Jay Offerdahl
EECS 560
Project 2

Project 2 Report: Hash Tables

Introduction

Overall, this program was much simpler to implement than the first project, as I have had recent experience with hash tables (other classes). I was able to get my customer/hash classes essentially finalized within a few hours, and spent most of the development time on the eleven commands we were tasked with implementing, as well as testing.

Program Structure

I use four classes in this program:

- main – handles interaction between the input (file) and the hash tables/database.
    - Reads “phonebook.txt” into the database to have a beginning set of customers.
    - Opens the input text file to use as testing (proj2data.txt is the file I used to test)
        - Reads line by line to determine what action to perform, and then what data to use with that action.
    - Handles most of the errors, such as “customer already exists/doesn’t exist”, or invalid input.
- Customer – the actual data objects that were being stored.
    - Stores name, address, number, and charges as member variables.
    - Built in methods for “makePayment” and “addCharge”, which determine correct input and throw exceptions accordingly.
    - Built in method to print customer data (printing the record).
    - Pointers to nextName, nextArea, nextNum to be used in the hash structures.
- HashTable – An all-in-one hash table for the three different types of tables implemented here.
    - Each table has a hashType, which is an enum which declares what kind of table it is. (Three options: CUST_NAME, CUST_AREA, CUST_NUM, which correspond to a name table, area table, and number table.)
    - Differentiated functions based on table type.
        - hash() takes into account the table type so it returns the correct hash value.
        - Insert() does the same, loops through different pointers depending on table type, so if it’s a name table, loops through Customer’s nextName pointers.
    - print() function simply prints out the hash table (in the array order), which was mainly used for testing.
    - find() returns a Customer object, which is especially useful when handling operations with customers. Find can be used on names or numbers, as specified.

In a general discussion, I again took memory management serious in this program. Since the Customer objects had to span three tables, I could only create them in main, and with this in mind, I had to clean them up in main as well. So, upon program exit (option 12), I go through literally every record and first remove it from the hash tables, and then delete the object in order to free up the memory.

Jay Offerdahl
EECS 560
Project 2

Testing

To begin, I setup a loop to pull all the data in from proj2data.txt in order to test my methods. I modified the given data file to include several of my own tests, which ensured proper functionality in all cases, including edge cases. For example, I had to make sure correct valid input was always given, which if not, threw an exception. In order to view the output, you can run the following commands:

```
make

./proj2 proj2data.txt          // or whatever data file you want to use
```

Looking at the Output

If we look at this output, we can see that valid input is checked rigorously, as any customer who is already in the database is not re-inserted, and if a customer doesn't exist in the database and an operation is attempted on their account (excluding insert), an error will be displayed. See below for a more in-depth discussion of testing each operation.

1. Inserting a customer by name is shown to work by inserting the record, verifying placement using the "find()" command, and then printing out the found record. I've shown this using Frank Brown as an example.
2. I also used Frank Brown as an example here, and it can be seen that the same information was input into the database as an insert by name, because the input was parsed accordingly for the number.
3. After inserting Frank Brown once, I removed this record by name. The name was then searched for in the database, and when it wasn't found, the message that it was successfully removed was printed.
4. Again, after inserting Frank Brown once more, I removed his record by number. The program clearly found the record, printed out the final bill, deleted the record from the database, and checked the database again in order to confirm that Frank Brown was no longer in the system.
5. Next, I made a payment on Frank Brown's account. The original balance was $1337.00, which I printed out using option 7 or 8. I made a payment of $337.00, which then brought the balance down to $1000.00. Next, I made another payment using Frank Brown's phone number for $500.00, bringing the balance down to $500.00. Lastly, I tested a $5000.00 payment when his balance was only $500.00, which printed out an error because that was larger than his current charges.
6. To test charging customers, I used good old Frank Brown again. I charged his account by name for $1000.00, bringing the balance from $500.00 to $1500.00. I then charged his account by number for an additional $500.00, bringing the balance up to $2000.00.
7. This was simple to test as I entered a name "Brown Frank F", and it told me the current charges on the account.
8. I did the same as option 7 except I used Frank Brown's phone number, which printed out the current amount.

9. Since name hashes were calculated using a MOD 47, values from 0 to 46 were allowed. Upon entering a value, the names at this hash were printed, not much else to test.
10. This was pretty easy to test as I would enter a valid input (1 – 31) for days of the month, and the customers who were to be billed that day were printed out. Due to the nature of the hash function for numbers, there was a cutoff at 25 or 26 days (around there) where no more names were to be printed out, so it makes sense that printing out bills for the 28th returned nothing.
11. I used this function extensively to test insertion/deletion. Before inserting into the database, I would print out all of the records in area code 785. Next, I would insert Frank Brown with a 785 area code, and print out the records in this area code again. I knew it was working because the record was not there originally, but it was there after insertion. This also worked the same way with deletion.

Problems Encountered

I actually had relatively few major problems this time around. Like I previously said, I have had recent experience implementing hash tables (for another class), so it took little time to build the hash tables. I did have to go back several times to update the functions because I needed them to perform in different ways for the three types of tables, however. Also, I had not written delete functions in hash tables before, so that took a bit of time.

Honestly, testing this was also very pain-free. With the large input we were given, it was easy to test each function, and I was able to target each function because before implementing a file read in mechanism with a while look, I specifically took user input from the same set of options and user input for those specific options. This allowed me to target each individual function so while writing them I could easily test with my own input.

It's worth mentioning that the provided test file had some spacing issues, so I fixed those and added several more lines of my own. I also developed my program in a Windows environment, so when I tested on a unix machine, I had trouble reading in the text file because Windows uses the \r escape character at the end of lines in text files, so I was getting bad output with my first tests on the cycle servers. After making the text file in a unix environment, everything worked smoothly again.

Lastly, like I said earlier, I took memory management seriously, again, so since Customer's are passed around three different hash tables, I had to clean up memory in main. This is located in the code for option 12 (exit), and you can see that I go through the name table, find every record, and remove it from each table, only then could I delete the record properly. After this was done, I deleted the rest of the objects I had dynamically allocated, and exited the program.

Conclusion

I believe that the testing I have performed here demonstrates full functionality of the program. I have included a README text file with instructions on how to perform you own tests on the outside functions, and if you have any problems, please feel free to contact me by email: jayofferdahl@ku.edu, or by cell phone: (573) – 673 – 5212.