Database of Records Project

Assignment:

The goal of this assignment is to develop a student employee record data retrieval system for Podunk University. To do so, you will bring together many of the concepts covered throughout the semester. You will implement a database system supporting access by several secondary keys. The data records will be stored in a disk file. Access to the records will be via primary key, with the primary key index being a hash table. The actual queries will be to secondary keys, indexed by Binary Search Trees (BSTs).

The database will store records for those students who are employees of the University. Database records will contain six fields. The first two fields will be strings of variable length. The first string field will be the name. The second string field will be the address. The third field is the ID string, and should be stored as an 8-byte character array. The fourth field is student's GPA, and should be stored as a double. Its value will be in the range 0.00 to 4.00. The fifth field is a code for the student's major. It is a four-character string. The sixth field is the salary (in cents), and should be stored as a 2-byte integer. When you create your data record object, you may reorder the fields within the object as you see fit. You may also reorder the fields as stored in the data file.

The input to the program will be a command file containing commands, read from standard input. Output for the program will be to standard output.

The only place where any given record will be stored in its complete form will be in the disk file. The disk file will be managed using the first-fit memory manager that you wrote for Project 3.

Any given search command will refer to one of the key fields in the data record, known as **secondary** keys. The secondary key fields that are supported by search are the Name field, the GPA field, the Major field, and the Salary field. For each secondary key field, there will be an associated BST, stored in memory. Thus, your program maintains four BSTs, each with its own key type and comparator function. BST nodes will **not** store complete records. Instead, a BST node will store the key value and the (8 byte) employee ID value for the associated record. The BSTs will be implemented using templates parameters for the data types and comparator functions.

The Employee ID field for the records serves as the **primary** key for the database. You will use a hash table to index the employee ID field. Recall that this is a 8-character ASCII quantity. The hash function you will use is as follows: Take the first four bytes interpreted as an integer value and add this to the second four bytes also interpreted as an integer value. Then, extract the middle two bytes of the 4-byte sum (you can do this by sifting the result down by 8 bits and then copying to an unsigned short integer). Finally, take the result and perform the modulus operation by the size of the hash table. The collision resolution method you will use will be quadratic probing. The hash table slots will store the primary key value (an 8-byte quantity), the byte offset in the file that where the associated record begins, and the total size of the record in bytes.

Input:

You program will be called **pud**, and it will have two command-line parameters. The first parameter is the name of the data record file. Note that this will be your binary file (mediated by the memory manager) and it will be initialized to be empty/zero length at the beginning of the program. The second parameter is the size (in slots) for the hash table.

The input file will consist of the commands **enter**, **delete**, **search**, and **makenull**. Each enter command will require two lines of input, with a format as in the following example:

enter John Doe: 1002 Anywhere Street
JOHNDOEX 3.62 CMSC 10.50

The first line of the **enter** command contains two character strings, separated by a colon (:). The first character string is the name key. The second string is the address field, but this field is simply data – it will never be used for searching. The name key will start with a letter and will not contain colons or line breaks. Before insertion into the database, excess spaces in both character strings should be removed. Specifically, any leading or trailing spaces are ignored, and multiple spaces are collapsed into a single space.

The second line of data contains the values for the remaining four fields. Field three is the employee identification number. It will always be exactly 8 ASCII bytes. If this value duplicates the ID of any existing record in the database, then this **enter** command is in error, and the record should not be inserted into the database. Field four is the GPA, and will be expressed in the form `x.xx`. Its value will be in the range 0.00 to 4.00. Values outside the range indicate an error – in which case the record should not be inserted. Duplicates for this field are to be expected. The fifth field is the student's major, and will be four character long. Duplications are allowed. The sixth field is the hourly salary, and will be expressed in the form `xx.xx`, where the number of digits preceding the decimal is not fixed. This value should be converted to a 2-byte integer expressing the salary in cents. Duplications are allowed for the salary field.

When performing an insert, you must make sure that the hash table is not full. If it is full, output a suitable message and do not insert the record into the database.

The **delete** command takes one of two forms:

delete John Doe
delete 1 2.34

The first form for the **delete** command contains a character string. This character string must match the string for some record's name key exactly, after blanks have been trimmed as described for insertion. After locating a corresponding record in the name index (if it exists), it should be deleted from the database. This involves removing the record from the hash table, from each of the four BSTs, and from the disk file. Use a tombstone to mark deletions from the hash table.

The second form of **delete** has two parameters. The first specifies a secondary key field. It must be either 1, 2, or 3, corresponding to the GPA, Major, or Salary fields, respectively. The second number is the key value to be deleted. Some record (if one exists) with that value will be deleted. Once again, the record must be deleted from the hash table and BSTs.

There are three forms for the **search** command, as shown by these examples:

search John Doe
search 2 CMSC
search 1 3.0 3.5

In the first form, the search command takes a name and prints all matching records. The second form takes two parameters. The first parameter should be in the range 1-3 (otherwise it is an error). This number tells which field is to be searched, corresponding to the GPA, Major, or Salary fields, respectively. The second parameter tells which value for the secondary key is to be found. You should print all matching records. In the third from, again the first parameter specifies which secondary is to be searched. The remaining parameters are the lower and upper bounds for a value range, respectively. These values are inclusive (that is, records that have exactly the upper or lower bound value are also matches). The lower bound must be less than the upper bound, or it is an error. Complete information for all records whose key values fall between the two bounds (inclusive) should be printed.

The **makenull** command will reinitialize the hash table, all BSTs, and the record file to be empty.

You can expect all input commands to be syntactically correct.

How Searches Work:

Given that you have have some records stored in your database, here is an example for how a search will operate. Consider the following search command:
search 3 10.00 20.00

This indicates a search in the salary field for all employees with a value between $10.00 and $20.00. Thus, you will begin by searching in the salary BST. The search command expresses the salary value in decimal, form, which should be converted into an integer (number of cents). Each entry in the BST with a matching key value (one in the range 1000-2000) will have stored with it the employee ID value for that record. You will search the hash table for each such ID value, to recover its position on the disk. For each such record, you will ask the memory manager to retrieve the record's bytes. All fields of this record will then be printed to standard output.

Since you are using quadratic probing as the collision resolution method, there is a possibility that not all slots in the tale will be probed for during a search. To avoid the possibility of an infinite loop, limit the number of probe steps to be the size of the hash table. Thus, if the hash table is size 10, you would not attempt more than 10 probes along the probe sequence for any search or insert operation. In the event where you do not find the record (or an empty slot) after this number of tries, consider the search or insert to have failed.