# DATA STRUCTURES & ALGORITHMS

## COMP H3025

### Lecture 2: Linked Lists



Lecturer: Stephen Sheridan

# PROBLEMS WITH AN ARRAY-BASED LIST

- The ADT list as described in the previous lecture has operations for add, remove and retrieve, given their positions in the list.

- A close examination of the array based implementation of the ADT list shows that an array is not always the best data structure to use to maintain a collection of data in list form.

- An array has a fixed size, but the ADT list can have an arbitrary length (amount of items).

- Therefore, we really cannot use an array to implement a list because the number of items in the list may exceed the size of the array.

# PROBLEMS WITH AN ARRAY-BASED LIST

- While the most intuitive means of imposing an order on data is to sequence it physically, this approach has its disadvantages.
- In a physical ordering, the successor of an item **X** is the next data item in the sequence after **X,** that is, the item to the right.
- That means that when we insert or delete an item we must physically reorder the items by shifting to the right or to the left.
- Shifting data can be VERY costly, especially if we need to insert and delete often. The cost associated with insertion and deletion will also depend on the size of the list.
- Essentially, an array-based implementation of a list is not much use in a real world situation!

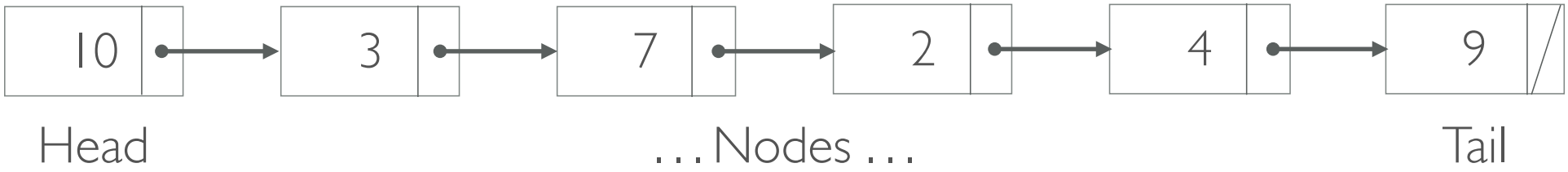
# WHAT ALTERNATIVES ARE AVAILABLE?

Array-based list (**physical sequence, contiguous block of memory**)

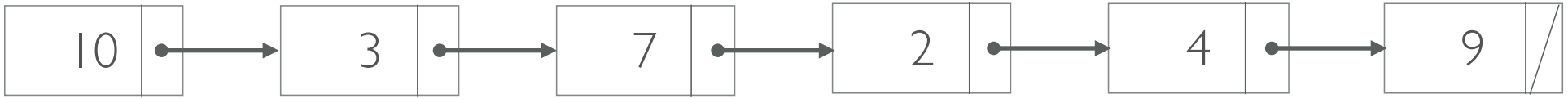
# LINKED LIST STRUCTURE

Array-based list (**physical sequence, contiguous block of memory**)

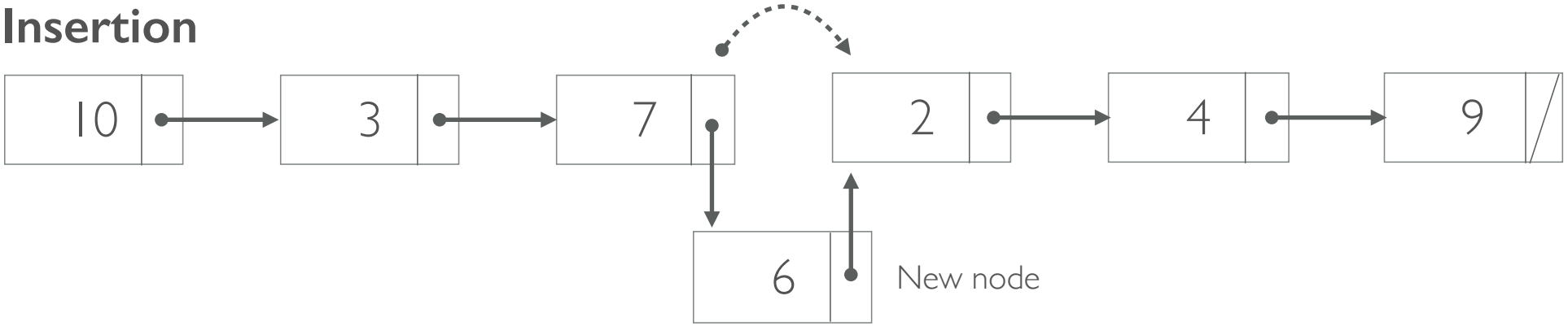

Linked list (**logical sequence, separate memory blocks**)



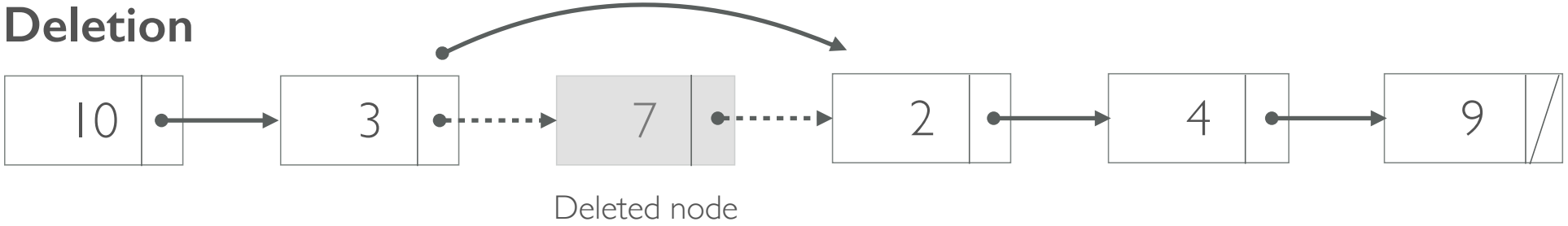
# LINKED LIST INSERTION AND DELETION




**Insertion**




**Deletion**



# LINKED LIST INSERTION AND DELETION

- In the diagrams on the previous slide, each item is actually *linked* to the next item.
- Therefore, if you know where an item is, you can determine its successor, which can be anywhere physically (in memory).
- This flexibility not only allows you to *insert* and *delete* items without shifting data, but it also allows you to increase the size of the list easily. A linked list will grow as needed.
- To insert a new item, simply select an insertion point and set two links.
- To delete an item, find its place in the list and change a link to bypass the item.
- Its easy to see why this data structure gets its name as all items are *linked* together.

# JAVA OBJECT REFERENCES

- Before we look at how linked lists can be implemented in Java, we must first examine how we can *refer to* (**reference**) objects.

- Java allows an object to reference another, and we can use this ability to build a linked list.

- When you declare a variable that refers to an object of a given class, you are creating a reference to the object.

```
String myString = new String("Hello");
String ref = myString;
```

# JAVA OBJECT REFERENCES

- Remember that an object does not come into existence until you use the **new** operator.

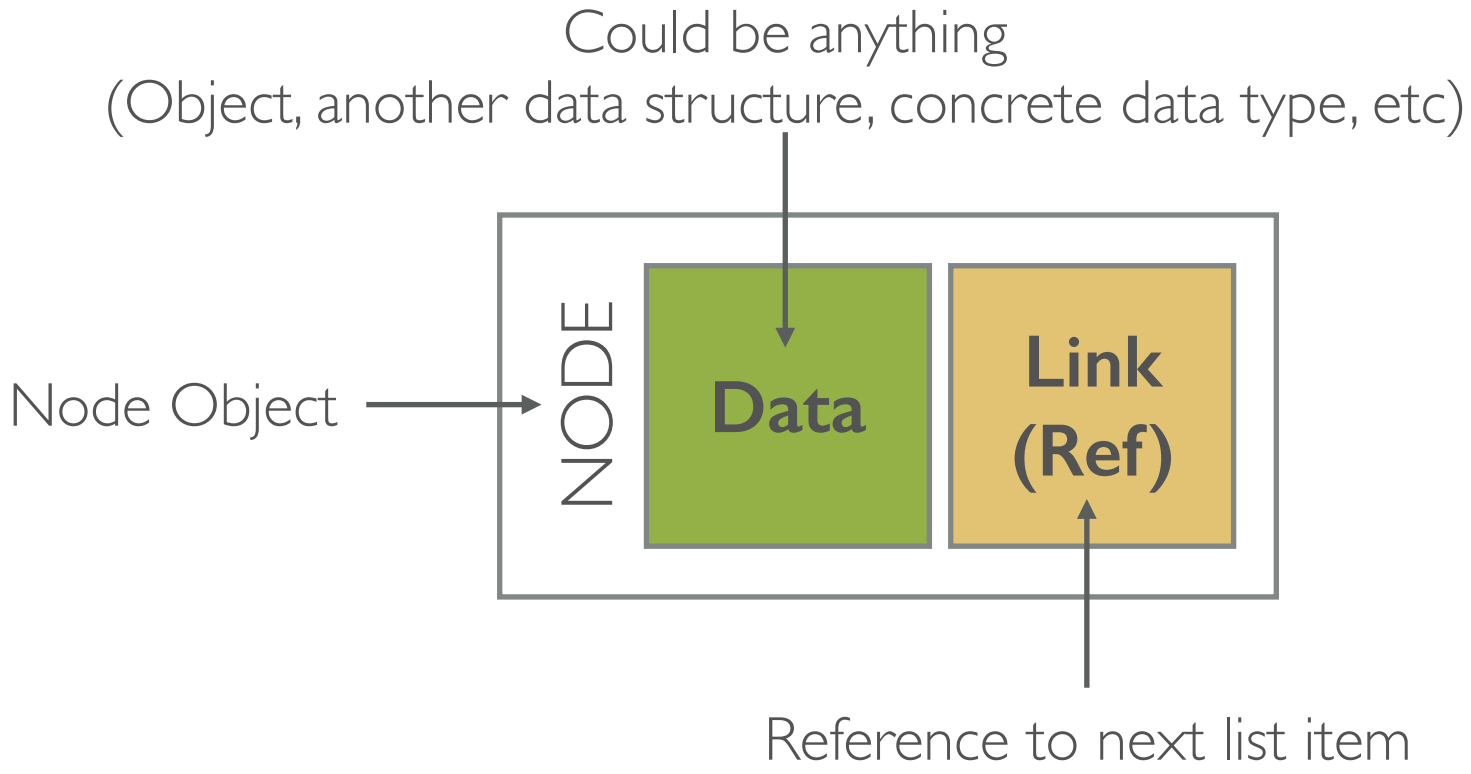
- A reference variable contains the location or memory address of an object.

- By using a reference to particular object, you can locate the object and access the objects public data and methods.

```
String myString = new String("Hello");
String ref = myString;
ref.toUpperCase();
```

# NODE STRUCTURE USING REFERENCES

- Each list item, called a node, contains a both data and a *link* to the next item.



# NODE STRUCTURE USING REFERENCES

- Each node in the linked list can be implemented as an object. The node data and link to the next node (ref) should be hidden (private) and should be accessed only through public methods (encapsulation).

- The following slide shows the Java code for a linked list node that stores integer values.

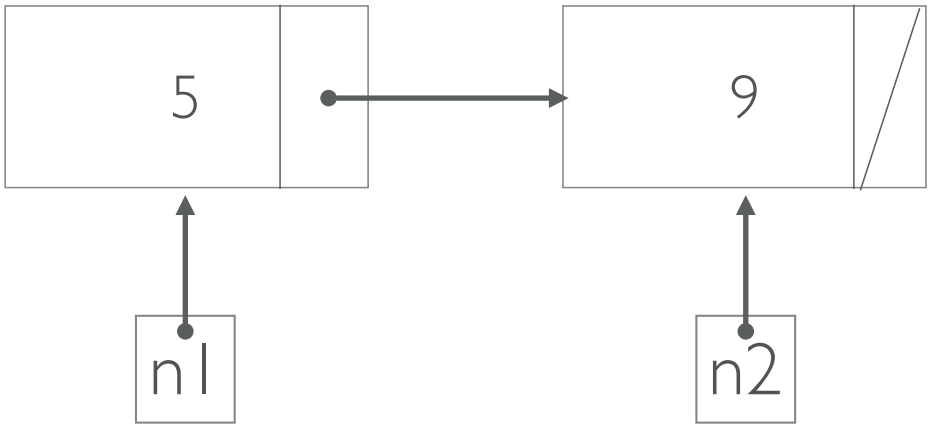
# INTEGER NODE JAVA CLASS

```
public class IntegerNode
{
  private int item;
  private IntegerNode next;

  public IntegerNode(int newItem)
  {
    item = newItem;
    next = null;
  } // end constructor

  public IntegerNode(int newItem, IntegerNode nextNode)
  {
    item = newItem;
    next = nextNode;
  } // end constructor

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

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

  public void setNext(IntegerNode nextNode)
  {
    next = nextNode;
  } // end setNext

  public IntegerNode getNext()
  {
    return next;
  } // end getNext

}  // end class IntegerNode
```

# USING THE INTEGER NODE CLASS

- We can use the IntegerNode class as follows:

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



# USING THE INTEGER NODE CLASS

- This definition of a node restricts the data to a single integer field.
- Since we would like to have this class be as reusable as possible, it would be better to change the data field to be of type **Object**.
- In Java, every class is ultimately derived from the class Object through inheritance.
- This means that any class created in Java could use this node definition for storing objects.

# OBJECT NODE JAVA CLASS

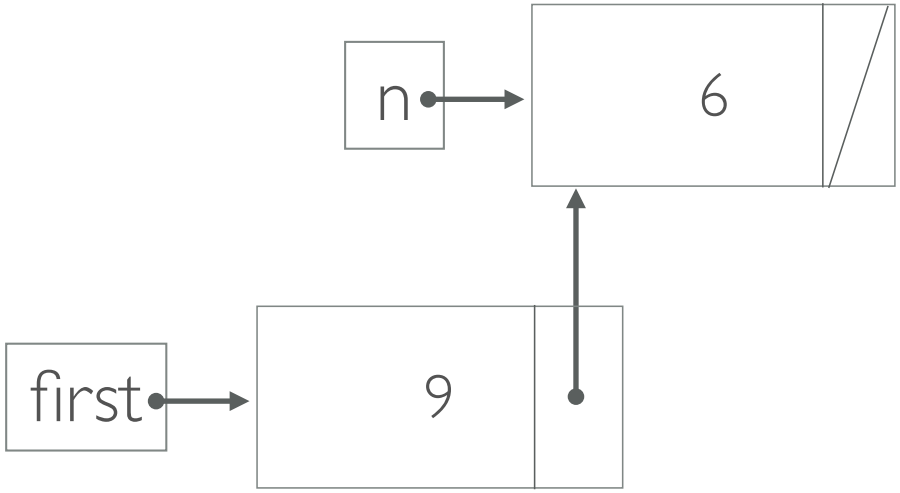
```
public class Node
{
  private Object item;
  private Node next;

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

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

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

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

  public void setNext(Node nextNode)
  {
    next = nextNode;
  } // end setNext

  public Node getNext()
  {
    return next;
  } // end getNext
} // end class Node
```

# USING THE OBJECT NODE CLASS

- We can use the Node class as follows:

NOTE: in this case we are using the second version of the **node** constructor to create the first node object.

```
Node n = new Node(new Integer(6));
Node first = new Node(new Integer(9), n);
```



# MAINTAINING A LINKED LIST (HEAD REF)

**Q:** What is the value of the data field next in the last node?

**A:** By setting this field to **null**, we can easily detect when we have reached the end of the linked list.

**Q:** Where can we find the start of the list?

**A:** As we have described the linked list so far, nothing references the beginning of the list.

# MAINTAINING A LINKED LIST (HEAD REF)

- If you do not know where a list begins, you cannot visit each element (**traverse**) the list.

- We therefore need an additional reference variable to help us keep track of the **first** node in the list. This reference variable is usually called the **head** of the list.

- The reference variable **head** does not exist within one of the nodes. Head is a simple reference variable that is external to the linked list.

# MAINTAINING A LINKED LIST (HEAD REF)

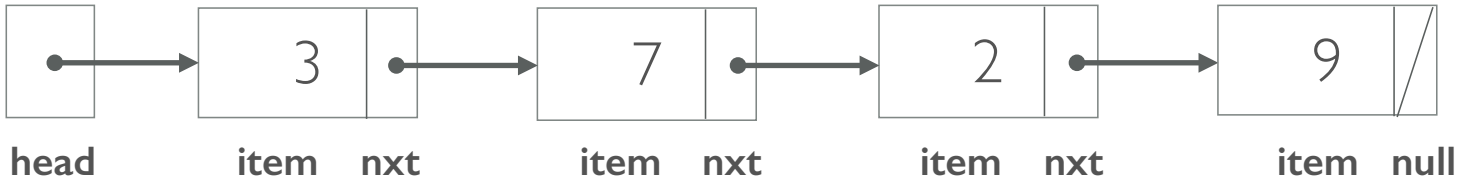
- The **head** reference always exists, even if there are no nodes in the list.

- The statement:

```
Node head = null;
```

- Creates the variable **head** whose initial value is null. This indicates that **head** does not reference anything and the list is empty.

# HEAD REFERENCE (DON'T LOOSE IT)





```
head = new Node(new Integer(3));
head = null;
```

# PROGRAMMING WITH LINKED LISTS

- We can develop algorithms to:
  - **display** the data portions of a linked list
  - **insert** items into a linked list
  - **delete** items from a linked list

- The linked list operations are the basis for many other data structures and will therefore be of great use to us.

# DISPLAYING THE CONTENTS OF A LINKED LIST

- Let us assume we have a linked list and want to display the data in the list. A high-level pseudocode solution might be:

- Let a variable **curr** reference the first node in the list (head).

```
Node curr = head;
while(the curr reference is not null){
   display the data portion of the current item;
   set the curr reference to the next field of the current
   node;
}
```

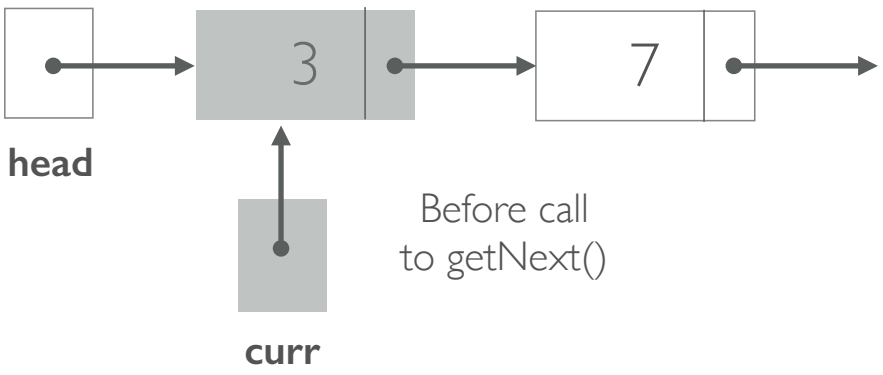
# DISPLAYING THE CONTENTS OF A LINKED LIST

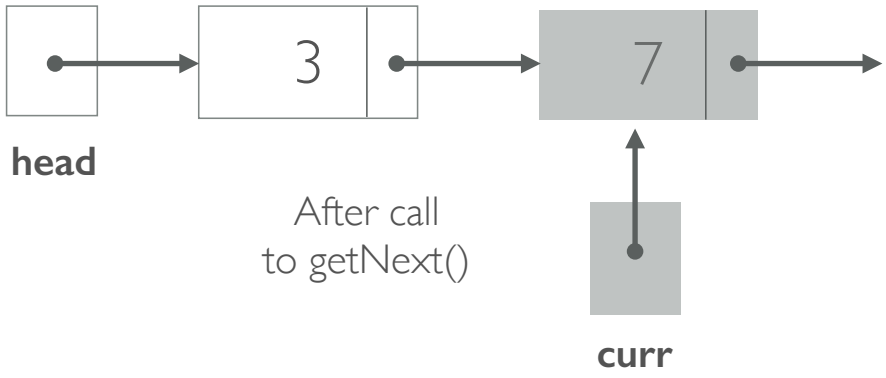
- To display the data portion of the current node, we use the Java statement:

```
System.out.println(curr.getItem());
```

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

```
curr = curr.getNext();
```



# DISPLAYING THE CONTENTS OF A LINKED LIST

- So some Java code to display the list contents might look as follows:

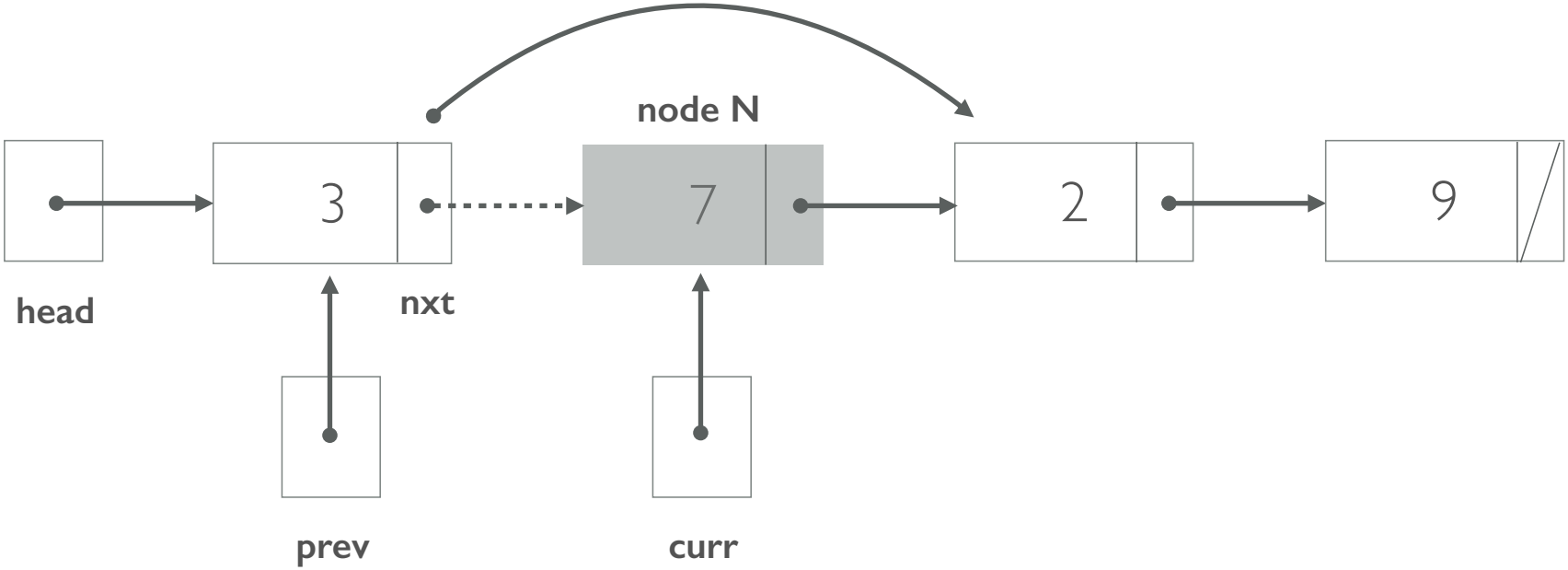
```
// Display the data in a linked list
// Loop invariant: curr references
// the next reference to be displayed
for(Node curr = head; curr != null; curr = curr.getNext())
{
    System.out.println(curr.getItem());
}
```

- The variable **curr** references each node in a non empty list during the course of the for loops execution. The data item is displayed at each iteration. After the last node is displayed, **curr** becomes **null** and the loop terminates.

# DELETING A SPECIFIED NODE FROM THE LIST

- In addition to the **head** reference, two other reference variables are needed during the deletion operation and they are **curr** and **prev**.

- In the diagram below, the task is to delete the node that **curr** references.



# DELETING A SPECIFIED NODE FROM THE LIST

- We can delete a node **N**, which **curr** references by altering the value of the reference **next** in the node that precedes node **N**.

- We need to set this field to reference the node that follows **N** thereby bypassing **N** on the chain.

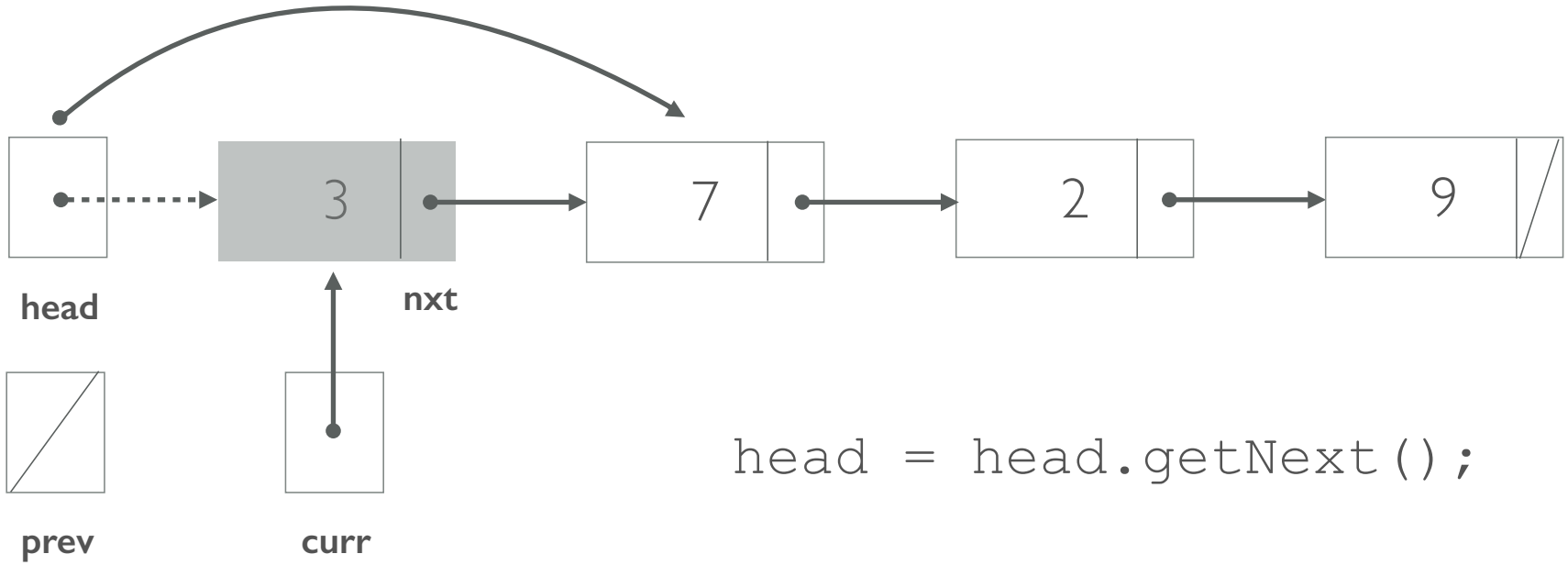
- The assignment statement to accomplish this looks as follows:

```
prev.setNext(curr.getNext());
```

# DELETING THE FIRST NODE

- Unfortunately we need to write a special case for deleting the first node in a linked list.

- 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.

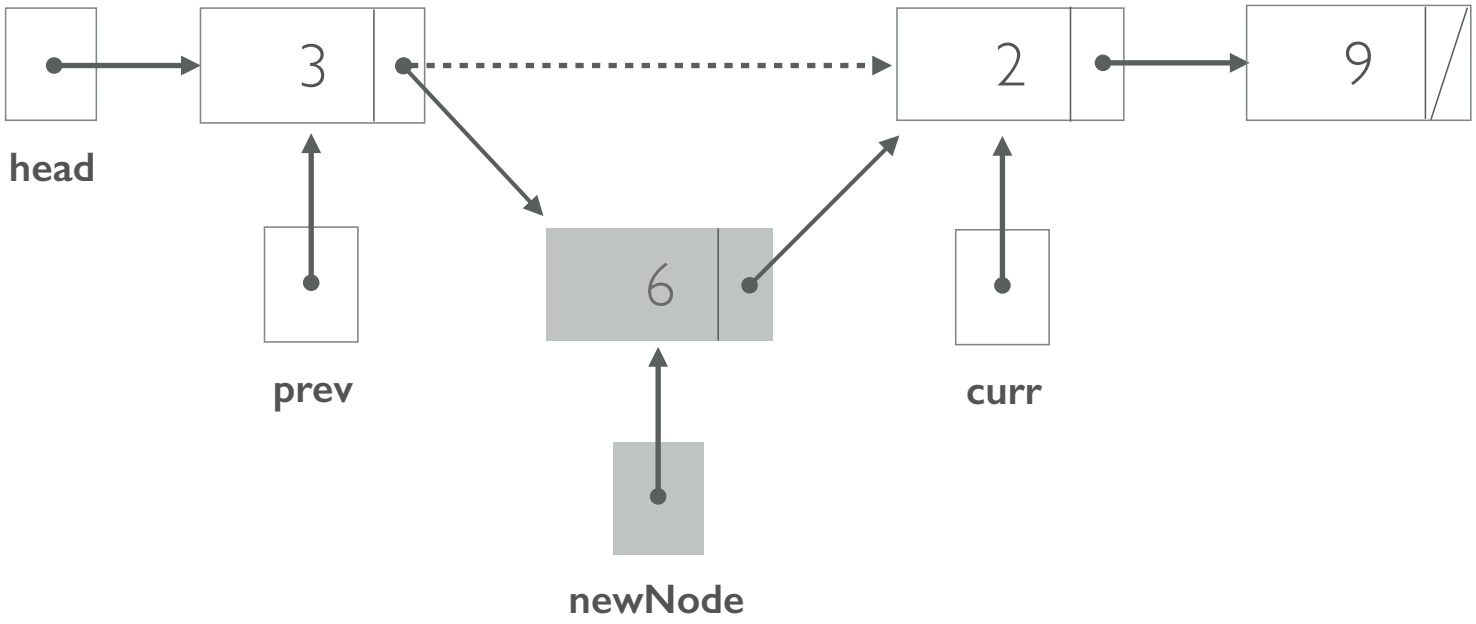



```
head = head.getNext();
```

# 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);
prev.setNext(newNode);
```



# POSITIONING THE NEW NODE

**Q:** Where does the newNode come from, …and go?

**A:** We establish **prev** and **curr** by traversing the linked list until we find the correct position on the list for the new item.

We use the **new** operator to create a new node

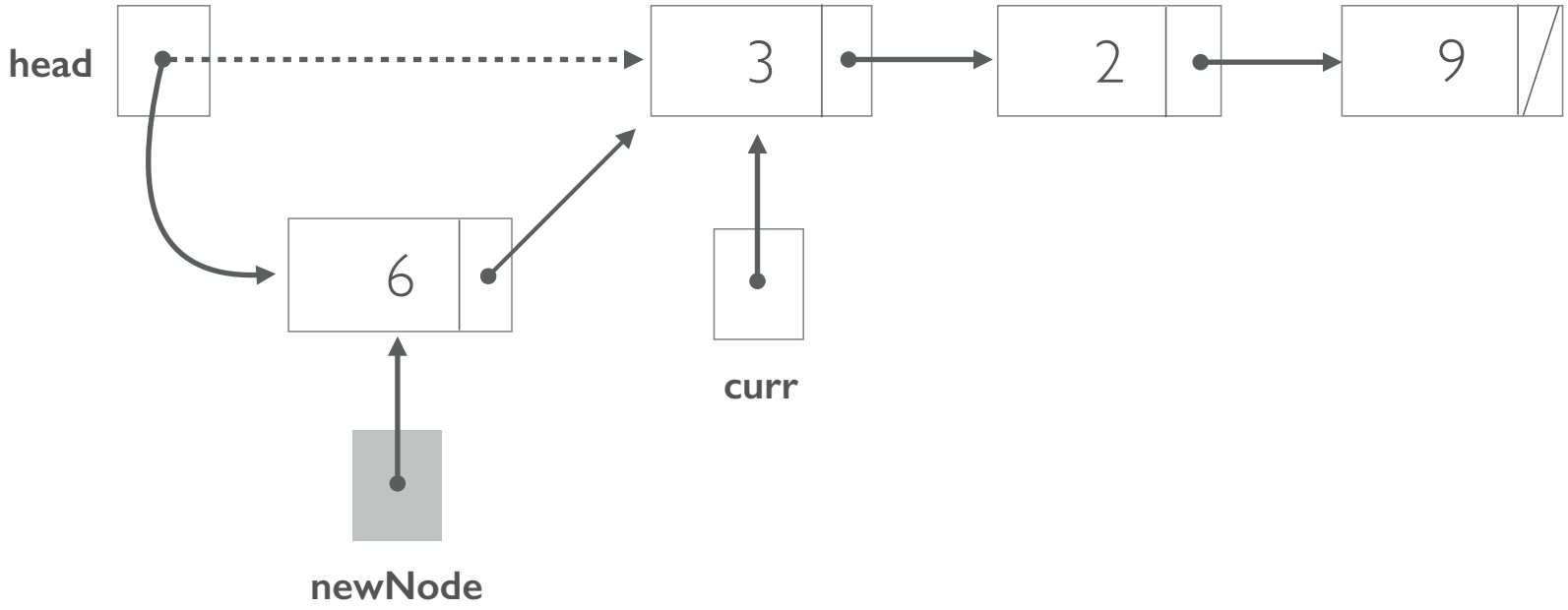
```
newNode = new Node(new Integer(6));
```

and we insert the **newNode** into the list as previously described.

# INSERTING AT THE HEAD

- Unfortunately we must deal with a special case for inserting new list items at the **head** of the list.

```
newNode.setNext(head);
head = newNode;
```



# INSERTION PROCESS

- The insertion process requires three high-level steps

1. **Determine** the point of insertion

2. **Create** a new node and store the new data in it

3. **Connect** the new node to the linked list by changing references.

# INSERTION INTO A SORTED LIST

- Assume we want to insert a new node into a sorted list. We need to determine the position of **prev** and **curr** based on the data value of the new node.

```
// Pseudocode
// Determine the point of insertion
// Initialise prev and curr to start traversal
prev = null;
curr = head;
// Advance prev and curr as long as newValue > current data item
// Do not go beyond the end of the list
while(curr != null && newValue > curr.getValue())
{
 prev = curr;
 curr = curr.getNext();
}
```

# COMPARISON OF IMPLEMENTATIONS

- **Arrays** (easy to use but have a fixed size)
  - The maximum number of items in an ADT can be difficult to predict
  - An array may waste memory
  - An array based implementation may be a good choice for a *small* list
- **Linked Lists** (no fixed size)
  - In a linked list, a node explicitly references the next node
  - You must traverse a linked list to access the **kth** node
  - The time to access the **kth** node in a linked list depends on **k**
  - Insertion and deletion on a linked list requires list traversal
  - Insertion and deletion on a linked list does not require shifting data

# TODO - WEEK 2

- Implement the ADT List using a referenced based linked list
  - Write out the specifications for the ADT List
  - Implement the ADT List as a Java class using references such that the "walls" are reinforced (see sample code).
  - This class must implement the ADT operations as public methods
  - 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