PROGRAMMING AND DATA STRUCTURES

DATA STRUCTURES: IMPLEMENTATION

HOURIA OUDGHIRI

FALL 2023

OUTLINE

- Implementations of the List interface
- Implementation of the Stack
- Implementation of the Queue
- Implementation of the Priority Queue

STUDENT LEARNING OUTCOMES

At the end of this chapter, you should be able to:

- Implement `List` using an array
- Implement `List` using linked nodes
- Implement `Stack` using ArrayList
- Implement `Queue` using LinkedList
- Implement `Priority Queue` using ArrayList
- Analyze the complexity of the operations of the five data structures

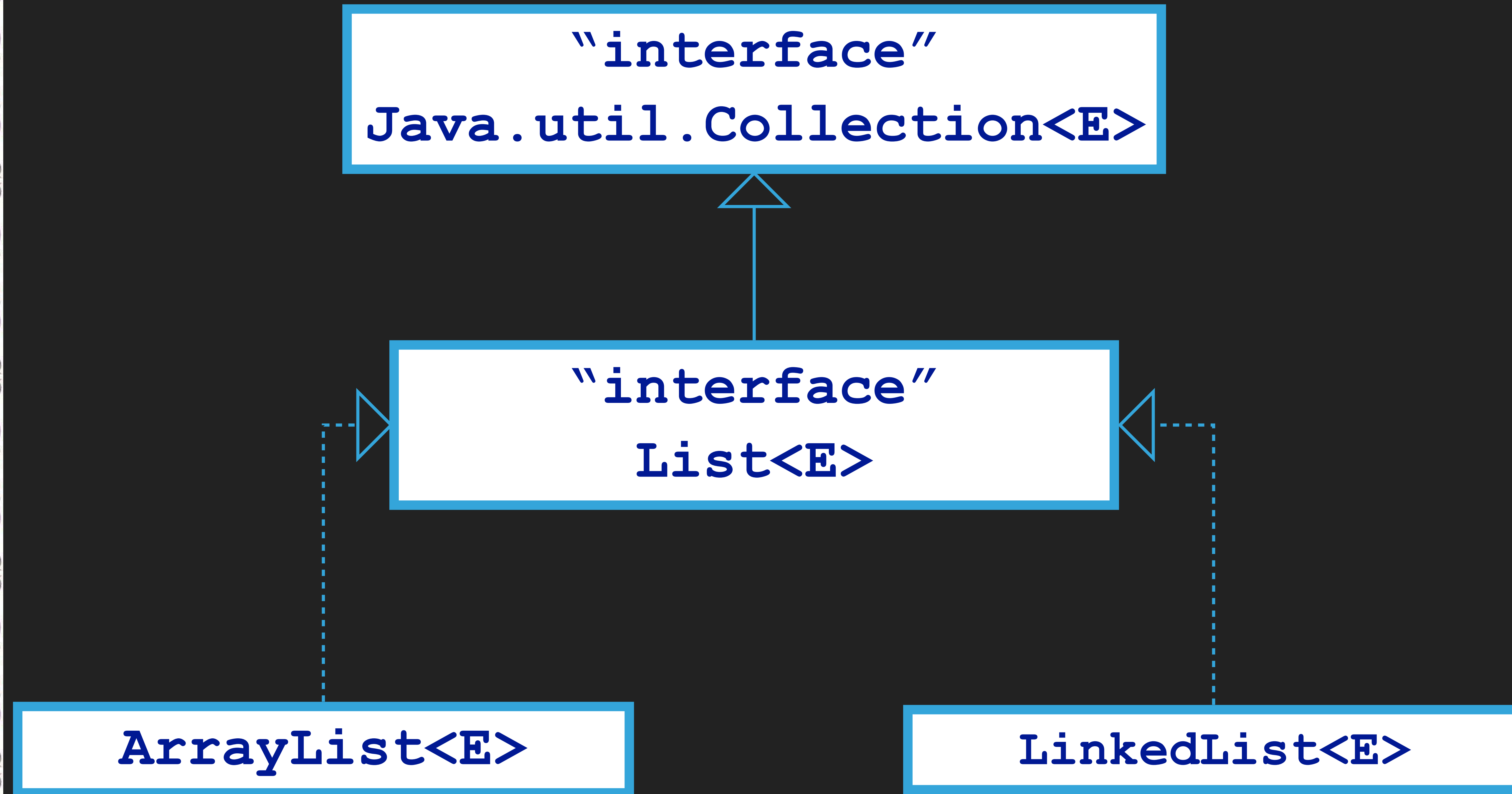
Why data structure implementation?

- Data Structures: List, Stack, Queue, PriorityQueue available in the Java API
- How are they implemented?
- How to create new data structures?
- How a data structure is implemented rather than only how to use it

List

- Store data in order
- Common operations on List
  - Retrieve an element from the list
  - Add a new element to the list
  - Remove an element from the list
  - Get the number of elements in the list

List


“interface”
Java.util.Collection<E>
“interface”
List<E>
ArrayList<E>
LinkedList<E>

List

- Array Based List: **`ArrayList<E>`**
  - Fixed array size when the list is constructed
  - New larger array created when the current array is full
- Linked List: **`LinkedList<E>`**
  - Size not fixed
  - Nodes are created when an item is added
  - Nodes are linked together to form the list

List

✦ Array Based List

0	1	2	3	4	5	6	7	8	9

Size = 0, Capacity = 10

22	32	13	56	77					
0	1	2	3	4	5	6	7	8	9

Size = 5, Capacity = 10

22	32	13	56	77	27	11	102	37	41
0	1	2	3	4	5	6	7	8	9

Size = 10, Capacity = 10

Size = 11, Capacity = 15

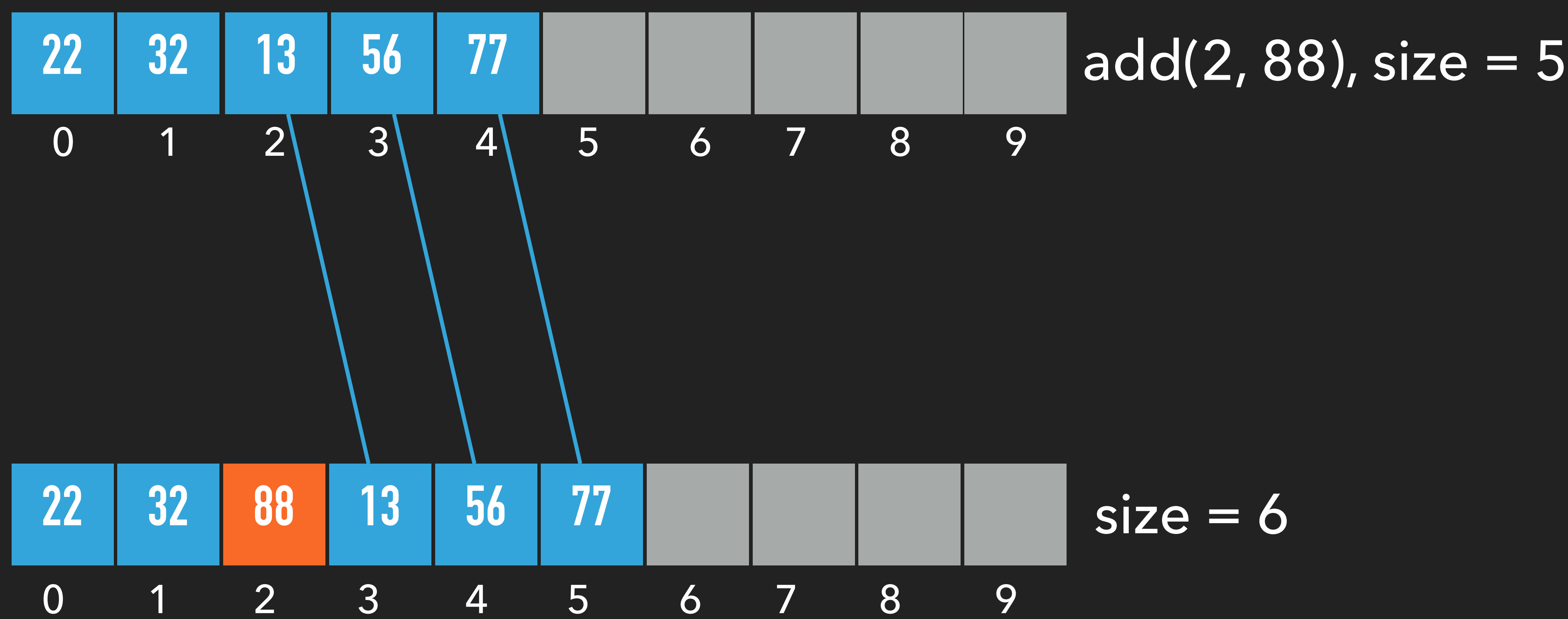
22	32	13	56	77	27	11	102	37	41	52				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Array Based List

- Inserting an element at a specific index
  - If (`size == capacity`), create a new array with new capacity = (1.5 * capacity) and copy all the elements from the current array to the new array. The new array becomes the new list
  - Shift all the elements after the index, modify element at index and increment the size

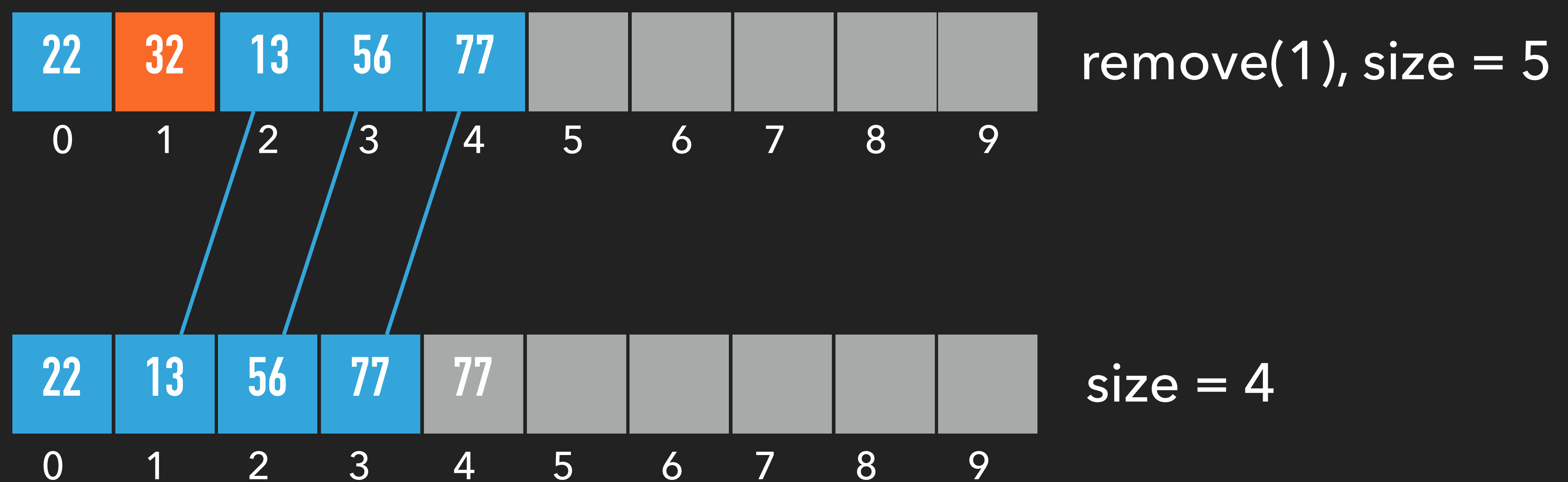
Array Based List

- Inserting an element at a specific index



Array Based List

- Removing an element at a specific index
  - Shift all the elements after the index and decrement the size



Array Based List

ArrayList<E>
-elements: E[] -size: int
+ArrayList() +ArrayList(int) +add(int, E): boolean +add(E): boolean +get(int): E +set(int, E): E +size(): int +clear(): void +isEmpty(): boolean +remove(int): E +trimToSize(): void -ensureCapacity(): void -checkIndex(int): void +toString(): String +iterator(): Iterator<E>

Array Based List

Test.java

```
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        ArrayList<String> cities = new ArrayList<>();
        cities.add("New York");
        cities.add("San Diego");
        cities.add("Atlanta");
        cities.add(0,"Baltimore");
        cities.add(2,"Pittsburg");
        // display the content of the list
        System.out.println(cities.toString());
        // iterator to display the elements of the list
        Iterator<String> cityIterator = cities.iterator();
        while(cityIterator.hasNext()) {
            System.out.print(cityIterator.next() + " ");
        }
        System.out.println();
        // get(index) to display the elements of the list
        for(int i=0; i<cities.size(); i++) {
            System.out.print(cities.get(i) + " ");
        }
        System.out.println();
        // remove(int)
        cities.remove(1);
        System.out.println(cities.toString());

    }
}
```

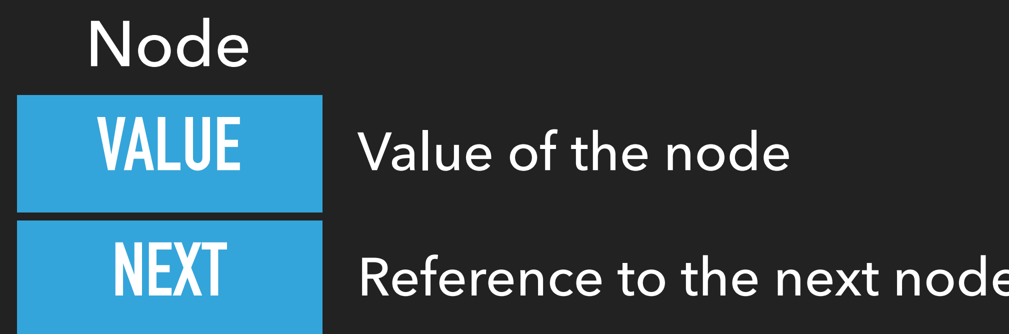
Array Based List

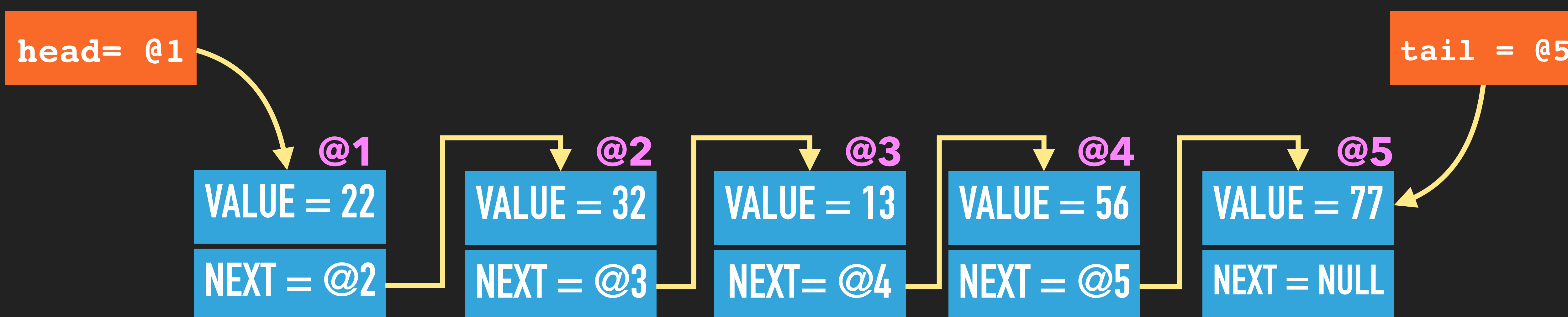
- Complexity of the **`ArrayList`** operations?

Method	Complexity	Method	Complexity
`ArrayList()`	O(1)	`iterator()`	O(1)
`ArrayList(int)`	O(1)	`trimToSize`	O(n)
`size()`	O(1)	`ensureCapacity`	O(n)
`checkIndex()`	O(1)	`add(int, E)`	O(n)
`get(int)`	O(1)	`remove(int)`	O(n)
`set(int, E)`	O(1)	`toString()`	O(n)
`isEmpty()`	O(1)	`add(E)`	O(1) - O(n)
`clear()`	O(1)		

List

- Linked List





Size = 5, Capacity: infinite

Linked List

- List implementation using linked nodes
  - Class **Node** (inner class - inside LinkedList)

Node
+value: E +next: Node +Node(E)

Linked List

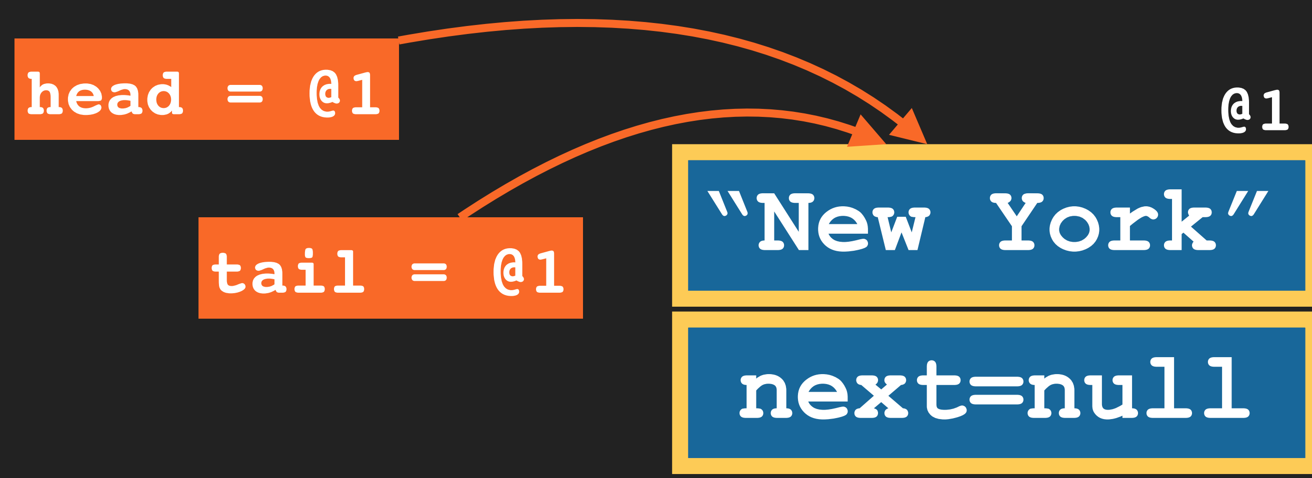
```
LinkedList<E>
-----------------------------
-head: Node
-tail: Node
-size: int
-----------------------------
+LinkedList()
+addFirst(E): void
+addLast(E): void
+add(E): boolean
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
+clear(): void
+isEmpty(): boolean
+size(): int
+iterator(): Iterator<E>
```

Linked List

```
Node head = null;
Node tail = null; size =0;
```

Empty list

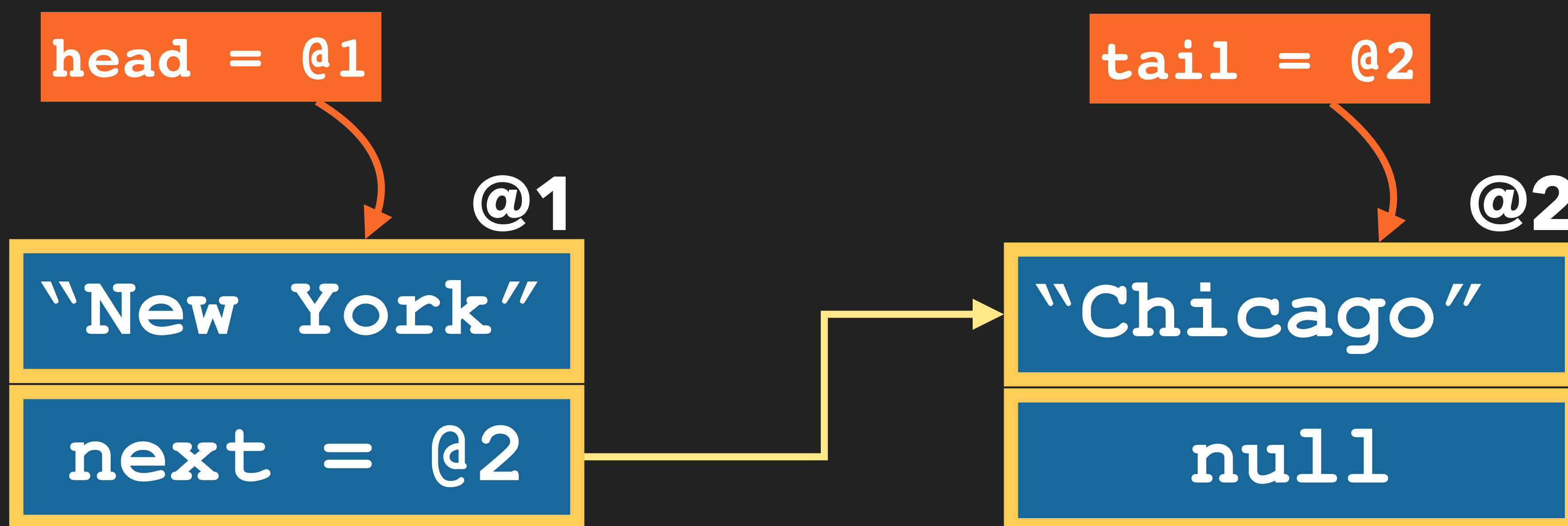
```
// Adding the first element
head = new Node("New York");
tail = head; size++;
```



One element in the list

Linked List

Adding an element at the end - **`addLast()`**



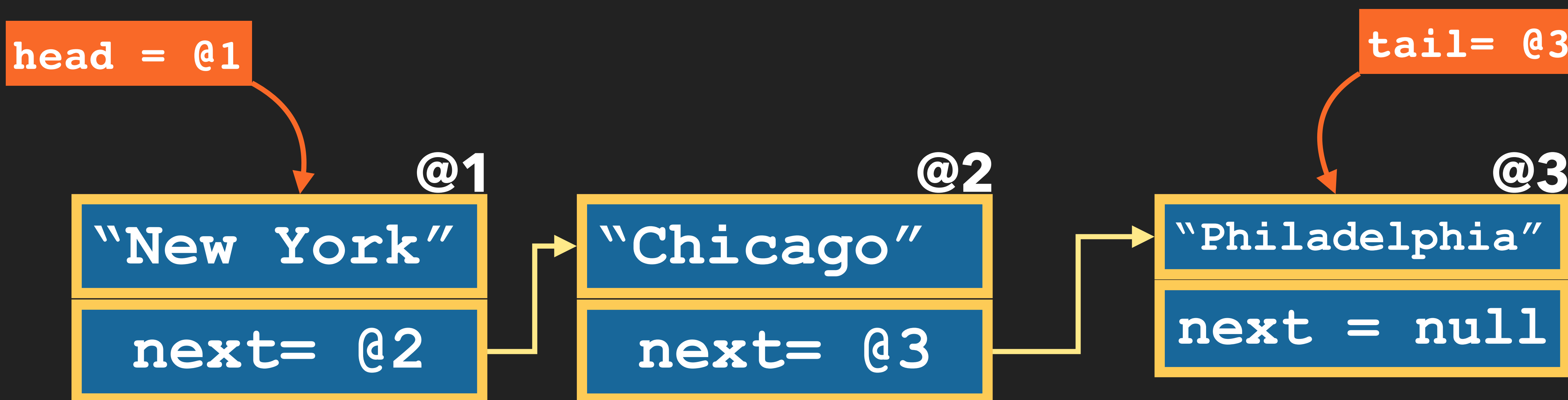
Two elements in the list

Step1 `tail.next = new Node("Chicago");`

Step2 `tail = tail.next; size++;`

Linked List

Adding an element at the end - `addLast()`

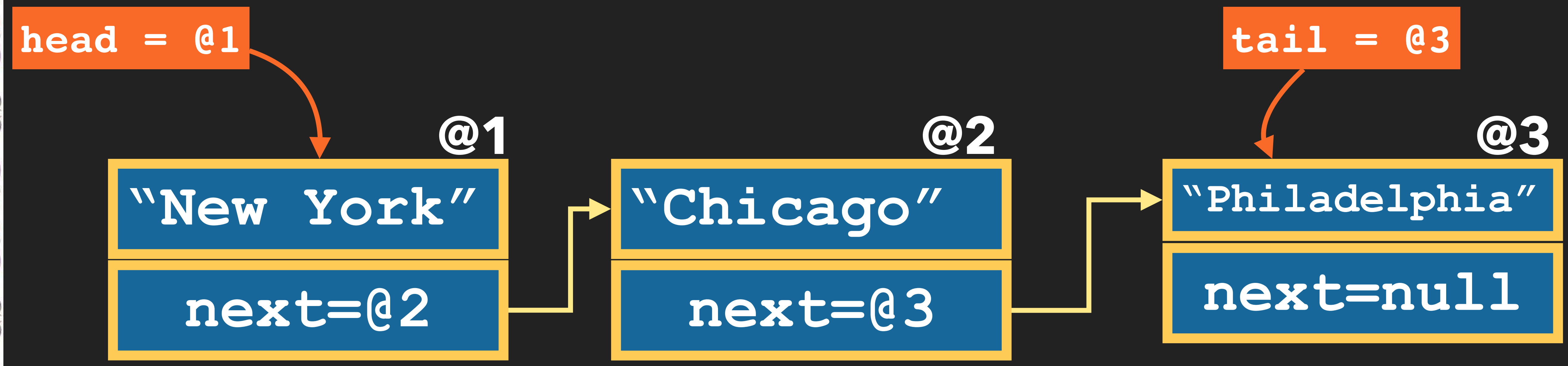


Three elements in the list

Step1
```
tail.next = new Node("Philadelphia");
```

Step2
```
tail = tail.next; size++;
```

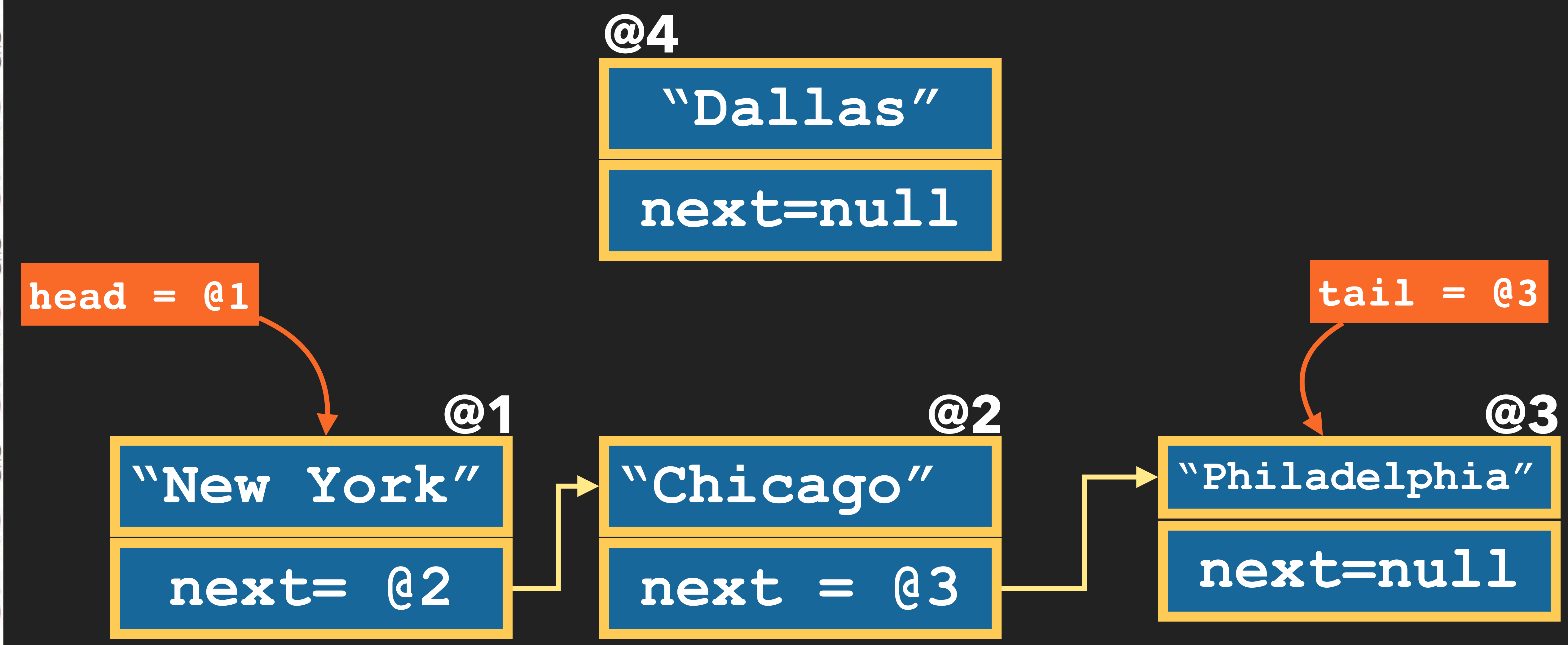
Linked List




```
// Traversal of the linked list nodes
Node node = head;
while(node != null){
   System.out.println(node.value);
   node = node.next;
}
```

Linked List

Adding an element at the head - `addFirst()`

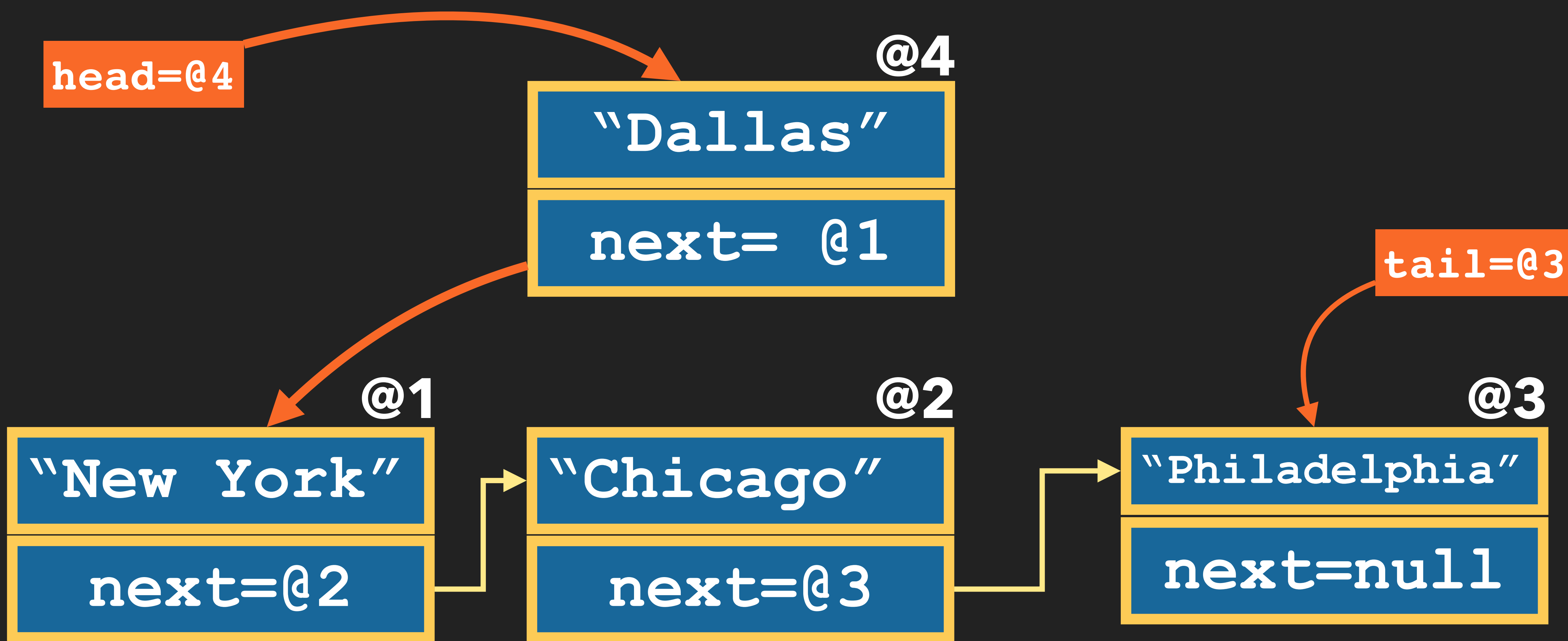


Step1

```
Node newNode = new Node("Dallas");
```

Linked List

Adding an element at the head - **`addFirst()`**

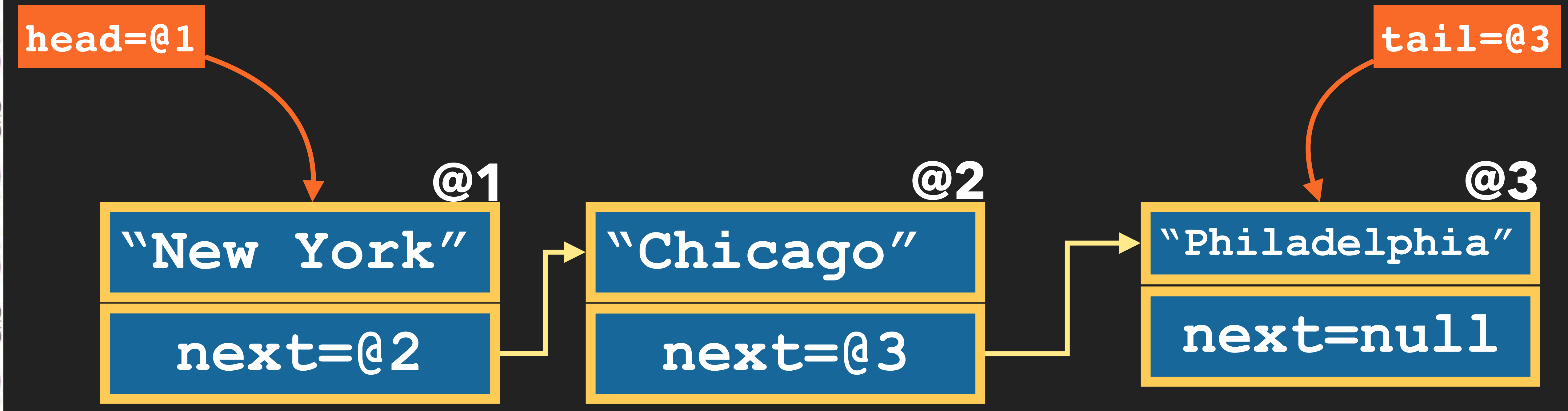


Step2

```
newNode.next = head; head = newNode; size++;
```

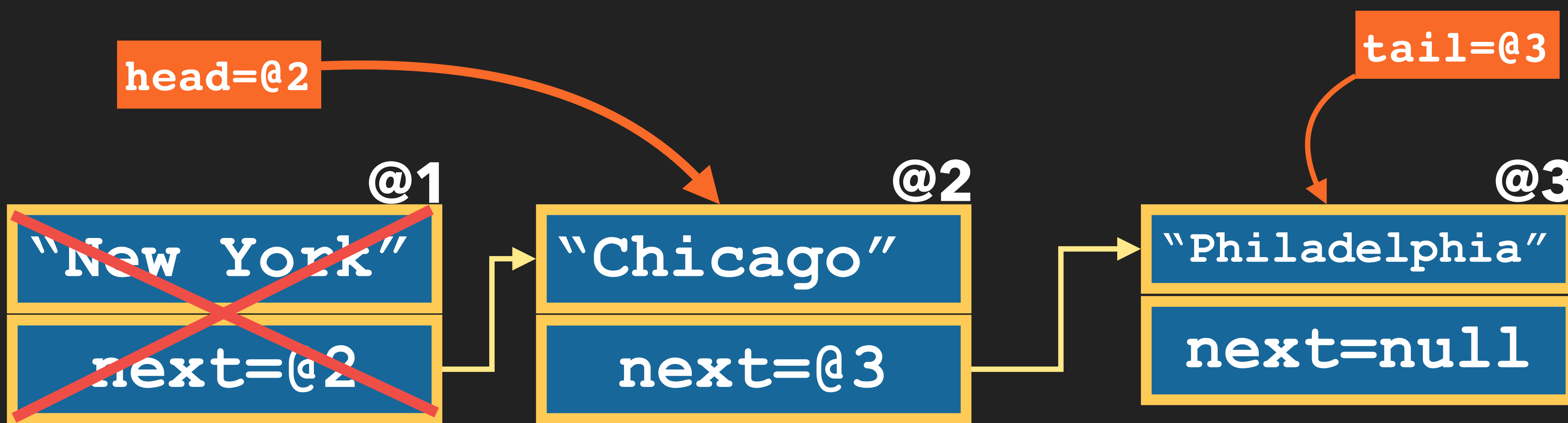
Linked List

Removing an element at the head- **`removeFirst()`**



Linked List

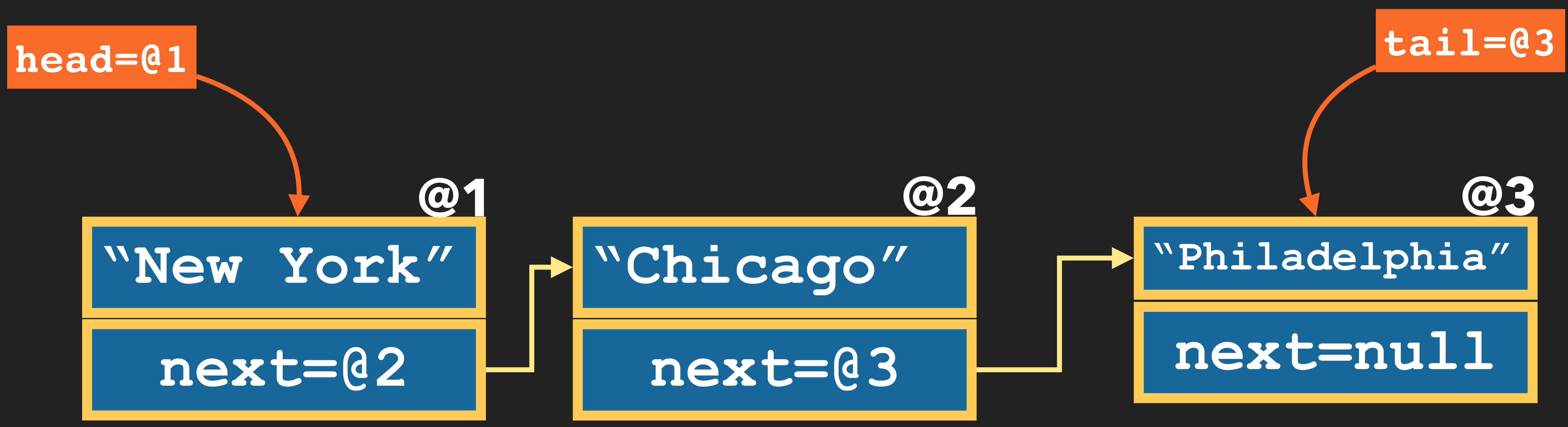
Removing an element at the head- **`removeFirst()`**




```
head= head.next;
size --;
```

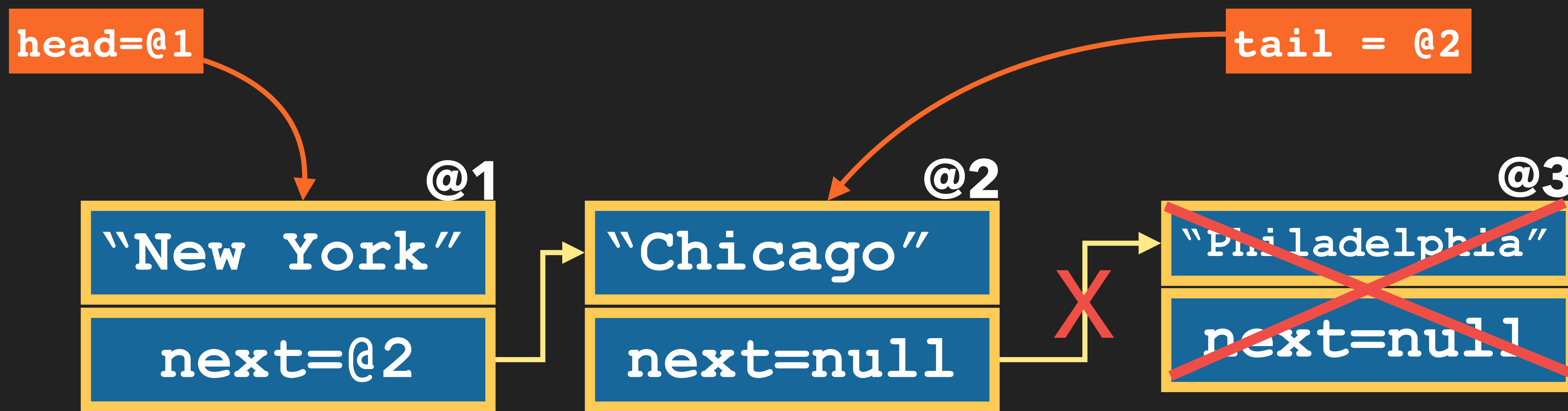
Linked List

Removing an element at the tail- **`removeLast()`**



Linked List

Removing an element at the tail- **removeLast()**




```
//go to the node before the last
node = head;
while(node.next != tail)
   node = node.next;
node.next= null;// node becomes tail
tail = node;
size--;
```

Linked List

Test.java

```
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        // Testing LinkedList
        System.out.println("\nLinkedList:");
        LinkedList<String> LLCities = new LinkedList<>();
        LLCities.add("Boston");
        LLCities.add("Philadelphia");
        LLCities.addFirst("San Francisco");
        LLCities.addFirst("Washington");
        LLCities.addFirst("Portland");
        System.out.println(LLCities.toString());
        cityIterator = LLCities.iterator();
        System.out.print("LinkedList (iterator): ");
        while(cityIterator.hasNext()) {
         System.out.print(cityIterator.next() + " ");
        }
        System.out.println();
        LLCities.removeFirst();
        System.out.println(LLCities.toString());
        LLCities.removeLast();
        System.out.println(LLCities.toString());
    }
}
```

Linked List

- Complexity of the **`LinkedList`** operations

Method	Complexity	Method	Complexity
`LinkedList()`	**O(1)**	`addFirst()`	**O(1)**
`size()`	**O(1)**	`addLast()`	**O(1)**
`clear()`	**O(1)**	`add(E)`	**O(1)**
`isEmpty()`	**O(1)**	`removeFirst()`	**O(1)**
`iterator()`	**O(1)**	`removeLast()`	**O(n)**
`getFirst()`	**O(1)**	`toString()`	**O(n)**
`getLast()`	**O(1)**		

Linked List

- Variations of Linked List
  - Doubly Linked List

    Every node is linked to the next and the previous nodes

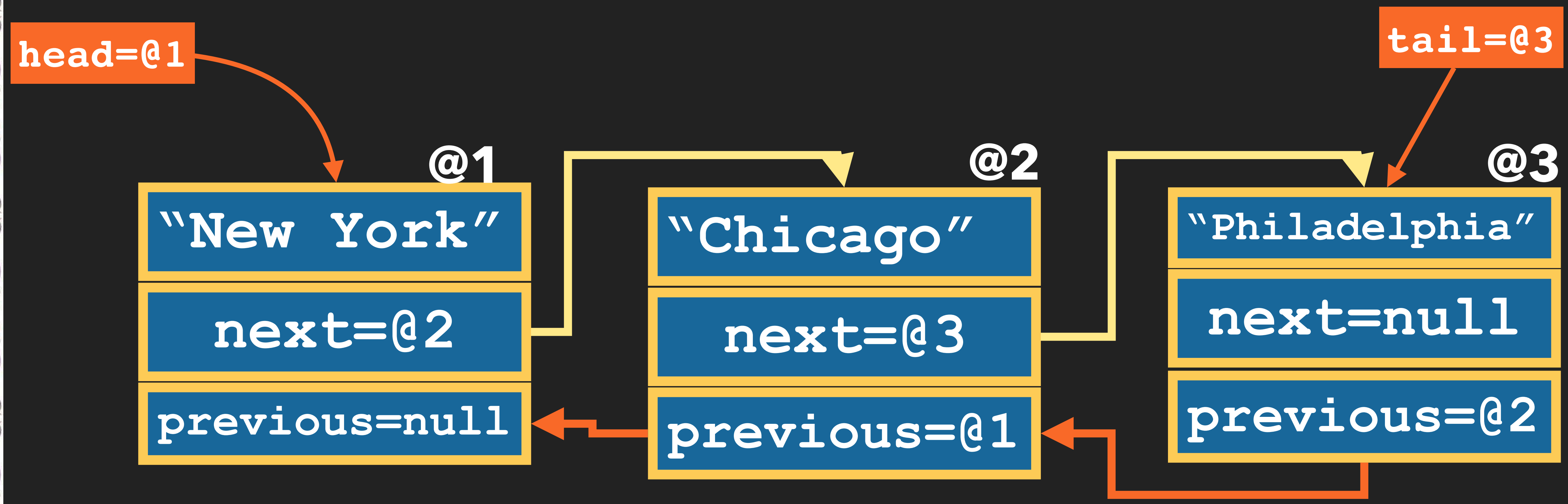
  - Circular Linked List

    Last element is linked back to the first element

Linked List

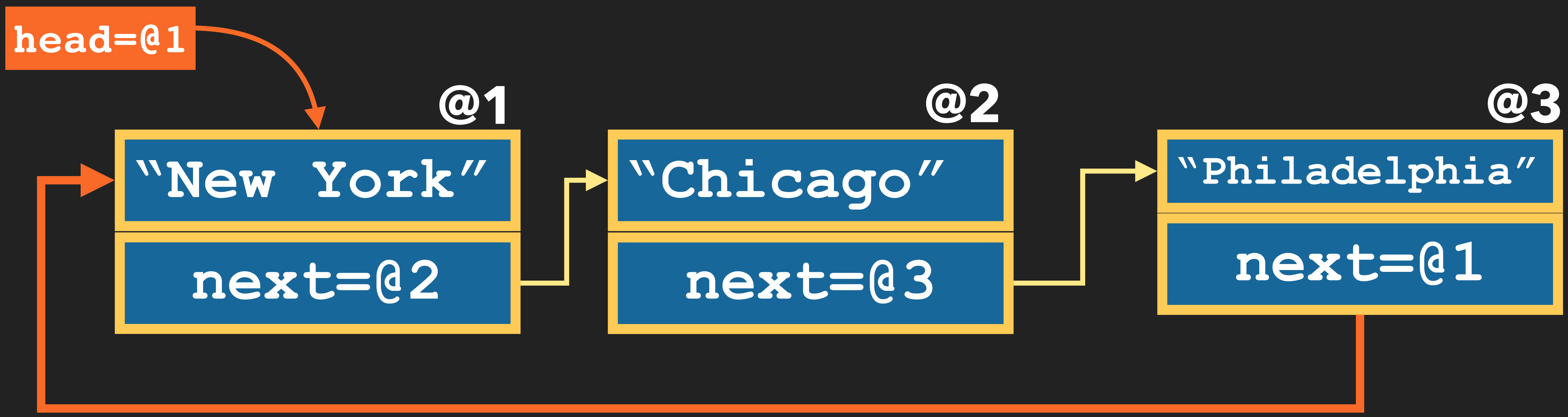
✦ Doubly Linked List

Improves the performance of removeLast (from **O(n)** to **O(1)**)



Linked List

- Circular Linked List



Stack and Queue

- Stack is implemented using an <u>array based list</u> (or linked list) with access only at the end of the list (or the head of the list)

- Queue is implemented using a <u>linked list</u> with access at the head and the tail

Stack

Stack<E>
-elements: ArrayList<E>
+Stack() +size(): int +isEmpty(): boolean +push(E): void +peek(): E +pop(): E +toString(): String

Stack

Test.java

```
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        // Testing Stack
        Stack<String> cityStack = new Stack<>();
        cityStack.push("New York");
        cityStack.push("San Diego");
        cityStack.push("Atlanta");
        cityStack.push("Baltimore");
        cityStack.push("Pittsburg");
        System.out.println("City Stack (toString): " +
                           cityStack.toString());
        System.out.print("City Stack (pop): ");
        while(!cityStack.isEmpty())
            System.out.print(cityStack.pop() + " ");
    }
}
```

Stack

- Complexity of the Stack operations

Method	Complexity
`Stack<>()`	**O(1)**
`peek()`	**O(1)**
`pop()`	**O(1)**
`push()`	**O(1)/O(n)**
`size()`	**O(1)**
`isEmpty()`	**O(1)**
`toString()`	**O(n)**

Queue

- Implemented using a LinkedList

Queue<E>
-list: LinkedList<E>
+Queue() +offer(E): void +poll(): E +peek(): E +size(): int +clear(): void +isEmpty(): boolean +toString(): String

Queue

Test.java

```
public class Test {
   public static void main(String[] args) {
     // Testing Queue
     Queue<String> cityQueue = new Queue<>();
     cityQueue.offer("New York");
     cityQueue.offer("San Diego");
     cityQueue.offer("Atlanta");
     cityQueue.offer("Baltimore");
     cityQueue.offer("Pittsburg");
     System.out.println("City Queue (toString): " +
                                   cityQueue.toString());
     System.out.print("City Queue (poll): ");
     while(!cityQueue.isEmpty())
       System.out.print(cityQueue.poll() + " ");
   }
}
```

Queue

- Performance of the Queue operations

Method	Complexity
`Queue<>()`	O(1)
`offer(E)`	O(1)
`poll()`	O(1)
`peek()`	O(1)
`size()`	O(1)
`clear()`	O(1)
`isEmpty()`	O(1)
`toString()`	O(n)

Priority Queue

- Queue with priority

```
PriorityQueue<E>
-----------------------------------------
-list: ArrayList<E>
-comparator: Comparator<E>
-----------------------------------------
+PriorityQueue()
+PriorityQueue(Comparator<E>)
+offer(E): void
+poll(): E
+peek(): E
+size(): int
+clear(): void
+isEmpty(): boolean
+toString(): String
```

Priority Queue

Test.java

```
public class Test {
   public static void main(String[] args) {
    // Testing PriorityQueue
    PriorityQueue<String> cityPriorityQueue = new PriorityQueue<>();
    cityPriorityQueue.offer("New York");
    cityPriorityQueue.offer("San Diego");
    cityPriorityQueue.offer("Atlanta");
    cityPriorityQueue.offer("Baltimore");
    cityPriorityQueue.offer("Pittsburg");
    System.out.println("\nCity Priority Queue: "+
                         cityPriorityQueue.toString());
    System.out.print("City Priority Queue (poll): ");
    while(!cityPriorityQueue.isEmpty()) {
      System.out.print(cityPriorityQueue.poll() + " ");
    }
  }
}
```

Priority Queue

- Complexity of the PriorityQueue operations

Method	Complexity
PriorityQueue()	O(1)
offer()	O(n)
poll()	O(n)
peek()	O(1)
size()	O(1)
isEmpty()	O(1)
clear()	O(1)
toString()	O(n)

Summary

- Data Structures
  - ✓ List - Array list and Linked List
  - ✓ Stack - implemented using ArrayList
  - ✓ Queues - Queue and PriorityQueue using LinkedList and ArrayList
- Complexity of the data structure operations