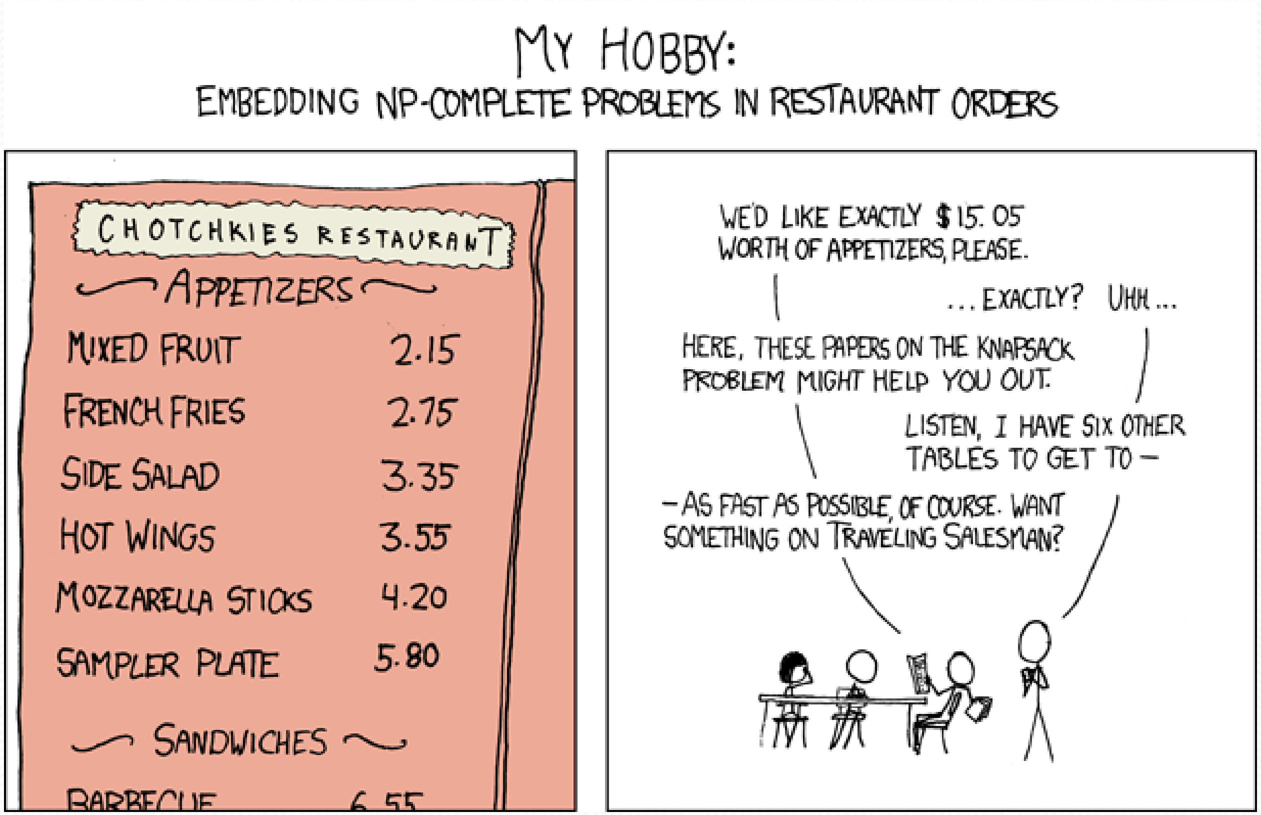
CSE 143

Lecture 6: References and linked nodes

reading: 16.1


MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS
CHOTCHKIES RESTAURANT
APPETIZERS
MIXED FRUIT 2.15
FRENCH FRIES 2.75
SIDE SALAD 3.35
HOT WINGS 3.55
MOZZARELLA STICKS 4.20
SAMPLER PLATE 5.80
SANDWICHES
BARBECUE 6.55
WE'D LIKE EXACTLY $15.05 WORTH OF APPETIZERS, PLEASE.
...EXACTLY? UHH...
HERE, THESE PAPERS ON THE KNAPSACK PROBLEM MIGHT HELP YOU OUT.
LISTEN, I HAVE SIX OTHER TABLES TO GET TO –
– AS FAST AS POSSIBLE, OF COURSE. WANT SOMETHING ON TRAVELING SALESMAN?

- NP-complete is a complexity class
  - No known polynomial time ($O(n)$, $O(n^5)$…) solutions!
  - Solutions are, for example, $O(2^n)$ – ouch!

Collection efficiency

- Complexity class of various operations on collections:

Method	ArrayList	Stack	Queue
`add` (or `push`)	O(1)	O(1)	O(1)
`add(`**index, value**`)`	O(N)	-	-
`indexOf`	O(N)	-	-
`get`	O(1)	-	-
`remove`	O(N)	O(1)	O(1)
`set`	O(1)	-	-
`size`	O(1)	O(1)	O(1)

- Could we build lists differently to optimize other operations?

Non-contiguous memory

- Array

42	-3	17	9

- Spread in memory

42			9		-3			17

Value semantics

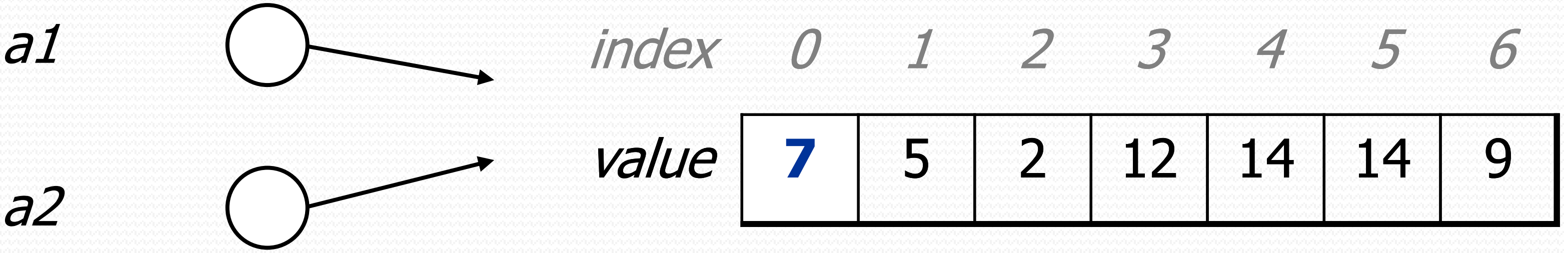
- **value semantics**: Behavior where values are copied when assigned to each other or passed as parameters.
  - When one primitive is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;         // x = 5, y = 5
y = 17;            // x = 5, y = 17
x = 8;             // x = 8, y = 17
```

Reference semantics

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - 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
```

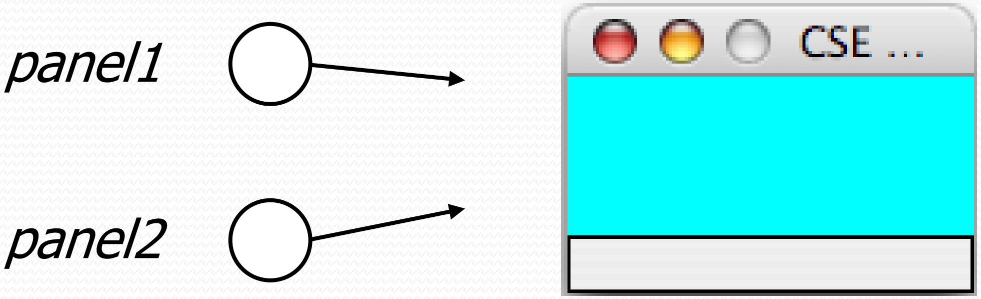


index	*0*	*1*	*2*	*3*	*4*	*5*	*6*
value	**7**	5	2	12	14	14	9

References and objects

- In Java, objects and arrays use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

```
DrawingPanel panel1 = new DrawingPanel(80, 50);
DrawingPanel panel2 = panel1;    // same window
panel2.setBackground(Color.CYAN);
```

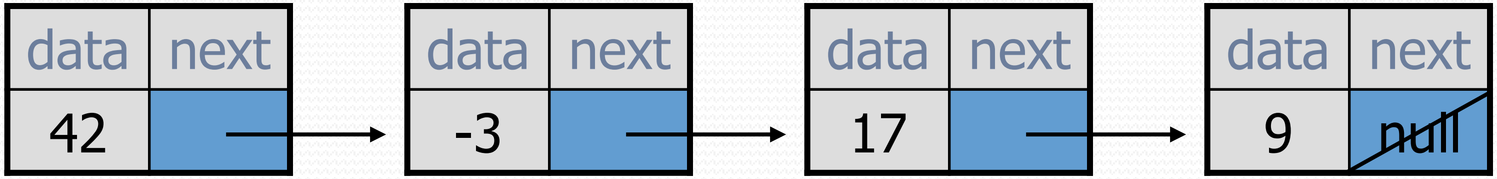


A list node class

```
public class ListNode {
    int data;
    ListNode next;
}
```

- Each list node object stores:
  - one piece of integer data
  - a reference to another list node

- `ListNode`s can be "linked" into chains to store a list of values:

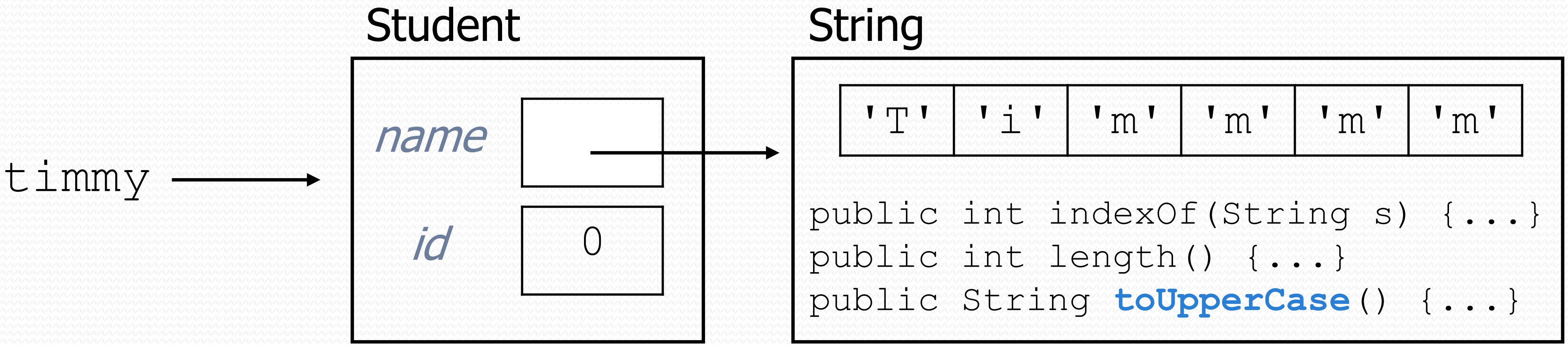


data next
42
data next
-3
data next
17
data next
9 null

Dereferencing

- **dereference**: To access data or methods of an object.
  - Done with the dot notation, such as `s.length()`
  - When you use a `.` after an object variable, Java goes to the memory for that object and looks up the field/method requested.

```
Student timmy = new Student();
timmy.name = "Timmmm";
String s = timmy.name.toUpperCase();
```



timmy

Student

name

id 0

String

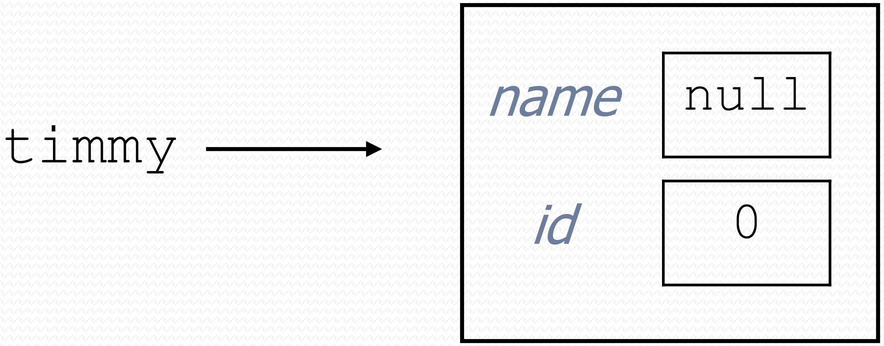
'T'	'i'	'm'	'm'	'm'	'm'

```
public int indexOf(String s) {...}
public int length() {...}
public String toUpperCase() {...}
```

Null pointer exception

- It is illegal to dereference `null` (it causes an exception).
  - `null` does not refer to any object; it has no methods or data.

```
Student timmy = new Student();
String s = timmy.name.toUpperCase();    // ERROR
```

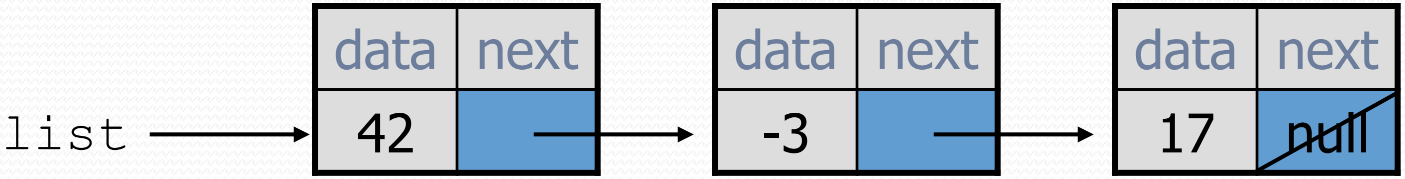


Output:

```
Exception in thread "main"
java.lang.NullPointerException
        at Example.main(Example.java:8)
```

List node client example

```
public class ConstructList1 {
    public static void main(String[] args) {
        ListNode list = new ListNode();
        list.data = 42;
        list.next = new ListNode();
        list.next.data = -3;
        list.next.next = new ListNode();
        list.next.next.data = 17;
        list.next.next.next = null;
        System.out.println(list.data + " " + list.next.data
                           + " " + list.next.next.data);
        // 42 -3 17
    }
}
```



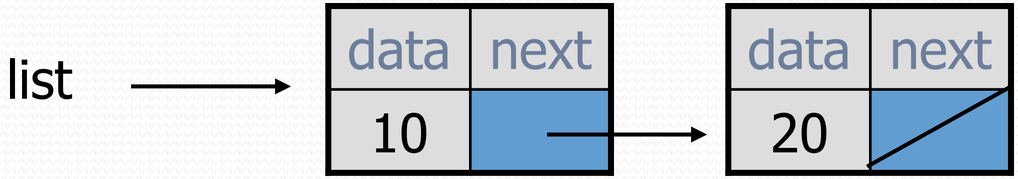
List node w/ constructor

```
public class ListNode {
    int data;
    ListNode next;

    public ListNode(int data) {
        this.data = data;
        this.next = null;
    }

    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

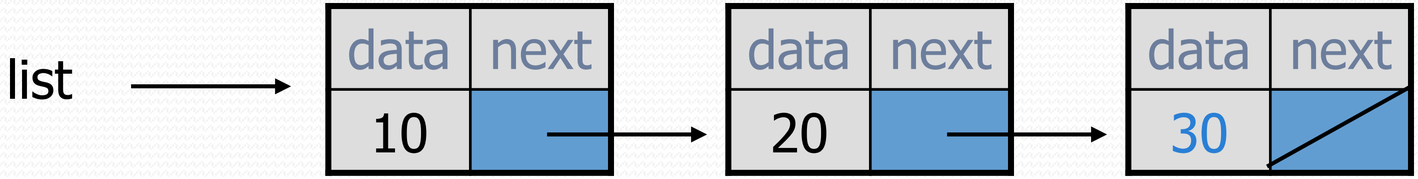
- Exercise: Modify the previous client to use these constructors.

Linked node problem 1

- What set of statements turns this picture:



- Into this?



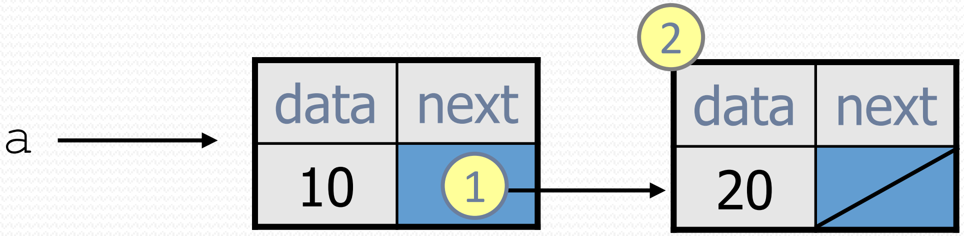
References vs. objects

variable = **value**;

a *variable* (left side of =) is an arrow (the base of an arrow)

a *value* (right side of =) is an object (a box; what an arrow points at)

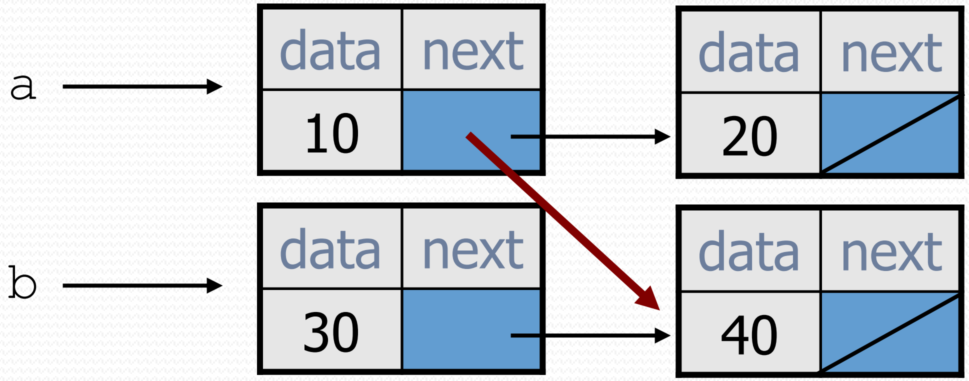
- For the list at right:
  - `a.next` = **value**;
    means to adjust where ① points
  - **variable** = `a.next;`
    means to make **variable** point at ②



Reassigning references

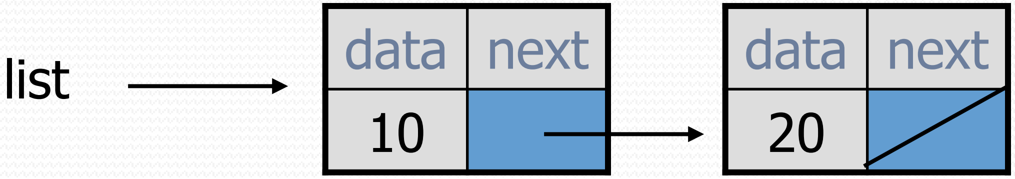
- when you say:
  - `a.next = b.next;`

- you are saying:
  - "Make *variable* `a.next` refer to the same *value* as `b.next`."
  - Or, "Make `a.next` point to the same place that `b.next` points."

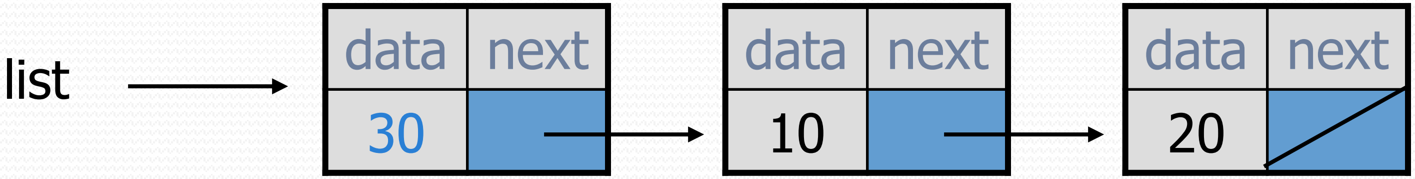


Linked node problem 2

- What set of statements turns this picture:

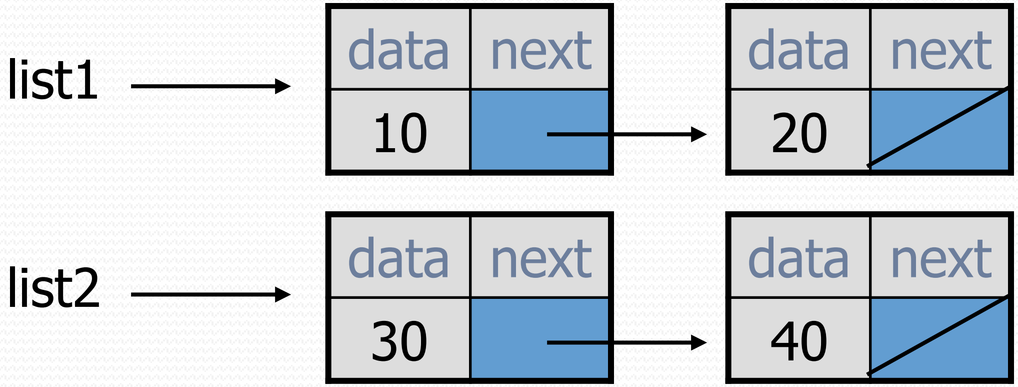


- Into this?

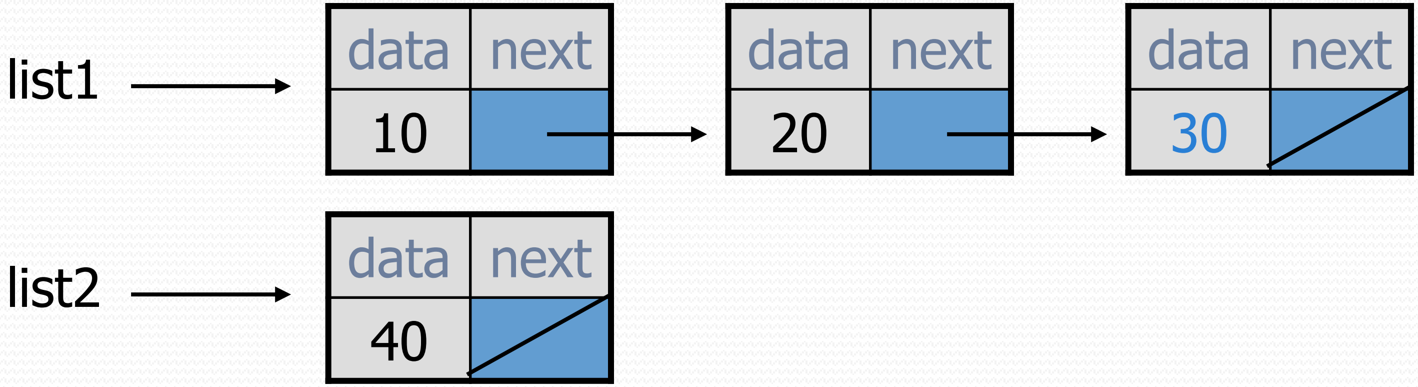


Linked node problem 3

- What set of statements turns this picture:

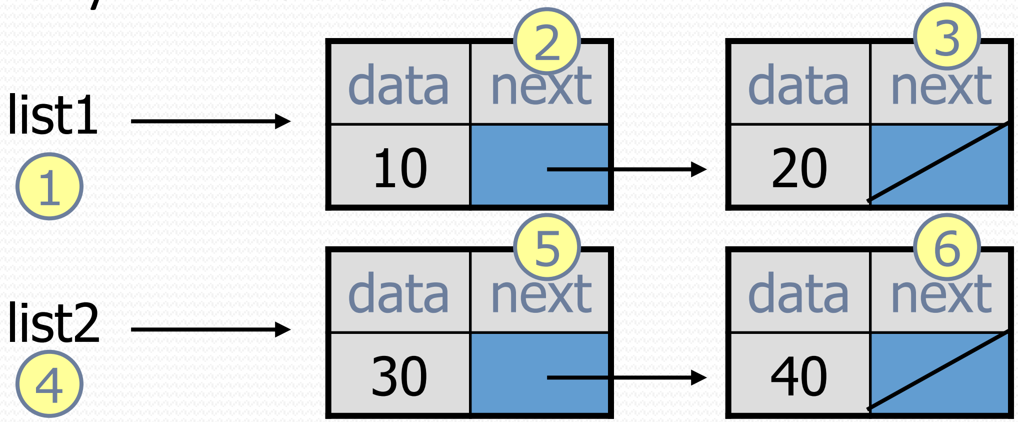


- Into this?

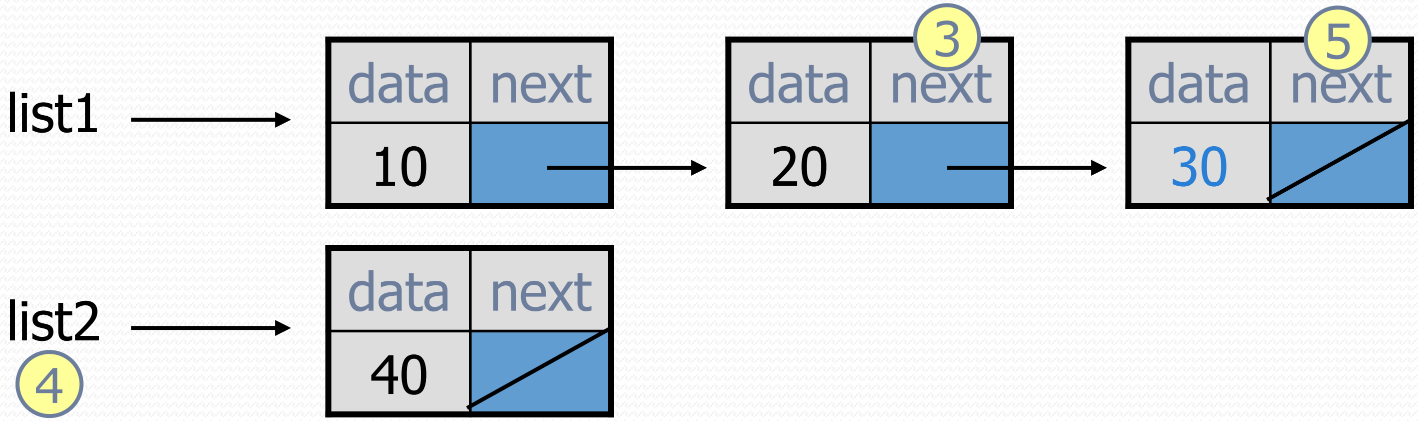


Linked node problem 3

- How many ListNode variables?

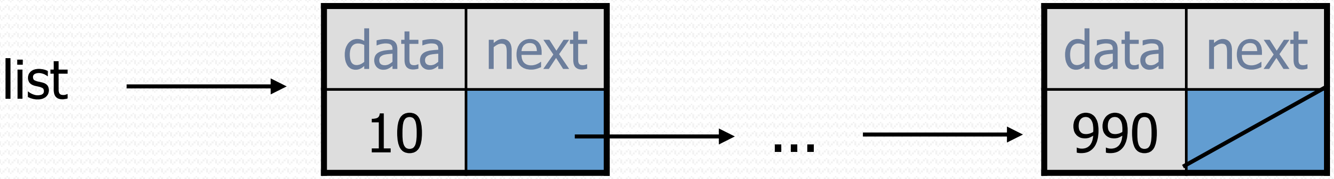


- Which variables change?

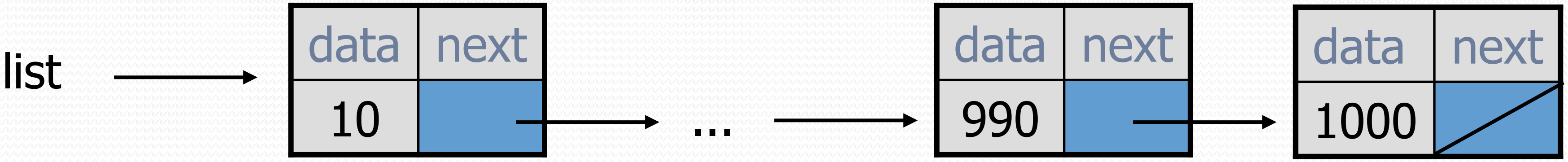


Linked node problem 4

- What set of statements turns this picture:



- Into this?


list
data
next
10
...
data
next
990
data
next
1000