**Hashing**

Jacob Albrecht  
 IT 378  
Illinois State University  
Normal, IL, USA  
jjalbr1@ilstu.edu

James Cohenour  
 IT 378  
Illinois State University  
Normal, IL, USA  
 jkcohen@ilstu.edu

**ABSTRACT**

Accessing data in an efficient manner is an important part when using a database. With large amounts of data, we must find the most efficient way possible to store and access the information. Consider the problem of searching an array for a given value. If the array is not sorted, the search might require examining each and every element of the array. If the array is sorted, we can use the binary search, and therefore reduce the worst-case runtime complexity to O(log n). In most cases this is sufficient enough, however, we can do better. We can search even faster if we know in advance the index at which that value is located in the array. Suppose we have a function that would tell us the index for a given value. With this function our search is reduced to just one probe, giving us a constant runtime O(1). The idea is that when we store a value at an index, we have a process to remember where that value is located and can then retrieve that value without looking at any other indexes. This process is called hashing and is an important method when retrieving and storing information.

**KEYWORDS**

Hash Tables, Hash Function, Algorithm Efficiency, Collision, Linear Probing, Quadratic Hashing, Double Hashing, Chaining

1. **Hashing Description**Hashing allows you to quickly access data stored in a database. Using a key value, you are able to search for, modify, add, and delete data. If done correctly, this can optimize the database to operate in constant time. A hash system that is perfectly optimized can actually find search results within only 1 or 2 accesses. This is far better than the O(log *n*) average cost required to do a binary search on a sorted array of *n* records, or the O(log *n*) average cost required to do an operation on a binary search tree. As simple as it sounds, hashing can be very difficult to implement. Designers need to pay careful attention to all of the details involved with implementing a hash system.  
     
   A hash table stores items in order to find them easier in future searches. Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0. The mapping between an item and the slot where that item belongs in the hash table is called the hash function. The hash function will take any item in the collection and return an integer in the range of slot names
2. **How It Works**

We first must create a hash function. A well-chosen hash function should produce a unique index for every corresponding value. A basic hash function would be something like taking the remainder of our value when dividing it by the array size and storing it at that index.

Hash Function = Value % ArraySize

if we are given the data set 11, 22, 33, 44, 55, 66, 77, 88, 99, 100 and storing them in an array of size 10, we would store the values at the index produced by the output of the hash function.

11 % 10 = 1 22 % 10 = 2 33 % 10 = 3

44 % 10 = 4 55 % 10 = 5 66 % 10 = 6

77 % 10 = 7 88 % 10 = 8 99 % 10 = 9

100 % 10 = 0

We then store these values in their corresponding index

| 100 | 11 | 22 | 33 | 44 | 55 | 66 | 77 | 88 | 99 |

Using these locations, we will always know where each value is stored by accessing the hash function. If we would like to know the location of value 55, we simple plug the value back into the hash function, producing its location.

Another example could be storing information about a group of people. Let's say we have an array of size 10 and we wish to store the information about a person named Mia and a person named Tim. We can create a hash function that adds up the ascii values of each character in their first name and takes the remainder when dividing the value by the array size.

Hash Function = ASCII Codes % ArraySize

Mia M = 77 I = 105 A = 97 = 279 % 10 = 9

Tim T = 77 I = 105 M = 109 = 298 % 10 = 8

We then store these values in their corresponding index,

|NULL|NULL|NULL|NULL|NULL|NULL|NULL|NULL|Tim|Mia|NULL|

This is extremely convenient when accessing information. If we want to know more about tim, we can find his information by plugging “tim” back into the hash function, producing his location. At this index, we can store other information such as his address, phone #, age, etc.

hashFunction(“Tim”) = 8 hashFunction(“Mia”) = 9

1. **Hashing Algorithms**Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots.   
   1. **Collision Resolution**Collisions occur when two values hash to the same index in the table. If we are careful when selecting a hash function, then the actual number of collisions will be few. Unfortunately, even under the best of circumstances, collisions are nearly unavoidable. To be practical, a database organized by hashing must store records in a hash table that is not so large that it wastes space. To balance time and space efficiency, this means that the hash table should be around half full. We would like to pick a hash function that stores the actual records in the collection such that each slot in the hash table has equal probability of being filled.

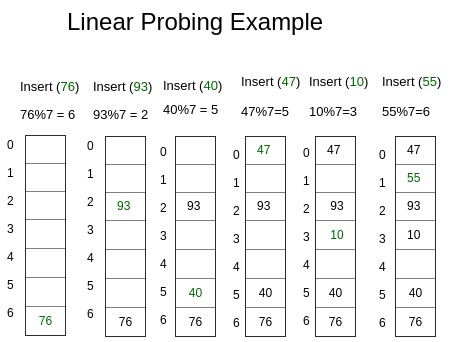
When looking at the previous number example in section 2, it is very unlikely that we would get a set of numbers that perfectly fit into the array. For example, let's say we want to store the 44 and 444 in the array table using the hash function,

Hash Function = Value % ArraySize

44 % 10 = 4 444 % 10 = 4

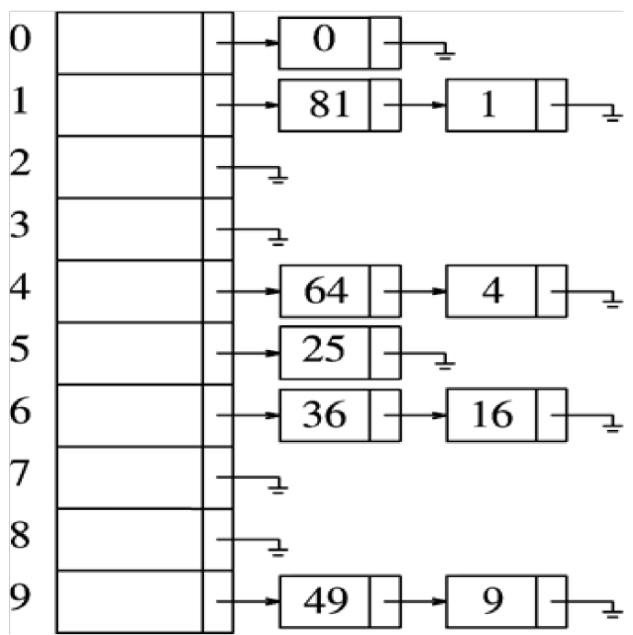
Our hash function produces the same location to store the both values. This is a collision, and is a major problem when hashing. Unfortunately, we normally have no control over the values we are given, so how well any particular hash function does depends on the distribution of the values within the allowable range. In some cases, incoming data are well distribute.

* + 1. **Linear Probing**One method for resolving collision looks into the hash table and tries to find another open slot to hold the item. A simple way to do this is to start at the original hash value position and then move in a sequential manner through the slots until we encounter the first slot that is empty. Note that we may need to go back to the first slot (circularly) to cover the entire hash table. This collision resolution process is referred to as open addressing in that it tries to find the next open slot or address in the hash table. By systematically visiting each slot one at a time, we are performing an open addressing technique called linear probing. In our previous example when we wanted to add 44 and 444 into the array, we would simply try to place 444 at index 5. If index 5 is already full, we then try index 6 and so on.



In the example above we notice collisions when we try to insert 47 and 55. In both cases we go to the next location and check if it is empty. We keep doing this until we find the next empty location. In the example 47 finds its location at index 0, and 55 finds its location at index 1.

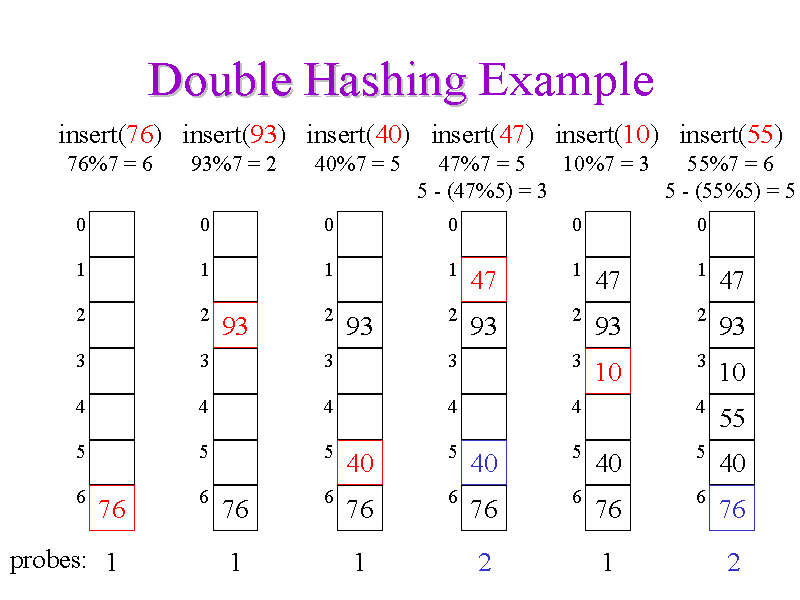
* + 1. **Chaining**An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient. In the previous example, when we try to insert 44 and 444 we run into a collision since both values want to be stored at index 4. Solving this issue using collision would simply create a linked list that connects 444 into the end of 44. Therefore, if we want to find 444 in memory, we simply find the location using the hash function, if the value is not at the first location, we sequentially go through the linked list until we find our value.



In the above example, we notice many instances where multiple collisions have occurred. We solved this by creating a linked list and attaching those values onto the ends of the previously located value at that index. In general, this should produce better results than linear probing. However, in extreme cases we may run into a set of values that all produce the same index when inserting into the hash function. In this case, we would simply be doing a sequential search of a large linked list, which is very inefficient. This would tell us that our hash function is not suitable for the information being inputted into the system;

**3.1.3 Double Hashing**

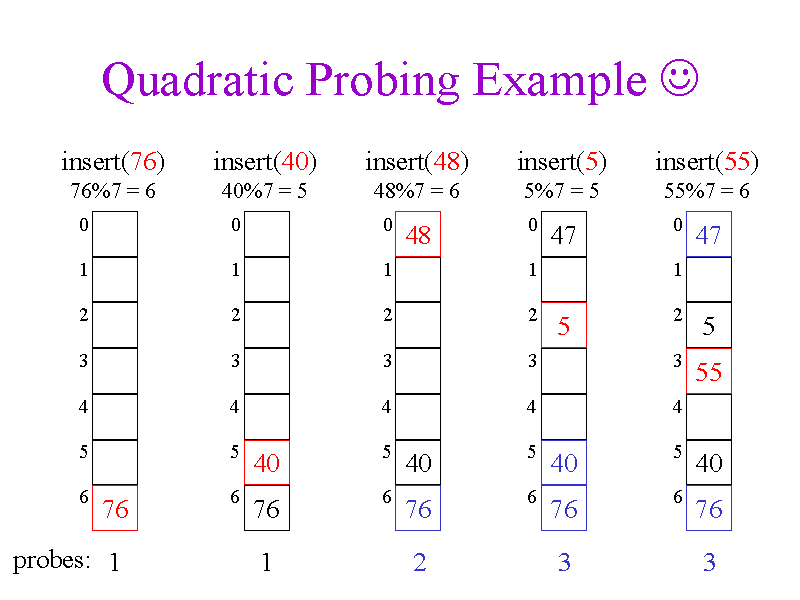
Like linear probing, double hashing uses one hash value as a starting point and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions.



In the example above, when we run into a collision we then use a separate hash function to find a new location for the value. For example, when trying to insert 47, we notice that the value 40 is in its spot. Therefore, we apply a second hash function to the value that takes the result from the first value. This produces a new location for the value to be inserted into.

**3.1.4 Quadratic probing**

A common problem that we see in hashing algorithms such as linear probing is primary clustering. This is when values are clustered around a specific part of the array. In linear probing, since the values check the very next spot, the values tend to not get distributed evenly across the array. Quadratic probing not only solves the issue with collision, but it also works to avoid primary clustering. Instead of continuously checking the very next position upon collision (linear probing), quadratic probing checks the next position, followed by the 2nd position, followed by the next 4th, next 9th, and so on.



in the example above, we run into a collision when trying to insert the value 5 into index 5. Therefore, we try the next position 6. We then try the 2nd position from index 5, which is 0, but it is also occupied. we then try the 4th position from index 5, which is 2, which is empty. Therefore, we place the value 5 at index 2. This helps to avoid clustering since we don't continuously check the next position in the array.

1. **Hashing in Databases**Hashing in Databases is useful in many ways. Databases are designed to store large amounts of data, and as such operations such as searching, adding, deleting, and updating data can become very slow. Hashing can solve this issue and can be implemented in a similar fashion to regular hashing. The indexes mark what is called a data bucket. This data bucket is essentially an implementation of the chaining collision solution. The hashing algorithm used would then point to a specific bucket, and each bucket acts as its own little database to be searched. The best
2. **Conclusion**

Accessing data in an efficient manner is an important part when using a database. With large amounts of data, we must find the most efficient way possible to store and access the information. Hashing is a convenient way to store and retrieve information that offers a time complexity better than all other retrieval algorithms.

**REFERENCES**

[1] Anon. Hashing Concepts. Retrieved May 7, 2019 from <https://www.tutorialcup.com/dbms/hashing-concepts.htm>

[2] Anon. 5.5. Hashing¶. Retrieved May 7, 2019 from http://interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/Hashing.html

[3] Anon. Hashing Tutorial. Retrieved May 7, 2019 from https://research.cs.vt.edu/AVresearch/hashing/introduction.php

[4] Hashing¶.” *5.5. Hashing - Problem Solving with Algorithms and Data Structures*, interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/Hashing.html.

[5] “5.5. Hashing¶.” 5.5. Hashing - Problem Solving with Algorithms and Data Structures, interactivepython.org/lpomz/courselib/static/pythonds/SortSearch/Hashing.html.

[6] *Quadratic Probing Example ?*, courses.cs.washington.edu/courses/cse326/00wi/handouts/lecture16/sld018.htm.