Problem Set 6: Searching

Please send back via NYU Classes

- A zip archive named as
  `PS06_<your name as Last_First>.zip`
  containing the C code files that implements all aspects of all problems.

Total points: 100

Points are awarded as follows:

- (20 points): Parsing command line and reading data
- (20 points): Linear Search
- (20 points): Binary Search
- (20 points): Hash Search
- (5 points): Time Complexity
- (15 Points): Clear code, sensible formatting, good comments

In this problem set, you will write a program that searches a phone book containing 500 names and returns the information associated with a last name entered at the terminal. Your program will implement three search methods:

- Linear Search
- Binary Search
- Hash Search

This scenario is somewhat of a "toy" problem, in that a 500-name phone book is small, and the user must enter the exact last name string. However, it is sufficient to illustrate how the three searching methods work.

Your program will be structured into 3 files:

- **`main.c`** (portions provided by instructor) This provides a framework that can be used for all searching methods
- **`main.h`** (provided by instructor – no need to add anything) parameters and data structures that are used by all files in the program
- **`hash.c`** (provided by instructor – no need to add anything) This code sets up the tables for hash search

This program is structured as multiple files, and we can compile and link it using the following command line:

```
gcc –o search main.c hash.c
```

You will find this in bash shell script **`build.sh`**.

(20 points): Parsing command line and reading data

Your program must have its command-line arguments as follows:

```
./search linear|binary|hash phone_book_file.csv
```

The | character indicates a choice amongst possible arguments.

`linear` specifies linear search
`binary` specifies binary search
`hash` specifies hash search
`phone_book.csv` is a file supplied by instructor containing an ASCII text, in the comma separated values format as <Last>,<First>,<PhoneNumber>

a. Parse the command line. Write code in **`main.c`** that parses the command line.

- If your program is executed without any command-line arguments, or with an incorrect number of command-line arguments, it should print a "usage" message and then exit.
- Determine if linear, binary, or hash is present and set the variable **`mode`** accordingly.
- Open the input file. Print an error with a diagnostic message if the file cannot be opened.

b. Initialize data structure. Your **main()** contains a data structure declaration:

```
struct PhoneBook phone_book[PB_LEN]
```

Initialize the data structure so that all strings are empty and the chain field that will be used in the hashed search has the value -1 (which is not a valid chain value), as shown here:

```
for (i=0; i<PB_LEN; i++) {
    phone_book[i].first[0] = 0;
    phone_book[i].last[0] = 0;
    phone_book[i].phone[0] = 0;
    phone_book[i].chain = -1;
}
```

c. Read phone book file. Read each line from the input file and enter the data into the phone book, one line per struct array element. The phone book file is comma separated value format as <Last>,<First>,<PhoneNumber>. Note that in this struct phone number is a string.

After all data has been read, close the input file and optionally print the phone book, depending on the value of **`db_flag`**.

Using **`strncpy`**() is recommended for filling in the string fields, with the "count" field set to NAME_LEN or NUM_LEN, as appropriate. Adding a NULL character as the last character array will ensure that the string is always NULL-terminated.

(20 points): Linear Search

Implement linear search, by filling in code for the function in **search.c**:

```
int linear_search(char *target, struct PhoneBook *phone_book, int num_entries)
```

The first argument is the last name to search, provided by the user in the terminal, and the second and third arguments specifies the phone book array and its length.

Note that in all searches, the search target is always the last name.

When an entry is found, return the index of the entry. In addition, be sure to treat the case in which the entry is not found, in which case the function should return a value of -1.

(20 points): Binary Search

Implement binary search, by filling in code for the function in search.c:

```
int binary_search(char *target, struct PhoneBook *phone_book, int num_entries)
```

This function uses the same arguments as for linear search, but requires that the phone book is ordered by the last name. The sorting is already implemented in the provided **`main.c`** using **`qsort`** and **`strncmp()`** in the sort comparison function.

When an entry is found in **`binary_search(),`** return the index of the entry. In addition, be sure to treat the case in which the entry is not found, in which case the function should return a value of -1.

(20 points): Hash Search

Implement hash search. You may want to review the lecture slides for hash searching.

You are provided by the instructor:

```
int hash_func(char *s)
```

This function returns an integer, and tries to produce different values for even slightly different strings. For example, the hash value of "aa" is 6208, and of "aaa" is 14369. In the hash map, these numbers will be used as indices of an array, so as to quickly access the information corresponding to the search target string.

```
void create_hash_map(struct PhoneBook *phone_book, int num_entries)
```

In this function, the phone_book[] data structure array is set up to use chaining to resolve hash collisions, using the `chain` member of `struct` PhoneBook.

For each last name in the phone book, the function `create_hash_map`() calls `hash_func()` using the last name string. That is, for each `i`, it uses `phone_book[i].last` as the argument to `hash_func(). It` uses the value `j` returned from `hash_func()` as the index in the static int array `hash_map[]`. The array value for that index `j` is the phone book index `i` for the last name that is being hashed, that is, the name in `phone_book[i].last`

For a given hash value `j`, if the value in `hash_map[j]` is already "taken" (unused entries were initialized to -1), then this is a "hash collision." It uses the chain feature to resolve the hash collision conflict, as:

- Given `j = hash_func(phone_book[i].last)`
- If (get_hash_map_value(j) `== -1)` then it sets `hash_map[j] = i` and is done.
- Otherwise, `k == hash_map[j],` and this is a hash collision.
- It inspects `phone_book[k].chain.` If this is -1, it sets `phone_book[k].chain = i` and it is done.
    - Otherwise, it follows the chain until it finds a chain value = -1, and sets it equal to `i`.

The code provided by the instructor prints statistics of the hash map. It should look like:

```
Hash table statistics:
  2.99% map occupancy; top map entry 16290 out of 16384 entries
Collisions: 10 ( 0.06%)
Average chain length 1.00
```

You must implement hash search. Complete the function:

```
int hash_search(char *target, struct PhoneBook *phone_book)
```

This should:

- Calculate the hash value for the target last name that was entered at the terminal console. Use your function hash_func() for this.
- Get the entry in hash_map[] using the hash value as the index. Note: since hash_map[] is declared using the modifier *static*, if is not visible outside of the file hash.c. Therefore, you must use the supplied function get_hash_map_value() to get the hash_map[] value. This is the index into the phone book data structure. Check the value of chain at that index. If it is -1 (no collision) then you are done, just return the index.
- If not -1 (that is, if there was a collision for this hash value):

- Check to see if the current last name is the one that you are looking for. If so, then return the index.
- Otherwise, follow the chain index to the next structure phone book array, and check to see if the current last name is the one that you are looking for. Repeat until the desired entry is found. Return the index of that entry of the phone book array.
- If you get to a chain value of -1 without finding the desired entry, it means there are no such last name in the phone book. Return -1.

(5 points): Time Complexity

What are the Big-O time complexities of the three search algorithms? What are the tradeoffs of binary search and hash search, for being faster than linear search?

Put your answers in file **TimeComplexity.txt** and submit this file along with your C source code files.