CX 4010 / CSE 6010
Assignment 2
Software Modules
Additional requirements for graduate students only
Minor clarifications since original posting

Initial Submission Due Date: 11:59pm on Thursday, September 15
Peer Reviews Due Date: 11:59pm on Thursday, September 22
Final Submission Due Date: 11:59pm on Thursday, September 29
Submit codes as a single zipfile as described herein to Canvas
Submit peer reviews through Canvas
48-hour grace period applies to all deadlines

This assignment consists of writing a program to implement a program module for a list using a **singly-linked-list** data structure. The list code should be structured as if it were part of a software library to be used with other programs written by someone else. You should assume that the data values are of type `integer` and that only positive integers will be entered. Graduate students will be required to maintain the list sorted from the smallest to the largest value.

The software should implement the following interface.

1. Use a `typedef` to define a data structure called `Node`, defined as a C `struct` representing a list item. It should include the data value and a pointer to the next node in the list. The implementation of the list data structure is not visible outside the module you are creating.

2. Write a function with the prototype

   ```
   Node *Initialize();
   ```

   Create a new node to set up the initial list and return a pointer to it. The list is initially empty—that is, it contains no elements. Return `NULL` if the operation fails, e.g., due to a lack of memory.

3. Write a function with the prototype

   ```
   int Insert(Node *listhead, int data);
   ```

   Insert the item of type `int` stored in `data` into the list that begins with `listhead`. Verify that the item is positive (you may assume it is an integer) before inserting it into the list. Return 0 if the operation succeeds, or 1 if it fails (including if a non-positive value was entered, in which case, also print a suitable error message to the screen). Graduate students: ensure the item is inserted in sorted order.

4. Write a function with the prototype

   ```
   Node *Search(Node *listhead, int data);
   ```

   Search for the value `data` within the list that begins with `listhead`. If the data value is not

found in the list, print a suitable error message. The function returns a pointer to the node with data field `data` or `NULL` if the data item is not present.

5. Write a function with the prototype

   ```
   int Delete(Node *listhead, int data);
   ```

   Remove the node with data field `data` from the list `listhead` if it is in the list. If the data value is not found in the list, print a suitable error message. Return 0 if the operation succeeds, or 1 if it fails, including if the data value is not in the list.

6. Write a function with the prototype

   ```
   int Minimum(Node *listhead);
   ```

   Return the minimum value of all the nodes in the list that begins with `listhead`, or 0 if the list is empty.

7. Write a function with the prototype

   ```
   int Maximum(Node *listhead);
   ```

   Return the maximum value of all the nodes in the list that begins with `listhead`, or 0 if the list is empty.

8. Write a function with the prototype

   ```
   int Predecessor(Node *listhead, int data);
   ```

   Return the data value of the node that precedes the node with the specified data value in the list that begins with `listhead`. If the list is empty, the specified data value is not found, or the specified data value is the first node in the list, print a suitable error message and return 0.

9. Write a function with the prototype

   ```
   int Successor(Node *listhead, int data);
   ```

   Return the data value of the node that follows the node with the specified data value in the list that begins with `listhead`. If the list is empty, the specified data value is not found, or the specified data value is the last node in the list, print a suitable error message and return 0.

10. Write a function with the prototype

    ```
    unsigned int Length(Node *listhead);
    ```

Return the number of items currently in the list that begins with `listhead` (0 if the list is empty).

11. Write a function with the prototype

    ```
    void Print(Node *listhead);
    ```

    Print the data items stored in the list that begins with `listhead` in order from first to last, without changing the contents of the list. Print a suitable message if the list is empty.

12. Write a function with the prototype

    ```
    void Finalize(Node *listhead);
    ```

    Delete the list that begins with `listhead` by releasing all memory used by all the contents of the data structure.

You should define all these items using node.h and node.c files. Then, you will write a main file that uses the auxiliary files to implement your list. Your main file should call `Initialize()` and `Finalize()` appropriately and in between use a simple menu (using your choice of characters or integers to specify each function) to allow users to interact with the list by choosing to apply any of the seven (nine) other functions.

Test your program and ensure that it can properly handle the following "error" cases.

- Inserting a non-positive number.
- Searching for a value that is not in the list.
- Searching for a value in an empty list.
- Deleting a value that is not in the list.
- Deleting a value from an empty list.
- Obtaining the maximum value of an empty list.
- Obtaining the minimum value of an empty list.
- Obtaining the predecessor of an empty list.
- Obtaining the predecessor of a value that is not in the list.
- Obtaining the predecessor of the first value in the list.
- Obtaining the successor of an empty list.
- Obtaining the successor of a value that is not in the list.
- Obtaining the successor of the last value in the list.
- Obtaining the length of an empty list.
- Printing an empty list.

Develop a "test script" for your program that demonstrates how your list performs in a variety of non-trivial circumstances, including some of the "error" cases above. Use the `Length()` and/or `Print()` functions to verify the operations work correctly. Although you should test that your program can handle many cases, you do not need to demonstrate all of them in the test output you will submit on your submission slides because you are limited to only 2 slides.

Your code should include the following files:

- **node.h:** Definition of the interface to the software that includes only the information necessary to use the module, including the function prototypes. It must *not* specify the internal implementation of the code.
- **node.c**: Implementations of the functions whose prototypes are given in node.h.
- **main.c:** Main program, including the menu interface for the user with calls to the functions.

Because your program will be written with the interface as defined, you should expect that your node.h and node.c files could be used with a main.c file written by someone else.

Moreover, your code (final submission) should successfully compile (using gcc) and run on the COC-ICE cluster. This does not mean we necessarily will try to run your code on the cluster, but if we have any difficulties compiling and running it, we will move it to that environment for further testing. If you are compiling and running your codes under linux, most likely you will not need to make any changes as a result of this requirement. More information about logging in to COC-ICE will be available soon.

You should submit to Canvas a single zipfile that is named according to your Georgia Tech login—the part that precedes @gatech.edu in your GT email address. To receive full credit, your code must be well structured and documented so that it is easy to understand. Be sure to include comments that explain your code statements and structure.

The zipfile should include the following files:

(1) your code (all .c and .h files). If you are using linux or Mac OS, you may use a makefile to compile and run your program, and you should include it if so. A sample makefile will be provided.

(2) a README text file (not formatted in a word processor, for example) that includes the compiler and operating system you used for compiling and running your code along with instructions on how to compile and run your program.

(3) a series of 4 slides composed in PowerPoint or similar software, saved either in PowerPoint or as a PDF and named slides.pptx or slides.pdf, structured as follows:

- Slide 1: your name and a brief explanation of how you developed/structured your program. This should not be a recitation of material included in this assignment document but should focus on the main structural and functional elements of your program (e.g., the purpose of any loops you used, the purpose of any if statements you used to change the flow of the program, the purpose of any functions you created, etc.). You are limited to one slide.
- Slides 2-3: a description of your most important tests and sample output produced by the test program. You are limited to two slides; we understand you will not be able to demonstrate all functionality here, so focus on what you think is most important.
- Slide 4: a brief exposition of what you found (or did not find) useful about the peer review process.

Initial submission

Your initial submission does not need to include the slides. It will not be graded for correctness or even tested (including for compatibility with gcc on the COC-ICE cluster) but rather will be graded based on the appearance of a good-faith effort to complete the majority of the assignment. However, you should be aware that **your peers will review your initial submission**.

Peer review assignment

In this assignment, you will have two weeks between the initial and final submission to allow you the opportunity to review others' submissions and provide feedback. Although the feedback you receive from your peers should be useful, we also expect that having the opportunity to view others' code will be a valuable experience for you as well. You will submit your final submission after the peer reviews have been completed so that you can benefit from the process. However, while you may find yourself inspired by what others have done, you should **be careful not to plagiarize from others' code**. **You should write and submit your assigned peer reviews while taking a break from looking at your code and you should not look at others' code again afterward.**

Peer review assignments will be made in Canvas after all initial assignments have been submitted. (peer review assignments should be made by Monday, September 19). **If you do not submit an initial assignment on time, you may not be able to complete the peer review assignment either.**

You will be assigned **three peer reviews**. Your peer review does not have to be long but it should address the following 5 topics. **Use the Comment box to submit your peer review.**

- Program execution: does the program compile and execute without error, and is the output clear?
- Program structure and implementation: is the structure of the code clear and easy to follow and are the implementation choices good?
- Program style and documentation: are the comments in the program useful and at the appropriate level, is the program easy to follow visually through indenting and blank lines, and does the program avoid copying the same code in multiple places (redundancy)?
- Overall strengths of the code.
- Any constructive suggestions for improvement.

Because of Canvas limitations, grades for your peer reviews will be reported in a separate peer review assignment. (You will not need to do anything extra for the official Canvas assignment, but that is how the grades will appear.) For information on how to use Peer Reviews in Canvas, including where to submit your peer reviews and where to find peer reviews of your submission, please see the following videos (or other similar ones you may find).

https://www.youtube.com/watch?v=cKyvKKq0mtc
https://community.canvaslms.com/t5/Video-Guide/Feedback-Overview-Students/ta-p/383514

Some hints:

- Start early!
- Identify a logical sequence for implementing the modules. Then implement one at a time, and verify each works properly before proceeding to implement the next module.