Lab04 Report Justin Schlag

1. Problem:

This lab needs doubly linked list that stores double values. Each node contains two references, one pointing to the next, and one to the previous. Bidirectional traversal, making insertions and deletions more flexible.

The challenge is to manage the pointers when adding, removing, and traversing elements, ensure list integrity. The program must include the functionalities such as adding and removing nodes, navigating through the list, and modifying values.

2. Solution Description:
To solve the problem, I made a doubly linked list (DoubleDoubleLL) that stores values and allows the traversal. It consists of an internal node class and various methods to manage the linked list operation.

The linked list is managed using three key references: head, tail, and current. The head points to the first node in the list, the tail points to the last node, and current is used to track the position during traversal. Insertion methods are designed to add new nodes either at the end of the list or directly after the current node, ensuring that all links are properly set to maintain the doubly linked list structure.

3. Problems encountered:
One main issue I had were several syntax errors at first due to incorrect method signatures. I also had a run-time error that took a while to figure out. Initially, the gotoNext() and gotoPrev() methods did not check whether current was null before attempting to move to the next or previous node. This caused a NullPointerException when trying to access current.next or current.prev.
I also had a logic error in the addAfterCurrent() method. When inserting a new node after the current position, the prev and next references were not being correctly updated in all cases, and especially when inserting after the tail node.
I also ran into troubles creating the remove method. This lab was definitely a learning curve but I enjoyed figuring out this puzzle.

4. In Java, when an object becomes **unreachable** because there are no remaining references to it, the Java Virtual Machine (JVM) automatically removes it through garbage collection (GC). The Garbage Collector (GC) reclaims memory by identifying and deleting

such objects, preventing memory leaks and optimizing performance.

5. Advantages: Dynamic sizing to grow or shrink in memory, making it more flexible. You can efficiently insert and delete elements. Better memory utilization as there is no wasted space.
Disadvantages: Slower random access, linked list requires O(n) traversal to access specific elements. It is cache inefficient, because linked lists are scattered in memory. Linked lists are also very complex to implement, way more than an array.

6. Advantages of a doubly linked list:
Bidirectional traversal (as stated earlier). Easier deletion of nodes. It also has more flexible insertions, as both next and previous pointers exist, allowing for insertions before or after a given node. Operations like reversing the list are more efficient.
Disadvantages:
Increased memory usage, having the prev pointer. More complex to implement, requiring much more logic for maintaining next and prev. This is slightly slower than SLLs. Singly linked list is the better choice if simplicity and memory efficiency are priority.

7.
while (temp.link != null) //incorrect loop condition
It means temp is not the last node, but it does not print the last node's value.
Fix: Use while (temp != null) instead

System.out.println(temp); //also incorrect. This prints the memory reference of the node instead od the string value. You must use temp.data instead.

8. Incorrect use of continue. If t.data == null, the continue statement skips the rest of the loop but does not move t to the next node, causing an infinite loop if a null data is encountered. Fix: Replace continue; with t = t.link; to properly move forward.

The loop starts with t = head, meaning the first node is checked twice—once before the loop (ret = head.data) and again inside the loop.

Fix: Change t = head; to t = head.link; so the loop starts from the second node.

Also If t.data == null, calling .length() will cause a NullPointerException.

Fix: Ensure t.data is not null before calling .length().

9. Incorrect replacement logic: temp = rValue; This does not change the data inside the node, it reassigns the temp reference itself. You must replace it with temp.data = rValue;

head = head.link; This incorrectly modifies head inside the loop, which can corrupt the linked list by shifting its starting point. Fix: Instead of modifying head, the loop should update temp = temp.link; to traverse properly.

Also the break; statement exits the loop after replacing only the first occurrence, instead of replacing all occurrences. Remove break; so the loop continues.

If temp.data is null, the continue; statement skips updating temp to the next node, causing an infinite loop.

Fix: Change continue; to temp = temp.link; to properly advance to the next node.

10. The head reference is not updated:

```
ListNode temp = head;

for (int i = 0; i < 5; i++) {

   temp = temp.link;

}
```

This only moves temp forward but does not update head, meaning the first five elements are not actually removed—head still points to the first node.

Fix: Assign head = temp; after the loop to update the list.

No Check for List Size (Risk of NullPointerException)

Problem: If the list contains fewer than 5 elements, temp.link may become null, leading to a NullPointerException when temp = temp.link is executed.

Fix: Add a check to stop if temp reaches null.