AlgoExpert   Quad Layout   00:00:00 | 00:00:00

Prompt   Scratchpad   Our Solution(s)   Video Explanation

Difficulty: 🟦 **Category:** Linked Lists **Successful Submissions:** 10,144+

Linked List Construction

Write a `DoublyLinkedList` class that has a `head` and a `tail`, both of which point to either a linked list `Node` or `None` / `null`. The class should support:

- Setting the head and tail of the linked list.
- Inserting nodes before and after other nodes as well as at given positions (the position of the head node is `1`).
- Removing given nodes and removing nodes with given values.
- Searching for nodes with given values.

Note that the `setHead`, `setTail`, `insertBefore`, `insertAfter`, `insertAtPosition`, and `remove` methods all take in actual `Node`s as input parameters—not integers (except for `insertAtPosition`, which also takes in an integer representing the position); this means that you don't need to create any new `Node`s in these methods. The input nodes can be either stand-alone nodes or nodes that are already in the linked list. If they're nodes that are already in the linked list, the methods will effectively be *moving* the nodes within the linked list. You won't be told if the input nodes are already in the linked list, so your code will have to defensively handle this scenario.

If you're doing this problem in an untyped language like Python or JavaScript, you may want to look at the various function signatures in a typed language like Java or TypeScript to get a better idea of what each input parameter is.

Each `Node` has an integer `value` as well as a `prev` node and a `next` node, both of which can point to either another node or `None` / `null`.

Sample Usage

```
// Assume the following linked list has already been created:
1 <-> 2 <-> 3 <-> 4 <-> 5
// Assume that we also have the following stand-alone nodes:
3, 3, 6
setHead(4): 4 <-> 1 <-> 2 <-> 3 <-> 5 // set the existing node with value 4 as the head
setTail(6): 4 <-> 1 <-> 2 <-> 3 <-> 5 <-> 6 // set the stand-alone node with value 6 as the tail
```

```
insertBefore(6, 3): 4 <-> 1 <-> 2 <-> 5 <-> 3 <-> 6 // move the existing node with value 3 before the existing node with value 6
insertAfter(6, 3): 4 <-> 1 <-> 2 <-> 5 <-> 3 <-> 6 <-> 3 // insert a stand-alone node with value 3 after the existing node with value
insertAtPosition(1, 3): 3 <-> 4 <-> 1 <-> 2 <-> 5 <-> 3 <-> 6 <-> 3 // insert a stand-alone node with value 3 in position 1
removeNodesWithValue(3): 4 <-> 1 <-> 2 <-> 5 <-> 6 // remove all nodes with value 3
remove(2): 4 <-> 1 <-> 5 <-> 6 // remove the existing node with value 2
containsNodeWithValue(5): true
```

Hints

Hint 1

When dealing with linked lists, it's very important to keep track of pointers on nodes (i.e., the "next" and "prev" properties on the nodes). For instance, if you're inserting a node in a linked list, but that node is already located somewhere else in the linked list (in other words, if you're moving a node), it's crucial to completely update the pointers of the adjacent nodes of the node being moved before updating the node's own pointers. The order in which you update nodes' pointers will make or break your algorithm.

Hint 2

Realize that the insertBefore() and insertAfter() methods can be used to implement the setHead(), setTail(), and insertAtPosition() methods; making the insertBefore() and insertAfter() methods as robust as possible will simplify your code for the other methods. Make sure to take care of edge cases involving inserting nodes before the head of the linked list or inserting nodes after the tail of the linked list.

Hint 3

Similar to Hint #2, realize that the remove() method can be used to implement the removeNodesWithValue() method as well as parts of the insertBefore() and insertAfter() methods; make sure that the remove() method handles edge cases regarding the head and the tail.

Optimal Space & Time Complexity

setHead, setTail, insertBefore, insertAfter, and remove: O(1) time | O(1) space insertAtPosition: O(p) time | O(1) space - where p is input position removeNodesWithValue, containsNodeWithValue: O(n) time | O(1) space - where n is the number of nodes in the linked list