Justin Schlag Lab03 Report

1. Problem

This lab requires creating a grocery list using a singly linked list. The program should allow adding, removing, and searching for grocery items while keeping track of their names and prices. It must also calculate the total cost of all items in the list.

2. Solution Description:
The program uses a singly linked list to store grocery items. Each item is represented by the GroceryItem class, which holds the name and price. The GroceryList class manages the linked list, allowing items to be added, removed, searched, and updated. It also includes methods to traverse the list and calculate the total cost of all items. The program follows object-oriented principles to keep the implementation structured and efficient.

3. Problems Encountered:
The head was set to an empty ListNode, which caused issues when adding the first item. Changing head to null and handling it properly in addItem() fixed the problem.

Logic Error in contains() – The method wasn't correctly checking for equality between grocery items. Ensuring equals() compared both name and value fixed this.

Some items with a value of 0.0 were not being included properly.

I also had a hard time starting up the program on VS code. For some reason my file systems were messed up and it was unable to show my progress.

4. Advantages of Linked list instead of an array

Dynamic Size – Linked lists can grow and shrink as needed without requiring a fixed size, unlike arrays.

Efficient Insertions/Deletions – Adding or removing elements in a linked list is faster because there's no need to shift elements, as required in an array.

No Wasted Memory – Linked lists allocate memory as needed, while arrays may have unused space if preallocated.

Arrays require a large block of memory, whereas linked lists can be stored in scattered locations.

5. Disadvantages

More Memory Usage – Each node in a linked list requires extra memory for storing pointers, unlike arrays, which only store data.

Slower Access Time – Linked lists require sequential access, meaning you must traverse the list to reach an element, whereas arrays allow random access using an index.

Increased Complexity – Managing pointers (inserting, deleting, and updating links) adds extra complexity compared to arrays.

This is a slower method overall, because iterating through a linked list requires following pointers one by one.

6. It is missing traversal step. So an infinite loop.
The printAllEvenValues() method never advances temp to the next node, causing an infinite loop when temp != null. Since temp never updates, the condition while(temp != null) runs forever, and the program doesn't print anything.

To fix: add temp = temp.link; to ensure that the loop progresses. This will prevent an infinite loop.

7. The issue here is that it is losing the rest of the list.

The current addFirst(int aData) method only sets head to the new node, without linking it to the existing list. As a result, all previous nodes are lost, and the list only contains the newly added node.

To keep the rest of the list, the new node's link should point to the current head before updating head.
corrected code:

```
public void addFirst(int aData) {

  ListNode newNode = new ListNode(aData, head); // Link new node to the existing list

  head = newNode; // Update head to the new node

}
```

8. The head is not updated:
The code creates a local variable temp and moves it to the next node (temp = temp.link;).

However, this does not change the actual head of the linked list, so the original list remains unchanged.

To fix: you must check if head is null. And move head to the next node, removing the first node ex:

```
public void removeFirst() {

   if (head != null) { // Check if the list is not empty

      head = head.link; // Move head to the next node

   }

}
```

This Updated head instead of modifying a local variable and ensured the first node is properly removed

9. The issues lies in the NullPointerException in removeLast()
To correctly remove the last node, we need to stop at the second-to-last node, then set its link to null.

corrected:

```
public void removeLast() {

 if (head == null) return; // If list is empty, do nothing

if (head.link == null) { // If only one node exists head = null; return; }

 ListNode temp = head; while (temp.link.link != null) { // Stop at the second-to-last node

temp = temp.link; }

temp.link = null; // Remove the last node }
```

This: Checks if the list is empty (head == null) → Do nothing.

Checks if there is only one node (head.link == null) → Set head = null.

Traverses to the second-to-last node (temp.link.link != null) → Ensures temp does not become null.

Sets temp.link = null to remove the last node.

10. The while loop condition while (temp.link != null) stops at the second-to-last node, so the last node's value is never multiplied into ret.

Fix: Change the Loop Condition to while (temp != null)

This ensures that all nodes, including the last one, are included in the product calculation.