Answer to Question 3.5

If we removed the special case lines in removeFirst() that set tail = null when the list becomes empty:

```
if (size == 0)
   tail = null;   // omitted
```

the class would still work correctly for the currently implemented methods (isEmpty, first, last, addFirst, addLast, removeFirst).
This is because all these methods use size as the single source of truth for determining whether the list is empty.

- isEmpty() checks size == 0.
- first() and last() both short-circuit and return null if isEmpty() is true, so they never try to use the stale tail.
- addLast() also checks isEmpty() before deciding whether to reuse tail.

Thus, even if tail is left pointing at a removed node, the methods won't break because the size guard prevents them from using it.

Why it is still risky

- The list's invariant (size == 0 ⇒ head == null && tail == null) is violated.
- The dangling tail prevents the last removed node from being garbage collected, leading to a memory leak.
- If future methods are added (for example, a method that directly prints the tail or inspects it without checking size), they may behave incorrectly because tail will not represent the true state of the list.

Conclusion

For the given implementation, the class would *still work* because of the consistent use of `size` checks. However, it is considered **bad practice and unsafe**, because it breaks the clean design invariant and makes the code error-prone if extended later.

Question 3.6

The method checks if the list has fewer than 2 nodes (head == null or head.getNext() == null); if so, it returns null.

It starts with a pointer walk at the head.

It moves forward while walk.getNext().getNext() != null, ensuring it stops just before the last node.

When the loop ends, walk points to the second-to-last node.

Finally, it returns walk.getElement(), the element stored in that node.

Question 3.9

```
public int size() {
    int count = 0;
    Node<E> walk = head;
    while (walk != null) {
        count++;
        walk = walk.getNext();
    }
    return count;
}
```

Explanation

1. Start a counter count = 0.

2. Begin traversal from the head node.

3. For each node, increment count and move to the next node (walk = walk.getNext()).

4. Stop when you reach the end (null).

5. Return the total count.

Question 3.12

```
public void rotate() {
    if (head != null && head != tail) {   // at least 2 nodes required
        Node<E> oldHead = head;          // save reference to current head
        head = head.getNext();           // move head to next node
        tail.setNext(oldHead);           // link old head after current tail
        oldHead.setNext(null);           // old head is now the last node
        tail = oldHead;                  // update tail reference
    }
}
```

Explanation

1. If the list has 0 or 1 node → nothing to rotate.
2. Save the current head as oldHead.
3. Move head to the second node (head.getNext()).
4. Attach oldHead after the current tail.
5. Set oldHead.next = null (it's now the last node).
6. Update tail to point to oldHead.

Answer 3.17

Approach 1

Algorithm 1: Using Extra Space (Hashing)

- Create a boolean array seen[n].
- Traverse A:

    - If seen[A[i]] == true → A[i] is the repeated number.
    - Else mark seen[A[i]] = true.
- Time: O(n)
- Space: O(n)

Algorithm 2: In-Place (Marking / Negation Trick)

- Since elements are in [1 … n−1], we can use the indices as markers.
- Traverse array:
    - Let val = abs(A[i]).
    - If A[val] is already negative → val is the repeated number.
    - Else mark A[val] = -A[val].
- Time: O(n)
- Space: O(1)
- Note: modifies the array.

Algorithm 3: Floyd’s Cycle Detection (Tortoise & Hare)

This is the most elegant, doesn’t modify array, and uses constant space.

- Think of array values as pointers: from index i you “jump” to A[i].
- Because there’s a duplicate, the structure forms a cycle.
- Use Floyd’s Tortoise & Hare algorithm:
    1. Initialize slow = A[0], fast = A[A[0]].
    2. Move slow = A[slow], fast = A[A[fast]] until they meet (cycle detected).
    3. Reset slow = 0, keep fast where it is.

4. Move both one step at a time until they meet again → the meeting point is the duplicate number.

- Time: O(n)
- Space: O(1)
- Does not modify the array.

Question 3.18

1. Hashing Approach (Easy to Code, O(n) Space)

1. Create a frequency array freq of size n (or a hash map).
2. Traverse B:
   - For each value x = B[i], increment freq[x].
3. Collect all x such that freq[x] > 1.
4. Return the five numbers.

- Time: O(n)
- Space: O(n)

---

2. In-Place Marking (Negation Trick, O(1) Extra Space)

Since all numbers are in [1 … n−5], we can use indices as markers:

1. Traverse array B:
   - Let val = abs(B[i]).
   - If B[val] < 0, then val has been seen → duplicate.

- Else mark it: B[val] = -B[val].

2. Collect duplicates.

- Time: O(n)
- Space: O(1)
- Note: This modifies the array (but can be reverted by another pass).

Answer 3.25

Algorithm: Concatenate L and M into L′

We want L′ = L + M (all nodes of L followed by all nodes of M).

1. Check if L is empty
   - If L.head == null, then simply return M as L′.
2. Check if M is empty
   - If M.head == null, then return L as L′.
3. Otherwise
   - Set L.tail.next = M.head (connect the last node of L to the first node of M).
   - Update L.tail = M.tail (so the new tail is M’s tail).
   - Update L.size = L.size + M.size (if you maintain size).
4. Return L (which now represents L′).

Answer 3.28

Algorithm (Iterative, In-Place)

1. Initialize:
   - prev = null
   - curr = head
2. Loop while curr != null:
   - Save next = curr.next (store next node).
   - Reverse pointer: curr.next = prev.
   - Move prev = curr.
   - Move curr = next.
3. At the end, prev will point to the new head.
   - Update head = prev.