CATALYST

[2:3] Linked Lists, Stacks, Queues

Helpful Knowledge

CS308

- Abstract data structures vs concrete data types

CS250

- Memory management (stack)
- Pointers

CS230

- Modular Arithmetic

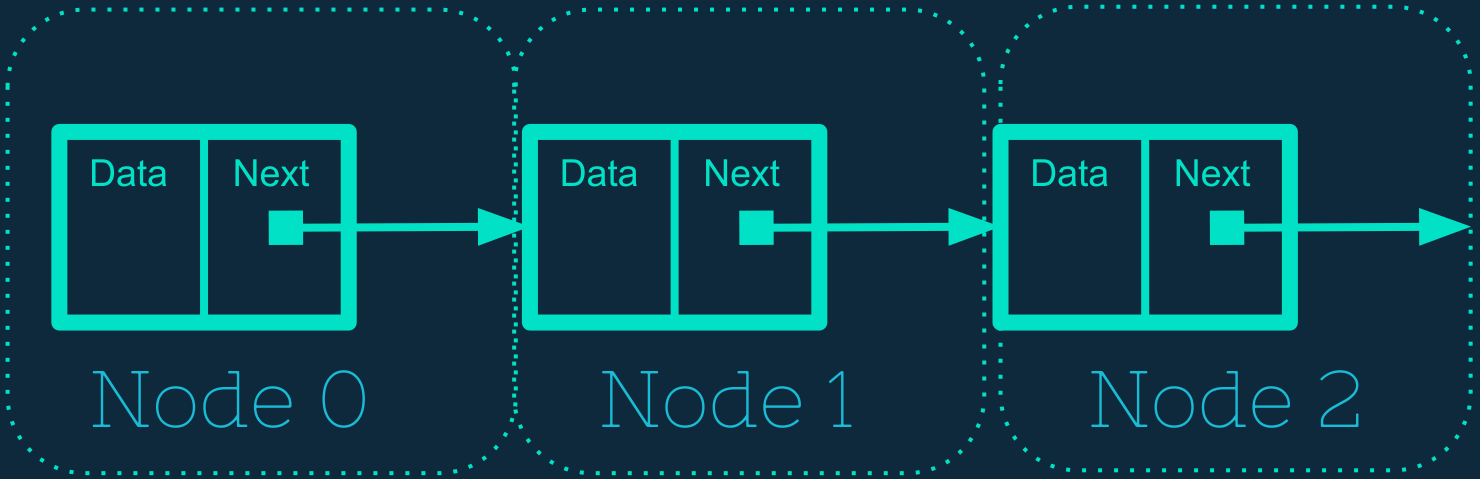
! ! ! ! !

There’s a lot of slides, but fear not - we’re taking two weeks to cover these VERY important topics.

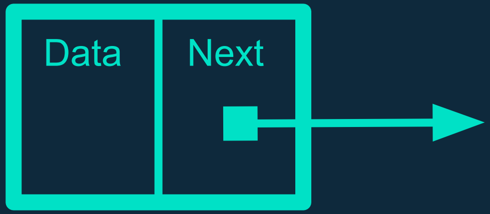
1

Singly Linked Lists

Concept 1: Singly Linked Lists are made of NODES

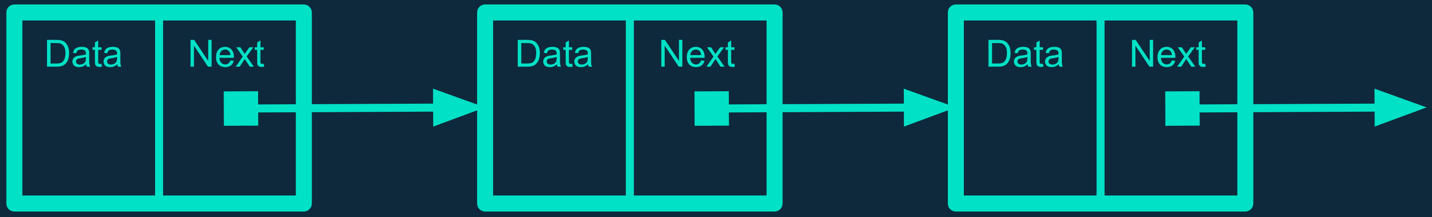


Concept 2: Each node has Data and a Next Pointer



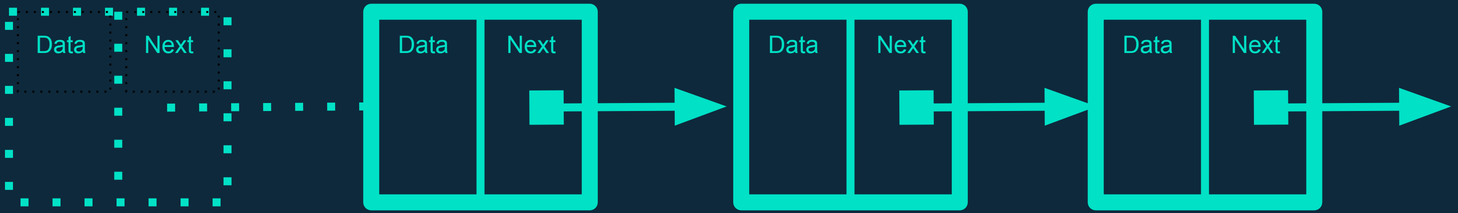
- Data can be any type - our examples will all be integers, but a LL can store strings, pointers, chars, etc.
- Next is either NULL or a pointer to another LL node. Our diagrams will show “next” as an arrow pointing to either nothing, or another node.

Concept 3: Access to SLL is given via a head



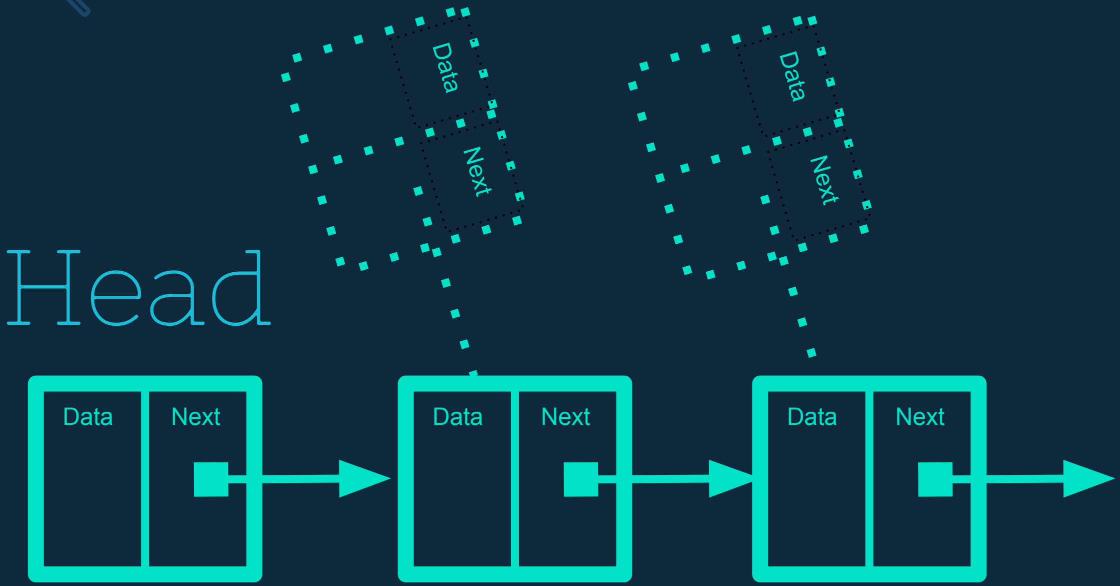
Head

Corollary: There may be things "before" your head in a larger list, and there's no way for you to know

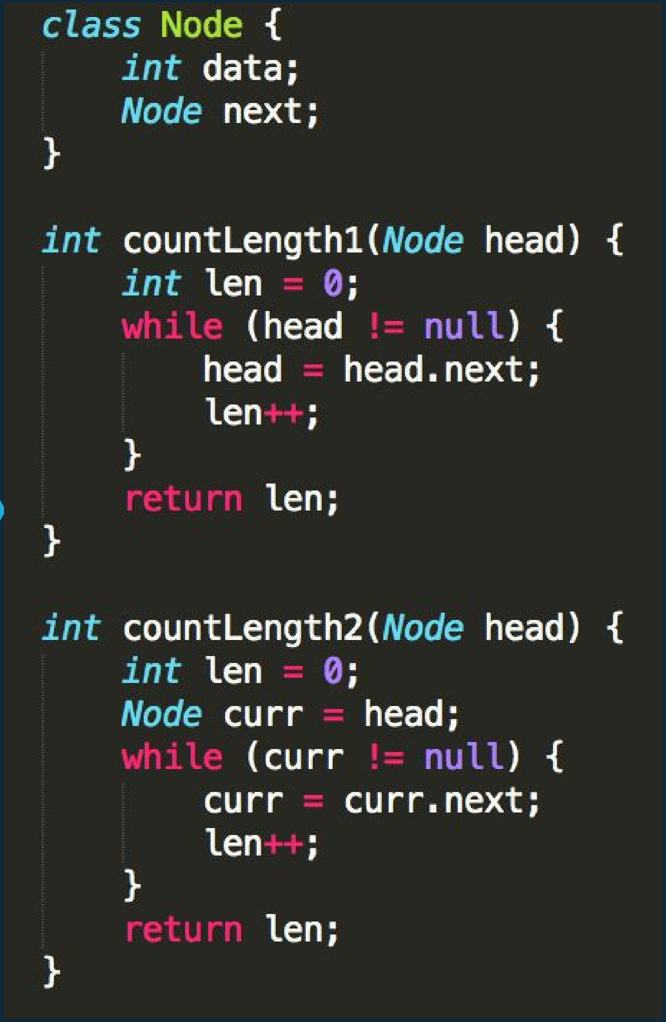


Head

Corollary: Many nodes may point to one node. However, one node cannot have multiple "next" pointers



Intro



```
class Node {
    int data;
    Node next;
}

int countLength1(Node head) {
    int len = 0;
    while (head != null) {
        head = head.next;
        len++;
    }
    return len;
}

int countLength2(Node head) {
    int len = 0;
    Node curr = head;
    while (curr != null) {
        curr = curr.next;
        len++;
    }
    return len;
}
```

countLength2 is better because even though both functions do the same thing using the same O(n) runtime and O(1) extra space, the second one **doesn’t mutate the input.**

CATALYST

Intro

Middle of SLL

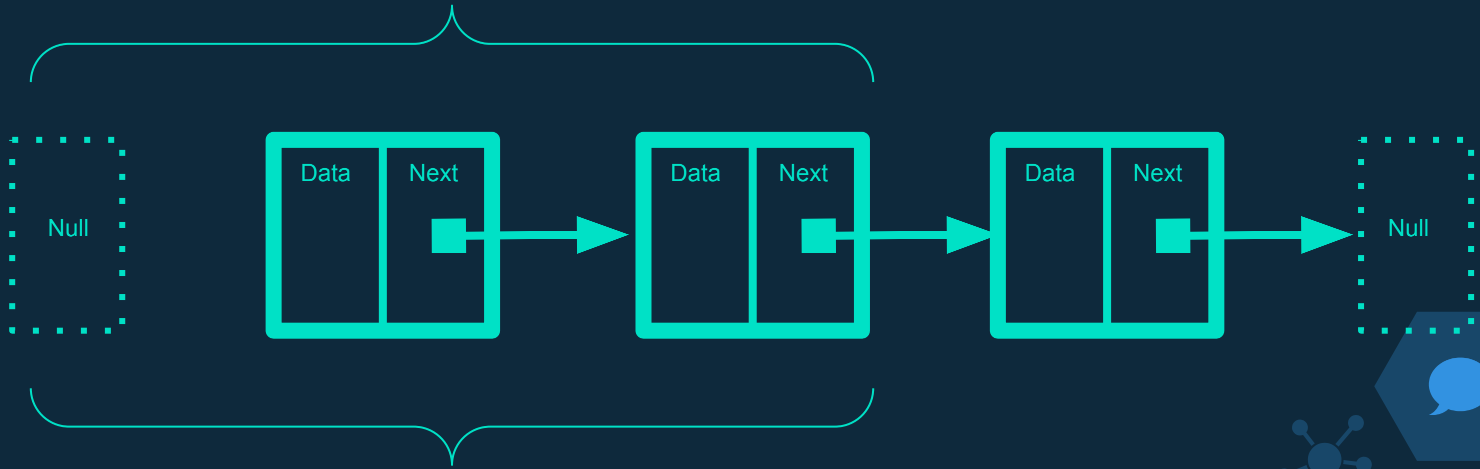
Return the middle node of a singly linked list. Can you do it using only ONE pass of the list?

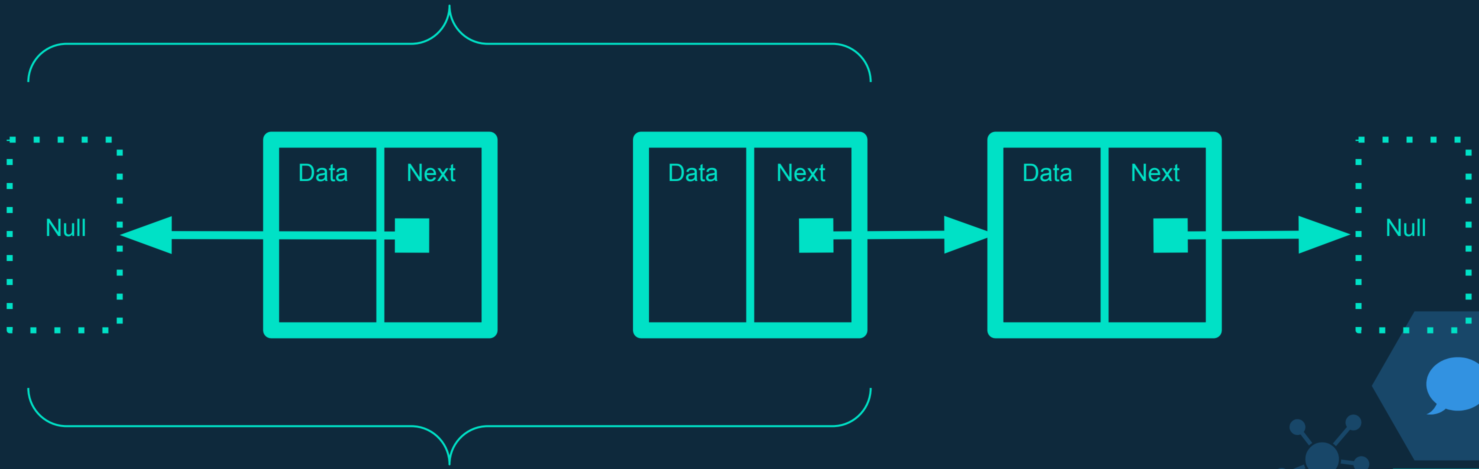
Related: return the kth to last node of a SLL

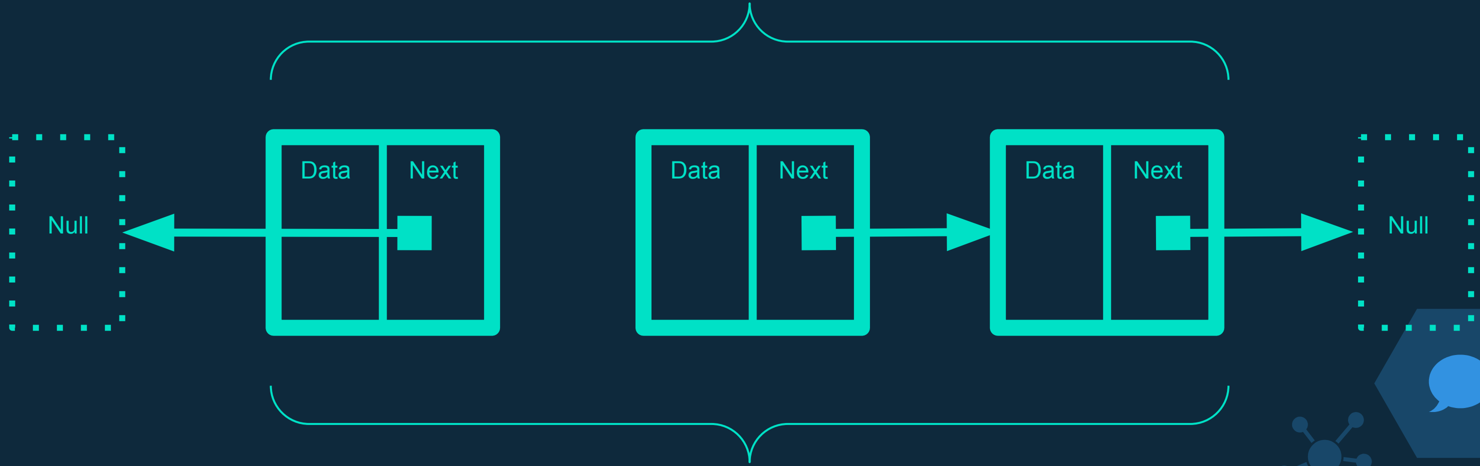
Reverse SLL

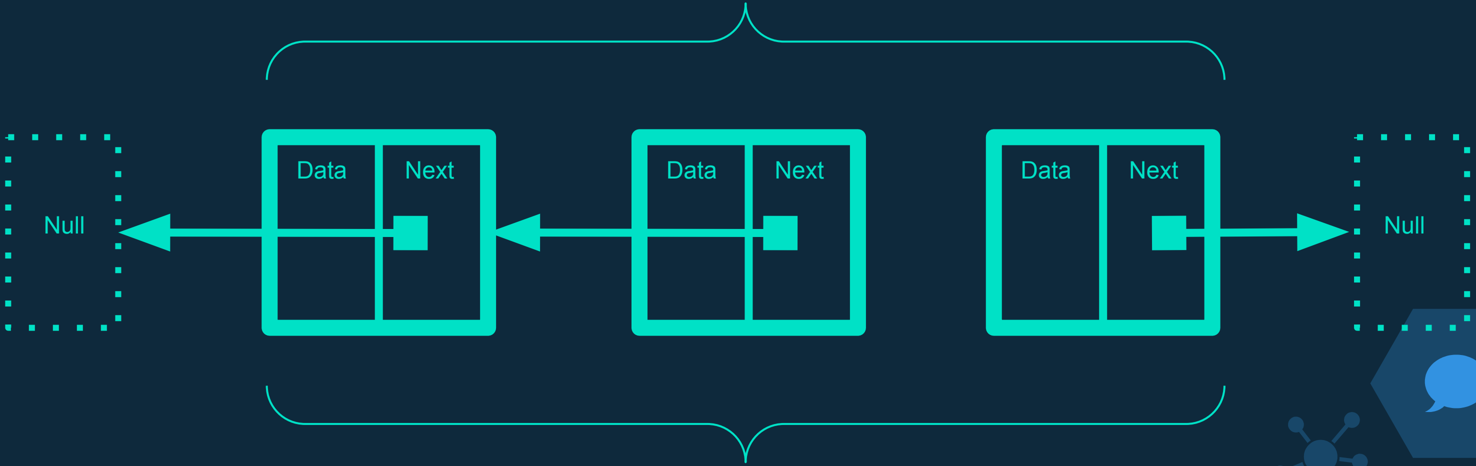
Med

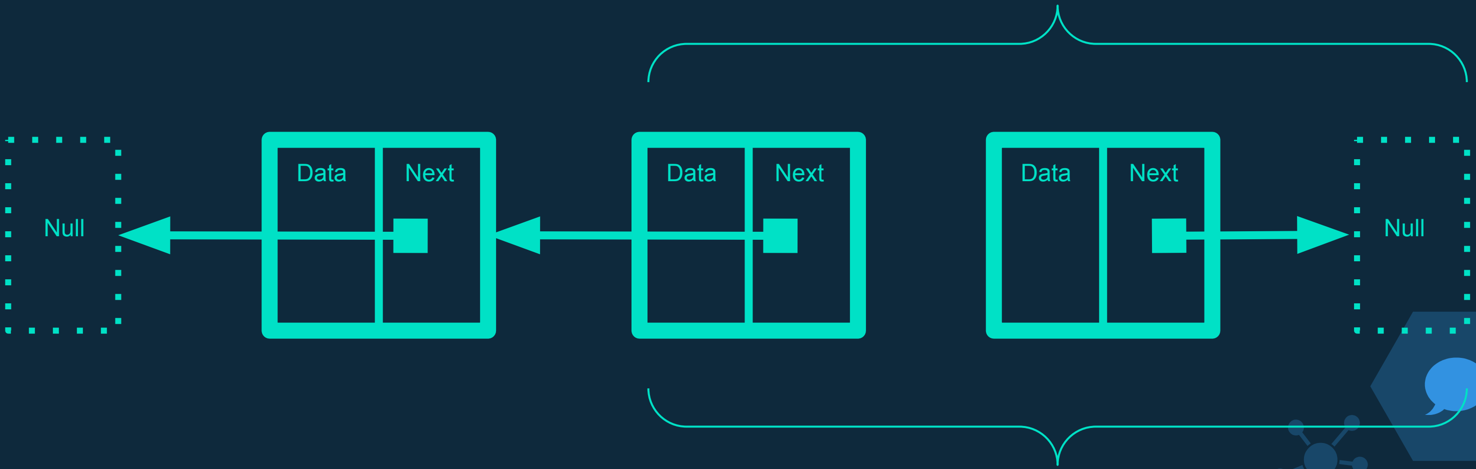
Reverse a SLL in-place with O(1) space complexity.

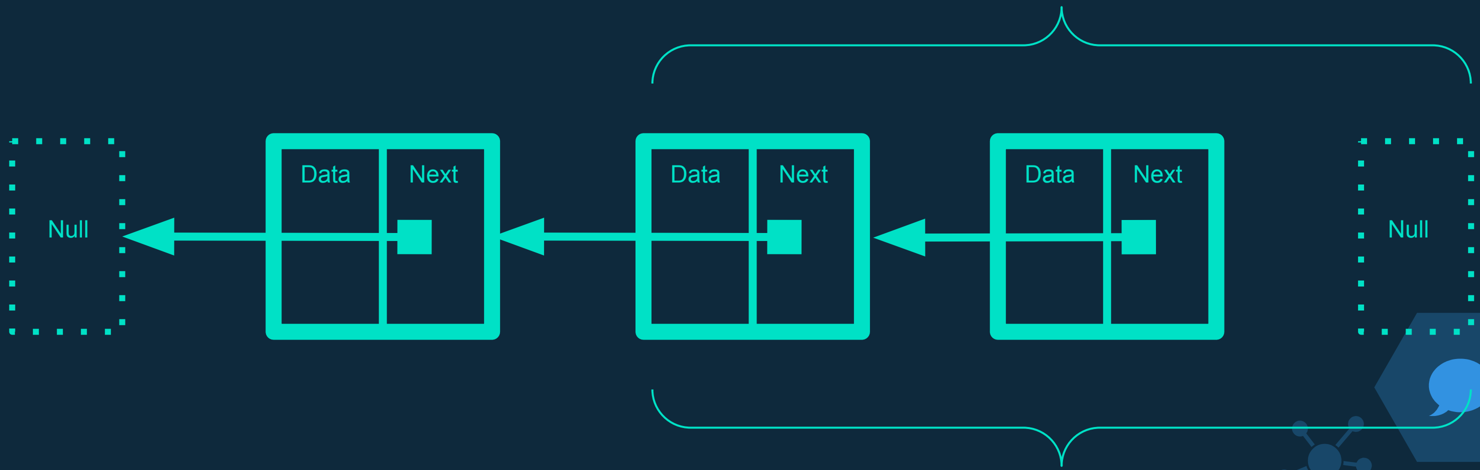

Null
Data
Next
Data
Next
Data
Next
Null


Null
Data
Next
Data
Next
Data
Next
Null


Null
Data
Next
Data
Next
Data
Next
Null

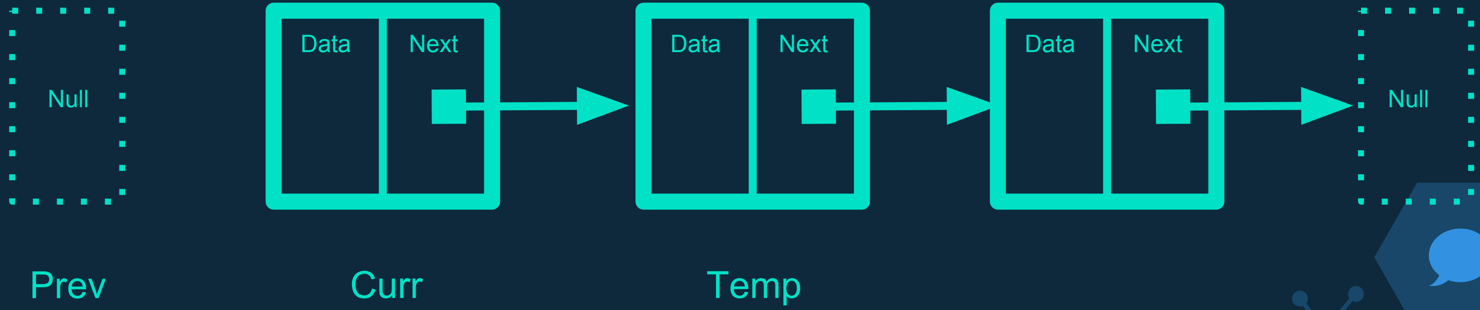

Null
Data
Next
Data
Next
Data
Next
Null


Null
Data
Next
Data
Next
Data
Next
Null


Null
Data
Next
Data
Next
Data
Next
Null

Brackets = Pointers

Let’s re-visit the process using pointers



Null

Data

Next

Data

Next

Data

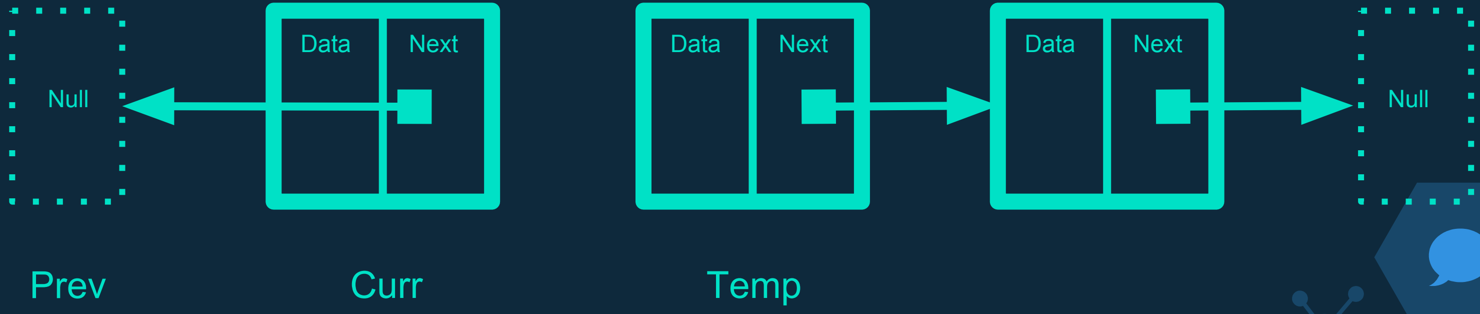
Next

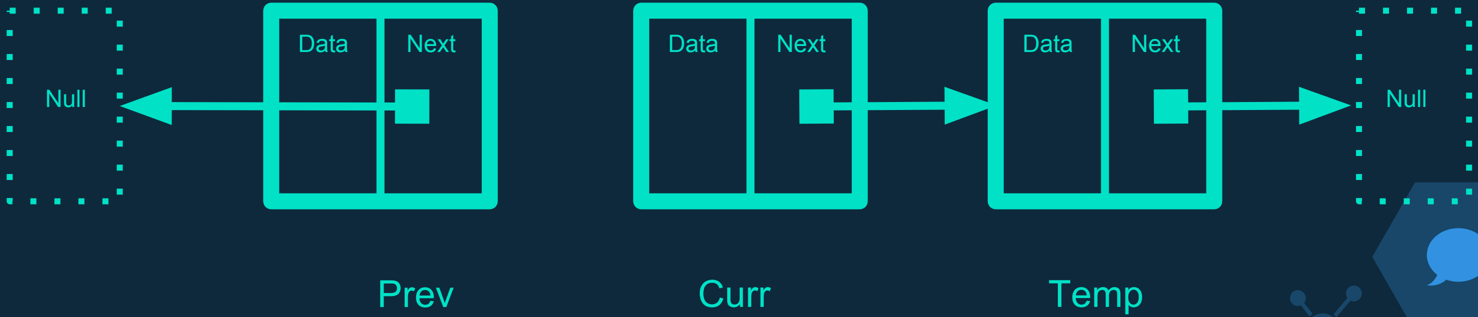
Null

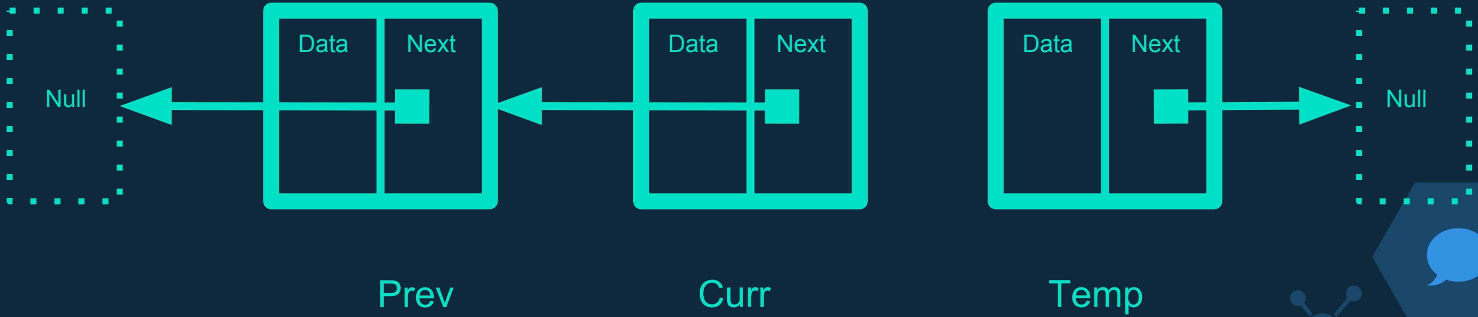
Prev

Curr

Temp


Null
Data
Next
Data
Next
Data
Next
Null
Prev
Curr
Temp


Null
Data
Next
Data
Next
Data
Next
Null
Prev
Curr
Temp



Null

Data Next

Data Next

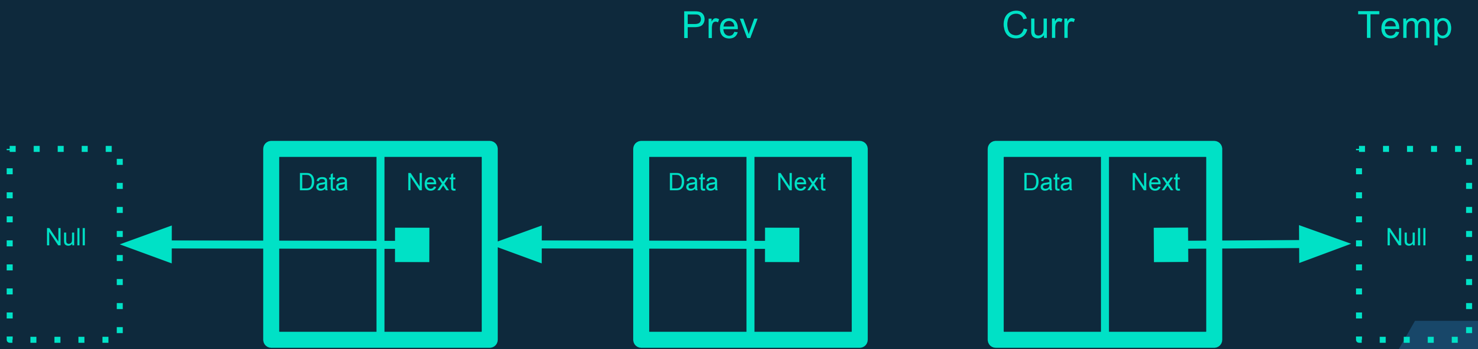
Data Next

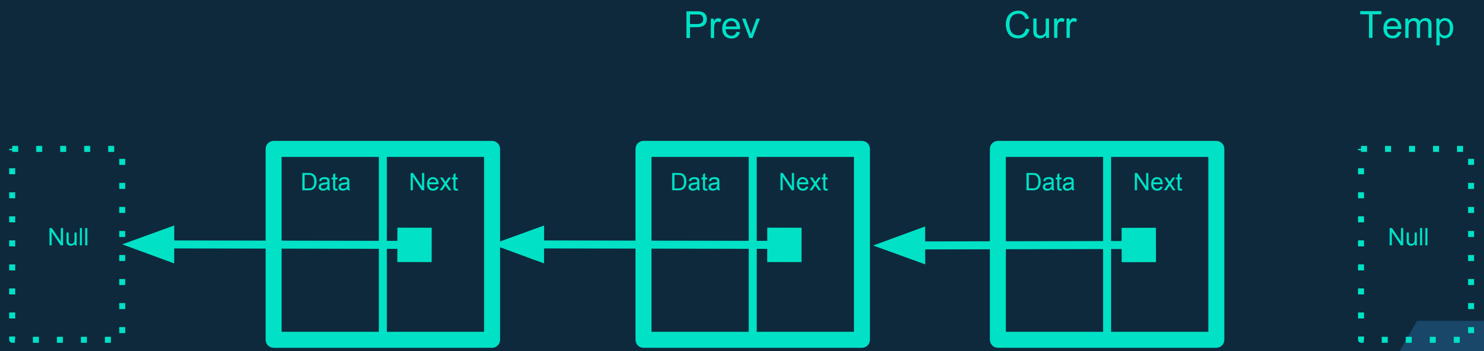
Null

Prev

Curr

Temp


Prev
Curr
Temp
Null
Data
Next
Data
Next
Data
Next
Null



Prev

Curr

Temp

Null

Data

Next

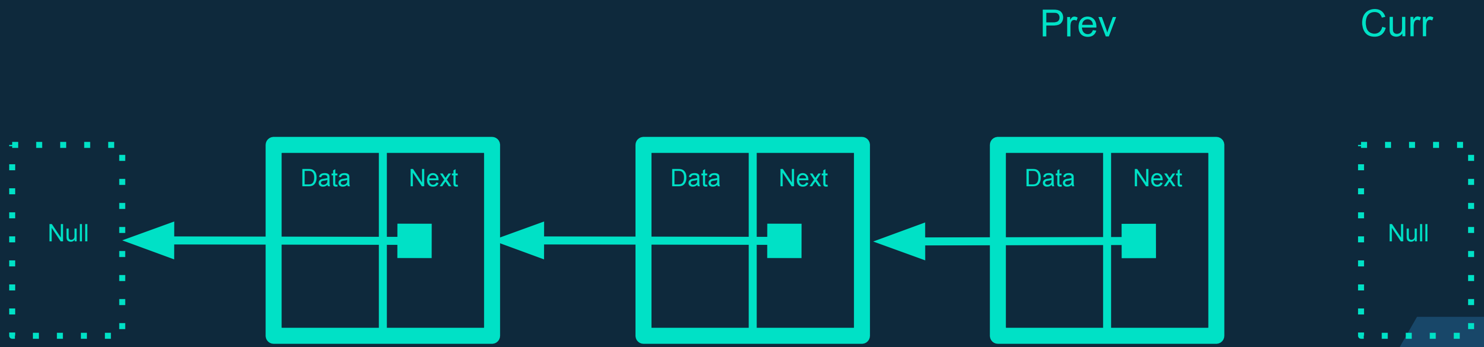
Data

Next

Data

Next

Null


Prev
Curr
Null
Data
Next
Data
Next
Data
Next
Null

Interviewing Tactics I

Using Extra Variables

Because you can’t “go back” in a SLL, use a traverser variable so you don’t lose track of the head, or any other important nodes

No Recursion

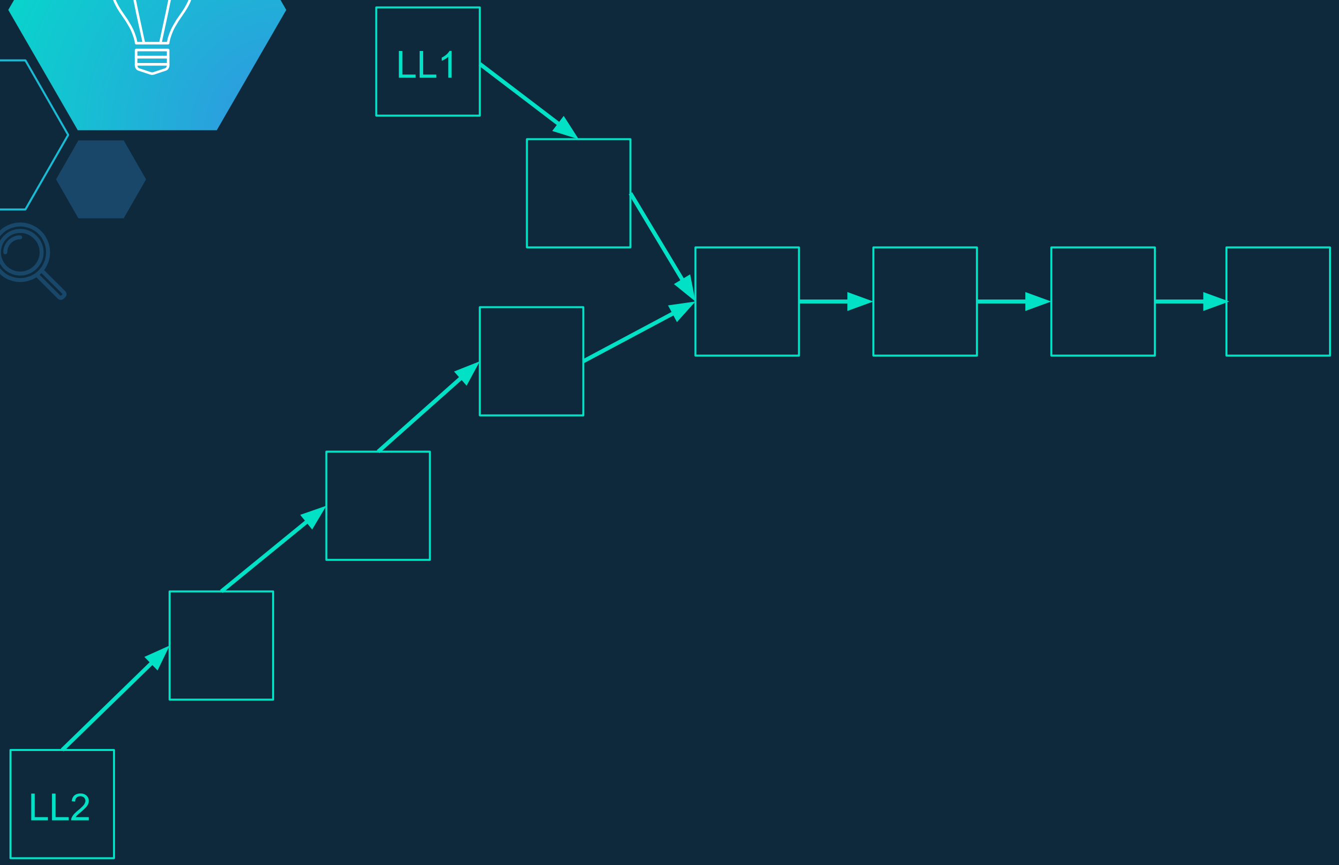
On a similar note, it’s possible to solve all these problems using recursion, but because you often lose the “first” part of your list through recursion, this is NOT recommended.

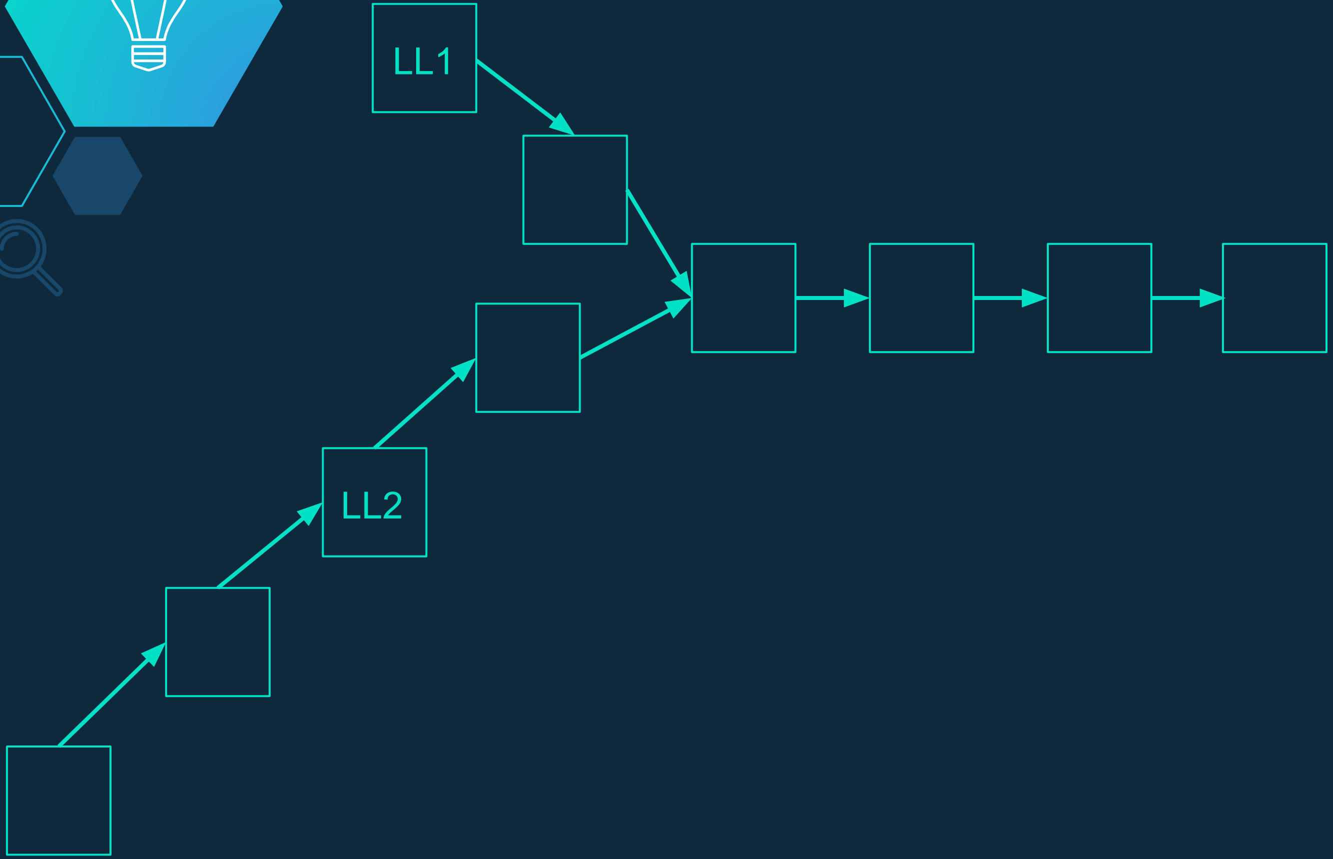
CATALYST

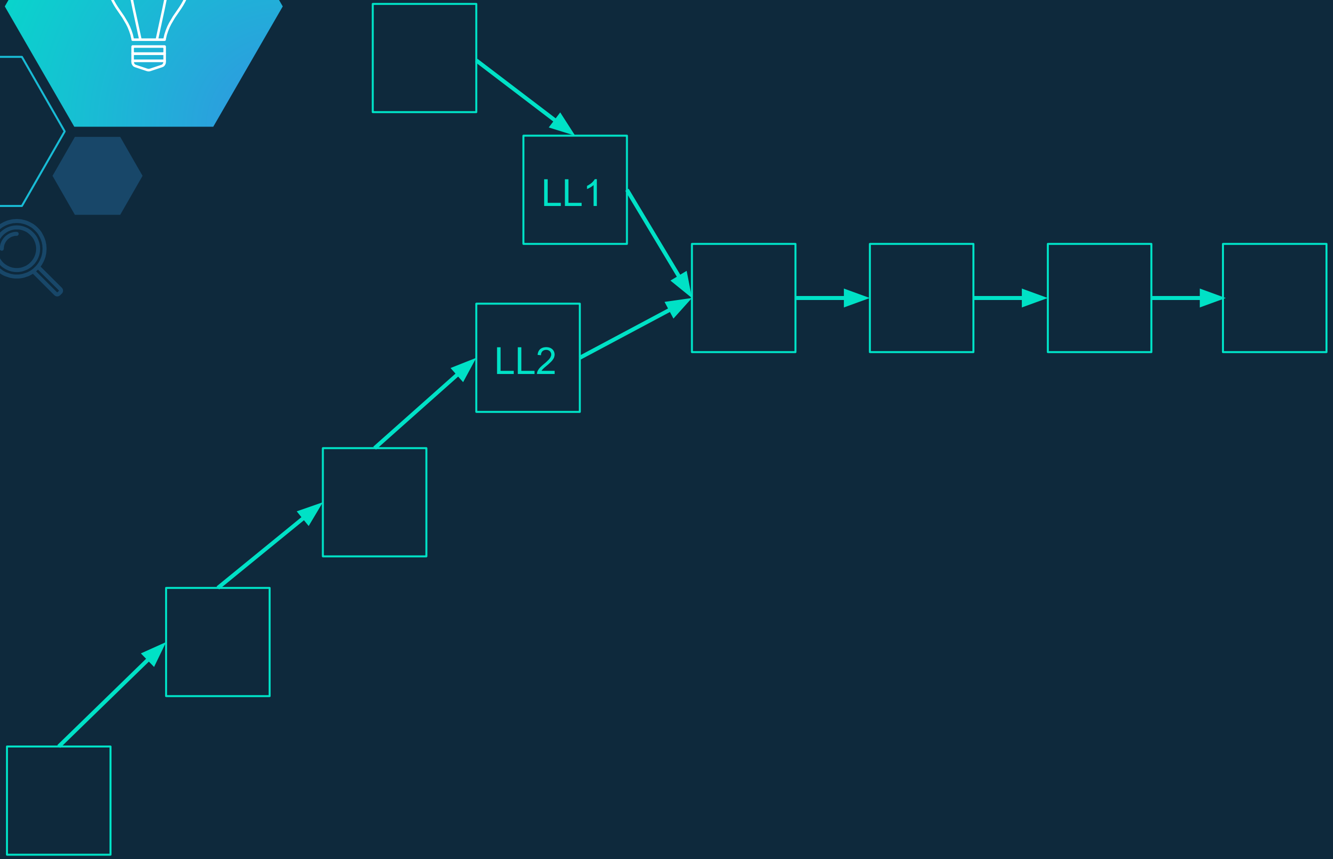
Intersection of SLL

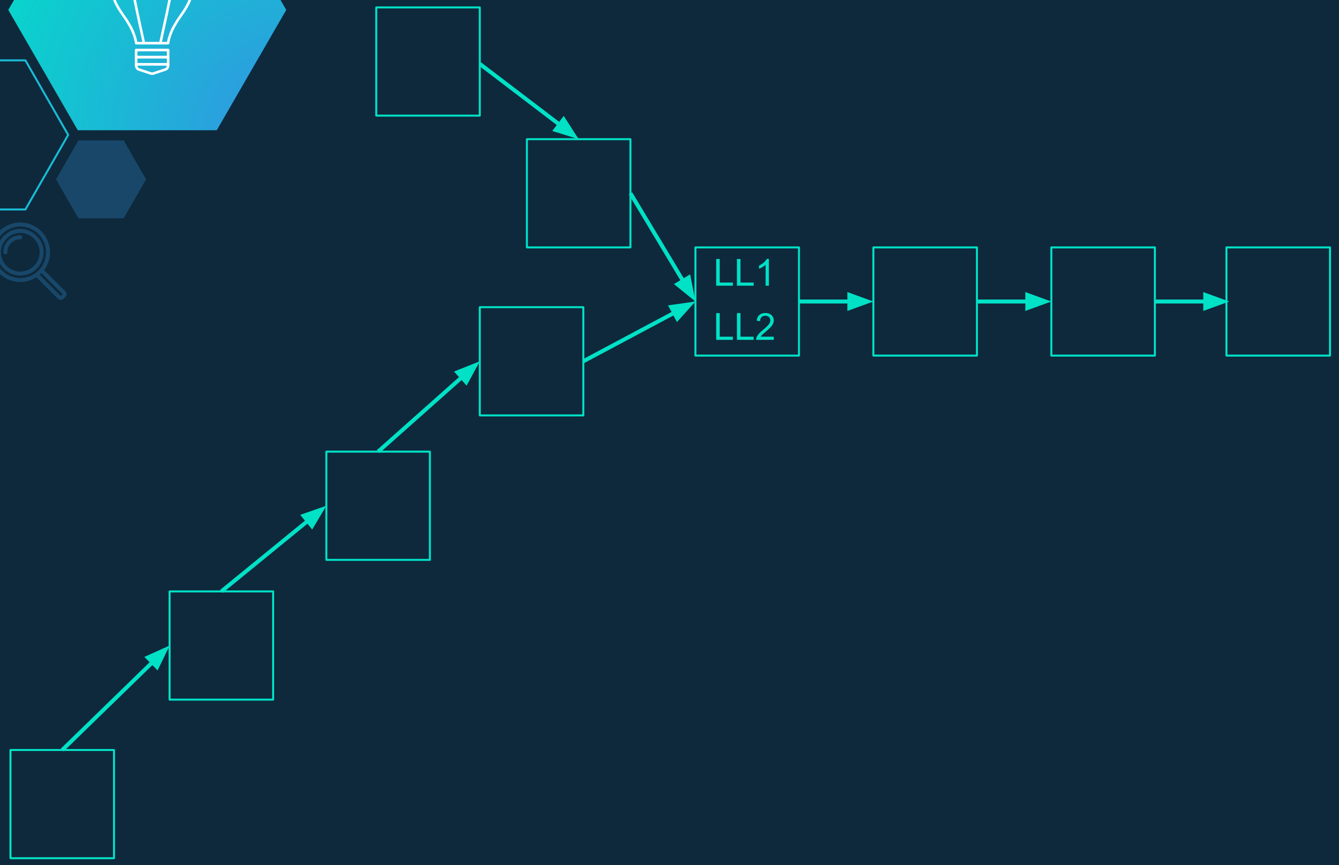
Med

I give you two SLL’s, which intersect at some node. Give me the first node which they intersect at.


LL1
LL2


LL1
LL2

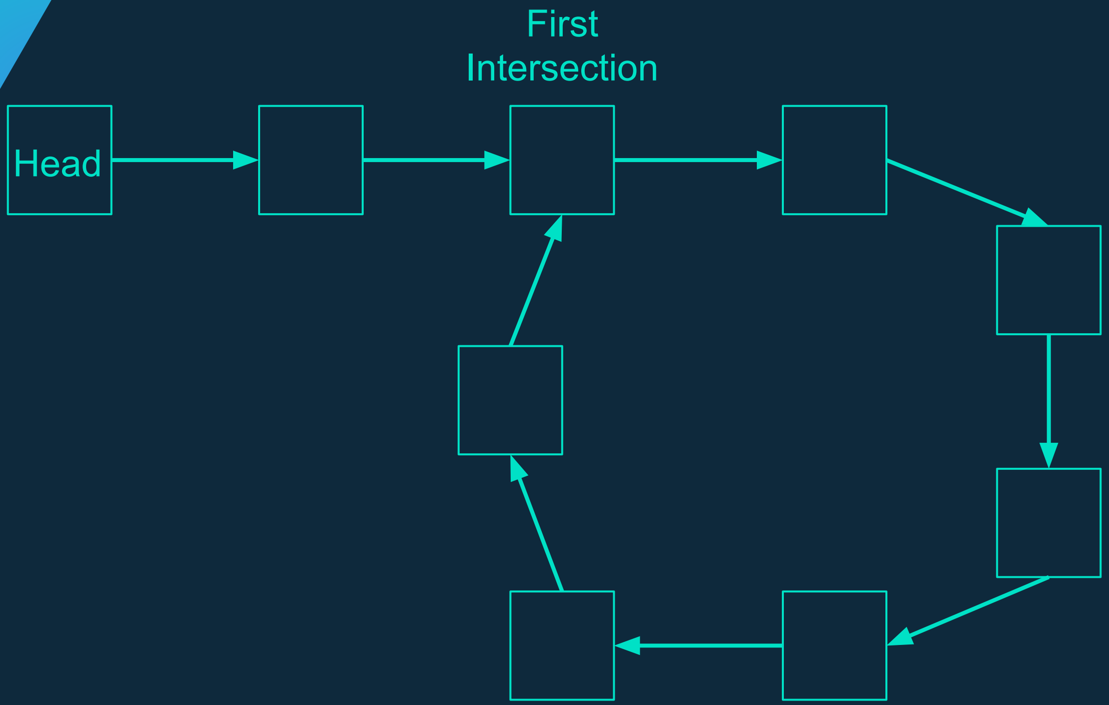

LL1
LL2


LL1
LL2

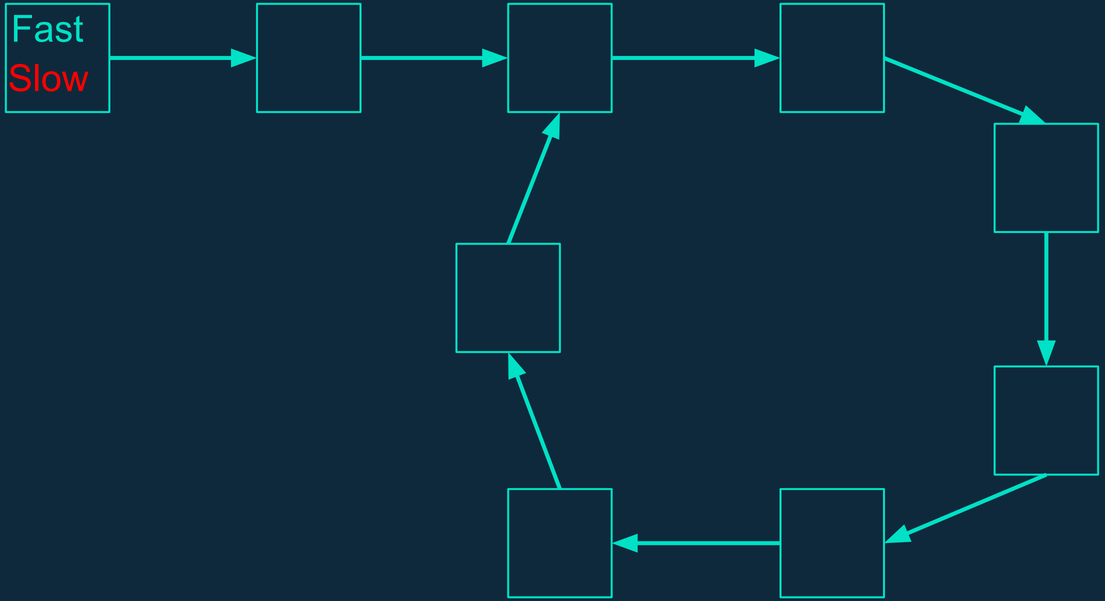
SLL Loop

Med

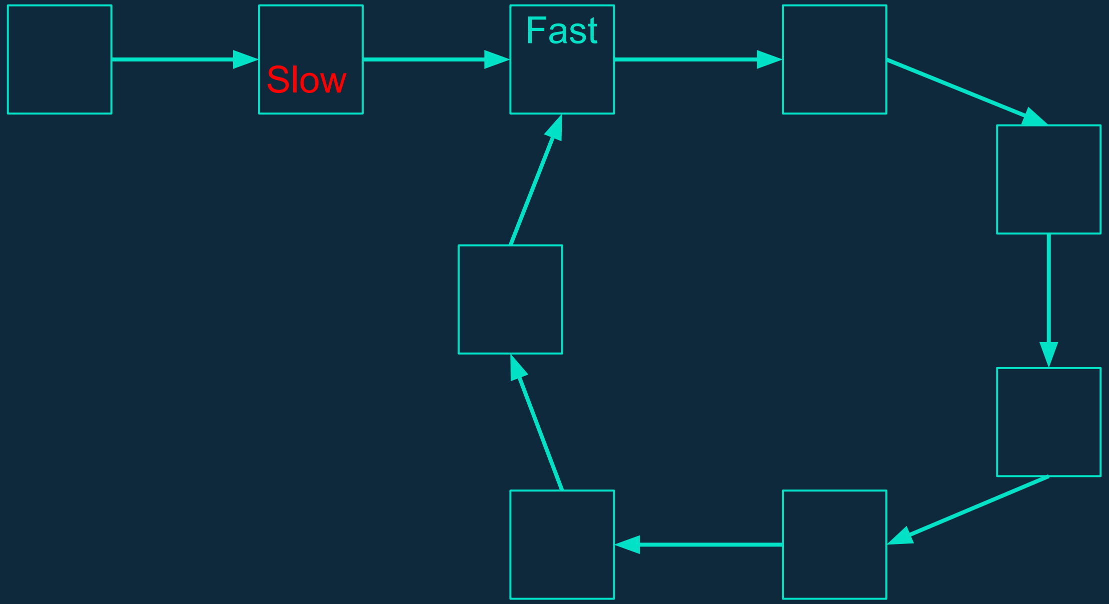
Given a SLL, determine if there’s a loop. If so, return the first node of the loop. Otherwise, return null.


First Intersection
Head

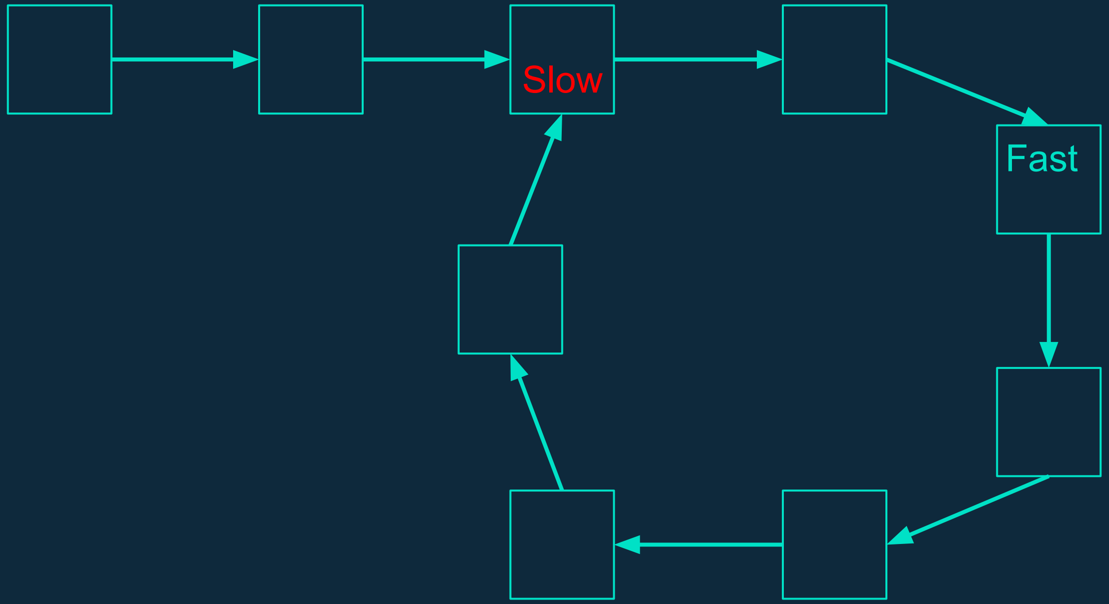
Part 1: Find whether loop exists



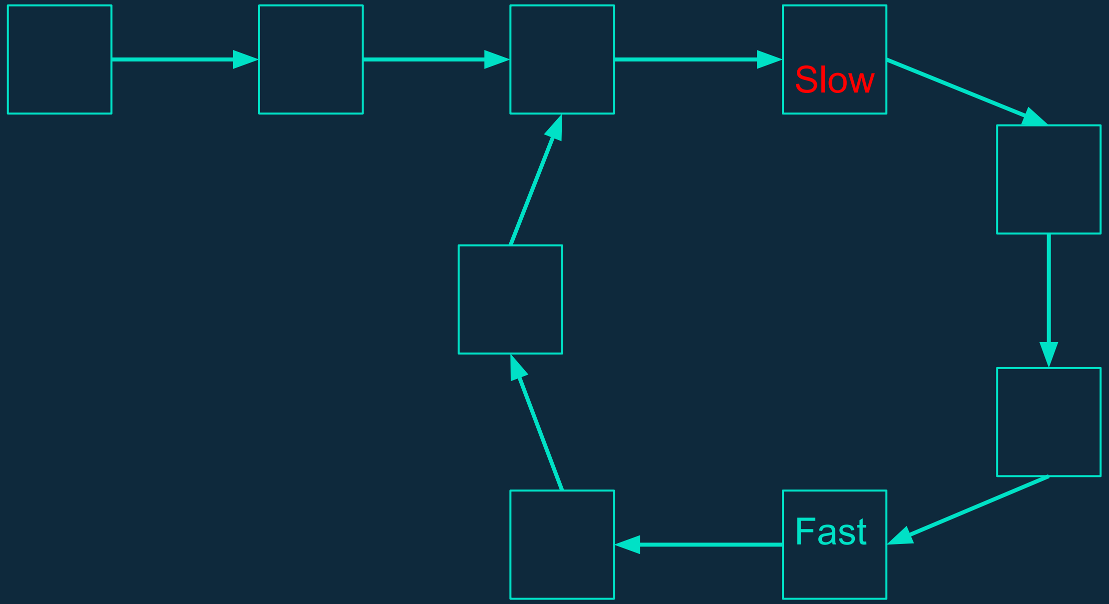
Part 1: Find whether loop exists



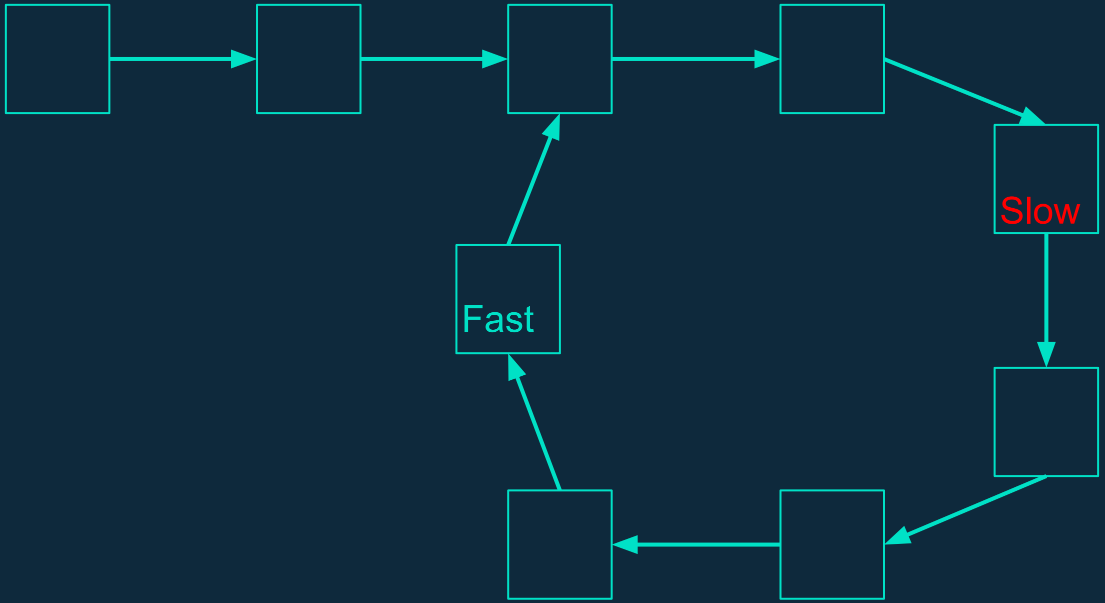
Part 1: Find whether loop exists



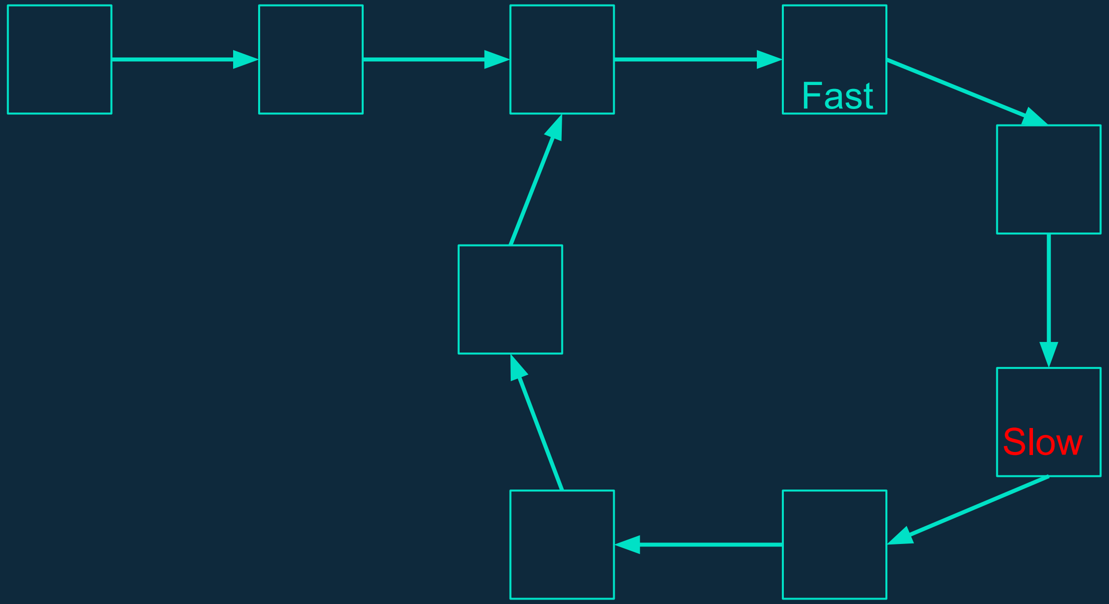
Part 1: Find whether loop exists



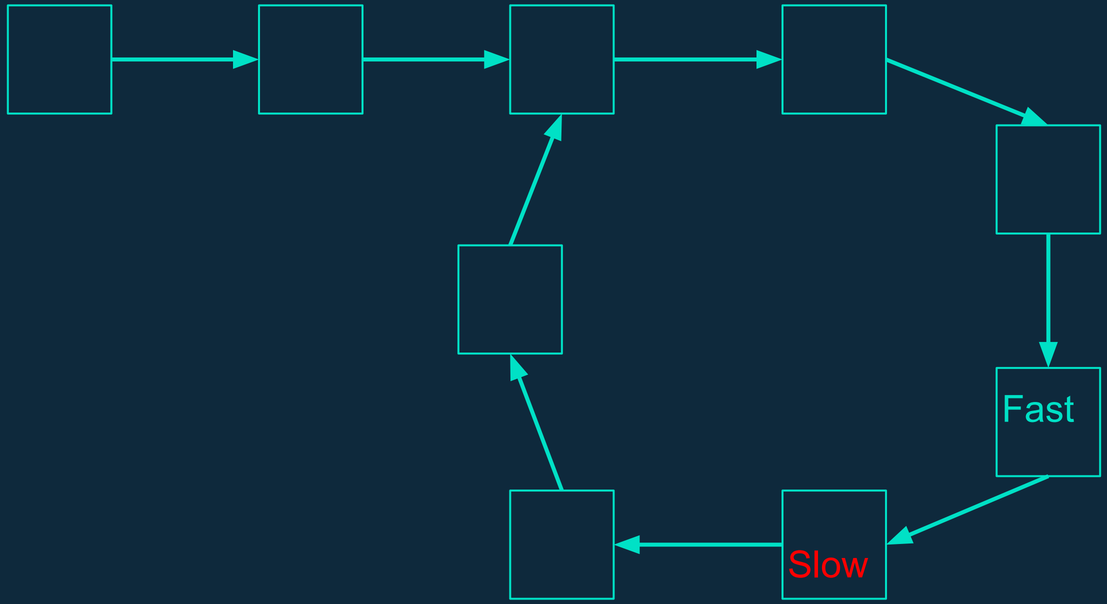
Part 1: Find whether loop exists



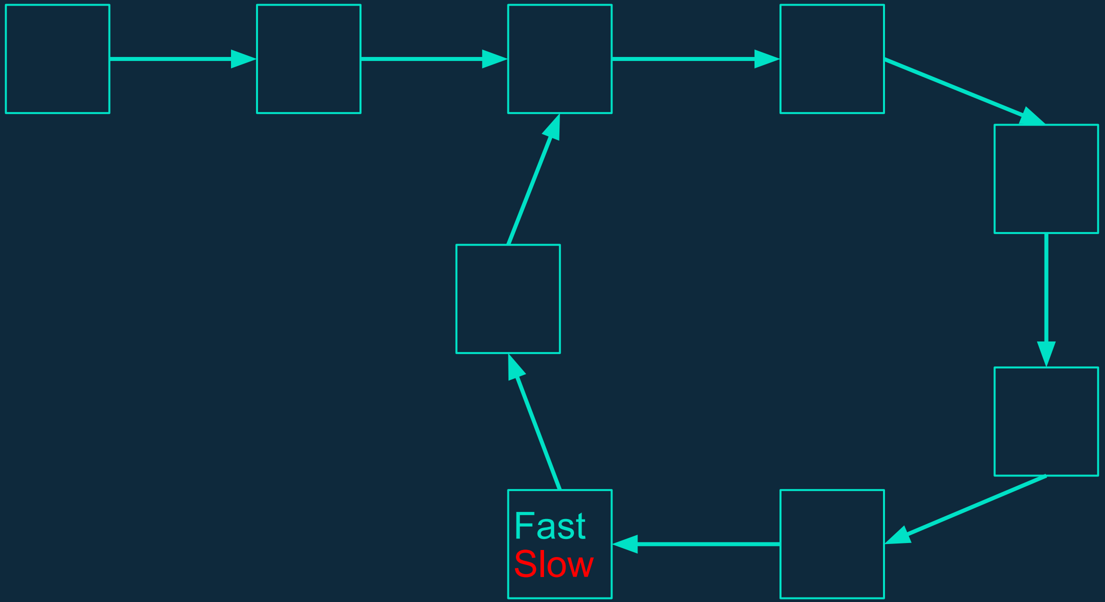
Part 1: Find whether loop exists



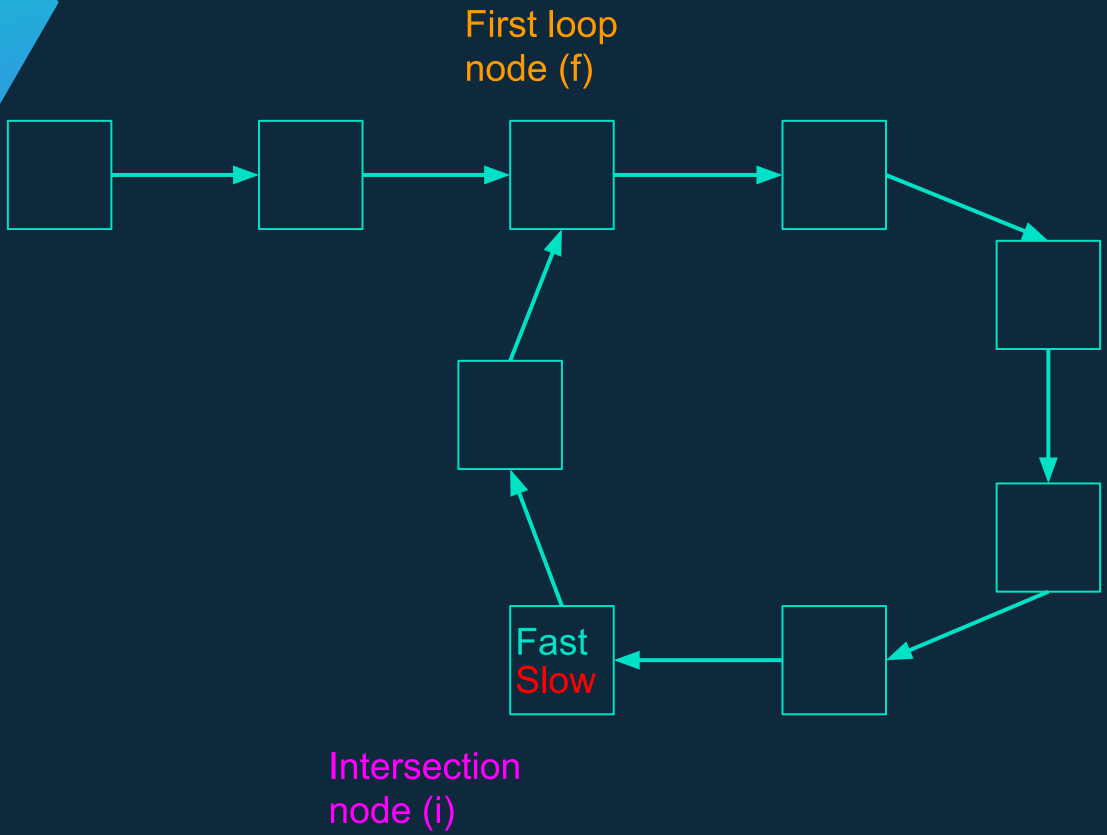
Part 1: Find whether loop exists



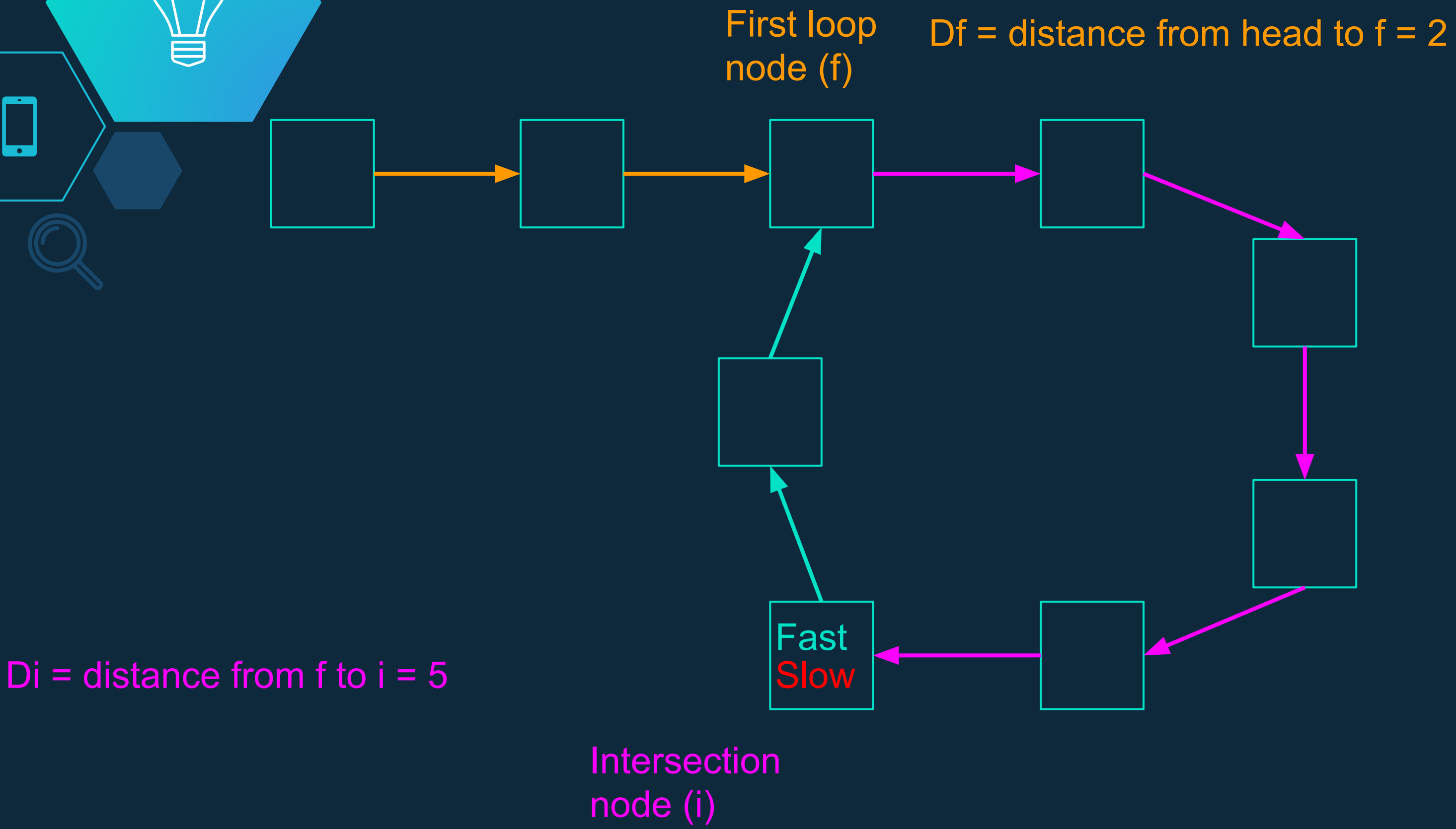
Part 1: Find whether loop exists



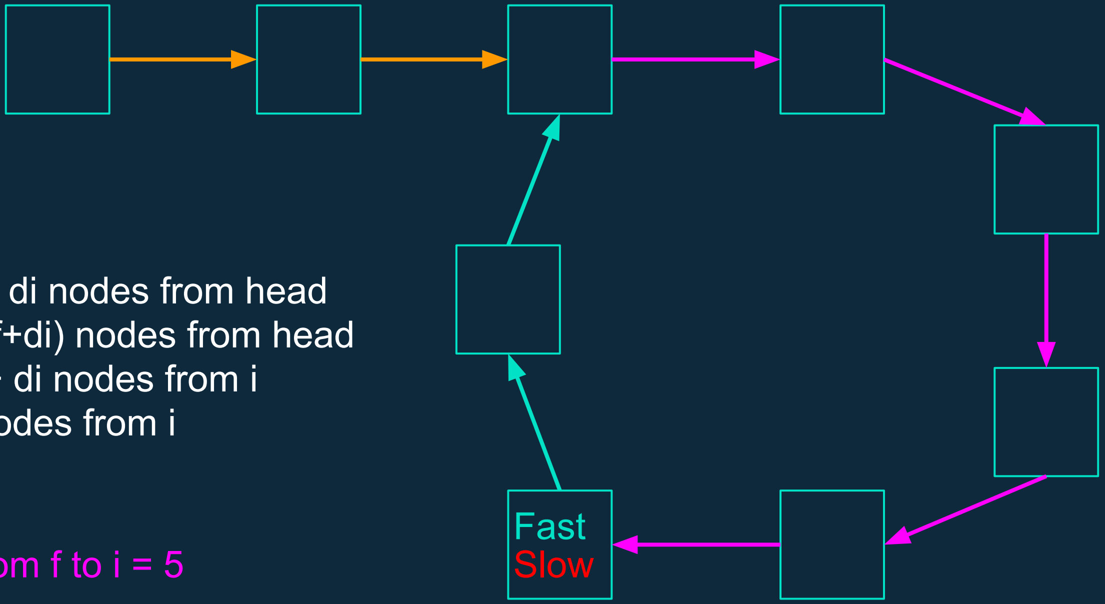
Part 2: Find the intersection node (More graphical approach)



Part 2: Find the intersection node (More graphical approach)


First loop node (f)
Df = distance from head to f = 2
Fast
Slow
Di = distance from f to i = 5
Intersection node (i)

Part 2: Find the intersection node (More graphical approach)



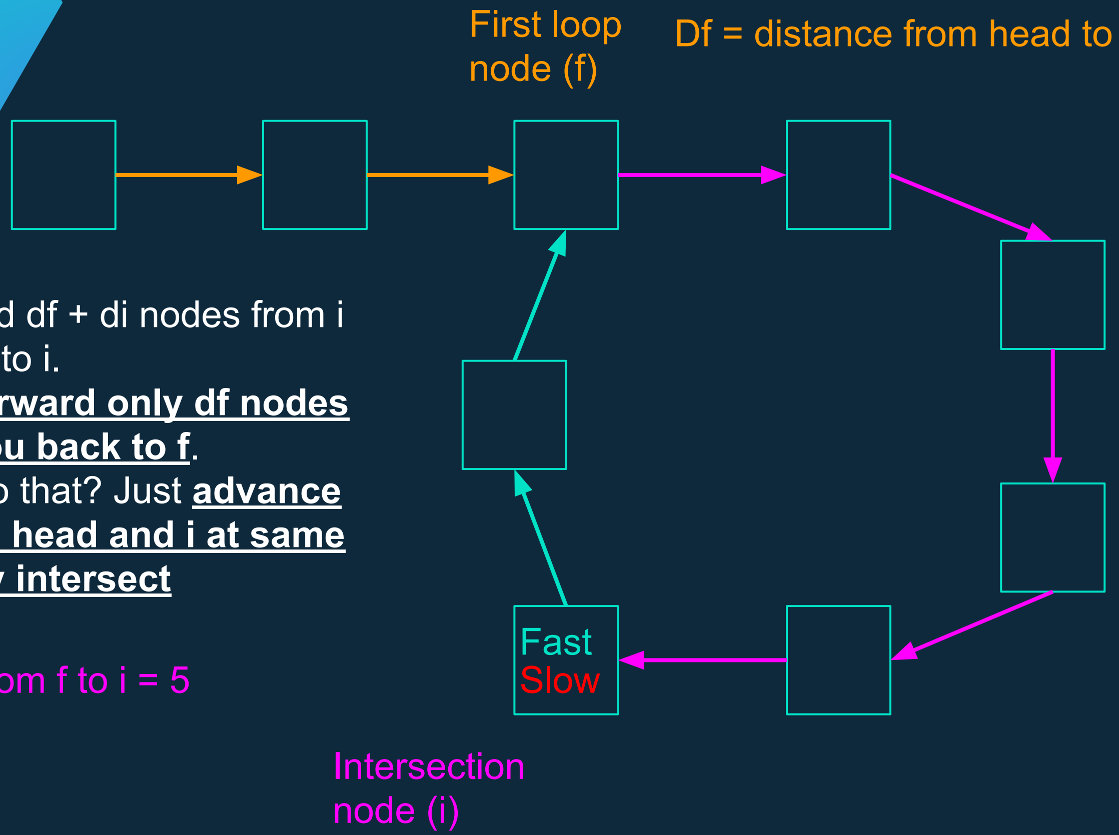
Df = distance from head to f = 2

Slow = df + di nodes from head
Fast = 2(df+di) nodes from head
          = df + di nodes from i
          = 0 nodes from i

Di = distance from f to i = 5

Part 2: Find the intersection node
(More graphical approach)

Df = distance from head to f = 2



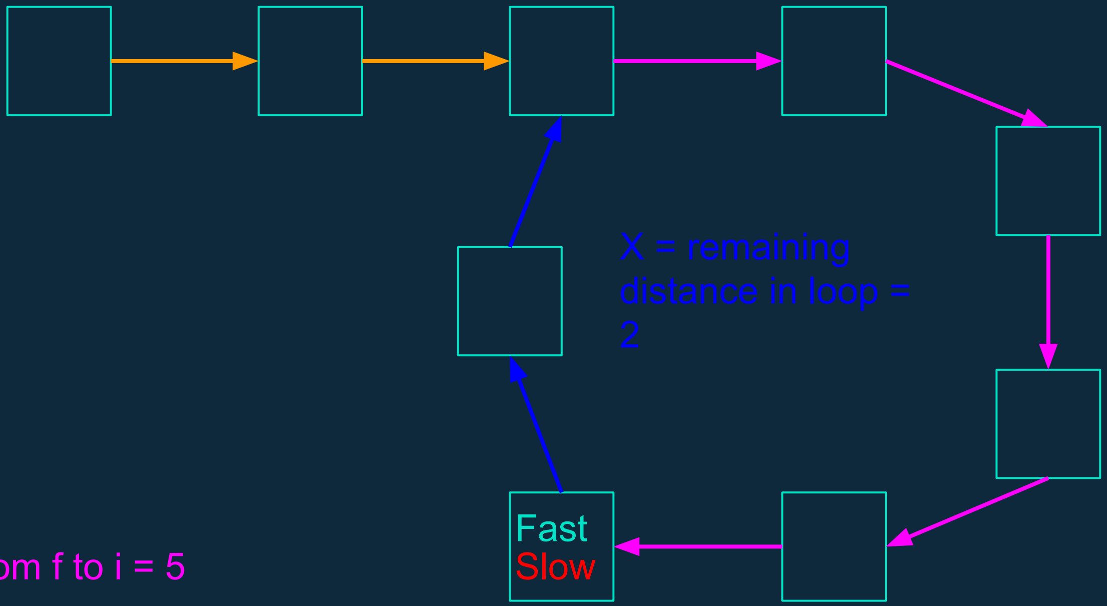
Moving forward df + di nodes from i gets you back to i.
So **<u>moving forward only df nodes should get you back to f</u>**.
How do you do that? Just **<u>advance pointers from head and i at same rate until they intersect</u>**

Di = distance from f to i = 5

Part 2: Find the intersection node (More mathy approach)

First loop node (f)

Df = distance from head to f = 2



Fast

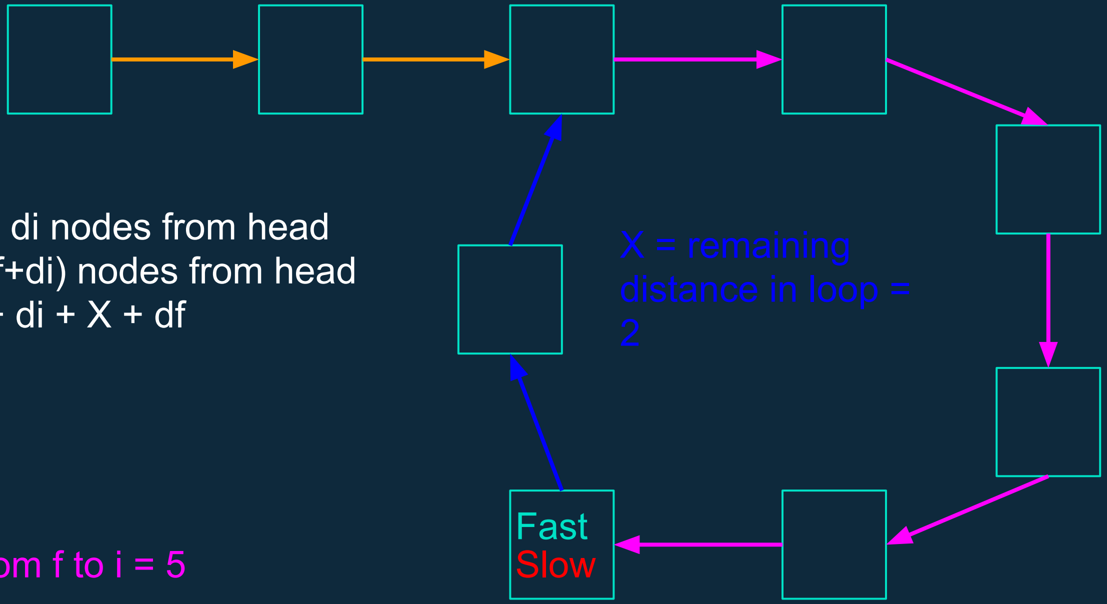
Slow

X = remaining distance in loop = 2

Di = distance from f to i = 5

Intersection node (i)

Part 2: Find the intersection node (More mathy approach)



First loop node (f)

Df = distance from head to f = 2

X = remaining distance in loop = 2

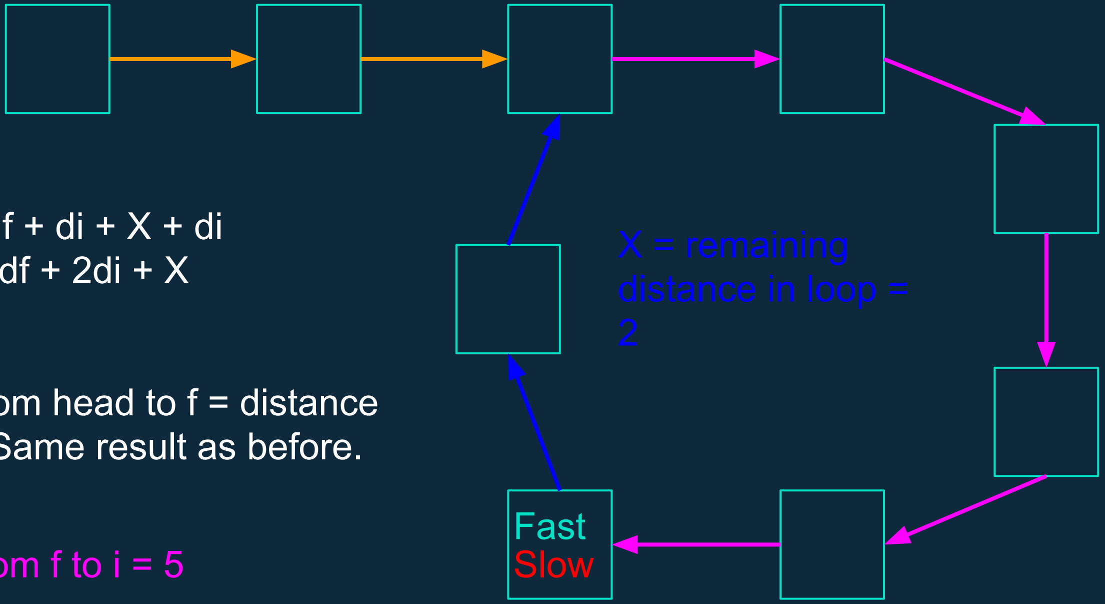
Fast
Slow

Intersection node (i)

Slow = df + di nodes from head
Fast = 2(df+di) nodes from head
= df + di + X + df

Di = distance from f to i = 5

Part 2: Find the intersection node (More mathy approach)



First loop node (f)

Df = distance from head to f = 2

X = remaining distance in loop = 2

Intersection node (i)

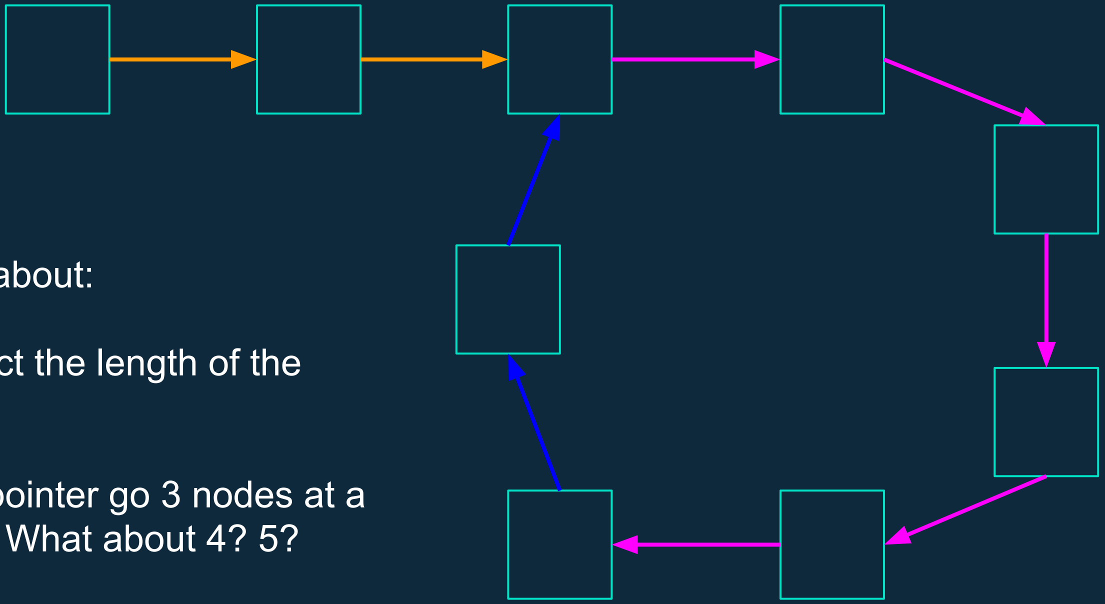
2(df+di) = df + di + X + di

2df + 2di = df + 2di + X

df = x

Distance from head to f = distance from i to f. Same result as before.

Di = distance from f to i = 5



Stuff to think about:

Can you detect the length of the cycle?

Can the fast pointer go 3 nodes at a time instead? What about 4? 5?

Interviewing Tactics II

Draw a Diagram

A lot of questions are very, very graphical. Don't be afraid to draw out a diagram and use multiple colors as necessary.

Play to your Advantages

The last problem had both a more graphically intuitive solution and a mathier one. Get to know which parts of your own reasoning ability are stronger, and capitalize on those strengths.

CATALYST

Add Two Numbers

Med

I give you two numbers, represented as SLL's. Return a new SLL representing their sum.

Ex: given a = 2->3->5, b = 9->2->7, return c = a+b = 1->1->6->2

Interviewing Tactics III

Edge Cases

Make sure your code works correctly in critical areas like the head or tail of an LL.

Input Preservation

As mentioned in the countLength() warm-up, try to not mutate the input whenever possible, unless the interviewer says it's okay.

Helper Functions

Sometimes you'll need a helper function which is actually decently complex. Expect to see more of these when we cover trees and graphs.

That was a lot to think about.

Let’s take a break and come back in 10 minutes.

2 Stacks

Concept 1: Stacks are First In, Last Out (FILO)

Just as with a stack of papers, whatever was last put on the pile will be the first one available on the top. Whatever you put in first, will come out last.

This is VERY useful for tasks where you need to keep a REVERSE order of elements on hand.

Concept 2: Stacks have push() and pop() methods

push(val) pushes the value into the stack, and pop() removes and returns the value at the top of the stack. Both methods run in O(1) time.

Stacks can also have more functions such as just peek(), but push() and pop() are fundamental.

Concept 3: Stacks are an abstract data type

Abstract data types (ADTs) can be implemented in several ways (think of them as interfaces/abstract classes)

Concrete data structures are implementations of abstract data types (think of them as java concrete classes)

Wait, you’re saying that SLL’s, Arrays, and Stacks are all related?

Intro

Implement Stack

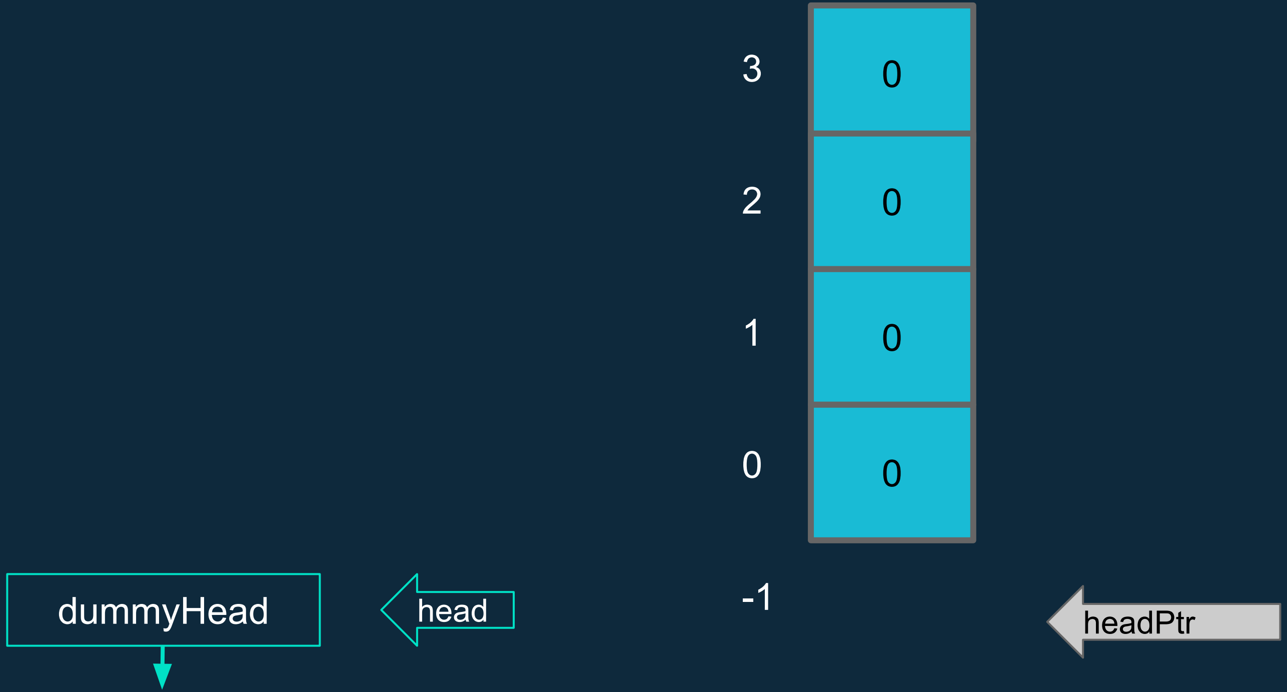
Implement a stack using a SLL. Then, implement a stack using an array.

Follow-up: implement two stacks using the same array

Stack using SLL or Array

Code

```
Stack s = new
Stack<Integer>();
```



3	0
2	0
1	0
0	0
-1	

dummyHead

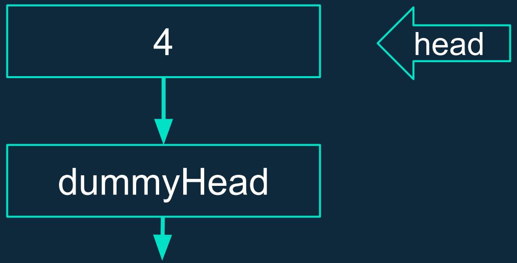
head

headPtr

Stack using SLL or Array

Code

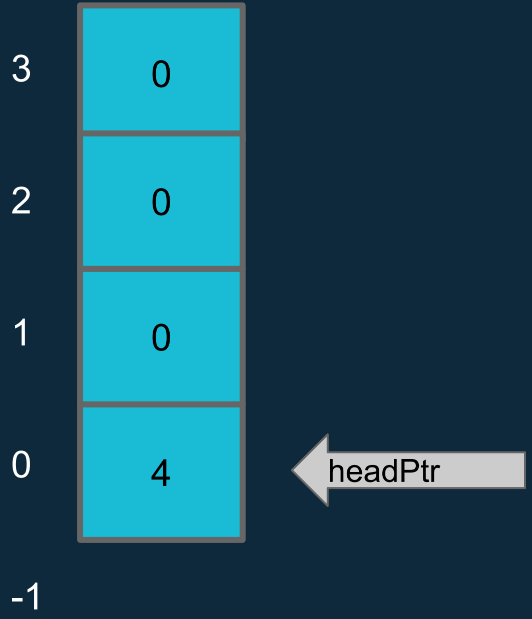
```
Stack s = new
Stack<Integer>();
s.push(4);
```



4

head

dummyHead



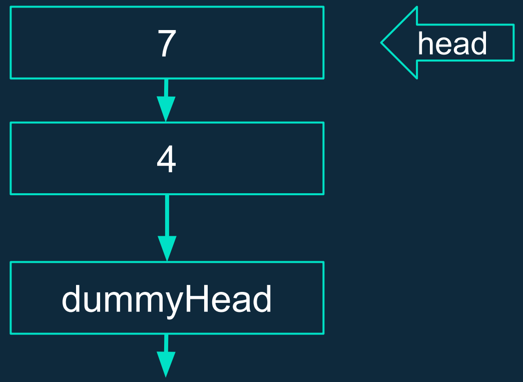
3	0
2	0
1	0
0	4
-1	

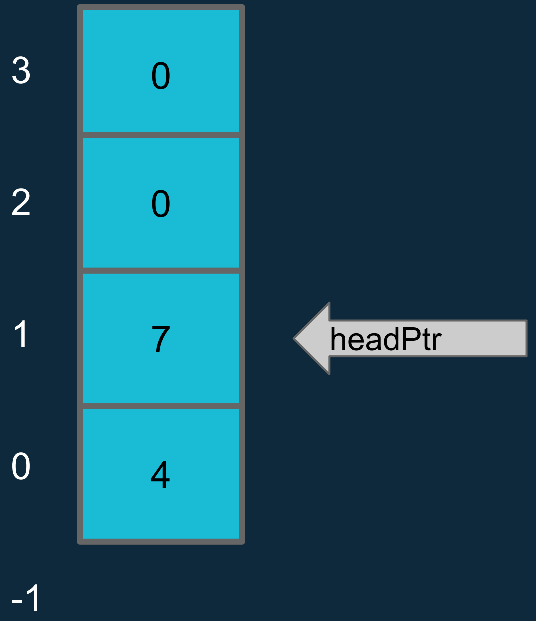
headPtr

Stack using SLL or Array

Code

```
Stack s = new
Stack<Integer>();
s.push(4);
s.push(7);
```

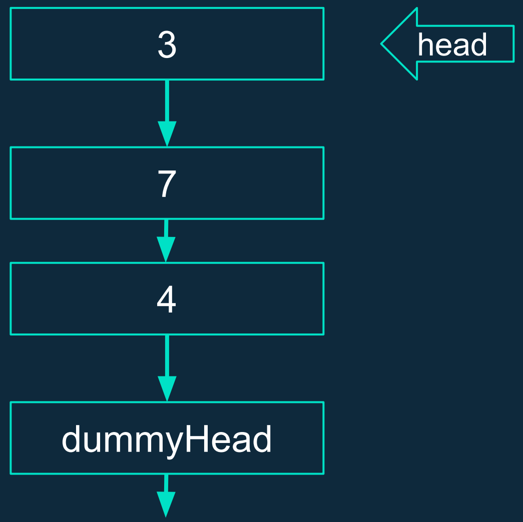


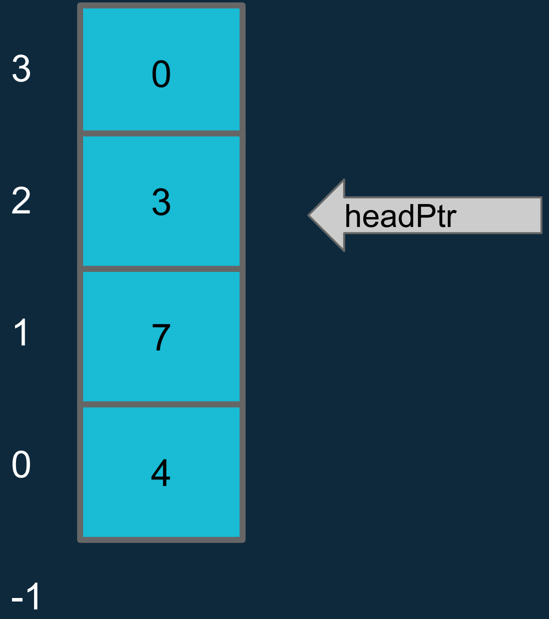


Stack using SLL or Array

Code

```
Stack s = new
Stack<Integer>();
s.push(4);
s.push(7);
s.push(3);
```

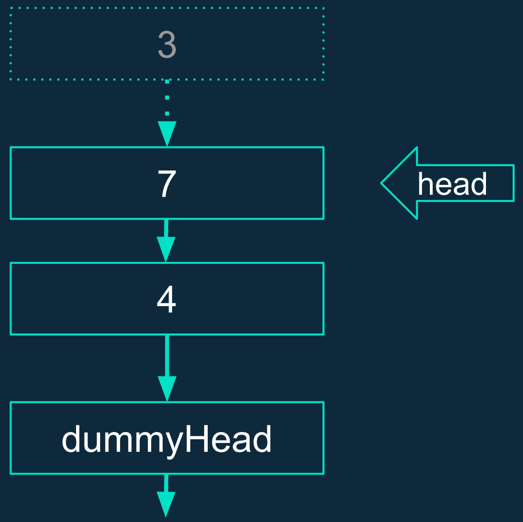


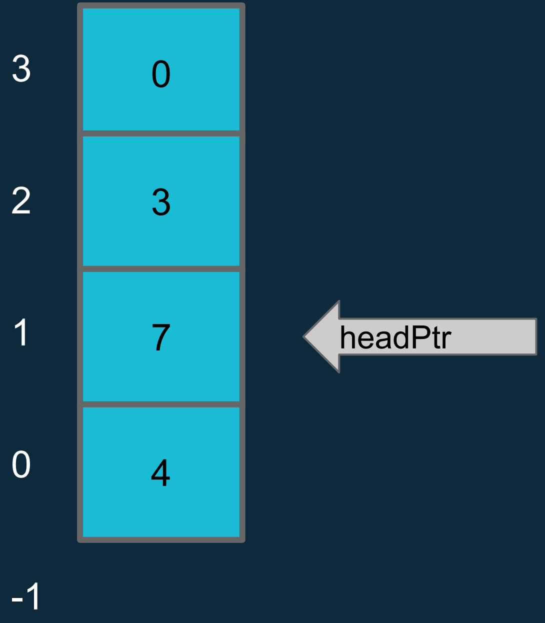


Stack using SLL or Array

Code

```
Stack s = new
Stack<Integer>();
s.push(4);
s.push(7);
s.push(3);
s.pop();
```



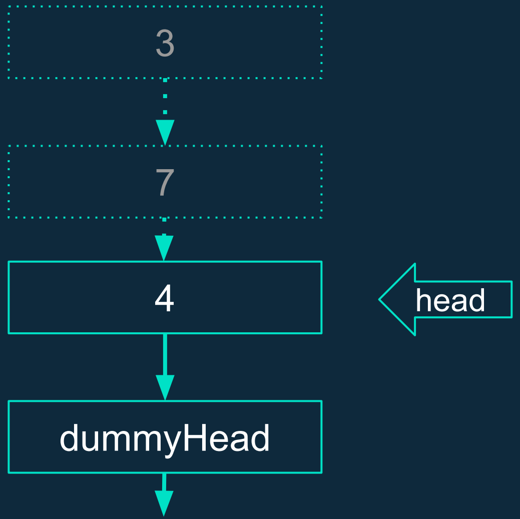


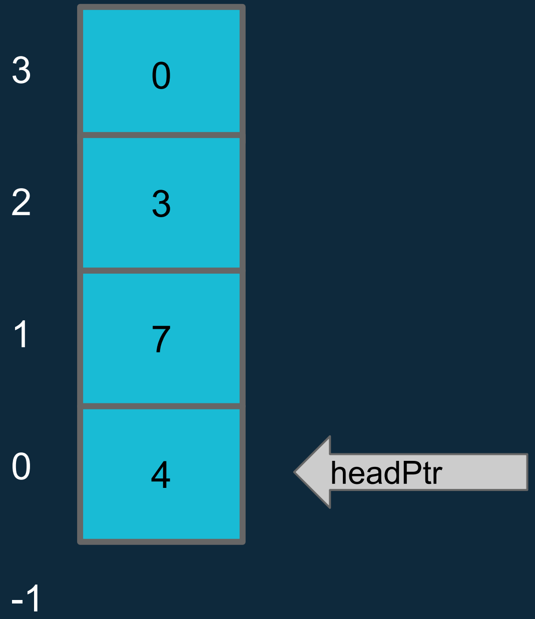
CATALYST

Stack using SLL or Array

Code

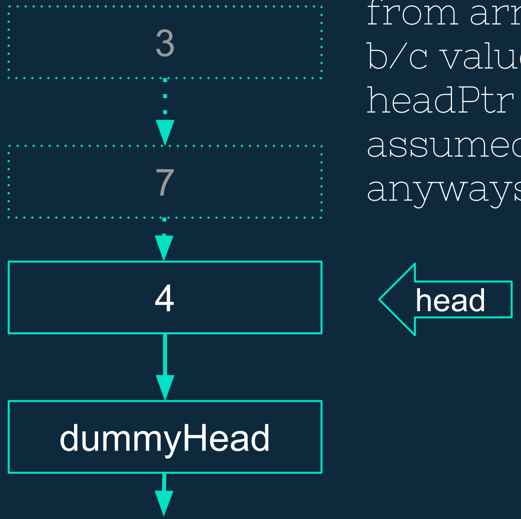
```
Stack s = new
Stack<Integer>();
s.push(4);
s.push(7);
s.push(3);
s.pop();
s.pop();
```



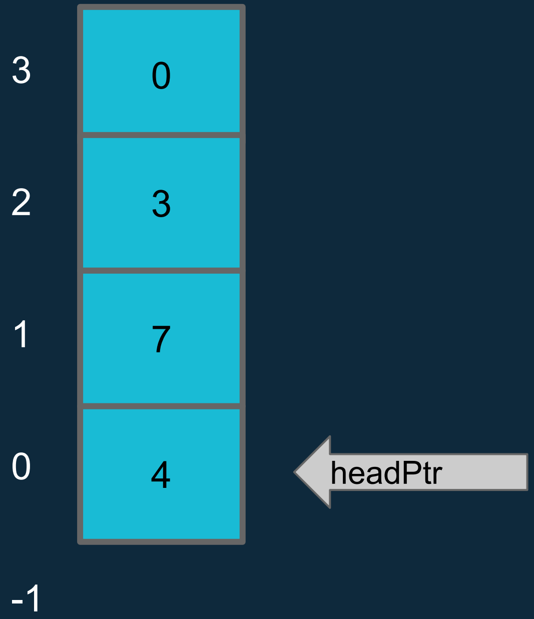


Stack using SLL or Array

NB: in languages like Java/Python, the 3 and 7 nodes are automatically removed from memory during garbage collection since they're not "useful" anymore as they're not referenced.



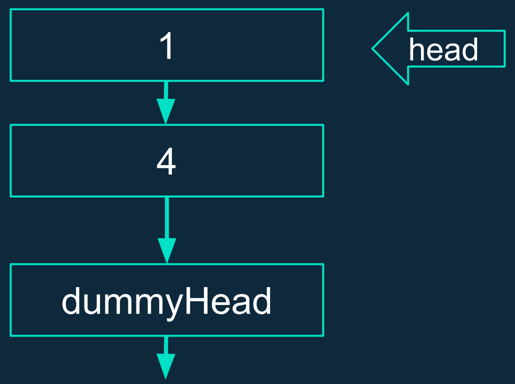
Also, no need to reset values to 0 when popping from array impl b/c values above headPtr are assumed "garbage" anyways

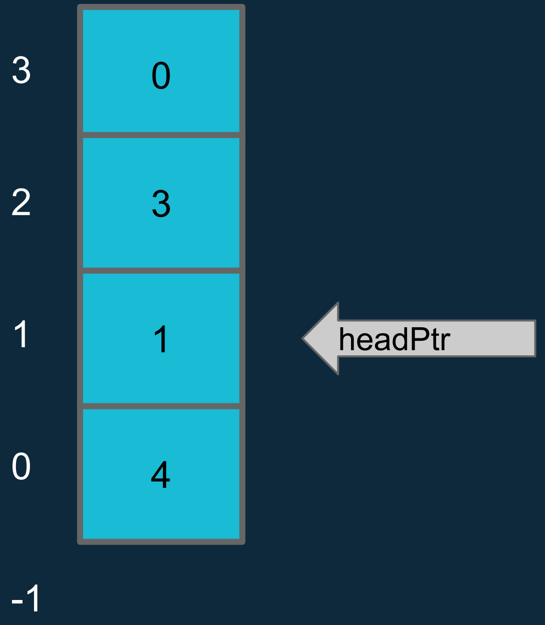


Stack using SLL or Array

Code

```
Stack s = new
Stack<Integer>();
s.push(4);
s.push(7);
s.push(3);
s.pop();
s.pop();
s.push(1);
```





CATALYST

Stack using SLL or Array

	SLL	Array

new Stack()	`dummyHead = new Node(0);`
	`head = dummyHead;`
		`int[] container = new int[SIZE];`
		`headPtr = -1`

void push(int val)	`Node newest = new Node(val);`
	`newest.next = head;`
	`head = newest;`
		`headPtr++;`
		`container[headPtr] = val;`

int pop()	`int val = head.data;`
	`head = head.next;`
	`return val;`
		`int val = container[headPtr];`
		`headPtr--;`
		`return val;`

boolean isEmpty()	`return head == dummyHead;`	`return headPtr == -1;`

CATALYST

Bracket Balancing I

Intro

It’s always infuriating resolving compile errors. Bracket/parentheses mismatches are often the cause.

I give you an ASCII string s that contains many characters, but you’re only interested in the characters ‘(‘, ‘)’, ‘[‘, ‘]’, ‘{‘, and ‘}’. You need to make sure that these brackets/parentheses are in a compileable order. This means that every bracket which is opened is closed at one point, and that all the brackets are closed in the correct order. For example, “{catalyst}”, “()[]{}” and “{[()bobby]}” should return true, but “[)” and “([)]” should return false.

Intro

Bracket Balancing II

Follow-up: I only care that the brackets ‘{‘ and ‘}’ are balanced now - all other characters can be ignored. Can you optimize your solution even further?

Hint: is a stack still necessary?

CATALYST

Med

Max Stack

Design a stack of integers that functions just as a normal stack would, but it also has a getMax() function which will correctly return the largest element in the stack in O(1) time.

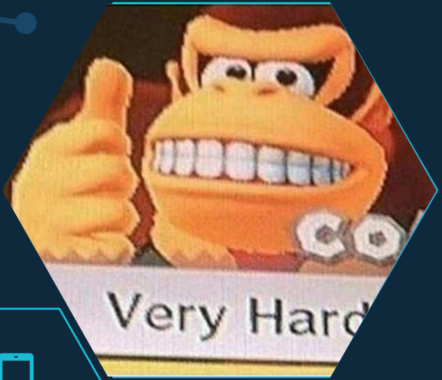
You should implement the constructor, isEmpty(), pop(), push(), and getMax() functions.

That was a lot to think about.

Let’s stop here and finish next week.

CATALYST

Prefix Calculator



Prefix notation is a math notation where the operators (+,-,x,/) are given before the operands instead of between them. For example, instead of 5+3 we'd have (+ 5 3). And instead of (6 - 2) * 9 we'd have (* (- 6 2) 9).

I'll give you a prefix notation math expression string containing integers (positive or negative), parentheses, and the +,-,x,/ functions. Assume that all syntax is valid, that all operations have two operands, and that all expressions are fully parenthesized. Output the integer result of that expression, truncating integer division down and not worrying about overflow.

Advanced “Calculator”

Great job! Now here’s the hard part(s). Can you do the following?

- Generalize such that operations can have more than 2 operands. For example, (+ 2 3 4) would give 9.
- Add variable binding (and then variable scoping)
- Add conditional statements
- Add function declaration
- Add support for strings and other data types
- Add evaluating multiple expressions in succession

If so, congrats - you’ve just built a Lisp interpreter.
https://cs61a.org/proj/scheme/

Note: this is actually very hard to do in a 1 hour interview.

3

Queues

Concept 1: Queues are First In, First Out (FIFO) Abstract Data Types

Think of a queue as a checkout line. Assuming no one cuts, whomever is in line earlier will check out earlier.

This is VERY useful for tasks where you need to keep a record of the order of elements

Concept 2: Queues have add() and remove() methods

add(val) adds the value to the queue. remove() removes and returns the earliest added value in the queue. Both methods run in O(1) time.

Queues can also have more functions such as just peek(), but add() and remove() are fundamental.

> Wait, you’re saying that SLL’s, Arrays, Stacks, and Queues are all related?

Stack using Queue

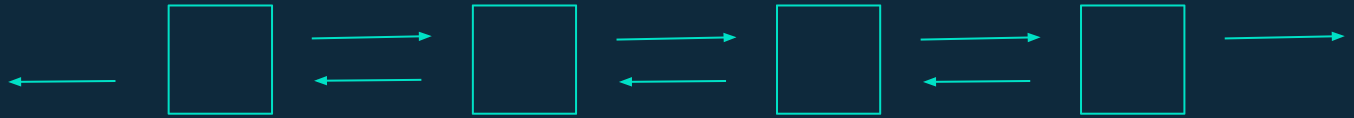
Med

Implement a stack using queues. What are the runtimes of push() and pop() with this implementation?

3

Doubly Linked Lists

Concept: Doubly Linked Lists have a PREV and NEXT pointer



DLL Assumptions

Validity

Say that we have 2 DLL nodes, a and b, and that a.next = b. It's technically possible for b.prev != a, but for all interview purposes, you should assume that if a.next = b, then b.prev = a.

Access

Since you can traverse a list both forwards and backwards with a DLL, it's possible to access the whole list with any arbitrary node.

However, usually we usually have pointers to the head and tail pointers.

Wait, you’re saying that EVERYTHING’s all related?

Implement Queue

Med

Implement a queue using a doubly linked list, and an array. Pay close attention to index out of bounds errors for the array.

Bonus: implement the queue using a circular DLL.

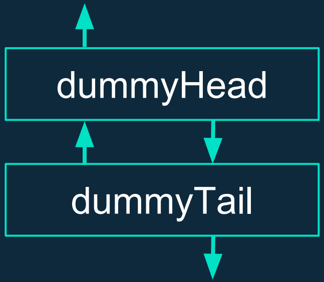
Queue using DLL or Array

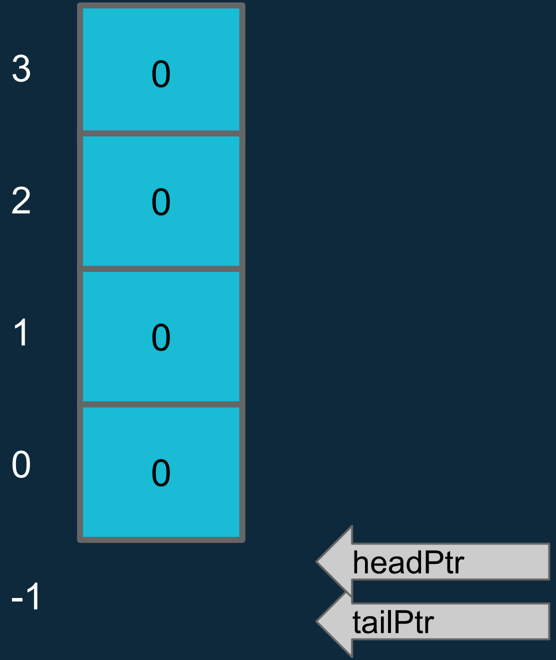
Code

Queue q = new Queue<Integer>();

DLL:

Arrow on left side is prev
Arrow on right side is next

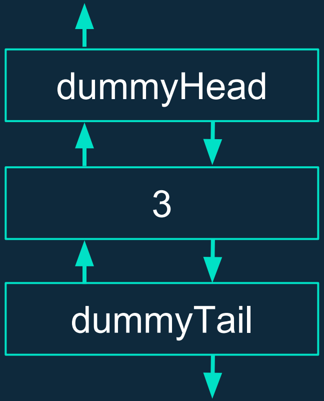


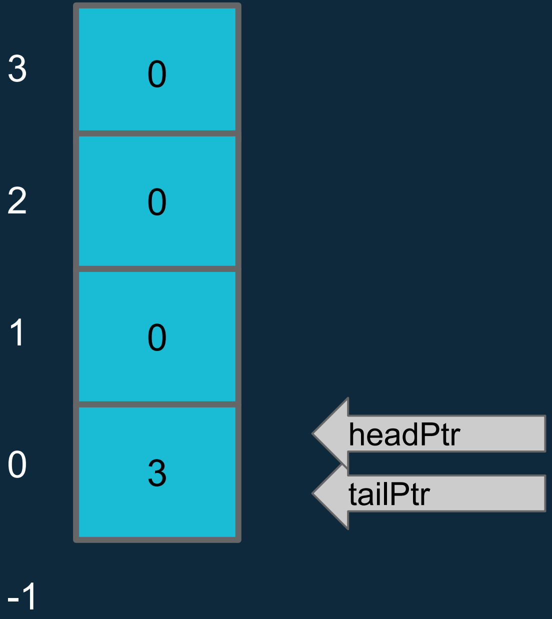


Queue using DLL or Array

Code

Queue q = new Queue<Integer>();
q.add(3);

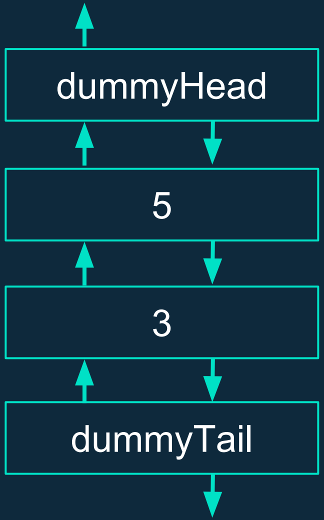


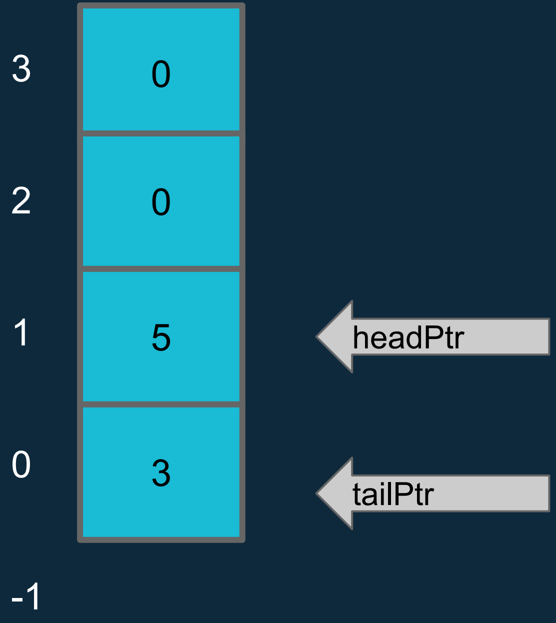


Queue using DLL or Array

Code

```
Queue q = new
Queue<Integer>();
q.add(3);
q.add(5);
```

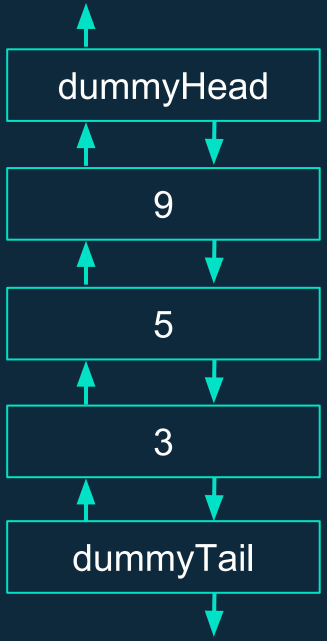


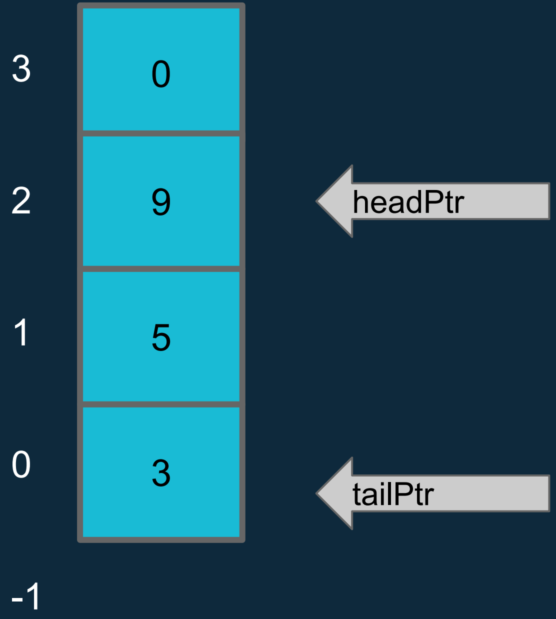


Queue using DLL or Array

Code

```
Queue q = new
Queue<Integer>();
q.add(3);
q.add(5);
q.add(9);
```

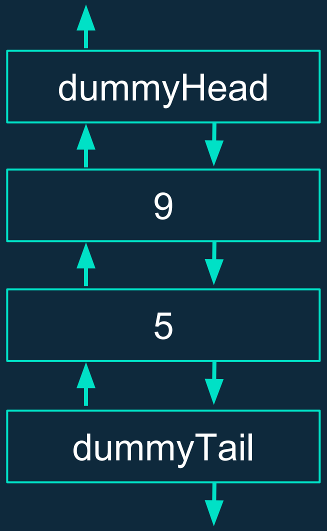


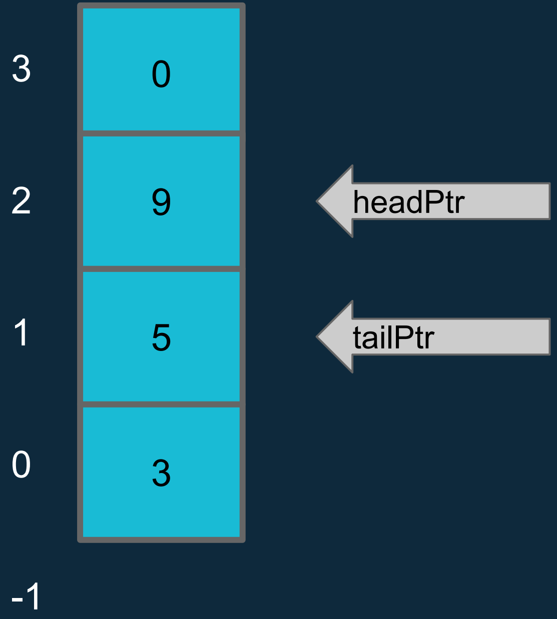


Queue using DLL or Array

Code

```
Queue q = new
Queue<Integer>();
q.add(3);
q.add(5);
q.add(9);
q.remove();
```

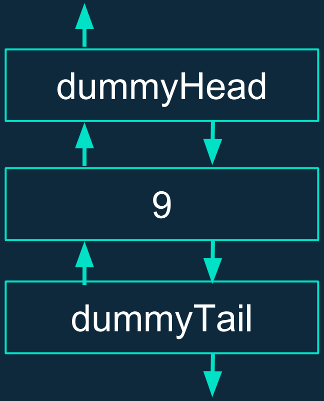


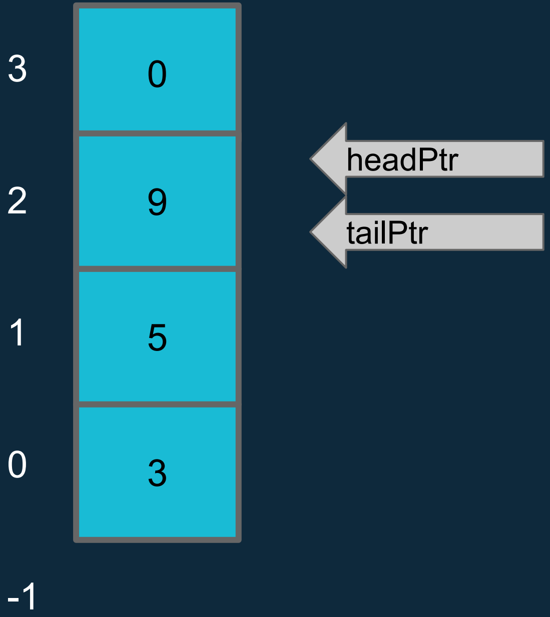


Queue using DLL or Array

Code

```
Queue q = new
Queue<Integer>();
q.add(3);
q.add(5);
q.add(9);
q.remove();
q.remove();
```

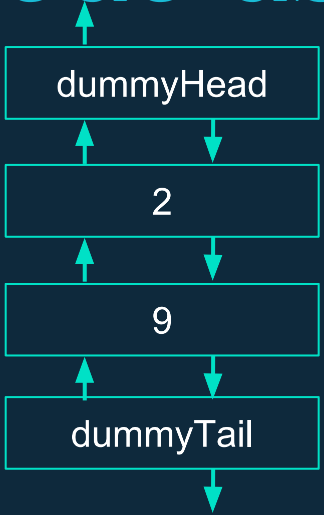


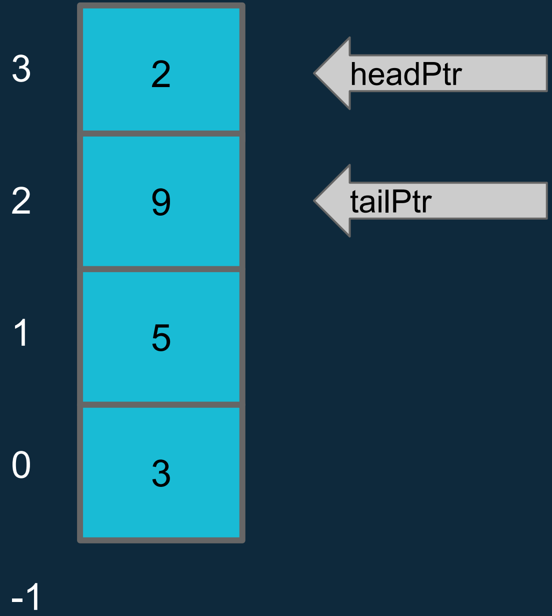


Queue using DLL or Array

Code

```
Queue q = new
Queue<Integer>();
q.add(3);
q.add(5);
q.add(9);
q.remove();
q.remove();
q.add(2);
```





CATALYST

Hard

Browser Cache

Caches allow for fast lookup of commonly accessed data. Think of them as a key => value store, much like a hashmap. (hint hint)

For example, your web browser has a cache of stored web pages that you've recently visited. Caches are finite in size, though - this cache removes the least recently used/visited page to make room for a new page when it runs out of storage, and thus is called a Least Recently Used (LRU) cache. Also, if a page is re-visited while it's still in the cache, it's set as the most recently visited page - no duplicates should exist in the cache. For more info check out the link below:

https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU

Browser Cache

Hard

Your job is to create a LRUBrowserCache that implements this functionality.

Your LRUBrowserCache API should include a constructor that takes in an int as the cache maximum size and a `String getHtml(String url)` that returns the webpage (as an HTML string) at a certain URL. You should optimize for time complexity. You have a `String findOnline(String url)` function that actually goes online and returns the webpage code (as a String), but you should minimize calls to this though since going online is MUCH slower than loading a local copy.

A Note about Elegance

You may notice that I have dummy variables a lot in my code. Are they necessary? Not necessarily. Are they elegant? Not really.

However, in many cases they can help you write cleaner methods that don’t need special if (someVar == null) checking, especially since the first and last nodes of linked lists tend to be edge case hotspots.

It’s definitely possible to write clean, elegant code without the need for dummy variables. But this is an *intro to interviewing* class, and these dummy variables can aid in understanding when you’re starting out.

Thanks!

(M)Any questions?

You can find us at: {
bobby.wang,
geng.sng,
ashka.stephen,
simran.singh
}@duke.edu