Heinz Ulrich Boehmer Fiehn
CSCI 2270

Chaining vs Open Addressing in Hash Tables

The goal of this project was to evaluate the performance of two different ways of dealing with collisions in hash tables. Specifically, we tested chaining and open addressing. Both of these methods solve the problem that arises whenever a has function returns the same index for two different keys. However, because of the way they deal with this, they both perform differently in different environments. To test the hash table we built, we were given a file that contains every MLB player on every team since 1985.

Chaining works by creating some sort of data structure whose head is accessed through the has table. Whenever a collision occurs, the data that is causing this collision is added to this data structure. For this project, I chose to use a linked list because the data set was small enough to guarantee that the resulting linked lists would be relatively small and thus using something with better search times –like a balanced tree– would be unnecessary. Also, the point of this investigation was to evaluate the performance of collision handling algorithms, not of the data structures used.

Open addressing works by looking for a free spot in the table once a collision occurs. There are several ways to do this, but I chose to use linear probing simply because I thought the data would be easier to interpret. Yes, quadratic probing would lead to fewer clumps of data and thus a more efficient hash table, but because the idea was to compare both collision resolution methods, I thought it made more sense to use the simpler one.

When I first started to create this, I decided to use a simple hashSum function as my hash sum function and while everything worked as expected, I soon realized that it was extremely inefficient and lead to many collisions. Thanks to Sunil's test cases on Piazza, I ended up going with the hash function he posted, as it proved to be much faster and consistent. This hash function works by starting out with a constant number and multiplying it by 101. The result of this is then added to the ASCII value of the char being evaluated. This process is repeated for all the chars in a string. Since the strings being used were a concatenation of a player's first and last names, the input strings ended up being pretty long, which meant that the number being generated by the function overflowed easily. Due to this, it is very important to specify a specific size for the variable, otherwise the function resulted in a segmentation fault. After the string has been analyzed, the modulo of the resulting number and the table size is performed in order to yield an index that is within bounds.

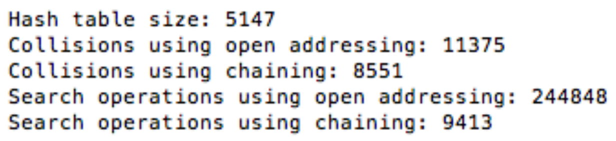
I decided to create a node struct for each player that contained all of the player's information, as well as next and previous pointers for the chaining hash table. I did this because I thought it was the easiest way of organizing the great amount of data we were given. Additionally, this allowed for me to store everything on the heap and not have to worry about running out of memory on the stack.

Heinz Ulrich Boehmer Fiehn
CSCI 2270

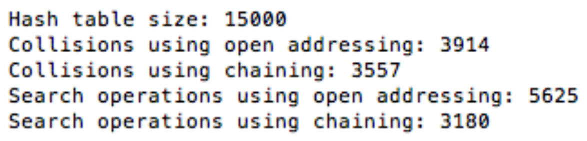
The data set used to test this was a pretty big file (around 26,000 lines long) that included not only the player's names, but also a number of stats associated with them. These stats included the year they were playing, their team, league, salary, birth year, birth country, weight, height, and the way they bat and throw. Because this file followed the chronology of the MLB, many players were listed more than once and were even part of different teams at different points. To deal with this, each time the program tried to add a player that already existed in the table, it added only the stats that changed to the existing player node. I decided to use a concatenation of the player's first and last names to create the hash table key because of the way the search function works. I briefly considered using the given playerID, in order to avoid having to deal with players that had the same name, but quickly realized that this would require having the user input the playerID whenever they wanted to search for a player, and thus immediately ruled this out.

We were also required to include a search function that took a name inputted by the user and displayed all the player's stats from both the chaining and open addressing hash tables. This algorithm also had to show how many search operations were needed to find the player using each of the collision resolution techniques.

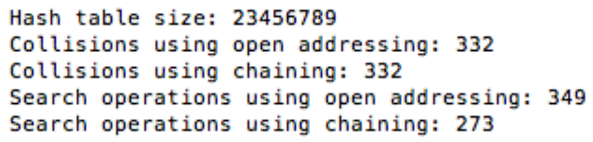
The actual results were pretty predictable. It makes sense that there will be more collisions for the open addressing algorithm, as the algorithm finds an empty space in the hash table to insert the new data. This means that a collision can occur even when the hash function gives two different hash table keys for two different strings, while the chaining hash table will only generate a collision if the hash function maps two different strings to the same hash table key. This, of course, also means that more search operations will be needed to both initialize the table and to access the data. In my tests, I used three different hash table sizes, 5147, 15000, and 23456789. I chose 5147 because it was the minimum size that the table could be to fit all the players, 15,000 because it seemed like a big enough jump to actually notice a difference in performance, and 23456789 because I thought it would be interesting to see how this behaved with an absurdly large, prime number. The results of the initialization of the three hash tables are shown below.



```
Hash table size: 5147
Collisions using open addressing: 11375
Collisions using chaining: 8551
Search operations using open addressing: 244848
Search operations using chaining: 9413
```



```
Hash table size: 15000
Collisions using open addressing: 3914
Collisions using chaining: 3557
Search operations using open addressing: 5625
Search operations using chaining: 3180
```



```
Hash table size: 23456789
Collisions using open addressing: 332
Collisions using chaining: 332
Search operations using open addressing: 349
Search operations using chaining: 273
```

It is apparent that during the initialization the chaining algorithm performed better in all cases, which matches up with my predictions. However, the margin by which it outperformed the open addressing algorithm decreases as the hash table size increases. Actually, when the size is 23456789, the number of collisions is exactly the same for both algorithms.

Heinz Ulrich Boehmer Fiehn
CSCI 2270

The search algorithm also offered valuable insight as to the performance of each algorithm. When searching for Ryan Zimmerman with a table size of 5147, chaining took 2 search operations and open addressing took 4. When a size of 15000 or 23456789 was used, both algorithms found the player without a single search operation, which goes to show just how much the table size affects the performance of a hash table. Nonetheless, the little valuable data that did come out of this test once again showed that chaining was more efficient than open addressing.

This project gave me some valuable insight into how the efficiency of a hash table changes depending on the collision resolution algorithm used. Although I did see some pretty great differences between using chaining and open addressing when initializing the hash table, the differences weren't all that great when searching. I suspect that with a much greater data set these differences would be highlighted a lot more than they were with the data we were provided with. Also, even though this difference is small for one search operation, it definitely adds up when running lots of them, which would heavily impact performance in the long run. So, from a performance point of view, it seems that chaining is superior to open addressing.