Singly Linked List (Implementation)

create a linked list, add a node, and add and remove data to a linked list (Reading time: 4 minutes)

In a singly linked list, we should be able to find, remove, and insert nodes. Before we can make a list, we first need to create the nodes.

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

We now created a constructor function that we can use each time we create a new node. By default, the new node's next value is null, and its data is equal to the data we pass as an argument.

The linked list will be a **class**. This class will have several properties, such as the remove function, add function, and find function. However, before we can do any of that, we need to create its constructor.

```
class SinglyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }
}
```

By default, the list doesn't have any nodes, and the length is equal to 0. If the list doesn't have any nodes, both the head (the first node in the list) and the tail (the last node in the list) don't exist, so their values are equal to **null**.

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

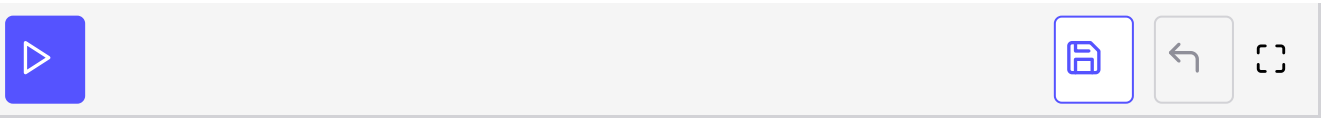
```
addNode(data) {
  const node = new Node(data);
  if (!this.head) {
    this.tail = node;
    this.head = node;
```

```
  } else {
    this.tail.next = node;
    this.tail = node;

  }
}
```

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 **SinglyLinkedList**. If there is no head in the list, meaning that there are no nodes at all in the list, the new node is both the head and the tail. The list would consist of only the new node!

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

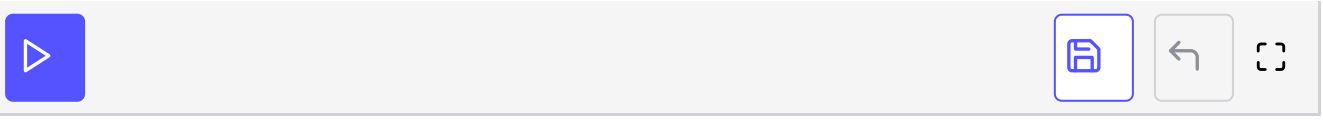


The above code snippet would result in:

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

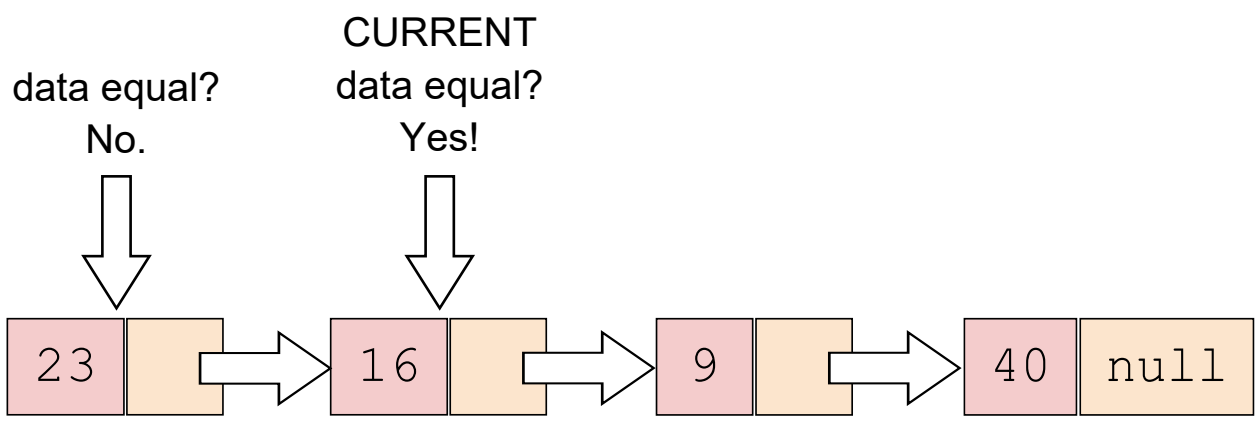
There's only one node in the list, which represents both the head and the tail.

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



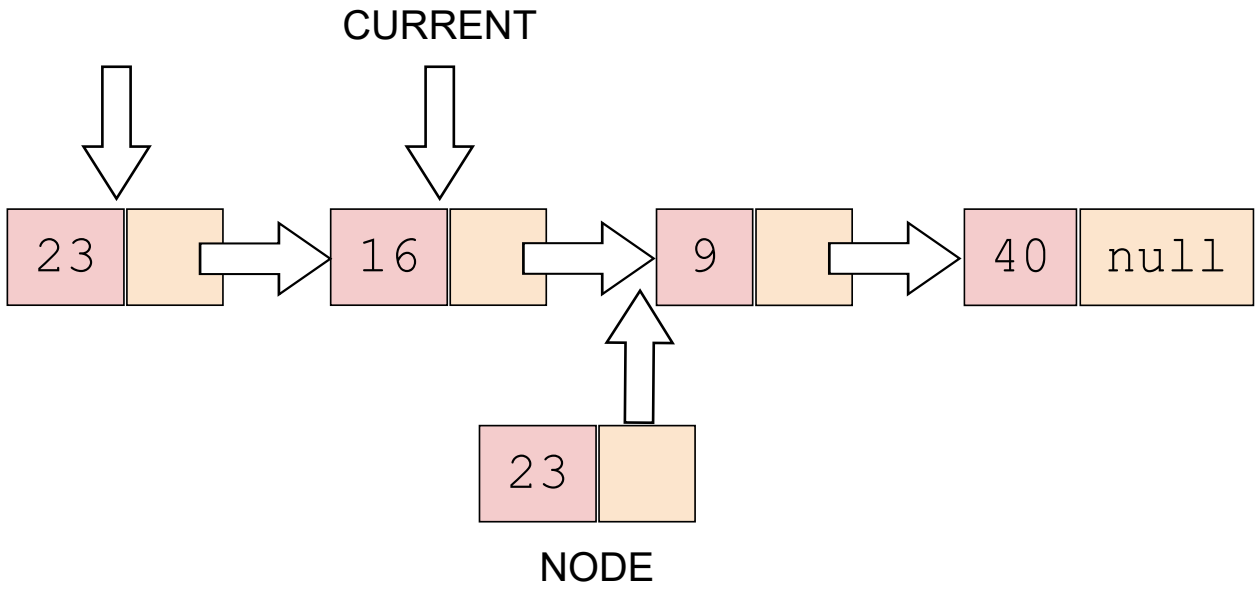
Inserting '23' after '16'

The **insertAfter** function receives two arguments: the data for the new node, and the data of the node after which we want to add the new node. As we start traversing the list again, we set the default value of the currently checked node to the head. While there is a head, meaning that the list isn't empty, we can start walking through the list. If the data of the currently checked node equals the data of the node we wanted to find, in order to add a new node after this node, we create the new node with the data we passed as an argument. If the currently checked node is the tail of the list, meaning that we're just adding a new node to the end of the list, the tail's next value is equal to the new node, and now the new node equals the tail. Else, the new node's next value equals the currently checked node's next value.

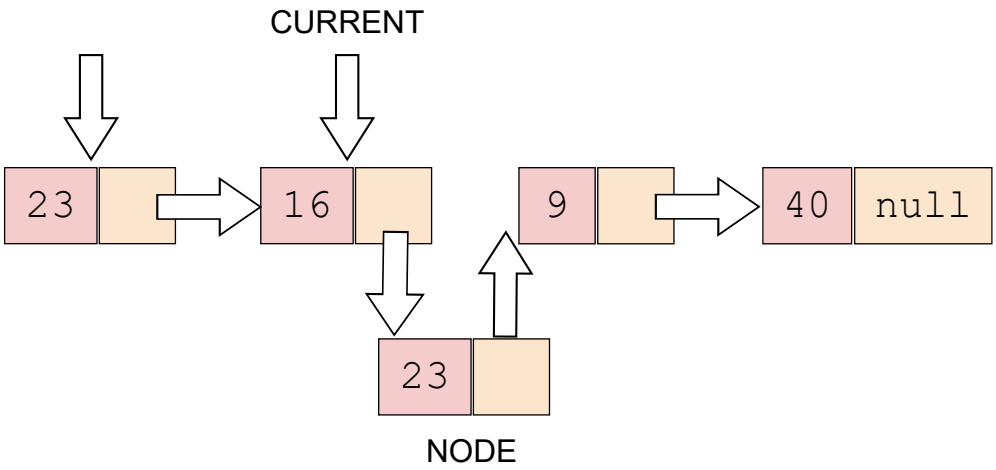


Filter through the list, until you find the node with the right data.

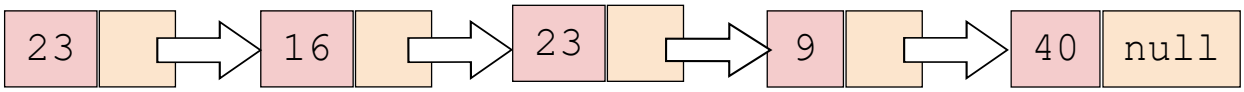
1 of 4



Create a new node, and set the new node's next value equal to the current node next value.



Set the current node's next value equal to the new node.



Now, the new node has been inserted!

4 of 4

The function to **remove** a node is as follows:

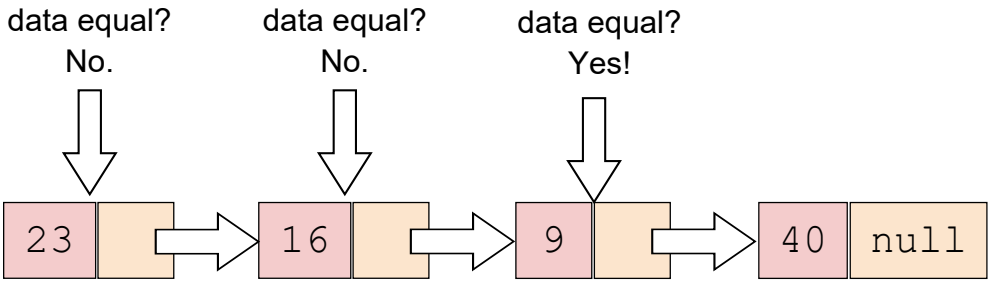
```
removeNode(data) {
  let previous = this.head;
  let current = this.head;
  while (current) {
    if (current.data === data) {
      if (current === this.head) {
        this.head = this.head.next;
      }
      if (current === this.tail) {
        this.tail = previous;
      }
      previous.next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

Removing '9' from the List

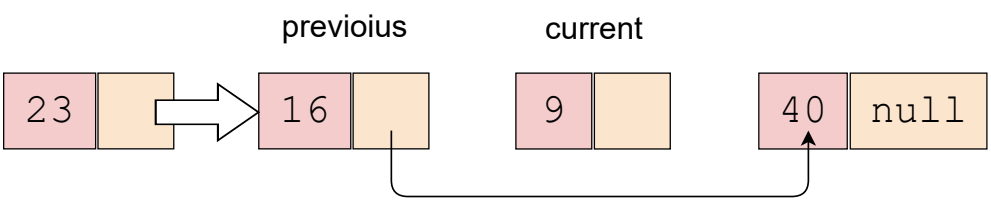
There are two variables, **previous** and **current**. **current** represents the

currently checked node, **previous** represents the currently checked node's previous node. To find the node we want to remove, we always start from the beginning of the list, the head. The values of the previous and current variables are now equal to the head. If there's a current value, meaning that the list consists of at least one node, we start to traverse the list. If the currently checked node's data is equal to the data we want to delete, we found the right node! Now, we need to check where this node is placed. Let's say that we want to remove the node with the data equal to **9**. We find the node, and set the previous node's next value equal to the next node.

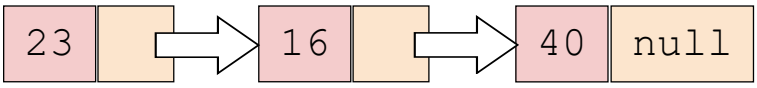
If the node we want to delete is the head of the list, we set the current head's node next value equal to **this.head**.



1 of 3



2 of 3



If we log this linked list, it would look like this:

```
{
   data: 23,
   next: {
     data: 16,
     next: {
        data: 9,
        next: {
          data: 40,
          next: null
        }
     }
  }
}
```

BEFORE DELETION

```
{
   data: 23,
   next: {
     data: 16,
     next: {
        data: 40,
        next: null
     }
  }
}
```

AFTER DELETION

In the next lesson, I will talk about the time complexity of the various functions of a linked list.