CSCI 132:
Basic Data Structures and Algorithms

Linked Lists

Reese Pearsall

Spring 2025

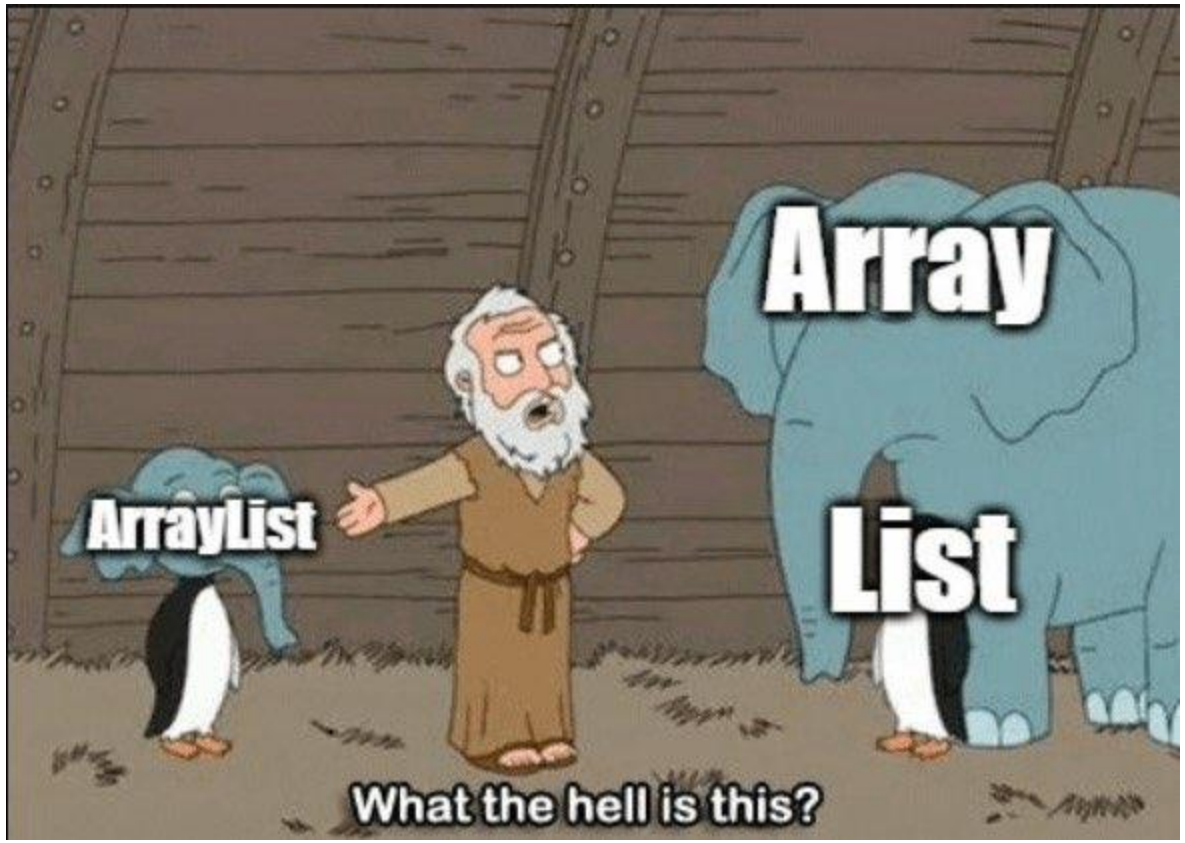
Announcements

Program 1 is due **Wednesday** at 11:59 PM

No class on Monday (President's day)

There will still be a lab due on Tuesday.

Wednesday *may* be a recorded lecture (no in-person lecture)



The List ADT

A **List** is a linear, ordered collection of elements

- Can dynamically grow and shrink in size

The List ADT

A **List** is a linear, ordered collection of elements

- Can dynamically grow and shrink in size

Very vague description. We can achieve this functionality in several different ways in java

The List ADT

A **List** is a linear, ordered collection of elements
- Can dynamically grow and shrink in size

Very vague description. We can achieve this functionality in several different ways in java

Any list should be able to:
- Get(index)
- Add(Element)
- Add(Element, index)
- Remove(Element)
- Remove(Index)
- Size()
- isEmpty()

The List ADT

A **List** is a linear, ordered collection of elements
- Can dynamically grow and shrink in size

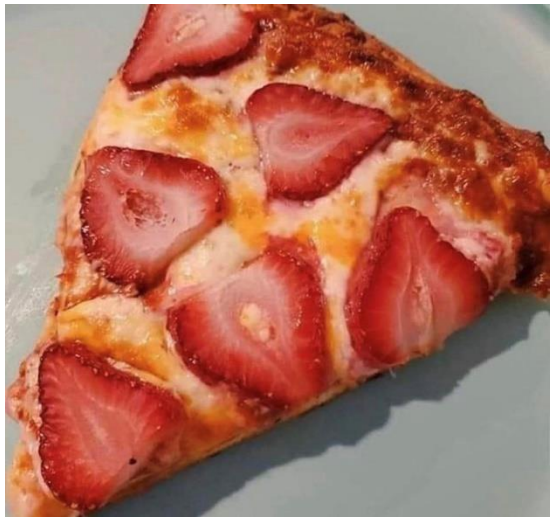
Very vague description. We can achieve this functionality in several different ways in java

Any list should be able to:
- Get(index)
- Add(Element)
- Add(Element, index)
- Remove(Element)
- Remove(Index)
- Size()
- isEmpty()

A List **Abstract Data Type (ADT)** describes *what* a list needs to do, rather than how to implement it

The Pizza ADT

a dish of Italian origin consisting of a flat, round base of dough baked with
a topping of tomato sauce and cheese, typically with added meat or vegetables.



The List ADT

A **List** is a linear, ordered collection of elements
- Can dynamically grow and shrink in size

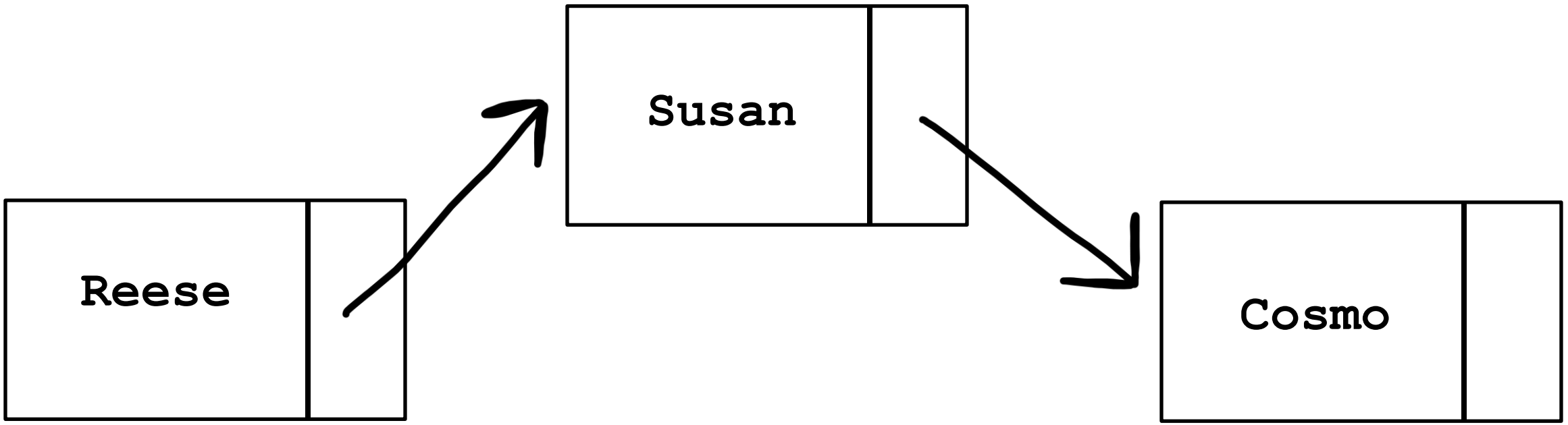
Very vague description. We can achieve this functionality in several different ways in java

Any list should be able to:
- Get(index)
- Add(Element)
- Add(Element, index)
- Remove(Element)
- Remove(Index)
- Size()
- isEmpty()

A List **Abstract Data Type (ADT)** describes *what* a list needs to do, rather than how to implement it

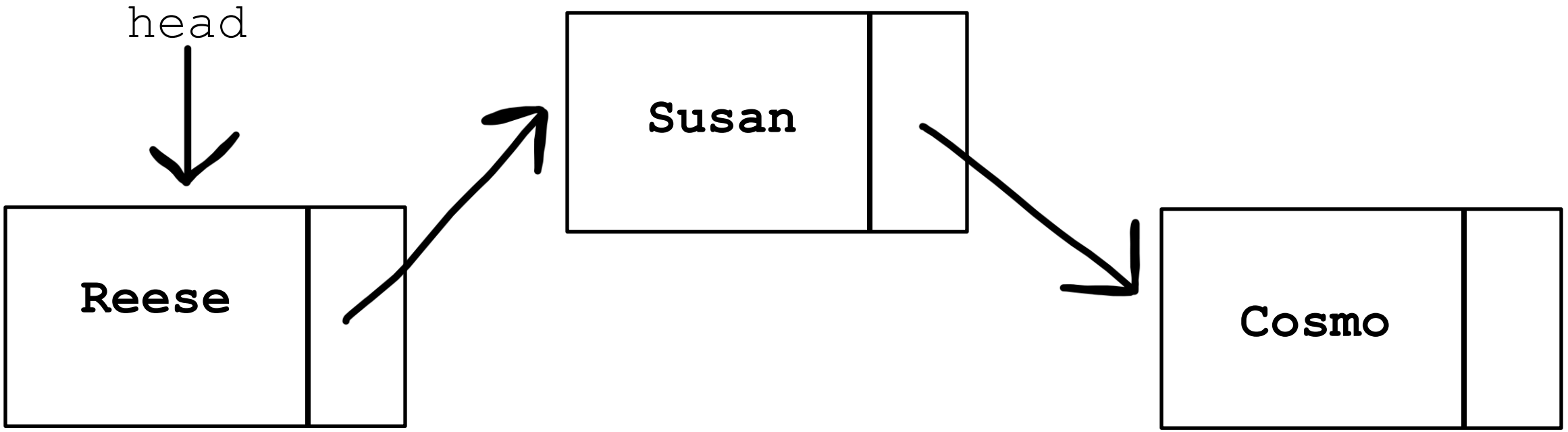
Implementations of a List:
- **ArrayLists**
- **Linked Lists**

A **Linked List** is a data structure that consists of a collection of connected nodes



Nodes consists of **data** (String, int, array, etc) and a **pointer to the next node**

A **Linked List** is a data structure that consists of a collection of connected nodes



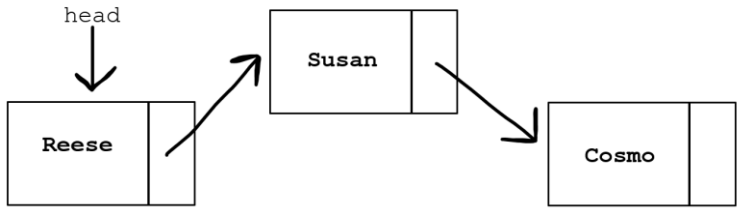
Nodes consists of **data** (String, int, array, etc) and a **pointer to the next node**

A Linked List also has a pointer to the start of the Linked List (`head`)

Node.java

Blueprint for a single node in our data structure. Nodes have data, and a pointer to the next node

LinkedList.java



Collection of nodes connected by pointers.

head pointer
size of linked list

Methods for adding, removing, searching for nodes

LinkedListDemo.java

```
public static void main(String[] args) {

    Node n1 = new Node("Reese");
    Node n2 = new Node("Susan");
    Node n3 = new Node("Cosmo");

    SinglyLinkedList ll = new SinglyLinkedList();

    ll.addToFront(n1);
    ll.addToFront(n2);
    ll.addToFront(n3);
```

Creates the LinkedList

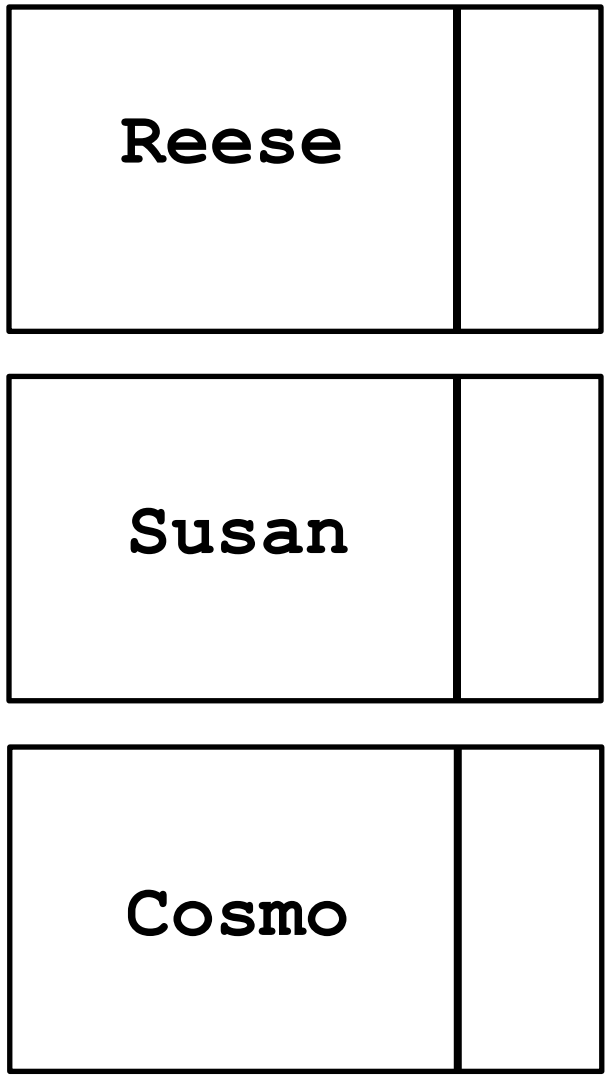
Calls methods to manipulate Linked List

A Linked List will hold `Node` objects

```
public class Node {

    private int age;
    private String name;

    private Node next;

    public Node(int a, String n) {
        this.age = a;
        this.name = n;
        this.next = null;
    }
```

Data

Pointer to next Node

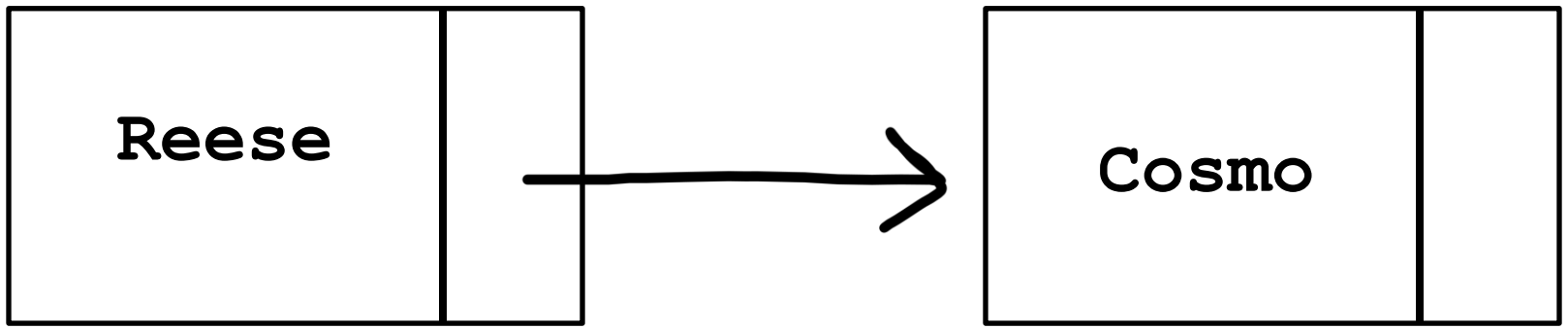


A Linked List will hold `Node` objects

```java
public void setNext(Node n) {
    this.next = n;
}

public Node getNext() {
    return this.next;
}

public String getData() {
    return this.name + ", Age: " + this.age;
}
```

```
System.out.println(reese.getNext().getData())
```

???

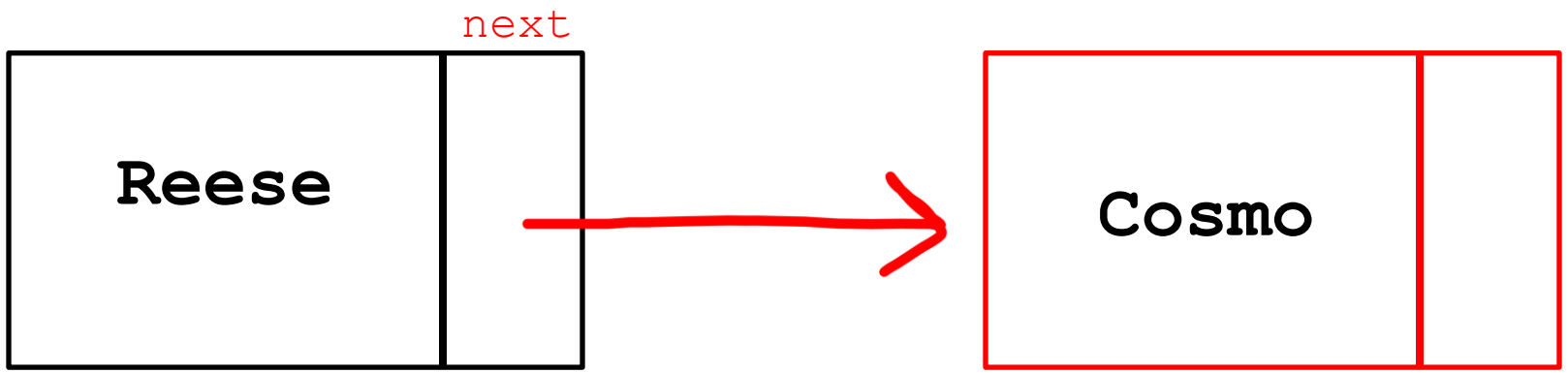


A Linked List will hold `Node` objects

```
public void setNext(Node n) {
    this.next = n;
}

public Node getNext() {
    return this.next;
}

public String getData() {
    return this.name + ", Age: " + this.age;
}
```

```
System.out.println(reese.getNext().getData())
```

This would print out the Cosmo node's data

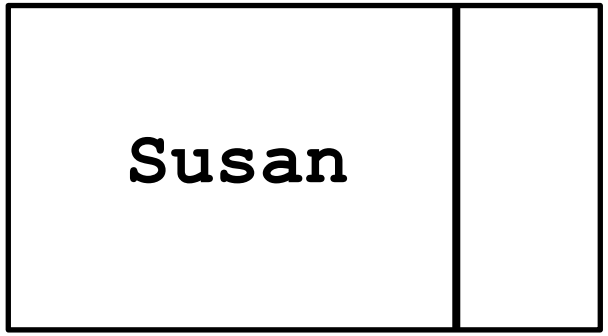


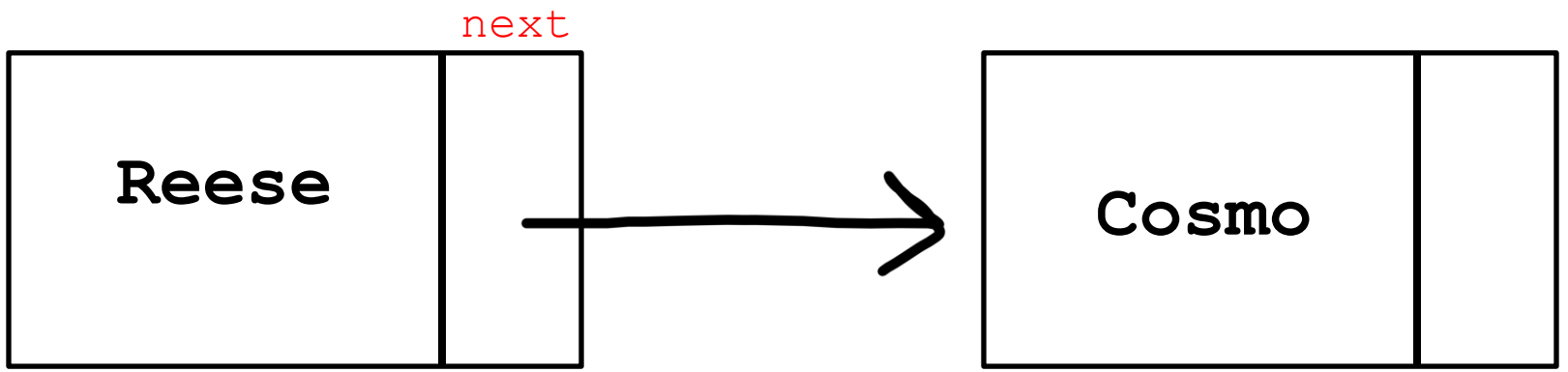
A Linked List will hold `Node` objects

```
public void setNext(Node n) {
    this.next = n;
}

public Node getNext() {
    return this.next;
}

public String getData() {
    return this.name + ", Age: " + this.age;
}
```

`reese.setNext(susan)`

???



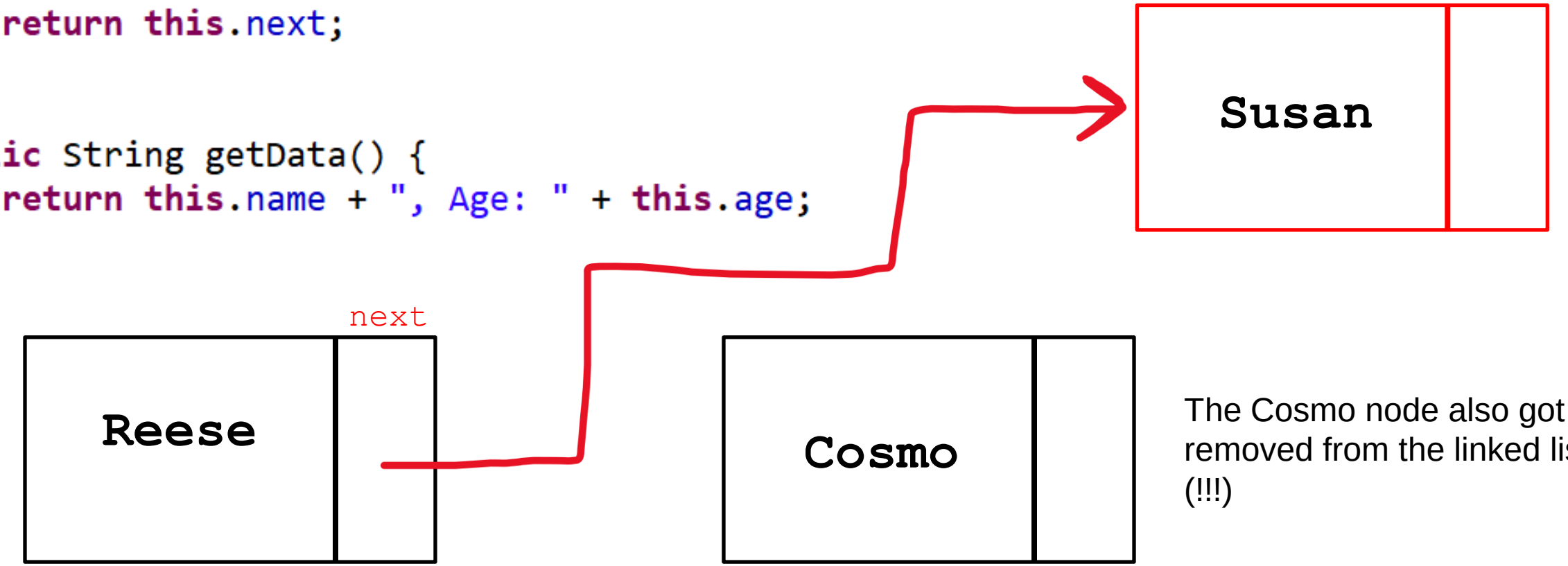


A Linked List will hold `Node` objects

```
public void setNext(Node n) {
    this.next = n;
}

public Node getNext() {
    return this.next;
}

public String getData() {
    return this.name + ", Age: " + this.age;
}
```

`reese.setNext(susan)`

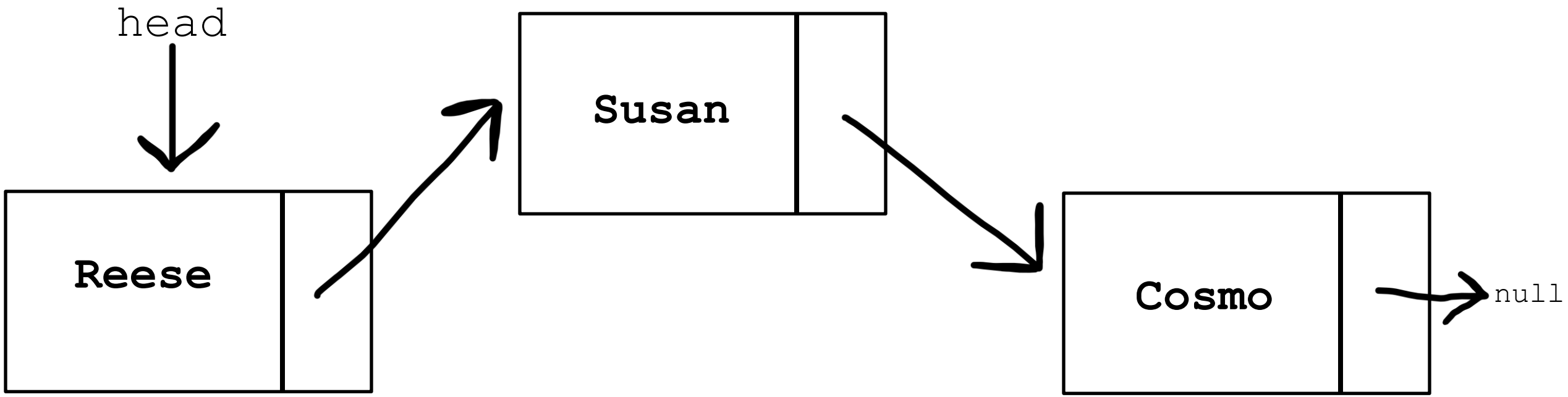
Set's the Reese's node `next` value to point to Susan



The Cosmo node also got removed from the linked list (!!!)

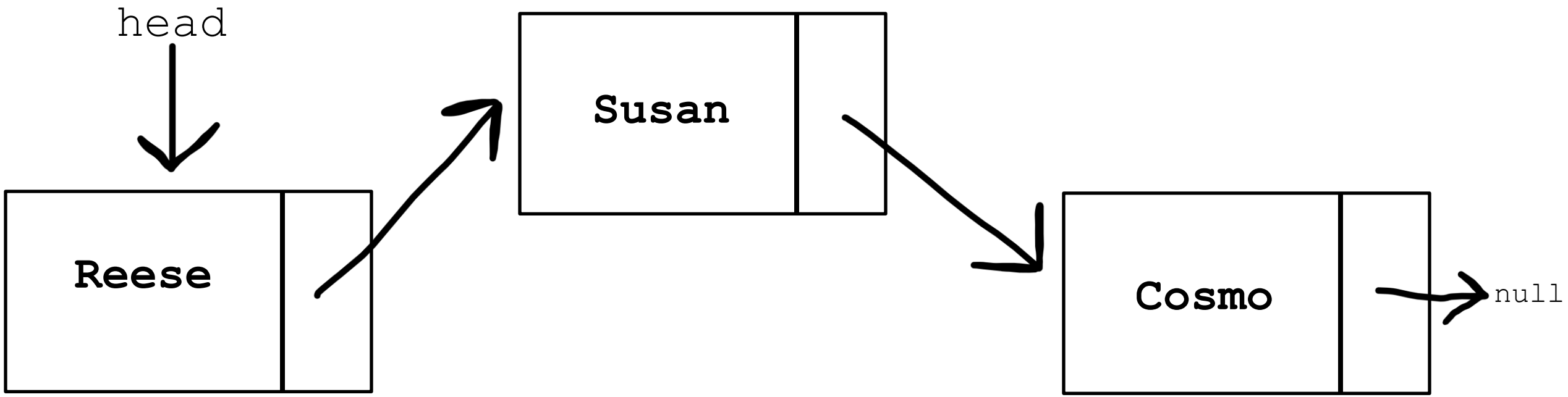
Linked List Creation

```
public class SinglyLinkedList {

    private Node head;
    private int size;

    public SinglyLinkedList() {
        head = null;
        size = 0;
    }
```



Linked List Methods

- `addToFront()` - adds new node to beginning of LL
- `addToBack()` – adds new node to end of LL
- `removeFirst()` – removes first node of LL
- `removeLast()` – removes last node of LL
- `printLinkedList()` – prints nodes and their data



Linked List Methods

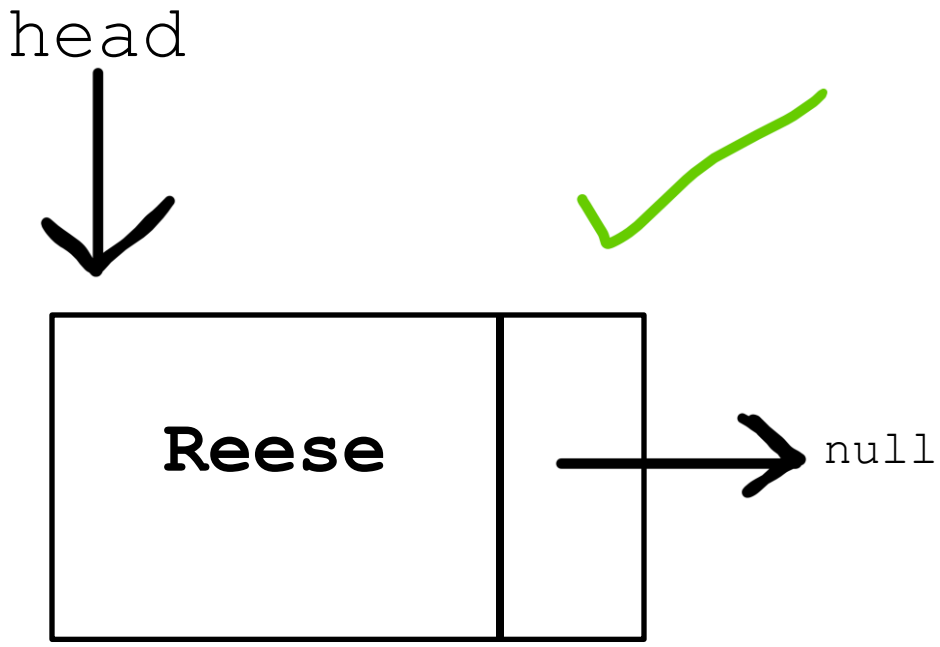
- `addToFront()` - adds new node to beginning of LL

What if the Linked List is empty?

- `addToFront()` - adds new node to beginning of LL

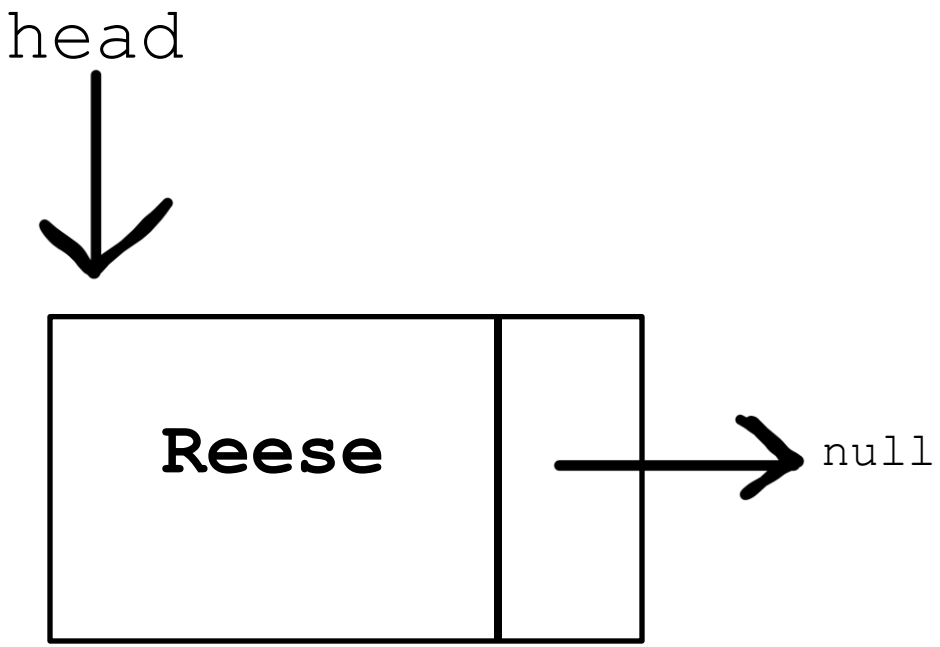
What if the Linked List is empty?

Set `head` equal to the new node



- `addToFront()` - adds new node to beginning of LL

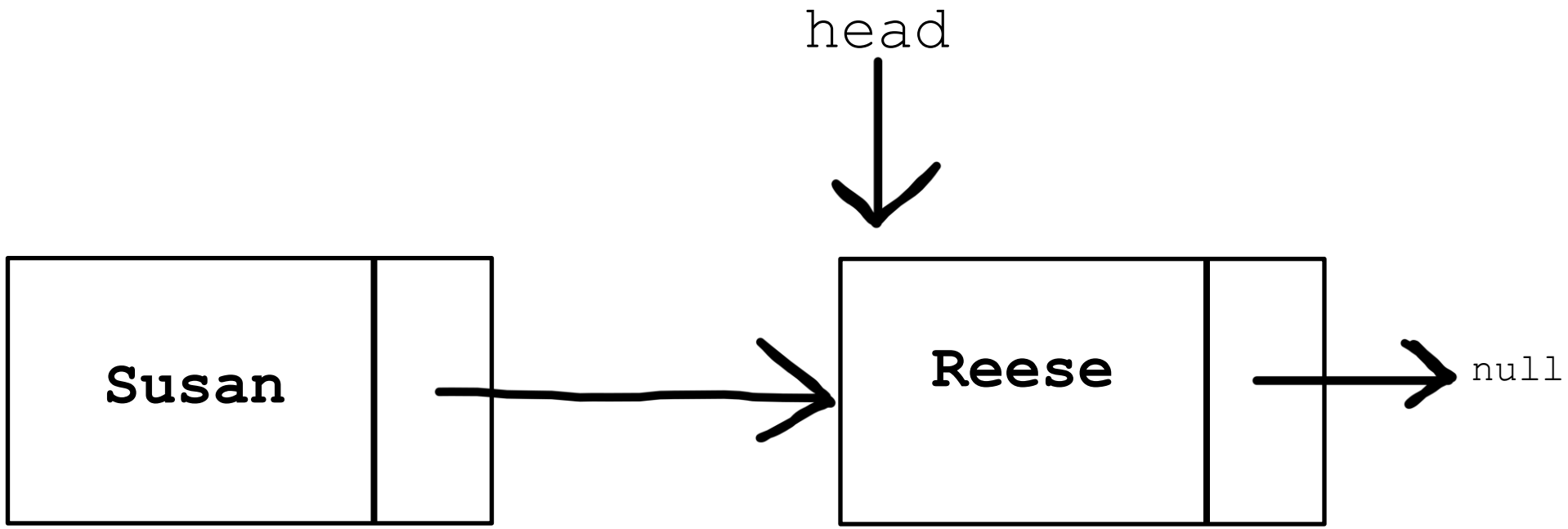
What if the Linked List is not empty?



Linked List Methods • `addToFront()` - adds new node to beginning of LL

What if the Linked List is not empty?

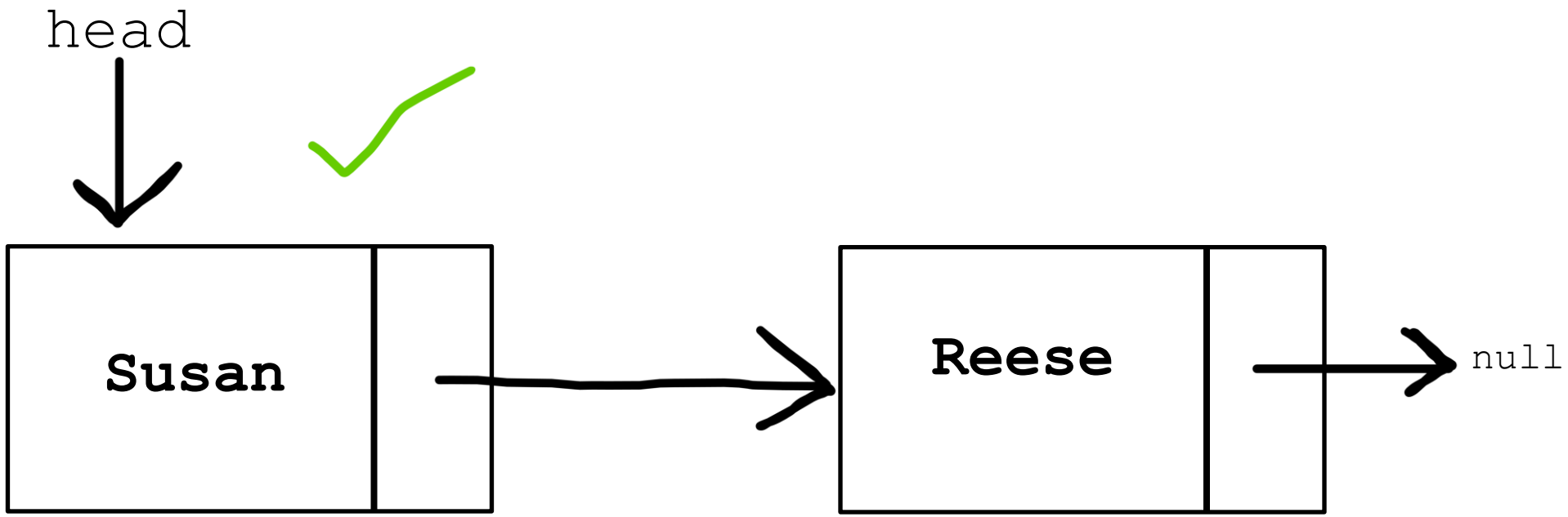
1. Set the new node's `next` value to `head`



Linked List Methods • `addToFront()` - adds new node to beginning of LL

What if the Linked List is not empty?

1. Set the new node's `next` value to `head`
2. Update `head` to point to new node

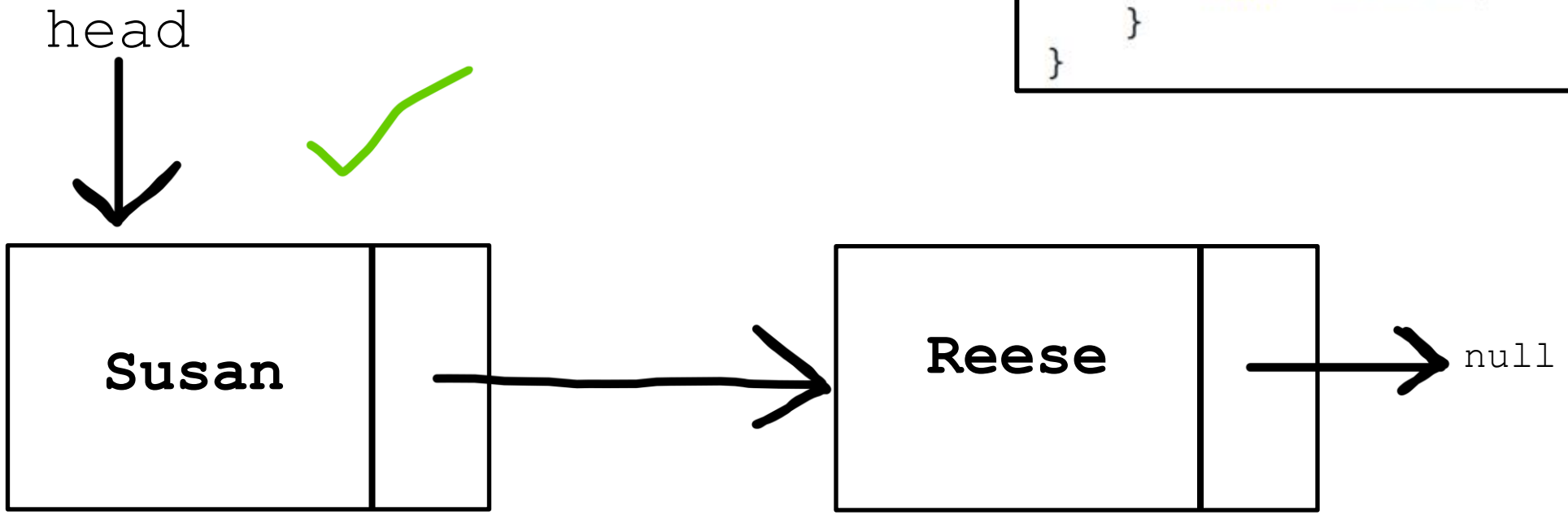


Linked List Methods • `addToFront()` - adds new node to beginning of LL

What if the Linked List is not empty?

1. Set the new node's `next` value to `head`
2. Update `head` to point to new node

```
public void addToFront(Node newNode) {
    if(head == null) {
        head = newNode;
    }
    else {
        newNode.setNext(head);
        head = newNode;
    }
}
```



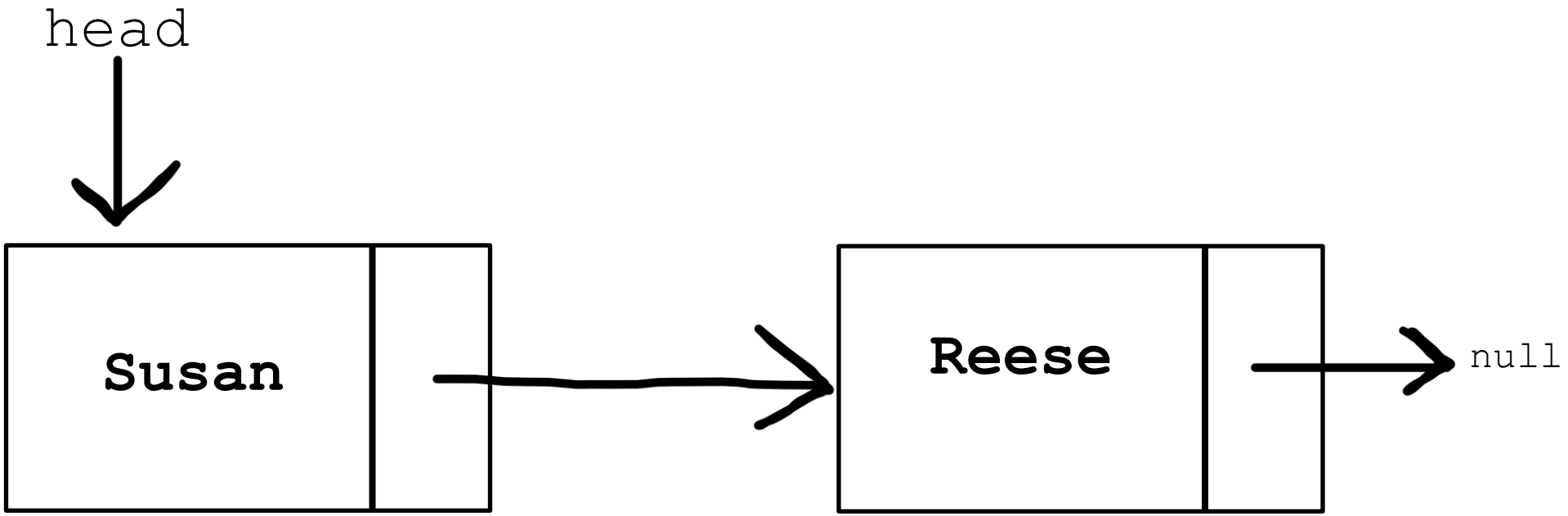
Linked List Methods

- `addToBack()` – adds new node to end of LL

We need to find the end of the Linked List, but we don't know how many Nodes there may be…

We need to find the last node!

- But how do we know if a node is the last node ???



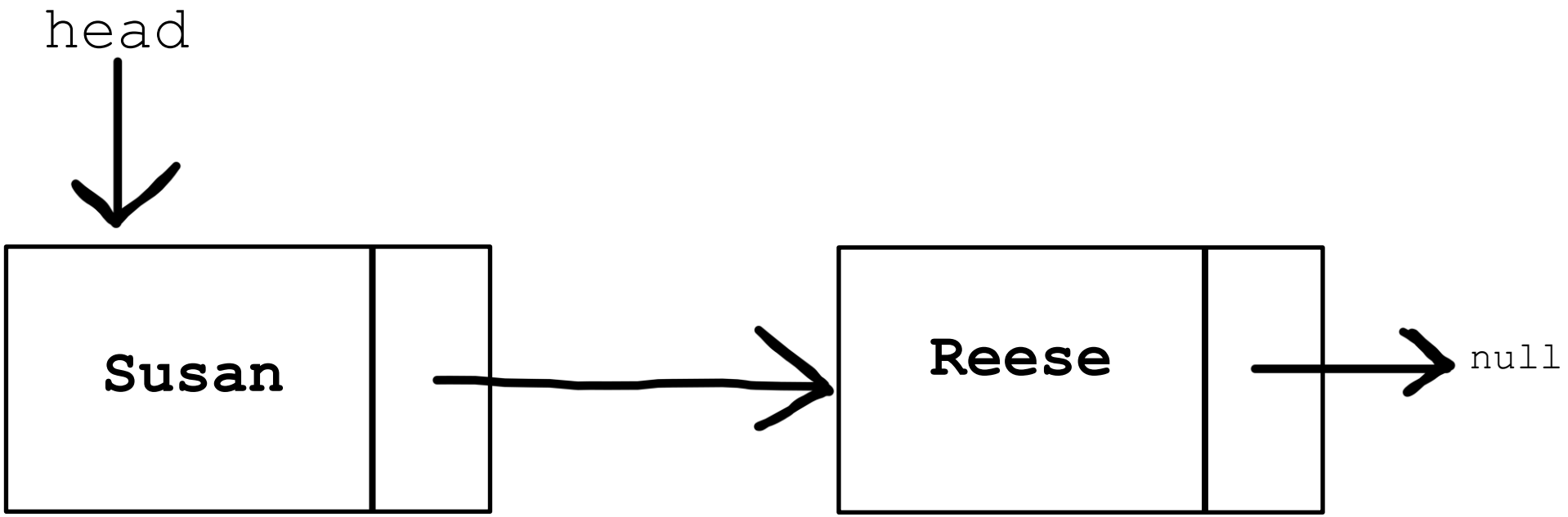
Linked List Methods

- `addToBack()` – adds new node to end of LL

We need to find the end of the Linked List, but we don't know how many Nodes there may be…

We need to find the last node!

- But how do we know if a node is the last node? If a node's `next` value is null



Linked List Methods

- `addToBack()` – adds new node to end of LL

We need to find the end of the Linked List, but we don't know how many Nodes there may be…

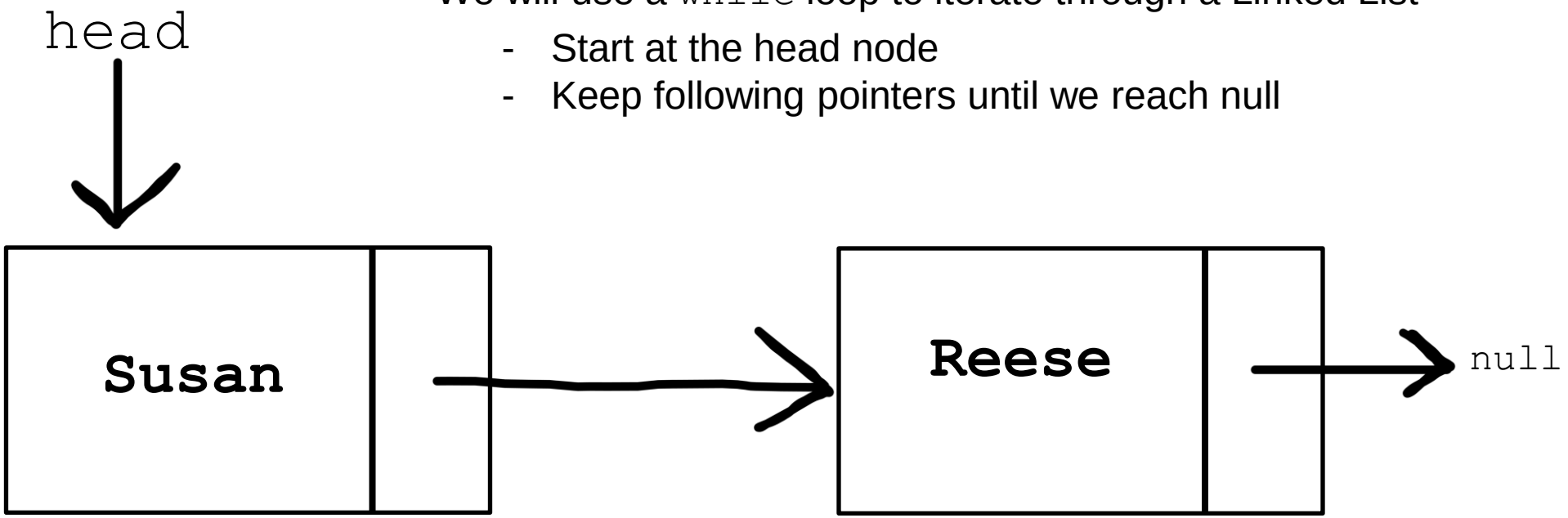
We need to find the last node!

- But how do we know if a node is the last node? If a node's `next` value is null

1. Traverse through the linked list until we find the last node

We will use a `while` loop to iterate through a Linked List

- Start at the head node
- Keep following pointers until we reach null



Linked List Methods

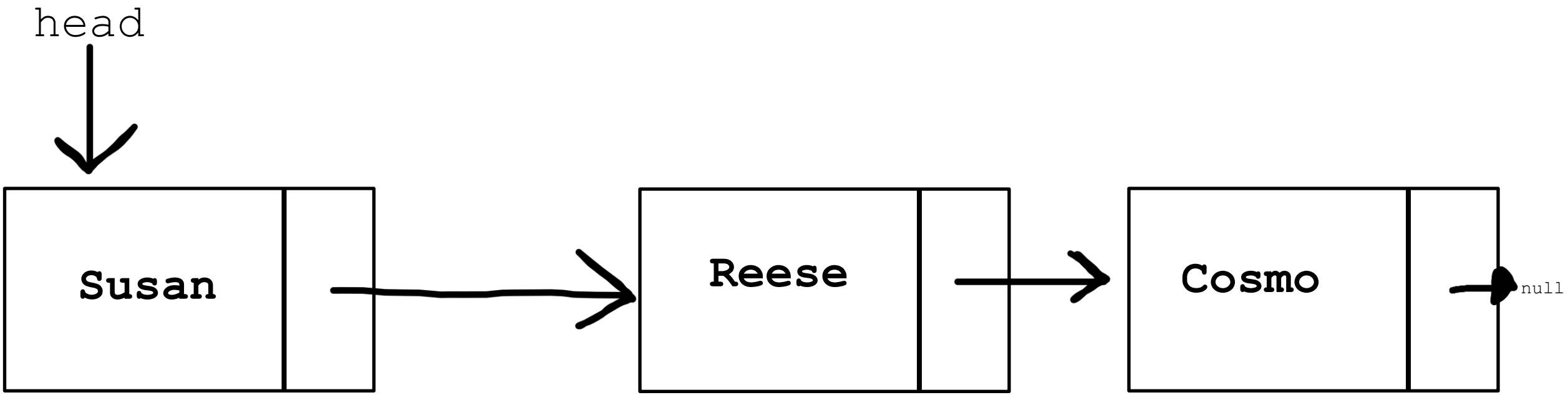
- `addToBack()` – adds new node to end of LL

We need to find the end of the Linked List, but we don't know how many Nodes there may be…

We need to find the last node!

- But how do we know if a node is the last node? If a node's `next` value is null

1. Traverse through the linked list until we find the last node
2. Set the last node's `next` value equal to the new node

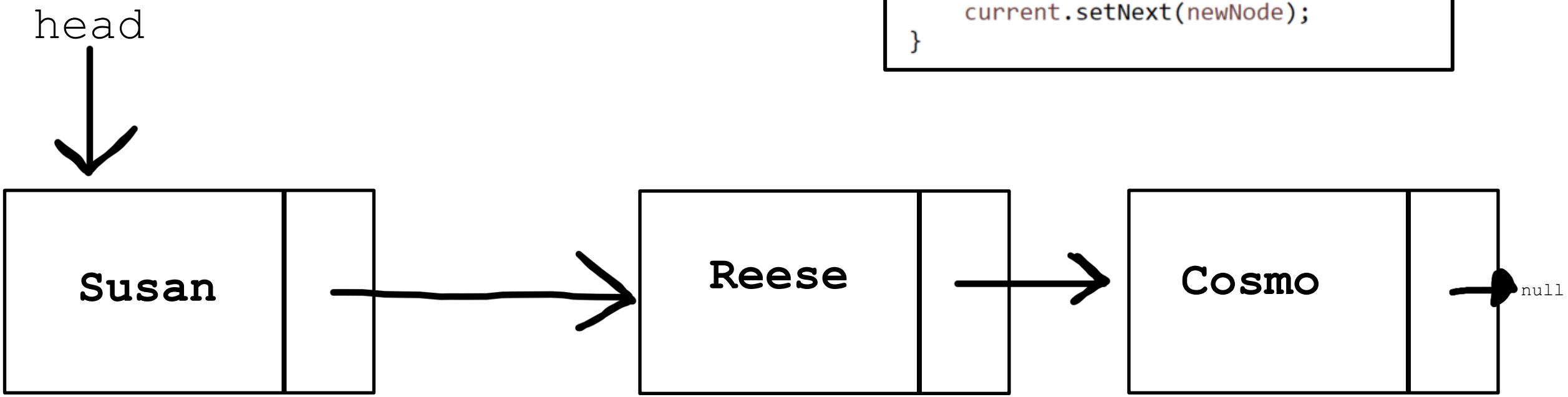


Linked List Methods

- `addToBack()` – adds new node to end of LL

1. Traverse through the linked list until we find the last node
2. Set the last node's `next` value equal to the new node

```
public void addToBack(Node newNode) {

    Node current = head;
    while(current.getNext() != null) {
        current = current.getNext();
    }
    current.setNext(newNode);
}
```



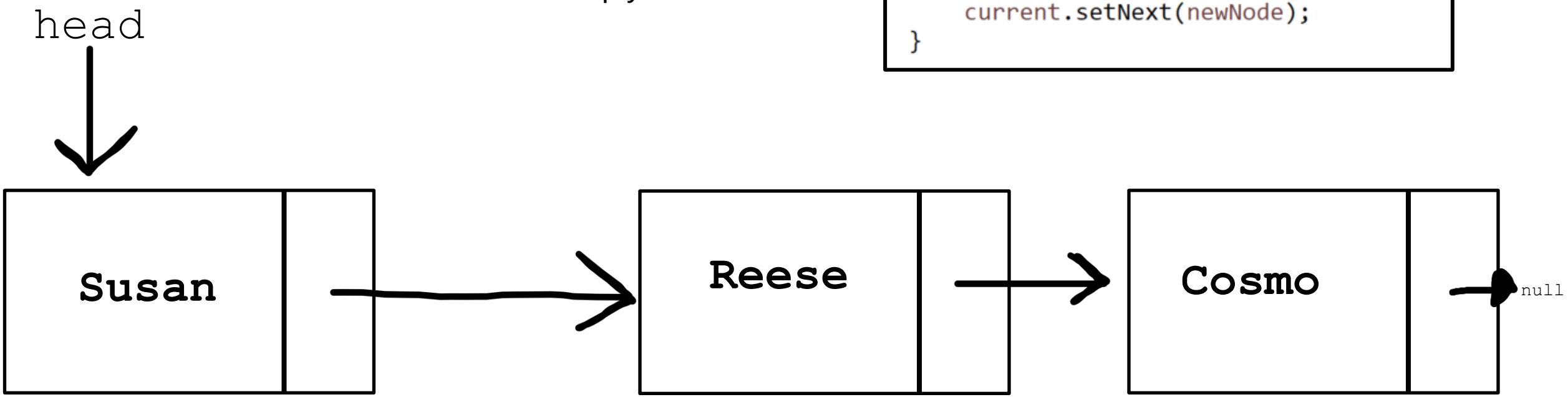
Linked List Methods

- `addToBack()` – adds new node to end of LL

1. Traverse through the linked list until we find the last node
2. Set the last node's `next` value equal to the new node

This method will fail if the LL is empty

```
public void addToBack(Node newNode) {

    Node current = head;
    while(current.getNext() != null) {
        current = current.getNext();
    }
    current.setNext(newNode);
}
```



Linked List Methods

- `printLinkedList()` – prints nodes and their data

Iterate through each Node in the LL, and print the data in that node

Linked List Methods

- `printLinkedList()` – prints nodes and their data

Iterate through each Node in the LL, and print the data in that node

```
public void printLinkedList() {

    Node current = head;
    while(current != null) {
        System.out.println(current.getData());
        current = current.getNext();
    }


}
```

Linked List Methods

- `printLinkedList()` – prints nodes and their data

Iterate through each Node in the LL, and print the data in that node

```
public void printLinkedList() {

    Node current = head;
    while(current != null){
        System.out.println(current.getData());
        current = current.getNext();
    }

}
```

Always start at the head node

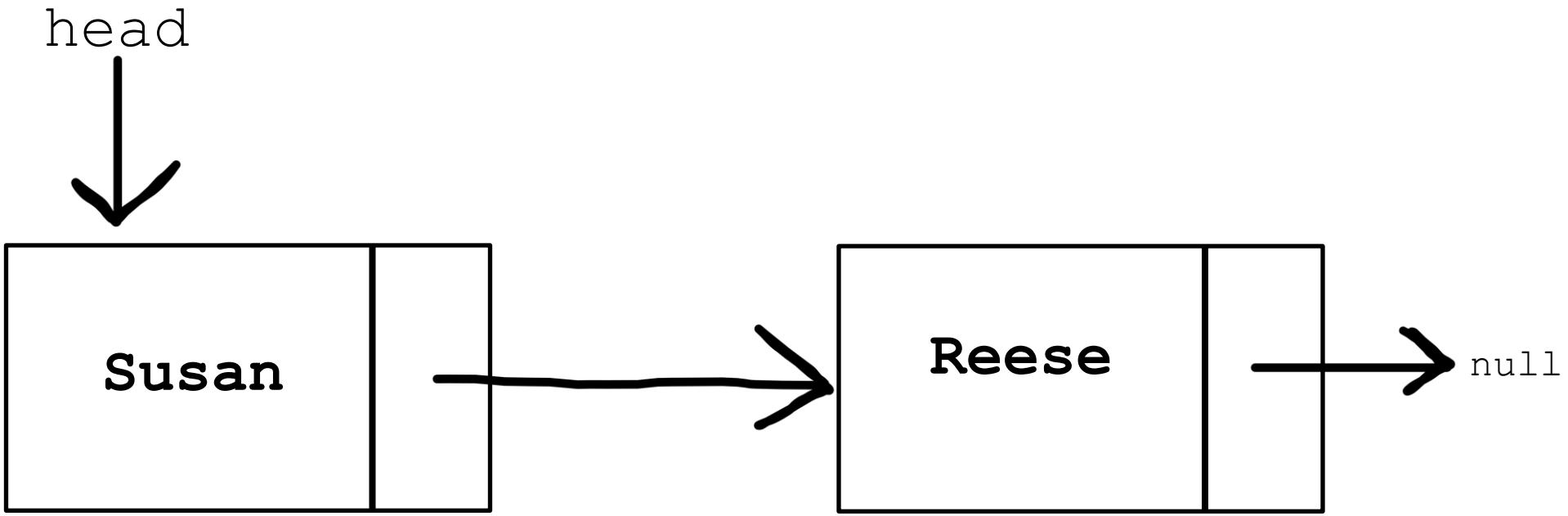
"Keep on looping until we reach the end of the LL"

This line updates the current node we are at

ie. "move to the next node"

Linked List Methods

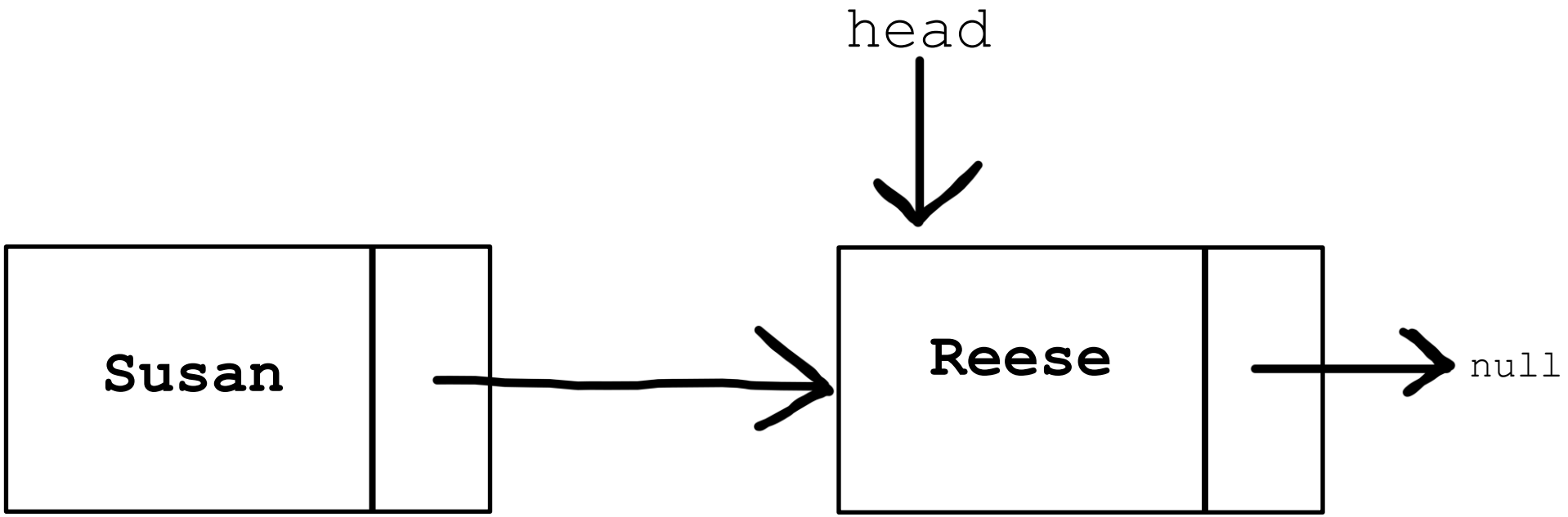
- `removeFirst()` – removes first node of LL



Linked List Methods

- `removeFirst()` – removes first node of LL

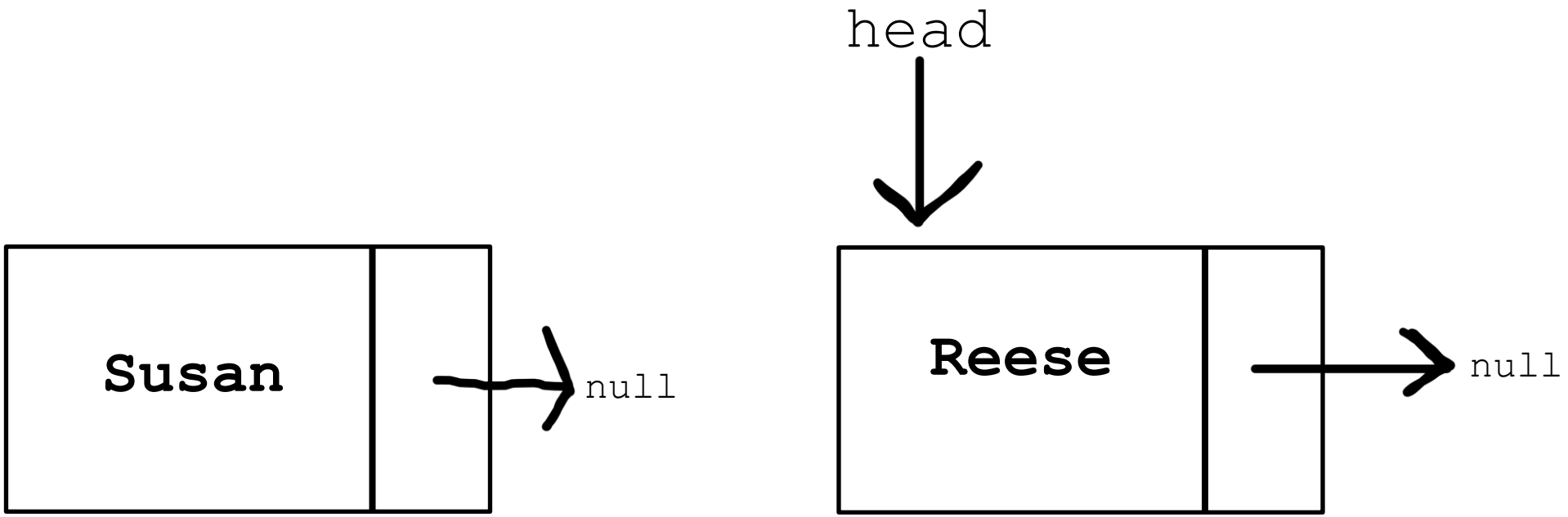
1. Update `head` to be the next node



Linked List Methods

- `removeFirst()` – removes first node of LL

1. Update `head` to be the next node
2. Update the old `head`'s `next` value to be `null`



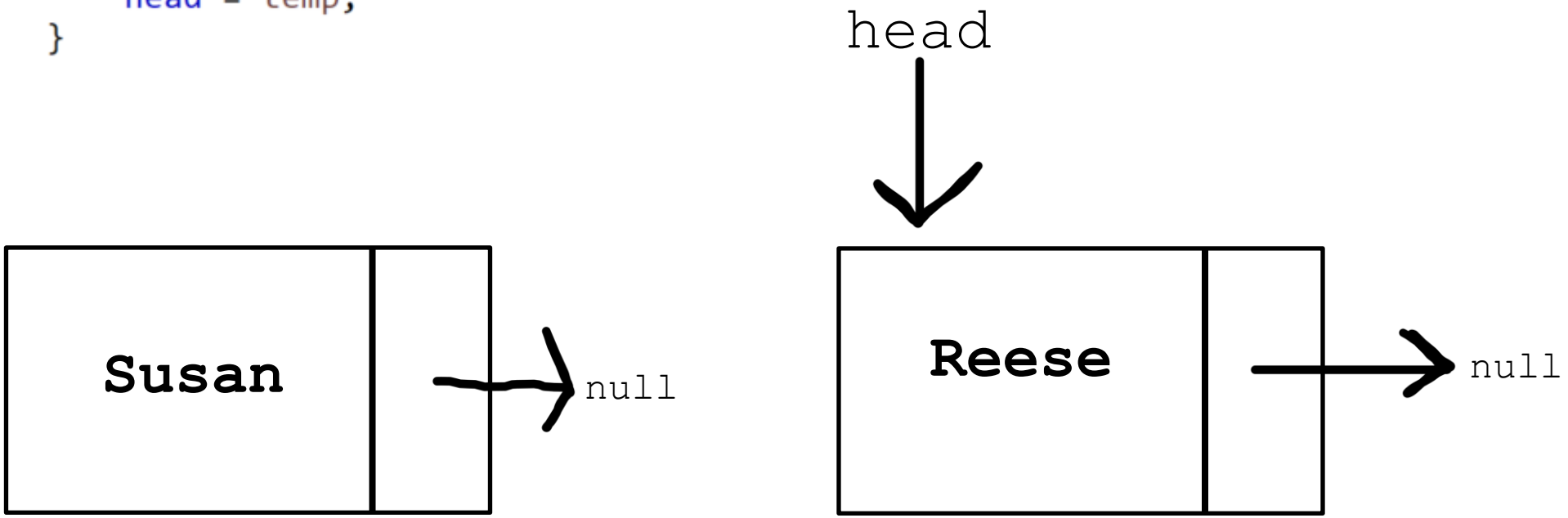
Linked List Methods

- `removeFirst()` – removes first node of LL

1. Update `head` to be the next node
2. Update the old `head`'s `next` value to be `null`

```
public void removeFirst() {
    Node temp = this.head.getNext();
    head.setNext(null);
    head = temp;
}
```

Create a new temporary variable to save 2nd node value

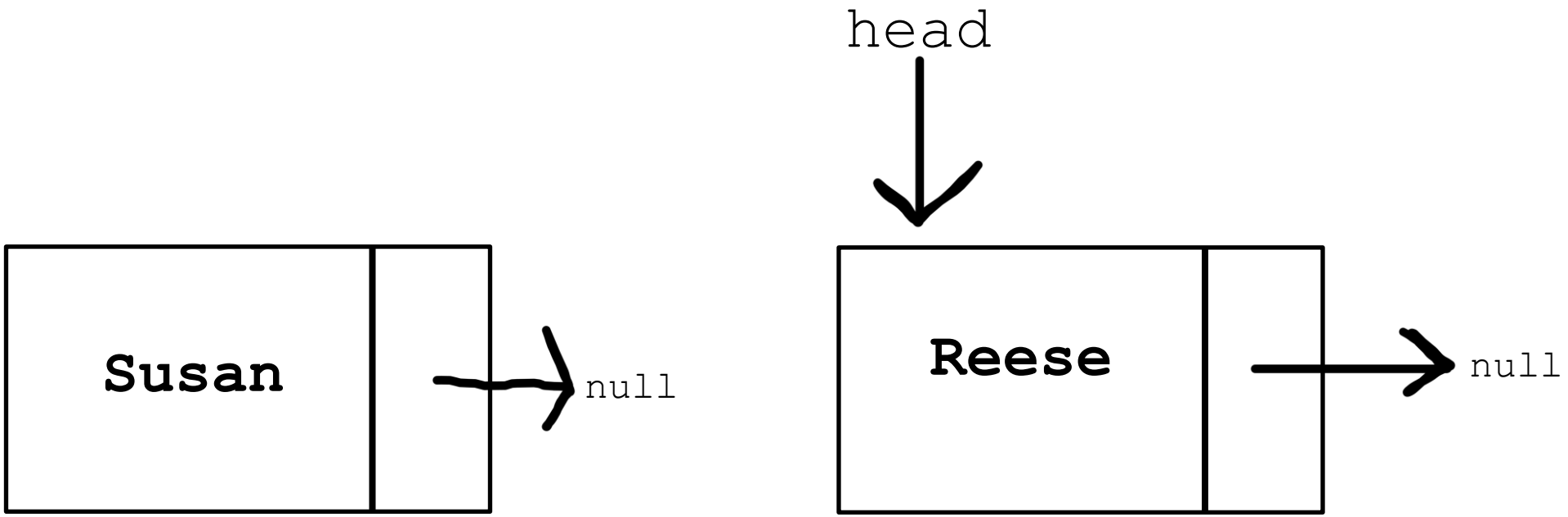


Linked List Methods

- `removeFirst()` – removes first node of LL

1. Update `head` to be the next node
2. Update the old `head`'s `next` value to be `null`

There's an easier way to do this



Linked List Methods

- `removeFirst()` – removes first node of LL

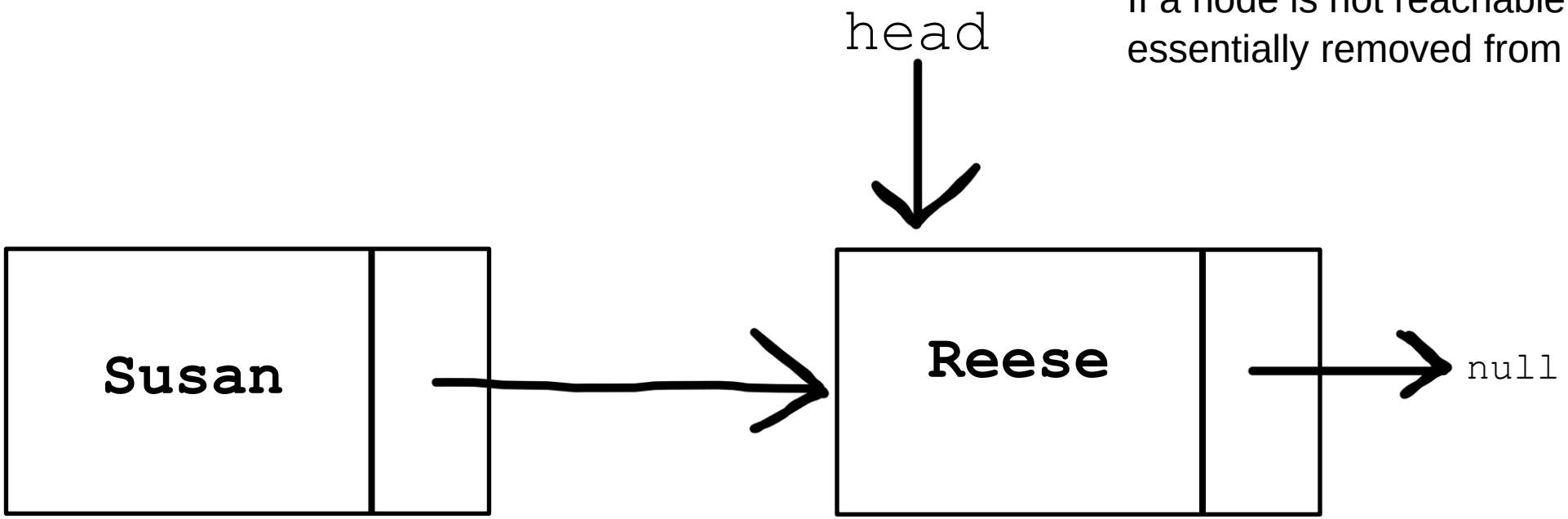
1. Update `head` to be the next node
2. ~~Update the old `head`'s next value to be `null`~~

There's an easier way to do this

We don't need to remove the pointer.

Remember, whenever we iterate or add something to a list, we always start from the `head` node

If a node is not reachable from the `head`, it is essentially removed from the LL !!



Linked List Methods

- `removeFirst()` – removes first node of LL

1. Update `head` to be the next node
2. ~~Update the old `head`'s next value to be `null`~~

There's an easier way to do this

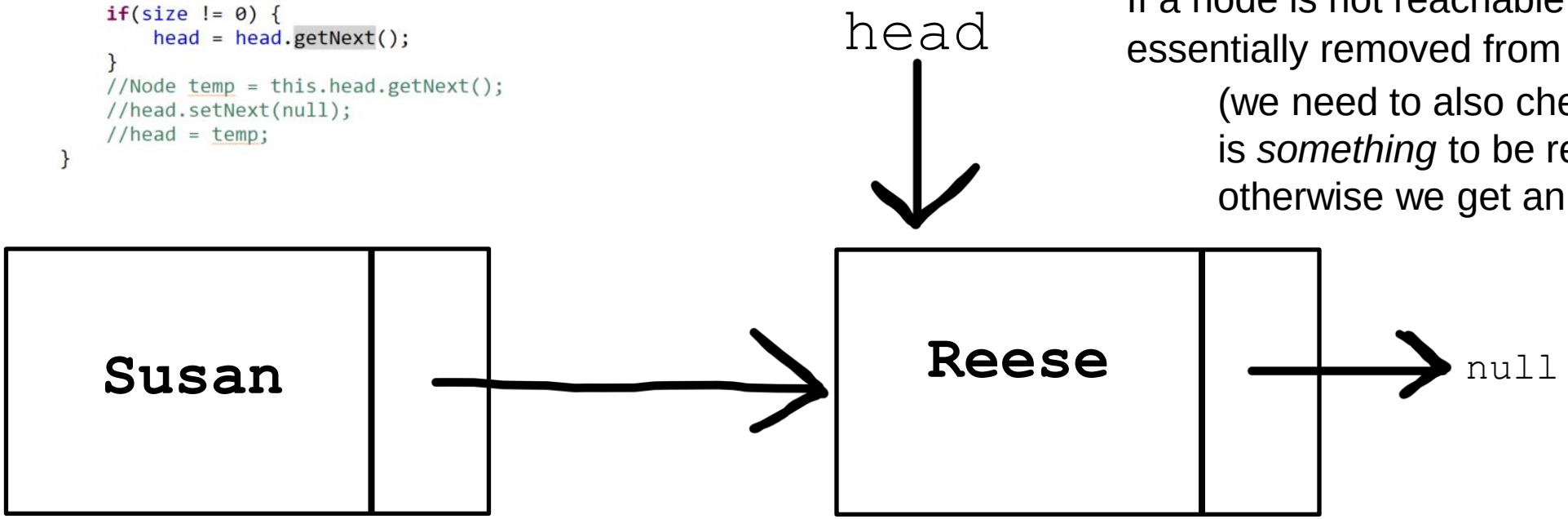
```
public void removeFirst() {

    if(size != 0) {
        head = head.getNext();
    }
    //Node temp = this.head.getNext();
    //head.setNext(null);
    //head = temp;
}
```

We don't need to remove the pointer.

Remember, whenever we iterate or add something to a list, we always start from the `head` node

If a node is not reachable from the `head`, it is essentially removed from the LL !!

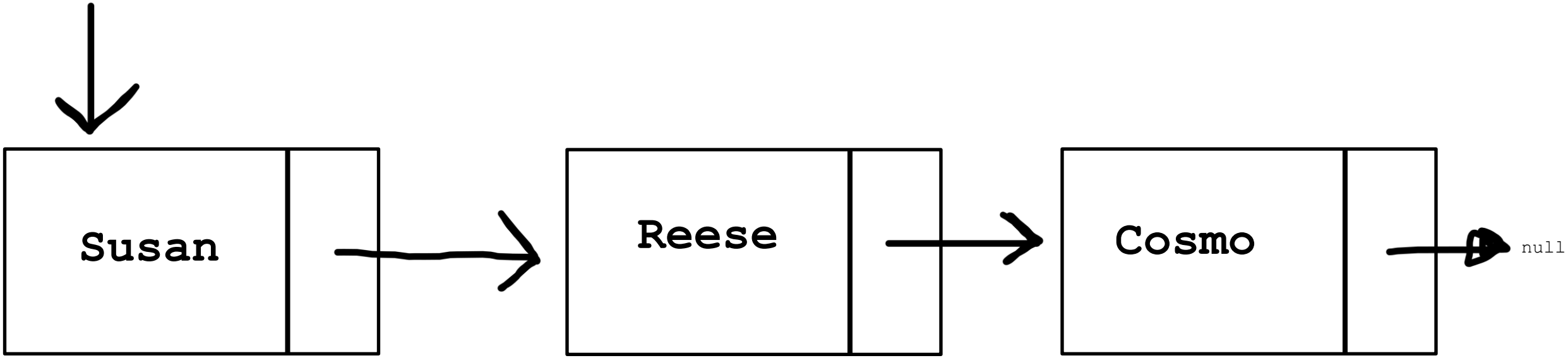
(we need to also check that there is *something* to be removed, otherwise we get an error)



Linked List Methods

- `removeLast()` – removes last node of LL

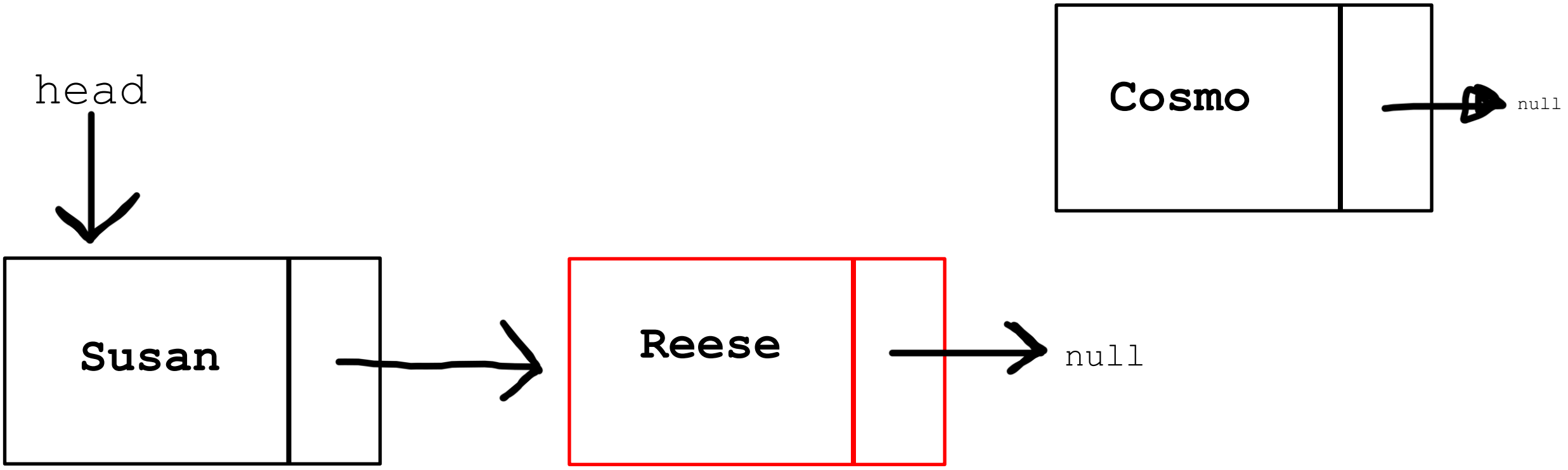
???



Linked List Methods

- `removeLast()` – removes last node of LL

1. Find the second to last node
2. Set that node's `next` value to `null`

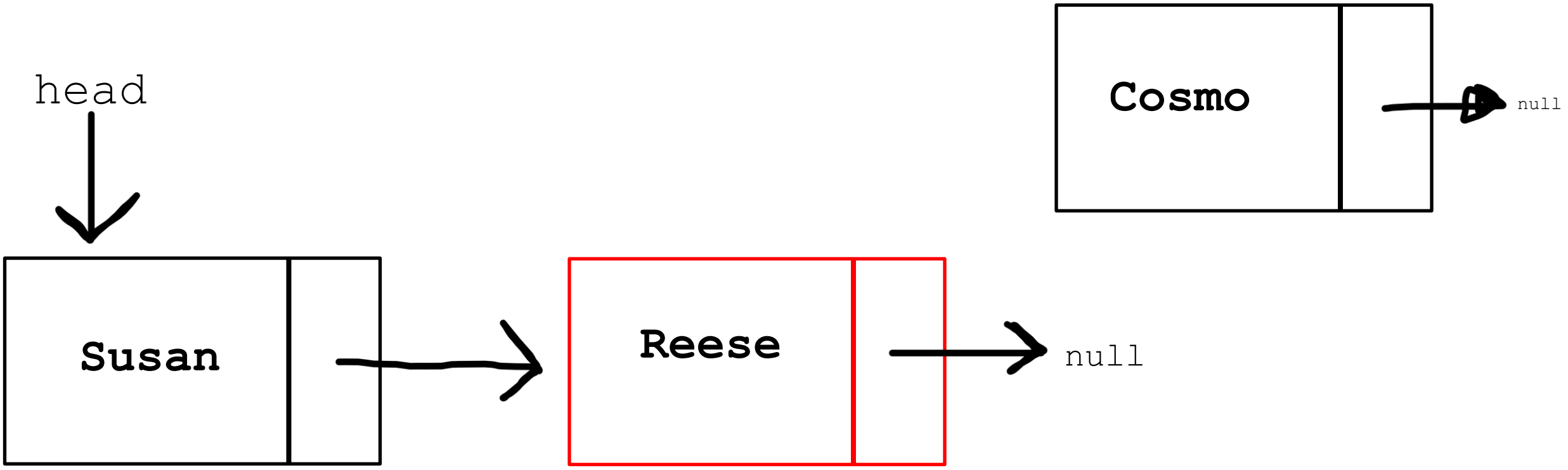


Linked List Methods

- `removeLast()` – removes last node of LL

1. Find the second to last node
2. Set that node's `next` value to `null`

```
public void removeLast() {

    Node current = head;
    while(current.getNext().getNext() != null) {
        current = current.getNext();
    }
    current.setNext(null);
}
```

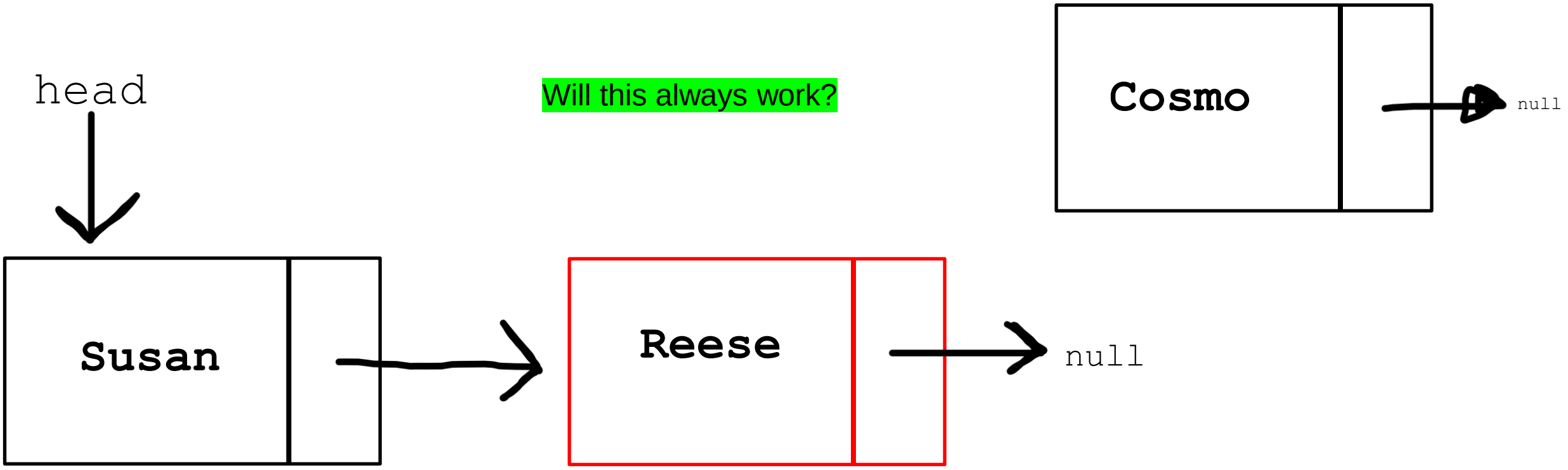


Linked List Methods

- `removeLast()` – removes last node of LL

1. Find the second to last node
2. Set that node's `next` value to `null`

```
public void removeLast() {

    Node current = head;
    while(current.getNext().getNext() != null) {
        current = current.getNext();
    }
    current.setNext(null);
}
```



Linked List Methods

- `removeLast()` – removes last node of LL

1. Find the second to last node
2. Set that node's `next` value to `null`

```
public void removeLast() {

    Node current = head;
    while(current.getNext().getNext() != null) {
        current = current.getNext();
    }
    current.setNext(null);
}
```

If the LL is empty, or of size 1, this will result in a NullPointerException