Paper Name: Data Structures
Unit No : VI
Lesson Name: Hashing and Collision Handling methods
Lesson Developer : Vandana Kalra, Assistant Professor
College/Department: College of Vocational Studies / Computer Science, University of Delhi

Table of Contents

1.4.1 Hashing

Searching a data structure is an important operation that is performed to retrieve the data quickly and efficiently. Linear Search and Binary Search are the two most commonly used techniques of searching any data structure. Both these techniques compare keys to find whether the element is present in the table or not. The search time in linear search is O(n) and in binary search is O(lg n). This search time can be reduced from these values to 1 or at least O(1), irrespective of the number of elements being searched.

1.4.1.1 Definition of Hashing

This is a different approach to searching a table which reduces the search time. The position of the key in the table is calculated based on the value of a key, which indicates the actual position of the data being searched in the table. The data can be directly accessed using the value of the key. Thus, in this method comparison of keys or any other test is not required to access the data, as also this method allows direct access to the actual data.

A ***hash function*** h is required to transform a particular key k into an index into the table. This index will take you directly to the actual data. The table must store values or keys of the type k, so that the keys generated by the hash function can be used as the index into the table.

Value addition: Did you know?
Heading Text: Types of Hashing
Body text: **Various Forms of Hashing** Hashing as a tool to associate one set or bulk of data with an identifier has many different forms of application in the real-world. Below are some of the more common uses of hash functions. • **String Hashing** Used in the area of data storage access. Mainly within indexing of data and as a structural back end to associative containers*(ie: hash tables)* • **Cryptographic Hashing** Used for data/user verification and authentication. A strong cryptographic hash function has the property of being very difficult to reverse the result of the hash and hence reproduce the original piece of data. Cryptographic hash functions are used to hash user's passwords and have the hash of the passwords stored on a system rather than having the password itself stored. Cryptographic hash functions are also seen as irreversible compression

functions, being able to represent large quantities of data with a signal ID, they are useful in seeing whether or not the data has been tampered with, and can also be used as data one signes in order to prove authenticity of a document via other cryptographic means.

- **Geometric Hashing**

  This form of hashing is used in the field of computer vision for the detection of classified objects in arbitrary scenes.

  The process involves initially selecting a region or object of interest. From there using affine invariant feature detection algorithms such as the Harris corner detector (HCD), Scale-Invariant Feature Transform (SIFT) or Speeded-Up Robust Features (SURF), a set of affine features are extracted which are deemed to represent said object or region. This set is sometimes called a macro-feature or a constellation of features. Depending on the nature of the features detected and the type of object or region being classified it may still be possible to match two constellations of features even though there may be minor disparities (such as missing or outlier features) between the two sets. The constellations are then said to be the classified set of features.

  A hash value is computed from the constellation of features. This is typically done by initially defining a space where the hash values are intended to reside - the hash value in this case is a multidimensional value normalized for the defined space. Coupled with the process for computing the hash value another process that determines the distance between two hash values is needed - A distance measure is required rather than a deterministic equality operator due to the issue of possible disparities of the constellations that went into calculating the hash value. Also owing to the non-linear nature of such spaces the simple Euclidean distance metric is essentially ineffective, as a result the process of automatically determining a distance metric for a particular space has become an active field of research in academia.

  Typical examples of geometric hashing include the classification of various kinds of automobiles, for the purpose of re-detection in arbitrary scenes. The level of detection can be varied from just detecting a vehicle, to a particular model of vehicle, to a specific vehicle.

- **Bloom Filters**

  A Bloom filter allows for the *"state of existence"* of a large set of possible values to be represented with a much smaller piece of memory than the sum size of the values. In computer science this is known as a membership query and is core concept in associative containers.

  The Bloom filter achieves this through the use of multiple distinct hash functions and also by allowing the result of a membership query for the existence of a particular value to have a certain probability of error. The guarantee a Bloom filter provides is that for any membership query there will never be any false negatives, however there may be false positives. The false positive probability can be controlled by varying the size of the table used for

the Bloom filter and also by varying the number of hash functions. Subsequent research done in the area of hash functions and tables and bloom filters by Mitzenmacher et al. suggest that for most practical uses of such constructs, the entropy in the data being hashed contributes to the entropy of the hash functions, this further leads onto theoretical results that conclude an optimal bloom filter (one which provides the lowest false positive probability for a given table size or vice versa) providing a user defined false positive probability can be constructed with at most two distinct hash functions also known as *pairwise independent hash functions*, greatly increasing the efficiency of membership queries. Bloom filters are commonly found in applications such as spell-checkers, string matching algorithms, network packet analysis tools and network/internet caches.
Source: http://www.partow.net/programming/hashfunctions/

The hash function will be used for inserting actual records or data at the right position indicated by the key generated by this function. To search an element in the table, we use the hash function to generate key and the element stored at that key is looked for. If it matches our search, we have found the element in the table. If not found, then the table does not contain our search element, which may be inserted into the table if required.

1.4.2 Hash Functions

A hash function is the main actor in hashing. This function generates keys that indicates locations where the elements should be inserted or retrieved. Hash functions are mostly used to speed up table lookup or data comparison tasks – such as finding items in a database, detecting duplicate records in a large file. The two main criteria in selecting a hash function are that it should be easy to compute, and it should achieve an even distribution of keys. A hash function can be of various types. Some of them are discussed below.

Value addition: Frequently Asked Question (FAQ)
Heading Text: What is a Hash Function?
Body text: **What is a Hash Function?** A hash function H is a transformation that takes a variable-size input m and returns a fixed-size string, which is called the hash value h (that is, h = H(m)). Hash functions with just this property have a variety of general computational uses, but when employed in cryptography the hash functions are usually chosen to have some additional properties.

The basic requirements for a cryptographic hash function are:

- the input can be of any length,
- the output has a fixed length,
- H(x) is relatively easy to compute for any given x ,
- H(x) is one-way,
- H(x) is collision-free.

A hash function H is said to be one-way if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that H(x) = h.

If, given a message x, it is computationally infeasible to find a message y not equal to x such that H(x) = H(y) then H is said to be a weakly collision-free hash function.

A strongly collision-free hash function H is one for which it is computationally infeasible to find any two messages x and y such that H(x) = H(y).

The hash value represents concisely the longer message or document from which it was computed; one can think of a message digest as a "digital fingerprint" of the larger document. Examples of well-known hash functions are MD2 and MD5 and SHA.

Perhaps the main role of a cryptographic hash function is in the provision of digital signatures. Since hash functions are generally faster than digital signature algorithms, it is typical to compute the digital signature to some document by computing the signature on the document's hash value, which is small compared to the document itself. Additionally, a digest can be made public without revealing the contents of the document from which it is derived. This is important in digital timestamping where, using hash functions, one can get a document timestamped without revealing its contents to the timestamping service.

Source: http://www.x5.net/faqs/crypto/q94.html

1.4.2.1 Division Hash Function

A hash function can use division operation as part of its definition to generate the key values. The only condition that should prevail every time the hash function is used is that the function should always generate valid index values. If the hash function generates key values which are not valid index values, the hashing can fail miserably. Thus, this should be guaranteed while defining the hash functions.

In order to satisfy the above condition, the hash function can use division modulo function. That is, each key to be inserted will be divided by the size of the table in which the values are to be stored. Then, the modulus or remainder of this division will be used to index into the table. This is also called division modulo and modulo operator (%) is used for the definition of the hash function, which is of the form –

h(k) = k % MAX_SIZE

Where h(k) is the hash function to be used, k is the key to be inserted and MAX_SIZE is the maximum size of the table into which the values are to be inserted.

For Example, assuming various values of keys and MAX_SIZE being 11, the hash function will generate the following values –

Key k	MAX_SIZE	Working	h(k)
80	11	80 % 11	3
38	11	38 % 11	5
59	11	59 % 11	4
65	11	65 % 11	10

The values generated by the hash function h(k) will be used to index the table to store the keys. That is, key 38 will be stored in the 5^{th} position in the table as the remainder of division of 38 by 11 is 5.

This is the most commonly used hash function. The advantage of this hash function is that it is simple and it is useful when we do not know much about the keys. However, not all values of MAX_SIZE are suitable. For Example, powers of 2 can be avoided; prime numbers can form good values for MAX_SIZE.

1.4.2.2 Folding Hash Function

In this method, the key is divided into several parts. These parts are combined or folded together in a particular manner and then transformed in a certain way to generate the target index into the table. The different parts can be combined together using simple operations such as addition, "xor'ing", and the like.

There are two types of folding – shift folding and boundary folding.

Shift Folding

In shift folding, the divided parts are put one beneath the other and then processed in a certain way.

For Example, the Social Security Number (SSN) 123 – 45 – 6789 can be divided into 3 equal parts as in 123, 456 and 789. They are placed one beneath the other and added as follows –

```
      1    2    3
      4    5    6
+     7    8    9
-----------------------------
1     3    6    8
-----------------------------
```

Figure 1.4.1
Source: self

This sum value cannot be used to index into the table since the table size can be a smaller value. Thus, this value can be further processed by dividing modulo by the maximum size of the table. Remember, the goal of any hash function is to generate keys which can be used to index into the table.

Further, the division of the key can be done in different ways, for example, 123 – 45 – 6789 can be divided into four equal parts and a remainder, 12, 34, 56, 78 and 9. These parts can be processed in a similar fashion as shown above.

Boundary Folding

In boundary folding, the divided parts are written on a piece of paper, and this paper is folded on the borders or boundaries of these parts. When we do this, every alternate part of the key appears reversed. As the name suggests, the folding happens on the boundaries of the divided parts of the key.

For Example, the Social Security Number (SSN) 123 – 45 – 6789 can be divided into 3 equal parts as in 123, 456 and 789. They are placed on a piece of paper, and the paper is folded on the borders of these parts, which appears as 123, 654 and 789, to be added as follows –

```
      1     2     3
      6     5     4
+     7     8     9
------------------------------
1     5     6     6
------------------------------
```

Figure 1.4.2
Source: self

This sum will be divided modulo the maximum size of the table into which the keys are to be inserted.

Value addition: Did you know?
Heading Text: Applications of Hashing
Body text: **Associative arrays**

Hash tables are commonly used to implement many types of in-memory tables. They are used to implement associative arrays (arrays whose indices are arbitrary strings or other complicated objects), especially in interpreted programming languages like AWK, Perl, and PHP.

Database indexing

Hash tables may also be used for disk-based persistent data structures and database indices (such as dbm) although balanced trees are more popular in these applications.

Caches

Hash tables can be used to implement caches, auxiliary data tables that are used to speed up the access to data that is primarily stored in slower media. In this application, hash collisions can be handled by discarding one of the two colliding entries—usually the one that is currently stored in the table.

Sets

Besides recovering the entry which has a given key, many hash table implementations can also tell whether such an entry exists or not.

Those structures can therefore be used to implement a set data structure, which merely records whether a given key belongs to a specified set of keys. In this case, the structure can be simplified by eliminating all parts which have to do with the entry values. Hashing can be used to implement both static and dynamic sets.

Object representation

Several dynamic languages, such as Python, JavaScript, and Ruby, use hash tables to implement objects. In this representation, the keys are the names of the members and methods of the object, and the values are pointers to the corresponding member or method.

Unique data representation

Hash tables can be used by some programs to avoid creating multiple character strings with the same contents. For that purpose, all strings in use by the program are stored in a single hash table, which is checked whenever a new string has to be created. This technique was introduced in Lisp interpreters under the name hash consing, and can be used with many other kinds of data (expression trees in a symbolic algebra system, records in a database, files in a file system, binary decision diagrams, etc.)

Source : http://en.wikipedia.org/wiki/Hash_table

1.4.2.3 Mid – Square Hash Function

The key value is squared in this method and the middle or mid part of the result is used as the index into the table. If this middle part is greater for the size of the table, then it is divided modulo the size of the table. In this method, the whole key participates in generating the index for the table, thus, there is a greater chance of finding a different address every time per key value.

For Example, if the key value is 2199, the squared result is 4835601. The middle part of this result is 356, which will now be used for indexing into the table. Say, if the size of the table is 1000, then

$$h(2199) = 356$$

that is, the hash function using mid – square method will produce 356 as the index value for key value 2199.

The advantage is that the result is not dominated by the distribution of bottom digit or top digit of the original key value.

The problem with this method is choosing the middle part of the result obtained by squaring the key value. To resolve this issue, it is advisable to choose the size of the table to be of a power of 2, such as 2^r so that the middle r bits are chosen. In such a case, it is easier to choose the middle part of the result using masking and shift operations.

Value addition: Frequently Asked Questions(FAQ)

Heading Text: Performance of Hashing Techniques

Body text:

Hashing Functions and Record Distributions

- The size of the key space is typically much larger than the space of hashed values.
- This means that more than one key will map to the same hash value.

Collisions

- **synonyms**
  Keys which hash to the same value.
- **collision**
  An attempt to store a record at an address which does not have sufficient room
- **packing density**

The ratio of used space to allocated space.

- For simple hashing, the probability of a synonym is the same as the packing density.

How Much Extra Memory Should be Used?

- Increasing memory (i.e., increasing the size of the hash table) will decrease collisions.

Source: http://www.comsci.us/fs/notes/ch11.html

1.4.2.4 Extract Hash Function

As the name suggests, certain values are extracted from the key to be used to compute the target address for indexing into the table. Each time, only a certain portion and combinations thereof are used for calculating the address.

For Example, the Social Security Number (SSN) 123 – 45 – 6789 can be partitioned into many ways, such as, the first four digits 1234, or the last four digits 6789, or the first two and last two combined together – 1289, and so on.

Certain values of the key which are likely to be the same in most of the records should be eliminated or should not be considered for extraction.

For Example, if we are storing the employee records of a Company named MoonLight Co. Ltd., and further assuming that the employee ID starts with ML. In such a case, while storing or retrieving the employee ID of the employees, first two characters of the key should not be extracted for calculating the address. This is so because every key will have the first two characters as ML, thereby not contributing to the uniqueness of the key.

The advantage is that it requires potentially shorter calculation and it allows us to remove similar data which is not going to affect hash anyway.

1.4.2.5 Radix Transformation Hash Function

In this method, the key is transformed from the current radix to a different radix in a numerical system. The value thus obtained will be divided modulo the size of the table in order to calculate the address.

For Example, if key is 345 in decimal number system, then its value will be 423 in the nonal number system, i.e. with base 9. This value will be divided modulo the size of the table, and the resulting value is to be used as the index into the table.

1.4.3 Advantages Vs Limitations

Hashing is a searching technique which reduces the search time. This is so because in this method, the keys are not compared to the already inserted values.

However, hashing does not guarantee unique index values. A hash function can generate same index values for different key values, thereby leading to collision. Thus, hashing involves certain amount of collision resolution methods, which will be discussed in the next section.

Value addition: Did you know?

Heading Text: Properties of Hashing

Body text:

Properties of Hashing

Good hash functions, in the original sense of the term, are usually required to satisfy certain properties listed below. Note that different requirements apply to the other related concepts (cryptographic hash functions, checksums, etc.).

Low cost

The cost of computing a hash function must be small enough to make a hashing-based solution more efficient than alternative approaches. For instance, a self-balancing binary tree can locate an item in a sorted table of n items with $O(\log n)$ key comparisons. Therefore, a hash table solution will be more efficient than a self-balancing binary tree if the number of items is large and the hash function produces few collisions and less efficient if the number of items is small and the hash function is complex.

Determinism

A hash procedure must be deterministic—meaning that for a given input value it must always generate the same hash value. In other words, it must be a function of the hashed data, in the mathematical sense of the term. This requirement excludes hash functions that depend on external variable parameters, such as pseudo-random number generators that depend on the time of day. It also excludes functions that depend on the memory address of the object being hashed, if that address may change during processing (as may happen in systems that use certain methods of garbage collection), although sometimes rehashing of the item can be done.

Uniformity

A good hash function should map the expected inputs as evenly as possible over its output range. That is, every hash value in the output range should be generated with

roughly the same probability. The reason for this last requirement is that the cost of hashing-based methods goes up sharply as the number of *collisions*—pairs of inputs that are mapped to the same hash value—increases. Basically, if some hash values are more likely to occur than others, a larger fraction of the lookup operations will have to search through a larger set of colliding table entries.

Note that this criterion only requires the value to be *uniformly distributed*, not *random* in any sense. A good randomizing function is (barring computational efficiency concerns) generally a good choice as a hash function, but the converse need not be is less than true.

Hash tables often contain only a small subset of the valid inputs. For instance, a club membership list may contain only a hundred or so member names, out of the very large set of all possible names. In these cases, the uniformity criterion should hold for almost all typical subsets of entries that may be found in the table, not just for the global set of all possible entries.

In other words, if a typical set of m records is hashed to n table slots, the probability of a bucket receiving many more than m/n records should be vanishingly small. In particular, if m n, very few buckets should have more than one or two records. (In an ideal "perfect hash function", no bucket should have more than one record; but a small number of collisions is virtually inevitable, even if n is much larger than m.

When testing a hash function, the uniformity of the distribution of hash values can be evaluated by the chi-square test.

Variable range

In many applications, the range of hash values may be different for each run of the program, or may change along the same run (for instance, when a hash table needs to be expanded). In those situations, one needs a hash function which takes two parameters—the input data z, and the number n of allowed hash values.

A common solution is to compute a fixed hash function with a very large range (say, 0 to 232−1), divide the result by n, and use the division's remainder. If n is itself a power of 2, this can be done by bit masking and bit shifting. When this approach is used, the hash function must be chosen so that the result has fairly uniform distribution between 0 and n−1, for any n that may occur in the application. Depending on the function, the remainder may be uniform only for certain n, e.g. odd or prime numbers.

It is possible to relax the restriction of the table size being a power of 2 and not having to perform any modulo, remainder or division operation -as these operation are considered computational costly in some contexts. When n is much lesser than 2^b take a pseudo random number generator (PRNG) function $P(key)$, uniform on the interval $[0, 2^b-1]$. Consider the ratio $q = 2^b / n$. Now the hash function can be seen as the value of P(key) / q. Rearranging the calculation and replacing the 2^b-division by bit shifting right (>>) b times you end up with hash function $n * P(key) >> b$.

Variable range with minimal movement (dynamic hash function)

When the hash function is used to store values in a hash table that outlives the run of the program, and the hash table needs to be expanded or shrunk, the hash table is referred to as a dynamic hash table.

A hash function that will relocate the minimum number of records when the table is resized is desirable. What is needed is a hash function $H(z,n)$ – where z is the key being hashed and n is the number of allowed hash values – such that $H(z,n+1) = H(z,n)$ with probability close to $n/(n+1)$.

Linear hashing and spiral storage are examples of dynamic hash functions that execute in constant time but relax the property of uniformity to achieve the minimal movement property.

Extendible hashing uses a dynamic hash function that requires space proportional to n to compute the hash function, and it becomes a function of the previous keys that have been inserted.

Several algorithms that preserve the uniformity property but require time proportional to n to compute the value of $H(z,n)$ have been invented.

Data normalization

In some applications, the input data may contain features that are irrelevant for comparison purposes. For example, when looking up a personal name, it may be desirable to ignore the distinction between upper and lower case letters. For such data, one must use a hash function that is compatible with the data equivalence criterion being used: that is, any two inputs that are considered equivalent must yield the same hash value. This can be accomplished by normalizing the input before hashing it, as by upper-casing all letters.

Continuity

A hash function that is used to search for similar (as opposed to equivalent) data must be as continuous as possible; two inputs that differ by a little should be mapped to equal or nearly equal hash values.

Note that continuity is usually considered a fatal flaw for checksums, cryptographic hash functions, and other related concepts. Continuity is desirable for hash functions only in some applications, such as hash tables that use linear search.

Source: http://en.wikipedia.org/wiki/Hash_function

1.5.1 Collision Handling Methods

"Every coin has two sides" is a well – known quote which says that everything or situation has two sides to it – one positive and one negative. The positive side of hashing is that it

reduces the search time and is easy to implement. The negative side of hashing, that is that it leads to collision, although is far bigger than the positive one, solutions have been found to deal with the situation.

Any hash function, whether simple or complex can lead to collision, which means that two different key values can be given the same index value in the table. Thus, collision resolution methods are required to be in place, which allocate a different location in the table each time a new key value is to be inserted.

1.5.2 Open Addressing

When two keys collide for the same index value, open addressing method is used to resolve the collision. The key entering the system goes through one of the hash functions generating an index value. This index value is searched in the table for inserting the key value. If the index position already has a value, we say there is a collision. In this method, the table is searched for the next available cell in it and the key is placed in that cell, if it is empty. This process is repeated until these same positions are searched repeatedly or the table is full. Open addressing is also known as closed hashing because collided keys are stored within the table.

The simplest case of open addressing is when the empty cell for the key to be inserted is looked linearly, that is, next available position is looked for.

For Example, consider the maximum size of the table is 10 and the hash function is division modulo. The value 25 is stored as follows –

0	1	2	3	4	5	6	7	8	9
					25				

Figure 1.5.1
Source: self

Key value 36 will be stored at position 6 and 47 will be stored at position 7, which makes the table look like –

0	1	2	3	4	5	6	7	8	9
					25	36	47		

Figure 1.5.2
Source: self

Now, when key value 85 enters the system, the hash function produces index value 5 which is looked into the table, but this position already has a key value stored in it. The next empty position is searched for in the table. The next two positions, that is, 6 and 7 are also

occupied, thus, these positions cannot be occupied. Searching the table linearly, the next empty and available position is 8, where this key will be inserted, making the table look like this –

0	1	2	3	4	5	6	7	8	9
					25	36	47	85	

Figure 1.5.3
Source: self

Cluster Build – up

Open addressing leads to cluster build – up. When a cluster is created, it will grow larger. The larger the cluster, the larger is the likelihood that it will grow even larger. A cluster is a collection of index positions which contain collided key values. These key values are stored at the next available empty position.

1.5.3 Chaining

Open addressing for collision resolution has the problem of creating clusters. Large clusters in the table are an impediment to insertion and retrieval of data in the table. Thus, chaining is a solution given to resolve the collisions.

In chaining, keys are not necessarily stored in the table. The table is a collection of pointers or references which store linked list of keys that hash to the same index value. The problem of cluster build – up is avoided.

Each position in the table is a pointer which links to the first key value, which hashes to this index value, say i. When a new key value hashes to the index position i, this key value is attached to the end of the linked list placed at this index position. This way, each collision is resolved by creating a chain of structures of key values hashing to the same position.

For Example, consider the maximum size of the table is 10 and the hash function is division modulo. The values 25, 36 and 47 are stored in the table as shown in Figure 1.5.4

When key value 85 is inserted into the table, there is a collision with key value 25. This collision is resolved by creating a linked list of values which originate from the actual index position which is 5. Each node consists of a data part and a pointer part. The data part stores the key value or the actual record to be stored. The pointer part stores the next pointer, which pointes to the next node inserted in the list or NULL, if it is the last node. The next node's data part is the key which had a collision while hashing with the previously inserted key values.

The same steps are executed for inserting key value 96 as it collides with key value 36.

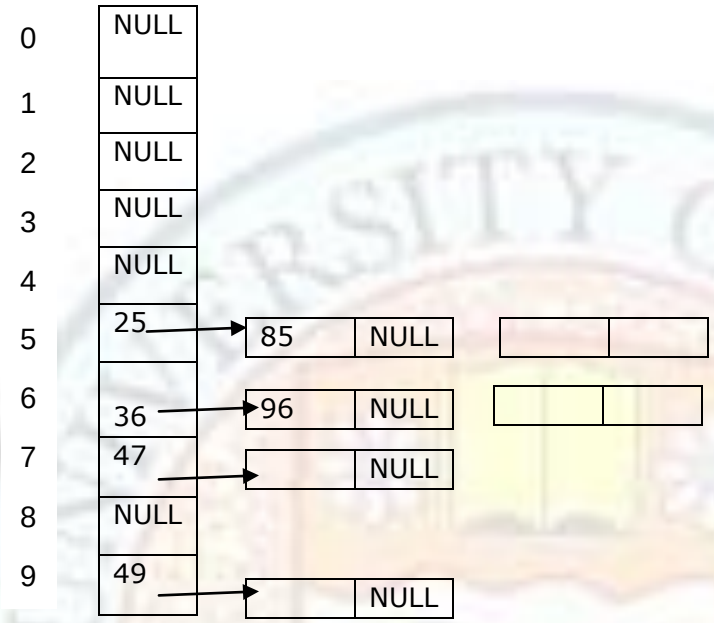


Figure 1.5.4
Source: self

When we insert key value 49, there is no collision and new node is inserted into the index position 9, forming a new linked list at this place. The same is true for key value 47. For all other index positions, NULL value is stored indicating that no key values are stored at these index positions.

The advantage of this method is that there is no need to search the table on collision and it can store as many keys as we want at one address.

The disadvantage is that it involves linked list operations.

1.5.3.1 Coalesced Hashing

The chaining method discussed above requires additional space for maintaining pointers. The table stores only pointers but each node of the linked list requires storage space for data as well as one pointer field. Thus, for n keys, n + MAX_SIZE pointers are needed, where MAX_SIZE is the maximum size of the table in which values are to be inserted. If the value of n is large, the space required to store this table is quite large.

The solution to this problem is called coalesced hashing or coalesced chaining. This method is the hybrid of chaining and open addressing. Each index position in the table stores key

value and a pointer to the next index position. The pointer generally points to the index position where the colliding key value will be stored.

In this method, the next available position is searched for a colliding key and is placed in that position. After each such insertion, pointer re – adjustment is required. After inserting the key values at the right place, the next pointer of the previous position is made to point to the position where the colliding key is inserted. In this method, instead of allocating new nodes for the linked list of keys with collision, empty position from the table itself is allocated.

For Example, the values 25, 36, and 47 will be inserted thus in the table –

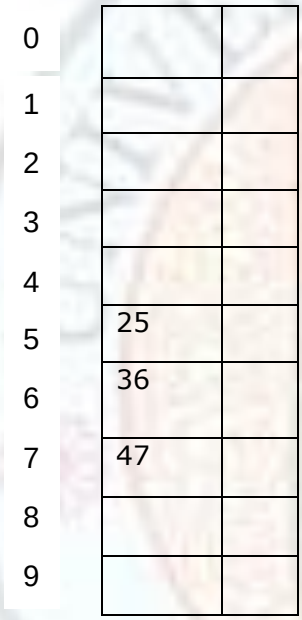
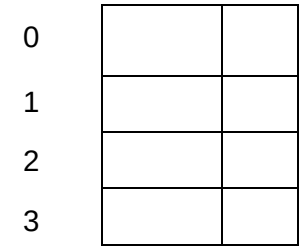


Figure 1.5.5
Source: self

Now, we insert key value 85 into this table. This method starts inserting the collided key values from the bottom of the table. Key value 85 will go in at index position 9 in the table and the pointer will be re – adjusted. That is, the next pointer of position 5 will point to index position 9.



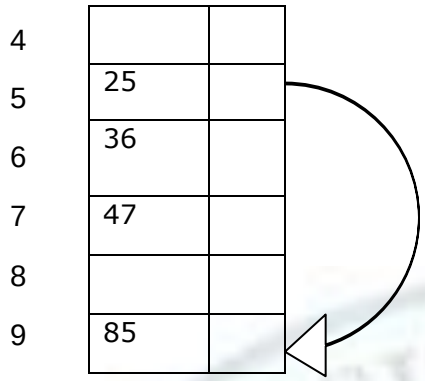


Figure 1.5.6
Source: self

Index position 9 is full and any key value hashing into this position will have to be inserted into the next available empty location, starting from the bottom of the table. So, if we insert key value 49 into the table, it will go into index position 8 with pointer re – adjustment. The table will look like –

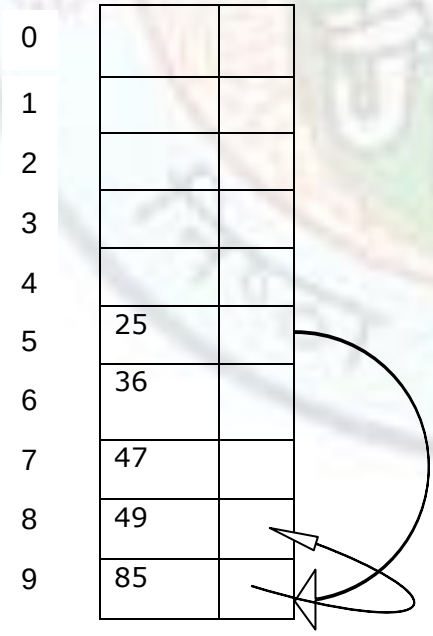


Figure 1.5.7
Source: self

This process will continue for all the colliding key values.

1.5.4 Bucket Addressing

Bucket addressing for collision resolution allows the table to store all colliding key values at the same position. This can be achieved by associating a bucket with each address. A **bucket** is simply, a block of space large enough to hold multiple key values. By using buckets, the problem of collision is not totally avoided, but is minimized to a certain extent.

In this implementation, hash table slots are grouped into buckets. The M slots of table are divided into B buckets, with each bucket consisting of M/B slots.

When the first key value is to be inserted into the table, the key value is hashed to determine the bucket where it will be placed . The subsequent key values may result in collision. If a collision occurs, the particular key value is stored in the bucket provided for this key value. If this bucket is also full, then this key value has to be stored in overflow bucket of infinite capacity at the end of the table. Overflow bucket is common for all the buckets.

For Example, consider the maximum size of the table is 10 and the hash function is division modulo. The value 25, 36 and 47 are stored as shown in the figure 1.5.8. In this example, the bucket size is 2.

These are the first few key values which do not lead to any collisions, and thus, are stored without any problem. The subsequent key values may result in collisions. The next key value to be inserted is 85, and using bucket addressing will go into the bucket provided with each index position. This is shown in the figure 1.5.8, where key value 85 is stored in the bucket provided at index position 5.

0		
1		
2		
3		
4		
5	25	85
6	36	
7	47	
8		
9		

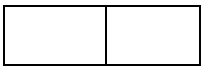


Figure 1.5.8
Source: self

The table will look thus, when we insert key values 96 and 49.

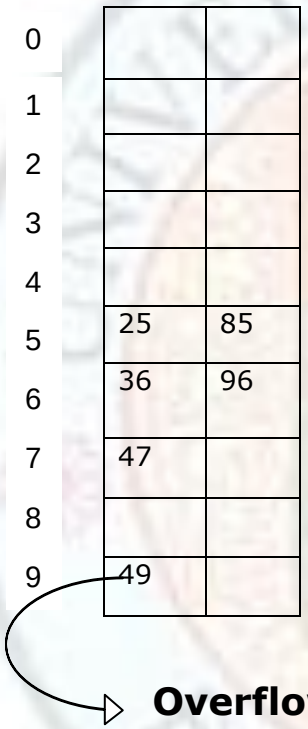


Figure 1.5.9
Source: self

Value addition: Did you know?

Heading Text: More Collision Resolution Methods

Body text:

Load factor

The performance of most collision resolution methods does not depend directly on the number *n* of stored entries, but depends strongly on the table's *load factor*, the ratio *n/s* between *n* and the size *s* of its bucket array. With a good hash function, the average lookup cost is nearly constant as the load factor increases from 0 up to 0.7 or so. Beyond that point, the probability of collisions and the cost of handling them increases.

On the other hand, as the load factor approaches zero, the size of the hash table increases with little improvement in the search cost, and memory is wasted.

Separate chaining with other structures

Instead of a list, one can use any other data structure that supports the required operations. By using a self-balancing tree, for example, the theoretical worst-case time of a hash table can be brought down to O(log n) 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 expects to have many entries hashed to the same slot (e.g. if one expects extremely non-uniform or even malicious key distributions).

The variant called array hash table uses a 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 and the TLB (Translation lookaside buffer), since slot entries are stored in sequential memory positions. It also dispenses with the next pointers that are required by linked lists, which saves space and despite frequent array resizing, space overheads incurred by operating system such as memory fragmentation, were found to be small.

An elaboration on this approach is the so-called 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), this variant has guaranteed constant worst-case lookup time, and low amortized time for insertion.

Robin Hood hashing

One interesting variation on double-hashing collision resolution is that of Robin Hood hashing. The idea is that a key already inserted may be displaced by a new key if its probe count is larger than the key at the current position. The net effect of this is that it reduces worst case search times in the table. This is similar to Knuth's ordered hash tables except that the criterion for bumping a key does not depend on a direct relationship between the keys. Since both the worst case and the variation in the number of probes is reduced dramatically, an interesting variation is to probe the table starting at the expected successful probe value and then expand from that position in both directions. External Robin Hashing is an extension of this algorithm where the table is stored in an external file and each table position corresponds to a fixed sized page or bucket with B records.

Cuckoo hashing

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilisation can be achieved.

Hopscotch hashing

Another alternative open-addressing solution is hopscotch hashing, which combines the approaches of cuckoo hashing and linear probing, yet seems in general to avoid their limitations. In particular it works well even when the load factor grows beyond 0.9. The algorithm is well suited for implementing a resizable concurrent hash table.

The hopscotch hashing algorithm works by defining a neighborhood of buckets near the original hashed bucket, where a given entry is always found. Thus, search is limited to the number of entries in this neighborhood, which is logarithmic in the worst case, constant on average, and with proper alignment of the neighborhood typically requires one cache miss. When inserting an entry, one first attempts to add it to a bucket in the neighborhood. However, if all buckets in this neighborhood are occupied, the algorithm traverses buckets in sequence until an open slot (an unoccupied bucket) is found (as in linear probing). At that point, since the empty bucket is outside the neighborhood, items are repeatedly displaced in a sequence of hops (in a manner reminiscent of cuckoo hashing, but with the difference that in this case the empty slot is being moved into the neighborhood, instead of items being moved out with the hope of eventually finding an empty slot). Each hop brings the open slot closer to the original neighborhood, without invalidating the neighborhood property of any of the buckets along the way. In the end the open slot has been moved into the neighborhood, and the entry being inserted can be added to it.

Source: http://en.wikipedia.org/wiki/Hash_table

1.5.5 Perfect Hash Function

A hash function is a perfect hash function when the hash function guarantees the generation of unique index values. In other words, a perfect hash function guarantees no collision mapping of keys to the index values in the table. A perfect hash function with values in a limited range can be used for efficient lookup operations, by placing keys in a table indexed by the output of the function. These are rare but if the data set is well – defined and known, we can design it. A perfect hash function is minimal if it results in a full table (i.e. no wasted space in the table). It maps n keys to n consecutive integers.

Summary

- Hashing is a searching technique which does not involve comparison between keys, thereby reducing the search time.
- A hash function is a function which converts a key value into an index position, using which the key will be inserted or searched in the table.
- There are various types of hashing depending on the type of hash functions used.
- Collision resolving methods are required in hashing as the hash functions do not guarantee unique index positions.
- A perfect hash function is a hash function which guarantees that a unique index position will be generated every time hashing is performed.

Exercises

1.1 Define ADT for an array.

1.2 Implement the algorithm for matrix addition using C++.

1.3 Differentiate between stack as an ADT and stack as a data structure.

1.4 List few benefits of using ADT.

1.5 How do you calculate address for the following statement?

```
int arr1[3][4][5][6][7];
```

Glossary

Boundary Folding: In boundary folding, every alternate divided part of the key appears reversed.

Bucket Addressing: In this method, buckets (a block of space large enough to hold multiple key values) are used to store colliding key values.

Chaining: In chaining, keys are not stored in the table; they are stored as linked lists starting at each index position.

Coalesced Hashing: In this method, the next available position is searched for a colliding key and is placed in that position. After each such insertion, pointer re – adjustment is required.

Hashing: It is a searching technique which does not involve comparison of keys.

Hash Function: It is a function which takes key value as the input and produces an index position of the table.

Open Addressing: In this method, the table is searched for the next available empty cell and the key is placed in that cell.

Perfect Hash Function: It is a hash function which guarantees that a unique index position will be generated every time hashing is performed.

Shift Folding: In this method, the divided parts of the key are put one beneath the other and then processed in a certain way.

References

1. Object-Oriented Programming in C++ , Robert Lafore.
2. C++ plus data structures, Nell Dale
3. Data Structures, Algorithms, And Applications In C++, Sartaj Sahni
4. Data Structures and Algorithms in C++, Adam Drozdek
5. http://en.wikipedia.org/wiki/Hash_table
6. http://en.wikipedia.org/wiki/Hash_function
7. http://www.x5.net/faqs/crypto/q94.html
8. http://www.partow.net/programming/hashfunctions/