Two important data structures: resizing array list, singly-linked list

Alastair F. Donaldson

Aims of this lecture

- Show how to implement a resizing array-based list
- Show how to implement singly-linked list
- Discuss the pros and cons of these kinds of lists
- Prepare for the question: what is we would like to be able to flexibly switch between lists?

Resizing array-based list

Fixed-capacity list: not very useful

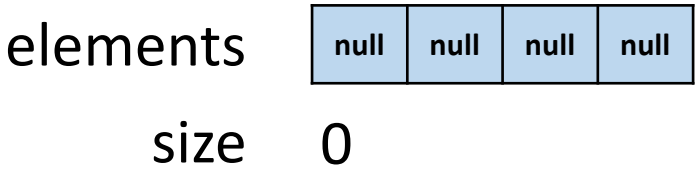
- If capacity is too small, we **run out of space**
- If capacity is too large, we **waste memory**

Let’s build a resizing array-based list

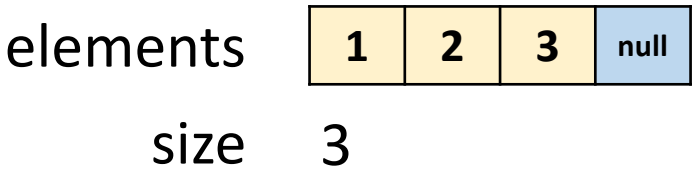
- Starts with an array of some **initial capacity**
- When array is full, switch to new array with **double the capacity**

Resizing array-based list

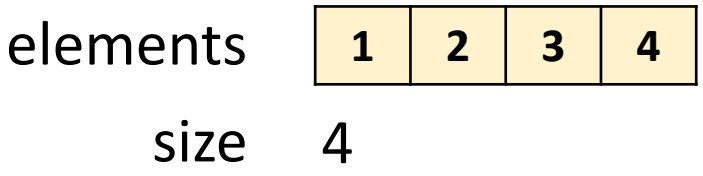
An empty list with capacity 4:



After adding three elements:

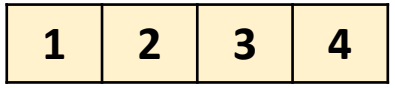


Adding one more element takes list to full capacity:



Resizing array-based list

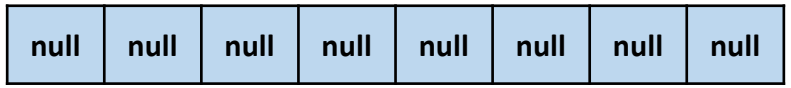
List at full capacity: elements



size 4

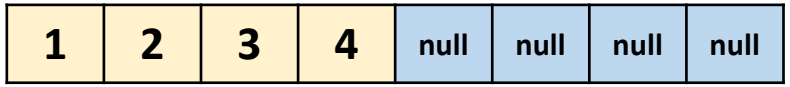
What happens when we add one more element?

newElements



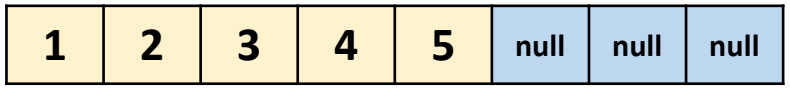
New array allocated with twice the capacity of **elements**

newElements



Existing data is coped in

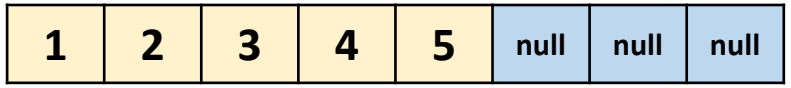
newElements



Value of new element is set

The elements **array** is replaced by the **newElements** array

elements



size 5

Resizing array-based list

A read-only property of this file, **private** means visible only within the file

```
package collections

private val DEFAULT_INITIAL_CAPACITY: Int = 16

class ResizingArrayList<T>(private val initialCapacity: Int) {

    init {
        if (initialCapacity < 0) {
            throw UnsupportedOperationException()
        }
    }

    constructor() : this(DEFAULT_INITIAL_CAPACITY)
```

Initialisation block, executed right after primary constructor

A **secondary** constructor

A secondary constructor calls primary constructor, via **this**

Creating an object using these two constructors

Creates a list with capacity 1024
using the **primary constructor**

```
val oneList = ResizingArrayList<Int>(1024)
val anotherList = ResizingArrayList<Int>()
```

Creates a list with default capacity
using the **secondary constructor**

A secondary constructor must use **this** to call some other constructor

Usually it calls the primary constructor – but could e.g. call another secondary constructor that in turn calls the primary constructor

Resizing array-based list

```
var size: Int = 0
    private set

private var elements: Array<T?> = clearedArray()

private fun clearedArray(): Array<T?> =
    arrayOfNulls<Any?>(initialCapacity) as Array<T?>
```

Array initialization logic extracted into a **helper method**, `clearedArray`, as we will use it again later

Most methods: same as for fixed-capacity list

```
fun get(index: Int): T = if (index !in 0..<size) {
    throw IndexOutOfBoundsException()
} else {
    elements[index]!!
}
```

Adding: what if we run out of space?

```
fun add(index: Int, element: T) {
    if (index !in 0..size) {
        throw IndexOutOfBoundsException()
    }

    for (i in size downTo index + 1) {
        elements[i] = elements[i - 1]
    }
    elements[index] = element
    size++
}
```

If **size** > **elements.size** we have an out-of-bounds access

Adding: what if we run out of space?

```
fun add(index: Int, element: T) {
    if (index !in 0..size) {
        throw IndexOutOfBoundsException()
    }
    if (size + 1 > elements.size) {
        elements = elements.copyOf(2 * elements.size)
    }
    for (i in size downTo index + 1) {
        elements[i] = elements[i - 1]
    }
    elements[index] = element
    size++
}
```

If we need more space, allocate a new array twice as large and copy the old array in

Clearing the list

```
fun clear() {
    elements = clearedArray()
    size = 0
}
```

Recall:

```
private fun clearedArray(): Array<T?> =
    arrayOfNulls<Any?>(initialCapacity) as Array<T?>
```

This is a **helper method** – captures common logic used multiple times in the class

Helper methods should be **private**: they do not provide services to other classes

Exercise: complete the `ResizingArrayList<T>` class

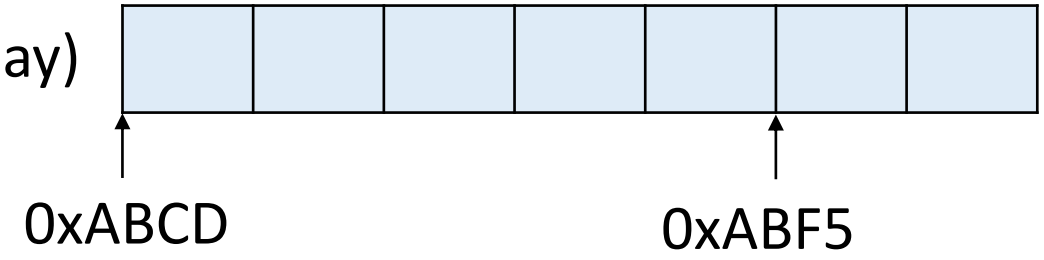
- Provide the full set of methods that were available for `FixedCapacityList<T>`

Properties of resizing array-based list

How efficient is it to get the element at index *i*?

Efficient: compute memory location *i* objects after start of array, return object at that location

Array starts at memory address 0xABCD (say)



Each element is an object reference
– 8 bytes on a 64-bit machine

`list.get(5)` – return the object reference stored at address:

0xABCD + 5 * 8 = 0xABF5

Array start Index Element size

`get` involves a multiply and an add – very fast

Properties of resizing array-based list

How efficient is it to add an element to the end of the list?

Usually, **very efficient**: insert element into next free slot

Occasionally, **very slow**: requires a resize operation

The cost of resizing is **amortized**: resizes are rare and become rarer as size of array is doubled on each resize

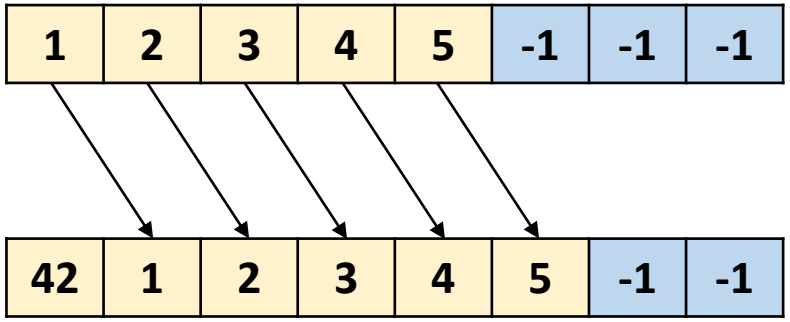
Properties of resizing array-based list

How efficient is it to add an element earlier in the list?

Expensive: all remaining elements must be moved

Worst case: insert at the start of the list – entire contents must be moved

```
add(index = 0, element = 42)
```

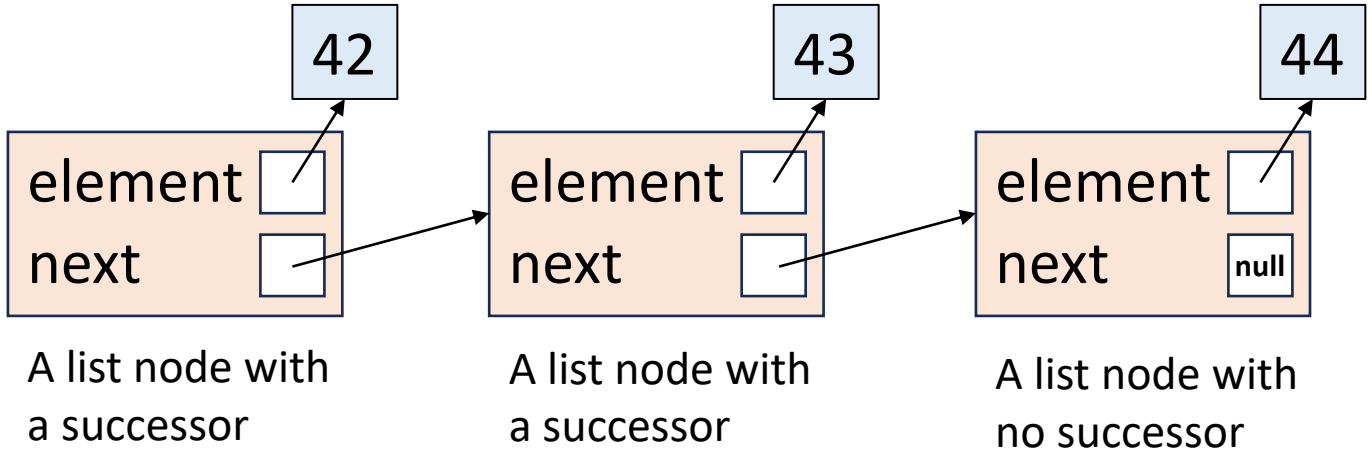


Singly-linked lists

A list represented as a chain of **nodes**

A node is a pair:

- A reference to an object – the element stored by the node
- A **nullable** reference to the next node – **null** if node has no successor



The node class

The default value of **next** is **null** – this is a **default parameter**

```
class Node<T>(var element: T, var next: Node<T>? = null)
```

Generic: we can have a node storing elements of any type **T**

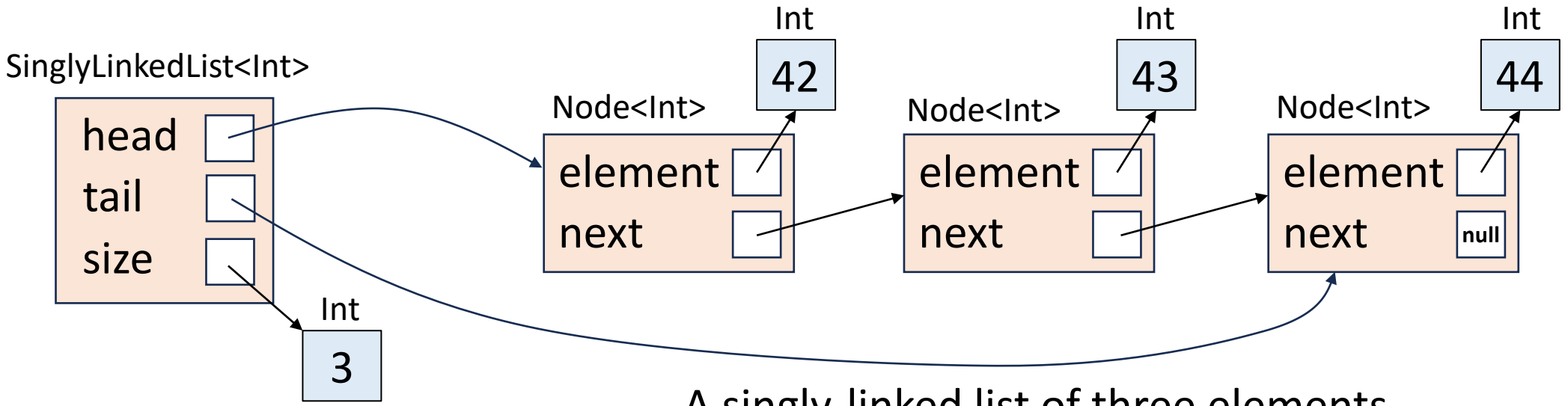
The **next** property is nullable: a node might not have a successor

We use var properties so that the node can be updated – necessary for a **mutable** list

A singly-linked list of nodes

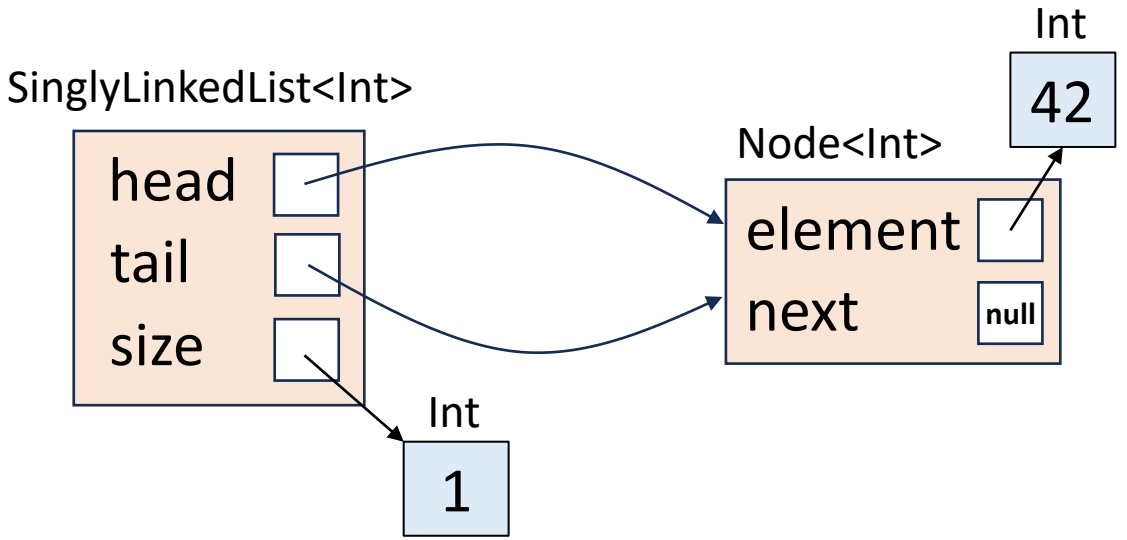
The list class comprises:

- The **head** of the list – (a reference to) a list node
- The **tail** of the list – (a reference to) a list node
- The **size** of the list – (a reference to) an integer

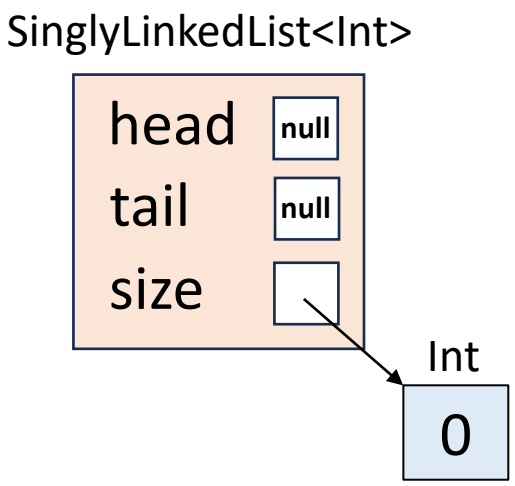


A singly-linked list of three elements

A singly-linked list with one element



An empty singly-linked list



Singly-linked list declaration: first attempt

Anyone can create instances of this **Node** class – that's bad; we only introduced it to support **SinglyLinkedList**

```
class Node<T>(var element: T, var next: Node<T>? = null)

class SinglyLinkedList<T>() {

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    var size: Int = 0
        private set
    ...
```

Empty primary constructor – better style to omit it

General rule: wherever possible, try to hide internal details of your classes – more on this later

Better: make `Node` private

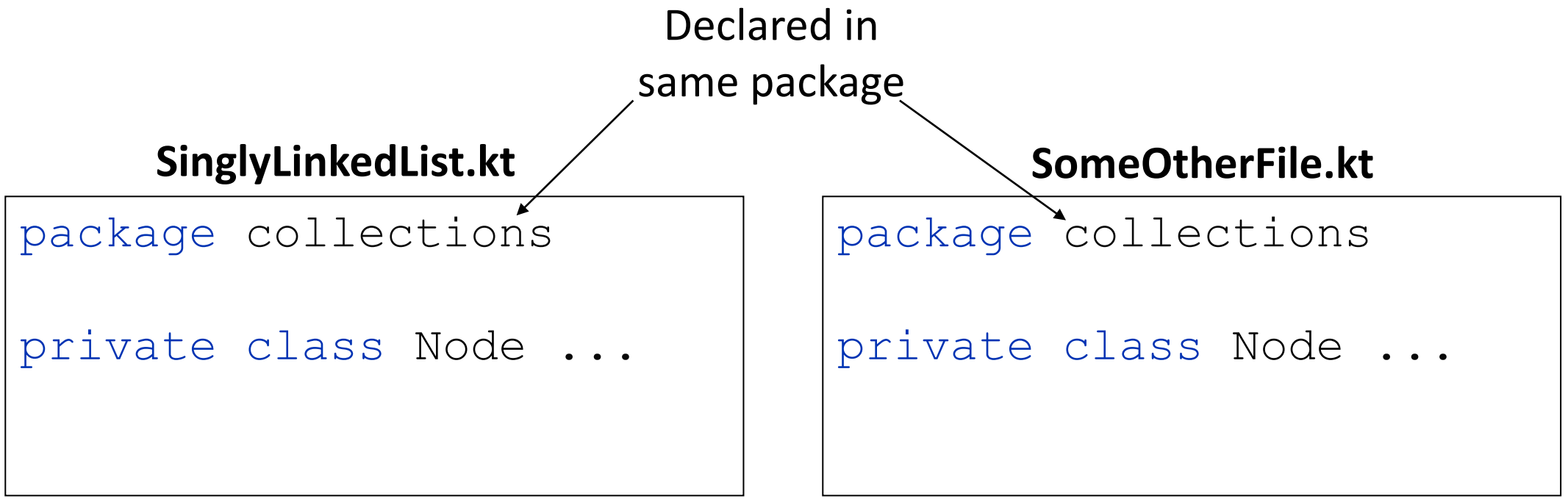
Now **Node** is only visible to code in this Kotlin file

```
private class Node<T>(var element: T,
                      var next: Node<T>? = null)

class SinglyLinkedList<T> {

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    var size: Int = 0
        private set
    ...
```

Problem: name clashes between files



Error: redeclaration of class **Node**

Best: make `Node` a private **nested class**

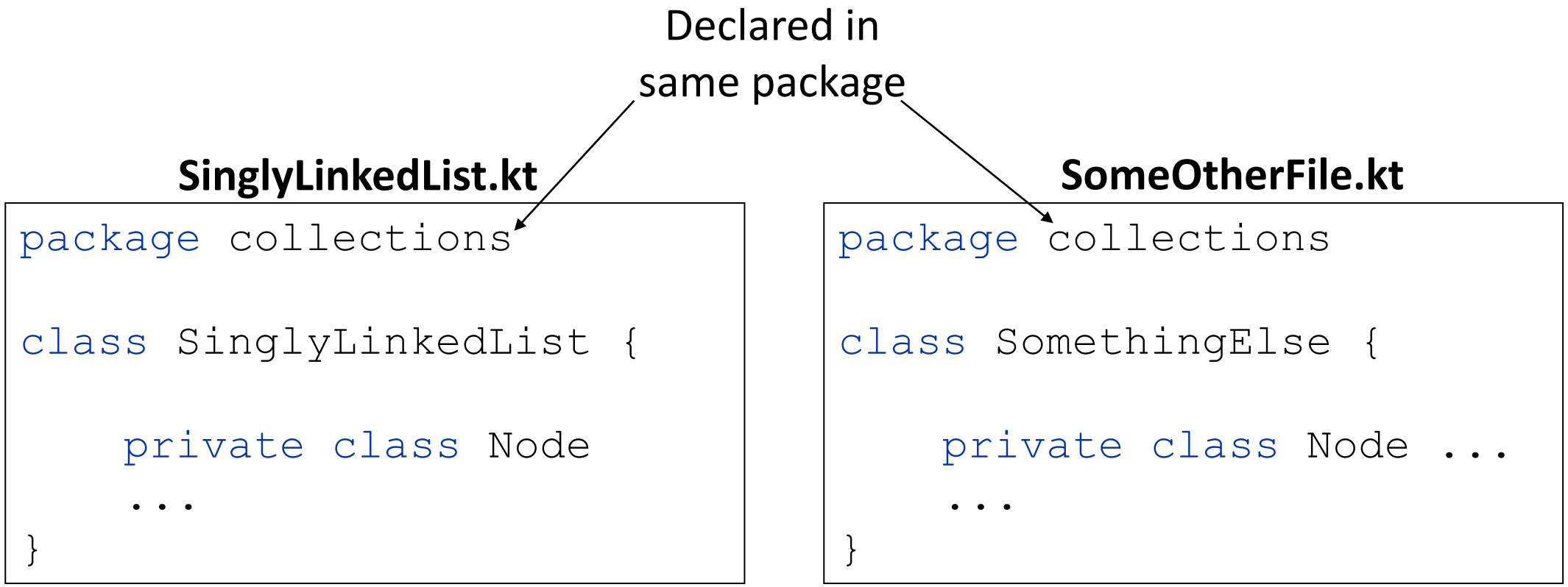
```
class SinglyLinkedList<T> {

    private class Node<T>(var element: T,
                          var next: Node<T>? = null)

    private var head: Node<T>? = null
    private var tail: Node<T>? = null
    var size: Int = 0
        private set
    ...
```

Because **Node** is nested inside SinglyLinkedList, its full name is **SinglyLinkedList.Node**

No more name clashes

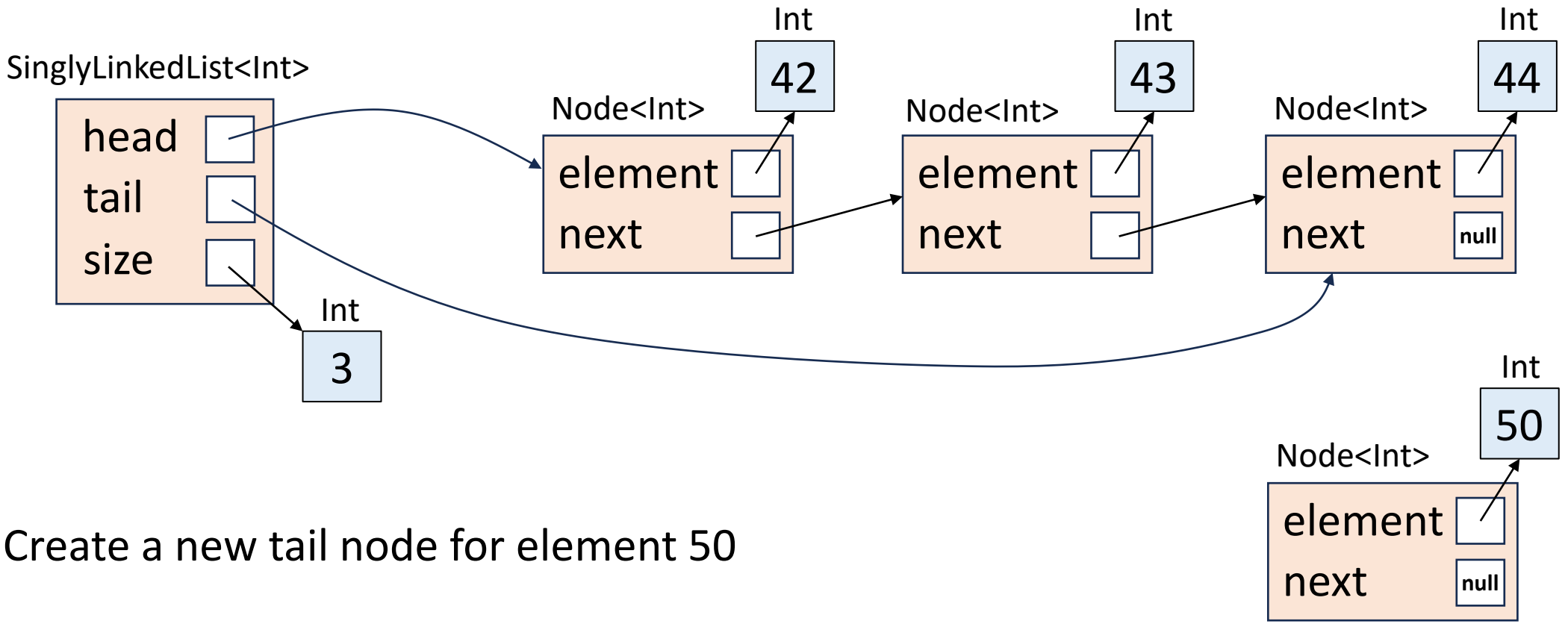


No redeclaration: **SinglyLinkedList**.**Node** and **SomethingElse**.**Node** can live in harmony!

Adding an element to the end of the list

`list.add(50)`

Add the integer 50 to the list

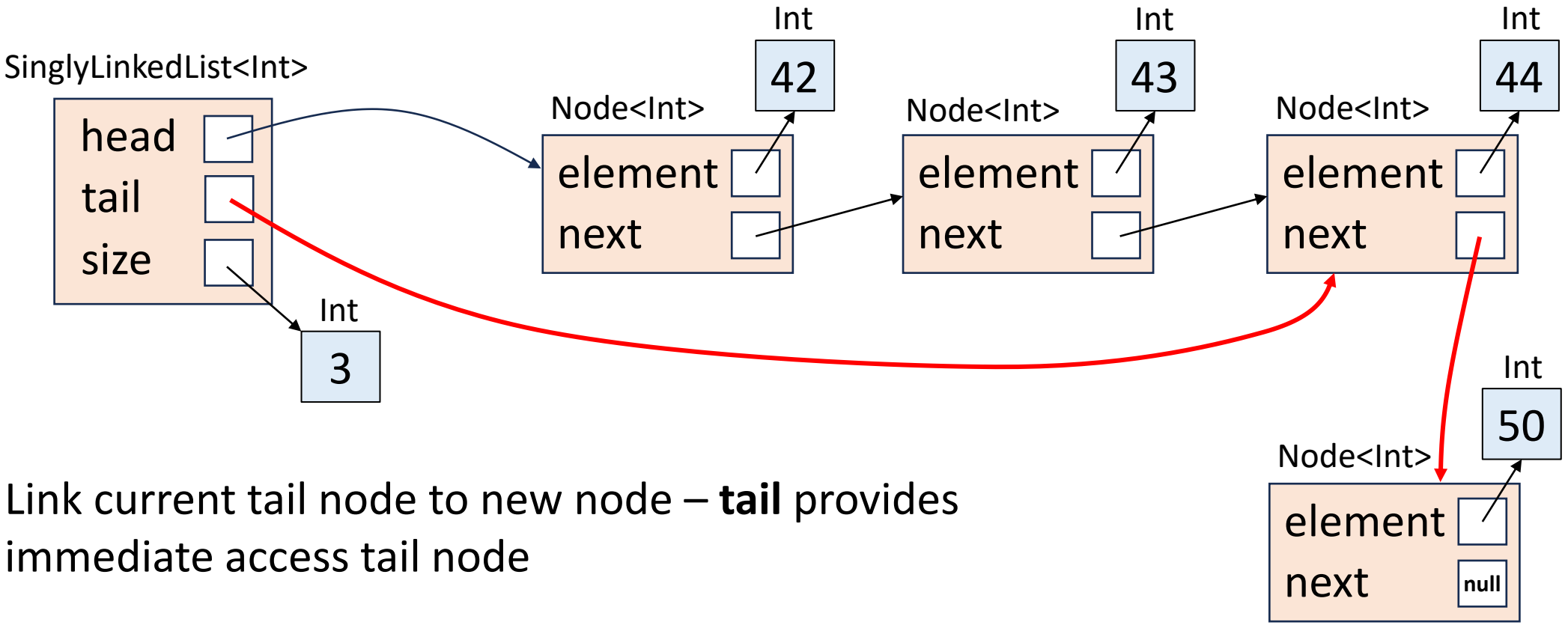


Create a new tail node for element 50

Adding an element to the end of the list

`list.add(50)`

Add the integer 50 to the list

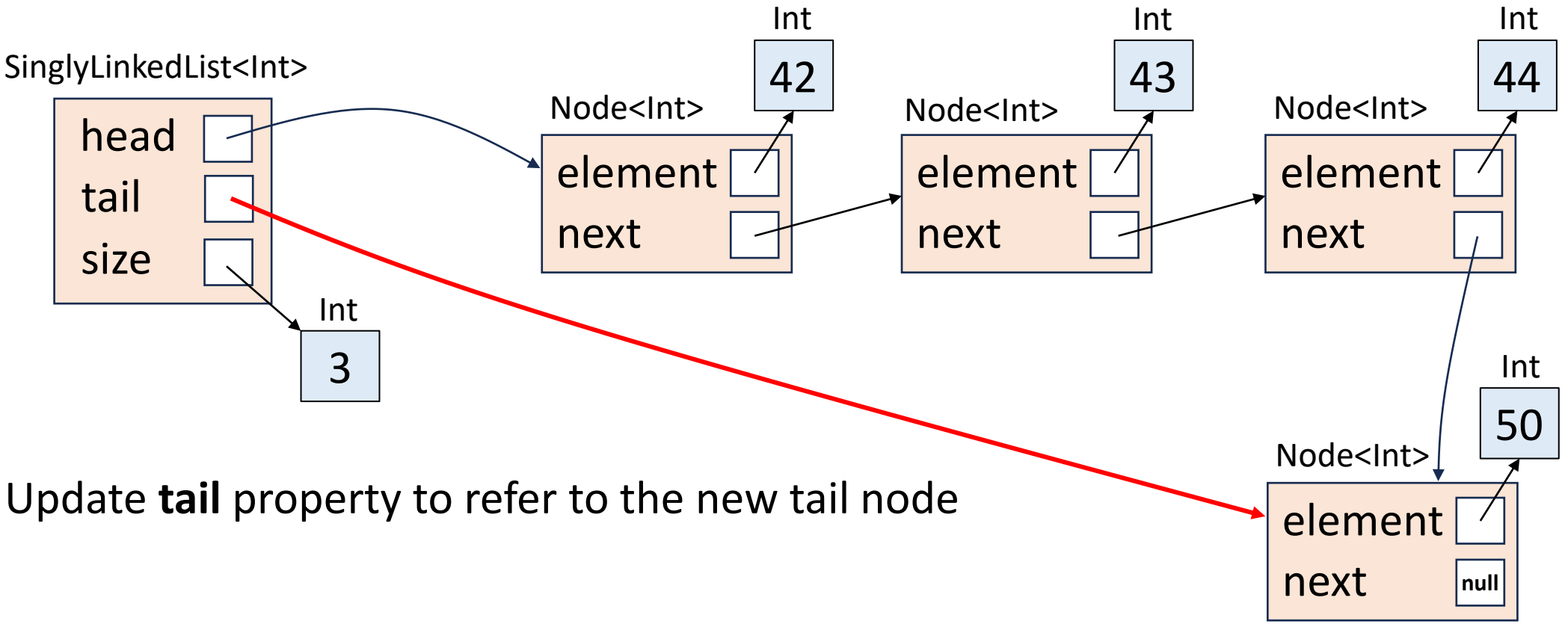


Link current tail node to new node – **tail** provides immediate access tail node

Adding an element to the end of the list

`list.add(50)`

Add the integer 50 to the list

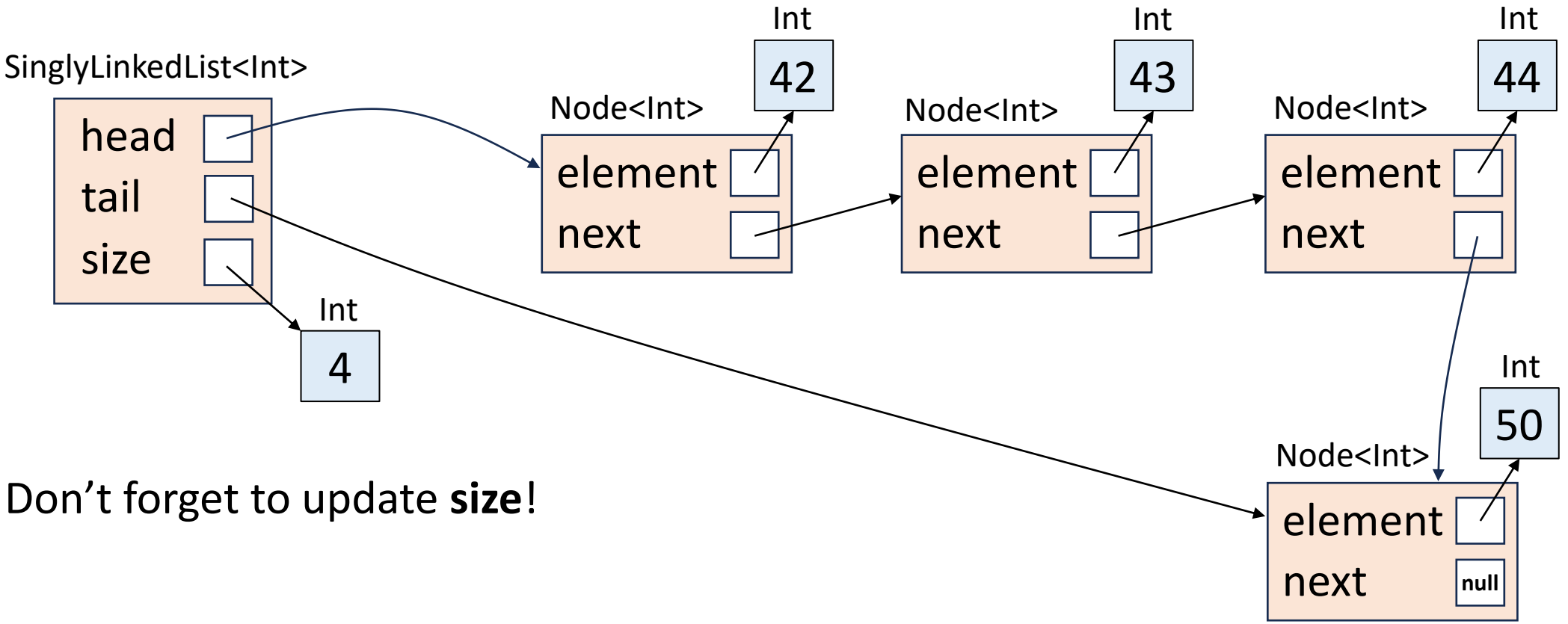


Update **tail** property to refer to the new tail node

Adding an element to the end of the list

`list.add(50)`

Add the integer 50 to the list



Don’t forget to update **size**!

An implementation of `add`

Exploits default parameter: new node's successor is **null** by default

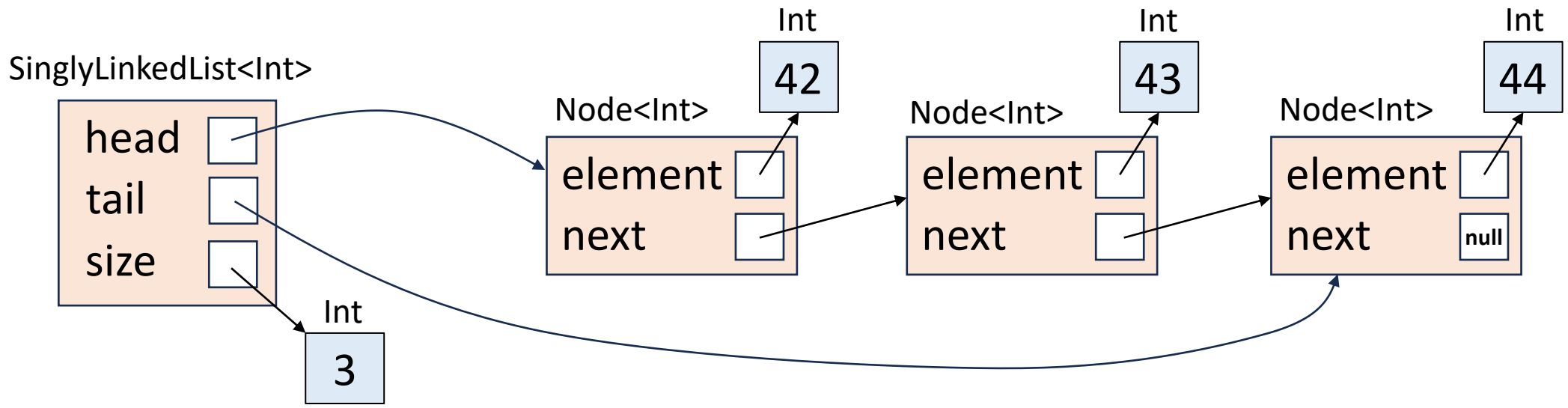
```
fun add(element: T) {
    size++
    val newNode = Node(element)
    if (head == null) {
        head = newNode
        tail = newNode
        return
    }
    tail!!.next = newNode
    tail = newNode
}
```

Special logic needed if the list is empty

Exercise: what invariant means we can be sure **tail** is not null here?

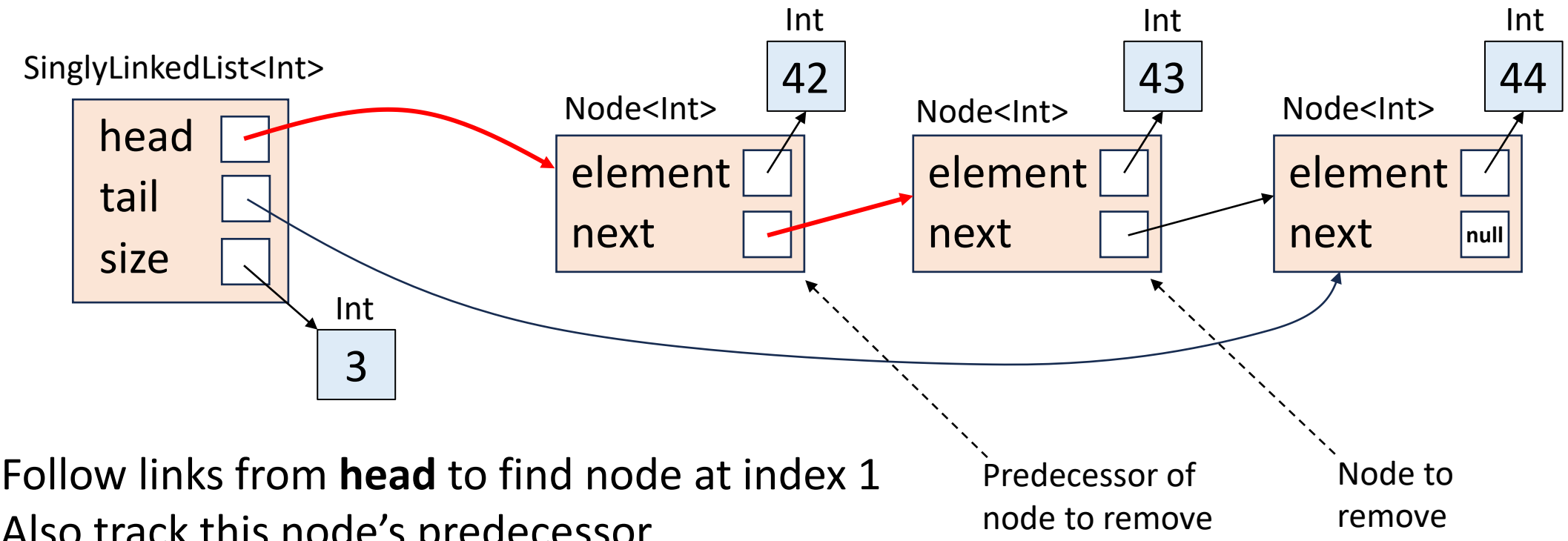
Removing an element

`list.removeAt(1)` Remove the element at index 1



Removing an element

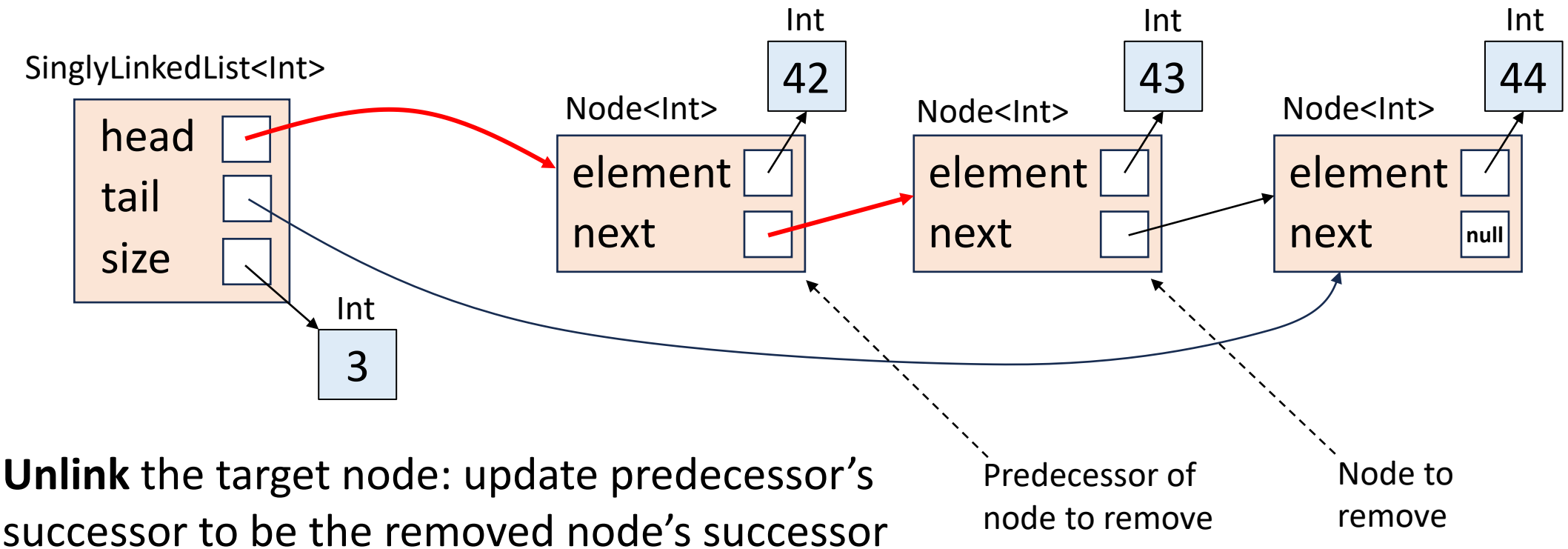
`list.removeAt(1)` Remove the element at index 1



Follow links from **head** to find node at index 1
Also track this node's predecessor

Removing an element

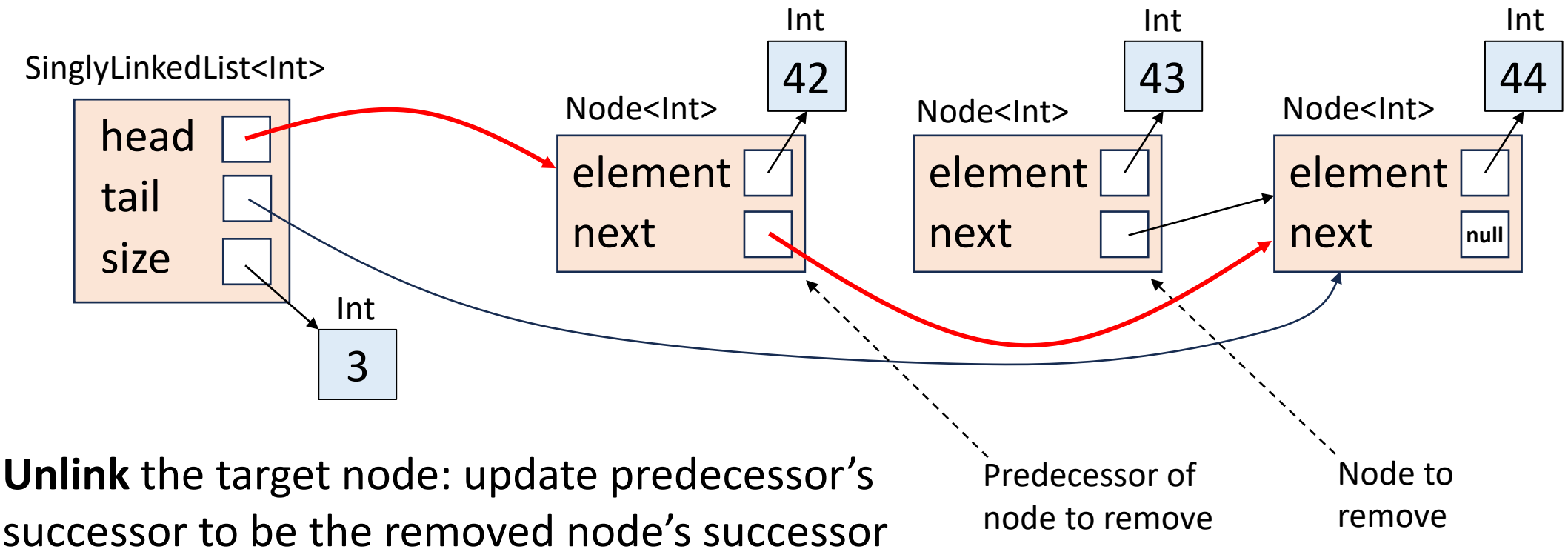
`list.removeAt(1)` Remove the element at index 1



Unlink the target node: update predecessor's successor to be the removed node's successor

Removing an element

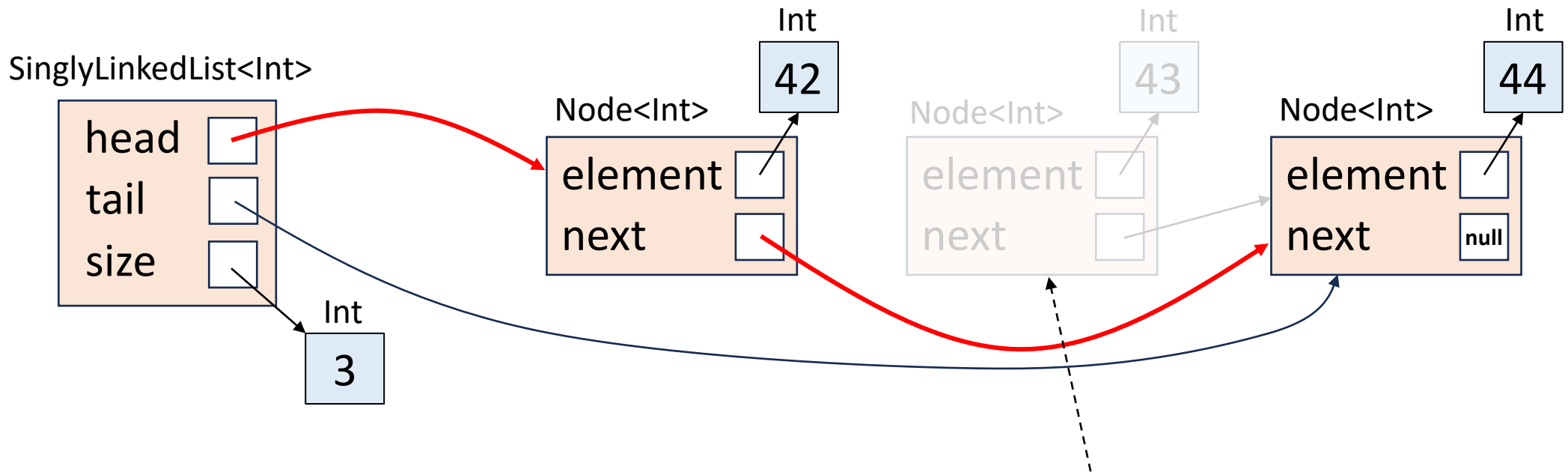
`list.removeAt(1)` Remove the element at index 1



Unlink the target node: update predecessor's successor to be the removed node's successor

Removing an element

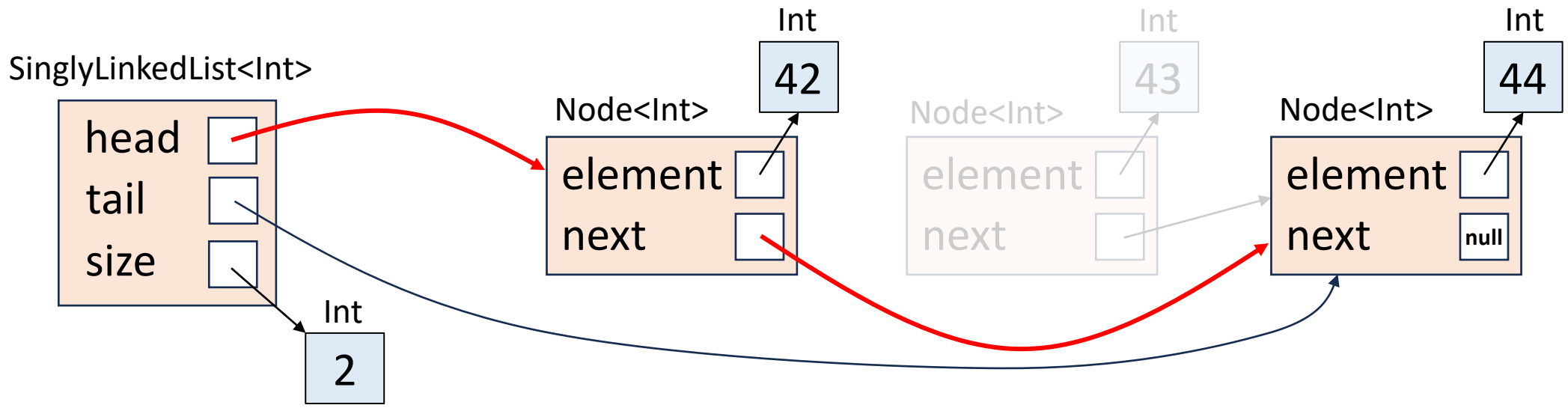
`list.removeAt(1)` Remove the element at index 1



This node is no longer part of the list – in due course it will be **garbage-collected** (more on that later)

Removing an element

`list.removeAt(1)` Remove the element at index 1



Don’t forget to update the size of the list!

An implementation of `removeAt`

```
fun removeAt(index: Int): T {
    if (index !in 0..<size) {
        throw IndexOutOfBoundsException()
    }
    val (previous: Node<T>?, current: Node<T>?) = traverseTo(index)
    val result = current!!.element
    unlink(previous, current)
    return result
}
```

```
private fun traverseTo(index: Int):
Pair<Node<T>?, Node<T>?> {
    var previous: Node<T>? = null
    var current: Node<T>? = head
    for (i in 0..<index) {
        previous = current
        current = current!!.next
    }
    return Pair(previous, current)
}
```

```
private fun unlink(previous: Node<T>?,
                   current: Node<T>) {
    if (previous == null) {
        head = current.next
    } else {
        previous.next = current.next
    }
    if (current == tail) {
        tail = previous
    }
    size--
}
```

Exercise: complete the `SinglyLinkedList<T>` class

- Provide the full set of methods that were available for `FixedCapacityList<T>` and `ResizingArrayList<T>`

Properties of singly-linked list

How efficient is it to get the element at index *i*?

Not efficient: we need to follow *i* links

Properties of resizing array-based list

How efficient is it to add an element to the end of the list?

Efficient: link new node to previous tail, update tail property

Properties of resizing array-based list

How efficient is it to add an element earlier in the list?

Efficient if we have a reference to predecessor of insertion point:
just link in the new node

Inefficient if all we know is the index *i* of insertion – need to chase *i* links

Array-based lists vs. linked lists vs. other data structures

- We will come back to these and other structures later on
- We will study their properties slightly more formally

Next: let's focus on what these lists have in common and how we can write code that works regardless of which list implementation we use