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. Be sure that this compiles on the Linux computers in the CSIL lab in room SRYE 4024 using the gcc compiler. You can access the Linux server remotely using ssh to one of the *csil-cpu* servers. Details can be found here:

http://www.sfu.ca/computing/about/support/csil/how-to-remote-access-to-csil.html

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.

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

You are to implement the following list manipulation routines:

- `LIST *ListCreate()` makes a new, empty list, and returns its reference on success. Returns a NULL pointer on failure.
- `int ListCount(list)` returns the number of items in list.

- `void *ListFirst(list)` returns a pointer to the first item in list and makes the first item the current item.
- `void *ListLast(list)` returns a pointer to the last item in list and makes the last item the current one.
- `void *ListNext(list)` advances list's current item by one, and returns a pointer to the new current item. If this operation advances the current item beyond the end of the list, a NULL pointer is returned.
- `void *ListPrev(list)` backs up list's current item by one, and returns a pointer to the new current item. If this operation backs up the current item beyond the start of the list, a NULL pointer is returned.
- `void *ListCurr(list)` returns a pointer to the current item in list.
- `int ListAdd(list, item)` adds the new item to list directly after the current item, and makes item the current item. If the current pointer is before the start of the list, the item is added at the start. If the current pointer is beyond the end of the list, the item is added at the end. Returns 0 on success, -1 on failure.
- `int ListInsert(list, item)` adds item to list directly before the current item, and makes the new item the current one. If the current pointer is before the start of the list, the item is added at the start. If the current pointer is beyond the end of the list, the item is added at the end. Returns 0 on success, -1 on failure.
- `int ListAppend(list, item)` adds item to the end of list, and makes the new item the current one. Returns 0 on success, -1 on failure.
- `int ListPrepend(list, item)` adds item to the front of list, and makes the new item the current one. Returns 0 on success, -1 on failure.
- `void *ListRemove(list)` Return current item and take it out of list. Make the next item the current one.
- `void ListConcat(list1, list2)` adds list2 to the end of list1. The current pointer is set to the current pointer of list1. List2 no longer exists after the operation.
- `void ListFree(list, itemFree)` delete list. itemFree is a pointer to a routine that frees an item. It should be invoked (within ListFree) as: `(*itemFree)(itemToBeFreed);`
- `void *ListTrim(list)` Return last item and take it out of list. Make the new last item the current one.
- `void *ListSearch(list, comparator, comparisonArg)` searches list starting at the current item until the end is reached or a match is found. In this context, a match is determined by the comparator parameter. This parameter is a pointer to a routine that takes as its first argument an item pointer, and as its second argument comparisonArg. Comparator returns 0 if the item and comparisonArg don't match, or 1 if they do. Exactly what constitutes a match is up to the implementor of comparator. If a match is found, the current pointer is left at the matched item and the pointer to that item is returned. If no match is found, the current pointer is left beyond the end of the list and a NULL pointer is returned.

Make sure to submit all relevant source files (.h and .c files for the list implementation and separate .c file for the test driver) for your assignment, as well as the makefile(s).