Lab2: Dictionary Array – an efficient implementation

Due Date: Monday, Feb 7 2010 by midnight

Background:

In Lab1, we learned how to use a static 2D array of characters of max size around 172,000x32. The space required to store this file is around 5MB, definitely a waste of memory when most files are smaller than 172,000 words. We did assign 32 bytes for each word but most words are less than 32 chars long (average word length in English is around 7). In Lab2, we will implement the same dictionary structure, but via a dynamically allocated array of dynamically allocated char pointers. Moreover, in lab1, we sorted *as we are inserting* to the array. In this lab, we will **load the array first** and **then sort the array** later using insertion sort. To **improve the run time** we will use techniques such as binary search to find the place to insert. Finally we will free all memory and check with the unix utility valgrind. Valgrind will allow us to check for "memory leaks", that is memory that is allocated but not freed. You will apply all that you have learned so far about arrays, pointers, passing pointers, malloc and free in this lab.

WARNING: This is the first time you will be exposed to dealing with dynamic pointers. You must start this program early to make sure you can take care of all the debugging issues early on.

What You'll Need

This program is divided into 3 files, dictlib.h, dictlib.c, dictlibtester.c and main.c. Download all code from /afs/andrew/course/15/123/downloads/lab2. This site contains the most updated code. Here are the files you will find.

- **Dictlib.h** is the header file that contains function prototypes that will be used by main.c.
- In **dictlib.c** we implement the functions declared in dictlib.h.
- **dictlibtester.c** is very important. This tester will do "stress" testing on all your functions. Your functions must "pass" all our tests to receive credit.
- In the **main program**, you will be calling dictlib functions to make things happen. But since we have tested all the dictlib functions before, this part should be an easy one.

dictlib.h

This is the main header file that contains all function prototype definitions. You need to include this file in all source code. DO NOT change any prototypes. We will justify why they are written the way they are.

dictlib.c

This is the implementation file where you develop code for each function separately. Be sure to test the functions as you develop them. Do not wait till the end to test things. Basic rule is do easy ones first. Say for example, you test if your binary search covers all cases. We are beginning to get stricter about the correctness of your code. So component testing is important.

main.c

This is the main function that reads a file, store in an array, sort it using insertion sort and then write it to a file that is sorted.

dictlibtester.c

This is the tester for dictlib.c. In order to use it, compile and run as follows.

- gcc –std=c99 dictlib.c dictlibtester.c
- ./a.out

You must pass all assertions to get full credit.

Assignment:

PART 1: You need to develop the following functions for part I. insertWord, resize, binarySearch and sortArray. The prototypes are given as:

```
int insertWord(char **, char*, int);
int resize(char ***, int* );
int sortArray(char **, int );
int binarysearch(char** , char* , int , int );
```

PART 2: You need to develop the following functions for part II. freeAll, and printArray. Here are the function prototypes for part. See starter code for more details.

```
void freeAll(char **, int );
```

void printArray(FILE* , char **, int);

PART 3: Now that you have thoroughly tested your dictlib, now try it with the main program. Everything should work as expected. To run the main program, copy the tested dictlib.c to PART 3 folder and run as:

- gcc –ansi –pedantic –Wall –std=c99 dictlib.c main.c
- time ./a.out infile outfile

Other requirements for Lab 2

- Do not use any additional data structures in your attempt to be clever/fast on the load.
- You should document your code as best you can. You should **especially** document anything that is "clever" or unusual.
- **Do not use strcpy to shuffle the words during insertion sort. Instead, copy the pointers!**
- You must NEVER create any garbage. You need to pass the valgrind test
- You will note that most function prototypes take addresses of variables from the calling program and manipulate the content directly. As such you have to deal with many * (a pointer), ** (a pointer to a pointer OR array of pointers)
- It is important to learn how to dereference various *'s. For example, dereferencing an int* leads to an int, dereferencing int** leads to an int*
- Be sure to come to class so you can learn all about *, **, and ***'s
- ---

Binary Search

In this lab, we will need binary search to speed up the insertion sort. Binary Search assumes that your array is sorted. The algorithm works as follows.

- Find the mid of the array using : low + (high-low)/2
- Suppose you are searching for the target in array A. Compare target to A[mid]
- If (target > A[mid]) search the array A[mid+1,high]
- If (target < A[mid]) search the array A[low, mid-1]
- If target is found or low > high stop the search

Caution: only 10% of the programmers know how to correctly implement binary search. Are you one of them?

Insertion Sort

In this lab, we will be loading all the strings into the array and then sort it using insertion sort as in Lab1. The difference in this lab is that we will use binary search to speed up the find process. So instead of linear search to find the

position to insert, we are using binary search. The idea of the insertion sort is as follows.

1. Assume that A[0..i] is sorted for i>=0
2. Now insert the element array[i+1] into the correct position of the array[0..i]
3. Continue this process from i=1 to n-1
4. Once you find the place to insert, you can move the entire memory block using memcpy

Source File Decomposition

This is our first C program where we try to solve the problem by using multiple source (*.c) and header files (*.h). You cannot change the prototypes given in dictlib.h but you can add helper functions as necessary. The main should only have the includes at the top and the main function below. All other function definitions are in a separate .c file and their corresponding prototypes in a separate .h file (which must be protected correctly with an *#ifndef*). Your main program is complete. Your assignment is to complete the dictlib.c functions. Test the functions in dictlib.c with dictlibtester.c and then run the program with main.c. To run the program and see the times, do the following.

- gcc –ansi –pedantic –Wall –std=c99 dictlib.c main.c
- time ./a.out infile outfile

Downloading Starter Code

You can download starter code from

/afs/andrew/course/15/123/downloads/lab2

Finalizing all Code

Once you develop each function, you can include all of the code in the dictlib.c file and compile main.c and dictlib.c as follows. This shows how to compile components of your code separately.

type > gcc -c –Wall –pedantic –ansi –std=c99 main.c

This will create the object file main.o. You are also asking the compiler to list all compiler warnings by using the flag -Wall

To compile dictlib.c

type > gcc -c –Wall –pedantic –ansi –std=c99 dictlib.c

This will create the object file dictlib.o.

Now to create the executable (called exec) you can

>gcc -o exec main.o dictlib.o

>./exec -t -s /afs/andrew/course/15/123/datafiles/filename out.txt

---

Data Files

- 10-words.txt (test file of 10 words - use this first!)
- 105-words.txt (test file of 105 words)
- 42K-words.txt
- 172K-words.txt **DON'T use this one until you are SURE you're done!**

Above listed and all other test data files can be found at /afs/andrew/course/15/123/datafiles (DO NOT COPY THESE to HANDIN)

DO NOT copy any data files. Instead when we run the program we will refer directly to the data file. If you leave any data files or outfiles in the handin folder, you will be penalized 5 points.

Grading your program

See rubric.txt for details

Sample Runtimes

Here are some sample run times for (when sorting with –s –t) flags. You perhaps can improve these times with various techniques (eg: memcpy). Also these times will vary from time to time, as you test this on a time shared machine.

10-words file: < 1 sec
105-words file: < 1 sec
42K-words file: < 3 sec
172K-words file: < 40 sec

Output format

There is no specific format of the output. For each run, you can find the run time. For example,

- time ./a.out infile outfile

must show the time it takes to read and sort a file.
Also output file will be formatted as words separated by a space. Here is part of an output file for 42K-words sorted

aahs aaliis aardvarks aardwolf aargh abaca aback abacterial abaft abalones abased abasements abaser abasers abashment abatable abator abattoirs abaxial abbacies abbacy abbey abbreviations abdication abdicators abdomen abducent abducentes abductio n abductor abductors abducts abed abeles abelias abelmosks aberrance aberrant aber rantly aberrated abettal abeyances abfarad abhorrer abidances abilities abiogenica lly abioses abiosis abjectness abjuration abjurations abjurers ablations ablative ablatively ablaut ablegate abler ables ablest abluted abnormalities aboard abohms abolishable abolisher abolitionary abolitionist abolitions abomasa abominably abor al aborally aborigine aborigines aborning abort aborter abortifacient aborting abo rtions abortive abortivenesses aborts ab……ETC

Memory leaks

You must check for memory leaks using valgrind.

Handing in your Solution

Log files: As in Lab1, you need to submit at least 2 log files to get 3 points. Selflog.txt contains, times recorded by you in the format given in lab1. Other file session.txt contains all data about your session activities. Always type date before starting a session. This will record the current date and time.

- script –a session.txt

The other four files (main.c, dictlib.c, dictlib.h, dictlibtester.c) should be in one folder (DO NOT ZIP). DO NOT SUBMIT ANY DATA FILES or OUTPUT files. If we find any data files in the folder, we may deduct points. To copy your lab2 folder to handin do the following.

>cp –r lab2 /afs/andrew.cmu.edu/course/15/123/handin/lab2/yourid

to submit your zip file.

Due Date

This assignment (all 3 parts tested) is due on or before Monday Feb 7th @ 11:59 Pm. Good Luck.