

## Assignment 4: Symbol Table

### 1. Introduction

In this assignment you will write a fifth implementation of the storage system used by the word frequency program shown in lecture. The main part of the program and shell scripts to feed it input are provided for you. These other parts will call on your functions to perform certain tasks.

This assignment will give you practice using structs, arrays, and pointers in C. You will also need to use some important system subroutines to perform string and memory allocation functions.

You will need to access a number of files we have prepared for you to use in this assignment. They all reside in the directory `~lib113/hw/wf`. You can link to them by typing

```
% ln -s ~lib113/hw/wf/* .      <-- there is a dot at the end
```

This gives you access to the files, but you will not be able to modify them. For information about what links are and how they work, refer to the Unix book.

### 2. The Unix Way to Get a Word Frequency Table

Consider the database of firstyear students in Harvard Yard. You might want to know the number of people in each dorm for a given year. A simple pipeline:

```
grep "CL=92" freshmen | cut -d: -f4 | cut -d= -f2
```

will generate a list of dorm names, one for each member of the class of 1992. What dorm names are there, and how many times does each name appear? The list looks something like:

```
straus
holworthy
grays
grays
thayer
grays
..etc..
```

We want a list that looks like

```
125 grays
23 greenough
160 holworthy
..etc..
```

That is, we want a list of the words that appear, we want the number of times each word appears in the input (the *frequency*), and we want the list sorted in alphabetical order.

The Unix tools `sort` and `uniq` are often used to produce this frequency table. Try this:

```
grep "CL=92" freshmen | cut -d: -f4 | cut -d= -f2 | sort | uniq -c
```

The `sort` command reads in all the input lines and outputs them in alphabetical order, and the `uniq -c` command replaces each sequence of repeated lines with a count and the line.

This method has two problems. The first problem is that the input may not be split into one word per line. The second problem is that sorting takes time. A long list of words can take a long time to sort, but the sorting is only being done to prepare the input for `uniq -c`.

### 3. A Proposal for a New Tool : `wordfreq`

The solution to these problems is to write a new C program, one that reads standard input, splits the input into words, counts the words, and prints an alphabetical list of all the words with their frequencies. That is, it produces the same result as sorting then `uniq -c`'ing, but without the time required to sort all the input.

#### 4. Four Solutions So Far

In lecture, we looked at four implementations of the wordfreq filing system. All the programs implement a storage system that holds words and associated integers. A table that associates values with strings is called a *symbol table*. The programs provide functions to insert new symbols into the table, retrieve data from the table, update values in the table, and to cursor through the table to see what is there.

All four solutions work but, but most have limitations or wasted memory. The fourth solution uses memory efficiently and has no artificial limitations. It has one problem: it was the slowest by far.

Your project is to write a fifth version of wlfiler, one uses linked lists but runs much faster than version 4.

#### 5. Speeding Up the Linked List Version

Version 4 uses a single, unsorted linked list to store all the 'rows' in the table. Any search for a word must traverse this list. The list is unsorted, so the program has to go all the way to the end to make sure a word is not on the list.

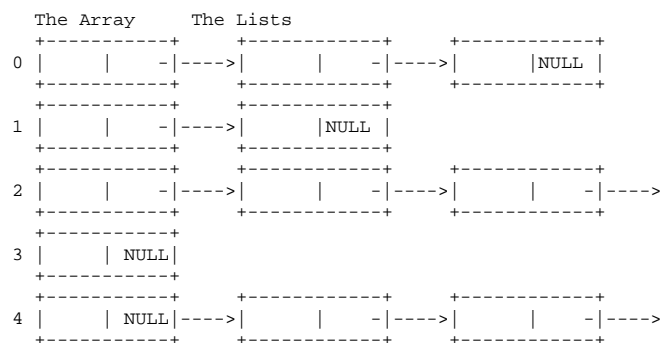
Your faster version will run about 50 times faster by making the following changes to the storage:

1. The single linked list will be replaced with 26 linked lists, one for each letter of the alphabet. All the words that start with the letter 'a' will be on the first list, all the words that start with 'b' will be on the second list, etc. This change will reduce the search time by, on average, a factor of 26.
2. Each of those linked lists will be kept in alphabetical order. By doing so, you can write code that will know when to stop looking for a word without going to the end of the list. On average, you will stop searching halfway through the list, further reducing your search time by a factor of 2.

Details of the structures and code follow.

#### 6. An Array of 26 Linked Lists

What does an array of linked lists look like? A diagram is:



The entries in the table are *structs* that contain the word, the frequency, and a pointer to the next item in the list. If there is no next item in that list, the pointer has the value NULL.

The lists of items are organized into an array. The array, seen at the left of the diagram, is an array of structs. Each struct is the head of the list. It does *not* contain a word. The `next` member of that struct points to the list. If there are no words on that list, that head struct contains a NULL pointer.

#### 7. The Data Structures to Implement the Symbol Table

How does this diagram translate into C code? Since there are two things in the picture, structs and the array, there must be two definitions:

The table consists of words and their frequencies. Each item in the table, then, consists of a string and an integer. The items are strung together in a linked list. The links in the list are therefore structs that look like:

```

struct link
{
    char      *word;    /* a ptr to the word, in malloc()ed memory */
    int       value;    /* an integer value; the frequency          */
    struct link *next;  /* points to next in list          */
} ;
typedef struct link LINK; /* LINK is shorthand for 'struct link' */

```

Having thus defined the data type, the array of pointers is created with:

```
LINK table[ARRAY_SIZE]; /* an array of head structs */
```

This statement defines an array of pointers to things of type LINK. There are ARRAY\_SIZE structs in the array.

## 8. Code to Implement the Symbol Table

The statement that defines the array of ARRAY\_SIZE structs creates the array of head structs, but does not create any symbols.

You only need to create storage for words when the main program asks your package to insert a new symbol in the table.

The library subroutine *malloc()* gets memory from the operating system and tells you where that memory is. You tell it how much you need. Typical code looks like:

```

LINK *lp;          /* plan to hold address in lp */

lp = (LINK *) malloc( sizeof(LINK) ) ;
if ( lp == NULL )
    error-condition-code;

```

This asks for a chunk of memory large enough to hold a link struct. The pointer lp contains the address of this chunk of memory.

You will then need memory for the word that will be stored in that link.

```

char *cp;          /* plan to hold address in cp */

cp = malloc( strlen(the_name) + 1 );          /* room for null */
if ( cp == NULL )
    error-condition-here;
strcpy(cp, the_name);                          /* store name */

```

You then need to put the pointer to the name in the symbol:

```
sp->name = cp ;
```

The last part, figuring out where the symbol goes, and inserting it in its list is discussed below.

## 9. The Assignment

The symbol table manager you will write will have a symbol table that consists of 26 lists, one for each letter of the alphabet. A symbol will be assigned to a list based on its initial letter. Each of these 26 lists will be kept in alphabetical order.

### 9.1. Functionality

Your program must implement all the functions in the sample programs (wlfiler1.c...) from lecture. The arguments, action, and return values for those functions are all documented in the code for wlfiler1.c.

Your program must implement one additional function:

```
word_delete(char str[])
```

This function is passed a string, and the function removes from the table the item containing that string. Your function must call the system function `free()` to recycle the memory used by the struct and by the string.

Your functions may assume that all the strings it is given begin with a lower case letter. Your functions do not have to check for valid data.

Your program will be linked with the program `wlmain.c` from lecture and must work with test data we shall provide.

The program will also be linked with another test program, one that will exercise its delete function.

## 9.2. Getting Started

Make sure you understand the operation of the sample programs from class and the purpose of the functions. You may notice as you study those programs that many of the functions are similar.

For your next step, think about how to design a better system. Good programming style leads one to eliminate duplicate code by calling a common helper function. For example, the `in_table()`, `lookup()`, and `update()` functions all do similar things. Why not have a single function to search the table for a string, and return a pointer to the struct? The caller can then decide what to do with that struct.

Finally, think through how to implement each of the operations in this new data structure. Use the fact that the lists are sorted to speed up searches. That is, do not search further than you need to.

## 9.3. Creating and Testing wordfreq

You need to include "wl.h" in your file or files. You need to compile `wlmain.c` with your program `wlfiler5.c` and build `wl5`

```
% cc wlmain.c wlfiler5.c -o wl5
```

To see if your program works, run the following shell script:

```
% ~lib113/hw/wf/wf.test
```

## 9.4. To Hand In

**Clean Code** Part of your grade will be based on your code passing `gcc -Wall` with no warnings or errors. You will lose two points for *each* warning that gcc reports about your program.

**typescript** Submit a typescript of your program being compiled with gcc and being tested by the test script.

**Digital Handin** From the directory containing your work, type

```
~lib113/handin wf
```

## 9.5. Extra Credit

What if you wanted to write a program that used two symbol tables? The `wlfilerX.c` series presents an interface to a single symbol table. An object-oriented approach would be more flexible. For ten points of extra credit, write a new file called `wlfiler6.c` that provides the following enhancements:

(a) Define a data type called a `syntab`, (b) modify `init_table()` to be a constructor, returning a pointer to a new, initialized `syntab`, (c) modify all the other functions so they accept as an additional argument a pointer to a `syntab`. For example, `in_table(syntab *tp, char word[])` would search the table pointed to by `tp` for the string stored in `word`.