Doubly Linked List (Implementation)

constructor, add a node at the beginning and in the center, remove a node

First of all, just like a singly linked list, we need to write a constructor function to create new nodes.

```
function Node(data) {
  this.data = data;
  this.next = null;
  this.previous = null;
}
```

By default, the node's previous and next values equal to null.

The linked list will again be a class. This class will have several properties, just like the singly linked list, such as the remove function, add function, and find function. We create a constructor with properties that every list has by default.

```
class DoublyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
    this.length = 0;
  }
}
```

Just like the singly linked list, we start off with the tail and head value equal to null: the list doesn't have any nodes yet, until we add them.

The function to **add** a node to the tail is as follows:

```
addNode(data) {
  const node = new Node(data);
  if (!this.head) {
    this.tail = node;
    this.head = node;
  } else {
    node.previous = this.tail;
    this.tail.next = node;
    this.tail = node;
  }
  this.length++;
}
```

First, we need to create a new node. As the **Node** function is a constructor function, we do this by typing **new Node(data)** with the data passed as an argument to **DoublyLinkedList**. If there is no head in the list, meaning that there are no nodes at all, the new node is both the head and the tail. The list would consist of only the new node!

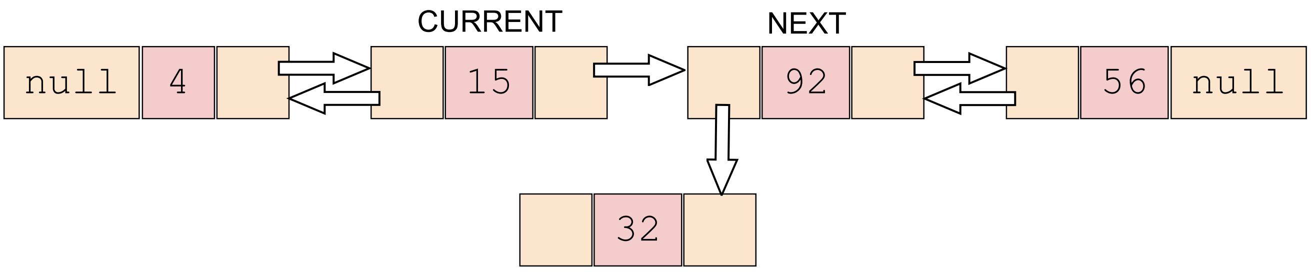
```
const list = new DoublyLinkedList();
list.addNode(23);
```

```
{
   data: 23,
   previous: null,
   next: null
}
```

There's only one node in the list, which represents both the head and the tail. It's almost the same as the singly linked list; the only difference is that we have to define the node's previous value in case the node isn't the head!

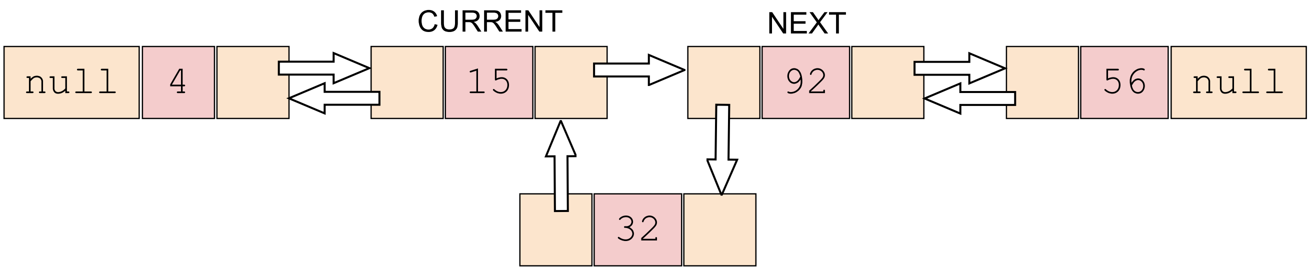
```
insertAfter(data, toNodeData) {
  let current = this.head;
  while (current) {
    if (current.data === toNodeData) {
      const node = new Node(data);
      if (current === this.tail) {
        this.addNode(data);
      } else {
        current.next.previous = node;
        node.previous = current;
        node.next = current.next;
        current.next = node;
        this.length++;
      }
    }
    current = current.next;
  }
}
```

Again, we start traversing the list from the head. While there is a head, meaning that the list isn't empty, we check whether the currently checked node's value is equal to the node after which we want to insert the new node. If we find that node and it happens to be the tail, we invoke the **addNode** function as that function adds nodes to the end of the list, so we won't have to repeat that logic. However, if the node is somewhere in the middle of the list (for example, if **data** is equal to **32** and **toNodeData** is **15**) this is how we handle the situation:



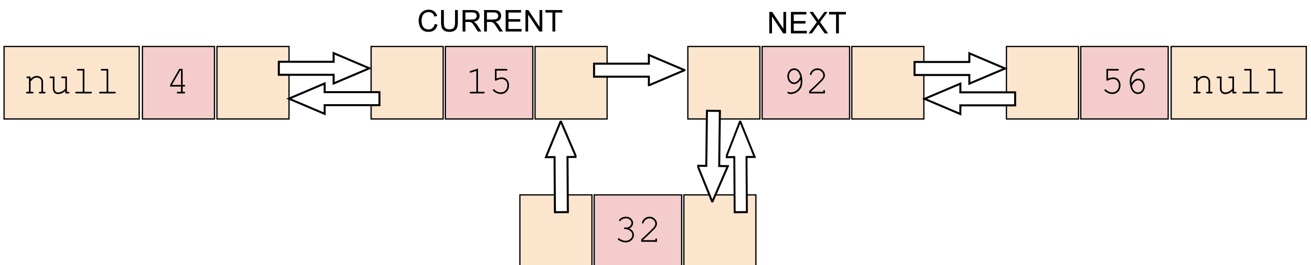
Set the currently checked node's next value's previous value (yes, bear with me) equal to the new node.

1 of 5



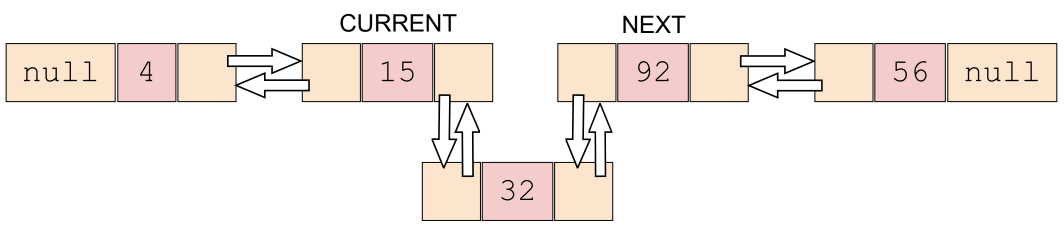
Then, we set the new node's previous value equal to the current node's next value.

2 of 5



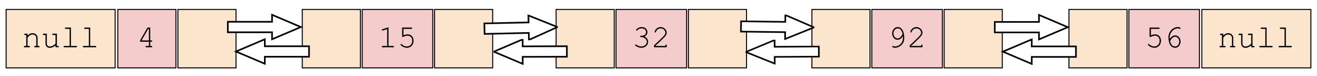
Then, we set the node's next value equal to the current next value.

3 of 5



Then, we set the node's next value equal to the current next value.

4 of 5



The node has been inserted!

5 of 5

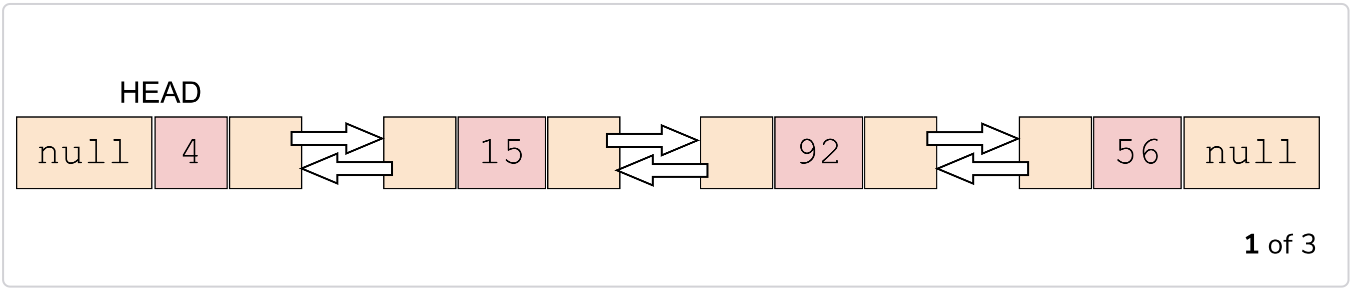
And in the end, we set the current node's next value equal to the new node. We've successfully inserted the new node in the list!

Removing a node:

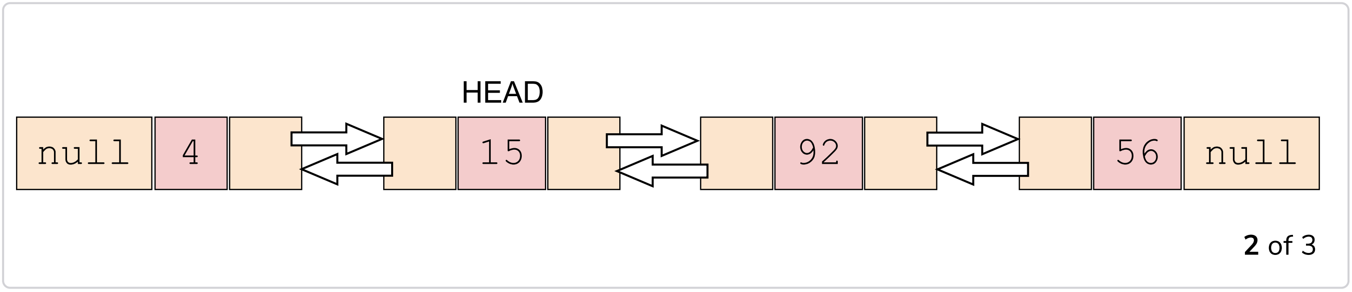
```
removeNode(data) {
  let current = this.head;
  while (current) {
    if (current.data === data) {
      if (current === this.head && current === this.tail) {
        this.head = null;
        this.tail = null;
      } else if (current === this.head) {
        this.head = this.head.next;
        this.head.previous = null;
      } else if (current === this.tail) {
        this.tail = this.tail.previous;
        this.tail.next = null;
      } else {
        current.previous.next = current.next;
        current.next.previous = current.previous;
      }

      this.length--
    }
    current = current.next;
  }
}
```

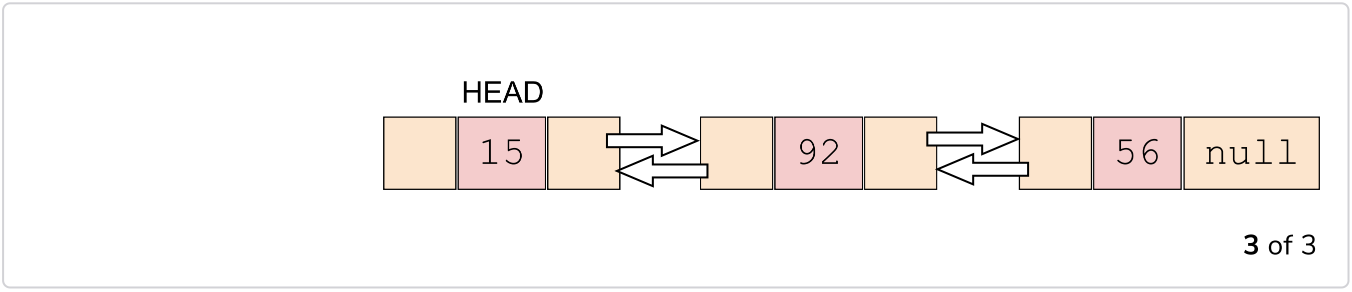
If the node we want to remove is the head, we first set the head's next node equal to the head. Then, we set the new head's previous node equal to null, which results in the deletion of the previous head node.



1 of 3



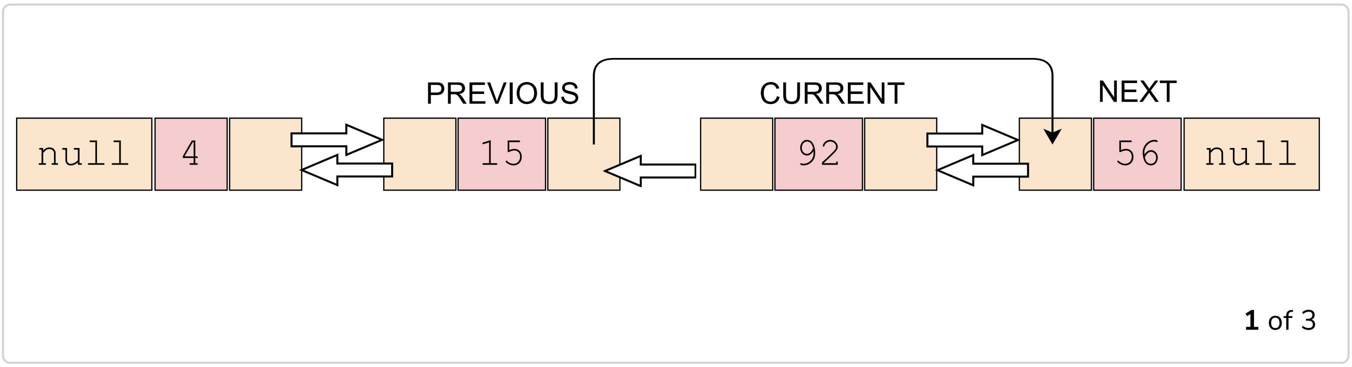
2 of 3

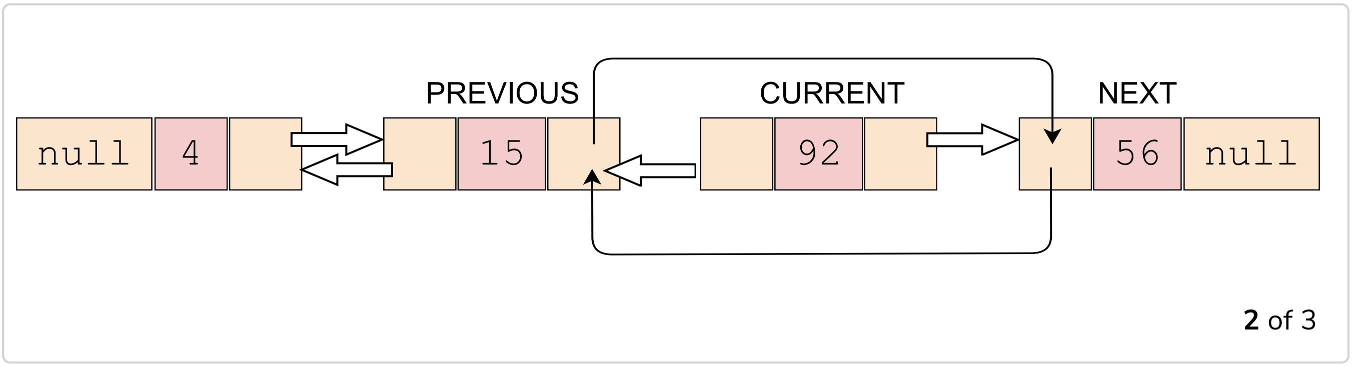


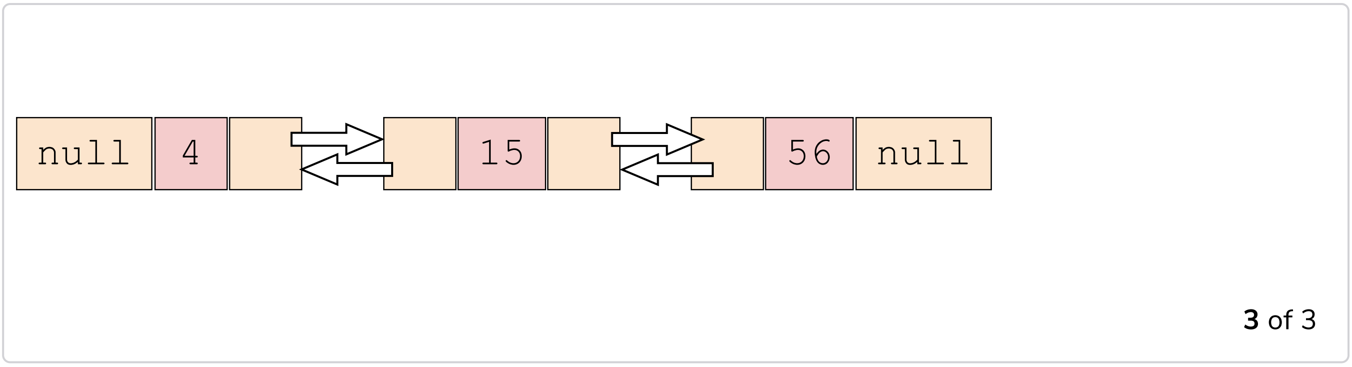
3 of 3

The same logic applies to delete the tail node, just the other way around.

If the node we want to delete is somewhere in the middle, we set the previous node's next value equal to the current's next node. Then, we set the next node's previous value equal to the previous node's next value, which results in the deletion of the current node.







In the next lesson, I will talk about the time complexity of the various functions of singly and doubly linked lists.