Topic 9.0: Linked Lists

Learning Goals:

- Create and manipulate LinkedLists
- Compare usage scenarios for Lists & Arrays
- Recursively traverse LinkedLists
- Draw Memory/Reference Diagrams for LinkedLists
- Explain the difference in running times and storage of lists & arrays

Lists

- We’ve seen collections of data like **Arrays**
- We have seen **ArrayLists**
    - A combination of Arrays and Lists
- **What are lists?**

Lists

- We've seen collections of data like **Arrays**
- We have seen **ArrayLists**
  - A combination of Arrays and Lists
- **What are lists?**

- General definition is a sequence of data items where each has a position

- Examples:
  - List of 5 Strings: "Hello", "World", "Computers", "are", "Cool"
  - List of 3 ints: 1, 2, 3
  - List of 0 doubles:

Notice we said has a **position,** not "**index-able**"

There is a difference

Lists

- We’ve seen collections of data

List Usage

- We’ve used lists in different ways
    - Full Arrays
    - Partially-full Arrays
    - ArrayLists (Basically “Java Managed” Partially-full Array

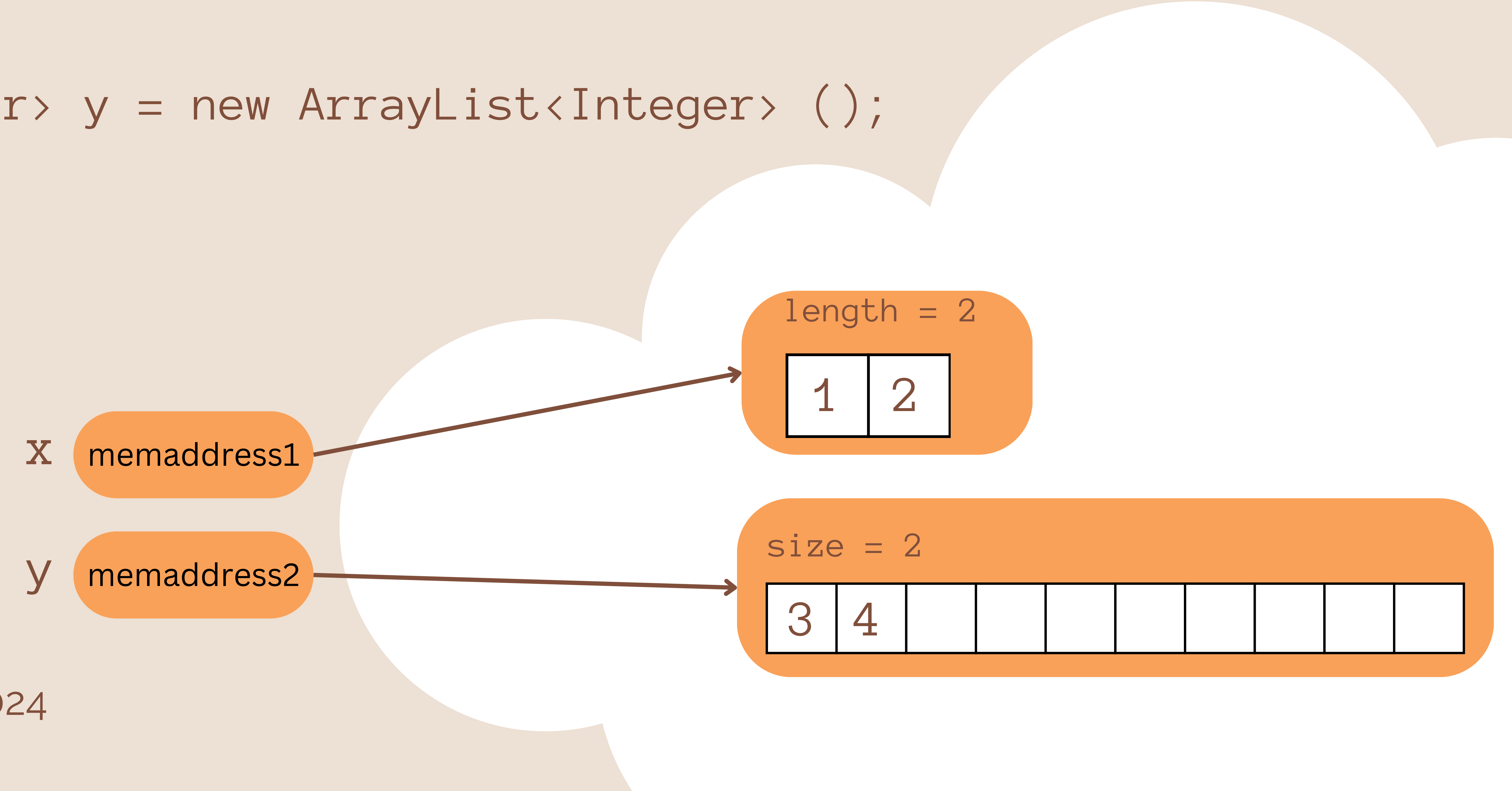
List Assumptions

- Normally when we think about **Lists** in coding we assume a few things
  - CRUD
  - **Create**: we can create new lists!
  - **Read**: we can get data out of the list (including the number of elements)
  - **Update**: we can set data in the list
  - **Delete**: we can remove items from the list (or maybe even empty the whole list)
- **Behaviours** like get/set/remove/etc can be (*hopefully*) done by index or by element

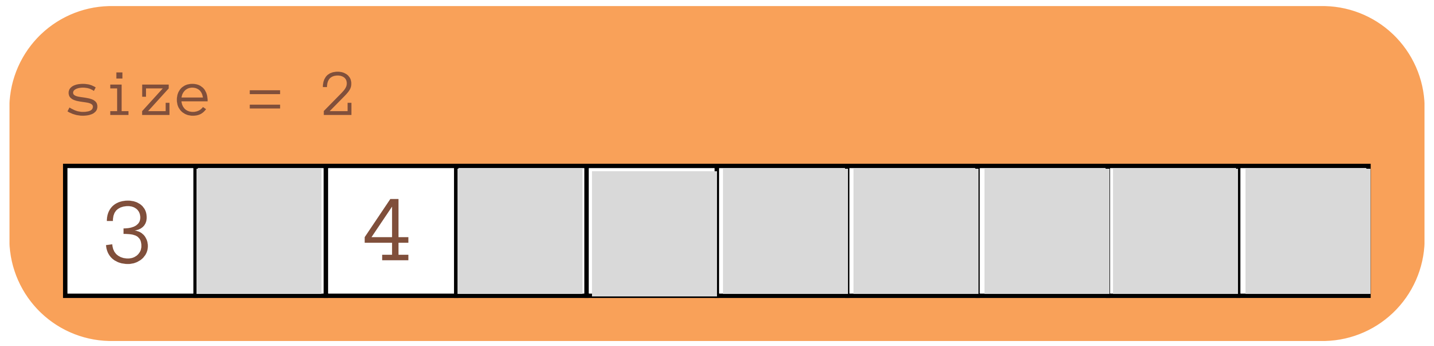
Physical Adjacency

- Arrays/ArrayLists
  - Stored in Contiguous/Continuous Memory

```
int[] x = {1,2};
ArrayList<Integer> y = new ArrayList<Integer> ();
y.add(3);
y.add(4);
```



Physical Adjacency



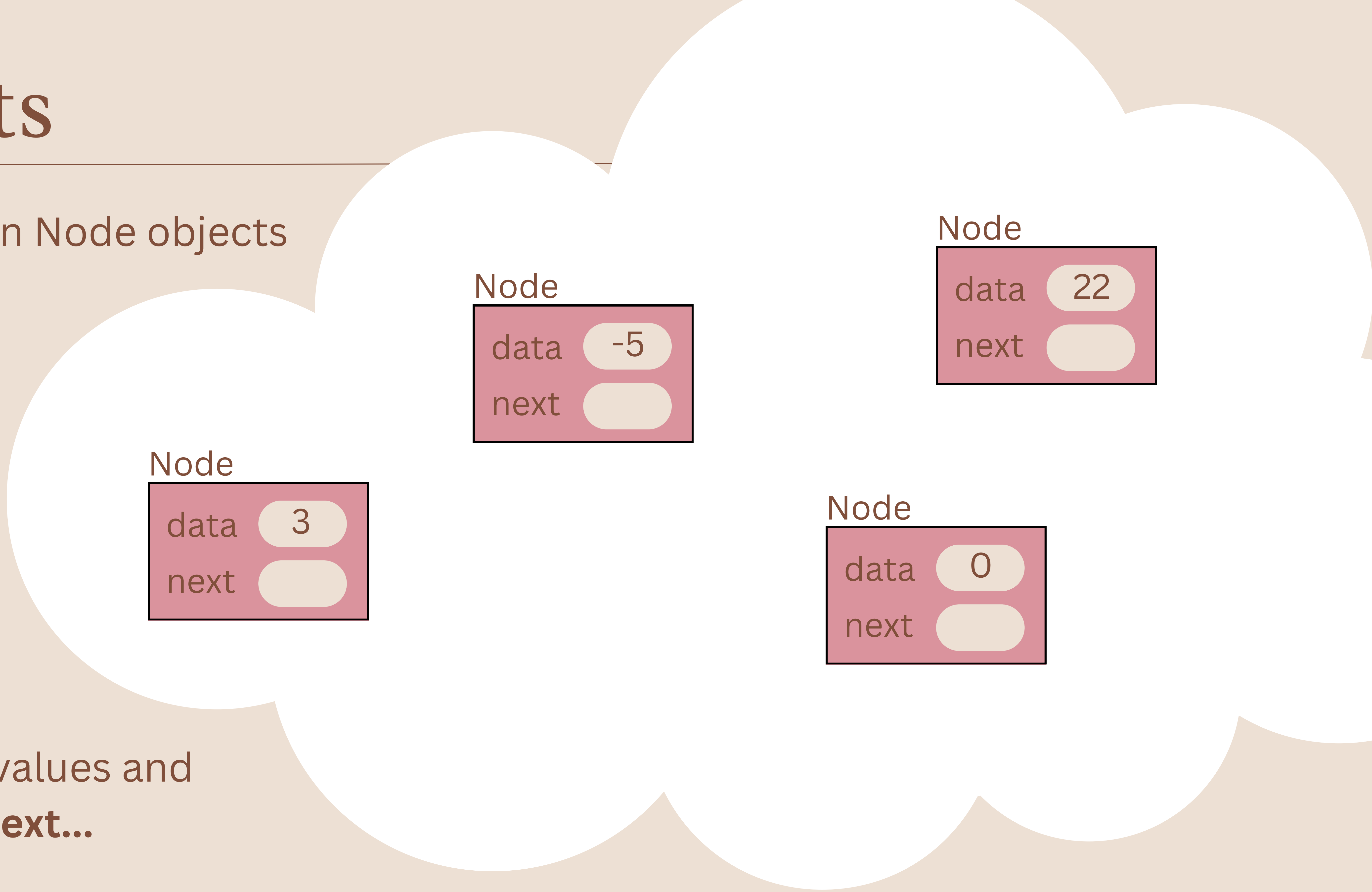
- Not stored with “gaps”
- To **add/delete** an element from the middle or the front requires other elements to be **shifted**
- The Array/ArrayList might become full (or be full all the time)
- To add another element (when full) requires a **complete re-build of the array into a new one**

A new kind of List (the Linked Kind)

- A **LinkedList** is a specific type of a list (there exists others) that solves the shifting AND rebuilding
    - but creates its own problems too

LinkedLists

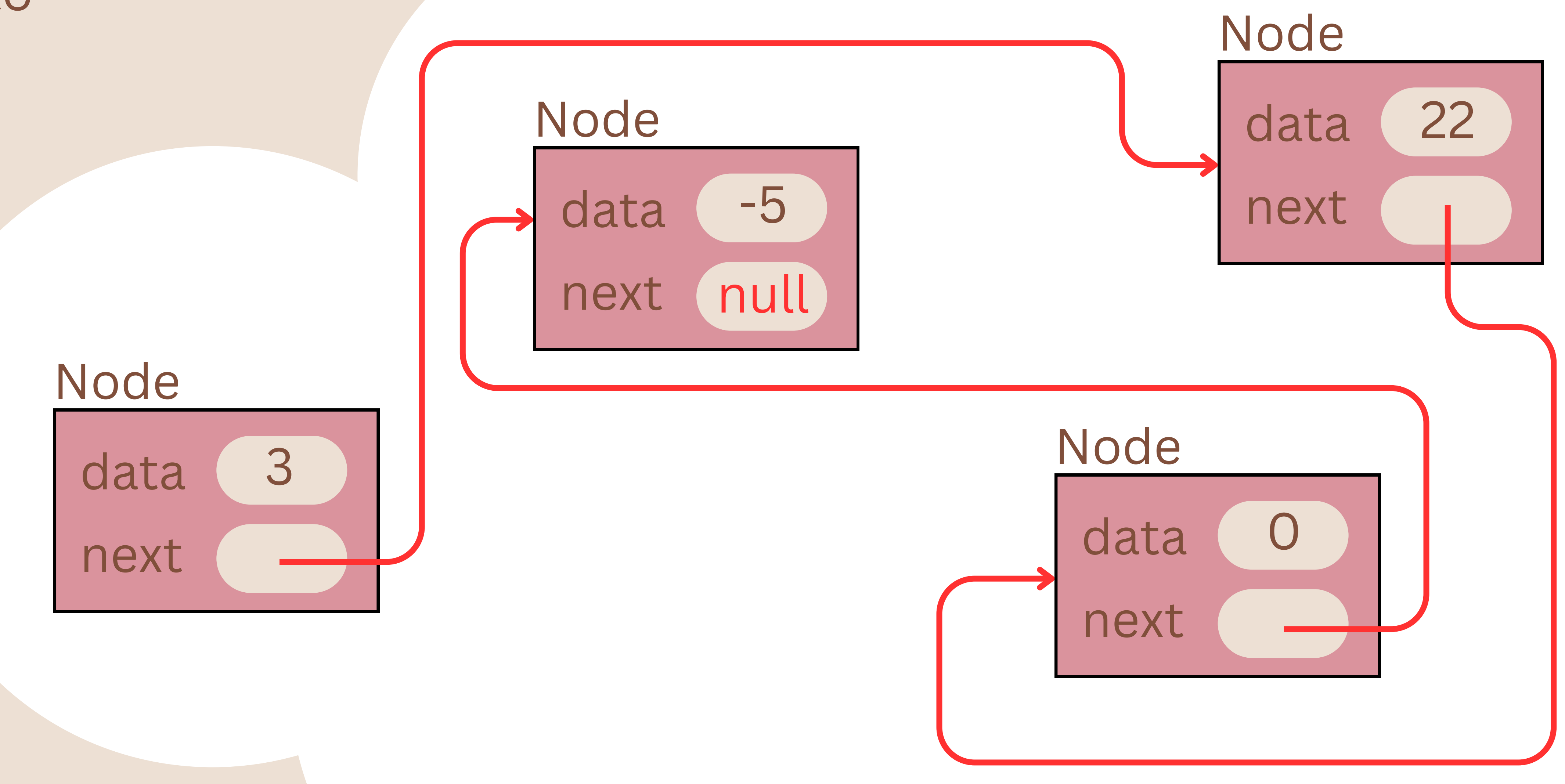
- We store our data in Node objects



- We store our data values and
- something called **next...**

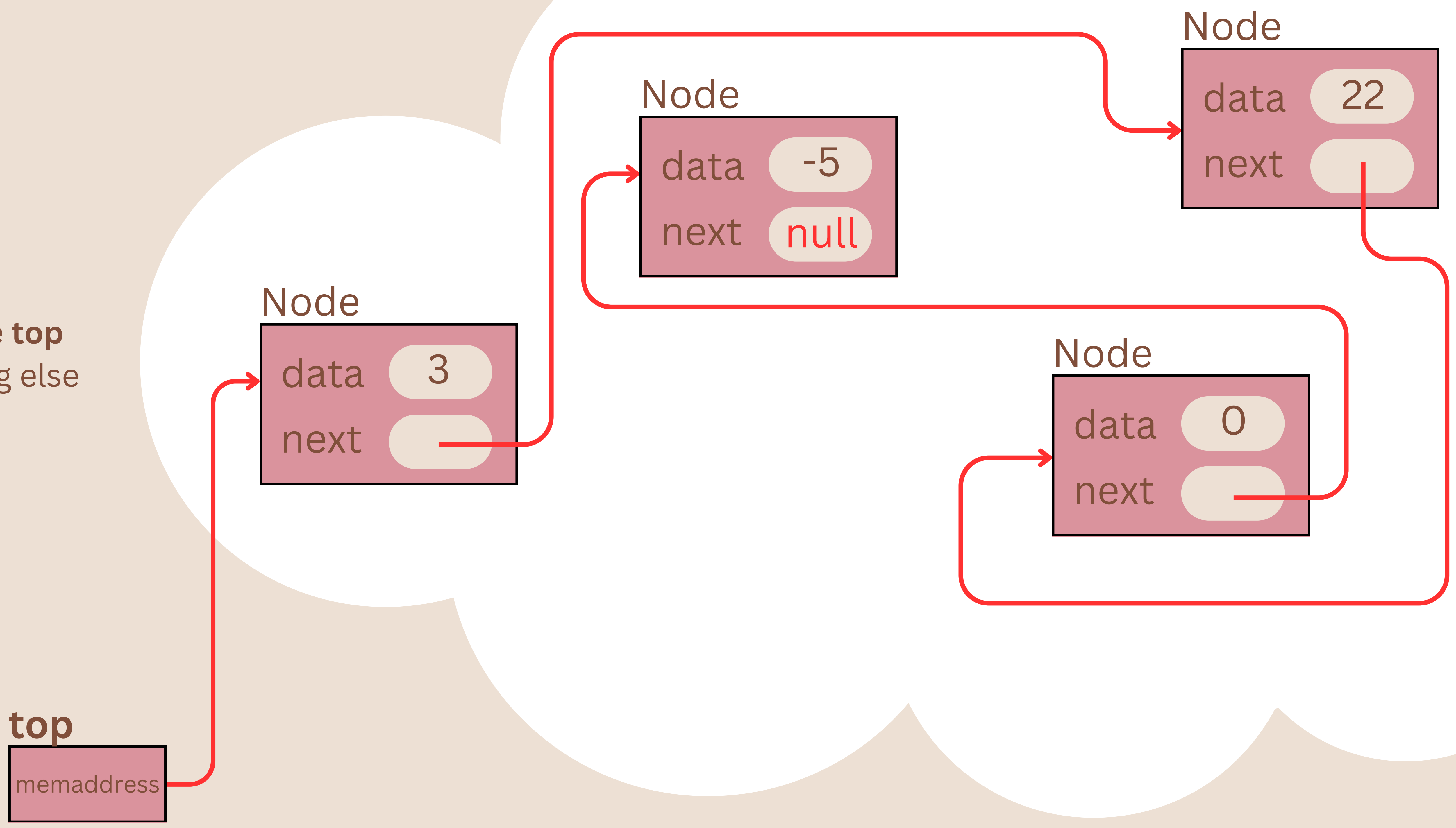
LinkedLists

- **next** contains the **reference** to the next **Node** in the list
- (or null if there isn't one)



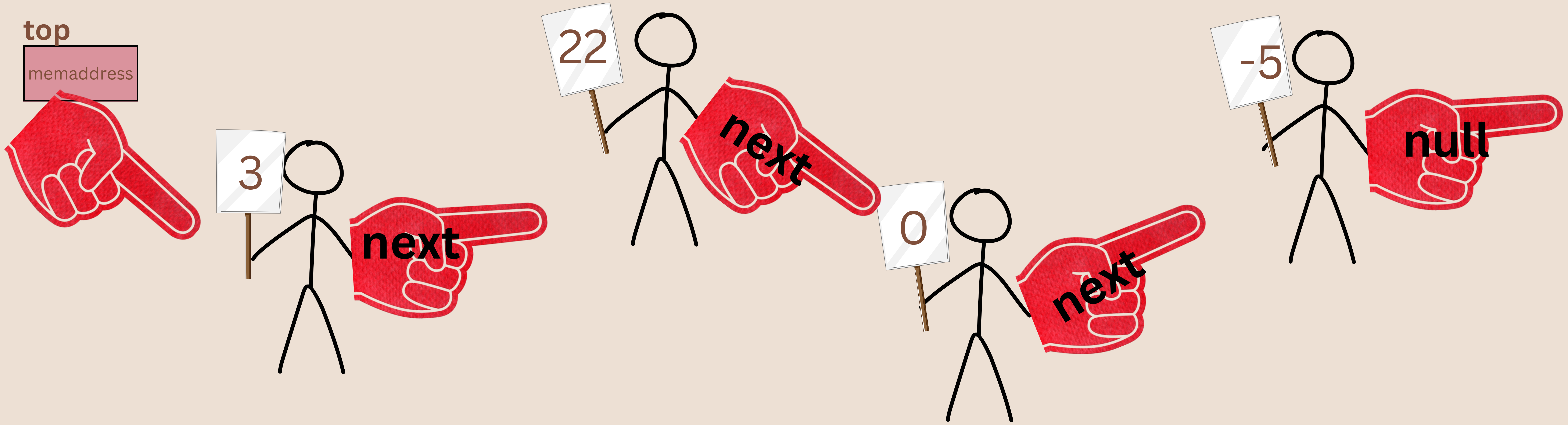
LinkedLists

- **next** contains the **reference** to the next **Node** in the list
- (or null if there isn't one)
- We only keep track of the **top node** and then everything else follows



LinkedLists

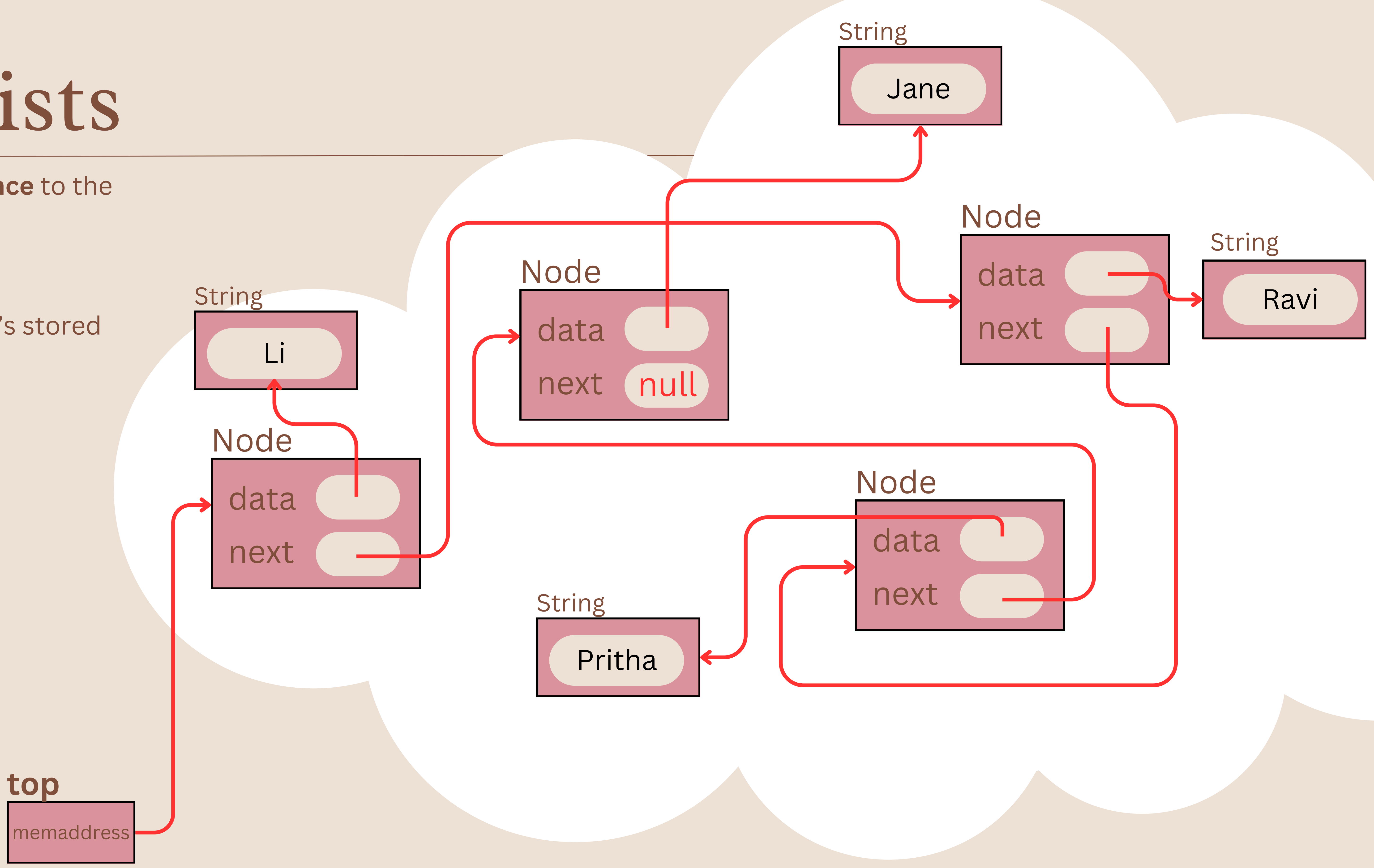
- **next** contains the **reference** to the next **Node** in the list
- (or null if there isn't one)
- It's a bit like a line of people with signs of information and big pointy fingers telling you which sign to read next



LinkedLists

- **next** contains the **reference** to the next **Node** in the list

- If the data is an **object**, it's stored as a **reference** too



LinkedList Creation

- Let’s start with a LinkedList of big ‘O’ Objects (the Java Object class) so that it can hold anything under the Object umbrella
    - (just like our default ArrayList<> could do)
- We need 2 classes
    - LinkedList & Node
    - This is a special scenario, we only want to have access to Nodes **inside** the LinkedList class (not main EVER)
    - Node is kind of like a special friend of LinkedList
    - If we define the Node class inside the LinkedList class, we don’t need getters/setters for the Node variables **and** this stays private from the main class
- See the LinkedList.java file for an example

Adding to a LinkedList

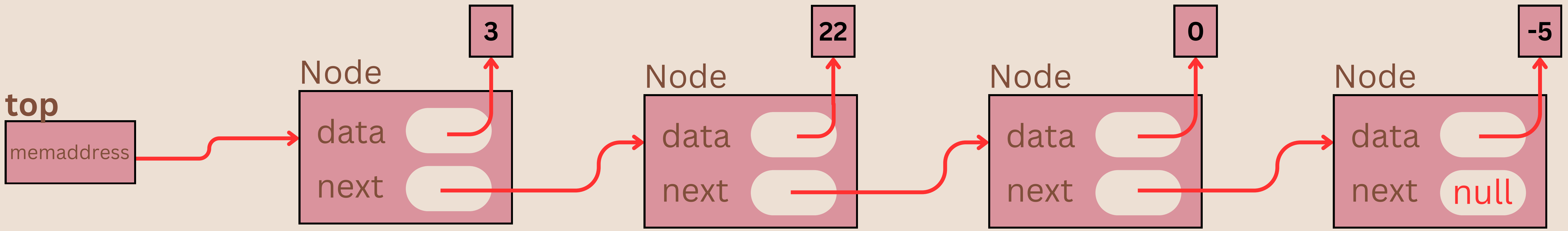
- If we can directly access our node data/next instance variables, we can get/set the values ***reasonably*** easy.
- What about adding new data to the list?
  - It's much easier to add new elements at the beginning
  - **Arrays** it is easier to add to the end (no shifting)

```
public void add(Object newItem) {
   Node newNode = new Node(newItem,top);
   top = newNode;
}
```

- What is happening in the list?

Adding to a LinkedList

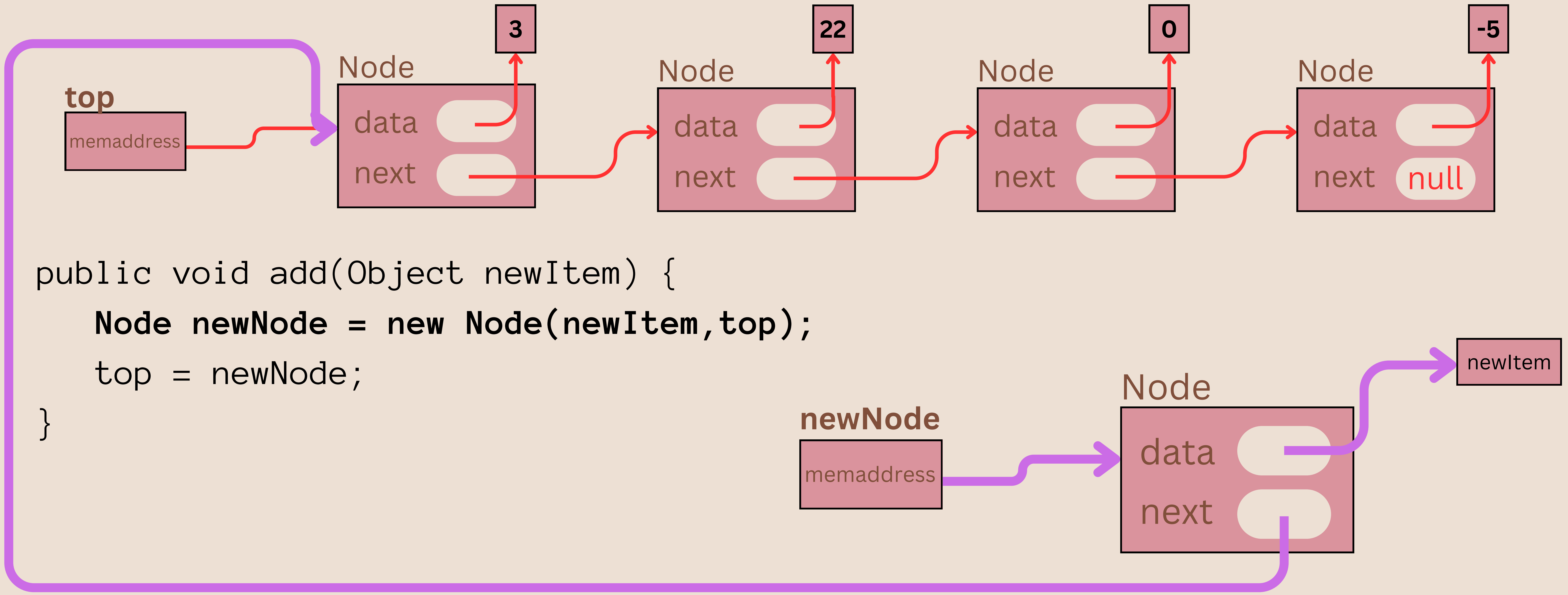
- For ease, let’s assume the data is Integers and the visual looks like this:



```
public void add(Object newItem) {
   Node newNode = new Node(newItem,top);
   top = newNode;
}
```

Adding to a LinkedList

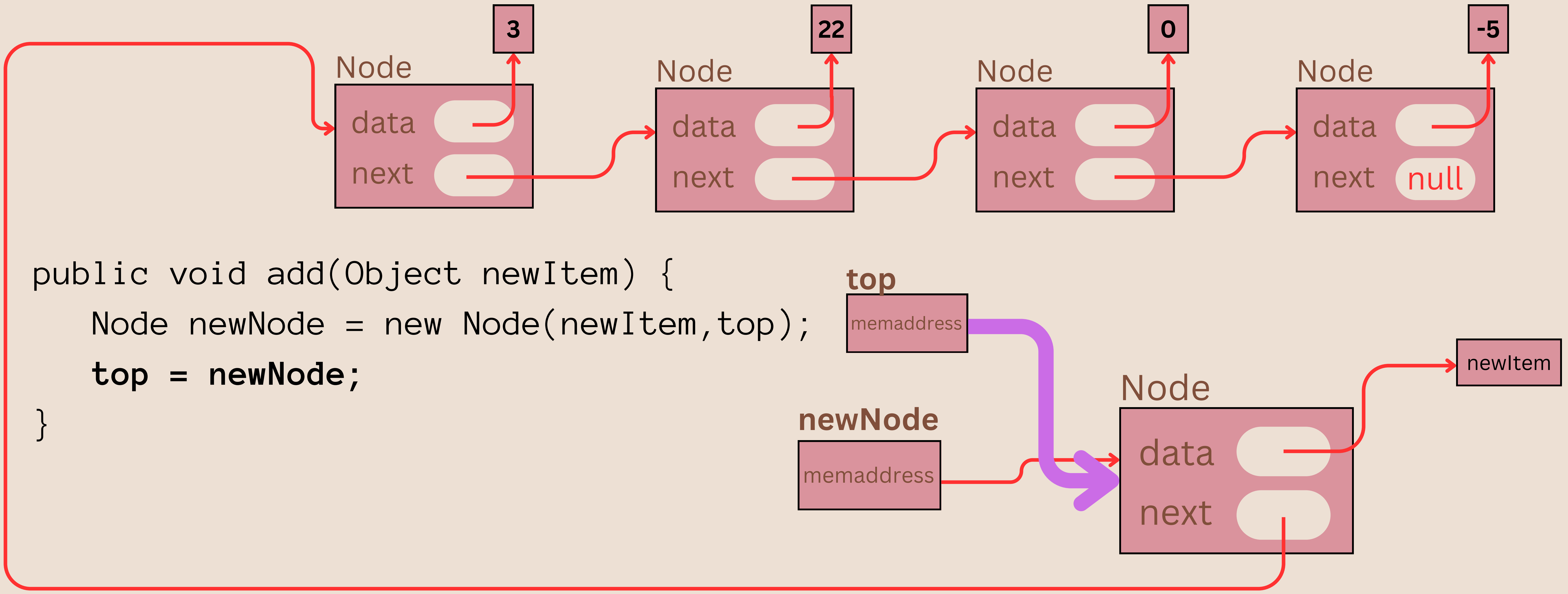
- For ease, let’s assume the data is Integers and the visual looks like this:



```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

Adding to a LinkedList

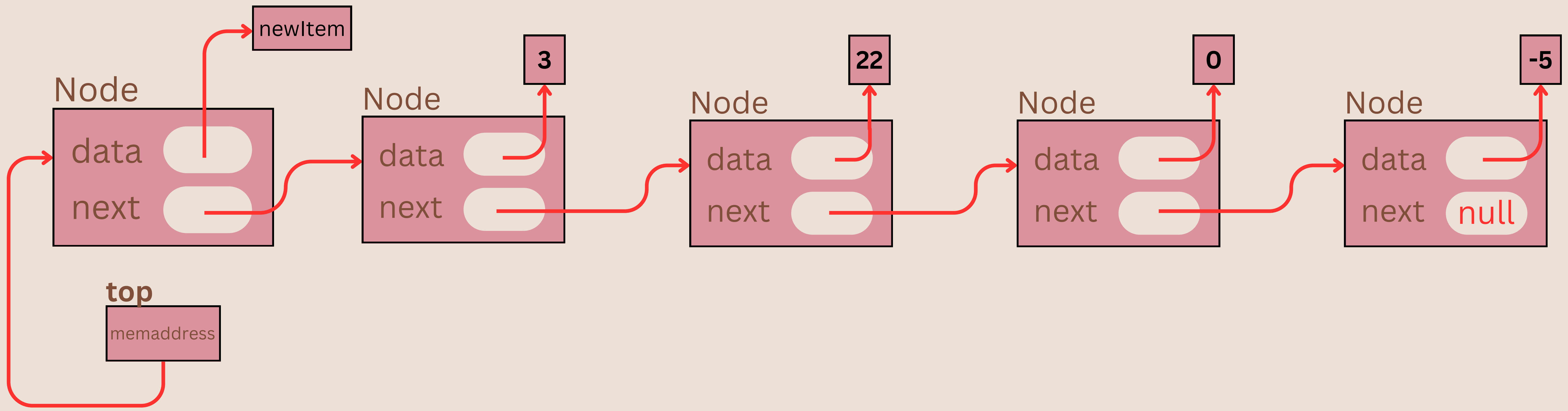
- For ease, let's assume the data is Integers and the visual looks like this:



```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```

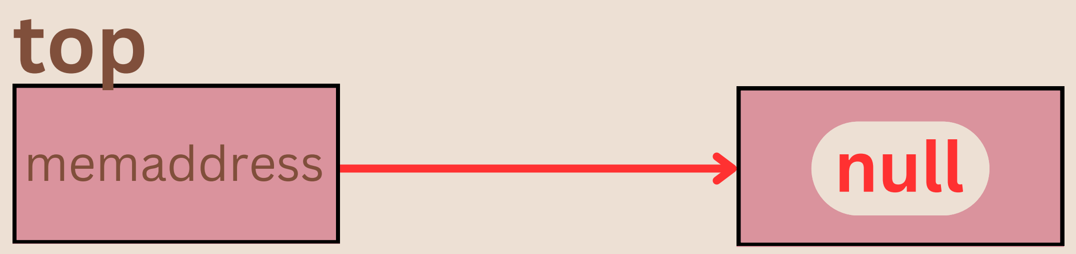
Adding to a LinkedList

- For ease, let’s assume the data is Integers and the visual looks like this:



Adding to a LinkedList

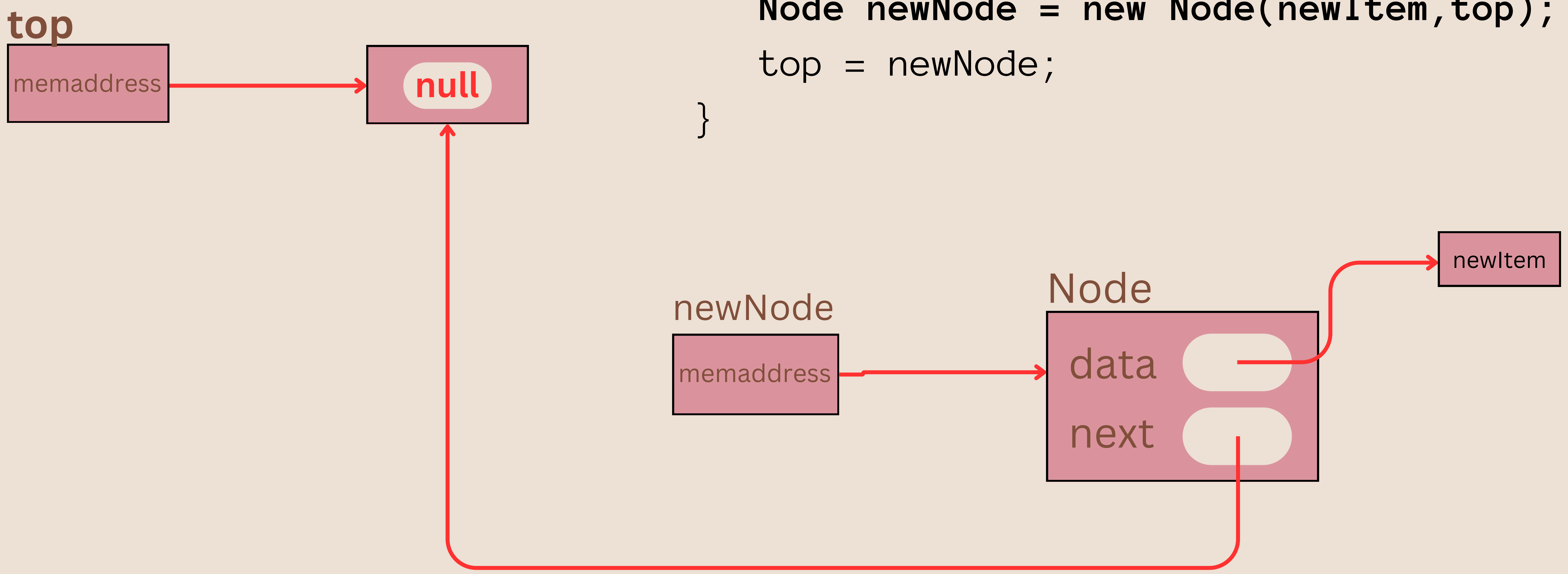
- What if the list is empty?



Adding to a LinkedList

- What if the list is empty?

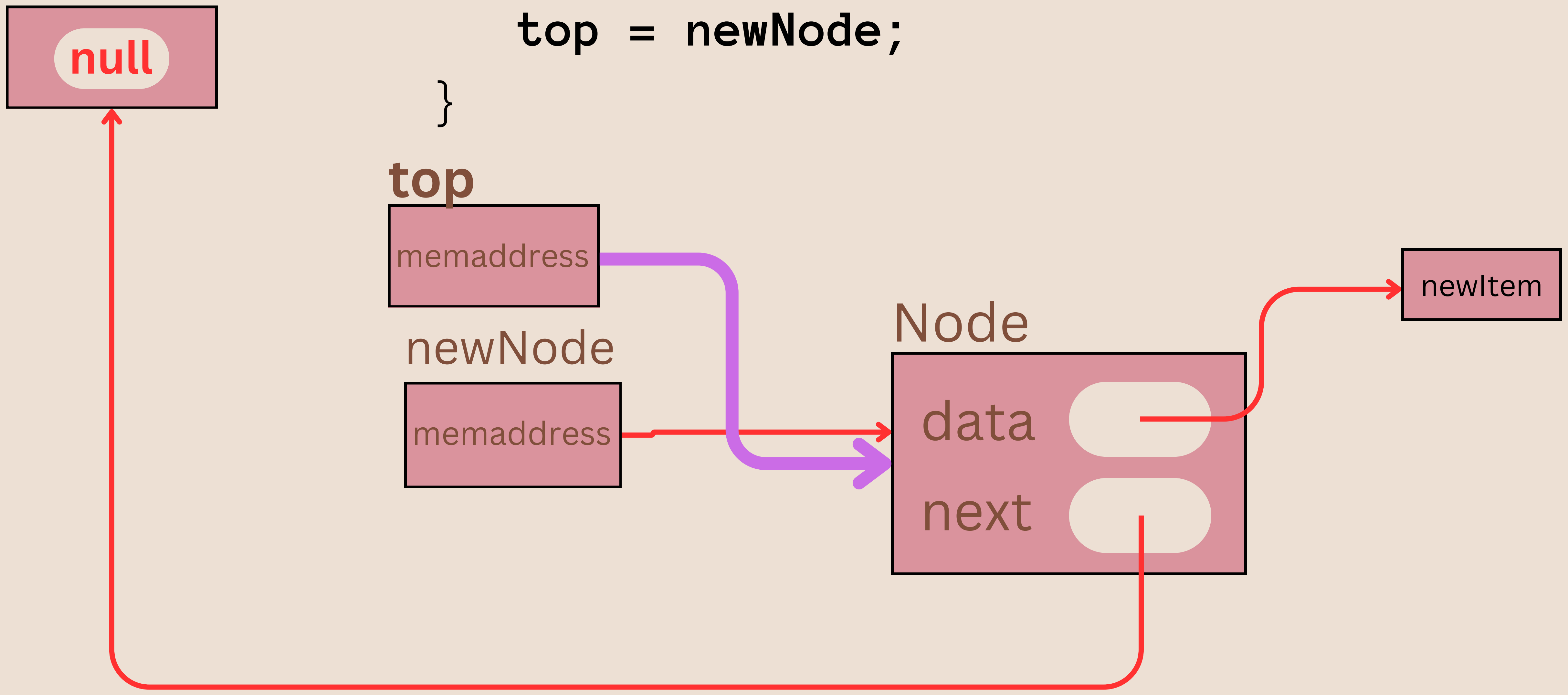
```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```



Adding to a LinkedList

- What if the list is empty?

```
public void add(Object newItem) {
    Node newNode = new Node(newItem,top);
    top = newNode;
}
```



toString in a LinkedList

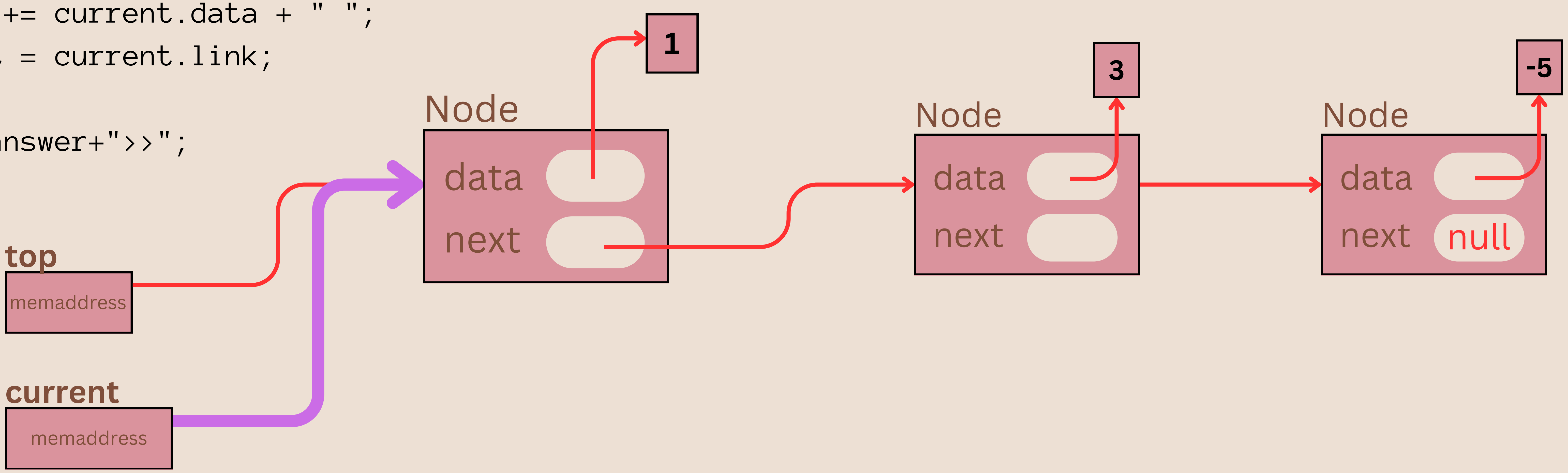
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<<



toString in a LinkedList

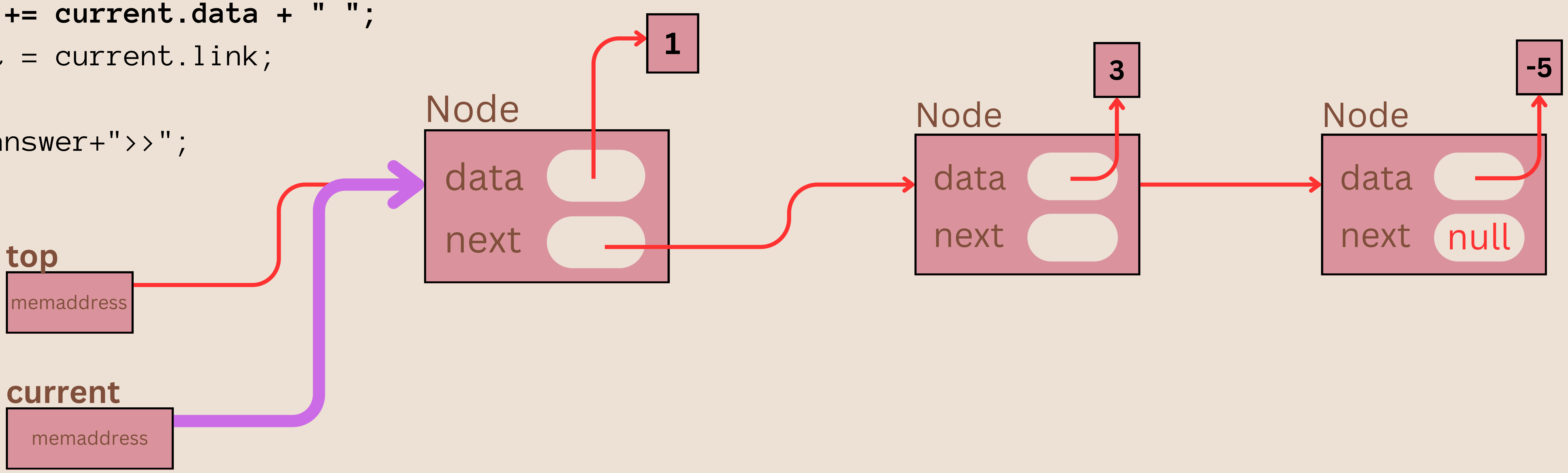
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1



© Lauren Himbeault 2024

toString in a LinkedList

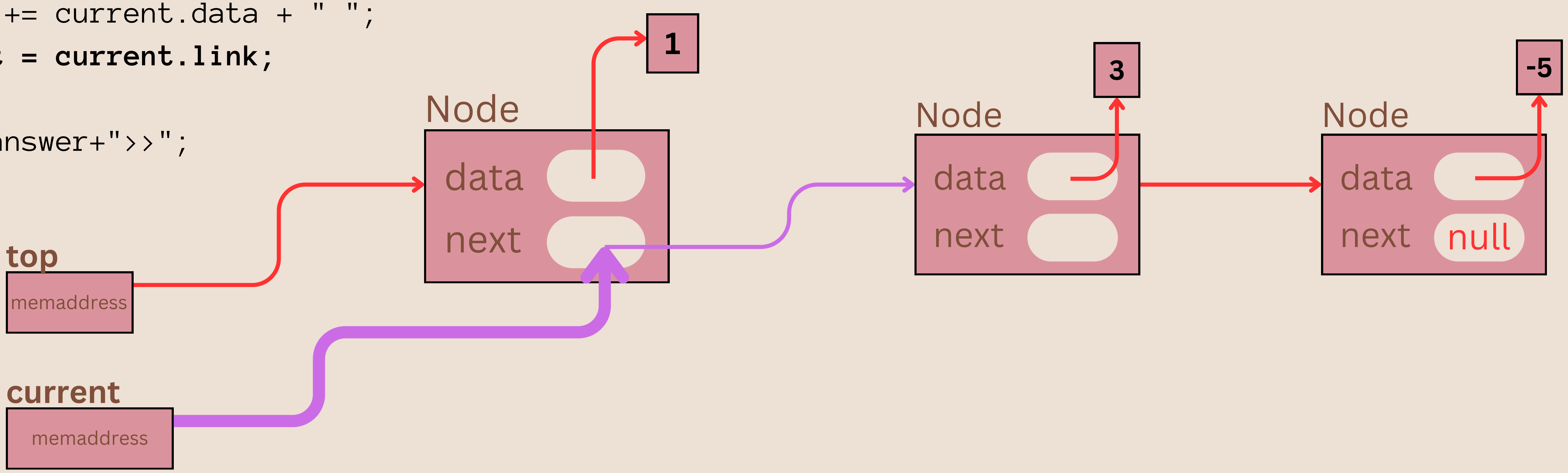
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1



toString in a LinkedList

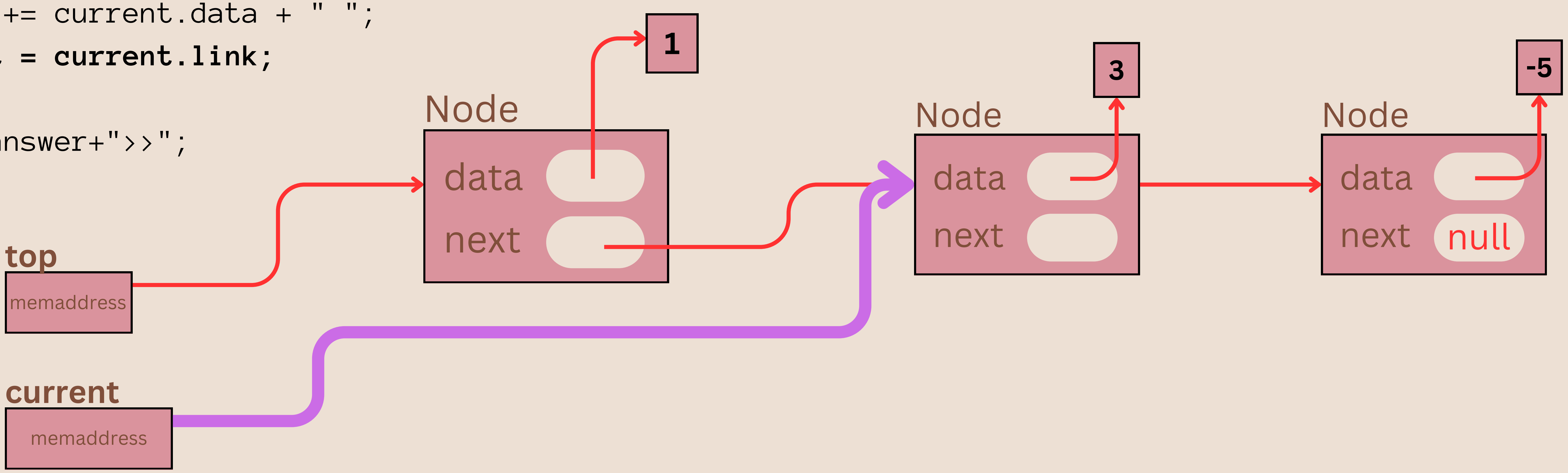
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1



toString in a LinkedList

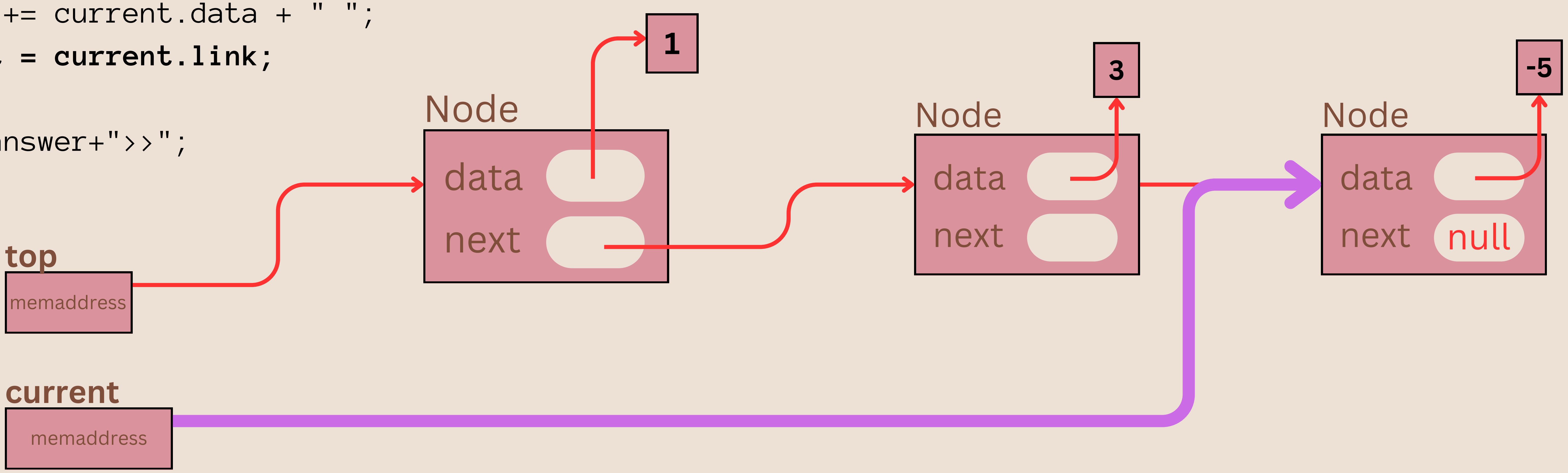
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1 3



toString in a LinkedList

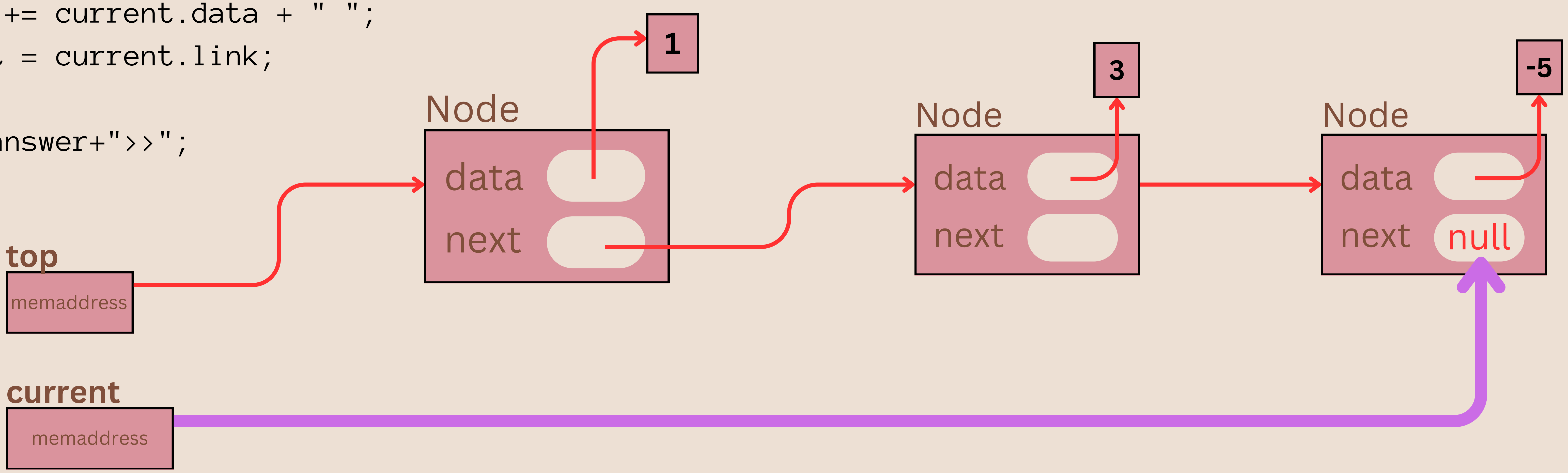
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1 3 -5



toString in a LinkedList

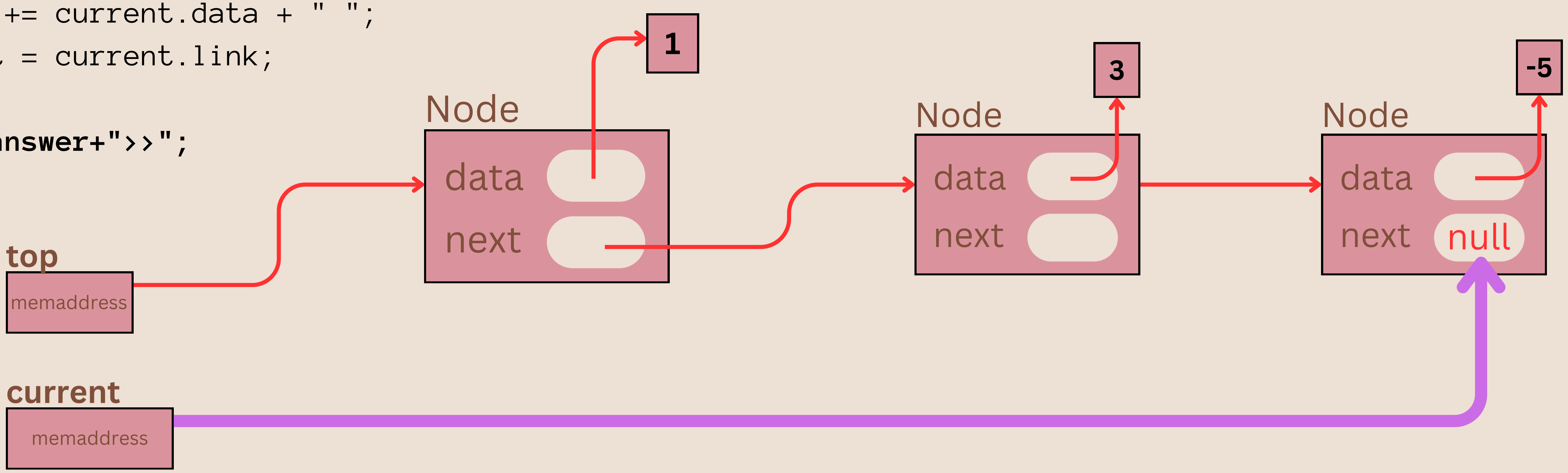
- This is typical of any method that has to traverse the list (go through all elements)

```
public String toString() {
   String answer = "<< ";
   Node current = top;

   while(current != null) {
    answer += current.data + " ";
    current = current.link;
   }
   return answer+">>";
}
```

Answer

<< 1 3 -5 >>



Pause & Practice: addToPostion (with me)

- **Try drawing/coding each of the following methods**
- **After trying it yourself:**
    - **continue to follow along in the video to see each of these sketched out/implemented**

- **addToPosition(int index, Object o)**
    - add the object ***o*** into an existing linked list at position ***index***
    - Don’t forget to check:
        - index > 0
        - index < number of objects in the list
    - consider a **private helper method Node getNode(int posn)**

Adding: Array vs LinkedList

- When adding to the middle of a list:
    - **Array**:
        - Must "shuffle" all elements after that (Loop needed)
        - Can access the desired position directly
        - Might get full and require expansion of the array
    - **Linked list:**
        - No need to shuffle anything. Very quick insertion.
        - Must follow the chain through all previous elements to find the right position (Loop needed)
        - Can't get full (Unless you completely run out of memory, which is unlikely)
    -

Pause & Practice: delete (with me)

- **remove(Object o) // remove by element, not index**
    - remove the object ***o*** from an existing linked list
    - Don’t forget to checkif the object:
        - is the last element
        - is a middle element
        - is the first element
        - is not in the list at all

Pause & Practice: delete (with me)

- Why does `while(current != null && !current.data.equals(key)) {` work?
- **Lazy Boolean Evaluation**
  - Since the two conditions on both sides of the && need to be true for the resulting expression to be true
  - programming languages only evaluate the second argument **if the first was not enough to determine the value of the expression.**
  - In this case, when the first argument is false, we know the expression will be false.
    - Therefore the second is not evaluated and no NullPointerException occurs

Pause & Practice: delete (with me)

- **remove(int index) // remove by index this time**
  - remove the object at **index** from an existing linked list
  - Don’t forget to checkif the object:
    - is the last element
    - is a middle element
    - is the first element
    - is not in the list at all
- Hint: Use your getNode method from before, it may help :)