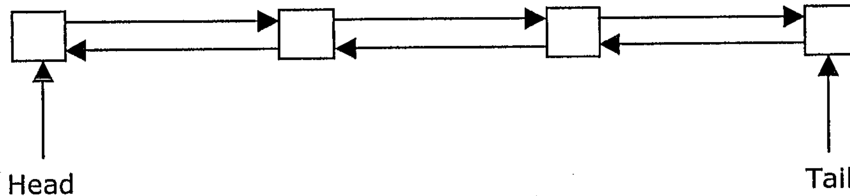
Linked Lists

In many languages it's the only alternative to an array.
It's another way to conceptualize or implement some aggregation of data.

The items in a linked list are not necessarily contiguous in memory, which may help us use memory more efficiently. (e.g. if the available memory is scattered in different segments)

Here is the idea:



Some operations are slower (like returning the 16th node), but a couple are actually faster (such as combining 2 lists, which won't require a loop).

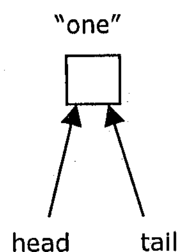
Java has a built-in LinkedList class in the API, but let's take time to understand what linked lists look like, before examining their implementation.

Q: How can you have an aggregation without an array or ArrayList?
A: Each node in our list has <u>pointers</u> (references) to the previous and next element in the list. These pointers could be null, as appropriate. For example, for the head of the list, there is no previous, so we would say that the head's previous is null. Similarly, the tail's next is also null. Maintaining 2 pointers for each element of the list will make its representation more complicated than an array.

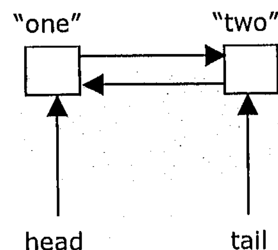
Q: How do you begin with an empty list?
A: Start with special "head" and "tail" references, and set them to null.
    head = null
    tail = null

Q: How do you insert the first node in the list?
A: Let's say the first object in our list is called "one". We would say:
    Head = one
    Tail = one
   And in the "one" object itself, we would set its previous and next pointers to null:
    one.prev = null
    one.next = null

Q: And the 2nd node? We want to go from

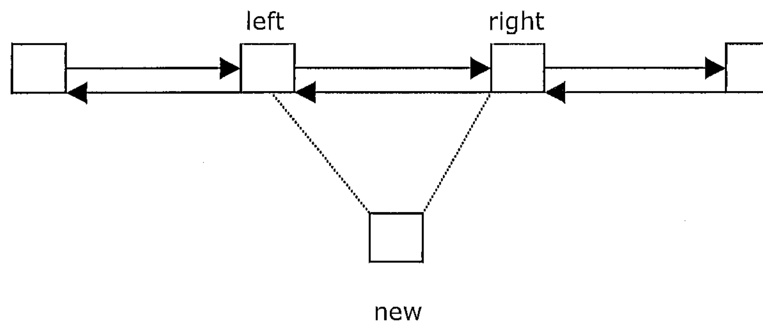


to this:



A: Here's how to add a 2nd element to the linked list. Just reset the pointers.
tail = two
one.next = two
two.prev = one
two.next = null
Note that we do not need to change the values of "head" or "one.prev".

Q: In general, how do you add a new object to the middle of a list?



A: Reset the pointers.
left.next = new (instead of right)
right.prev = new (instead of left)
new.prev = left
new.next = right

Also think about what would happen if we were adding something to the front of the list. If the "new" element had to go before "one" then we would say:
head = new (instead of one)
one.prev = new (instead of null)
new.next = one

Q: Okay, how about deleting a node from the list? Let's say that between elements "left" and "right" we have a node called "victim" that we want to delete.

A: In the usual case, we are deleting something from the middle of the list. In this case,
left.next = right
right.prev = left
And now there is nobody left pointing to the victim, so the Java garbage collector reclaims its memory.

Ah, but what if we are deleting the head or tail element of the list?
If we're deleting the head element, (there is no "left" in this case) we say:
head = right
right.prev = null

Similarly, if we're deleting the tail element, (there will be no "right"):
tail = left
left.next = null

Q: Can we implement our BagInterface using a linked list strategy? How would we implement: size(), indexOf(), remove(), combine(), equals(), sort(), swap()?
A: Let's think about those before answering!