Software Development 3 (F27SG)

Week 6 Revision

Rob Stewart

Overview of data structures so far

Arrays

- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Overview of data structures so far

Arrays

- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists

- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

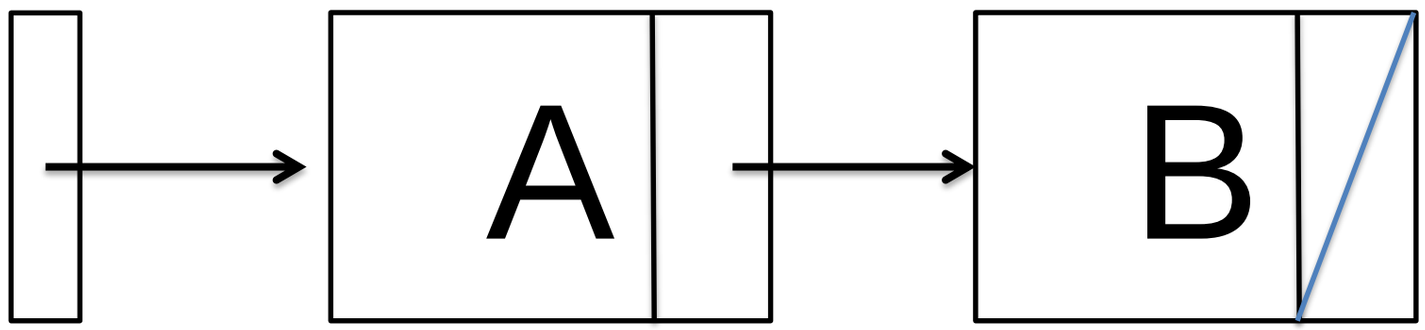
An Aside on Java

- Object o;
  - Declares a variable of type Object
  - Only allocates enough space to hold a memory address
  - o holds a **reference**
    - In this case a null reference
- o = **new** Object();
  - Allocates memory space to hold an instance of type Object
  - Returns the address of that memory location
  - and stores it in the variable o
    - Hence the assignment operator (=)
- o is not the Object, but tells us where the object lives

An Aside on Java

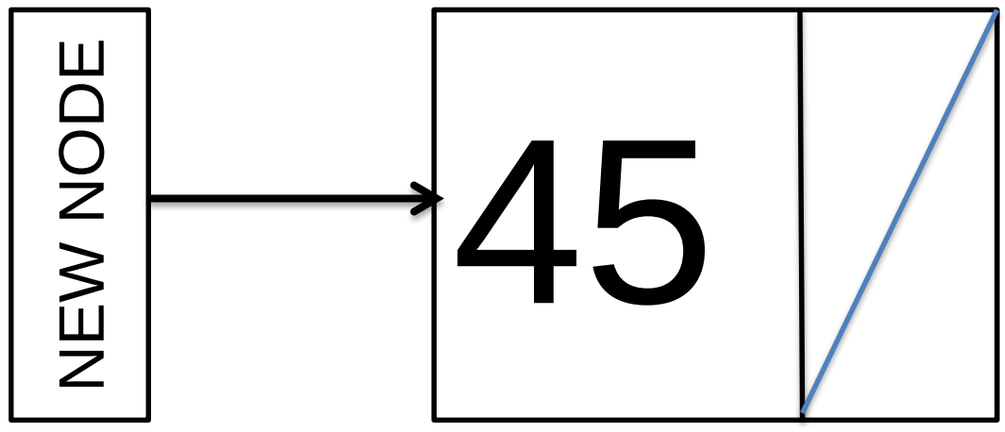
- When you pass o around
  - You are not giving the actual object
    - but a reference to it
- Object p = o;
  - p is assigned the **same object reference** as o
  - i.e. they point to the same object
  - It **does not** copy the object

Linked List Nodes

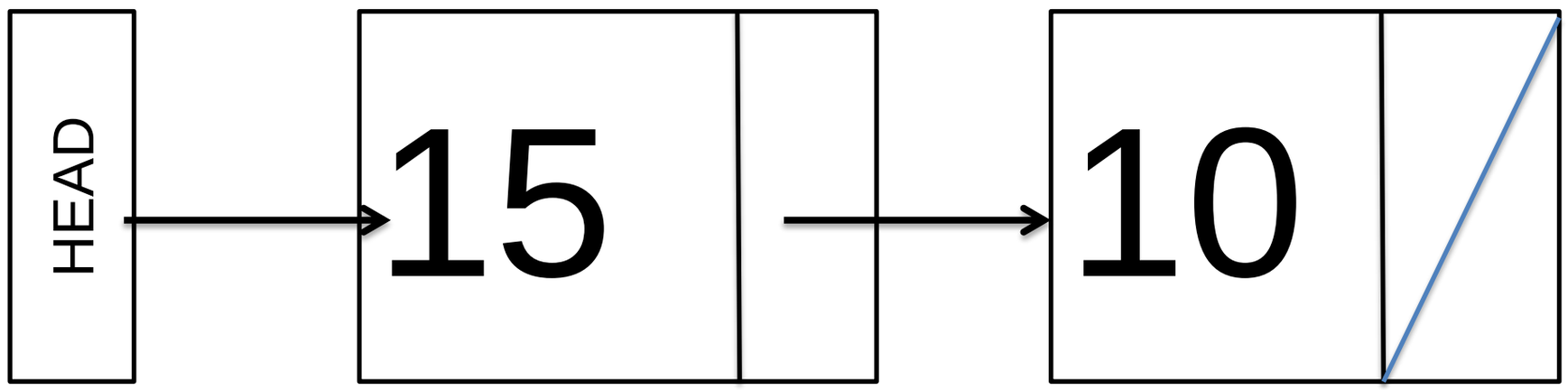


- We can only go in the direction of the arrows
  - i.e. A holds a reference to B (B doesn't hold a reference to A)
- A Node without a sibling should have its reference set to **null**
  - We need to be explicit and careful about this in code
  - This represents the end of DDS

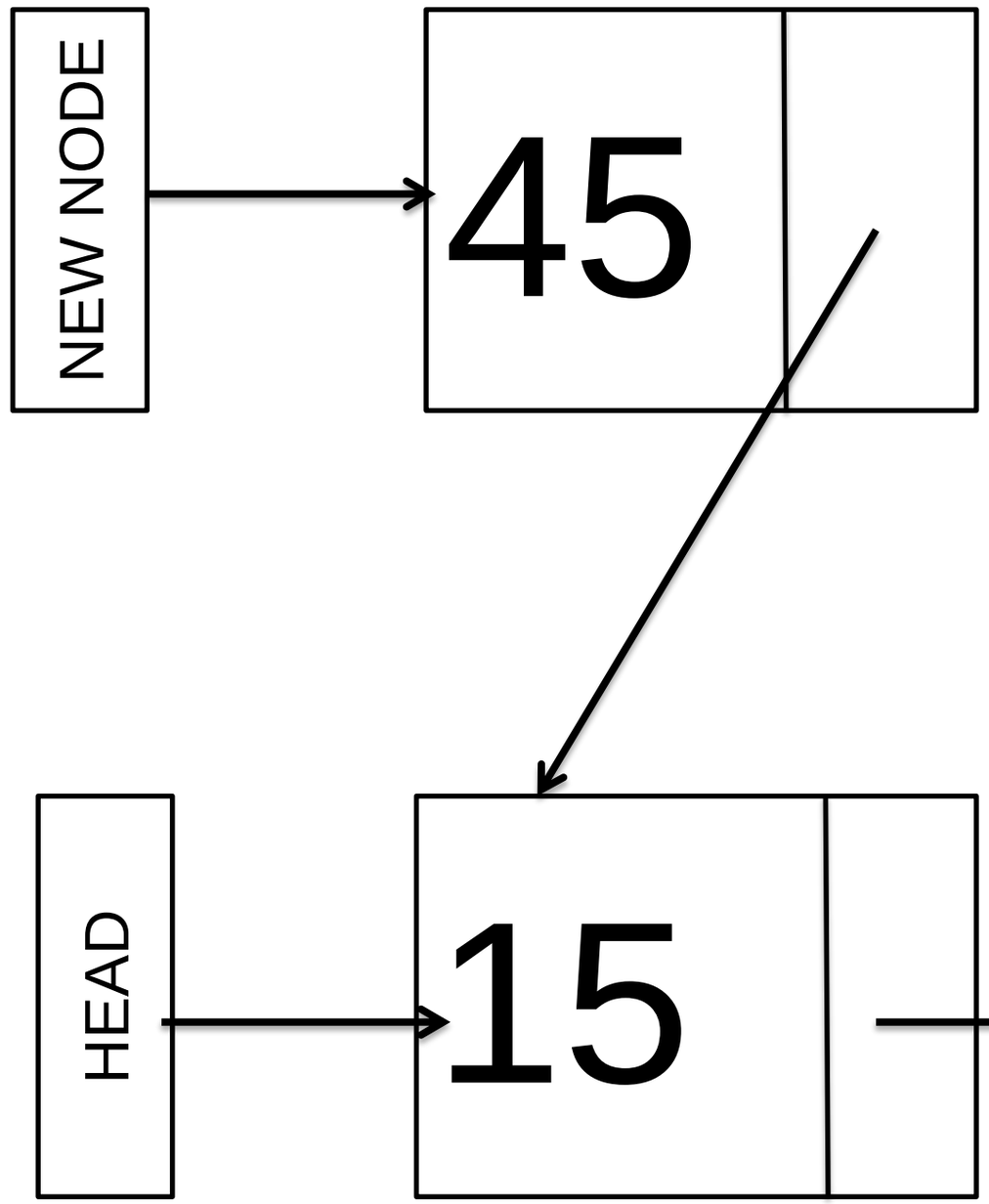
Insert at Head



1. Create the new node

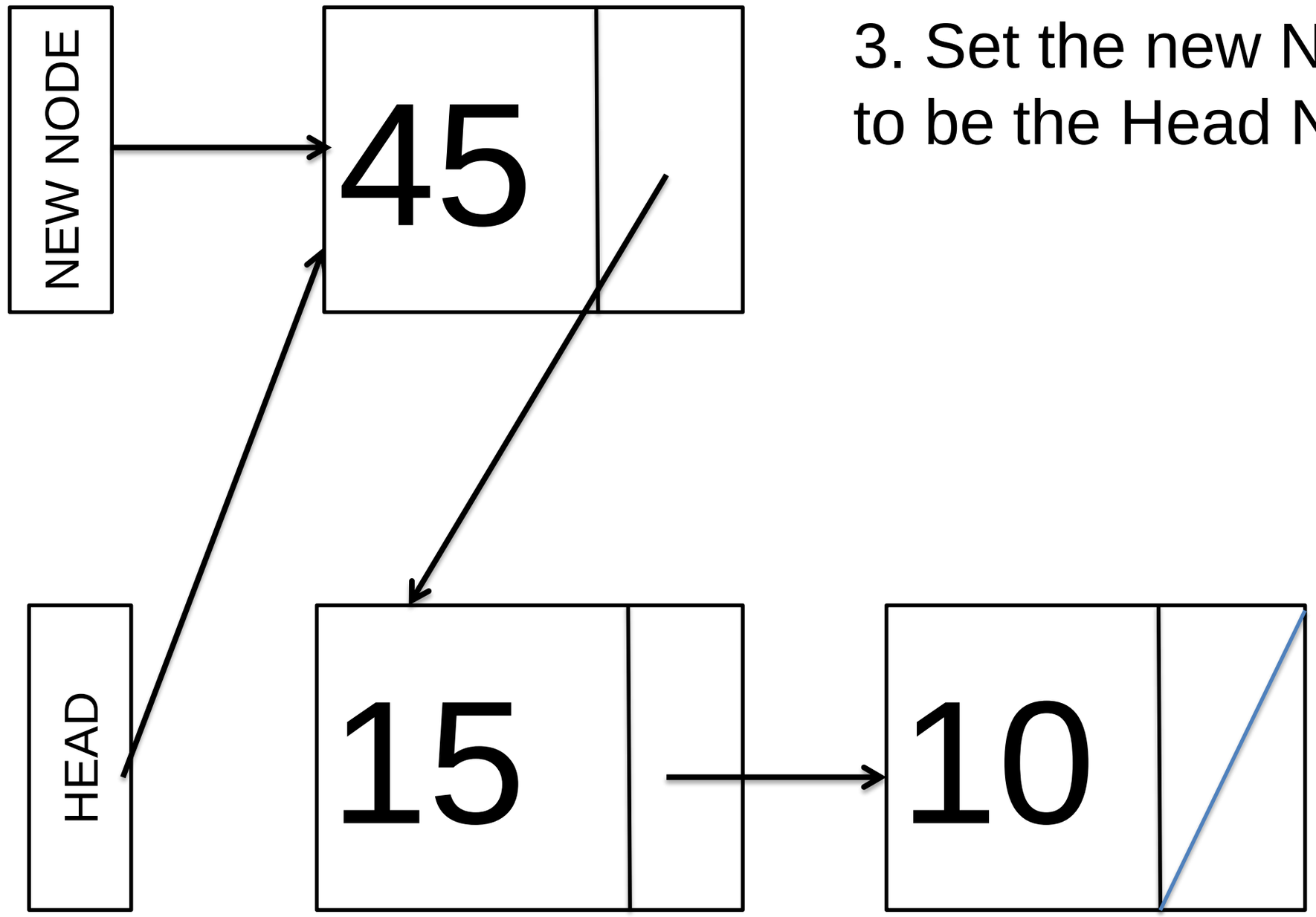


Insert at Head



2. Set New node Reference to Existing Head Node

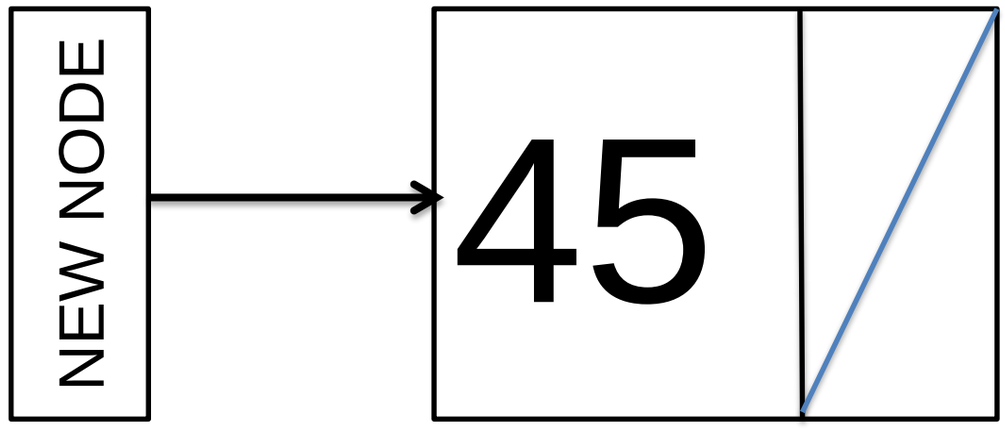
Insert at Head

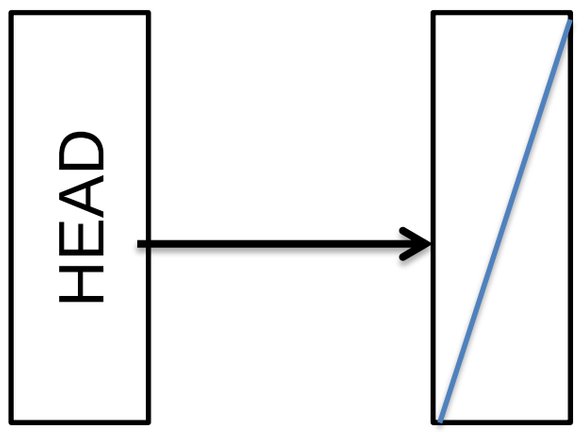


3. Set the new Node to be the Head Node

What if the list is Empty?

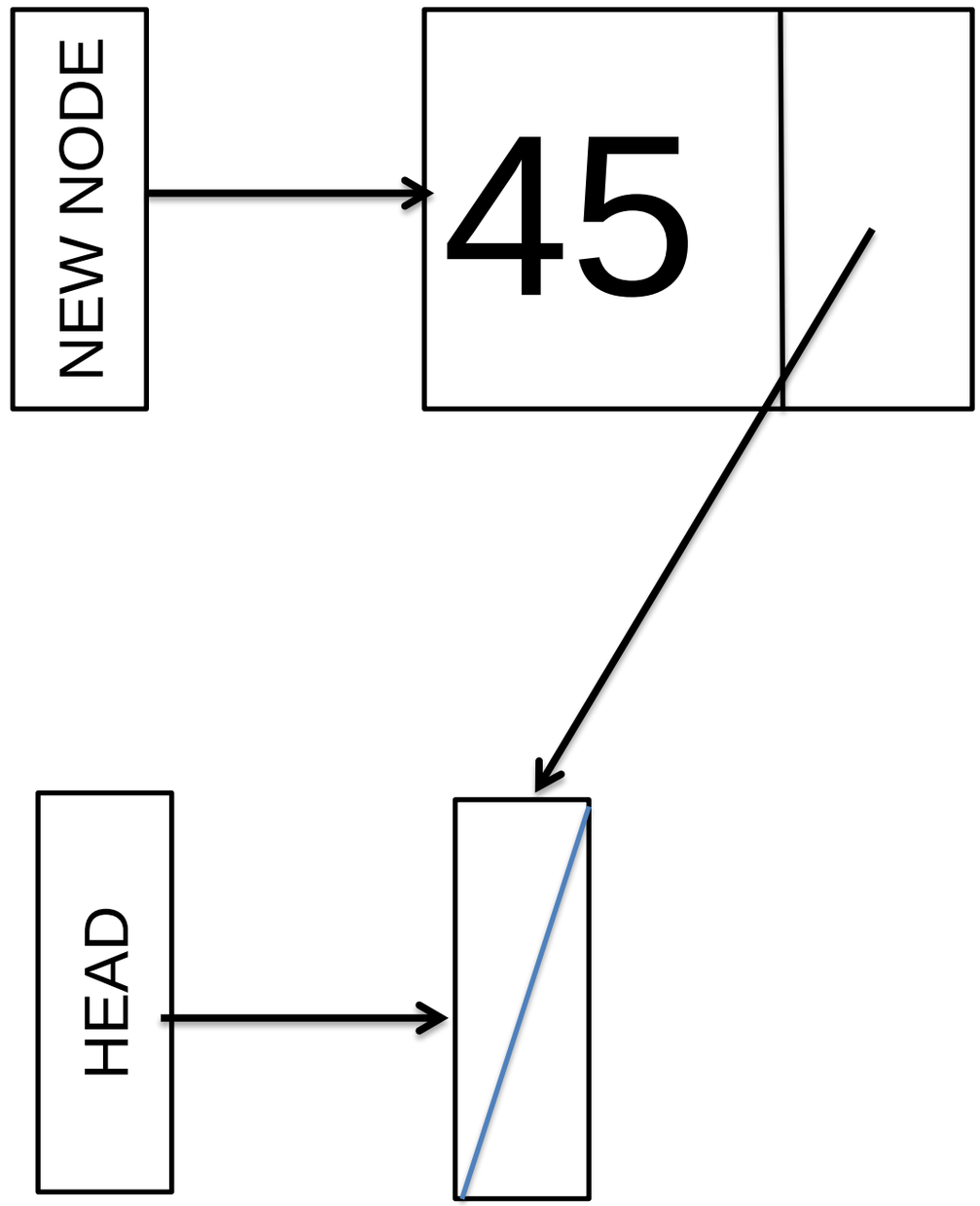
Insert at Head (Empty List)





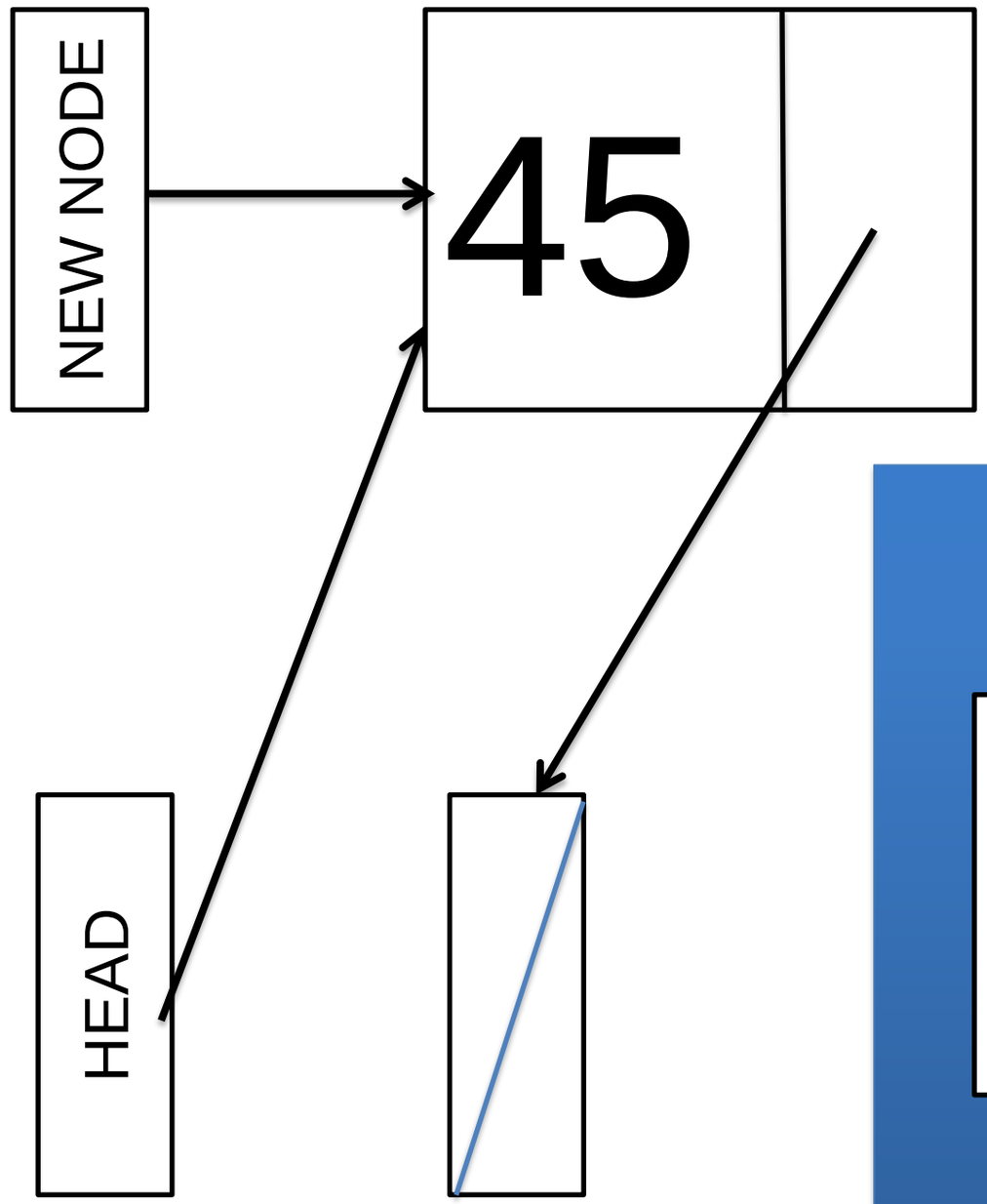
1. Create the new node

Insert at Head



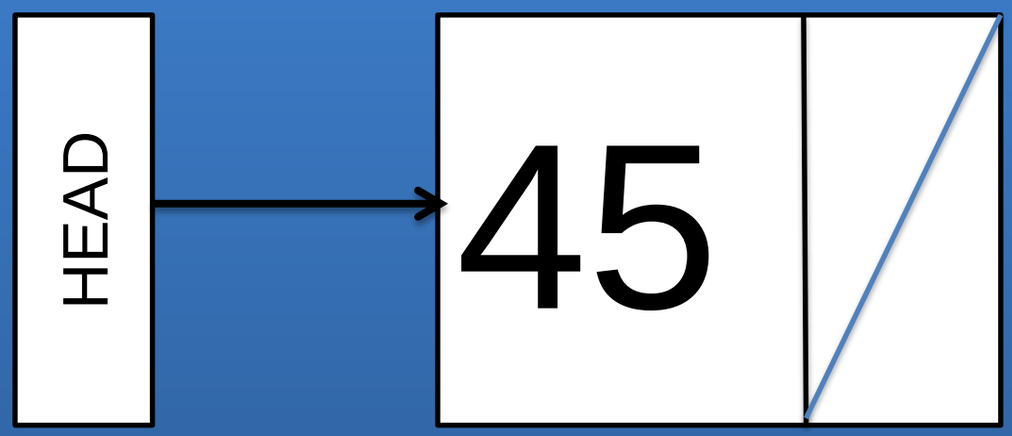
2. Set New node Reference to Existing Head Node

Insert at Head

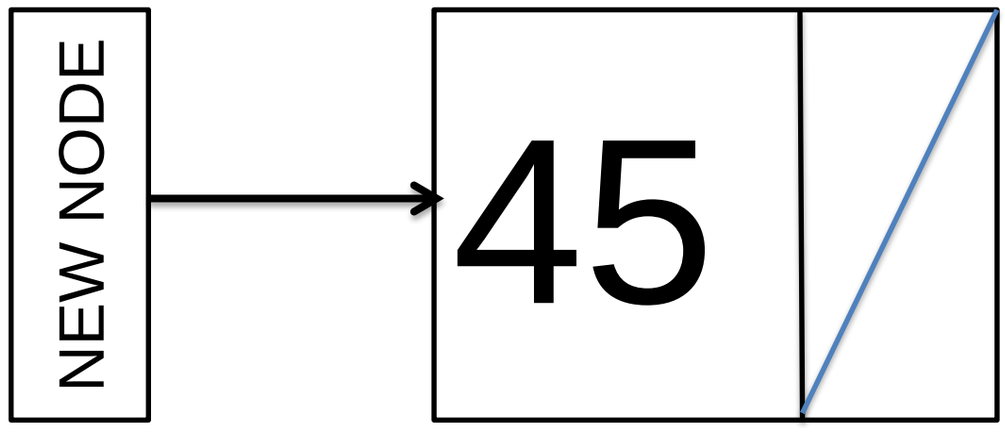


3. Set the new Node to be the Head Node

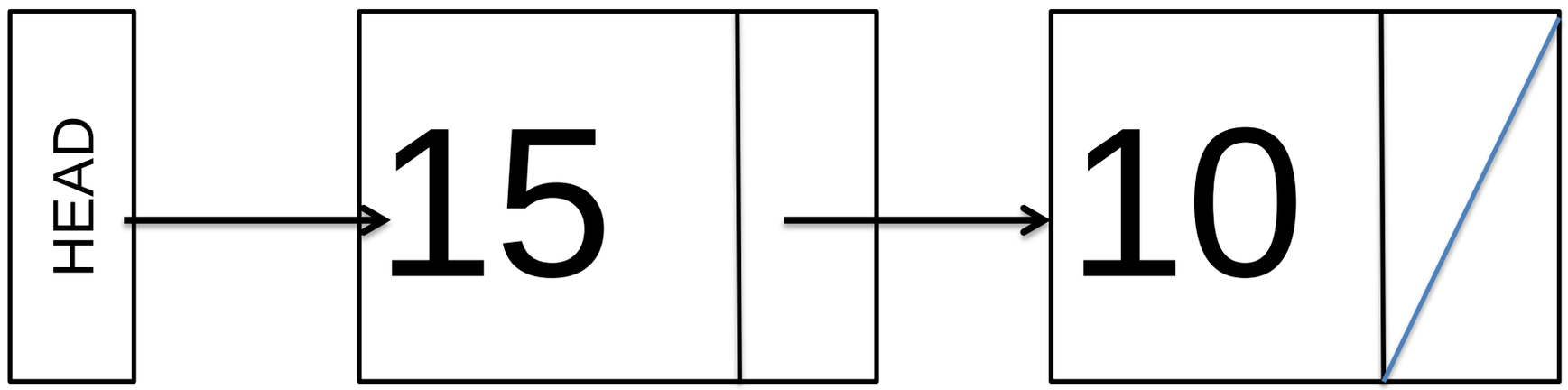
Which is the same as:



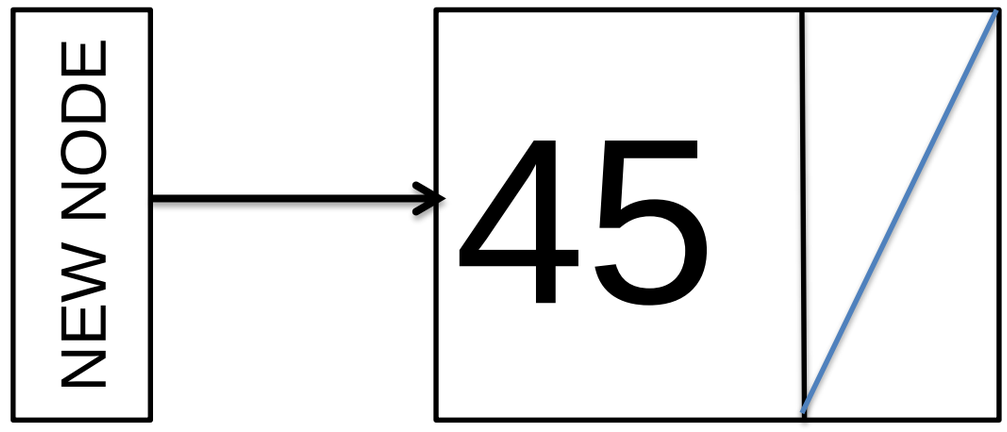
Insert at Tail



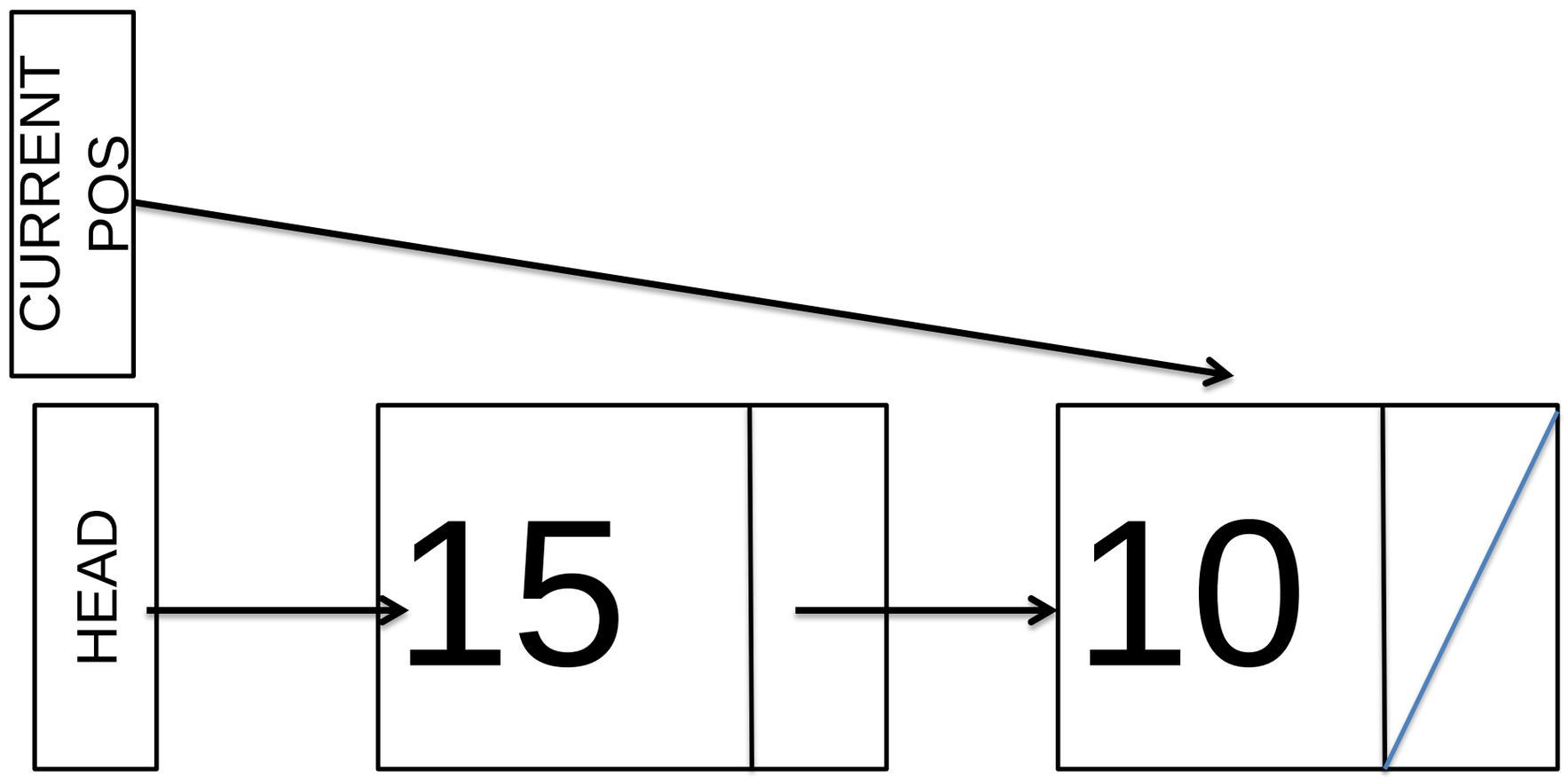
1. Create the new node



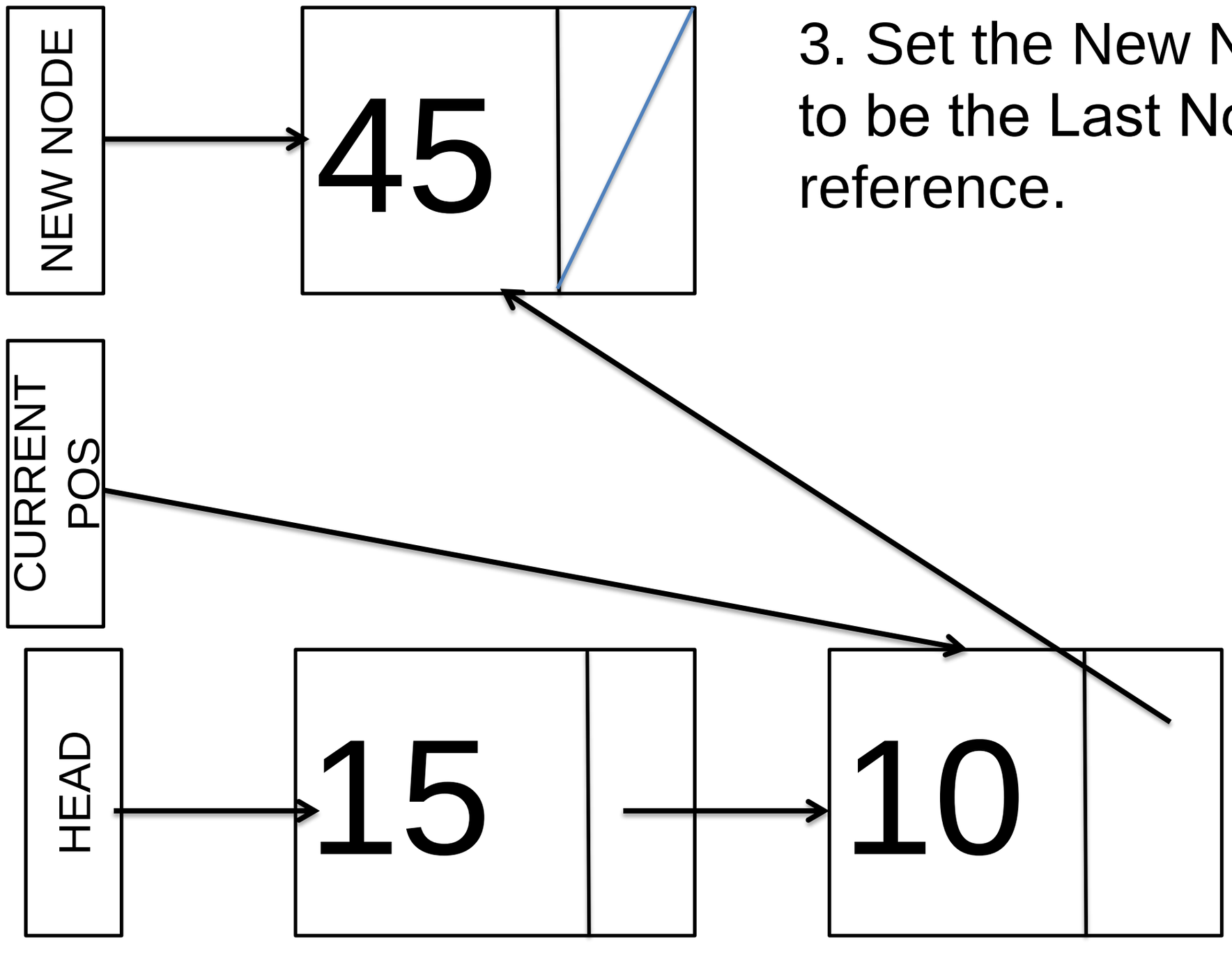
Insert at Tail



2. Iterate to Find End of List



Insert at Tail

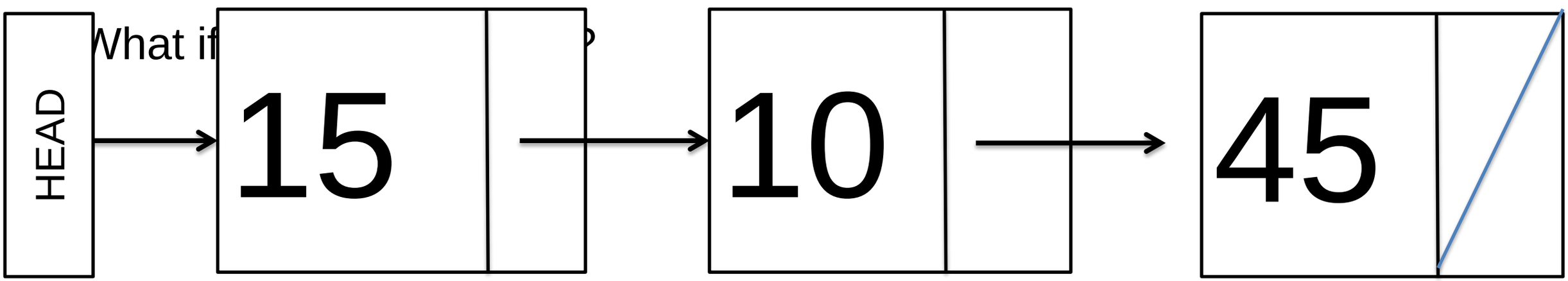


3. Set the New Node to be the Last Node's reference.

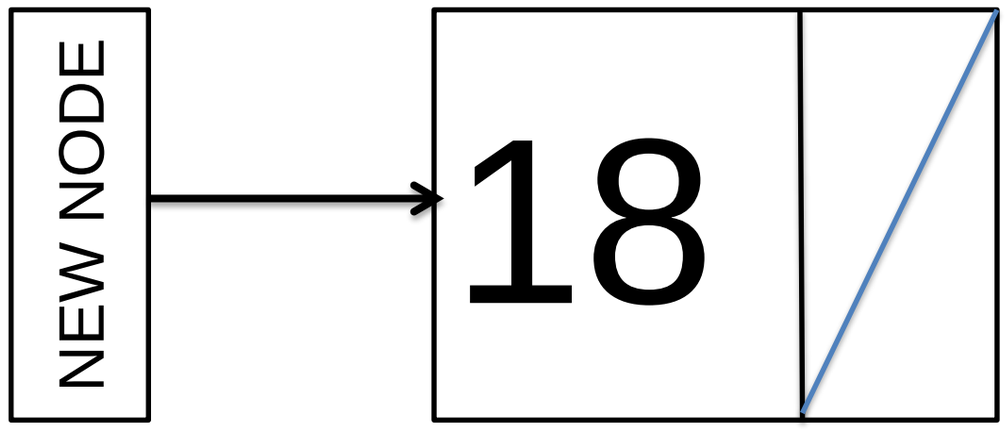
Insert at Tail

- Which Leaves us with this..
- Note because we add at end there is no final fix-up step

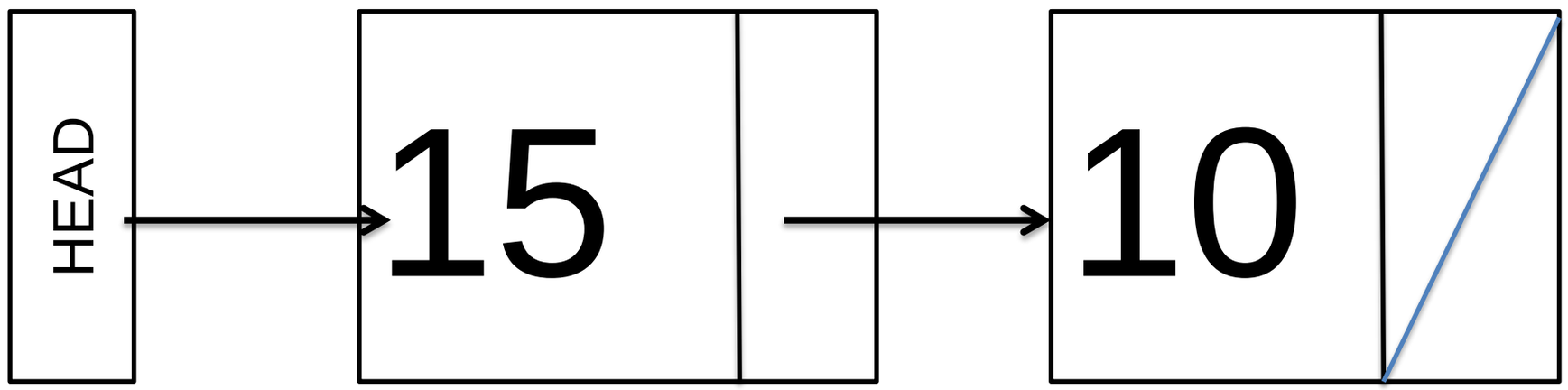
What if



Insert in the middle

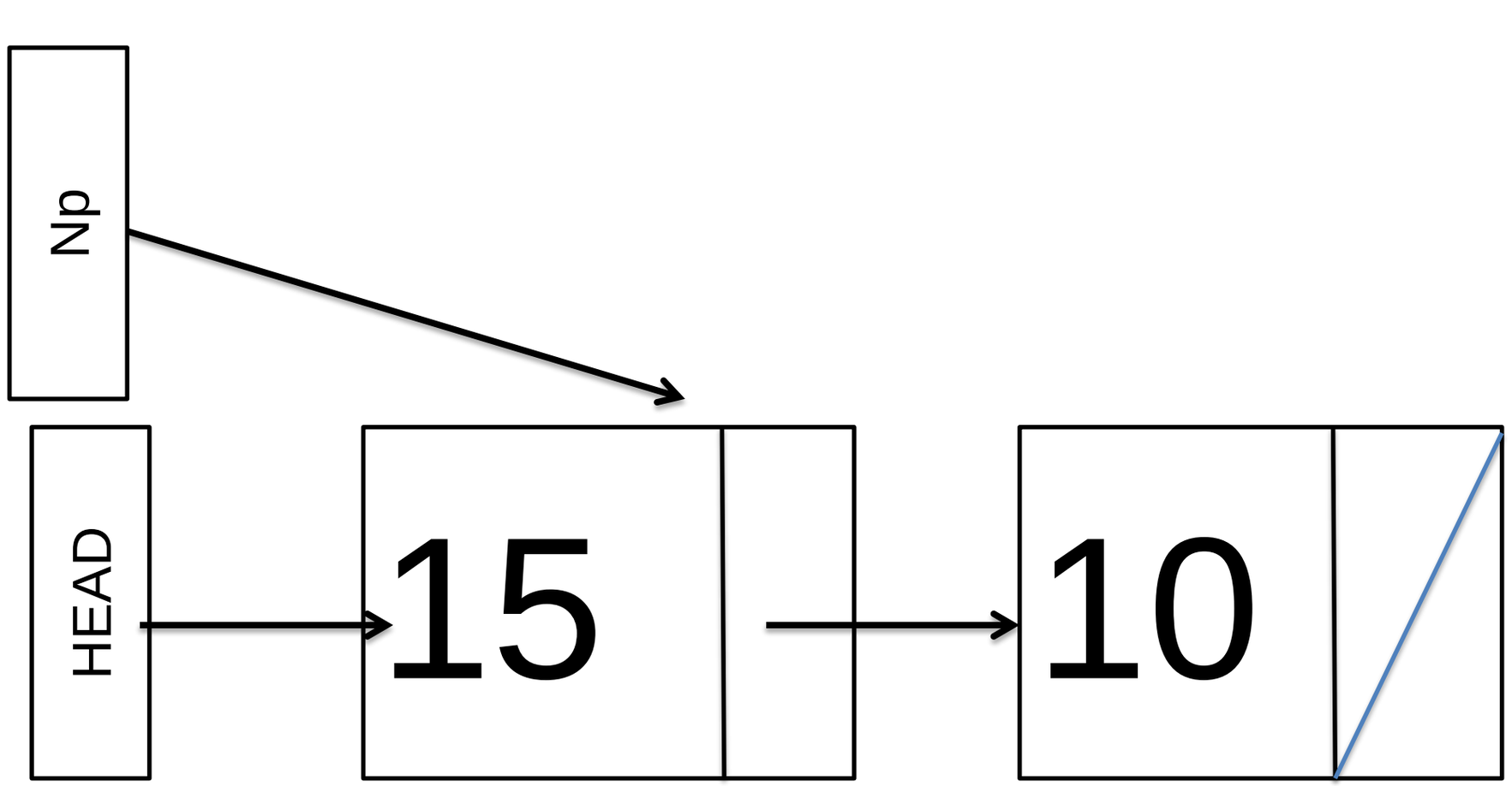


1. Create the new node

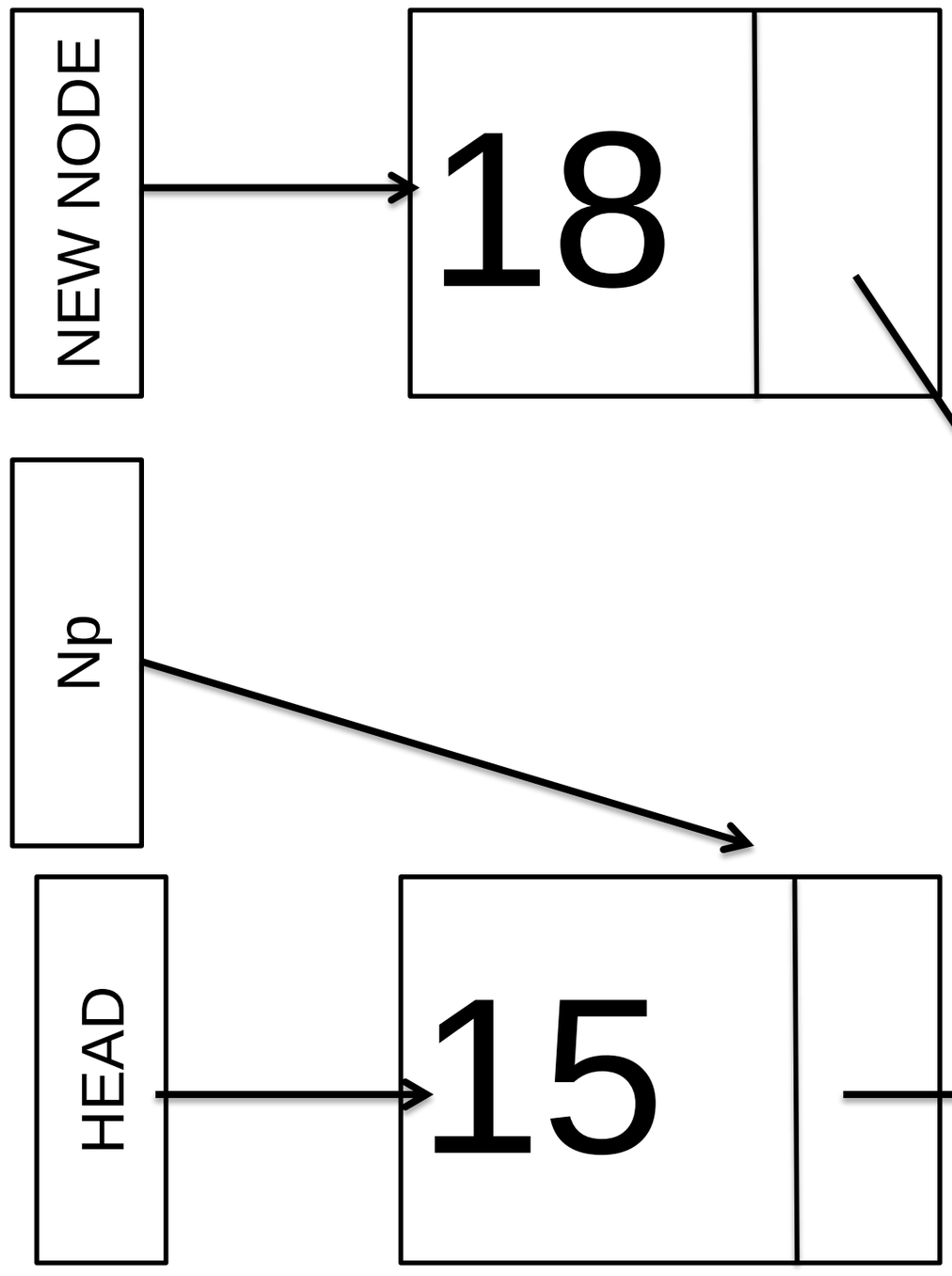


Insert in the middle

2. Find the Prior Node (Np) – Note we are inserting in order here



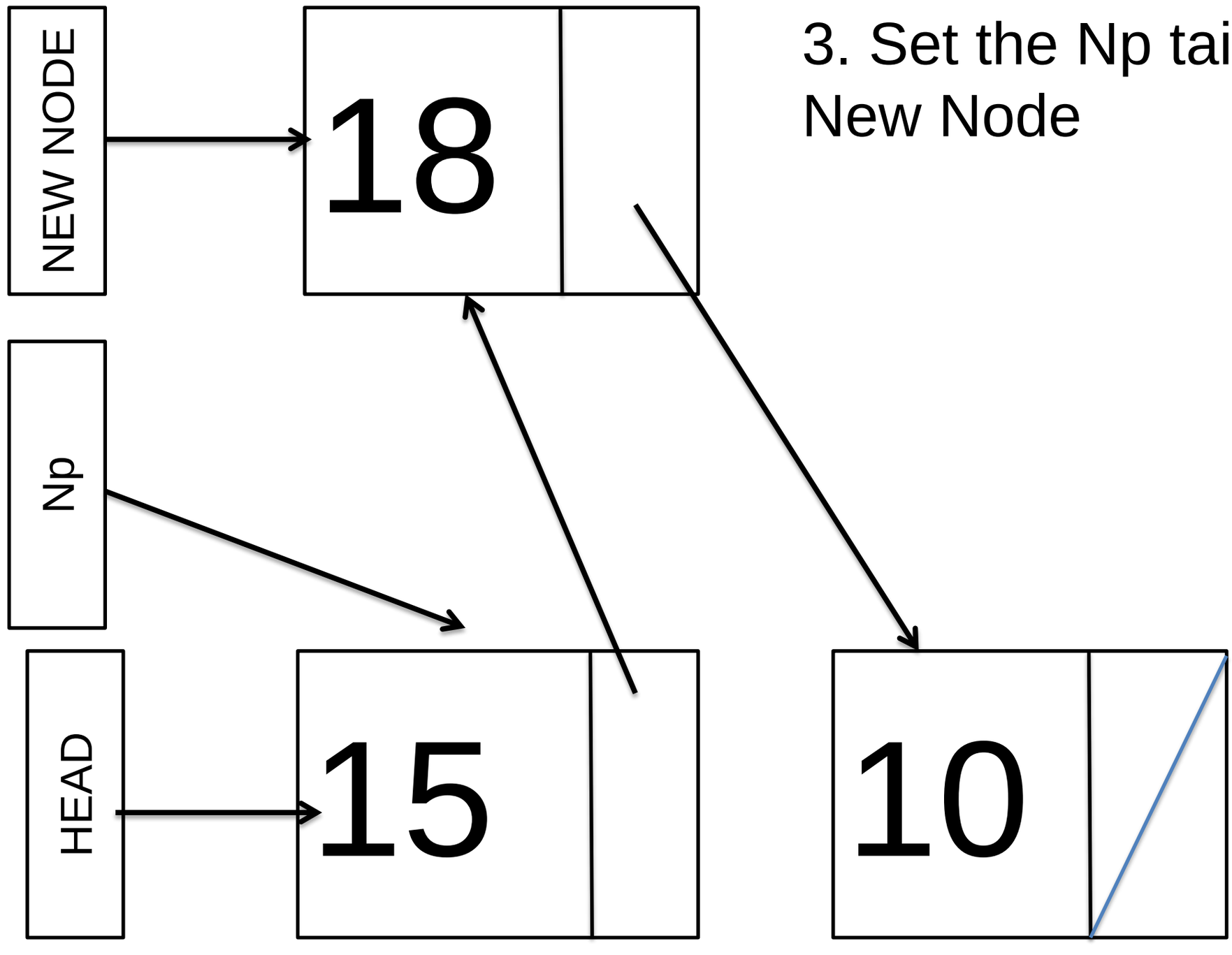
Insert in the middle



2. Set Np tail as New Node tail

WHY?

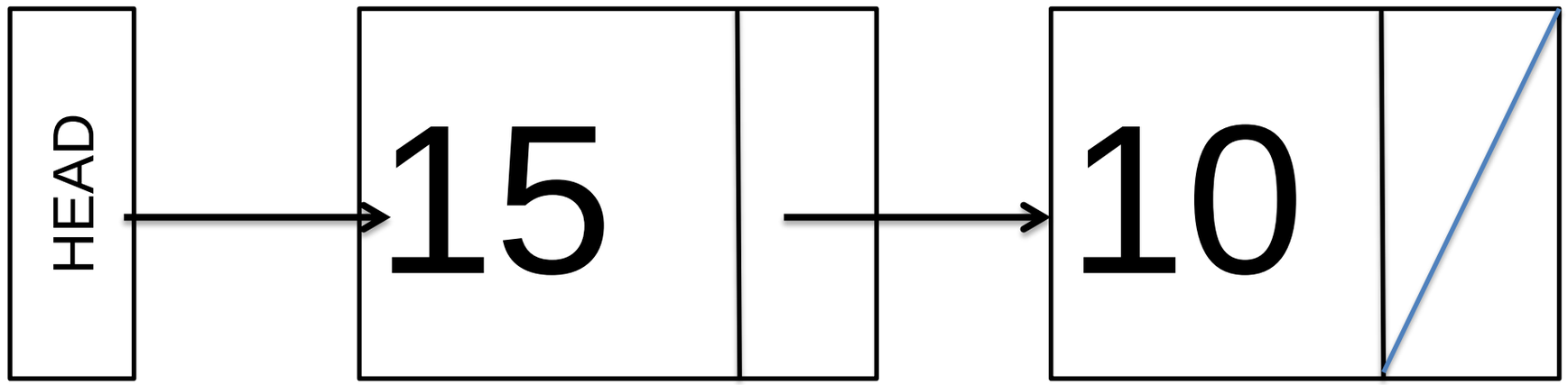
Insert in the middle



3. Set the Np tail to New Node

Delete from Head

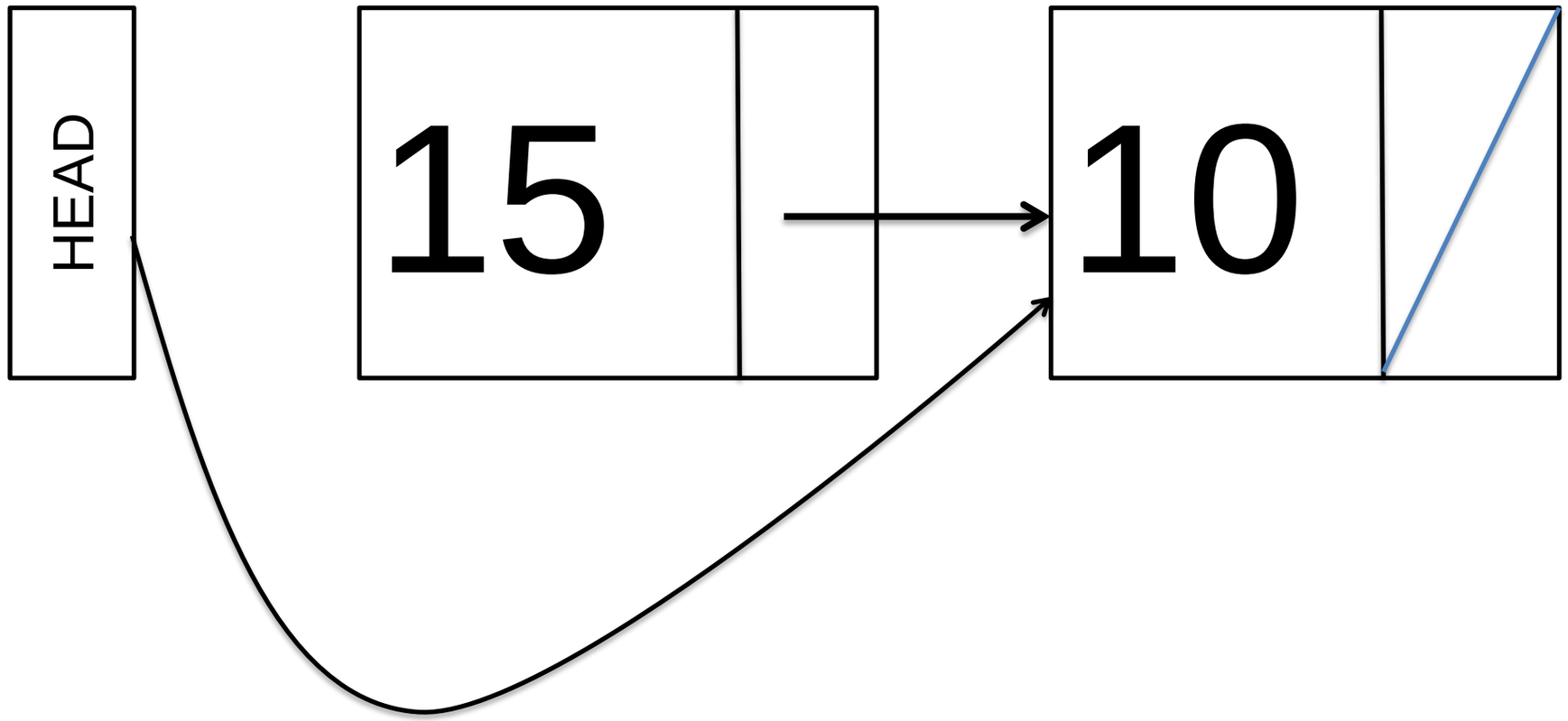
1. Check the List isn't Null



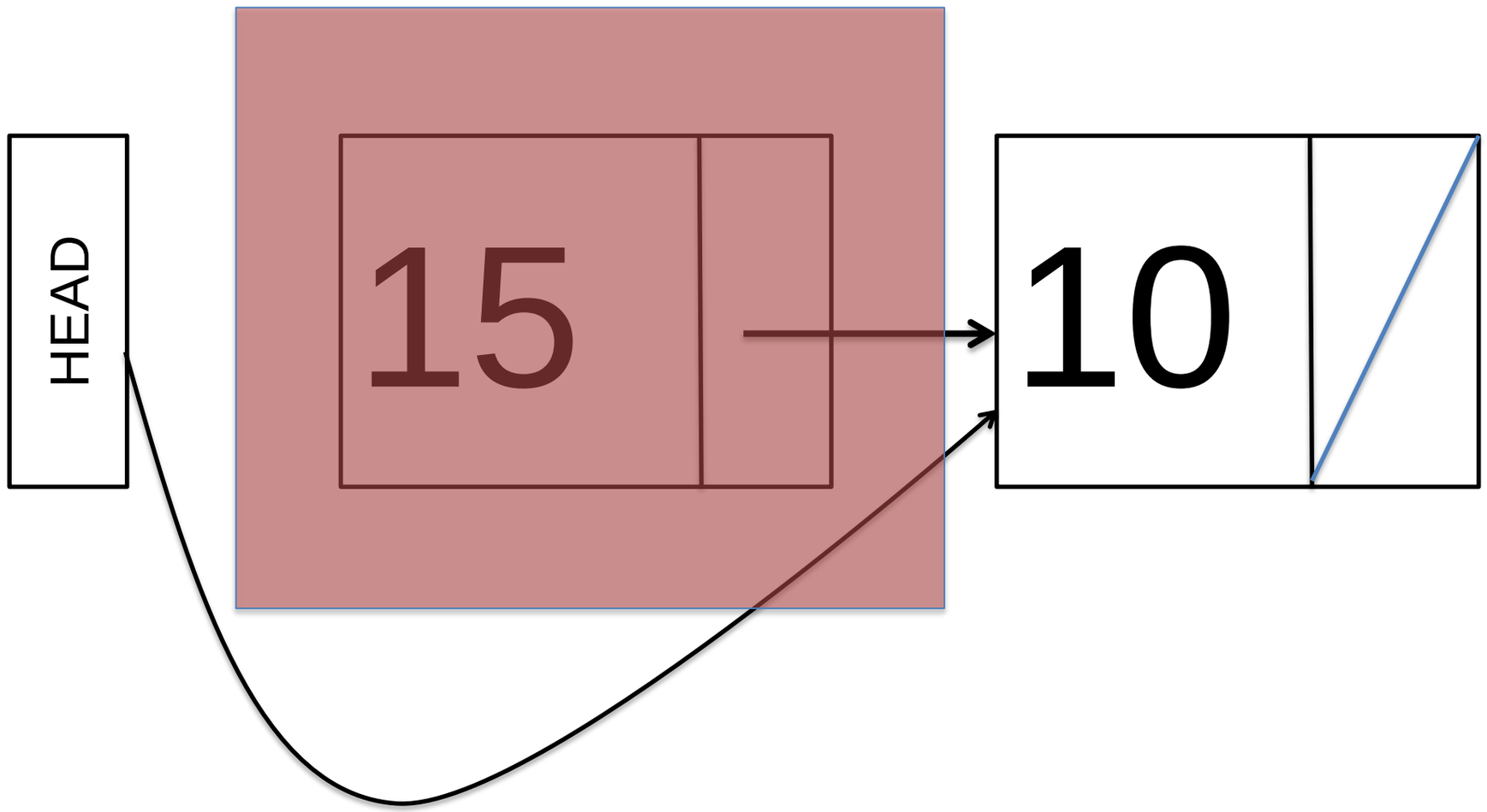
HEAD == NULL returns FALSE

Delete From Head

2. Set the Head to the First Node's Tail

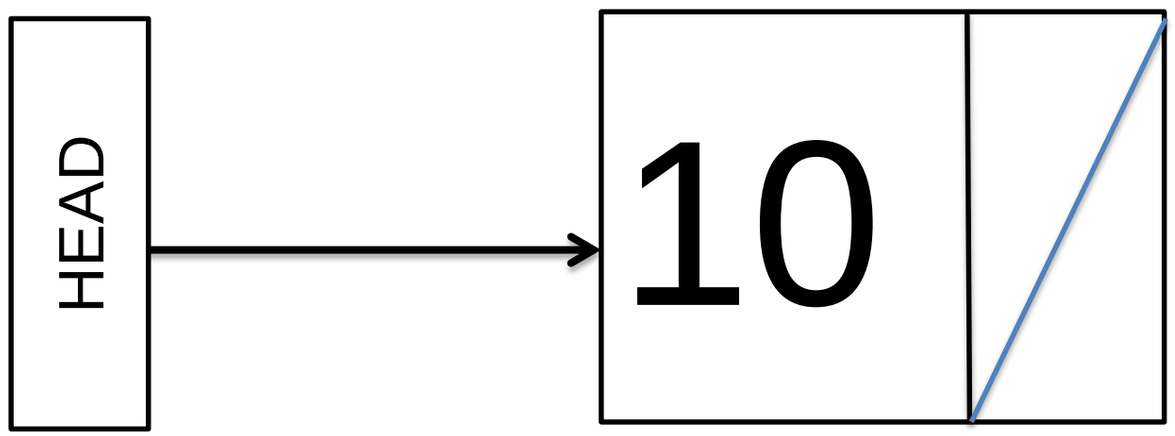


Delete From Head



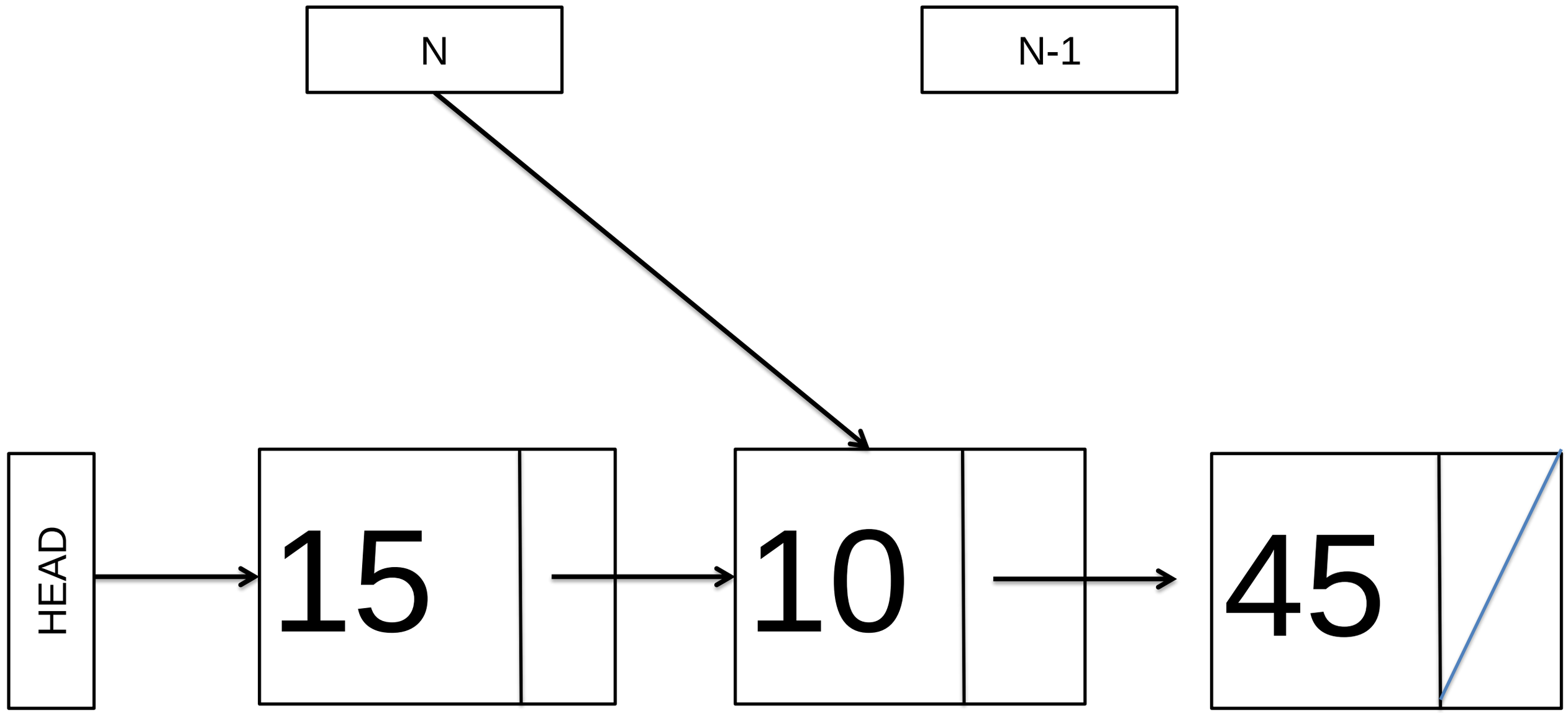
What Happens to this Node?

Delete From Head



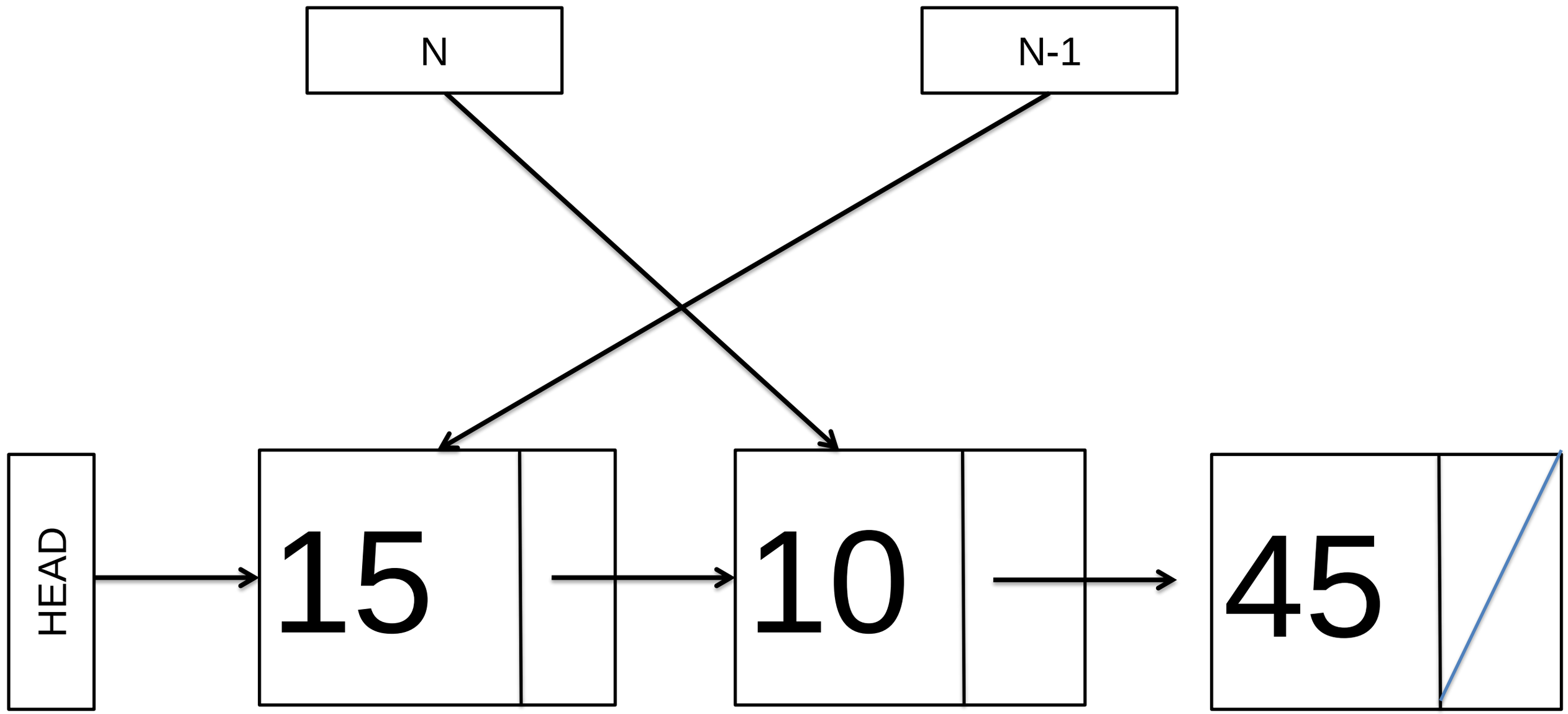
Delete Node 10

1. Find Node N



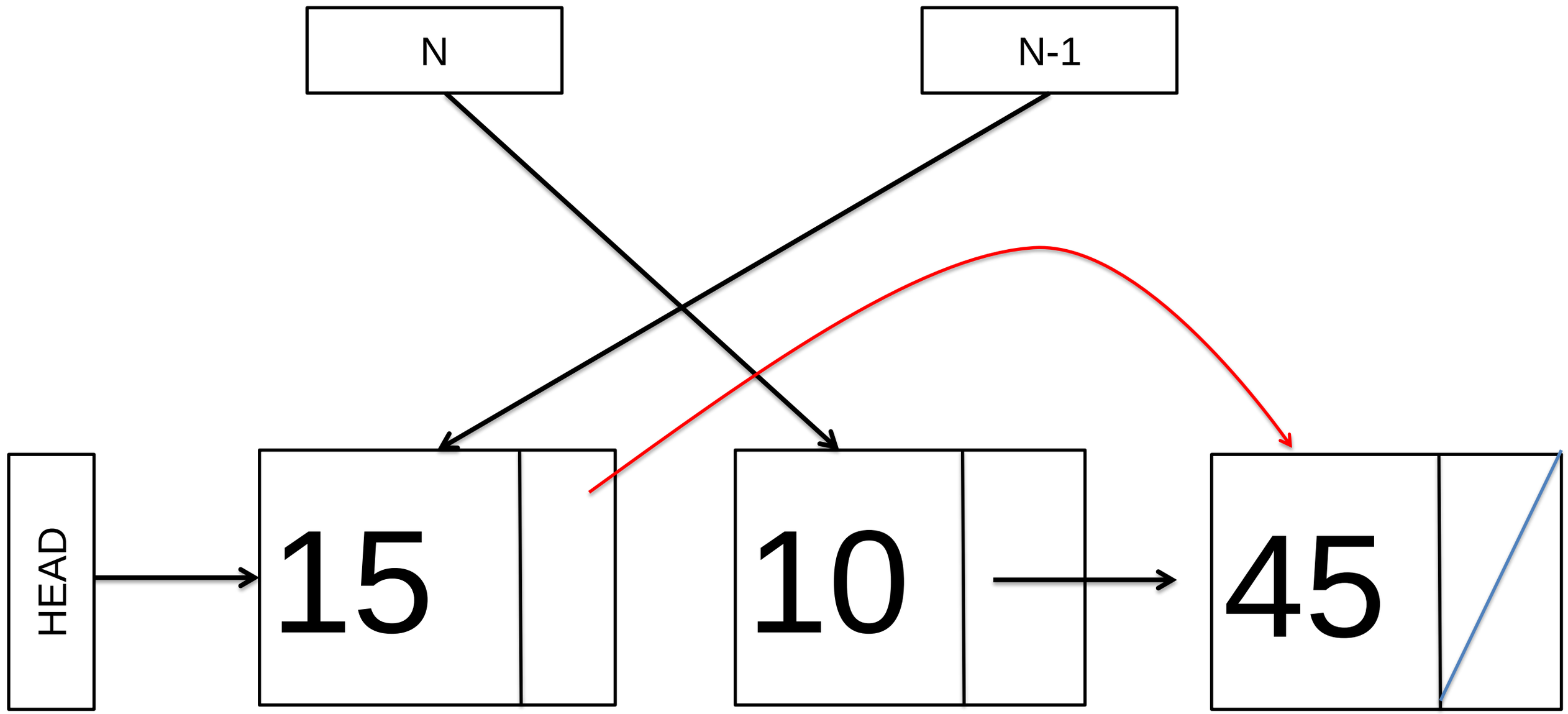
Delete Node 10

2. Find Node Prior to N (N-1



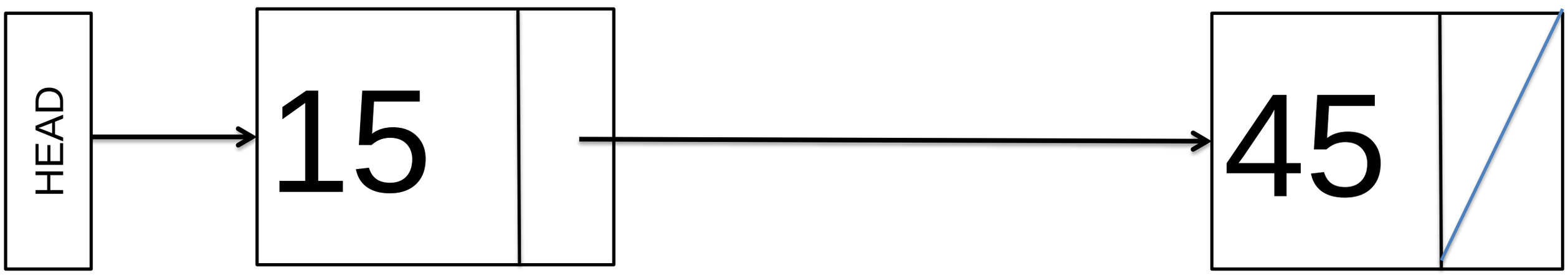
Delete Node 10

3. Make Node N-1 point to N+1



Delete Node 10

Variables N and N-1 are temp so after garbage collection



Overview of data structures so far

Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

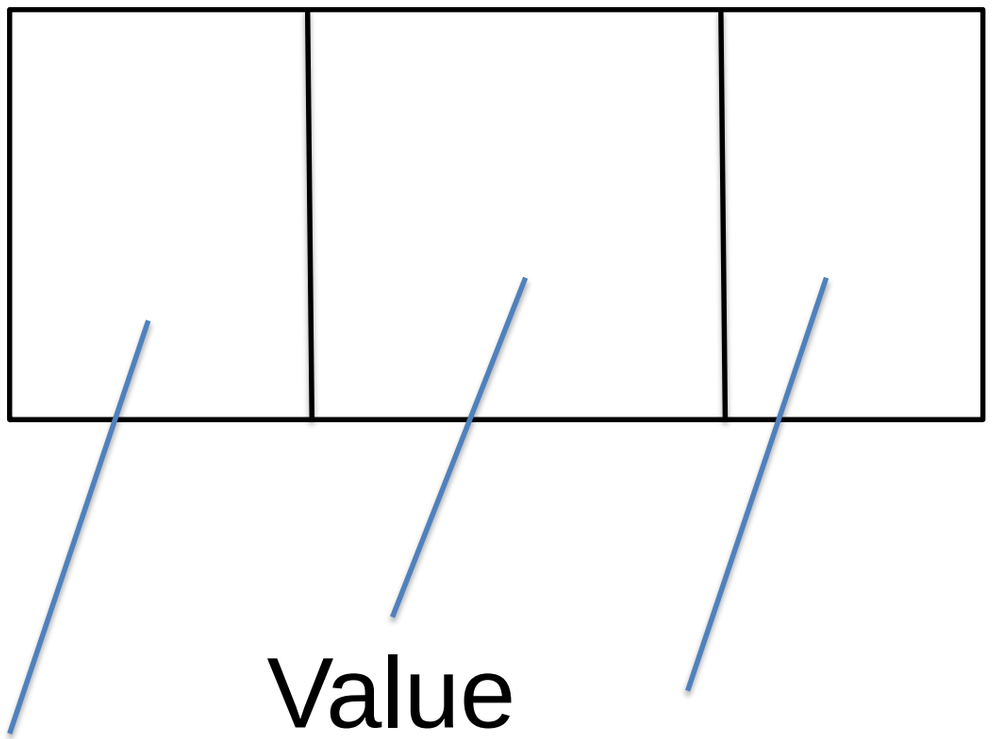
Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

Doubly Linked List Nodes

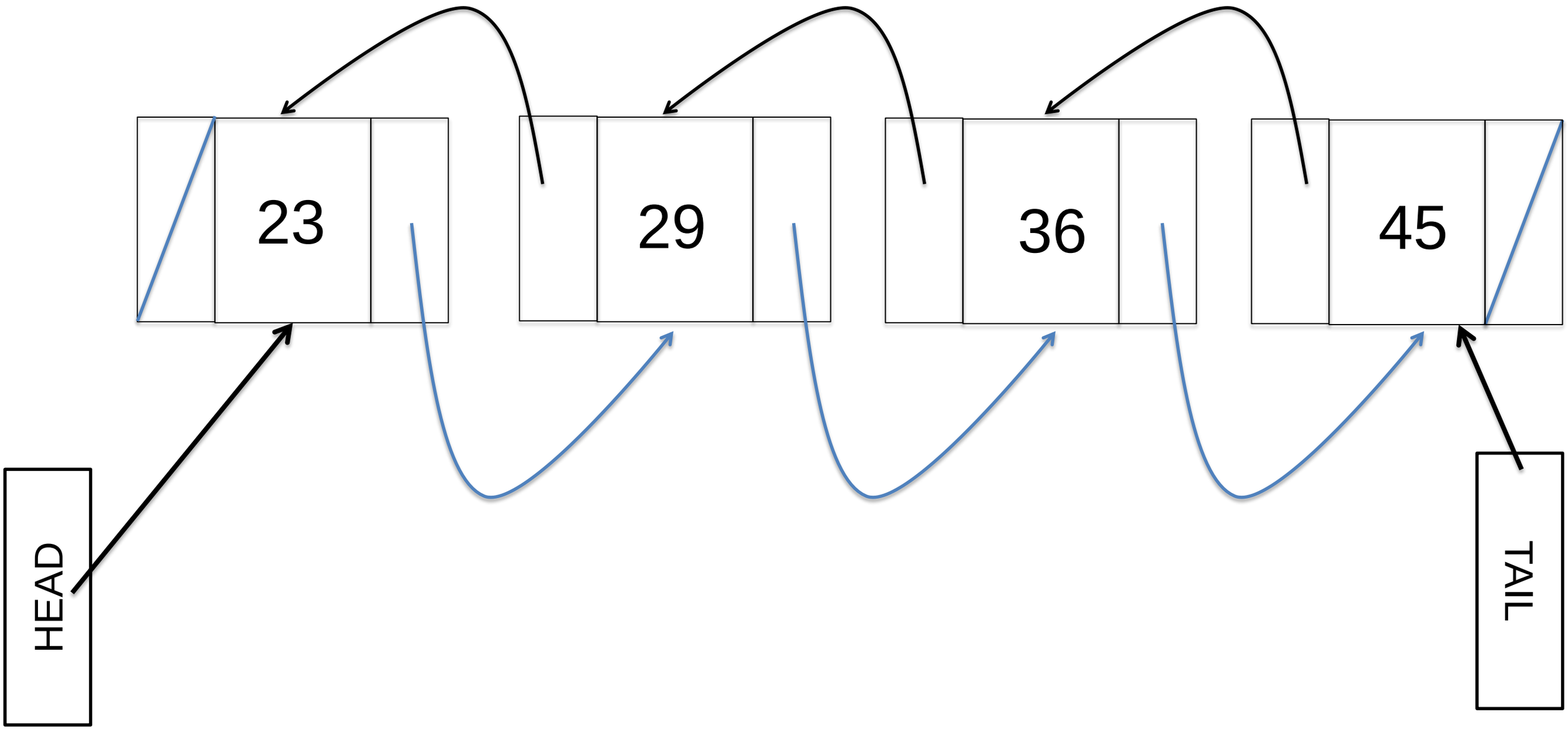
```
public class MyNode{
  public Object value;
  public MyNode nextNode;
  public MyNode prevNode;

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



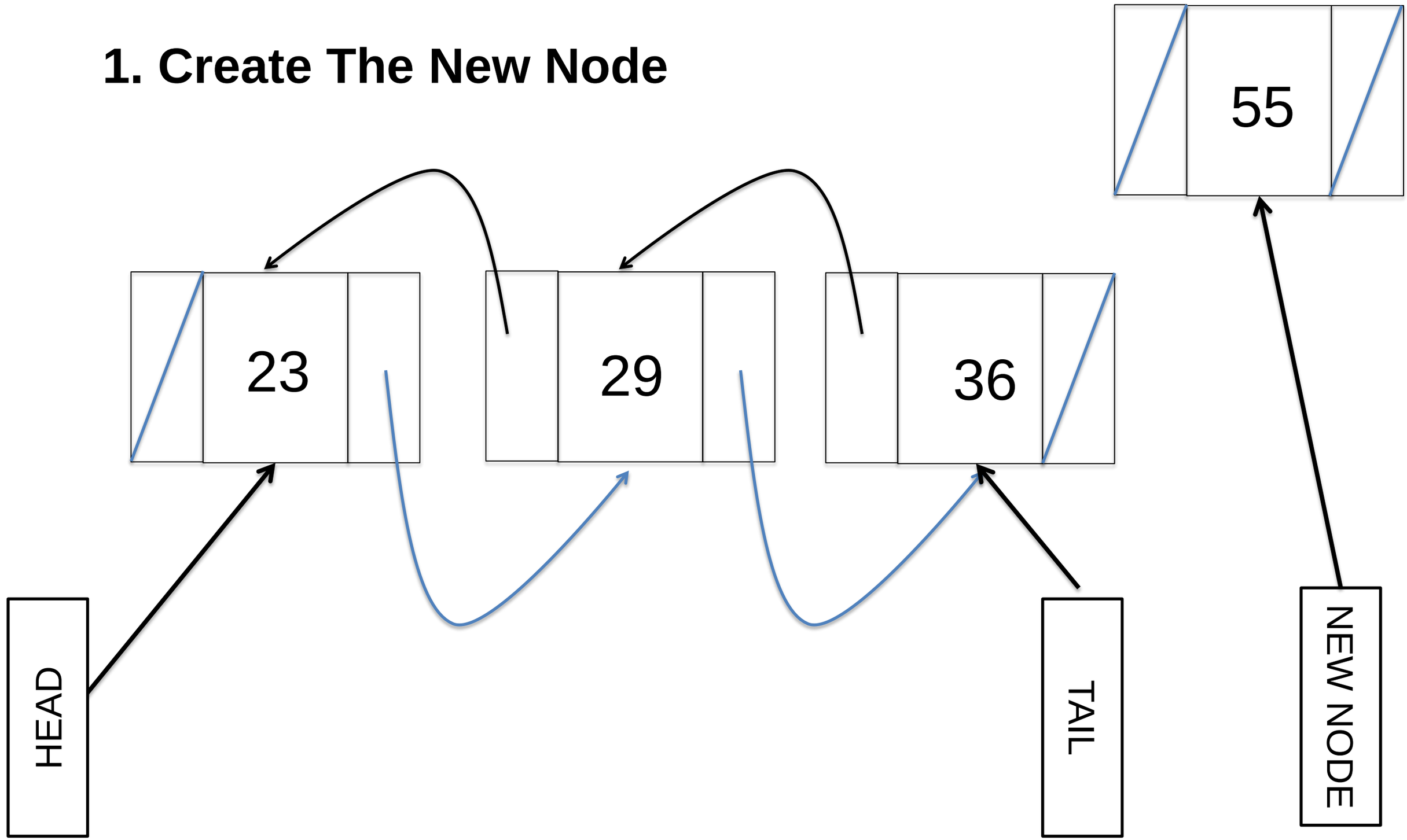
Doubly Linked Lists

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



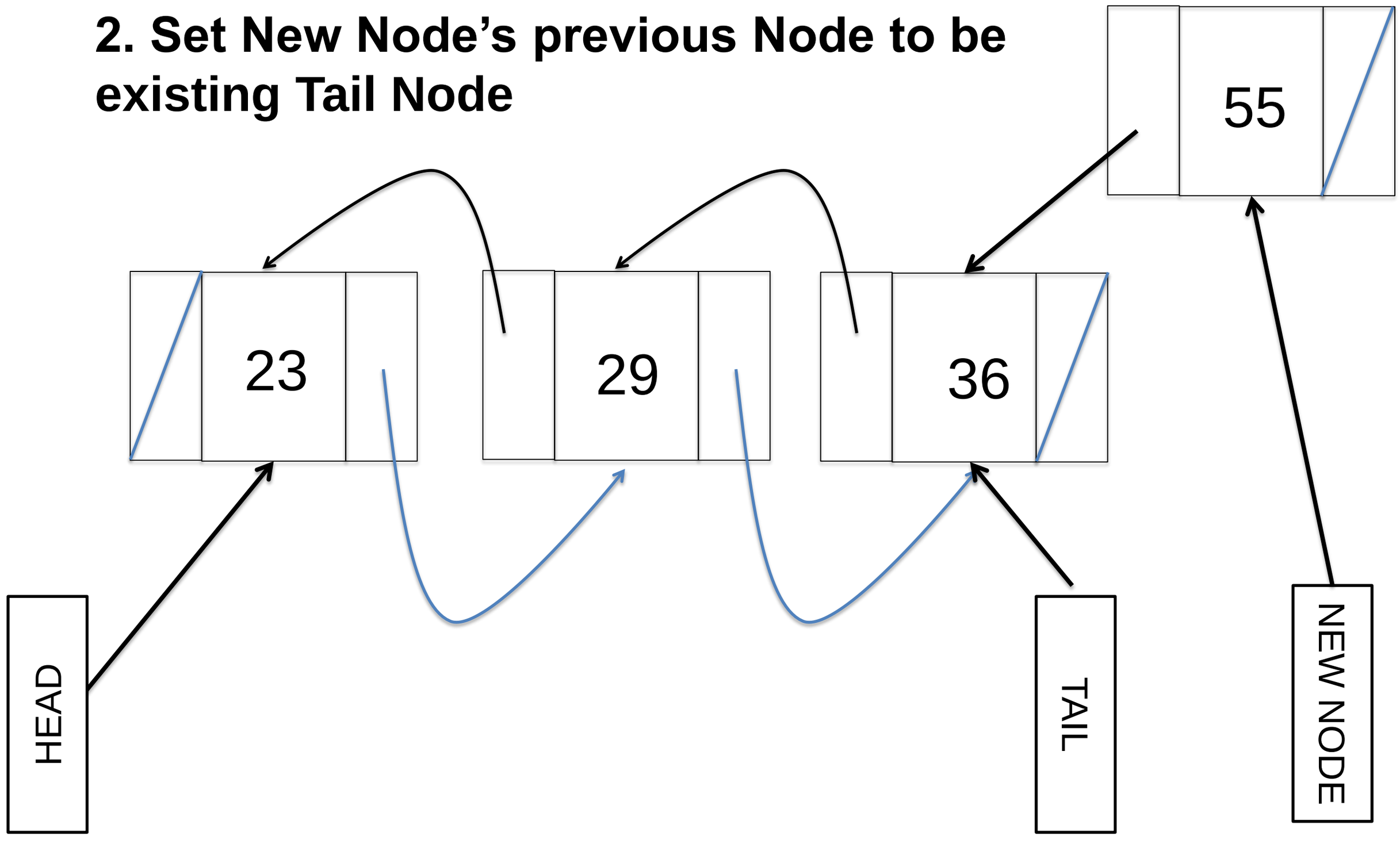
Insert At Tail

1. Create The New Node



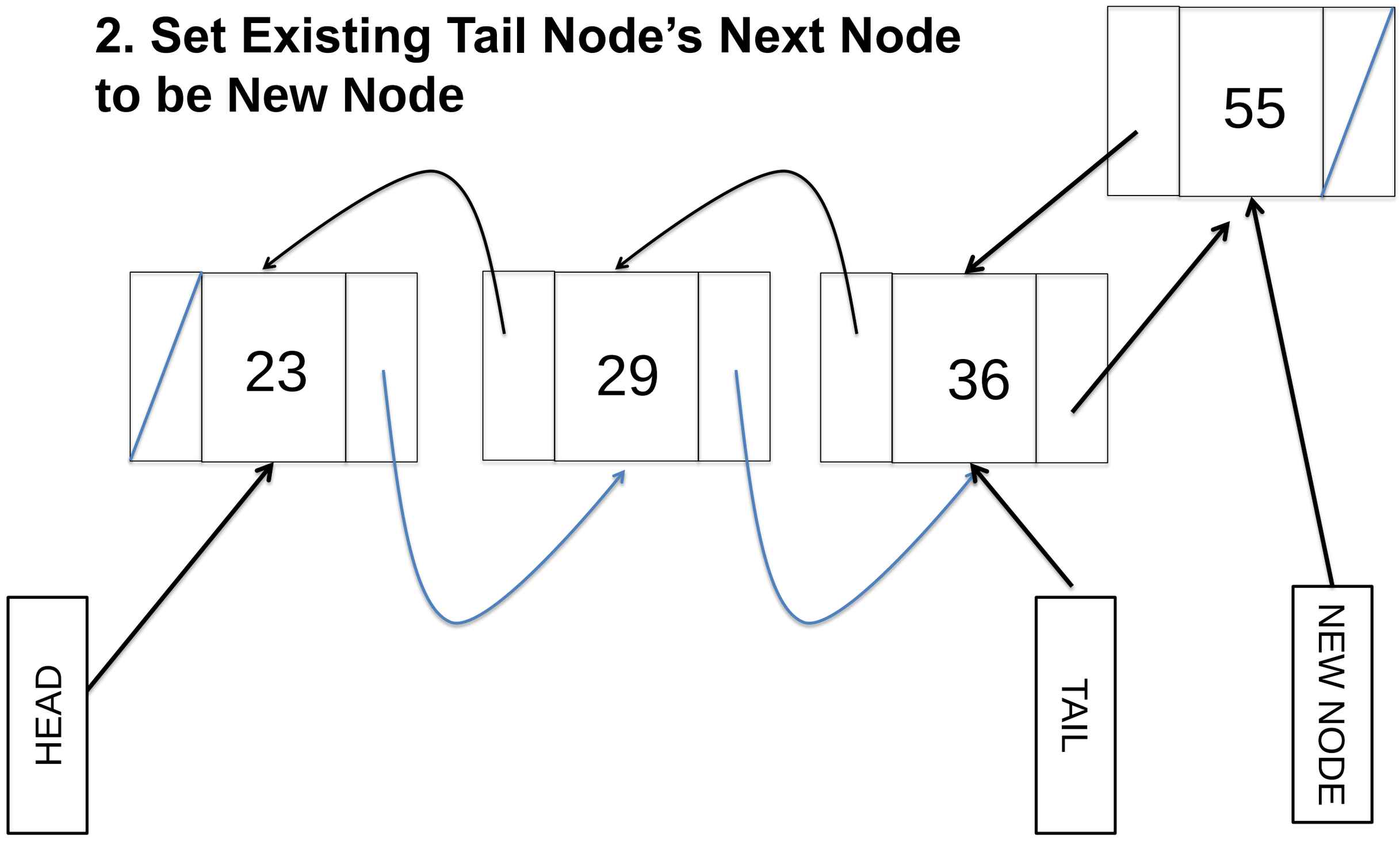
Insert At Tail

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



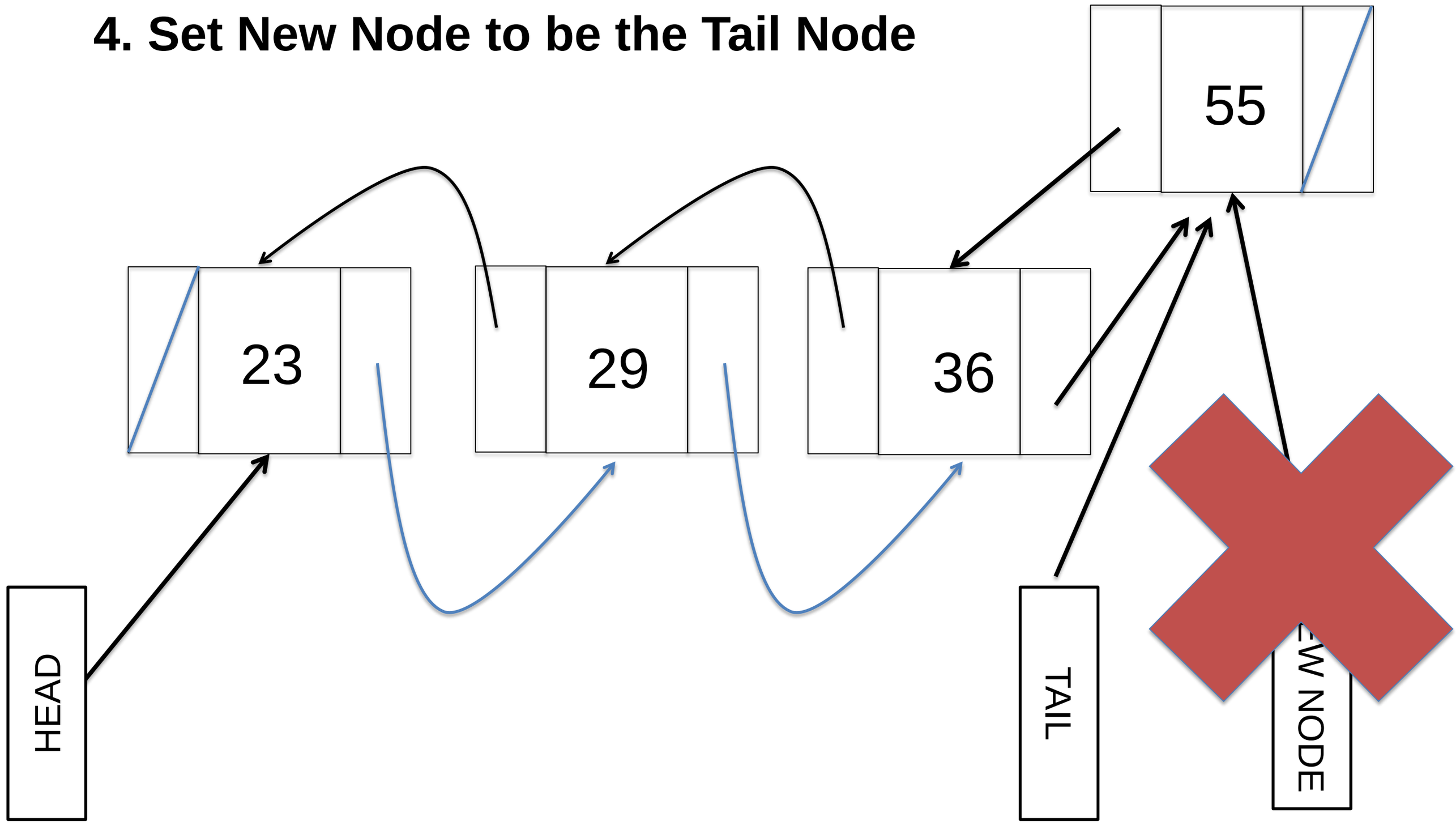
Insert At Tail

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



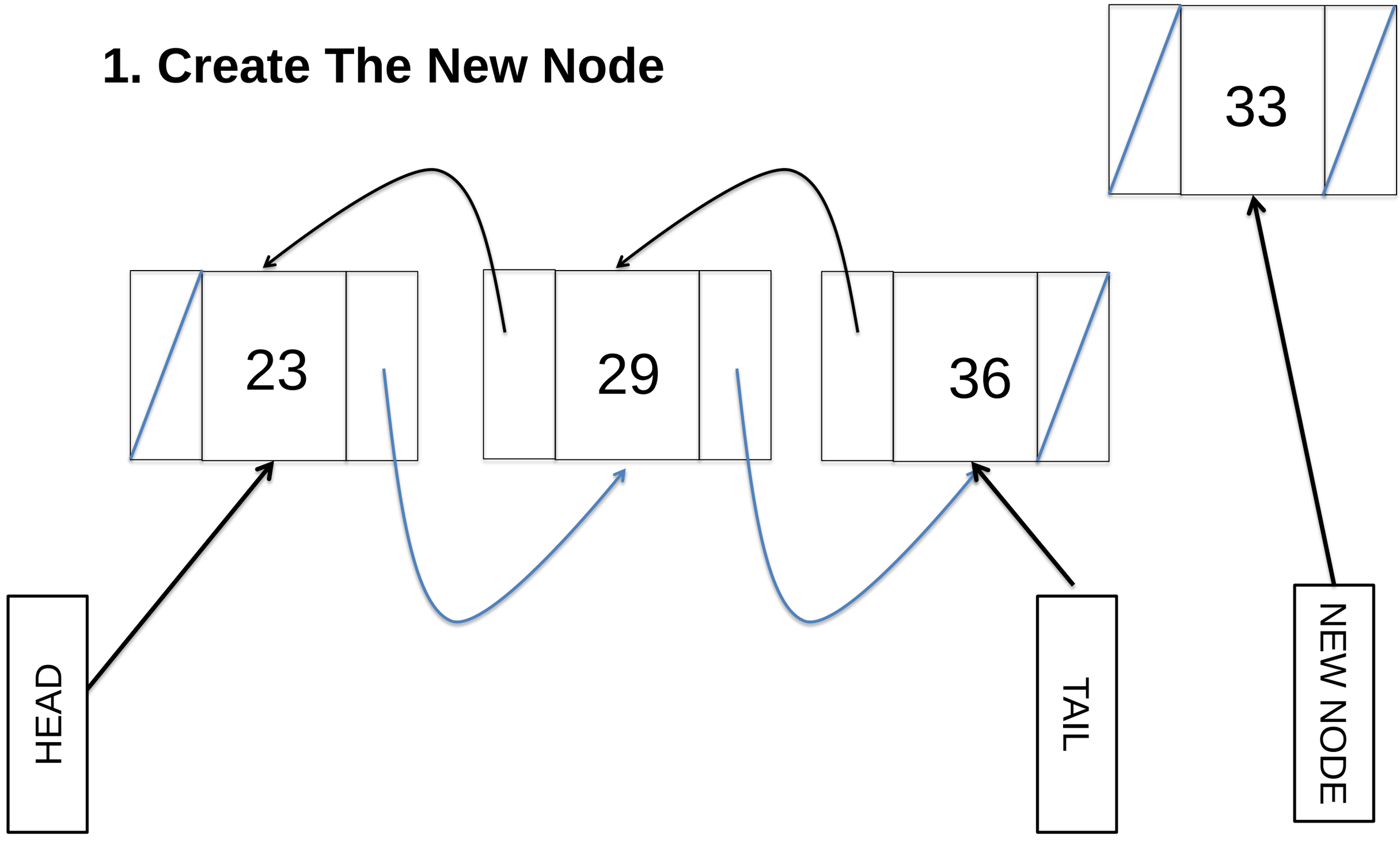
Insert At Tail

4. Set New Node to be the Tail Node



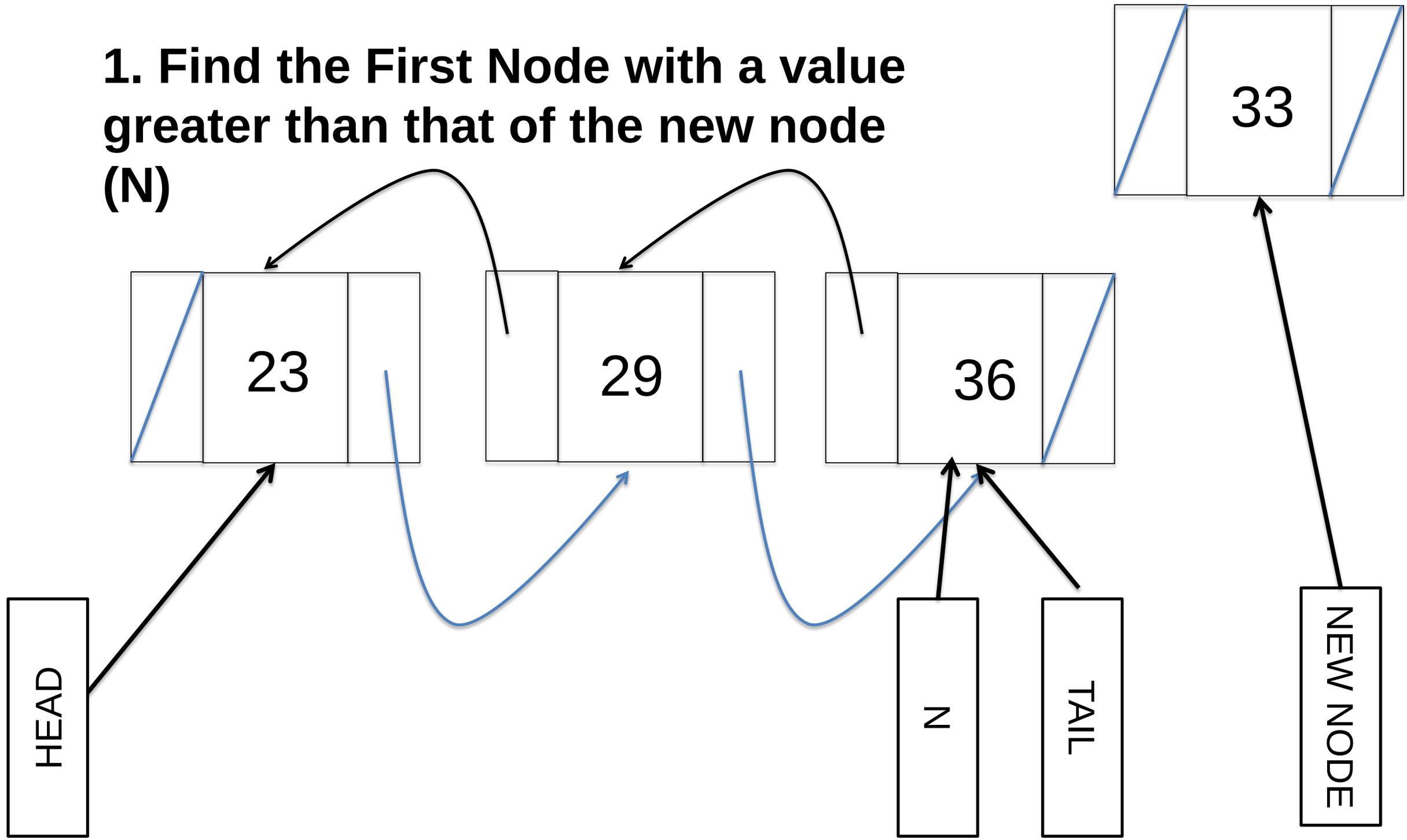
Insert In Middle

1. Create The New Node



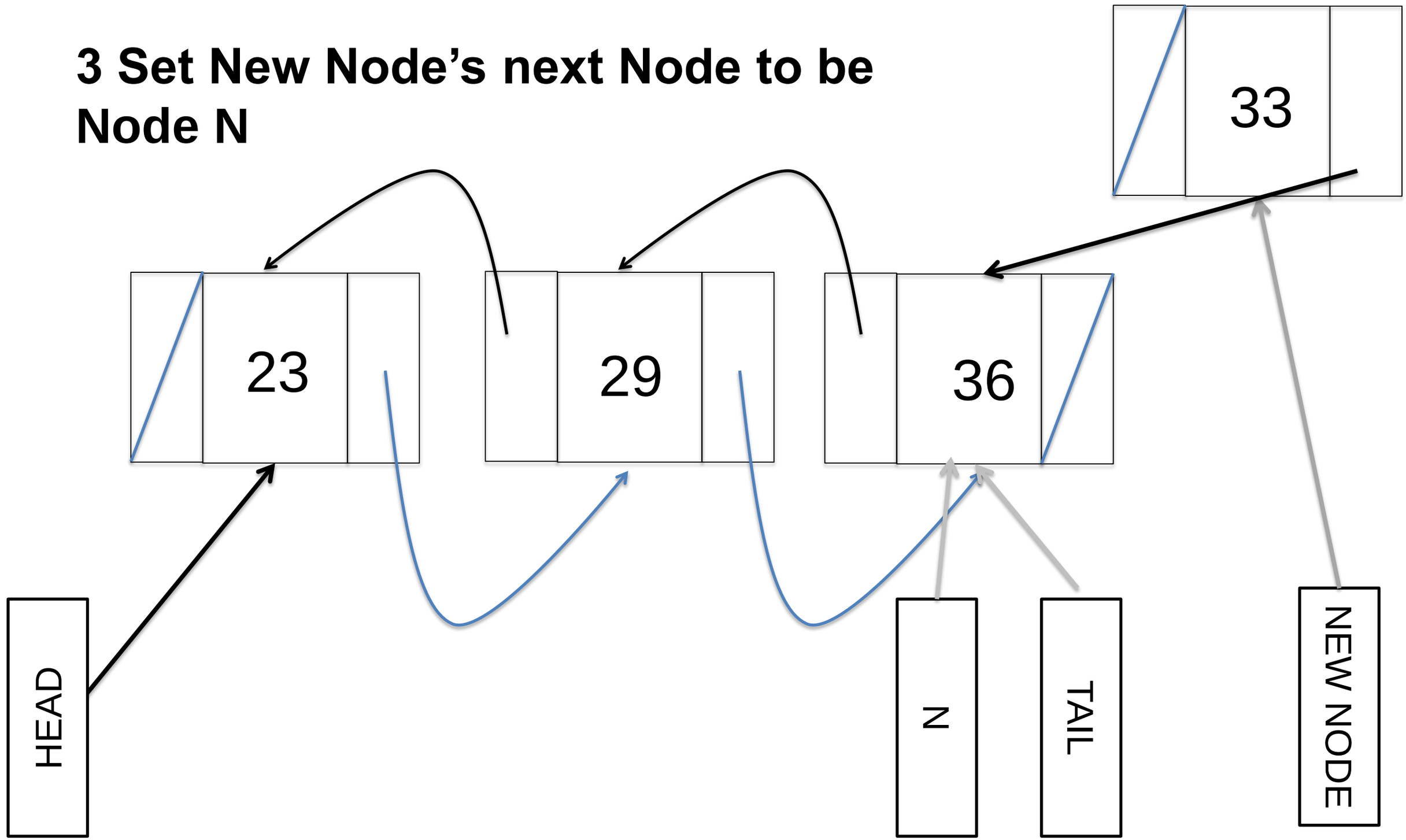
Insert In Middle

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



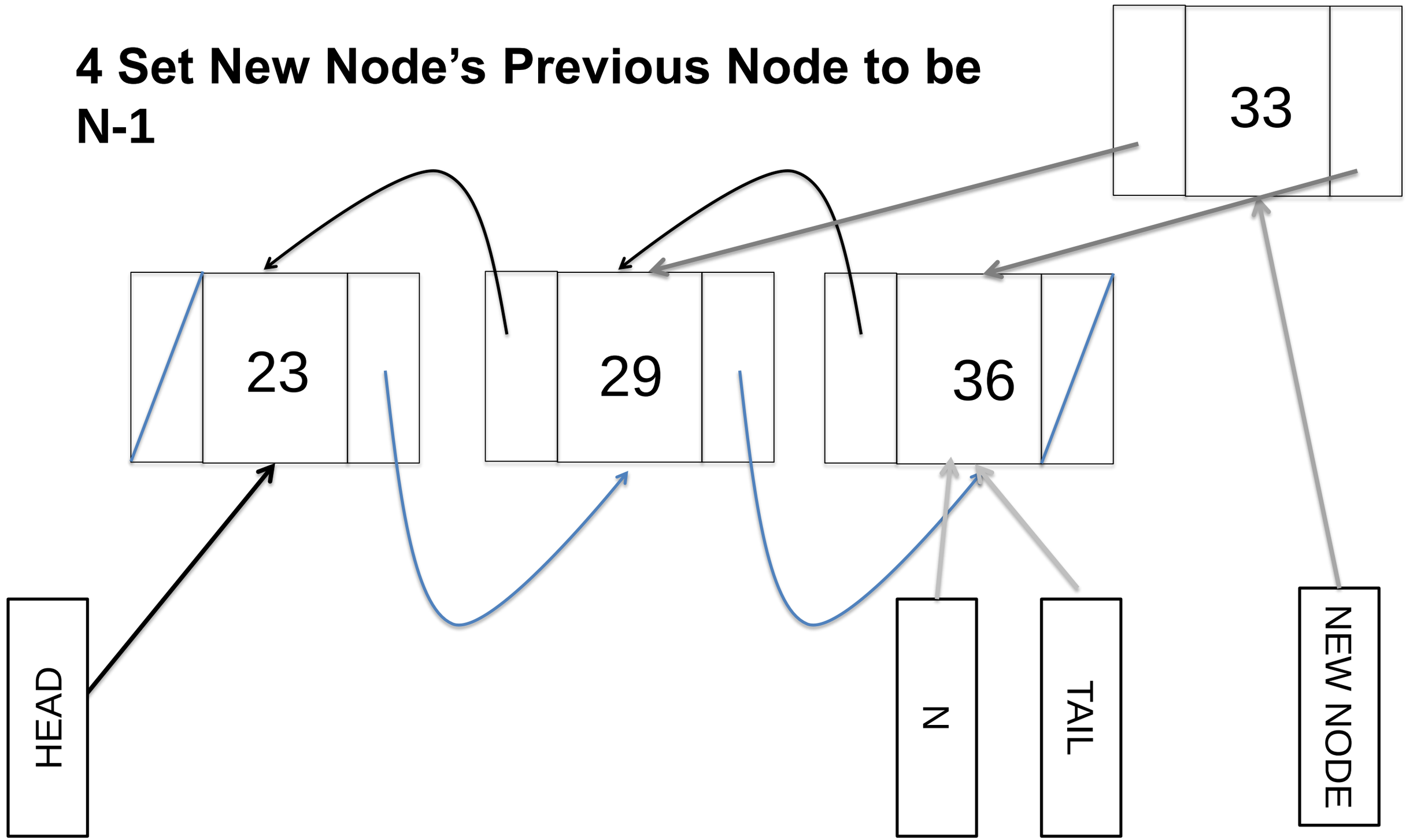
Insert In Middle

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



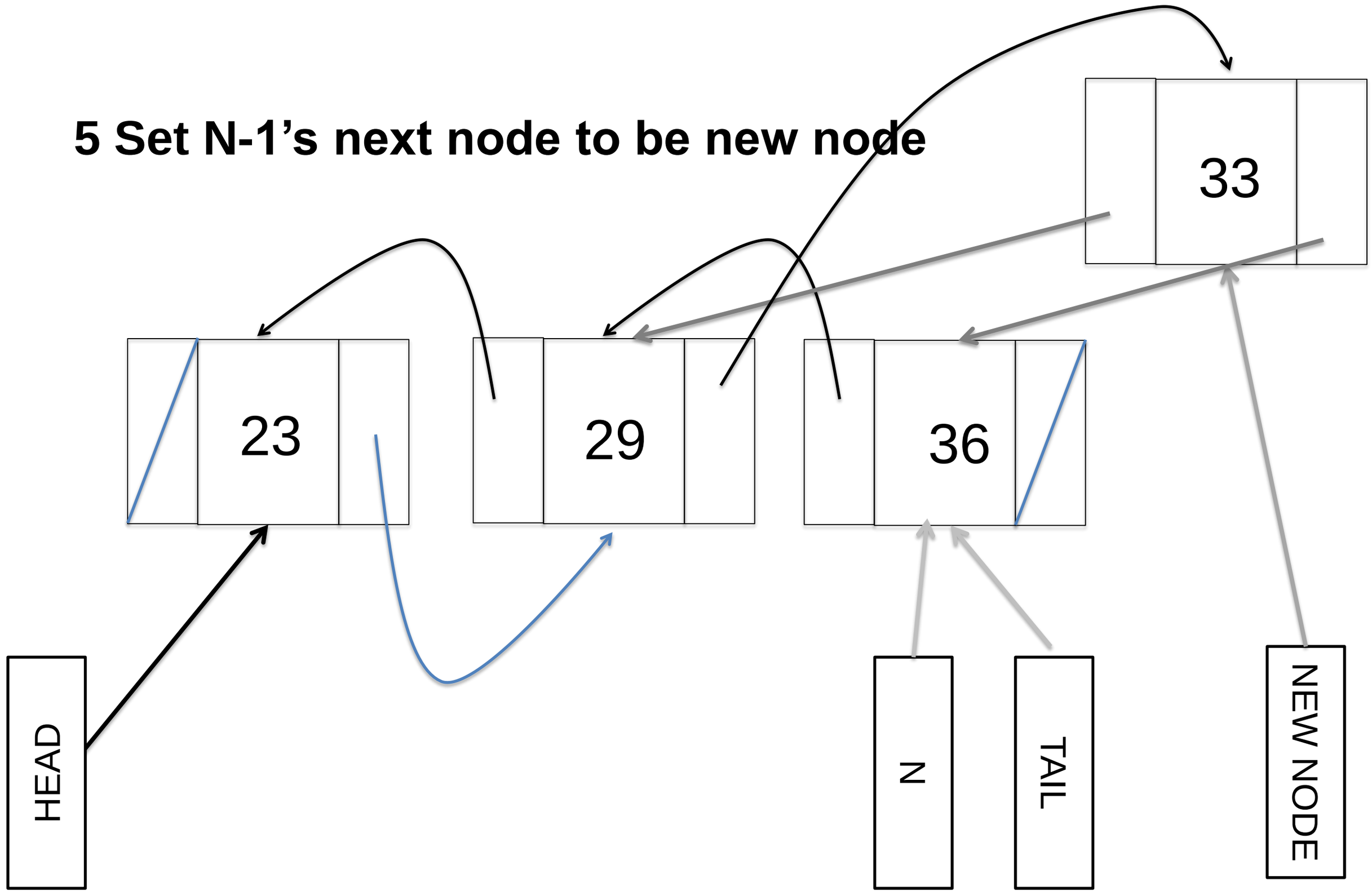
Insert In Middle

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



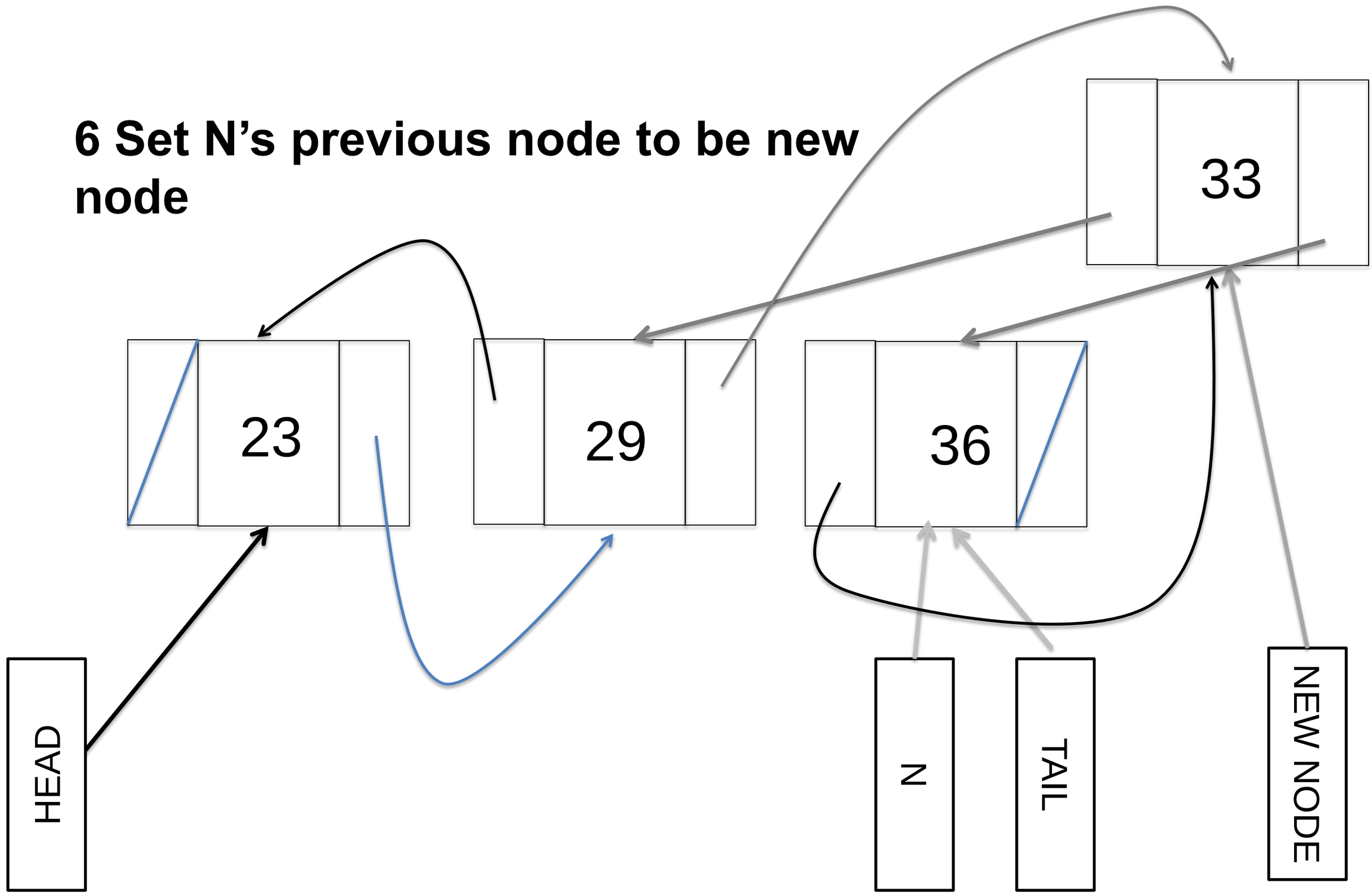
Insert In Middle

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



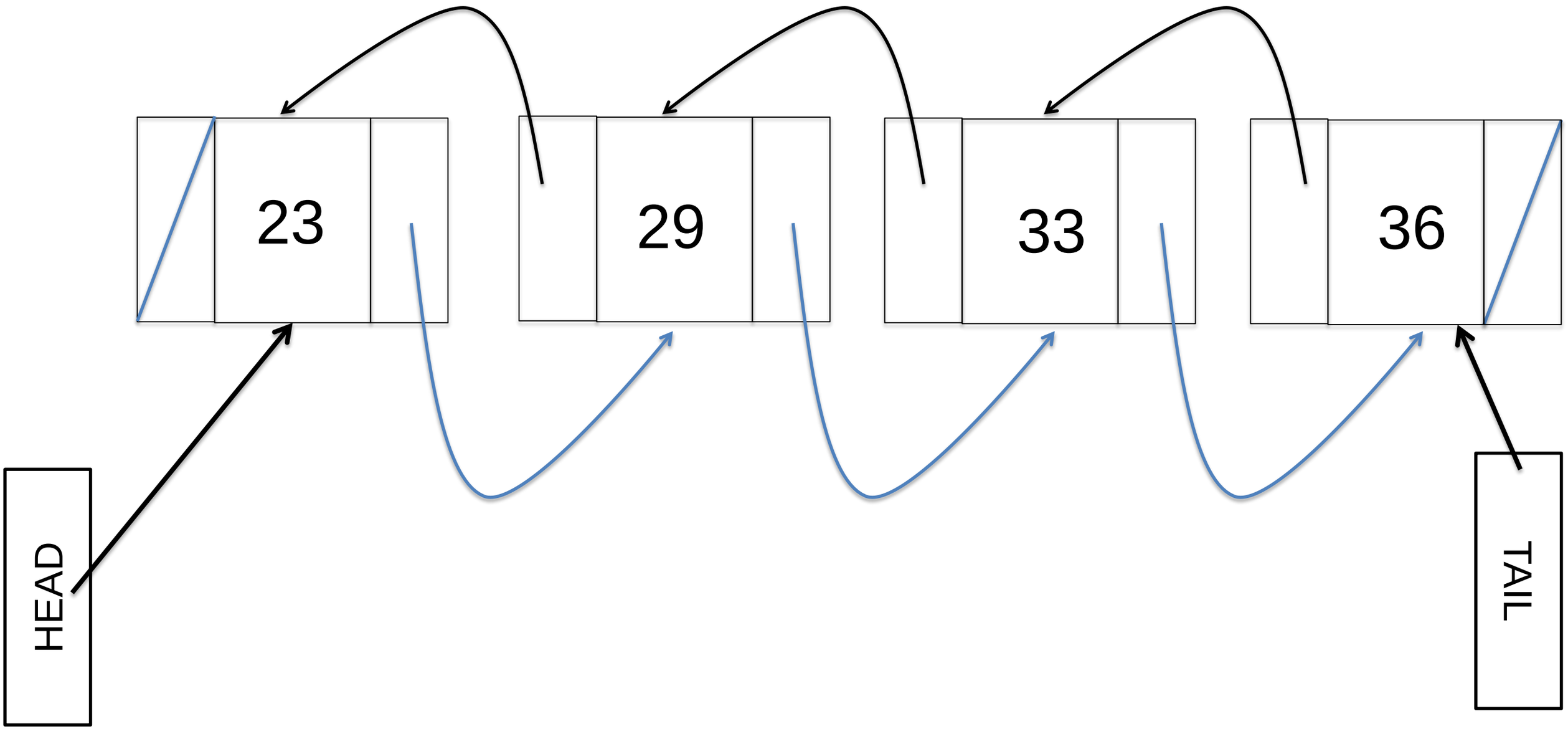
Insert In Middle

6 Set N's previous node to be new node



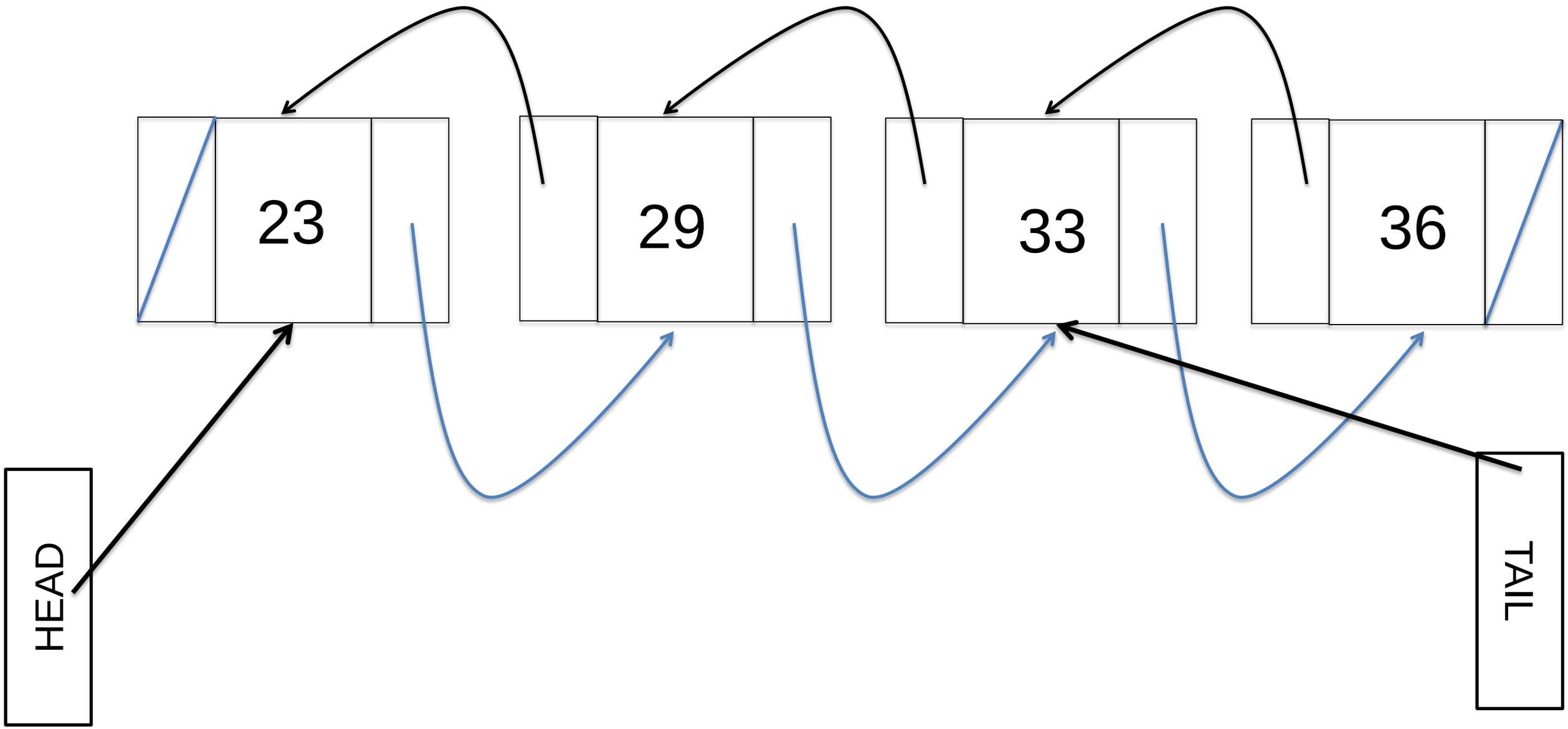
Insert In Middle

Which leads to this



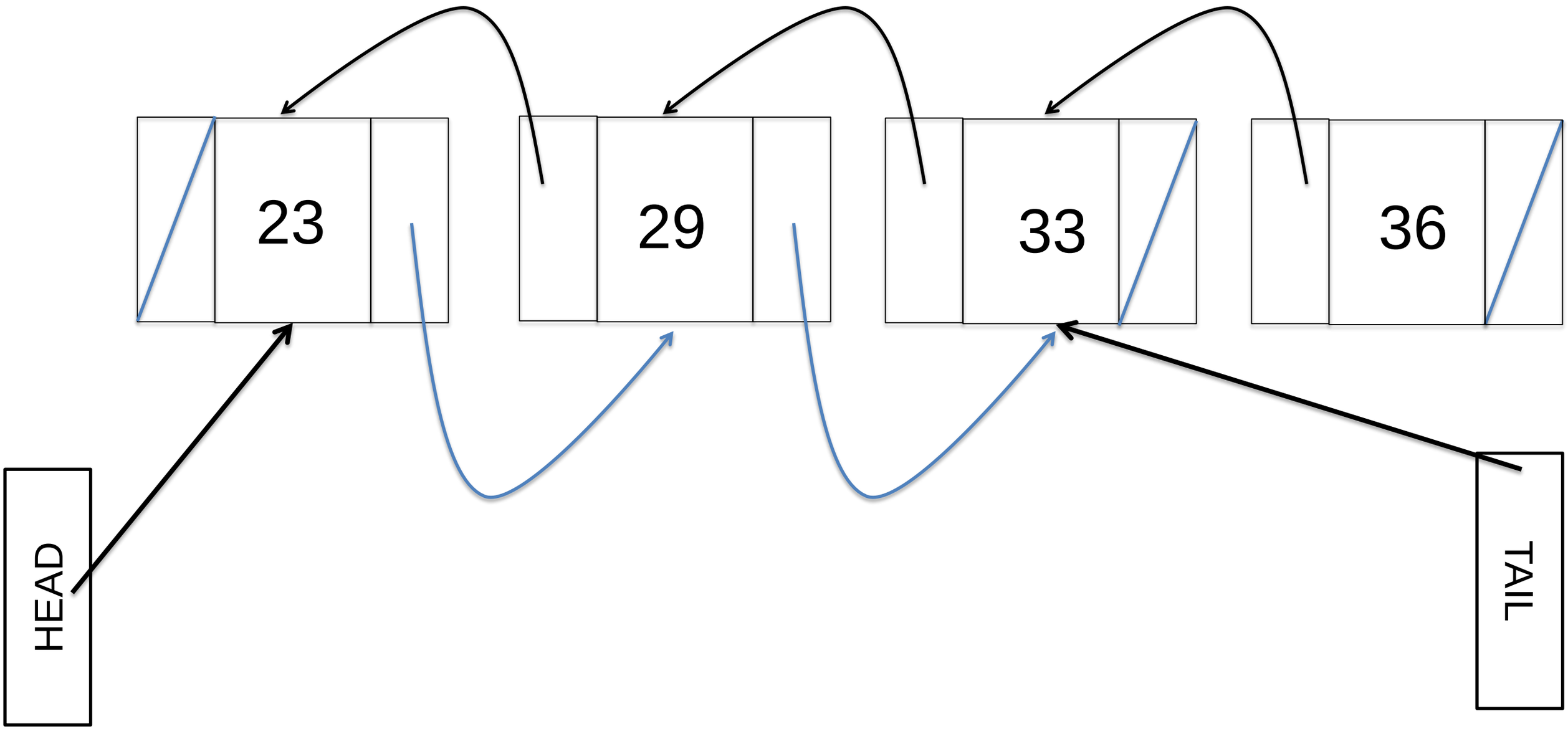
Delete at Tail

1. Set Tail to Current Tails previous Node



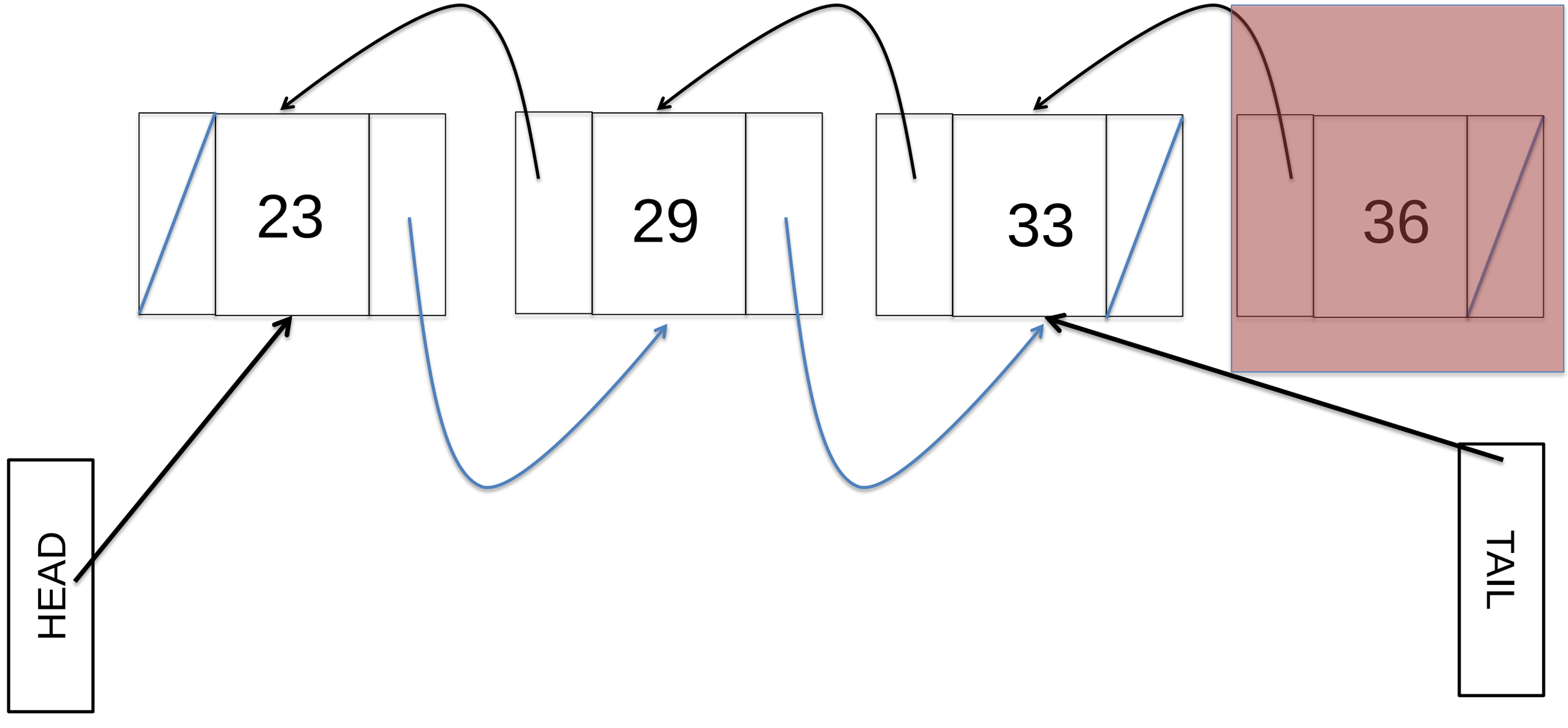
Delete at Tail

2. Set new Tail's next Node to Null



Delete at Tail

What happens to this Node?



Overview of data structures so far

boolean isEmpty()
Object top() throws StackException
Object top() throws StackException
void push()

Interfaces

Stack

Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

The Stack ADT (2)

- The data is the type of elements stored
- Main operations
  - **push**(object) - adds object to the top of the stack
  - **pop**() - remove and returns element at the top
- Auxiliary operations
  - **top**() - return element at top (also called peep())
  - **size**() - return number of elements
  - **isEmpty**() - check if empty
- Error conditions: pop/top of empty stack

The Stack ADT as a Java Interface

- We can represent the Stack ADT as a Java interface
- Elements are of type **Object** which is inherited by all other objects
- We need to create a **StackException** for error conditions

```
public interface StackI {
   public int size();
   public boolean isEmpty();
   public Object top() throws StackException;
   public void push(Object element);
   public Object pop() throws StackException;
}
```

Overview of data structures so far

boolean isEmpty()
Object top() throws StackException
Object top() throws StackException
void push()

Interfaces

Stack

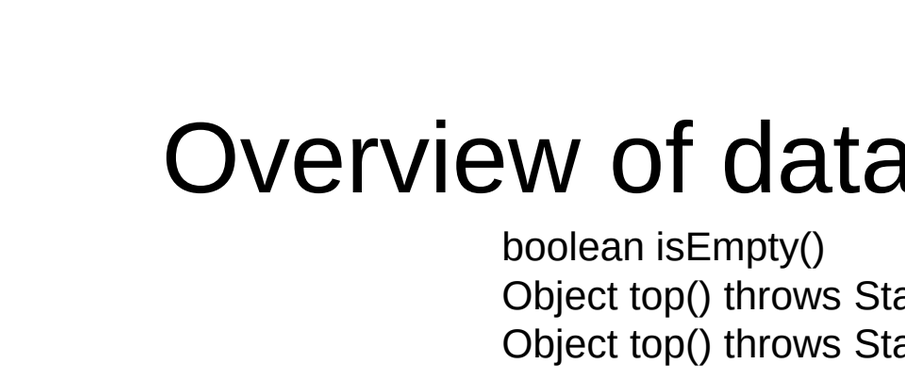
implements

Classes

StackArray

StackLinkedList

uses



Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

The Stack ADT as an Array

- Keep an array **S** of Objects, and an element **top** pointing to the top element of the stack

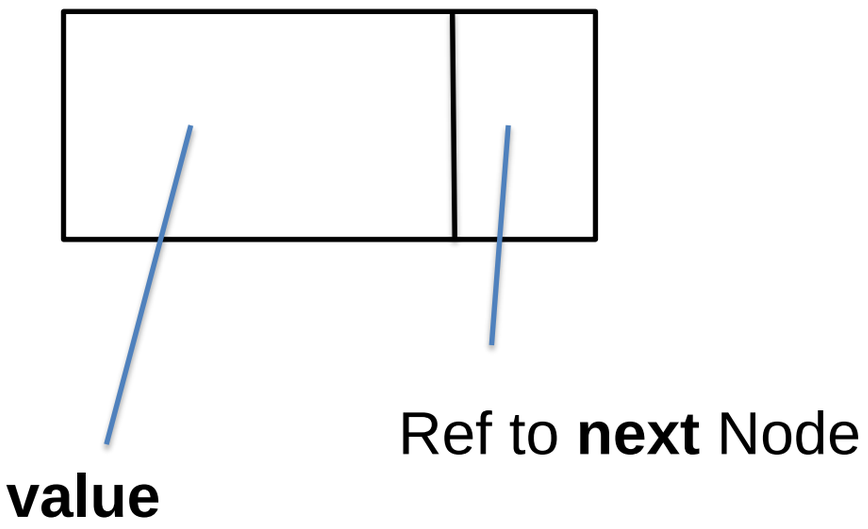

```
public class Stack implements Stack I{
   private int top;
   private int capacity;
   private Object[] S;
   private static int MAX = 100; // default size
```

Demo: Stack ADT implemented with an array

The Stack ADT as linked structure

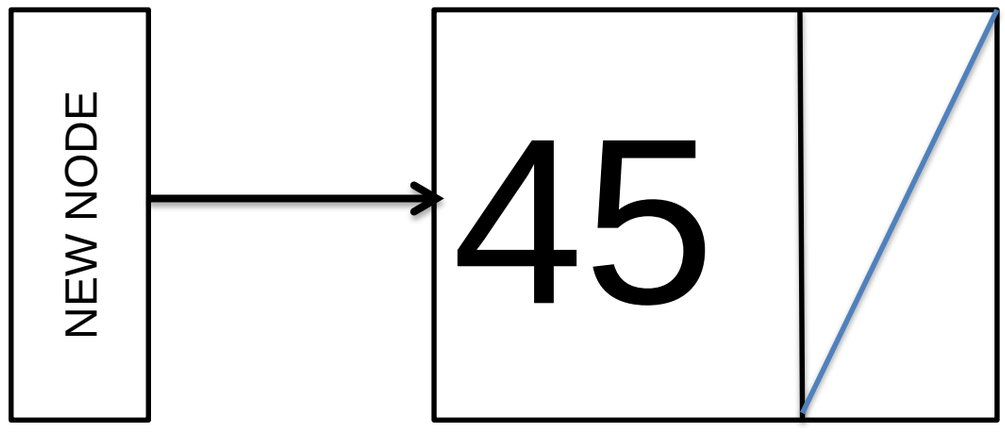
```
class Node{
  Object value;
  Node next;

  public Node(Object e, Node n){
      element = e;
      next = n;
   }
   }
}
```

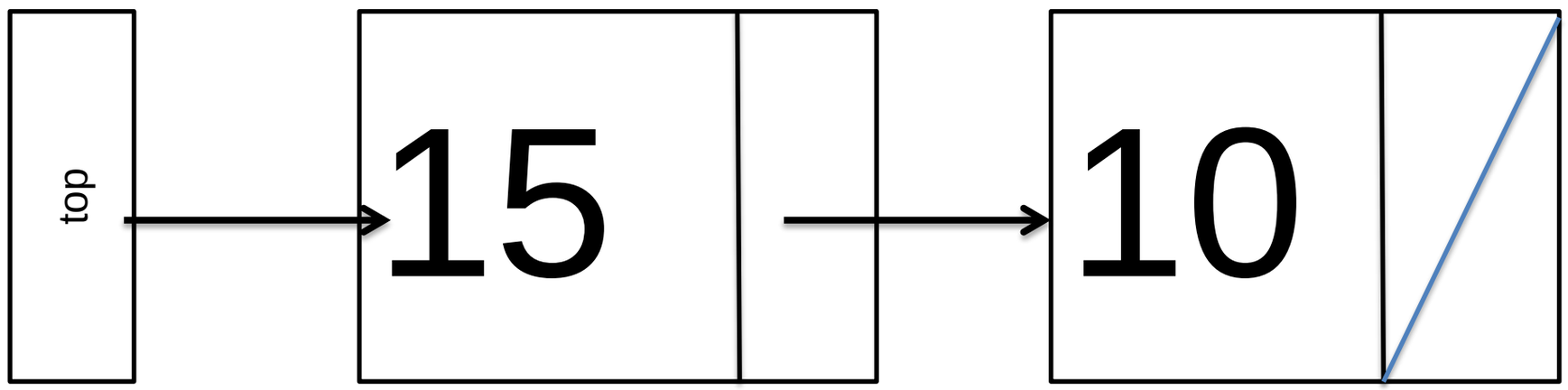


- The stack has to contain a **node**
  - Contains the data
  - We will call this **top**
  - This should initially be a **null** pointer
- This will be incremented/decremented by push and pop

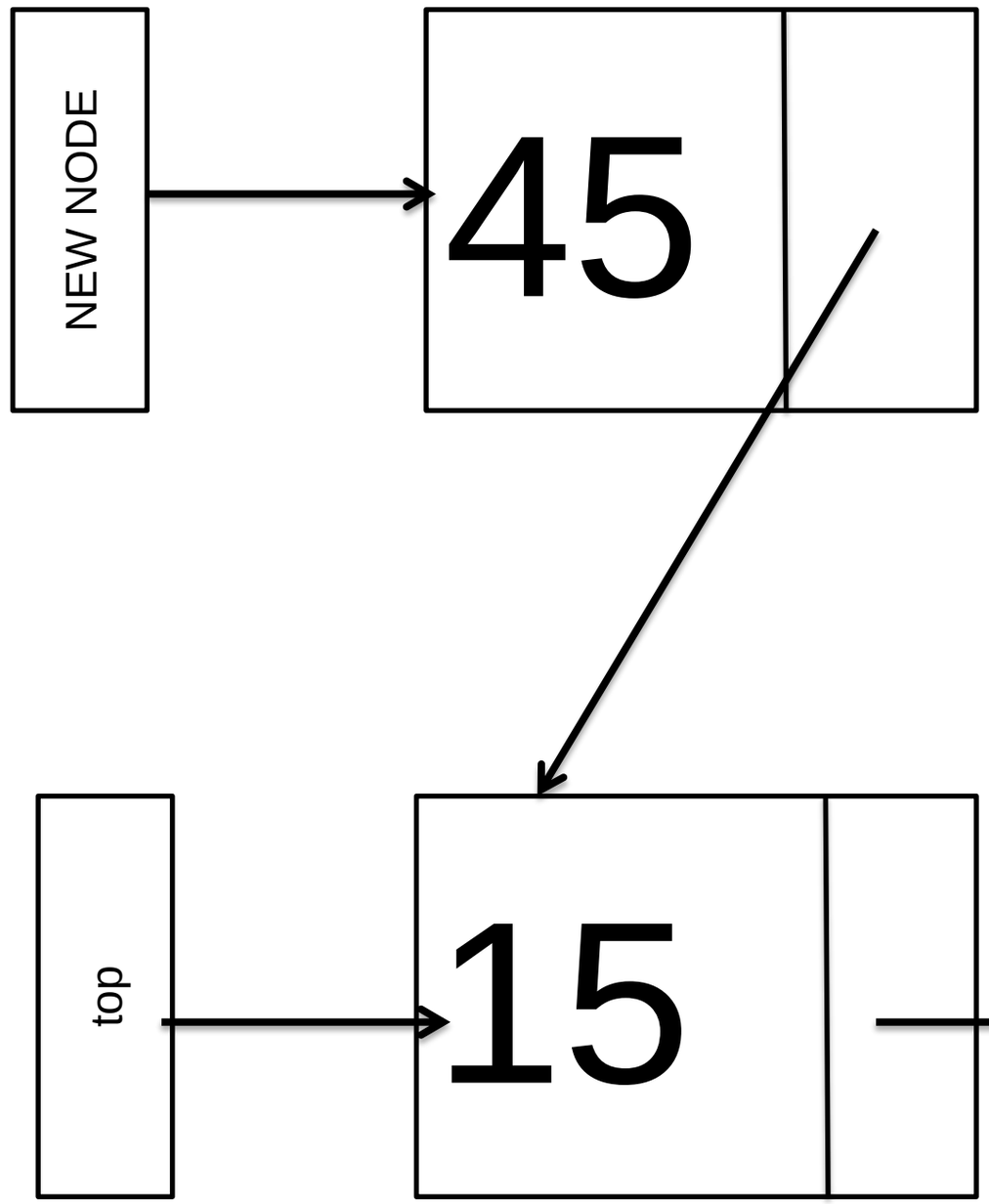
Pushing an element



1. Create the new node

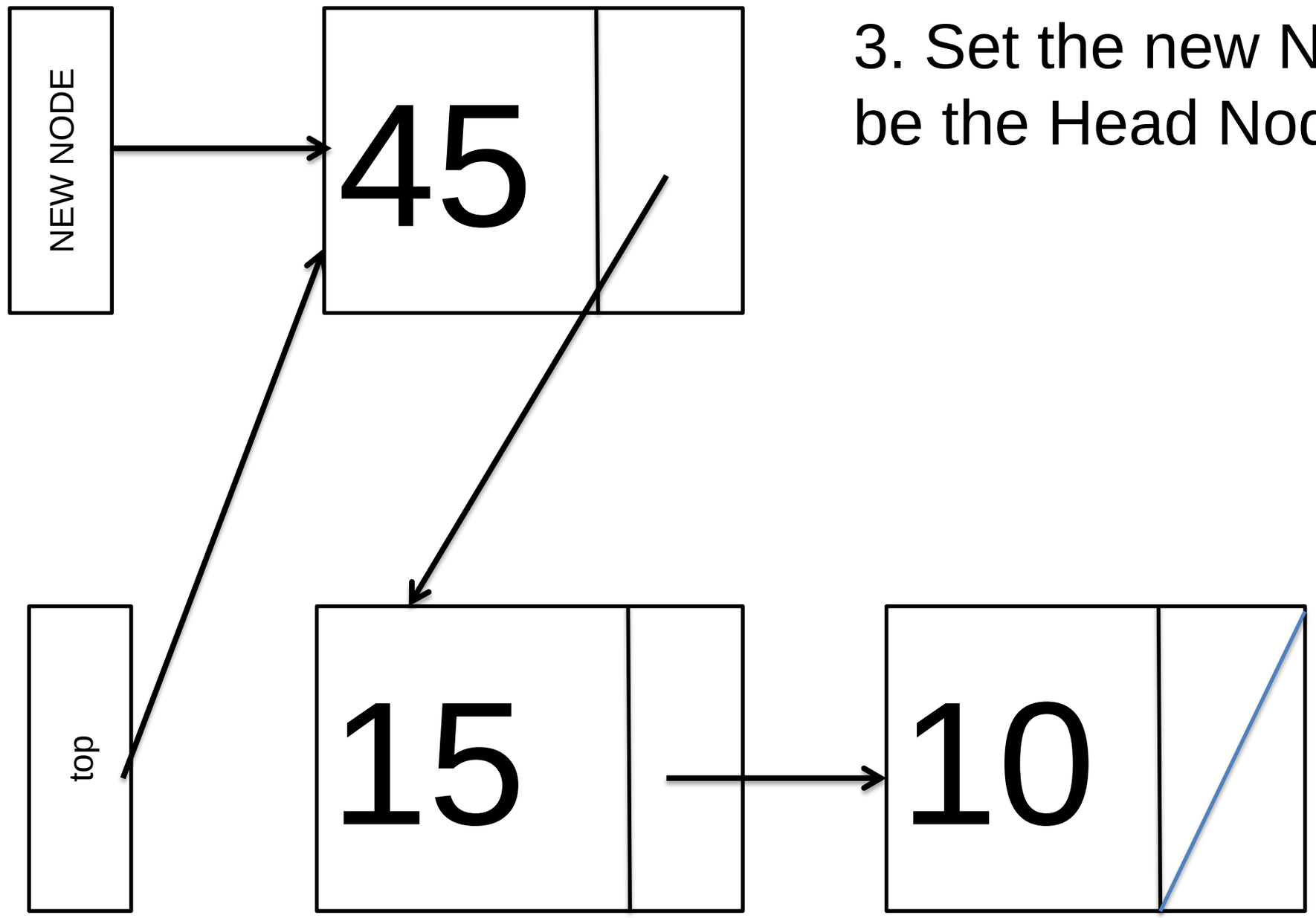


Pushing an element



2. Set New node Reference to Existing Head Node

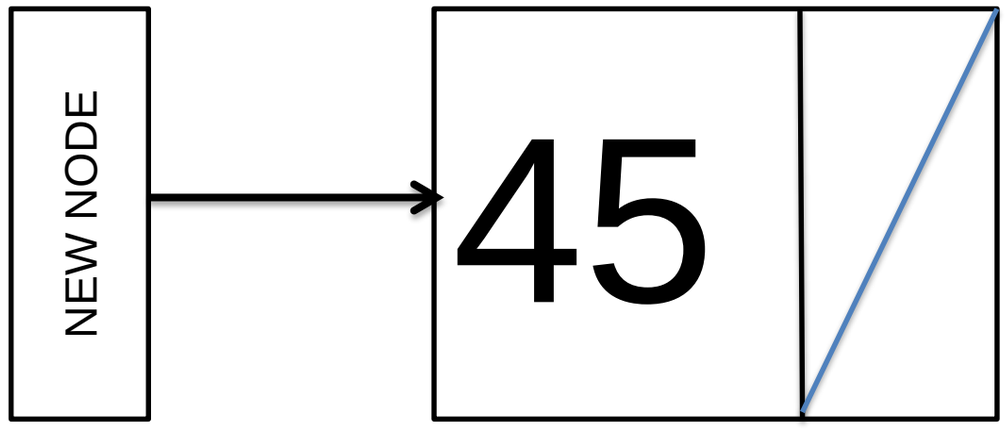
Pushing an element



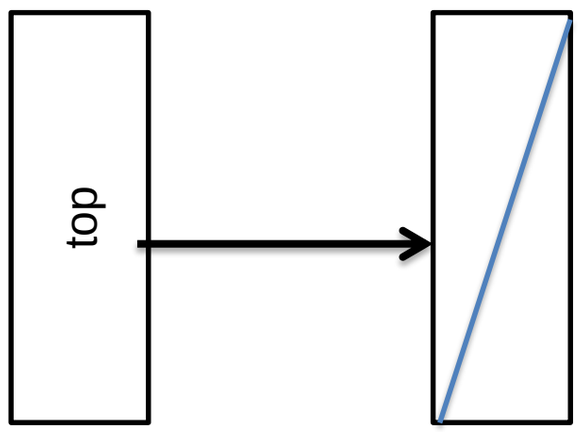
3. Set the new Node to be the Head Node

What if the stack is Empty?

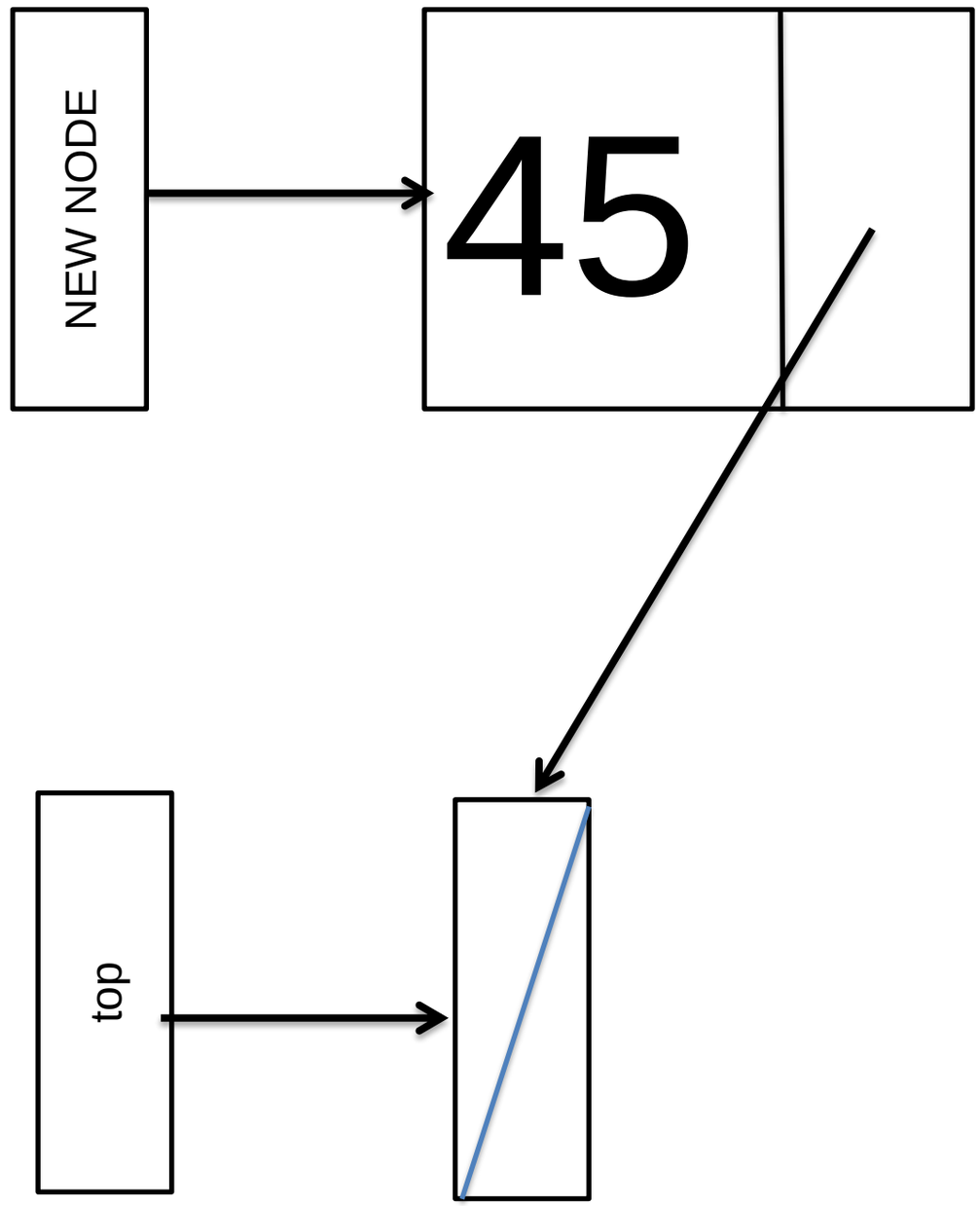
Pushing an element (Empty stack)



1. Create the new node

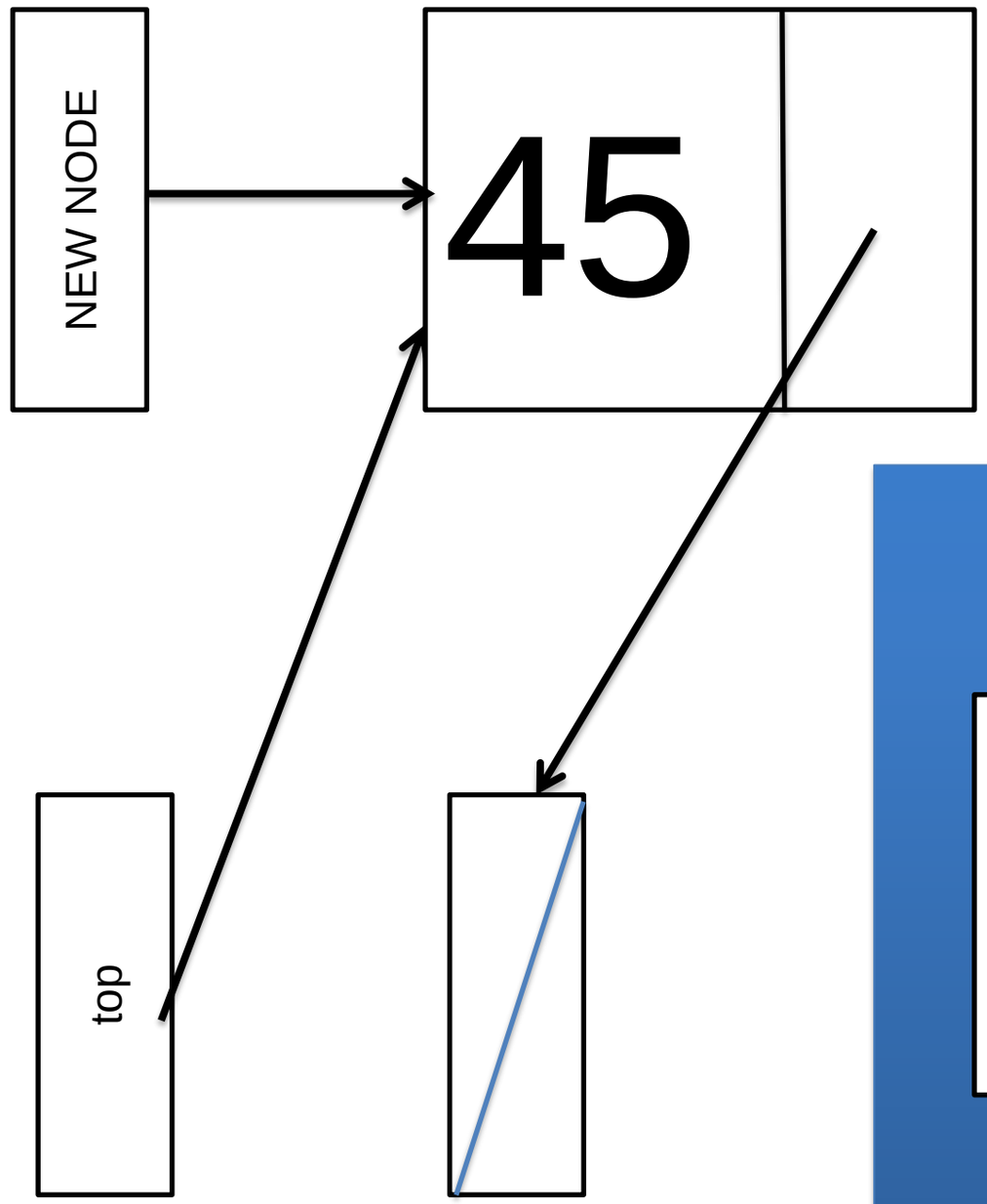


Pushing an element



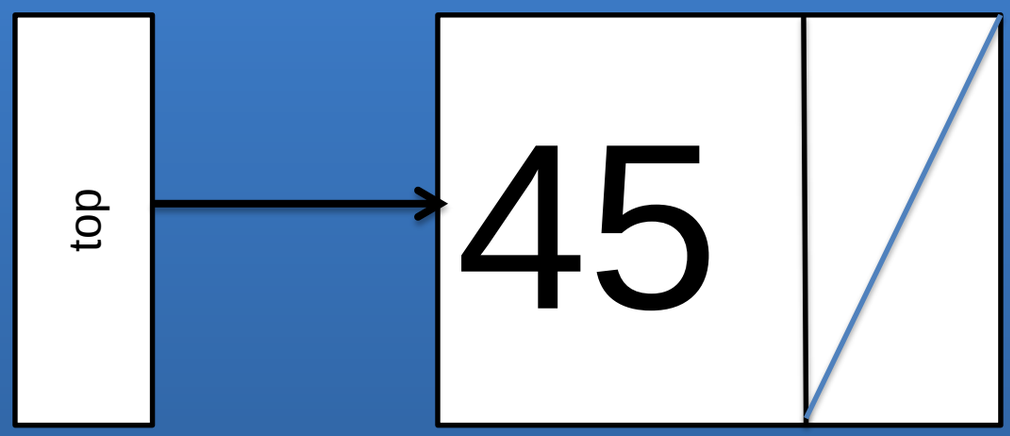
2. Set New node Reference to Existing Head Node

Pushing an element



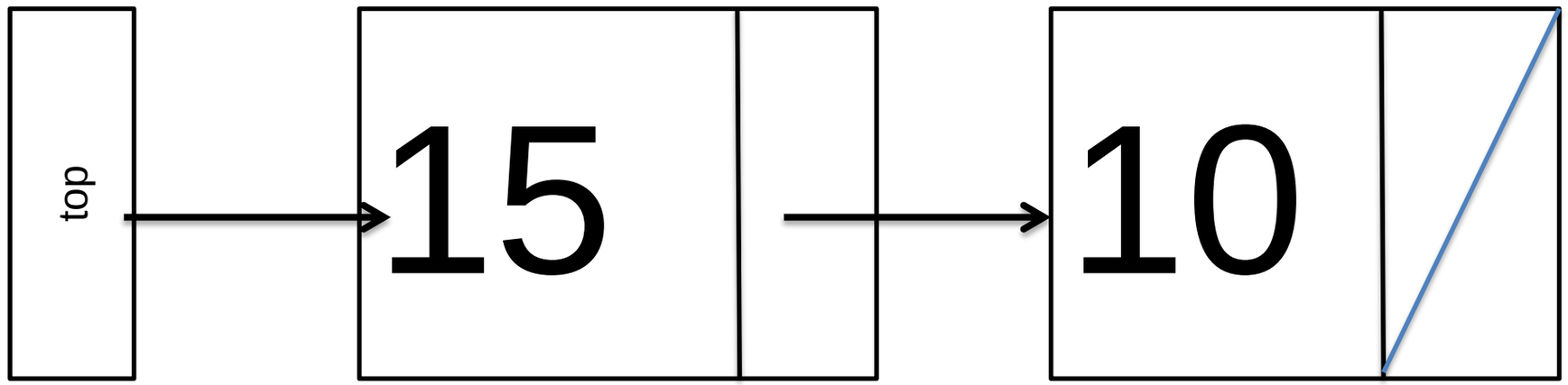
3. Set the new Node to be the Head Node

Which is the same as:



Popping

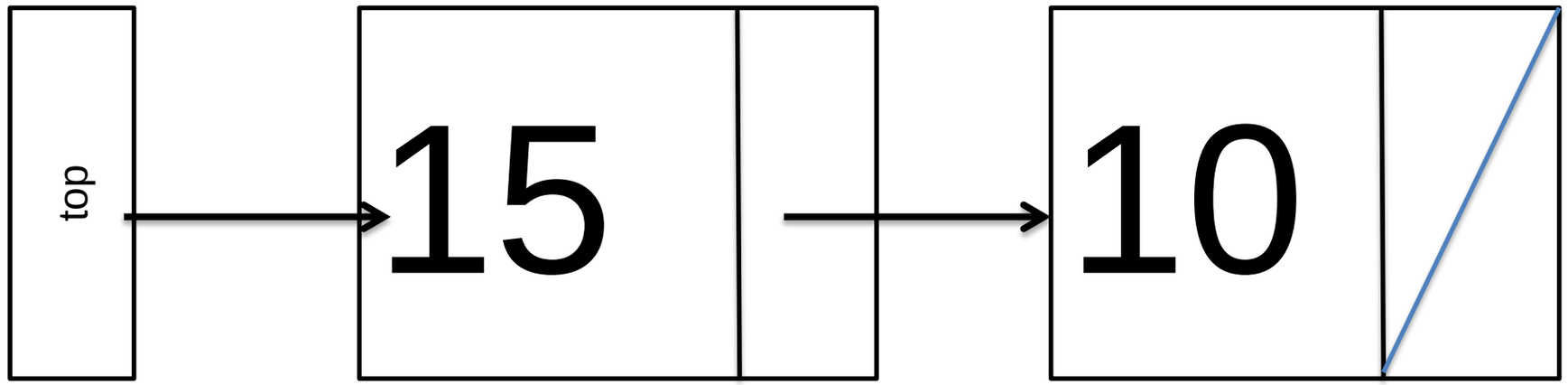
1. Check the List isn’t Null



HEAD == NULL throws a **StackException**

Popping

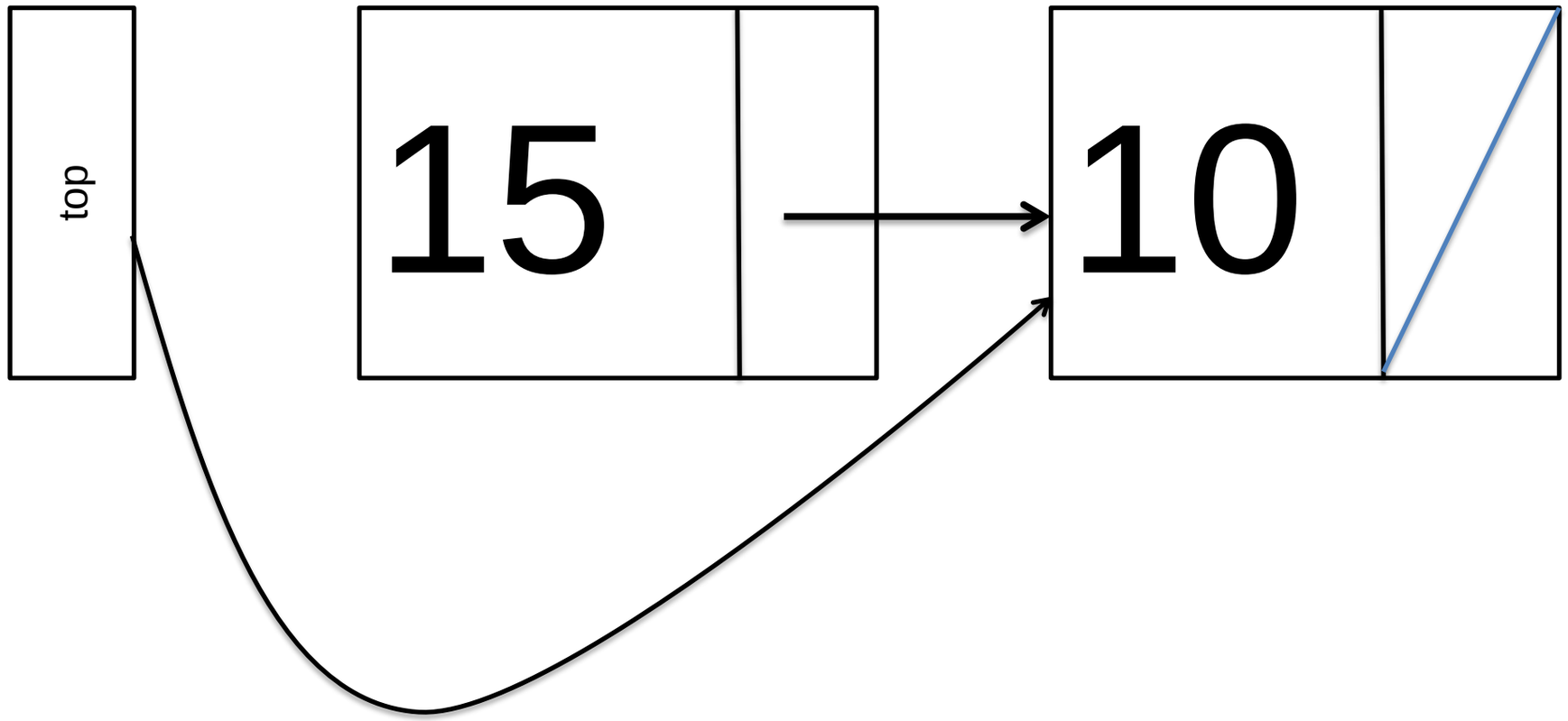
2. Get the first element



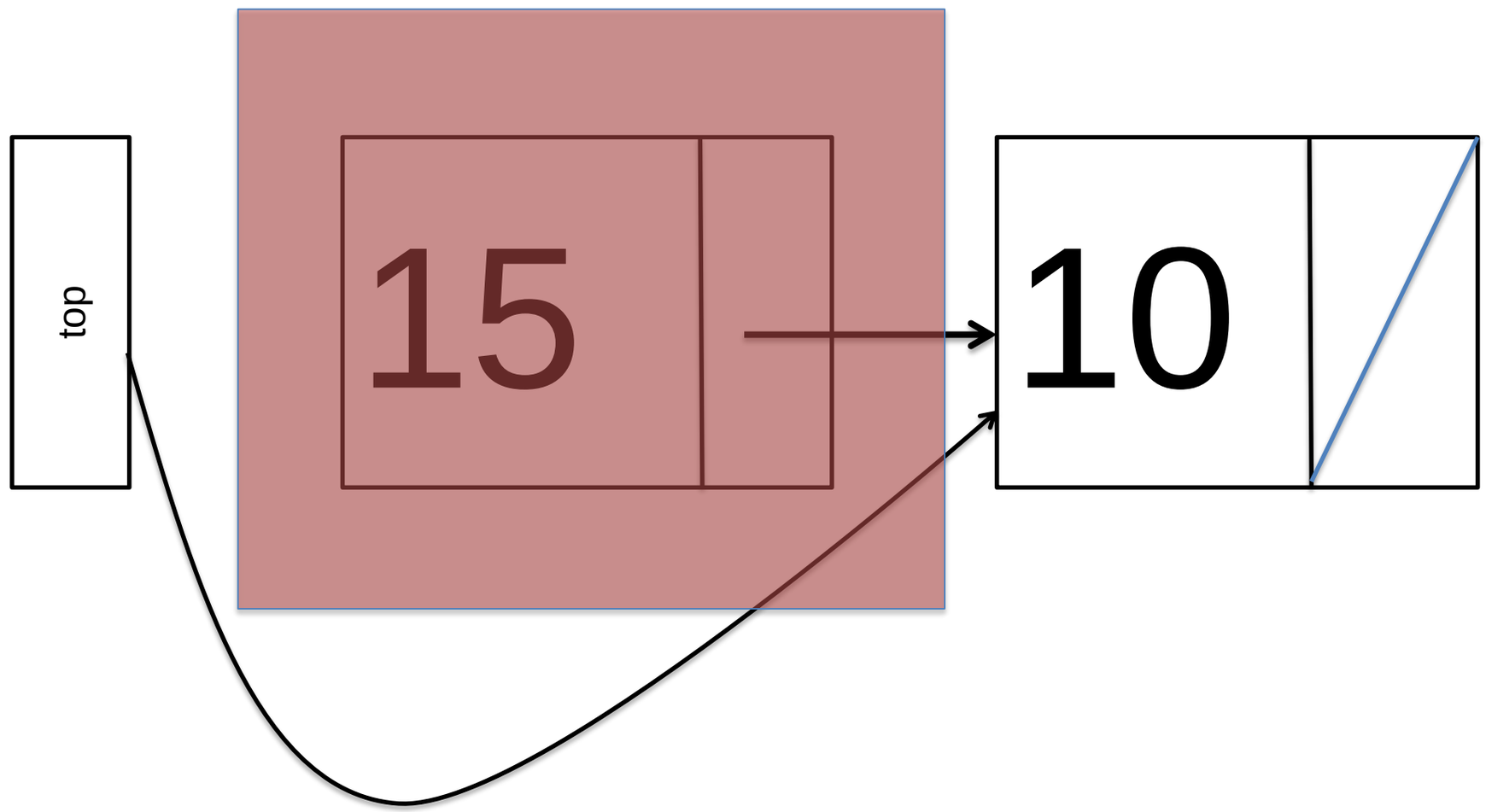
tmp = 15;

Popping

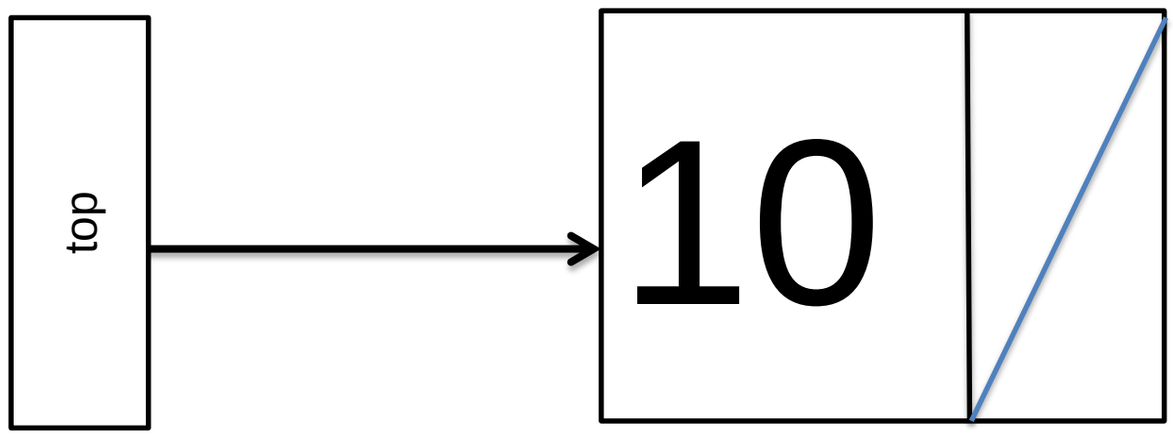
3. Set top to the next element of top



Popping



Popping



4. Return the tmp element (15 in this case)

The **size** operation

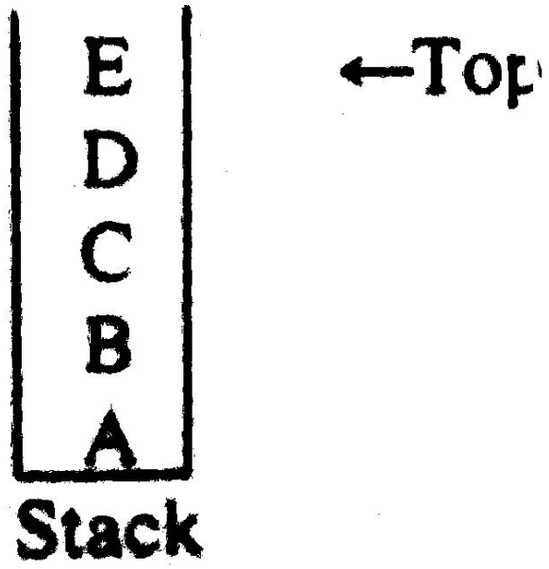
- Can we do this in **O(1)?**
  - Not using current representation
  - Need to iterate the full lists ~ O(N)
- Solution
  - We add a size field to our stack
  - Should be incremented by a push
  - Should be decrement by a pop

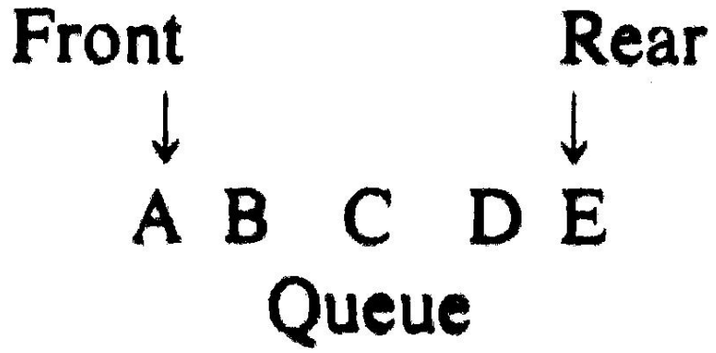
We will benchmark both versions later:

- Naïve: follow *nextNode* object references, keeping count.
- Optimised: add and maintain a *size* field.

Demo: Stack ADT implemented with a linked list

Stacks versus Queues





Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

Overview of data structures so far

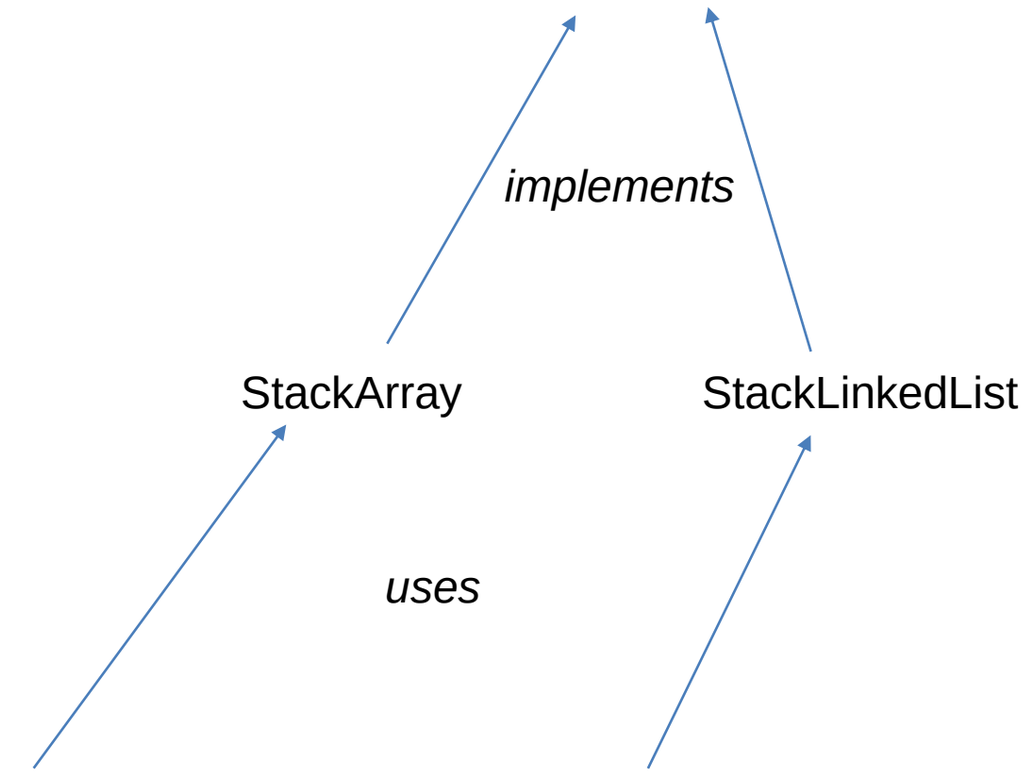
boolean isEmpty()
Object top() throws StackException
Object pop() throws StackException
void push(Object element)

void enqueue(Object element)
Object dequeue() throws QueueException
Object front() throws QueueException
int size()
boolean isEmpty()

Interfaces

Stack

Queue



Classes

StackArray

StackLinkedList

Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

Overview of data structures so far

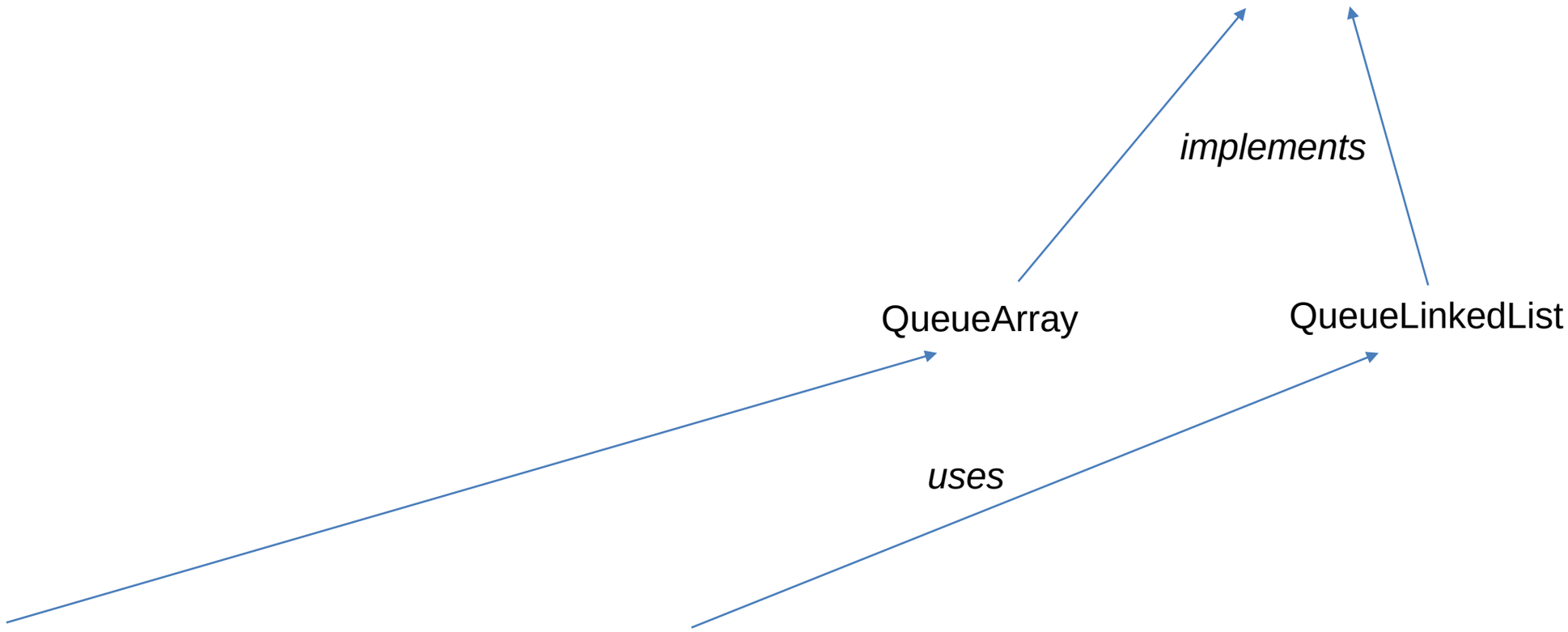
Interfaces

boolean isEmpty()
Object top() throws StackException
Object pop() throws StackException
void push(Object element)

Stack

void enqueue(Object element)
Object dequeue() throws QueueException
Object front() throws QueueException
int size()
boolean isEmpty()

Queue



Classes

Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

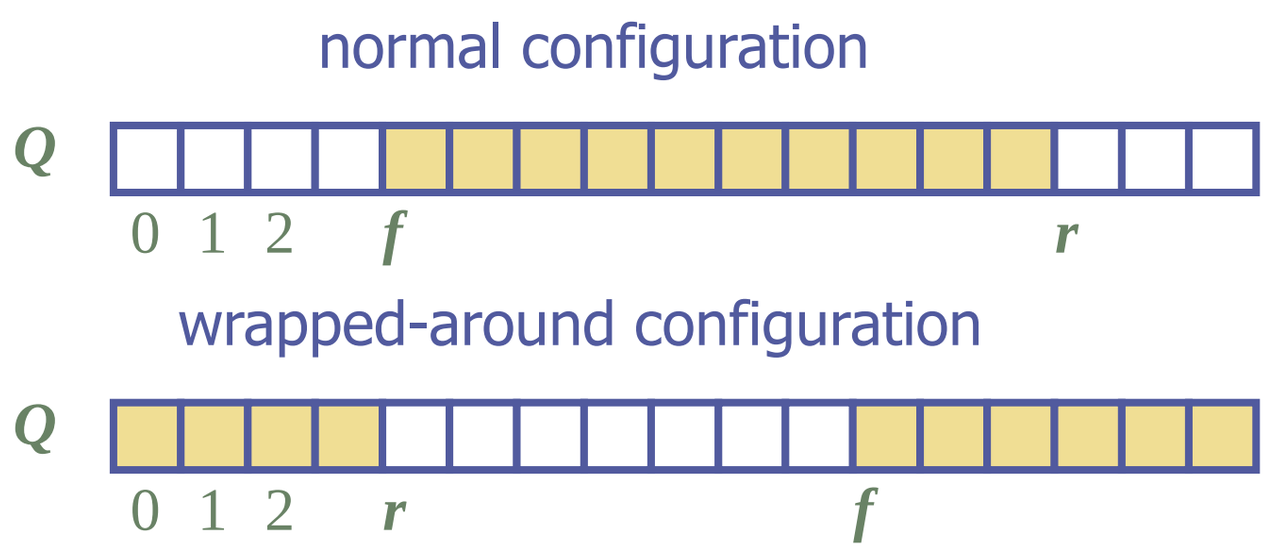
Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

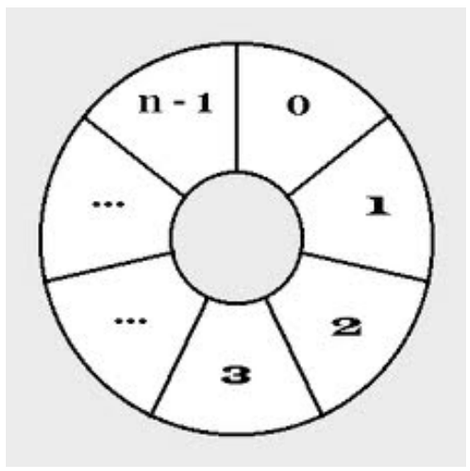
A Queue Example

Operation	***Output***	*Q: (front,...,rear)*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*“error”*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

Queue Implementation "Wrap Around"

- What happens when **r** reaches the end of the array
  - … and there may still be free elements before **f**?
- A solution is to "wrap" **r** around to the front of the list

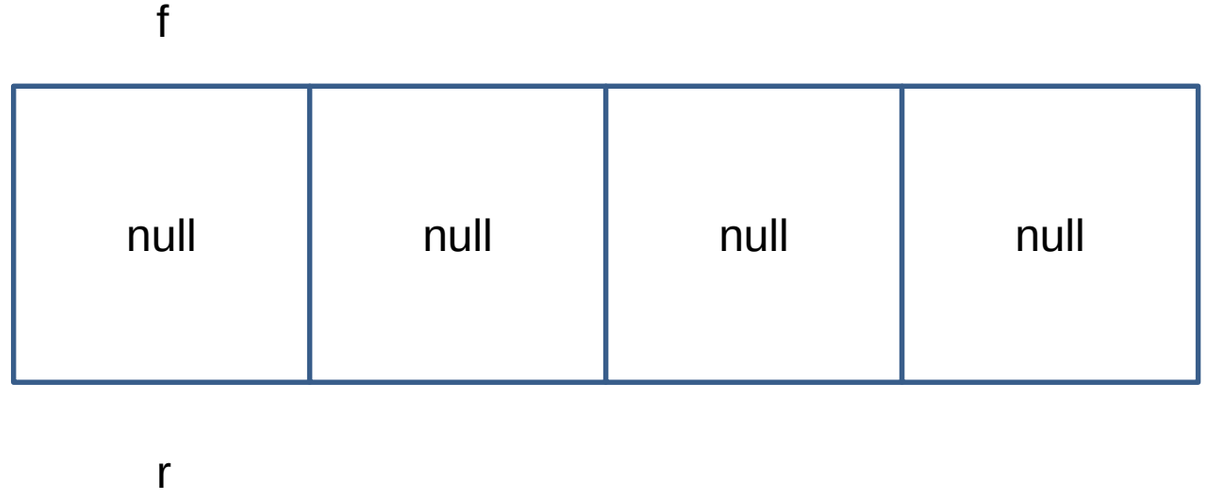




- This can be seen as a circular list
  - where the last element is followed by the first

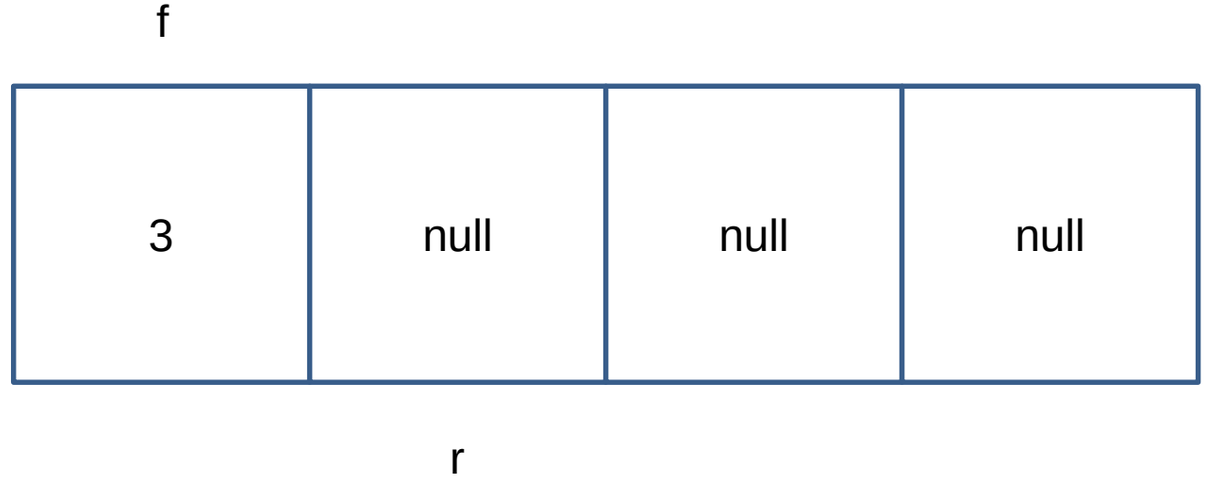
Implementing Queues using Arrays

Queue q = new Queue(3)



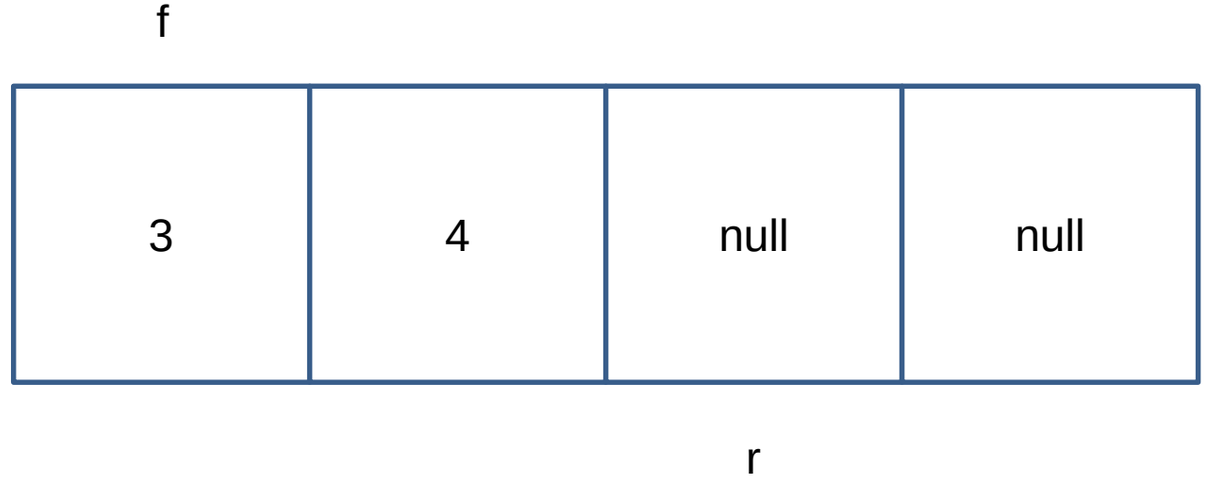
Implementing Queues using Arrays

q.enqueue(3)



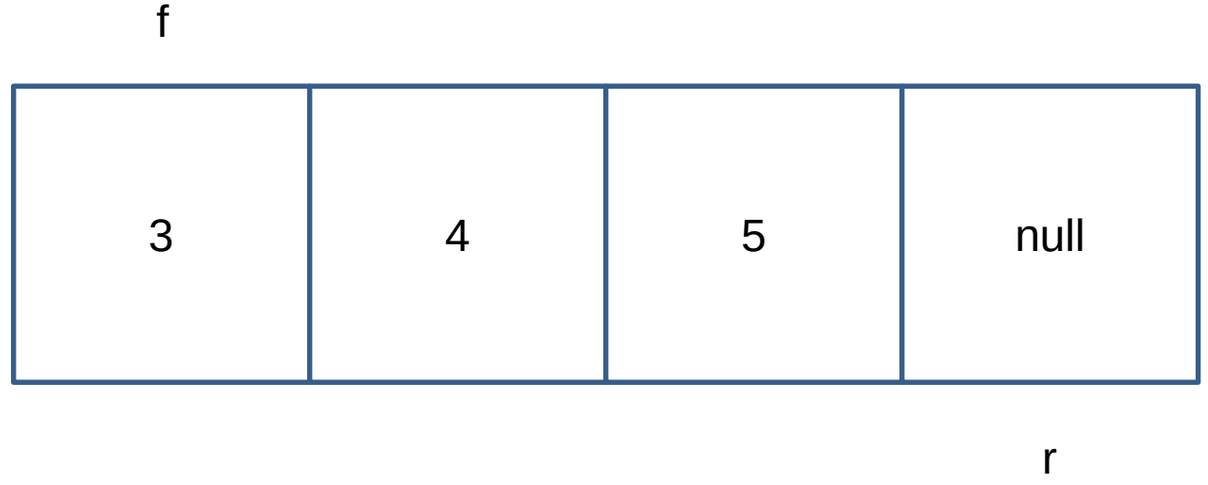
Implementing Queues using Arrays

q.enqueue(4)



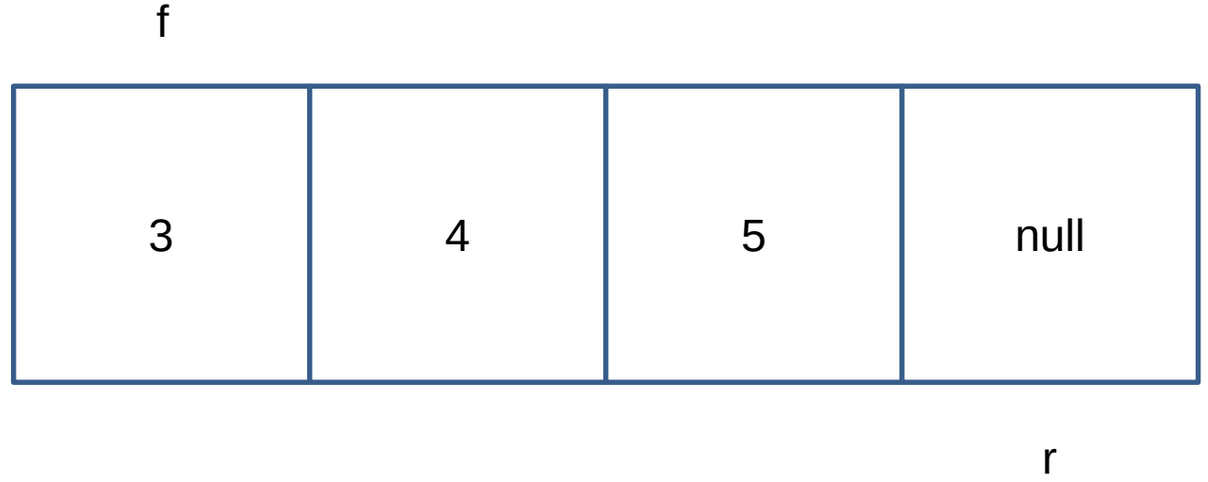
Implementing Queues using Arrays

q.enqueue(5)



Implementing Queues using Arrays

q.enqueue(6)




```
public void enqueue(Object e) throws QueueException {
 if (size() == capacity - 1)
     throw new QueueException("Queue is full.");
  ...
```

Implementing Queues using Arrays

q.dequeue()

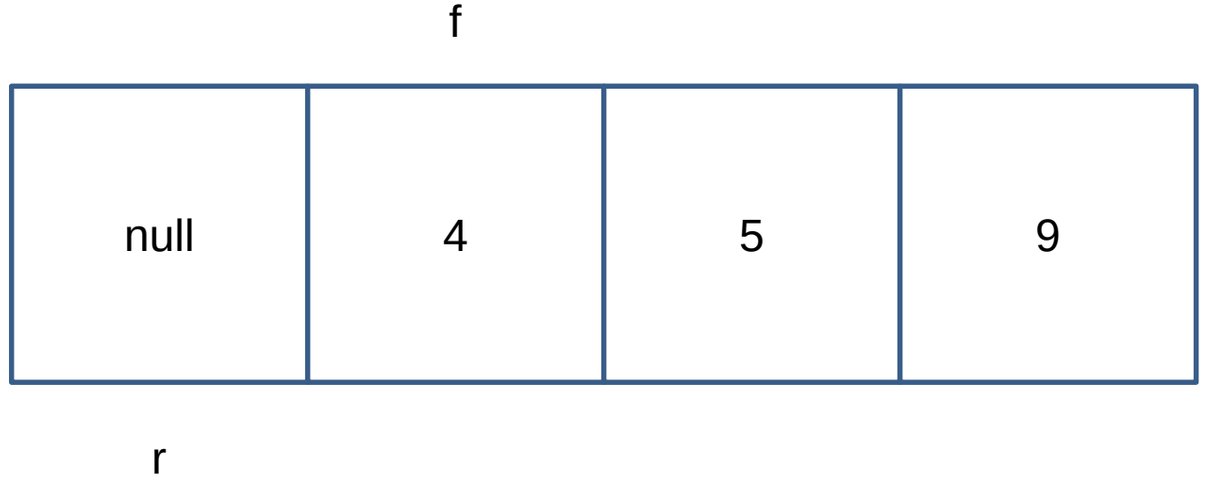
f

null	4	5	null

r

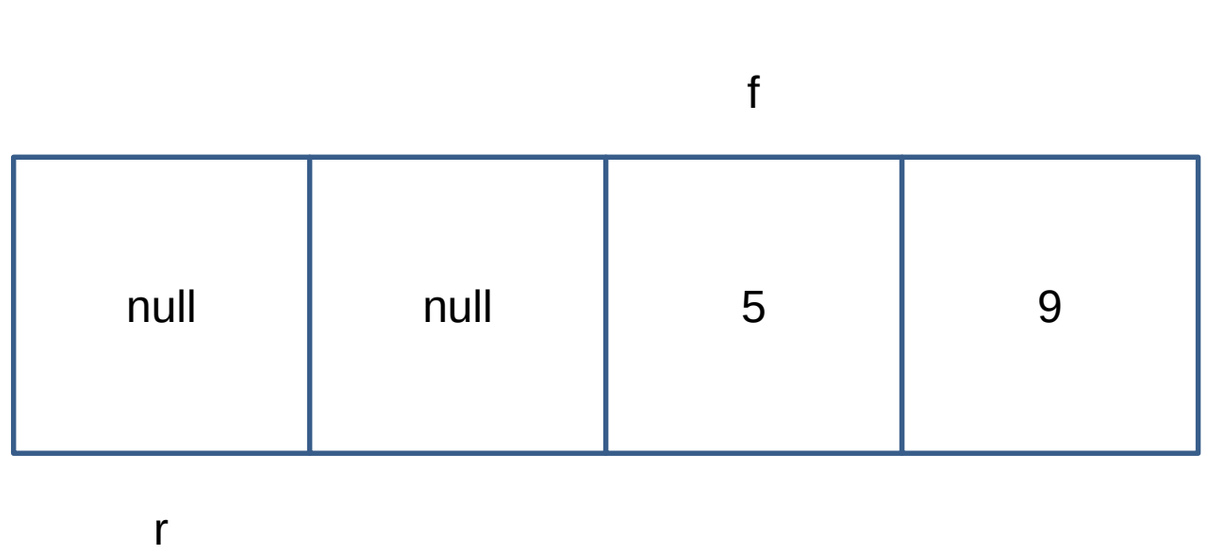
Implementing Queues using Arrays

q.enqueue(9)



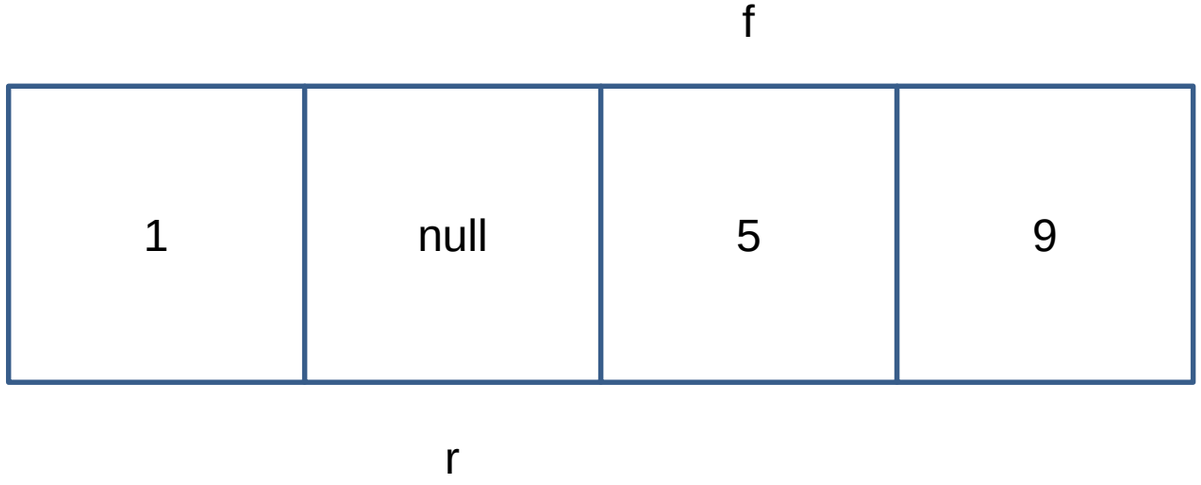
Implementing Queues using Arrays

q.dequeue()



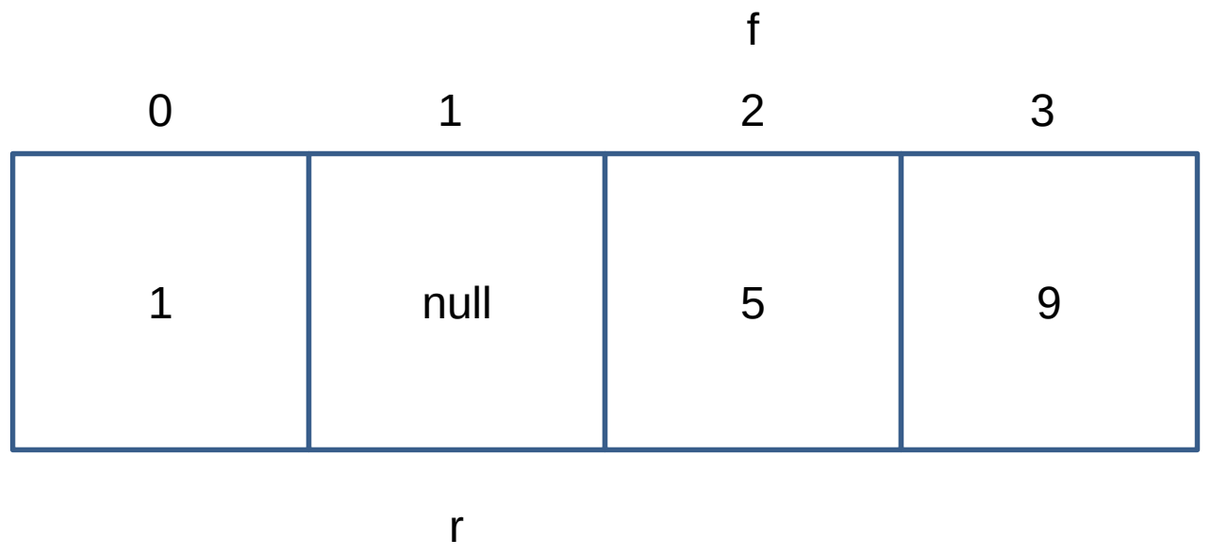
Implementing Queues using Arrays

q.enqueue(1)

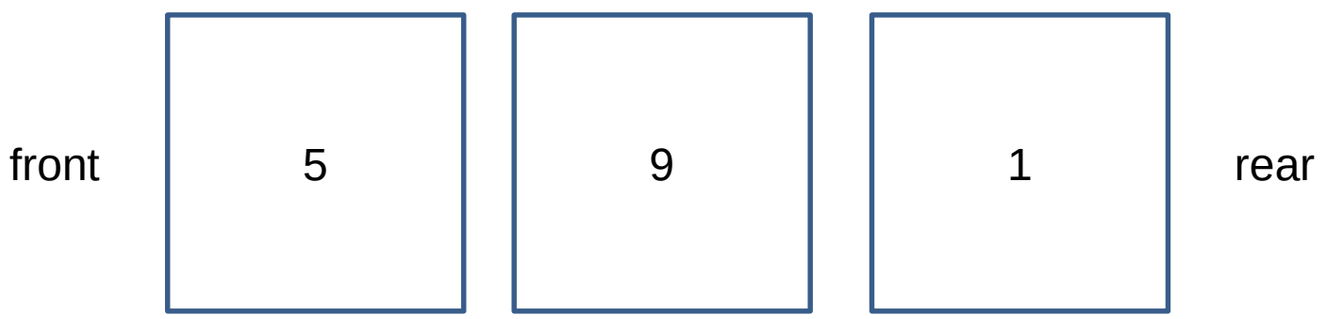


Implementing Queues using Arrays

Array implementation



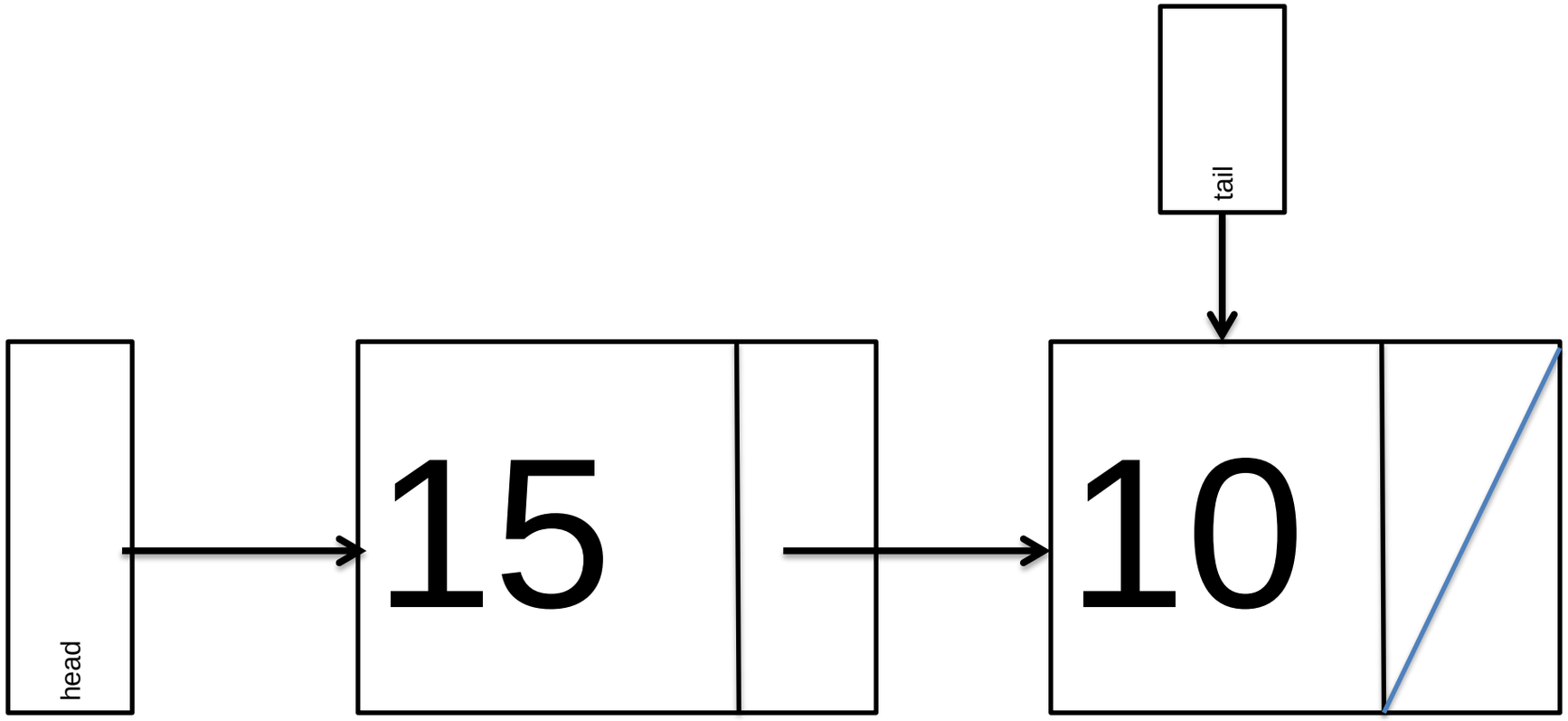
Queue ADT



Demo: Queue ADT implemented with an array

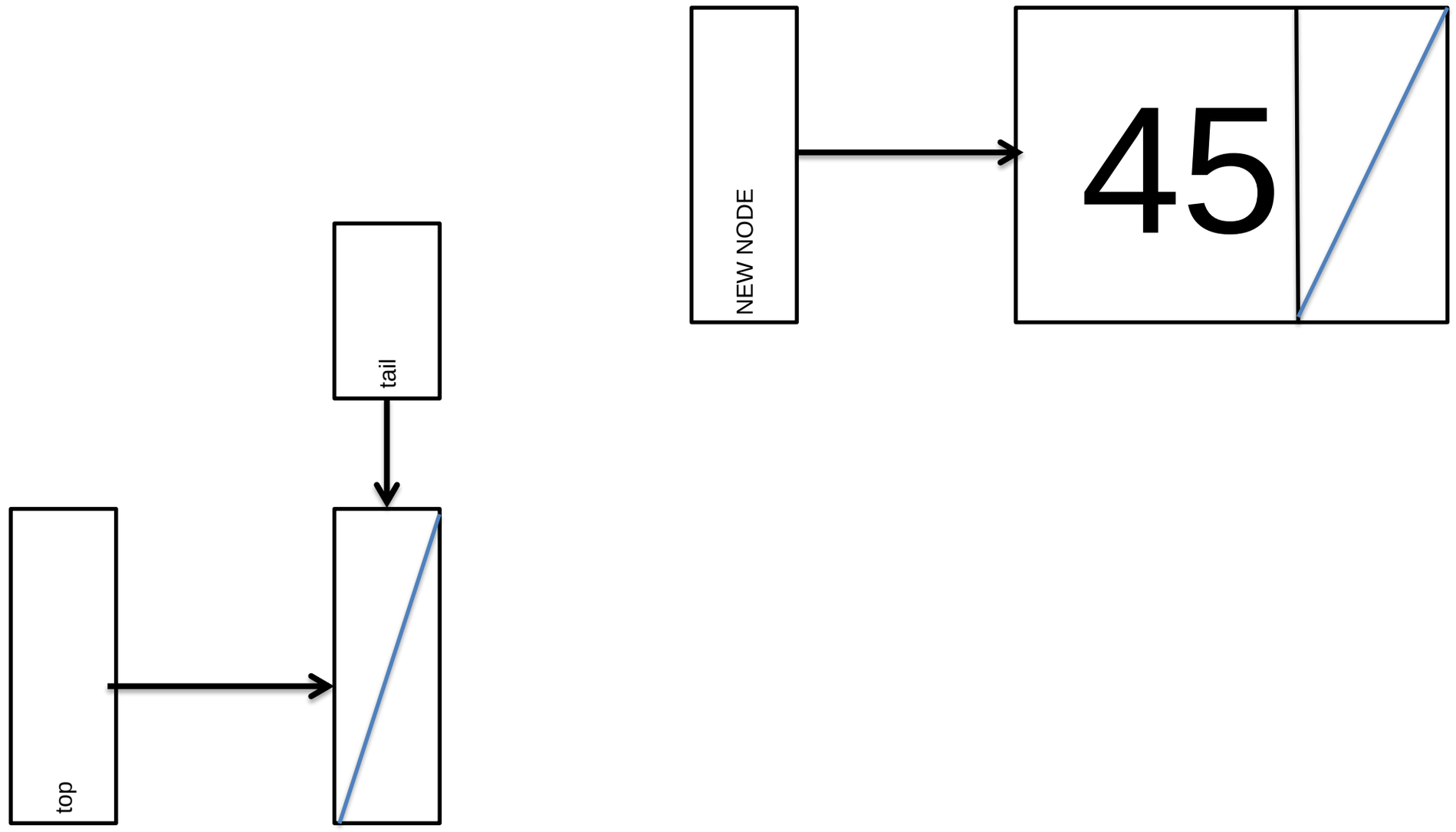
Queues as linked lists

- We can also implement a queue as a linked list
- We will enqueue to one side of the list
  - and dequeue to the other end
- For efficiency reasons we therefore keep track of the **head** and **tail** of the list



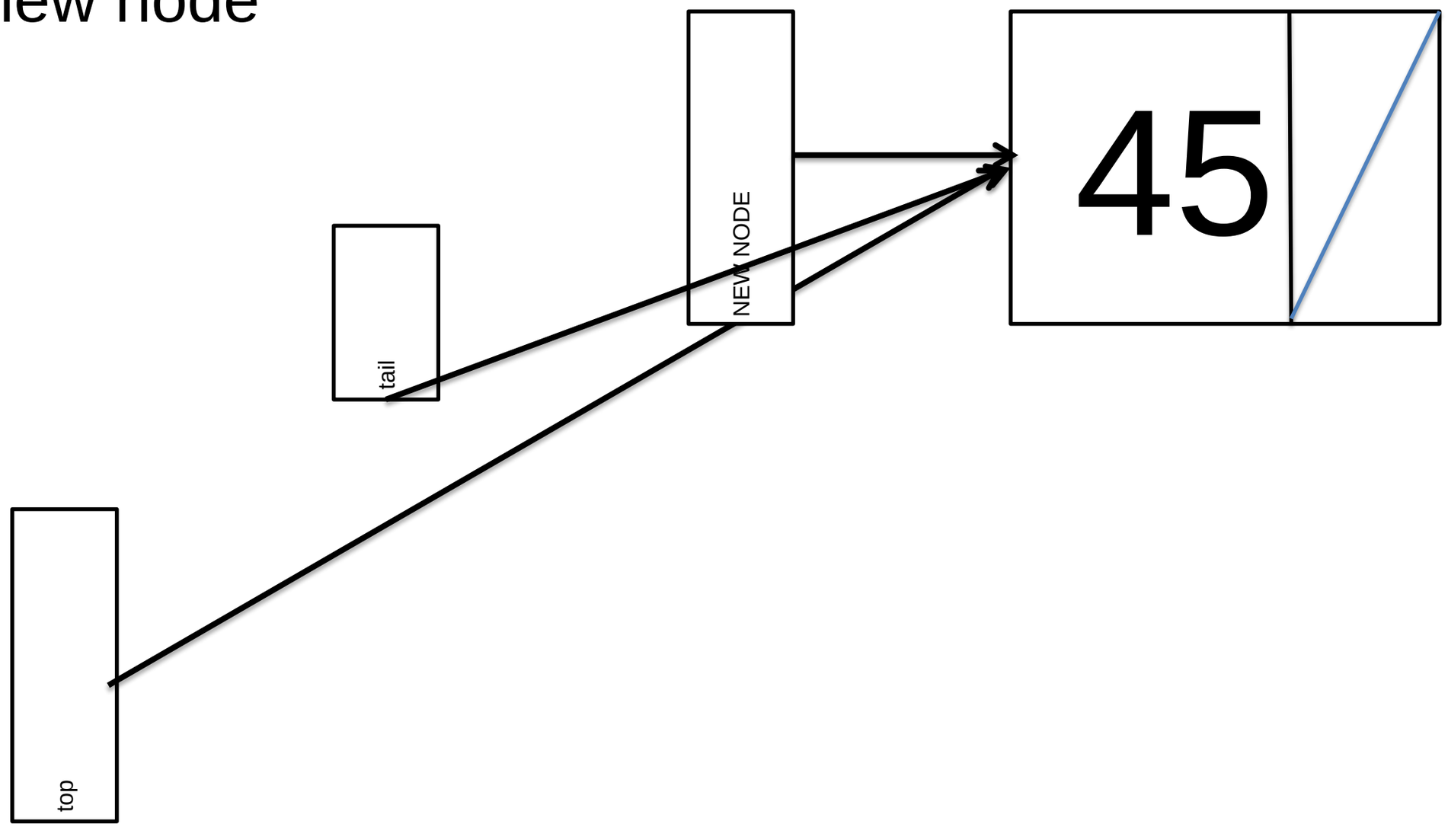
Enqueue for empty queue

1. Create the new node



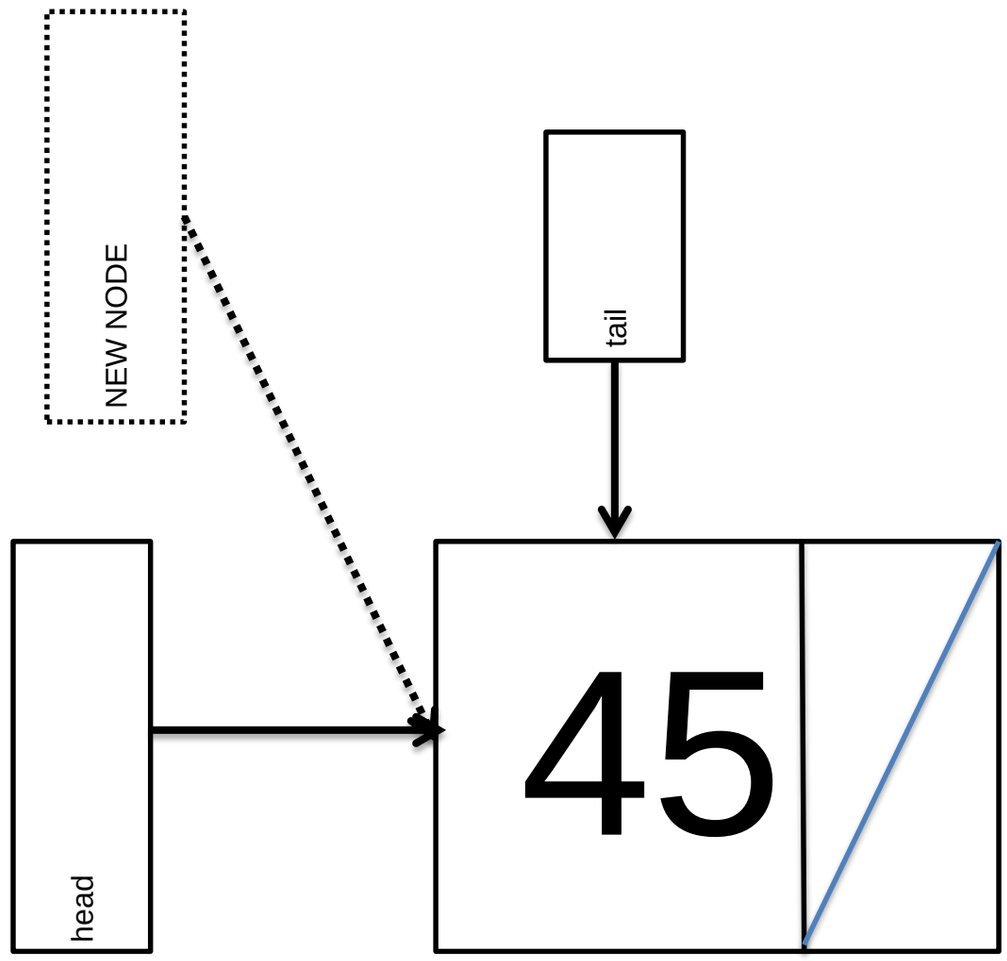
Enqueue for empty queue

2. Set head and tail to point to new node



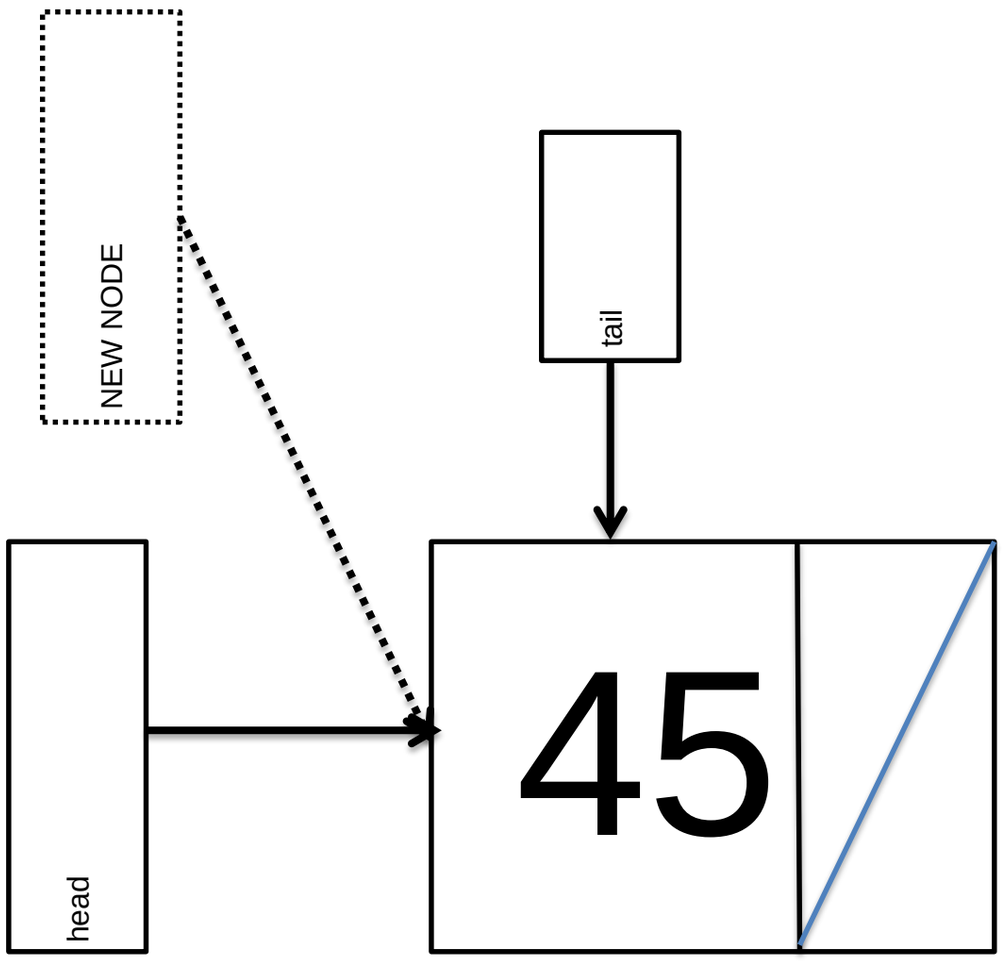
Enqueue for empty queue

Same as:



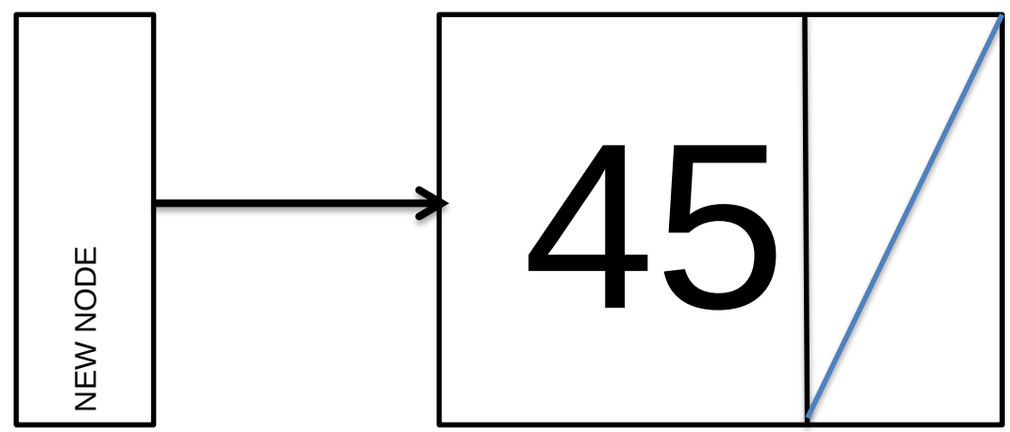
Enqueue for empty queue

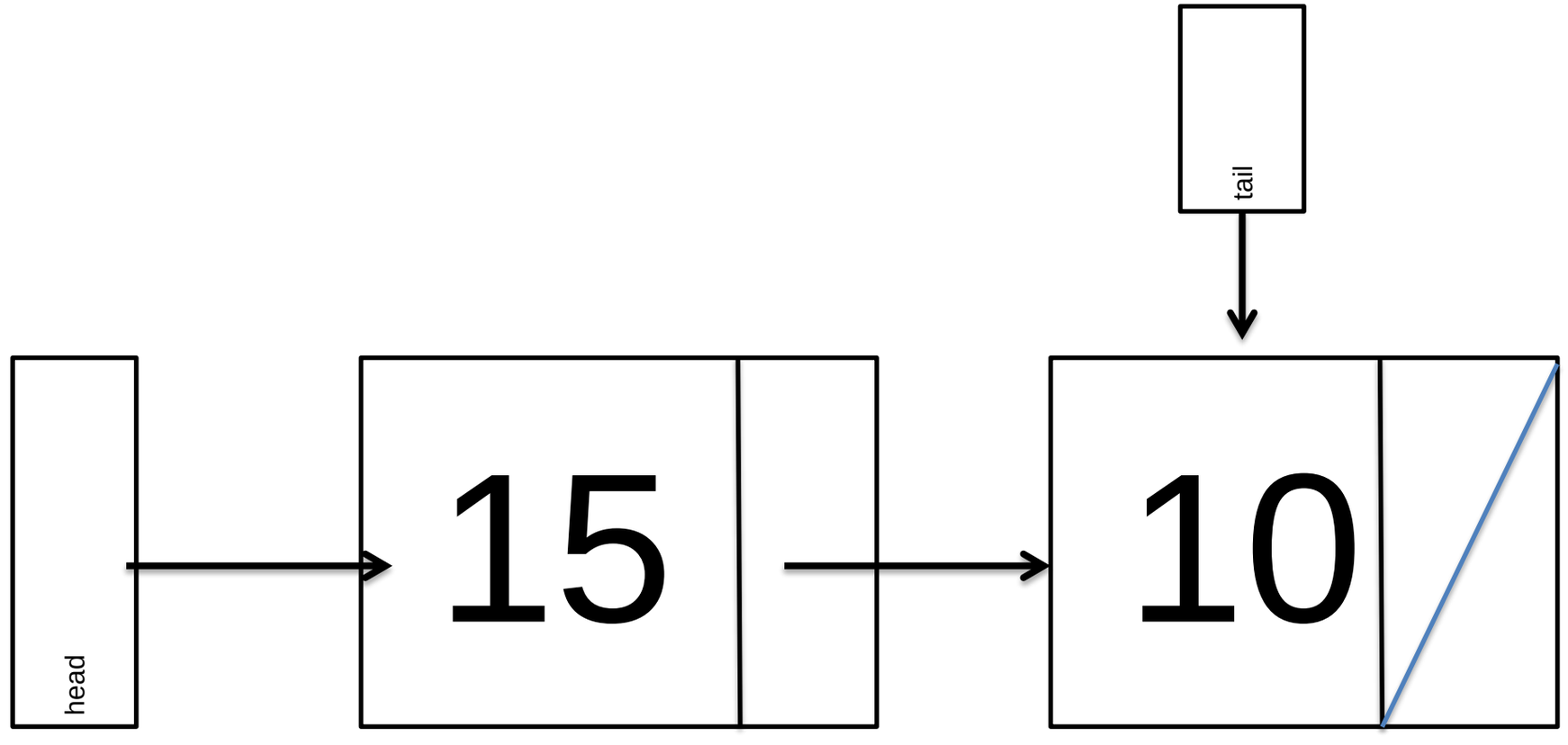
3. Increment **size** by one



Enqueue non-empty queue

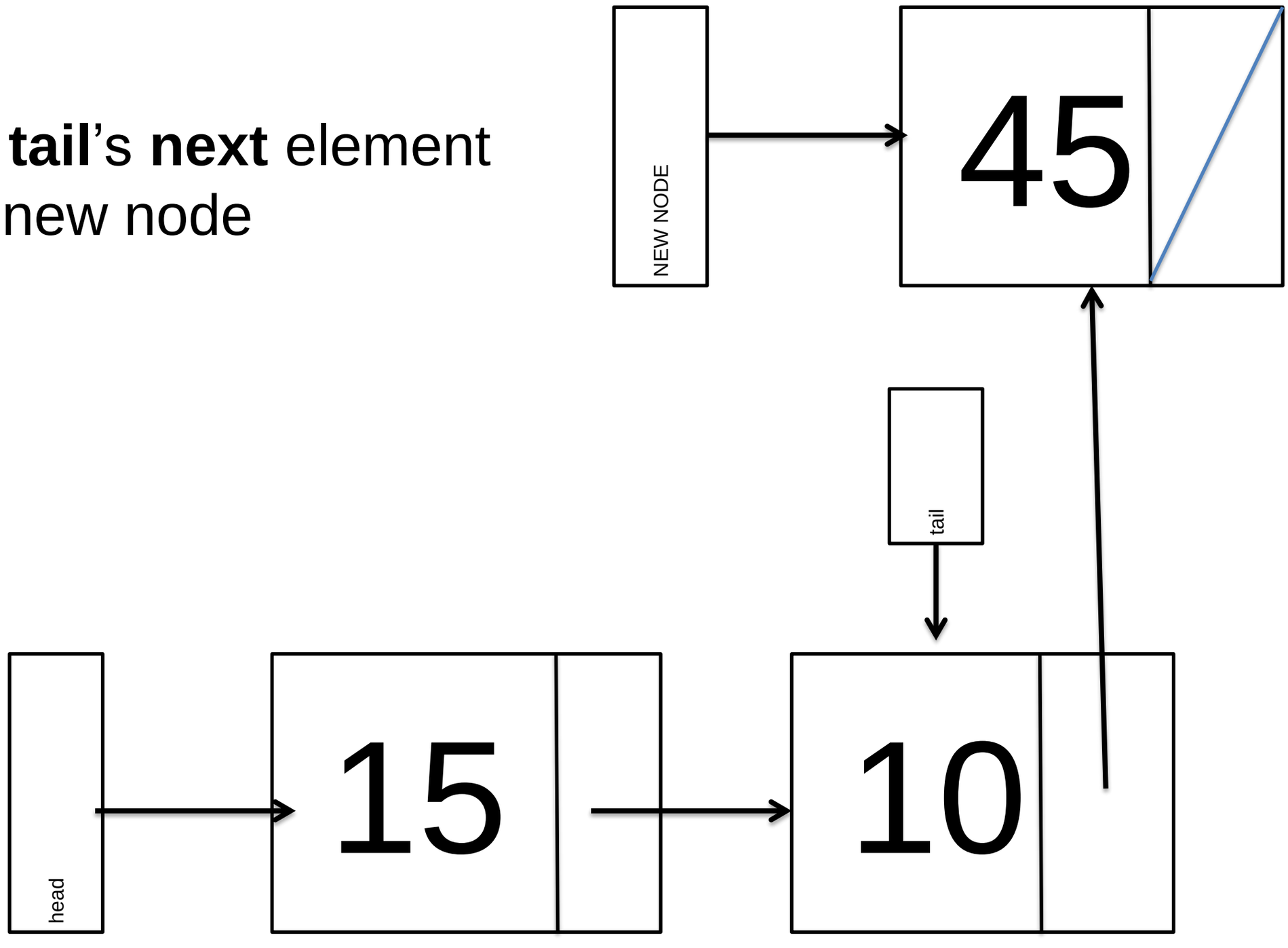
1. Create the new node





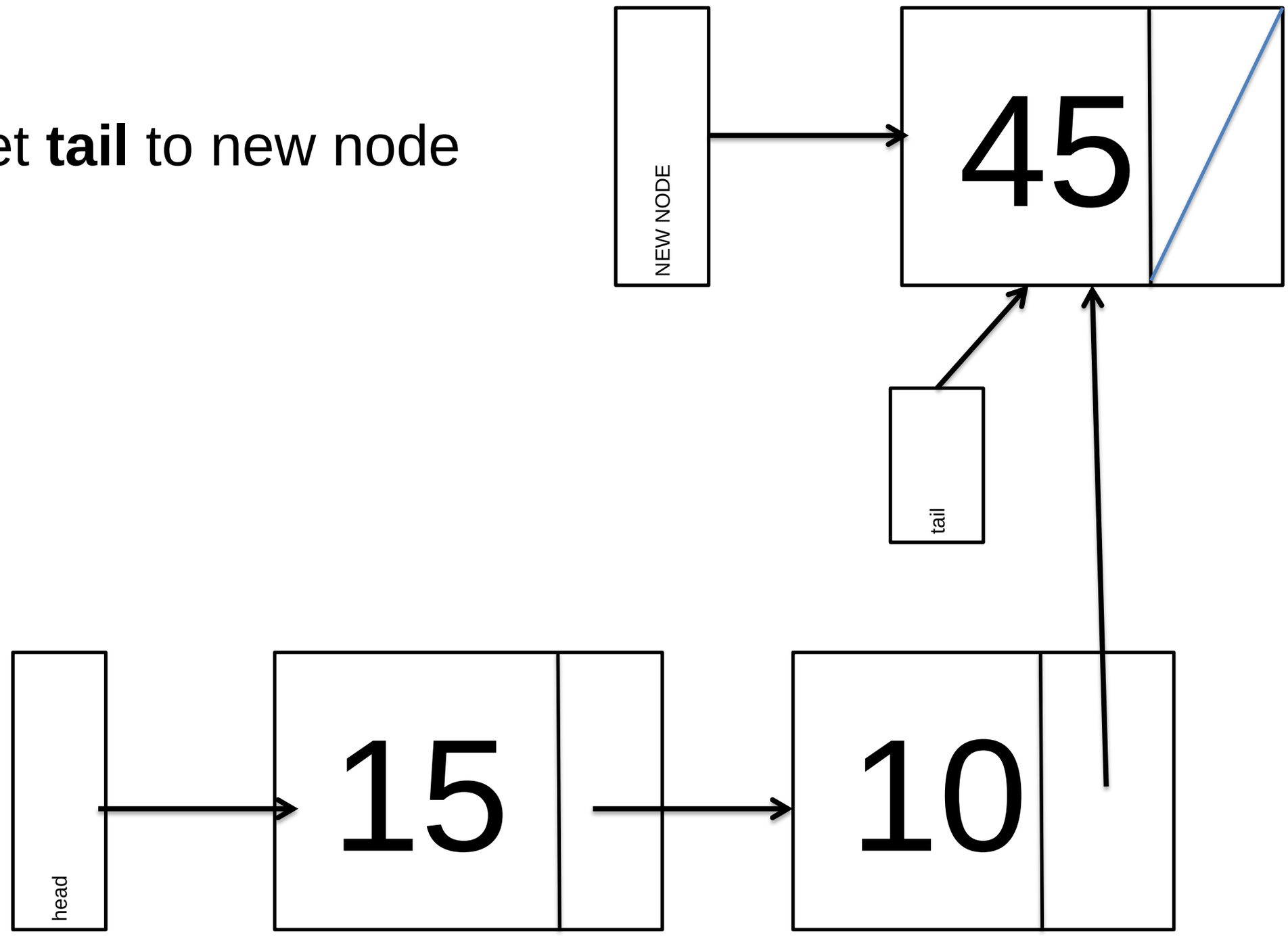
Enqueue non-empty queue

2. Set **tail**’s **next** element to the new node



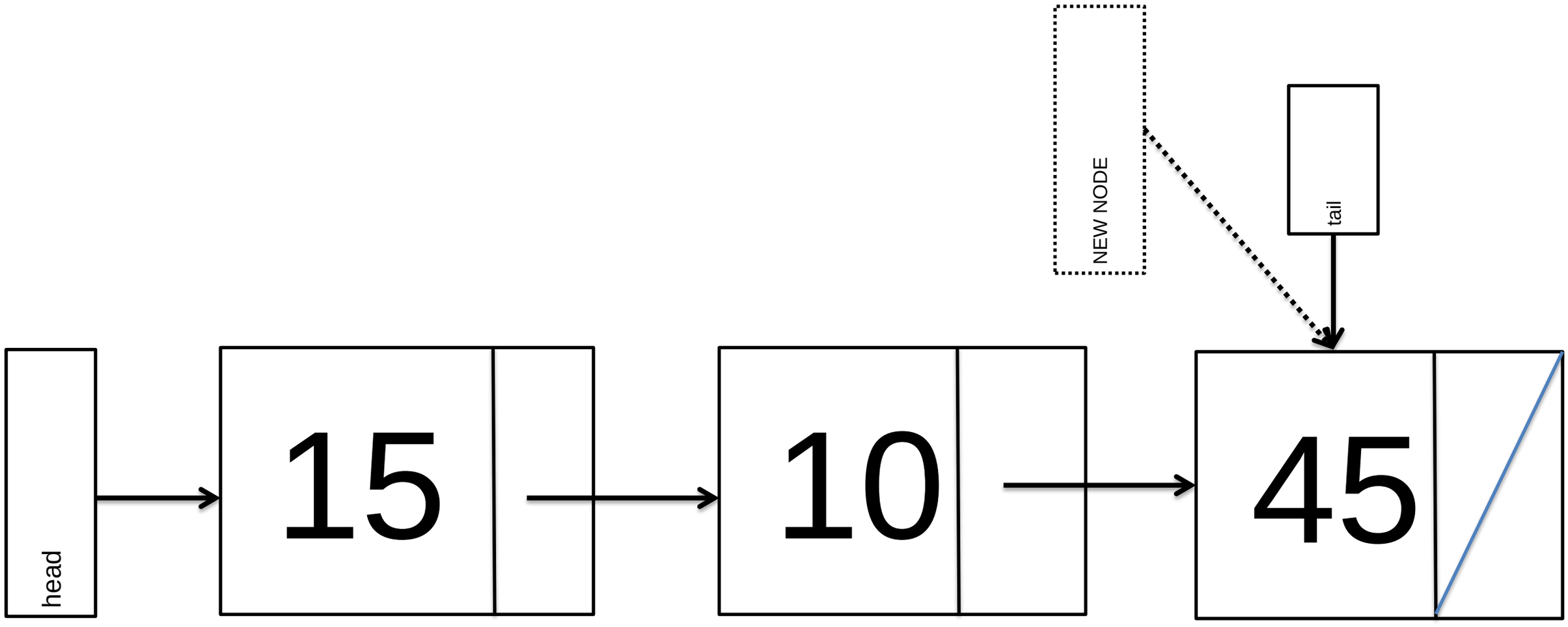
Enqueue non-empty queue

3. Set **tail** to new node



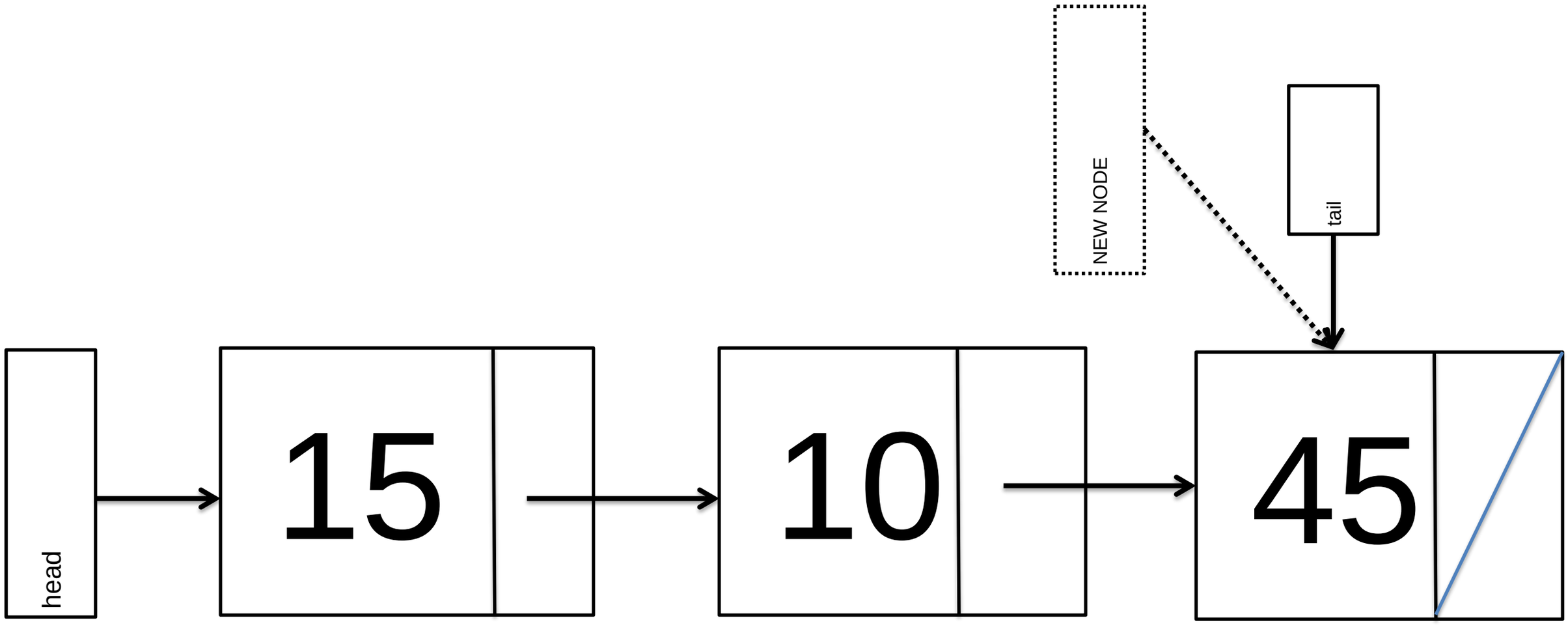
Enqueue non-empty queue

… which is the same as



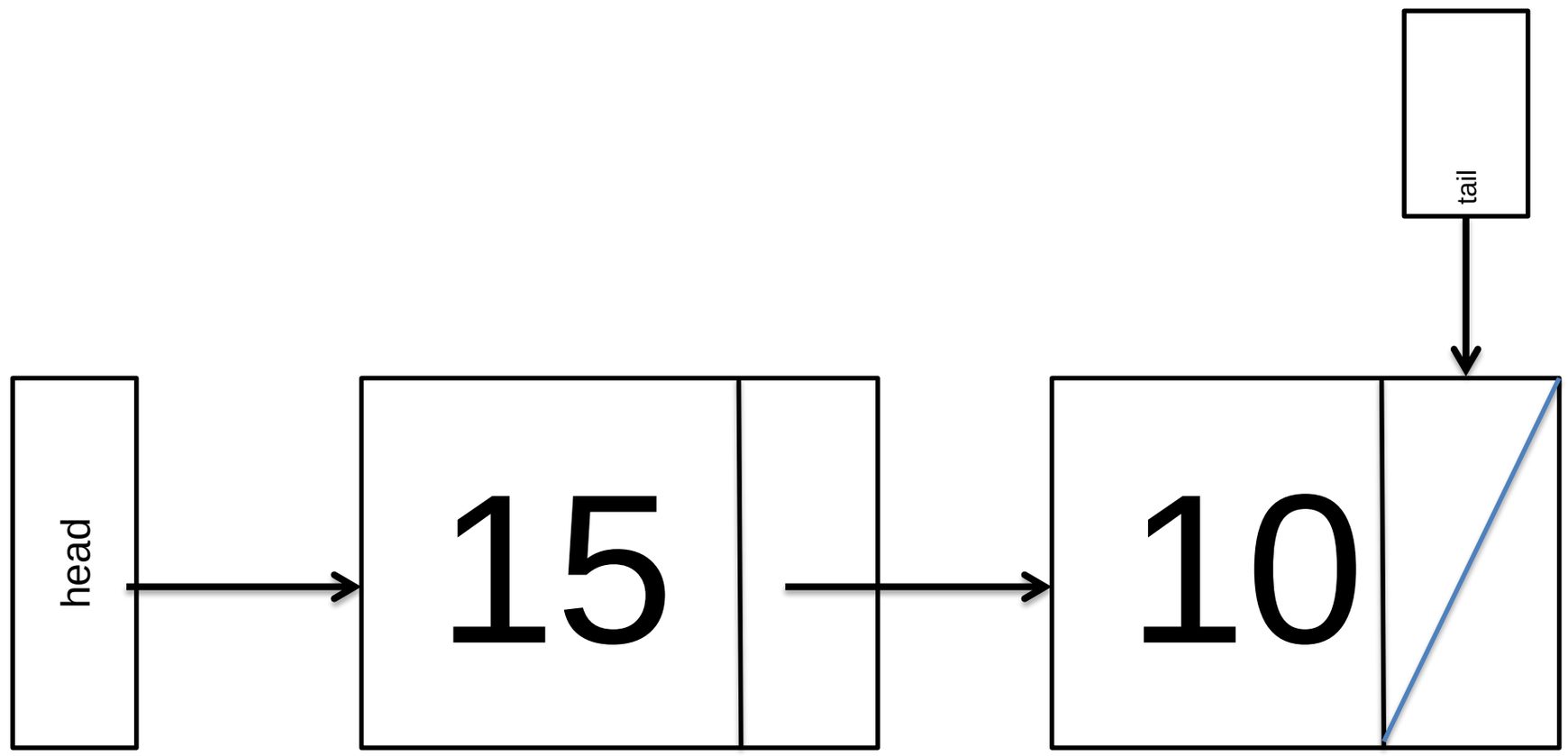
Enqueue non-empty queue

4. Increment **size** by one



Dequeue

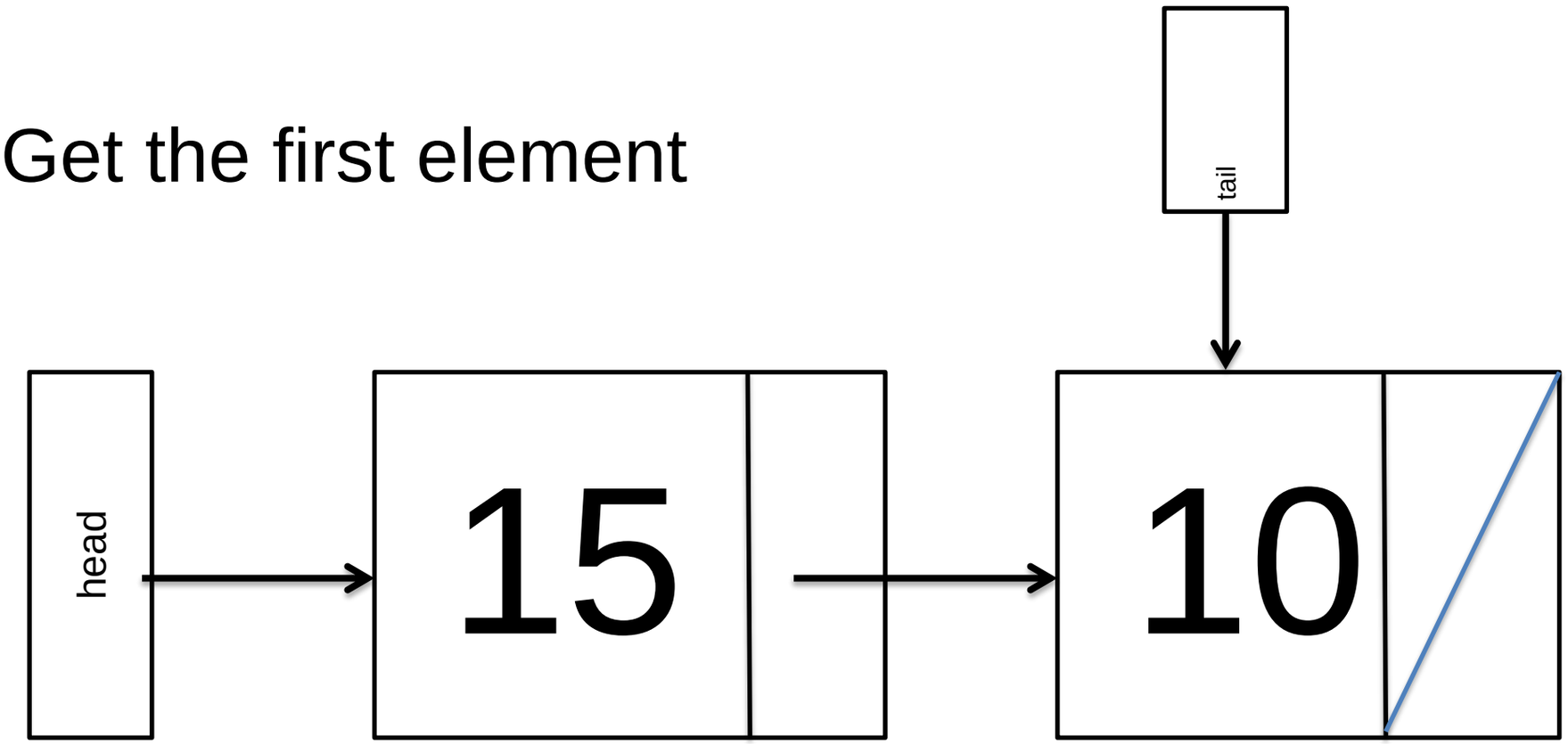
1. Check the queue isn’t Null



Head == NULL throws a **QueueException**

Dequeue

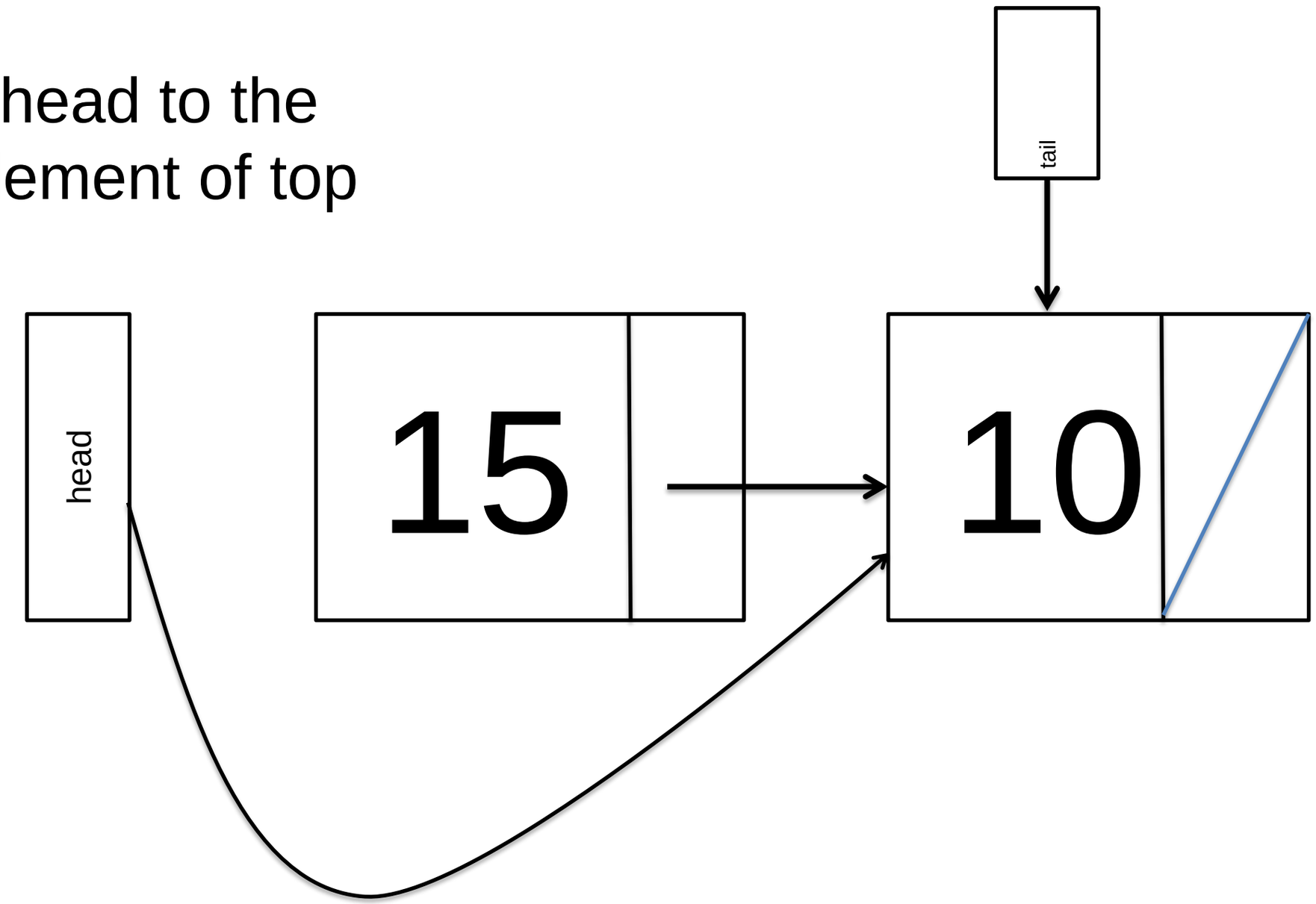
2. Get the first element



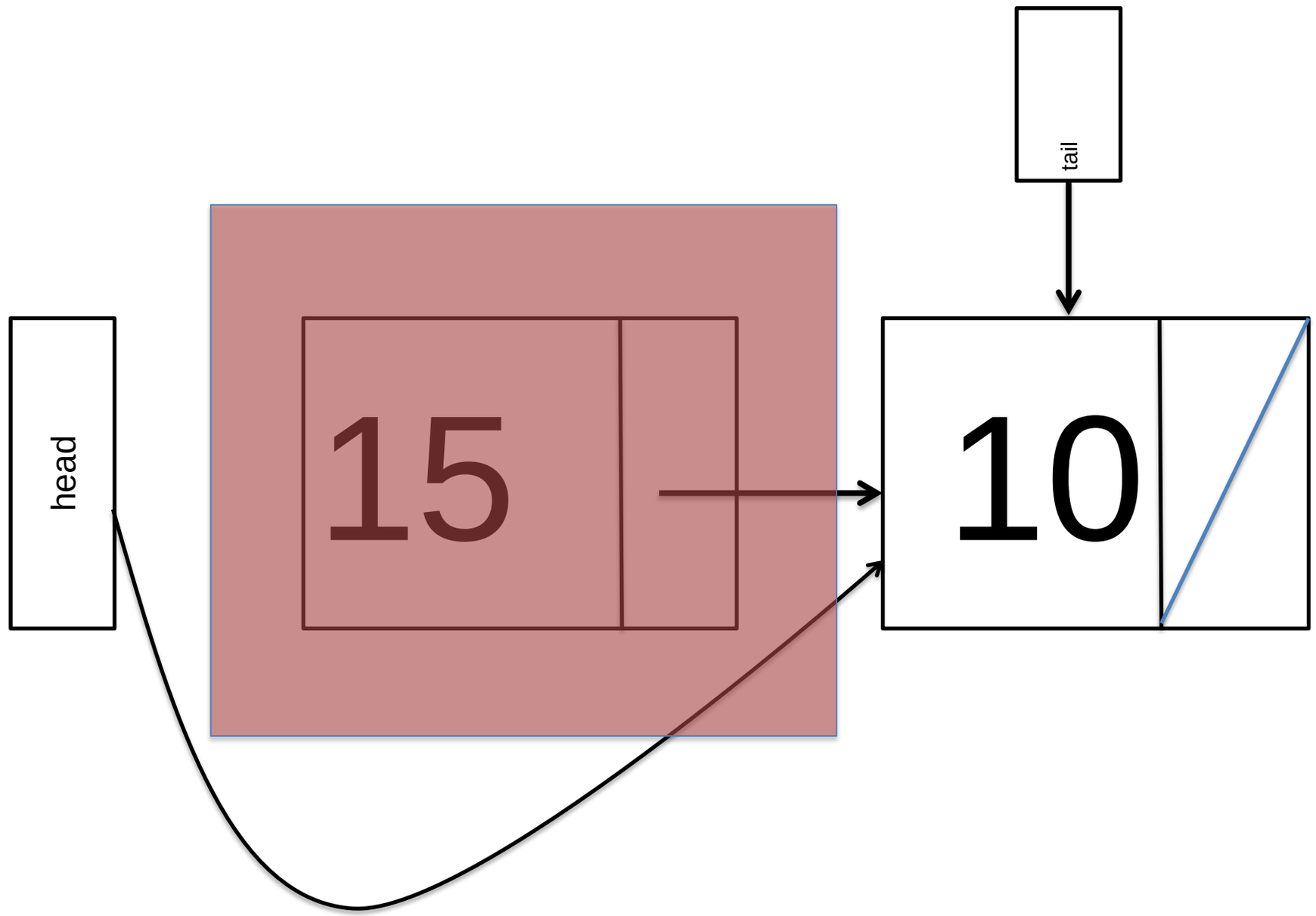
tmp = 15;

Dequeue

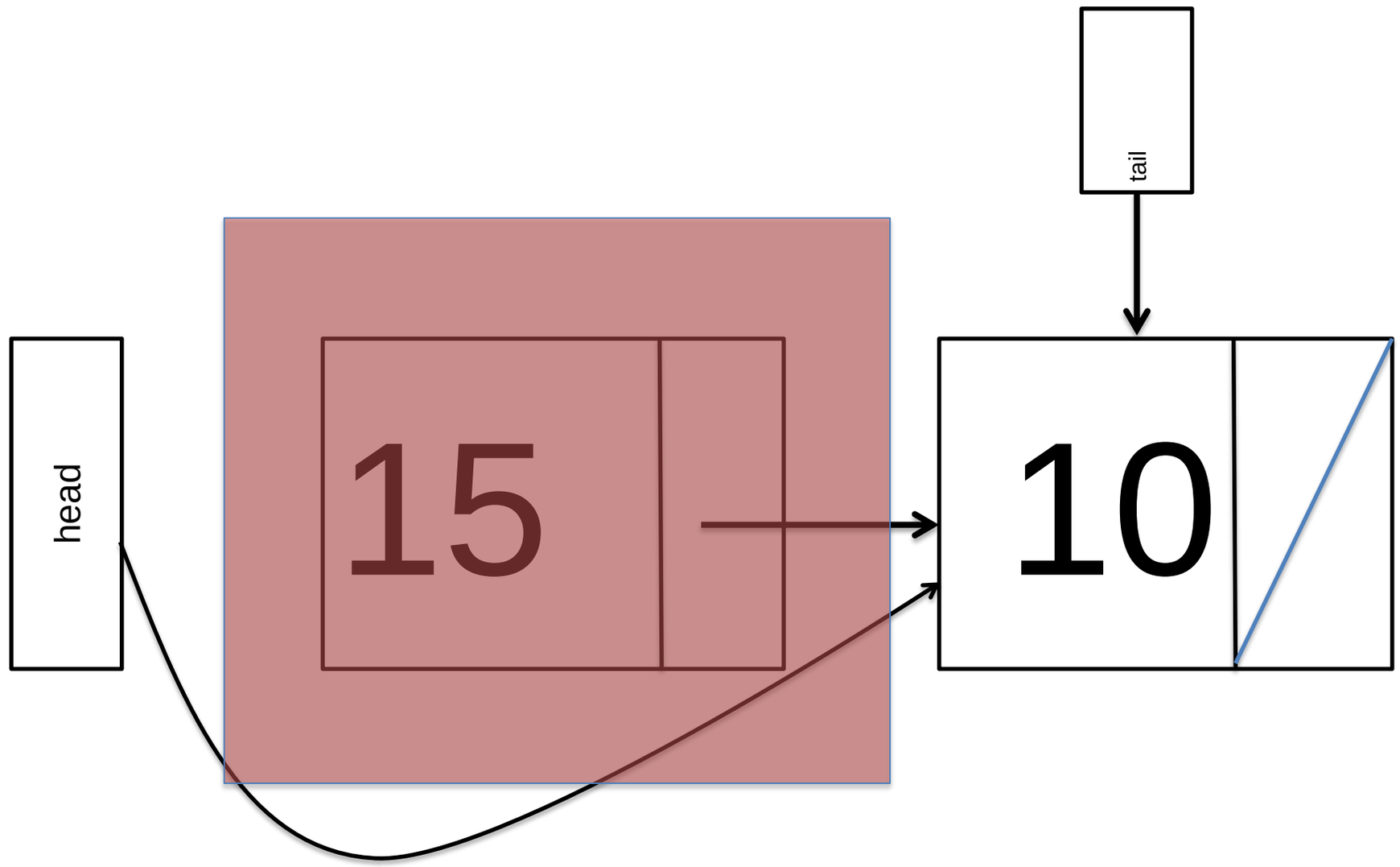
3. Set head to the next element of top



Dequeue

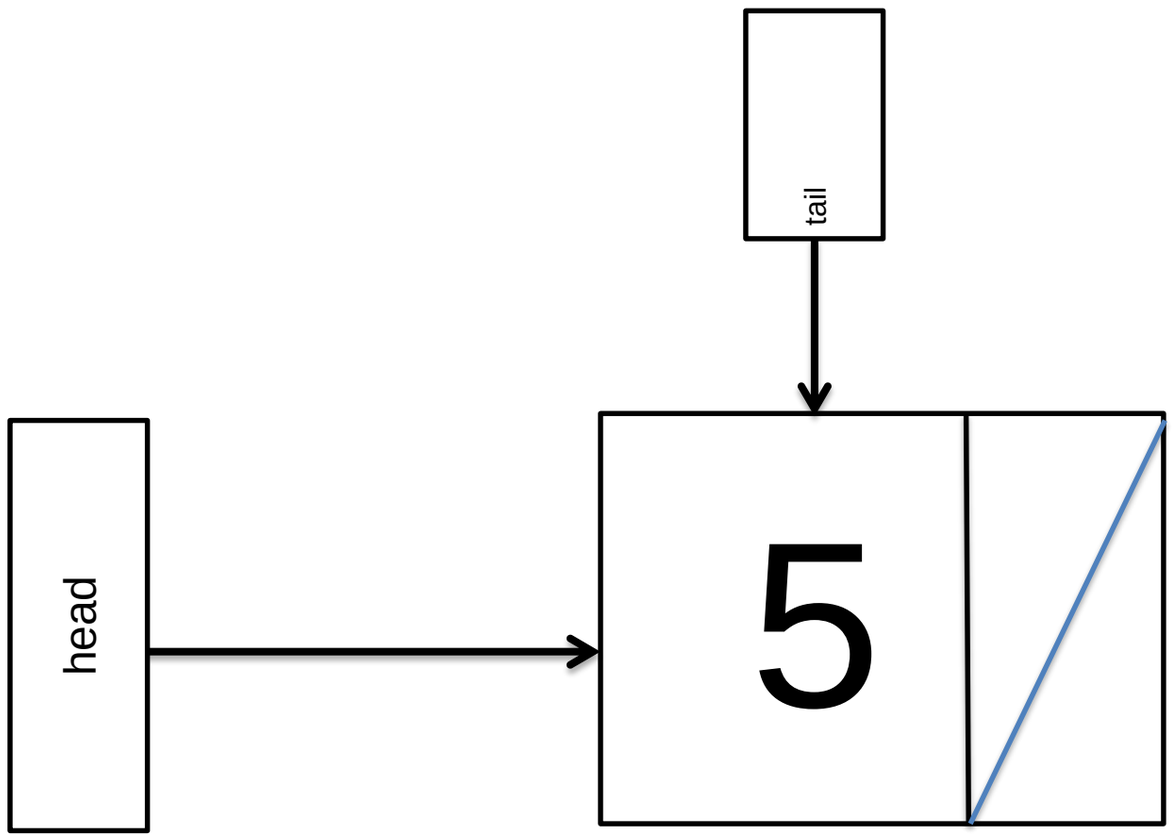


Dequeue



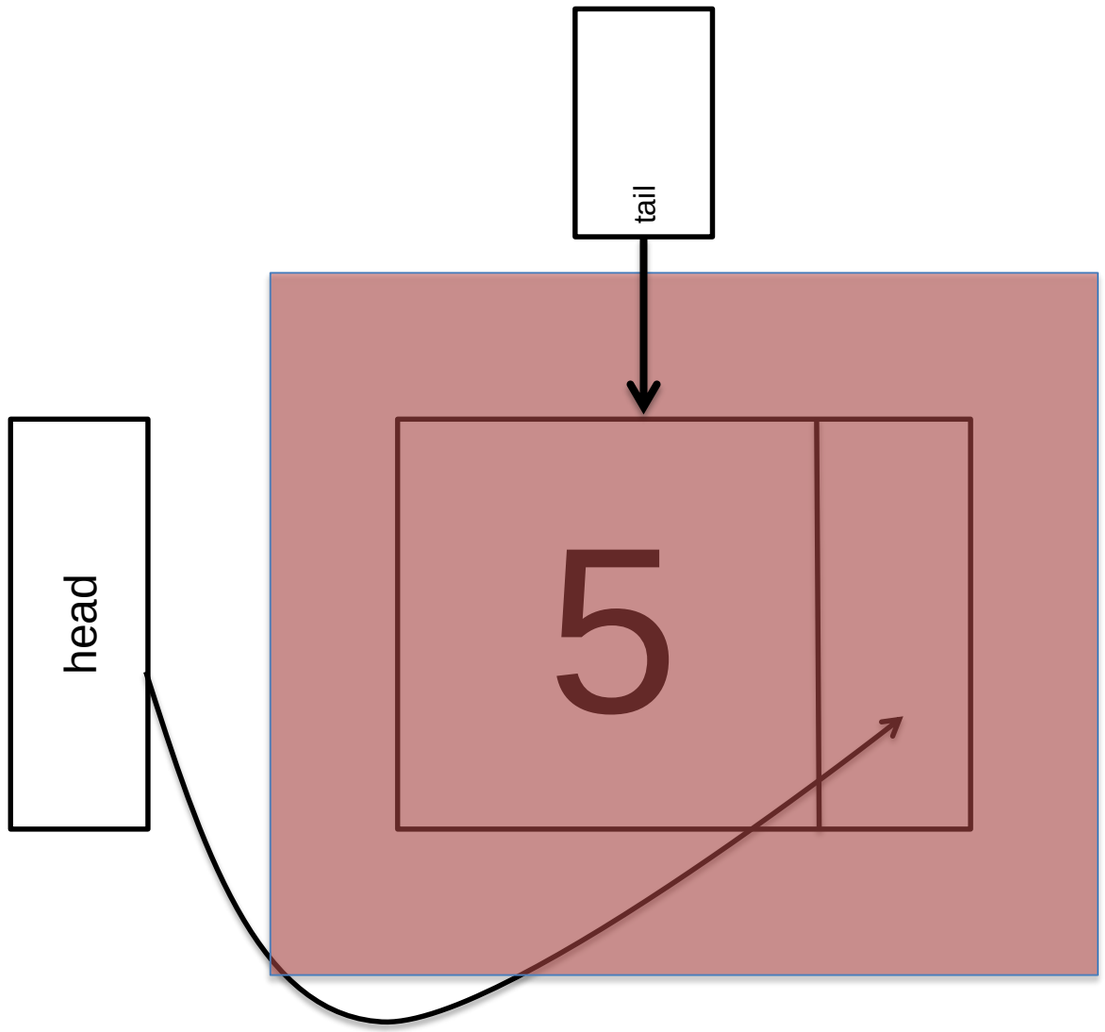
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



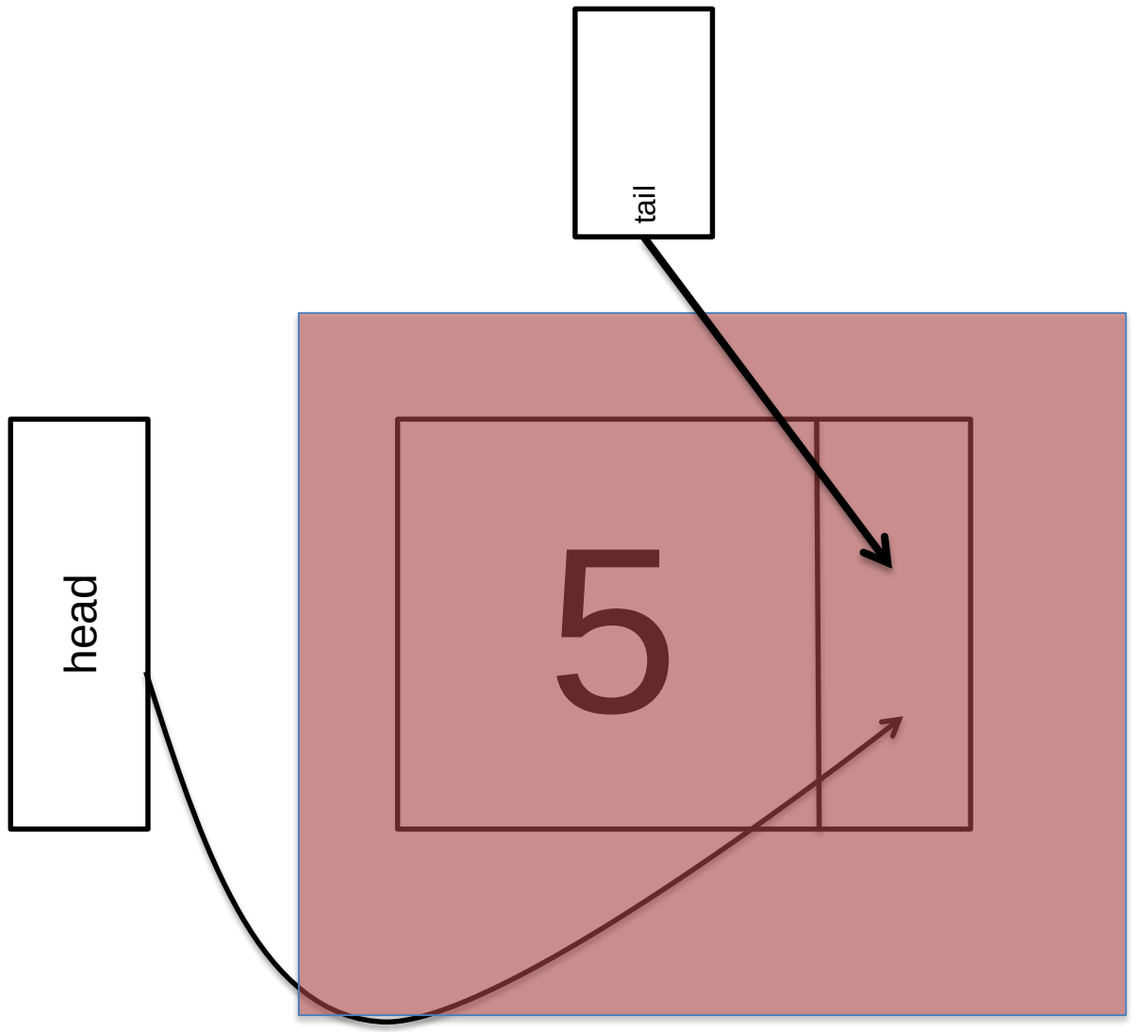
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



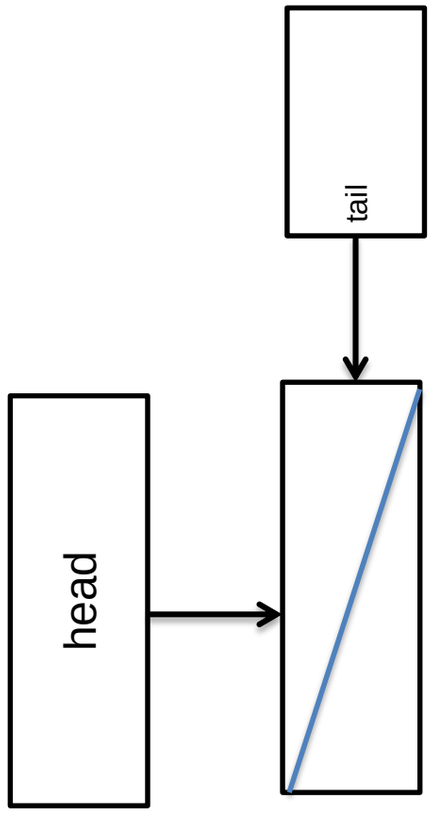
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



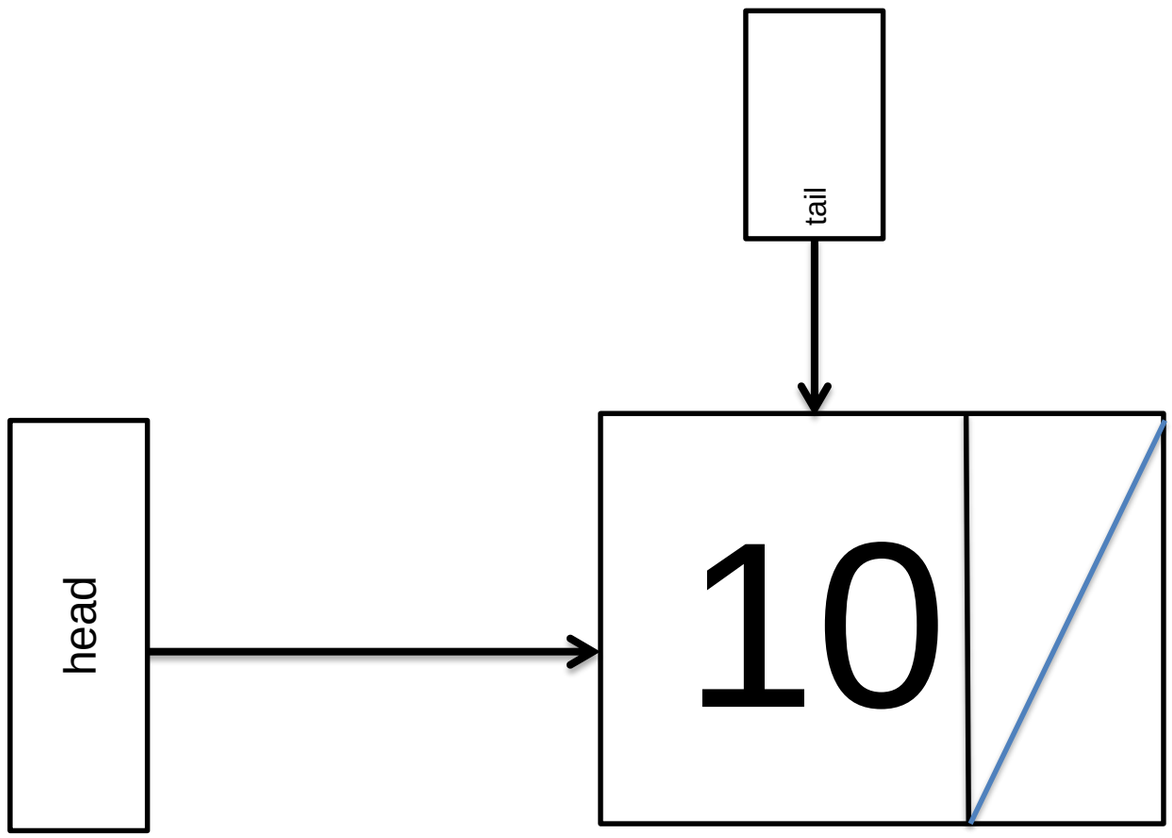
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



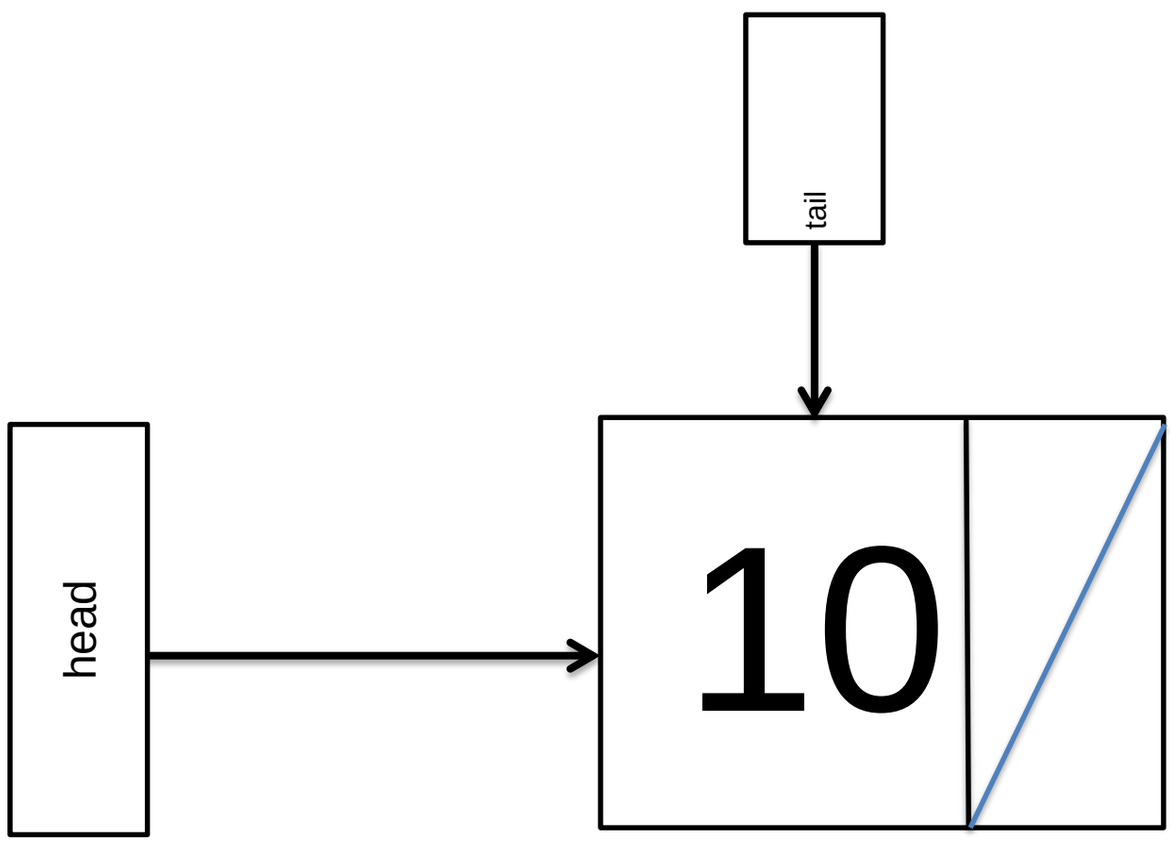
4. Set **tail** to NULL if new queue is empty

Dequeue



5. Decrement **size** by 1

Dequeue



6. Return the tmp element (15 in this case)

Demo: Queue ADT implemented with a linked list

Overview of data structures so far

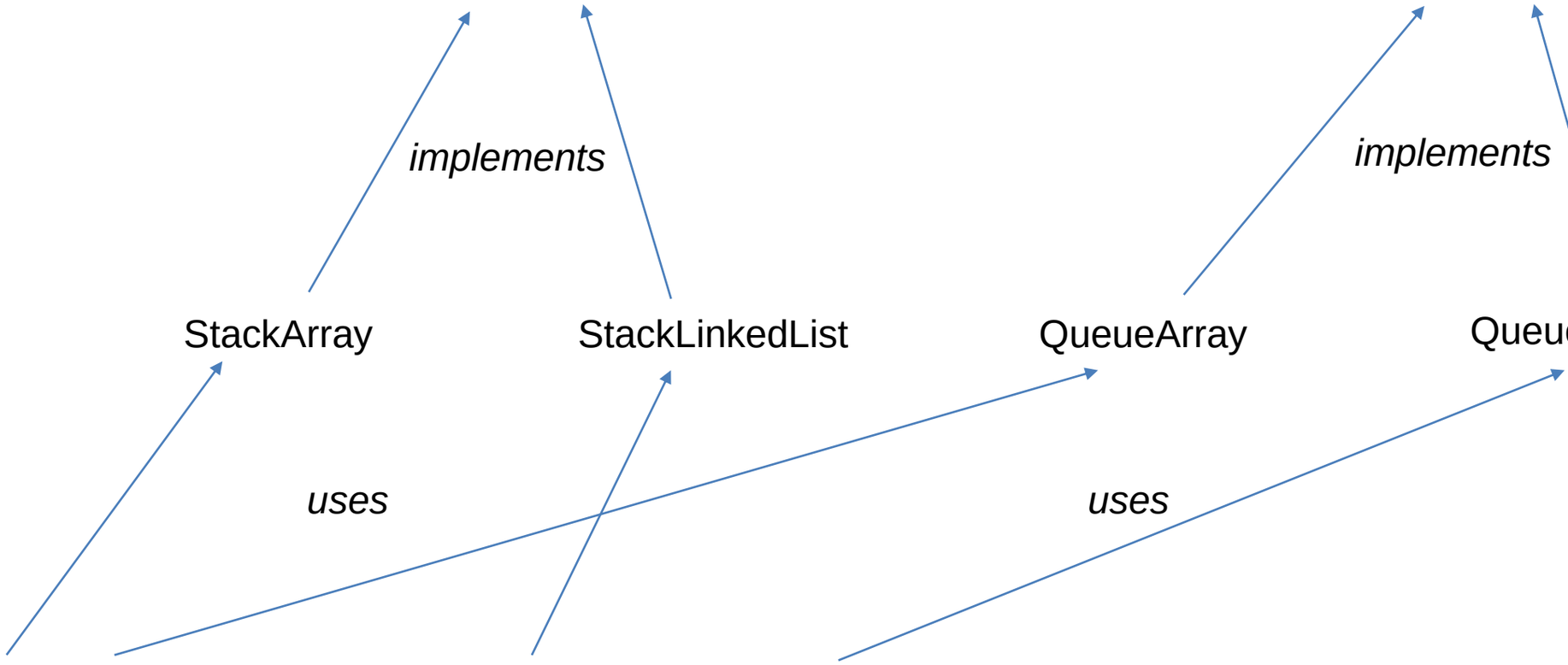
Interfaces

boolean isEmpty()
Object top() throws StackException
Object pop() throws StackException
void push(Object element)

Stack

void enqueue(Object element)
Object dequeue() throws QueueException
Object front() throws QueueException
int size()
boolean isEmpty()

Queue



Classes

StackArray

StackLinkedList

QueueArray

QueueLinkedList

Arrays
- Acccess values with [i]
- Update values with [i]
- E.g.

```
x = array[5];
array[7] = 15;
```

Linked Lists
- Acccess values with object references
- Add values
  - At head or tail
- Remove values
  - At head or tail

Doubly Linked Lists
- Same operations as Linked Lists
- Implementation is
  - More efficient
  - Also more detailed

Arrays versus Linked Lists: Implementation Details

- **Arrays**
  - Manually maintain size counter for push/pop and enqueue/dequeuer
  - Write new elements directly into array positions with “[i]”
  - Read elements directly from array positions with “[i]”
  - Fast access
  - ***Limitation***: fixed size stacks and queues

- **Linked Lists**
  - For efficiency, manually maintain a size counter
  - Use object references to locate nodes in stacks and queues
    - Global references *e.g.* head and last
    - Object references within other objects *e.g.* node.nextNode()
  - Removing elements at the end of a linked list is O(N)
  - ***Flexibility***: this is a dynamic data structure, it can grow and shrink

- **Doubly Linked Lists**
  - Makes traversal of a list easier
  - Potential time complexity saving
  - ***Efficiency***: removing elements from tail of a list
  - ***Implementation***: more book keeping involved with object references

Java data structures

- **List** *interface* https://docs.oracle.com/javase/8/docs/api/java/util/List.html
  - Implementations include: ArrayList, LinkedList, …

- **Stack** *interface* *https://docs.oracle.com/javase/7/docs/api/java/util/Stack.html*
  - Implementations include: List (interface), RandomAccess, …

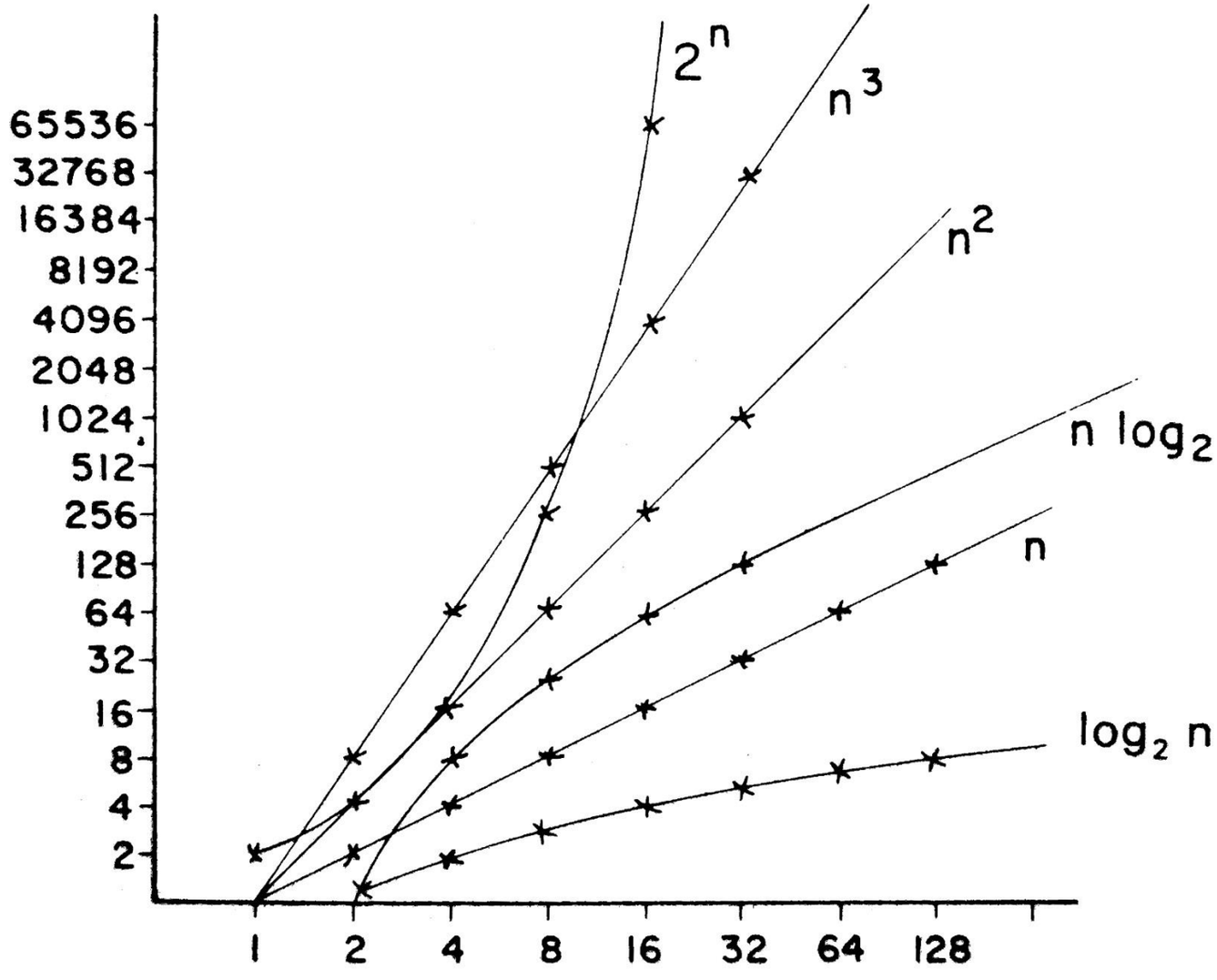
- **Queue** *interface* https://docs.oracle.com/javase/7/docs/api/java/util/Queue.html
  - Implementations include: LinkedList, ArrayDeque, …

This separation between

- **Interfaces** - the application programming interface (API).
- **Implementations –** implement interfaces, providing the user with potentially multiple to choose from.

is what we have covered in SD3 so far.

Illustration of Growth Rates



Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

Demo: benchmarking stacks and queues

- Measuring time complexity of
  - Complexity of “size” on a linked list based stack
    - Naïve “size” with list traversal.
    - Optimised with “size” counter.
  - Queue operations.
  - Stack operations.

Our target: low complexity of our stack/queue algorithms.

O(1) is better than O(N)