**CHAPTER 2**

**THEORY BACKGROUND**

Hashing is a refinement of the simple idea that searching can be made more efficient if one can quickly narrow one's search by locating a subset of items to search. Each item is characterized by a key, a part of the item which uniquely identifies it; such as an employee-number. Hashing creates a subset of items to search by computing from the key a characteristic value, called the hash, and then groups items with the same hash; the group is called a bin. There are several methods for grouping items; the most common makes a linked list of the items with the same hash. To retrieve an item the program computes the hash from the key that you want to search for, and it then searches for the item in the bin associated with the hash. If item keys are the names of people and one uses the first character of the last name as the hash, then a hash search will work well for last names starting with X, because not many last names start with X. But the search works poorly for finding Mr. John Smith, because there are many names that begin with S. A clean hash function, when given a series of keys, gives back a reasonable approximation of a uniform random distribution of hashes. This means that all bins will probably be about the same size, so that, on the average, searching will take about the same amount of time to find a record in any bin. Hash searching is the most powerful method of retrieving data known. It works well even with rather poor hash functions, that is those that do not distribute the input cleanly. It is so powerful that most automated retrieval mechanisms today are based on hashing.

**2.1 History of Hashing**

The idea of hashing started in the earliest days of computing. The first true electronic computers began to run in 1949 and 1950. A proposal for hash search was described by Hans Peter Luhn in an IBM technical memorandum in 1953. What he wanted was a function that would deliberately abuse keys producing practically the equivalent of the mathematical concept of uniformly distributed random variables.

Luhn's goal for producing uniform randoms is one approach, but often in computer science the goal of getting a completely even distribution has been substituted for it. The change in goal is significant and leads to two completely different lines of research both of which are commonly called `hashing'. Creating even distributions can only be done by considering the structure of the keys, so the method can never be `general'. However creating random uniform distributions can be done without respect to the structure of the key, and so it can be provided as a standard part of a language library. The word `hashing', as used in this paper, refers only to the goal of producing a uniform random distribution of a key set.

In some sense, it is commonly believed that it is impossible to create a general hash function that would work for any set of keys. This is because all hash function have a worst case, in which the hash function fails to hash at all, and all records fall into a single bin. But it is silly to define a general hash function in a way that makes it not a hash function, which is what this argument effectively does.

It is better to define a general hash function as one which distributes hash values from any independent domain in a way that approximates a random uniform variable. Then, it is only the probability of a bad hash that comes into question. Gonnet studied this and came to the conclusion that the number of items in the biggest bin grew as inverse factorial of the number of items hashed. This growth rate of the worst case of the average table is so slow above thirty items that essentially it never gets any worse. So a general hash function which approximates a random uniform variable for all inputs will have a very low probability of producing a bad search table.

It is also impossible to create a general hash function that works better that uniform random for any arbitrary set of keys. However if one knows the keys in advance, one can tailor hash functions to perform better than a random uniform table. For a language library function which, in general, can't know keys in advance, a random uniform hash is the best that can be done.

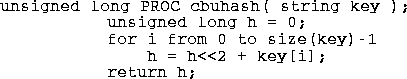
Much work has been done on specialized hash functions trying to minimize the distribution for give key sets or for keys derived from a given probability distribution. As a consequence little serious work has been done on general hash functions themselves. Recently, however, there have been a few papers which provide notable exceptions to the general. From these three papers have come a growing recognition of principles which govern the evaluation of and requirements for general hash functions [1].

2.1.1 Library Hash Functions

In 1989, Bruce McKenzie and his co-workers at the University of Canterbury, Christchurch, New Zealand, developed several methods for evaluating hash functions and by studying and measuring many hash functions they empirically discovered odd behavioral properties of most of the commonly used hash functions. The performance problems exposed by this hash function testing showed that the average cost of searching can range over several orders of magnitude using the same hash function varying only the output range. This seemed to be directly attributable to the sensitivity of the result on the modulus applied at the end of the hashing operation. This property makes most functions unacceptable as a library hash function because they need to be tested for the specific range over which output is expected.

2.1.1.1 CBU Hash

In the McKenzie paper, the authors conclude that for hashing program identifiers, the following linear hash function, presented here in its simplest form, is a good hash function.



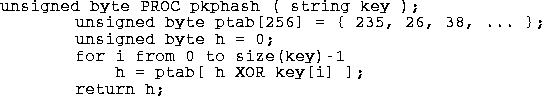
**Figure 2.1 Sample Code of CBU Hash**

This function is named CBU hash for CanterBury University, in Christchurch, New Zealand, where it was developed. It works by using random variations in low-order bits of the key characters to create noise which masks the high order bits of the characters.  To finish the mixing the resulting hash value must be divided by an odd number and the remainder used as the operational hash. This means that one must always have a odd number of bins in the hash table. This hash function works particularly badly for hash tables whose size is an integral power of two.

This CBU hash function has the advantage that it is fast to compute, it seems to work well for an ASCII character string domain, and it has a broad range for which it works. This algorithm and variations have been used in modern programs, for example the Revision Control System uses the same algorithm but with an exclusive-or replacing the plus in the computation of the hash.

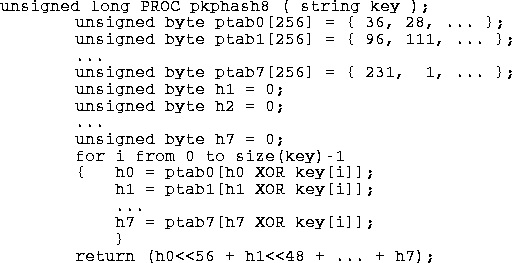
2.1.1.2 PKP Hash

In 1990, Peter K. Pearson published a random walk hash algorithm which produced integers from 0 to 255.  He claimed this algorithm worked well for text strings (with 8-bit bytes). Pearson uses a compile time static table, ptab, of unsigned byte values [0-255] that have been randomly permuted which he uses to introduce noise into the key. The Pearson algorithm is:



**Figure 2.2 Sample Code of PKP Hash**

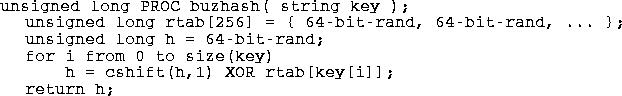
A defect in this hash algorithm is that for hash ranges greater than 0-255 PKP hash becomes more costly. In fact for a 64-bit range it will cost roughly 7 times more to compute than for a range of 0-255.  With the inner loop expanded for efficiency, the algorithm is as follows:



**Figure 2.3 Simple Code of PKP Hash with Inner Loop**

2.1.1.3 BUZ Hash

At the same time Pearson's hash function was published. By reading his algorithm, it was sure that a new algorithm could be designed that would compute a large hash value more cheaply.  At the same time, it is suggested to look at the McKenzie article in Software Practice and Experience. By using McKenzie's techniques for measurement, it is found that a hash algorithm which requires the use of a random table 256 words large, where a word is defined as the maximum data size that an XOR function can be applied to easily in the programming language. Depending on the programming language and the machine this word size can assume different values: using C on a modern PC this is 32 bits, in PL/I on an IBM 360 this is 2048 bits, and in Java on a byte-code machine it is 64 bits.  If assume unsigned long is 64-bits, and strings have 8-bit bytes then a library hash function suitable for Java might look like the following:

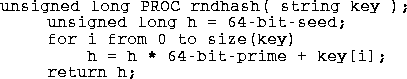


**Figure 2.4 Simple Code of BUZ Hash**

The cshift is a circular shift (or rotation), that can be applied in either direction. The rtab table is a compile time constant array that should have the property that in every bit position vertically in the table, there should be exactly 128 zeros and 128 ones. This makes the probability of change in all the hash bit positions ½, giving a uniform random distribution. The Java BUZ hash is designed to work for keys shorter than 65 characters; this does not limit programs much because rarely are keys encountered that are greater than 64 characters long.

2.1.1.4 Other Hash Functions

A hash function based on a pseudo-random number generator has been around for several years. It is not able to trace its source to a publication.  It works on a linear congruential random number generator, but any generator that provides a satisfactory random distribution modified to fold in the bytes of the key should also work.  Here is the code for a 64-bit version of this function; here it makes no difference what byte size comes from the string.



**Figure 2.5 Simple Code of Other Hash Functions**

Note that this function tends to be a bit slower than PKP hash and BUZ hash since it requires one multiply for every byte of key. To really use this function a good value for the multiplier must be found and the resulting hash function tested. This has not been done for a 64-bit word. The academic cryptographic community has produced several secure hash functions that distribute keys randomly, but for general retrieval use cryptographic hash functions tend to work too slowly to satisfy most users [1].

**2.2 Hash Table**

A hash table (also called a map, a lookup table, an associative array, or a dictionary) is a container that allows direct access by any index type. It works like an array or vector except that the index variable need not be an integer. A good analogy is a dictionary: the index variable is the word being looked up, and the last indexing element is its dictionary definition. A table is a sequence of pairs. The first component of the pair is called the key. It serves as the index into the table, generalizing the subscript integer used in arrays. The second component is called the value of its key component. It contains the information being looked up. In the dictionary example, the key is the word being looked up, and the value is that word’s definition (and everything else listed for that word).

A table is also called a map because the keys being mapped into their values can be thought like a mathematical function: f(key) = value. Tables are also called an associative arrays because they can be implemented using two parallel arrays: the keys in one array and the values in the other. A general hash table looks like in Figure 2.1: an array of Objects indexed by their hash values. This requires that the range of the hush function match the range of index values in the array. This is almost always managed by simply using the remainder operator modulo the size of the array [2].

|  |  |
| --- | --- |
| 0 |  |
| 1 |  |
| 2 | Ohr |
| 3 | Rad |
| 4 | Uhr |
| 5 | Hut |
| 6 | Tag |
| 7 |  |
| 8 | Tor |

**Figure 2.6 Hash Table**

**2.2.1 Conflicting Requirements of Hash Table**

In general, a hash function needs to satisfy somewhat conflicting requirements:

* A hash table’s size should not be excessively large compared to the number of keys, but it should be sufficient to not jeoparsdize the implementation’s time efficiency.
* A hash function needs to distribute keys among the cells of the hash table as evenly as possible. (This requirement makes it desirable, for most applications, to have a hash function dependent on all bits of a key, not just some of them).
* A hash function has to be easy to compute.

**2.3 Collision in Hashing**

If a hash table’s size m is chosen to be smaller than the number of keys n, it will occur collisions, a phenomenon of two (or more) keys being hashed into the same cell of the hash table. Collision in hashing can be seen in Figure 2.2. But collisions should be expected even if m is considerably larger than n. In face, in the worst case, all the keys could be hashed to the same cell of the hash table. Fortunately, with an appropriately chosen hash table size and a good hash function, this situation happens very rarely. Still, every hashing scheme must have a collision resolution mechanism. This mechanism is different in the two principal versions of hashing : open hashing (also called separate chaining) and closed hashing (also called open addressing) [3].

Ki Kj

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  |  |  |  |
| 0 | **. . .** | b | **. . .** | m-1 |

**Figure 2.7 Collision of Two Keys in Hashing : h(Ki) = h(Kj)**

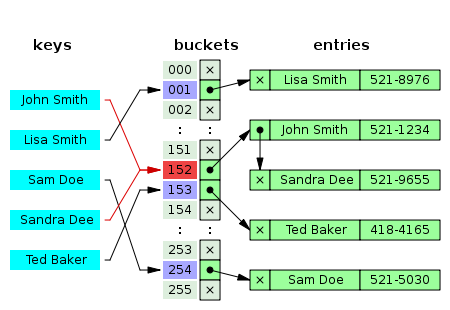
* 1. **Collision Handling Techniques**

Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Followings are the ways to handle collisions:

* Separate Chaining (also called open hashing): The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
* Open Addressing (also called closed hashing): In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, table slots is examined one by one until the desired element is found or it is clear that the element is not in the table.
  + 1. Separate Chaining

Open Hashing is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list. In this technique when a data needs to be searched, it might become necessary (worst case)  to traverse all the nodes in the linked list to retrieve the data. Note that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

In the method known as separate chaining, each bucket is independent, and has some sort of [list](https://en.wikipedia.org/wiki/List_%28abstract_data_type%29) of entries with the same index. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation. In a good hash table, each bucket has zero or one entries, and sometimes two or three, but rarely more than that. Therefore, structures that are efficient in time and space for these cases are preferred. Structures that are efficient for a fairly large number of entries per bucket are not needed or desirable. If these cases happen often, the hashing is not working well, and this needs to be fixed.



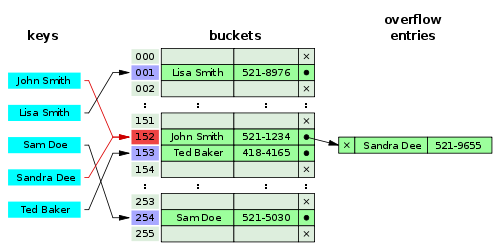
**Figure 2.8 Hash Collision Resolved by Separate Chaining**

#### 2.4.1.1 Separate Chaining with Linked Lists

Chained hash tables with [linked lists](https://en.wikipedia.org/wiki/Linked_list) are popular because they require only basic data structures with simple algorithms, and can use simple hash functions that are unsuitable for other methods. The cost of a table operation is that of scanning the entries of the selected bucket for the desired key. If the distribution of keys is [sufficiently uniform](https://en.wikipedia.org/wiki/SUHA), the average cost of a look up depends only on the average number of keys per bucket that is, it is roughly proportional to the load factor. For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots. For example, a chained hash table with 1000 slots and 10,000 stored keys (load factor 10) is five to ten times slower than a 10,000-slot table (load factor 1); but still 1000 times faster than a plain sequential list. For separate-chaining, the worst-case scenario is when all entries are inserted into the same bucket, in which case the hash table is ineffective and the cost is that of searching the bucket data structure. If the latter is a linear list, the look up procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the table.

The bucket chains are often searched sequentially using the order the entries were added to the bucket. If the load factor is large and some keys are more likely to come up than others, then rearranging the chain with a [move-to-front heuristic](https://en.wikipedia.org/wiki/Move-to-front_heuristic) may be effective. More sophisticated data structures, such as balanced search trees, are worth considering only if the load factor is large (about 10 or more), or if the hash distribution is likely to be very non-uniform, or if one must guarantee good performance even in a worst-case scenario. However, using a larger table and/or a better hash function may be even more effective in those cases. Chained hash tables also inherit the disadvantages of linked lists. When storing small keys and values, the space overhead of the next pointer in each entry record can be significant. An additional disadvantage is that traversing a linked list has poor [cache performance](https://en.wikipedia.org/wiki/Locality_of_reference), making the processor cache ineffective.

2.4.1.2 Separate Chaining with List Head Cells

Some chaining implementations store the first record of each chain in the slot array itself.The number of pointer traversals is decreased by one for most cases. The purpose is to increase cache efficiency of hash table access. The disadvantage is that an empty bucket takes the same space as a bucket with one entry. To save space, such hash tables often have about as many slots as stored entries, meaning that many slots have two or more entries.[](https://en.wikipedia.org/wiki/File:Hash_table_5_0_1_1_1_1_0_LL.svg)

**Figure 2.9 Hash Collision by Separate Chaining with Head Records in the Bucket Array**

#### 2.4.1.3 Separate Chaining with Other Structures

Instead of a list, one can use any other data structure that supports the required operations. For example, by using a [self-balancing tree](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree), the theoretical worst-case time of common hash table operations (insertion, deletion, lookup) can be brought down to [O(log n)](https://en.wikipedia.org/wiki/Big_O_notation) rather than O(n). However, this approach is only worth the trouble and extra memory cost if long delays must be avoided at all costs (e.g., in a real-time application), or if one must guard against many entries hashed to the same slot (e.g., if one expects extremely non-uniform distributions, or in the case of web sites or other publicly accessible services, which are vulnerable to malicious key distributions in requests).

The variant called [array hash table](https://en.wikipedia.org/w/index.php?title=Array_hash_table&action=edit&redlink=1) uses a [dynamic array](https://en.wikipedia.org/wiki/Dynamic_array) to store all the entries that hash to the same slot. Each newly inserted entry gets appended to the end of the dynamic array that is assigned to the slot. The dynamic array is resized in an exact-fit manner, meaning it is grown only by as many bytes as needed. Alternative techniques such as growing the array by block sizes or pages were found to improve insertion performance, but at a cost in space. This variation makes more efficient use of [CPU caching](https://en.wikipedia.org/wiki/CPU_cache) and the [translation lookaside buffer](https://en.wikipedia.org/wiki/Translation_lookaside_buffer) (TLB), because slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space. Despite frequent array resizing, space overheads incurred by the operating system such as memory fragmentation were found to be small.

An elaboration on this approach is the so-called [dynamic perfect hashing](https://en.wikipedia.org/wiki/Dynamic_perfect_hashing),where a bucket that contains k entries is organized as a perfect hash table with k2 slots. While it uses more memory (n2 slots for n entries, in the worst case and n\*k slots in the average case), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion. It is also possible to use a [fusion tree](https://en.wikipedia.org/wiki/Fusion_tree) for each bucket, achieving constant time for all operations with high probability [4].

2.4.1.4 Two Most Important Chaining Techniques

The basic idea of separate chaining techniques is to utilize additional data structures, e.g. linked lists, for collision resolution. Each hash table key has its own list for collision resolution. The advantage of chaining techniques relies in the ability to easily resolve conflicts and the permanent possibility to insert new keys without resizing the hash table. Hence, applying separate chaining techniques may result in hash table load factors beyond 1, which is impossible for open hashing techniques. On the other hand chaining techniques may result in drastic performance penalties when the hash table degenerates, e.g. to a linked list. Such a degeneration may also increase the cache miss rates dramatically. The two most important chaining techniques are:

* Move to front
* Exact fit

2.4.1.4.1 Move to Front

The move to front technique consists of the hash table and a linked list for each key to handle the collisions. Whenever a conflict in a hash table key occurs, the new key is appended to the list’s end. The core idea of this technique is whenever a key is accessed it is moved to the beginning of the linked list. In case of searching a key in the hash table’s conflict resolution list, the same parts of memory are accessed repeatedly. The move to front technique exploits the locality by moving recently used keys to the front of the list. This technique speeds up the future search for the same key. Hence, frequently accessed keys tend to occupy positions at the beginning of the list, while rarely used ones are typically stored at the list’s end. This behavior reduces the average look up time for keys.

2.4.1.4.2 Exact Fit

The exact fit technique provides a cache-aware option for separate chaining technique. It keeps an array of neighboring entries instead of a linked list for colliding keys per hash table item. Therefore, this method is able to exploit higher memory locality than techniques applying linked lists, where items are typically stored in random memory locations. Hence, the array entries for conflict resolution are stored in a contiguous way in memory, resulting in less cache misses and decreased lookup time. Figure 2.10 outlines a hash table exploiting the exact fit technique.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Hash Table |  | 0 | 1 |  |
| 0 |  |  | K2 D2 | K2 D2 |  |
| 1 |  |  | NULL |  |  |
| **.**  **.**  **.** | **.**  **.**  **.** |  | 0 |  |  |
| i |  |  | K1 D1 |  |  |
| **.**  **.**  **.** | **.**  **.**  **.** |  | 0 | 1 | 2 |
| m-1 |  |  | Kn Dn | K4 D4 | K5 D5 |

**Figure 2.10 Hash Table Applying Exact Fit Technique**

The benefit of the exact fit technique is simultaneously a drawback as the technique requires storing colliding hash table entries in a contiguous way. Thus inserting a new value can results in allocating memory and copying the old array’s content to the new contiguous location [5].

* + 1. Open Addressing

In another strategy, called open addressing, all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table.The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value.

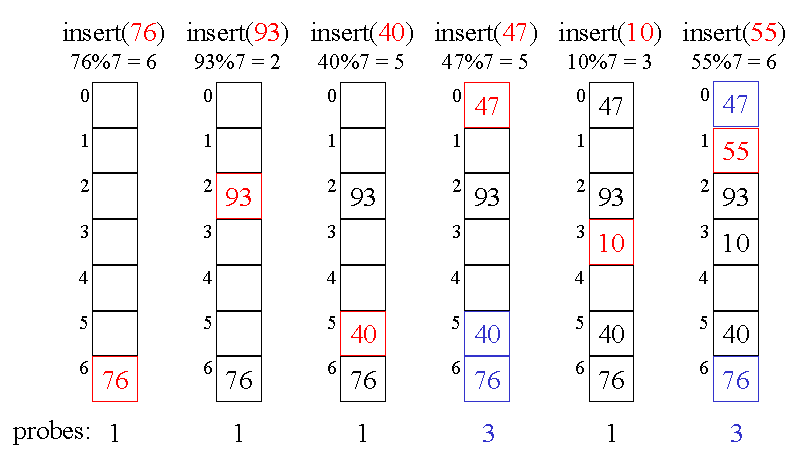
A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs. Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the same hash value; whether they are adjacent or nearby is completely irrelevant. Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two [words](https://en.wikipedia.org/wiki/Word_%28computer_architecture%29).

Open addressing avoids the time overhead of allocating each new entry record, and can be implemented even in the absence of a memory allocator. It also avoids the extra indirection required to access the first entry of each bucket (that is, usually the only one). It also has better [locality of reference](https://en.wikipedia.org/wiki/Locality_of_reference), particularly with linear probing. With small record sizes, these factors can yield better performance than chaining, particularly for lookups. Hash tables with open addressing are also easier to [serialize](https://en.wikipedia.org/wiki/Serialization), because they do not use pointers.

Well-known probe sequences include:

* [Linear probing](https://en.wikipedia.org/wiki/Linear_probing), in which the interval between probes is fixed (usually 1).
* [Quadratic probing](https://en.wikipedia.org/wiki/Quadratic_probing), in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation.
* [Double hashing](https://en.wikipedia.org/wiki/Double_hashing), in which the interval between probes is computed by a second hash function [6].
  + - 1. Linear Probing

Linear probing is a scheme in [computer programming](https://en.wikipedia.org/wiki/Computer_programming) for resolving [collisions](https://en.wikipedia.org/wiki/Hash_collision) in [hash tables](https://en.wikipedia.org/wiki/Hash_table), [data structures](https://en.wikipedia.org/wiki/Data_structure) for maintaining a collection of [key–value pairs](https://en.wikipedia.org/wiki/Attribute%E2%80%93value_pair) and looking up the value associated with a given key. It was invented in 1954 by [Gene Amdahl](https://en.wikipedia.org/wiki/Gene_Amdahl), [Elaine M. McGraw](https://en.wikipedia.org/wiki/Elaine_M._McGraw), and [Arthur Samuel](https://en.wikipedia.org/wiki/Arthur_Samuel) and first analyzed in 1963 by [Donald Knuth](https://en.wikipedia.org/wiki/Donald_Knuth). Linear probing is a form of [open addressing](https://en.wikipedia.org/wiki/Open_addressing). In these schemes, each cell of a hash table stores a single key–value pair. When the [hash function](https://en.wikipedia.org/wiki/Hash_function) causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell [7].

**Figure 2.11 Hash Collision Resolved by Linear Probing**

* + - 1. Quadratic Probing

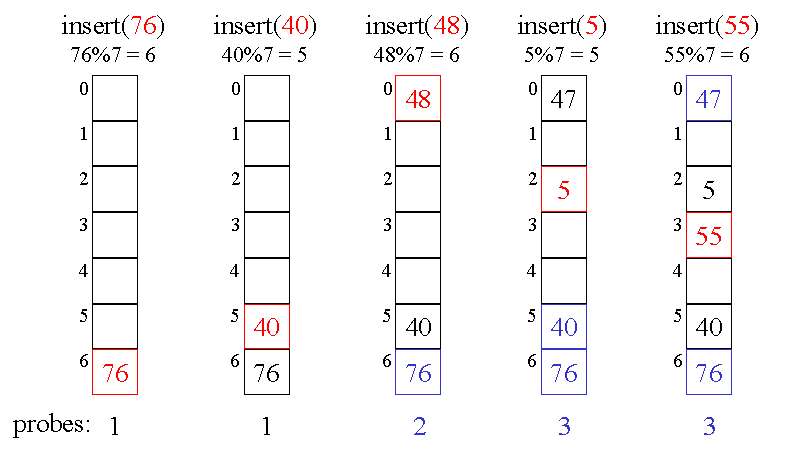
Quadratic probing is an open addressing scheme in [computer programming](https://en.wikipedia.org/wiki/Computer_programming) for resolving collisions in [hash tables](https://en.wikipedia.org/wiki/Hash_table)—when an incoming data's hash value indicates it should be stored in an already-occupied slot or bucket. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary [quadratic polynomial](https://en.wikipedia.org/wiki/Quadratic_polynomial) until an open slot is found. For a given hash value, the indices generated by [linear probing](https://en.wikipedia.org/wiki/Linear_probing) are as follows:

H + 1 , H + 2 , H + 3 , H + 4 , . . . , H + k {\displaystyle H+1,H+2,H+3,H+4,...,H+k} H+1, H+2, H+3, H+4, … , H+k

This method results in [primary clustering](https://en.wikipedia.org/wiki/Primary_clustering), and as the cluster grows larger, the search for those items hashing within the cluster becomes less efficient. An example sequence using quadratic probing is:

H+12, H+32, H+32, H+42, … , H+k2

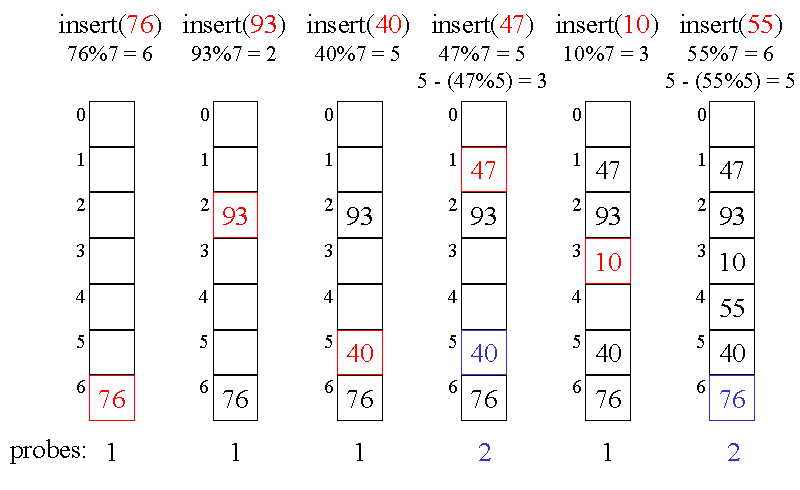
Quadratic probing can be a more efficient algorithm in a closed hash table, since it better avoids the clustering problem that can occur with linear probing, although it is not immune. It also provides good memory caching because it preserves some [locality of reference](https://en.wikipedia.org/wiki/Locality_of_reference); however, linear probing has greater locality and, thus, better cache performance [8].

****

**Figure 2.12 Hash Collision Resolved by Quadratic Probing**

* + - 1. Double Hashing

Double hashing is a [computer programming](https://en.wikipedia.org/wiki/Computer_programming) technique used in [hash tables](https://en.wikipedia.org/wiki/Hash_table) to resolve [hash collisions](https://en.wikipedia.org/wiki/Hash_collision), in cases when two different values to be searched for produce the same hash key. It is a popular [collision](https://en.wikipedia.org/wiki/Hash_collision)-resolution technique in [open-addressed](https://en.wikipedia.org/wiki/Open_addressing) hash tables. Double hashing is implemented in many popular [libraries](https://en.wikipedia.org/wiki/Library_%28computing%29). Like [linear probing](https://en.wikipedia.org/wiki/Linear_probing), it 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](https://en.wikipedia.org/wiki/Linear_probing) and [quadratic probing](https://en.wikipedia.org/wiki/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 and the effects of clustering [10].



**Figure 2.13 Hash Collision Resolved by Double Hashing**

* 1. **Choosing a Good Hash Function**

A good hash function and implementation algorithms are essential for good hash table performance, but may be difficult to achieve. A basic requirement is that the function should provide a [uniform distribution](https://en.wikipedia.org/wiki/Uniform_distribution_%28discrete%29) of hash values. A non-uniform distribution increases the number of collisions and the cost of resolving them. Uniformity is sometimes difficult to ensure by design, but may be evaluated empirically using statistical tests, e.g., a [Pearson's chi-squared test](https://en.wikipedia.org/wiki/Pearson%27s_chi-squared_test#Discrete_uniform_distribution) for discrete uniform distributions.

The distribution needs to be uniform only for table sizes that occur in the application. In particular, if one uses dynamic resizing with exact doubling and halving of the table size s, then the hash function needs to be uniform only when s is a [power of two](https://en.wikipedia.org/wiki/Power_of_two). Here the index can be computed as some range of bits of the hash function. On the other hand, some hashing algorithms prefer to have s be a [prime number](https://en.wikipedia.org/wiki/Prime_number).The modulus operation may provide some additional mixing; this is especially useful with a poor hash function.

For [open addressing](https://en.wikipedia.org/wiki/Open_addressing) schemes, the hash function should also avoid clustering, the mapping of two or more keys to consecutive slots. Such clustering may cause the look up cost to skyrocket, even if the load factor is low and collisions are infrequent. The popular multiplicative hash is claimed to have particularly poor clustering behavior. [Cryptographic hash functions](https://en.wikipedia.org/wiki/Cryptographic_hash_function) are believed to provide good hash functions for any table size s, either by [modulo](https://en.wikipedia.org/wiki/Modulo_operation) reduction or by [bit masking](https://en.wikipedia.org/wiki/Mask_%28computing%29). They may also be appropriate if there is a risk of malicious users trying to [sabotage](https://en.wikipedia.org/wiki/Denial_of_service_attack) a network service by submitting requests designed to generate a large number of collisions in the server's hash tables. However, the risk of sabotage can also be avoided by cheaper methods (such as applying a secret [salt](https://en.wikipedia.org/wiki/Salt_%28cryptography%29) to the data, or using a [universal hash function](https://en.wikipedia.org/wiki/Universal_hash_function)) [6].

**2.6 Perfect Hash Function**

A perfect hash function for a set S is a [hash function](https://en.wikipedia.org/wiki/Hash_function) that maps distinct elements in S to a set of integers, with no [collisions](https://en.wikipedia.org/wiki/Hash_collision). A perfect hash function has many of the same [applications](https://en.wikipedia.org/wiki/Hash_function#Applications) as other hash functions, but with the advantage that no collision resolution has to be implemented. In mathematical terms, it is a [total](https://en.wikipedia.org/wiki/Partial_function#Total_function) [injective function](https://en.wikipedia.org/wiki/Injective_function). If all keys are known ahead of time, a [perfect hash function](https://en.wikipedia.org/wiki/Perfect_hash_function) can be used to create a perfect hash table that has no collisions. If [minimal perfect hashing](https://en.wikipedia.org/wiki/Perfect_hash_function#Minimal_perfect_hash_function) is used, every location in the hash table can be used as well. Perfect hashing allows for [constant time](https://en.wikipedia.org/wiki/Constant_time) lookups in all cases. This is in contrast to most chaining and open addressing methods, where the time for lookup is low on average, but may be very large, O(n), for some sets of keys [10].

* 1. **Some Uses of Hash Function**

A hash function is any [function](https://en.wikipedia.org/wiki/Function_%28mathematics%29) that can be used to map [data](https://en.wikipedia.org/wiki/Data_%28computing%29) of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. One use is a data structure called a [hash table](https://en.wikipedia.org/wiki/Hash_table), widely used in computer software for rapid data lookup. Hash functions accelerate table or database look up by detecting duplicated records in a large file. An example is finding similar stretches in DNA sequences. They are also useful in [cryptography](https://en.wikipedia.org/wiki/Cryptography). A [cryptographic hash function](https://en.wikipedia.org/wiki/Cryptographic_hash_function) allows one to easily verify that some input data maps to a given hash value, but if the input data is unknown, it is deliberately difficult to reconstruct it (or equivalent alternatives) by knowing the stored hash value. This is used for assuring [integrity](https://en.wikipedia.org/wiki/Data_integrity) of transmitted data, and is the building block for [HMACs](https://en.wikipedia.org/wiki/Hash-based_message_authentication_code), which provide [message authentication](https://en.wikipedia.org/wiki/Message_authentication).

Hash functions are related to (and often confused with) [checksums](https://en.wikipedia.org/wiki/Checksums), check digits, [fingerprints](https://en.wikipedia.org/wiki/Fingerprint_%28computing%29), [randomization functions](https://en.wikipedia.org/wiki/Randomization_function), error-correcting codes, and ciphers. Although these concepts overlap to some extent, each has its own uses and requirements and is designed and optimized differently. The Hash Keeper database maintained by the American National Drug Intelligence Center, for instance, is more aptly described as a catalogue of file fingerprints than of hash values.

2.7.1 Using in Hash Table

Hash functions are primarily used in [hash tables](https://en.wikipedia.org/wiki/Hash_table), to quickly locate a data record (e.g., a [dictionary](https://en.wikipedia.org/wiki/Dictionary) definition) given its [search key](https://en.wikipedia.org/wiki/Primary_key) (the headword). Specifically, the hash function is used to map the search key to an index; the index gives the place in the hash table where the corresponding record should be stored. Hash tables, in turn, are used to implement [associative arrays](https://en.wikipedia.org/wiki/Associative_array) and [dynamic sets](https://en.wikipedia.org/wiki/Set_%28abstract_data_type%29).

Typically, the domain of a hash function (the set of possible keys) is larger than its range (the number of different table indices), and so it will map several different keys to the same index. Therefore, each slot of a hash table is associated with (implicitly or explicitly) a [set](https://en.wikipedia.org/wiki/Set_%28mathematics%29) of records, rather than a single record. For this reason, each slot of a hash table is often called a bucket, and hash values are also called bucket indices. Thus, the hash function only hints at the record's location it tells where one should start looking for it. Still, in a half-full table, a good hash function will be typically narrow the search down to only one or two entries.

### 2.7.2 Using in Building Caches

Hash functions are also used to build [caches](https://en.wikipedia.org/wiki/Cache_%28computing%29) for large data sets stored in slow media. A cache is generally simpler than a hashed search table, since any collision can be resolved by discarding or writing back the older of the two colliding items. This is also used in file comparison.

2.7.3 Usage in Finding Duplicate Records

When storing records in a large unsorted file, one may use a hash function to map each record to an index into a table T, and to collect in each bucket T[i] a [list](https://en.wikipedia.org/wiki/List_%28computing%29) of the numbers of all records with the same hash value i. Once the table is complete, any two duplicate records will end up in the same bucket. The duplicates can then be found by scanning every bucket T[i] which contains two or more members, fetching those records, and comparing them. With a table of appropriate size, this method is likely to be much faster than any alternative approach (such as sorting the file and comparing all consecutive pairs).

2.7.4 Useful in Protecting Data

A hash value can be used to uniquely identify secret information. This requires that the hash function is [collision-resistant](https://en.wikipedia.org/wiki/Collision-resistant), which means that it is very hard to find data that generate the same hash value. These functions are categorized into cryptographic hash functions and provably secure hash functions. Functions in the second category are the most secure but also too slow for most practical purposes. Collision resistance is accomplished in part by generating very large hash values. For example, [SHA-1](https://en.wikipedia.org/wiki/SHA-1), one of the most widely used cryptographic hash functions, generates 160 bit values.

### 2.7.5 Usage in Finding Similar Records

Hash functions can also be used to locate table records whose key is similar, but not identical, to a given key; or pairs of records in a large file which have similar keys. For that purpose, one needs a hash function that maps similar keys to hash values that differ by at most m, where m is a small integer (say, 1 or 2). If one builds a table T of all record numbers, using such a hash function, then similar records will end up in the same bucket, or in nearby buckets. Then one need only check the records in each bucket T[i] against those in buckets T[i+k] where k ranges between −m and m.

This class includes the so-called [acoustic fingerprint](https://en.wikipedia.org/wiki/Acoustic_fingerprint) algorithms, that are used to locate similar-sounding entries in large collection of [audio files](https://en.wikipedia.org/wiki/Audio_file). For this application, the hash function must be as insensitive as possible to data capture or transmission errors, and to trivial changes such as timing and volume changes, compression, etc.

### 2.7.6 Finding Similar Substrings

The same techniques can be used to find equal or similar stretches in a large collection of strings, such as a document repository or a [genomic database](https://en.wikipedia.org/wiki/Biological_database). In this case, the input strings are broken into many small pieces, and a hash function is used to detect potentially equal pieces, as above. The [Rabin-Karp algorithm](https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_string_search_algorithm) is a relatively fast [string searching algorithm](https://en.wikipedia.org/wiki/String_searching_algorithm) that works in [O(n)](https://en.wikipedia.org/wiki/Big_O_notation) time on average. It is based on the use of hashing to compare strings.

### 2.7.7 Geometric Hashing

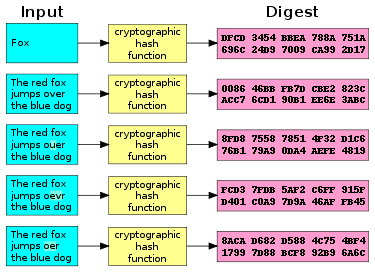
This principle is widely used in [computer graphics](https://en.wikipedia.org/wiki/Computer_graphics), [computational geometry](https://en.wikipedia.org/wiki/Computational_geometry) and many other disciplines, to solve many [proximity problems](https://en.wikipedia.org/wiki/Proximity_problem) in the [plane](https://en.wikipedia.org/wiki/Plane_%28geometry%29) or in [three-dimensional space](https://en.wikipedia.org/wiki/Three-dimensional_space), such as finding [closest pairs](https://en.wikipedia.org/wiki/Closest_pair_problem) in a set of points, similar shapes in a list of shapes, similar [images](https://en.wikipedia.org/wiki/Image_processing) in an [image database](https://en.wikipedia.org/wiki/Image_retrieval), and so on. In these applications, the set of all inputs is some sort of [metric space](https://en.wikipedia.org/wiki/Metric_space), and the hashing function can be interpreted as a [partition](https://en.wikipedia.org/wiki/Partition_%28mathematics%29) of that space into a grid of cells. The table is often an array with two or more indices (called a [grid file](https://en.wikipedia.org/wiki/Grid_file), grid index, bucket grid, and similar names), and the hash function returns an index [tuple](https://en.wikipedia.org/wiki/Tuple). This special case of hashing is known as [geometric hashing](https://en.wikipedia.org/wiki/Geometric_hashing) or the grid method. Geometric hashing is also used in [telecommunications](https://en.wikipedia.org/wiki/Telecommunication) (usually under the name [vector quantization](https://en.wikipedia.org/wiki/Vector_quantization)) to [encode](https://en.wikipedia.org/wiki/Code_%28communications%29) and [compress](https://en.wikipedia.org/wiki/Data_compression) [multi-dimensional signals](https://en.wikipedia.org/wiki/Multidimensional_signal_processing).

### 2.7.8 Standard Uses of Hashing in Cryptography

Some standard applications that employ hash functions include authentication, message integrity (using an [HMAC](https://en.wikipedia.org/wiki/HMAC) (Hashed MAC)), message fingerprinting, data corruption detection, and digital signature efficiency. A cryptographic hash function is a mathematical algorithm that maps data of arbitrary size to a bit string of a fixed size (a [hash function](https://en.wikipedia.org/wiki/Hash_function)) which is designed to also be [one-way function](https://en.wikipedia.org/wiki/One-way_function), that is, a function which is infeasible to invert. The only way to recreate the input data from an ideal cryptographic hash function's output is to try a large number of possible inputs to see if they produce a match. [Bruce Schneier](https://en.wikipedia.org/wiki/Bruce_Schneier) has called one-way hash functions "the workhorses of modern [cryptography](https://en.wikipedia.org/wiki/Cryptography)".[[1]](https://en.wikipedia.org/wiki/Cryptographic_hash_function#cite_note-1) The input data is often called the message, and the output (the hash value or hash) is often called the message digest or simply the digest.

The ideal cryptographic hash function has four main properties:

* it is quick to compute the hash value for any given message
* it is [infeasible](https://en.wikipedia.org/wiki/Computational_complexity_theory#Intractability) to generate a message from its hash value except by trying all possible messages
* a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
* it is infeasible to find two different messages with the same hash value [11]



**Figure 2.14 A Cryptographic Hash Function at Work**

**2.8 Advantages of Hash Tables**

The main advantage of hash tables over other table data structures is speed. This advantage is more apparent when the number of entries is large (thousands or more).

Hash tables are particularly efficient when the maximum number of entries can be predicted in advance, so that the bucket array can be allocated once with the optimum size and never resized.

If the set of key-value pairs is fixed and known ahead of time (so insertions and deletions are not allowed), one may reduce the average lookup cost by a careful choice of the hash function, bucket table size, and internal data structures.

In particular, one may be able to devise a hash function that is collision-free, or even perfect (see below). In this case the keys need not be stored in the table.

**2.9 Disadvantages of Hash Tables**

Hash tables can be more difficult to implement than self-balancing binary search trees. Choosing an effective hash function for a specific application is more an art than a science. In open-addressed hash tables it is fairly easy to create a poor hash function.

Although operations on a hash table take constant time on average, the cost of a good hash function can be significantly higher than the inner loop of the lookup algorithm for a sequential list or search tree. Thus hash tables are not effective when the number of entries is very small. (However, in some cases the high cost of computing the hash function can be mitigated by saving the hash value together with the key).

For certain string processing applications, such as spell-checking, hash tables may be less efficient than tries, finite automata, or Judy arrays. Also, if each key is represented by a small enough number of bits, then, instead of a hash table, one may use the key directly as the index into an array of values. There are no collisions in this case.  
 The entries stored in a hash table can be enumerated efficiently (at constant cost per entry), but only in some pseudo-random order. Therefore, there is no efficient way to efficiently locate an entry whose key is nearest to a given key. Listing all n entries in some specific order generally requires a separate sorting step, whose cost is proportional to log(n) per entry. In comparison, ordered search trees have lookup and insertion cost proportional to log(n), but allow finding the nearest key at about the same cost, and ordered enumeration of all entries at constant cost per entry.

If the keys are not stored (because the hash function is collision-free), there may be no easy way to enumerate the keys that are present in the table at any given moment.  
 Although the average cost per operation is constant and fairly small, the cost of a single operation may be quite high. In particular, if the hash table uses dynamic resizing, an insertion or deletion operation may occasionally take time proportional to the number of entries. This may be a serious drawback in real-time or interactive applications.  
 Hash tables in general exhibit poor locality of reference that is, the data to be accessed is distributed seemingly at random in memory. Because hash tables cause access patterns that jump around, this can trigger microprocessor cache misses that cause long delays. Compact data structures such as arrays, searched with linear search, may be faster if the table is relatively small and keys are integers or other short strings. According to Moore's Law, cache sizes are growing exponentially and so what is considered "small" may be increasing. The optimal performance point varies from system to system.

Hash tables become quite inefficient when there are many collisions. While extremely uneven hash distributions are extremely unlikely to arise by chance, a malicious adversary with knowledge of the hash function may be able to supply information to a hash which creates worst-case behavior by causing excessive collisions, resulting in very poor performance (i.e., a denial of service attack). In critical applications, either universal hashing can be used or a data structure with better worst-case guarantees may be preferable [12].

**2.10 Summary**

Hashing: Enables access to table items that is relatively constant and independent of the items. Hashing is a method to store data in an array so that storing, searching, inserting, and deleting data is fast. For this every second needs an unique key. The basic idea is to search for the correct position of a record with comparisons but to compute the position within the array.

Hash Function: Maps the search key of a table into a location that will contain the item. These well-defined procedure or mathematical function which convert large, possibly variable-sized amount of data into a small data, usually single integer that may serve as an index into an array. The value returned by a hash function are called hash values, hash codes, hash sums, or simply hashes.

Hash Table: An array that contains the table items, as assigned by a hash function a hash table or hash map is a data structure that uses a hash function to efficiently map certain identifiers or keys (e.g., amit) to associated values (e.g., their telephone numbers). The hash function is used to transform the key into the index (the hash) of an array element where the corresponding value is to be sought. Also hash table is a shortage location in memory or on disk that records the hashed values created by the hashing algorithm.

Load factor: The ration n/s between n and s of its bucket array where n is the number of stored entries in the hash table. The performance of most collision resolution methods depends on this load factor.