Hashing

Hashing is a technique that is used to uniquely identify a specific object from a group of similar objects. Some examples of how hashing is used in our lives include:

- In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
- In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value. It is also used in many encryption algorithms.

Assume that you have an object and you want to assign a key to it to make searching easy. To store the key/value pair, you can use a simple array like a data structure where keys (integers) can be used directly as an index to store values. However, in cases where the keys are large and cannot be used directly as an index, you should use ***hashing*.**

In hashing, large keys are converted into small keys by using **hash functions**. The values are then stored in a data structure called **hash table**. Hash Tables are also commonly known as **Hash Maps**.

The idea of hashing is to distribute entries (key/value pairs) uniformly across an array. Each element is assigned a key (converted key). By using that key you can access the element in **O(1)** time. Using the key, the algorithm (hash function) computes an index that suggests where an entry can be found or inserted. Worst case is O(n)

There are different Searching Algorithms such as **Linear Search** and **Binary Search** in which the search time is dependent on the Number of Elements. In Hash Tables, less key comparisons are made which thereby helps to perform search operation in a Constant Time. Therefore, the Search Time is not dependent on the Number of Elements.

Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This element can be used as an index to store the original element, which falls into the hash table.

```
hash = hashFnc(key);
```

2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

```
index = hash % hashTableSize
```

In this method, the hash is independent of the hash table size and it is then reduced to an index (a number between 0 and hashTableSize − 1) by using the modulus operator (%).

Hash function

- A hash function is any function that can be used to map a data set of an arbitrary size to a data set of a 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.

- Hashing functions are one-way functions (trap-door functions)

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.

Note: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Collision resolution techniques - *Linear probing (open addressing or closed hashing)*

In open addressing, 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. Otherwise, array by looking into the next cell until an empty cell is found. This technique is called linear probing.

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. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

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…
```

Hash collision is resolved by open addressing with linear probing.

Data Item

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

For the following example the key is student's id number and data has student's name. The size of the array defined as 10. Constants are also defined to indicate whether the location "empty, deleted or occupied".

```
// capacity of the Hash Table
#define SIZE    10

// constants for status field
#define EMPTY    0
#define OCCUPIED 1
#define DELETED  2

// Example's Data Structures
typedef struct {
   int   stdID;
   char  stdName[20];
} std_t;

//Hash Data Structures
typedef struct {
   std_t info;
   int   status;
} hash_t;
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashFnc(int key) {
   return (key % SIZE);
}
```

Initialize Hash Table

Store EMPTY to all elements of the hash table status field.o

```
void initHashTable(hash_t hashTable[], int hashSize) {
   for (int i = 0; i < hashSize; i++)
      hashTable[i].status = EMPTY;
}
```

Basic Operations

The basic primary operations of a hash table:

- **Search** − Searches an element in a hash table.
- **Insert** − inserts an element in a hash table.
- **Delete** − Deletes an element from a hash table.

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
int searchInHash(int key, hash_t hashTable[], int hashSize) {
   int position, start, offset;
   start = hashFnc(key);
   for (offset = 0; offset < hashSize; offset++) {
      position = (start + offset) % hashSize;
      if (hashTable[position].status == EMPTY)
         return -1; // not found
      if (hashTable[position].status == OCCUPIED &&
            hashTable[position].info.stdID == key){
         printf("\nSearched item is found in %d trial.\n", (offset + 1));
         return position; // found: position is returned
      }
   }
   return -1; // not found
}
```

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code. The location will be marked in some special way (“OCCUPIED”).

```
The function returns error code:
      1: record inserted into hash table successfully
      2: duplicate record cannot be inserted
      3: hash table limit exceeded!


int insertIntoHash(std_t *std, hash_t hashTable[], int hashSize) {
    int position, start, offset;
    if (searchInHash(competitor->cId, hashTable, hashSize) != -1)
        return 2; // duplicate record cannot be inserted
    else {
        start = hashFnc(competitor->cId);
        for (offset = 0; offset < hashSize; offset++) {
           position = (start + offset) % hashSize;
           if (hashTable[position].status != OCCUPIED) {
             hashTable[position].info = *competitor;
             hashTable[position].status = OCCUPIED;
             return 1; // record inserted into hash table
           }
        }
        return 3; // hash table limit exceeded
   }
}
```

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code.

The location must not be left as an ordinary “empty spot”. The location must be marked in some special way (“DELETED”) so that a search can tell that the spot used to have something in it.

```
int deleteFromHash(int key, hash_t hashTable[], int hashSize) {
   int position = searchInHash(key, hashTable, hashSize);
   if (position == -1)
      return 0; // id not found
   hashTable[position].status = DELETED;
   return 1; // deleted from hash table successfully
}
```