Implementing a Linked List

If you had to give a TED Talk/design a course about one of your interests, what would you choose?

When we implement a linked list, we include two fields in the Class. What are those two fields

Node tail

Node head

int size

String[] values

I have a List l with 3 elements in it (e.g. <4, 9, 1>). Which of the following would raise an exception?

- l.insert(0, 10)
- l.insert(3, 22)
- l.insert(2, 23)
- l.insert(1, 34)
- l.insert(4, 23)

"I would like to do small group activities during lecture on Friday"

True

False

Insert

Two Cases to Handle

Insert at the head of the list

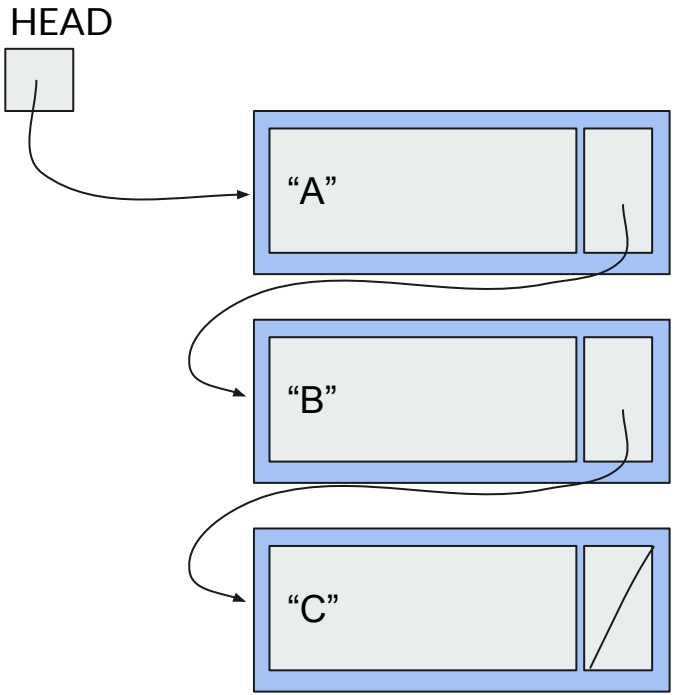
- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

Insert anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Create a new node with the specified data, call this node *new*
- Set *new.next* to be *current.next*
- Set *current.next* to be *new*

Inserting at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

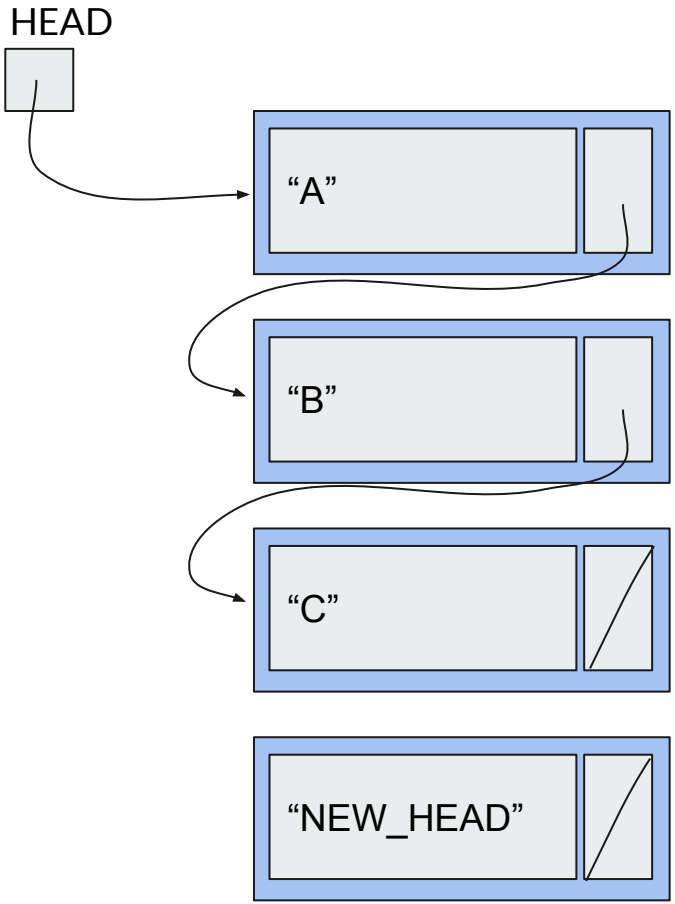



```
l.insert(0, “NEW_HEAD”)
```

Inserting at the head

- **Make a new Node with the specified data**
- Set the next field of the new node to be head.
- Set head to be the new Node

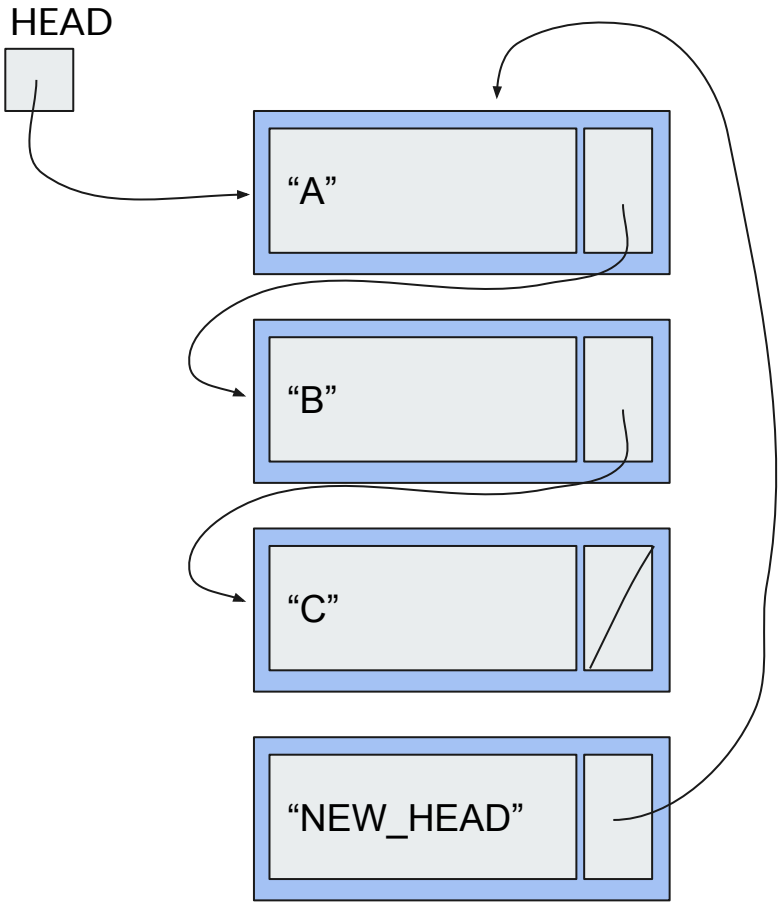
```
l.insert(0, “NEW_HEAD”)
```



Inserting at the head

- Make a new Node with the specified data
- **Set the next field of the new node to be head.**
- Set head to be the new Node

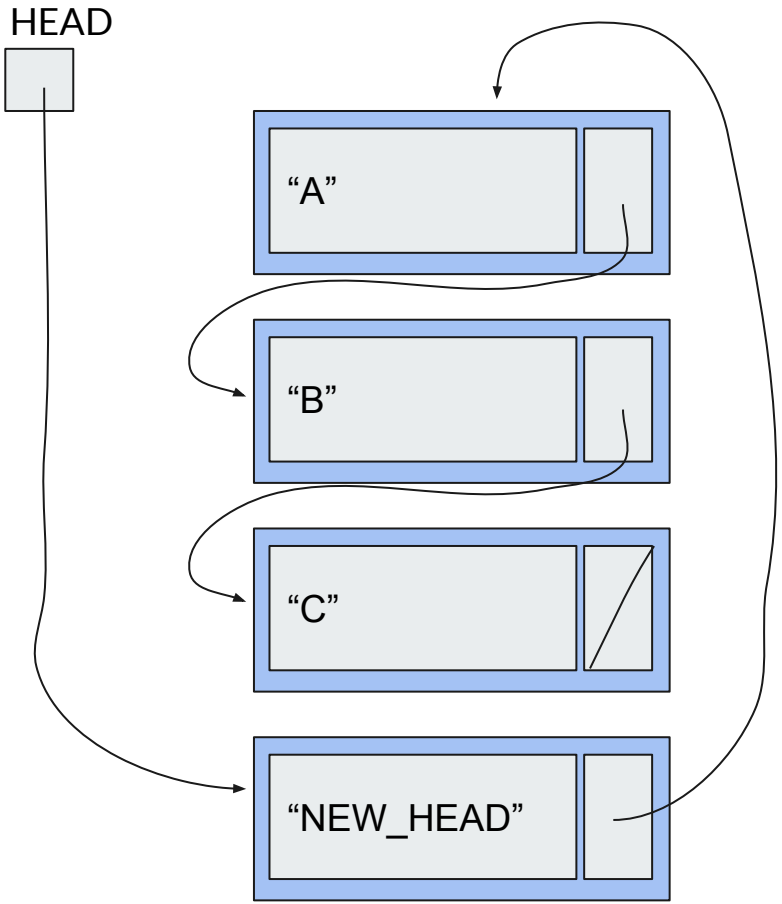
```
l.insert(0, “NEW_HEAD”)
```



Inserting at the head

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- **Set head to be the new Node**

```
l.insert(0, “NEW_HEAD”)
```

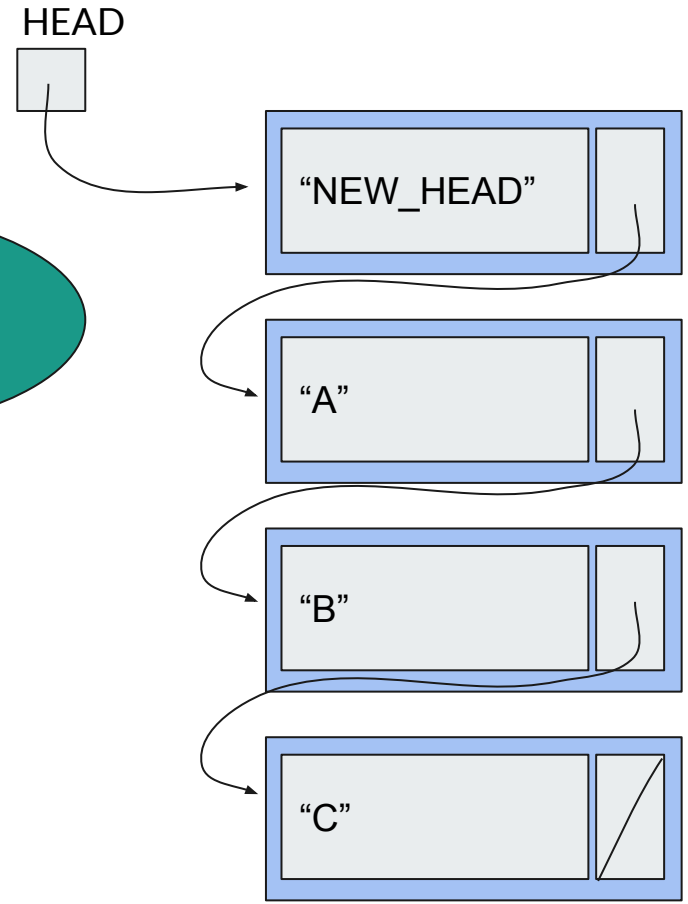


Inserting at the head

Rearranging for style, no change actually made to the list!

- Make a new Node with the specified data
- Set the next field of the new node to be head.
- Set head to be the new Node

```
l.insert(0, “NEW_HEAD”)
```

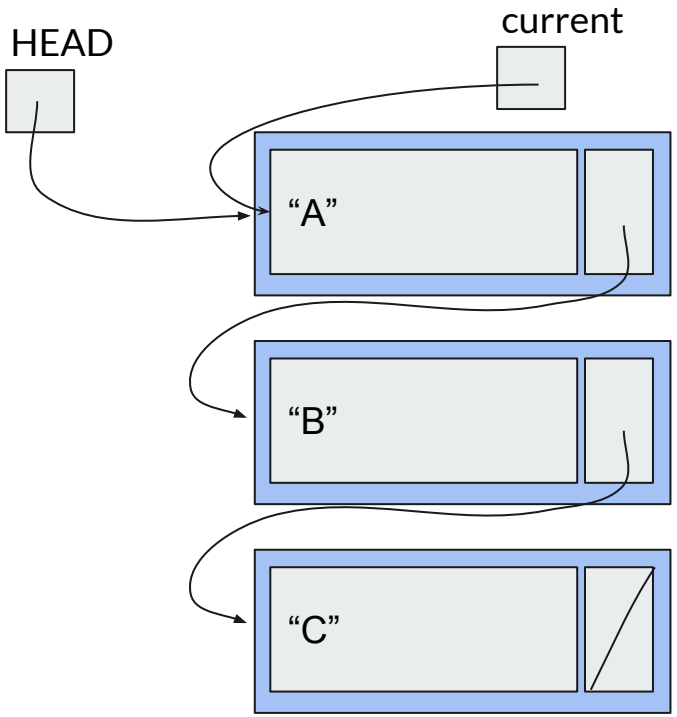


Inserting within the List

Insert anywhere else

- **Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current***
- Create a new node with the specified data, call this node *new*
- Set *new.next* to be *current.next*
- Set *current.next* to be *new*

```
l.insert(2, "NEW_NODE")
```

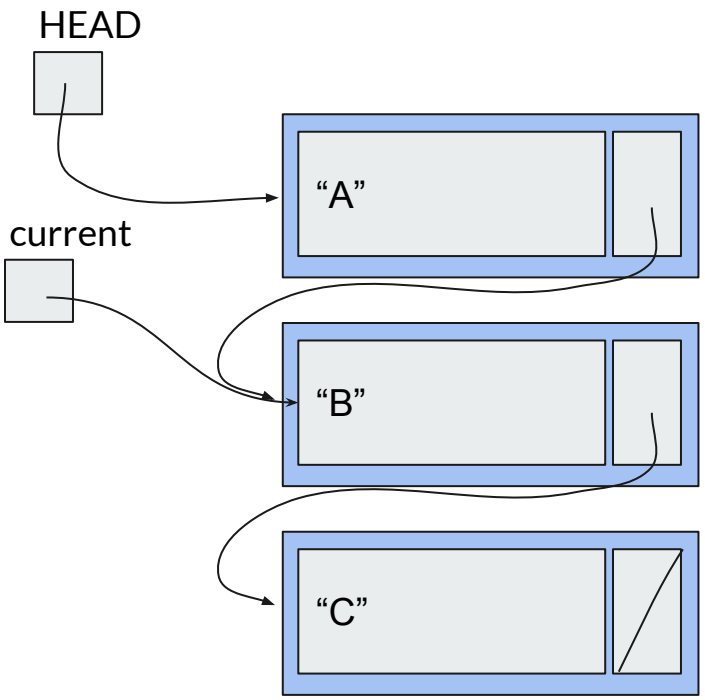


Inserting within the List

Insert anywhere else

- **Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current***
- Create a new node with the specified data, call this node *new*
- Set *new.next* to be *current.next*
- Set *current.next* to be *new*

```
l.insert(2, "NEW_NODE")
```

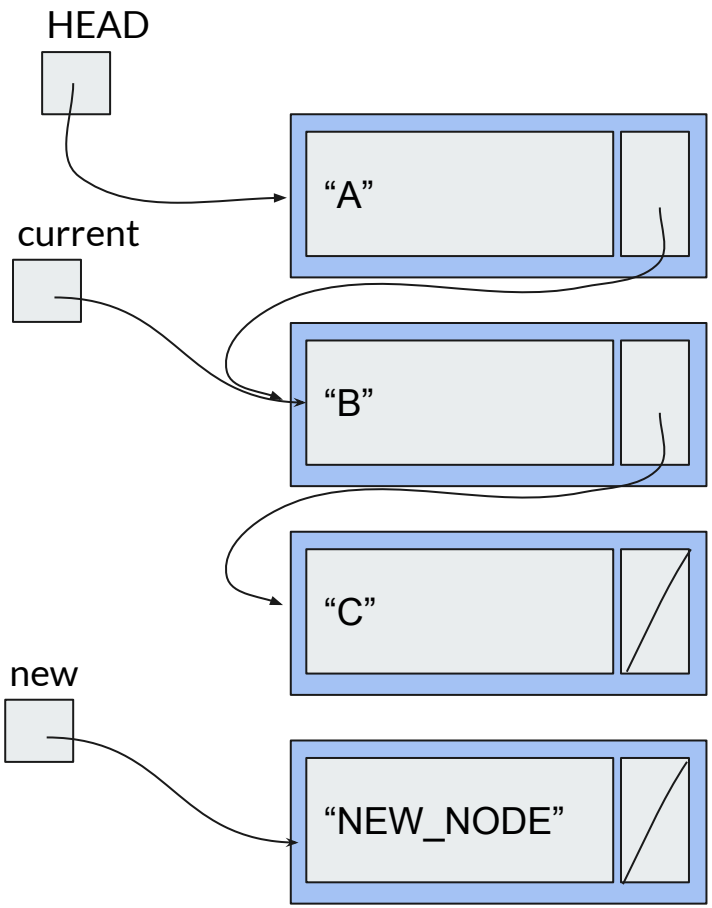


Inserting within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- **Create a new node with the specified data, call this node *new***
- Set *new.next* to be *current.next*
- Set *current.next* to be *new*

```
l.insert(2, "NEW_NODE")
```

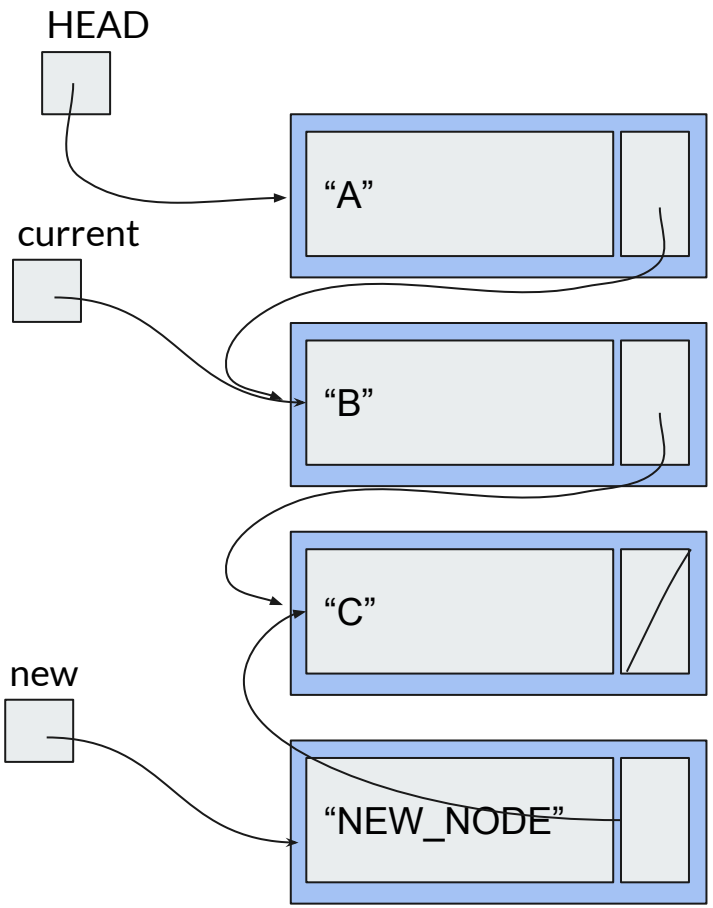


Inserting within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Create a new node with the specified data, call this node *new*
- **Set *new.next* to be *current.next***
- Set *current.next* to be *new*

```
l.insert(2, "NEW_NODE")
```

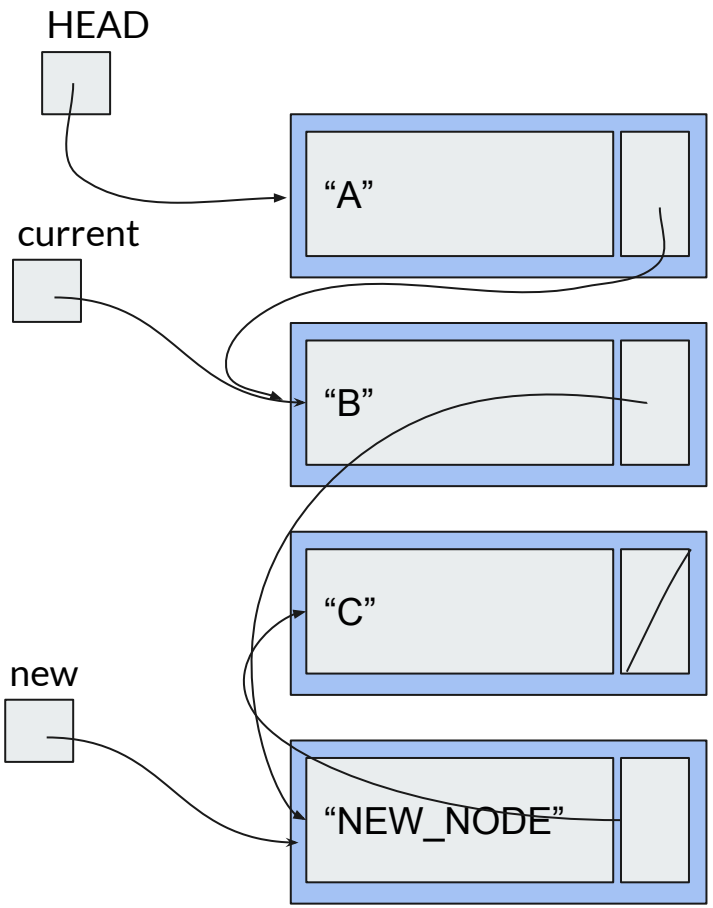


Inserting within the List

Insert anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Create a new node with the specified data, call this node *new*
- Set *new.next* to be *current.next*
- **Set *current.next* to be *new***

```
l.insert(2, "NEW_NODE")
```



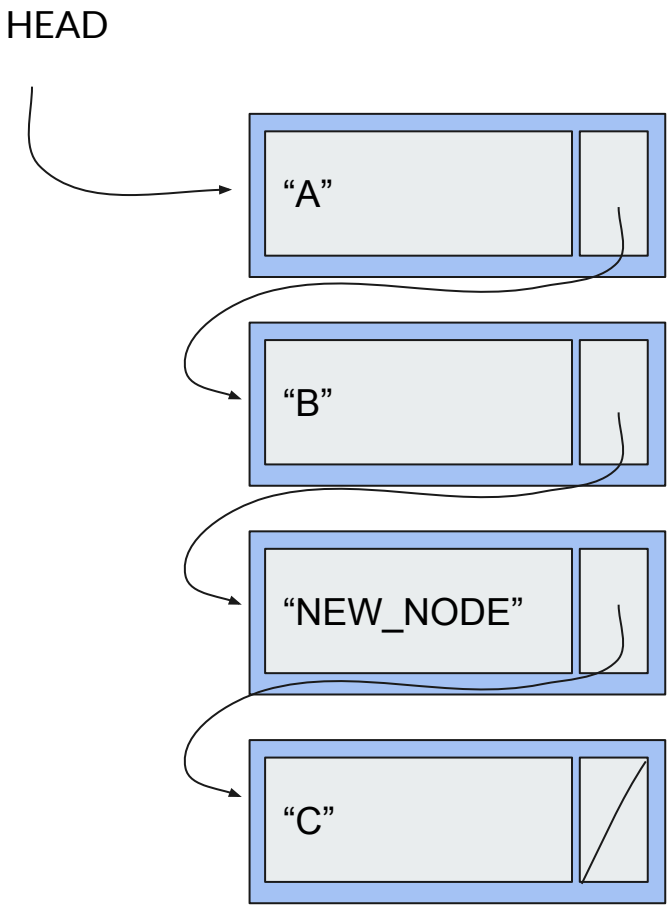
Inserting within the List

Rearranging for style, no change actually made to the list!

Insert anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Create a new node with the specified data, call this node *new*
- Set *new.next* to be *current.next*
- **Set *current.next* to be *new***

```
l.insert(2, "NEW_NODE")
```



Let’s do it!

Append

Two Cases to Handle

Append to empty list

Append to non-empty list

Two Cases to Handle

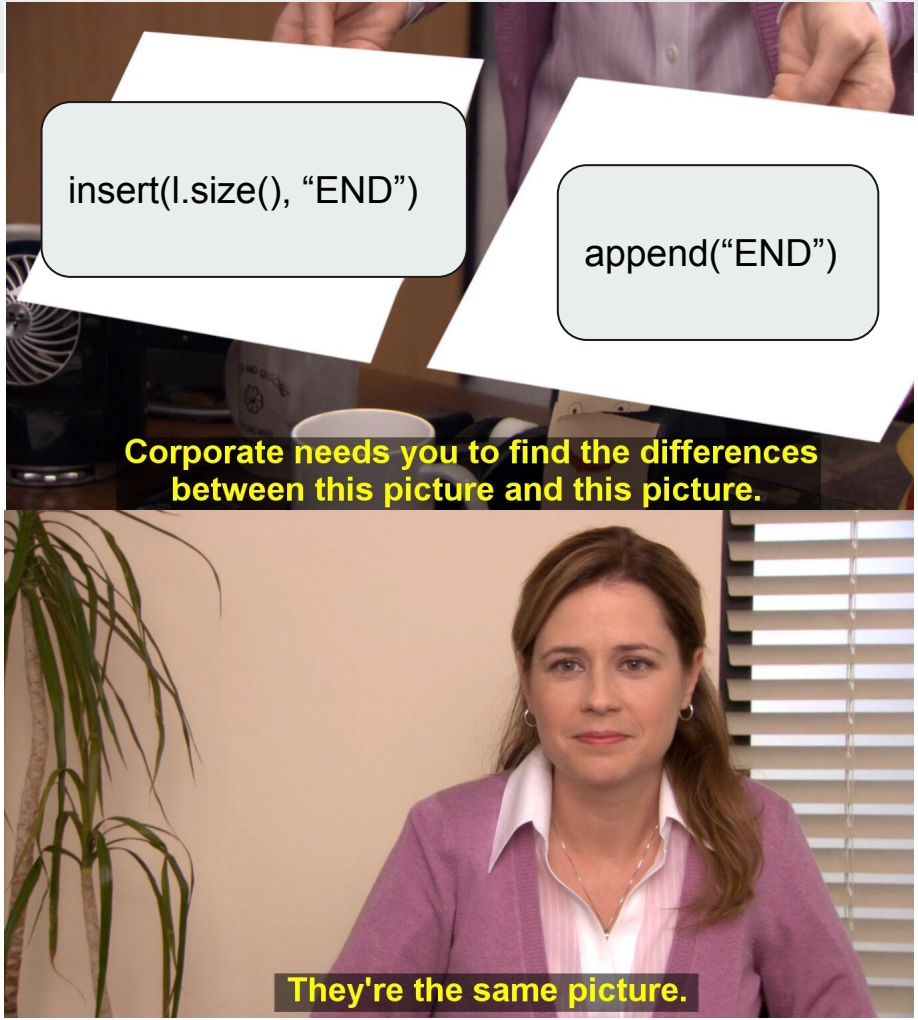
Append to empty list

Append to non-empty list

Technically true, but why repeat ourselves?

We already wrote insert, which can insert at the end of the list!

Let’s just use that instead.



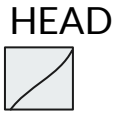
```
public boolean append(String it) {
  return this.insert(this.size, it);
}
```

Get

Two cases to handle:

1. List is empty
2. List non-empty

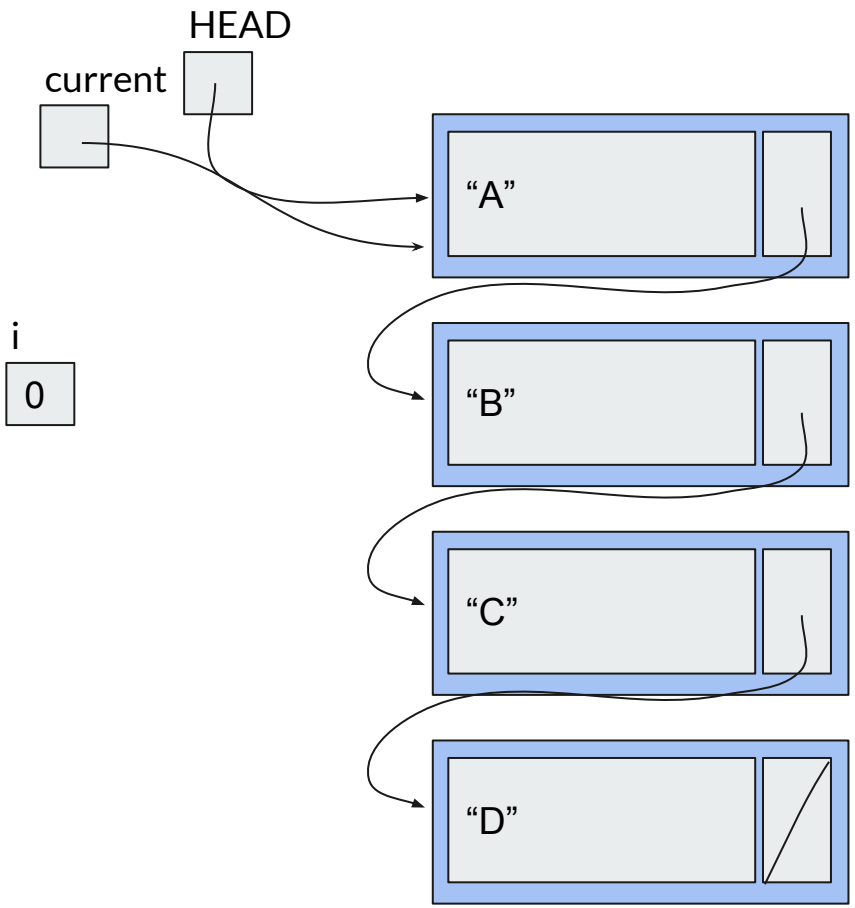
Two cases to handle:



1. **List is empty**
    a. **Just return default value of “” (empty String)**
2. List non-empty

Two cases to handle:

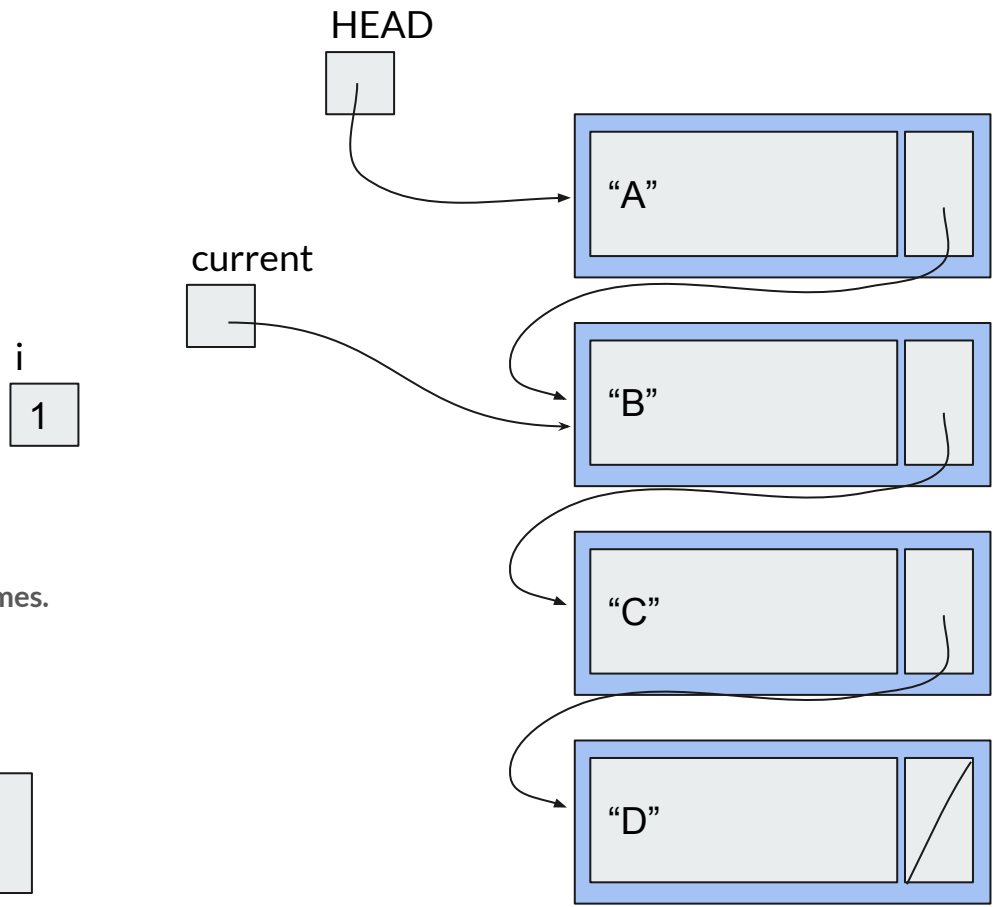
1. List is empty
2. **List non-empty**
    a. **Start at head**




```
l.get(2);
```

Two cases to handle:

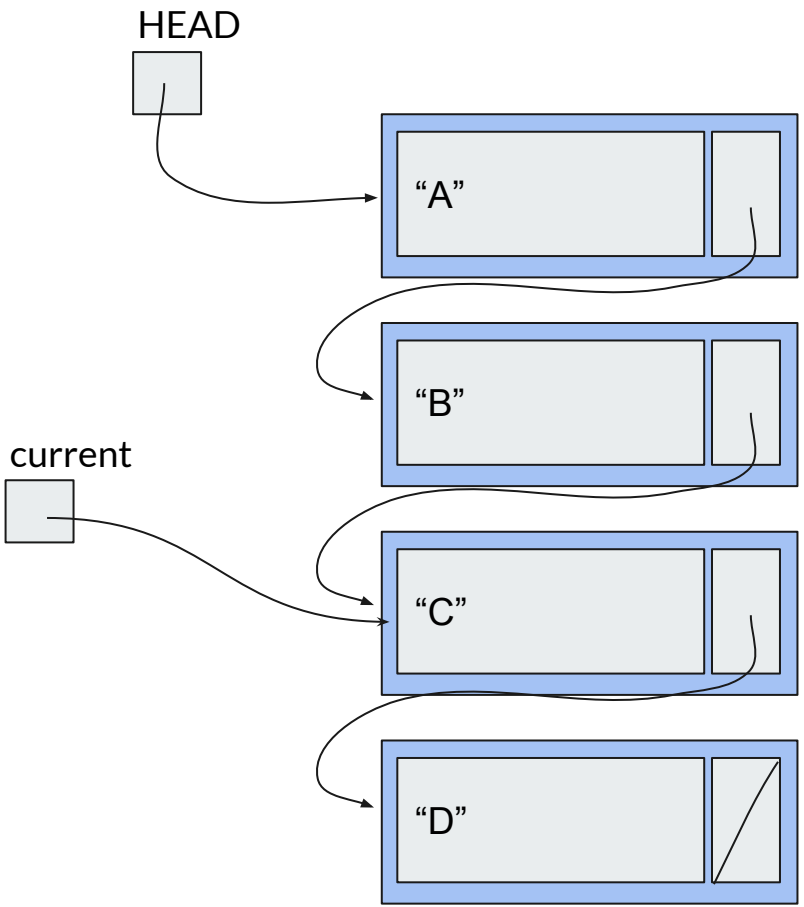
1. List is empty
2. **List non-empty**
   a. Start at head
   b. **Follow next pointers *index* times.**




```
l.get(2);
```

Two cases to handle:

1. List is empty
2. **List non-empty**
   a. Start at head
   b. **Follow next pointers *index* times.**



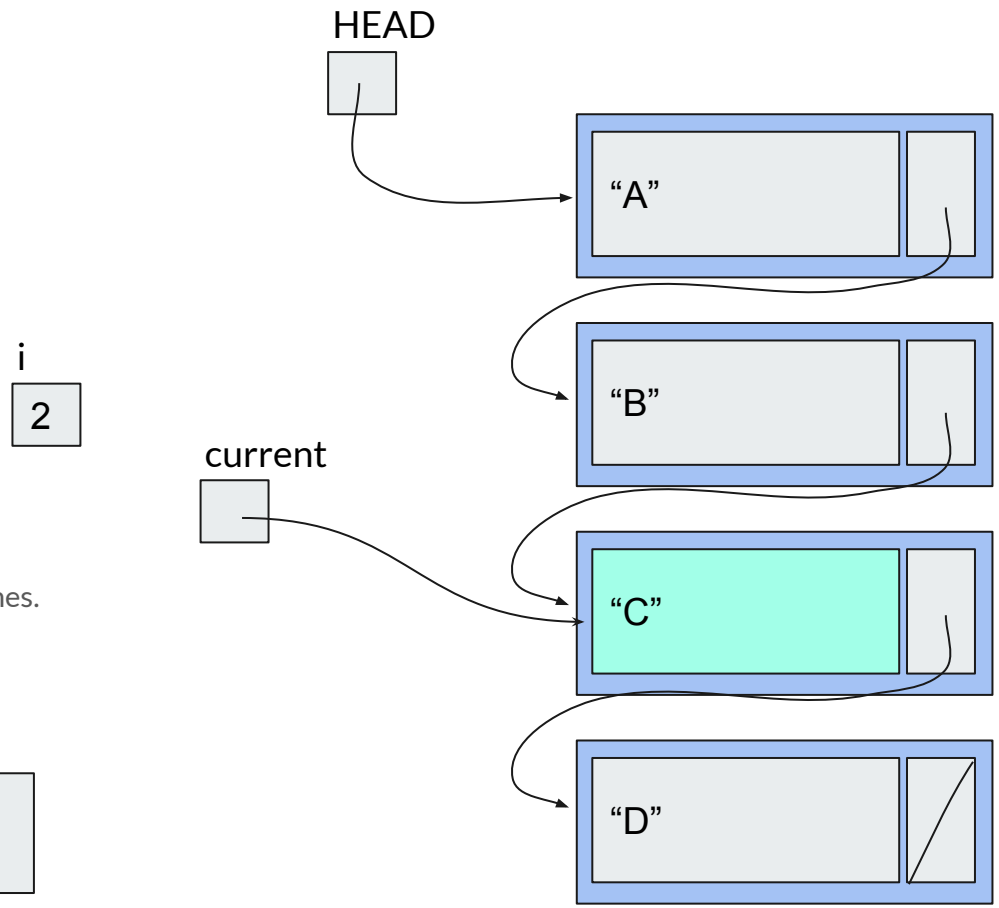
i

2

```
l.get(2);
```

Two cases to handle:

1. List is empty
2. **List non-empty**
   a. Start at head
   b. Follow next pointers *index* times.
   c. **Return current.data**




```
l.get(2);
```

Writing *get*. What goes here?

```
if (index < 0 || index >= size) {

    ____________________________________________

}
```

Writing *get*. What goes here?

```
if (index == 0) {
    ____________________________________
}
```

Organize the lines on the right to the correct location for the blanks on the left. This code should handle the case in *get* where we're getting from a valid index that's not the head.

```
// move to the node at index
A. _____
B. _____
while (i < index) {
    C. _____
    D. _____
}
E. _____
```

1	curr = curr.next;
2	i++;
3	*Node curr = head;*
4	return curr.data;
5	*int i = 0;*

```
// move to the node at index
A. _____
B. _____
while (i < index) {
    C. _____
    D. _____
}
E. _____
```

1	curr = curr.next;	C
2	i++;	D
3	*Node curr = head;*	B
4	return curr.data;	E
5	*int i = 0;*	A

```java
public String get(int index) {
        if (index < 0 || index ≥ size) {
            throw new IllegalArgumentException();
        }
        if (index == 0) {
            return head.data;
        }
        // move to the node at index
        int i = 0;
        Node curr = head;
        while (i < index) {
            curr = curr.next;
            i++;
        }
        return curr.data;
}
```

Remove

Two Cases to Handle (after checking for valid input)

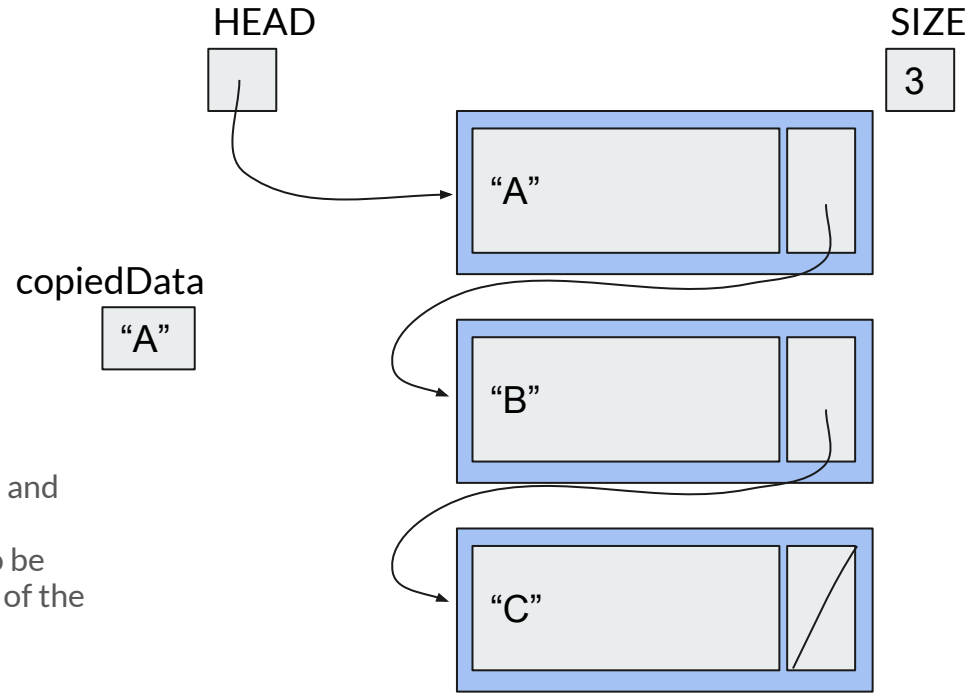
Remove at the head of the list

- Copy the data in *head*
- If *head* is the only node, remove it and decrement the size of the list
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- Return the data

Remove anywhere else

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Copy the data in current.next
- Set current.next to current.next.next
- Decrement the size of the list and return the saved data.

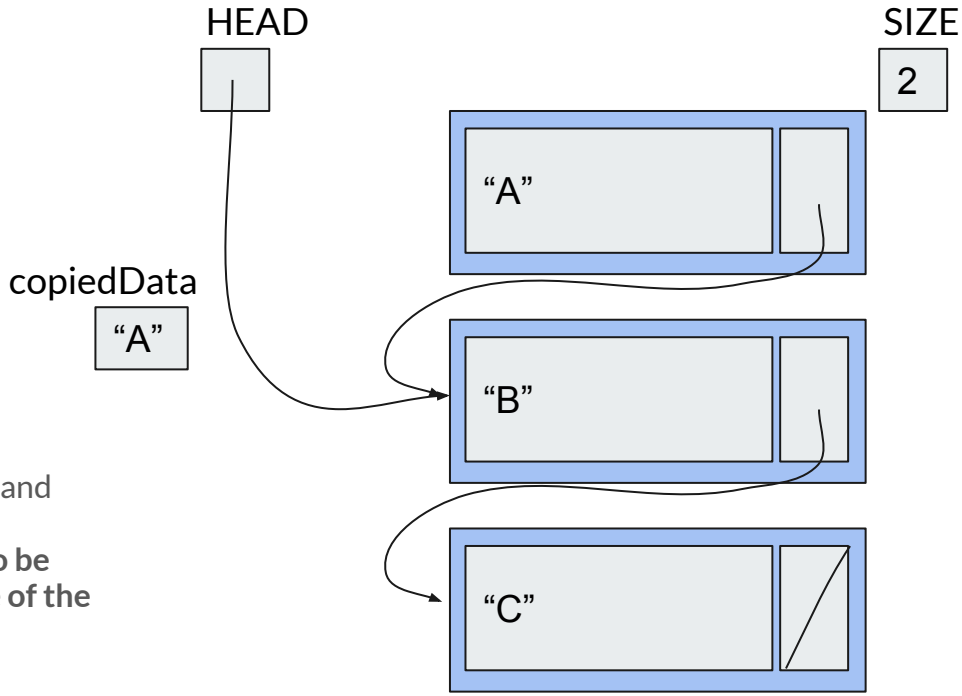
Removing at the head



- **Copy the data in *head***
- If *head* is the only node, remove it and decrement the size of the list
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- Return the data

```
l.remove(0)
```

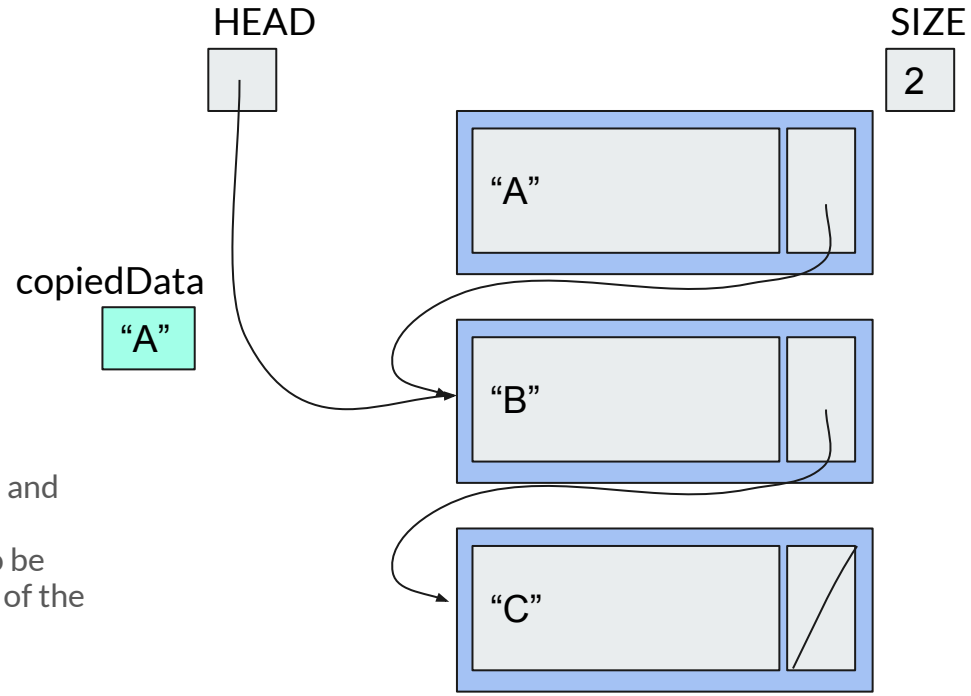
Removing at the head



- Copy the data in *head*
- If *head* is the only node, remove it and decrement the size of the list
- **If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list**
- Return the data

```
l.remove(0)
```

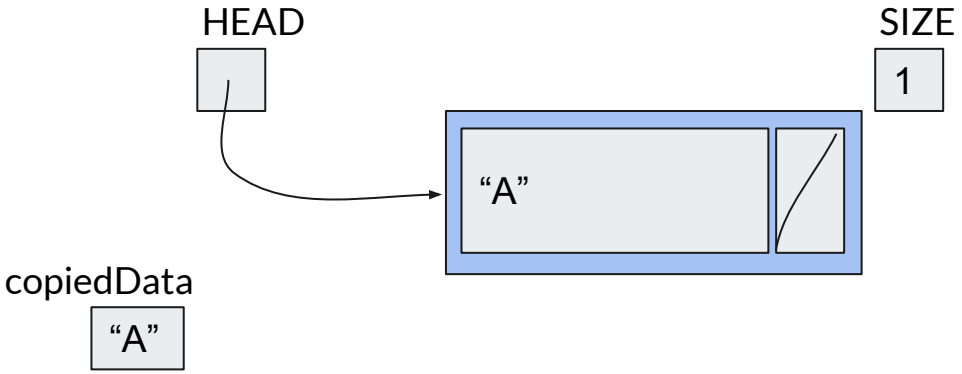
Removing at the head



- Copy the data in *head*
- If *head* is the only node, remove it and decrement the size of the list
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- **Return the data**

```
l.remove(0)
```

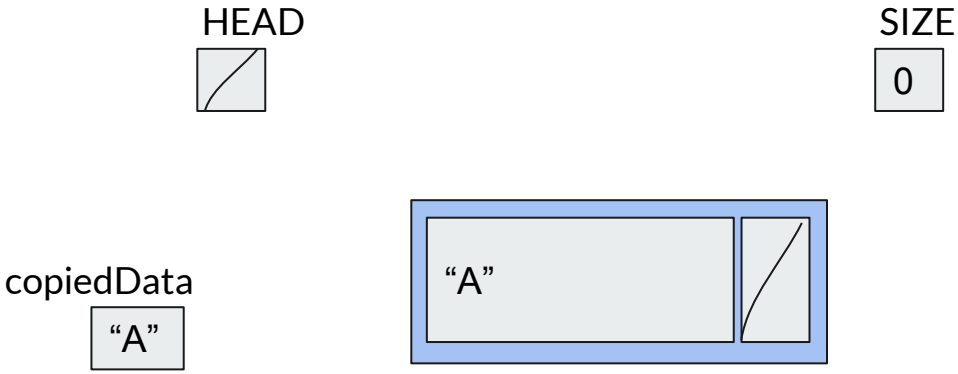
Removing at the head, take 2



- **Copy the data in *head***
- If *head* is the only node, remove it and decrement the size of the list
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- Return the data

```
l.remove(0)
```

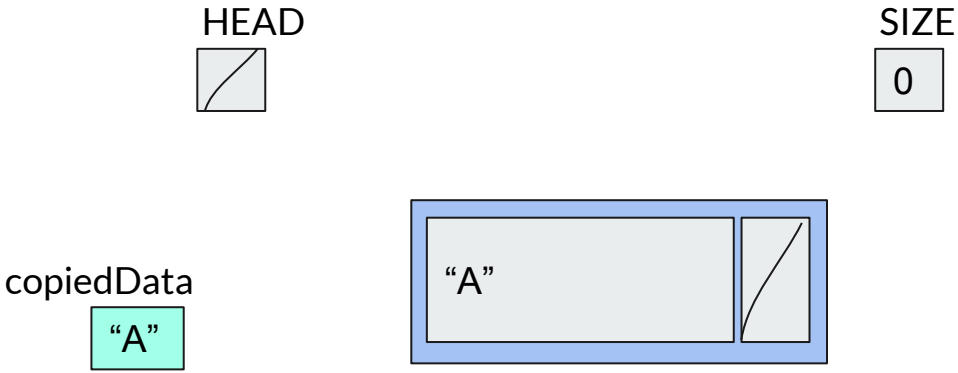
Removing at the head, take 2



- Copy the data in *head*
- **If *head* is the only node, remove it and decrement the size of the list**
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- Return the data

```
l.remove(0)
```

Removing at the head, take 2

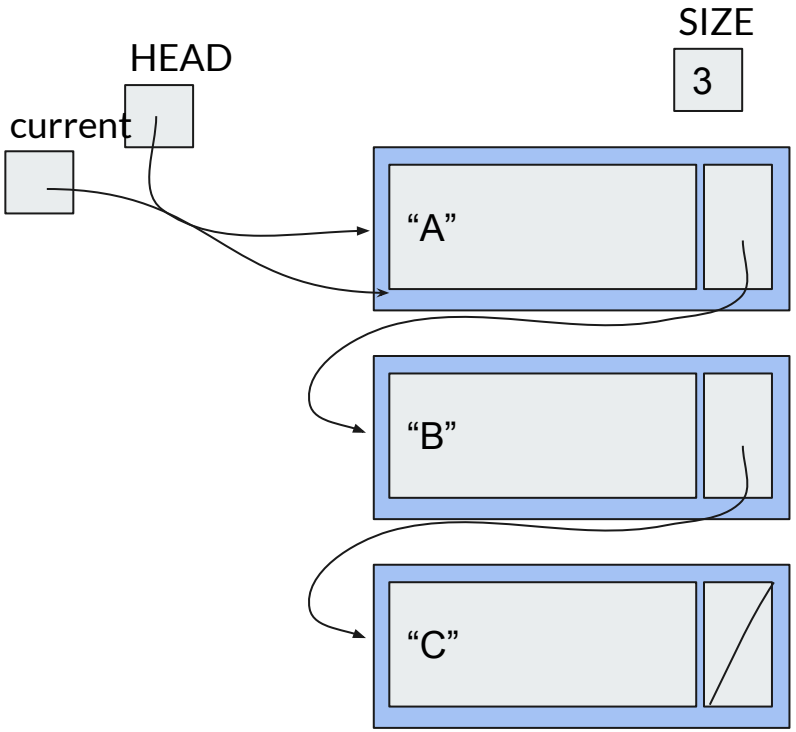


- Copy the data in *head*
- If *head* is the only node, remove it and decrement the size of the list
- If *head* has a *next*, then set *head* to be *head.next* and decrement the size of the list
- **Return the data**

```
l.remove(0)
```

Removing within the List

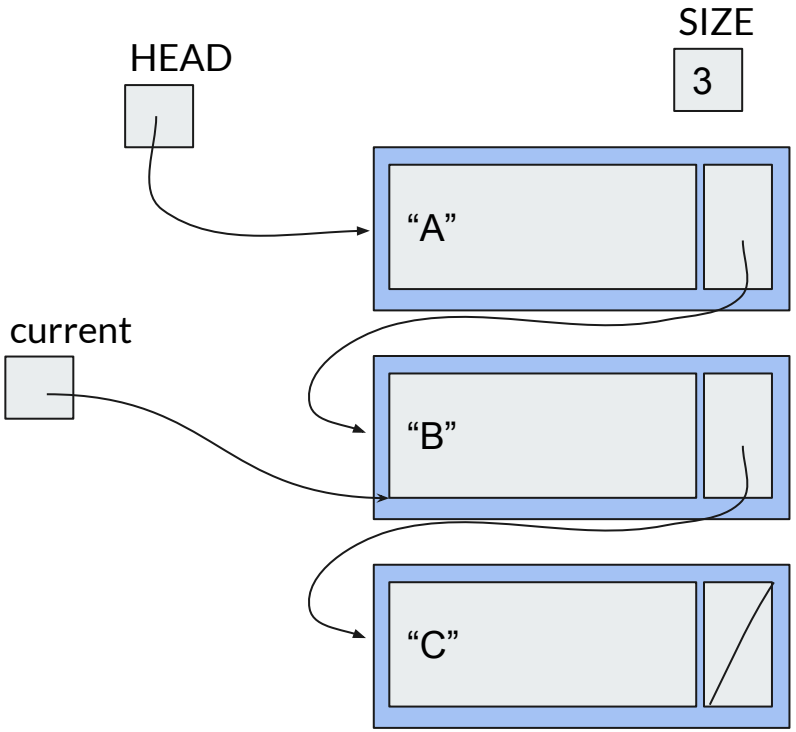
- **Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current***
- Copy the data in current.next
- Set current.next to current.next.next
- Decrement the size of the list and return the saved data.




```
l.remove(2);
```

Removing within the List

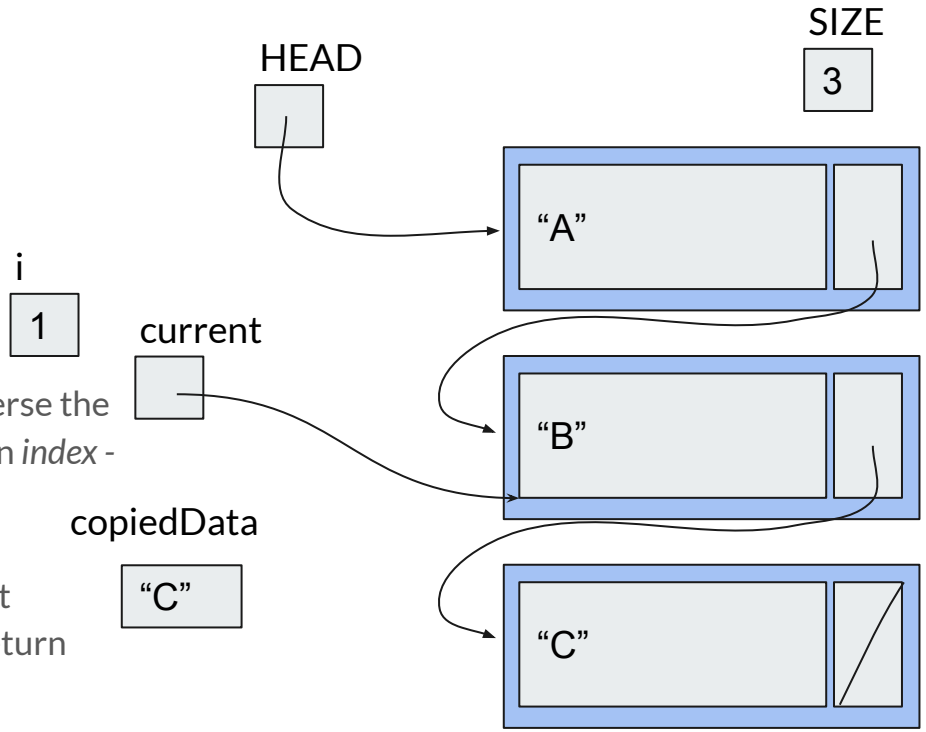
- **Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current***
- Copy the data in current.next
- Set current.next to current.next.next
- Decrement the size of the list and return the saved data.




```
l.remove(2);
```

Removing within the List

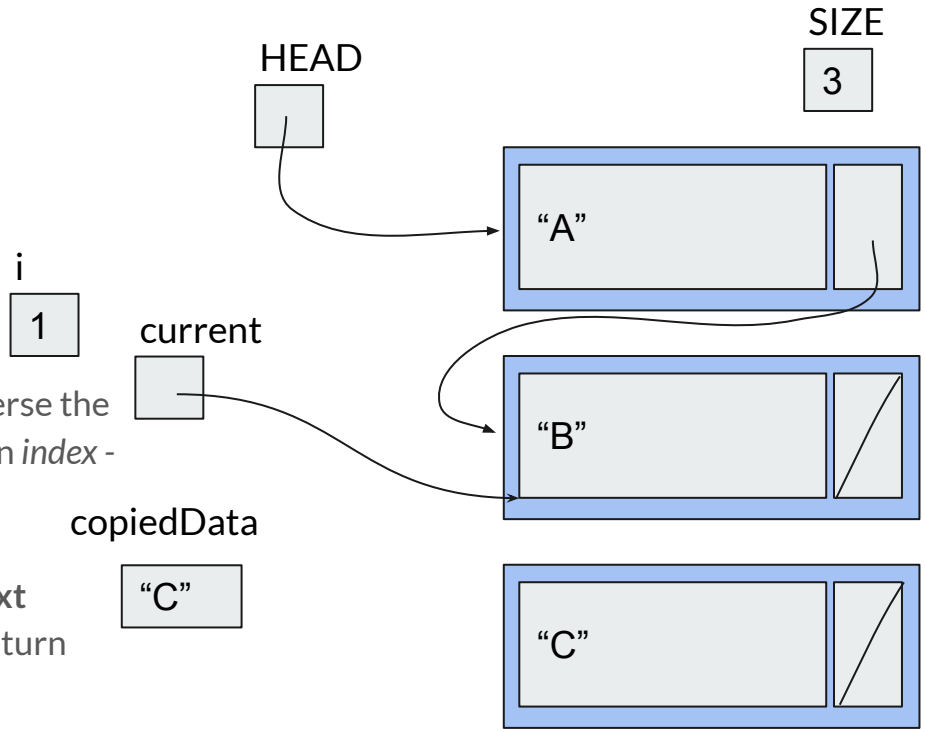
- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- **Copy the data in current.next**
- Set current.next to current.next.next
- Decrement the size of the list and return the saved data.




```
l.remove(2);
```

Removing within the List

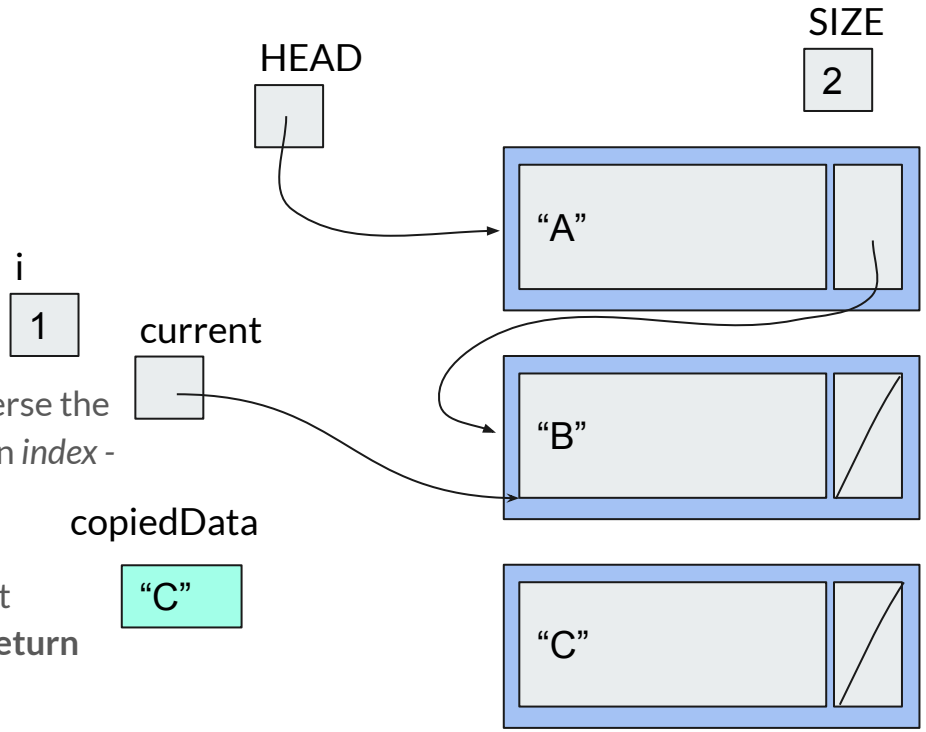
- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Copy the data in current.next
- **Set current.next to current.next.next**
- Decrement the size of the list and return the saved data.




```
l.remove(2);
```

Removing within the List

- Start at the head of the list and traverse the nodes until you're at node at position *index - 1*, call this node *current*
- Copy the data in current.next
- Set current.next to current.next.next
- **Decrement the size of the list and return the saved data.**




```
l.remove(2);
```

```java
public String remove(int index) {
    if (index < 0 || index >= size) {
        throw new IllegalArgumentException();
    }
```

The start of *remove*

Unscramble the lines on the right to finish *remove*. (error checking already done)

```
String value = "";
if (A. ________) { // remove at head
    B. ________
    C. ________
} else {
    // move to the correct position
    D. ________
    E. ________;
    while (F. ________) {
        G. ________
        H. ________
    }
    I. ________ // Remember value
    J. ________ // remove node
}
K. ________
L. ________
```

1	curr.next = curr.next.next;
2	*int i = 0;*
3	return value;
4	Node curr = head;
5	size--;
6	*index == 0*
7	value = curr.next.data;
8	i++;
9	curr = curr.next;
10	value = head.data;
11	*i < index - 1*
12	head = head.next;

```
String value = "";
if (A. ________) { // remove at head
    B. ________
    C. ________
} else {
    // move to the correct position
    D. ________
    E. ________;
    while (F. ________) {
        G. ________
        H. ________
    }
    I. ________ // Remember value
    J. ________ // remove node
}
K. ________
L. ________
```

1	curr.next = curr.next.next;	J
2	*int i = 0;*	D
3	return value;	L
4	Node curr = head;	E
5	size--;	K
6	*index == 0*	A
7	value = curr.next.data;	I
8	i++;	H
9	curr = curr.next;	G
10	value = head.data;	B
11	*i < index - 1*	F
12	head = head.next;	C

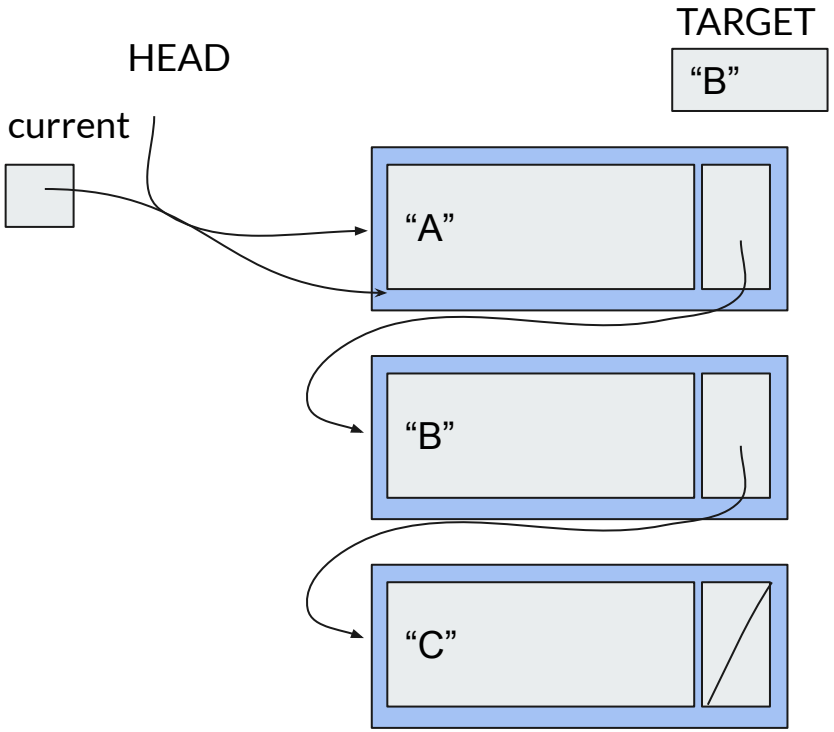
```java
public String remove(int index) {
    if (index < 0 || index ≥ size) {
        throw new IllegalArgumentException();
    }
    String value = "";
    if (index == 0) { // remove at head
        value = head.data;
        head = head.next;
    } else {
        // move to the correct position
        int i = 0;
        Node curr = head;
        while (i < index - 1) {
            curr = curr.next;
            i++;
        }
        value = curr.next.data; // Remember value
        curr.next = curr.next.next; // remove node
    }
    size--;
    return value;
}
```

Contains

Just one case! (Whew...)

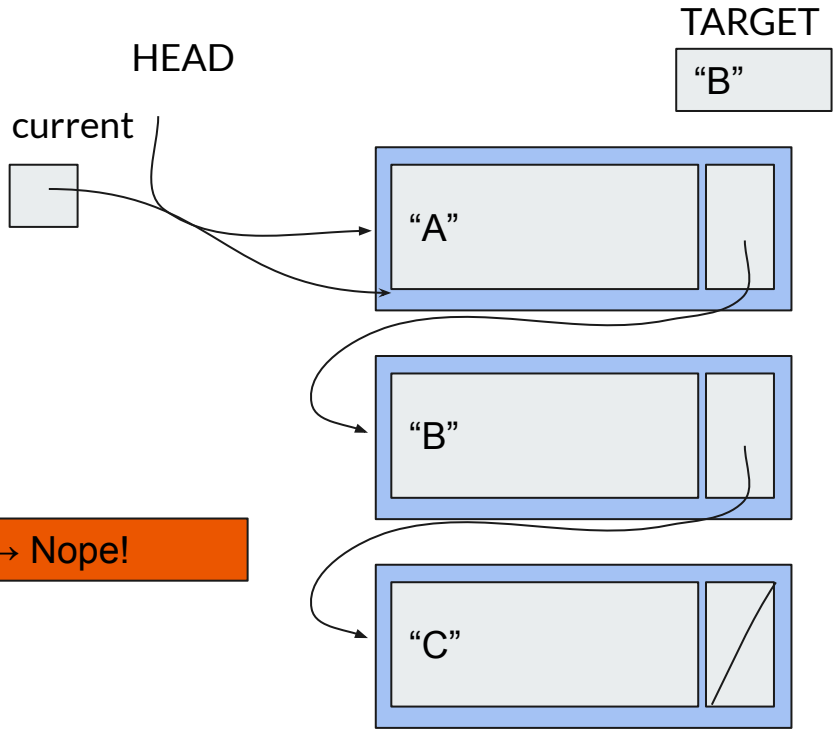
Searching the list for a target value

- **Start at the head of the list**
- Check each Node's data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.




```
l.contains("B");
```

Just one case! (Whew...)



Searching the list for a target value

- Start at the head of the list
- **Check each Node's data to see if it matches the target**
- If so, return true
- Keep going until current Node is null, and then return false.

"B".equals("A") → Nope!

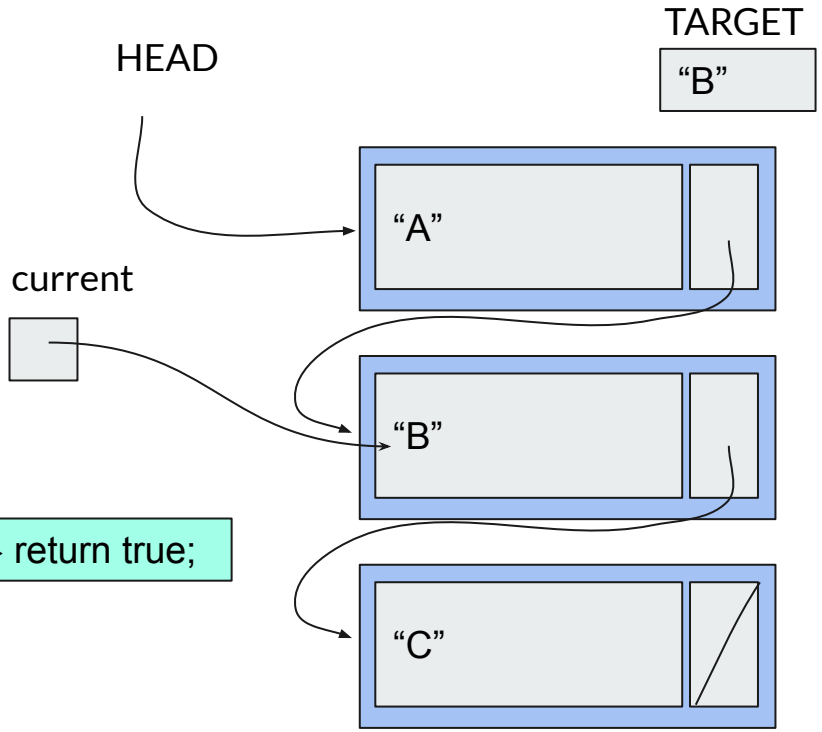
```
l.contains("B");
```

Just one case! (Whew...)

Searching the list for a target value

- Start at the head of the list
- Check each Node's data to see if it matches the target
- **If so, return true**
- Keep going until current Node is null, and then return false.

"B".equals("B") → return true;

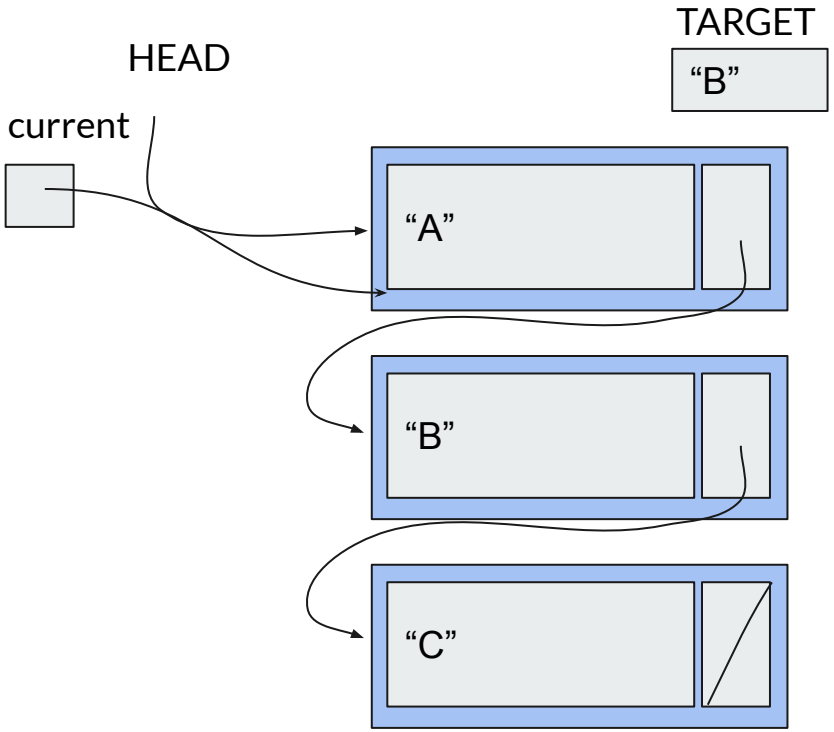



```
l.contains("B");
```

What if it's not there?

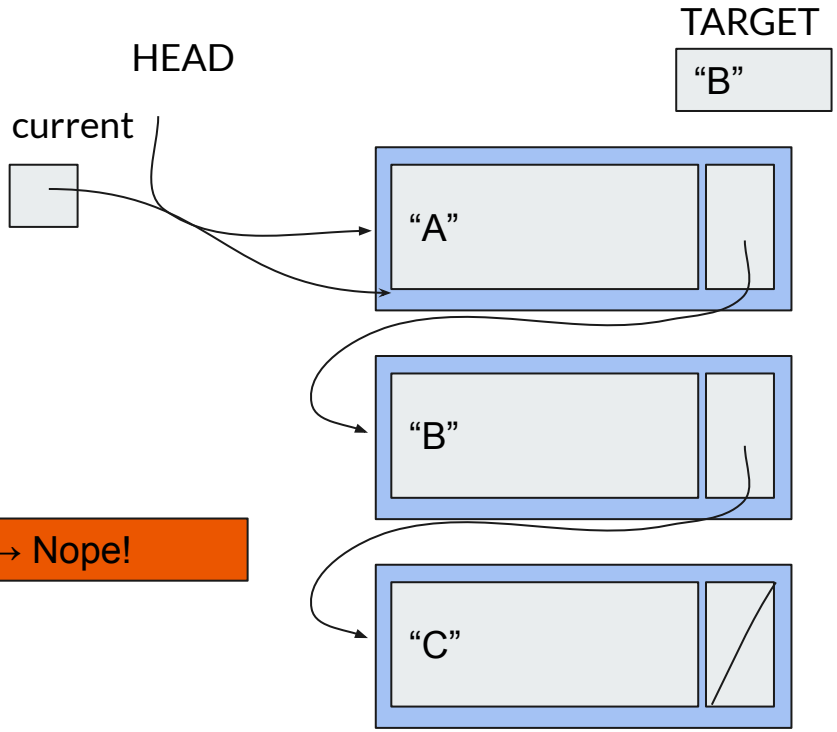
Searching the list for a target value

- **Start at the head of the list**
- Check each Node's data to see if it matches the target
- If so, return true
- Keep going until current Node is null, and then return false.




```
l.contains("D");
```

What if it's not there?



Searching the list for a target value

- Start at the head of the list
- **Check each Node's data to see if it matches the target**
- If so, return true
- Keep going until current Node is null, and then return false.

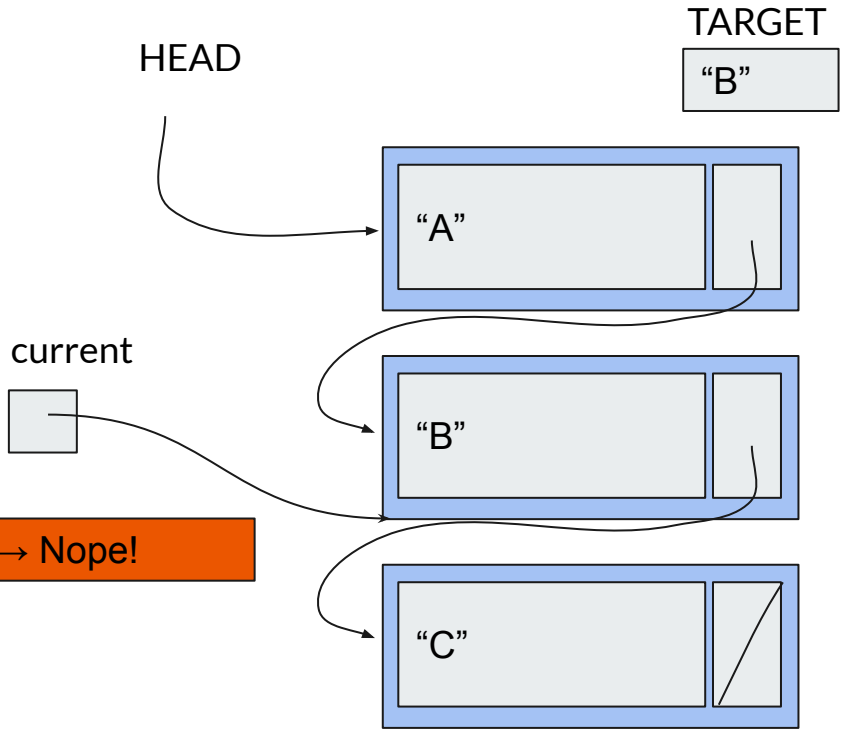
"D".equals("A") → Nope!

```
l.contains("D");
```

What if it's not there?

Searching the list for a target value

- Start at the head of the list
- **Check each Node's data to see if it matches the target**
- If so, return true
- Keep going until current Node is null, and then return false.



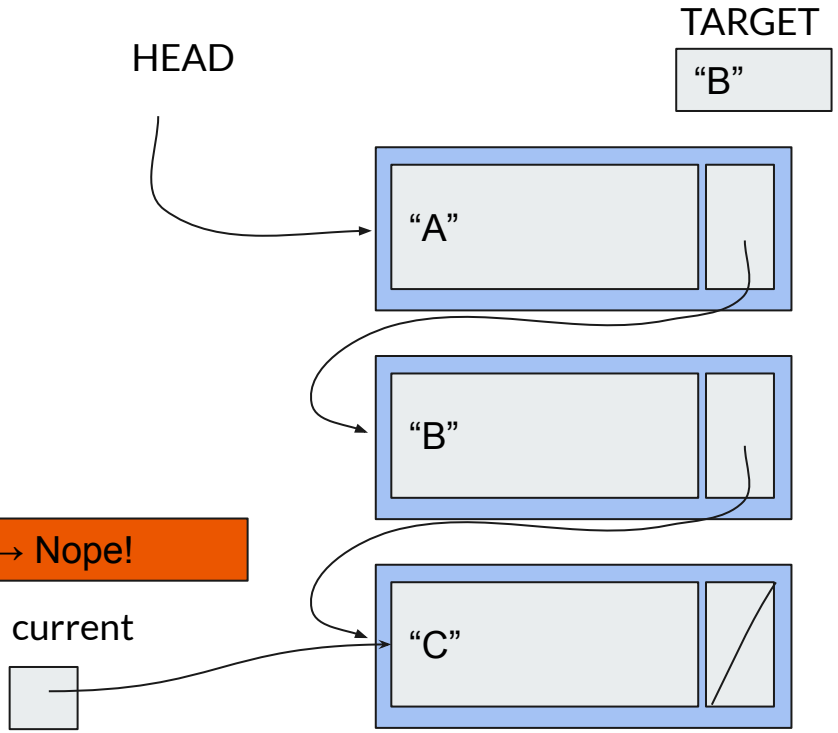

```
l.contains("D");
```

What if it's not there?

Searching the list for a target value

- Start at the head of the list
- **Check each Node's data to see if it matches the target**
- If so, return true
- Keep going until current Node is null, and then return false.

"D".equals("C") → Nope!



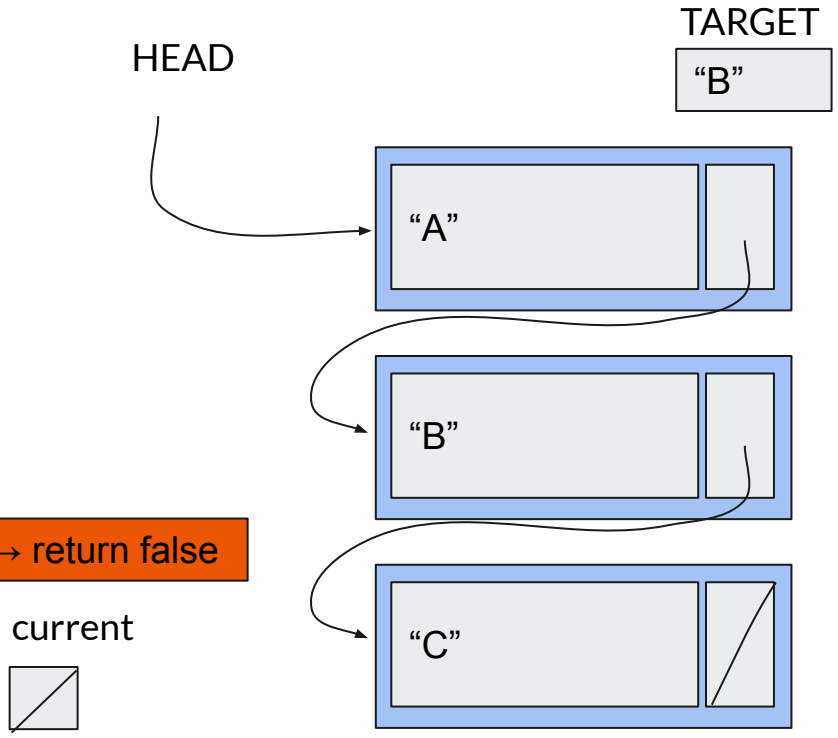

```
l.contains("D");
```

What if it's not there?

Searching the list for a target value

- Start at the head of the list
- **Check each Node's data to see if it matches the target**
- If so, return true
- Keep going until current Node is null, and then return false.

Current == null → return false




```
l.contains("D");
```

The rest

size()

Returns the current number of elements in the list.

```
// Returns number of elements
// in this list
public int size() {
    return size;
}
```

isEmpty()

Returns true when the list is empty.

```
public boolean isEmpty() {
    return (head == null);
}
```