

# CS350: Data Structures

## Doubly Linked Lists

---

James Moscola

Department of Engineering & Computer Science

York College of Pennsylvania



# Doubly Linked Lists

---

- **Adds an additional pointer to a the list nodes that points to the previous node in the list**
- **Traversable in either the forward or backward direction**
- **Resolves the issue of removing the last node in the linked list**
  - Becomes a  $O(1)$  operation as opposed to  $O(N)$



# Doubly Linked List Operations

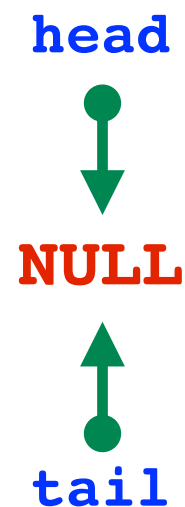
---

- **Basic operations include:**
  - **insert / add**
  - **remove / removeFirst / removeLast**
- **Additional operations may include:**
  - **getFirst / getLast**
  - **find**
  - **isEmpty**
  - **makeEmpty**

# Doubly Linked List Implementation

---

- **Basic implementation uses `head` and `tail` pointers that points to the first node and the last node in the list**
  - Both pointers points to **NULL** upon initialization when no nodes exist in the list

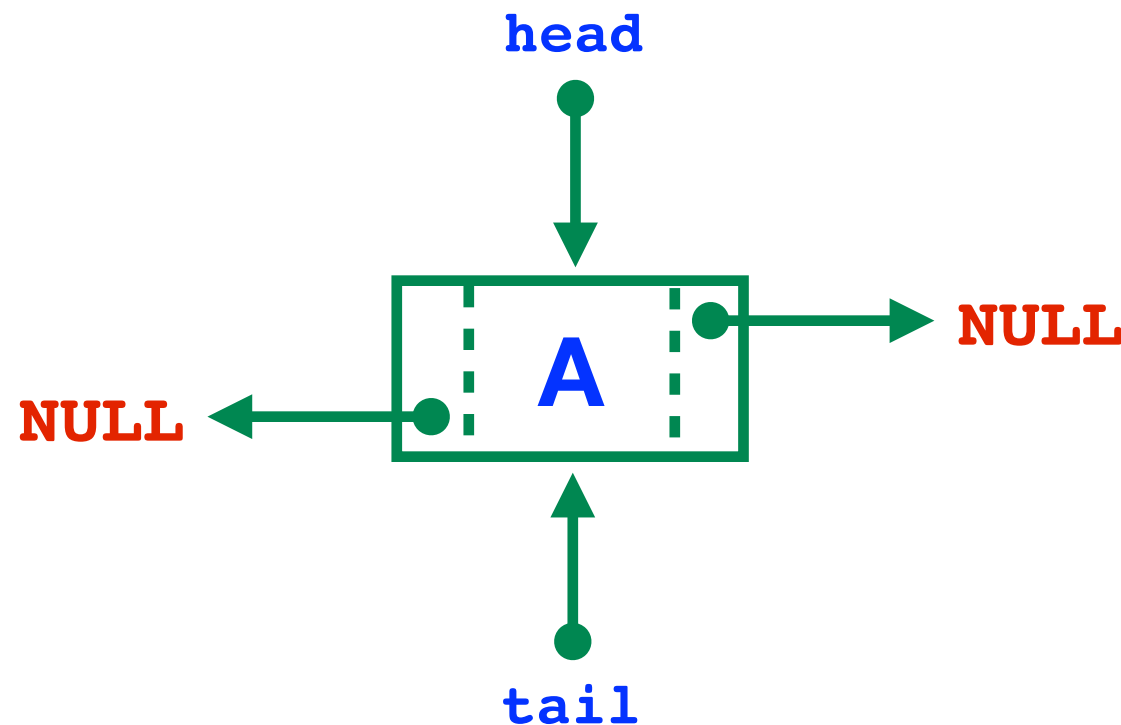


- **Depending on implementation, insertion may take place at the head of the list, at the tail of the list, or at some other specified node**

# After Inserting a Node

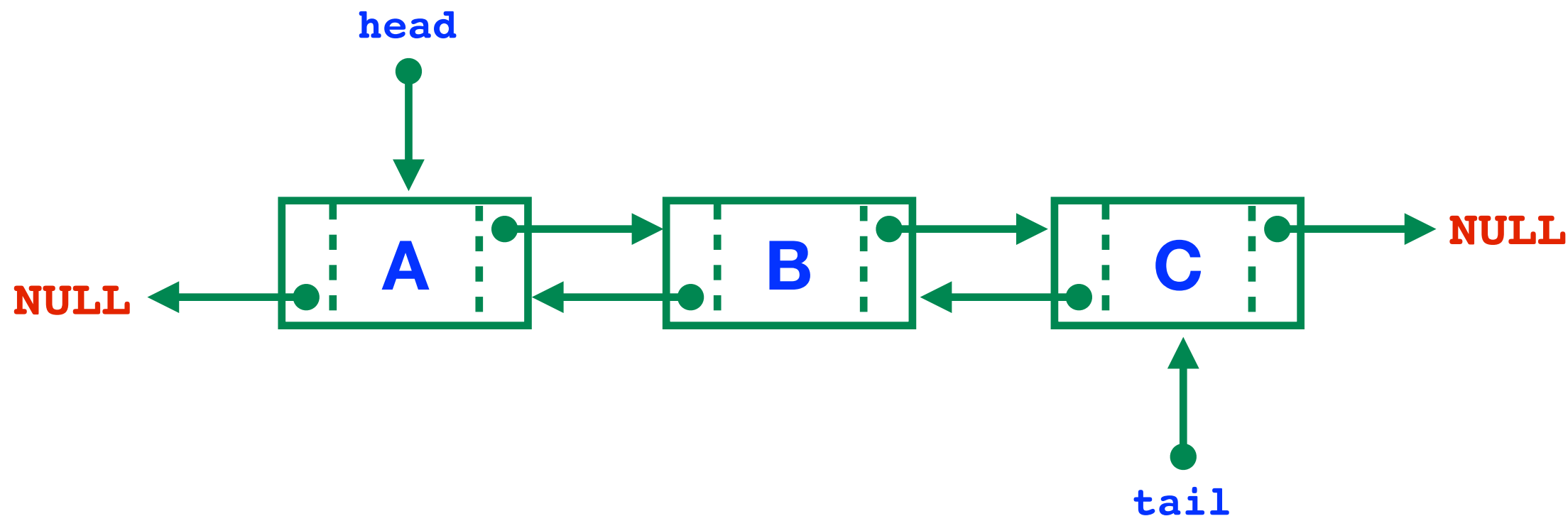
---

- In this illustration, nodes are inserted at the tail end (nodes can be inserted at either the head or the tail)
  - After inserting the first node, *A*
  - The *head* and the *tail* pointers are reassigned to point to the first node



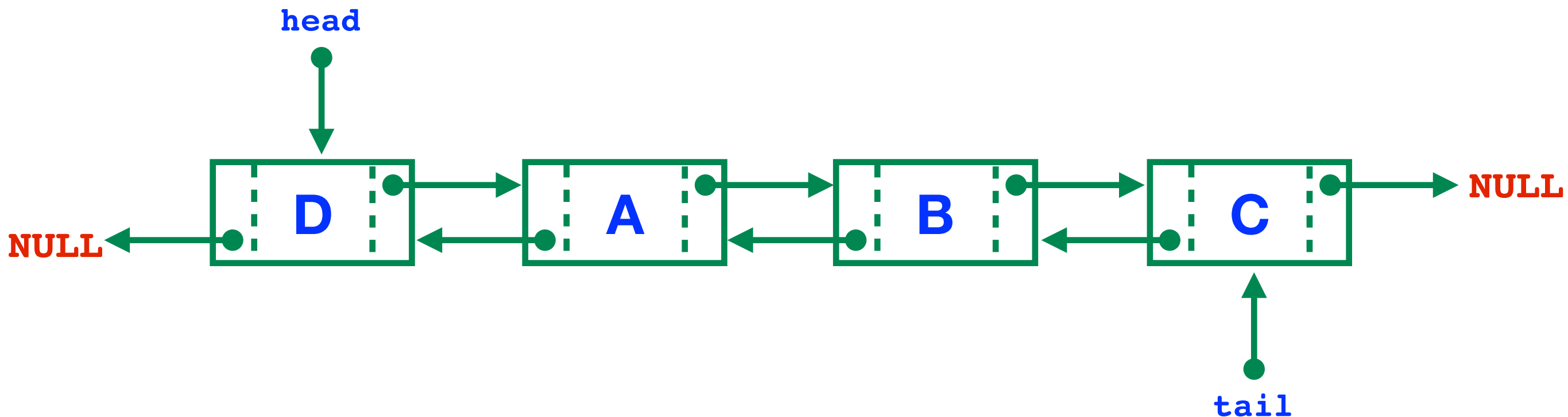
# After Inserting Additional Nodes

- In this illustration, nodes are inserted at the tail end (nodes can be inserted at either the head or the tail)
  - After inserting nodes in the sequence A, B, C
  - The **tail** pointer advances with each insertion at the **tail** end



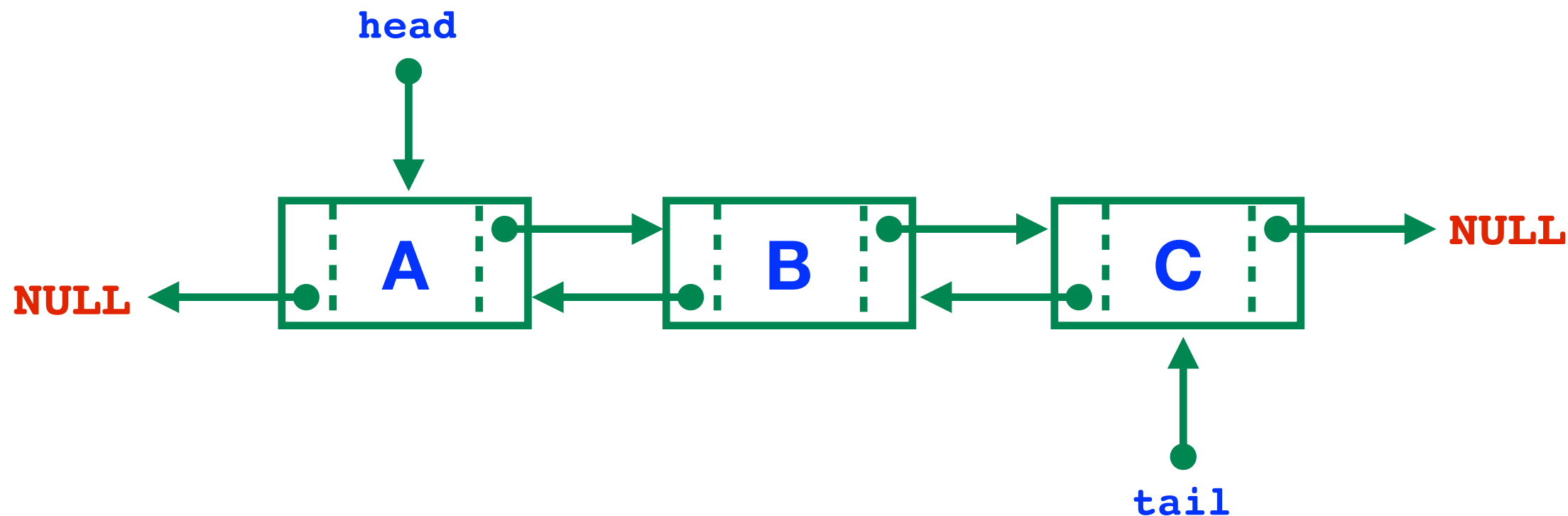
# After Inserting Additional Nodes

- In this illustration, nodes are inserted at the head
  - After inserting the node D
  - The **head** pointer retreats with each insertion at the **head**



# After Removing the First Node

- Nodes can be removed from the **head** of the list or the **tail**
  - After removing a single node from the **head** of the list
  - The **head** pointer advances with a removal from the **head** of the list

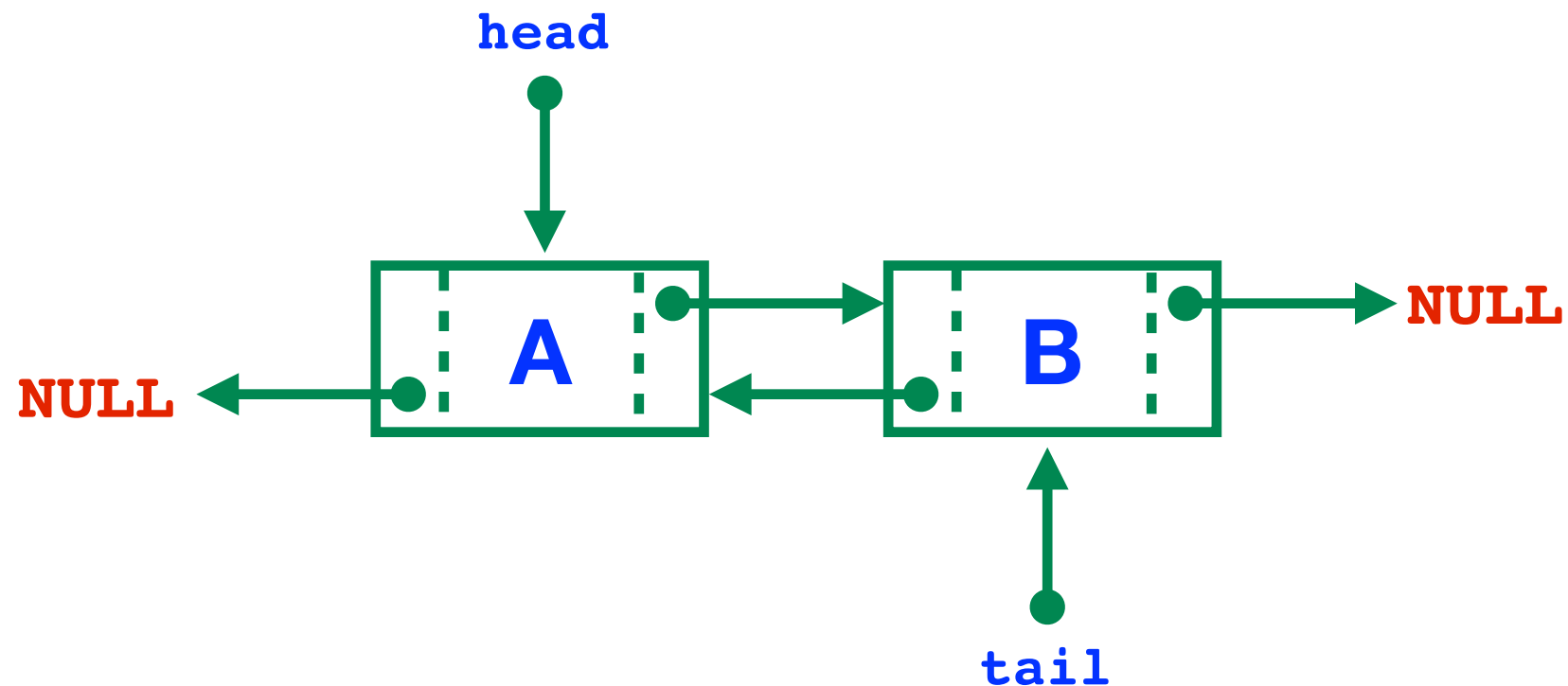




# After Removing the Last Node

---

- **Nodes can be removed from the **head** of the list or the **tail****
  - After removing a single node from the **tail** of the list
  - The **tail** pointer retreats with a removal from the **tail** of the list



# Doubly Linked List Implementation

---

```
public class DLinkedListNode<E> {  
    public E data;  
    public DLinkedListNode<E> next;  
    public DLinkedListNode<E> prev;  
}
```

# Doubly Linked List Implementation

---

```
// Inserts at the tail of the list

public void append (E data) {
    DLinkedListNode<E> newNode = new DLinkedListNode<E>( );
    newNode.data = data;    // assign data to newNode
    newNode.prev = tail;
    tail.next = newNode;
    tail = newNode;
}
```

**This method is oversimplified, what happens if this is called when the list is empty?**

# Doubly Linked List Implementation

---

## Fixed append method

```
// Inserts at the tail of the list

public void append (E data) {
    DLinkedListNode<E> newNode = new DLinkedListNode<E>();
    newNode.data = data; // assign data to newNode
    if (isEmpty()) {
        head = tail = newNode;
    } else {
        newNode.prev = tail;
        tail.next = newNode;
        tail = newNode;
    }
}
```

# Doubly Linked List Implementation

---

```
// Inserts at the head of the list

public void prepend (E data) {
    DLinkedListNode<E> newNode = new DLinkedListNode<E>();
    newNode.data = data;    // assign data to newNode
    newNode.next = head;
    head.prev = newNode;
    head = newNode;
}
```

**This method is oversimplified, what happens if this is called when the list is empty?**

# Doubly Linked List Implementation

---

// Removes node from head of list and returns its value

```
public E removeFirst() {  
    if (head != null) {  
        E nodeData = head.data;  
        head.next.prev = null;  
        head = head.next;  
        return nodeData;  
    } else {  
        return null;  
    }  
}
```

# Considerations for Linked List Implementation

---

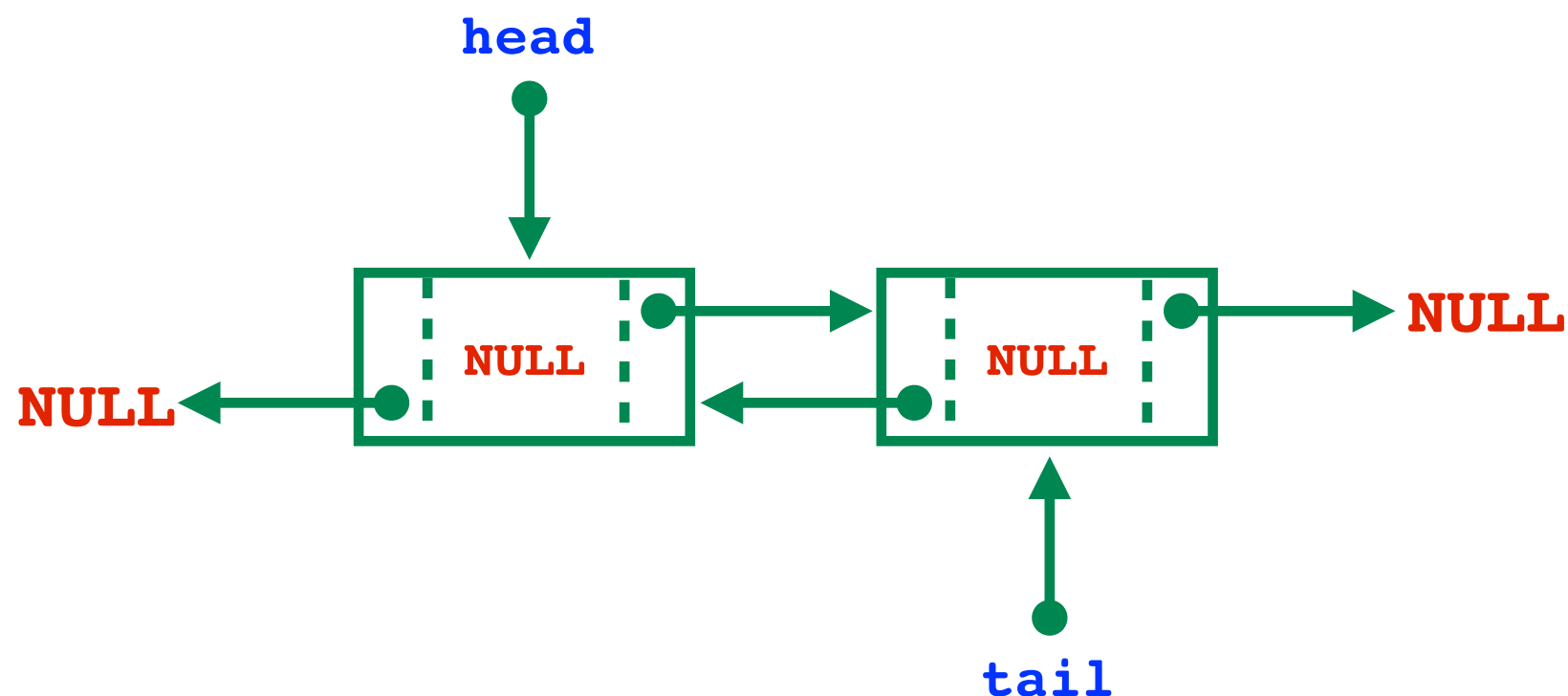
- **Implementation as previously shown requires error checking in the insertion and removal methods to check for edge cases (i.e. checking for an empty list)**
- **To improve the speed of operations, it is possible to remove these tests**
  - Tradeoff: speedup comes at the expense of one/two additional 'dummy' nodes in the Linked List
- **Idea: create one or two dummy nodes (sentinel nodes) that exists in the linked list at ALL times**
  - Eliminates the need to always check for **NULL**
  - Generalized the insertion and removal methods

# Doubly Linked List with Two Sentinel Nodes

---

- **To check for empty:** `(head.next == tail);`
- **When traversing the list check to see if current position points to either the `head` or the `tail` to determine if at the end of the list**

## Empty list

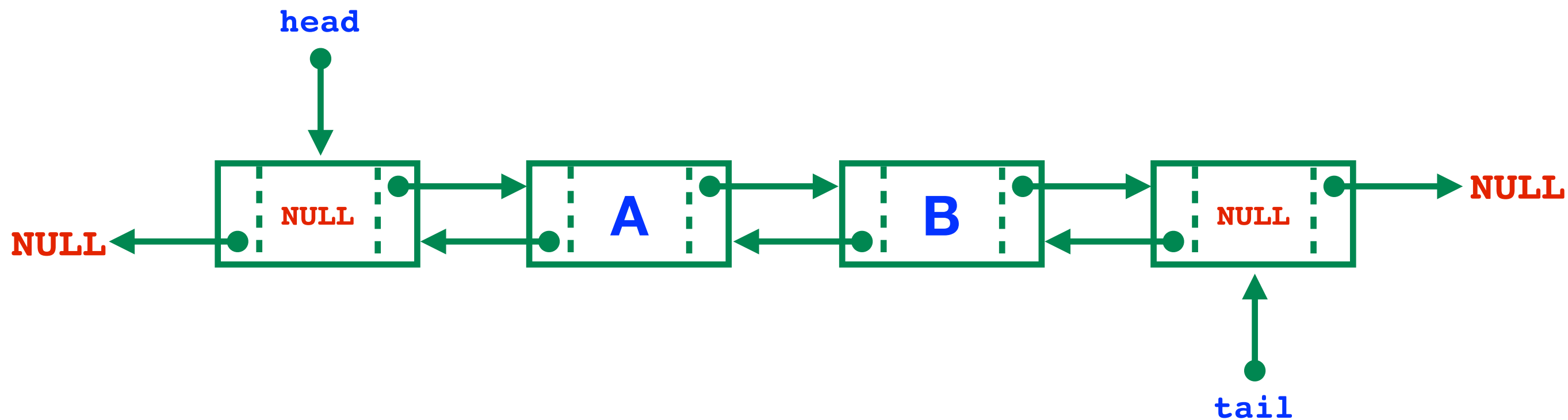




# Doubly Linked List with Two Sentinel Nodes

---

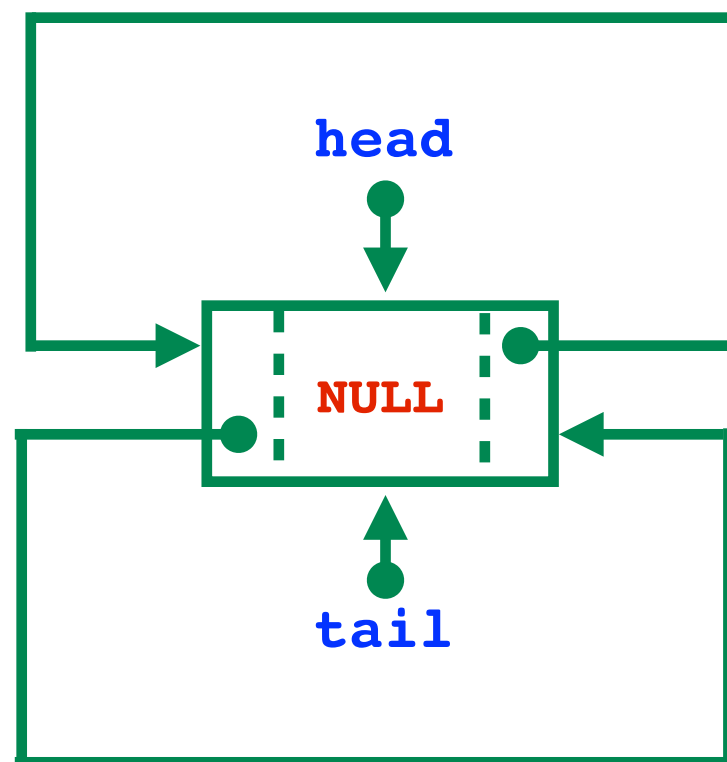
## List with two nodes



# Circular Doubly Linked List with a Sentinel Node

---

## Empty list



**How should `isEmpty()` be implemented?**

# Circular Doubly Linked List with a Sentinel Node

---

## List with two nodes

