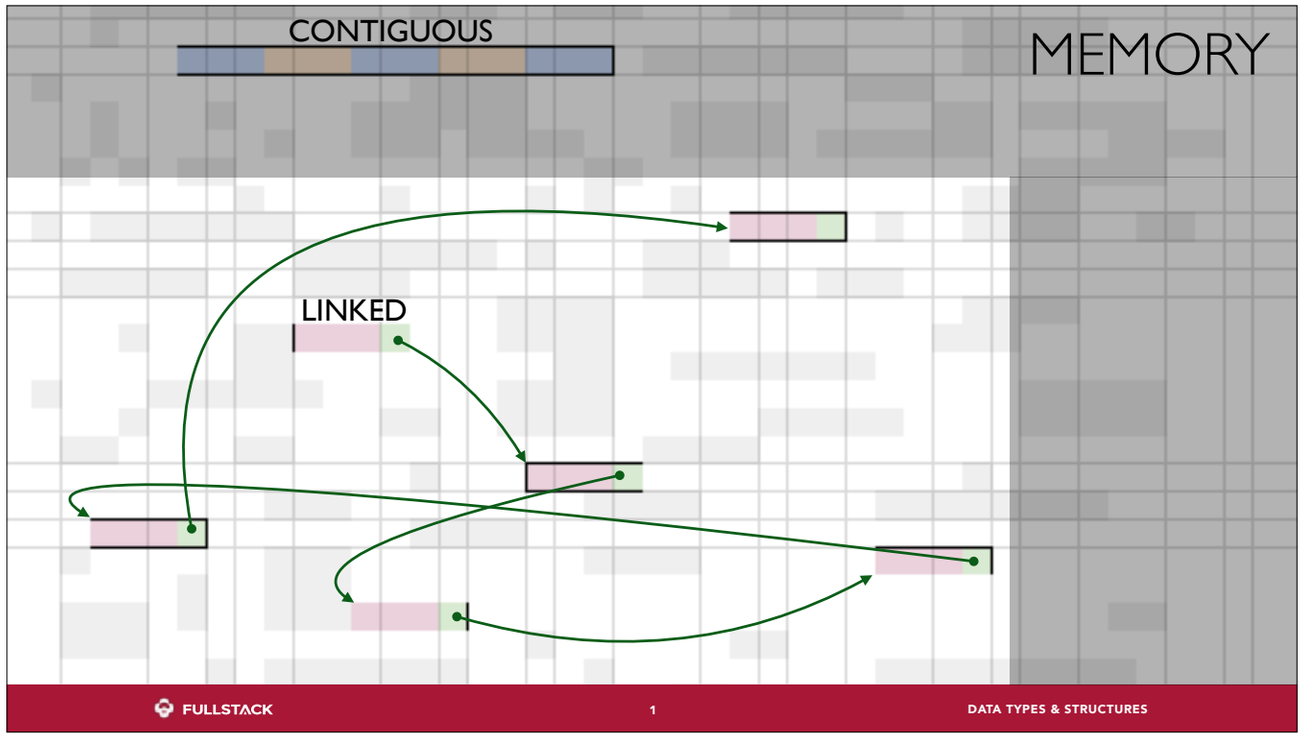


Data structures can typically be stored either in one solid block (contiguous) or in many small blocks, each with a memory address for the next block (linked). What are some pros and cons of each approach? Contiguous pros: good physical locality of memory means faster in practice. No wasted space for memory addresses. Linked pros: infinite adding so long as there are open spaces big enough for element + address.

Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12

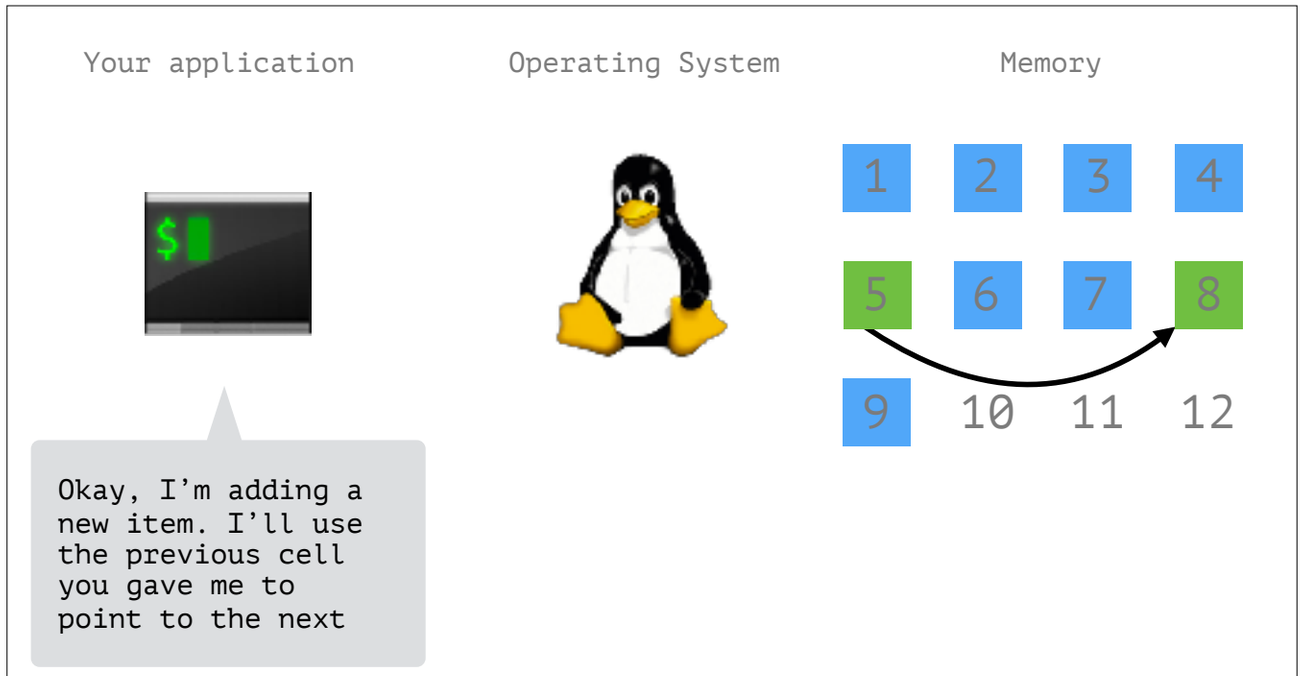
Your application

Operating System

Memory

1	2	3	4
5	6	7	8
9	10	11	12

I just need one block of memory for now...doesn't matter where!


Your application
Operating System
Memory
1
2
3
4
5
6
7
8
9
10
11
12
Okay, I'm adding a new item. I'll use the previous cell you gave me to point to the next

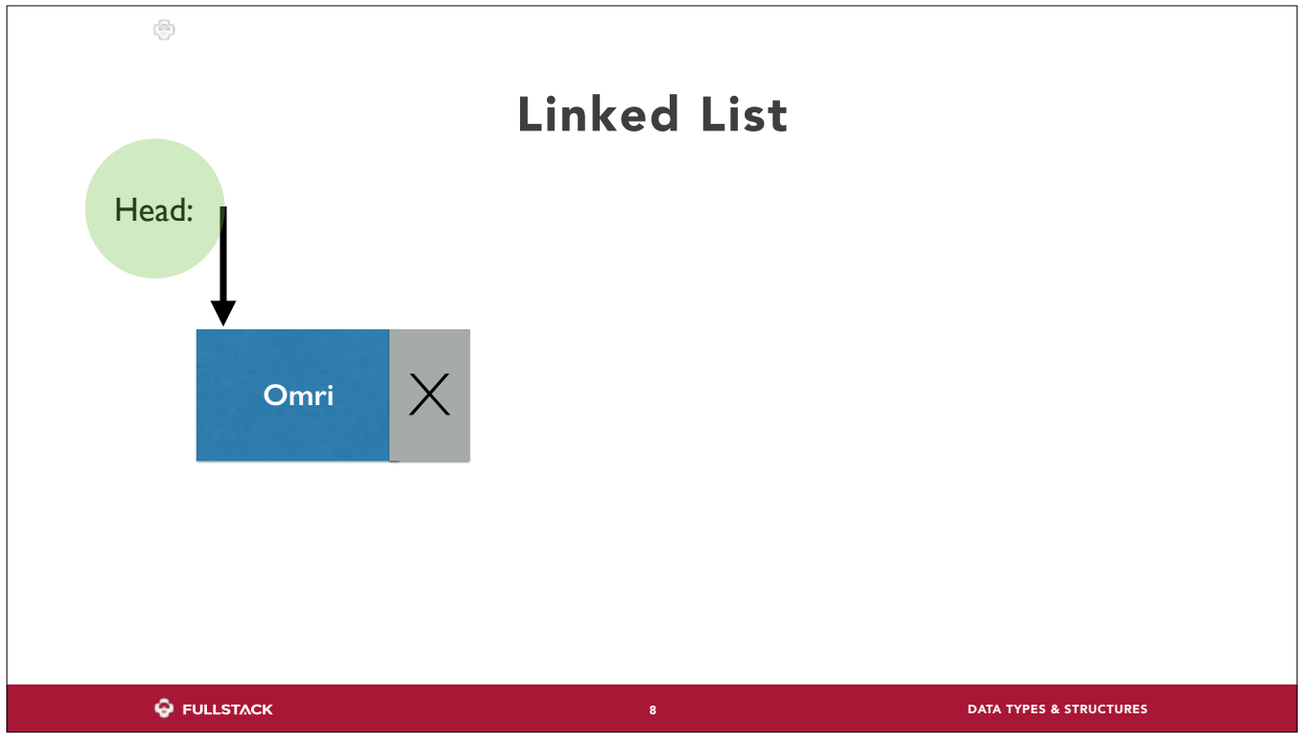

LINKED LISTS

The Linked List DS

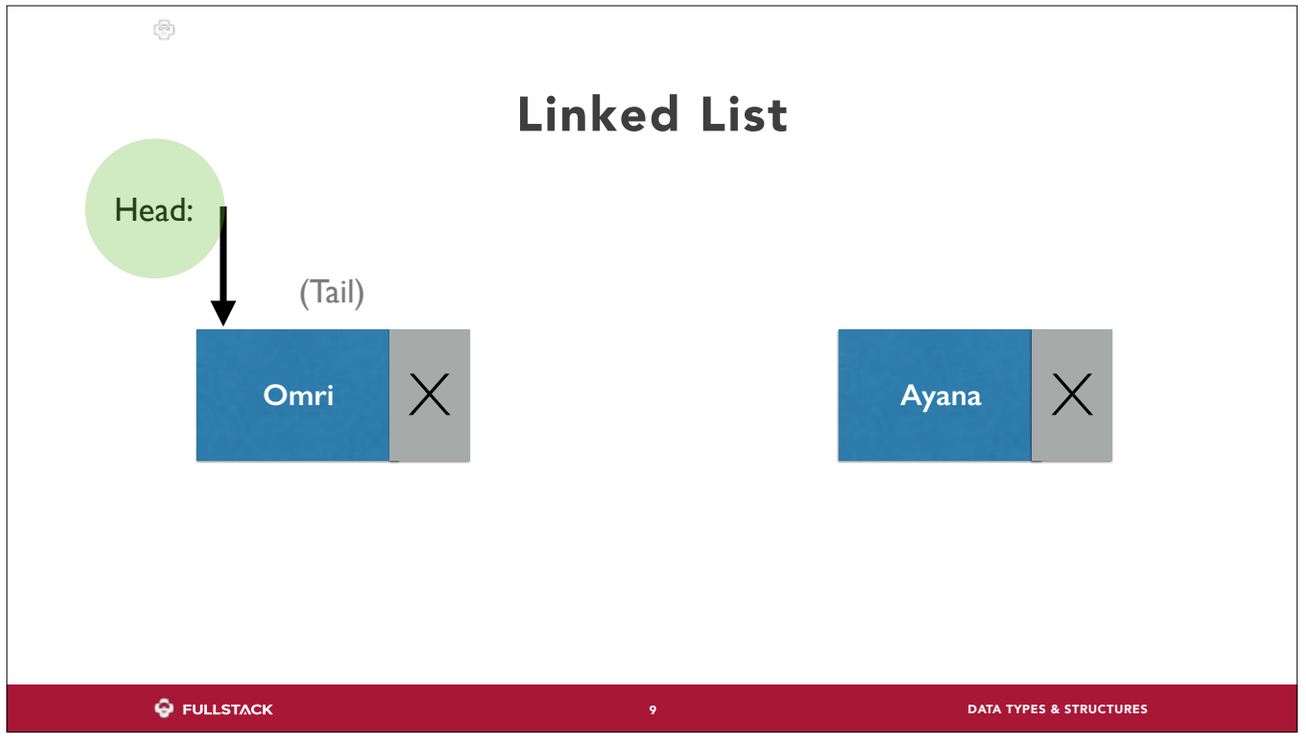
- Data structure used for *list, stack, queue, deque* ADTs etc.
- Uses *nodes* which encapsulate a value and pointer(s)
- Main entity holds reference(s) to just a head and/or tail node
  - the "handle(s)"
- Each node then *points* to the next and/or previous node
  - "singly-linked" (unidirectional) vs. "doubly-linked" (bidirectional)

Linked List

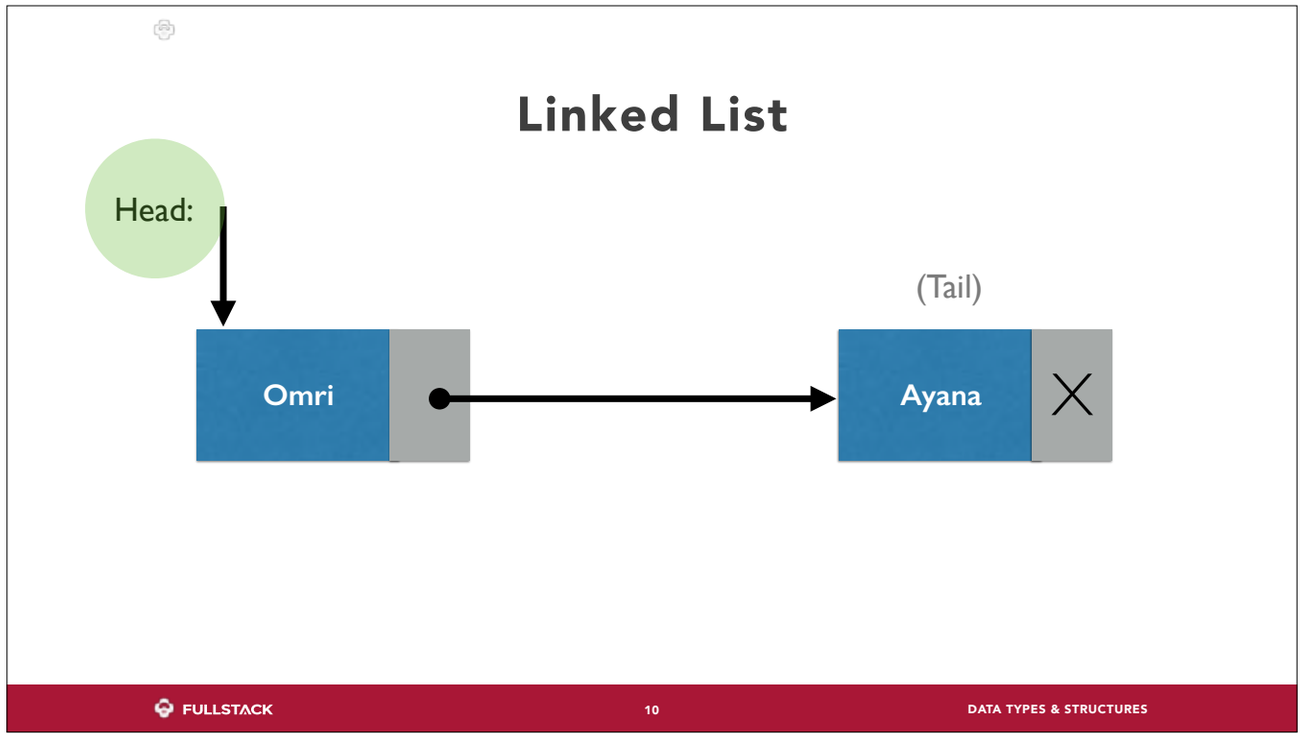
When it comes to implementations, the LinkedList is often considered synonymous with its handle or handles (in this case just an empty "head" reference).



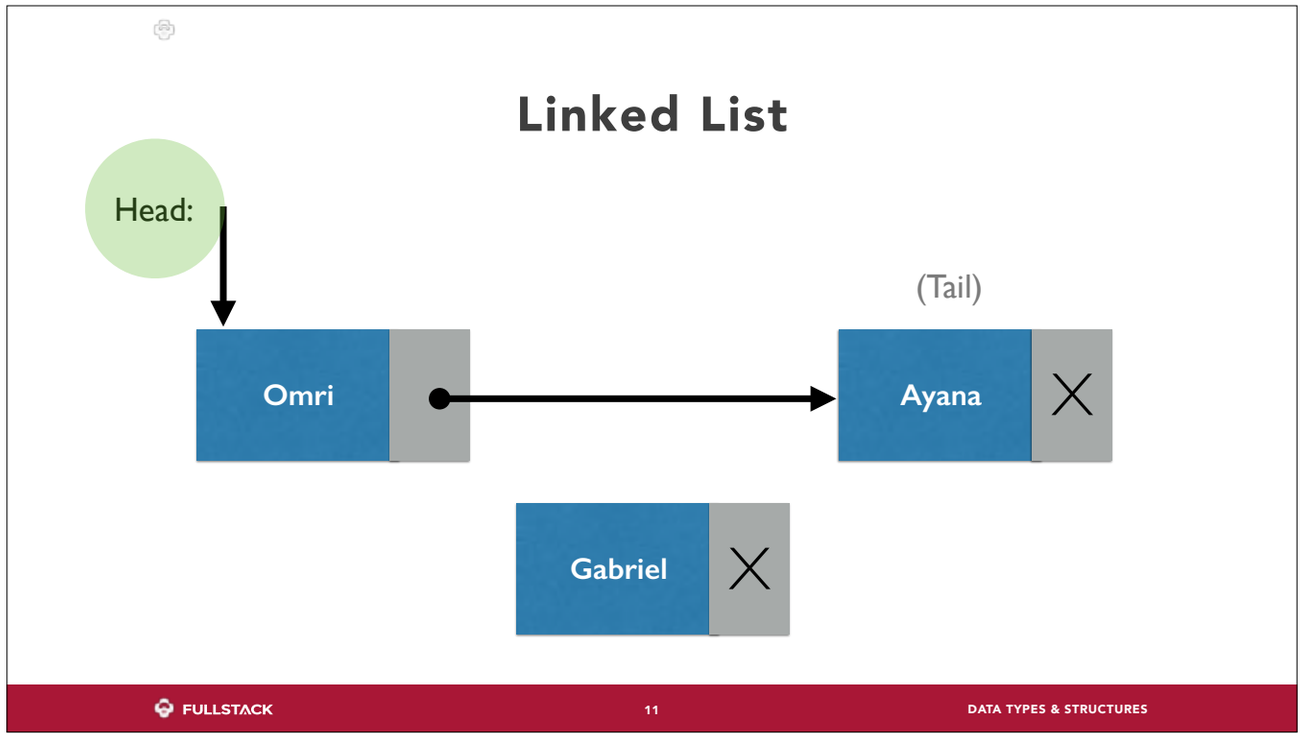
This is a Linked List containing one node. The node has a value (Omri) and an empty `next` pointer. In this case it only has one pointer (no `previous`) so this is going to be a singly-linked list.



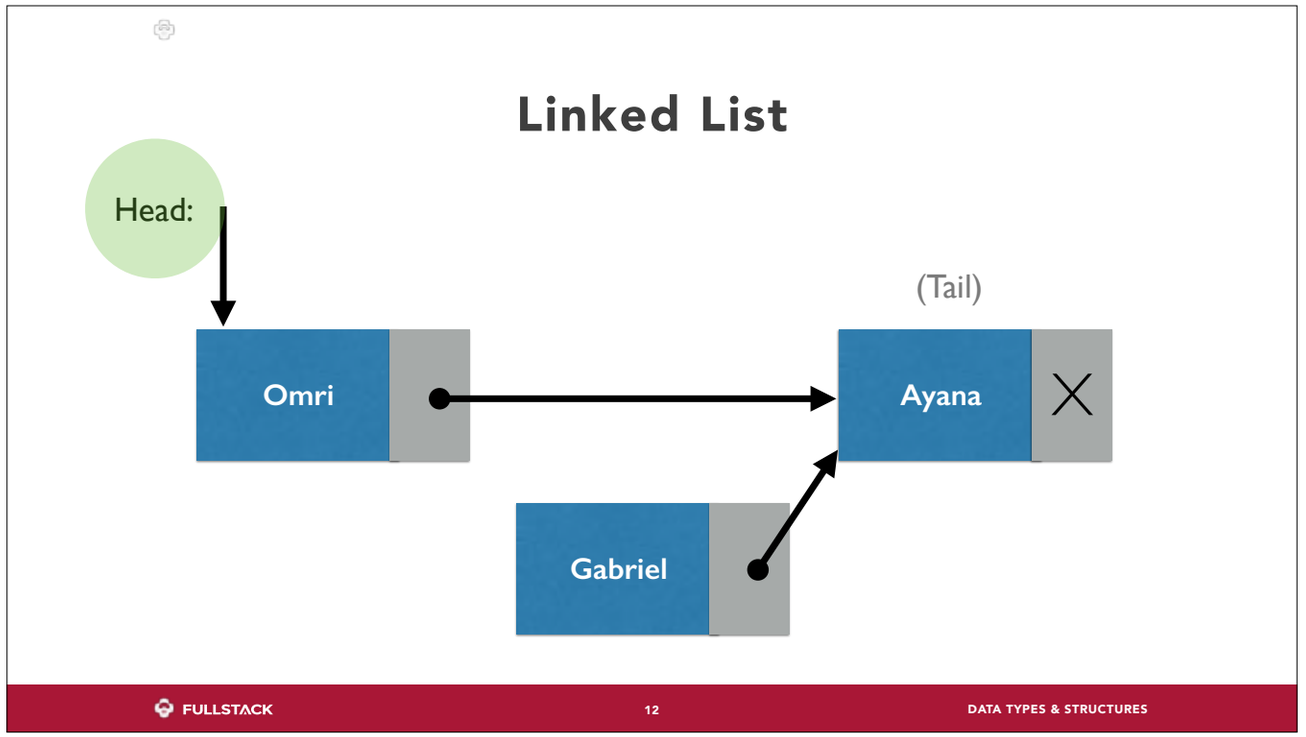
Partway through adding to the tail — we have created a new Ayana node, but it isn't connected to our list yet.



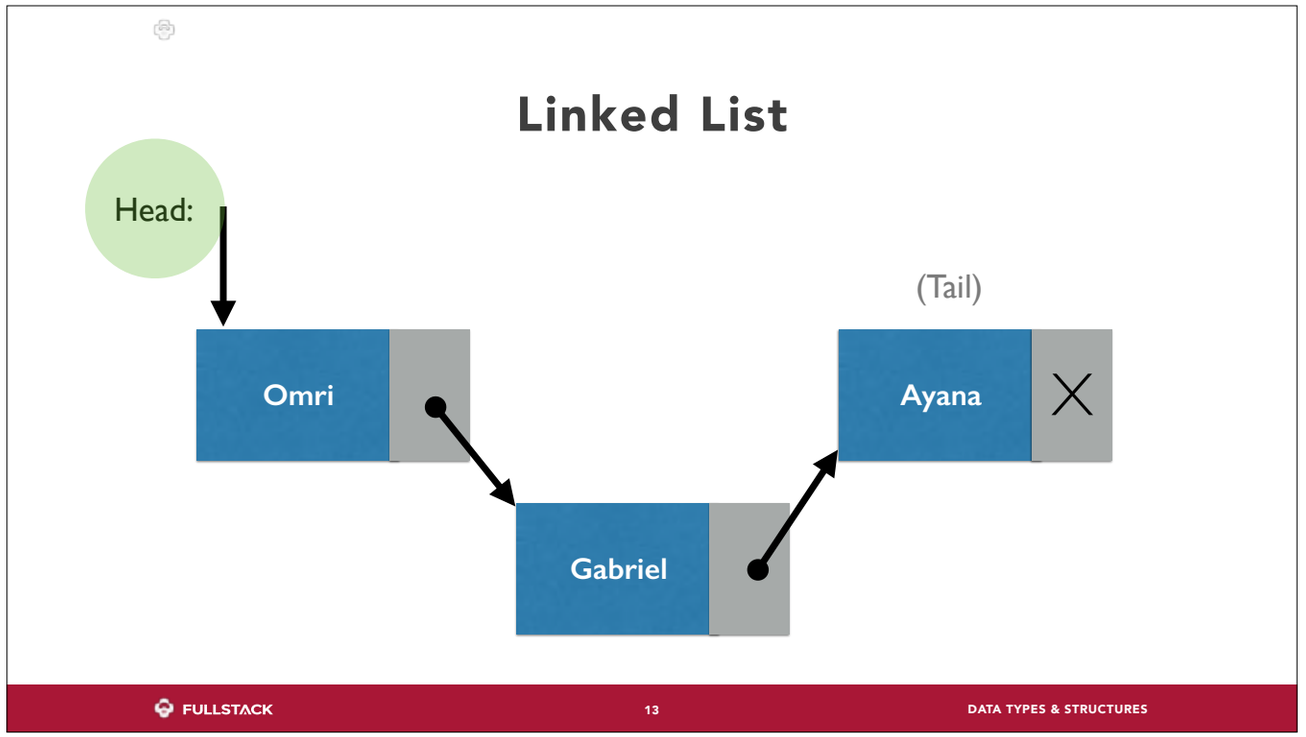
Set the Omri node's `next` pointer to the Ayana node, and we would say that Ayana is the new tail node. Can I get to Omri from Ayana? If I want to reach Ayana, I have to start at `head` and then follow `next`.



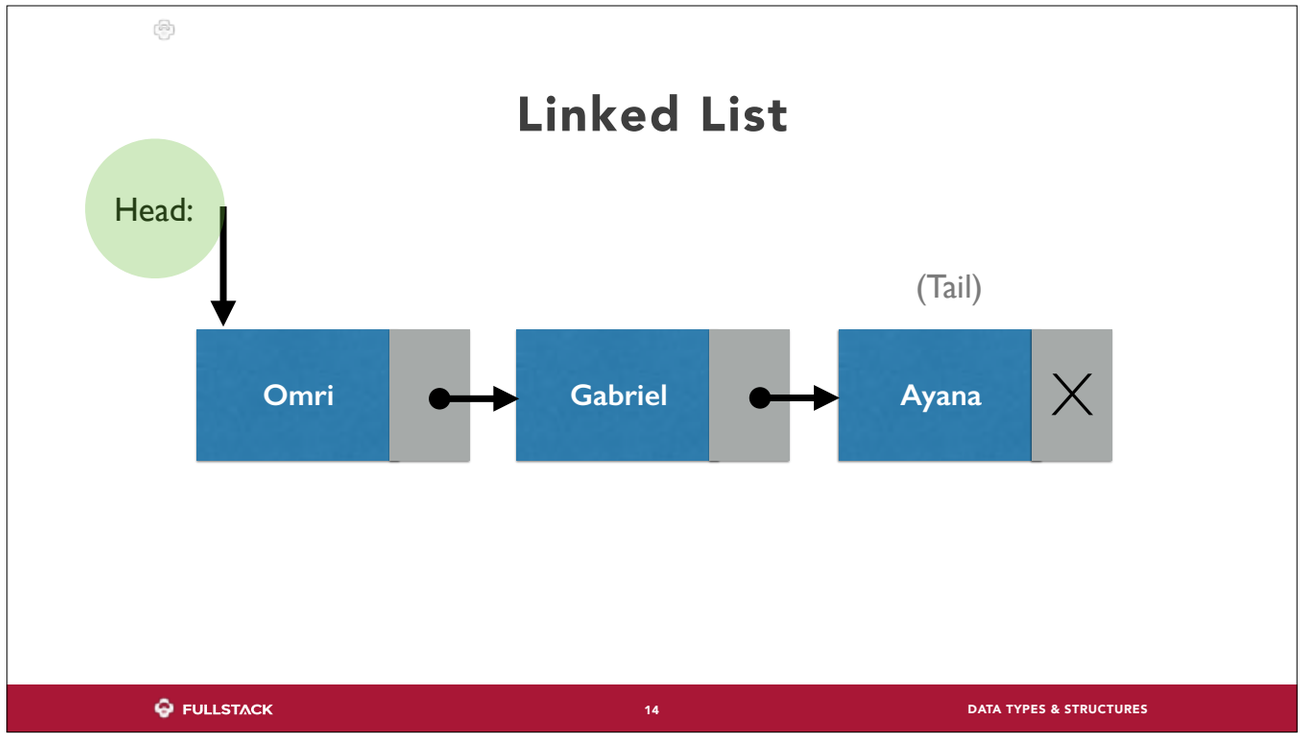
What if I want to add something in the middle of the list?



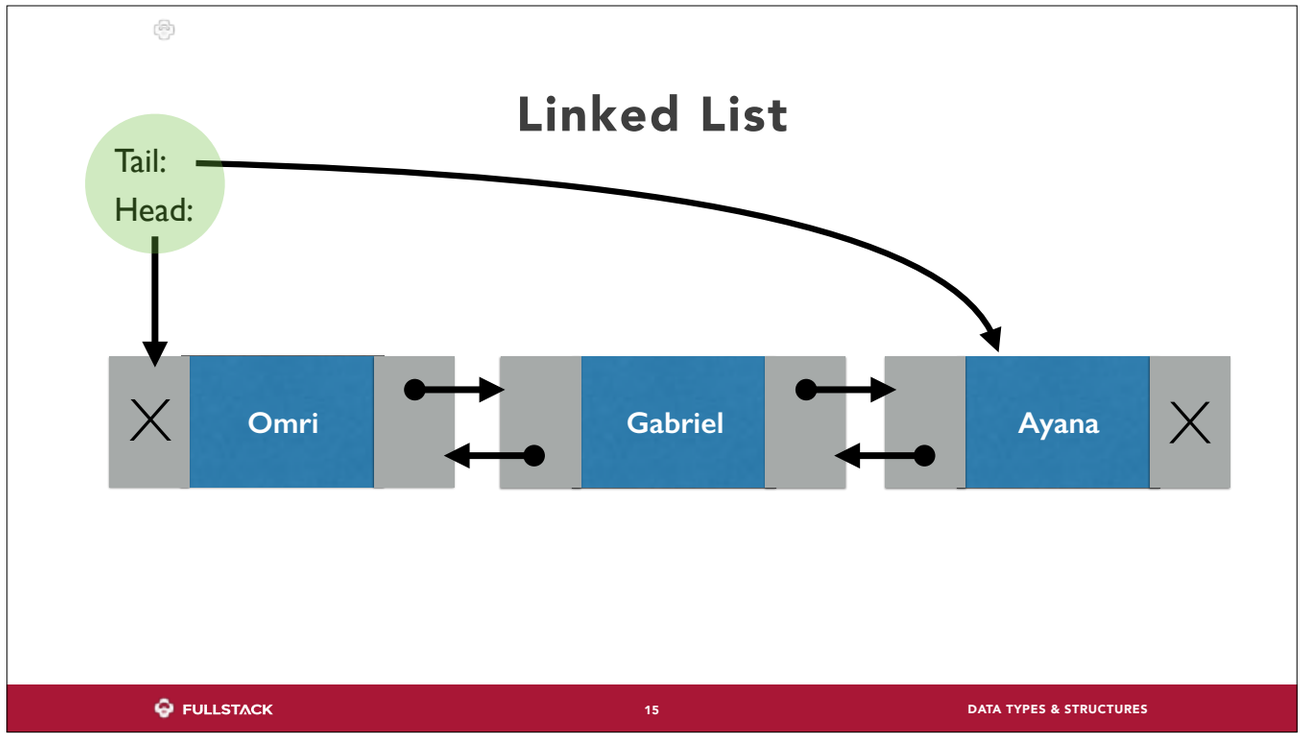
The best thing about Linked Lists is that *insertion* (adding in the middle) always takes the same number of steps, regardless of how big the list is. Again, we call that "constant time." First, point the new node's `next` to the old later node.



Then, change the previous node's `next` to point to the new node.


Linked List
Head:
(Tail)
Omri
Gabriel
Ayana
FULLSTACK
14
DATA TYPES & STRUCTURES

That's it!



Here is an example of a doubly-linked list with head and tail pointers. Think about the pros and cons of this approach.

(some) pros/cons: Linked Lists vs Arrays

Operation	Linked List	Typed Array
Reach element in middle	Must crawl though nodes	Constant time
Insert in middle or start	Constant time (if we have ref).	Must move all following elements
Add element to end	With handle, constant time	Constant time
Space per element	Container + element + pointer(s)	Just element!
Total space	Grows as needed	Pre-reserved & limited*
Physical locality	Not likely	Best possible

**Dynamic* arrays keep track of their limit, and if it is exceeded, the array is copied into a new bigger array (which is expensive but hopefully happens infrequently).

Part 1 of the workshop!