Hashing

71 Hash-function Hashing CodeMonk

Introduction:

To store the key/value pair, we can use simple array like data structure where keys directly can be used as index to store values, but in case when keys are large and can't be directly used as index to store value, we can use technique of **Hashing**.
In hashing, large keys are converted into small ones, by using **hash functions** and then the values are stored in data structure called **hash tables**. The idea of hashing is to distribute the entries (key / value pairs) across an array uniformly. Each element is assigned a key(converted one) and using that key we can access the element in **O(1)** time. Given a key, the algorithm(hash function) computes an index that suggests where the entry can be found or inserted.
Hashing is often implemented in two steps:

1. The element is converted into an integer, using the Hash Function, and can be used as an index to store original element, which falls into the hash table.
2. The element is stored in the hash table, where it can be quickly retrieved using hashed key.

hash = hashfunc(key)
index = hash % array_size
In this method, the hash is independent of the array size, and it is then reduced to an index (a number between 0 and array_size − 1) using the modulo operator (%).

Hash Function
A hash function is any function that can be used to map dataset of arbitrary size to dataset of fixed size which falls into the hash table. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. In order to achieve a good Hashing mechanism, it is essential to have a Good Hash Function.
A *Good Hash Function* has 3 basic requirements:

1. **Easy to Compute** – A Hash function should be easy to compute. It should not become an algorithm in itself.

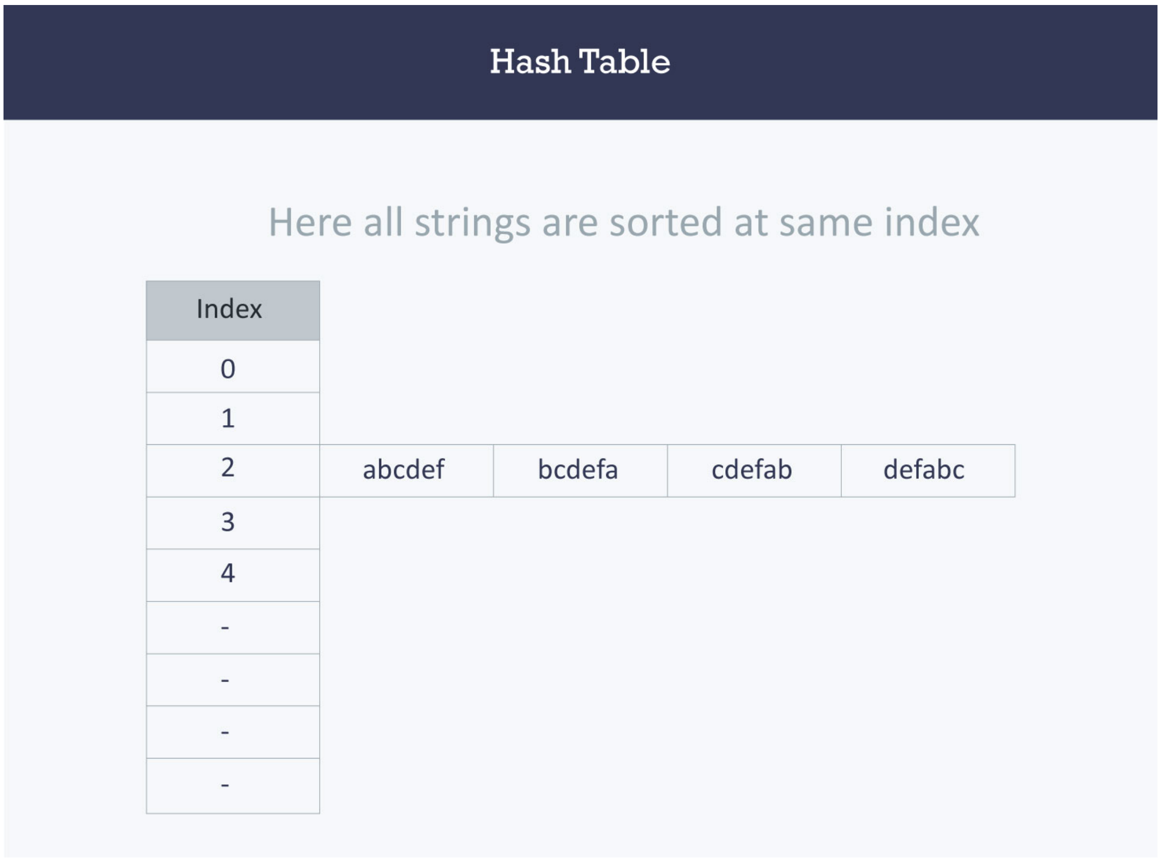
2. **Uniform Distribution** – It should provide an uniform distribution across the Hash Table and should not result in Clustering.

3. **Less Collisions** – Collision occurs when pairs of elements are mapped to the same hash value. It should avoid collisions as far as possible.

Note: No matter how good our Hash Function is, collisions are bound to occur. So, we need to manage them through various Collision Resolution Techniques in order to maintain the performance of our Hash Table.

Let's understand the need of good hash function:
Example: Let's say we have to store strings in the hash table using hashing.
{"abcdef", "bcdefa", "cdefab" , "defabc" }
To compute the index for storing the strings, we will use hash function which states that:
The index for a particular string, will be equal to the sum of the ascii values of the characters modulo 599.
As 599 is prime number, it will reduce the possibility of same index of different strings (collisions). It is recommended to use prime number in case of modulo.
As we know ascii value of a = 97, b = 98, c= 99, d = 100, e = 101, f = 102.
As all the strings contains same characters with different permutations, so the sum for will be same, i.e 599.
So the hash function will compute same index for all the strings, and strings will be stored in the hash table in the given format below. As the index of all the strings is same, so we can create a list on that particular

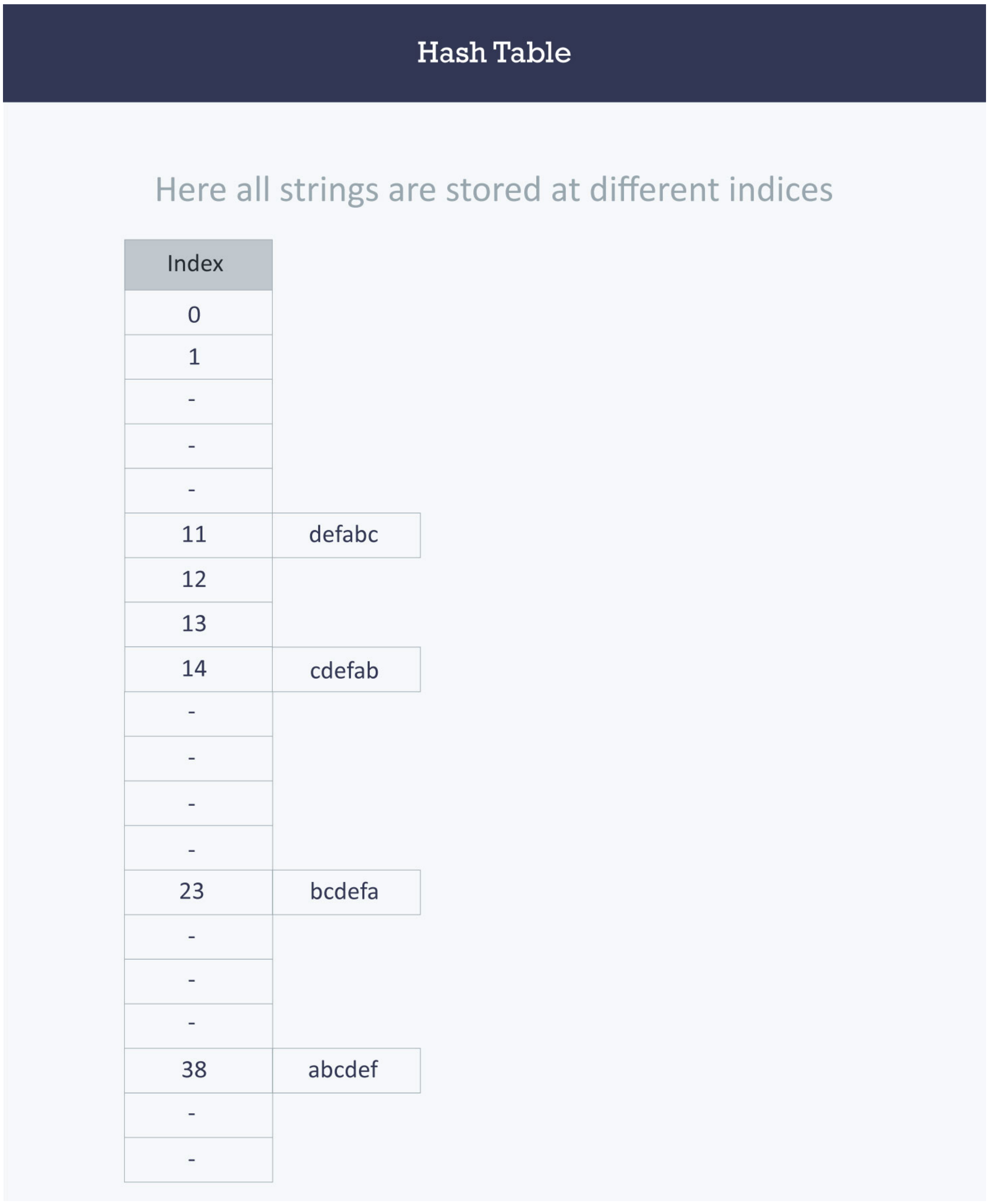
index and can insert all the strings in that list.



So to access a particular string, it will take O(n) time where n is the number of strings, which shows that the hash function is not a good hash function.

Let's try different hash function: The index for a particular string, will be equal to sum of ascii values of character multiplied by their respective order in the string and then modulo it with 2069 (prime number).

String	Hash function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11



Hash Table
Hash Table is a data structure used to store key / value pairs. Hash table uses a hash function to compute an index into an array where the element will be inserted or searched. By using a good hash function, hashing can perform extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is **O(1)**. Let us consider string S. We need to count the frequency of all the characters in the string S.

```
string S = “ababcd”
```

Simplest thing we can do it to iterate over all the possible characters and count their frequency one by one. The time complexity of this approach is **O(26*N)** where **N** is the size of the string and there are 26 possible characters.

```
void countFre(string S)
{
    for(char c = 'a';c <= 'z';++c)
    {
        int frequency = 0;
        for(int i = 0;i < S.length();++i)
            if(S[i] == c)
                frequency++;
        cout << c << ' ' << frequency << endl;
    }
}
```

Output:

```
a 2
b 2
c 1
d 1
e 0
f 0
…
z 0
```

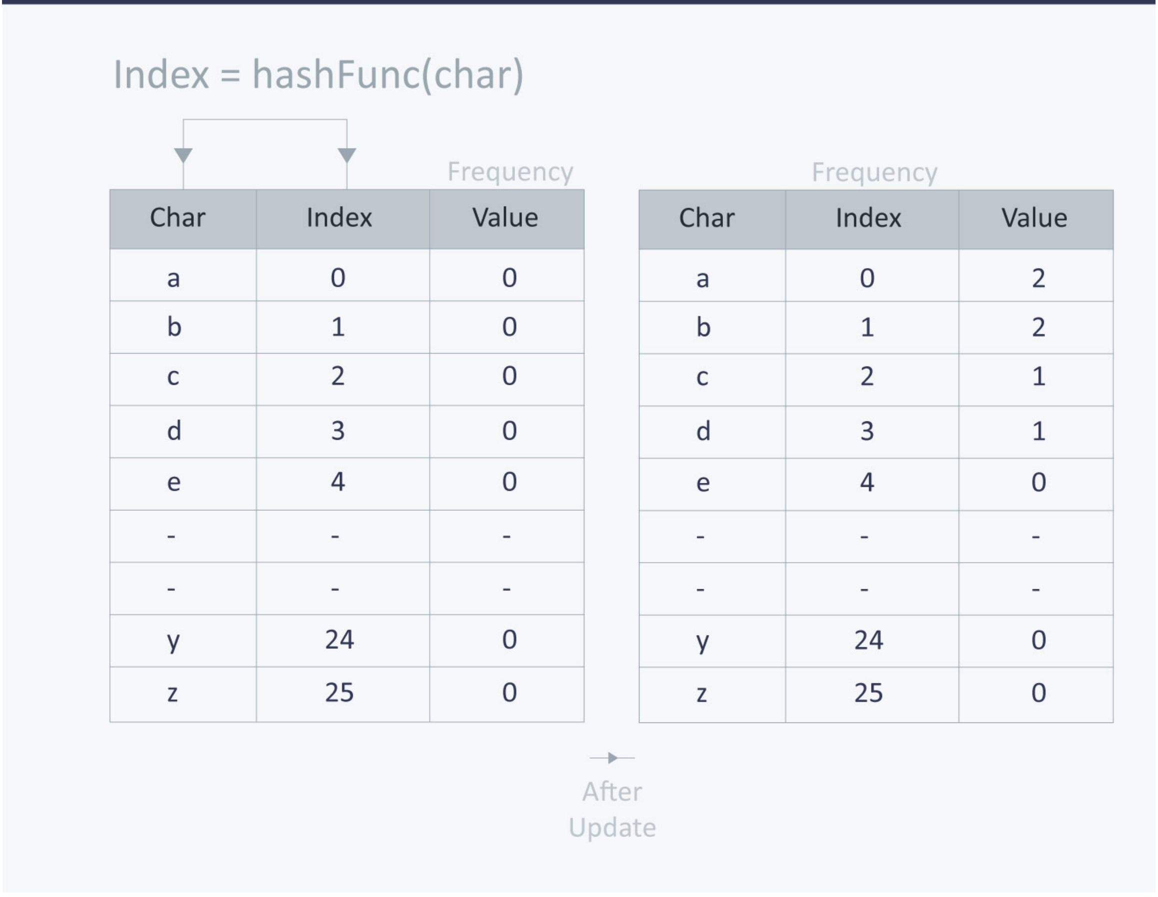
Let's apply hashing in this problem. We take an array Frequency of size 26 and hash the 26 characters with indices of the array using the Hash Function. Now, we will simply iterate over the string and for each character, we will increase the value in the Frequency at the corresponding index. The complexity of this approach is **O(N)** where **N** is the size of the string.

```
int Frequency[26];

int hashFunc(char c)
{
    return (c - 'a');
}

void countFre(string S)
{
    for(int i = 0;i < S.length();++i)
    {
        int index = hashFunc(S[i]);
        Frequency[index]++;
    }
    for(int i = 0;i < 26;++i)
        cout << (char)(i+'a') << ' ' << Frequency[i] << endl;
}
```

Output:

```
a 2
b 2
c 1
d 1
e 0
f 0
…
z 0
```

Hash Table



Char	Index	Value
a	0	0
b	1	0
c	2	0
d	3	0
e	4	0
-	-	-
-	-	-
y	24	0
z	25	0

Char	Index	Value
a	0	2
b	1	2
c	2	1
d	3	1
e	4	0
-	-	-
-	-	-
y	24	0
z	25	0

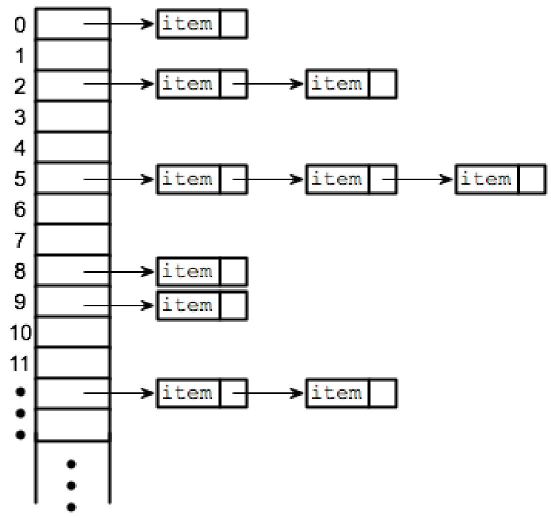
Load Factor

Load factor is the number of entries divided by the size of the hash table i.e., **n / k** where **n** is the number of entries and **k** is the size of the hash table. If the load factor is kept reasonable, the hash table will perform well, if we are using a good hash function. If the load factor grows too large, the hash table will become slow, or it may fail to work (depending on the hash function used). The expected constant time property of a hash table assumes that the load factor is kept below some bound. A low load factor is not especially beneficial. As the load factor approaches 0, the proportion of unused areas in the hash table increases. This will result in wasted memory.

Collision Resolution Techniques

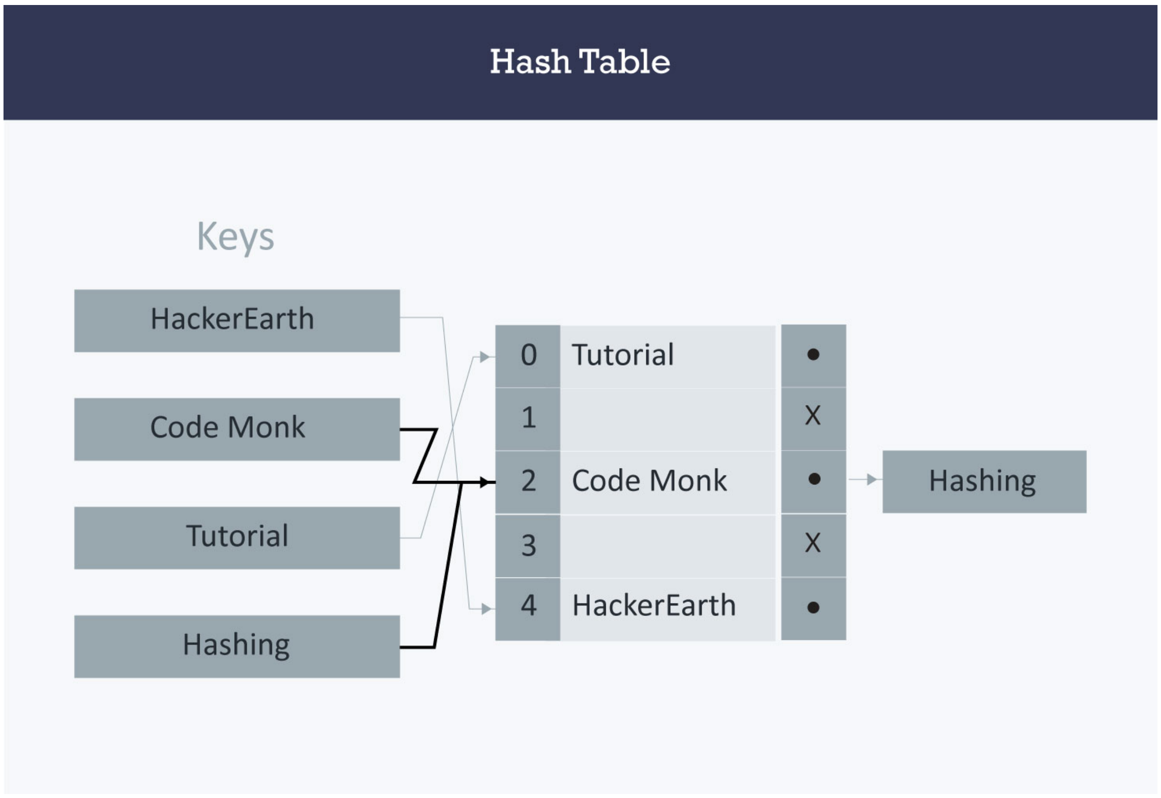
- **Separate Chaining (Open Hashing)**

Separate Chaining is one of the most common and widely 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 we will simply insert it into the particular linked list. If there is any collision, i.e two different elements have same hash value, then we will simply store both the elements in same linked list.



The cost of a lookup is that of scanning the entries of the selected linked list for the desired key. If the distribution of keys is sufficiently uniform, the average cost of a lookup depends only on the average number of keys per linked list - i.e., it is roughly proportional to the load factor. For this reason, chained hash tables remain effective even when the number of table entries n is much higher than the number of slots.

For separate-chaining, the worst-case scenario is when all entries are inserted into the same linked list. The lookup procedure may have to scan all its entries, so the worst-case cost is proportional to the number n of entries in the table.



As shown in the above diagram, **CodeMonk** and **Hashing** both hash to the value **2**. Now, the linked list at the index **2** can hold only one entry so the next entry, in this case **Hashing** is linked (attached) to the entry of **CodeMonk**.

Implementation of Hash Table with Separate Chaining (Open Hashing)

Assumption: Hash Function will return an integer from 0 to 19.

```
vector <string> hashTable[20];
int hashTableSize=20;
```

Insert:

```
void insert(string s)
{
            // Compute the index using Hash Function
    int index = hashFunc(s);
    // Insert the element in the linked list at the particular index
    hashTable[index].push_back(s);
}
```

Search:

```
void search(string s)
{
    // Compute the index using Hash Function
    int index = hashFunc(s);
    // Search the linked list at that particular index
    for(int i = 0;i < hashTable[index].size();i++)
    {
        if(hashTable[index][i] == s)
        {
            cout << s << " is found!" << endl;
            return;
        }
    }
    cout << s << " is not found!" << endl;
}
```

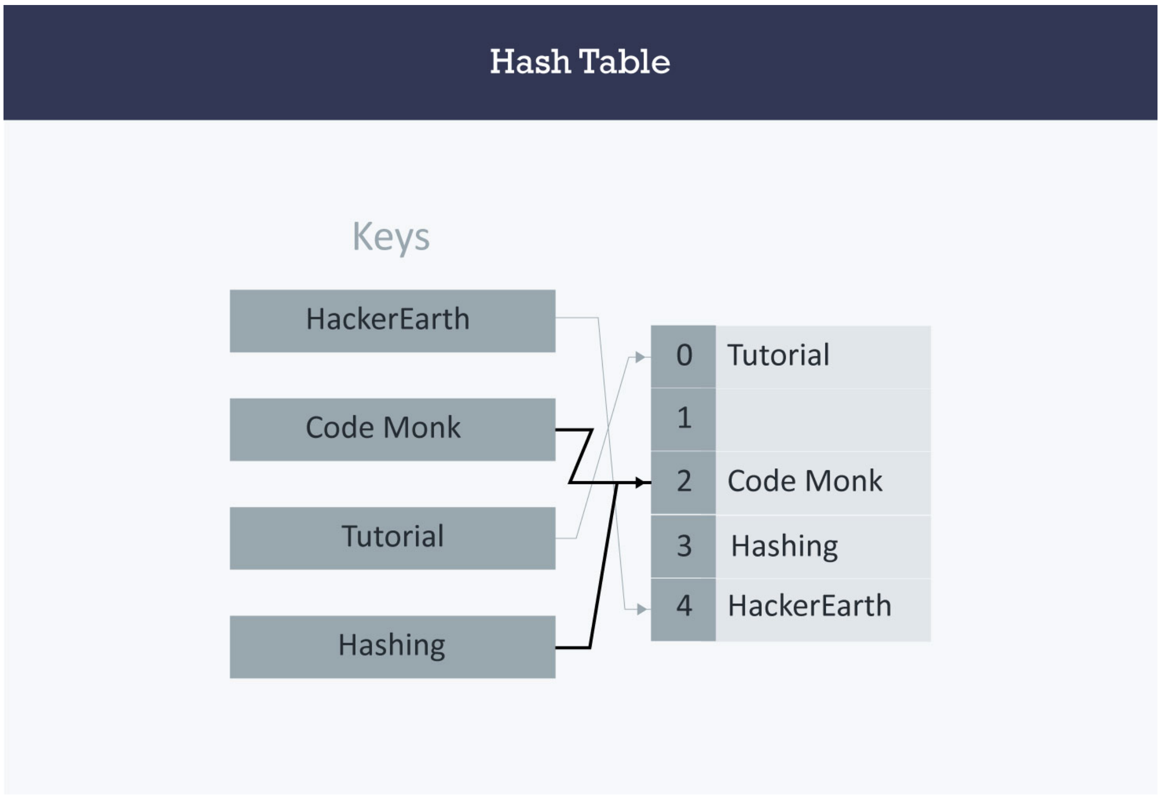
- **Linear Probing - (Open Addressing or Closed Hashing)**

In open addressing, all entry records are stored in the array itself, instead of linked lists. When a new entry has to be inserted, we compute the hashed index of hash value and then the array is examined, starting with the hashed index. If the slot at hashed index is unoccupied, then the entry record is inserted in slot at hashed index otherwise it will be proceed in some probe sequence, until an unoccupied slot is found. Probe sequence is the sequence which we follow while traversing through the entries. We can have different interval between successive entry slots or probes in different probe sequence. When searching for an entry, the array is scanned in the same sequence, until either the target element is found, or an unused slot is found, which indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location ("address") of the item is not determined by its hash value. Linear Probing is when the interval between successive probes is fixed (usually to 1).

Let’s say, hashed index for a particular entry is **index**. Then probing sequence for linear probing is :

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

and so on…



Hash collision resolved by open addressing with linear probing. Since **CodeMonk** and **Hashing** are hashed to the same index i.e. **2**. So we will store **Hashing** at **3** as the interval between successive probes is **1**.

Implementation of Hash Table with Linear Probing
Assumption: There are no more than 20 elements in the data set and Hash Function will return an integer from 0 to 19. The data set must have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
```

Insert:

```
void insert(string s)
{
    // Compute the index using the Hash Function
    int index = hashFunc(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != "")
        index = (index + 1) % hashTableSize;
    hashTable[index] = s;
}
```

Search:

```
void search(string s)
{
    // Compute the index using the Hash Function
    int index = hashFunc(s);
     // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != s and hashTable[index] != "")
        index = (index + 1) % hashTableSize;
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

- **Quadratic Probing**

Quadratic Probing is same as Linear Probing, the only difference being the interval between successive probes or entry slots. Here, when at hashed index for an entry record, the slot is already occupied then we start traversing until we do not get an unoccupied slot and the interval between slots is computed by adding the successive value of arbitrary polynomial in the original hashed index. Let's say, hashed index for an entry is **index**, and at **index** there is occupied slot, so probe sequence can be:

index = index % hashTableSize

index = (index + 1^2) % hashTableSize

index = (index + 2^2) % hashTableSize

index = (index + 3^2) % hashTableSize

and so on…

Implementation of Hash Table with Quadratic Probing
Assumption: There are no more than 20 elements in the data set and Hash Function will return an integer from 0 to 19. The data set must have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
```

Insert:

```
void insert(string s)
{
    // Compute the index using the Hash Function
    int index = hashFunc(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    int h = 1;
    while(hashTable[index] != "")
    {
        index = (index + h*h) % hashTableSize;
             h++;
    }
    hashTable[index] = s;
}
```

Search:

```
void search(string s)
{
    // Compute the index using the Hash Function
    int index = hashFunc(s);
     // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
   int h = 1;
    while(hashTable[index] != s and hashTable[index] != "")
    {
        index = (index + h*h) % hashTableSize;
             h++;
    }
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

- **Double Hashing**

Double Hashing is same as Linear Probing, the only difference being the interval between successive probes. Here, the interval between probes is computed by using two hash functions. Let's say hashed index for an entry record is index which is computed by one hashing function, and at index the slot is already occupied, then we start traversing in a particular probing sequence to look for unoccupied slot and probing sequence will be:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

and so on... here **indexH** is the hash value computed by another hash function.

Implementation of Hash Table with Double Hashing
Assumption: There are no more than 20 elements in the data set and Hash Functions will return an integer from 0 to 19. The data set must have unique elements.

```
string hashTable[21];
int hashTableSize = 21;
```

Insert:

```
void insert(string s)
{
    // Compute the index using the Hash Function1
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
    // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != "")
        index = (index + indexH) % hashTableSize;
    hashTable[index] = s;
}
```

Search:

```
void search(string s)
{
    // Compute the index using the Hash Function
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
     // Search for an unused slot and if the index will exceed the hashTableSize
    // we will roll back
    while(hashTable[index] != s and hashTable[index] != "")
        index = (index + indexH) % hashTableSize;
    // Element is present in the Hash Table or not
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

Applications

- **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).
- **Database Indexing:** Hash tables may also be used as disk-based data structures and database indices (such as in dbm).
- **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.
- **Object Representation:** Several dynamic languages, such as Perl, Python, JavaScript, and Ruby, use hash tables to implement objects.
- Hash Functions are used in various algorithms to make their computing faster/

Practice Problems:
1. https://www.hackerearth.com/problem/algorithm/vada-pav-list-3/description/
2. https://www.hackerearth.com/problem/algorithm/substrings-count-3/description/
3. https://www.hackerearth.com/problem/algorithm/akash-and-the-assignment-1-12/description/

Solve Problems

Like 91 Tweet G+1 3

AUTHOR

Akash Sharma
Problem Setter and Tester ...
Dehradun
7 notes

TRENDING NOTES

Sorting And Searching Algorithms - Time Complexities Cheat Sheet
written by Vipin Khushu

CodeMonk Dynamic Programming II
written by Tanmay Sahay

Efficient Factorials Calculation !
written by Ankur Anand

Trie, Suffix Tree, Suffix Array
written by pkacprzak

Technique to play online Bot games
written by Catalin Stefan Tiseanu

more ...

ABOUT US

Blog
Engineering Blog
Updates & Releases
Team
Careers
In the Press

HACKEREARTH

API
Chrome Extension
CodeTable
HackerEarth Academy
Developer Profile
Resume
Campus Ambassadors
Get Me Hired
Privacy
Terms of Service

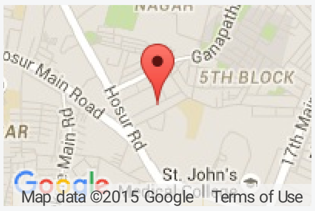
DEVELOPERS

AMA
Code Monk
Judge Environment
Solution Guide
Problem Setter Guide
Practice Problems
HackerEarth Challenges
College Challenges
College Ranking

RECRUIT

Developer Sourcing
Lateral Hiring
Campus Hiring
FAQs
Customers
Annual Report

REACH US



IIIrd Floor, Salarpuria Business Center,
4th B Cross Road, 5th A Block,
Koramangala Industrial Layout,
Bangalore, Karnataka 560095, India.

contact@hackerearth.com
+91-80-4155-4695
+1-650-461-4192