5 - Singly Linked Lists

Mark Lesson Complete View Code Prev Next

Suggested Problems

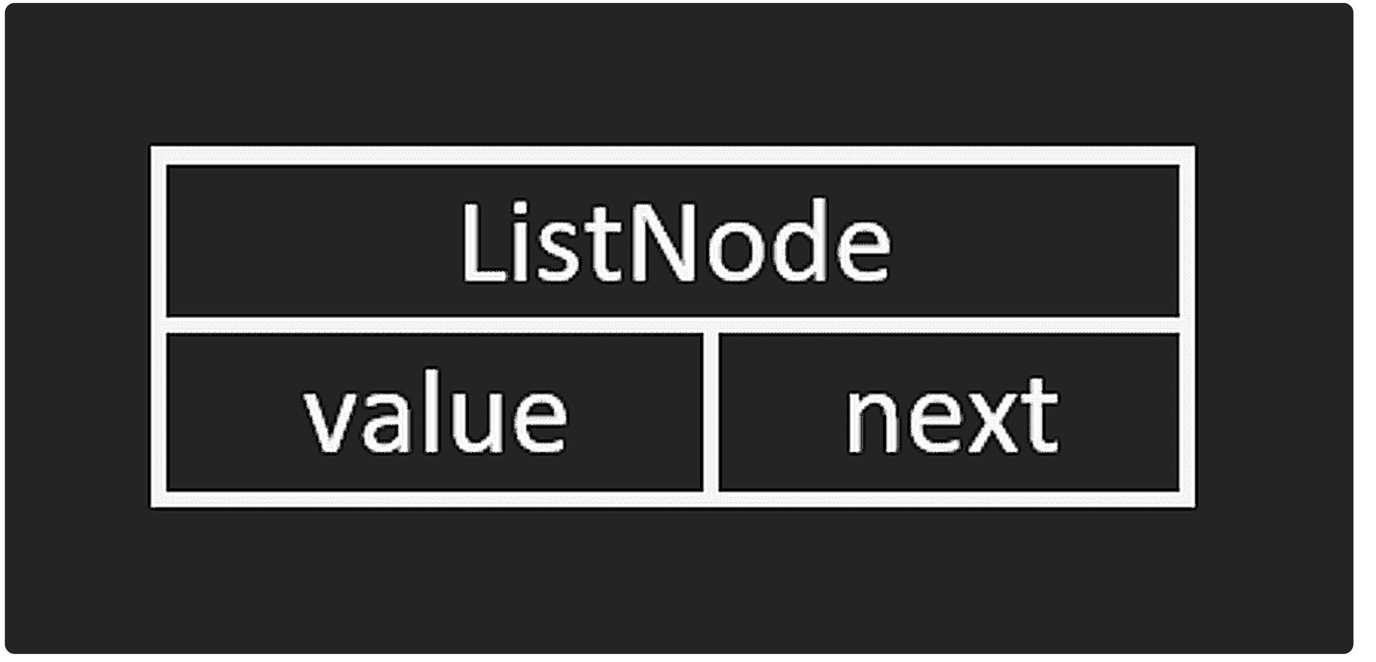
Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Reverse Linked List	Easy		C++
☐	☆	Merge Two Sorted Lists	Easy		C++

Linked Lists

A linked list is another data structure that is like an array in the sense that it stores elements in an ordered sequence, but there are also differences.

The first difference is that linked lists are made up of objects called `ListNode` 's. This object contains two attributes:

1. `value` - This stores the value of the node, the value can be anything - a character, an integer, etc.
2. `next` - This stores the reference to the next node in the linked list. The picture below visualizes the `ListNode` object. This will make more sense a little later on.

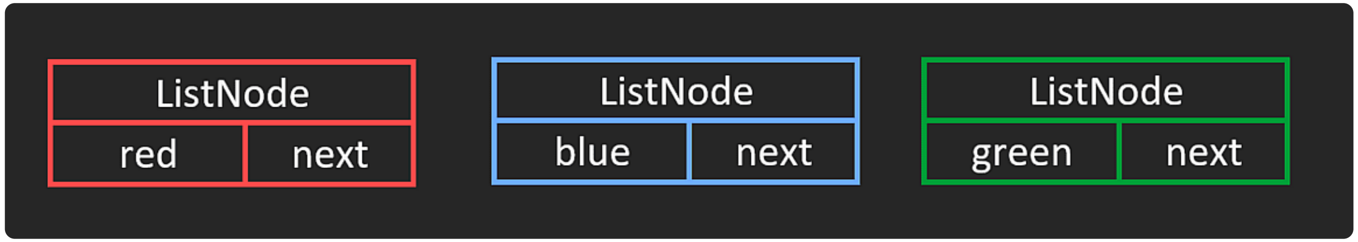


Creating a Linked List from scratch

Chaining these `ListNode` objects together is what results in a linked list. Creating your own `ListNode` class would look like the following in pseudocode.

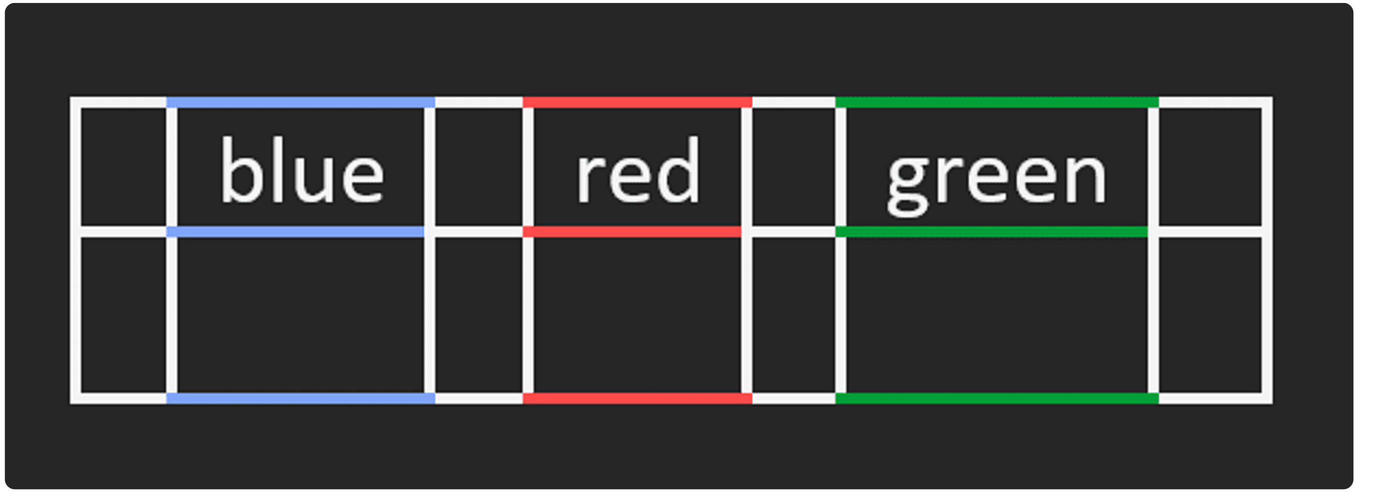
```
class ListNode:
    constructor (value, next):
        1. Set value to the desired value, i.e. integer, char, etc.
        2. Set the next pointer to the desired node, null by default
```

Let's look at an example of how these `ListNode` objects can be chained together to build a desired LinkedList. Suppose that we have three `ListNode` objects – `ListNode1` , `ListNode2` , `ListNode3` , and we instantiate them with the following values as seen in the visual below.



In this case, our value attribute is a string - red, blue, green.

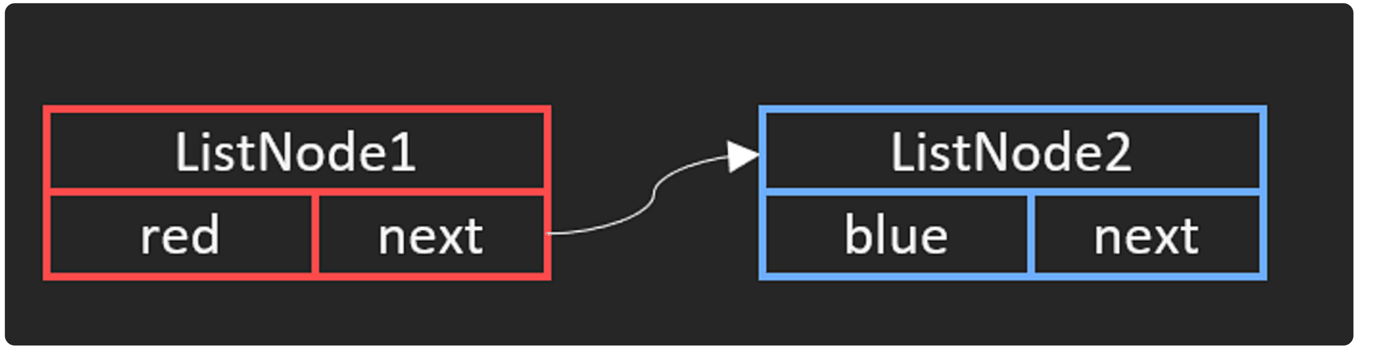
> *At a lower level, upon instantiation, these objects would get stored in an arbitrary order in the memory. We cannot control the order in which the operating system stores these objects in memory, and for our purpose, it is not very relevant but still good to know. The visual below gives a glimpse of how the nodes would be stored in memory.*



Upon instantiation, the nodes would be stored in an arbitrary order in the memory.

Next, we would need to make sure that our next pointers point to another `ListNode` , and not `null` .

```
ListNode1.next = ListNode2
```

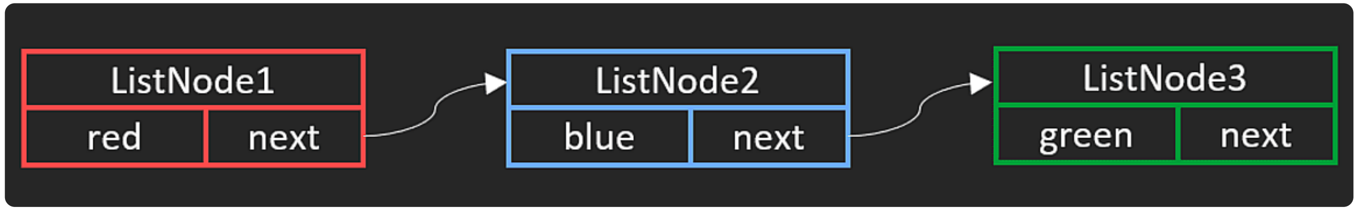


`ListNode1` 's next pointer points to `ListNode2 - ListNode2` comes after `ListNode1` in the Linked List

> *The address for `ListNode2` is retrieved from memory.*

Next, setting the `next` pointer for `ListNode2` and `ListNode3` .

```
ListNode2.next = ListNode3
ListNode3.next = null
```



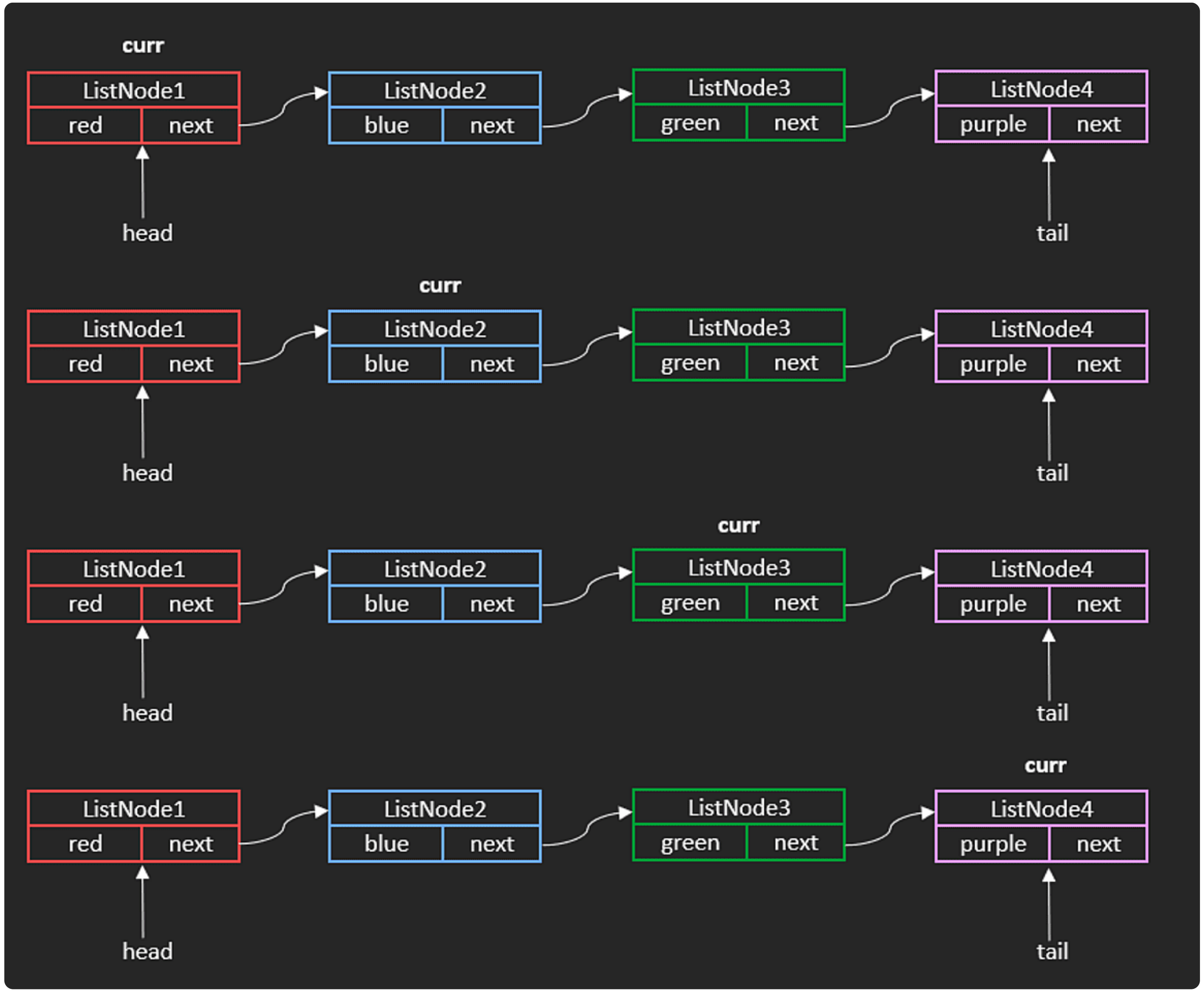
> *Since `ListNode3` is the last node in the LinkedList, its `next` pointer will point to `null`*

Traversal

To traverse a linked list from backward to forward, we can just make use of a simple while loop.

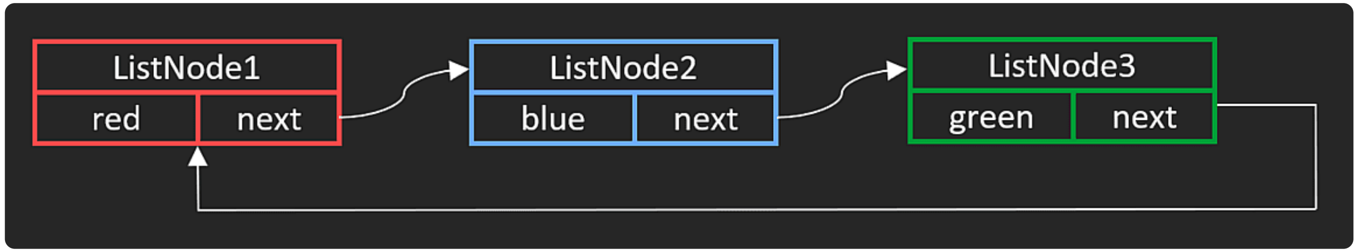
```
cur = ListNode1
while cur is not null:
    cur = cur.next
```

To break down the code, we start the traversal at the beginning, `ListNode1`, and assign it to a variable `cur`, denoting the current node we are at. We keep running the while loop and updating our `cur` to the next node until we encounter a node that is `null` — meaning we are at the end of the linked list and traversal is finished. Traversal is $O(n)$.



Circular Linked List

An interesting scenario presents itself if `ListNode3` 's next pointer points to `ListNode1` instead of `null` . This would create an infinite while loop and the program will crash. This is because we would never reach the end of the linked list. After `ListNode3` , `ListNode3.next` would point to `ListNode1` , which goes to `ListNode2` , which goes `ListNode3` , and back to `ListNode1` , creating a never ending cycle.



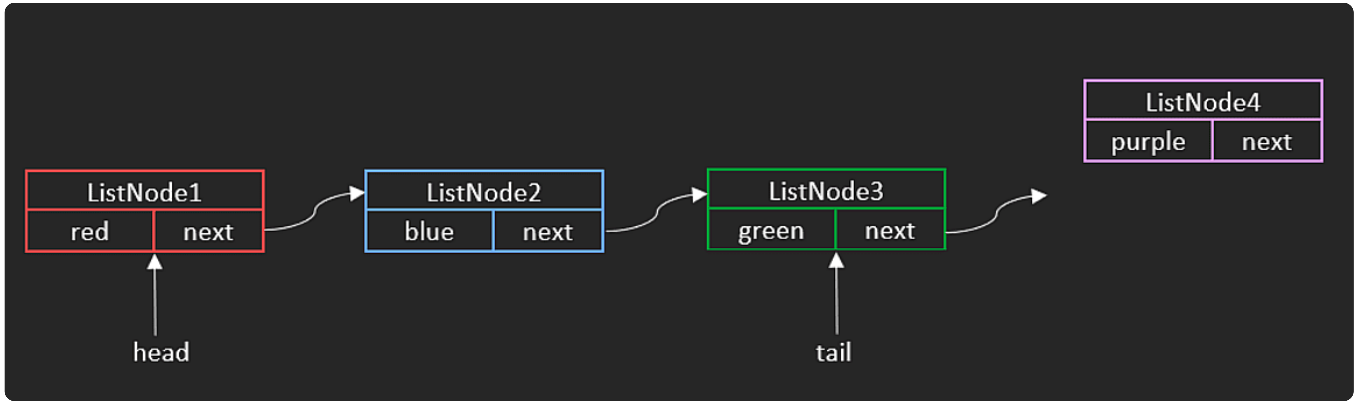
Operations of a Singly Linked List

Linked Lists have a `head` , and a `tail` pointer. The `head` pointer points to the very first node in the linked list, `ListNode1` , and the `tail` pointer points to the very last node — `ListNode3` . If there is only one node in the Linked List, the `head` and the `tail` point to the same node.

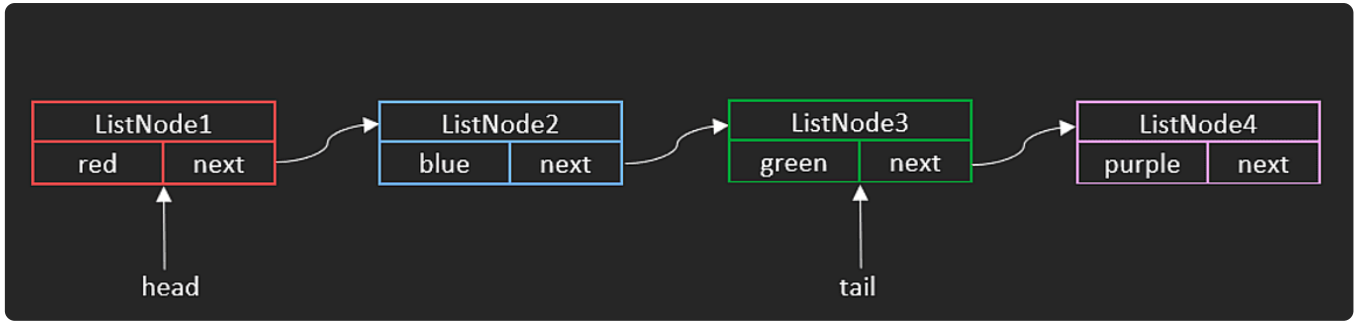
Appending

An advantage that Linked Lists have over arrays is that adding a new element can be performed in $O(1)$ time. No shifting is required after adding another node and we already have the references to `head` and `tail` . Taking our example from above, if we wanted to append a `ListNode4` to the end of the list, we would be appending to the tail. Once `ListNode4` is appended, we update our tail pointer to be at

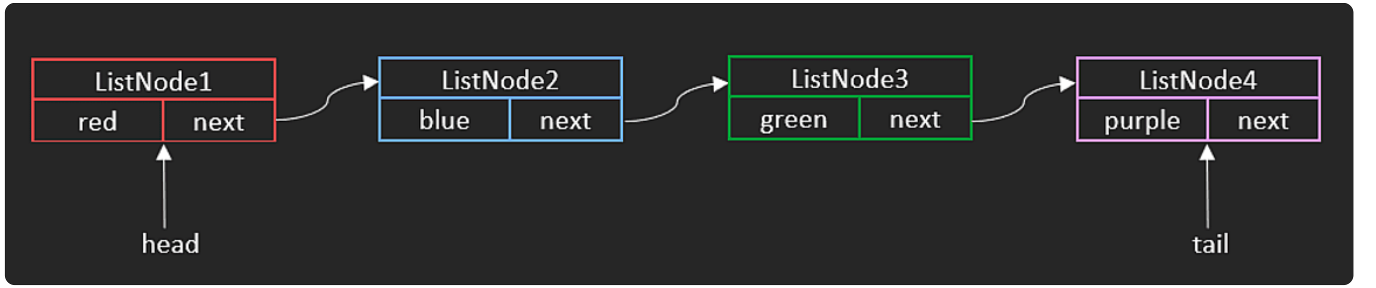
`ListNode4` . This operation would be done in $O(1)$ time since it is only one operation. The steps would look like the following, with code.




```
tail.next = ListNode4
```




```
tail = ListNode4
```

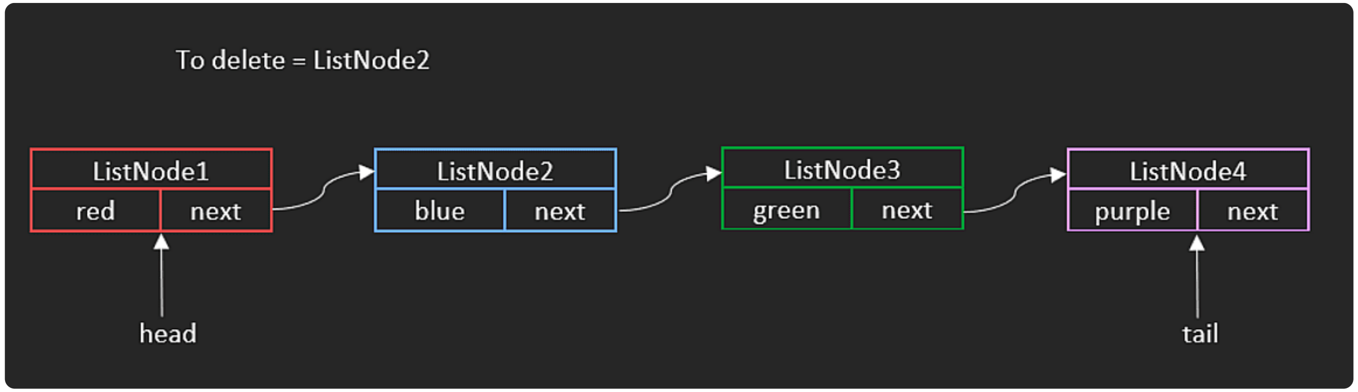


> *While the insertion operation itself is $O(1)$ because no shifting of nodes is required, we still have to traverse to get to an insertion point if we do not have reference to the insertion position.*

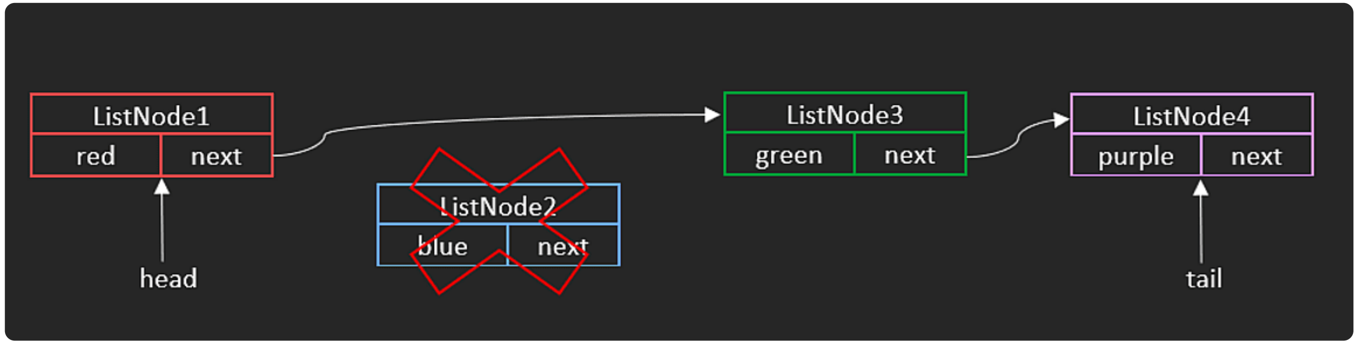
Deleting from a Singly Linked List

Deletion at the head of a singly linked list will take $O(1)$, since it is at the beginning and is a single operation. Again, the traversal itself will take n steps if you do not have the reference to the node . The way to delete a specific node, say `y` , is to skip over it - update `y` 's previous node's `next` pointer to the node that comes after `y` . This is called chaining `next` pointers together.

Visualizing this in code makes more sense. Taking the previous example, suppose we want to delete `ListNode2` . Currently, our `head` points to `ListNode1` , and `head.next` points to `ListNode2` . Since `ListNode2` will cease to exist, we need to update our `head.next` pointer to `ListNode3` , which can be accessed by chaining `next` pointers like `head.next.next` . This makes sense since `head.next` is `ListNode2` , and logically, `head.next.next` would be `ListNode3` .




```
head.next = head.next.next
```



Updated linked list after deletion of `ListNode2` . Notice that now `ListNode1` 's next pointer points to `ListNode3` , instead of `ListNode2`

> *It can be assumed that `ListNode2` memory space will be cleared since most language have garbage collection.*

Closing Notes

Linked Lists definitely outshine arrays in certain scenarios. For example, in the queues chapter, we will see how using Linked Lists to implement the queue data structure can be beneficial compared to arrays.

Operation	Big-O Time Complexity	Note
Access	$O(n)$	
Search	$O(n)$	

Operation	Big-O Time Complexity	Note
Insertion	$O(1)$*	Assuming you have the reference to the node at the desired position
Deletion	$O(1)$*	Assuming you have the reference to the node at the desired position

Contact: neetcodebusiness@gmail.com

Github Privacy Terms