Two-Choice Hashing: The Power of Two Choice

Andy Mee (atm112@uakron.edu)

Abstract

Hashing is a form of organizing data in such a way that allows for searching for a data element in a quick fashing, such that it is as close to linear time as possible. With use of the modulus operator (%), we are able to find a position within an array for a particular data element to be placed. Some hashing tables allow elements to share the same position (typically via a list). Other tables do not. This does give us the issue of collision, where a data element's home position is already occupied by another data element. As a result, different hashing methods were created to deal with collision handling in their own unique ways. One of these methods is known as two-choice hashing. This method gives data two options when finding a spot in our hash table for a data item.

Introduction

The purpose of this study is to determine the efficiency of two-choice hashing as well as to gain a deeper understanding as to how it works. Comparing two-choice hashing to another method of hashing without the second choice, we will see how they compare to one another and determine if this truly reduces the chance of collisions to speed up both insertion and searching. We may also briefly discuss the space complexity of the differing hashing structures.

Discussion

There are generally two methods of dealing with collisions. One of these techniques is known as closed hashing or open addressing. On collision, an item being inserted into the hash table must find a different position. The more frequent there are collisions, the longer insertion

and deletions will take in your hash. There are numerous methods implemented to try and avoid as many collisions as possible. One of these methods is two-choice hashing.

Hypothesis: It is believed that with the inclusion of two choice hashing, the number of collisions may be reduced in half roughly. By constantly having two potential positions for an item to be placed in, insertion, searching, and deletion should be reduced as such.

An overview on hashing and two-choice hashing

A hashing table is a powerful structure used to reduce the time of searching to as close to constant time as possible. Most hashing tables are implemented simply with an underlying, sometimes dynamically allocated, array. Data within the hashing table is paired with a search key. These keys are most often numeric values used to determine where within the hash an item will be placed. This position is determined with a hashing function. The most common hashing function takes the numeric key value and uses the modulus operator on the size of the hash table to find what is considered the home position within the array, that being the initial position in the array after the hashing function is first used.

- **(Key % table size) = home position**

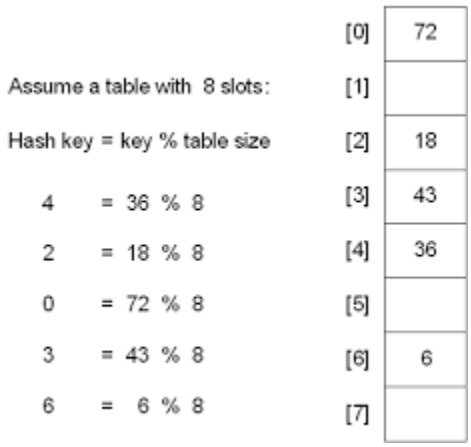

Image 1: a visualization of this basic hashing function

We will see later that not only are there other hashing functions, but it is also necessary at times to use differing hash functions. The use of the modulus operator is a common theme amongst different hashing functions. This is to ensure that keys yield positive numbers that are between zero and the size of the hashing table.

Hash tables usually support the same 3 basic functions: an insert function, a search function to find data mapped to a specific key, and a delete function to remove data holding a desired key. Complexity of these functions tend to vary amongst different hashing structures and will be discussed more in depth later. The goal for all of them is to keep searching as close to linear time as possible.

It is also important to note each key must be unique to the rest. No two keys may be the same. This being said, this does not mean that the position yielded from the hashing function will not be the same. In fact, this is an unfortunately common and nearly unavoidable occurrence. When two keys yield the same home position in the hash, this is known as a collision. Collisions provide a problem when it comes to searching efficiently for key values. When a key is already occupying the home position of another key, the question arises as to where this newer key should go. The Solution to this problem is known as collision resolution.

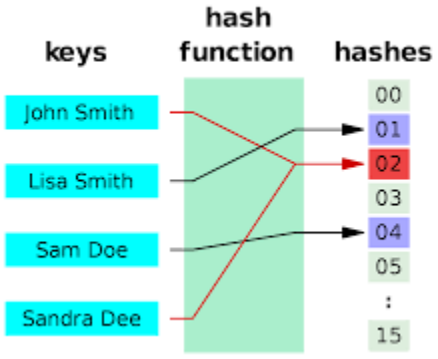

Image 2: example of hash function on key values leading to collision

As mentioned in the beginning of the discussion, there are two categories of collision resolution. Open hashing, or separate chaining, is a method of collision resolution that allows two keys to occupy the same space. This is typically done with the use of dynamic lists. These methods tend to be less space efficient and prove to be difficult when implemented on disk space. The other category, the one in which our structure falls in, is known as closed hashing, or open addressing (not to be confused with open hashing). This does not allow two keys to occupy the same space. Upon collision, the new key must find another position within the hash table. Often, this will lead to a sequence of other collisions until the new key is able to find an open position. This search for a free position is known as probe sequence.

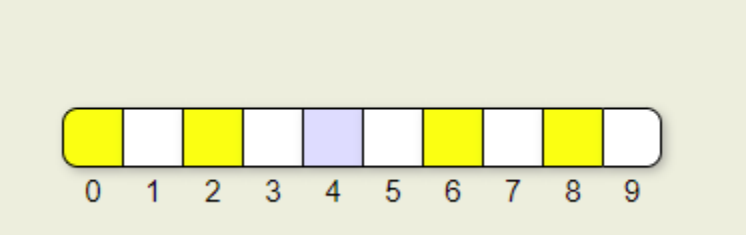

Image 3: home position at index 4. Probing distance of two, Every position highlighted in yellow will be a position checked in the probing sequence

A probe sequence will pick some distance away from its current position to check for an available position within the hash. Linear probing will pick a set distance away to check whether a position is available. If not, it will continue probing the same distance (wrapping around if it reaches the end of the table) until either a free position is reached or the probing sequence reaches back around to the home position. Programmers try to avoid situations like this, but in the event this happens, the hash is re-hashed, where a new, larger hash table is created and values within the old hash table are hashed into the new table.

Linear probing alone will typically lead to an issue known as primary clustering. Upon a collision, the resulting probing will lead values to be bunched close together. Values close by will

increase chances of future collisions. Increasing the space between probing or even having random probing positions would reduce primary clustering. However, despite the larger distance yielded by longer linear probing or quadratic probing, if the distance remains the same, we run into the problem of secondary clustering, where values will tend to bunch up around these predetermined distances. This will still increase the amount of collisions, albeit, not as bad.

We can further reduce this secondary collision with a method known as double hashing. This determines the probing distance based on the key being inserted with a separate hashing function. This will be the collision resolution method our hashing structure will use. The specific hashing function that will be used will be similar to the main hashing function, but the value will be one plus modulus operator on one less than the size of the hash table. This should prevent the probing distance from being either the size of the table or from being zero (avoiding going nowhere within the hash.

- **1 + (Key % (table size -1) = probing distance**

One more thing worth mentioning is that the size of the table can have an effect on the number of collisions you may encounter. The use of prime numbers will help generate unique values with the hashing function. For testing purposes, the varying sizes of hashing tables will be prime numbers, though there might be some experimentation otherwise.

While these are decent ways to prevent collisions, they are unfortunately very possible. One would ask if there is any way to further prevent these collisions. Upon collision, a key will likely follow almost the same path as other keys with the same home position. What if the key value had another path that led to an available slot sooner? If you were to use a different hashing function in this instance, would you find it sooner? This is the idea that gives us the basis for a two-choice hash.

The two choice hash works about what you would expect from the name. When initially hashing, not only do you use your initial hashing function, but a second hashing function is used to check for availability in another part spot in the hashing table. This hashing function must differ from the first one being used. In the case of our hash, the second hashing function will divide the key by the total number of spaces within the table. This value is then taken and has the modulo operator used on it with the size of the hash table again. The resulting value is the second choice for our initial hashing.

- **(Key / table size) % table size = second home position.**

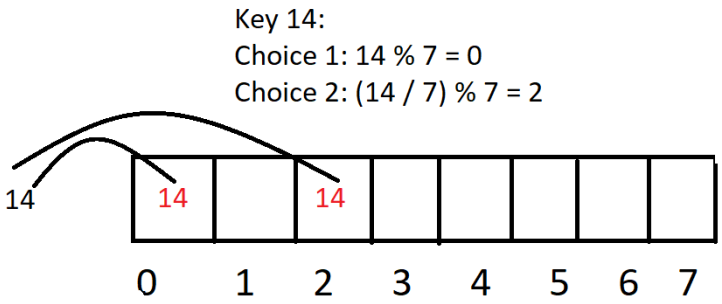

Image 4: A crude drawing (provided by me) of the initial hashing of a two choice hash

The item will then be inserted into whichever sloat is available over the two. If they both are, typically, the first location is favored. Upon collision in both home positions, the structure must initiate some sort of collision resolution. In some cases, bucket hashing is used, and the set with the least amount of entries is favored. As mentioned before, with our hash, we will be using double hashing. The two positions will then probe along the hash in parallel. Yielding the same probing distance may not prove to be effective. To further eliminate the chance of collision, and to prevent possibly searching in the same places as one another as well as skipping the same slots, the second choices probing distance will be one further than the

first's. We must also keep track of how far the second choice has probed. In the event that it reaches the same position as it started, it must be handled in the same way the first choice is. Two boolean values will be used to check whether or not both have fully wrapped around to the beginning.

In the event that no position could be found with any of the hashing methods, the table will be rehashed to fit double the size of the prior table. This is anticipated to almost never happen, but in the event that it does, this is the back up plan to ensure that the element still gets inserted within the hash.

Analysis of Two-Choice Hash

Now that we have an idea of how our two choice hash will work, we can now begin to predict just how it will perform. Being a hash table, there are three major operations that it must use. These operations are insertion, as described to exhaustion above, search and delete. These operations utilize other functions to help it carry out its work. There are 3 hashing functions used to generate values for the hash to use. All three of these hash functions are expected to have a complexity of Θ (1) since they need only to calculate and provide some numeric value. These functions will be known as hash1, hash2, and doubleValue. In the event that rehashing must occur, the size of the hash table is doubled and the entirety of the old hash is re-hashed over. This function should have an average complexity of $\Theta(n)$, n being the size of the old hash table since it will always step through, ensuring every element gets copied over. This means that this function should grow linearly with regards to the size of the table.

It is worth noting that our rehash function will use the insert function to rehash the values onto the table. While collisions may still occur, there should be no chance that the table will

rehash again. This does mean that the complexities of both the insert and rehash functions have an affect on one another.

The three main operations for our hash should be expected to all behave and grow the same. After all, the main goal of a hash structure is to maintain that searching is done as fast as possible. This is done by modeling the method of searching after the patterns followed by the insertion function. The searching operation will use the same exact hashing functions as insertion for predicting where a key value will be. The delete operation utilizes the search function to find a key value. If found, the delete function replaces the value at that location with our empty indicator, -1. Instead of rehashing in the event of not being able to find the key value, search will return the value -1, signifying that no position was found. Delete will do nothing if this value is returned, but if a position is returned, the value -1 will replace the key value that was formerly in the position returned by search.

These three functions are designed to perform as close to instant as possible. Upon either of the hashing functions finding no collision or incorrect value, these operations should always perform best case (big omega) of constant time (1). In fact, they are expected to almost always perform with complexity of $\Theta(1)$. Even in the event of an initial collision at both locations, the desired location is expected to be found not too far away. Because the hash is effectively always checking two locations at once, this is expected to cut the time of every major operation in half, as it is less likely that both positions being checked will not be available, providing a greater chance the operation performs as close to constant time as possible. The more entries within the hash table, however, the more likely there to be collisions in the positions we are checking. This would lead us to be more likely to reach our worst case complexity.

For search and delete, if the value is not found after initially hashing, the probing sequence will begin. The length of the probing process is determined by the choice with the lowest hashing distance to complete one full wrap around the hash table, assuming the desired value does not exist within the table. This distance could possibly be equivalent to linear probing, meaning it would require every spot within the hash table to be visited. At worst, this process grows with O(n) linear complexity. Insertion is expected to perform more or less the same, but must also account for the worst case scenario being that the rehash function gets called as well, which we know also has linear complexity. This should still grow linearly, but would mean that insert's worst case complexity would be closer to O(2n).

A well maintained hash table may try to take measures to avoid these worst case scenarios. A hash table is most likely to have significantly less collisions if the number of entries remains somewhere below the maximum size of the table. Our table does not automatically rehash after this threshold has been reached, but the user has the liberty to rehash at any point they like (so long as the new table size is larger than the prior).

In terms of space complexity, our hash table is expected to grow linearly with respect to the desired size of the hash table (Θ (n)). There will always be a constant 3 variables that will lie under the hash's total size, being the pointer to the dynamically allocated array. These all should be 4 bytes each (unless the machine uses otherwise). For our table and for research purposes, we will only store the integer key values within our array. Hashes in practice will store any other sort of data paired with these key values, and the size of the space allocated for the array will be fixed according to that.

The amount of overhead space will be equal to the number of slots not taken up by a key within our table. In theory, best case, every space could be taken up by some key and have no

overhead. As we discussed earlier, the closer we get to filling the allotted space of the table, the less efficient our table will become. We may also run the risk of a key value not being hashed to our table, resulting in a rehash. The way our hash table is set up doubles the size of the original hash table. If you had originally planned to fill out the entire table, at best, after a rehash, your overhead will now become double the original size of the hash table.

In an attempt to both keep operation speeds to a minimum, and to avoid risk of an unwanted rehashing, users should try and amortize by keeping the number of elements they wish to store anywhere below half the size of their table. If the number of entries becomes about half the size of the table, ideally, the table should be rehashed to maintain efficiency. This specific number of entries is known as the load factor.The tradeoff of having the space overhead be roughly half the size of the table will be to have speedy operations and avoid the risk of doubling the total amount of space used. For our hash, the insertion function does not automatically rehash once it hits the load factor. If a user were to use this hash, it would be up to them to rehash. For the sake of our experiment, you will see later that we will try to fill up to the maximum size of the hash table. Separate chaining has a different formula to find the load factor, but for open addressing hashes, this value is simply half of the hash table (N / 2).

Two Choice Hash Vs. Single choice hash

At this point, we should have a relatively good idea of how this two-choice hash will perform. It was stated earlier that a two-choice should make it less likely for a collision to occur. This is due to the concept of power of two choices. This is a concept of load balancing that allows random values to be more evenly distributed. This should make it so keys are less likely

to be bunched close to one another, which should ultimately keep insert, search, and delete times to a shorter time.

It is one thing to say this in theory, but how would we actually know whether two choice hashing leads to any sort of improvement in speed? In order to be sure, empirical analysis must be performed.

We could compare the performance of our two-choice hash with that of a hash that only checks one location at a time. In order to determine the effectiveness of the two-choice, it would be important that the opposing hash uses relatively the same collision resolution scheme that ours does, which is double hashing.

Fortunately for us, another experimenter has provided their own implementation of a hash that uses double hashing. This data structure will be tested against ours for determining the effectiveness of two-choice. It is worth noting that the other experimenter's code uses different logic for determining the second hash for the offset that is calculated. The difference in performance would probably be negligible. Considering that the other hash structure uses the same collision resolution, it is safe to assume that the complexity would be just about the same. It is anticipated that since it does not use double hashing, it will generally be slower with the operations.

Testing

Within our testing, we will be measuring the time it takes the operations of both structures to completely process a set of data. This will be done by the classic load and unload method. Over varying sizes of data sets and table sizes, values will be inserted into both hashes, measure the total time it takes to insert every value, then delete every value that was inserted within both tables and measure that time. These values will be randomly generated to prevent

any sort of pattern for insertion of keys. In order to keep track of which values are inserted, a separate array will be created, holding these values and traversing this list to call the delete function on these values.

We know that if we were to keep the number of elements inserted onto the table below half of the table's size, the number of collisions will be reduced. As weird as it may seem, we actually want to encounter some collisions. We wish to see if including a second choice will have any effect on collision handling. For our experiment's case, we will attempt to fill both hashes to their full capacity. With less available space, we wish to see if a two choice hash would be able to find an open position faster.

We know that having a prime value will decrease the chance of collision. Initially, the plan was to test against both prime and non prime numbers alike. However, the other experimenter's code provided ran into issues when some of the values were not prime. The range of table sizes will be from 101 - 150001, picking prime numbers in between. For each value of n, there will be three trials to account for the randomness of the numbers entered.

Results

After testing every problem size, we yield the following results. Time is measured in seconds on these charts.

TwoChoice	n = 101	503	1009	5003	10007	50021	100003	150001
Insert								
Trial 1	0	0	0	0	0	0.001	0.003	0.002
Trial 2	0	0.001	0	0.001	0	0.001	0.002	0.002
Trial 3	0	0	0.001	0	0.001	0.001	0.002	0.003
Search								
Trial 1	0	0	0	0.001	0.001	0	0.001	0.002
Trail 2	0	0	0	0	0	0.001	0.001	0.002
Trial 3	0	0	0	0	0	0.001	0.003	0.001
Delete								
Trial 1	0	0	0	0.008	0.076	6.295	45.478	122.685
Trial 2	0	0	0	0.008	0.067	5.319	46.324	121.251
Trial 3	0	0	0	0.008	0.067	6.287	45.621	88.212
Total								
Trial 1	0	0	0	0.009	0.077	6.296	45.479	122.689
Trial 2	0	0.001	0	0.009	0.067	5.321	46.327	121.255
Trial 3	0	0	0.001	0.008	0.068	6.289	45.626	88.216

Results from measuring two-choice hash

Double	n = 101	503	1009	5003	10007	50021	100003	150001
Insert								
Trial 1	0	0	0	0.001	0.004	0.488	3.673	9.402
Trial 2	0	0	0	0	0.004	0.42	3.737	9.207
Trial 3	0	0.001	0	0.001	0.005	0.469	3.617	8.572
Search								
Trial 1	0	0	0	0.002	0.01	0.559	2.956	5.87
Trail 2	0	0	0	0.003	0.01	0.484	2.98	5.832
Trial 3	0	0	0	0.001	0.008	0.525	3.03	5.428
Delete								
Trial 1	0	0	0	0.006	0.021	3.021	23.338	61.498
Trial 2	0	0	0	0.003	0.021	2.722	23.457	59.552
Trial 3	0	0	0	0.003	0.025	3.049	23.617	55.738
Total								
Trial 1	0	0	0	0.009	0.035	4.068	29.967	76.77
Trial 2	0	0	0	0.006	0.035	3.626	30.174	74.591
Trial 3	0	0.001	0	0.005	0.038	4.043	30.264	69.738

Results from measuring double hash

As we can see from the following results, our hash table was able to produce a faster insert and search function. As a matter of fact, seemingly, no matter the size, insert and search performed in almost instant time with our hash structure, as opposed to the other structure which seemed to gradually take more time as the size of the problem grew by a couple of seconds. It is safe to say that this evidence supports that we can expect our hash to perform inserts and searches to as close to constant time as possible. As compared to the other hash, we can also reasonably draw that two-choice helped us achieve this.

One thing that was not particularly anticipated was how long it took to delete (or attempt to delete) every value within the table. It would seem that the other experimenter's code yields a faster delete function, which appears to be roughly twice as fast. Thinking about what might cause the delete function to take so long to complete a task load, there are a couple of factors that may lead to the case. While not very likely, there is a chance for multiple duplicate numbers to be generated within our set of data. Especially with how imperfectly random the rand() function can sometimes be. Our insert function does not allow a duplicate key to be inserted if it is found within the hash whilst trying to insert. That would mean that the value would only be deleted once. Any time after trying to delete the same value, the function must, at worst, traverse the whole table. Having to do this multiple times on a sufficiently large data set can prove to be taxing. Assuming it tries to delete all values not present in the set of data, this delete algorithm is worst case O(n^2). It is hard to say why the other code runs the delete operations faster than ours.

This lackluster performance of the delete algorithm may be a good example of how the rest of the functions would perform. The Insert and search algorithms give us evidence for how they perform on average, and the delete can be interpreted as relatively close to worst case (or rather a less desirable case.

Aside from our delete operation, this data gives us evidence that a two-choice hash system would improve a hash table's performance. A few seconds may not feel like a whole lot when it comes to the time the other structure took, but if the size of the problem grew even larger, insertion and searching would grow faster.

In a separate testing function, the total number of bytes used up by a given hashing table can be calculated. It adds the constant 12 bytes there will be for the pointer and data members

and adds it to the size of the rest of the table (int size * n). It will also calculate the amount of overhead within your hash table if any.

Conclusion

We have spent a good while gaining an understanding of what a hashing table is, how it works. We looked at different ways to prevent collisions from happening to try to keep insertion and searching at a minimum time cost. Most importantly, we have gained a deeper understanding of what two-choice hashing is. Through our analysis and empirical research, we were able to see the power of two choices in action. Based on the evidence we have provided to us, we have seen that providing a second choice within our search for a position for our key, we are able to find said position much faster and as close to constant time as possible.

Whether this is one of the only methods on an open addressed hash that gives us near constant time is hard to say. There may be other factors as to why this hash could insert and search faster than the other once choice. Based on our results, one could say that a tradeoff of this implementation would be that deletion can be costly if you are not careful.

We have seen how fast the two choice hash can perform with no spacial overhead, but if you are willing to deal with half space taken up as overhead, you can expect an even more efficient hash. Though, if you are willing to take the risk of slower operation, we have also seen that this hash is capable of operating at maximum capacity.

We are aware that an ideal hash is capable of searching and inserting at near constant time. A hash table that provides multiple ways to handle collisions is a great way to achieve this. It is entirely possible that a two-choice hash with double hashing collision resolution is potentially one of these ideal hashing structures.

Sources

1. 2-choice hashing wikipedia

https://en.wikipedia.org/wiki/2-choice_hashing

2. 2-choice hashing programming guide

https://programming.guide/2-choice-hashing.html

3. ODSA 10.07 Improved Collision Resolution

https://canvas.instructure.com/courses/4088313/assignments/27651721

4. Hashing Part 3 - Open addresing_default

Hashing Part 3 - open addressing_default - Data Structures 010 (uakron.edu)

5. Brandon Kulmala's implementation of a Double Hash -bjk74@uakron.edu

Images:

Image 1: http://faculty.cs.niu.edu/~freedman/340/340notes/340hash.htm

Image 2: https://en.wikipedia.org/wiki/Hash_collision

Image 3:

https://canvas.instructure.com/courses/4088313/assignments/27651721?module_item_id=60214323