COSC 1437 – GE2

Part 1: This program is based on Chapter 8 Programming Challenge 8 (Search Benchmarks) on Page 488. Use the following file names for this part: C08PC18.CPP and C08PC18.EXE.

Write and test **five** separate functions, one for each of the following **five** algorithm steps. Make sure each function works correctly before trying Step **6**.

Algorithm Step **1**: Use the rand function to populate an int array passed as the first parameter with 1000 unique integers in the range 0 to 9999. The size of the array should be passed as the second parameter. Note that the array is an OUT parameter and the size is an IN parameter.

Algorithm Step **2**: Randomly generate an array index value from 0 to 999. Use this value as an index into the array passed in as the first parameter. Return the indexed array value using a return statement. The array is the first parameter (IN parameter) to the function. The size of the array is passed as the second parameter. The value returned from this function will be your search key for Step 3 and Step 5.

Algorithm Step **3**: Use a linear search to locate the key. Keep track of the number of comparisons needed to find the key. The array is the first parameter, the array size is the second parameter, and the search key is the third parameter. All parameters are IN parameters. Return the number of comparisons. On average the number of comparisons should be about 500.

Algorithm Step **4**: Sort the array in ascending order. The array is the first parameter (IN/OUT). The size of the array is the second parameter (IN).

Algorithm Step **5**: Use the binary search algorithm to find the key in the array. Keep track of the number of comparisons needed to find the key. The array is the first parameter, the array size is the second parameter, and the search key is the third parameter. All parameters are IN parameters. Return the number of comparisons. On average the number of comparisons should be about 9.

Step **6**: Write a main function. This function should seed the random number generator based on the system time. Then set up a loop that loops 1000 times. In the loop body, call each of the **five** functions, accumulating data on the number of comparisons for the linear search and the number of comparisons of the binary search.

When the loop ends, display a summary as follows:

After 1000 tests on 1000 element arrays, the linear search results were:
The minimum number of comparisons to find the key was: NNN
The maximum number of comparisons to find the key was NNN.
The average number of comparisons to find the key was: NNN.N

After 1000 tests on 1000 element arrays, the binary search results were:
The minimum number of comparisons to find the key was: NNN
The maximum number of comparisons to find the key was NNN.
The average number of comparisons to find the key was: NNN.N

Part 2:
This batch mode program reads and processes a text file using command line input redirection. The program will also use a command line argument to select the sort order.

For example, this program can be run from the command line as in the examples below:

```
GE2Part2.exe asc < TestFile1.txt > GE2Part2NameAsc.txt
GE2Part2.exe des < TestFile1.txt > GE2Part2NameDes.txt
```

The first line in the file used by this program is an unsigned int value indicating the number of lines of data that follow. Each additional line in the file contains a student name. The format of the text file is as follows:

100
Smith, John J.
Adams, Susan Ann

The student name always starts at the beginning of the line. Assume the name may be up to 30 characters including whitespace characters but not counting the end of line characters. Lines are terminated with CR and LF characters (Windows style text files created using Notepad).

The program should do the following:

0. Process and save the command line argument for later use.

1. Read the number of data items from the file.

2. Dynamically allocate one array on the heap. This dynamically allocated array is an array of pointers and each pointer in the array will point to one student name stored in a dynamically allocated string object.

3. Loop while there are still data lines to be read from the file:

3-1. Read a line from the file into a temp string variable

3-2. Copy the temp variable into a dynamically allocated string object.

3-3. Place the address of the dynamically allocated string object in the array of pointers.
//can't dynamically allocated a string object unless it is a pointer
//might as well pass directly to pointer, unless the string is supposed to be a 1-d array.

3-4. End of Loop

4. Based on the command line argument, sort the array of pointers using the proper sort order. Note that only the pointers in the array of pointers are rearranged. The string objects and the names stored inside the string object do not get modified.

5. Send the sorted results to standard output. The output format should be similar to the input format except the items should be properly sorted. It should be possible to use the output file created by this program as the input file to this program. The code that sends results data to standard output should be encapsulated into one function to make code porting to other environments easier.

6. Free all dynamically allocated memory. You must use a loop to free the memory for the dynamically allocated string objects and then use a single statement to free the dynamically allocated array of pointers.

How to approach this problem:

1. Take the high level algorithm given above and refine it by adding the necessary detail so that you can partition your program into functions.

2. Create a structure chart showing the required communications between your functions.

3. Write the declarations for your functions making sure the parameters and return types agree with your structure chart.

4. Use a bottom up approach (or combination approach) to implement and test each function individually.

5. Once all functions have been verified to work, add them to the final program and test everything together.