Half-Exam 2 Feedback Name:

Problem 1

Rubric:

4	check & throw exception
4	Get and return previous element. See ⑤.
5	Find head
8	Walk from one node to the next correctly
7	Set the correct value. See ⑤
12	Iterate to the correct node. See -4 ④ wiping elements before the node. See ⑥ for off-by-one.

① Only need an E for the temp variable, not a whole new Node(…). In particular, you do not need to splice a new node in the list for set(), only for add()!

② -2 An empty list has a null head reference. Watch out for head.value or head.next crashing your code!

③ -2 Do not call size(). Use the size variable. (Problem specifies: "Do not call other methods of the LinkedList class.") Leaving the parenthesis off indicates you know that you have a cached variable and don't need to loop through the list to determine this. Similarly, don't need to call current.next(). Just use current.next to refer to the variable itself.

④ -4 If the code current.value = element is placed inside the loop, all the element before the given one will be set, too! Move this code after the loop to avoid this bug.

⑤ -4 returning the element stored, not the node at that index. `E saved = current.value;` instead of `E saved = current;`

-4 set the element stored, not the node itself. `current.value = element;` instead of `current = element;`

⑥ -3 Off by one. Stops at (or inserts the edited note) at the node just before or after the current node. A loop `Node current = head; for(int i = 0; i < index; i++) {current = current.next;}` is sufficient to get a pointer pointing to the node at index `index`.

⑦

⑧

⑨

⑩

⑪

⑫

⑬

⑭

⑮

⑯

⑰

⑱

⑲

⑳

Problem 4

① For description, look at big picture rather than translating code to words. Instead of writing "The loop went twice, pointing the head to the node with Charles", write "the first half of the list is deleted."

Problem 5

① Part e: You don't need to consider copying the array for Java's implementation because, most of the time, it is not necessary to copy the array. In the rare cases where a copy is needed, the internal capacity is doubled so that we do not need to copy the array again for a long time, such that, on average, it takes O(1) per add to do the copy. This is called "amortized" runtime, and we assume it throughout the quarter. On the rother hand, shifting the elements **does** cost O(n) when e.g., n/2 elements must be shifted.