






# Linked Lists vs. Lists

Let's pit the two data structures against each other to find out which is more efficient.

The main difference between lists and linked lists is in the way elements are inserted and deleted. As for linked lists, insertion and deletion at the **head** happen in a constant amount of time, whereas at **tail**, it takes  $O(n)$  time. In the case of lists, it takes  $O(n)$  time to insert or delete a value. This is because of the different memory layouts of both the data structures. Lists are arranged contiguously in the memory, while nodes of a linked list may be dispersed in the memory. This memory layout also affects access operation; contiguous layout of lists allows us to index the list; thus, access operation in the list is  $O(1)$ , whereas, for a linked list, we need to perform a traversal thus, it becomes  $O(n)$ . The following table sums up the performance tradeoff between list and linked list:

| Operation        | Linked List | List               |
|------------------|-------------|--------------------|
| access           | $O(n)$      | $O(1)$             |
| insert (at head) | $O(1)$      | $O(n)$             |
| delete (at head) | $O(1)$      | $O(n)$             |
| insert (at tail) | $O(n)$      | $O(n) / O(1)^*$    |
| delete (at tail) | $O(n)$      | $O(n) / O(1)^{**}$ |




In a linked list, insertion and deletion happen in a constant amount of time    given the pointer of the node to be deleted/inserted. You can simply add or delete a node between two Nodes. However, if we were to delete the tail, we would still have to traverse the linked list and update the tail pointer and set it to the previous node. This will take  $O(n)$  time. In a list, the two operations would take  $O(n)$  time since addition or deletion would mean shifting the whole list left or right. The access operation in lists, as we talked about earlier, is much faster as you can simply use the index of a list to access the value you need. In linked lists, there is no concept of indexing. To reach a certain element in the list, we must traverse it from the beginning.

**Note:**\* Insertion at tail for **arrays** like data structures is in  $O(n)$ , but in python, the **append** method for lists is able to do it in  $O(1)$ .

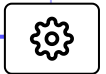
**Note:\*\*** Deletion at tail for **arrays** like data structures is in  $O(n)$ , but in python, the **pop** method for lists is able to do it in  $O(1)$ .

As you can see, there is a trade-off between the features provided by both structures. You will understand more about the working of linked lists in the lessons that follow.

Next, we'll look at all the functions that should be present in a standard **LinkedList** class.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) 



[← Back](#)

Singly Linked Lists (SLL)

Basic Linked List Operations

 Completed[Report an Issue](#)