**A Comparison of Linear, Double, and Quadratic Hashing and Their Uses**

Brandon Brodzinski

Department of Computer Science

Valdosta State University

1500 N Patterson St Valdosta, GA, United States

E-MAIL: [bjbrodzinski@valdosta.edu](mailto:bjbrodzinski@valdosta.edu)

***Abstract—* Hashing is the process of turning a given key into another value. Many computer programmers utilize this process by implementing hash functions into their code. The hash function is a mathematical algorithm that creates an output called the hash value which is stored in a hash table. Collisions happen when two different inputs generate the same output after going through the hash function. If multiple collisions are happening at once clustering can be encountered since multiple elements are being put next to each other. Three different types of hash functions deal with collisions in different ways: linear hashing, double hashing, and quadratic hashing. In this paper, we compare and contrast how these algorithms can be implemented, the pros/cons of each algorithm, and when they should or should not be used.**

***Keywords— Linear Hashing, Double Hashing, Quadratic Hashing, Hash Function, Hash Value***

**I. Introduction**

Hashing is a special kind of programing function that takes a variable input size and changes it to a fixed data size. In other words, hashing is a way to compress data. This is done by running an input though a function, called a hash function, to produce a reduced value that is more easily stored. The first designs of hashing functions were created sometime in the late 1970s and 80s, with even more hashing functions emerging in the 90s. However, most of these hashing functions were discarded because of their security flaws.

Hashes are usually composed of three different parts. These are the hash value, hash function, and value or input. The basic hash function is similar to: hashValue = (hashFunction \* value) % amountOfValues. The value, or input, is whatever you either input into the hash function or have altered by the hash function, in this case multiplied by, to be compressed. The hash function is how you compress your input. Each hashing method, in this case linear, double, and quadratic, has a different hash function that will be discussed later in this paper. Finally, the hash value is the output or result you get from hashing a value. The basic hash function is very limited in its use. Some hashing functions alter or deviate from this basic hash function, however the idea is the same.

With the around 50 years that hashing has been around, there have been many different variations that have been developed to solve some of hashing’s problems. These problems include collisions and cluttering. Collision is where you input two different values into a hash function, but get the same output for both inputs. This is because data is being compressed into a narrow range, where some values can overlap. All three hashing methods, linear, double, and quadratic, solve this issue in varying ways, though there are more ways than those to solve collisions. The second issue is that hashing often has problems with cluttering. Cluttering is when you have many different inputs trying to be assigned the same hash value. If you use an array with size 50, as an example, you may have twenty hashes centered on index 10, but every other index is empty or you may have one or two other hashes at the end of the array with nothing in the middle. Like with collision, the three hashing methods discussed in this paper solve this in different ways, though there are more than those methods to help with cluttering.

**II. Background**

We decided that one implementation of hashing we would use would be on ISBNs, or International Standard Book Numbers. ISBNs were introduced in the 70s by Gordon Foster for the booksellers W. H. Smith. As they began to move to a computerized system in 1967, they decided they needed a quick and easy way to identify books, which gave birth to the Standard Book Numbering system (SBN). Coincidentally, at the same time, the International Organization for Standardization (ISO) Technical Committee on Documentation (TC 46) was looking for just that kind of system to use internationally. After a meeting was held, the ISBN was adopted internationally, and is being used to this day.

Since the primary function of ISBNs is to store books digitally, it makes sense that you would want a way that made it quick and easy to store them. Because the number is so large, it would be a lot of storage space to store the entire number in something like an array. Since the number needed to be shortened to save space, it is the perfect candidate to demonstrate how hashing works.

When you encrypt data, the plaintext password is run through some sort of software that turns the data into another form, so that only people with the decryption key can access it. The plaintext password is the password as is, with no sort of change made, so sending it to the server to be stored with no protection is very dangerous. When the password is created and protected, it needs to be stored within the website’s servers somewhere so it can be referenced when the user attempts to log in. If encryption is used, the password travels to the system where it will be stored and is decrypted upon arrival. This is a safeguard so that if the data is intercepted in transport by someone who should not have access to it, it will be an unencrypted mess. However, if the person who intercepts the data can figure out the decryption key, they can now decrypt the data and know the plaintext passwords of the users. This is a two-way form of encryption; what is encrypted can be decrypted.

Hashing is one way, meaning that it is not decryptable. When using hashing to store a password, the password can be run through a hash function, and the produced hash is sent to the website to be stored. That way, if it is to be intercepted, there is no way of knowing what the password could have been, as it is just a set of numbers. So now, when the password is entered to log in, the thing being referenced for verification is not the plain text password, but rather the hash of the password. No private data ever has to make the trip to the server side.

There is a fundamental difference between ISBNs and passwords. Since ISBNs are a standardized system used worldwide, every single one is unique. However, since passwords are user-generated, they are often repeated between separate users unknowingly. In fact, in 2020 it was revealed that 103,170,552 people are sharing the same password of “123456”. That is why, to combat the chance of two identical passwords being hashed into the same thing, the process of salting is used. Salting, which will be explained in more detail, appends unique bits to the end of the original password so that no two passwords are the same. Even if two accounts use the same password, it is not possible for them to use the same username, for obvious reasons. That’s why the username will generate the salt to be used to ensure no confusion will arise when storing passwords.

**III. Algorithm**

***1)*** ***Linear Hashing***

Linear Hashing is a dynamic data structure invented by Witold Litwin in 1980 and has been analyzed by Baeza-Yates and Soza-Pollman [2]. This type of algorithm resolves collisions in hash tables by linearly putting the hash value in the next open slot in the array. The linear hash function pseudocode is as follows: H(x, i) = (H(x) + i) % numberOfValues. With linear hashing, there will be better cache performance but more likely to run into clustering issues. The reason for this is because as multiple values go through the hash function and encounter a collision it is moving over + i. This means that it will take longer to find an open slot since it has to go over the array one by one until it finds a spot. The advantage of linear probing is that it requires less memory and is not as complex as the other functions which leads to easier implementation [3]. The disadvantages are primary clustering ***—*** the tendency for some collision resolution schemes to create long runs of filled slots near the hash function position of keys***—*** and long probe sequence times [3]. An example of where linear hashing could be implemented effectively is in network routers where memory is scarce. Since linear hashing doesn’t require any new allocations, elements can be placed confidently into the hash table with no risk of a malloc fail.

***2)*** ***Double Hashing***

Double Hashing was made to help solve the common issue of collisions in hashing. While it is not perfect, it is more effective than some other hashing methods. Double hashing has a higher computational cost than linear hashing but is quicker when dealing with collisions. This is because double hashing requires the use of two hash functions instead of one, hence the name. Essentially, the double hashing function is: hashValue = (firstHash + (numberOfCollisions \* secondHash)) % numberOfValues. What makes this function different from, for example, linear hashing, is the use of the section: (numberOfCollisions \* secondHash). Double hashing keeps track of the number of collisions a certain value has when it is put through the function. If there are no collisions, then this section of the function is irrelevant, but it still requires computational power, hence the higher computational cost compared to linear hashing. However, if there is a collision, then numberOfCollisions is incremented and the hash function is run again, where the second hash now affects the final hash value. Once the first hash and the section involved with collisions have been calculated and added together, the result goes through modular division by how many total values you are going to put through the hash function. The final result is your hash value.

***3)*** ***Quadratic Hashing***

Quadratic Hashing falls in between linear and double hashing in terms of clustering and performance. This is because it does not require a second hash function like in quadratic hashing, hence saving computational cost, but still more costly than linear hashing. The basic quadratic hash function is hashValue = (hash + (numberOfCollisions^2)) % numberOfValues. This shows why it is called quadratic hashing, because the number of collisions a hash value has alters the hash function like a quadratic function. One collision would give you 1^2, three collisions would give you 3^2 and so forth. So, once the hash has been added to the number of collisions squared, the hash goes through modular division by the total amount of values you are going to hash. This gives you the final hash value.

4) ***Salting a Hash***

In Cryptography, salting refers to adding random data to the input of a hash function in order to guarantee a unique set of outputs [1]. Salting is used in order to provide a more secure log-in experience for users. When every hash is unique it becomes harder for hackers to detect a pattern in your hashing process and break-in. In order to make sure salts generate unique hashes for every user, we need to make sure that every salt is unique. If we had attached the same salt to every user when two users entered the same password they would get the same salt attached resulting in longer, but still, identical passwords to be hashed. In our second project where we created a log-in system to display the security benefits of salting hash functions, we created salts based on the user’s log-in name in order to ensure unique salts for every user. This way even if two users had the same password their different usernames would generate unique hashes for them. Fig. 1 shows how two identical passwords can end up with different hashes due to unique salting.

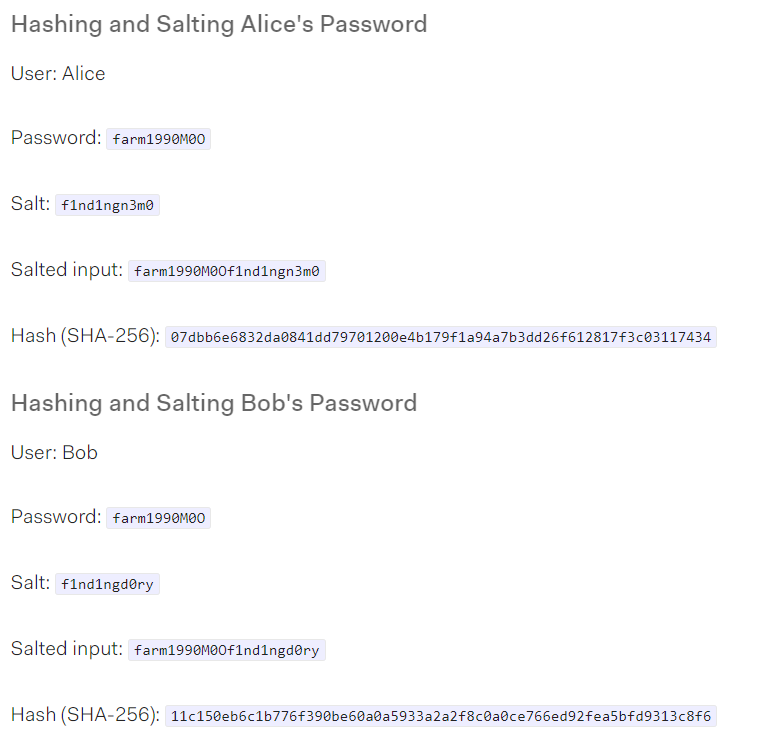


Fig.1 Two users with the same password having different output hashes [1].

**IV. Implementation**

In order to decide what hash function we wanted to use for a cryptography program we decided to first make a program that would allow us to compare the three different hash functions. We decided the best way to go about this is by submitting data sets of different sizes to the three different hash functions. While processing the data the program tracked the time it took for the data to process and the amount of collisions each function had. We later would use the time and collision count to draw comparisons between the different functions. In Fig. 2 you can see how in order to track time the program would record the current time when the function begins and then subtract it from the time when the function ended. In order to track collisions we simply had a variable that was set to zero that would increase by 1 every time a collision occurred. This variable was then set back to zero when the function completed and the collision count was printed before moving on to the next hash function.



Fig.2 Function to gather data on how efficient linear hashing is.

We then decided based on our findings from the hash comparator tool to use the quadratic hash function for our second project where we explored the use of hashing and hash functions for cryptology. To begin with we would store a list of usernames and passwords entered by different users. Then in order to allow multiple users to have the same password we would use the username as a salt. To do this we would concatenate the username and password then hash the combined string in order to get the hashed password for the user that would be stored in a separate list. Unfortunately a hashed password ends up looking like a very long confusing looking series of numbers. In order to remedy this and store them in a way that makes it more convenient for the programmer to work with we employed the quadratic hash function here to make a more organized table that would hold the hashed passwords. In Fig. 3 you can see the hashing function that used a simple hash function, and a quadratic hash function in case of collisions, in order to convert the array of hashed passwords into a new array.

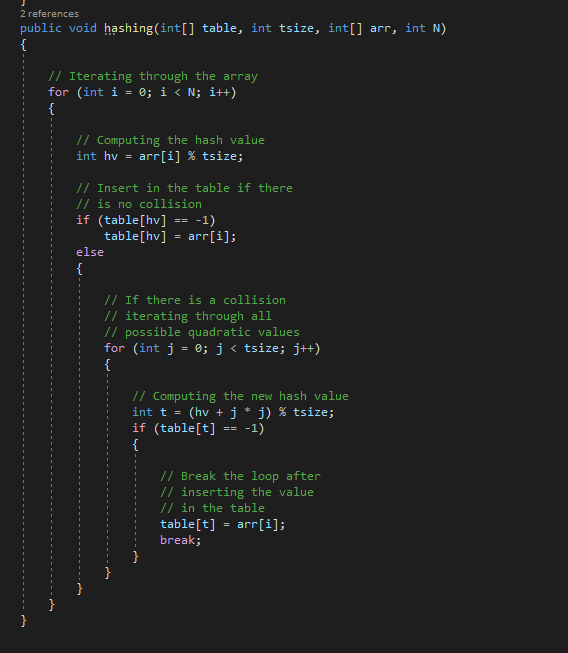
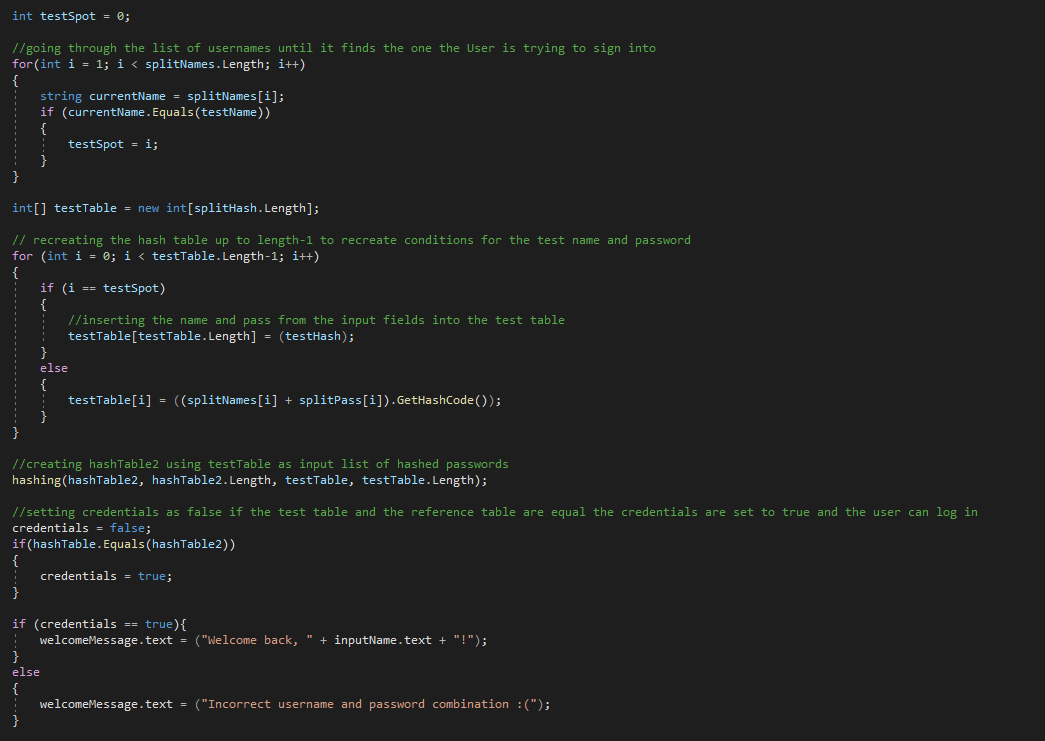


Fig.3 Hash table creator using quadratic hash function.

In the future if a user wanted to log in the way we would verify their credentials is by taking their username and finding which username it matched with in the username list. We would then put all the names and passwords into a hash function, except when we reach the username that matches the one the user imputed instead of using the password from the list we would use the one they submitted. After all the names and passwords are sorted this table is then compared to the original hash table we built with our store names and passwords. If the two tables matched then the user can login. This process can be seen in Fig. 4 where the location of the username is recorded then put through the hashing process in order to compare the two tables.

Fig.4 User credential verification system.

**V. Results and Analysis**

When analyzing the run times of the different hash types, there are some obvious performance issues. While these exact runtimes may differ depending on the specifications of the PC it is being run on, the correlation to each other hash types will be the same. The obvious outlier is Double Hashing. Double Hashing has the worst run time out of all the hashing types by a lot. There is an exponential increase in runtime as the number of ISBNs being input increases. Quadratic and Linear stay close to each other, but a performance gap presents itself as the ISBN count gets higher. Quadratic stays at a low runtime as input goes up, but Linear begins to increase noticeably.[2]

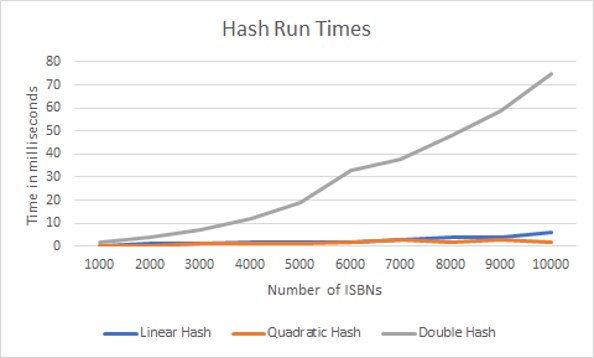
****

Fig.5 Runtime analysis of all hashing methods [2].

When examining the collision count however, the graphs show different placings. Linear has a much higher collision count than the other hash types. Quadratic and Double stay close, but Double has less collisions by a noticeable amount. With all of the counts, there is a pretty distinct upward line. The collision count is very predictable, even more so than run times.[3]

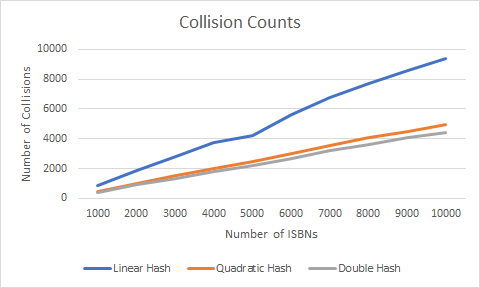
****

Fig.6 Collision count analysis of all hashing methods [3].

So why does Double have such a large runtime gap? When looking at how Double actually performs its hash functions, it performs two calculations to hash (hence the name). Because there is a huge number of computations that it has to perform, the runtime is naturally higher. However, because Double creates an extremely unique hash code for each value passed through, it has the lowest collision count out of the three methods we tested. Linear keeps a low runtime for most of our testing, but we see an increase at the end. However, Linear has the highest collision count of our tested methods. The reason we see this is because of Linear’s addition based function, which is a very fast equation. Because the method handles collisions by checking the next available slot, a lot of clustering occurs, which results in high collision counts and higher runtimes. Quadratic’s statistics stay tame throughout testing, with it being the overall fastest method and having the second most collisions of the bunch. Because of its multiplication based function, there is a nice balance of unique outputs and low run times.

Knowing this, we can decide what situation is best for each method. Linear works best with smaller data sets or data sets where you can predict a low collision count. Double works best for when you want low collision counts, but as time is not a factor. Quadratic is the best all around choice, providing a nice balance of speed and low collision counts

**VI. Conclusion and Future Work**

In conclusion, after investigating all three hashing functions we have decided that quadratic hashing would most likely be used. Linear probing has better cache performance but has the most clustering issues. Doubling probing has no clustering problems however has poor cache performance overall. Therefore quadratic hashing falls in between linear and double hashing in terms of clustering and performance. The formula being used H(k, i) = (h'(k) + i \* i) allows for the idea to skip regions in the table to avoid possible clusters. In the future, we would like to explore more advanced forms of encryption for our password project. This would create more layers of security while using hash tables to keep everything organized. Another possibility we would like to look into is what would be the benefits of using the slowest hash function instead of the fastest. If we used double hashing we could theorize the slower hash function could slow down brute force attacks on server security. Looking at the pros of quadratic hashing it would be intriguing to see the contrast that double hashing would have on certain situations. As a result of our study, we saw quadratic hashing has many uses for people to utilize in their code and gives a well balanced form of clustering and performance making it the most viable option to use out of the three.

**References**

**[1] Arias, D. (2021, February 25). *Adding salt to hashing: A better way to store passwords*. Auth0. Retrieved from** [**https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.**](https://auth0.com/blog/adding-salt-to-hashing-a-better-way-to-store-passwords/.)

**[2] *Linear Hashing*. DBpedia. (n.d.). Retrieved November 19, 2021, from https://dbpedia.org/page/Linear\_hashing.**

**[3] Priya, B. P. (2021, July 8). State the advantages and disadvantages of collision resolution strategies. Tutorials Point. Retrieved November 19, 2021, from** [**https://www.tutorialspoint.com/state-the-advantages-and-disadvantages-of-collision-resolution-strategies**](https://www.tutorialspoint.com/state-the-advantages-and-disadvantages-of-collision-resolution-strategies)

**[4] *What is double hashing?* Educative. (n.d.). Retrieved November 19, 2021, from** [**https://www.educative.io/edpresso/what-is-double-hashing**](https://www.educative.io/edpresso/what-is-double-hashing)**.**

**[5] ClaudiuClaudiu 211k155155 gold badges458458 silver badges660660 bronze badges. (1957, February 1). *Difference between hashing a password and encrypting it*. Stack Overflow. Retrieved November 20, 2021, from** [**https://stackoverflow.com/questions/326699/difference-between-hashing-a-password-and-encrypting-it**](https://stackoverflow.com/questions/326699/difference-between-hashing-a-password-and-encrypting-it)**.**

**[6] NordPass. (n.d.). Top 200 Most Common Password List 2021. https://nordpass.com/most-common-passwords-list/**

**[7] *History*. ISBN.org. (n.d.). Retrieved November 20, 2021, from** [**https://www.isbn.org/ISBN\_history**](https://www.isbn.org/ISBN_history)**.**

**[8]Golden, A. (n.d.). What is the History of the ISBN? Books Tell You Why.** [**https://blog.bookstellyouwhy.com/what-is-the-history-of-the-isbn**](https://blog.bookstellyouwhy.com/what-is-the-history-of-the-isbn)