**CSCE 4523 Database Management Systems**

**Homework 1**

**Due: Monday, February 1 at 11:59pm**

**By: Andre Fuentes**

**Objectives**

The objective of this homework was to build a system that can create a file-based database from a csv. This system is a fixed length record system. The system is able to store, manipulate and display records stored inside of it.

**Approach**

I implemented the database using Python3. My first task was breaking up the problems in front of me into rational partitions. I decided to keep it simple and use two classes. One to handle the user interface and the other to handle the database functions. I began working on the database functions first to find a way to store the records. Python has a very helpful CSV reader that stores rows of a CSV into a python dictionary. From this dictionary, I was able to find the sizes of each field which came out to “{'ID': 8, 'Region': 3, 'State': 3, 'Code': 5, 'Name': 84, 'Type': 38, 'Visitors': 10}” This is a total record size of 151. With this information I left justified all of the records with trailing spaces (I apologize for how small the text is, but here is an example record):

1488805 NE VA SHEN Shenandoah\_National\_Park National\_Park 126112145

1488838 NE VA FRSP Fredericksburg\_and\_Spotsylvania\_County\_Battlefields\_Memorial\_National\_Military\_Park National\_Military\_Park 42957488

There are spaces inside of the fields, so when storing a field value, I replaced each space with an underscore. This made parsing the data file very convenient because I could simply read the line and split the data between the spaces then replace the underscores with spaces once I wanted to display the record to the user.

The config file was used to hold the field sizes, the total record size, and the total spaces in the database. I decided to hold each individual field size and the total record size in the config so I wouldn’t have to re-calculate the total field size every time I opened the database. I also stored the total spaces in the database rather than just the number of actual records, because the total spaces is more useful when trying to use binary search on the data. The config:

{'ID': 8, 'Region': 3, 'State': 3, 'Code': 5, 'Name': 84, 'Type': 38, 'Visitors': 10}

{'record\_size': 151}

{'num\_records': 382}

{'num\_record\_slots': 764}

For sample code I used the python binary search given by Dr. Gauch. Though I had to modify the get\_record function so that if it landed on a blank record it would check the next record below it and record how many times it shifted. Then in the binary search function instead of just returning the string of the record, I used my own function to return a dictionary of the record for easy use. Handling overflow was a bit of a challenge. The first problem was to detect when overflow occurs. Overflow occurs once we try to insert a record, but the position is filled, so evaluate whether the ID is greater or less than the ID we found through binary search. If the value is less than we need to check the item before it, and if the value is greater than we need to check the item below it. Once we have determined the item below (or above depending on ID) is also filled we then have to handle overflow. To handle the overflow, I called the order\_db method to rewrite the file. As I was calculating where the record should go, I stored the record prior to the one to be inserted. This allowed me to just write records one by one from the old file to the new, with spaces in-between and once I found the record previous to the record I am inserting, I print previous -> blank -> new record -> blank and finish writing the rest of the file. There are two edge cases though, one where there is no previous file (inserting at the head), this was an easy fix. I just passed a key word argument and notified order\_db that it was the first item to be written. The final edge case was the last argument. To fix this if the record was not inserted into the DB at any point, it inserts at the end.

**Results**

I did a lot of error checking in my code and I believe I made it unbreakable. For starters, in my UI I ensured all of the data being passed into my database manager was what the manager was expecting. For example, if the person is trying to update, display, etc. a record the code ensures that the data being passed to the manager is a positive integer. This does the bulk of the error checking and worked very well. The next step was to ensure that no data being inserted into records were too large for specific fields. I created a truncate\_data function that takes in the field to the record you’re trying to change and the value. If the value is larger than the expected size of that field, then the data is truncated so it doesn’t change the field or record size. This allows the user to pretty much type anything into the database and not break it. Though it will not save all of the data the user types if it is too long. From an efficiency standpoint, I believe my code is fairly good. I refrained from iterating over a lines and files more than once when I didn’t have to. Using a binary search tree rather than a linear search is definitely a more optimal solution. My code behaved exactly as expected

**Testing**

I began my testing with the sample provided by Dr. Gauch. All of the tests were successful, and I was able to verify the database was working as planned (typescript). This type script tested I also did much of my own testing. For example:

I tested my overflow with two records at the very top and the very bottom of the program to make sure my overflow was working on the edges of the database. I also did these two ways: Once inserting a record at the end/beginning of a file that was smaller than the second record. So, it should rewrite the database and have the first(smaller) item appear first and the second(larger) appear after. Then I reversed this by adding a larger record first then the smaller. So, after we insert the record the smaller should be inserted before the larger.

EX: showing database files (inserting larger then smaller at end of file):

2775875 NC DC BEPA Belmont-Paul\_Women's\_Equality\_National\_Monument ...

2775900 AR AR ZEMA Kids\_on\_the\_block ...

(Next insert)

2775875 NC DC BEPA Belmont-Paul\_Women's\_Equality\_National\_Monument ...

2775890 AZ AZ RAZZ Pineapple\_coconut ...

2775900 AR AR ZEMA Kids\_on\_the\_block ...

EX: showing database files (inserting larger then smaller at end of file):

2775875 NC DC BEPA Belmont-Paul\_Women's\_Equality\_National\_Monument ...

2775890 AZ AZ RAZZ PINEAPPLE\_COCONUT ...

(Next insert)

2775875 NC DC BEPA Belmont-Paul\_Women's\_Equality\_National\_Monument ...

2775890 AZ AZ RAZZ PINEAPPLE\_COCONUT ...

2775900 AR AR ZEMA Kids\_on\_the\_block ...

EX: showing database files (inserting larger then smaller at the beginning of file):

200 SM SM SMOL NOT\_THE\_SMALLEST ...

2877 IM AZ CHIR Chiricahua\_National\_Monument ...

(Next insert)

1 VS VS VSML ACUTALLY\_THE\_SMALLEST ...

200 SM SM SMOL NOT\_THE\_SMALLEST ...

2877 IM AZ CHIR Chiricahua\_National\_Monument ...

1 VS VS VSML ACTUALLY\_THE\_SMALLEST ...

2877 IM AZ CHIR Chiricahua\_National\_Monument ...

(next insert)

1 VS VS VSML ACTUALLY\_THE\_SMALLEST ...

200 SM SM SMOL NOT\_THE\_SMALLEST ...

2877 IM AZ CHIR Chiricahua\_National\_Monument ...

I also attempted to update fields of data that didn’t exist, and update fields beyond the field size. The UI either rejected the data given by the user and asked it again, or the UI accepted the data, and the database truncated the data as necessary.

(Check update\_test typescript)

I attempted to add duplicate sets of data (Check duplicate\_test), this was successful because if the add\_record function found the record we were trying to add it would not allow it.

I tested the create report method by deleting the first second and 10th record, and ensured it still printed out ten non-blank records. This test was successful because even after the 3 deletes from items in the first ten records, the creapte\_report method still printed out the top ten records(report\_test).

I tested to ensure no actions could be called on the database if one was not open. This was successful and a message would pop up to the user if they tried to access a method that needs a database to be open. (check non\_openDB\_test).

All of my tests were very successful.

**Typescript**

Check the file “typescript” for the main test. All of the other text files are testing from the results section.