CSCI 132:
Basic Data Structures and Algorithms

Circular Linked Lists

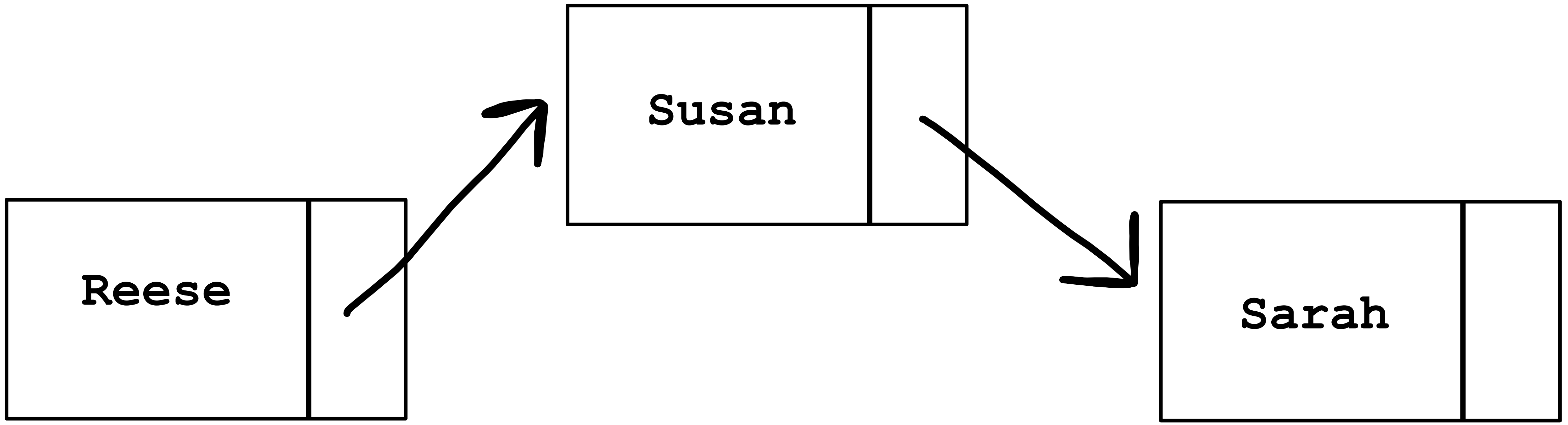
Reese Pearsall

Spring 2024

Program 2 due Friday March 8th @ 11:59 PM

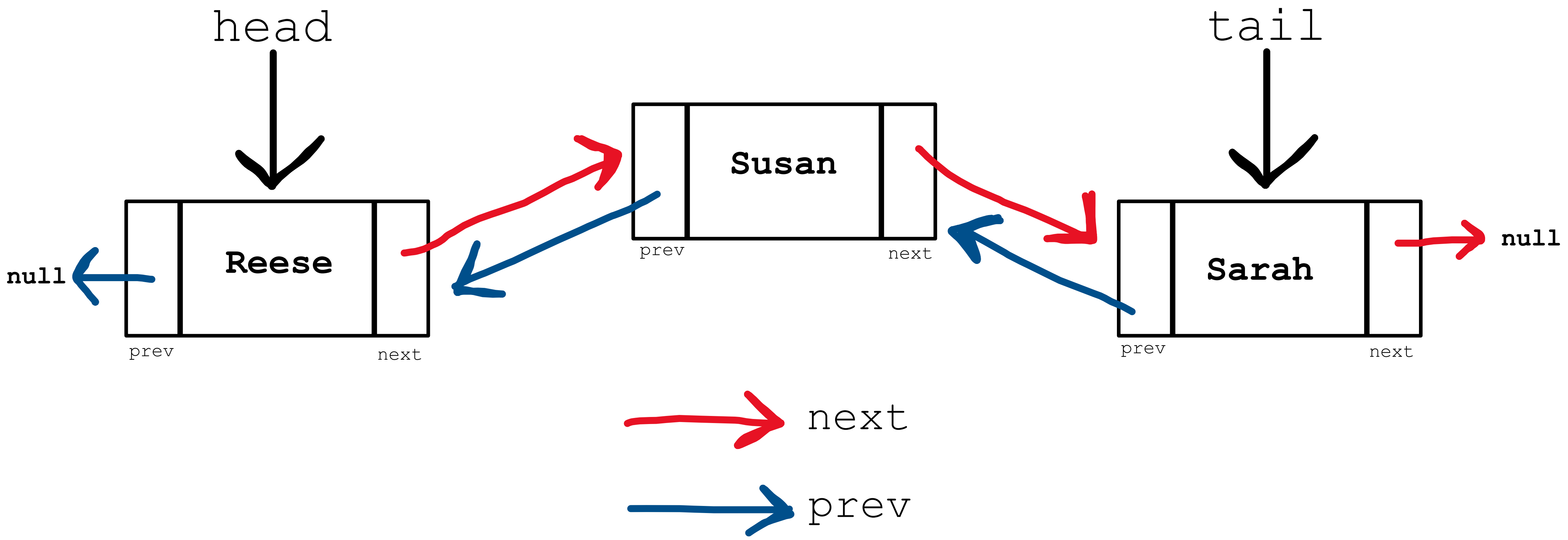
→ After today, you should be able to complete it.

A **Linked List** is a data structure that consists of a collection of connected nodes

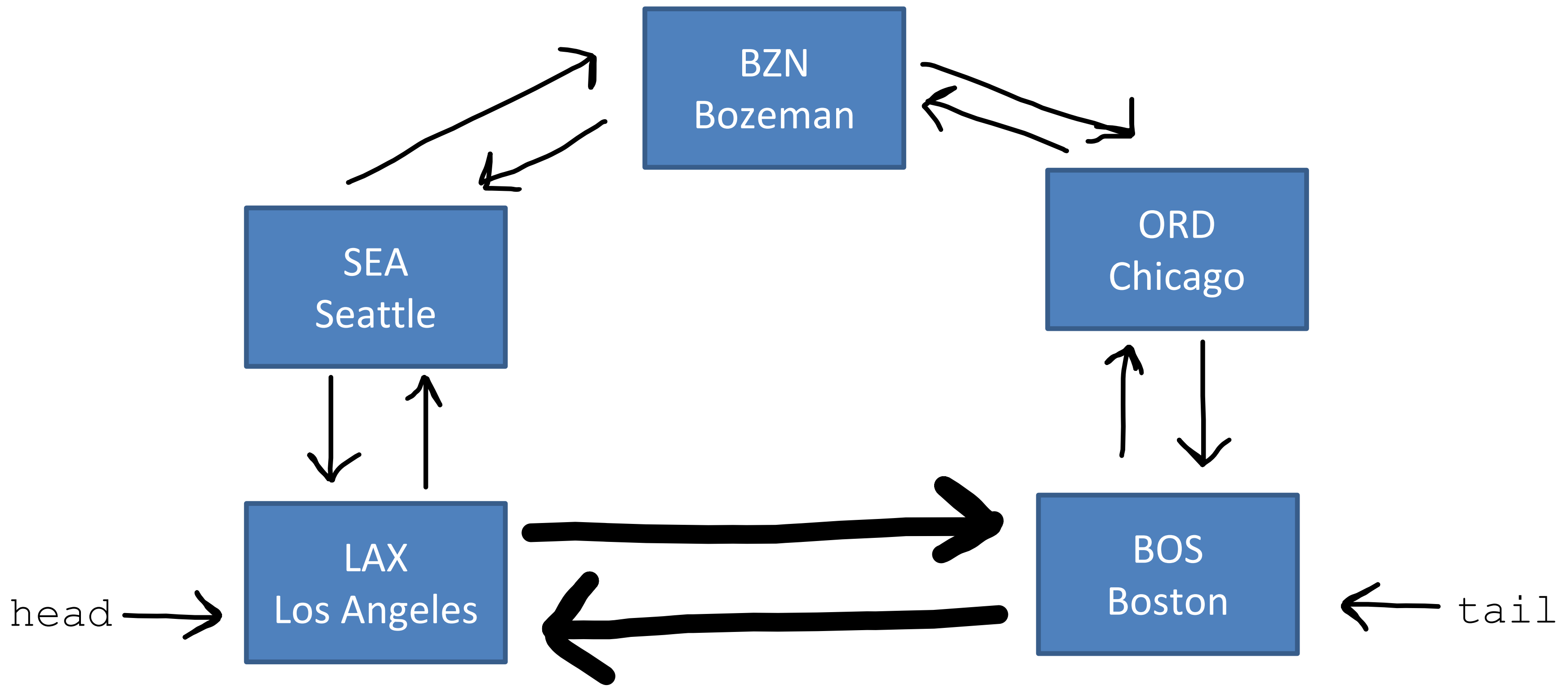


Nodes consists of **data** (String, int, array, etc) and a **pointer to the next node**

A **Doubly Linked List** keeps track of the next node and the previous node

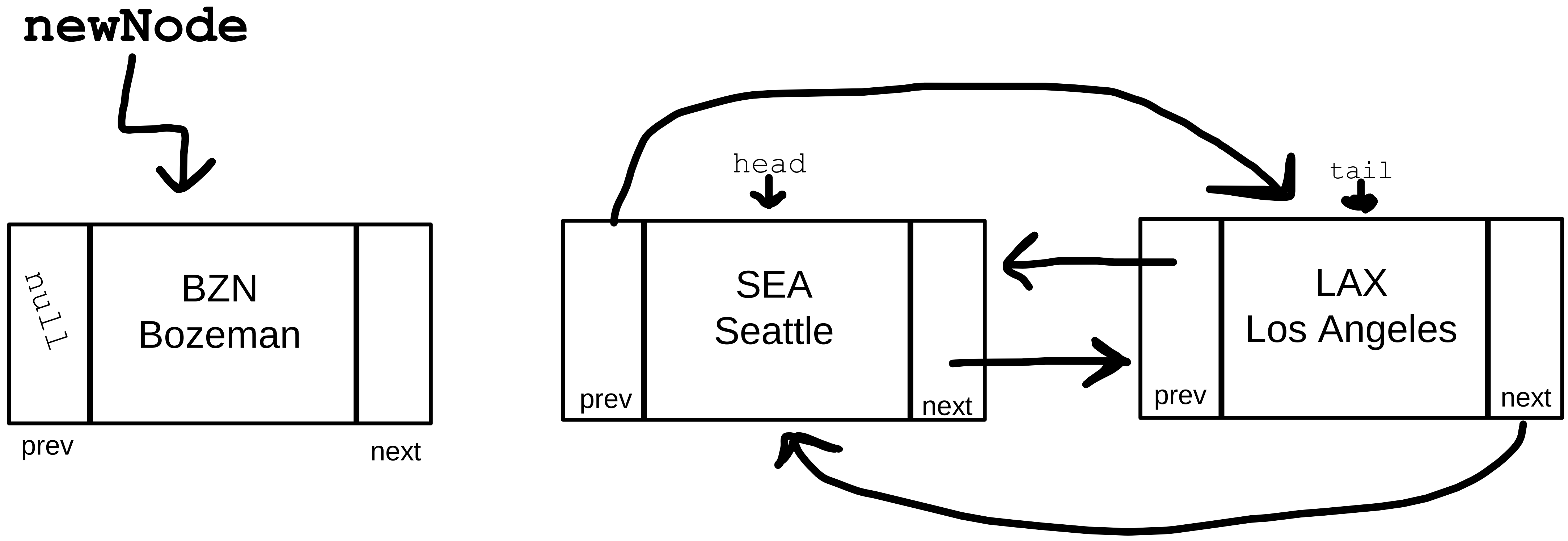


A **Circular Linked List** is a linked list where the first and last node are connected, which creates a circle



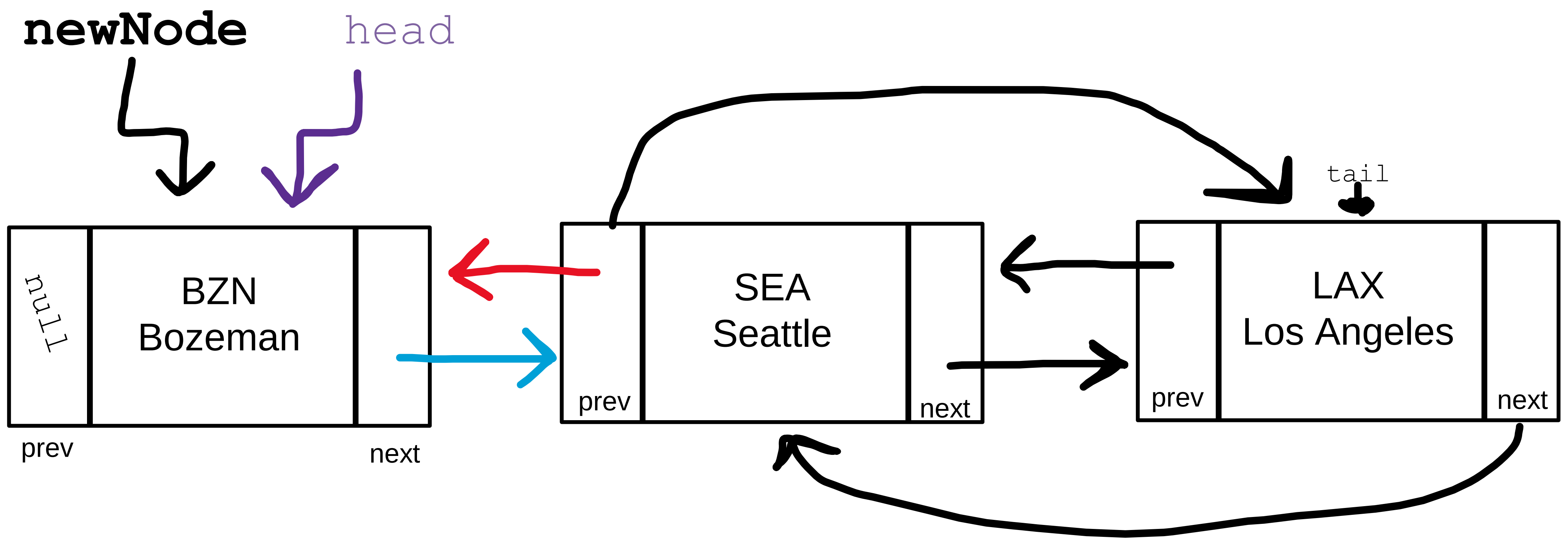
We will take our Doubly Linked List Implementation, and convert it into a Circular Doubly Linked List

Case 2: The user is inserting a node at the very beginning (N = 1)



???

Case 2: The user is inserting a node at the very beginning (N = 1)



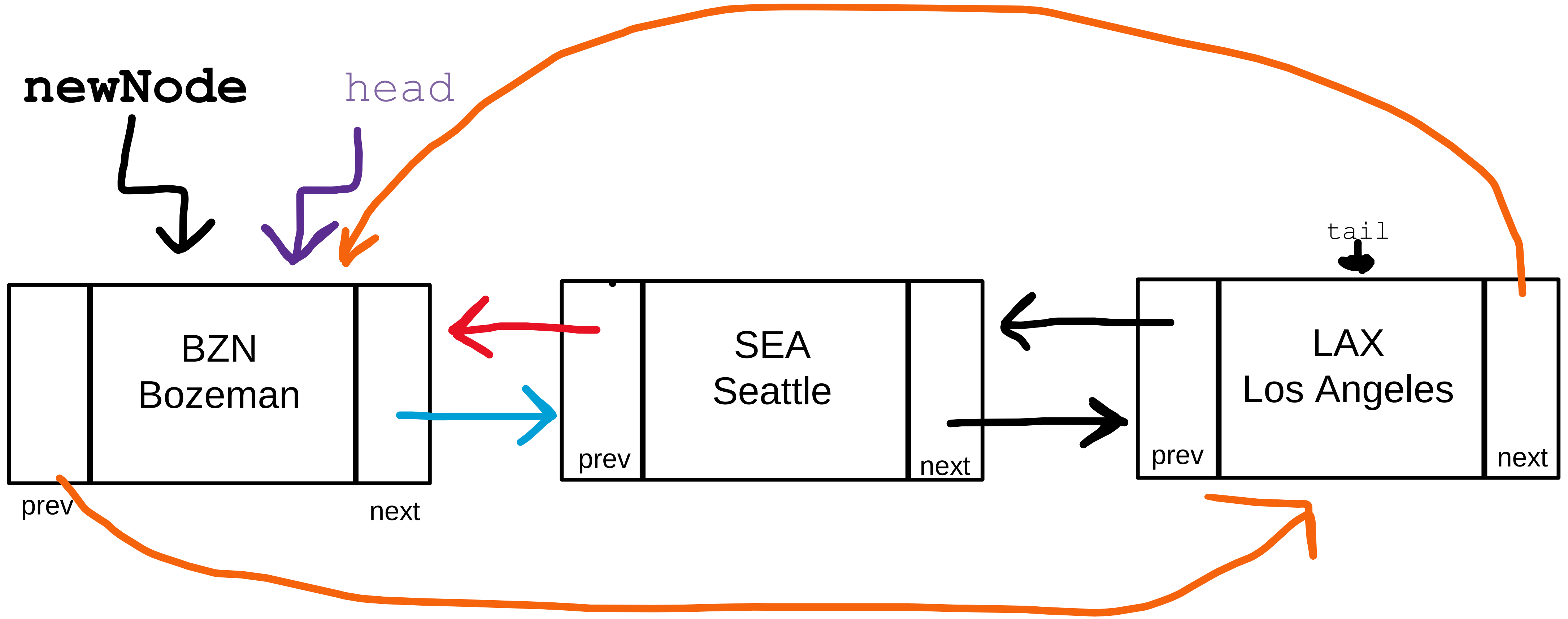
Update the `head` node `prev` value to `newNode`

Update the `newNode`'s `next` value to be the current `head` node

Update the `head` node to be the `newNode`

NEW: Because this is a circular linked list, we need to make sure our tail and head are connected

Case 2: The user is inserting a node at the very beginning (N = 1)



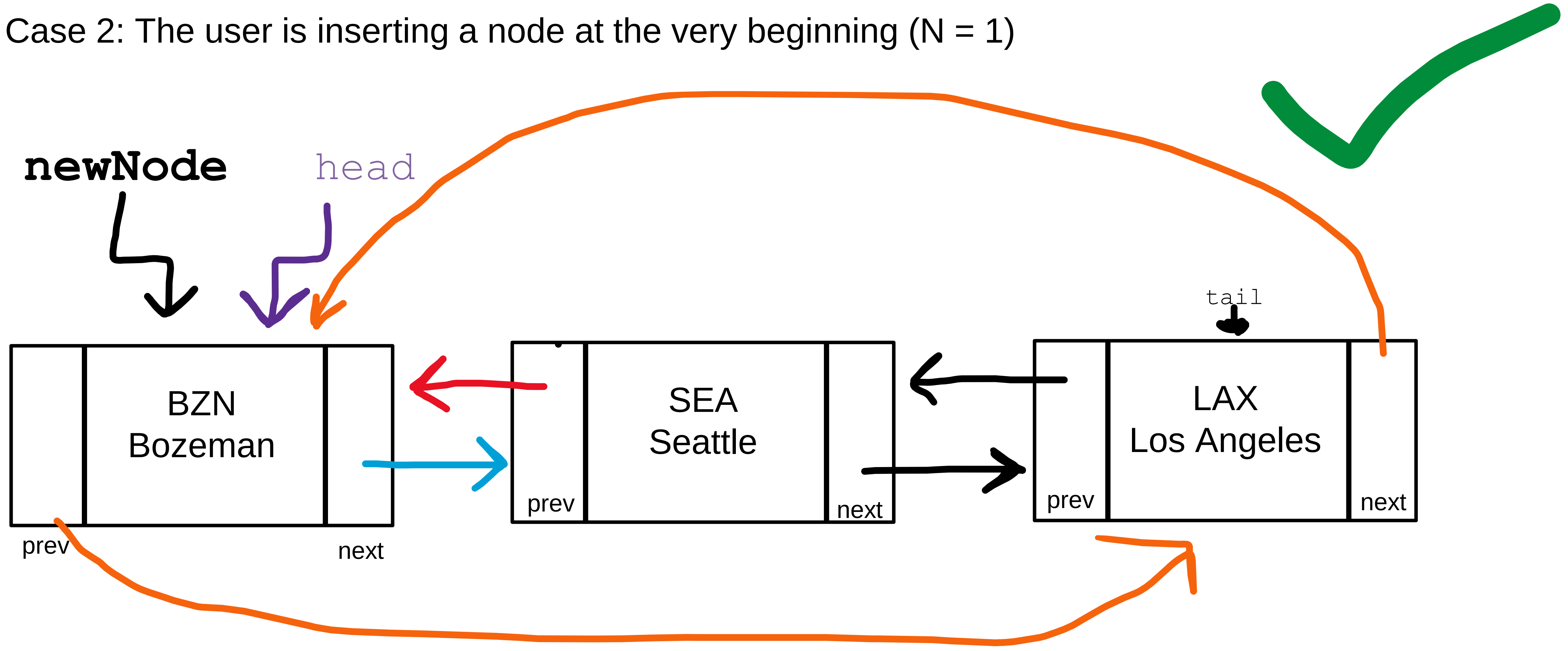
Update the `head` node `prev` value to `newNode`

Update the `newNode`'s `next` value to be the current `head` node

Update the `head` node to be the `newNode`

NEW: Reconnect the `head` and `tail` node

Case 2: The user is inserting a node at the very beginning (N = 1)



Update the `head` node `prev` value to `newNode`

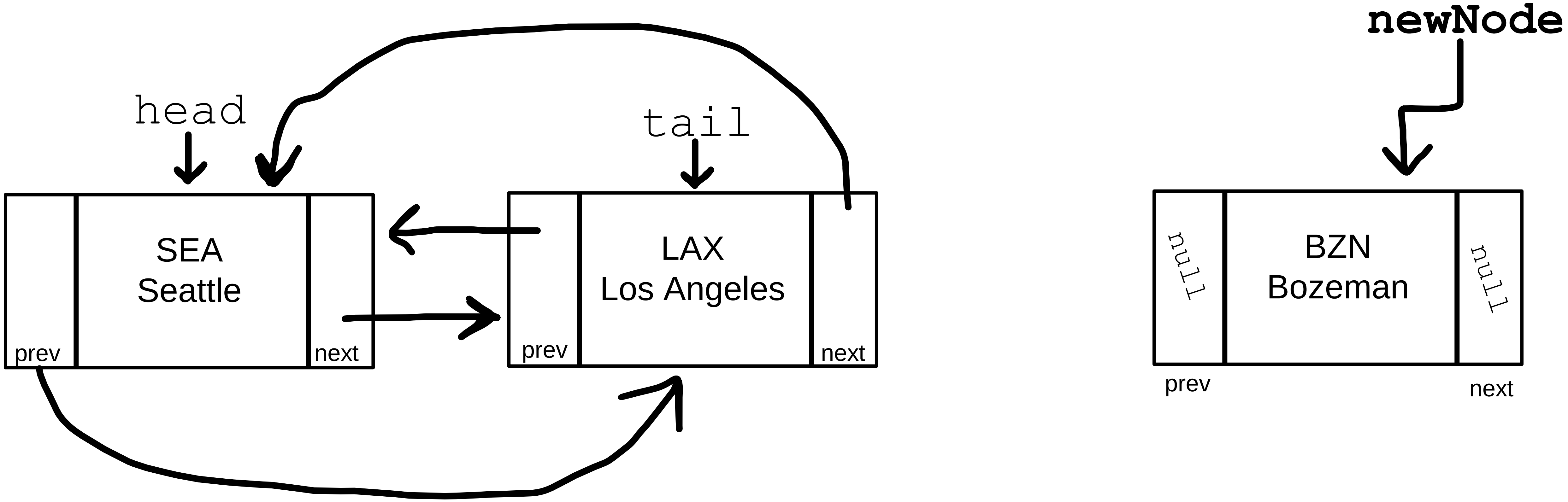
Update the `newNode`'s `next` value to be the current `head` node

Update the `head` node to be the `newNode`

NEW: Reconnect the `head` and `tail` node

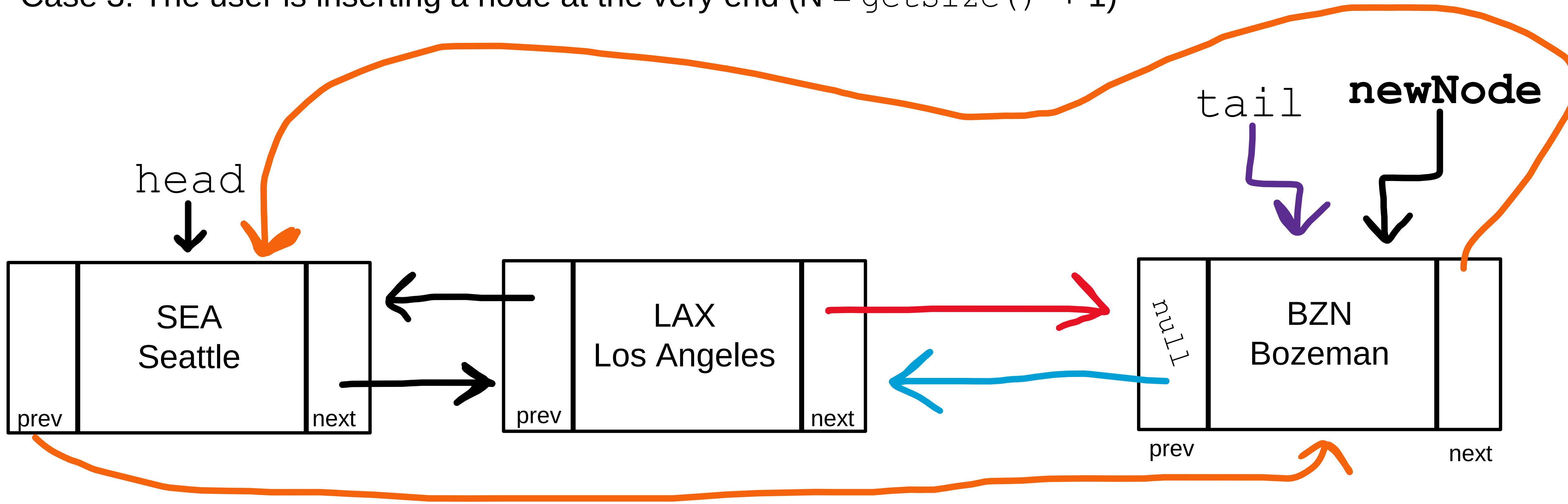
- **`insert(newNode, N)`** — Insert new node (`newNode`) at spot **N**

Case 3: The user is inserting a node at the very end (N = `getSize()` + 1)



- **`insert(newNode, N)`** – Insert new node (`newNode`) at spot **N**

Case 3: The user is inserting a node at the very end (N = `getSize()` + 1)



Update the `tail` node `next` value to `newNode`

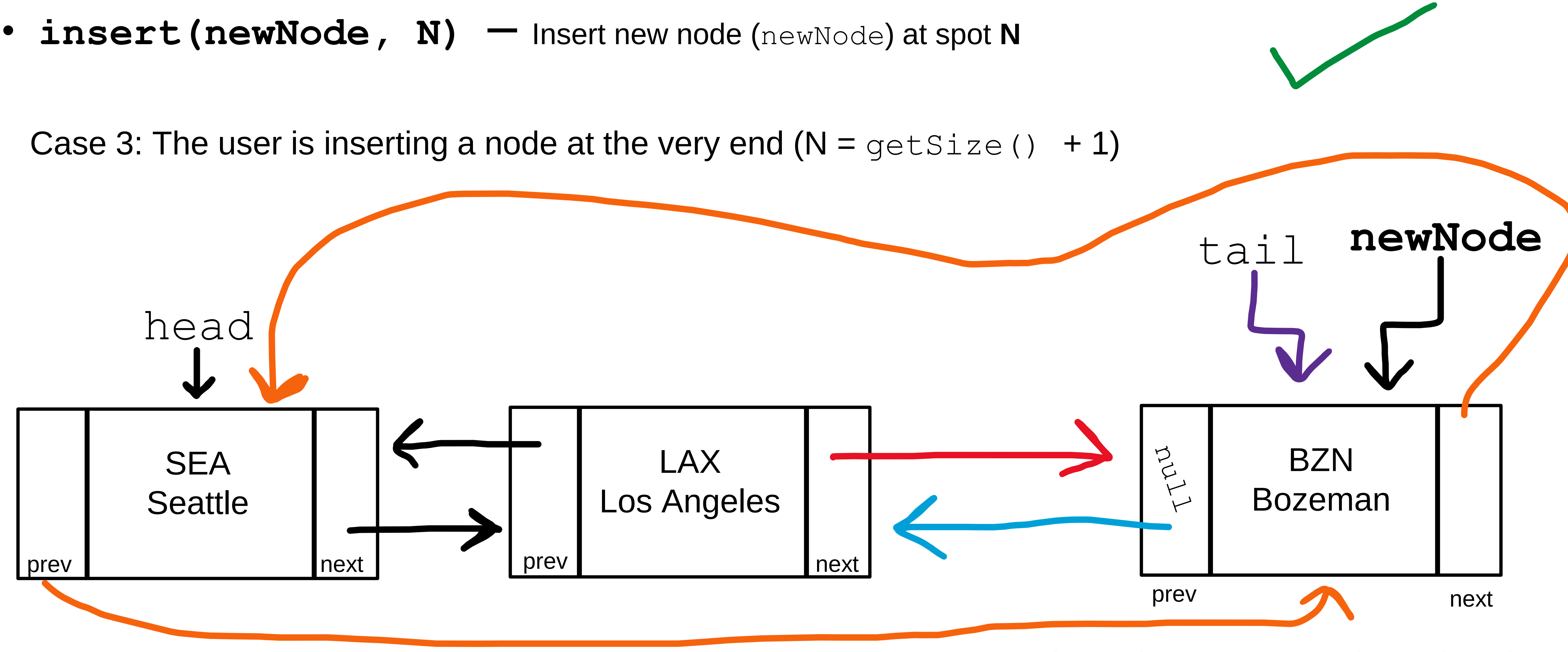
Update the `newNode`'s `prev` value to be the current `tail` node

Update the `tail` node to be the `newNode`

NEW: Reconnect the `head` and `tail` node

- **`insert(newNode, N)`** — Insert new node (`newNode`) at spot **N**

Case 3: The user is inserting a node at the very end (N = `getSize()` + 1)



Update the `tail` node `next` value to `newNode`

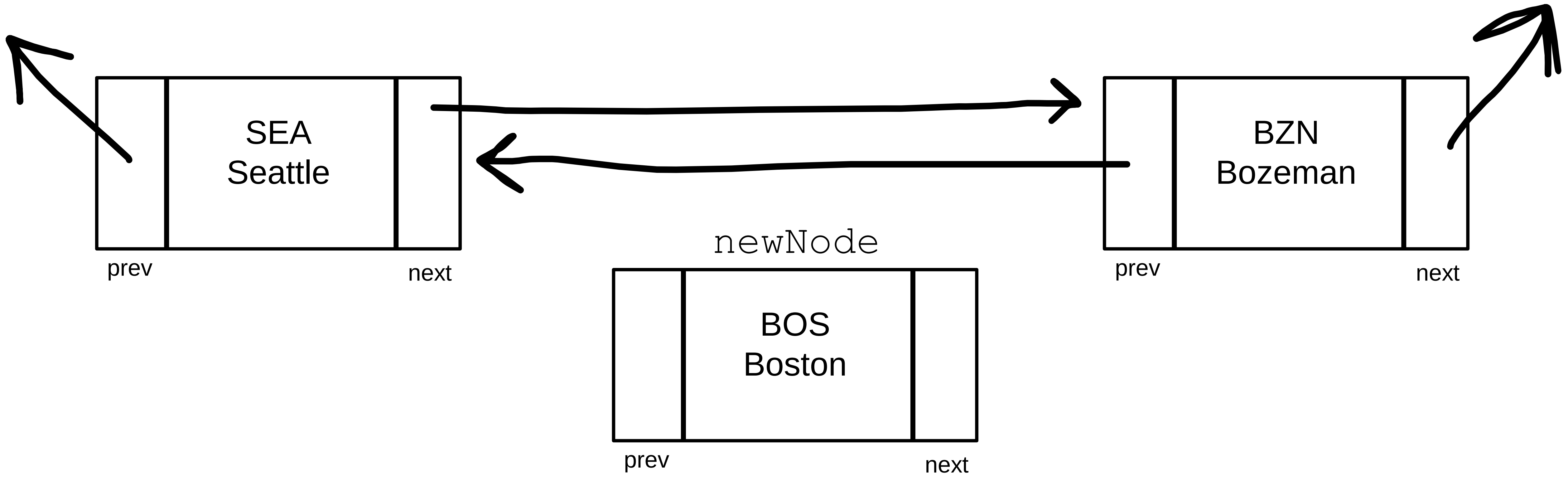
Update the `newNode`'s `prev` value to be the current `tail` node

Update the `tail` node to be the `newNode`

NEW: Reconnect the `head` and `tail` node

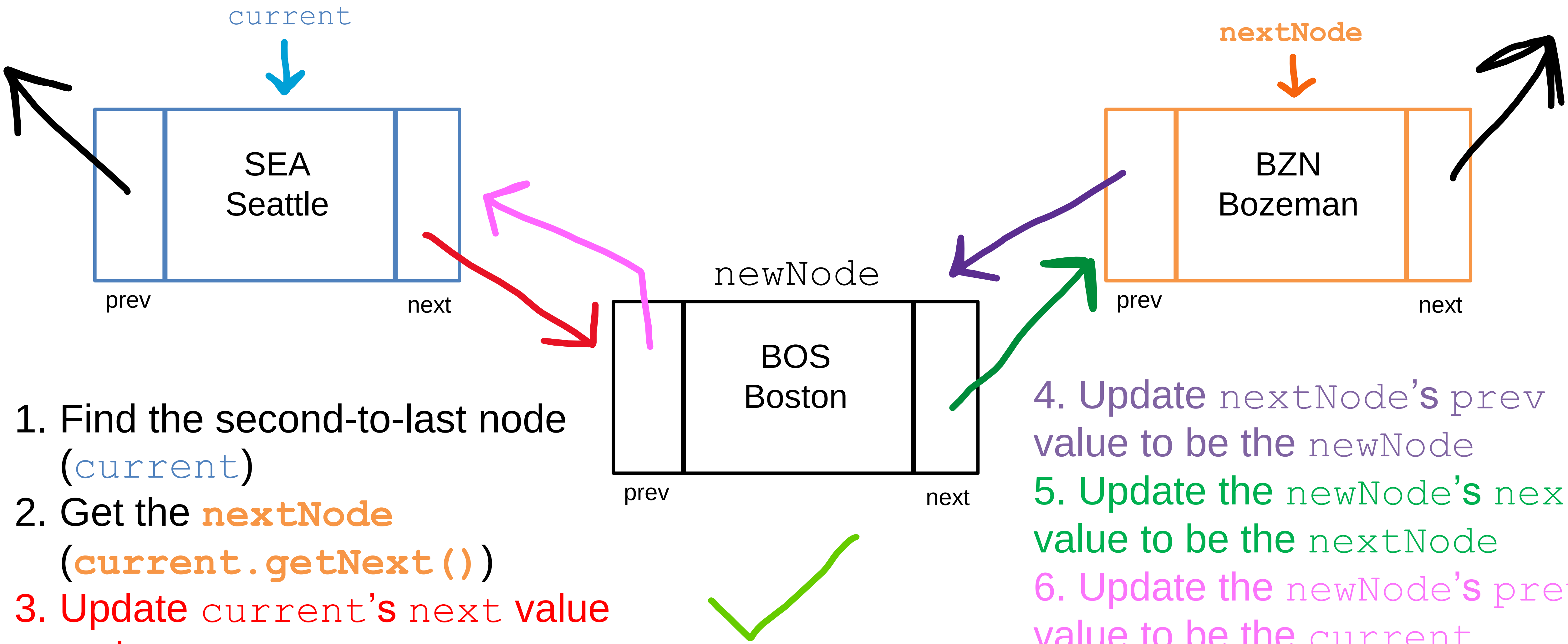
- **`insert(newNode, N)`** – Insert new node (`newNode`) at spot **N**

Case 4: The user is inserting a node somewhere in the middle of the LL



- **`insert(newNode, N)`** — Insert new node (`newNode`) at spot **N**

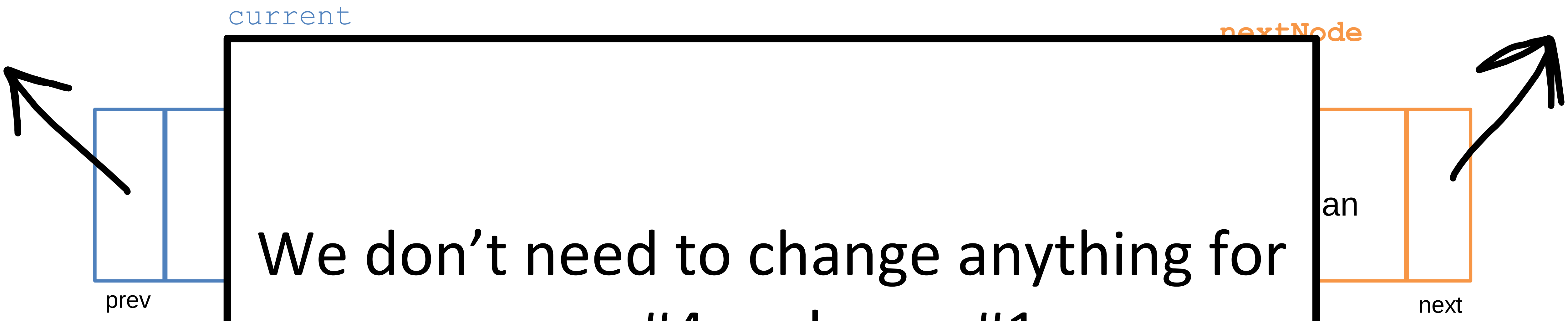
Case 4: The user is inserting a node somewhere in the middle of the LL



1. Find the second-to-last node (`current`)
2. Get the `nextNode` (`current.getNext()`)
3. Update `current`'s `next` value to the `newNode`
4. Update `nextNode`'s `prev` value to be the `newNode`
5. Update the `newNode`'s `next` value to be the `nextNode`
6. Update the `newNode`'s `prev` value to be the `current`

- **`insert(newNode, N)`** — Insert new node (`newNode`) at spot **N**

Case 4: The user is inserting a node somewhere in the middle of the LL



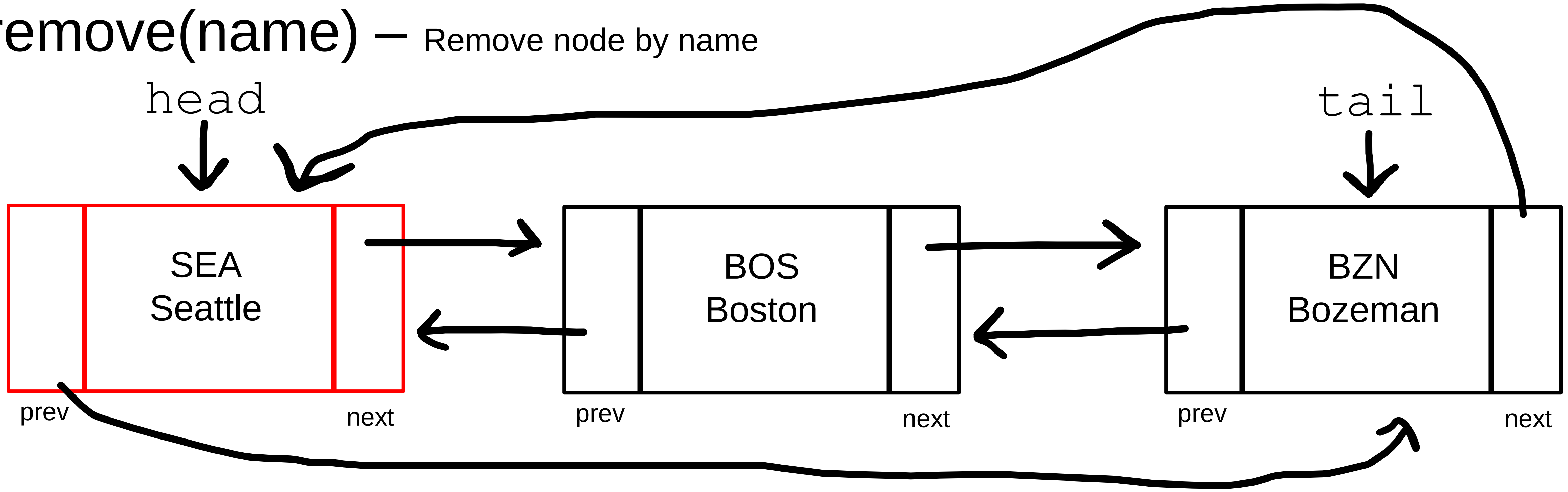
We don't need to change anything for case #4 and case #1

1. Find the s
   (curren
2. Get the n
   (`current.getNext()`)
3. Update `current`'s `next` value to the `newNode`

Node's `prev`
newNode
ewNode's `next`
value to be the nextNode

6. Update the `newNode`'s `prev` value to be the `current`

- remove(name) – Remove node by name

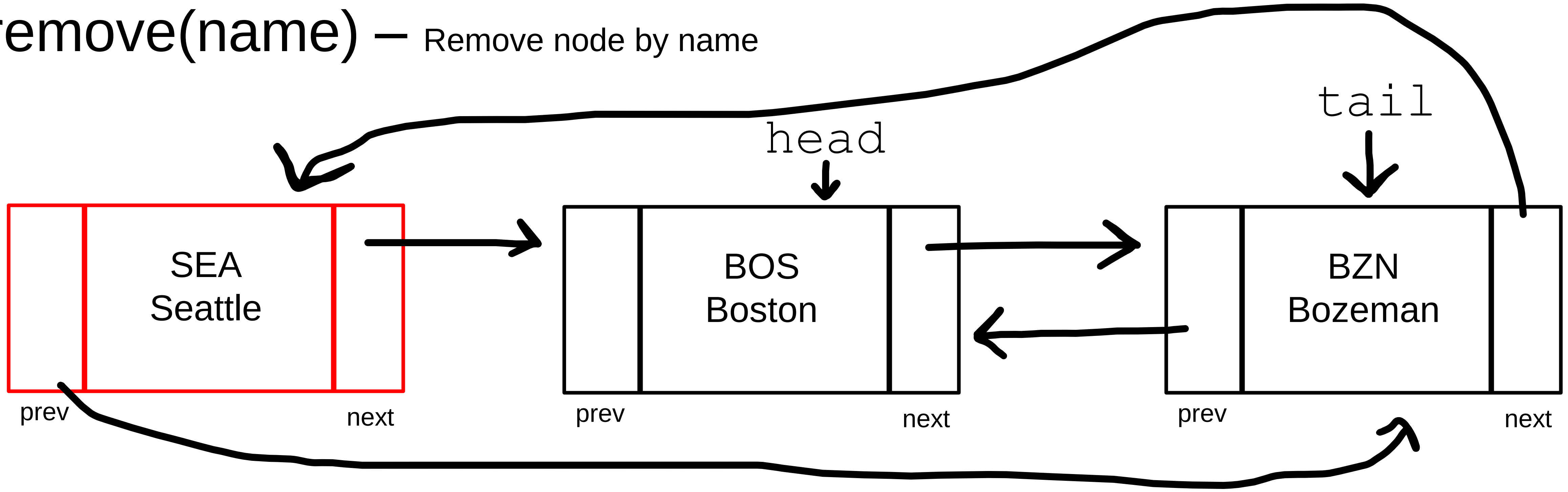


1. Traverse the Linked List and look for a match

`remove("SEA")`

What if the removed node is the head?

- remove(name) – Remove node by name



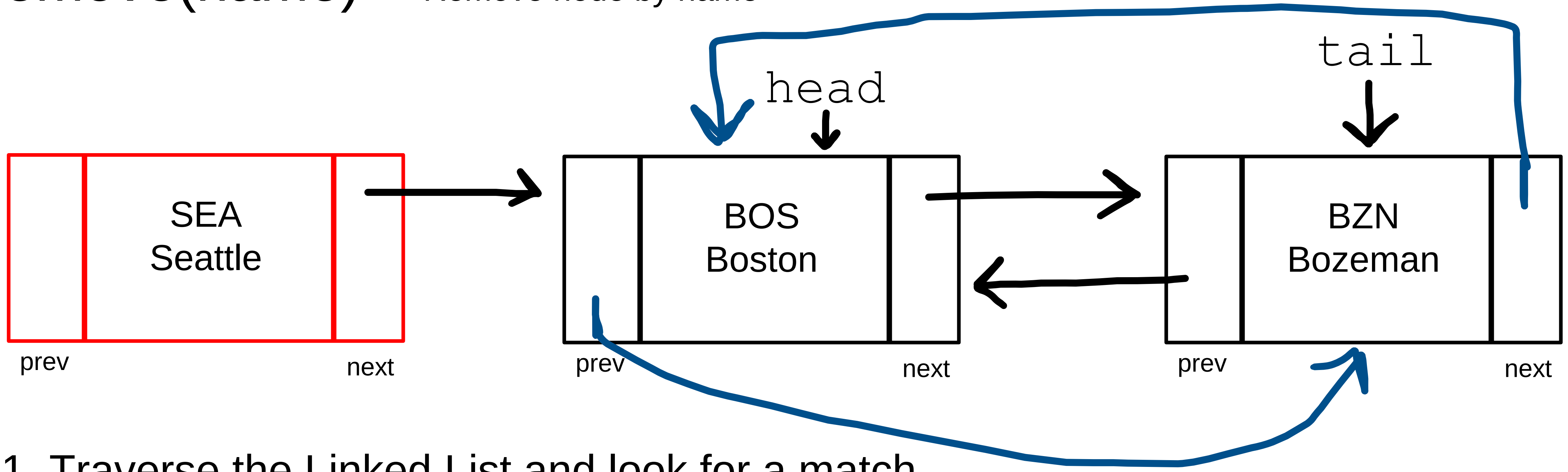
1. Traverse the Linked List and look for a match

`remove("SEA")`

What if the removed node is the head?

2. Update the `head` to be the next node
3. Update the new `head`'s `prev` value to be null

- remove(name) – Remove node by name



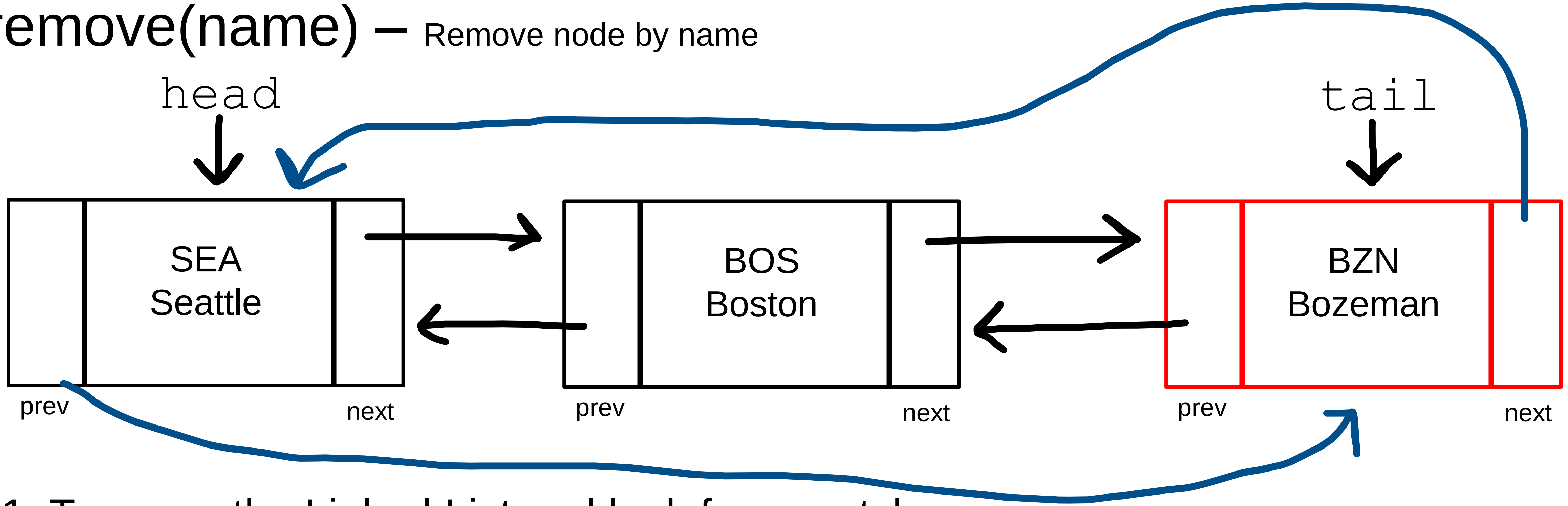
1. Traverse the Linked List and look for a match

`remove("SEA")`

What if the removed node is the head?

2. Update the `head` to be the next node
3. ~~Update the new `head`'s `prev` value to be null~~
4. NEW: Reconnect the `head` and `tail` nodes

- remove(name) – Remove node by name

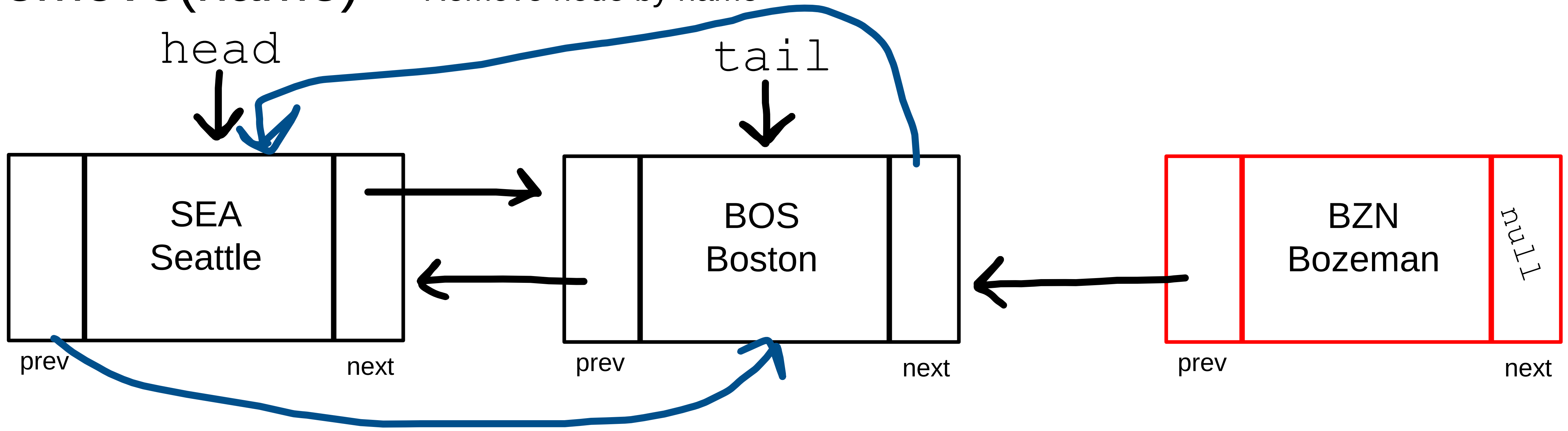


1. Traverse the Linked List and look for a match

`remove("BZN")`

What if the removed node is the tail?

- remove(name) – Remove node by name



1. Traverse the Linked List and look for a match

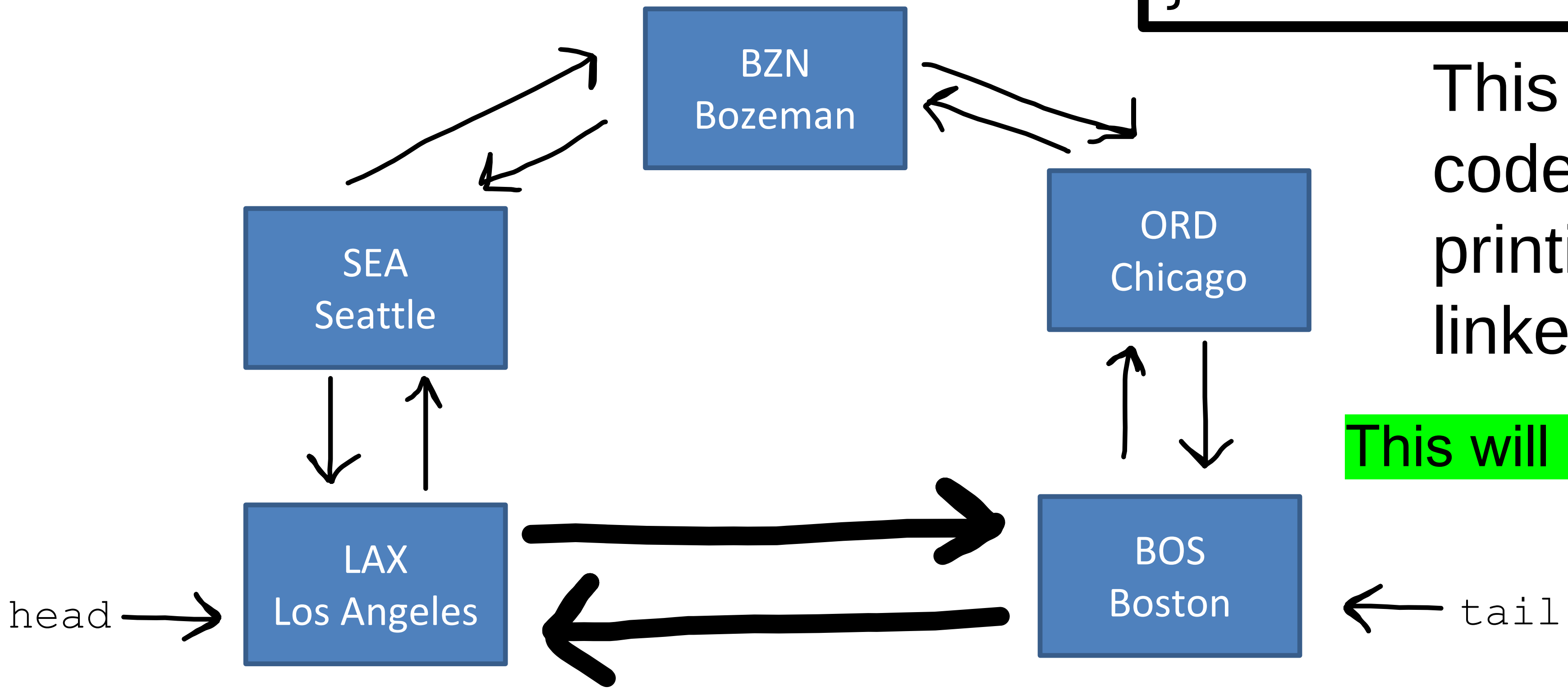
`remove("BZN")`

What if the removed node is the tail?

2. Update the `tail` to be the previous node
3. ~~Update the new `tail`'s `next` value to be `null`~~
4. NEW: Reconnect the `head` and `tail` nodes

Traversing a Circular Linked List

```
public void printList() {
   Node current = this.head;
   while(current != null) {
      current.printNode();
      current = current.getNext();
   }
}
```

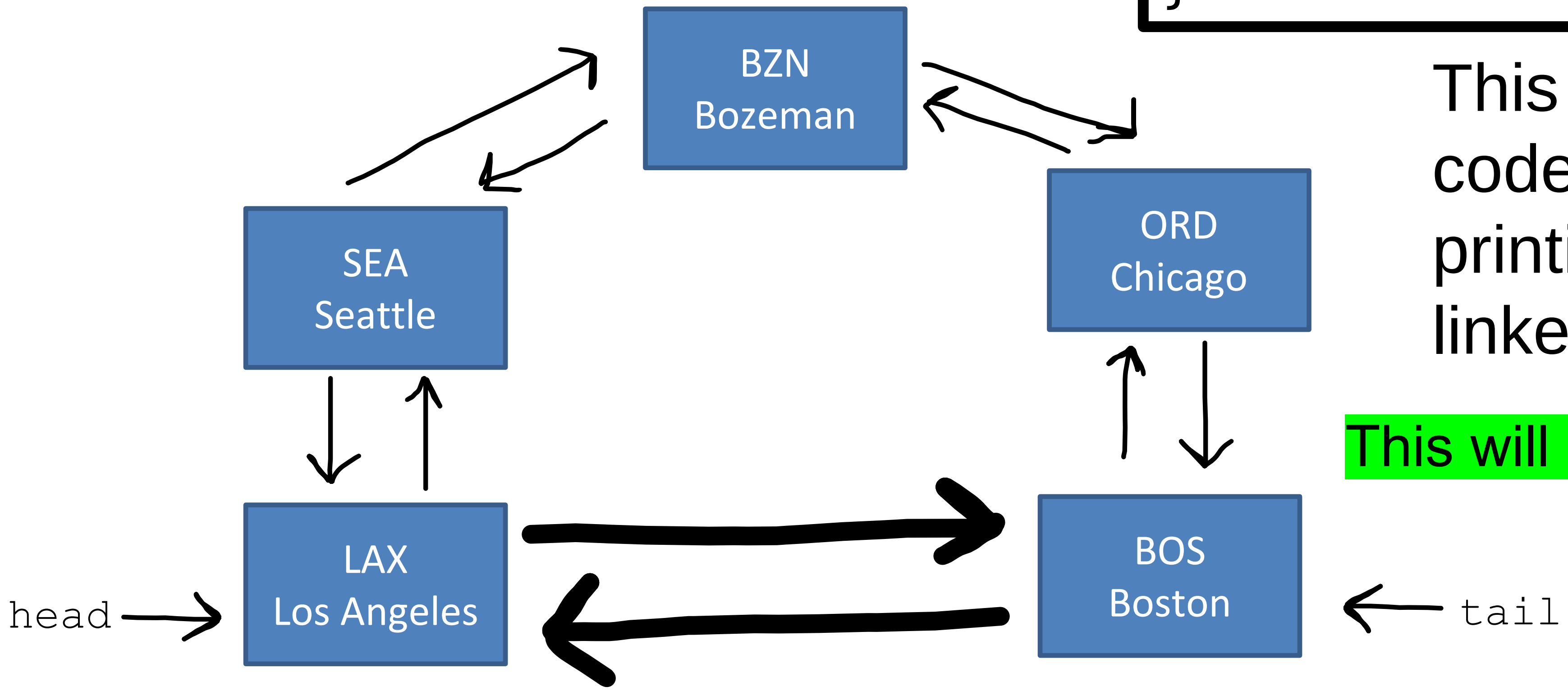


This was our previous code for traversing and printing out nodes in a linked list

This will no longer work because…

Traversing a Circular Linked List

```
public void printList() {
   Node current = this.head;
   while(current != null) {
      current.printNode();
      current = current.getNext();
   }
}
```

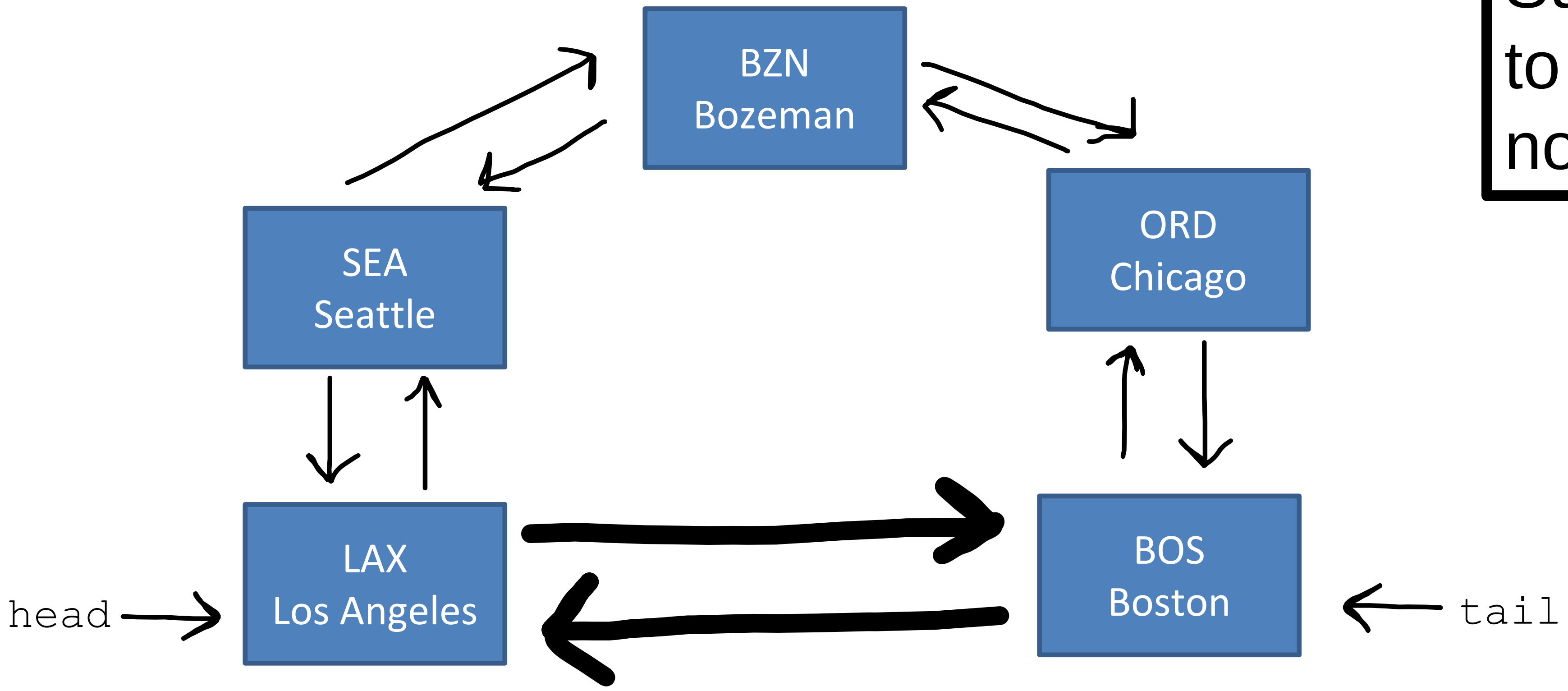


This was our previous code for traversing and printing out nodes in a linked list

This will no longer work because…

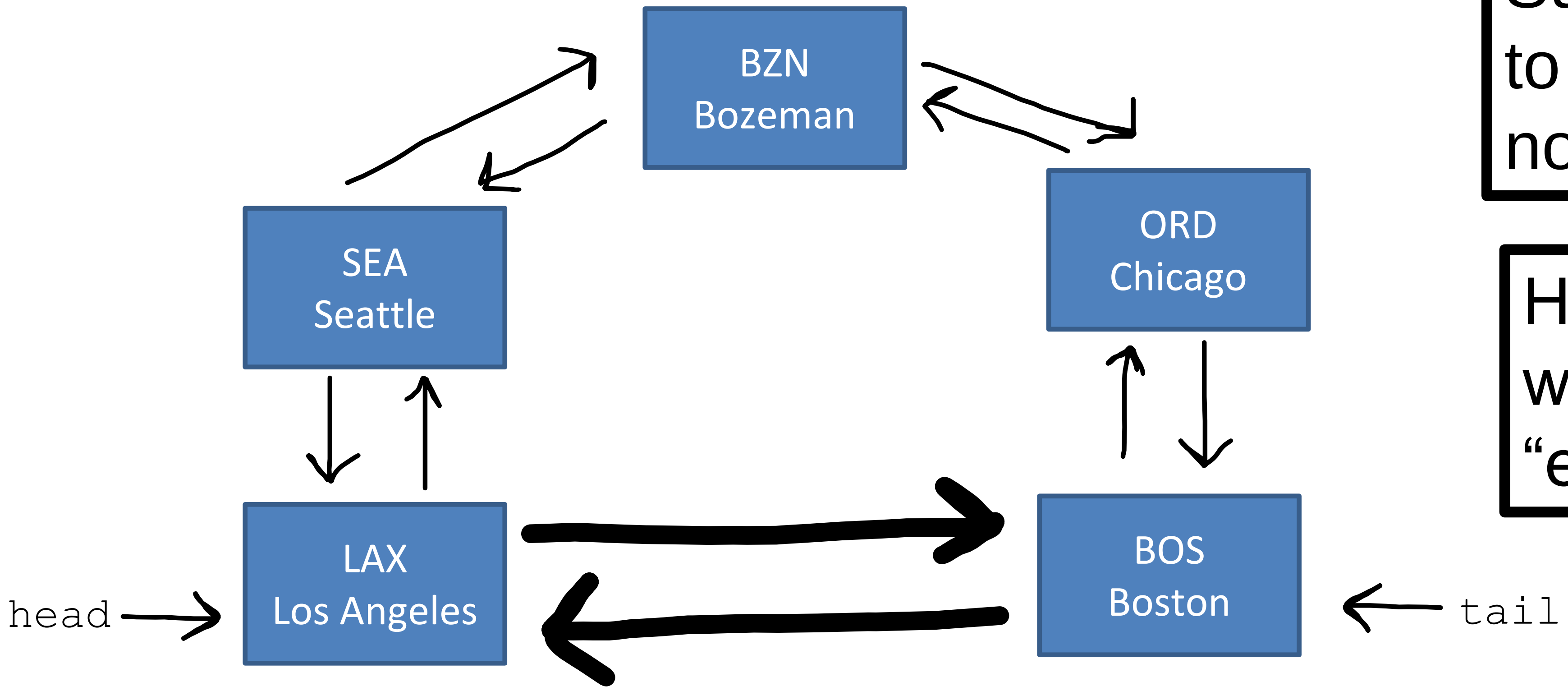
We will never reach `null`

Traversing a Circular Linked List



Suppose our goal is to print out each node only once

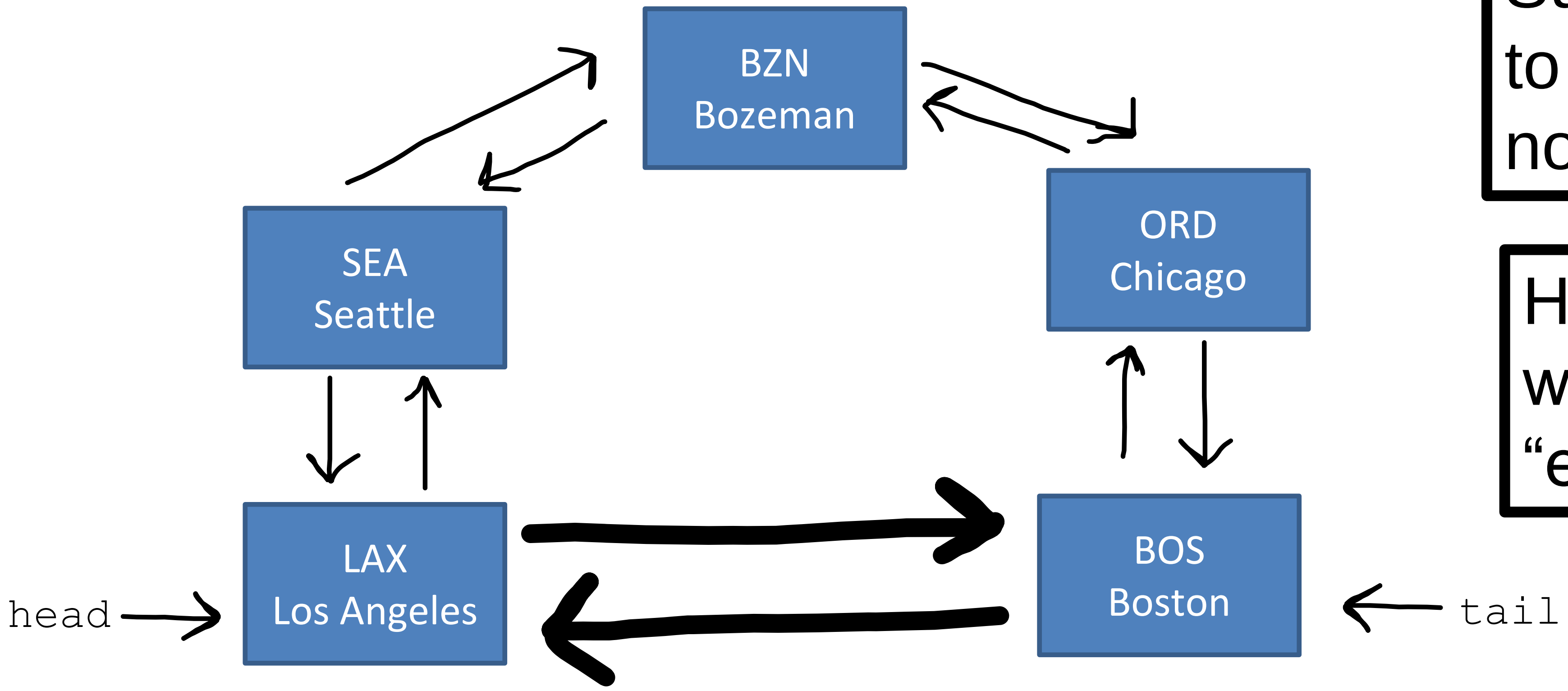
Traversing a Circular Linked List



Suppose our goal is to print out each node only once

How do we know that we've reached the "end" of the CLL ?

Traversing a Circular Linked List

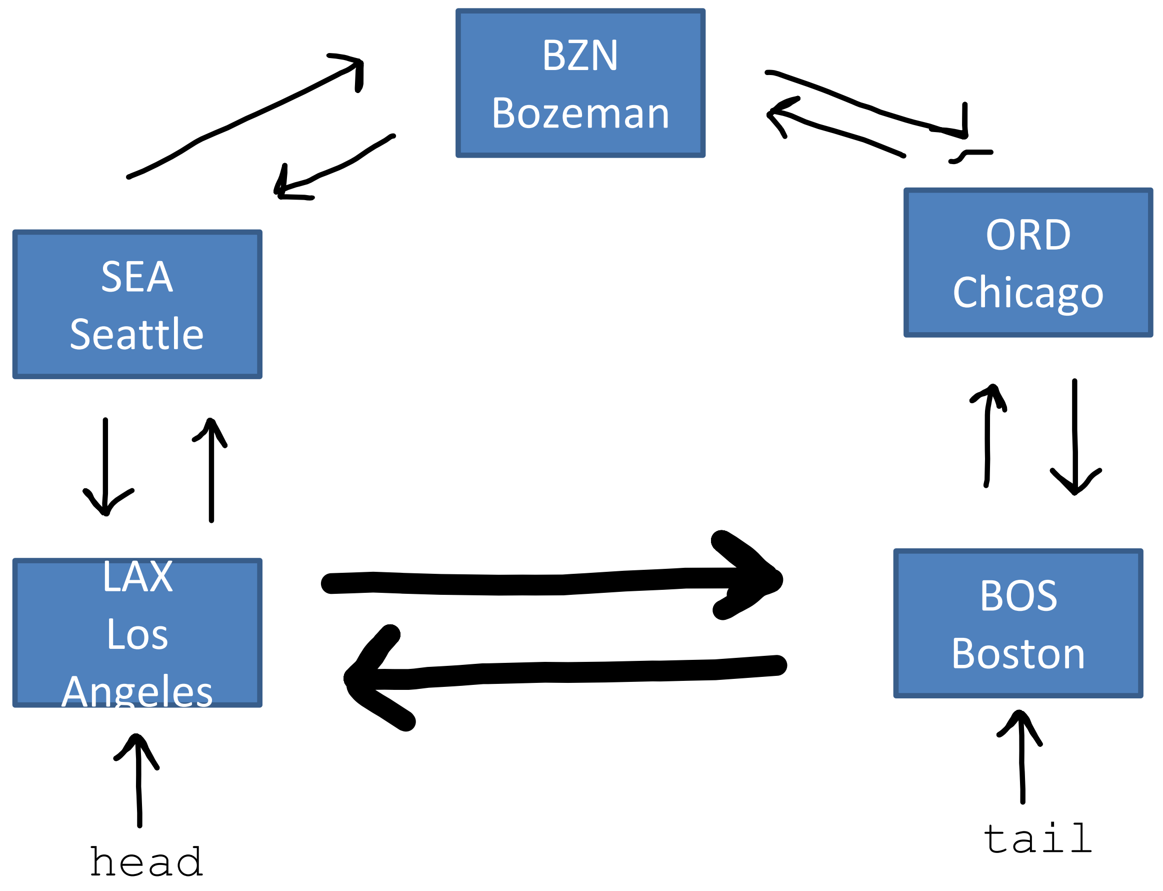


Suppose our goal is to print out each node only once

How do we know that we've reached the "end" of the CLL ?

If we start from the `head`, we should stop looping once we reach the `head` again

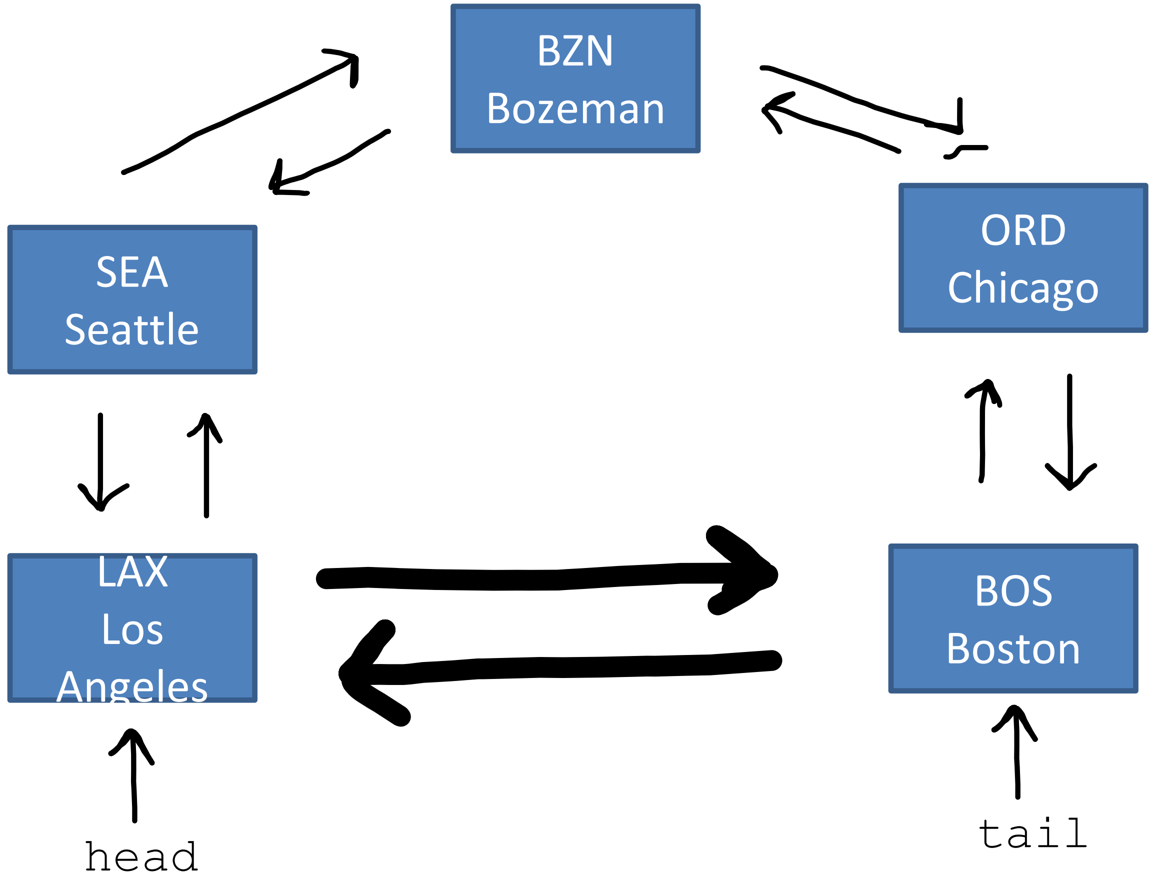
Traversing a Circular Linked List



If we start from the `head`, we should stop looping once we reach the `head` again

```
public void printLinkedList() {
    Node current = this.head.getNext();
    while(current != this.head) {
        current.printNode();
        current = current.getNext();
    }
}
```

Traversing a Circular Linked List

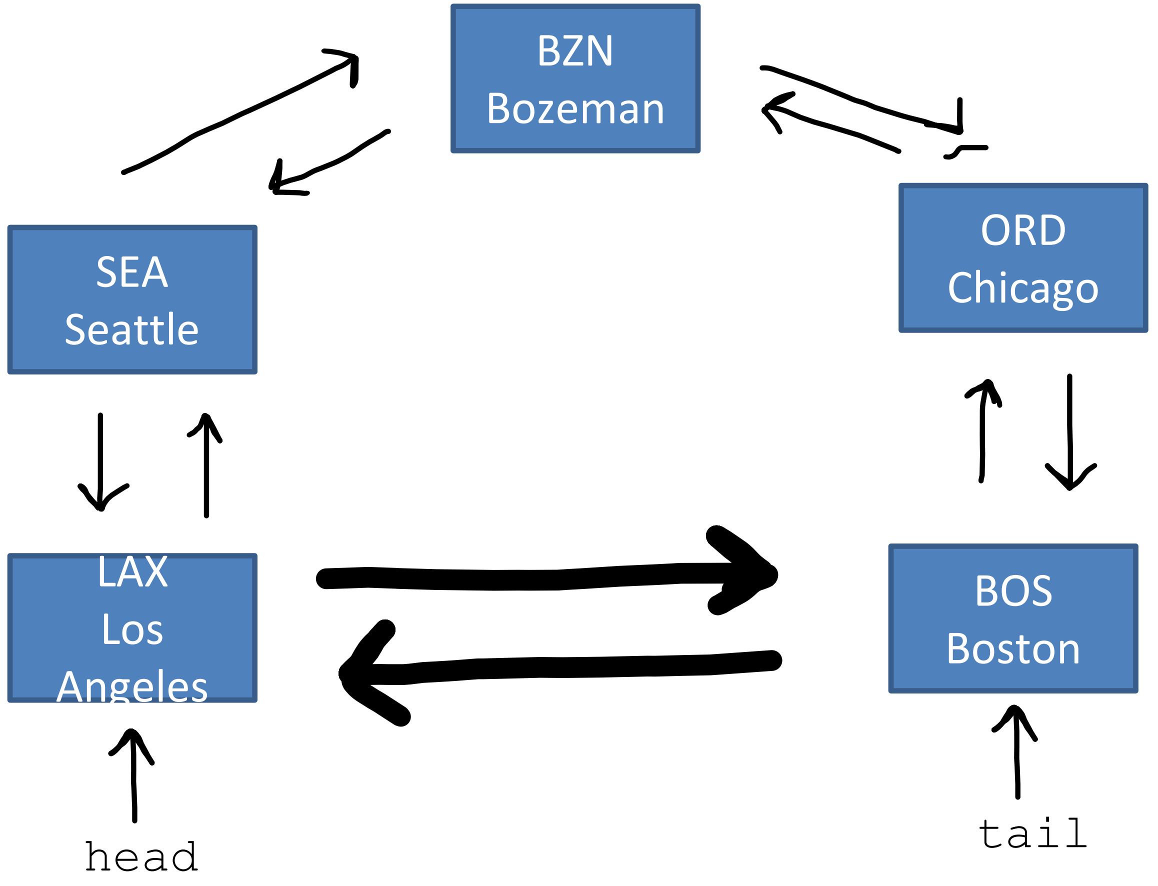


If we start from the `head`, we should stop looping once we reach the `head` again

```
public void printLinkedList() {
    Node current = this.head.getNext();
    while(current != this.head) {
        current.printNode();
        current = current.getNext();
    }
}
```

This **won't** work because…

Traversing a Circular Linked List



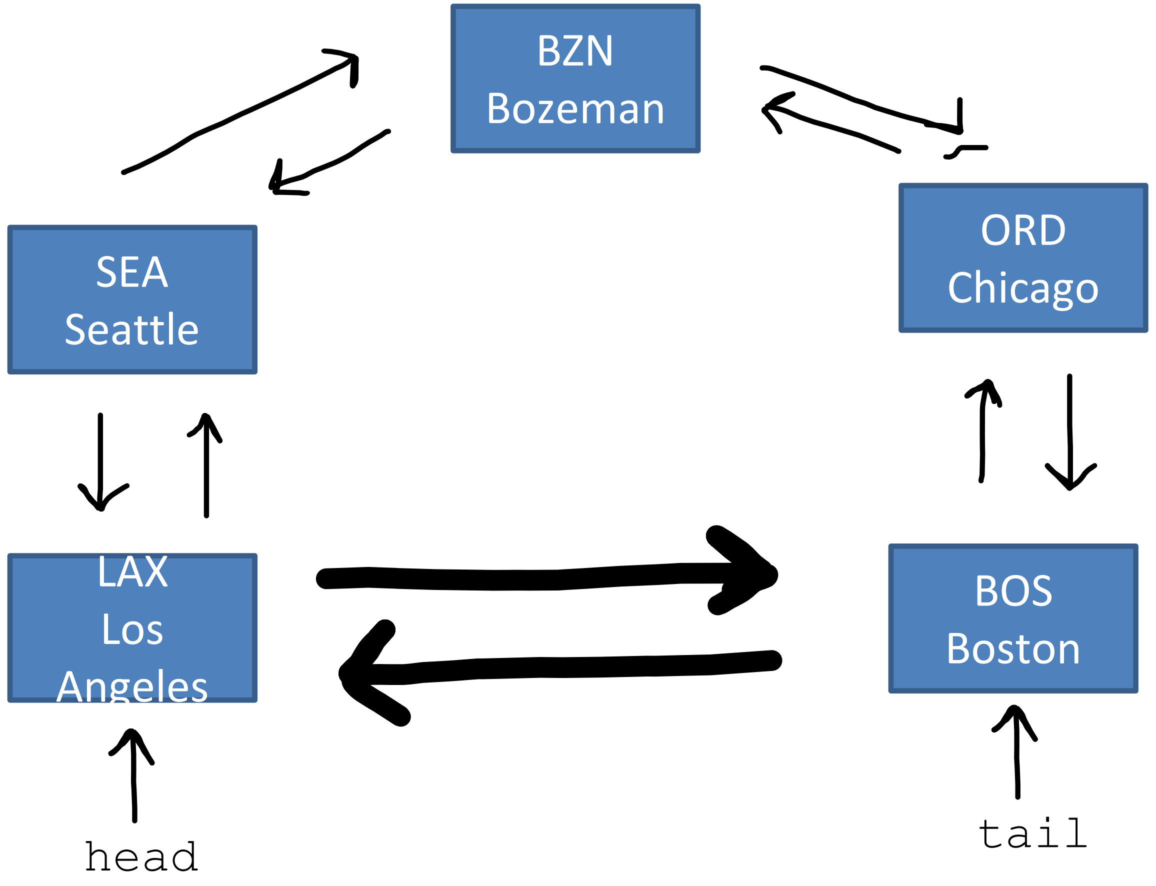
If we start from the `head`, we should stop looping once we reach the `head` again

```java
public void printLinkedList() {
    Node current = this.head.getNext();
    while(current != this.head) {
        current.printNode();
        current = current.getNext();
    }
}
```

This **won't** work because… The `head` node will never be printed out

Traversing a Circular Linked List

If we start from the `head`, we should stop looping once we reach the `head` again



```java
public void printLinkedList() {
    Node current = this.head;
    do {
        current.printNode();
        current = current.getNext();
    }
    while(current != this.head);
}
```

A **do/while** loop executed the body of the loop, and then checks the looping condition