CSCI 132:
Basic Data Structures and Algorithms

Linked Lists (Part 2)

Doubly Linked List

Reese Pearsall

Spring 2024

Announcements

Program 2 (Circular Linked Lists)

- We will try to talk about it on Friday

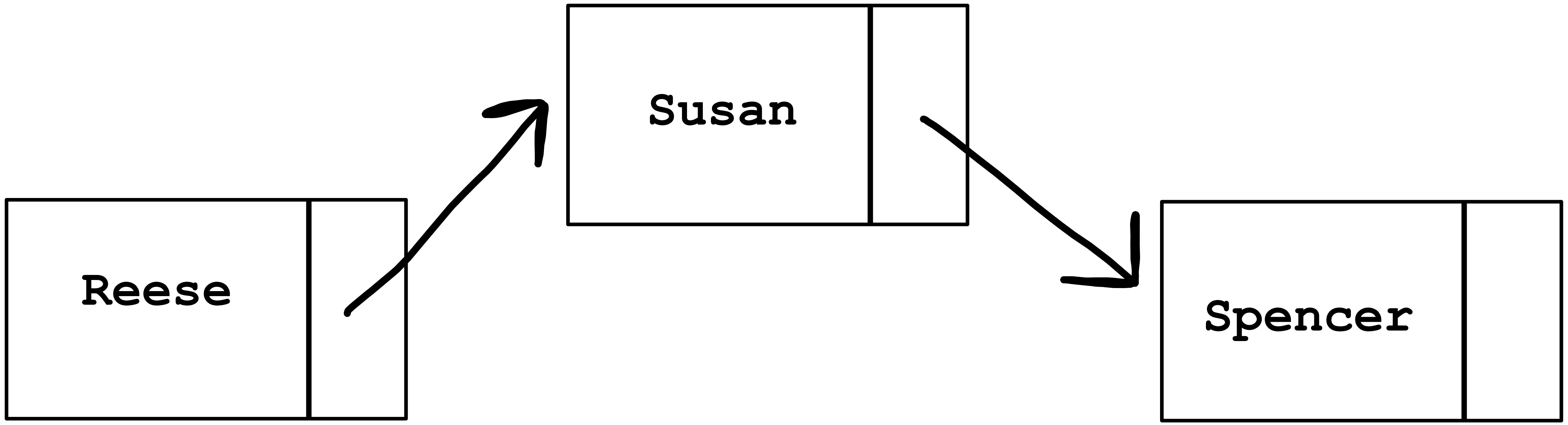
Next week we are covering some important stuff ☺

- *(Not a good week to ignore the class)*

Linked List data structures be like:
I know a guy who knows a guy

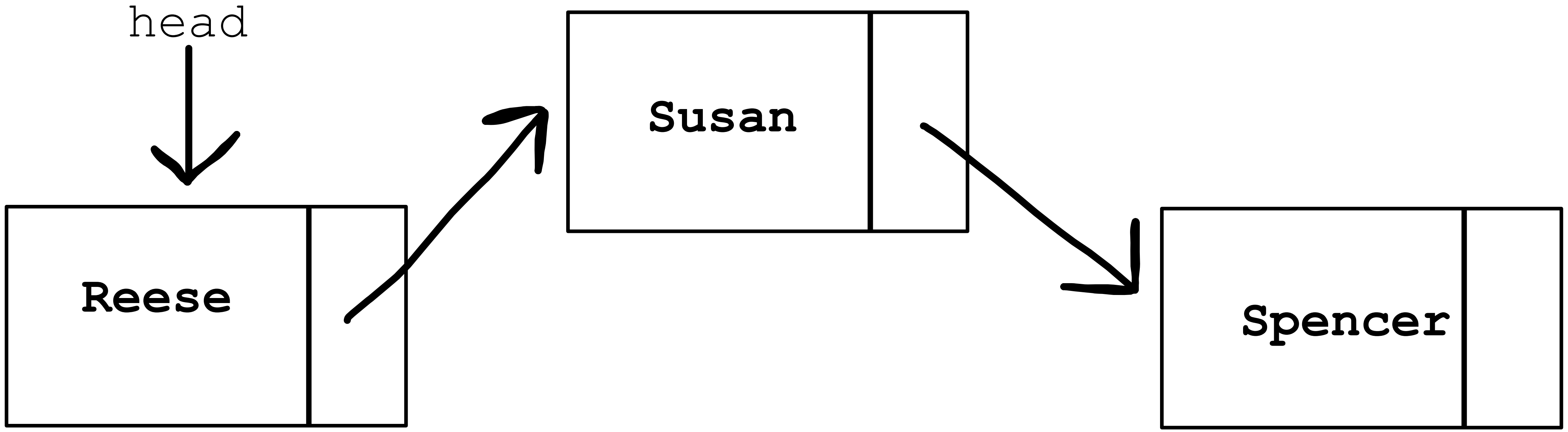
when you ask stack overflow how to get the first element in a linked list
You should have gone for the head

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 **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**

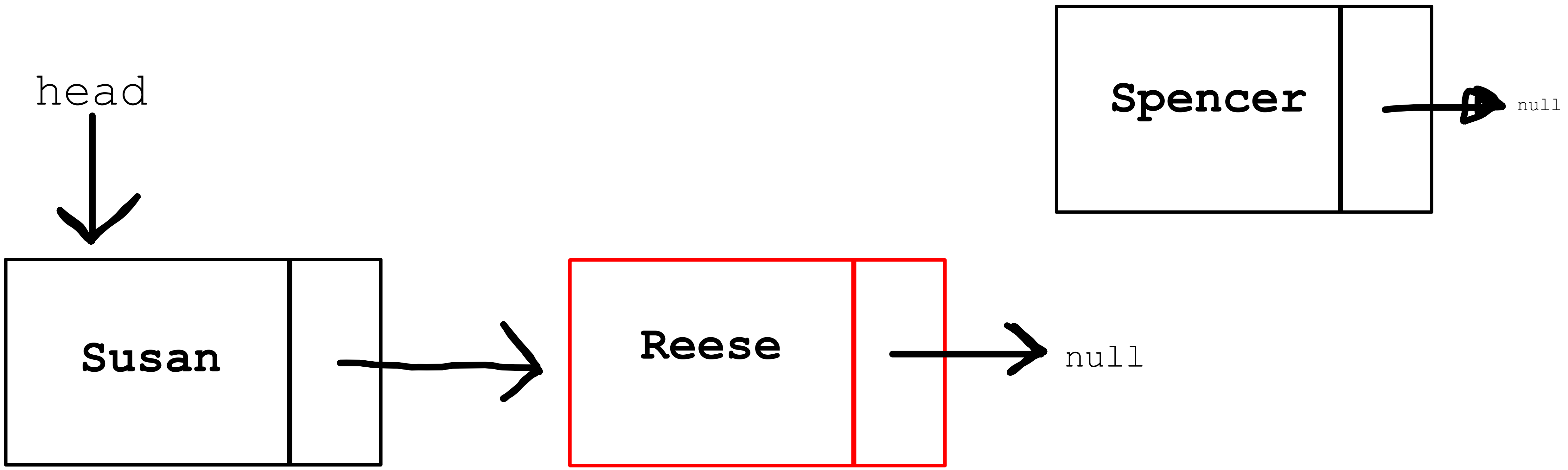
A Linked List also has a pointer to the start of the Linked List (`head`)

Linked List Methods

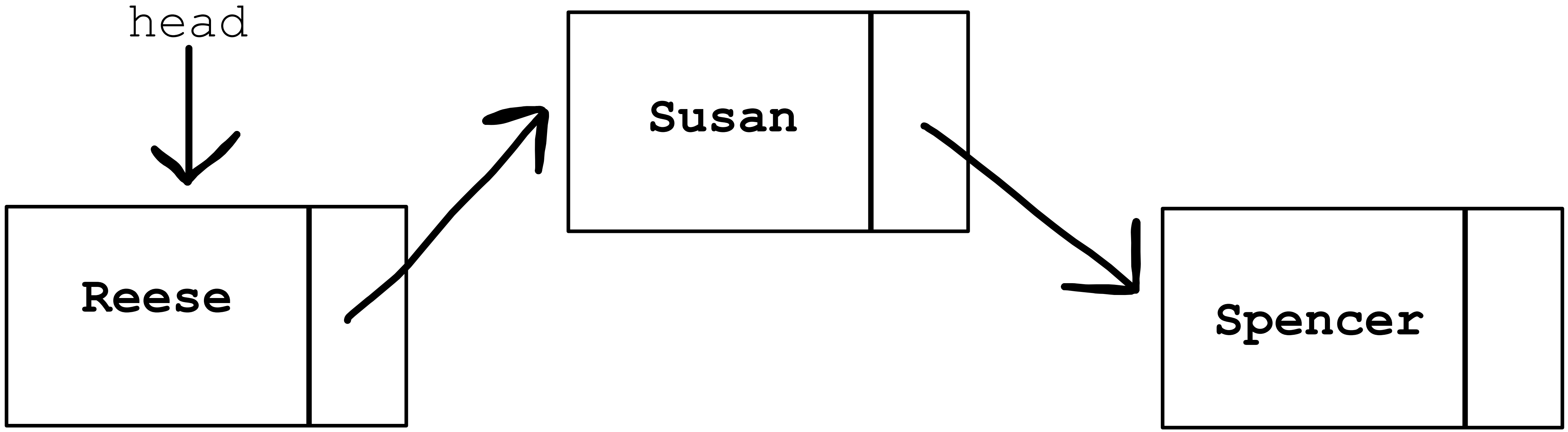
- `removeLast()` – removes last node of LL

```
public void removeLast() {
        if(size == 0) {
                return;
        }
        else if(size == 1) {
                head = null;
        }
        else {
                Node current = head;
                while(current.getNext().getNext() != null) {
                        current = current.getNext();
                }
                current.setNext(null);
        }
}
```

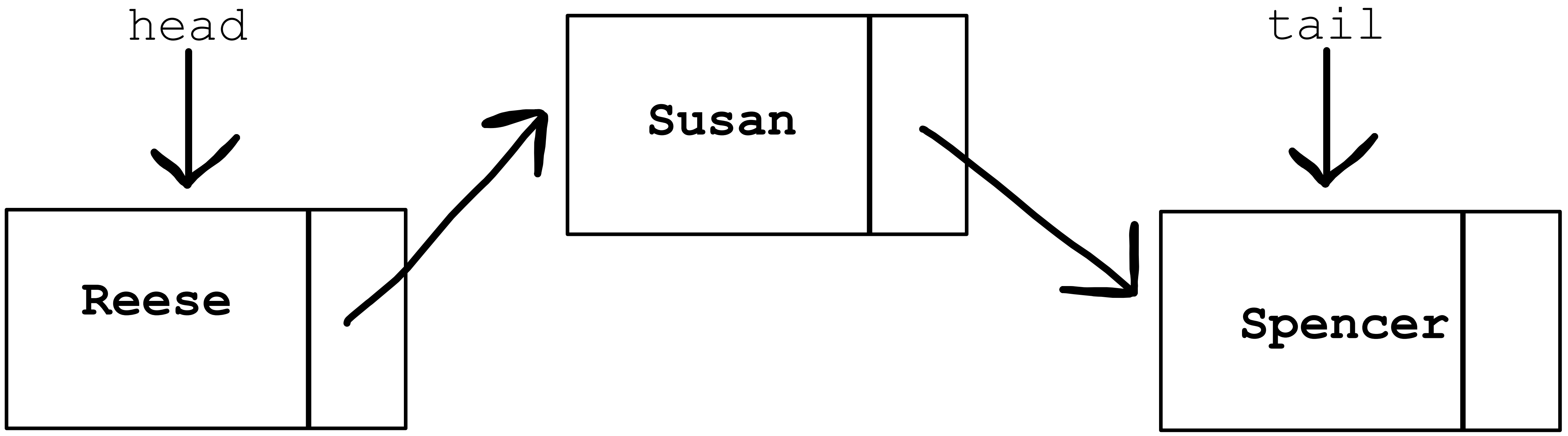
1. Find the second to last node
2. Set that node's `next` value to `null`



A **Singly Linked List** only keeps track of the next node

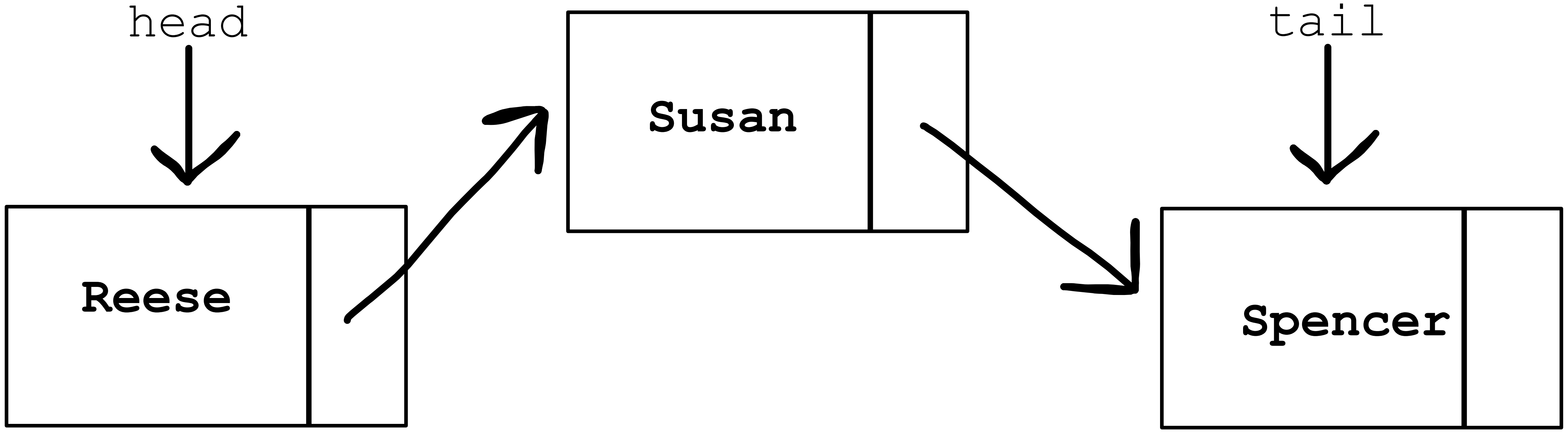


A **Singly Linked List** only keeps track of the next node



The `tail` of a linked list is a pointer to the last node

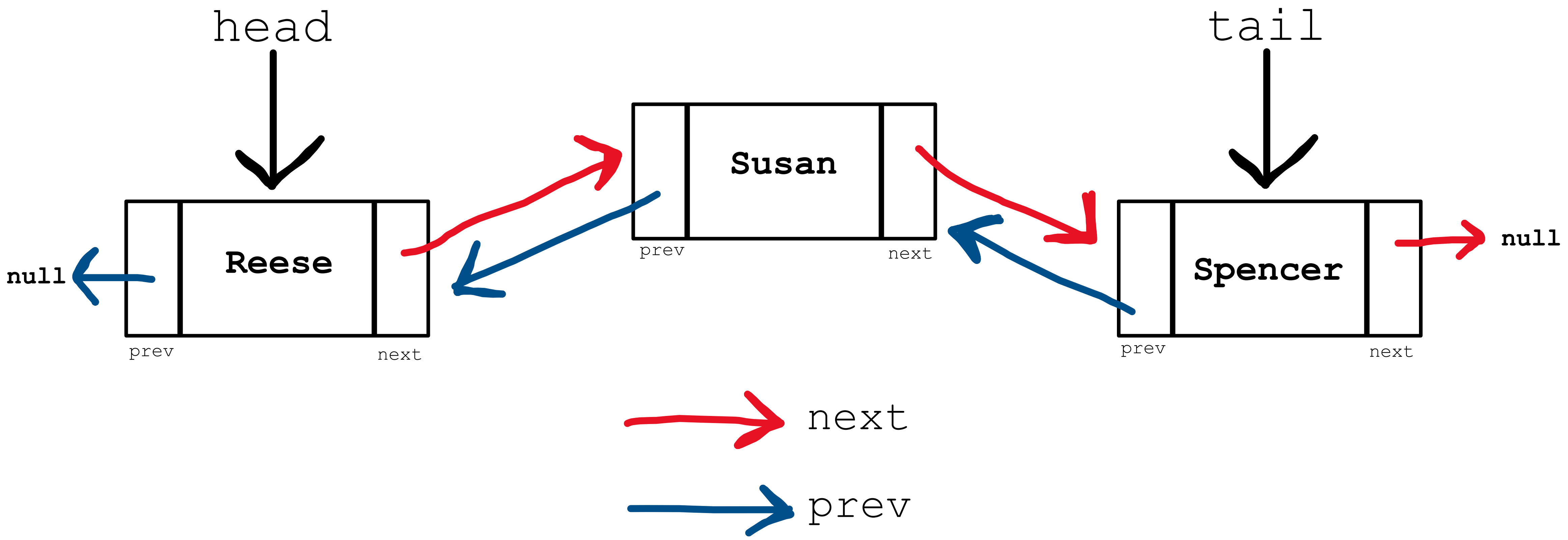
A **Singly Linked List** only keeps track of the next node



The `tail` of a linked list is a pointer to the last node

This makes adding to/removing from the end of a linked list easier

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

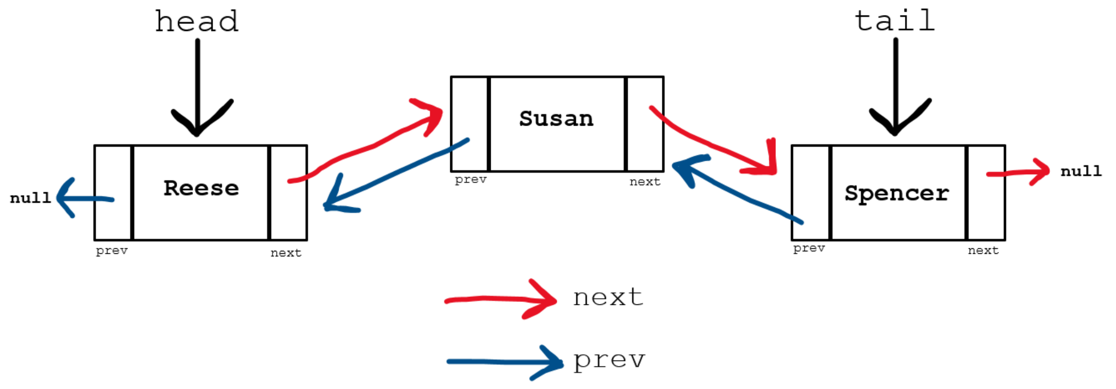


A **Doubly Linked List** keeps track of the next node and the previous node

Doubly Linked List Methods

- insert(newNode, N) – Insert new node at spot N
- remove(name) – Remove node by name
- remove(N) – Remove node by Spot #

- printReverse() – Prints LL in reverse order



Java File I/O

Let’s read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

Java File I/O

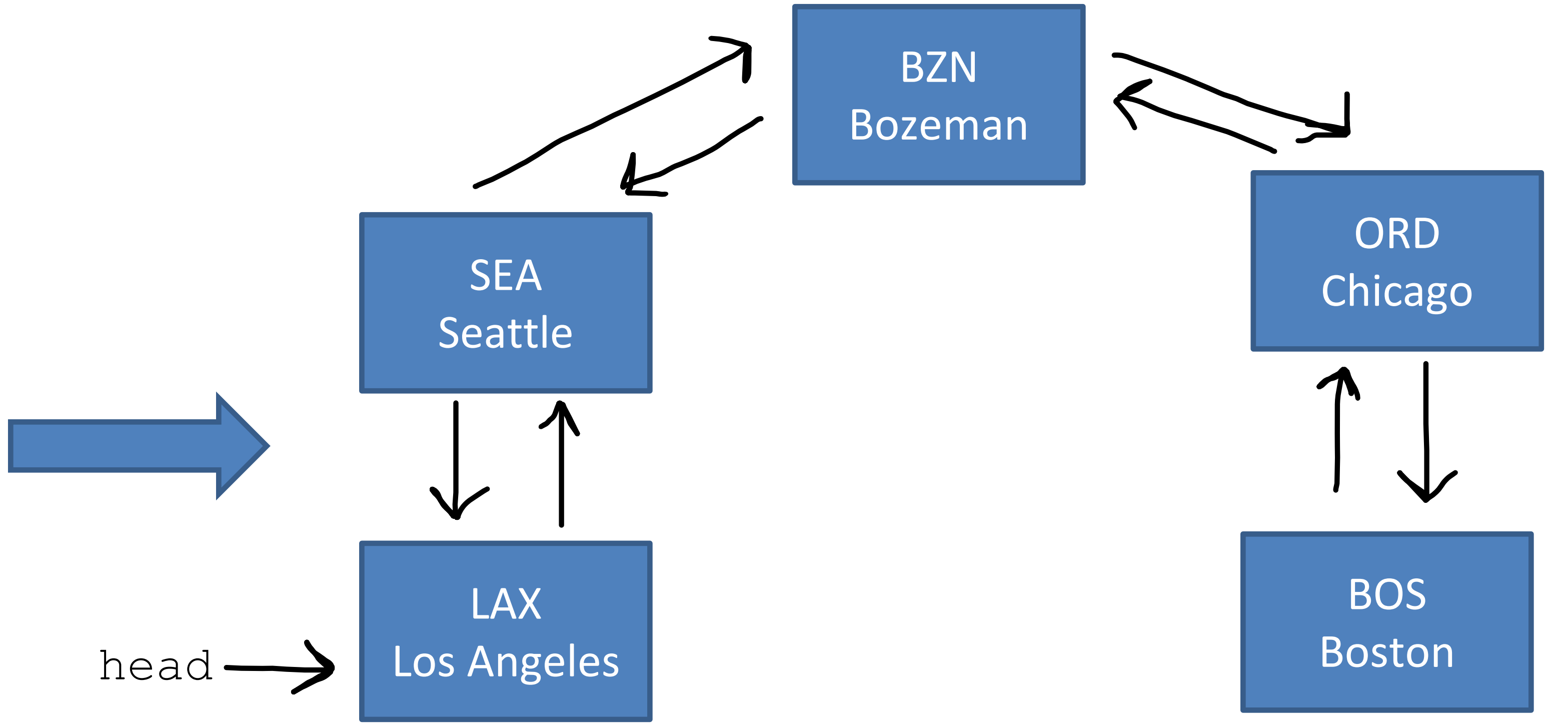
Let's read in node information **from a file**

(We can also do it with the Scanner)

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```



Java File I/O

Let's read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles

SEA,Seattle

BZN,Bozeman

ORD,Chicago

BOS,Boston
```

1. Iterate through each line of the file

```java
BufferedReader br = new BufferedReader(new FileReader(filename));
String line = "";
while( (line=br.readLine()) != null){


}
```

Java File I/O

Let's read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

1. Iterate through each line of the file

```
BufferedReader br = new BufferedReader(new FileReader(filename));
String line = "";
while( (line=br.readLine()) != null){


}
```

"Iterate through each line in the file until we reach the end"

Java File I/O

Let’s read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

1. Iterate through each line of the file
2. Parse each line using `.split()`

```
while( (line=br.readLine()) != null){
    String[] vals = line.split(",");
```

“LAX,Los Angeles” → vals =

0	1
LAX	**Los Angeles**

`.split(“,”)` will “split” the string everything it sees a comma, returns an array of the splitted string

Java File I/O

Let's read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

1. Iterate through each line of the file
2. Parse each line using `.split()`

```
while( (line=br.readLine()) != null){

    String[] vals = line.split(",");
```

"LAX,Los Angeles" → vals =

0	1
LAX	**Los Angeles**

"SEA,Seattle" → vals =

0	1
SEA	**Seattle**

Java File I/O

Let’s read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

1. Iterate through each line of the file
2. Parse each line using `.split()`
3. Create Node object using information from file

airports.txt

```
LAX,Los Angeles

SEA,Seattle

BZN,Bozeman

ORD,Chicago

BOS,Boston
```

```
1  while( (line=br.readLine()) != null){
       2  String[] vals = line.split(",");

          String code = vals[0];
      3   String location = vals[1];

          Node n = new Node(code, location);
          insert(n,size+1);
   }
```

Java File I/O

Let's read in node information **from a file**

There are tons of way to read from a file in Java. We will use the BufferedReader library

1. Iterate through each line of the file
2. Parse each line using `.split()`
3. Create Node object using information from file
4. Insert new node at the end of the linked list

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

```
1  while( (line=br.readLine()) != null){
2        String[] vals = line.split(",");

         String code = vals[0];
3        String location = vals[1];

         Node n = new Node(code, location);
4        insert(n,size+1);
   }
```

Java File I/O

```
while( (line=br.readLine()) != null){
        String[] vals = line.split(",");

        String code = vals[0];
        String location = vals[1];

        Node n = new Node(code, location);
        insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles

SEA,Seattle

BZN,Bozeman

ORD,Chicago

BOS,Boston
```

Java File I/O

```
while( (line=br.readLine()) != null){
        String[] vals = line.split(",");

        String code = vals[0];
        String location = vals[1];

        Node n = new Node(code, location);
        insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles

SEA,Seattle

BZN,Bozeman

ORD,Chicago

BOS,Boston
```

line = "LAX,Los Angeles"

Java File I/O

```java
while( (line=br.readLine()) != null){
        String[] vals = line.split(",");

        String code = vals[0];
        String location = vals[1];

        Node n = new Node(code, location);
        insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

line = ”LAX,Los Angeles”

vals[] =

0	1
LAX	**Los Angeles**

Java File I/O

```
while( (line=br.readLine()) != null){
        String[] vals = line.split(",");

        String code = vals[0];
        String location = vals[1];

        Node n = new Node(code, location);
        insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

line = "LAX,Los Angeles"

vals[] =

0	1
LAX	**Los Angeles**

```
code     = "LAX"
location = "Los Angeles"
```

Java File I/O

```java
while( (line=br.readLine()) != null){
      String[] vals = line.split(",");

      String code = vals[0];
      String location = vals[1];

      Node n = new Node(code, location);
      insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

line = ”LAX,Los Angeles”

vals[] =

0	1
LAX	**Los Angeles**

code = “LAX”
location = “Los Angeles”

Java File I/O

```java
while( (line=br.readLine()) != null){
        String[] vals = line.split(",");

        String code = vals[0];
        String location = vals[1];

        Node n = new Node(code, location);
        insert(n,size+1);
}
```

airports.txt

```
LAX,Los Angeles
SEA,Seattle
BZN,Bozeman
ORD,Chicago
BOS,Boston
```

line = "LAX,Los Angeles"

vals[] =

0	1
LAX	**Los Angeles**

code = "LAX"
location = "Los Angeles"

n = LAX Los Angeles

```
BufferedReader br = new BufferedReader(new FileReader(filename));
String line = "";
while( (line=br.readLine()) != null){
      String[] vals = line.split(",");

      String code = vals[0];
      String location = vals[1];

      Node n = new Node(code, location);
      insert(n,size+1);
}
```

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

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

Case 1: The Linked List is Empty

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

Case 1: The Linked List is Empty

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

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

Case 1: The Linked List is Empty

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

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 1: The Linked List is Empty

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

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

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 1: The Linked List is Empty

How do we know if the linked list is empty?

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

Case 1: The Linked List is Empty

How do we know if the linked list is empty?

If the `head` and `tail` are `null`

If the `size` is 0

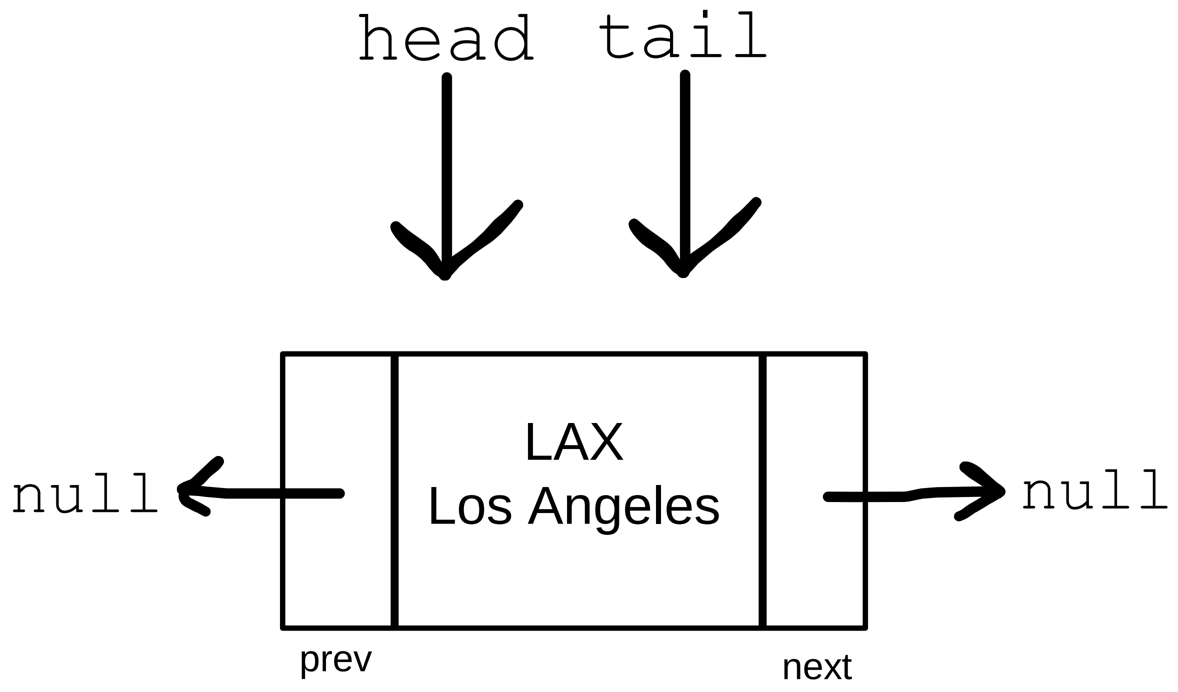
- **`insert(newNode, N)`** **—** Insert new node (`newNode`) at spot **N**

Case 1: The Linked List is Empty

???

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

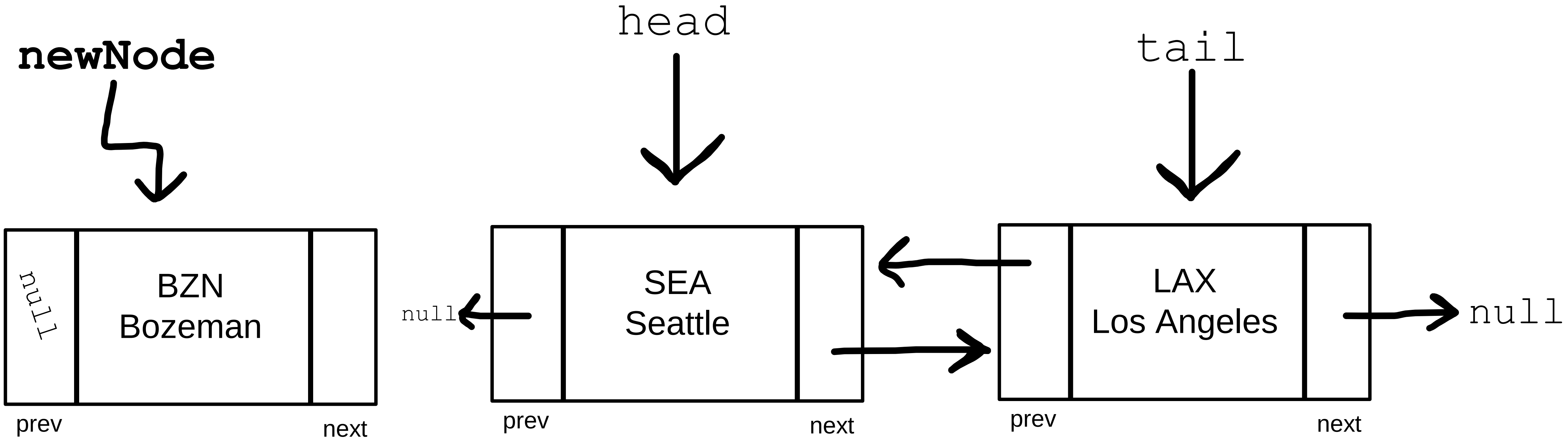
Case 1: The Linked List is Empty



Set the `tail` and `head` to be the `newNode`

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

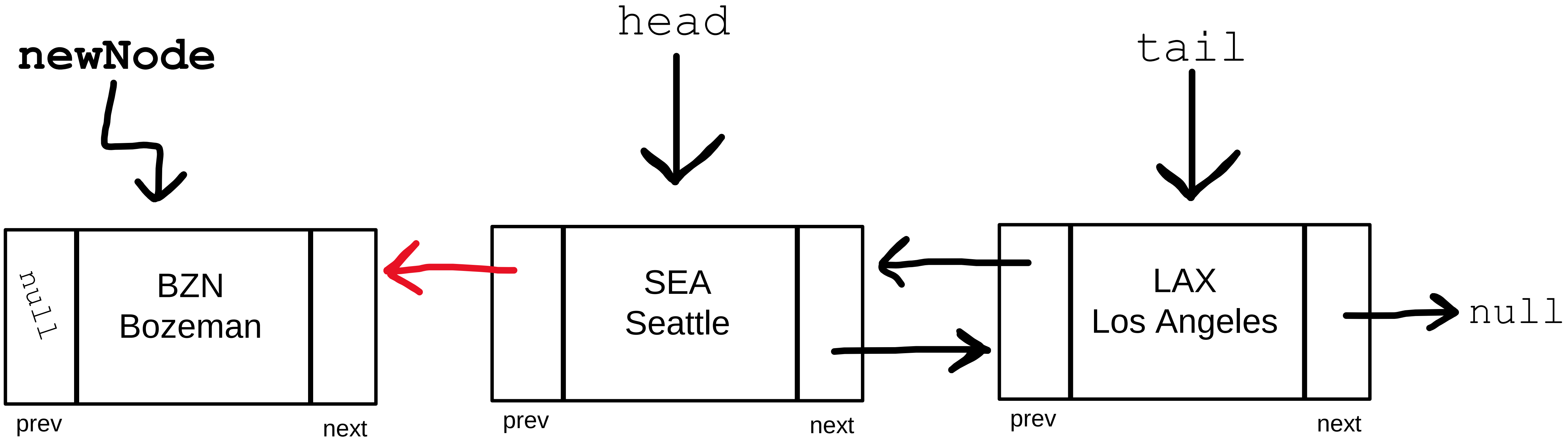
Case 2: The user is inserting a node at the very beginning (N = 1)



???

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

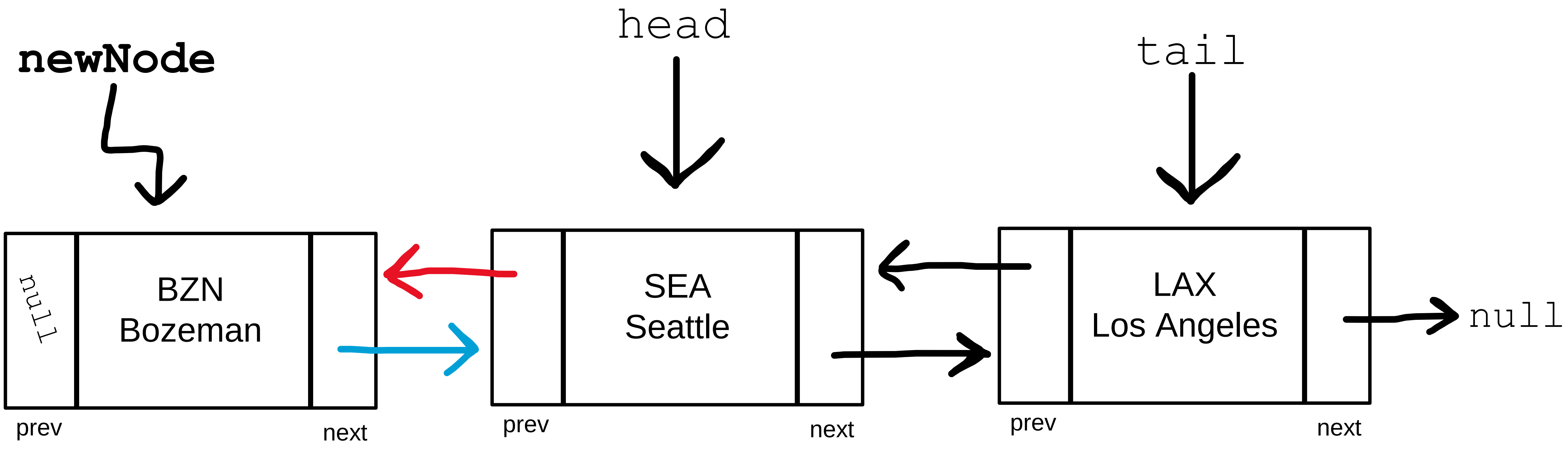
Case 2: The user is inserting a node at the very beginning (N = 1)



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

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

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

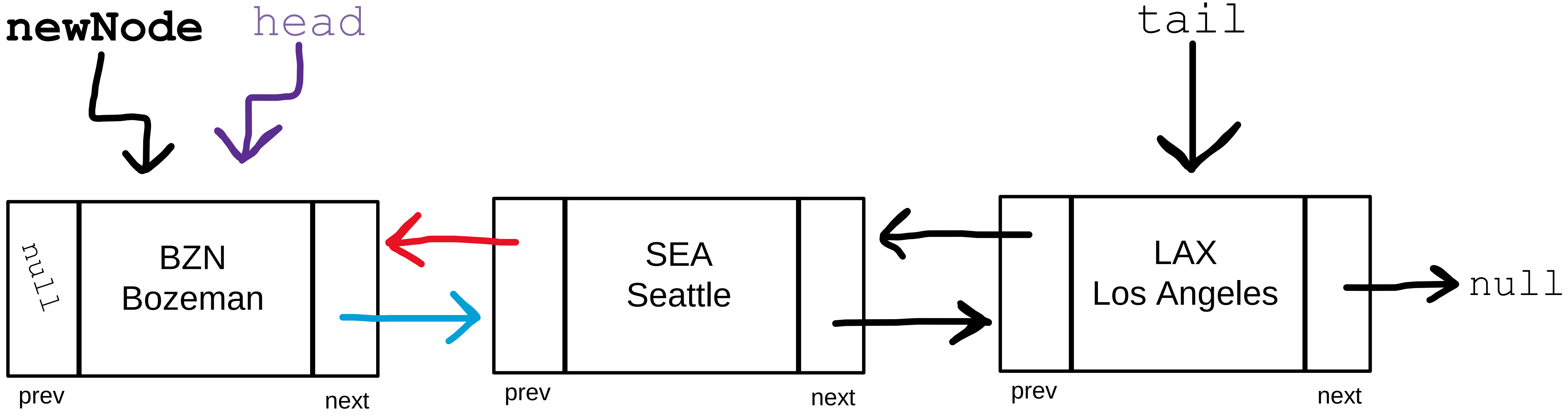


Update the `head` node `prev` value to `newNode`

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

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

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



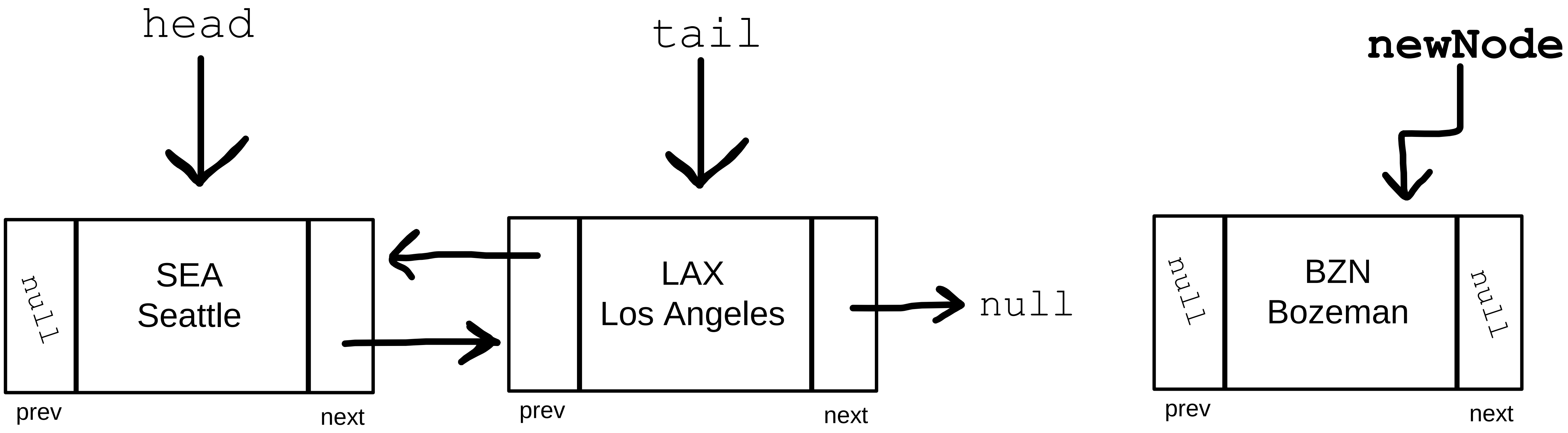
Update the `head` node `prev` value to `newNode`

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

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

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

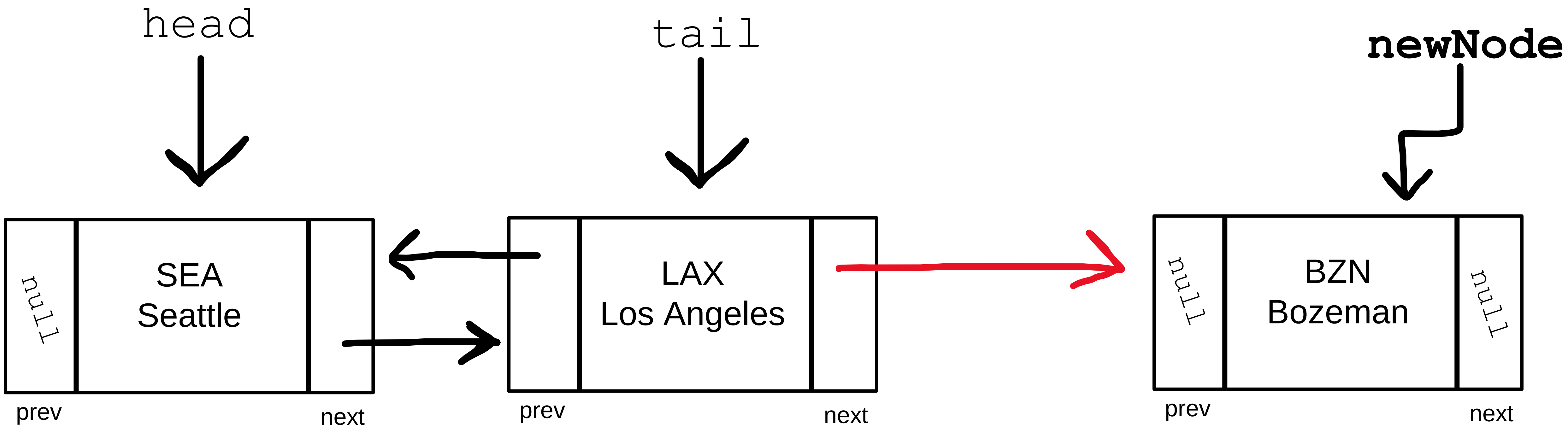
Case 3: The user is inserting a node at the very end (N = `getSize()` + 1)



`insert(newNode, 3)`

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

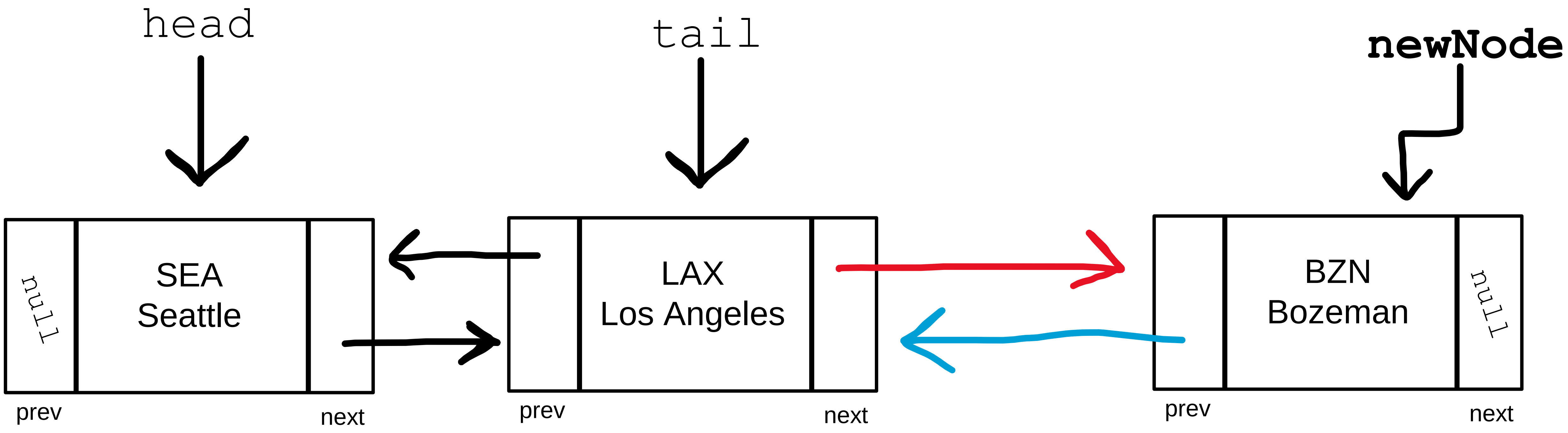
Case 3: The user is inserting a node at the very end (N = `getSize()` + 1)



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

- **`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)

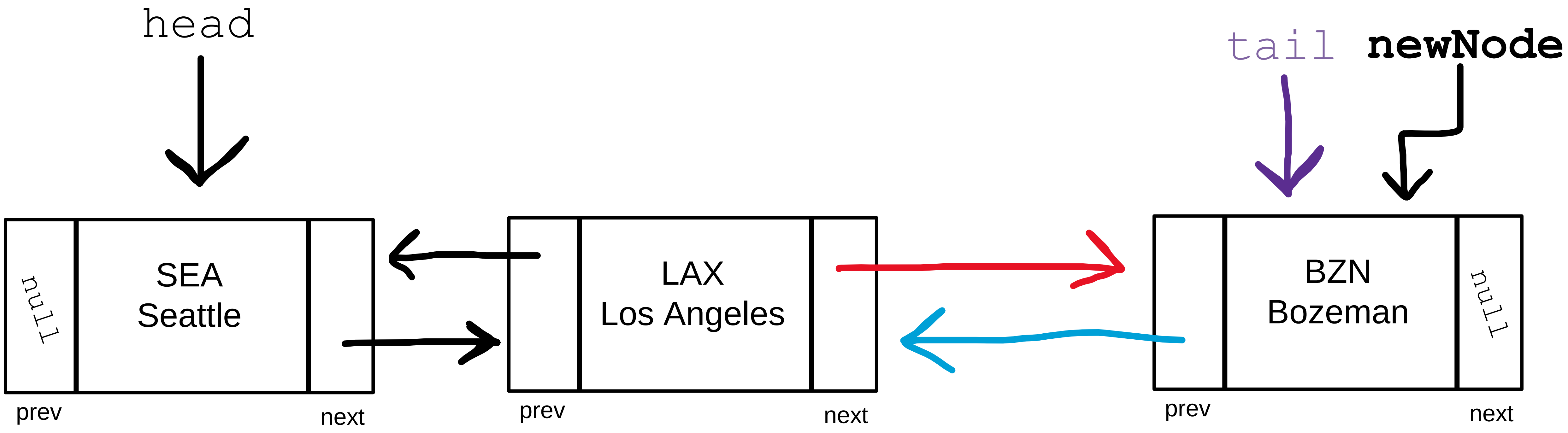


Update the `tail` node `next` value to `newNode`

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

- **`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)



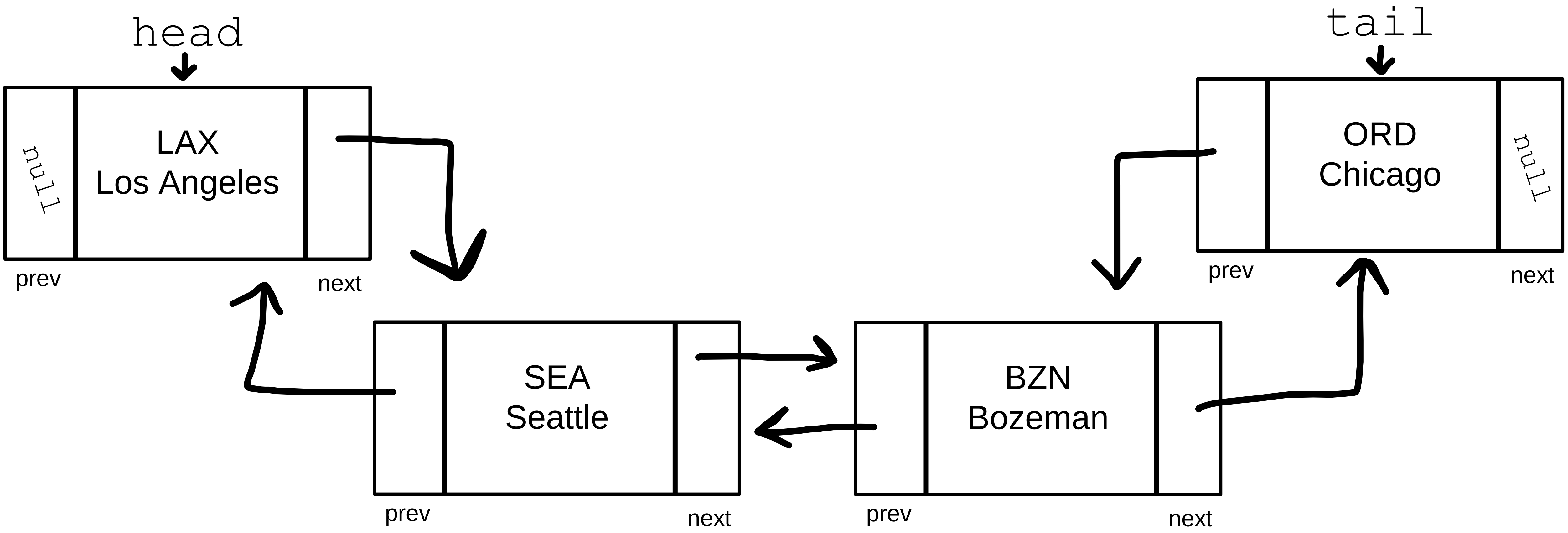
Update the `tail` node `next` value to `newNode`

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

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

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

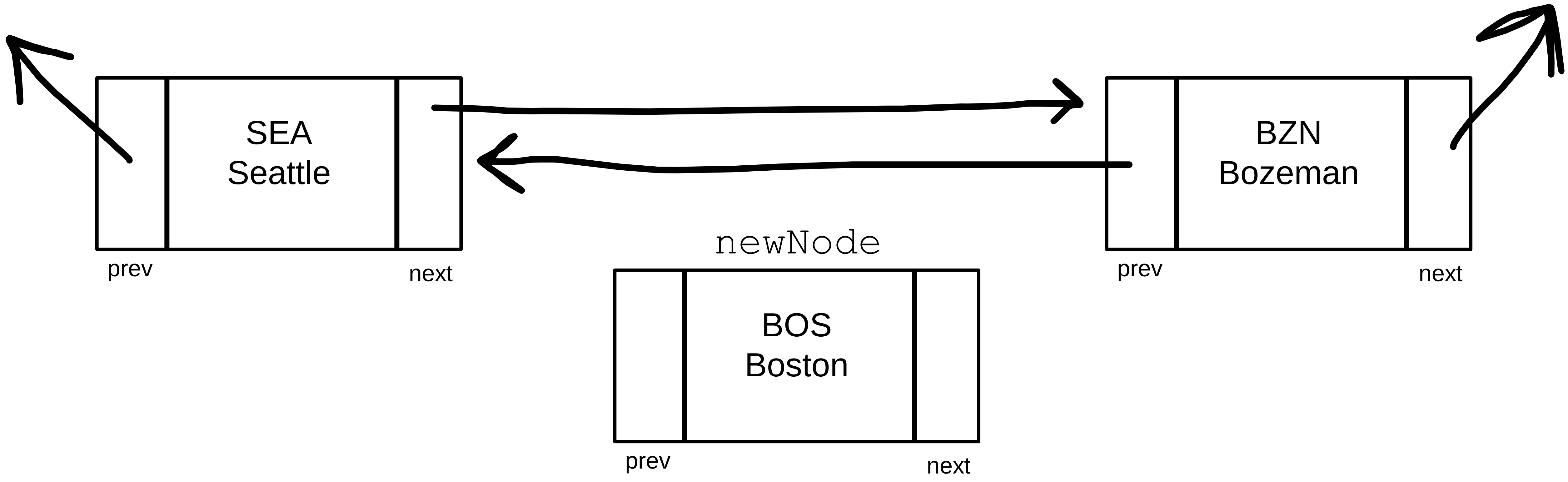
Case 4: The user is inserting a node somewhere in the middle of the LL



`insert(newNode, 3)`

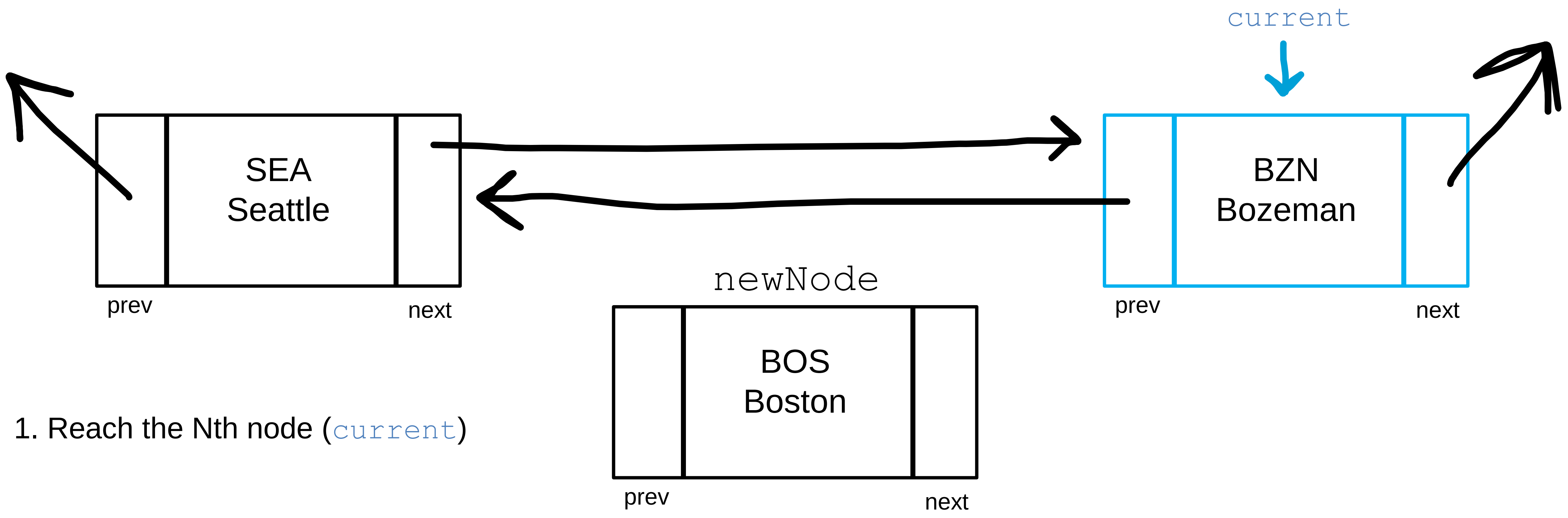
- **`insert(newNode, N)`** – Insert new node (`newNode`) at spot **N**

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**

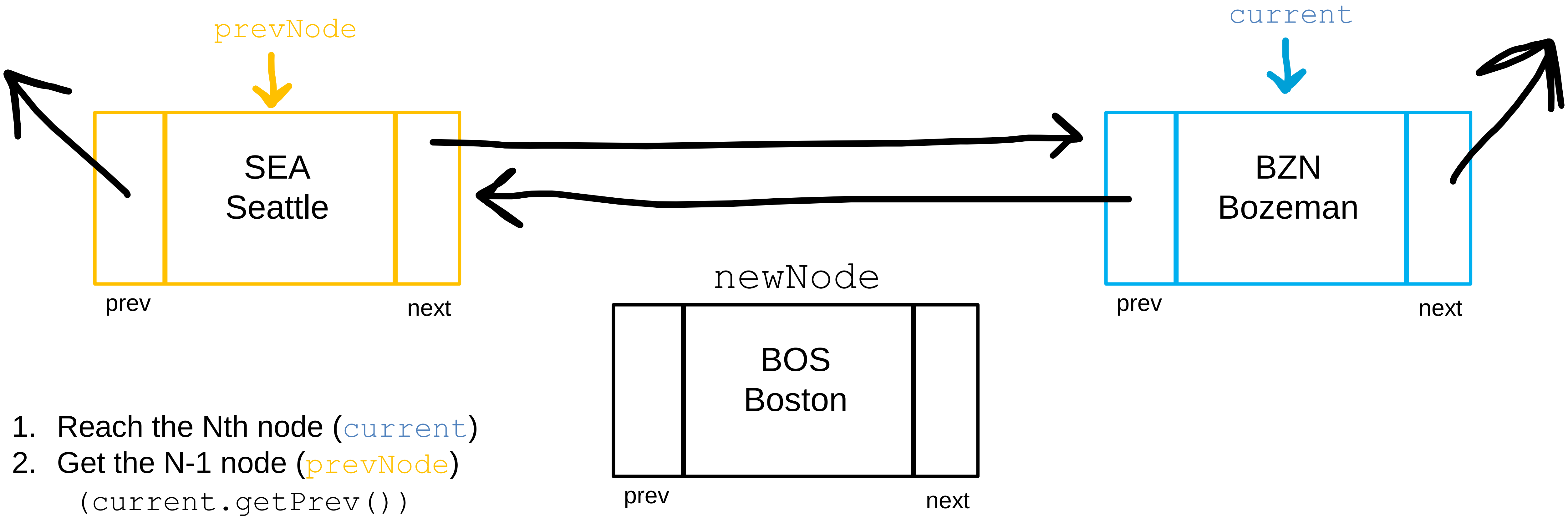
Case 4: The user is inserting a node somewhere in the middle of the LL



1. Reach the Nth node (`current`)

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

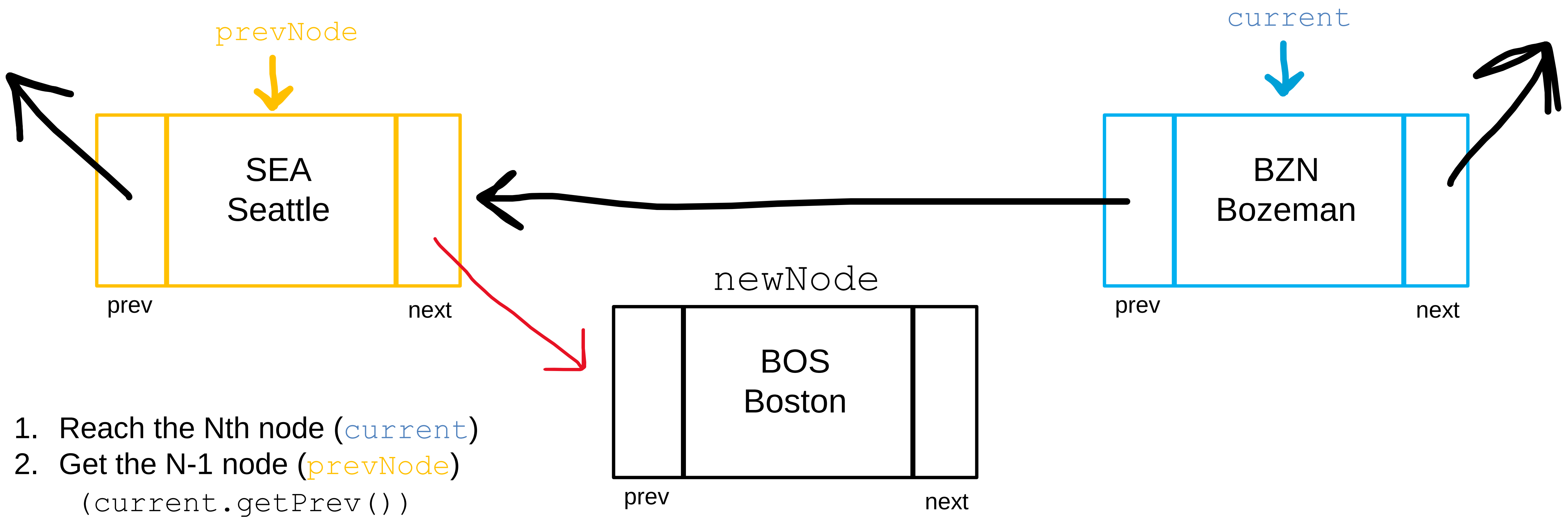
Case 4: The user is inserting a node somewhere in the middle of the LL



1. Reach the Nth node (`current`)
2. Get the N-1 node (`prevNode`)
   `(current.getPrev())`

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

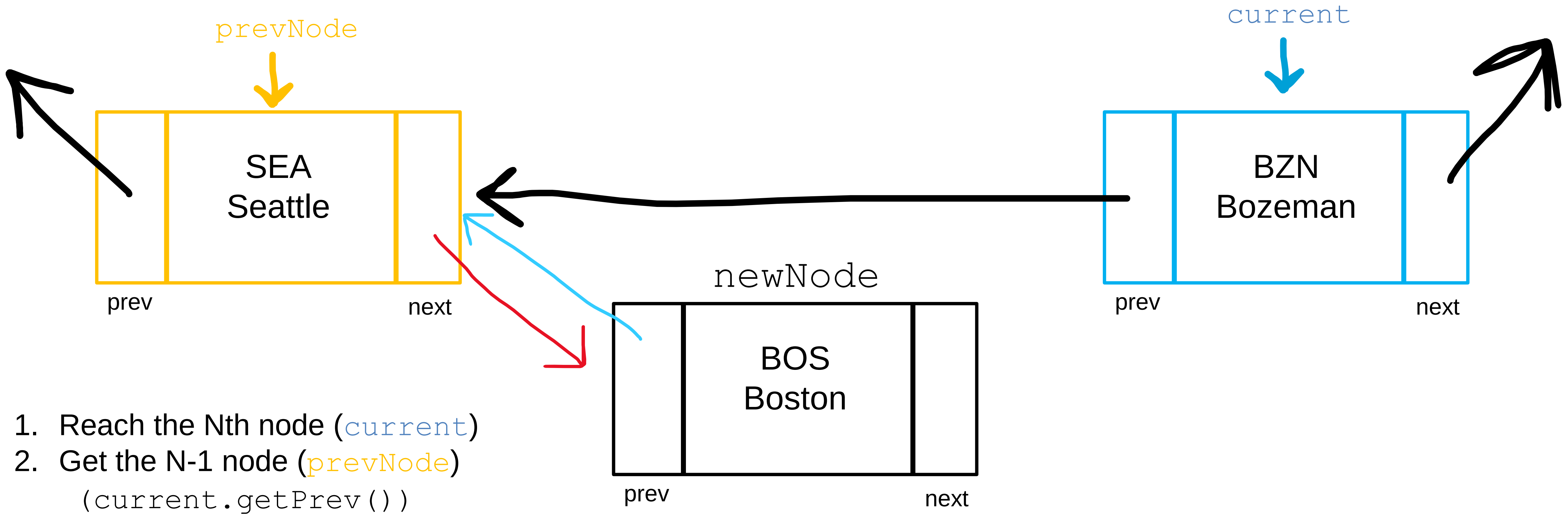
Case 4: The user is inserting a node somewhere in the middle of the LL



1. Reach the Nth node (`current`)
2. Get the N-1 node (`prevNode`)
   (`current.getPrev()`)
3. Update `prevNode`'s `next` pointer

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

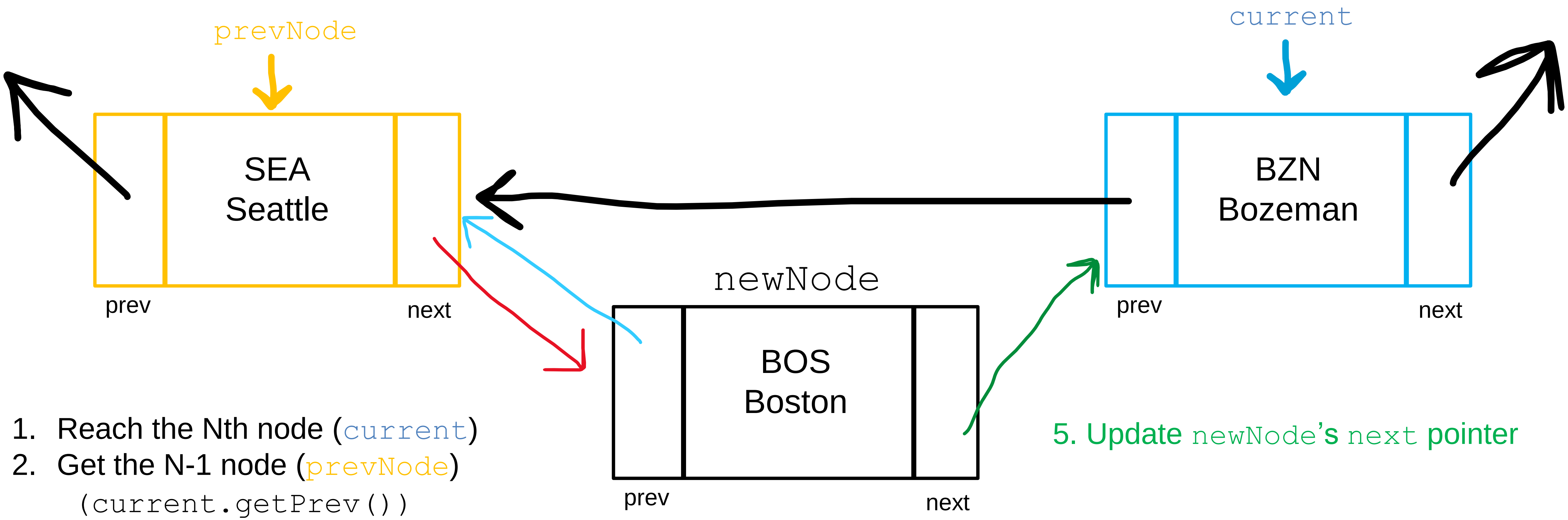
Case 4: The user is inserting a node somewhere in the middle of the LL



1. Reach the Nth node (`current`)
2. Get the N-1 node (`prevNode`)
   `(current.getPrev())`
3. Update `prevNode`'s `next` pointer
4. Update `newNode`'s `prev` pointer

- **`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

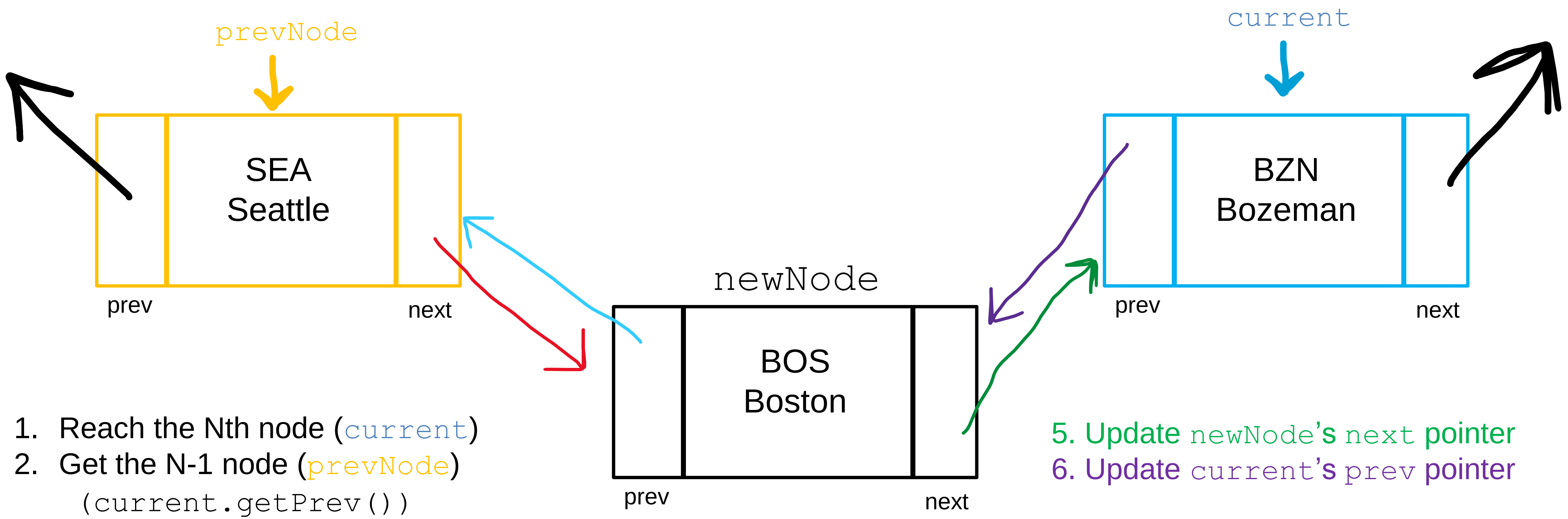


1. Reach the Nth node (`current`)
2. Get the N-1 node (`prevNode`)
   `(current.getPrev())`
3. Update `prevNode`'s `next` pointer
4. Update `newNode`'s `prev` pointer

5. Update `newNode`'s `next` pointer

- **`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. Reach the Nth node (`current`)
2. Get the N-1 node (`prevNode`)
   `(current.getPrev())`
3. Update `prevNode`'s `next` pointer
4. Update `newNode`'s `prev` pointer

5. Update `newNode`'s `next` pointer
6. Update `current`'s `prev` pointer

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

```
public void insert(Node newNode, int n) {
```

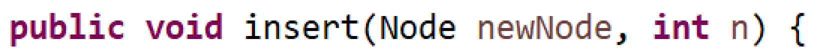
Case 1: The Linked List is Empty

```
//Case #1 Linked List is empty
if(this.size == 0) {
    this.head = newNode;
    this.tail = newNode;
}
```

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

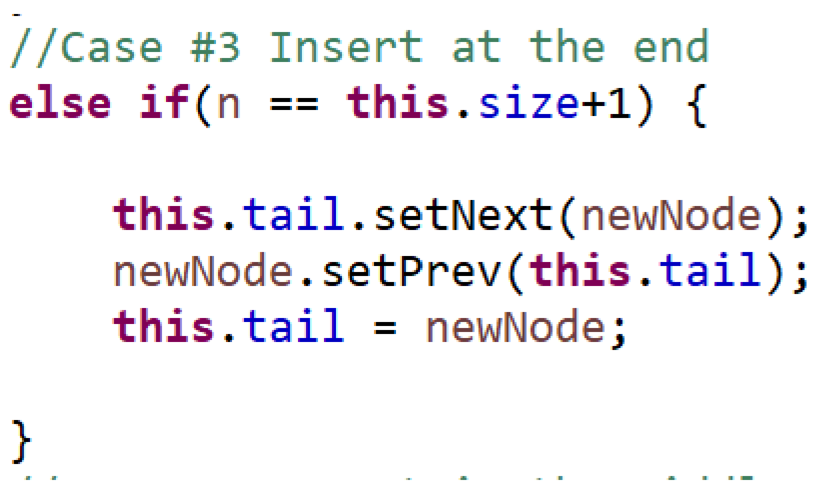
```
//Case #2 Insert at the beginning
else if(n == 1) {

    this.head.setPrev(newNode);
    newNode.setNext(this.head);
    this.head = newNode;

}
```

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



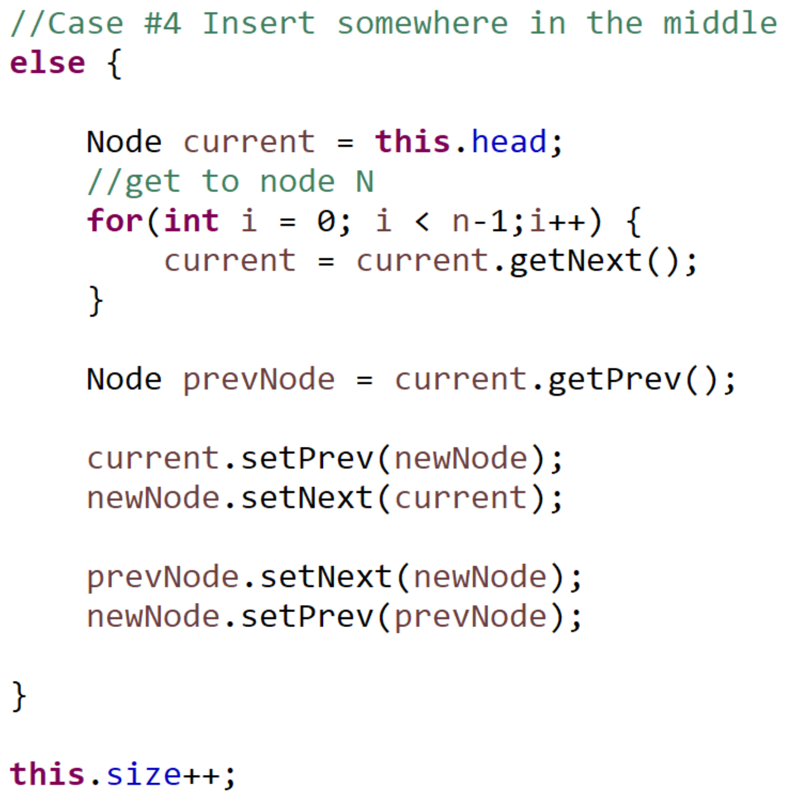
```java
public void insert(Node newNode, int n) {
```

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



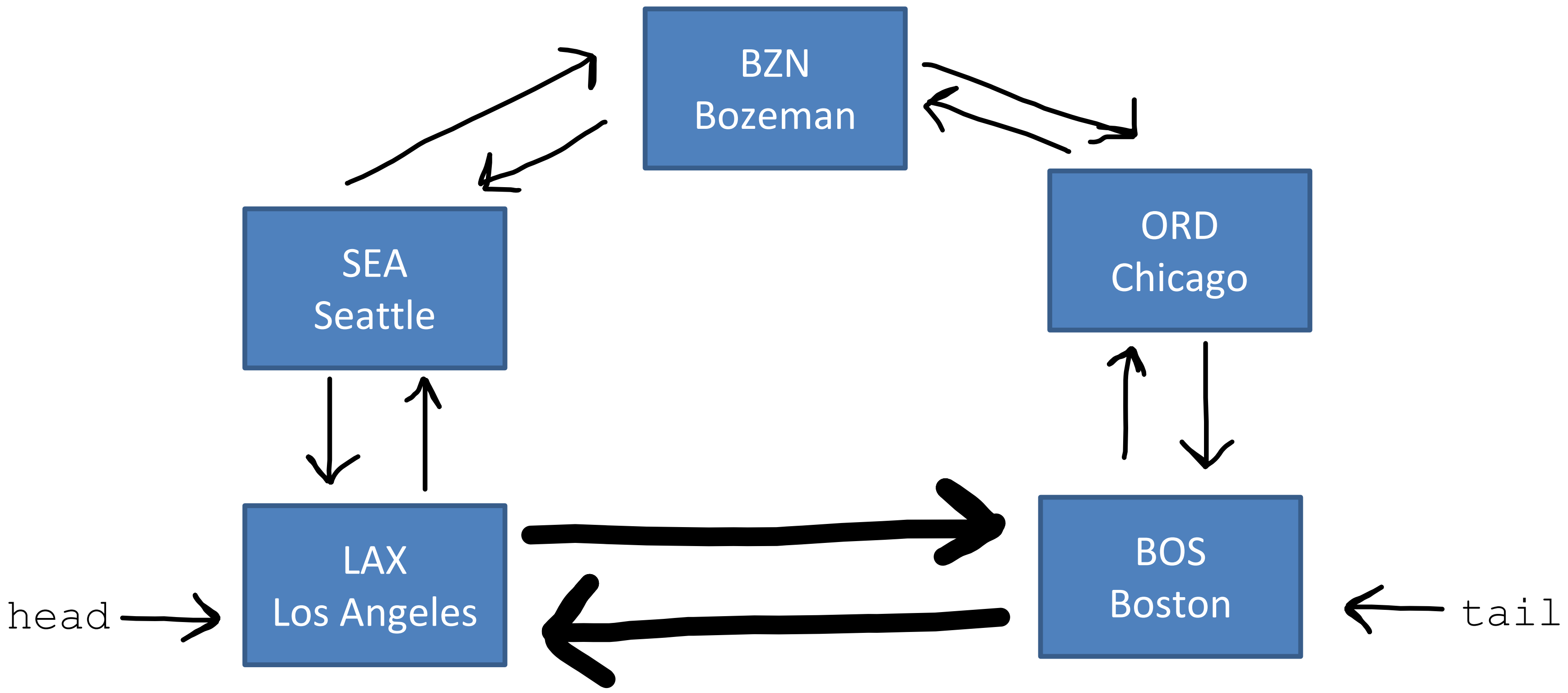
```java
//Case #3 Insert at the end
else if(n == this.size+1) {

    this.tail.setNext(newNode);
    newNode.setPrev(this.tail);
    this.tail = newNode;

}
```

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

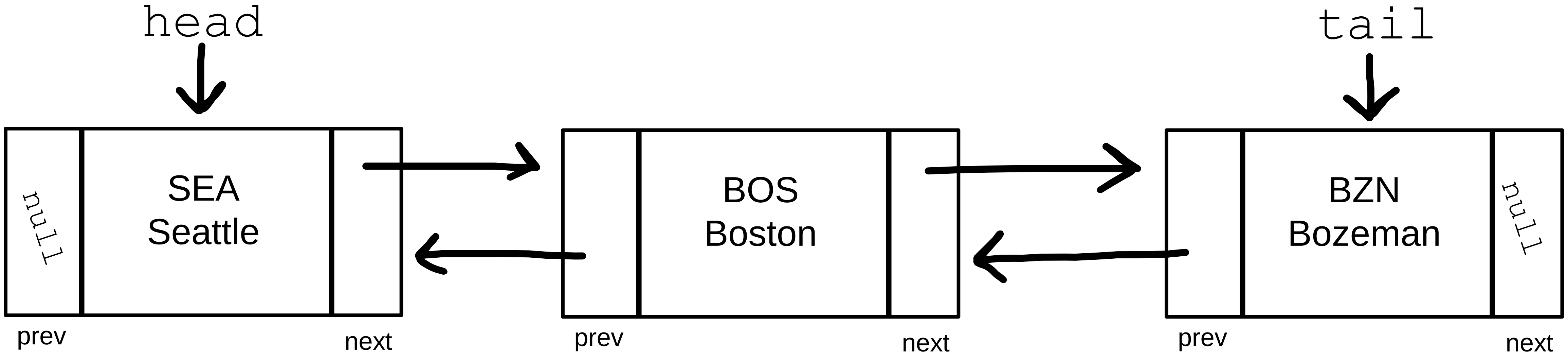


```java
//Case #4 Insert somewhere in the middle
else {

    Node current = this.head;
    //get to node N
    for(int i = 0; i < n-1;i++) {
        current = current.getNext();
    }

    Node prevNode = current.getPrev();

    current.setPrev(newNode);
    newNode.setNext(current);

    prevNode.setNext(newNode);
    newNode.setPrev(prevNode);

}

this.size++;
```

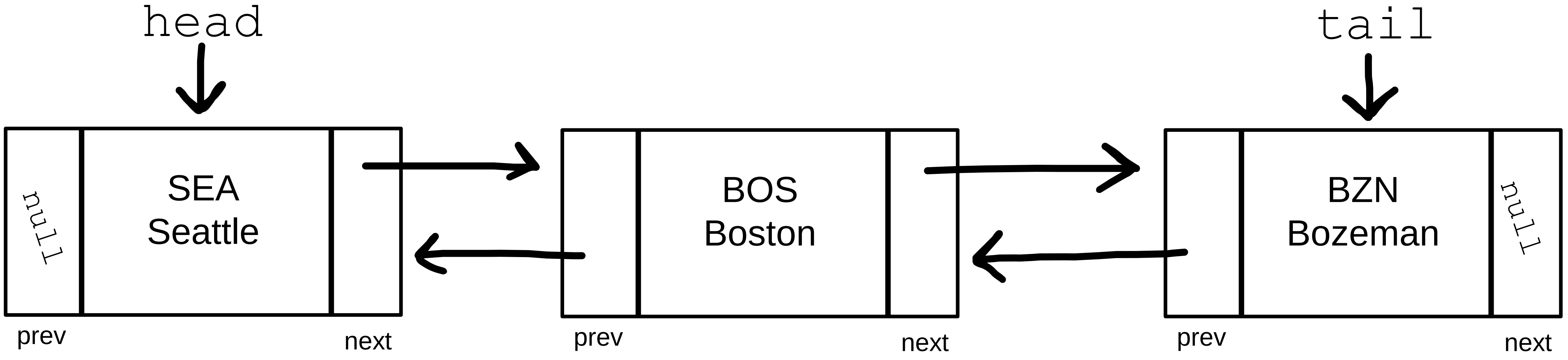
A **Circular Linked List** is a linked list where the first and last node are connected, which creates a circle



- remove(name) – Remove node by name

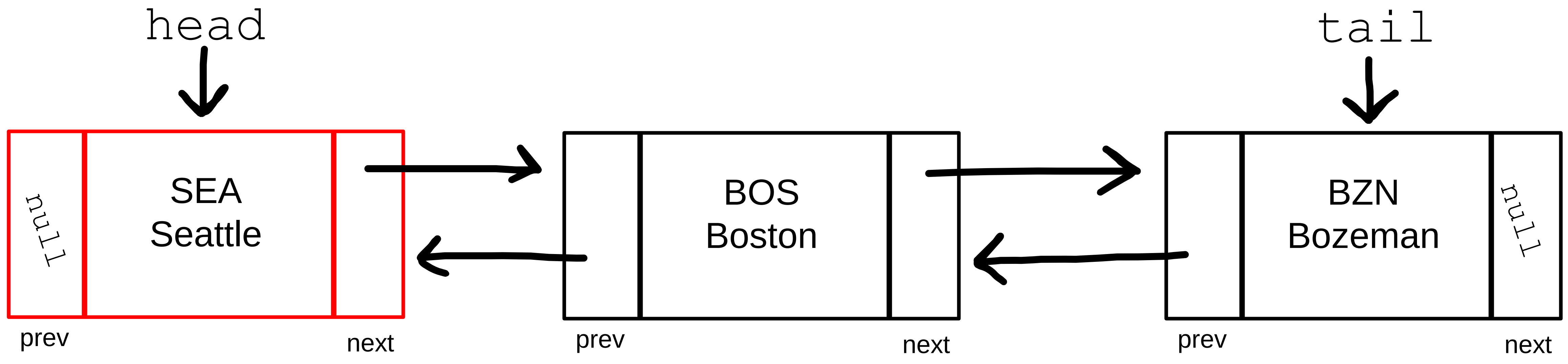


- remove(name) – Remove node by name



1. Traverse the Linked List and look for a match

- remove(name) – Remove node by name

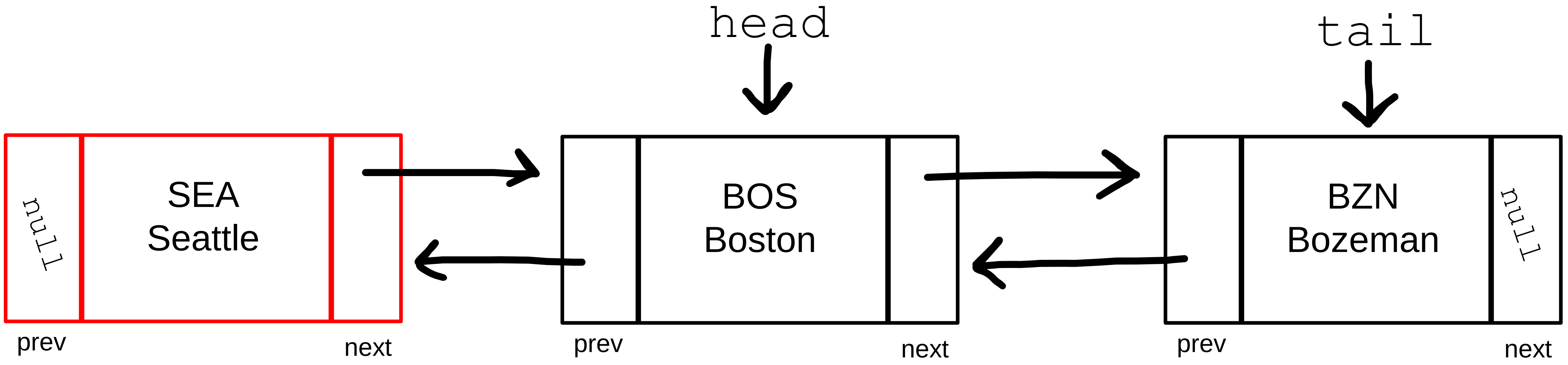


1. Traverse the Linked List and look for a match

`remove("SEA")`

What if the removed node is the head?

- remove(name) – Remove node by name



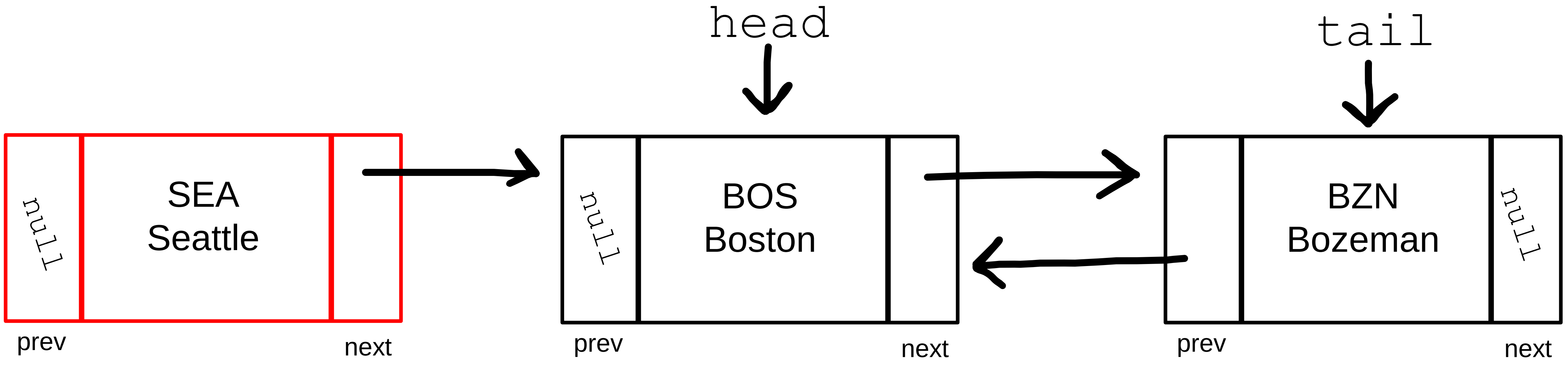
1. Traverse the Linked List and look for a match

`remove("SEA")`

What if the removed node is the head?

2. Update the `head` to be the next node

- remove(name) – Remove node by name



1. Traverse the Linked List and look for a match

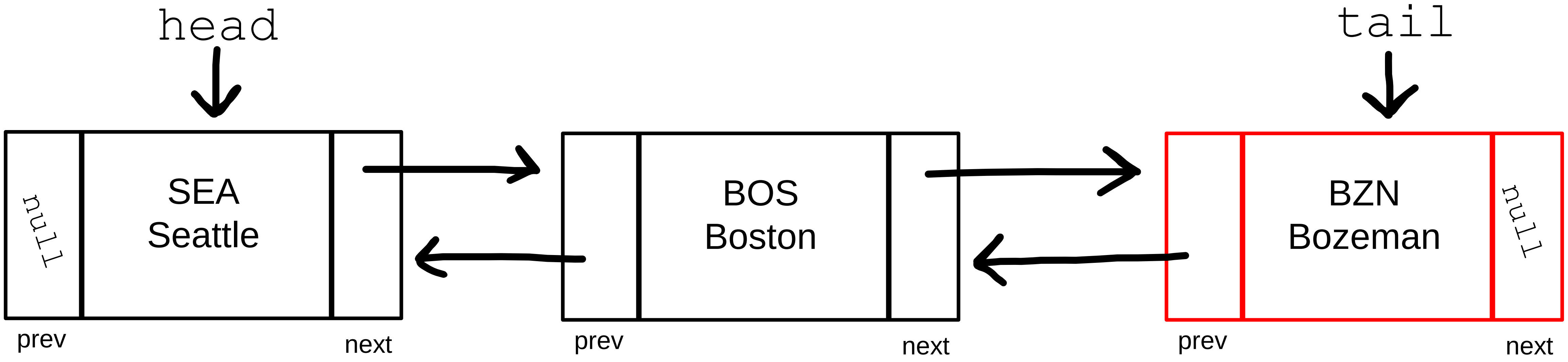
`remove("SEA")`

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

We can longer reach the SEA node from the head node, so it is effectively removed

- remove(name) – Remove node by name

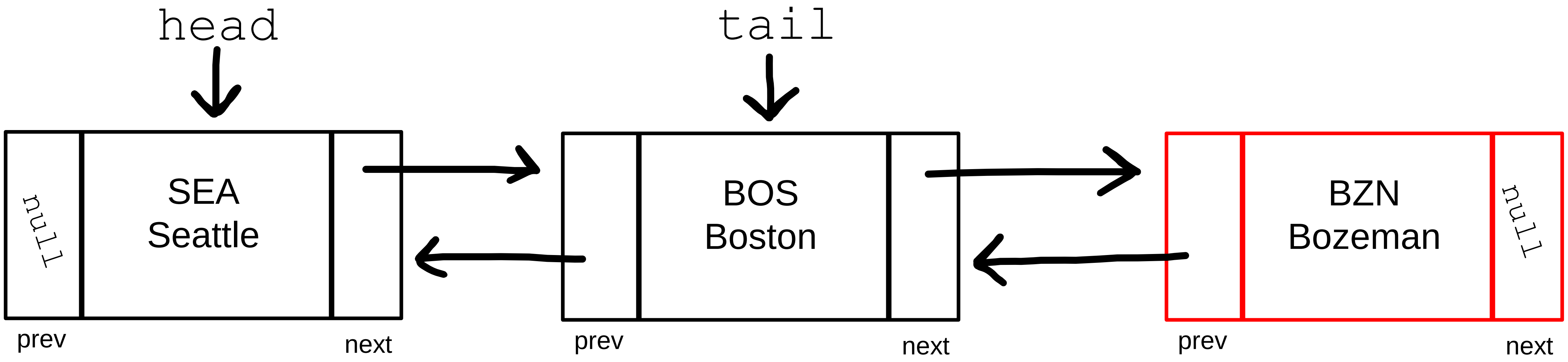


1. Traverse the Linked List and look for a match

`remove(“BZN”)`

What if the removed node is the tail?

- remove(name) – Remove node by name



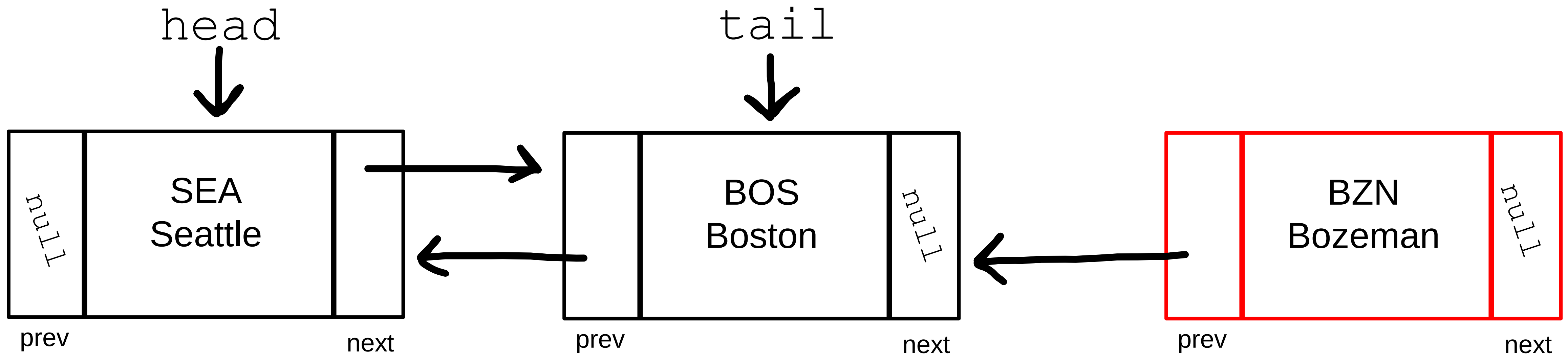
1. Traverse the Linked List and look for a match

`remove("BZN")`

What if the removed node is the tail?

2. Update the `tail` to be the previous node

- remove(name) – Remove node by name



1. Traverse the Linked List and look for a match

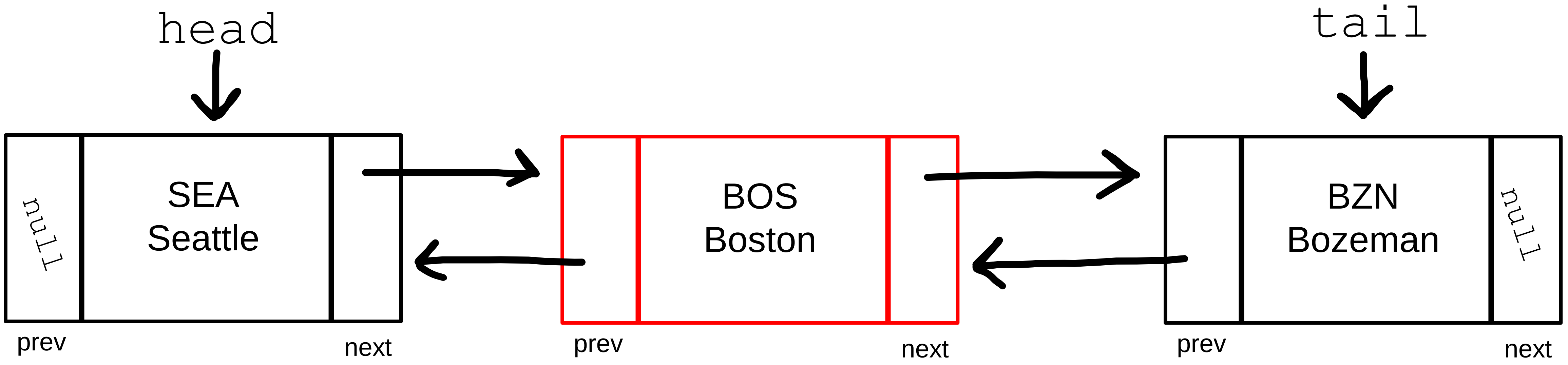
`remove("BZN")`

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

We can longer reach the BZN node from the head node, so it is effectively removed

- remove(name) – Remove node by name

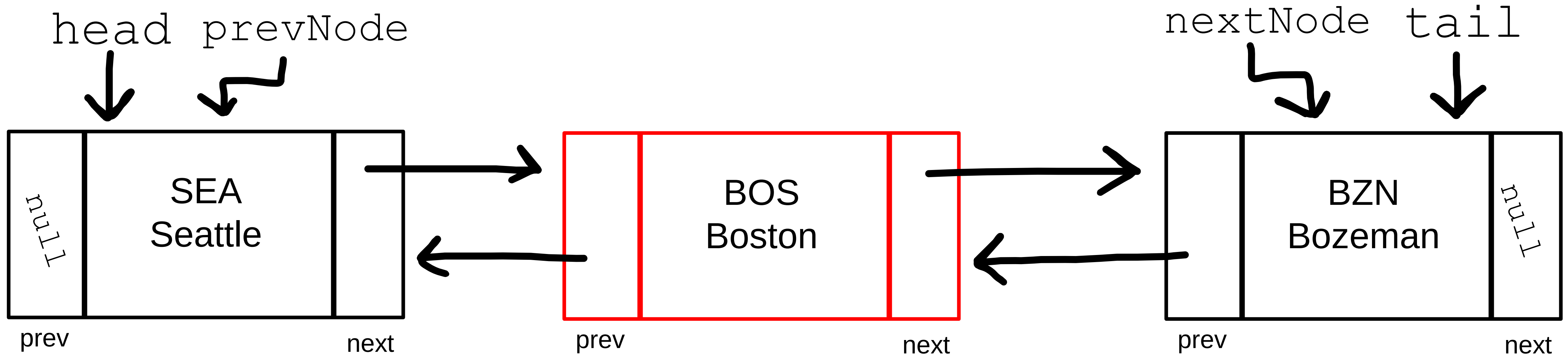


1. Traverse the Linked List and look for a match

`remove(“BOS”)`

What if the removed node is somewhere in the middle?

- remove(name) – Remove node by name



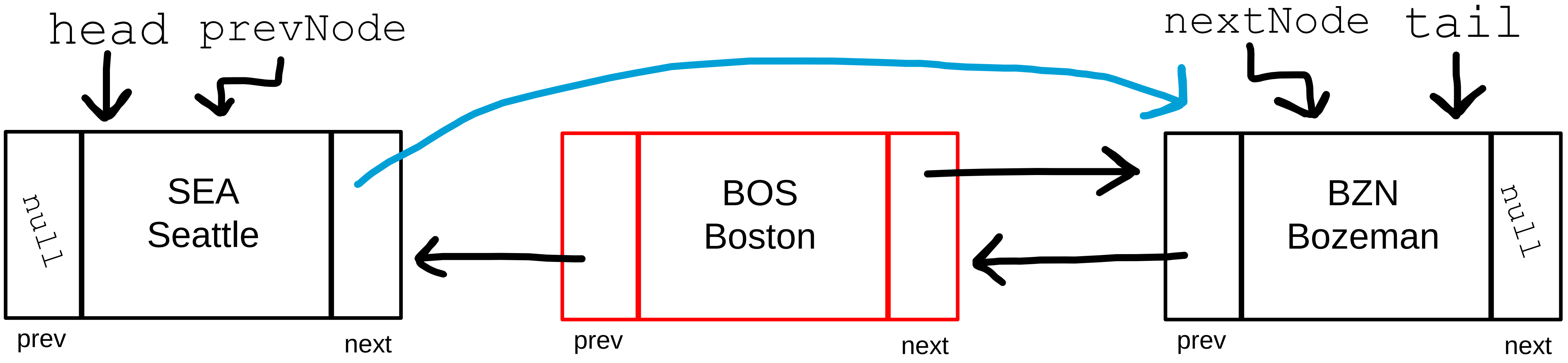
1. Traverse the Linked List and look for a match

`remove("BOS")`

What if the removed node is somewhere in the middle?

2. Retrieve the previous node and next node of the to-be-removed node

- remove(name) – Remove node by name



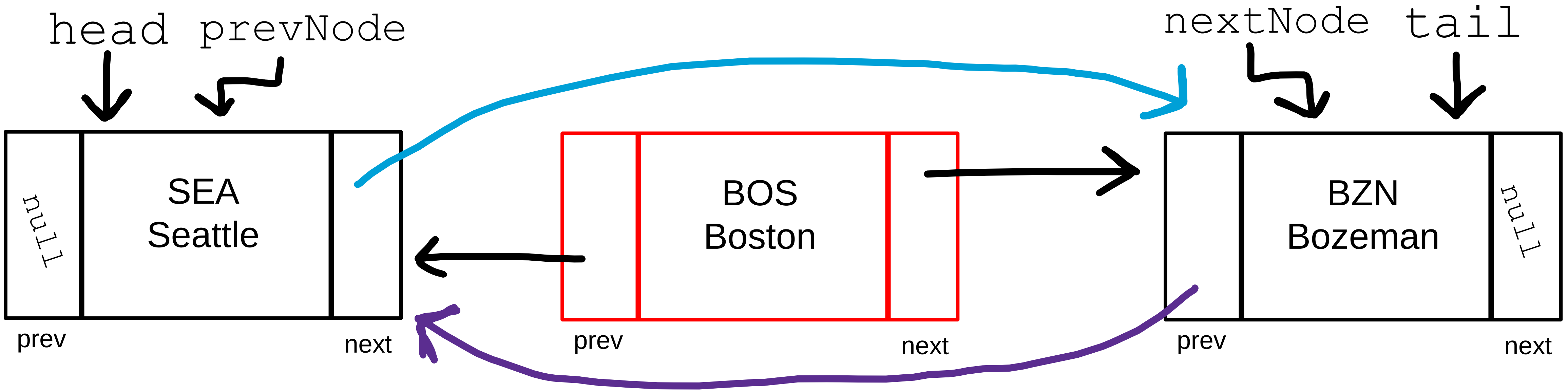
1. Traverse the Linked List and look for a match

`remove("BOS")`

What if the removed node is somewhere in the middle?

2. Retrieve the previous node and next node of the to-be-removed node
3. Update `prevNode`'s `next` value to be the `nextNode`

- remove(name) – Remove node by name



1. Traverse the Linked List and look for a match

`remove("BOS")`

What if the removed node is somewhere in the middle?

2. Retrieve the previous node and next node of the to-be-removed node
3. Update `prevNode`'s `next` value to be the `nextNode`
4. Update `nextNode`'s `prev` value to be `prevNode`