

You may complete Task 1 and Task 2 after successfully completing exercise D, and all previous sections of the assigned lab. After completing Task 1 and 2, you should complete all parts of the lab before attempting Task 3.

Before you start these tasks, make sure to copy the contents of [addendum.c](#) and [addendum.h](#) into your respective string_list source code files.

Task 1: Reversing String Linked List

In this task, you will reverse the linked list you just created by implementing the [string_list_reverse](#) function. The function signature provided below:

```
void string_list_reverse(string_list_t* lst);
```

This function accepts an already initialized list, and reverses the contents. This function does not return a value, instead, it should modify the [string_list_t](#) struct, updating its pointer to reference the reversed list.

Hint: Since you have already implemented the [string_list_destroy](#) and [string_list_insert](#) functions, consider how you could use these functions to help you efficiently create a reversed list. You may have to create a new list before updating [string_list_t](#).

Here is a example run to test the reverse command:

```
Enter a command. Type "help" to see available options.
> insert 5
> insert 19
> insert hello
> print
hello
19
5
> reverse
List reversed.
> print
5
19
hello
> █
```

Task 2: Inserting at the ith Position

In this task, you will implement the `string_list_insert_at_position` function to insert a string into the linked list at a specified position. The function signature provided below.

```
bool string_list_insert_at_position(string_list_t* lst, size_t position, char* str);
```

This function takes the following parameters:

- **lst**: A pointer to an already initialized linked list (`string_list_t` structure).
- **position**: The index (0-based) where the string should be inserted.
- **str**: A pointer to the string to insert.

The function updates the list to reflect the new element at the desired position and returns `true` if the insertion is successful. If the position is invalid, the function should handle the error gracefully, but printing out an error and returning `false`.

Note that to insert at position X, you need to update the pointer from the previous node to the new node. You can achieve this by creating a trailing pointer that tracks the previous node during traversal. You may also want to consider any special boundary cases, and see if you can reuse the `string_list_insert` to simplify the implementation.

Here is an example run to test the `inserti` command:

```
Enter a command. Type "help" to see available options.
> inserti 0 cat
Inserted "cat" at position 0.
> print
cat
> insert dog
> print
dog
cat
> inserti 1 duck
Inserted "duck" at position 1.
> print
dog
duck
cat
> inserti 3 cow
Inserted "cow" at position 3.
> print
dog
duck
cat
cow
> inserti 10 rabbit
Error: Index 10 is out of bounds. Valid range is 0 to 4.
Failed to insert at position 10. Ensure the position is valid.
```

Task 3: Keeping Track of Size

In the current implementation of the `string_list_t` linked list, the length of the list is calculated dynamically by traversing the nodes each time the `string_list_length` function is called. While this approach works, it is inefficient, especially for large lists. A better approach is to store the size of the list by keeping track of it in the `string_list_t` structure. This is also referred to as memoization of the list.

When implementing memoization consider the following:

- Think about how you can modify the `string_list_t` struct to include a size field that stores the current length of the list?
- How would the `string_list_length` function change to incorporate this update?
- Identify which additional functions in the current implementation would need to be modified to maintain the correct size of the list?