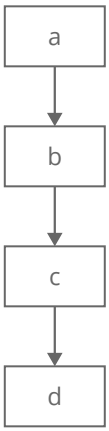
Interview Cake



Linked List

Data Structure (/data-structures-reference)

Quick reference

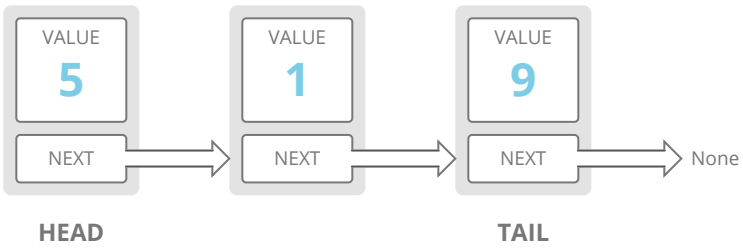
A **linked list** organizes items sequentially, with each item storing a pointer to the next one.

Picture a linked list like a chain of paperclips linked together. It's quick to add another paperclip to the top or bottom. It's even quick to insert one in the middle—just disconnect the chain at the middle link, add the new paperclip, then reconnect the other half.

An item in a linked list is called a **node**. The first node is called the **head**. The last node is called the **tail**.

> Confusingly, *sometimes* people use the word **tail** to refer to "the whole rest of the list after the head."

	Worst Case
space	$O(n)$
prepend	$O(1)$
append	$O(1)$
lookup	$O(n)$
insert	$O(n)$
delete	$O(n)$



> Unlike an array, consecutive items in a linked list are not necessarily next to each other in memory.

Strengths:

- **Fast operations on the ends**. Adding elements at either end of a linked list is $O(1)$. Removing the first element is also $O(1)$.
- **Flexible size**. There's no need to specify how many elements you're going to store ahead of time. You can keep adding elements as long as there's enough space on the machine.

Weaknesses:

- **Costly lookups**. To access or edit an item in a linked list, you have to take $O(i)$ time to walk from the head of the list to the ith item.

Uses:

- **Stacks (/concept/stack) and queues (/concept/queue)** only need fast operations on the ends, so linked lists are ideal.

In C++

C++ provides a built in linked list implementation. Here's how we'd use it to construct the linked list above:

C++

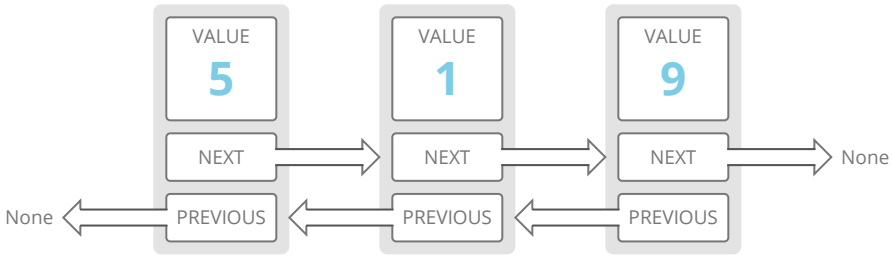
```
list<int> ll;


ll.push_back(5);
ll.push_back(1);
ll.push_back(9);
```

Doubly Linked Lists

In a basic linked list, each item stores a single pointer to the next element.

In a **doubly linked list**, items have pointers to the next *and the previous* nodes.



Doubly linked lists allow us to traverse our list *backwards*. In a *singly* linked list, if you just had a pointer to a node in the *middle* of a list, there would be *no way* to know what nodes came before it. Not a problem in a doubly linked list.

Not cache-friendly

Most computers have caching systems that make reading from sequential addresses in memory faster than reading from scattered addresses (/article/data-structures-coding-interview#ram).

Array (/concept/array) items are always located right next to each other in computer memory, but linked list nodes can be scattered all over.

So iterating through a linked list is usually quite a bit slower than iterating through the items in an array, even though they're both theoretically $O(n)$ time.

See also:

- Array (/concept/cpp/array)
- Queue (/concept/cpp/queue)
- Stack (/concept/cpp/stack)

Linked List Coding Interview Questions

Delete Node »

Write a function to delete a node from a linked list. Turns out you can do it in constant time! keep reading »

(/question/cpp/delete-node)

Does This Linked List Have A Cycle? »

Check to see if a linked list has a cycle. We'll start with a simple solution and move on to some pretty tricky ones. keep reading »

(/question/cpp/linked-list-cycles)

Reverse A Linked List »

Write a function to reverse a linked list in place. keep reading »

(/question/cpp/reverse-linked-list)

Kth to Last Node in a Singly-Linked List »

Find the kth to last node in a singly-linked list. We'll start with a simple solution and move on to some clever tricks. keep reading »

(/question/cpp/kth-to-last-node-in-singly-linked-list)

Find Repeat, Space Edition BEAST MODE »

Figure out which number is repeated. But here's the catch: do it in linear time and constant space! keep reading »

(/question/cpp/find-duplicate-optimize-for-space-beast-mode)

All Questions ➜ (/all-questions)
