Hashing in C

Hashing

Hashing is an efficient method to store and retrieve elements.

It’s exactly same as index page of a book. In index page, every topic is associated with a page number. If we want to look some topic, we can directly get the page number from the index.

Likewise, in hashing every value will be associated with a key. Using this key, we can point out the element directly.

In searching method, to reduce the time complexity than any other data structure hashing concept is introduced which has O(1) time in the average case and the worst case it will take O(n) time.

Hashing is a technique with faster access to elements that maps the given data with a lesser key for comparisons. In general, in this technique, the keys are traced using hash function into a table known as the hash table.

Hash Table

Hash table is a data structure that represents data in the form of key-value pairs. Each key is mapped to a value in the hash table. The keys are used for indexing the values/data. A similar approach is applied by an associative array.

Data is represented in a key value pair with the help of keys as shown in the figure below. Each data is associated with a key. The key is an integer that point to the data.

In a hash table, the keys are processed to produce a new index that maps to the required element. This process is called hashing.

Hash Function

The hash function is a function that uses the constant-time operation to store and retrieve the value from the hash table, which is applied on the keys as integers and this is used as the address for values in the hash table.

A **hash function** is any function that can be used to map data of arbitrary size to fixed-size values. The values returned by a hash function are called *hash values*, *hash codes*, *digests*, or simply *hashes*. The values are used to index a fixed-size table called a *hash table*. Use of a hash function to index a hash table is called *hashing*.

To achieve a good hashing mechanism, It is important to have a good hash function with the following

basic requirements:

1. Easy to compute: It should be easy to compute and must not become an algorithm in itself.
2. Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3. Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Types of a Hash Function

1. Division method

In this method, the hash function is dependent upon the remainder of a division.

Example: elements to be placed in a hash table are 42,78,89,64 and let's take table size as 10.

Hash (key) = Elements % table size;

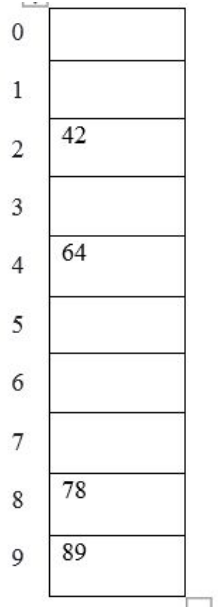
2 = 42 % 10;

8 = 78 % 10;

9 = 89 % 10;

4 = 64 % 10;

The table representation can be seen as:



2. Mid Square Method

In this method, the middle part of the squared element is taken as the index.

Element to be placed in the hash table are 210, 350, 99, 890 and the size of the table be 100.

210* 210 = 44100, index = 1 as the middle part of the result (44100) is 1.

350* 350 = 122500, index = 25 as the middle part of the result (122500) is 25.

99* 99 = 9801, index = 80 as the middle part of the result (9801) is 80.

890* 890 = 792100, index = 21 as the middle part of the result (792100) is 21.

3. Digit Folding Method

In this method, the hash key is obtained by dividing the elements into various parts and then combine the parts by performing some simple mathematical operations.

Element to be placed are 23576623, 34687734.

- hash (key) = 235+766+23 = 1024
- hash key) = 34+68+77+34 = 213

In these types of hashing suppose we have numbers from 1- 100 and size of hash table =10. Elements = 23, 12, 32

Hash (key) = 23 % 10 = 3;

Hash (key) = 12 % 10 = 2;

Hash (key) = 32 % 10 = 2;

From the above example notice that both elements 12 and 32 points to 2nd place in the table, where it is not possible to write both at the same place such problem is known as a collision. To avoid this kind of problems there are some techniques of hash functions that can be used.

Limitations of a Hash Table

- If the same index is produced by the hash function for multiple keys then, conflict arises. This situation is called collision.

  To avoid this, a suitable hash function is chosen. But, it is impossible to produce all unique keys.Thus a good hash function may not prevent the collisions completely however it can reduce the number of collisions.

However, we have other techniques to resolve collision.

Types of Collision Resolution Techniques

1. Separate chaining (open hashing)

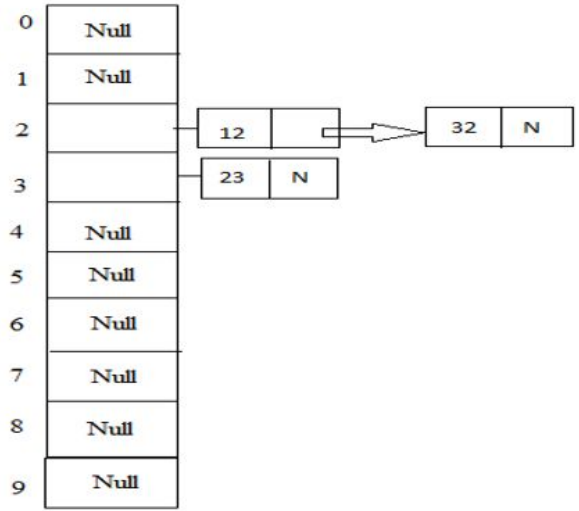
Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list.

Example: 23, 12, 32 with table size 10.

Hash (key) = 23 % 10 = 3;

Hash (key) = 12 % 10 =2;

Hash (key) = 32 % 10 =2;



2. Open Addressing (closed hashing)

- Linear Probing

In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let's assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

index = index % hashTableSize

index = (index + 1) % hashTableSize

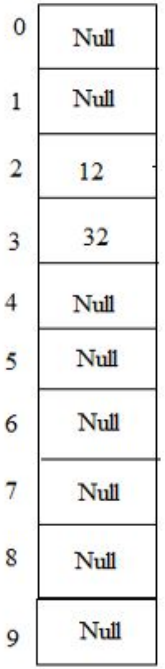
index = (index + 2) % hashTableSize

index = (index + 3) % hashTableSize and so on…

Example: 12, 32 with table size 10

Hash (key) = 12 % 10 =2;

Hash (key) = 32 % 10 =2;



In this diagram 12 and 32 can be placed in the same entry with index 2 but by this method, they are placed linearly.

- Quadratic probing

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize

index = (index + 1^2) % hashTableSize

index = (index + 2^2) % hashTableSize

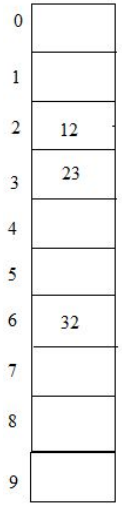
index = (index + 3^2) % hashTableSize and so on…

Example: 23, 12, 32 with table size 10

Hash (key) = 23 % 10 = 3;

Hash (key) = 12 % 10 =2;

Hash (key) = 32 % 10 =2;



In this, we can see that 23 and 12 can be placed easily but 32 cannot as again 12 and 32 shares the same entry with the same index in the table, as per this method hash (key) = (32 + 1*1) % 10 = 3. But in this case table entry with index 3 is placed with 23 so we have to increment x value by 1. Hash (key) = (32 + 2 * 2) % 10 = 6. So we now can place 32 in the entry with index 6 in the table.

- ## Double hashing

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot. The probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;

index = (index + 2 * indexH) % hashTableSize;

and so on…

Here, indexH is the hash value that is computed by another hash function.

Conclusion

Hashing is one of the important techniques in terms of searching data provided with very efficient and quick methods using hash function and hash tables. Each element can be searched and placed using different hashing methods. This technique is very faster than any other data structure in terms of time coefficient.