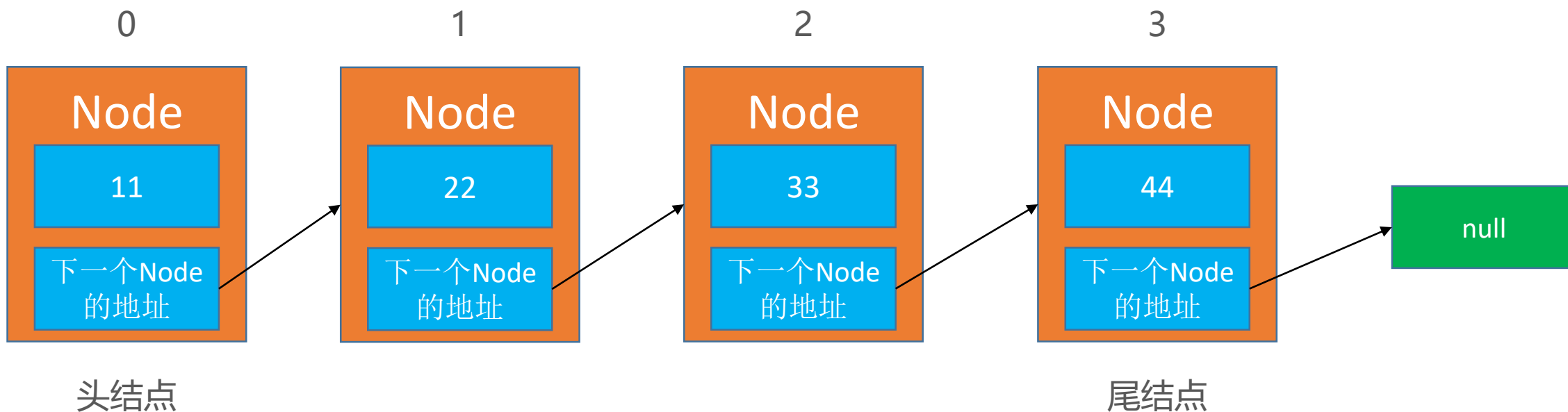


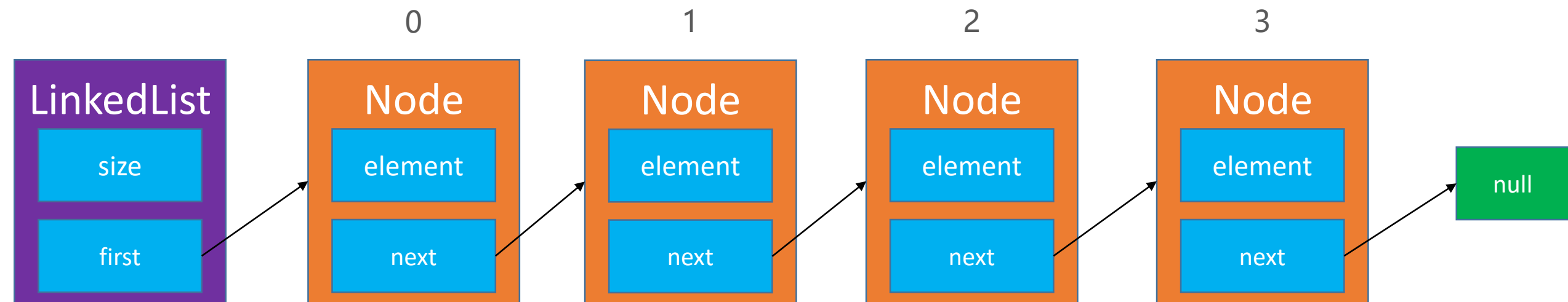
链表

链表 (Linked List)

- 动态数组有个明显的缺点
 - 可能会造成内存空间的大量浪费
- 能否用到多少就申请多少内存?
 - 链表可以办到这一点
- 链表是一种链式存储的线性表，所有元素的内存地址不一定是连续的

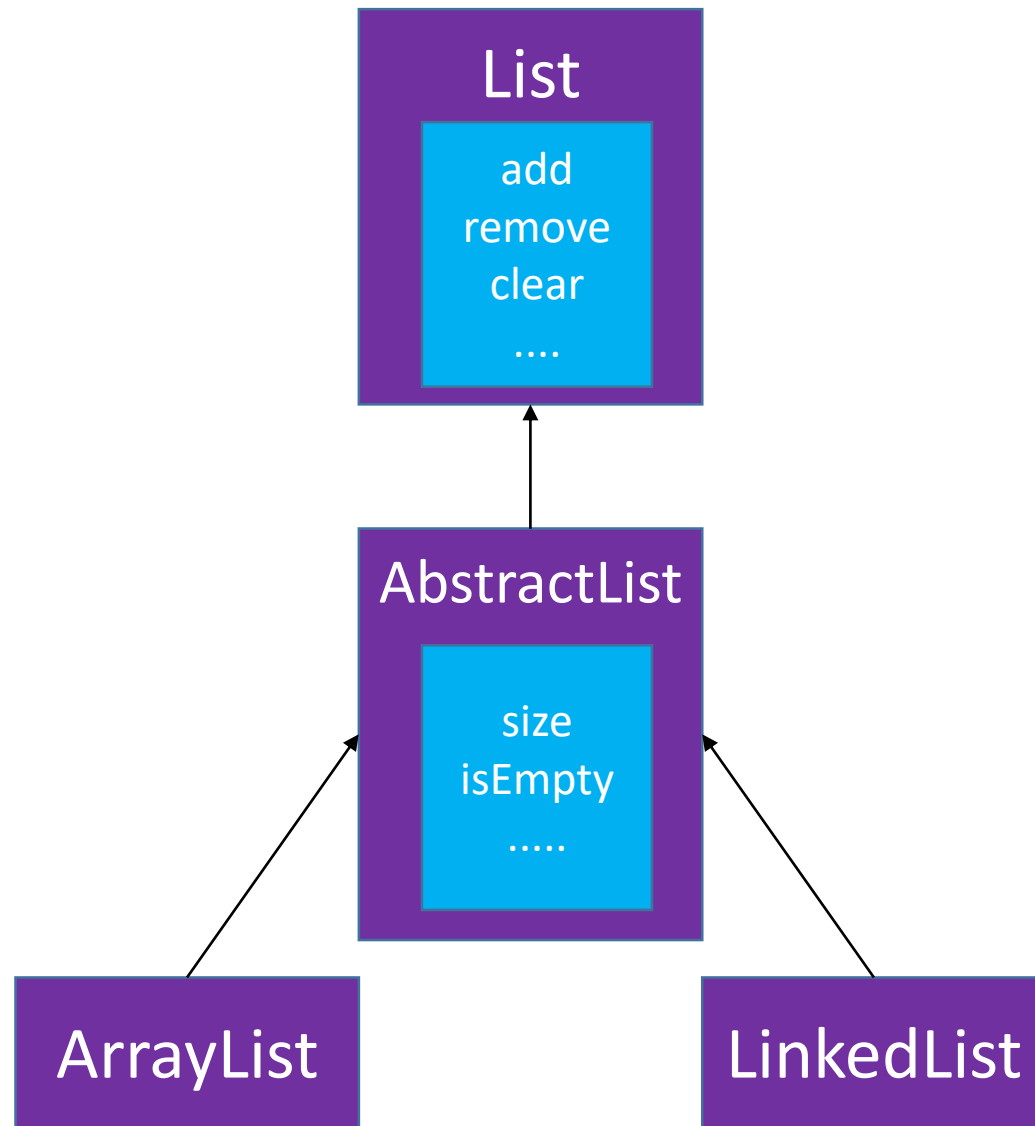
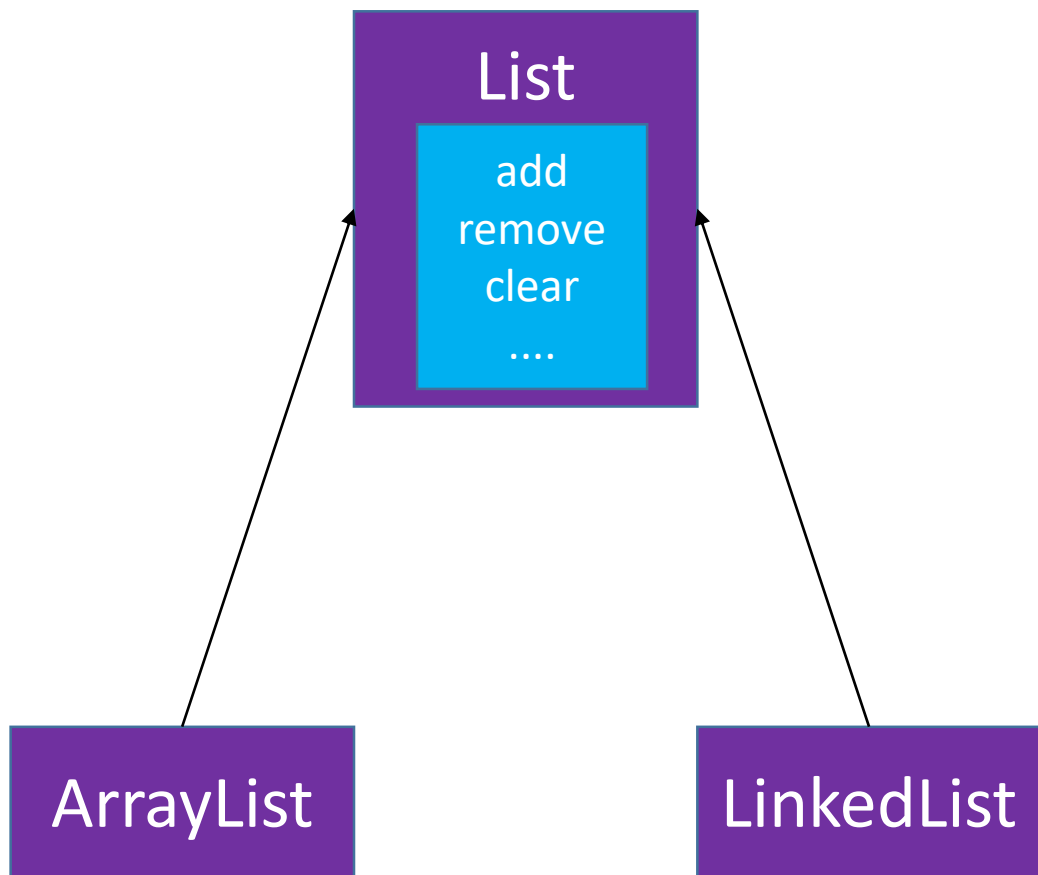


链表的设计

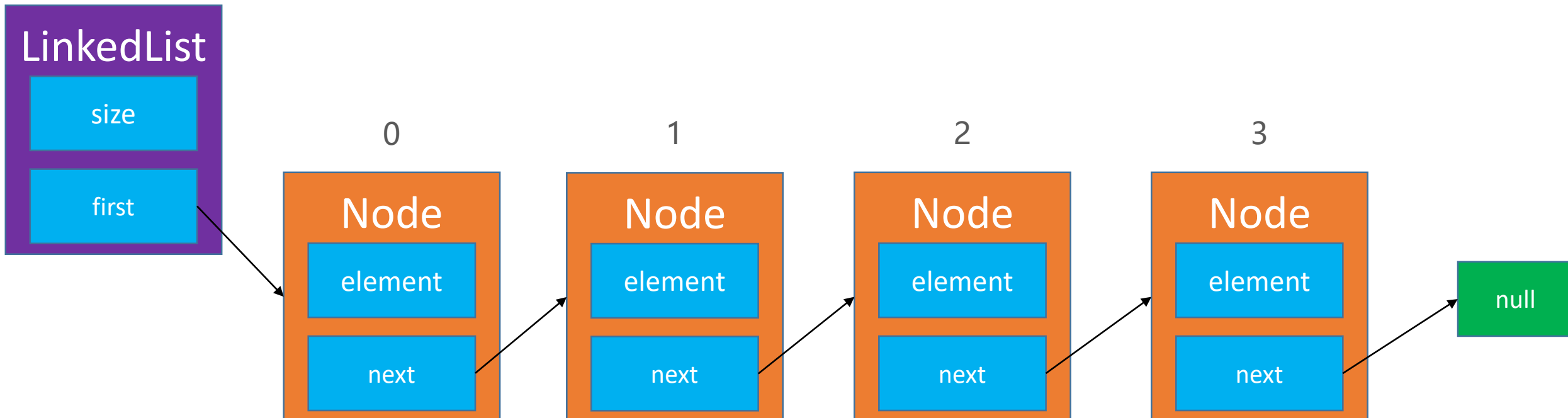


接口设计

- 链表的大部分接口和动态数组是一致的



清空元素 – clear()

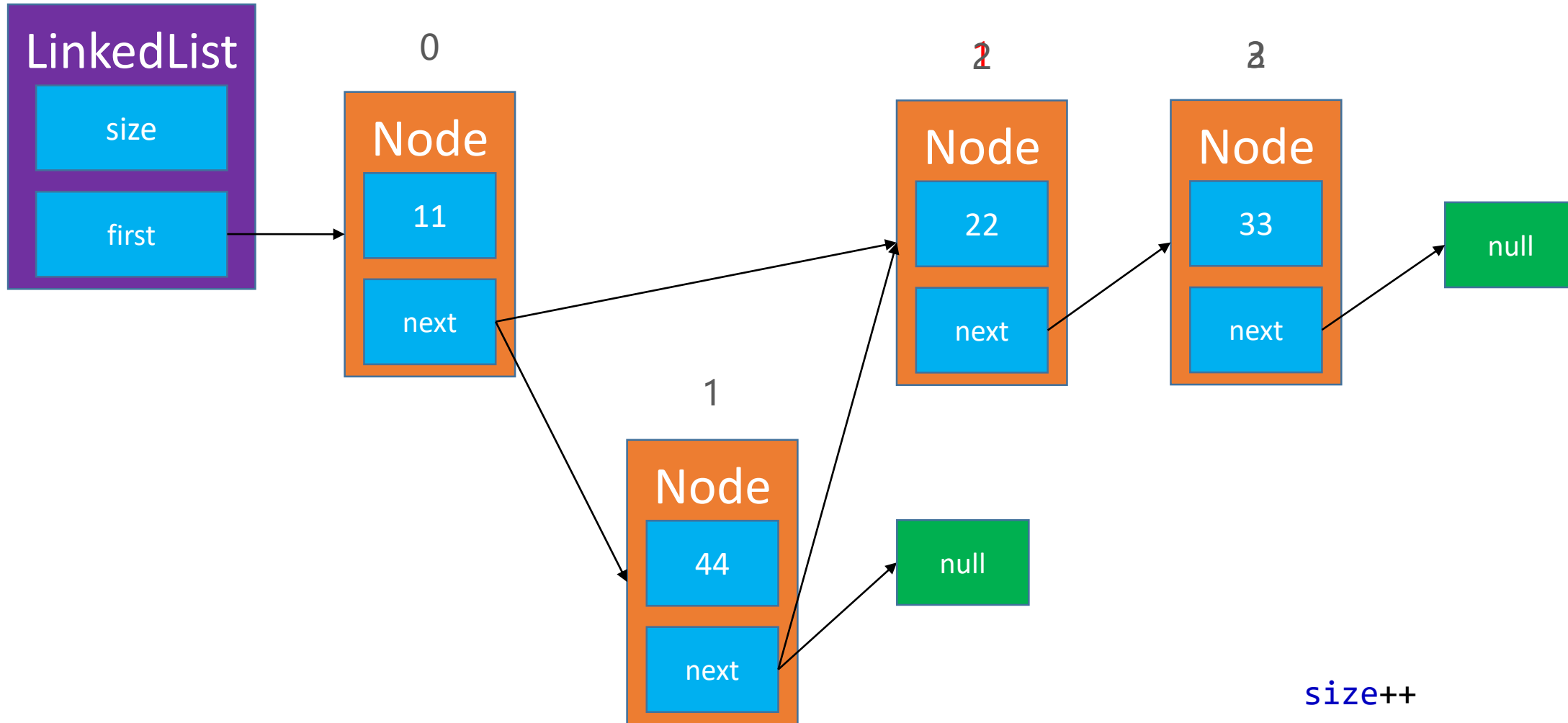


`size = 0;`

`first = null;`

思考: `next` 需要设置为 `null` 么?

添加元素 - add(int index, E element)



node方法用于获取index位置的节点

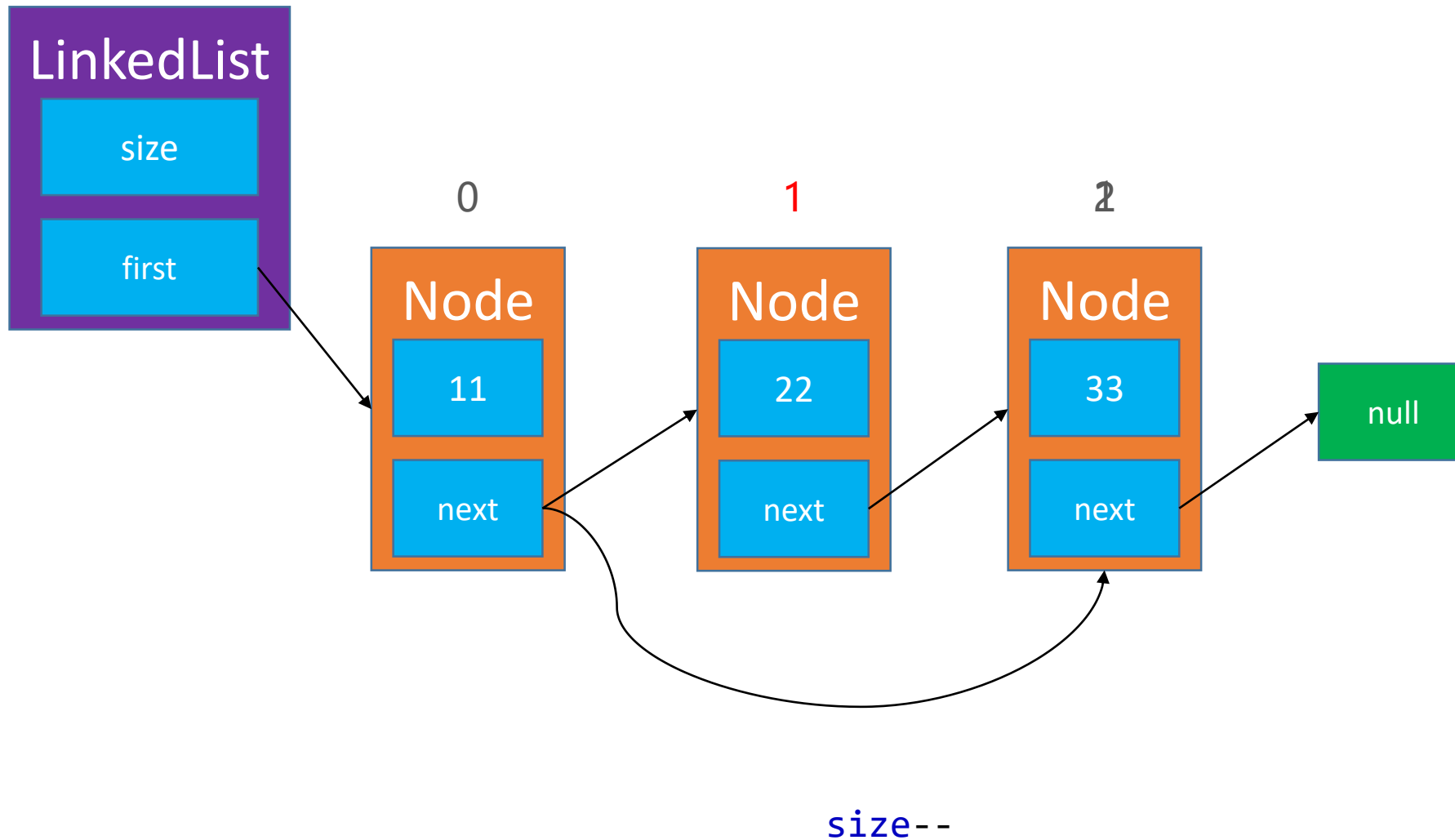
```
private Node<E> node(int index) {  
    rangeCheck(index);  
  
    Node<E> node = first;  
    for (int i = 0; i < index; i++) {  
        node = node.next;  
    }  
    return node;  
}
```

添加元素 – 注意0位置

```
public void add(int index, E element) {  
    rangeCheck(index);  
  
    if (index == 0) {  
        first = new Node<>(element, first);  
    } else {  
        Node<E> prev = node(index - 1);  
        prev.next = new Node<>(element, prev.next);  
    }  
  
    size++;  
}
```

在编写链表过程中，要注意边界测试，比如 `index` 为 `0`、`size - 0`、`size` 时

删除元素 - remove(int index)



删除元素 – 注意0位置

```
public E remove(int index) {  
    rangeCheck(index);  
  
    Node<E> node = first;  
    if (index == 0) {  
        first = first.next;  
    } else {  
        Node<E> prev = node(index - 1);  
        node = prev.next;  
        prev.next = node.next;  
    }  
  
    size--;  
  
    return node.element;  
}
```

推荐一个神奇的网站

■ <https://visualgo.net/zh>



练习 – 删除链表中的节点

■ <https://leetcode-cn.com/problems/delete-node-in-a-linked-list/>

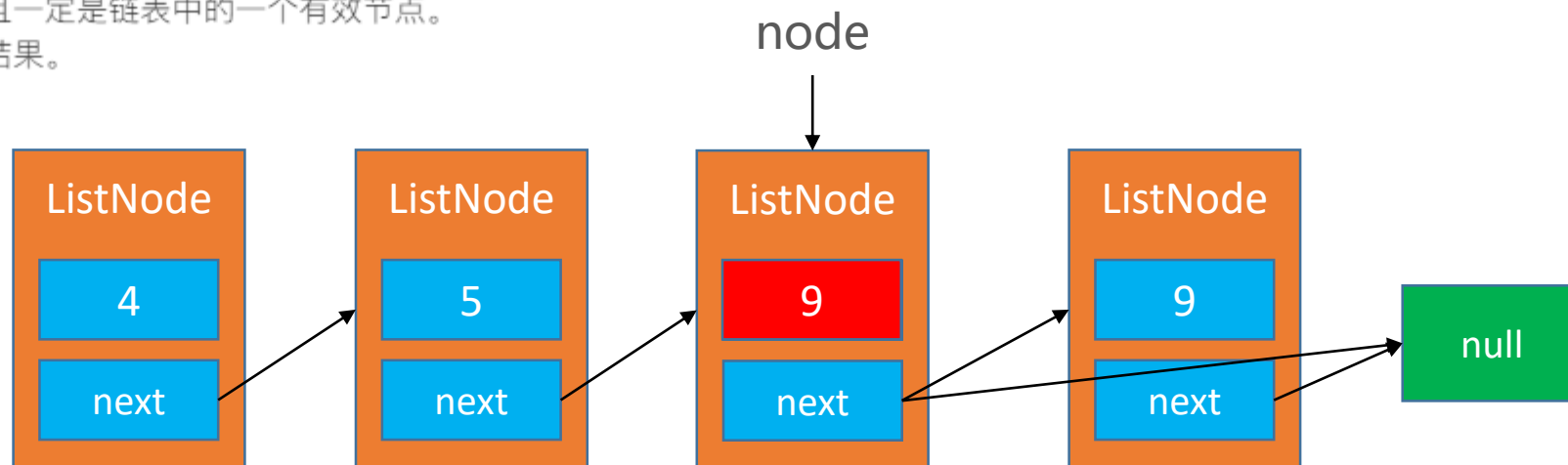
输入: head = [4,5,1,9], node = 1

输出: [4,5,9]

解释: 给定你链表中值为 1 的第三个节点, 那么在调用了你的函数之后, 该链表应变为 4 -> 5 -> 9.

说明:

- 链表至少包含两个节点。
- 链表中所有节点的值都是唯一的。
- 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
- 不要从你的函数中返回任何结果。



练习 – 反转一个链表

- <https://leetcode-cn.com/problems/reverse-linked-list/>

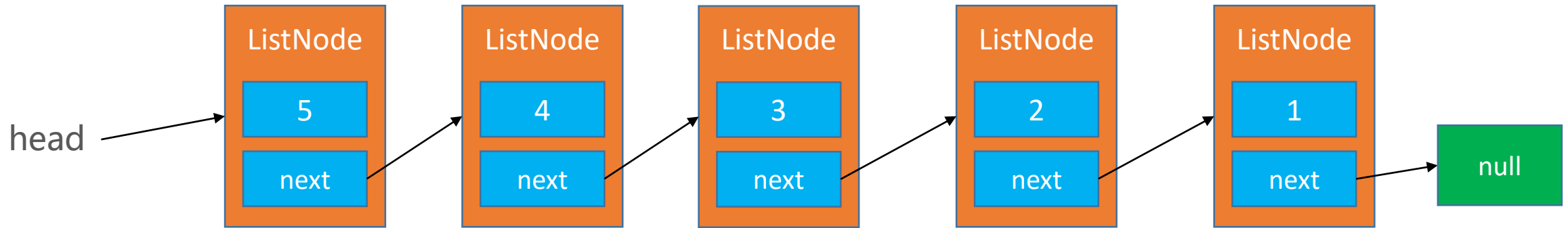
示例:

输入: 1->2->3->4->5->NULL

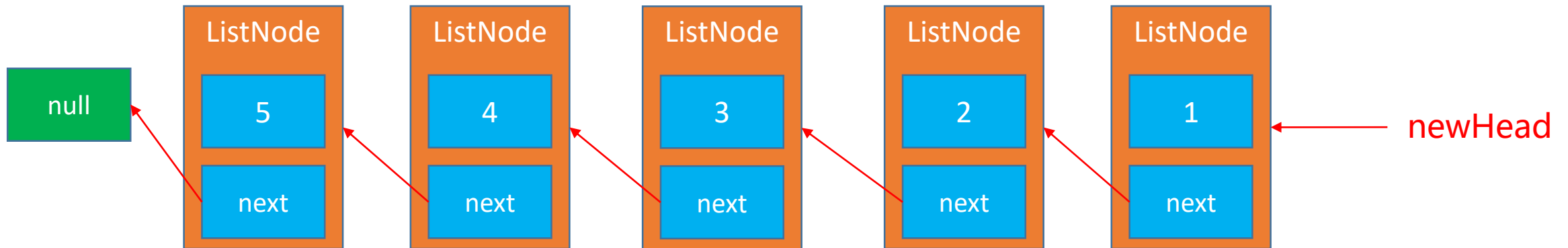
输出: 5->4->3->2->1->NULL

- 请分别用递归、迭代（非递归）两种方式实现

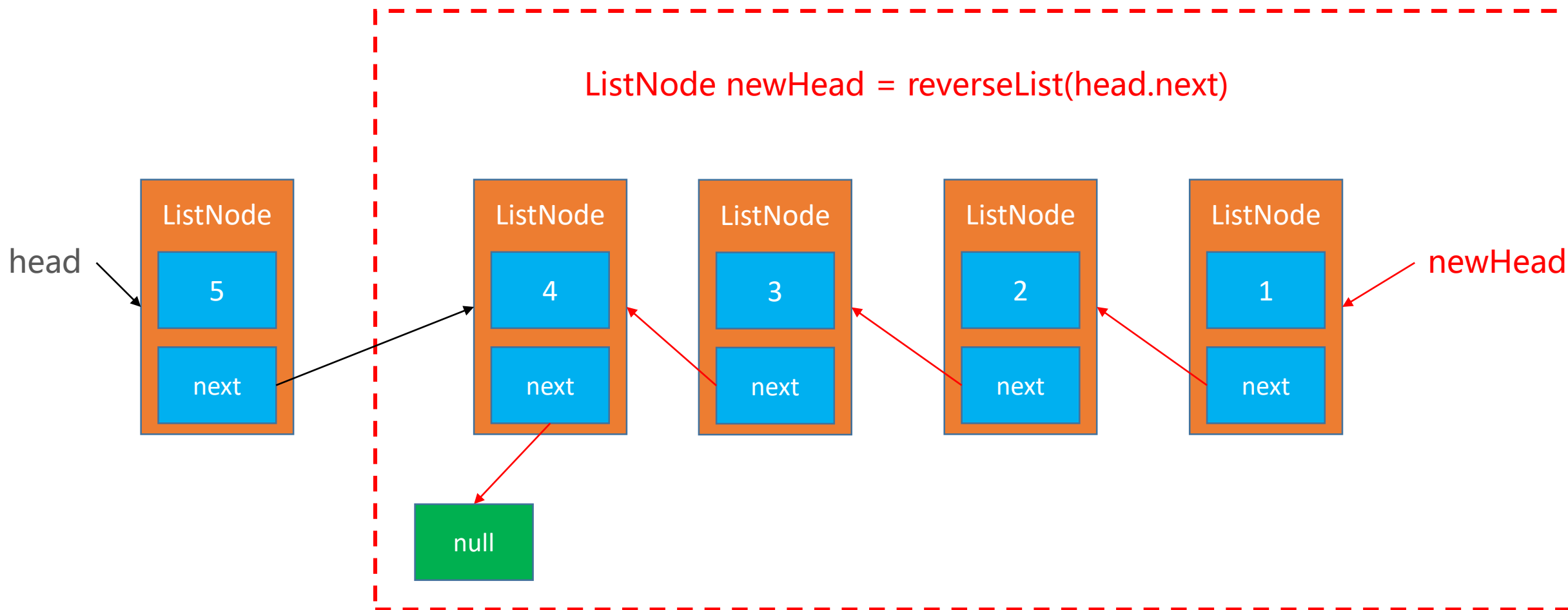
练习 - 反转一个链表 - 递归



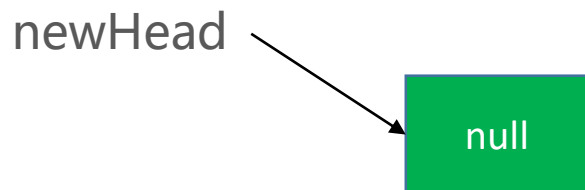
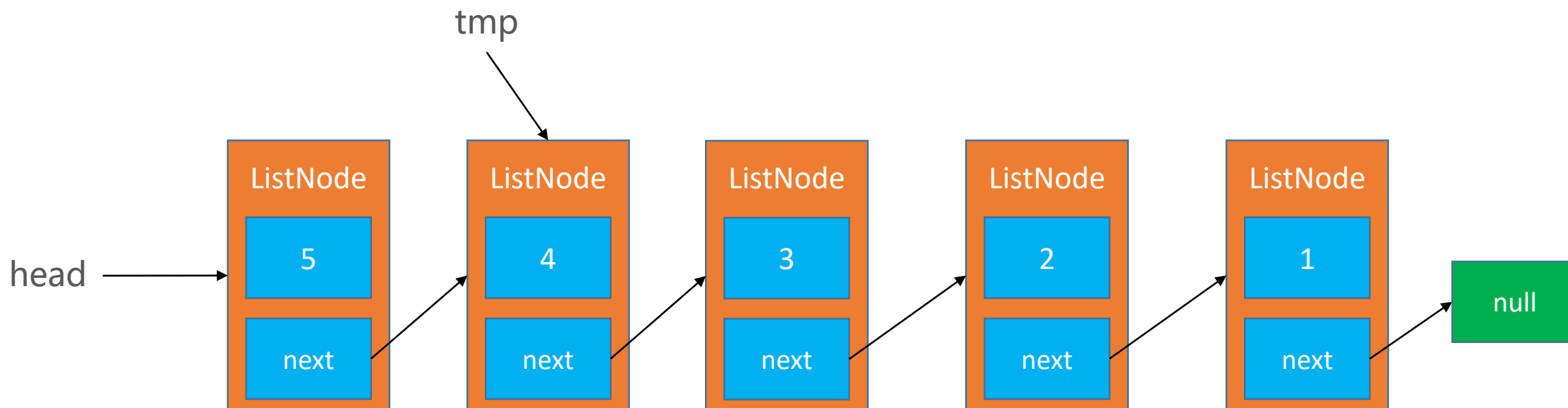
`ListNode newHead = reverseList(head)`



练习 - 反转一个链表 - 递归



练习 - 反转一个链表 - 非递归 - 头插法



```
ListNode newHead = null;
while (head != null) {
    ListNode tmp = head.next;
    head.next = newHead;
    newHead = head;
    head = tmp;
}
```


练习 – 判断一个链表是否有环

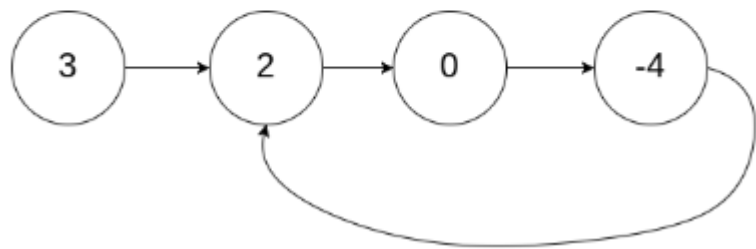
■ <https://leetcode-cn.com/problems/linked-list-cycle/>

示例 1:

输入: head = [3,2,0,-4], pos = 1

输出: true

解释: 链表中有一个环, 其尾部连接到第二个节点。

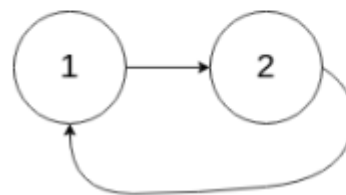


示例 2:

输入: head = [1,2], pos = 0

输出: true

解释: 链表中有一个环, 其尾部连接到第一个节点。



示例 3:

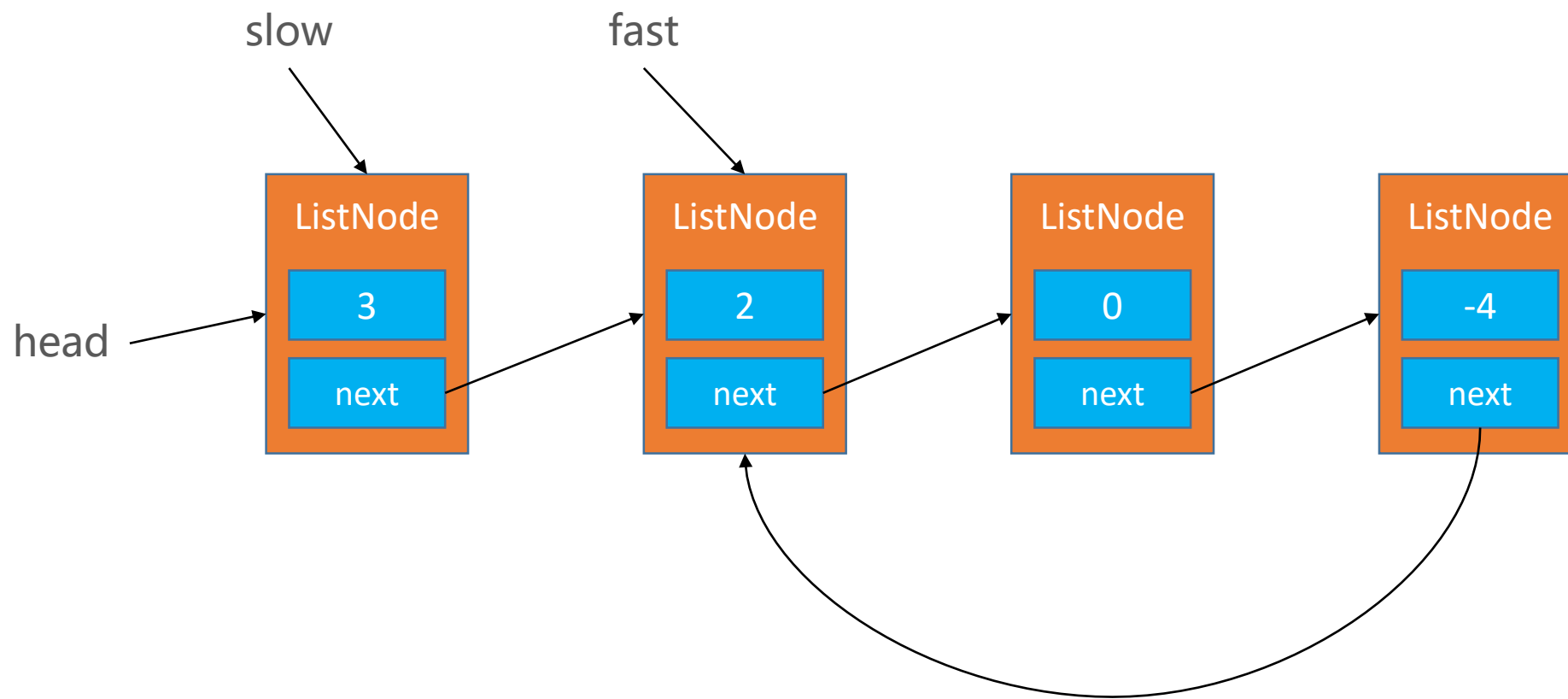
输入: head = [1], pos = -1

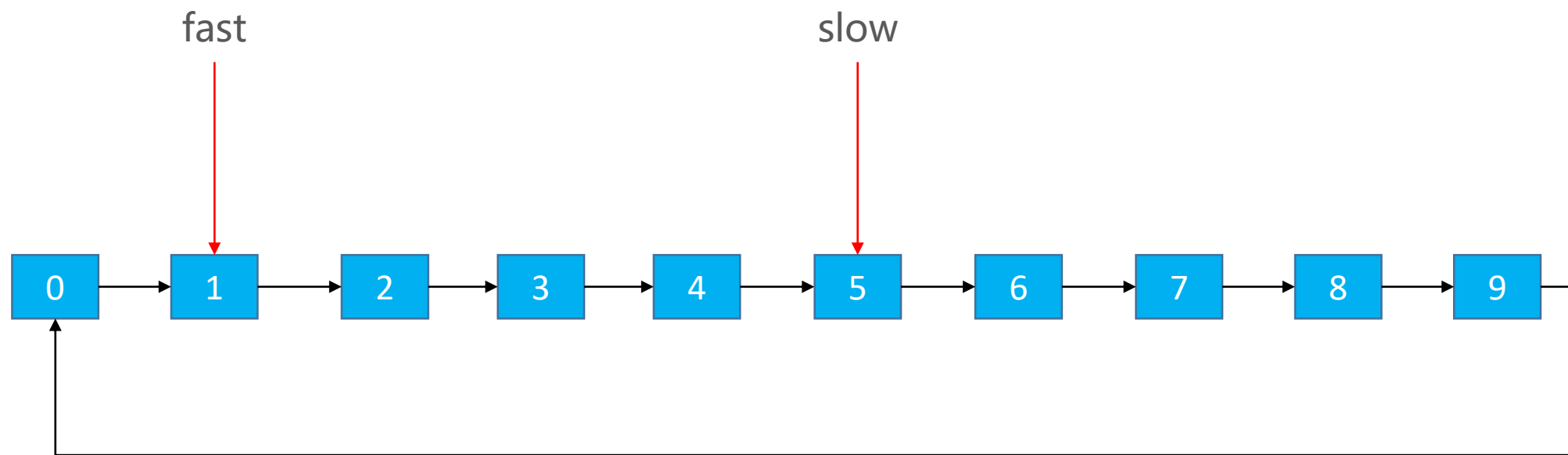
输出: false

解释: 链表中没有环。



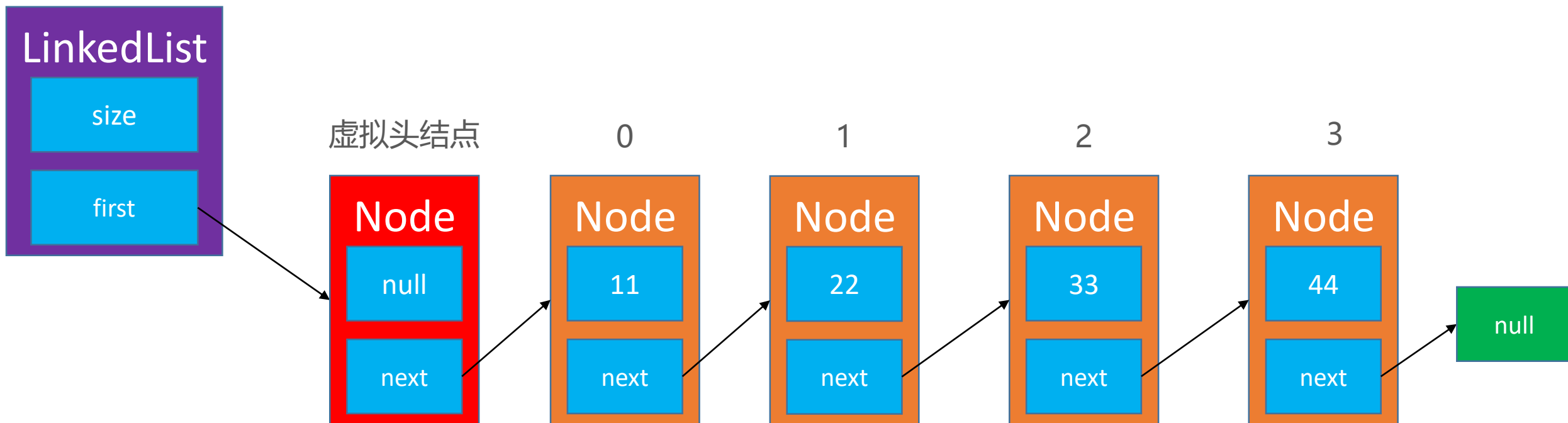
练习 – 判断一个链表是否有环





虚拟头结点

- 有时候为了让代码更加精简，统一所有节点的处理逻辑，可以在最前面增加一个虚拟的头结点（不存储数据）



```
public LinkedList() {  
    first = new Node<>(null, null);  
}
```

虚拟节点 – node方法

```
private Node<E> node(int index) {  
    rangeCheck(index);  
  
    Node<E> node = first.next;  
    for (int i = 0; i < index; i++) {  
        node = node.next;  
    }  
    return node;  
}
```

虚拟节点 – 添加、删除

```
public void add(int index, E element) {  
    rangeCheckForAdd(index);  
  
    Node<E> prev = (index == 0) ? first : node(index - 1);  
    prev.next = new Node<>(element, prev.next);  
  
    size++;  
}
```

```
public E remove(int index) {  
    rangeCheck(index);  
  
    Node<E> prev = (index == 0) ? first : node(index - 1);  
    Node<E> node = prev.next;  
    prev.next = node.next;  
  
    size--;  
    return node.element;  
}
```

复杂度分析

- 最好情况复杂度
- 最坏情况复杂度
- 平均情况复杂度

数组的随机访问

索引	元素
0	11
1	22
2	33
3	44
4	55
5	66

- 数组的随机访问速度非常快
- `elements[n]`的效率与n是多少无关

动态数组、链表复杂度分析

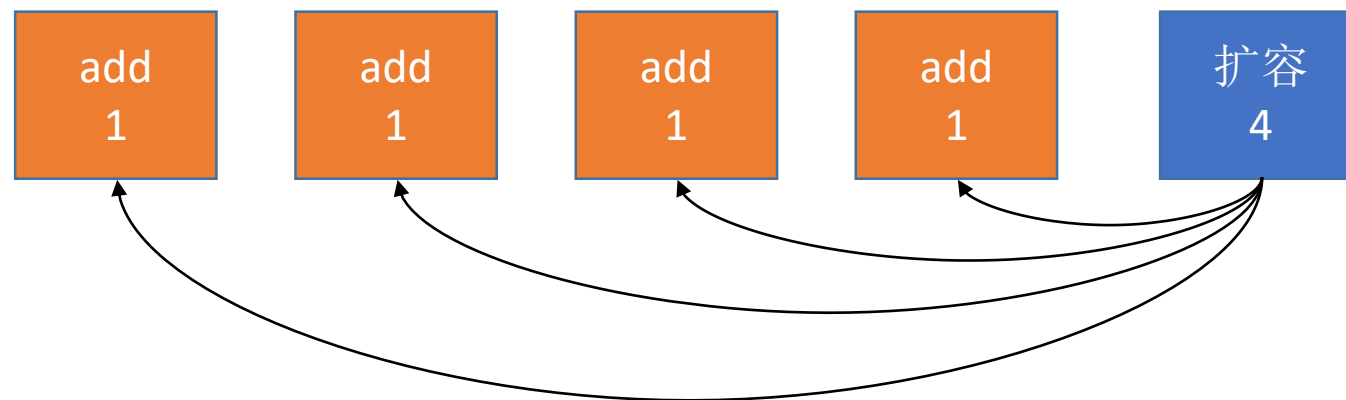
	动态数组			链表		
	最好	最坏	平均	最好	最坏	平均
add(int index, E element)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
remove(int index)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)
set(int index, E element)	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)
get(int index)	O(1)	O(1)	O(1)	O(1)	O(n)	O(n)

size 是数组规模 n

动态数组add(E element)复杂度分析

- 最好: $O(1)$
- 最坏: $O(n)$
- 平均: $O(1)$
- 均摊: $O(1)$

假设最大容量是4



相当于每次 add 的操作次数是 2, 也就是 $O(1)$ 复杂度

- 什么情况下适合使用均摊复杂度
- 经过连续的多次复杂度比较低的情况后, 出现个别复杂度比较高的情况

动态数组的缩容

- 如果内存使用比较紧张，动态数组有较多的剩余空间，可以考虑进行缩容操作
 - 比如剩余空间占总容量的一半时，就进行缩容
- 如果扩容倍数、缩容时机设计不得当，有可能会引起复杂度震荡

11	22	33	44
----	----	----	----

双向链表

- 此前所学的链表，也叫做单向链表
- 使用双向链表可以提升链表的综合性能

LinkedList

size

first

last

Node

element

prev

next

0

Node

element

prev

next

1

Node

element

prev

next

2

Node

element

prev

next

3

Node

element

prev

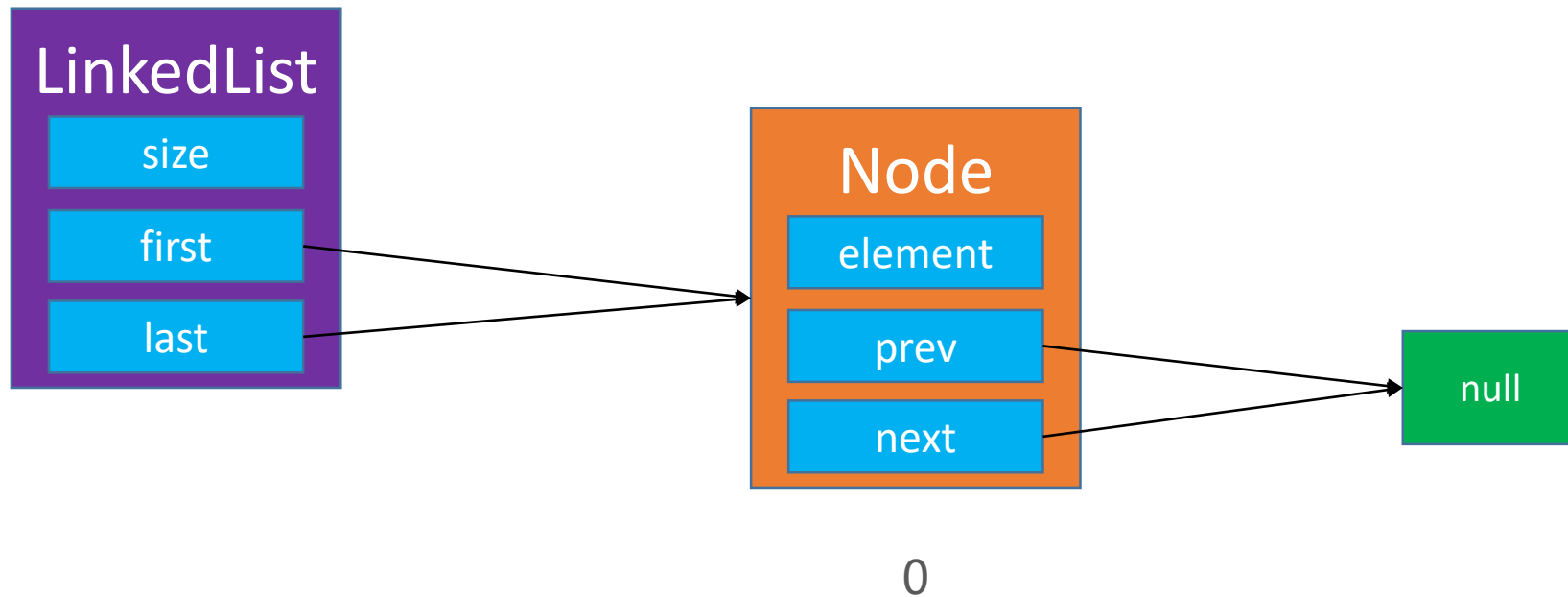
next

4

null

null

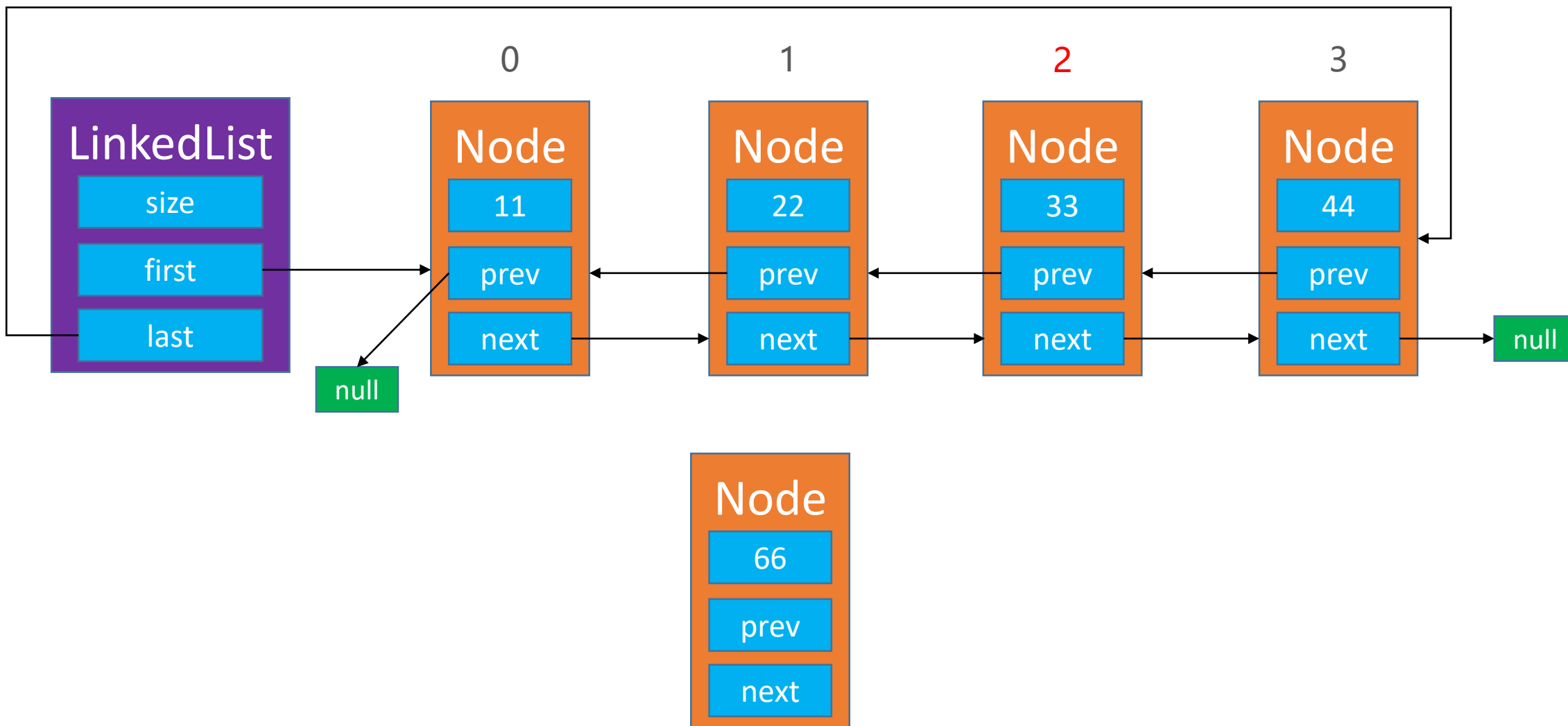
双向链表 - 只有一个元素



双向链表 – node方法

```
private Node<E> node(int index) {  
    rangeCheck(index);  
  
    if (index < (size >> 1)) {  
        Node<E> node = first;  
        for (int i = 0; i < index; i++) {  
            node = node.next;  
        }  
        return node;  
    } else {  
        Node<E> node = last;  
        for (int i = size - 1; i > index; i--) {  
            node = node.prev;  
        }  
        return node;  
    }  
}
```

双向链表 – add(int index, E element)



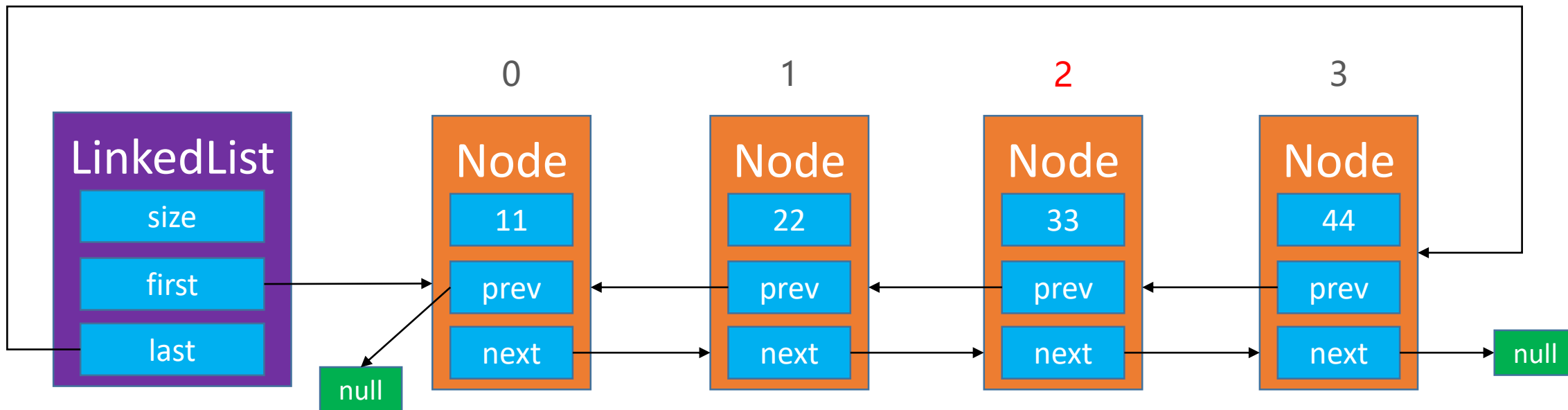
双向链表 – add(int index, E element)

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    if (index == size) {
        Node<E> oldLast = last;
        last = new Node<>(oldLast, element, null);
        if (oldLast == null) {
            first = last;
        } else {
            oldLast.next = last;
        }
    } else {
        Node<E> next = node(index);
        Node<E> prev = next.prev;
        Node<E> current = new Node<>(prev, element, next);
        next.prev = current;

        if (prev == null) {
            first = current;
        } else {
            prev.next = current;
        }
    }
    size++;
}
```


双向链表 – remove(int index)



双向链表 – remove(int index)

```
public E remove(int index) {  
    rangeCheck(index);  
  
    Node<E> node = node(index);  
    Node<E> prev = node.prev;  
    Node<E> next = node.next;  
  
    if (prev == null) {  
        first = next;  
    } else {  
        prev.next = next;  
    }  
  
    if (next == null) {  
        last = prev;  
    } else {  
        next.prev = prev;  
    }  
  
    size--;  
  
    return node.element;  
}
```

双向链表 vs 单向链表

■ 粗略对比一下删除的操作数量

□ 单向链表: $1 + 2 + 3 + \dots + n = \frac{(1+n) * n}{2} = \frac{n}{2} + \frac{n^2}{2}$, 除以n平均一下是 $\frac{1}{2} + \frac{n}{2}$

□ 双向链表: $(1 + 2 + 3 + \dots + \frac{n}{2}) * 2 = \frac{(1 + \frac{n}{2}) * \frac{n}{2}}{2} * 2 = \frac{n}{2} + \frac{n^2}{4}$, 除以n平均一下是 $\frac{1}{2} + \frac{n}{4}$

□ 操作数量缩减了近一半

双向链表 vs 动态数组

- **动态数组**：开辟、销毁内存空间的次数相对较少，但可能造成内存空间浪费（可以通过缩容解决）
- **双向链表**：开辟、销毁内存空间的次数相对较多，但不会造成内存空间的浪费
- 如果频繁在**尾部**进行**添加**、**删除**操作，**动态数组**、**双向链表**均可选择
- 如果频繁在**头部**进行**添加**、**删除**操作，建议使用**双向链表**
- 如果有频繁的（**在任意位置**）**添加**、**删除**操作，建议使用**双向链表**
- 如果有频繁的**查询**操作（随机访问操作），建议使用**动态数组**
- 有了双向链表，单向链表是否就没有任何用处了？
 - 并非如此，在哈希表的设计中就用到了单链表
 - 至于原因，在哈希表章节中会讲到

LinkedList源码分析

- JDK 中的 `java.util.LinkedList`

- 双向链表

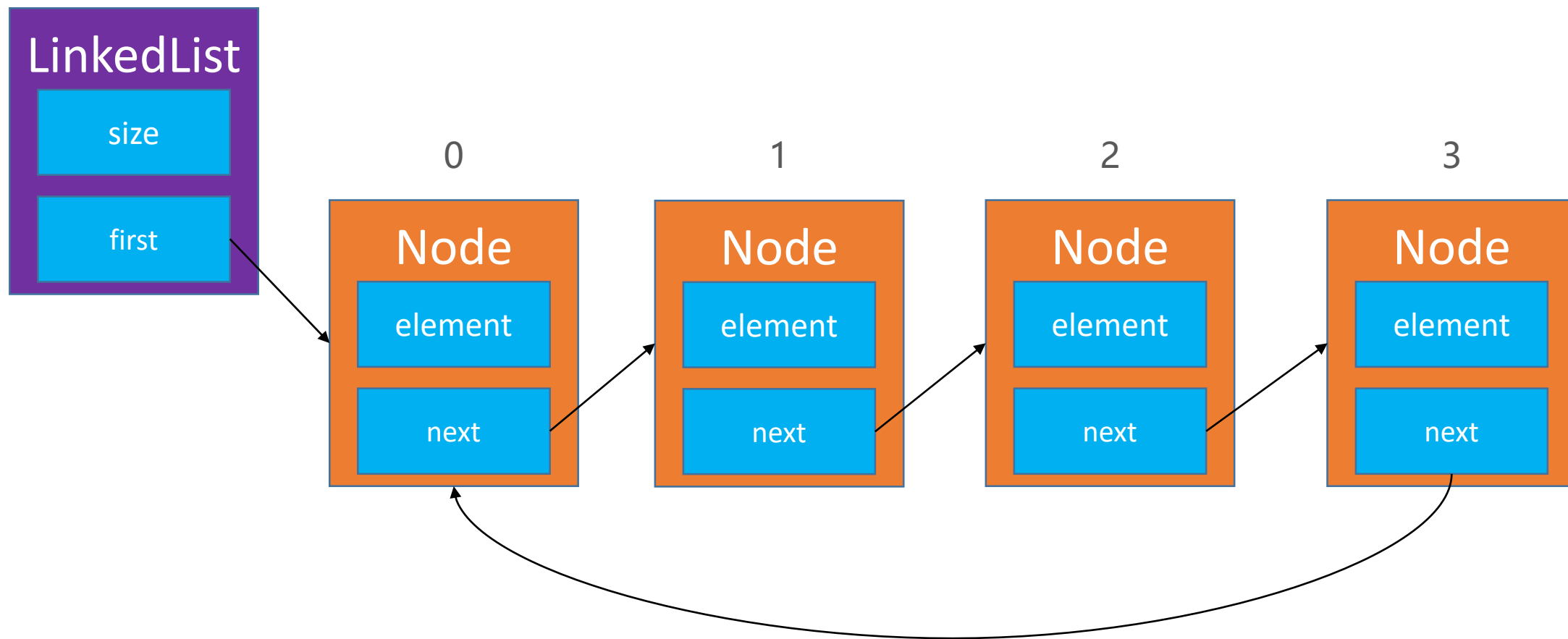
- `clear` 分析

- 多出来的接口

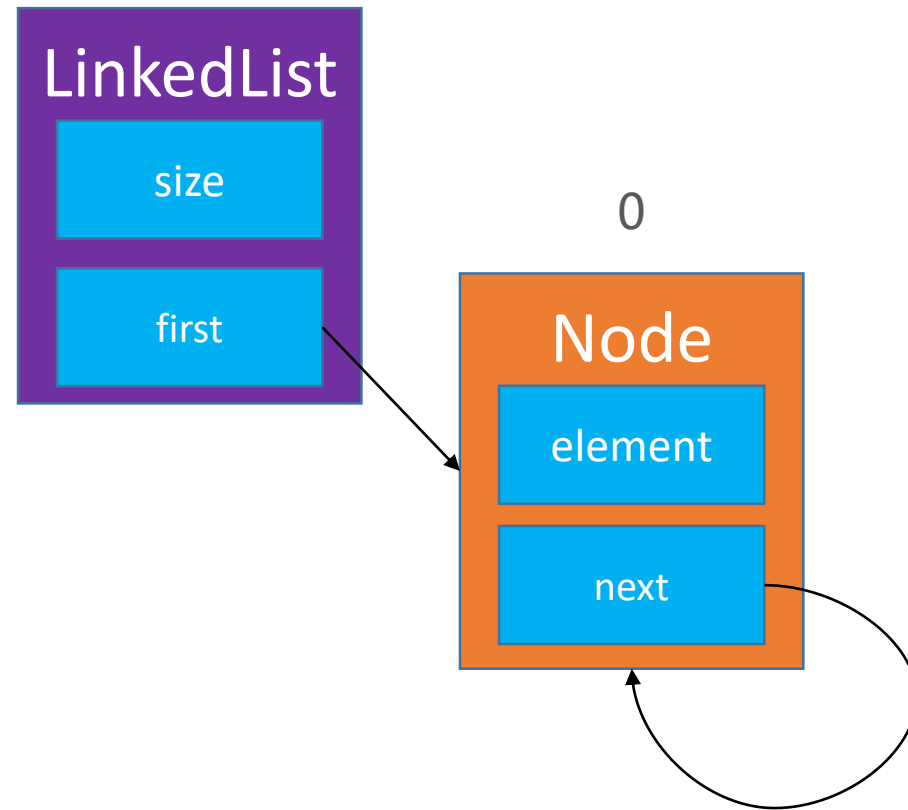
- ✓ `...First`

- ✓ `...Last`

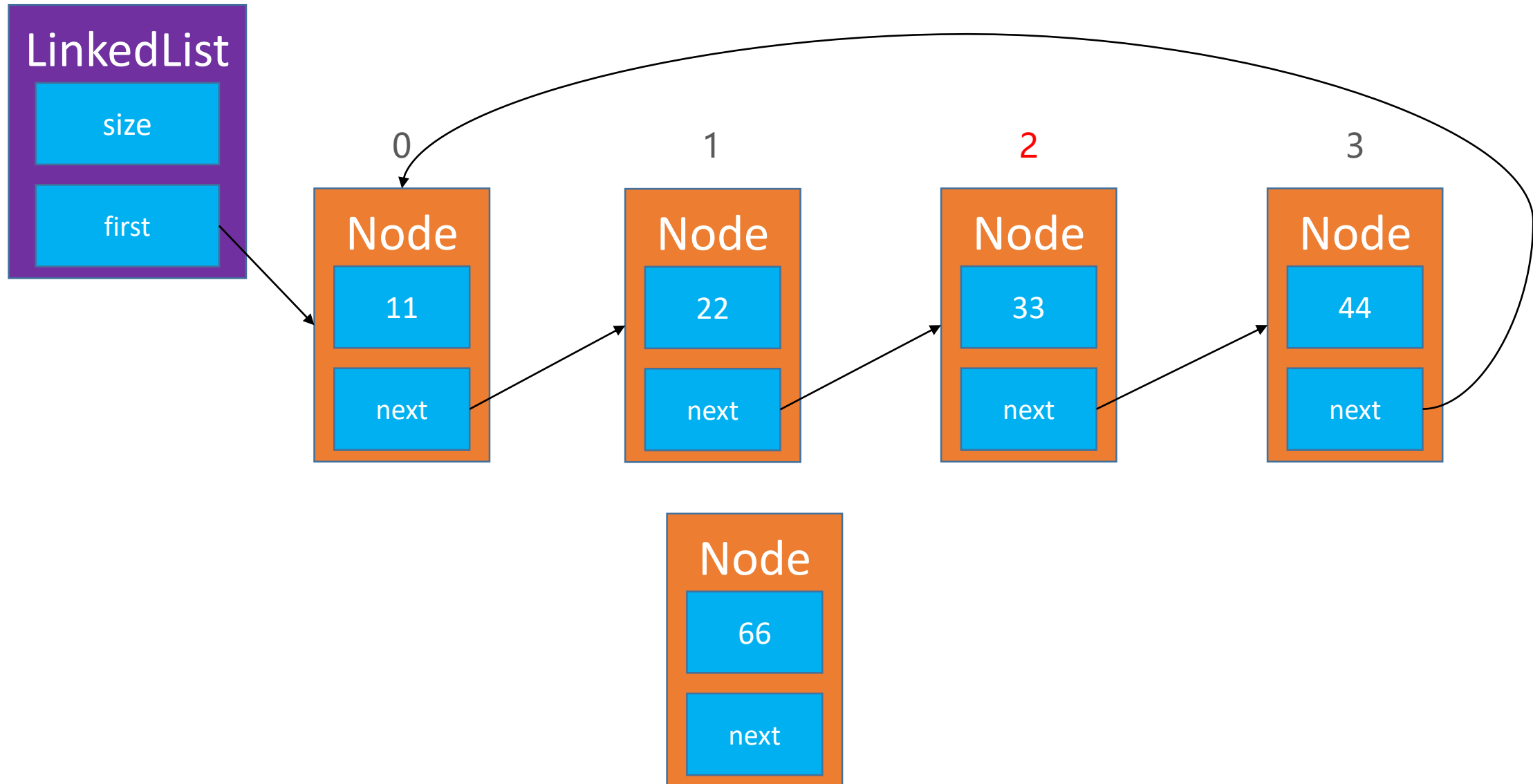
单向循环链表



单向循环链表 – 只有1个节点



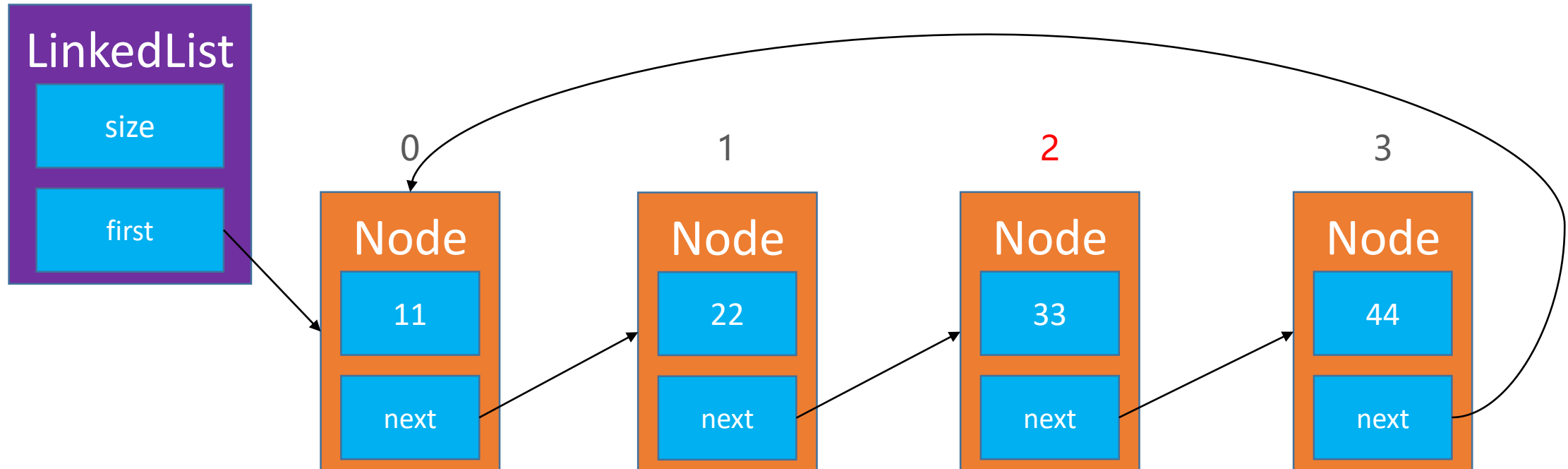
单向循环链表 – add(int index, E element)



单向循环链表 – add(int index, E element)

```
public void add(int index, E element) {  
    rangeCheckForAdd(index);  
  
    if (index == 0) {  
        Node<E> newFirst = new Node<>(element, first);  
        Node<E> last = (size == 0) ? newFirst : node(size - 1);  
        last.next = newFirst;  
        first = newFirst;  
    } else {  
        Node<E> prev = node(index - 1);  
        prev.next = new Node<>(element, prev.next);  
    }  
    size++;  
}
```

单向循环链表 – remove(int index)



单向循环链表 – remove(int index)

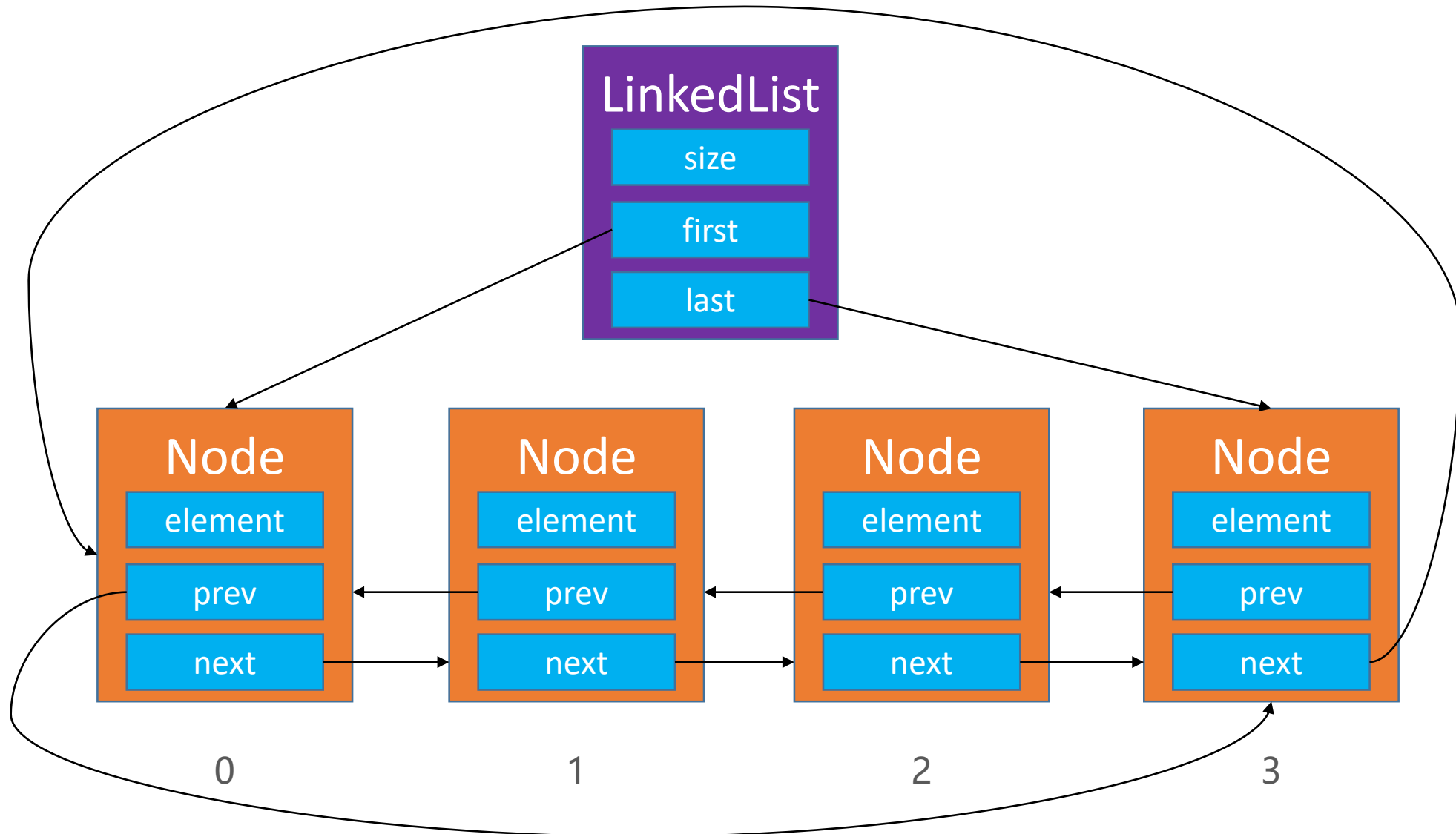
```
public E remove(int index) {
    rangeCheck(index);

    Node<E> node = first;
    if (index == 0) {
        if (size == 1) {
            first = null;
        } else {
            Node<E> last = node(size - 1);
            first = first.next;
            last.next = first;
        }
    } else {
        Node<E> previous = node(index - 1);
        node = previous.next;
        previous.next = node.next;
    }

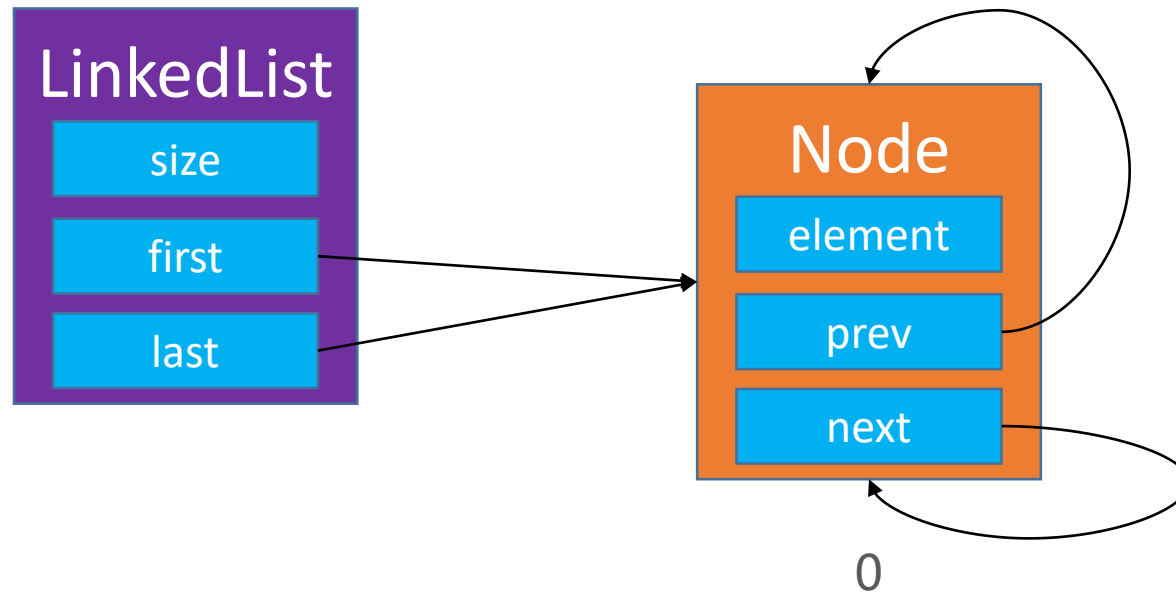
    size--;

    return node.element;
}
```

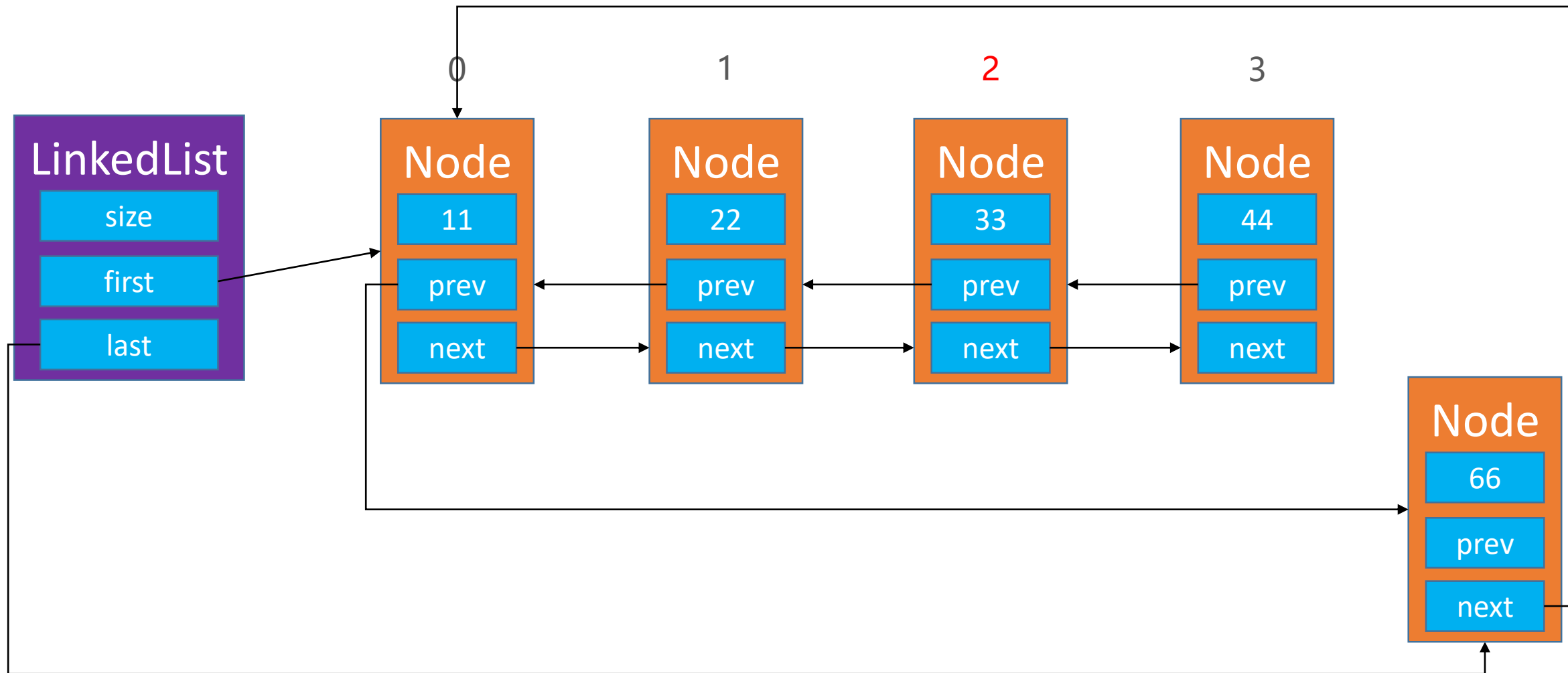
双向循环链表



双向循环链表 – 只有1个节点



双向循环链表 – add(int index, E element)



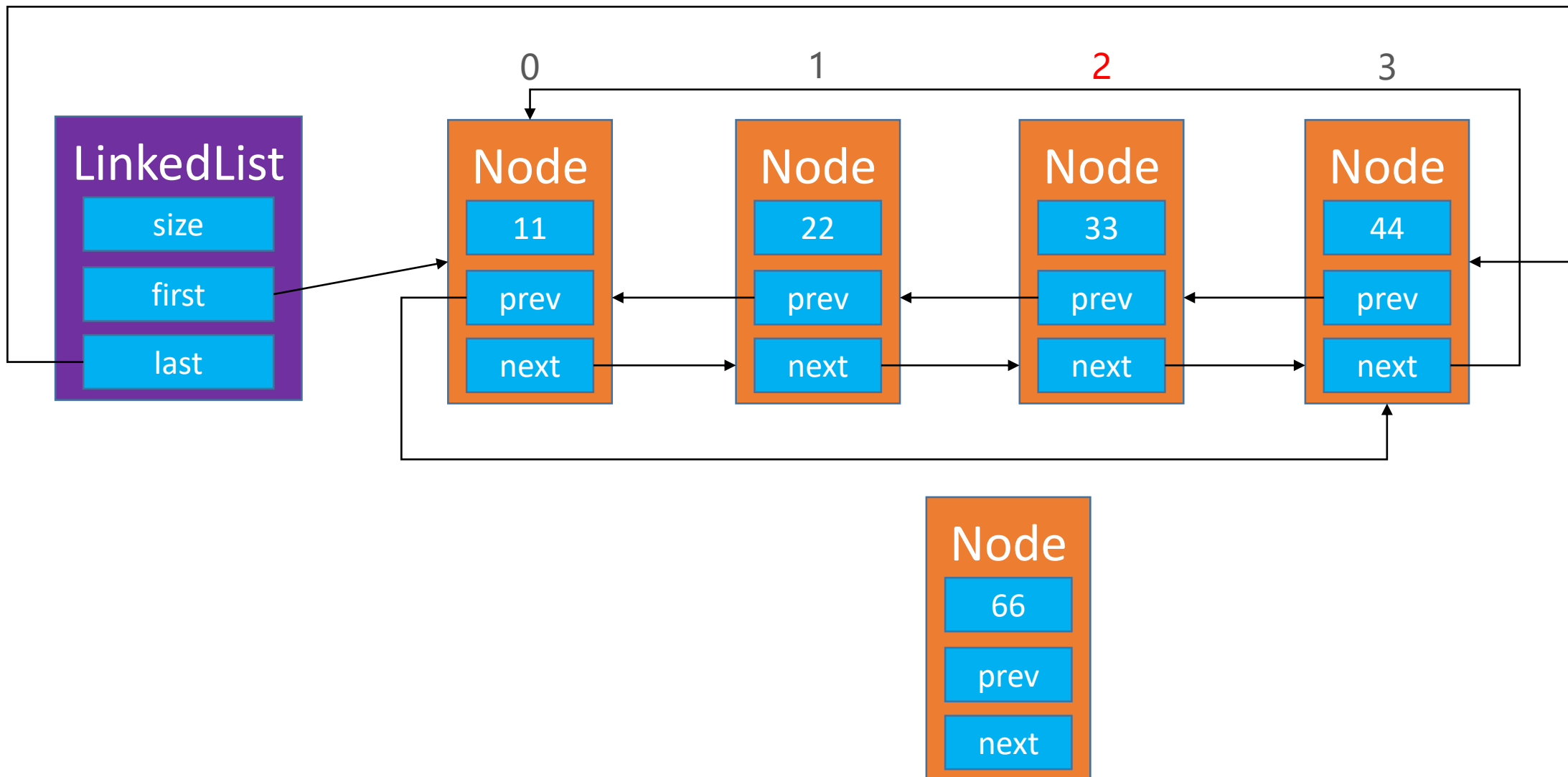
双向循环链表 – add(int index, E element)

```
public void add(int index, E element) {
    rangeCheckForAdd(index);

    if (index == size) {
        Node<E> oldLast = last;
        last = new Node<>(oldLast, element, first);
        if (oldLast == null) {
            first = last;
            first.next = first;
            first.prev = first;
        } else {
            oldLast.next = last;
            first.prev = last;
        }
    } else {
        Node<E> next = node(index);
        Node<E> prev = next.prev;
        Node<E> current = new Node<>(prev, element, next);
        next.prev = current;
        prev.next = current;

        if (next == first) {
            first = current;
        }
    }
    size++;
}
```

双向循环链表 – remove(int index)



双向循环链表 – remove(int index)

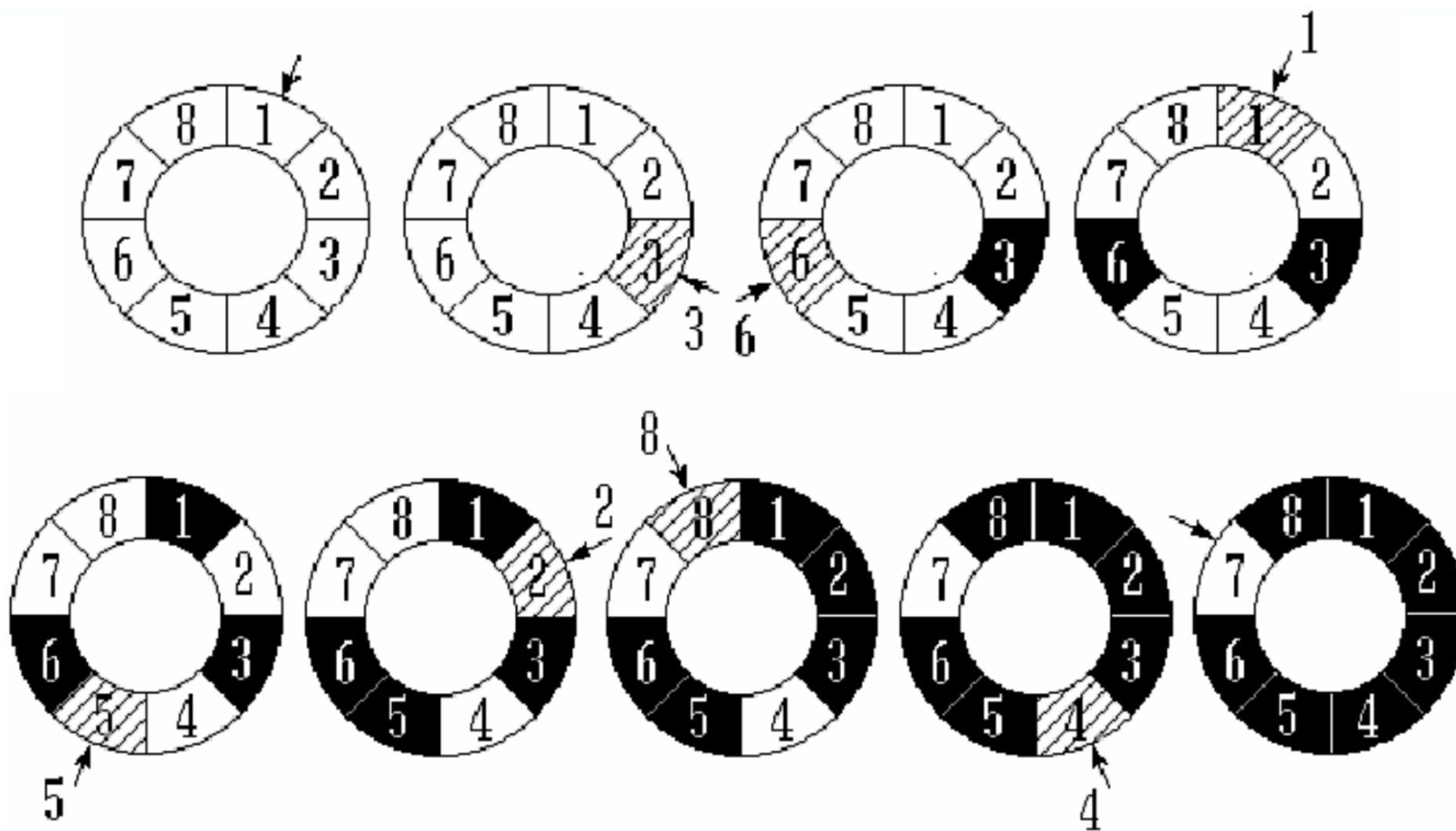
```
public E remove(int index) {
    rangeCheck(index);

    Node<E> node = node(index);
    if (size == 1) {
        first = null;
        last = null;
    } else {
        Node<E> prev = node.prev;
        Node<E> next = node.next;
        prev.next = next;
        next.prev = prev;

        if (node == first) {
            first = next;
        }

        if (node == last) {
            last = prev;
        }
    }
    size--;
    return node.element;
}
```

练习 – 约瑟夫问题 (Josephus Problem)



如何发挥循环链表的最大威力？

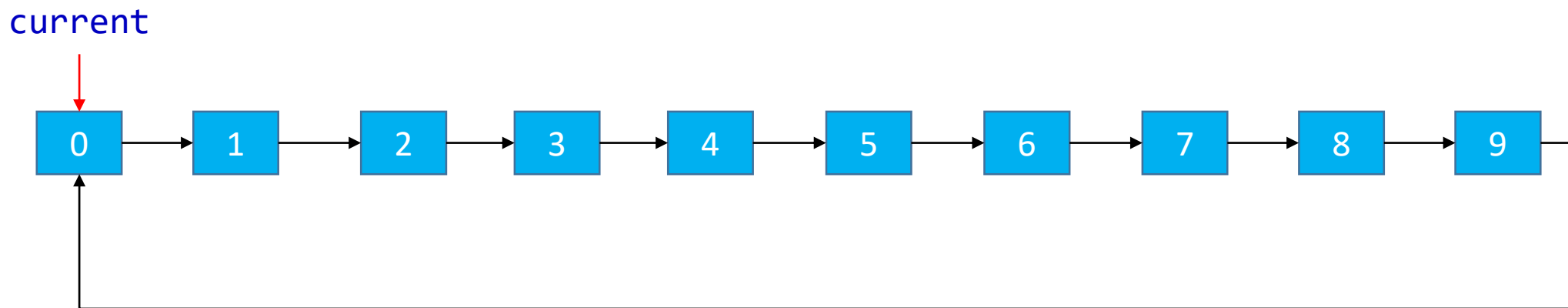
■ 可以考虑增设1个成员变量、3个方法

□ `current` : 用于指向某个节点

□ `void reset()` : 让 `current` 指向头结点 `first`

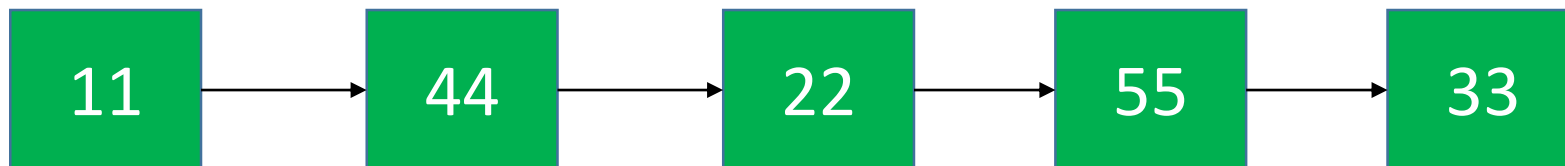
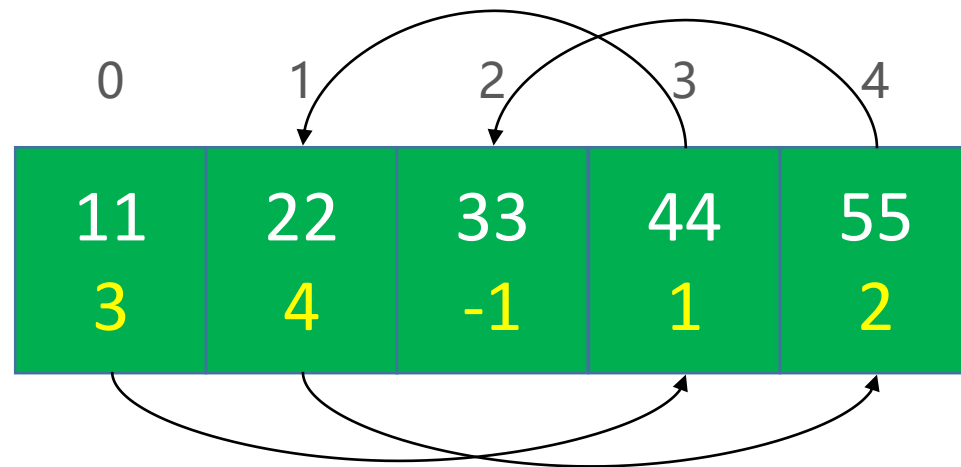
□ `E next()` : 让 `current` 往后走一步, 也就是 `current = current.next`

□ `E remove()` : 删除 `current` 指向的节点, 删除成功后让 `current` 指向下一个节点



静态链表

- 前面所学习的链表，是依赖于指针（引用）实现的
- 有些编程语言是没有指针的，比如早期的 BASIC、FORTRAN 语言
- 没有指针的情况下，如何实现链表？
- 可以通过数组来模拟链表，称为静态链表
- 数组的每个元素存放 2 个数据：值、下个元素的索引
- 数组 0 位置存放的是头结点信息



- 思考：如果数组的每个元素只能存放 1 个数据呢？
- 那就使用 2 个数组，1 个数组存放索引关系，1 个数组存放值

作业

- 移除链表元素

- <https://leetcode-cn.com/problems/remove-linked-list-elements/>

- 删除排序链表中的重复元素

- <https://leetcode-cn.com/problems/remove-duplicates-from-sorted-list/>

- 链表的中间结点

- <https://leetcode-cn.com/problems/middle-of-the-linked-list/solution/>

作业 – ArrayList能否进一步优化?

0	1	2	3	4	5	6	7	8
	22	99	44	55	66	77		