University of Wisconsin
LA CROSSE
Computer Science

Week 10:
Doubly Linked Lists

CS 220: Software Design II — D. Mathias

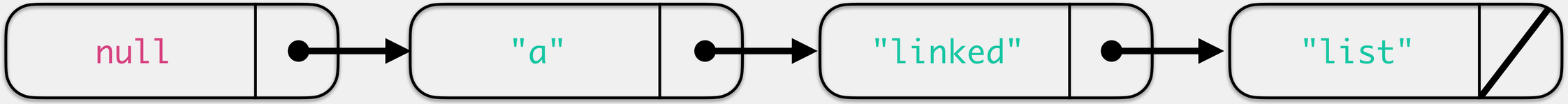
Singly Linked Lists

Allow us to link to the next node in the list

Consider the following methods; which are guaranteed to touch every node?

```
indexOf(E e)
lastIndexOf(E e)
getFirst()
getLast()
```

And, can we even iterate in reverse? (without needing to actually reverse)



Singly Linked Lists

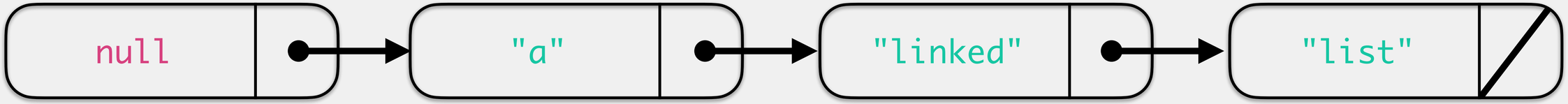
Allow us to link to the next node in the list

Consider the following methods; which are guaranteed to touch every node?

```
indexOf(E e)
lastIndexOf(E e)
getFirst()
getLast()
```

Why must these touch every node but the others don't?

What could we do to change that?



Moving back in a Singly Linked List

```
public ListNode<E> moveBack(ListNode curr) {
    if(curr != firstNode)
    {
        ListNode pred = firstNode;
        while(pred.nextNode != curr) {
            pred = pred.nextNode;
        }
        return pred;
    }
    return null;
}
```

It’s possible but not very efficient.

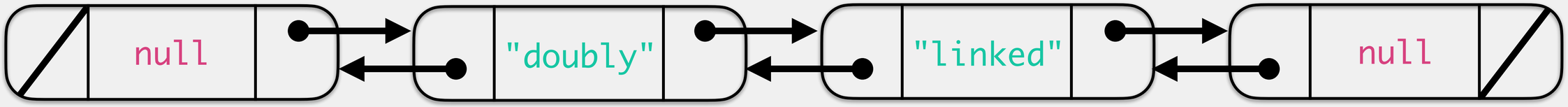
Doubly Linked Lists

Two modest changes to our singly linked list

- addition of a tail sentinel node
- a second link in each node to point to the previous node

Key advantages

- all “last” methods will now execute in similar time to the “first” versions
- can iterate in reverse
- no longer need to track previous node when performing operations

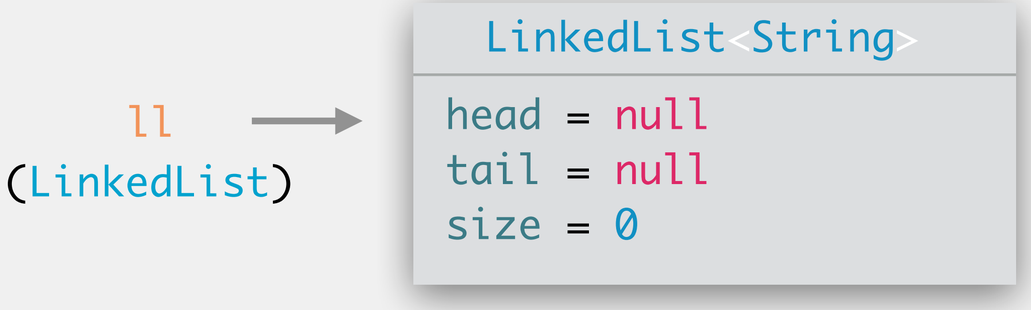


Sentinel Nodes Revisited

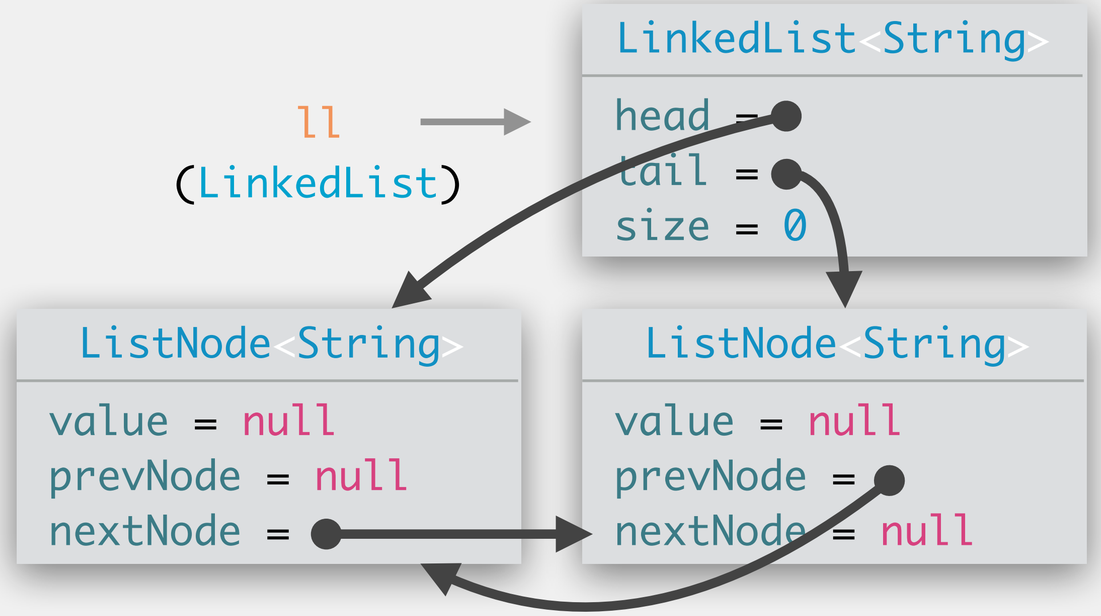
sentinel nodes are dummy nodes that hold a `null` value and indicate the end of the list

Empty List

without sentinel nodes
(what we had been doing)



with sentinel nodes
(what we will continue to do)



LinkedList: Attributes

```
LinkedList<E>

- head : ListNode<E>
- tail : ListNode<E>
- size : int

...
```

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

- will be a null sentinel node when the list is empty

`tail`: reference to the last node in the list

- will be a null sentinel node when the list is empty
- head and tail will point at each other

This is what Java's linked list representation looks like!

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
+ indexOf(Object o) : int
+ iterator() : Iterator<E>
+ lastIndexOf(Object o) : int
+ remove(int index) : E
+ remove(Object o) : boolean
+ size() : int
...
```

All the same methods as before

More complicated to implement in some sense…

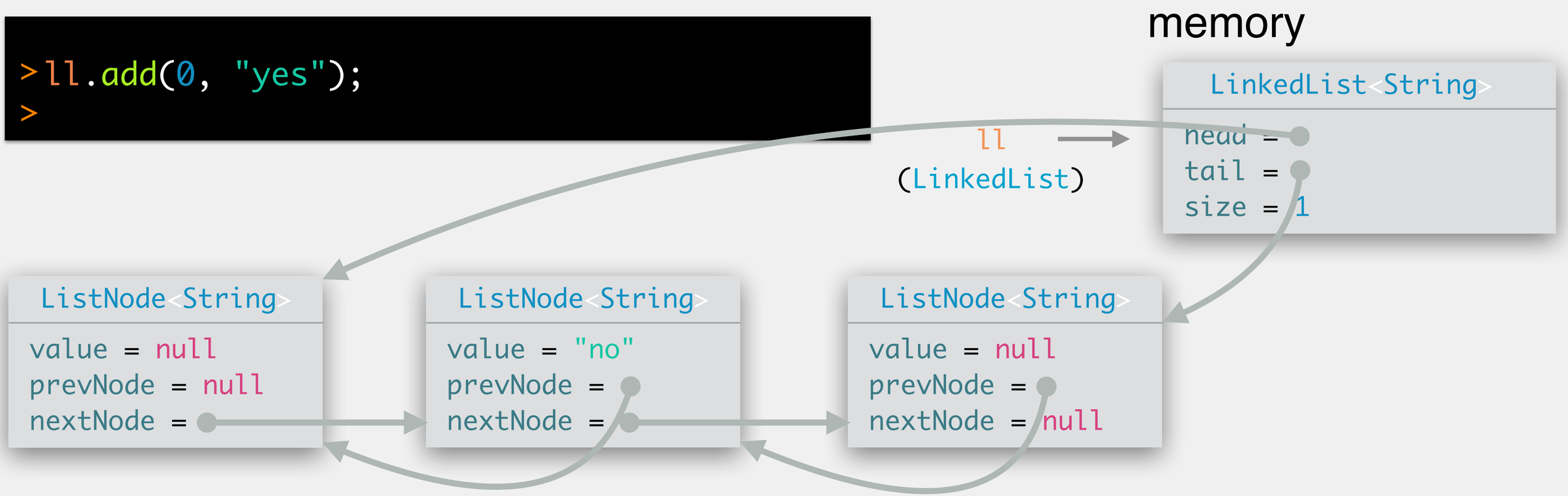
…but more efficient than singly linked lists

LinkedList: Add Methods

arguments: index to add at, element to add

behavior: adds value to that position in the linked list

```
>ll.add(0, "yes");
>
```

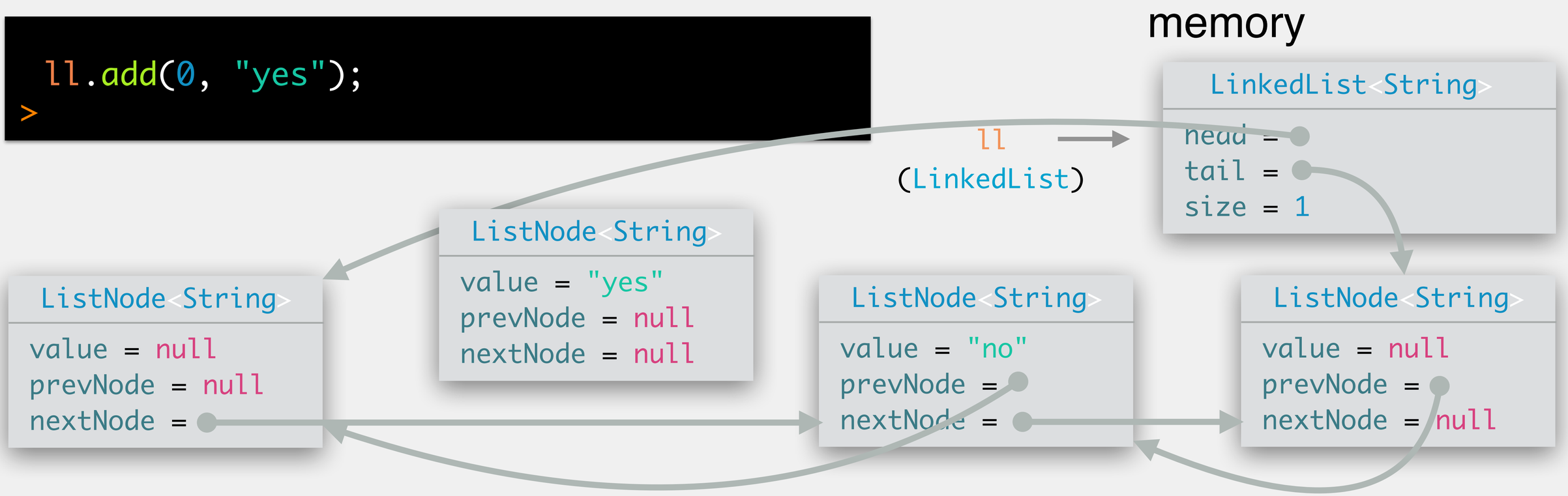


LinkedList: Add Methods

arguments: index to add at, element to add

behavior: adds value to that position in the linked list

```
ll.add(0, "yes");
>
```

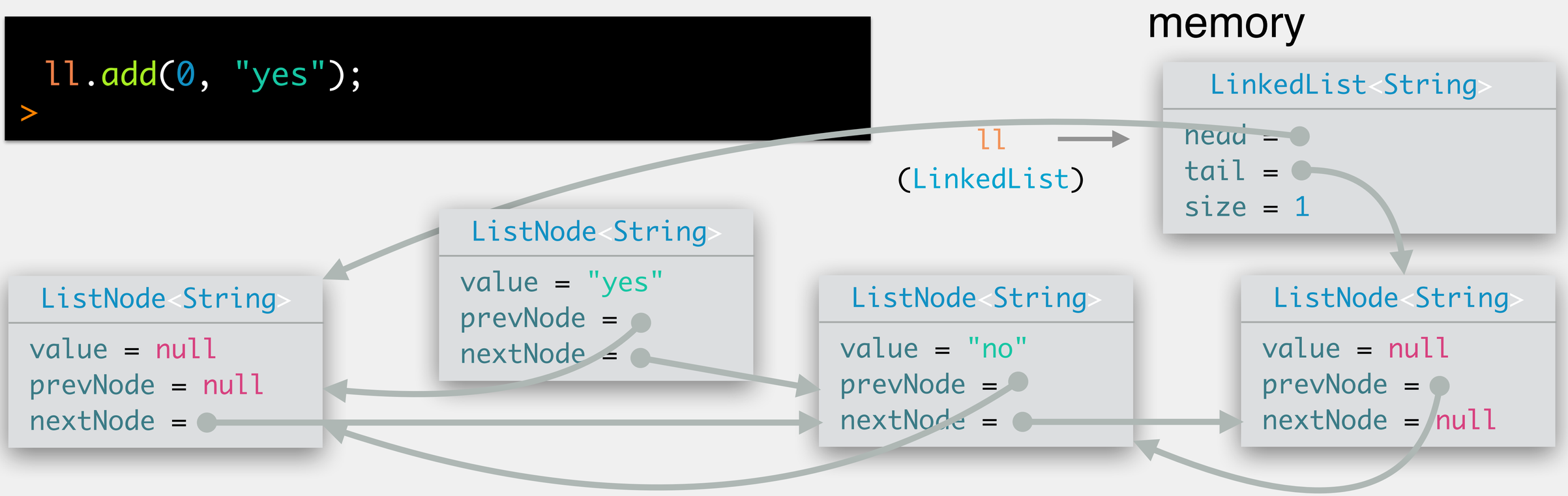


LinkedList: Add Methods

arguments: index to add at, element to add

behavior: adds value to that position in the linked list

```
ll.add(0, "yes");
>
```

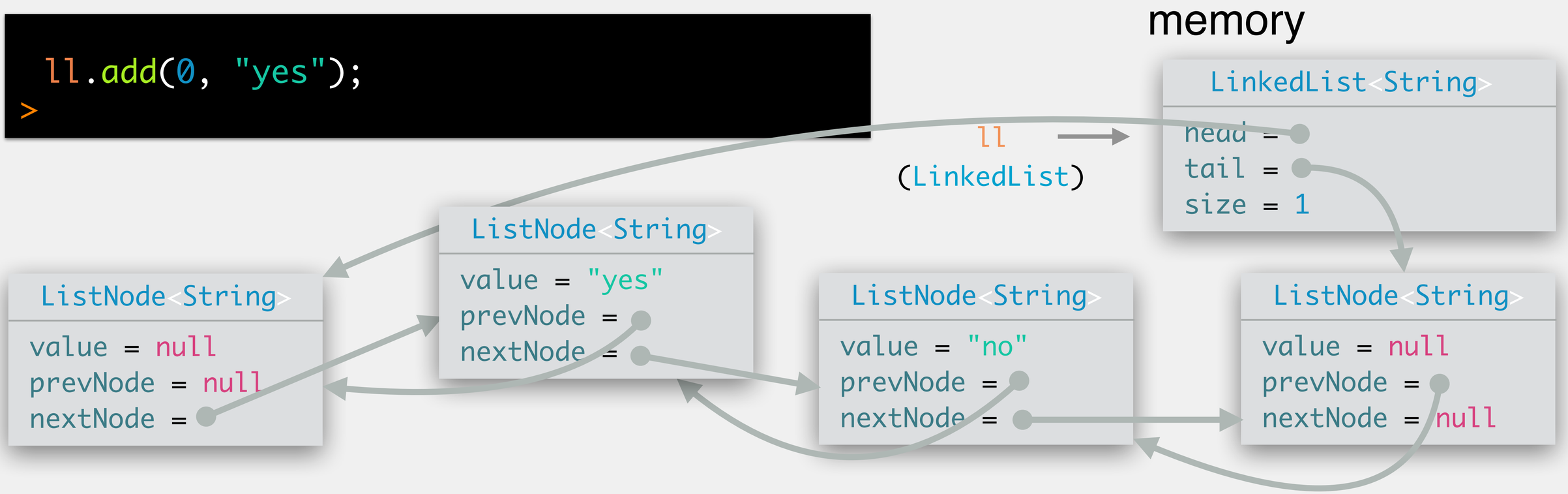


LinkedList: Add Methods

arguments: index to add at, element to add

behavior: adds value to that position in the linked list

```
ll.add(0, "yes");
>
```



Exercise: Doubly Linked List Methods

`public void clear()`

deletes all the nodes and sets `head` to point to `tail`; `isEmpty()` should be `true` after this

`public void addLast(E e)`

adds the element to the end of the list

`public void reverse()`

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

`public void swap(ListNode n1, ListNode n2)`

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

Write an iterator that iterates over the linked list in reverse