Project 3 Report

Carlyn Thomsen, Kay Harris, Vi Nguyen, Michael Samatas

Project Description

In this project we had to take a file from the user and use the information it contains to load, print, manipulate, and save three tables. Specifically, the ability to insert, update, select, delete, display, and write was required. The user-inputted file contained two types of data: CSV file names and queries. The CSV files had to be split up and loaded into the tables while the queries needed to be broken down and acted upon. To manage the data and implement these functions, each table required a hash table and a linked list, as well as careful orchestration of hashing functions, collision avoidance, comparisons, and a bunch of pointers. Additionally, this project requires cooperation and communication from all group members to avoid overlap.

Major Functionality 1 (Kay Harris)

Before one.cpp can be explained, the abstract class **Table** needs to be brought up. **Table** is little more than a header with 8 virtual functions: the deconstructor, insert, update, select, del(ete), display, write, and bonus. The other parts needed to implement these functions anyways, and this abstract class allowed polymorphism to be used, greatly simplifying the part one code. It was decided that all of these functions would be void and print their own messages because any intermediate data for select, delete, display, and bonus would require access to private structs and vary from table to table. Returning strings to be printed seemed rather pointless. Each function gets the required tuple in the form of a vector<string> that is split in part one.

In one.cpp, the first definition is the **query** regex. Using conditionals, **query** was made in such a way that it could parse all 6 query types, plus the bonus, into their basic parts: instruction, tuple, table name, and bonus columns

query: (.+?)\\((?:\\(([^\\)]+)\\),([^\\)]+)(?:\\),(.+))?)?\\)
Tuple and table name use [^\\)]+ because they always end at a ‘)’ which is a very unambiguous stopping point. Since I wanted instruction to take the ‘(‘ in PROJECT(SELECT, it’s written to stop at a ‘((‘ or ‘()’ based on the conditional. I added another conditional for the bonus columns, which takes everything up to the final ‘)’

The next regex created is **splitter**, which is used in the **split** function. **Splitter** is designed to find and remove the first item in a tuple. By doing this repeatedly, **split**

can break down a tuple of any size into a vector, making dedicated regexes for each **Table** unnecessary

splitter: (\"[^\"]+\"|[^,]+),?(.*)
The first capture group has two cases: one that will find text between quotes, and one that will find all the text up until the next comma. This will therefore always get the first item in the tuple. Assuming this isn't the only item in the tuple, the comma is skipped and the second capture group will get the rest of the line. If that is the only item, the ',?' and '.*' happily take no input

The **read** function is where most of the processing occurs. First, an instance of each of the three **Tables** is created and a lambda is created to look them up by name. A lambda was chosen over a function because it could access the **Tables** without having them passed in every time. Next, the user is prompted for an input file. The three csv files are then read in, and, by using the lambda, polymorphism, and generic **split** function, the data in the csv file is inserted into each **Table** as simply as table(name)->insert(split(line)). Each remaining line in the input file is then broken down using the **query** regex into a smatch that contains all the parts. An if chain selects the right function to use, the right table(s) are found, and the corresponding data is passed in. After that, **read** and the rest of the program end.

Major Functionality 2 (Michael Samatas)

Part two is responsible making a table to store the geography data and making inserting, updataing, selecting, deleting, and outputting functions for the table. For the **hash()** function I used Mid-Square Hashing which takes the key and squares it then takes the middle numbers and then % by the size of the table. The **collision strategy** i use quadratic probing which when there is a collision the probe increases by one, and the hashed key + probe * C1 + probe^2 * C2 % the table size.

The **insert()** function uses a while loop to loop while there is room in the table. While there is room in the table we check the first spot that we hash in the table is empty. If the table is empty then we insert the given data into the table and add the position to the linked list. If the spot in the table is not empty we check if the data in the spot is equal to the given data. If they are the same then we print out a message that the data is already in the table. If neither of the conditions pass then we use our collision strategy to find the next spot to put the data and check the conditions again until we find an open spot. When the table runs out of space we will pop out of the while loop and make more room in the table by increasing the size and re hashing the data back in to the table. The **update()** function loop checks if there is room in table ,then checks if the key in the given table matches the key in the data table. If they

match then the update function will replace the data in the table with the given data. If the keys don't match then we use the collision strategy to check the next position in the table. If there are no matched in the table the we print an error message. The **select()**, and **delete()** functions are very similar first they check if there is a key value or a "*" if there is no key value then we use the secondary index, our link list to look at each of the eight parts of data and check for a value or a "*" for all parts of the link list we then take all the data in the link list that matched the given data. If they match then the select function will print the selected data, and the delete function will delete the data are replaced it with a tombstone. if there is a key value the functions loop while there is room in the table. Then they check if the key that is given matched the key in the data table. If they match then the select function will print the selected data, and the delete function will delete the data are replaced it with a tombstone. If it doesn't match then it uses the collision strategy to look in the next position until we run out of size in the table. When we pop out of the loop then we print that there is no matching data. The **display()** and **write()** functions are similar, we loop for the size of the table and print out all the spaces in the table that are not blank the only difference is that the display table uses cout and the write function uses output to write into a file.

Major Functionality 3 (Carlyn Thomsen)

This portion of the project dealt with the part3 Class or the Age Table. The part3 class is a child class of the abstract Table class. We thought it would be very efficient to use abstract classes and polymorphism for this project, because parts 2-4 all need to implement the same functions. As a group, we decided that it would be a smart idea for the Table class to implement the functionality of passing a vector of strings that are already split in part one to each table, which will allow the easiest access to specific information. The **Age Table** was created using a vector of Entry structs. This allowed easy hashing and the ability to access certain elements in each spot of the struct inside of the vector. In part3.cpp, the hashing function was **Modulo Hashing** and the collision strategy was **Linear Probing**.

For the **hashf()** function, it takes in a key string, which is the geoid, and makes it into a substring without the first 7 characters of the string. This makes the key into a more simple and unique ID, as the first 7 characters are the same for each ID. The function then makes the key into an integer in order to perform the modulus operation and return the correct value. For the **insert()** function, it takes in a *vector<string> args* as stated above, and sets *int pos* to the value that is returned when the hashf() function is passed with the first value of the string of args, which is the geoid of a specific Entry. The function then begins going through the hash table

and checks if the hash table at that position is empty using the bool *isEmpty*. The function only inserts the Entry using linear probing if the position is empty; otherwise, it displays a message that there is already an entry with that key value. The **update()** function goes through the hash table, and if args[0] (the geoid) matches the geoid at that position of the hashMap, it sets the contents of args equal to that position of the hash table. If a position with that geoid doesn't exist, the user is notified and no other action is taken. For the **select()** function, it goes through the hash table and checks if args[0] is " * ", and keeps track of the specific rows that match. If it isn't, then it searches the hash table for a geoid that matches args[0]. If args[0] is " * ", then it searches the linked-list for a pointer pointing to a position that holds that geoid. The function then displays to the user how many, if any, entries matched the query and the specific rows that matched. The **delete()** function, goes through the hash table and checks if args[0] is " * ". If it isn't, then it searches the hash table for a geoid that matches args[0], and deletes that entry and sets the boolean value *isEmpty* for that position to true. If args[0] is " * ", then it searches the linked-list for a pointer pointing to a position that holds that geoid and deletes that entry. The **display()** function displays the content of the hash table if the space at that position is not empty. It displays the content in tabular format with the table scheme displayed as column titles. The **write()** function writes the content of the hash table to a separate output file, as long as the file opens. If the file doesn't open, it notifies the user that the file was unable to open. It displays the content in tabular format with the table scheme displayed as column titles.

Major Functionality 4 (Vi Nguyen)

Part 4 handles disability table and other functions including insert, update, select, delete, display, write, and bonus that are related to disability table. We created class **disability** derived from Kay's base class **Table**, which implements a hash table mapping keys **geo_name** to values. This class has a struct **Entry** as an entry of this table, and this struct has a vector<string> **data** to store all its data (geo_name, hearing disability, etc,…) for easy comparing and modifying. Struct Entry also has overloading operator == with a vector<string> ; this would compare this vector<string> and data by looping through an int i (which works because they both have the same size), and if any elements in data did not match with its corresponding element in the vector<string> and both of them are not a "*" then it would return false, or else it returned true. This allowed us to find entries that will match with any vector<String> that Kay would give, even with "*".

It doesn't matter what collision strategy we have for this table, so we randomly picked chaining. Thus, we decided that a vector of lists **hashTable** with size 10 will be

suitable to hold all the data for the table, as we just needed to insert entries at the end of lists and using lists will make insert and delete entries easy since the size of our vector will never change. Additionally, because the keys for this table are strings, we thought multiplicative string hashing is a suitable hashing function, so we implemented it in a private function **hash**. This function takes in string key and int size (which will be the size of **hashTable**, 10) and return an int position. This position is not the position of the entry with matching key but the position of the list that holds that entry. **Hash** is built based on the code that Dr. Helsing provided. We also have a function **toString**, which will take in a vector of string and add each element together into one big string for output messages.

We also have a struct **Node** in disability to set up our secondary index which is a linked list as there was no need for us to put it in a separate file. **Node** has a pointer to an entry in the table and a pointer to the next node. Additionally, we added a Node pointer **head** in our class to hold the first node of the linked list. We set **head** as a null pointer in our class constructor and we built a destructor for the linked list as later on, we would use **new**. After a couple test, we found out there was no need to create a separate function to insert node or delete node, so we would just implement it when we implemented insert and deleting entries. This would be explained in more detailed when we explained our insert and delete function.

All major function takes in a vector<string> **args**. Insert() first check if there is already an entry that stores **args**. This was done by calling the hashing function to get a position **pos**, and loop through (**pos**+1)th list to compare each entry with **args**. If there was an entry that hold **args**, we would print out an appropriate message and exit Insert() function. If there wasn't, then we would create an entry with **args** and put it at the end of the (**pos**+1)th list. A temporary node that points to this entry was also created and added to the front of the linked list, and **head** would be set to this temporary node. Additionally, Update() also checked if there was already an entry that stores **args** like insert, but instead of just printing out an message, it also updates the data inside entry to be the same as **args**. Linked list was not used in this function.

Kay implemented her Select() function to not return a value, but since a returned value is needed for bonus(), we created a private rSelect() function that return a vector of entry result and Select() would simply call this function. If the key was not a "*", then rSelect would search for existed entries just like insert(), but instead of exit the function it would add this entry to result. If the key was "*", then we looked for the entry through our linked list. While looping through our linked list, we compared the entries that each node pointed to with **args** then proceeded just like without "*" as key. Because of the way we did our overloading operator ==, we would be able to compare the entry and **args** even if there are "*" in args However. we still split the

functions into two different cases because knowing what list to look for entries (in the "regular" key situation) would speed up the process as linked list would look through the entire table. Then based on the size of result, we would print out the appropriate message, which was no entry for 0, one entry for 1 and multiple entries for the rest. Delete() was very similar to rSelect(), but we decided to loop through linked list straight away instead of creating two different cases because no matter what the case was, we would still need to delete the entry from both the hash table and the linked list. Another difference between Delete() and rSelect() is that we modified the hash table and linked list instead of just adding entry to result. This was done by using three node pointer **temp, pre** and **current** which would hold the node we wanted to delete, the previous node and the current node. So as we looping through the linked list, if the current node matches with args, we set the current node as **temp** and move **current** to the next node. Then we check if the matching node was **head,** and if true, then we set **head** to the next node; if not, we connected **pre** with the next node. We then got the (**pos**+1)th list that would store the matching entry using **hash**, but instead of using **args**[0] as key, which could cause a problem if it was a "*", we use **data**[0] of the entry temp pointing to. Next, as we using chaining as our collision strategy and lists to store our entries, we could remove the matching entry from the list by using list.remove(entry). Finally we deleted **temp** and printed out the appropriate output. However, if the current node does not match with **args**, then we simply set the current node as **temp** and move **current** to the next node.
For the purpose of Display(),Write() and Bonus(), we created a vector<string> **scheme** in disability that holds the scheme of the hash table as we would needed it when printing out the output. For Display(), we added a vector<int> locally to hold the sizes that would be use with setw for formatting our output. Then we printed/wrote the key, which was **scheme**[0], and the scheme and the entries inside table by looping through scheme and linked list in Display()/Write(). We could loop through **hashTable** to print/write our table but we chose to use linked list so that the entry would be in chronological order. In addition, besides **args**, Bonus() takes in a vector<string> **cols** that's supposed to tell us which column(s) the user wanted to print out. First, we called rSelect and stored the return values in a vector of entry entries. As we looped through each elements of entries, we checked which data of entry the user wanted to print out by looping through **cols** and checking if each element of **cols** match with any of scheme and printed out the associated scheme and data.