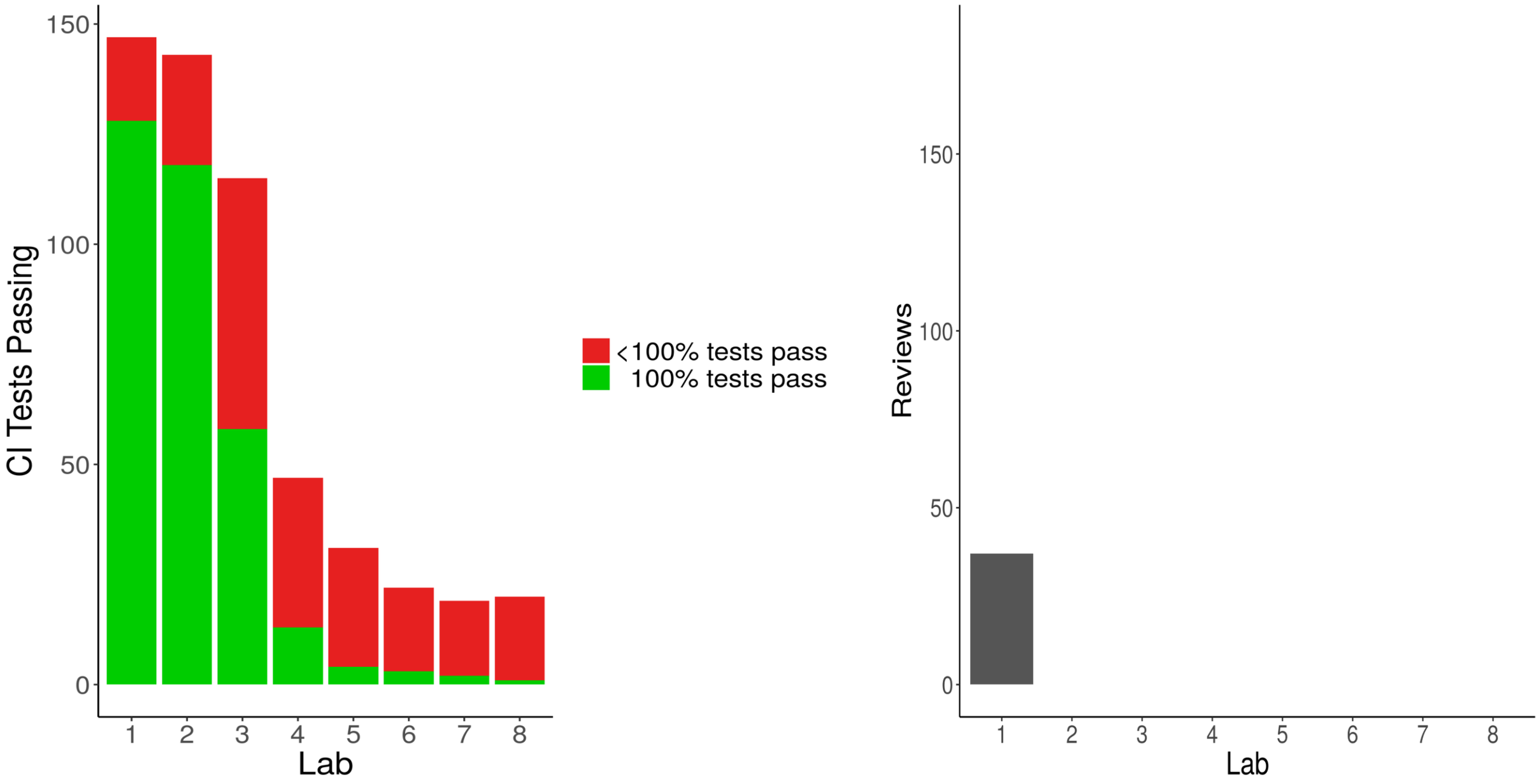
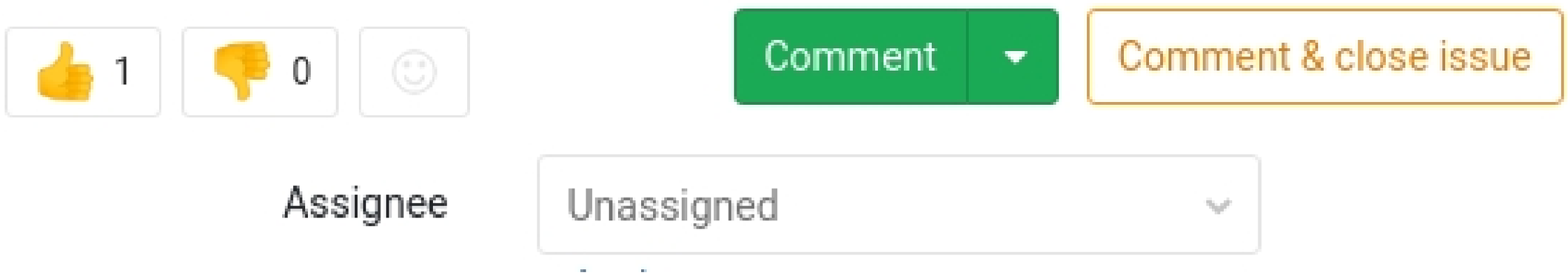

CI Tests Passing
150
100
50
0
1
2
3
4
5
6
7
8
Lab
<100% tests pass
100% tests pass
Reviews
150
100
50
0
1
2
3
4
5
6
7
8
Lab

- 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 8

Dynamic Stacks

Rob Stewart

Outline

- we implemented a stack using an array (lecture 4)
- Today: **pictorially** implement stack with linked list

- **You will then implement this in the lab**

- First.. let's revise the stack ADT

The Stack ADT

- 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

Advantages

- Linked lists
  - It can change size dynamically
  - Allocated only the space you need
- Arrays
  - Random access
  - Can be faster
  - More efficient representation of arrays in memory

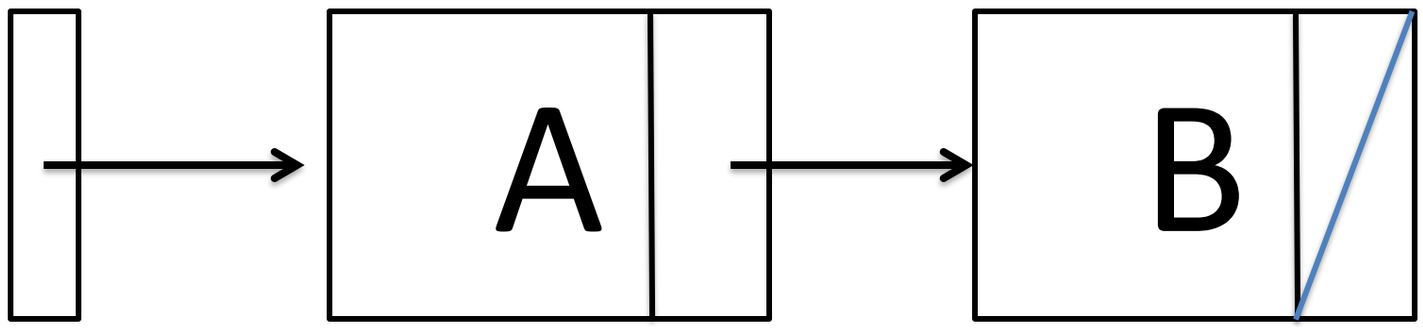
The Stack ADT as a Java Interface

- We can represent the Stack ADT as a Java interface
- Elements are of type **Object**
- 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;
}
```

Linked Lists revision

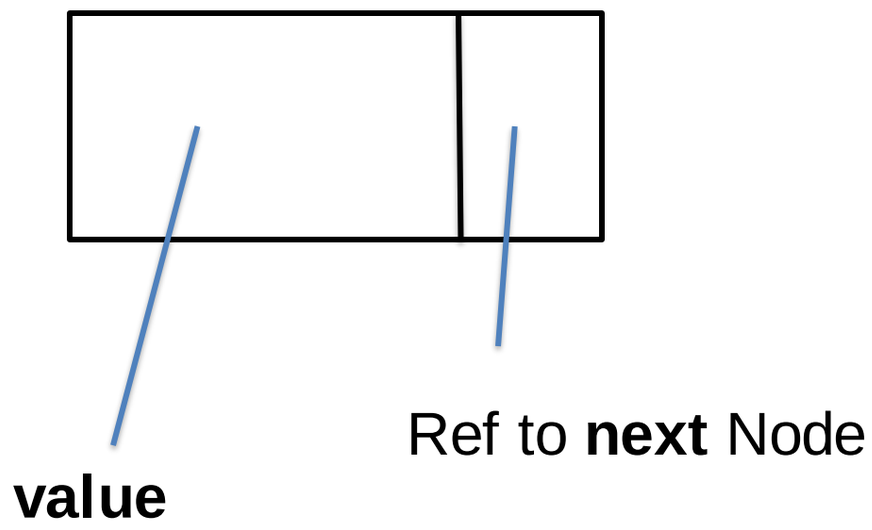
- A Linked List is a **Linear** ordered sequence of **Nodes**
- Each **Node** contains
  - a data item
  - a reference to another Node of the same type (Class)
- We have a reference (**head**) to the first element
- Traverse through the chain of nodes to visit each one



Stacked as linked structure - data

```
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

First a linked list exercise…

Assume a LinkedList Class where the following operations can be performed:

- void addAtTail(int i);
  - adds the given value to the end of the linked-list;
- void removeValueAtIndex(int i);
  - removes the value at the index specified in i
- void addAtHead(int i);
  - adds the given value to the beginning of the linked-list

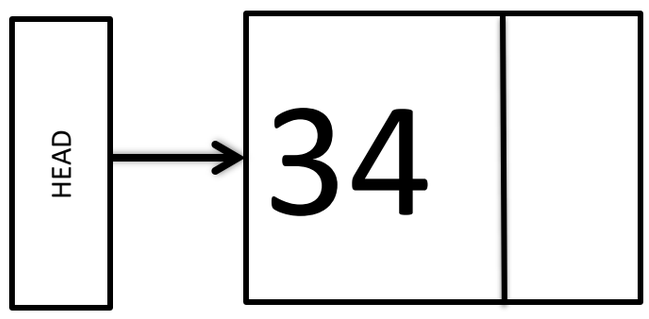
```
LinkedList ll = new LinkedList();
ll.addAtTail(34);
ll.addAtTail(22);
ll.addAtHead(33);
```

First a linked list exercise…

Assume a LinkedList Class where the following operations can be performed:

- void addAtTail(int i);
  - adds the given value to the end of the linked-list;
- void removeValueAtIndex(int i);
  - removes the value at the index specified in i
- void addAtHead(int i);
  - adds the given value to the beginning of the linked-list

```
LinkedList ll = new LinkedList();
ll.addAtTail(34);
ll.addAtTail(22);
ll.addAtHead(33);
```

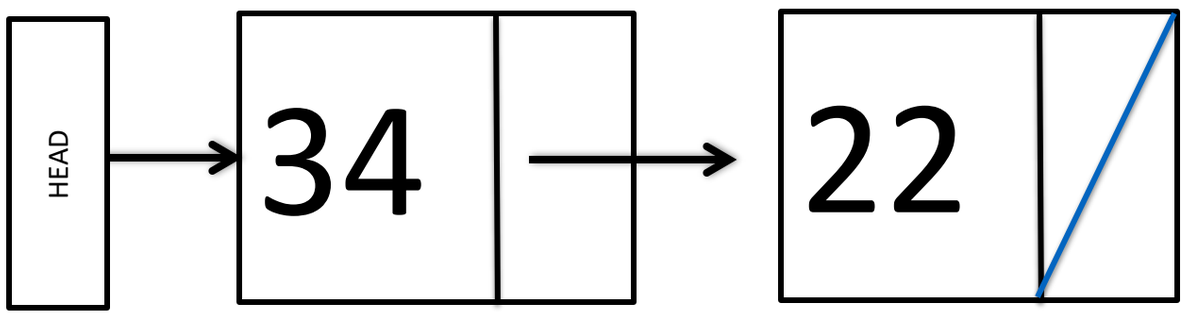


First a linked list exercise…

Assume a LinkedList Class where the following operations can be performed:

- void addAtTail(int i);
  - adds the given value to the end of the linked-list;
- void removeValueAtIndex(int i);
  - removes the value at the index specified in i
- void addAtHead(int i);
  - adds the given value to the beginning of the linked-list

```
LinkedList ll = new LinkedList();
ll.addAtTail(34);
ll.addAtTail(22);
ll.addAtHead(33);
```

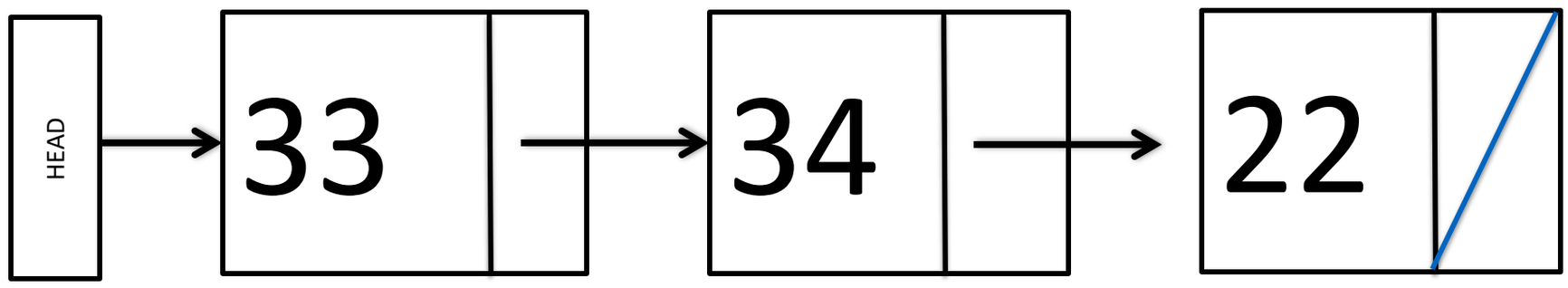


First a linked list exercise…

Assume a LinkedList Class where the following operations can be performed:

- void addAtTail(int i);
  - adds the given value to the end of the linked-list;
- void removeValueAtIndex(int i);
  - removes the value at the index specified in i
- void addAtHead(int i);
  - adds the given value to the beginning of the linked-list

```
LinkedList ll = new LinkedList();
ll.addAtTail(34);
ll.addAtTail(22);
ll.addAtHead(33);
```



Stack as a linked list

- We always aim for O(1) constant complexity
  - push, pop, size, isEmpty, top
  - We achieved this with **array** based implementation
- **Push**: create new node, add it to linked list
- **Pop**: remove node from linked list
- When we are using a **linked list** data structure
  - `push()` add at tail, or head of linked list?
  - `pop()` remove from tail, or head of linked list?

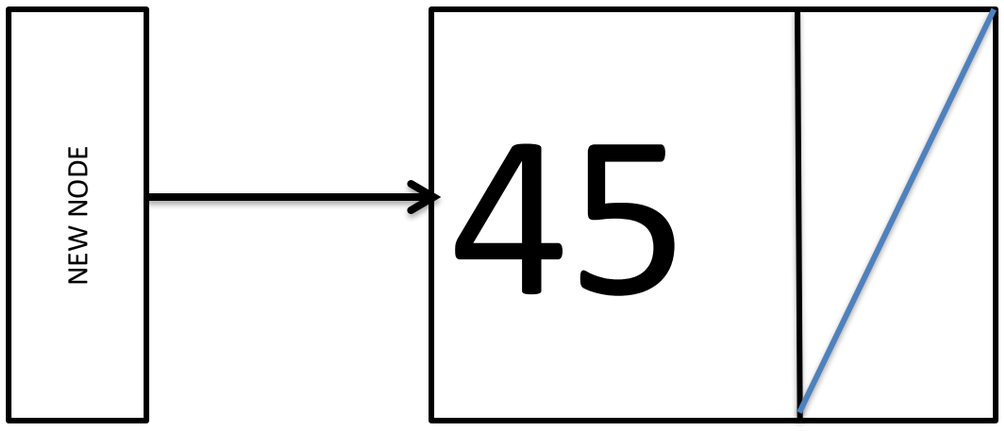
www.socrative.com / `student login' / room: SD32019

Pushing an element

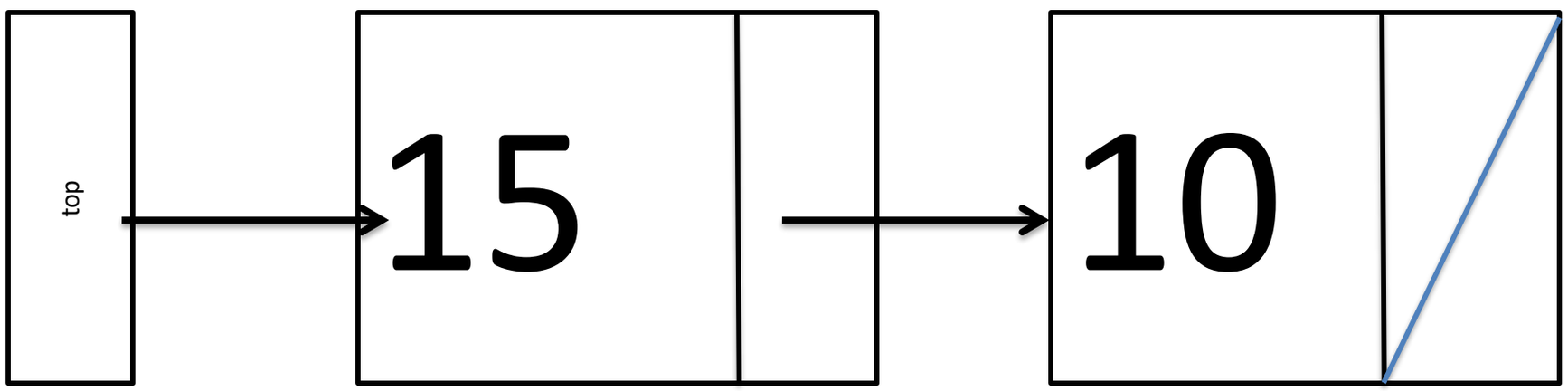
- Solution:
  - **Add to the front**
- Why?
  - This is a constant operation
  - Adding to the rear will force us to traverse the list
    - i.e. linear time operation

- How to push
  - Create the New Node
  - Set its reference to be the existing **top**
  - Set the New Node to be the **top** Node

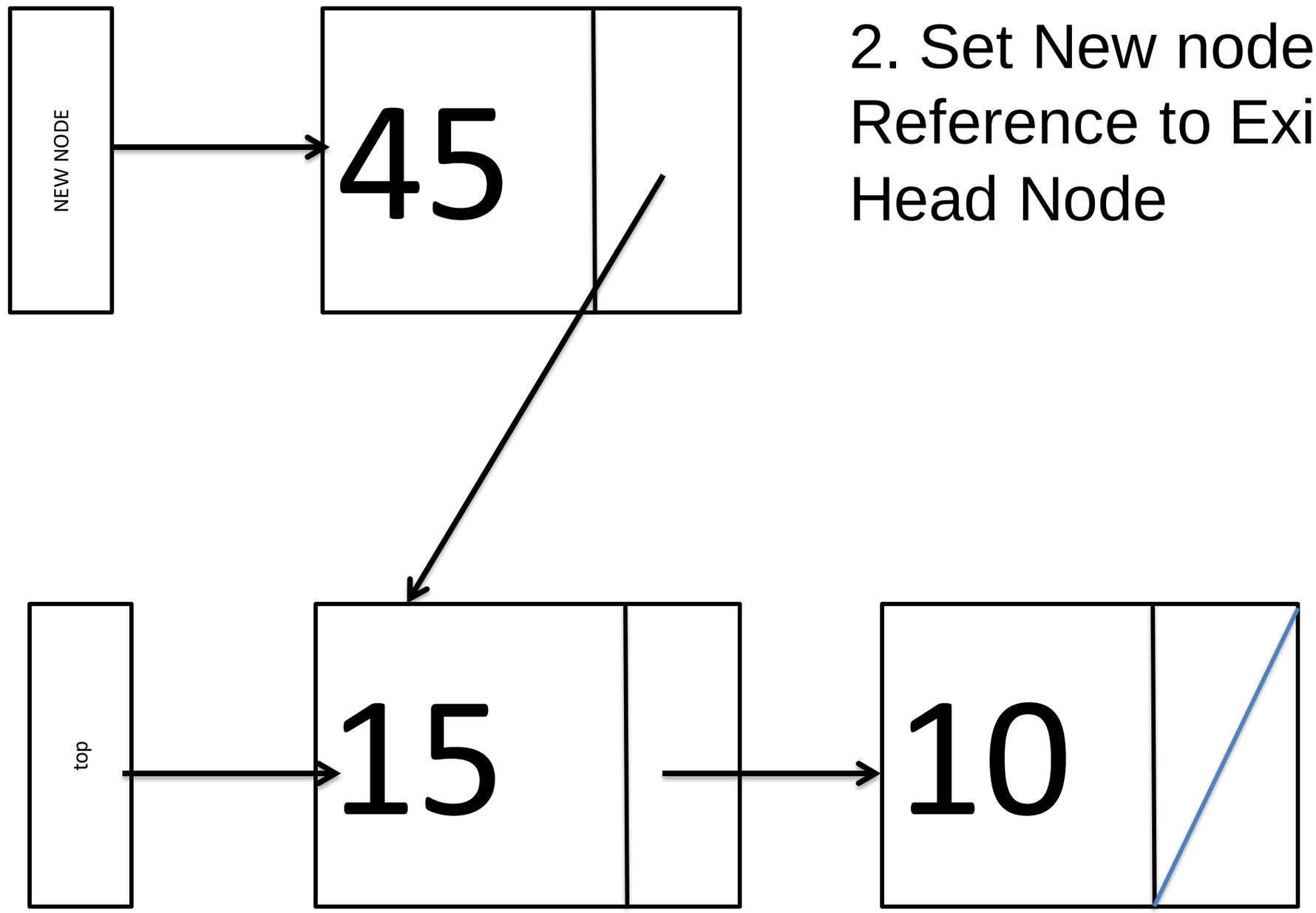
Pushing an element



1. Create the new node

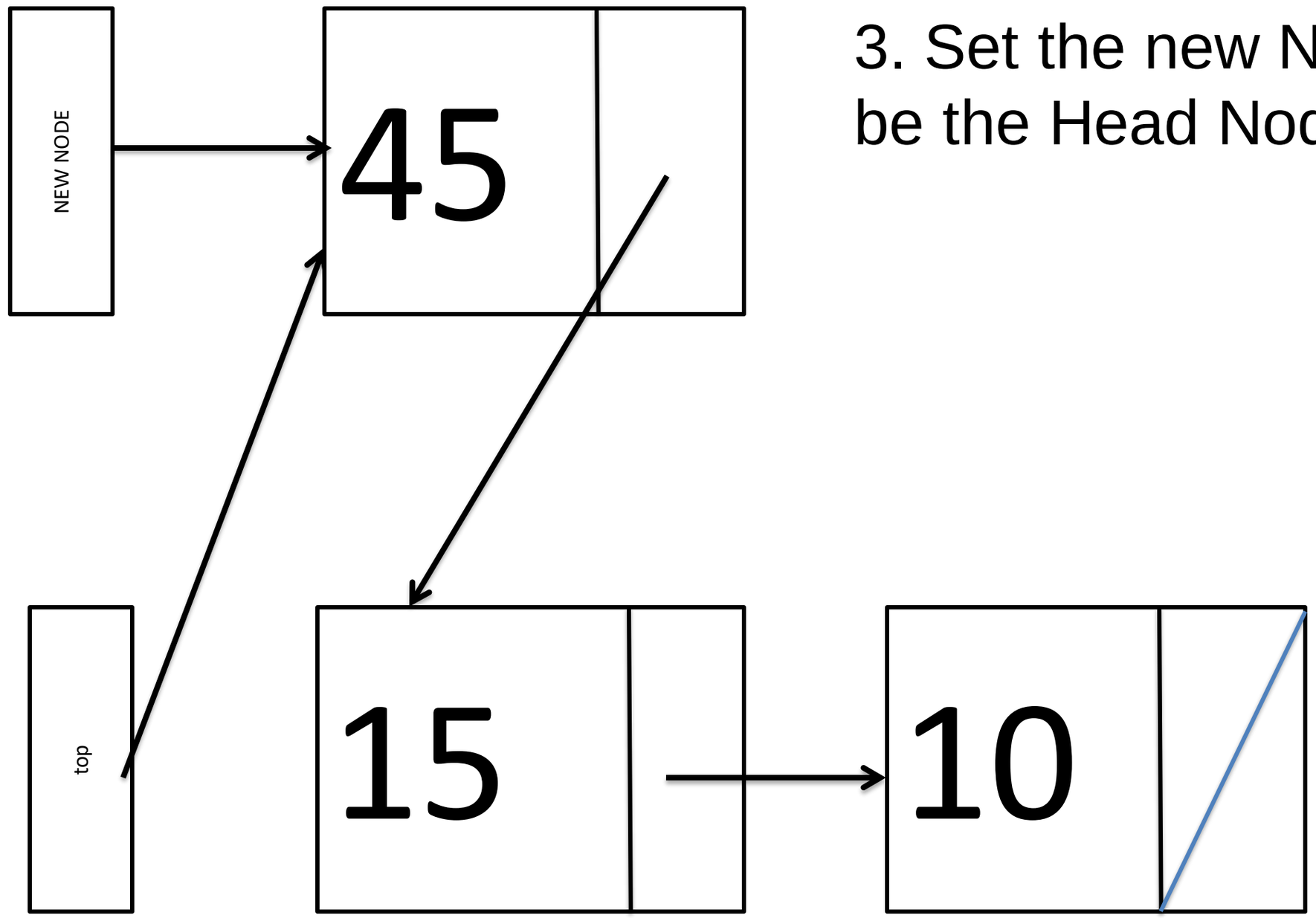


Pushing an element



2. Set New node Reference to Existing Head Node

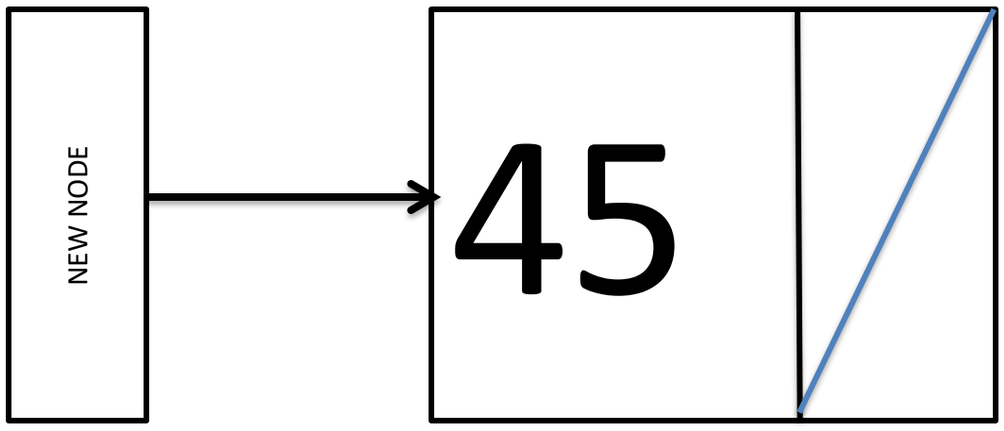
Pushing an element



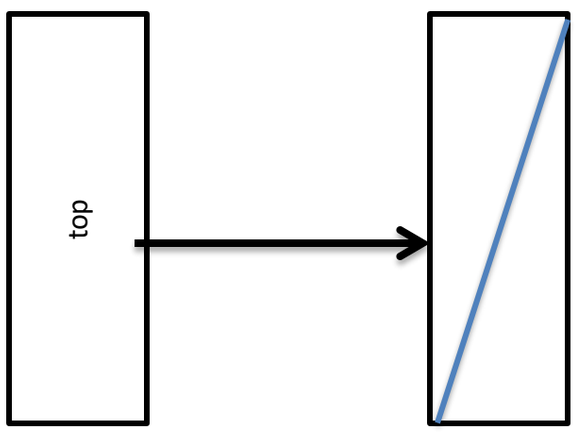
3. Set the new Node to be the Head Node

What if the stack is Empty?

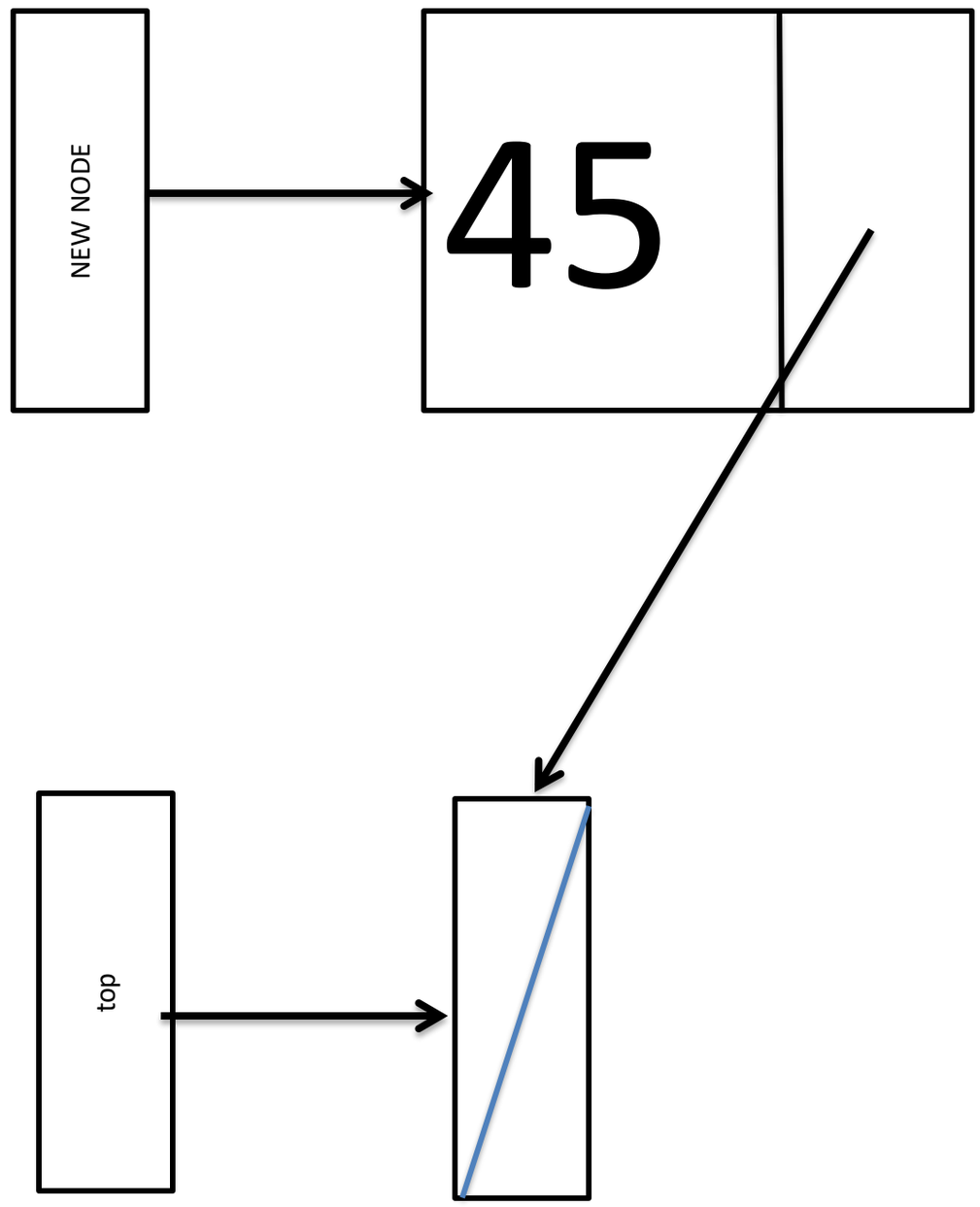
Pushing an element (Empty stack)



1. Create the new node

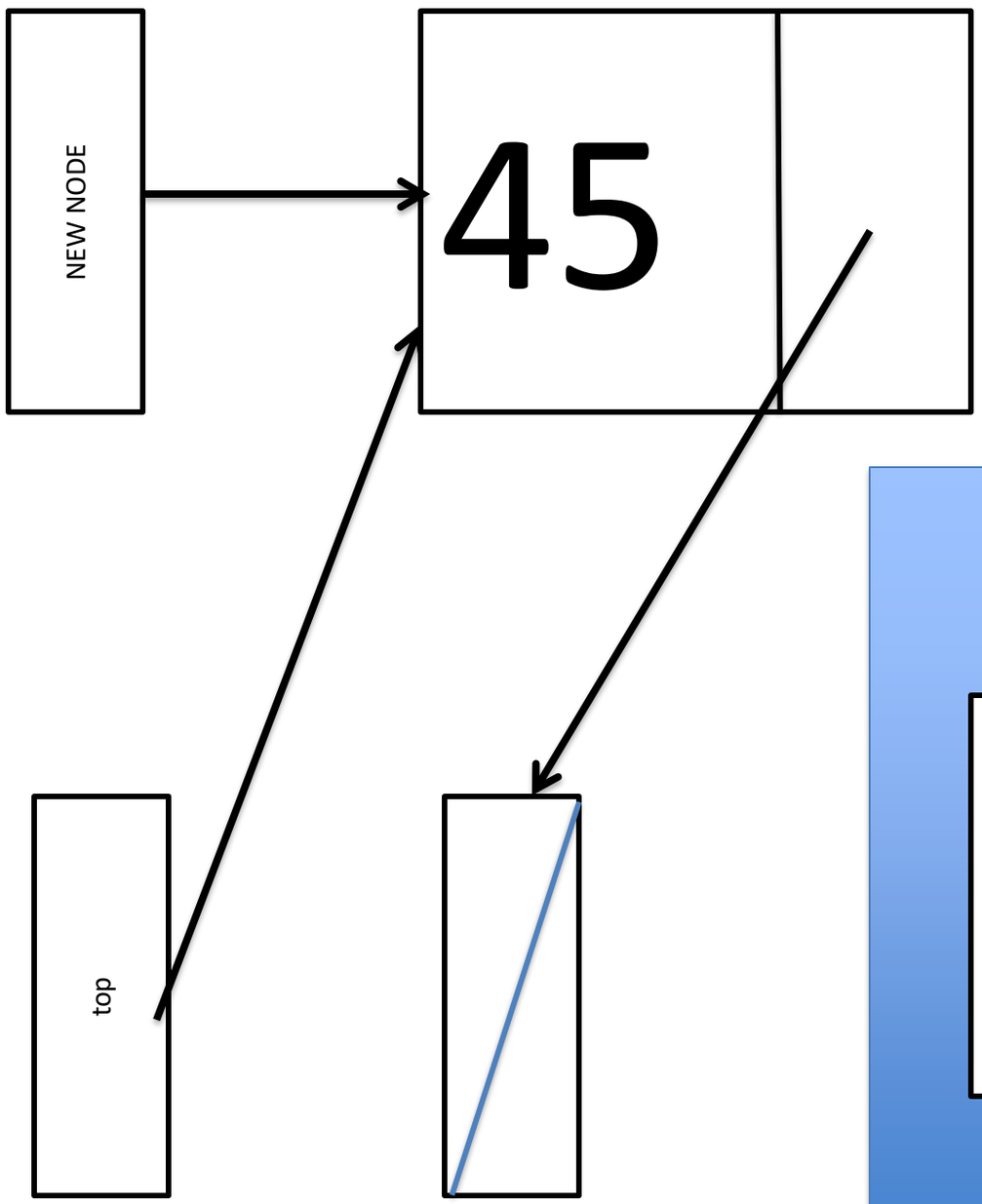


Pushing an element



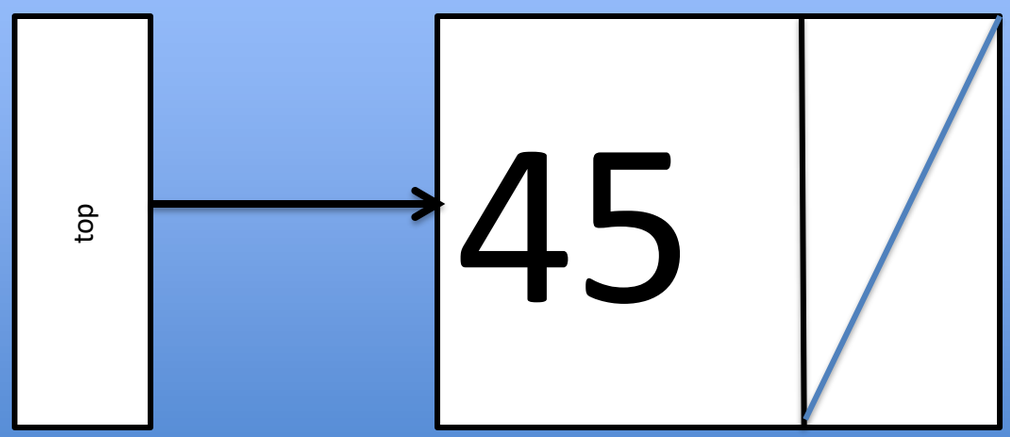
2. Set New node Reference to Existing Head Node

Pushing an element



3. Set the new Node to be the Head Node

Which is the same as:



Popping an element

- Again, this should be constant time
- Q: should we remove the first or last element?

www.socrative.com / `student login' / room: SD32018

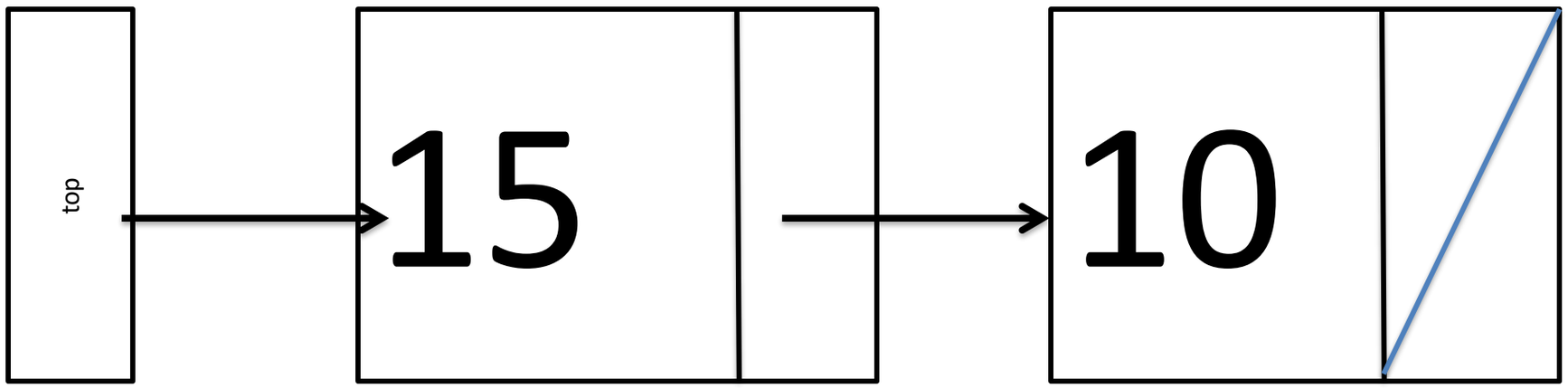
Popping an element

- Solution
  - Remove the first
- Why?
  - Firstly, as we are adding to the first we also have to remove it
    - If not this will not be a stack!
  - As with push, removing the last element will be require that we traverse the list
    - This operation is linear

- Solution:
  - Check that the stack is not empty
    - Can’t remove something if it isn’t there
    - In this case raise an **StackException**
  - Get the element of the current top list
  - Make the top’s next Node the new top node
  - Return the element

Popping

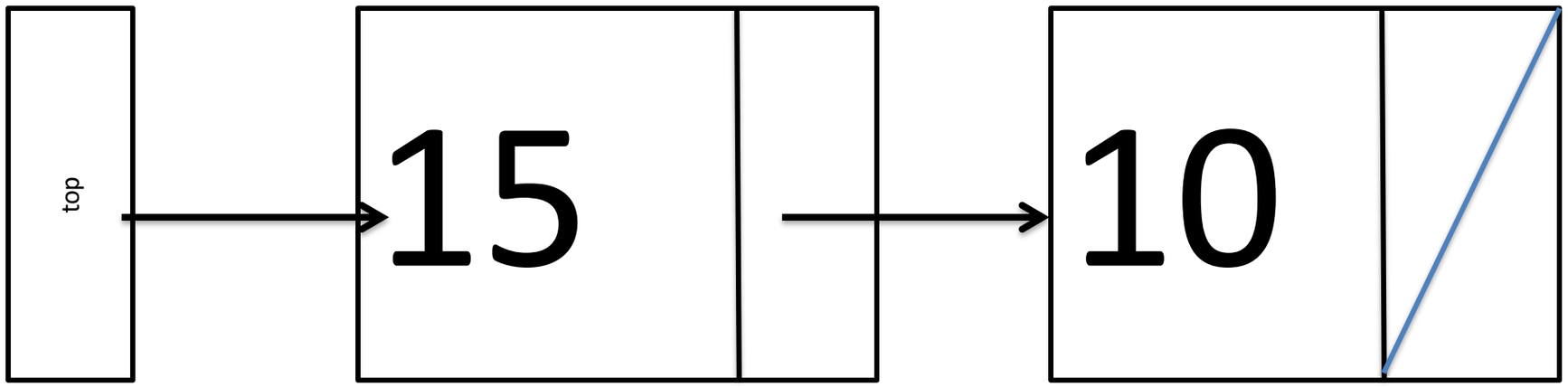
1. Check the List isn’t Null



HEAD == NULL throws a **StackException**

Popping

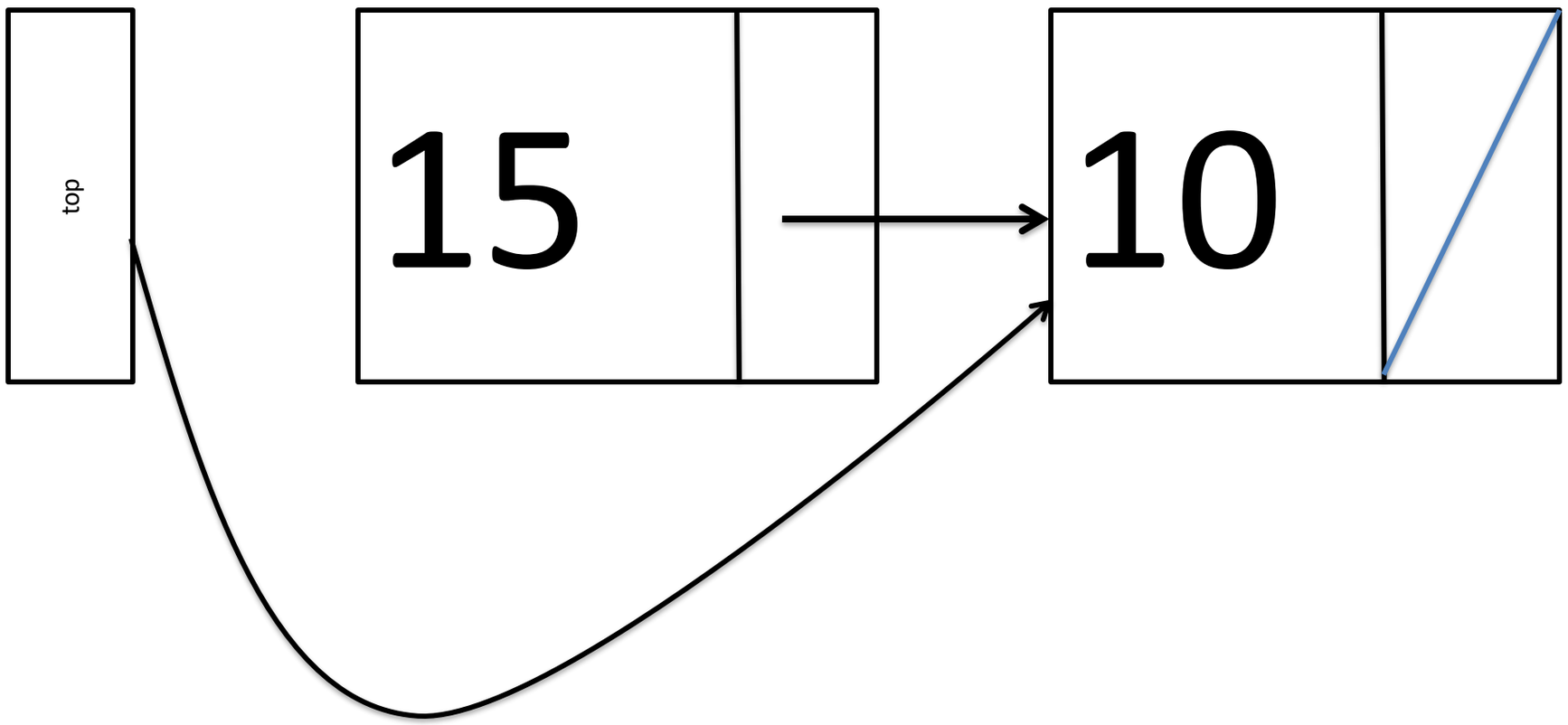
2. Get the first element



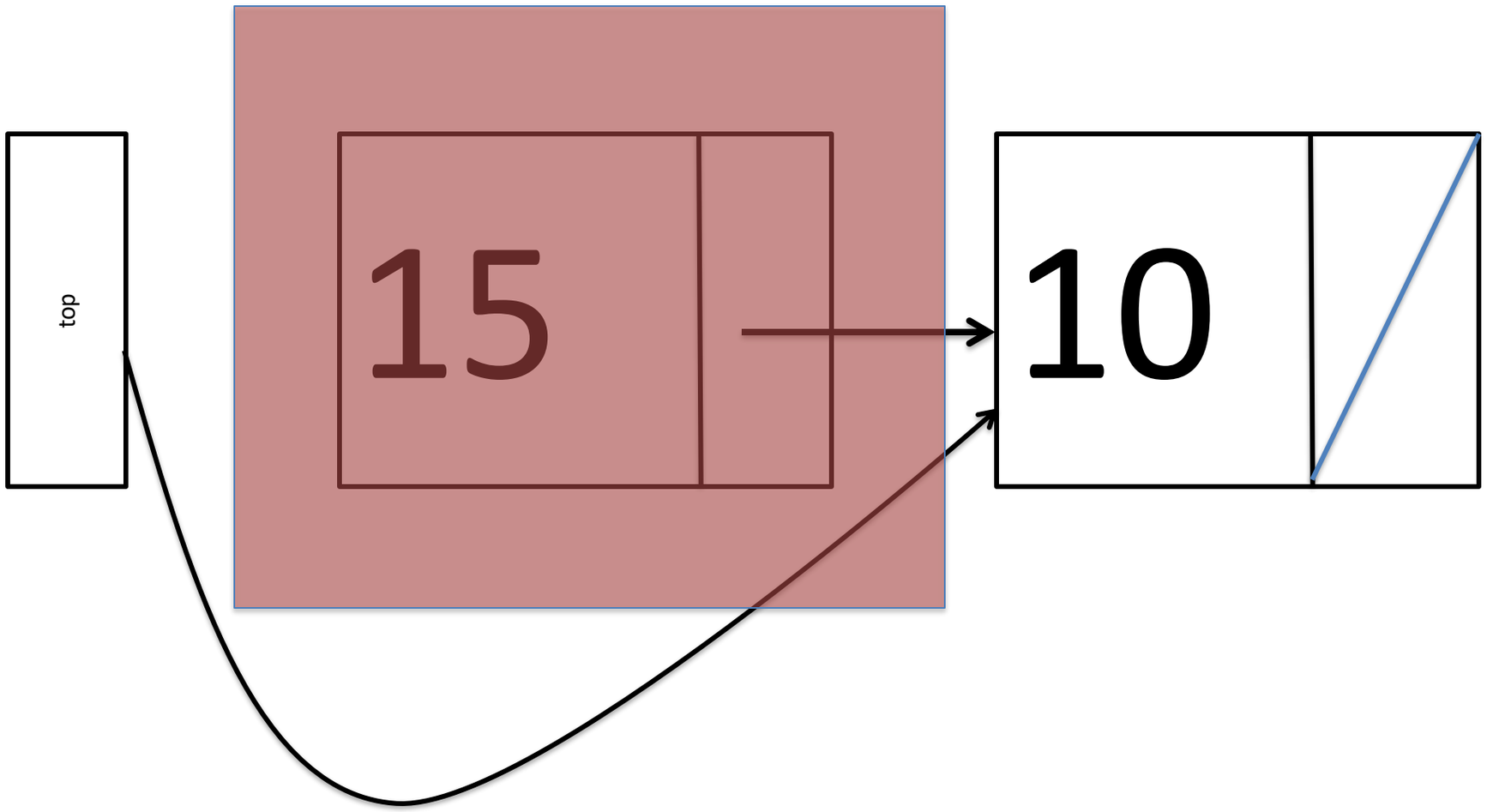
tmp = 15;

Popping

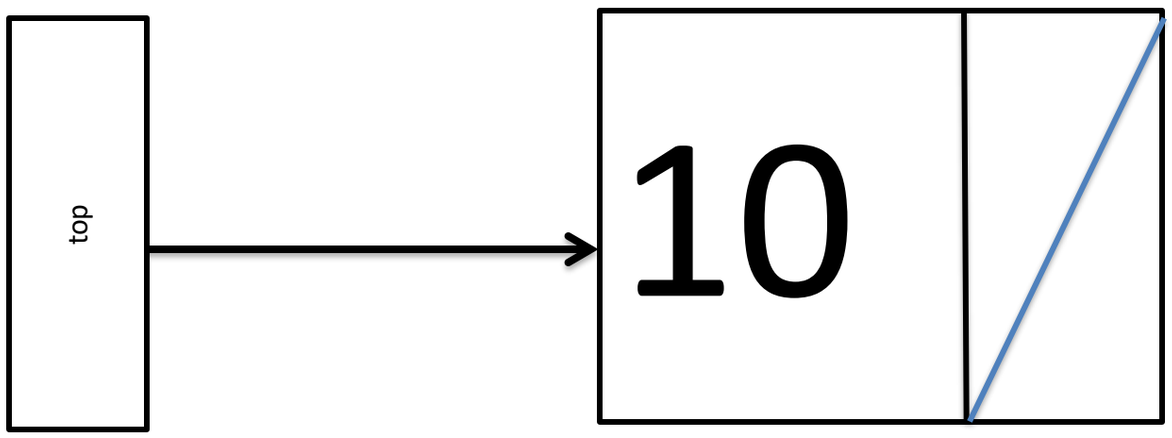
3. Set top to the next element of top



Popping



Popping



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

The **top** operation

- Here we only return the element of top
- This is trivially constant time

The **isEmpty** operation

- When is the stack empty?
- Solution
  - When **top == null**
- Why is this O(1)

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

Summary

- Representing a stack as a linked list
- Advantage over array representation is:
  - it can grow dynamically
  - while all operations are constant time O(1)
- You will implement today's pictures in Java (lab 4)
- Attendance sheet
- Next lecture: **Queues**