Singly Linked List

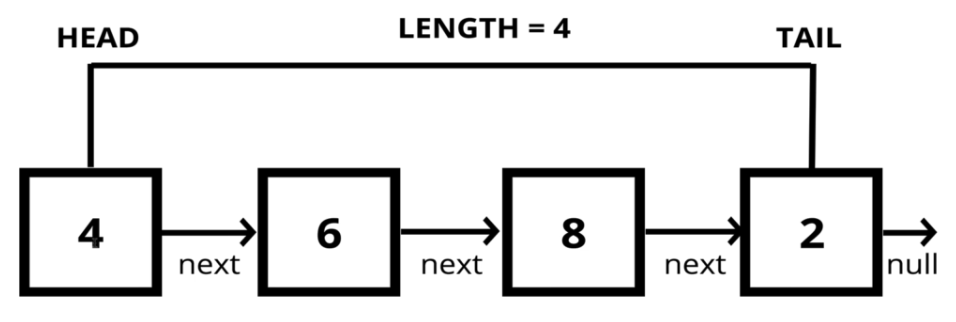
Bunch of nodes pointing to other nodes. One node points to the next one,  
sort of lie a chain.

Linked lists consists of nodes, and each node has a value and a pointer to another node or null.

Each element is a node.

A data strcture that contains a head, tail and length property.

We don’t keep track of every single item in the middle. We just keep track of the head and from the head we can figure out the next one and from that the next one and so on until the end.



There exist no index.

If we want to access something from this list, we’ll start at the beginning and will ask for the next item and from there the next one until we found the item we were looking for.

In singly lined lists, each node is only connected to one direction, to the next node.

In contrast, doubly lined list also has a connection pointing back to the previous one.

Think of it as a fortress square mall with no elevators. You can’t directly go to the 3rd floor. To get to the third floor, we’ll start with the first one, then take the stairs to the second then you’ll go to the third one.

To insert element at the beginning, all we have to do is to make that node as the new head and have it point to the old head.

Unlike an array where every single item would have to be re-indexed. It has this *cascade ripple effect*.

|  |  |
| --- | --- |
| Lists | Arrays |
| Do not have indexex.  Connected via nodes with a next pointer.  Random Access is not allowed. | Indexed in order.  Insertion and deletion can be expensive.  Can quickly be accessed at a specific index. |

You’d want to use a lined list if you have really care about insertion and deletion especially if you’re woring with a long data set and you don’t need random access, you just need to store it in a list.

A node is very simple. It just stores:

* value pieve of data
* next reference to next node

**Node**

* val
* next

**Doubly Linked List**

* head
* tail
* length

class Node {

    constructor(val){

        this.val = val;

        this.next = null;

    }

}

Because in the beginning, nothing comes after it.

let first = new Node("Hi")

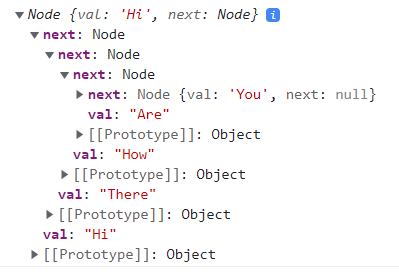
first.next = new Node("There")

first.next.next = new Node("How")

first.next.next.next = new Node("Are")

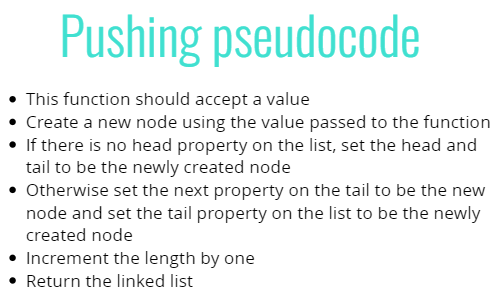
first.next.next.next.next = new Node("You")

console.log(first);



Push Method

Pusing to the end of the list is super easy.If we have a thousand or millions of items in a list, we don’t traverse the whole thing, as long as we’re keeping track of the last item in the list.



    push(val) {

        let node = new Node(val);

        if(this.length == 0){

            this.head = node;

        }else{

            this.tail.next = node;

        }

        this.tail = node;

        this.length++;

        return this;

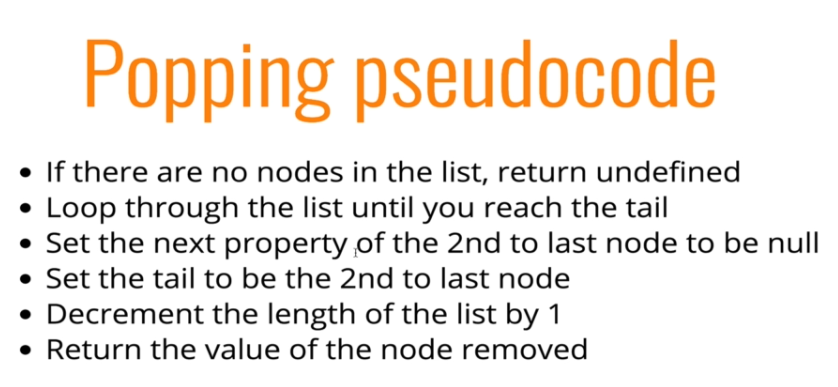
    }

Pop method

Removing the node from the end of the Linked list.

Pop seems simple as we’re keeping track of the last item. We should just be able to return it.

Problem is before we have to remove it, we have to assign a new tail. And that involves going all the way through the lists from the beginning because we don’t have a backward pointer.



  pop() {

    if (this.length == 0)

      return undefined;

    let x = this.head;

    let y = this.head.next;

    while (y.next) {

      x = x.next;

      y = y.next;

    }

    this.tail = x;

    this.tail.next = null;

    this.length--;

    if (this.head == this.tail) {

      this.head = null;

      this.tail = null;

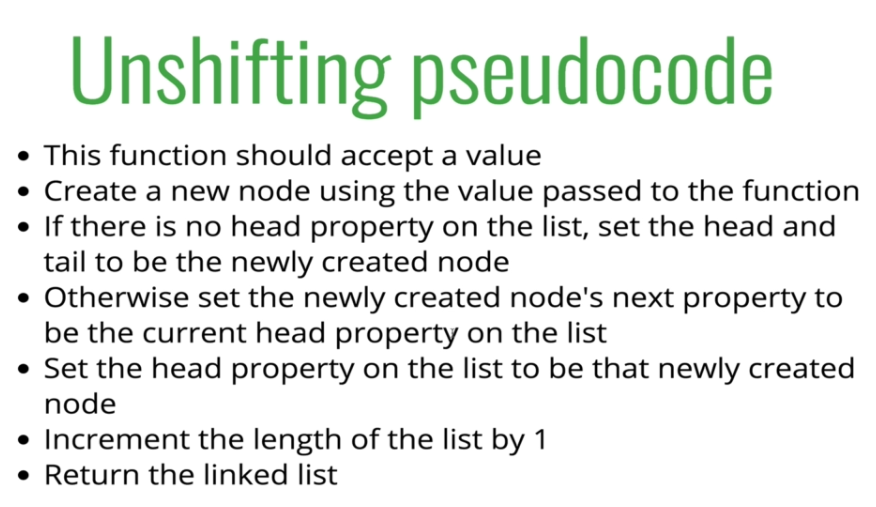
    }

    return y;

  }

UnShift method

Adding a new node to the beginning of the Linked List.



  unshift(val) {

    let newNode = new Node(val);

    if (this.length == 0) {

      this.head = newNode;

      this.tail = newNode;

    } else {

        newNode.next = this.head;

        this.head = newNode;

    }

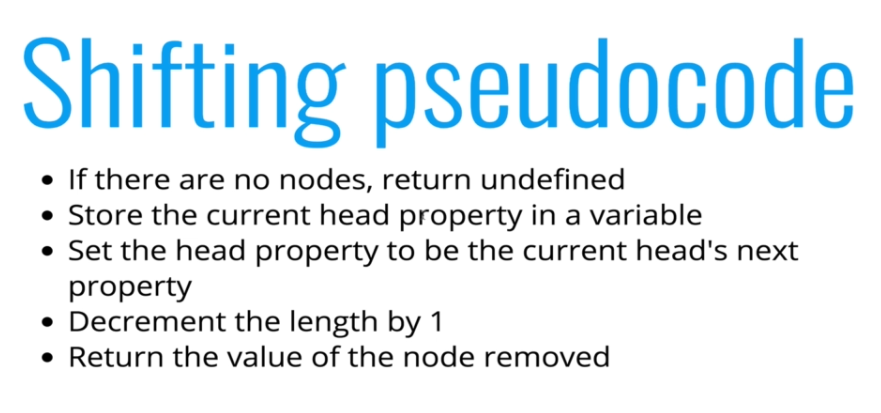
    this.length++;

    return this;

  }

Shift method

Removing the node from the beginning of the Linked list.



  shift() {

    if (this.length == 0) {

      return undefined;

    }

    let prevHead = this.head;

    this.head = prevHead.next;

    this.length--;

    if (this.length == 0) {

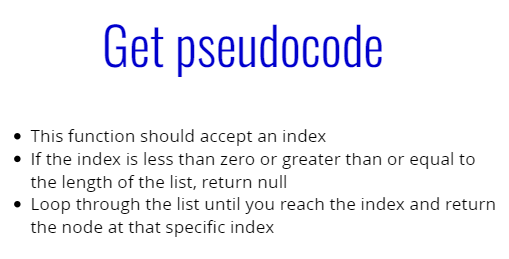
      this.tail = null;

    }

    return prevHead;

  }

GET  
  
 Retreiving a node by its position in the Linked list  
Takes a number then traverse list that many times  
There exist  no indices that correspond to each item.



  get(i) {

    if (i < 0 || i >= this.length) {

      return null;

    }

    let counter = 0;

    let currentNode = this.head;

    while (counter < i) {

      currentNode = currentNode.next;

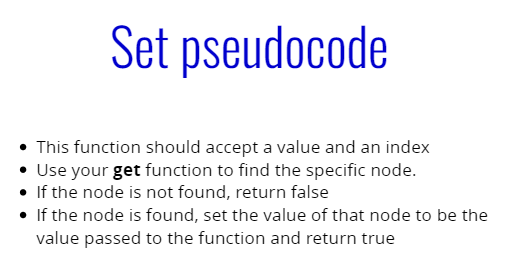
      counter++;

    }

    return currentNode;

  }

SET  
  
 Changing the value of a node based on its position.



  set(i, value) {

    let foundNode = this.get(i);

    if (foundNode) {

      foundNode.val = value;

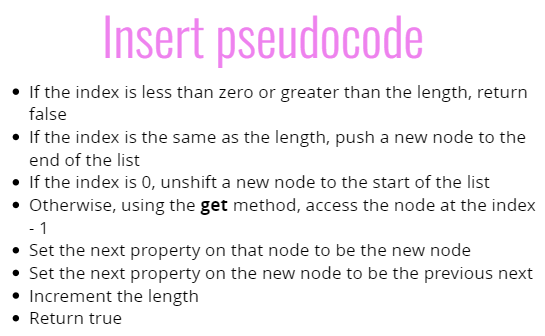
      return true;

    }

    return false;

  }

INSERT  
  
 Adding a node to the Linked List at a specific position.  
Sort of like set method but instead of updating an existing node it is going to insert a new node.  
  
Ideally your method should return only true or false.  
Problem is if we're gonna rely on our push and unshift method, these don't return true or false.  
So figure out a way.



  insert(i, value) {

    if (i < 0 || i > this.length) return false;

    if (i === this.length) {

      this.push(value);

      return true;

    }

    if (i === 0) {

      this.unshift(value);

      return true;

    }

    let newNode = new Node(value);

    let prevNode = this.get(i - 1);

    let nextNode = prevNode.next;

    prevNode.next = newNode;

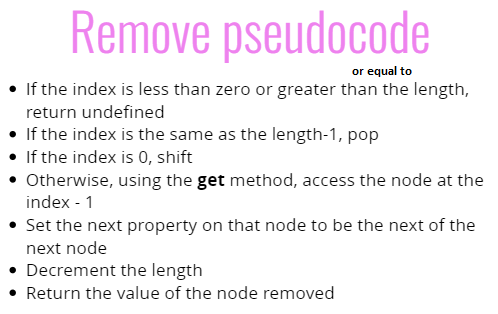
    newNode.next = nextNode;

    this.length++;

    return true;

  }

REMOVE  
  
 Removing a node from the linked list at a specific position.



  remove(i) {

    if (i < 0 || i >= this.length) return false;

    if (i == this.length - 1) {

      return this.pop();

    }

    if (i == 0) {

      return this.shift();

    }

    let prevNode = this.get(i - 1);

    let removedNode = prevNode.next;

    let nextNode = removedNode.next;

    prevNode.next = nextNode;

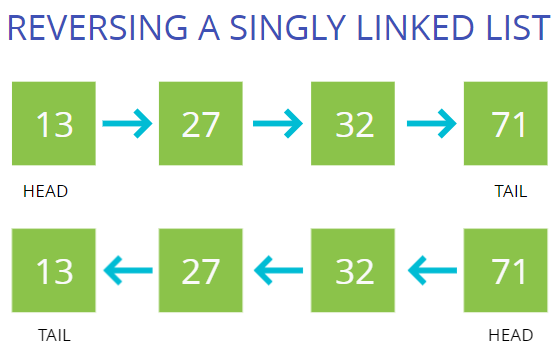
    this.length--;

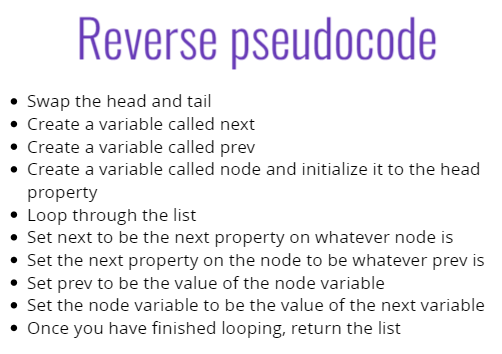
    return true;

  }

Comes in interview

REVERSE  
 Reversing the Linked list in place  
You traverse and reverse





  reverse() { A -> B -> C -> D

    let currentNode = this.head;

    this.head = this.tail;

    this.tail = currentNode;

    let prevNode = null;

    let nextNode;

    for (let i = 0; i < this.length; i++) { // while (currentNode)

      nextNode = currentNode.next;

      currentNode.next = prevNode;

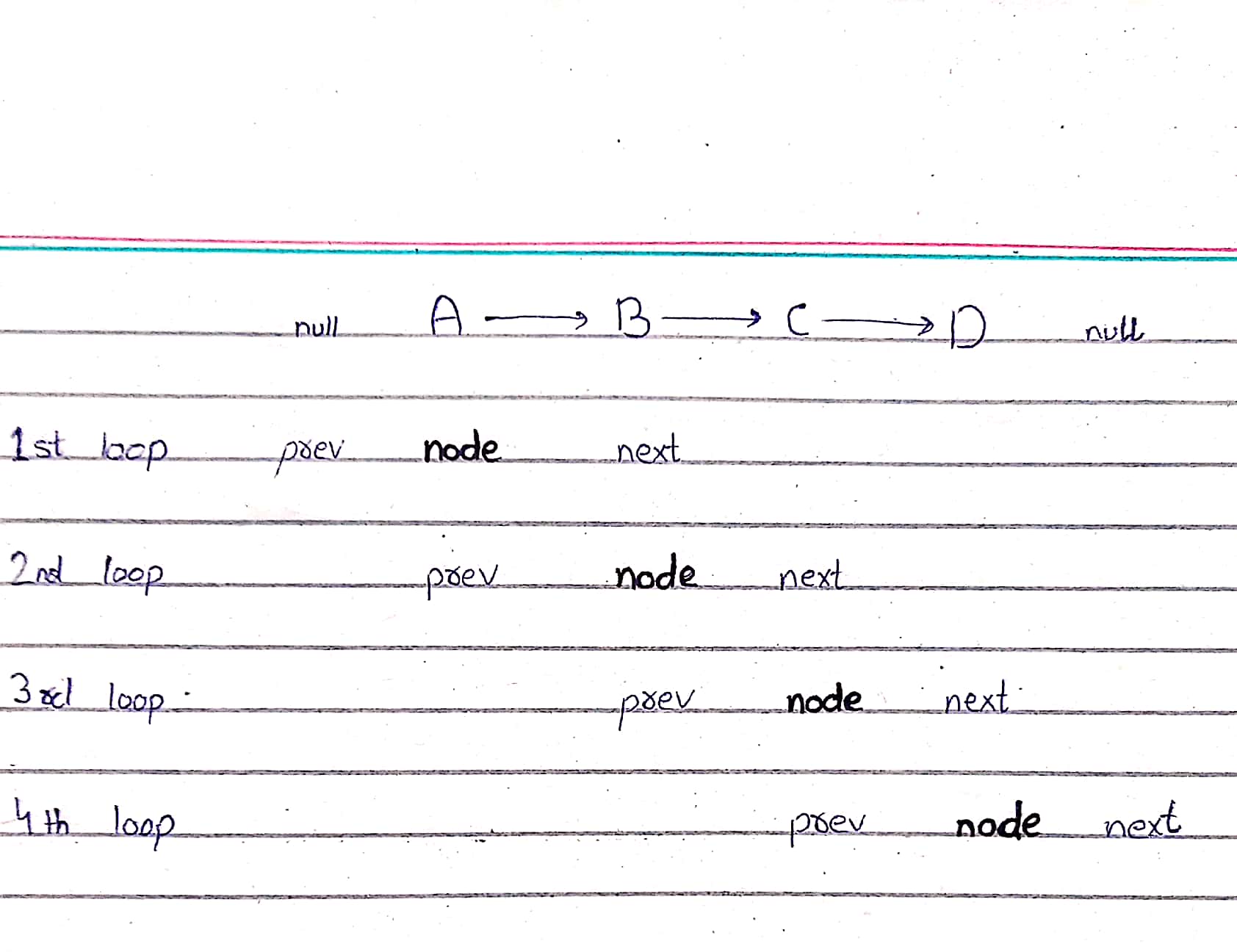
      prevNode = currentNode;

      currentNode = nextNode;

    }

    return this; D -> C -> B -> A

  }



Traversing the list

  traverse() {

    let myhead = this.head;

    while (myhead) {

      console.log(myhead.val);

      myhead = myhead.next;

    }

  }

Converting singly linked list to an array

Just for understanding, it’s not something we’d usually do.

  print(){

    let node = this.head;

    let arr = [];

    while(node){

      arr.push(node.val)

      node = node.next;

    }

    console.log(arr);

  }

Time Complexity

Insertion: O(1) Constant time in case of inserting at start or end.

That is not the case with arrays. In arrays, if we’re inserting at the end then it takes constant time.  
For inserting at start, its O(n).

So, Singly Linked list is good at inserting data.

Removal: O(1) or O(n) Depends on where we’re removing from.

If we’re removing from start, its O(1)  
 But it is difficult to remove at the end bcz we need to find the item right before the last item and that involves iterating the entire list.

In arrays, its O(1) for removing item at end and O(n) for removing at start or in-between bcz once we remove an item we have to reindex the whole array.

Searching: O(n) Looking for a specific value eg, if list contains 72

Accessing: O(n) Getting a certain node.  
 In arrays, its O(1) as arrays have random access.

Wrapping it up, Singly Lined Lists *excel at insertion(anywhere) and deletion(at start) compared to arrays*.

