Thomas Hudson

Professor James Pepe

CS401

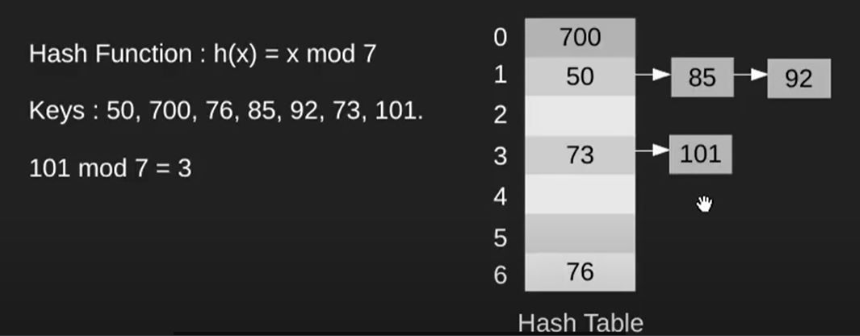
Jun 16, 2020

Hashing Algorithms and How to Use Them

**What is a Hashing Algorithm?**

A hashing algorithm is a function that generates a reproducible but seemly random hash value or hash code based on an input of variable sizes. A usable hashing algorithm must perform in three ways. The generation of the hash should be as quick as possible while also performing a process that is very difficult to reverse. The algorithm should exhibit what is known as the avalanche effect: a single change in input results in a large change in output. Lastly, and most importantly, the algorithm should prevent collisions. That is, no two distinct inputs should yield the same output from the algorithm (an output collision), while the same input should always yield the same output.

These requirements are based on common use cases for hashing algorithms. The default usage for a hashing algorithm is within a hash map. Hash maps are a common data structure used to create efficient key-value pair lookups. In a hash map, each piece of data (most commonly an object in object oriented languages) receives a key. This key is unique to the object and is used to search the hash map for the object. For example, a person’s name could be a key. The hashing algorithm takes in the key and generates a hash of that key that corresponds to a memory location where the data will be stored. Below is a very simple example of a hash table.



In this case, the hash function takes the number to be inserted into the hash table and returns the remainder when dividing by seven. This remainder corresponds to a location in the hash table. To understand why hash maps are so impressive we must consider the primary operations insertion, searching, and deletion in comparison to alternative data structures like linked lists.

Searching in a hash map ideally consists of a single operation yielding a Big-O of O(1) (constant time). This is done by hashing an input key which yields a memory location of the target if the memory location exists. This process is the same regardless of the size of the hash map. In an array, searching involves comparing the input with every item contained in the array. This yields a Big-O of O(n), linear time. Since deletion and insertion also involve locating the target or its space, these operations within an array again results in a Big-O of O(n) while deletion within a hash map remains at constant time. It should be noted however, deletion and insertion from many simpler data structures like stacks, queues, linked lists, and doubly linked lists can match the efficiency of hash maps in deletion and insertion operations. In summary the hash map’s primary benefit in terms of time complexity lies within the search operation. However, this efficiency relies on a strong hashing algorithm.

In order for a hash map to function as has been described, a hash map requires that each key results in a unique location, else, the hash map suffers from what is called a collision. Collisions take place when two or more items are mapped to the same location within the hash map. Generally a collision can be handled by the hashing algorithm. Most commonly a linked list data structure is created within each container as shown above. However, as I just stated, linked lists are far less efficient for searching at a time complexity of O(n). Thus, the higher the number of collisions within a hash map, the less efficient the data structure becomes. This leads us into the pigeonhole principle. In mathematics the pigeonhole principle states that if n items are put into m containers if n is greater than m there must be more than one item in at least one container. Similarly to this, a hashing algorithm has a finite number of potential outputs. In our example above the hashing algorithm has just seven outputs. Thus, even under perfect conditions, a collision will occur after adding more than seven items, most likely, much sooner. The solution then, is to create a hashing algorithm with the highest number of spaces possible. Unfortunately, the more outputs that a hashing algorithm has, the more complex the hashing algorithm becomes and the more operations hashing a key requires. In the worst case scenario hash maps yield a Big-O of O(n) for searching, deletion, and insertion.

Beyond use in hash maps, hashing algorithms have several other purposes such as message verification, and data security. In regards to message verification, hashing algorithms can take a file and create a “digital signature” by hashing the file (Pound). Similarly to a written signature, a digital signature is intended to be a unique identifier of a particular file, just as a unique location is given to each item in a hash map. This digital signature is used in two key scenarios. First, the digital signature of a file can be used to check if the contents of the file match what is expected. This prevents the addition of malware to a file going undetected. Second, the digital signature can be used to verify that the file was transported successfully. After transmission, if the hashing algorithm yields the same signature, it can be assumed that the files integrity was maintained. This relies on the avalanche effect requirement for hashing algorithms, even a small change in input should result in a large change in the output from the algorithm. Many common hashing algorithms have been broken (more on this later) so this method of data signature verification should only be trusted with the most current hashing algorithms.

The last common use for hashing algorithms is in data security. A common naïve approach to authentication is to use a hashing algorithm to hash the password that the user enters. This way you are not storing the user’s password, and since hashing algorithms should not be easy to reverse, storying and verifying passwords in this manor is mildly secure but this should be taken with a grain of salt. The problem with this method stems from rainbow tables. Rainbow tables, are tables that have been created from leaked password lists containing hashed versions of common passwords (Pound). If a password in your database is within this list, the intruder can look up the password in the table and gain access to the user’s data. The best option when using hashing algorithms for authentication takes in the grain of salt for you in a method called hashing and salting. Instead of inputting a password directly into the hashing algorithm a random string of characters unique to each user (the salt) is added to the password thus making the password infinitely less likely to be found in a rainbow table (Scott). Now that the uses of hashing algorithms have been explored we will take a deeper look at their origins and some generalizable implementations.

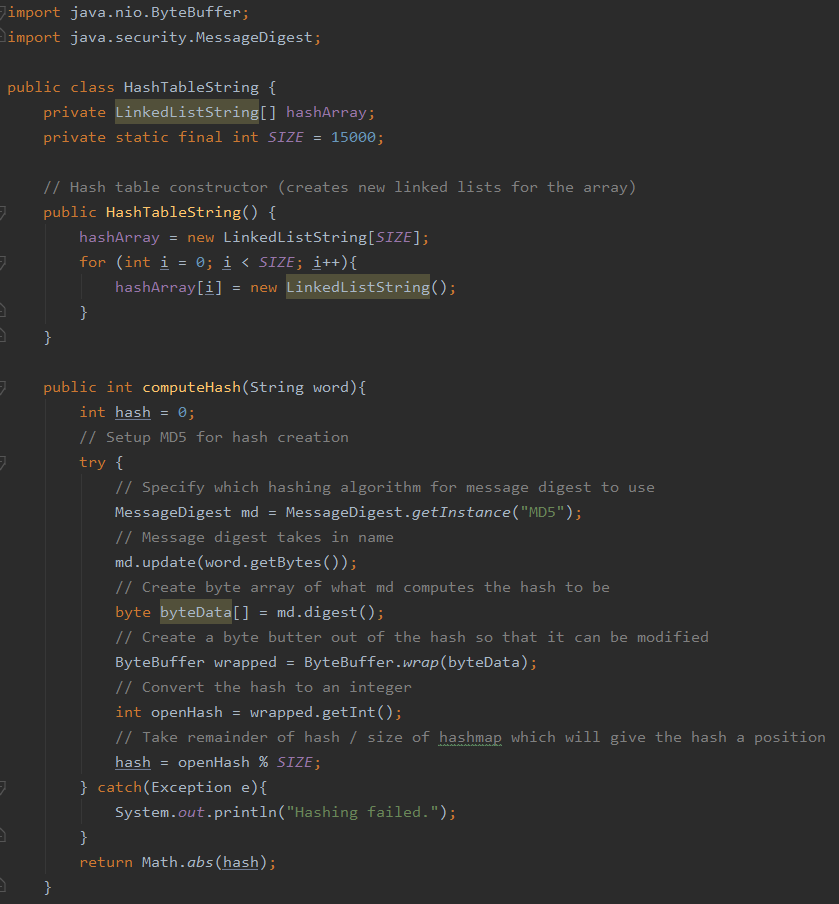
**Story of Hashing Algorithms**

The term hash in computer science was originally coined by Hans Peter Luhn of IBM in January of 1953, referring to a function that would create a hash of a file that would allow the file to be compressed and save space. These functions would evolve into what we know today as hashing algorithms. The hash algorithm as we know it did not come into existence until the late 1970’s when Diffie and Hellman identified the need for a one-way hash function used to create a digital signature of a document. The second identifier of what we know as a hash function, collision resistance, was not formalized until 1987. Two years later, in 1989, the first true hashing algorithm was released, MD2 written by Ronald Rivest (Preneel). MD2 was the precursor to modern hashing algorithms like SHA2.

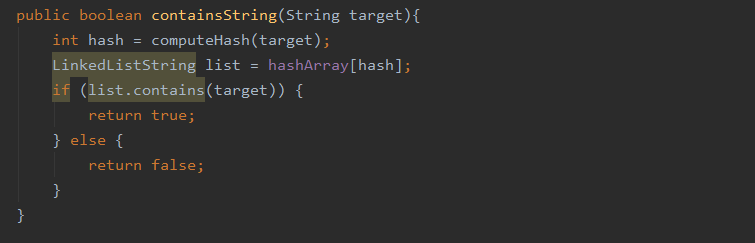
The world of hashing for security is essentially an arms race. As computers become more powerful, the speed with which they can crack a hash is reduced. Thus, every few years hashing algorithms must be updated to increase complexity such that brute forcing the hash is not physically possible within a reasonable amount of time. Secondly, even if a hash cannot be brute forced, a hashing algorithm can also be “broken”. A hashing algorithm is “broken” if it is possible to purposefully recreate a hash. This would break the requirements for the use of a hashing algorithm as a digital signature as malware could now be added to a file in such a way that it retained the same digital signature. A great example of this kind of attack took place in 2017 in a hashing collision attack of SHA-1 known as SHAttered (Scott 2:34). This attack was created by Google to demonstrate the dangers of using SHA-1. The current standard for hashing for security is the SHA-2 family of hashing algorithms which produce hashes between 224 and 512 bits, but older algorithms are still used for other purposes like hash maps which I will show next.

**Hashing Algorithm Implementation in Java**

In Java, the usage of hashing algorithms is done with the message digest class of the security library. The message digest class allows for the usage of many algorithms most commonly MD5, SHA-1, and SHA-2. For the first implementation I will use MD5 to create hashes of words in a spell checking program. The program creates a hash map of the 10000 most common words in the English language, reads in a file, and checks if words within a file are contained within the hash map. If the words are not within the hash map they are capitalized and returned in an output file. The most relevant section of the program is the function which creates a hash of a given word is shown below.

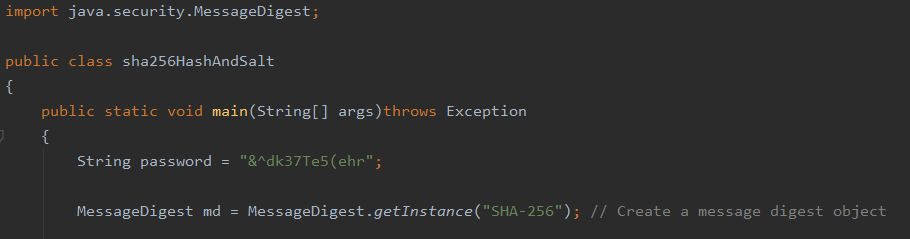


The compute hash function begins by initializing the hash of type integer that will be returned. Due to the potential for a no such algorithm exception, the setup of message digest for MD5 is contained within a try/catch block. First, thealgorithm used is specified to be MD5 creating an instance of the message digest class. Next, we use the message digest instance’s update method setting the internal hash variable to the byte representation of the given word. Third, the digest method uses the MD5 algorithm to create a hash of the input word stored within a hash array. Due to the immutability of byte objects within object we convert the byte array to a byte buffer which can then be converted to an integer. We then take the remainder of the hash divided by the size of the hash array which results in a value between zero and the size of our hash table’s array (a valid location in our hash table). This compute hash function is used to map each word to a position in the hash table’s array when it is originally created. When words from the input file are read, their hash is computed, the same hash that would have been yielded if the word had been input in the dictionary originally. The hash’s corresponding location (linked list) is checked for a matching word (shown below).



In this example in order to avoid collision the hash table was created with 15,000 containers instead of 10,000, the the hash table is at 66% capacity. Managing space is the second major burden when utilizing hash tables. Upon initialization of a hash table, an array of size n, n being the maximum number of containers the hash table can utilize, is created. In the above example there is a capacity of 15000. If it was a necessity that we increase this capacity the change is not as simple as increasing the size of the internal array. The hashing algorithms would also need to be changed resulting in requirement for all items within the hash table be rehashed. In java, rehashing of a hash table is done automatically at a specified capacity level. This capacity level is often called a load factor, the example above currently has a capacity of 66%. Java’s standard load factor is 75%, thus when more than 11,250 items are inserted into the hash map, the JVM will automatically go through the process of rehashing, something to be weary of when using hash maps (Oracle).

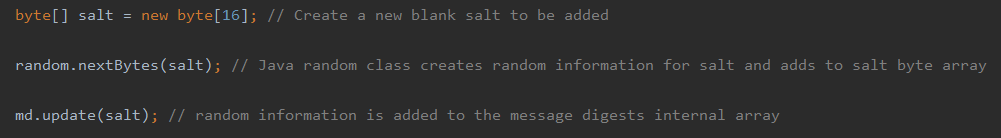
As previously discussed, hashing algorithms are also commonly used for security purposes. Since the creation of a digital signature has essentially been implemented above through the creation of a hash for a hash table, our next implementation will use the hashing algorithm for password storage using the hashing and salting technique. This will elucidate the necessity for hashing and salting over simply hashing user passwords. Similarly to the usage of MD5 for the hash map created earlier the usage of the java class message digest of the security library is used. The current hashing standard is the SHA-2 family, in this case we will be using SHA-256 which provides 2256 potential hashes.



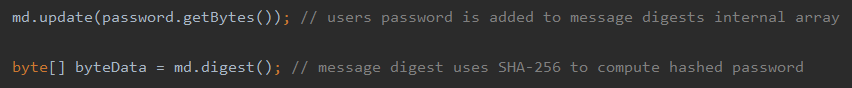
Here we have created an instance of the message digest class which will use SHA-256. Next we will salt the password.

The basic idea of salting a password is to add random information to the password which significantly strengthens the password by essentially creating another randomly generated password for the user. This nearly eliminates the possibility for two users to have the same password. This is important because two or more users having the same password significantly increases the chances of a password being broken. This was demonstrated in a hack on Adobe in 2013 in which password hints were stored with passwords. When brought together these hints allowed hackers to guess the common password.

Java also provides a class within the security library for generating random information that can be used for salting with the secure random class. We use this class to generate a random byte array.



This array is then added to the message digests class instance’s internal byte array with the update method. We then do the same with the input password.



Finally, using the digest method of the message digest class, SHA-256 is used to compute a hash with combined information from the input password and the salt that was generated. It should be noted that the random information that was generated earlier is required to generate a matching hash and check a password. The random information should be stored with usernames and does not necessarily need to be encrypted.

**Conclusion**

The hashing algorithm finds itself at the center of many different computer processes that would not be possible without it. Hashing can be used for file identification, cyber security, and the hash map data structure. It has also inspired other functions like file compression and encryption (Brailsford). File identification allows for reduced amounts of internet traffic by eliminating the need for resending a file. In cyber security hashing provides a strong solution for authentication security, significantly stronger than other solutions like encryption. For databases and quick reference tools that utilize key-value lookup, hash maps provide a significantly more efficient solution. Hashing algorithms are one of the most useful tools in a computer scientists toolbox.

Works Cited

Ajtai, Miklós. "The complexity of the pigeonhole principle." *Combinatorica* 14.4 (1994): 417-

433.

Preneel, Bart. "The first 30 years of cryptographic hash functions and the NIST SHA-3

competition." *Cryptographers’ track at the RSA conference*. Springer, Berlin, Heidelberg, 2010.

Oracle. "Java Security Package." N.p., 11 Mar. 2020. Web. 07 July 2020.

Scott, Tom. “Hashing Algorithms and Security - Computerphile” University of Nottingham*,* Nov

8, 2013.

Scott, Tom. “How NOT to Store Passwords! - Computerphile” University of Nottingham*,* Nov

20, 2013.

Pound, Mike. “SHA: Secure Hashing Algorithm - Computerphile” University of Nottingham*,*

Apr 11, 2017.

Pound, Mike. “Password Cracking- Computerphile” University of Nottingham*,* Jul 13, 2016.

Brailsford, David. “Compression - Computerphile” University of Nottingham*,* Jun 21, 2013.