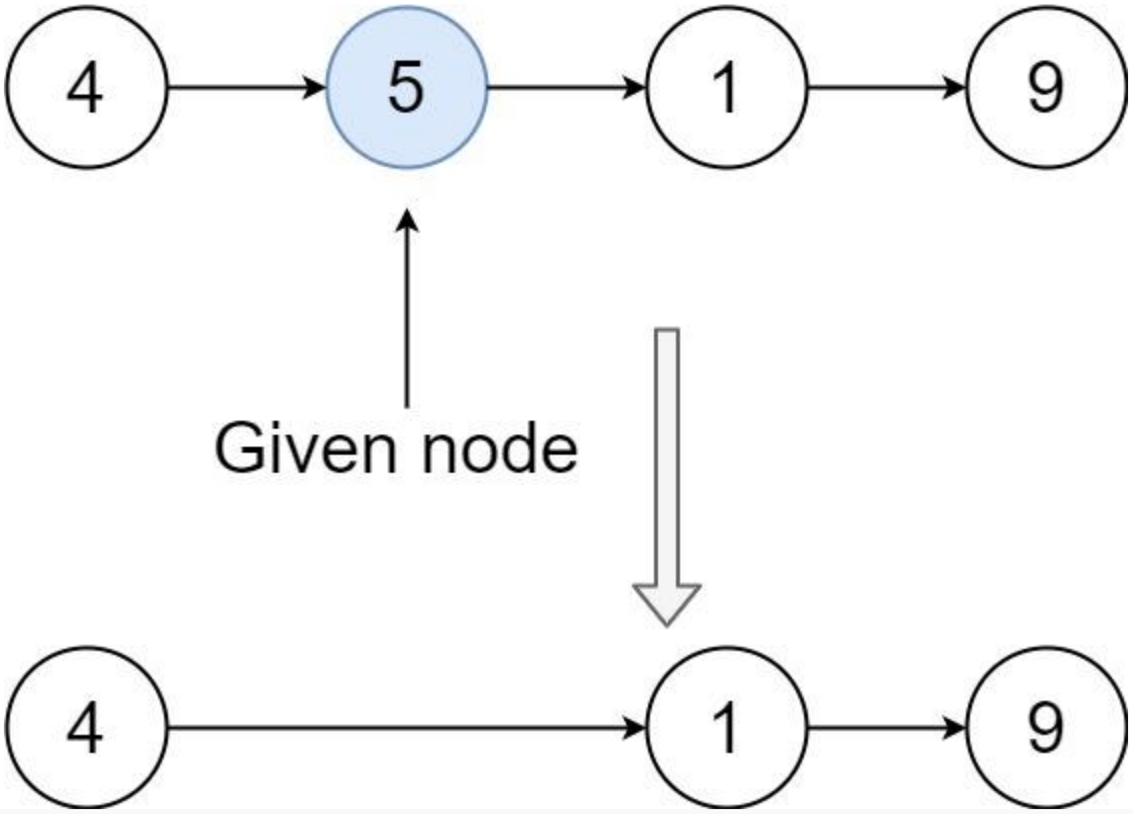
Assignment 4 (Linked List)

Total 70 points
Each question carries 10 marks

Q1. Write a function to **delete a node** in a singly-linked list. You will **not** be given access to the `head` of the list, instead you will be given access to **the node to be deleted** directly.

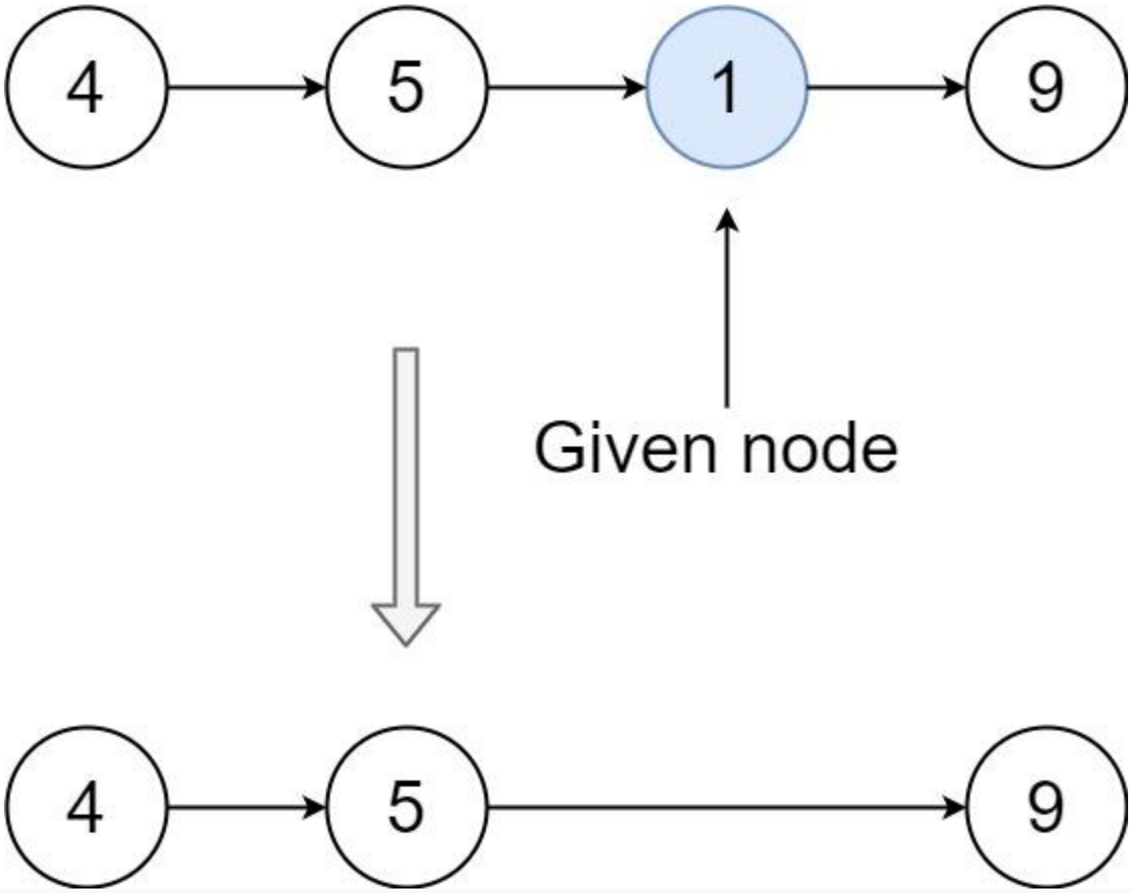
It is **guaranteed** that the node to be deleted is **not a tail node** in the list.

Example 1:



```
Input: head = [4,5,1,9], node = 5

Output: [4,1,9]

Explanation: You are given the second node with value 5, the linked list should
become 4 -> 1 -> 9 after calling your function.
```

Example 2:



```
Input: head = [4,5,1,9], node = 1

Output: [4,5,9]

Explanation: You are given the third node with value 1, the linked list should become
4 -> 5 -> 9 after calling your function.
```

Example 3:

```
Input: head = [1,2,3,4], node = 3

Output: [1,2,4]
```

Example 4:

```
Input: head = [0,1], node = 0

Output: [1]
```

Example 5:

```
Input: head = [-3,5,-99], node = -3

Output: [5,-99]
```

Constraints:

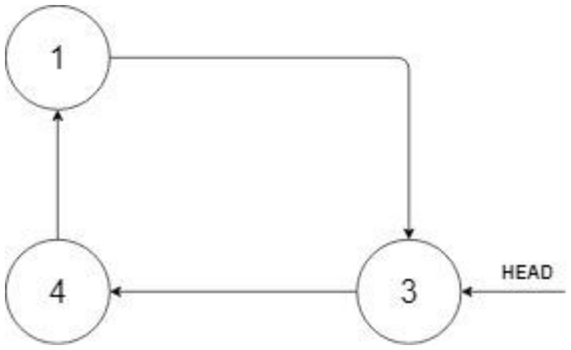
- The number of the nodes in the given list is in the range `[2, 1000]`.
- `-1000 <= Node.val <= 1000`
- The value of each node in the list is **unique**.
- The `node` to be deleted is **in the list** and is **not a tail** node

Q2. Given a Circular Linked List node, which is sorted in ascending order, write a function to insert a value `insertVal` into the list such that it remains a sorted circular list. The given node can be a reference to any single node in the list and may not necessarily be the smallest value in the circular list.

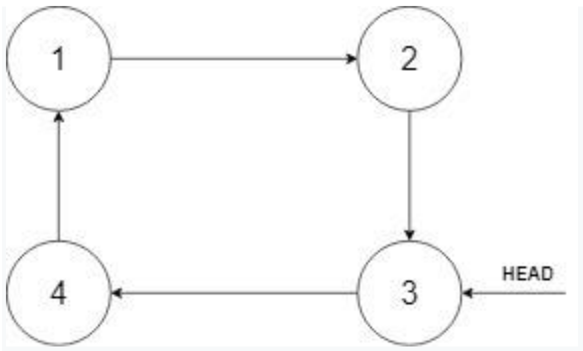
If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the circular list should remain sorted.

If the list is empty (i.e., the given node is `null`), you should create a new single circular list and return the reference to that single node. Otherwise, you should return the originally given node.

Example 1:



```
Input: head = [3,4,1], insertVal = 2

Output: [3,4,1,2]

Explanation: In the figure above, there is a sorted circular list of three elements.
You are given a reference to the node with value 3, and we need to insert 2 into the
list. The new node should be inserted between node 1 and node 3. After the insertion,
the list should look like this, and we should still return node 3.
```



Example 2:

```
Input: head = [], insertVal = 1

Output: [1]

Explanation: The list is empty (given head is null). We create a new single circular
list and return the reference to that single node.
```

Example 3:

```
Input: head = [1], insertVal = 0

Output: [1,0]
```

Constraints:

- `0 <= Number of Nodes <= 5 * 10`^{4}
- `-10`^{6} `<= Node.val, insertVal <= 10`^{6}

Q3. Design your implementation of the circular double-ended queue (deque).

Implement the `MyCircularDeque` class:

- `MyCircularDeque(int k)` Initializes the deque with a maximum size of `k`.
- `boolean insertFront()` Adds an item at the front of Deque. Returns `true` if the operation is successful, or `false` otherwise.
- `boolean insertLast()` Adds an item at the rear of Deque. Returns `true` if the operation is successful, or `false` otherwise.
- `boolean deleteFront()` Deletes an item from the front of Deque. Returns `true` if the operation is successful, or `false` otherwise.
- `boolean deleteLast()` Deletes an item from the rear of Deque. Returns `true` if the operation is successful, or `false` otherwise.
- `int getFront()` Returns the front item from the Deque. Returns `-1` if the deque is empty.

- `int getRear()` Returns the last item from Deque. Returns `-1` if the deque is empty.
- `boolean isEmpty()` Returns `true` if the deque is empty, or `false` otherwise.
- `boolean isFull()` Returns `true` if the deque is full, or `false` otherwise.

Example 1:

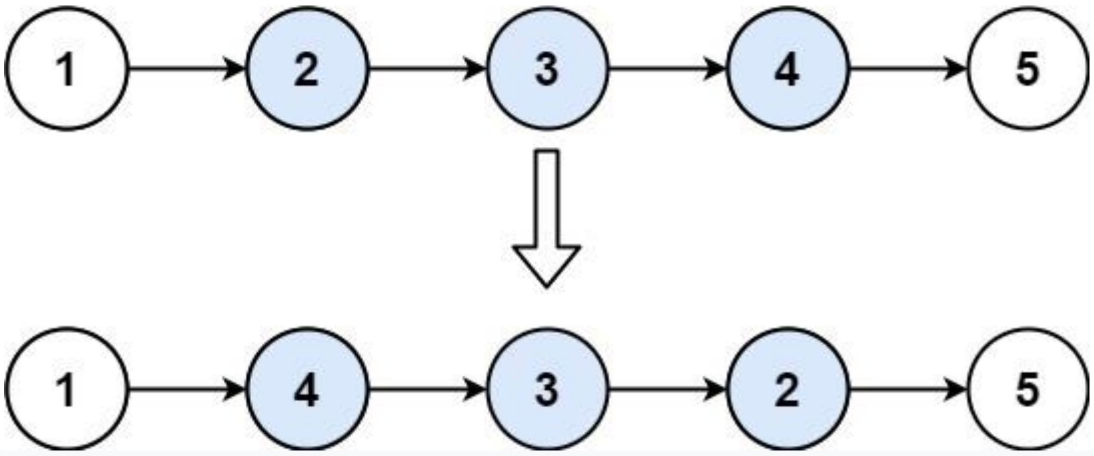
```
Input

["MyCircularDeque", "insertLast", "insertLast", "insertFront", "insertFront",
"getRear", "isFull", "deleteLast", "insertFront", "getFront"]

[[3], [1], [2], [3], [4], [], [], [], [4], []]

Output

[null, true, true, true, false, 2, true, true, true, 4]



Explanation

MyCircularDeque myCircularDeque = new MyCircularDeque(3);

myCircularDeque.insertLast(1);  // return True

myCircularDeque.insertLast(2);  // return True

myCircularDeque.insertFront(3); // return True

myCircularDeque.insertFront(4); // return False, the queue is full.

myCircularDeque.getRear();      // return 2

myCircularDeque.isFull();       // return True

myCircularDeque.deleteLast();   // return True

myCircularDeque.insertFront(4); // return True

myCircularDeque.getFront();     // return 4
```

Constraints:

- `1 <= k <= 1000`
- `0 <= value <= 1000`
- At most `2000` calls will be made to `insertFront`, `insertLast`, `deleteFront`, `deleteLast`, `getFront`, `getRear`, `isEmpty`, `isFull`.

Q4. Given the `head` of a singly linked list and two integers `left` and `right` where `left <= right`, reverse the nodes of the list from position `left` to position `right`, and return *the reversed list*.

Example 1:



```
Input: head = [1,2,3,4,5], left = 2, right = 4

Output: [1,4,3,2,5]
```

Example 2:

```
Input: head = [5], left = 1, right = 1

Output: [5]
```

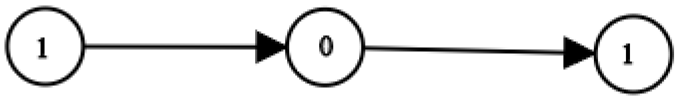
Constraints:

- The number of nodes in the list is `n`.
- `1 <= n <= 500`
- `-500 <= Node.val <= 500`
- `1 <= left <= right <= n`

Q5. Given `head` which is a reference node to a singly-linked list. The value of each node in the linked list is either 0 or 1. The linked list holds the binary representation of a number.

Return the *decimal value* of the number in the linked list.

Example 1:



```
Input: head = [1,0,1]

Output: 5

Explanation: (101) in base 2 = (5) in base 10
```

Example 2:

```
Input: head = [0]

Output: 0
```

Example 3:

```
Input: head = [1]

Output: 1
```

Example 4:

```
Input: head = [1,0,0,1,0,0,1,1,1,0,0,0,0,0,0]

Output: 18880
```

Example 5:

```
Input: head = [0,0]

Output: 0
```

Constraints:

- The Linked List is not empty.
- Number of nodes will not exceed `30`.
- Each node's value is either `0` or `1`.

Q6. Design a queue that supports `push` and `pop` operations in the front, middle, and back.

Implement the `FrontMiddleBack` class:

- `FrontMiddleBack()` Initializes the queue.
- `void pushFront(int val)` Adds `val` to the **front** of the queue.
- `void pushMiddle(int val)` Adds `val` to the **middle** of the queue.
- `void pushBack(int val)` Adds `val` to the **back** of the queue.
- `int popFront()` Removes the **front** element of the queue and returns it. If the queue is empty, return `-1`.
- `int popMiddle()` Removes the **middle** element of the queue and returns it. If the queue is empty, return `-1`.
- `int popBack()` Removes the **back** element of the queue and returns it. If the queue is empty, return `-1`.

Notice that when there are **two** middle position choices, the operation is performed on the **frontmost** middle position choice. For example:

- Pushing `6` into the middle of `[1, 2, 3, 4, 5]` results in `[1, 2, 6, 3, 4, 5]`.
- Popping the middle from `[1, 2, 3, 4, 5, 6]` returns `3` and results in `[1, 2, 4, 5, 6]`.

Example 1:

```
Input:

["FrontMiddleBackQueue", "pushFront", "pushBack", "pushMiddle", "pushMiddle",
"popFront", "popMiddle", "popMiddle", "popBack", "popFront"]

[[], [1], [2], [3], [4], [], [], [], [], []]

Output:

[null, null, null, null, null, 1, 3, 4, 2, -1]



Explanation:

FrontMiddleBackQueue q = new FrontMiddleBackQueue();

q.pushFront(1);   // [1]

q.pushBack(2);    // [1, 2]

q.pushMiddle(3);  // [1, 3, 2]

q.pushMiddle(4);  // [1, 4, 3, 2]

q.popFront();     // return 1 -> [4, 3, 2]

q.popMiddle();    // return 3 -> [4, 2]
```

```
q.popMiddle();    // return 4 -> [2]

q.popBack();      // return 2 -> []

q.popFront();     // return -1 -> [] (The queue is empty)
```

Constraints:

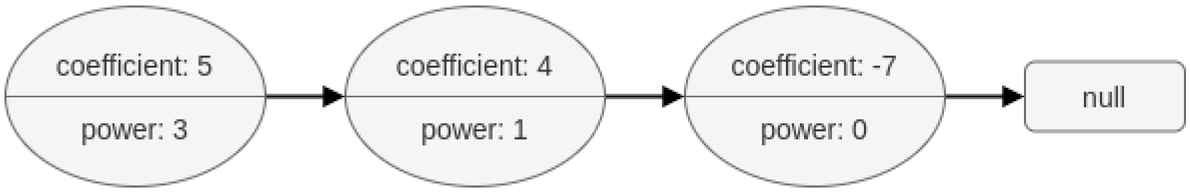
- `1 <= val <= 10^9`
- At most `1000` calls will be made to `pushFront`, `pushMiddle`, `pushBack`, `popFront`, `popMiddle`, and `popBack`.

Q7. A polynomial linked list is a special type of linked list where every node represents a term in a polynomial expression.

Each node has three attributes:

- `coefficient`: an integer representing the number multiplier of the term. The coefficient of the term $9x^4$ is `9`.
- `power`: an integer representing the exponent. The power of the term $9x^4$ is `4`.
- `next`: a pointer to the next node in the list, or `null` if it is the last node of the list.

For example, the polynomial $5x^3 + 4x - 7$ is represented by the polynomial linked list illustrated below:



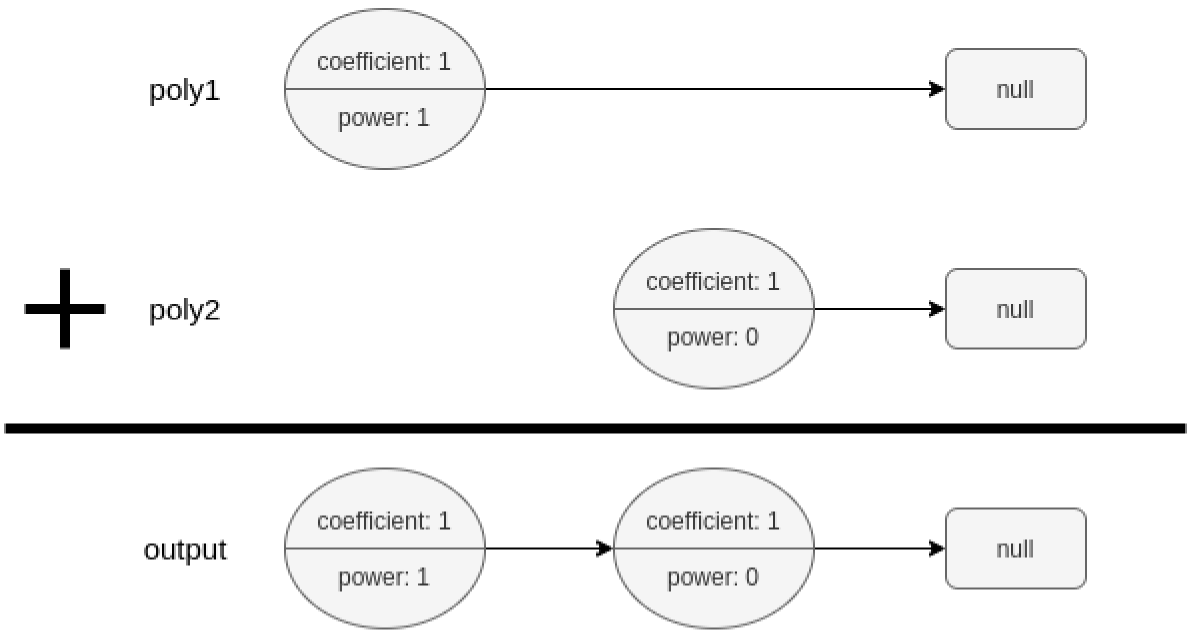
The polynomial linked list must be in its standard form: the polynomial must be in **strictly** descending order by its `power` value. Also, terms with a `coefficient` of `0` are omitted.

Given two polynomial linked list heads, `poly1` and `poly2`, add the polynomials together and return *the head of the sum of the polynomials*.

`PolyNode` format:

The input/output format is as a list of `n` nodes, where each node is represented as its `[coefficient, power]`. For example, the polynomial $5x^3 + 4x - 7$ would be represented as: `[[5,3],[4,1],[-7,0]]`.

Example 1:



```
Input: poly1 = [[1,1]], poly2 = [[1,0]]

Output: [[1,1],[1,0]]

Explanation: poly1 = x. poly2 = 1. The sum is x + 1.
```

Example 2:

```
Input: poly1 = [[2,2],[4,1],[3,0]], poly2 = [[3,2],[-4,1],[-1,0]]

Output: [[5,2],[2,0]]

Explanation: poly1 = 2x² + 4x + 3. poly2 = 3x² - 4x - 1. The sum is 5x² + 2. Notice
that we omit the "0x" term.
```

Example 3:

```
Input: poly1 = [[1,2]], poly2 = [[-1,2]]

Output: []

Explanation: The sum is 0. We return an empty list.
```

Constraints:

- `0 <= n <= 10`^{4}
- `-10`^{9}` <= PolyNode.coefficient <= 10`^{9}
- `PolyNode.coefficient != 0`
- `0 <= PolyNode.power <= 10`^{9}
- `PolyNode.power > PolyNode.next.power`