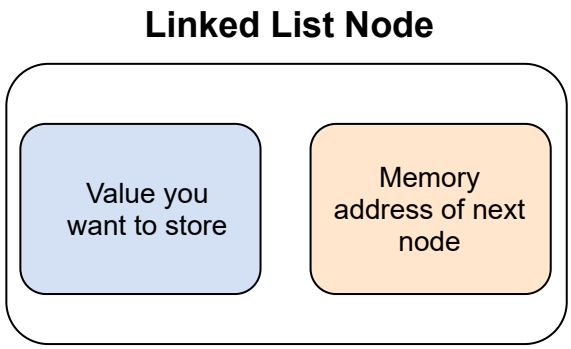
Linked List

This lesson discusses the complexity of operations on a linked list.

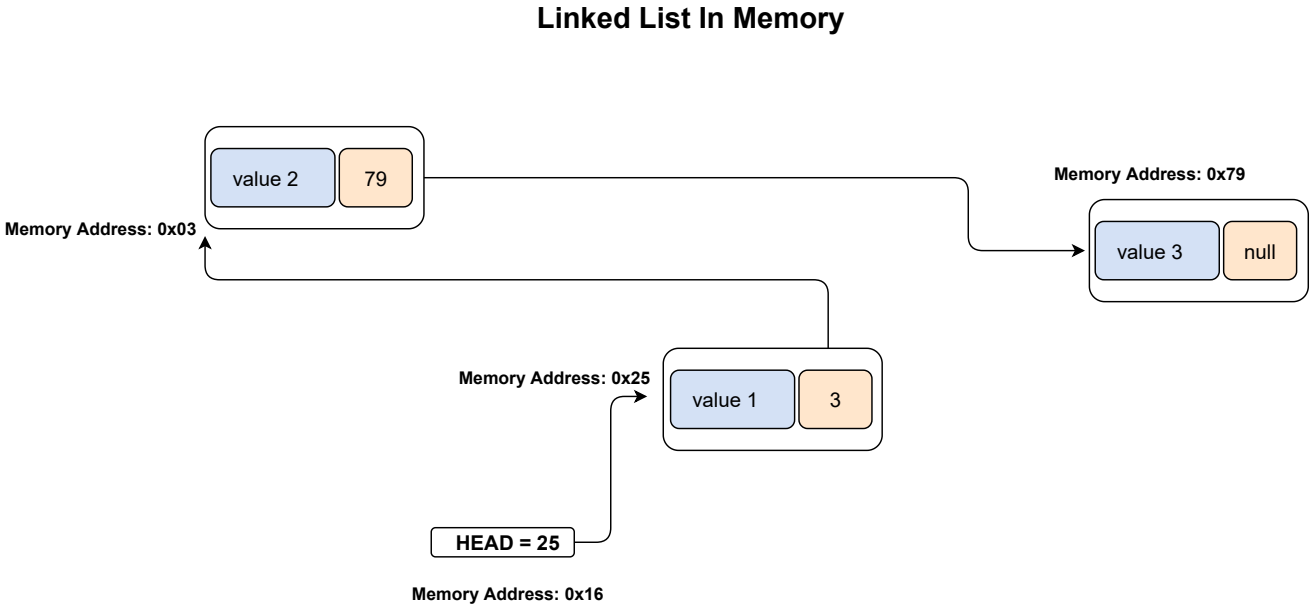
Linked list is the natural counterpart to the array data-structure. In linked list's case we trade more time for less space. A linked list is made up of nodes, and each node has the following information:

- value
- pointer to next node



You can think of a linked list as a chain. If you have a single value, you'll have a linked list of one node. If you require 100 values to store, your linked list will be 100 nodes long. Contrast it to an array, and whether you need to store one value or a 100, you'll need to declare an array of size 100 or the maximum values you intend to eventually store.

Linked list is able to save on memory and be slow in retrieval because it ***doesn't require contiguous allocation of memory***. Nodes of a linked list could be all over the place in a computer's memory. This placement requires us to always remember the first or head node of the linked list. Below is one possible representation of a linked list consisting of three nodes in memory. Note that `head` is a variable that contains the *address* of the first node in the linked list. If we lose the `head`, we lose the linked list. Unlike an array, you can see that the nodes of the linked list can be anywhere.



Given the above representation, it becomes very easy to reason about the various operations in a linked list.

- If we want to retrieve the ***nth*** element in the list, we'll need to walk the list from the head all the way to the ***nth*** element. In the worst case, we could be asked to retrieve the last element of the list. Therefore, in the worst case, the retrieval would need us to walk to the back of the linked list for a total of **n** steps, which would make the complexity of retrieval for a linked list O(n).
- Note that retrieving the first element or the head is still a O(1) or constant time operation.
- Insertion is an interesting operation. If you add at the start of the linked list, it is a constant time operation. If you desire to add an element at the ***nth*** position, where n could be the very end of the linked list, then the insert operation takes O(n) time.

Another way to think about these operations is to ask yourself this question: if the size of the linked list increases, does it affect how many steps are required to perform a given operation? If we insert at the head of the linked list, no matter how long our linked list is, the operation will take a constant number of steps. However, adding to the very end of the linked list requires us to first reach the last element, and only then are we be able to append a new element.

Therefore, the size of the linked list starts to matter and affects complexity.

Improving Runtimes

At times, runtimes of operations can be improved on data structures. For instance, if your application logic requires adding new elements at the tail of a linked list, then one can maintain a `tail` pointer (similar to the `head` pointer) that will always point to the end of the linked list. This makes insertion at the end of a linked list O(1) operation instead of O(n).

Space Complexity

The astute reader would notice that we are also storing an additional variable with each node of the linked list - a variable that points to the next node in the chain. So if we have a linked list of 1000 nodes, then we have one thousand of these `next` pointer variables taking up memory too! The space complexity would be:

> $$n + n * (pointer\ variable\ size)$$

Since the pointer variable size would be a constant number of bytes in any language, we'll be left with,

> $$n + n * (constant)$$
>
> $$n + n * c$$
>
> $$(c + 1)n$$
>
> $$O(n)$$

Pop Quiz

Q

What would the time complexity to find the predecessor of a node in a linked list?

Check Answers