Linked List

Lecture Question

Task: Write an append method for our linked list

- Write a method in the week11.LinkedListNode class (from the repo) named append that:
  - Is initially called on the head node of a linked list
  - Takes a value of type A as its only parameter
  - Returns Unit
  - Creates a new node containing the input value and appends it to the end of the linked list

Recall - Array

- Sequential
  - One continuous block of memory
  - Random access based on memory address
    - address = first_address + (element_size * index)
- Fixed Size
  - Since memory adjacent to the block may be used
  - Efficient when you know how many elements you'll need to store

Array

Program Stack	
Main Frame	name:myArray, value:1503

Program Heap	
1503	myArray[0]
...	myArray[1]
...	myArray[2]
	myArray[3]
	...
	[used by another program]

- Arrays are stored on the heap
- Pointer to index 0 goes on the stack
- add index * sizeOfElement to 1503 to find each element
  - This is called random access

Recall - Linked List

- Sequential
  - Spread across memory
  - Each element knows the memory address of the next element
    - Follow the addresses to find each element
- Variable Size
  - Store new element anywhere in memory
  - Add store the memory address in the last element
    - or new element stores address of first element

Linked List

Program Stack	
Main Frame	name:myList, value:506

Program Heap	
506	name:value, value:5
...	name:next, value:795

Program Heap	
795	name:value, value:3
...	name:next, value:416

Program Heap	
416	name:value, value:1
...	name:next, value:null

- myList stores a list containing: [5,3,1]
- Last link stores null
  - We say the list is "null terminated"
  - When we read a value of null we know we reached the end of the list

Linked List

Program Stack	
Main Frame	name:myList, value:506

Program Heap	
506	name:value, value:5
...	name:next, value:795

Program Heap	
795	name:value, value:3
...	name:next, value:416

Program Heap	
416	name:value, value:1
...	name:next, value:null

```
class LinkedListNode[A](var value: A, var next: LinkedListNode[A]) {
}
```

```
var myList: LinkedListNode[Int] = new LinkedListNode[Int](1, null)
myList = new LinkedListNode[Int](3, myList)
myList = new LinkedListNode[Int](5, myList)
```

- We create our own linked list class by defining a node
  - A node represents one "link" in the list
- The list itself is a reference to the first/head node

Linked List Algorithms

- We know the structure of a linked list
- How do we operate on these lists?
- We would like to:
  - Find the size of a list
  - Print all the elements of a list
  - Access elements by location
  - Add/remove elements
  - Find a specific value

Size

- Navigate through the entire list until the next reference is null
  - Count the number of nodes visited
- Could use loop. Recursive example shown

```
def size(): Int = {
  if(this.next == null){
    1
  }else{
    this.next.size() + 1
  }
}
```

To String

- Same as size, but accumulate the values as strings instead of counting the number of nodes
- Recursion makes it easier to manage our commas
  - ", " is only appended if it's not the last element

```
override def toString: String = {
  if (this.next == null) {
    this.value.toString
  }else {
    this.value.toString + ", " + this.next.toString
  }
}
```

Access Element by Location

- Simulates array access
- Take an "index" and advance through the list that many times
- MUCH slower than array access
  - Calls next n times - O(n) runtime
  - ex. apply(4) is the same as this.next.next.next.next

```
def apply(i: Int): LinkedListNode[A] = {
  if (i == 0) {
    this
  } else {
    this.next.apply(i - 1)
  }
}
```

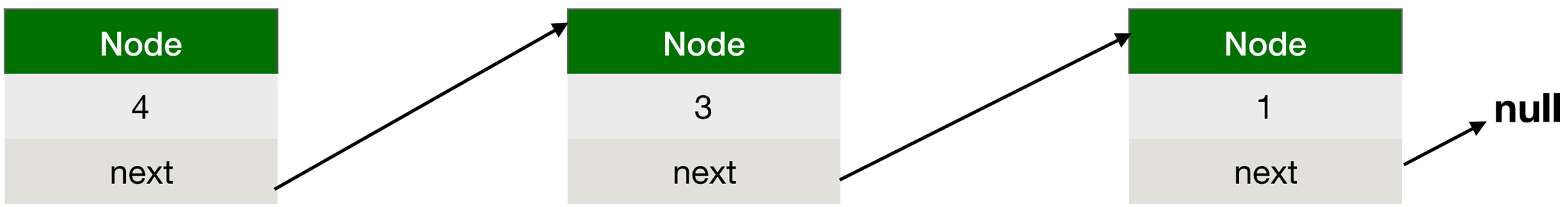
Access Element by Location

- Simulates array access
- Take an "index" and advance through the list that many times
- MUCH slower than array access
  - Calls next n times - O(n) runtime
  - ex. apply(4) is the same as this.next.next.next.next

```
def apply(i: Int): LinkedListNode[A] = {
  if (i == 0) {
    this
  } else {
    this.next.apply(i - 1)
  }
}
```

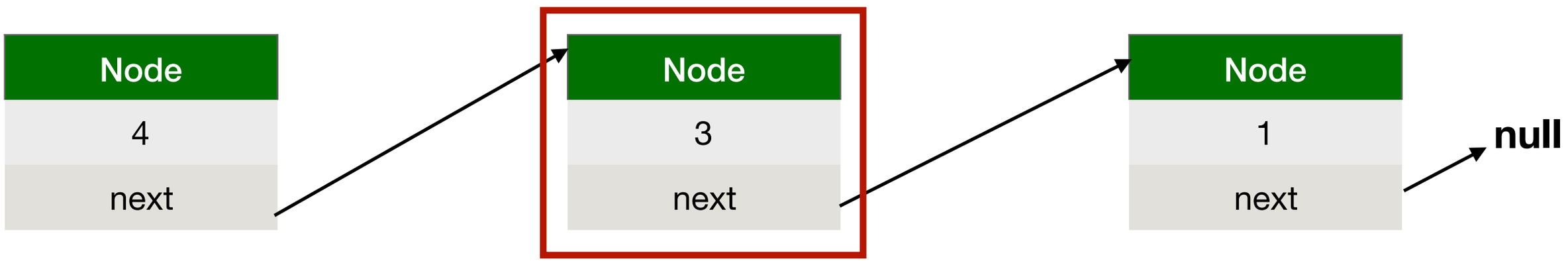
Add an Element

- To add an element we first need a reference to the node before the location of the new element
- Update the next reference of this node
- Want to add 2 in this list after 3



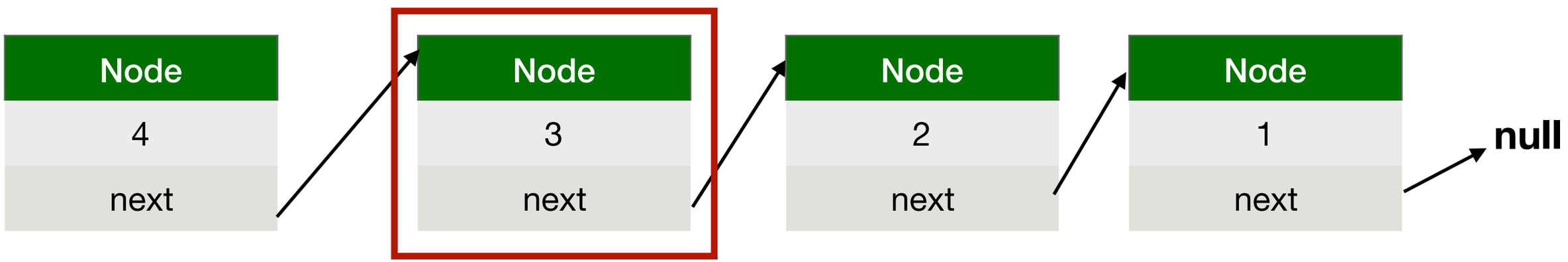
Add an Element

- Need reference to the node containing 3



Add an Element

- Need reference to the node containing 3
- Create the new node with next equal to this node's next
- This node's next is set to the new node



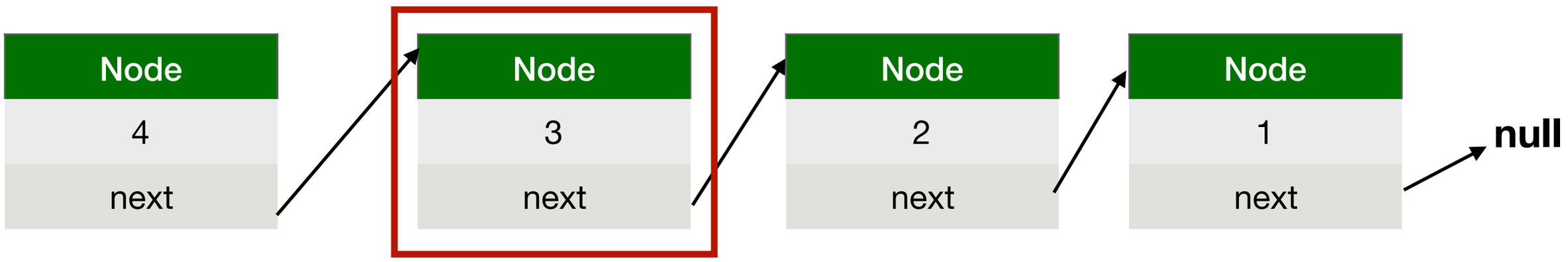
Add an Element

- Need reference to the node containing 3
- Create the new node with next equal to this node's next
- This node's next is set to the new node

```
def insert(element: A): Unit = {
  this.next = new LinkedListNode[A](element, this.next)
}
```

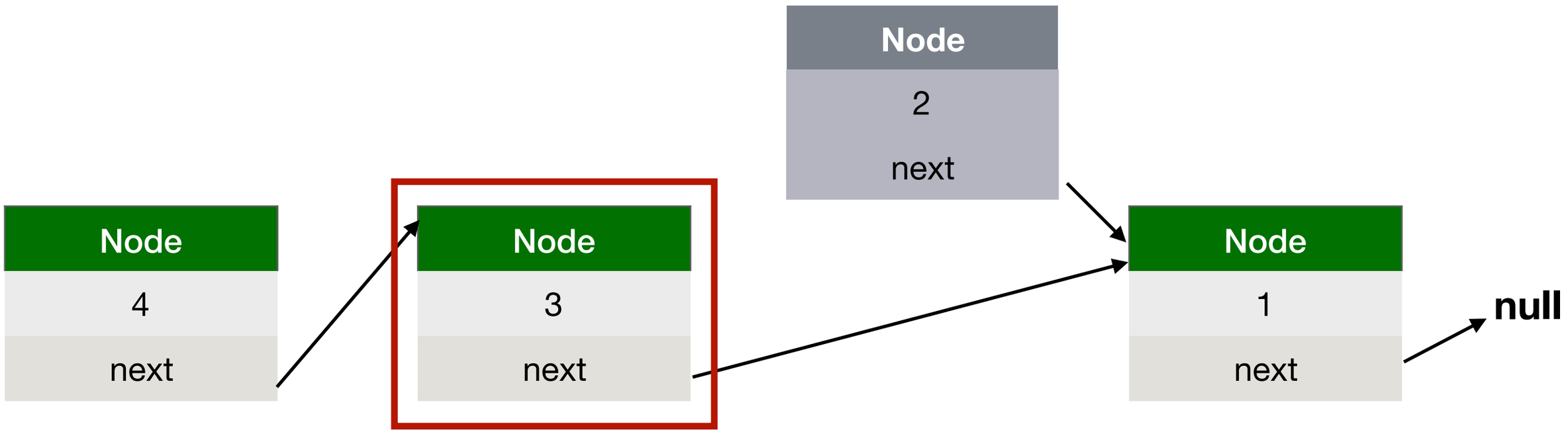
Delete a Node

- Want to delete the node containing 2
- Need a reference to the previous node



Delete a Node

- Update that node's next to bypass the deleted node
  - Don't have to update deleted node
  - The list no longer refers to this node



Delete a Node

- Update that node's next to bypass the deleted node
  - Don't have to update deleted node
  - The list no longer refers to this node

```
def deleteAfter(): Unit = {
  this.next = this.next.next
}
```

Lecture Question

Task: Write an append method for our linked list

- Write a method in the week11.LinkedListNode class (from the repo) named append that:
  - Is initially called on the head node of a linked list
  - Takes a value of type A as its only parameter
  - Returns Unit
  - Creates a new node containing the input value and appends it to the end of the linked list