Project Description

What to Submit: you should submit as a single compressed ýle (i.e., zip):
(i) a PDF ýle with your report, and
(ii) your C code (properly commented)

A. Description. In this project, we will build a simplistic in–memory database accessible through a naive command–line text–based user interface (i.e., the user types commands in order to interact with the in–memory database). Your interface should interpret script files, in addition to accepting one command at a time from the keyboard, and use a user–level library that implements the in–memory database.

We will start by describing the user interface. The executable of your program must be named `naivedb`. Your `makefile` must ensure this. The `naivedb` program must support the following usage: `naivedb [scriptfile]`. When no arguments are given, `naivedb` should enter a loop in which it accepts one command at a time as keyboard input (e.g., using `scanf`). The user types a command, and then presses 'return' on the keyboard to execute it (i.e., the command will end with a newline character). Your program **must block** until the command completes and, if the return code is abnormal, print out a message to that effect. Alternatively, your program should open the given file (if one is provided) and interpret the contents as a sequence of commands to execute. You may assume that each line of the script file corresponds to one command.

Your user interface should accept and support the following commands:

- `quit`: When this command is encountered, `naivedb` should stop processing commands, prompt the user for confirmation (i.e., "Are you sure you want to exit? All files will be lost! Y/N"), accept the user input, delete all data structures and intermediate files (if any) created, and exit.
- `Output redirection`: Your `naivedb` program should allow output to be redirected to a file (e.g., executing the command `srchindx -o ALT indx.txt flightdata > foo.txt` should execute `srchindx` in the flightdata directory and print its output in file foo.txt).
- `create`: This command is used for creating new files, directories, and links. This is a separate program that can be executed by `naivedb`, or can be executed from a terminal by running its executable file. The `create` program should support the following usage:
  - `create -f filepath` creates an empty, ordinary file whose name is given in the specified path. The path can be an absolute pathname, a relative pathname, or just a bare filename, in which case the file should be created in the current directory. The new file should have permission 0640 specified in octal.
  - `create -d dirpath` creates a new directory whose name is given in the specified path. The new file should have permission 0750 specified in octal.

  - `create -h oldname linkname` creates a `hard` link. oldname is the path or name of an existing file, and linkname is the path or name of the hard link to be created.
  - `create -s oldname linkname` creates a `symbolic` (i.e., soft) link. oldname is the path or name of an existing file, and linkname is the path or name of the symbolic link to be created.

- `fileconverter`: This command is used for converting a binary file containing flight data records into a set of text files. You can assume that the binary file has been produced error-free and that records in the binary file correspond one-to-one to records (one for each line) in the text file. The `fileconverter` program should support the following usage: `fileconverter infile outdirectory`, where parameter `infile` is the name of the input file, and `outdirectory` indicates the directory where the output files are to be stored.

  You may assume that each line of the output text file contains information about a single flight in the format: `AirlineCodeFlightNumber OriginAirportCode DestinationAirportCode <DepartureDate>`, where the four items are separated by single whitespace characters and the format of <DepartureDate> is `YYYY-MM-DD HH:MM`. Note that this format is easily comparable. For example, a record would look like "AA43 DFW DTW 2016-01-06 11:00", with "2016-01-06 11:00" being the `DepartureDate`.

  Given a binary input file, your `fileconverter` program should produce a collection of text files (using the `create` command), such that each file contains only flights operated by a single airline. The set of files should be stored in `outdirectory`. For example, all American Airlines flights should be stored in `AA.txt`, whereas all Delta flights should be stored in `DL.txt`. Additionally, flight records in a text file should be sorted in order of <DepartureDate>.
- `indexer`: This command is used to parse a set of files and create an inverted index, which maps each airport code found in the files to the subset of files that contain that code. Your inverted index should additionally maintain the frequency with which each airport code appears in each file. The inverted index should be maintained in **sorted** order by airport code. The executable version of your `indexer` program must support the following usage: `indexer [inverted-index file name] directory`, where the first argument, `inverted-index file name`, is the name of a file that your program should create to hold the inverted index. The second argument, `directory`, is the name of the directory that contains files your indexer should index. If only a directory is provided as argument (i.e., the first argument is a directory), your `indexer` program should operate on the specified directory and generate the default index file "`invind.txt`". You may assume that files in a directory cannot have the same name. The index file should be stored in the following format:

  ```
  <list> term
  name1 count1 name2 count2 name3 count3 name4 count4 name5 count5
  </list>
  ```

  For example,

  "ATL" $\rightarrow$ ("AA.txt", 10), ("DL.txt", 3)

  "DFW" $\rightarrow$ ("DL.txt", 1)

Note that the above depiction provides a *logical view* of the inverted index. In your program, you have to define appropriate *data structures* to hold the mappings (i.e., airport code → list of files), the list of records, and the records (file name, count). After constructing the entire inverted index in memory, only then should your program save it to a file.

Additionally, the mapping is to be maintained in *ascending* sorted order based on the ASCII coding of characters (i.e., "A" before "B" and "AA" before "AB"), whereas records in each list are to be maintained in *descending* sorted order based on frequency counts of the terms in the files.

- `srchindx`: This command is used to perform a search over a file or a directory using an inverted index file. Specifically, your `srchindx` is to support the following usage: `srchindx flag terms [inverted-index file name] [path]`, where the argument `inverted-index file name` denotes the name of the file that your `indexer` has created to hold the inverted index, and [path] is the filename to a file or directory. Given a filename, your program should search for `terms` in that filename only. Given a directory, your program should search for `terms` in all files in the directory (and subdirectories, if any) recursively. `terms` is a list of terms to be used for search and can be 1 or more. If more than one terms are provided, they will be separated by semicolons (e.g., `ALT;DFW`). Argument `flag` indicates if search is to be performed on origin (`-o`) or destination (`-d`) airports accordingly. For example,
  - o Execution of command `srchindx -o ALT;DFW indx.txt flightdata`, should search for records with origin airports ALT or DFW. The search should be performed across all files under directory flightdata using the inverted index stored in file indx.txt
  - o Issuing command `srchindx -d ALT flightdata`, should result in search of records with destination airport ALT in all files under directory flightdata using the inverted index stored in file invind.txt (i.e., the default filename for the inverted index).

  To facilitate efficient search, your program is to maintain **in memory** the following two data structures:
  - – A **hash table** to store records from the inverted file created by `indexer` associated with **origin** airports. The key will be the origin airport itself. Additionally, you will use chaining, according to which, each element of the array is the head of a linked list. At index y in the array, the linked list contains all of the objects whose keys have hash value y. Thus, to insert an element O with key k into the hash table, the following steps need to be performed: (i) a new linked list node N needs to be constructed to store O, (ii) the hash value $y = h(k)$, where $h()$ is a given hash function, must be computed, (iii) if $table[y]$ is null (i.e., the list at location y is empty), then simply set $table[y] = N$, otherwise, insert N into the existing linked list stored in $table[y]$.
  - – Given a **destination** airport, your program should construct a **binary search tree** (BST), whose nodes store information about each airline flying to that airport, and the number of flights each airline operates in that airport. Specifically, each node has a key (i.e., an airline with at least one flight to the given airport), one data value (i.e., the

number of flights to the airport), a left child, and a right child (the children are also nodes). At the "top" of the tree is a special node, called the root, which is not the child of any other node. Your BST should keep keys in sorted order, so that lookup and insertion can use the principle of binary search. Specifically, given a node N with key k, the left subtree of N should contain only nodes whose keys precede k, while the right subtree of N should contain only nodes whose keys succeed k in lexicographical order.

Given a set of terms, your program should construct a hash table and a binary search tree **for each of the terms** by inserting information from files as listed by the `ls` program. Your program should not do any tree rebalancing, and your hash table size will be a function of your teammebers ids. Specific instructions about the derivation of the hash table size will be provided around the time of your progress report.

For each of the terms provided in the input, your program should print to `stdout`, the following:

- The contents of the hash table, with one array element per line. Each array element is a list of nodes at that index. Each node should display the origin airport and number of airlines operating from that airport. Each line should be of the form: `index: listNode1` → `listNode2` → ⋯ → `listNodeN` → `NULL`.
- An in–order traversal of the BST. For each node, output the airline code.

Extra Credit: Plot the execution time (in seconds) of your `srchindx` program to perform a search given a set of airports. Specifically, draw the following figures:

(i) Running time of your program with and without BST as a function of the number of (a) airlines, and (b) airports in the database.

(ii) Running time of your program with and without hashtable as a function of the number of (a) airlines, and (b) airports in the database.

Does the running time of your `srchindx` program change with the size of the set of airports provided as input? Try passing in (a) just 1 airport, (b) 10 airports, and (c) 100 airports, and report your observations.

B. Error Handling. For cases not covered by this specification, you may specify and implement a reasonable behavior. Additionally, your program must detect the following fatal errors.

- A wrong number of command line arguments is provided. In this case, your `naivedb` program should print an error message and stop.
- The script file provided could not be opened. In this case, your `naivedb` program should print an error message and stop.
- A pathname given to either of the `create`,`fileconverter`, `indexer` or `srchindx` programs cannot be opened. In this case, the corresponding program should print an error message including the reason for failure (e.g., doesn't exist) and exit. Note that your `naivedb` program should **not** exit in this case, but instead continue executing commands. The only thing your `naivedb` program should be aware of is whether the program it called (e.g., `srchindx`) encountered an error or not.

- Wrong number of arguments given to either one of `create`, `fileconverter`, `indexer` or `srchindx` programs. The `create` (similarly for `fileconverter`, `indexer`, and `srchindx`) program should print an error message and stop. However, your `naivedb` program should not exit, but instead continue executing commands.
- The specified input file provided to `fileconverter` cannot be opened. In this case, the `fileconverter` program should print an error message including the reason for failure and exit. Note that your `naivedb` program should **not** exit in this case but instead continue executing commands.
- An invalid command is given to your `naivedb`. In this case, `naivedb` should print an error message and then continue processing commands.

C. Remarks.

- You may not utilize the `system()` function to complete any of the tasks.
- You may not have your `naivedb` program execute any existing bash utility programs (e.g., ls, ln, mkdir). It must utilize system calls to execute other programs. In turn, these programs must issue system calls directly (i.e., they too cannot use the `system()` function or execute any of the existing bash programs).
- When your source code is compiled and linked, it should produce **five** (5) executable files, namely, `naivedb`, `create`, `fileconverter`, `indexer`, and `srchindx`.
- Your `naivedb` program should execute each of the `list`, `create`, and `fileconverter` programs by creating a child process and having that child process execute the corresponding executable for that command, while the parent process (i.e., the `naiveinterface` program) waits for the child process to finish.
- The `create fileconverter`, `indexer`, and `srchindx` programs should accept their arguments via the `argv` parameter.
- For your `fileconverter` program, you may assume that the given binary file will be non–empty, and that it will additionally contain the correct representation of each record of the corresponding text file.
- If any part of your programs fails to compile, you will receive a zero on the entire assignment.

D. Structural Requirements. Your submission must *at least* contain the following files:

1. A C source file named `naivedb.c` with just the `main` function for your shell program.
2. A C source file named `create.c` with just the `main` function for the list program.
3. A C source file named `fileconverter.c` with just the `main` function for the fileconverter program.
4. A C source file named `interfaceFunctions.c`, which contains functions for handling the commands (i.e., `quit` and output redirection) that the shell program must perform.
5. A C source file named `input.c` with the function `char* getLine(FILE* stream)`. This function reads one line from the specified file and returns the result as a null terminated string. Note that this can be used to read from stdin by calling it as getLine(stdin). This source file can additionally contain helper functions.

6. A header file containing the structure definition(s) for your BST and function prototypes for functions related to your BSt table, i.e., inserting a node into the table, retrieving the number of flights given an airline code, and printing the contents of the tree to stdout.
7. A header file containing the structure definition(s) of your hash table and the function prototypes for functions related to your hash table.
8. The C file `hashFunction.c` which will be provided to you on Blackboard, and contains a hash function that you **must** use in your program.
9. Your `makefile` should include a clean target.