**Hash Function in Data Structure:**

There are almost 150 Zettabytes of data getting generated each day, which is equivalent to 150 trillion Gigabytes of data. With such an enormous speed of data growth, there emerges a need to store this data in an effective and efficient manner. By an effective and efficient manner of storage, we mean a way that will provide us the flexibility of retrieving the data in a minimal amount of time, because the more time required for an operation will directly increase the cost associated with that particular operation. So, in order to reduce the cost of an operation and do that task in an efficient manner, we need to reduce the retrieval time of the data for that particular task. And the solution for reducing the retrieval time is the Hash function or hash table. The hash function is used to map or bind the data to a particular hash value and then that hash value will be used as an index or a key to store that value in the hash table. The main benefit of storing data in the hash tables is that the retrieval time of the data stored in the hash tables is of unit time. That means the data that is stored in the hash table can be read with time complexity of O(1). So, in this way, the hash tables play an important role in drastically reducing the time required for reading the data from the hash tables. And for the working of the hash tables, it requires a hash function. Now let us see what is a hash function and how it works.

Hash Function can be defined as an algorithm or a function that is used to map or convert data of bigger size or length to a fixed or small index or hash value. In other words, a hash function can be defined as an algorithm that will be used to convert the data of higher length or size to data that is within a fixed range or size.

The input parameter that is passed to a hash function is the input data that needs to map to some hash data. And the output or result provided by a hash function depicts the hash value or the hashes that are associated with that input parameter value. The hash functions are associated with the hash tables that will actually store the data in the memory and the hash function is used only to map those values to the hash tables. The hash value returned by the hash function for the data item passed as an input parameter is then further used as an index to map or store that input data into the hash table. Or, we can say that the hash value returned by the hash function for the data item passed as an input parameter is used as a key for storing that data which will help in the easy and efficient retrieval of the stored data.

For an ideal hash function to work, it should satisfy two basic properties or conditions so that it can deliver optimal results within a specified span of computation period. These two basic properties or conditions for an efficient hash function to store data in the hash table are:

* Firstly, the hash function should be very fast in calculating and delivering the results. The speed is one of the main and crucial parameters that will affect the overall efficiency of a hash function. The relation between the speed of computation of the results and efficiency of a hash function is inversely proportional, which means more time required for computation and delivering the results lesser the efficiency of the hash function and vice-versa. So, ideally, it is required that the time span that is required for the calculations and delivering the results of a hash function should be as least as possible in order to maintain the efficiency of the hash function.
* Once the speed of creating the result is achieved, the next step is to deliver the results correctly and accurately. So, the second most important parameter is the accuracy of the results generated by the hash function. So, it is required that the result generated by the hash function should be unique and accurate because the hash values that are generated by the hash function acts as keys while sorting the data in the hash tables. So, the uniqueness of the hash value generated ensures that no two data should be mapped to the same key or hash value. That is why the accuracy and uniqueness of the result generated affects the efficiency of the whole hash function which in return affects the efficiency of storing data in the hash tables.

These are two main conditions that need to be satisfied by a hash function while generating the output data or hash values in order to maintain the efficiency of a hash function.

In conjunction with hash tables, hash functions are used to store and retrieve data items or data records. The hash function translates each datum or record associated with a key into a hash number which is used to index the hash table. When an item is to be added to the table, the hash code may index an empty slot (which is also called a bucket), in which case the item is added to the table there. The way input data is mapped to the indexes of the hash table using the hash function results in the creation of different types of hashing. In this article, we are going to see two main types of hashing that have their own benefits and drawbacks. The two main types of hashing types that we are going to understand are the chained hashing method and the open address hashing method.

In chained hashing, each slot that is present in the hash table acts as a head node for the input element that has got that index as a hash value output of the hash function. So, if that index's head node is empty then data is added at that head node otherwise if some data is already present at that index's head node, then the new incoming data is appended or added after that head node. In short, we can say that the indexes of the hash tables act as the head nodes of the linked list.

For instance, if we have ten indexes in a hash table starting from 0(zero) and ending at 9(nine). Then we have ten individual separate linked lists and the head nodes of all these ten different linked lists are stored at the indexes of this hash table. Then a hash function is used to map or store values in these different linked lists.

The major benefit of the chained hashing is that we can store any amount of data in this format. For storing a lot of data, we just need to add data or append data to the last existing object or data in the linked list corresponding to the index value or hash value returned for that data by the hash function. But storing more data in the chained hash tables reduces the data searching or data retrieval efficiency of the hash table. Because for instance if the linked list preset at the index 1 have n elements stored in it then the time required for the searching or retrieval of the last element of that linked list will be O(n) which is far greater than the time required for the searching or retrieval of the data in open address chaining of the hash tables.

In open addressing hash tables, the hash or the key value is calculated and then the input data is mapped or placed at the index value that is returned by the hash function. The major difference between the chained hashing method and the open addressing hashing method is that we can add any amount of data in the chained hashing technique but in the open addressing hashing technique the data added is equal to the number of indexes present in that hash table.

For instance, if we have ten indexes in a hash table starting from 0(zero) and ending at 9(nine). Then we can only store ten data in this type of hash table.

But one of the major benefits of the open addressing hash tables is that it requires constant time for the retrieval of data stored in the hash tables.

Other than these depending upon the computational logic, the hash function is used to create the resultant hash values, there are also different types of hash functions. Some of the major types of hash functions are:

1. Mid Square Hash Function
2. Division Hash Function
3. Folding Hash Function

Other than these hash functions mentioned above, the user can use any type of hash logic that the user wants to implement and create a hash function according to their needs.

**Let us understand the concept of hashing and the use of hash function in the whole process with the help of an example.**

Let us assume we have a hash table having ten slots or indexes starting from the index value or slot value zero(0). And the last slot value is nine(9). And the hash function we are using in this example is the modulus hash function, which means the input data that is passed as a parameter to the hash function will undergo the modulus operation as a part of the hashing and then the result of this modulus operation is returned as the output of the hash function that will act as an index or slot key to store or map that input data in the hash table.

Initially, the hash table looks like this. All the slots in the hash table will be empty.

|  |  |
| --- | --- |
| **Slot Number** | **Data in the slot** |
| Slot Number 0 | Empty |
| Slot Number 1 | Empty |
| Slot Number 2 | Empty |
| Slot Number 3 | Empty |
| Slot Number 4 | Empty |
| Slot Number 5 | Empty |
| Slot Number 6 | Empty |
| Slot Number 7 | Empty |
| Slot Number 8 | Empty |
| Slot Number 9 | Empty |

So, now let us assume the input data is 25. We pass this input data is passed to the hash function. In the hash function, the modulus operation is performed, the modulus of the 25 will be 5. So, the resultant value that will be returned by the hash function as the hash value for the input data 25 is 5. Thus, the input data having value 25 will be stored in the hash table slot number 5. The hash table after adding the data at slot number 5 is like:

|  |  |
| --- | --- |
| **Slot Number** | **Data in the slot** |
| Slot Number 0 | Empty |
| Slot Number 1 | Empty |
| Slot Number 2 | Empty |
| Slot Number 3 | Empty |
| Slot Number 4 | Empty |
| Slot Number 5 | 25 |
| Slot Number 6 | Empty |
| Slot Number 7 | Empty |
| Slot Number 8 | Empty |
| Slot Number 9 | Empty |

So, now let us assume the input data is 1. We pass this input data to the hash function. In the hash function, the modulus operation is performed, the modulus of the 1 will be 1. So, the resultant value that will be returned by the hash function as the hash value for the input data having value 1 will be stored in the hash table slot number 1. The hash table after adding the data at slot number 1 is like this:

|  |  |
| --- | --- |
| **Slot Number** | **Data in the slot** |
| Slot Number 0 | 1 |
| Slot Number 1 | Empty |
| Slot Number 2 | 493 |
| Slot Number 3 | Empty |
| Slot Number 4 | Empty |
| Slot Number 5 | 25 |
| Slot Number 6 | Empty |
| Slot Number 7 | Empty |
| Slot Number 8 | 68 |
| Slot Number 9 | Empty |

So, this is how the final hash table looks like after adding data to the table four times.

Now let us assume the input data is 975. We pass this input data to the hash function. In the hash function, the modulus operation is performed, the modulus of the 975 will be 5. So, the resultant value that will be returned by the hash function as the hash value for the input data 975 is 5. Thus, the input data having value 975 will be stored in the hash table slot number 5. But slot number 5 is already occupied by the data having a value of 25. So, this is the constraint of the open-chain hashing technique that we can store only some specific amount of data in the hash table.

Once the storage is done, the next step is to retrieve the data from the hash table. For the search operation, the same hash function is used to find the data stored in the hash table. In search operation, the search key is again passed to the hash function and the slot number or index is calculated and the data from that index is retrieved and it is matched with the search key. If the search key and the data fetched from the hash table match then the search operation is successful otherwise if both the data don't match the search operation is unsuccessful.

For the actual real-life application of the hashing and hash tables, it is needed that we should implement the concept of hash tables with the use of hash function programmatically. So, for better understanding how we can code a hash table with the use of hash functions let us write a sample java code that will simulate the functioning of a hash table programmatically.

|  |
| --- |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

Code:

1. // a public class named HashEntry is created that will consist of the most basic Functionalities of the Hash Table that we will create
2. **public** **class** HashEntry {
3. // The class HashEntry has two private class variables named key and value that will represent the actual key and value stored in the hash Table
4. **private** **int** key;
5. **private** **int** value;
6. // a parameterized constructor with two parameters that will be used to initialize the class variables
7. HashEntry(**int** key, **int** value) {
8. **this**.key = key;
9. **this**.value = value;
10. }
11. // a public Function named getValue to retrieve the value from the object of the HashEntry class is written
12. **public** **int** getValue() {
13. **return** value;
14. }
15. // a public Function named setValue to set value to the object of the HashEntry class is written
16. **public** **void** setValue(**int** value) {
17. **this**.value = value;
18. }
19. a **public** Function named getKey to retrieve the key from the object of the HashEntry **class** is written
20. **public** **int** getKey() {
21. **return** key;
22. }
23. } //end of the HashEntry class
24. // a public class named DeletedEntry is created that will inherit the base class HashEntry that will have all the Functionalities needed to delete data from the Hash Table
25. **public** **class** DeletedEntry **extends** HashEntry {

28. // a private static variable named entry is created
29. **private** **static** DeletedEntry entry = **null**;

32. // a private default constructor is created that will call the parent constructor with the use of super() Function
33. **private** DeletedEntry() {
35. **super**(-1, -1);
37. }

40. // logic for finding the unique entry that needs to be deleted
41. **public** **static** DeletedEntry getUniqueDeletedEntry() {
43. **if** (entry == **null**)
45. entry = **new** DeletedEntry();
47. **return** entry;
49. }
51. }// end of DeletedEntry class


55. // Scanner class is imported to use input Functionality
56. **import** java.util.Scanner;


60. // a public class named HashMap is created that will consist of all the calling Functions
61. **public** **class** HashMap {

64. // a private int variable named TABLE\_SIZE is created that will represent the size of the Hash Table
65. **private** **final** **static** **int** TABLE\_SIZE = 128;

68. // A static array of the HashEntry objects is created that will act as a Hash Table
69. **static** HashEntry[] table;

72. // A default constructor is created to initialize the Hash Table
73. HashMap() {
75. table = **new** HashEntry[TABLE\_SIZE];
77. **for** (**int** i = 0; i < TABLE\_SIZE; i++)
79. table[i] = **null**;
81. }

84. // a public Function named get() with one parameter  is created that will be used to find data in the hash Table
85. **public** **int** get(**int** key) {
87. **int** hash = (key % TABLE\_SIZE);
89. **int** initialHash = -1;
91. **while** (hash != initialHash
93. && (table[hash] == DeletedEntry.getUniqueDeletedEntry() || table[hash] != **null**
95. && table[hash].getKey() != key)) {
97. **if** (initialHash == -1)
99. initialHash = hash;
101. hash = (hash + 1) % TABLE\_SIZE;
103. }
105. **if** (table[hash] == **null** || hash == initialHash)
106. // if key is not found in the hash Table then -1 is returned
107. **return** -1;
109. **else**
110. // if the search key is present in the hash Table then the value associated with that key is returned
111. **return** table[hash].getValue();
113. }

116. // a public Function named put() with two parameters is created to add new data in the hash Table
117. **public** **void** put(**int** key, **int** value) {

120. // hash value is calculated by modulus operation
121. **int** hash = (key % TABLE\_SIZE);
123. **int** initialHash = -1;
125. **int** indexOfDeletedEntry = -1;
127. **while** (hash != initialHash
129. && (table[hash] == DeletedEntry.getUniqueDeletedEntry() || table[hash] != **null**
131. && table[hash].getKey() != key)) {
133. **if** (initialHash == -1)
135. initialHash = hash;
137. **if** (table[hash] == DeletedEntry.getUniqueDeletedEntry())
139. indexOfDeletedEntry = hash;
141. hash = (hash + 1) % TABLE\_SIZE;
143. }
145. **if** ((table[hash] == **null** || hash == initialHash)
147. && indexOfDeletedEntry != -1)
149. table[indexOfDeletedEntry] = **new** HashEntry(key, value);
151. **else** **if** (initialHash != hash)
153. **if** (table[hash] != DeletedEntry.getUniqueDeletedEntry()
155. && table[hash] != **null** && table[hash].getKey() == key)
157. table[hash].setValue(value);
159. **else**
161. table[hash] = **new** HashEntry(key, value);
163. }