Data Structures

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

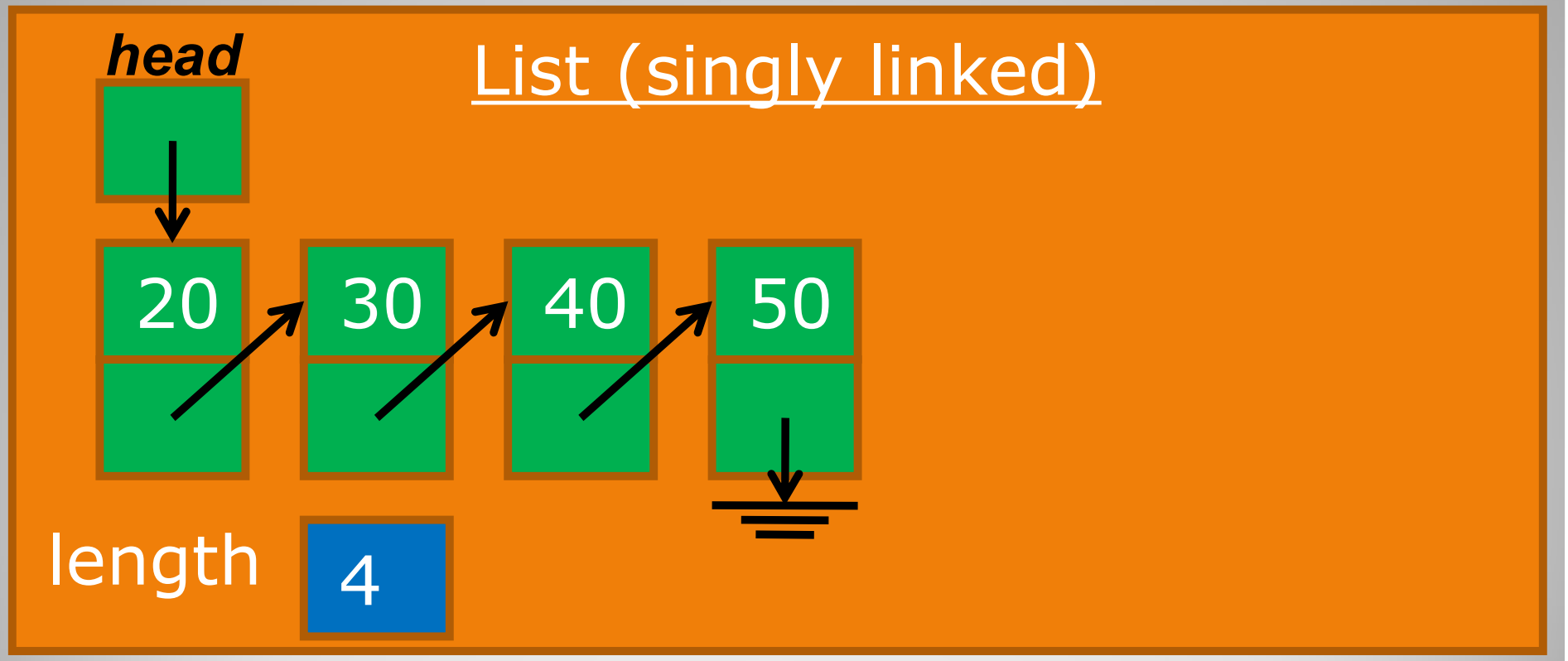
Today's Lecture

- Ordered Doubly-linked lists

Singly-Linked List

- A singly-linked list has links going in only one direction.
- For example…

- Pointers only go forward.

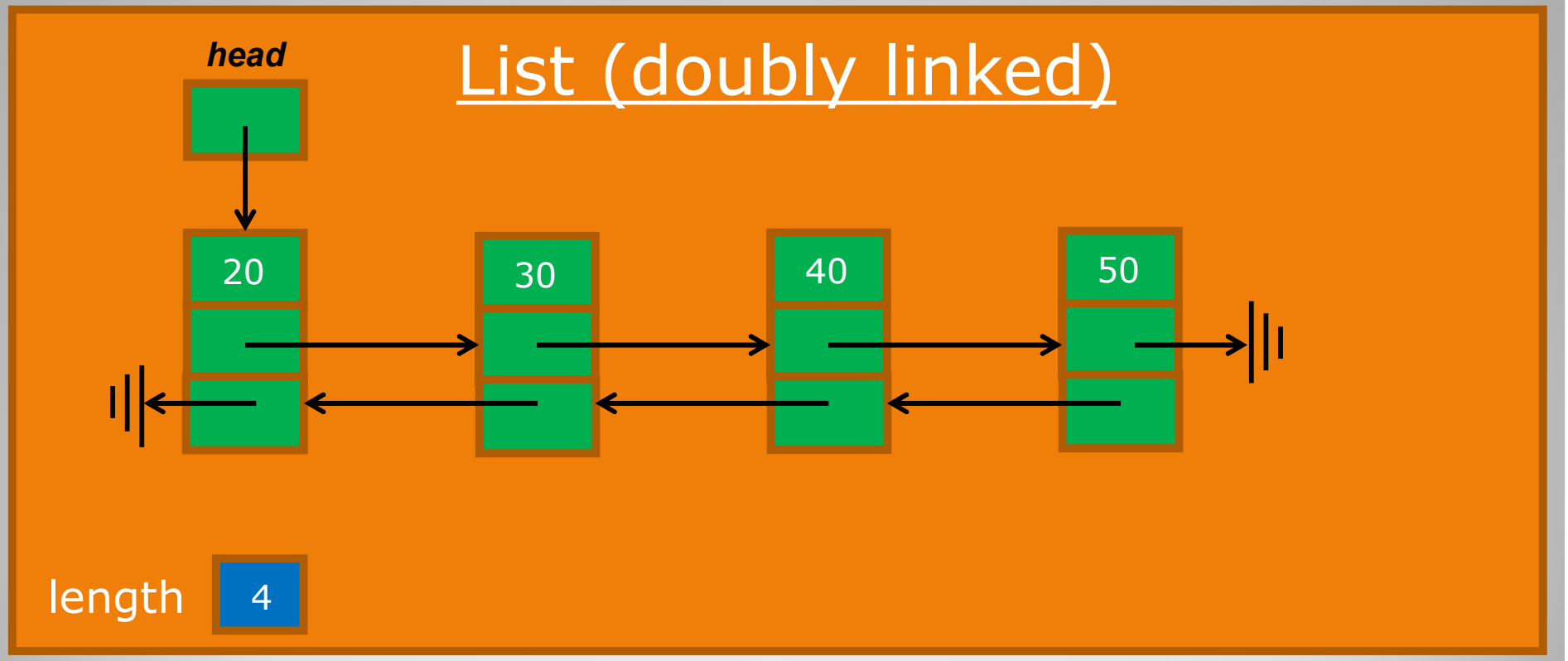


Singly-linked List

Doubly-Linked List

- A doubly-linked list has links going in BOTH directions.
- For example…

- Pointers go in BOTH directions.
- Each node has a next and previous pointer.



Doubly-linked List

List Interface

- Here is the List Interface we will be using (same as for the singly-linked list):

```
public interface List {
    public void insertItem(int item);
    public void deleteItem(int item);
    public boolean hasItem(int target);
    public int retrieveItem(int target) throws Exception;
    public void makeEmpty();
    public boolean isFull();
    public int getLength();
}
```

Note: Java has it own predefined List interface but it is more complicated, so we are using our own version.

OrderedList Class (doubly linked)

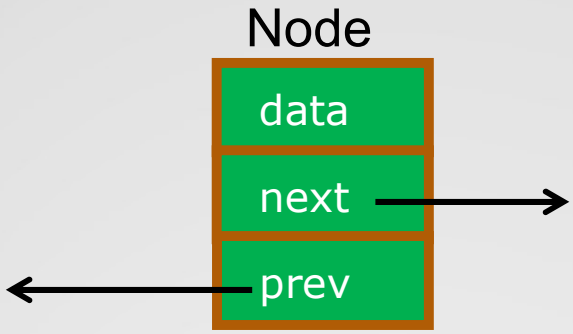
- We will write an OrderedList class that implements our List interface.
- This implementation will have both next and previous links in each node.

```
public class OrderedList implements List
{
   // Implementation code goes here
}
```

Node (doubly linked)

- The doubly-linked-list data structure requires that we keep TWO links at each node.
- Each item in the list will be a "Node" (not just the data).
- A node stores data references to both the next node and the previous node.
- Defined Node as an inner class within the ordered list class.

```
class Node {
   Declare int data       // Data item
   Declare Node next      // Points to NEXT node
   Declare Node prev      // Points to PREVIOUS node
}
```



- Doubly-linked-based **private** members (same as for singly-linked):

```
class  OrderedList {
    Declare Node head
    Declare int length

    // Public members go here…
}
```

List (doubly linked) Class Member Variables

- **What should the OrderedList constructor do?**

List (doubly linked) - Constructor

- **What should the OrderedList constructor do?**

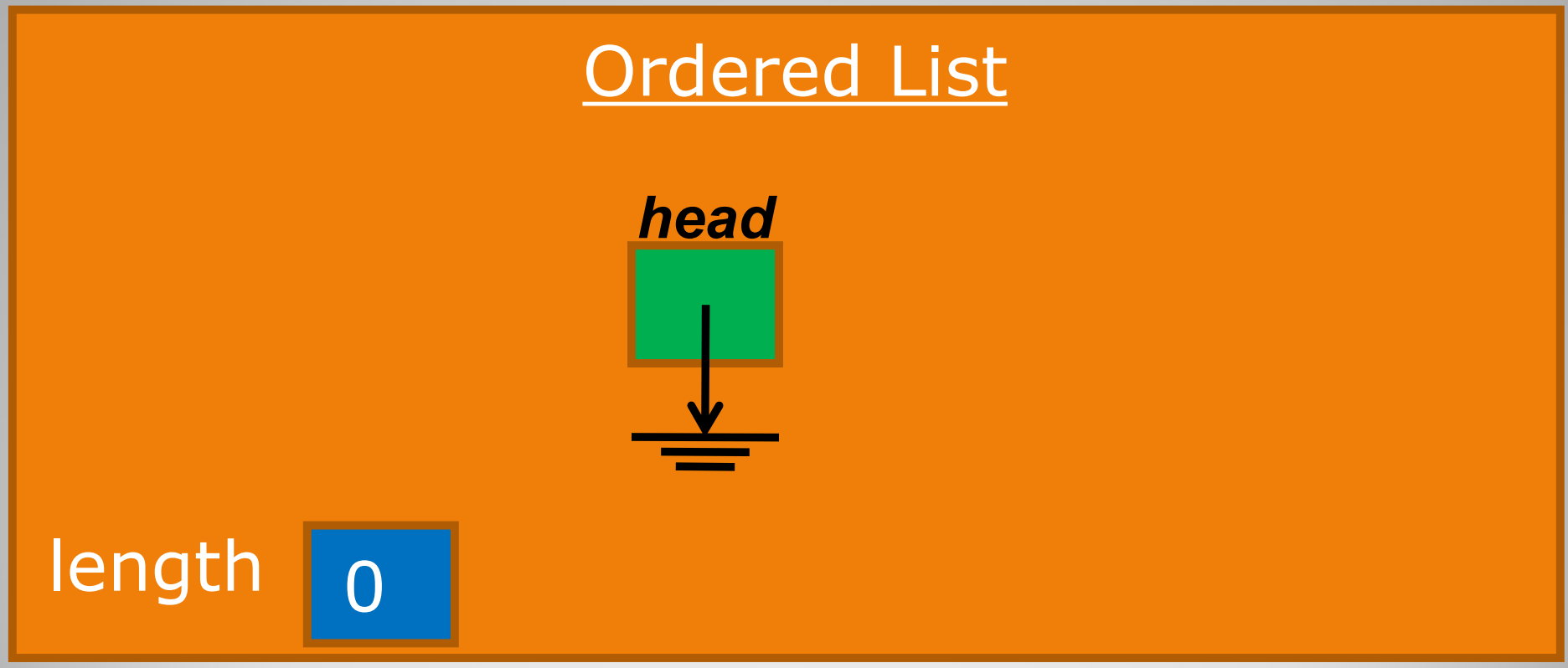
OrderedList Constructor

Set length to 0 // Sets the # of elements to 0

Set head to null

List (doubly linked) - Constructor

- Ordered list AFTER default constructor runs.



List (doubly linked) - Constructor

```
isFull() returns boolean
    Declare Node location
    try
        Set location to new Node instance
        Set location to null
        return false
    catch OutOfMemoryError exception
        return true
```

Check to see if you can allocate memory.

If you CAN, then the list is NOT full so return false.

If you CANNOT allocate memory, then the list is full.

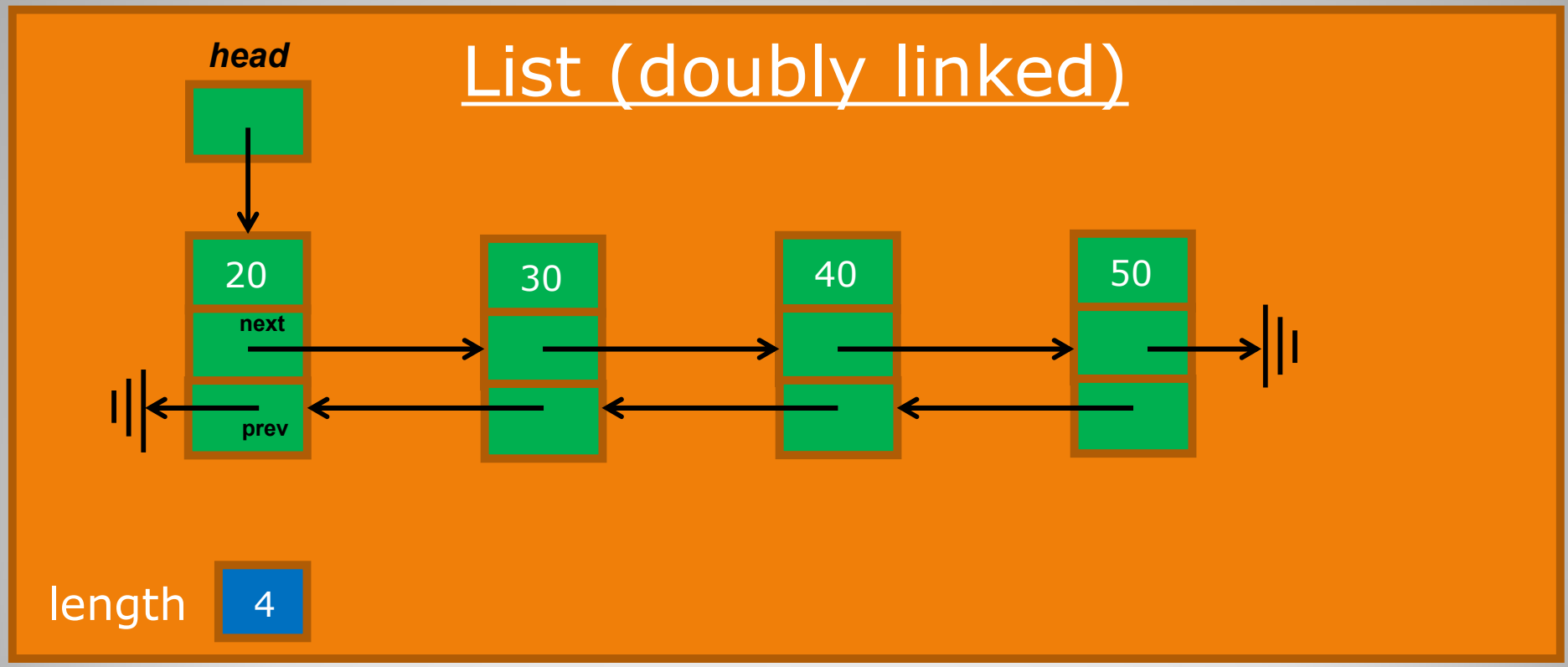
List (doubly linked) - isFull

How do you insert an item?

Where does it go in the list?

List (doubly linked) - insertItem

- Where would a new item go? How is it inserted?
  ol.insertItem(10)



List (doubly linked) - insertItem

Since the list is ***ordered*** (and there are no other constraints) we can put it anywhere in the list.

The easiest place to insert is at the beginning (same as for singly linked).

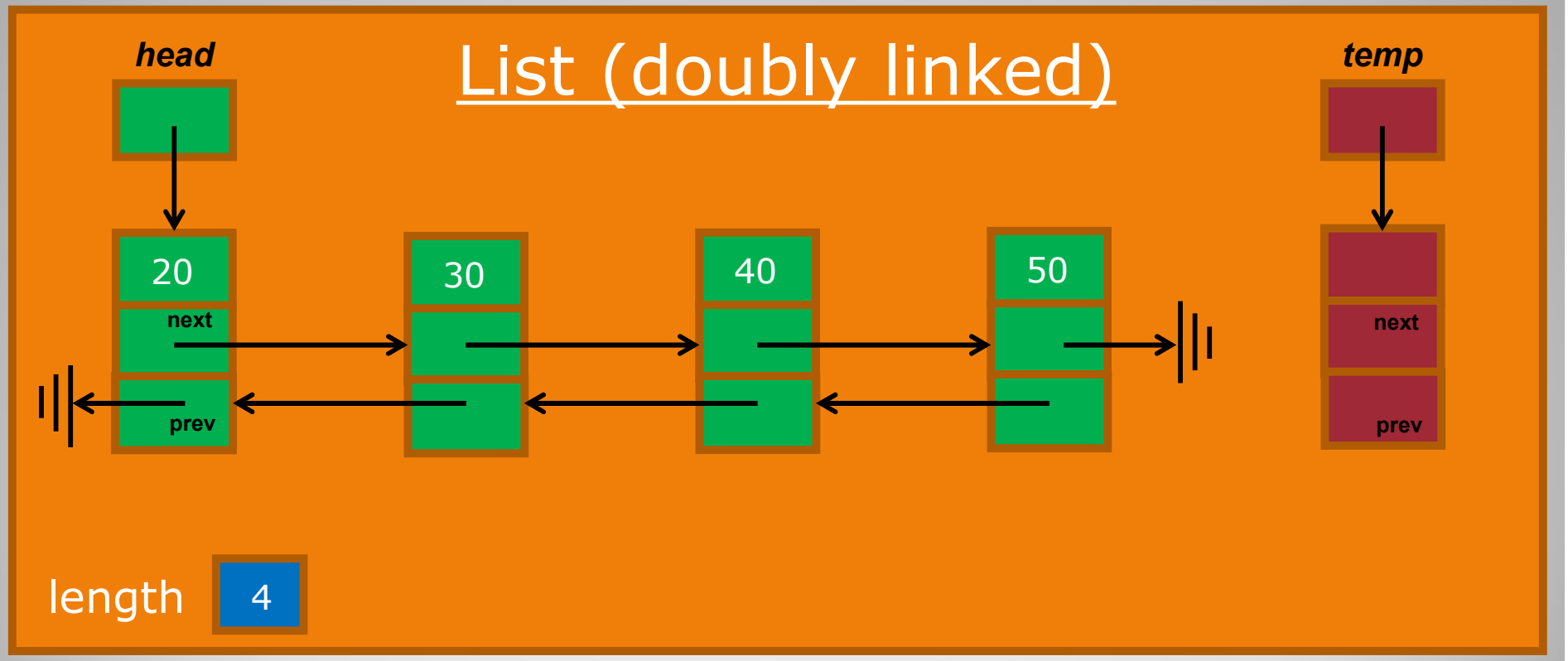
insertItem Pseudocode

1. Create a new Node instance (dynamically allocate).
2. Set the fields on the new Node.
   A. Set the data.
   B. Set the next pointer to the current head.
   C. Set the prev pointer to null.
3. Set the original head's previous to the new Node.
4. Set the head pointer to the new Node.
5. Increment the length of the list.

List (doubly linked) - insertItem

ol.insertItem(10)

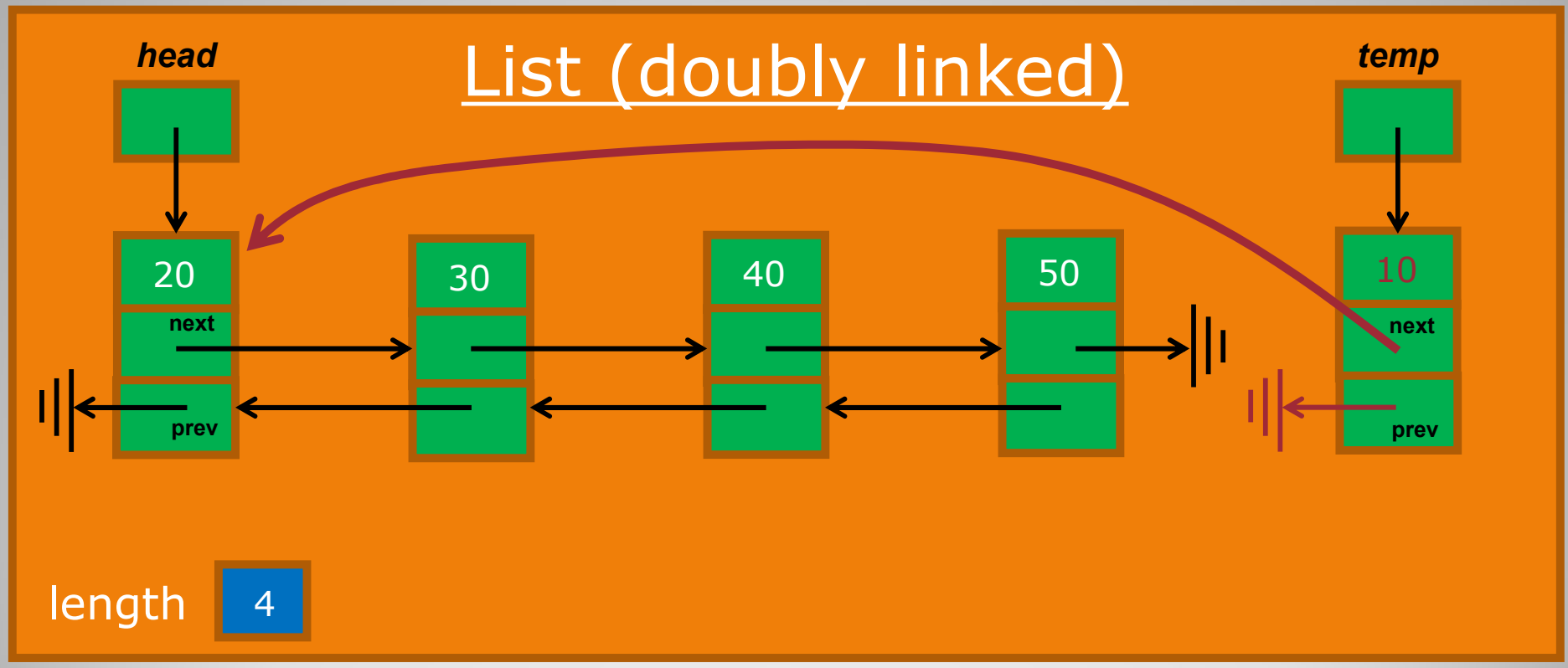
1. Create a new Node instance (dynamically allocate).



List (doubly linked) - insertItem

ol.insertItem(10)

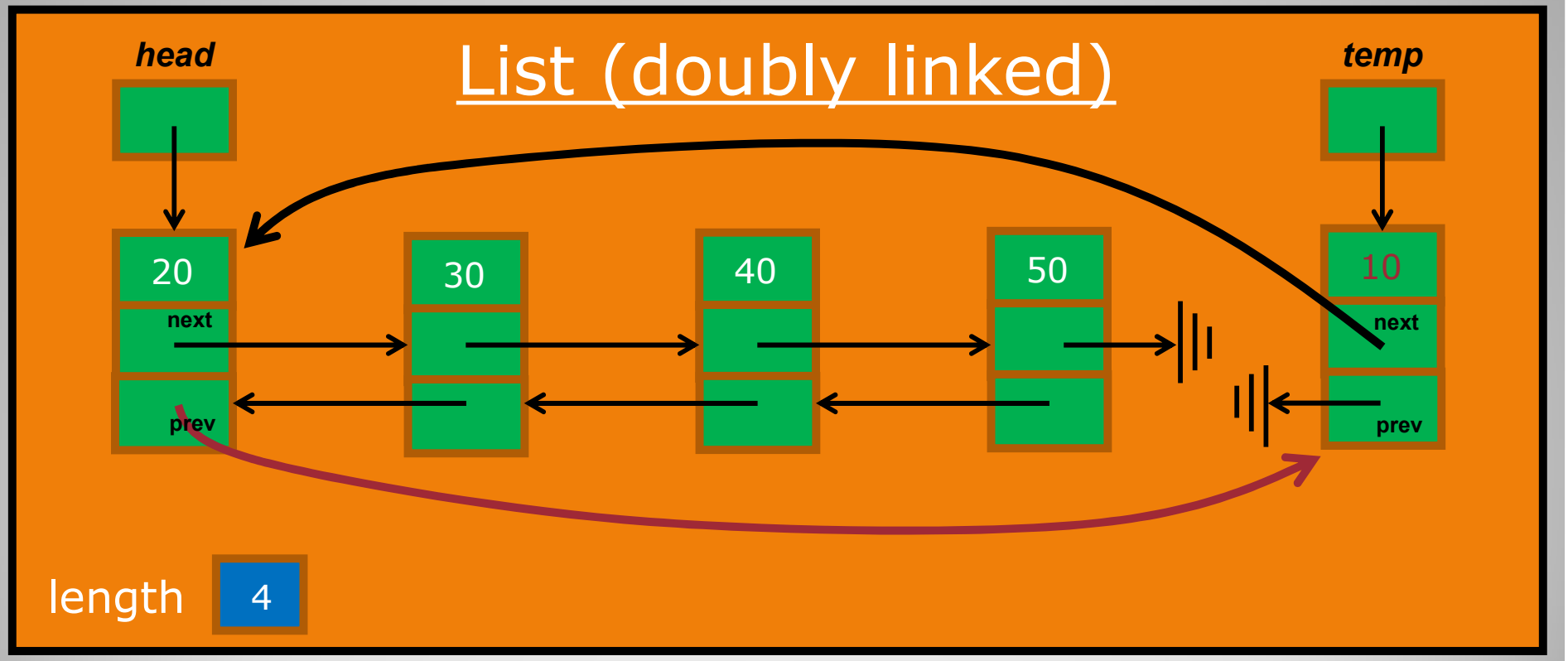
2. Set the fields on the new Node. Set data, next, and prev. Next points to current head. Prev points to null.



List (doubly linked) - insertItem

ol.insertItem(10)

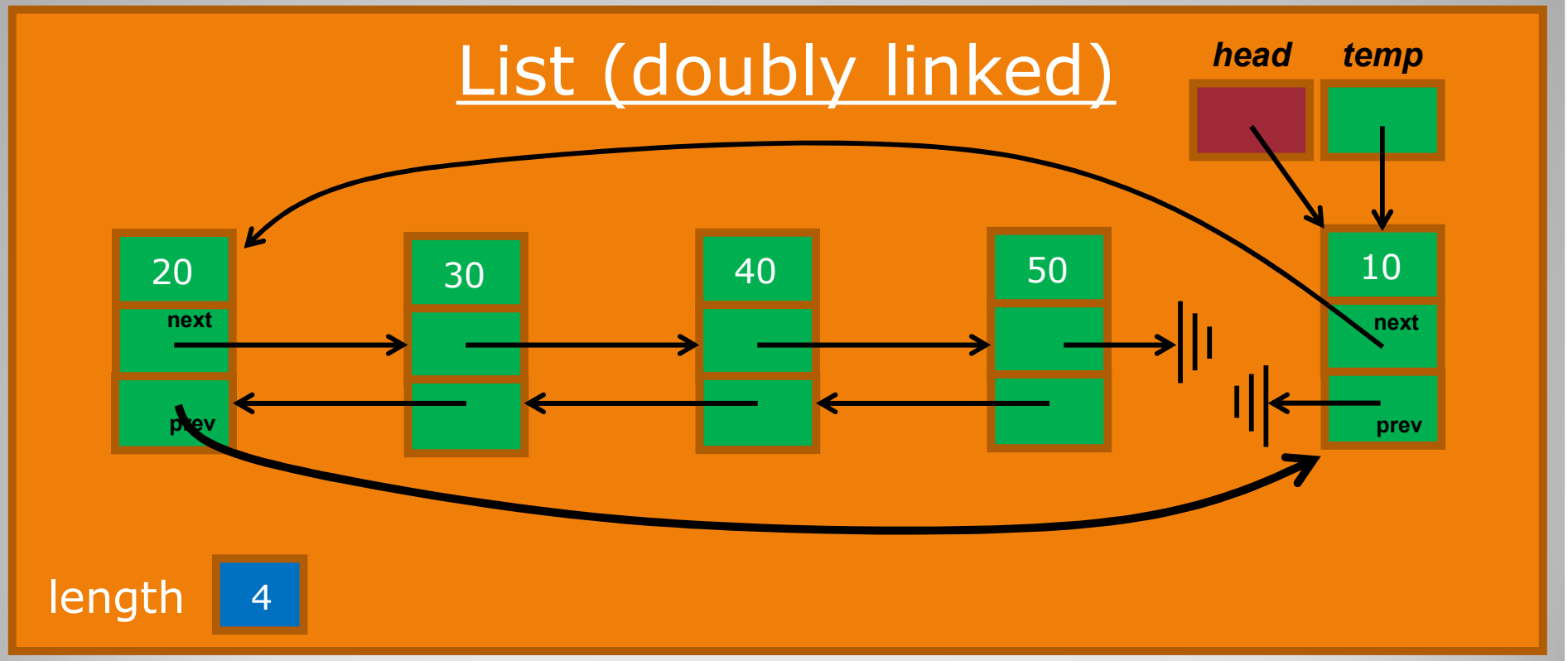
3. Set the original head's previous to the new Node.



List (doubly linked) - insertItem

ol.insertItem(10)

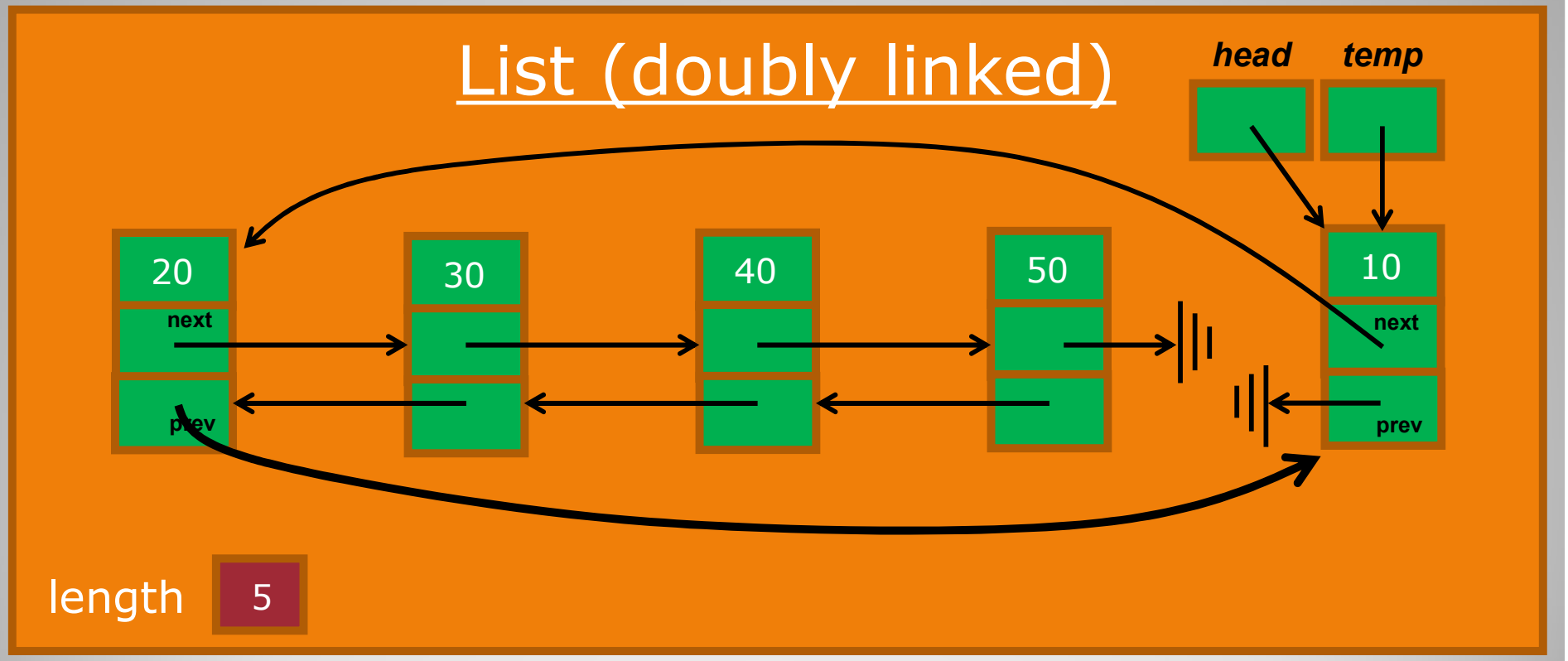
4. Set the list start pointer (head) to the new Node.



List (doubly linked) - insertItem

ol.insertItem(10)

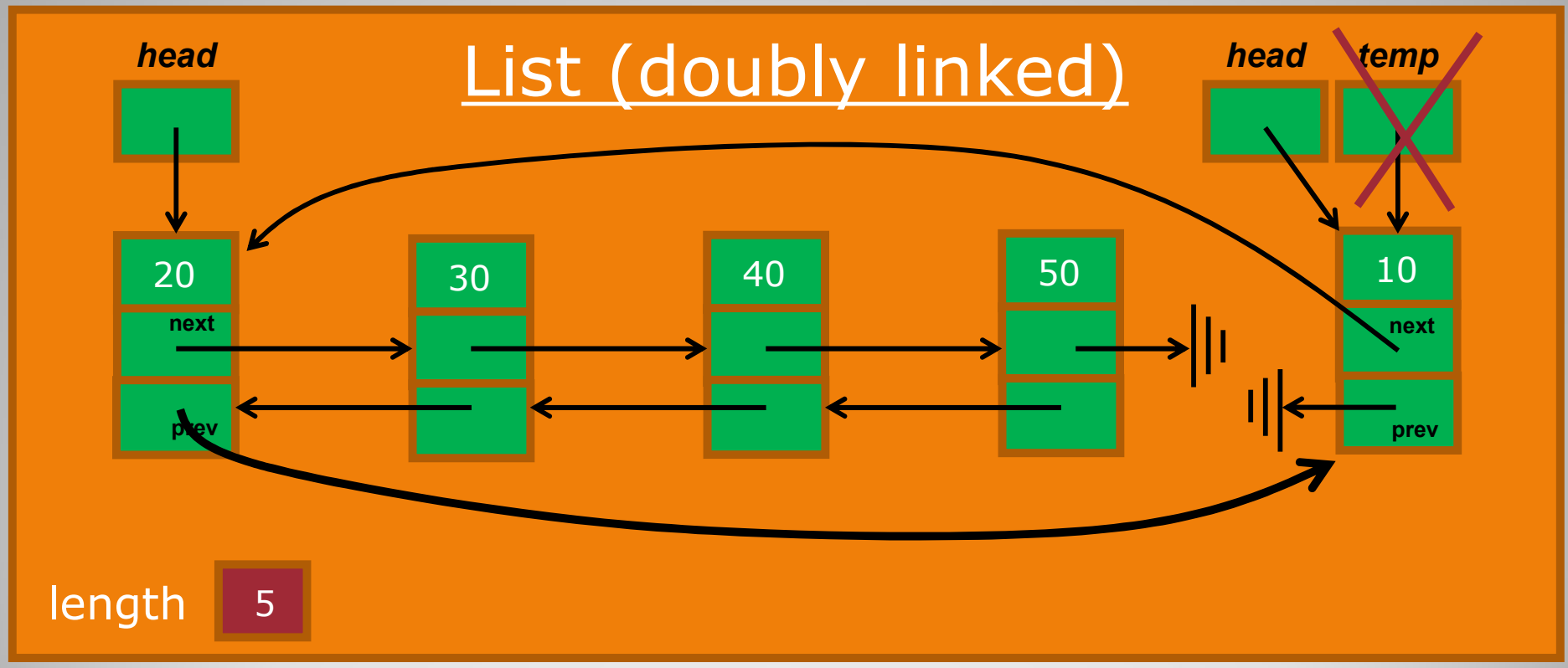
4. Increment the length of the list.



List (doubly linked) - insertItem

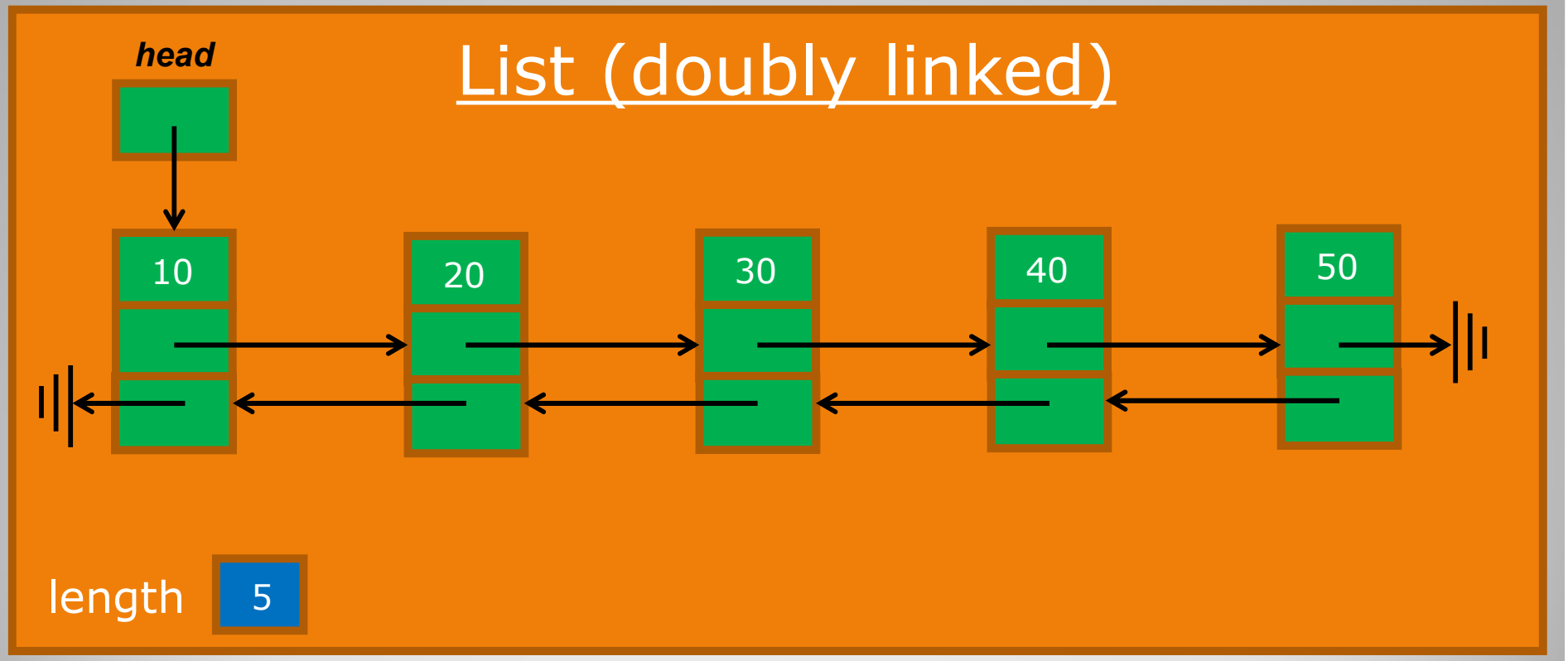
ol.insertItem(10)

When the insertItem method ends the temp pointer will go out of scope and disappear.



List (doubly linked) - insertItem

This picture is ***LOGICALLY EQUIVALENT*** to the previous slide!!!



List (doubly linked) - insertItem

```
insertItem(int item)
    Declare Node temp

    Set temp to new Node instance

    Set temp.data to item
    Set temp.next to head
    Set temp.prev to null

    if (head not equal to null)
       Set head.prev to temp

    Set head to temp
    Increment length
```

If list was empty, then head will be null (only set prev if there is actually a node at the head)

List (doubly linked) - insertItem

List (doubly linked) - hasItem

- Start from beginning and visit each node.

Same as for singly-linked list

```
hasItem(int target) returns boolean
    Declare Node location
    Set location to head
    while (location not equal to null)
        if (location.data equals target)
            return true

        Set location to location.next

    return false
```

Now we will move on to deleteItem…

List (doubly linked) - deleteItem

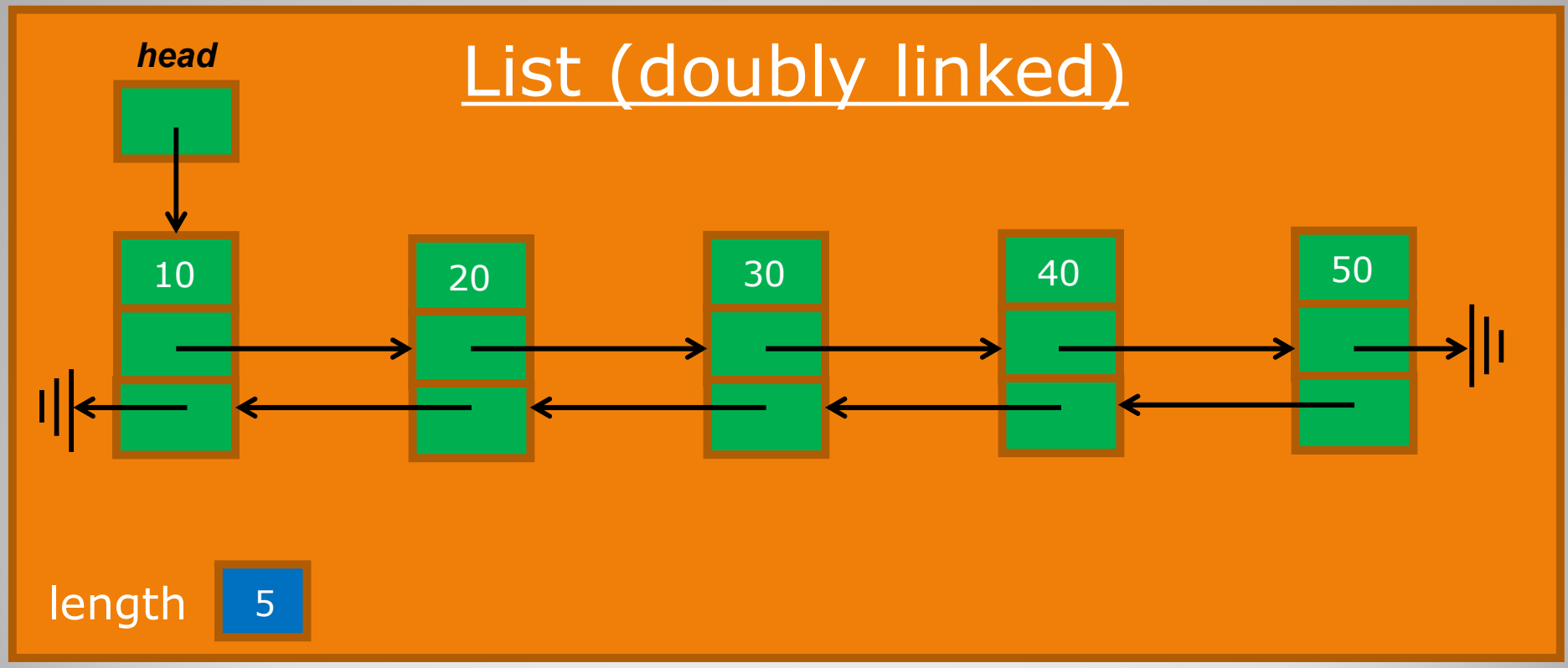
deleteItem Pseudocode (Detailed)

1. Find the target item to delete. Can be one of two cases:
   a) The start item is the target item.
   b) The target is somewhere else in the list.
2. Update the pointers in the list so that the target item is removed.
3. Set the target to null so memory for that node can eventually be given back to the system.
4. Decrement the length.

List (doubly linked) - deleteItem

We will now delete 10 from the list (10 is at the head)

ol.deleteItem(10)

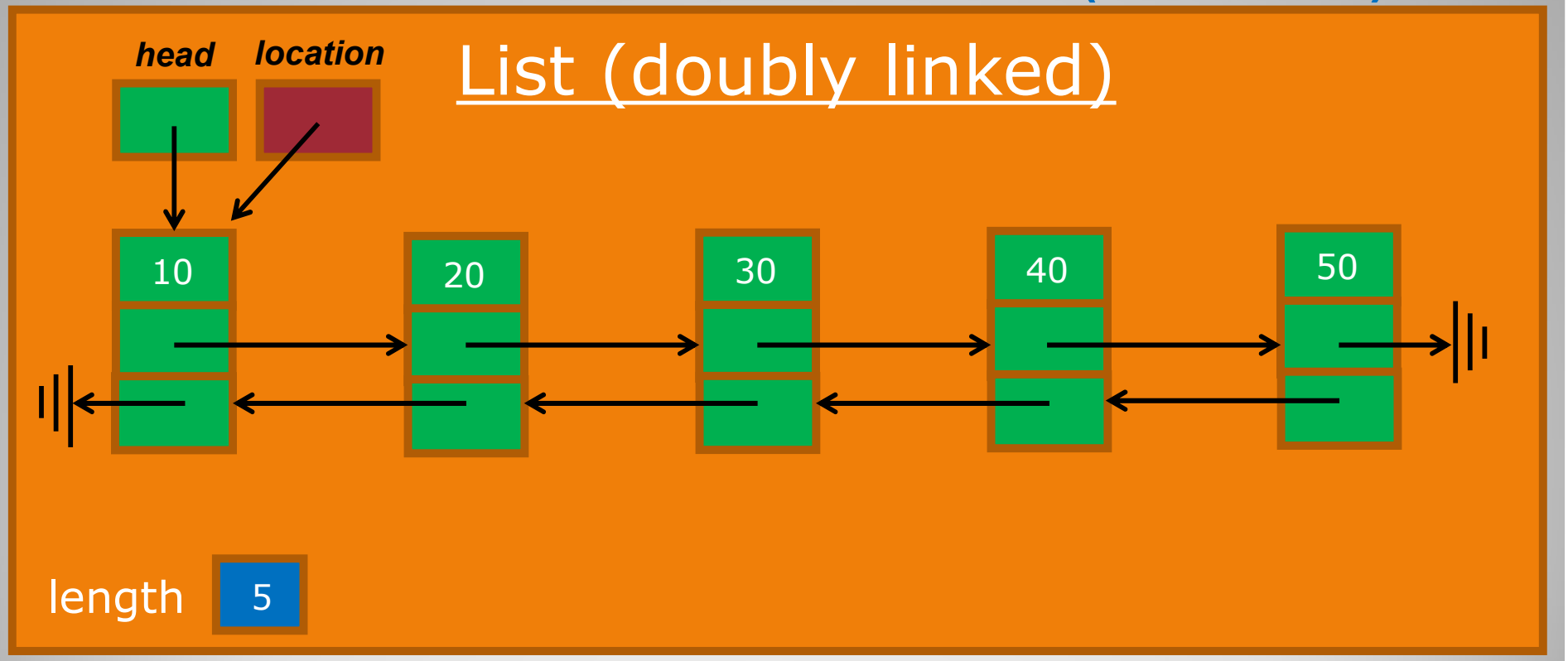


List (doubly linked) - deleteItem

Node location = head
if (item equals location.data)
head = head.next
head.prev = null
location = null
decrement length

Delete Code For Head Node

Target to delete is 10 (head item)

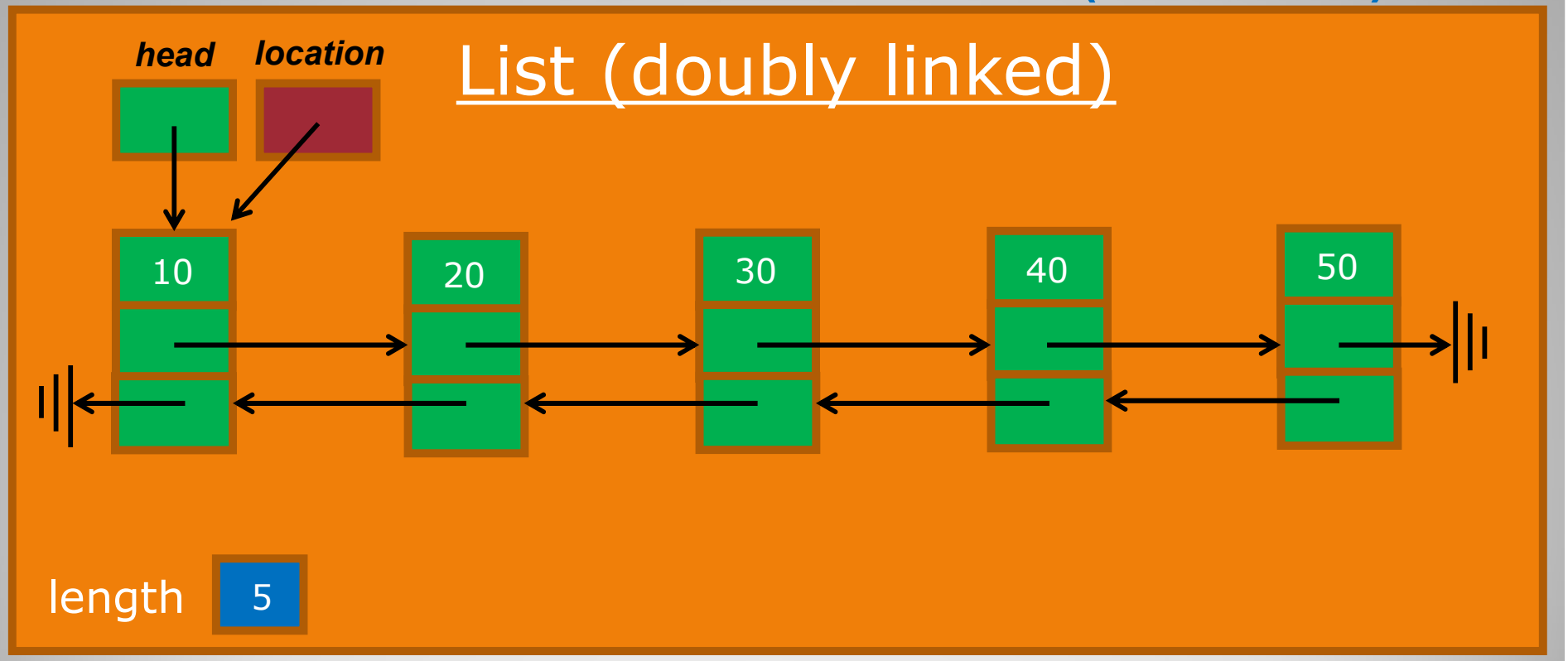


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

The if evaluates to true

Target to delete is 10 (head item)

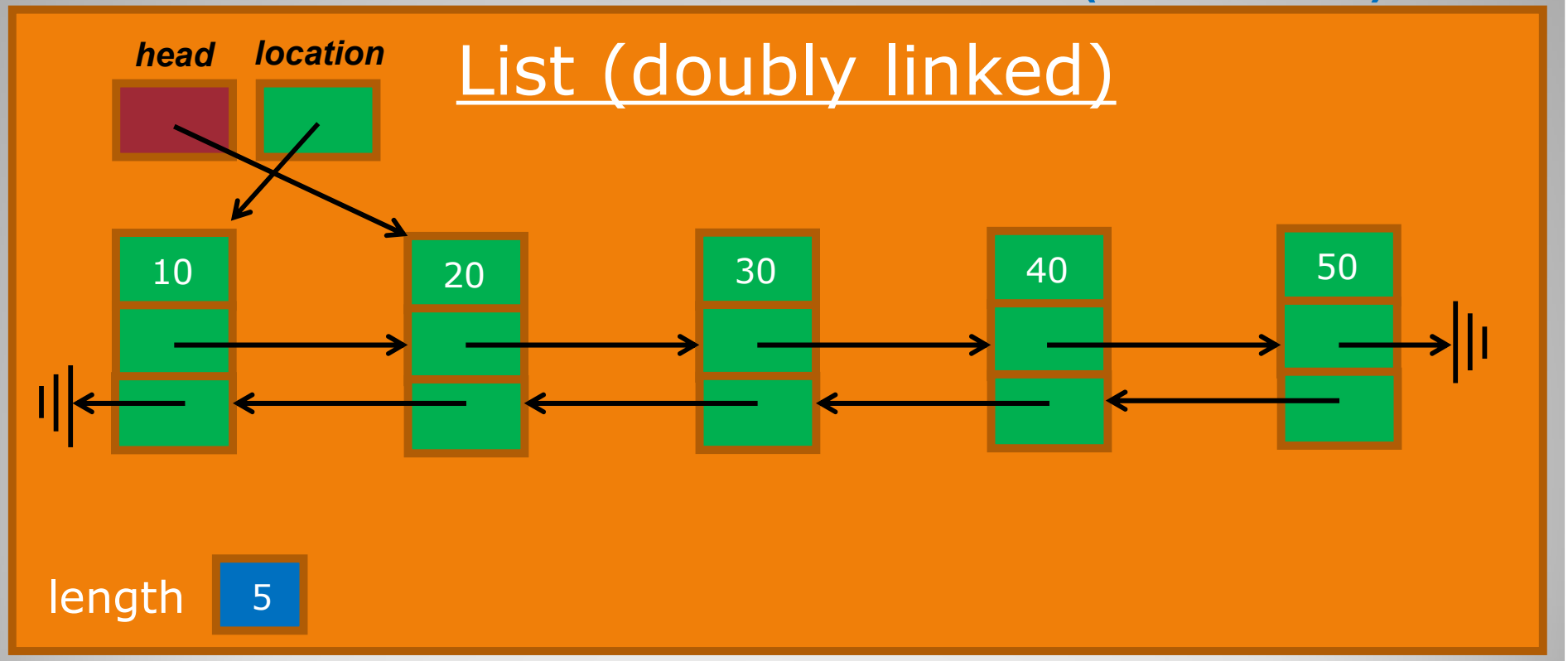


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

Move head to next node

Target to delete is 10 (head item)

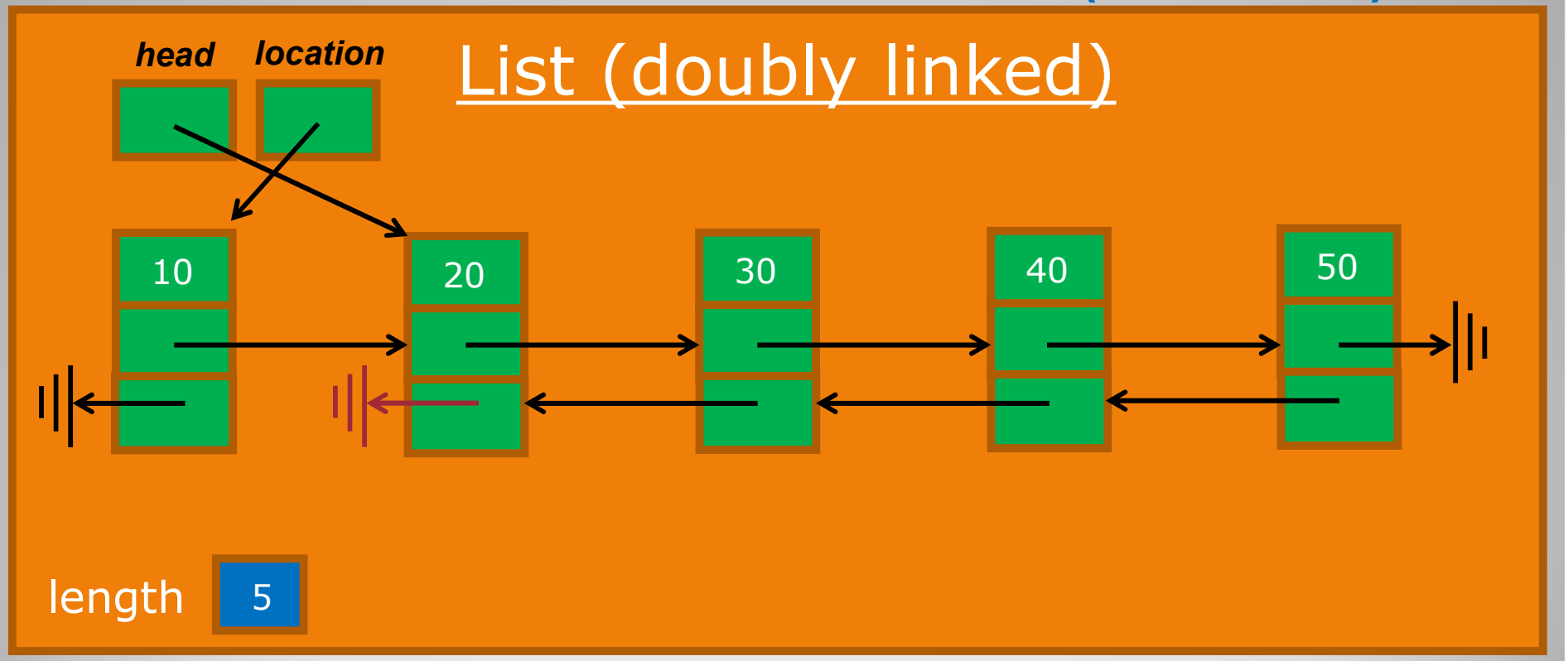


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

Set previous of new head to null

Target to delete is 10 (head item)

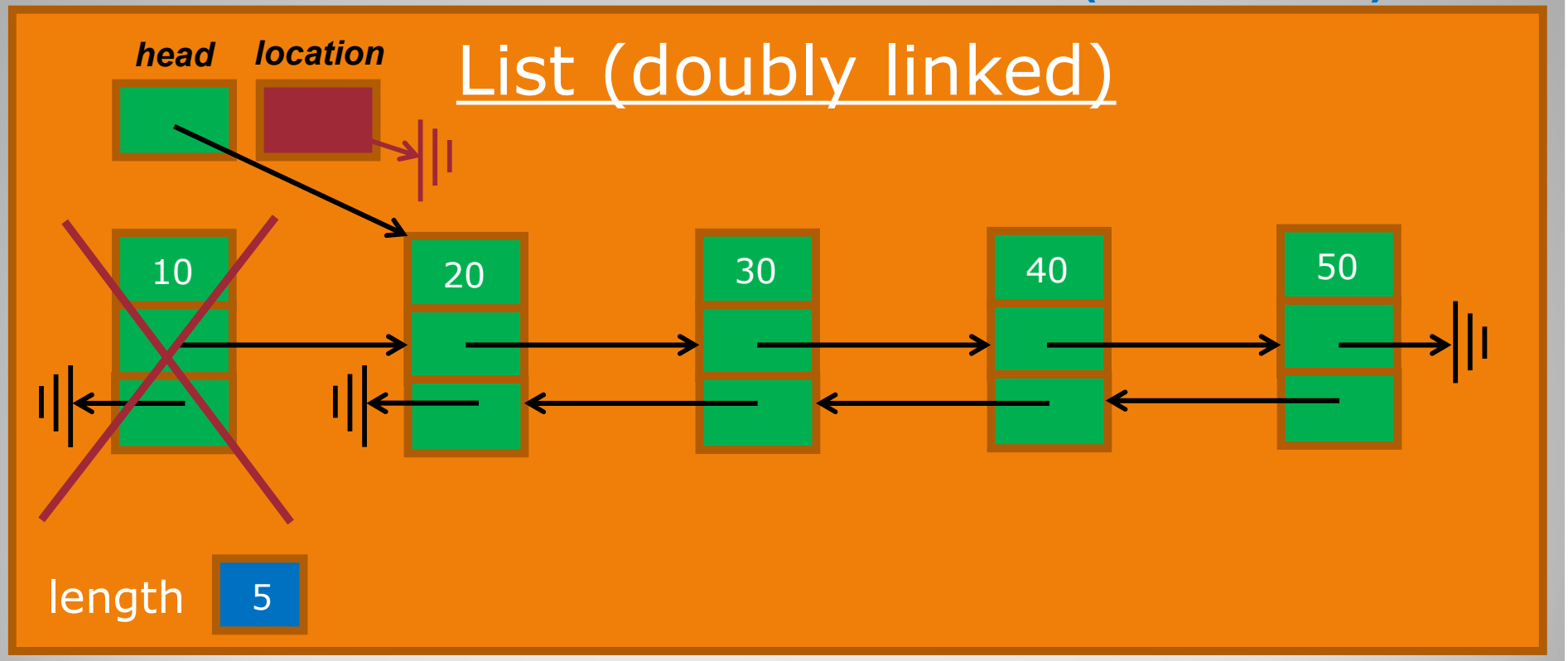


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

Makes the node a candidate for garbage collection

Target to delete is 10 (head item)

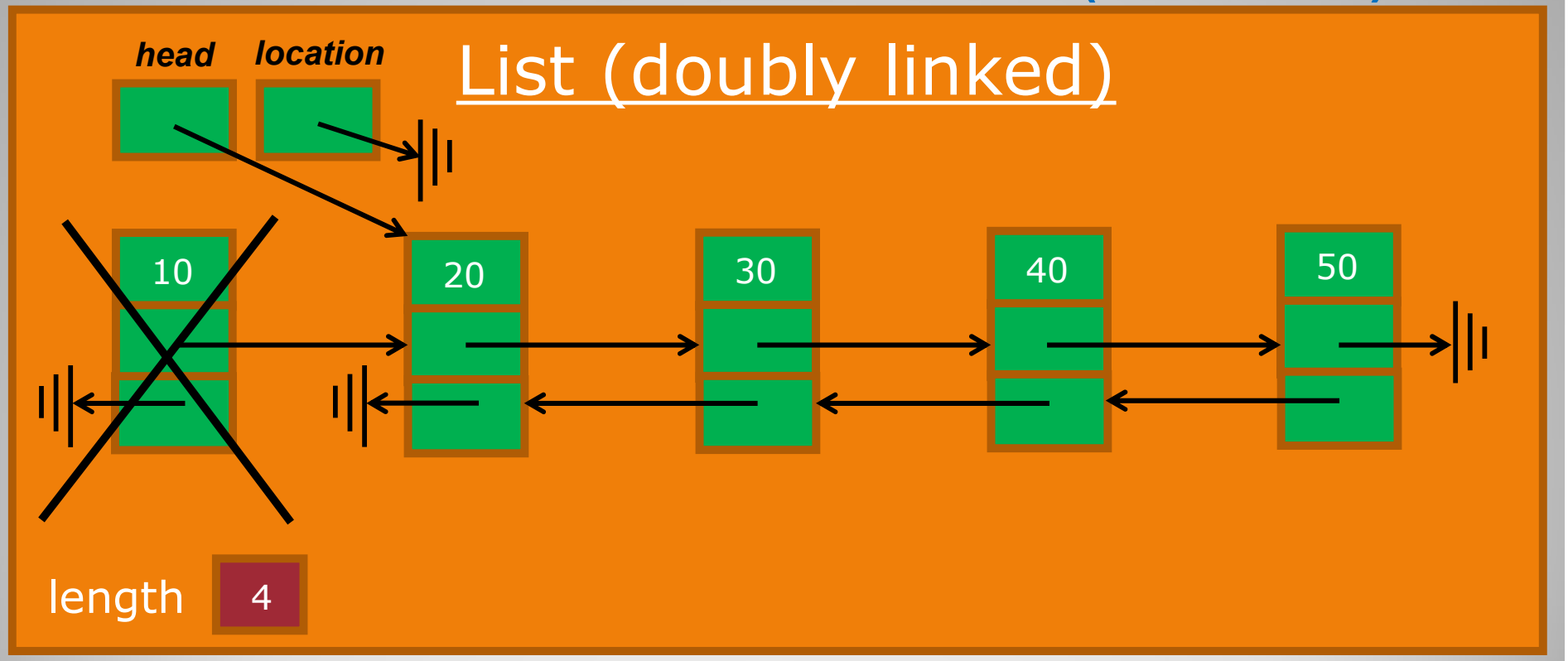


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

Decrement the length

Target to delete is 10 (head item)

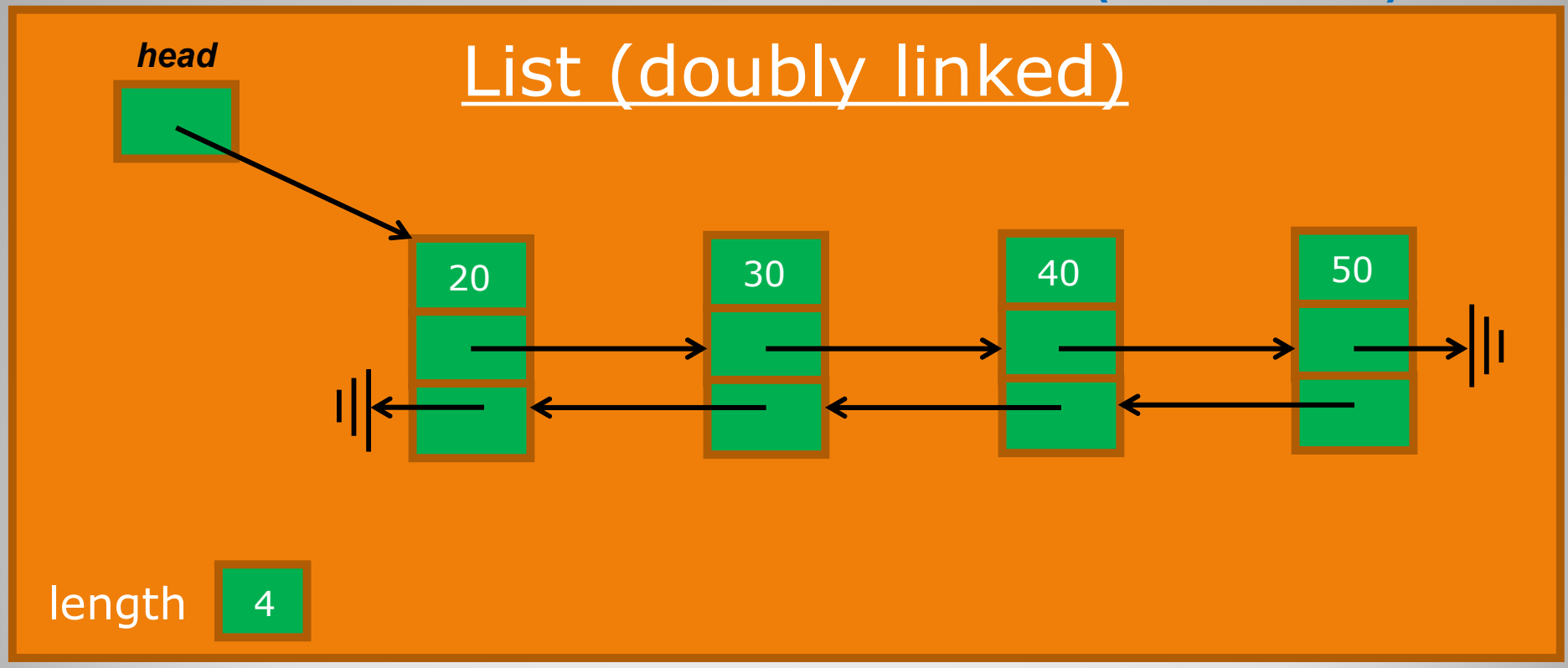


List (doubly linked) - deleteItem

```
Node location = head
if (item equals location.data)
        head = head.next
        head.prev = null
        location = null
        decrement length
```

This is what list looks like after the deletion is complete

Target to delete is 10 (head item)



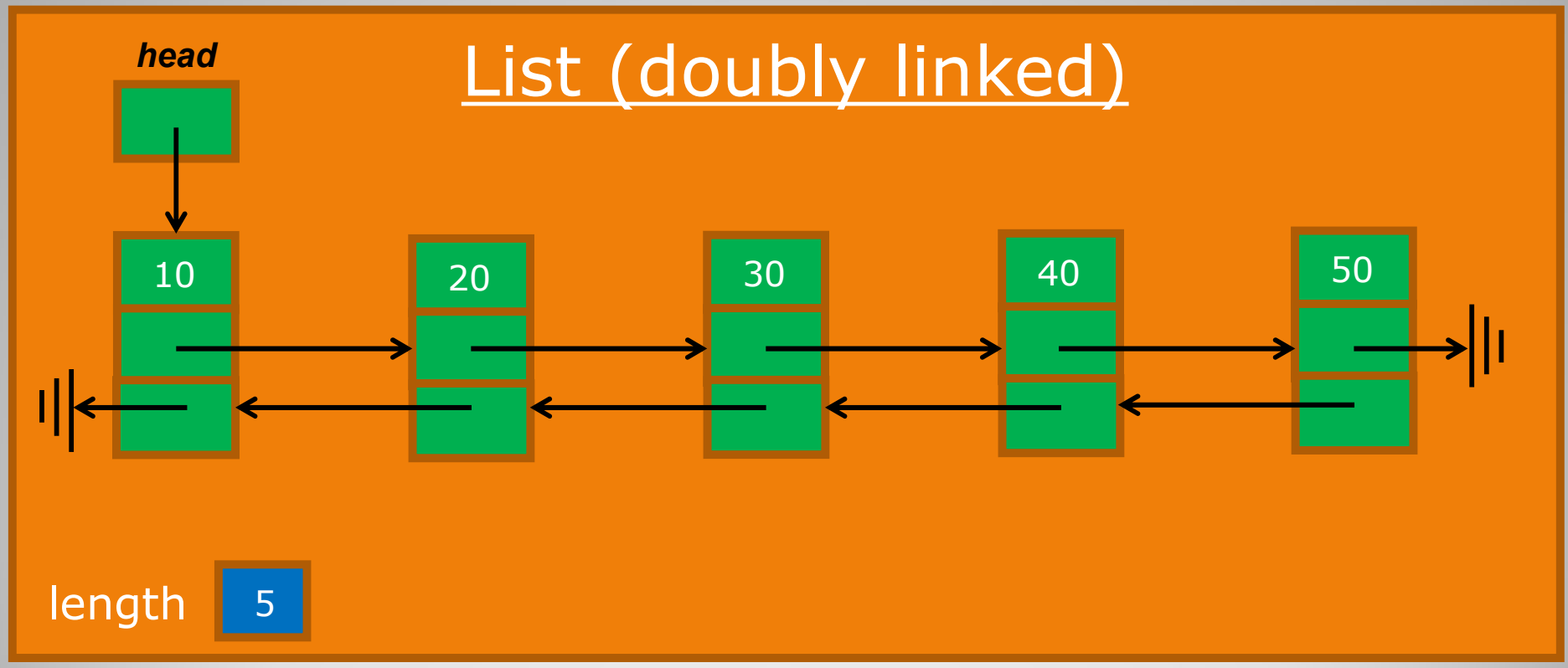
List (doubly linked) - deleteItem

Now delete an item from the middle of the doubly-linked list…

List (doubly linked) - deleteItem

We will now delete 30 from the list (30 is in middle of the list)

ol.deleteItem(30)

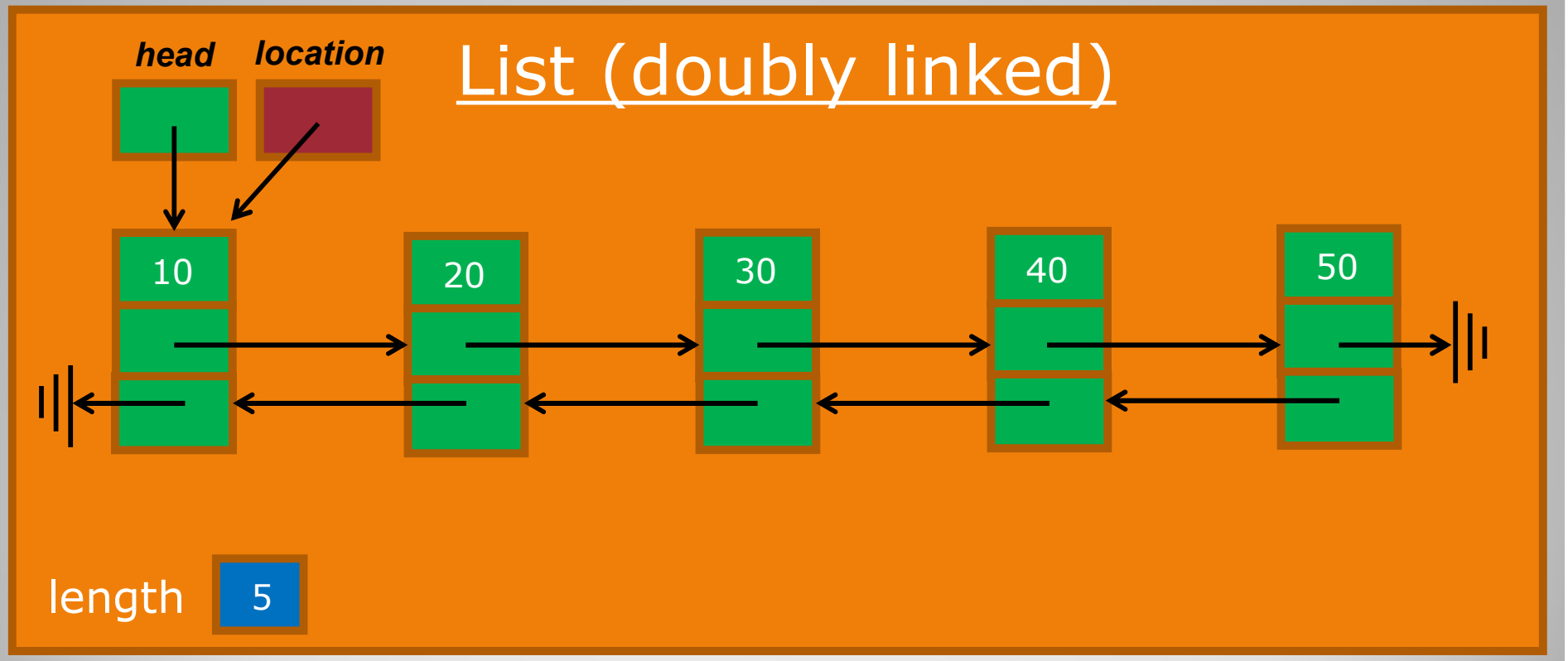


List (doubly linked) - deleteItem

location = head
while ((location != null) and (item != location.data))
location = location.next
endWhile

Target is 30

Set location to head

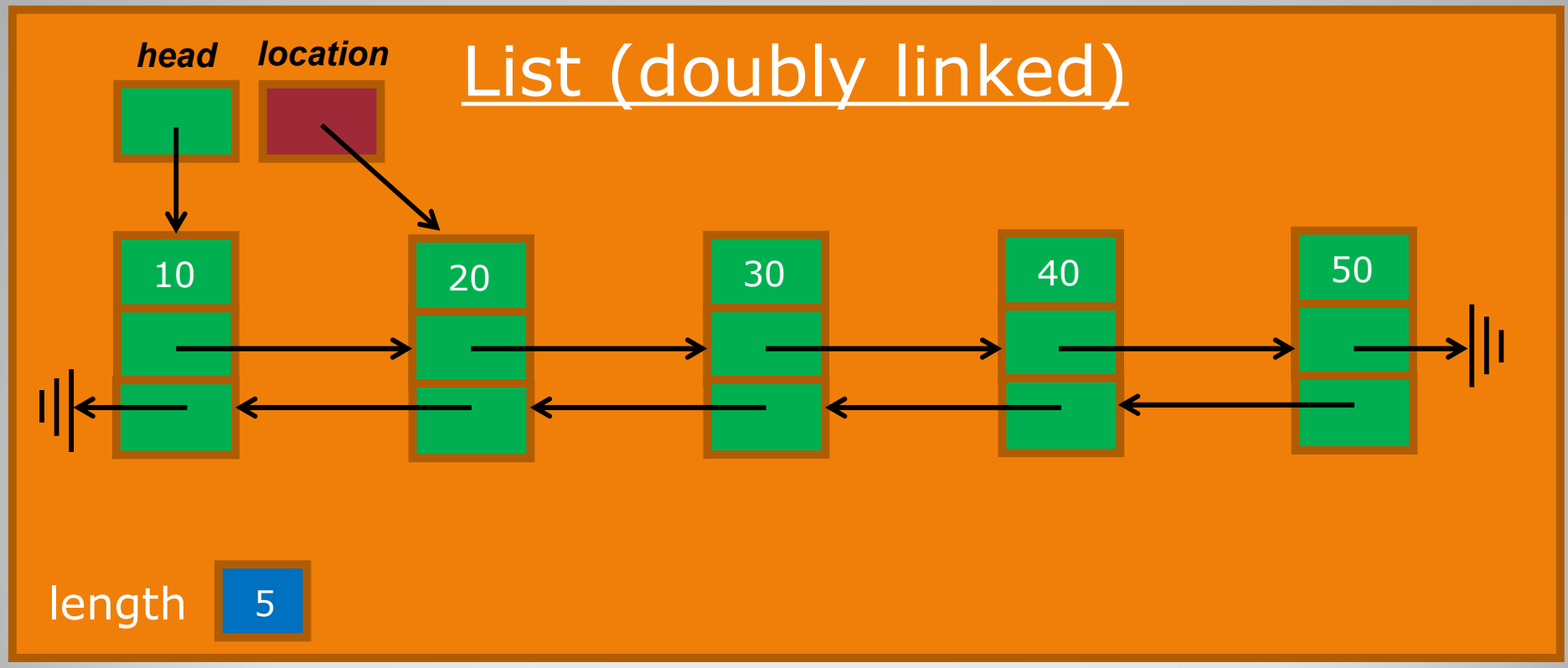


List (doubly linked) - deleteItem

```
location = head
while ( (location != null) and (item != location.data))
      location = location.next
endWhile
```

Target is 30

Keep moving location to the next node while not at the target

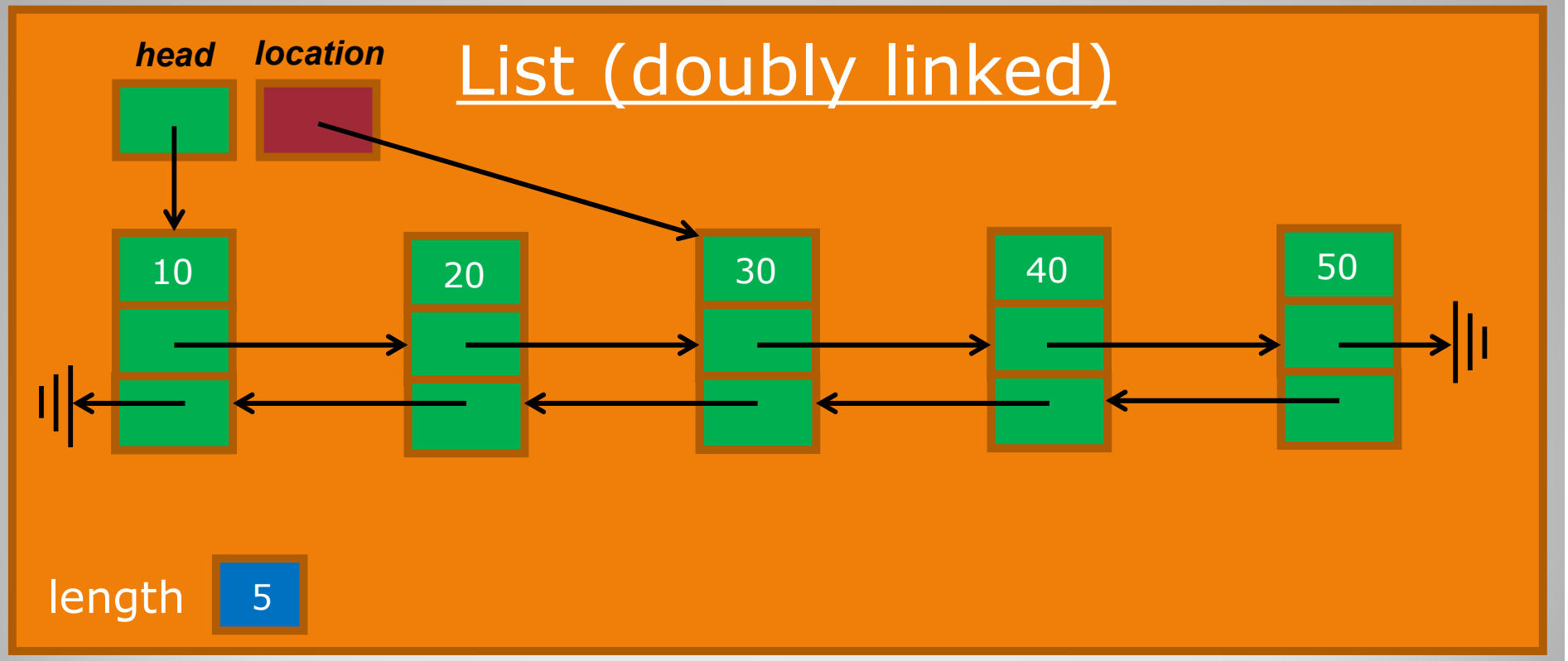


List (doubly linked) - deleteItem

```
location = head
while ( (location != null) and (item != location.data))
      location = location.next
endWhile
```

Target is 30

Keep moving location to the next node while not at the target

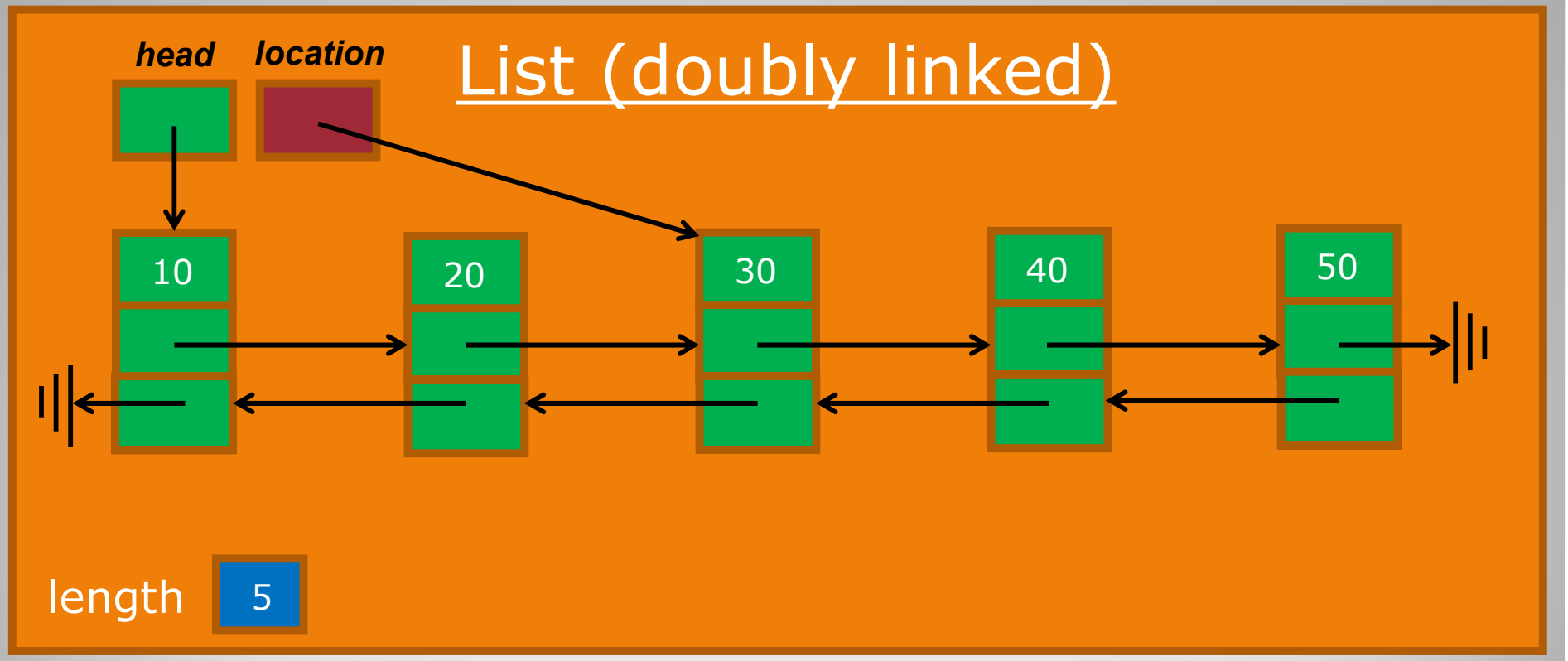


List (doubly linked) - deleteItem

if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length

Target is 30

If location is null then the target is NOT in the list

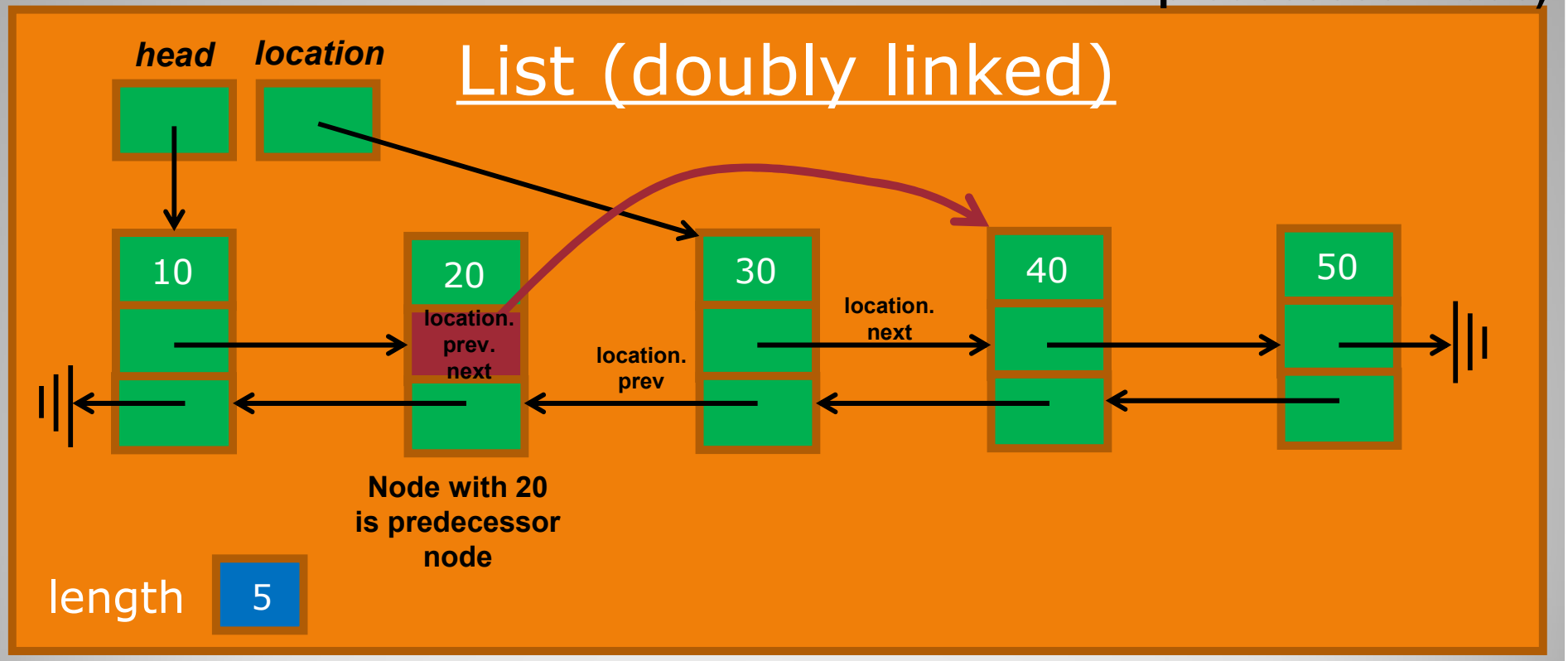


List (doubly linked) - deleteItem

Target is 30

if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length

Update next of predecessor node (location.prev is predecessor node)

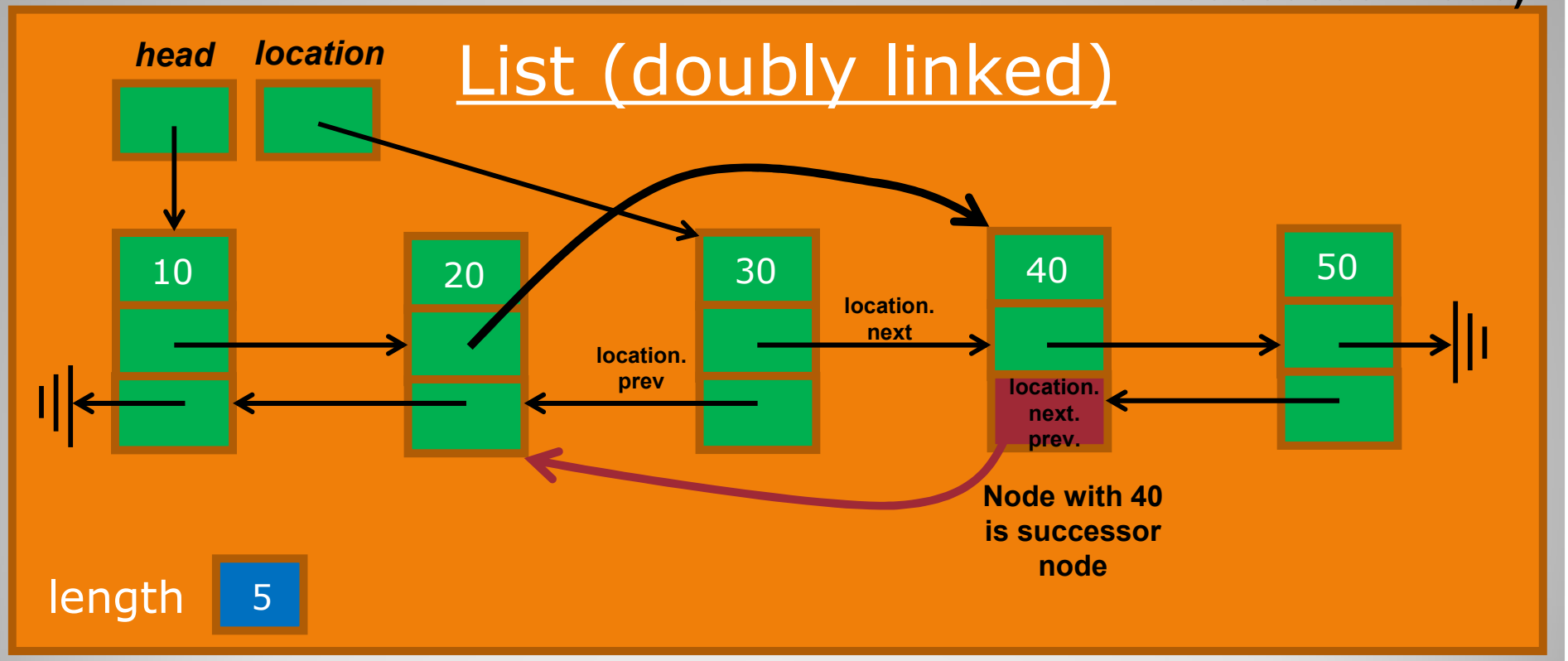


List (doubly linked) - deleteItem

if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length

Target is 30

Update prev of successor node (location.next is successor node)

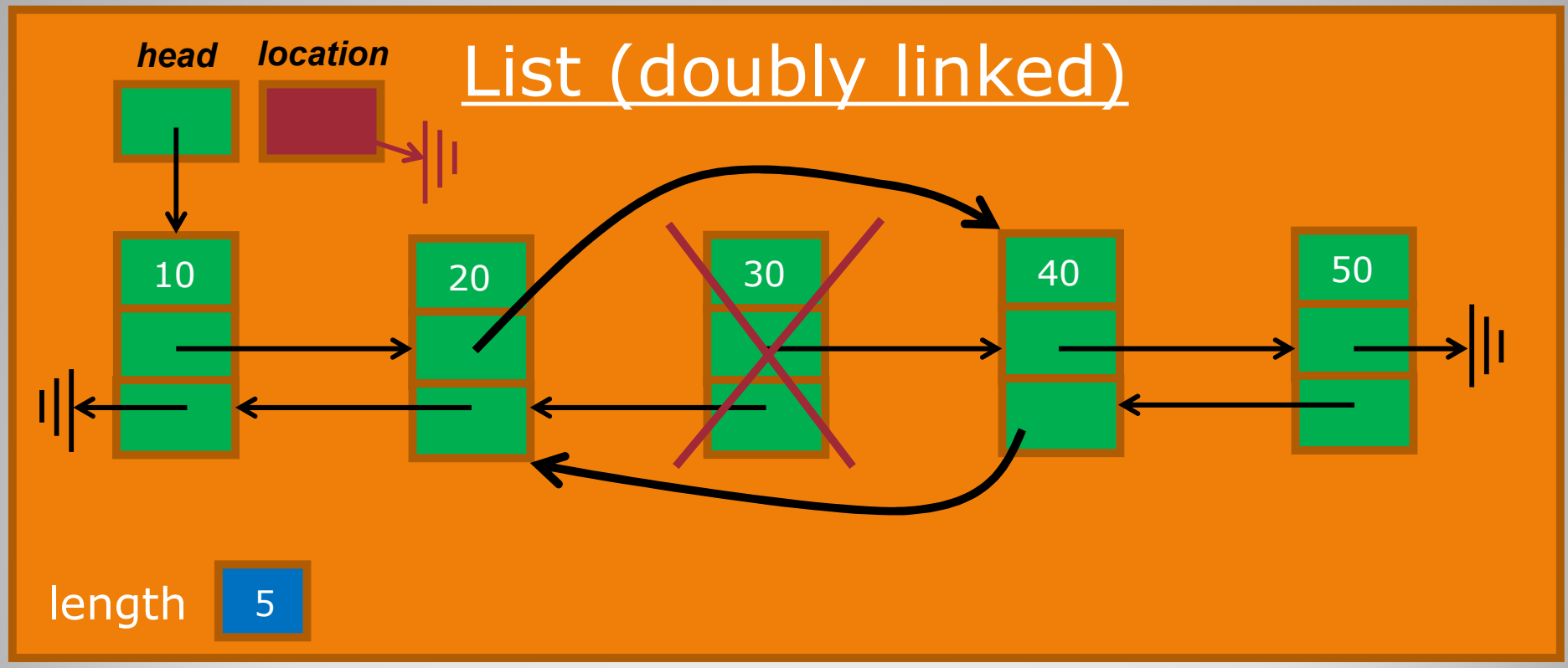


List (doubly linked) - deleteItem

if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length

Target is 30

Makes the node a candidate for garbage collection

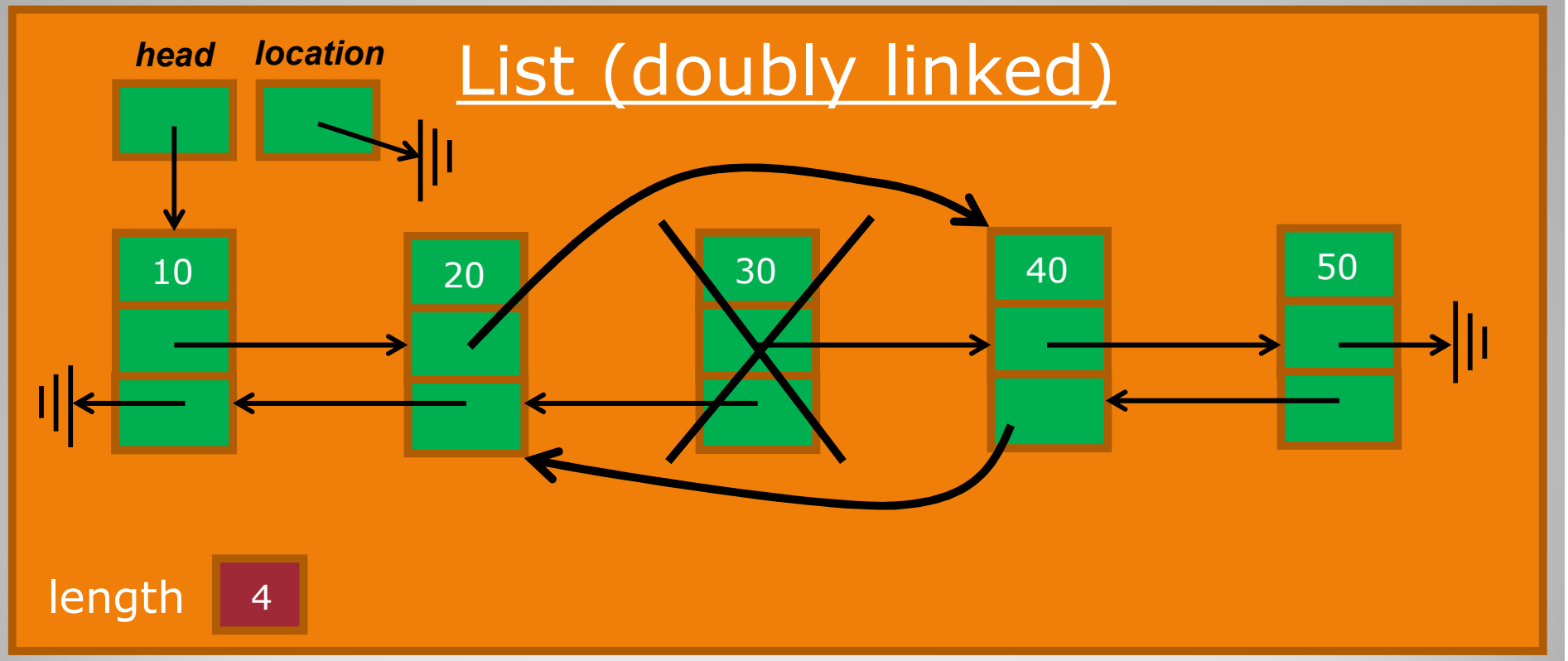


List (doubly linked) - deleteItem

```
if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length
```

Target is 30

Decrement the length

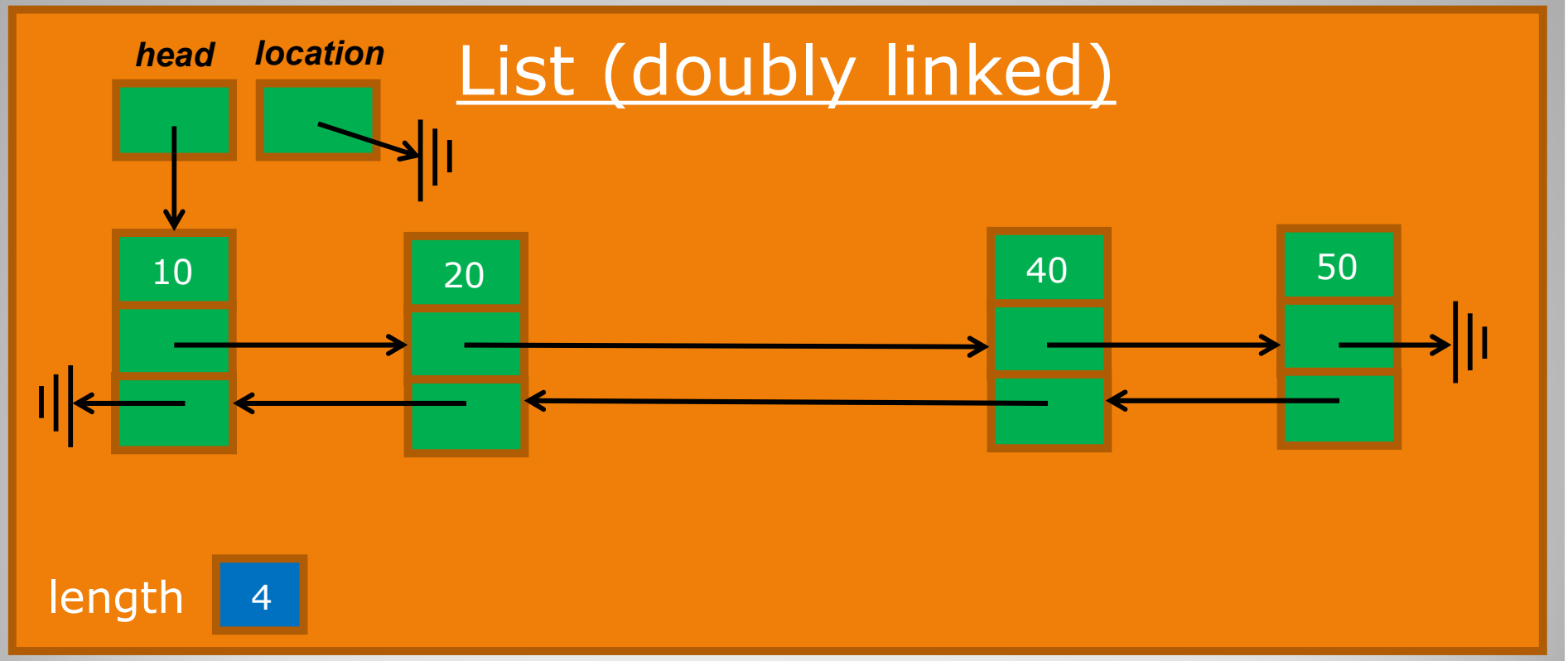


List (doubly linked) - deleteItem

```
if (location == null) then return
if (location.prev != null) then location.prev.next = location.next
if (location.next != null) then location.next.prev = location.prev
location = null
Decrement length
```

Target is 30

This is what list looks like after the deletion is complete



List (doubly linked) - deleteItem

Ordered List – makeEmpty

```
makeEmpty()
   Set head to null
   Set length to 0
```

Same as for singly-linked list

```
void makeEmpty()
    Declare Node temp
    while  head not equal to null
           Set temp to head
           Set head to head.next
           Set temp to null
    end While

    Set length to 0
```

Below is a slower version. It explicitly sets all nodes to null. This is unnecessary since the garbage collection will find those nodes for us.

Now we will move on to iterators…

Iterators

IteratorForwardBackward Interface

- We will use our own iterator interface.
- The iterator will be built into the class (OrderedList can implement this interface).

```
public interface IteratorForwardBackward {
    int iterGetData();
    void iterMoveNext();
    void iterMoveStart();
    void iterMovePrev();
    void iterMoveEnd();
    boolean iterIsValid();
}
```

```
iterMovePrev()
    if (iter not equal null)
        Set iter to iter.prev



iterMoveEnd()
    Set iter to head
    while ((iter not equal to null) and (iter.next not equal to null))
        Set iter to iter.next
```

Iterator Methods for Traversing in Reverse

```
// Code to declare and populate list goes here…

ol.iterMoveEnd()
while (ol.iterIsValid() equals true)
    Print ol.iterGetData()
    ol.iterMovePrev()
```

Put the iterator at the end of the list

Keep going while iterator is valid

Print the data retreived using the iterator

Go to the previous node

Iterator – Using the iterator in reverse

Now we will finish with Big-O…

Big-O Comparison

- It is important to know the approximate runtime cost operations when you create a data structure.
- What are the Big-O runtimes for the list implementations?

Big-O Comparison

Operation	Cost
makeEmpty	???
isFull	???
getLength	???
hasItem	???
retrieveItem	???
insertItem	???
deleteItem	???

Big-O Comparison – Ordered List (Doubly Linked-list)

Operation	Cost
makeEmpty	O(1)
isFull	O(1)
getLength	O(1)
hasItem	O(n)
retrieveItem	O(n)
insertItem	O(1)
deleteItem	O(n)

Big-O Comparison – Ordered List (Doubly Linked-list)

- **End of Slides**

End of Slides