DATA STRUCTURES & ALGORITHMS

COMP H3025

Lecture 3: Circular Lists & Doubly Linked Lists

Lecturer: Stephen Sheridan

LINKED LIST VARIATIONS

- In our previous lecture we covered linear linked lists where we kept a *reference* to the **HEAD** node and **NEXT** node.

- There are a number of variations on referenced based lists that extend on the ideas covered in our last lecture. They are:

  - Tail references

  - Circular Lists

  - Doubly Linked Lists

TAIL REFERENCES

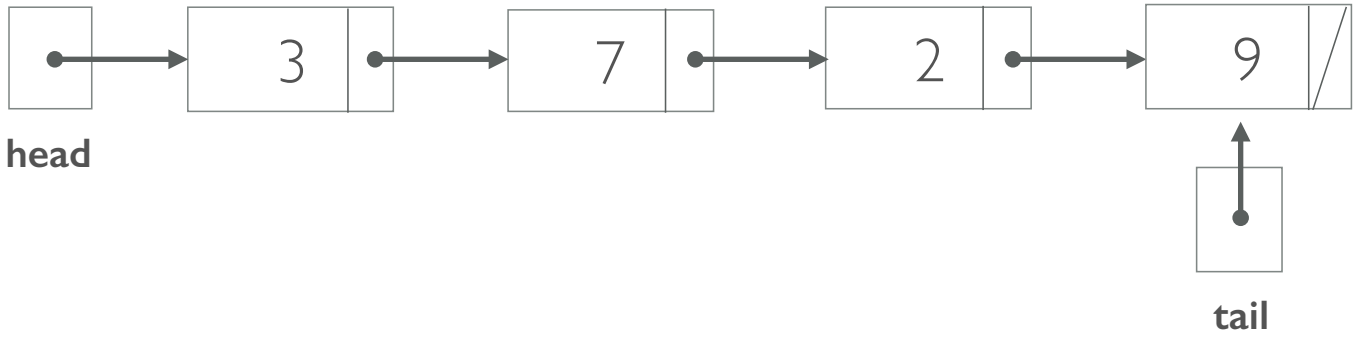
- In many situations, we simply want to add an item to the end of a list.
- For example, maintaining a list of requests for a library book would require that each request of added to the end of a waiting list.
- You could use an ADT "Waiting List" as follows:

```
waitingList.add(request, waitingList.size()+1);
```

- This adds a request to the end of the waiting list.

TAIL REFERENCES

- Each time we add a new request, we must get/find the last node in the list.
- One way to accomplish this is to traverse the list each time you add a new request (node).
- A better way is to use a **tail** reference to remember where the end of the linked list is - just like **head** refers to the start of the list.
- Just like **head**, **tail** is external to the list.



TAIL REFERENCES

- We can then perform the insert using the single statement:

```
tail.setNext(new Node(request, null));
```

- This statement sets the next reference in the last node in the list to point to a newly allocated node.

- Update tail so that it references the new last node by writing:

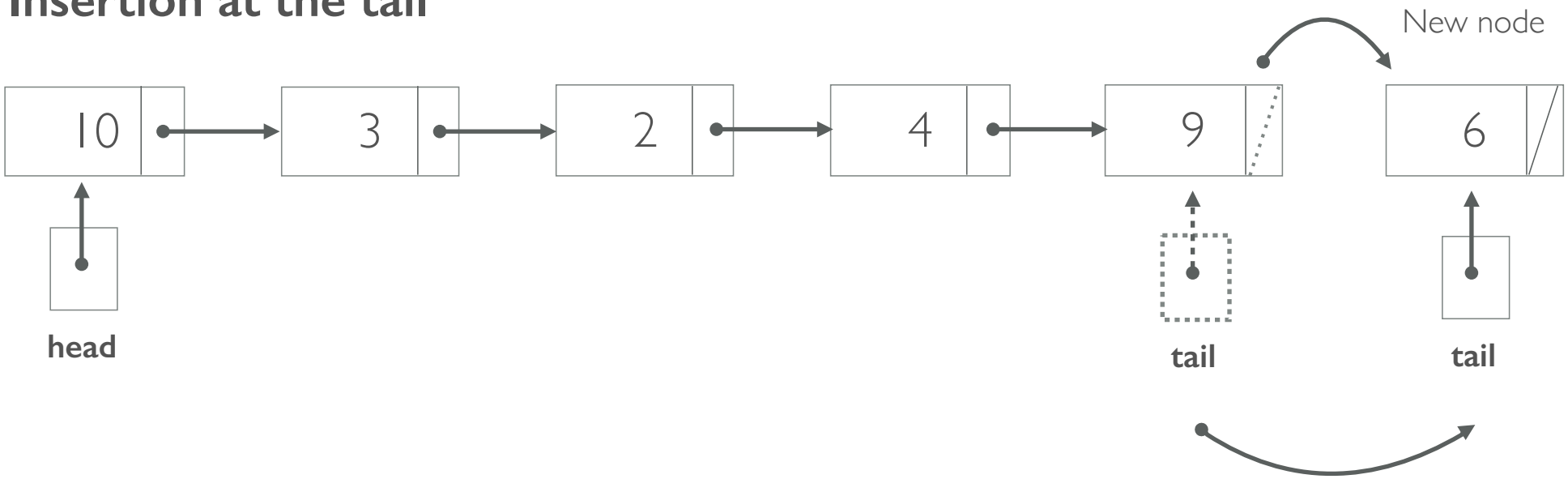
```
tail = tail.getNext();
```

- Initially, when you insert the first item into an empty linked list, tail like head is **null.**

TAIL REFERENCES

```
tail.setNext(new Node(request, null));
```

Insertion at the tail




```
tail = tail.getNext();
```

CIRCULAR LISTS

- When you use a computer that is part of a network, you share the services of a server with other users.

- The system must organise the users so that each can be given access to resources in turn. Users can frequently **log on** and **log off** the system.

- A linked list of users would allow the system to maintain order without shifting names when it makes insertions and deletions from the list.

CIRCULAR LISTS

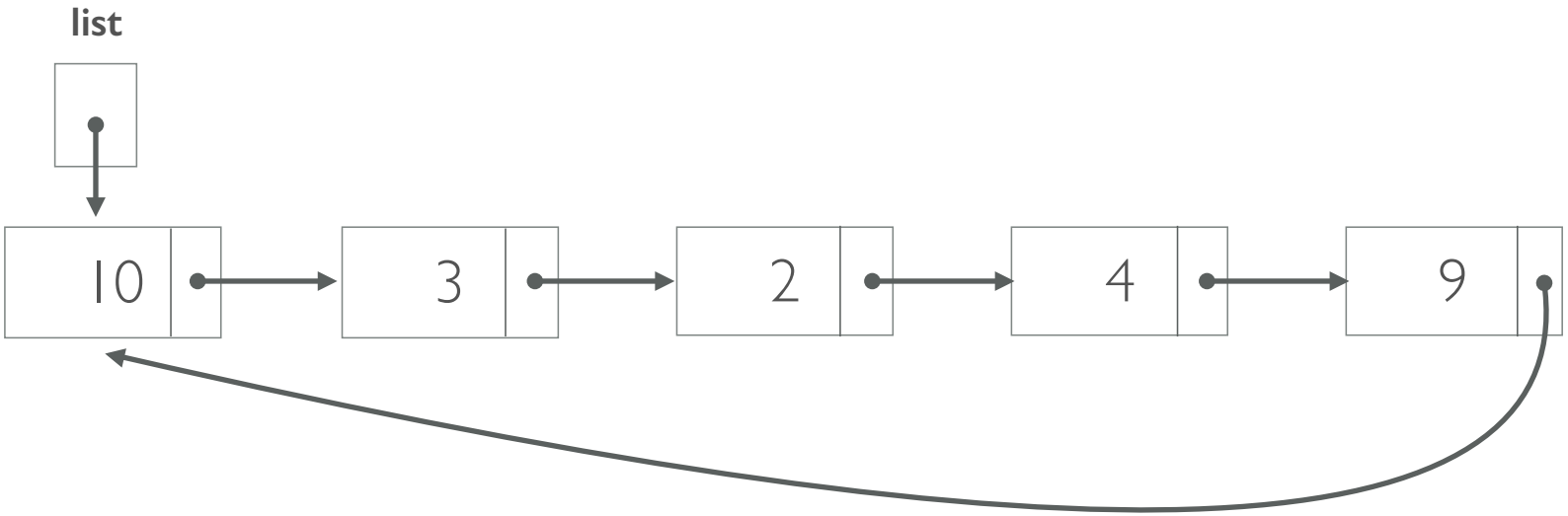
Q: The system could work its way along the list allowing each user to *use* resources in turn. But what happens when we reach the end of the list?

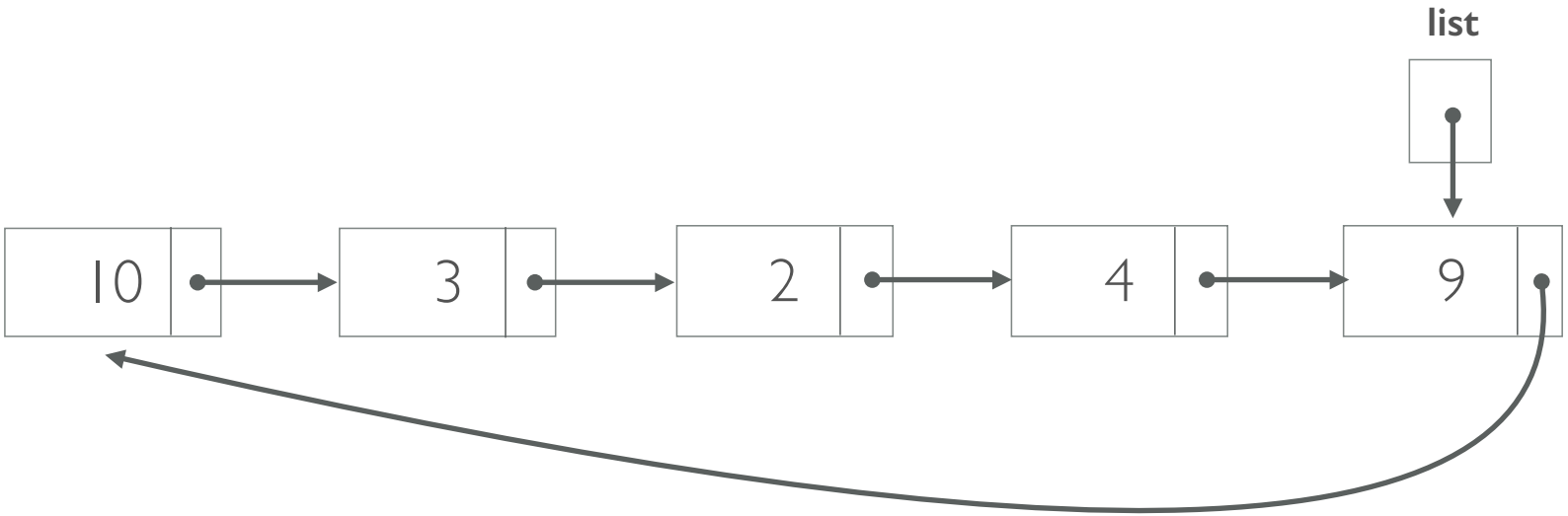
A: The system must go to the beginning of the list and continue from there.

- In this case it is not convenient for the last node in the list to reference **null**.

- If we were to make the last node in the list refer back to the first node, we would have a **Circular List**.

CIRCULAR LISTS (LIST REFERENCE)





CIRCULAR LISTS

Q: How do you know if you have traversed the complete list?

A: By simply comparing the current reference **curr** to the external reference **list**.

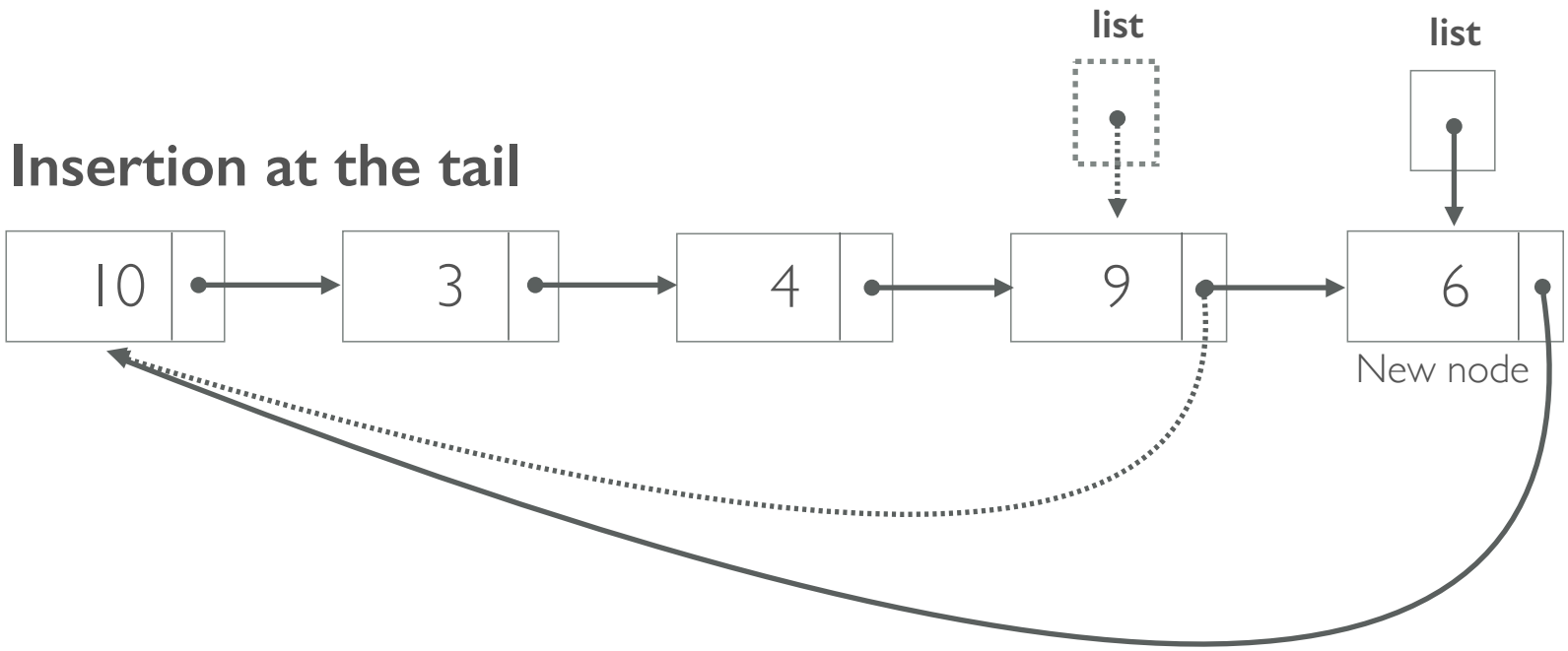
```
// Display the data in a circular linked list
// List references the last node
if (list != null){

    // list not empty
    Node first = list.getNext(); // reference first node
    Node curr = first;

    // Loop invariant: curr refs next node to display
    do{
        //write output
        System.out.println(curr.getItem());
        curr = curr.getNext();
    }while(curr != first)

} //End if
```

CIRCULAR LISTS




```
Node newNode = new Node(user, list.getNext());
list.setNext(newNode);
list = newNode;
```

DOUBLY LINKED LISTS

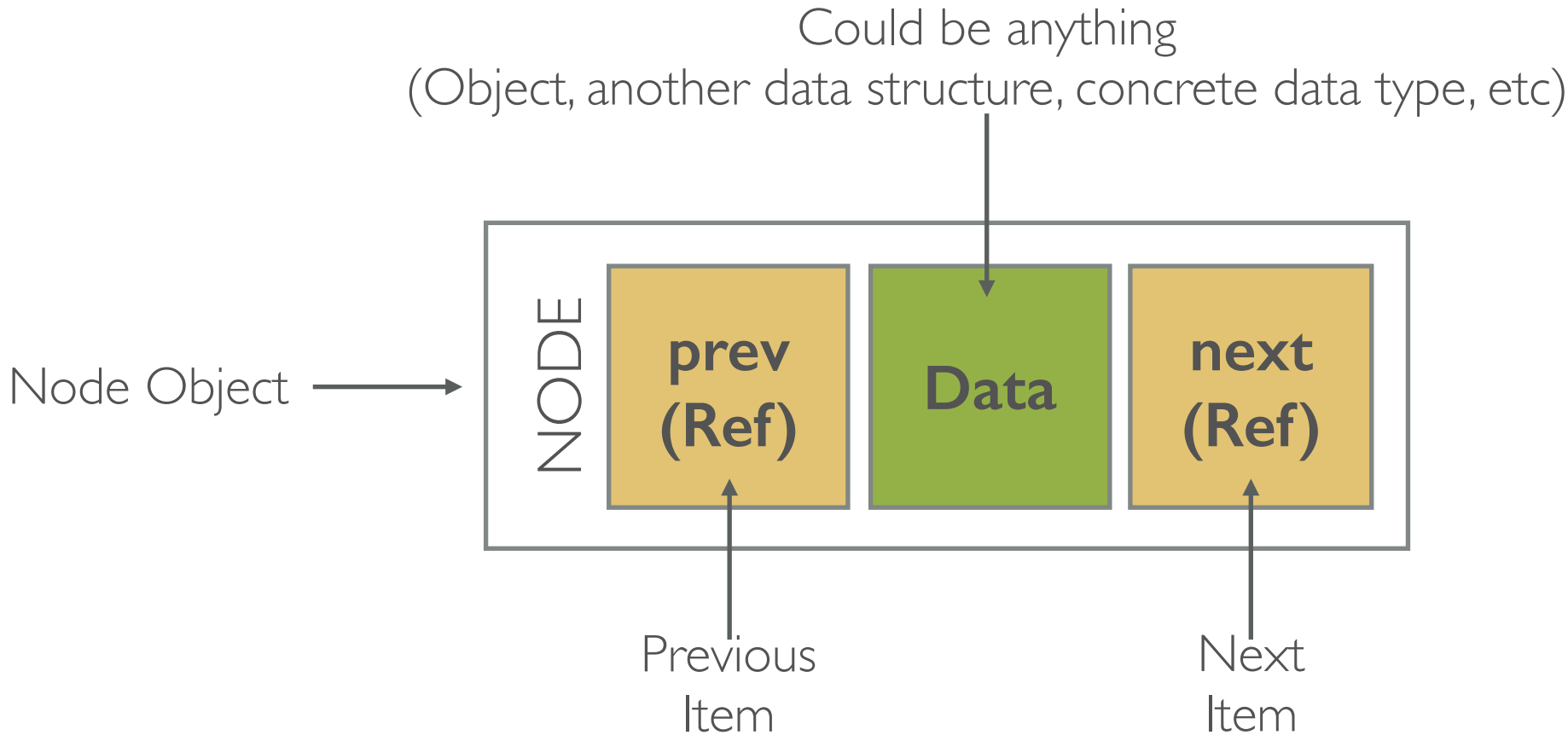
- One limitation of the referenced based lists we have covered so far is that we can only traverse the list nodes in one direction.

- That is, we can only visit the **next** node in the list from any given position.

- We can get around this problem by adding a second field to each node that refers to the previous node in the list. We refer to such lists as **Double Linked Lists**.

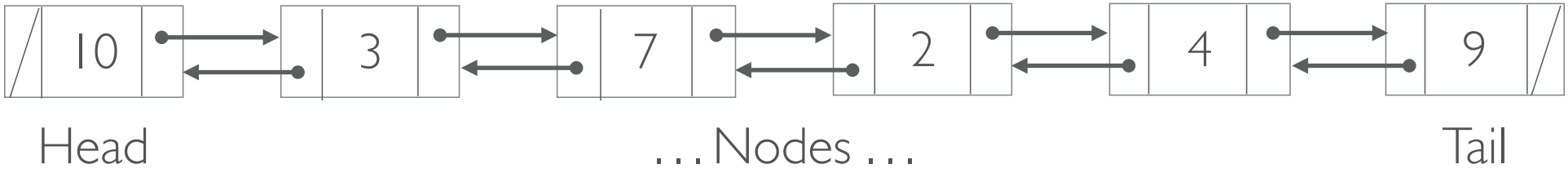
DOUBLY LINKED LIST NODE

- Each node contains data, a *link* to the next item and a *link* to the previous item.

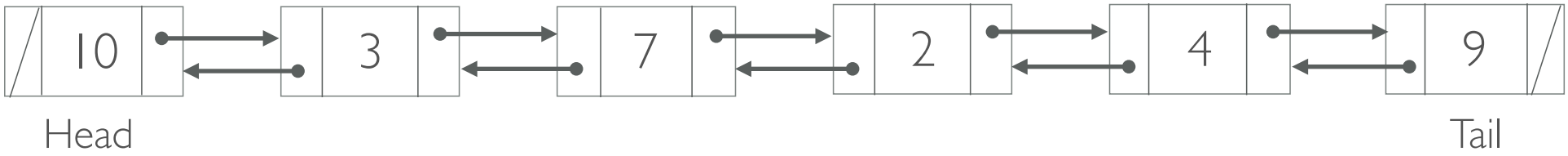


DOUBLY LINKED LISTS

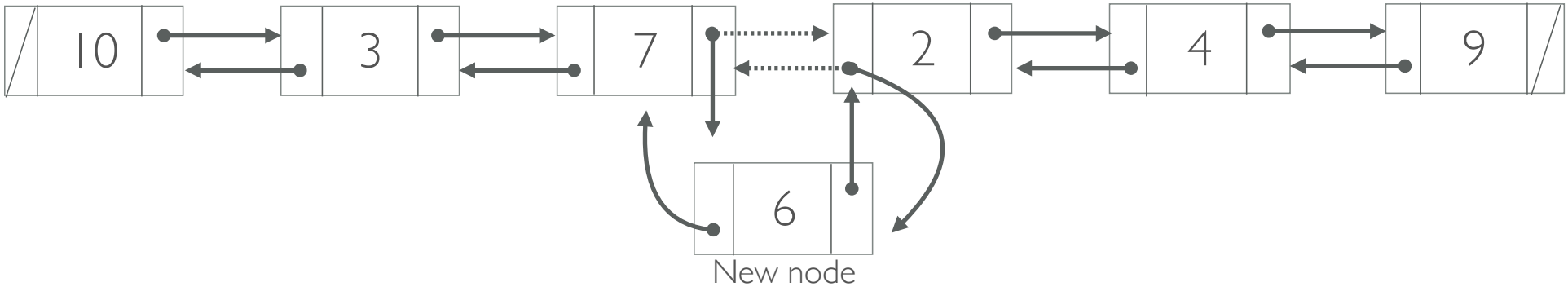
- Each node contains data, a *link* to the next item and a *link* to the previous item.



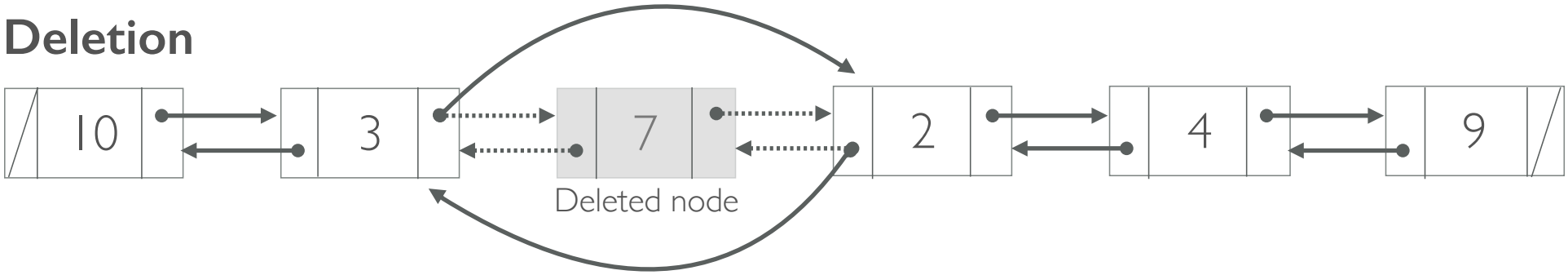
DOUBLY LINKED LISTS



Insertion



Deletion



DOUBLY LINKED NODE CLASS

```
public class Node
{
  private int item;
  private Object previous;
  private Object next;

  public Node(Object newItem)
  {
    item = newItem;
    previous = null;
    next = null;
  } // end constructor

  public Node(Object newItem, Node
prevNode, Node nextNode)
  {
    item = newItem;
    previous = prevNode;
    next = nextNode;
  } // end constructor
```

```
  public void setItem(int newItem)
  {
    item = newItem;
  } // end setItem

  public int getItem()
  {
    return item;
  } // end getItem

  public void setPrevious(Node prevNode)
  {
    previous = prevNode;
  } // end setPrev
  public void setNext(Node nextNode)
  {
    next = nextNode;
  } // end setNext
  public Node getPrevious()
  {
    return previous;
  } // end getPrevious
  public Node getNext()
  {
    return next;
  } // end getNext

}  // end class IntegerNode
```

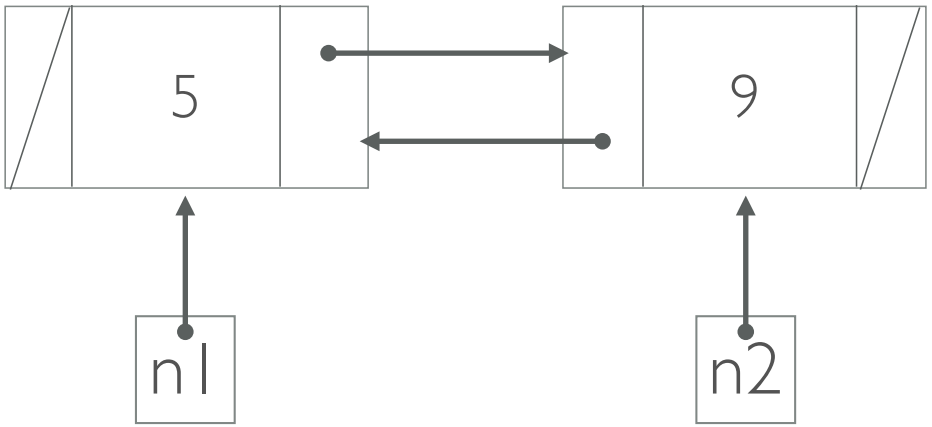
USING THE DOUBLY LINKED NODE CLASS

- We can use the doubly linked Node class as follows:

```
Node n1 = new Node(); // Create an Node object
Node n2 = new Node(); // Create an Node object

n1.setItem(5); // Set data item in first node
n2.setItem(9); // Set data item in second node

n1.setNext(n2); // Create a link between n1 and n2  (n1 -> n2)
n2.setPrevious(n1); // Create a link between n1 and n2  (n1 <- n2)
```

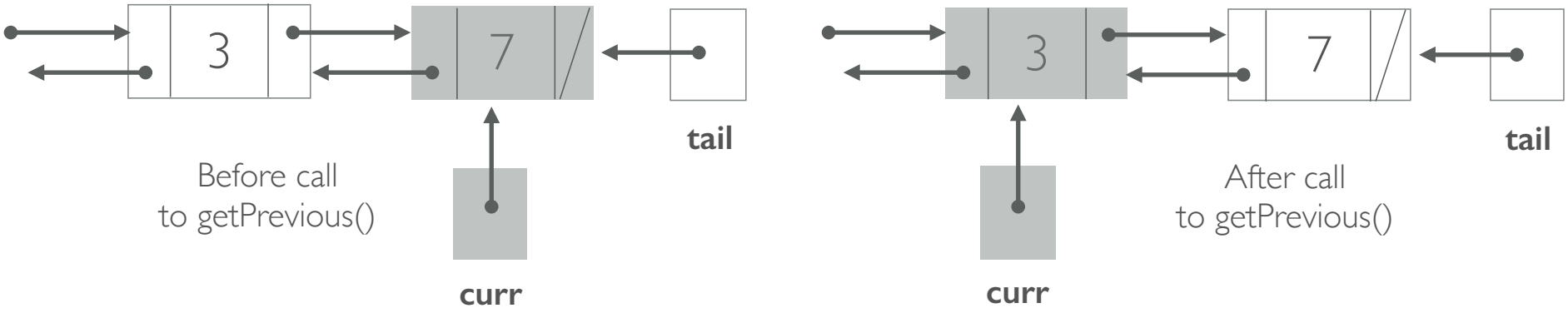


DISPLAYING THE CONTENTS OF A DOUBLY LINKED LIST

- To display items in a doubly linked list we can of course traverse the list in the usual way from **head** to **tail**. However, we also have the option of traversing the list from **tail** to **head**.

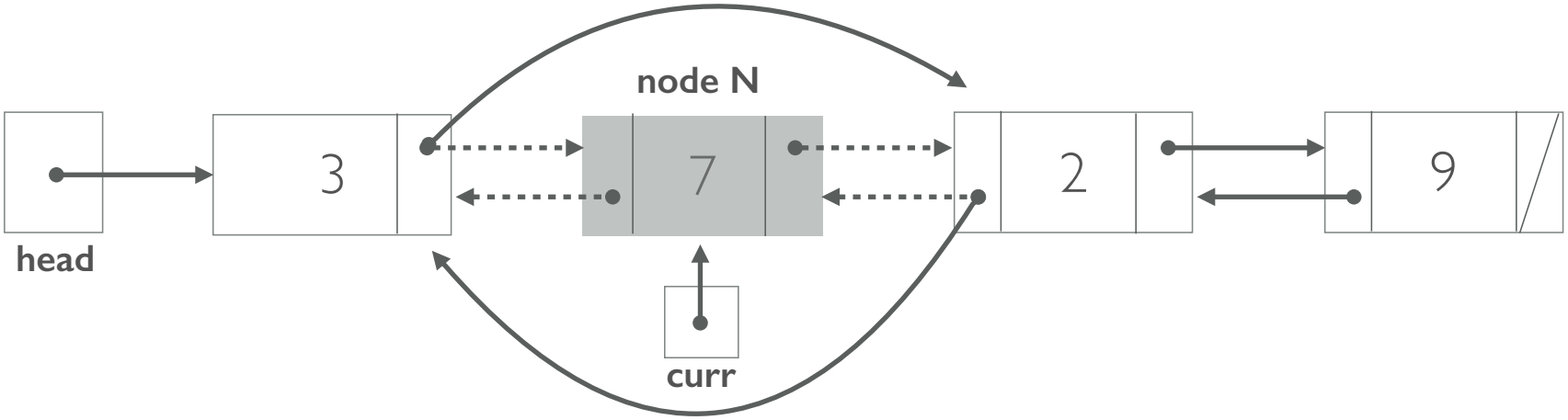
- To advance the current position to the previous node we use the Java statement:

```
curr = curr.getPrevious();
```



DELETING A SPECIFIED NODE FROM THE LIST

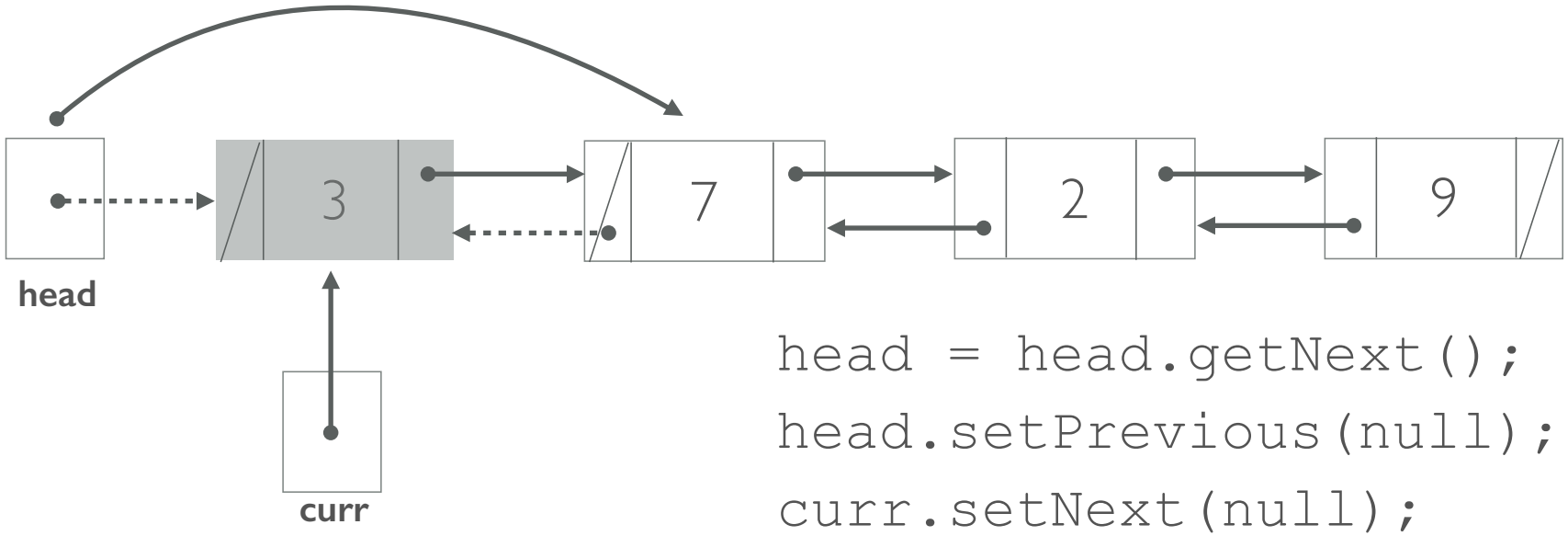
- Deleting a node in a doubly linked list still requires us to find the node to be deleted and keep a reference to it in **curr**. Now that we have a reference to previous and next we can use the following code.




```
curr.getPrevious().setNext(curr.getNext());
curr.getNext().setPrevious(curr.getPrevious());
curr.setNext(null); //Garbage collection
curr.setPrevious(null); // Garbage collection
```

DELETING THE FIRST NODE

- When you delete the first node of the list, the value of **head** must be changed so that the second node in the list is now the first and the second nodes previous link is set to **null**.

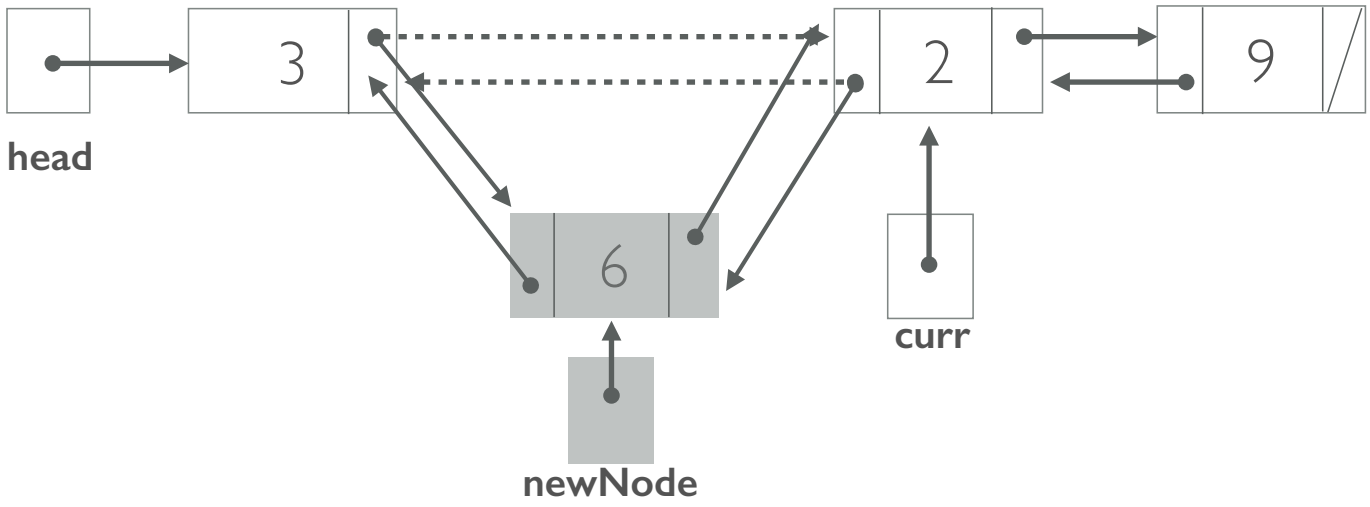



```
head = head.getNext();
head.setPrevious(null);
curr.setNext(null);
```

INSERTING A NODE AT A SPECIFIED POSITION

- We can insert the node, which a reference variable **newNode** references, between two nodes that **prev** and **curr** reference.

```
newNode.setNext(curr);
newNode.setPrevious(curr.getPrevious());
curr.setPrevious(newNode);
newNode.getPrevious().setNext(newNode);
```



TODO - WEEK 3

- Implement the ADT List using a doubly linked list
  - Implement all the methods listed in the **DListInterface** class on MOODLE.
  - Implement two display methods - displayHeadTail and displayTailHead
  - Add a displayList method to the linked list class.
  - Add a method called listLargest that returns the largest data item
  - Write a small driver program that demonstrates the use of the ADT List
  - Your program should print results to the screen in a simple manner