

SD2x1.1

Motivating the need for data structures

Chris

**Let's say we want to keep
track of some numbers.**

What are some operations we might like to perform?

Keeping track of numbers: operations

- Add another number to the collection
 - Possibly at a certain location
- Remove a number from the collection
 - Possibly from a certain location
- Determine whether the collection contains a given number
- Retrieve a number at a given location
- Retrieve the numbers in sorted order
- Retrieve the numbers in the order in which we added them to the collection

nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30

```
int nums[ ] = new int[4];  
nums[0] = 28;  
nums[1] = 14;  
nums[2] = 17;  
nums[3] = 30;
```

```
int nums[ ] = { 28, 14, 17, 30 };
```

What's wrong with using an array?

Limitation #1:

Need to know the size in advance

nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30

```
int nums[ ] = new int[4];  
nums[0] = 28;  
nums[1] = 14;  
nums[2] = 17;  
nums[3] = 30;  
  
. . .
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]
28	14	17	30	X

```
int nums[ ] = new int[4];  
nums[0] = 28;  
nums[1] = 14;  
nums[2] = 17;  
nums[3] = 30;  
  
.  
.  
. .  
  
nums[4] = 16;
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	?	?	?	?

```
int nums[ ] = new int[8];  
nums[0] = 28;  
nums[1] = 14;  
nums[2] = 17;  
nums[3] = 30;
```

Limitation #2:

Cannot easily insert an element in front of others

count = 4							
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	?	?	?	?

```
int nums[ ] = new int[8];  
nums[0] = 28;  
nums[1] = 14;  
nums[2] = 17;  
nums[3] = 30;
```

```
int count = 4; // number of elements in array
```

```
...
```

count = 4

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	?	?	?

```
int nums[ ] = new int[8];
nums[0] = 28;
nums[1] = 14;
nums[2] = 17;
nums[3] = 30;

int count = 4; // number of elements in array

. . .

nums[count] = 16; // adds to end
```

count = 5

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	?	?	?

```
int nums[ ] = new int[8];
nums[0] = 28;
nums[1] = 14;
nums[2] = 17;
nums[3] = 30;

int count = 4; // number of elements in array

. . .

nums[count] = 16; // adds to end
count++;
```

count = 5

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	41	?	?

```
int nums[ ] = new int[8];
nums[0] = 28;
nums[1] = 14;
nums[2] = 17;
nums[3] = 30;

int count = 4; // number of elements in array

. . .

nums[count] = 16; // adds to end
count++;
nums[count] = 41;
```

count = 6

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	41	?	?

```
int nums[ ] = new int[8];
nums[0] = 28;
nums[1] = 14;
nums[2] = 17;
nums[3] = 30;

int count = 4; // number of elements in array

. . .

nums[count] = 16; // adds to end
count++;
nums[count] = 41;
count++;
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	41	?	?

A light blue arrow points upwards from a blue box containing the number 22 towards the first row of the table.

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	41	41	?

22

nums[6] = nums[5];

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	16	16	41	?

22

↑

```
nums[6] = nums[5];  
nums[5] = nums[4];
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	30	30	16	41	?

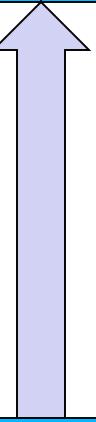
22

↑

```
nums[6] = nums[5];  
nums[5] = nums[4];  
nums[4] = nums[3];
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	17	17	30	16	41	?

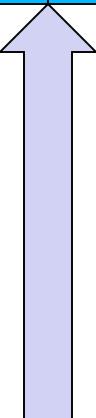
22



```
nums[6] = nums[5];  
nums[5] = nums[4];  
nums[4] = nums[3];  
nums[3] = nums[2];
```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	14	14	17	30	16	41	?

22

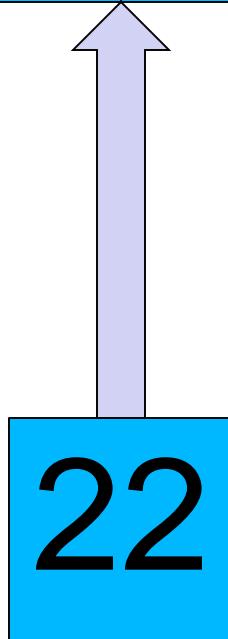


```

nums[6] = nums[5];
nums[5] = nums[4];
nums[4] = nums[3];
nums[3] = nums[2];
nums[2] = nums[1];

```

nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	?



```

nums[6] = nums[5];
nums[5] = nums[4];
nums[4] = nums[3];
nums[3] = nums[2];
nums[2] = nums[1];
nums[1] = 22;

```

Limitation #3:

Some operations can be very slow

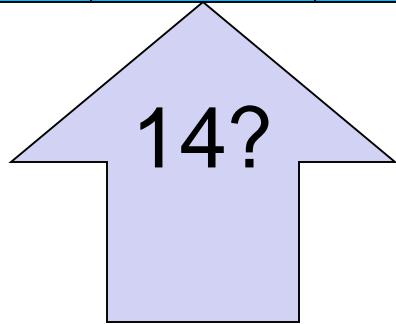
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11

14

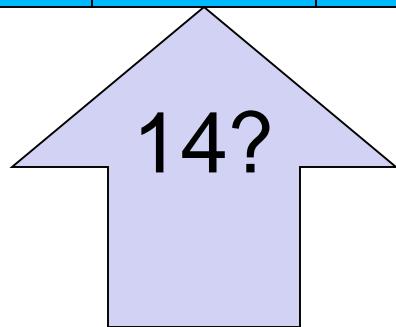
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11

14?

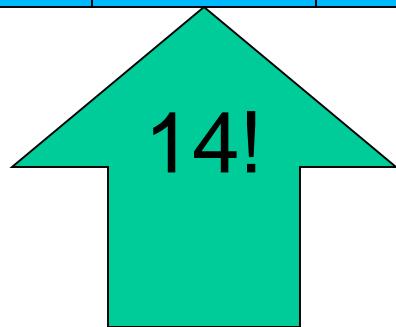
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



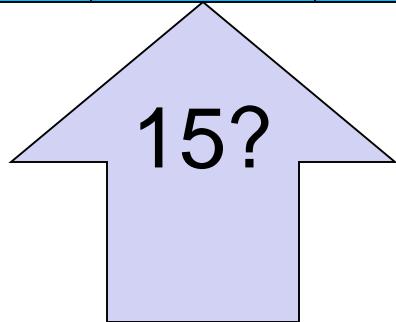
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11

15

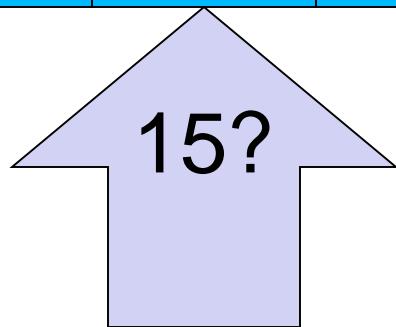
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11

15?

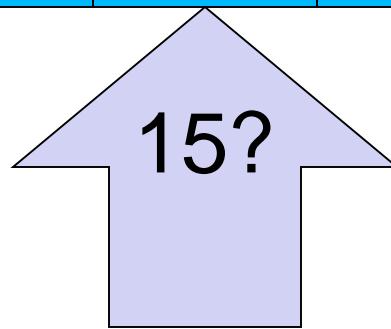
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



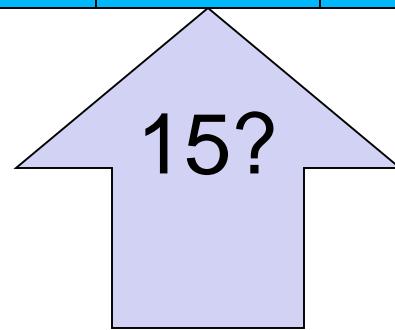
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



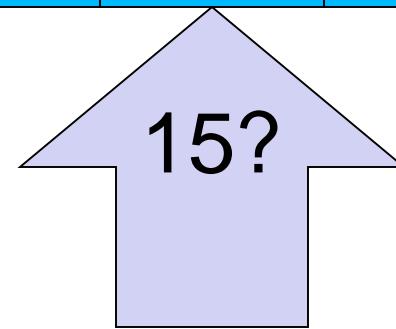
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



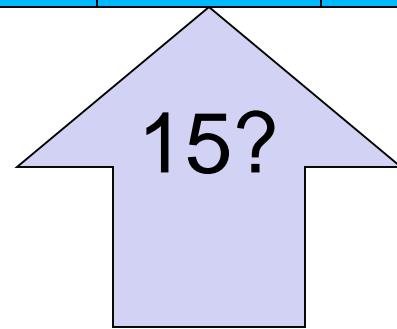
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



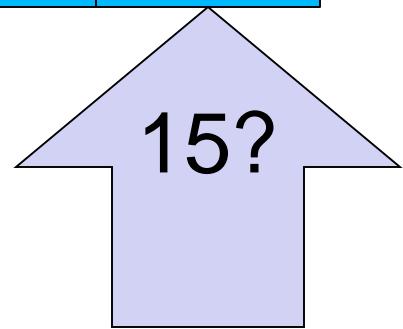
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



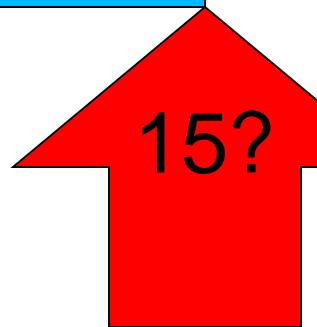
nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11



nums[0]	nums[1]	nums[2]	nums[3]	nums[4]	nums[5]	nums[6]	nums[7]
28	22	14	17	30	16	41	11

```
public boolean contains (int[] nums, int target) {  
    for (int i = 0; i < nums.length; i++) {  
        if (nums[i] == target) {  
            return true; // found it!  
        }  
    }  
    return false; // couldn't find it  
}
```

Limitations of Arrays

- Need to know the number of elements when the array is created
 - Too small: cannot add more elements
 - Too large: wasted space
- Cannot easily insert an element (or remove element) except at end
 - Need to shift all elements, assuming enough space!
- Some operations can be very slow

SD2x1.2

LL: structure, add

Kathy

Using arrays to keep track of elements

- Need to know the number of elements when the array is created
 - Too small: cannot add more elements
 - Too large: wasted space
- Cannot easily insert an element in front of others
 - Need to shift all elements, assuming enough space!

nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30

nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30

28

14

17

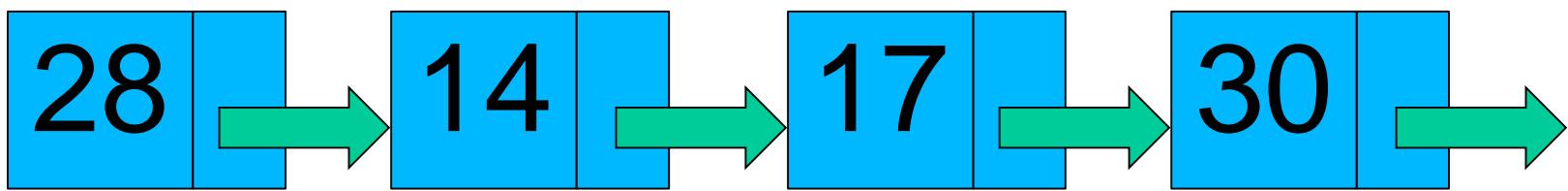
30

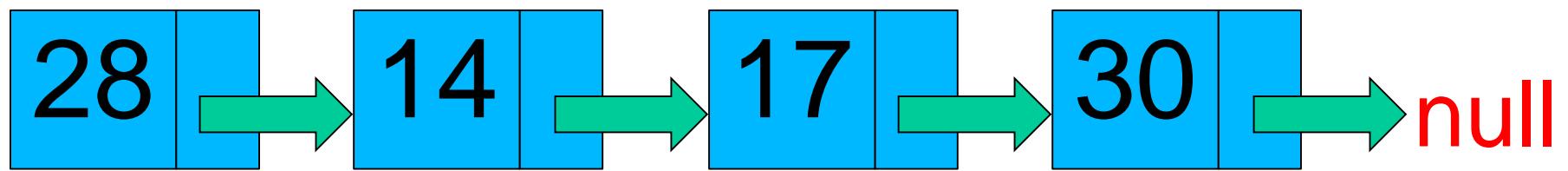
28

14

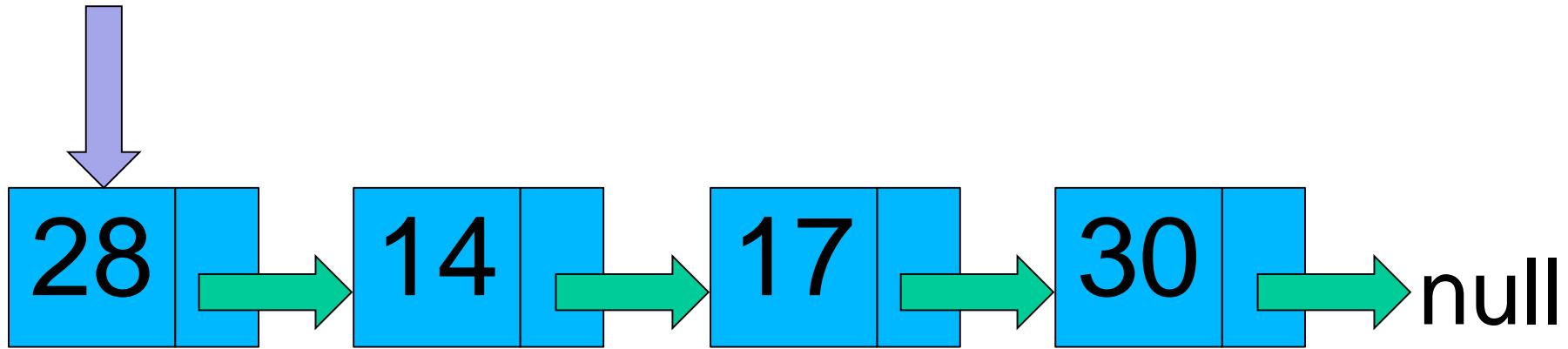
17

30

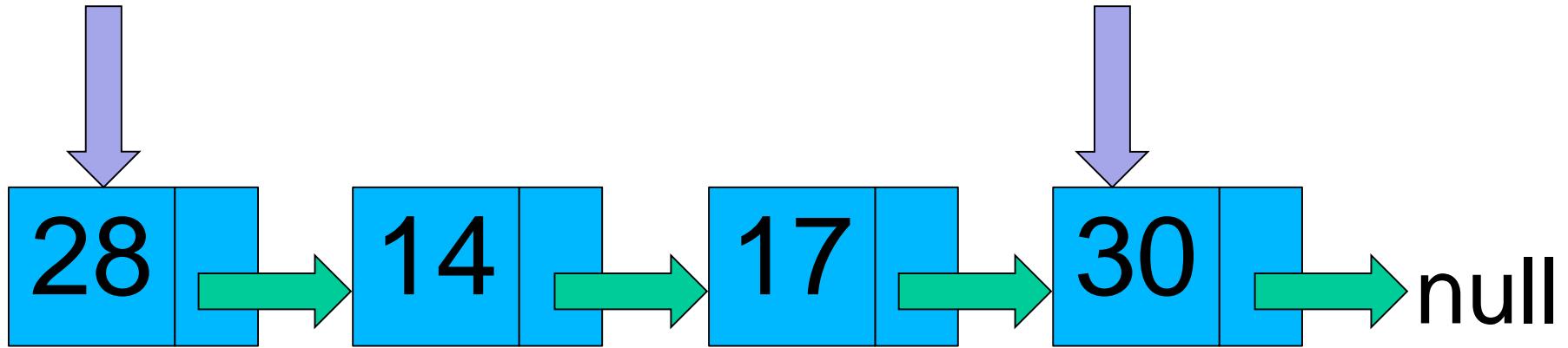




head

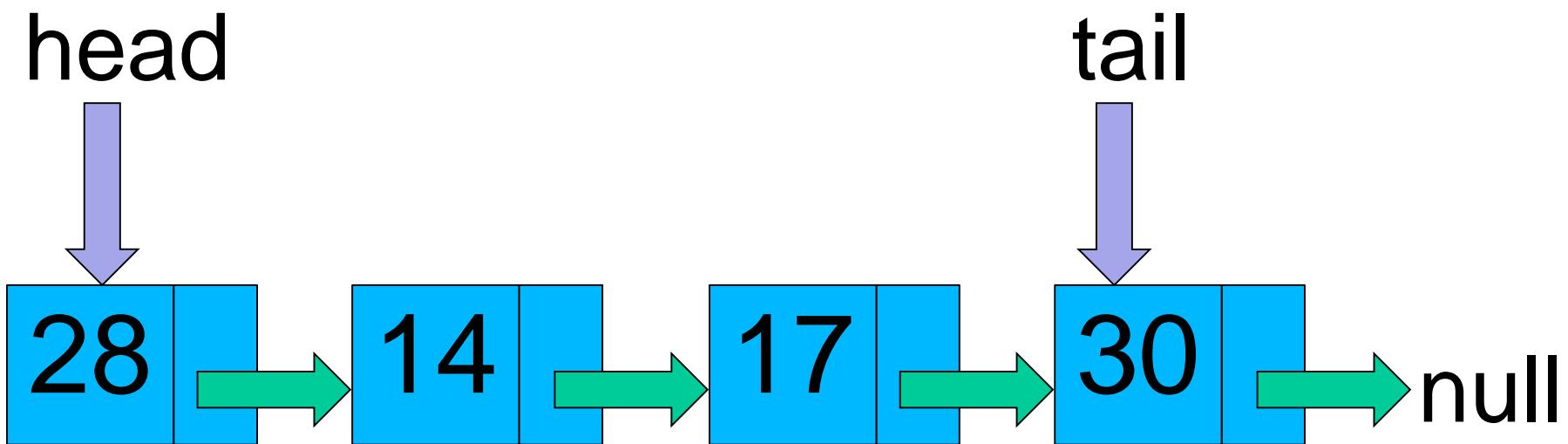


head



tail

Linked List!



LinkedList class definition

```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

LinkedList class definition

```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

LinkedList class definition

```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

LinkedList class definition

```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

LinkedList class definition

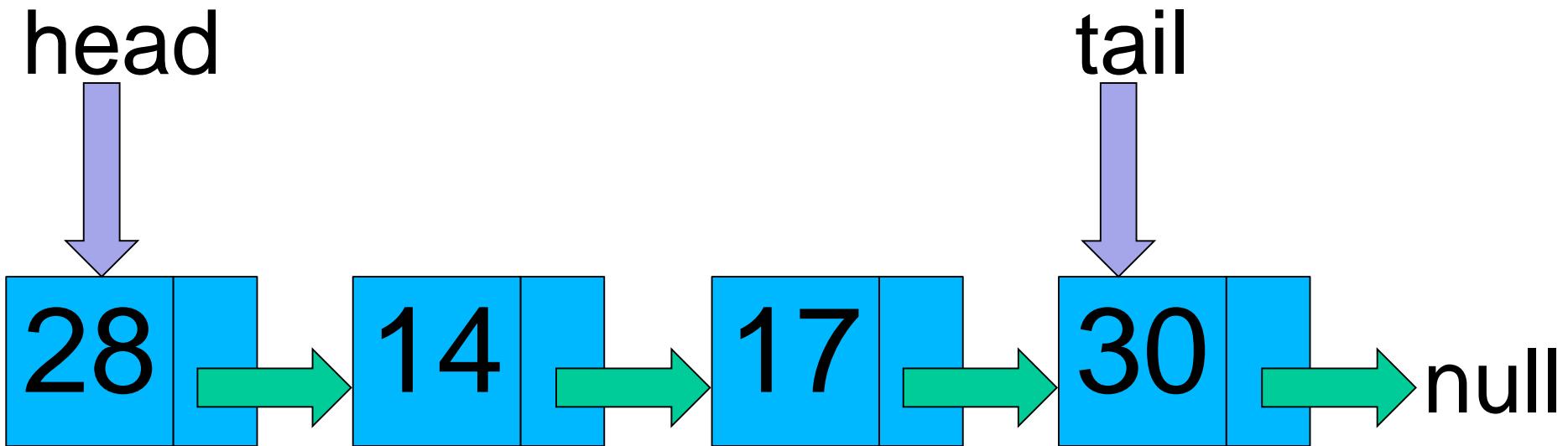
```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

LinkedList class definition

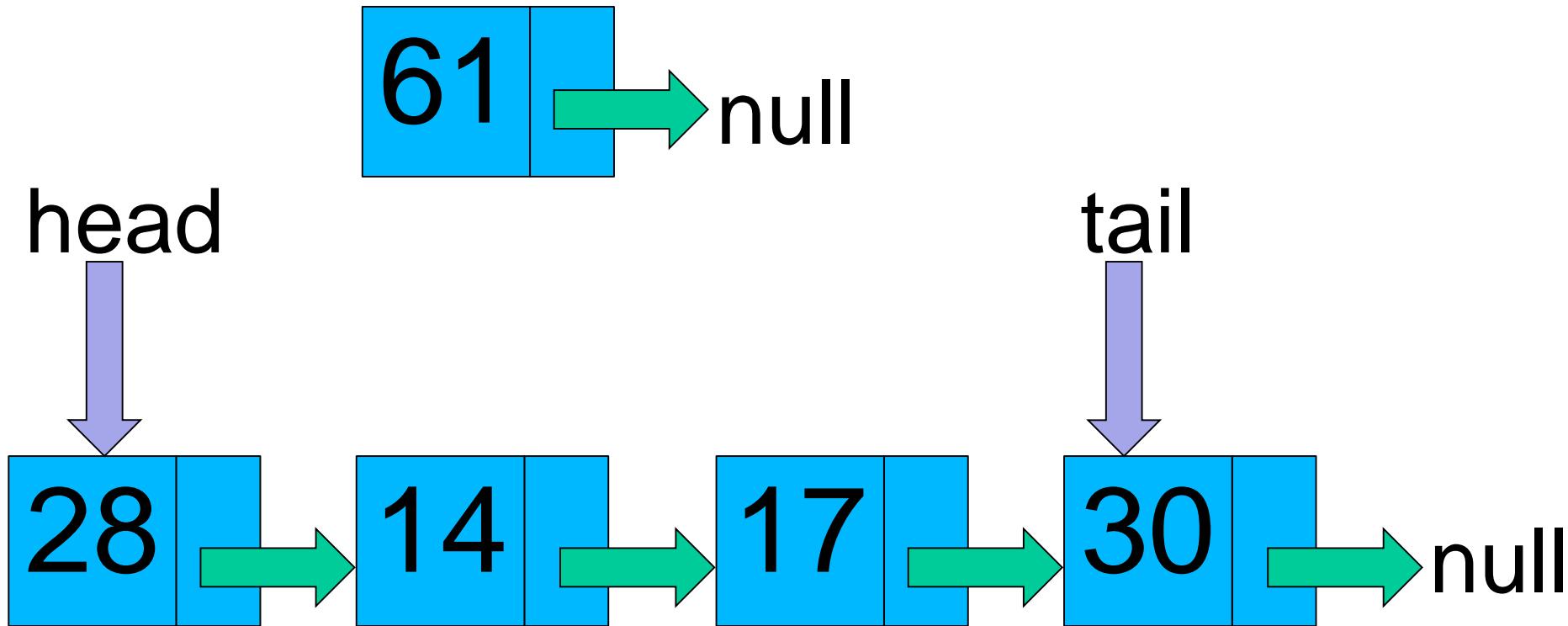
```
public class LinkedList {  
  
    class Node {  
        int value;  
        Node next = null;  
        Node(int value) {  
            this.value = value;  
        }  
    }  
  
    protected Node head = null;  
    protected Node tail = null;  
  
    . . .
```

How do we add a value to the Linked List?

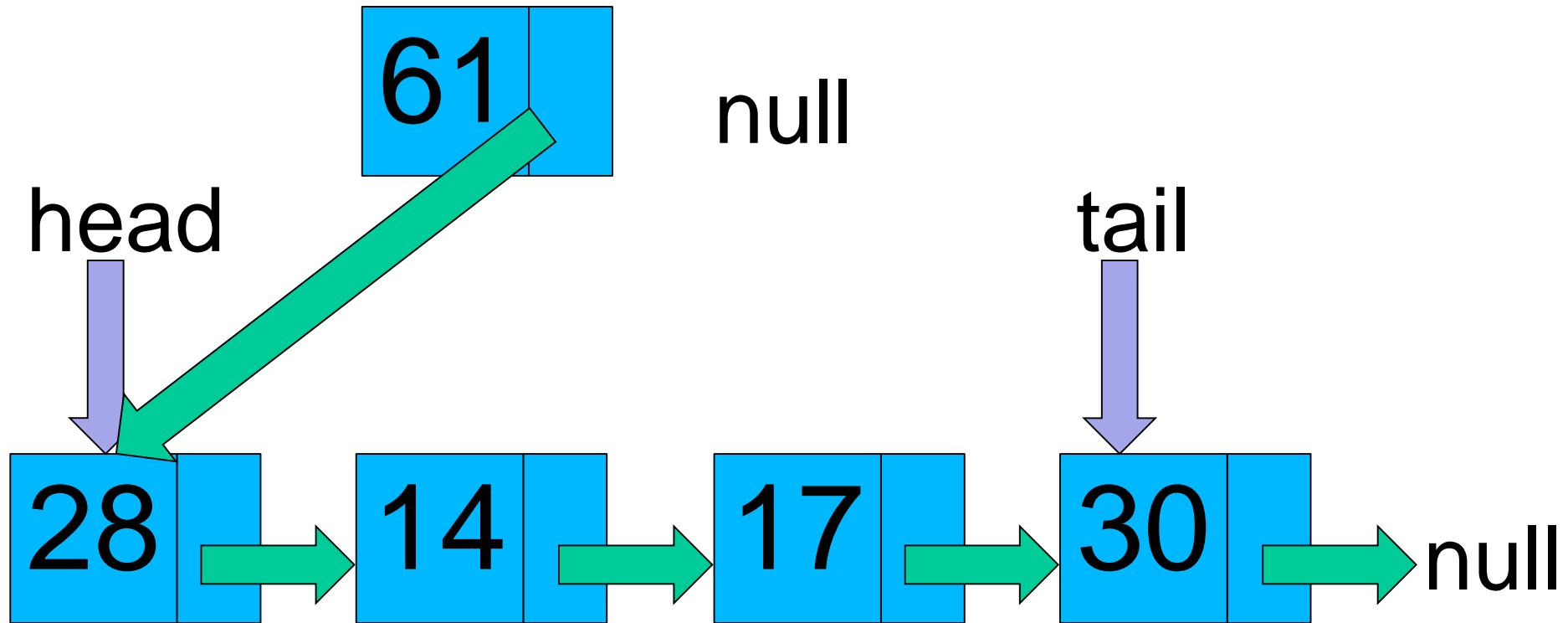
Adding to the front



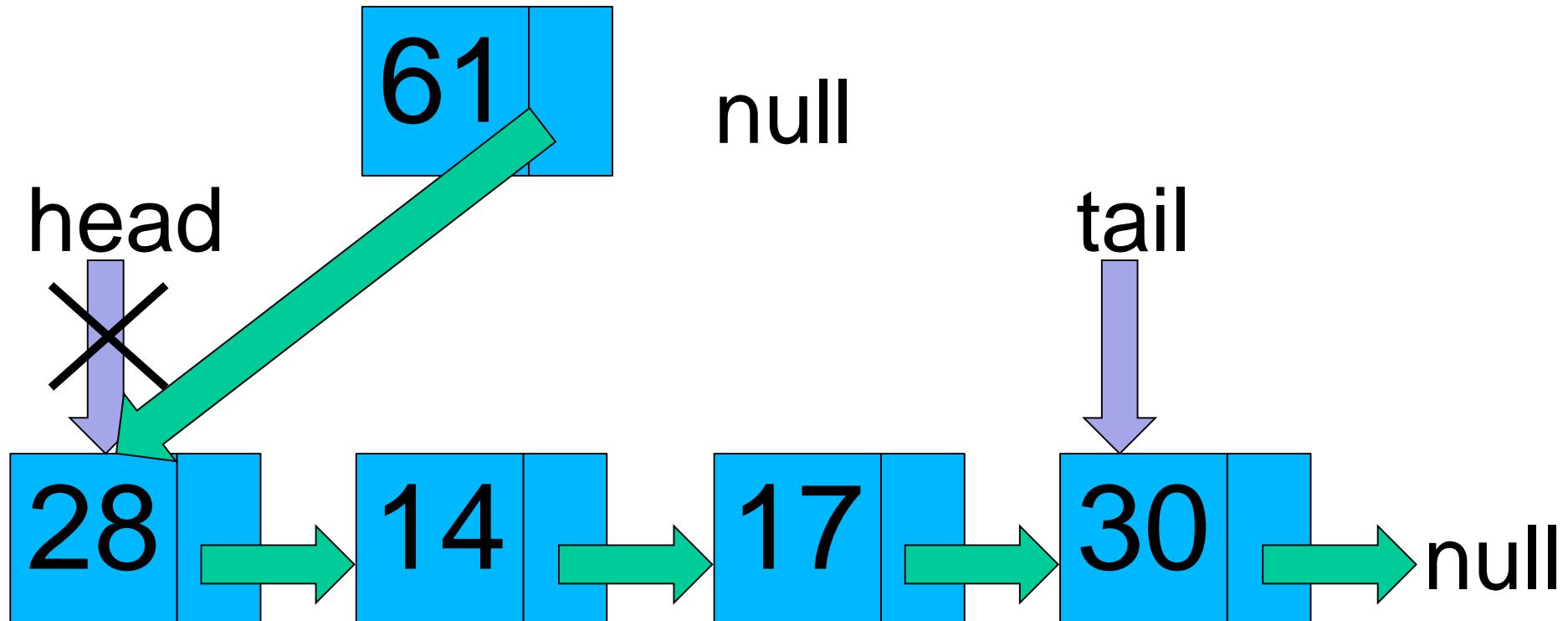
Adding to the front



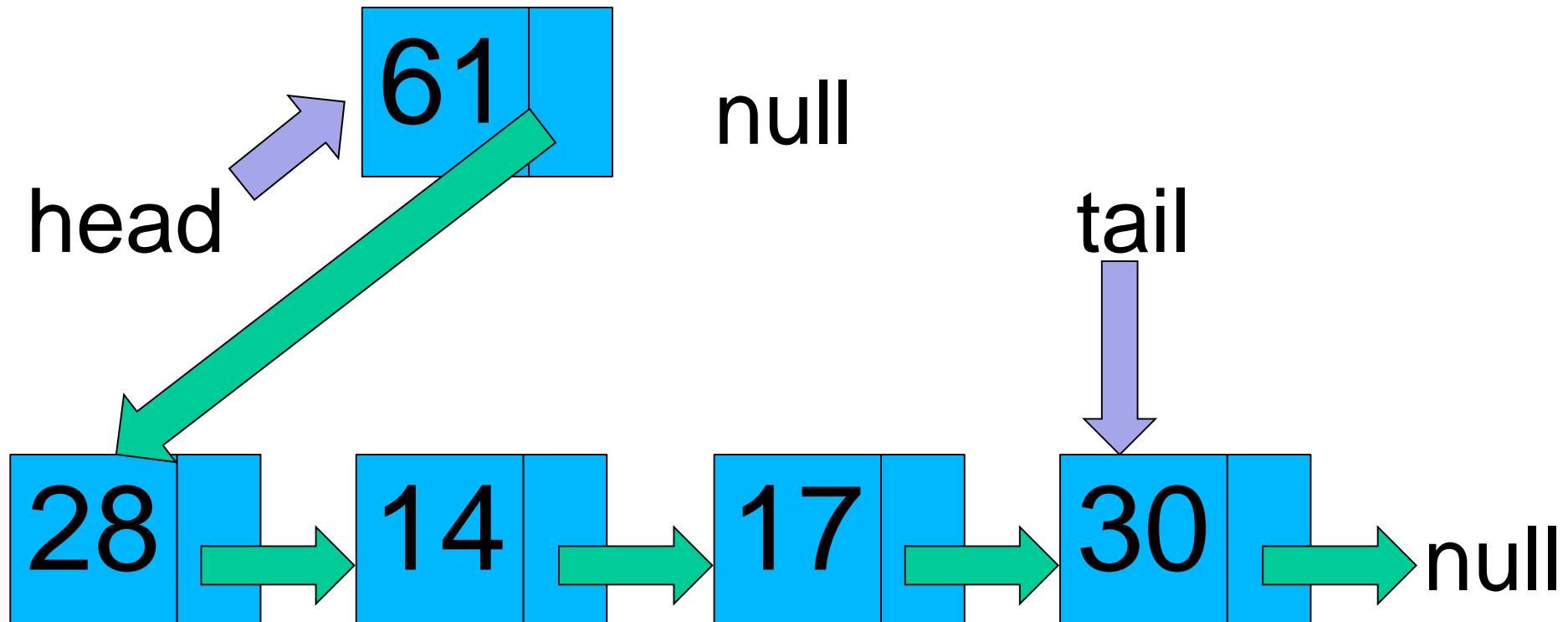
Adding to the front



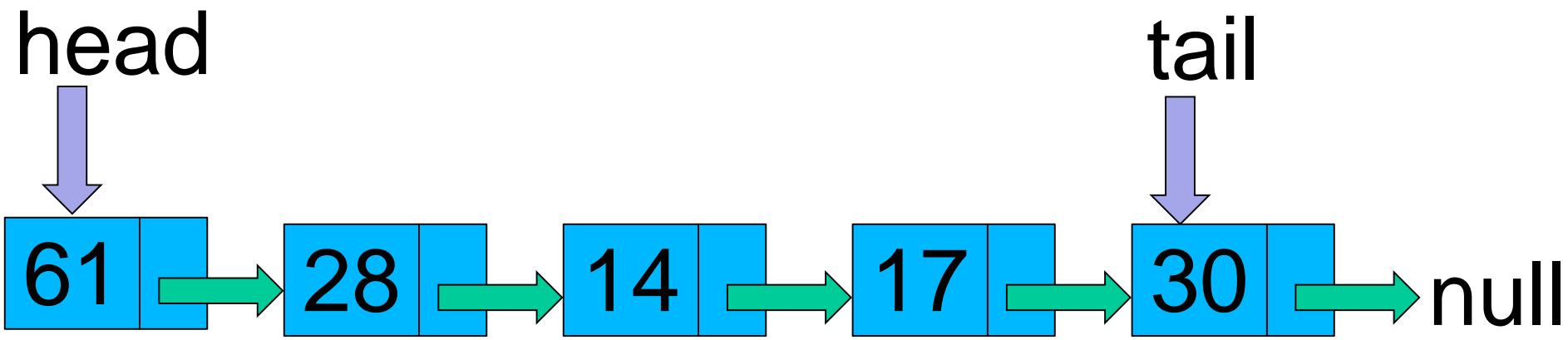
Adding to the front



Adding to the front



Adding to the front



LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
public void addToFront(int value) {  
    Node newNode = new Node(value);  
    newNode.next = head;  
    head = newNode;  
}  
}
```

LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

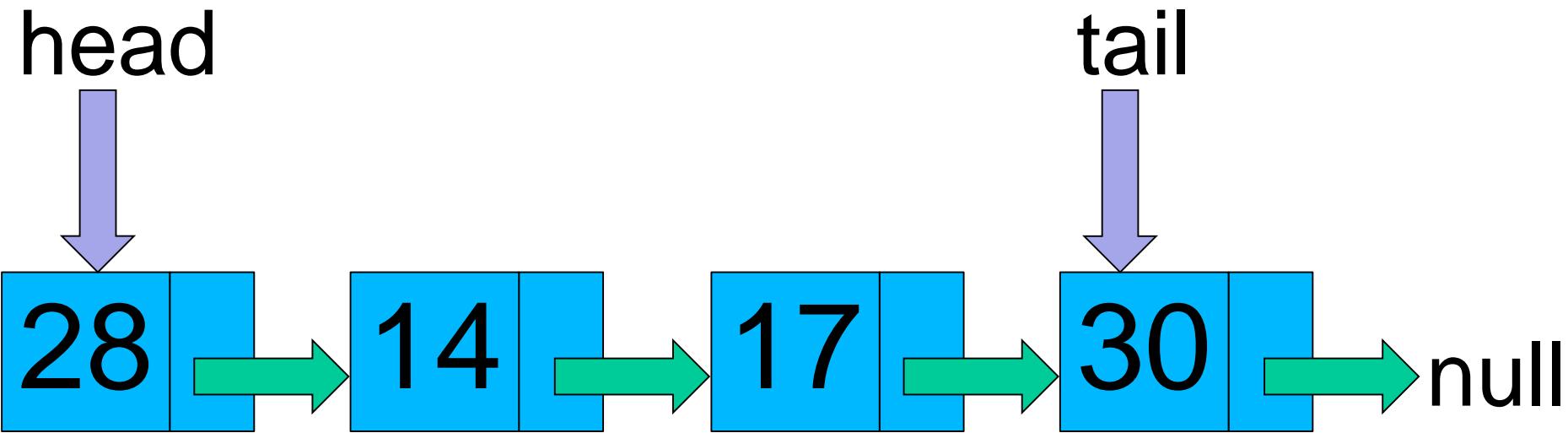
LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
    }  
}
```

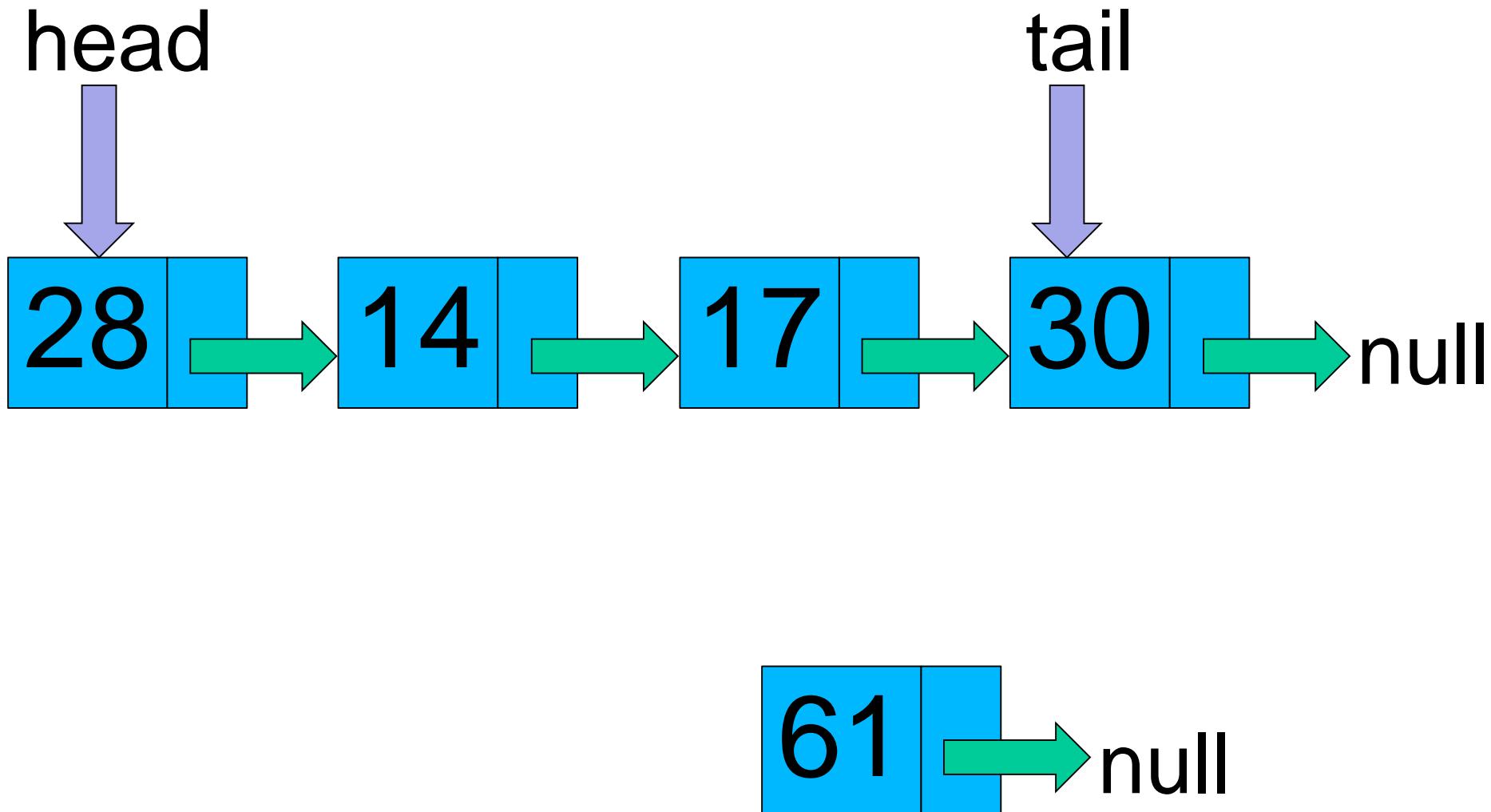
LinkedList: Adding to the front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void addToFront(int value) {  
        Node newNode = new Node(value);  
        newNode.next = head;  
        head = newNode;  
        if (newNode.next == null) {  
            tail = newNode;  
        }  
    }  
}
```

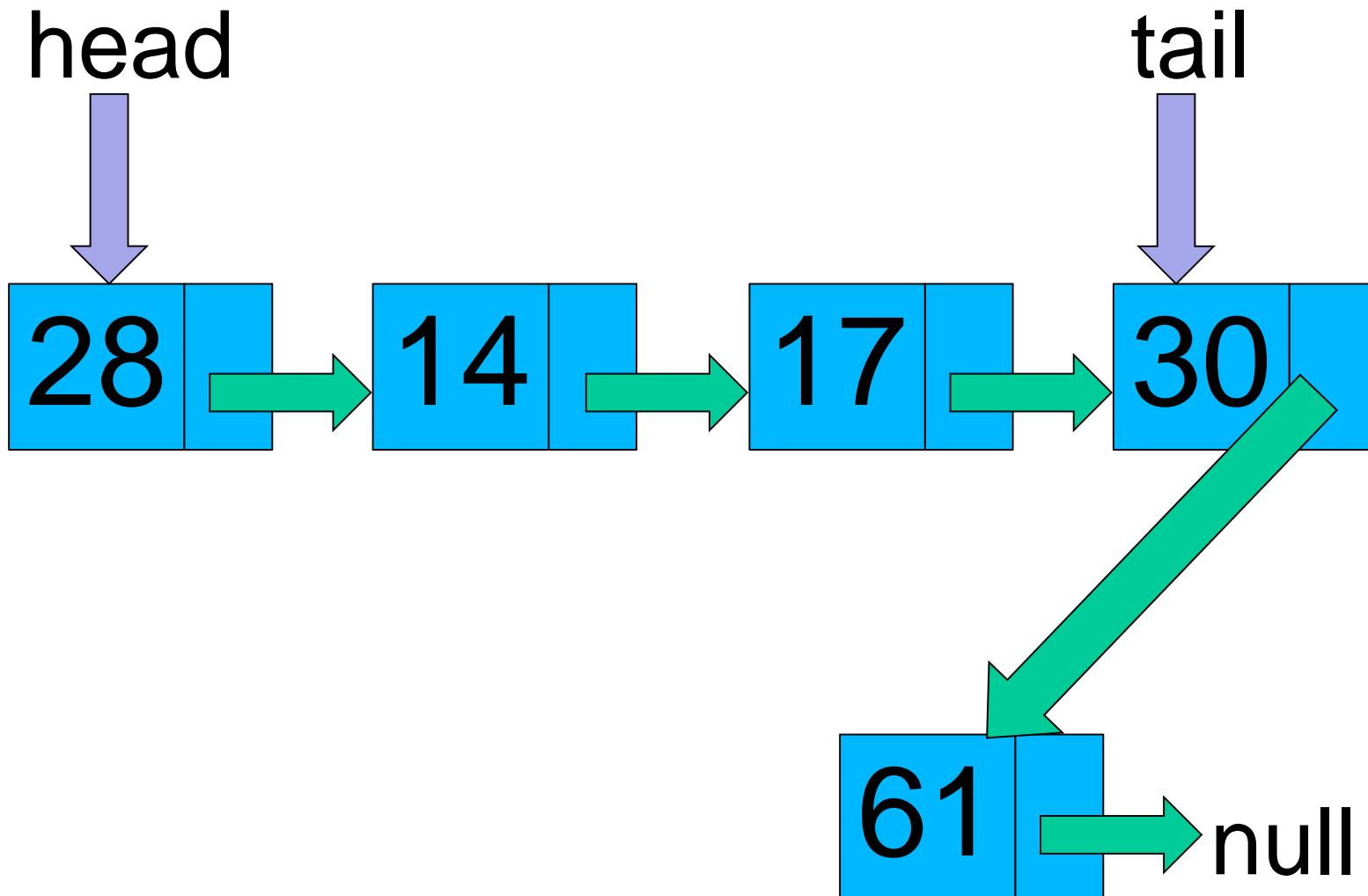
Adding to the back



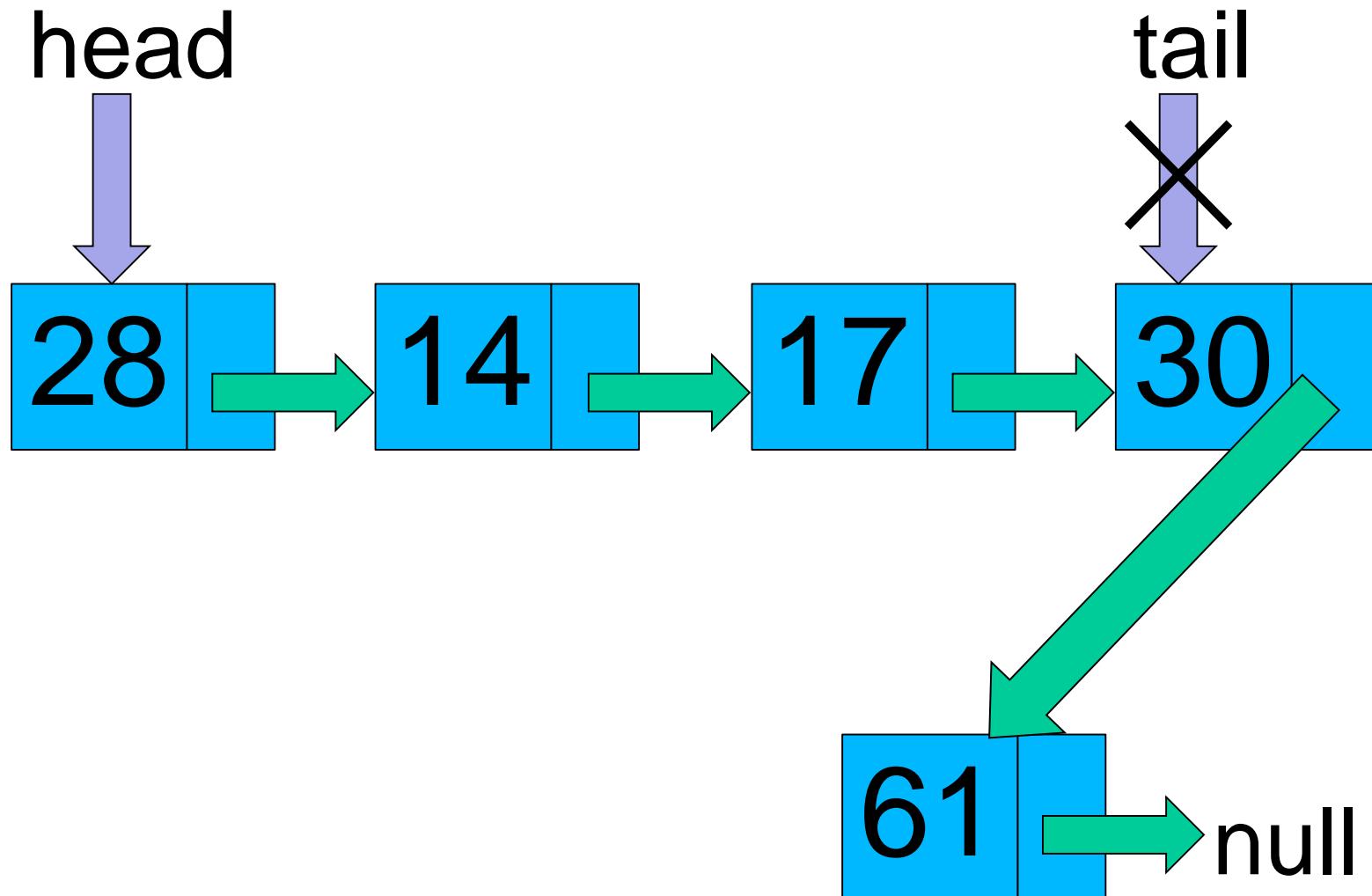
Adding to the back



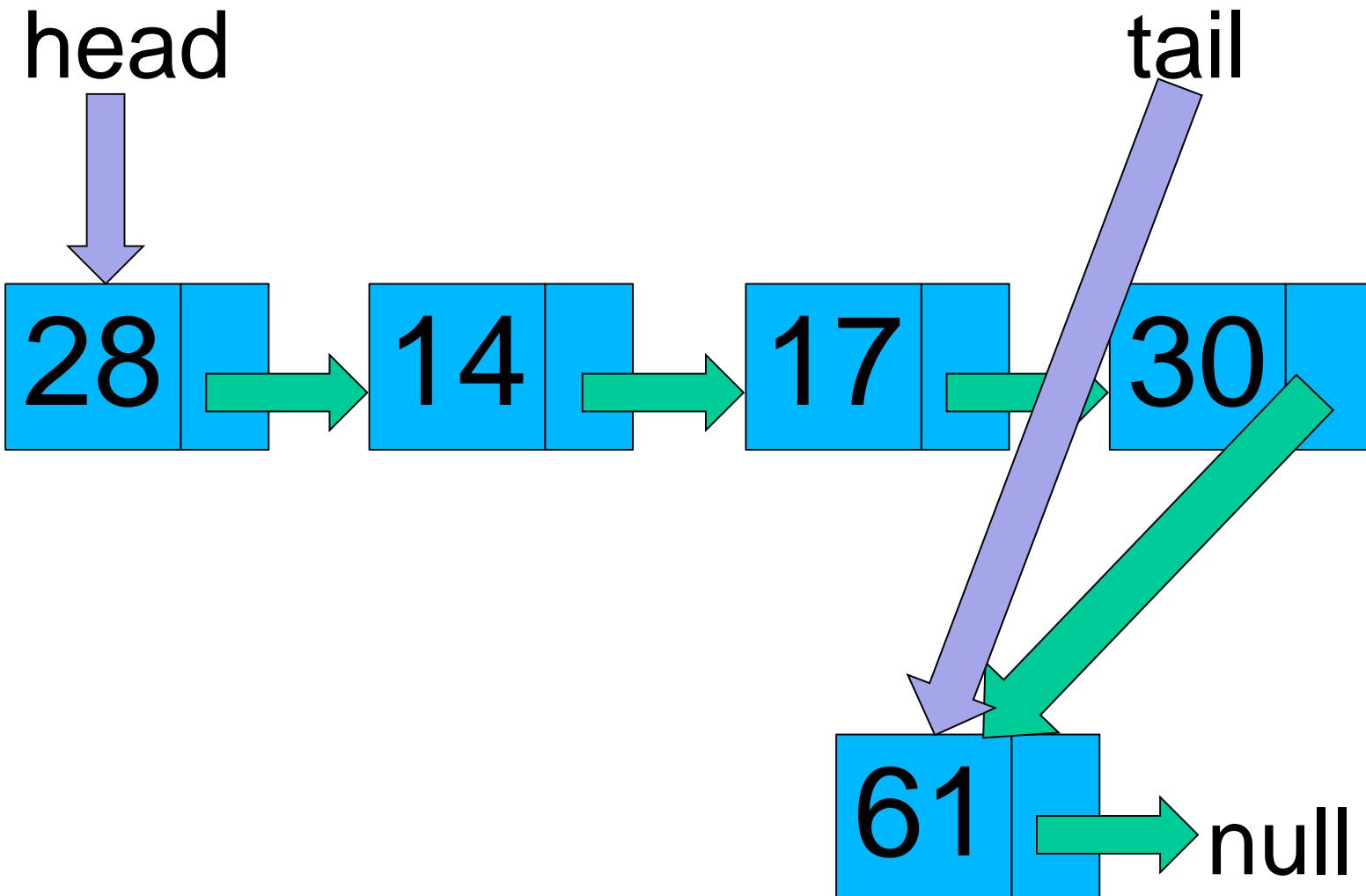
Adding to the back



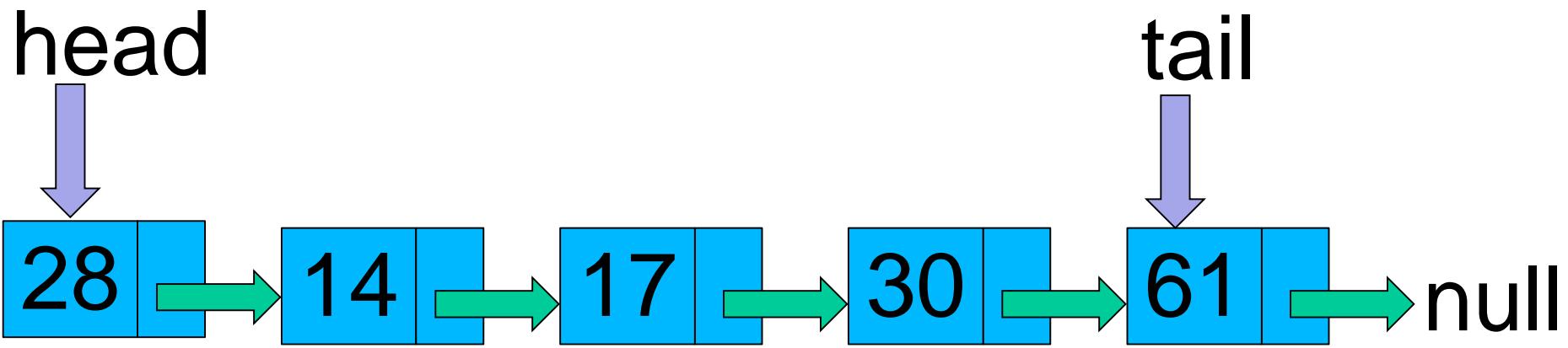
Adding to the back



Adding to the back



Adding to the back



LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        tail.next = newNode;  
        tail = newNode;  
    }  
}
```

LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
public void addToBack(int value) {  
    Node newNode = new Node(value);  
    tail.next = newNode;  
    tail = newNode;  
}  
}
```

LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        tail.next = newNode;  
        tail = newNode;  
    }  
}
```

LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        tail.next = newNode;  
        tail = newNode;  
    }  
}
```

LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        tail.next = newNode;  
        tail = newNode;  
    }  
}
```

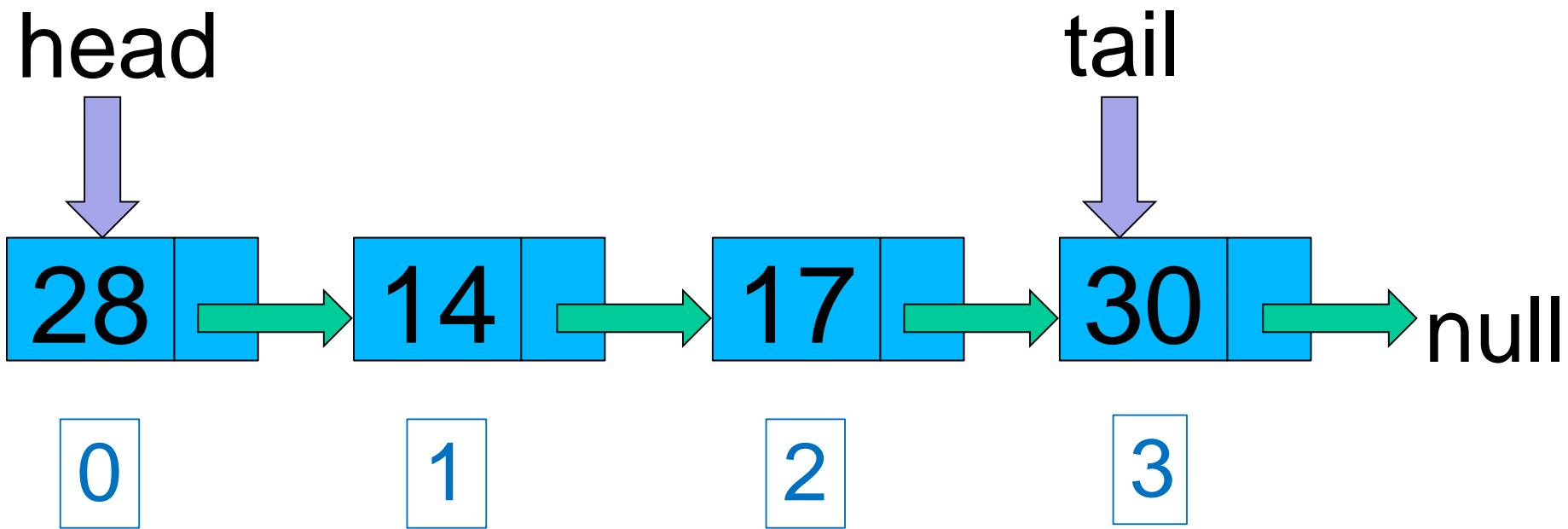
LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        tail.next = newNode;  
        tail = newNode;  
    }  
}
```

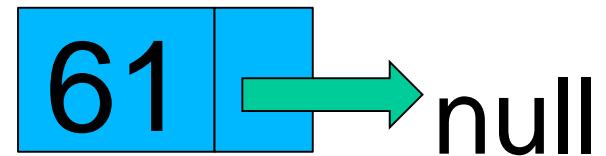
LinkedList: Adding to the back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void addToBack(int value) {  
        Node newNode = new Node(value);  
        if (tail == null) {  
            head = newNode;  
        } else {  
            tail.next = newNode;  
        }  
        tail = newNode;  
    }  
}
```

Adding at Index

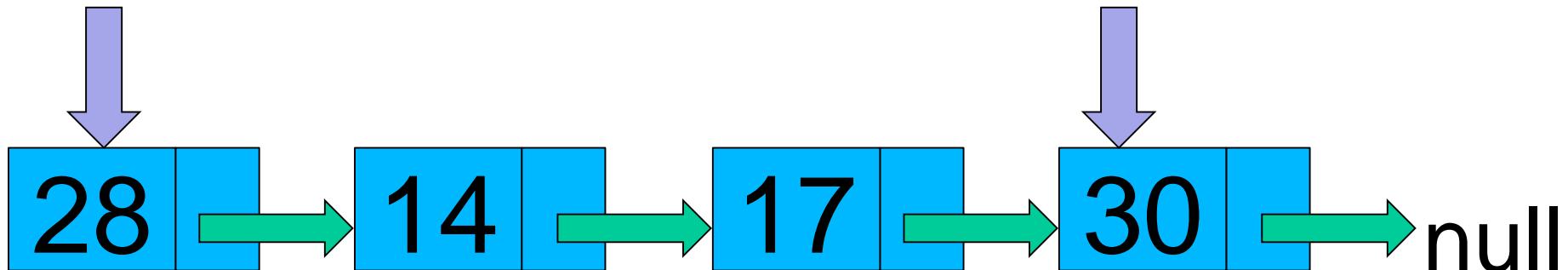


Adding at Index



head

tail



0

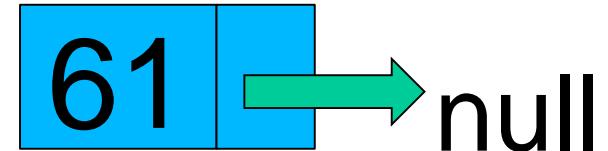
1

2

3

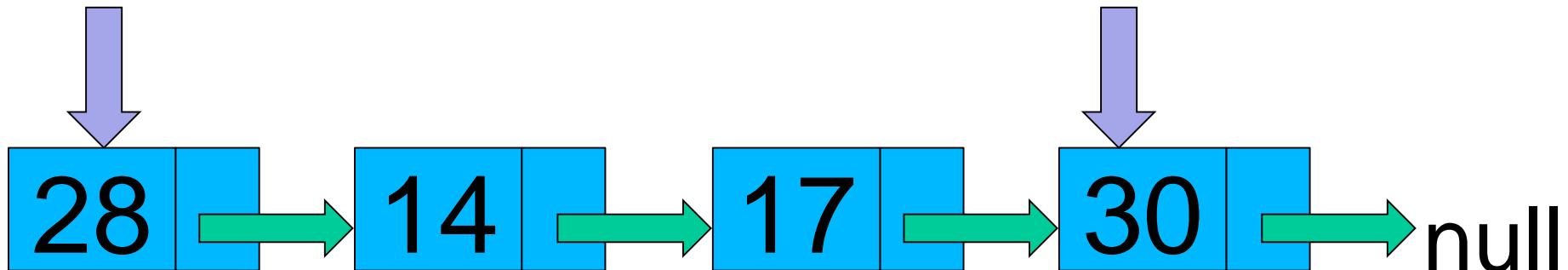
Adding at Index

index = 2



head

tail



0

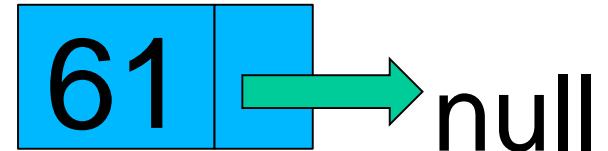
1

2

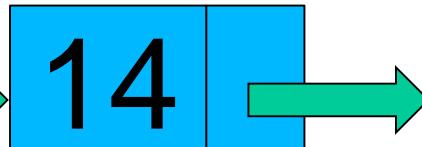
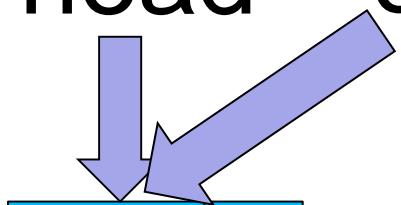
3

Adding at Index

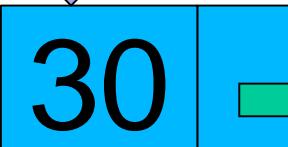
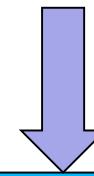
index = 2



head current



tail



null

0

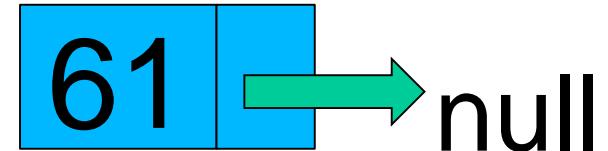
1

2

3

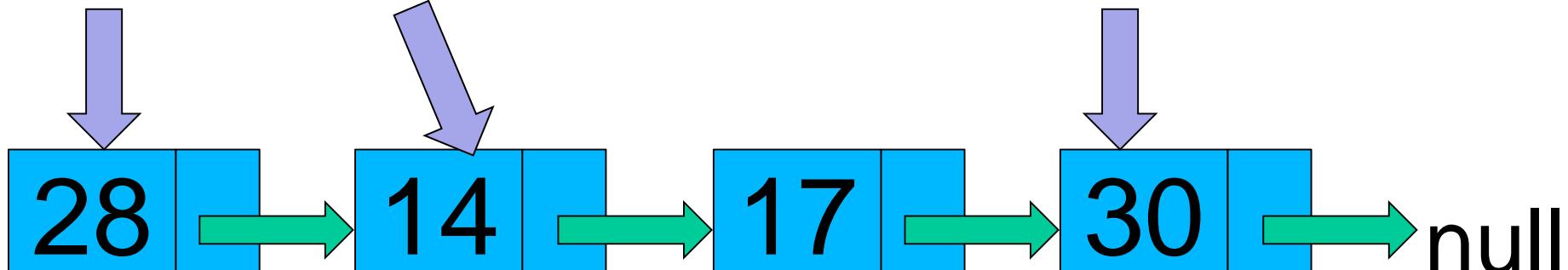
Adding at Index

index = 2



head current

tail



0

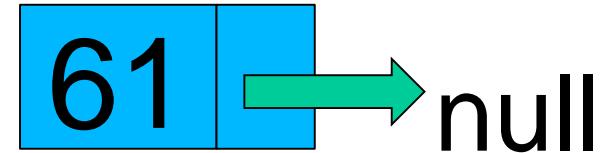
1

2

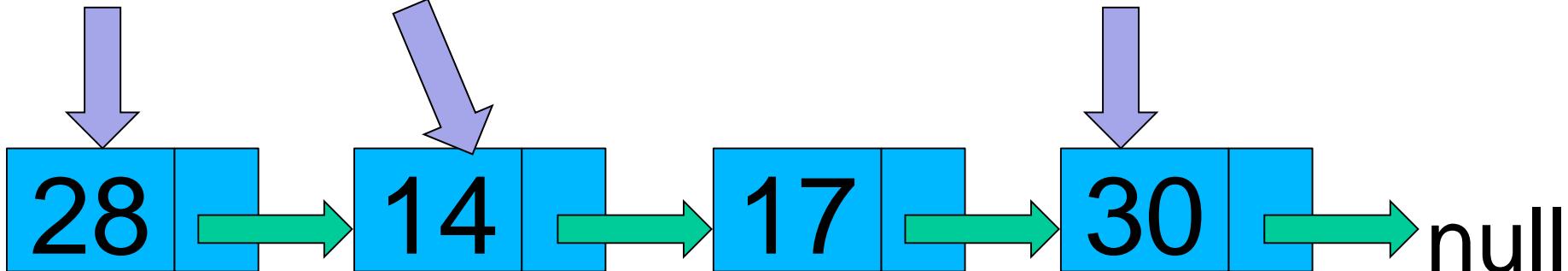
3

Adding at Index

index = 2



head current



0

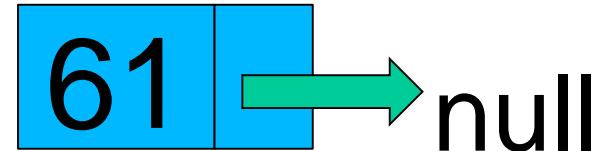
1

2

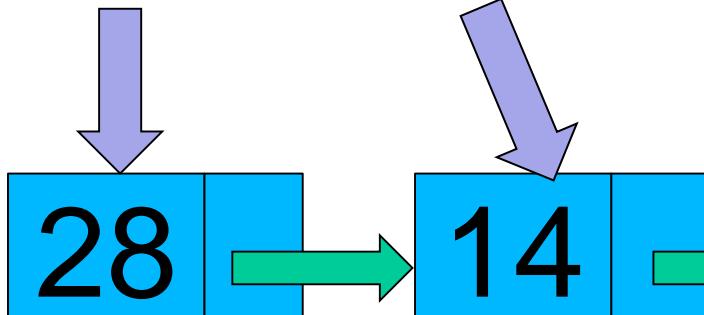
3

Adding at Index

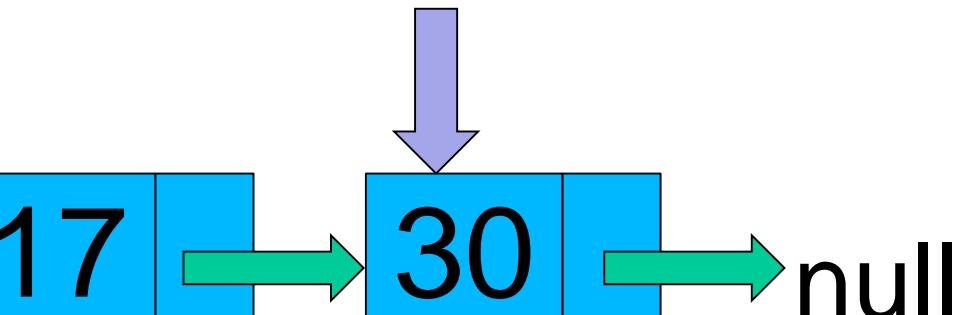
index = 2



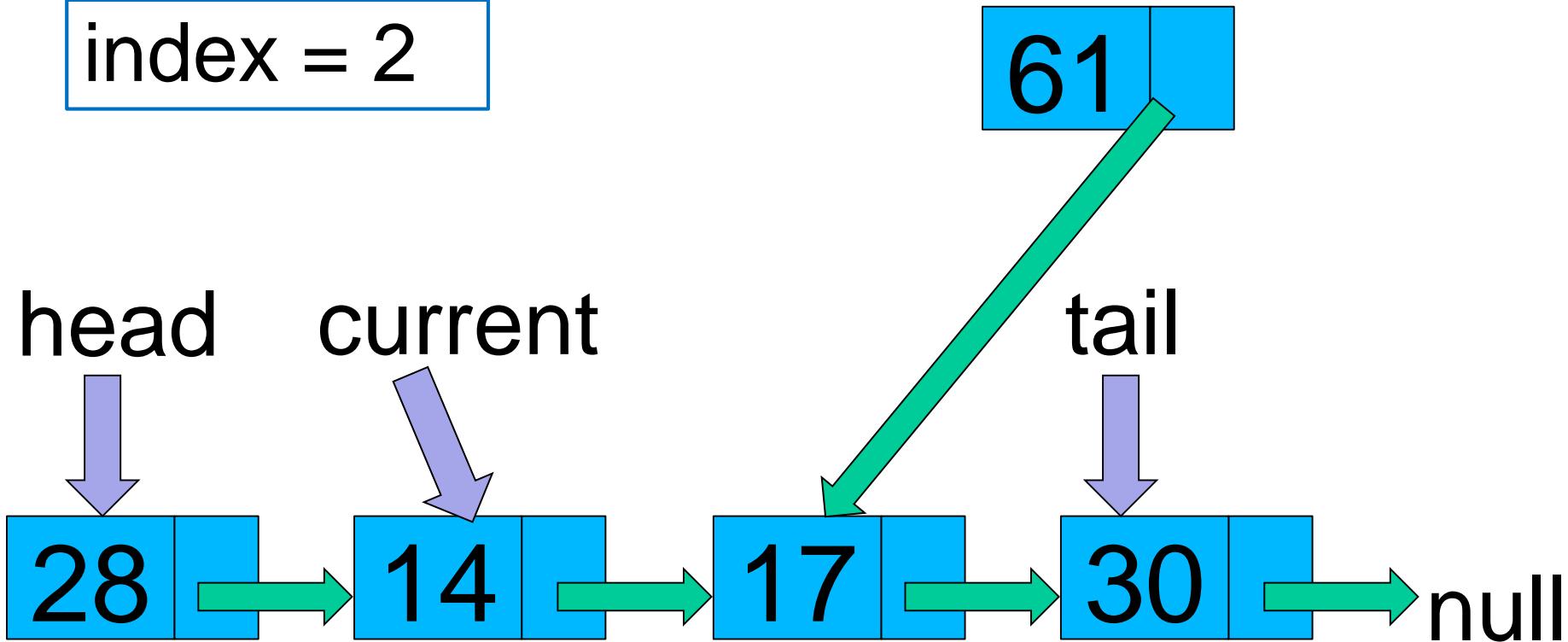
head current



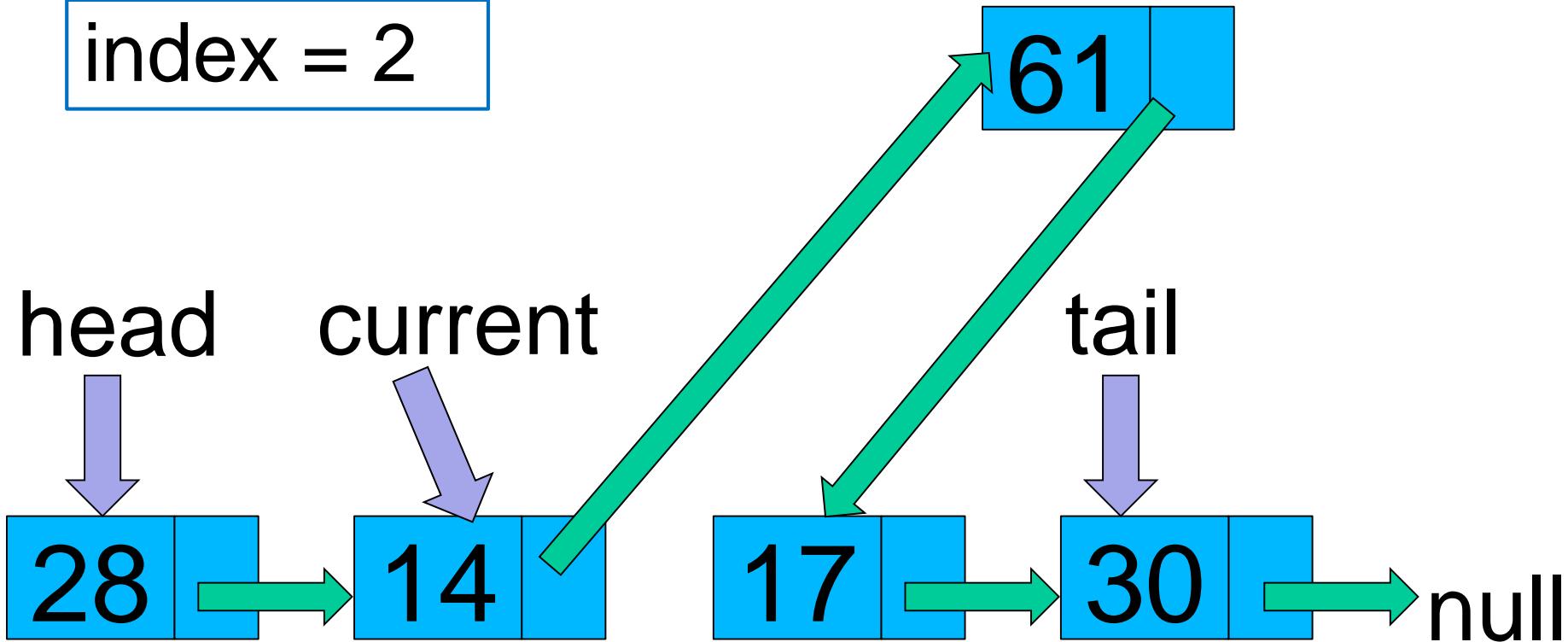
tail



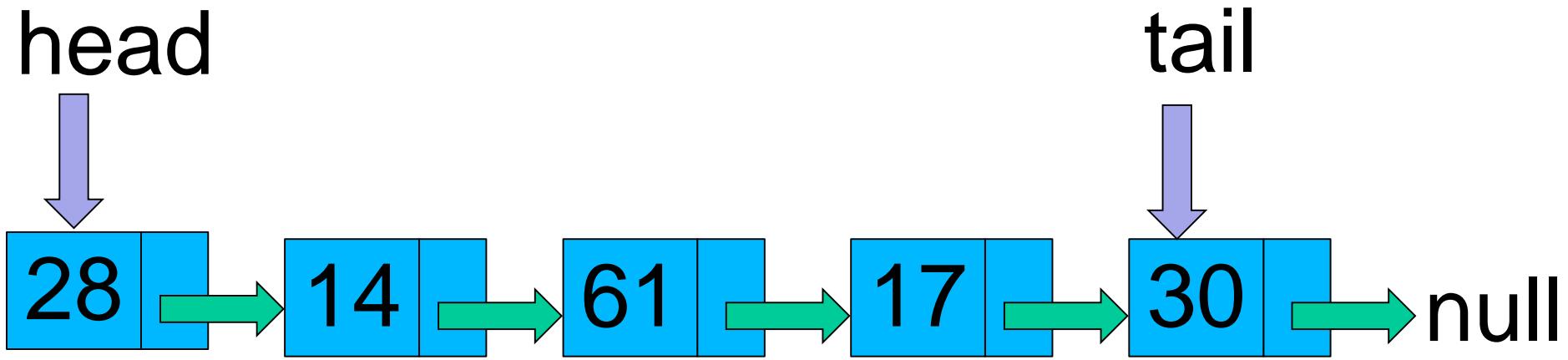
Adding at Index



Adding at Index



Adding at Index



LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
public void addAtIndex(int index, int value) {  
    Node newNode = new Node(value);  
    Node current = head;  
    for (int i = 0; i < index - 1; i++) {  
        current = current.next;  
    }  
    newNode.next = current.next;  
    current.next = newNode;  
}  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public class LinkedList {  
    . . .  
    public void addAtIndex(int index, int value) {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        newNode.next = current.next;  
        current.next = newNode;  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedList: Adding at Index

```
public void addAtIndex(int index, int value) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) { // adding to head  
        addToFront(value);  
    } else {  
        Node newNode = new Node(value);  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        if (current.next == null) { // adding to tail  
            tail = newNode;  
        } else {  
            newNode.next = current.next;  
            current.next = newNode;  
        }  
    }  
}
```

LinkedLists compared to arrays

- **Don't** need to know the number of elements when the Linked List is created
- **Can** easily insert an element in front of others

SD2x1.3

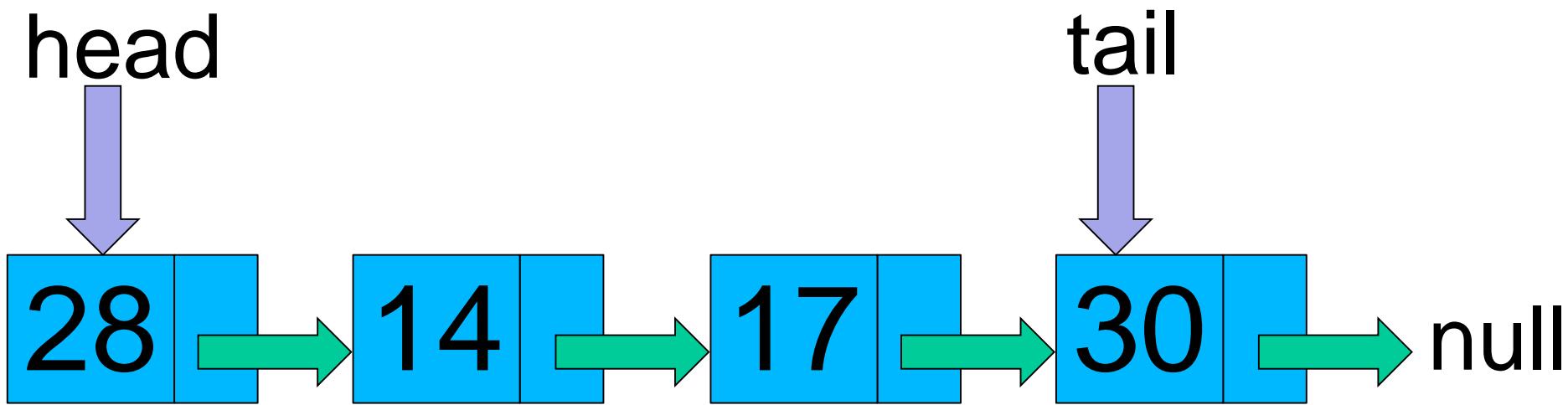
**LL: get by index, contains
Chris**

What else can we do with a Linked List?

- Determine whether the Linked List contains a particular value
- Get the value at a particular index

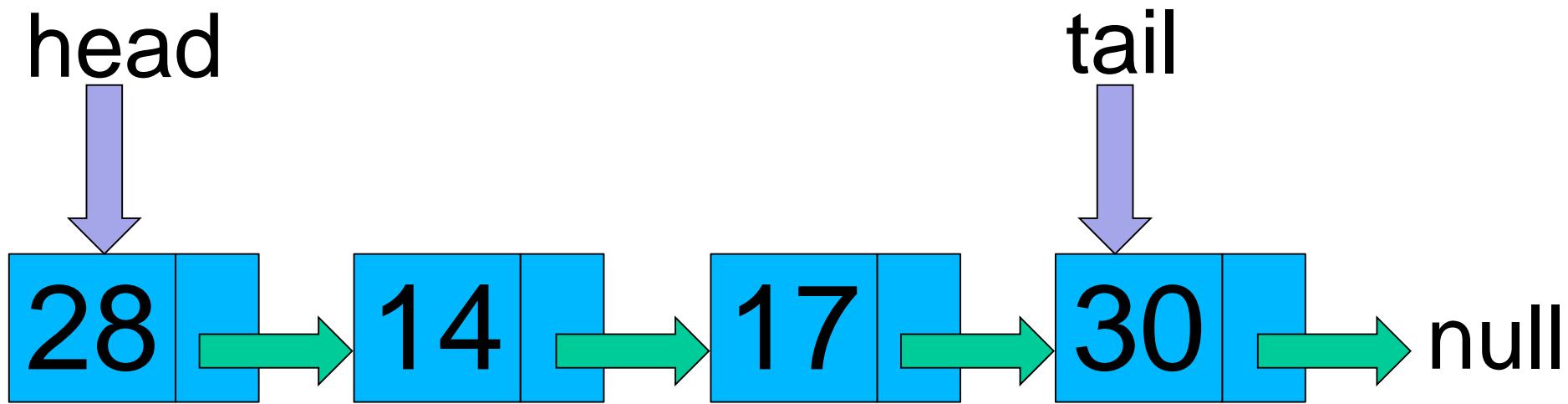
**How do we determine
whether the Linked List
contains a particular value?**

LinkedList contains the value



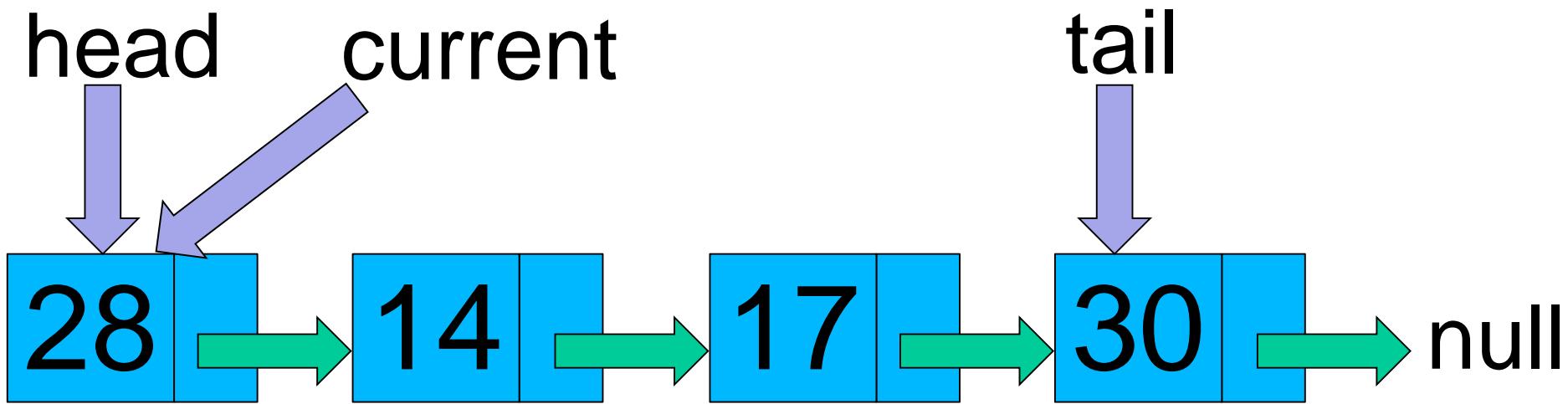
LinkedList contains the value

value = 17



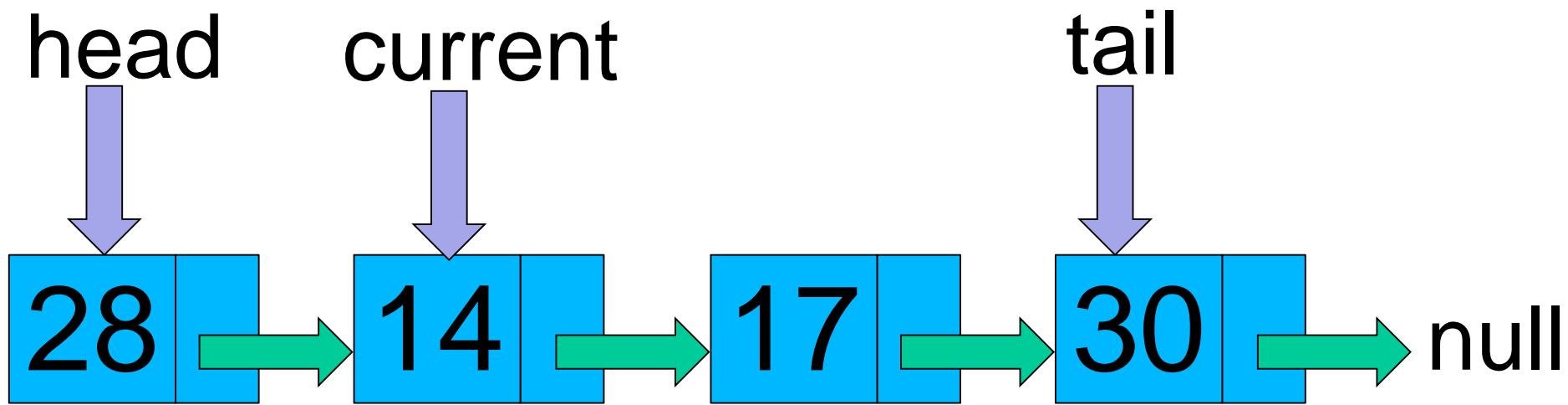
LinkedList contains the value

value = 17



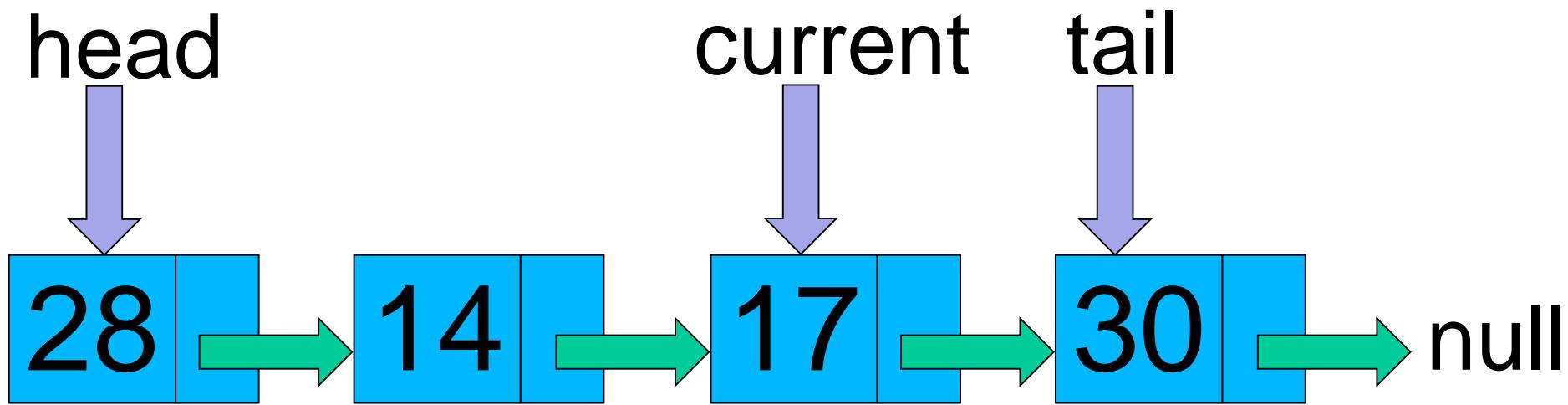
LinkedList contains the value

value = 17



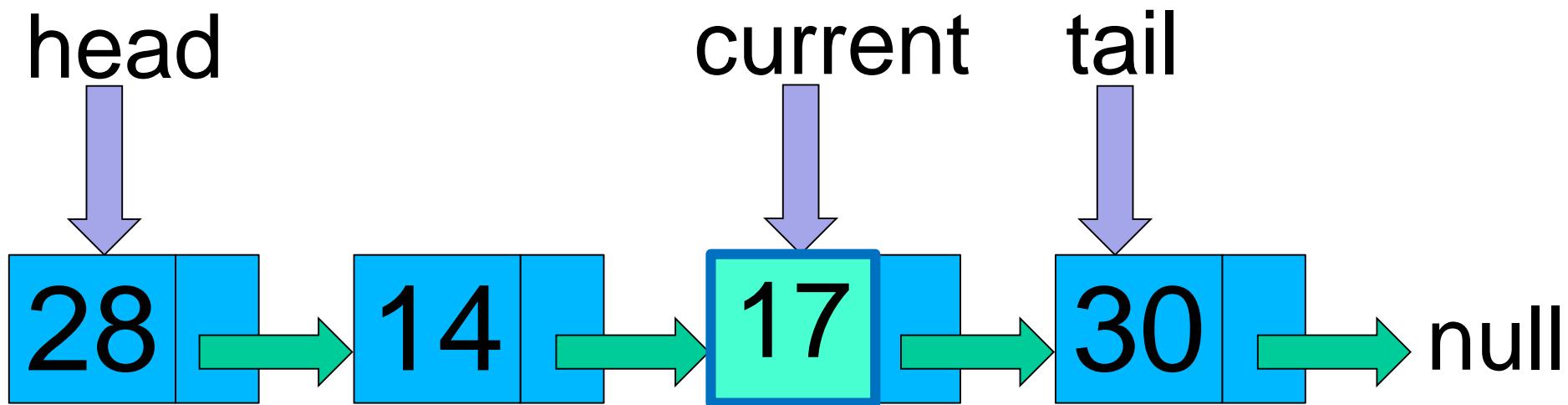
LinkedList contains the value

value = 17

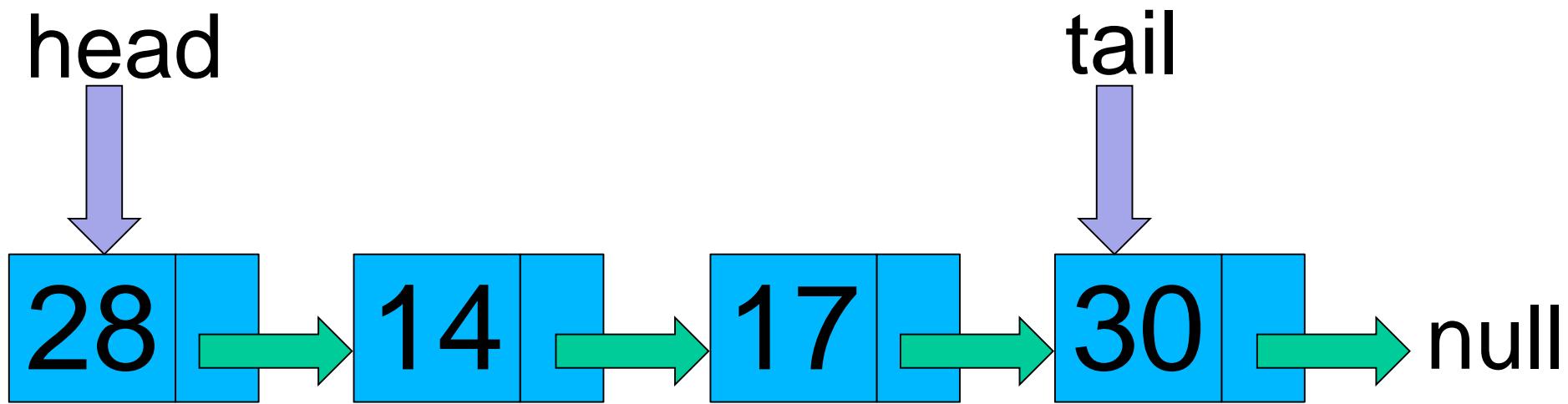


LinkedList contains the value

value = 17

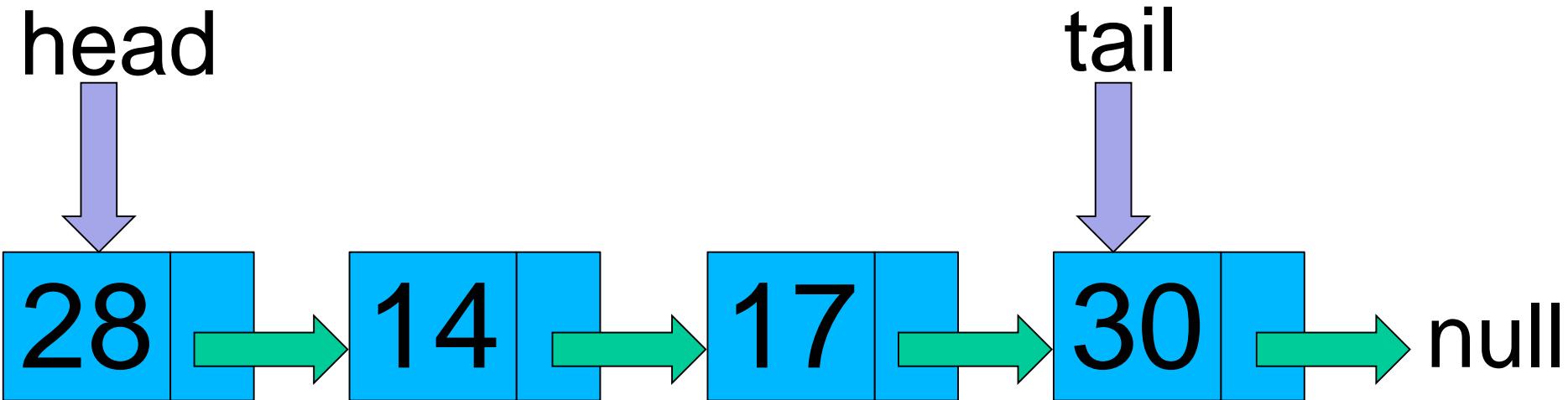


LinkedList does not contain the value



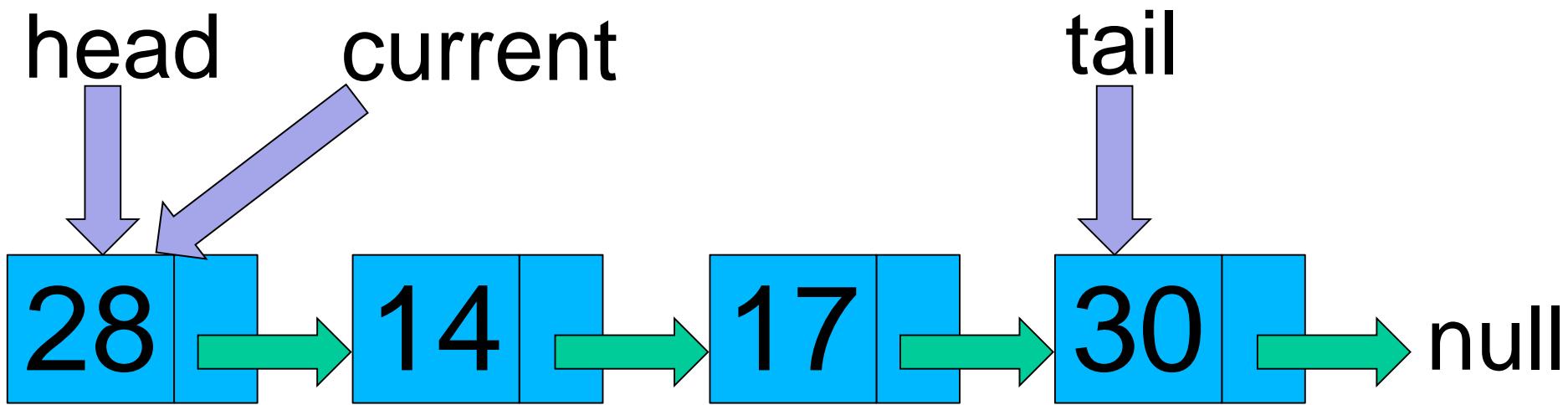
LinkedList does not contain the value

value = 7



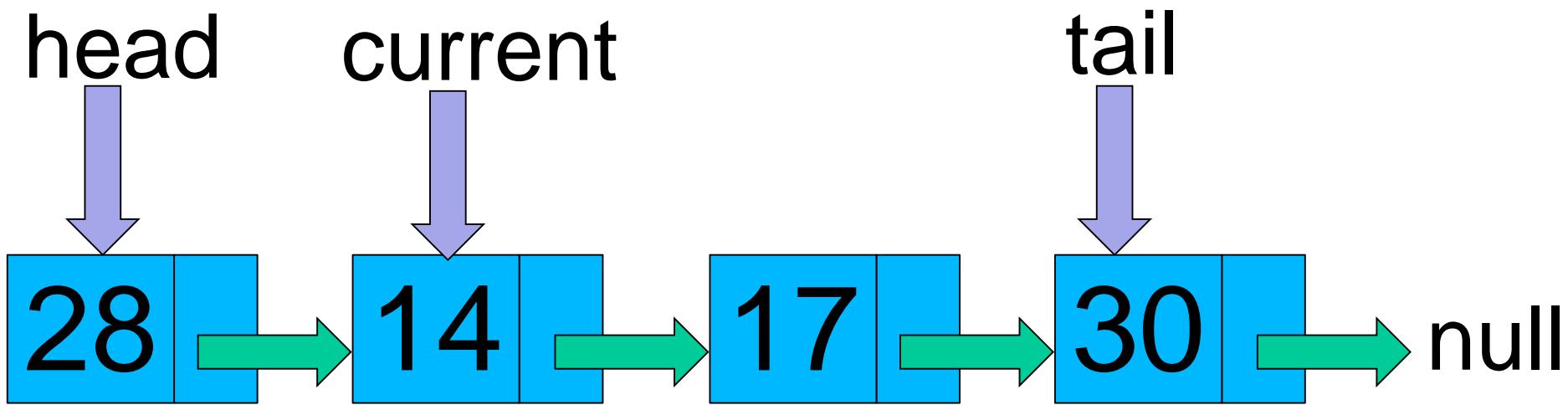
LinkedList does not contain the value

value = 7



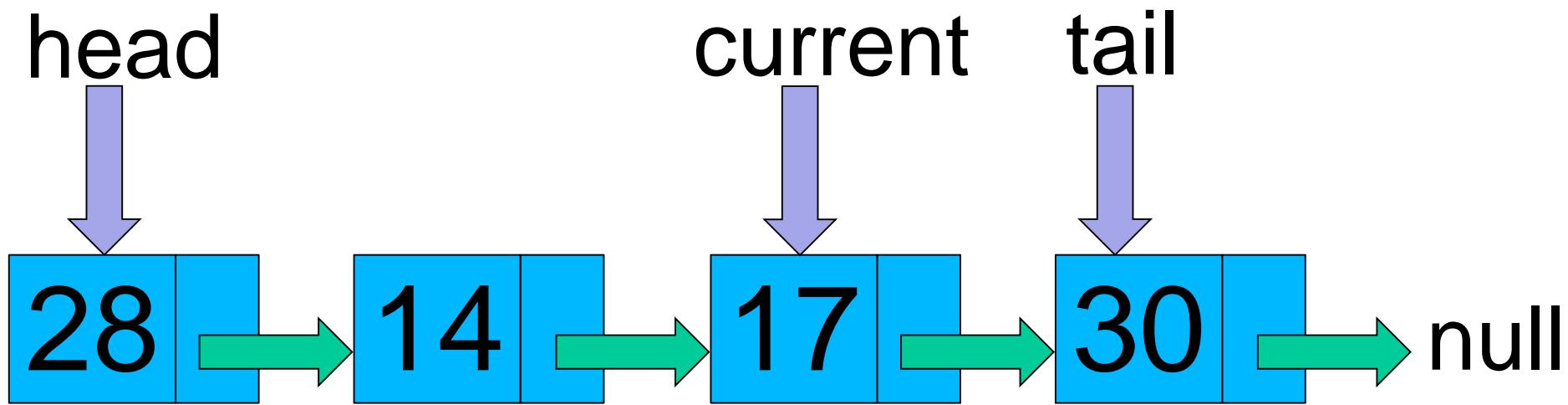
LinkedList does not contain the value

value = 7



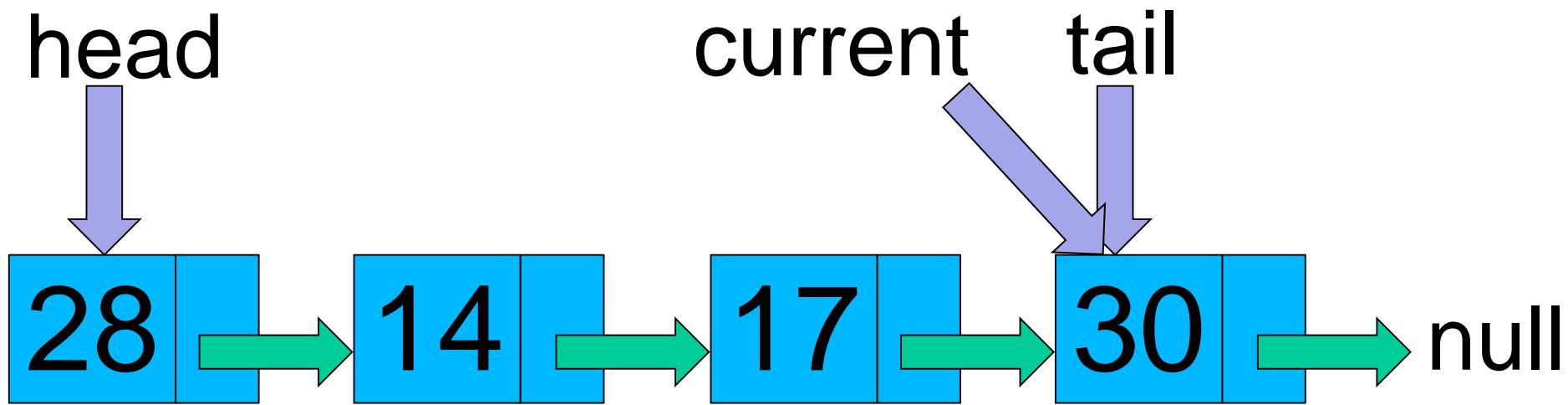
LinkedList does not contain the value

value = 7



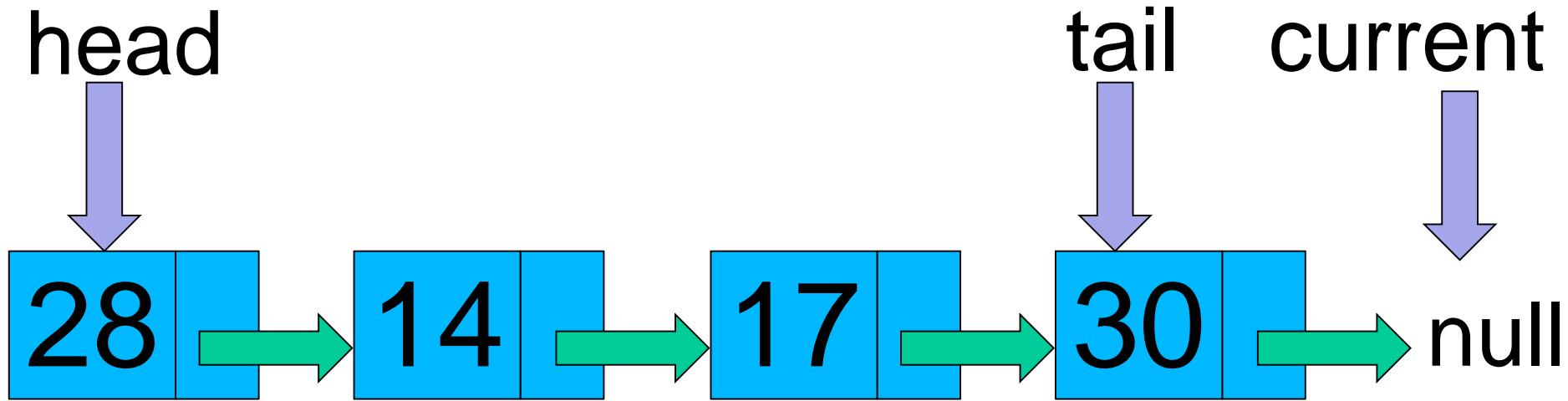
LinkedList does not contain the value

value = 7



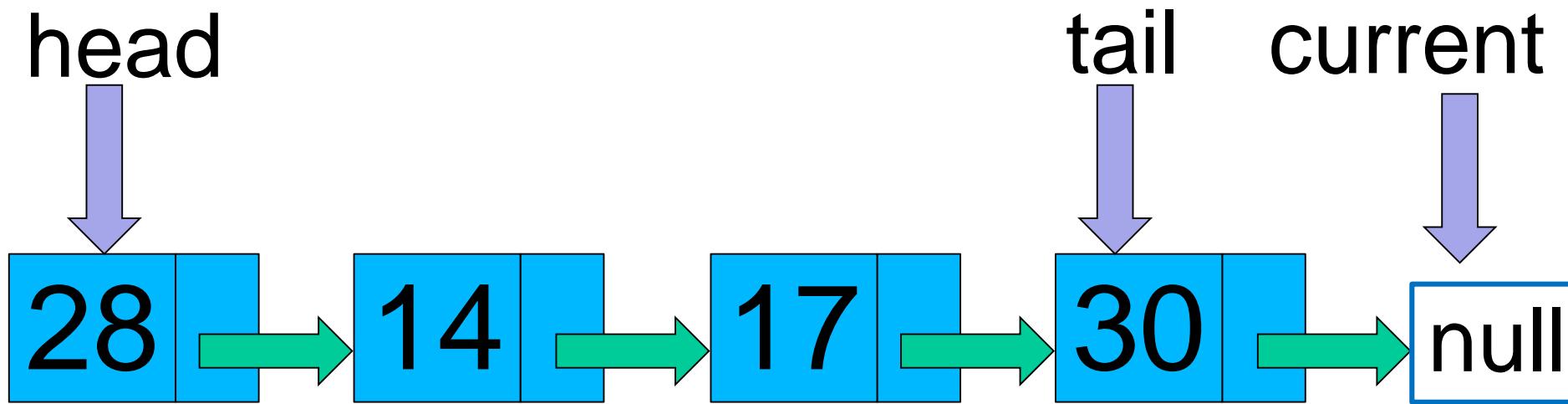
LinkedList does not contain the value

value = 7



LinkedList does not contain the value

value = ~~7~~



LinkedList: contains

```
public class LinkedList {  
    . . .  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

LinkedList: contains

```
public class LinkedList {  
    . . .  
public boolean contains(int value) {  
    Node current = head;  
    while (current != null) {  
        if (current.value == value) {  
            return true;  
        }  
        current = current.next;  
    }  
    return false;  
}  
}
```

LinkedList: contains

```
public class LinkedList {  
    . . .  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

LinkedList: contains

```
public class LinkedList {  
    . . .  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

LinkedList: contains

```
public class LinkedList {  
    . . .  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

LinkedList: contains

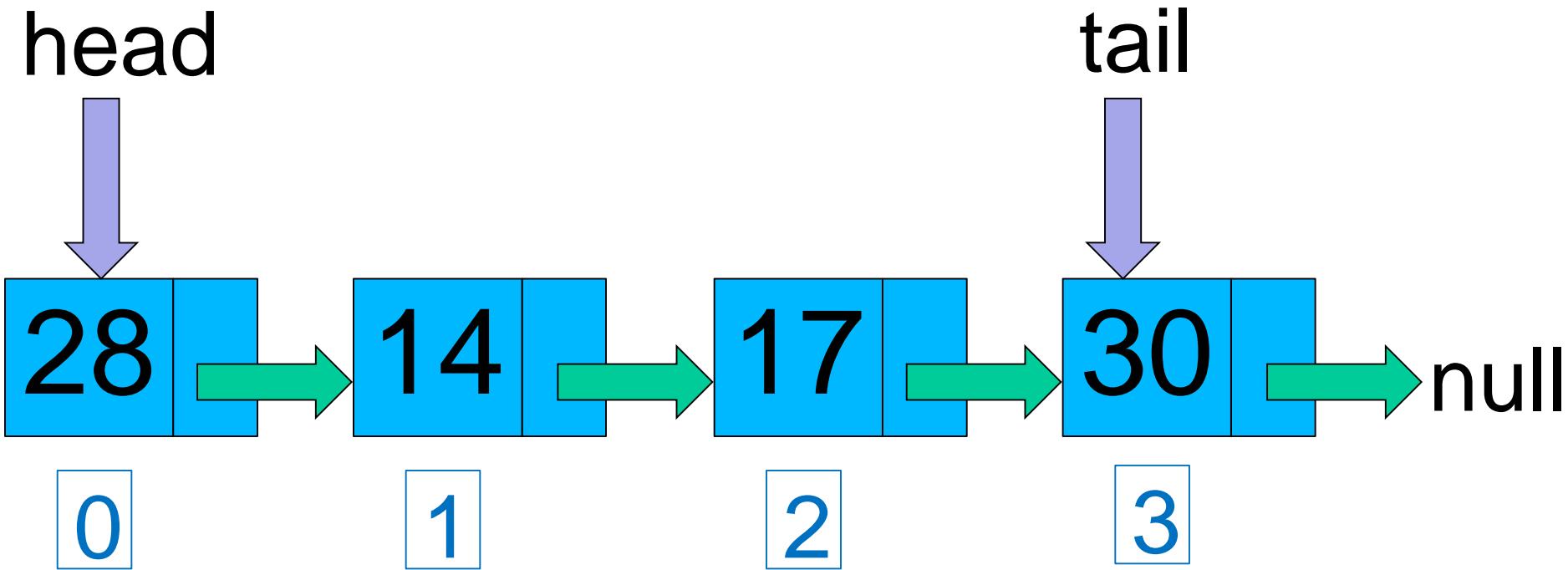
```
public class LinkedList {  
    . . .  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

LinkedList: contains

```
public class LinkedList {  
    . . .  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) {  
                return true;  
            }  
            current = current.next;  
        }  
        return false;  
    }  
}
```

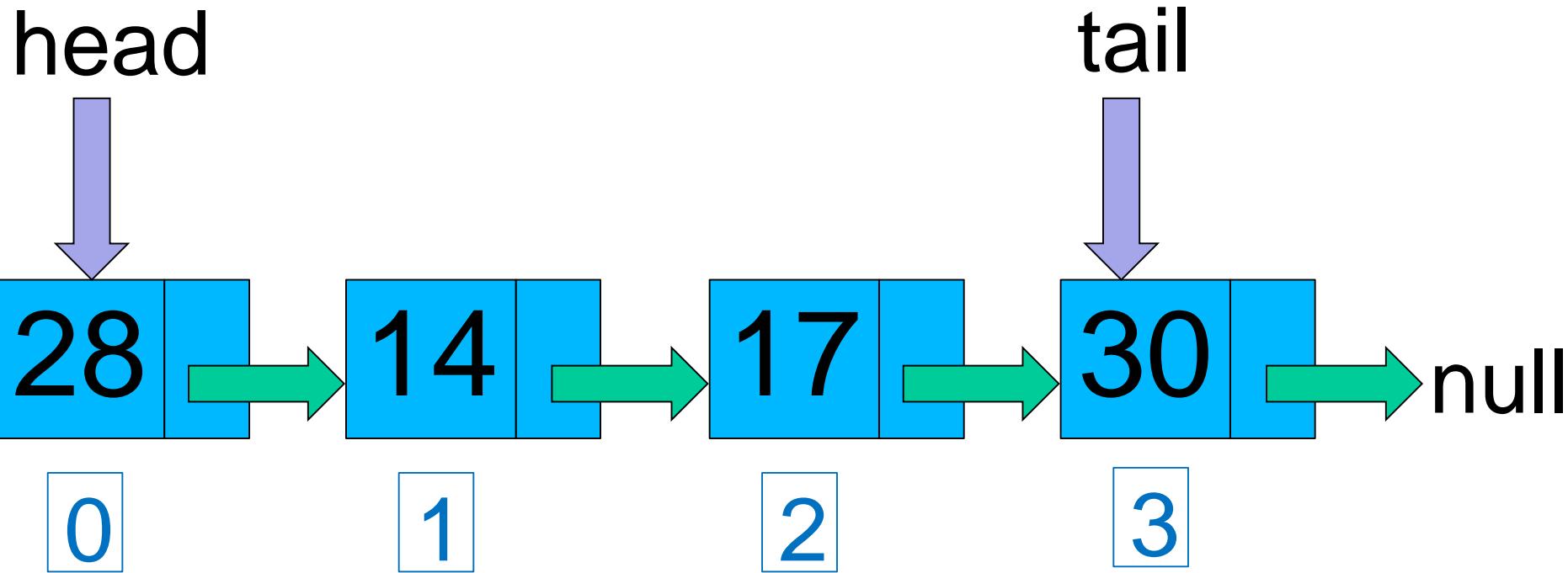
How do we get the element at an index in the Linked List?

LinkedList: Get by index



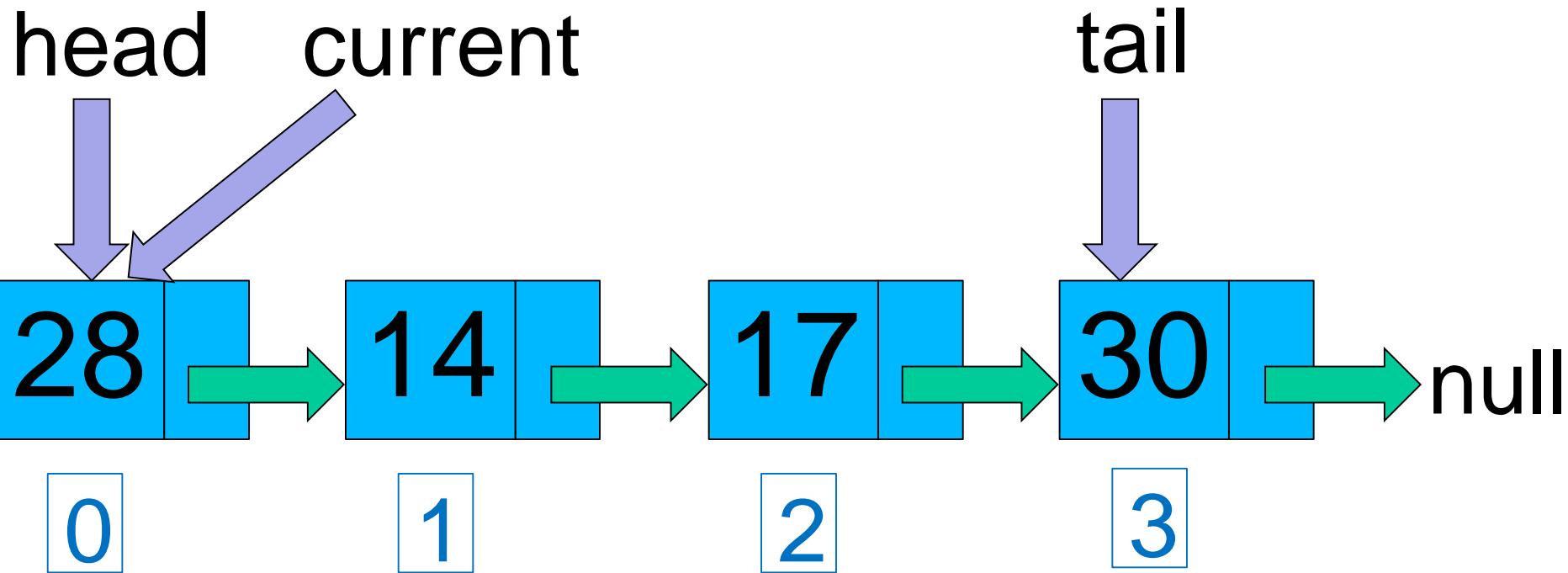
LinkedList: Get by index

index = 3



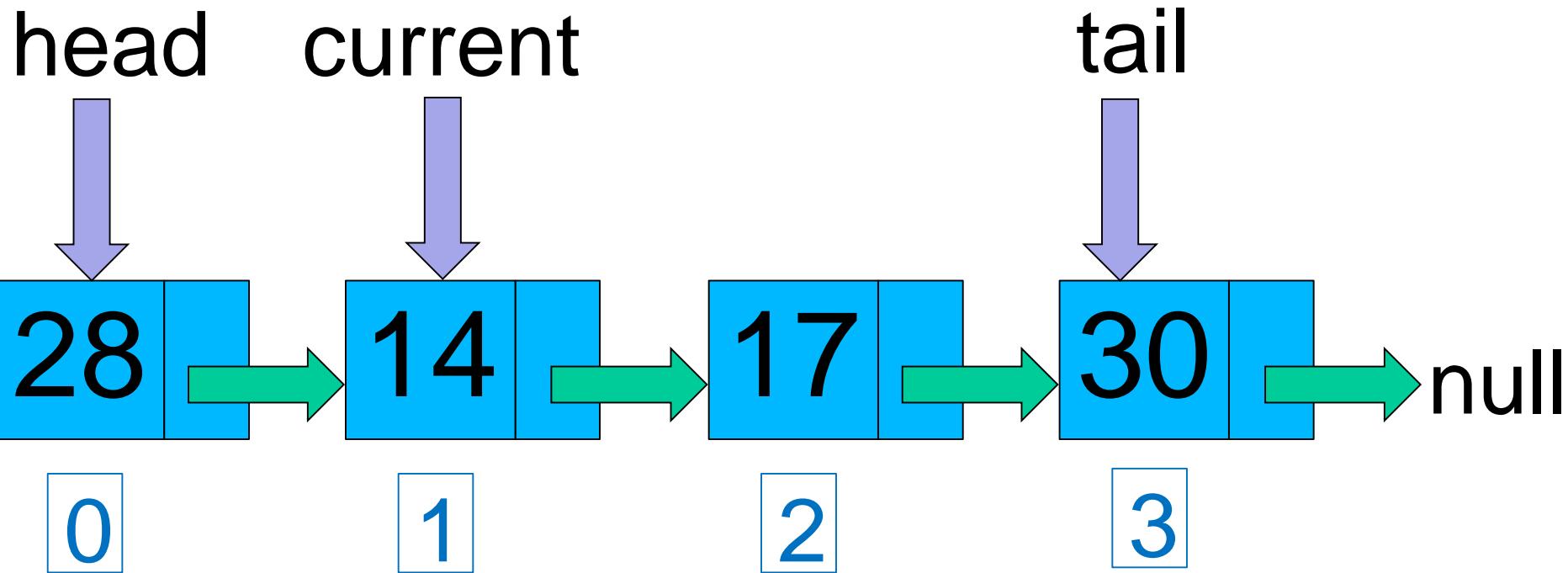
LinkedList: Get by index

index = 3



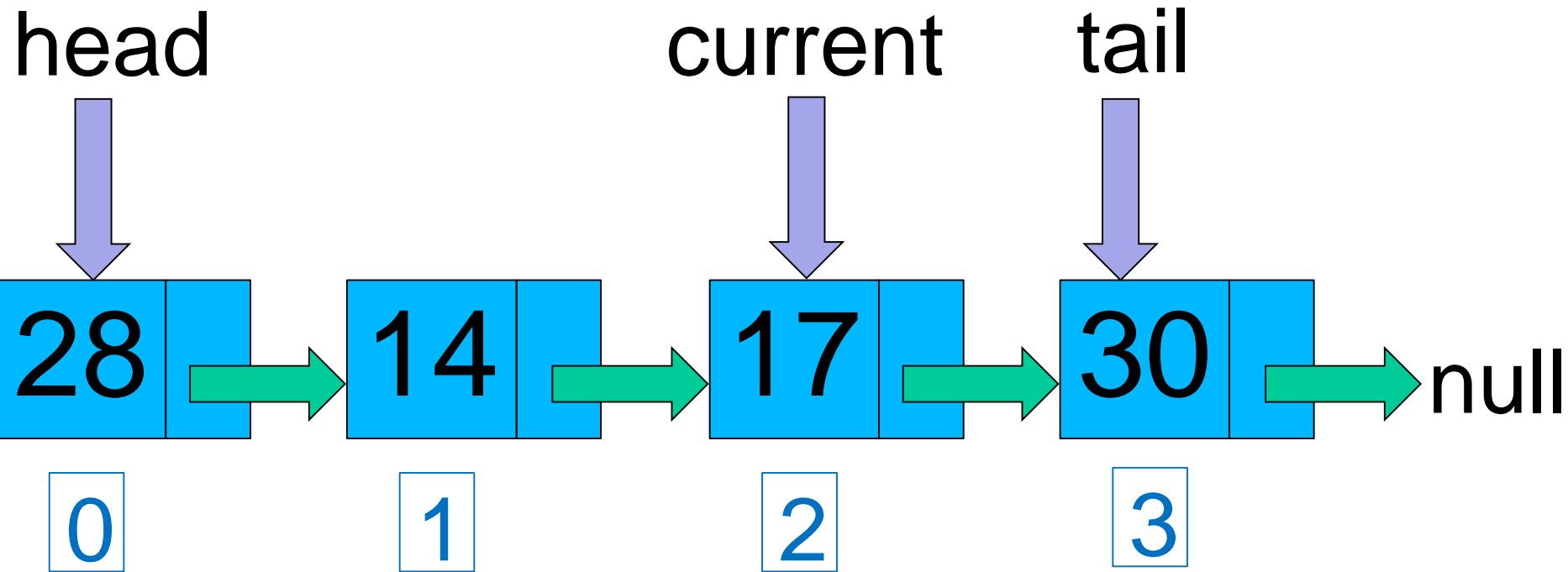
LinkedList: Get by index

index = 3



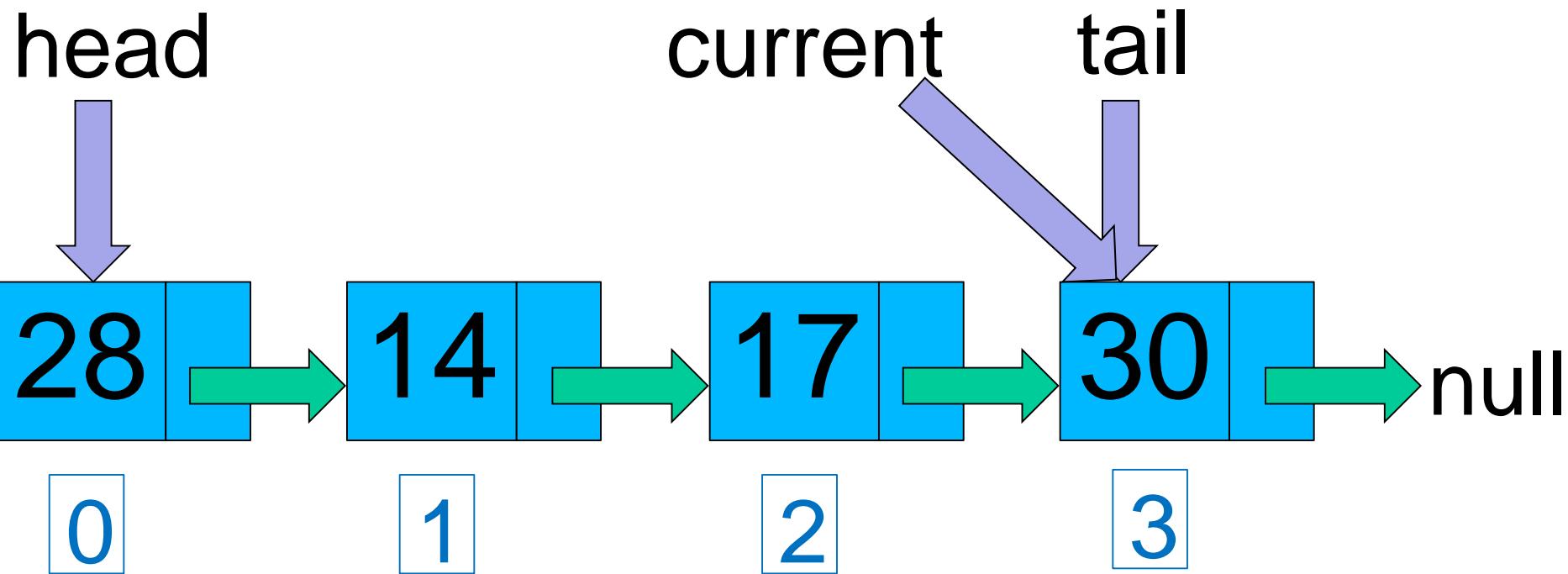
LinkedList: Get by index

index = 3



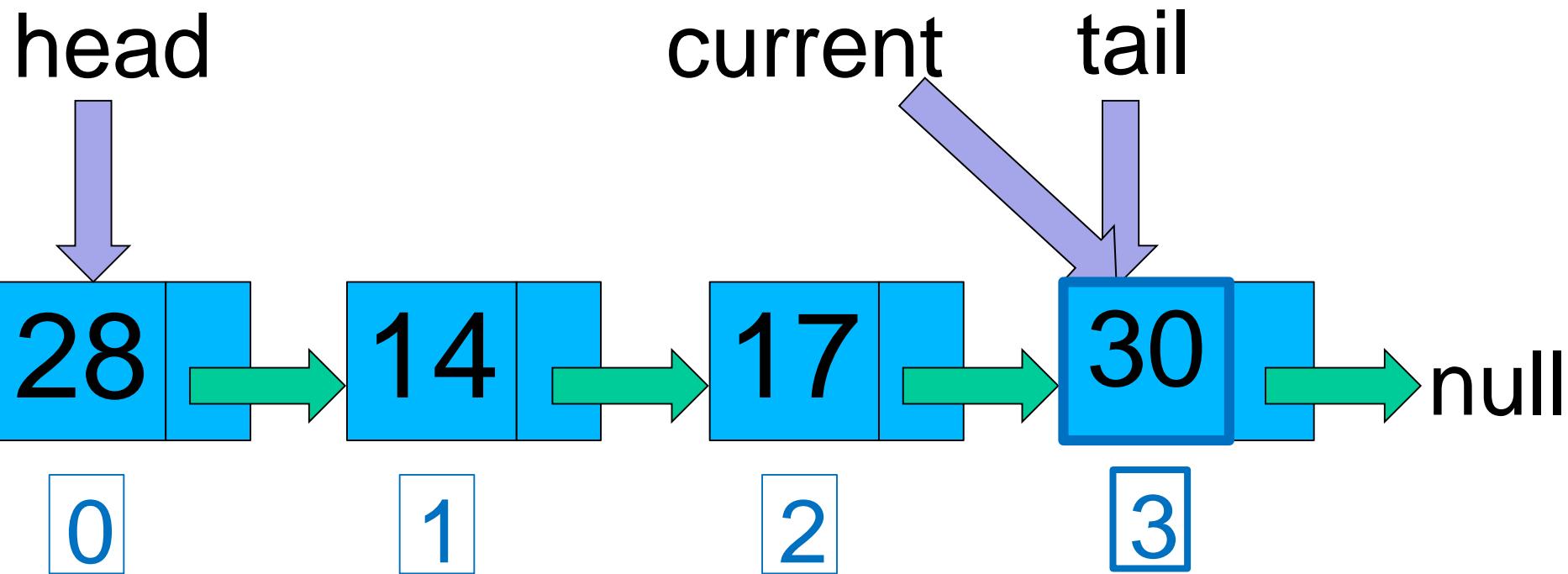
LinkedList: Get by index

index = 3



LinkedList: Get by index

index = 3



LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            current = current.next;  
        }  
        return current.value;  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
public int getByIndex(int index) {  
    Node current = head;  
    for (int i = 0; i < index; i++) {  
        current = current.next;  
    }  
    return current.value;  
}  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            current = current.next;  
        }  
        return current.value;  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            current = current.next;  
        }  
        return current.value;  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            current = current.next;  
        }  
        return current.value;  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            current = current.next;  
        }  
        return current.value;  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        if (index < 0) {  
            throw new IndexOutOfBoundsException();  
        } else {  
            Node current = head;  
            for (int i = 0; i < index; i++) {  
                if (current == null || current.next == null) {  
                    throw new IndexOutOfBoundsException();  
                }  
                current = current.next;  
            }  
            return current.value;  
        }  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        if (index < 0) {  
            throw new IndexOutOfBoundsException();  
        } else {  
            Node current = head;  
            for (int i = 0; i < index; i++) {  
                if (current == null || current.next == null) {  
                    throw new IndexOutOfBoundsException();  
                }  
                current = current.next;  
            }  
            return current.value;  
        }  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
    public int getByIndex(int index) {  
        if (index < 0) {  
            throw new IndexOutOfBoundsException();  
        } else {  
            Node current = head;  
            for (int i = 0; i < index; i++) {  
                if (current == null || current.next == null) {  
                    throw new IndexOutOfBoundsException();  
                }  
                current = current.next;  
            }  
            return current.value;  
        }  
    }  
}
```

LinkedList: Get by index

```
public class LinkedList {  
    . . .  
  
    public int getByIndex(int index) {  
        if (index < 0) {  
            throw new IndexOutOfBoundsException();  
        } else {  
            Node current = head;  
            for (int i = 0; i < index; i++) {  
                if (current == null || current.next == null) {  
                    throw new IndexOutOfBoundsException();  
                }  
                current = current.next;  
            }  
            return current.value;  
        }  
    }  
}
```

LinkedLists compared to arrays

- Don't need to know the number of elements when the LinkedList is created
- Can easily insert an element in front of others

LinkedLists compared to arrays

- Don't need to know the number of elements when the LinkedList is created
- Can easily insert an element in front of others
- Still may need to compare against all elements in order to search for an element

LinkedLists compared to arrays

- Don't need to know the number of elements when the LinkedList is created
 - Can easily insert an element in front of others
-
- Still may need to compare against all elements in order to search for an element
 - Getting an element by index requires following links

SD2x1.4

LL: removing

Kathy

Review: Linked Lists

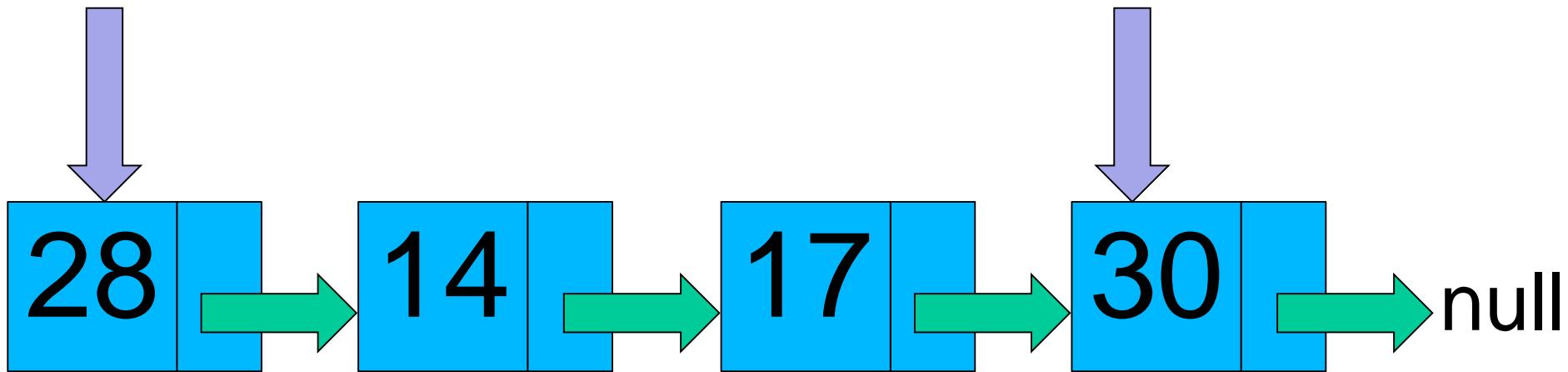
- Series of “Nodes” with each one linked to the next
- Can insert a value by creating a new Node and updating the links
- Can retrieve a value by traversing the links

How do we remove a value from the Linked List?

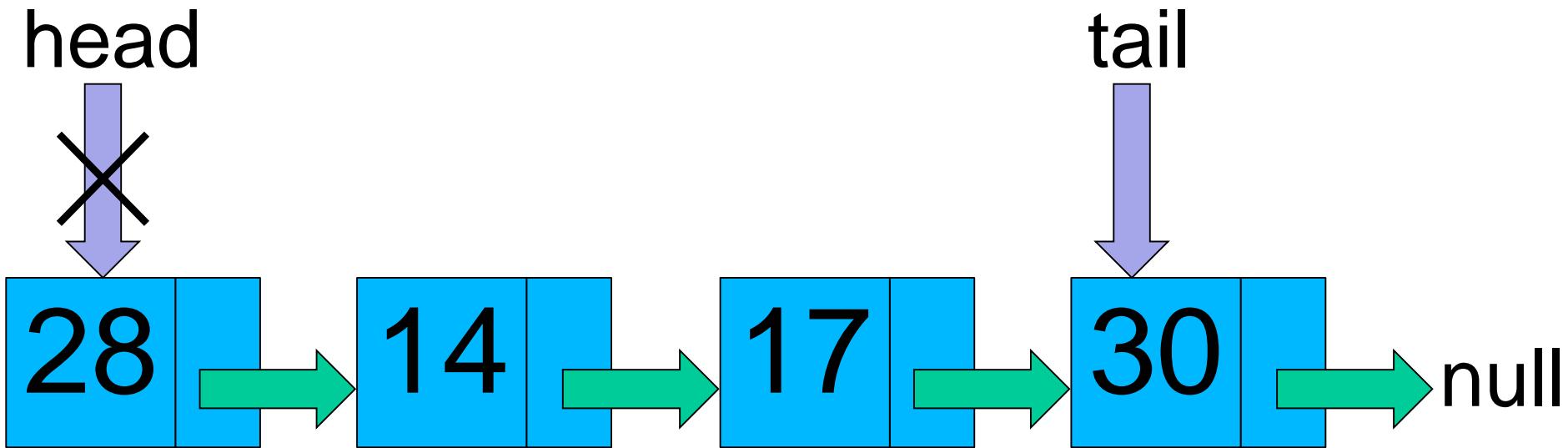
LinkedList: Remove from the front

head

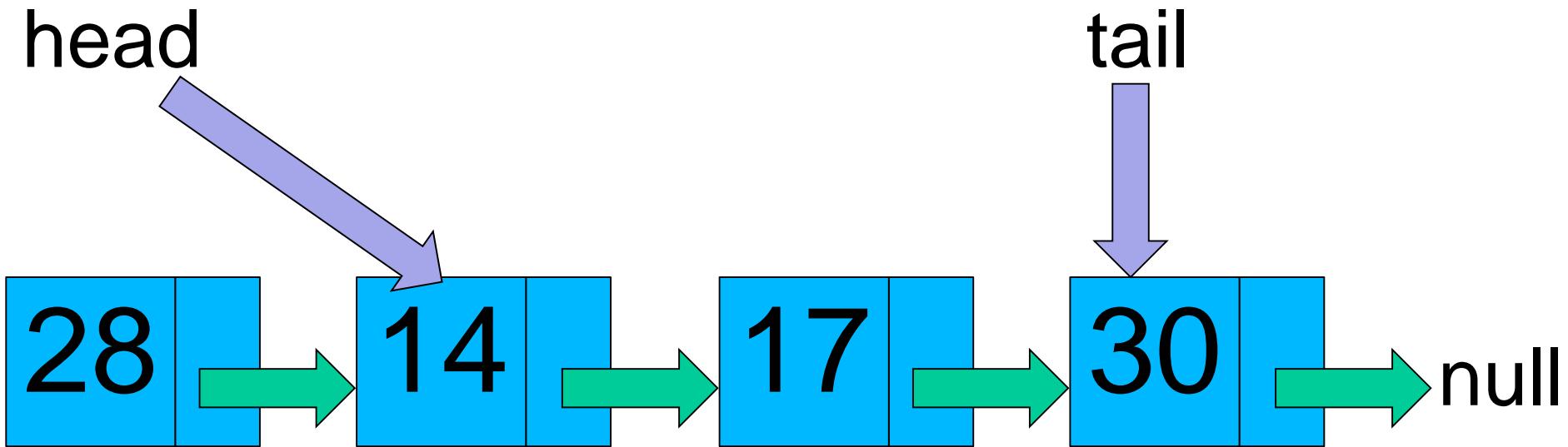
tail



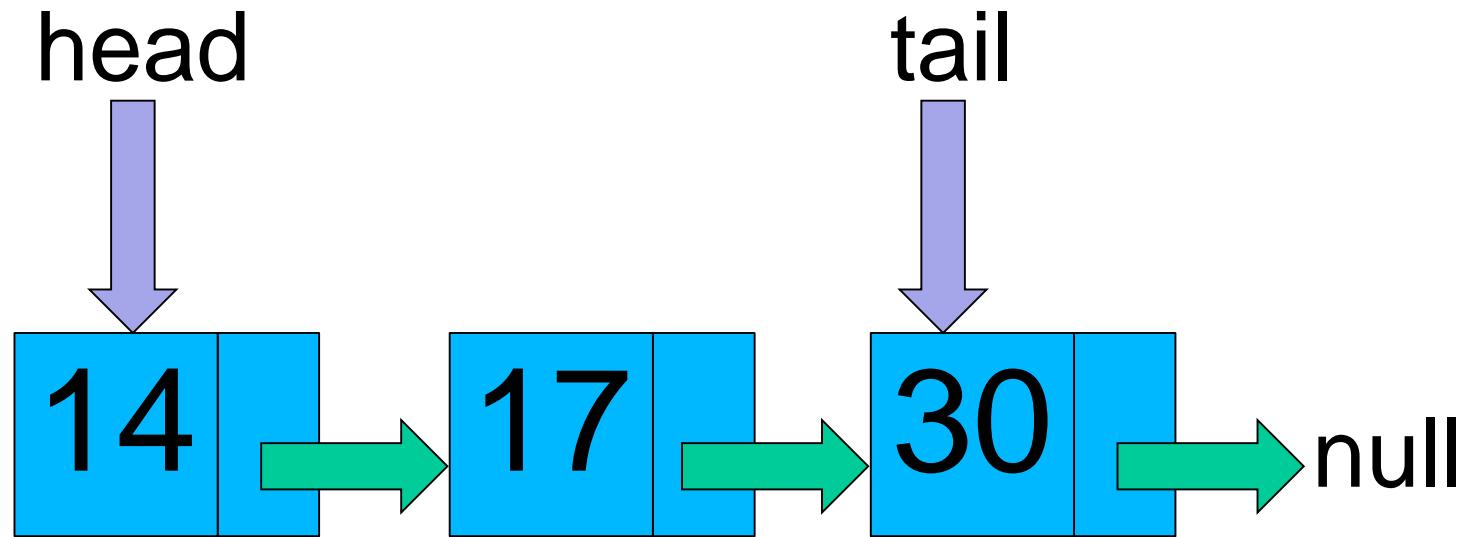
LinkedList: Remove from the front



LinkedList: Remove from the front



LinkedList: Remove from the front



LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        head = head.next;  
    }  
}
```

LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        head = head.next;  
    }  
}
```

LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        head = head.next;  
    }  
}
```

LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        if (head != null) {  
            head = head.next;  
        }  
        if (head == null) {  
            tail = null;  
        }  
    }  
}
```

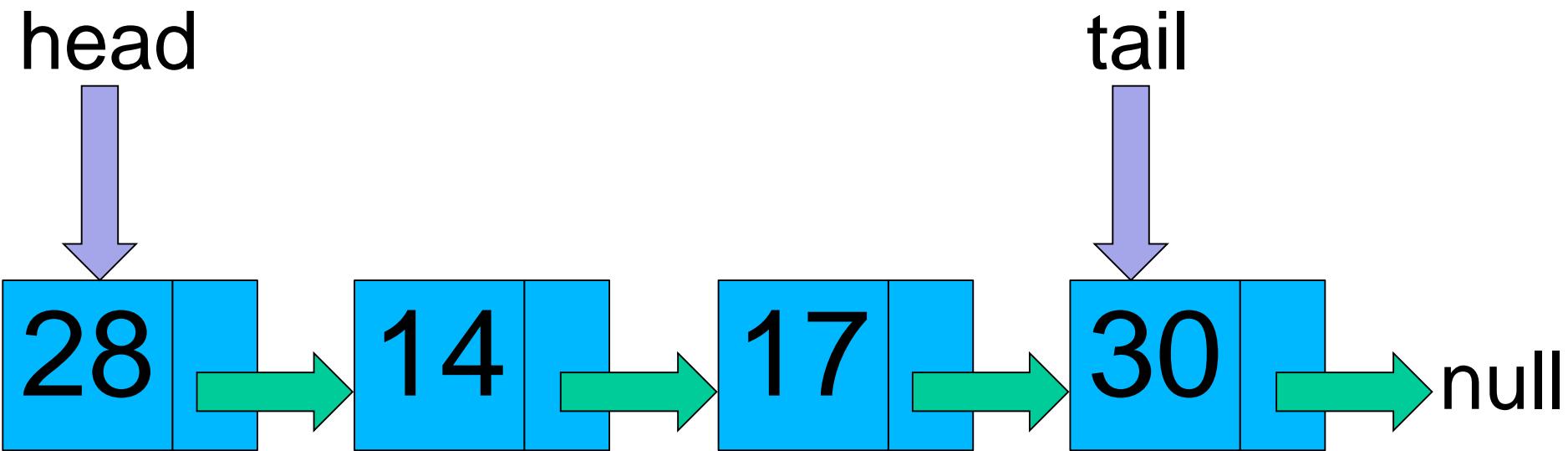
LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        if (head != null) {  
            head = head.next;  
        }  
        if (head == null) {  
            tail = null;  
        }  
    }  
}
```

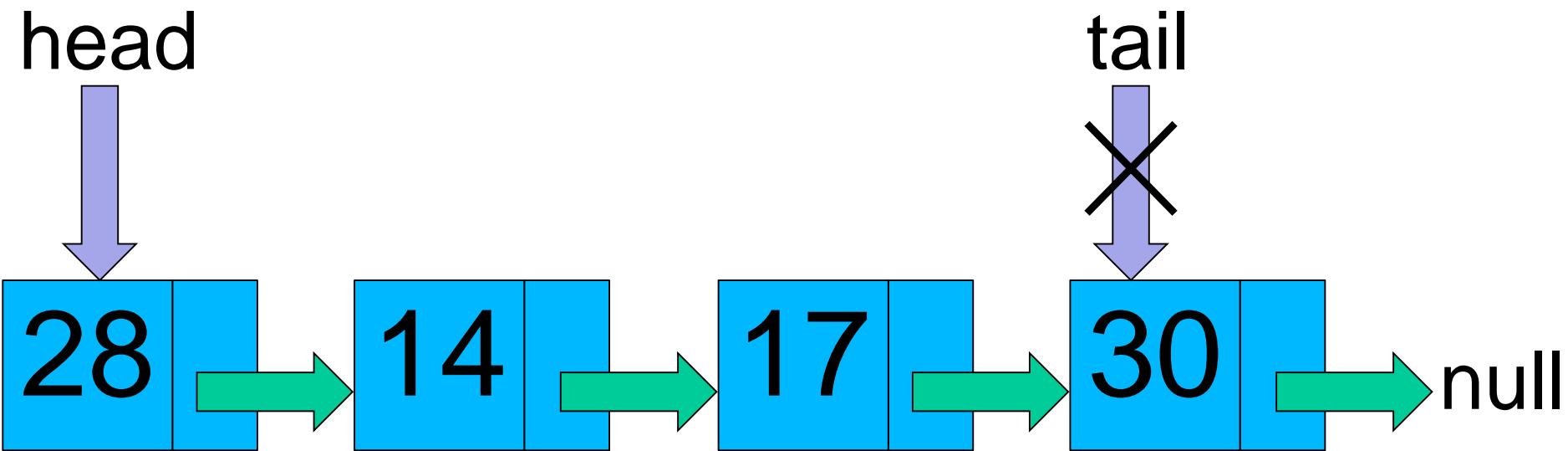
LinkedList: Remove from front

```
public class LinkedList {  
    . . .  
    protected Node head = null;  
  
    public void removeFromFront() {  
        if (head != null) {  
            head = head.next;  
        }  
        if (head == null) {  
            tail = null;  
        }  
    }  
}
```

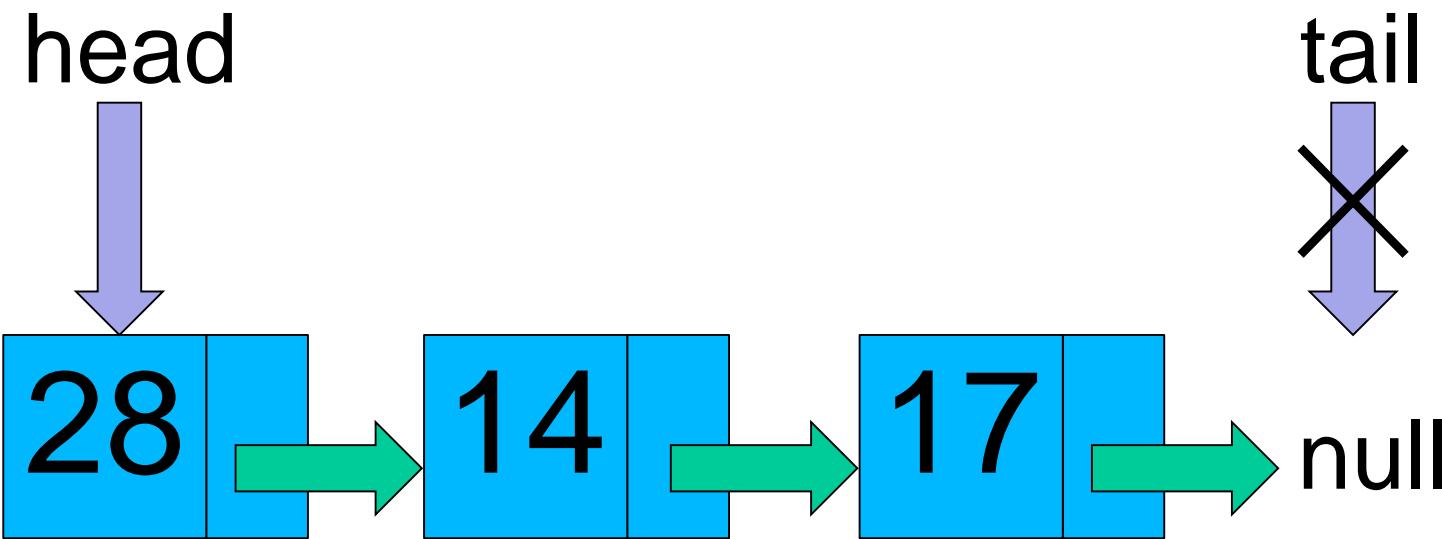
LinkedList: Remove from back



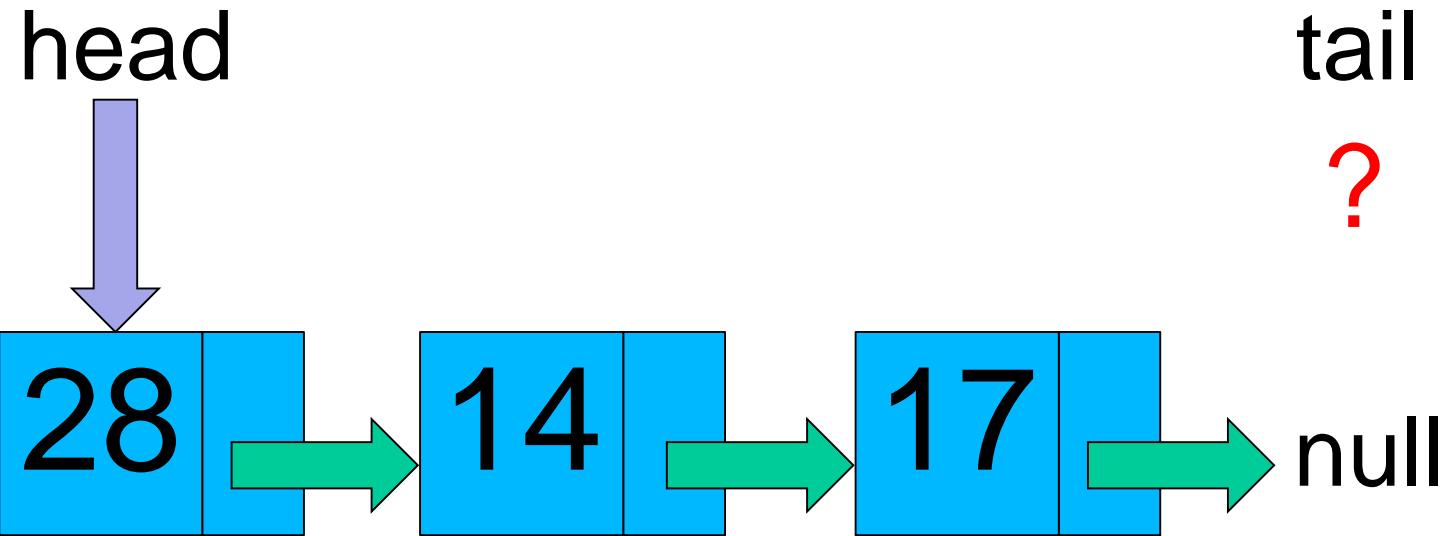
LinkedList: Remove from back



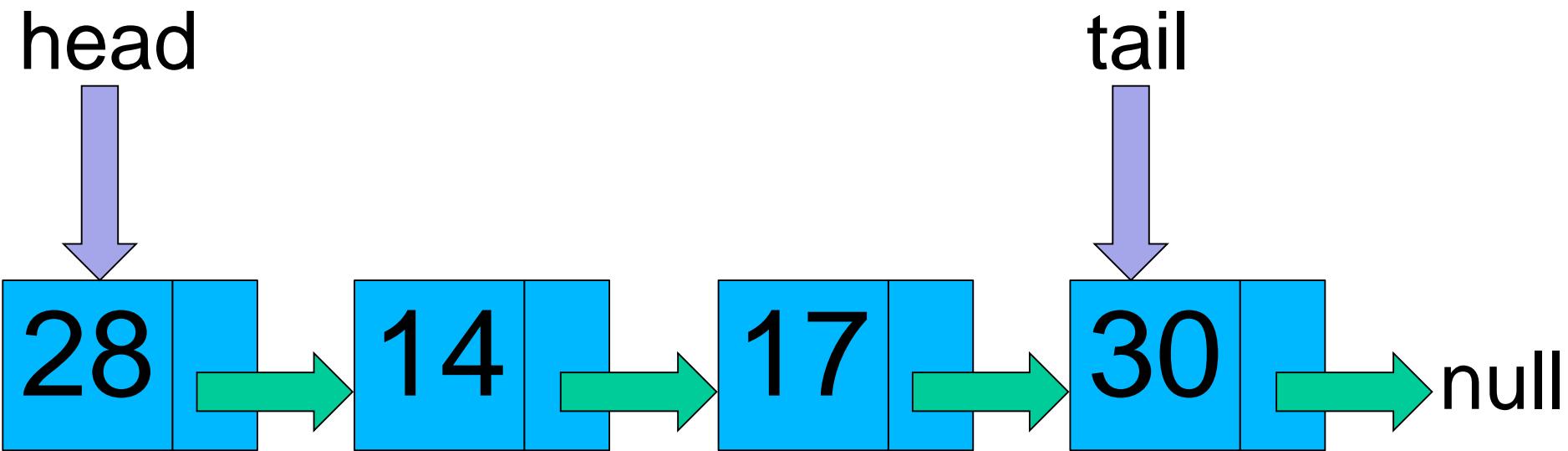
LinkedList: Remove from back



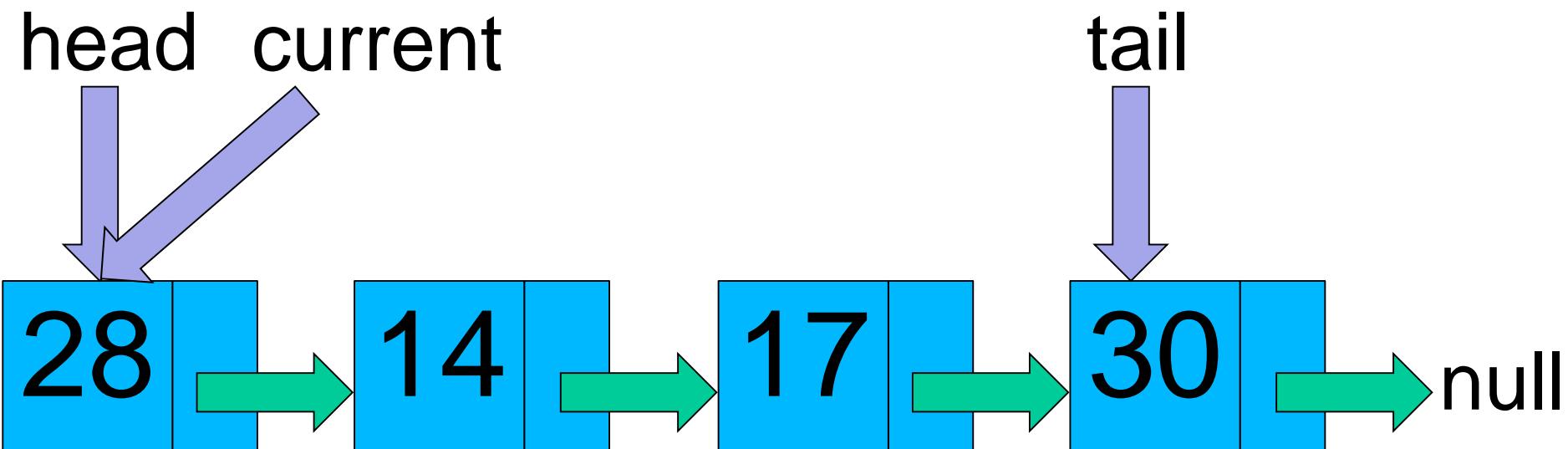
LinkedList: Remove from back



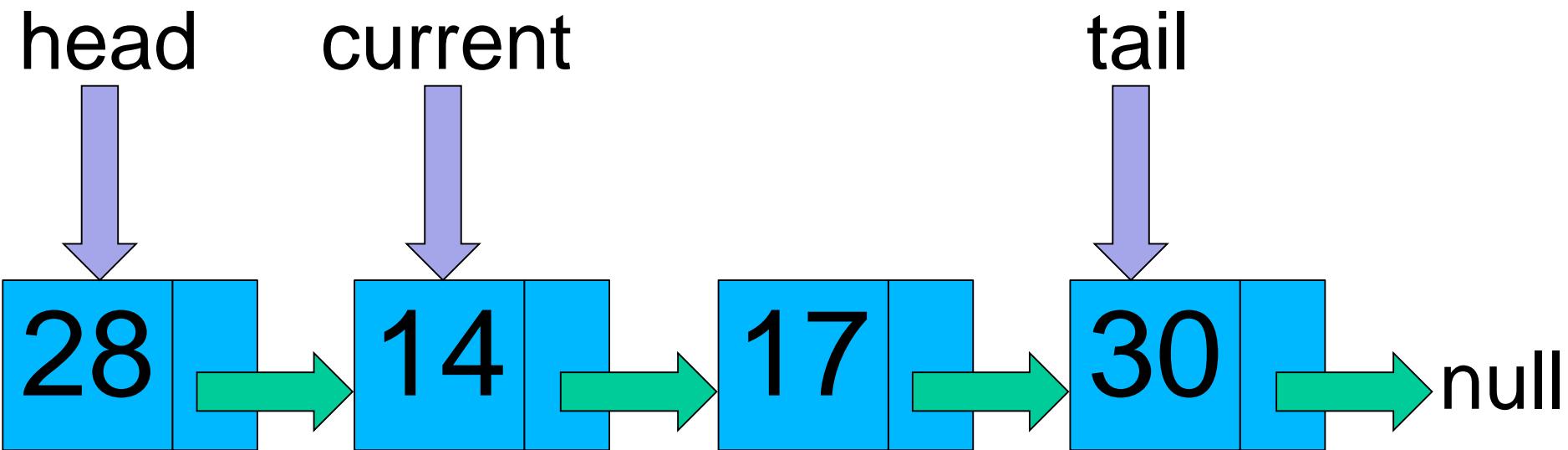
LinkedList: Remove from back



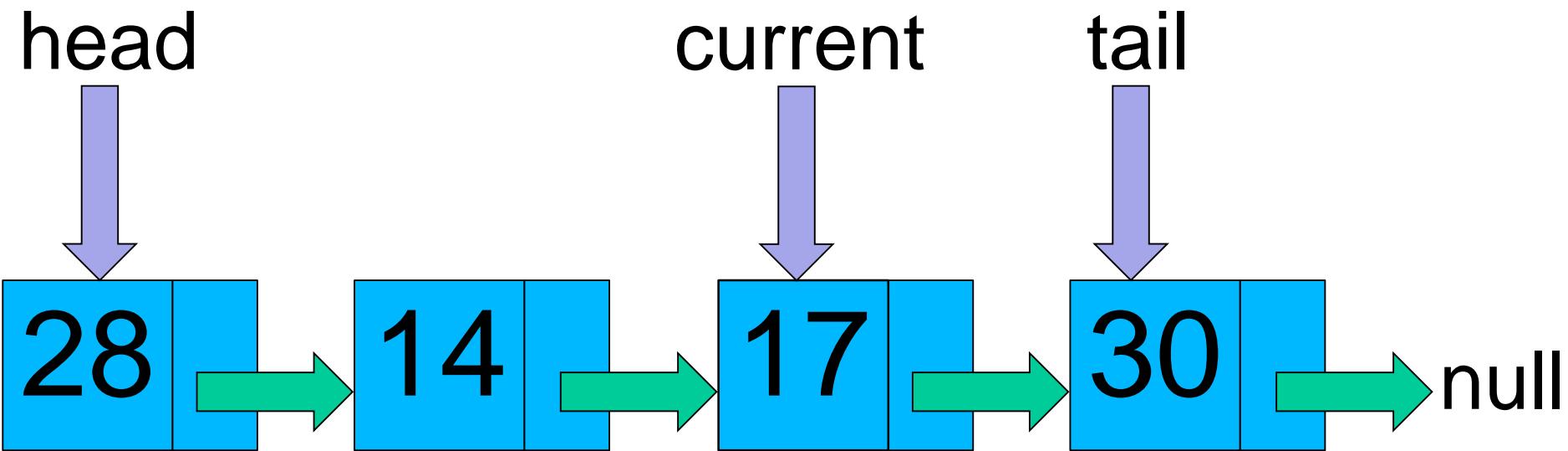
LinkedList: Remove from back



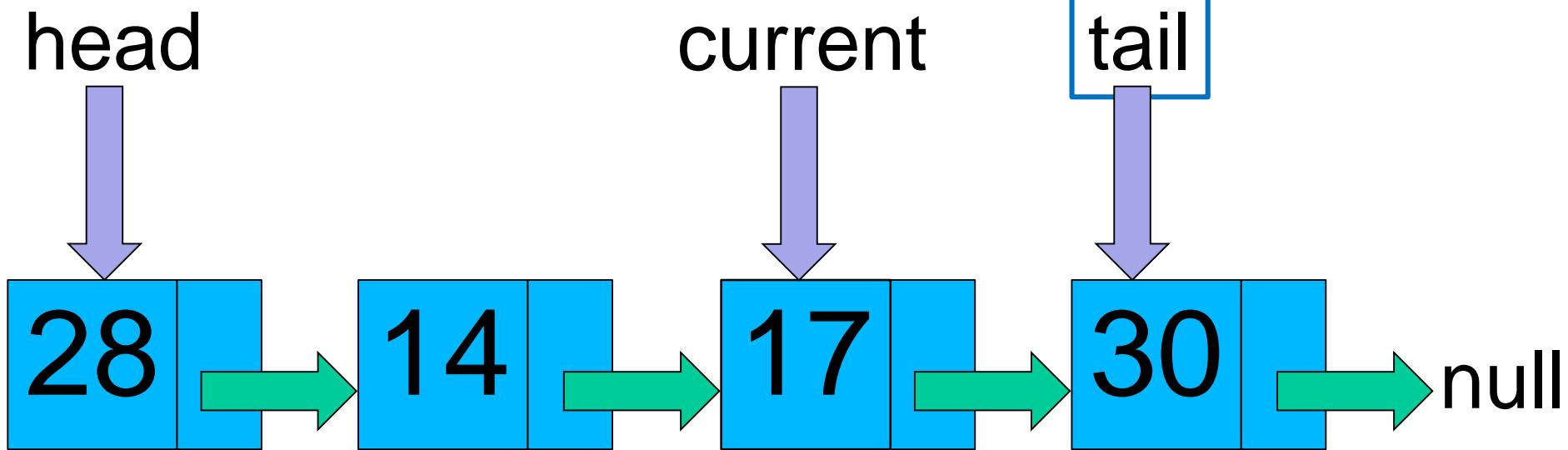
LinkedList: Remove from back



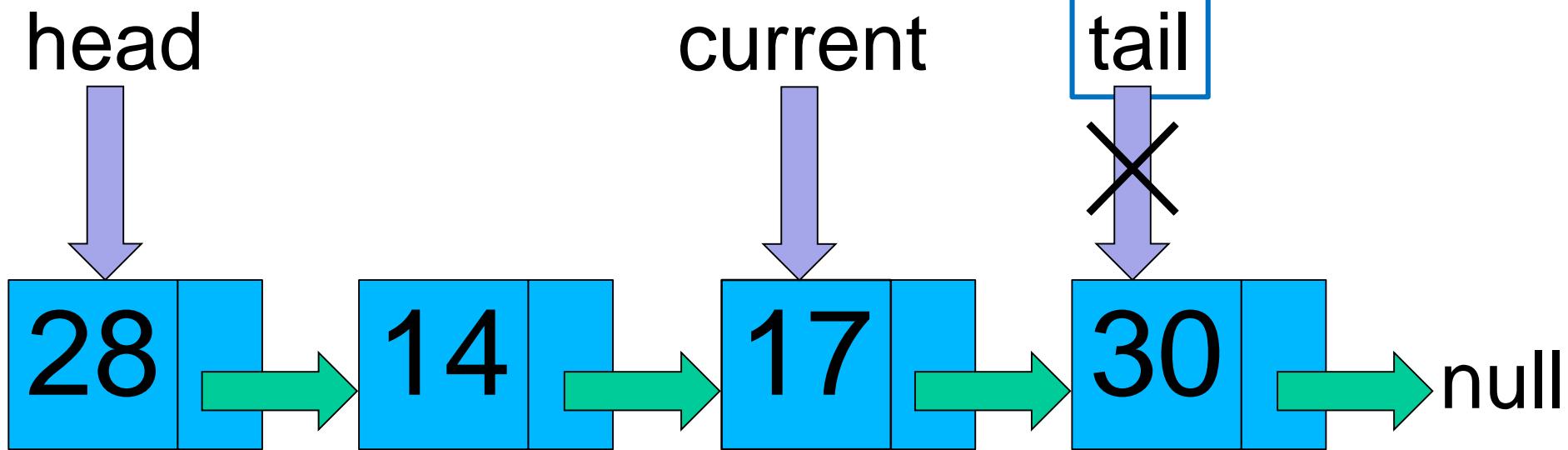
LinkedList: Remove from back



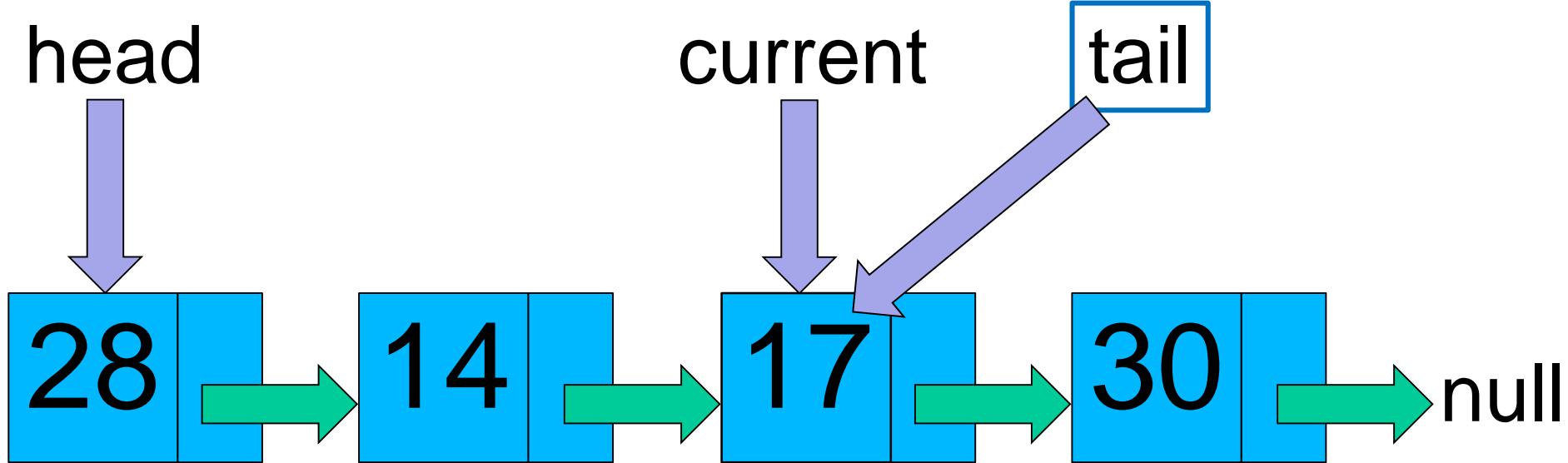
LinkedList: Remove from back



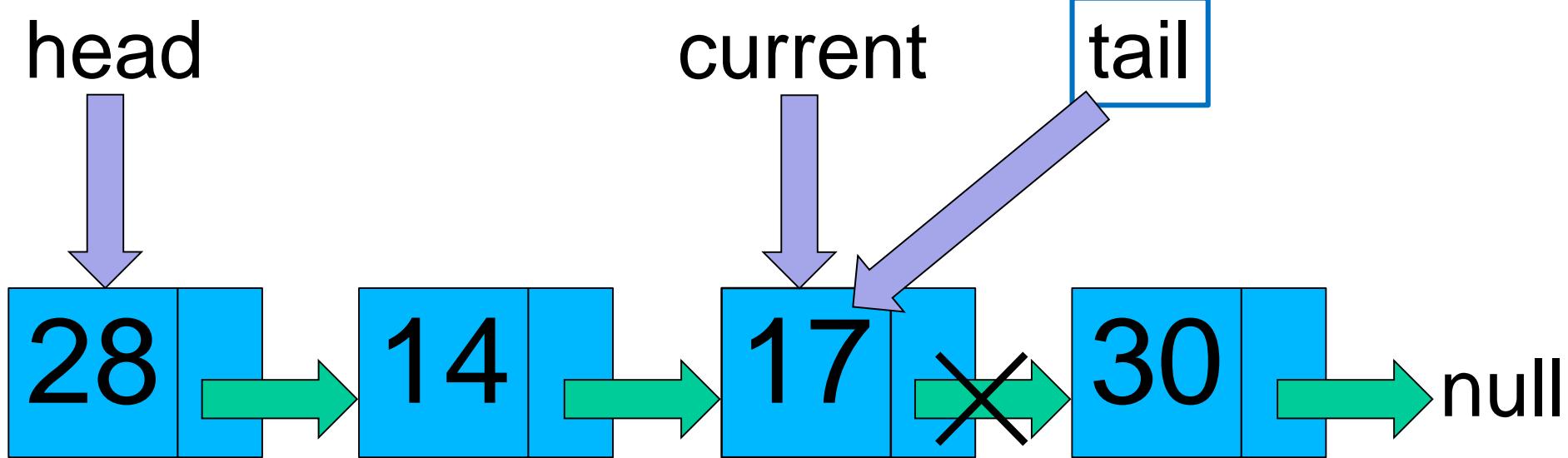
LinkedList: Remove from back



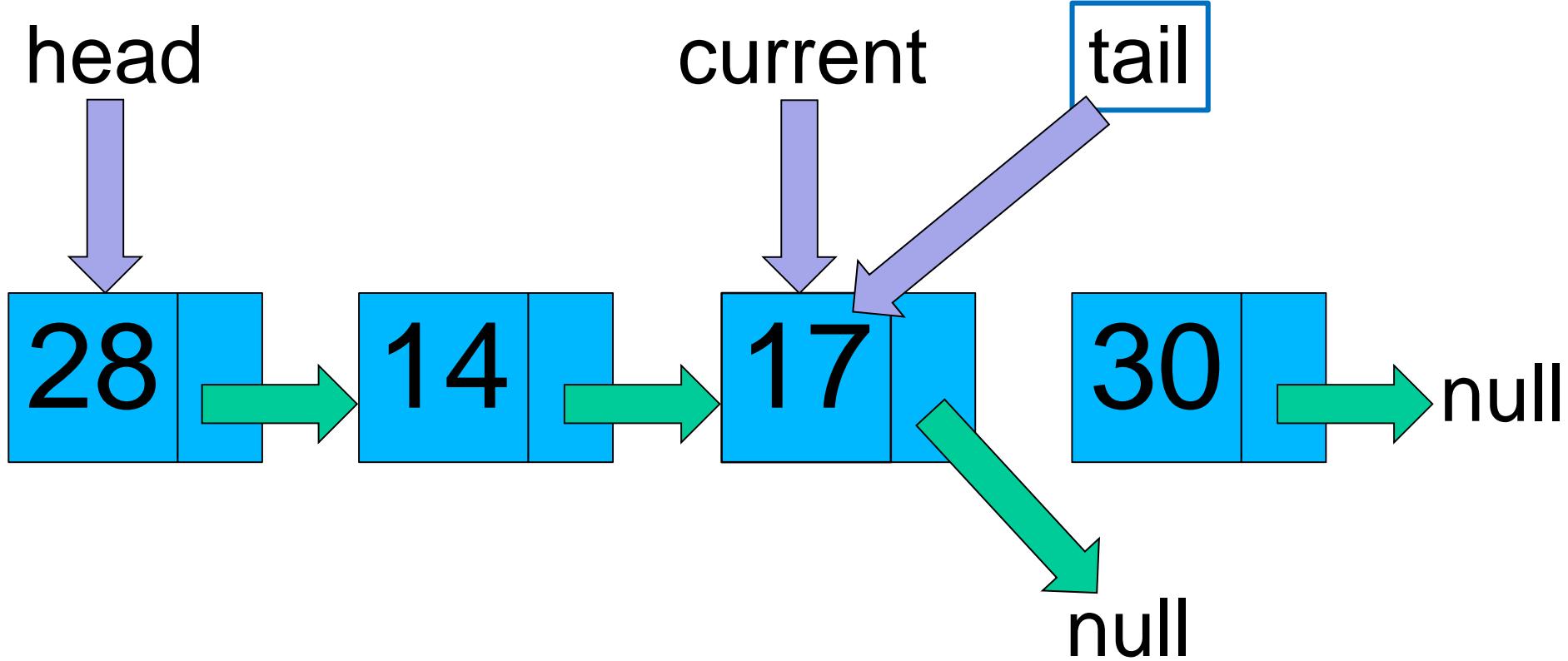
LinkedList: Remove from back



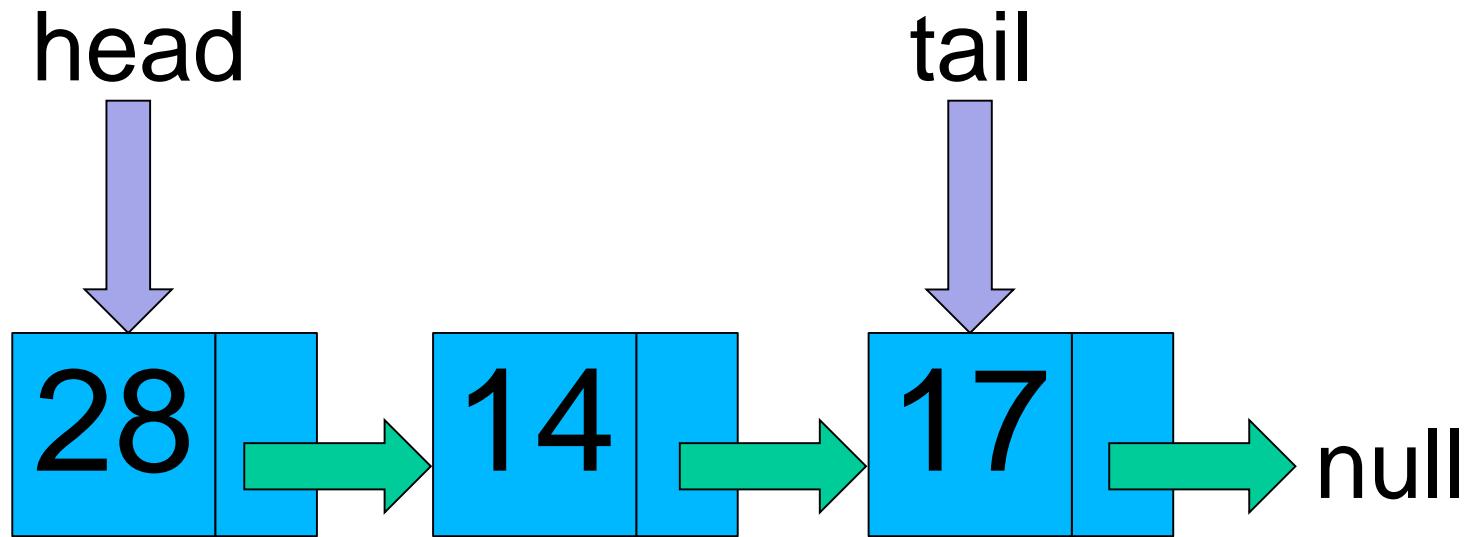
LinkedList: Remove from back



LinkedList: Remove from back



LinkedList: Remove from back



LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        Node current = head;  
        while (current.next != tail) {  
            current = current.next;  
        }  
        tail = current;  
        current.next = null;  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        if (head == null) { // empty list  
            return;  
        } else if (head.equals(tail)) { // single element list  
            head = null;  
            tail = null;  
        } else {  
            Node current = head;  
            while (current.next != tail) {  
                current = current.next;  
            }  
            tail = current;  
            current.next = null;  
        }  
    }  
}
```

LinkedList: Remove from back

```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        if (head == null) { // empty list  
            return;  
        } else if (head.equals(tail)) { // single element list  
            head = null;  
            tail = null;  
        } else {  
            Node current = head;  
            while (current.next != tail) {  
                current = current.next;  
            }  
            tail = current;  
            current.next = null;  
        }  
    }  
}
```

LinkedList: Remove from back

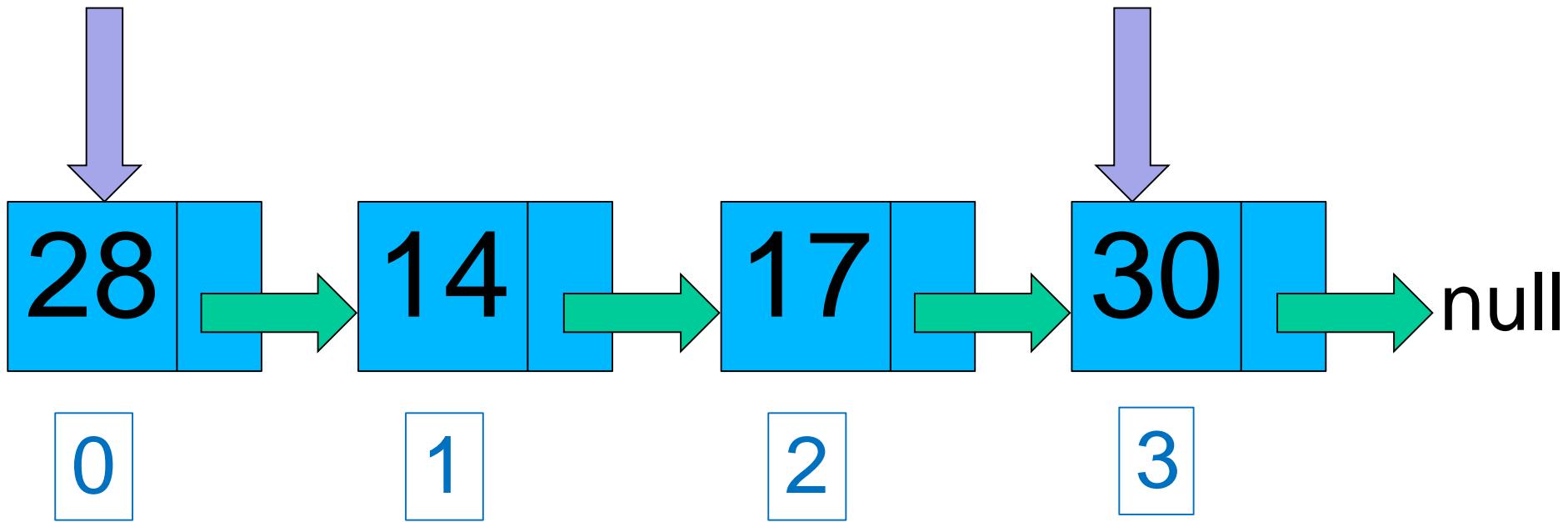
```
public class LinkedList {  
    . . .  
    protected Node tail = null;  
  
    public void removeFromBack() {  
        if (head == null) { // empty list  
            return;  
        } else if (head.equals(tail)) { // single element list  
            head = null;  
            tail = null;  
        } else {  
            Node current = head;  
            while (current.next != tail) {  
                current = current.next;  
            }  
            tail = current;  
            current.next = null;  
        }  
    }  
}
```

LinkedList: Remove at index

index = 2

head

tail

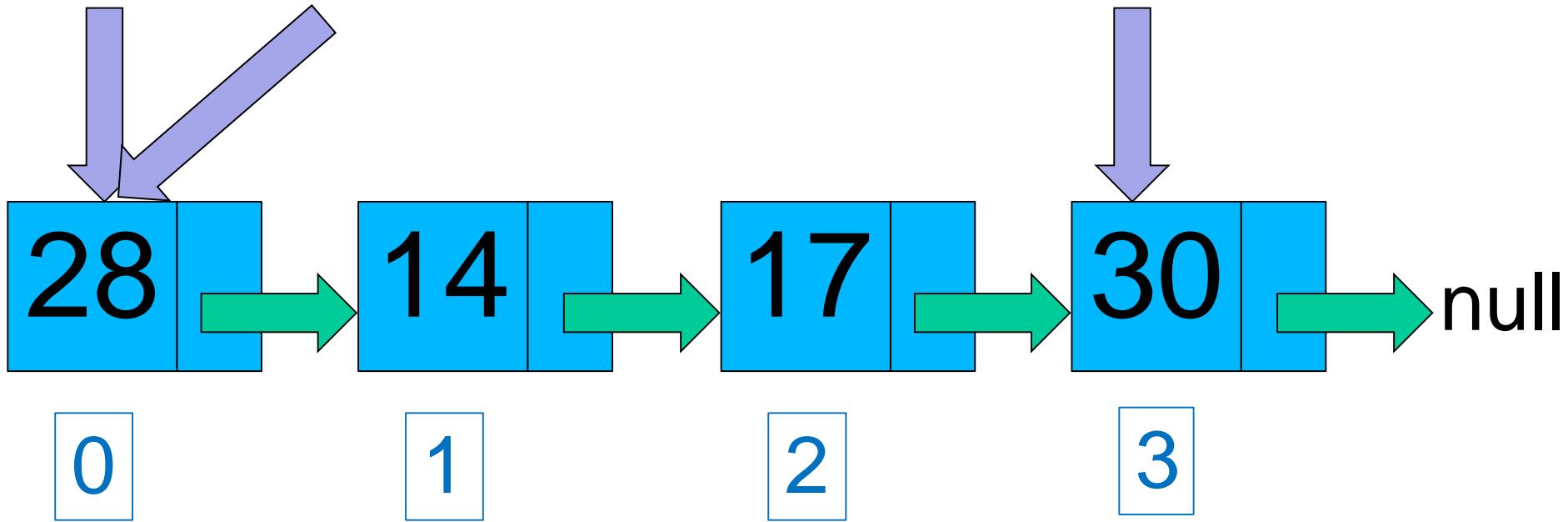


LinkedList: Remove at index

index = 2

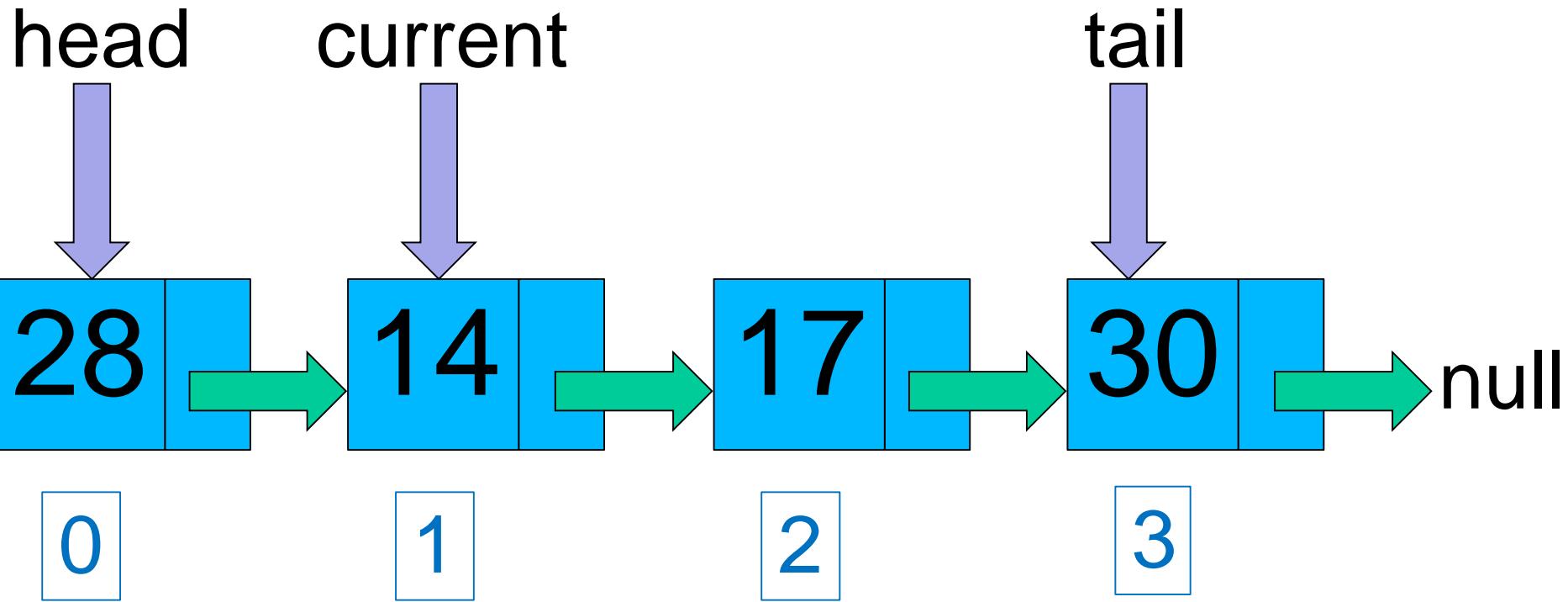
head current

tail



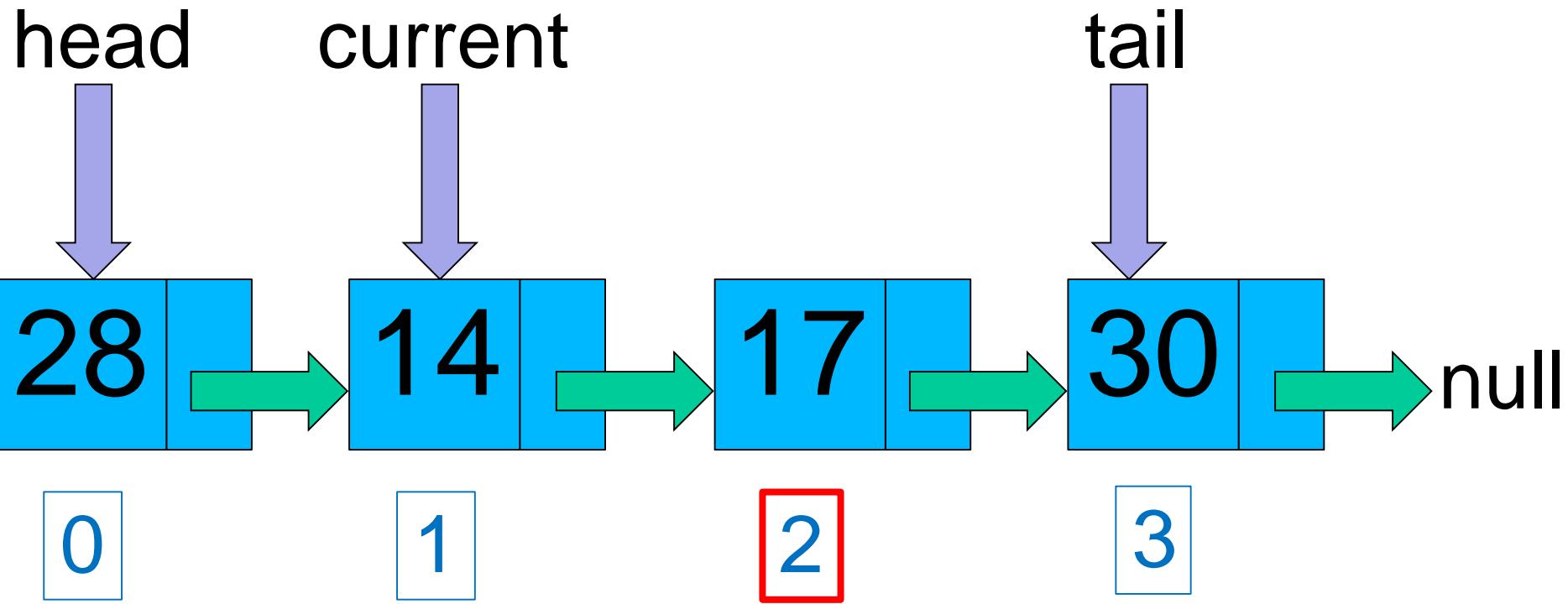
LinkedList: Remove at index

index = 2



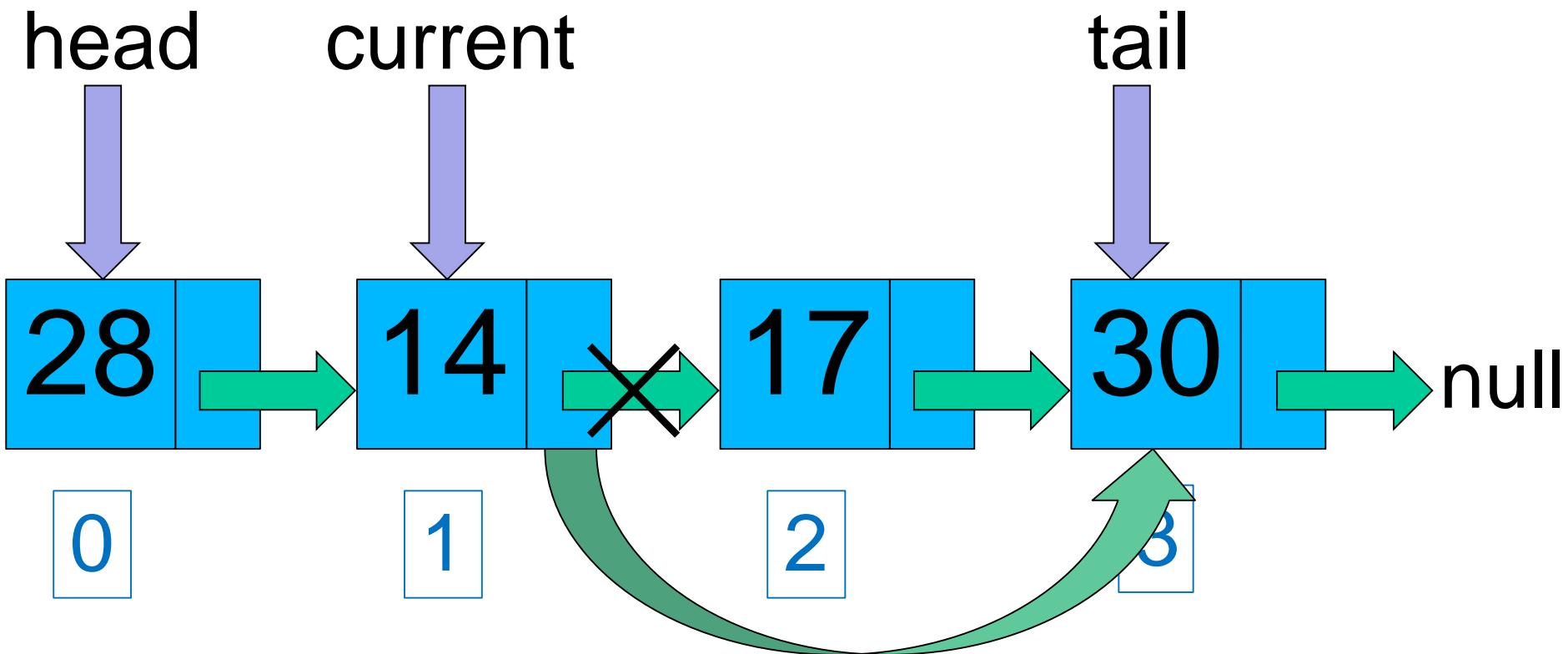
LinkedList: Remove at index

index = 2



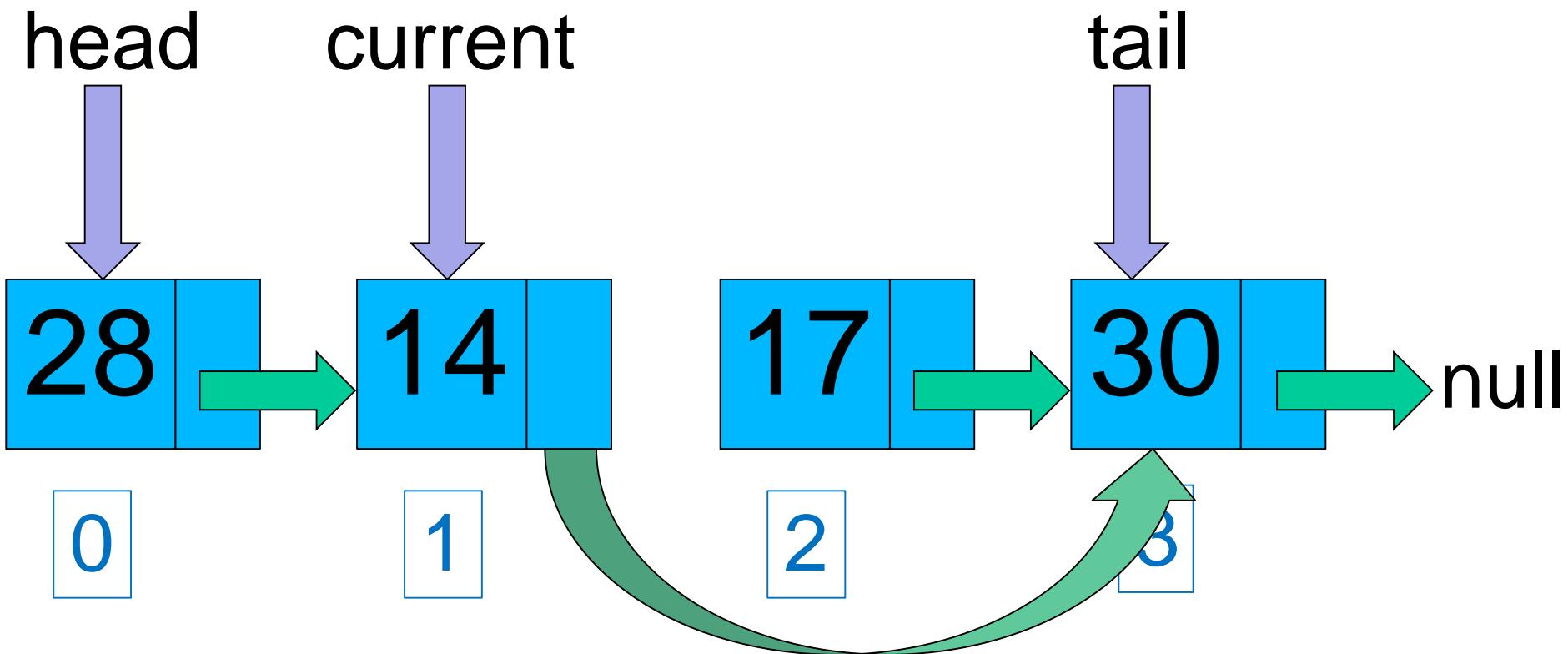
LinkedList: Remove at index

index = 2

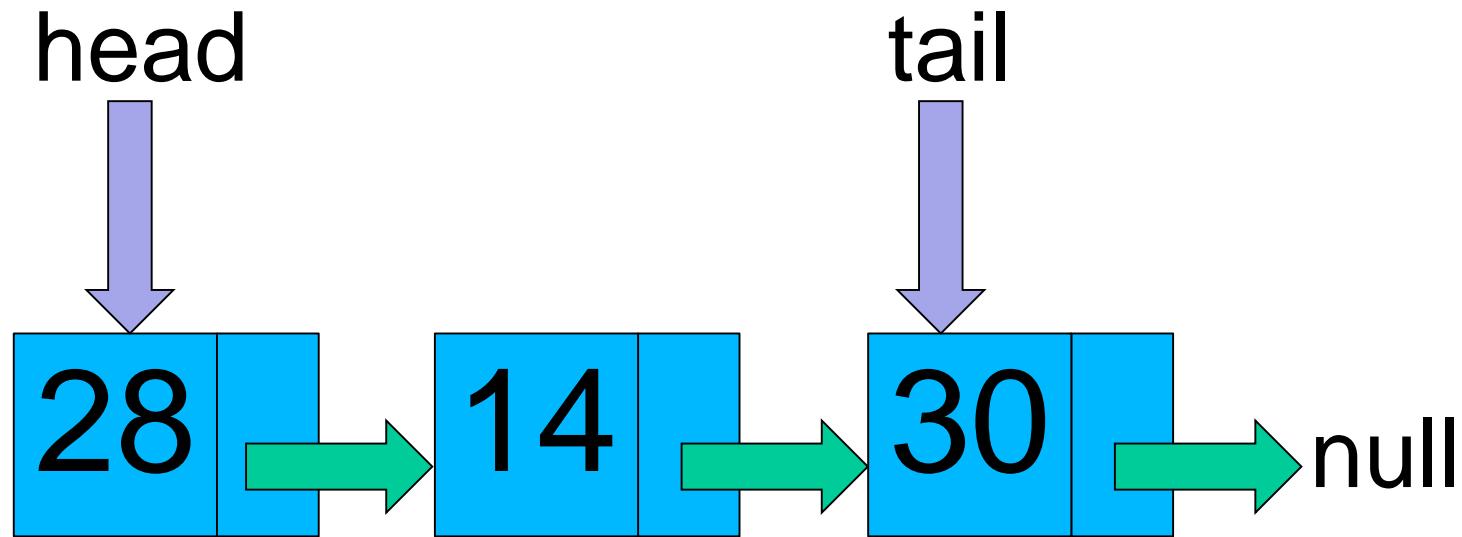


Removing at index

index = 2



LinkedList: Remove at index



LinkedList: Remove at index

```
public class LinkedList {  
    . . .  
    public void removeAtIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = current.next.next;  
    }  
}
```

LinkedList: Remove at index

```
public class LinkedList {  
    . . .  
public void removeAtIndex(int index) {  
    Node current = head;  
    for (int i = 0; i < index - 1; i++) {  
        current = current.next;  
    }  
    current.next = current.next.next;  
}  
}
```

LinkedList: Remove at index

```
public class LinkedList {  
    . . .  
    public void removeAtIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = current.next.next;  
    }  
}
```

LinkedList: Remove at index

```
public class LinkedList {  
    . . .  
    public void removeAtIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = current.next.next;  
    }  
}
```

LinkedList: Remove at index

```
public class LinkedList {  
    . . .  
    public void removeAtIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            current = current.next;  
        }  
        current.next = current.next.next;  
    }  
}
```

LinkedList: remove at index

```
public void removeAtIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) {  
        removeFromFront();  
    } else {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        current.next = current.next.next;  
        if (current.next == null) {  
            tail = current;  
        }  
    }  
}
```

LinkedList: remove at index

```
public void removeAtIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) {  
        removeFromFront();  
    } else {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        current.next = current.next.next;  
        if (current.next == null) {  
            tail = current;  
        }  
    }  
}
```

LinkedList: remove at index

```
public void removeAtIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) {  
        removeFromFront();  
    } else {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        current.next = current.next.next;  
        if (current.next == null) {  
            tail = current;  
        }  
    }  
}
```

LinkedList: remove at index

```
public void removeAtIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) {  
        removeFromFront();  
    } else {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        current.next = current.next.next;  
        if (current.next == null) {  
            tail = current;  
        }  
    }  
}
```

LinkedList: remove at index

```
public void removeAtIndex(int index) {  
    if (index < 0) {  
        throw new IndexOutOfBoundsException();  
    } else if (index == 0) {  
        removeFromFront();  
    } else {  
        Node current = head;  
        for (int i = 0; i < index - 1; i++) {  
            if (current == null) {  
                throw new IndexOutOfBoundsException();  
            }  
            current = current.next;  
        }  
        current.next = current.next.next;  
        if (current.next == null) {  
            tail = current;  
        }  
    }  
}
```

Review: Linked Lists

- Series of “Nodes” with each one linked to the next
- Can insert a value by creating a new Node and updating the links
- Can retrieve a value by traversing the links

Review: Linked Lists

- Series of “Nodes” with each one linked to the next
- Can insert a value by creating a new Node and updating the links
- Can retrieve a value by traversing the links
- Can easily remove value at front or back of Linked List, or at a given index

SD2x1.5

LL in Java; ArrayList

Kathy

How are `LinkedLists` implemented in the Java API?

LinkedList in Java API (1)

- **java.util.LinkedList<E>**
- **add(Element):** adds element to end (tail) of list; same as addLast
- **add(index,Element):** adds element at specified index in list
- **addFirst:** adds element to front (head) of list
- **set(index,Element):** replaces element at specified index with specified element

LinkedList in Java API (2)

- **java.util.LinkedList<E>**
- **contains**: indicates whether list contains element
- **get**: returns element at specific index
- **getFirst**: returns element at front of list
- **getLast**: returns element at end of list

LinkedList in Java API (3)

- **java.util.LinkedList<E>**
- **remove**: returns and removes element at front of list; same as removeFirst
- **remove(index)**: returns and removes element at specified index
- **removeLast**: returns and removes element at end of list

```
import java.util.LinkedList;
. . .
LinkedList<Integer> numbers = new LinkedList<Integer>();
Scanner in = new Scanner(System.in);
int input = 0;

while (true) {
    System.out.print("Enter a number: ");
    input = in.nextInt();
    if (input == 0) {
        break;
    }
    numbers.add(input);
}

System.out.print("Enter another number: ");
input = in.nextInt();
if (numbers.contains(input))
    System.out.println(input + " is in the list");
else
    System.out.println(input + " is not in the list");
```

```
import java.util.LinkedList;
. . .
LinkedList<Integer> numbers = new LinkedList<Integer>();
Scanner in = new Scanner(System.in);
int input = 0;

while (true) {
    System.out.print("Enter a number: ");
    input = in.nextInt();
    if (input == 0) {
        break;
    }
    numbers.add(input);
}

System.out.print("Enter another number: ");
input = in.nextInt();
if (numbers.contains(input))
    System.out.println(input + " is in the list");
else
    System.out.println(input + " is not in the list");
```

```
import java.util.LinkedList;
. . .
LinkedList<Integer> numbers = new LinkedList<Integer>();

Scanner in = new Scanner(System.in);
int input = 0;

while (true) {
    System.out.print("Enter a number: ");
    input = in.nextInt();
    if (input == 0) {
        break;
    }
    numbers.add(input);
}

System.out.print("Enter another number: ");
input = in.nextInt();
if (numbers.contains(input))
    System.out.println(input + " is in the list");
else
    System.out.println(input + " is not in the list");
```

```
import java.util.LinkedList;
. . .
LinkedList<Integer> numbers = new LinkedList<Integer>();
Scanner in = new Scanner(System.in);
int input = 0;

while (true) {
    System.out.print("Enter a number: ");
    input = in.nextInt();
    if (input == 0) {
        break;
    }
    numbers.add(input);
}

System.out.print("Enter another number: ");
input = in.nextInt();
if (numbers.contains(input))
    System.out.println(input + " is in the list");
else
    System.out.println(input + " is not in the list");
```

```
import java.util.LinkedList;
. . .
LinkedList<Integer> numbers = new LinkedList<Integer>();
Scanner in = new Scanner(System.in);
int input = 0;

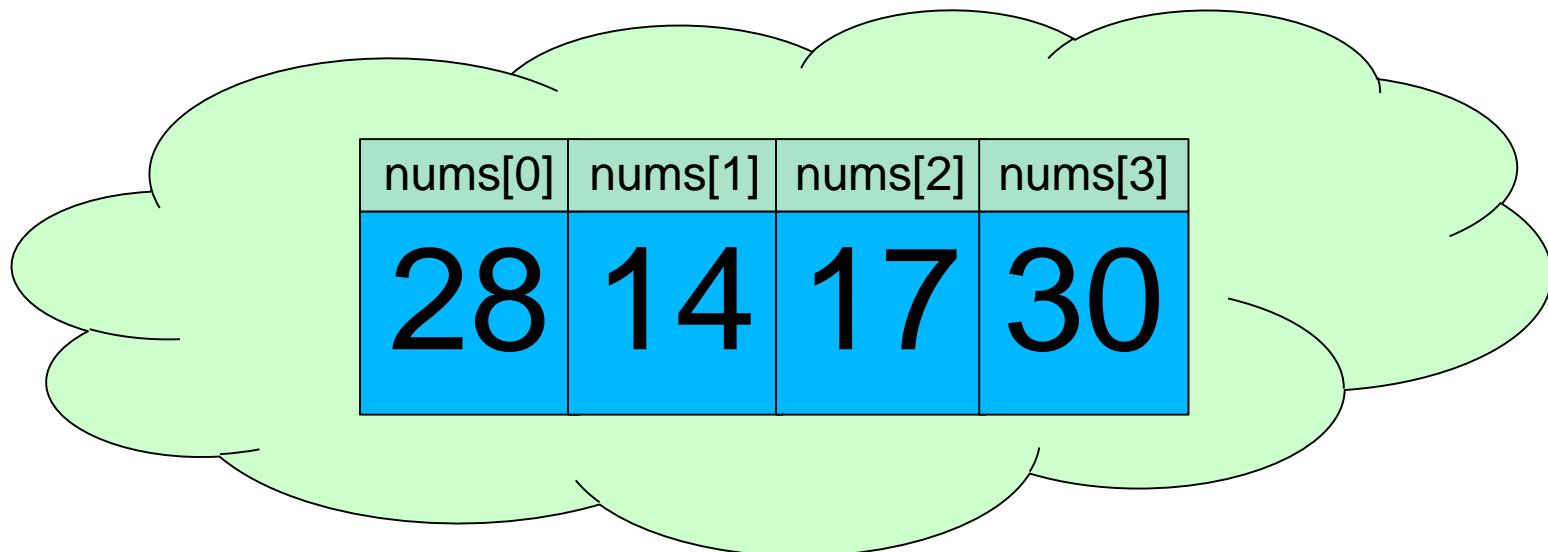
while (true) {
    System.out.print("Enter a number: ");
    input = in.nextInt();
    if (input == 0) {
        break;
    }
    numbers.add(input);
}

System.out.print("Enter another number: ");
input = in.nextInt();
if (numbers.contains(input))
    System.out.println(input + " is in the list");
else
    System.out.println(input + " is not in the list");
```

Are LinkedLists our only option?

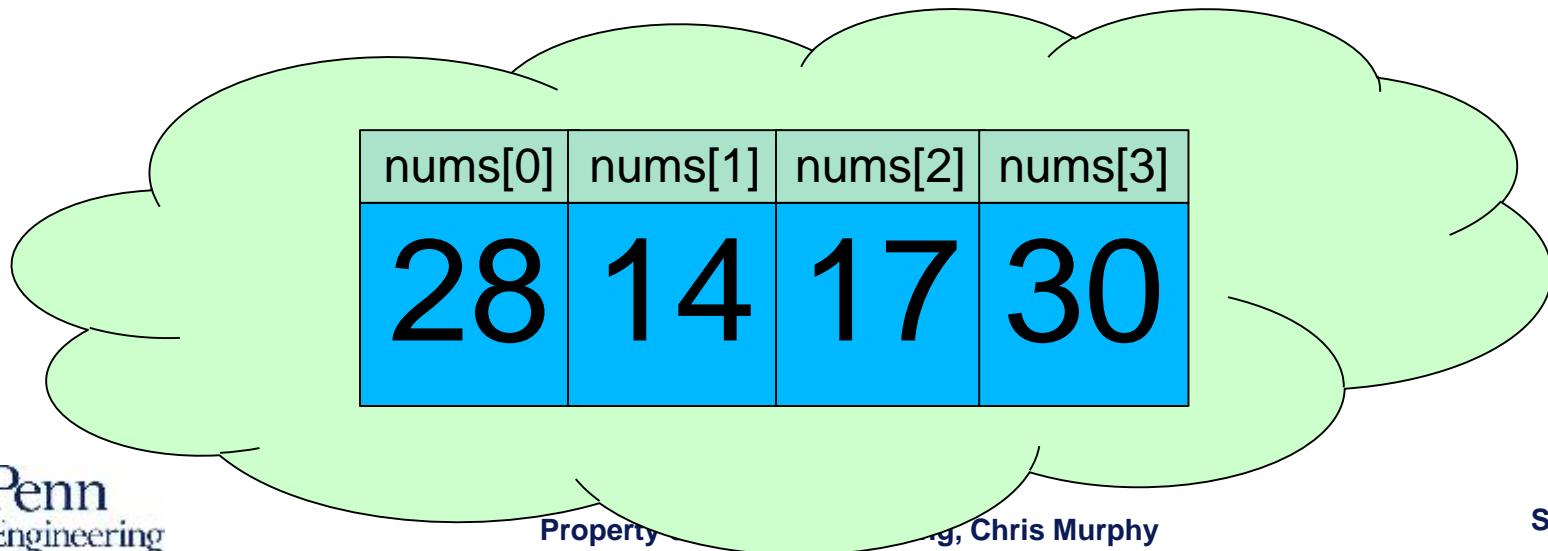
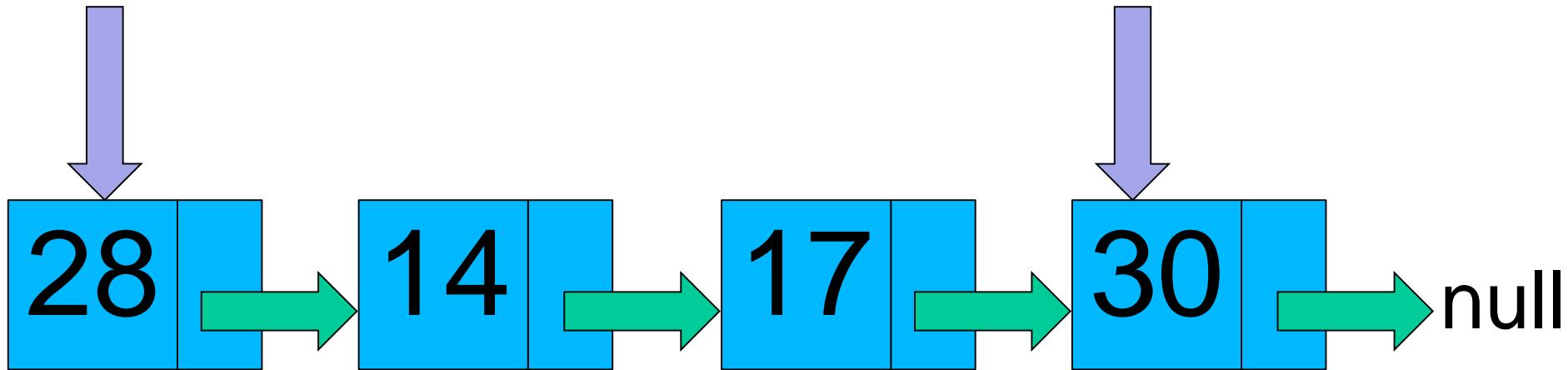
ArrayList

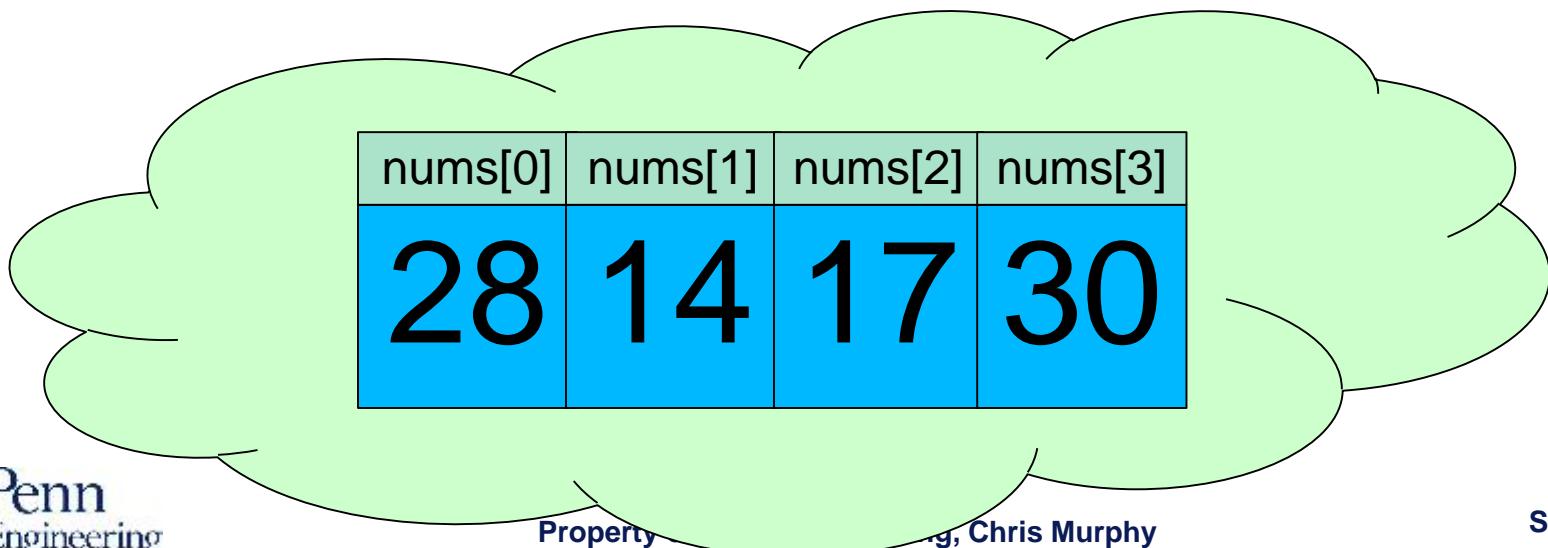
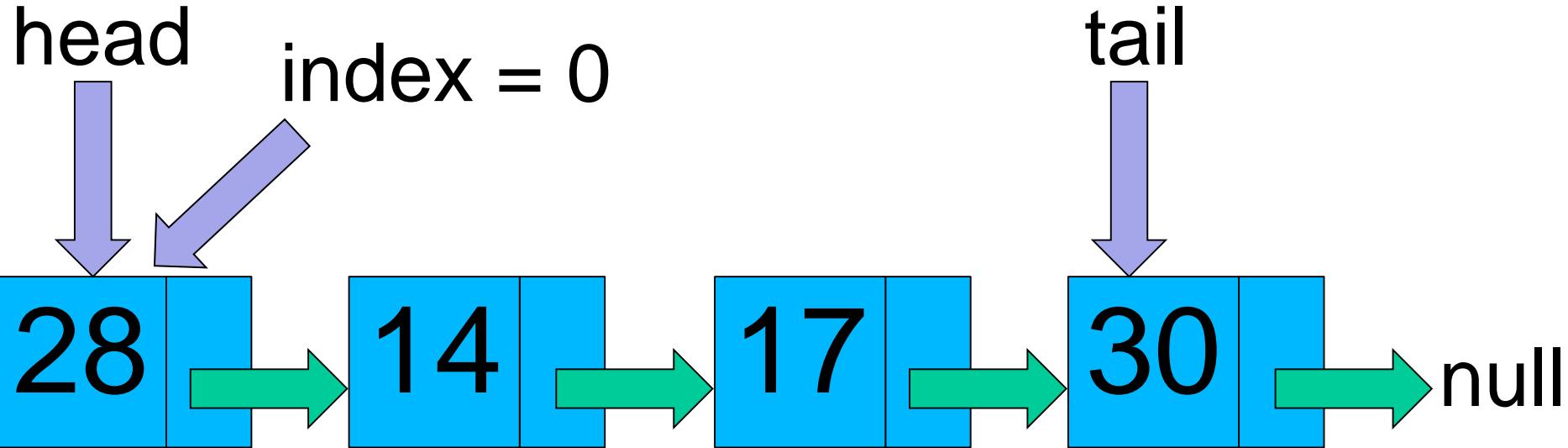
- A data structure that “wraps” an underlying array and provides basic functionality: add, remove, contains, get by index, etc.



head

tail

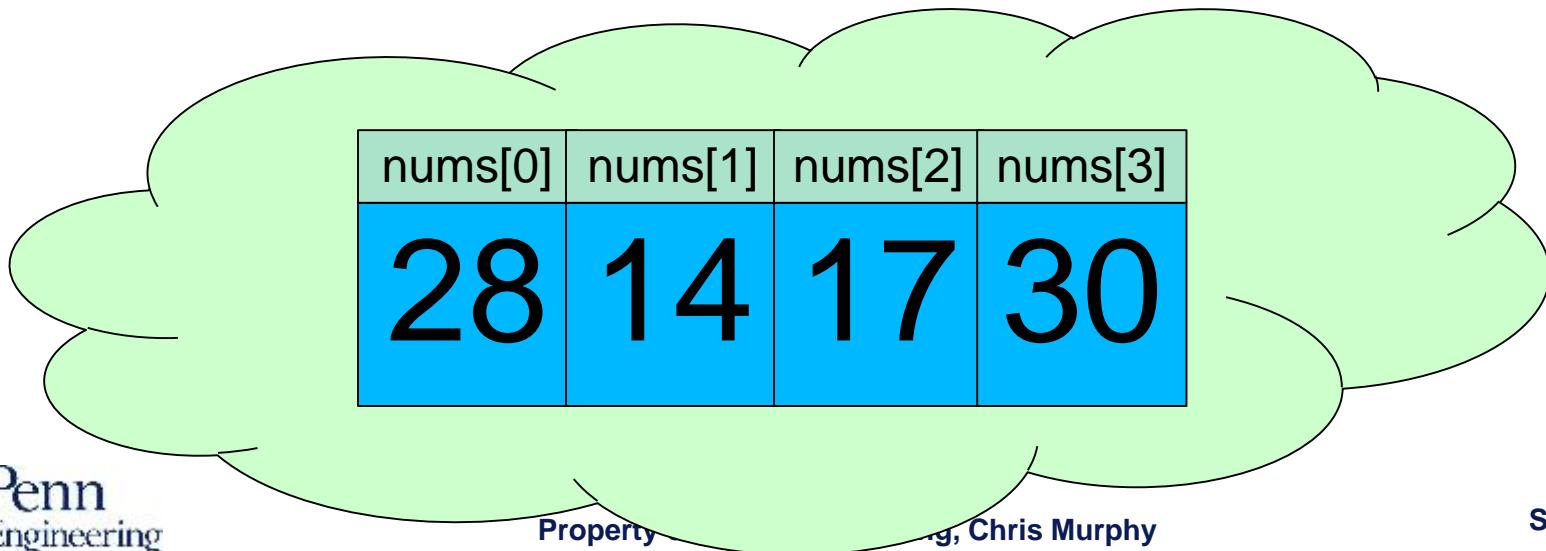
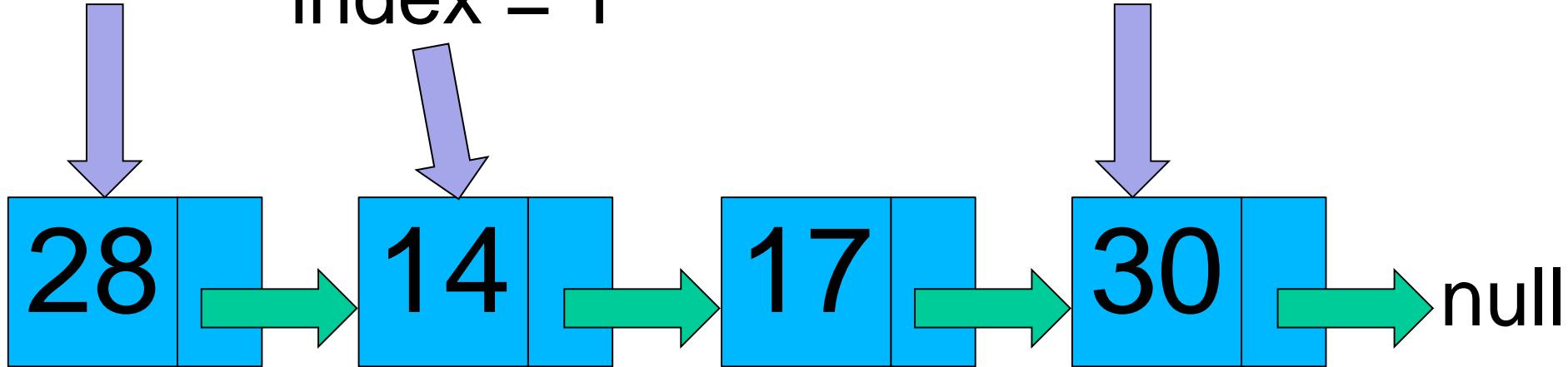


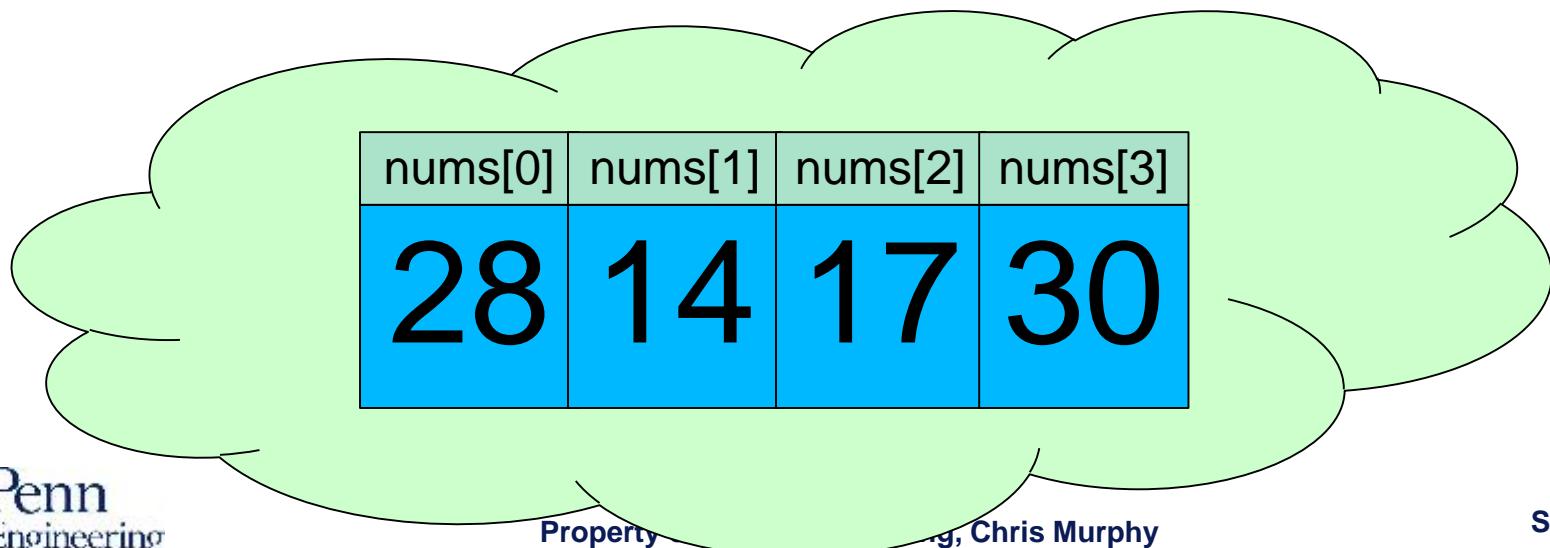
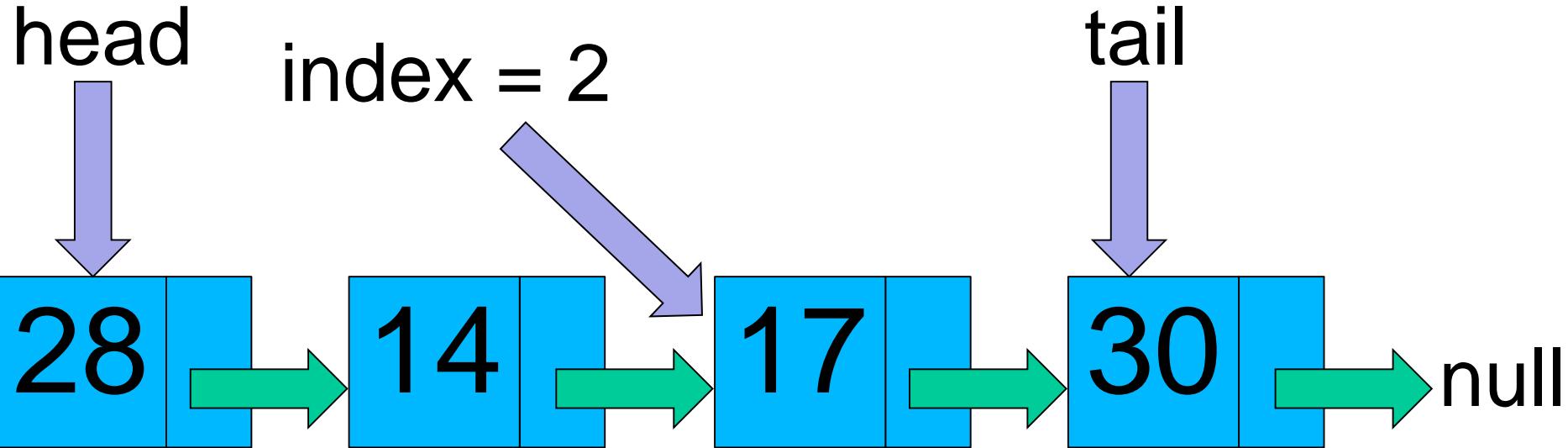


head

index = 1

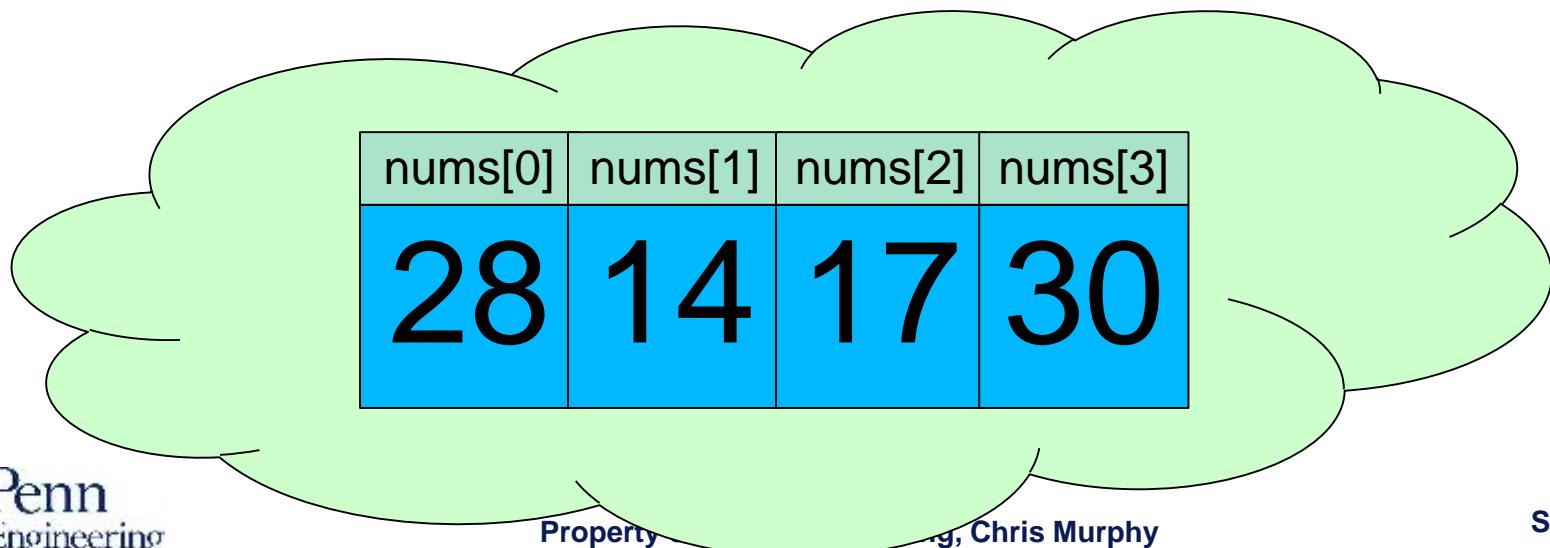
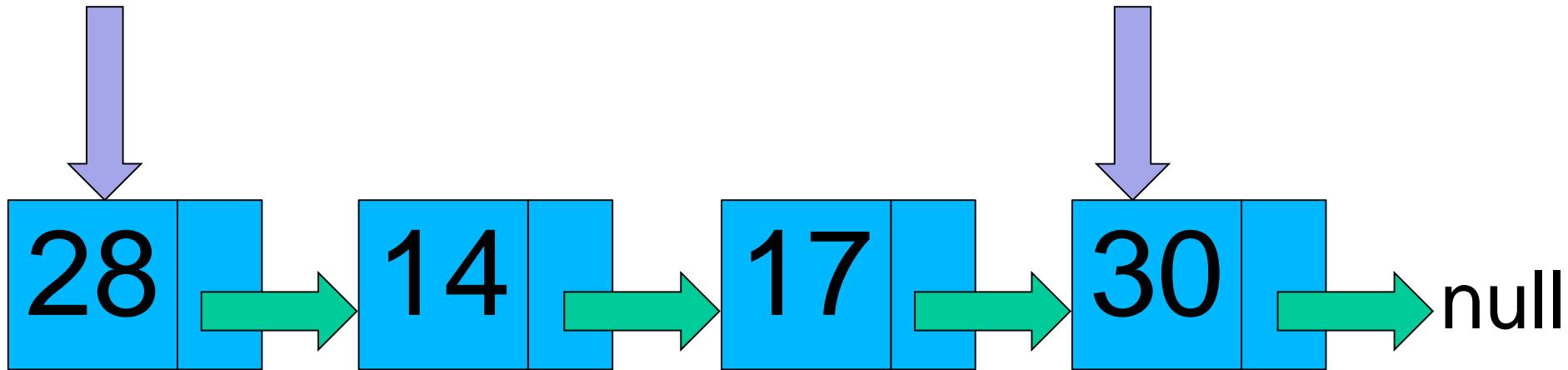
tail





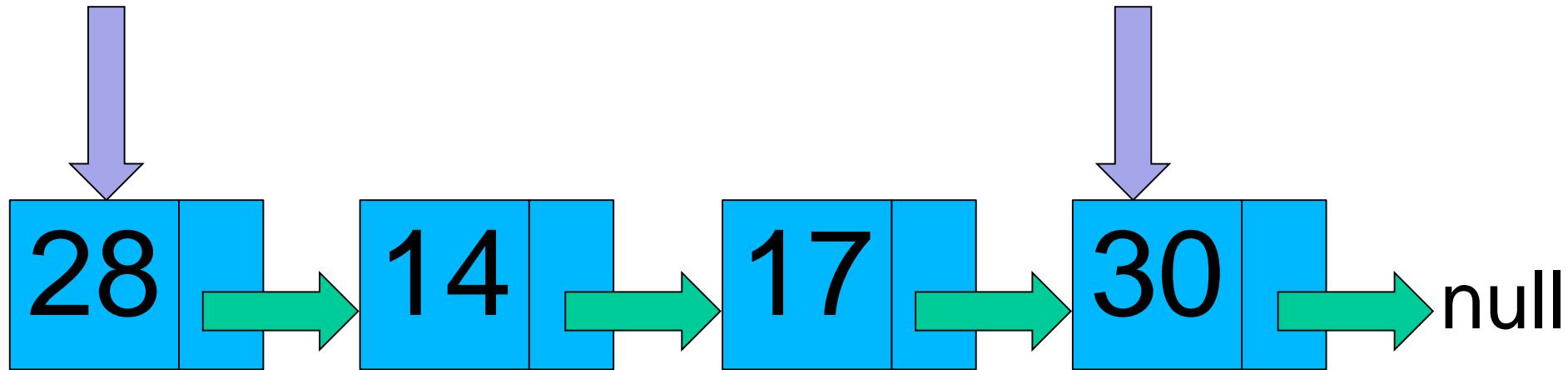
head

tail

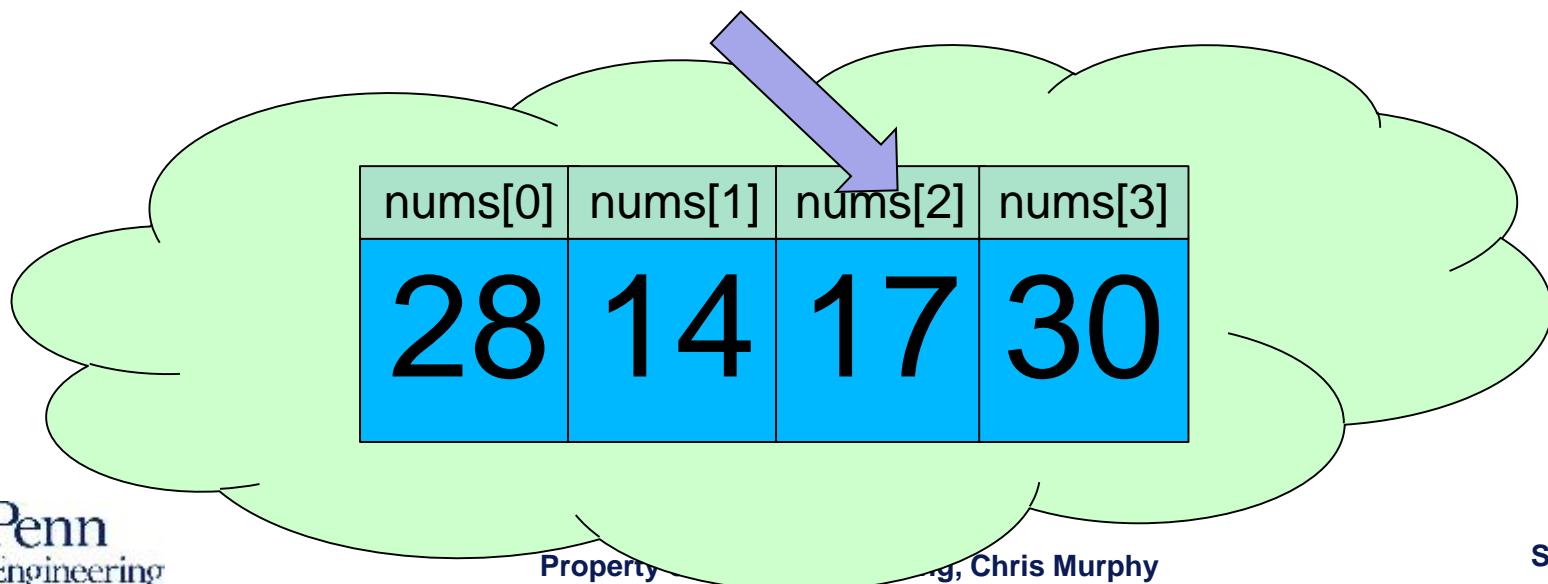


head

tail



index = 2



ArrayList in Java API

- **java.util.ArrayList<E>**
- A class that “wraps” an underlying array and provides basic functionality: **add, remove, contains, get by index, etc.**
- Benefit over LinkedLists: can get an element by its index without needing to step through the list
- Disadvantage vs. LinkedList: potentially unused space; slower to add anywhere except end

Lists in Java API

- The `java.util.LinkedList` and `java.util.ArrayList` classes implement the **`java.util.List` interface**
- This interface defines basic List operations: add, remove, contains, get by index, etc.
- Methods can use a List without needing to know its underlying implementation

```
import java.util.List;  
.  
. . .  
  
public static int findMax(List<Integer> values) {  
    int maxValue = values.get(0);  
    for (Integer value : values) {  
        if (maxValue < value) {  
            maxValue = value;  
        }  
    }  
    return maxValue;  
}
```

```
import java.util.List;  
.  
.  
.  
  
public static int findMax(List<Integer> values) {  
    int maxValue = values.get(0);  
    for (Integer value : values) {  
        if (maxValue < value) {  
            maxValue = value;  
        }  
    }  
    return maxValue;  
}
```

```
import java.util.List;
. . .
public static int findMax(List<Integer> values) {
    int maxValue = values.get(0);
    for (Integer value : values) {
        if (maxValue < value) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

```
import java.util.List;
. . .
public static int findMax(List<Integer> values) {
    int maxValue = values.get(0);
    for (Integer value : values) {
        if (maxValue < value) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

```
import java.util.List;
. . .
public static int findMax(List<Integer> values) {
    int maxValue = values.get(0);
    for (Integer value : values) {
        if (maxValue < value) {
            maxValue = value;
        }
    }
    return maxValue;
}
```

```
import java.util.List;  
.  
.  
.  
  
public static int findMax(List<Integer> values) {  
    int maxValue = values.get(0);  
    for (Integer value : values) {  
        if (maxValue < value) {  
            maxValue = value;  
        }  
    }  
    return maxValue;  
}
```

```
import java.util.List;
. . .
public static int findMax(List<Integer> values) {
    int maxValue = values.get(0);
    for (Integer value : values) {
        if (maxValue < value) {
            maxValue = value;
        }
    }
return maxValue;
}
```

```
public static void replaceEvenTokens(List<String> tokens,  
                                    String replace) {  
  
    for (int i = 0; i < tokens.size(); i++) {  
        if (i % 2 == 0) {  
            tokens.set(i, replace);  
        }  
    }  
}
```

```
public static void replaceEvenTokens(List<String> tokens,  
                                     String replace) {  
  
    for (int i = 0; i < tokens.size(); i++) {  
        if (i % 2 == 0) {  
            tokens.set(i, replace);  
        }  
    }  
}
```

```
public static void replaceEvenTokens(List<String> tokens,  
                                    String replace) {  
  
    for (int i = 0; i < tokens.size(); i++) {  
        if (i % 2 == 0) {  
            tokens.set(i, replace);  
        }  
    }  
}
```

```
public static void replaceEvenTokens(List<String> tokens,  
                                    String replace) {  
  
    for (int i = 0; i < tokens.size(); i++) {  
        if (i % 2 == 0) {  
            tokens.set(i, replace);  
        }  
    }  
}
```

```
public static void replaceEvenTokens(List<String> tokens,  
                                    String replace) {  
  
    for (int i = 0; i < tokens.size(); i++) {  
        if (i % 2 == 0) {  
            tokens.set(i, replace);  
        }  
    }  
}
```

Java Collections API

- LinkedList and ArrayList are part of the **Java Collections API**
- A collection of commonly used data structures, algorithms, and utilities

Collections API

- **java.util.Collection<E>** interface
- Represents a collection/group of elements
- Defines basic operations such as **add**, **remove**, **contains**, **size**, **etc.**
- More general than `java.util.List` interface: does not assume anything about the structure of the collection

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
        Collection<String> vocabLearnedSoFar ,
        String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty( )
            || !vocabLearnedSoFar.contains(wordToRemove)) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
        Collection<String> vocabLearnedSoFar ,
        String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty( )
            || !vocabLearnedSoFar.contains(wordToRemove)) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
    Collection<String> vocabLearnedSoFar ,
    String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty( )
        || !vocabLearnedSoFar.contains(wordToRemove)) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
        Collection<String> vocabLearnedSoFar ,
        String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty()
            || !vocabLearnedSoFar.contains(wordToRemove)) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
        Collection<String> vocabLearnedSoFar ,
        String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty( )
            || !vocabLearnedSoFar.contains(wordToRemove) ) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

```
import java.util.Collection;
. . .
public static boolean removeFromVocabLearned(
        Collection<String> vocabLearnedSoFar ,
        String wordToRemove) {
    if (vocabLearnedSoFar.isEmpty( )
            || !vocabLearnedSoFar.contains(wordToRemove)) {
        return false;
    } else {
        vocabLearnedSoFar.remove(wordToRemove);
        return true;
    }
}
```

Iterating over a Collection

- **java.util.Iterator<E>** interface
- Allows for step-by-step iteration over the elements of a Collection without needing to know its underlying structure
- Can get it by using the Collection's **iterator()** method

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

```
import java.util.Collection;
import java.util.Iterator;
. . .
public void printAll(Collection<String> words) {
    Iterator<String> iterator = words.iterator();
    while (iterator.hasNext()) {
        System.out.println(iterator.next());
    }
}
```

Java Collections API

- The Java LinkedList and ArrayList classes both implement the List interface
- The List interface extends the Collection interface, which is even more general
- All Collections have Iterators that can be used to retrieve each element

SD2x1.6

Queues; Stacks

Kathy

Review: Linked Lists

- Series of “Nodes” with each one linked to the next
- Can easily add and remove values to front, rear, and middle of list
- Foundation of many other data structures

Motivating Example

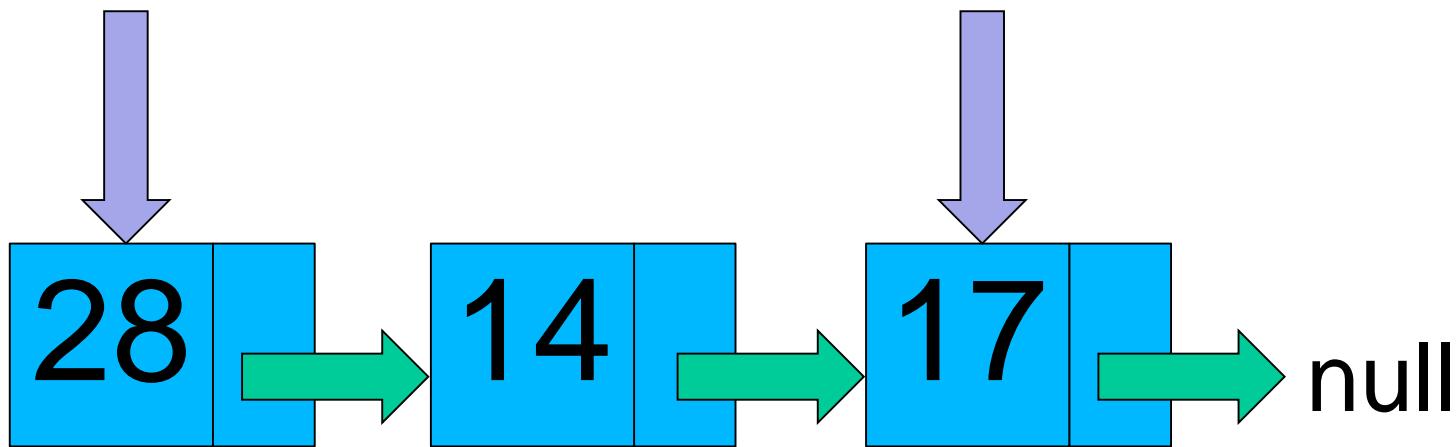
- Let's say we're creating system to track requests for tickets to an event
 - request made for x people
 - cancellation of n people
- Need to maintain the order in which these requests are processed!

Motivating Example (cont'd)

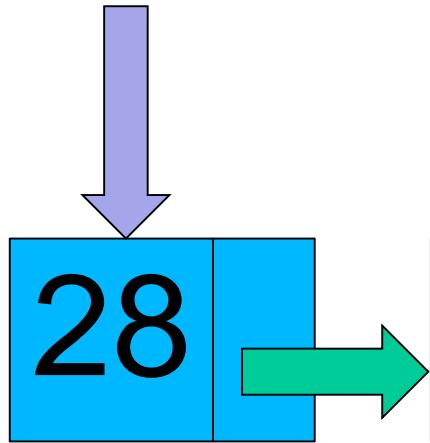
- Lists in general allow us to add elements at arbitrary positions
 - and also remove them from arbitrary positions
- But in this case we want to **only** be able to:
 - add to the rear
 - remove from the front

head

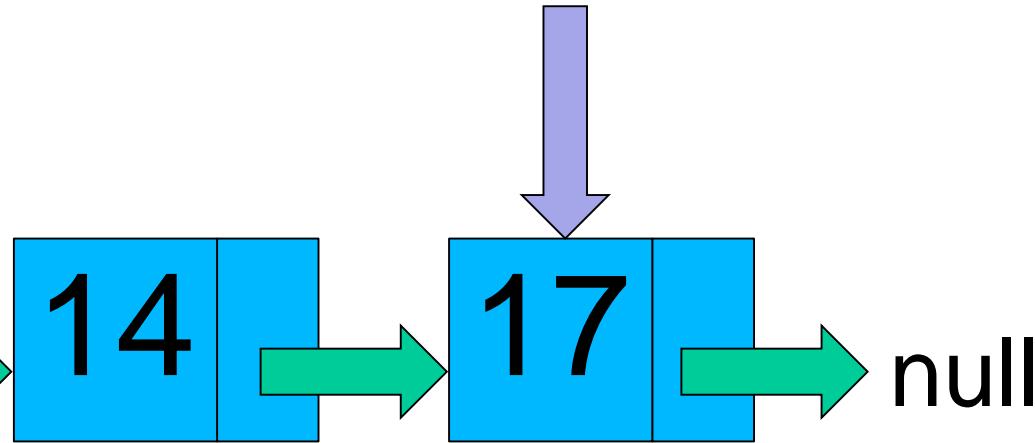
tail



“front”

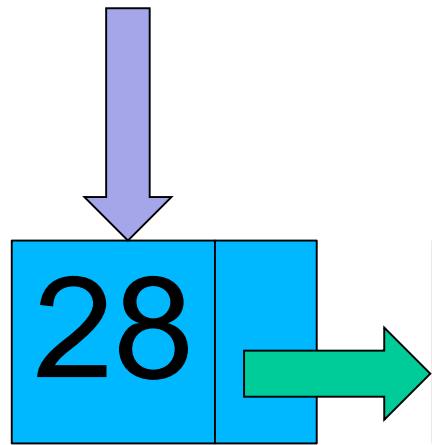


tail

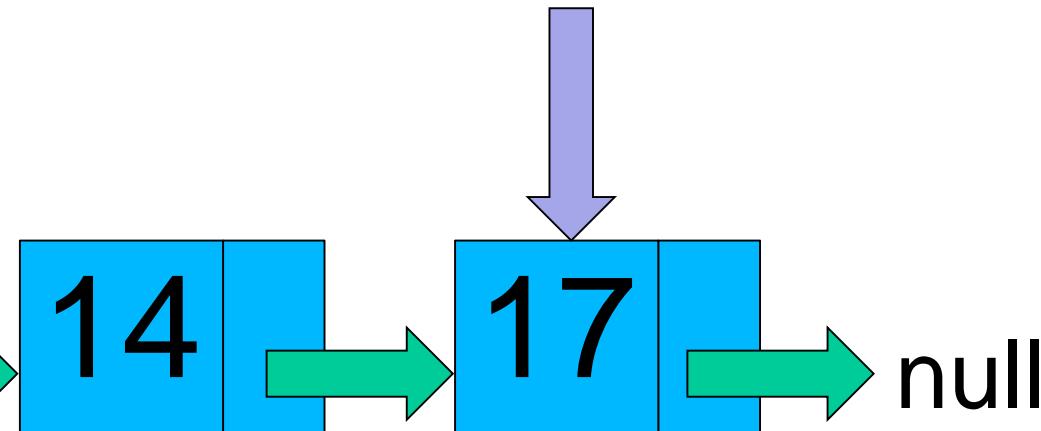


null

“front”



“rear”

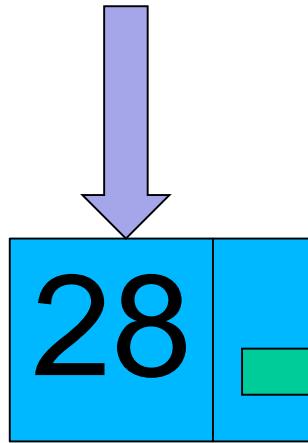




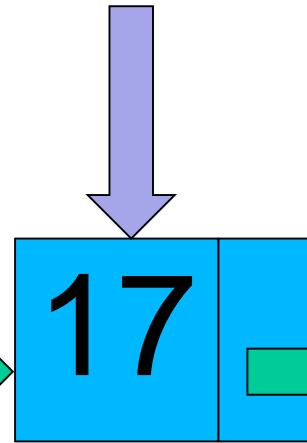
Queue!

Add:
insert at the rear of the
Queue

“front”

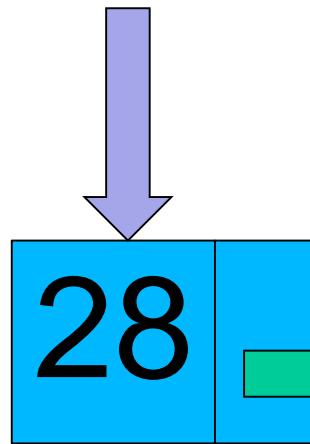


“rear”



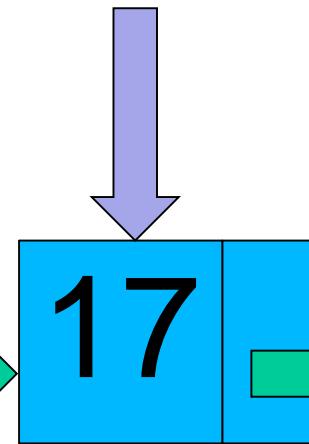
null

“front”



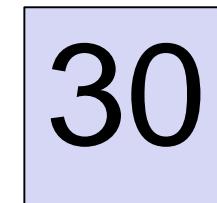
14

“rear”

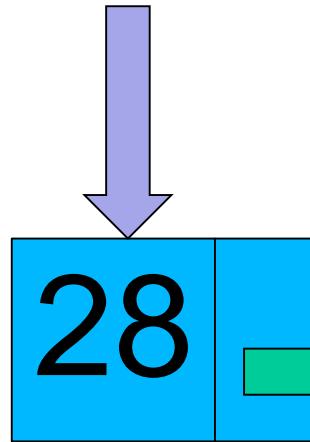


17

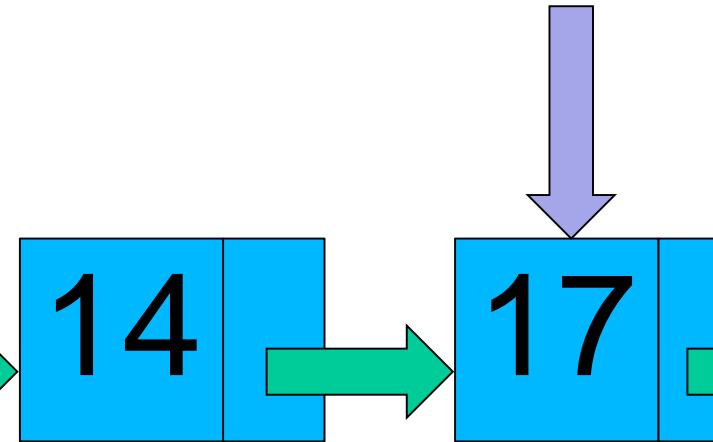
null



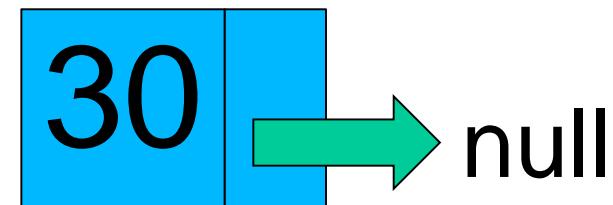
“front”



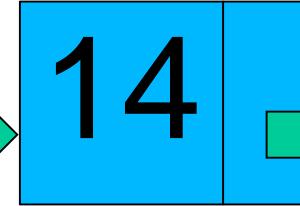
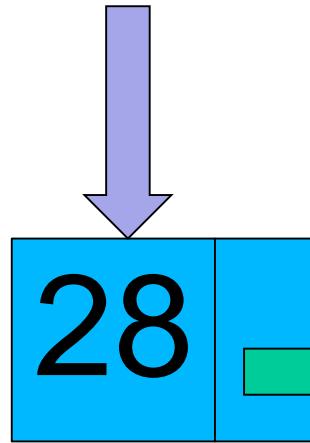
“rear”



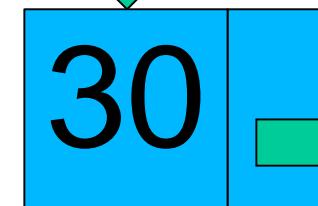
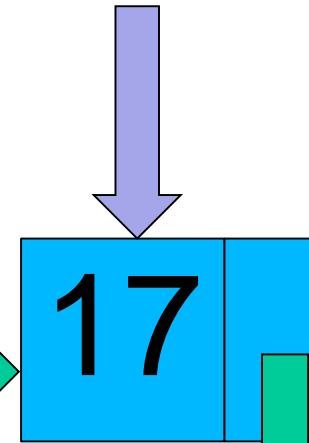
null



“front”

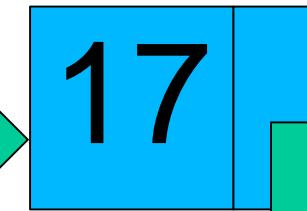
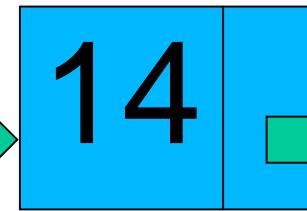
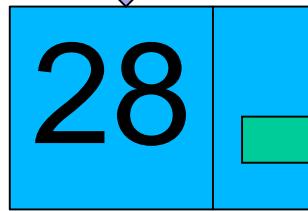
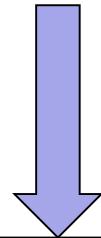


“rear”

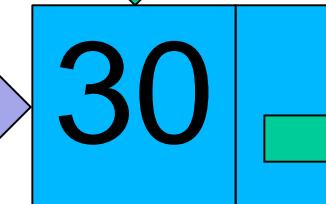


null

“front”



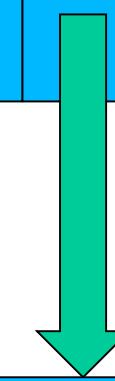
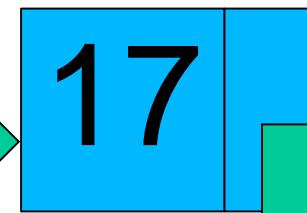
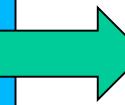
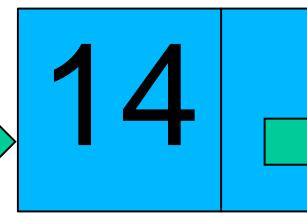
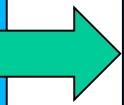
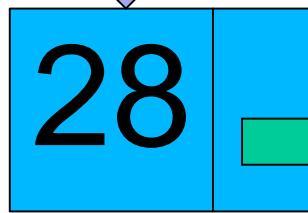
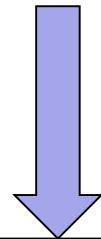
“rear”



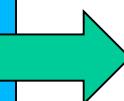
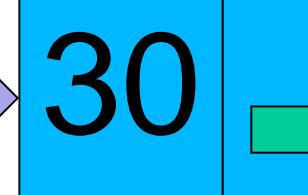
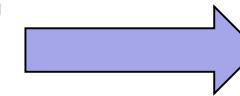
null

**Remove:
remove from the front of
the Queue**

“front”

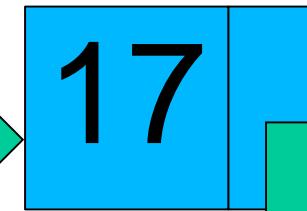
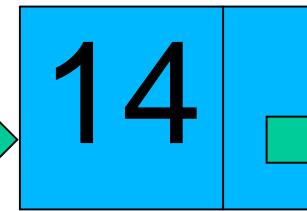
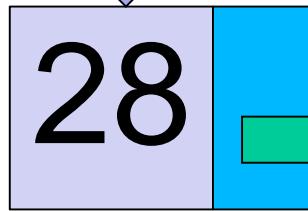
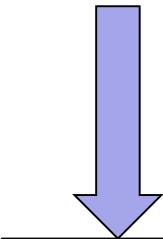


“rear”

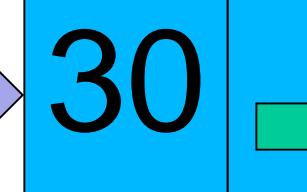


null

“front”

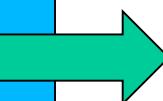
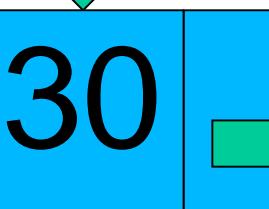
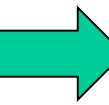
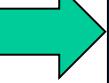
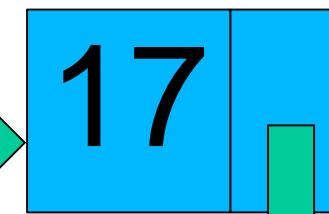
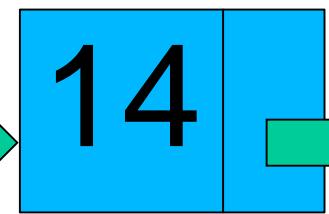
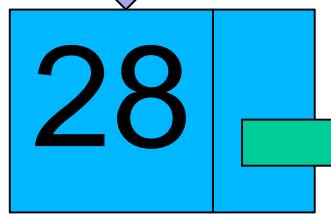
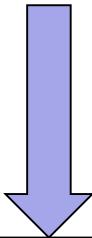


“rear”



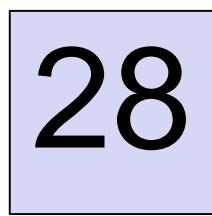
null

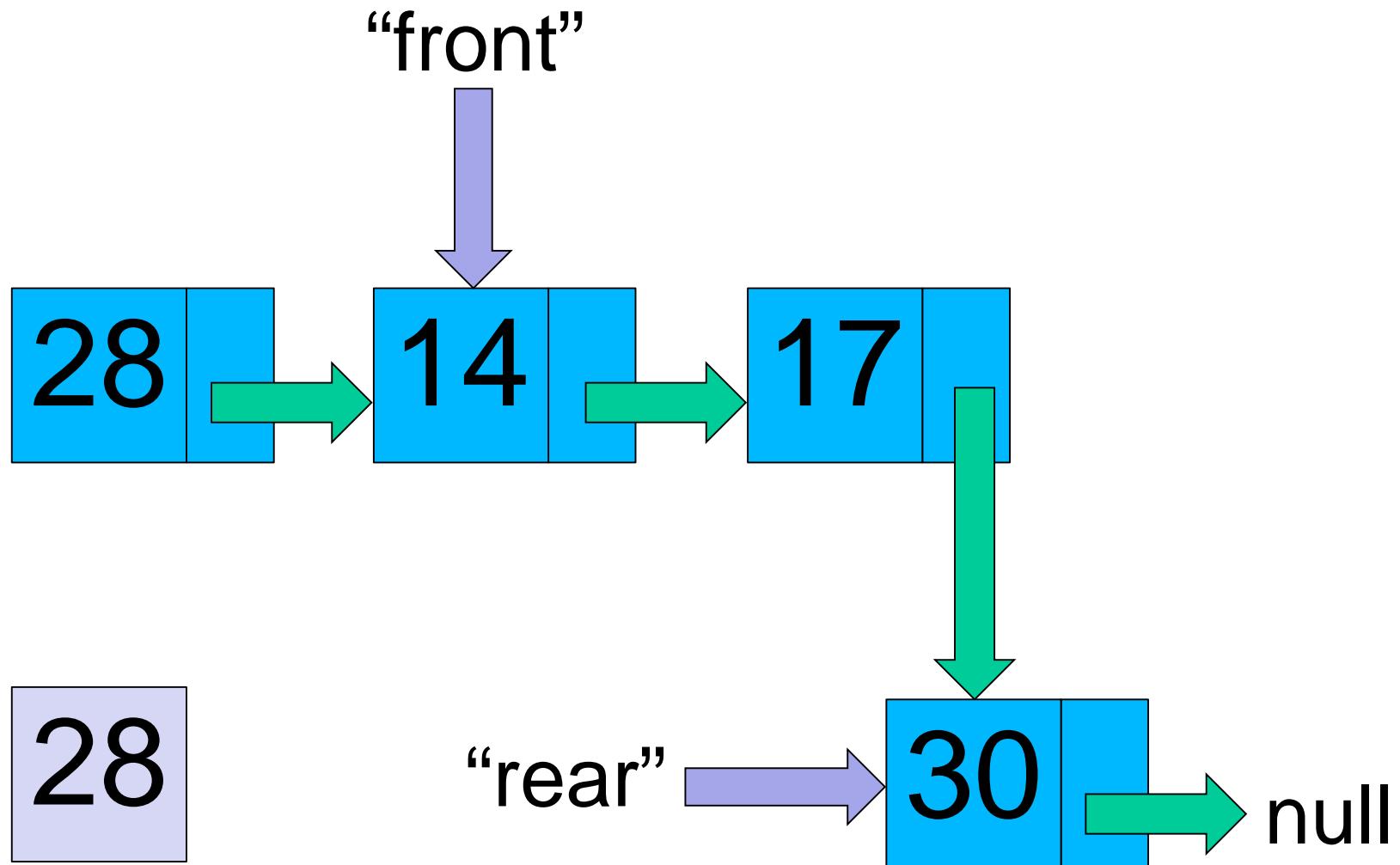
“front”



null

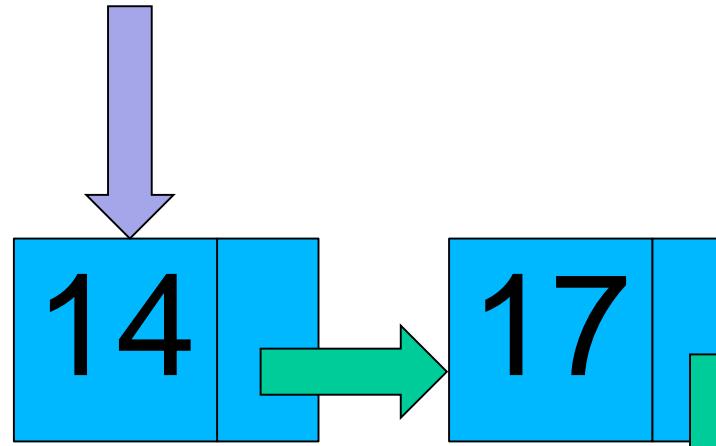
“rear”



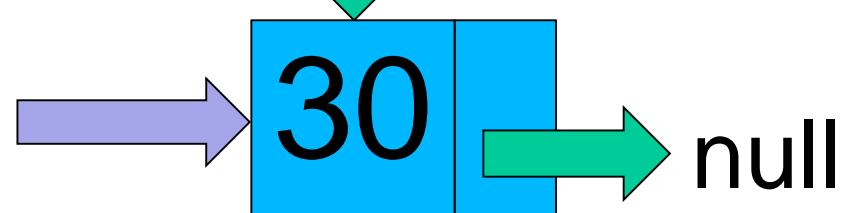


28

“front”



“rear”



```
import java.util.LinkedList;  
  
public class Queue {  
  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;

public class Queue {

    protected LinkedList list = new LinkedList();

}

}
```

```
import java.util.LinkedList;  
  
public class Queue {  
  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;  
  
public class Queue {  
  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;

public class Queue {

    protected LinkedList list = new LinkedList();

    public void add(Object value) {
        list.add(value); // adds to end
    }

}
```

```
import java.util.LinkedList;

public class Queue {

    protected LinkedList list = new LinkedList();

    public void add(Object value) {
        list.add(value); // adds to end
    }

    public Object remove() {
        return list.removeFirst(); // removes from front
    }

}
```

```
import java.util.LinkedList;

public class Queue {

    protected LinkedList list = new LinkedList();

    public void add(Object value) {
        list.add(value); // adds to end
    }

    public Object remove() {
        if (list.isEmpty())
            return null;
        return list.removeFirst(); // removes from front
    }

}
```

Queues in Java API

- **java.util.Queue<E>** interface
- **isEmpty**: tests if queue is empty
 - inherited from Collection
- **add**: inserts object into rear of queue
- **peek**: returns (but does not remove) object at front of queue
- **remove**: returns (and removes) object at front of queue
- implementations: `java.util.LinkedList`, etc.

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
public void addTicketRequest(int numPeople) {  
    ticketRequests.add(numPeople);  
}  
  
public int processUntilNoCapacity() {  
    int numRequestsProcessed = 0;  
    while (!ticketRequests.isEmpty()) {  
        int remainAfterRequest = (availableCapacity  
                                - ticketRequests.peek());  
        if (remainAfterRequest < 0)  
            return numRequestsProcessed;  
        availableCapacity -= ticketRequests.remove();  
        numRequestsProcessed++;  
    }  
    return 0;  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
public int processUntilNoCapacity() {  
    int numRequestsProcessed = 0;  
    while (!ticketRequests.isEmpty()) {  
        int remainAfterRequest = (availableCapacity  
                                - ticketRequests.peek());  
        if (remainAfterRequest < 0)  
            return numRequestsProcessed;  
        availableCapacity -= ticketRequests.remove();  
        numRequestsProcessed++;  
    }  
    return 0;  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

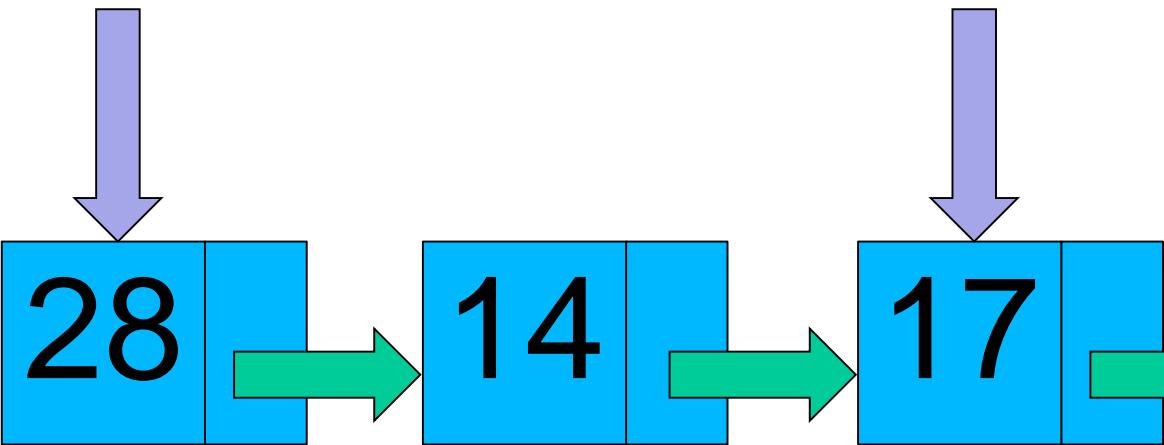
```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

```
public class EventAvailableCapacity {  
  
    protected Queue<Integer> ticketRequests;  
    protected int availableCapacity;  
  
    public EventAvailableCapacity(int maxCapacity) {  
        ticketRequests = new LinkedList<Integer>();  
        availableCapacity = maxCapacity;  
    }  
  
    public void addTicketRequest(int numPeople) {  
        ticketRequests.add(numPeople);  
    }  
  
    public int processUntilNoCapacity() {  
        int numRequestsProcessed = 0;  
        while (!ticketRequests.isEmpty()) {  
            int remainAfterRequest = (availableCapacity  
                - ticketRequests.peek());  
            if (remainAfterRequest < 0)  
                return numRequestsProcessed;  
            availableCapacity -= ticketRequests.remove();  
            numRequestsProcessed++;  
        }  
        return 0;  
    }  
}
```

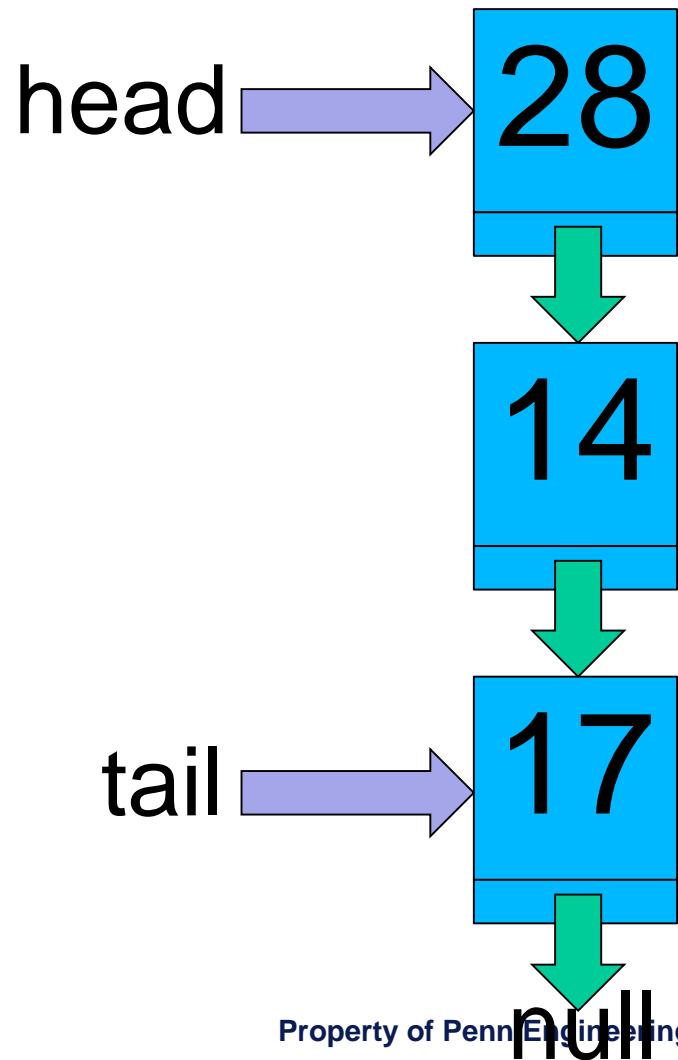
Motivating Example #2

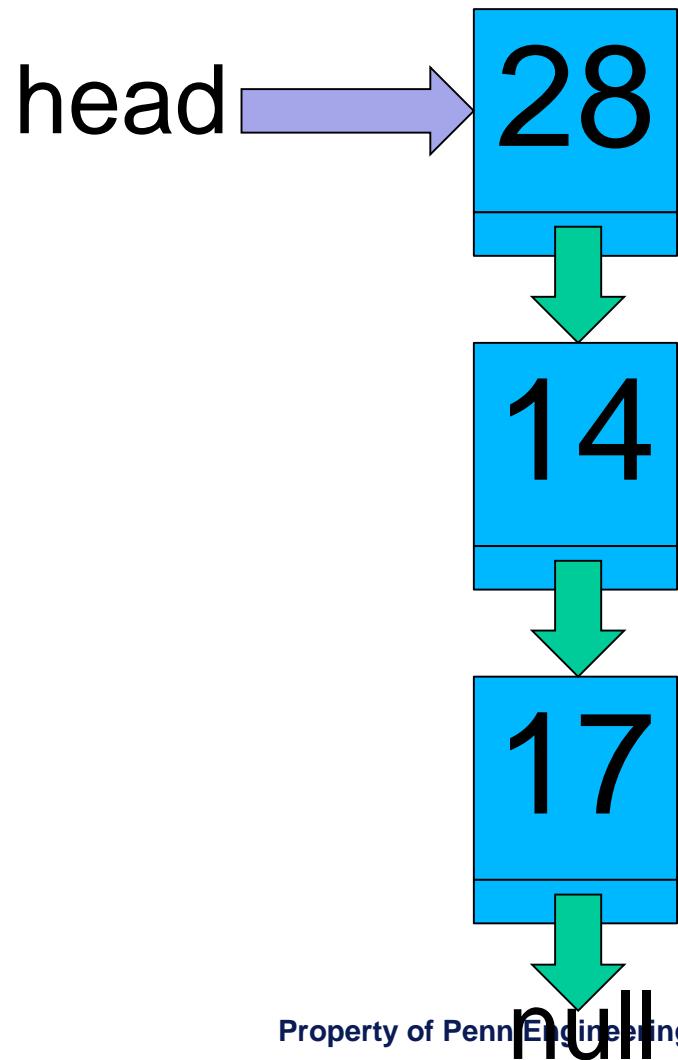
- Now let's say we want to implement the “undo” functionality of a text editor
- Remember the operations in the reverse order in which they occurred
 - Operations: “insert c”, “delete k”, “insert m”
 - Undo ordering: “insert m”, “delete k”, “insert c”
- Here we want a List that will **only** be able to:
 - add to the **front**
 - and also remove from the front

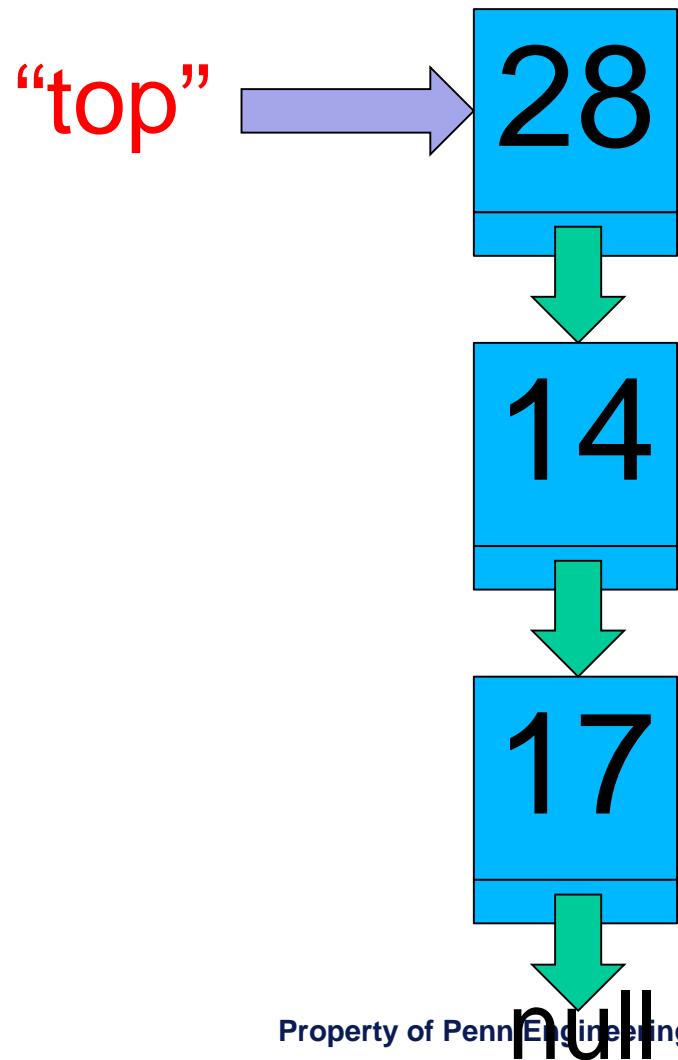
head



tail





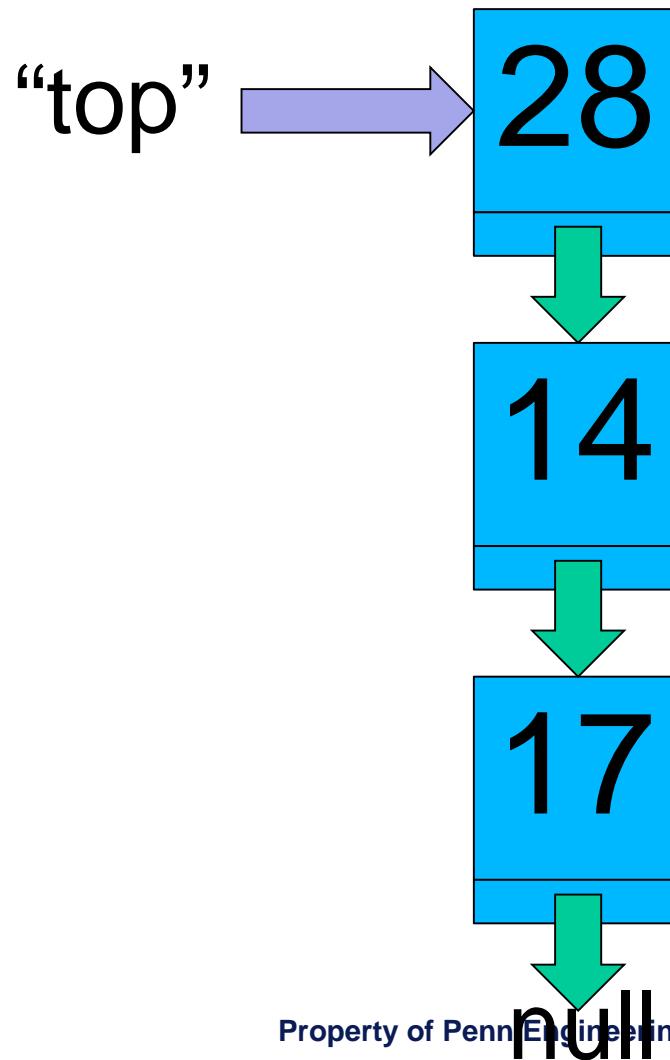


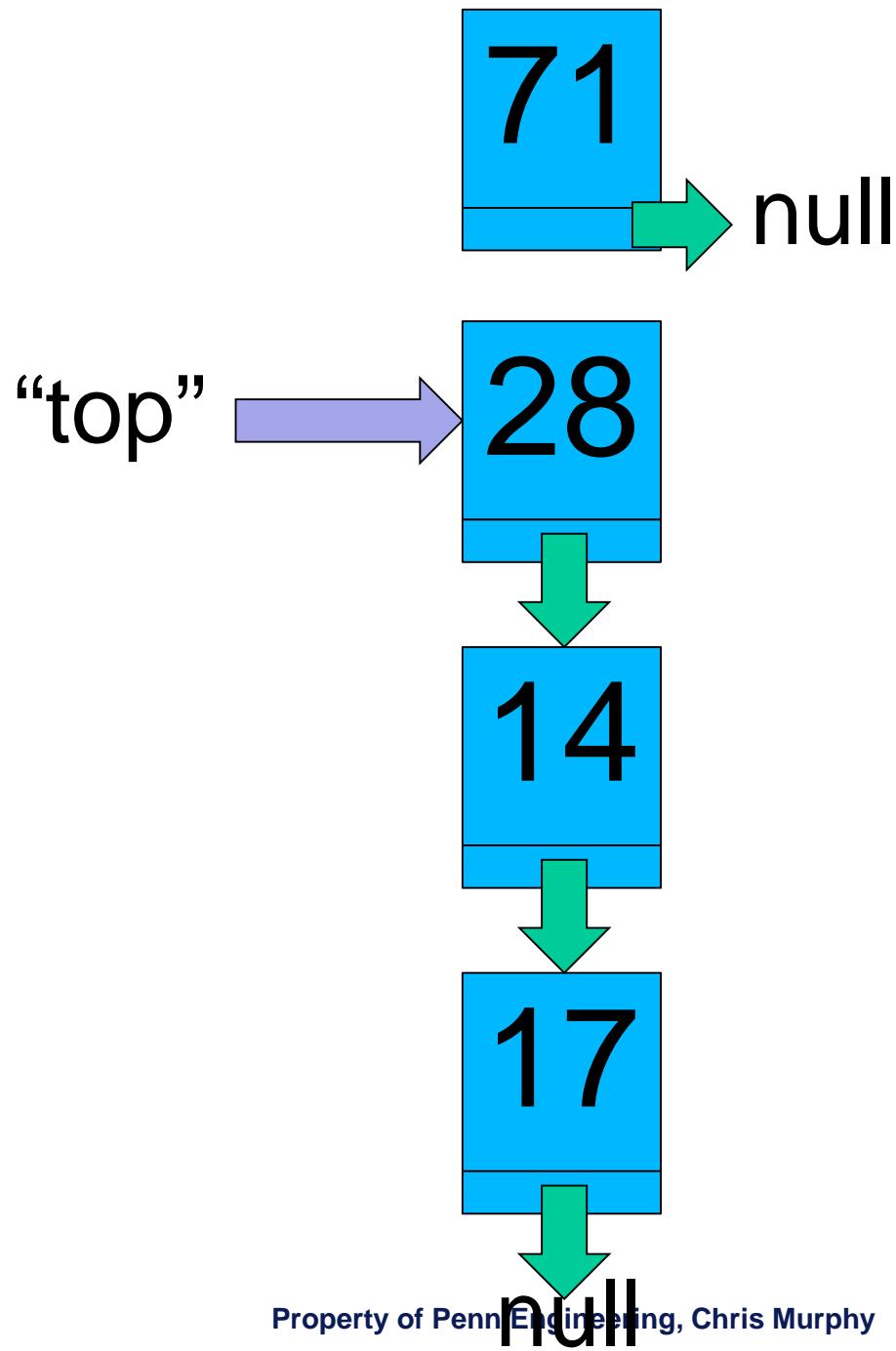


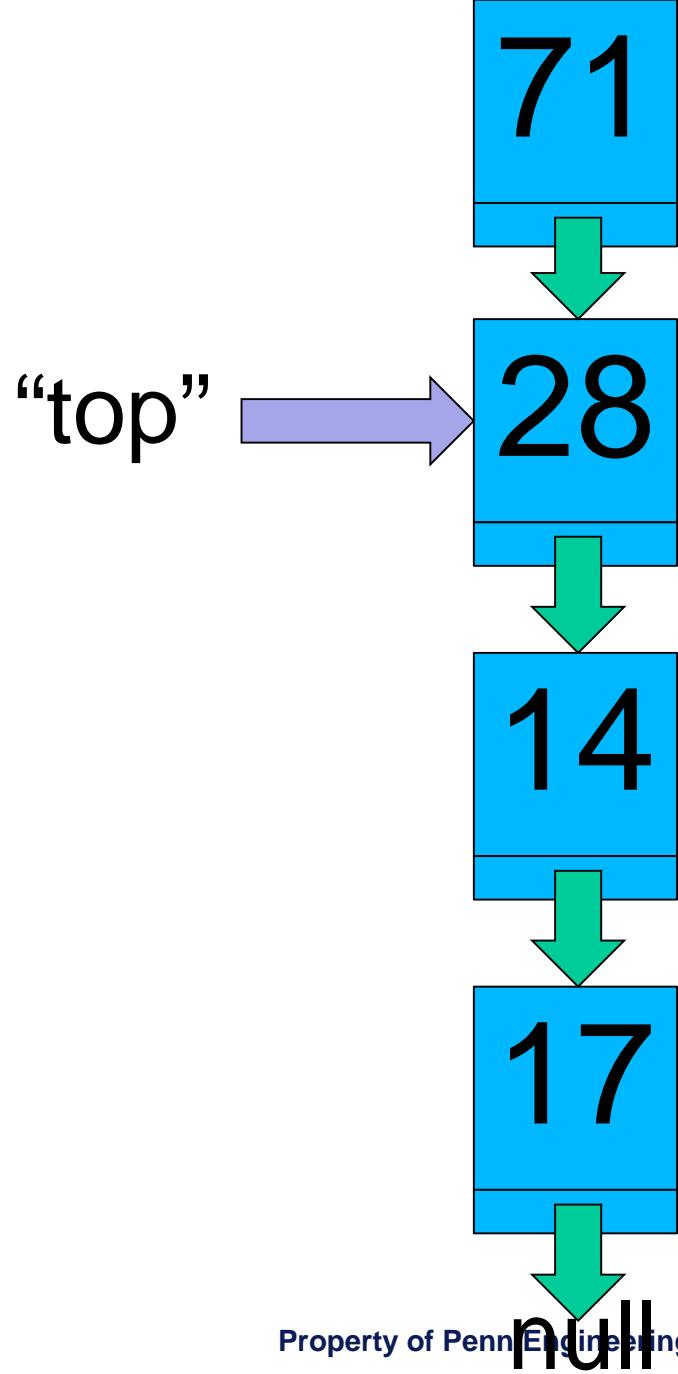
Stack!

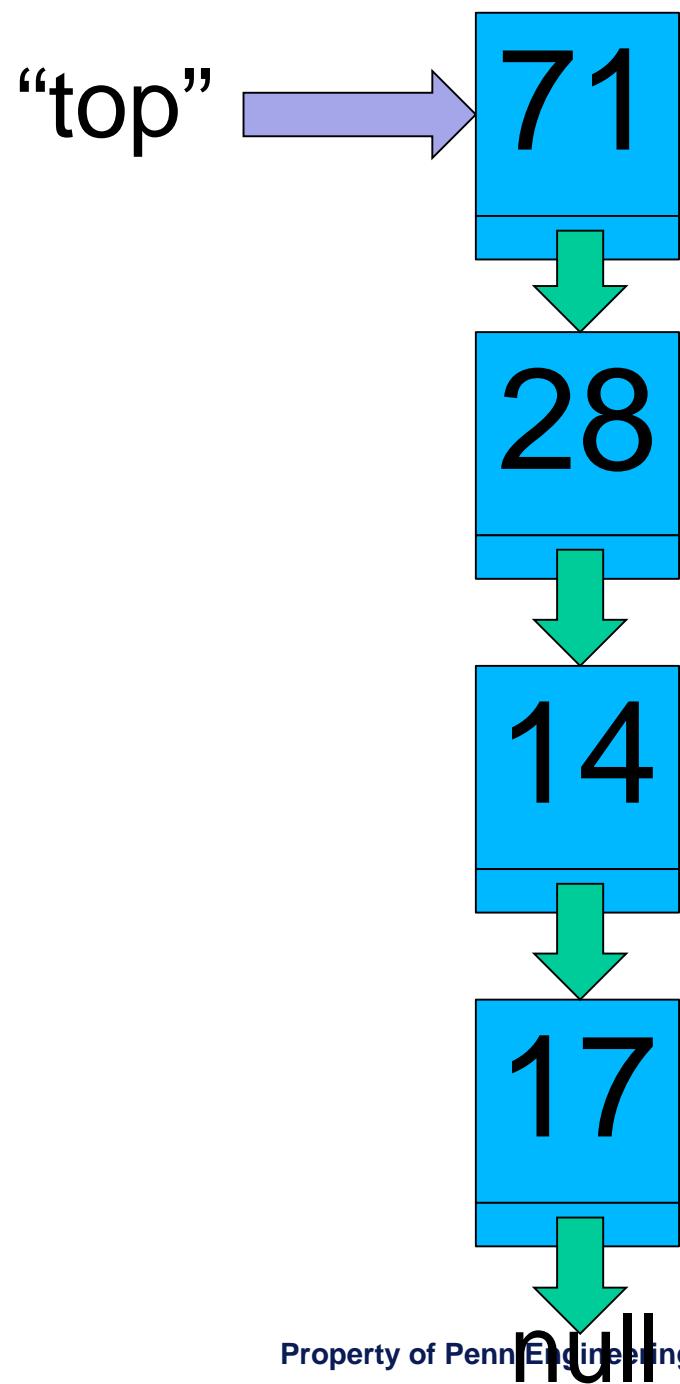
**Push:
add to the top of the Stack**

71

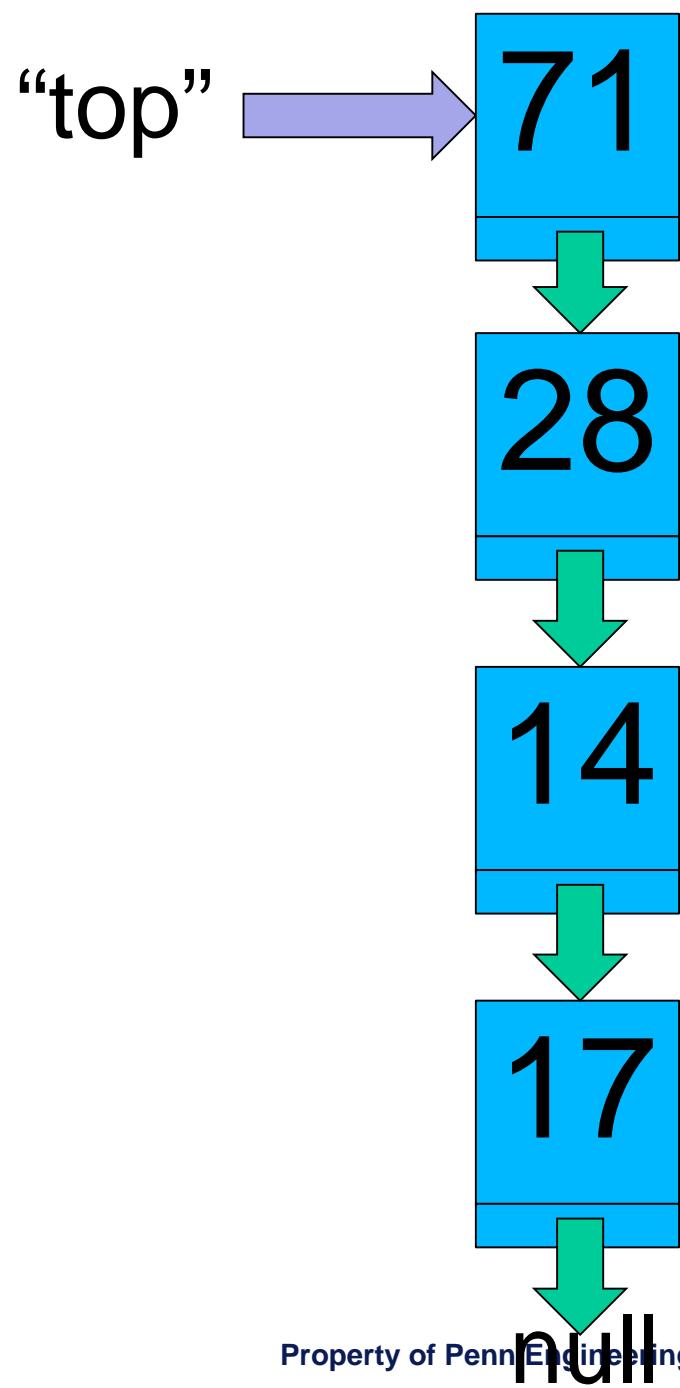


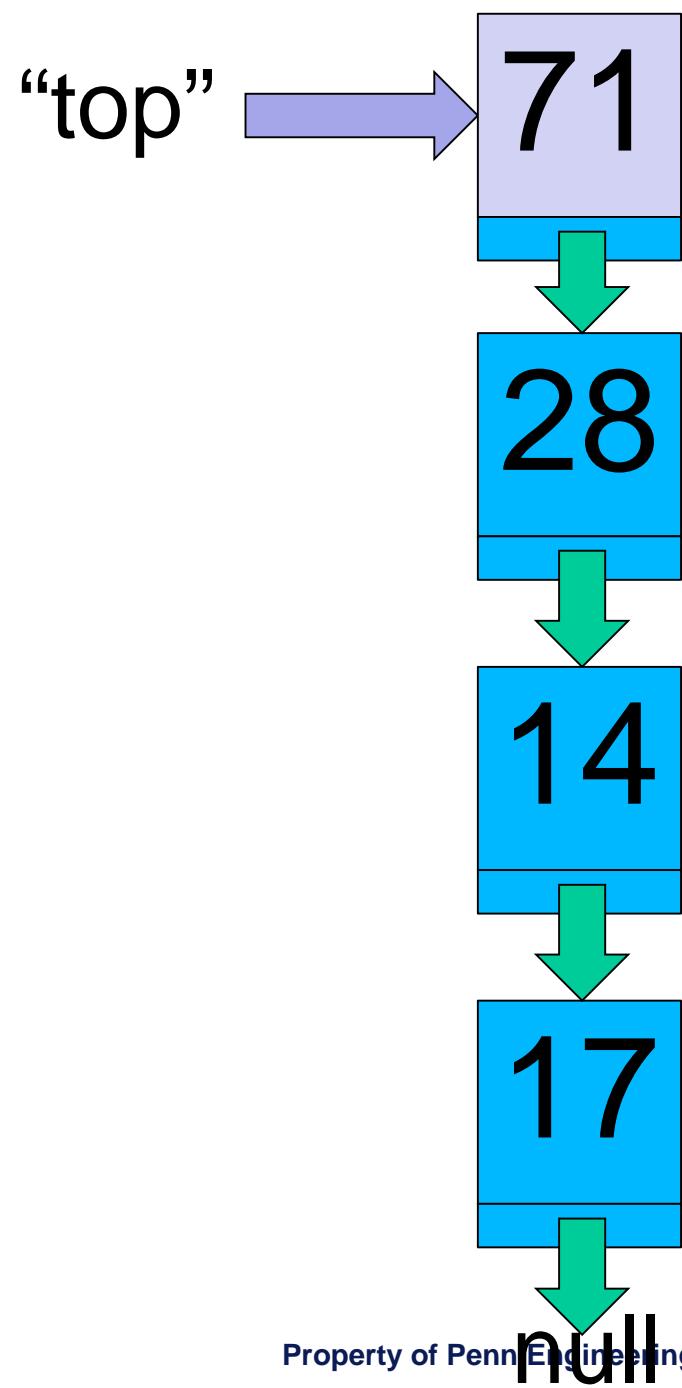


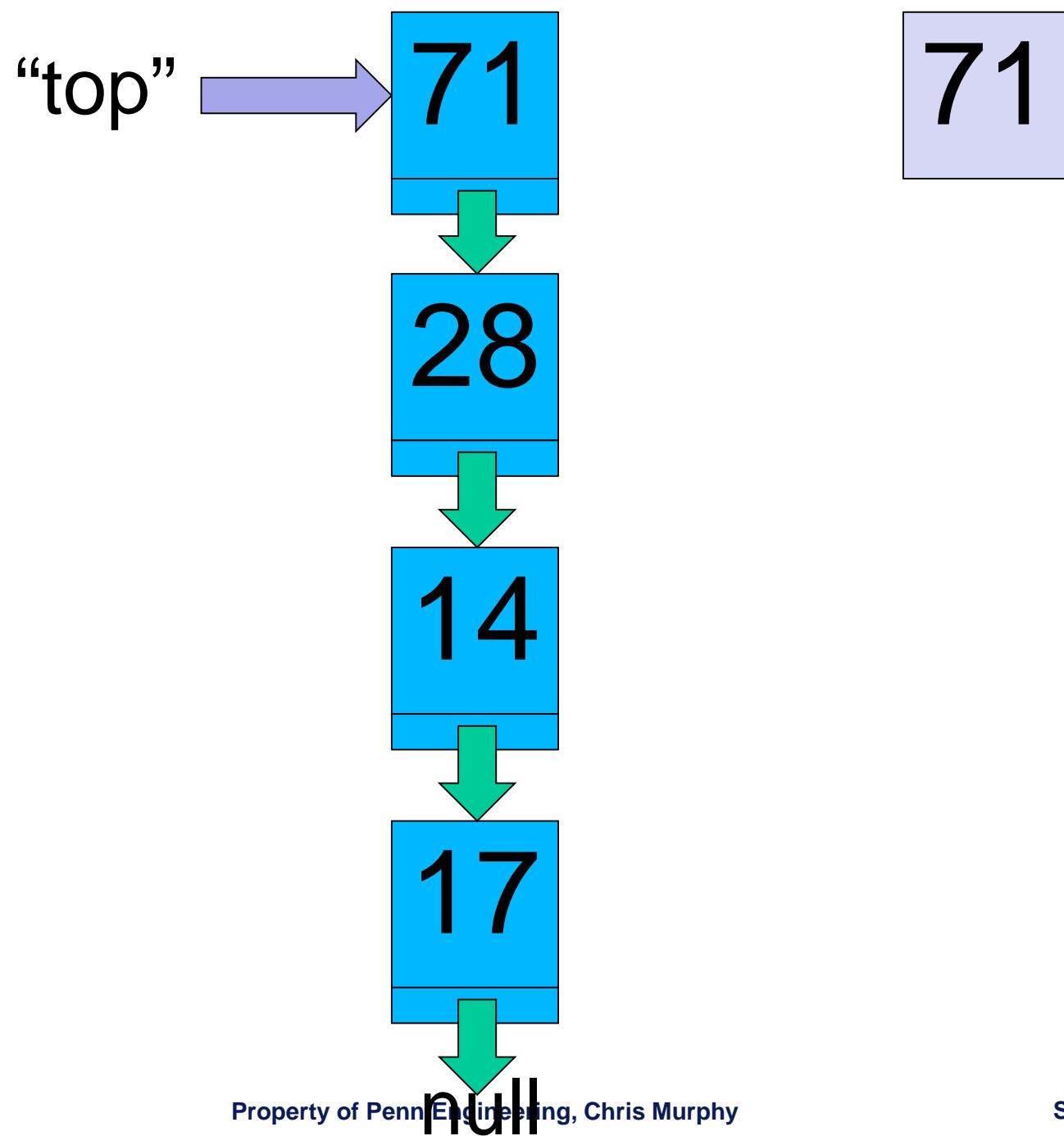


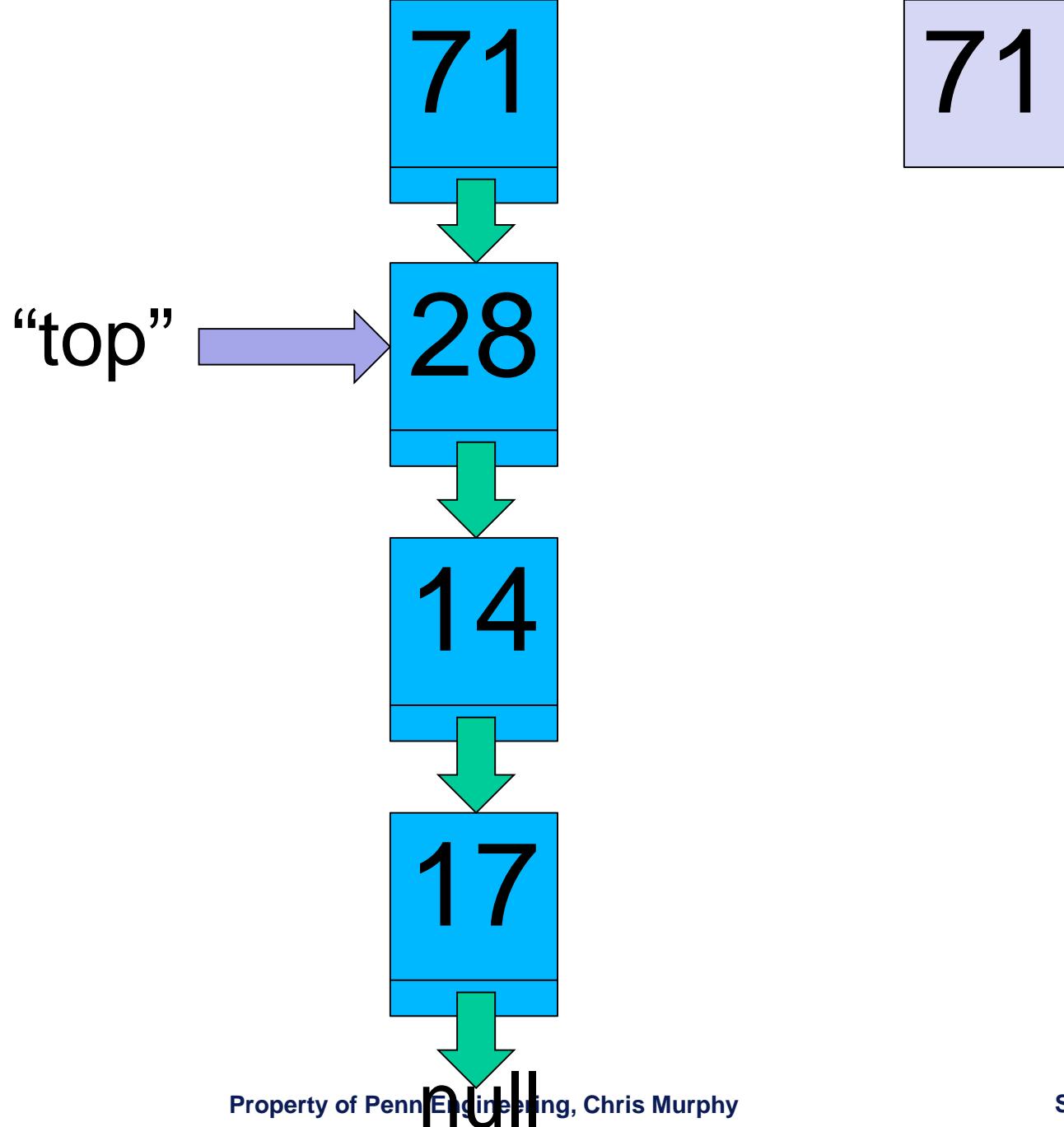


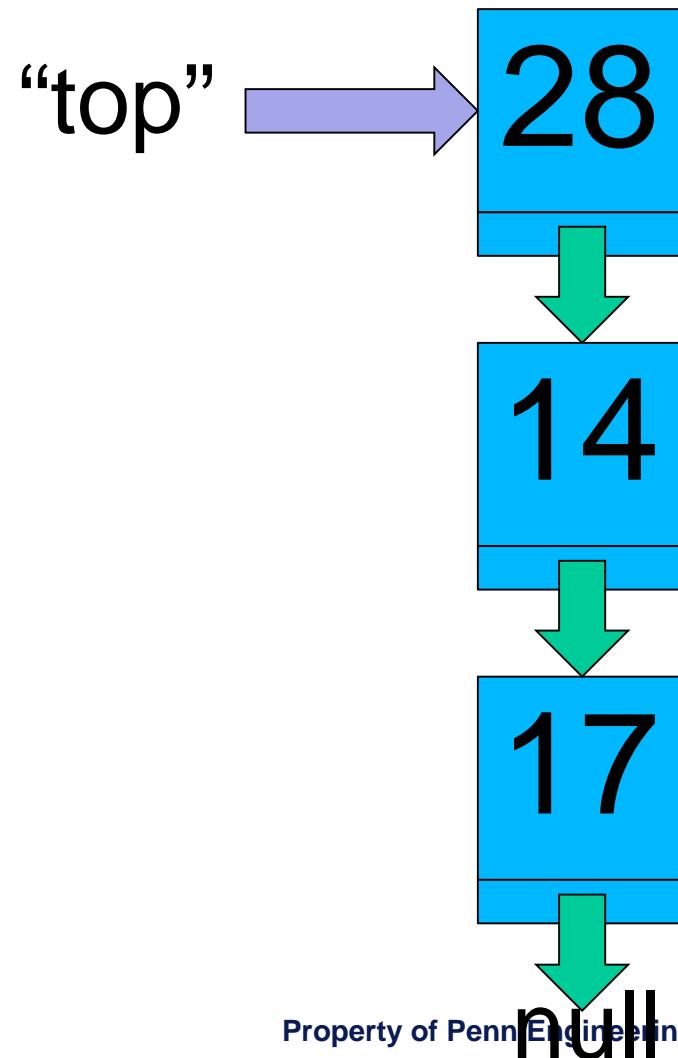
Pop:
remove from the top of the Stack











```
import java.util.LinkedList;

public class Stack {

    protected LinkedList list = new LinkedList();

}

}
```

```
import java.util.LinkedList;  
  
public class Stack {  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;  
  
public class Stack {  
  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;  
  
public class Stack {  
  
    protected LinkedList list = new LinkedList();  
  
}
```

```
import java.util.LinkedList;

public class Stack {

    protected LinkedList list = new LinkedList();

    public void push(Object value) {
        list.addFirst(value);
    }

}
```

```
import java.util.LinkedList;

public class Stack {

    protected LinkedList list = new LinkedList();

    public void push(Object value) {
        list.addFirst(value);
    }

    public Object pop() {
        return list.removeFirst();
    }

}
```

```
import java.util.LinkedList;

public class Stack {

    protected LinkedList list = new LinkedList();

    public void push(Object value) {
        list.addFirst(value);
    }

    public Object pop() {
        if (list.isEmpty())
            return null;
        return list.removeFirst();
    }

}
```

```
import java.util.LinkedList;

public class Stack {

    protected LinkedList list = new LinkedList();

    public void push(Object value) {
        list.addFirst(value);
    }

    public Object pop() {
        if (list.isEmpty())
            return null;
        return list.removeFirst();
    }

    public Object peek() {
        return list.getFirst();
    }
}
```

Stacks in Java API

- **java.util.Stack<E>**
- **empty**: tests if stack is empty
- **peek**: returns (but does not remove) object on top of stack
- **pop**: returns and removes object on top of stack
- **push**: places an object on top of stack

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
}  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
public String undo() {  
    if (!canUndo()) {  
        return null;  
    }  
    return history.pop();  
}  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
        return history.pop();  
    }  
}
```

```
public class TextEditorHistory {  
  
    protected Stack<String> history;  
  
    public TextEditorHistory() {  
        history = new Stack<String>();  
    }  
  
    public void addToHistory (String currentVersion) {  
        history.push(currentVersion);  
    }  
  
    public boolean canUndo() {  
        return !history.empty();  
    }  
  
    public String undo() {  
        if (!canUndo()) {  
            return null;  
        }  
return history.pop();  
    }  
}
```

Stacks and Queues



Stack:

Last In, First Out (LIFO)

Queue:

First In, First Out (FIFO)

SD2x1.7

LL vs. AL; Big-Oh; motivating Sets

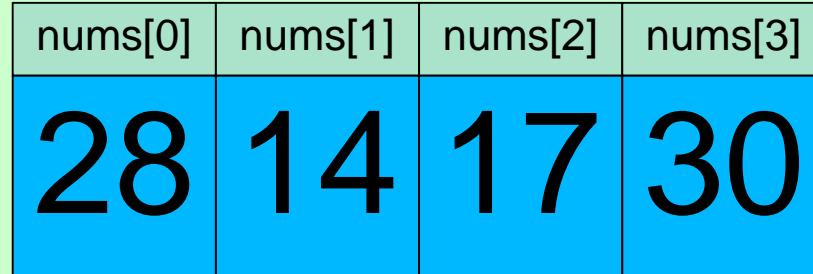
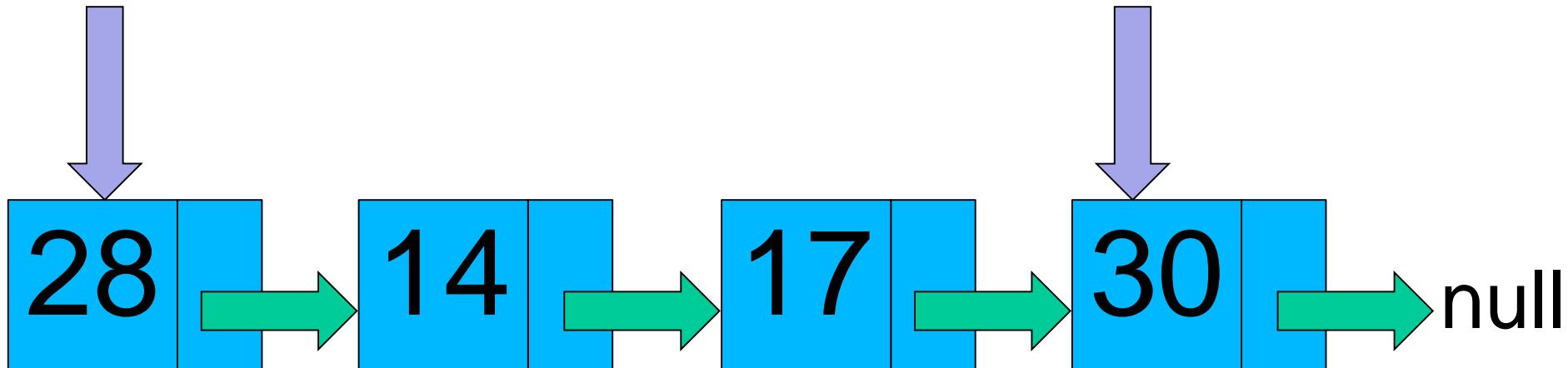
Kathy

Comparing LinkedList vs ArrayList

- Let's say we have a LinkedList of n numbers
- And an ArrayList containing the same numbers
- How many steps do we need in order to get an element in the list **by its index?**

head

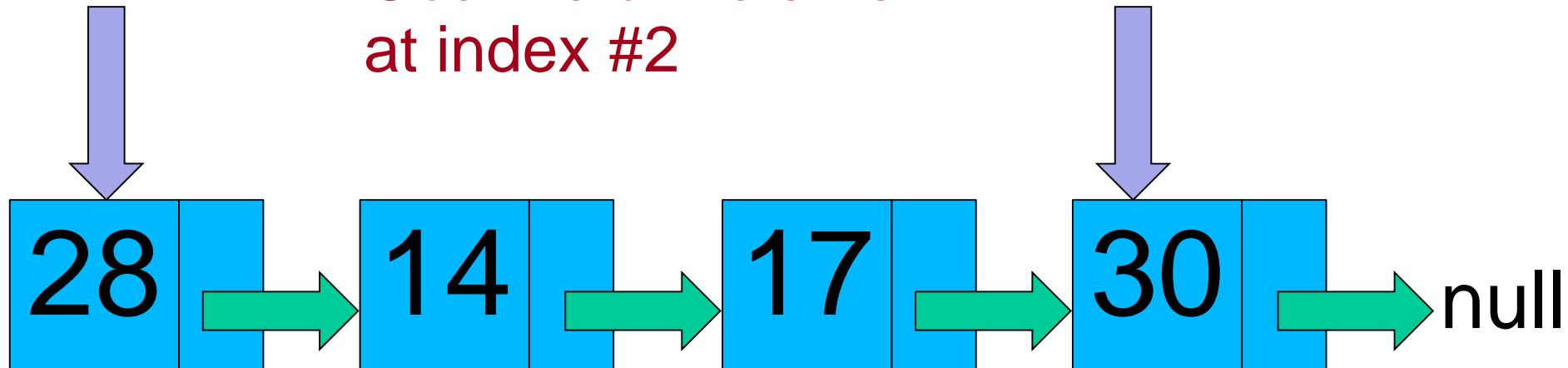
tail



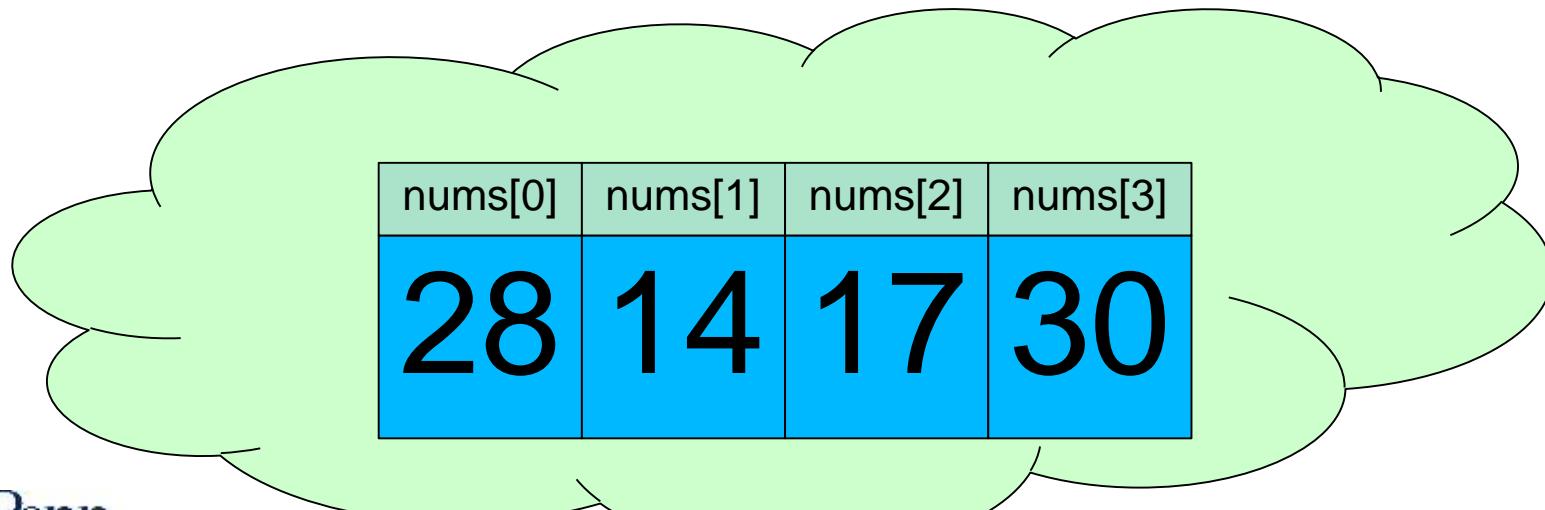
head

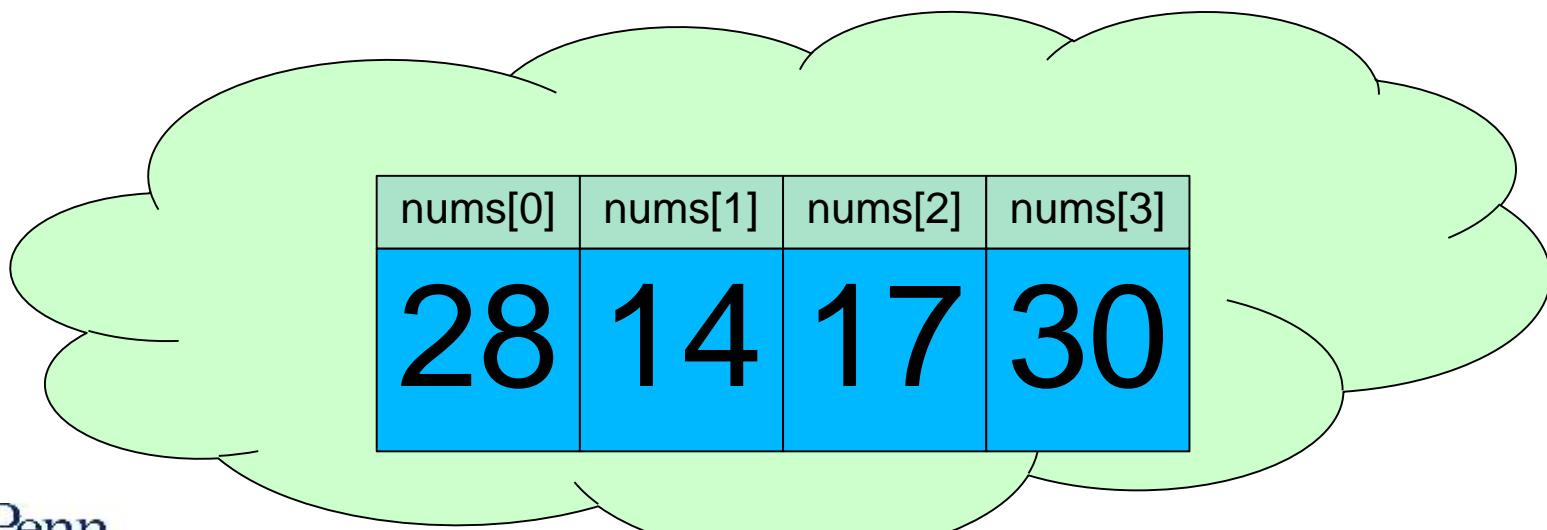
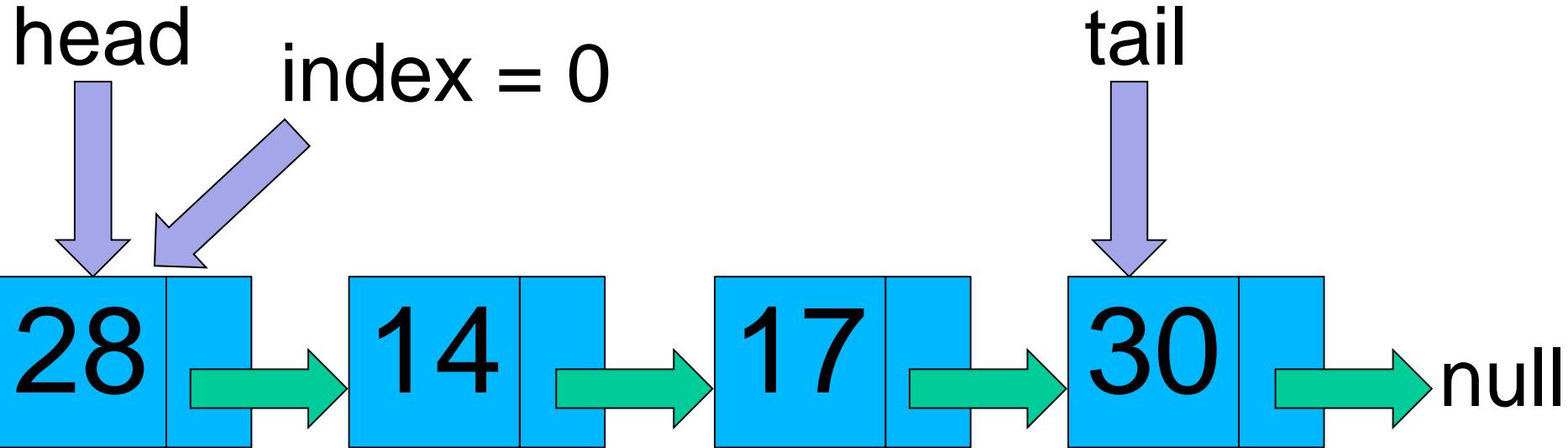
Goal: return element
at index #2

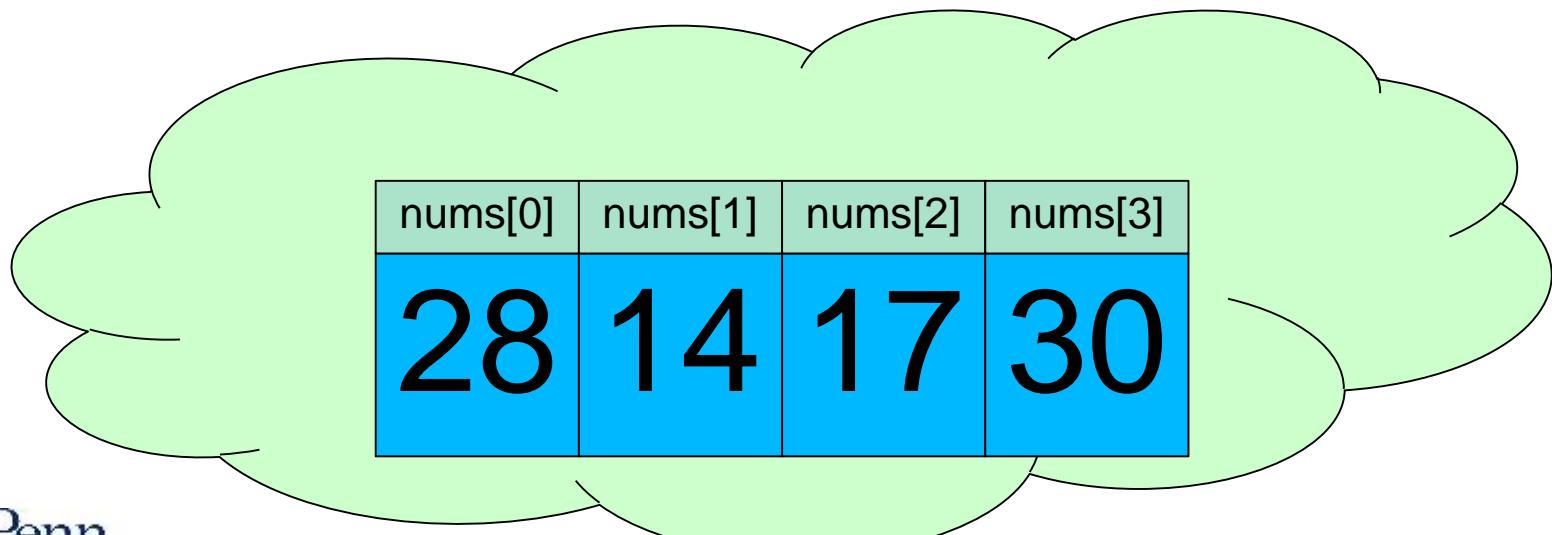
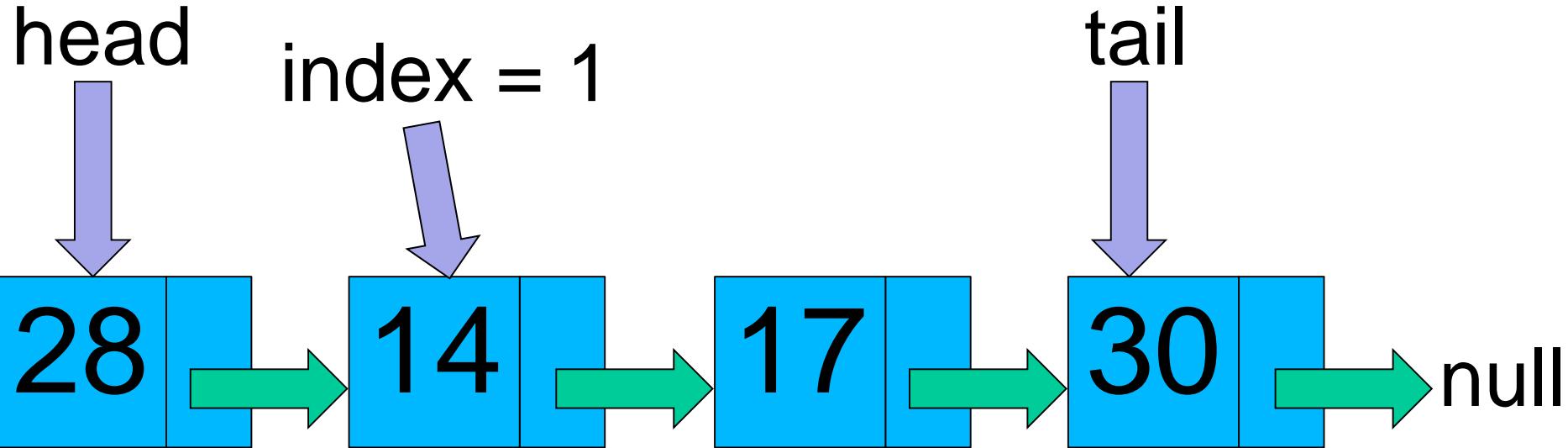
tail

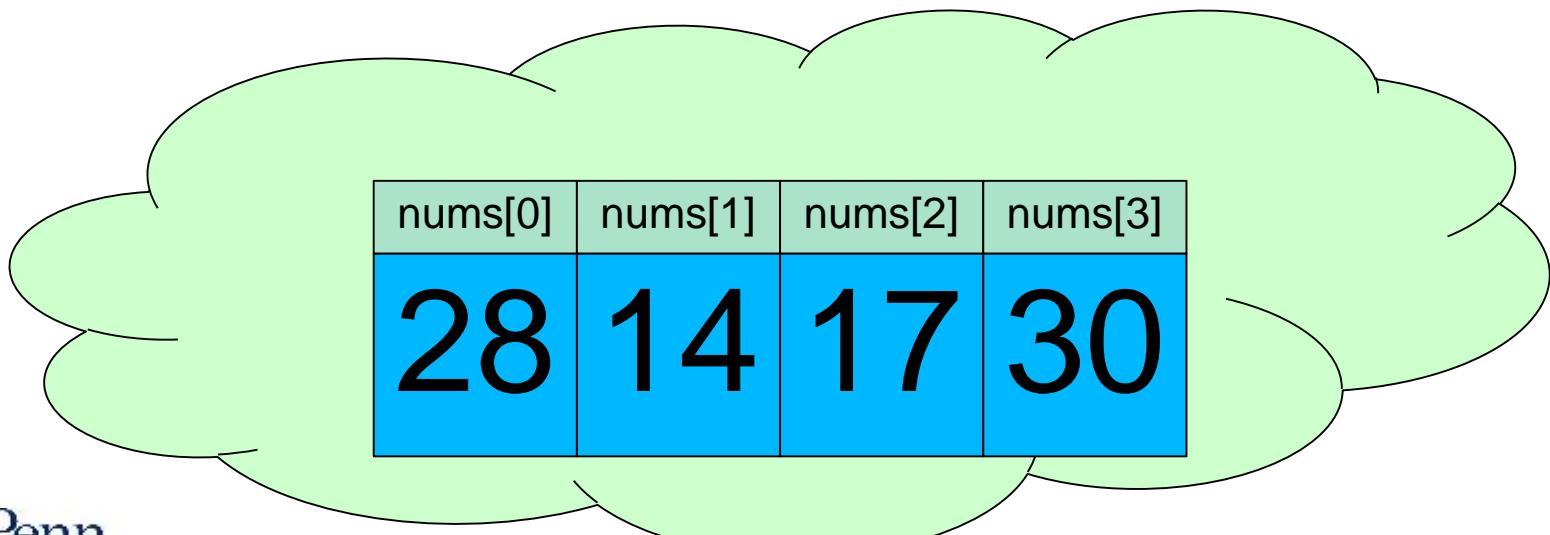
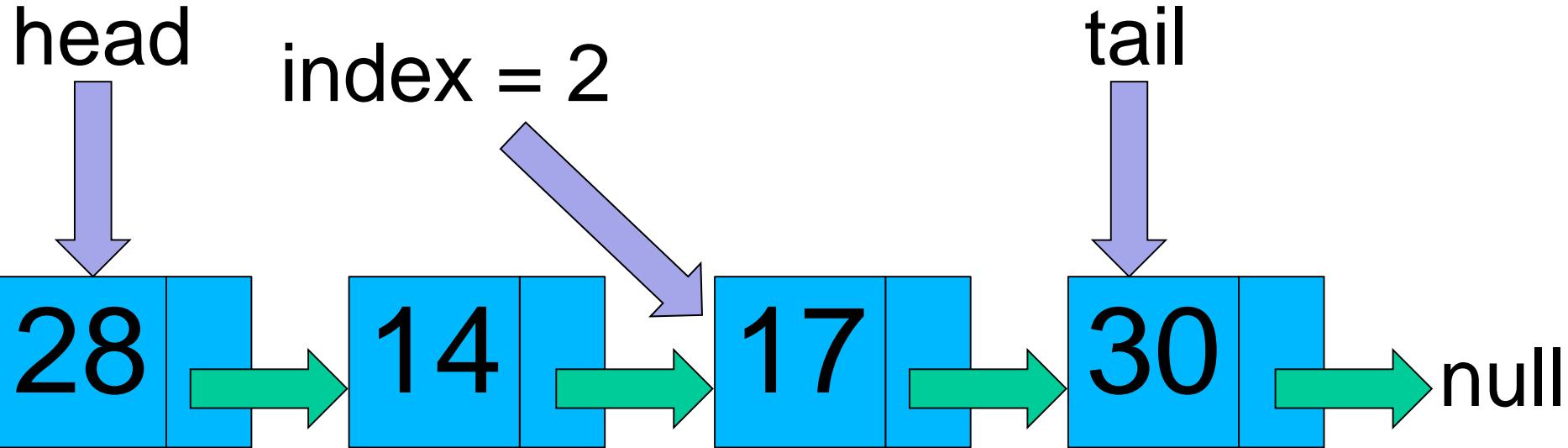


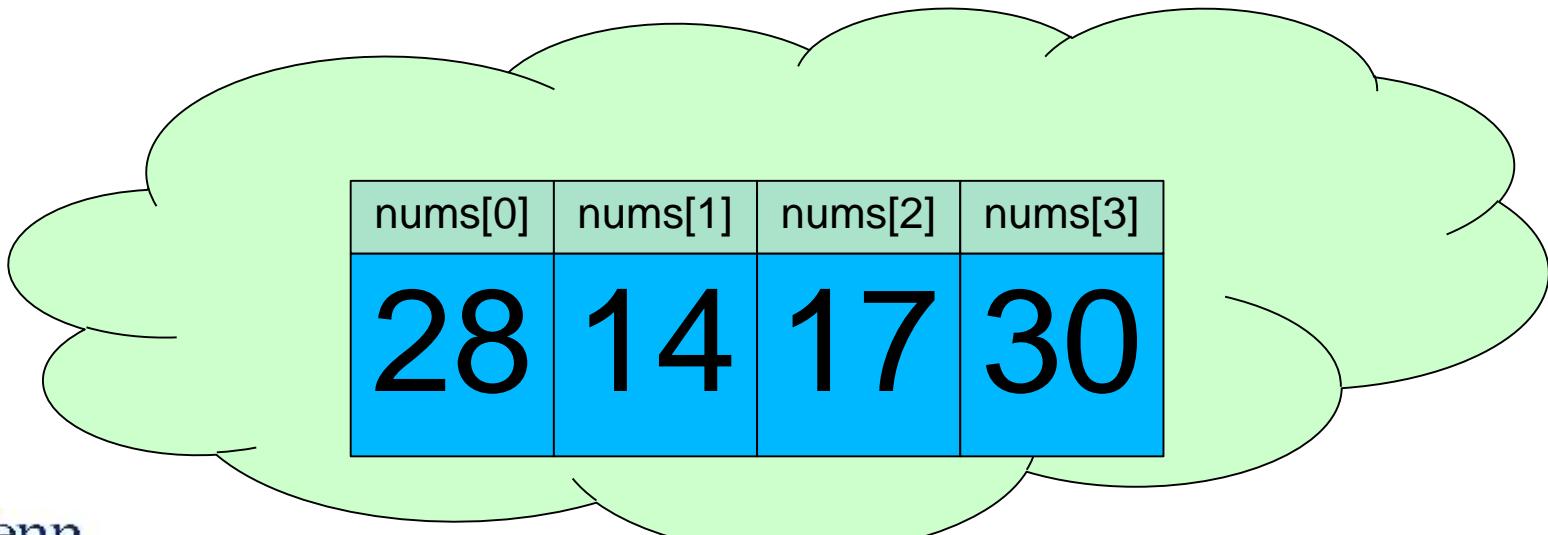
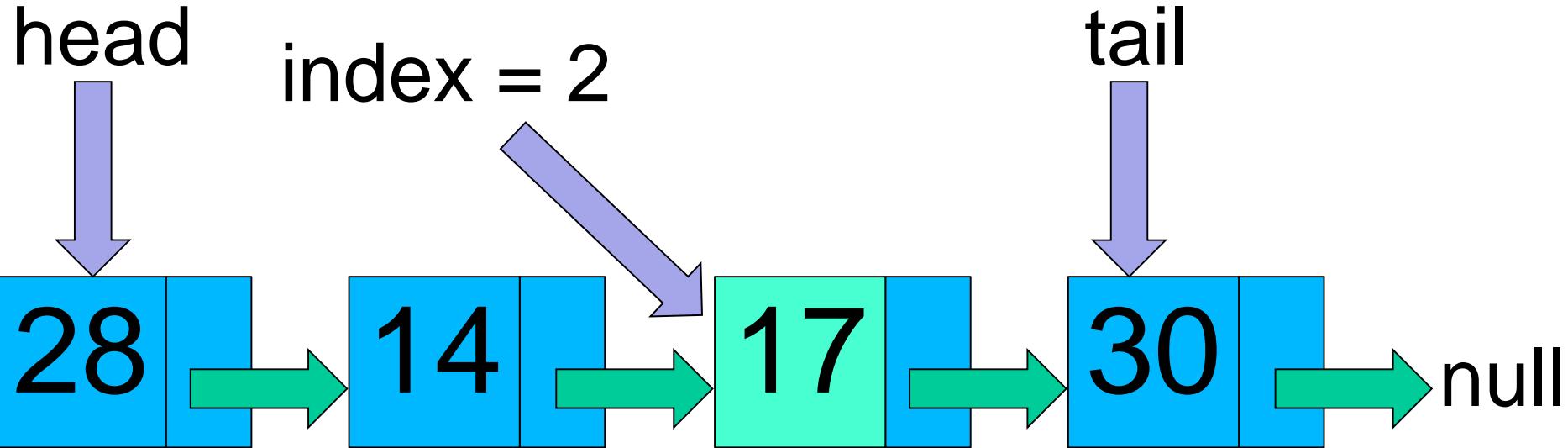
nums[0]	nums[1]	nums[2]	nums[3]
28	14	17	30





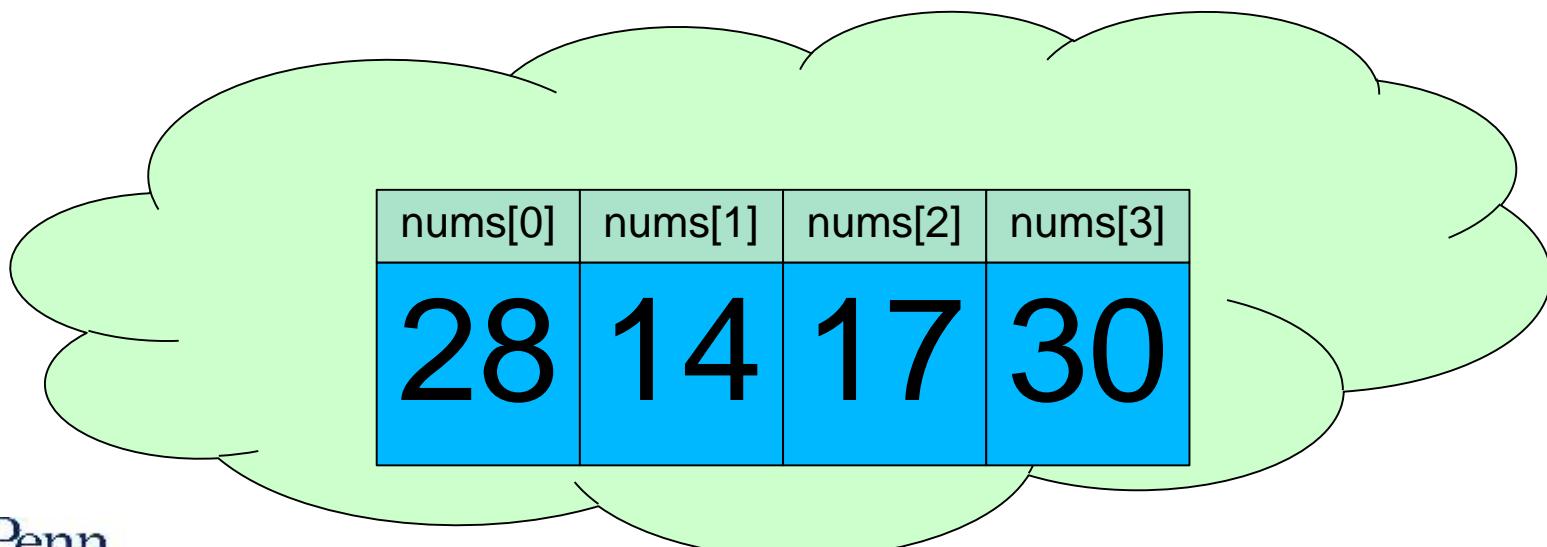
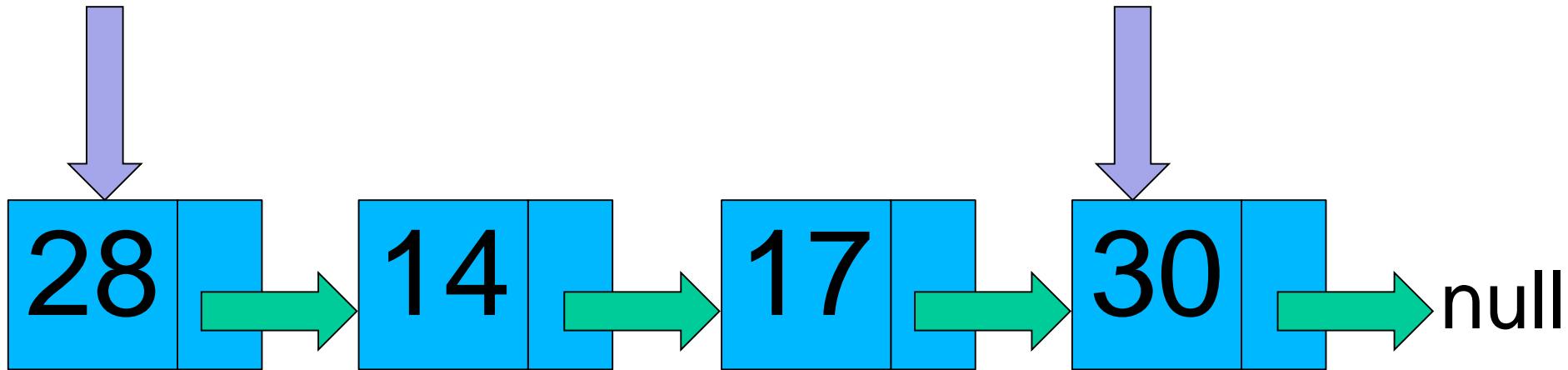






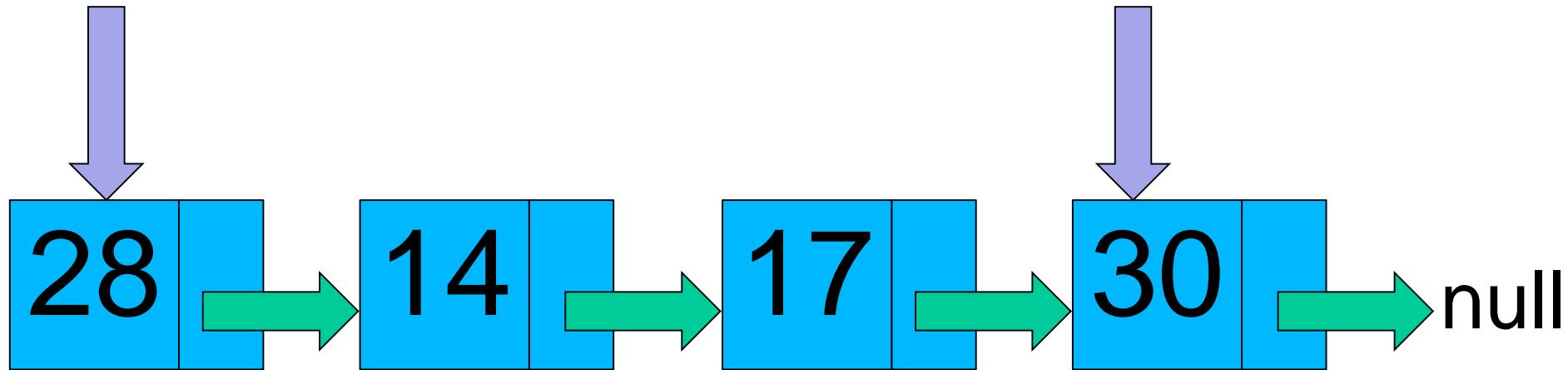
head

tail

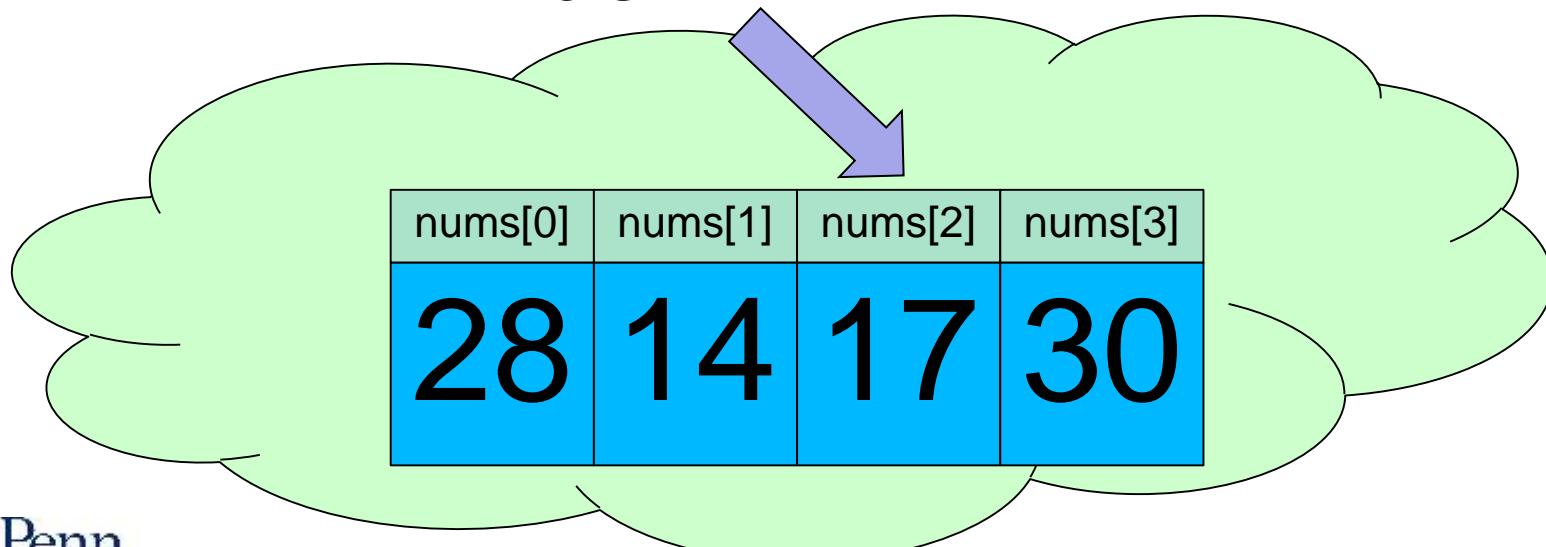


head

tail

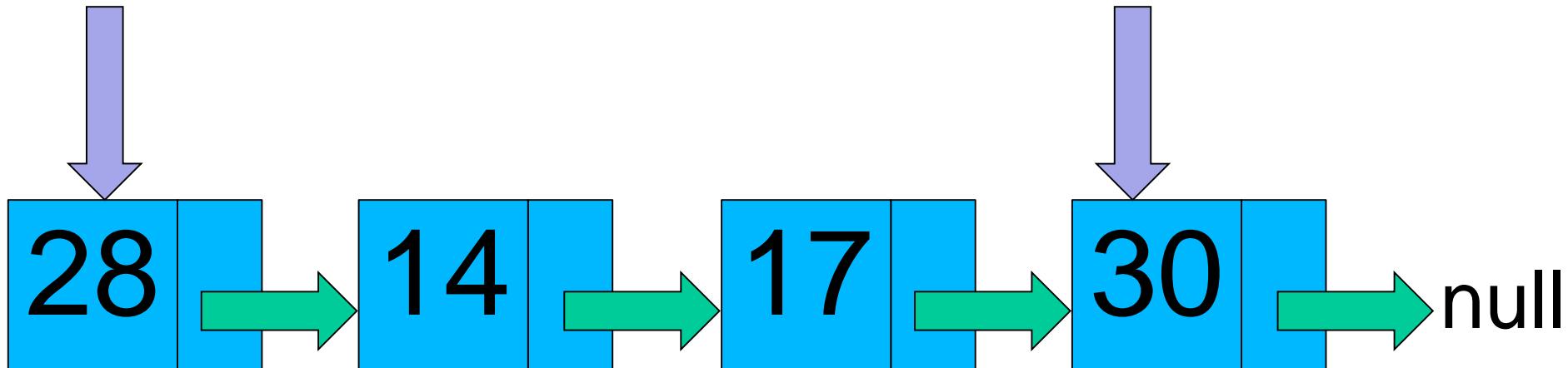


index = 2

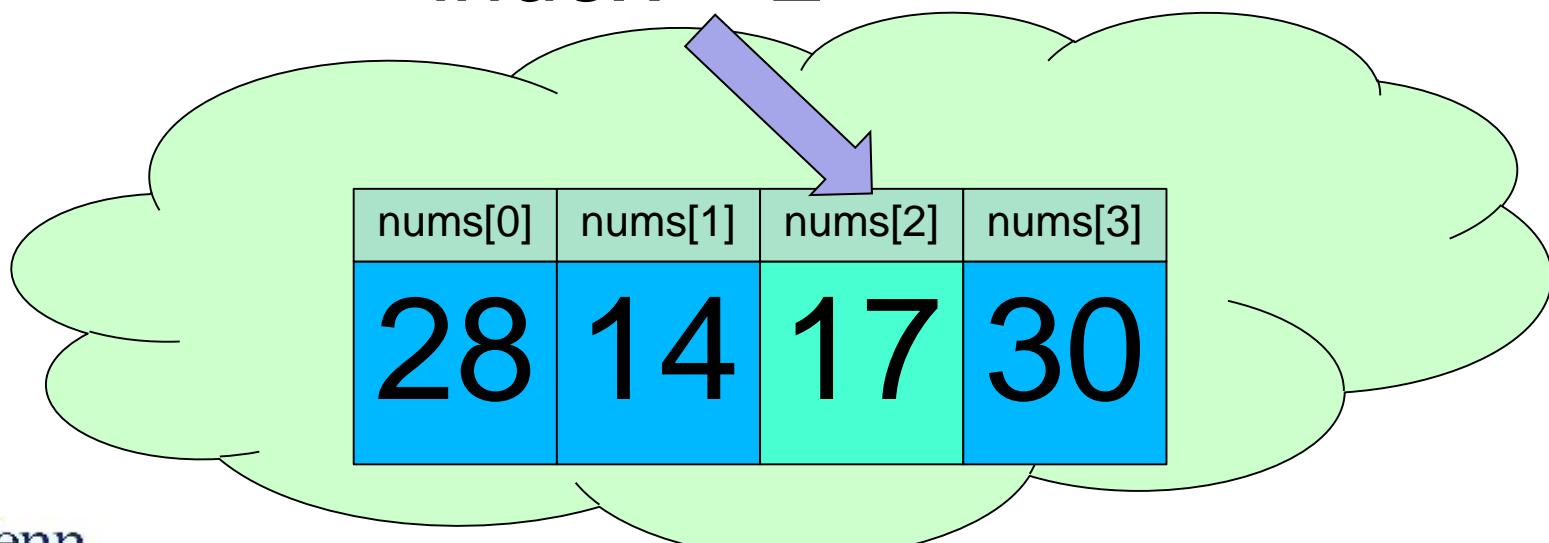


head

tail



index = 2



```
public class LinkedList {  
  
    protected Node head = null;  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            // error handling omitted  
            current = current.next;  
        }  
        return current.value;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            // error handling omitted  
            current = current.next;  
        }  
        return current.value;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            // error handling omitted  
            current = current.next;  
        }  
        return current.value;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public int getByIndex(int index) {  
        // error handling omitted  
        return values[index];  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            // error handling omitted  
            current = current.next;  
        }  
        return current.value;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public int getByIndex(int index) {  
        // error handling omitted  
        return values[index];  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public int getByIndex(int index) {  
        Node current = head;  
        for (int i = 0; i < index; i++) {  
            // error handling omitted  
            current = current.next;  
        }  
        return current.value;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public int getByIndex(int index) {  
        // error handling omitted  
        return values[index];  
    }  
    . . .
```

Comparing LinkedList vs ArrayList

- Let's say we have a LinkedList of n numbers
- And an ArrayList containing the same numbers
- In the **worst case**, how many steps do we need in order to find an element in the list by its index?
- LinkedList: n steps
- ArrayList: 1 step!

Big-O Notation

- A way of expressing the performance or complexity of an algorithm (in the worst case)
- Can be a measure of execution time or memory usage (“time complexity” or “space complexity”)
- Expressed as a function of the size of the data set

Common Big-O Expressions

- **O(1): Constant**

The algorithm's complexity is independent of the size of the data set

- **O(n): Linear**

The algorithm's complexity is linearly proportional to the size of the data set

- **O(n²): Quadratic**

The algorithm's complexity is proportional to the square of the size of the data set

Comparing LinkedList vs ArrayList

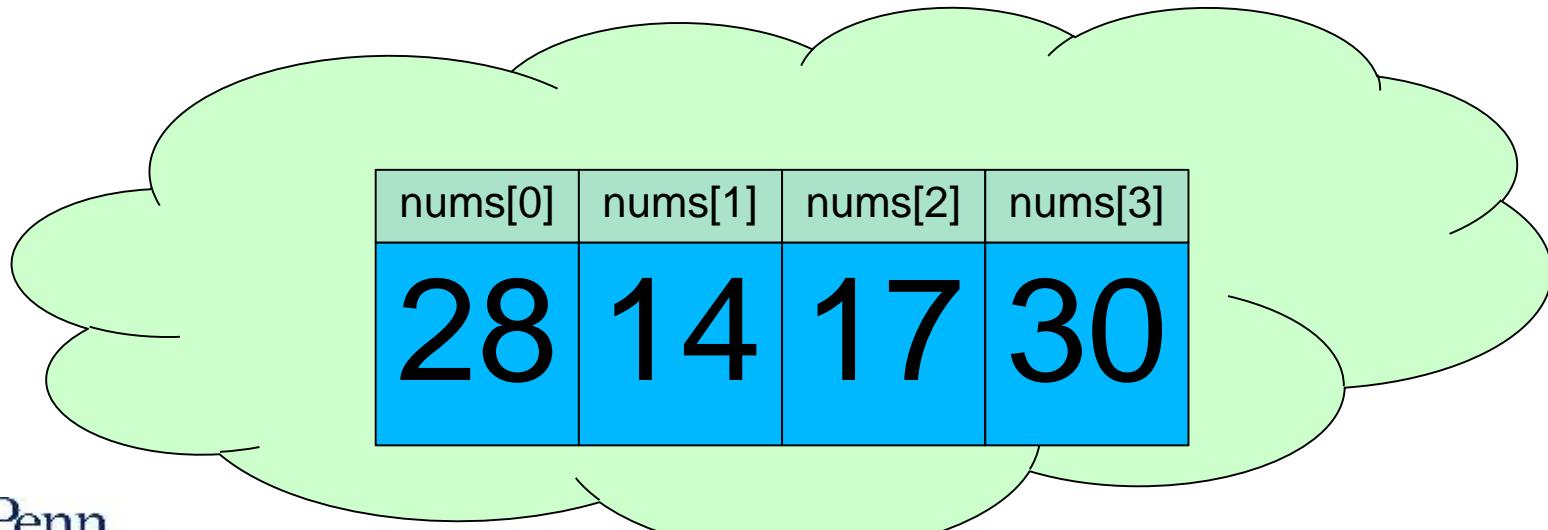
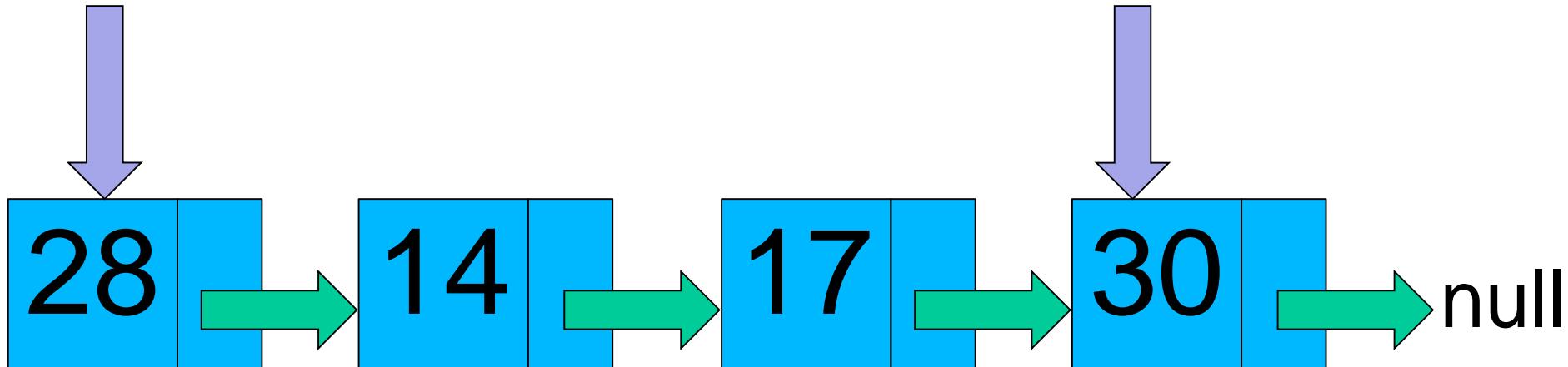
- Let's say we have a LinkedList of n numbers
- And an ArrayList containing the same numbers
- What is the Big-O complexity of finding an element in the list by its index?
- LinkedList: $O(n)$
- ArrayList: $O(1)$

Contains: LinkedList & ArrayList

- Let's say we have a LinkedList of n numbers
- And an ArrayList containing the same numbers
- In the **worst case**, how many steps do we need in order to determine if the List contains a given number?

head

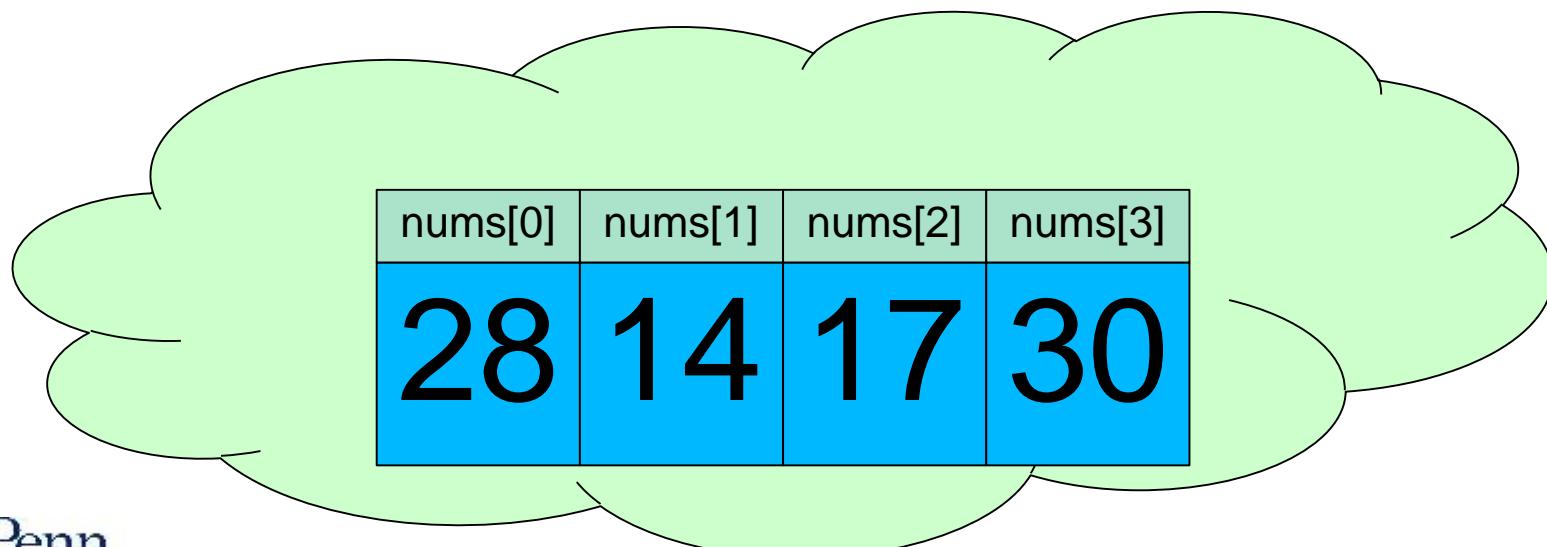
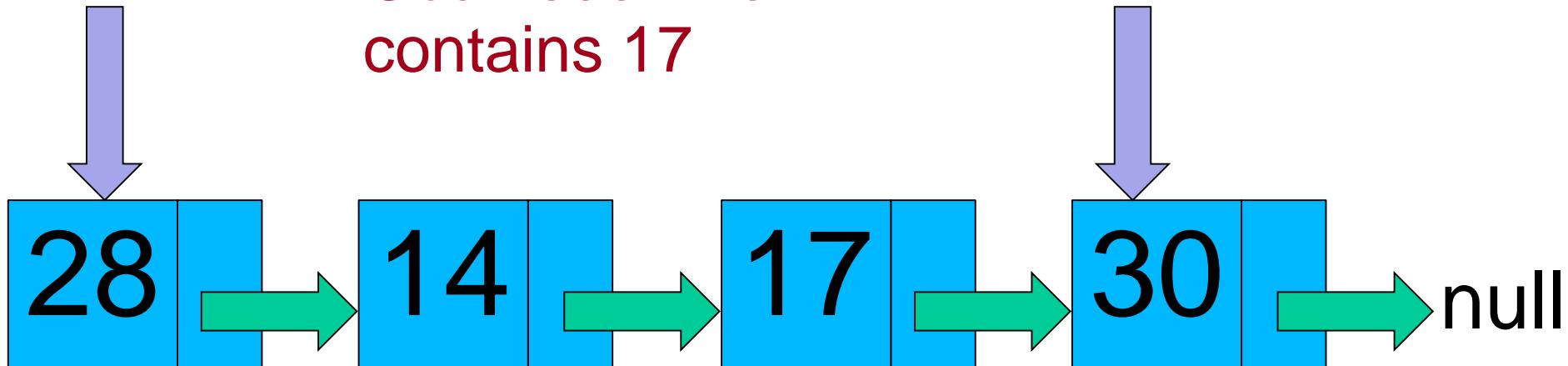
tail

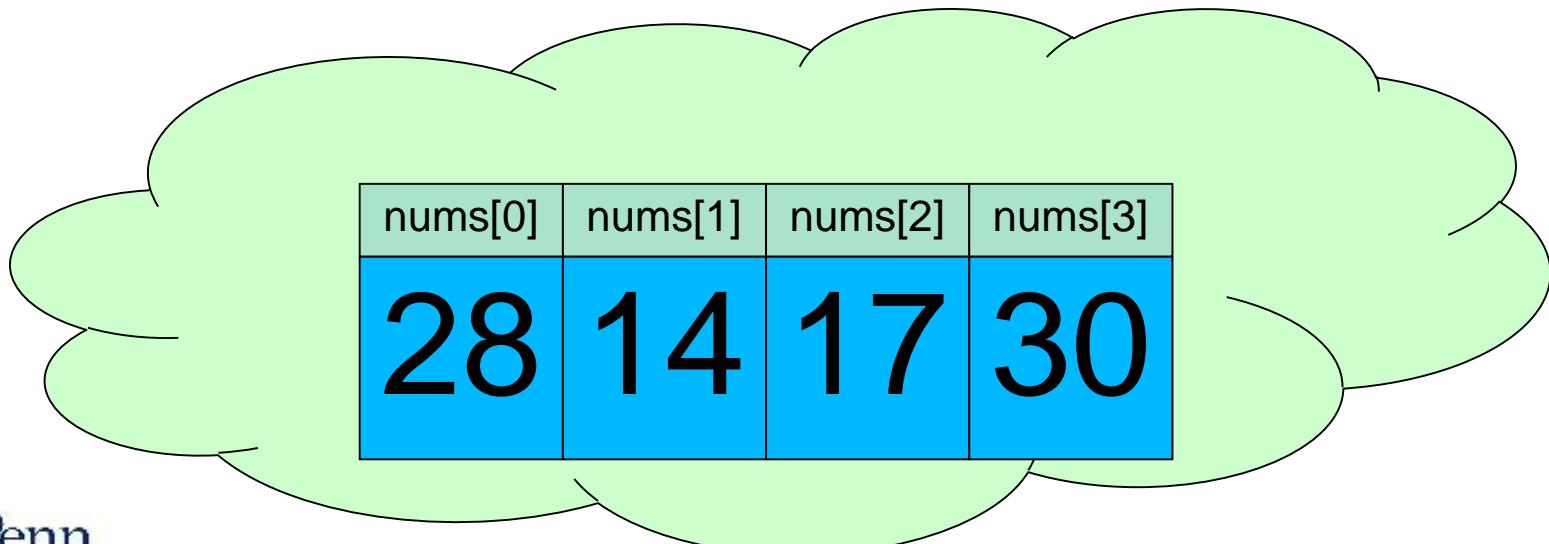
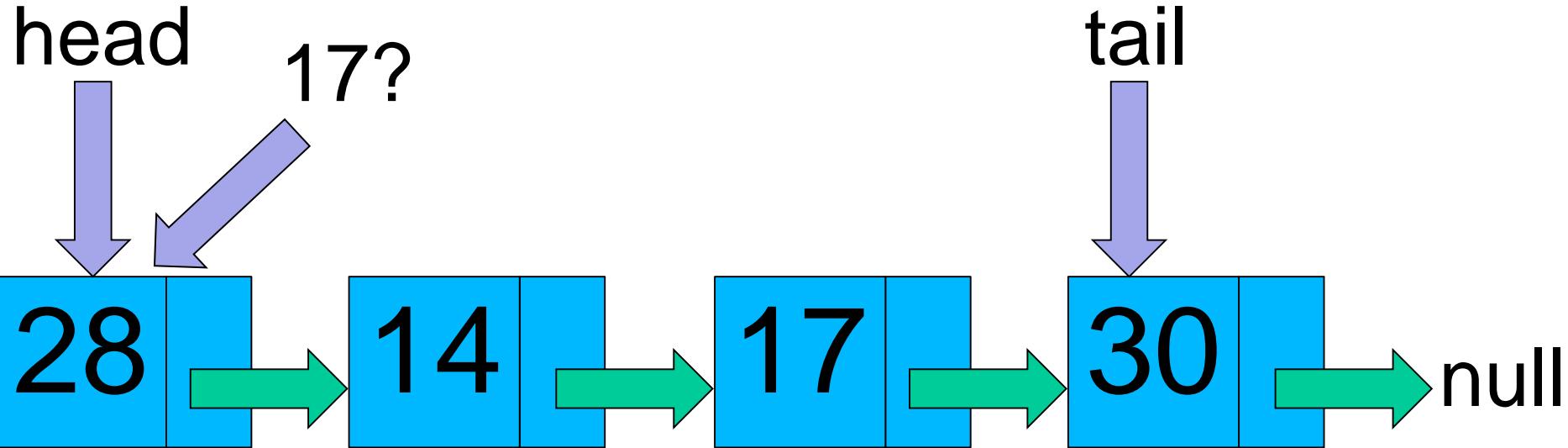


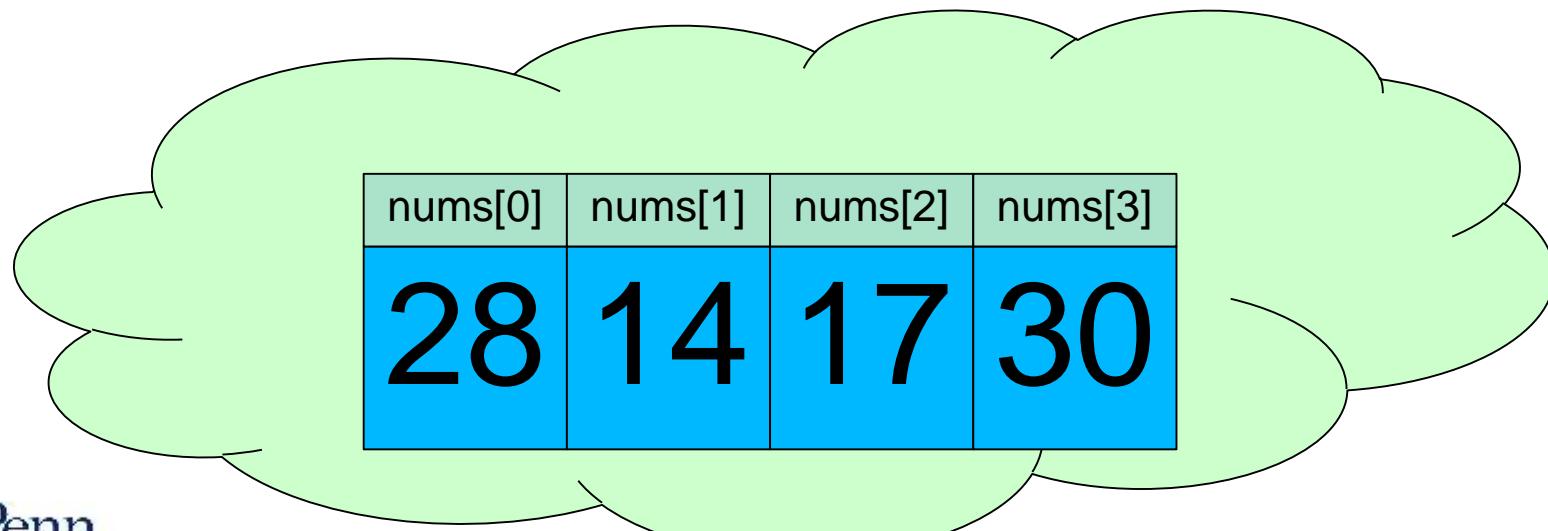
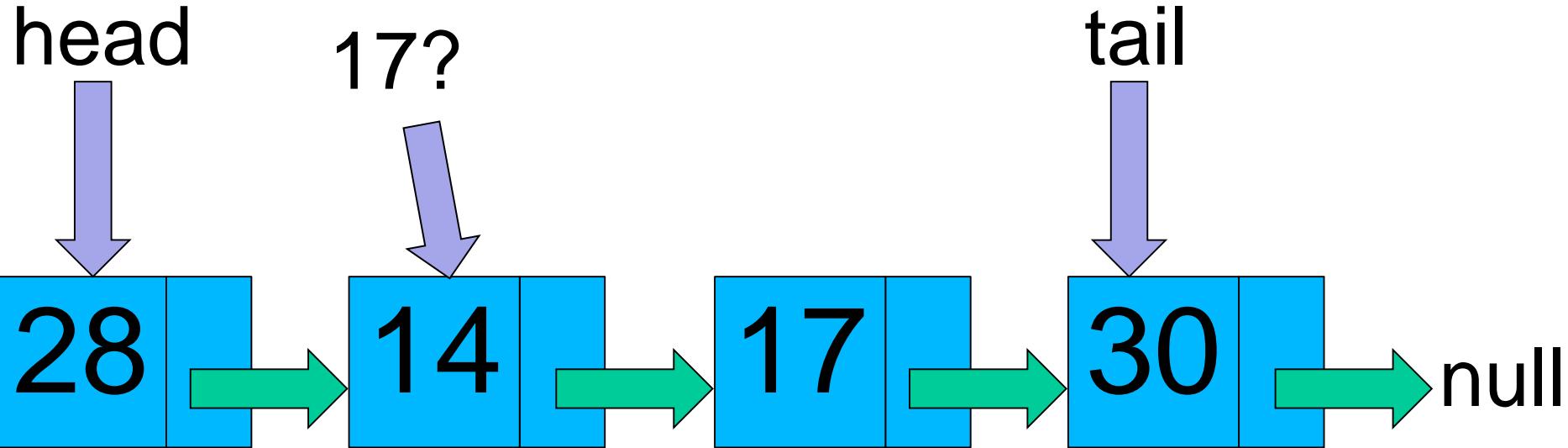
head

tail

Goal: see if list
contains 17



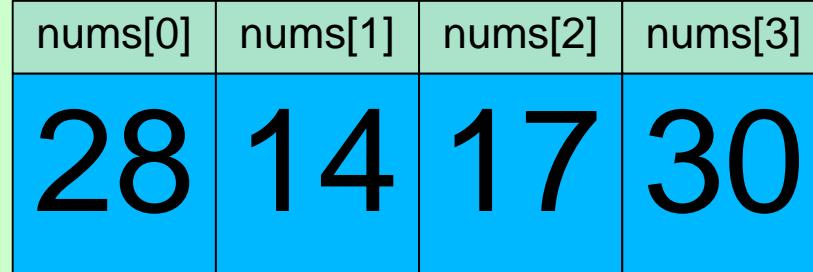
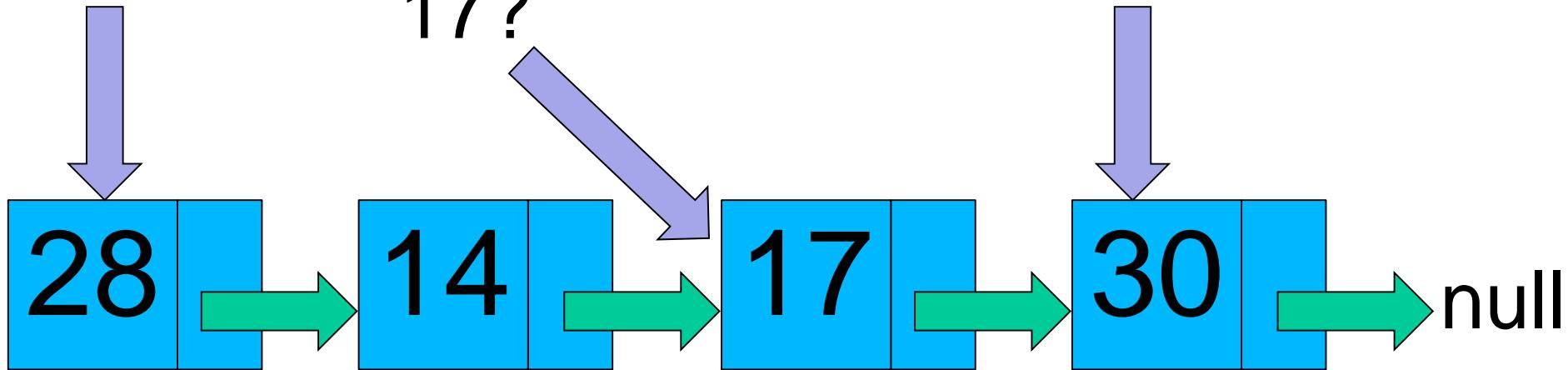




head

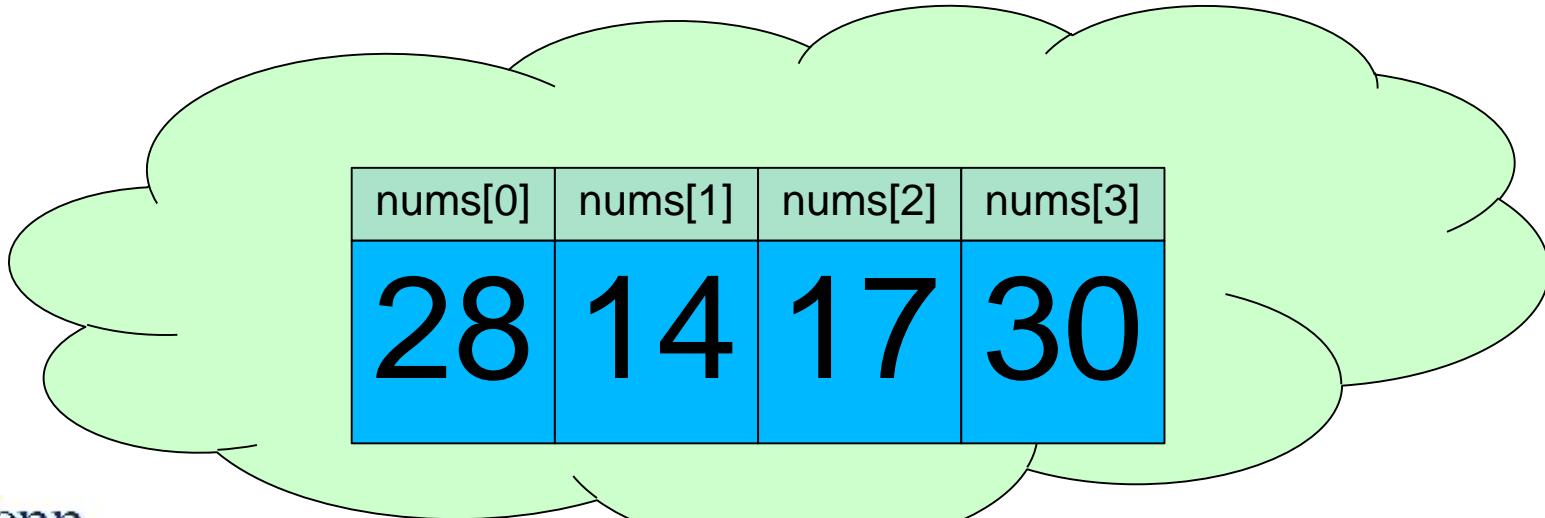
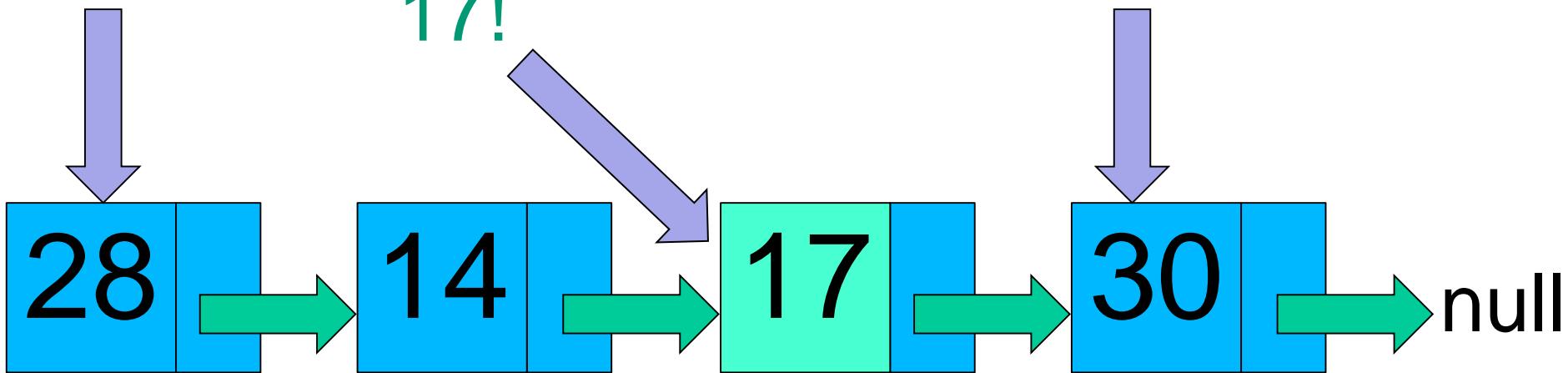
17?

tail



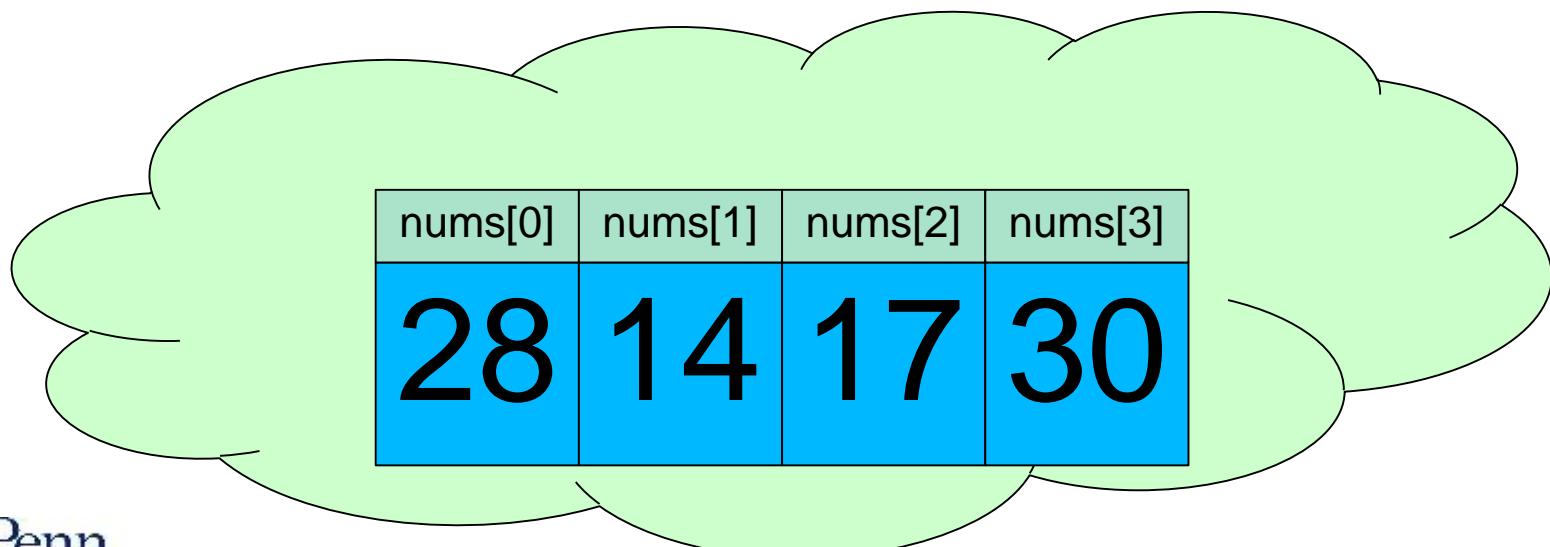
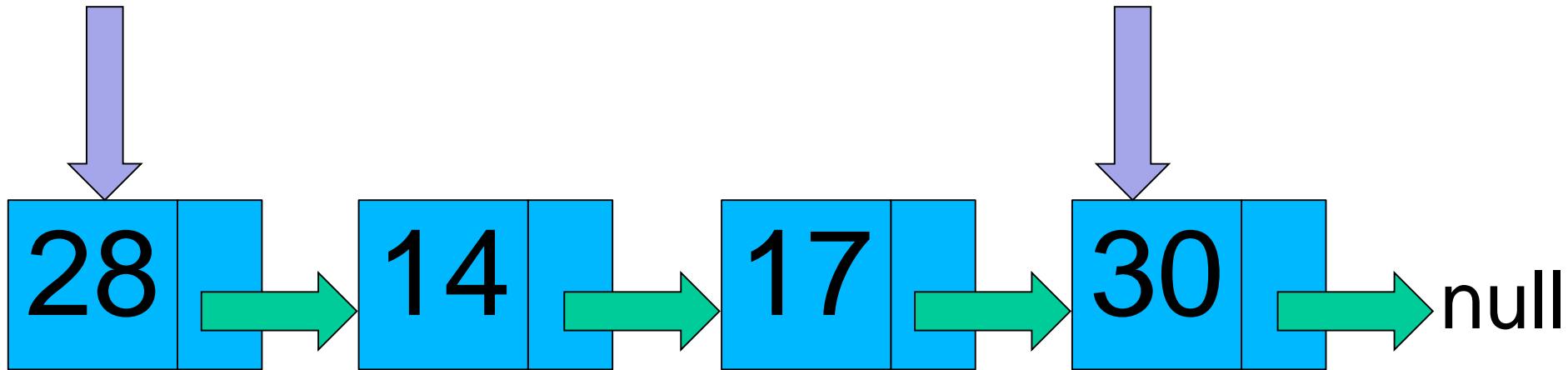
head

tail



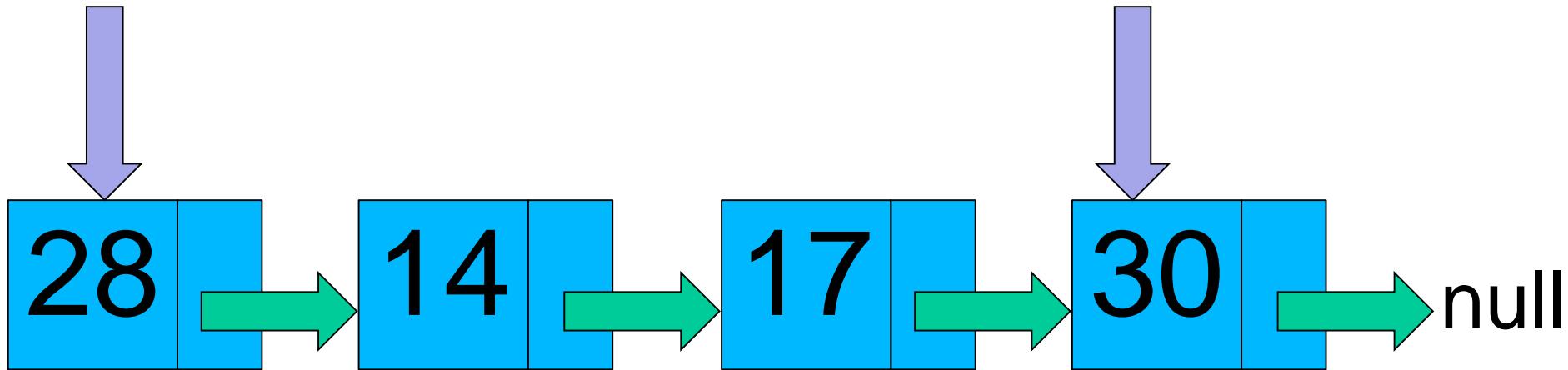
head

tail

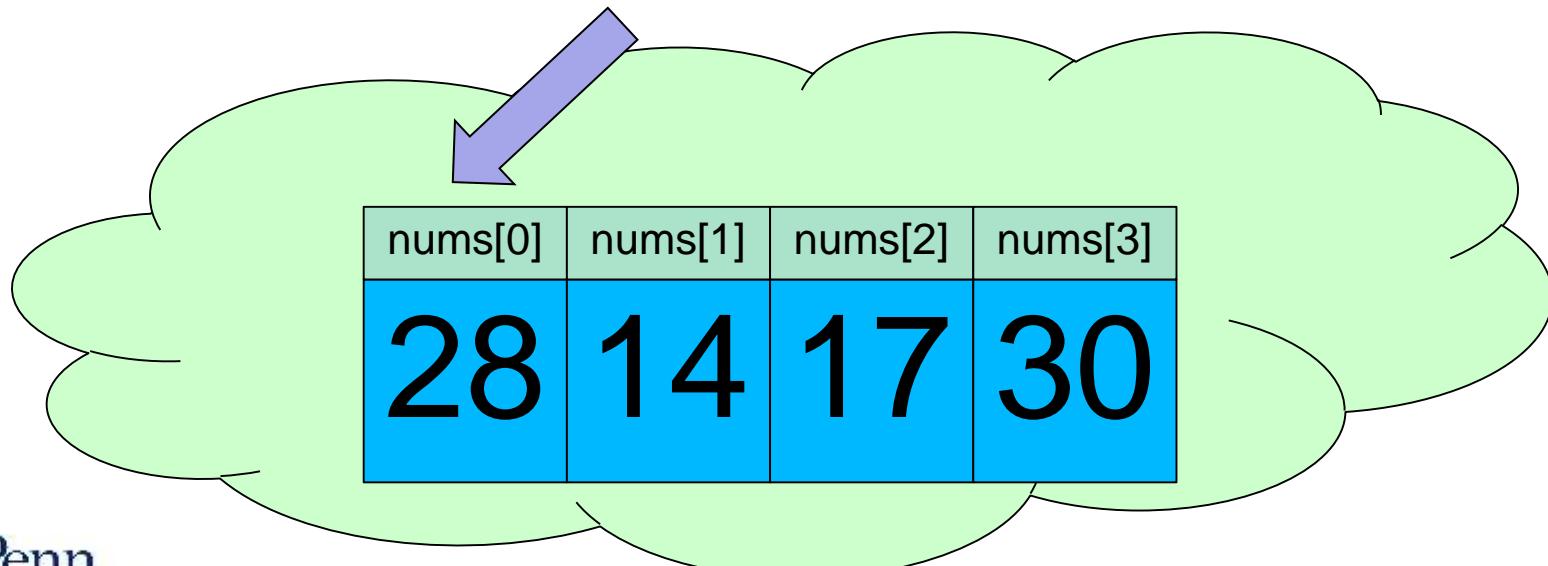


head

tail

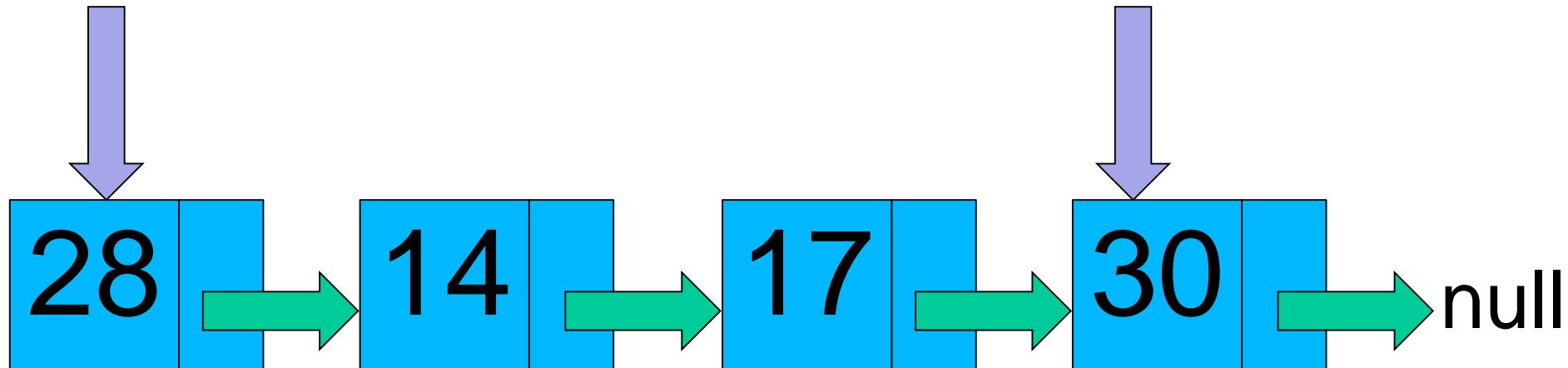


17?

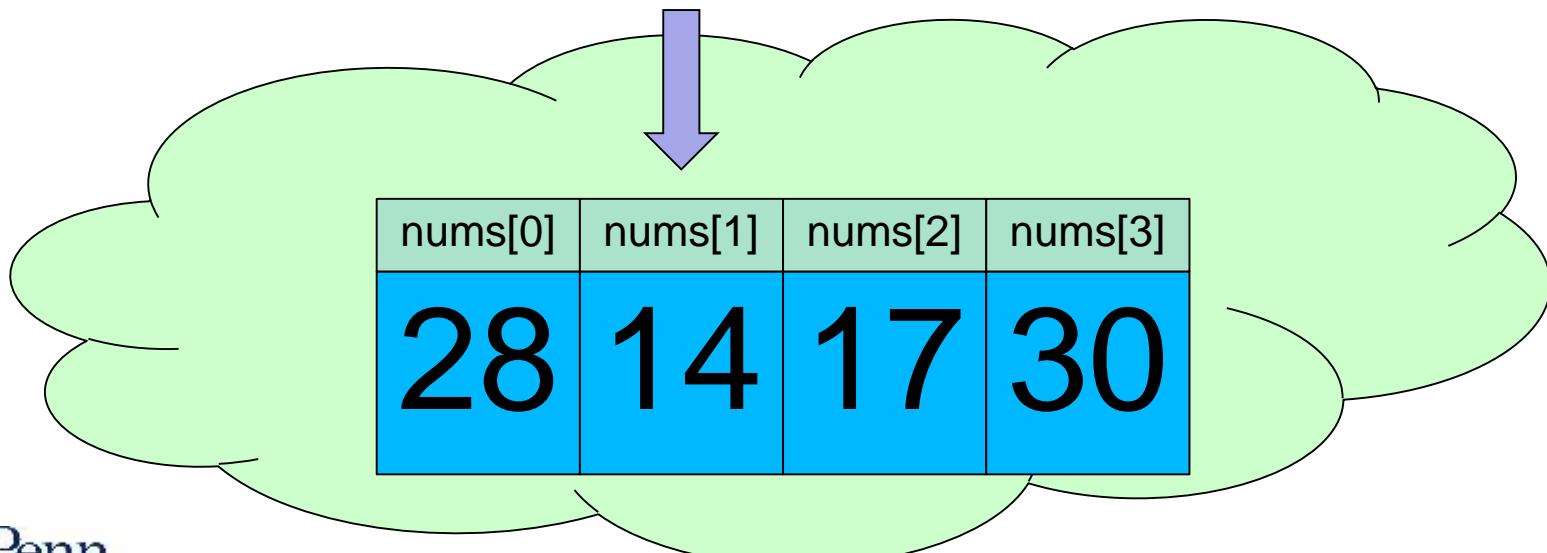


head

tail

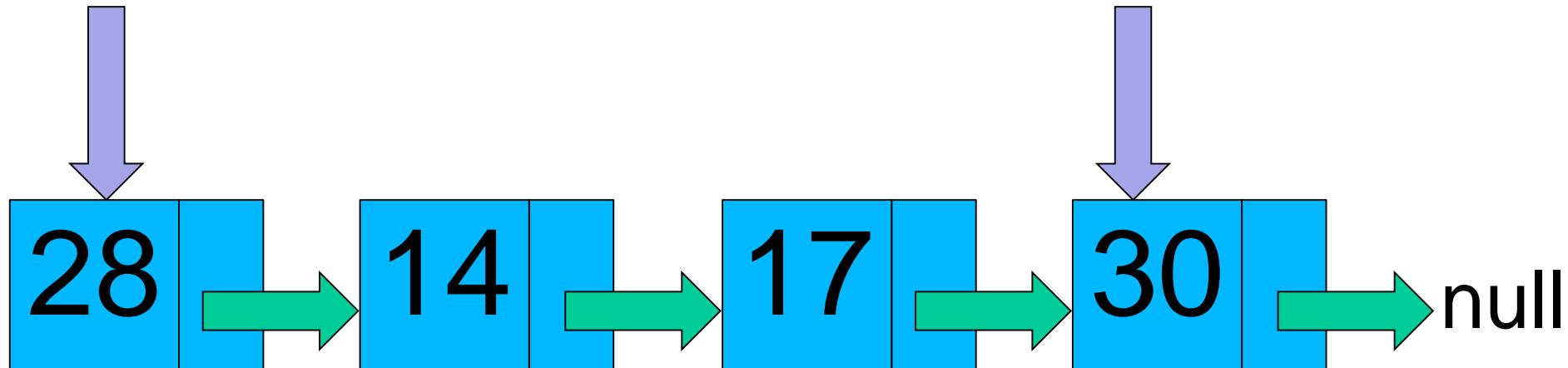


17?

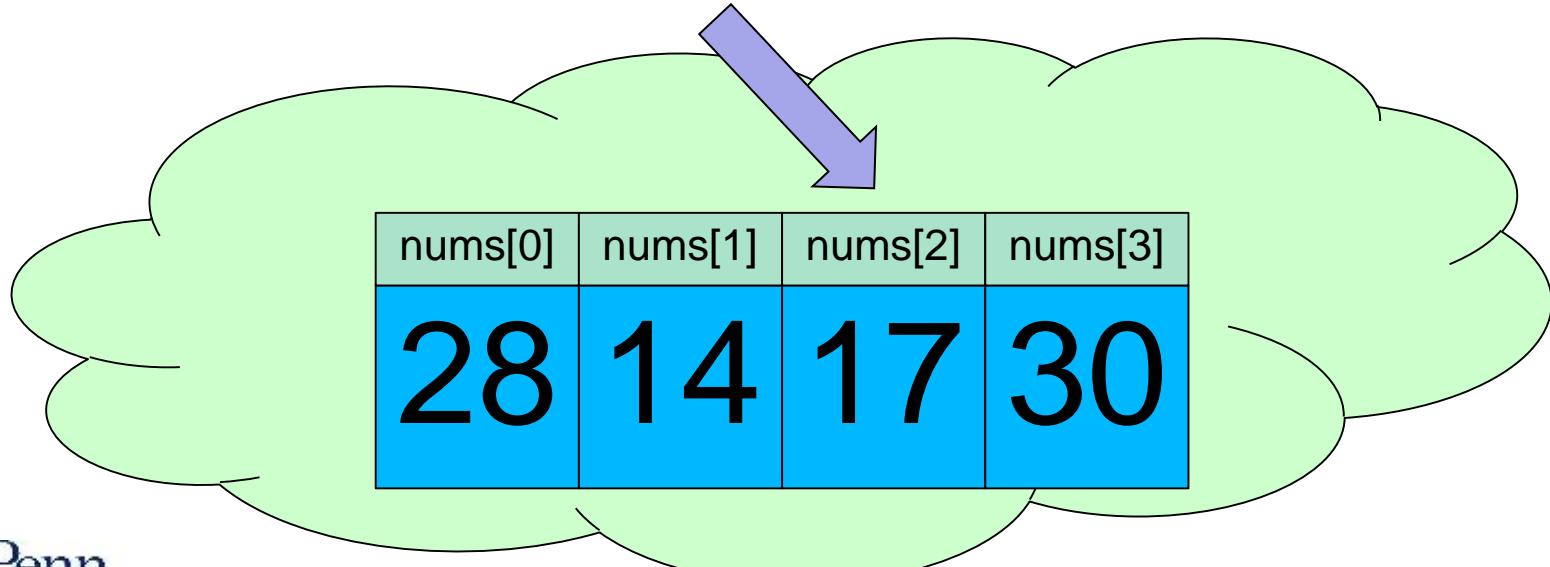


head

tail

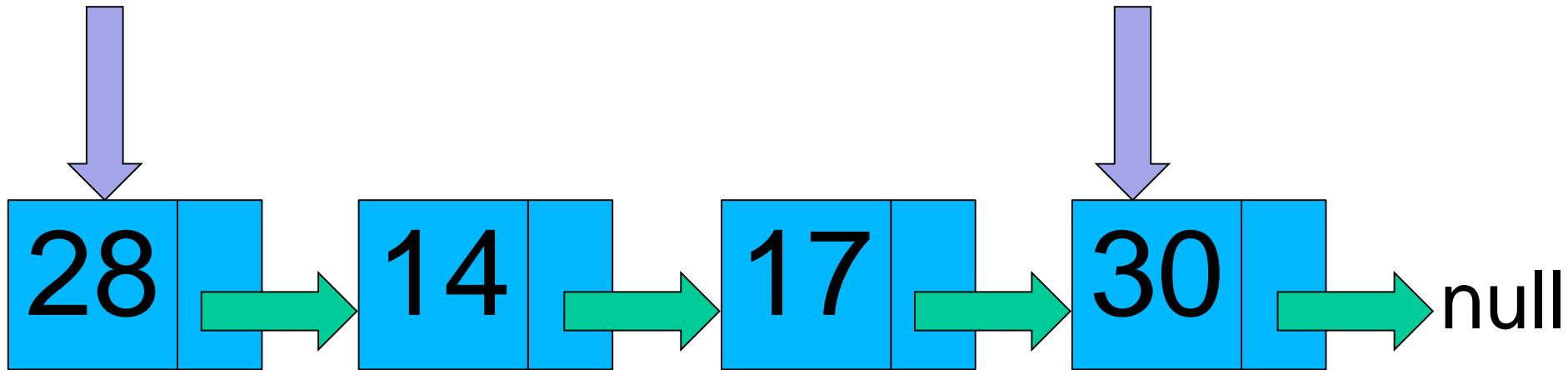


17?

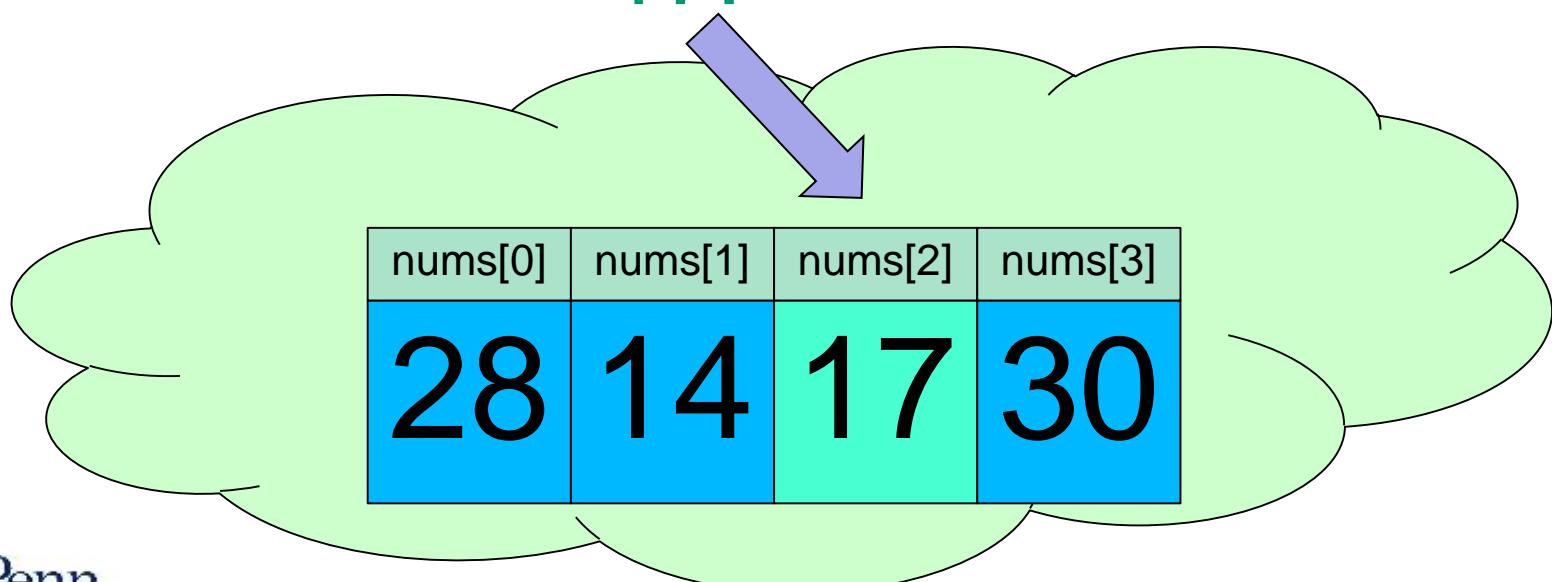


head

tail



17!



```
public class LinkedList {  
  
    protected Node head = null;  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) return true;  
            else current = current.next;  
        }  
        return false;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) return true;  
            else current = current.next;  
        }  
        return false;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) return true;  
            else current = current.next;  
        }  
        return false;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public boolean contains(int value) {  
        for (int i = 0; i < values.length; i++) {  
            if (values[i] == value) return true;  
        }  
        return false;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) return true;  
            else current = current.next;  
        }  
        return false;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public boolean contains(int value) {  
        for (int i = 0; i < values.length; i++) {  
            if (values[i] == value) return true;  
        }  
        return false;  
    }  
    . . .
```

```
public class LinkedList {  
  
    protected Node head = null;  
  
    public boolean contains(int value) {  
        Node current = head;  
        while (current != null) {  
            if (current.value == value) return true;  
            else current = current.next;  
        }  
        return false;  
    }  
    . . .
```

```
public class ArrayList {  
    protected int values[];  
  
    public boolean contains(int value) {  
        for (int i = 0; i < values.length; i++) {  
            if (values[i] == value) return true;  
        }  
        return false;  
    }  
    . . .
```

Contains: LinkedList & ArrayList

- Let's say we have a LinkedList of n numbers
- And an ArrayList containing the same numbers
- In the **worst case**, how many steps do we need in order to determine if the List contains a given number?
- LinkedList: $O(n)$
- ArrayList: $O(n)$
- Can we do better?

Improving upon Lists

- We could improve the efficiency of the “contains” functionality by assuming/enforcing certain restrictions
- Ideally, we’d want to know that the element we’re seeking could **only** be in one spot
- Or that the number of spots it could be in is independent of the total number of elements
- This would be $O(1)$
- This may come with some tradeoffs!

Sets

- A collection of elements such that:
 - no element is duplicated
 - an inserted element is not guaranteed to be placed in a certain position
 - an element's position is not guaranteed to remain the same
- LinkedLists and ArrayLists are **not** Sets
- Sets are not necessarily $O(1)$ but give us the possibility of being better than $O(n)$

SD2x1.8

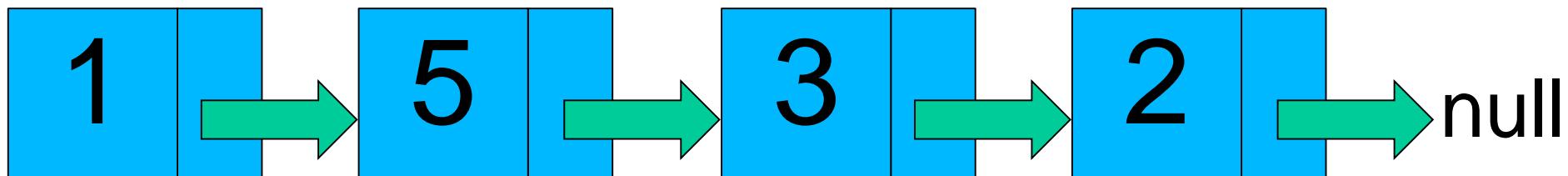
Hash Set structure

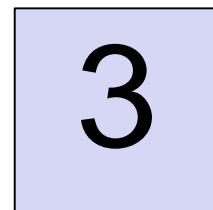
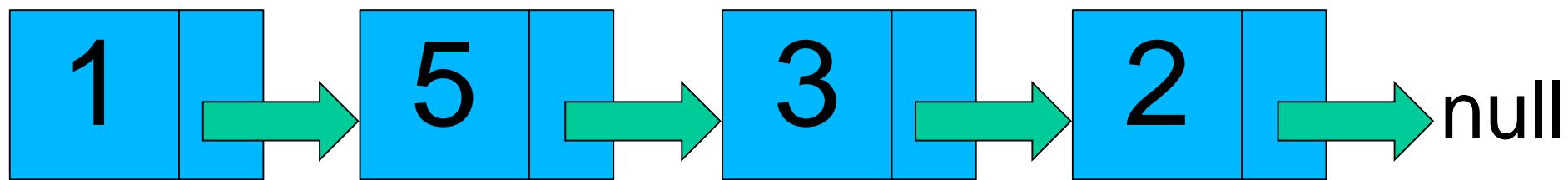
Chris

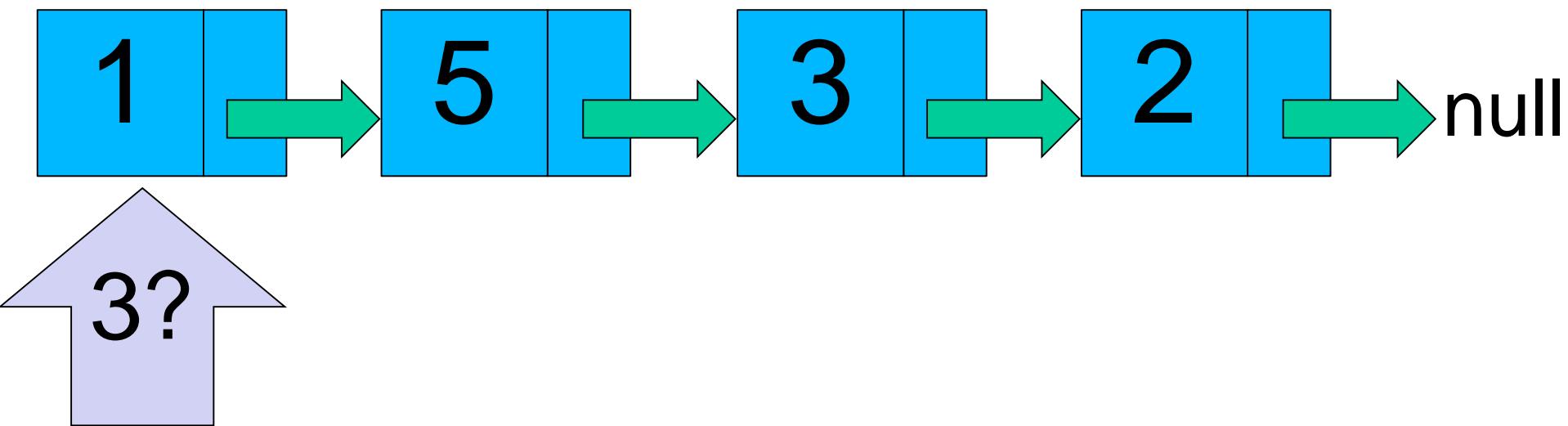
Finding a Number

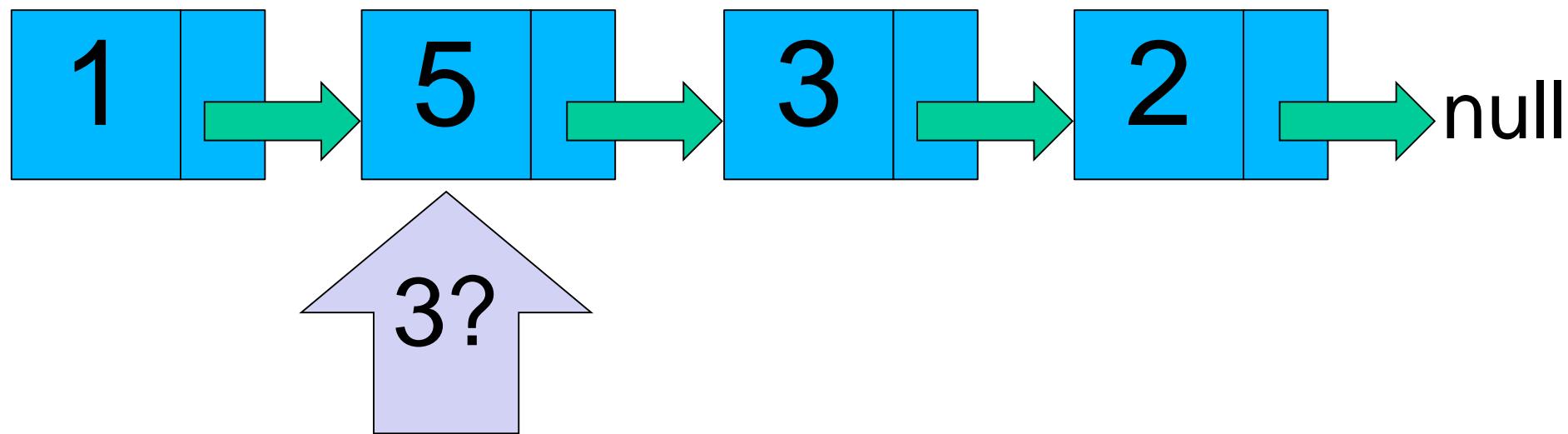
- Let's say we want to keep track of a collection of integers
- And then be able to determine whether the collection contains some integer
- We'll restrict ourselves to integers between 0 and 5

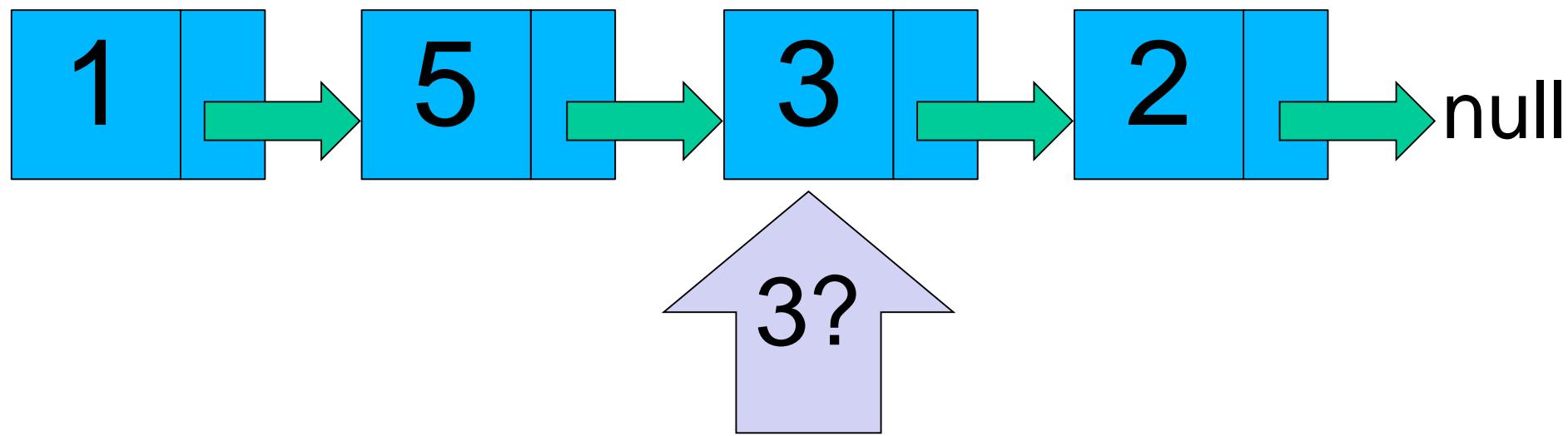
LinkedList Approach

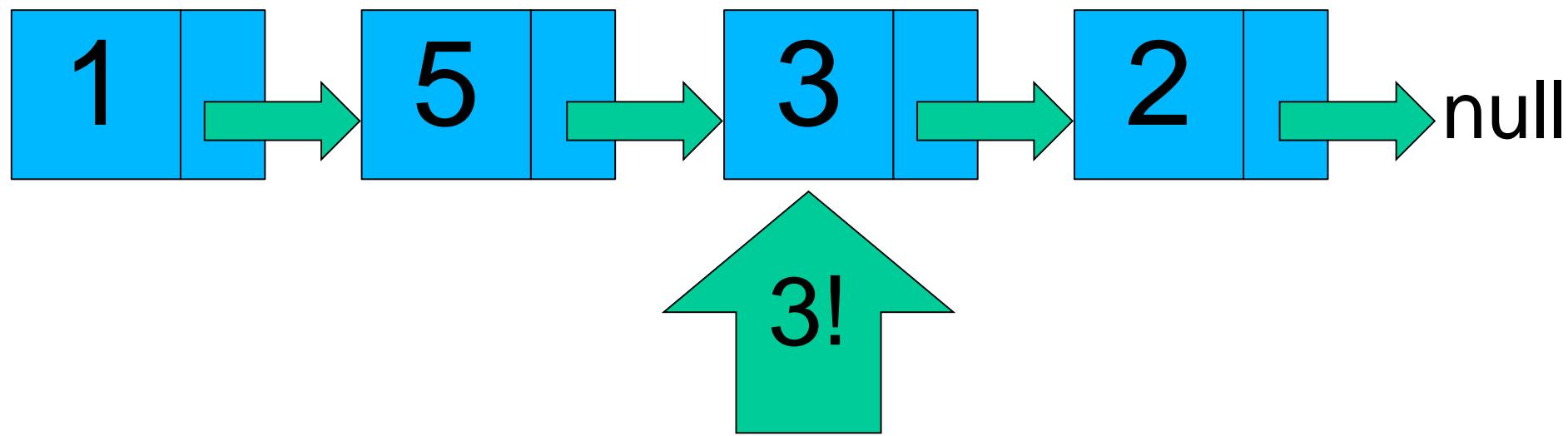


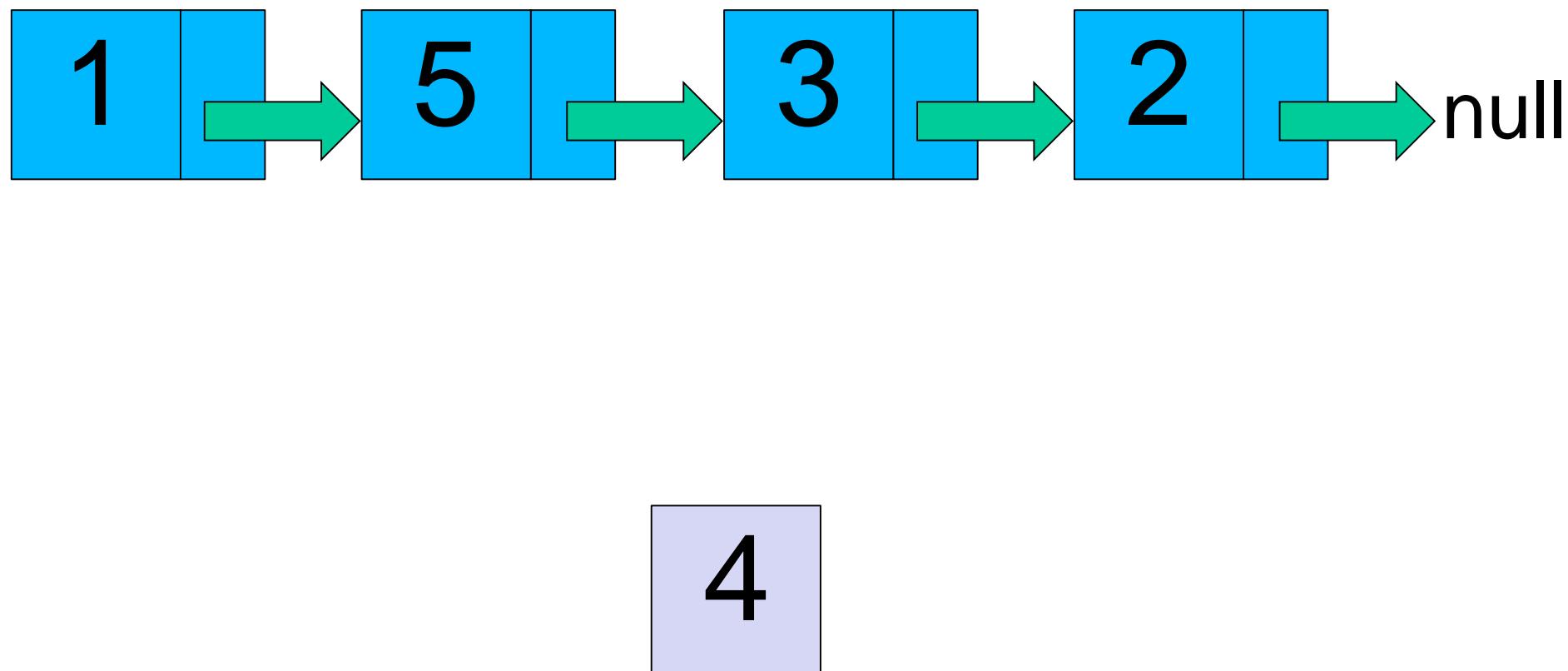


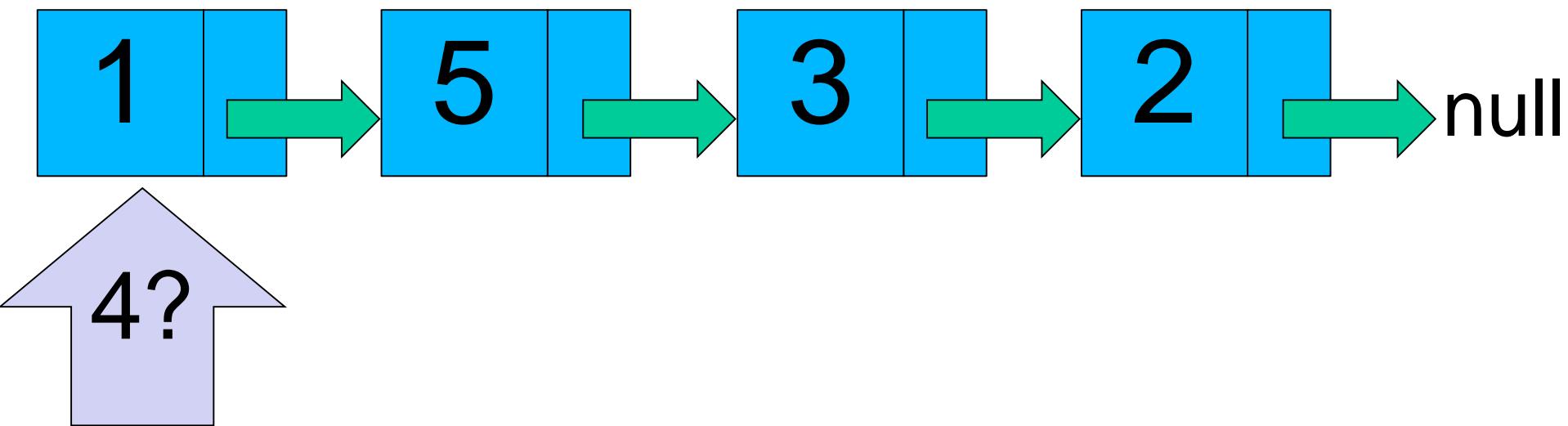


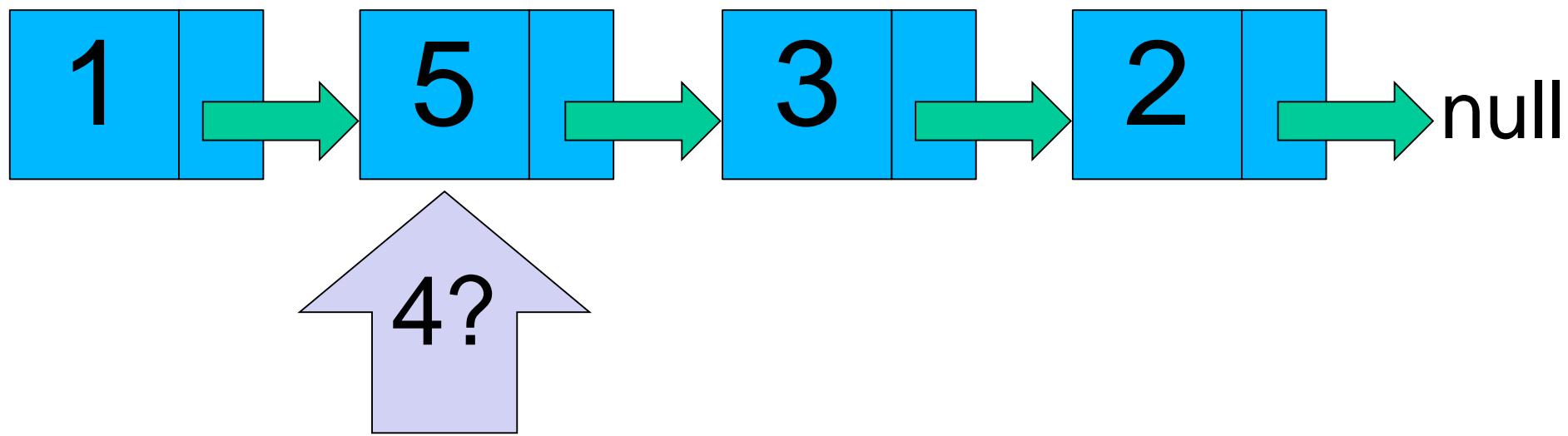


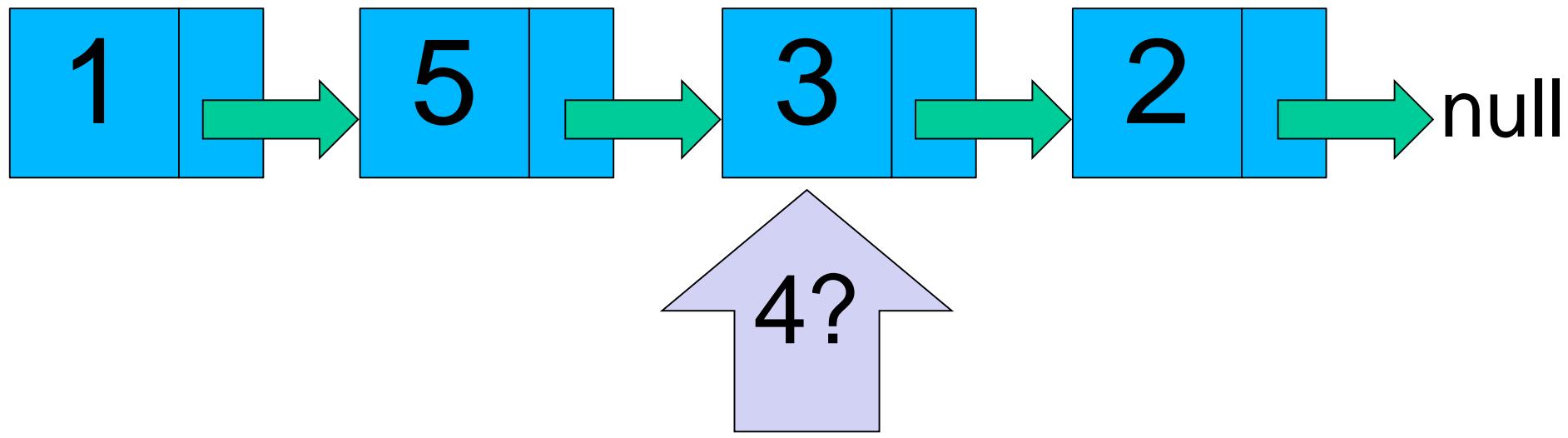


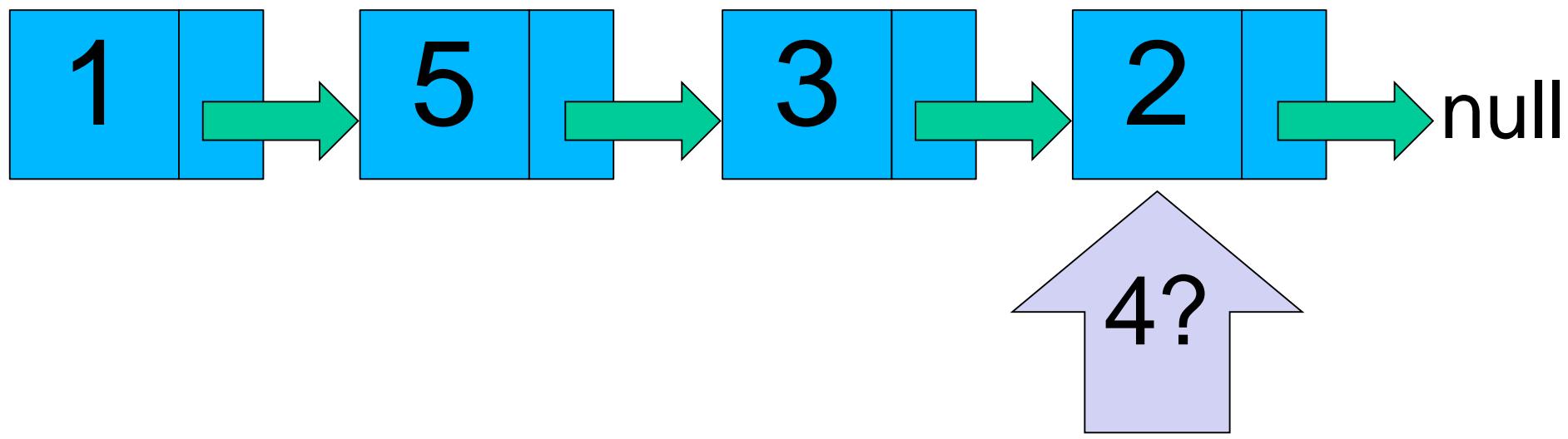


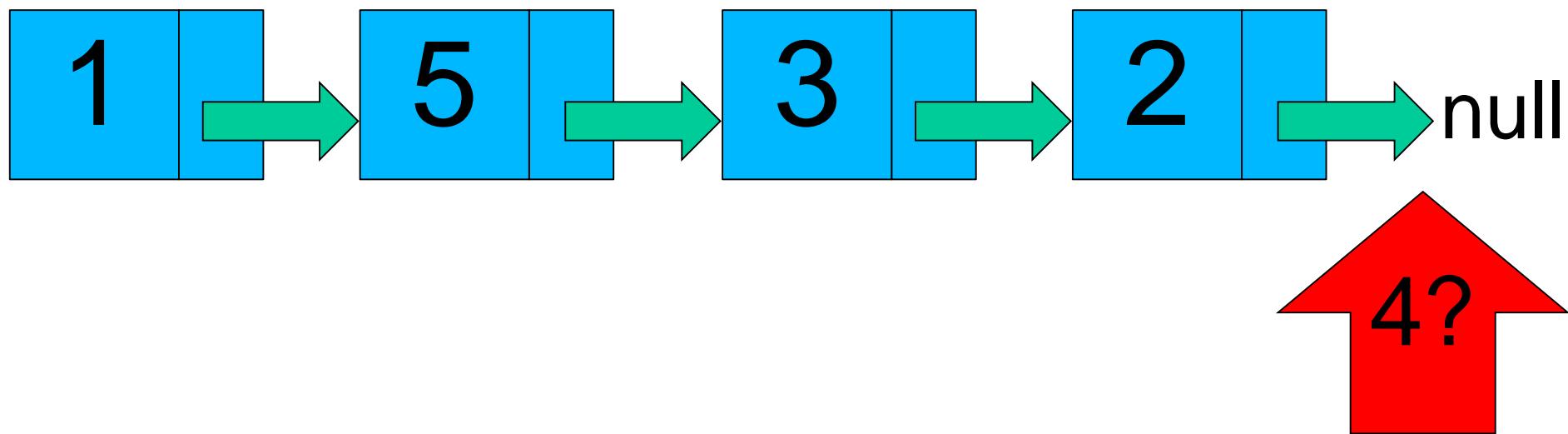


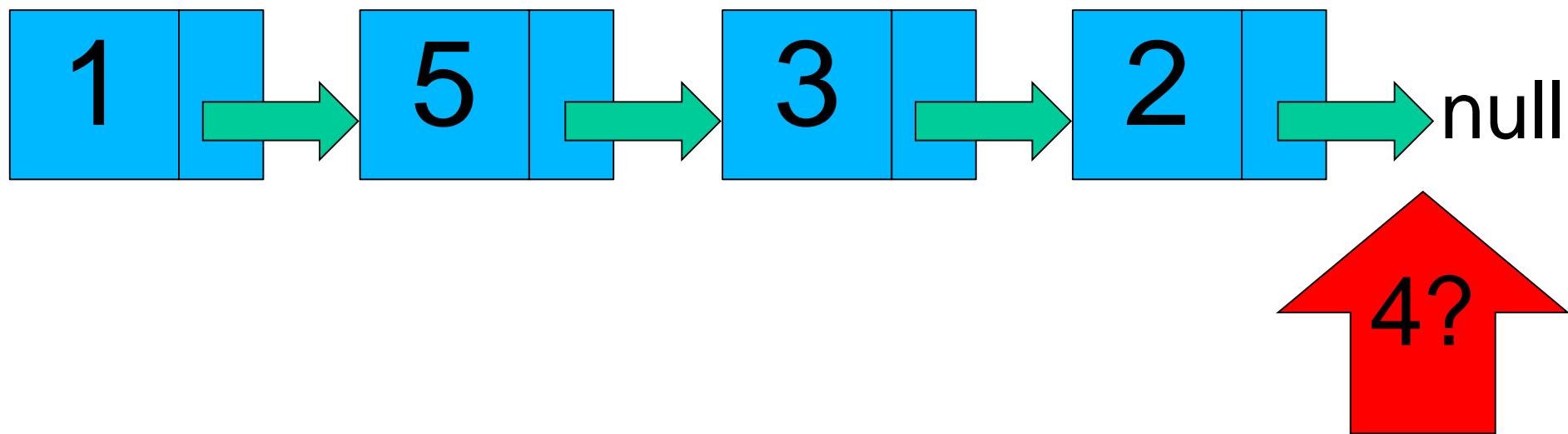












This is $O(n)$ in a LinkedList

0

1

2

3

4

5

false

true

true

true

false

true

3

0

1

2

3

4

5

false

true

true

true

false

true

3?

0

1

2

3

4

5

false

true

true

true

false

true

3?

0

1

2

3

4

5

false

true

true

true

false

true

3?

0

1

2

3

4

5

false

true

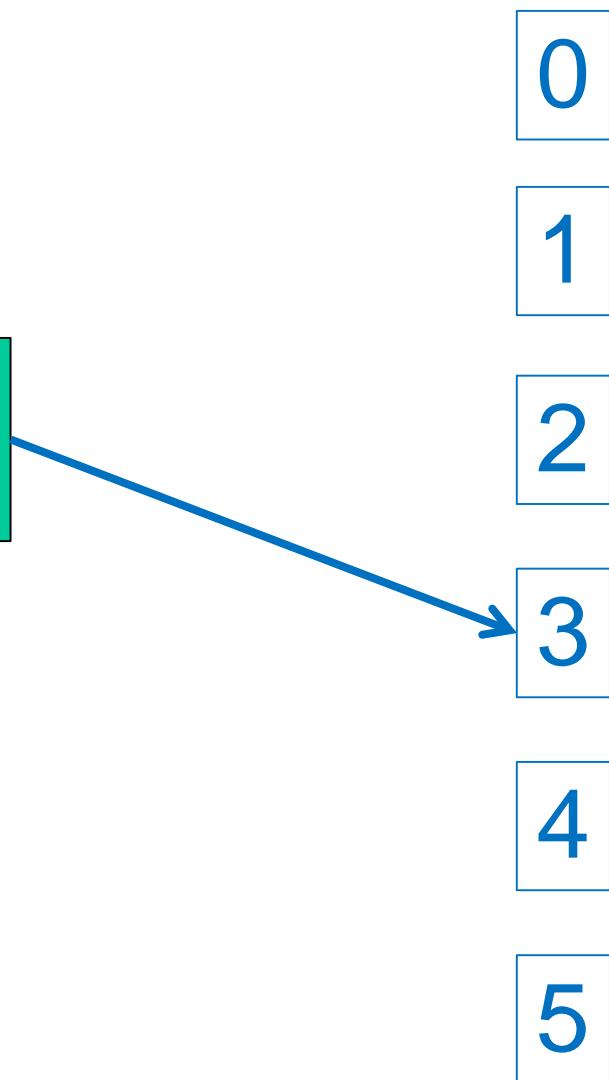
true

true

false

true

3!



false
true
true
true
false
true

4

0

1

2

3

4

5

false

true

true

true

false

true

4?

0

1

2

3

4

5

false

true

true

true

false

true

4?

0

1

2

3

4

5

false

true

true

true

false

true

4?

0

1

2

3

4

5

false

true

true

true

false

true

4?

0

1

2

3

4

5

false

true

true

true

false

true

4?

Now this is O(1)!

0

1

2

3

4

5

false

true

true

true

false

true

How would we do
this for Strings?

0	false
1	true
2	true
3	true
4	false
5	true

Hash Sets!

HashSet class definition

```
public class HashSet {  
    private String[] values;  
  
    public HashSet(int size) {  
        values = new String[size];  
    }  
    . . .
```

HashSet class definition

```
public class HashSet {  
  
    private String[] values;  
  
    public HashSet(int size) {  
        values = new String[size];  
    }  
  
    . . .
```

HashSet class definition

```
public class HashSet {  
  
    private String[] values;  
  
    public HashSet(int size) {  
        values = new String[size];  
    }  
  
    . . .  
}
```

How do we add a value to the hash set?

HashSet: Adding a value

0

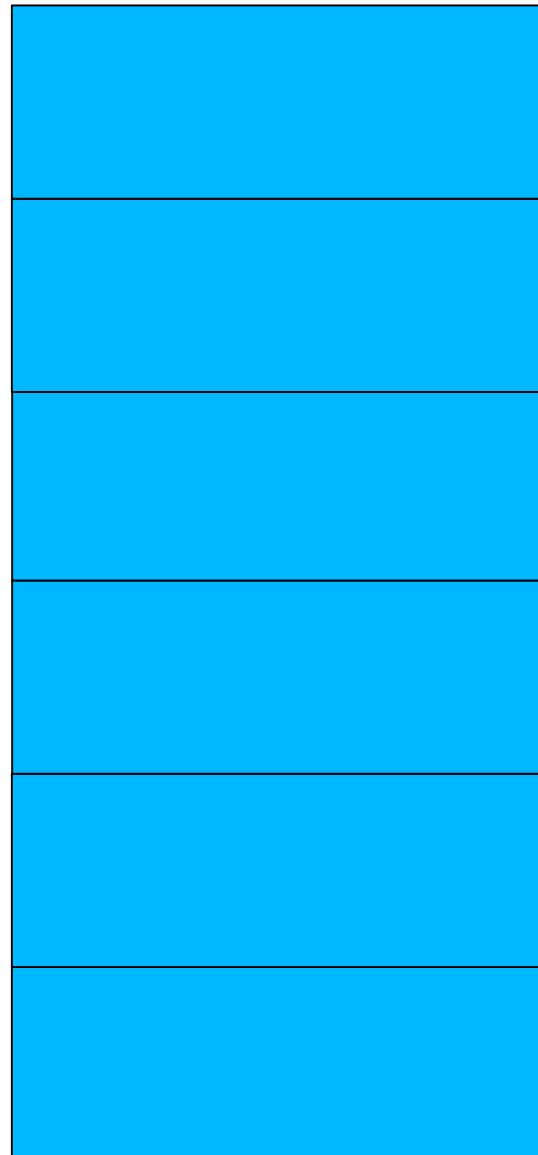
1

2

3

4

5



HashSet: Adding a value

“dog”

0

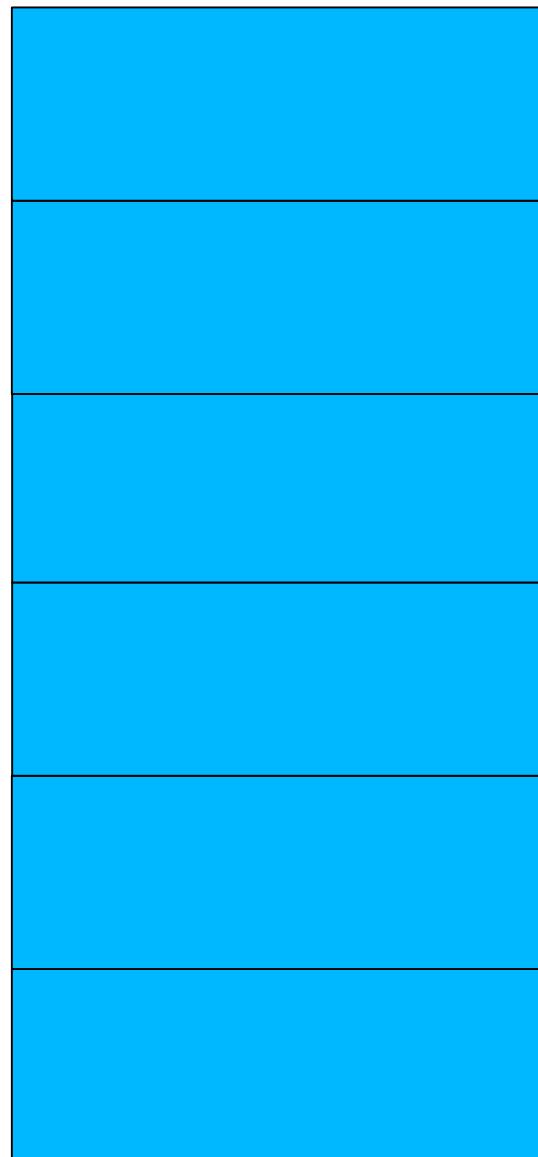
1

2

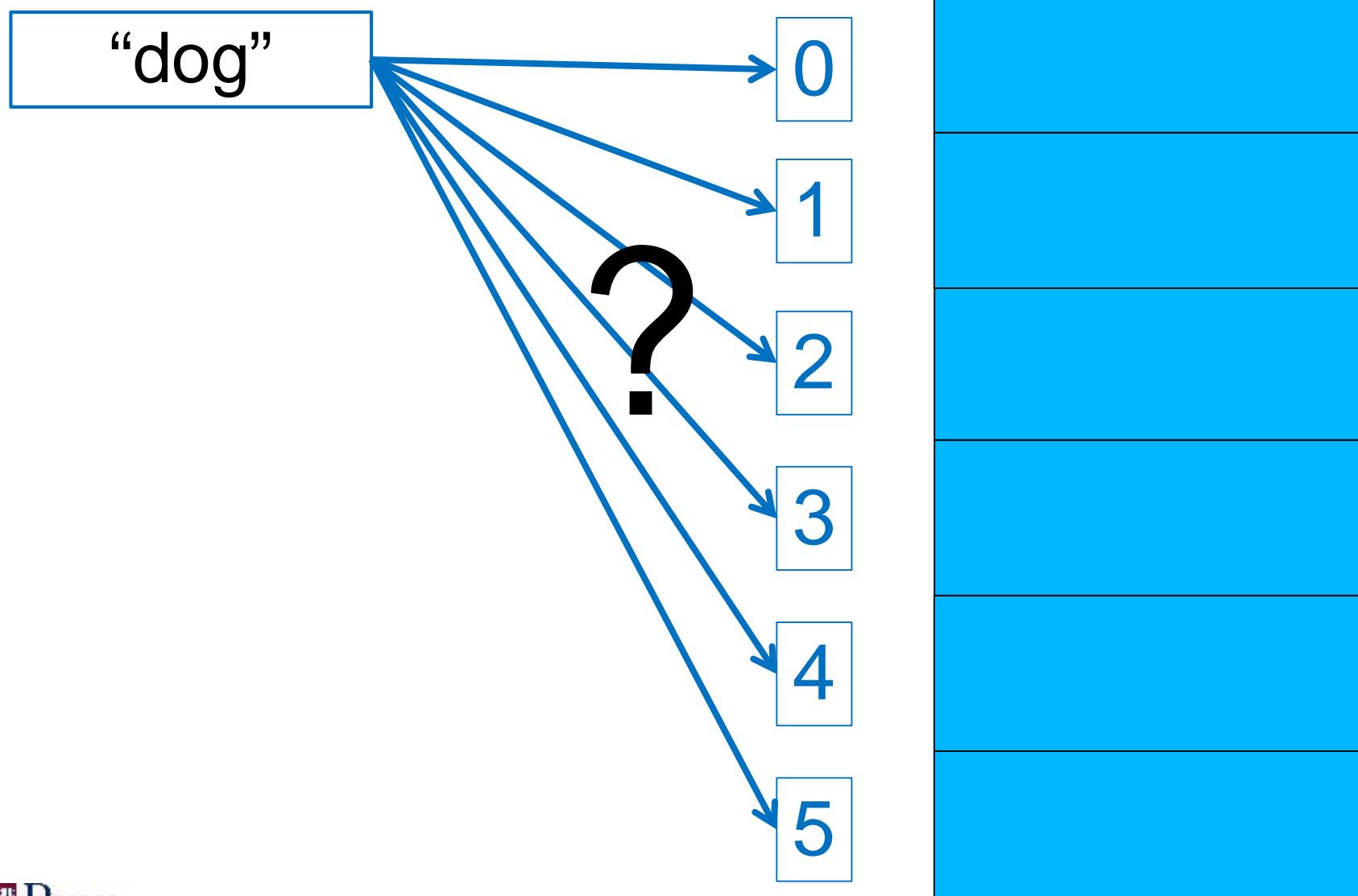
3

4

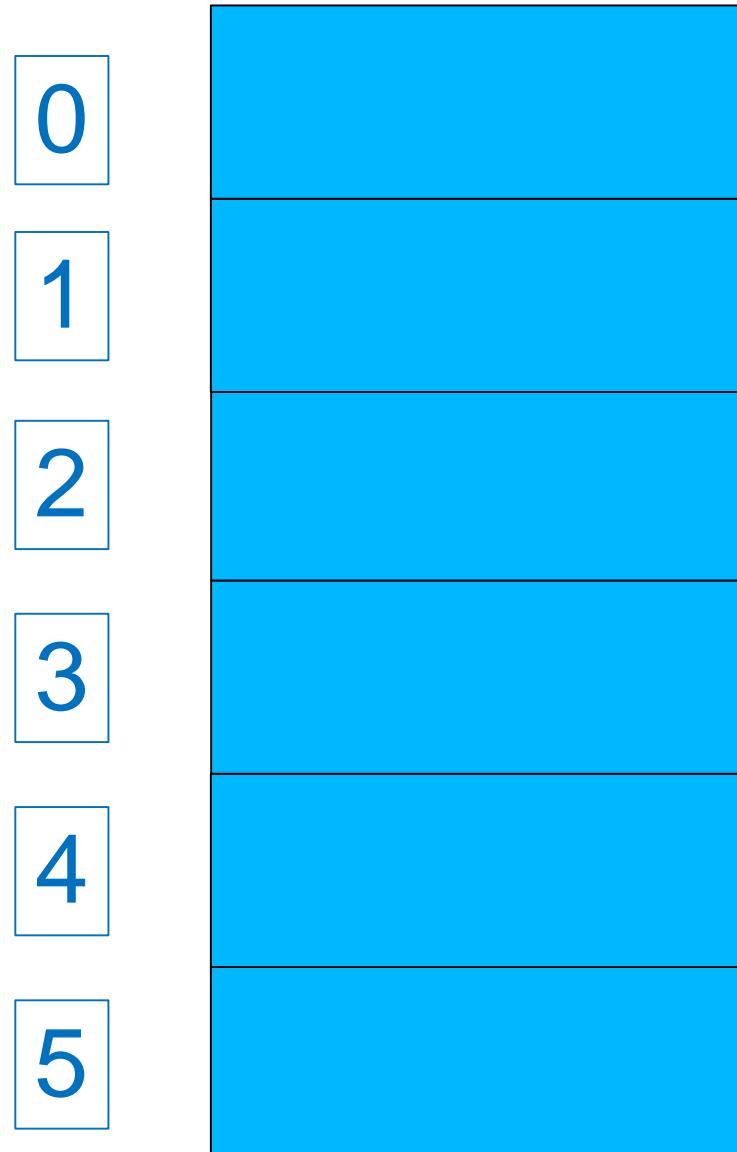
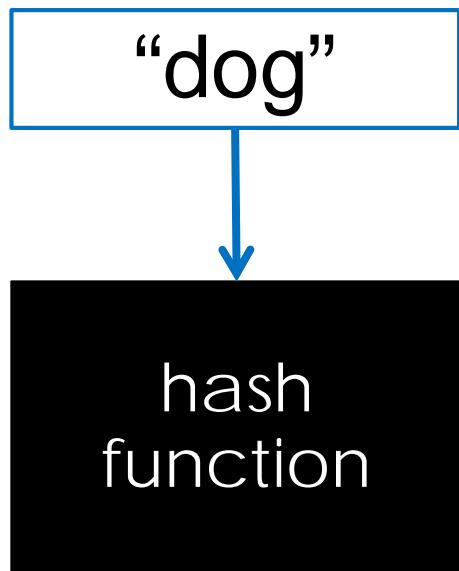
5



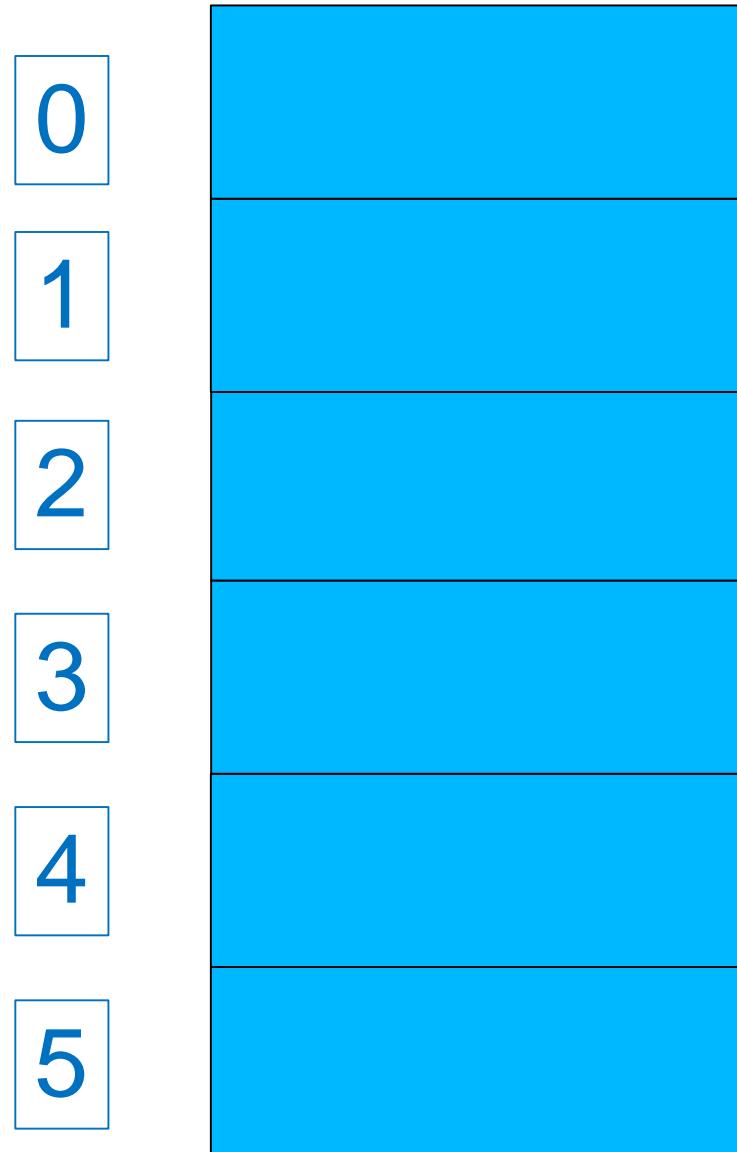
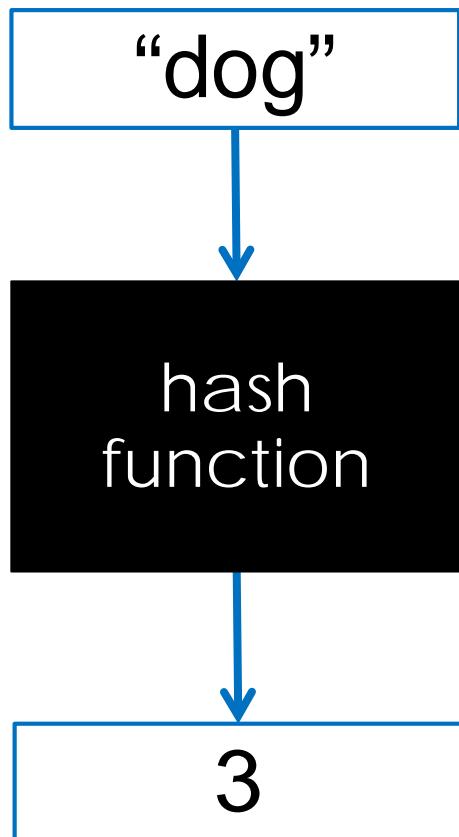
HashSet: Adding a value



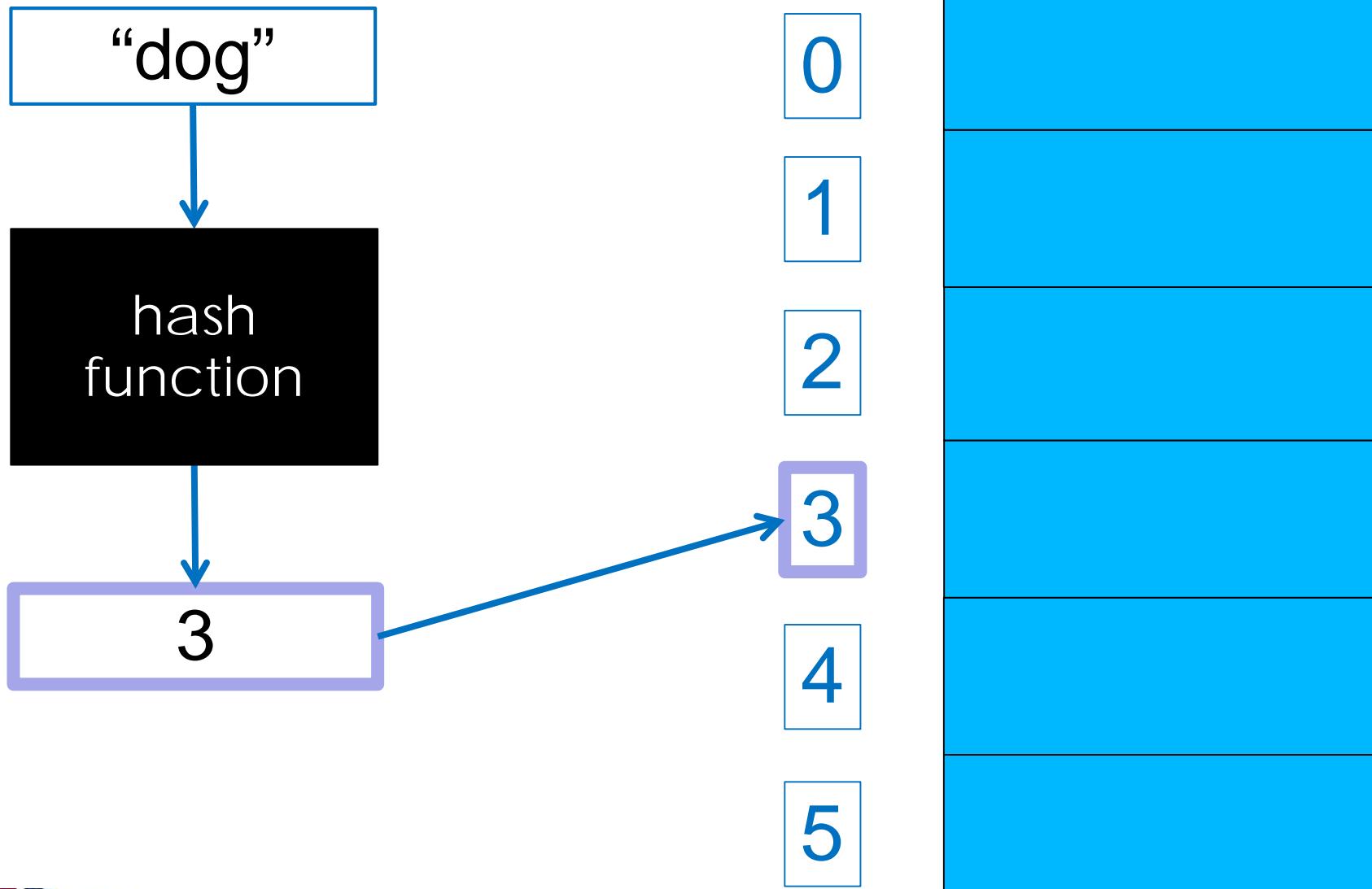
HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value

hash
function

0

1

2

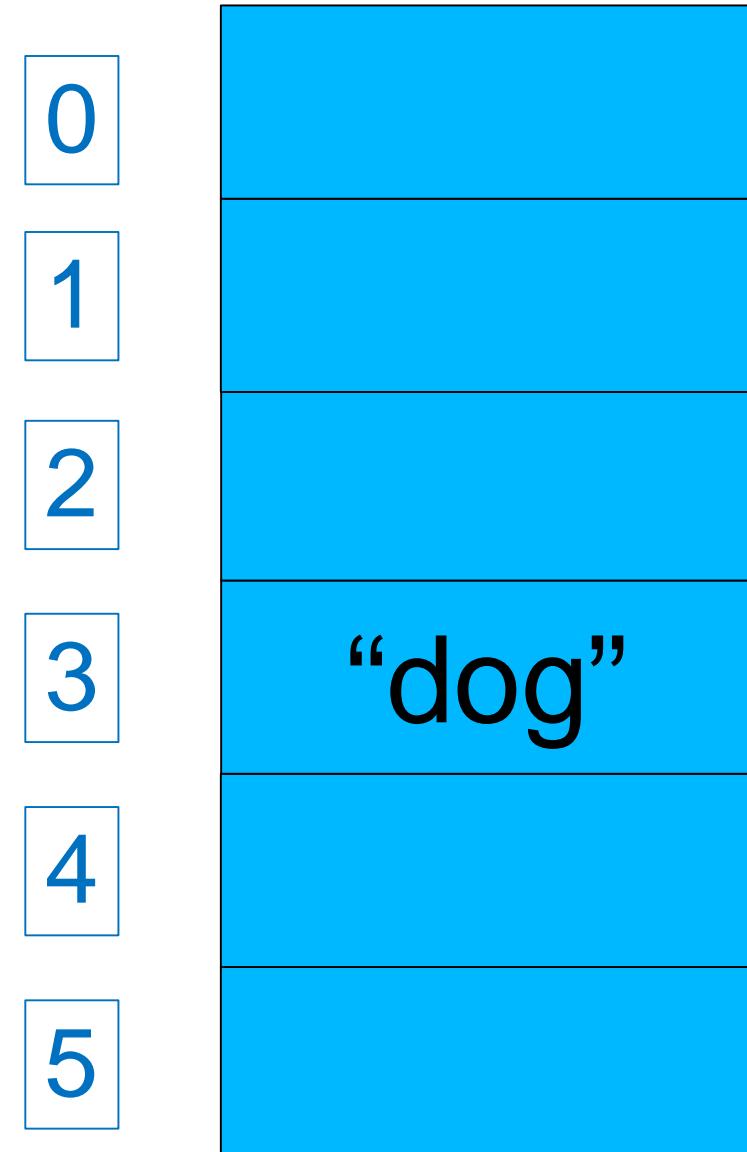
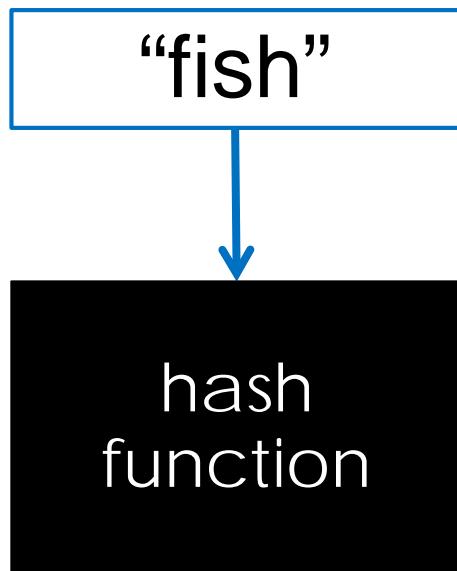
3

4

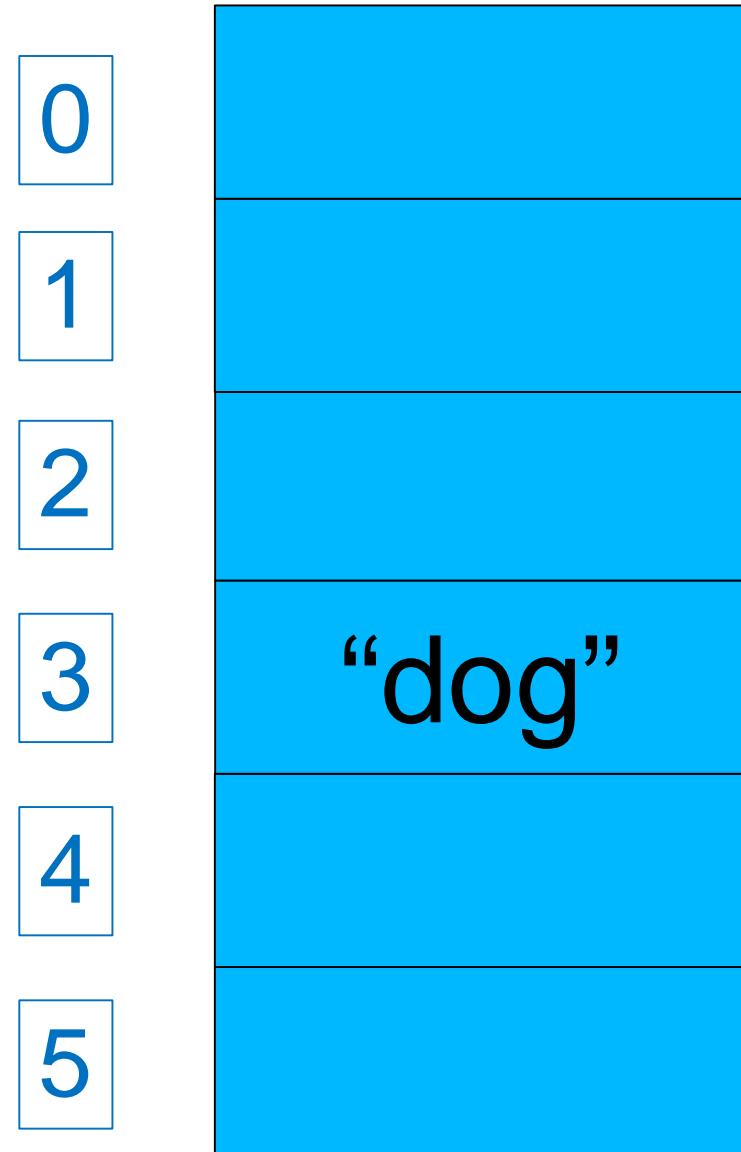
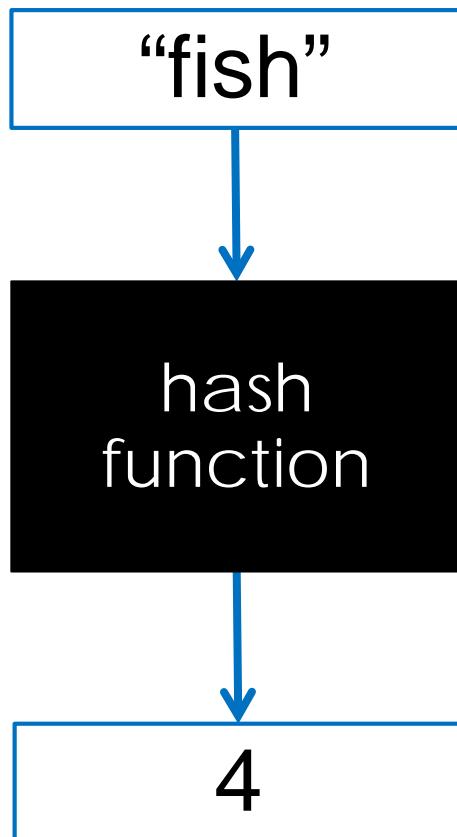
5



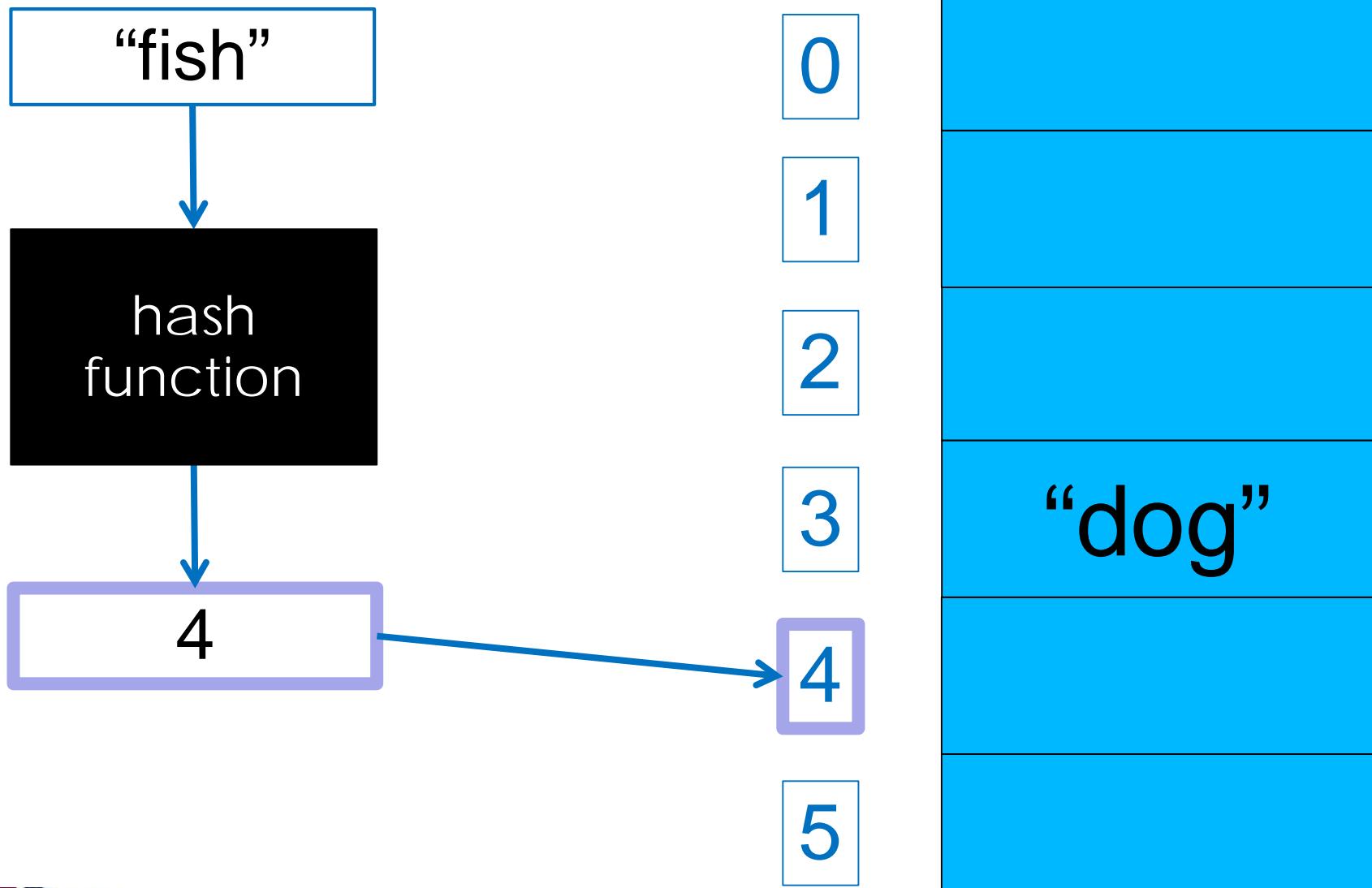
HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value

hash
function

0

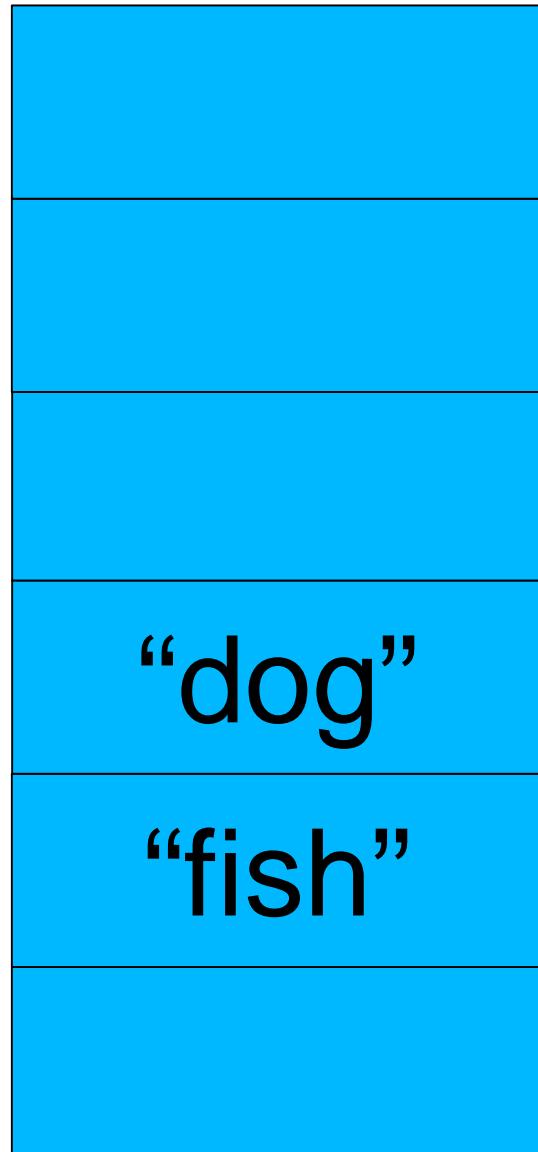
1

2

3

4

5



HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
public boolean add(String value) {  
    int index = hashCode(value);  
    if (values[index] == null) {  
        values[index] = value;  
        return true;  
    }  
    return false;  
}  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

**What if the hashcode
is greater than the size of
the hash set?**

HashSet: Adding a value

“elephant”

0

1

2

3

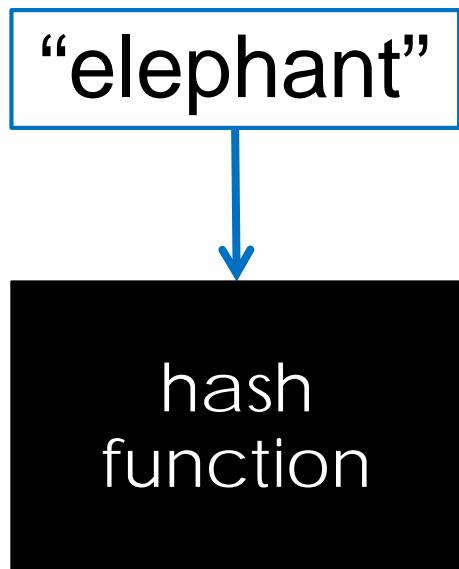
4

5

“dog”

“fish”

HashSet: Adding a value



0

1

2

3

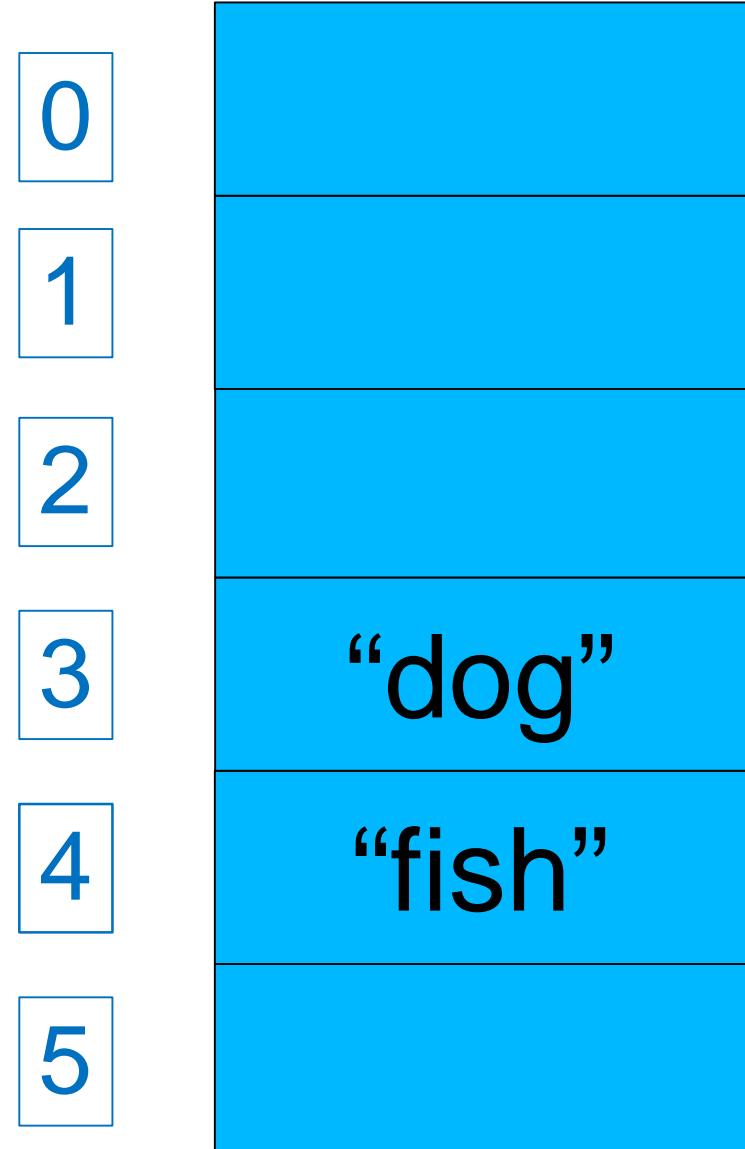
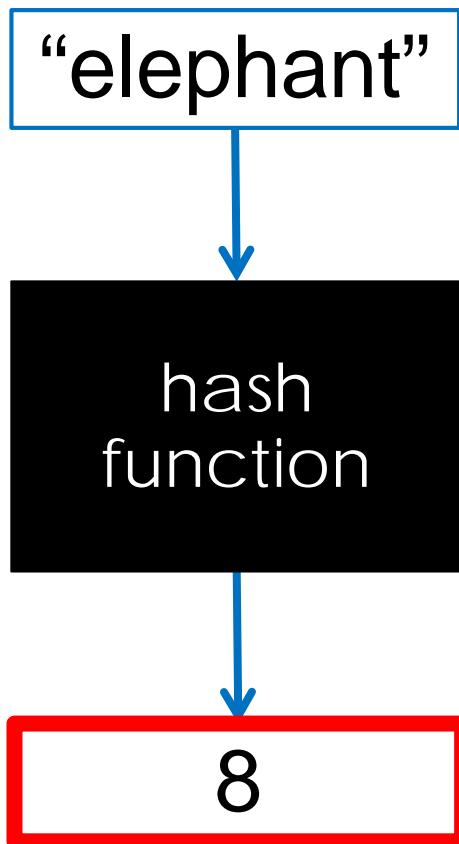
4

5

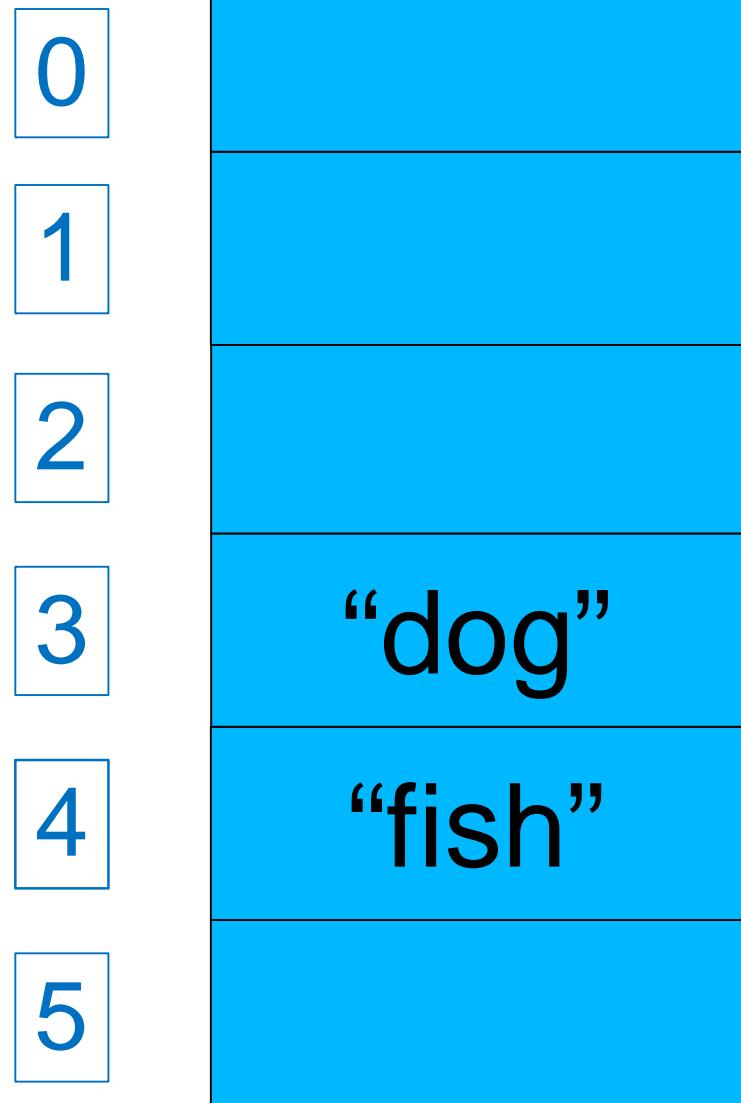
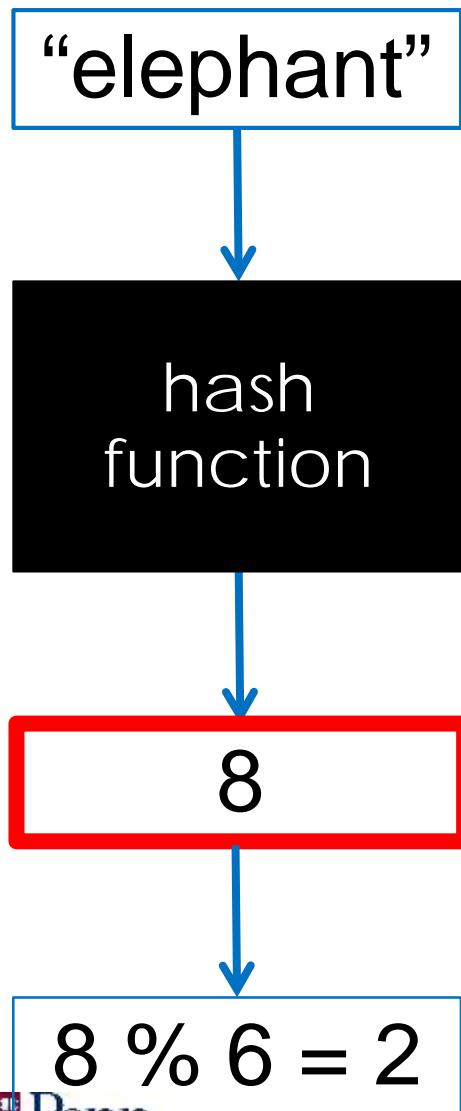
“dog”

“fish”

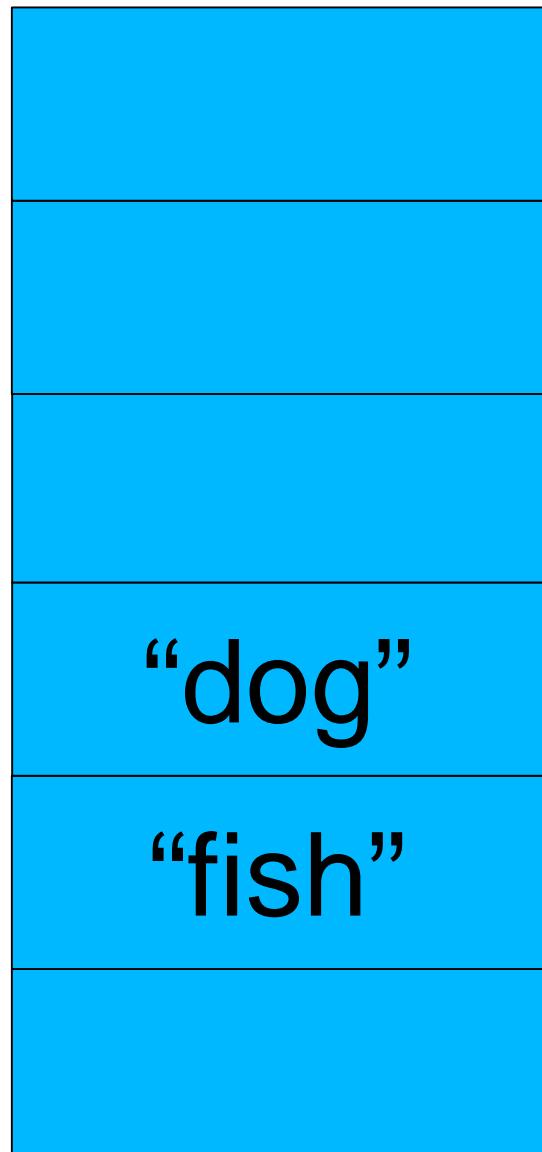
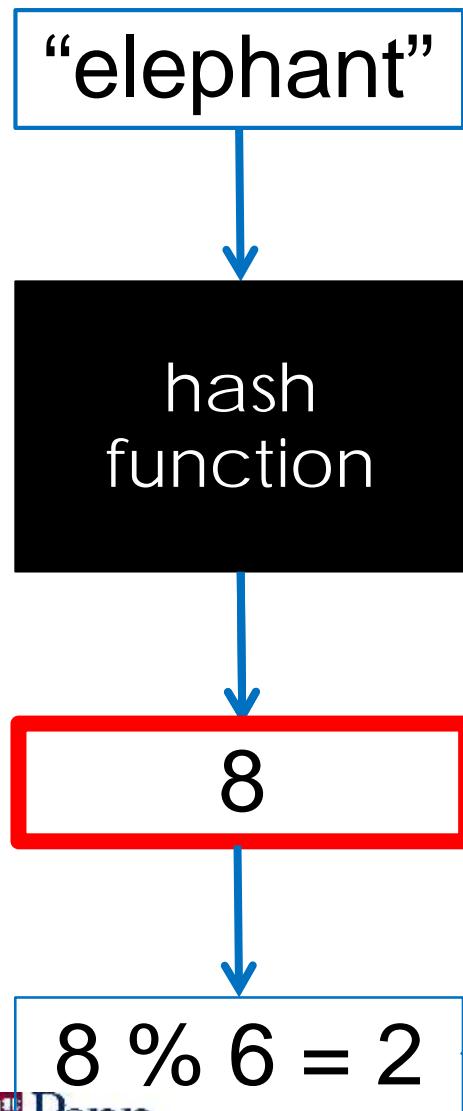
HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value



HashSet: Adding a value

hash
function

0

1

2

3

4

5

“elephant”

“dog”

“fish”

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value);  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet: Adding a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        int index = hashCode(value) % values.length;  
        if (values[index] == null) {  
            values[index] = value;  
            return true;  
        }  
        return false;  
    }  
}
```

How do we see if the hash set contains a value?

HashSet contains the value

0

1

2

3

4

5

“elephant”

“dog”

“fish”

HashSet contains the value

“dog”

0

1

2

3

4

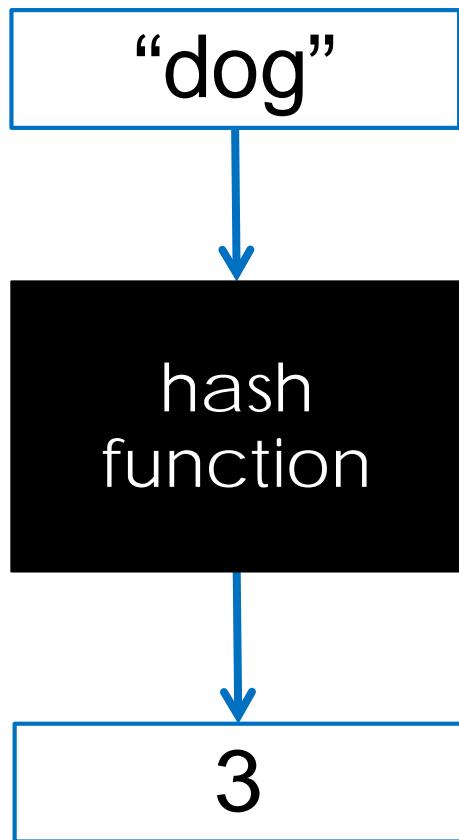
5

“elephant”

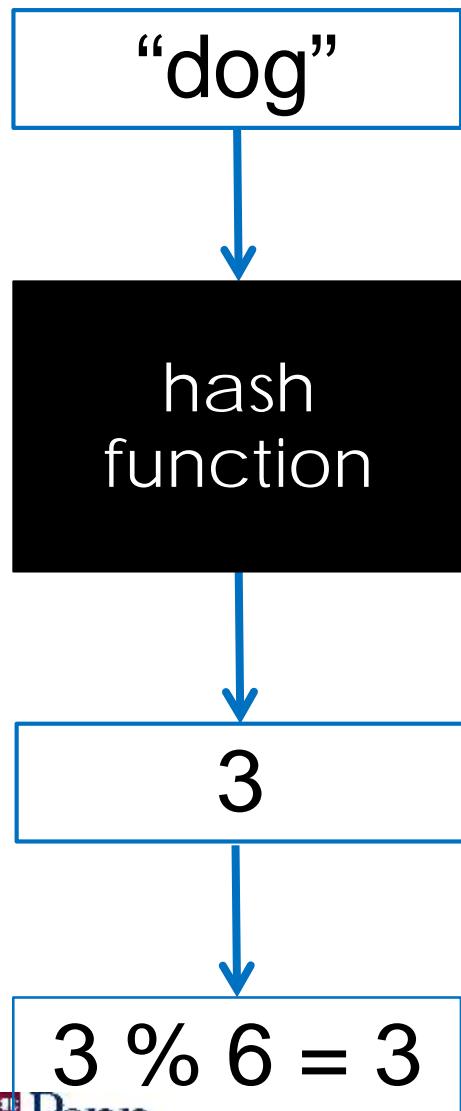
“dog”

“fish”

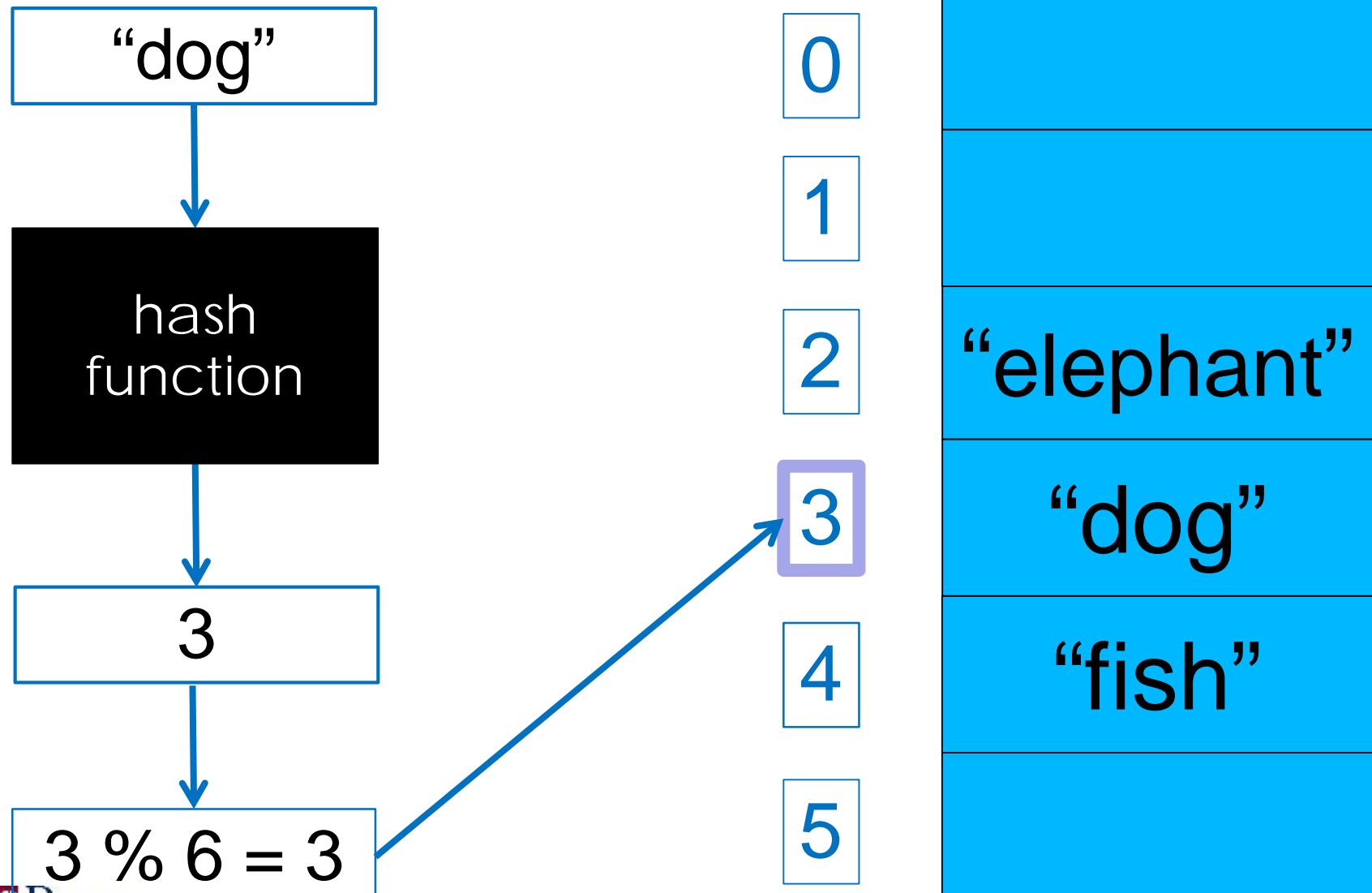
HashSet contains the value



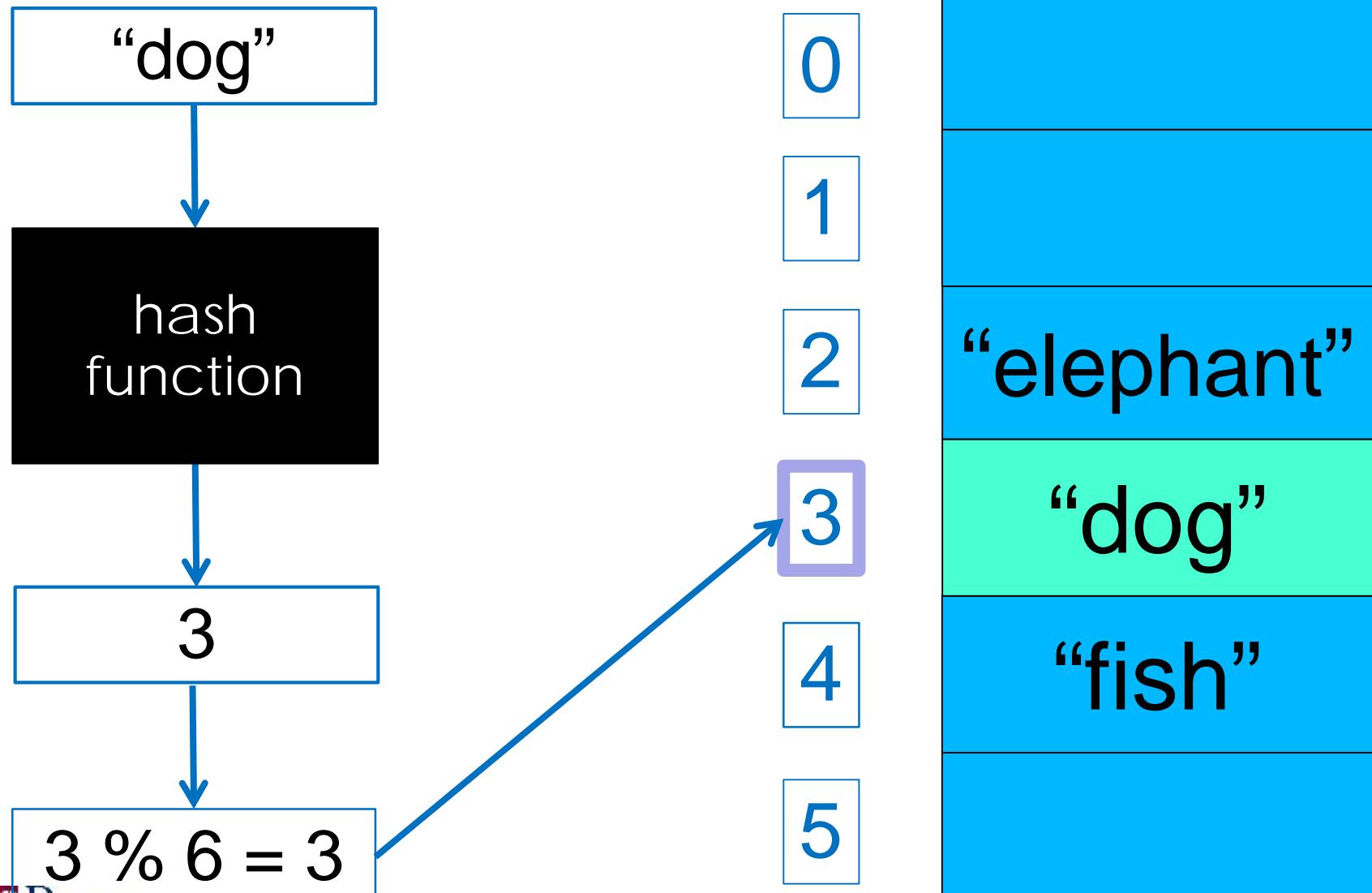
HashSet contains the value



HashSet contains the value



HashSet contains the value



HashSet does not contain the value

“panda”

hash
function

0

1

2

3

4

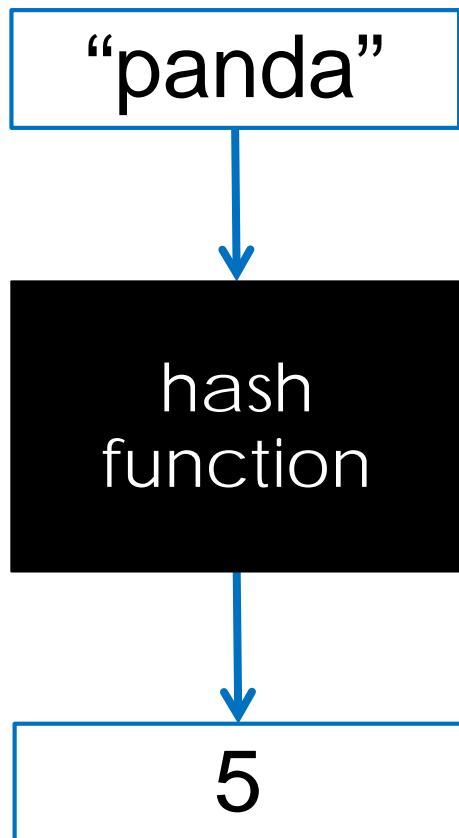
5

“elephant”

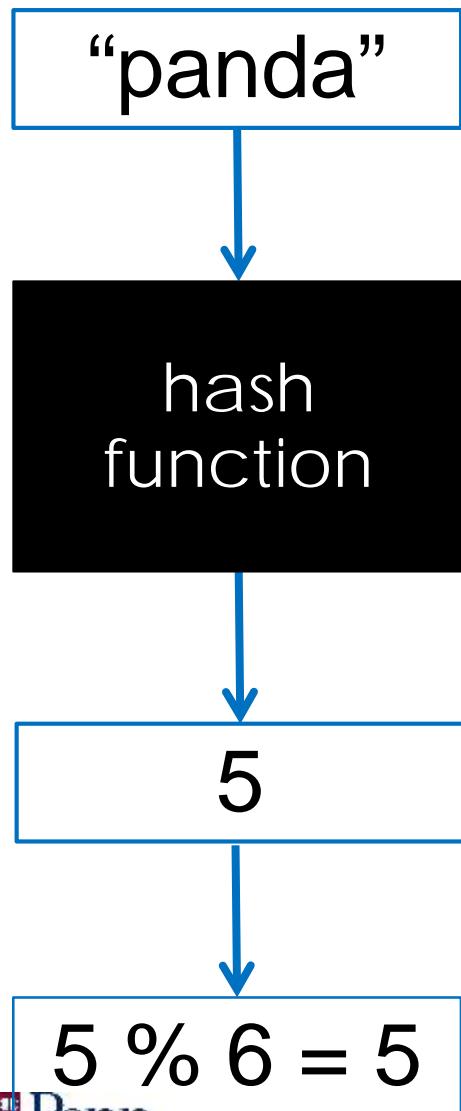
“dog”

“fish”

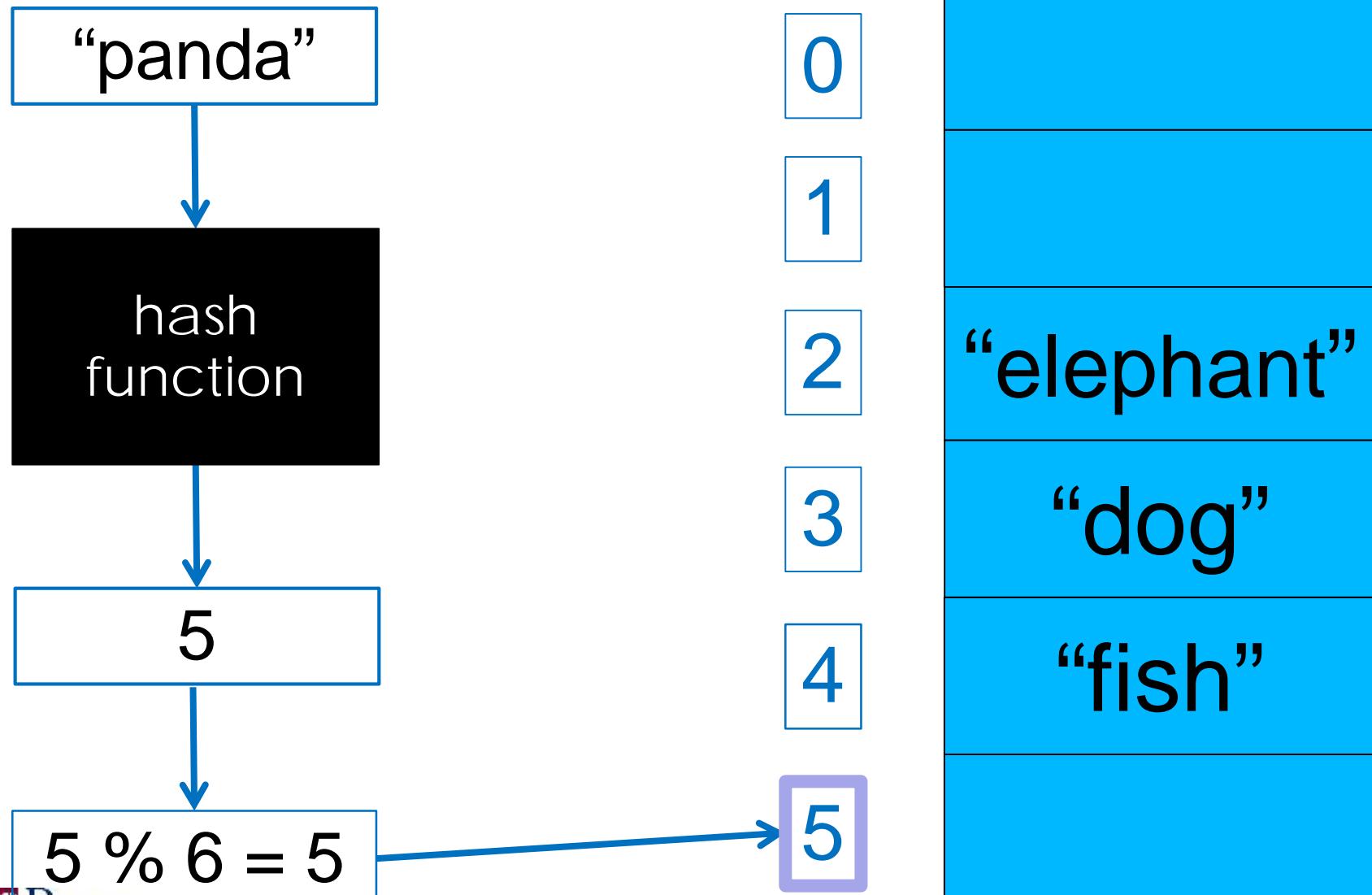
HashSet does not contain the value



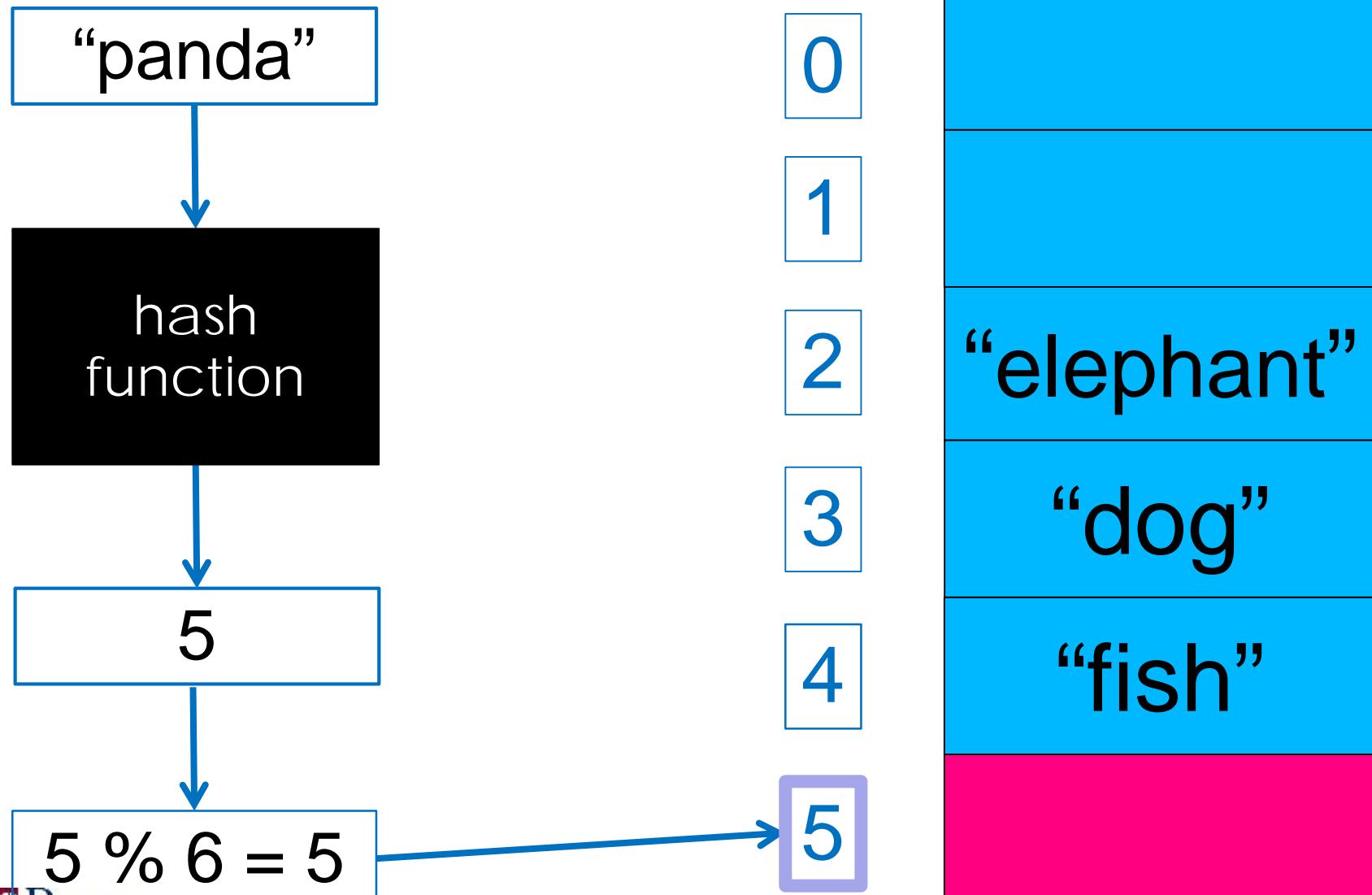
HashSet does not contain the value



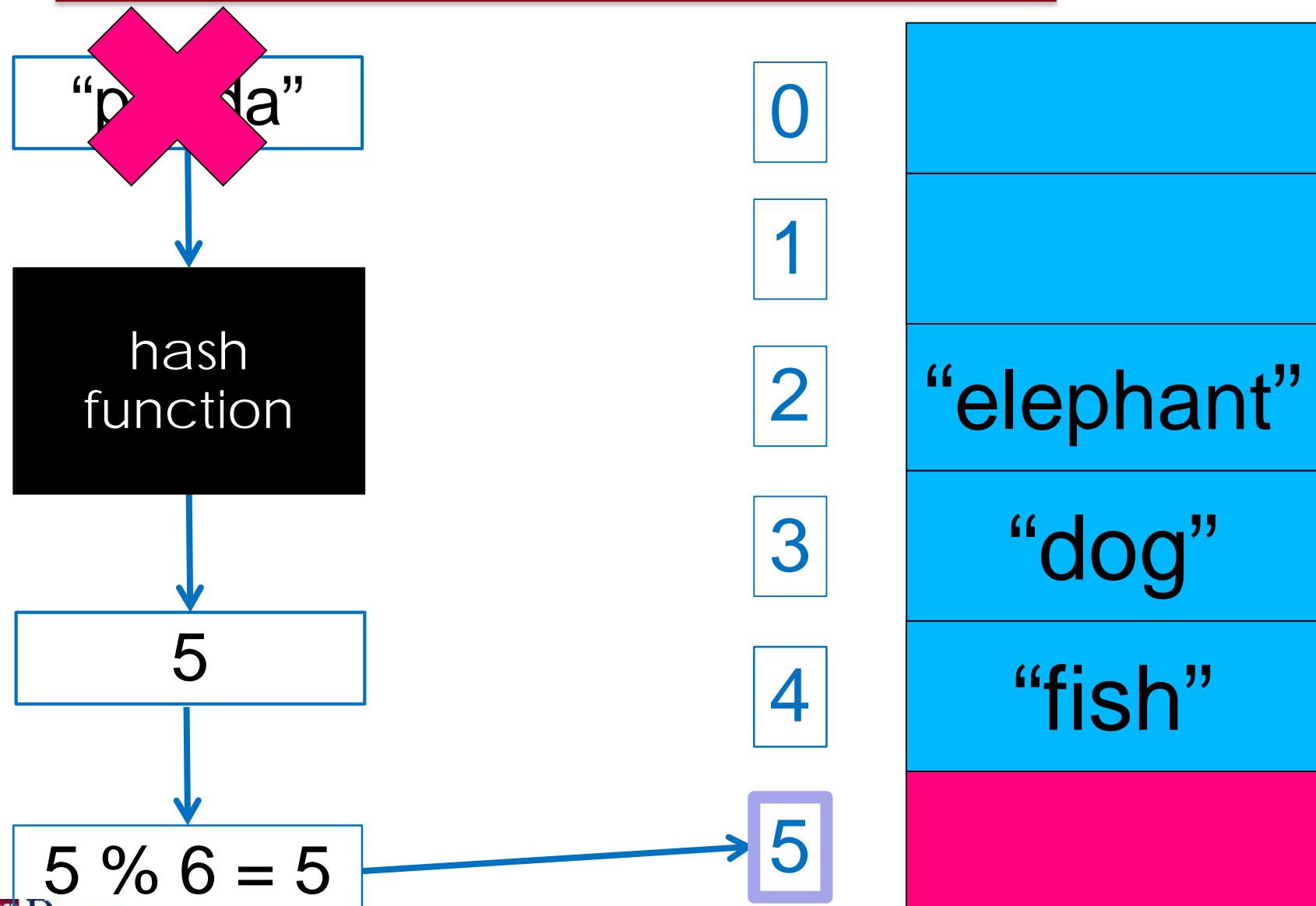
HashSet does not contain the value



HashSet does not contain the value



HashSet does not contain the value



HashSet: Contains a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % values.length;  
        return value.equals(values[index]);  
    }  
}
```

HashSet: Contains a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
public boolean contains(String value) {  
    int index = hashCode(value) % values.length;  
    return value.equals(values[index]);  
}  
}
```

HashSet: Contains a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % values.length;  
        return value.equals(values[index]);  
    }  
}
```

HashSet: Contains a value

```
public class HashSet {  
    . . .  
    private String[] values;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % values.length;  
        return value.equals(values[index]);  
    }  
}
```

Recap: HashSets

- Elements are stored in an array
- Each element has an associated **hash code** that is used to determine its index in the array
- Adding and finding an element are $O(1)$!
- But this is a very naïve implementation...

SD2x1.9

Hash Sets collision handling

Kathy

What happens if two elements have the same hash code?

HashSet: Adding a value

0

1

2

3

4

5

“elephant”

“dog”

“fish”

HashSet: Adding a value

“cat”

0

1

2

3

4

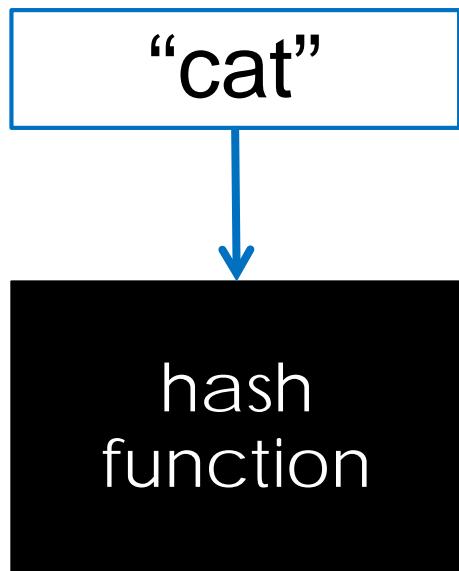
5

“elephant”

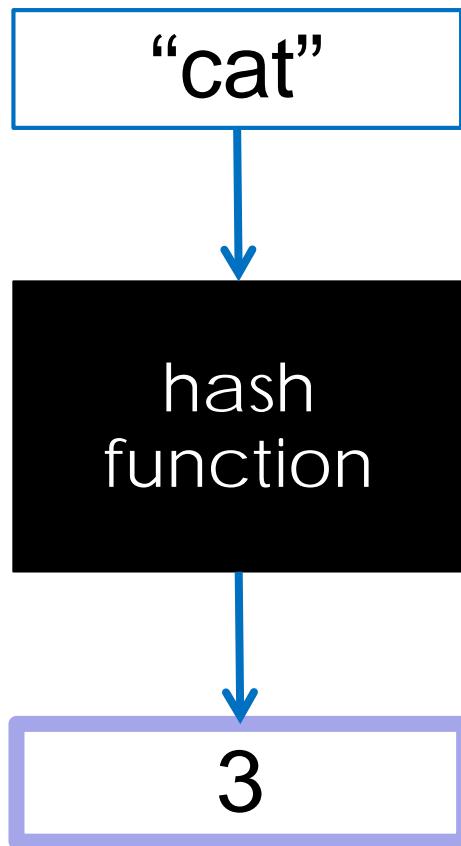
“dog”

“fish”

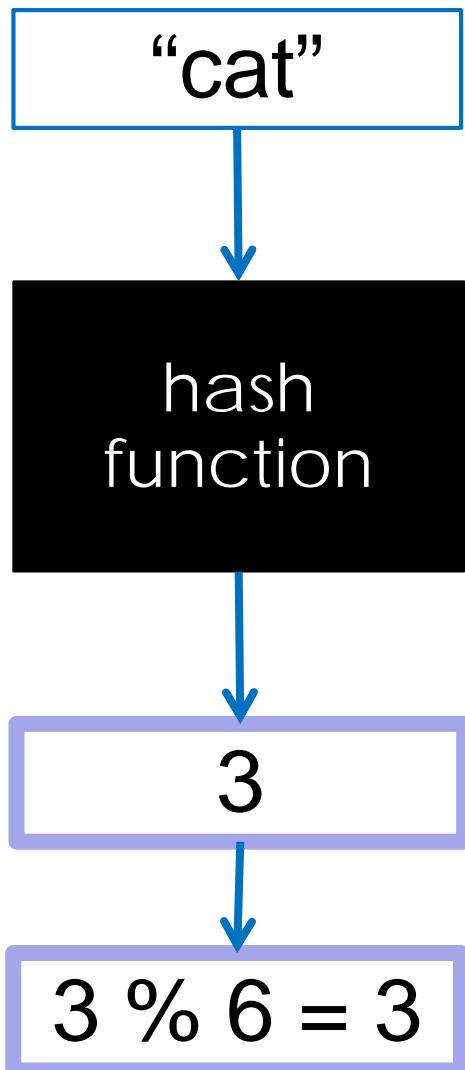
HashSet: Adding a value



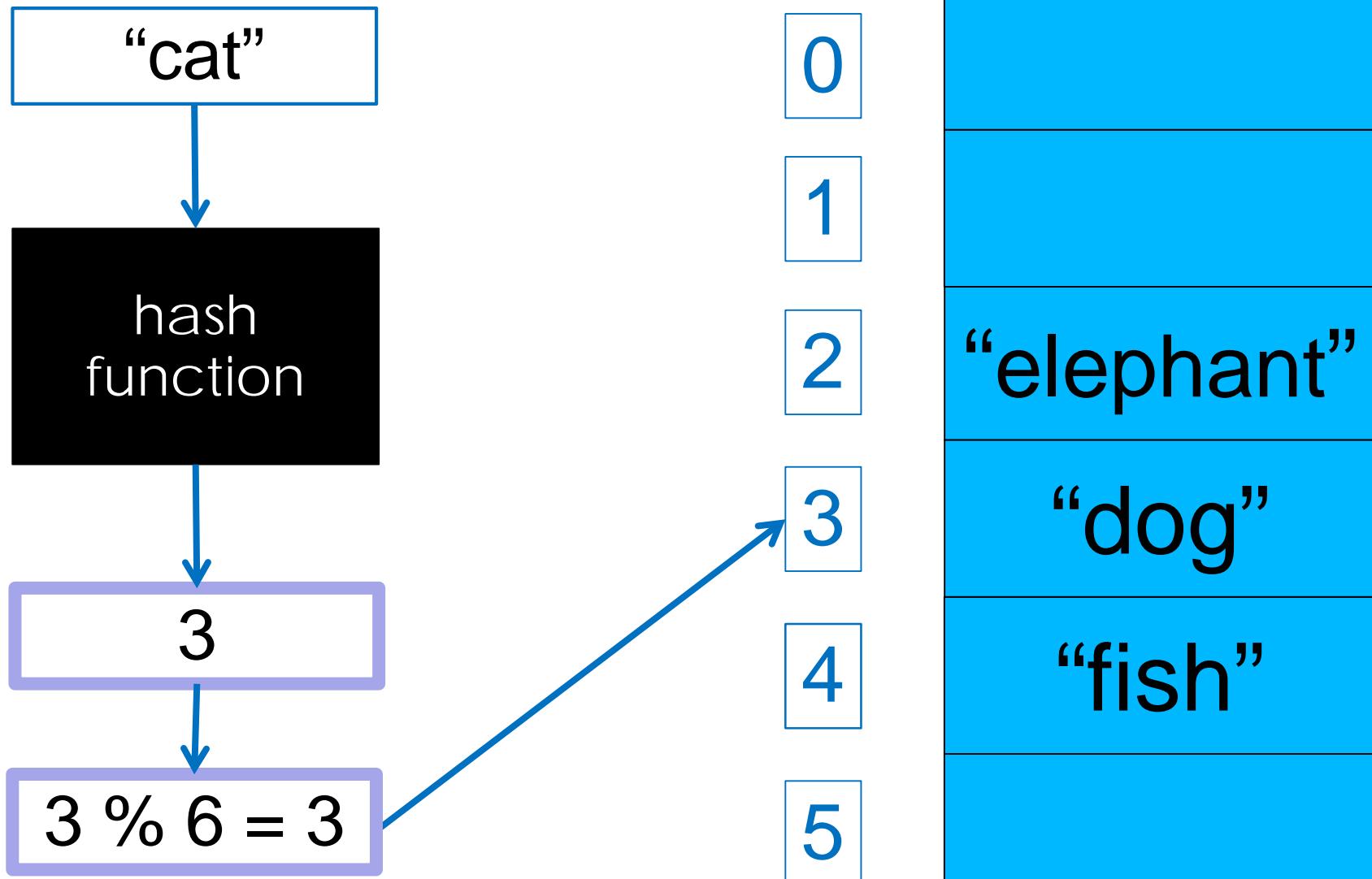
HashSet: Adding a value



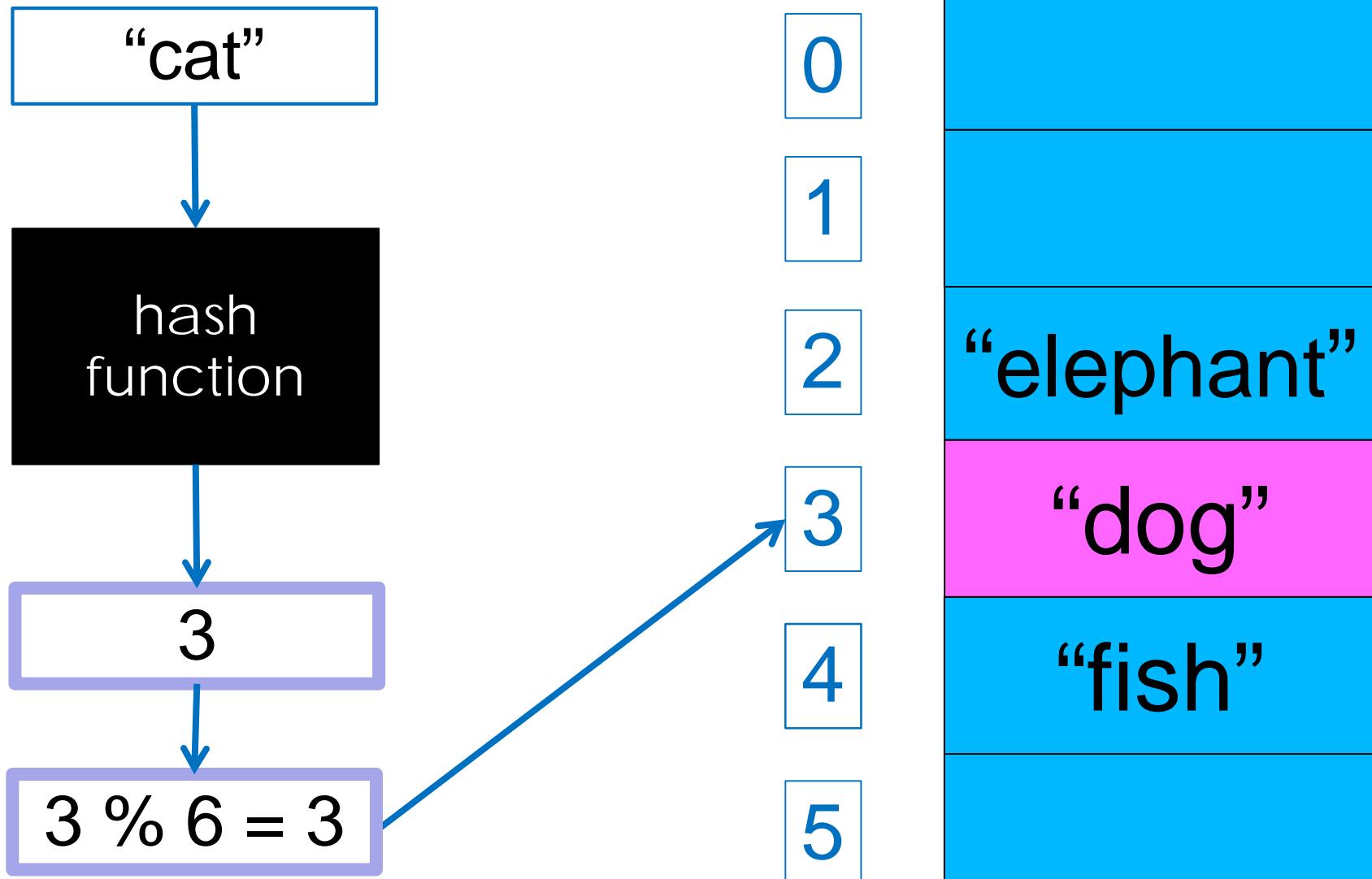
HashSet: Adding a value



HashSet: Adding a value

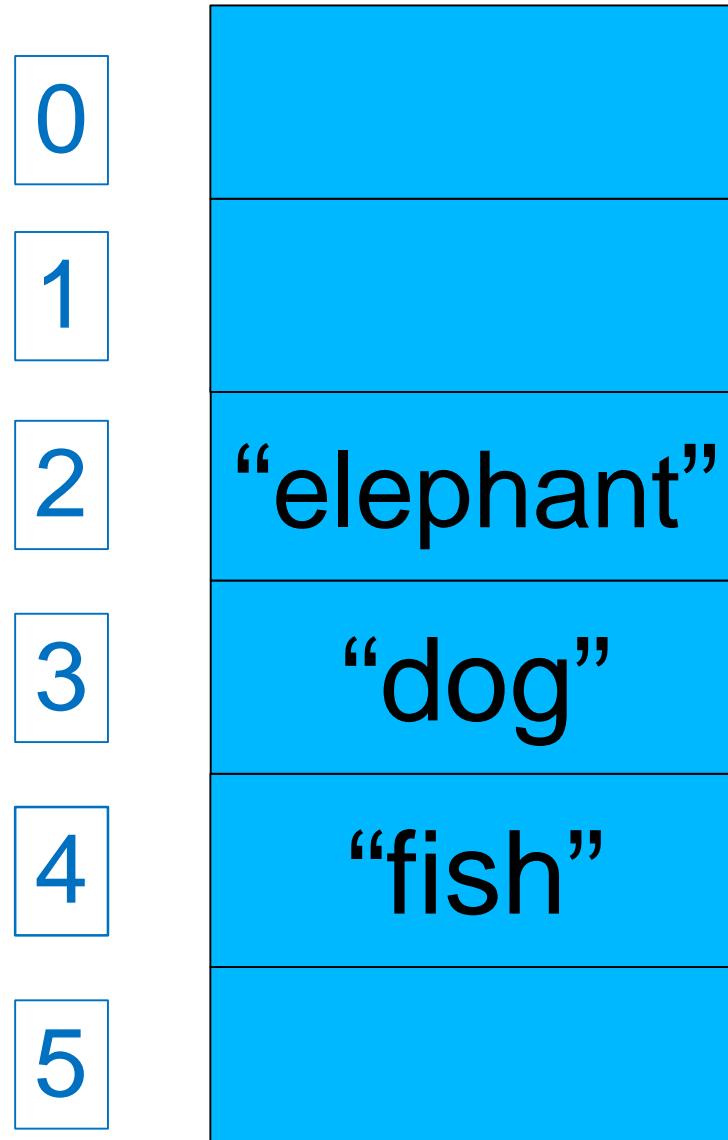


HashSet: Adding a value

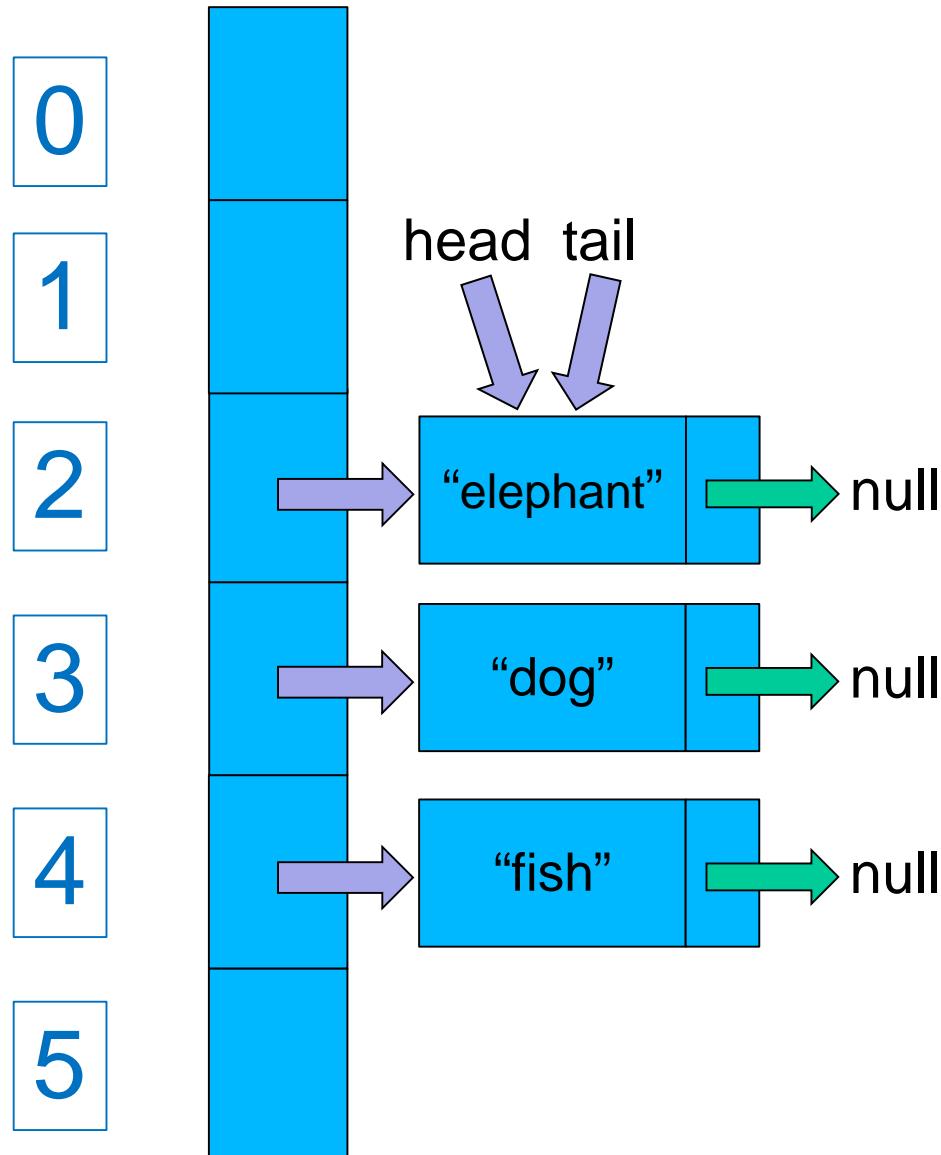


How can we resolve collisions?

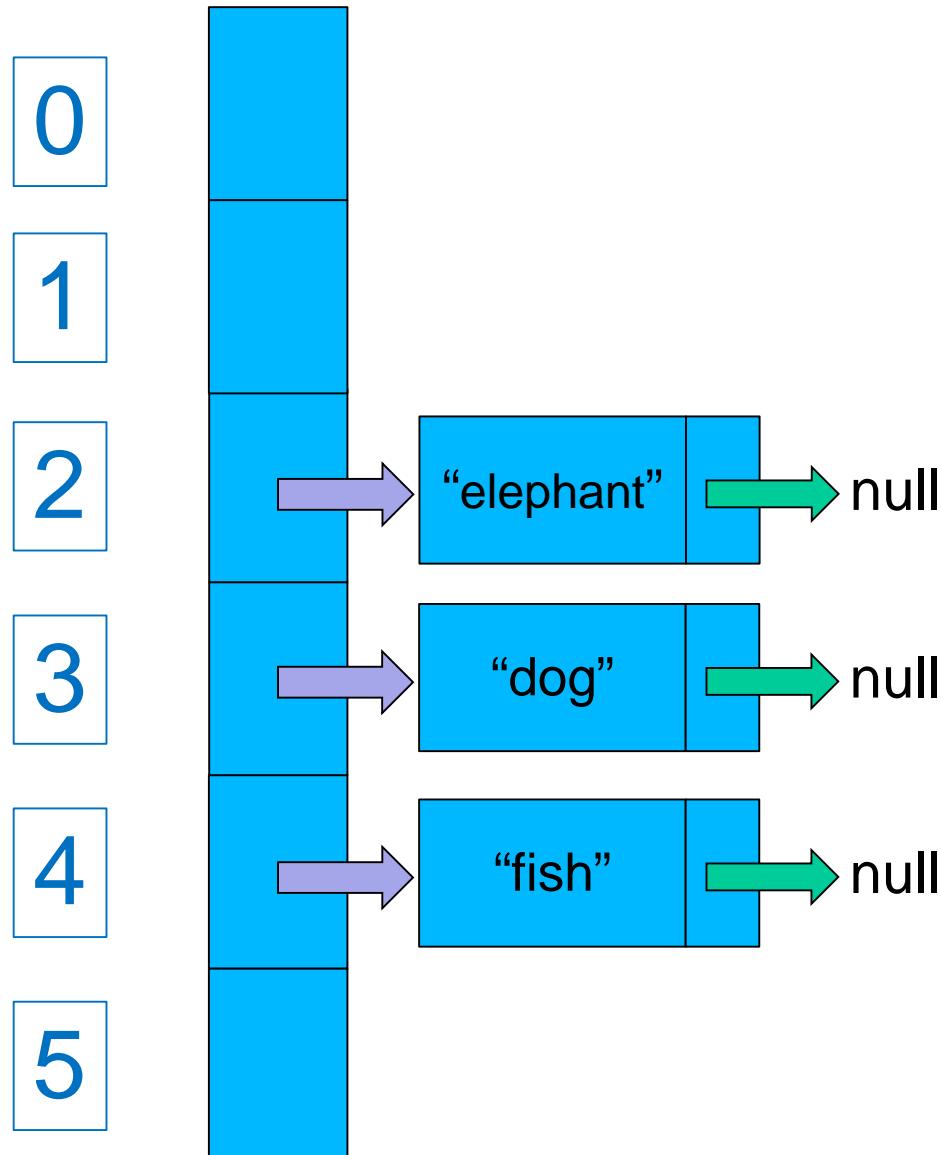
HashSet: Separate chaining



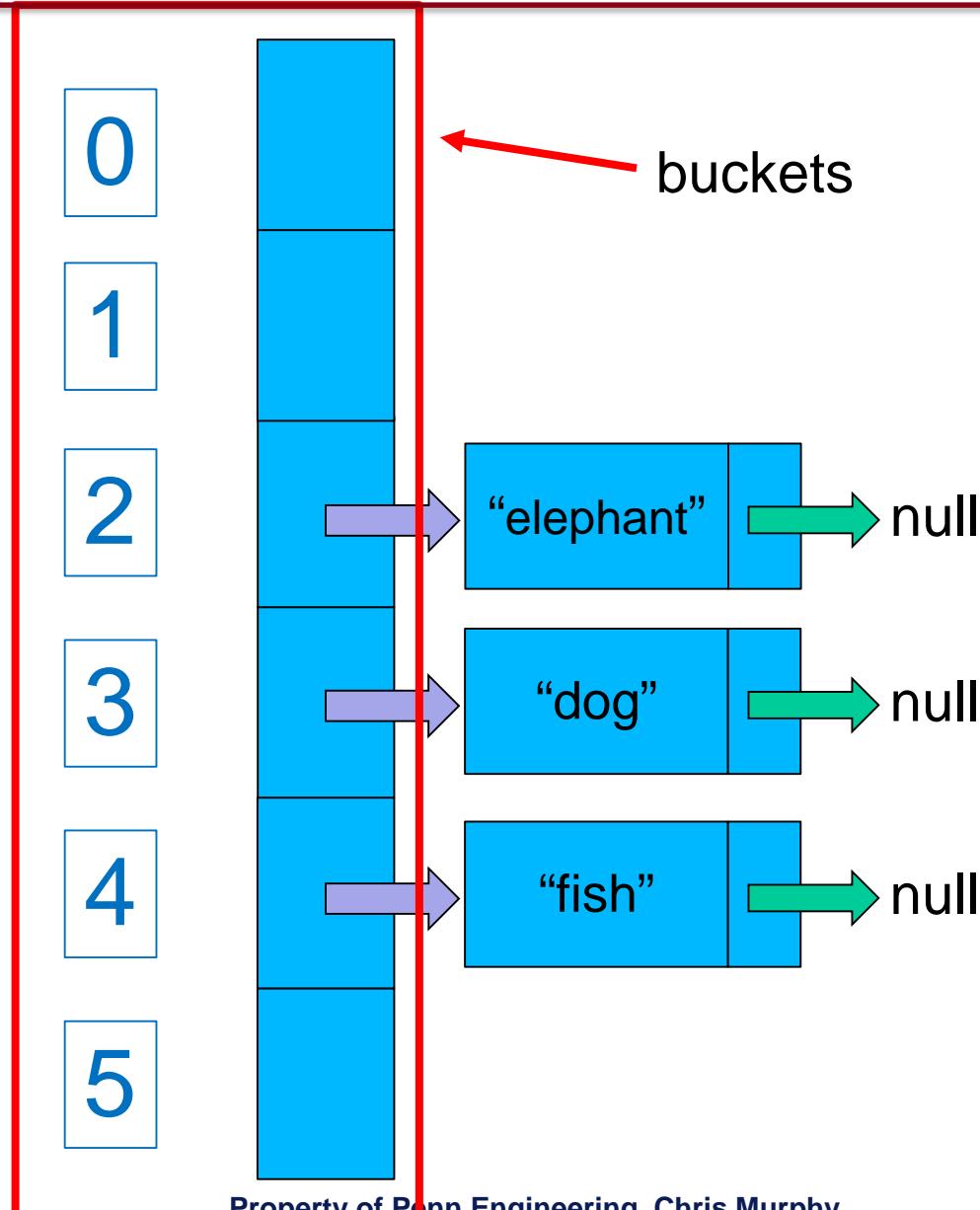
HashSet: Separate chaining



HashSet: Separate chaining



HashSet: Separate chaining



HashSet using separate chaining

```
public class HashSet {  
  
    private LinkedList<String>[ ] buckets;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    . . .  
}
```

HashSet using separate chaining

```
public class HashSet {  
  
    private LinkedList<String>[ ] buckets;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    . . .  
}
```

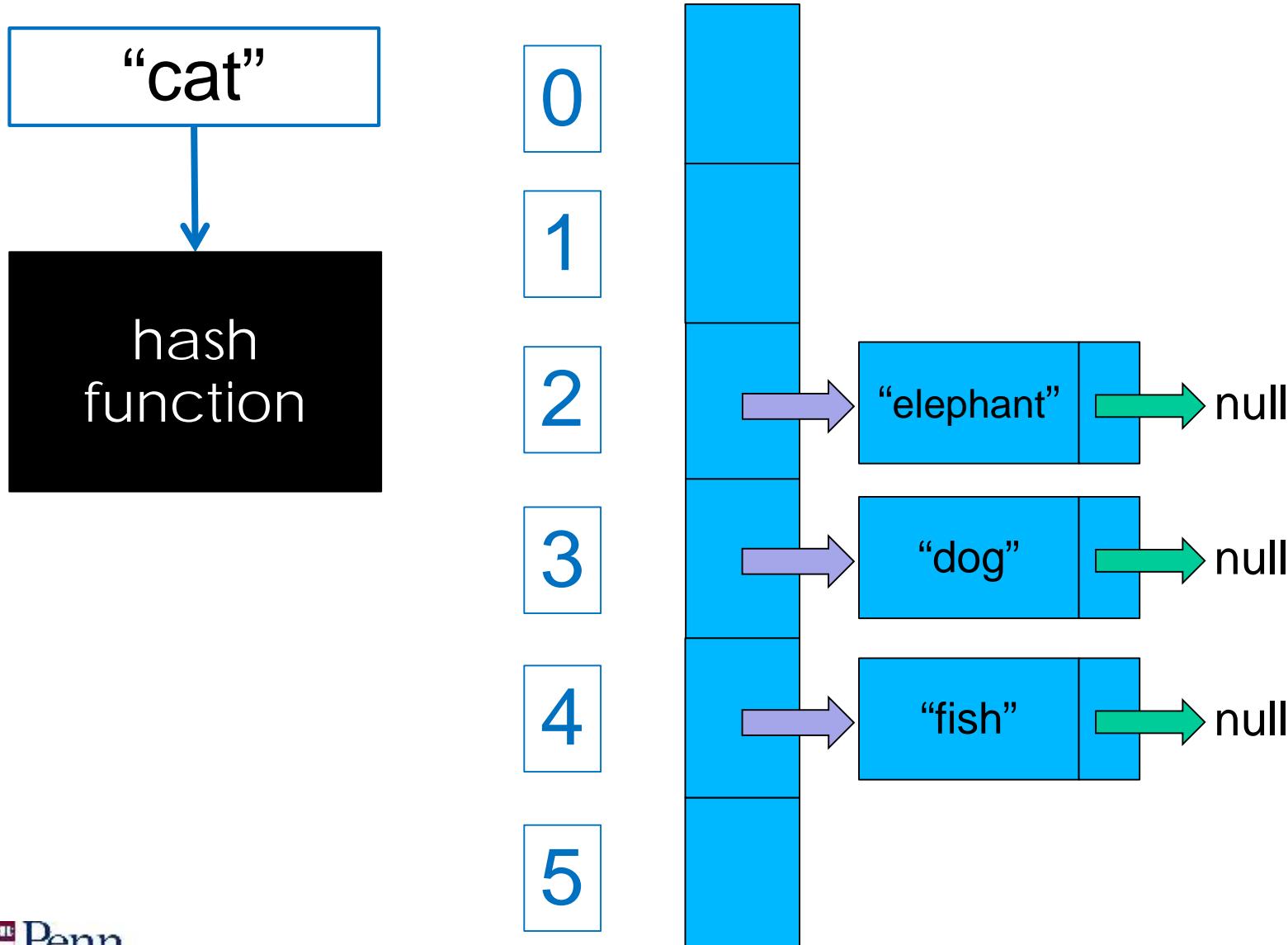
HashSet using separate chaining

```
public class HashSet {  
  
    private LinkedList<String>[ ] buckets;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    . . .  
}
```

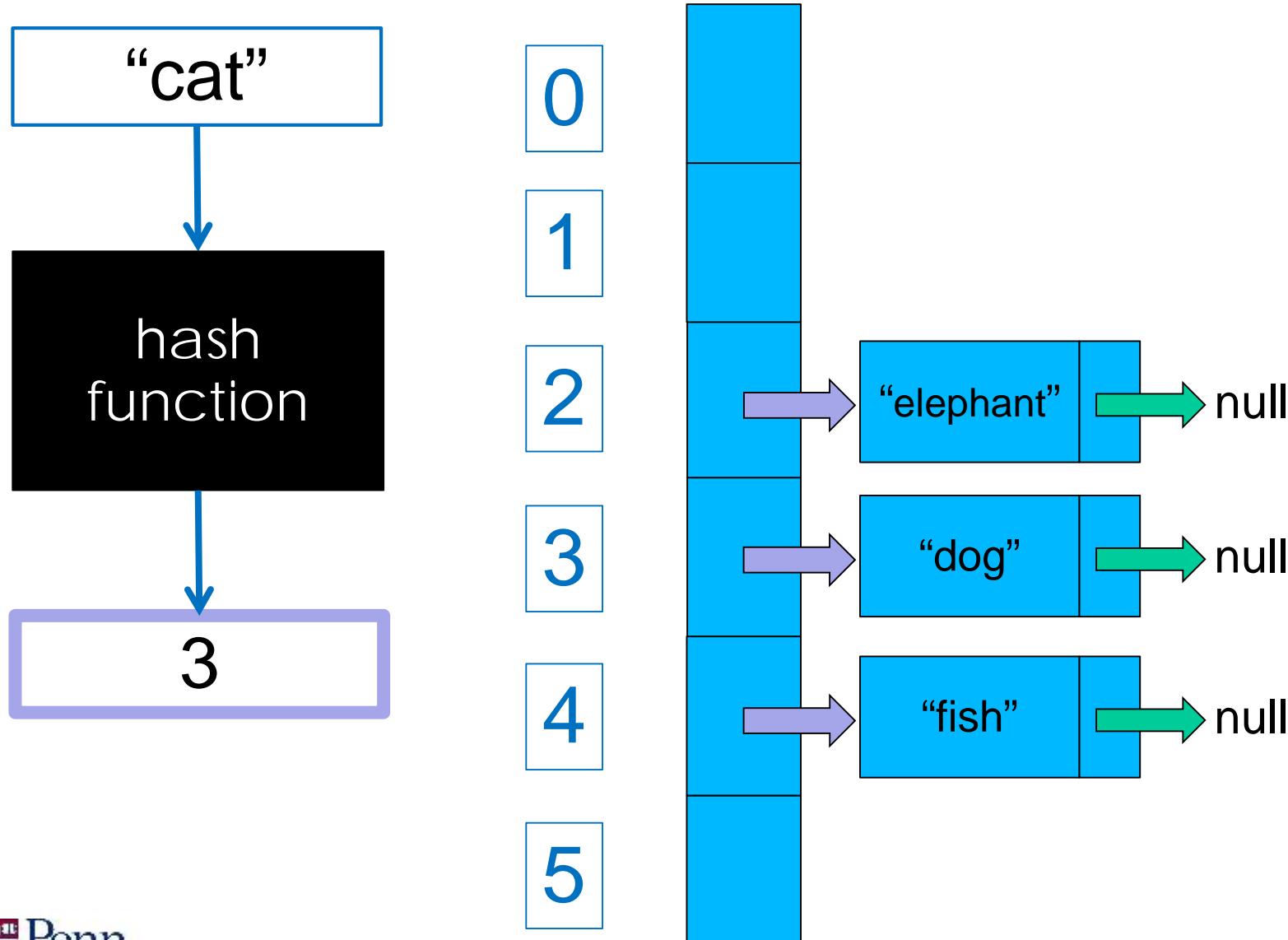
HashSet using separate chaining

```
public class HashSet {  
  
    private LinkedList<String>[ ] buckets;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    . . .  
}
```

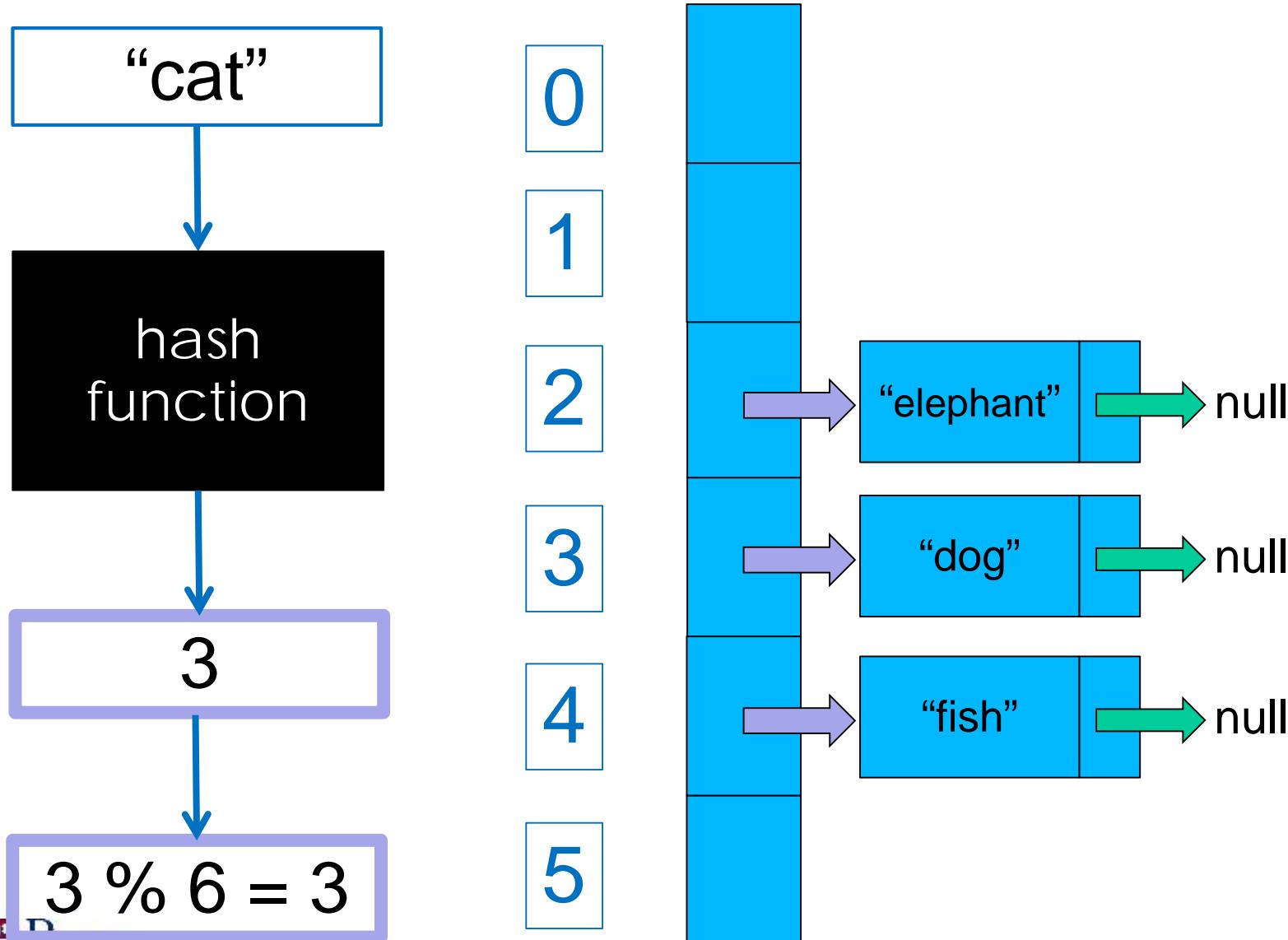
Separate Chaining: Adding a value



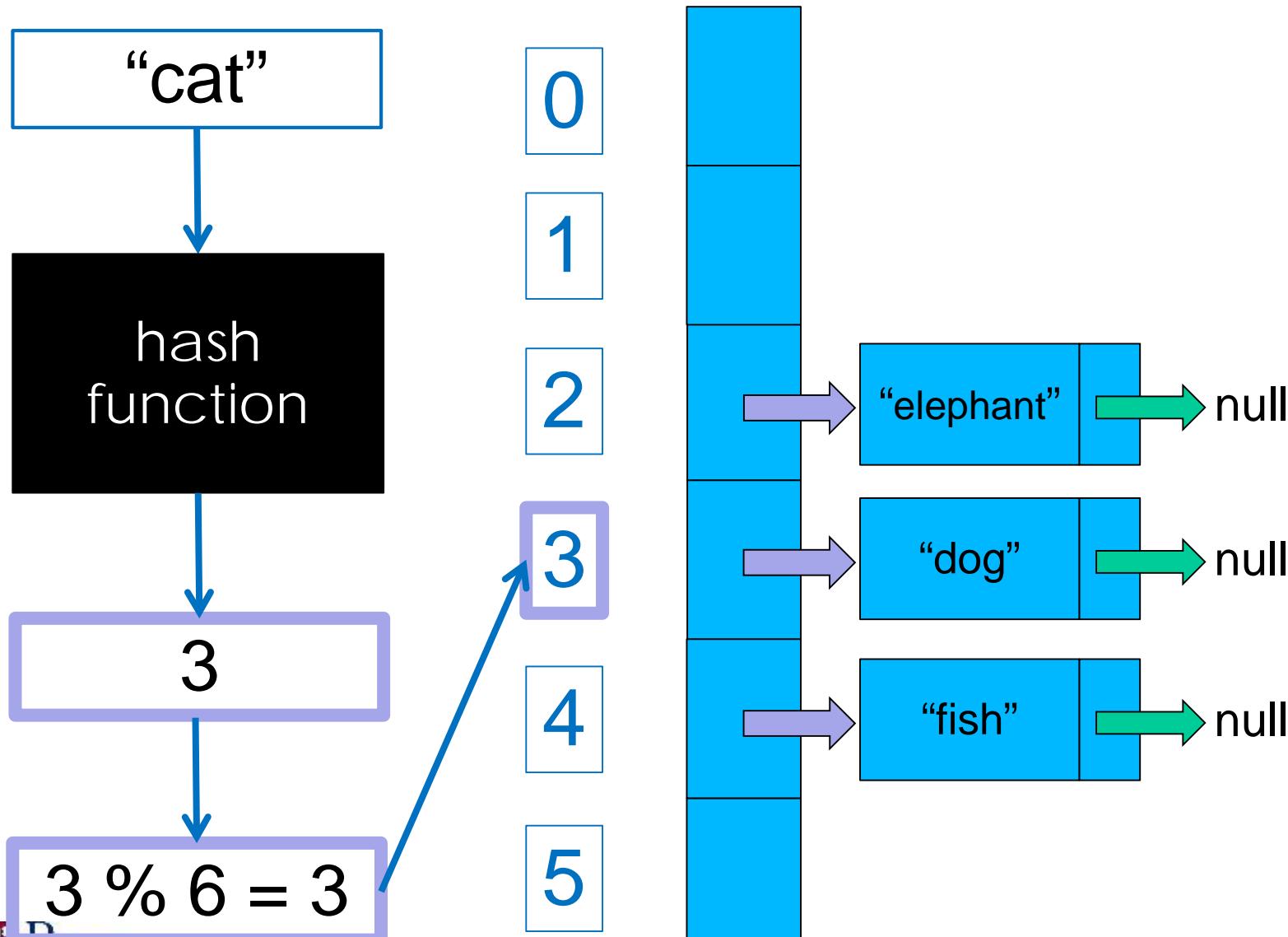
Separate Chaining: Adding a value



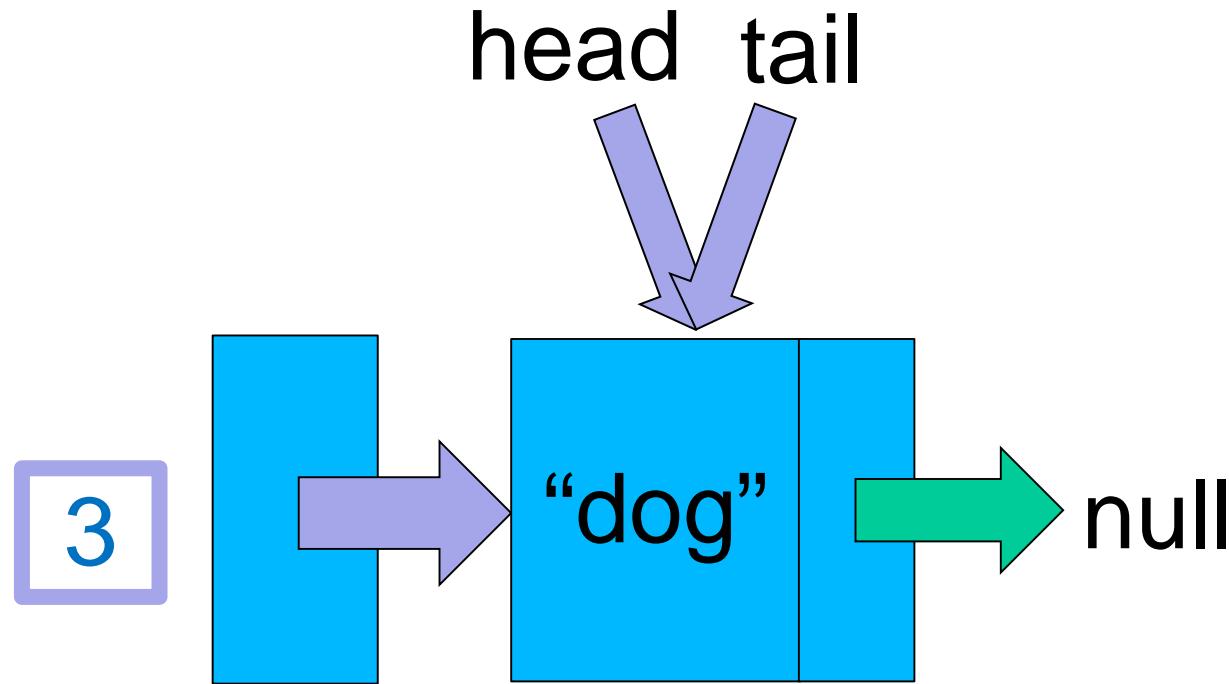
Separate Chaining: Adding a value



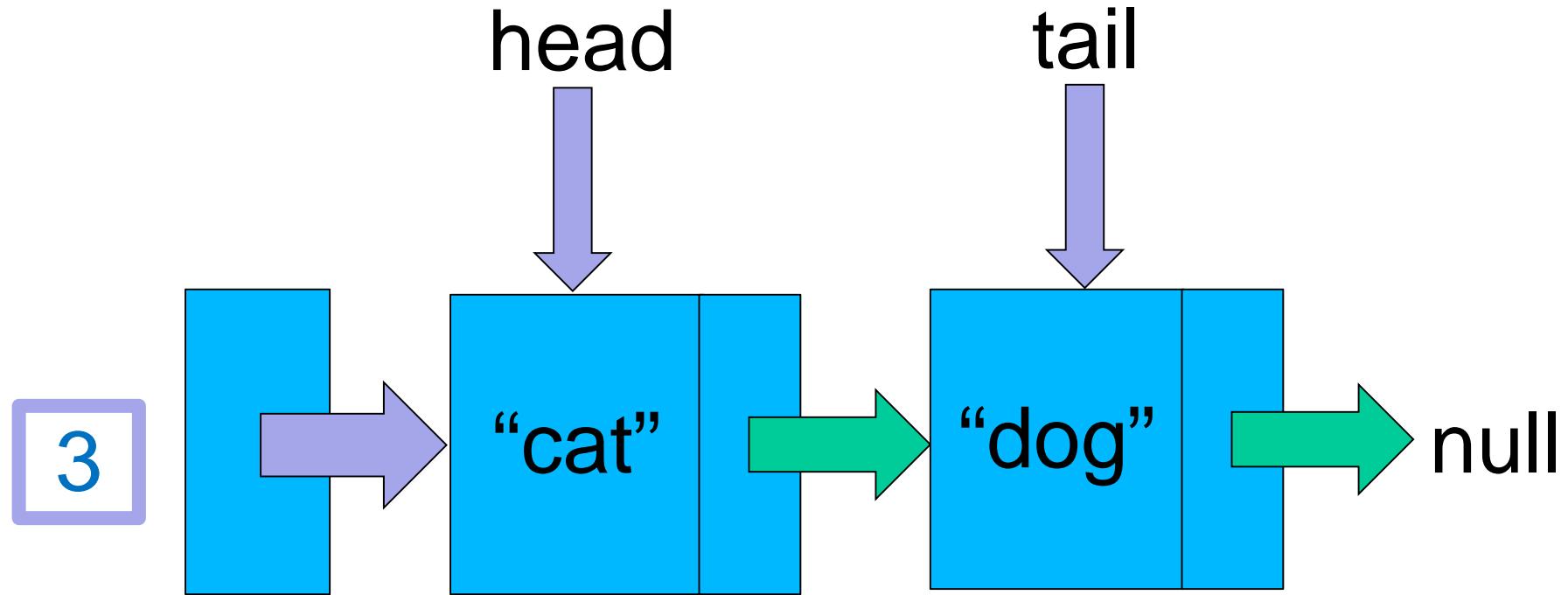
Separate Chaining: Adding a value



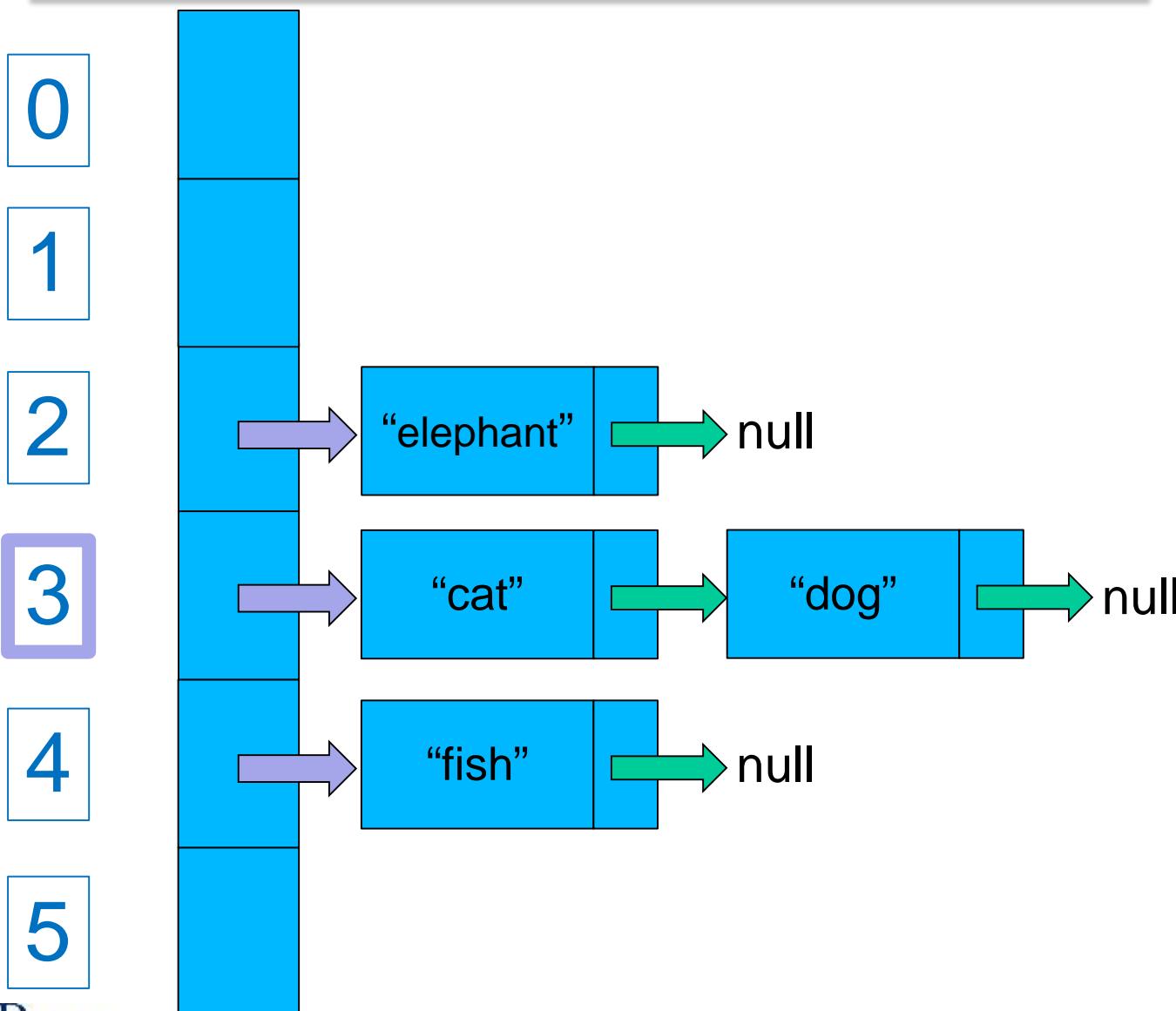
Separate Chaining: Adding a value



Separate Chaining: Adding a value



Separate Chaining: Adding a value



HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
public boolean add(String value) {  
    if (!contains(value)) {  
        int index = hashCode(value) % buckets.length;  
        LinkedList<String> bucket = buckets[index];  
        bucket.addFirst(value);  
        return true;  
    }  
    return false;  
}  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
private LinkedList<String>[] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

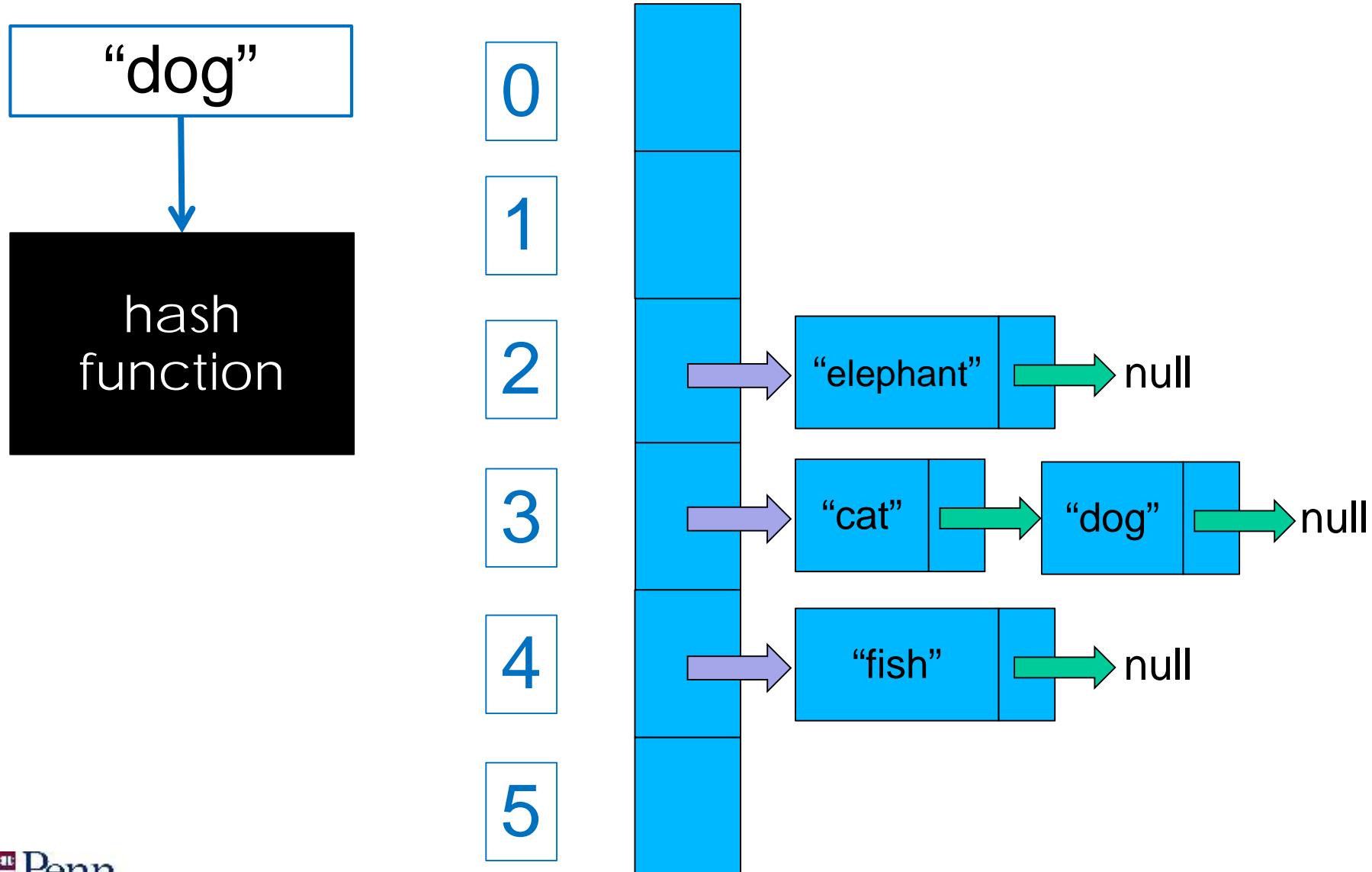
HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

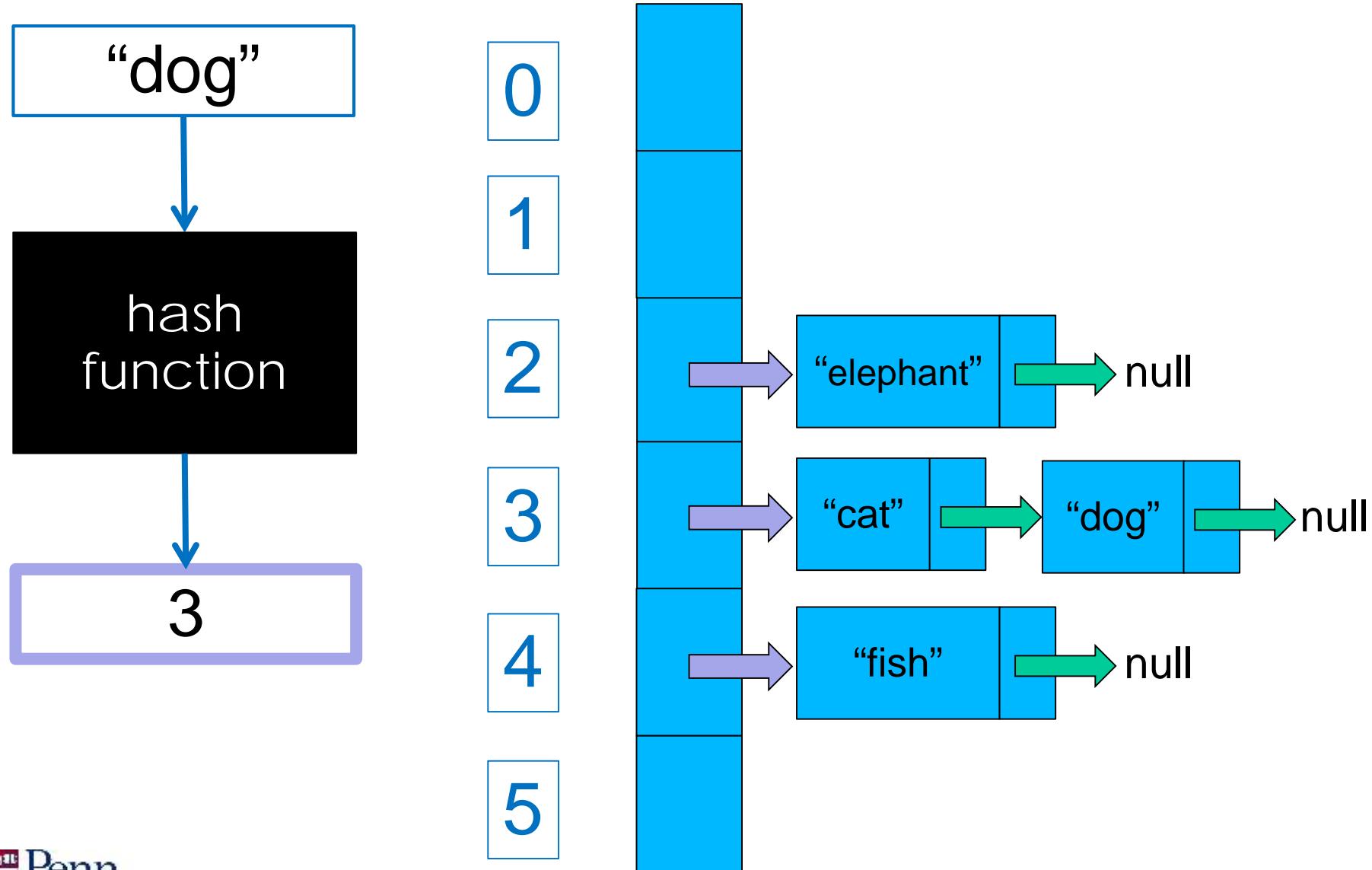
HashSet with separate chaining: add

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

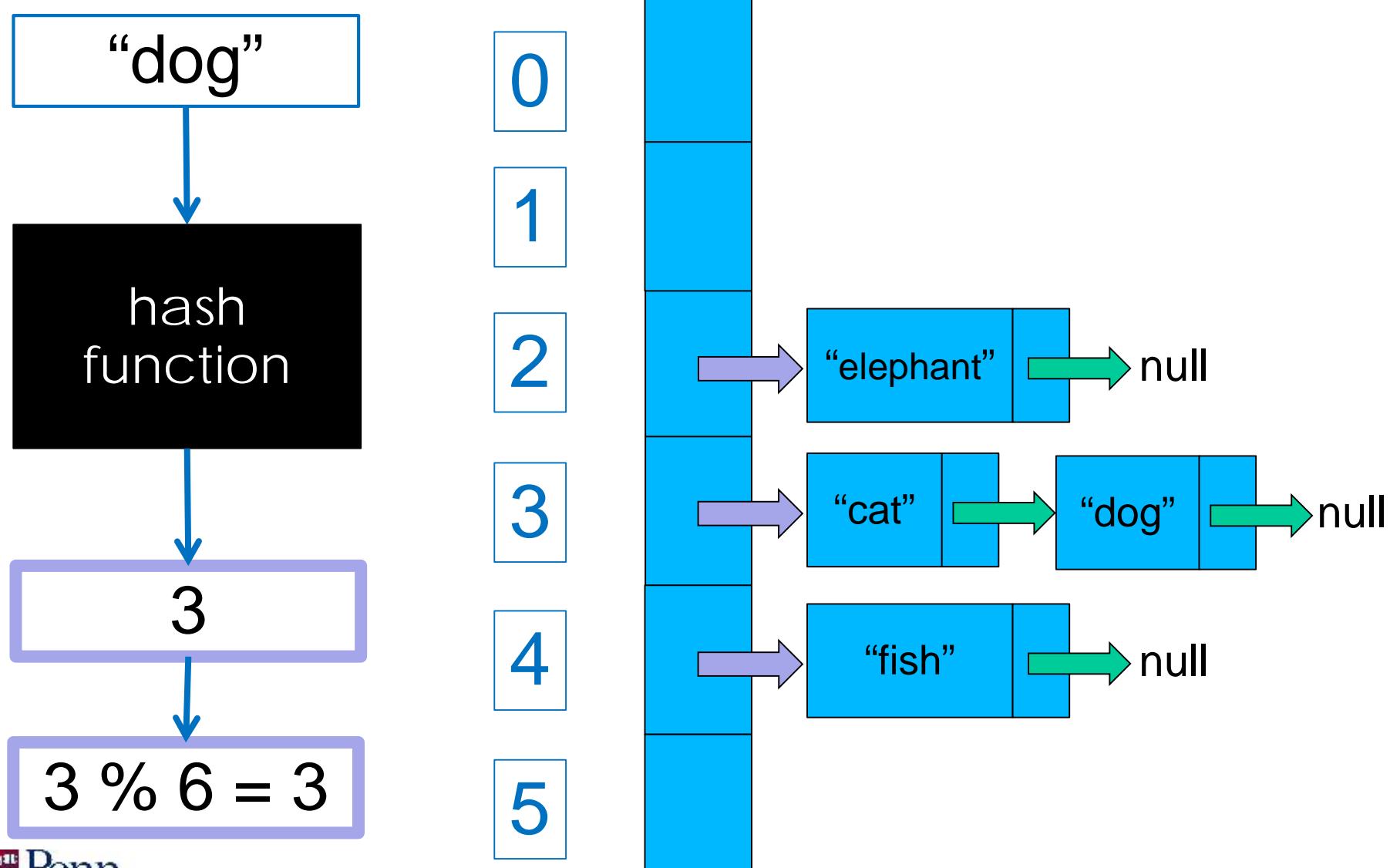
HashSet contains “dog”



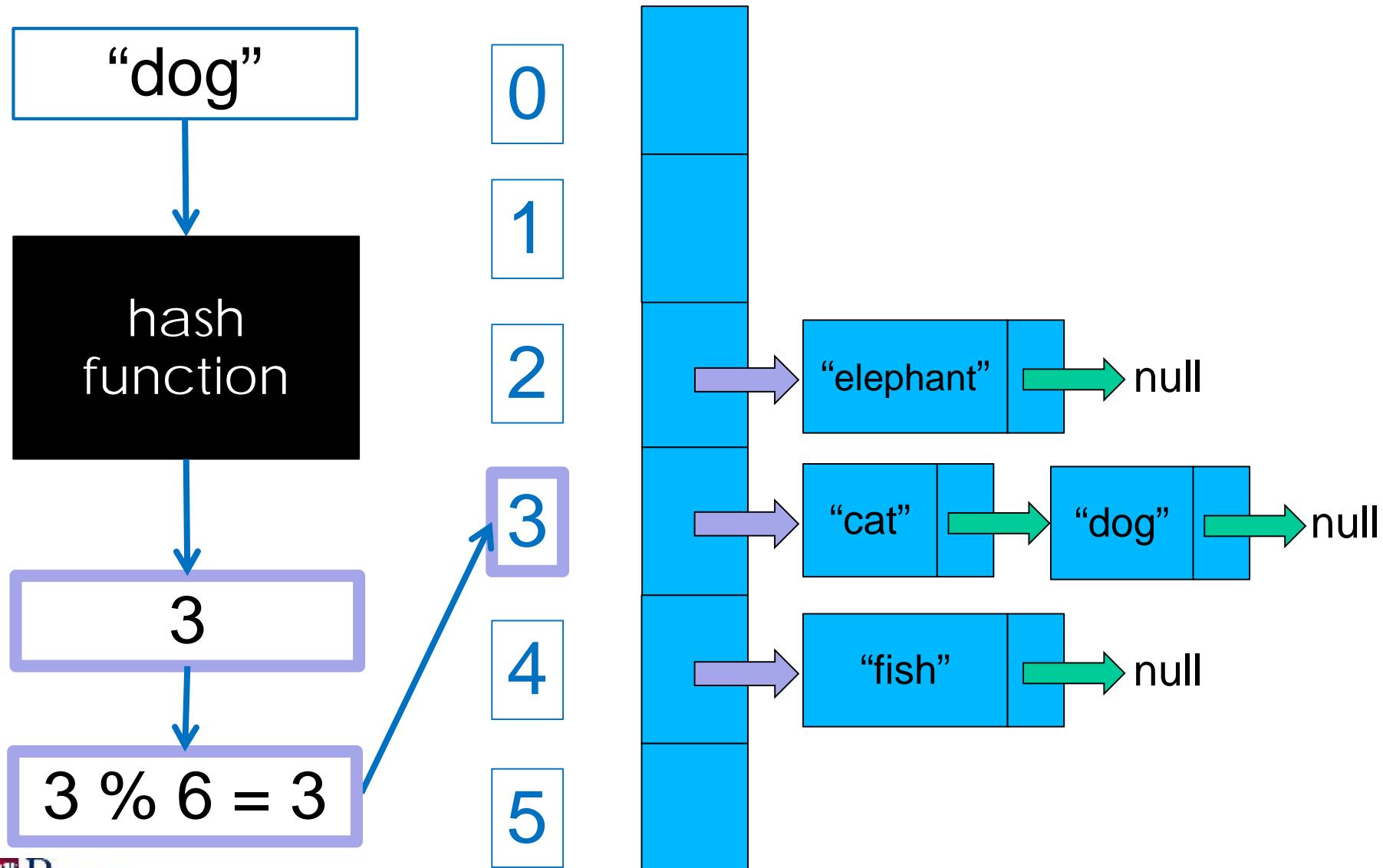
HashSet contains “dog”



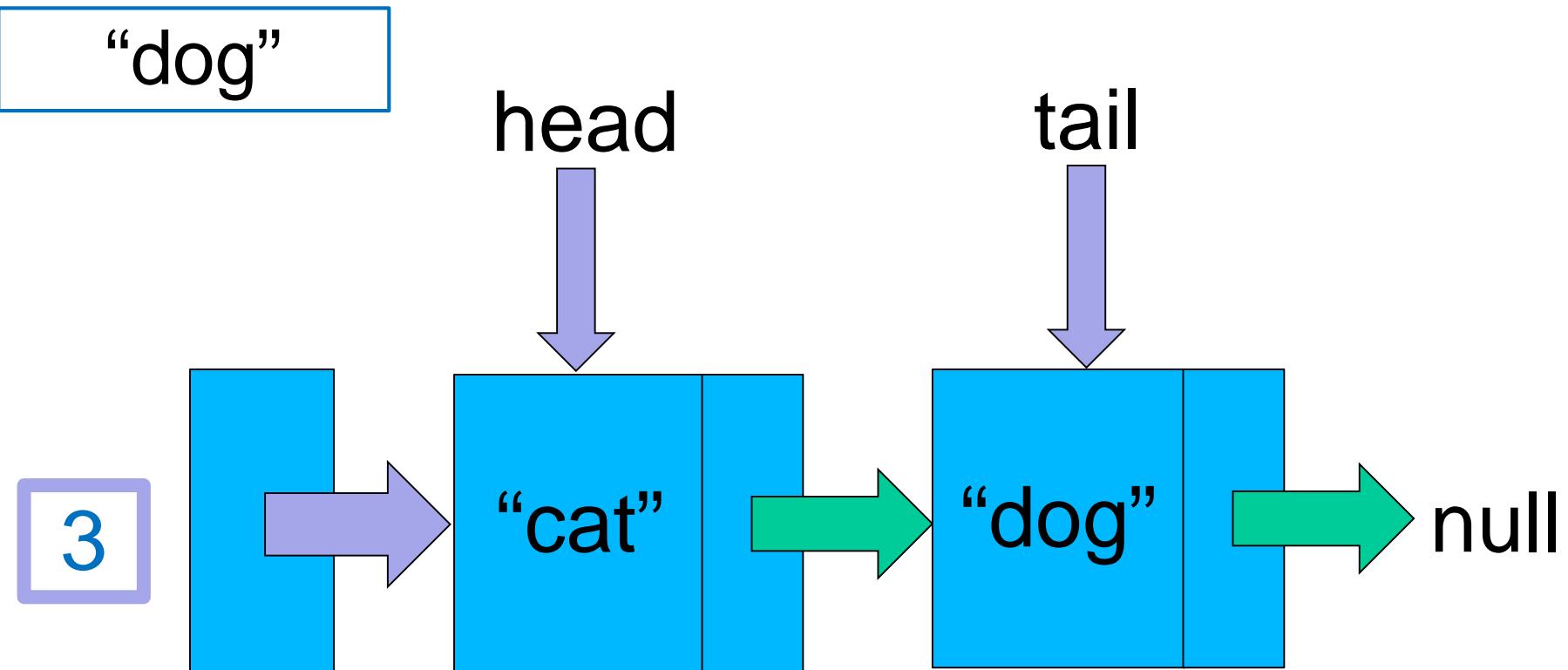
HashSet contains “dog”



HashSet contains “dog”



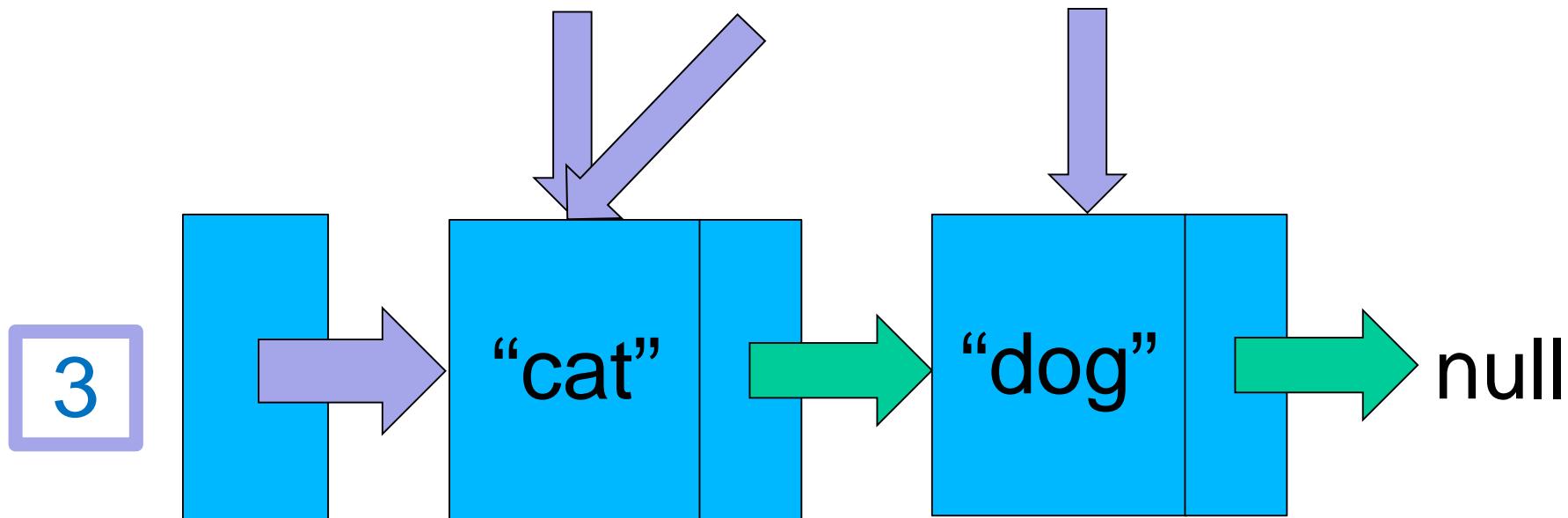
HashSet contains “dog”



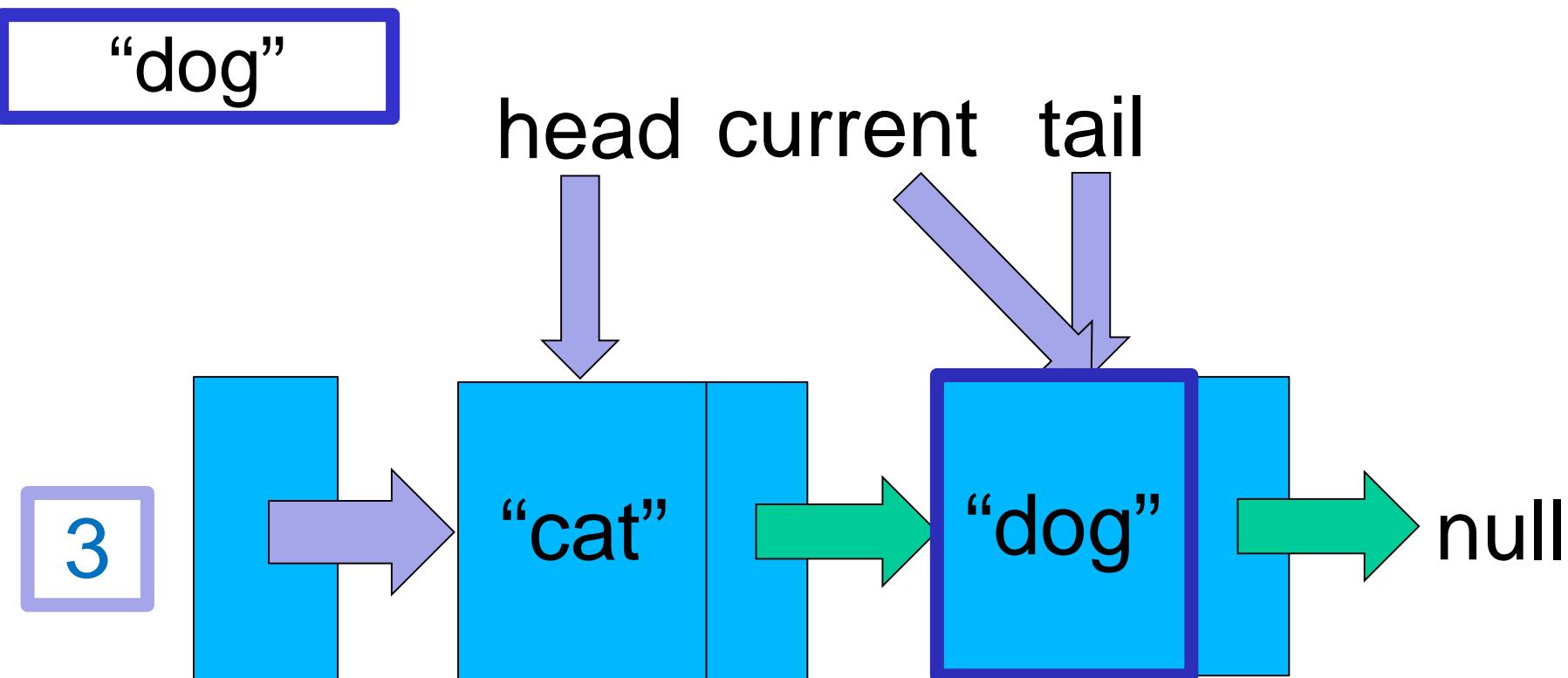
HashSet contains “dog”

“dog”

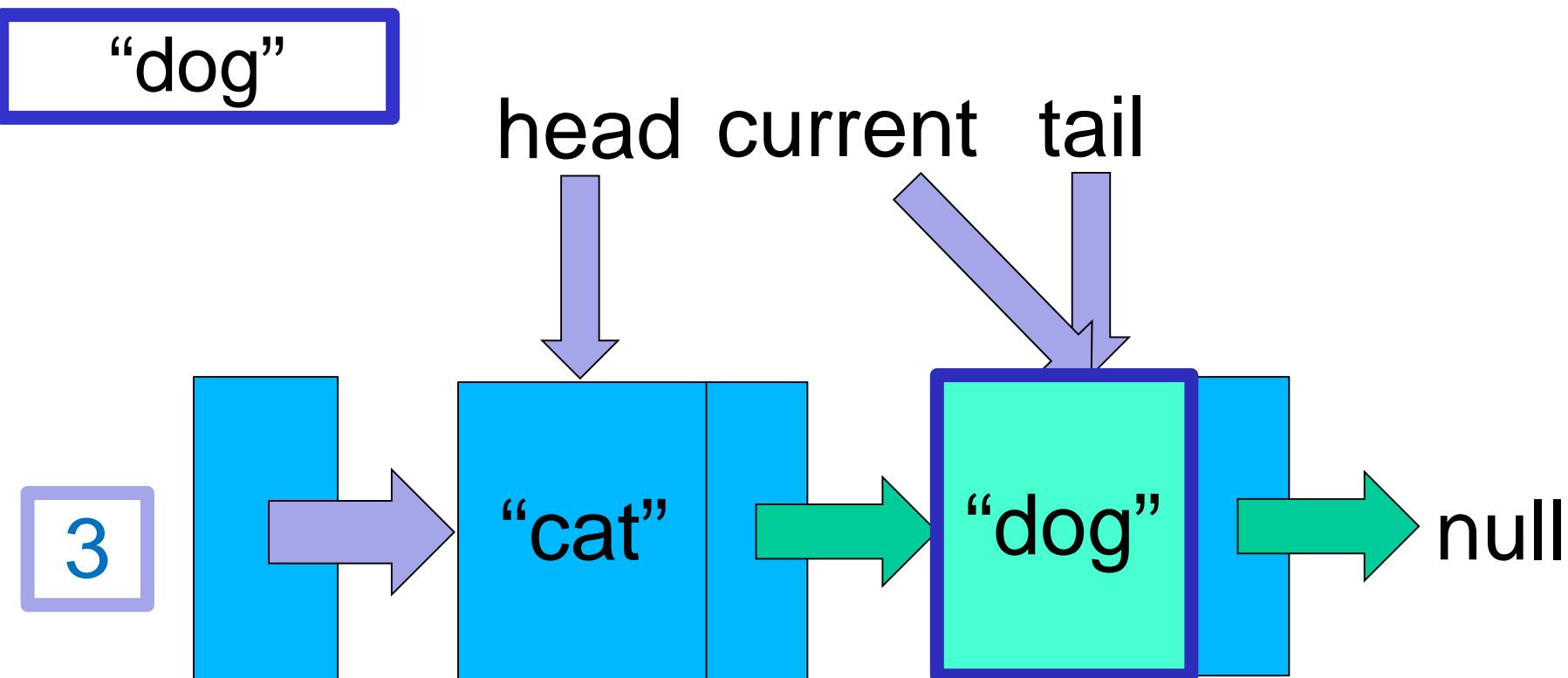
head current tail



HashSet contains “dog”



HashSet contains “dog”



HashSet with separate chaining: contains

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % buckets.length;  
        LinkedList<String> bucket = buckets[index];  
        return bucket.contains(value);  
    }  
}
```

HashSet with separate chaining: contains

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
public boolean contains(String value) {  
    int index = hashCode(value) % buckets.length;  
    LinkedList<String> bucket = buckets[index];  
    return bucket.contains(value);  
}  
}
```

HashSet with separate chaining: contains

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % buckets.length;  
        LinkedList<String> bucket = buckets[index];  
        return bucket.contains(value);  
    }  
}
```

HashSet with separate chaining: contains

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % buckets.length;  
        LinkedList<String> bucket = buckets[index];  
        return bucket.contains(value);  
    }  
}
```

HashSet with separate chaining: contains

```
public class HashSet {  
    . . .  
    private LinkedList<String>[ ] buckets;  
  
    private int hashCode(String value) {  
        return value.length();  
    }  
  
    public boolean contains(String value) {  
        int index = hashCode(value) % buckets.length;  
        LinkedList<String> bucket = buckets[index];  
        return bucket.contains(value);  
    }  
}
```

Recap: HashSets with Separate Chaining

- Elements are stored in arrays of Linked Lists, or “buckets”
- Once the hash code is used to determine the bucket number, all operations are the same as for Linked Lists

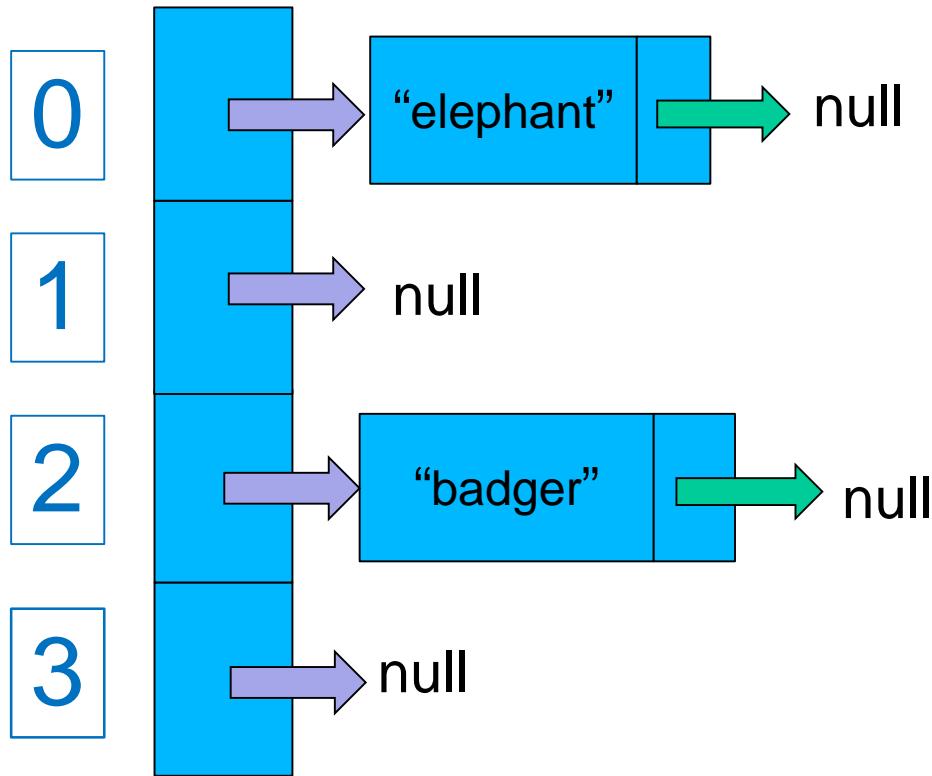
SD2x1.10

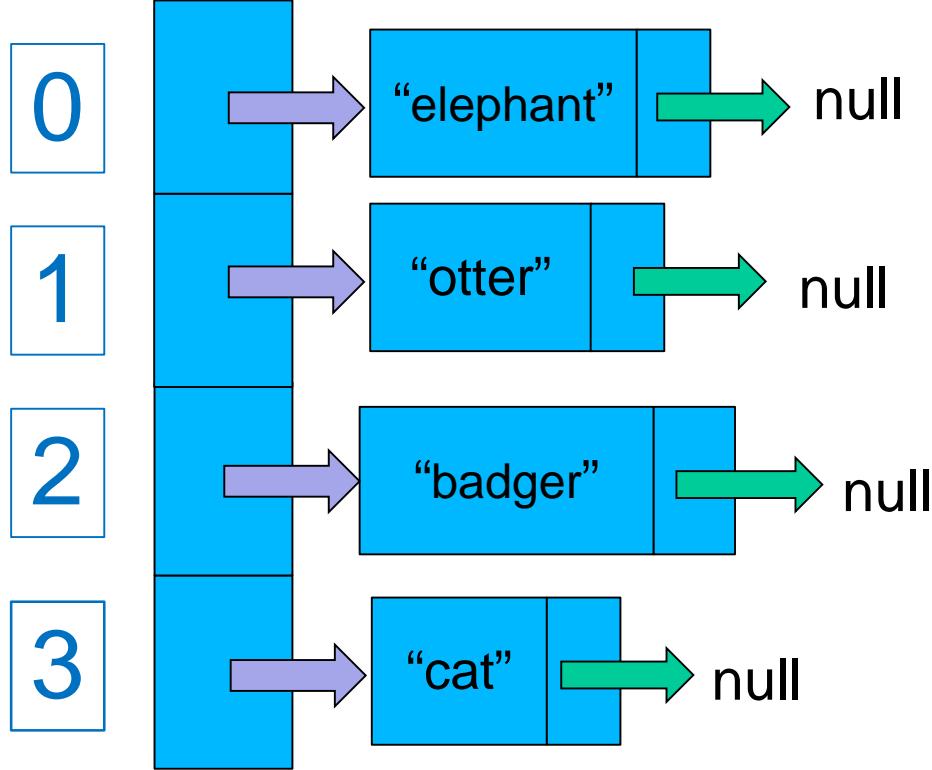
Hash sets

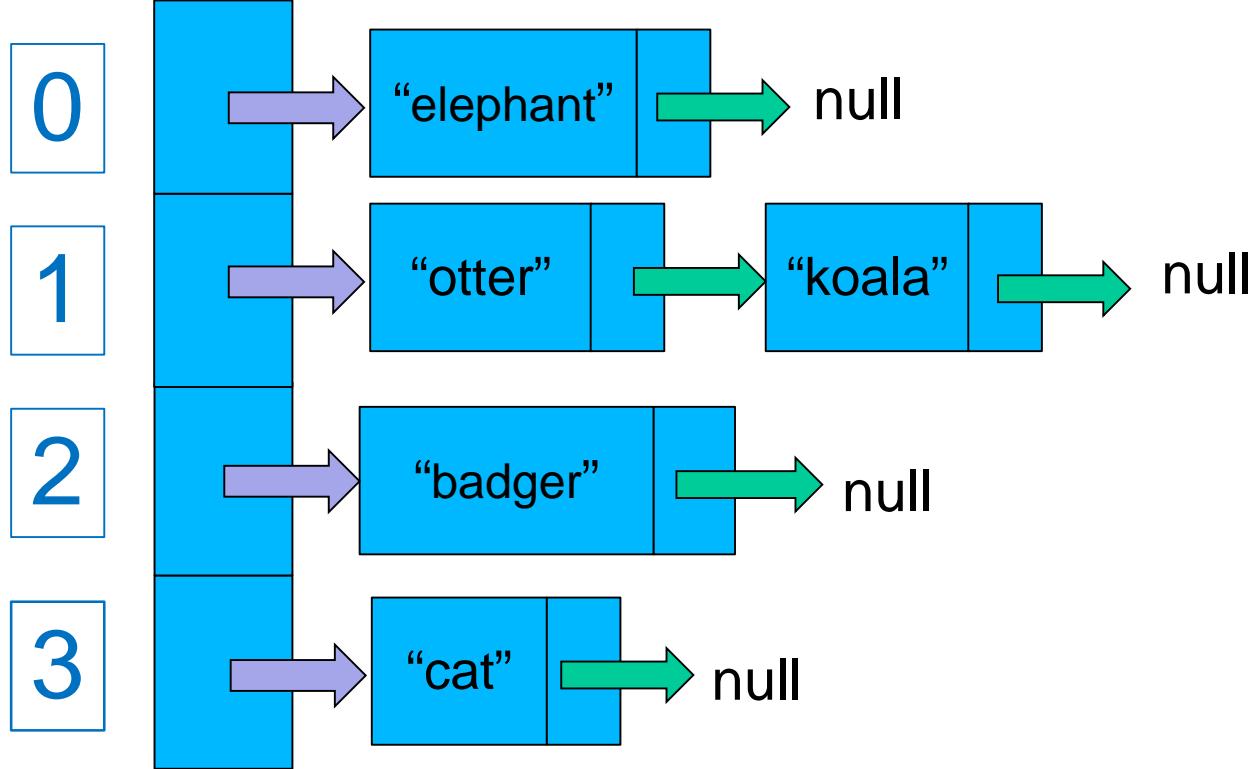
(init capacity, load factor)

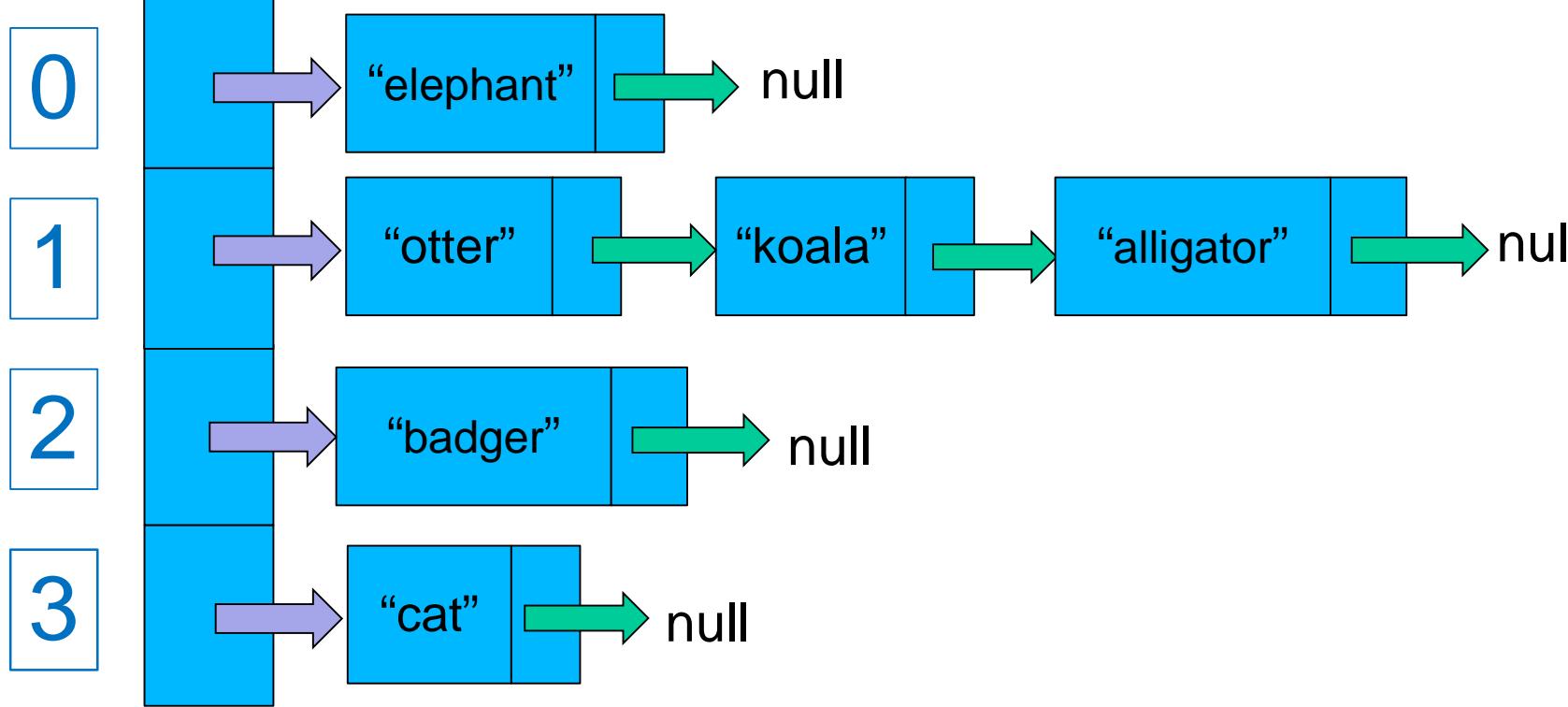
Kathy

What happens if the buckets get too full?









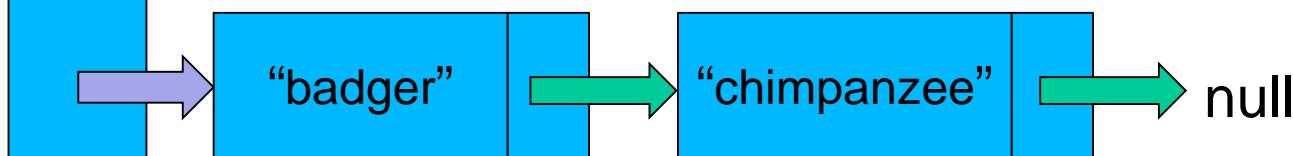
0



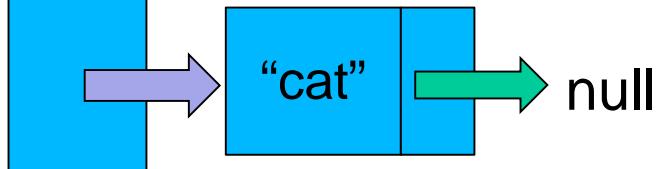
1



2



3



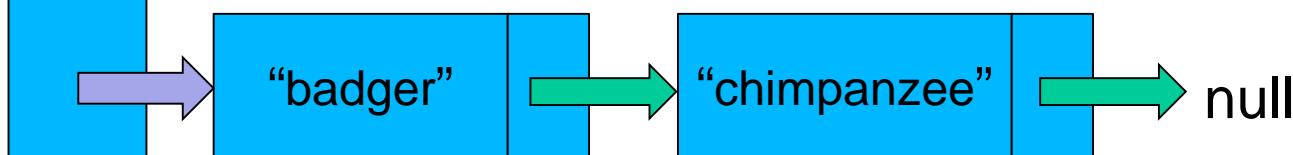
0



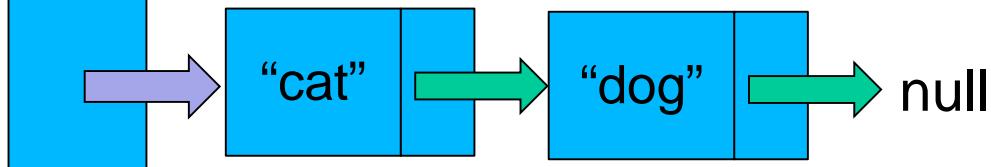
1



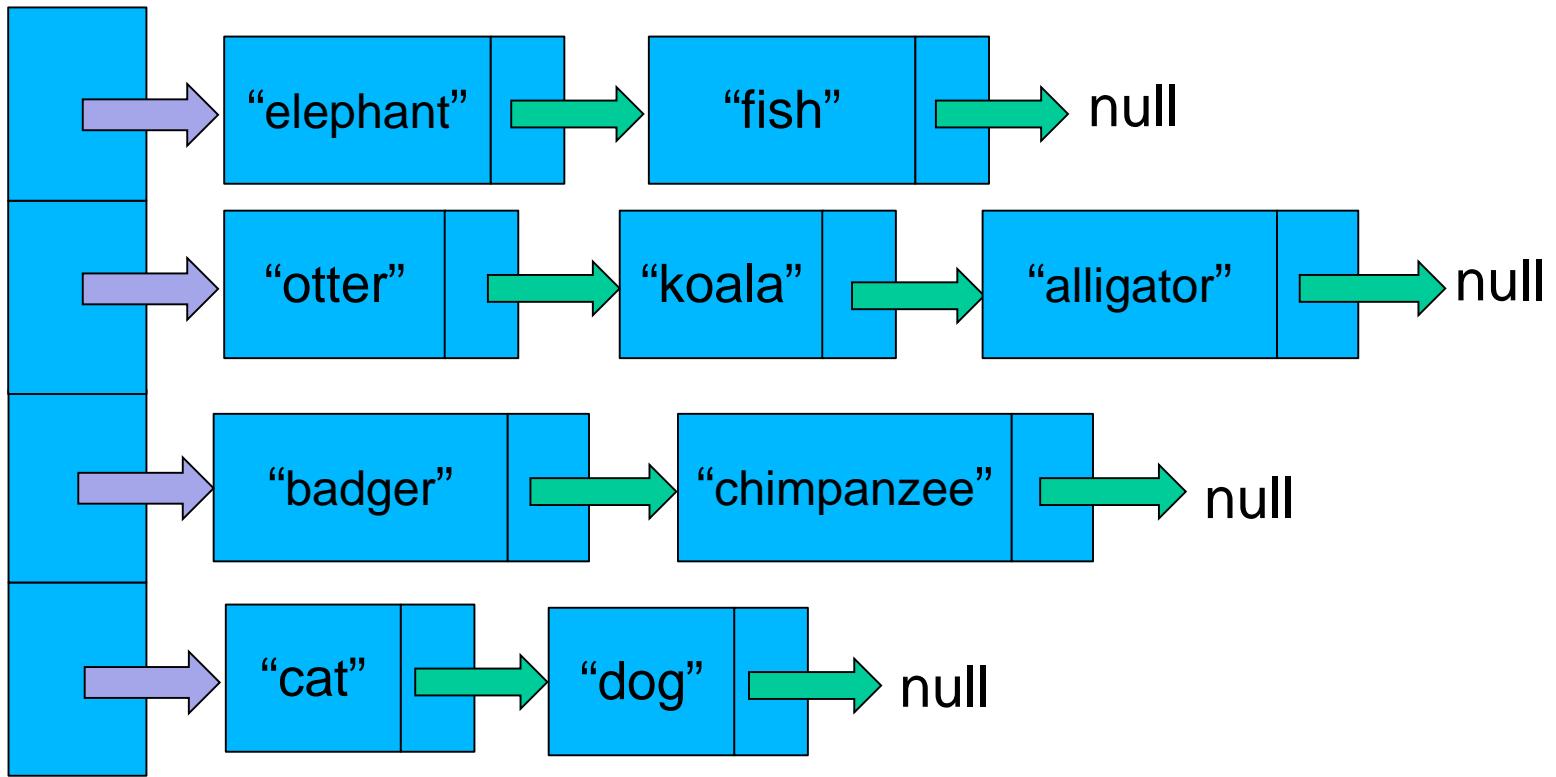
2



3



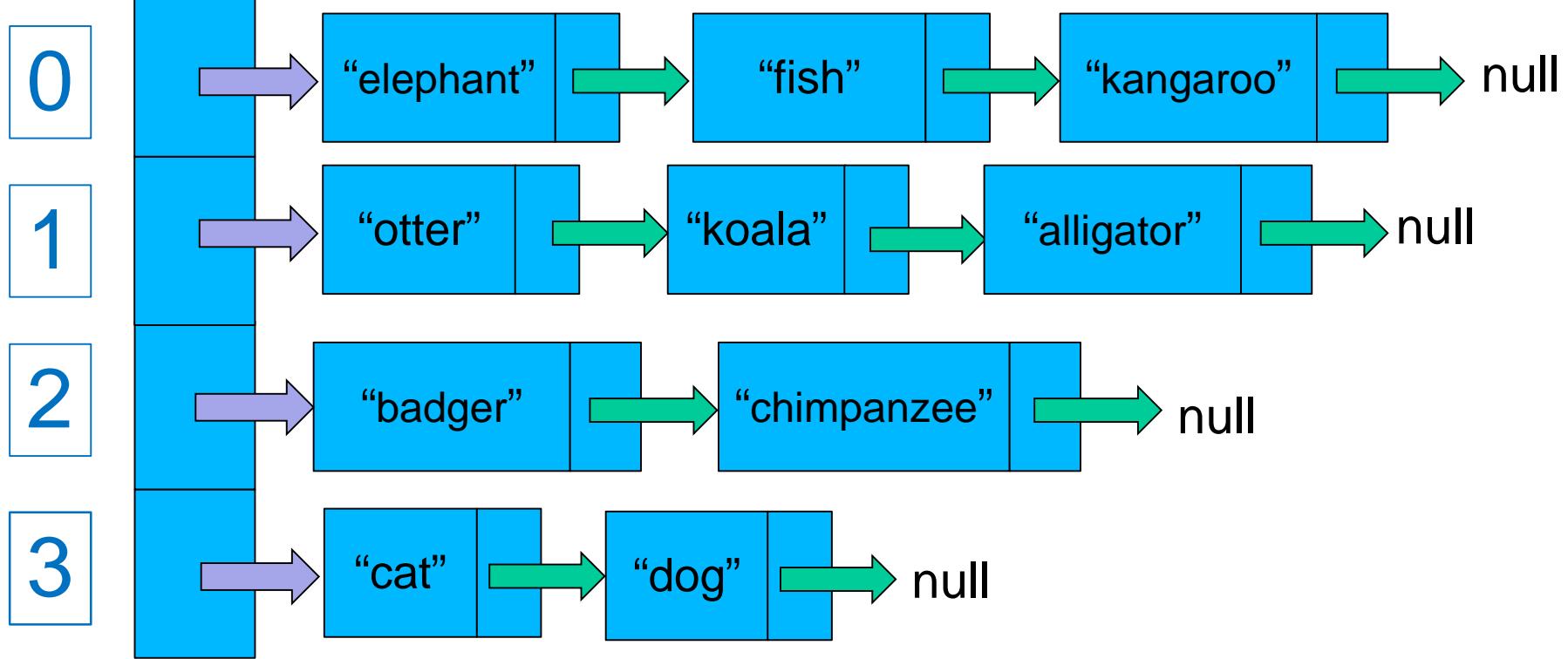
0

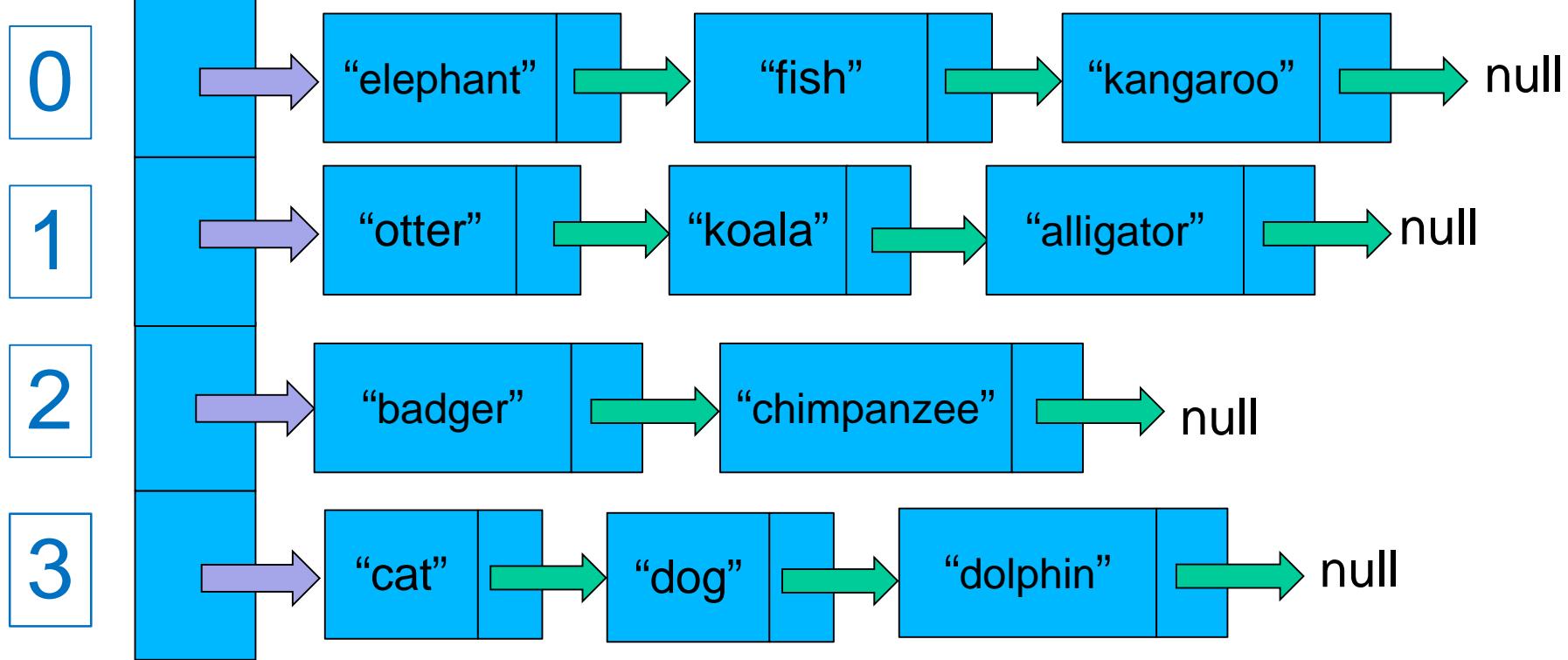


1

2

3





Solution:

Let the HashSet automatically
resize the number of buckets

Basic HashSet

```
public class HashSet {  
  
    private LinkedList<String>[ ] buckets;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    . . .  
}
```

Self-Resizing HashSet

```
public class HashSet {  
  
    private LinkedList<String>[] buckets;  
    private int currentSize = 0;  
  
    public HashSet(int size) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
    }  
  
    . . .  
}
```

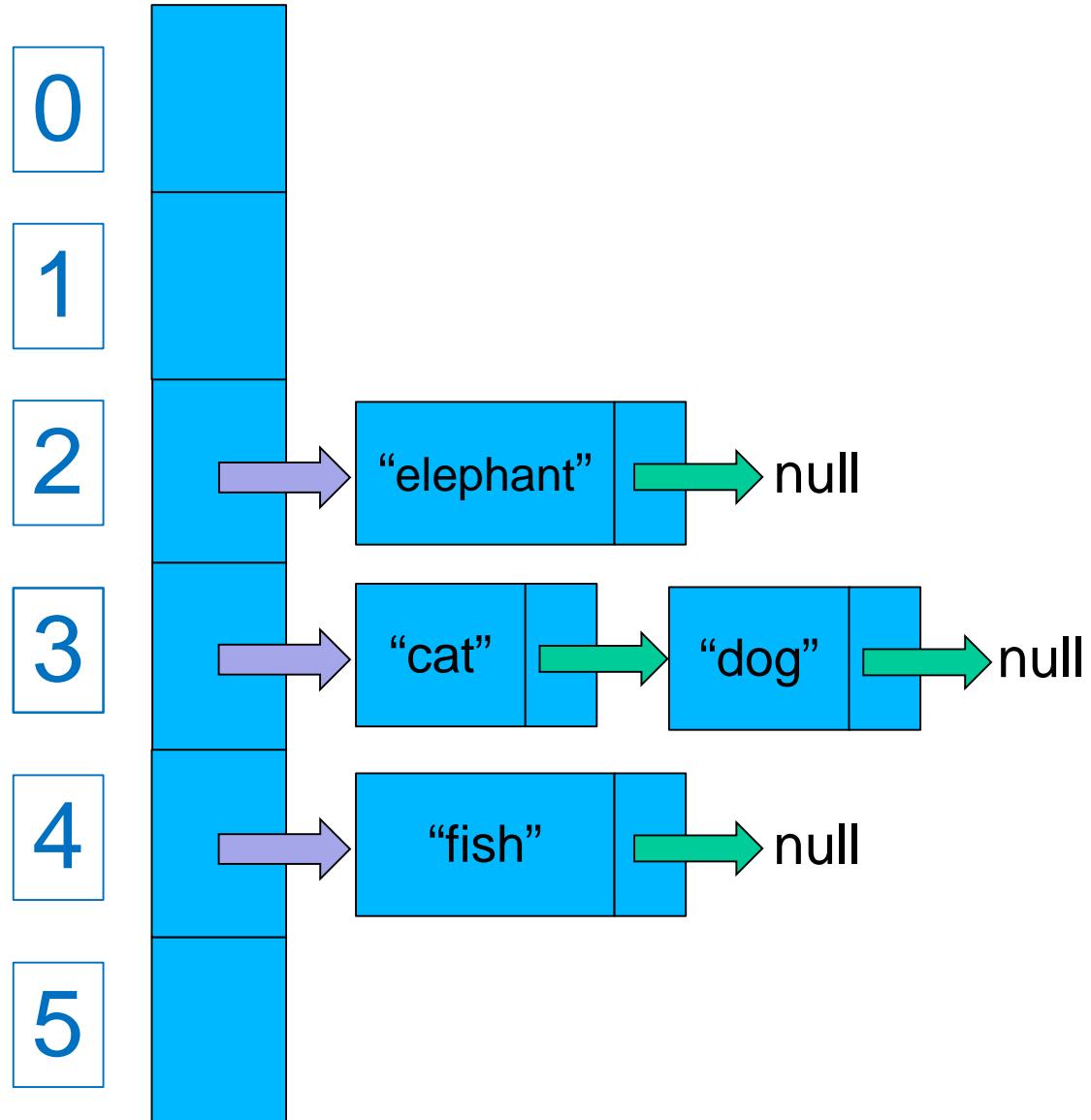
Self-Resizing HashSet

```
public class HashSet {  
  
    private LinkedList<String>[] buckets;  
    private int currentSize = 0;  
    private double loadFactor;  
  
    public HashSet(int size, double loadFactor) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
        this.loadFactor = loadFactor;  
    }  
  
    . . .
```

Self-Resizing HashSet

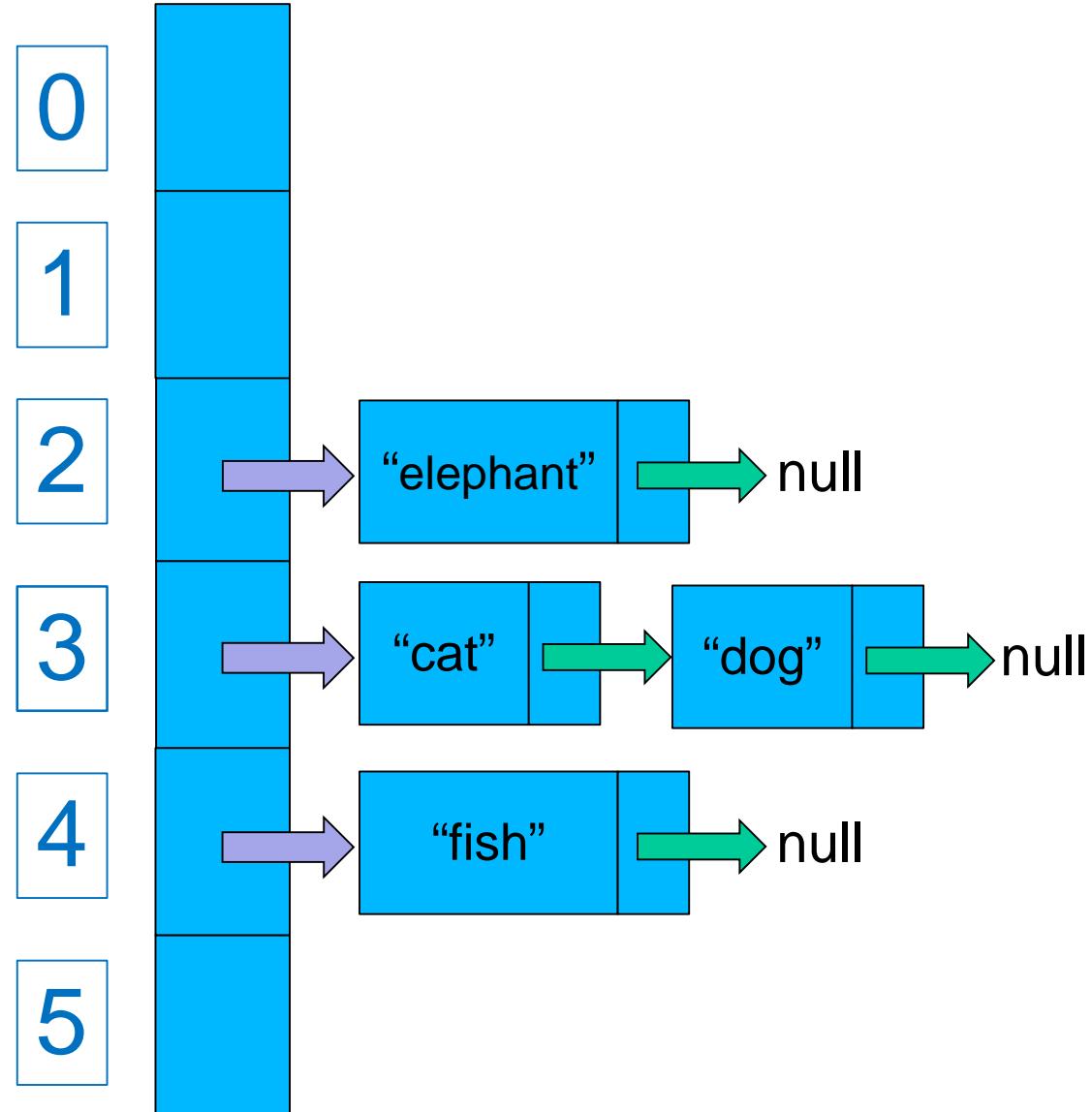
```
public class HashSet {  
  
    private LinkedList<String>[] buckets;  
    private int currentSize = 0;  
    private double loadFactor;  
  
    public HashSet(int size, double loadFactor) {  
        buckets = new LinkedList[size];  
        for (int i = 0; i < size; i++) {  
            buckets[i] = new LinkedList<String>();  
        }  
        this.loadFactor = loadFactor;  
    }  
  
    . . .
```

Balance between time
and space tradeoffs



Operations so far:

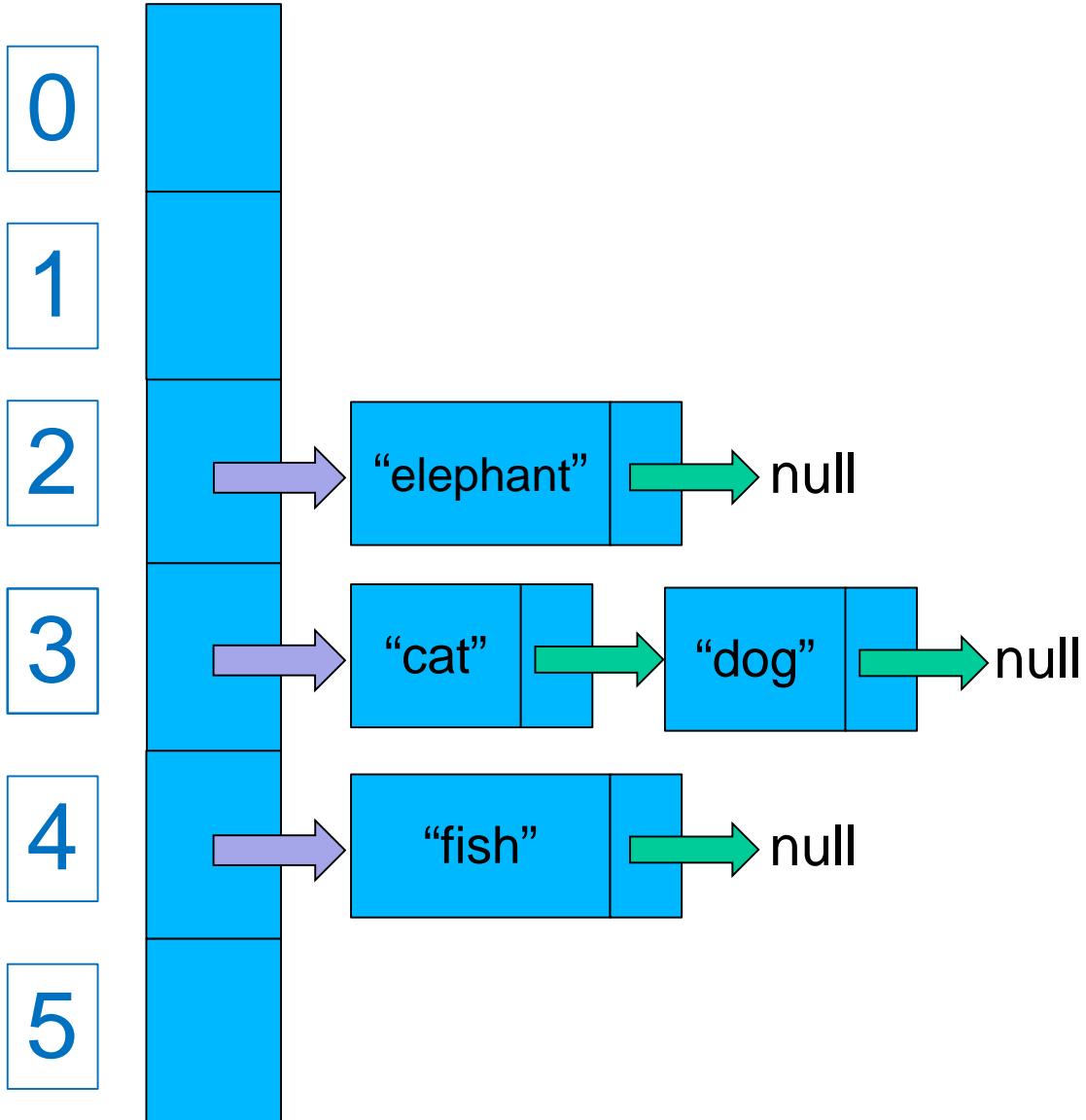
```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```



Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

```
loadFactor = 0.75  
buckets.length = 6
```

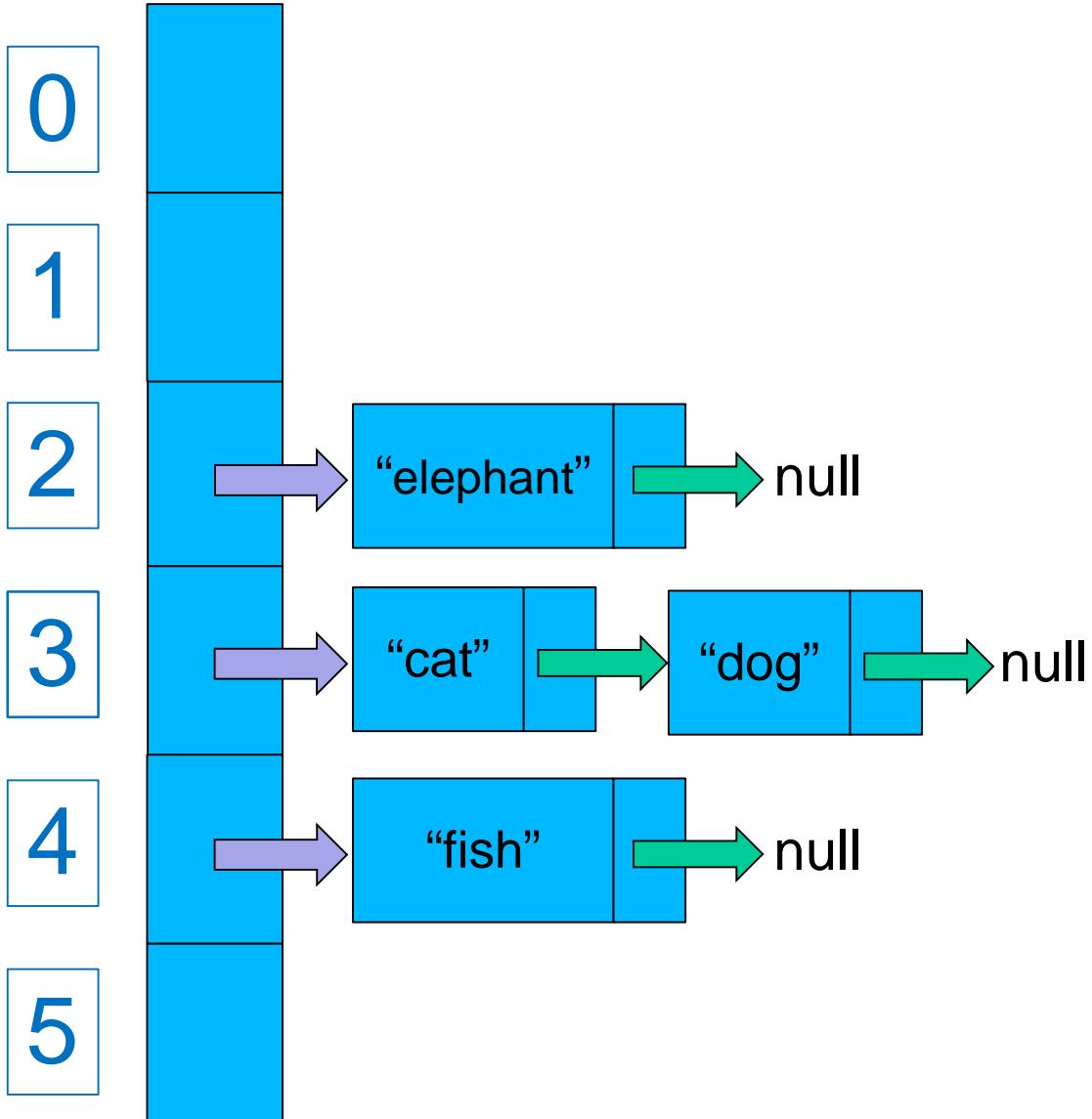


Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

`loadFactor = 0.75`
`buckets.length = 6`

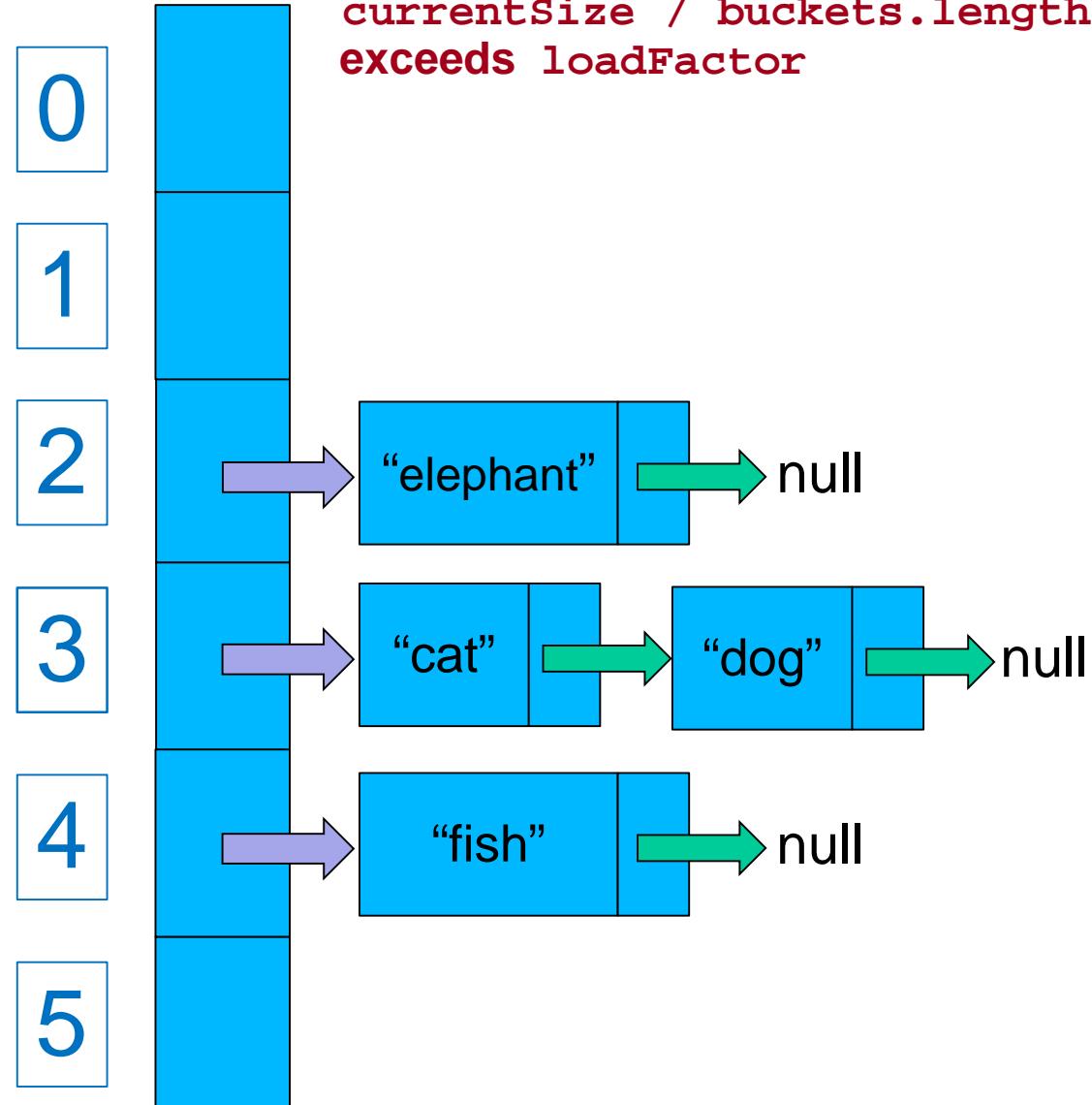
`currentSize = 4`



Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

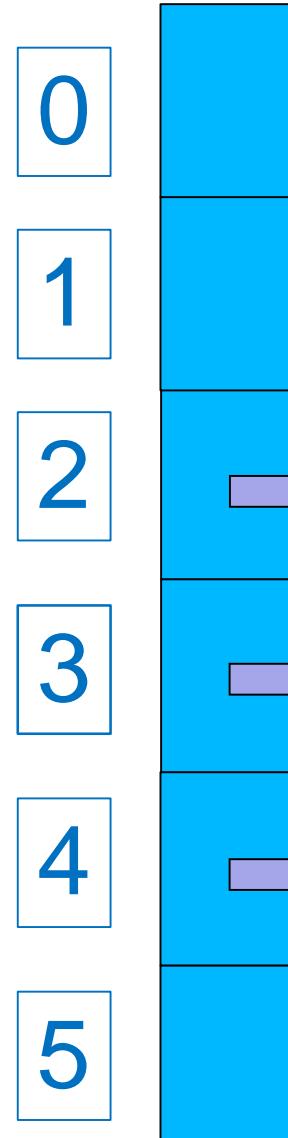
loadFactor = 0.75
buckets.length = 6
currentSize = 4



Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

loadFactor = 0.75
buckets.length = 6
currentSize = 4



Resize when the average number of elements per bucket, `currentSize / buckets.length`, exceeds `loadFactor`

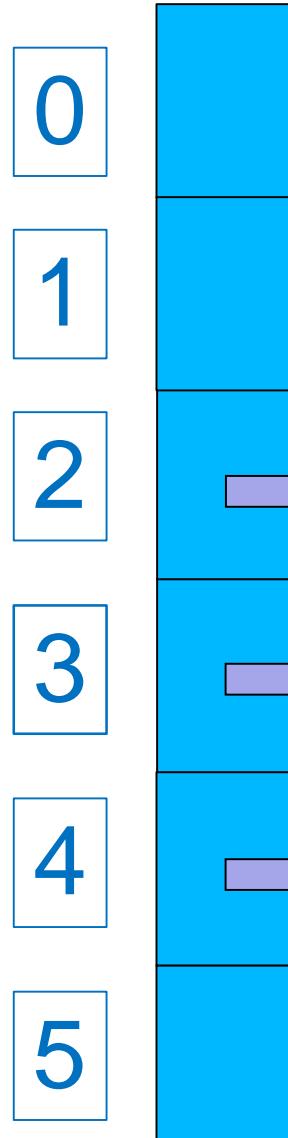
Currently:

$$4 / 6 = 0.67$$

Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

loadFactor = 0.75
buckets.length = 6
currentSize = 4



Resize when the average number of elements per bucket, `currentSize / buckets.length`, exceeds `loadFactor`

Currently:

$$4 / 6 = 0.67$$

No need to resize!

Operations so far:

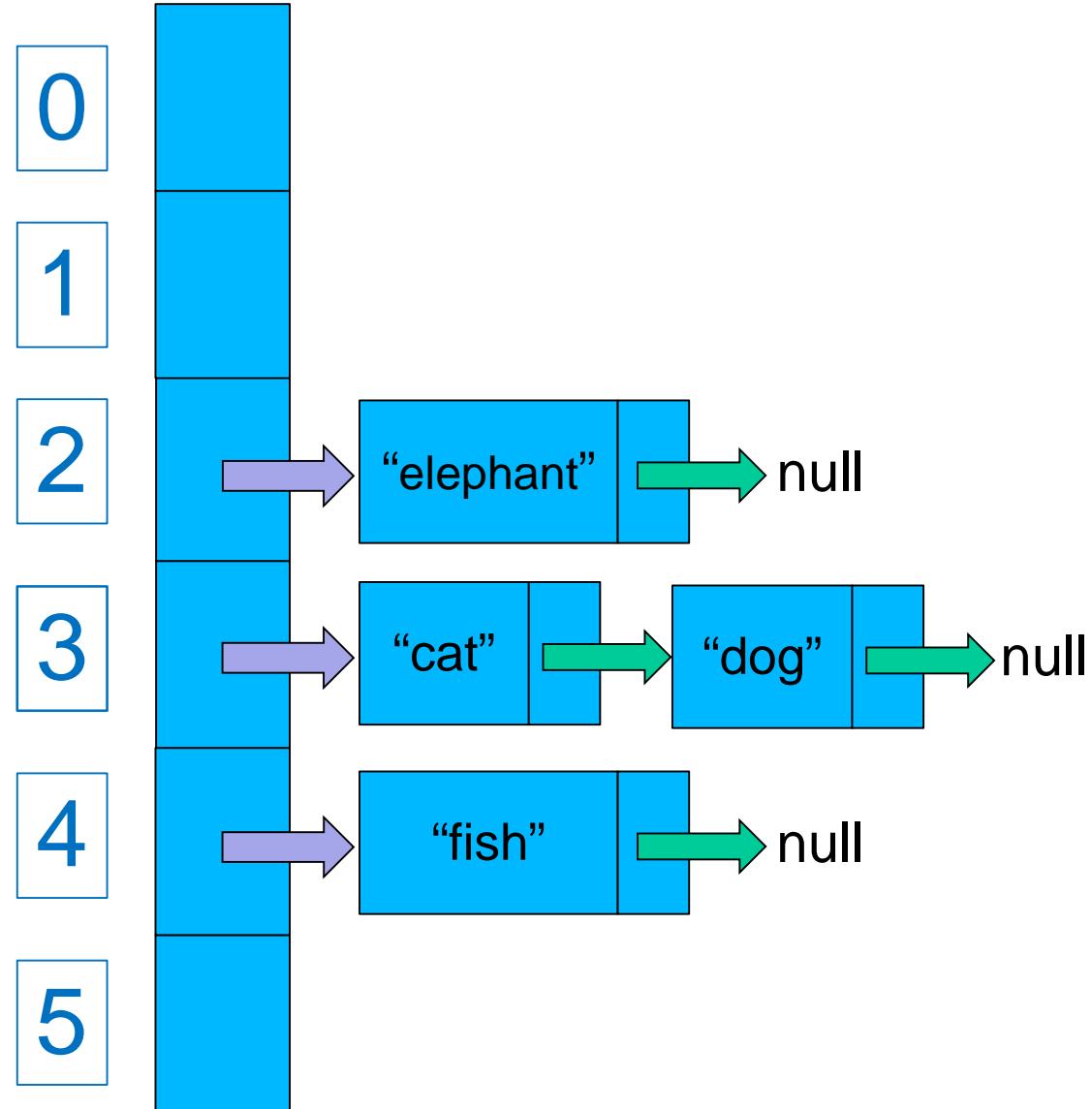
```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

New addition:

hashSet.add("woodchuck");

loadFactor = 0.75
buckets.length = 6

currentSize = 4



Operations so far:

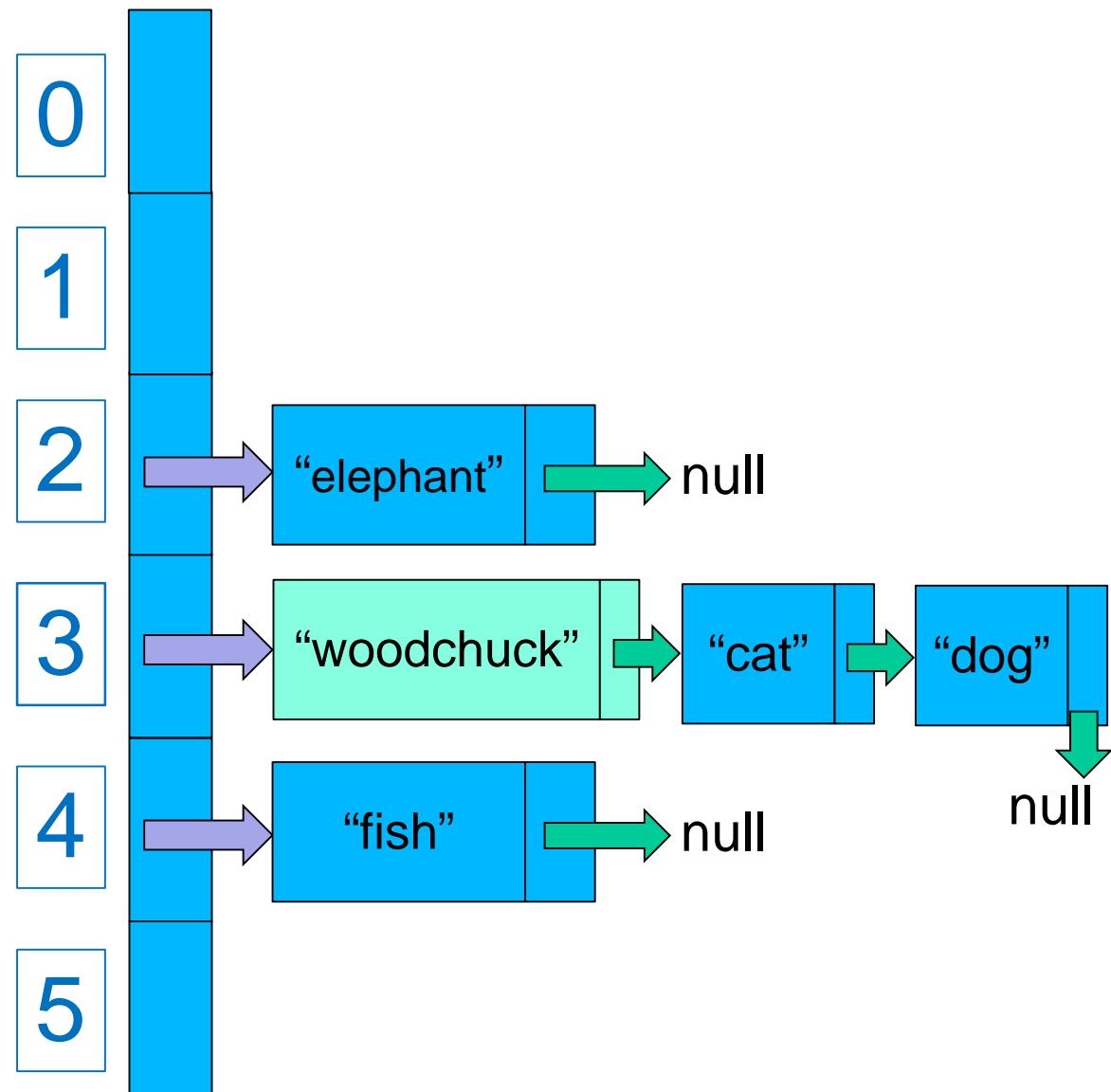
```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

New addition:

```
hashSet.add("woodchuck");
```

```
loadFactor = 0.75  
buckets.length = 6
```

```
currentSize = 4
```



Operations so far:

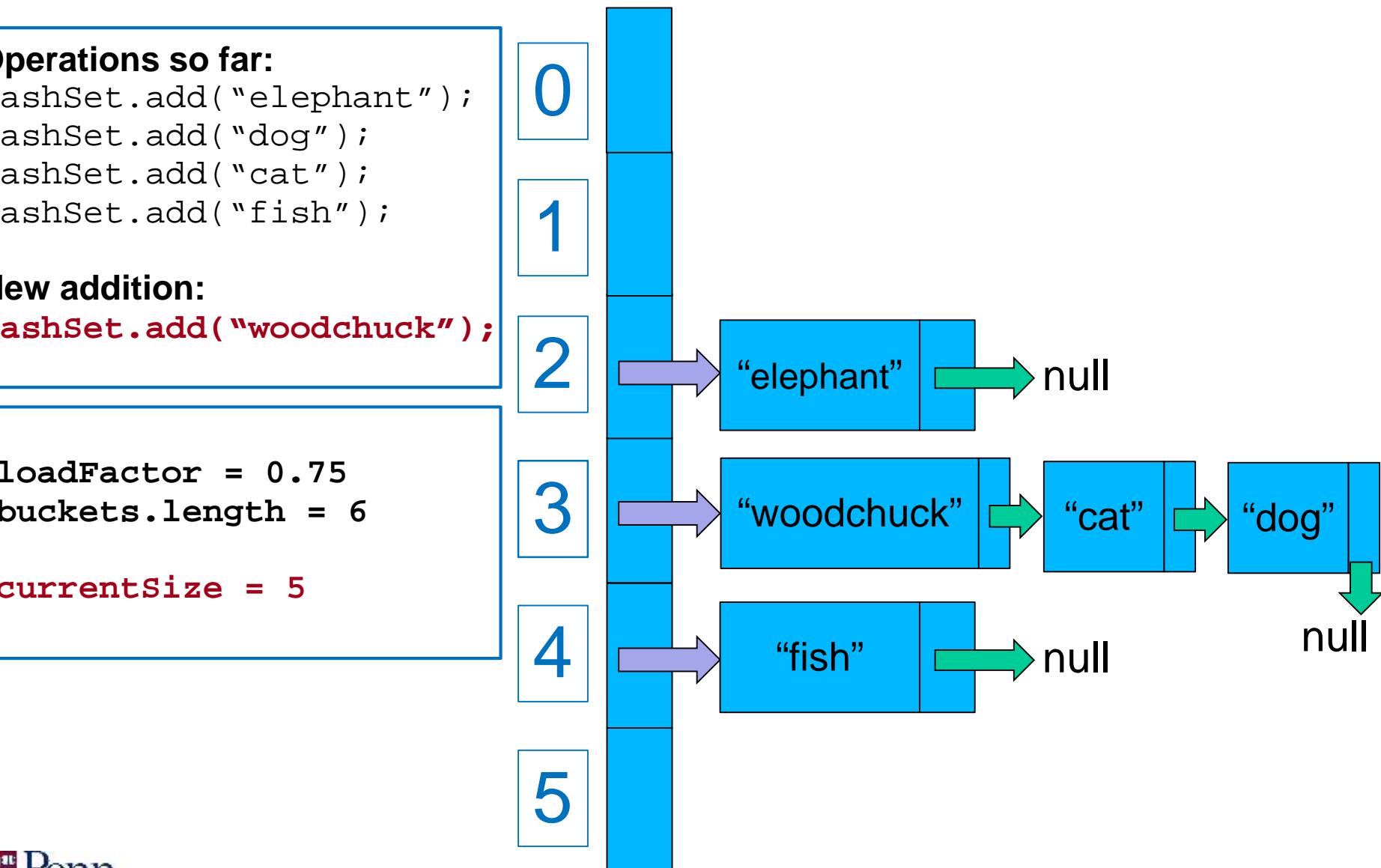
```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

New addition:

```
hashSet.add("woodchuck");
```

```
loadFactor = 0.75  
buckets.length = 6
```

```
currentSize = 5
```



Operations so far:

```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

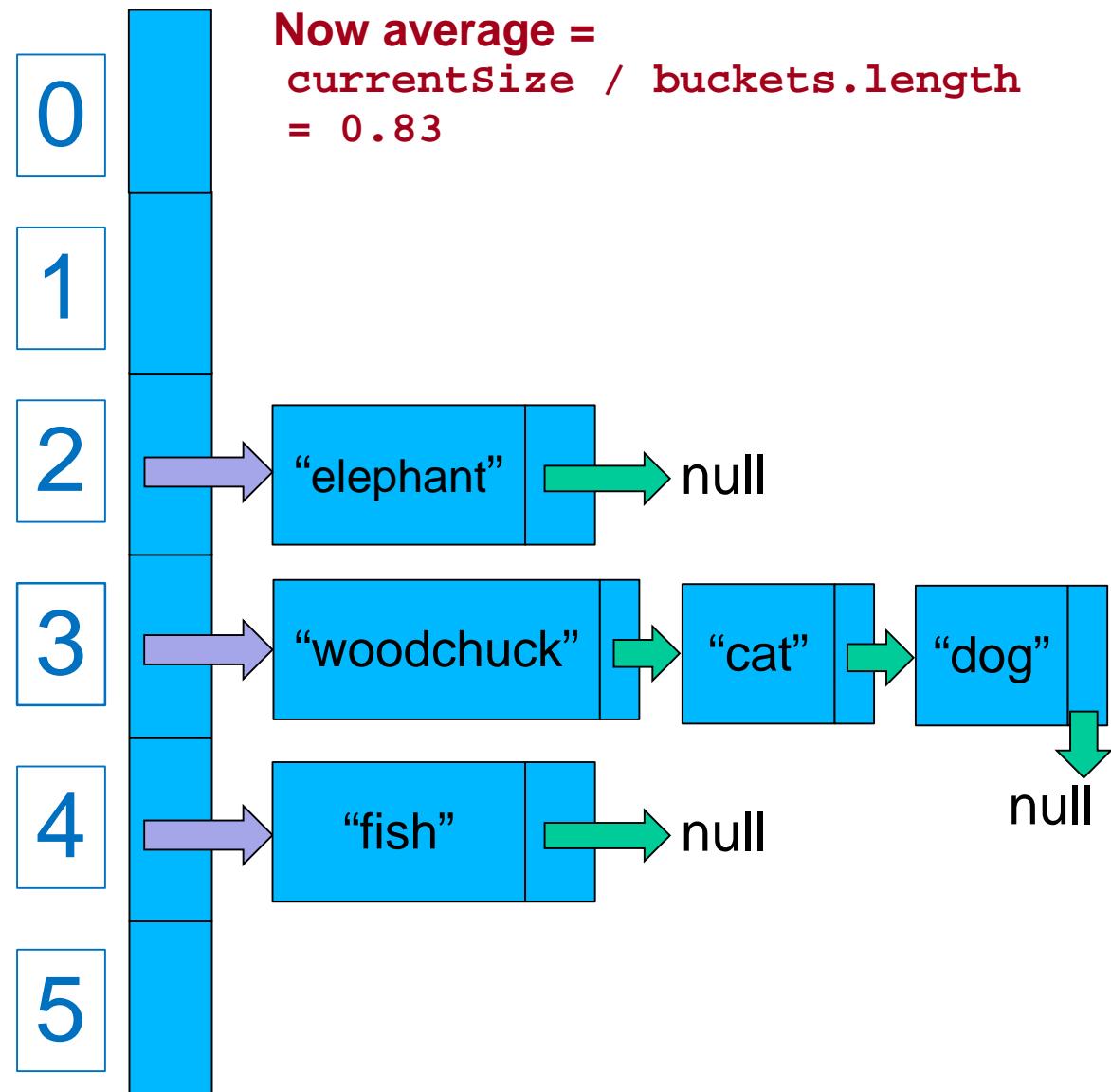
New addition:

```
hashSet.add("woodchuck");
```

loadFactor = 0.75
buckets.length = 6

currentSize = 5

Now average =
currentSize / buckets.length
= 0.83



Operations so far:

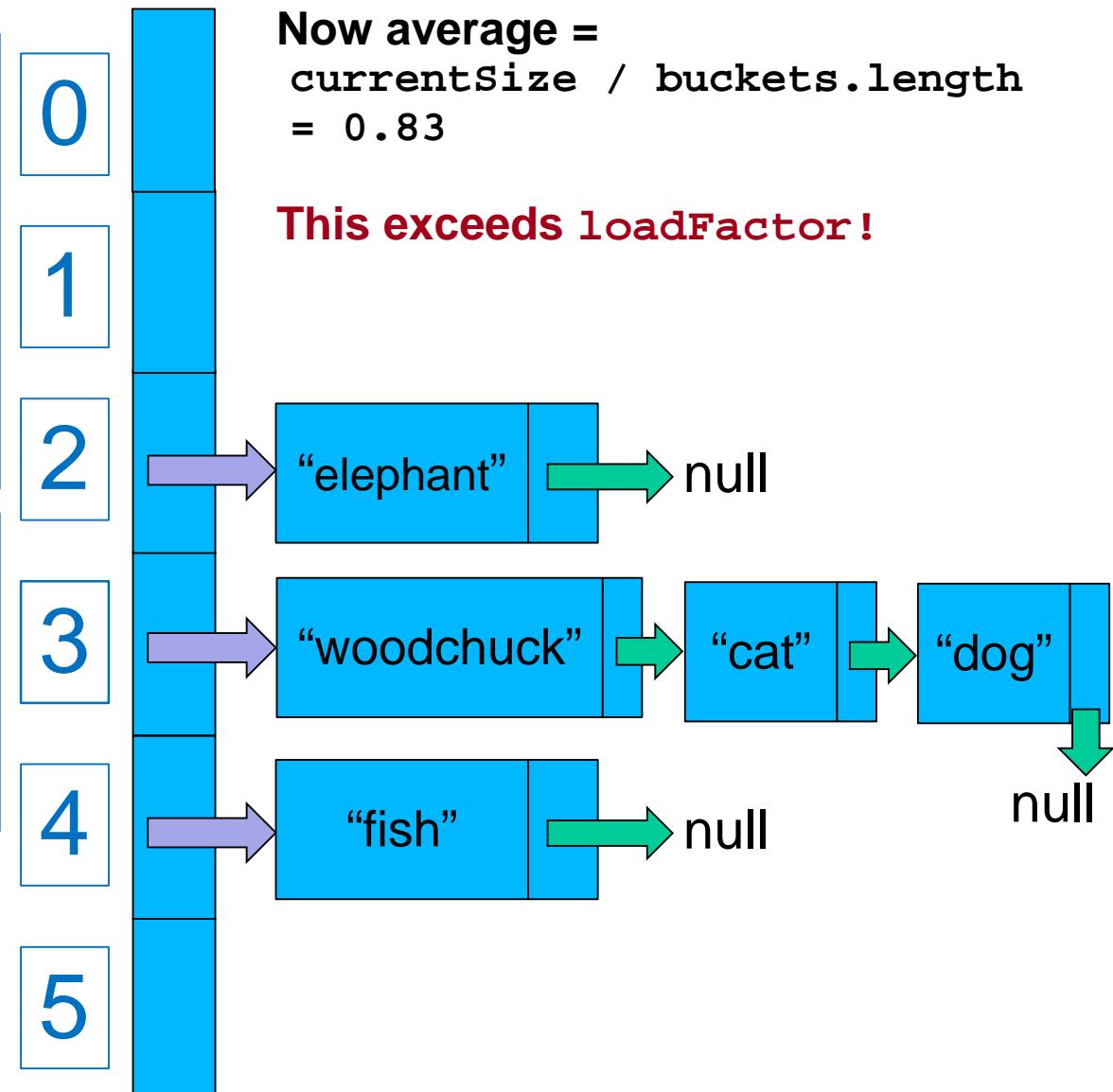
```
hashSet.add("elephant");  
hashSet.add("dog");  
hashSet.add("cat");  
hashSet.add("fish");
```

New addition:

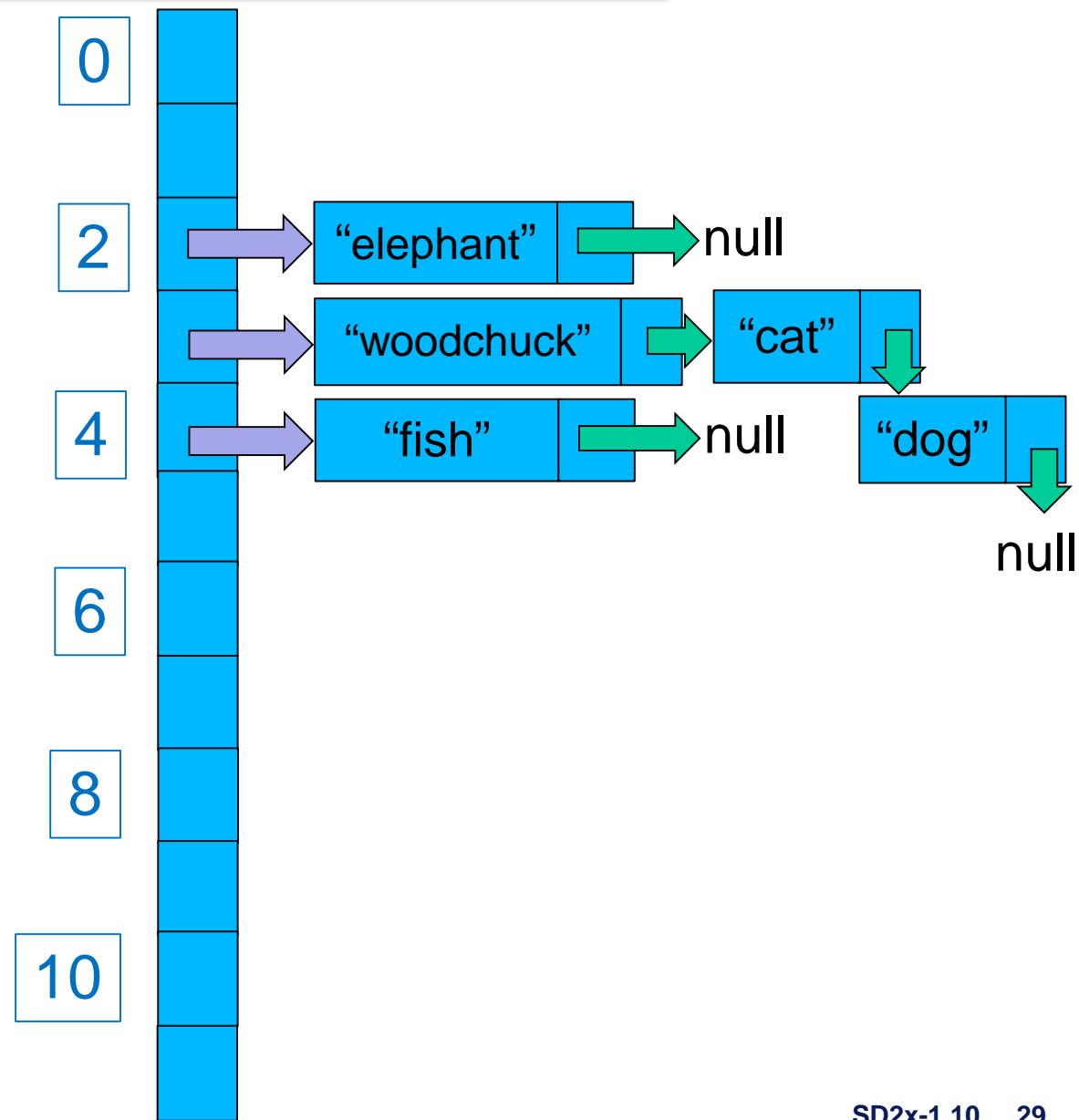
hashSet.add("woodchuck");

loadFactor = 0.75
buckets.length = 6

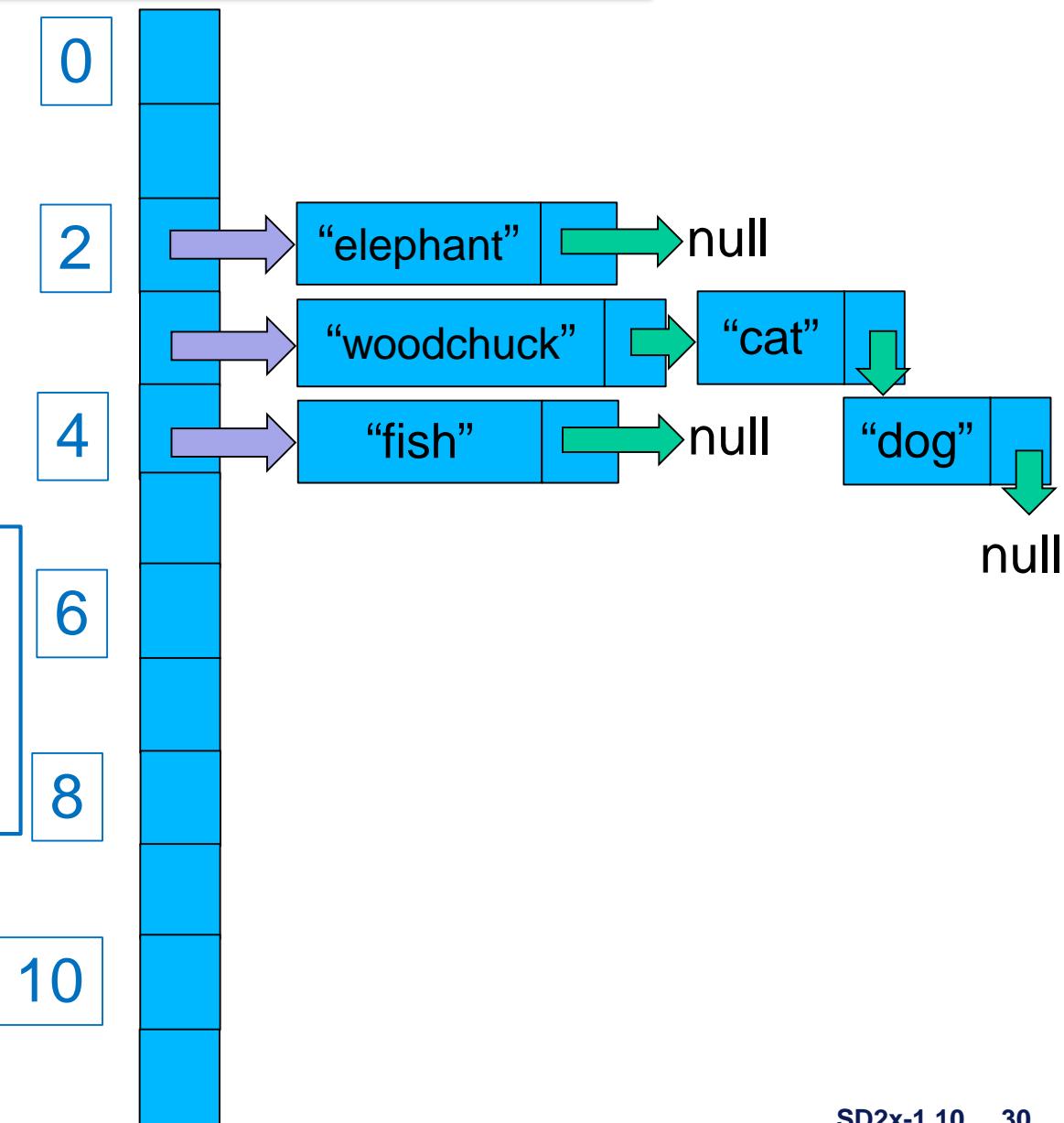
currentSize = 5



1. Double number of buckets



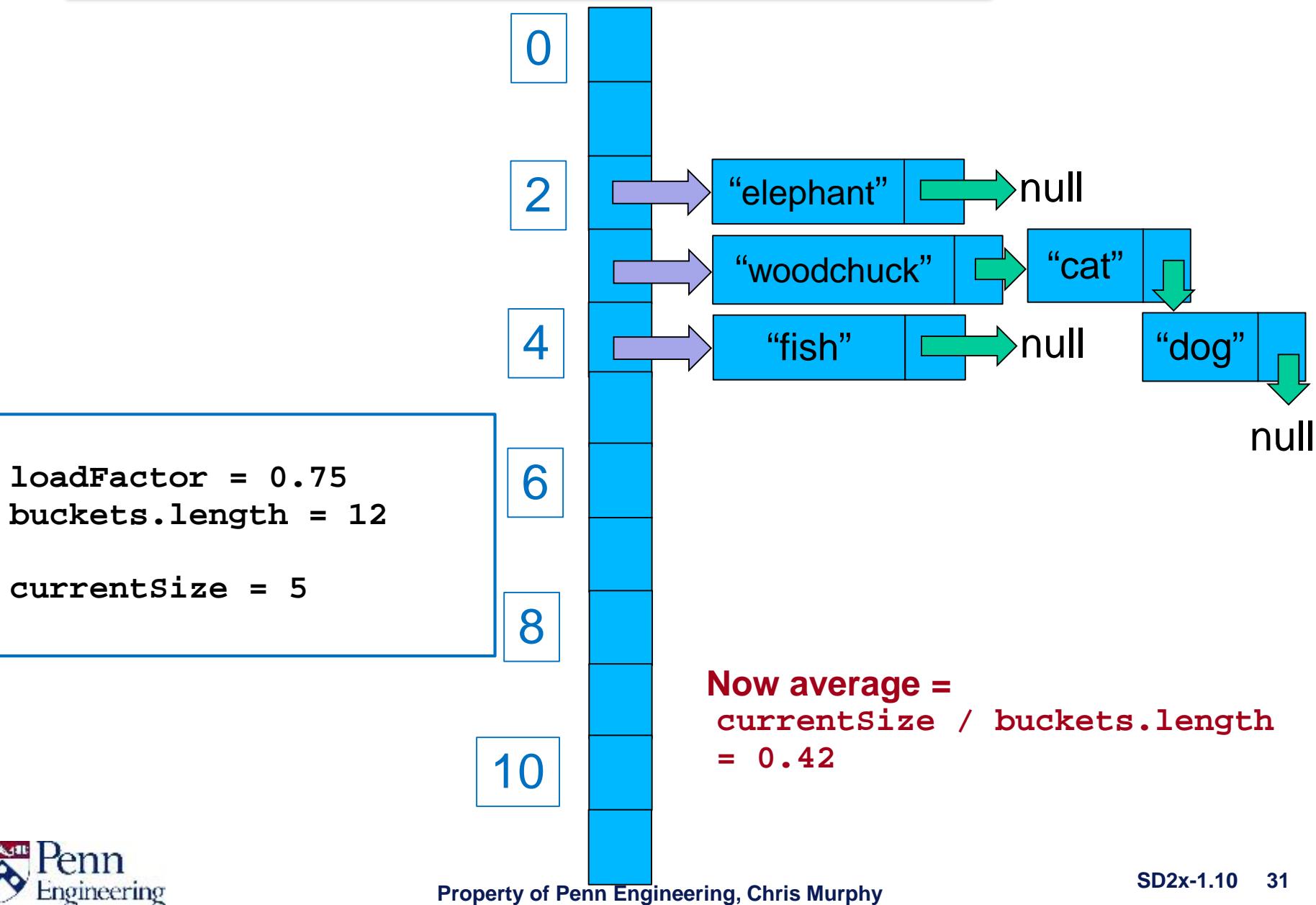
1. Double number of buckets



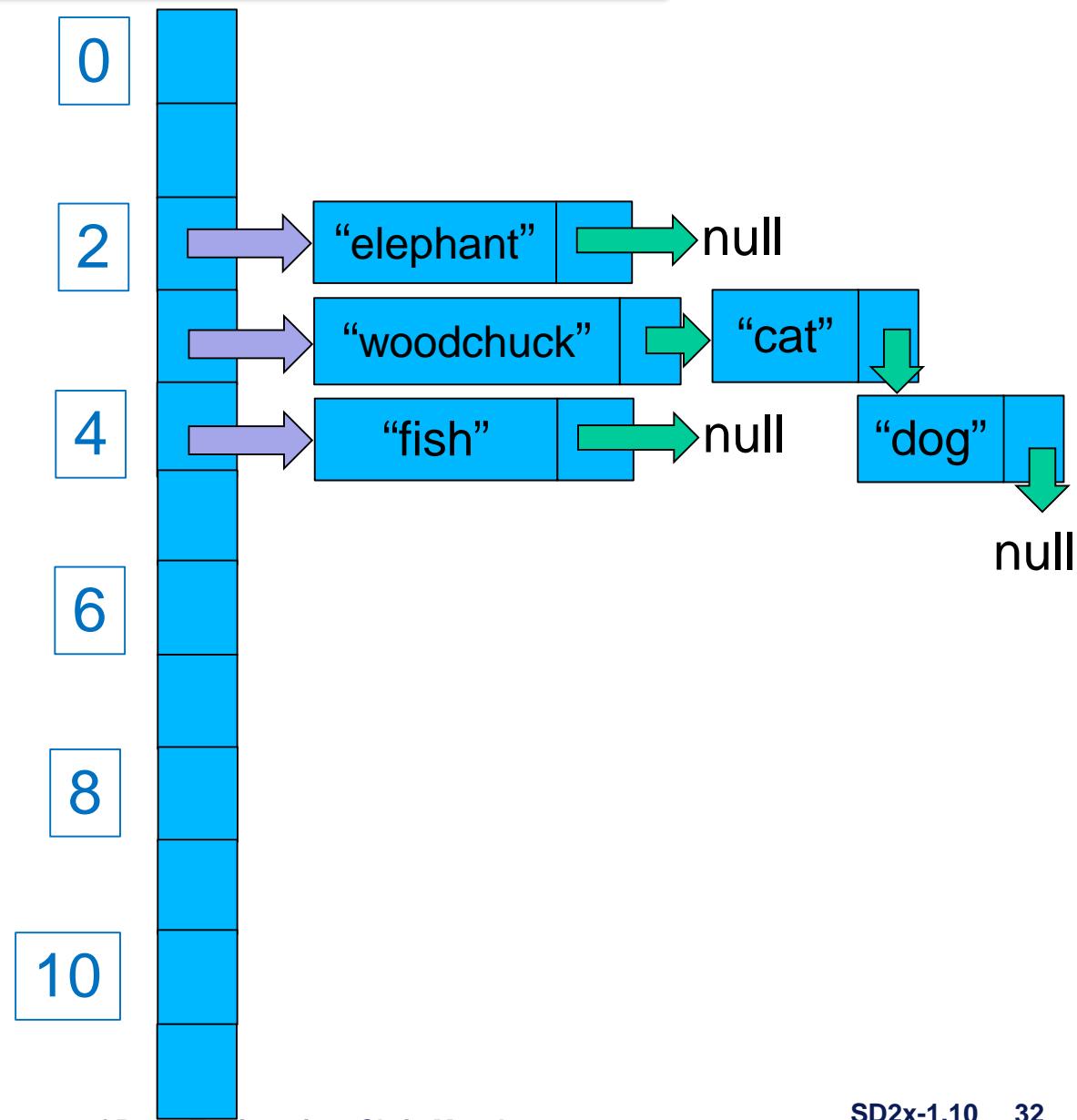
SD2x-1.10 30

Penn
Engineering

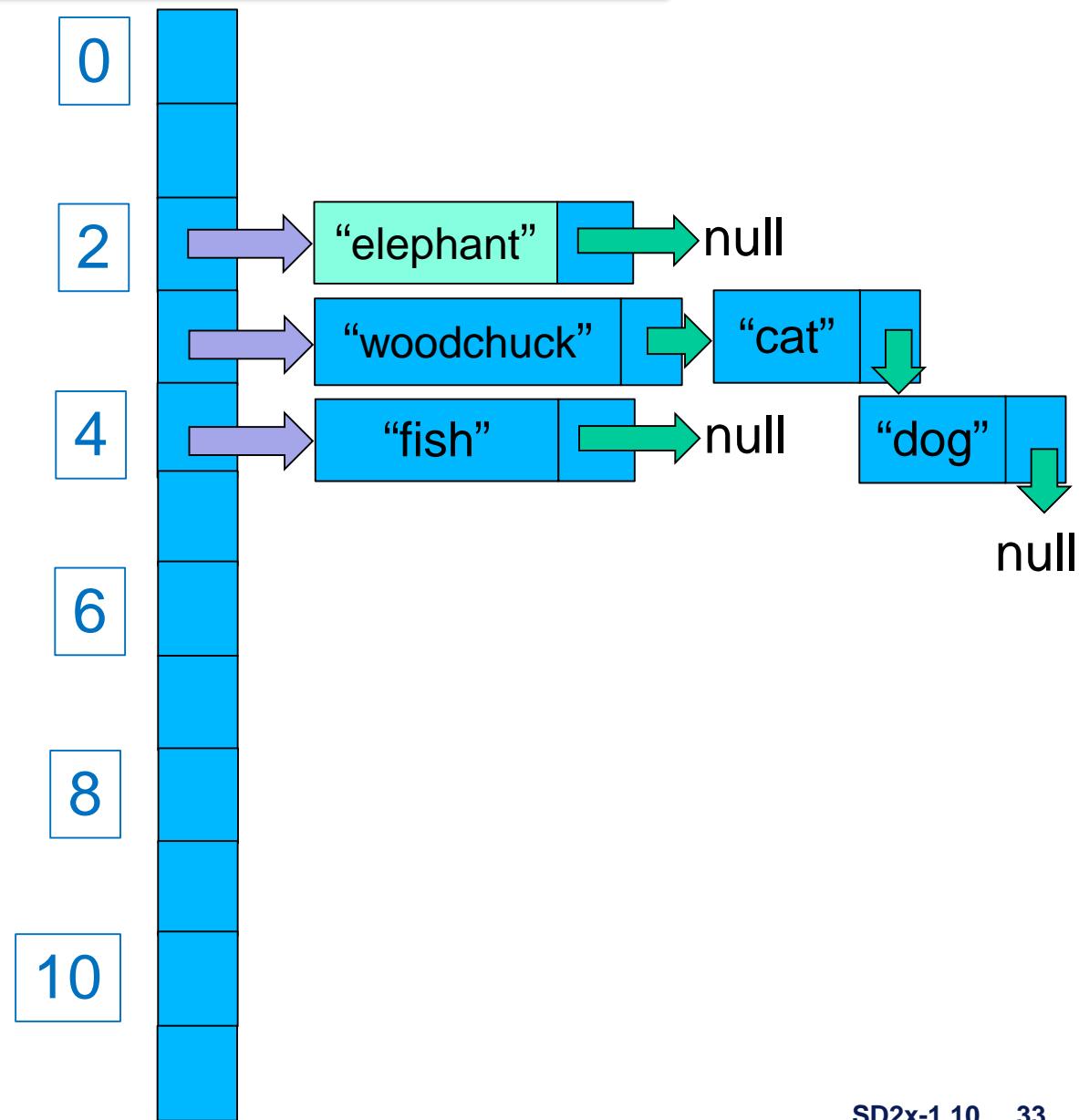
1. Double number of buckets



2. Re-insert existing values

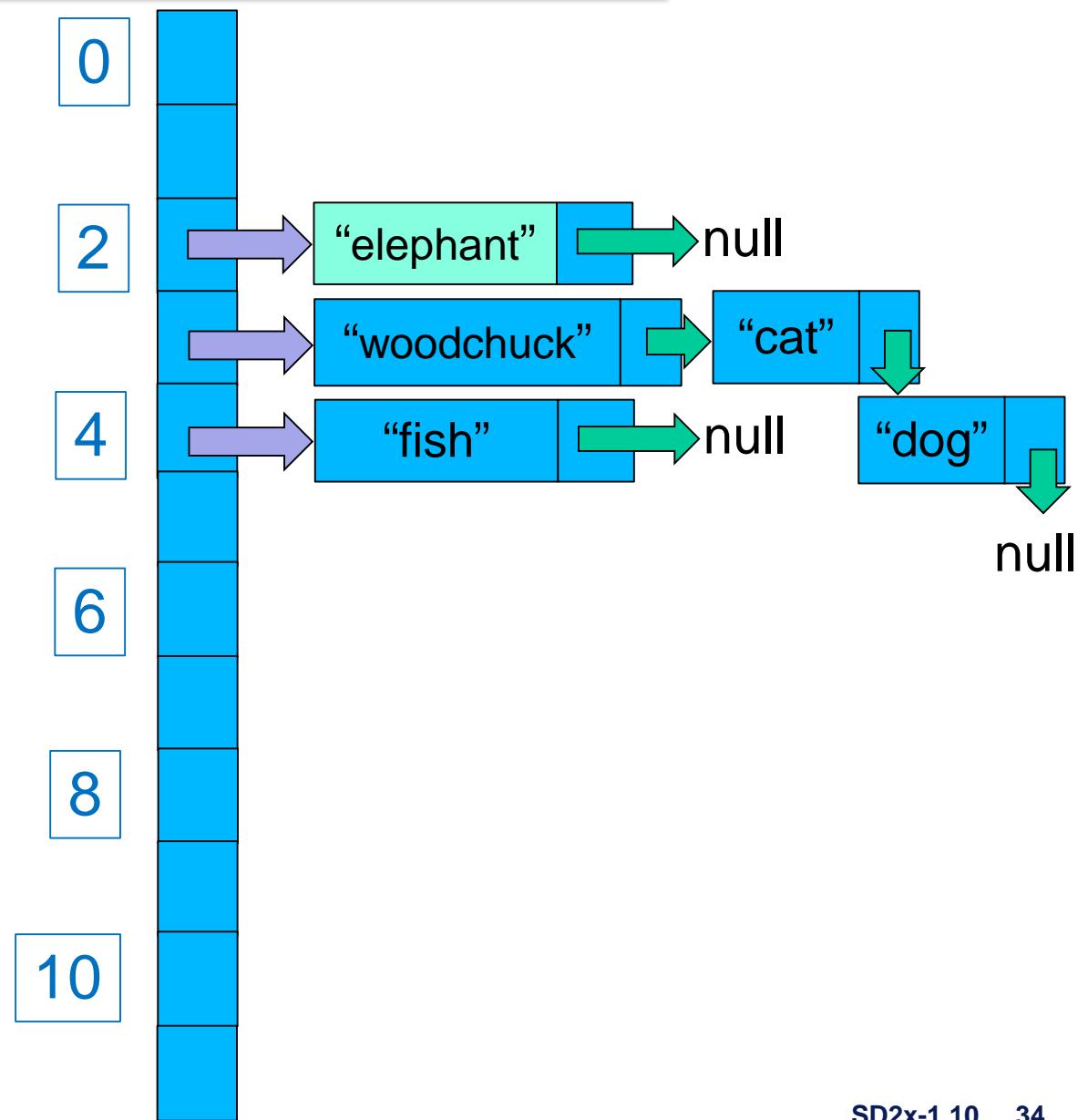


2. Re-insert existing values

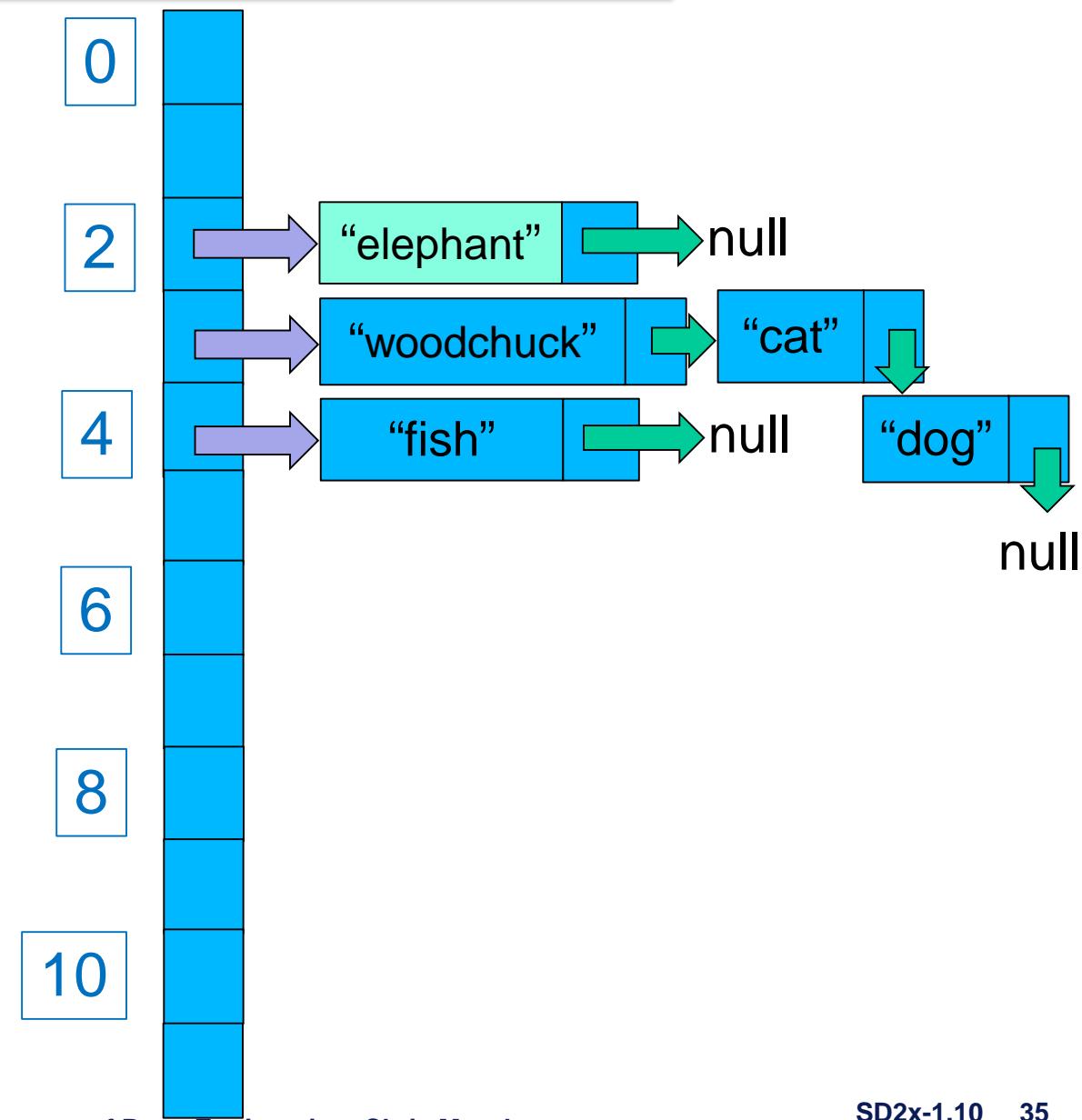
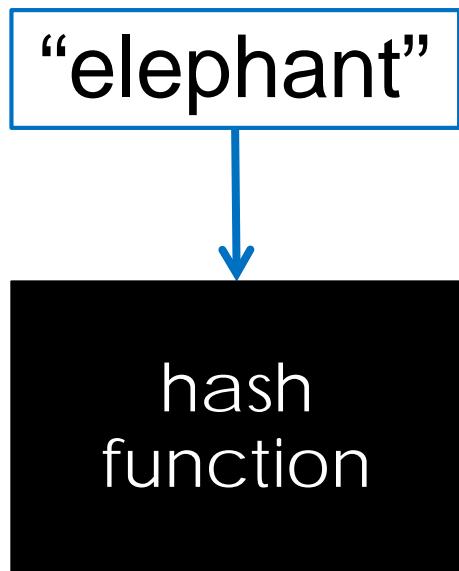


2. Re-insert existing values

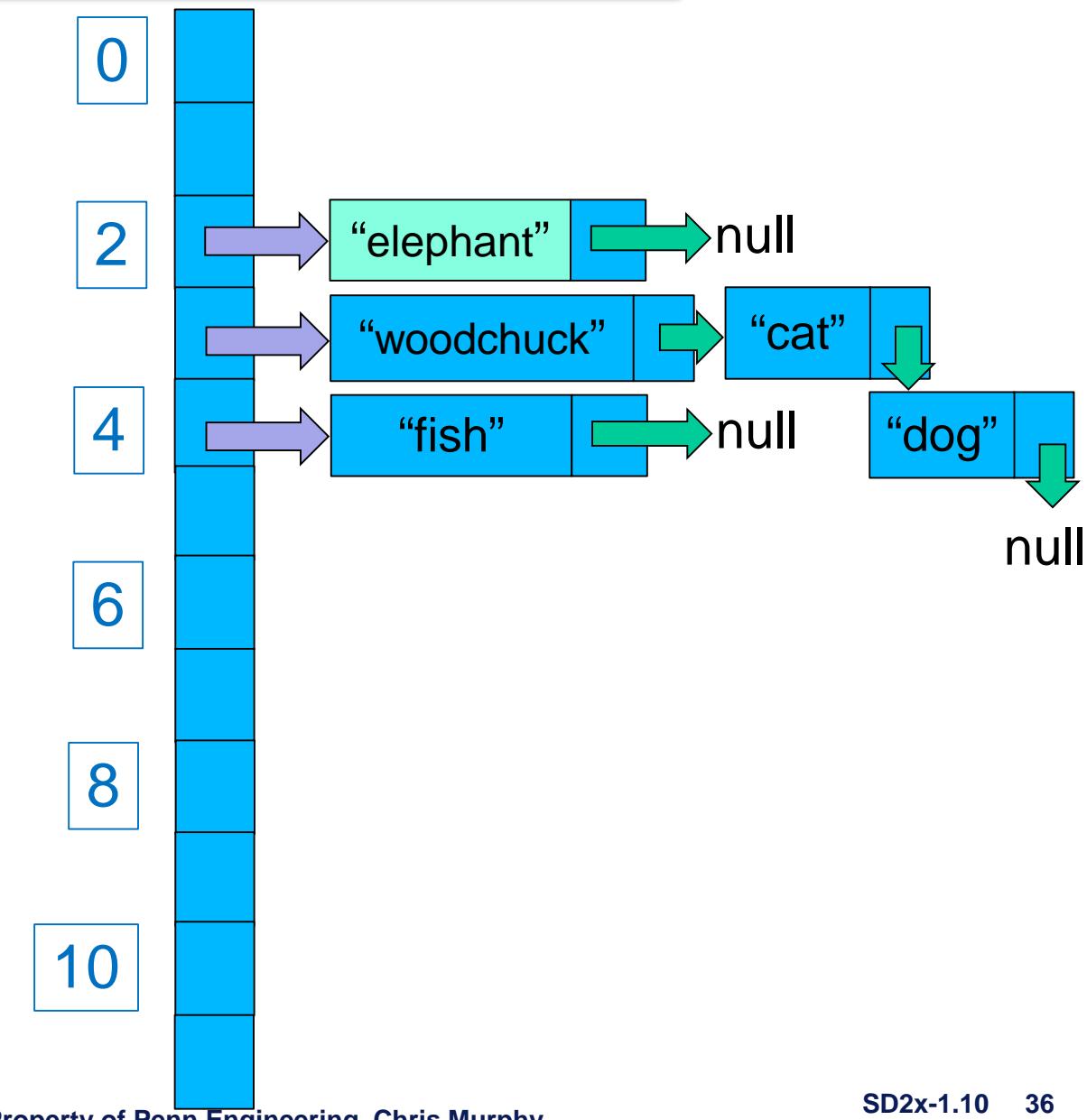
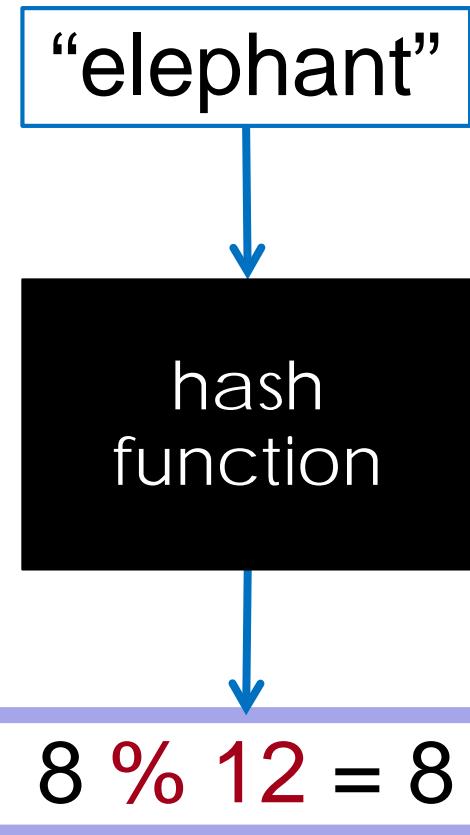
“elephant”



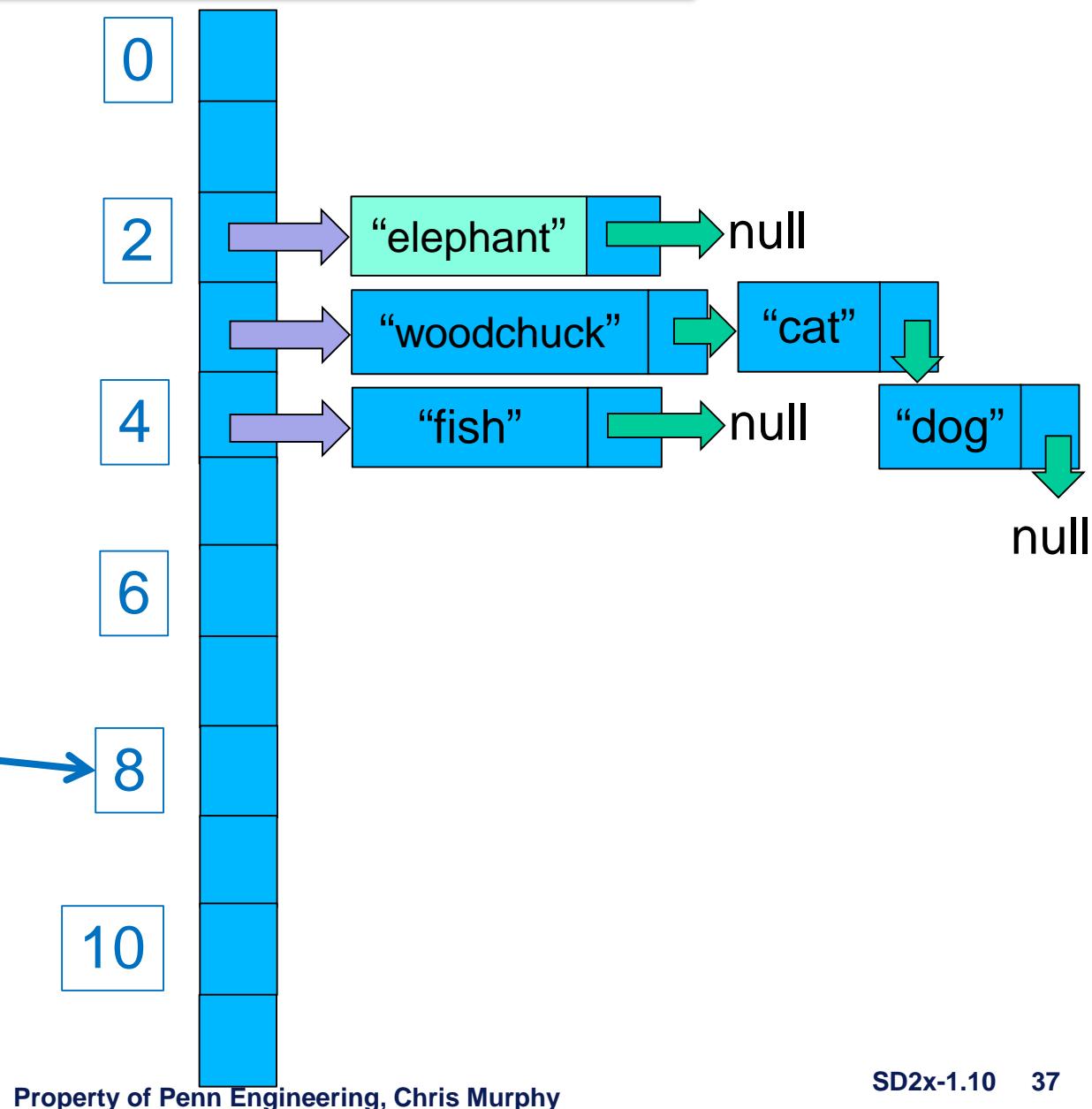
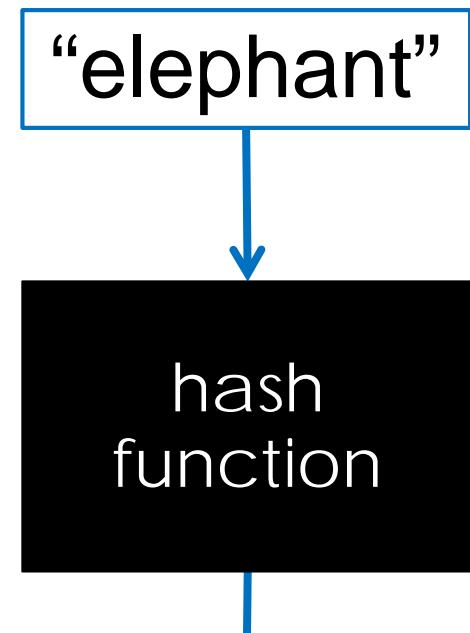
2. Re-insert existing values



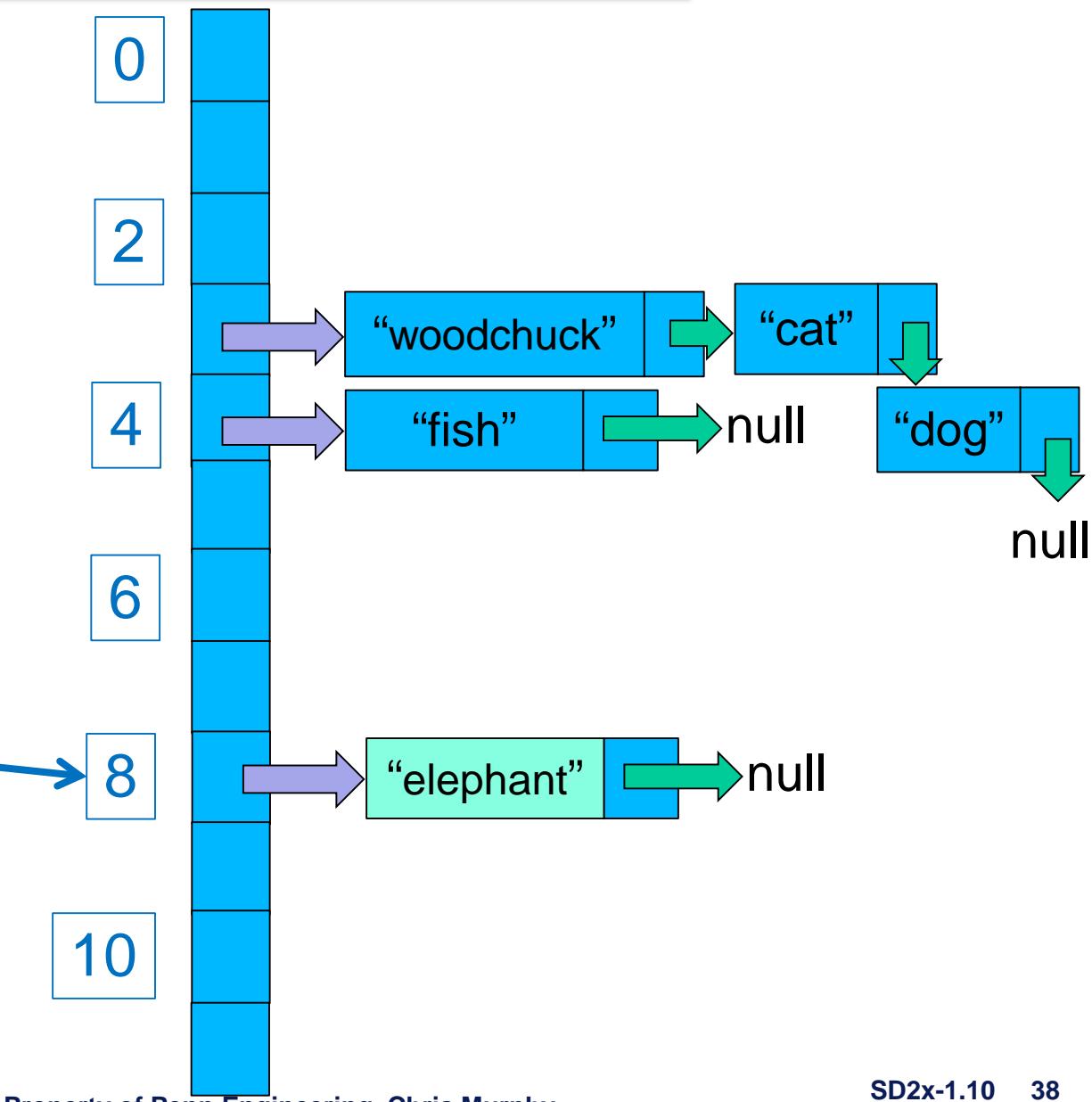
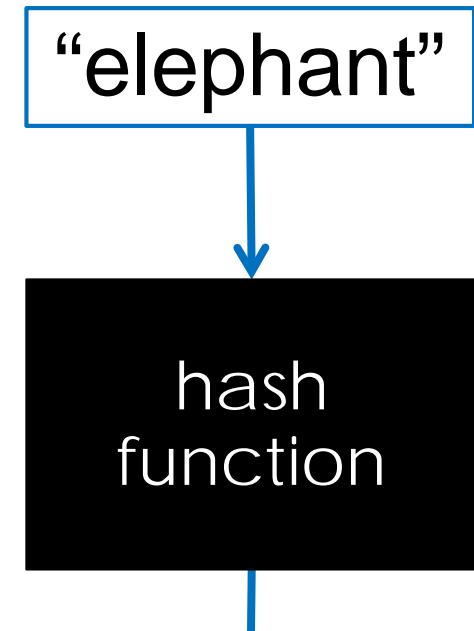
2. Re-insert existing values



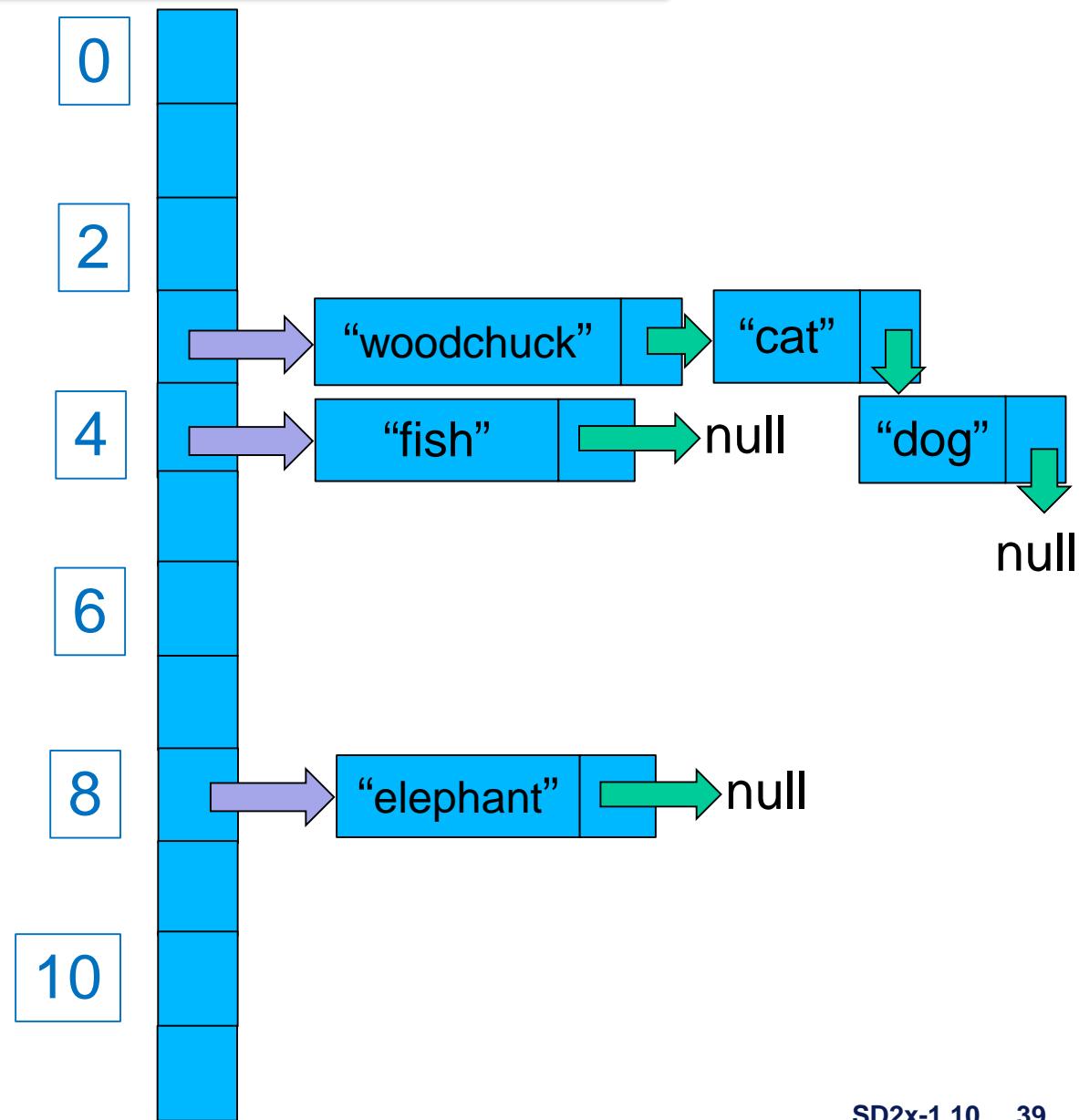
2. Re-insert existing values



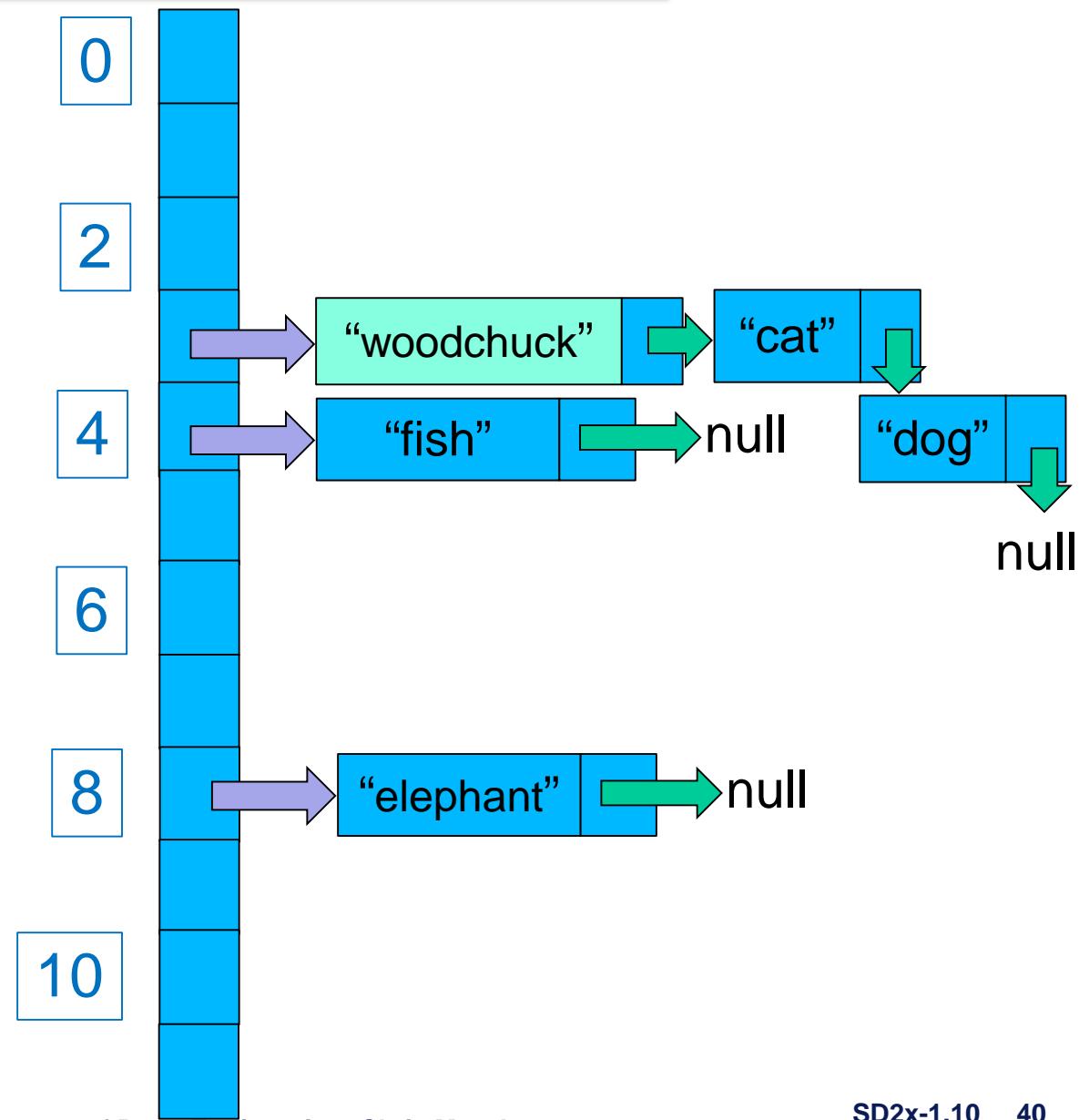
2. Re-insert existing values



2. Re-insert existing values

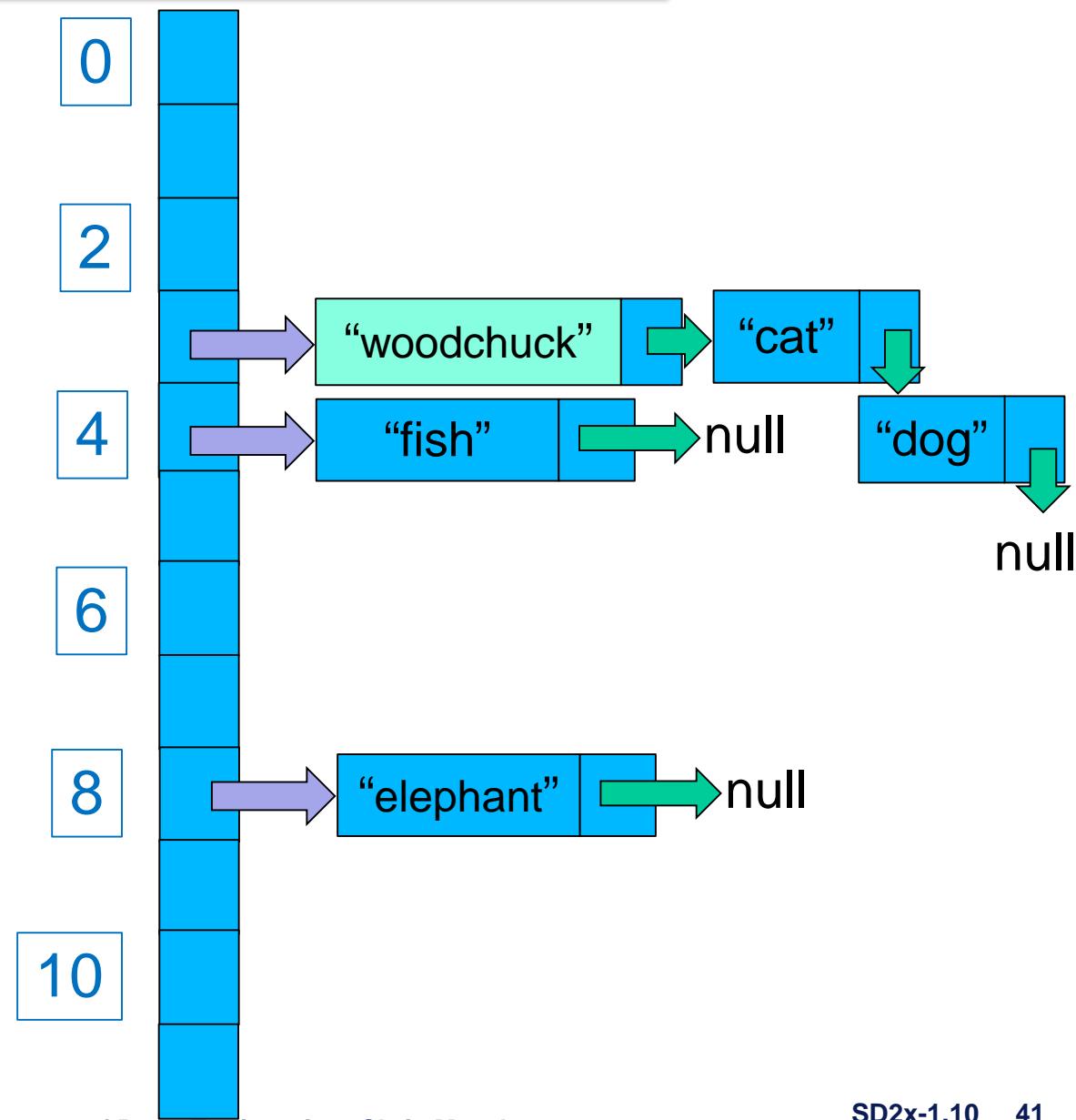


2. Re-insert existing values



2. Re-insert existing values

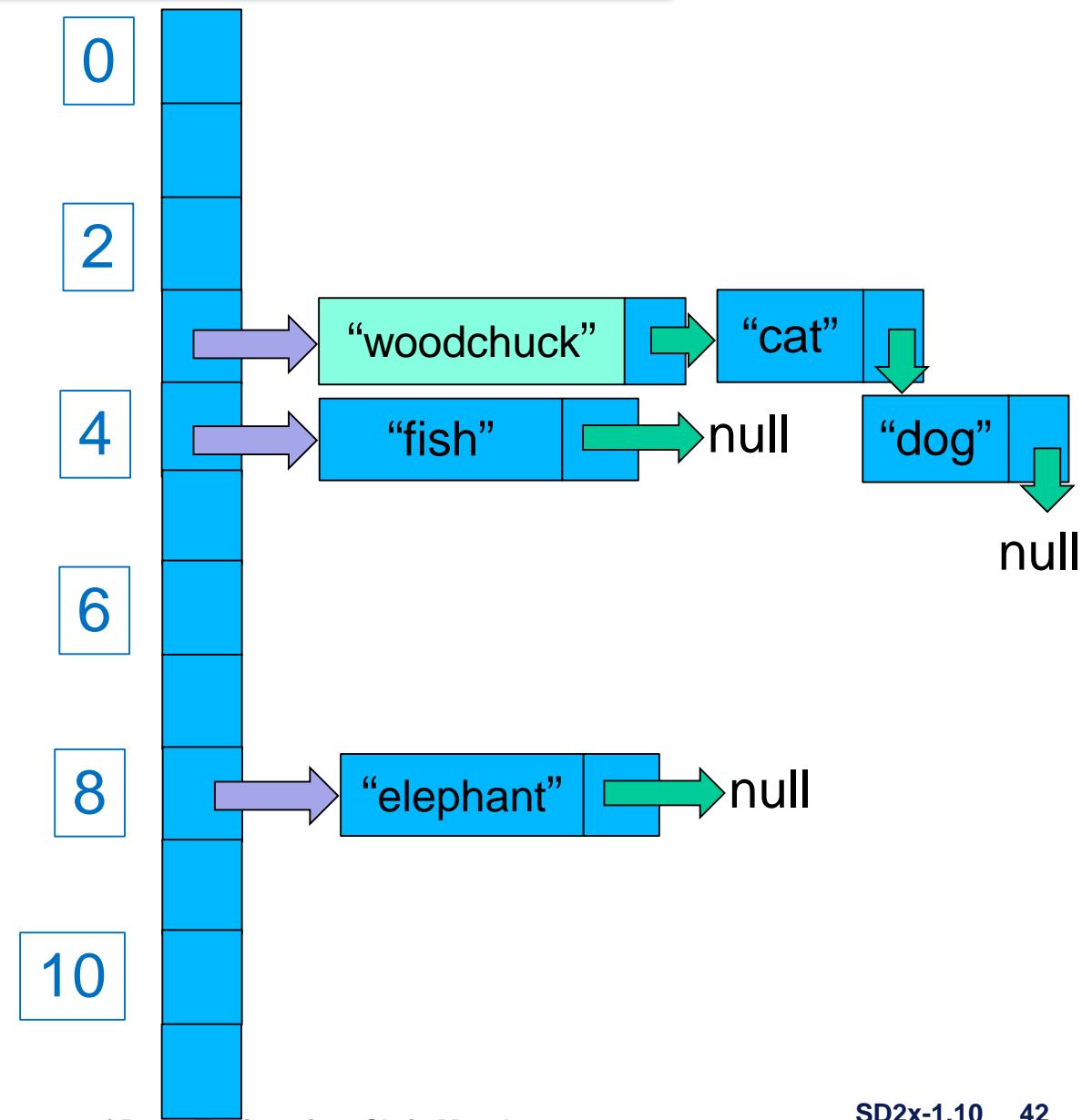
“woodchuck”



2. Re-insert existing values

“woodchuck”

hash
function

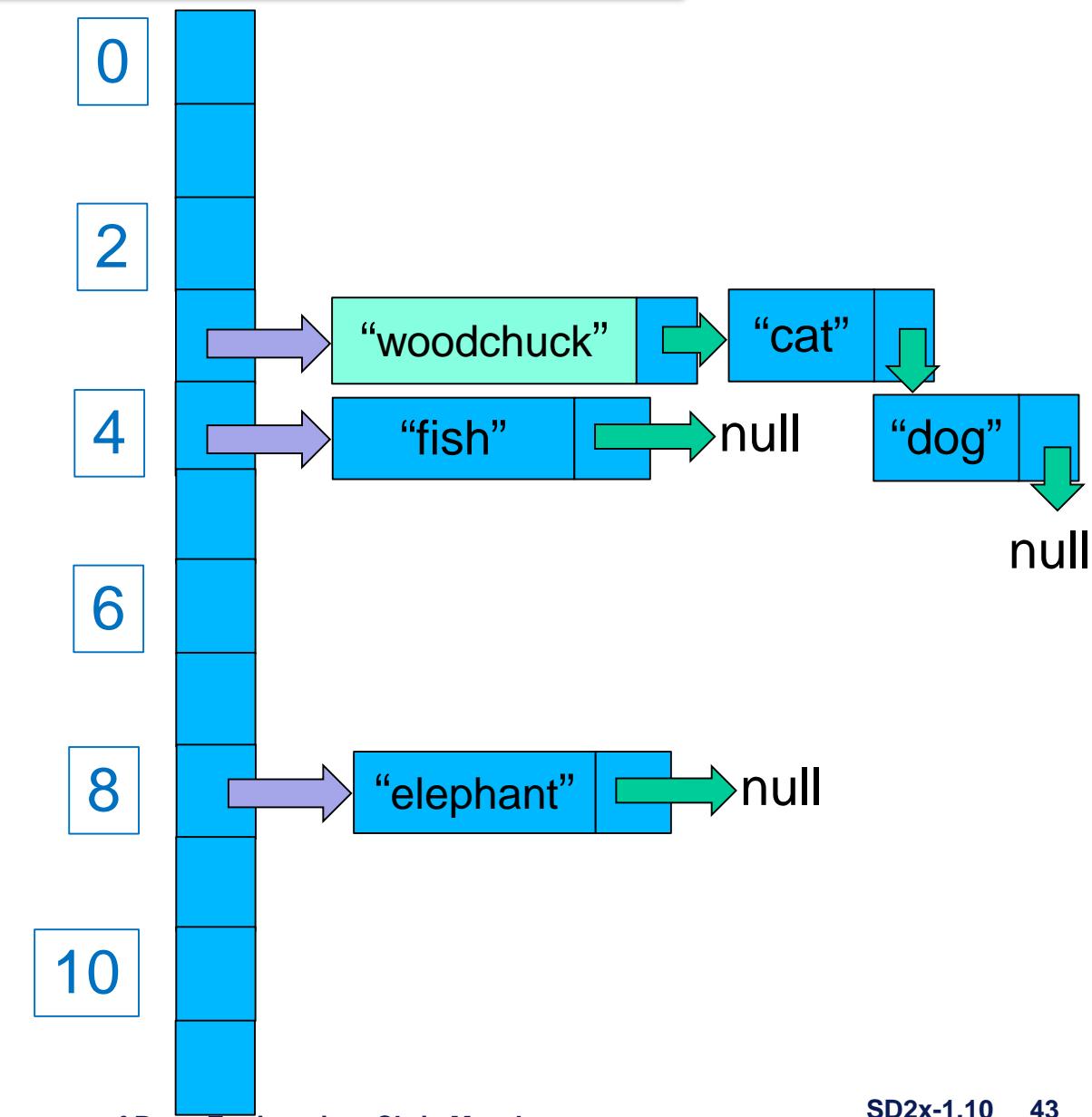


2. Re-insert existing values

“woodchuck”

hash
function

$$9 \% 12 = 9$$

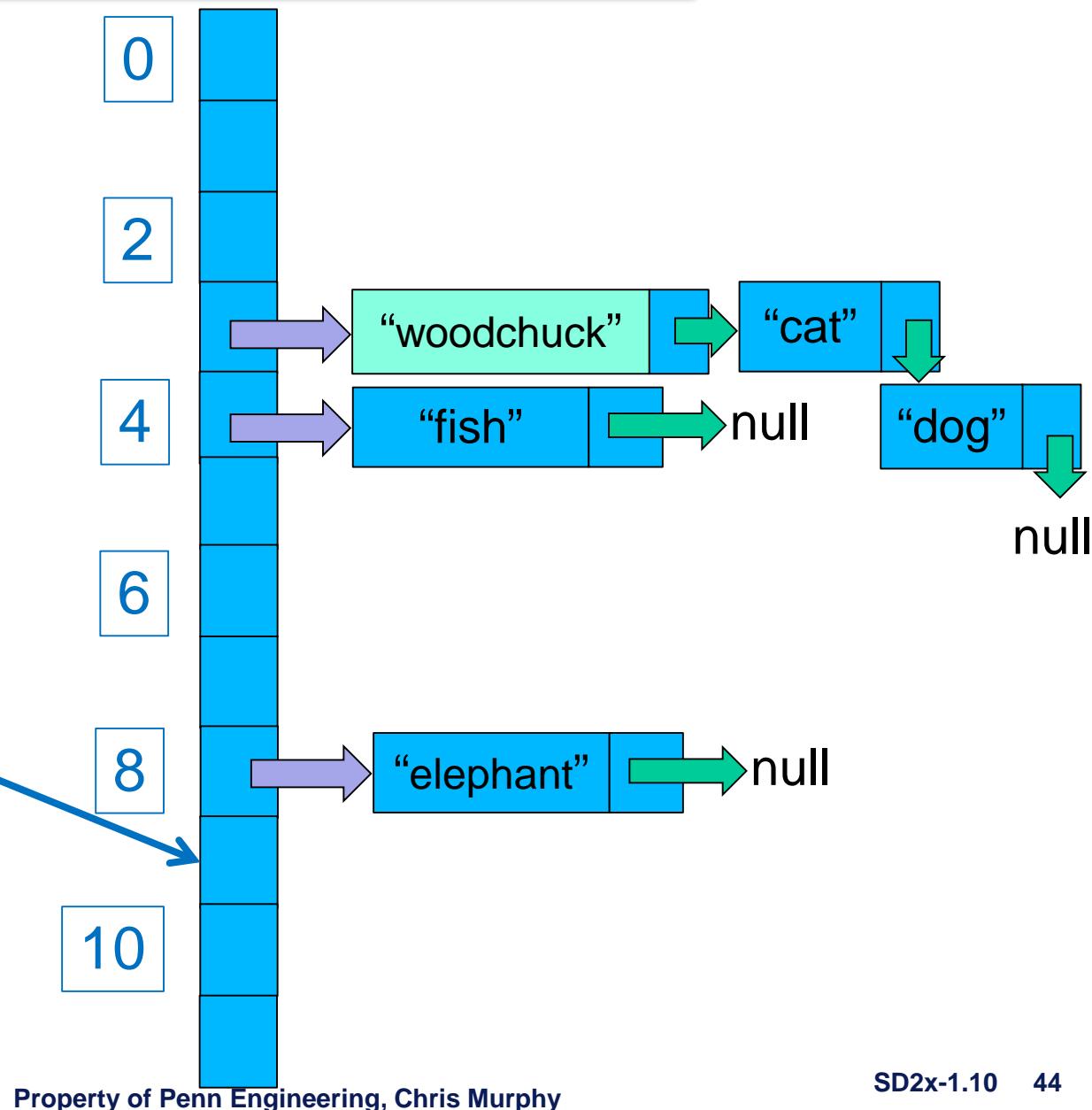


2. Re-insert existing values

“woodchuck”

hash
function

$$9 \% 12 = 9$$

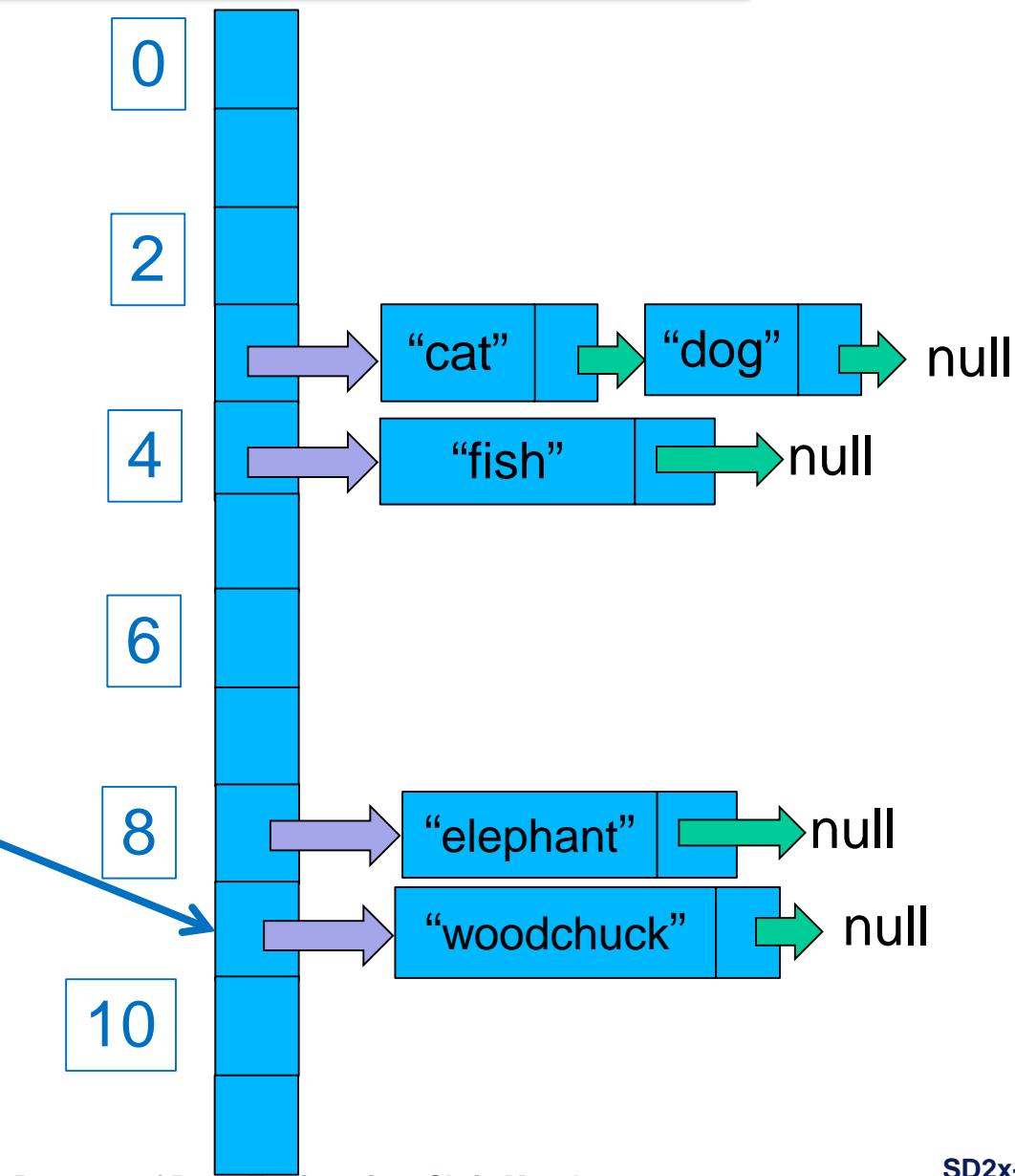


2. Re-insert existing values

“woodchuck”

hash
function

$$9 \% 12 = 9$$

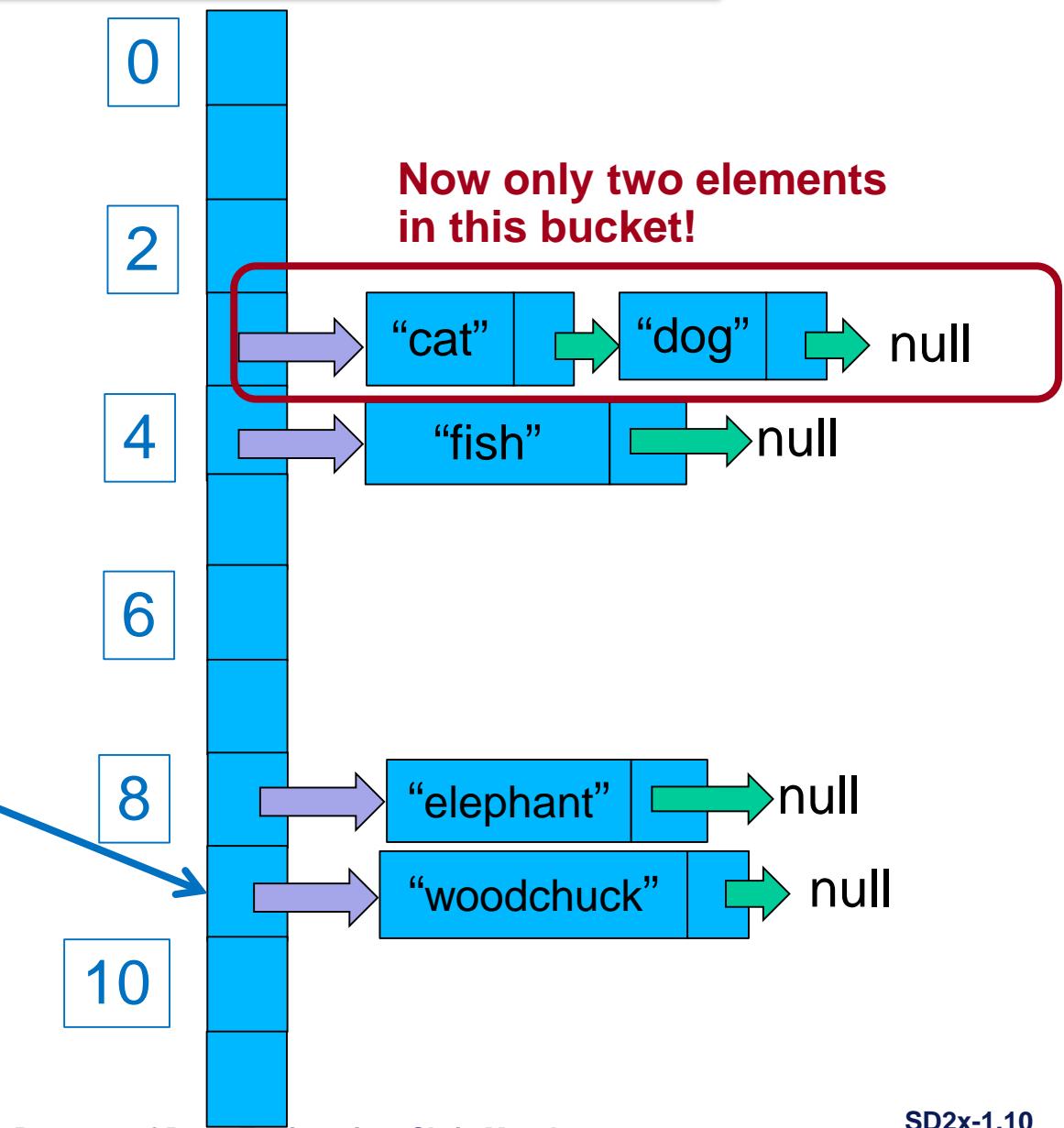


2. Re-insert existing values

“woodchuck”

hash
function

$$9 \% 12 = 9$$



Recall: HashSet without resizing

```
public class HashSet {  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
        }  
        return true;  
    }  
    return false;  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize/(double)buckets.length;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
        }  
        return true;  
    }  
    return false;  
}
```

HashSet with resizing

```
public class HashSet {  
  
    private int currentSize = 0;  
    private double loadFactor;  
  
    . . .  
  
    public boolean add(String value) {  
        if (!contains(value)) {  
            int index = hashCode(value) % buckets.length;  
            LinkedList<String> bucket = buckets[index];  
            bucket.addFirst(value);  
            currentSize++;  
  
            double averageLoad = currentSize / (double) numBuckets;  
            if (averageLoad > loadFactor) {  
                reinsertAll();  
            }  
            return true;  
        }  
        return false;  
    }  
}
```

Reinserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Reinserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
private LinkedList<String>[] buckets;  
  
private void reinsertAll() {  
    LinkedList<String> oldBuckets[] = buckets;  
    buckets = new LinkedList[buckets.length * 2];  
  
    for (LinkedList<String> bucket : oldBuckets) {  
        for (String element : bucket) {  
            int index = hashCode(element) % buckets.length;  
            LinkedList<String> newBucket = buckets[index];  
            newBucket.addFirst(element);  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Re-inserting

```
public class HashSet {  
    . . .  
  
    private LinkedList<String>[ ] buckets;  
  
    private void reinsertAll() {  
        LinkedList<String> oldBuckets[ ] = buckets;  
        buckets = new LinkedList[buckets.length * 2];  
  
        for (LinkedList<String> bucket : oldBuckets) {  
            for (String element : bucket) {  
                int index = hashCode(element) % buckets.length;  
                LinkedList<String> newBucket = buckets[index];  
                newBucket.addFirst(element);  
            }  
        }  
    }  
}
```

Recap: HashSets

- Elements are stored in arrays of Linked Lists, or “buckets”
- If the buckets get too full, we lose the ability to quickly find elements
- The HashSet is able to resize itself so that elements in some buckets may move to others, making it faster to find elements

SD2x1.11

HashSets and Maps in Java API

Chris

How are HashSets implemented in the Java API?

HashSets in Java API

- **java.util.HashSet<E>**
- **add:** adds element to set if not already present
- **contains:** indicates whether set contains element
- **remove:** removes from set element if present
- **clear:** removes all elements
- **isEmpty:** indicates whether set contains no elements
- **size:** returns number of elements in set

```
HashSet<String> words = new HashSet<String>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a word: ");
    input = in.nextLine();
    if (input.equals("q")) {
        break;
    }
    if (words.add(input))
        System.out.println("Added " + input + " to the set");
    else
        System.out.println(input + " is already in the set");
}

System.out.print("Enter another word: ");
input = in.nextLine();
if (words.contains(input))
    System.out.println(input + " is in the set");
else
    System.out.println(input + " is not in the set");
```

```
HashSet<String> words = new HashSet<String>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a word: ");
    input = in.nextLine();
    if (input.equals("q")) {
        break;
    }
    if (words.add(input))
        System.out.println("Added " + input + " to the set");
    else
        System.out.println(input + " is already in the set");
}

System.out.print("Enter another word: ");
input = in.nextLine();
if (words.contains(input))
    System.out.println(input + " is in the set");
else
    System.out.println(input + " is not in the set");
```

```
HashSet<String> words = new HashSet<String>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a word: ");
    input = in.nextLine();
    if (input.equals("q")) {
        break;
    }
    if (words.add(input))
        System.out.println("Added " + input + " to the set");
    else
        System.out.println(input + " is already in the set");
}

System.out.print("Enter another word: ");
input = in.nextLine();
if (words.contains(input))
    System.out.println(input + " is in the set");
else
    System.out.println(input + " is not in the set");
```

```
HashSet<String> words = new HashSet<String>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a word: ");
    input = in.nextLine();
    if (input.equals("q")) {
        break;
    }
    if (words.add(input))
        System.out.println("Added " + input + " to the set");
    else
        System.out.println(input + " is already in the set");
}

System.out.print("Enter another word: ");
input = in.nextLine();
if (words.contains(input))
    System.out.println(input + " is in the set");
else
    System.out.println(input + " is not in the set");
```

Using Sets

- Let's say we're writing a program that reads a file and keeps track of the number of times various Strings appear in the file
- We can have a collection (HashSet) of Strings, but we also need to have some associated values

“dog”

3

“cat”

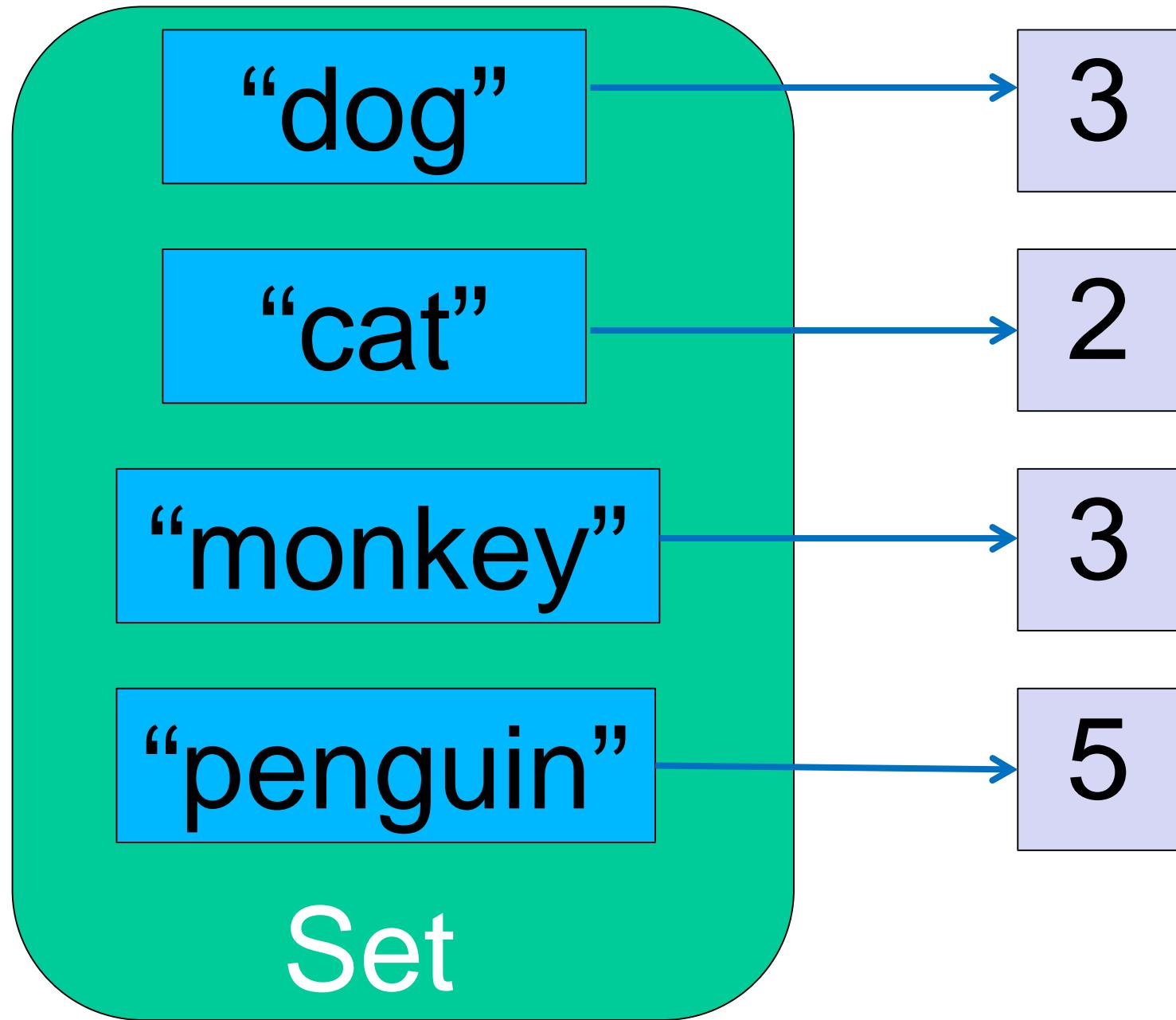
2

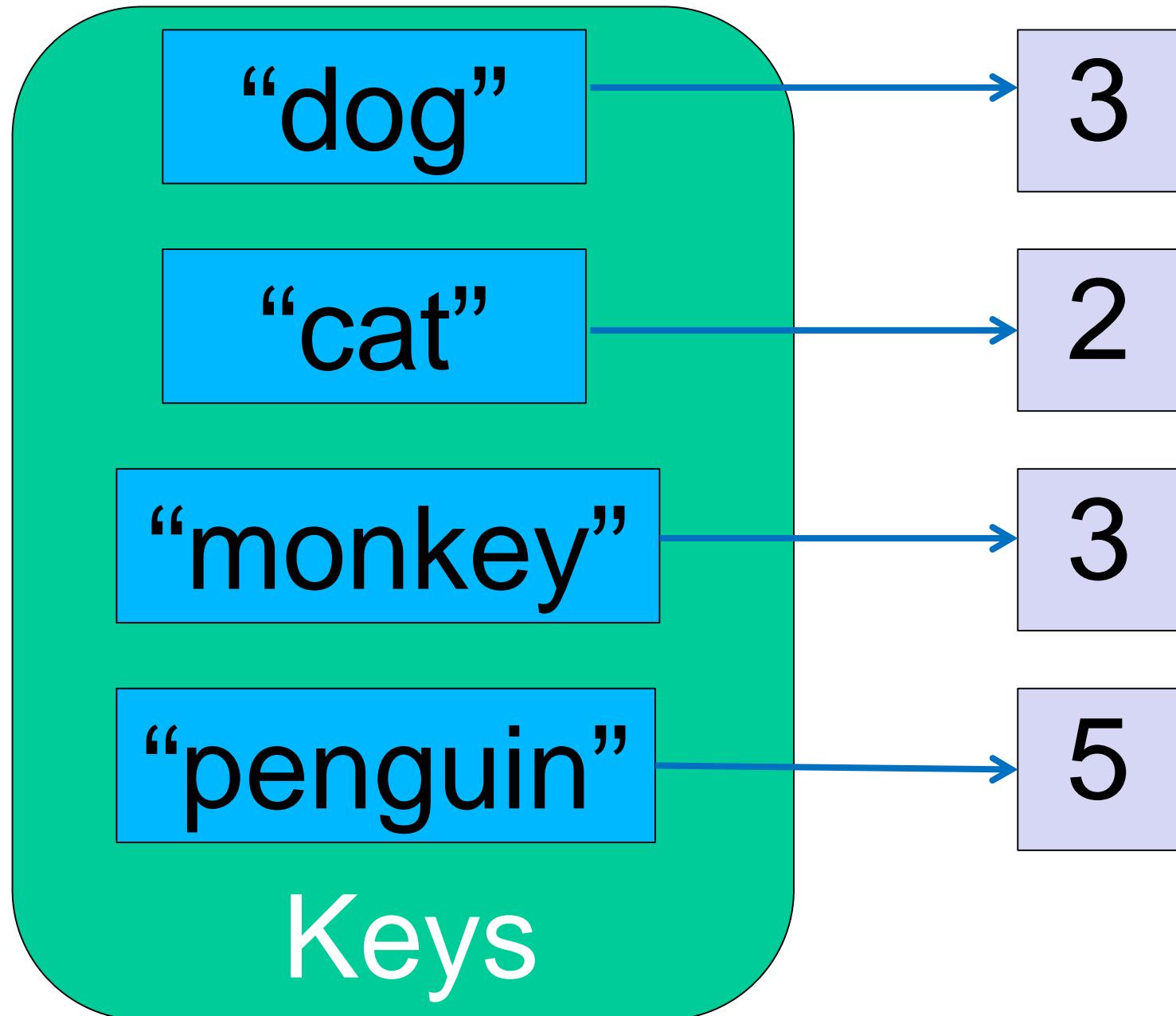
“monkey”

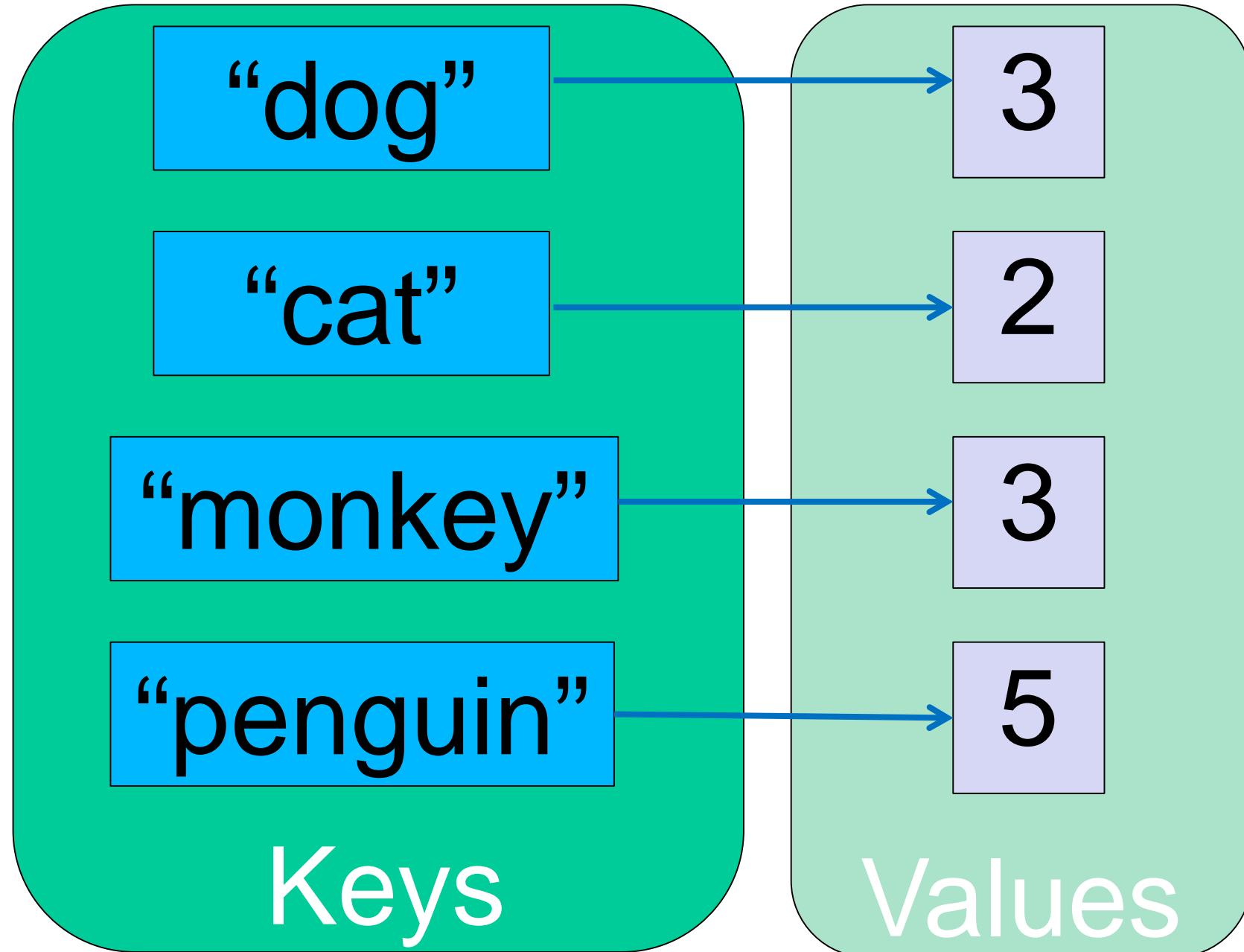
3

“penguin”

5







Maps in Java API

- A mapping of (a Set of) **Keys** to **Values**
- **java.util.Map<K,V>** interface
- **put(K, V)**: associates value V with key K
- **get(K)**: returns value for key K
- **containsKey(K)**: indicates whether there is a mapping for key K
- **containsValue(V)**: indicates whether there is any key that maps to value V
- Implementations include **java.util.HashMap**

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

```
HashMap<String, Integer> wordCount =
    new HashMap<String, Integer>();

Scanner in = new Scanner(System.in);
String input = null;

while (true) {
    System.out.print("Enter a string: " );
    input = in.nextLine();
    if (input.equals("q"))
        break;
    if (wordCount.containsKey(input)) {
        int count = wordCount.get(input);
        wordCount.put(input, count+1);
    }
    else wordCount.put(input, 1);
}

System.out.print("Enter another string: " );
input = in.nextLine();
if (wordCount.containsKey(input)) {
    int count = wordCount.get(input);
    System.out.println(input + " appears " + count + " times");
}
else System.out.println(input + " is not in the map");
```

How would we print all key/value pairs in a Map?

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}
```

EntrySets in Java API

- **java.util.Map.Entry<K,V>**
- Object that represents a single key/value pair (“entry”) in a Map
- You can access the Set of Entries using the Map’s **entrySet** method

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}

// EntrySet way of iterating over all elements
for (Map.Entry<String, Integer> entry : wordCount.entrySet())
{
    String word = entry.getKey();
    int count = entry.getValue();
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}

// EntrySet way of iterating over all elements
for (Map.Entry<String, Integer> entry : wordCount.entrySet())
{
    String word = entry.getKey();
    int count = entry.getValue();
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}

// EntrySet way of iterating over all elements
for (Map.Entry<String, Integer> entry : wordCount.entrySet())
{
    String word = entry.getKey();
    int count = entry.getValue();
    System.out.println(word + " appears " + count + " times");
}
```

```
// naive way of iterating over all elements
Set<String> keys = wordCount.keySet();
for (String word : keys) {
    int count = wordCount.get(word);
    System.out.println(word + " appears " + count + " times");
}

// EntrySet way of iterating over all elements
for (Map.Entry<String, Integer> entry : wordCount.entrySet())
{
    String word = entry.getKey();
    int count = entry.getValue();
    System.out.println(word + " appears " + count + " times");
}
```

Recap: HashSets and HashMaps

- Java API HashSet is a self-resizing implementation
- HashMap: HashSet in which each element in the set (“key”) is mapped to a corresponding “value”
- EntrySet: Set of key/value pairs that simplifies iteration over elements in the map