Question 1:

In order to check for memory leaks with added solution you need to use MSVC compiler. Other compilers might not work properly with `_CrtDumpMemoryLeaks();`

Question 2:

Insert function: (Requested function.)

1. **Insert when head is null. Expecting LinkedList_NOK.**
2. **Insert NULL. Expecting LinkedList_NOK.**
3. **Insert test_strings[0]. Verify by comparing *head->data to test_strings[0]. Expecting LinkedList_OK.**
4. **Insert test_strings[1]. Verify by comparing *head->data to [test_strings[0]; *head =head->next; compare *head->data to test_strings[1]; Expecting LinkedList_OK.**
5. **Insert all elements from test_strings[] and compare each node data with node data from expected linked list. Expecting LinkedList_OK.**
6. **Insert all elements from test_strings[] + 1 extra value and compare each node data with node data from expected linked list. Expecting comparison to fail.**
7. **Insert element from test_string[]. Check if *head->next = NULL.**
8. **Insert element from test_string[]. Insert again after first element. Check if *head->next->next = NULL;**

Remove function:(Requested function.)

1. **Remove when head is null. Expecting LinkedList_NOK.**
2. **Remove NULL. Expecting LinkedList_NOK.**
3. **Insert test_string[0]. Remove test_string[0]. Check if *head->data = NULL; Expecting LinkedList_OK.**
4. **Insert test string[0], test_string[1]. Remove test_string[1](Remove last element). Check if *head->data = test_string[0]. Expecting LinkedList_OK.**
5. **Insert test string[0], test_string[1], test_string[2]. Remove test_string[0](Remove head). Check if linked list node data = expected linked list node data. Expecting LinkedList_OK.**
6. **Insert all test_string[] elements. Remove test_string[3](Remove from middle). Check by comparing each node data to expected linked list(That does not contain test_string[3]) node data. Expecting LinkedList_OK.**
7. **Insert all test_string[] elements. Remove last element. Check if last element-> next = NULL.**

Deallocation function: (This function is used to free linked list nodes memory. Without deallocation memory leak will happen.)

1. **Deallocate when head is null. Expecting LinkedList_NOK.**
2. **Insert all test_string[] elements. Deallocate. Check if head = NULL. Expecting LinkedList_OK.**
3. **Insert all test_string[] elements. Deallocate twice. Check if head is null. Expecting LinkedList_OK.**

4. **Insert 1 test_string[] element. Deallocate. Check is head is null. Expecting LinkedList_OK.**

Create node function: (This function is used to achieve the requirement that insert and delete methos are called using a tstr_node* argument instead of a char* argument. Otherwise, this code would be moved to insert function.)

1. **Call create_node() when maximum allowed memory is reached. Expecting LinkedList_NOK.**
2. **Call create_node(). Expecting LinkedList_Ok.**

Print function: (This function prints list elements and size of the list on LL_Test program for manual check.)

1. **Print when head is null. Expecting LinkedList_NOK.**
2. **Insert all test_string[] elements. Print list. Expecting LinkedList_OK & Size = expected size**

CleanInput & convertAndCheckForLetters methods are additional functions used for manual testing in the LL_Test.

CleanInput could be tested with some modifications to be able to check for length of input.

ConvertAndCheckForLetters could be tested using other characters that are not letters.

Additional tests:

Memory leak test.

Speed test when inserting a new node in a big sorted linked list.

Speed test when deleting a node in a big sorted linked list.

Testing on different platforms.

Question 3:

For implementing the tests it would be nice to create some functions that would evaluation:

create_list(char* test_string);

list_is_equal(tstr_node head1, tstr_node** head2);**

strs_equal(char* string1, char* string2);

Char* test_String[num_string] = {“string1”,…"string[num_string]”};

Creating an expecting linked_list with strings to be compared to the linked list under test with elements from test_string[].

And it would also be required to use a test framework or design one that would call the function under test and compare its result to expected result and offer a feedback and documentation.

A test function could look similar to:

ASSERT_RESULT (expected value, actual)

ASSERT_RESULT_PTR(expected value, actual)

So we can test like this:

```
Void test_sorted_link_insert(void){
        ASSERT_RESULT(LinkedList_Ok, sorted_link_insert(&head, node_data1);
}
```

ASSERT_RESULT_PTR(node_data1->data, head->data)

ASSERT_RESULT_EQUAL(equal, list_is_equal(expected_head, head))

Memory leak test can be done by calling _CrtDumpMemoryLeaks from crtdbg.h or by using an external tool.

Speed test can be done by calling t = clock() before executing insert method and another call clock() - t at the end of it. (T = clock(); sorted_link_insert(); T = clock() - T; T = time of execution.)

Question 4: (dst = destination; src = source; n = num)

1. **Dst = src, n = sizeof src**
2. **Dst > src, n = sizeof src**
3. **Dst > src, n > sizeof src**
4. **Dst > src, n < sizeof src**
5. **Dst < src, n = sizeof src**
6. **Dst < src, n > sizeof src**
7. **Dst < src, n < sizeof src**
8. **Src = &dst[x], x < sizeof dst (overlapping dst & src)**
9. **Dst = &src[x], x < sizeof src (overlapping dst & src)**
10. **Dst = null;**
11. **Src = null;**
12. **Different types (int, char, unsigned long, signed char, etc)**
13. **Testing on different platforms.**

Question 5:

Class Node with data and next attribute that is used to create new node (Node(Data) will make a new node with data = data and next Null)

Class LinkedList:

- **Display function prints every node of linked list starting from head node. If head == Null return –1**
- **Sorted link insert function adds a new node. If head == null makes new node as head and return 0. If new node data is < head data makes new node as head and returns 0. If else,**

searches for next node data <= data and makes previous node point to new node and new node point to next node.

- **Sorted link remove function removes a node. If head == null returns –1. If not, goes node by node until node data == data to be removed, then makes previous node point to what node to be removed points to and if node to be removed is the actual node, then makes head == next node.**