Programming Assignment 3 (PA3) - History of Words

Milestone Due: **Wednesday, May 15 @ 11:59 pm**
Final Due: **Wednesday, May 22 @ 11:59 pm**

Assignment Overview	Getting Started	Walkthrough and Example Input	Detailed Overview	Milestone Functions
Post-Milestone Functions	Unit Testing	README File	Extra Credit	Turnin Summary

Assignment Overview

The purpose of this programming assignment is to gain familiarity with structs, pointers, dynamic memory allocation, resizing arrays, string operations, sorting, searching, and reading text files. We will be working with dynamic memory allocation functions like `calloc`, `realloc`, and `free`; string functions like `strncpy` and `strncmp`; stdio functions like `fopen`, `fgets`, `fclose`; and stdlib functions like `qsort` and `bsearch`. We will get further practice with looping constructs, conditional constructs, function calls, and working with global variables.

You are an etymologist studying the history of words. You want to develop a program that can read in data files on word occurrences in each decade from year 1800 - year 2009. The data is formatted in files which contain the word, decade, and the count of that word for the decade. Your mission is to allow other fellow etymologists to find out the stats for specific words and even plot them on a graph. It is now up to you to reveal the history of words and there's a lot at stake (~10% of your overall grade).

Remember to read the man pages (RTFM--Read The Friendly Manual) in order to lookup specific C functions. For example, if you would like to know what parameters `qsort()` takes, type `man -s3 qsort`. Also, make good use of the tutors in the lab. They are there to help you learn more, point you to useful resources, and help you get through the course!

IMPORTANT NOTE for Assembly routines:

1. Make sure you **do not** use registers other than **r0-r3** as scratch registers in your assembly functions. Allocate local variables on the stack instead.
2. Only **fp, lr** are pushed to the stack.
3. Note that values in **r0-r3** will not be preserved after function calls.
4. Remember to note in the **function header** what stack variables you used (i.e somevar -- [fp - 8] -- for calculating length, etc). Stack variables include local variables, formal parameters, and additional arguments. Refer to style guidelines for the specific way to format the header.

Make sure you start early! This assignment will likely take longer than the previous assignments and will be more difficult. If you do not start early, you will very likely not finish this assignment on time!

(many students have tried testing this hypothesis in the past -- please learn from their mistakes!)

Grading

- **README: 10 points** - See README Requirements here and questions below
  - http://cseweb.ucsd.edu/~ricko/CSE30READMEGuidelines.pdf
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 10 points** - See Style Requirements here
  - http://cseweb.ucsd.edu/~ricko/CSE30StyleGuidelines.pdf
- **Correctness: 75 points**
  - **Milestone (20 points)** - To be distributed across the Milestone functions (see below)
  - Make sure you have all files tracked in Git.
- **Extra Credit: 2 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

Gathering Starter Files:

The first step is to gather all the appropriate files for this assignment. Connect to pi-cluster via ssh.

```
$ ssh cs30xyz@pi-cluster.ucsd.edu
```

Create and enter the pa3 working directory.

```
$ mkdir ~/pa3
$ cd ~/pa3
```

Copy the starter files from the public directory.

```
$ cp ~/../public/pa3StarterFiles/* ~/pa3/
```

Starter files provided:

pa3.h	pa3Strings.h	pa3Globals.c
test.h	testaddWordCount.c	printTable.c
Makefile		

You will also need to copy over intervalContains.s, findCommand.c, shouldPrompt.c from pa2. **If you have not done so already, make sure you fix any style issues in this file before continuing on to the rest of your code!** You don't want to get the same style deduction from previous PAs.

```
$ cp ~/pa2/intervalContains.s ~/pa3/
$ cp ~/pa2/findCommand.c ~/pa3/
$ cp ~/pa2/shouldPrompt.c ~/pa3/
```

Preparing Git Repository:

You are required to use Git with this and all future programming assignments. Refer to the PA0 writeup for how to set up your local git repository.

Walkthrough and Example Input

Sample Executable

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa3test
```

Walkthrough

Let's walk through how to use the executable. Here is the usage, which can be viewed by using the -h flag:

```
Usage: ./pa3 [-n numSlots] [-s size] [-h]
Interactive program for studying word use frequency across decades.
Uses files from '/home/linux/ieng6/cs30x/public/pa3Data/'.
  -n numSlots   The number of slots in the hash table.
                Must be in range [3, 421]. Defaults to 11.
  -s size       The size of files to be used. Files are identified by
                {letter}_{size} (e.g. a_5 has size 5). Must be one of
                {1, 5, 10, 50, 100, 500}. Defaults to 50.
  -h            Prints this help message.
```

As described in the usage statement, this program can be used to view the frequency that a word is used across decades. Data for these words is collected in `~/../public/pa3Data/`. When the program reads this data, it stores it in a large table with a number of slots, which can be given by the -n flag. The -s flag indicates the size of the data file. The file contains data for this many distinct words. We only

WARNING: The files in **`~/../public/pa3Data`** are **LARGE**. Do **NOT** copy them into your directory as you may exceed your disk quota. If you exceed your disk quota and it is near the deadline, you might not be able to do **anything** (not even running make) and **will not be able to turn in the assignment**. It takes time for ACMS to help reset your quota.

Since all of the flags are optional, let's just run the executable with no arguments, to use their default values (numSlots = 11, size = 50). User input is given in **bold text**.

```
cs30xyz@pi-cluster-001:pa3$ ./pa3

Commands:
        plot    word -- Display a plot showing a word's popularity over time.
        data    word -- List word occurrences by decade for a word.
        help         -- Display this message.
```

Here we can see the commands that the interactive mode supports. `plot` will show a graph of the word's use over every decade.

```
>>> plot and

 (3079848197) 100% |                                                 x
               96% |
               92% |
               88% |
               84% |
               80% |
               76% |
               72% |
               68% |
               64% |
               60% |                                              x
               56% |
               52% |
               48% |
               44% |
               40% |                                           x
               36% |
               32% |
               28% |                                        x
               24% |
               20% |                                     x
               16% |
               12% |                                  x
                8% |                        x x x x x
                4% |              x   x x x
 (   31173884)   0% | x x x x x    x
                    ----------------------------------------------
                      1   1   1   1   1   1   1   1   1   1   2
                      8   8   8   8   8   9   9   9   9   9   0
                      0   2   4   6   8   0   2   4   6   8   0
                      0   0   0   0   0   0   0   0   0   0   0
```

The plot is scaled so that we only consider the range of counts between the minimum and maximum values (shown on the left of the y-axis in parentheses). The percentage represents how far the count is in between the minimum and maximum counts. The x-axis is labeled by each decade, skipping every other decade for space's sake. However, the data for all decades are still plotted on the graph (check the `x` marks to be sure).

Back to Top

`data` will show the word counts per decade in a tabular format, while also noting the decade when the word was the most popular.

```
>>> data and

      Decade          Count
      ------     ----------
       1800        31173884
       1810        47261995
       1820        74009030
       1830        95664389
       1840       119873316
       1850       164572019
       1860       139413201
       1870       167922793
       1880       219899433
       1890       258683468
       1900       351708100
       1910       321054145
       1920       300434093
       1930       281767927
       1940       280426701
       1950       417230586
       1960       750360597
       1970       986348910
       1980      1259626856
       1990      1916323016
       2000      3079848197

'and' was used most frequently in 2000 with 3079848197 occurrences
```

Note that "and" has no ties for the decade in which it was most frequently used, so the message just says "used most frequently in".

If there is a tie between decades for the highest count, then the program only states the most recent decade:

```
>>> data the

      Decade          Count
      ------     ----------
        ...         ...
       1980      2962338524
       1990      4294967295
       2000      4294967295

'the' was used most frequently recently in 2000 with 4294967295 occurrences
```

Note that "the" ties for the highest count in both decades 1990 and 2000, so we report 2000 as "most frequently recently". (The "..." is used below to shorten the example -- feel free to try it yourself to see the full output.) **If there is a tie, make sure to use the message that includes "recently", but if there is no tie then use the message that does not include "recently".**

If you try out other words, you'll notice that almost every word has increasingly higher counts as the decades reach recent times. This is due to the fact that there is just more data available to collect now than there was in

1800. If we wanted to combat this skewing effect, we would need to normalize the counts for every word based on the number of source materials it appeared in that decade, but we won't be doing this in PA3.

Despite the skewing of word counts, we can find some words whose highest counts appear in decades before 2000. Some examples include "unfrequently", "quhilk", and "zwingle". Try these out with the -s flag set to 500 (it may take some time to load the data). Notice that these words are often obsolete ones, which explains why they were used more before 2000 than after 2000.

As usual, press Ctrl+D (^D) to exit.

```
>>> ^D
cs30xyz@pi-cluster-001:pa3$
```

Example Input

NOTE:

1. The output of your program MUST match exactly as it appears in the `pa3test` output. You need to pay attention to everything in the output, from the order of the error messages to the small things like extra newlines at the end (or beginning, or middle, or everywhere)!
2. **We are not giving you any sample outputs, instead you are provided some example inputs. You are responsible for trying out all functionality of the program; the list of example inputs is not exhaustive or complete. It is important that you fully understand how the program works and you test your final solution thoroughly against the executable.**

Example input that has error output:

```
cs30xyz@pi-cluster-001:~/pa3$ ./pa3 420
cs30xyz@pi-cluster-001:~/pa3$ ./pa3 -n -s
cs30xyz@pi-cluster-001:~/pa3$ ./pa3 -s 70
```

Example input that has normal output:

```
cs30xyz@pi-cluster-001:~/pa3$ ./pa3
cs30xyz@pi-cluster-001:~/pa3$ ./pa3 -h
cs30xyz@pi-cluster-001:~/pa3$ ./pa3 -n 50 -s 100
```

Commands in interactive mode with error output:

```
>>> f
>>> plot
```

Commands in interactive mode with normal output:

```
>>> plot zwingle
>>> data the
```

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```
void createWordData( wordData_t * wordData, const char * wordStr,
                     unsigned int decade, unsigned int count );
int addWordCount( tableSlot_t * slot, const char * word, unsigned int decade,
                  unsigned int count );
int buildWordTable( wordTable_t * table, unsigned int size );
void buildPlot( char plotArr[PLOT_ROWS][PLOT_COLS], const wordData_t * word,
                  unsigned int minCount, unsigned int maxCount );
int findCommand( const char * cmdString, const char * commands[] );
void printPlot( char plotArr[PLOT_ROWS][PLOT_COLS], unsigned int minCount,
                  unsigned int maxCount );
void printData( const wordData_t * word );
void interactiveLoop( const wordTable_t * table );
int shouldPrompt( void );
void printTable( const wordTable_t * table );
```

Assembly routines:

```
unsigned int computeHash( const char * str );
unsigned int myURem( unsigned int dividend, unsigned int divisor );
unsigned int findSlotIndex( const char * wordStr, size_t numSlots );
int compareWordData( const void * w1, const void * w2 );
int intervalContains( int start, int end, int value );
```

For the Milestone, you will need to complete:

computeHash.s	myURem.s	findSlotIndex.s
compareWordData.s	createWordData.c	addWordCount.c

Back to Top

Process Overview:

The following is an explanation of the main tasks of the assignment, and how the individual functions work together to form the whole program.

Fig. 1. The wordTable_t Table Structure

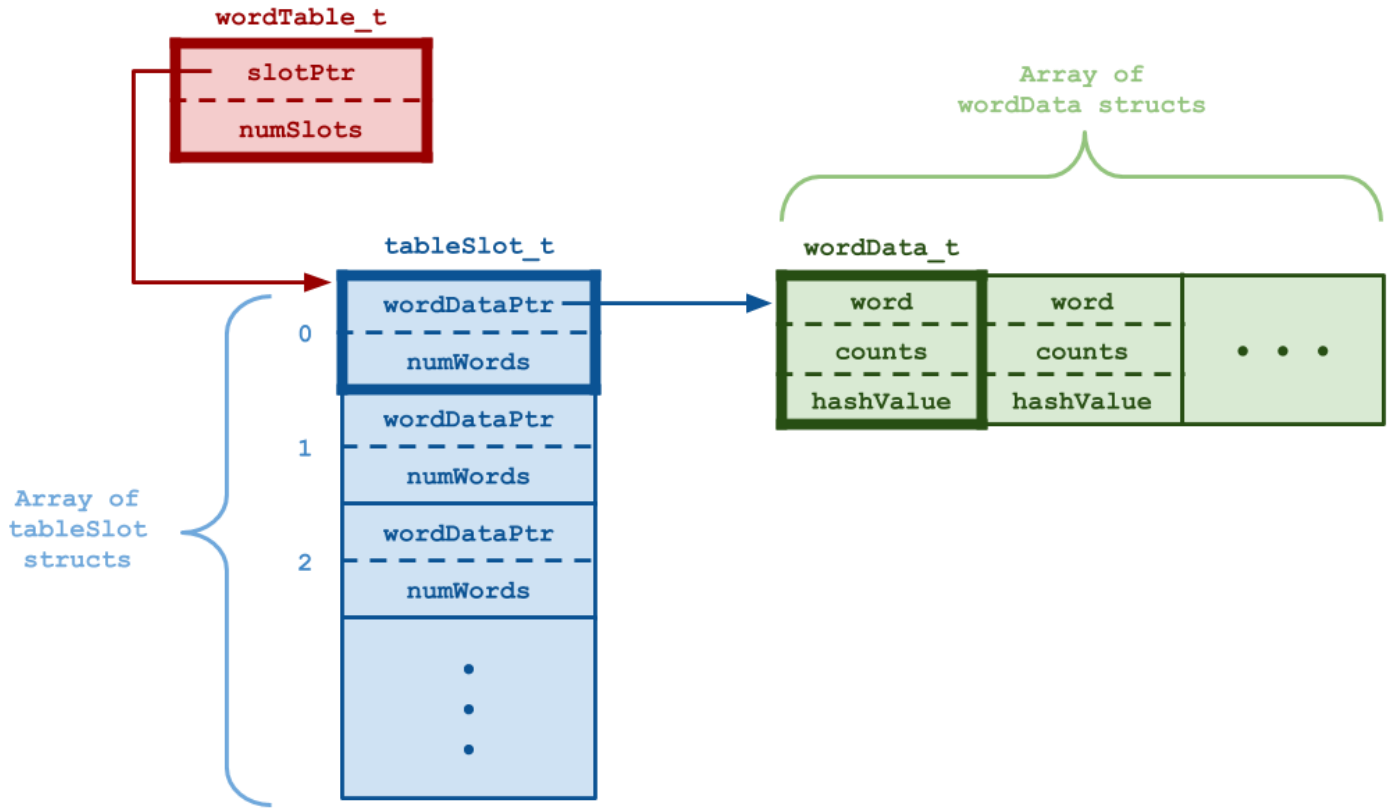
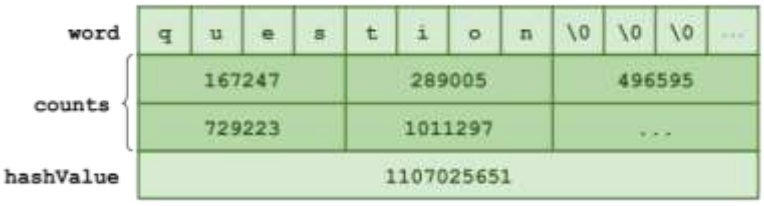


Fig. 2. The wordData_t Struct Type (note: diagram not to scale)

```
struct wordData {
  char word[MAX_WORD_SIZE];
  unsigned int counts[NUM_OF_DECADES];
  unsigned int hashValue;
};
typedef struct wordData wordData_t;
```



We store data on each word in a `wordData_t`, typedef'd from `struct wordData` (see Figure 2). The struct has a member to hold the word itself (word), a member which holds the counts for all 21 decades (counts), and a member holding the hash value of the word string (hashValue). We store all the words in a giant hash table structure (Figure 1), modding the word's hash value by the table size to choose the `tableSlot_t` to insert into.

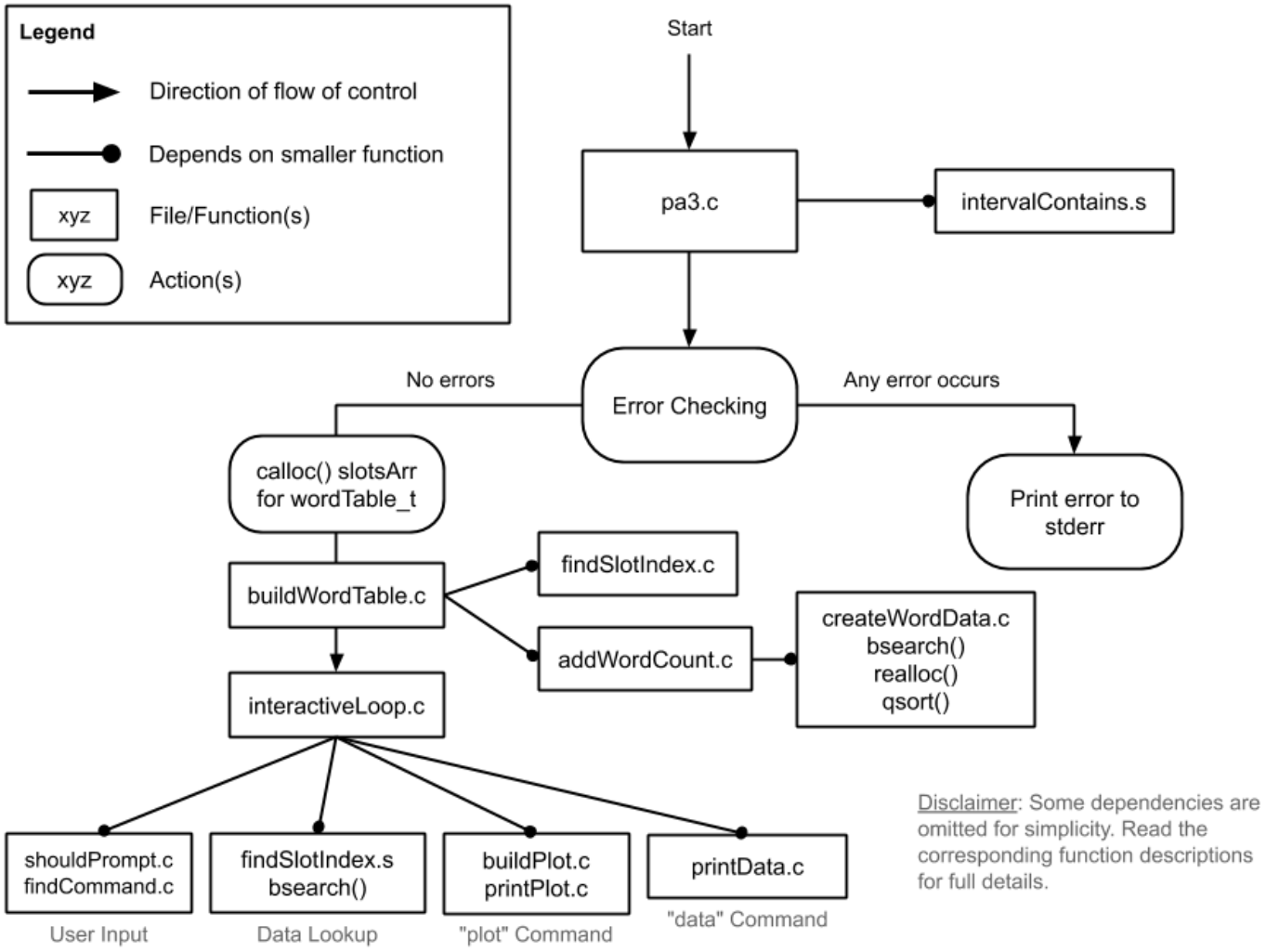
Each `tableSlot_t` is like a hash table bucket, which may have collisions since the table size is most likely smaller than the number of words. Inside each bucket, all of the `wordData_t` are sorted on the hashValue, and then on the word string itself, for easy lookup (i.e. binary search).

Back to Top

Note that the counts array will have 21 elements for the 21 decades between 1800-2000. We can't directly use the decade number to access an element of this array, since the year is in the range 1800-2000 but the counts array only has indices 0-20. This means that to be able to access the counts array, we have to map the year of a decade to its corresponding index. For example:

- Decade 1800 maps to index 0
- Decade 1810 maps to index 1
- Decade 1820 maps to index 2
- ... and so on.

How exactly you should do this is up to you. (Hint: it should be a one-line formula.)



The executable should, at a high level, follow the sequence of steps below:

1. Parse command-line arguments.
    a. If any error occurs, print the corresponding message to stderr and exit with failure.
2. Dynamically allocate space for the hash table structure.
3. Use buildWordTable() to read files from ~/../public/pa3Data/ and populate the table with word data.
4. Use interactiveLoop() to launch the interactive loop and prompt for user input.
    a. Handle each command to plot a graph or show a table of data for words.

Back to Top

Milestone Functions to be Written

Listed below are modules to be written for the milestone.

---

computeHash.s

```
unsigned int computeHash( const char * str );
```

This function will be used to create the hash value of a string. This function creates an unsigned integer hash value from `str` with the following hashing algorithm. Your task is to translate this algorithm into assembly. Use the global variables defined in `pa3Globals.c` to access these constants in assembly. Refer to the discussion worksheet on accessing global variables in assembly.

```
unsigned int hash = HASH_START_VAL;
for ( int i = 0; str[i] != '\0'; i++ ) {
   hash = hash * HASH_PRIME + str[i];
}
return hash;
```

Return Value: The hash value of `str`.

---

myURem.s

```
unsigned int myURem( unsigned int dividend, unsigned int divisor );
```

Calculates the remainder when dividing `dividend` by `divisor`, where `dividend`, `divisor`, and the remainder are all unsigned integers. Hint: This will be extremely similar to a function you have written for previous PAs.

Reasons for error:

- `divisor` is zero → result is undefined (we will not be checking for divide by 0)

Return Value: The remainder.

---

findSlotIndex.s

```
unsigned int findSlotIndex( const char * wordStr, size_t numSlots );
```

This function will calculate the index into the slot array for the string `wordStr` in a slot array of size `numSlots`. Please strictly follow the order of the steps below.

1. Calculate the hash value (using your `computeHash()` function) of the `wordStr` string.
2. Map the hash value to a valid index in the slot array by modding the hash value by `numSlots` (use your `myURem()` function).

Return Value: The index of `wordStr` in the slot array.

Note on comparison functions:
In this assignment you will be using two of the C standard library array sorting and searching functions: `qsort()` and `bsearch()`. Both of these functions take in a pointer to a comparison function (in order to sort the array, you need to provide instructions on how to compare the elements of the array--this is the purpose of the following comparison function). The comparison function must take in two `const void *`'s as parameters.

Your comparison function should behave according to the following standard conventions:

- Return -1 if the first argument should be ordered earlier than the second
- Return 0 if the arguments are equal
- Return 1 if the first argument should be ordered later than the second

Refer to `man -s3 qsort` and `man -s3 bsearch` for more information.

---

compareWordData.s

```
int compareWordData( const void * w1, const void * w2 );
```

This function takes two void pointers in its signature, but you will pass in `wordData_t` pointers (`const wordData_t *`) and compare them. This comparator should first compare the `wordData_t`'s hash values. If they have the same hash value, then compare the `wordData_t`'s words using `strncmp`, with the maximum number of characters to compare being `MAX_WORD_SIZE`.

Note: The hashValue member is declared as an unsigned int. This may affect what kind of branching instructions you should when comparing the two hashValues. Hint: `blo` is the unsigned version of `blt`, and `bhi` is the unsigned version of `bgt`. See your lecture notes (Notes #5) for other unsigned branching instructions.

Return Value: -1 if

- The `wordData_t` that w1 points to has a smaller hashValue than the `wordData_t` that w2 points to, OR
- The hashValues are the same but strncmp returns a value < 0.

1 if

- The `wordData_t` that w1 points to has a larger hashValue than the `wordData_t` that w2 points to, OR
- The hashValues are the same but strncmp returns a value > 0.

0 if the hashValues are the same and strncmp returns 0.

---

createWordData.c

```
void createWordData( wordData_t * wordData, const char * wordStr,
                     unsigned int decade, unsigned int count );
```

Populates the fields of the `wordData` struct to represent the word in `wordStr`.

1. First, calculate the index of the decade.
    a. Hint: Do some math using `MIN_DECADE` and `YEARS_IN_DECADE`.
2. Copy `wordStr` into `wordData`'s `word` field using `strncpy`, making sure to manually null terminate the string by setting the last character of the array to the null character '\0'. Check `man -s3 strncpy` , it does not guarantee null terminated strings.

    a. Hint: What's the largest that the word could be?
3. Set all the numbers in `wordData`'s `counts` array to 0
    a. Then set the item of the `counts` array at the index of the decade to be `count`.
4. Set `wordData`'s hash value using `computeHash()`.

---

addWordCount.c

```
int addWordCount( tableSlot_t * slot, const char * word, unsigned int decade,
                  unsigned int count );
```

This function will add the count for the `word` and the decade to the tableSlot.

First, calculate the index of the decade, just as in `createWordData`.

Make a `wordData_t` allocated on the stack (Hint: do not use any function ending in -alloc). Use `createWordData` to populate this `wordData_t` using `word`, `decade`, and `count`.

Try to find if there is already a `wordData_t` for `word` in `slot`:

- To search the `slot`'s `wordDataPtr`, use `bsearch()` `(man -s3 bsearch)`.
- If `bsearch()` returns a non-null pointer, there is already a `wordData_t` in slot for word.
    - Update the count inside the existing `wordData_t` for the specific decade by adding count to it.
    - After summing the counts, return 0.
- If `bsearch()` returns a null pointer, then we'll need to add room for another `wordData_t` in the slot's `wordDataPtr`.
    - We'll use realloc to dynamically allocate a new `wordData_t` at the end of the array. This is where you need to use `realloc()` `(man -s3 realloc)`. Make sure you really understand how this function works before using it. If you don't use `realloc()` properly, you could introduce incredibly challenging bugs into your code that could take days to debug--don't do this to yourself!
    - If memory allocation failed:
        - Print out `STR_ERR_MEM_EXCEEDED` to `stderr`.
        - Free and zero out (set to `NULL`) slot's `wordDataPtr`.
        - Return -1.
    - Assign the new `wordData_t` into the end of the array, using the `wordData_t` you allocated on the stack previously.
    - Make sure to update slot's `numWords` to reflect the new word you just added.
    - Sort the array using `qsort()`.
    - Return 0.

Reasons for error:

- If realloc fails, print `STR_ERR_MEM_EXCEEDED` to `stderr`, **free the slot's `wordDataPtr`**, and return -1.

Return Value: -1 if any error occurred, 0 otherwise.

Post-Milestone Functions to be Written

Listed below are the modules to be written after the milestone functions are complete.

---

intervalContains.s

```
int intervalContains( int start, int end, int value );
```

Copy from PA2. No code changes necessary **BUT** make sure to double check the style is correct to prevent repeated deductions from previous PAs.

---

shouldPrompt.c

```
int shouldPrompt( void );
```

Copy from PA2. No changes necessary.

---

findCommand.c

```
int findCommand( const char * cmdString, const char * commands[] );
```

Copy from PA2. No code changes necessary **BUT** make sure to double check the style is correct to prevent repeated deductions from previous PAs.

---

printTable.c

```
void printTable( const wordTable_t * table );
```

Pretty-prints the wordTable_t to `stdout`. Provided for you as a convenience function to see if your wordTable_t structure is correct. Feel free to use this when testing buildWordTable.c or inside pa3.c to double check your structure.

---

buildPlot.c

```
void buildPlot( char plotArr[PLOT_ROWS][PLOT_COLS], const wordData_t * word,
                unsigned int minCount, unsigned int maxCount );
```

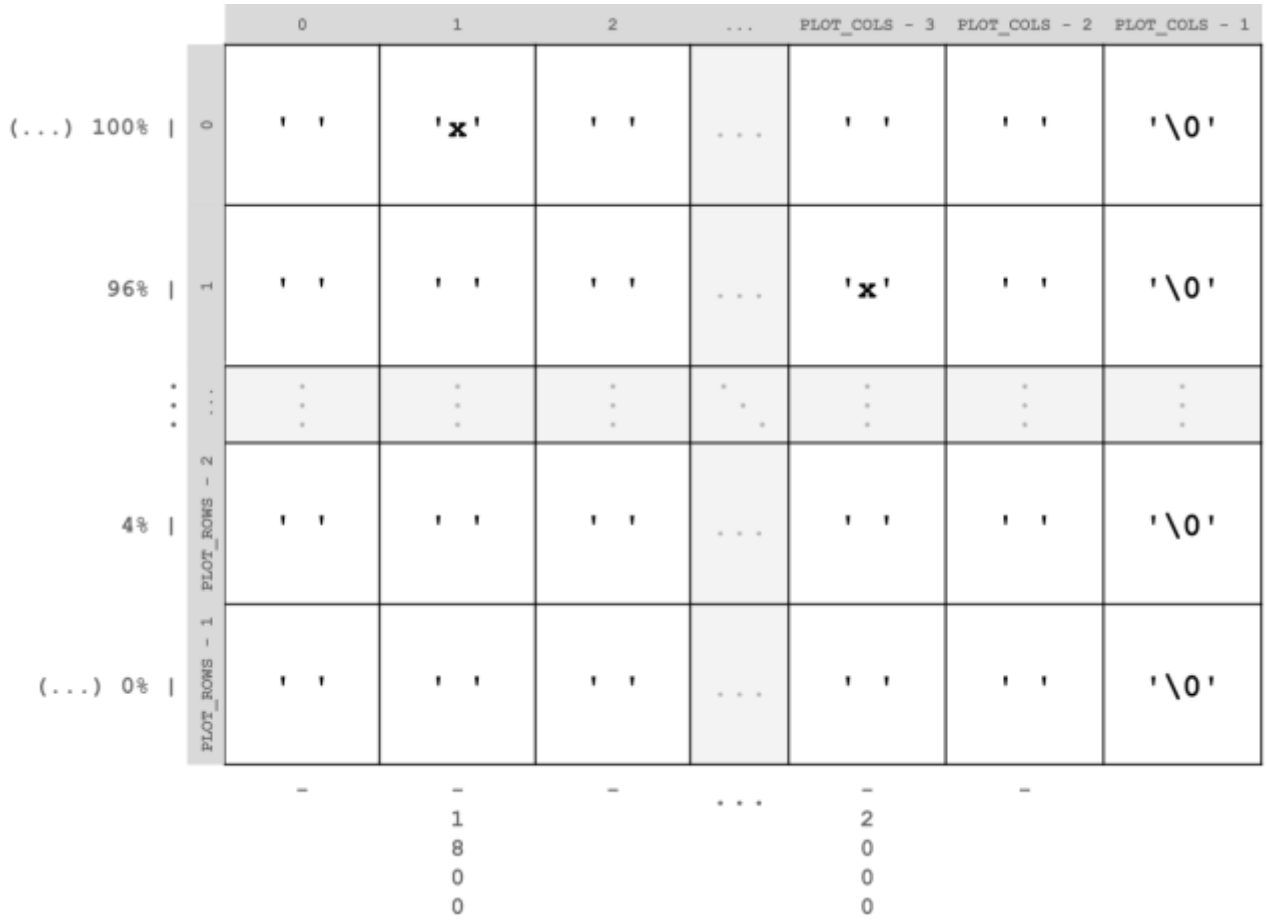
This function will populate `plotArr`, a 2d-array that contains the plotted data for the word data within `word`. This is the plot without axes. This graph will be printed later, and therefore must be initialized in such a way to make printing the graph trivial. To initialize `plotArr`, this function should go through the count of each decade for `word`, plotting each data point at a row and column in the 2d-array (see diagram below). Please refer to the reference executable for further clarification on the format of these graphs.

`minCount` is the minimum count of all decades and `maxCount` is the maximum of all decades, which are used to rescale the plot to 0% - 100% and make it look more reasonable.

The count values are very large, so you should work with a truncated value for every count. Define new variables `truncMin`, `truncMax`, and `truncCount` which are to hold the `minCount`, `maxCount`, and `count` values (from the `wordData_t`) which are integer-divided by 100. For example, a count of 10472387 should be truncated to 104723 (effectively, the last two decimal digits are removed).

Things to consider:

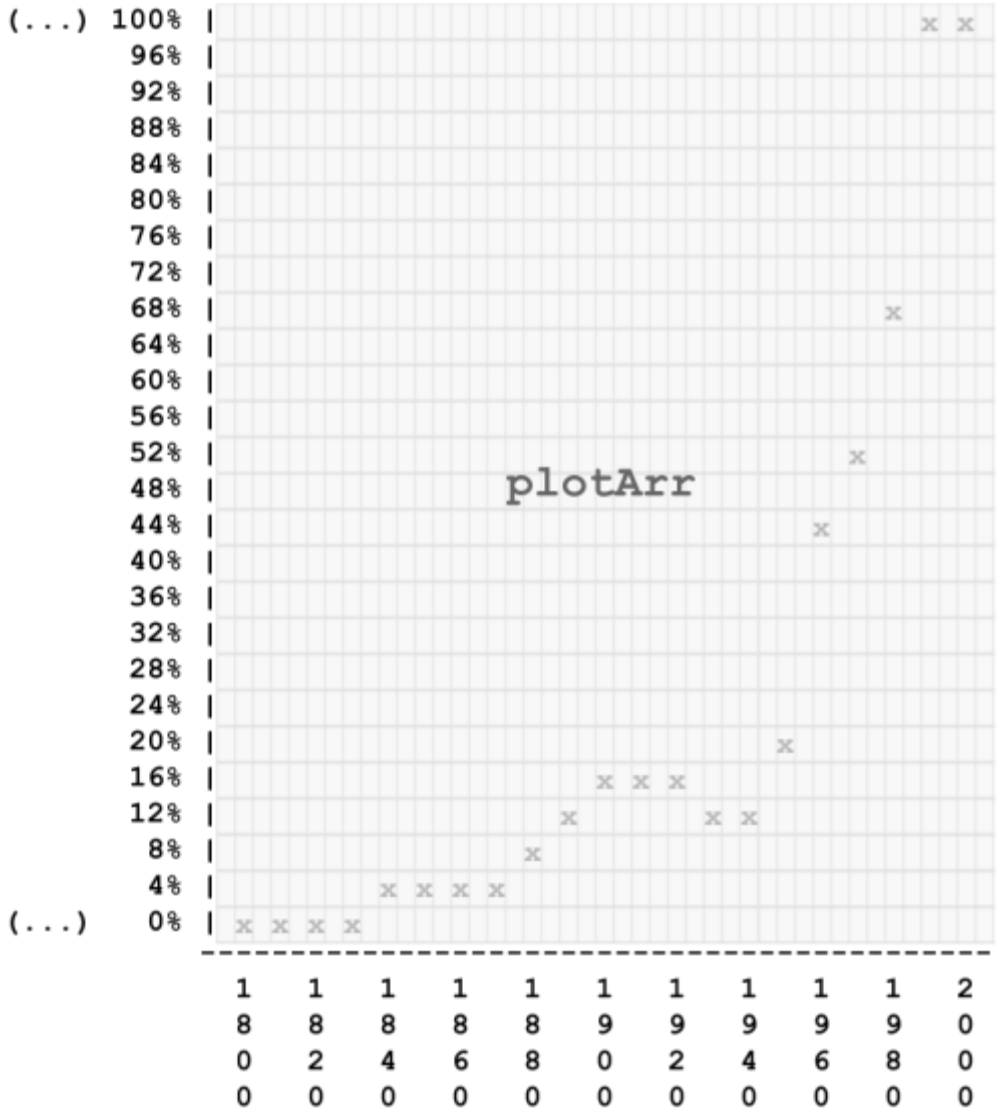
- Given `count` of a certain decade, truncate it to `truncCount` (like described above), and the percentage can be calculated as $\frac{(truncCount - truncMin) * 100}{truncMax - truncMin}$.
- If `truncMin` and `truncMax` are both 0, define the percentage to be 0.
- Otherwise, if `truncMin == truncMax`, define the percentage to be 100.
- Make considerations for the fact that percentage != row index. For example, a word has percentage 39%, it will fall under row [36%, 40%), which is actually row index 16.
- There are `PLOT_ROWS` rows and `PLOT_COLS` columns in total. The X axis refers to the decade and Y axis refers to the percentage of the count of that decade out of the difference between max and min.
- The difference in percentage between each row is `ROW_RESOLUTION`%.
- Notice that there is an extra column between every 2 decades which does not receive a plot marker `'x' (PLOT_LINE_CHAR)`. This means every even column will not have a `PLOT_LINE_CHAR`.
- Use `PLOT_LINE_CHAR` to indicate the percentage for each decade column and fill the array with `PLOT_SPACE_CHAR` for the rest. Make sure to set the last character of each row to be null terminator character `'\0'`.



Note: Numbers on a dark gray background represent row/column indices of the array. The X (decade) and Y (percent) axis labels are shown next to the corresponding rows and columns.

printPlot.c

```
void printPlot( char plotArr[PLOT_ROWS][PLOT_COLS], unsigned int minCount,
                unsigned int maxCount );
```



Prints the graph that has been previously initialized by `buildPlot()` to `stdout`. It is recommended that you iterate through the rows of `plotArr` in order to print each line of the graph. Note that you must figure out how to print out the labels for each row and column (hint: `STR_PLOT_ROW` and `STR_PLOT_X_AXIS`). Notice for the first row, `maxCount` is printed in parentheses and for the last row, `minCount` is printed as well (Hint: Use `STR_PLOT_ROW_BOUND`).

---

printData.c

```
void printData( const wordData_t * word );
```

This function prints out the word statistics in a tabular format for the given `wordData_t` to `stdout`.

First print the header for the stats by using the provided `STR_STATS_HEADER` found in `PA3Strings.h`. Then go through the `counts` array, printing out the stats for each decade by using `STR_STATS_FMT_COUNT`.

After printing out the statistics for the `word` in each decade, you will also need to print out a message, `STR_HIGH_COUNT`, indicating the year that the `word` had the highest count in (you should probably keep track of this information as you're looping through the `counts` array).

- If there are two decades in which the counts of the `word` are **the same** and are **the highest**, make sure to include `STR_MOST_RECENT` in your message (see the walkthrough for an example of this case).
- If not, then print the stats without this extra string (instead, pass an empty string).

---

buildWordTable.c

```
int buildWordTable( wordTable_t * table, unsigned int size );
```

This function will build the `table` by reading in data from files of a particular `size` inside directory `DEFAULT_DATA_DIR`. You will need to open each file inside the directory, populate the `table` with data from each file, and close each file. Following the steps below should aid you in this process.

Figure out what files we will collect data from. Do the following **for each file**:

- First you will need to build the path of the files you are trying to read. See the `STR_PATH_JOIN` format string in `pa3Strings.h` and consider using `snprintf()`. Each filename has the format of `<letter>_<size>` (`letter` will vary from `'a'` .. `'z'`). Take a look at the files in ~/../public/pa3Data/ for an example of the filename format.
- After you build the path, try opening it. (hint: `man -s3 fopen`)
    - If we can't open the file, print `STR_ERR_OPENING_FILE` to `stderr` and continue reading other files.
- If the file successfully opens, read each line of the file one at a time (read up to `BUFSIZ` characters for each line). Keep track of the line number you are on, since you will need this for error messages. Line numbering starts at 1 for each file. Continue to read lines until you reach the end of the file.
- For each line read from the file:
    - Make sure the line is null terminated. `fgets()` reads a `'\n'` when reading each line. `strchr()` should help you here. Hint: look back to PA2 for what to do.
    - The line contains the word, decade, and count separated by whitespace. (`strtok()` will help you again.)
        - If any of the 3 strings cannot be tokenized, print `STR_ERR_PARSING_LINE` to `stderr` and continue to read the next line.
    - Try converting decade and count to `unsigned int` (see `man -s3 strtoul`).
        - Note that the **count** can be very large. If it exceeds `ULONG_MAX`, just clip the count at `ULONG_MAX`. We are **not** treating this overflow as an error.
        - If there was any error parsing either number, print `STR_ERR_PARSING_LINE` to `stderr` and continue to read the next line.
    - After both numbers are converted successfully, we can store them in the `table` now!
        - Use `findSlotIndex()` to determine what index in the table the word hashes to.
        - Try adding the word to that slot with `addWordCount()`, if the call is not successful, close the file (`man -s3 fclose`) and return -1 immediately.
- After you are done reading the file, close it, then proceed to the next file.

Some notes about data files:

- Each file has the name format `<letter>_<size>`, where `letter` means the file contains words that start with that `letter`, and `size` means the number of unique words in the file.

- Each line of the file will be formatted as: `word decade count` (They are separated by tabs).
- You cannot assume any specific ordering of the lines within the file, so keep this in mind when writing your parsing algorithm.

Reasons for error:

- Error when opening a file.
- Error when tokenizing a line into the word, decade and count.
- Error when parsing the decade and count into numbers.
- Error when adding the word into table.

Return Value: -1 when any `addWordCount()` call failed, 0 otherwise.

---

interactiveLoop.c

```
void interactiveLoop( const wordTable_t * table );
```

This function is very similar to what you did in PA2 `commandLoop.c`. It allows the user to interactively search through the `table` for a specific word. This function first prints out a list of possible commands (provided in `pa3Strings.h:STR_HELP`). It should only print this usage message if `shouldPrompt()` returns 1. The function then prompts the user (using `PRINT_PROMPT`) to enter a command. This function continues to reprompt the user until the user enters `Ctrl-D` (signals EOF).

Note: Make sure to print to **`stdout`** for all output in this function.

An overview of the user-interactive process:

1. Read the command and argument(s) entered by the user (`man -s3 fgets`). Note that `fgets()` will include the newline character as part of the string read, so immediately after your call to `fgets()`, replace the newline character with the null character (`man -s3 strchr`).

   You should read user input in a for-loop with the following structure:
   `for (PRINT_PROMPT; condition; PRINT_PROMPT) { /* Parse line of input */ }`

   All the steps below should be within this for-loop structure. Note that when the following instructions say "reprompt the user" or "reprompt", it means you should print the prompt again and start parsing a new line of input. *Hint:* something about the way this for-loop is written will make this easier than it sounds.

2. After reading a line, parse the command that was entered by the user by using `strtok()` and the provided token separator string `DELIMS`. Extract the command name portion of the user's input from what was read by `fgets()`.
   - If the string could not be tokenized (i.e. the line was empty), reprompt the user.
   - Otherwise, use `findCommand()` to determine which command was entered.
     - If the command is unrecognized, then print the `STR_ERR_BAD_COMMAND` error message and reprompt the user.

Now that you have the index of the command, you will need to do different things depending on the command entered:

3. If the "help" command was entered, print the `STR_HELP` message and reprompt the user.

4. If the "data" or "plot" commands were entered, parse the one argument (`word`) that follows, using `strtok()`.
    - If the argument is missing, print the `STR_ERR_COMMAND_MISSING_ARG` error message, then reprompt.
    - If there are trailing arguments, print the `STR_ERR_COMMAND_EXTRA_ARG` error message and reprompt.
    - If no errors have occurred so far, try to find the target `wordData_t` that contains information for `word`. Note that this is similar to how you looked up a `wordData_t` in `addWordCount()`.
        i. Create a temporary `wordData_t` with `word`, so that we can use it to search for the target (hint: `createWordData()`)
        ii. Find the index of the slot that might contain `word` in the table (hint: `findSlotIndex()`)
        iii. Get the target `wordData_t` in the table (hint: `man -s3 bsearch`)
            - If the target is not found, that means we don't have data associated with this `word`. Print `STR_NO_DATA` and then reprompt user.
        iv. If the target is found, execute the corresponding command:
            - `plot` → Make a 2d char array with `PLOT_ROWS` x `PLOT_COLS` dimensions. Calculate the min and max count for the word of all decades then call `buildPlot()` and then `printPlot()`.
            - `data` → Print data related to this word using `printData()`.
5. If `Ctrl+D` was entered, then print the `STR_END_PROMPT` before terminating the program.

As a recap, if at any point you encounter an error in parsing the argument, print the corresponding error message, then reprompt the user and restart the procedure for parsing commands and arguments. Again, be sure to print to `stdout` for all output!

Otherwise, if you have successfully parsed the command and valid argument(s), then call on the respective function of the command the user entered. After doing so, print out the prompt and repeat the parsing procedures above.

---

pa3.c

```
int main( int argc, char * argv[] );
```

This is the main driver of your program. Its main tasks are to parse any command line arguments that are passed in and start the user-interactive mode. By default, the number of slots in the table will be `DEFAULT_NUM_SLOTS` and the data file size will be `DEFAULT_FILE_SIZE`. See the usage message in the above examples for details on what number of slots and data file size refer to.

In this function, you will use `getopt()` to read arguments from the command line. This function is commonly used for parsing flags and arguments. Think of this function as scanning the command line for flags and reading arguments immediately after flags if flags require an argument. Check the man page (`man -s3 getopt`) and refer to discussion slides/worksheet for more detail on how to use this function.

1. `HELP_FLAG` (-h) passed in:
    a. Print out the usage message using the given format string to `stdout`.

    b. NOTE, the first argument to the format string is the program name. How can you get that?
    c. Return `EXIT_SUCCESS`.
2. `NUM_SLOTS_FLAG` (-n) passed in:
    a. Store the number of slots for the table to be processed later. Remember what gets passed in from command line are strings, not integers.
3. `FILE_SIZE_FLAG` (-s) passed in:
    a. Store the file size to be processed later. This value is used to determine which data files to use.

If you encounter an unknown flag (e.g. `./pa3 -x`) or if a known flag is provided without its required argument (e.g. `./pa3 -n`), getopt() will automatically print an error message to stderr for you. All you need to do in this case is to return `EXIT_FAILURE`.

Remember that character literals (e.g. 'h') count as magic numbers, so be sure to use the constants defined in pa3.h so you won't receive style deductions for this.

After checking for known flags in the command line arguments, check if there are additional arguments passed in. If so, print `STR_ERR_EXTRA_ARG` to `stderr` and return `EXIT_FAILURE`.

If no parsing errors occur above, continue to process passed in arguments (if given), using `strtoul()` to convert the -n and -s flag arguments to unsigned ints. Only do this for flags that have been provided, otherwise just use the default value for the corresponding flag.

You should convert and check for errors **in the following order**. If you encounter an error, you should print the corresponding error message to **stderr** then return `EXIT_FAILURE` immediately:

1. **-n numSlots**
    a. Check if errno has been set. If so, print `STR_ERR_CONVERTING` with the corresponding argument name string (see `pa3Strings.h`).
    b. Check if there is an invalid character in the string. If so, print `STR_ERR_NOT_INT` with the corresponding argument name string.
    c. Check that the number of slots is in the interval [`MIN_NUM_SLOTS`, `MAX_NUM_SLOTS`] (hint: What function could you use to do this that rhymes with "interval retains"?). If it isn't, then print `STR_ERR_NUM_SLOTS_INTERVAL`.
2. **-s size**
    a. Check if errno has been set. If so, print `STR_ERR_CONVERTING` with the corresponding argument name string.
    b. Check if there is an invalid character in the string. If so, print `STR_ERR_NOT_INT` with the corresponding argument name string.
    c. Check that the size is within the `ALLOWED_FILE_SIZES` array (see `pa3.h`). If it isn't, then print `STR_ERR_SIZE_INVALID`.

Now, you are ready to build the table.

1. Allocate a `wordTable_t` on the stack. You fill need to initialize its `numSlots` member to the `numSlots` value you parsed above.
2. Allocate a zero-filled array of `tableSlot_t` using `calloc()`. This should go into the `slotPtr` member of the `wordTable_t`. If memory allocation error occurs, print out the error message `STR_ERR_MEM_EXCEEDED` to `stderr` and return `EXIT_FAILURE`. Otherwise, proceed to call

`buildWordTable()` on the stack-allocated `wordTable_t`. If this function fails, return `EXIT_FAILURE`.

3. If the call to buildWordTable() fails (returns -1), then you must free any dynamically allocated memory and return `EXIT_FAILURE`.

Now that everything is set up, enter user interactive mode by calling `interactiveLoop()`.

After `interactiveLoop()` returns, you must free all dynamically allocated memory within the wordTable_t hashtable structure. Remember that this structure has multiple dynamically allocated blocks of memory for each slot, so you will need a loop to free all of them.

Return `EXIT_SUCCESS` afterwards.

Reasons for error:

- Invalid flags passed in (`getopt()` will print the error message for you).
- Argument passed in is invalid due to incorrect format or invalid input (not in range or not valid file size).
- Memory allocation issues due to `*alloc()` functions.
- Failed to `buildWordTable()`. Free dynamically allocated memory then return `EXIT_FAILURE`.

Return Value: `EXIT_FAILURE` if any error occurs, `EXIT_SUCCESS` if program exit due to user exiting from the interactive loop.

Unit Testing

You are provided with only one basic unit test file for the Milestone function, `addWordCount()`. This file only has minimal test cases and is only meant to give you an idea of how to write your own unit test files. **You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all of the unit test files. You need to add as many unit tests as necessary to cover all possible use cases of each function. You will lose points if you don't do this!** You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function. The Makefile includes the rules for compiling these tests. Keep in mind that your unit tests will not build until all required files for that specific unit test have been written. These test files **will be collected with the Milestone**, they must be complete before you submit your Milestone.

Unit tests you need to complete:

```
testcomputeHash.c
testmyURem.c
testfindSlotIndex.c
testcompareWordData.c
testcreateWordData.c
testaddWordCount.c
```

To compile:

```
$ make testaddWordCount
```

To run:

```
$ ./testaddWordCount
```

(Replace "`testaddWordCount`" with the appropriate file names to compile and run the other unit tests)

We also supply you an extra set of "optimized" Makefile targets for your tester files. These targets will compile your tester file with all optimizations turned on, similar to how your Milestone functions will be compiled for grading. Sometimes you will get (un)lucky where your program appears to work even when there is something wrong (like incorrect memory layout or modifying a register you shouldn't). Compiling with optimizations will expose some of these hidden problems. Again, this is how the milestone will be tested during grading, so use the optimization targets to try to expose these issues.

However, compiling this way prevents you from using gdb to debug, so make sure to compile the regular way when debugging and try the optimized targets after your tests pass.

To compile with optimizations on:

```
$ make testaddWordCount-opt
```

To run:

```
$ ./testaddWordCount-opt
```

README File

Remember to follow all of the guidelines outlined in the README Guidelines. If you did the extra credit, write a program description for it in the README file as well.

Questions to Answer in your README:

[Unix]

1. You are trying to open your pa3Globals.c file. You have already typed "vim pa3G". Which *single* key can you press to autocomplete the command to "vim pa3Globals.c"?
2. What is the command to search for tabs in all the C files in your current directory? Your command should print out the line numbers of the tabs as well.

[C]

3. Suppose that we define the variable `wordDataPtr` like so:

```
wordData_t * wordDataPtr = calloc( 9, sizeof( wordData_t ) );
```

   (a) What is the value of `sizeof(wordDataPtr)` and
   (b) Why?
4. Give **two** different situations in which dynamic memory allocation (use of malloc/calloc/realloc) would be necessary.
5. **(a)** What is the difference between the & and * operator?
   (b) Complete the code below (provide code for the TODO and the blanks) such that `foobar` swaps the values of x and what y points to in main:

```
void foobar( int * a, int * b ) { /* TODO */ }
int main ( int argc, char * argv [] ) {
      int x = 420;
      int * y = malloc( __ );
      *y = 240;
      foobar( __ , __ );
      free( y );
      return 0;
}
```

[Vim]

6. What is the command to turn a lowercase character into an uppercase character and vice versa?
7. What is the command to repeat the last executed command?

[Academic Integrity]

8. There is an hour before the deadline and you've just submitted your working solution. Your friend messages you online and tells you that one of their functions isn't working properly. They ask you to send them a portion of your code. What are three actions you could take to act with integrity?

Extra Credit

There are 2 points total for extra credit on this assignment.

- Early turnin: **[2 Points]** 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)

Turnin Summary

See the turnin instructions here. Your file names must match the below *exactly* otherwise our Makefile will not find your files.

Milestone Turnin:
Due: Wednesday night, May 15 @ 11:59 pm

Files required for the Milestone:

computeHash.s	myURem.s	findSlotIndex.s
compareWordData.s	createWordData.c	addWordCount.c
testcomputeHash.c	testmyURem.c	testfindSlotIndex.c
testcompareWordData.c	testcreateWordData.c	testaddWordCount.c

Final Turnin:
Due: Wednesday night, May 22 @ 11:59 pm

Files required for the Final Turn-in:

compareWordData.s	addWordCount.c	pa3.h
computeHash.s	buildPlot.c	pa3Strings.h
findSlotIndex.s	buildWordTable.c	pa3Globals.c
intervalContains.s	createWordData.c	test.h
myURem.s	findCommand.c	Makefile
	interactiveLoop.c	README
	pa3.c	
	printData.c	
	printPlot.c	
	printTable.c	
	shouldPrompt.c	
	testaddWordCount.c	
	testcompareWordData.c	
	testcomputeHash.c	
	testcreateWordData.c	
	testfindSlotIndex.c	
	testmyURem.c	

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.