Simon Fraser University
School of Computing Science
CMPT 300: Assignment #1

List Implementation

This assignment is designed to get you into the swing of things with respect to C and UNIX. You can develop on any Linux/Unix based system, but be sure that it compiles correctly on the CSIL Linux computers using the `gcc` compiler (details on course website).

Allowances

You are expected to do this assignment **alone** and are not allowed to share your code or look at the code of someone else. We will be carefully analyzing the code submitted to look for signs of plagiarism so please don't do it! If you are unsure about what is allowed please come talk to myself or the TA. More notes:

- You may follow any guides you like online as long as it is providing you information on how to solve the problem rather than a solution to the assignment.

- If receiving help from someone, they must be helping you debug **your** code, not sharing their code or writing it for you.

- You may not resubmit code you have previously submitted for any course or offering.

- The instructor or TAs may conduct followup interviews with students to discuss their implementation and design in order to verify the originality of the assignment.

What to do

For this assignment you are going to implement the `List` abstract data type. The list data structure is widely used throughout operating system programming. Although you should all be experts at list manipulation, it will serve to refresh your list skills and get you back onto UNIX and into C programming. Also, the routines you implement here will hopefully be useful in your subsequent assignments.

- Each list element (node) is able to hold one item.

- An item is any C data type that can be pointed to - so your node structure should have a `(void *)` field in it to reference the item held by that node.

- Every list has the notion of a current item, which can refer to any item in the list. The current pointer may also point beyond the end or before the beginning of this list. If the current pointer is before or beyond the list, a routine returning its value will return a `NULL` pointer.

You will have to create the user-defined type `List`. An instance of this type refers to a particular list and will be an argument to most of your list manipulation routines. As with all code that is written for operating systems, the goal here is efficiency. You should temper implementation efficiency with a significant dose of code elegance (though hopefully one can be accomplished without compromising the other).

- The implementation must use **statically allocated arrays** for list nodes and list heads.

  - If the nodes are exhausted then trying to add an item to a list fails.

  - If the heads are exhausted then trying to create a new list fails.

  - Removing an item from a list frees the node which held the item, making the node available for future use.

  - Freeing a list destroys the list, making its head available for future use.

- In the interest of efficiency you must not use any "searches" to find free nodes or heads.

You must create a test program that adequately exercises each of the functions in the given `list.h` enough to give confidence that your implementation is correct.

You must create `list.h` and `list.c`. A template for `list.h` can be found on the course website. You may modify this file as needed (such as the data types defined); however, you must not alter the function prototypes, or the name of the `#define` constants, because the marking test code will depend heavily on this part being unmodified. Carefully review the documentation for each function in `list.h` as it specifies what you must implement.

Important Constraints

- When creating or deleting a new list, you must not search through heads. When creating or deleting a node in a list, you must not search through nodes.

- When advancing forward or backward through a list, you must not search through the list. You are allowed to do a one-time non-constant time set-up / initialization of your data structure(s) the very first time the user calls `List_create()`

- C is loosely typed: when returning a `void*` the compiler may not provide full type safety. Ensure that all your functions which return pointers to items (which are `void*`) are doing just that: returning a pointer to an item stored in a node, not a pointer to a node.

- You may not use dynamic memory allocation (`malloc()`). Your `.c` file creates a single array of `Node` structs to serve as the pool of nodes shared by all lists. Likewise, create a single array of list heads (`List` structs) to be the pool of lists that the module can support. Your code will link these `List` and `Node` structs, via the pointers they store, to create an efficient linked list module.

- Your `.c` file may include additional "private" (internally linked `static`) functions. The `.h` file must not expose any "private" functions. (i.e., your `.h` file must only expose those functions listed in the provided `list.h`).

- Any global variables must be `static` (internally linked).

What to Hand in

Submit to CourSys (https://courses.cs.sfu.ca/) a ZIP file of your project folder including:

1. `list.h`
2. `list.c`
3. Your own C program which uses and exercises your `List` data type.
4. A `makefile` to build your project. Must support the following commands
   "`make`": builds the program (but not run it);
   "`make clean`": erases all build products (i.e., your executable and `.o` files).

We will add to your project a test program to mark the correctness of your list implementation. If you have changed the file names, or constant names / function prototypes in the `list.h` file then our test code will not execute and you will likely receive a very low grade.

All submissions will automatically be compared to each other, and to similar assignments given in previous semesters for unexplainable similar submissions. Please make sure you do your own original work.