# Linked Lists

```
public class StrangeObject {
    String name;
    StrangeObject other;
}
```

# Preliminaries

- Arrays are not always the optimal data structure:
  - An array has fixed size – needs to be copied to expand its capacity
  - Adding in the middle of an array requires copying all subsequent elements
- ArrayLists have the same issues since they use arrays to store their data.

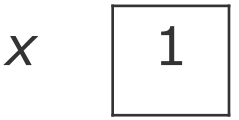
# Objects and references

- Object variables do not actually store an object; they store the address of an object's location in the computer's memory (references / pointers).

- Example:

```
int [] values  = new int[5];
```

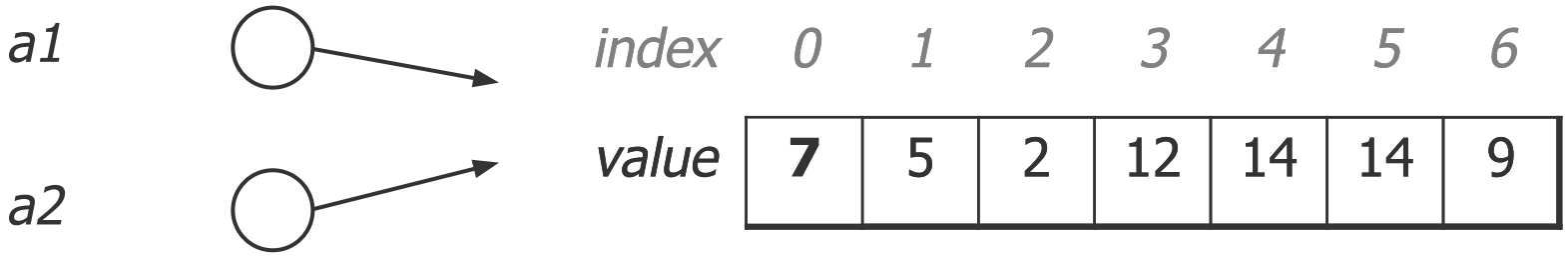

```
int x = 1;
```



# Java References

- When one reference variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.

```
int[] a1 = {4, 5, 2, 12, 14, 14, 9};
int[] a2 = a1; //refers to same array as a1
a2[0] = 7;
System.out.println(a1[0]);    // 7
```



# Self references

- Consider the following class:

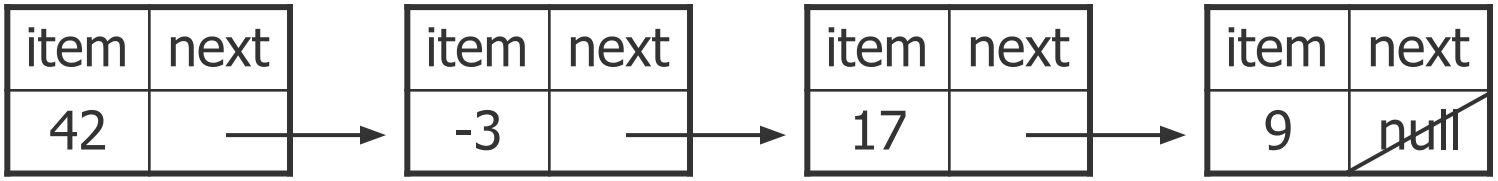
```
public class StrangeObject {
    String name;
    StrangeObject other;
}
```

  - Will this compile?

# Linking self-referential nodes

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

- Each node object stores:
  - one piece of integer data
  - a reference to another node
- `IntegerNode` objects can be "linked" into chains to store a list of values:



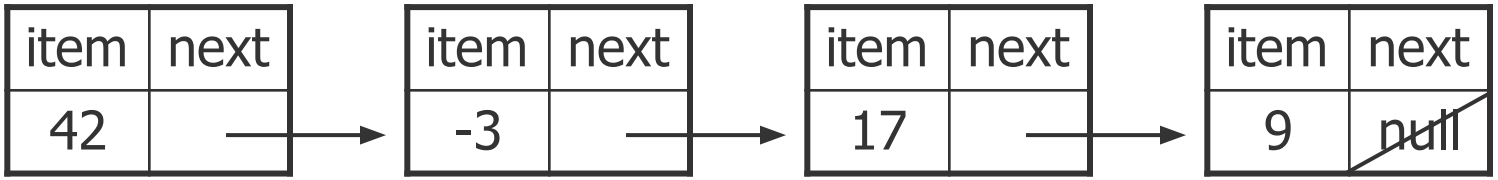
# The complete IntegerNode class

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

      public IntegerNode(int item) {
          this.data = item;
          this.next = null;
      }
      public IntegerNode(int item, IntegerNode next) {
          this.item = item;
          this.next = next;
      }
      public void setNext(IntegerNode nextNode) {
          next = nextNode;
      }
      public IntegerNode getNext() {
          return next;
      }
      public Object getItem() {
          return item;
      }
      public void setItem(Object item){
          this.item = item;
      }
}
```

# Exercise

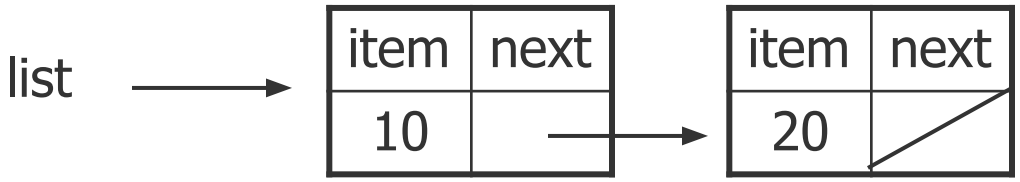
```
public class IntegerNode {
 private int item;
 private IntegerNode next;

 public IntegerNode(int item) {...}

 public IntegerNode(int item, IntegerNode next) {...}

 public void setNext(IntegerNode nextNode) {...}

 public IntegerNode getNext() {...}
}
```

Exercise: Write code to produce the following list

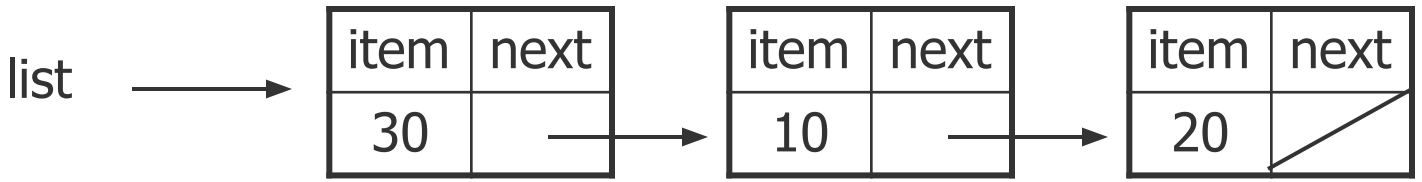


# Exercise

- What set of statements turns this list:

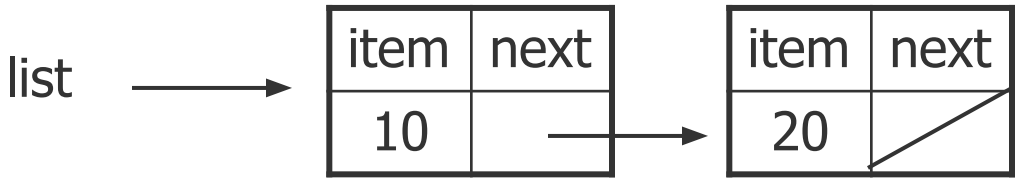



- Into this?

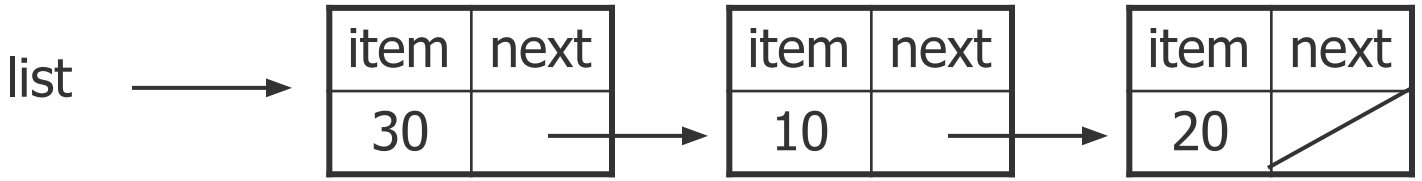


# Exercise

- What set of statements turns this list:



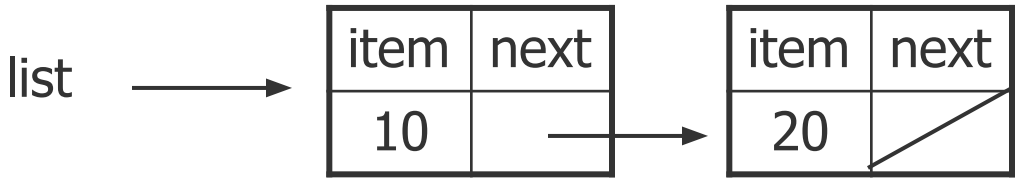

- Into this?




```
list = new IntegerNode(30, list);
```

# Exercise

- Let's write code that creates the following list:




Which is correct?

a)

```
list = new IntegerNode(10, new IntegerNode(20));
```

b)

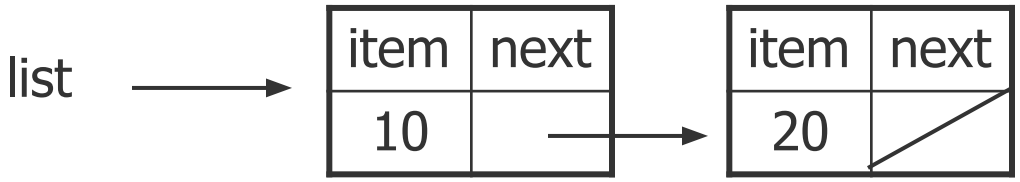
```
list = new IntegerNode(20, new IntegerNode(10));
```

c)

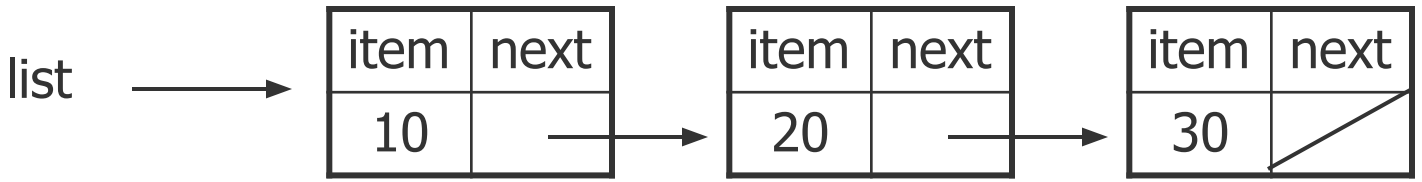
```
Neither will correctly produce that list
```

# Exercise

- What set of statements turns this list:

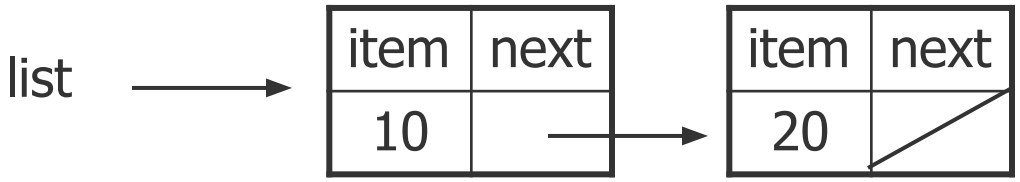



- Into this?

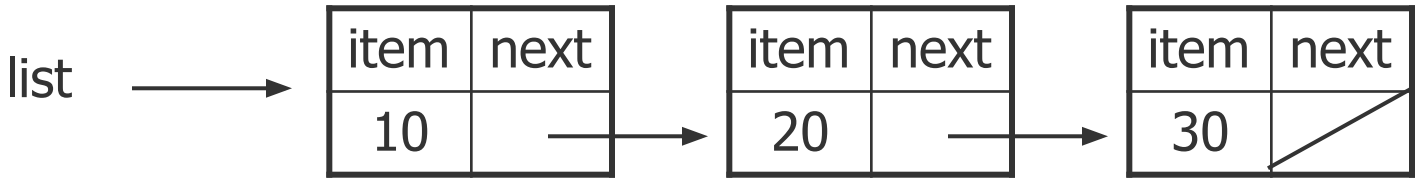


# Exercise

- What set of statements turns this list:




- Into this?




```
list.getNext().setNext(new IntegerNode(30));
```

# A more flexible version

```
public class Node {
     private Object item;
     private Node next;
     public Node(Object item) {
          this.item = item;
          this.next = null;
     }
     public Node(Object item, Node next) {
          this.item = item;
           this.next = next;
     }
     public void setNext(Node nextNode) {
          next = nextNode;
     }
     public Node getNext() {
          return next;
     }
     public Object getItem() {
          return item;
     }
     public void setItem(Object item){
          this.item = item;
     }
}
```

Node node = new Node (5);
Java will convert 5 to an instance of class `Integer`

# Printing a linked list

- Suppose we have a chain of nodes:


- And we want to print all the items.

# Printing a linked list

- Start at the **head** of the list.
- While (there are more nodes to print):
  - Print the current node's **item**.
  - Go to the **next** node.

- How do we walk through the nodes of the list?

```
head = head.getNext();    // is this a good idea?
```

# Printing a linked list

- To not lose the reference to this first node:

```
Node current = head;
```


- Move along a list by advancing a `Node` reference:

```
current = current.getNext();
```

# Printing a linked list

Code for printing the nodes of a list:

```
Node head = ...;

Node current = head;
while (current != null){
    System.out.println(current.getItem());
    current = current.getNext();
}
```

Similar to array code:

```
int[] a = ...;

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

# Printing a linked list

Same thing with a for loop

```
Node head = ...;

for (Node current = head; current != null; current =
current.getNext()){
    System.out.println(current.getItem());
}
```

the array version

```
int[] a = ...;

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

# Interim summary – why should I care?

- Linked list:
  - a self referential structure
- Advantage over arrays – no bound on capacity – can grow/shrink as needed (a dynamic structure)
- Linked lists are the basis for a lot of data structures
  - stacks, queues, trees
- The primary alternative to arrays

# The list interface

| Method | |
|---|---|
| `object get(index)` | Returns the element at the given position |
| `index indexOf(object)` | Returns the index of the first occurrence of the specified element |
| `add(object)` | Appends an element to the list |
| `add(index, object)` | inserts given value at given index, shifting subsequent values right |
| `object remove(index)` | Removes the element at the specified position (and returns it) |
| `object remove(object)` | Removes the element that corresponds to the given object (and returns it) |
| `int size()` | returns the size of the list |
| `boolean isEmpty()` | indicates if the list is empty |
| `clear()` | removes all elements from the list |

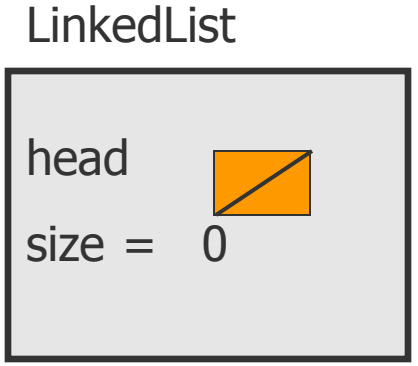
index is an int, and object is of type Object

# The list interface

```
public interface ListInterface {
   public boolean isEmpty();
   public int size();
   public void add(int index, Object item)
      throws ListIndexOutOfBounds;
   public void add(Object item);
   public void remove(int index)
      throws ListIndexOutOfBounds;
   public void remove(Object item);
   public Object get(int index)
      throws ListIndexOutOfBounds;
   public void clear();
}
```

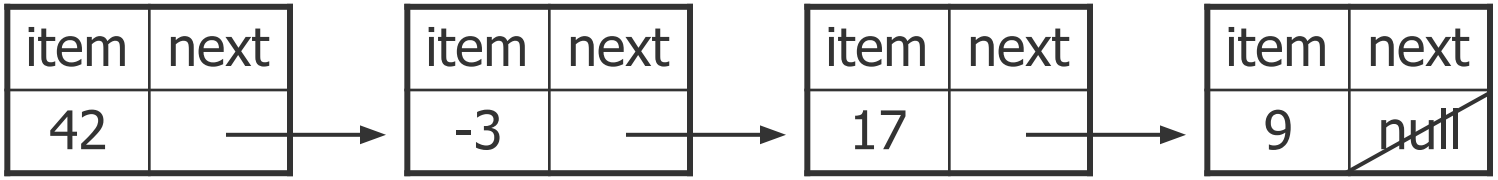
# Linked List: constructor

```
public class LinkedList {
    private Node head;
    private int size;

    public LinkedList() {
        head = null;
        size = 0;
    }
...

}
```



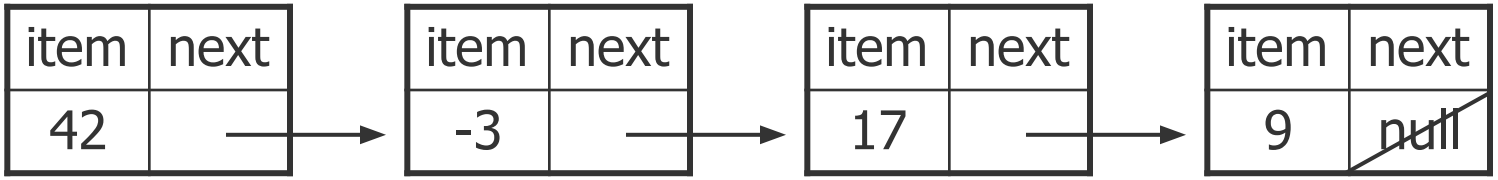
# Implementing add

- How do we add to a linked list at a given index?



# Implementing add

- How do we add to a linked list at a given index?
  - Did we consider all the possible cases?



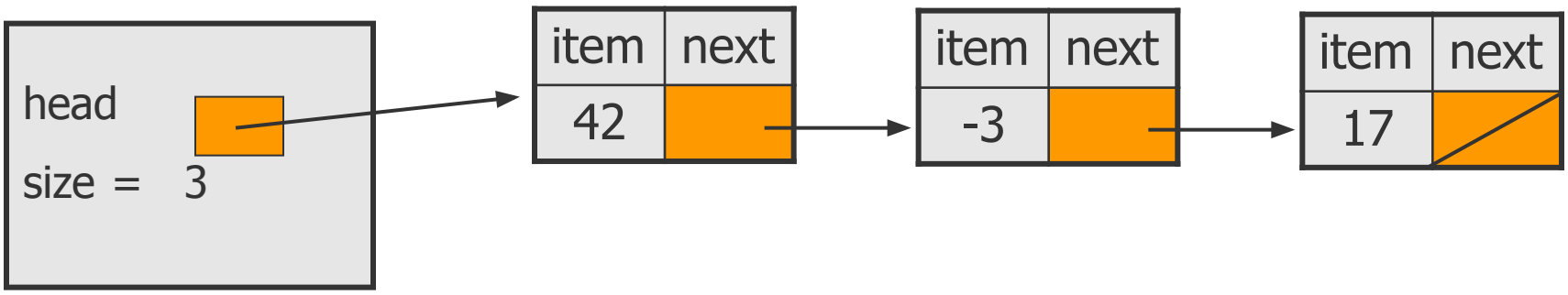
# The add method

```
public void add(int index, Object item){
    if (index<0 || index>size)
        throw new IndexOutOfBoundsException(”out of bounds”);
    if (index == 0) {
        head = new Node(item, head);
    }
    else {   // find predecessor of node
        Node curr = head;
        for (int i=0; i<index-1; i++){
            curr = curr.getNext();
        }
        curr.setNext(new Node(item, curr.getNext()));
    }
    size++;
}
```

# Implementing `remove`

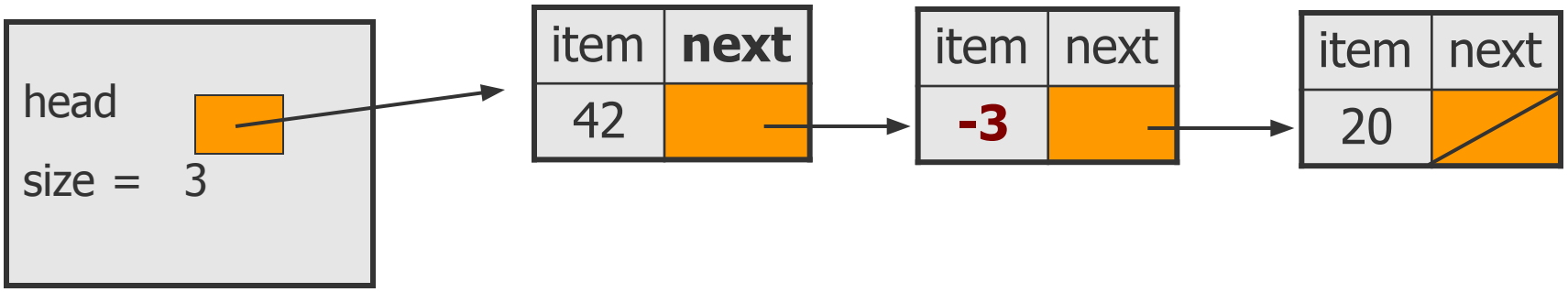
```
// Removes value at a given index
public void remove(int index) {
    ...
}
```

- How do we remove a node?

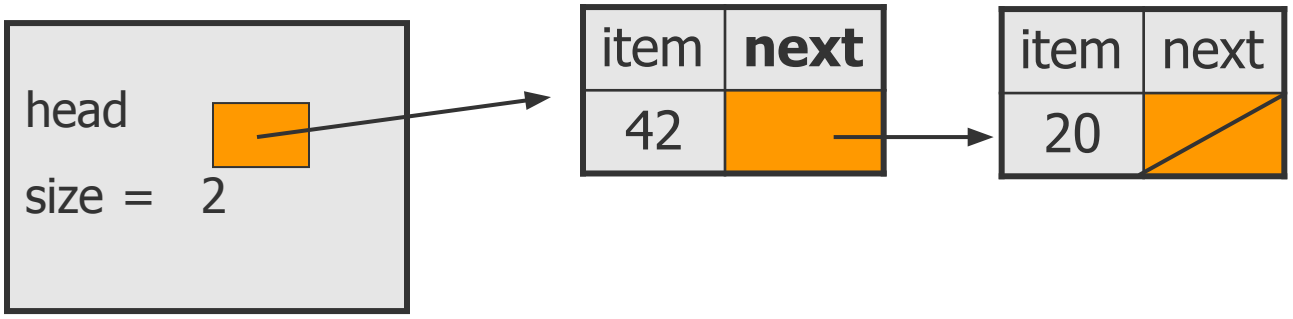


# Removing a node from a list

- Before removing element at index 1:

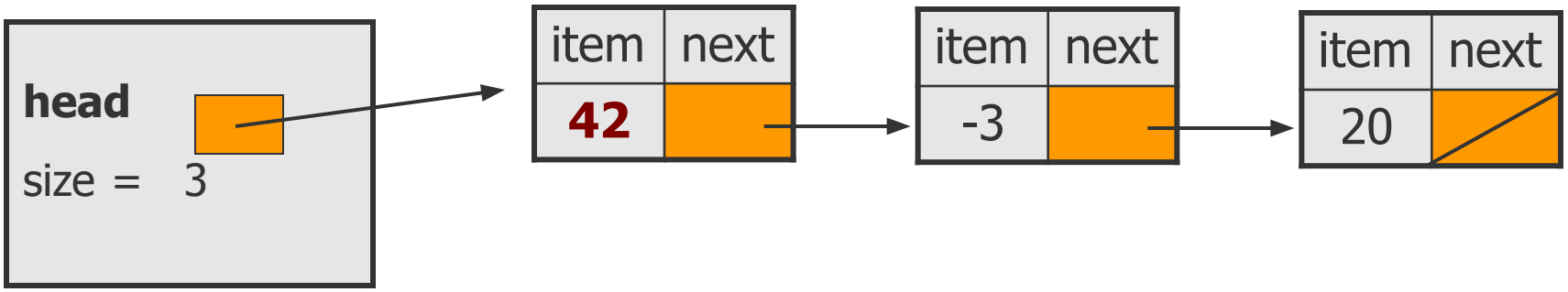



- After:

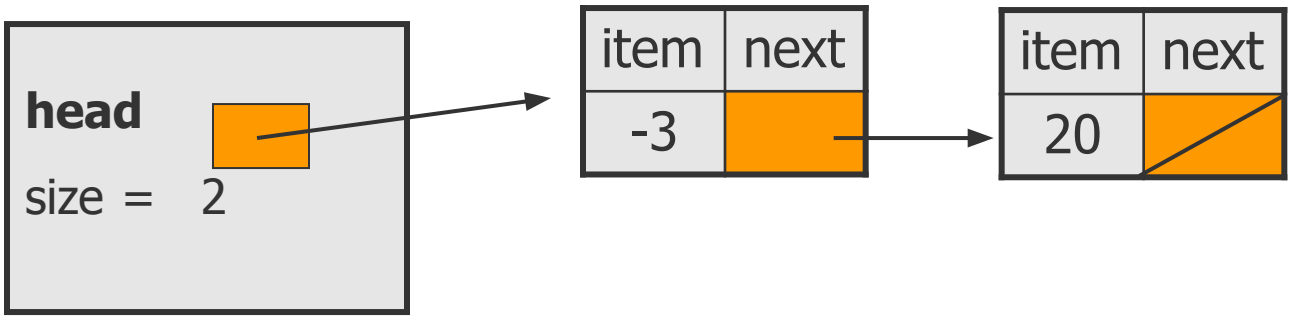


# Removing the first node from a list

- Before removing element at index 0:

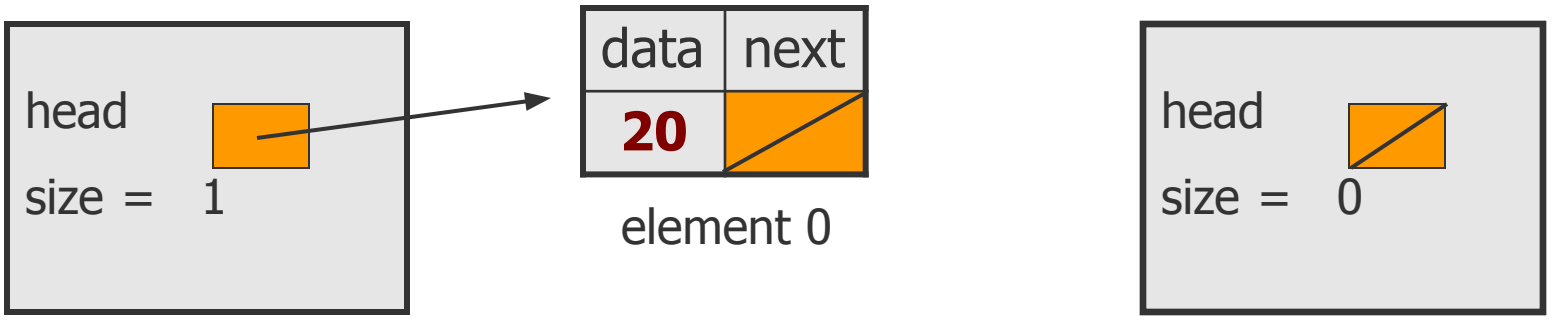



- After:



# List with a single element

- Before: After:




  - We must change head to `null`.
  - Do we need a special case to handle this?

# The remove method

```
public void remove(int index) {
    if (index<0 || index >= size)
        throw new IndexOutOfBoundsException
            ("List index out of bounds");
    if (index == 0) {
        // special case: removing first element
        head = head.getNext();
    } else {
        // removing from elsewhere in the list
        Node current = head;
        for (int i = 0; i < index - 1; i++) {
            current = current.getNext();
        }
        current.setNext(current.getNext().getNext());
    }
    size--;
}
```

# The clear method

- How do you implement a method for removing all the elements from a linked list?

# The clear method

```
public void clear() {
  head = null;
}
```

- Where did all the memory go?
- Java's garbage collection mechanism takes care of it!
- An object is elligible for garbage collection when it is no longer accessible (cyclical references don't count!)

- In C/C++ the programmer needs to release unused memory explicitly

# Linked lists recursively

- We would like to print the elements in a linked list recursively.
  - What would be the signature of the method?
  - Base case?
  - Recursive case?

# Recursive linked list traversal – which is correct?

a
```
private void writeList(Node node) {
    if (node != null) {
        System.out.println(node.getItem());
        writeList(node.getNext());
    }
}
```

b
```
private void writeList(Node node) {
    if (node != null) {
        writeList(node.getNext());
        System.out.println(node.getItem());
    }
}
```

# Recursive linked list traversal

```
private void writeList(Node node) {
    //precondition:  linked list is referenced by node
    //postcondition: list is displayed. list is
   unchanged
    if (node != null) {
       // write the first item
       System.out.println(node.getItem());
       // write the rest of the list
       writeList(node.getNext());
    }
}
```

# Recursive backward traversal

- We have two ways for recursively traversing a string backwards:
  - Write the last character of the string s
  - Write string s minus its last character backward

  And
  - Write string s minus its first character backward
  - Write the first character of string s

# Recursive backward traversal

- Translated to our problem:
  - write the last node of the list
  - write the list minus its last node backward

  And
  - write the list minus its first node backward
  - write the first node of the list

  Which of these strategies is better for linked lists?

# Recursive backward traversal

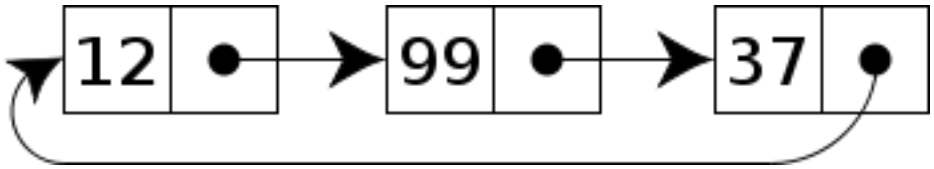
```
private void writeListBackward (Node node) {
    //precondition:  linked list is referenced by node
    //postcondition: list is displayed. list is
   unchanged
    if (node != null) {
       // write the rest of the list
       writeListBackward(node.getNext());
       // write the first item
       System.out.println(node.getItem());
    }
}
```

# Recursive add method

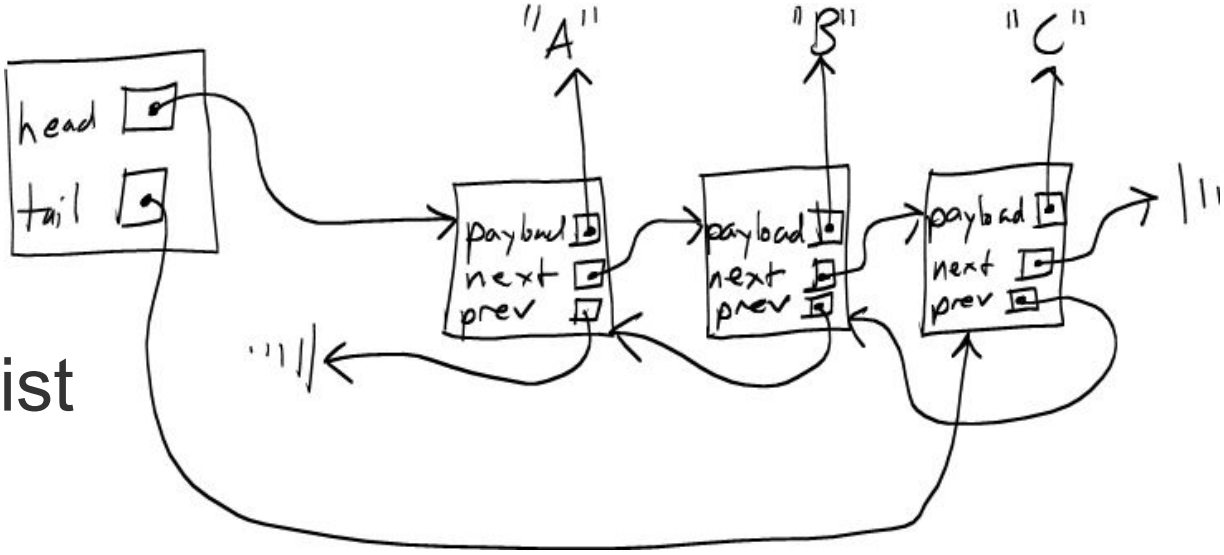
```
public void add(Object item) {
    head = addRecursive(head, item);
}
private Node addRecursive(Node node, Object item) {
    if (node == null) {
        node = new Node(item, node);
    }
    else {// insert into the rest of the linked list
        node.setNext(addRecursive(
                        node.getNext(), item));
    }
    return node;
}
```

# Variations

- Circular linked list




- Doubly linked list




- What are the advantages and disadvantages of a doubly linked list?

image from: http://en.wikipedia.org/wiki/Linked_list