Lab 10 - Singly-Linked Lists

General Requirements

None of the functions you write should call `calloc` or `realloc`.

None of the functions you write should perform console input; i.e., contain `scanf` statements. Unless otherwise specified, none of your functions should produce console output; i.e., contain `printf` statements.

You must format your C code so that it adheres to one of two commonly-used conventions for indenting blocks of code and placing braces (K&R style or BSD/Allman style). Instructions were given in the VS Code installation instructions.

Finish each exercise (i.e., write the function and verify that it passes all of its tests) before you move on to the next one. Do not leave testing until after you have written all your functions.

Getting Started

Step 1: Depending on your operating system, download (**lab10_win.zip** or **lab10_mac.zip**) the starting code from Brightspace and extract the zipped file.

Step 2: Double click on lab10.code-workspace

Step 3: lab10_xxx folder contains the following files:

- singly_linked_list.c contains three fully implemented functions: `push`, `length` and `print_list`. This file also contains incomplete definitions of the five functions you have to design and implement.
- singly_linked_list.h contains the declaration for the nodes in a singly-linked list (see the `typedef` for `intnode_t`) and prototypes for functions that operate on this linked list. **Do not modify singly_linked_list.h.**
- main.c contains a simple *test harness* that exercises the functions in singly_linked_list.c. Unlike the test harnesses provided in previous labs, this one does not use the sput framework. The harness does not compare the actual and expected results of each test and keep track of the number of tests that pass and fail. Instead, the expected and actual results are displayed on the console, and you have to review this output to determine if the functions are correct. **Do not modify `main()` or any of the test functions.**

Step 4: Run the project. It should build without any compilation or linking errors.

Step 5: Execute the project. The test harness will show that functions `count`, `max_value`, `value_index`, `extend` and `pop` do not produce correct results (look at the output printed in the console window and, for each test case, compare the expected and actual results). This is what we'd expect, because you haven't started working on the functions that the test harness tests.

Step 6: Review the functions `push`, `length`, and `print_list` we covered during the lectures. Trace and visualize the code step y step. Make sure you understand the algorithm for traversing a linked-list's nodes. Make sure you understand why the loop condition in `length` is slightly different from the one in `print_list`.

Step 7: Open singly_linked_list.c in the editor and do Exercises 1 through 5. If you become "stuck" while working on the exercises, consider using C Tutor to help you discover the problems in your solution.

Exercise 1

File singly_linked_list.c contains an incomplete definition of a function named `count`. The function prototype is:

```
int count(intnode_t *head, int target);
```

Parameter `head` points to the first node in a linked list.

This function counts the number of nodes that contain an integer equal to `target` and returns that number.

This function should return 0 if the list is empty (parameter `head` is `NULL`).

Finish the implementation of `count`. Use the console output to help you identify and correct any flaws. Verify that `count` passes all of its tests before you start Exercise 2.

Exercise 2

File singly_linked_list.c contains an incomplete definition of a function named `max_value`. The function prototype is:

```
int max_value (intnode_t *head);
```

Parameter `head` points to the first node in a linked list.

This function returns the largest number stored in the linked list. It should terminate (via `assert`) if the list is empty (parameter `head` is `NULL`).

Finish the implementation of `max_value`. Use the console output to help you identify and correct any flaws. Verify that `max_value` passes all of its tests before you start Exercise 3.

Exercise 3

File singly_linked_list.c contains an incomplete definition of a function named `value_index`. The function prototype is:

```
int value_index(intnode_t *head, int target);
```

Parameter `head` points to the first node in a linked list.

This function returns the index (position) of the first node in the list that contains an integer equal to `target`. The function uses the numbering convention that the first node is at index 0, the second node is at index 1, and so on. The function should return -1 if the list is empty (parameter `head` is `NULL`) or if `target` is not in the list.

Finish the implementation of `value_index`. Use the console output to help you identify and correct any flaws. Verify that `value_index` passes all of its tests before you start Exercise 4.

Exercise 4

File singly_linked_list.c contains an incomplete definition of a function named `extend`. The function prototype is:

```
void extend(intnode_t *head, int *other);
```

Parameters `head` and `other` point to the first nodes in two distinct linked lists. (In other words, `head` and `other` don't point to the same linked list.)

The function extends the linked list pointed to by `head` so that it contains *copies* of the nodes stored in the linked list pointed to by `other`.

The function terminates (via `assert`) if the linked list pointed to by `head` is empty.

Finish the implementation of `extend`.

Note 1: Something along the lines of: `tail->next = other;`

where `tail` points to the last node in the list pointed to by `head`, is **not** correct. This simply "glues" the first node of the linked list pointed to by `other`, to the tail node of the linked list pointed to by `head`.

Note 2: Your `extend` function must not call the `append` function that was presented in lectures. This would be inefficient, because `append` would be called once for every node that is appended to the list pointed to by `head`, and `append` traverses the entire list every time it is called. Hint: an efficient solution requires exactly one traversal of each of the two lists.

Note 3: Your `extend` function is permitted to call the `push` function that is defined in singly_linked_list.c.

Use the console output to help you identify and correct any flaws. Verify that `extend` passes all of its tests before you start Exercise 5.

Exercise 5

You are going to implement a function that is the inverse of `push`: it retrieves the value stored in a linked-list's head node and removes that node from the linked list.

File singly_linked_list.c contains an incomplete definition of a function named `pop`. The function prototype is:

```
intnode_t *pop(intnode_t *head, int *popped_value);
```

Parameter `head` points to the first node in a linked list.

This function copies the value in the list's head node to the location pointed to by parameter `popped_value` and returns a pointer to the first node in the modified list. (Hint: remember that `pop` must deallocate the head node correctly, because all nodes were allocated from the heap.) The function should terminate via `assert` if the linked list is empty.

Finish the implementation of `pop`. Use the console output to help you identify and correct any flaws. Verify that `pop` passes all of its tests.

Wrap-up

Submit singly_linked_list.c to Brightspace.

Before submitting your lab work:

- Review *Important Considerations When Submitting Files to Brightspace* (lab3 file). Remember: submit early, submit often.
- Make sure that singly_linked_list.c has been formatted to use K&R style or BSD/Allman style, as explained in *General Requirements*.
- Ensure you are submitting the file that contains your solutions, and not the unmodified file you downloaded from Brightspace!

You are permitted to make changes to your solutions and resubmit the files as many times as you want, up to the deadline. You need to demo your work to TAs during the lab for grades.

Homework Exercise - Visualizing Program Execution

You are expected to be able to understand and draw diagrams that depict the execution of short C functions that manipulate linked lists, using the same notation as C Tutor. This exercise is intended to help you develop your code tracing/visualization skills.

If you didn't use C Tutor to help you implement the solutions to the exercises, use the tool to visualize the execution of your `count`, `max_value`, `value_index`, `extend` and `pop` functions. For each function:

1. Click on this link to open C Tutor.
2. Copy/paste the `intnode_t` declaration from singly_linked_list.h and your solution to one of the exercises (the function definition) into the C Tutor editor. If required, copy the definitions of `push`, `length` and `print_list` into C Tutor. You can comment out the `assert` calls.
3. Write a short `main` function that exercises your function.
4. *Without using C Tutor,* trace the execution of your program. Draw memory diagrams that depict the program's activation frames just before your function returns. Use the same notation as C Tutor.
5. Use C Tutor to trace your program one statement at a time, stopping just before your function returns. Compare your diagrams to the visualizations displayed by C Tutor. If C Tutor complains that your program is too long, delete the comments above the function definitions.
6. Do not submit your solutions for this exercise.