Algorithms and Data Structures for Beginners

7 / 35

About

0 Introduction FREE

Arrays

1 RAM FREE

2 Static Arrays

3 Dynamic Arrays

4 Stacks

Linked Lists

6 - Doubly Linked Lists

□ Mark Lesson Complete

View Code

Prev

Next

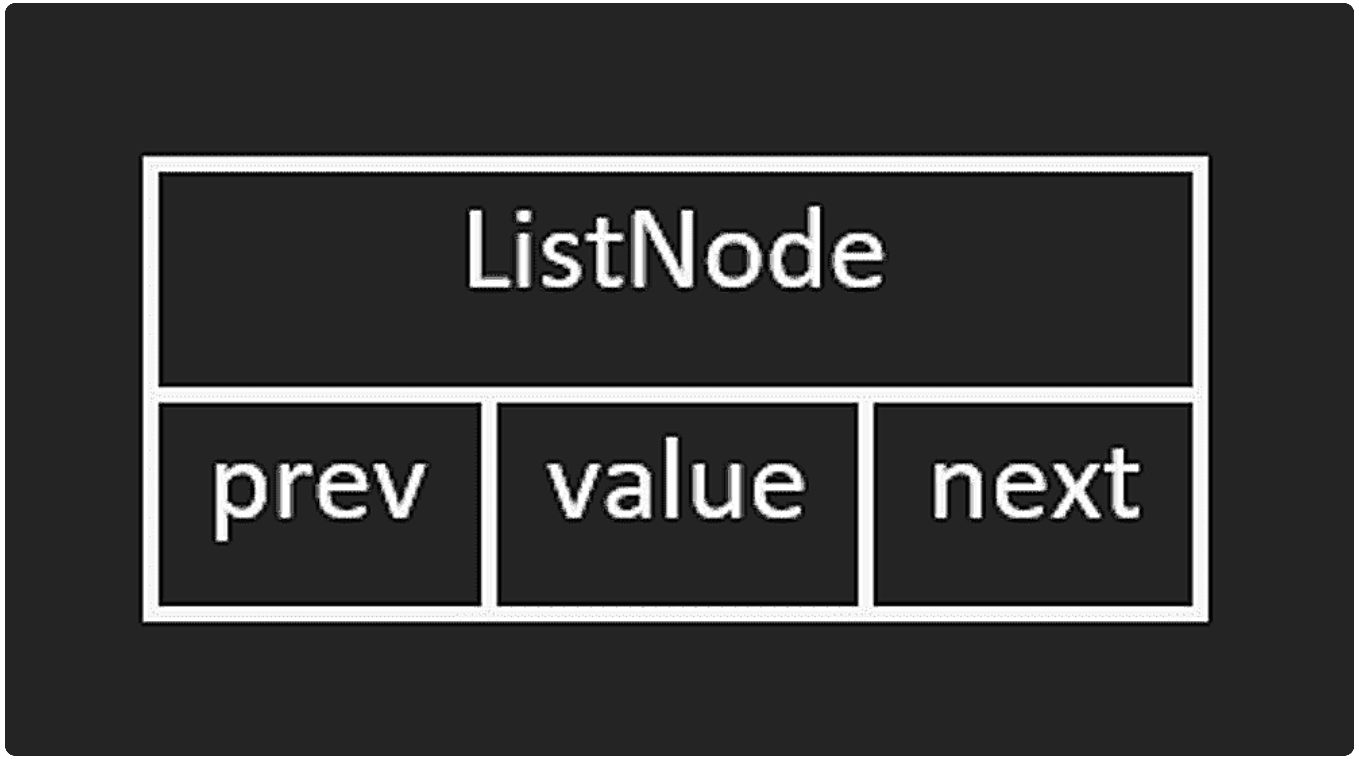
5 Singly Linked Lists FREE

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
☐	☆	Design Linked List	Medium	📹	
☐	☆	Design Browser History	Medium	📹	

Doubly Linked Lists

Having learned about singly linked lists, let’s next learn about its variation - the Doubly Linked List. As the name implies, it's called doubly because each node now has two pointers. We have a `prev` pointer which points to the previous node, in addition to the `next` pointer. If the `prev` pointer points to null, it is an indication that we are at the start of the linked list.



Operations of a Doubly Linked Lists

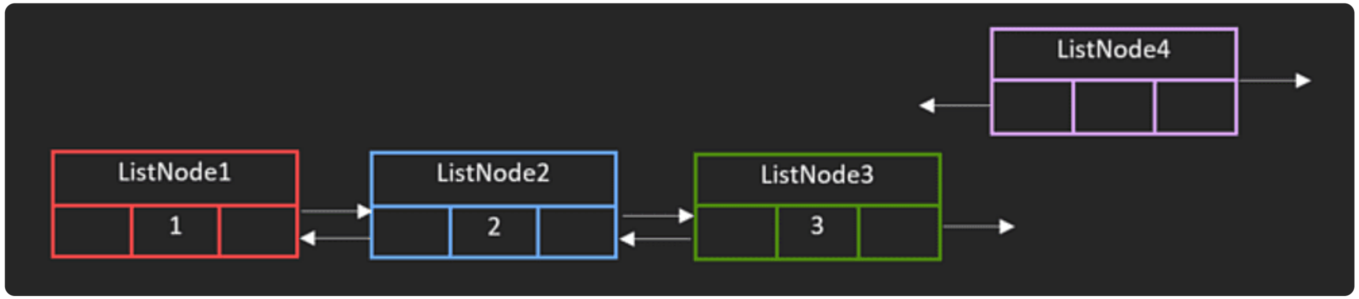
Insertion

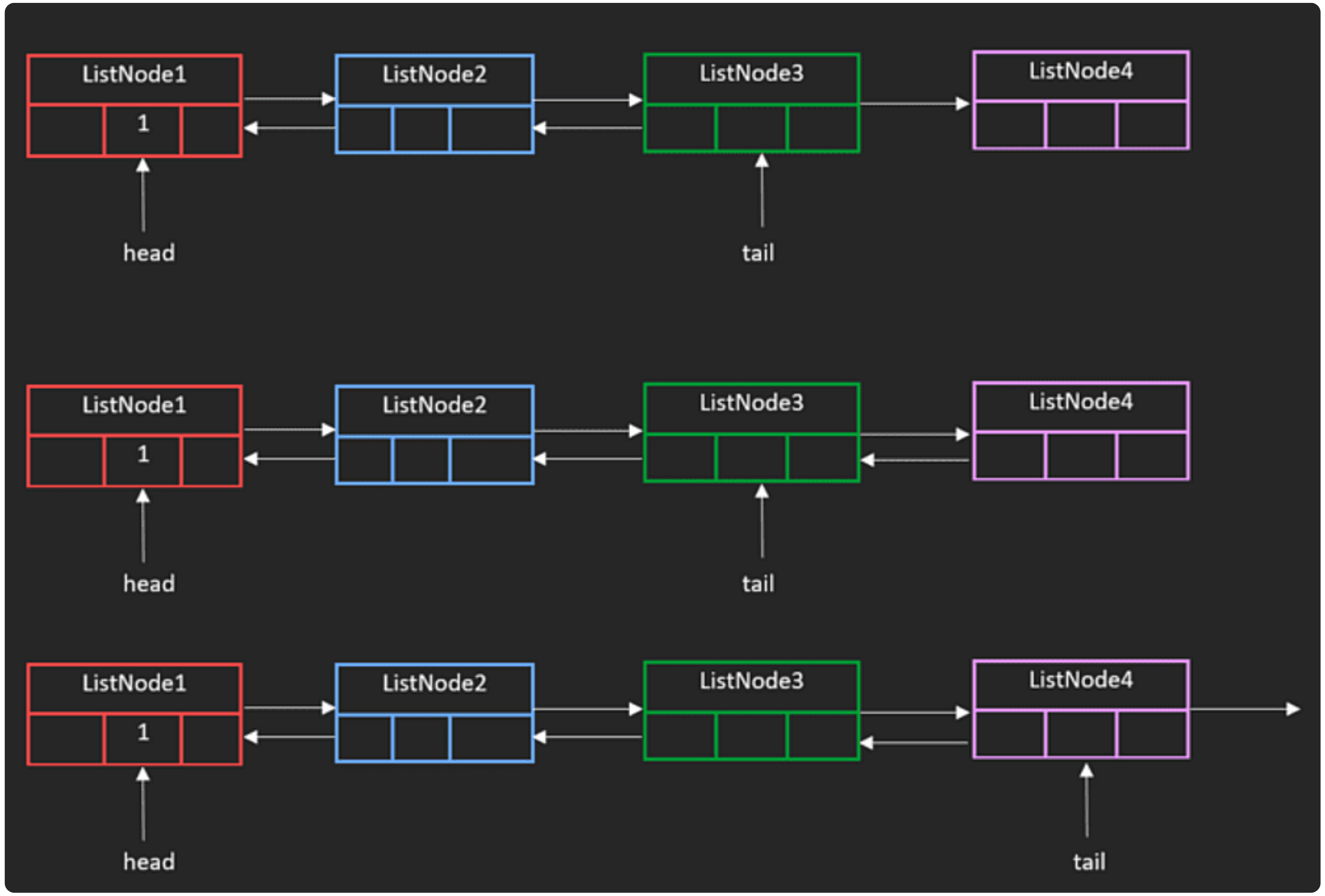
Similar to the singly linked list, adding a node to a doubly linked list will run in $O(1)$ time. Only this time, we have to update the `prev` pointer as well.

For example, looking at the visual below, we have three nodes in our linked list, `ListNode1` , `ListNode2` and `ListNode3` . Now we have another node, `ListNode4` , that we wish to insert. We know the we will have to update the `next` pointer of

`ListNode3` and the `prev` pointer of `ListNode4`. The pseudocode below demonstrates this along with the step by step visual.

```
tail.next = ListNode4
ListNode4.prev = tail
tail = tail.next
```



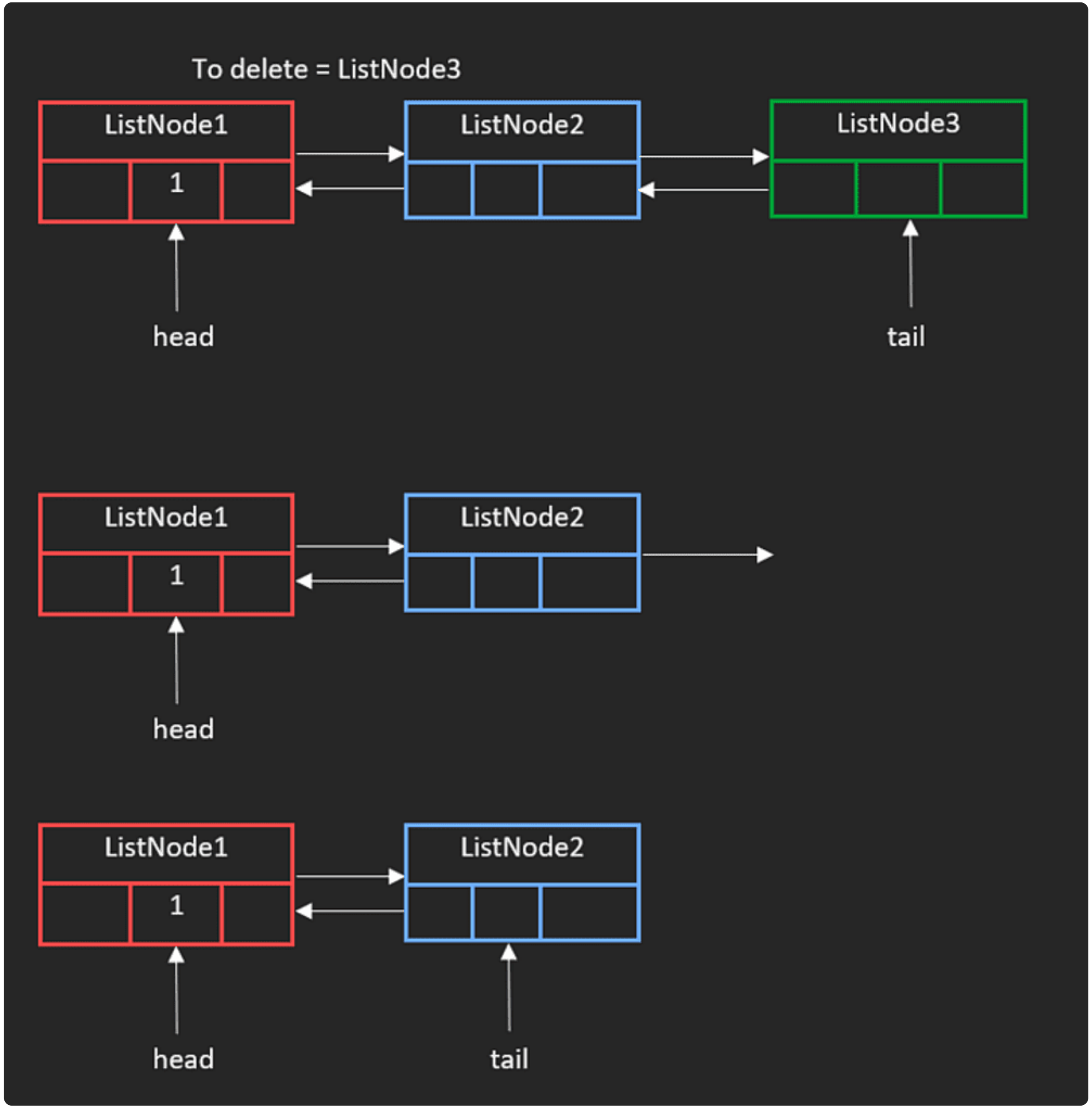


Deletion

Going back to the example with the three nodes, deleting is also a $O(1)$ operation. There is no shifting or traversal required. Instead, in this case adjusting the `prev` pointer is required. The following pseudocode and visual demonstrate this.

```
ListNode2 = tail.prev
ListNode2.next = null
```

```
tail = ListNode2
```


To delete = ListNode3
ListNode1
1
ListNode2
ListNode3
head
tail
ListNode1
1
ListNode2
head
ListNode1
1
ListNode2
head
tail

> *You might have figured out that appending and removing from the end of linked lists are both $O(1)$ operations which is similar to the `push` and `pop` operations of the stack. As mentioned earlier, a stack is just an abstract interface that can also be implemented using linked lists.*

> *If the target node is not the `head` or the `tail`, you must arrive at the node before deletion, which is $O(n)$.*

Access

Similar to singly linked lists, we cannot randomly access a node. So in the worst case, we will have to traverse n nodes before reaching the desired node. This operation runs in $O(n)$.

Closing Notes

This chapter might seem more familiar than expected, but that is because the only major difference between singly and doubly linked lists is that the doubly linked list has a `prev` pointer, which requires more operations when inserting and deleting nodes.

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

> *Compared to arrays, linked lists are less efficient when accessing a random element due to lack of an in-built index. So while arrays will access in $O(1)$ in all cases, linked lists are limited by $O(n)$ unless you are accessing the head node.*

Contact: neetcodebusiness@gmail.com

Github Privacy Terms