

---

CMSC 341

Lecture 7 Lists

Prof. Katherine Gibson

---

---

# Today's Topics

- **Linked Lists**
  - vs Arrays
  - Nodes
- **Using Linked Lists**
  - “Supporting Actors” (member variables)
  - Overview
  - Creation
  - Traversal
  - Deletion

---

# Linked Lists vs Arrays

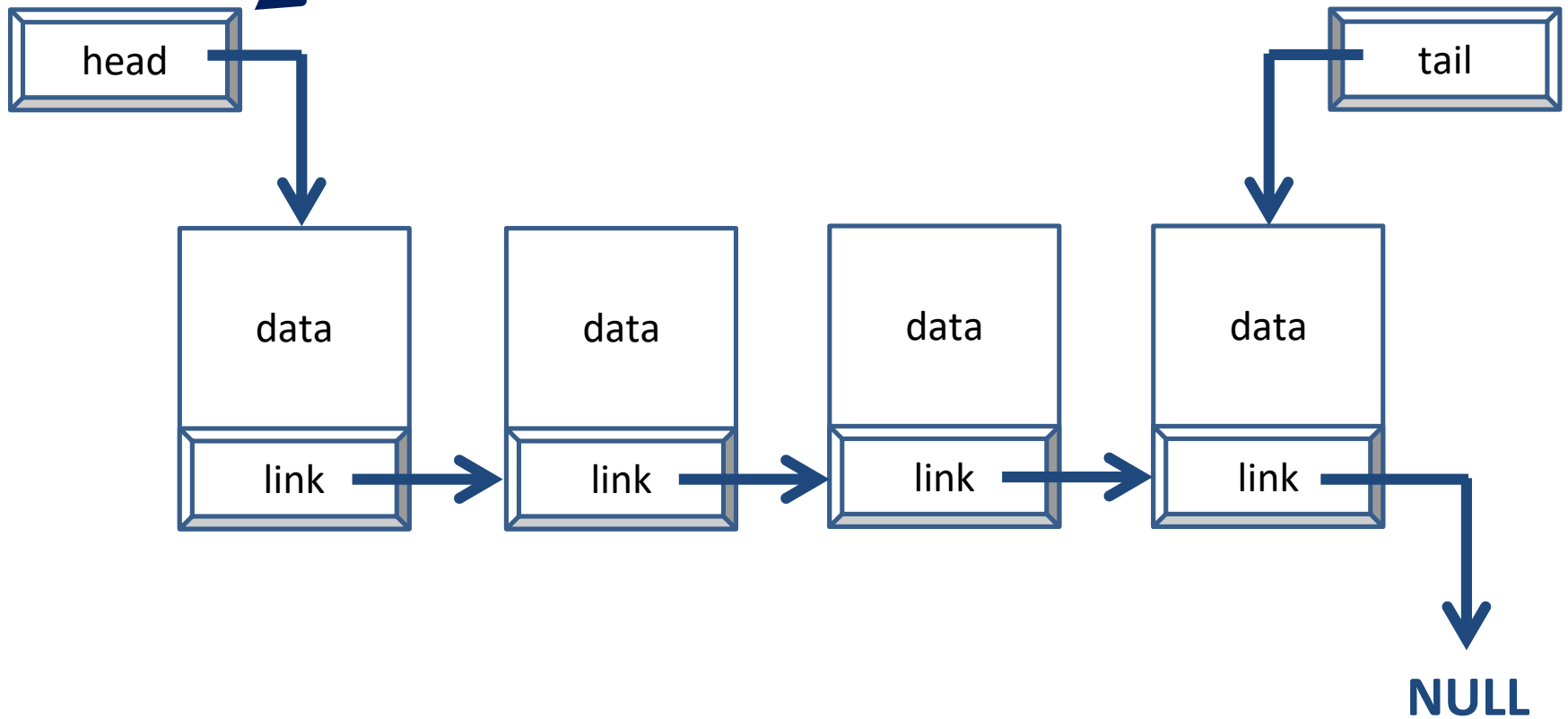
---

# What is a Linked List?

- Data structure
  - Dynamic
  - Allow easy insertion and deletion
- Uses nodes that contain
  - Data
  - Pointer to next node in the list

# Example Linked List

*In these diagrams, a doubly-outlined box indicates a pointer.*

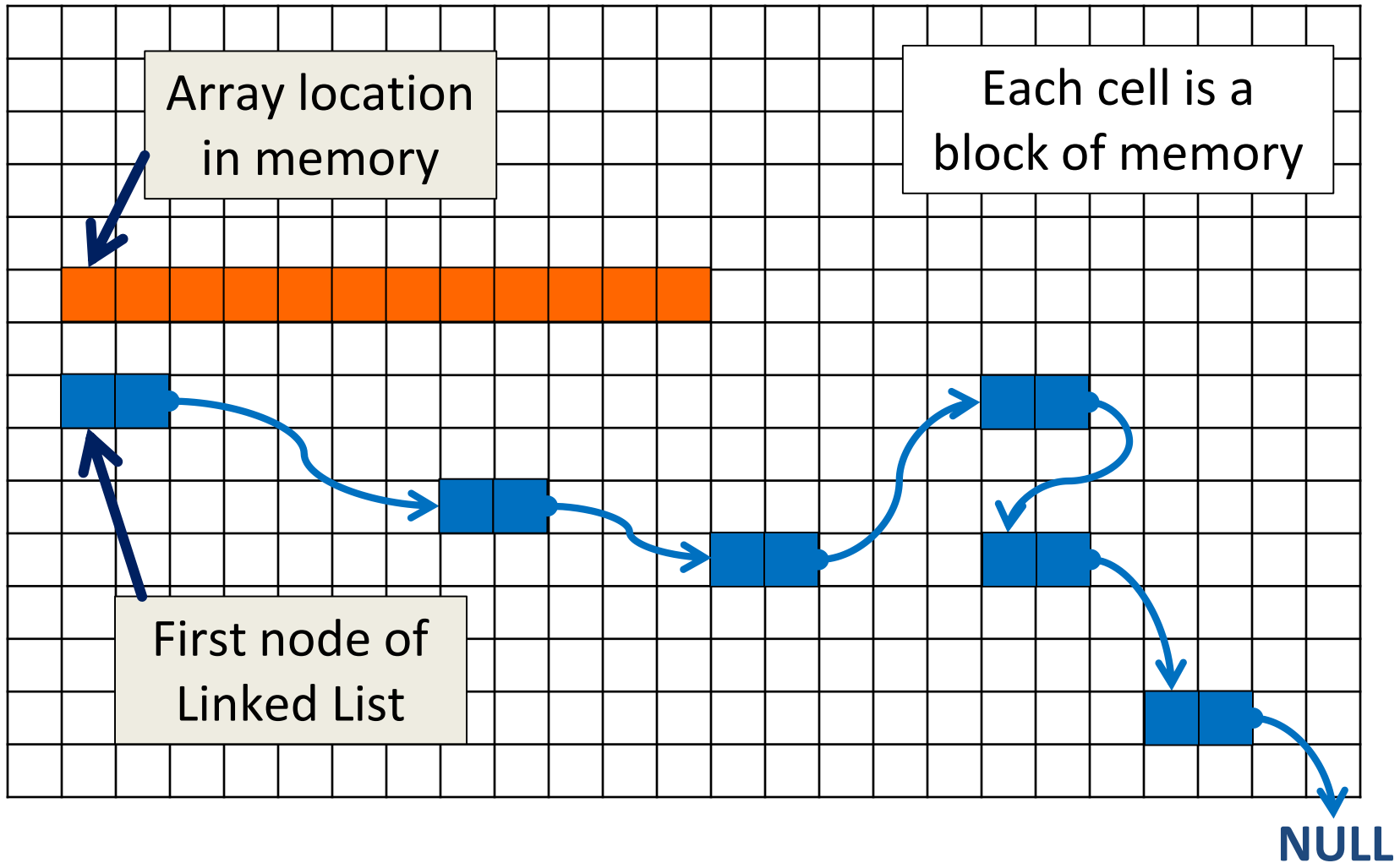


---

# Why Use Linked Lists?

- We already have arrays!
- What are some disadvantages of an array?
  - Size is fixed once created
  - Inserting in the middle of an array takes time
  - Deletion as well
  - Sorting
  - Requires a *contiguous* block of memory

# Arrays vs Linked Lists in Memory



# (Dis)Advantages of Linked Lists

- Advantages:
  - ❑ Change size easily and constantly
  - ❑ Insertion and deletion can easily happen anywhere in the Linked List
  - ❑ Only one node needs to be contiguously stored
- Disadvantages:
  - ❑ Can't access by index value
  - ❑ Requires management of memory
  - ❑ Pointer to next node takes up more memory

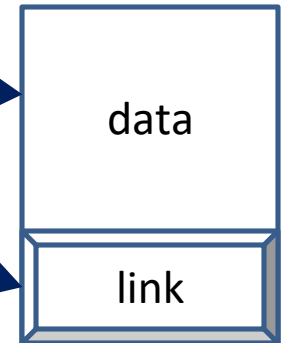


---

# Nodes

# Nodes

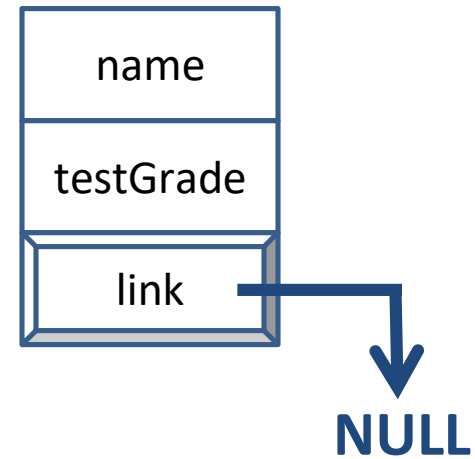
- A node is one element of a Linked List
- Nodes consist of two main parts:
  - Data stored in the node
  - Pointer to next node in list
- Often represented as structs



# Code for Node Structure

```
struct Node
{
    String name;
    int    testGrade;
    struct Node *link;

    // constructor
    // accessors
    // mutators
};
```



link can point to other nodes

two options:

1. another Node
2. NULL

---

# “Supporting Actors” of Linked Lists (Member Variables)

# “Supporting Actors” of a Linked List

- Five member variables used to create and keep track of a Linked List
  - All five variables are **private** members
  - All of them are pointers to a Node
  - **FRONT** (or **HEAD**) points to front of list
  - **REAR** (or **TAIL**) points to end of list
  - **INSERT** used in node creation
  - **CURR** (or **CURSOR**) used to “traverse” list
  - **PREVIOUS** used to “traverse” list

---

# The **FRONT** Node Pointer

- **FRONT** points to the very first node in the Linked List
- What if the Linked List is empty?
  - Points to NULL

---

# The **REAR** Node Pointer

- **REAR** points to the very last item in the Linked List
  - Useful when inserting nodes at the end
- What if there is only one item in the Linked List?
  - Points to the same item as **FRONT**
- What if the Linked List is empty?
  - Points to NULL

---

# The **INSERT** Node Pointer

- **INSERT** is used when we are creating and inserting a new node into the Linked List

**INSERT = new Node;**

- We'll see an example of this soon



---

# The **CURR** and **PREV** Node Pointers

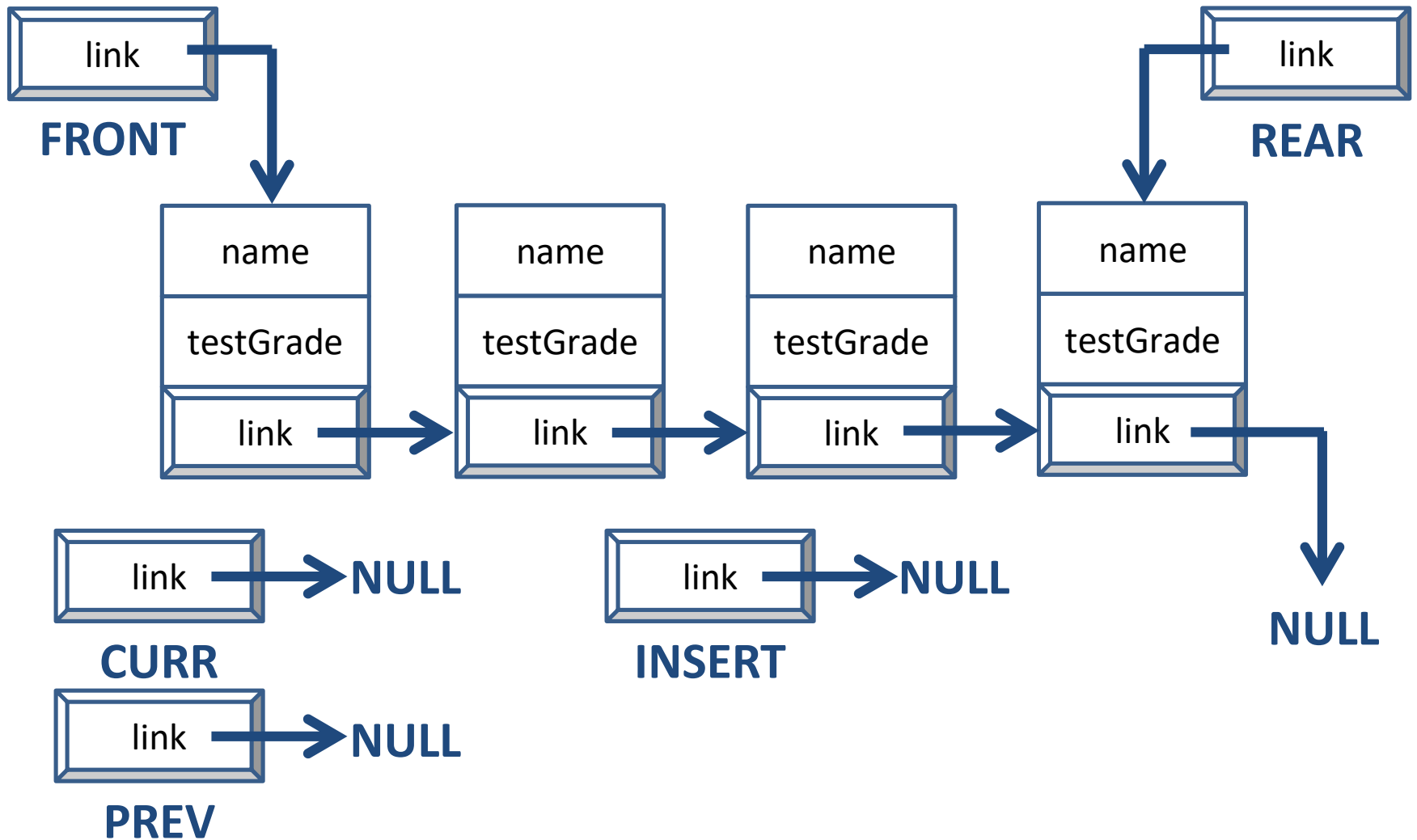
- The **CURR** and **PREV** node pointers are used to “traverse” or travel down the length of a Linked List
- Why do we need two nodes to do this?

---

# Linked List Overview



# Example Linked List (Again)



---

# Important Points to Remember

- Last node in the Linked List points to **NULL**
- Each node points to either another node in the Linked List, or to **NULL**
  - Only one link per node
- **FRONT** and **REAR** point to the first and last nodes of the Linked List, respectively

---

# Managing Memory with Linked Lists

- Hard part of using Linked Lists is ensuring that none of the nodes go “missing”
- Think of Linked List as a train
  - (Or as a conga line of Kindergarteners)
- Must keep track of where links point to
- If you’re not careful, nodes can get lost in memory (you have no way to find them)

---

# Linked List Functions

- What functions does a Linked List class implementation require?
- `Linked_List` constructor
  - Initialize all member variables to `NULL`
- `insert()`
- `remove()`
- `printList()`
- `isEmpty()`

---

# Linked Lists' “Special” Cases

- Linked Lists often need to be handled differently under specific circumstances
  - Linked List is empty
  - Linked List has only one element
  - Linked List has multiple elements
  - Changing something with the first or last node
- Keep this in mind when you are coding



---

# Creation of a Linked List

# Creation of a New Linked List

- Call constructor

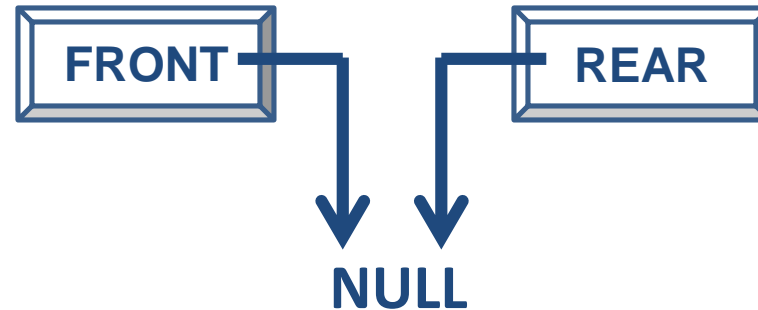
```
Linked_List test = new Linked_List();
```

- What does the constructor do?

```
// constructor definition
Linked_List() {
    FRONT    = NULL;
    REAR     = NULL;
    INSERT   = NULL;
    CURR     = NULL;
    PREV     = NULL;
}
```

- Why are they all set to **NULL**?

# Current State of Linked List **test**

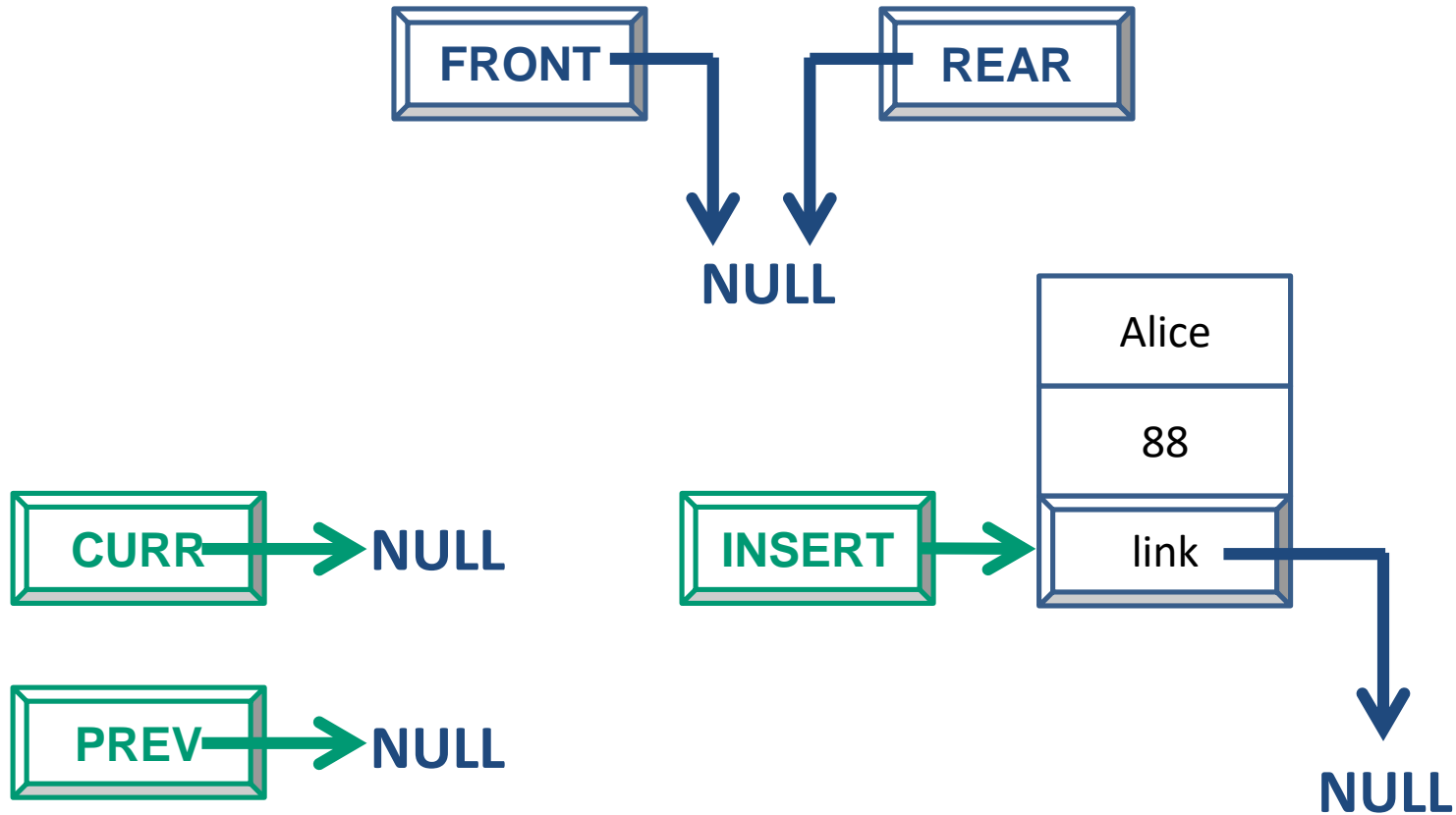


# Inserting the First Node

- What do we do first?
  - Allocate space for the node, using **INSERT**
  - Initialize Node's data
  - Then what?
    - What are the two cases we care about?

```
void Linked_List::insert (String name, int score) {  
    INSERT = new Node()  
    // initialize data  
    INSERT.setName (name);  
    INSERT.setGrade(score);  
    // what do we do?
```

# Current State of Linked List **test**

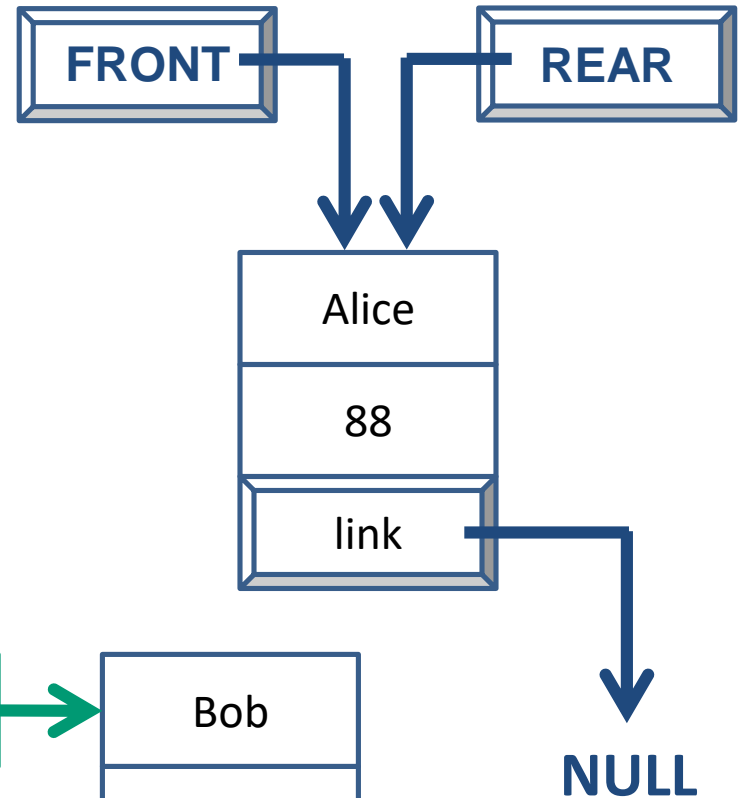


# Insertion: Empty Linked List Case

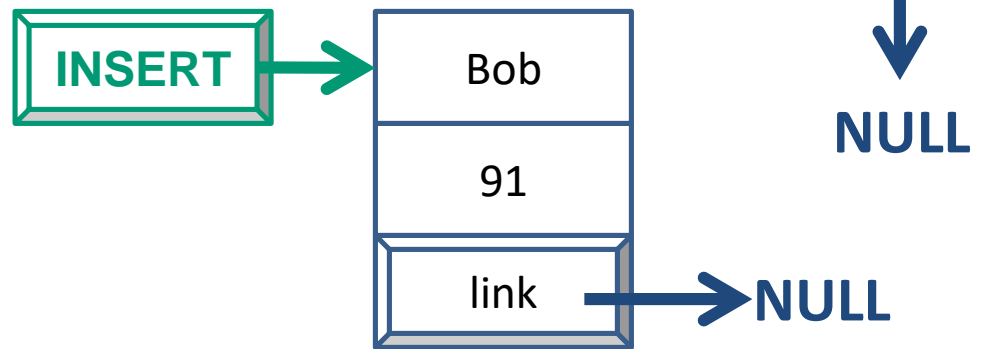
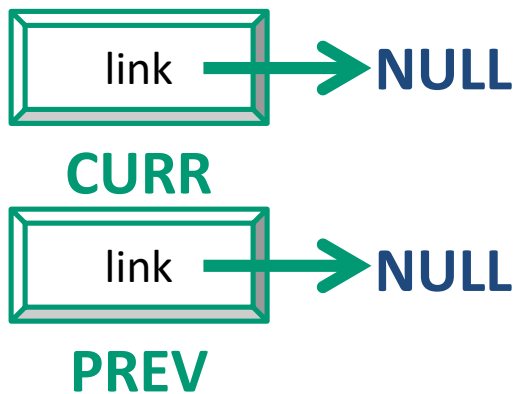
- If the Linked List is empty, what do we do?
- **FRONT** and **REAR** point to the new Node
- What else should we do?

```
void Linked_List::insert (String name, int score) {  
    // previous code...  
    if ( isEmpty() ) {  
        FRONT = INSERT;  
        REAR  = INSERT;  
    }  
    INSERT = NULL;  
}
```

# Current State of Linked List **test**



Let's create another new Node that we want to insert into our Linked List



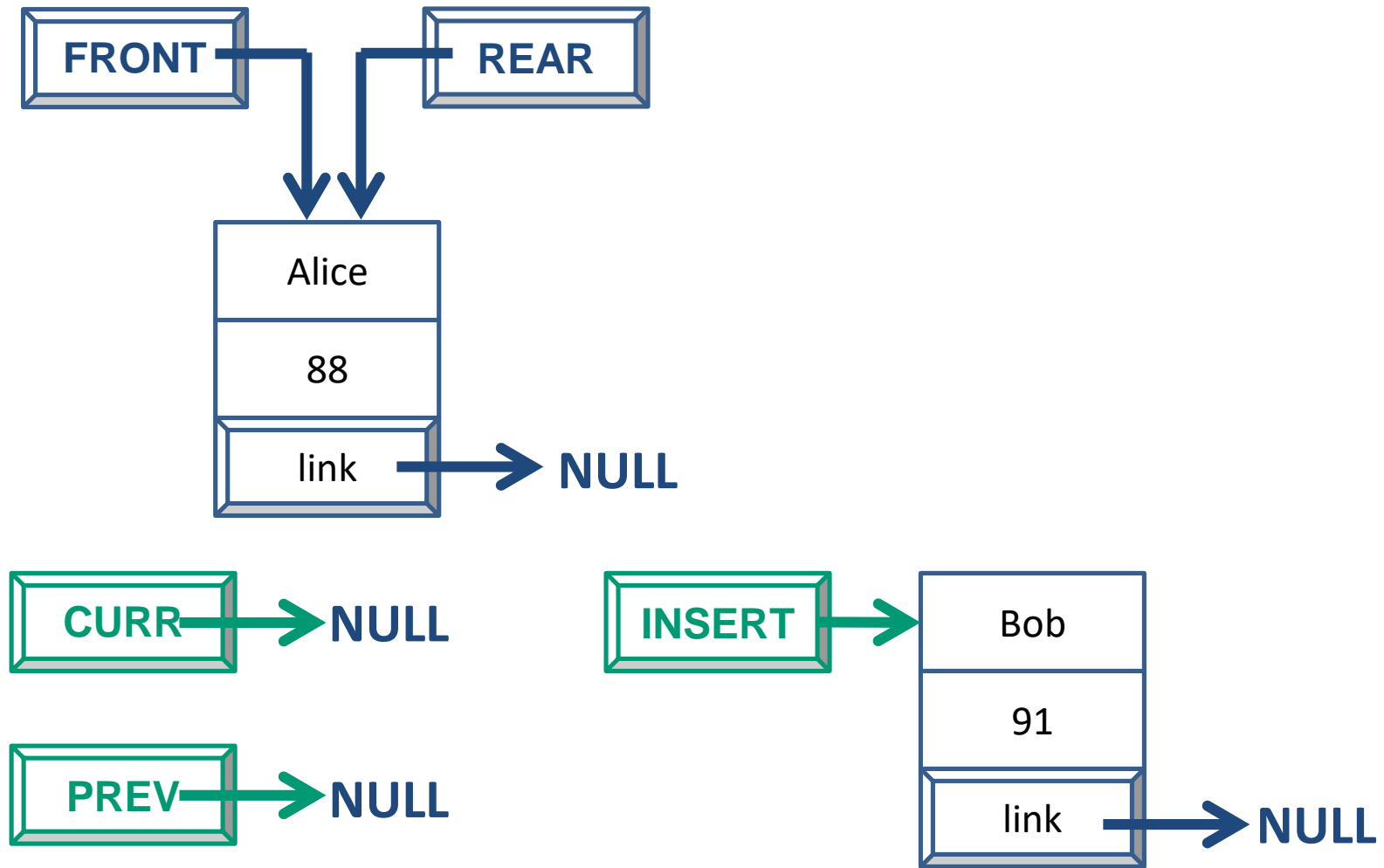
# Insertion: Non-Empty Linked List Case

- Now that the Linked List is not empty, how does our insert() function change?
  - Let's trace these changes

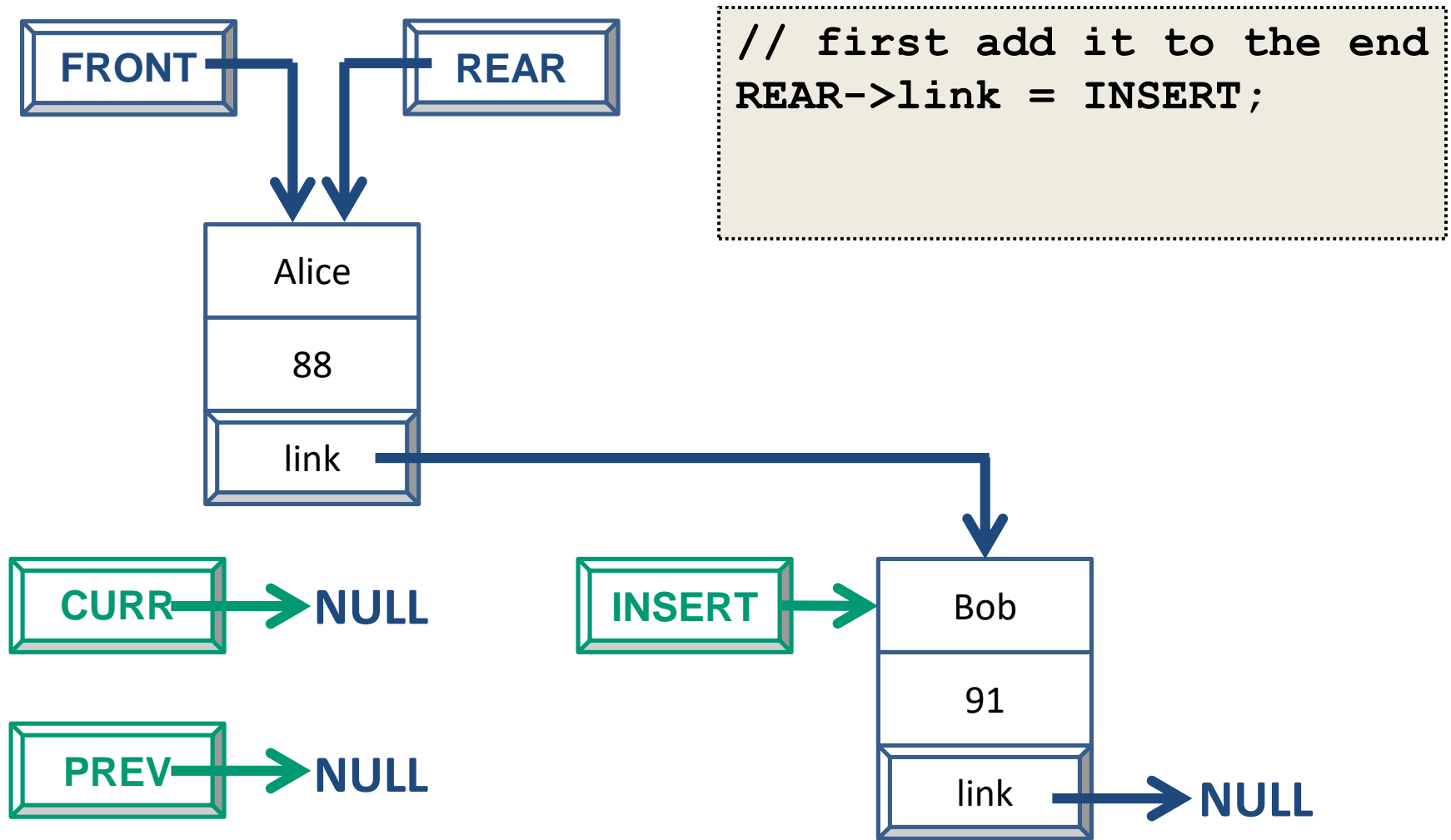
```
void Linked_List::insert (String name, int score) {
    ... // previous code for empty list
    else {
        // first add it to the end of the list
        REAR->link = INSERT;
        // then update REAR to point to the new last
        REAR = INSERT;
    }
    // rest of code...
```



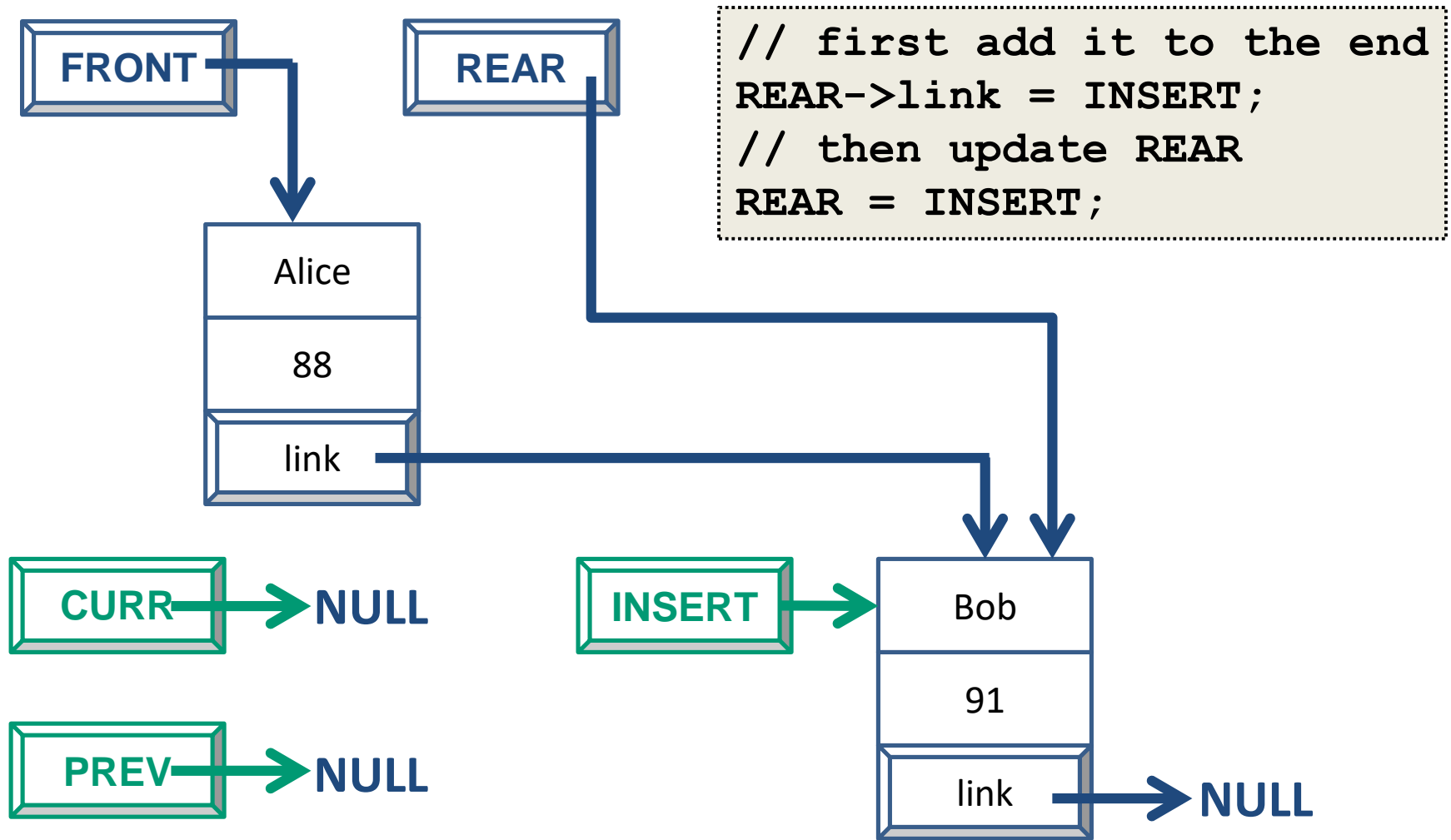
# Current State of Linked List **test**



# Current State of Linked List **test**



# Current State of Linked List **test**

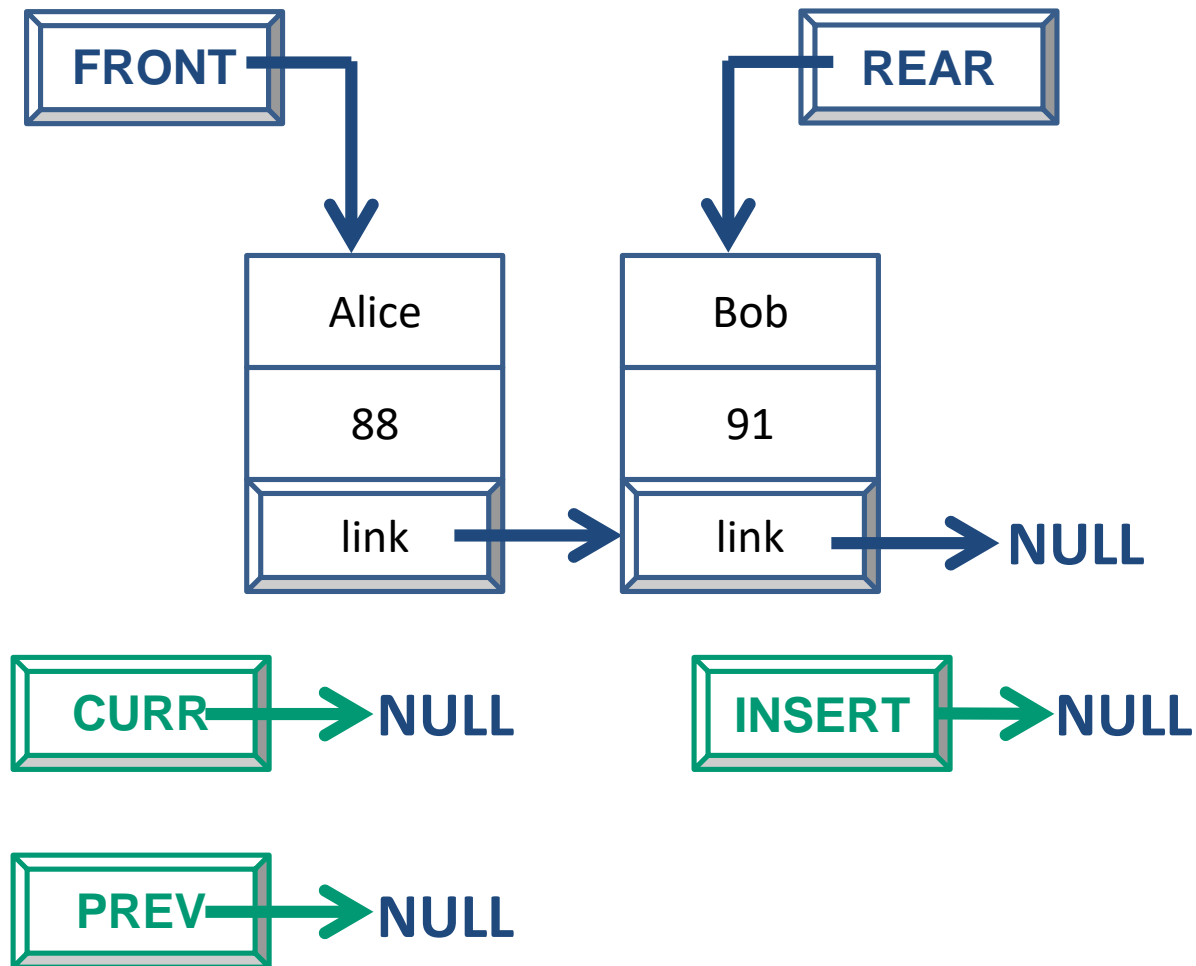


# Final `insert()` Code

- Combine the `REAR = INSERT` from the `if` and `else` statements

```
void Linked_List::insert (String name, int score) {
    INSERT = new Node()
    // initialize data
    INSERT.setName (name);
    INSERT.setGrade(score);
    if ( isEmpty() ) {
        FRONT = INSERT;           // update for first item
    } else {
        REAR->link = INSERT; // add to end of list
    }
    REAR    = INSERT;           // update end of list
    INSERT = NULL;             // reset INSERT
}
```

# Current State of Linked List **test**



---

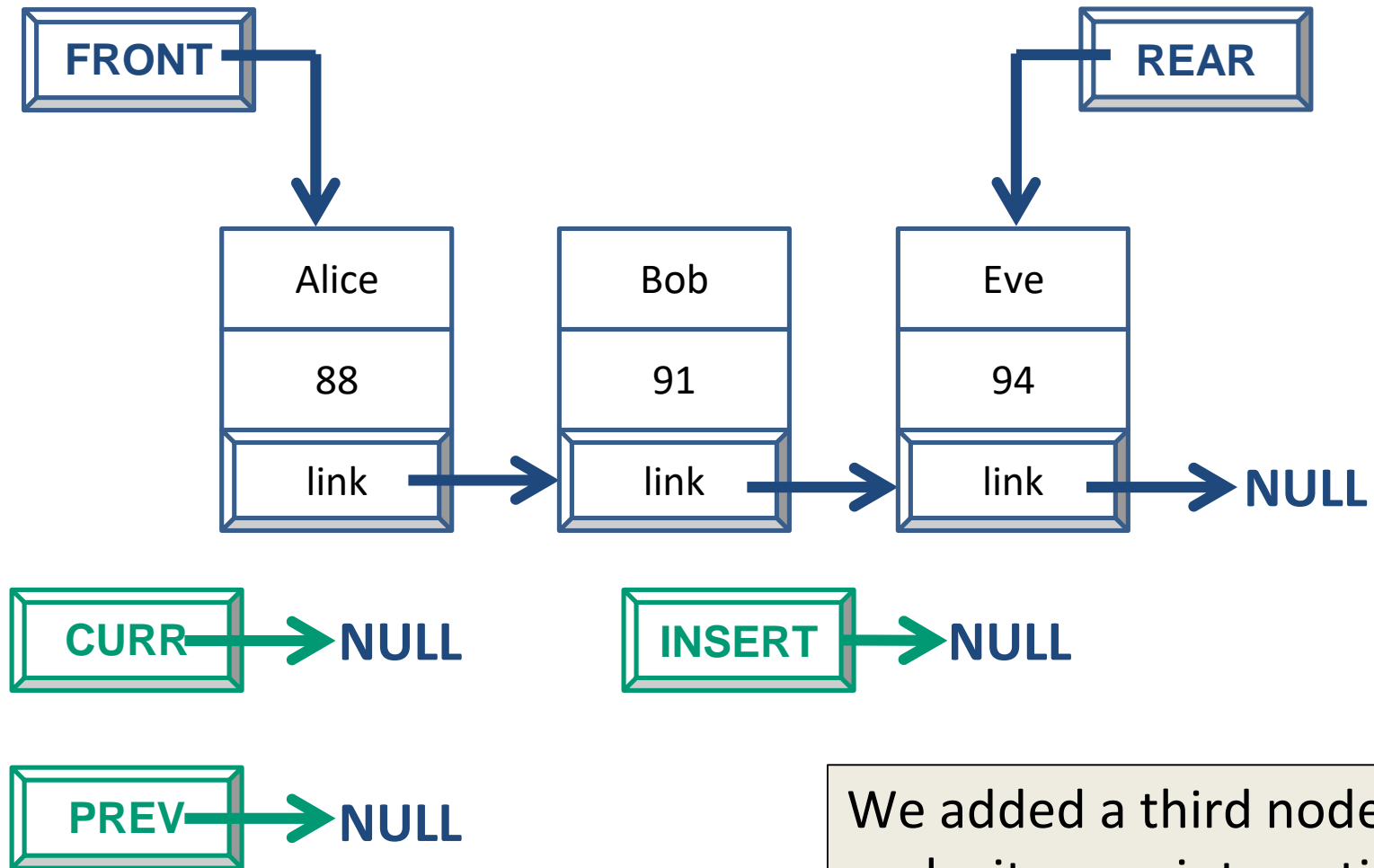
# Traversal of a Linked List

---

# Traversing the Linked List

- When would we need to traverse our list?
  - Printing out the contents
  - Searching for a specific node
  - Deleting a node
  - Counting the size of the list
    - (Better done with an updated member variable)
- We'll show the code for printing the list

# Our Linked List Now



We added a third node to make it more interesting.



# Before Traversing the Linked List

- What do we do first?
  - Check to see if the Linked List is empty
  - If it is, what should we do?
    - Print out a message
    - Return from the function

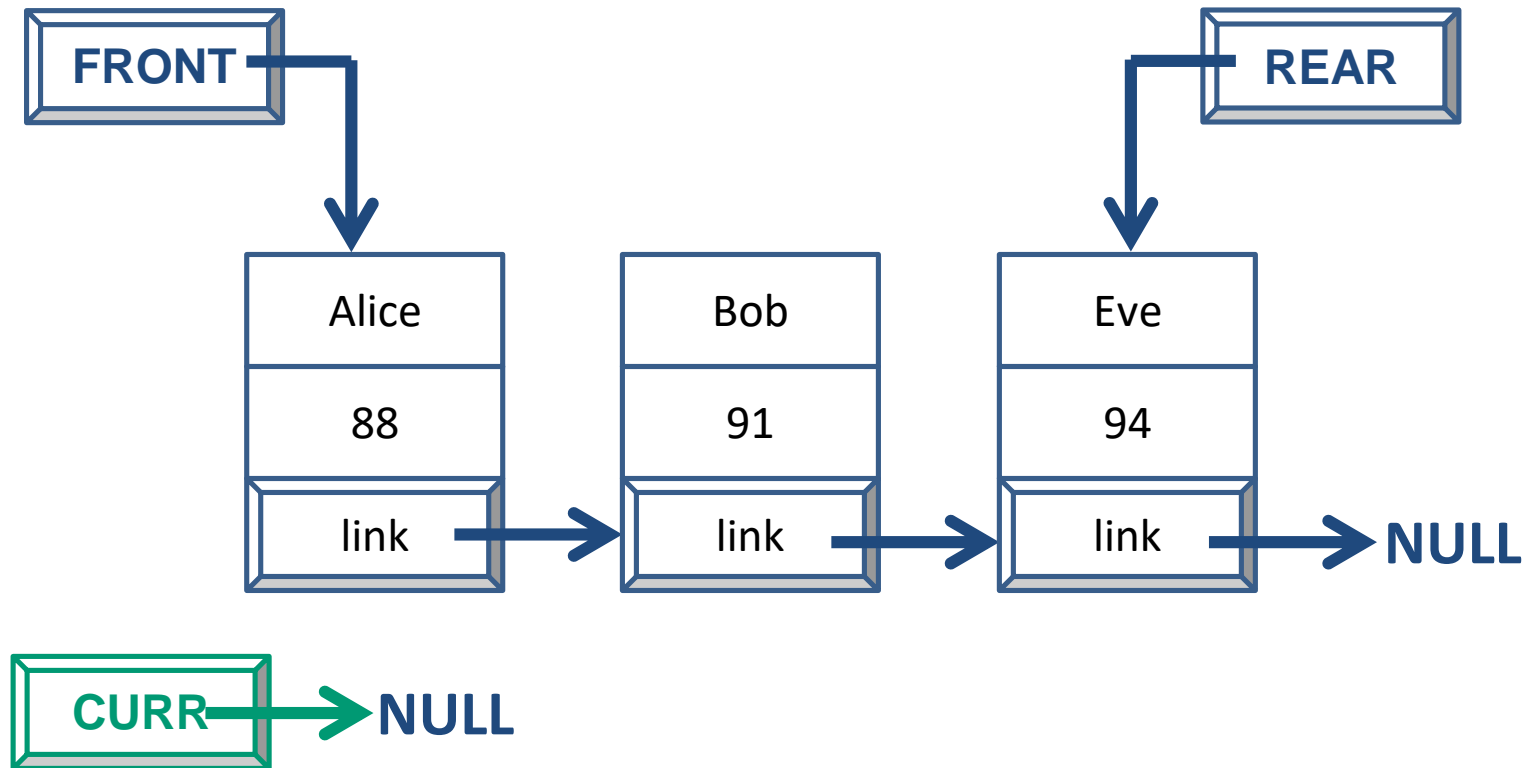
```
void Linked_List::printList() {  
    if ( isEmpty() ) {  
        cout << "This list is empty!";  
        return;  
    }  
    // rest of the function
```

---

# Planning out the Traversal

- If the Linked List is not empty, then we begin traversing the Linked List
  - How do we start?
  - How do we know when to stop?
  - How do we move from one node to another?
  - *Hint: Using CURR alone will work for this*
- Take a look at the diagram again, and think about the steps we need to take

# Exercise: Traversing a Linked List



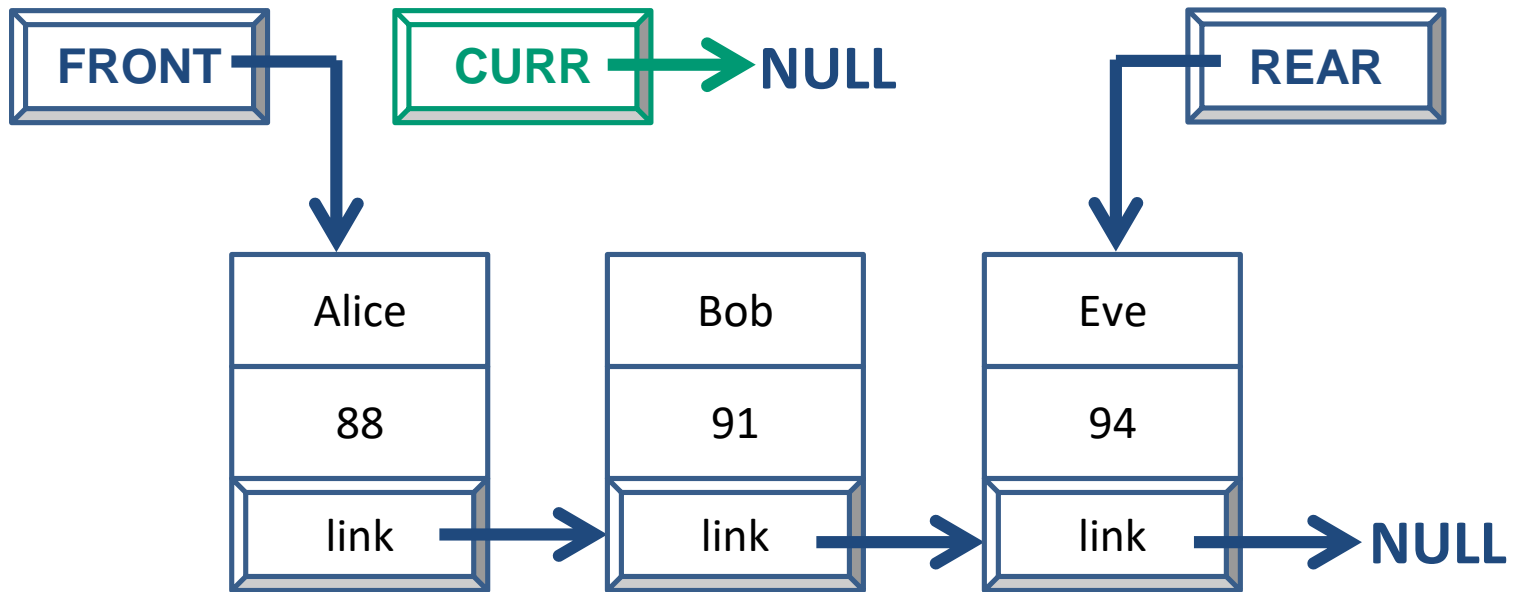
We don't need **INSERT** or **PREV** to traverse the Linked List.

# Traversing the List

- To control our traversal, we'll use a loop
  - Initialization, Termination Condition, Modification
    1. Set **CURR** to the first node in the list
    2. Continue until we hit the end of the list (**NULL**)
    3. Move from one node to another (using **link**)

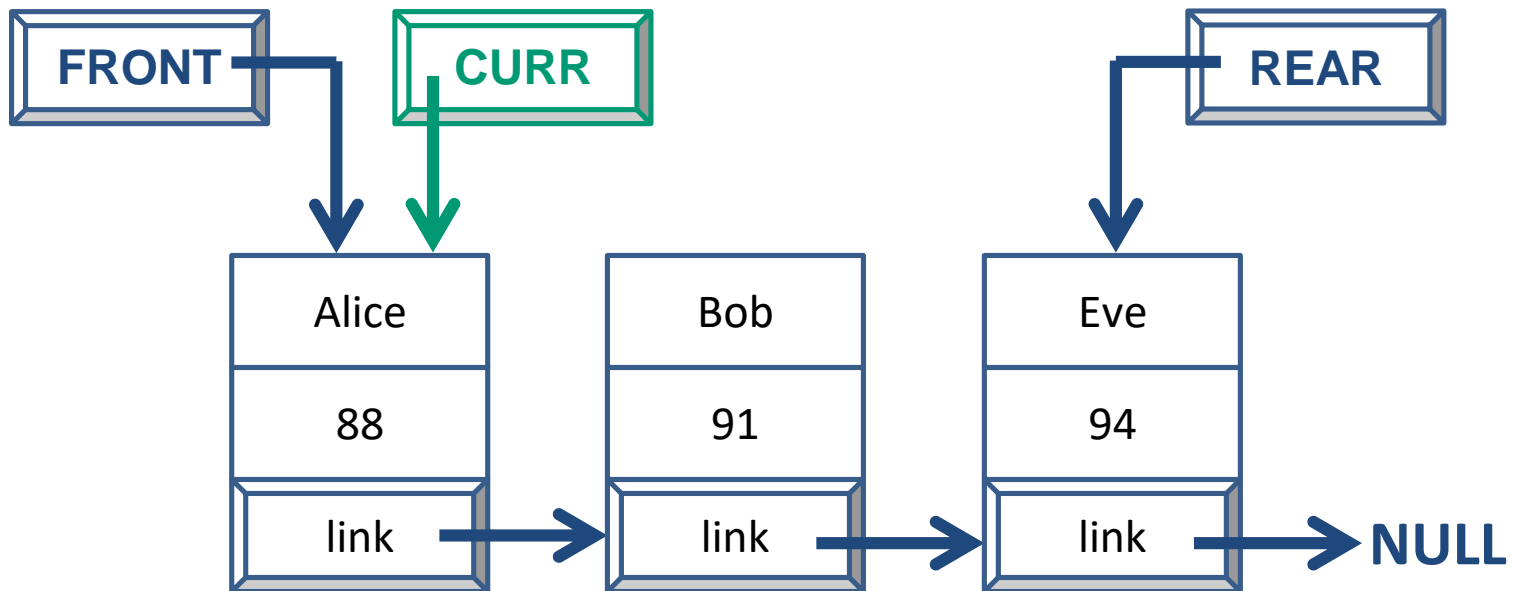
```
void Linked_List::printList() {  
    // prev code (checking if empty)  
    for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {  
        // print the information  
        cout << "Name is " << CURR->getName() << endl;  
        cout << "Grade is " << CURR->getGrade() << endl;  
    }  
}
```

# Demonstration of Traversal



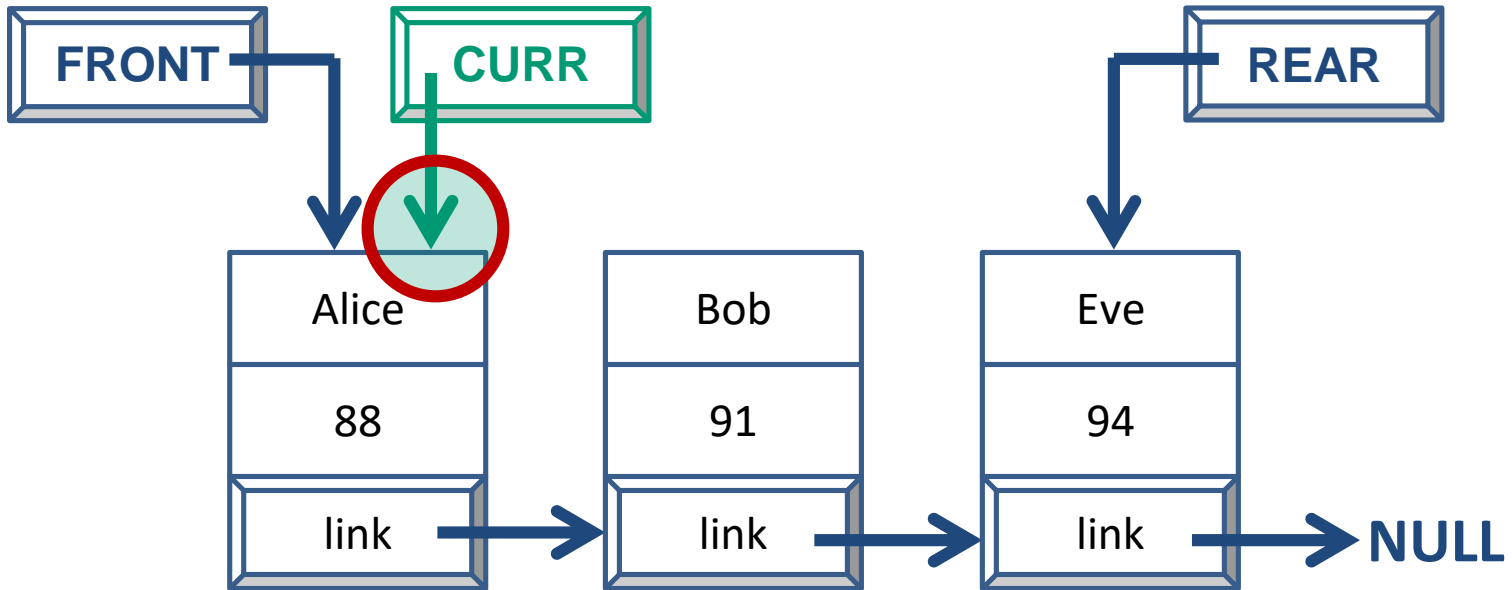
```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal



```
for (CURR = FRONT, CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal

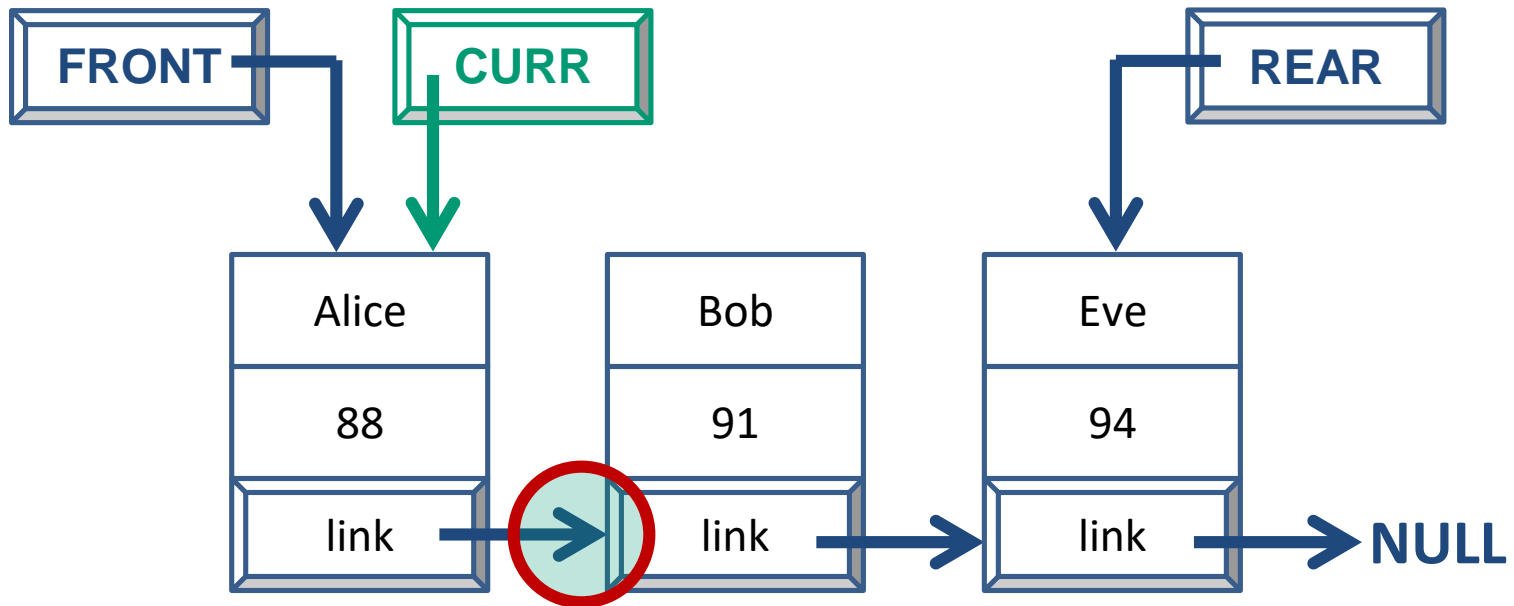


```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```



```
// print information (Alice)
```

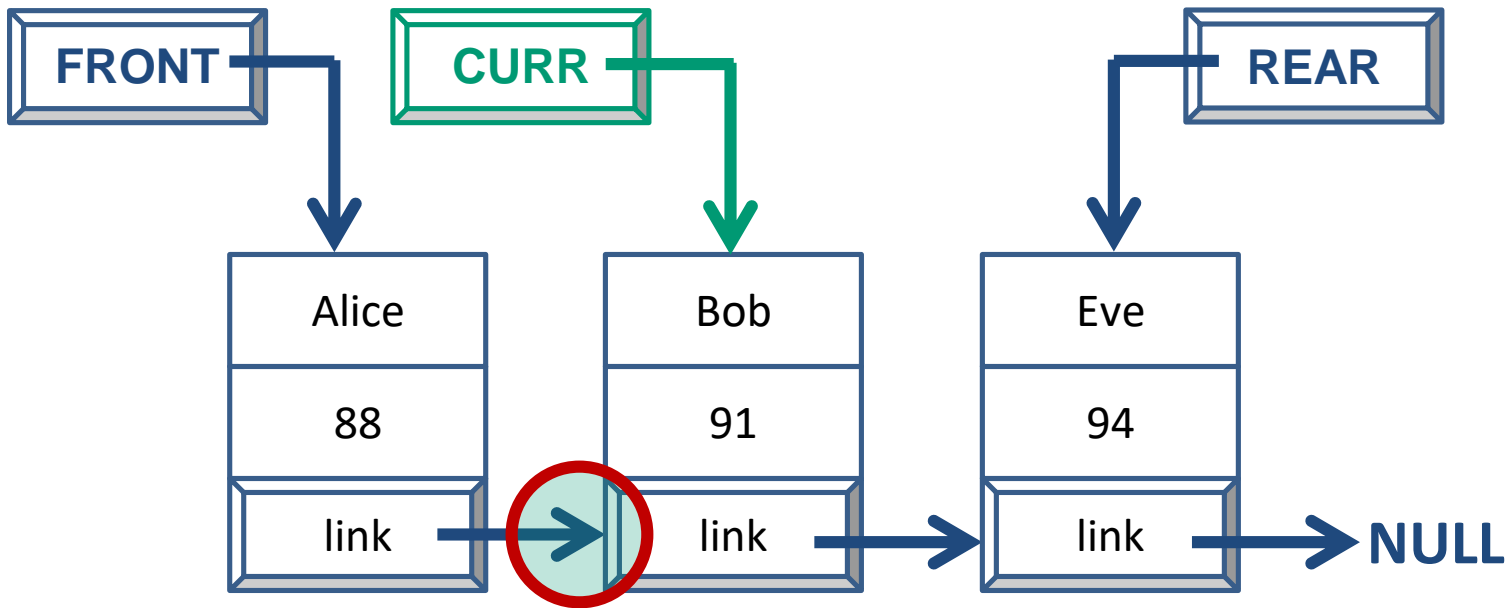
# Demonstration of Traversal



```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

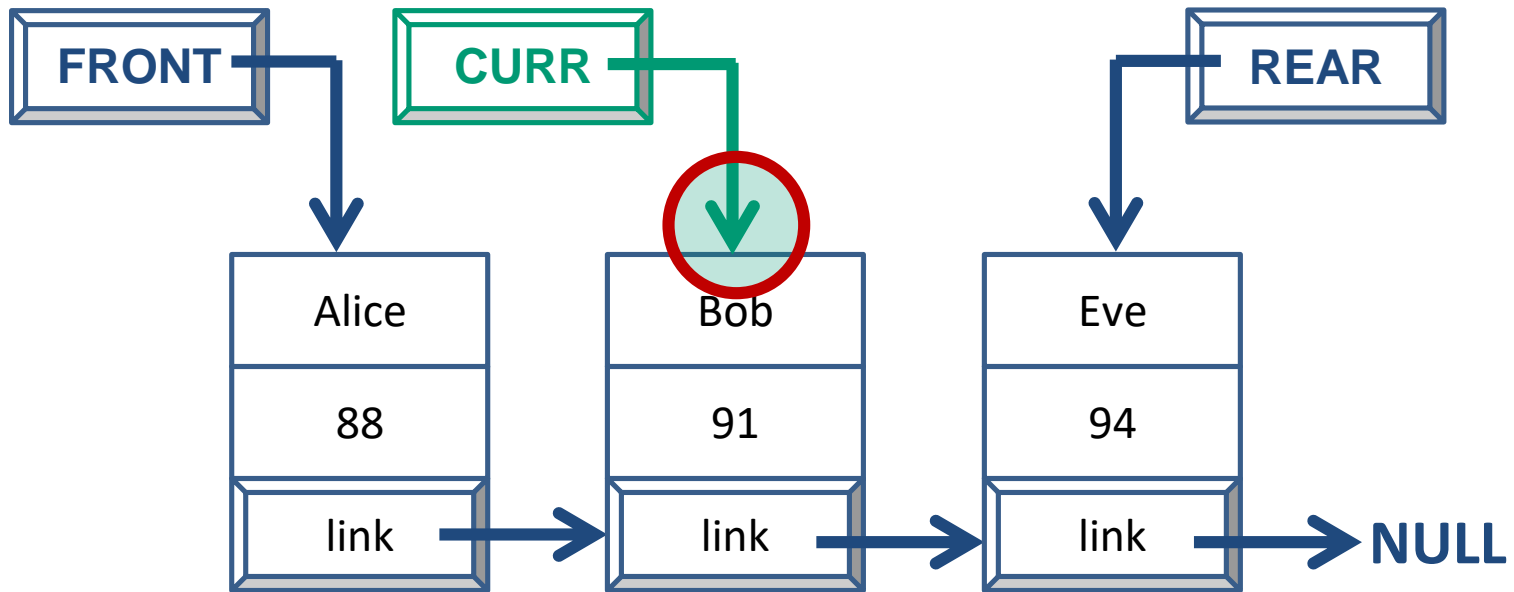


# Demonstration of Traversal



```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal

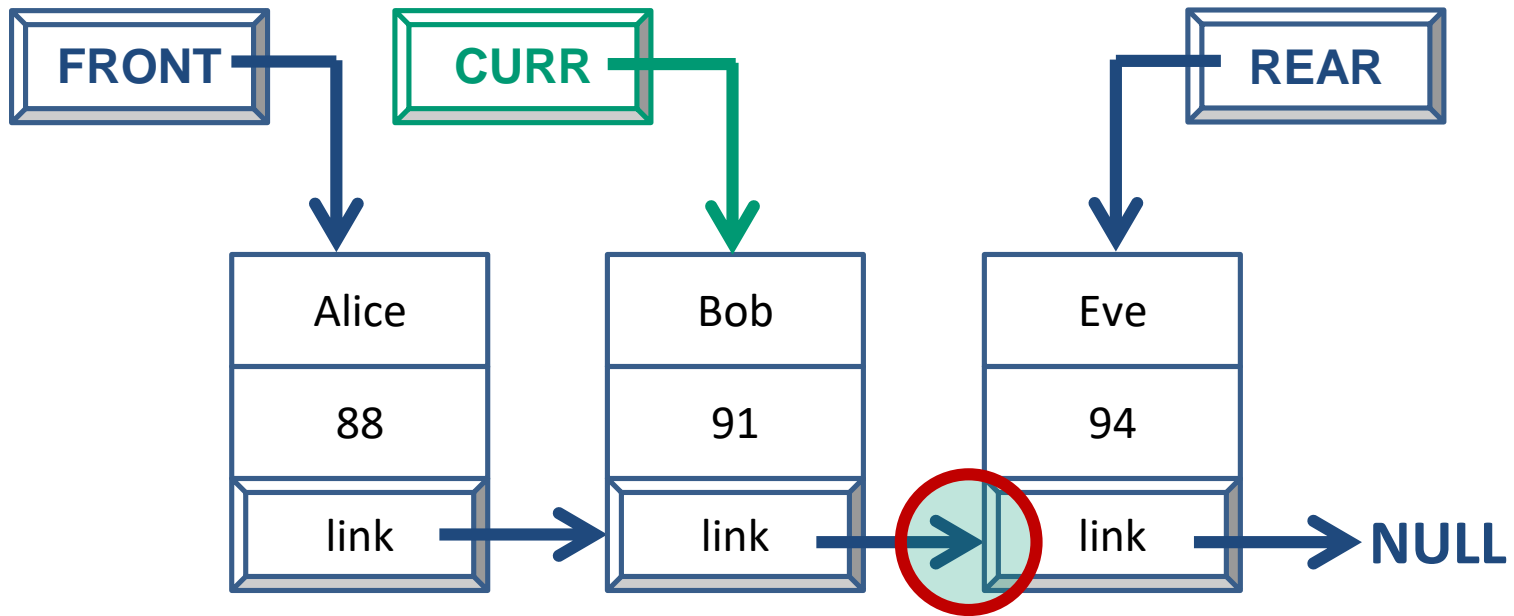


```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```



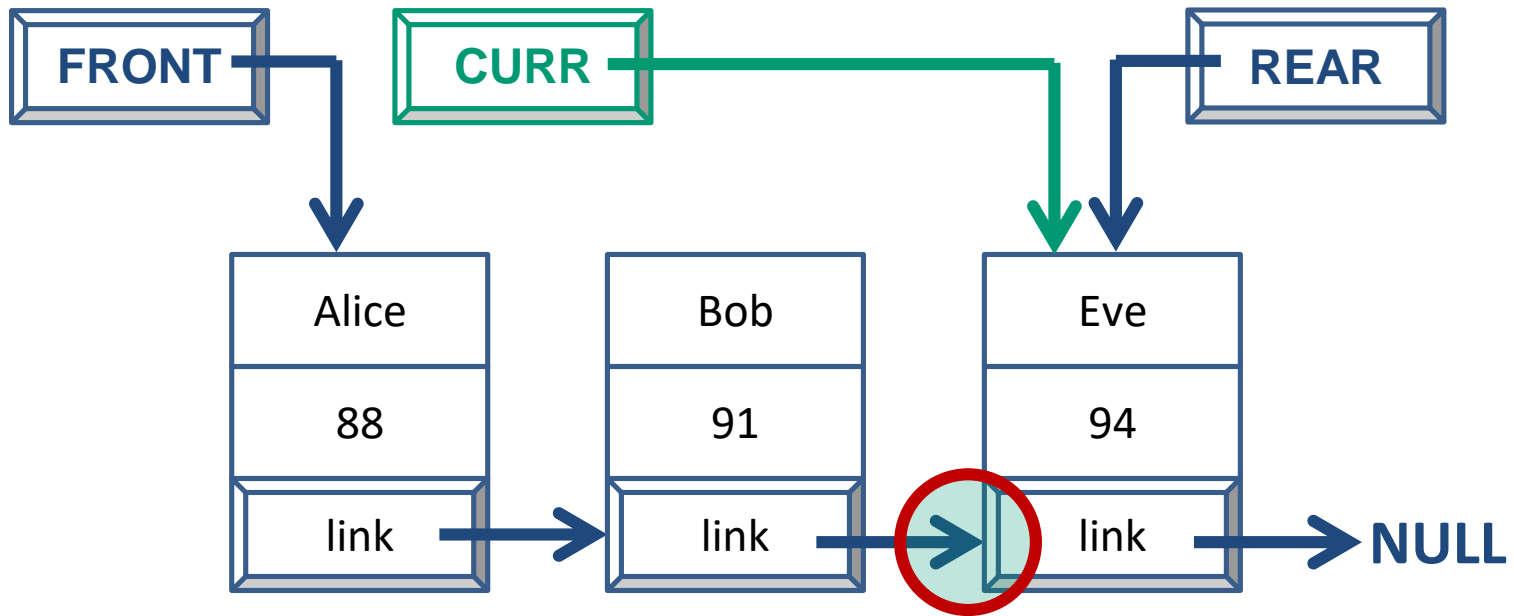
```
// print information (Bob)
```

# Demonstration of Traversal



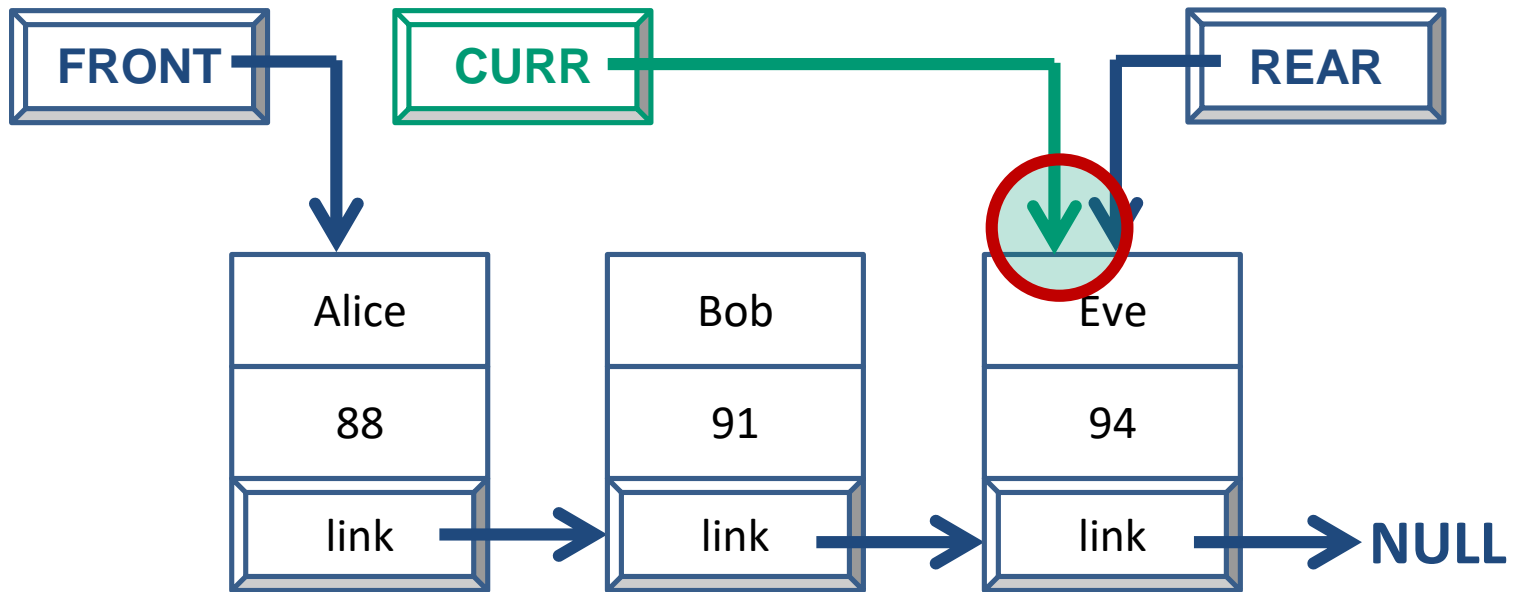
```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal



```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal

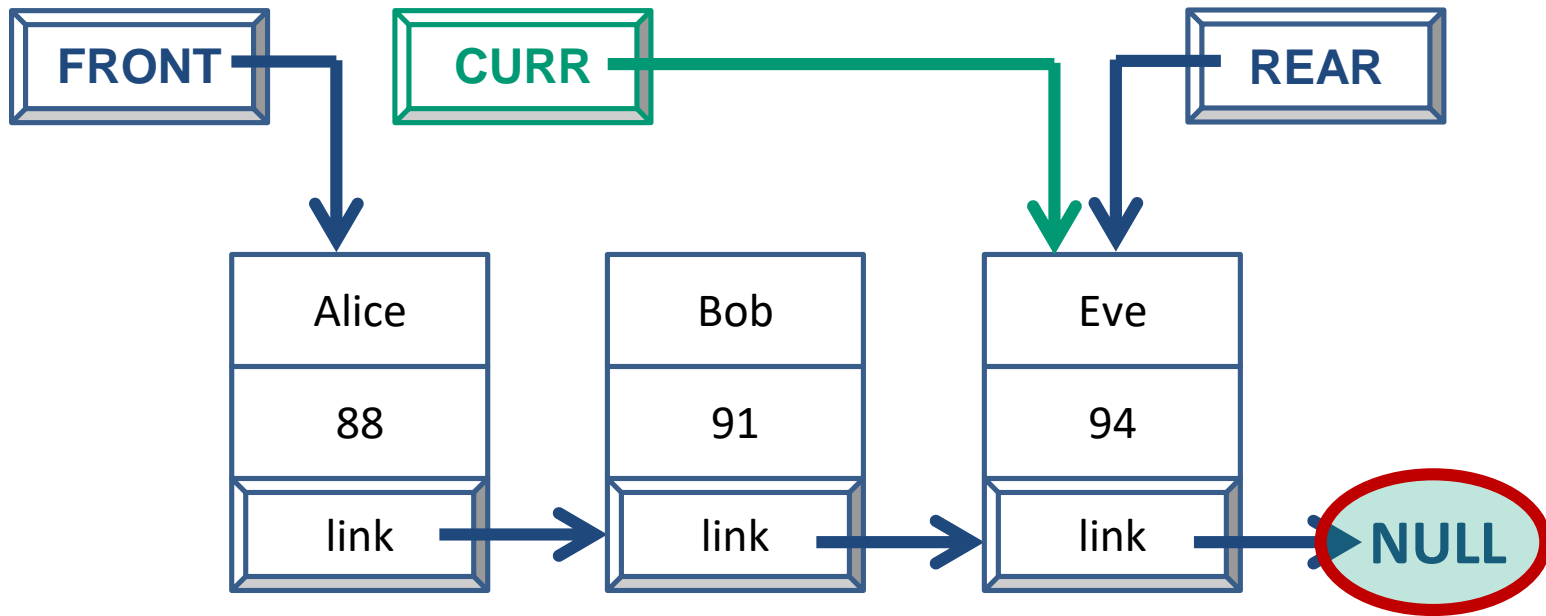


```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```



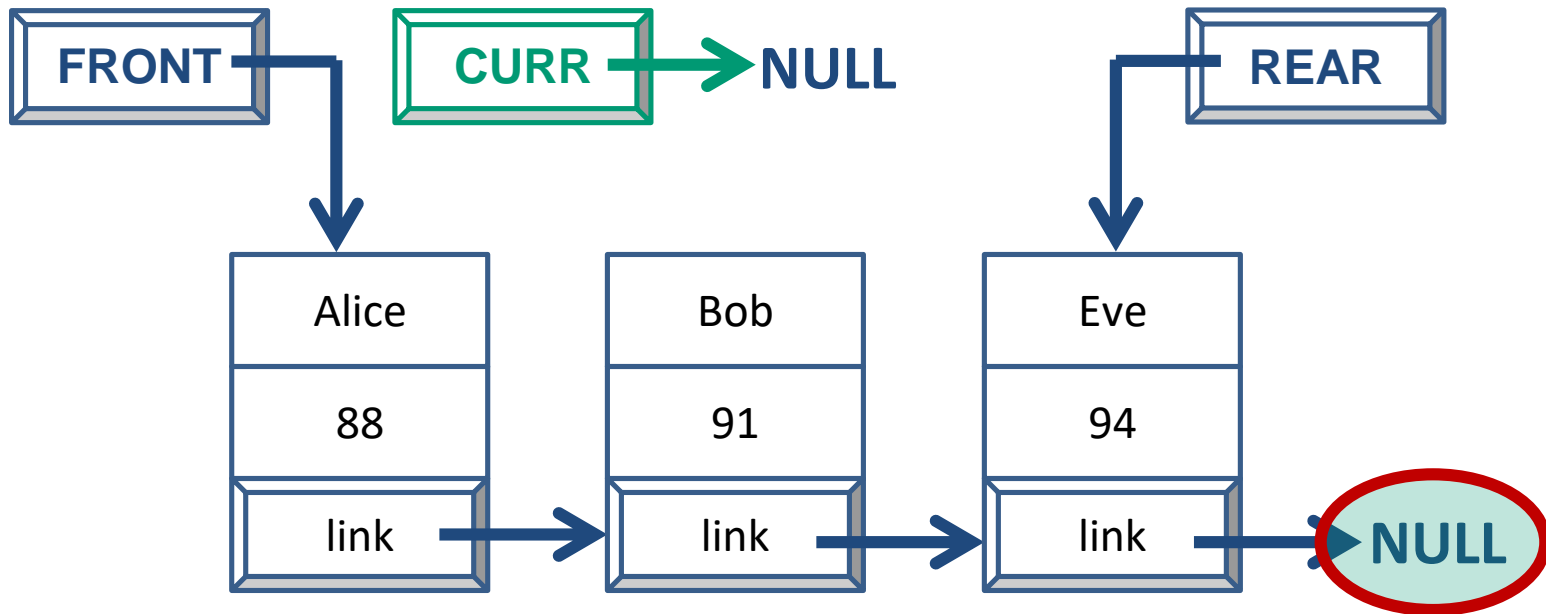
```
// print information (Eve)
```

# Demonstration of Traversal



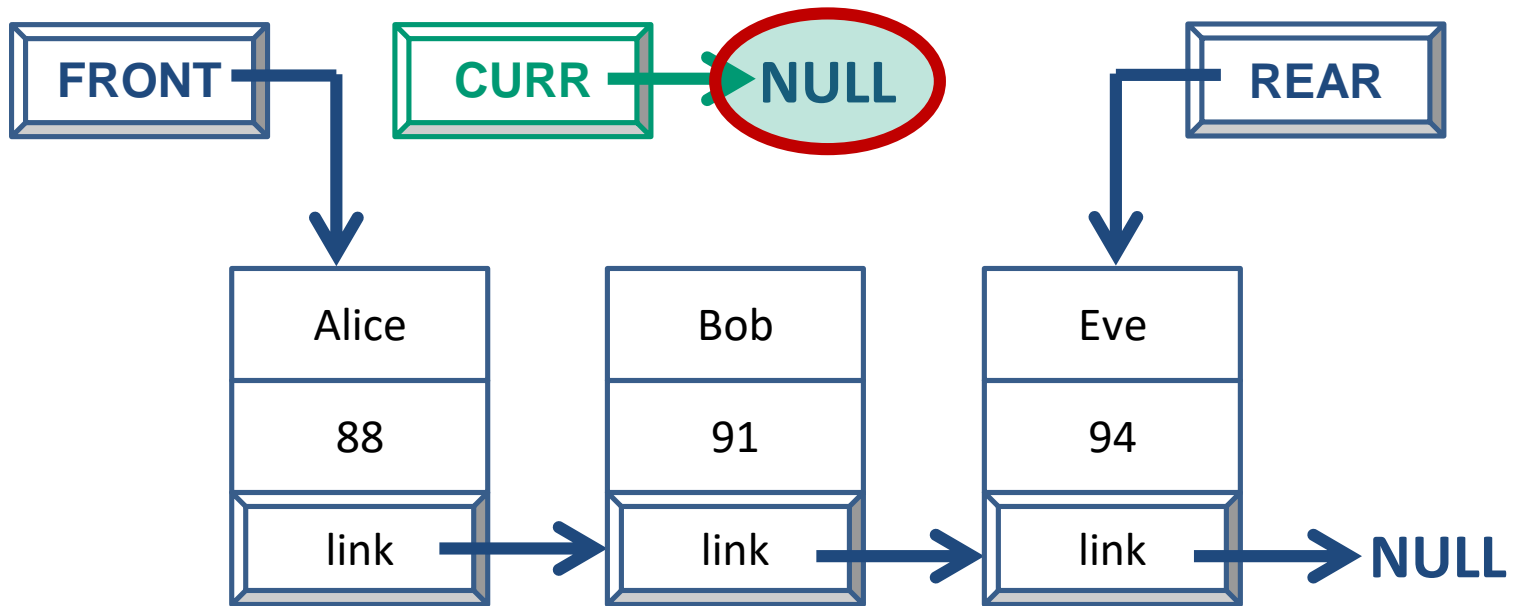
```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal



```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

# Demonstration of Traversal



```
for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
```

**X**

```
} // exit the loop
```



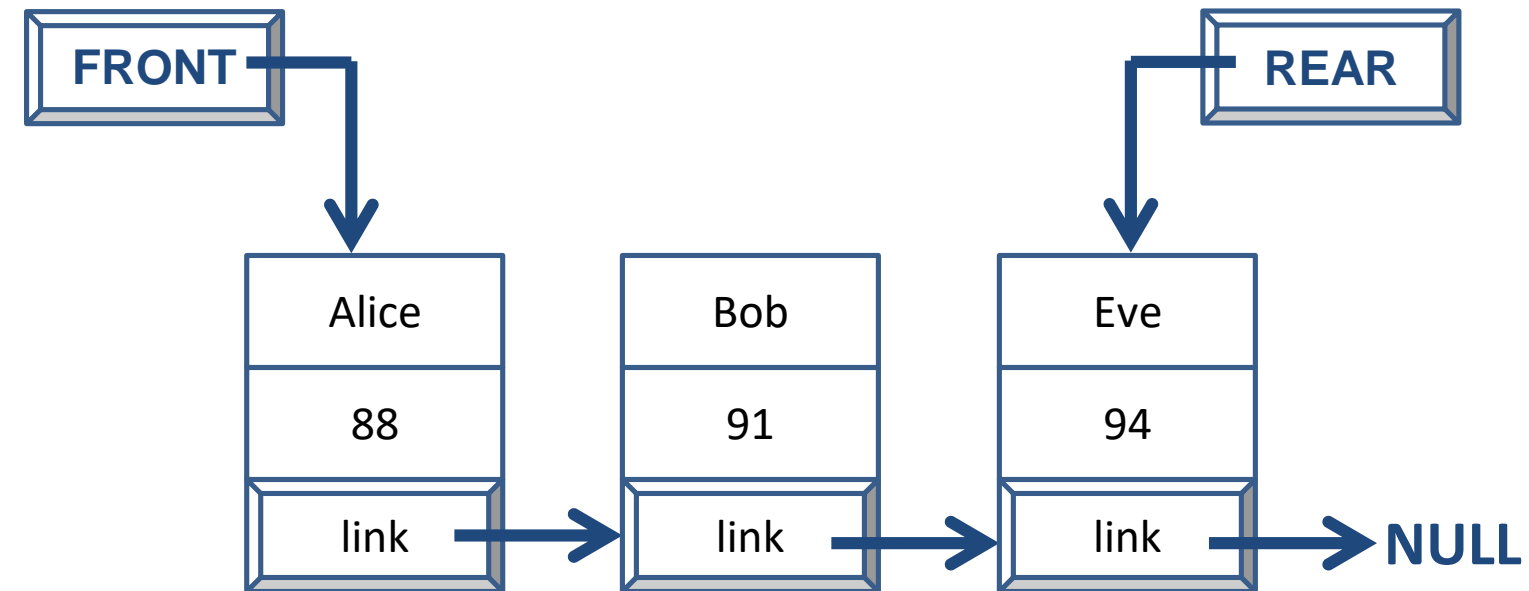
---

# Deletion from a Linked List

# Special Cases

- Deletion has many special cases, including...
  - Deleting the only node
  - Deleting the last node
  - Deleting the first node
  - Deleting any “middle” node
- We will need to use **CURR** and **PREV** here
  - Why? What will we use **PREV** for?

# Exercise: Deleting from a Linked List



**CURR** → NULL

**PREV** → NULL

What steps would you take to remove Bob? And then Alice?

---

# Traversing for Deletion

- We will use **CURR** and **PREV** to keep track of where we are in the Linked List
- We will search for the target
  - If found, we will delete the node
    - And update the **link** of the node before it
  - If not found, we will return **False**
    - If we reach the end of the list (**NULL**)

# Looking at the Code

```
boolean Linked_List::remove(String target) {
    CURR = PREV = NULL;

    for (CURR = FRONT; CURR != NULL; CURR = CURR->link) {
        if (CURR->name == target) {
            // WE MADE A MATCH!
            // here's where the deletion will happen
            return true;
        } else {
            PREV = CURR;
            // the for loop will move CURR to next node
        }
    }
    return false;
}
```

# Deletion Code

- What are the three possible locations?
  1. First node in the list
  2. Last node in the list
  3. Node in the middle of the list

```
if (CURR->name == target) {  
    // WE MADE A MATCH!  
  
    if (CURR == FRONT) {}           // first node  
    else if (CURR == REAR ) {}      // last node  
    else {}                          // middle of the list  
  
}
```

# Deletion Code

- Inside each conditional, you must first fix the links around the target node
- Then delete the target node (CURR)

```
if (CURR->name == target) {  
    // WE MADE A MATCH!  
  
    if (CURR == FRONT) {}           // first node  
    else if (CURR == REAR ) {}     // last node  
    else {}                         // middle of the list  
    delete CURR;  
}
```

---

# Order of Deletion Operations

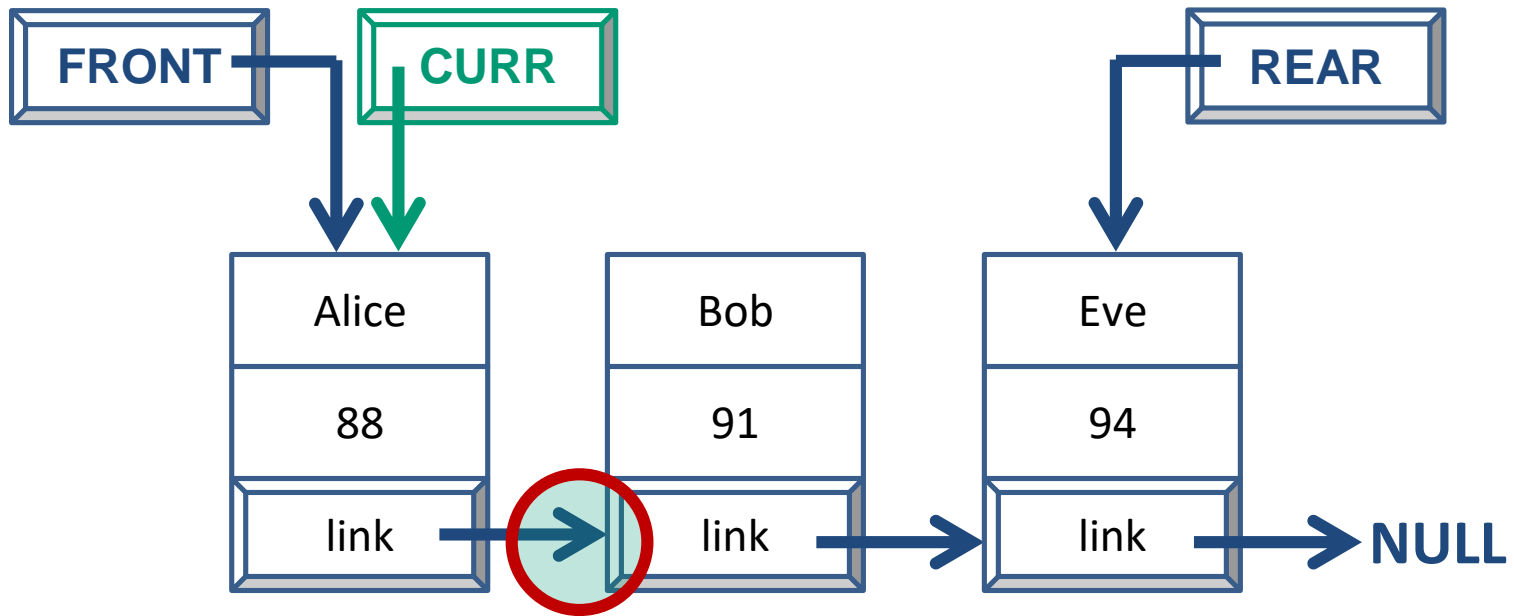
- IMPORTANT:
- Deleting a node is the **last** thing that happens
- Before deletion, you must update **all** of the other nodes that currently point to it



---

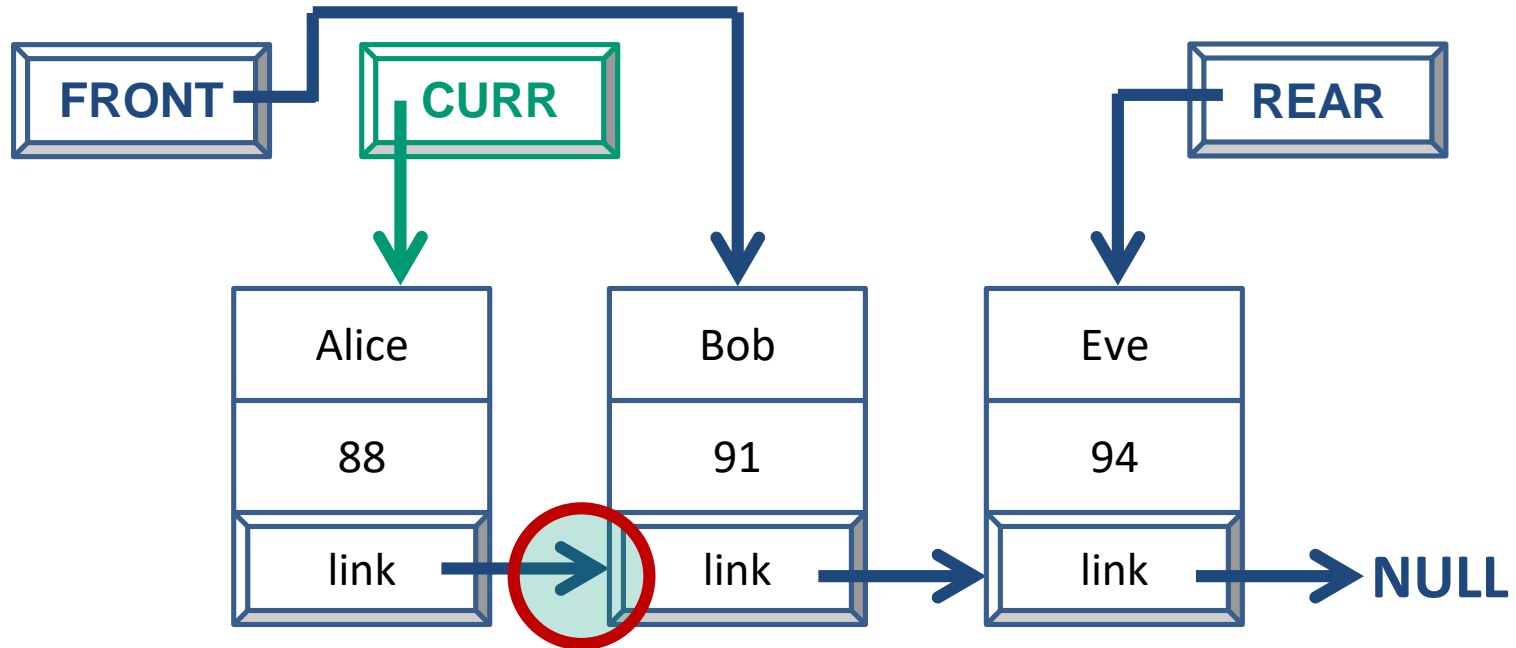
# Deletion Case 1: First Node in Linked List

# Deletion Case 1: First Node



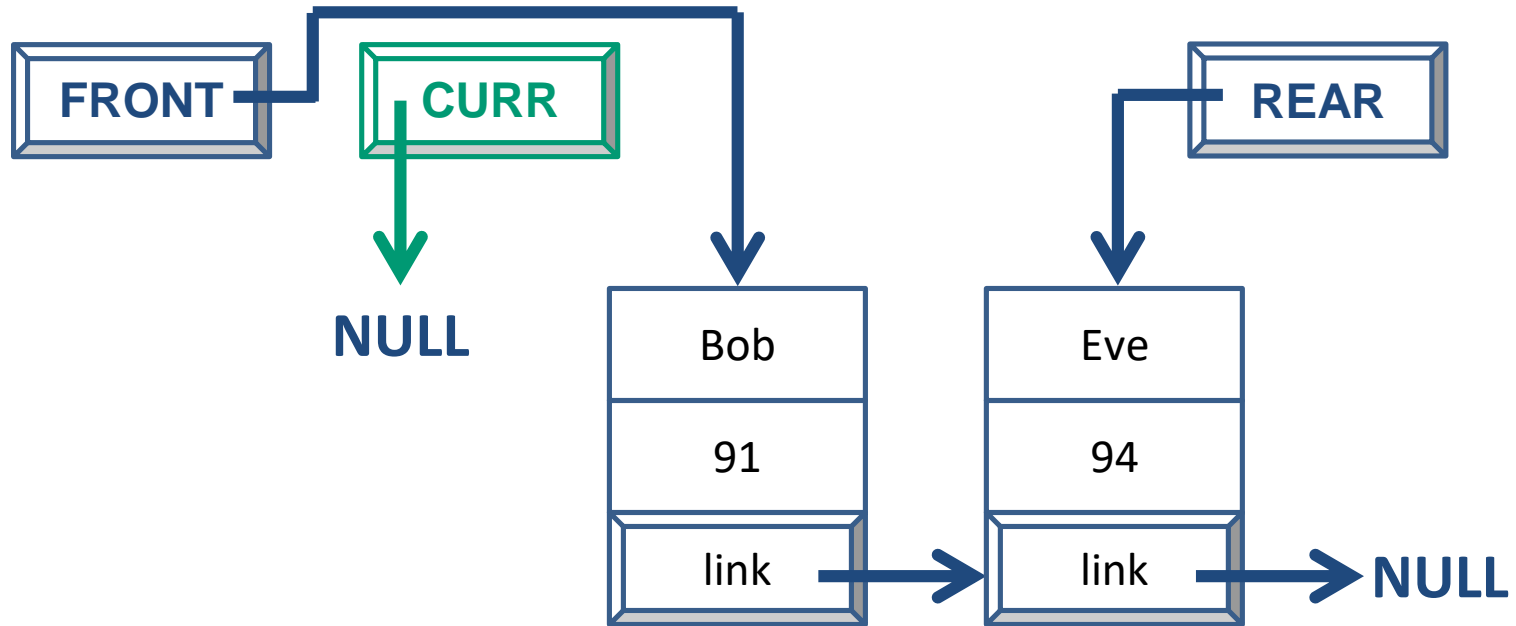
```
if (CURR == FRONT) {  
    FRONT = FRONT->link;  
}
```

# Deletion Case 1: First Node



```
if (CURR == FRONT) {  
    FRONT = FRONT->link;  
}  
delete CURR;
```

# Deletion Case 1: First Node

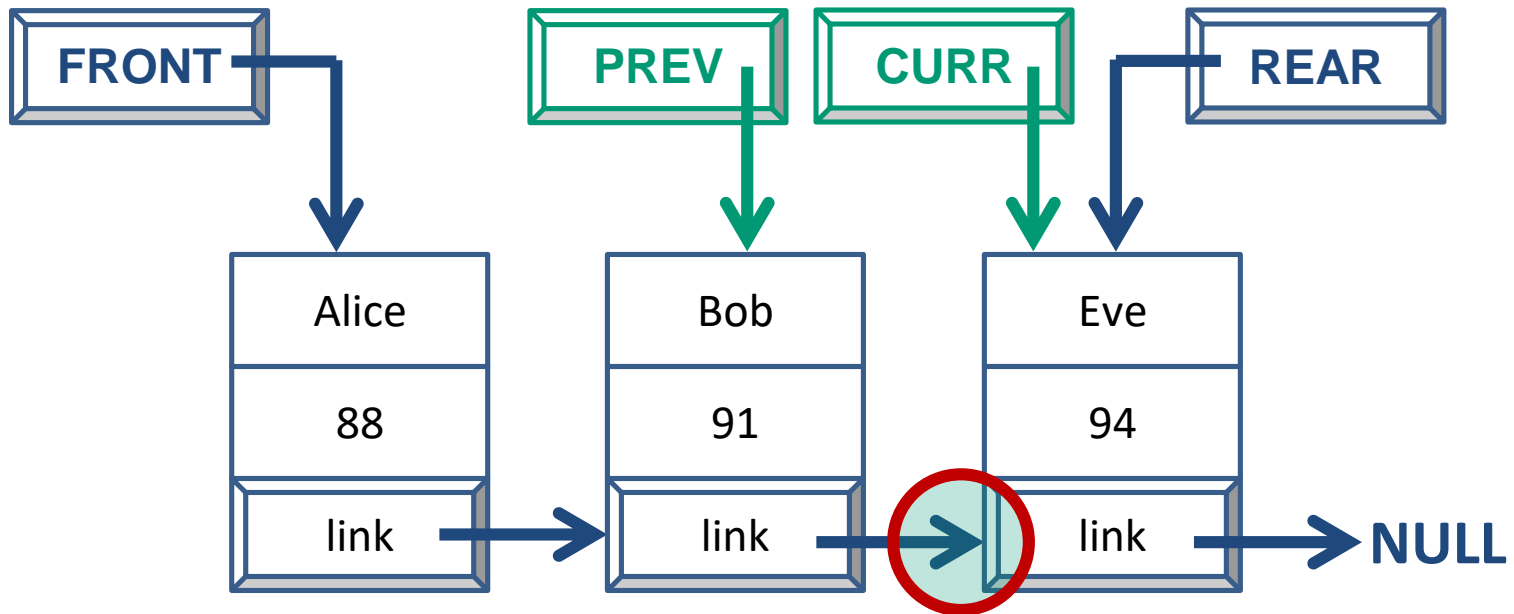


```
if (CURR == FRONT) {  
    FRONT = FRONT->link;  
}  
delete CURR;
```

---

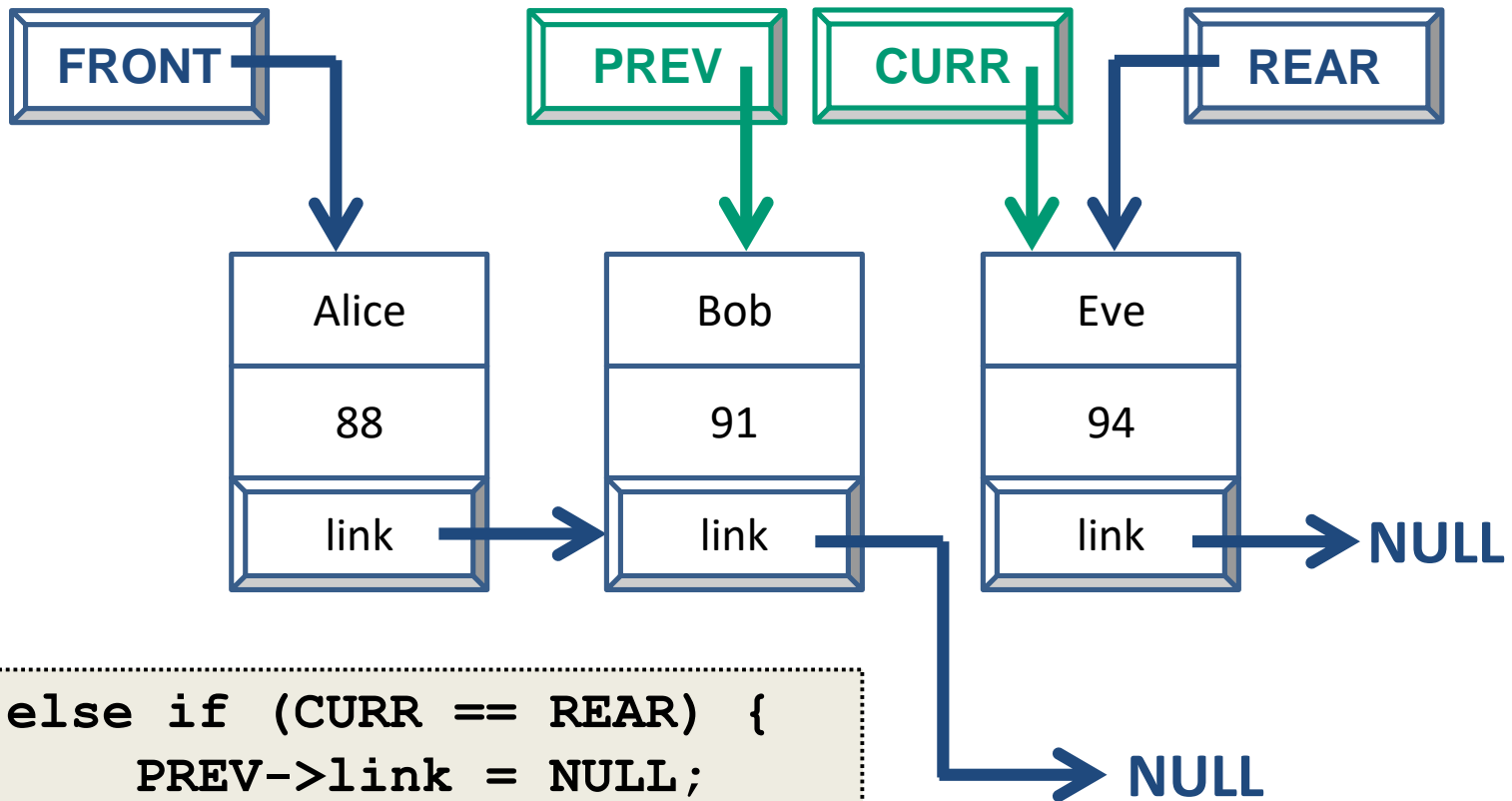
## Deletion Case 2: Last Node in Linked List

# Deletion Case 2: Last Node



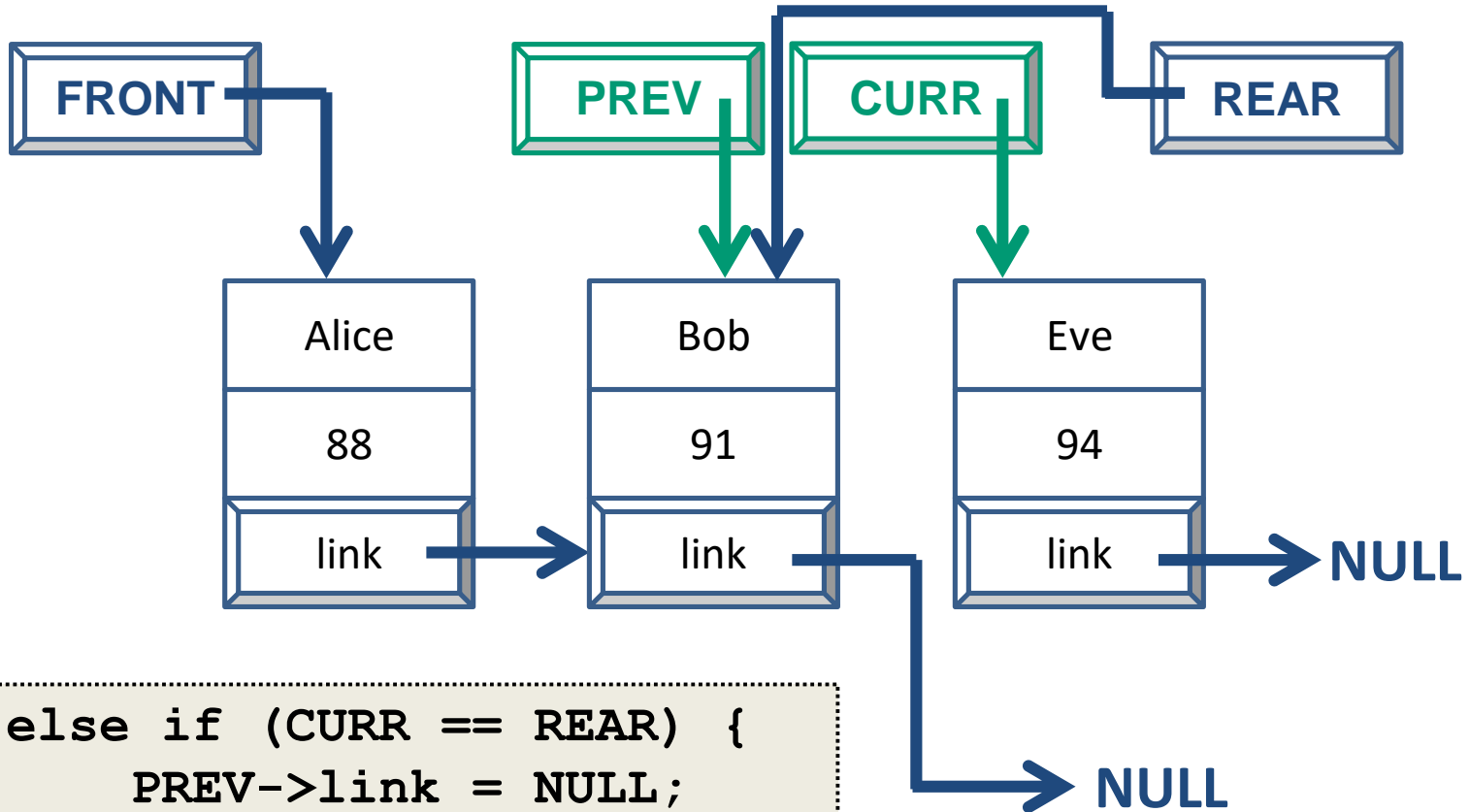
```
else if (CURR == REAR) {  
    PREV->link = NULL;  
}
```

# Deletion Case 2: Last Node



```
else if (CURR == REAR) {  
    PREV->link = NULL;  
    REAR = PREV;  
}
```

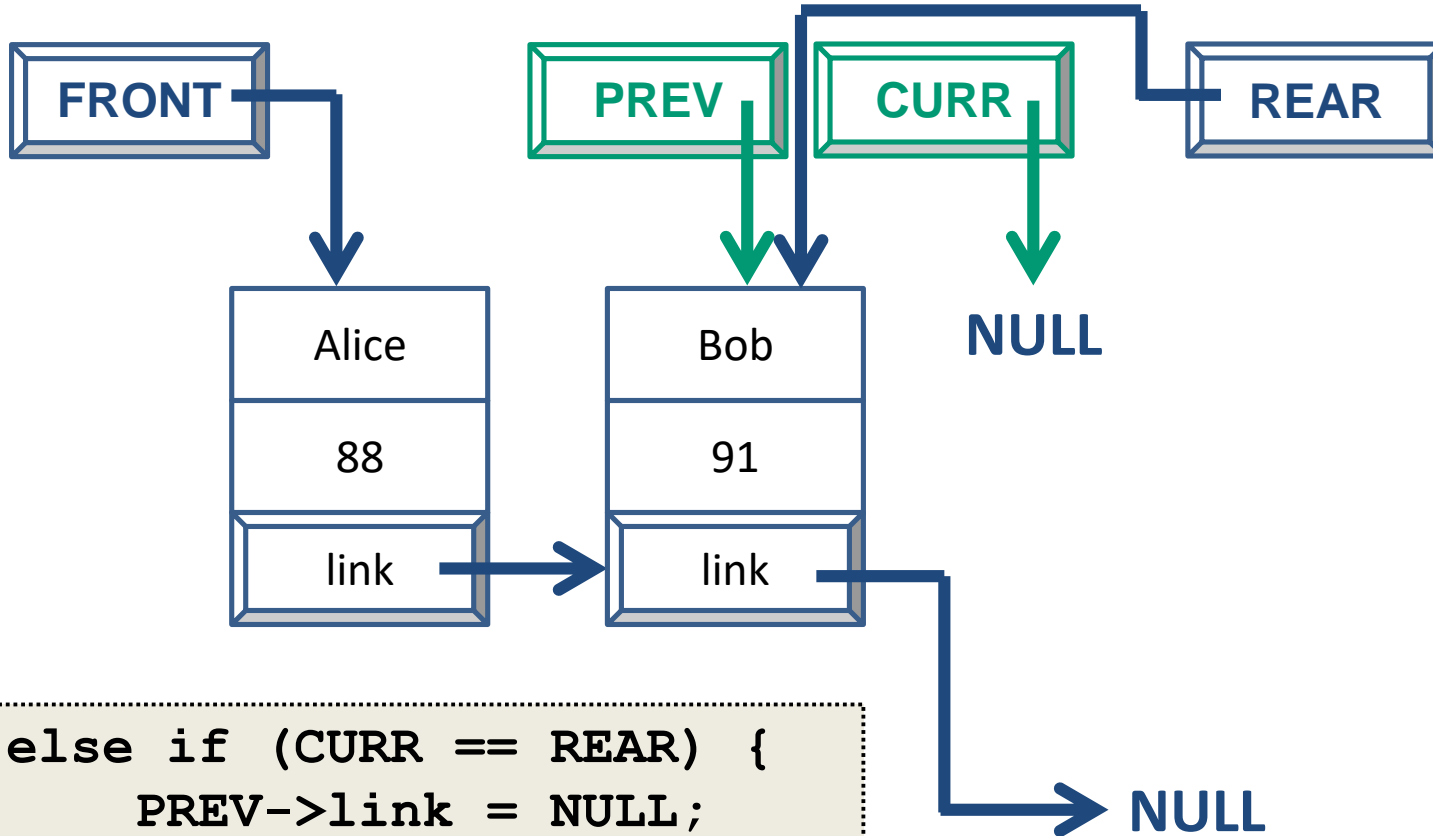
# Deletion Case 2: Last Node



```
else if (CURR == REAR) {  
    PREV->link = NULL;  
    REAR = PREV;  
}  
delete CURR;
```



# Deletion Case 2: Last Node

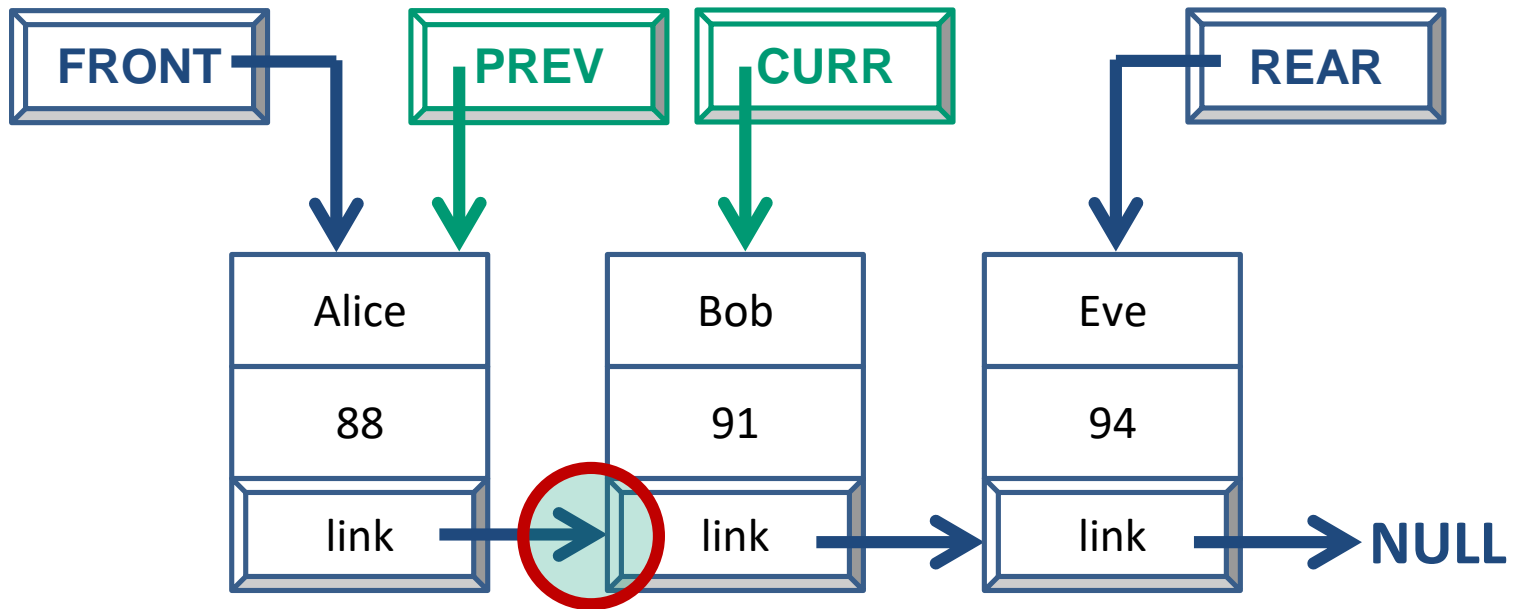


```
else if (CURR == REAR) {  
    PREV->link = NULL;  
    REAR = PREV;  
}  
delete CURR;
```

---

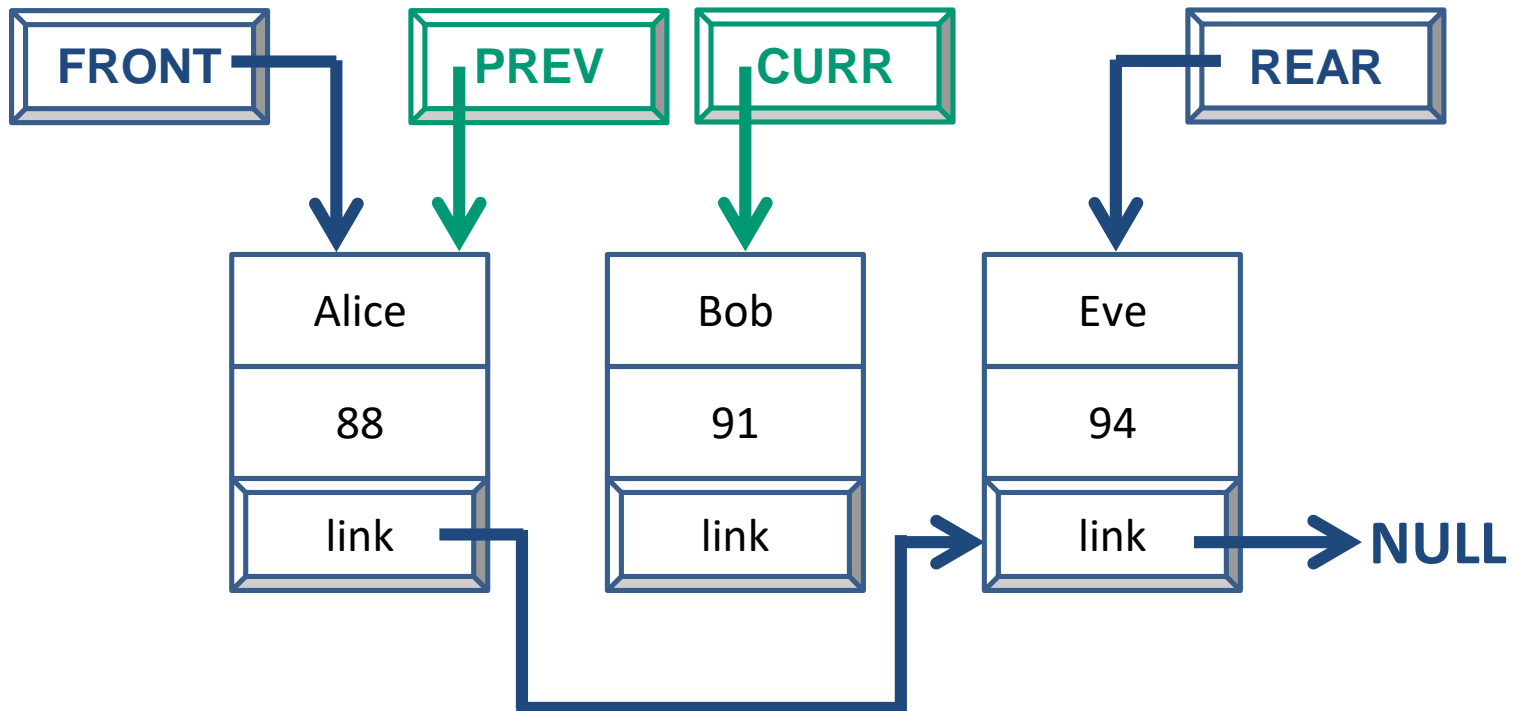
# Deletion Case 3: Node in Middle of Linked List

# Deletion Case 3: Middle Node



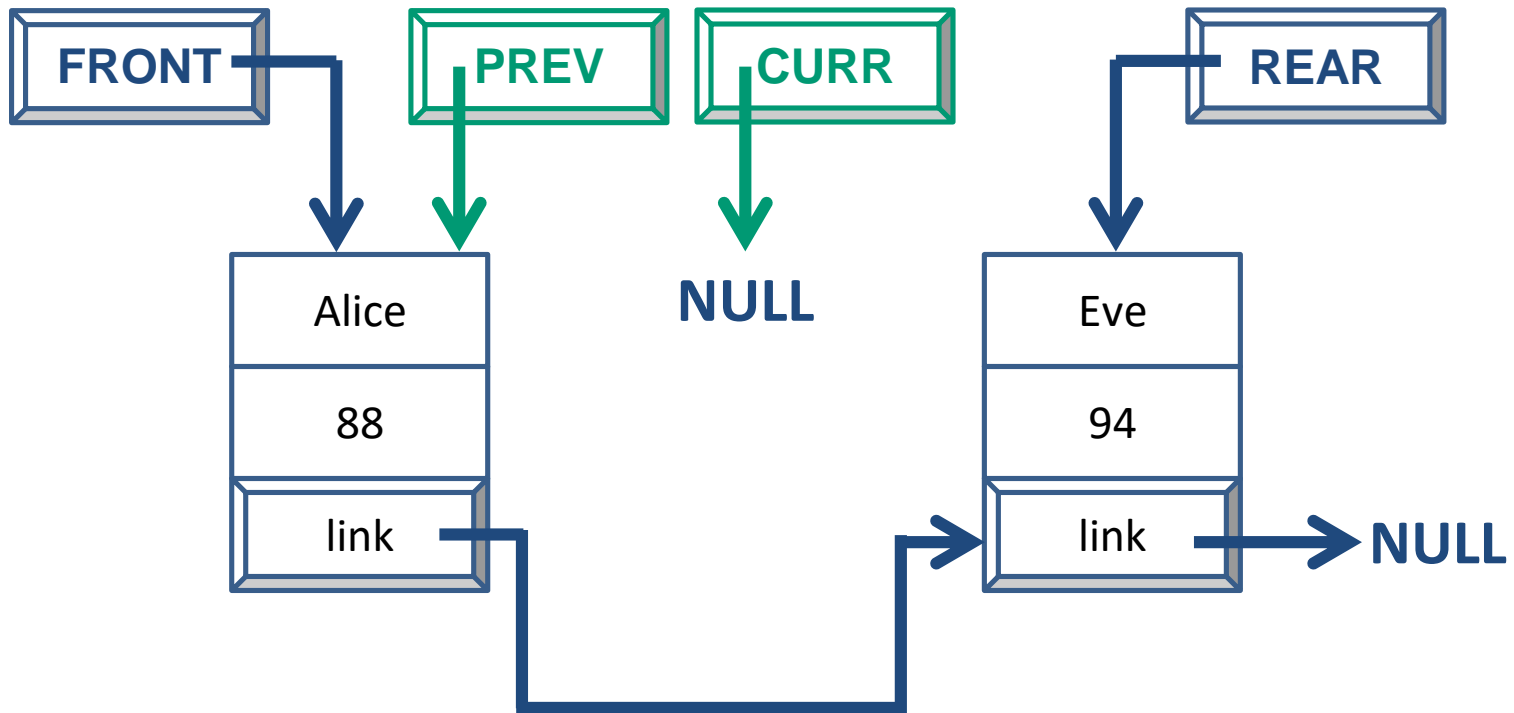
```
else { //middle node
    PREV->link = CURR->link;
}
```

# Deletion Case 3: Middle Node



```
else { //middle node
    PREV->link = CURR->link;
}
delete CURR;
```

# Deletion Case 3: Middle Node

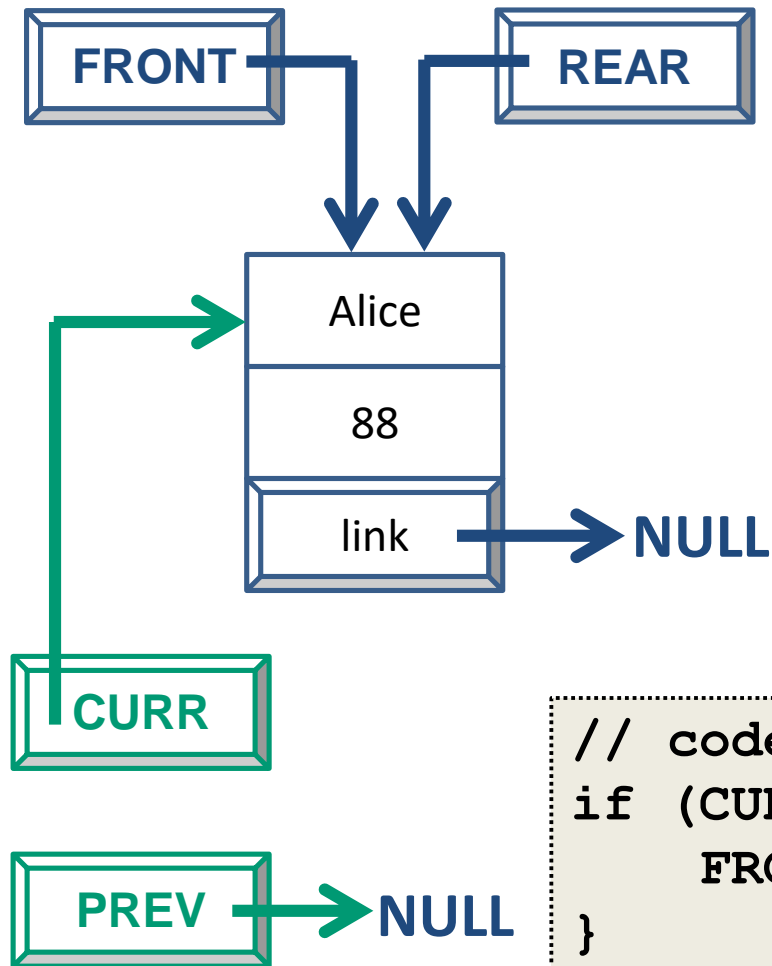


```
else { //middle node
    PREV->link = CURR->link;
}
delete CURR;
```

---

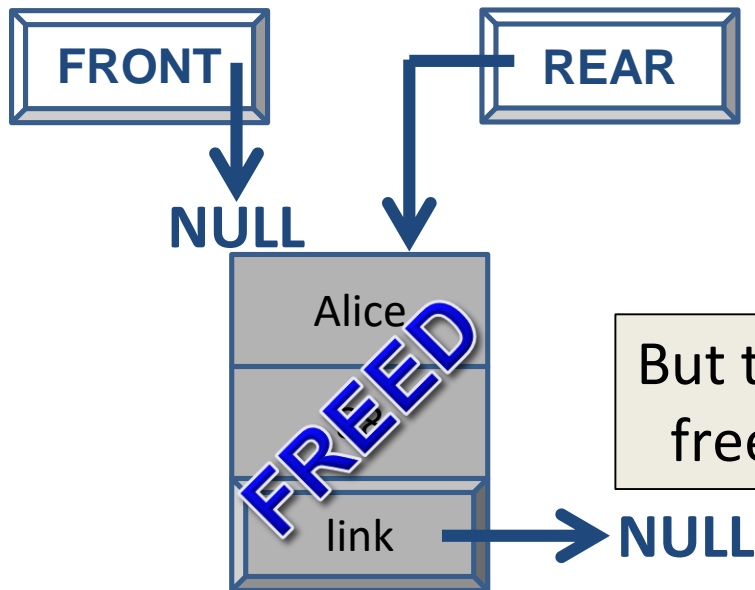
# Special Deletion Case: Only Node in Linked List

# Special Deletion Case: Only Node



```
// code that currently handles this
if (CURR == FRONT) {
    FRONT = FRONT->link;
}
delete CURR;
```

# Special Deletion Case: Only Node



What happens to **REAR** with this code? What does it point to?

But that memory has been freed! That's not good...



```
// code that currently handles this
if (CURR == FRONT) {
    FRONT = FRONT->link;
}
delete CURR;
```



# Special Deletion Case: Only Node

- If we are removing the only node from a Linked List, we need to set both **FRONT** and **REAR** to point to **NULL**

```
// new case for last node
if (CURR == FRONT && CURR == REAR) {
    FRONT = FRONT->link;
    REAR = REAR->link;
    // or FRONT = NULL;
    //     REAR = NULL;
}
delete CURR;
```

---

# Questions about Linked Lists

---

# Announcements

- Homework 2 is out
  - Due Tuesday, September 22nd at 8:59:59 PM
  
- Project 1 is out
  - Due Tuesday, September 29th at 8:59:59 PM
  
- Next Time:
  - Stacks and Queues