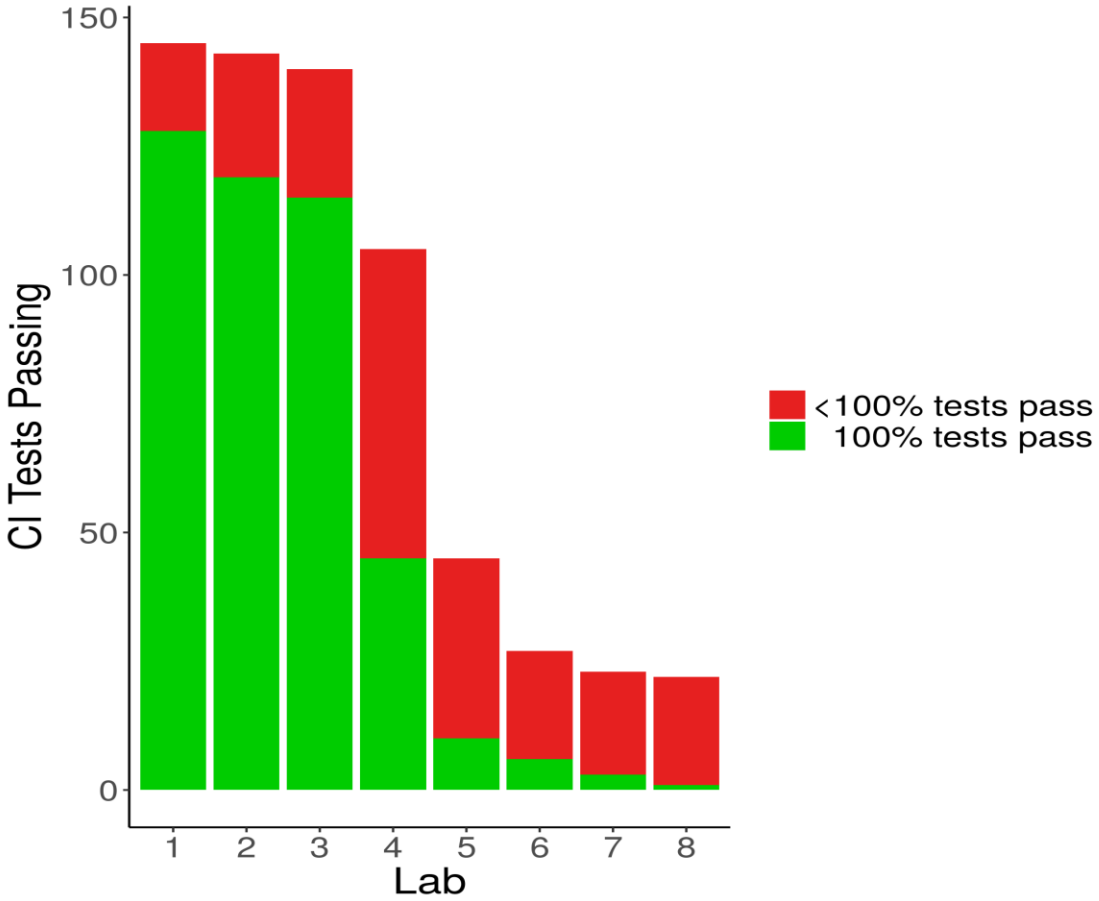
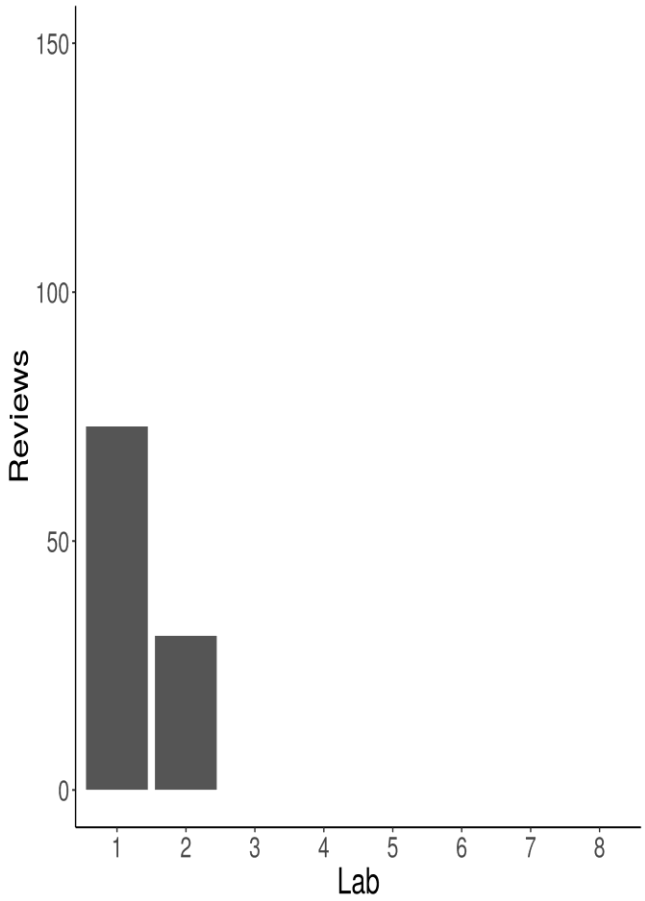

150
100
50
0
CI Tests Passing
1
2
3
4
5
6
7
8
Lab
<100% tests pass
100% tests pass


150
100
50
0
Reviews
1
2
3
4
5
6
7
8
Lab

Group 1 (Friday 8th)

- **Lab 5**: queues, linked lists
- (Lab 4 deadline)

Group 2 (Monday 11th)

- **Lab 5**: queues, linked lists
- (Lab 4 deadline)

Next week is **reading week for SD3**

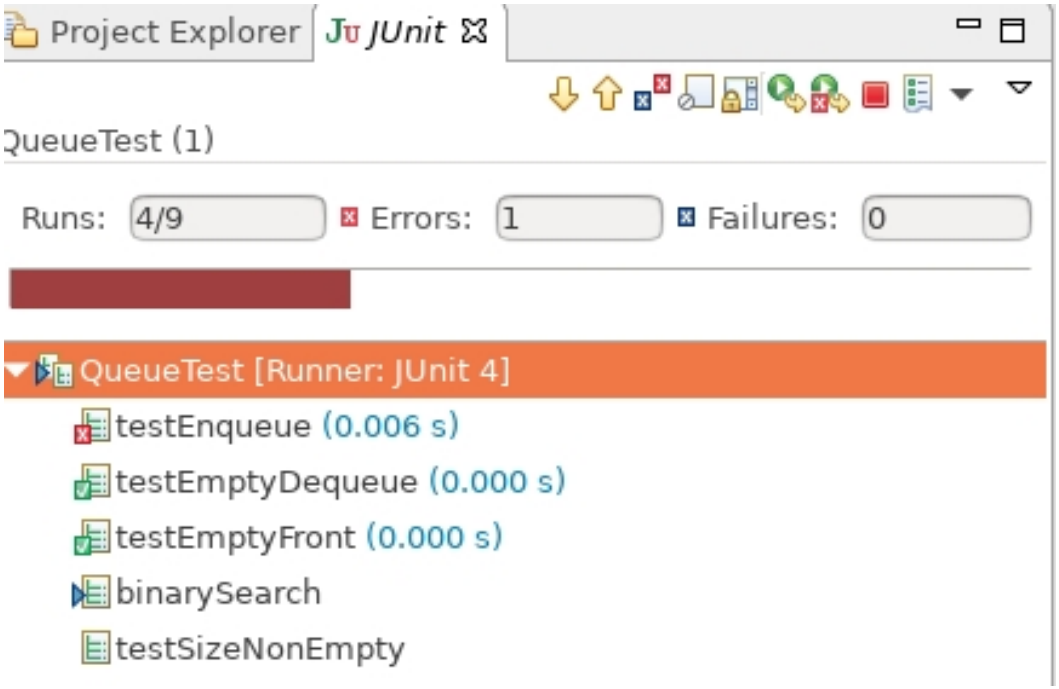
- **No lectures:** Thursday 14th, Friday 15th
- Read *"Week 6 revision slides"* on Vision
- Group 2 lab is on (Monday 11th)

Keep an eye out for infinite loops:

- while loops that never exit `while (<condition>) { .. }`

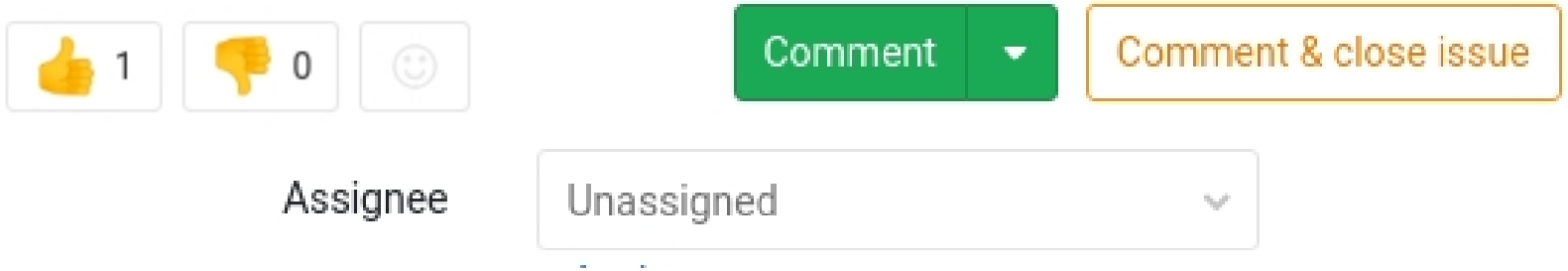
- Recursive method that doesn't converge to a base case

```
public int foo(int i) {
    if (<condition>) { return i; }
    else { … ; return foo(i);
    }
}
```



- Writing peer feedback
- Copy/paste code you're talking about
- Offer code alternatives

- **Responding to feedback**
- Discuss the code review
- Is there anything contentious in the review?
- Ask in the comments if anything unclear
- ***Thank the reviewer!***



Software Development 3 (F27SG)

Lecture 10

Doubly Linked Lists

Rob Stewart

Overview

- About linked lists we covered that
  - adding elements to Linked Lists is easy,
  - but removing elements at the tail or middle of the list is hard
- Today we will develop **doubly** Linked-Lists to overcome these problems
  - worked examples
  - Java implementations

Time Complexity

	Arrays	Singly Linked lists
Insert/delete at **start**	O(1)	O(1)
Insert/delete in **middle**	O(1)	~ O(N)
Insert/delete at **end**	O(1)	O(N)

- Quite annoying that linked lists mostly O(n)
- Even if we already have the Node we want to delete: we still traverse the whole list to find it **because we need to update the previous node reference**
- Can use Doubly Linked Lists instead
  - Small space cost vs potential time complexity saving
  - Makes traversal easier and sorting possible

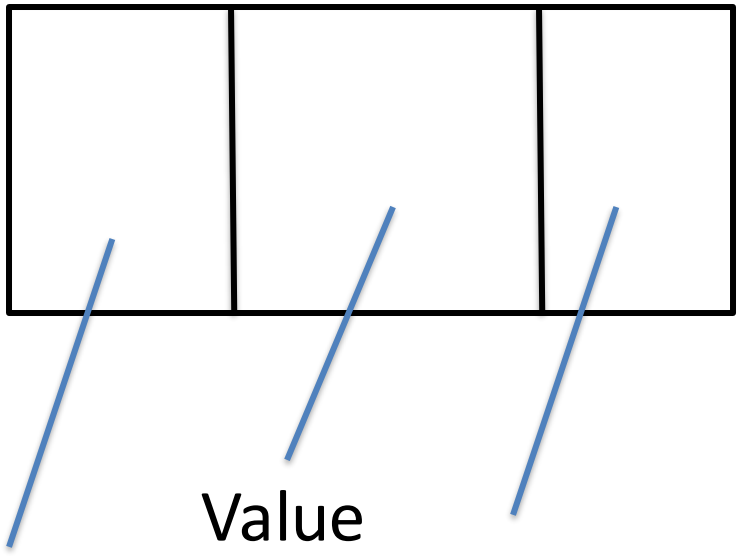
Doubly Linked Lists

- Doubly Linked Lists supports 2 way traversal
- Each Node has 2 references to other Nodes
  1. One holds a reference to **next node** (N+1)
     - As with singly linked lists
  2. Other reference: **previous node** (N-1)
  - We hold a global reference to the
    - first (`head`) and last (`tail`) nodes

Doubly Linked List Nodes

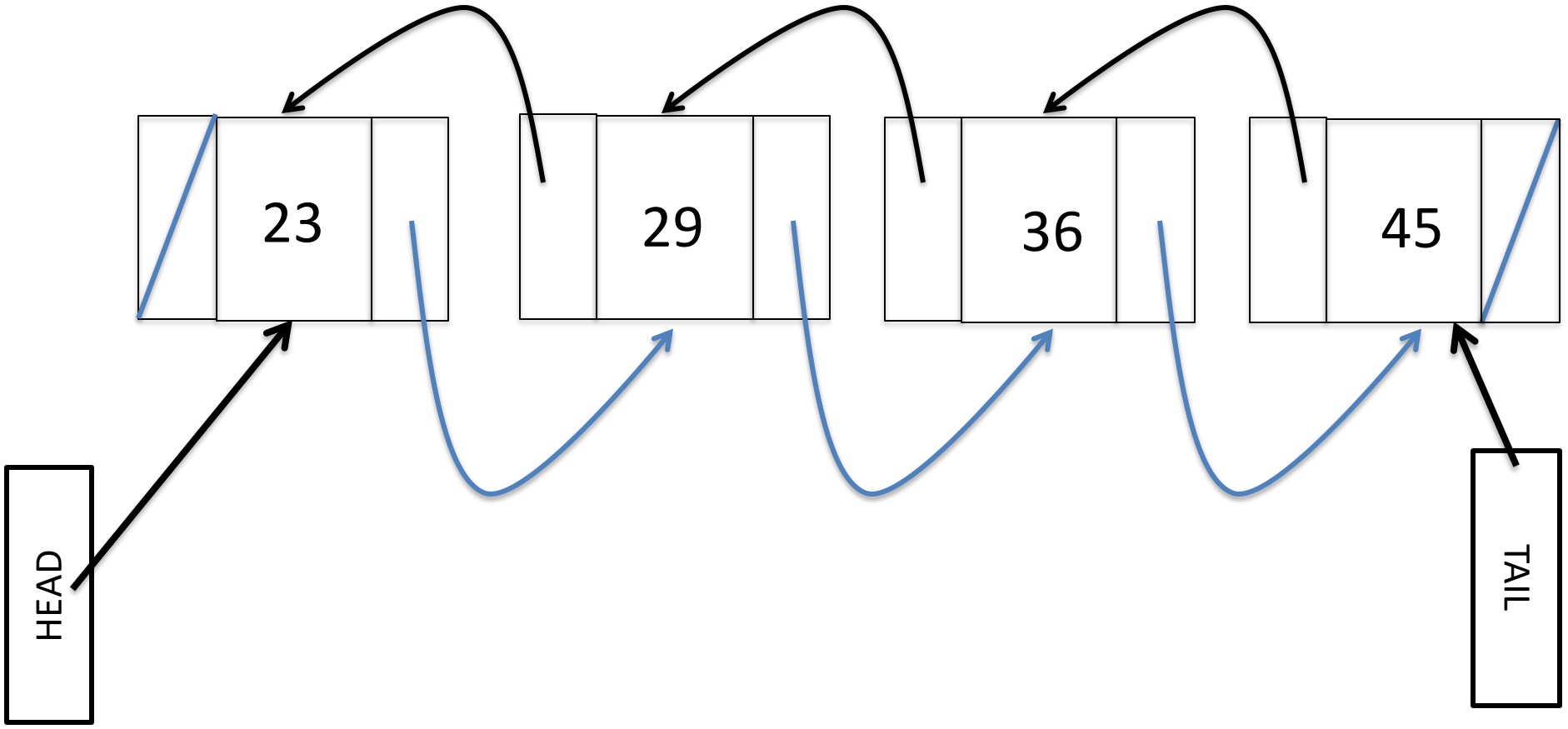
```
public class Node{
  public Object value;
  public Node nextNode;
  public Node prevNode;

public Node(Object val){
      value = val;
      nextNode = null;
      prevnode = null;
}
};
```



Doubly Linked Lists

- Can Traverse the List in Both Directions
  - Still not Random access, but can get Previous Node
  - Makes **removal at end** and **sorting lists** easier



Doubly Linked List Operations

- All operations are similar to singly linked lists
- However a **few more things to tidy up**
- We will look at a smaller subset of operations to show this:
  - Add at head
  - Add at tail
  - Add in middle
  - Delete at tail
- You should be able to infer the others from these

Insert at Head

- (Almost) Same as Single Linked-Lists
  - Since there is no previous node to link to

1. Create the New Node
2. Set its next Node to be the existing head Node
3. Set head Node's prev to be new Node
   - Added functionality versus singly linked lists
4. Set the new Node to be the Head Node

Insert at Tail

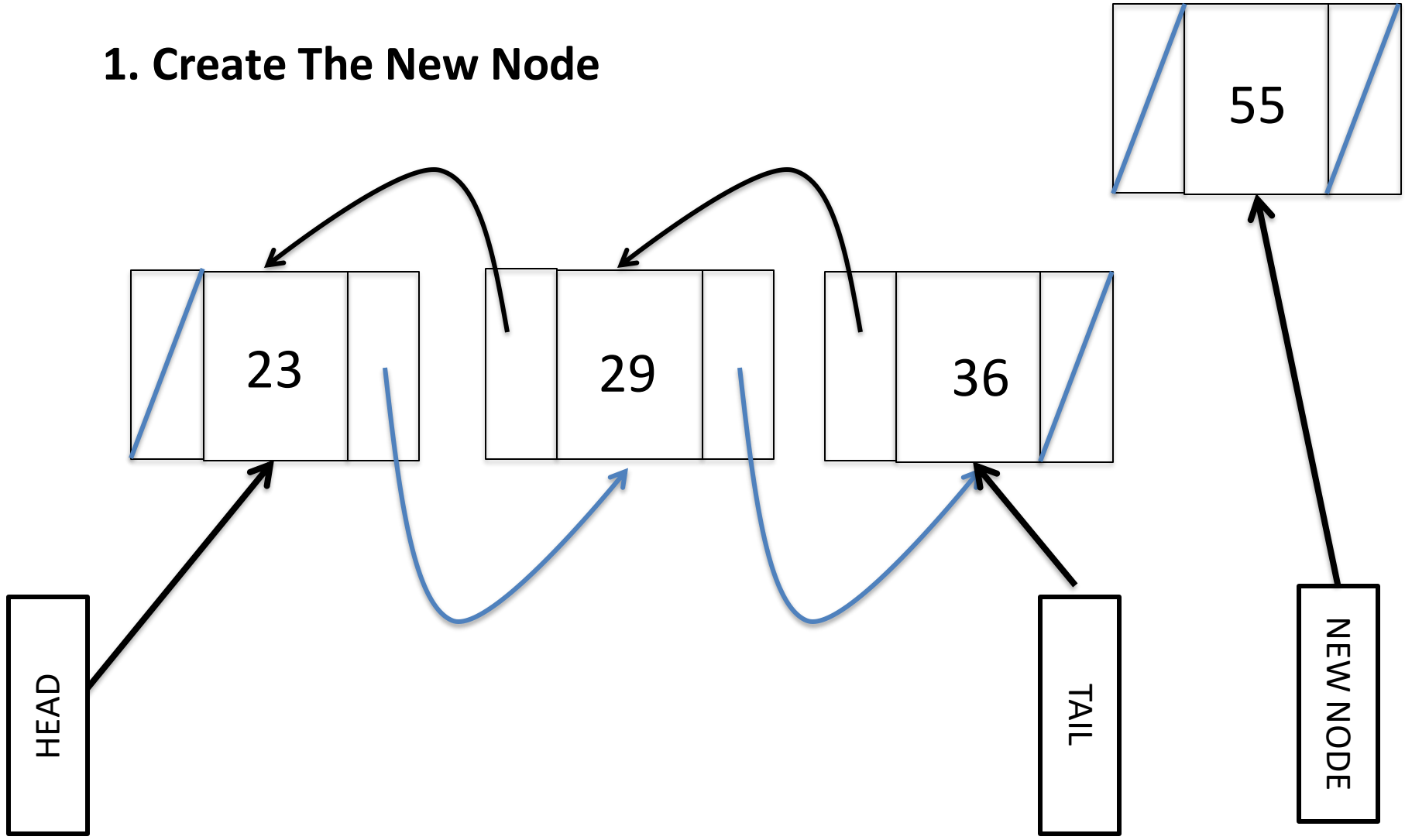
- Even Easier than with a Single Linked List
  - Since we hold a reference to the last node

1. Create New Node
2. Set New Node's previous Node to be existing Tail
3. Set New Node as the existing Tail Node's next Node
4. Set the tail reference to reference the new node

- Changes the operation from O(n) to O(1)
  - We don't need to traverse the entire list

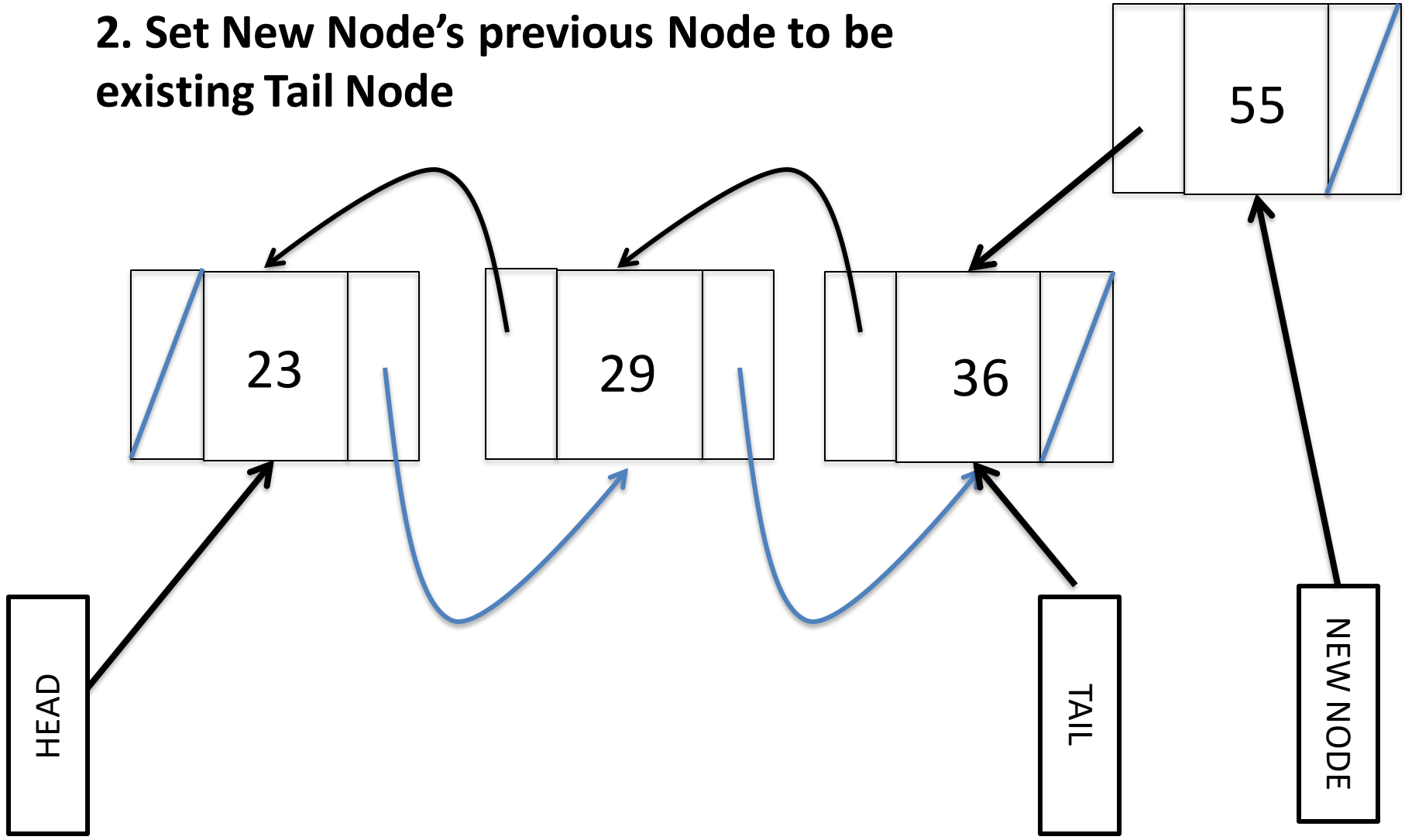
Insert At Tail

1. Create The New Node



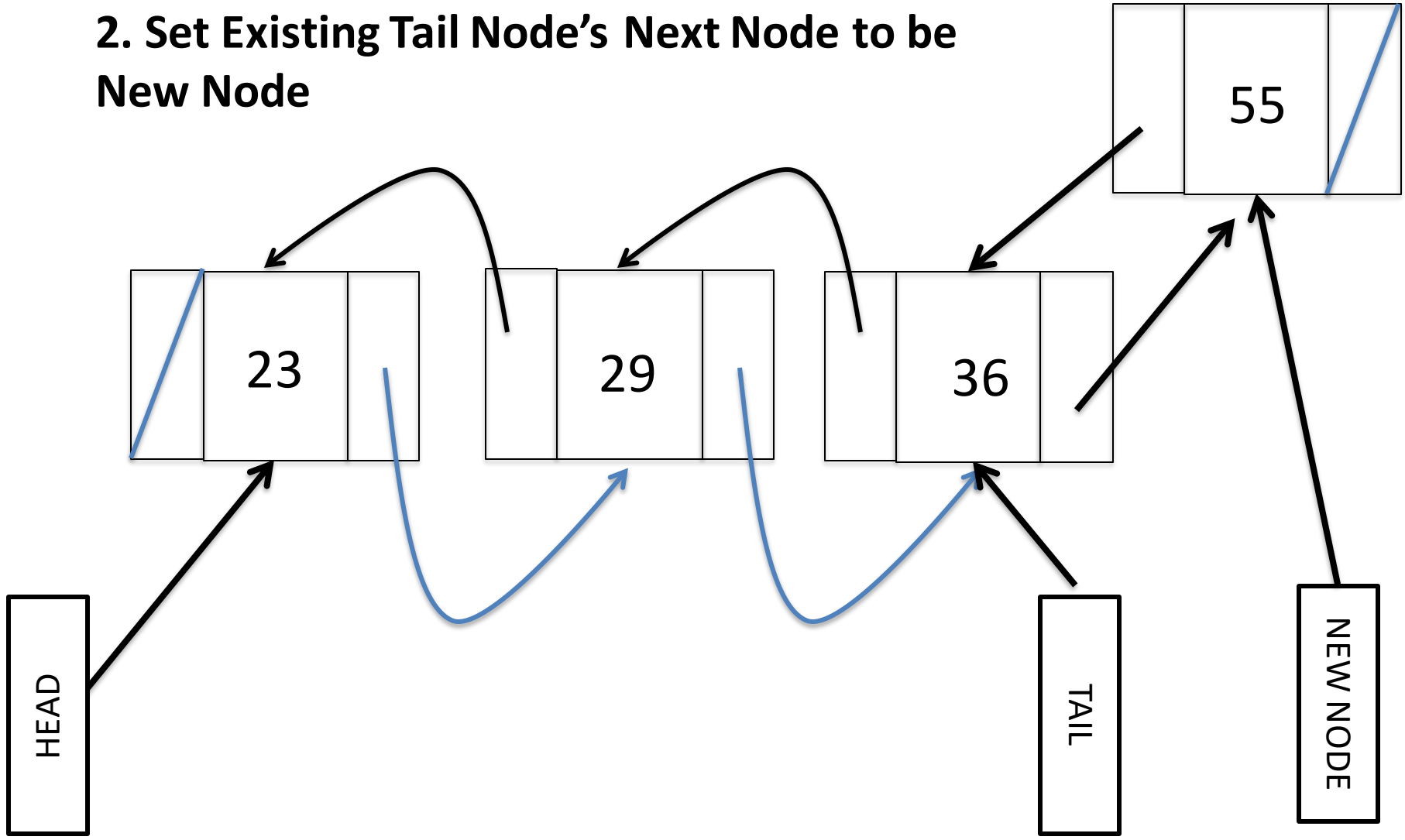
Insert At Tail

2. Set New Node's previous Node to be existing Tail Node



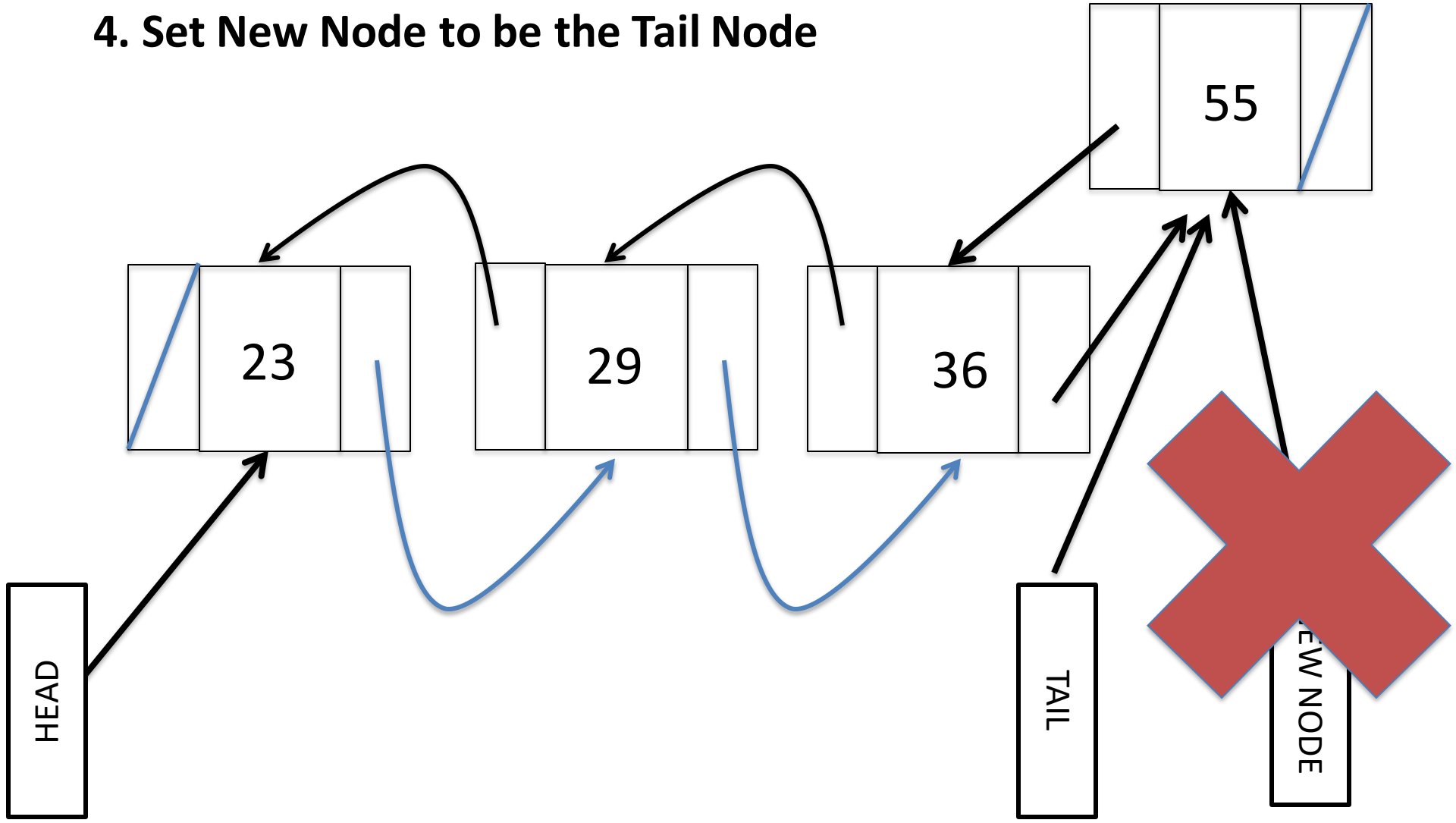
Insert At Tail

2. Set Existing Tail Node's Next Node to be New Node



Insert At Tail

4. Set New Node to be the Tail Node



Insert in Middle

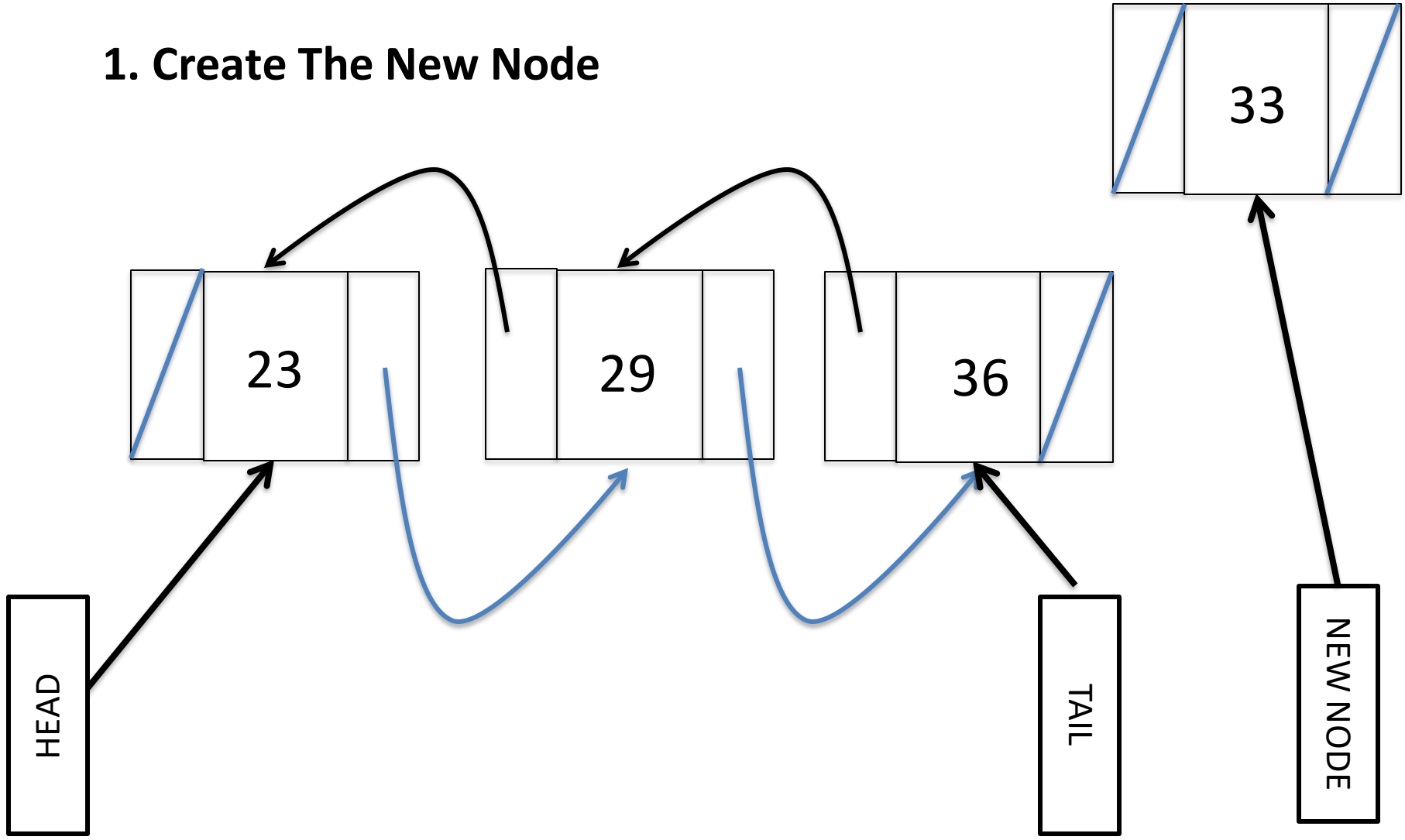
- As with Single Linked List you need a function to decide where to insert the New Node
  - We are assuming that we keep the list in ascending order (the “Hard” case in Single Linked List

1. Create New Node
2. Find the first Node with a value greater than that in New Node (N)
3. Set New Node’s next Node to be N
4. Set New Node’s previous Node to be (N-1)
5. Set N-1’s Next Node to be New Node
6. Set N’s previous Node to be New Node

- Still O(n) worst case
- Application dependent whether this is useful or not

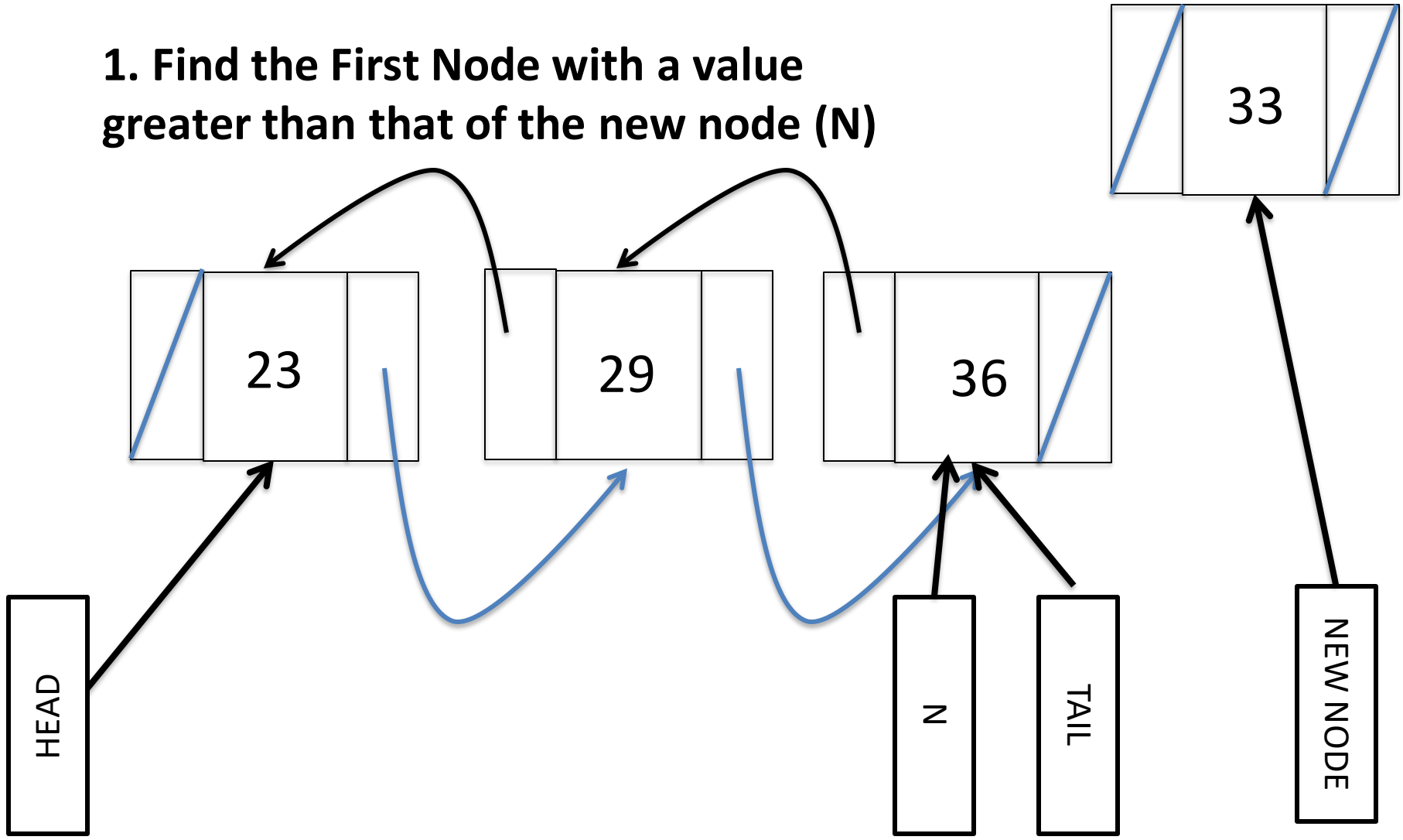
Insert In Middle

1. Create The New Node



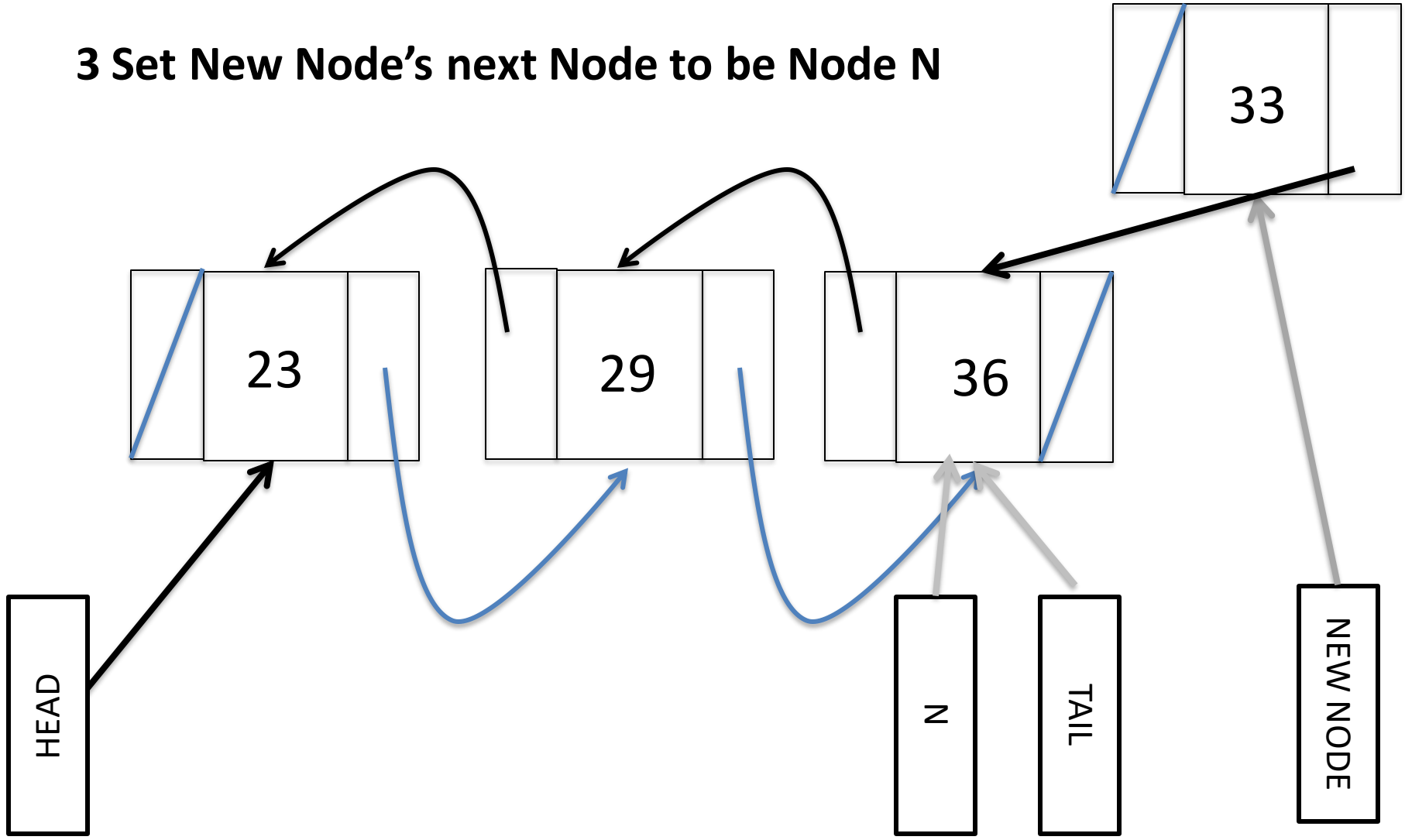
Insert In Middle

1. Find the First Node with a value greater than that of the new node (N)



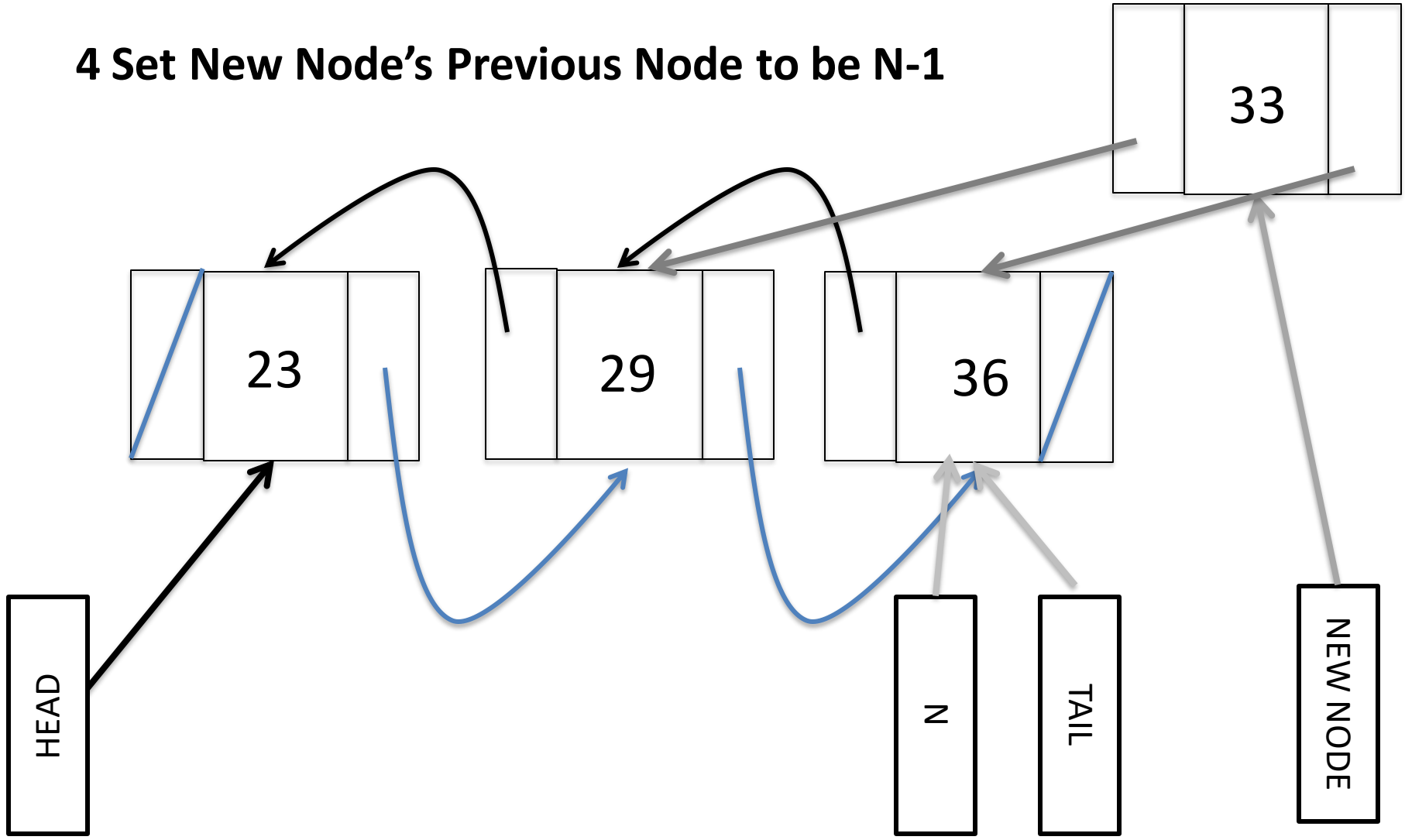
Insert In Middle

3 Set New Node's next Node to be Node N



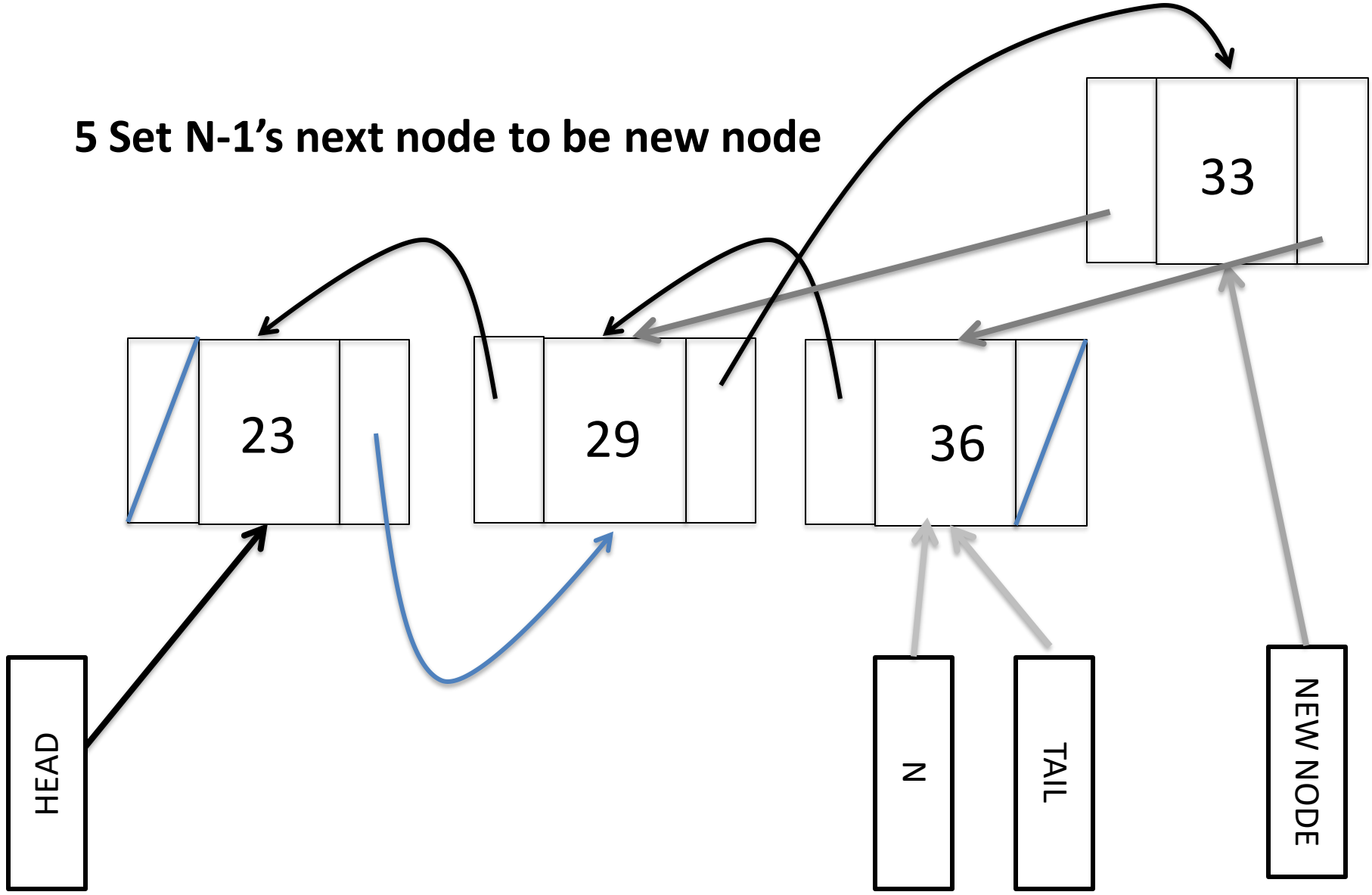
Insert In Middle

4 Set New Node's Previous Node to be N-1



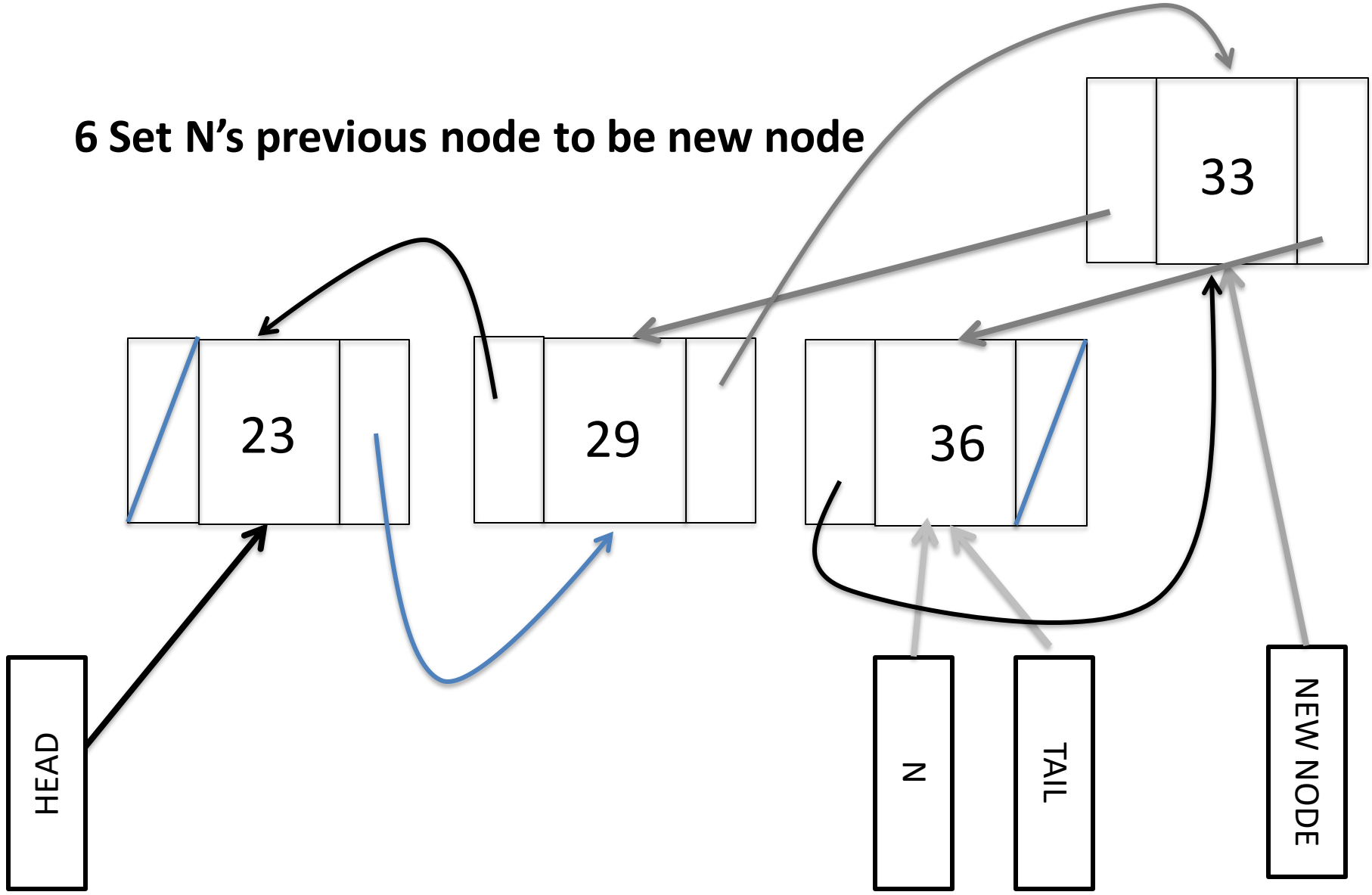
Insert In Middle

5 Set N-1's next node to be new node



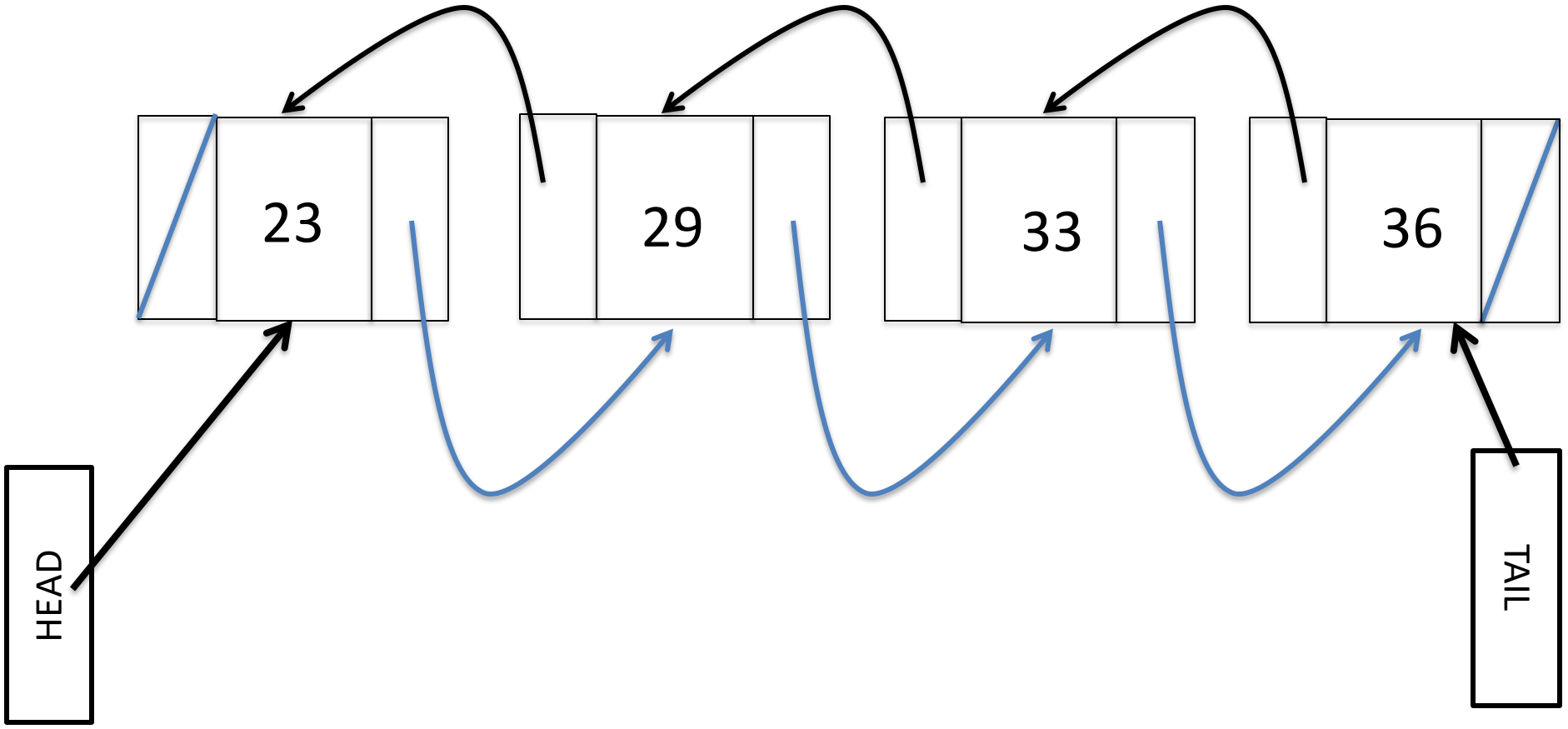
Insert In Middle

6 Set N's previous node to be new node



Insert In Middle

Which leads to this



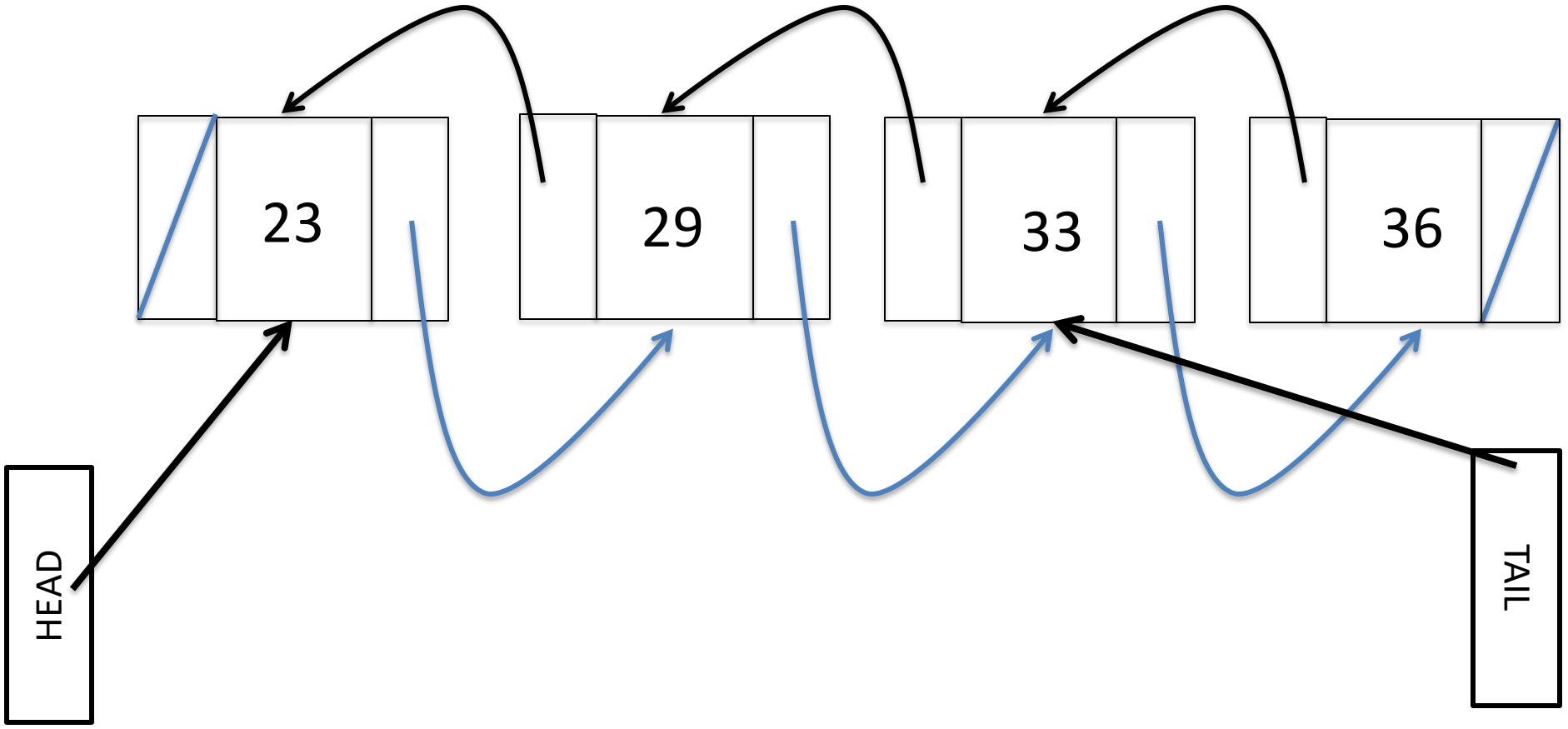
Delete at Tail

- Again, as we have a tail reference this moves from O(n) to O(1)

- Set Tail to be current tail's previous node
- Set new Tail's next Node to Null
  - I.e. if tail in Node N set N-1's next Node to be null

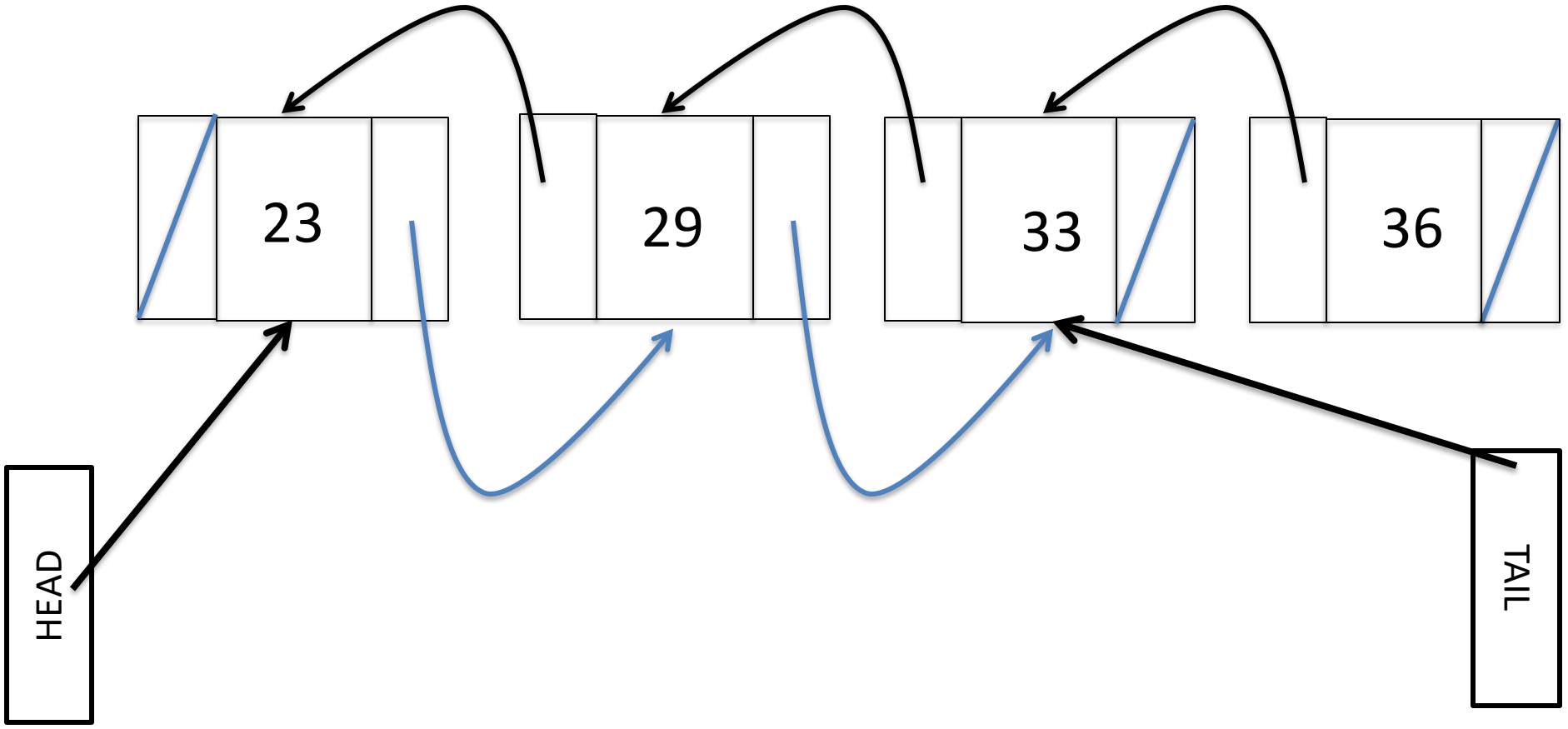
Delete at Tail

1. Set Tail to Current Tails previous Node



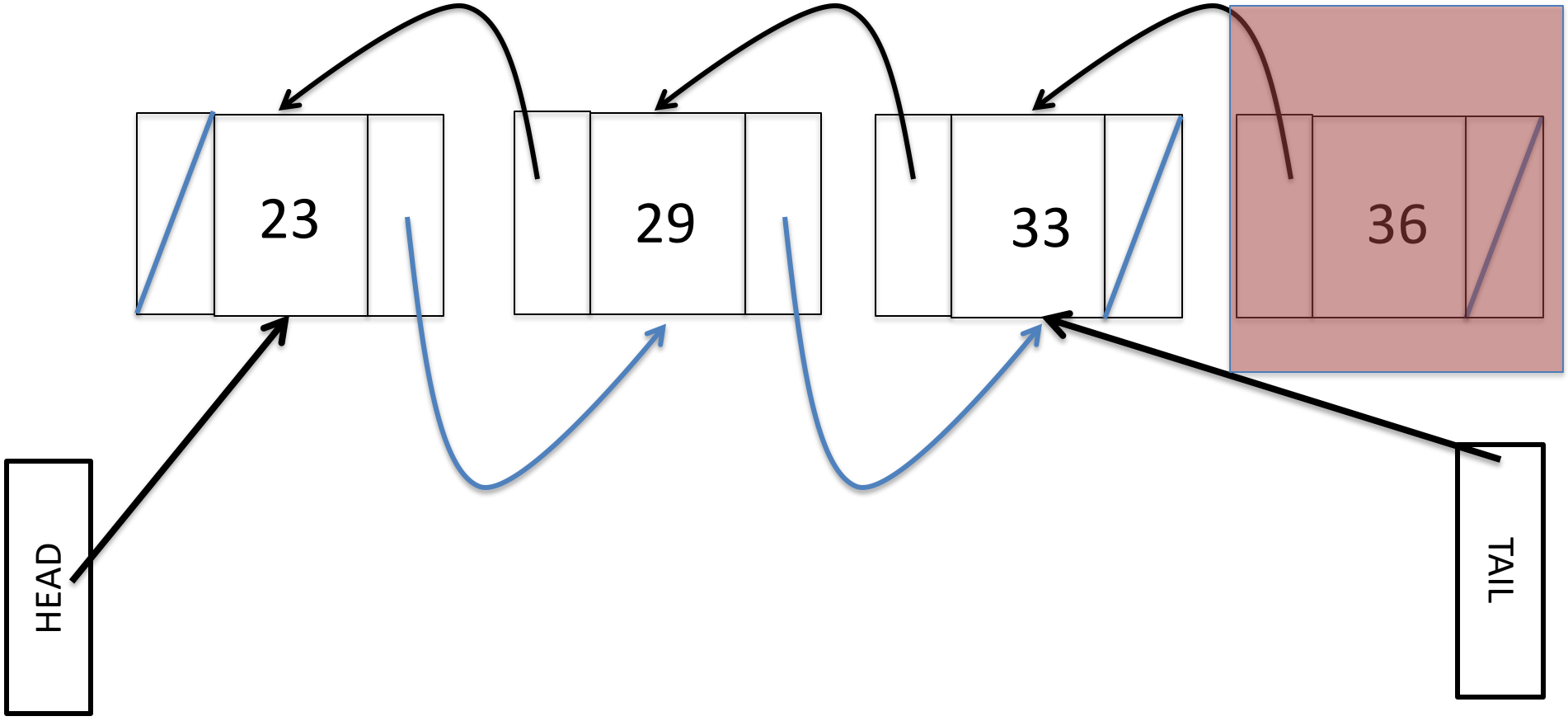
Delete at Tail

2. Set new Tail's next Node to Null

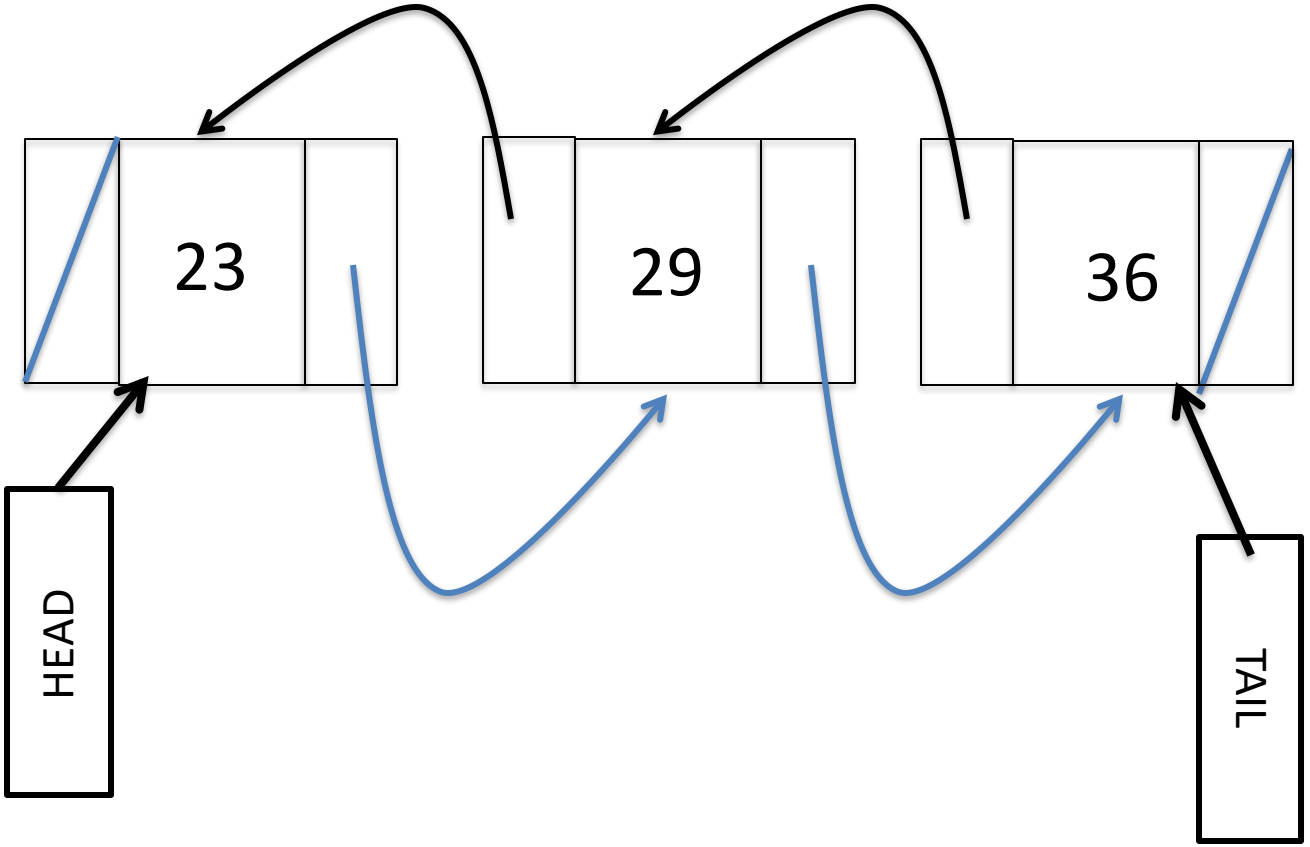


Delete at Tail

What happens to this Node?



Exercise
reverse the doubly linked list



Some Issues

- Need to be careful over the order of steps
  - Sometimes easy to leave yourself without a reference to an object you need later
  - E.g. when inserting in middle setting Node N's prev node to be the newNode before accessing N-1 means you can't access N-1
- Also **need to be careful on empty lists**
  - Either adding to an empty list or deleting from a 1 Node list
  - You might need to set the head and tail variables
  - We will illustrate this in Java

Implementation

```
public class DLinkedList {
  private class Node {
    private Object value;
    private Node nextNode;
    private Node prevNode;

    public Node(Object v) {
      value = v;
      nextNode = null;
      prevNode = null;
    }

    public Object getValue() {
      return value;
    }

    public void setValue(Object v) {
      value = v;
    }

    public Node getNextNode() {
      return nextNode;
    }

    public void setNextNode(Node n) {
      nextNode = n;
    }

    public Node getPrevNode() {
      return prevNode;
    }

    public void setPrevNode(Node n) {
      prevNode = n;
    }
  }

  private Node headNode;
  private Node tailNode;

  public DLinkedList() {
    headNode = null;
    tailNode = null;
  }
```

Implementation - add

```
public void addAtHead(Object o) {
   Node newNode = new Node(o);
   newNode.setNextNode(headNode);
   if (headNode != null)
      headNode.setPrevNode(newNode);
   headNode = newNode;
   // special case for empty list
   if (tailNode == null)
      tailNode = newNode;
}
```

```
public void addAtTail(Object o) {
   Node newNode = new Node(o);
   // this means that headNode == null too!
   if(tailNode == null){
      tailNode = newNode;
      headNode = newNode;
   }else{
      newNode.setPrevNode(tailNode);
      tailNode.setNextNode(newNode);
      tailNode = newNode;
   }
}
```

Implementation - delete

```
public Object deleteAtHead() {
  // list is empty
  if(headNode == null){
    return null;
  }
  // singleton: must update tailnode too
  if(headNode == tailNode){
    Object res = headNode.getValue();
    headNode = null;
    tailNode = null;
    return res;
  }
  // more than 1 element
  Object res = headNode.getValue();
  headNode = headNode.getNextNode();
  headNode.setPrevNode(null);
  return res;
}
```

```
public Object deleteAtTail() {
  // list is empty
  if(tailNode == null){
    return null;
  }
  // singleton: must update tailnode too
  if(headNode == tailNode){
    Object res = tailNode.getValue();
    headNode = null;
    tailNode = null;
    return res;
  }
  // more than 1 element
  Object res = tailNode.getValue();
  tailNode = tailNode.getPrevNode();
  tailNode.setNextNode(null);
  return res;
}
```

Using Linked Lists

- **So that is all on Linked Lists!**
- Linked Lists common in most languages
- In Java: `Java.util.LinkedList<E>`
  - Uses Generics, so can be instantiated to store any class
  - **Used in SD2:** `NewsFeed.java` in SD2 Lecture 5:

```
ArrayList<Post> posts;
```

  - Go take a look through it!
    - http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
  - Easy conversion to/from arrays
    - So you get the best of both worlds