CSCI 2270 Project: Hash Table Performance Evaluation

Aparajithan Venkateswaran
Instructor: Rhonda Hoenigman
TA: Yang Li

Purpose

The purpose of this assignment is to build a hash table using two different collision resolution algorithms – Open Addressing and Chaining – and compare their performance.

A hash table is a mapping data structure built upon an array whose elements' location (in the array) is determined by a property of the element. A hash function is used to convert this property into a valid index location. On large datasets, accessing an element takes nearly constant time (depending upon the hash function and other elements). A perfect hash function, in theory, will ensure that no two elements are assigned the same index. However, it is very difficult to design one. This means that it's possible for two elements to be assigned the same index. This problem is called *collision* and is the greatest drawback in hash tables. There are multiple algorithms to handle collisions. We will evaluate the performance of two such algorithms – *Open Addressing* and *Chaining* - against the same database and hash function.

Procedure

The hash function we will use is a seed based hash function. The function is a loop of multiplying by 101 and adding to the ASCII value of each string character. Finally, take the remainder when divided by the table size as the hash value.

```
int hash = 0, seed = 101;
for (int i = 0; i < key.length(); i++) {
    hash = (hash * seed) + key[i];
}
return hash % tableSize;
```

The two collision resolution algorithms we are comparing are:

1. Open Addressing: Upon encountering a collision, this algorithm finds the next empty spot in the hash table and stores the element there, wrapping around the array if necessary. This is called *Linear Probing*. This means that we could potentially run out of space in the hash table which will cause us to overwrite existing data. This also means that not all elements are stored at the location determined by the hash function. Thus, while searching for an element, we may potentially need to look at the entire table before concluding anything.
   In this algorithm, we have choice between using an array of pointers to store in the hash table and using an array of the actual value itself. It will not affect the operations we perform on the data structure.

2. Chaining: Upon encountering a collision, this algorithm creates a linked list at that position in the hash table and "chains" all elements that have the same index given by the hash

function into a linked list (or any other dynamic array). This guarantees that all elements are stored at the location determined by the hash function. So, we only need to search the linked list at that location instead of going through the entire hash table.
In this algorithm, we are forced to store pointers to the value in the hash table. This is because we are essentially creating a linked list at every index in the table with each location (the pointer) serving as the head for the linked list.

There are 5147 unique players (considering their name, birth country and year). So, the default hash table size used was 5147. Then, to see how the number of collisions and search operations depend on the hash table size, we will increase the table size up to 25000 and see its effects.

While searching for a player in the hash table, we will look at the player name and any one team they played for to ensure we eliminate other players who share the same name.

Data

The raw data of baseball players provided consisted of player name, their birth year and country, weight, height, the teams they played for and the year they played for that team, the league that team belonged to, their salary while playing for that team, and their handedness for throwing and batting. The dataset consists of 26420 entries and 5147 unique players.

For computing purposes, I built two separate structs. One struct was for the team and consisted of salary, year, and team and league IDs. The other struct was the player itself and consisted of first and last name, birth year and country, height and weight, handedness and a vector of `Teams` (the struct) the player played for. The `Player` struct also consisted of a pointer to another `Player`. This was used for the Chaining algorithm (to build linked lists). This pointer was not used when implementing Open Addressing algorithm.

To generate the hash table key, the first and last names were concatenated and passed into the hash function described earlier in this report.

The hash table itself consisted of a dynamically allocated array of pointers to `Player`. These were initialized to `NULL` in the constructor. When using the Chaining algorithm, the `Players` with same indices were chained at that location by adding a new player to the end of the linked list. With the Open Addressing algorithm, each index either contained exactly one player or a `NULL` pointer.

Results

My intuition, based on how both these algorithms work, suggests that Chaining is better than Open Addressing. This is because Chaining guarantees that an element, if present in the database, is always present at the index given by the hash value. Open Addressing, on the other hand, does a poor job of using the hash value effectively and placing many elements at positions that is different from their hash value. Let us now analyze both algorithms and compare how well they perform.

Table 1 - Collision Analysis

Table Size	Collisions		Search Operations	
	Open Addressing	Chaining	Open Addressing	Chaining
5147	2594	1951	277156	9790
10000	1387	1191	9657	5192
15000	924	842	5072	3418

20000	706	648	3597	2638
25000	625	580	3162	2453

As we can see from Table 1, Chaining always encounters fewer collisions than Open Addressing. Also, the number of search operations done before inserting an element is smaller by a marginal amount for Chaining than Open Addressing. This trend is consistent for different sizes of the hash table. This indicates that creating a hash table by Chaining is computationally less expensive than Open Addressing.

As an aside, note that as we increase the size of the hash table, both the number of collisions and search operations decrease for both algorithms. This decrease (gradient) also slowly decreases as the table size increases. Also, note that the numbers for both algorithms appear to converge to the same value as we increase the table size. This is a very interesting trend and suggests that for tables that have a small occupied space to total space ratio, the collision resolution algorithm we use doesn't matter. But, we will see below that Chaining algorithm is more suitable in practice.

Table 2 - Search Analysis with table size 5147

Search Query			**Open Addressing**	**Chaining**
First Name	**Last Name**	**Team**		
Gene	Garber	ATL	0	0
Logan	Forsythe	TBR	4	0
Jason	Motte	CHN	20	0
Desmond	Jennings	TBA	31	3
*Cory	Synder	CLE	5146	0

* - Player not present in the database

After making a few search queries, from Table 2, we can see that Open Addressing takes, on average, more number of operations to find the requested player in the database than Chaining. This follows from our intuition that Chaining is better than Open Addressing. In Chaining, all elements with the same index determined by the hash function is present in the same location. This means that it is easier to locate the element (or find that the element doesn't exist) among a few elements having the same hash value. Open Addressing, in contrast, does not always place elements at the location described by its hash value. This means, we need to potentially do a linear search on the entire hash table to find the element. And for elements that don't exist in the database, we may need to search the entire table to ensure that it's not present.

Conclusion

From our observations, we can conclude that Chaining is a better algorithm than Open Addressing (with Linear Probing) in practice. Even though, we could achieve similar number of collisions with both algorithms at a large table size, a lot of memory is wasted in doing so. Besides, it is much faster to retrieve data from a Chained hash table than an Open Addressed table. And since, all practical uses of hash tables involve retrieving data from the table, it is only sensible that this operation should be fast. Thus, Chaining is better suited for resolving collisions. But, these data and conclusions only hold true for a static dataset. It would be interesting to study the effects of Chaining and Open Addressing on a dynamic dataset (such as a live Twitter feed) where it would be necessary to dynamically increase the hash table size as we read in more data.