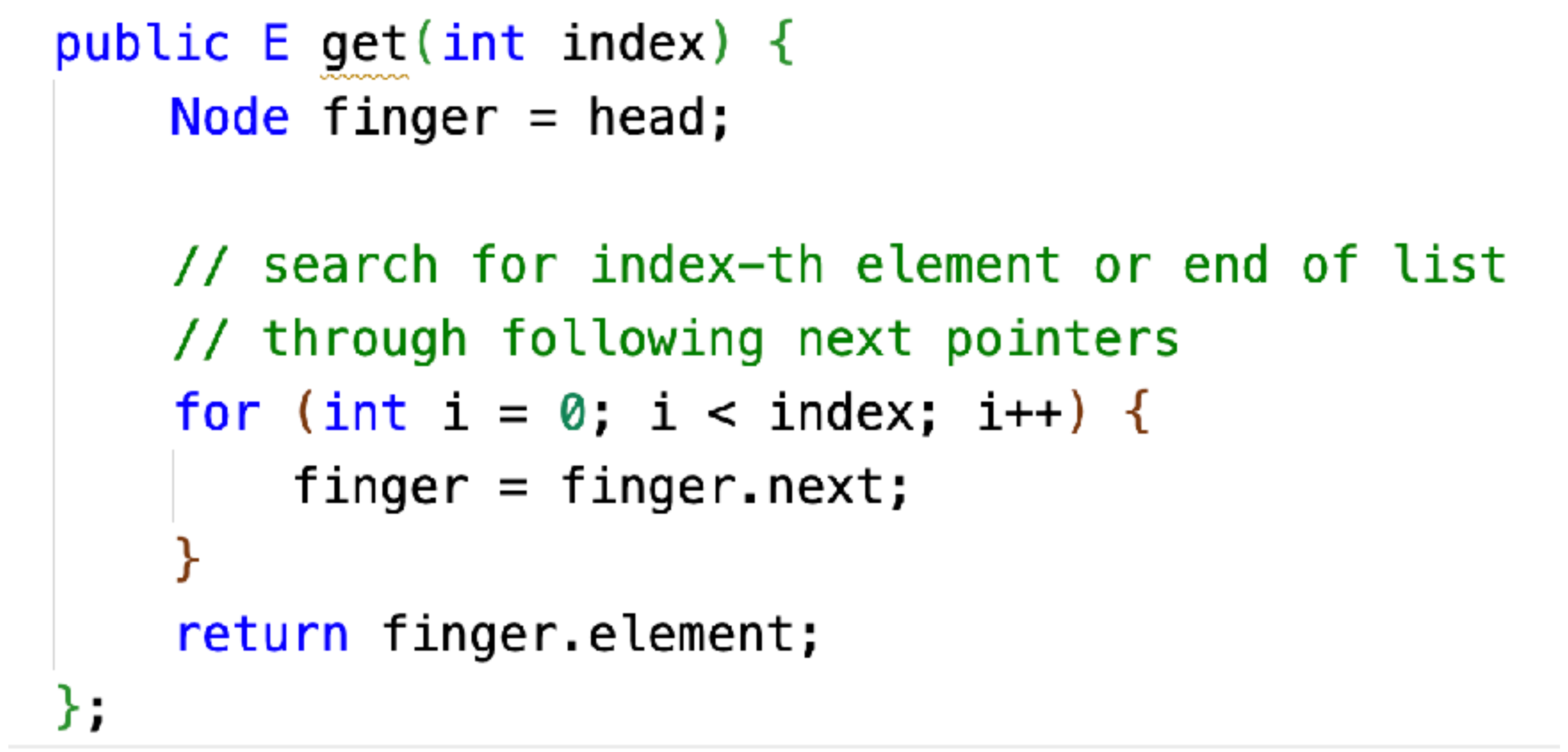
SinglyLinkedList: add (v1)

```java
//adds to the head of the list
public void add(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;
};
```

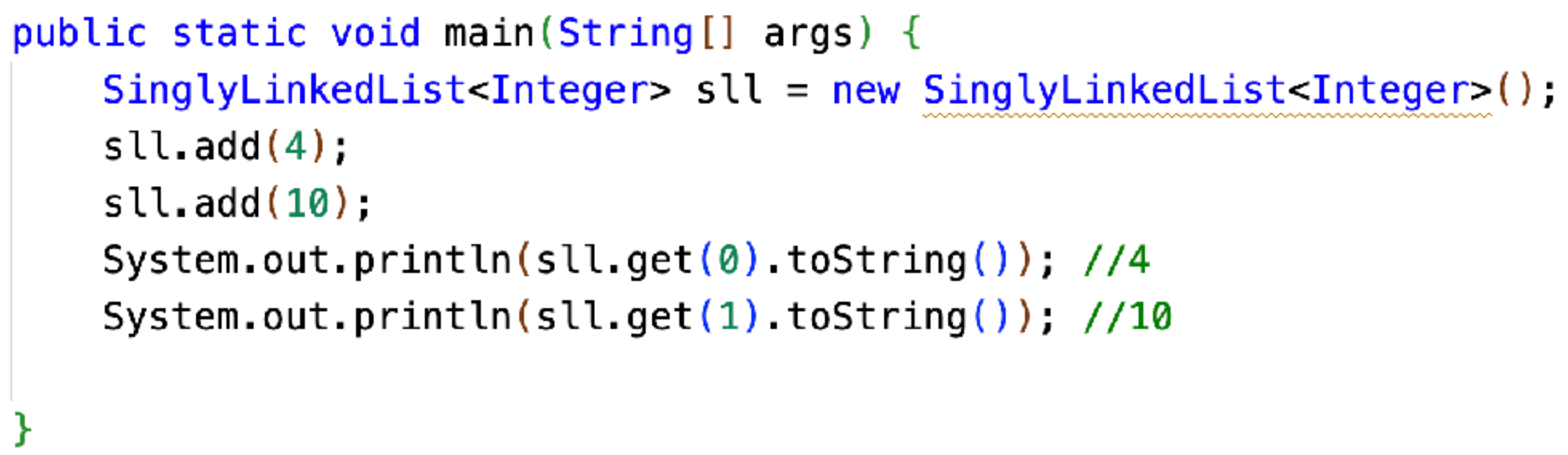
```java
public static void main(String[] args) {
    SinglyLinkedList<Integer> sll = new SinglyLinkedList<Integer>();
    sll.add(4);
    sll.add(10);
    System.out.println(sll); //just prints out memory address
}
```

Unfortunately, we don’t have a toString() yet, but let’s implement get(int index) for now to check.

SinglyLinkedList: get



```java
public E get(int index) {
    Node finger = head;

    // search for index-th element or end of list
    // through following next pointers
    for (int i = 0; i < index; i++) {
        finger = finger.next;
    }
    return finger.element;
};
```



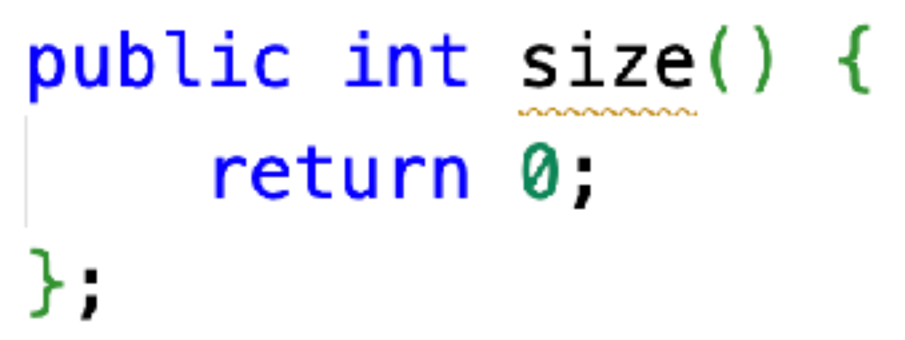
```java
public static void main(String[] args) {
    SinglyLinkedList<Integer> sll = new SinglyLinkedList<Integer>();
    sll.add(4);
    sll.add(10);
    System.out.println(sll.get(0).toString()); //4
    System.out.println(sll.get(1).toString()); //10

}
```

Why do we need to call .toString()? The type in SLL is Integer, not int — Integer is an reference type, not a primitive type

In Java, generics do not support primitive types (https://stackoverflow.com/questions/2721546/why-dont-java-generics-support-primitive-types)

SinglyLinkedList: size, recursive (v1)

A SLL, conceptually, is a recursive data structure. But how can we write recursive code if the class itself is not recursive? The size method doesn't take in any arguments, so calling size recursively is strange.

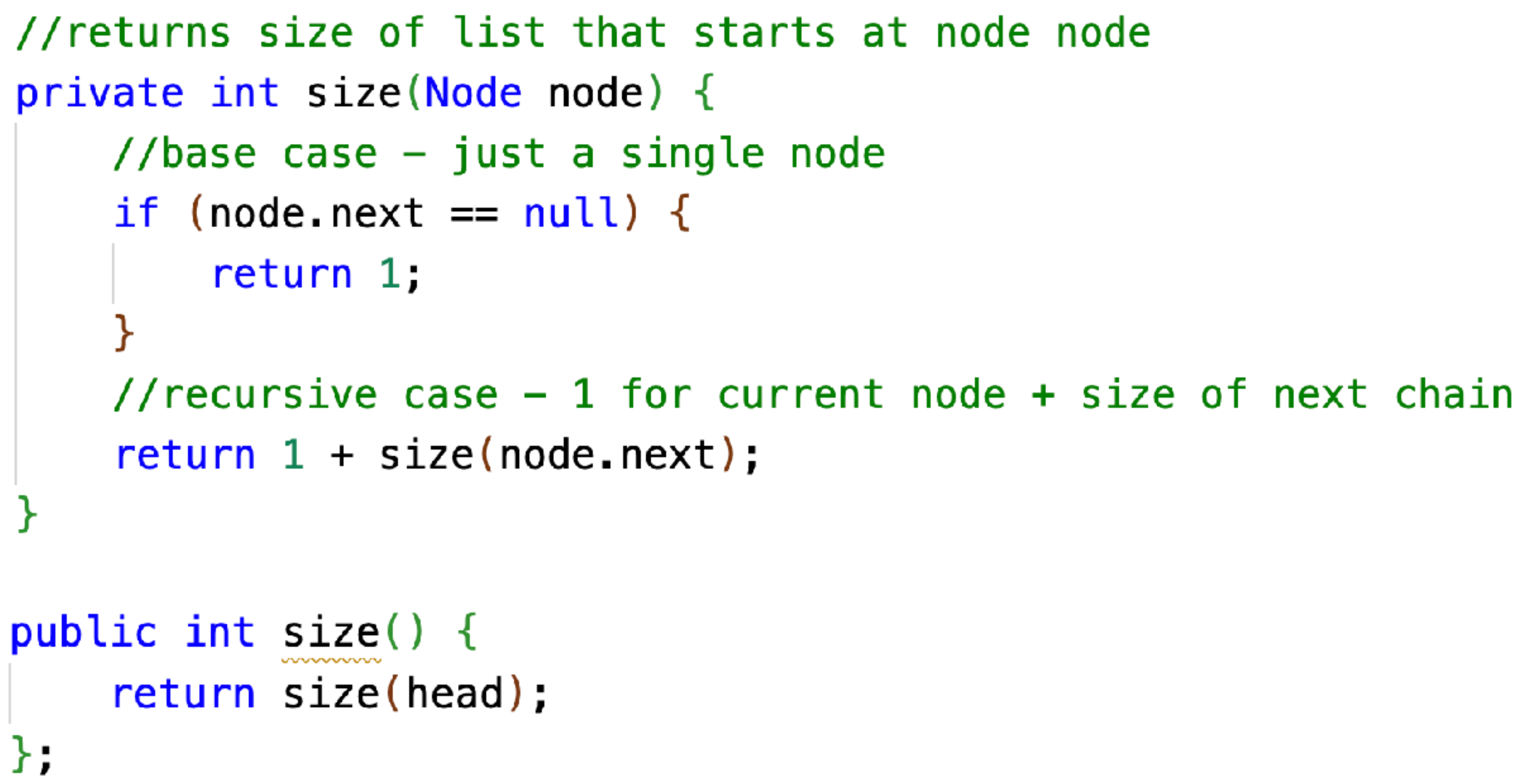


```
public int size() {
    return 0;
};
```

What is the base case?
How can you call a size helper recursively to get closer to the base case?

SinglyLinkedList: size, recursive (v1)

What is the base case? -> just one node in the SLL, size = 1
How can you call a size helper recursively to get closer to the base case? -> 1 + size(rest of SLL)



```java
//returns size of list that starts at node node
private int size(Node node) {
    //base case - just a single node
    if (node.next == null) {
        return 1;
    }
    //recursive case - 1 for current node + size of next chain
    return 1 + size(node.next);
}

public int size() {
    return size(head);
};
```

Private recursive helper methods

To implement a recursive method in a class that is not itself recursive (e.g. SLList):

1. Create a private recursive helper method.

2. Have the public method call the private recursive helper method.

```java
//returns size of list that starts at node node
private int size(Node node) {
    //base case - just a single node
    if (node.next == null) {
        return 1;
    }
    //recursive case - 1 for current node + size of next chain
    return 1 + size(node.next);
}

public int size() {
    return size(head);
};
```

size v1 run time

How efficient is size?

Suppose size takes 2 seconds on a list of size 1,000.

How long will it take on a list of size 1,000,000?

slido.com code #626

A. 0.002 seconds.
B. 2 seconds.
C. 2,000 seconds.
D. 2,000,000 seconds.

```
//returns size of list that starts at node node
private int size(Node node) {
    //base case - just a single node
    if (node.next == null) {
        return 1;
    }
    //recursive case - 1 for current node + size of next chain
    return 1 + size(node.next);
}

public int size() {
    return size(head);
};
```

size v1 run time

Size runs in O(n) because it iterates through the whole list.

Thus, if we double the input, we double the runtime (1:1 scale).

So 100x'ing the input 100x's the run time.

How efficient is size?

Suppose size takes 2 seconds on a list of size 1,000.

How long will it take on a list of size 1,000,000?

A. 0.002 seconds.
B. 2 seconds.
C. 2,000 seconds.
D. 2,000,000 seconds.

```java
//returns size of list that starts at node node
private int size(Node node) {
    //base case - just a single node
    if (node.next == null) {
        return 1;
    }
    //recursive case - 1 for current node + size of next chain
    return 1 + size(node.next);
}

public int size() {
    return size(head);
};
```

Towards are more efficient size v2

We can do better. What if I told you we can write size so it always runs in O(1) with an inclusion of a single variable?

```java
public class SinglyLinkedList<E> implements List<E> {

    private class Node {
        E element;
        Node next;
    }

    private Node head;
    private int size;
```

```java
public int size() {
    return size;
}
```

```java
//adds to the head of the list
public void add(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;

    //increment size
    size++;
};
```

Let's just keep track of size and update size when we add/remove stuff
It's redundant (we could always calculate size from the list), but it saves us time.

“There ain’t no free lunch”

By maintaining a special size variable that **caches** (saves) the size of the list, we can put aside data to speed up retrieval (a common SWE practice).

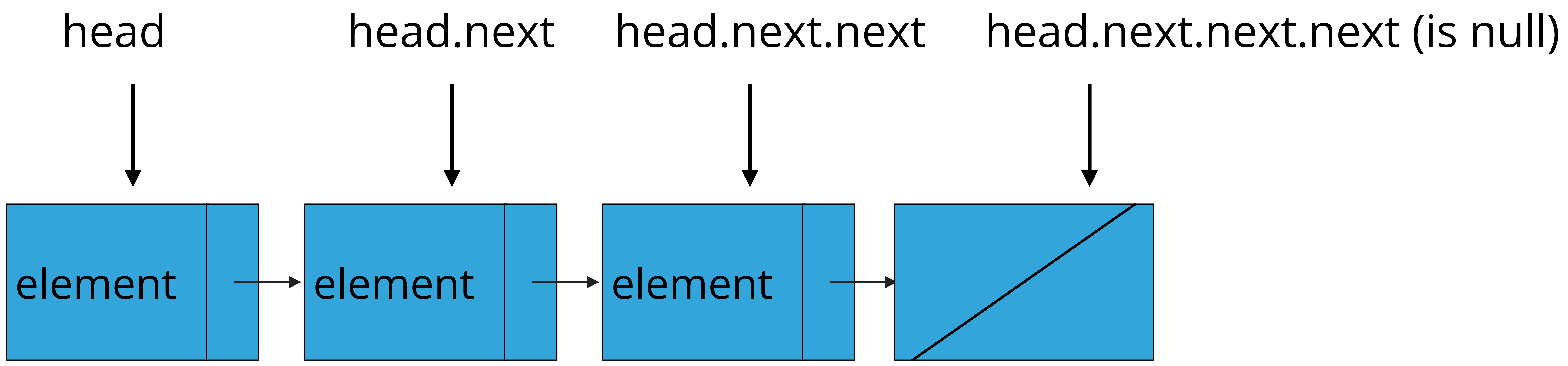
“There ain’t no free lunch”: we need to add a redundant variable, but spreading work over each add call (O(1) work) is a net win.

Pointer danger

In a SLL, we use .next to traverse through the list.

However, there's nothing preventing us from chaining on .next calls—e.g., calling node.next.next.

It gets conceptually messy, so it may not be the best SWE design to mess too much with .next pointers, but it's important to understand and trace.



Worksheet time!

- Do Qs 1&2 on your worksheet.

Suppose x is a reference to a Node and that node is not the last one on the linked list. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

Suppose x and t are references to different Nodes. What is the effect of the following code fragment?

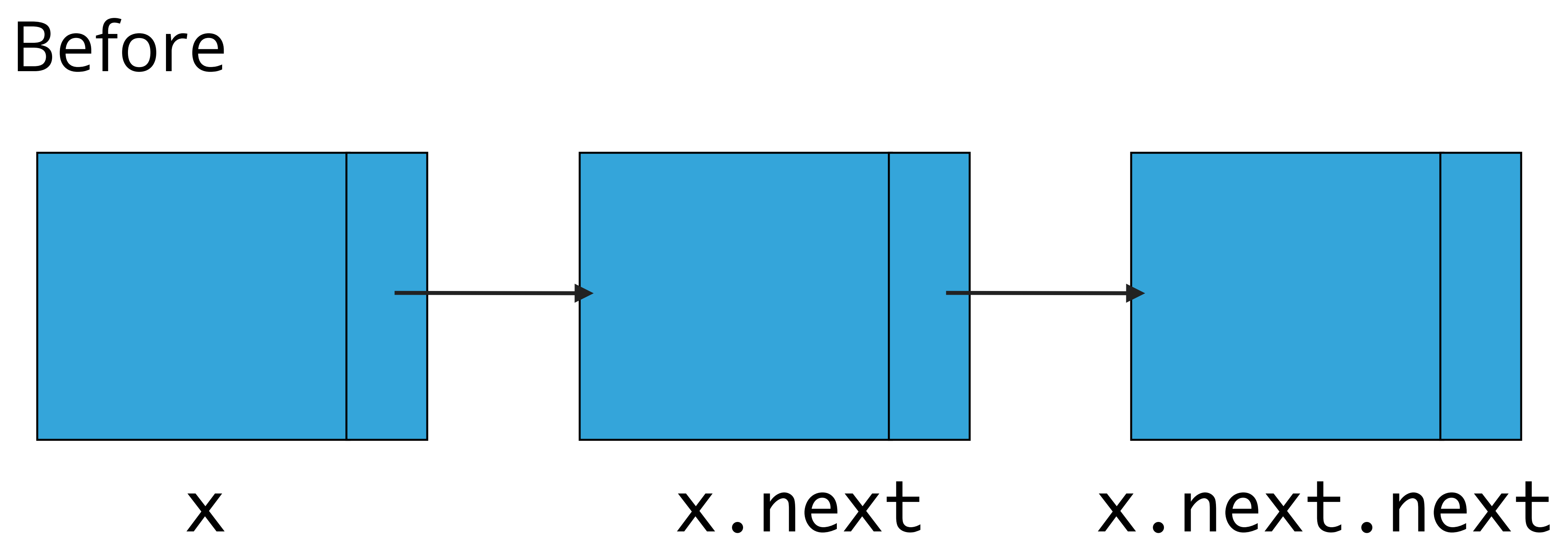
```
t.next = x.next;

x.next = t;
```

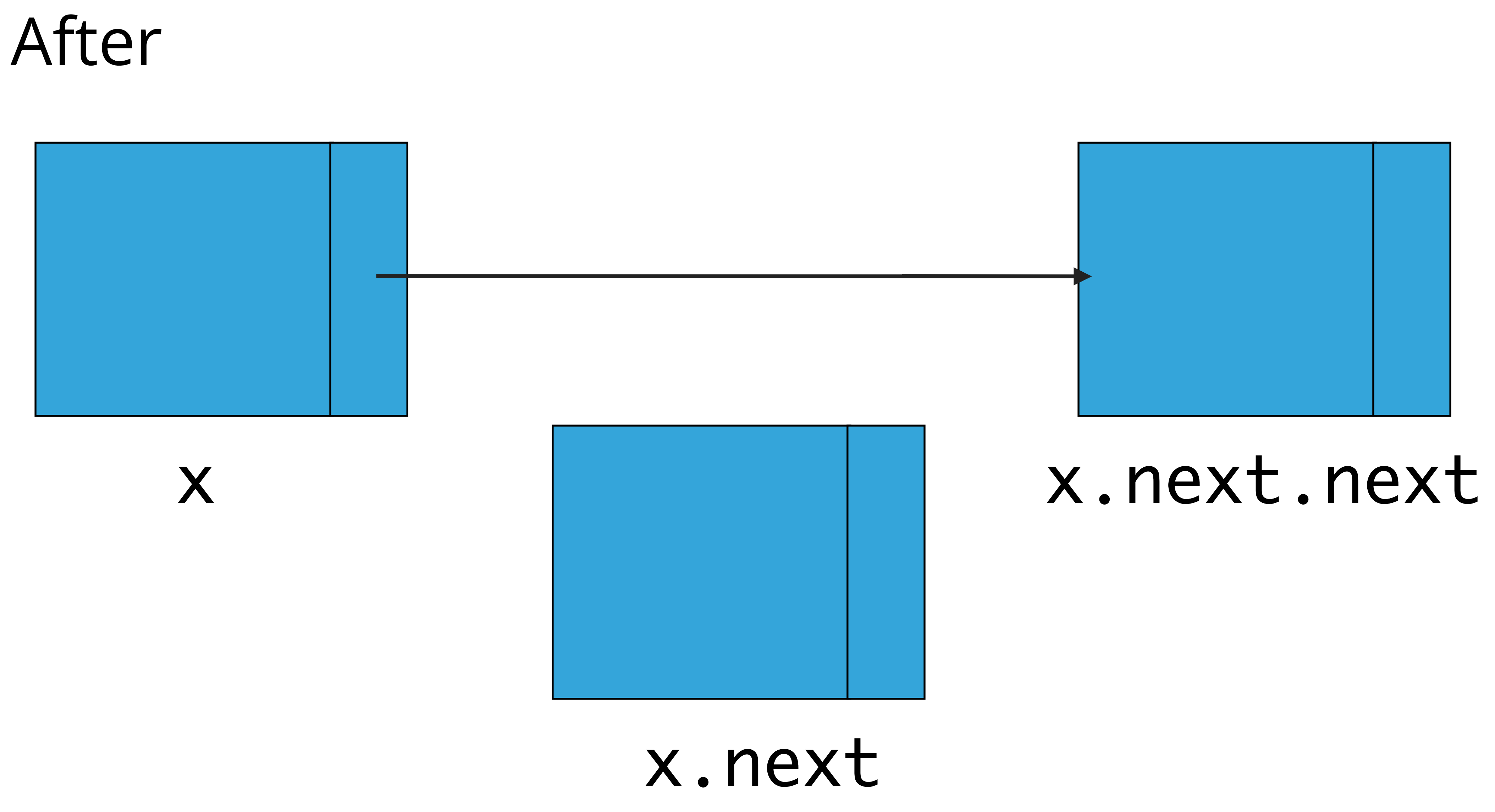
Worksheet answers

Suppose x is a reference to a Node and that node is not the last one on the linked list. What is the effect of the following code fragment?

```
x.next = x.next.next;
```

It removes the node after x.



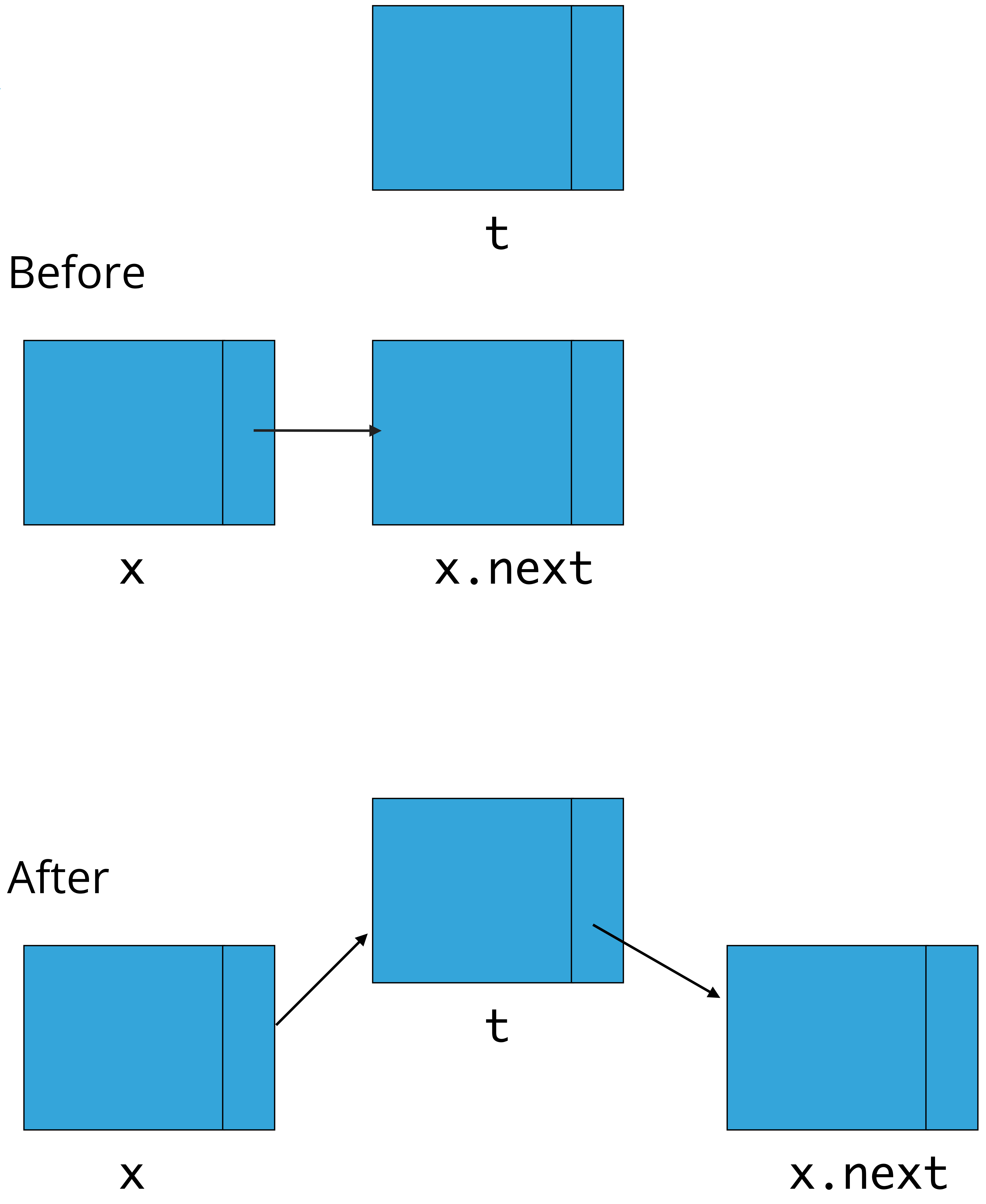


Worksheet answers

Suppose x and t are references to different Nodes. What is the effect of the following code fragment?

```
t.next = x.next;

x.next = t;
```

It inserts t between x and x.next.



More Implementation

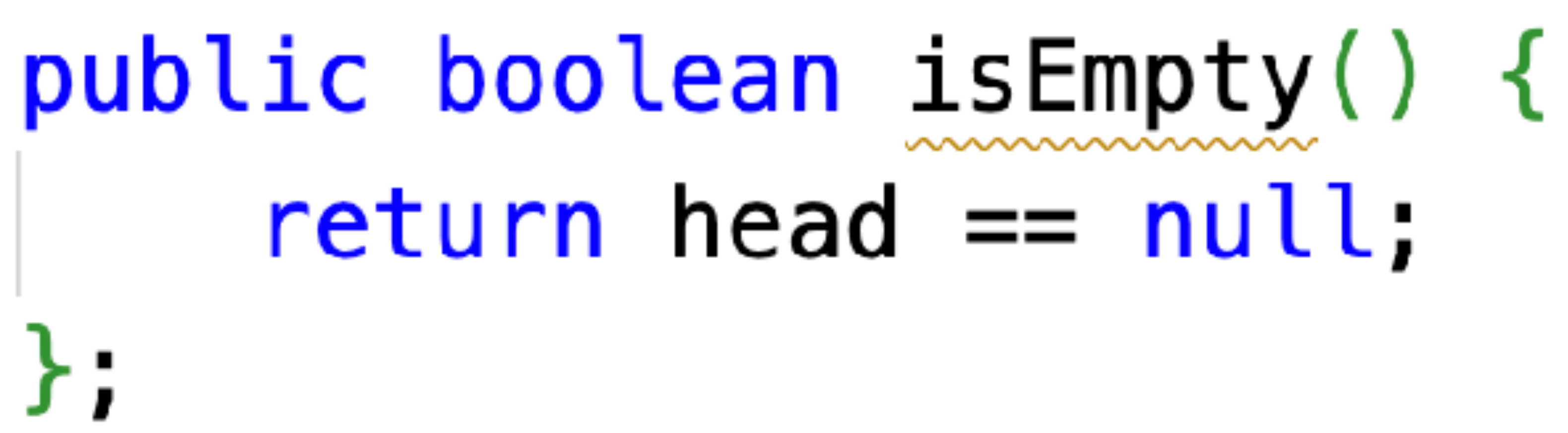
Reminder: Interface List

```
public interface List <E> {
    void add(E element); ✅
    void add(int index, E element);
    void clear();
    E get(int index); ✅
    boolean isEmpty();
    E remove();
    E remove(int index);
    E set(int index, E element);
    int size(); ✅
}
```

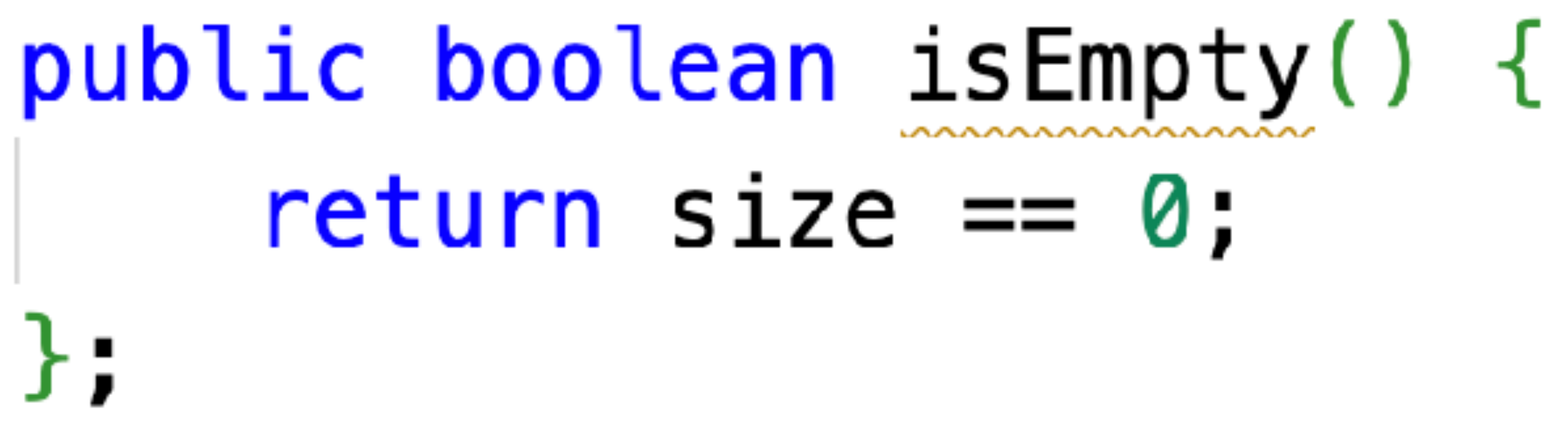
We’ve already implemented 3 abstract methods for our SLList so far…

isEmpty()

What does it mean for a SLL to be empty? (There are 2 options)



```java
public boolean isEmpty() {
    return head == null;
};
```



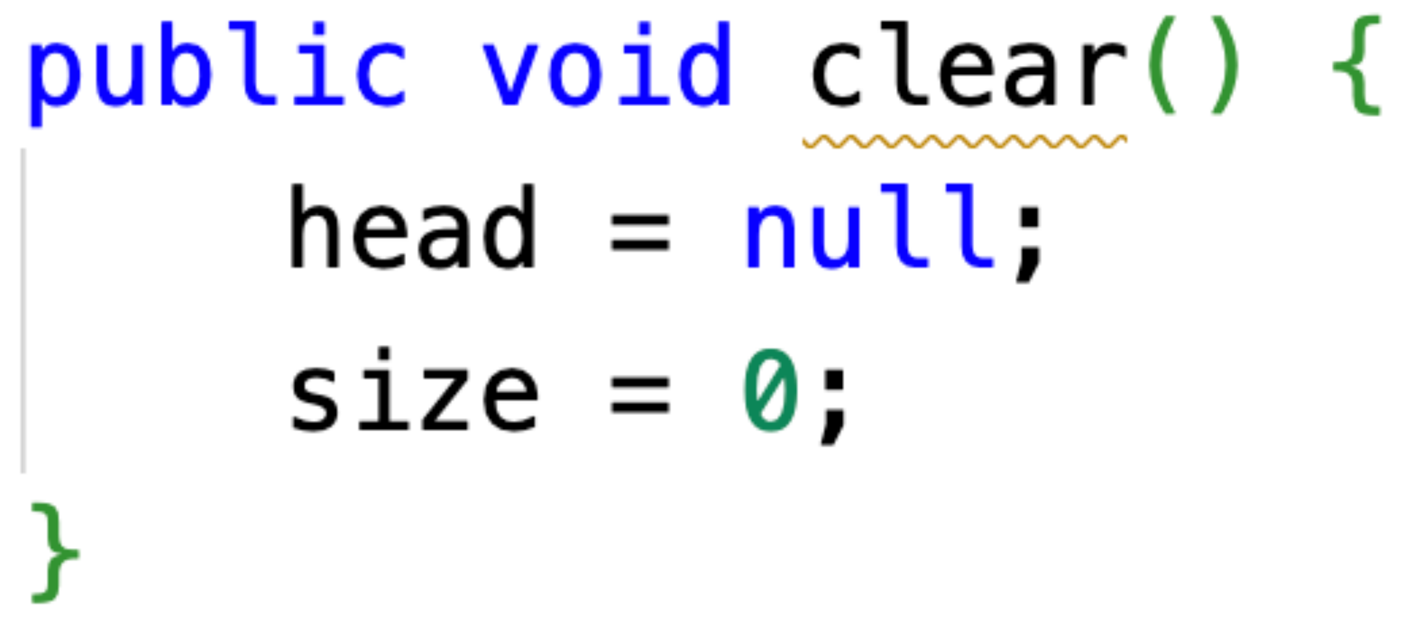
```java
public boolean isEmpty() {
    return size == 0;
};
```

Note that we haven't explicitly defined a construct for SLList yet. That's fine—the default constructor will initialize its instance variables (head, size) to be null and 0.

clear()

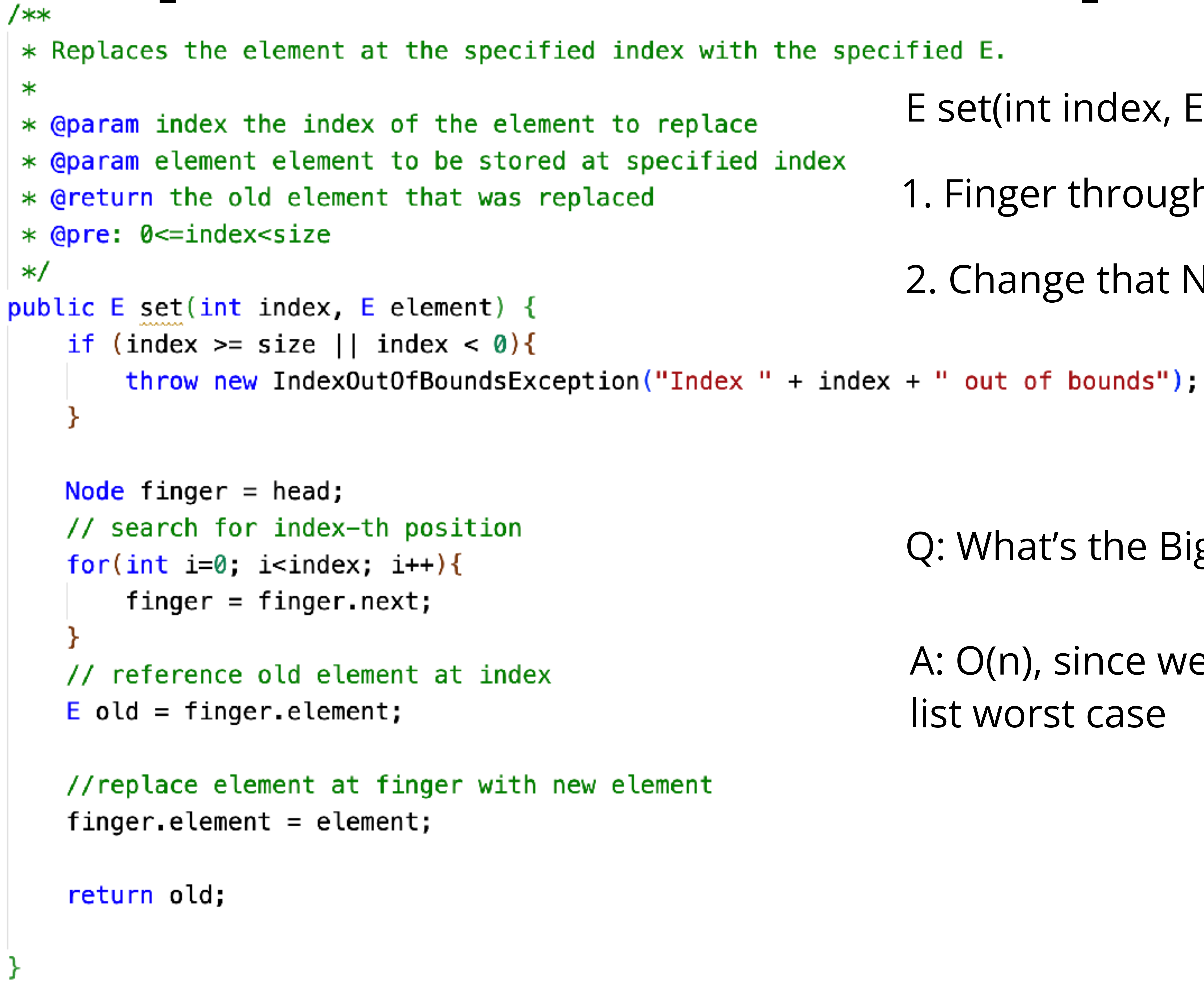
How can we shortcut clearing a SLL besides calling remove() many times?

Just reset our instance variables, head and size. Garbage collection will handle the rest.



```java
public void clear() {
    head = null;
    size = 0;
}
```

Replace element at a specified index



```java
/**
 * Replaces the element at the specified index with the specified E.
 *
 * @param index the index of the element to replace
 * @param element element to be stored at specified index
 * @return the old element that was replaced
 * @pre: 0<=index<size
 */
public E set(int index, E element) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }

    Node finger = head;
    // search for index-th position
    for(int i=0; i<index; i++){
        finger = finger.next;
    }
    // reference old element at index
    E old = finger.element;

    //replace element at finger with new element
    finger.element = element;

    return old;

}
```

E set(int index, E element)

1. Finger through list to get the Node at the requested index

2. Change that Node's element

Q: What's the Big O run time for set()?

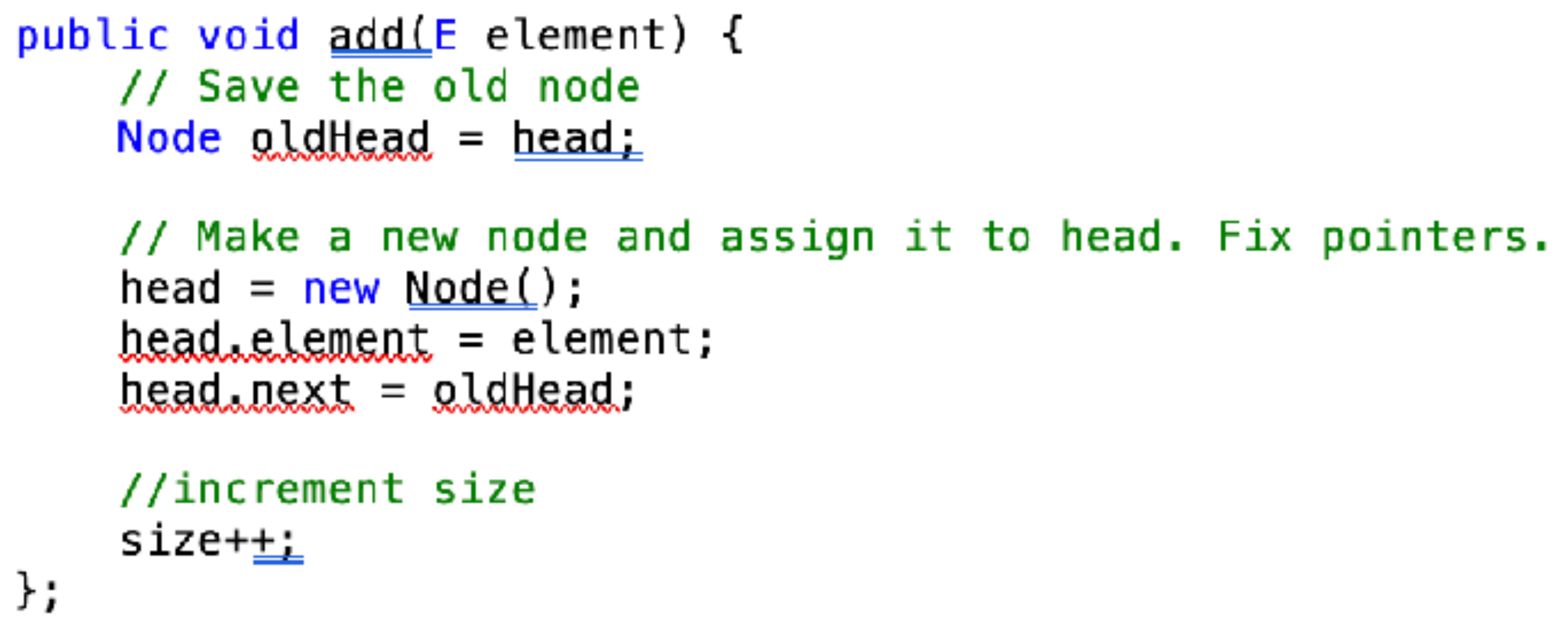
A: O(n), since we need to iterate through the whole list worst case

Worksheet time!

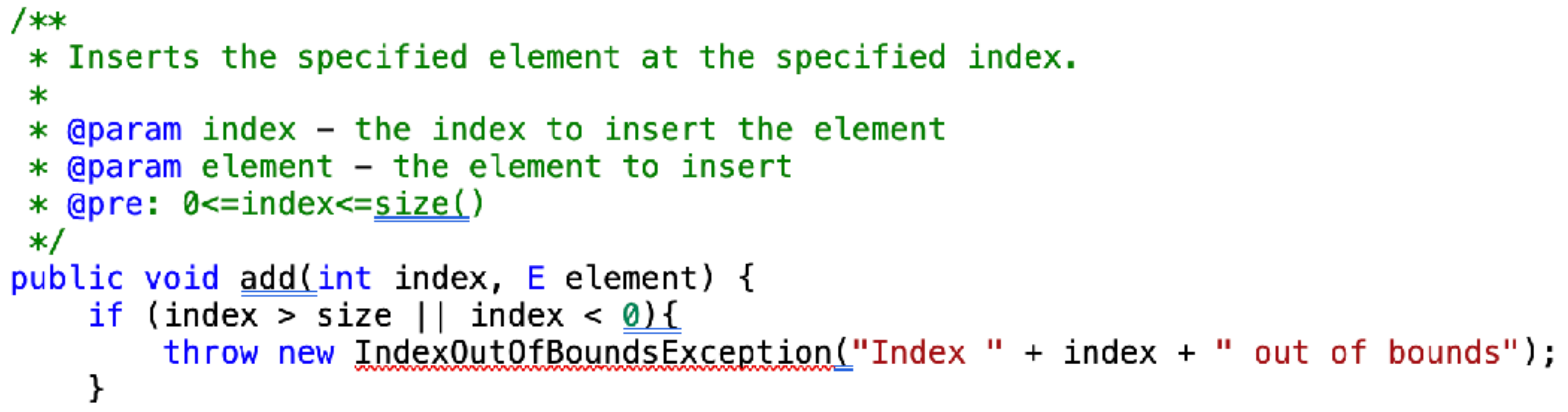
- We talked about inserting elements at the head of an SLL. What about at a specific index?

Hint: try to call the already implemented add()!

Hint: think about worksheet Q2!



```
public void add(E element) {
    // Save the old node
    Node oldHead = head;

    // Make a new node and assign it to head. Fix pointers.
    head = new Node();
    head.element = element;
    head.next = oldHead;

    //increment size
    size++;
};
```



```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index - the index to insert the element
 * @param element - the element to insert
 * @pre: 0<=index<=size()
 */
public void add(int index, E element) {
    if (index > size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
```

Worksheet answers

```
/**
 * Inserts the specified element at the specified index.
 *
 * @param index
 *            the index to insert the element
 * @param element
 *            the element to insert
 * @pre: 0<=index<=size()
 */
public void add(int index, E element) {
    if (index > size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    if (index == 0) {
        add(element); //can call already implemented add
    } else {
        Node previous = null;
        Node finger = head;
        // search for index-th position
        for(int i=0; i<index; i++){
            previous = finger;
            finger = finger.next;
        }
        // create new value to insert in correct position.
        Node current = new Node();
        current.next = finger;
        current.element = element;
        // make previous value point to new value.
        previous.next = current;

        size++;
    }

};
```

We are inserting between previous and finger

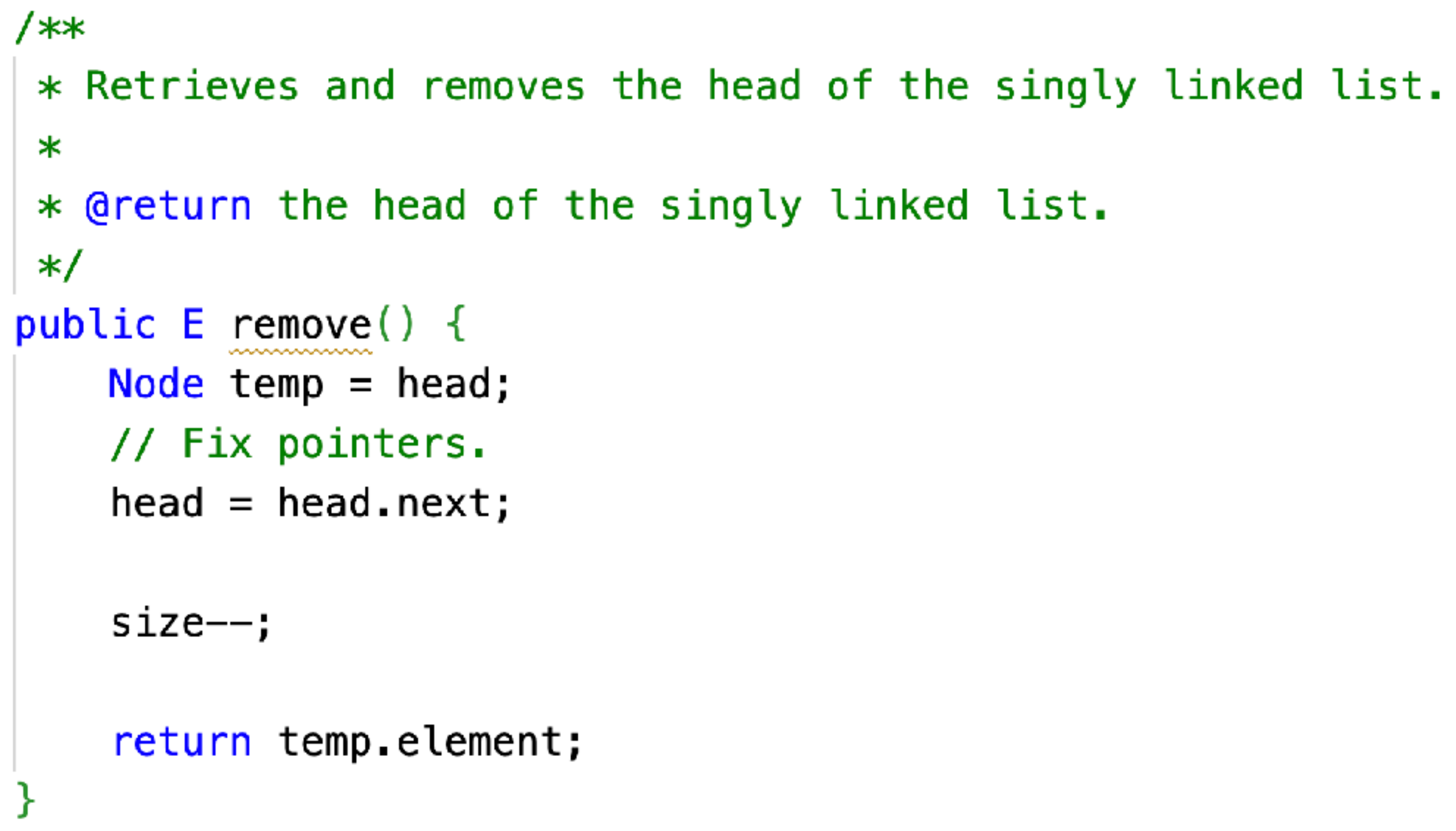
have to make sure .next links are replaced for both “previous” and “current”

Retrieve and remove head

E remove()

1. How do we remove the head? Think about your worksheet.

2. Decrement size

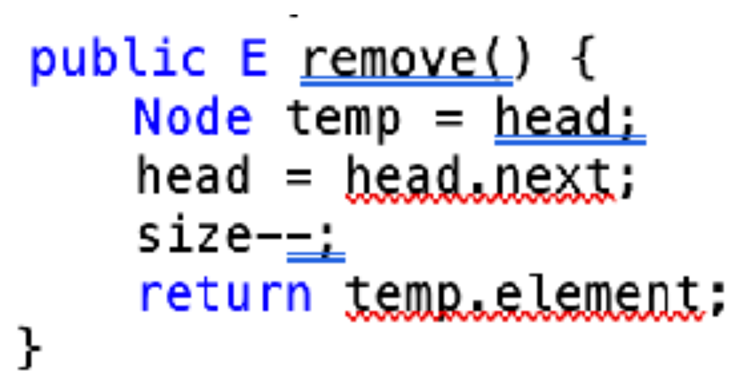


```java
/**
 * Retrieves and removes the head of the singly linked list.
 *
 * @return the head of the singly linked list.
 */
public E remove() {
    Node temp = head;
    // Fix pointers.
    head = head.next;

    size--;

    return temp.element;
}
```

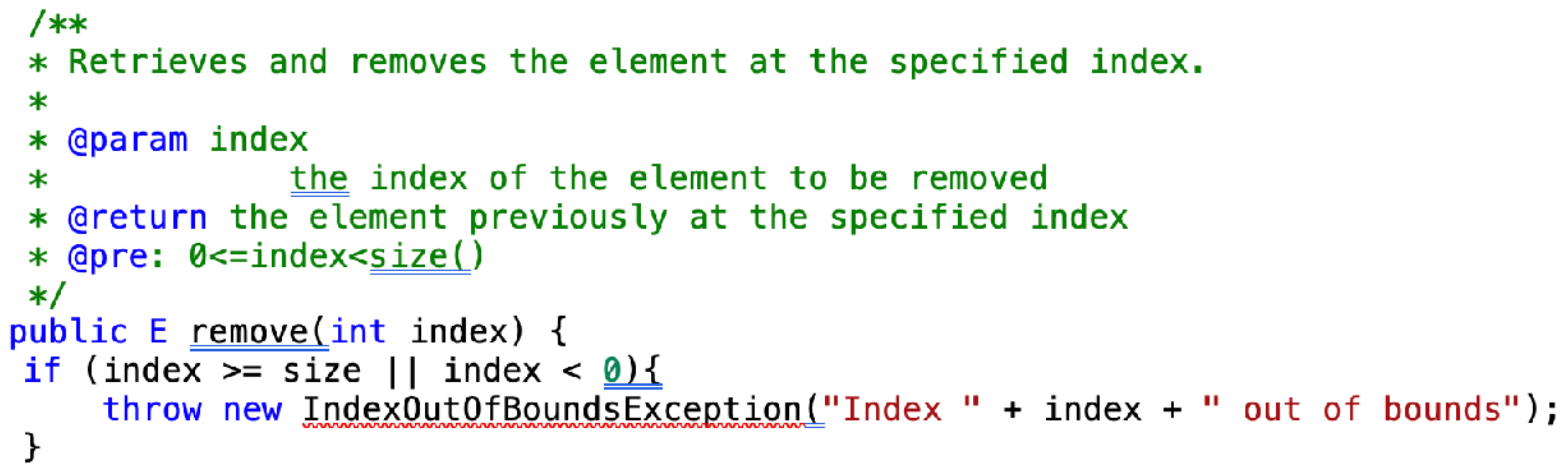
Worksheet time!

- We talked about removing elements at the head of an SLL. What about at a specific index?

Hint: combine add(int index) with remove()



```java
public E remove() {
    Node temp = head;
    head = head.next;
    size--;
    return temp.element;
}
```



```java
 /**
 * Retrieves and removes the element at the specified index.
 *
 * @param index
 *             the index of the element to be removed
 * @return the element previously at the specified index
 * @pre: 0<=index<size()
 */
public E remove(int index) {
 if (index >= size || index < 0){
     throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
 }
```

Worksheet answers

```java
/**
 * Retrieves and removes the element at the specified index.
 *
 * @param index
 *            the index of the element to be removed
 * @return the element previously at the specified index
 * @pre: 0<=index<size()
 */
public E remove(int index) {
    if (index >= size || index < 0){
        throw new IndexOutOfBoundsException("Index " + index + " out of bounds");
    }
    if (index == 0) {
        return remove();
    } else {
        Node previous = null;
        Node finger = head;
        // search for value indexed, keep track of previous
        for(int i=0; i<index; i++){
            previous = finger;
            finger = finger.next;
        }
        previous.next = finger.next;

        size--;
        // finger's value is old value, return it
        return finger.element;
    }
}
```

Lecture 6 wrap-up

- SLLs are linear data structures that are not continuous in memory since we use .next pointers to construct the list.
- Next time: Doubly Linked Lists, runtimes, and comparison to ArrayLists
- Lab this week: learn how to use the Java debugger
- Part I of Darwin due Tues 11:59pm
  - If you don't have a partner yet / haven't gotten started yet, let's resolve this now.
- Checkpoint I is in 2 weeks. If you have SDRC accommodations, please schedule them **now**. We cannot offer alternate proctoring in-class (e.g., if you have extra time, please get it proctored via the SDRC). 1 double sided handwritten cheat sheet OK.

Resources

- Linked lists from the textbook: https://algs4.cs.princeton.edu/13stacks/
- See slides following this for more practice problems

Bonus practice problem

- Add a deleteLast method in the SinglyLinkedList class that removes the last node of a singly linked list. Think of edge cases.

Bonus answer

```
public void deleteLast() {
       if (!isEmpty()) {
           if (size == 1) {
               head = null;
           } else {
               Node current = head;
               for (int i = 0; i < size - 2; i++) {
                   current = current.next;
               }
               current.next = null;
           }
           size--;
       }
   else{//throw some appropriate exception}
   }
```