Homework 6

Due Thursday, April 3, by midnight via Sparky Electronic handin.

Reading

- Lecture slides for classes C11-14.
- Relevant sections in Kernighan & Ritchie, Chapters 1-7, covering the material presented in the classes above. In particular K&R 7.5 & Appendix B

Homework

- **Outline:**
  - In this assignment you will work with structures, malloc/free, command line arguments, and files.
  - You will build an acronym dictionary file decoder. Command line arguments will determine the source of the dictionary file and where the input/output comes from/goes to.
  - Your program will read in an acronym dictionary of abbreviations and phrases from a file into structures.
  - Decoding: It will read in an input text and replace any acronyms from the dictionary with ONE of its matching phrases from the dictionary. The user will be asked to select the phrase to replace it with. Statistics about the replacements and dictionary will be printed to the screen.
  - Sample input files are given on the website with the Homework link.
  - Do each part of the assignment. Make sure that your program functions for at least each part, so you can obtain partial credit for that Part. Place comments at the top of your file specifying which parts are complete and what does not work.

- **Part 1: Opening and Parsing Files with Main arguments**
  - The main function of you program will use the command-line arguments to specify the operations.
  - Your program should accept one of the following flags in any order and are used to specify the input/output streams.
    - `-d` for the dictionary filename, if omitted use default "`AcroDict.txt`"
    - `-i` for input filename, if omitted input comes from stdin (keyboard, newline or enter is termination of input).
    - `-o` for output filename, if omitted use default "output.txt"

    EX: `a.out –o output_file.txt –d my_dict.txt`

    This decodes the string taken from the user keyboard using the dictionary loaded from my_dict.txt and outputs the un-acronymed text to output_file.txt
  - To handle the file arguments may use `fprintf` and `fscanf` and any of the file functions from the `stdio.h` header. Make sure to check and handle error cases (missing file, etc).
  - Begin this homework by first writing a main program that handles the command-line arguments (function `getopt`() is allowed, TAs will go over in recitation on 3/24), opens the specified input type and outputs each word (any number, characters, punctuation separated by spaces) to the specified output type one word per line. This will make sure you can read and write to/from files and handle the command line arguments properly.

- **Part 2: Build the Acronym Dictionary using Malloc/Free**
  - Structures variables MUST be of the data type specified and MUST declared in the order given. You may change the name of the variables.
  - Assume that acronyms will be in CAPTIALS only in the input stream.
  - The Acronym Dictionary will be stored in a tree structure. Each node of the tree will have the following structure type:

```
struct dNode {
   char acronym[5]; //acronyms can be at most 5 letters
   int num_children; //# of acronyms below node in tree
   int num_phrases; //# of phrases associated w/this acronym

   //linked list of other acronyms with this dNode as a parent
   struct dNode *siblings;

   struct dNode *children; //pointer to acronyms children

   //linked list of possible phrases which acronym map to
   struct pNode *phrases;

};
```

- Each level of the tree appends a letter to the acronym. When a `dNode` contains a full acronym, such as “LOL”, then the list of phrases contain all possible expansions.
- The children of a `dNode` are longer acronyms which contain the parent as the prefix. Eg. “LOL” is the parent of “LOLL”, “LOLO”, “LOLQ”. (FYI, I made these up =)).
- Siblings linked lists and children linked lists should be stored in alphabetical order.
- The `num_children` variable is the total number of acronyms on all levels below this node in the sub-tree.
- The `num_phrases` variable is the total number of phrases for this acronym (Eg. Length of phrases linked list).
- Acronym contains the current letters of the `dNode`, Eg. “L”, “LO”, “LOL”, “LOLL”, “LOLO”, “LOLQ”
- The second structure `pNode`, is a node in the linked list to hold all possible phrases the acronym can be expanded to.

```
struct pNode{
   char phrase[30];  //string to hold the phrase (at most 29 chars)
   struct pNode* next; // pointer to next phrase in linked list

   // statistic about usage of phrase from dictionary file
   int freq;

};
```

- `pNode` linked lists should be stored from highest to lowest frequency.
- Assume that the word field in both structures is encoded in ASCII, and obey C string convention (terminated by '\0').

To use the structures:

- The dictionary input file has one acronym per line (a single word) followed by a frequency number and then one phrase for that acronym.

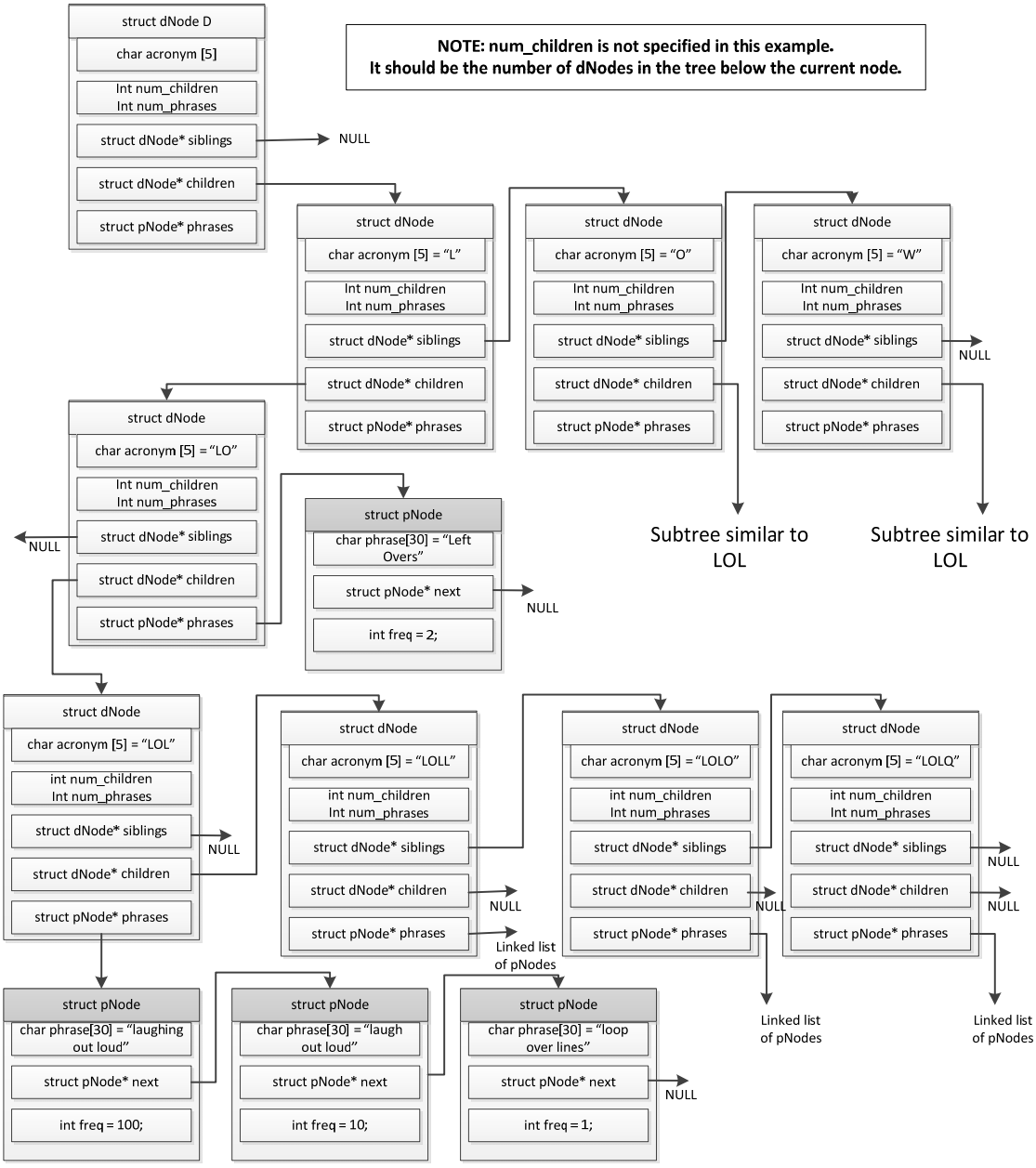
  acronym freq string of words for phrase

- An acronym can have multiple lines of input in a file, one for each possible phrase it expands to.
- Assume the dictionary file has NO ERRORS.
- In your main create the head dictionary node for your tree.

```
struct dNode D;
```

- Process the dictionary file line by line with the following steps, building a tree-like structure as shown in the **figure**.
  - malloc a new `dNode` instance for each substring of the acronym (“L”, “LO”, “LOL”), moving down a level in the tree (making each new dNode a child of the previous substring). Remember to increment the number of children of the node as you move down the tree.
  - Once at the dNode of the full acronym, malloc a pNode for the phrase and initialize the frequency and phrase string.

- As you add new acronyms to the tree-like structure, perform string compares to match the substrings with the current dNode acronym values. Only add new dNodes when a prefix of the acronym does not exist.
- Write functions to insert a new acronym, to traverse and print out sub-tress of the dictionary in the same format as it was read in into the output stream, and to add a new phrase to an existing acronym. These function will be useful in Part 3.
- Additionally, create a function to delete your entire acronym dictionary. Make sure to also `free` your all of your malloc memory before exiting the program.
- Modify your main function to parse the dictionary file and test that it is correct by printing it back to the screen or a file.



- **Part 3: Decoding Acronyms in text**
  - In your main function, once you have parsed your dictionary file and created your structure you will now replace all acronyms within an input stream with one of the possible phrases. Eg. If you find LOL

in the text, you will prompt the user to replace it with “laughing out loud”, ”laugh out loud” or “loop over lines”.

- Read in the input file line-by-line. You will need to print this complete line of text if there is an acronym found in the line (you may want to keep a copy).

- Tokenize each line into words.

    - If the word is not an acronym, print it to the output stream.

    - If it is an acronym, search the dictionary tree for the acronym.

        - If found, print the line of text, and prompt the user for which phrase to replace it with in the output.

```
“I am LOL. Are you LOT?”
Replace LOL with
1: “laughing out loud”
2: “laugh out loud”
3: “loop over lines”
4: Enter new phrase
Choice (1-4)? 1
```

          Once the user has selected a phrase, output the phrase to the output stream and continue to search the line.

          If the user selects enter a new phrase, prompt the user to enter the new phrase (29 characters or less) and add it to the dictionary with a frequency of 1. pNodes with the same frequency can be in any order in the list.

```
Enter the new phrase: loo ooo loo
Added “loo ooo loo” to Dictionary.
```

          Print the phrase to the output file and move on to the next word.

        - If not found, prompt the user to add it to the Dictionary. Allow only 1 phrase to be added. The user MUST add the new acronym to the dictionary.

```
“I am LOL. Are you LOT?”
LOT is not found.
Enter a new phrase for LOT: lots of time
Added LOT with “lots of time” to Dictionary.
```

- When finished replacing acronyms in the file, ask the user if they want to save the current dictionary. Prompt for the dictionary filename and output the dictionary file.

```
Do you want to save the current dictionary (Y/N)? Y
Output filename: new_dict.txt
Dictionary saved to new_dict.txt
```

- Print out the following statistics to the screen:

    - Total number of `dNodes` in the dictionary
    - Total number of `pNodes` in dictionary
    - Size of entire dictionary in memory (bytes) (both `dNodes` and `pNodes`) after processing input file
    - Total number of acronyms replaced in the input text
    - Print out a sub-tree for a user specified letter, meaning print all acronyms starting with “L” for example. Each acronym, followed all phrases

```
LO: “left overs”

LOL: “laughing out loud”, “laugh out loud”, “loop over lines”

LOLL: …
```

```
LOLO: …
LOLQ: …
```

- Remember before you exit the program you must delete your entire dictionary and phrase list to free all the memory!
- Punctuation should be copied over directly and not considered part of the word. Punctuation IS NOT ALLOWED within the acronyms. Eg, LO'L will never occur in the input.

- **Testing:**
  - We will be running your code on Sparky, with a much larger input file. We will be directly comparing your output results to the expected results. Make sure to follow the dictionary file formatting exactly.
  - Always test on sparky to ensure everything works correctly! You will lose points if you code does not compile or operate correctly on Sparky.

Electronic Submission:

Log into your account on *Sparky* (using *ssh* or another terminal program). Link to instructions: http://it.stonybrook.edu/services/sparky-unix

ftp your *.c file containing your source code to your account on *Sparky*, if it is not on sparky already.

From your account on *Sparky*, submit an electronic copy of your program to the *Sparky cse220* account by doing the following:

- Change directories to the directory where your *.c file you want to hand in is located.
- Type and execute the command: **~cse220/submit**
- Follow the instructions in the submit program. Entering 1 to upload your file, 2 to check your file status, 3 to exit the submit program.
  - Option 1: To upload your file, type the filename including the extension.
    - If you are having problems, try typing **./<filename.c>**
    - If your submission is successful, you will get an acknowledgement right away, along with the number of lines in your submission file.
  - Option 2: This confirms that your file has been submitted and copied to the *handin* directory in the *cse220* class account. The number of lines will be the size of your submission +1 as the program will append a comment with the date and time you submitted your file to the end of your program.
  - Option 3: This exits the submit program and takes you back to your terminal prompt.
- Note: If you submit more than once, each new submission overwrites the previous one, so only the last submission will be retained.
- Note: You may only submit a single file per homework assignment.

```
FILE *fopen(char *path, char *mode);
int fclose(FILE *file);
size_t fread(void *ptr, size_t size, int n, FILE *file);
```

- The `fread` function is used to read n objects, each of size `size`, from the open file specified by `file`, and it places the data read into memory starting at the location specified by `ptr`. The region of memory pointed at by `ptr` must be at least `n*size` bytes long, because that is how many bytes of data will be read if the fread is successful. The value returned by `fread` is the number of objects (anywhere from 0 to n) that were read successfully. If this value is less than n it means either that end of file was reached or that a read error occurred.
- The `fread` function is designed to read several objects at a time into an array. For this homework, we will only need to read one object at a time, so you should always pass the value 1 for the parameter n. The type size_t is ultimately defined (in stdio.h) to be an unsigned integer type of a suitable width. You don't really have to know what it really is; just use `size_t` as if it is a predefined C type.

```
size_t fwrite(void *ptr, size_t size, int n, FILE *file);
```

- The `fwrite` function is used to write n objects, each of size `size` and stored in memory starting at the location specified by `ptr`, into the open file specified by file. The value returned by `fwrite` is the number of objects successfully written; if this is less than n it means that an error occurred. Just as for `fread` we will only need to call `fwrite` with 1 as the value of the parameter n.

```
int fseek(FILE *file, long offset, int whence);
```

- In the C standard I/O library, each open file has an associated offset, which specifies the position within the file at which the next `fread` or `fwrite` will be performed. When a file is initially opened, the offset is set to 0. Each time a `fread` or `fwrite` is performed, the offset is incremented by the number of bytes read or written. The purpose of the `fseek` function is to permit the offset to be set directly. This permits random access to the data within a file, rather than just sequential access.
- In a call to `fseek`, the file parameter is a pointer that was previously obtained from a call to `fopen`. The offset is the desired offset, specified in units of bytes. The whence parameter is a flag that determines how the offset is to be interpreted. For our purposes, you should always pass the constant SEEK_SET, which means the offset is interpreted as the number of bytes from the beginning of the file (i.e. the beginning of the file corresponds to offset 0).

```
void rewind(FILE *file);
```

- A call `rewind(file)` function has the effect of a call to

  ```
  fseek(file, 0L, SEEK_SET);
  ```

  but in addition the call to `fseek` clears any error state that is associated with `file`.

Note: You should not intermix calls to `fread` and `fwrite` on the same open file without calling rewind whenever you switch from reading to writing and vice versa. Also, the "L" in the "0L" parameter above means "the value 0 of type long".