

Unit 04: Lists

Anthony Estey

CSC 115: Fundamentals of Programming II

University of Victoria

Unit 04 Overview

- ▶ Related Reading:

- ▶ Textbook Chapter 5

- ▶ Learning Objectives: (You should be able to...)

- ▶ understand the concept of and be able to implement the Node data structure
 - ▶ understand the concept of a linked list data structure, consisting of a linked collection of nodes
 - ▶ describe the differences between arrays and linked lists from a memory perspective
 - ▶ implement the List ADT operations with both a singly and doubly linked list
 - ▶ describe the running time of list operations implemented using a linked list, specifically when compared to an array implementation

The List ADT

- ▶ ADT List Operations:
 - ▶ Create an empty list
 - ▶ Determine whether a list is empty
 - ▶ Determine the number of items in a list
 - ▶ Add an item at a given position in a list
 - ▶ Remove the item at a given position in a list
 - ▶ Get the item at a given position in a list
 - ▶ Remove all items from a list
- ▶ Items are referenced by their position in a list:
 - ▶ (1st, 2nd, 3rd, etc)

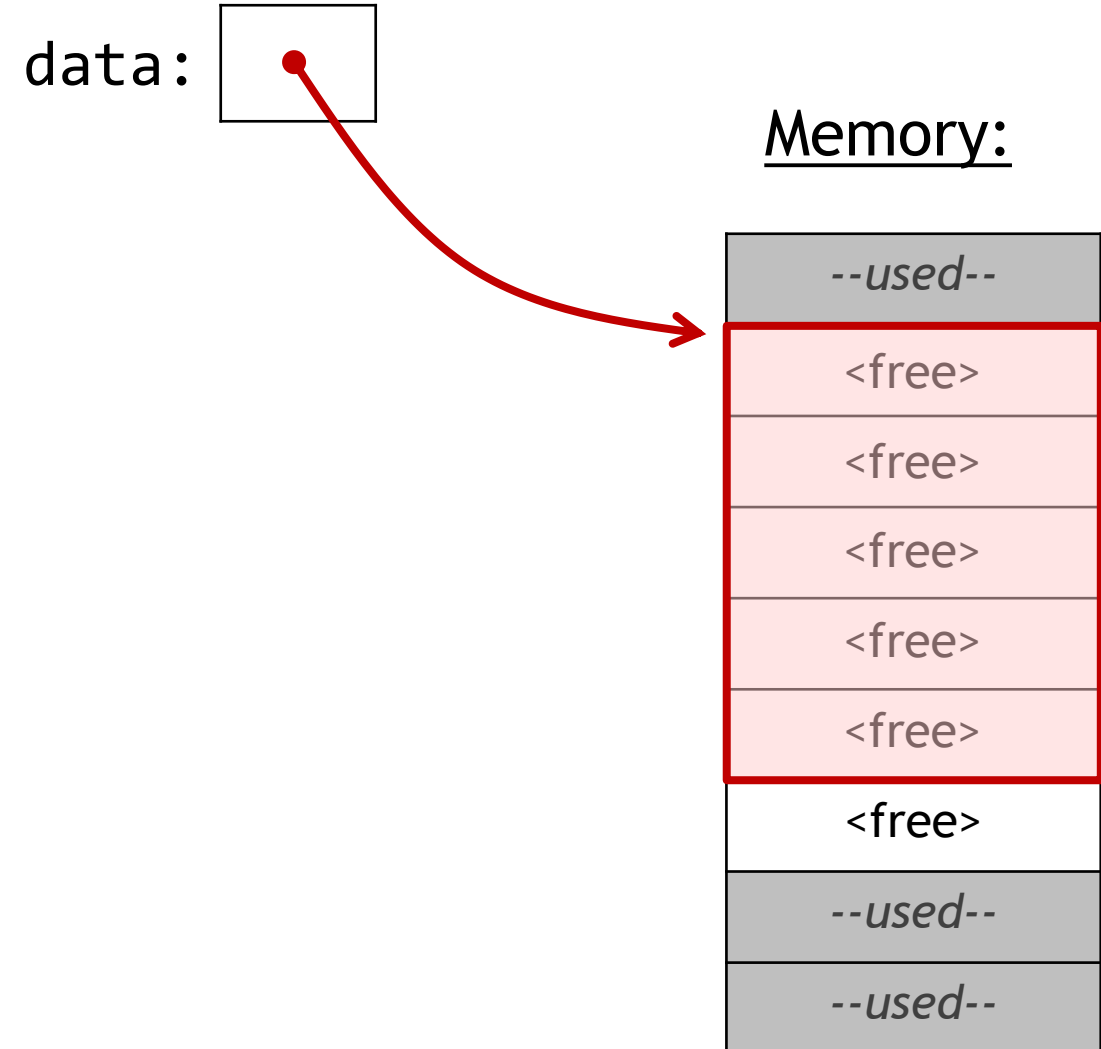
Last unit we introduced
the idea of the List ADT

List Implementation

- ▶ In this unit, we will explore two ways of implementing a list
 - ▶ Using an array (something we are familiar with by now)
 - ▶ Using a linked list (something we will see for the first time)
- ▶ After exploring the two different implementations, we will evaluate the strengths and weaknesses of both approaches, and specifically focus on the following:
 - ▶ Memory usage
 - ▶ Runtime speed

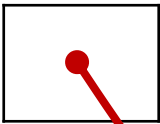
Array implementation

```
int[] data = new int[SIZE];
```



Array implementation

```
int[] data = new int[SIZE];
```

data: 

Memory:

► Positives:

► Easy to access and update array values:

```
data[0] = 3;
```

```
data[1] = 9;
```

```
data[2] = 14;
```

```
System.out.println(data[1]); // outputs 9
```

Very fast operations!



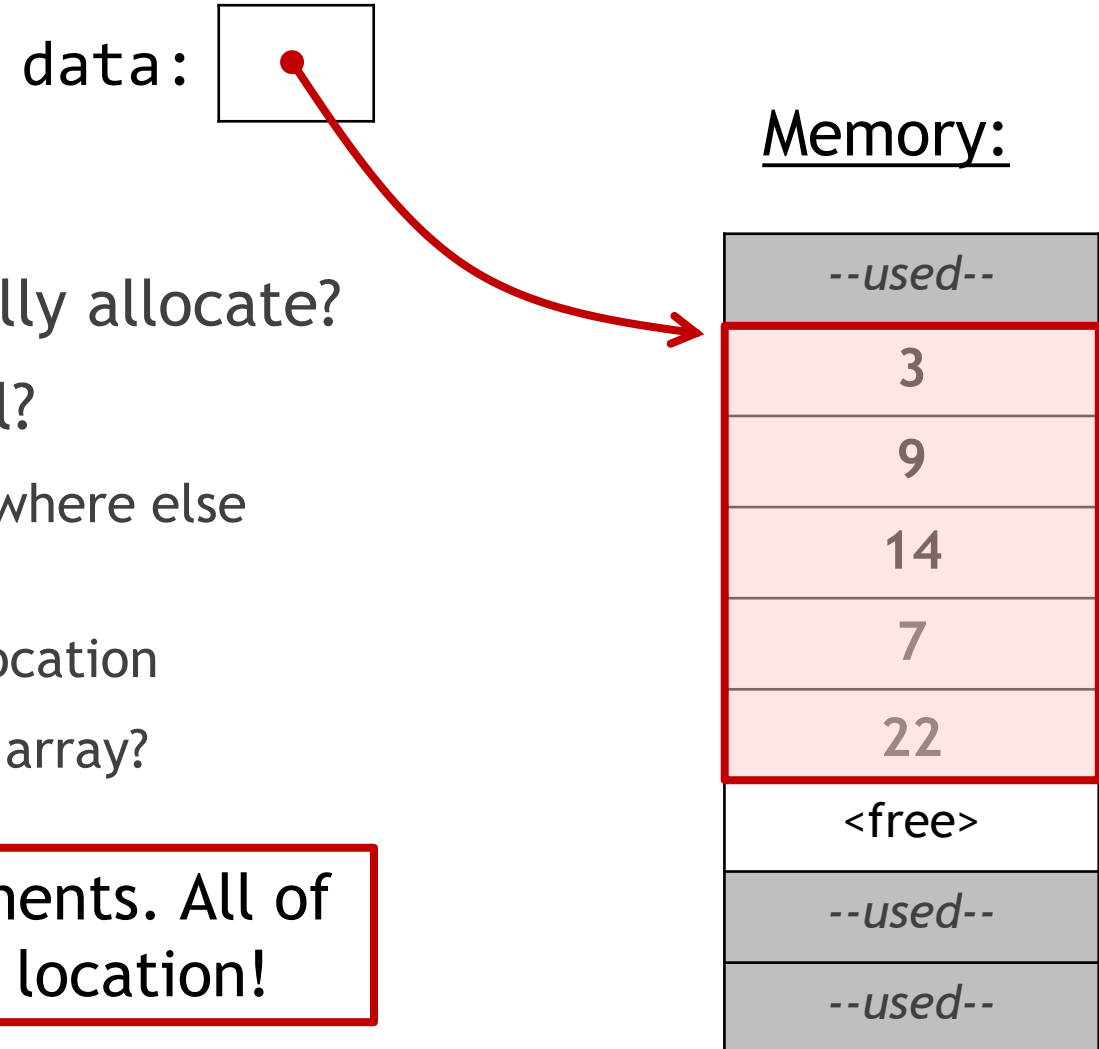
Array implementation

► Negatives:

► Memory usage:

- How much memory should we initially allocate?
- What happens when the array is full?
 - We will need to allocate memory somewhere else (find a location in memory big enough)
 - And copy all of the values to the new location
 - How big should we make the expanded array?

Imagine an array with a billion elements. All of them need to be copied to the new location!



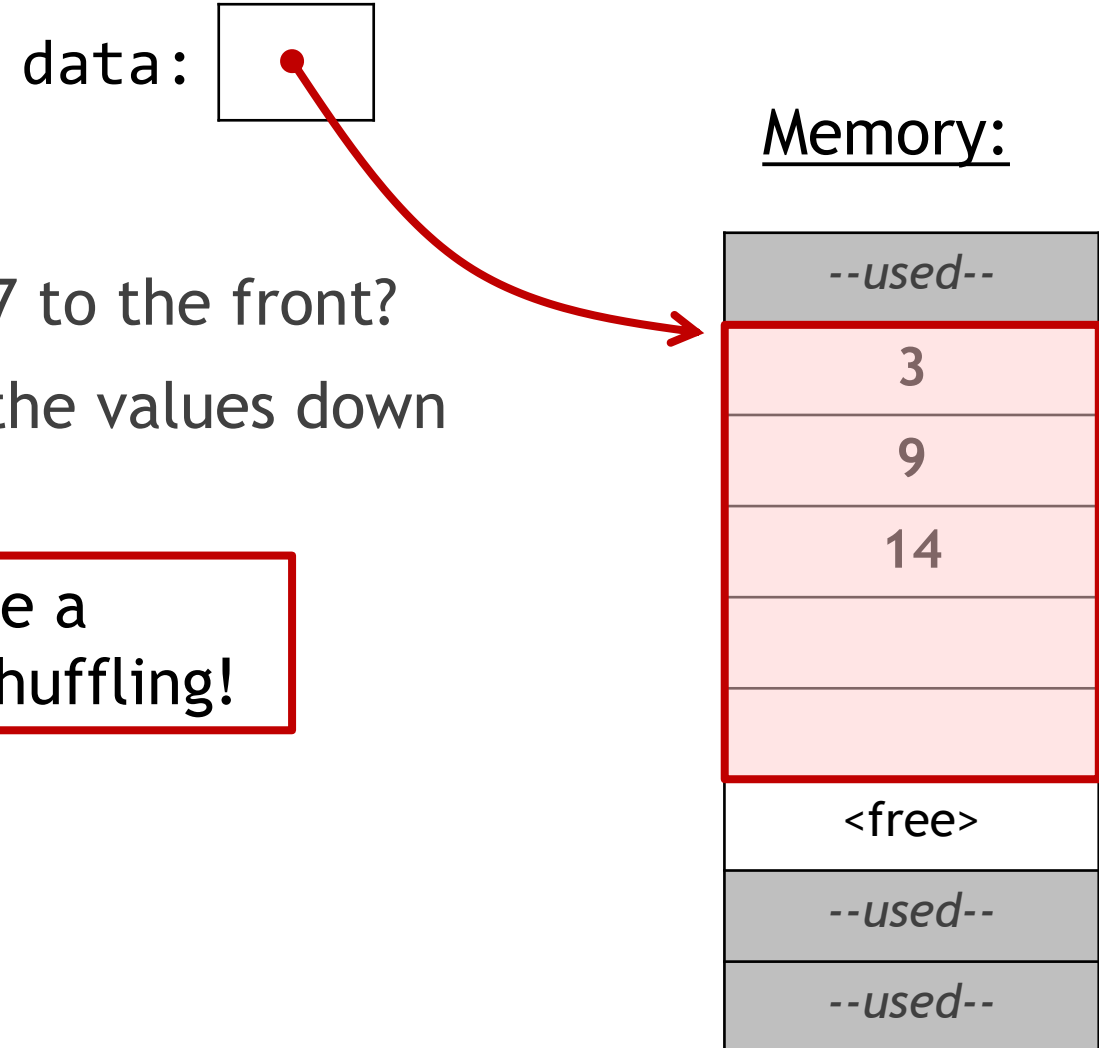
Array implementation

- ▶ Negatives:

- ▶ Maintaining element order

- ▶ What happens if we want to insert 7 to the front?
- ▶ We will need to shuffle the rest of the values down

Insert to the front when there are a billion elements means a lot of shuffling!



Summary: Array implementation

- ▶ Positives:

- ▶ Easy to access and update values

- ▶ Negatives:

- ▶ Memory allocation:

- ▶ allocate a lot of extra space and there is wasted memory
 - ▶ not enough space means we need to keep expanding the array and copying all of the array data into the new location in memory

- ▶ Insertion:

- ▶ inserting a single element may require a complete reshuffling

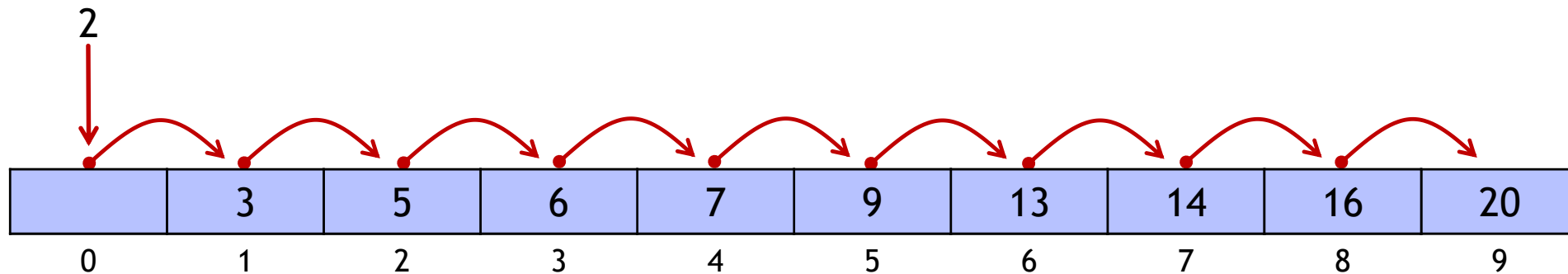
Linked Lists - Motivation

- ▶ Imagine we have chosen to implement the List ADT using an array...
- ▶ Assume our list currently has the following items in it:

3	5	6	7	9	13	14	16	20	
0	1	2	3	4	5	6	7	8	9

Linked Lists - Motivation

- ▶ Imagine we have chosen to implement the List ADT using an array...
- ▶ Assume our list currently has the following items in it:
- ▶ What would happen if we call `addFront(2);`

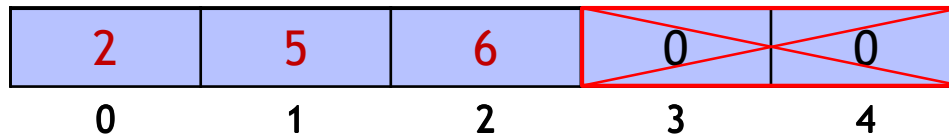


- ▶ A linked list allows for fast insertion/removal from the front or back of a list without the need to **shuffle** all other items up or down an index

What is a linked list?

- ▶ Assume we want to implement a list...
- ▶ Array:
 - ▶ allocate space...
 - ▶ insert elements 2, 5, and then 6, one at a time

```
int[] data = new int[SIZE];
```



Create a smaller array?

When it fills up, a new larger array must be allocated, and all of the elements need to be copied over

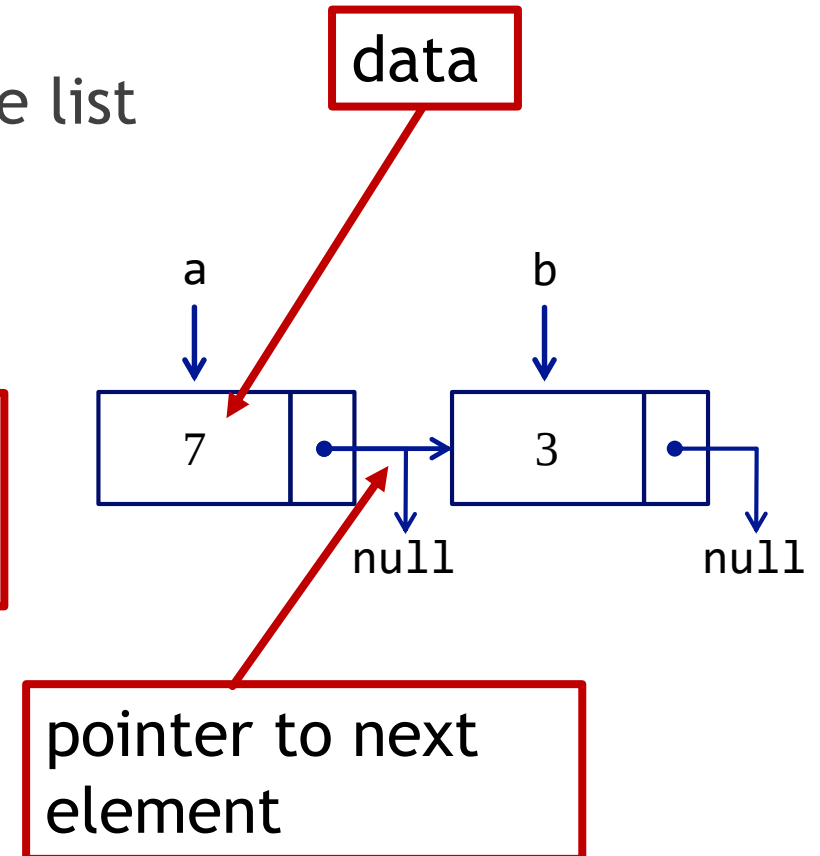
In Java, arrays have a **fixed length**

What is a linked list?

- ▶ A **linked list** is a data structure composed of **nodes** linked together
- ▶ A **node** is a data structure that contains:
 - ▶ data (whatever we want to store in the list)
 - ▶ a pointer to the location of the next element in the list
 - ▶ (sometimes a pointer to the previous element too)

```
public class Node {  
    private int data;  
    private Node next;  
    ...  
}
```

```
Node a = new Node(7, null);  
Node b = new Node(3, null);  
a.next = b;
```



Iteration implementation

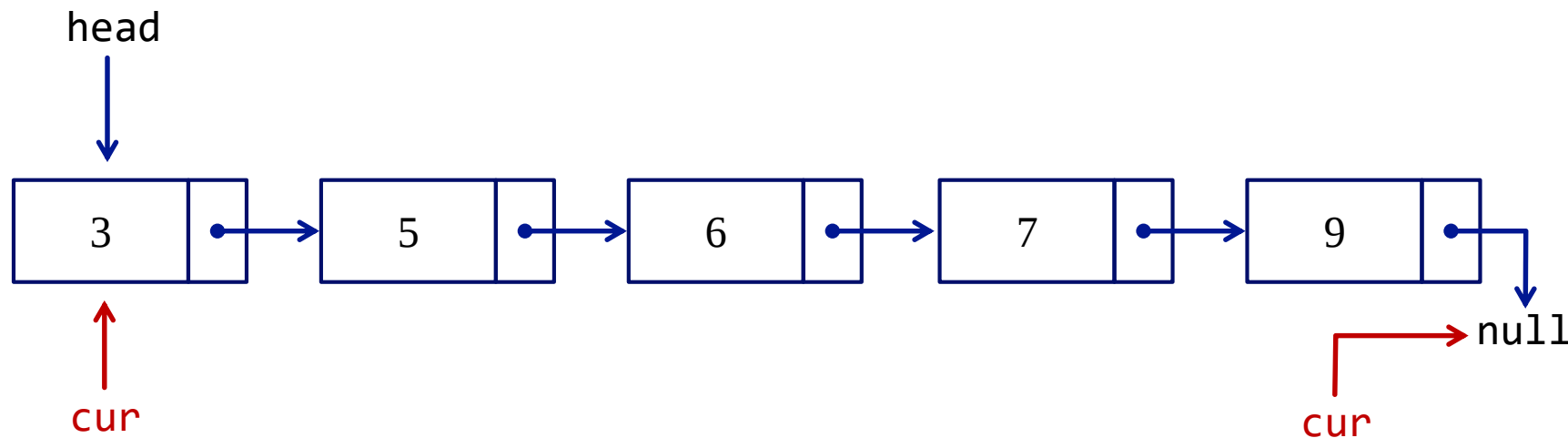
- We know how to iterate through all of the items in an array:

3	5	6	7	9
0	1	2	3	4

```
for (int i = 0; i < array.length; i++) {  
    System.out.println(array[i]);  
}
```

Iteration implementation

- ▶ With linked lists, we need to keep a reference to the head of the list. From there, we can reach all subsequent elements:



```
for (Node cur = head; cur != null; cur = cur.next) {  
    System.out.println(cur.data);  
}
```

Iteration

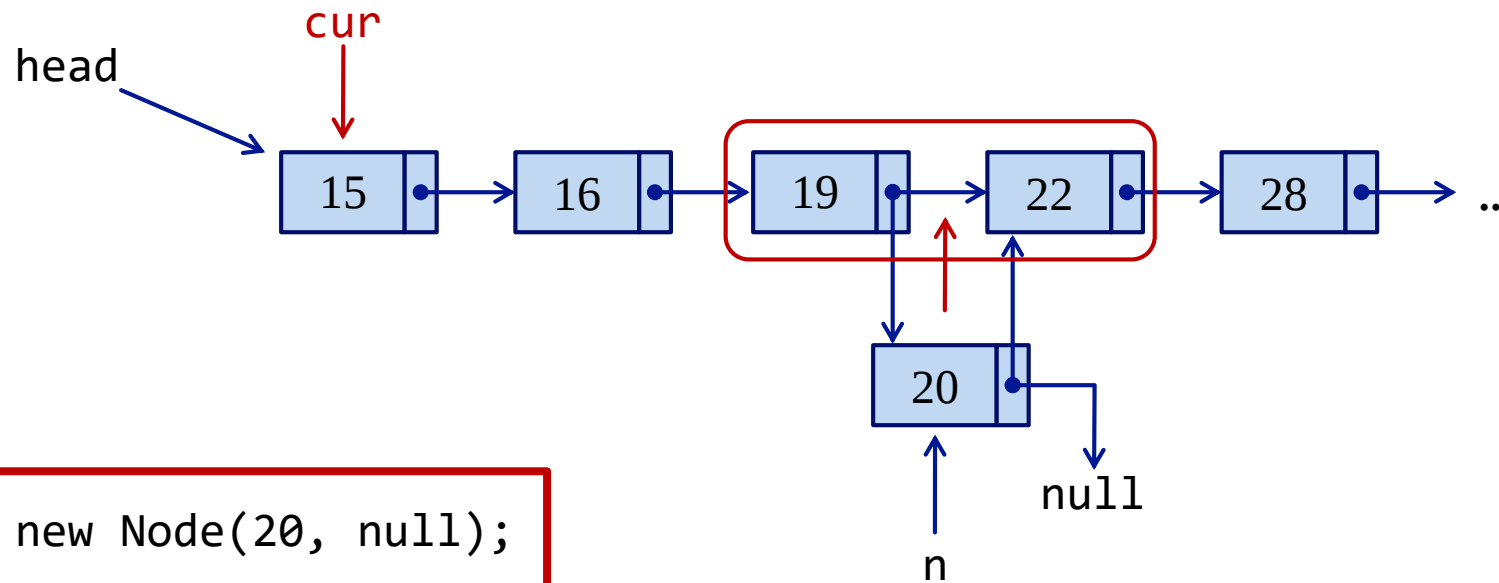
- ▶ With linked lists, we need to keep a reference to the head of the list. From there, we can reach all subsequent elements:

```
Node cur = head;
while (cur != null) {
    System.out.println(cur.data);
    cur = cur.next;
}
```

```
for (Node cur = head; cur != null; cur = cur.next) {
    System.out.println(cur.data);
}
```

Insertion

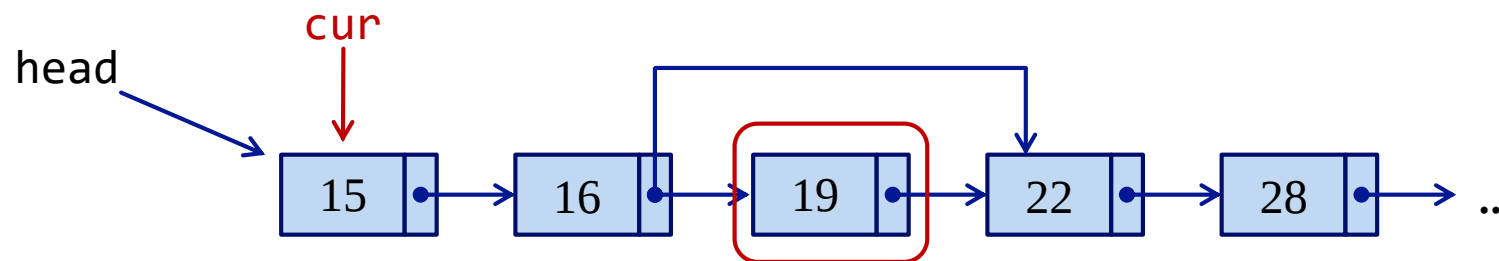
- ▶ First, determine where to insert the new node
- ▶ Then, update the next pointers appropriately



```
Node n = new Node(20, null);  
n.next = cur.next;  
cur.next = n;
```

Removal

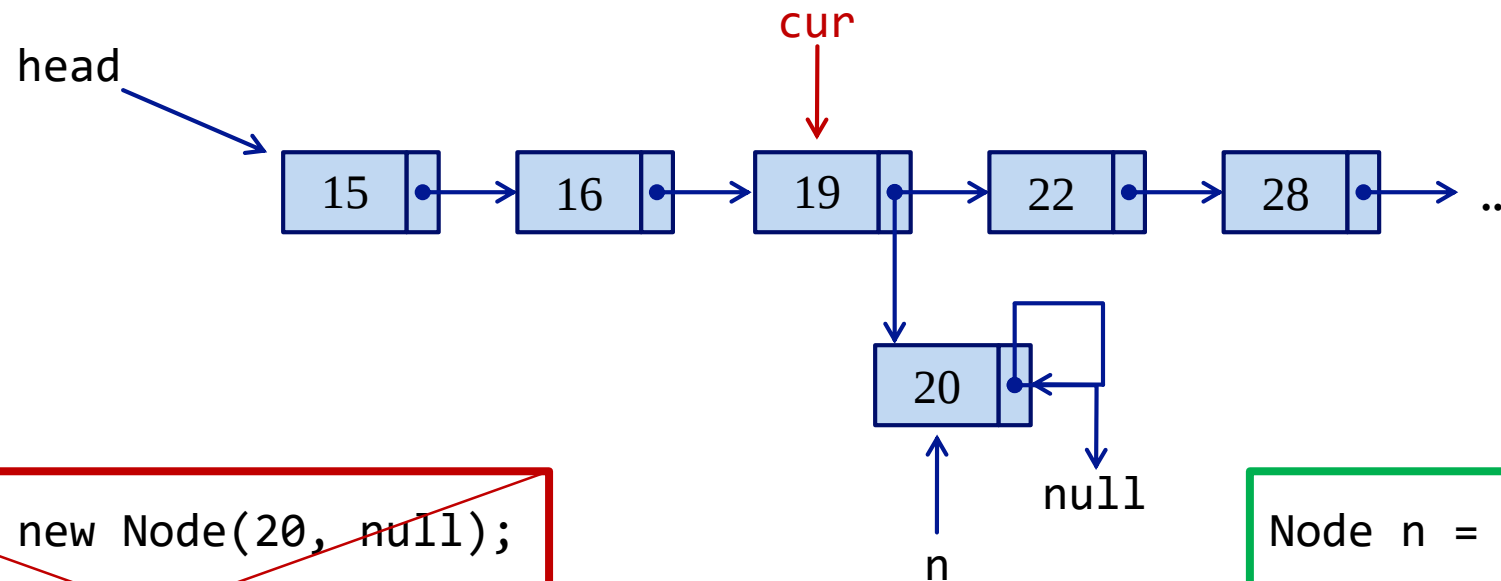
- ▶ First, locate the element *preceding* the one to remove
- ▶ Then, update the next pointers so that the deleted node is skipped
 - ▶ Java's garbage collection will delete of any object that nothing points to



```
cur.next = cur.next.next;
```

Order of operations is important!

- Let's revisit our insertion example, and assume we want to insert a node with data value 20 between node's 19 and 22.

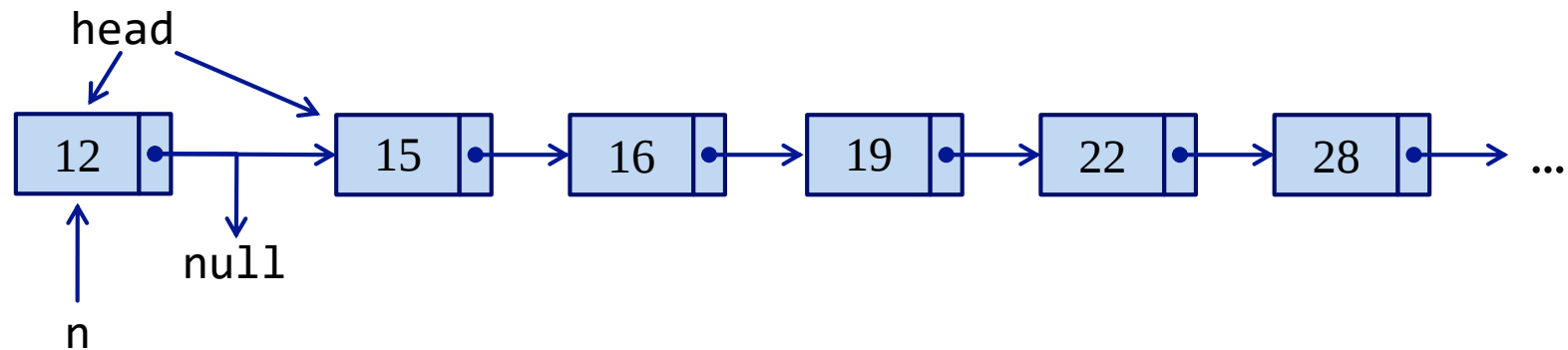


```
Node n = new Node(20, null);  
cur.next = n;  
n.next = cur.next;
```

```
Node n = new Node(20, null);  
n.next = cur.next;  
cur.next = n;
```

Adding an item to the front of a list

- ▶ First, determine where to insert the new node
- ▶ Then, update the next pointers appropriately

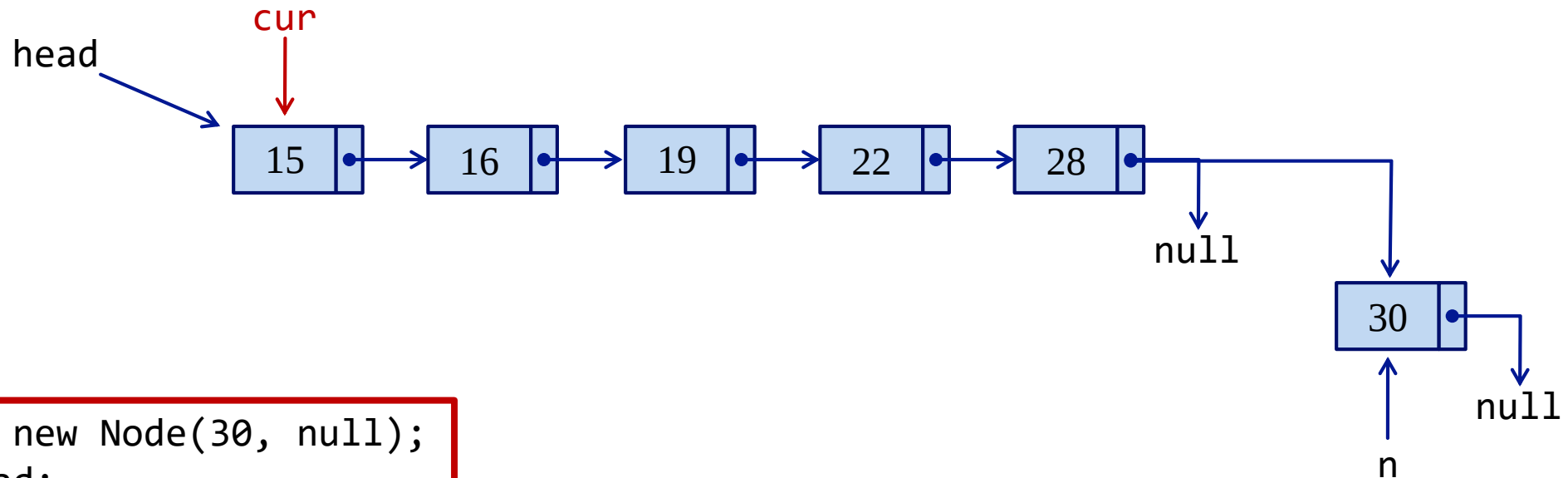


```
Node n = new Node(12, null);  
n.next = head;  
head = n
```

```
Node n = new Node(12, head);  
head = n
```

Adding an item to the back of a list

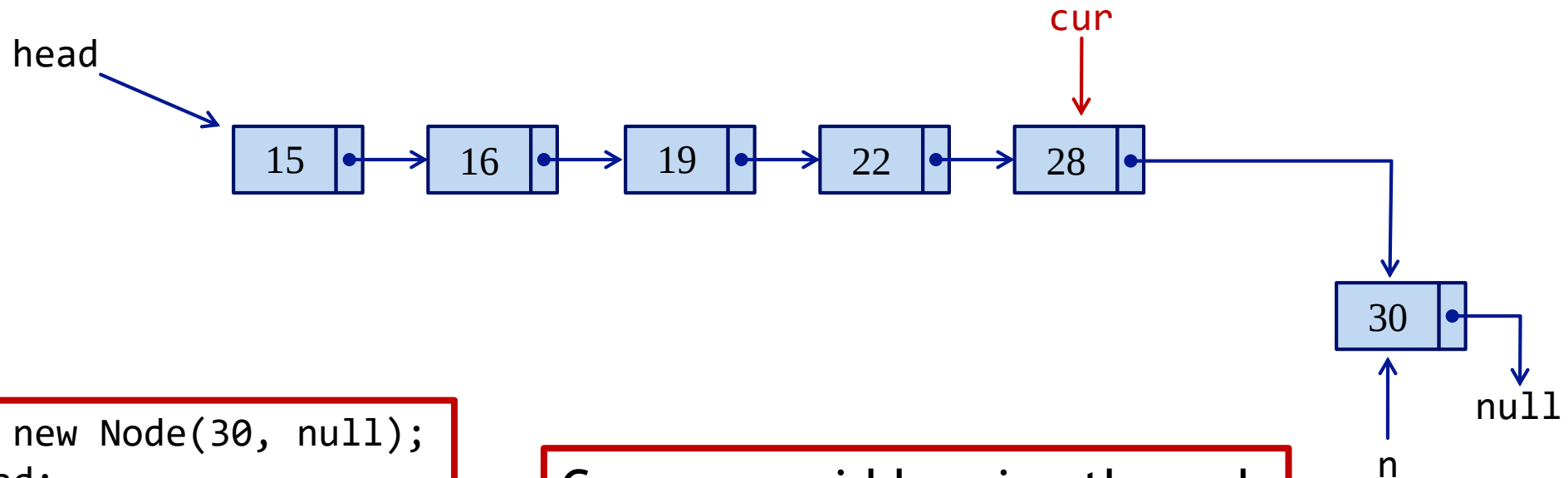
- ▶ First, determine where to insert the new node
- ▶ Then, update the next pointers appropriately



```
Node n = new Node(30, null);  
cur = head;  
while (cur.next != null) {  
    cur = cur.next;  
}  
cur.next = n;
```

Adding an item to the back of a list

- ▶ First, determine where to insert the new node
- ▶ Then, update the next pointers appropriately

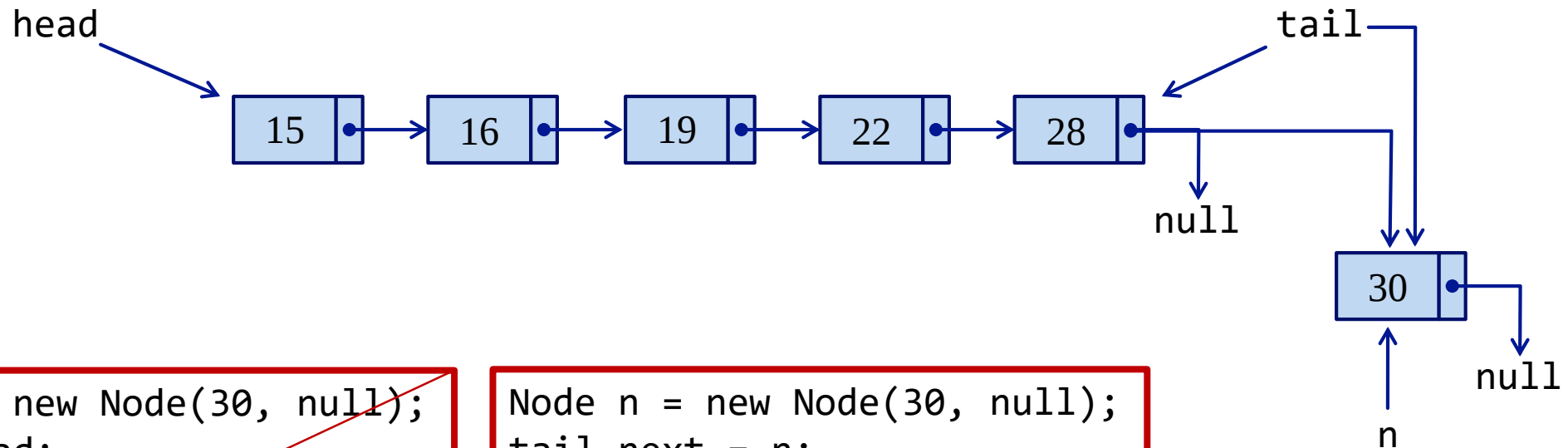


```
Node n = new Node(30, null);  
cur = head;  
while (cur.next != null) {  
    cur = cur.next;  
}  
cur.next = n;
```

Can we avoid looping through the whole list in order to insert an item at the back?

Tail Reference

- Idea: We have a reference to the front (head) of our list
 - Why don't we do the same with the back (tail)



```
Node n = new Node(30, null);  
cur = head;  
while (cur.next != null) {  
    cur = cur.next;  
}  
cur.next = n;
```

```
Node n = new Node(30, null);  
tail.next = n;  
tail = n;
```

Linked List Variations

```
public class IntegerLinkedList implements IntegerList {  
    private int numElements;  
    private Node head;  
  
    public IntegerLinkedList() {  
        head = null;  
  
        numElements = 0;  
    }  
    ...  
}
```

- ▶ Suppose we have a linked list with a head reference as defined above
 - ▶ operations at the back of the list have require us to traverse the whole list

Linked List Variations

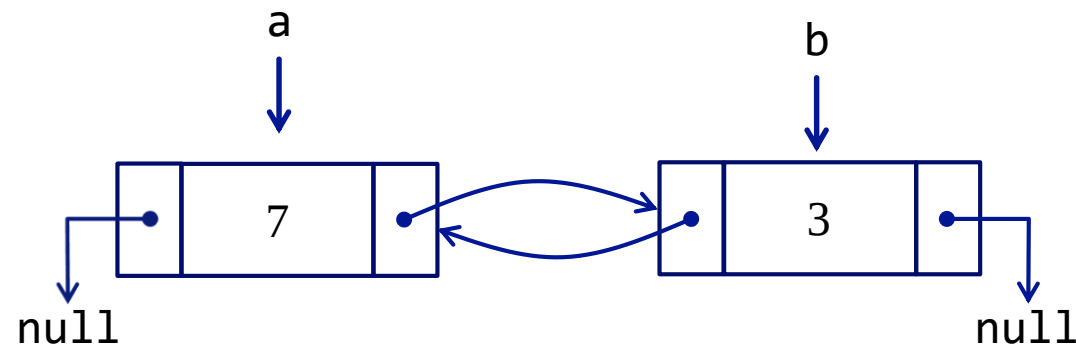
```
public class IntegerLinkedList implements IntegerList {  
    private int numElements;  
    private Node head;  
    private Node tail;  
  
    public IntegerLinkedList() {  
        head = null;  
        tail = null;  
        numElements = 0;  
    }  
    ...  
}
```

- ▶ Suppose we have a linked list with a head reference as defined above
 - ▶ operations at the back of the list have require us to traverse the whole list
 - ▶ Adding a tail reference requires very little overhead, and makes inserting/removing from the back of the list much more efficient

Doubly-linked list

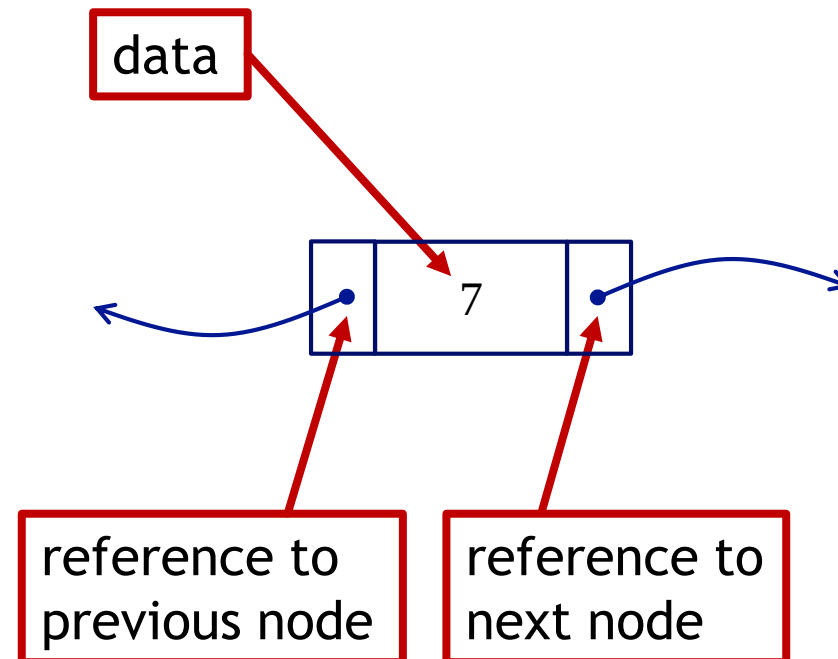
- ▶ A **doubly-linked list** is a linked list where each node keeps a reference to both the preceding *and* following nodes in the chain.
- ▶ A **node** is a data structure that contains:
 - ▶ data (whatever we want to store in the list)
 - ▶ a pointer to the location of the next element in the list
 - ▶ (sometimes a pointer to the previous element too)

```
public class Node {  
    private int data;  
    private Node prev;  
    private Node next;  
    ...  
}
```



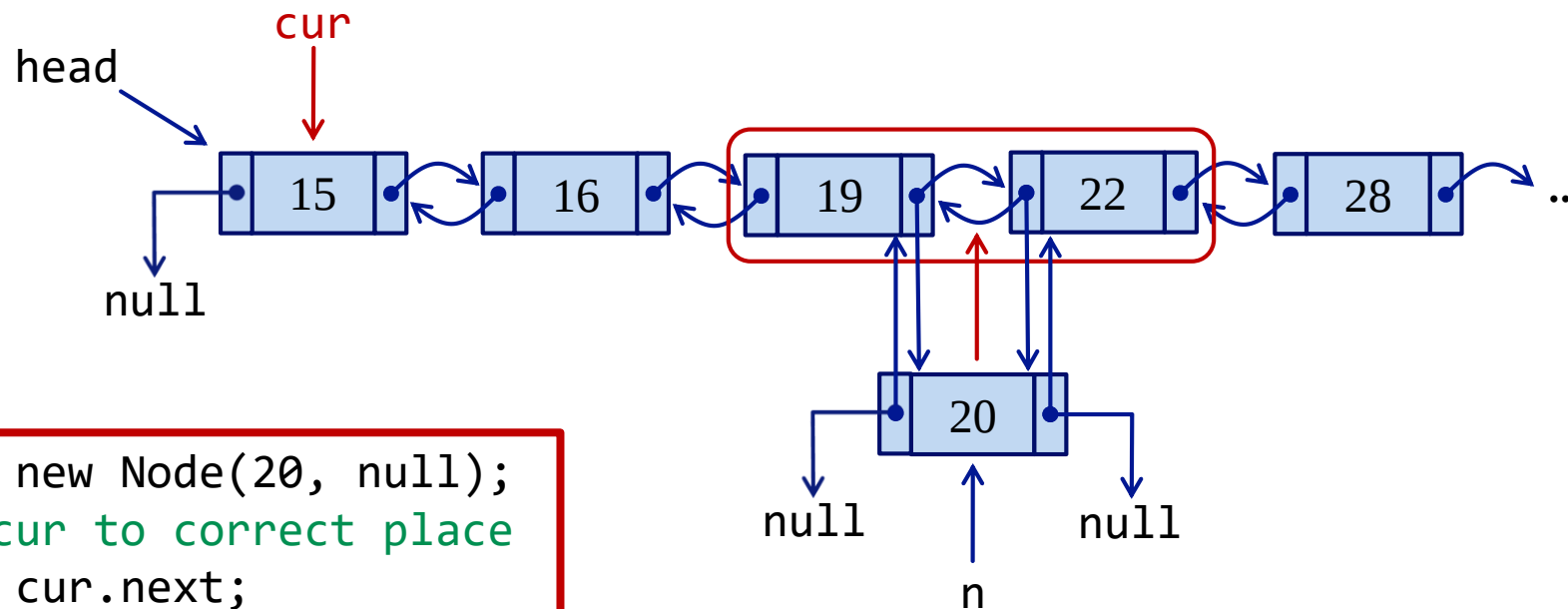
Doubly-linked List

- ▶ Node definition contains an additional reference pointer
 - ▶ This allows us to traverse the list in either direction!



Insertion

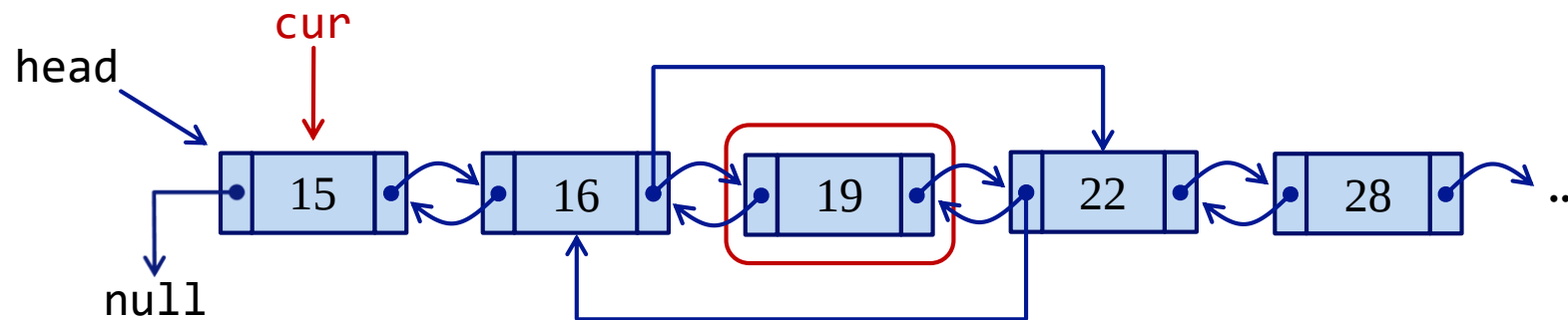
- ▶ First, determine where to insert the node
- ▶ Then, update pointers so that the order is correct



```
Node n = new Node(20, null);  
// move cur to correct place  
n.next = cur.next;  
n.prev = cur;  
cur.next.next = n;  
cur.next = n;
```

Removal

- ▶ First, locate the element to remove
- ▶ Then, update the next pointers so that the deleted node is skipped



```
cur.next.prev = cur.prev;  
cur.prev.next = cur.next;  
cur = null;
```