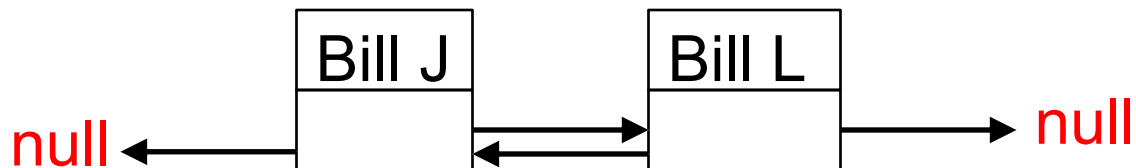


# CSCI 136

## Data Structures & Advanced Programming

Doubly Linked Lists

Fall 2020

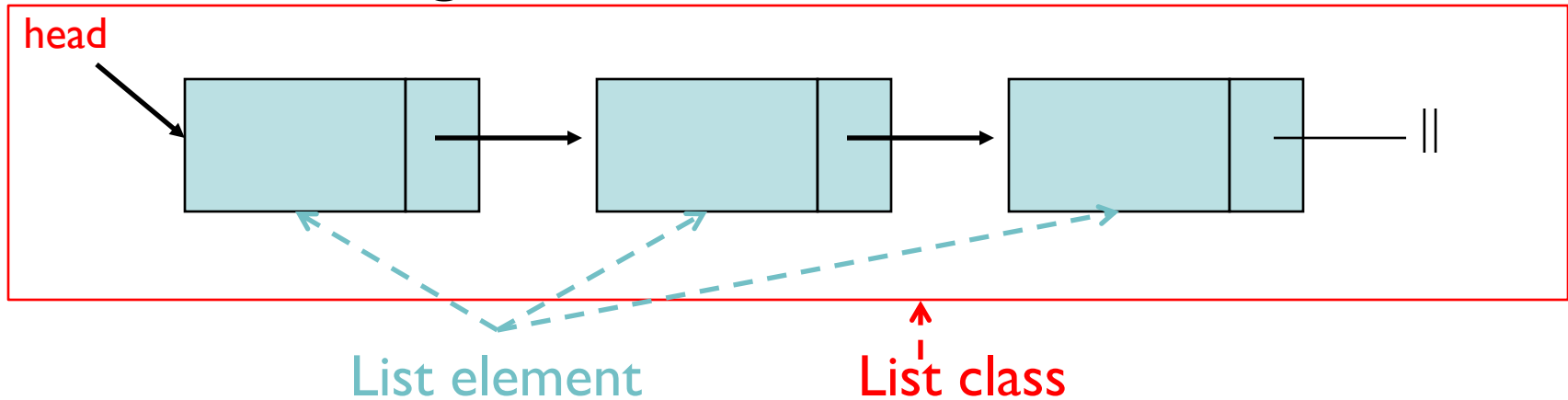


# This Video

- Continue discussing lists w/ linked structures
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists

# Recall: Linked List Basics

- There are two key aspects of Lists
  - The list elements
    - Store data, point to neighboring element(s)
  - The list itself
    - Manages/organizes the elements into a cohesive list
- Visualizing lists:



# Recall: SinglyLinkedList Basics

- SinglyLinkedList class has:
  - Public methods that implement List interface
  - Internal state/details that are hidden from the user
- Each “node” has:
  - A data value
  - A next variable that identifies the next element in the list
- The SinglyLinkedList class keeps a reference only to the first list element (head)

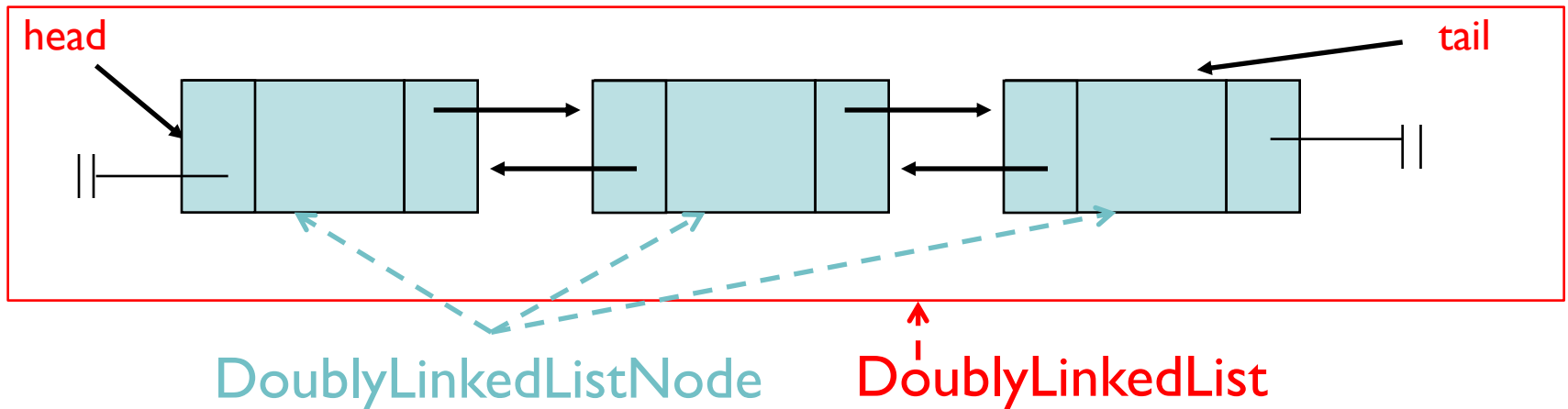
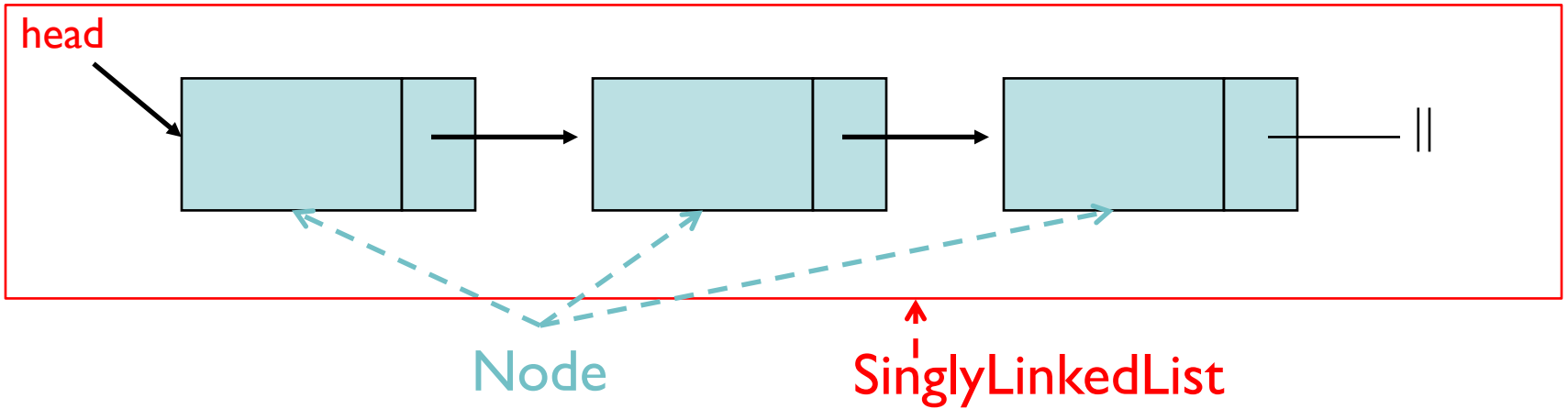
# SinglyLinkedList Summary

- More control over space usage than Vectors
- Easy to access the front of list:  $O(1)$ 
  - Direct access to head: yay!
- Difficult to access later elements:  $O(n)$ 
  - We must always start our traversals at head
  - We must always go forward

# DoublyLinkedLists

- Keep reference/links in **both** directions
  - Can therefore traverse forwards and backwards!
- DoublyLinkedListNode class's instance variables:
  - E value;  
DoublyLinkedListNode next;  
DoublyLinkedListNode prev;
- This adds one more reference *per node*, but overall space overhead still proportional to number of elements

# Linked List Visualization



# DoublyLinkedList Tradeoffs

- ALL tail operations (including `removeLast`) are fast!
  - Why? We have direct access to the tail node & *its predecessor*
- But, additional *code complexity* in each list operation
  - Example: `add(E d, int index)` has four cases to consider now:
    - empty list
    - add to front
    - add to tail
    - add in middle
- Some additional space consumption (*previous*)
  - but space overhead is still  $O(n)$  like SLL and Vector



```
public class DoublyLinkedListNode<E> {
    protected E data;
    protected DoublyLinkedListNode<E> nextElement;
    protected DoublyLinkedListNode<E> previousElement;

    // Constructor "stitches" new node btwn existing nodes
    public DoublyLinkedListNode(E v,
                                DoublyLinkedListNode<E> next,
                                DoublyLinkedListNode<E> previous) {

        data = v;

        nextElement = next;
        if (nextElement != null)
            nextElement.previousElement = this;

        previousElement = previous;
        if (previousElement != null)
            previousElement.nextElement = this;
    }
}
```

```
public void add(int i, E value) {
    if (i == 0) addFirst(value); // head
    else if (i == size()) addLast(value); // tail
    else {
        // find items before and after insertion point
        DoublyLinkedListNode<E> before = null;
        DoublyLinkedListNode<E> after = head;
        while (i > 0) {
            before = after;
            after = after.next();
            i--;
        }
        // create new value to "splice" into list
        // note: constructor properly updates neighbors
        DoublyLinkedListNode<E> insertedNode =
            new DoublyLinkedListNode<E>(value, after, before);
        count++;
    }
}
```

# Vectors vs. SLL vs. DLL

Operation	Vector	SLL	DLL
size	$O(1)$	$O(1)$	$O(1)$
addLast	$O(1)$ or $O(n)$ (if resize)	$O(n)$	$O(1)$
removeLast	$O(1)$	$O(n)$	$O(1)$
getLast	$O(1)$	$O(n)$	$O(1)$
addFirst	$O(n)$	$O(1)$	$O(1)$
removeFirst	$O(n)$	$O(1)$	$O(1)$
getFirst	$O(1)$	$O(1)$	$O(1)$
get(i)	$O(1)$	$O(n)$	$O(n)$
set(i)	$O(1)$	$O(n)$	$O(n)$
remove(i)	$O(n)$	$O(n)$	$O(n)$
contains	$O(n)$	$O(n)$	$O(n)$
remove(o)	$O(n)$	$O(n)$	$O(n)$