CIS361 System Programming

A Self-Organizing List

(Due March 23, 2017)

Many problems using lists have the characteristic that tests to determine if a value is present in a list occur much more frequently than insertions or deletions. Furthermore, it is also often the case that once such a test has been performed, the likelihood is increased that the same element will be subsequently required. In such situations that linear running time of the typical search operation can be improved by always moving an element to the front of the list on a successful search. A data structure that tries to improve future performance based on current usage is said to be self-organizing.

In this project, you will design and implement a self-organizing list, including a data structure and its associated operations. A unique operation for a self-organizing list is search. Instead of simply looping over the elements until we exhaust the list or find the requested value, the search operation will move the value on a successful search to the front of the list. As such, a subsequent search for the requested value will be performed very quickly. Note that the task of moving the element to the front of the list can be accomplished with assistance of some other operations on the list. You need to determine what operations a self-organizing list should provide.

Also you will use your self-organizing list to build a program that will scan a C source file, recognize all identifiers and count their occurrences, as well as output those identifiers along with their numbers of occurrences in the source file.

In a programming language, an identifier is a sequence of one or more letters, digits, and underscores, but it cannot begin with a digit. Here, for simplicity, you may consider a keyword in C, like include, as an identifier; however, words in a program comment are not nor those in a string constant (i.e., anything that appears between double quotes).

Use a linked list to store a sequence of nodes. Each node holds an identifier and its number of occurrences. When an identifier is inserted, check and see if it is already in the list. If it is, update the number of occurrences for that identifier. Otherwise, create a new node.

Design your self-organizing list as an abstract data type (ADT), meaning that the client code can use the list without having to know the detail of its implementation. According to the “separation of concerns” principle, create several files to organize processing elements that serve for different purposes in your program, such as:

- list.h contains type definitions and function prototypes for a self-organizing list
- list.c contains function definitions for a self-organizing list
- proj2.c contains main() and other functions you have introduced

Your proj2.c has to include list.h that provides what your program needs to use (like the interface of a class in an object-oriented program) and the related .c file hides what it does not need to know (like the implementation of a class). Since you will introduce a number of functions as needed in this program, you

should give a meaningful name to each of them and provide comments to help the reader understand its functionality. Look at the example in Lecture Notes 5 – Separate Compilation (a PPT that can be found in the instructor's website) for how to organize elements of a data structure as separate files.

In addition, use command-line arguments to pass the input and output filenames into your program. A sample C source file, sample.c, can be found in the "Proj 2 - W17" folder on the instructor's Web site. Use that file as the input file when you test your program. When your program works correctly, get a hardcopy of the source code, the makefile, and program output for submission.