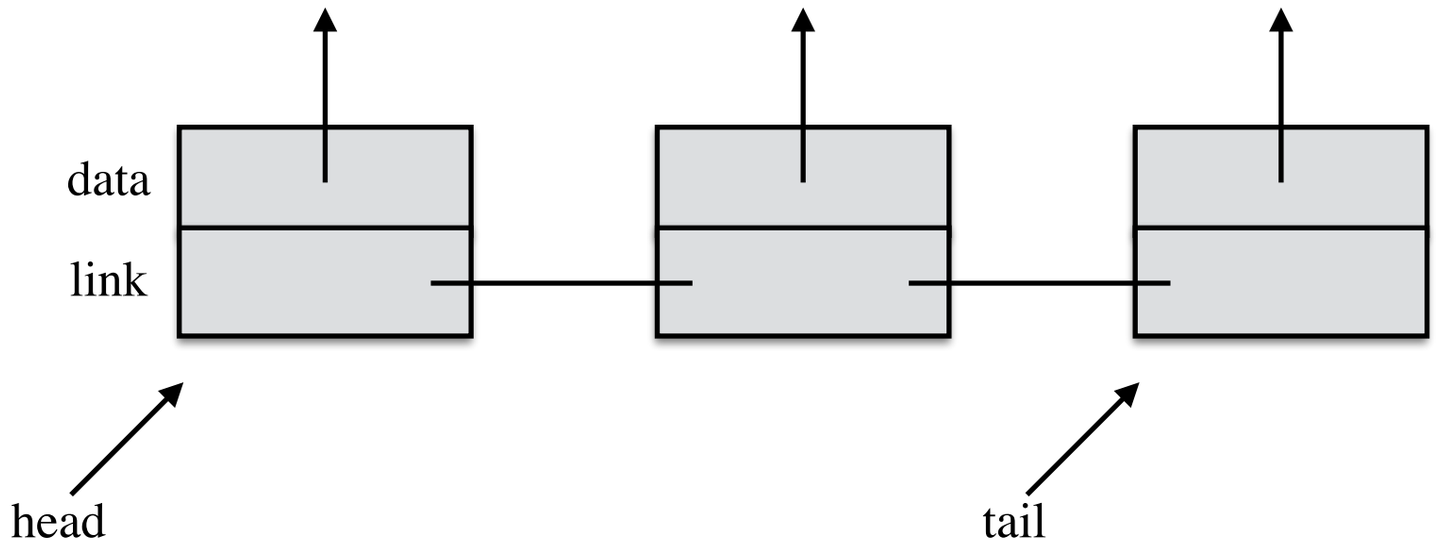
Linked Lists

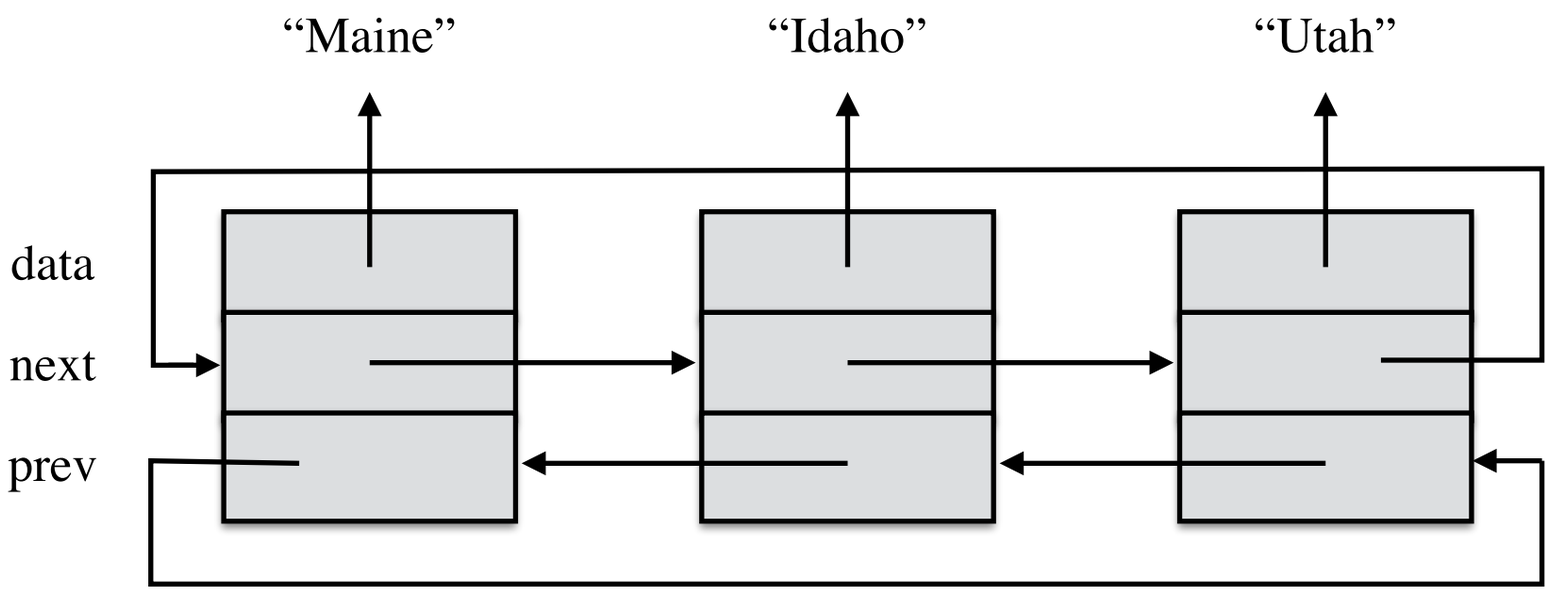

data
link
head
tail

Circularly Linked Lists w/ Sentinel

Resources:

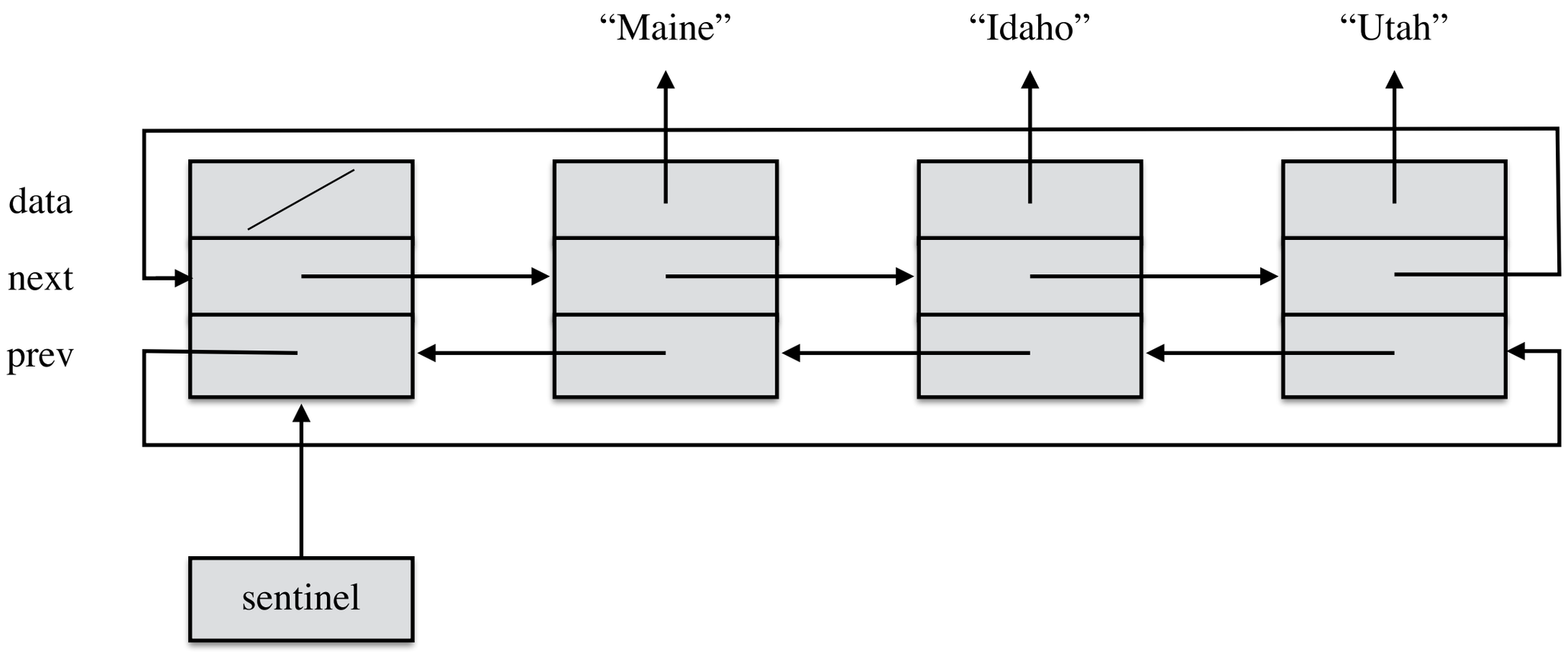
SentinelDLL.java *implements* **CS10LinkedLists.java**

Representation



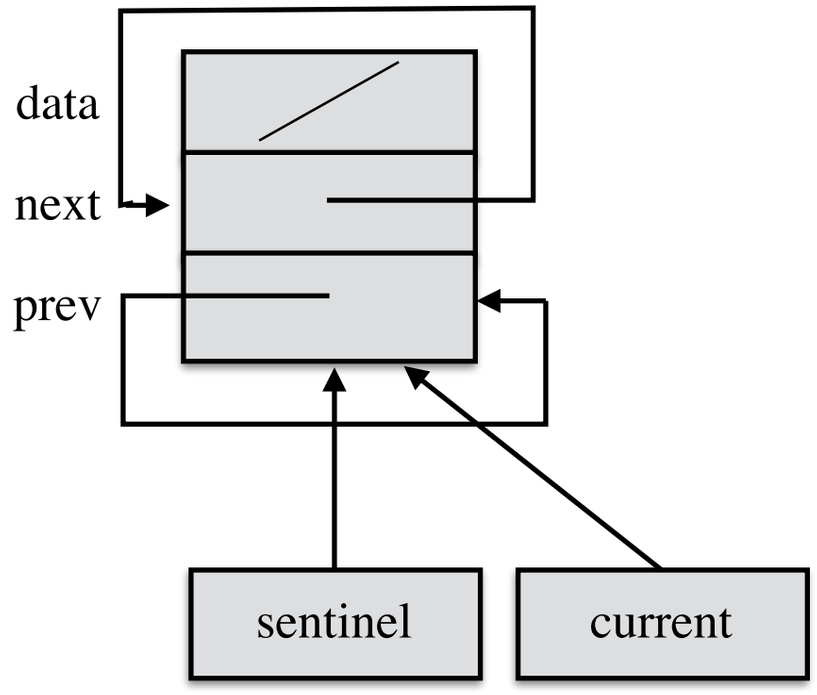
NOTE: Each of these references points not to individual instance data, but rather to an entire “node” object.

Representation (w/ Sentinel)

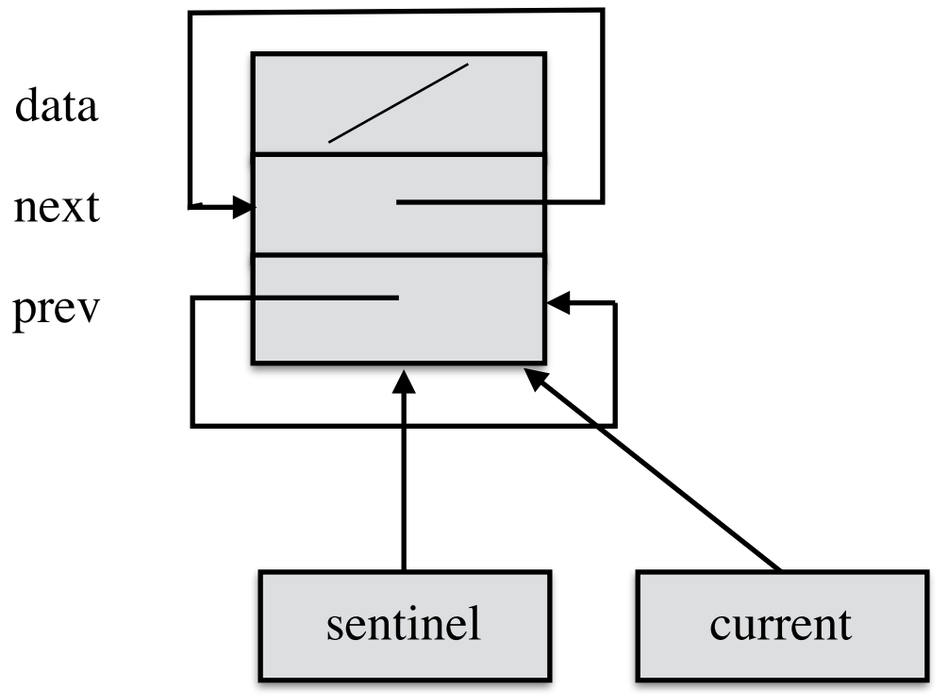


Empty List

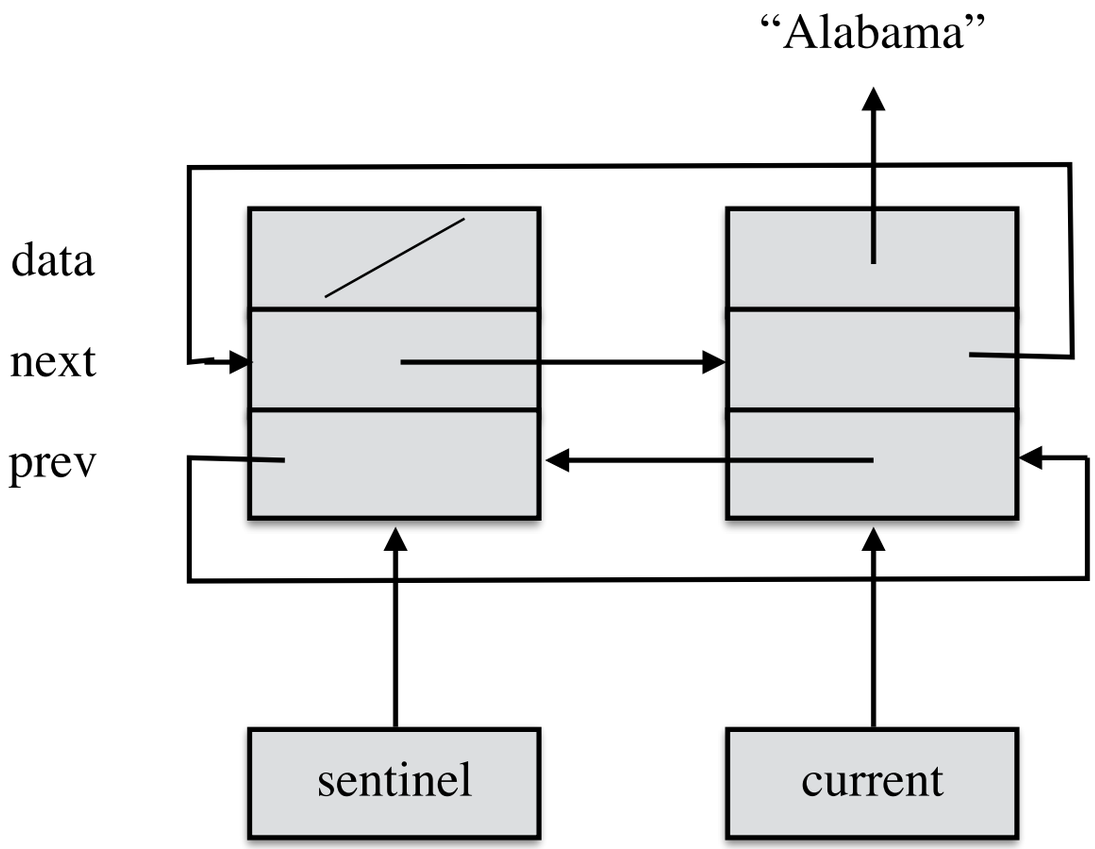
```
private Element<T> sentinel; // sentinel, serves as head and tail
private Element<T> current;  // current position in the list


. . .


/**
 * Constructor for a circular, doubly linked list with a sentinel.
 * Makes an empty list.
 */
public SentinelDLL() {
    sentinel = new Element<T>(null);
    clear();
}

/*
 * @see CS10LinkedList#clear()
 */
public void clear() {
    // Make the list be empty by having the sentinel point to itself
    // in both directions.
    sentinel.next = sentinel.previous = sentinel;

    // There's only one place to put the current reference...
    current = sentinel;
}
```



Adding (before)



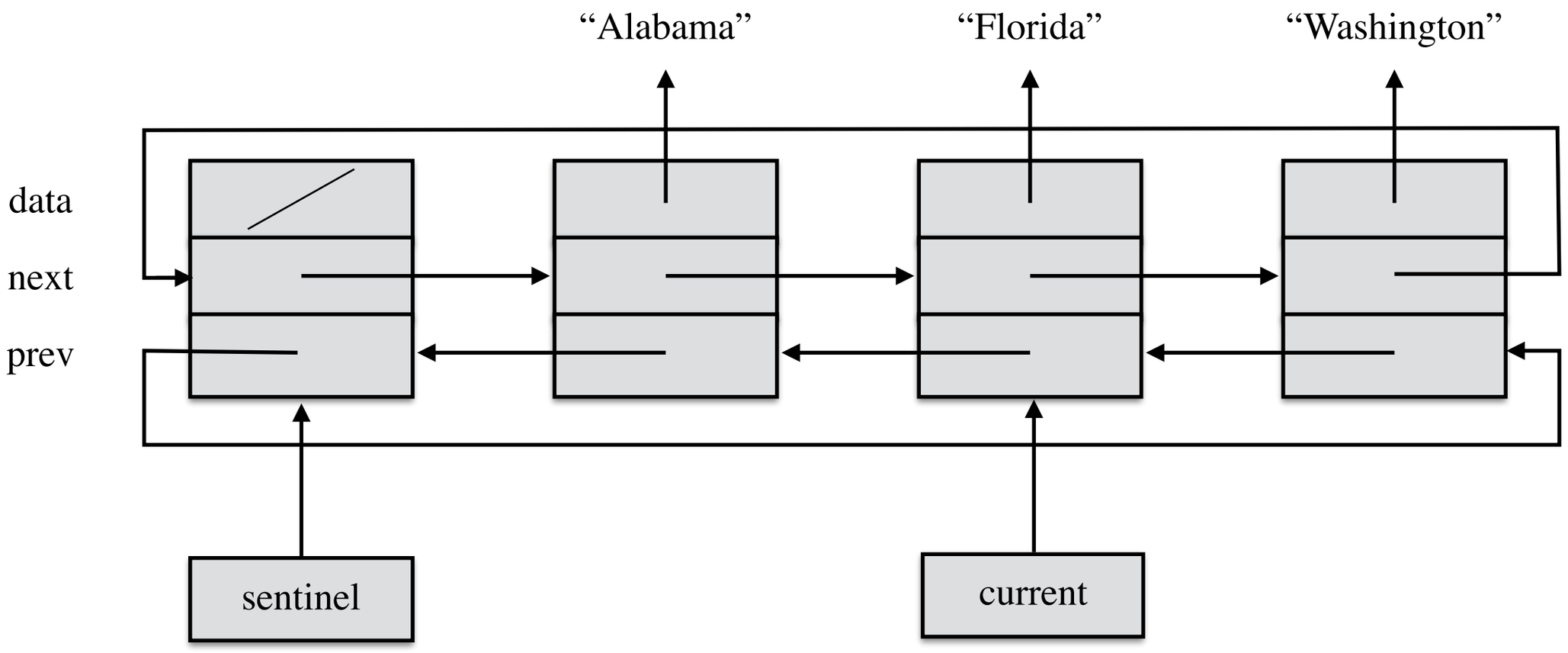
Adding (after)



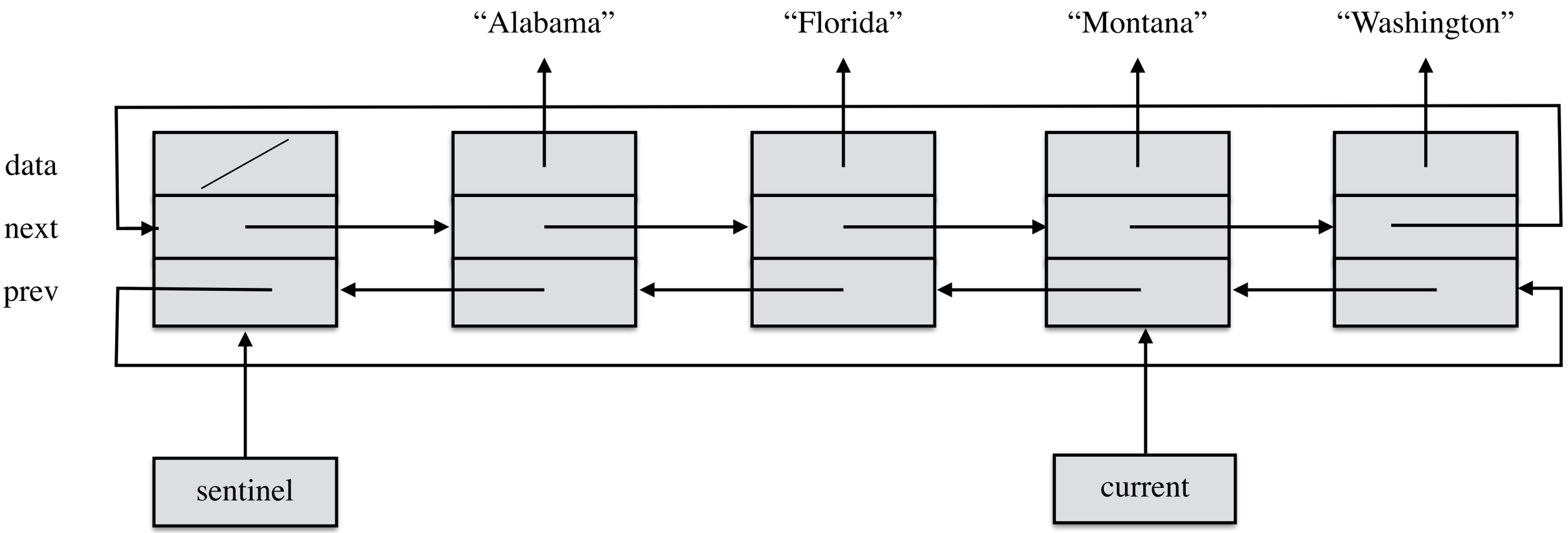
Adding (code)

```
/**
 * @see CS10LinkedList#add()
 */
public void add(T obj) {
    Element<T> x = new Element<T>(obj); // allocate a new element

    // Splice in the new element.
    x.next = current.next;
    x.previous = current;
    current.next.previous = x;
    current.next = x;

    current = x; // new element is current position
}
```

Adding (before)



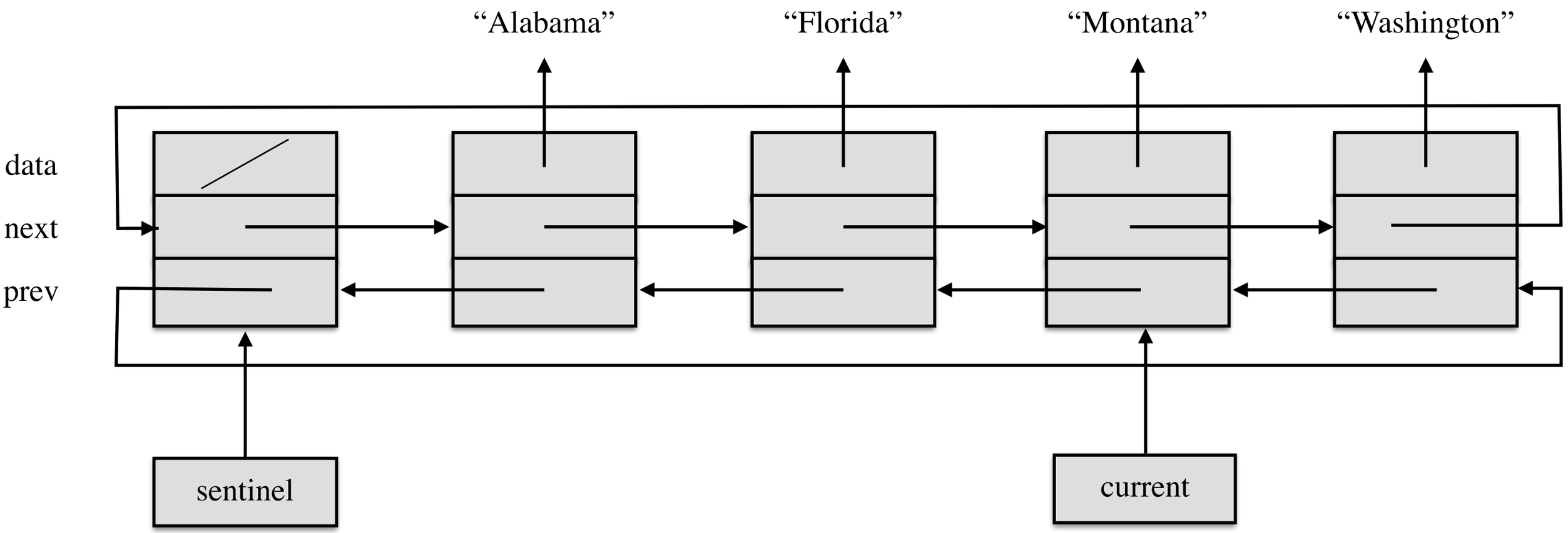
Adding (after)



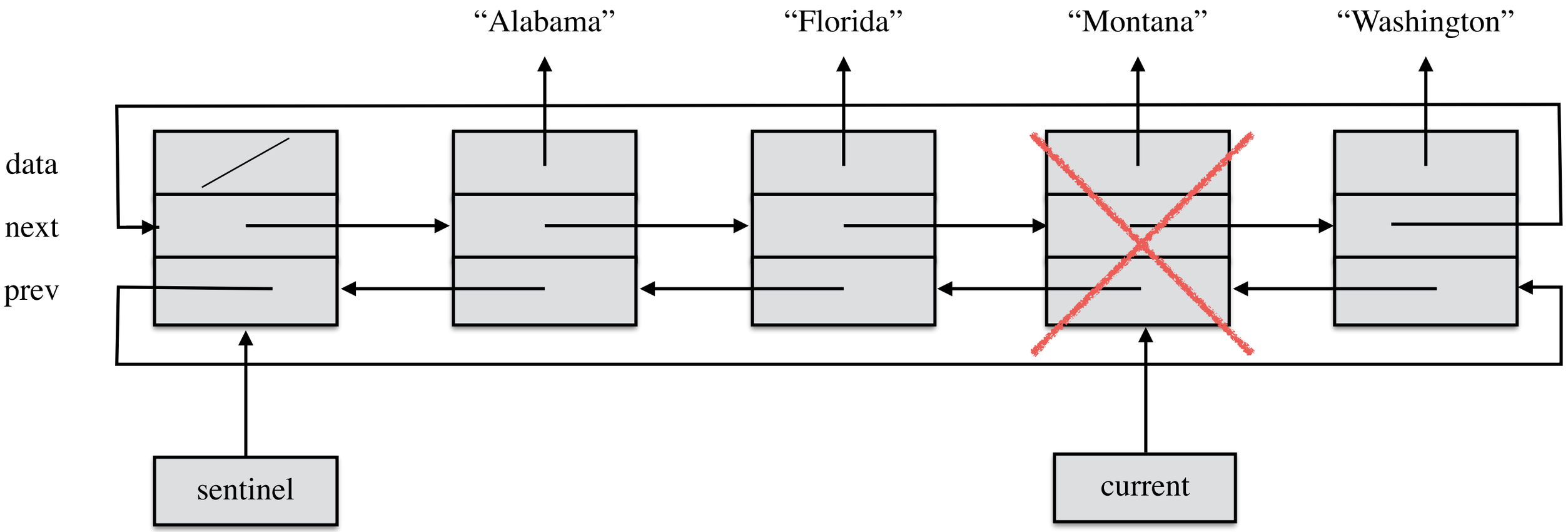
Removing (code)

```
/**
 * * @see CS10LinkedList#remove()
 */
public void remove() {
    // Do not ever let the sentinel be deleted!
    if (hasCurrent()) {
        // Splice out the current element.
        current.previous.next = current.next;
        current.next.previous = current.previous;
        current = current.next; // make successor the new current
    } else
        System.err.println("No current item");
}
```

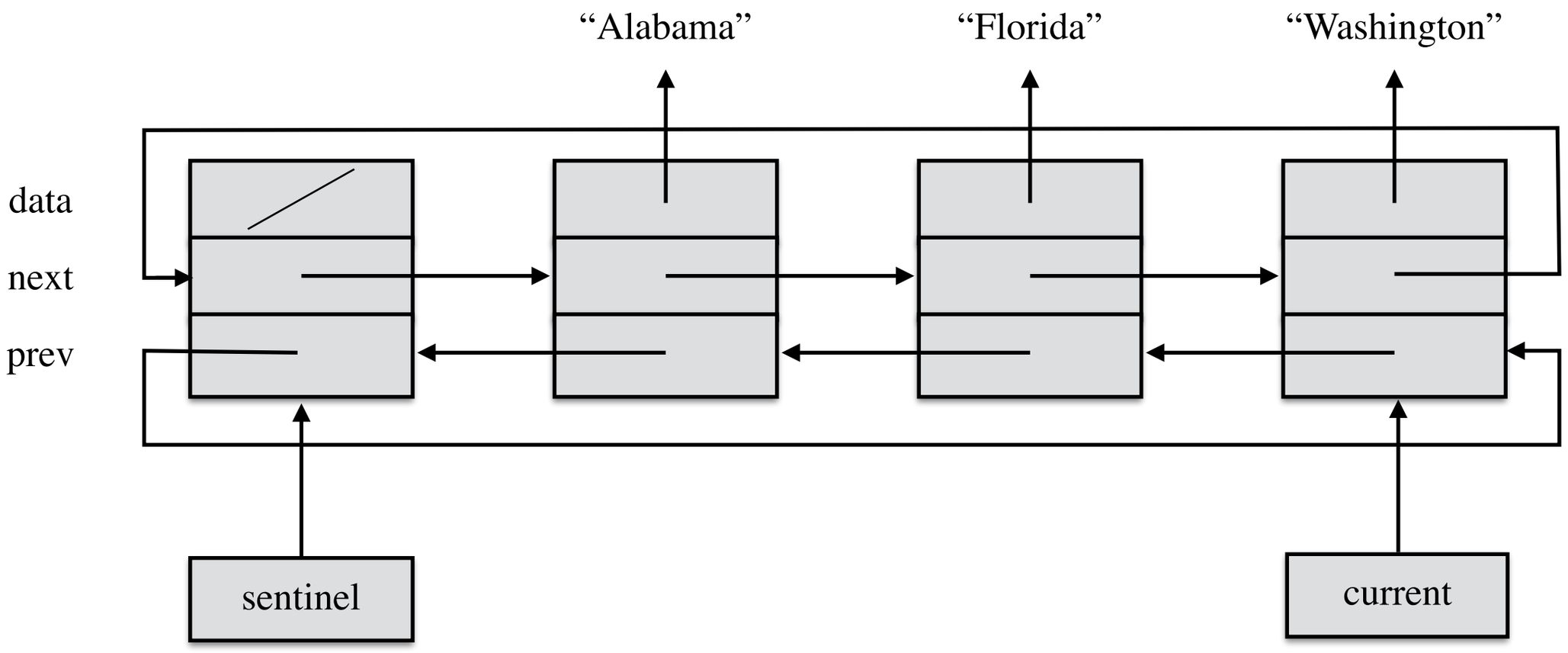
Removing (before)



Removing (before)



Removing (after)



Searching (code)

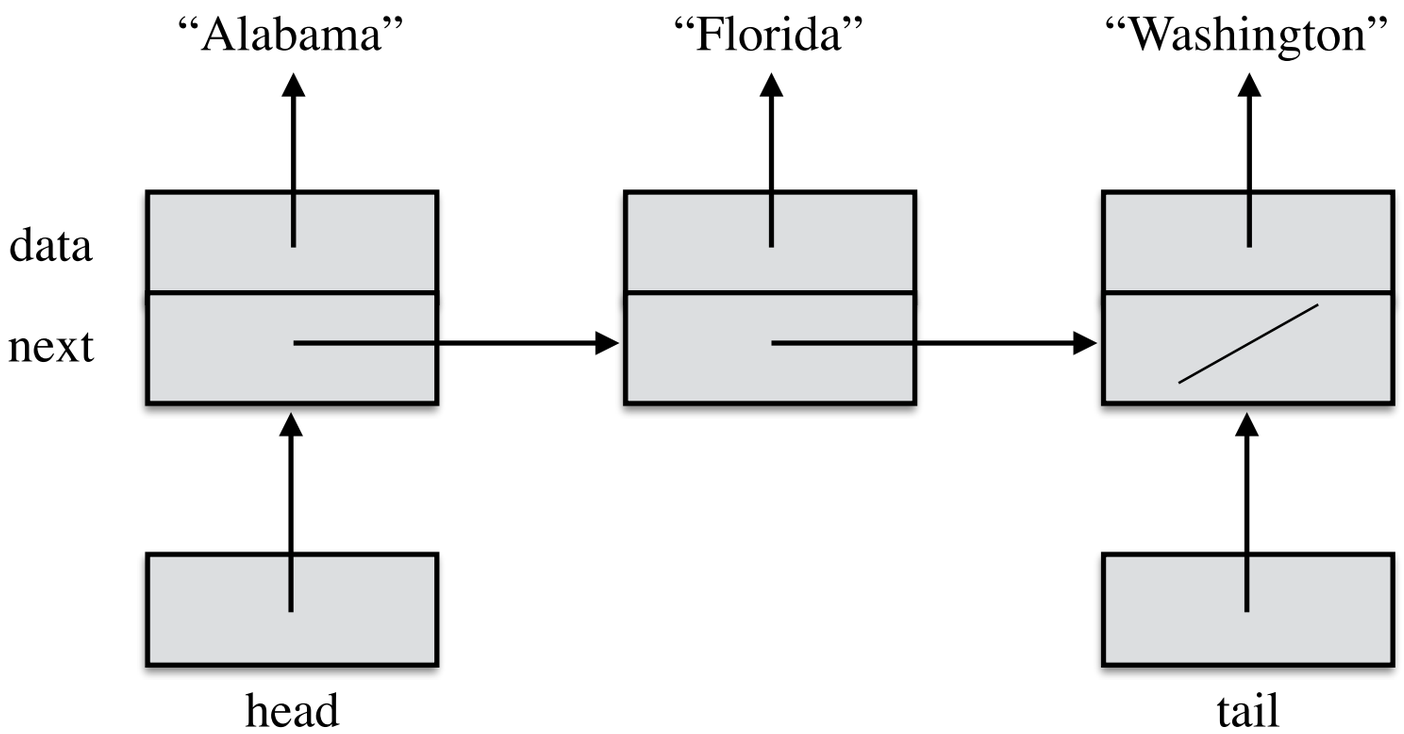
```
/**
 * @see CS10LinkedList#contains()
 */
public boolean contains(T obj) {
    Element<T> x;

    // Since we have a sentinel that isn't being used for a "real" element,
    // we put the object we are looking for in the sentinel. That way, we
    // know we'll find it some time during the search.
    sentinel.data = obj;

    for (x = sentinel.next; !x.data.equals(obj); x = x.next)
        ;

    // We dropped out of the loop because we found the target object in an
    // element that x references. If we found it in the sentinel, it wasn't
    // really in the list. If we found it before getting back to the sentinel,
    // it was in the list.

    // Put the sentinel back into its null state.
    sentinel.data = null;

    if (x == sentinel)
        return false; // tell the caller that we did not find the object
    else {
        current = x; // remember where we found it
        return true; // and tell the caller
    }
}
```

Singly Linked Lists

Resources:

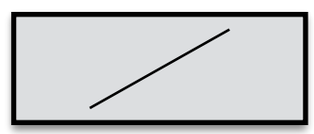
SLL.java *implements* **CS10LinkedLists.java**

Representation

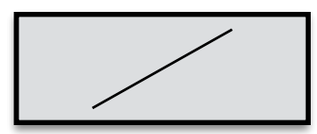


Empty List

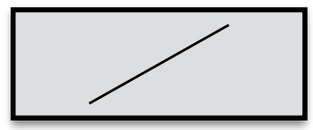
```
private Element<T> current; // current position in the list
private Element<T> head;    // head of list
private Element<T> tail;    // tail of list

. . .

/**
 * Constructor to create an empty singly linked list.
 */
public SLL() {
    clear();
}

/**
 * @see CS10LinkedList#clear()
 */
public void clear() {
    // No elements are in the list, so everything is null.
    current = null;
    head = null;
    tail = null;
}
```



head

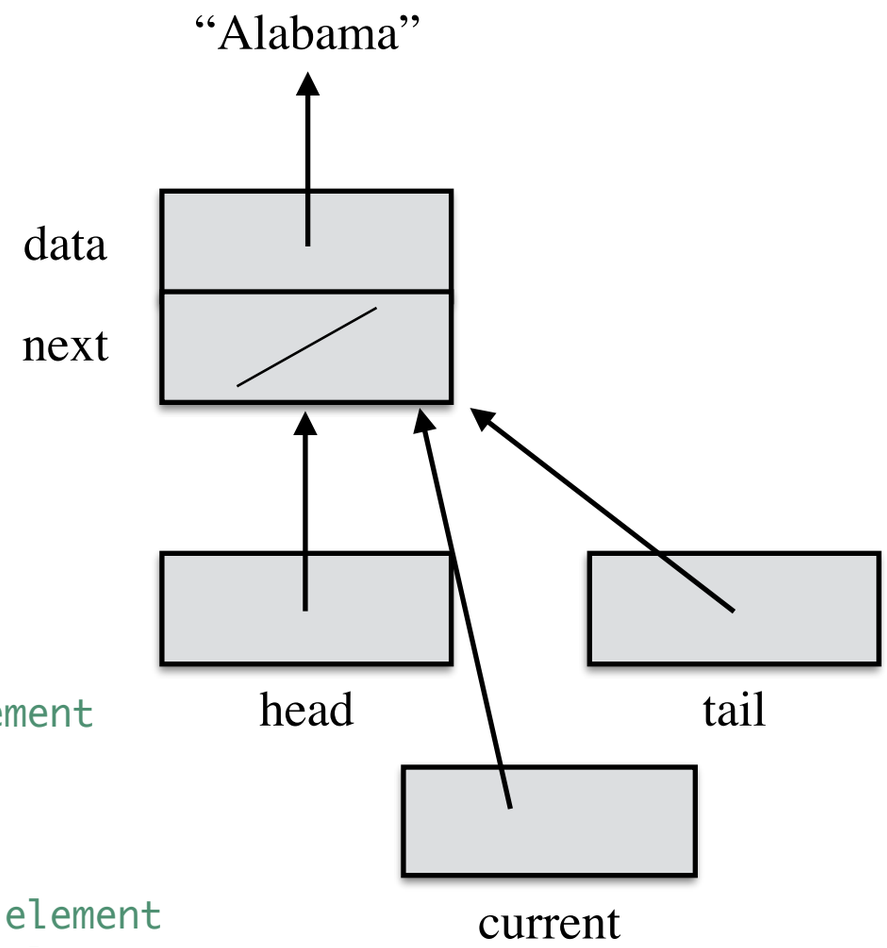


tail



current

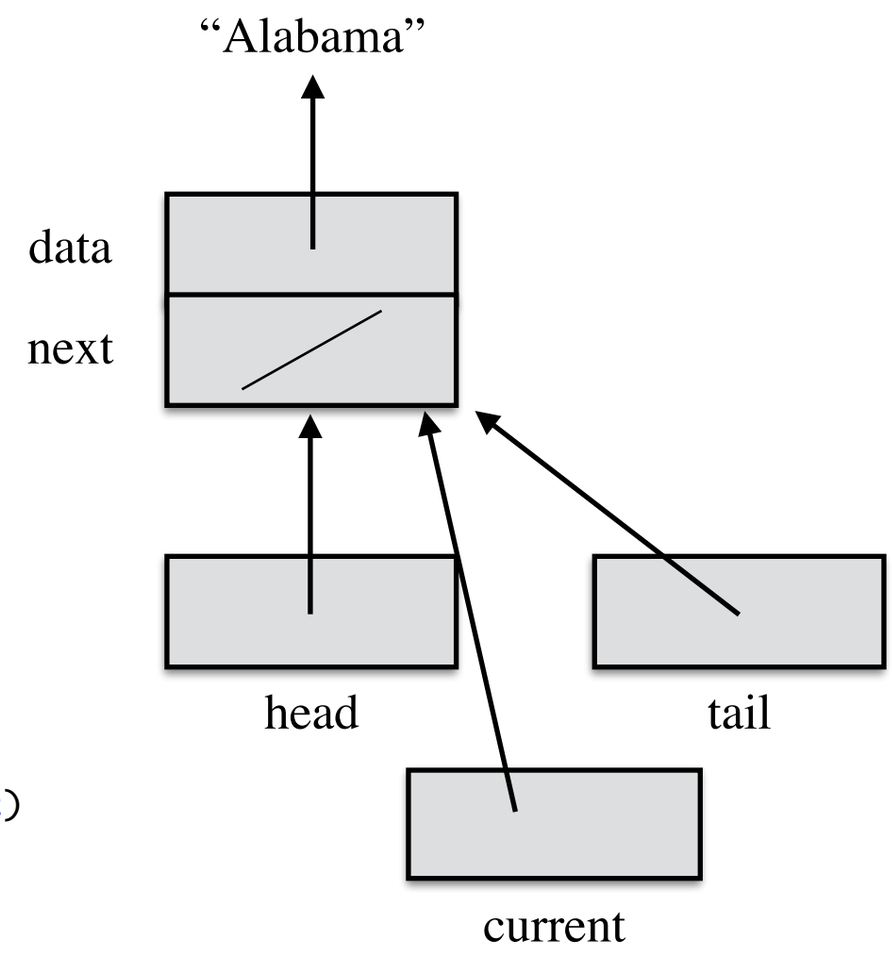
Adding




```
/**
 * @see CS10LinkedList#add()
 */
public void add(T obj) {
    Element<T> x = new Element<T>(obj); // allocate a new element

    // There are two distinct cases, depending on whether the new element
    // is to be the new head.
    if (hasCurrent()) {
        // The new element is not the new head.
        x.next = current.next; // fix the next reference for the new element
        current.next = x;      // fix the next reference for current element
    } else {
        // The new element becomes the new head.
        x.next = head; // new element's next pointer is old head
        head = x;      // and the new element becomes the head
    }

    // And check whether we need to update the tail.
    if (tail == current)
        tail = x;

    current = x; // new element is current position
}
```

Removing

```java
/**
 * * @see CS10LinkedList#remove()
 */
public void remove() {
    Element<T> pred; // current element's predecessor

    if (!hasCurrent()) { // check whether current element exists
        System.err.println("No current item");
        return;
    }

    if (current == head) {
        head = current.next; // no predecessor, so update head
        pred = null;
    } else {
        // NOTE: Finding the predecessor makes remove a linear-time
        // operation, rather than a constant-time operation.
        for (pred = head; pred != null && pred.next != current; pred = pred.next)
            ;

        // At this point, either pred == null, in which case we never found
        // the current element on the list (an error), or pred.next == current.
        if (pred == null) {
            System.err.println("Current item is not part of list.");
            return;
        }

        pred.next = current.next; // splice current out of list
    }

    // If we're removing the tail of the list, update that information.
    if (tail == current)
        tail = pred;

    current = current.next; // make the successor the current position
}
```



Searching (code)

```
/**
 * @see CS10LinkedList#contains()
 */
public boolean contains(T obj) {
    Element<T> x;

    for (x = head; x != null && !x.data.equals(obj); x = x.next)
        ;

        // We dropped out of the loop either because we ran off the end of the
        // list (in which case x == null) or because we found s (and so x !=
null).
        if (x != null)
            current = x;

        return x != null;
    }
```

SLL Variations

1. It is also possible to have a dummy list head, even if the list is not circular. If we do so, we can eliminate some special cases, because adding at the head becomes more similar to adding anywhere else. (Instead of changing the head you update a next field.)

2. (Lab 3) It is also possible to have current reference the element before the element that it actually indicates, so that removal can be done in constant time. It takes a while to get used to having current reference the element before the one that is actually “current."

3. It is also possible to have a circular singly linked list, either with or without a sentinel.

Inheritance
vs.
Interfaces

Resources:

SimpleList.java
GrowingArray.java
SinglyLinked.java
ListTest.java

Capturing Similarities

- Inheritance and Interfaces aren't all that different — they both offer a means of capturing similarities between objects.

- Inheritance says that a subclass "is-a" superclass, plus more (e.g., extra data, new or redefined methods)

- Interfaces specify a collection of methods headers (names, parameters, return types) without "bodies" implementing them.

Why use Inheritance?

- Can use (inherit) some common data or method implementations

- Ex. “Shape” class
  - data
    - name
    - color
    - x,y position
  - methods
    - toString()
    - getColor()/setColor()

- Subclasses get these “for free” and don’t have to implement this same stuff in each of their own classes.

Why use Interfaces?

- Ideal for specifying an Abstract Data Type (ADT)
- ADTs: collection of operations independent of data representation.
- Inheritance can lead to tricky and ambiguous scenarios.
  - Inherits from multiple superclasses
    - How to resolve different instance variables with the same name?
    - How to resolve different implementations of a method with the same name?
- Basic interface for lists: *SimpleList.java*

Growing Array List Implementation

Resources:

GrowingArray.java *implements* **SimpleList.java**

A growing array

- A GrowingArray *implements* SimpleList

- Rather than using a series of Element objects linked together, use an array

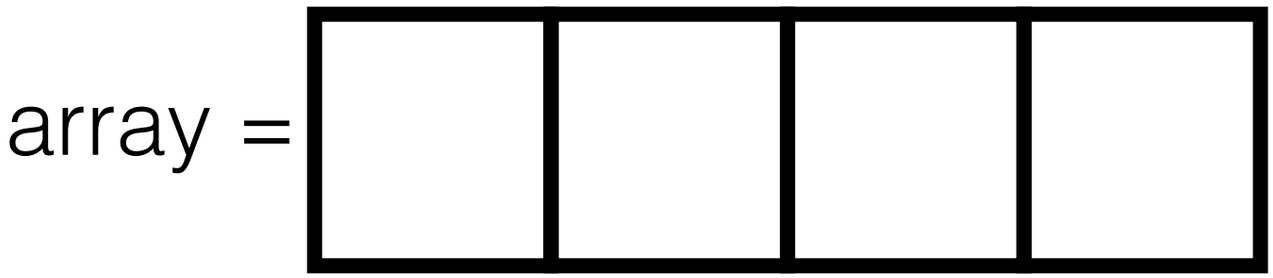
- Problem: arrays are created with a fixed length.

- Solution: “grow!”
  - create a new array (twice as big) and copy elements over
  - Keep track of “size” (# of elements in the list)
  - add()/remove() may require us to slide elements over

add()
until we grow!

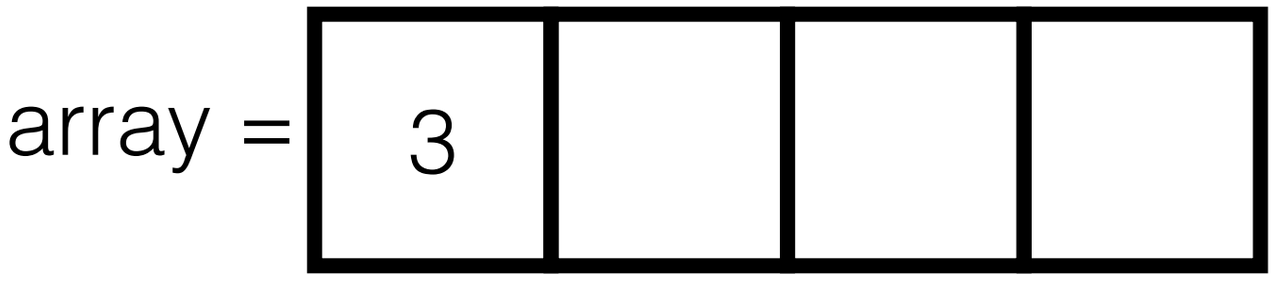
add()

size = 0



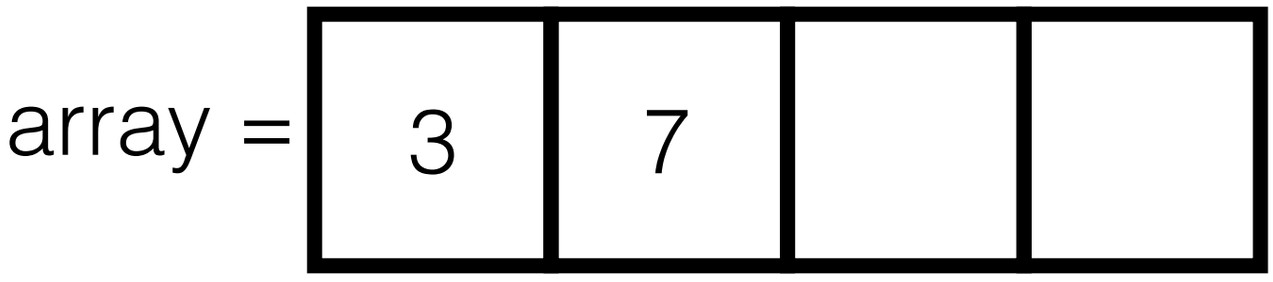
add()

size = 1



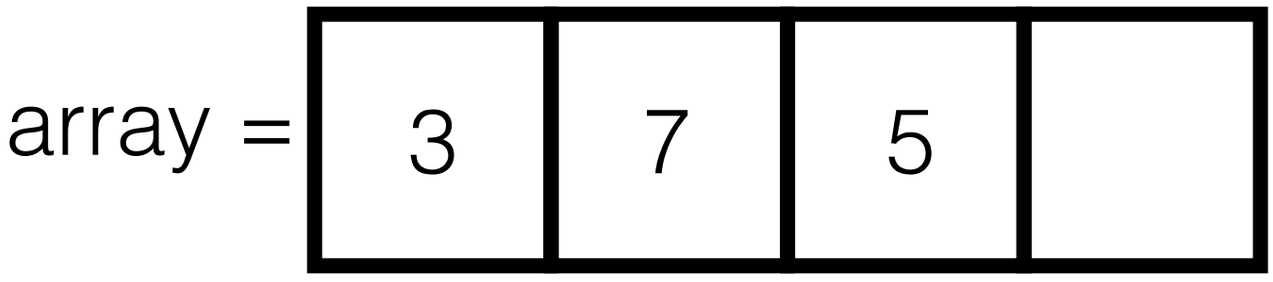
add()

size = 2



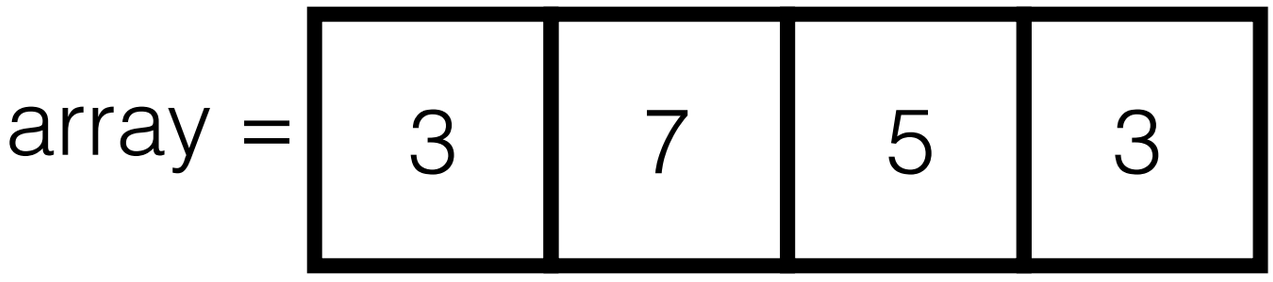
add()

size = 3



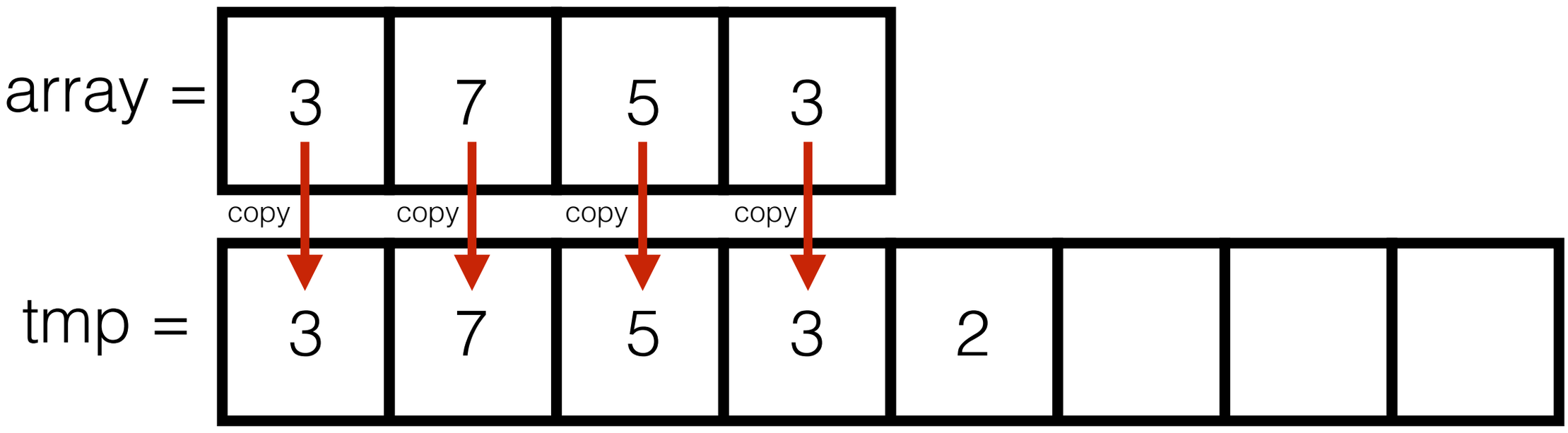
add()

size = 4



add() w/ “grow”

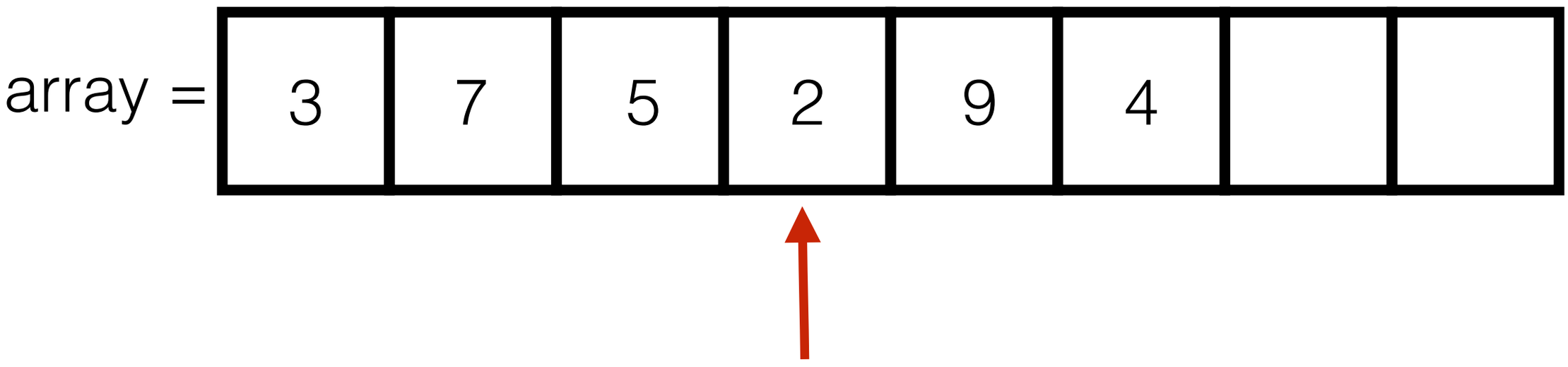
size = 5



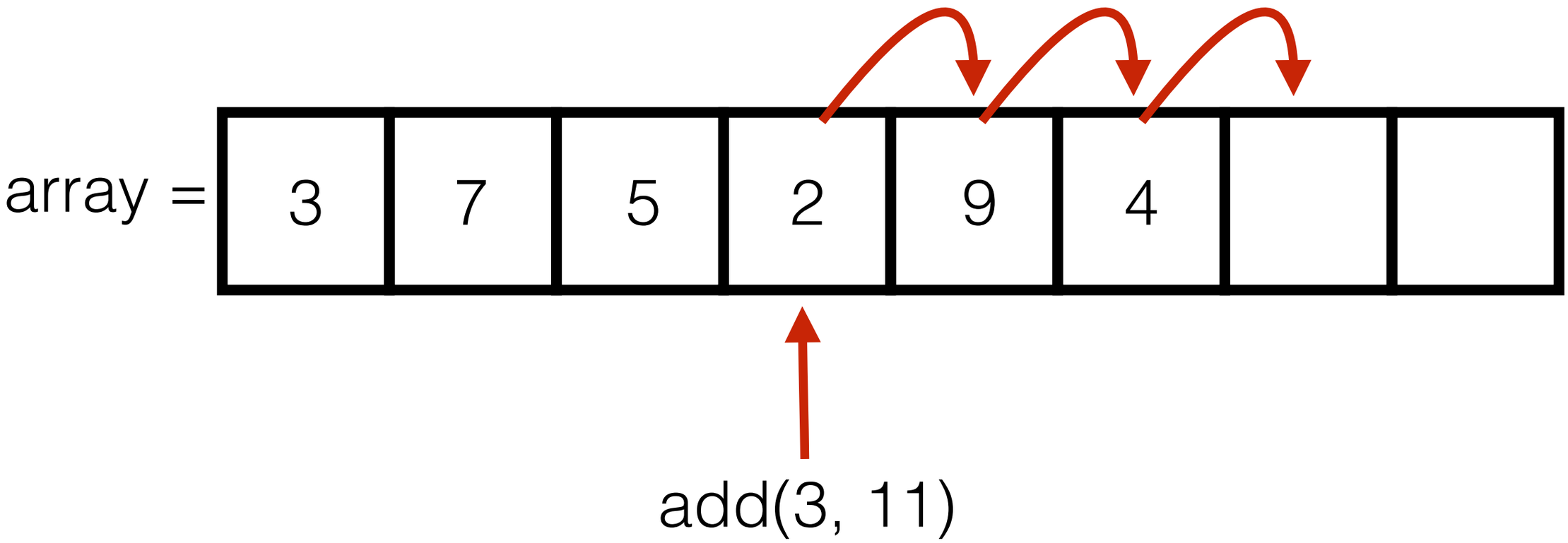
array = tmp

add()
in the middle

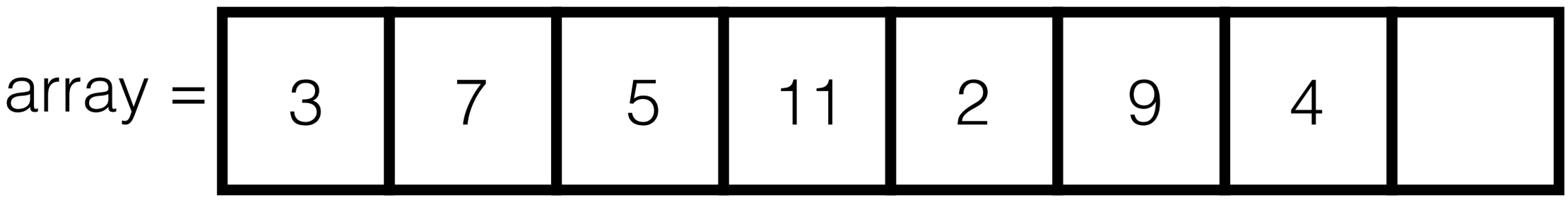
size = 6



size = 6

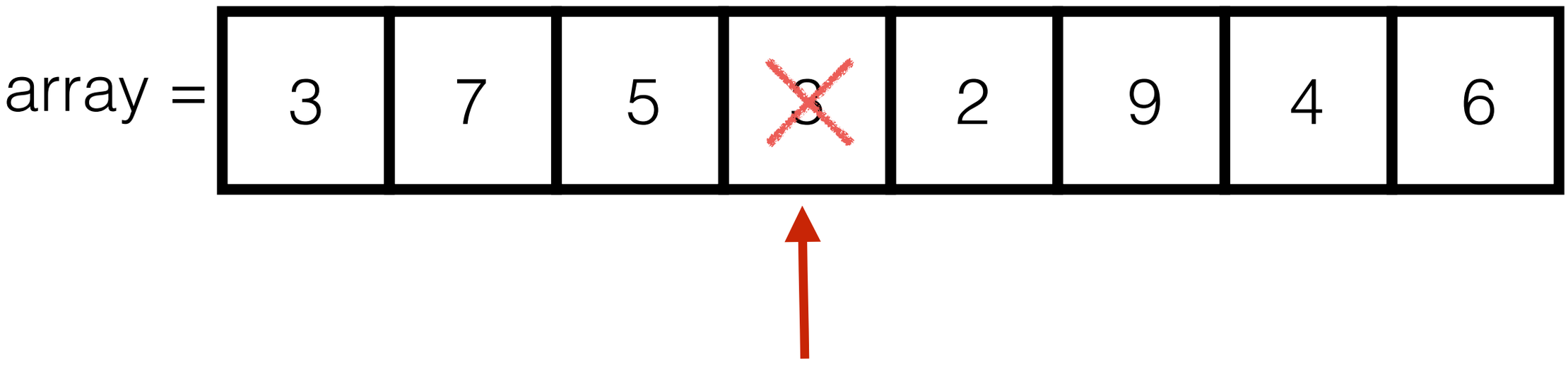


size = 7

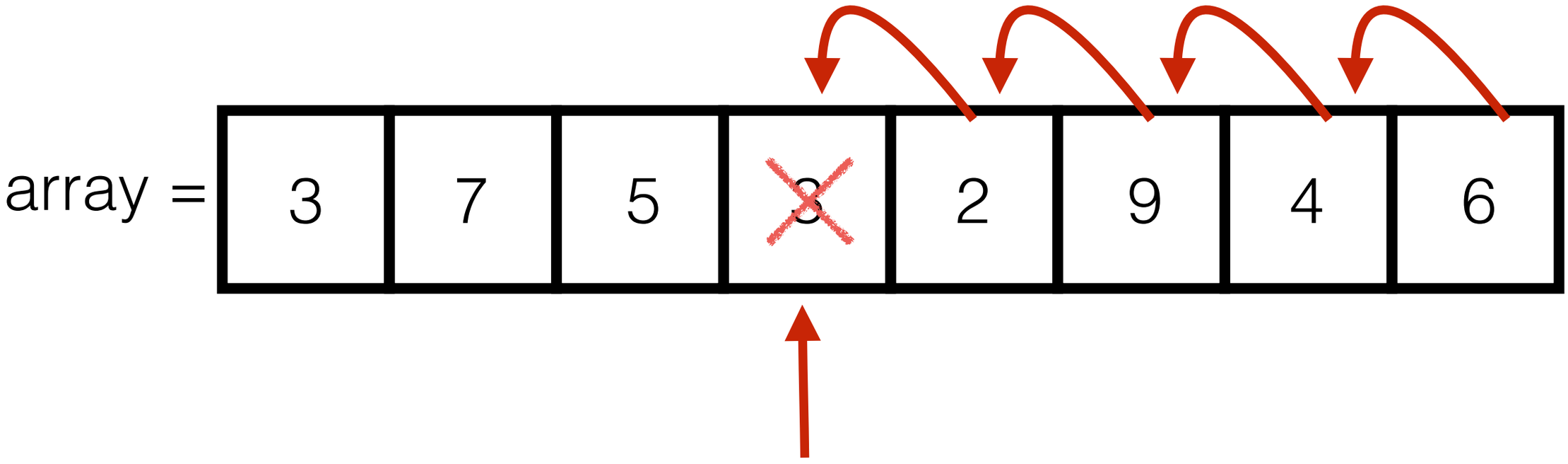


remove()
in the middle

size = 8



size = 8



size = 7