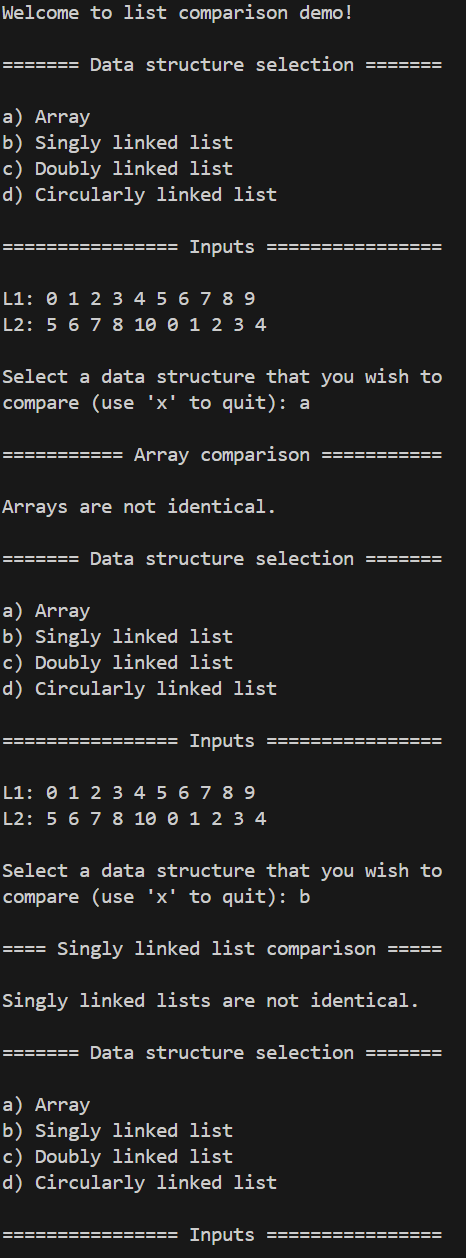
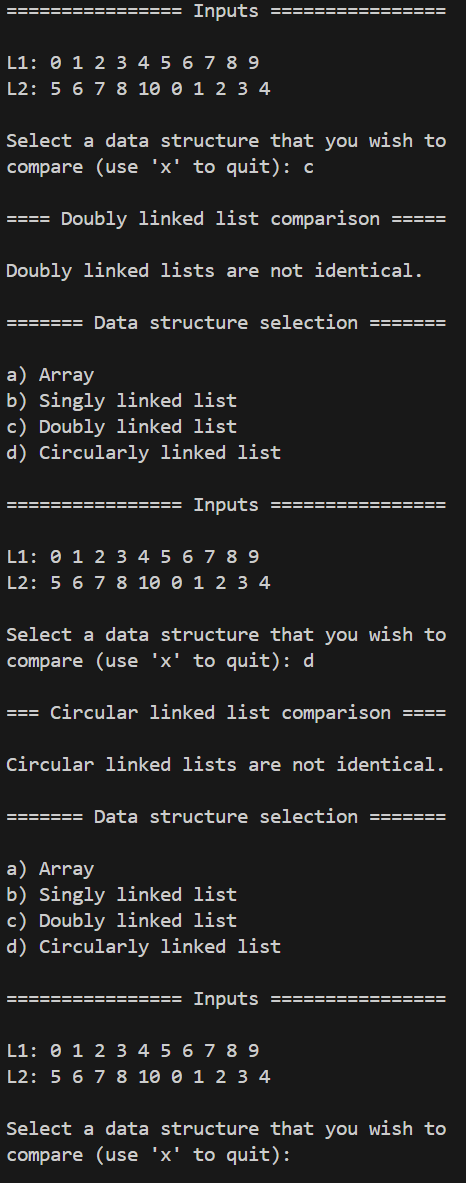
Submit screenshots of the results, code files, and a writeup describing your solutions in a zip file.



**Figure 1.** Console output for two identically sized lists containing one mismatched element value.

All implementations assume that the input lists are sorted in non-decreasing order, i.e., contain no duplicate elements. In addition, all comparison algorithms terminate early in certain scenarios, such as when the list sizes are not equivalent. All algorithms return a Boolean value indicating whether the lists match, despite potentially having different starting points.

**Array Comparison:**

Array comparison begins by iterating through the two lists’ elements until two elements are found to match. The index at which this occurs is used as a pivot for subsequent comparisons. If a pivot value is found which does not exceed the size of the arrays, then elements to the right of this pivot index are compared with the elements at the beginning of the other array. If all elements until the end of the pivoted array are found to match the beginning indices of the other array, then execution continues. At this point, elements from the beginning of the pivoted array up until the pivot index are compared with the remaining elements of the other array. If the elements before and after the pivot of the pivoted array match the elements in sequential order of the other array, the arrays are equivalent with a different starting point. The algorithm has linear time complexity and constant space complexity.

**Singly Linked List Comparison:**

Singly linked list comparison begins by iterating through the two lists’ elements and simultaneously moving the front node to the rear of one of the lists, until the two front nodes match in value. If this occurs in fewer swaps than there are nodes in the lists, then execution continues. At this point, the remaining elements in the two lists are checked for comparison by moving the front node to the back of each list. If at any point during this iterative behavior, the front nodes do not match in value, then the algorithm terminates, and the lists are inequivalent. If the number of iterations of this second step match the number of nodes in the lists less 1 (given that the original front nodes were already found to match), then the lists are equivalent. The algorithm has linear time complexity and linear space complexity, since copies of the lists are created. Otherwise, the input lists are modified and potentially left in an unknown state if execution terminates early, by virtue of the node swaps from front to back.

**Doubly Linked List Comparison:**

Doubly linked list comparison begins by iterating through the two lists’ elements and simultaneously swapping the front node of one of the lists to the back (or vice versa, depending on if the value is found to be greater or less than the reference element since we know that the lists are sorted), until the two front nodes match in value. If this occurs in fewer swaps than there are nodes in the lists, then execution continues. At this point, the remaining elements in the two lists are checked for comparison removing the front elements from each. If at any point during this iterative behavior, the front nodes do not match in value, then the algorithm terminates, and the lists are inequivalent. If after this second step, both lists are empty, then the lists are equivalent. The algorithm has linear time complexity and linear space complexity, since copies of the lists are created. Otherwise, the input lists are modified and potentially left in an unknown state if execution terminates early, by virtue of the node swaps from front to back (or vice versa) and/or node removals.

**Circular (Singly) Linked List Comparison:**Circular linked list comparison begins by iterating through the two lists’ elements and simultaneously advancing the cursor of one of the lists, until the two front nodes match in value. If this occurs in fewer advances than there are nodes in the lists, then execution continues. At this point, the remaining elements in the two lists are checked for comparison by advancing the cursor for each and comparing the value of the new front node. If at any point during this iterative behavior, the front nodes do not match in value, then the algorithm terminates, and the lists are inequivalent. If the number of iterations of this second step match the number of nodes in the lists less 1 (given that the original front nodes were already found to match), then the lists are equivalent. The algorithm has linear time complexity and linear space complexity, since copies of the lists are created. Otherwise, the input lists are modified and potentially left in an unknown state if execution terminates early, by virtue of the cursor advances.

**Future Improvements:**

The code can be improved through further polishing, more explicit commenting, as well as allowing the user to select their own input data.