Stack and Queue

Stack and Queue

- Data structures with specific purposes
  - Restricted features
- All operations are very efficient
  - Inefficient operations are not allowed
- We'll see a stack and queue using linked lists
- *Scala has builtin Stack and Queue classes

Stack

- LIFO
  - Last in First out
  - The last element pushed onto the stack is the first element to be popped off the stack
- Only the element on the top of the stack can be accessed

Stack Methods

- Push
    - Add an element to the top of the stack
- Pop
    - Remove the top element of the stack

Stack Implementation

- Implement a Stack class by wrapping a linked list
- Stack uses the linked list and adapts its methods to implement push and pop

```
class Stack[A] {

  var top: LinkedListNode[A] = null

  def push(a: A): Unit = {
    this.top = new LinkedListNode[A](a, this.top)
  }

  def pop(): A = {
    val toReturn = this.top.value
    this.top = this.top.next
    toReturn
  }

}
```

Stack Usage

- Create a new empty Stack
- Call push to add an element to the top
- Call pop to remove an element
- Same exact usage when using Scala's builtin Stack

```scala
val stack = new Stack[Int]()
stack.push(3)
stack.push(7)
stack.push(2)
stack.push(-5)

val element = stack.pop()
```

Stack Usage

- We can use Scala's list as a Stack
    - The preferred way to use the concept of a stack in practice

```
@deprecated("Stack is an inelegant and potentially poorly-performing wrapper around List.
            Use List instead: stack push x becomes x :: list; stack.pop is list.tail.", "2.11.0")
class Stack[+A] protected (protected val elems: List[A])
```

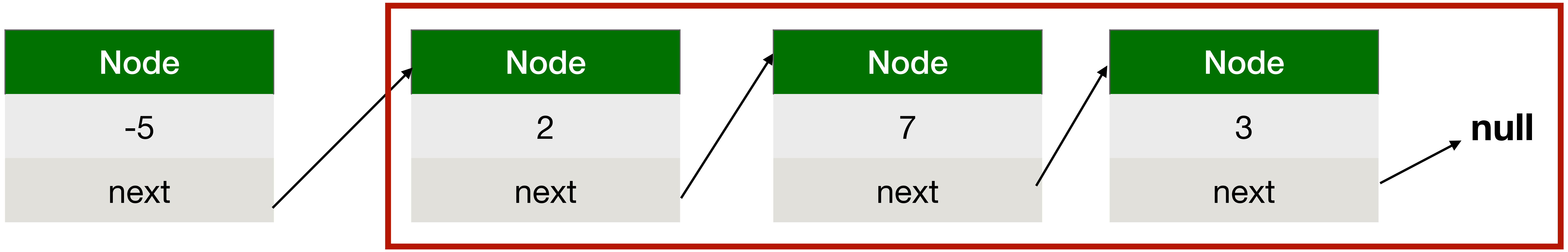
- This is very efficient!
- But wait.. doesn't this create a new list each time an element is pushed or popped since List is immutable?
    - No.. well, kind of

```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.tail
```

Stack Usage

- Before -5 is pushed, the stack is equal to nodes in the red box
- After pushing -5, the red box is unchanged
- A new List **is** returned, but it reuses the old List
    - No need to recreate the entire List

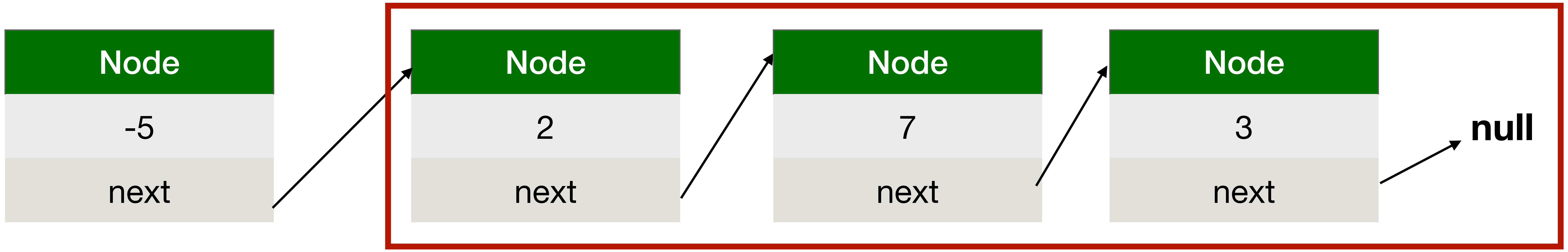
```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.drop(1)
```



Stack Usage

- Same efficiency when -5 is popped
- The red box never changed, but we update the reference stored in the stack variable
- Other parts of the program can share parts of a List without having their changes affect each other

```
var stack = List[Int]()
stack = 3 :: stack
stack = 7 :: stack
stack = 2 :: stack
stack = -5 :: stack

val element = stack.head
stack = stack.drop(1)
```



Queue

- FIFO
  - First in First out
  - The first element enqueued into the queue is the first element to be dequeued out of the queue
- Elements can only be added to the end of the queue
- Only the element at the front of the queue can be accessed

Queue Methods

- Enqueue
  - Add an element to the end of the queue
- Dequeue
  - Remove the front element in the queue

Queue Implementation

- Implement a Queue class by wrapping a linked list
- Queue needs a reference to the first and last element

```
class Queue[A] {

  var front: LinkedListNode[A] = null
  var back: LinkedListNode[A] = null

  def enqueue(a: A): Unit = {
    if (back == null) {
      this.back = new LinkedListNode[A](a, null)
      this.front = this.back
    } else {
      this.back.next = new LinkedListNode[A](a, null)
      this.back = this.back.next
    }
  }

  def dequeue(): A = {
    val toReturn = this.front.value
    this.front = this.front.next
    if(this.front == null){
      this.back = null
    }
    toReturn
  }

}
```

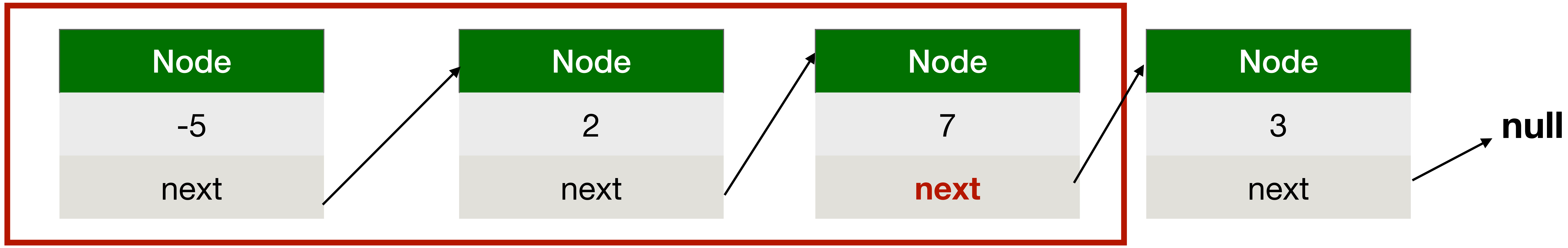
Queue Usage

- Create a new empty Queue
- Call enqueue to add an element to the back
- Call dequeue to remove the element at the front
- Same exact usage when using Scala's builtin Queue
  - [based on mutable List just like our implementation]

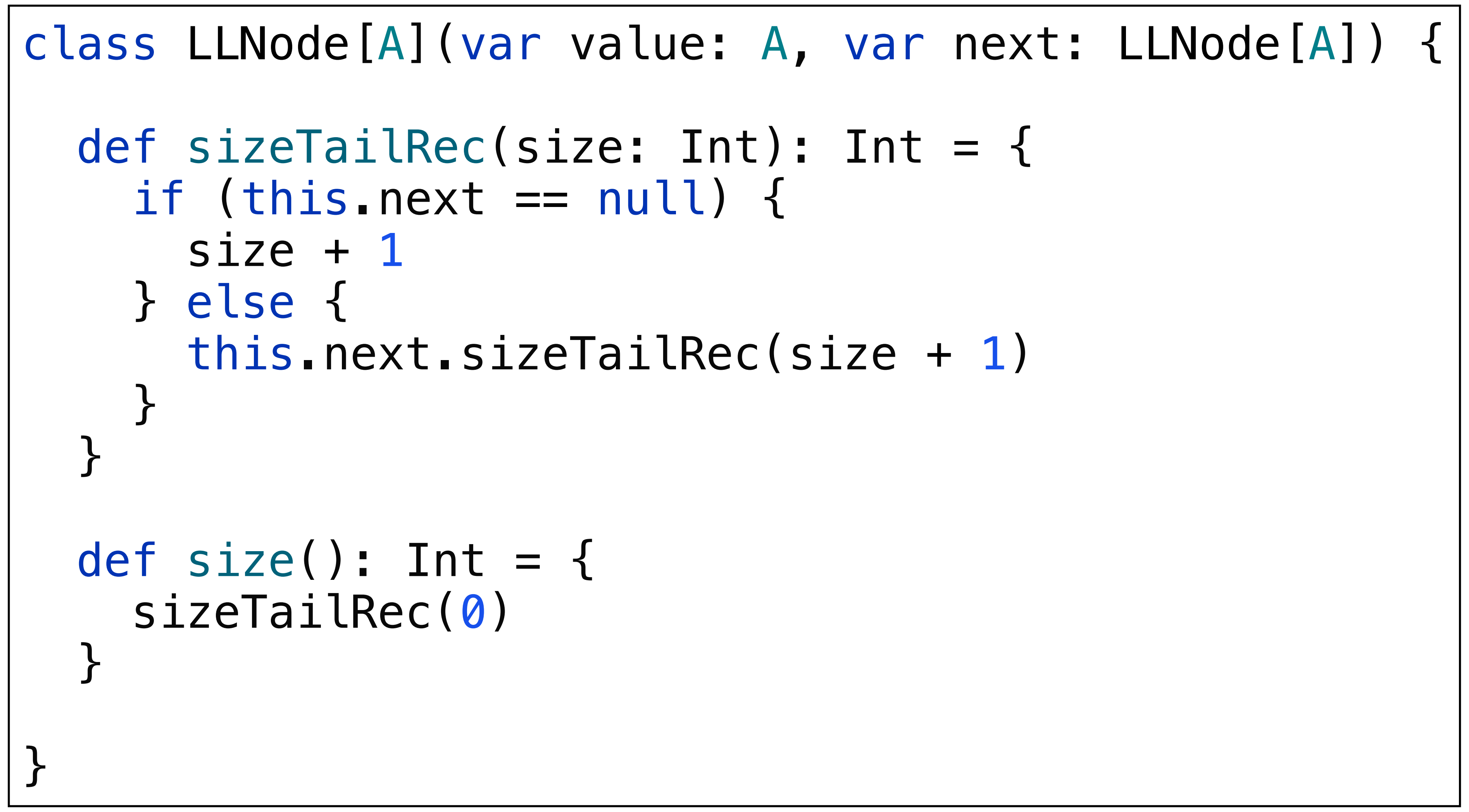
```
val queue = new Queue[Int]()
queue.enqueue(3)
queue.enqueue(7)
queue.enqueue(2)
queue.enqueue(-5)

val element = queue.dequeue()
```

Queue Usage

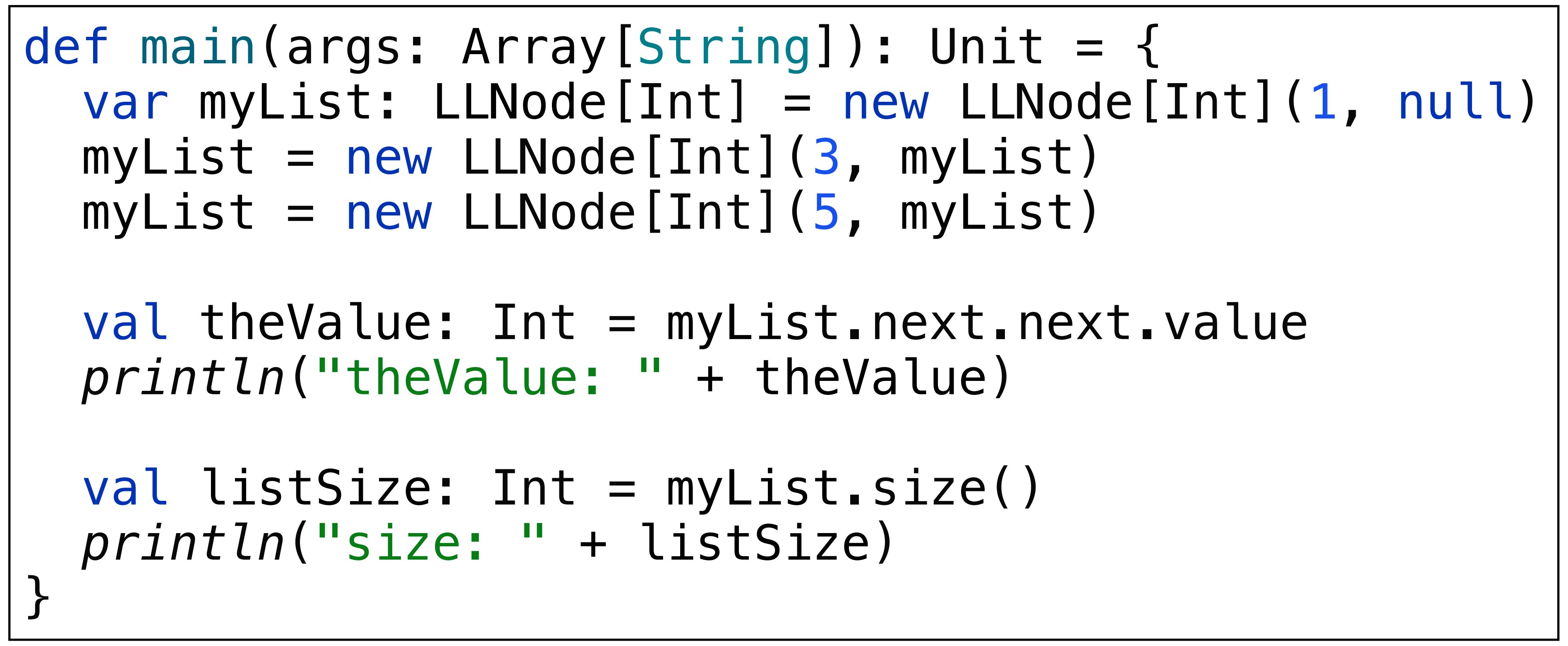
- No efficient way to use an immutable List as a queue
- To enqueue 3 the list in the red box must change
  - The next reference of the node containing 7 has to be updated
  - This List cannot be [should not be] used by other parts of the program since the List is changing



Memory Diagrams



```scala
class LLNode[A](var value: A, var next: LLNode[A]) {

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```



```scala
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Let's walk through this code
- We expect to build the list
  - [5, 3, 1]
- Print the value 1 and the size 3

Stack

Name	Value

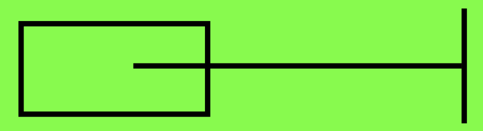
Heap

in/out

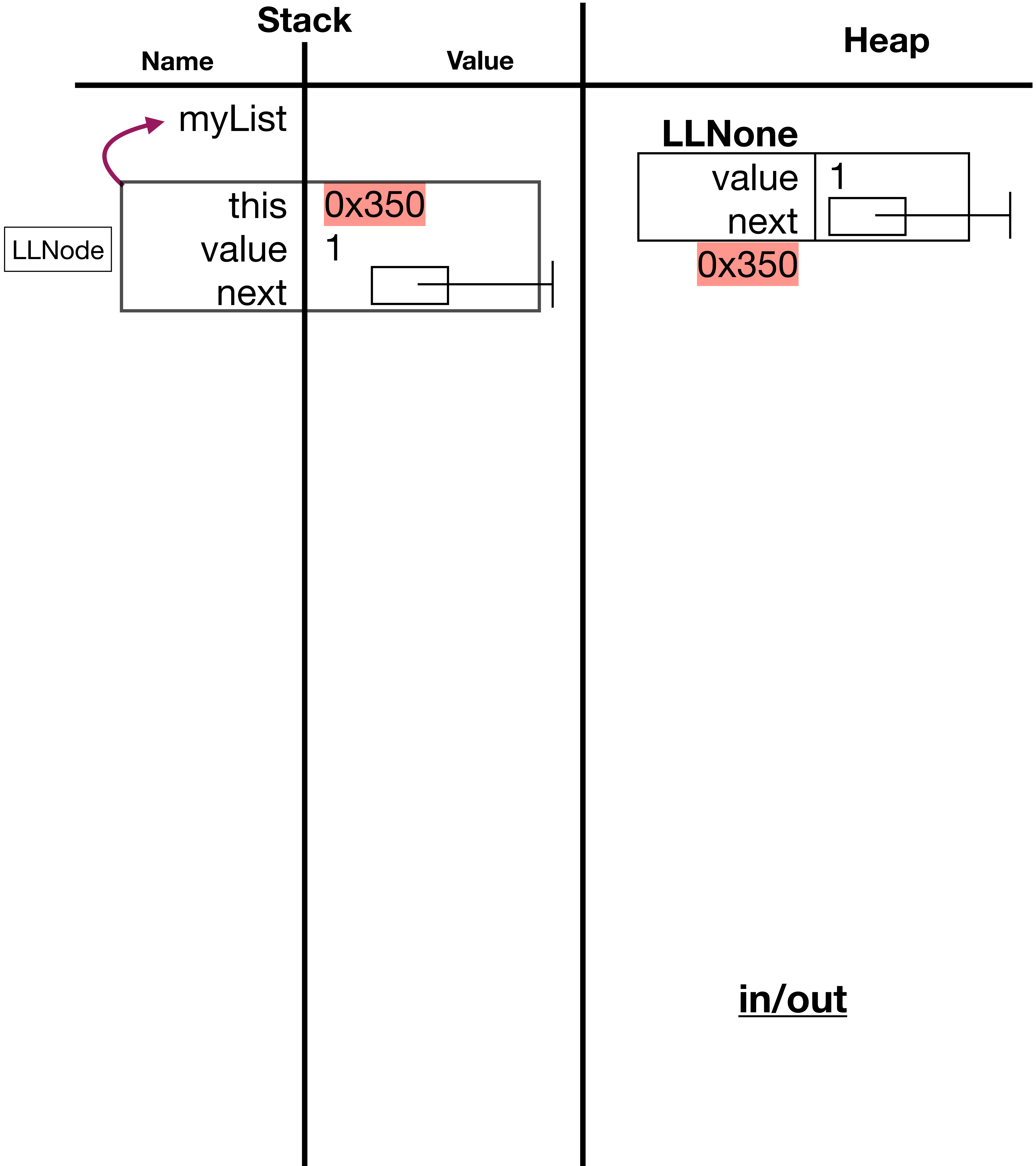
```
class LLNode[A](var value: A, var next: LLNode[A]) {

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Create an object
- To represent null you'll see:



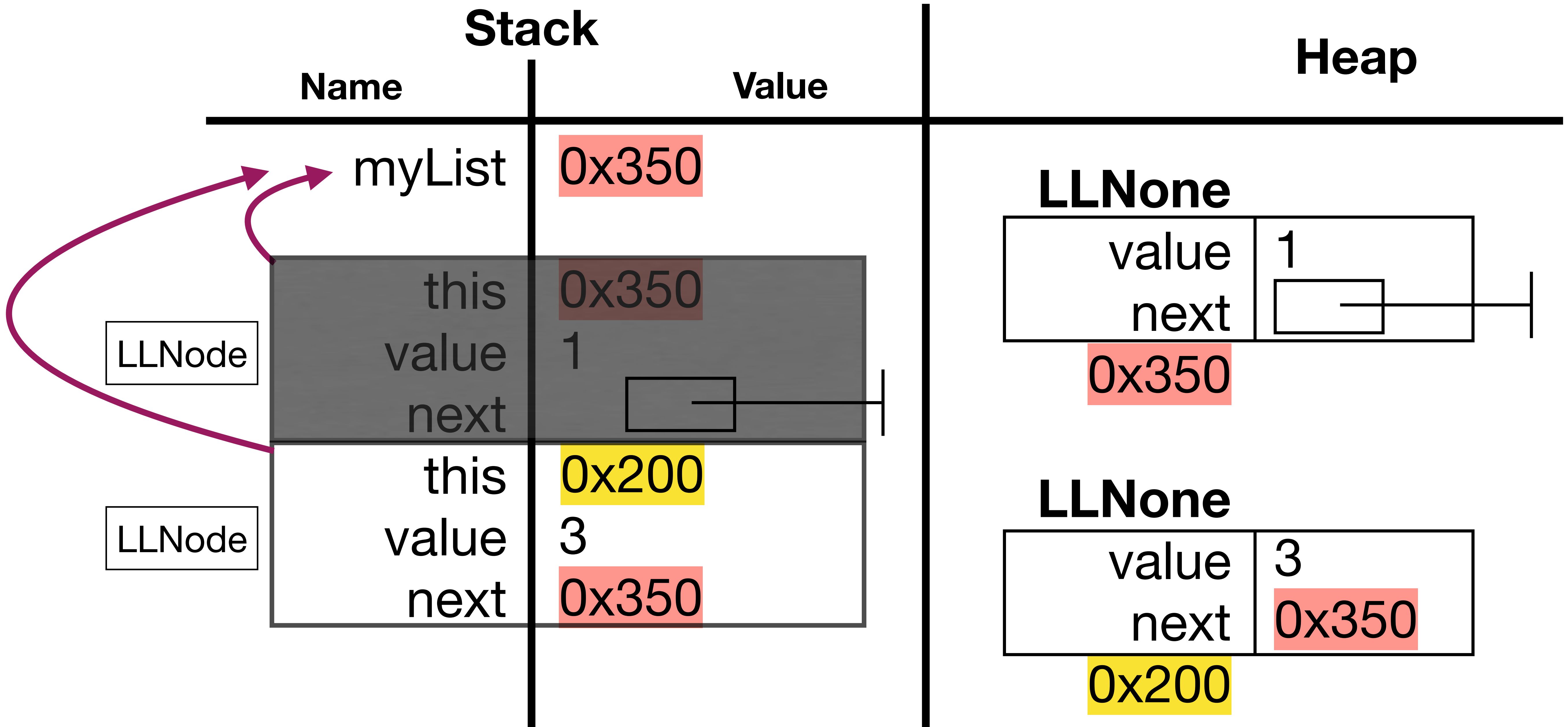
- Thanks, I hate it



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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Call the constructor again
- Pass myList (0x350) as next



Stack

Name	Value
myList	0x350
this	0x350
value	1
next	
this	0x200
value	3
next	0x350

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

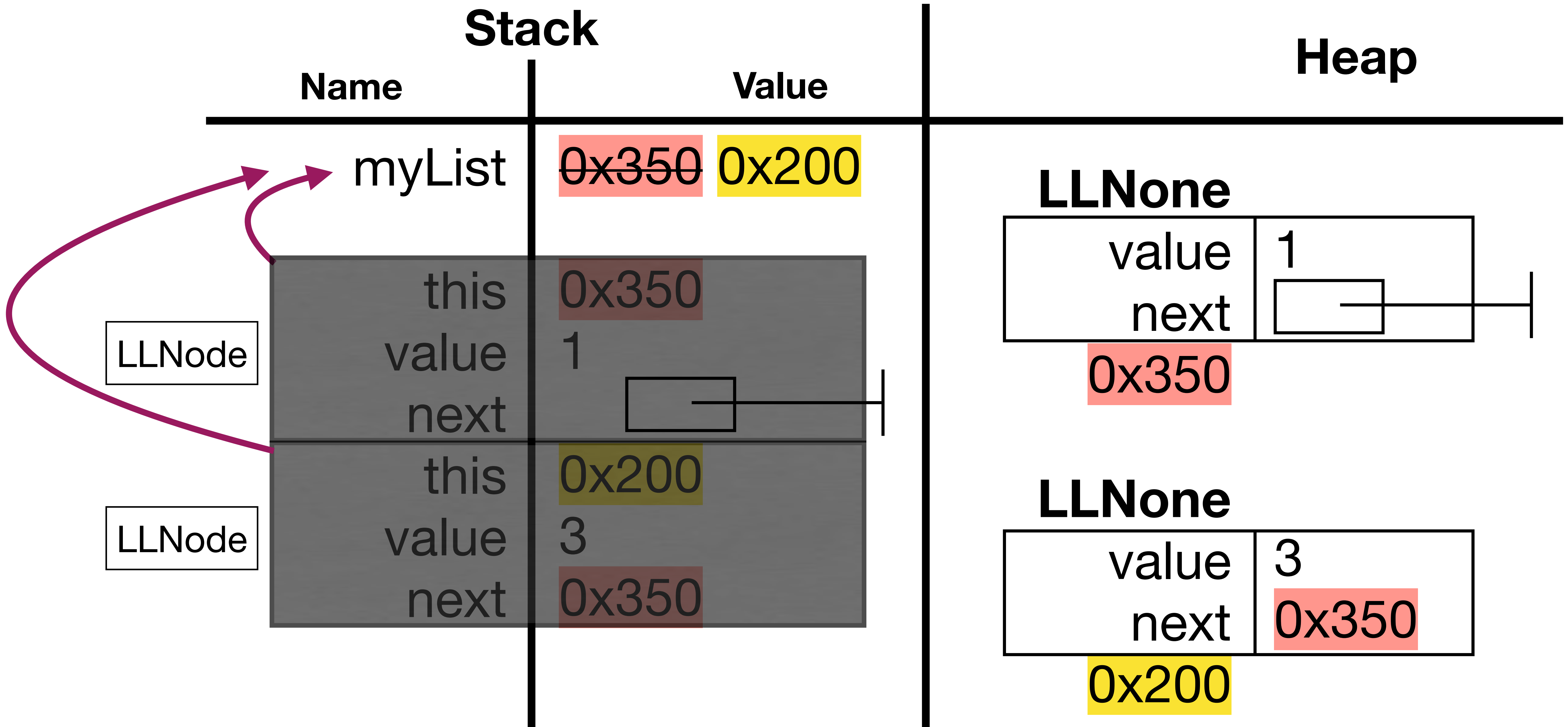
0x200

in/out

```scala
class LLNode[A](var value: A, var next: LLNode[A]) {

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```scala
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Reassign myList to the reference returned by the constructor
- myList now stores 0x200 which has a next of 0x350



Stack

Name	Value
myList	~~0x350~~ 0x200
this	0x350
value	1
next	
this	0x200
value	3
next	0x350

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

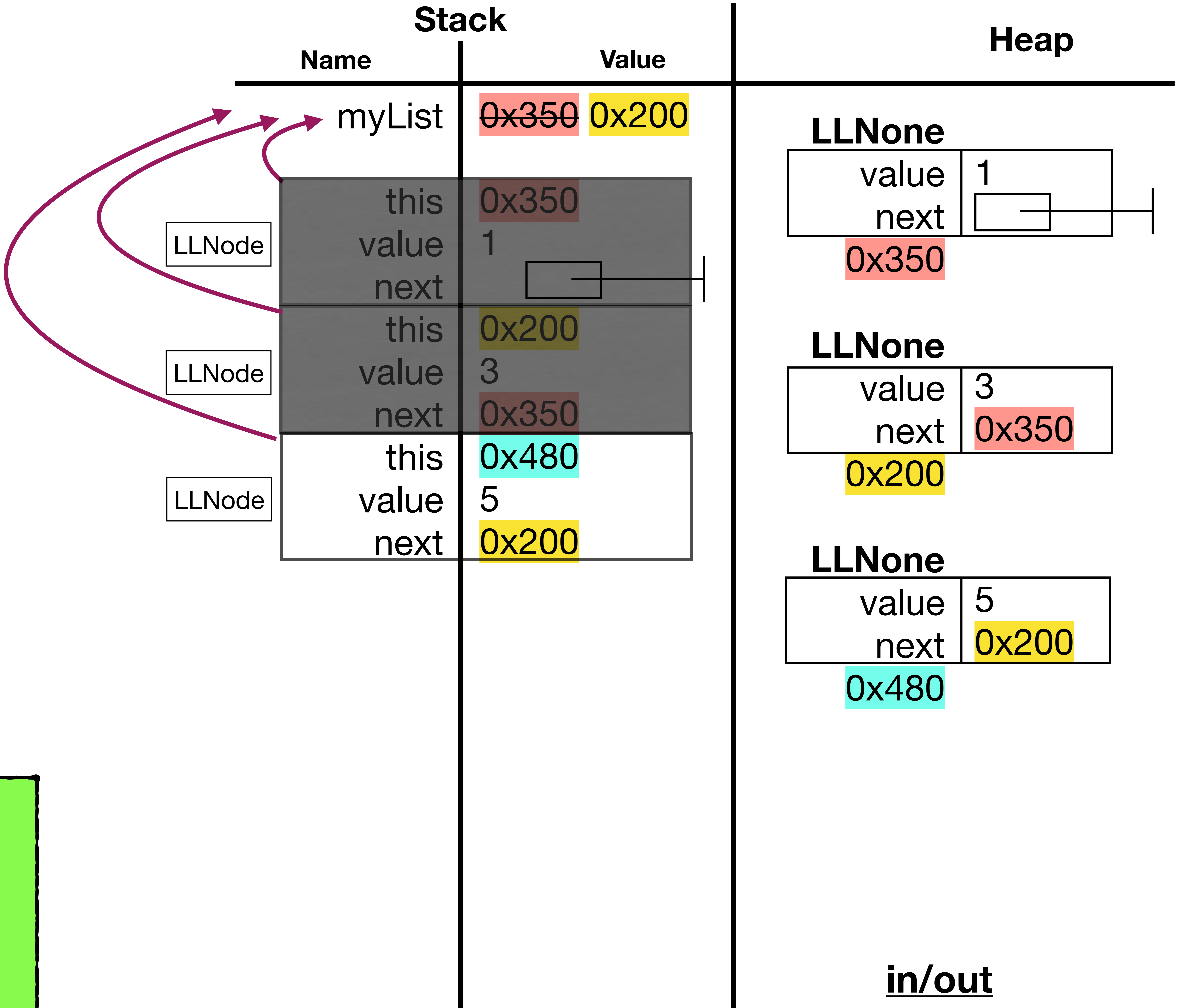
0x200

in/out

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Repeat the process for the node containing 5



Stack

Name	Value
myList	~~0x350~~ 0x200
LLNode this	0x350
value	1
next	
LLNode this	0x200
value	3
next	0x350
LLNode this	0x480
value	5
next	0x200

Heap

LLNone
value 1
next
0x350

LLNone
value 3
next 0x350
0x200

LLNone
value 5
next 0x200
0x480

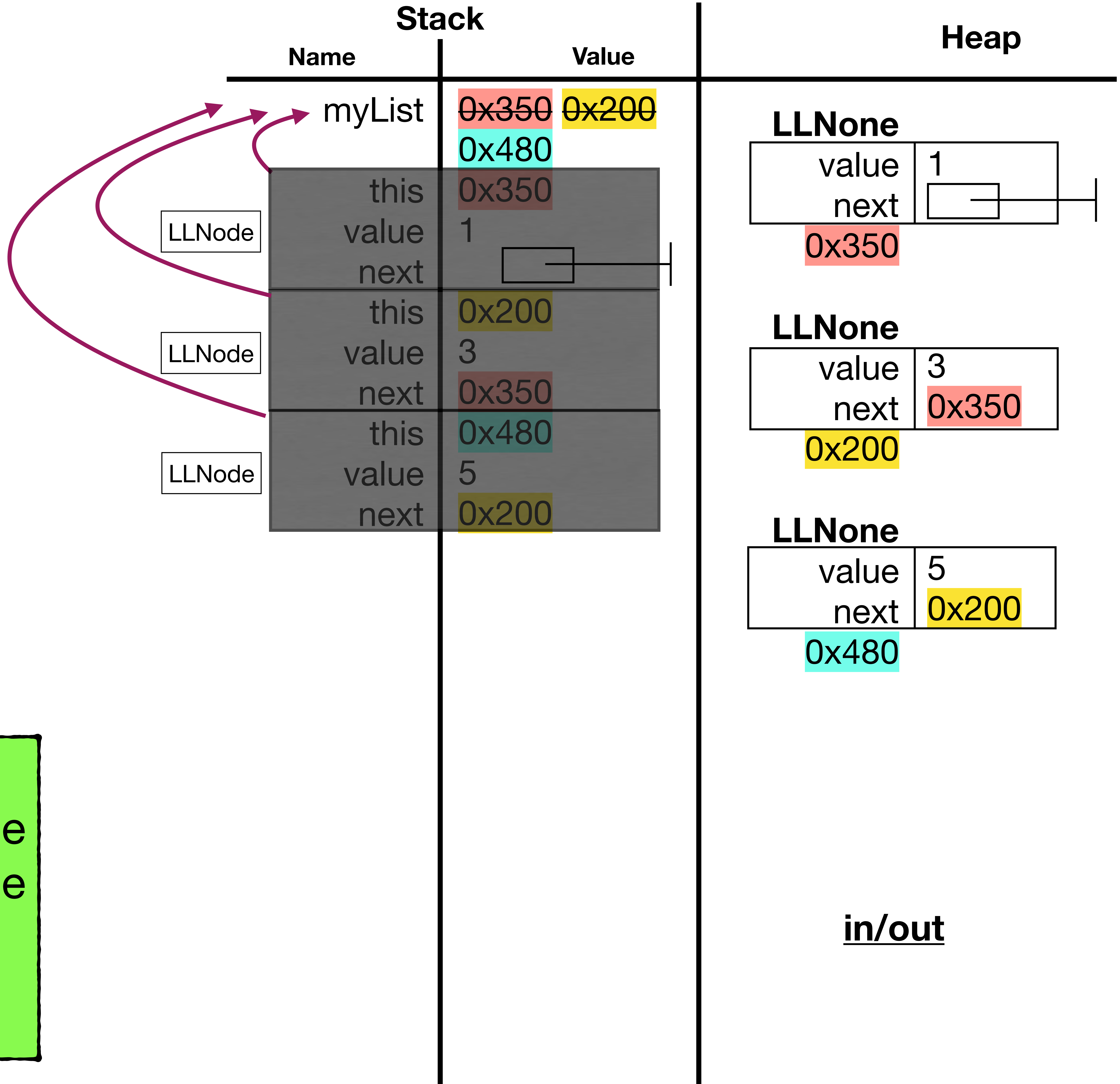
in/out

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- myList now refers to a node containing 5, which refers to a node containing 3, which refers to a node containing 1, which refers to null

- The list is (5, 3, 1)



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

0x200

LLNone

value	5
next	0x200

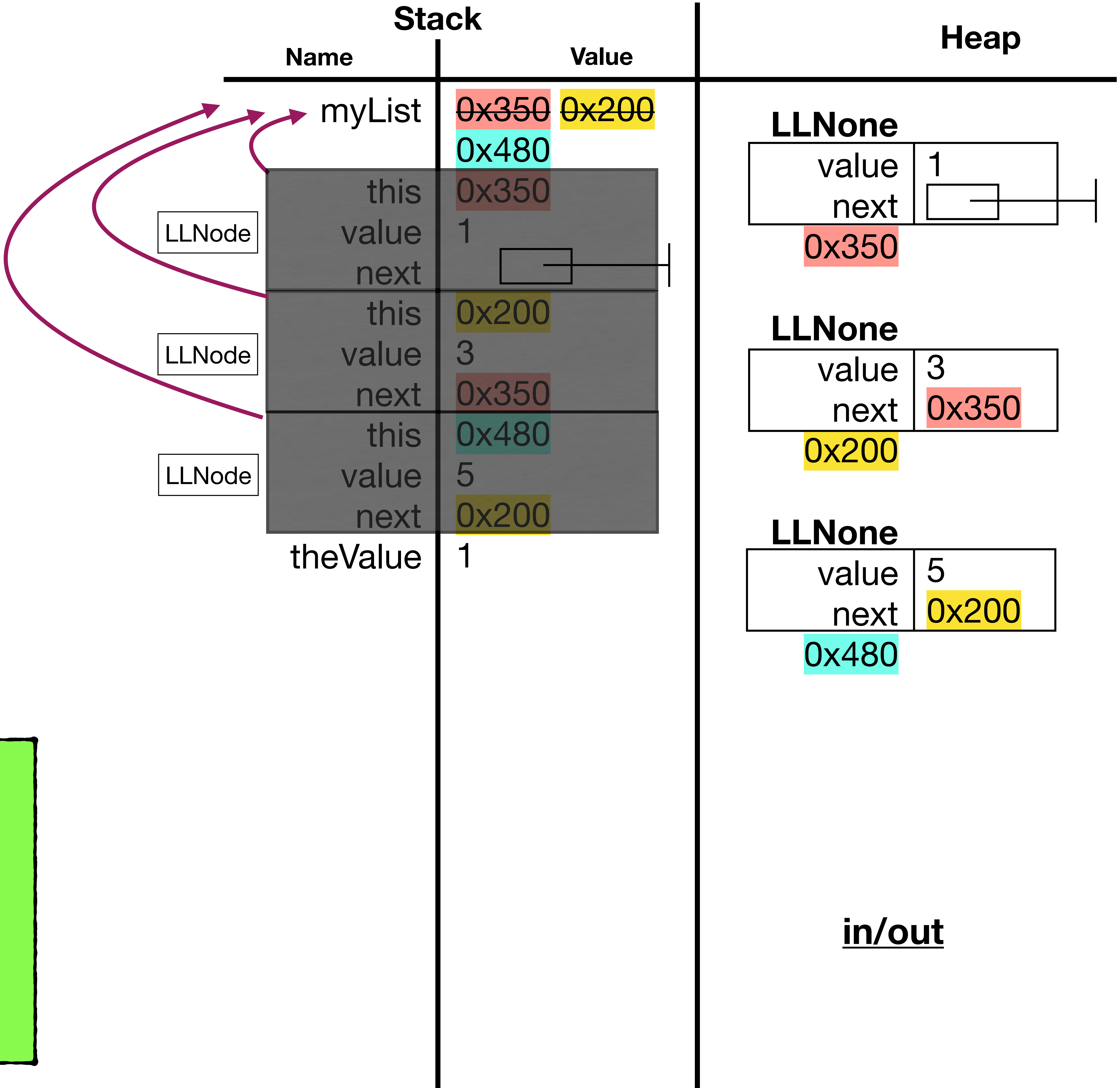
0x480

in/out

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Follow the references in next
- Find the value 1

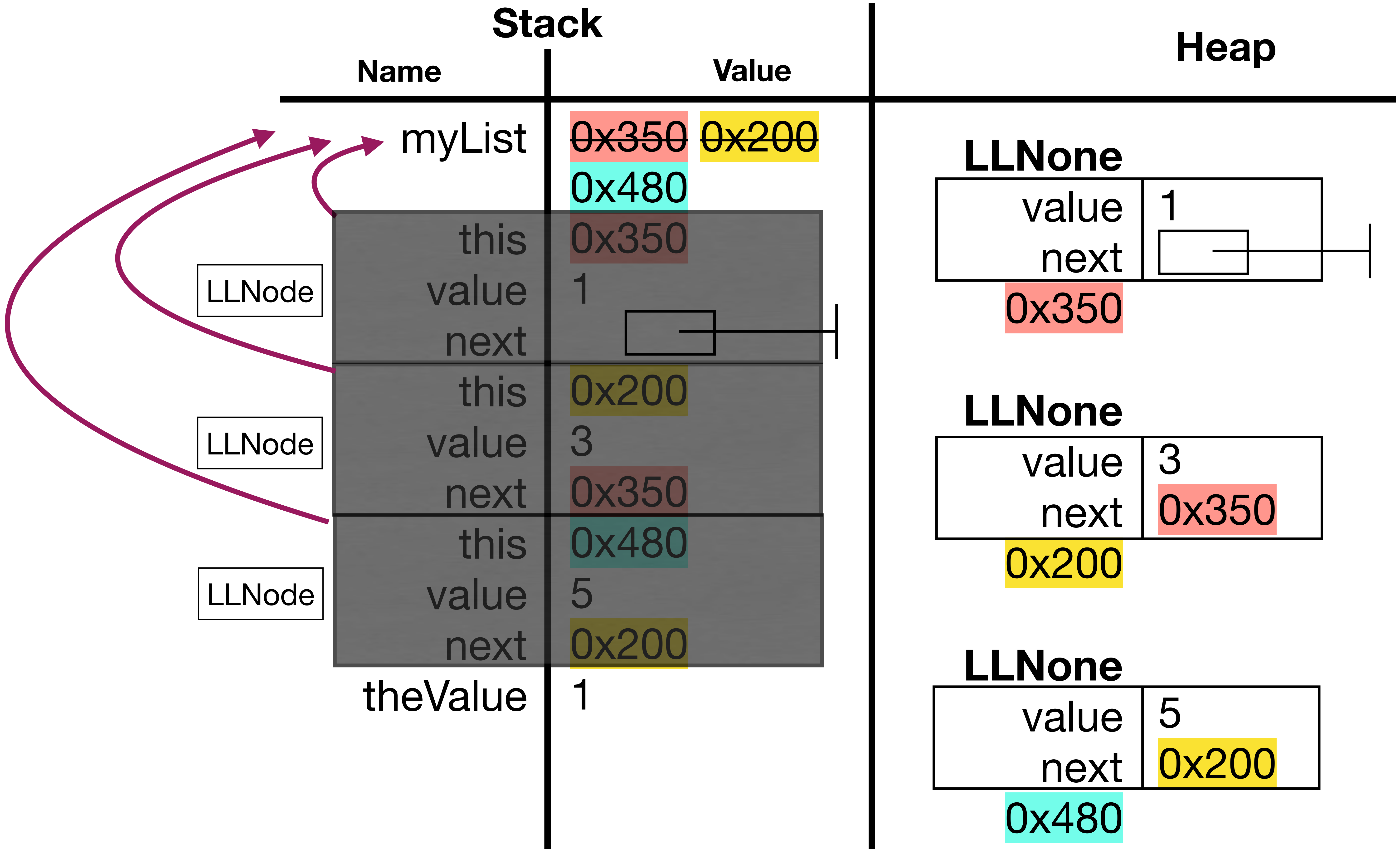


	Stack	
	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Print 1 to the screen



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

0x200

LLNone

value	5
next	0x200

0x480

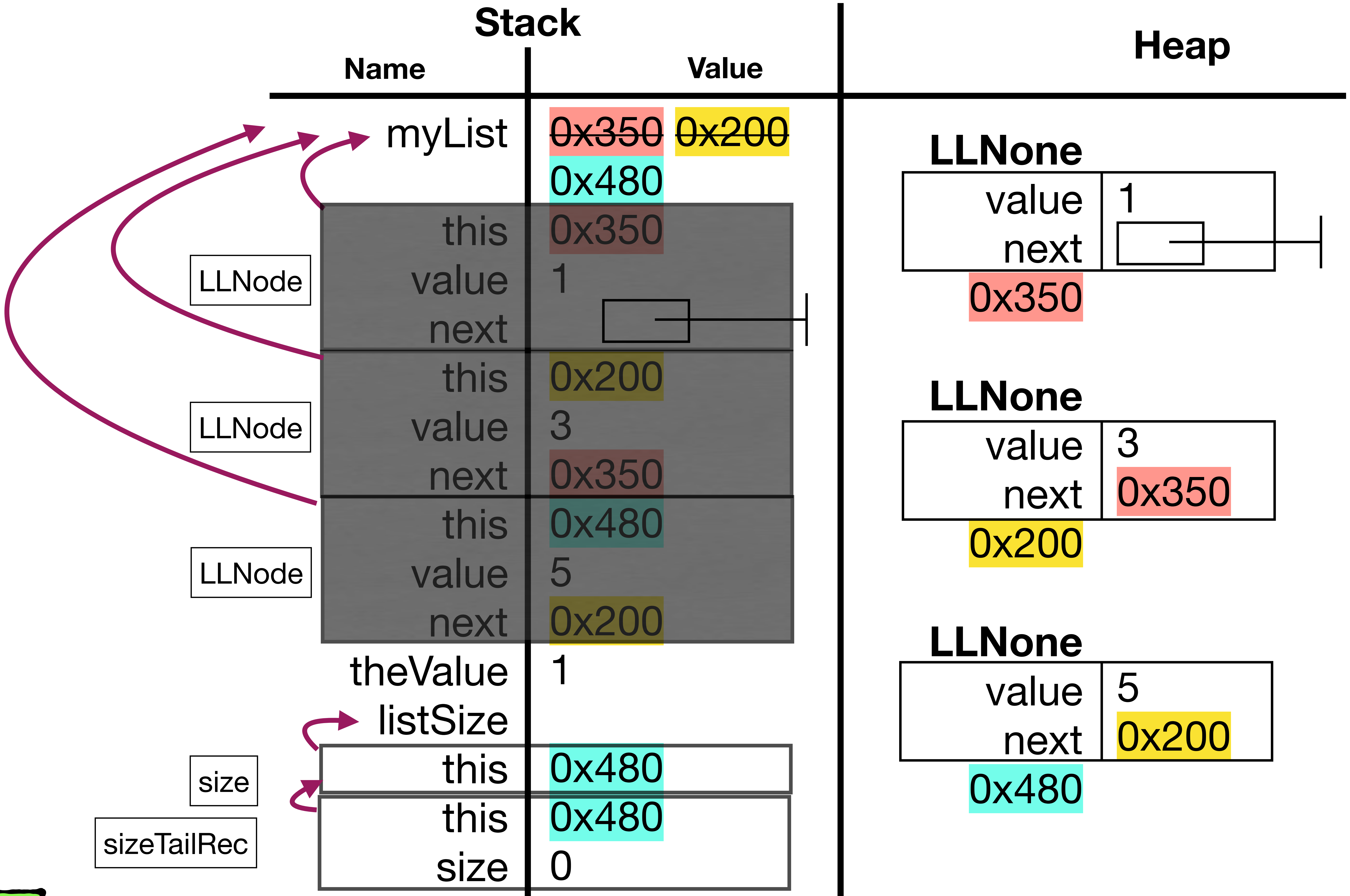
in/out

theValue: 1

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- It's time for recursion!



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1
	listSize	
size	this	0x480
sizeTailRec	this	0x480
	size	0

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

0x200

LLNone

value	5
next	0x200

0x480

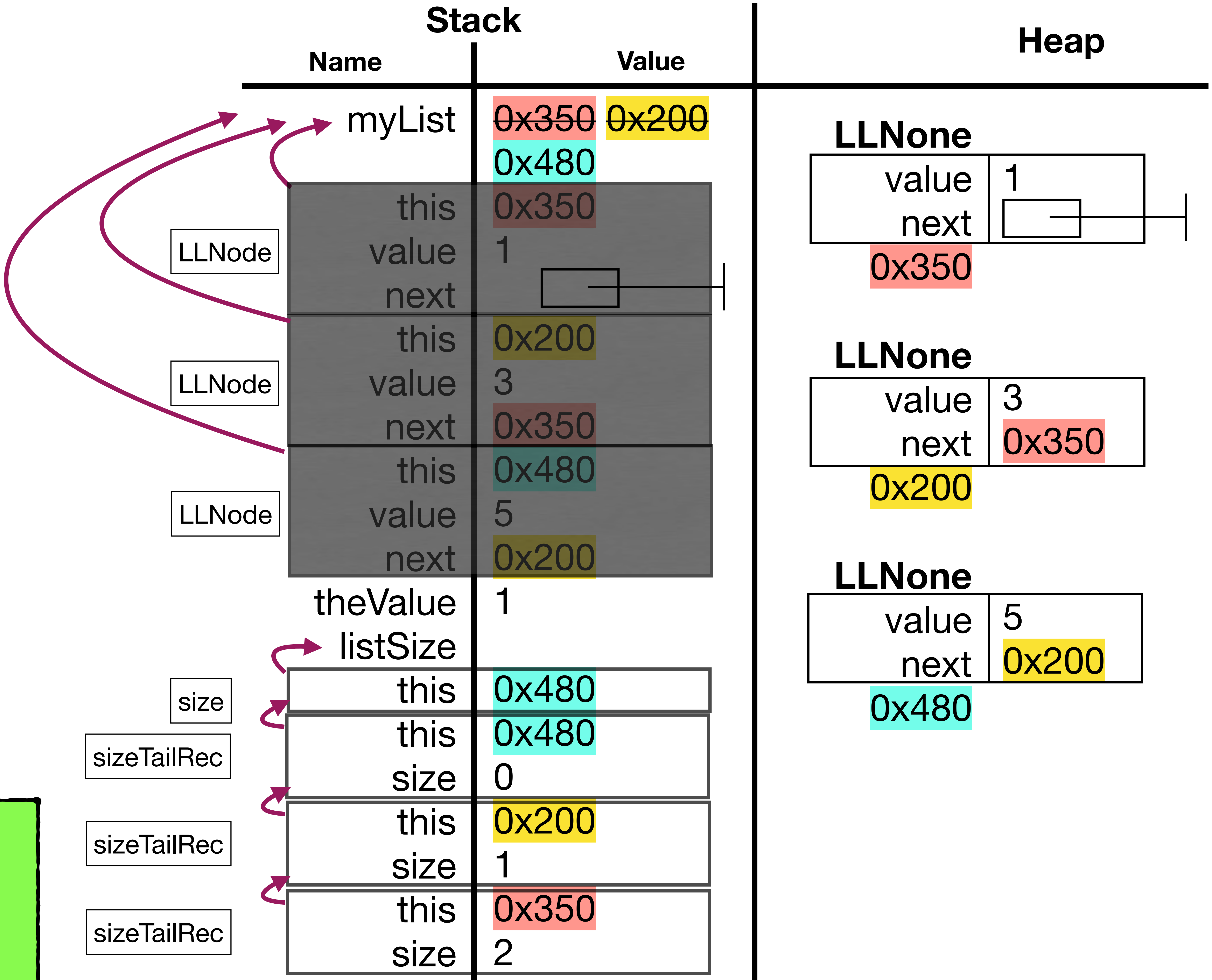
in/out

theValue: 1

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Keep making calls until we reach the base case



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1
	listSize	
size	this	0x480
sizeTailRec	this	0x480
	size	0
sizeTailRec	this	0x200
	size	1
sizeTailRec	this	0x350
	size	2

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

0x200

LLNone

value	5
next	0x200

0x480

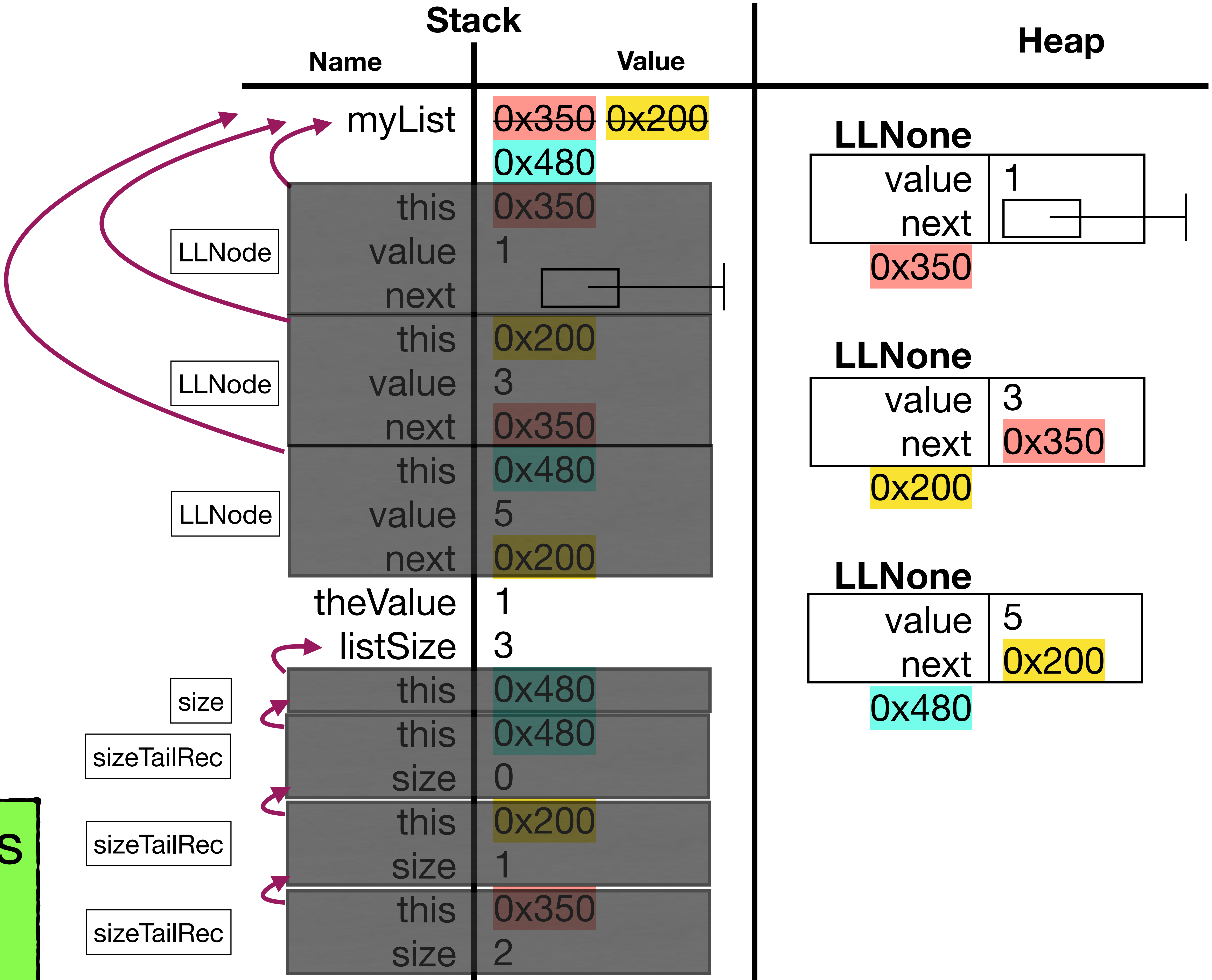
in/out

theValue: 1

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Return 3 up the recursive calls



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1
	listSize	3
size	this	0x480
sizeTailRec	this	0x480
	size	0
sizeTailRec	this	0x200
	size	1
sizeTailRec	this	0x350
	size	2

Heap

LLNone
value 1
next
0x350

LLNone
value 3
next 0x350
0x200

LLNone
value 5
next 0x200
0x480

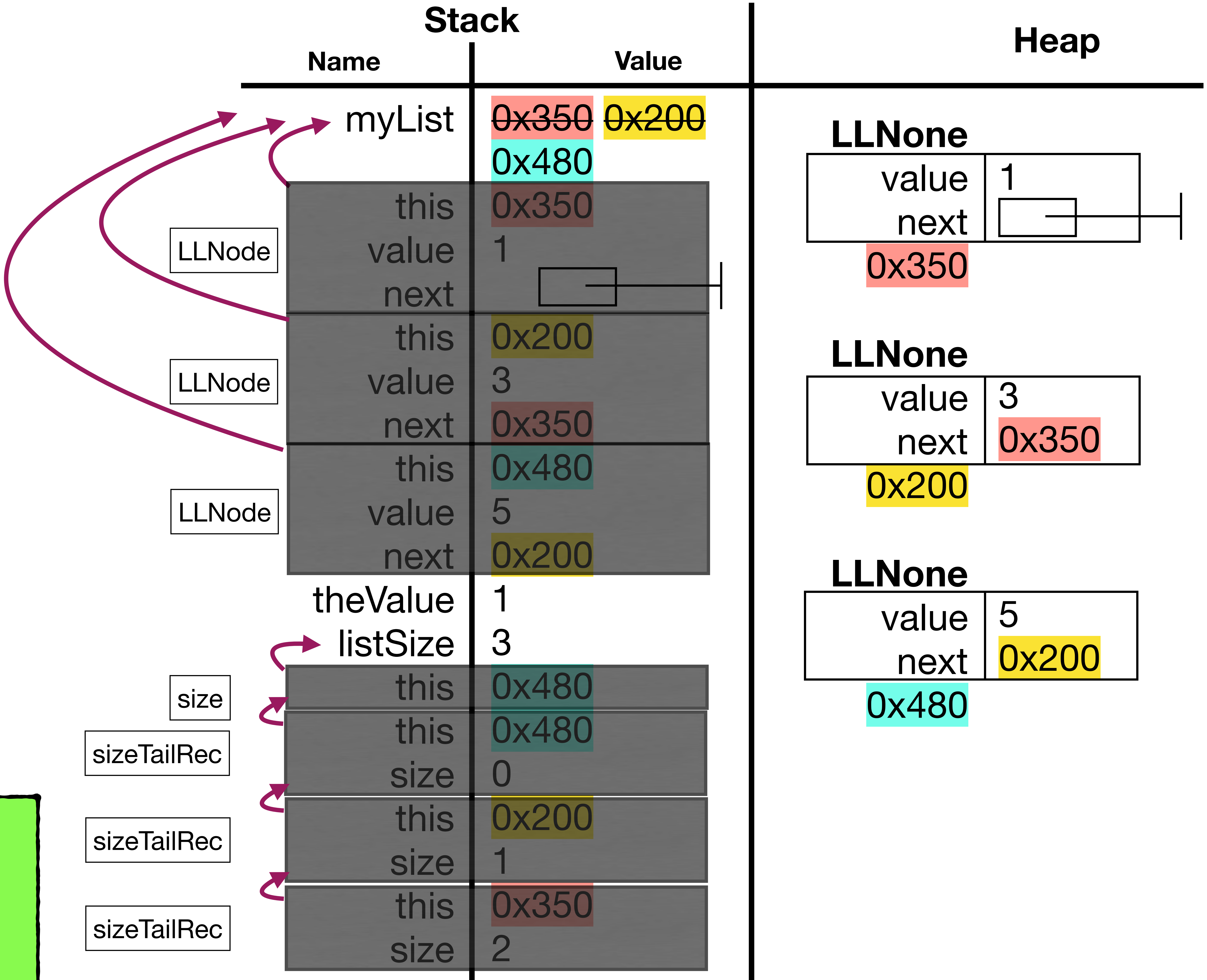
in/out

theValue: 1

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

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

  def size(): Int = {
    sizeTailRec(0)
  }

}
```

```
def main(args: Array[String]): Unit = {
  var myList: LLNode[Int] = new LLNode[Int](1, null)
  myList = new LLNode[Int](3, myList)
  myList = new LLNode[Int](5, myList)

  val theValue: Int = myList.next.next.value
  println("theValue: " + theValue)

  val listSize: Int = myList.size()
  println("size: " + listSize)
}
```

- Print 3 to the screen



Stack

	Name	Value
	myList	~~0x350~~ ~~0x200~~ 0x480
LLNode	this	0x350
	value	1
	next	
LLNode	this	0x200
	value	3
	next	0x350
LLNode	this	0x480
	value	5
	next	0x200
	theValue	1
	listSize	3
size	this	0x480
sizeTailRec	this	0x480
	size	0
sizeTailRec	this	0x200
	size	1
sizeTailRec	this	0x350
	size	2

Heap

LLNone

value	1
next	

0x350

LLNone

value	3
next	0x350

0x200

LLNone

value	5
next	0x200

0x480

in/out

theValue: 1
size: 3