

CHAPTER 11

HASHING

“Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.” -Isaac Asimov

In the previous chapters, we have talked about different data storing and data accessing technique in the data structure. Irrespective of linear and non-linear data structure, the data are stored sequentially and searching of data is done in a sequential order. However, in some cases when the amount of data, then too large parallels accessing of memory space can increase the efficiency of the program. This can be achieved by Hashing.

Hashing or hash addressing is the process of mapping a large amount of data items to a smaller table with the help of a hash function. Hashing is a searching technique, which is essentially independent of the number of elements.

A hash table or Hashmap is a data structure, which uses a hash function to generate a key corresponding to the associated value. The advantage of this searching method is its efficiency to handle a large amount of data item in a given collection.

The two primary problems associated with the creation of hash tables are:

- i) The efficient hash function is designed so that it distributes the index values of inserting objects uniformly across the table.
- ii) The efficient collision resolution algorithm is designed so that it computes an alternative index for an object whose hash index corresponds to an object previously inserted into the hash table.

Hash Functions

A hash function is a function or process that map variable length data to fixed size data. The value generated by the hash function is called hash value, hash codes, hash sum or simply hashes. A hash function is used to speed up table lookup or data comparison task such as searching for an item in a database, find redundant data in a database or large files.


Definition: The fixed process to convert a key to a hash value is known as the hash function.

Hashing function or hash function maps the key with the corresponding key address or location. It provides the key to address transformation. Hash function (H) is used by a hash table to compute a set (L) of a memory address from the set K of the key. Thus, hash function denoted as: $H : K \rightarrow L$


The mapping between the key value and the location is not easy to maintain. The hash function will generally map several different keys to the same index. If the desired record is in the location given by the index, then there will not be any problem. However, if the record is not found, then either another hash function or same hash function is used.


KEY FEATURES

 Hash Functions

 Linear Probing

 Quadratic Probing

 Double Hashing

 Separate Chaining

Choice of Hash Function

The choice of the hash function is a very difficult job in hashing. It is not possible for selecting a perfect hash function, which is suitable for all different kinds of problem. There may be three possible ways to choose a hash function.

- Perfect Hash Function: There is no feasibility of this type of hash function if the data is large because practically it is not possible for huge data.
- Desirable Hash Function: For these hash function the address space should be small and collision should be kept very less or minimum.

Criteria of Hash Function

The two main criteria used in selecting a hash function are:

- Function H should be very easy and quick to compute.
- The function H should as far as possible uniformly distribute the hash address throughout the set L. So, there is a minimum number of the collision.

Typical Hash functions are as follows:

Division method

Division method is the simplest and most common way of implementing a hash function for a large volume of data. The aim of this function is to compute the index of a hash table.

The hash function H is defined by

$$H(K) = K \bmod m, \quad [\text{generates index value from } 0 \text{ to } m-1]$$

or

$$H(K) = (K \bmod m) + 1, \quad [\text{generates index value from } 1 \text{ to } m]$$

The number m is usually chosen to be a prime number (since this frequently minimizes the number of collisions). Here, the first function denotes the remainder when k is divided by m. The second function is used if the hash address ranges from 1 to m rather than from 0 to m-1.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100. Select M = 97 is a prime number

$$H(3205) = 3205 \bmod 97 = 4$$

$$H(7148) = 7148 \bmod 97 = 67$$

$$H(2345) = 2345 \bmod 97 = 17$$

Mid squares method

In mid square method, the key is multiplied by itself (i.e. k^2) and select a number of digits from the middle of the result. How many digits you select will depend on your table size and the size of your hash key. If the square is considered as the decimal number, the table size must be a power of 10. The mid square method squares the key value and then takes out the middle digits of the square result.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100.

K:	3205	7148	2345
K^2 :	10272025	51093904	5499025
H(K):	72	93	99

Folding method

The key K is partitioned into a number of parts, K_1, K_2, \dots, K_r , where each part except possibly the last has the same number of digits as the required address. Then the parts are added together, ignoring the last carry.

$$H(K) = K_1 + K_2 + \dots + K_r$$

Folding method can be classified into three types.

- Pure Fold method
- Fold Shifting method
- Fold Boundary

Pure Fold method: In this technique, parts of numbers are added together without any change.

Example:

Suppose key values are 3205, 7148, 2345 and table size 100.

$$H(3205) = 32 + 05 = 37$$

$$H(7148) = 71 + 48 = 19 \quad [\text{ignoring the last carry}]$$

$$H(2345) = 23 + 45 = 68$$

Fold Shifting method: In this method the alternate parts are reversed before addition.

Example:

Suppose key is 1522756

$$H(1522756) = 01 \ 52 \ 27 \ 56 \ (\text{as the number contains 7 digits, so 0 is padded with left most digit})$$

$$H(1522756) = 10 + 52 + 72 + 56 = 36 \quad [\text{ignoring the last carry}]$$

Fold Boundary: Here, the first and last part of the number are reversed before addition.

Example:

Suppose key is 1522756

$$H(1522756) = 01 \ 52 \ 27 \ 56 \ (\text{as the number contains 7 digits, so 0 is padded with left most digit})$$

$$H(1522756) = 10 + 52 + 27 + 65 = 54 \quad [\text{ignoring the last carry}]$$

Operations on Hash Table

The hash functions used to perform different operations on hash tables.

Table 11.1 Operations on Hash table

<u>Operation</u>	<u>Description</u>
<u>Insertion</u>	This operation inserts a key in the Hash Table.
<u>Deletion</u>	<u>This</u> operation removes a key from the Hash Table.
<u>Searching</u>	This operation searches a desired key value within the Hash Table.

Collision Resolutions

Sometimes a hash function H may produce the same values rather than distinct values; it is possible that two different keys K_1 and K_2 will produce the same hash address. This situation is called **Collision**. Some methods must be used to resolve it.

There are two broad ways of collision resolution technique.

- i) Open Addressing /Closed Hashing (in array based implementation)
- ii) Separate Chaining / Open Hashing (in array of linked list implementation)

There are many ways of Open Addressing/ Close Hashing

- Linear Probing
- Quadratic Probing
- Double Hashing or Rehashing

Linear Probing

Linear probing is a collision resolution technique that used to resolve the collision by sequentially searching the hash table for a first available location. We assume that the hash table T with m location is circular so that T(0) comes after T(m - 1).

For insertion, a new record with key k is to be added to the hash table T, with a memory location $H(K) = h$, if an empty slot is available, otherwise try with $h+1$, $h+2$, ... etc. for first available location.

For searching a record R with key K, find out memory location $H(k) = h$, and then searches the table T by linearly searching the location, which $h+1$, $h+2$, ... until finding R or meeting an empty location, which indicates an unsuccessful search.

Insertion in a Hash Table

In the linear probing, key will be inserted into the hash table if an empty slot is available. When a collision occurs, the key will be stored in the next available slot in the hash table, assuming that the table is not already full. This is done by linear search (i.e. Linear probe) for an empty slot from the collision slot. When end of table reached during linear search, the search will wrap around to the beginning of the table and continue from there. If an empty slot is not found, then the table is full.

Example:

Suppose input keys in hash table are {65, 12, 27, 38, 49, 80, 10, 35, 97} and hash table size is 10, where, hash key = key mod table size

	Index	Value
	[0]	80
	[1]	
65 mod 10 = 5	[2]	12
12 mod 10 = 2	[3]	
27 mod 10 = 7	[4]	
38 mod 10 = 8	[5]	65
49 mod 10 = 9	[6]	
80 mod 10 = 0	[7]	27
	[8]	38
	[9]	49

Figure 11.1a: Linear Probing without collision

	Index	Value
	[0]	80
	[1]	10
Add remaining keys 10, 35, 97 to the previous hash table:	[2]	12
10 mod 10 = 0, collision occurs, inserted in the next slot.	[3]	97
35 mod 10 = 5, collision occur, inserted in the next slot	[4]	
	[5]	65
	[6]	35
	[7]	27
	[8]	38
	[9]	49

Figure 11.1b: Linear Probing Showing Collision

Following algorithm describes the process of insertion of key in a hashtable using a Linear Probing collision resolution technique. The Flag array is used to indicate whether HashTable consist of key value at HashIndex. If HashTable has key at HashIndex then HashIndex of flag array is set to true else it is set to false. Initially flag array is initialised with false.

Algorithm of Insertion in Linear Probing

Algorithm: INSERT_LINEAR_PROBING (HashTable, Key, Size)
[HashTable is an array represents a hash table, Key is a value to be inserted as key and Size is the size of the HashTable]

1. Set HashIndex = Key mod Size
2. Repeat while HashTable[hashIndex] ≠ NULL
3. If HashTable[hashIndex] = NULL then
 - i) Set HashTable[hashIndex] = Key
 - ii) Set Flag[HashIndex] = true
 - iii) Return
 Else
 Set HashIndex =(HashIndex + 1) mod Size
 [End of If]
4. Return

Searching in Hash Table

In linear probing, searching a key in the hashtable with the slot at location $H(K) = h$, and continue linear search the adjacent slots in the table with $h+1, h+2, \dots$ until finding either an empty slot, which indicates an unsuccessful search or finding a slot whose stored key.

Following algorithm describes the process of searching a key in a hash table using a Linear Probing technique.

Algorithm of Searching with Linear Probing**Algorithm: SEARCH_PROBING (HashTable, Key, Size)**

[HashTable is an array represents a hashtable, Key is a searching key and Size is the size of the HashTable]

1. Set HashIndex = Key mod Size
2. Set count=0
3. Repeat while HashTable[hashIndex] \neq Key and count<size
 Set hashIndex = (hashIndex + 1) mod Size
 count = count+1
 [End of While]
4. If HashTable[hashIndex] = Key then
 Print "FOUND"
 Else
 Print "NOT FOUND"
 [End If]
5. Return

Deletion in a Hash Table

In the following algorithm, the process of deleting a key in a hashtable using a Linear Probing technique is described. Primarily searching a key in the hashtable with the slot at location $H(K) = h$, and continue linear search the adjacent slots in the table with $h+1, h+2, \dots$ until finding either an empty slot, which indicates an unsuccessful search or finding a slot whose stored key. When the key is found, then that location is set to NULL and flag at that location is set to false.

Algorithm of Deletion in Linear Probing**Algorithm: DELETE_PROBING (HashTable, Key, Size)**

[HashTable is an array represents a hash table, Key is value to be deleted from HashTable and Size is the size of the HashTable]

1. Set HashIndex = Key mod Size
2. Set count = 0
3. Repeat while HashTable[HashIndex] \neq Key and count<size
 Set HashIndex=(HashIndex + 1) mod Size
 count = count + 1
 [End of while]
4. If HashTable[HashIndex] = Key then
 i) Set HashTable[HashIndex] = NULL
 ii) Set Flag[HashIndex] = false
 Else
 Print "NOT FOUND"
 [End If]
5. Return

The main drawback of linear probing is that records tend to form primary cluster, that is a contiguous block of items and when a new key hashes in the cluster, then the cluster size increases and

the cluster need several attempts to resolve the collision. As a result increases the average search time for a record. Insertion and searching time depend on the length of the cluster.

An example of primary clustering:

Example:

Suppose input keys {89, 18, 49, 58, 69} and table size is 10. Using Linear probing keys are inserted in the index shown in the following table.

Index	Value
[0]	49
[1]	58
[2]	69
[3]	
[4]	
[5]	
[6]	
[7]	
[8]	18
[9]	89

Figure 11.2: Primary Clustering in Linear Probing

To overcome this limitation of linear probing another collision resolution technique is used which is known as Quadratic Probing.

Quadratic Probing

Suppose a record R with key K has the hash address $H(K) = h$. Then, instead of searching the location with address $h, h+1, h+2, \dots$ we search the locations by address

$h, h+1, h+4, h+9, h+16, \dots, h+i^2, \dots$

If the number m of location in the table T is a prime number, then the above sequence will access half of the locations in T . Quadratic probing is a more efficient algorithm in a closed hash table, since it better avoids the primary clustering problem that can occur linear probing.

With quadratic probing, there is no guarantee of finding an empty cell once the table gets more than half-full or even before the table gets half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve the collision.

Quadratic probing leads to secondary clustering, when two keys do only have the same collision chain if their initial position is the same. As it turns out, secondary clustering prevents us from guaranteeing an insert if the table is greater than half full.

Secondary clustering is less severe in terms of performance hit than primary clustering. It is a process to keep clusters formation by using Quadratic Probing. The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

Example:

Suppose input keys are {89, 18, 49, 58, 69}, hash table size is 10, where, $\text{hashkey} = \text{key} \bmod \text{table-size}$

	Index	Value
89 % 10 = 9	[0]	49
18 % 10 = 8	[1]	
49 % 10 = 9, 1 attempt needed $1^2 = 1$ spot movement is required	[2]	58
	[3]	69
58 % 10 = 8, 2 attempt needed $2^2 = 4$ spots movement are required	[4]	
	[5]	
69 % 10 = 9, 2 attempt needed $2^2 = 4$ spots movement are required	[6]	
	[7]	
	[8]	18
	[9]	89

Figure 11.3: Linear Probing with cluster

Using quadratic probing there is a probability of secondary clustering when more than half of the hash table is filled up. That means elements that hash to the same hash key will always probe the same alternative cells.

Searching a key in a Hash Table using Quadratic Probing

Following algorithm describes the process of inserting a key in a hash table using a Quadratic Probing technique.

Algorithm of Searching in Quadratic Probing

Algorithm: INSERT_QUAD_PROB(HashTable, key, size)

[HashTable is an array represents a hash table, Key is a searching key and Size is the size of the HashTable]

1. Set HashIndex = Key mod size
2. If HashTable[HashIndex] = NULL then
 - a) Set HashTable[HashIndex] = Key
 - b) Set flag[HashIndex] = 1
 - c) Return
- else
 - a) Set count = 0
 - b) Set index = HashIndex
 - c) Repeat while HashTable[HashIndex] ≠ NULL
 - Set count = count + 1
 - Set HashIndex = HashIndex + count * count


```

    Set HashIndex = HashIndex mod size
[End of Loop]
d) If HashTable[HashIndex] = NULL then
    Set HashTable[HashIndex] = Key
    Set flag[HashIndex] = 1
[End of If]
[End of IF]
3. Return

```

Double Hashing

Double hashing is a popular collision resolution technique in open addressed hash table. It uses two hash functions:

- i) Like linear probing First hash function H is used as a starting address and
- ii) Second hash function H' is used to find out the interval to skip a variable amount. Suppose a record R with key K has the hash address $H(K) = h$ and $H'(K) = h' \neq m$, then we search the location with the address
 $h, h + h', h + 2h', h + 3h' \dots$

If m is a prime number, then the above sequence will access all the location in the table T.

Second hash function in double hashing has the following couple of requirements

- It must never evaluate to 0.
- Must make sure that all cells can be probed.

A popular hash function for double hashing is $\text{Hash2}(\text{key}) = R - (\text{key} \% R)$ where R is a prime number that is smaller than the size of the table.

Example:

Suppose input keys {89, 18, 49, 58, 69} and table size:10

Largest prime number less than 10 is 7	Index	Value
Hash1(key) = key % 10 and Hash2(key) = 7 - (key % 7)	[0]	69
Hash1(89) = 89 % 10 = 9	[1]	
Hash1(18) = 18 % 10 = 8	[2]	
Hash1(49) = 49 % 10 = 9, collision occur	[3]	58
Hash2(49) = 7 - (49 % 7)	[4]	
= 7 position move from [9]	[5]	
Hash1(58) = 58 % 10 = 8, collision occur	[6]	49
Hash2(58) = 7 - (58 % 7)	[7]	
= 5 position move from [8]	[8]	18
Hash1(69) = 69 % 10 = 9, collision occur	[9]	89
Hash2(69) = 7 - (69 % 7)		
= 1 position move from [9]		

Figure 11.4: Double Hashing

Advantage:

- It effectively eliminates clustering.
- It can allow the table to become nearly full.

Disadvantage

The main drawback of this method is in the implementation of deletion.

Insertion a key in a Hash Table using Double Probing

Following algorithm describes the process of inserting a key in a hash table using a Double Probing collision resolution technique.

Algorithm of Insertion in Double Probing

Algorithm: INSERT_DOUBLE_PROB(HashTable, Key, Size)
[HashTable is an array represents a hash table, Key is a searching key and Size is the size of the HashTable]

1. Set HashIndex = Key mod size
Set Flag[HashIndex] = false
2. If HashTable[HashIndex] = NULL then
 - a) Set HashTable[HashIndex] = Key
 - b) Set Flag[HashIndex] = true
- else
 - i) Set move = prime - (Key mod prime)
 - ii) set count = 1
 - iii) Repeat while HashTable[HashIndex] ≠ NULL
 - a) Set HashIndex = HashIndex + count * move
 - b) Set HashIndex = HashIndex mod size
 - c) set count=count+1
 - [End of Loop]
 - iv) Set HashTable[HashIndex] = Key
 - v) Set Flag[HashIndex] = true
 - [End of IF]
3. Return

Separate Chaining

In this technique, each bucket is independent and has the same sort of list of entries with the same key value. The time for hash table operation is the time to find the bucket, in addition, the time for the list operation.

Suppose, the hash table contains m slots. Each slot in the hash table contains a pointer to a linked list (separate chain) and the list stores the elements hashed to that slot. Hence, this method is known as Separate Chaining. A new key can be placed anywhere within a chain, as no ordering among the keys is used. As such, the new node is inserted at the end of a chain.

The hash table containing m pointers can be defined as follows:

```
node *Hash[m];
```

Example:

Suppose input keys are {20, 10, 1, 44, 64, 36, 30, 17, 46} and hash function $h(K) = K \bmod 10$.

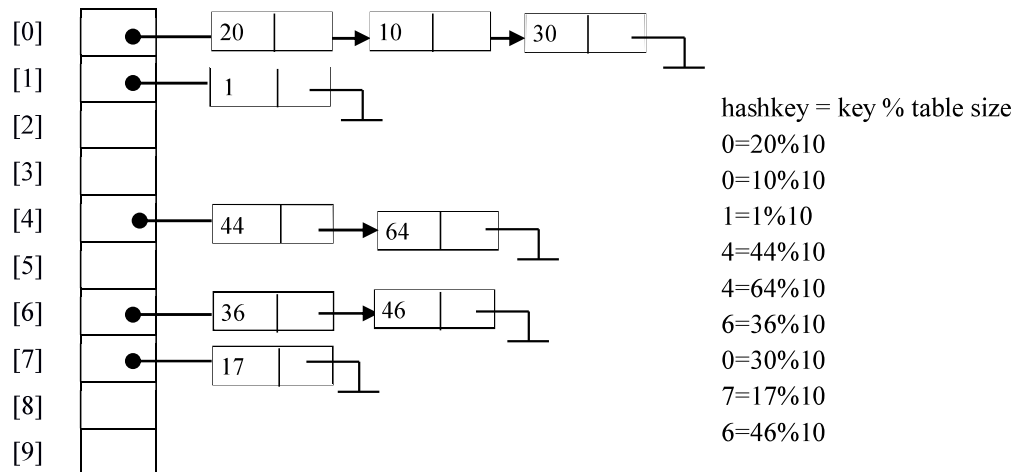


Figure 11.5: Separate Chaining

The main drawback to chaining is that one needs more memory spaces for the link field and there are m memory slots for the pointer array. An additional disadvantage is that traversing a linked list has poor cache representation.

- Cost is proportional to the length of the list: Cost in terms of time complexity is proportional to the length of the list.
- Worst case: If the hash function is not efficient then it generates same hash key that increases the length of one list.
- M too large: if the value of m is too large then the number of chains becomes a more and more chain will be empty.
- M too small: If the value of m is too small, that increase the length of the list.

Table 11.2: Type of Collision Resolution Technique

Feature	Linear Probing	Quadratic Probing	Double Hashing
Efficiency	Fastest among three.	Easy to implement and deploy	Make more efficient use of memory
Probe use	Use few probes.	Uses extra memory for links and it does not probe all locations in the table.	Use few probes, but take more time.
Demerit	Primary clustering is a limitation.	Secondary clustering is the limitation.	More complicated to implement.
Computing Interval	The interval between probes is fixed and it is 1.	The interval between probes increases proportionally to the hash value.	The interval between probes is computed using the second hash function.

Problems for which Hash Tables are not suitable are:

- The problem for which data ordering is required.
- Problems having multi-dimensional data.
- Prefix searching if keys are long and of variable length.
- The problem that has dynamic data.
- The problem in which data does not have any unique key.

Advantages of Hash Table

- It can efficiently handle a large volume of data.
- The speed of searching a data item increased as parallel searching can be applied here.
- Look up cost may be reduced by proper hash function, hash table size and internal data structure.

Disadvantages of Hash Table

- Hash table implementation is more difficult than search tree.
- Hash table does not allow duplicate keys.
- In some cases, the cost of a good hash function is more than that of the search tree.
- For a dynamic hash table cost of insertion and searching of the element in the hash table is more.
- If collision occurs very frequently in the hash table then the efficiency of hash table degrades.

Load Factor

The performance of hash table depends on two factors, initial capacity and load factor. Initial capacity is the number of cells in the hash table. A Load Factor (α) of a hash table is defined as the ratio of the number of elements (n) in the hash table to the table size ($tsize$).

$$\alpha = n / tsize$$

Load Factor is a measure which decides when exactly to increase the hash table capacity, so that get and put operation can still have $O(1)$ complexity.

For linear probing α tends to 1 ($0 \leq \alpha \leq 1$), that implies collision probability is higher in linear probing. Load Factor tends to zero (0) implies that the proportion of unused space on hash table is increasing. However, there is no necessary reduction in search cost. The result is wasted memory.

Generally, the default load factor (0.75) indicates better performance for a hash table. The expected number of entries in the hash table and its' load factor should be considered when the initial capacity of the hash table is defined. If the initial capacity is more than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

Summary

- Hashing is one of the most suitable techniques for storing and retrieving data when data volume is more than the storage volume.
- The choice of best hash function depends on the application.
- Collision resolution is one of the key feature of hashing process.

Exercises

1. What do you mean by hashing? What are the applications where you will prefer hash tables to other data structures?
2. Why do the hash functions need to be simple?