Programming Assignment 3 (PA3) - Anagrams

Milestone Due: **Wednesday, May 11 @ 11:59pm**
Final Due: **Tuesday, May 17 @ 11:59 pm**

Assignment Overview

The purpose of this programming assignment is to gain more experience with C programming, the Standard C Library routines, system calls, dynamic memory allocation on the Heap using `malloc()` and `realloc()`, sorting using `qsort()` and associated compare functions, reading data from files with `fgets()`, and more string manipulation using `strlen()`, `strchr()`, `strncpy()`, and `strcmp()`.

In this assignment you are writing an interactive program that takes words from `stdin` and searches for the corresponding anagrams from a user-provided anagrams file. The program will run until the user types the control sequence to indicate no more input (`EOF`). In Unix-based environments the sequence is `ctrl-D` (`^D`); in DOS-based environments it is `ctrl-Z` (`^Z`).

The user can specify the following options when running the program:

- The `-f` flag is the only required flag for the program. It requires an argument which indicates the name of the anagrams file to load into memory. The anagrams file must have a `.af` extension.
- The optional `-b` flag allows you to specify the number of bits of the `hashKey` to store for each anagram. By default, the `hashKey` is 32 bits. The larger the size of the `hashKey`, the less likely it is that collisions will occur.
- The optional `-c` flag will allow you to see the number of collisions that occur. A collision occurs when the hash keys of two words are the same, but the words are not actually anagrams of each other.
- The `-h` flag overrides all other flags and will print out the long usage statement which describes the possible command line arguments.

Make sure you start early! Remember that you can and should use `man` in order to lookup information on specific C functions. For example, if you would like to know what type of parameters `qsort()` takes, type `man -s3c qsort`. Also, take advantage of the tutors in the lab. They are there to help you learn more on your own, and to help you get through the course!

Grading

- **README: 5 points** - See README File section
- **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 section
- **Correctness: 80 points**
    - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
    - Make sure you have all files tracked in Git.
- **Extra Credit: 5 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 ieng9 via ssh (replace cs30xzzz with YOUR cs30 account).

```
$ ssh cs30xzzz@ieng9.ucsd.edu
```

Create and enter the pa3 working directory.

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

Copy the starter files from the public directory.

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

Copy your `isInRange.s` from your `pa1` directory.

```
$ cp ~/pa1/isInRange.s ~/pa3/
```

Starter files provided:

`pa3.h`	`pa3Strings.h`	`pa3Globals.c`
`test.h`	`testparseArgs.c`	`Makefile`

Preparing Git Repository:

Refer to previous writeups for preparing your Git repository. This will be required again for PA3.

Creating Anagram Files

You will be provided with a sample stripped executable to create an anagrams file (file extension `.af`) in order to test your program. Run it using the following command:

```
$ ~/../public/pa3createAF dictionaryFile anagramsFile
```

Where `dictionaryFile` is the name of a dictionary file (single words separated by newlines) and `anagramsFile` is the name of the anagram file that will be output (must end in `.af`).

Sample Output

A sample stripped executable located in the public directory is provided for you to compare the output of your program. 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/pa3ref
```

If there is a discrepancy between the sample output in this document and the `pa3ref` output, follow the `pa3ref` output.

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors

1.1. No argument.

```
[cs30xzzz@ieng9]:pa3$ ./pa3

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.2.1. Too few arguments (didn't specify a filename).

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f
./pa3: option requires an argument -- f

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.2.2. Too few arguments (didn't specify the number of hash bits).

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -b
./pa3: option requires an argument -- b

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.3.1. Too many arguments (extra operand).

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f file1 file2

     Extra operand 'file2'.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.3.2. Too many arguments (extra operands). Note only the first extra operand is reported.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f file1 file2 file3

     Extra operand 'file2'.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.4. Hash bit is too large to be converted to an integer.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -b 999999999999999999999

     Converting "999999999999999999999" base "10": Result too large

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.5. Hash bit is not a valid integer.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -b 123abc

     "123abc" is not an integer.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

1.6. Hash bit is outside of [1 - 32] range.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -b 33

     Number of hash bits must be within the range of [1 - 32].

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

2. Anagram File Errors

2.1. Filename doesn't have .af extension.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f file1

     Error opening anagrams file; must have .af extension.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

2.2. Can't read file because of not having the read permission.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f ~/../public/data.af

     Error opening anagrams file; permission denied.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

2.3. File is invalid.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f file1.af

     Error opening anagrams file; invalid file.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

3. Other Errors

3.1. Running out of memory.

```
[cs30xzzz@ieng9]:pa3$ ulimit -d 8
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af

     Could not load anagrams file. Memory limit likely exceeded.

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Try './pa3 -h' for more information.
```

4. Valid Output

4.1. Long usage.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -h

Usage: ./pa3 -f anagramsFile [-b hashBits] [-c] | [-h]
Interactive anagram search.
  -f <anagramsFile>  File with .af extension to read anagrams from.
  -b <hashBits>      Number of bits to extract from the hash value;
                     Must be a decimal value within the range [1 - 32].
  -c                 Display the number of hash key collisions.
  -h                 Print the long usage.
```

4.2. Word with several anagrams.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af

Enter a word to search for anagrams [^D to exit]: stop
Anagram(s) are: post spot stop

Enter a word to search for anagrams [^D to exit]: ^D
```

4.3. Word with only itself as an anagram.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af

Enter a word to search for anagrams [^D to exit]: hello
Anagram(s) are: hello

Enter a word to search for anagrams [^D to exit]: ^D
```

4.4. Word with no anagram.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af

Enter a word to search for anagrams [^D to exit]: ironman
No anagrams found.

Enter a word to search for anagrams [^D to exit]: ^D
```

4.5. Multiple words.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af
```

```
Enter a word to search for anagrams [^D to exit]: captain
Anagram(s) are: captain

Enter a word to search for anagrams [^D to exit]: america
Anagram(s) are: America

Enter a word to search for anagrams [^D to exit]: civil
Anagram(s) are: civil

Enter a word to search for anagrams [^D to exit]: war
Anagram(s) are: raw war

Enter a word to search for anagrams [^D to exit]: ^D
```

4.6. Exit successfully with no word entered.

```
[cs30xzzz@ieng9]:pa3$ ./pa3 -f data.af

Enter a word to search for anagrams [^D to exit]: ^D
```

Detailed Overview

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

C routines:

```
int main( int argc, char * argv[] );
int parseArgs( int argc, char * const argv[], struct argInfo * argInfo );
int loadAF( FILE * stream, struct anagramInfo * anagramInfo );
int loadAnagram( FILE * stream, struct anagram * anagram );
int findAnagrams( const struct argInfo * argInfo,
                  const struct anagramInfo * anagramInfo );
int createAnagram( const char * src, struct anagram * anagram );
void lowerCaseSortString( const char * src, int n, char * dest );
void usage( FILE * stream, enum usageMode u, const char * progName );
```

Assembly routines:

```
long hashString( const char * str );
long truncateHashKey( long hashKey, long numBits );
int charCompare( const void * p1, const void * p2 );
int hashKeyMemberCompare( const void * p1, const void * p2 );
int anagramCompare( const void * p1, const void * p2 );
int isInRange( long minRange, long maxRange, long value, long exclusive );
```

For the Milestone, you will need to complete:

```
truncateHashKey.s          hashString.s              charCompare.s
lowerCaseSortString.c      parseArgs.c
```

Process Overview:
The following is an explanation of the main tasks of the assignment, broken into 4 parts.

1. Parse command-line arguments

There are 4 possible flags specified by the user (`-f, -b, -c, -h`). When parsing the arguments, you will need to check that all required flags are specified and all flags that expect an argument are followed by an argument. You will also need to check for the error conditions outlined in the file descriptions below.

2. Print error message or execute program

After parsing the command-line arguments, the program proceeds to either print an error message from parsing the command line arguments or open up the specified anagrams file.

3. Load anagrams from anagrams file into memory

The function `loadAF()` loads the anagrams from the file into a dynamically allocated array of anagram structs. It does this by delegating to `loadAnagram()`, which reads a single anagram from the file at a time. This is repeated until the end of the file is reached.

4. Finding Anagrams

Once the array of anagrams has been created, we begin the interactive session. The user will be prompted for a word, and the program will wait until the user enters a word or exits by pressing `ctrl-D`. When a word is entered, a hash key is calculated for the user's word and a new `struct anagram` is created. The program performs a binary search on the anagrams array to find if there is a `struct anagram` with a matching hash key. Since the size of the `hashKey` can be specified by the user, there is a possibility that two anagram structs have matching hash keys, but are not anagrams. This is considered a 'collision,' and the number of collisions should be recorded if the `-c` flag is set. If the hash keys and the lowercase sorted versions of the words match, the word will be printed and the adjacent structs in the array will then be processed. Once the anagrams are printed or it is determined that no anagrams were found, the user is then prompted to enter another word.

C Functions to be Written

Listed below are the modules to be written in C.

---

pa3.c

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

This function is the main driver for the program and will be delegating a majority of the functionality to the other functions you are writing. In this function, whenever an error is encountered, immediately print the appropriate error message, print the short usage statement, and return, indicating failure.

1. Parse the command line arguments.
2. If the help flag was set, print the long usage statement to `stdout` and return, indicating success.
3. Process the anagram file. All valid anagram files must have a `.af` extension.
    a. What can you compare to check if the string entered by the user has a `.af` extension?
4. Open the anagram file in read mode. Note that `fopen()` sets `errno` in the case of an error.
    a. What do you need to do with `errno` before calling a function that uses it for error reporting?
    b. Check if there was a file permissions error, indicated by `errno` (`man -s3c fopen`).
    c. If any other errors are indicated by `errno`, the anagram file is invalid.

5. Load the anagrams into memory.
6. If the hash bits flag was set, truncate all the hash keys. By default, all of the hash keys are 32 bits.
7. In order to efficiently search for anagrams, the array of anagrams must be sorted. Consider using the `qsort()` function, and one of the comparison functions you have written to sort the array. (See the "Note on comparison functions" box below the description for `truncateHashKey.s` for more information).
8. The process of prompting the user for an anagram and searching through the anagrams array will now be handled by the `findAnagrams()` function.
9. Free all the memory that was dynamically allocated for the anagrams and close the anagram file.

Reasons for error:

Note: All `parseArgs()` related errors (see Reasons for error: in the `parseArgs.c` file description) should be printed from `parseArgs.c`. All other error strings in this program should be printed from this file, `pa3.c`. In this file, whenever an error is encountered, immediately print the appropriate error message, print the short usage statement, and return indicating failure.

- Error returned from `parseArgs()` (only print usage statement and return in this case)
- Errors returned from `loadAF()`, or `findAnagrams()`.
- Anagram file is missing `.af` extension.
- Anagram file cannot be opened due to a file permissions error.
- Anagram file is otherwise invalid (as indicated by `errno`).

Return Value: `EXIT_SUCCESS` on success, `EXIT_FAILURE` on failure.

---

parseArgs.c

```
int parseArgs( int argc, char * const argv[], struct argInfo * argInfo );
```

This function is responsible for parsing the command-line arguments and filling all the member fields of `struct argInfo` accordingly. This function must support the short options described in the long usage statement. The primary tool used to accomplish this is `getopt()`. One of the best resources for this function is the man page (`man -s3c getopt`). Make sure to use the constants provided in `pa3.h`.

Flags:

`-f`	The user is required to use this flag. Requires argument that indicates the name of the anagrams file.
`-b`	Requires argument that indicates the number of lower bits to use from the `hashKey`. This argument must be converted from a string to a long, and must be within the range [1 - 32].
`-c`	No required arguments.
`-h`	No required arguments. If this flag is set, set the appropriate bit in `argInfo`'s `flags` and return (successfully) right away.

To keep track of which optional flags were set by the user, you need to perform a bitwise operation to set each flag's corresponding bit in the `flags` member of `argInfo`.

Reasons for error:

Note: For ALL errors in this file, as soon as an error is detected, print the appropriate error message and return immediately, indicating failure. Do NOT print the usage statement from this file, this will be done in `pa3.c`.

- `getopt()` indicates an invalid flag or missing flag argument (and automatically prints the appropriate error message for you).
- `hashBits` is too large to be successfully converted to a long.
- `hashBits` contains invalid characters.
- `hashBits` is outside the required range.
- After `getopt()` completes and no other errors are found, check for these errors in the following order:
    1. No `-f` flag is parsed
    2. There are extra arguments left after `getopt()` completes.

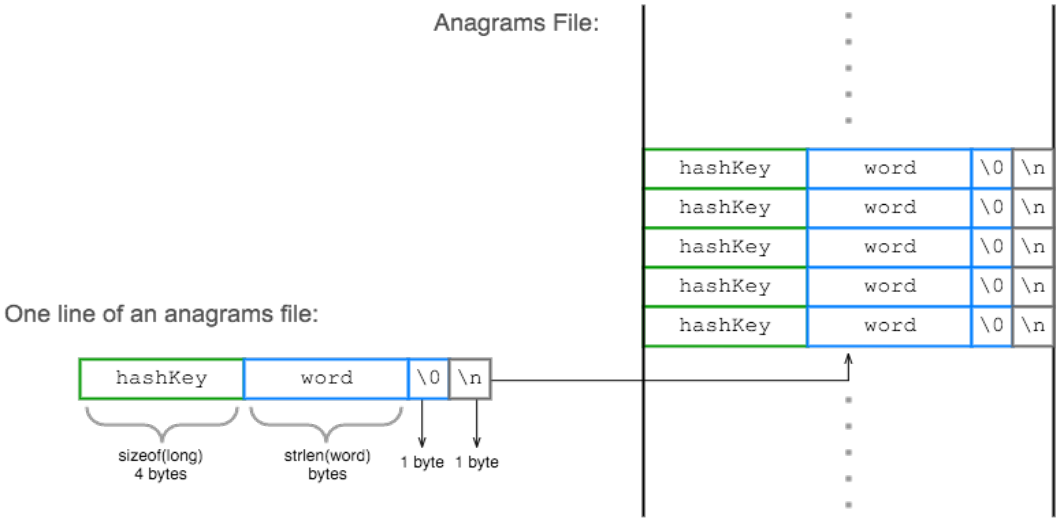
Return Value: Zero on success, nonzero on failure.

---

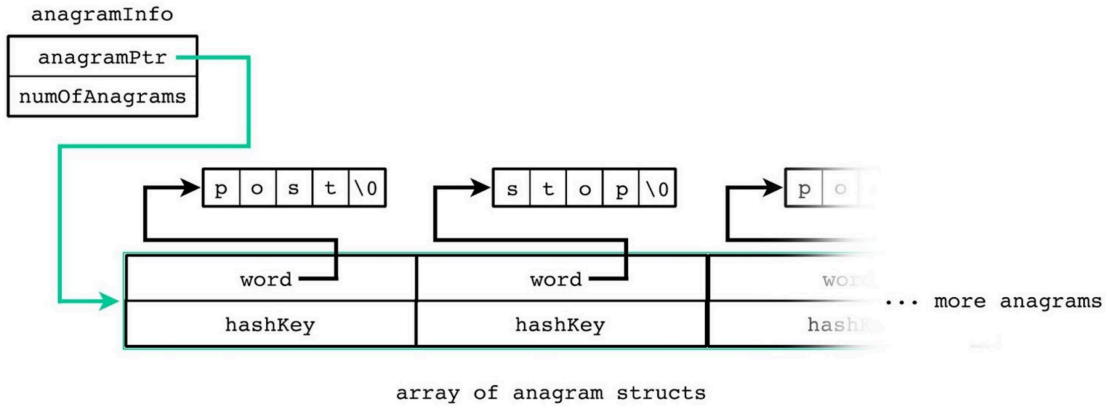
loadAF.c

```
int loadAF( FILE * stream, struct anagramInfo * anagramInfo );
```

This function will load all the anagrams from the anagrams file (`stream`) into memory, populating the passed in `anagramInfo` struct. It will repeatedly delegate to `loadAnagram()` to read one anagram at a time and append it to the dynamically allocated array of anagrams. In order to create space in the anagrams array for each new anagram read from the file, the array needs to be expanded using `realloc()` (see `man -s3c realloc`).



At the end of the function, the `struct anagramInfo` should contain a pointer to the dynamic array of struct anagrams and the number of anagrams read from the file.



Reasons for error:

- Run out of memory in `loadAnagram()` or when expanding the array of anagram structs. If this happens, `free()` all dynamically allocated memory and return right away.

Return Value: Zero on success, nonzero on failure.

---

loadAnagram.c

```
int loadAnagram( FILE * stream, struct anagram * anagram );
```

This function will load a single anagram from the anagrams file (`stream`) and will populate the fields of the passed in `anagram` struct. When populating the fields of the anagram, you need to dynamically allocate space for the `word` member, in which you will store a copy of the word read from the file.

Things to consider:

- How can you read the fixed-length hash key from the file?
- How can you read the variable-length word from the file?
- When reading data from the file, how do you know if the end of file was reached?
- File reading: `man -s3c stdio`, `man -s3c fgets`, `man -s3c fread`
- Copying memory: `man -s3c strncpy`, `man -s3c memcpy`

Reasons for error:

- Run out of memory (`malloc()` fails to allocate memory; `man -s3c malloc`). If this happens, return right away.

Return Value: Positive if an anagram is successfully read from the file, zero if the end of file was reached, or negative if an error occurs.

---

findAnagrams.c

```
int findAnagrams( const struct argInfo * argInfo,
                  const struct anagramInfo * anagramInfo );
```

This function prompts the user to enter a word to search for anagrams of. It then gets the word input from the user (from `stdin`) and prints any matching anagrams. This happens in a loop until the user enters `ctrl-D` (`^D`) to signify the end of input (`EOF`), or an error occurs when creating an anagram.

Interacting with the user:

1. Read the word entered by the user (`man -s3c fgets`). Note that `fgets()` will include the newline character as part of the string read, so replace the newline character with the null character (`man -s3c strchr`).
2. Create a new anagram from the word entered by the user. If an error is detected, return right away.

3. If the hash bits flag was set, truncate the newly created anagram's `hashKey` accordingly.
4. Find and print all matching anagrams in the array of anagrams stored in the `anagramInfo` struct.

    a. Search for a matching anagram (based on the `hashkey`) in the array of anagrams. See the "Note on comparison functions" box below the description for `truncateHashKey.s` for more information on using `bsearch()` with one of the comparison functions.

    b. If a match is found, print all of the adjacent struct anagrams with **matching `hashKey` values** and **matching lowercase-sorted `words`** in alphabetical order (going left to right this happens implicitly as the array is sorted on the `word` member as a secondary comparison in the `anagramCompare()` that was used with `qsort()`). Note that the initial `struct anagram` found as a match could exist at any point within the contiguous block of matching structs.

    c. When we find a matching `hashKey`, we also need to check the lowercase-sorted version of the `word` because it is possible that two anagram structs can have the same `hashKey` but are not actually anagrams of each other (meaning their lowercase-sorted `word`s don't match). We will define this situation as a **collision**. If the collision flag is set, we need to keep track of and print the number of collisions that occurred after printing out the anagrams of the entered word. (If no anagrams were found, we still want to print out the number of collisions if the flag was set).

Reasons for error:

- Run out of memory when creating an anagram (check return value from `createAnagram()`).

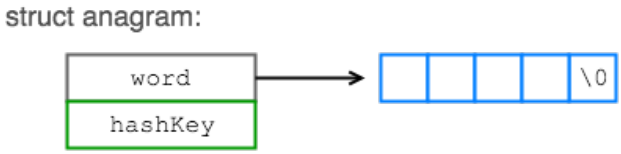
Return Value: Zero on success, nonzero on failure.

---

createAnagram.c

```
int createAnagram( const char * src, struct anagram * anagram );
```

This function will create a new anagram by populating the passed in `struct anagram`. An anagram has two components: the `word`, and the `hashKey`.



`word`	A copy of the `src` passed in. You will need to dynamically allocate space for the `word`.
`hashKey`	Create the `hashKey` by using the `hashString()` function with the lowercase-sorted version of the `word`. By using the lowercase-sorted version of the `word`, we ensure that if two words are anagrams, they will have the same `hashKey`.

Reasons for error:

- Run out of memory (`malloc()` fails to allocate memory; `man -s3c malloc`).

Return Value: Zero on success, nonzero on failure.

---

lowerCaseSortString.c

```
void lowerCaseSortString( const char * src, int n, char * dest );
```

This function will take the `src` string (of length `n`) and will store a copy in `dest` where all the characters are converted to lowercase and sorted in alphabetical order.

Things to consider:

- How can you copy a string of `n` characters? (`man -s3c string`)
- See the "Note on comparison functions" box below the description for `truncateHashKey.s` for information that will help you sort the characters in alphabetical order (using `qsort()` and `charCompare()`).
- `man -s3c tolower`

Return Value: None.

---

usage.c

```
void usage( FILE * stream, enum usageMode u, const char * progName );
```

Print the appropriate usage message to the specified `stream`. If the `usageMode` is `USAGE_SHORT`, print the short usage message. Otherwise, print the long usage message (see `pa3.h` and `pa3Strings.h`).

Return Value: None.

Assembly Functions to be Written

Listed below are the modules to be written in Assembly.

---

hashString.s

```
long hashString( const char * str );
```

This function will be used to create the hash key of an anagram. This function creates an integer (long) hash key from `str` by using an algorithm similar to Java's `String.hashCode()`, given below. Use the global variables defined in `pa3Globals.c` to access these constants in assembly.

```
long hash = HASH_START_VAL;
int strLen = strlen(str);
for( int i = 0; i < strLen; i++ ) {
     hash = hash * HASH_PRIME + str[i];
}
return hash;
```

Return Value: The hash key of `str`.

---

truncateHashKey.s

```
long truncateHashKey( long hashKey, long numBits );
```

In this module we want to truncate the `hashKey` by extracting the lower number of bits specified by `numBits`. If the `hashKey` is stored in a 32-bit long, what are some bitwise operations we can perform to remove the unwanted upper bits?

Return Value: Truncated `hashKey`.

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 3 comparison functions). The comparison function must take in two `const void *`'s as parameters.

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

- Return -1 if the first argument is less than the second
- Return 0 if the arguments are equal
- Return +1 if the first argument is greater than the second

Refer to `man -s3c qsort` and `man -s3c bsearch` for more information.

---

charCompare.s

```
int charCompare( const void * p1, const void * p2 );
```

This function takes two pointers to characters (the prototype uses two `void` pointers, but it can be assumed that they are `char` pointers) and compares them. **Do not use subtraction**; in the milestone unit testing we will check for exactly -1, 0, or 1. This function must be a leaf subroutine.

Return Value: -1 if the first `char` is smaller, 0 if the `char`'s are the same, or +1 if the first `char` is larger.

---

hashKeyMemberCompare.s

```
int hashKeyMemberCompare( const void * p1, const void * p2 );
```

This function takes two pointers to struct anagrams (the prototype uses two `void` pointers, but it can be assumed that they are `struct anagram` pointers) and compares their hash keys. **Do not use subtraction** to compare the hash keys, as this can cause overflow. Use the offsets made available in `pa3Globals.c`.

Return Value: -1 if the first anagram's hash key is smaller, 0 if the anagrams' hash keys are the same, or +1 if the first anagram's hash key is larger.

---

anagramCompare.s

```
int anagramCompare( const void * p1, const void * p2 );
```

This function takes two pointers to struct anagrams (the prototype uses two `void` pointers, but it can be assumed that they are `struct anagram` pointers) and compares their hash keys. If their hash keys are the same, it then compares their words (how can you compare two strings? [`man -s3c string`]). Use the offsets made available in `pa3Globals.c`.

Return Value: -1 if the first anagram is smaller, 0 if the anagrams are the same, or +1 if the first anagram is larger.

---

isInRange.s

```
int isInRange( long minRange, long maxRange, long value, long exclusive );
```

Copied from PA1, no changes necessary.

Unit Testing

You are provided with basic unit test files for the Milestone functions of this assignment. These have minimal test cases and are only meant to give you an idea of how to write your own tests. **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 5 unit test files. 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 and running these tests. Keep in mind that your unit tests will not build until all required files for the unit tests have been written. These test files are not being collected for the Milestone and will only be collected for the final turnin (however, they should already be written by the time you turn in the Milestone because you should be using them to test your Milestone functions).

Unit tests you need to complete:

```
testtruncateHashKey.c
testhashString.c
testcharCompare.c
testlowerCaseSortString.c
testparseArgs.c
```

To compile:

```
$ make testcharCompare
```

To run:

```
$ ./testcharCompare
```

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

README File

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing "make"--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ???).

- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) showing your tests covered all parts of your code (test coverage). (Be more specific than diff'ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

Questions to Answer in the README

1. How do you maintain your integrity even when you're stressed, pressured or tired?

Extra Credit

There are 5 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)
- **[2 Points]** Modify your program to print out the average number of anagram set collisions when the `-c` flag is set. An anagram set is the set of all words that are anagrams of one another. An anagram set collision is when two different anagram sets map to the same `hashKey`.

  To calculate the average number of anagram set collisions, divide the total number of anagram set collisions (stored as an `int`) by the total number of unique hash keys (stored as an `int`). Cast each total to a `float` when dividing to calculate the average as a `float`. The average must be calculated as a `float` to receive credit.

  You will calculate and print your results in a new file named `printHashStatsEC.c` that will be called a single time immediately before the call to `findAnagrams()` in `pa3EC.c` ONLY if the `-c` flag is set. You will print your results using the exact string:

  `"\nAverage number of set collisions: %.4f\n"`.

  The argument to `%.4f` is the average number of set collisions as a `float`. Again, you must use this EXACT string or you will not get credit.
- **[1 Point]** Modify your program to <u>not</u> print out an exact match of the word entered for search. This exact match is case sensitive, for example "Stop" is not an exact match with "stop".

If you are attempting any of the extra credit modifications, you should create four new files: `pa3EC.c`, `pa3EC.h`, `findAnagramsEC.c`, and `printHashStatsEC.c`. All the the new files except for `printHashStatsEC.c` should be copies of your existing files with modifications to implement the extra credit. You do not need to implement all of these files (i.e. all of the extra credit), they just need to be present. If all of these files are not present we will not be able to compile your extra credit executable and you will not receive any points for the extra credit.

A target has been added to the `Makefile` that can be run with `'make pa3EC'` to build the extra credit executable, `pa3EC`. Be sure to `'make clean'` before doing so if you are getting strange compile errors.

A reference solution implementing the extra credit has been provided in the public directory called `'pa3refEC'`. Your extra credit executable's output will need to match this program's for the extra credit part(s) that you implement.

Milestone Turn-in Instructions

Milestone Turn-in - due Wednesday night, May 11 @ 11:59 pm [15 points of Correctness Section]

Before final and complete turnin of your assignment, you are required to turnin several modules for the Milestone check.

Files required for the Milestone:

`truncateHashKey.s`	`hashString.s`	`charCompare.s`
`lowerCaseSortString.c`	`parseArgs.c`	

Each module must pass all of our unit tests in order to receive full credit.

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All Makefile test cases for the milestone functions must compile successfully via the commands `make test***`.

In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script.

```
$ cse30_pa3milestone_turnin
```

To verify your turn-in:

```
$ cse30verify pa3milestone
```

Final Turn-in Instructions

Final Turn-in - due Tuesday night, May 17 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README file (see above), you are ready to turn it in. Before you turn in your assignment, you should do `make clean` in order to remove all the object files, lint files, core dumps, and executables.

Files required for the Final Turn-in:

`anagramCompare.s`	`loadAF.c`	`pa3Strings.h`
`charCompare.s`	`loadAnagram.c`	`parseArgs.c`
`createAnagram.c`	`lowerCaseSortString.c`	`truncateHashKey.s`
`findAnagrams.c`	`pa3.c`	`usage.c`
`hashKeyMemberCompare.s`	`pa3.h`	`Makefile`
`hashString.s`	`pa3Globals.c`	`README`
`isInRange.s`		
`testtruncateHashKey.c`	`testcharCompare.c`	`testparseArgs.c`
`testhashString.c`	`testlowerCaseSortString.c`	

Use the above names *exactly* otherwise our Makefiles will not find your files.

How to Turn in an Assignment

Use the following turnin script to submit your full assignment before the due date as follows:

```
$ cse30turnin pa3
```

To verify your turn-in:

```
$ cse30verify pa3
```

Up until the due date, you can re-submit your assignment via the scripts above. Note, if you turned in the assignment early for extra credit and then turned it in again later (after the extra credit cutoff), you will no longer receive early turn-in credit.

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. Late assignments WILL NOT be accepted.

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.

Style Requirements

You will be graded on style for all the programming assignments. The requirements are listed below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like non-integers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef ... #endif).
- Never have hard-coded magic numbers (any number other than -1, 0, or 1 is a magic number). This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.