

Theory: Doubly linked list

⌚ 4 minutes 0 / 2 problems solved

Skip this topic

Start practicing

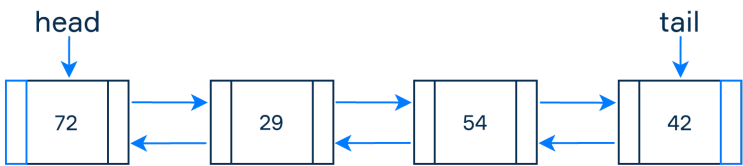
659 users solved this topic. Latest completion was 31 minutes ago.

§1. Essentials

In a doubly linked list, every node stores data, a reference to the next node (*next*) and a reference to the previous node (*prev*).



The following picture demonstrates a list with four nodes that store numbers:



As you can see, the first node doesn't have a previous node, and the last node isn't followed by another one.

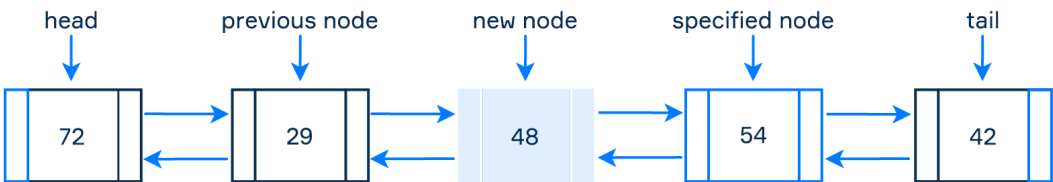
It is possible to iterate over a doubly linked list in both directions starting from head or tail because every node has *prev* and *next* references. So, we don't even need to reverse it to print elements in the reverse order.

§2. Operations

The basic operations of a doubly linked list are the same as for a singly linked list. In this topic, we'll consider only two operations that can be performed more efficiently in the case of a doubly linked list.

1) Sometimes we need to add a new element before a *specified node* to which we have a reference. It can be done quite easily: we simply take the *previous node* of the *specified node* and set the *next* reference of the *previous node* to the new one. We also need to set both references of the new node and modify the *prev* reference of the *specified node* to refer to the new node. This operation has $O(1)$ time complexity.

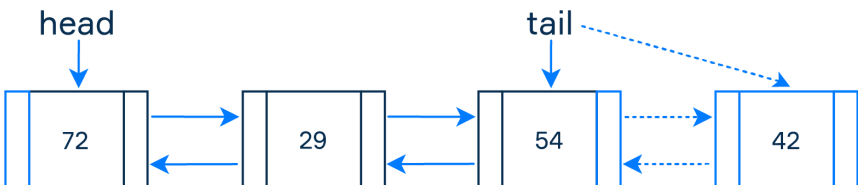
The following picture demonstrates adding element 48 as a new node before the read node (with 54).



If the list was singly linked, we would be forced to look for the previous node of the specified one, and that takes $O(n)$ time.

2) The operation of removing the last element is also more efficient when using doubly linked lists. All we need is to modify the next reference of the second-to-last node to nothing (null or nil, depending on the language) and reset the *tail* reference. We can access the second-to-last node through the *prev* reference of the *tail* node the reference to which we have. This operation has $O(1)$ time complexity.

In the following example, we remove the last node (with 42) from the list and reassign tail to the new last node (54):



Current topic:

[Doubly linked list](#) ...

Topic depends on:

✗ [Singly linked list](#) ...

Topic is required for:

[Doubly linked list in Java](#) ...

Table of contents:

[1 Doubly linked list](#)

[§1. Essentials](#)

[§2. Operations](#)

[§3. Conclusion](#)


[Feedback & Comments](#)

In the case of a singly linked list, we don't have the *prev* reference and must look for the second-to-last node traversing the list, which takes $O(n)$ time.

§3. Conclusion

A doubly linked list is an unbound linear data structure like a singly linked list, but it provides more flexibility and optimizes some basic operations because it uses an additional reference between the nodes. This is the main reason why this kind of lists is common in standard libraries of many programming languages. But there is one drawback: storing additional references requires more space.

 Report a typo

47 users liked this theory.  didn't like it. What about you?



Start practicing

[Comments \(0\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)