Michael Merola
Professor Rhonda Hoenigman
CSCI 2270 Data Structures
5 May 2017

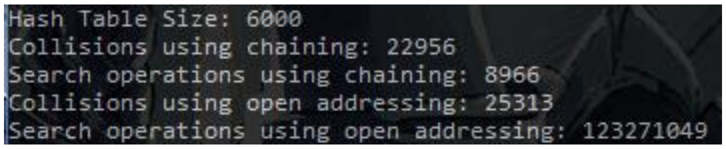
Hash Tables Final Project

For this project, I worked with Krish Dholakiya to program a hash table data structure from scratch. The purpose of this hash table is to store the data pertaining to every baseball player in America since 1985. Every baseball player has personal data including their birth year, weight, height, and even which hands they throw and bat with. Additionally, the dataset includes information about the player's career every year by describing the player's team, league, and salary for that year. The program should allow a user to traverse and search a fully-functional hash table of the table provided. Although the purpose of the hash table is to store data, this project's purpose is to evaluate the efficiency of two different collision resolution methods: chaining and open addressing. Chaining resolves collisions within hash tables by creating linked lists at the hash key with the collision. Open Addressing is different because it entails traversing the linked list from the collided hash key until an open spot is available to place the new object. We want to determine which resolution method is more effective by building a working hash table and evaluating the data to reach a conclusion.

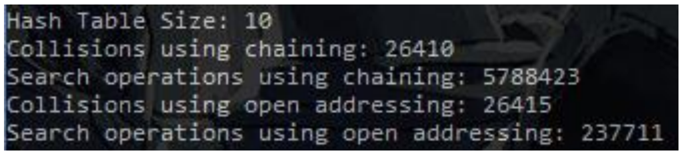
I started building the hash table by implementing a class object. I built the class to include a struct for the player and careerData objects so that I could use a vector of player objects to hold all of the data from the player datasheet. The class includes a function to read data in from a file named in a command-line argument. First the data is traversed and stored in a vector using the chaining method to resolve collisions. The program then reads in the same data to another vector, but resolves collisions using the open addressing solution. Before I moved on to build the functionality to read the entire data file, I implemented a main function that includes a simple menu and can take arguments from the command line. The main function serves as the starting and ending point of the program, and it directs the entire program. I read in the file by using filestream and stringstream objects and traversed the file line by line and added each element of a line to a new player object. At the end of each line, I would attempt to add the new player object to the hash table. I generate a hash key for the specific player and I check if there is any other player at the specific index. If the index is empty, then the player object is simply added to that spot in the array. The functionality of building the hash table up until this point is exactly the same for both methods of collision resolution. For chaining, I implemented the vector array to be a pointer to player objects for the purpose of creating linked lists at indexes who have multiple player hashes. Each player object has a pointer that points to the next player object. This implementation of a linked list is the centerpiece of the chaining method. If there is a collision in the hash table, the method will traverse the linked list to either find a match or place the player object at the end of the list. For open addressing, I did not use a secondary data structure to assist the process. This method will traverse the entire linked list after the hash key until it finds a key that has an open position. After I completed the implementation of each collision resolution method, I finished by creating the search function. My search function will traverse both the chaining and open addressing vectors to produce an evaluation of search operations, but it only prints out the data from the chaining vector. It searches by first generating the hash key for the player when the user provides the player's first/last name and birth year. It then goes to that hash and then either traverses the linked list at that hash or traverses the hash table depending on which vector of data it is using. The whole project required skill and knowledge of the data structures we have used and learned in class and on the assignments, as well as the ability to use pointers and efficiently debug segmentation fault errors. From working on this project, I have noticed a significant improvement to my coding proficiency as compared to the beginning of the year.

The data set is comprised of statistics for each baseball player and details about their career. The file contains multiple instances of a player and different years and teams that he played for. After building the hash table for the dataset, each player has his own unique position in the data structure, and his career statistics are recorded in their own personal array. In this format, the data is much more accessible and easily read and navigated. For this project, I made my own hash function to create keys for each player in the table. I did not use the given HashSum function because its max range of index values was too small. I created my hash by multiplying each ascii value of the player's first name together and adding it to the same quantity of the player's last name. I then added the player's birth year and modulated by the table size. The resulting quantity is a hash key with a range that I have observed to typically go above 6000. This extended hash key range allows for the hash table to make more efficient use of the space that is allocated to it. There are fewer collisions when most hash keys generated are unique from each other, which improves the performance of the data structure as well as enhances usability and efficiency. The data can be used to analyze performance of each collision method, as well as to analyze baseball player stats throughout the years that they played. This usage of a hash table has several applications to actual use of hash tables. Data analysts can easily traverse hash tables to find, compare, and store data.

From starting my project, I made predictions about the performance of each collision resolution method. I predicted that the chaining method would be more effective because it combines data structures to cleanly store data even if it collides at certain keys. I was skeptical of the open addressing solution because it seems inefficient to potentially traverse the entire hash table looking for an open position or for a match to a search. It seems like a messy and inefficient solution for collisions. With a hash table size of 6000, I found that open addressing had both more collisions and search operations than the chaining method did. Surprisingly, when I used a hash table size of 10, the collisions for both methods became very close in size. However,

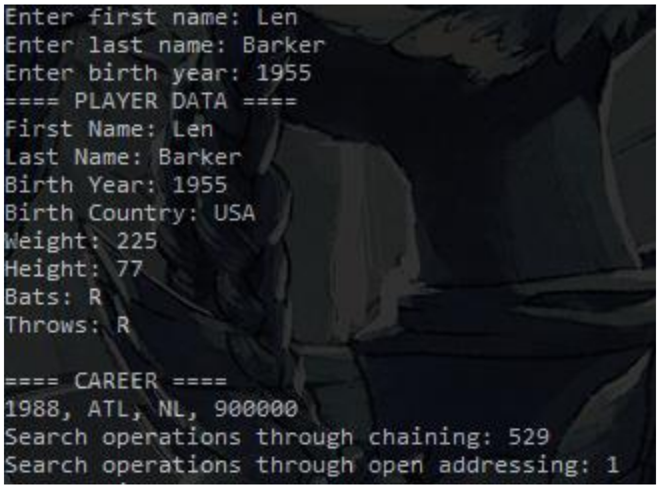


```
Hash Table Size: 6000
Collisions using chaining: 22956
Search operations using chaining: 8966
Collisions using open addressing: 25313
Search operations using open addressing: 123271049
```



```
Hash Table Size: 10
Collisions using chaining: 26410
Search operations using chaining: 5788423
Collisions using open addressing: 26415
Search operations using open addressing: 237711
```

the search operations for chaining became much larger than the open addressing method. Because the open addressing method finds empty spots in the hash table, it actually more efficiently utilizes the space and size of a hash table when the size of the table is large. Although the chaining method might be more useful for smaller table sizes, the fact that it incorporates two different data structures makes it much slower and inefficient. As shown from my screenshot of the output when I search for a player, the search operations needed for the chaining method greatly exceed those needed for the open addressing method (529 v 1). In this perspective, it is easy to see how much more efficient the open addressing method works, although it may seem messy. Open addressing more properly utilizes the hash table's potential. From the start to finish of this project, I have come to the conclusion that the open addressing method of collision resolution is the most efficient method for utilizing hash tables and datasets.



```
Enter first name: Len
Enter last name: Barker
Enter birth year: 1955
==== PLAYER DATA ====
First Name: Len
Last Name: Barker
Birth Year: 1955
Birth Country: USA
Weight: 225
Height: 77
Bats: R
Throws: R

==== CAREER ====
1988, ATL, NL, 900000
Search operations through chaining: 529
Search operations through open addressing: 1
```