Operating Systems (CS:3620)

Assignment 1 Total points: 100

Due: 2 Weeks (by 11:59 pm of September 12, 2016)

This assignment will contribute 10% to your final grades.

Submission: Please submit your source code through ICON. Zip all the source code before submission. Please make sure you name the zip file in the following way:
Grading: Please make sure your source code compiles with GCC, otherwise you will not obtain any points. Please comment your source code so that it is easy for the TA to understand it. Your program will be graded automatically against some TA-generated test cases.
Cheating and Collaboration: *This is an individual project, you can discuss with your peers but cannot copy source code. Please do not copy source code from Internet. Think about the worst-case scenario when you can get caught.*

1. (40 points) **Binary Record Sorting in Ascending Order**— You will write a simple sorting program. This program should be invoked as follows:

```
shell% ./fastsort -i inputfile -o outputfile
```

The above line means the users typed in the name of the sorting program ./fastsort and gave it two inputs: an input file to sort called inputfile and an output file to put the sorted results into called outputfile .

Input files are generated by a program we give you called generate.c (this file will be provided to you).

After running generate , you will have a file that needs to be sorted. It will be filled with binary data, of the following form: a series of 100-byte records, the first four bytes of which are an unsigned integer key, and the remaining 96 bytes of which are integers that form the rest of the record.

Something like this (where each letter represents two bytes):

kkRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRR

See the following figure for details.

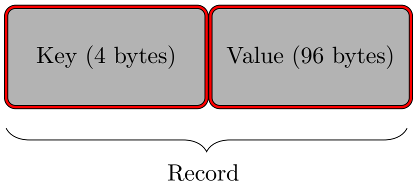

Figure 1: Format of one binary record

Your goal: to build a sorting program called fastsort that takes in one of these generated files and sorts it based on the 4-byte key (the remainder of the record should of course be kept with the same key). The output is written to the specified output file.

Some Details

Using generate is easy. First you compile it as follows:

```
shell% gcc -o generate generate.c -Wall -Werror
```

Note: you will also need the header file sort.h to compile this program.

Then you run it:

```
shell% ./generate -s 0 -n 100 -o /tmp/outfile
```

There are three flags to generate . The -s flag specified a random number seed; this allows you to generate different files to test your sort on. The -n flag determines how many records to write to the output file, each of size 100 bytes. Finally, the -o flag determines the output file, which will be the input file for your sort.

The format of the file generated by the generate.c program is very simple: it is in binary form, and consists of those 100-byte records as described above. A common header file sort.h has the detailed description.

Another useful tool is *dump.c* (it will be given to you). This program can be used to dump the contents of a file generated by generate or by your sorting program.

Hints: In your sorting program, you should just use `open()`, `read()`, `write()`, and `close()` to access files. See the code in generate or dump for examples.

If you want to figure out how big in the input file is before reading it in, use the `stat()` or `fstat()` calls.

To exit, call `exit()` with a single argument. This argument to `exit()` is then available to the user to see if the program returned an error (i.e., return 1 by calling `exit(1)`) or exited cleanly (i.e., returned 0 by calling `exit(0)`).

The routine `malloc()` is useful for memory allocation. Make sure to exit cleanly if `malloc` fails!

If you don't know how to use these functions, use the man pages or . For example, typing `man malloc` will provide you information about how to use the malloc function.

Assumptions and Errors:

32-bit integer range. You may assume that the keys are unsigned 32-bit integers.

File length: May be pretty long! However, there is no need to implement a fancy two-pass sort or anything like that; the data set will fit into memory. Invalid files: If the user specifies an input or output file that you cannot open (for whatever reason), the sort should EXACTLY print: Error: Cannot open file foo , with no extra spaces (if the file was named foo) and then exit.

Too few or many arguments passed to program: If the user runs fastsort without any arguments, or in some other way passes incorrect flags and such to fastsort, print Usage: fastsort -i inputfile -o outputfile and exit.

Important: On any error code, you should print the error to the screen using fprintf() , and send the error message to stderr (standard error) and not stdout (standard output). This is accomplished in your C code as follows:

```
fprintf(stderr, ``whatever the error message is\n'');
```

2. (10 points) **Big Integer Addition**— As we have seen in the class, `int` datatype can hold an integer of size 4 bytes. We have also seen the type `long long int` which can hold up to 8-byte integer. Often times, while performing cryptographic operations (e.g., encryption, decryption) you require to `add` or `multiply` integers that are 1000 digits long. Your goal is to implement the addition operation of such big integers. Note that, these integers will not fit in a typical primitive data type. We are required to store these big integers in some other form. For instance, one can store them in a char array where each digit of such numbers occupy one position in that array. You will be given a skelton C file which will deal with taking such a pair of big integers as input. Your objective is to implement the addition function and the print function to output the result. **You can safely assume that the number of digits in these big integers are less than 10000**. **You can also safely assume the numbers are positive**.

3. (15 points) **Big Integer Multiplication**— In this problem, you will extend the previous problem by implementing the `multiplication` operation of big integers. Note that, these integers will not fit in a typical primitive data type. We are required to store these big integers in some other form. For instance, one can store them in a char array where each digit of such numbers occupy one position in that array. You will be given a skelton C file which will deal with taking such a pair of big integers as input. Your objective is to implement the multiplication function and the print function to output the result. **You can safely assume that the number of digits in these big integers are less than 10000**. **You can also safely assume the numbers are positive**.

4. (35 points) **Dictionary Search**— A lot of times we are required to search for words in a large dictionary. This problem requires you to implement the insert and search functionality in a large dictionary. In this problem, you will be given a large dictionary file. **Each input line of this dictionary will contain a word whose length will be maximum 64 characters**. To make search efficient (i.e., $\mathcal{O}(\log n)$), you will require to order the dictionary as a binary search tree. You will need to implement the function responsible for creating a binary search tree from a given dictionary file. You do not have to be worried about taking inputs because you will be provided a skeleton file that reads words from the dictionary file. You will then be required to implement a function that takes a word as input and searches the tree to see whether the word can found in the dictionary. Your function should return *true* if the word can be found in the dictionary or else it should return *false*. In addition, you will need to implement 3 functions to print the binary search tree in a pre-, in-, and post-order form. **You can consult a data-structure textbook to determine how to add and search in a binary search tree**.