# Arrays/Linked Lists Exercises

Download our starter code <../dsa-arrays-linked-lists.zip>.

We've supplied you with a ***Node*** class and a constructor for the ***LinkedList*** class. Here are descriptions of the methods you should write for instances of ***LinkedList***:

***push(val)***
Appends a new node with value ***val*** to the tail. Returns undefined.

***unshift(val)***
Add a new node with value ***val*** to the head. Returns undefined.

***pop()***
Remove & return tail value. Throws error if list is empty.

***shift()***
Remove & return head value. Throws error if list is empty.

***getAt(idx)***
Retrieve value at index position ***idx***. Throws error if index is invalid.

***setAt(idx, val)***
Set value of node at index position ***idx*** to ***val***. Throws error if index is invalid.

***insertAt(idx, val)***
Insert a new node at position ***idx*** with value ***val***. Throws error if index is invalid. Returns undefined.

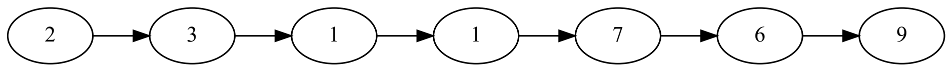
***removeAt(idx)***
Remove & return value at position ***idx***. Throws error if index is invalid.

> **Warning: Go Slow & Check Your Code!**
>
> It's very easy to make methods that don't work for every case — make sure you properly handle cases of items being at the start, middle, or end of the list, as well as handling empty lists.

## Average Of List

Given a linked list containing numbers, return the average of those numbers.

For example:



would return 4.142857142857143.

### Further Study

## Doubly Linked Lists

Doubly Linked Lists are just like Singly Linked Lists, but each node has a pointer to the previous node as well as the next one. Implement a class for ***DoublyLinkedList*** with the same methods as above (be mindful of opportunities to speed up your code now that each node has two pointers!)

## Reverse In Place

Write a function that reverses a linked list *in place* — not by creating a new list or new nodes.

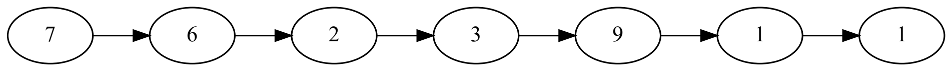
## Sort Sorted Linked Lists

Write a function that is passed two linked lists, ***a*** and ***b***, both of which are already sorted.

It should return a *new* linked list, in sorted order.

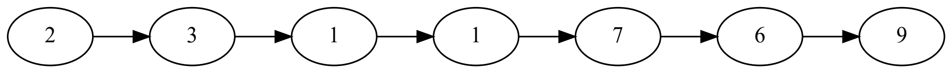
## Pivot Around Value

Imagine we have a singly-linked linked list:



In this challenge, you'll be given a value and you want to rearrange the items in the linked list so that all items with data less than the given value are in the first half, and items with greater than or equal to the given value are in the second half.

For example, for the value 5:



Notice that this list *isn't sorted*; all we need to do is "pivot" it around the given value. Otherwise, items are still in the same order as they were (7 came before 6 in the original list, so it still does — but both of them are greater than 5, so they appear in the second half).

For example:

```
let ll = new LinkedList([7, 6, 2, 3, 9, 1, 1])

ll.pivot(5)

// now list is 2 3 1 1 7 6 9
```

If the given pivot value is in the list, it should appear in the second half (with other greater-than-or-equal-to values):

```
let ll = new LinkedList([7, 6, 2, 5, 3, 5, 9, 1, 1])

ll.pivot(5)

//  now list is 2 3 1 1 7 6 5 5 9
```

## Circular Arrays

In this challenge, you will create a “circular array” — like a list ADT but the end wraps around to the beginning (which makes for some interesting problems).

A circular array is defined by having a start and indexes (be sure to think about **optimizing runtime** for indexing, since we’ll do this so much more often than adding items to it):

```
let circ = new CircularArray()
circ.addItem('harry')
circ.addItem('hermione')
circ.addItem('ginny')
circ.addItem('ron')

circ.printArray()
// harry
// hermione
// ginny
// ron

circ.getByIndex(2)  // ginny
circ.getByIndex(15) // null
```

Because the last item circles back around to the first item, you can rotate the list and shift the indexes. Positive numbers rotate the list start (defined as the index 0) to the right (or higher indexes):

```
let circ = new CircularArray()
circ.addItem('harry')
circ.addItem('hermione')
circ.addItem('ginny')
circ.addItem('ron')

circ.rotate(1)
circ.printArray()
// hermione
// ginny
// ron
// harry
```

```
circ.getByIndex(2)  // ron
```

And negative numbers rotate the list start to the left (or lower indexes):

```
let circ = new CircularArray()
circ.addItem('harry')
circ.addItem('hermione')
circ.addItem('ginny')
circ.addItem('ron')

circ.rotate(-1)
circ.printArray()
// ron
// harry
// hermione
// ginny

circ.getByIndex(2)  // hermione
```

And you can also rotate more than once around the ring:

```
let circ = new CircularArray()
circ.addItem('harry')
circ.addItem('hermione')
circ.addItem('ginny')
circ.addItem('ron')

circ.rotate(-17)
circ.getByIndex(1)  // harry
```

If you add a new item after rotating, it should go at the **end of the list in its current rotation**:

```
let circ = new CircularArray()
circ.addItem('harry')
circ.addItem('hermione')
circ.addItem('ginny')
circ.addItem('ron')

circ.rotate(-2)
circ.addItem('dobby')

circ.printArray()
// ginny
// ron
// harry
// hermione
// dobby
```

Want a hint about the data structure?

**Data Structure**

Think about the data structure you’d want to use to store the items.

While it’s tempting to use something like a Linked List, the runtime to find an item by index in a linked list is ***O(n)***.

You can use a standard array to store the items, but you’ll need to think about how to keep track of the head and handle rotations.

## Solution

See Our solution <solution/index.html>.