UNIVERSITY of WISCONSIN
LA CROSSE
COMPUTER SCIENCE

Week 07:
Linked Lists

CS 220: Software Design II — D. Mathias

The `Collection` and `List` Interfaces

```
Collection
{interface}

+ add(E e) : boolean
+ clear()
+ contains(Object o) : boolean
+ equals(Object o) : boolean
+ isEmpty() : boolean
+ iterator() : Iterator<E>
+ remove(Object o) : boolean
+ size() : int
...
```

```
List
{interface}

+ add(int index, E e) : boolean
+ indexOf(Object o) : int
...
```

`Collection` describes a group of objects

`List` holds data in a linear fashion

Together, we can ask questions like...

- what is the last index of a particular value?
- is the list empty?
- how many values are in there?

At least two different ways to implement

- array
- linked nodes

A Basic Guide to Data Structures

Common data structures are defined by two components

- the *interface* that describes how they behave
- the *implementation* that describes how they store data

implementation

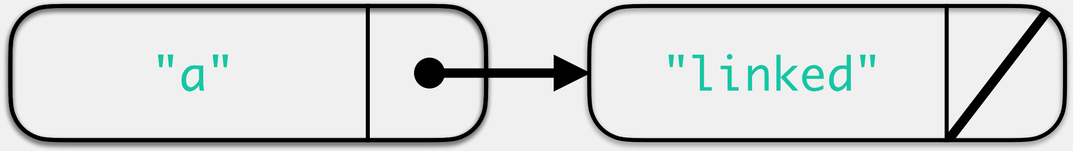
interface	resizable array	linked list	hash table	balanced tree
set			`HashSet`	`TreeSet`
list	`ArrayList`	`LinkedList`		
map			`HashMap`	`TreeMap`

LinkedList

- Class that implements list methods by storing values in a set of linked nodes
- Abstracts away many of the actions we would need to do manually
  - creating new nodes to hold more data
  - shifting links around for nodes

LinkedList

- Class that implements list methods by storing values in a set of linked nodes
- Abstracts away many of the actions we would need to do manually
  - creating new nodes to hold more data
  - shifting links around for nodes

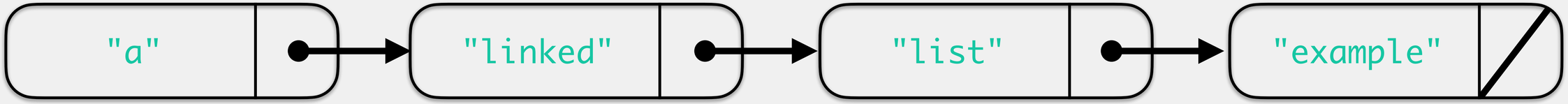


LinkedList

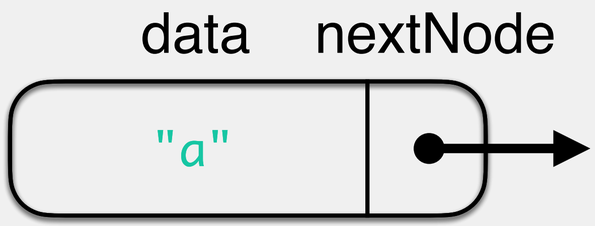
- Class that implements list methods by storing values in a set of linked nodes
- Abstracts away many of the actions we would need to do manually
  - creating new nodes to hold more data
  - shifting links around for nodes

LinkedList

- Class that implements list methods by storing values in a set of linked nodes
- Abstracts away many of the actions we would need to do manually
  - creating new nodes to hold more data
  - shifting links around for nodes



The Humble Node




```
public class ListNode<E>{

    private E value;
    private ListNode nextNode;

    public ListNode(E e) { value = e; }

    public void setNext(ListNode next) {
      nextNode = next;
    }

    public ListNode getNext() {
       return nextNode;
    }

    public E getValue() { return value; }
}
```

Recursive data structure

i.e., at least some of the attributes of the class are the class type itself

Node will store…

data

a reference to the next node

this is how we do linking!

The First, Last, and Intermediate Nodes

```
// reference to the first node
ListNode<String> firstNode = new ListNode<>("a");

//  assume we have linked
//  some nodes to firstNode

//  this while loop iterates through nodes

ListNode<String> node = firstNode.getNext();
while(node.getNext() != null) {
    System.out.println(node.getValue());
    node = node.getNext();
}

// what is the value of node when the loop ends?
```

We hold a reference to the first node

Need an intermediate node? Start at beginning, traverse each node

When this loop ends, where are we in the list?

We know we're at the last node because its next node value is `null`

The First, Last, and Intermediate Nodes

```
// reference to the first node
ListNode<String> firstNode = new ListNode<>("a");

//  assume we have linked
//  some nodes to firstNode

//  this while loop iterates through nodes

ListNode<String> node = firstNode.getNext();
while(node != null) {
    System.out.println(node.getValue());
    node = node.getNext();
}

// what is the value of node when the loop ends?
```

We hold a reference to the first node

Need an intermediate node? Start at beginning, traverse each node

When this loop ends, where are we in the list?

node is null so we are no longer in the list.

LinkedList: Attributes

```
LinkedList<E>
- firstNode : ListNode<E>
- size : int
...
```

`firstNode`: reference to the first node in the list

- will be null when the list is empty

`size`: current amount of data

- starts at 0

This is all we need!

LinkedList: Methods

```
LinkedList<E>
...
+ add(E e) : boolean
+ add(int index, E e) : void
+ addFirst(E e) : void
+ addLast(E e) : void
+ contains(Object o) : boolean
+ getFirst() : E
+ getLast() : E
+ getNext() : E
+ indexOf(Object o) : int
+ iterator() : Iterator<E>
+ lastIndexOf(Object o) : int
+ remove(int index) : E
+ remove(Object o) : boolean
+ size() : int
...
```

Some methods required by `Collection`, some required by `List`

Some unique to `LinkedList`

- e.g., `getFirst()` allows us to get the first item in the `LinkedList`

Inner Classes

- Can nest one class inside another class
- Why would we want to do this?
  - need a class that will only be used by the class it is contained in
    - can only instantiate objects inside the outer class
  - similar to a private method only being used by the class it is contained in
- Linked list nodes are a great candidate for this
  - only want to be able to use them as part of the linked list class
  - programmer should **not** be responsible for directly creating/manipulating them

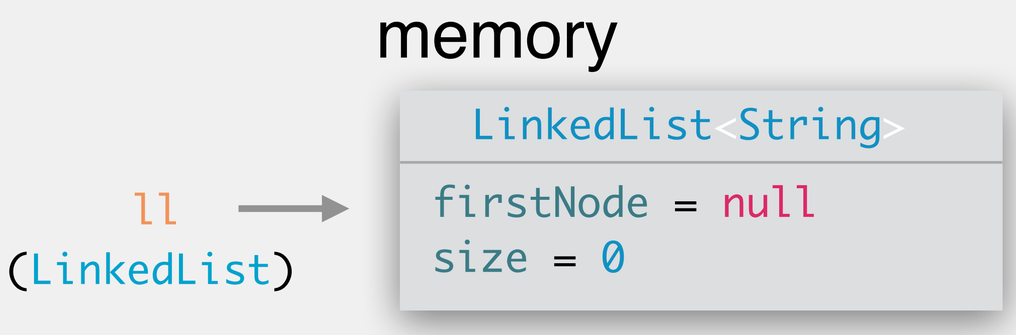
LinkedList: Constructor

arguments: nothing

returns: a new `LinkedList` object

```
new LinkedList<T>();
```

```
>LinkedList<String> ll = new LinkedList<>();
>
```



LinkedList: Add Methods

Four primary methods:

- `add(E e) : boolean`
  - works like ArrayList
  - will always return true for LinkedList
- `add(int index, E e) : void`
  - works like ArrayList
- `addFirst(E e) : void`
- `addLast(E e): void`

LinkedList: Add Methods

arguments: element to add

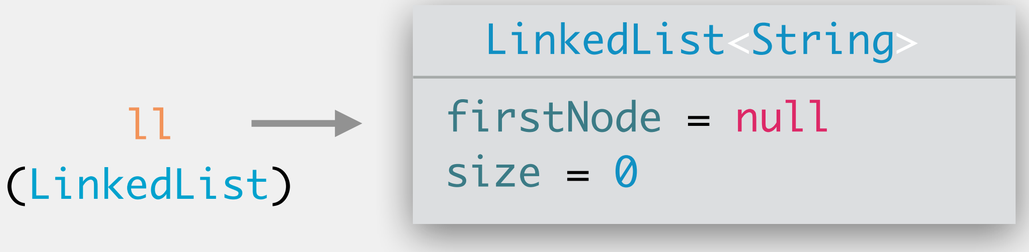
returns: boolean indicating whether the list has changed

will always return true for LinkedList

behavior: adds value to end of LinkedList

```
>LinkedList<String> ll = new LinkedList<>();
>ll.add("yes");
 ll.add("no");
 ll.add("maybe");
```

memory



LinkedList: Add Methods

arguments: element to add

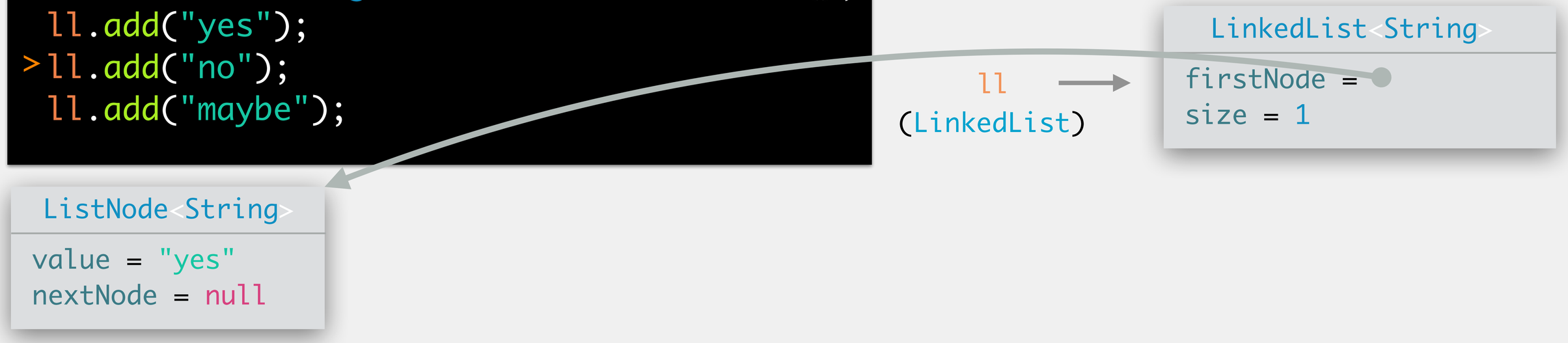
returns: boolean indicating whether the list has changed

will always return true for LinkedList

behavior: adds value to end of LinkedList

```
 LinkedList<String> ll = new LinkedList<>();
 ll.add("yes");
>ll.add("no");
 ll.add("maybe");
```

memory



LinkedList: Add Methods

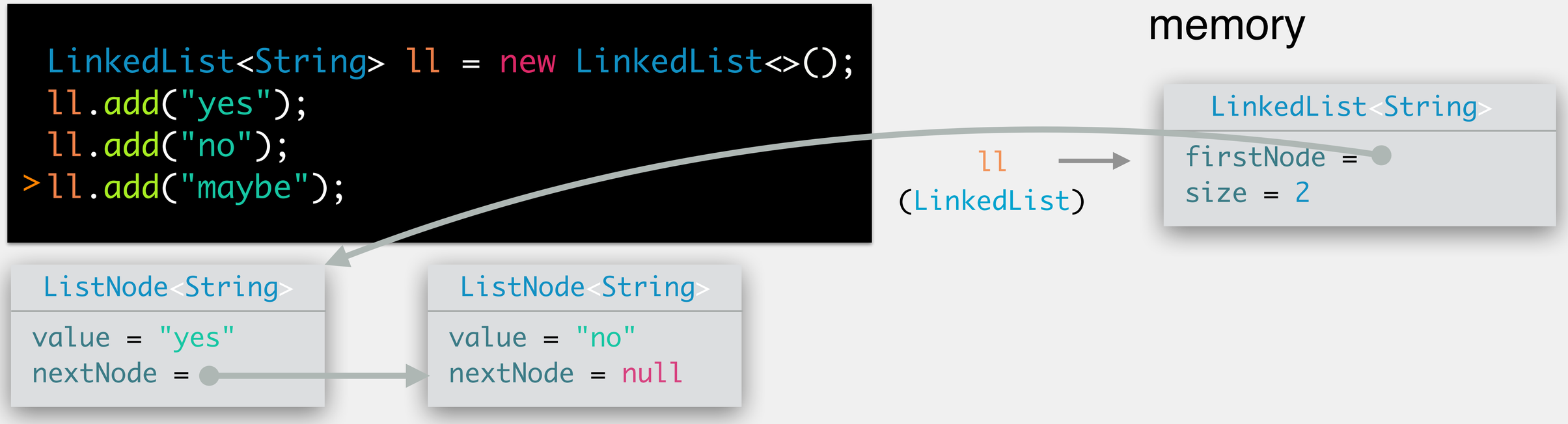
arguments: element to add

returns: boolean indicating whether the list has changed

will always return true for LinkedList

behavior: adds value to end of LinkedList

```
 LinkedList<String> ll = new LinkedList<>();
 ll.add("yes");
 ll.add("no");
>ll.add("maybe");
```



LinkedList: Add Methods

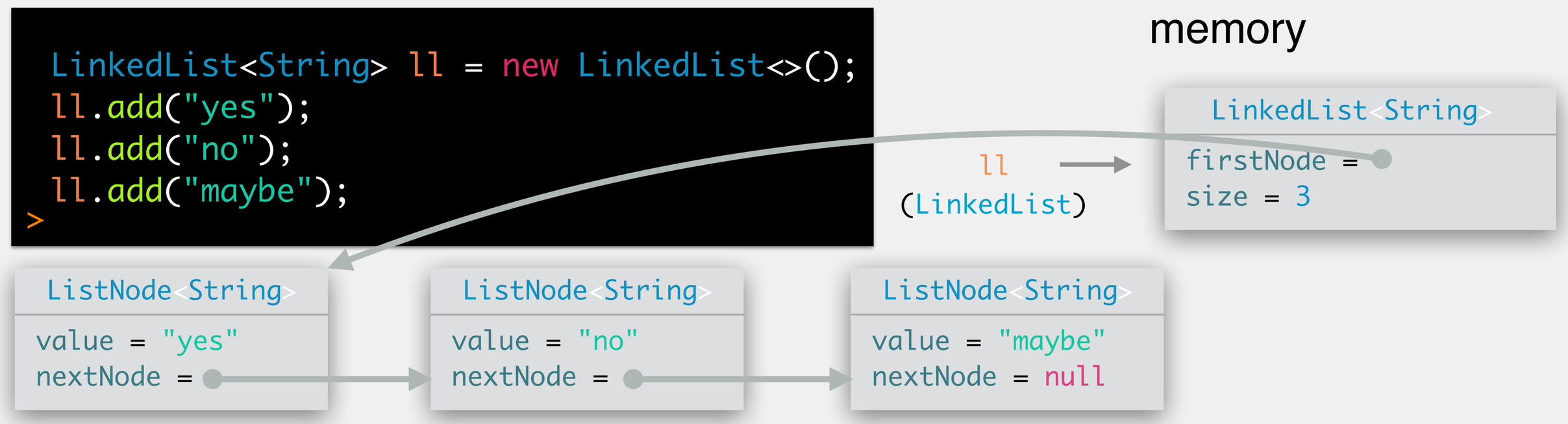
arguments: element to add

returns: boolean indicating whether the list has changed

will always return true for LinkedList

behavior: adds value to end of LinkedList

```
LinkedList<String> ll = new LinkedList<>();
ll.add("yes");
ll.add("no");
ll.add("maybe");
>
```



LinkedList: Add Methods

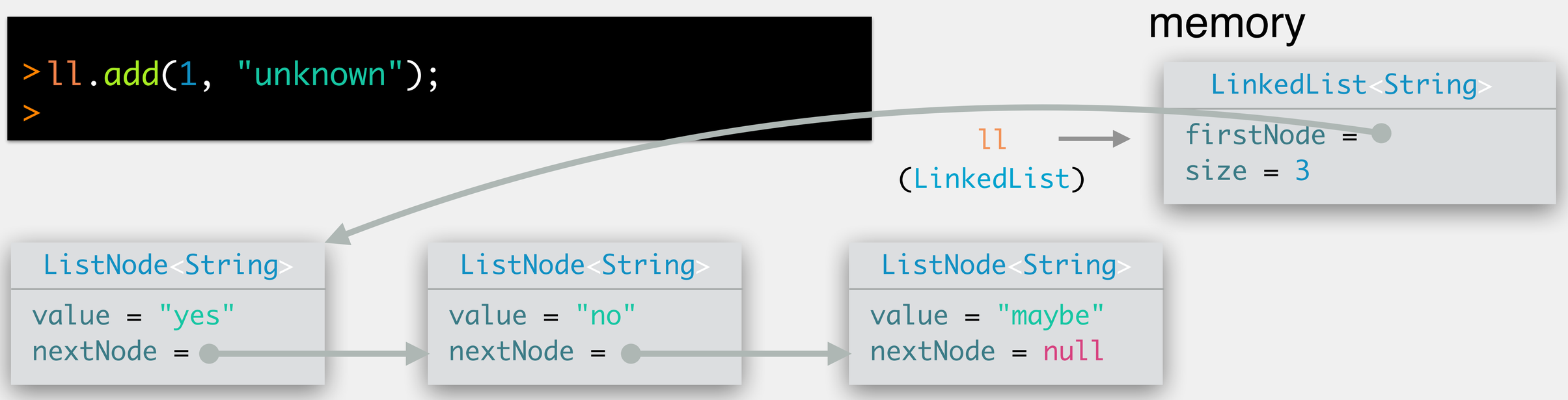
arguments: index to add at, element to add

returns: void

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to the index of LinkedList

```
>ll.add(1, "unknown");
>
```



LinkedList: Add Methods

N.B.: This next part is tricky! Need to make sure you don't lose a link to the node currently at index 1

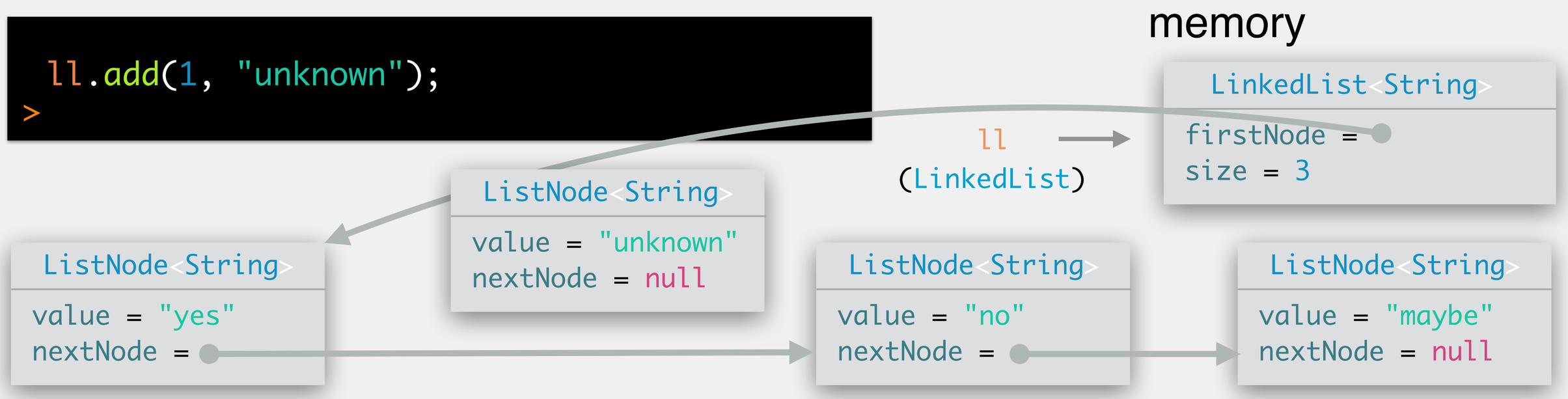
arguments: index to add at, element to add

returns: void

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to the index of LinkedList

```
 ll.add(1, "unknown");
>
```



LinkedList: Add Methods

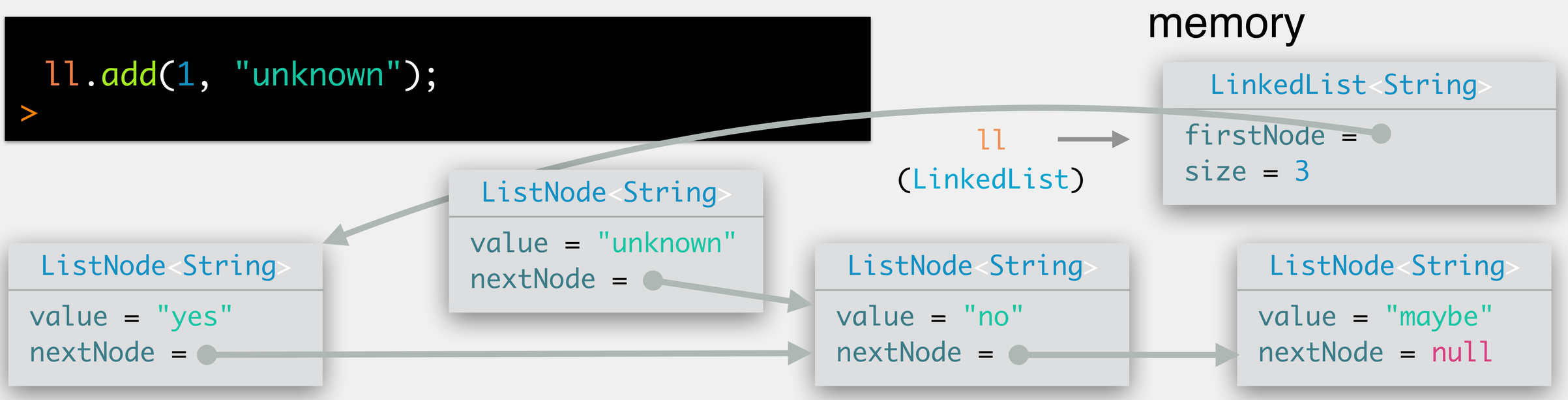
arguments: index to add at, element to add

returns: void

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to the index of LinkedList

```
 ll.add(1, "unknown");
>
```



LinkedList: Add Methods

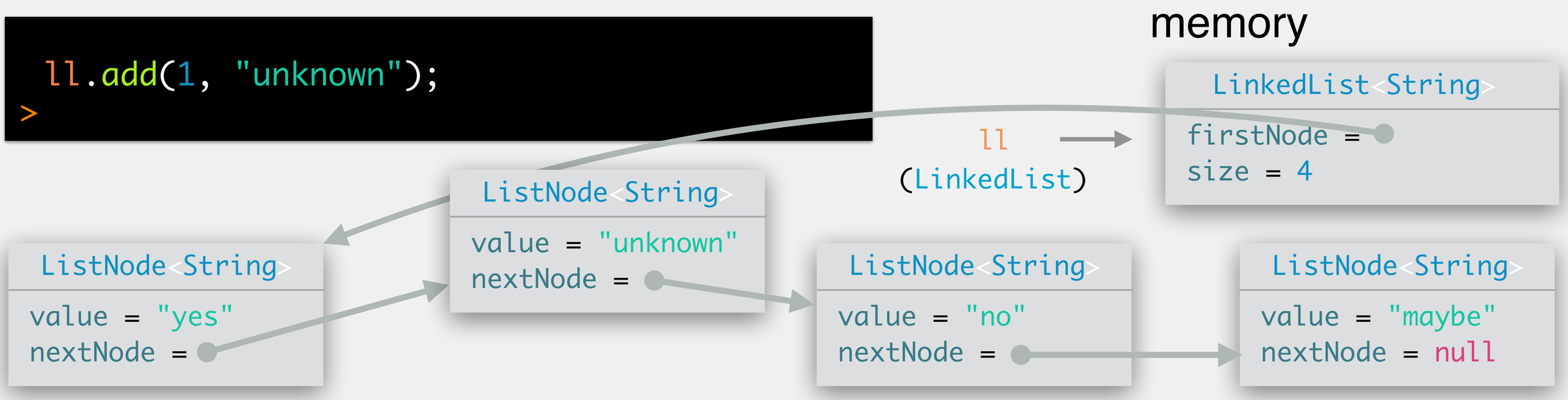
arguments: index to add at, element to add

returns: void

throws: `IndexOutOfBoundsException` if index is negative or greater than size

behavior: adds value to the index of LinkedList

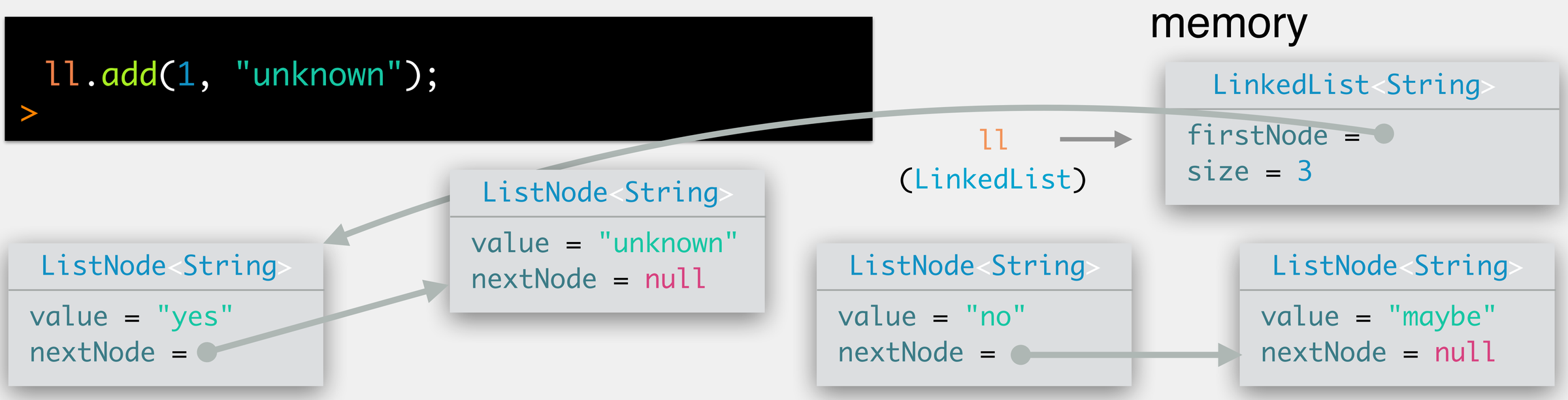
```
 ll.add(1, "unknown");
>
```



LinkedList: Add Methods (the wrong way)

DO NOT DO IT THIS WAY!

```
ll.add(1, "unknown");
>
```

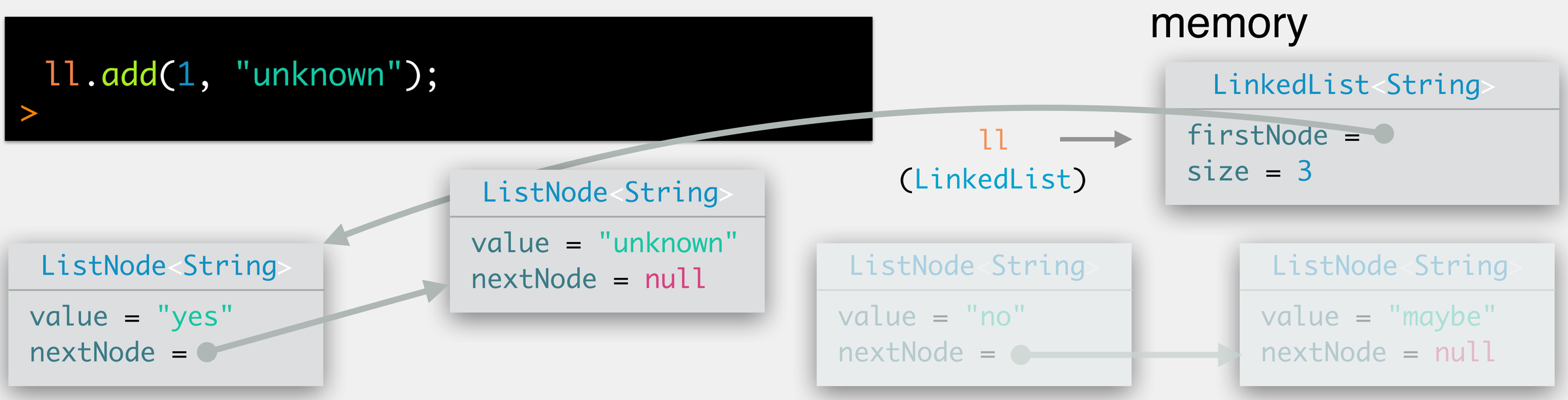


LinkedList: Add Methods (the wrong way)

ORDER MATTERS!

By first making the previous node point to the new node we lose th e remainder of the list.

```
 ll.add(1, "unknown");
>
```



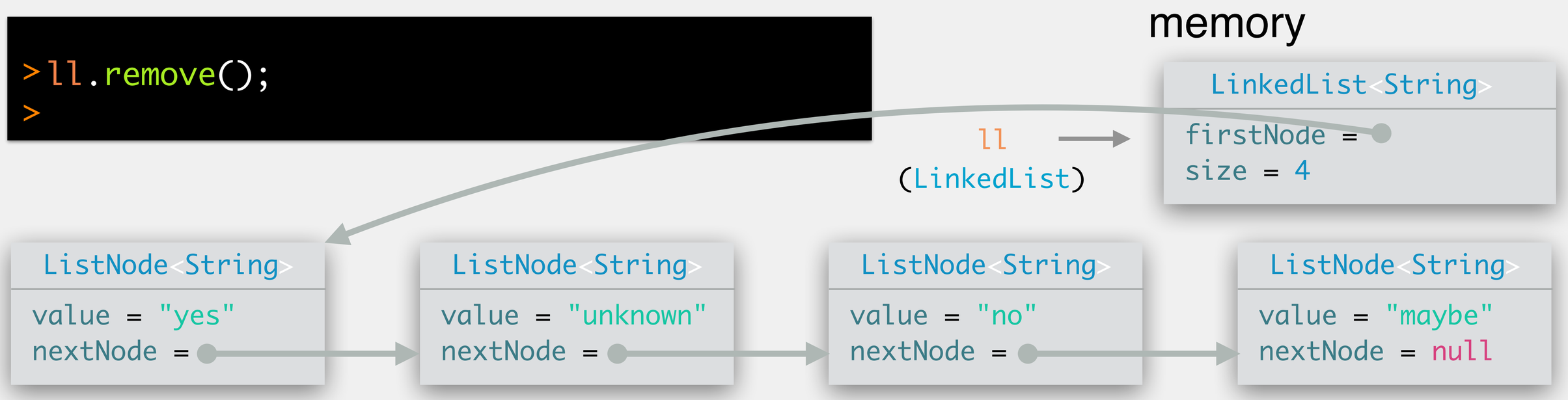
LinkedList: Remove Methods

arguments: none

returns: E, value held at index 0

behavior: removes node at beginning of LinkedList and returns value

```
>ll.remove();
>
```



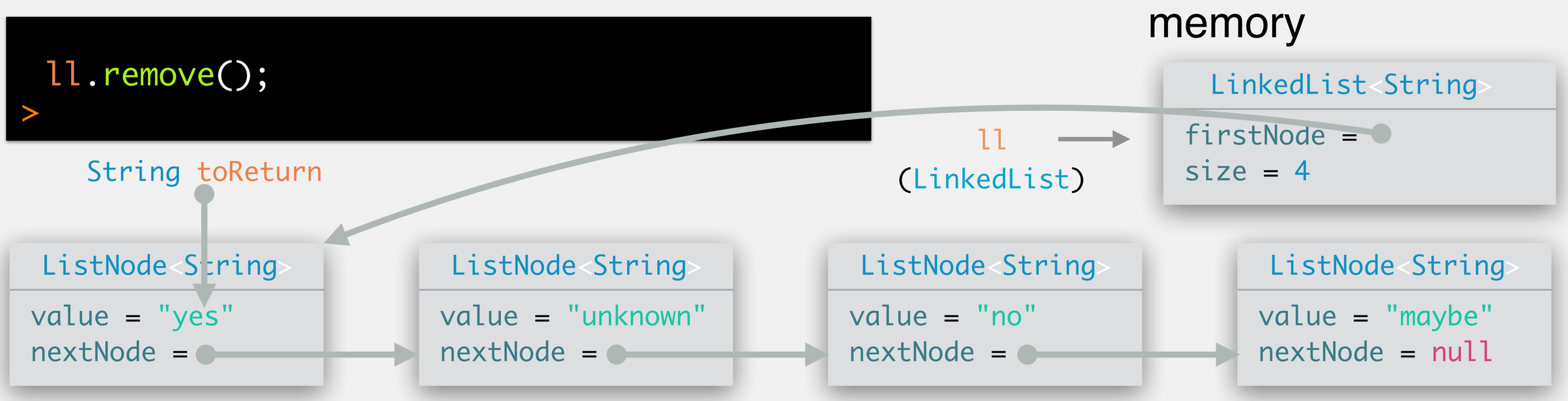
LinkedList: Remove Methods

arguments: none

returns: E, value held at index 0

behavior: removes node at beginning of LinkedList and returns value

```
 ll.remove();
>
```



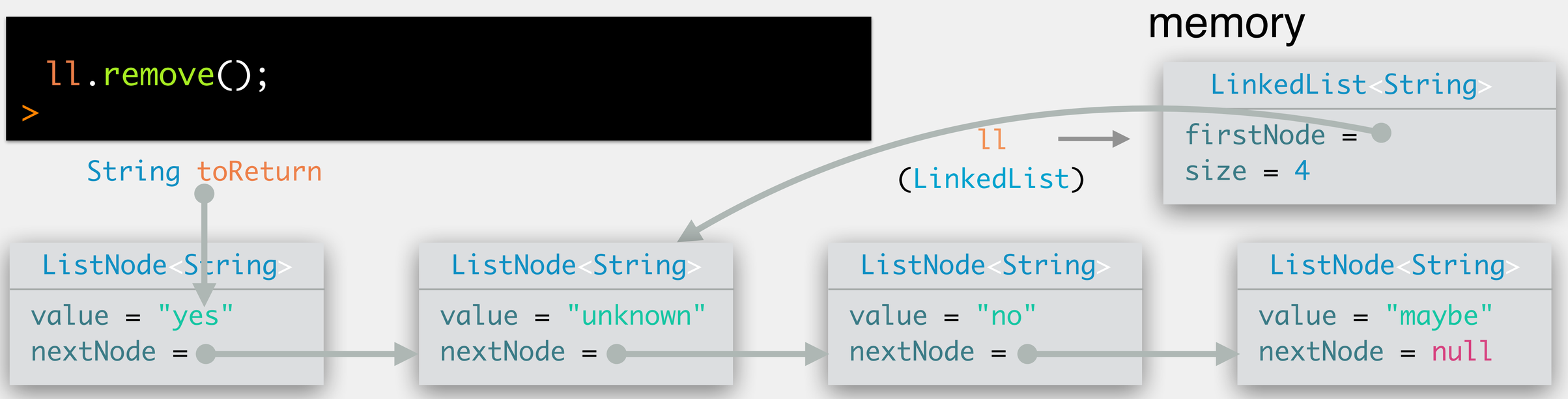
LinkedList: Remove Methods

arguments: none

returns: E, value held at index 0

behavior: removes node at beginning of LinkedList and returns value

```
ll.remove();
>
```



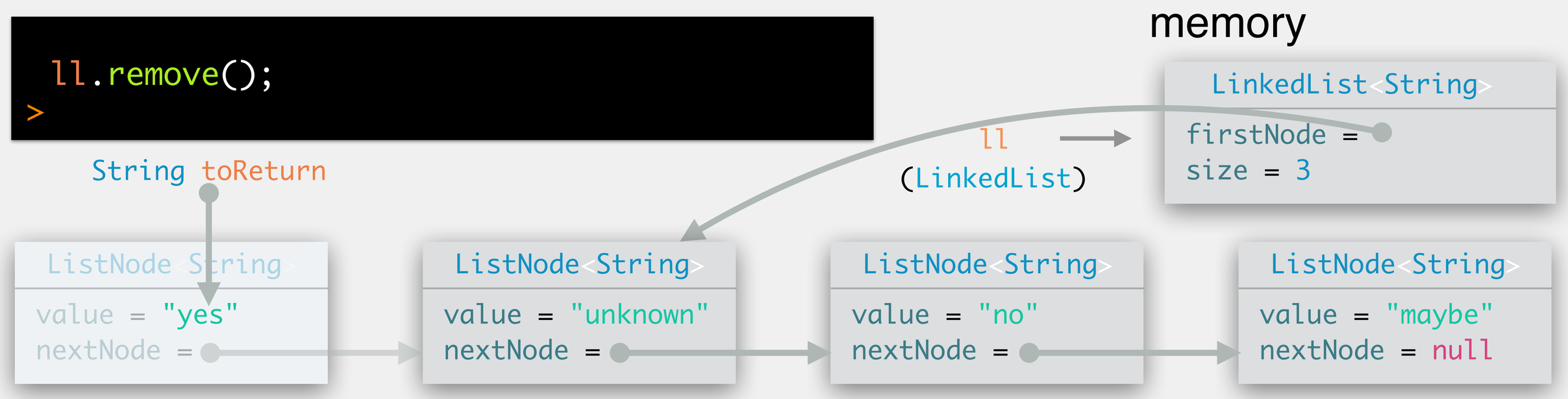
LinkedList: Remove Methods

arguments: none

returns: E, value held at index 0

behavior: removes node at beginning of LinkedList and returns value

```
ll.remove();
>
```



LinkedList: Remove Methods

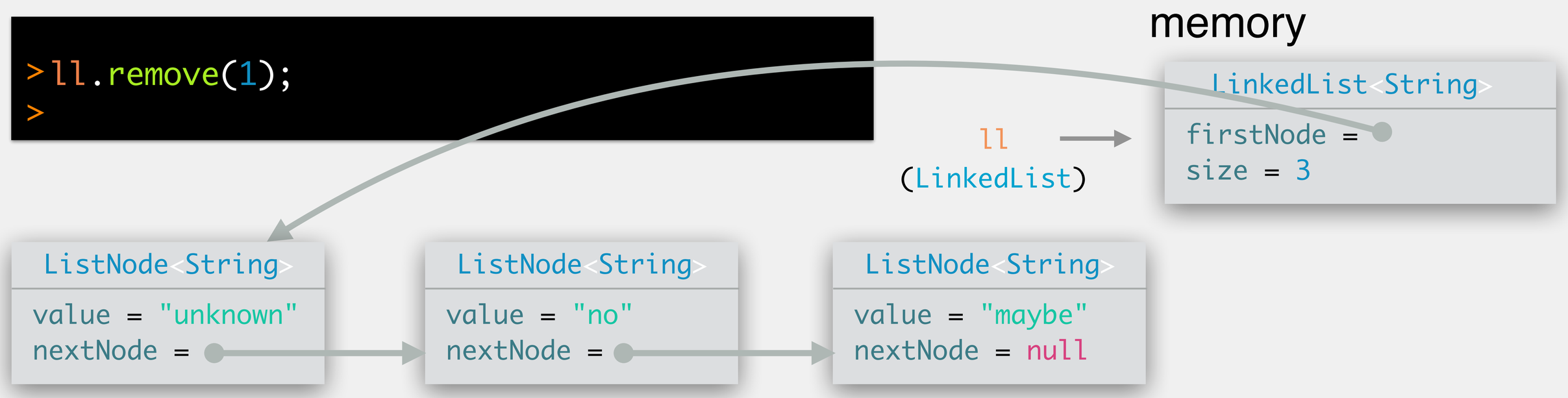
arguments: index (`int`) position to remove

returns: `E` of the value removed

throws: `IndexOutOfBoundsException` if index < 0 ll index >= size

behavior: removes node at index and returns value

```
>ll.remove(1);
>
```



LinkedList: Remove Methods

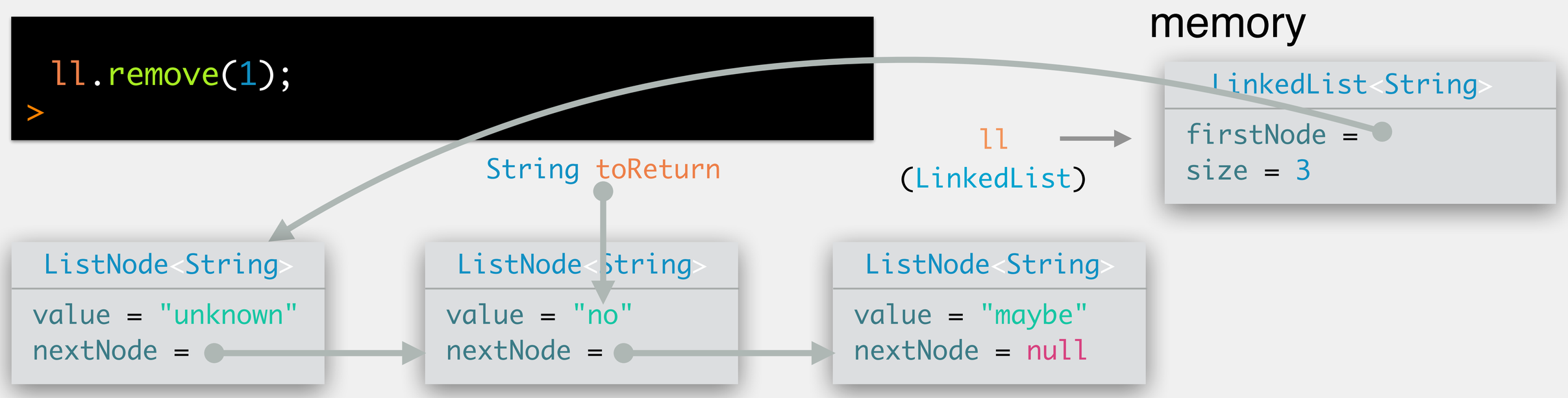
arguments: index (`int`) position to remove

returns: `E` of the value removed

throws: `IndexOutOfBoundsException` if index < 0 ll index >= size

behavior: removes node at index and returns value

```
 ll.remove(1);
>
```



LinkedList: Remove Methods

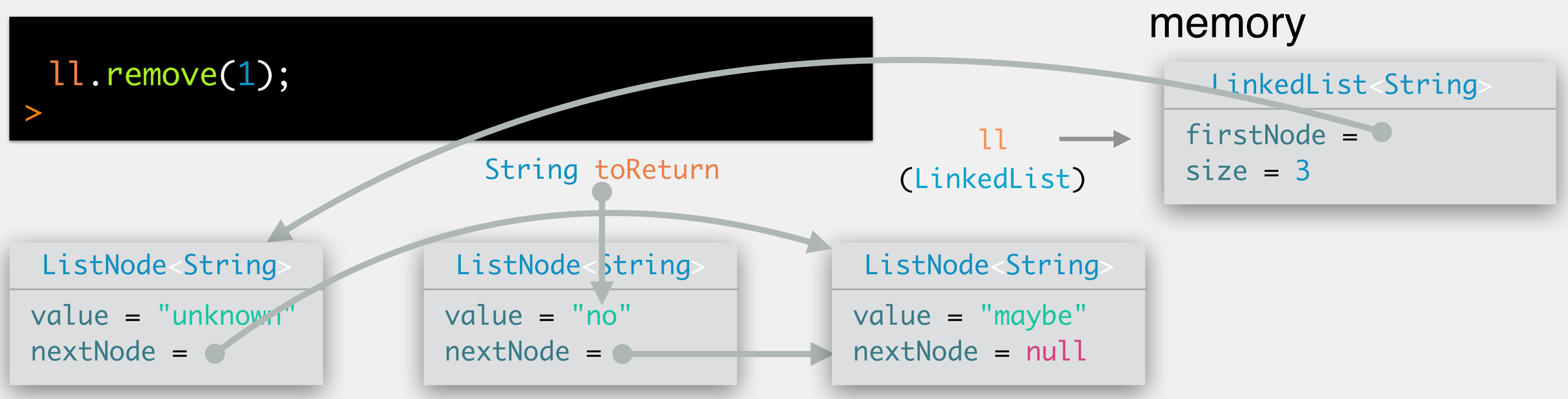
arguments: index (`int`) position to remove

returns: `E` of the value removed

throws: `IndexOutOfBoundsException` if index < 0 ll index >= size

behavior: removes node at index and returns value

```
 ll.remove(1);
>
```



LinkedList: Remove Methods

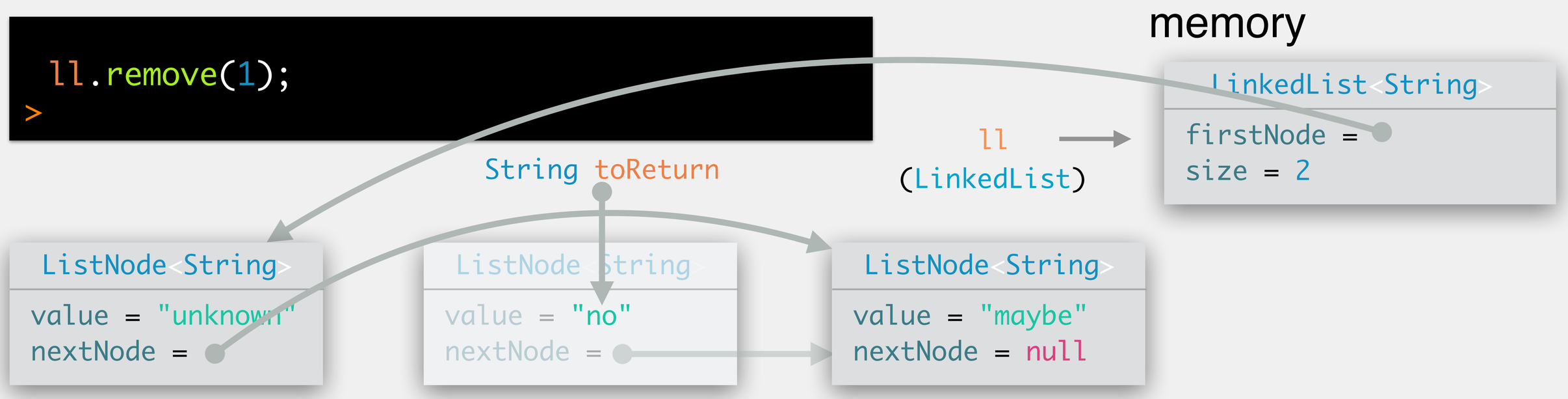
arguments: index (`int`) position to remove

returns: `E` of the value removed

throws: `IndexOutOfBoundsException` if index < 0 ll index >= size

behavior: removes node at index and returns value

```
 ll.remove(1);
>
```



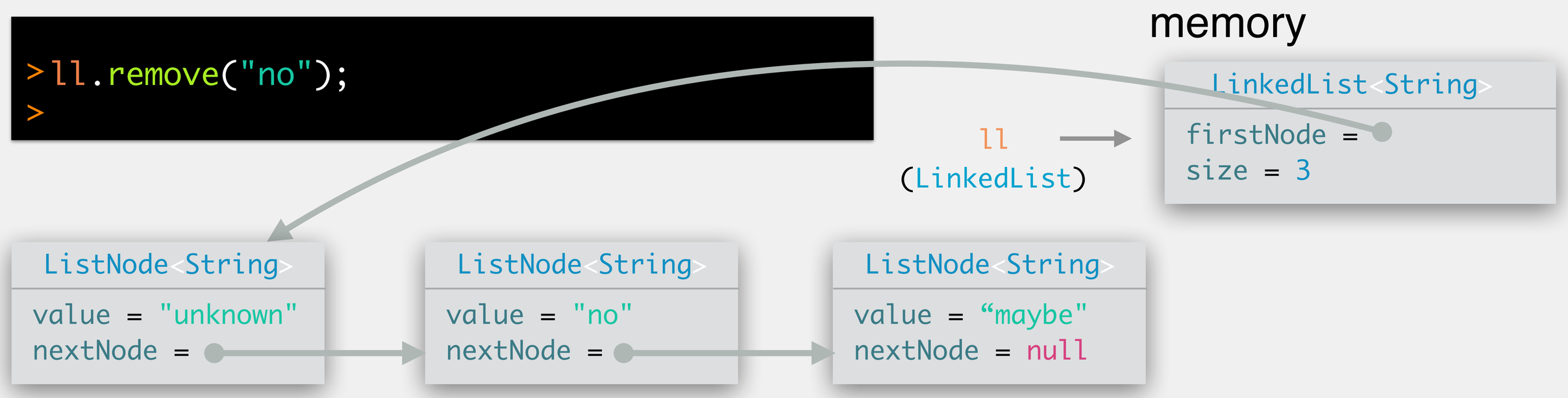
LinkedList: Remove Methods

arguments: value (`E`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value

```
>ll.remove("no");
>
```



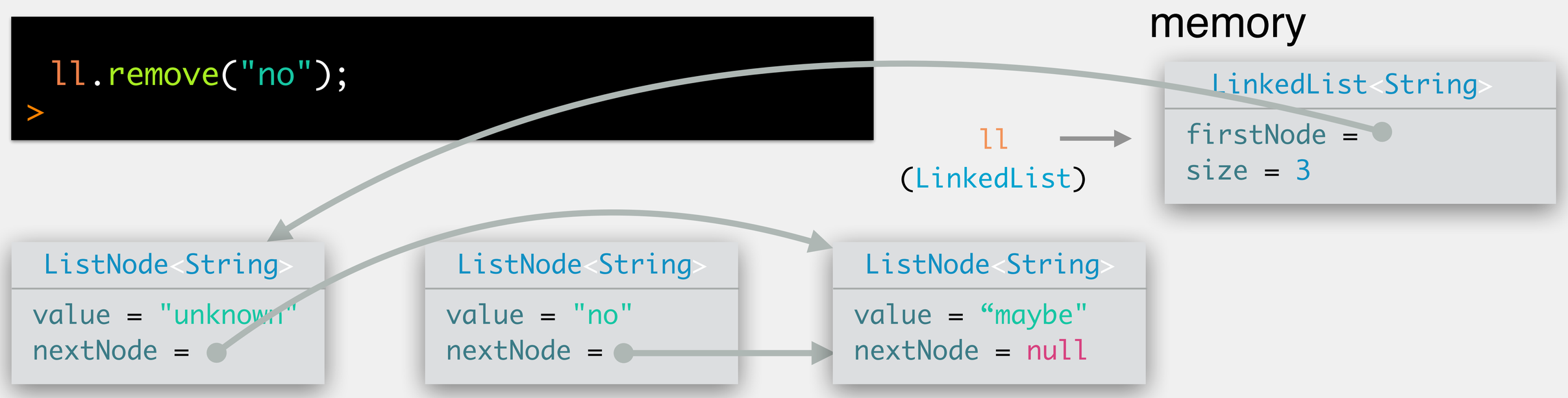
LinkedList: Remove Methods

arguments: value (`E`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value

```
ll.remove("no");
>
```



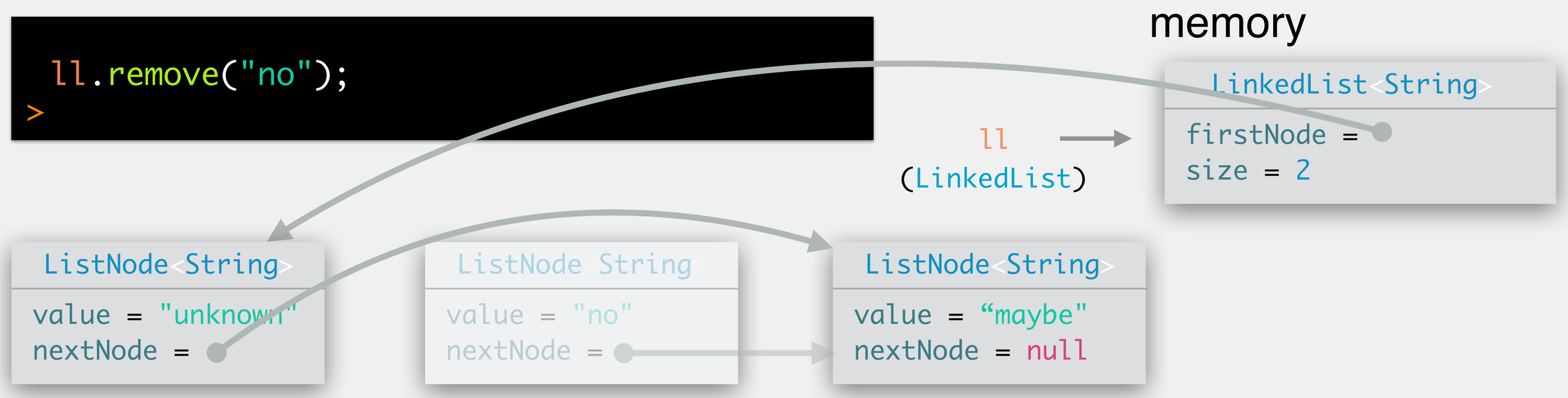
LinkedList: Remove Methods

arguments: value (`E`) to remove; removes first occurrence

returns: `boolean` indicating whether or not the list changed

behavior: removes first occurrence of value

```
ll.remove("no");
>
```



Exercise: Linked Lists

Write the following methods; do not allocate new nodes:

`public void swap(int index1, int index2)`

swaps those two nodes at these indices in the list, leaving the rest of the list unchanged

`public void concatenate(LinkedList ll)`

add the linked list argument to the end of this linked list

`public void shuffle(LinkedList ll)`

add the nodes of the linked list argument between the nodes of this linked list

e.g., if this linked list is `n11`, `n12`, `n13` and the argument is `n21`, `n22`, `n23`, this method should produce `n11`, `n21`, `n12`, `n22`, `n13`, `n23`; excess nodes from one list should be added to the end

`public void reverse()`

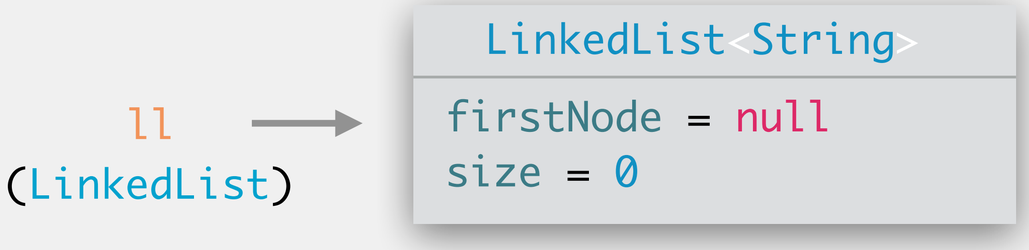
reverse the linked list, such that `head` points to the end

Sentinel Nodes

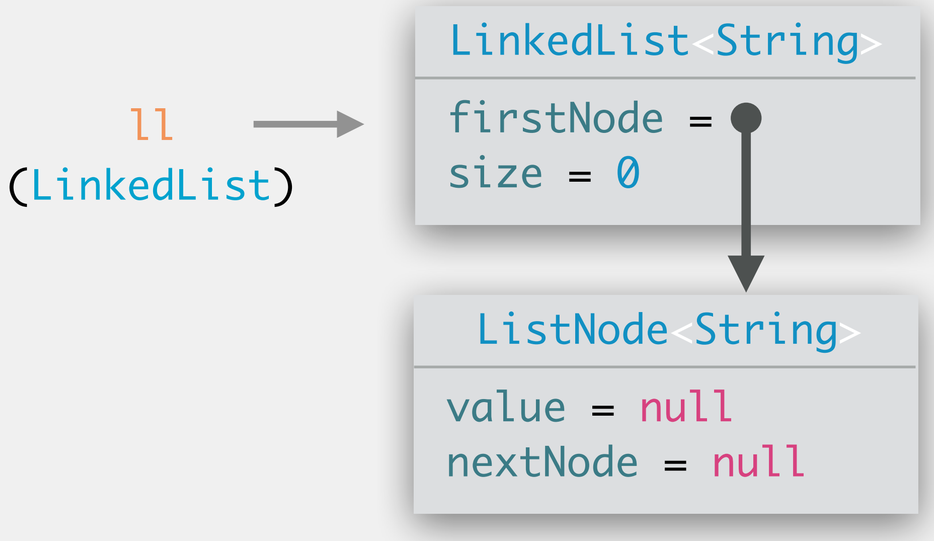
sentinel nodes are dummy nodes that hold a `null` value and indicate the end of the list

Produces cleaner code

without sentinel nodes
(what we've been doing)



with sentinel nodes
(another approach)



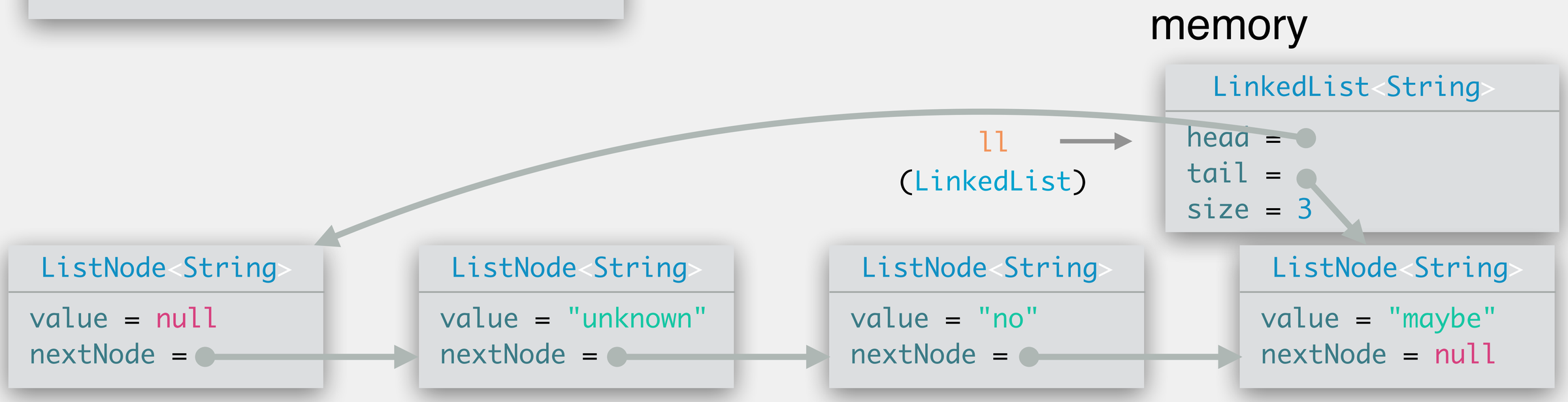
Head and Tail Nodes

```
LinkedList<E>
- head : ListNode<E>
- tail : ListNode<E>
- size : int
...
```

Often want to jump to the end of the list

Can add an additional attribute to point to the last node in the list

Typically called the *head* (first node) and *tail* (last node)



Exercise: LinkedList Methods

Write code for the following `LinkedList` methods:

- `public int size()`

  returns the size of the linked list

- `public boolean isEmpty()`

  returns `true` if the list is empty, `false` if not

- `public void clear()`

  deletes all the nodes and sets `firstNode` to `null`; `isEmpty()` should be `true` after executing this method

- `public String get(int index)`

  returns the value at the specified index; throws an `IndexOutOfBoundsException` if the index is outside the appropriate bounds

Exercise: LinkedList Methods

Write code for the following `LinkedList` methods:

`public E set(int index, E e)`

replaces the value at `index` with `e`; returns the string originally stored at `index`

`public boolean contains(E e)`

returns `true` if `e` is contained in the list, `false` if not

`public int indexOf(E e)`

returns the index of the first occurrence of `e`; returns -1 if the element does not exist in the list

`public int lastIndexOf(E e)`

returns the index of the last occurrence of `e`; returns -1 if the element does not exist in the list

Exercise: LinkedList Methods

Write code for the following `LinkedList` methods:

- `public E getFirst()`

  gets the first value in the list

- `public E getLast()`

  gets the last value in the list

- `public void addFirst(E e)`

  adds the element to the beginning of the list

- `public void addLast(E e)`

  adds the element to the end of the list