



ChatGPT ▾

Share ...

- New chat
- Search chats
- Library
- Sora
- GPTs
- Projects NEW

Chats

Remove linked list elements

Huzz meaning in memes

Middle node solution

Code review middle node

Explain error message

Array index increment difference

CA Aggregator

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.

Example 1:

Input: head = [1,2,6,3,4,5,6], val = 6

Output: [1,2,3,4,5]

Example 2:

Input: head = [], val = 1

Output: []

Example 3:

Input: head = [7,7,7,7], val = 7

Output: []

Constraints:



1 <= Node.val <= 50

0 <= val <= 50

```
/*
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *   this.val = (val===undefined ? 0 : val)
 *   this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @param {number} val
 * @return {ListNode}
 */
var removeElements = function(head, val) {
  let current = f ↓ ;
  let previous = null;

  if(head == null) {
    return [];
  }

  while(head.val == current.val){
    current = current.next;
    if(current == null){
      return [];
    }
  }

  while(current){
    if(current.val == val){
      current = current.next;
      previous.next = current;
    } else{
      let nextCurrent = current.next;
      current = nextCurrent;
      previous.next = current;
    }
    previous = current;
  }
  return head;
};
```

Good question 👍 This is a classic linked list problem.

Your code is almost there, but it has some logic issues:

Issues in your code

1. Returning `[]` instead of `null`
 - In linked lists, if the list is empty, we return `null`, not an empty array.
2. While loop to skip values at head is incorrect



- You wrote `while(head.val == current.val)` but what you really want is `while head itself has the`

3. Pointer updates are wrong

- You are setting `previous.next = current` in both cases, which can break links unnecessarily.
- You only need to adjust `previous.next` when deleting a node.

Correct Approach

- First, move `head` forward until it is not equal to `val` (because `head` itself may need removal).
- Then traverse using two pointers:
 - `current` → the node being checked
 - `previous` → the last node that is **not deleted**
- If `current.val == val`, remove it by doing `previous.next = current.next`.
- Else, move `previous` forward.

Fixed Code

```
javascript
var removeElements = function(head, val) {
  // Step 1: remove nodes from the beginning if they match val
  while (head !== null && head.val === val) {
    head = head.next;
  }

  // Step 2: traverse the list
  let current = head;
  let previous = null;

  while (current !== null) {
    if (current.val === val) {
      // remove current
      previous.next = current.next;
    } else {
      previous = current;
    }
    current = current.next;
  }

  return head;
};
```

Dry Run Example

Input: `head = [1,2,6,3,4,5,6], val = 6`

- Remove from beginning? No (`head = 1`).
- Traverse:
 - `1` → keep
 - `2` → keep
 - `6` → delete → link `2 -> 3`
 - `3` → keep
 - `4` → keep
 - `5` → keep
 - `6` → delete → link `5 -> null`

⚡ This is the **standard solution**.

Would you like me to also show you a **dummy node approach** (a cleaner trick where you don't need to handle `head` separately)?

🔗 ⏪ ⏴ ⏵ ⏹ ⏸ ⏹ ⏺

difference between double equal to and triple equal to?

Great question! Let's break it down clearly 🤓

In JavaScript, both `==` (double equal) and `===` (triple equal) are **comparison operators**, but they behave differently.

1. `==` (Double Equal → *Loose Equality*)

- Compares values only, not types.
- If the types are different, JavaScript tries to convert (coerce) one type into another before comparing.

Example:

```
js Copy code
```

```
5 == "5"    // true → because "5" is converted to number
0 == false  // true → because false is converted to 0
null == undefined // true → they are considered loosely equal
```

2. === (Triple Equal → Strict Equality)

- Compares both value and type.
- No type conversion is done. If the types differ, it's immediately **false**.

Example:

```
js Copy code
```

```
5 === "5"    // false → number vs string
0 === false  // false → number vs boolean
null === undefined // false → different types
5 === 5      // true
```

👉 Rule of Thumb

- Use **==** (strict equality) almost all the time because it avoids hidden bugs caused by type coercion.
- Use **==** only when you intentionally want type conversion (rare in real-world code).

👉 Do you want me to also show you a table of common tricky cases (**==** vs **===**) so you can memorize the differences better?

Copy Down Up Left Right More



```
while (head != null && head.val === val) {
    head = head.next;
}
```

Is it enough to move the head to the next node if the head's val is the same as val in the question? My doubt is that the head that we currently skip will still point to our next node, shouldn't we cut that link or something?

Excellent doubt! — this shows you're really thinking about how linked lists work under the hood.

Let's go step by step!

1. What happens when we do



```
while (head != null && head.val === val) {
    head = head.next;
}
```

Suppose the list is:

```
bash Copy code
```

```
head → [7] → [7] → [3] → [4]
val = 7
```

- First iteration: `head.val === 7` → move `head = head.next`.
Now `head` points to the second `[7]`.
- Second iteration: still `7`, so `head = head.next`.
Now `head` points to `[3]`.

At this point, we've lost references to the first two `[7]` nodes.



2. Do we need to cut the links manually?

Not really ✗. Why?

- JavaScript (and most modern languages) has **garbage collection**.
- When nothing references those skipped nodes anymore, they're automatically cleaned up.
- So by simply moving `head`, you effectively "remove" the nodes from the list.

3. Why it works safely

Remember, the linked list is defined by what your `head` points to.

- If you move `head` forward, the old nodes are no longer part of your list.
- Even if their `next` pointers still exist, they don't matter because your program can't reach them anymore.

4. Visualization

Bef

css

Copy code

head → [7] → [7] → [3] → [4]



After skipping:

css

Copy code

head → [3] → [4]

The [7] → [7] chain is still in memory for a moment, but unreachable → GC will delete them.

So yes — just updating `head = head.next` is enough.

You don't need to explicitly "cut the links" unless you're coding in a manual memory management language (like C).

frees memory explicitly?

🔗 ⌂ ⌂ ⌂ ⌂ ⌂ ⌂ ⌂

```
var removeElements = function(head, val) {
  let sentinel = new ListNode(0, head);
  let current = sentinel;
  while (current && current.next) {
    if (current.next.val === val) {
      current.next = current.next.next;
    } else {
      current = current.next;
    }
  }
  return sentinel.next;
};
```



Explain the sentinel approach for a beginner, I'm unable to understand it at all

Perfect — you picked up the **sentinel (dummy node)** approach, which is one of the cleanest ways to solve this problem 🍏

Let's go step by step as if you're seeing it for the first time.

💡 Problem Recap

We want to remove nodes from a linked list if their value = `val`.

The hard part is when the `head` itself has to be removed.

Because if `head` gets deleted, the "entry point" of the ↴ changes.

That makes the code messy if you handle `head` separately.

🌟 The Sentinel (Dummy Node) Trick

The idea is:

- Create a fake node before the real head.
- This fake node is called a **sentinel** (or dummy node).
- Its `.next` points to the actual head.

This way, even if the real `head` gets deleted, we still have a "fixed handle" to the list.

1. Creating the sentinel

```
js Copy code
let sentinel = new ListNode(0, head);
```

Imagine the list:

```
head → [1] → [2] → [6] → [3]
```

After adding sentinel:

```
scss Copy code
sentinel(0) → [1] → [2] → [6] → [3]
```

Now we never lose track of the list, because sentinel is always safe at the front.

2. Traversing with `current`

```
js Copy code
let current = sentinel;
```

We start from sentinel instead of head.

- If `current.next.val === val` → delete that node.
- Else → move `current` forward.

3. Deleting a node

```
js Copy code
current.next = current.next.next;
```

This skips over the unwanted node.

Example:

```
css Copy code
current → [2] → [6] → [3]
```

```
ini Copy code
current.next = current.next.next;
```

Now:

```
css Copy code
current → [2] → [3]
```

4. Returning the new head

```
js Copy code
return sentinel.next;
```

Why `sentinel.next`?

- Because `sentinel` was just a fake node.
- The real list starts after it.
- If the original head was deleted, `sentinel.next` will correctly point to the new head.

🔑 Why Sentinel is Great

Without sentinel, you'd need special logic like:

- If head's value = val → move head
- Else → normal traversal

With sentinel:

- You treat head just like any other node.

- The code is cleaner and uniform.

Example Dry Run

Input: head = [6,1,2,6,3], val = 6

SCSS

 Copy code

```
sentinel(0) → [6] → [1] → [2] → [6] → [3]
```

- Start current = sentinel.
- Check current.next → value = 6 → remove it.

css

 Copy code

```
sentinel → [1] → [2] → [6] → [3]
```

- Move current to [1].
- Check next = [2] → keep.
- Move to [2].
- Next = [6] → remove it.

css

 Copy code

```
sentinel → [1] → [2] → [3]
```

Final list: [1,2,3]

👉 Does this explanation click now, or should I also draw a **step-by-step picture diagram** of pointer movements?



current.next = current.next.next;

Lets consider sentinel → 6 → 6 → 6 → 7 → 1 → null where head points to the first 6 and current points to sentinel just before head

I'm confused about this line. What does it do actually? The current never moves to the next node right? It always stays at the sentinel node. I'm so damn confused

Ahh, I see exactly where you're stuck — great catch 🎉

This current.next = current.next.next; line is a little tricky if you haven't visualized it carefully. Let's break it down step by step with your example.

Example List

sql

 Copy code

```
sentinel → [6] → [6] → [6] → [7] → [1] → null
head points to first [6]
current = sentinel
```

Step 1: Check

js

 Copy code

```
if (current.next.val === val) {
    current.next = current.next.next;
}
```

- current = sentinel
- current.next = [6]
- current.next.val = 6 (equal to val)

So we do:

lua

 Copy code

```
current.next = current.next.next
```

Step 2: What does it actually do?

- `current.next` was pointing to the first [6]
- `current.next` is the second [6]

So now we update the link:

```
ini
sentinel.next = second [6]
```

[Copy code](#)

Resulting list:

```
css
sentinel → [6] → [6] → [7] → [1] → null
```

[Copy code](#)

The first [6] is gone (no one points to it anymore).

Step 3: Why doesn't current move?

Correct — in this case, `current` does not move.

Why? Because we only skipped a node.

We want to check again because maybe the next node is also a 6

So after deletion:

- `current = sentinel` (still)
- `current.next = second [6]`

Next loop iteration, we check again:

- Is `current.next.val === 6`? Yes → remove it.

We keep doing this until the first non-6 node ([7]) is found.

Step 4: When does current move forward?

When the value is not equal to `val`, we move `current`:

```
js
else {
    current = current.next;
}
```

[Copy code](#)

So:

- After all 6s are deleted, we'll see [7].
- Since `7 !== 6`, we finally do `current = current.next` and move forward.

Key Idea

- `current.next = current.next.next` → deletes the next node (skips it).
- `current` stays where it is so we can keep checking the new `current.next`.
- Only when `current.next` is safe (not equal to `val`) do we move `current` forward.

💡 Think of `current` as a "gatekeeper":

- He looks ahead (at `current.next`).
- If the node is bad → he cuts it out.
- If the node is good → he steps forward.

Would you like me to draw a small step-by-step picture diagram (like ASCII art) showing how the list changes after each deletion so it sticks in your head?

[Open](#) [Close](#) [Download](#) [Up](#) [Down](#) [Left](#) [Right](#) [...](#)

I'm a bit too confused. Dry run it step by step for a beginner in detail with the example list I gave

Thought for 17s

Nice — let's do a very explicit, beginner-friendly dry run. I'll show each loop iteration, what `current` and `current.next` are, what check runs, what pointer update happens, and the resulting list.

Starting setup

We have this list (I'll name nodes N1..N5 for clarity):

```
ruby -e "print Node[1..5].map{|n| n.val}.join(' → ')"
```

[Copy code](#)

```

sentinel -> N1(6) -> N2(6) -> N3(6) -> N4(7) -> N5(1) -> null
head points to N1 (first 6)
current = sentinel
val = 6

```

Iteration table (step-by-step)

Iter	current points to	current.next points to	Check (current.next.val === 6 ?)	Action taken	Resulting list (sentinel -> ...)	Comm
0 (start)	sentinel	N1(6)	Yes (6 === 6)	current.next = current.next.next sentinel.next = N2	sentinel -> N2(6) -> N3(6) -> N4(7) -> N5(1) -> null	N1 still nobod anym curr
1	sentinel	N2(6) ↓	Yes	sentinel.next = N3	sentinel -> N3(6) -> N4(7) -> N5(1) -> null	Delete sentin
2	sentinel	N3(6) ↑	Yes	sentinel.next = N4	sentinel -> N4(7) -> N5(1) -> null	Delete sentin
3	sentinel	N4(7)	No (7 !== 6)	current = current.next → current = N4	sentinel -> N4(7) -> N5(1) -> null	We mo becau: curren
4	N4(7)	N5(1)	No (1 !== 6)	current = current.next → current = N5	sentinel -> N4(7) -> N5(1) -> null	Move
5	N5(1)	null	Loop condition current && current.next is false (NO current.next)	loop ends	Final list: sentinel -> N4(7) -> N5(1) -> null	Done.

Final result

- sentinel.next is N4, so return sentinel.next gives the new head:

Why current doesn't move when you delete

- When you do current.next = current.next.next, you are removing the node after current.
- current stays so you can immediately check the new current.next.
That's exactly what you want for consecutive deletions (multiple 6s in a row).
- Only when current.next is not a deletion candidate do you advance current .

Memory note (JS-specific)

- Nodes you "skip" become unreachable (no references) and will be cleaned up by garbage collection.
You don't need to manually free them.

If you like, I can show the same trace as a simple ASCII animation (one line per iteration) so you can see the arrows change — say "yes" and I'll paste it.

Lets consider 0th iteration, ie, start. When you do current.next = current.next.next, I get that current.next which was previously pointing to N1 would now point to N2. But sentinel.next would still point to N1 right? I thought sentinel and current 2 different pointers pointing to the same node sentinel and while current.next = current.next.next would make current.next point to N2, sentinel.next would still point to N1 still as both are different

Ahhh 🍏🍊 now I see your exact confusion — and it's a really important beginner pointer concept.
You're absolutely right that current and sentinel are two different variables, but here's the key:

👉 They are not two separate nodes. They both reference the same node object in memory.

So when you write:

```

js  ↵ A-1: https://repl.it/@dmitrykalinichenko/linked-list-deletion#index.js
let sentinel = new ListNode(0, head);
let current = sentinel;

```

- sentinel points to the dummy node object.

- `current` also points to the same exact dummy node object.

Think of it like having two remotes controlling the same TV. If one changes the channel, the other will also see the change, because the TV itself changed.

Step 0 Example in Detail

We start with:

```
kotlin
```

```
sentinel (dummy node object)  
.val = 0
```

Copy code

Both variables:

```
scss
```

```
sentinel → (dummy node)  
current → (same dummy node)
```

Copy code

Now we run:

```
js
```

```
current.next = current.next.next;
```

Copy code

- `current.next` was N1
- `current.next.next` is N2
- So we set `current.next = N2`

Now the dummy node itself has changed:

```
ts
```

```
(dummy node).next = N2
```

Copy code

And since `sentinel` and `current` both point to that *same* dummy node,
`sentinel.next` is now N2 as well.

Key Realization

- `sentinel` and `current` are **two references to the same object**.
 - When you modify the object through one reference, the change is visible through the other.
- So after `current.next = current.next.next`, the link from dummy → N1 is replaced by dummy → N2.
N1 is now unlinked (lost from the list).

Mental Model

Imagine the dummy node is a box:

```
[ Dummy box: val=0 | next -> N1 ]
```

- `sentinel` is a sticky note labeled "sentinel" stuck on the box.
- `current` is another sticky note labeled "current" stuck on the same box.

If you open the box and change `next` from N1 → N2, both sticky notes are still pointing at the **same modified box**.

- So: No, `sentinel.next` does NOT still point to N1.

Because there is only one dummy node, and its `.next` pointer is updated.

Do you want me to draw this with a diagram showing both references before and after the assignment so you can visually see why they both reflect the change?



+ Ask anything

0

...