Optimal Algorithms
Hashing
Peter M. Maurer
Why Hashing is $\Theta(n)$.

As in binary search, hashing assumes that keys are stored in an array which is indexed by an integer. However, hashing attempts to bypass the normal searching mechanism by translating the search key into an integer and using that integer to access the item directly. To be specific, assume that the underlying array has an index range from 1 to $M$, and that the hashing function is $H$. Given a string $s$, $H(s)$ is an integer in the range 1..$M$. The hashing function is $\Theta(1)$, and the array access is also $\Theta(1)$, so at first glance it appears that hashing must also be $\Theta(1)$. However, this is not the case. Unfortunately the function $H(s)$ is not one-to-one. In other words, there are two keys $s_1$ and $s_2$ such that $s_1 \neq s_2$, but $H(s_1) = H(s_2)$. This is known as a collision. When collisions occur between keys, one of the keys must be located somewhere else in the table other than the position given by $H(s)$. There are several methods for determining the position of the second key when it is initially placed in the table. One is to do a linear search for the next available slot and place the key there. Another method (which is generally considered to be best) is to convert the array into a table of pointers, and store all colliding keys as a linked list. The table contains a pointer to the head of each linked list. All of the most popular methods for resolving collisions are based on some form of linear search. It is this linear search that makes hashing $\Theta(n)$.

To see why this is so, first consider the worst case. Obviously, the more collisions that occur, the worse the running time will be, so let’s assume that every key causes a collision. In other words, the hashing function, $H(s)$, returns the same value for every string $s$. For table lookups, the worst case is still the case where the key does not appear in the table. Under these conditions, every table-lookup will require a linear search of $n$ items, which requires $\Theta(n)$ time.

Although it is theoretically possible to create a worst-case set of strings for any statically defined hashing function, most hashing functions are designed to make the worst case extremely unlikely. To gain some appreciation of how well hashing could be expected to perform on the average, let’s assume that $H(s)$ gives an evenly distributed set of indices. Because binary search is able to handle an arbitrary number of keys, it is only fair to assume that our hashing algorithm can do the same. This eliminates those classes of hashing algorithms that store all keys in a fixed sized table, but strictly speaking these algorithms are not “correct” because they fail when the size of the input exceeds the size of the table. The linked-list type of hash table does not suffer from this deficiency. We will assume that the hash table is of fixed size, and that the number of keys is much larger than the maximum table index, $M$. Once $M$ keys have been added to the table, a collision must occur on the next key (unless, of course, one has already occurred). Under the assumption that indices are evenly distributed, the number of keys that hash to any
particular index, $i$, is equal to $\frac{n}{M}$. During a table lookup, the average case would require a linear search of $\frac{n}{2M}$ keys, which is again $\theta(n)$.

From the above analysis, it would appear that hashing is not as efficient as binary search. In fact, if we are dealing with millions or billions of keys, this will probably be the case. However it is important to note that although hashing is linear that it is $M$ times faster than a normal linear search, and $M$ is generally a reasonably large number. In the asymptotic notation, the constant $1/M$ is “absorbed” into the constant of proportionality. One of the motivations for using asymptotic notation is that constants of proportionality tend to be fairly close to one another, and hence can be safely ignored. This is not the case for hashing. The constant of proportionality is extremely small compared to that for linear search. Hashing is typically used in cases where the number of entries in the table will not be significantly larger than $M$. If we assume that this is the case, and that the hashing function $H(s)$ is well designed, hashing will exhibit behavior very close to $\theta(1)$, and will perform significantly faster than binary search. Another important point is the amount of time required to create the table. Hashing is usually used in an environment where the table must be created dynamically one key at a time, and where table look-ups are interspersed with table insertions. If a sorted array were used to create the table, each insertion would require approximately $\theta(n)$ operations to insert a key in its proper position. This would give $\theta(n^2)$ performance for inserting $n$ keys. There are methods for creating binary search trees on the fly, but rebalancing the trees requires a significant amount of overhead. (This is a whole topic unto itself, and I don’t want to digress too far here.) Again, under the two assumptions, that $H(s)$ is well designed, and that the number of keys, $n$, is not significantly larger than the size of the table $M$, the amount of time required to insert a key into the hash table will be approximately $\theta(1)$. This gives approximately $\theta(n)$ performance for inserting $n$ keys. However, if the number of keys is significantly larger than the size of the table, or if the probability of collisions is high, then the time bound for inserting $n$ keys will be $\theta(n^2)$. Strictly speaking, the amount of time required to insert a new entry into a linked-list hash table is $\theta(1)$, if you don’t care about inserting duplicates. The worst-case $\theta(n)$ performance is based on the assumption that a preliminary table look-up will be required to avoid inserting duplicates.

**Improving Worst-Case Hashing.**

The worst-case analysis of hashing was based on the assumption that a linear search would be required to resolve collisions. This assumption causes a factor of $n$ to appear in all time bounds. If the performance of collision resolution could be improved, it should be possible to improve the worst-case time bound. Suppose that instead of a linear search, a binary search was used to resolve collisions. In this case, the worst case-time-bound for table lookups could be reduced to $\frac{\lg n}{2M}$, which is $\theta(\lg n)$.

Another intriguing possibility is to use hashing itself to resolve collisions. Assume that if a collision occurs using $H(s)$, that a second function $H_1(s)$ will be used to resolve the collision. To keep things simple, let’s assume that second-level hash tables are used to store the keys hashed by $H_1(s)$. This would require one second-level table for each
entry in the first table. Since it is possible for \( H_1(s) \) to produce collisions, some assumptions must be made about collision resolution in the second-level hash tables. If we assume that a standard linear search is used to resolve second level collisions, worst-case behavior would occur when every key caused a collision on both levels, giving a worst case performance of \( \theta(n) \). Assuming evenly distributed keys at both levels, and an extremely large value for \( n \), the number of comparisons that would be required is \( \frac{n}{2MN} \), where \( M \) is the size of the first-level table, and \( N \) is the size of the second-level tables. If we assume all tables are the same size, the result is \( \frac{n}{2M^2} \).

As yet, we have not obtained any improvement in the asymptotic time bound, although we have significantly reduced the constant of proportionality. Let’s carry this idea to its natural conclusion. Assume that we have an infinite sequence of hashing functions \( H_0(s, H_1(s, ..., H_i(s)) ..., \) where \( H_0(s) \) will be used for the level-one tables, \( H_1(s) \) will be used for the level-two tables, and so forth. We will assume that these functions are applied successively until no collision occurs. Immediately we have a problem with our worst-case assumption of getting a collision on every key, because this would imply an infinite recursion. If we were clever enough to design such an algorithm, then we would certainly be clever enough to guarantee that this would not occur. Let us assume, instead, that we get evenly distributed keys at ever level, and try to determine just how well we could do under these circumstances. For simplicity, assume that all tables are of size \( M \), and that the number of keys, \( n \), is much larger than \( M \). As we add levels to the hash table, the amount of space for storing keys (or pointers to linked-lists of keys) becomes larger. \( M \) keys can be stored at level 1, \( M^2 \) keys can be stored at level 2, \( M^3 \) keys at level 3, and so forth. At some level \( k \) we will have enough space to store all keys, and at this level, linear searches will no longer be required. We have reached this level when \( M^k \geq n \). Note that \( k = \lceil \log_M n \rceil \). Since the operations at each level require constant time, this gives us a time bound of \( \theta(\lg n) \).

**Perfect Hashing**

As pointed out above, it is the resolution of collisions that prevents hashing from being \( \theta(1) \). Suppose it were possible to eliminate collisions entirely. This would require a hash function that was somehow adaptable to its environment, at least to the degree that it could handle a table of arbitrary size. Let us assume further, that we have a function \( H(s) \) that is one-to-one. In other words, if \( H(s_1) = H(s_2) \) then \( s_1 = s_2 \). This isn’t really a reasonable assumption, for the following reasons. As pointed out above, it is necessary to assume that the size of the table can be arbitrarily large which implies that the length of the strings must be arbitrarily large as well, since there are only a finite number of characters that could occur in any string. However, let’s assume that the length of the strings is limited to some “reasonable” value, say 8. Let’s also assume that the strings are limited to certain “reasonable” characters, say upper and lower case letters. Including the null string, this gives us \( 52^8 + 52^7 + 52^6 + 52^5 + 52^4 + 52^3 + 52^2 + 52^1 + 52^0 \) different strings that could be used as an argument to \( H \). To simplify matters, let’s assume that all strings are exactly 8 characters long, which gives us a total of \( 52^8 \) different strings that could be used
as an input to $H$. Now if we assume that our table is indexed by an unsigned 32-bit integer, we would have a total of $2^{32}$ table entries. Observe that $2^{32} = 16^8$. Which is bigger $52^8$ or $16^8$? And just how big is a table with $2^{32}$ entries anyway? If we do nothing more than store the string, this will require $2^{32} \cdot 2^3 = 2^{35}$ bytes. To put this in perspective $2^{35}$ bytes is 32 gigabytes.

Nevertheless, let us pursue this dream of “perfect hashing” a bit further. If $H(s)$ truly was a one-to-one function, then collisions would be impossible. Assuming that $H(s)$ runs in $\Theta(1)$ time, hashing would indeed be an $\Theta(1)$ algorithm. However, the assumption that $H(s)$ is an $\Theta(1)$ process is not realistic either. Remember that it is necessary for our algorithm to be able to handle an arbitrarily large number of strings. If $H(s)$ is truly a perfect hashing function, it must look at every character of $s$. To see why this is so, imagine that there is a character in $s$ at position $i$ that is not examined by $H$. Any string that differed from $s$ only in position $i$ would hash to the same index as $s$, contradicting the assumption that $H$ was one-to-one. Since there are only a finite number of characters that can appear in a string, the necessity of handling an arbitrarily large number of strings demands that we also be able to handle arbitrarily long strings. Since $H$ is required to examine every character of every string it processes, it must take more time for long strings than for short strings. The minimum permissible length for the maximum length string in a set of $n$ strings is proportional to $\log n$. For example, if we have a set of 258 strings, we know that at least one string must have a length greater than 2. There is one string of length zero, and 256 strings of length 1, for a total of 257. The 258th string must have a length of at least 2. Continuing in this fashion, any set of strings of size $n = 1 + \sum_{i=0}^{256} 256^i$ must have a string of length at least $k+1$. Furthermore, 

$$\left\lceil \log_{256} n \right\rceil = \left\lceil \log_{256} \left( 1 + \sum_{i=0}^{256} 256^i \right) \right\rceil = k + 1.$$ 

Since all logarithms are proportional to one another, $k+1$ is proportional to $\log n$. Thus, even a “perfect” hashing algorithm must be at least of order $\Theta(\log n)$. 