PROGRAMMING AND DATA STRUCTURES

# DATA STRUCTURES: IMPLEMENTATION

HOURIA OUDGHIRI

FALL 2021

# OUTLINE

- Implementations of the List
- 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 the List using an array
- Implement the List using linked nodes
- Implement the Stack using an array
- Implement the Queue using a LinkedList
- Implement the Priority Queue using an array
- Analyze the complexity of the operations on all the data structures

# Why data structure implementation?

- Data Structures: List, Stack, Queue, PriorityQueue available in Java API
- How are they implemented?
- How to create new data structures?
- Become a data structure designer rather than a data structure user

# List

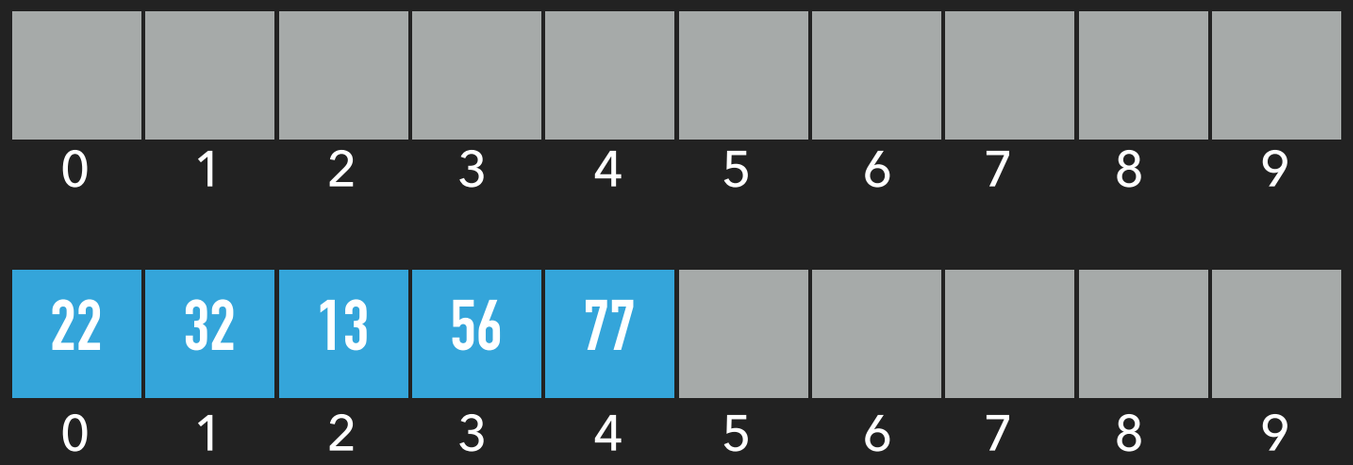
- Store data in order
- Common operations on List
  - Retrieve an element from the list
  - Add a new element into the list
  - Remove an element from the list
  - Get the number of elements in the list

# 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

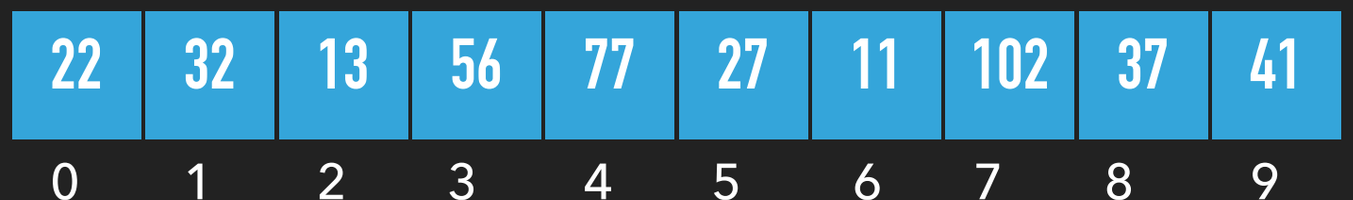
## ✦ Array Based List




Size = 0, Capacity = 10

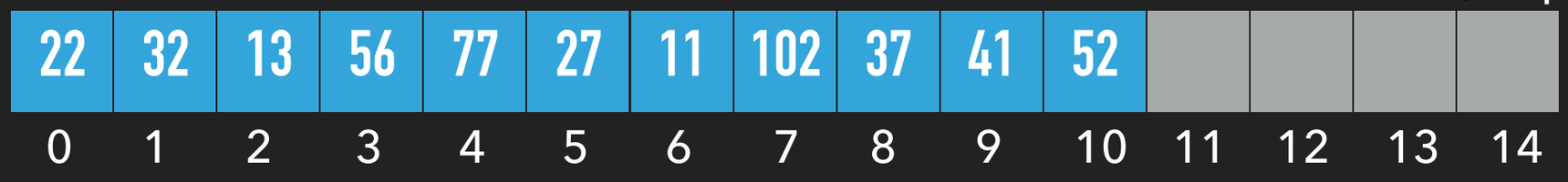



Size = 5, Capacity = 10




Size = 10, Capacity = 10

Size = 11, Capacity = 15

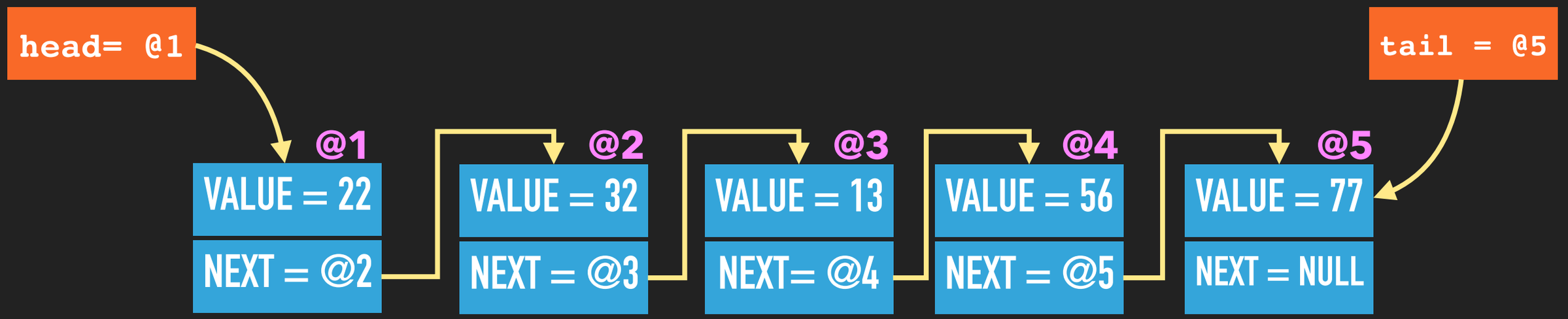


# List

## ✦ Linked List

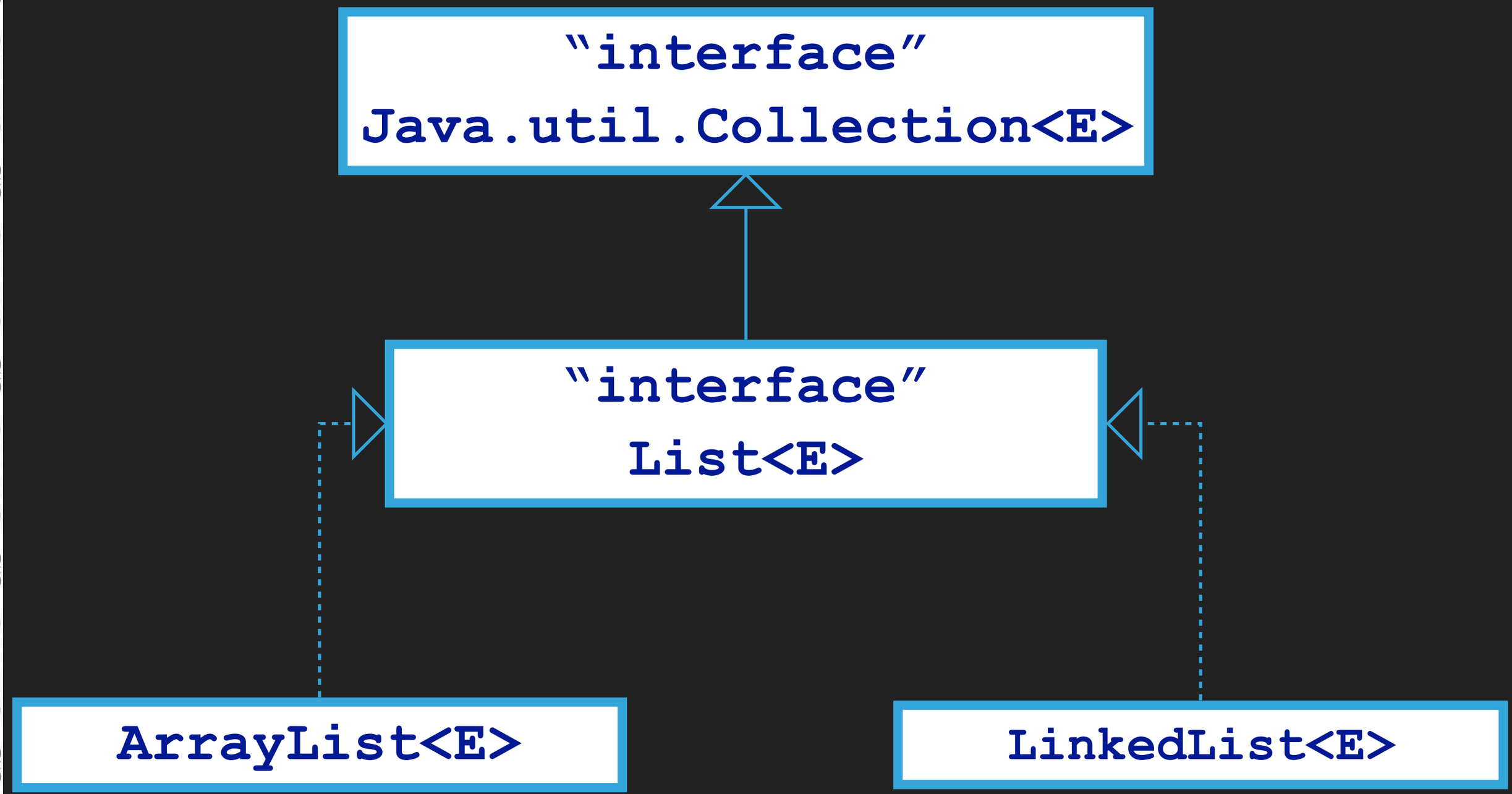
Node

| VALUE | Value of the node |
|---|---|
| NEXT | Reference to the next node |




Size = 5, Capacity: infinite

# List


"interface"
Java.util.Collection<E>
"interface"
List<E>
ArrayList<E>
LinkedList<E>

# Array Based List

- Inserting an element at a specific index

- If (`size == capacity`), create a new array with new size = (1.5 * size) 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 increase the size by 1

# 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 decrease the size by 1

# Array Based List

**ArrayBasedList<E>**

-elements: E[]
-size: int

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

# Array Based List

**ArrayBasedList.java**

```
public class ArrayBasedList<E> {
   // data members
   private E[] elements;
   private int size;
   // Constructors
   public ArrayBasedList() {
      elements = (E[]) new Object[10];
      size = 0;
   }
   public ArrayBasedList(int capacity) {
      elements = (E[]) new Object[capacity];
      size = 0;
   }
```

# Array Based List

ArrayBasedList.java

```
// Adding an item to the list (2 methods)
public boolean add(E item) {
    return add(size, item);
}
public boolean add(int index, E item){
    if(index > size || index < 0)
      throw new ArrayIndexOutOfBoundsException();
    ensureCapacity();
    for(int i=size-1; i<index; i--)
      elements[i+1] = elements[i];
    elements[index] = item;
    size++;
    return true;
}
```

# Array Based List

ArrayBasedList.java

```
// Getter and Setter
public E get(int index) {
    checkIndex(index);
    return elements[index];
}
public E set(int index, E item) {
    checkIndex(index);
    E oldItem = elements[index];
    elements[index] = item;
    return oldItem;
}
// Size of the list
public int size() { return size; }
// Clear the list
public void clear() { size = 0; }
// Check if the list is empty
public boolean isEmpty() { return (size == 0);}
```

# Array Based List

ArrayBasedList.java

```
// Removing an object from the list
public boolean remove(Object o) {
    E item = (E) o;
    for(int i=0; i<size; i++)
      if(elements[i].equals(item)){
        remove(i);
        return true;
      }
    return false;
}
// Removing the item at index from the list
public E remove(int index) {
      checkIndex(index);
      E item = elements[index];
      for(int i=index; i<size-1; i++)
        elements[i] = elements[i+1];
      size--;
      return item;
}
```

# Array Based List

ArrayBasedList.java

```
// Shrink the list to size
public void trimToSize() {
     if (size != elements.length) {
        E[] newElements = (E[]) new Object[size];
        for(int i=0; i<size; i++)
           newElements[i] = elements[i];
        elements = newElements;
     }
}[
// Grow the list if needed
private void ensureCapacity() {
   if(size >= elements.length) {
      int newCap = (int) (elements.length * 1.5);
      E[] newElements = (E[]) new Object[newCap];
      for(int i=0; i<size; i++)
           newElements[i] = elements[i];
      elements = newElements;
   }}
```

# Array Based List

ArrayBasedList.java

```
// Check if the index is valid
private void checkIndex(int index){
     if(index < 0 || index >= size)
        throw new ArrayIndexOutOfBoundsException(
         "Index out of bounds. Must be between 0 and "+
        (size-1));
}
// toString() method
public String toString() {
     String output = "[";
     for(int i=0; i<size-1; i++)
       output += elements[i] + " ";
     output += elements[size-1] + "]";
     return output;
}
```

# Array Based List

ArrayBasedList.java

```
// Iterator for the list
public Iterator<E> iterator(){
        return new ArrayIterator();
}
// Inner class that implements Iterator<E>
private class ArrayIterator implements Iterator<E>{
   private int current = -1;

   public boolean hasNext() { return current < size-1; }

   public E next() { return elements[++current]; }
}
```

# Array Based List

Test.java

```
public class Test {
   public static void main(String[] args) {
      ArrayBasedList<String> cities = new ArrayBasedList<>();
      cities.add("New York");
      cities.add("San Diego");
      cities.add("Atlanta");
      cities.add("Baltimore");
      cities.add(“Pittsburg");
      // toString() to display the content of the list
      System.out.println(cities.toString());
      // iterator to visit and 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 visit and display the elements of the list
      for(int i=0; i<cities.size(); i++) {
         System.out.print(cities.get(i) + " ");
      }
   }
```

# Array Based List

- What is the complexity of the operations in the ArrayBasedList?

# Array Based List

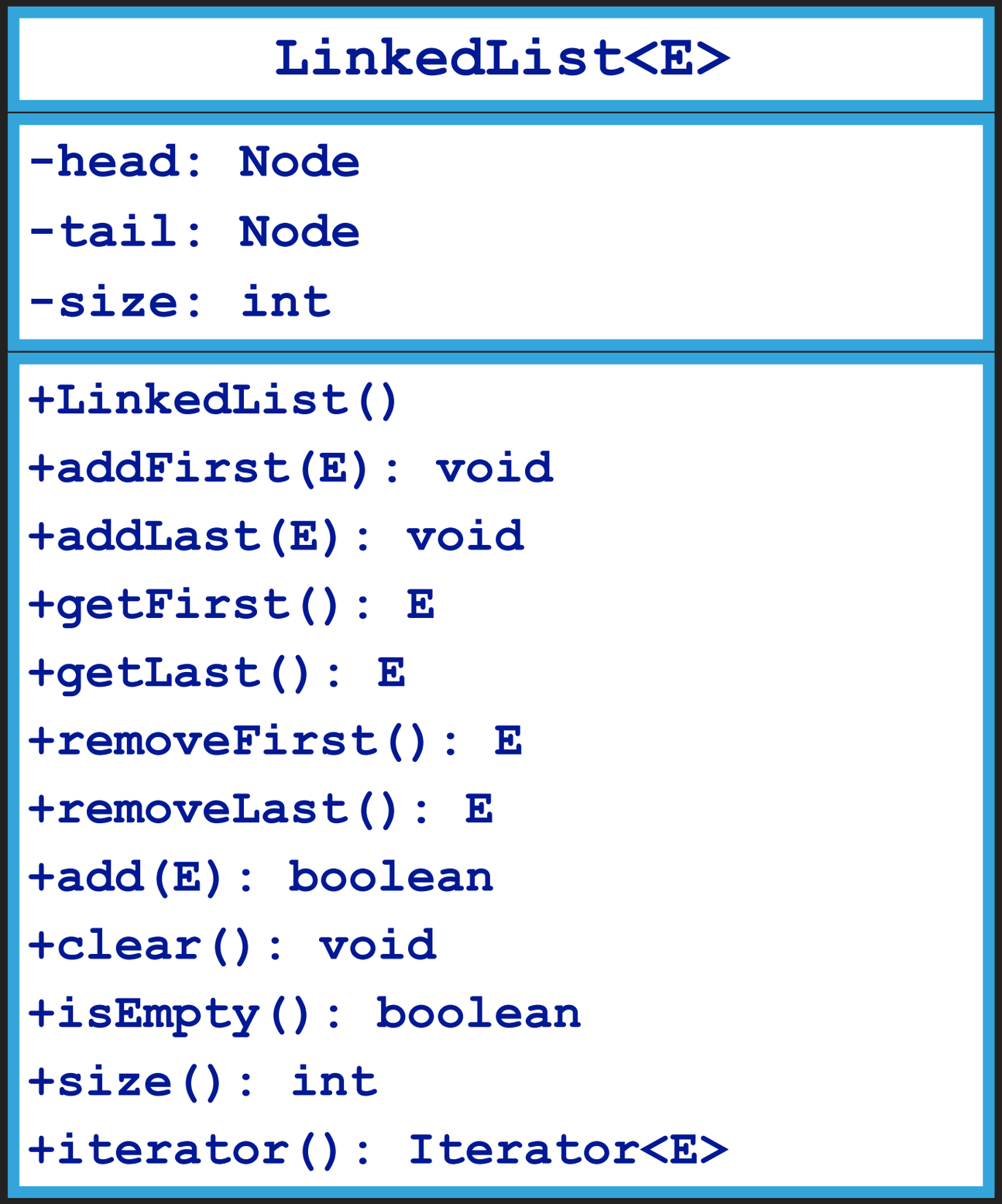
| Method | Complexity | Method | Complexity |
|---|---|---|---|
| `ArrayList()` | **O(1)** | `iterator()` | **O(1)** |
| `ArrayList(int)` | **O(1)** | `trimToSize` | **O(n)** |
| `size()` | **O(1)** | `ensureCapacity` | **O(n)** |
| `checkIndex()` | **O(1)** | `add(int, E)` | **O(n)** |
| `get(int)` | **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)** | | |

# Linked List

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

| **Node** |
| --- |
| **+value: E**<br>**+next: Node**<br>**+Node(E)** |

# Linked List

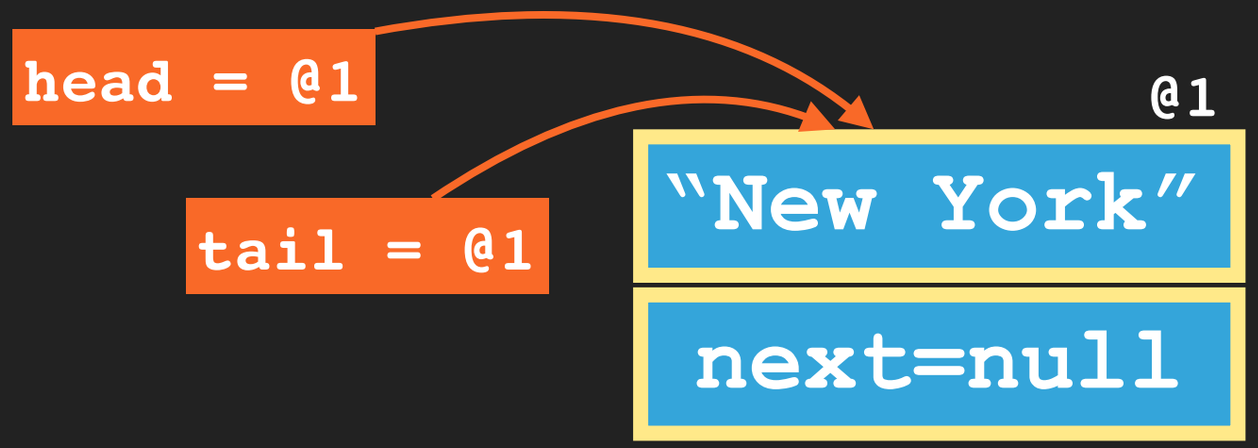

LinkedList<E>
-head: Node
-tail: Node
-size: int
+LinkedList()
+addFirst(E): void
+addLast(E): void
+getFirst(): E
+getLast(): E
+removeFirst(): E
+removeLast(): E
+add(E): boolean
+clear(): void
+isEmpty(): boolean
+size(): int
+iterator(): Iterator<E>

# Linked List

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

Empty list

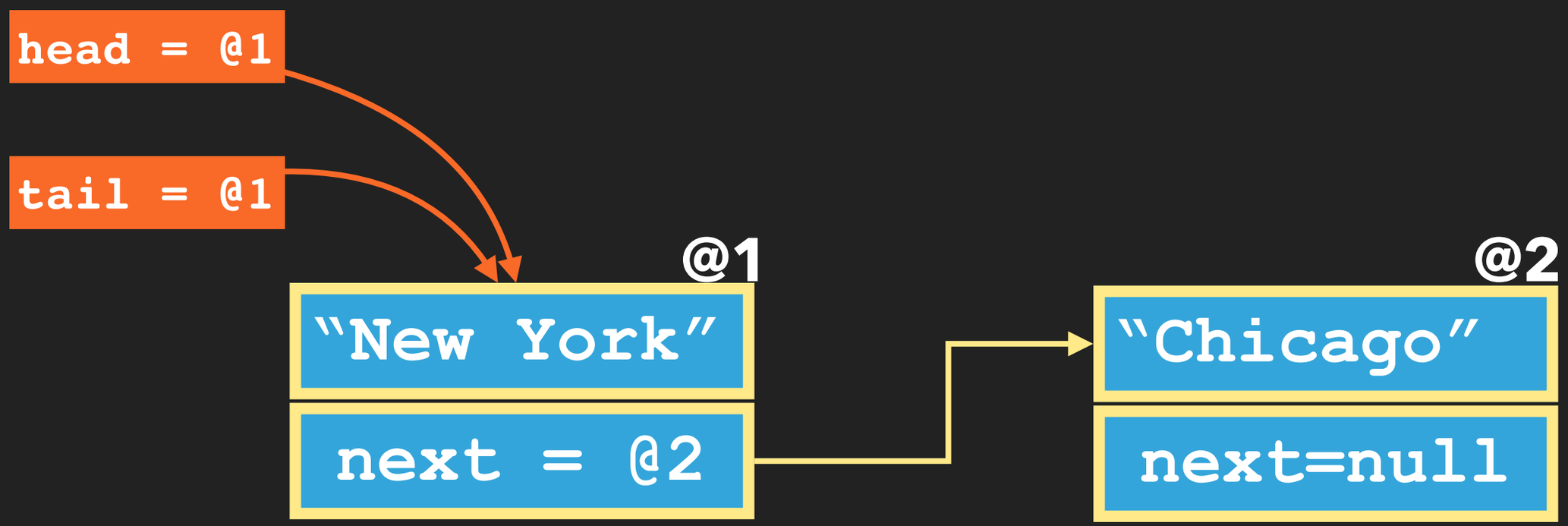
```
// Adding the first element
head = new Node("New York");
tail = head; size++;
```




One element in the list

# Linked List

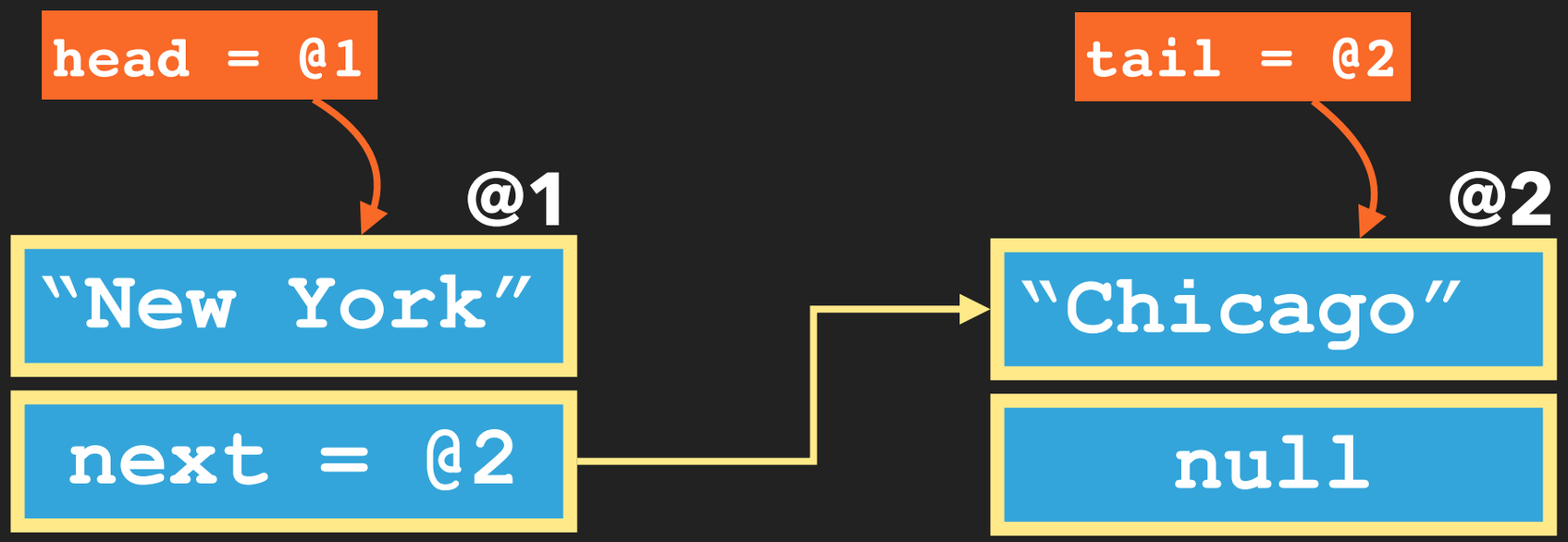
## Adding an element at the end - `addLast()`




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

# Linked List

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

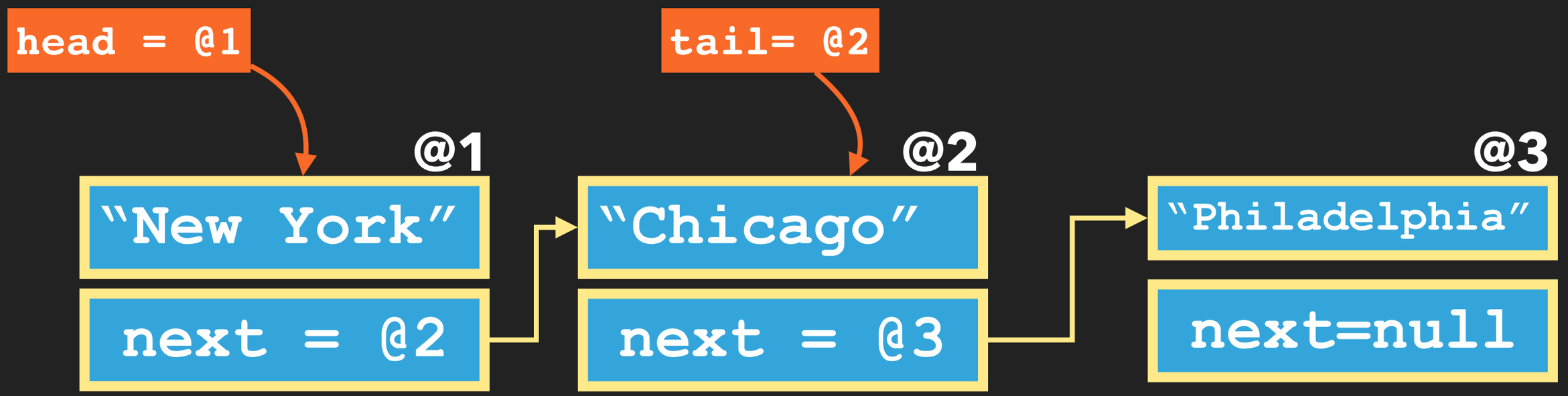



Two elements in the list

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

# Linked List

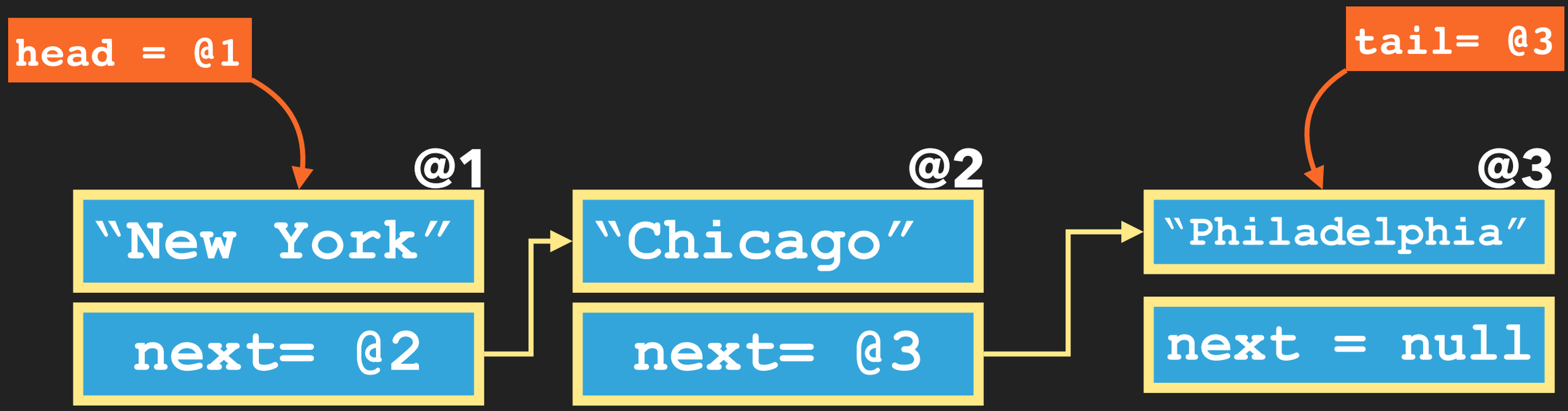
## Adding an element at the end - `addLast()`




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

# Linked List

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

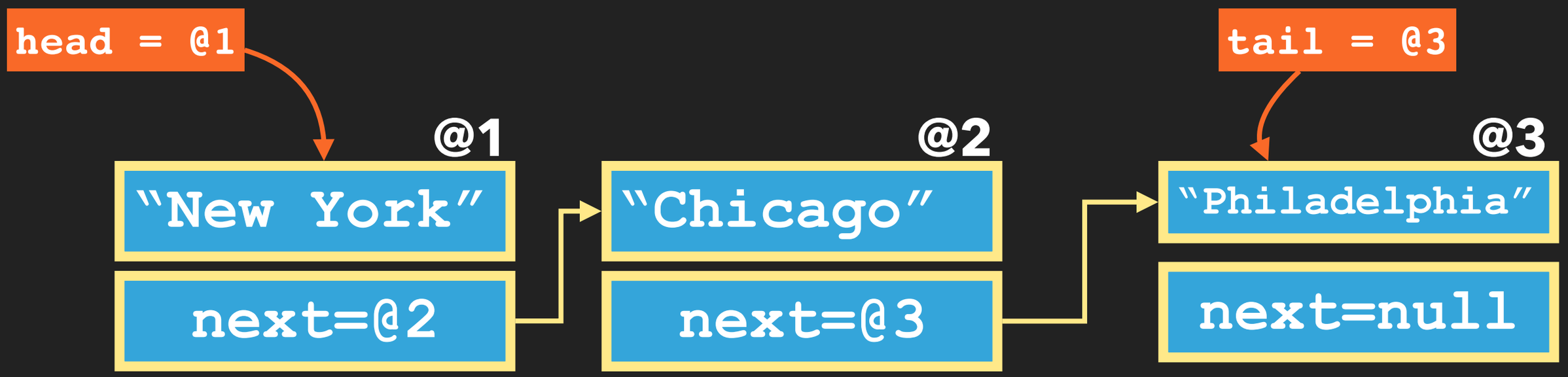



Three elements in the list

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

# Linked List

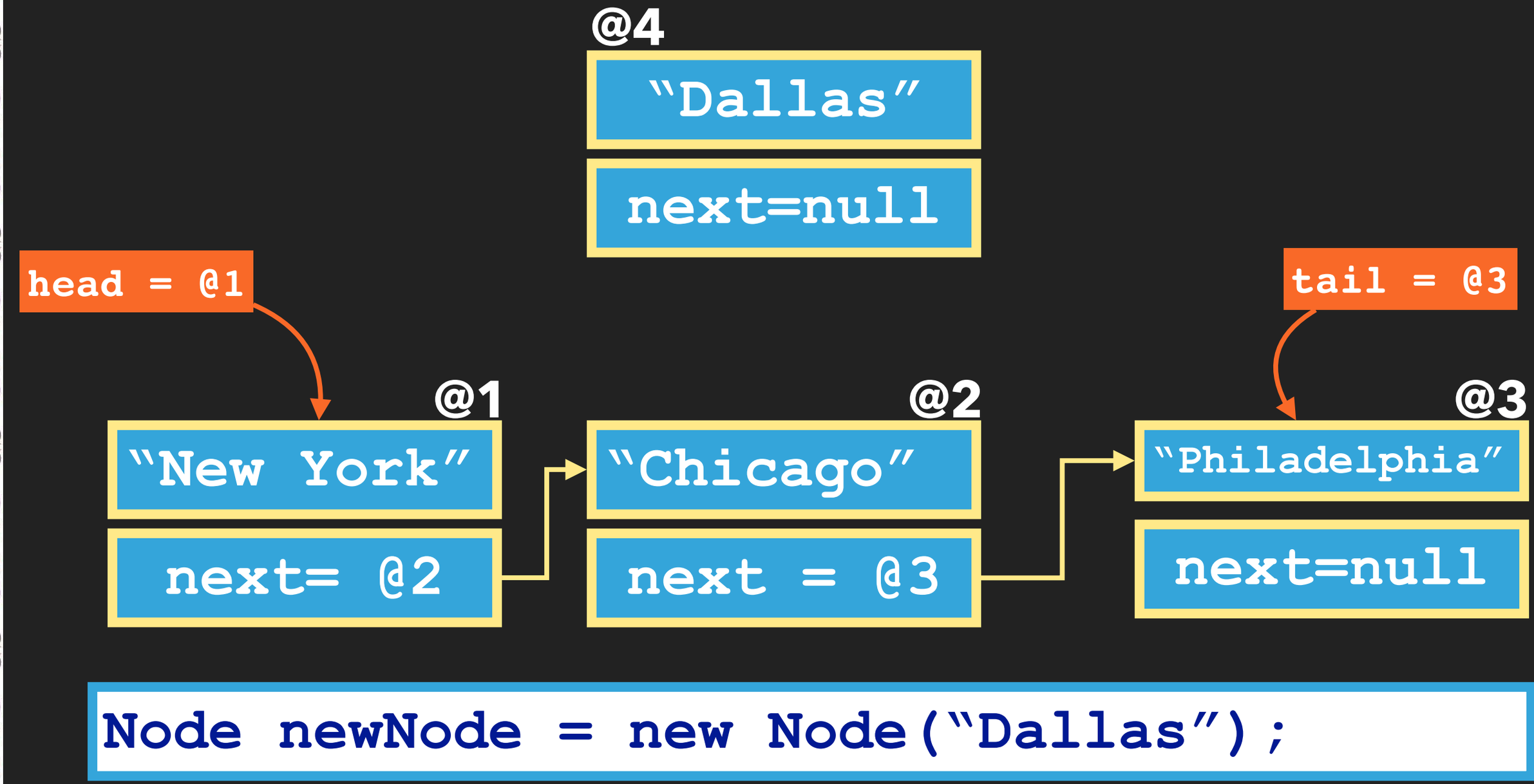
## List traversal




```
Node node = head;
while(node != null){
   System.out.println(node.value);
   node = node.next;
}
```

# Linked List

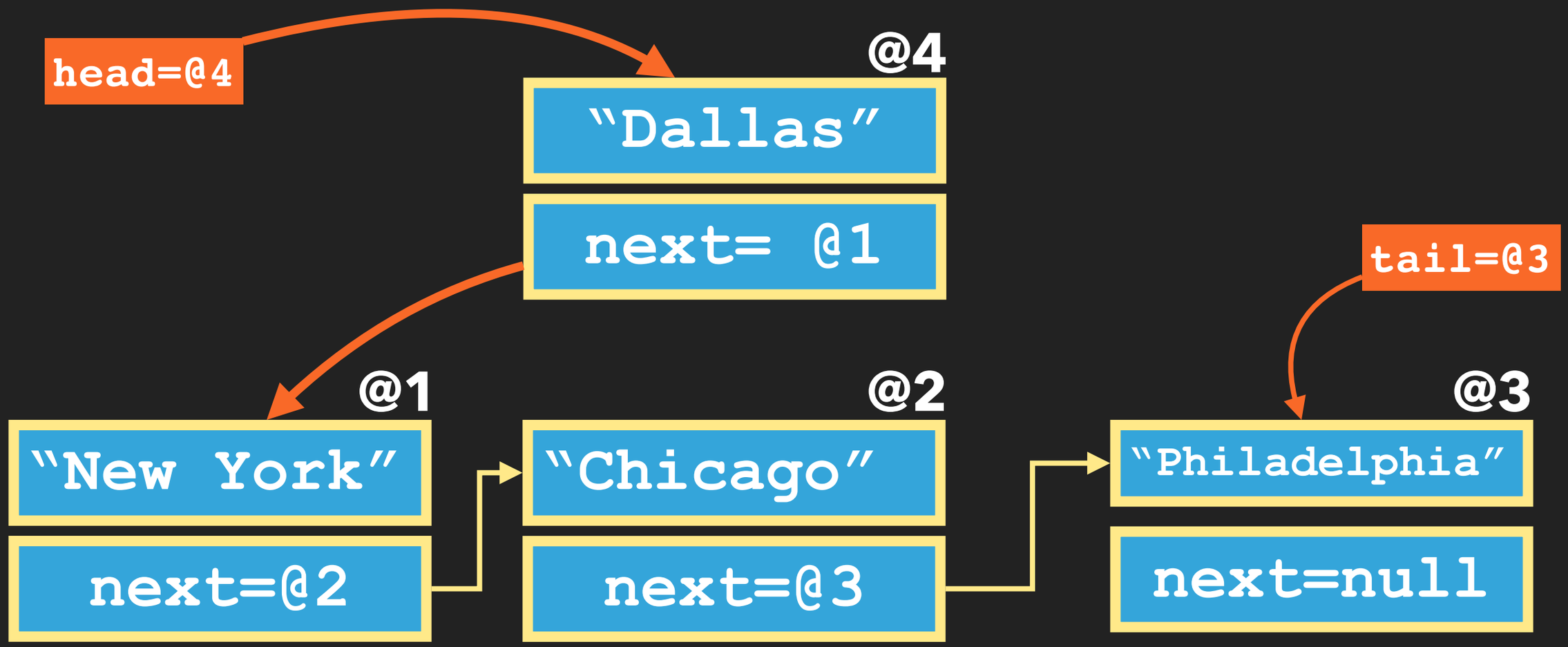
## Adding an element at the head - `addFirst()`




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

# Linked List

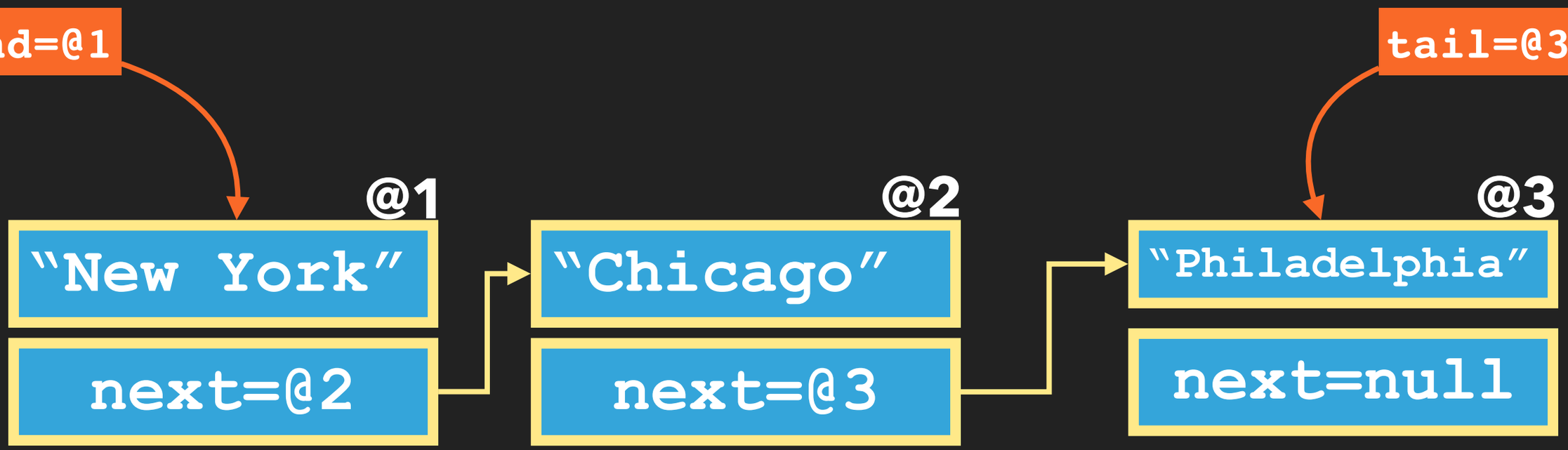
## Adding an element at the head - `addFirst()`




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

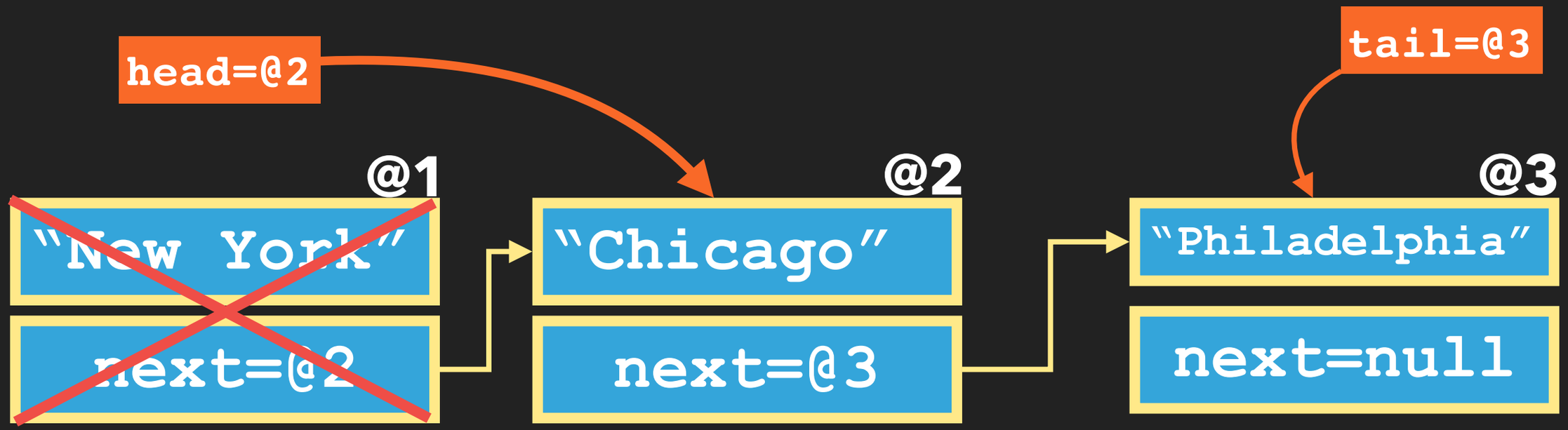
# Linked List

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



# Linked List

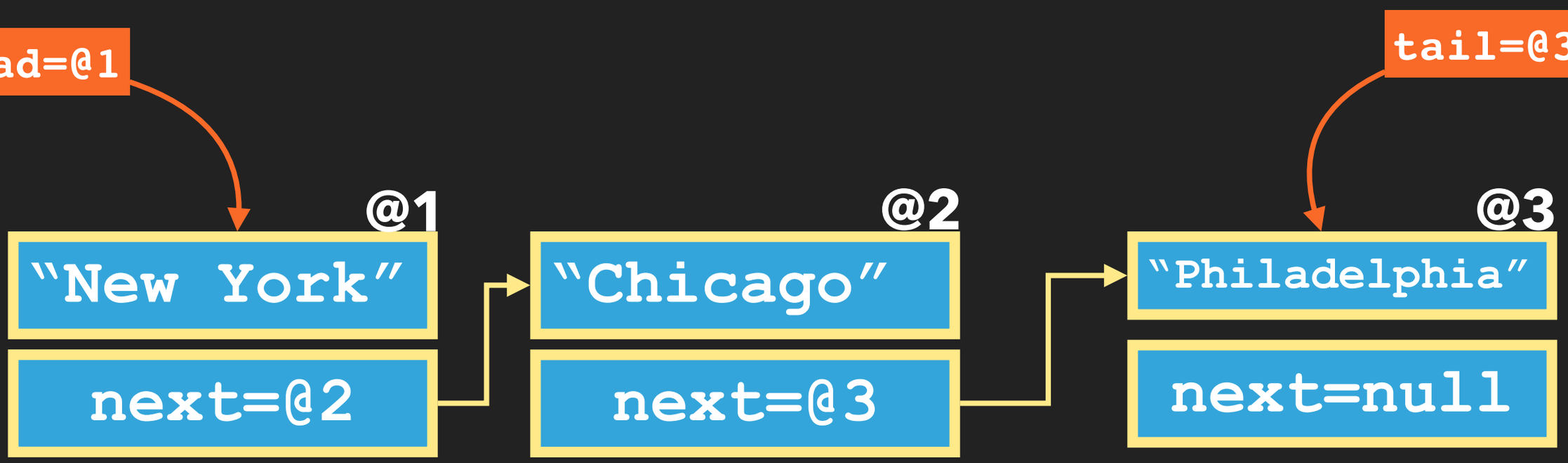
## Removing an element at the head- `removeFirst()`




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

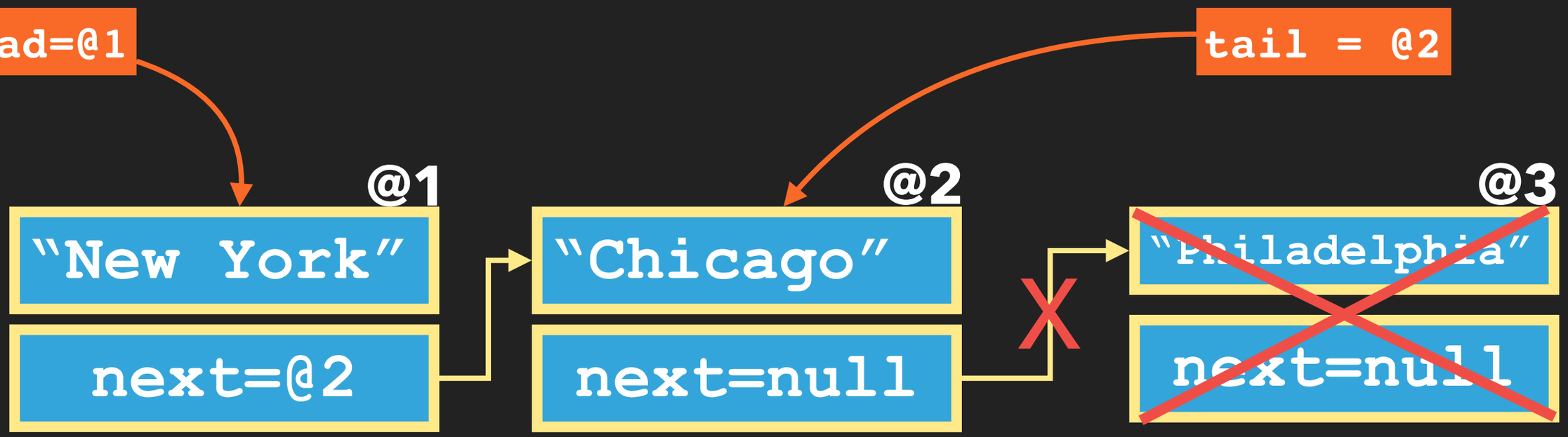
# Linked List

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



# Linked List

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




```
Previous of tail(@2).next= null;
tail = @2; size--;
```

# Linked List

LinkedList.java

```
public class LinkedList<E>{
   // Data members
   private Node head, tail;
   int size;
   // Inner class Node
   private class Node{
      E value;
      Node next;
      Node(E initialValue){
         value = initialValue; next = null;
      }
   }
   // Constructor
   public LinkedList() {
      head = tail = null;
      size = 0;
   }
```

# Linked List

LinkedList.java

```
// Adding an item to the list
public boolean addFirst(E item) {
     Node newNode = new Node(item);
     if(head == null) { head = tail = newNode; }
     else { newNode.next = head;
            head = newNode;
     }
     size++; return true;
}
public boolean addLast(E item) {
     Node newNode = new Node(item);
     if(head == null) { head = tail = newNode; }
     else { tail.next = newNode; tail = newNode; }
     size++; return true;
}
public boolean add(E item) {
     return addFirst(item);
}
```

# Linked List

LinkedList.java

```
// Retrieving an item from the list
public E getFirst() {
    if (head == null)
      throw new NoSuchElementException();
    return head.value;
}

public E getLast() {
    if (head == null)
      throw new NoSuchElementException();
    return tail.value;
}
```

# Linked List

LinkedList.java

```
// Removing an item from the list
public boolean removeFirst() {
     if (head == null) throw new NoSuchElementException();
     head = head.next;
     if(head == null) tail=null;
     size--; return true;
}
public boolean removeLast() {
     if (head == null) throw new NoSuchElementException();
     if(size == 1) return removeFirst();
     Node current = head;
     Node previous = null;
     while(current.next != null) {
       previous = current; current = current.next;
     }
     previous.next = null; tail = previous;
     size—; return true;
}
```

# Linked List

LinkedList.java

```
// toString() method
public String toString() {
    String output = "[";
    Node node = head;
    while(node != null) {
      output += node.value + " ";
      node = node.next;
    }
    output += "]";
    return output;
}
// clear, check if empty, and size of the list
public void clear() { head = tail = null; size = 0; }
public boolean isEmpty() { return (size == 0); }
public int size() { return size; }
```

# Linked List

LinkedList.java

```
// Generating an iterator for the list
public Iterator<E> iterator(){
      return new LinkedListIterator();
}
private class LinkedListIterator implements Iterator<E>{
    private Node current = head;
    public boolean hasNext() {
      return (current != null);
    }
    public E next() {
      if(current == null)
        throw new NoSuchElementException();
      E value = current.value;
      current = current.next; return value;
    }
}
```

# Linked List

Test.java

```
public class Test{
   public static void main(String[] args) {
    LinkedList<String> cityList;
    cityList = new LinkedList<>();
    cityList.addFirst("Boston");
    cityList.addFirst("Philadelphia");
    cityList.addFirst("San Francisco");
    cityList.addFirst("Washington");
    cityList.addFirst(“Portland");

    System.out.println(cityList.toString());

    Iterator<String> LLIterator = cityList.iterator();
    System.out.print("LinkedList (iterator): ");
    while(LLIterator.hasNext()) {
      System.out.print(LLIterator.next() + " ");
    }
  }
}
```

# Linked List

- What is the time complexity of the operations in LinkedList?

# Linked List

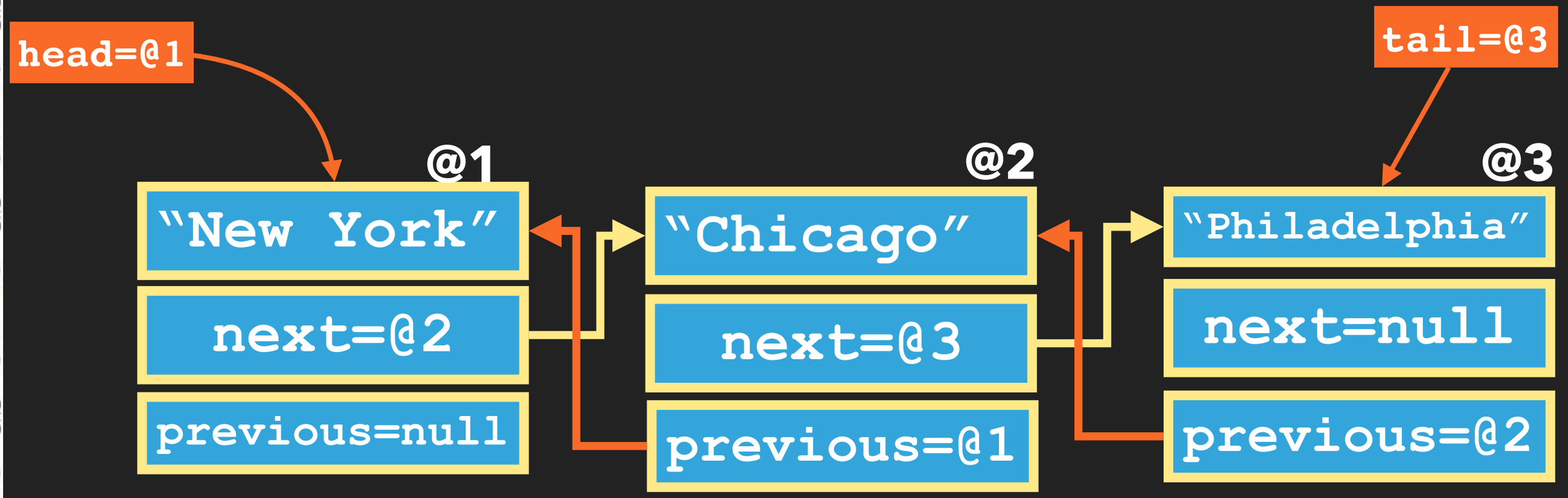
| Method | Complexity | Method | Complexity |
|---|---|---|---|
| `LinkedList()` | **O(1)** | `addFirst()` | **O(1)** |
| `size()` | **O(1)** | `addLast()` | **O(1)** |
| `clear()` | **O(1)** | `add(E)` | **O(1)** |
| `isEmpty()` | **O(1)** | `removeFirst()` | **O(1)** |
| `iterator()` | **O(1)** | `removeLast()` | **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 elements
  - Circular Linked List
    - Last element is linked back to the first element

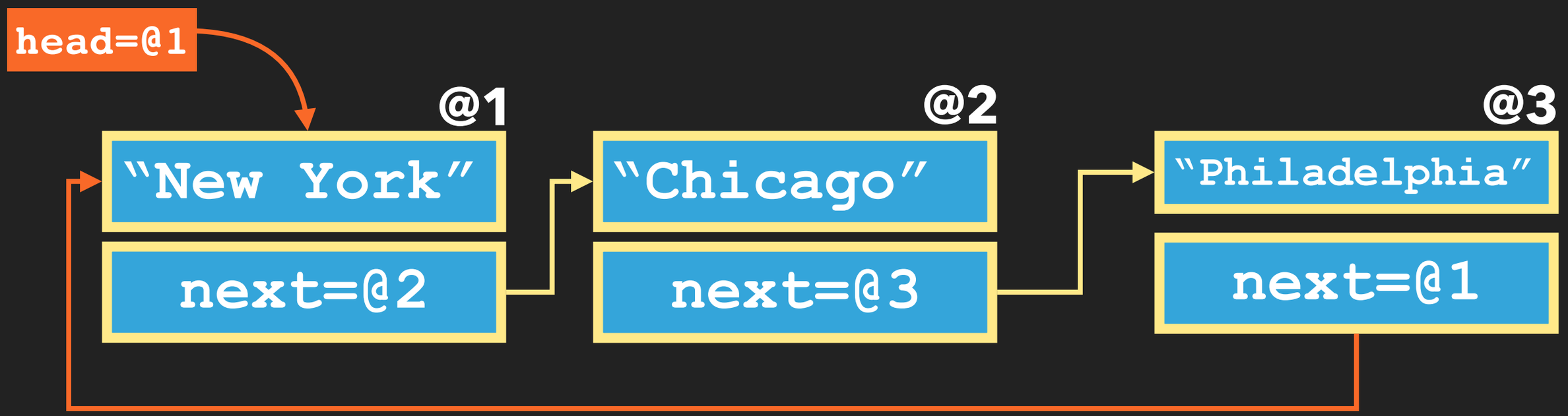
# Linked List

- 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 array based list with access only at the end of the list
- Queue is implemented using a linked list with access at the head and the tail

# Stack

Stack.java

```
public class Stack<E> {
   private ArrayList<E> elements;
   public Stack() { elements = new ArrayList<>();}
   public Stack(int capacity) {
      elements = new ArrayList<>(capacity);}
   public int size() {  return elements.size();}
   public boolean isEmpty() {return elements.isEmpty();}
   public void push(E item) {elements.add(item);}
   public E peek() {
      if(isEmpty())
         throw new EmptyStackException();
      return elements.get(size()-1);}
   public E pop() {
      if(isEmpty())
         throw new EmptyStackException();
      E value = peek();  elements.remove(size()-1);
      return value;}
   public String toString() {
      return "Stack: " + elements.toString();}
}
```

# Stack

Test.java

```
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() + " ");
```

```
City Stack (toString): Stack: [New York, San Diego, Atlanta, Baltimore, Pittsburg]
City Stack (pop): Pittsburg Baltimore Atlanta San Diego New York
```

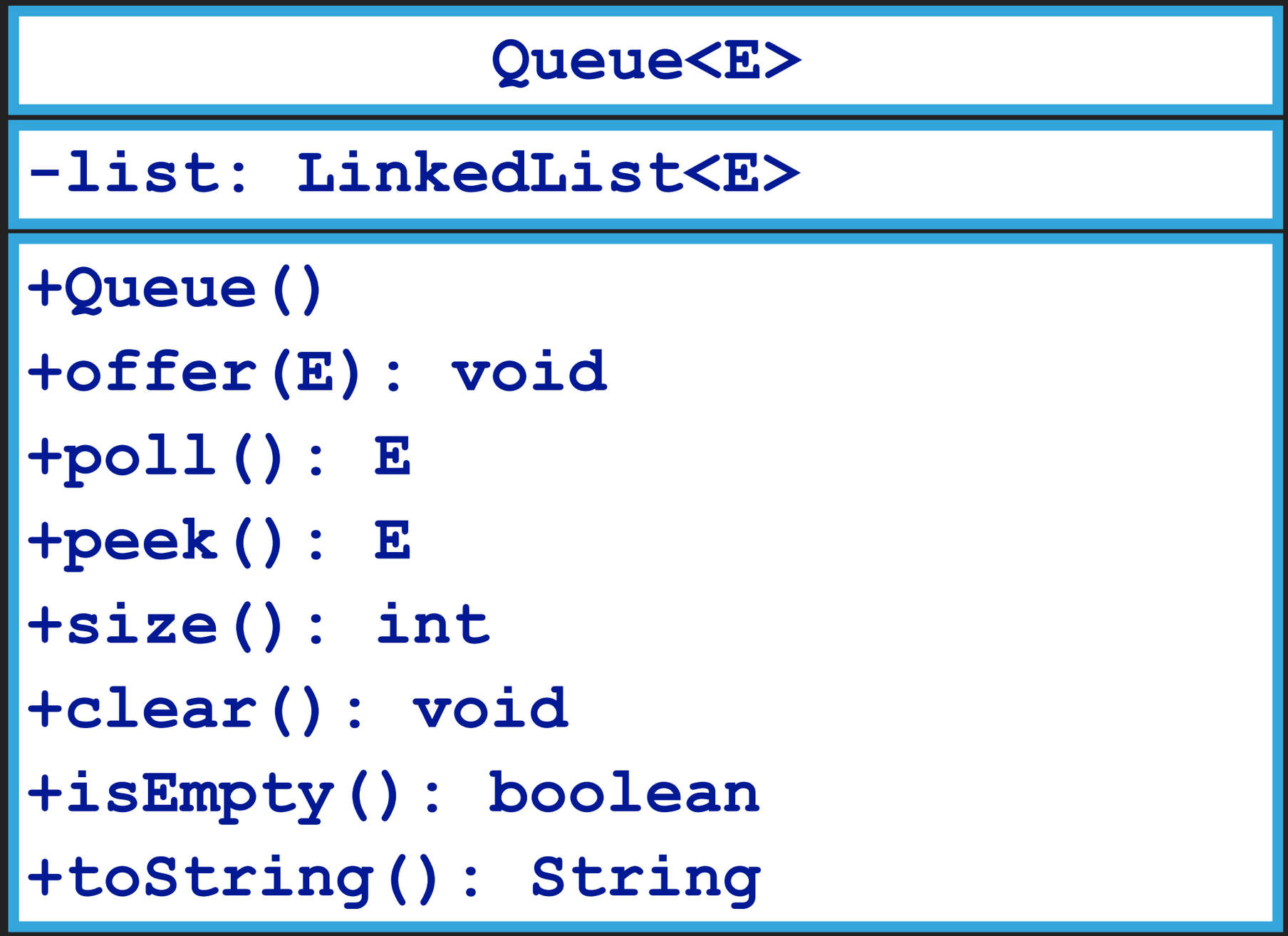
# Stack

- Performance of the 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 LinkedList



# Queue

Queue.java

```
public class Queue<E> {
  private LinkedList<E> list;
  public Queue(){ list=new LinkedList<>(); }
  public void offer(E item){ list.addLast(item); }
  public E poll(){
    E value = list.getFirst();
    list.removeFirst(); return value;
  }
  public E peek(){ return list.getFirst(); }
  public String toString(){
    return "Queue: " + list.toString();
  }
  public int size(){ return list.size(); }
  public void clear(){ list.clear(); }
  public boolean isEmpty(){ return list.size()==0; }
}
```

# Queue

Test.java

```
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() + " ");
```

```
City Queue (toString): Queue: [New York San Diego Atlanta Baltimore Pittsburg ]
City Queue (poll): New York San Diego Atlanta Baltimore Pittsburg
```

# Queue

- Performance of the 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

PriorityQueue.java

```
public class PriorityQueue<E> {
 private ArrayList<E> list;
 private Comparator<E> comparator;
 public PriorityQueue() {
     list = new ArrayList<>();
     comparator = null; }
public PriorityQueue(Comparator<E> c) {
     list = new ArrayList<>();
     comparator = c;
}
public E poll() {
    E value = list.get(0);
    list.remove(0); return value;
}
```

# Priority Queue

PriorityQueue.java

```
public void offer(E item) {
 int i, c;
 for(i=0; i<list.size(); i++){
  if(comparator == null)
   c = ((Comparable<E>)item).compareTo(list.get(i));
  else
   c = comparator.compare(item, list.get(i));
  if(c < 0)
    break;
 }
 list.add(i, item);
}
```

# Priority Queue

PriorityQueue.java

```
public E peek() {
 return list.get(0);
}

public String toString() {
    return "Priority Queue: " + list.toString();
 }

 public int size() { return list.size(); }
 public void clear() { list.clear(); }
 public boolean isEmpty() { return list.size() == 0;}
}
```

# Priority Queue

Test.java

```
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() + " ");
}
```

```
City Priority Queue: Priority Queue: [Atlanta Baltimore New York Pittsburg San Diego
City Priority Queue (poll): Atlanta Baltimore New York Pittsburg San Diego
```

# Priority Queue

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