Anthony Stem
CS 124 | Fall 2021
Project 5 Report

Dataset

This dataset is a subset of the dataset maintained by the Extrasolar Planets Encyclopaedia as of February 28, 2018. It contains data for 3732 confirmed exoplanets from various sources and missions such as Kepler, CoRoT, K2, Kelt, OGLE, and more.[1] This dataset was found on Kaggle and originally contained 97 different columns of data; however, only five columns from the original dataset were used for this project. In addition, a new column called “row” was introduced to create a unique identifier for each row in the dataset. The six fields used in the project are as follows:

• *row* - a unique integer; indicates the row in the database.
• *name* - a string; indicates the name of the exoplanet.
• *mass* - a double; represents the mass of an exoplanet (in terms of Jovian mass J_M).
• *planetaryRadius* - a double; represents the radius of the exoplanet (in terms of Jovian radii J_R).
• *orbitalPeriod* - a double; represents the orbital period (in Earth days) of the exoplanet around its host star.
• *discoveryYear* - an integer; represents the year in which the exoplanet was discovered.

The entries in this dataset are ordered numerically by *row* in ascending order.

The primary reason I chose this dataset is because I have always been fascinated by space, especially exoplanets and extrasolar objects. I wanted a dataset that would allow me to analyze and explore the various properties and attributes of different exoplanets.

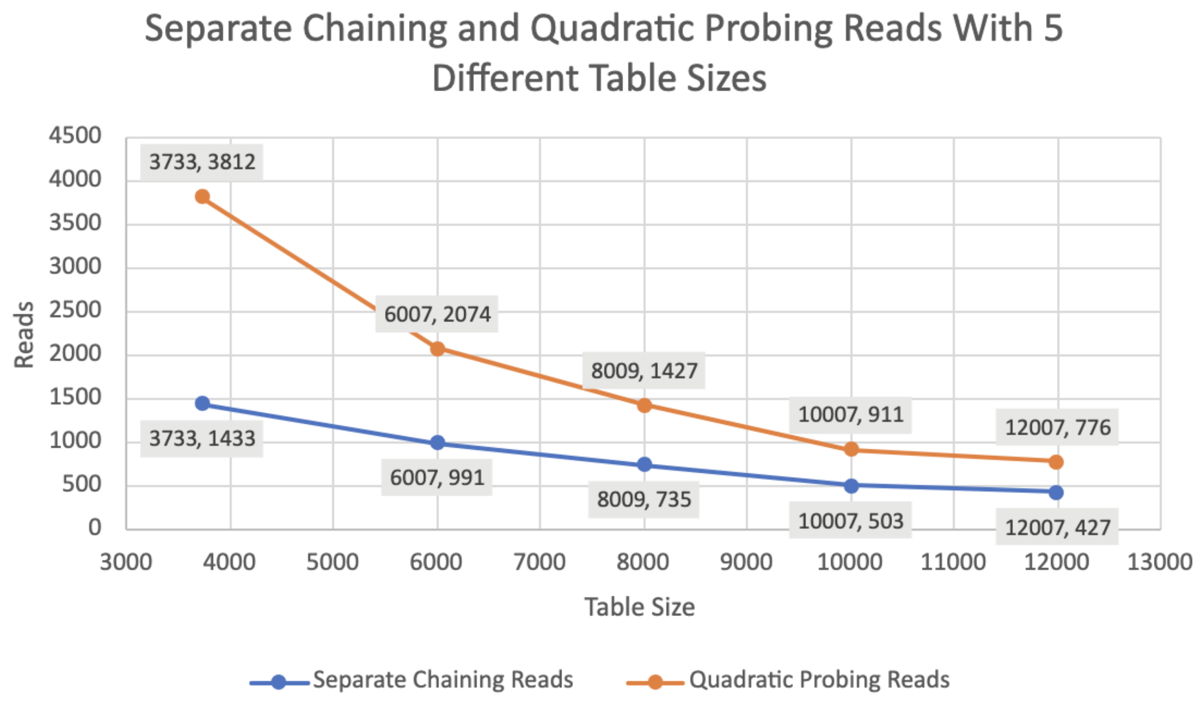
Hashing With Varying Table Sizes Results

To test how different table sizes affect the number of reads required to insert a value into a hash table for separate chaining and quadratic probing, I used the initial table size values of 3733 (size of data set, which is 3732 elements, rounded to the next prime number), 6007, 8009, 10007, and 12007, which are all prime numbers.

After running tests on all five values, the reads for separate chaining and quadratic probing, as well as the final table sizes were collected and are displayed in the table and graph shown below:

[1] Original Source: http://exoplanet.eu/
Kaggle: https://www.kaggle.com/eduardowoj/exoplanets-database

Initial Table Size	Separate Chaining Reads	Quadratic Probing Reads	Final Table Size (Quadratic Probing)
3733	1433	3812	7477
6007	991	2074	12037
8009	735	1427	8009
10007	503	911	10007
12007	427	776	12007
Average Reads (Separate Chaining): 817.8		***Average Reads (Quadratic Probing):*** 1800	
Maximum Reads (Separate Chaining): *1433*		***Maximum Reads (Quadratic Probing):*** *3812*	



What is the maximum and the average number of reads?

The maximum and the average number of reads for separate chaining were 817.8 and 1433 reads, respectively. As for quadratic probing, the maximum and the average number of reads were 1800 and 3812 reads, respectively. Overall, separate chaining had fewer reads on average, as well as a smaller maximum number of reads compared to quadratic probing.

For the probing hash tables, what is the size after inserting all of your elements?

Two out of the five table sizes tested with quadratic probing required rehashing, which involves doubling the initial table size n and rounding to the nearest prime number p where $n < p$, since the number of Exoplanet objects inserted into the tables were more than half the size of the initial table size. The tables which required rehashing to finish inserting all of the Exoplanet objects were the tables with initial sizes of 3733 and 6007. After inserting all of the elements into these two tables, they had final table sizes of 7477 and 12037, respectively. The three other tables which had initial sizes of 8009, 10007, and 12007 all remained the same size after inserting all of the elements.

Which hash table size is best for the data set?

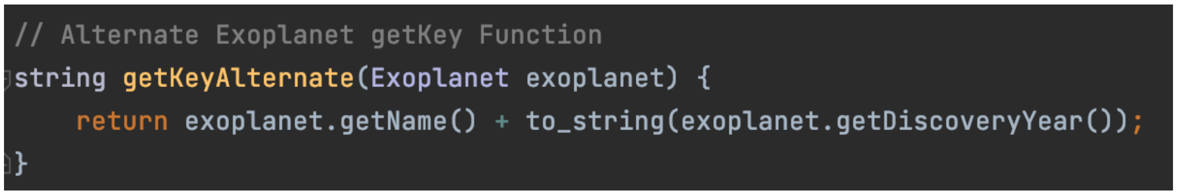
Out of the five different hash table sizes tested, the optimal hash table size in terms of the number of reads required to insert data is the hash table with the size of 12007. The hash table with size 12007 required only 427 reads for separate chaining and 776 reads for quadratic probing, which were both the smallest number of reads out of all the table sizes tested.

Modified getKey & Hash Function

For this experiment, I modified the *getKey* function, which I called *getKeyAlternate*, and the hash function, which I called *modifiedHornerHash*, to determine which combination (normal horner hash, normal *getKey*; normal horner hash, *getKeyAlternate*; *modifiedHornerHash*, normal *getKey*; *modifiedHornerHash*; *getKeyAlternative*) is optimal for the Exoplanet dataset used in this analysis.

Modified getKey Function

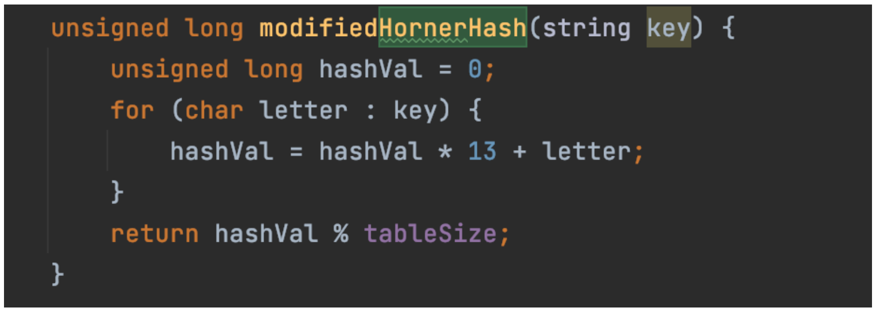
The modified getKey function *getKeyAlternate* concatenates the name of the Exoplanet object and the discovery year for that Exoplanet object. The code for this modified getKey function is shown below:



```
// Alternate Exoplanet getKey Function
string getKeyAlternate(Exoplanet exoplanet) {
    return exoplanet.getName() + to_string(exoplanet.getDiscoveryYear());
}
```

Modified Horner Hash Function

The modified Horner Hash function *modifiedHornerHash* changes the normal Horner Hash value from 37 to 13. The code for the modified Horner Hash function is shown below:



```
unsigned long modifiedHornerHash(string key) {
    unsigned long hashVal = 0;
    for (char letter : key) {
        hashVal = hashVal * 13 + letter;
    }
    return hashVal % tableSize;
}
```

Hashing With Varying getKey Functions & Hash Functions Results

The most efficient table size with the least amount of reads from the previous tests was 12007. Therefore, I used 12007 for each of the three tests for the combinations of modified getKey and Hash functions. Below are the read results for the three different getKey and hash function combinations:

Modified getKey & Normal Horner Hash

The test on the combination with the modified getKey function *getKeyAlternative* and the normal Horner Hash function *hornerHash* yielded 496 reads for separate chaining and 636 reads for quadratic probing.

Normal getKey & Modified Horner Hash

The test on the combination with the modified Horner Hash function *modifiedHornerHash* and the normal *getKey* function yielded 321 reads for separate chaining and 630 reads for quadratic probing.

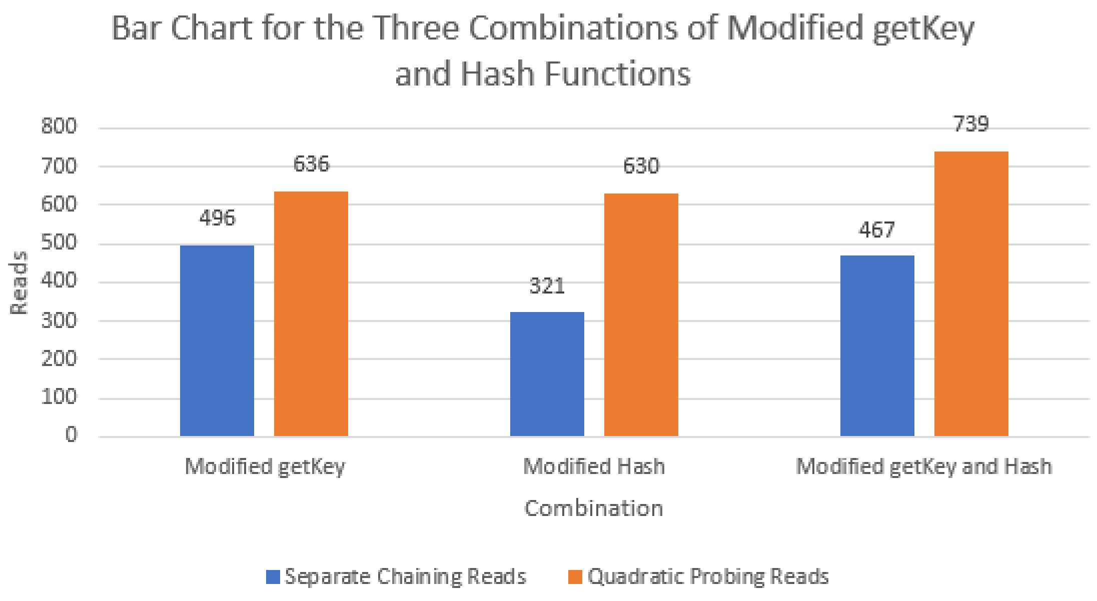
Modified getKey & Modified Horner Hash

The test on the combination with the modified *getKey* function *getKeyAlternate* and the modified Horner Hash function *modifiedHornerHash* yielded 467 reads for separate chaining and 739 reads for quadratic probing.

Overall Results

Below is a summary of the results for the tests for all three combinations:

Combination	**Table Size**	**Separate Chaining**	**Quadratic Probing**
Modified getKey Function	12007	496	636
Modified Hash Function	12007	321	630
Modified getKey & Hash Function	12007	467	739



What is the effect on the placement of elements in the hash table?

The modified getKey function with the original hash function likely resulted in more collisions for separate chaining, making the linked lists longer on average. Because of this, more reads were required to insert elements since we needed to transverse longer linked lists. However, for quadratic probing, there was likely less primary clustering, resulting in fewer reads.

The modified hash function with the original getKey function likely resulted in much fewer collisions (more dispersed placements), making there be smaller linked lists and less traversing (reads) required to insert elements. In addition, this likely made there be even less primary clustering for quadratic probing, resulting in even fewer reads.

The modified hash and getKey functions did not work very well together. Their combination likely resulted in a lot of clustering compared to the modified hash and regular getKey combination. Therefore, the reads for separate chaining and quadratic probing were made worse than the modified hash function/regular getKey combination and slightly better than the modified getKey/regular hash function combination for separate chaining but worse for quadratic probing.

Which hash/getKey combination is best for your data set?

The modified getKey function with the original hashing function had 496 reads for separate chaining and 636 reads for quadratic probing. The number of reads for quadratic probing improved over just using the original *getKey* function and original hash function; however, the number of reads for separate chaining was worse compared to the original *getKey* and hash functions.

The modified hashing function *modifiedHornerHash* with the original *getKey* function definitely improved the number of reads for separate chaining compared to the previous combination considering it only required 321 versus the 496 that the modified getKey required. It also slightly improve the number of reads for quadratic probing as the number of reads decreased from 636 to 630.

Interestingly, using both a modified getKey function and modified hashing function required 467 reads and 739 reads. This means it yielded worse results than just using a modified hash function for both separate chaining and quadratic probing, but it yielded better results for separate chaining and worse results for quadratic probing compared to just using a modified getKey function.

Overall, the optimal hash and getKey combination for the Exoplanet data set is the one with the **modified hashing function *modifiedHornerHash* and original *getKey* function** since it required the least amount of reads for both collision detection methods. This implies that the modified horner hash function *modifiedHornerHash* was better suited for this data set while the modified getKey function *getKeyAlternate* was worse for this data set compared to the original.

Analysis

Consider how removing and searching compares with inserting, in terms of the number of hash elements read. Would they read more, less, or the same?

Based on how the removing (*remove* function) and searching (*find* function) operations are programmed and designed to work, we can get an idea as to how the number of reads required for removing and searching compares with inserting in a hash table.

For separate chaining, the number of reads required for both removing and searching will likely be similar to inserting since all three rely on searching through the linked list at the specified index. The reads for the *insert* function for separate chaining come from the required calls on the *find* function which is the same as the *remove* function (except for the removal part), so it is likely that all operations will have a similar number of reads for inserting, removing, and searching.

For quadratic probing, searching and removing will both likely require fewer reads compared to the number of reads required for inserting into a hash table. The *insert* function requires calling the *find* function. However, during an insertion the table may need to be rehashed, which increases the number of reads required to insert an object significantly. Therefore, independently, the *find* function will likely require fewer reads than the *insert* function. Additionally, the *remove* function is independent of the *insert* function, but, like was the case for separate chaining, it functions very similarly to the *find* function (except for the deletion part). Therefore, removing in a table using quadratic probing will likely require fewer reads than inserting.

How often would each of the three operations (inserting, removing, searching) typically be used with your data set? Which hash collision method works best for your data set? Why?

This exoplanet data set is very dynamic as new exoplanets are being discovered every day throughout our galaxy across various active exoplanet search projects. Therefore, inserting would likely be the most used operation for this data set while searching and removing would be the least used since it is much less likely (or less important) that someone would need to search for an exoplanet or that a discovered exoplanet would have to be removed from the data set.

Considering how often each operation (inserting, removing, and searching) would likely be used for this data set, the hash collision method that works best for this data set is **separate chaining**. This is because separate chaining is very good at making fast insertions since it just appends collided objects to linked lists at each index in the table. In addition, the results show that using separate chaining consistently required fewer reads compared to quadratic probing when inserting Exoplanet objects into the hash tables across all tests, including the tests with different combinations of getKey and hash functions.