Data Structures Lab 11

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif

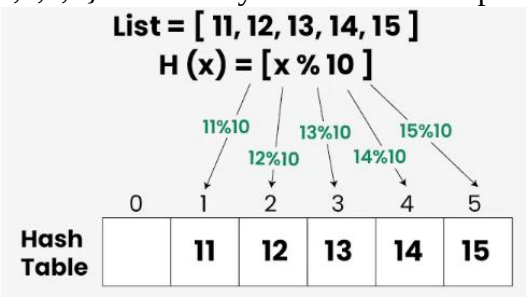
Note:

- Maintain discipline during the lab.
- Topics: **{Hash Table, Insertion, Searching, Deletion, Collision, Open and Closed Hashing}**
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Hashing Data Structure:

Hashing is a technique used in data structures that efficiently **stores and retrieves** data in a way that allows for quick access. It involves **mapping data** to a specific index in a hash table using a **hash function** that enables fast retrieval of information based on its key. The **efficiency of mapping** depends on the efficiency of the **hash function** used. This method is commonly used in databases, **c**aching systems, and various programming applications to optimize search and retrieval operations. The great thing about hashing is, we can achieve all three operations (search, insert and delete) in O(1) time on average.

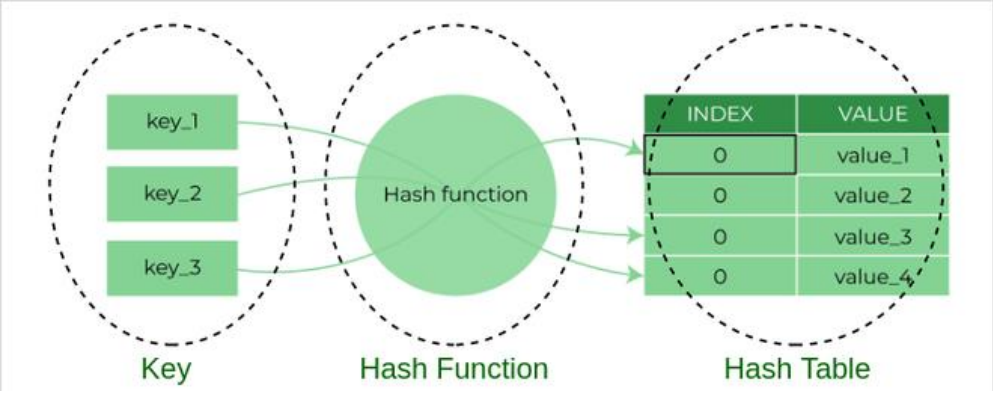
Let a hash function h(x) map the value x at the index **x%10** in an Array. For example, if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.



Components of Hashing:

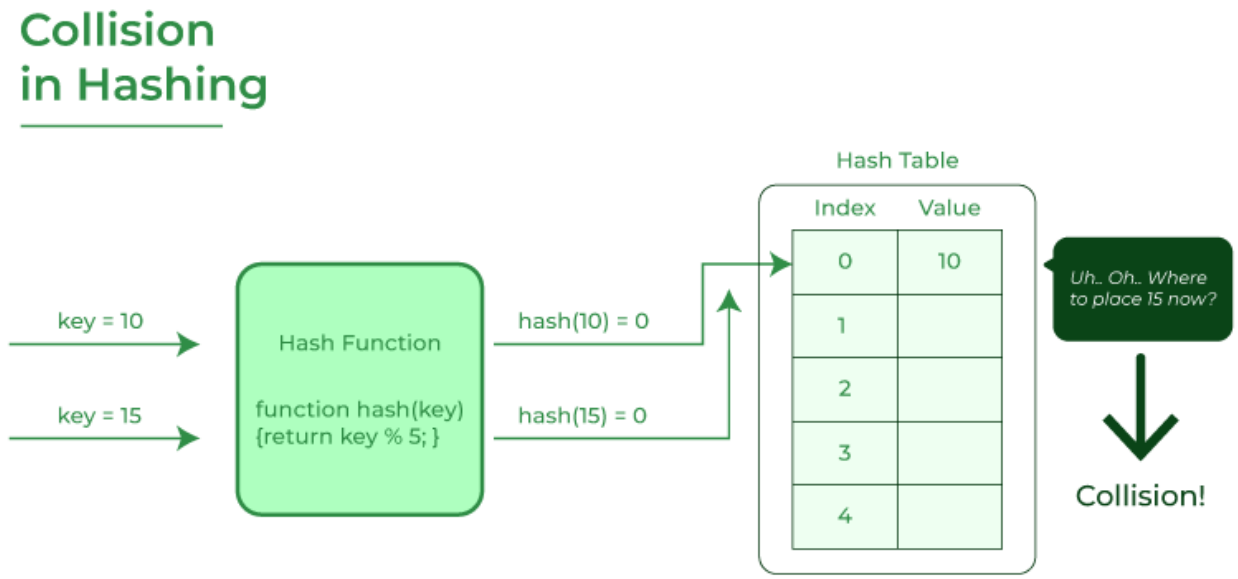
There are majorly three components of hashing:

1. **Key:** A Key can be anything string or integer which is fed as input in the hash function the technique that determines an index or location for storage of an item in a data structure.
2. **Hash Function:** The hash function receives the input key and returns the index of an element in an array called a hash table. The index is known as the hash index.
3. **Hash Table:** Hash table is a data structure that maps keys to values using a special function called a hash function. Hash stores the data in an associative manner in an array where each data value has its own unique index.



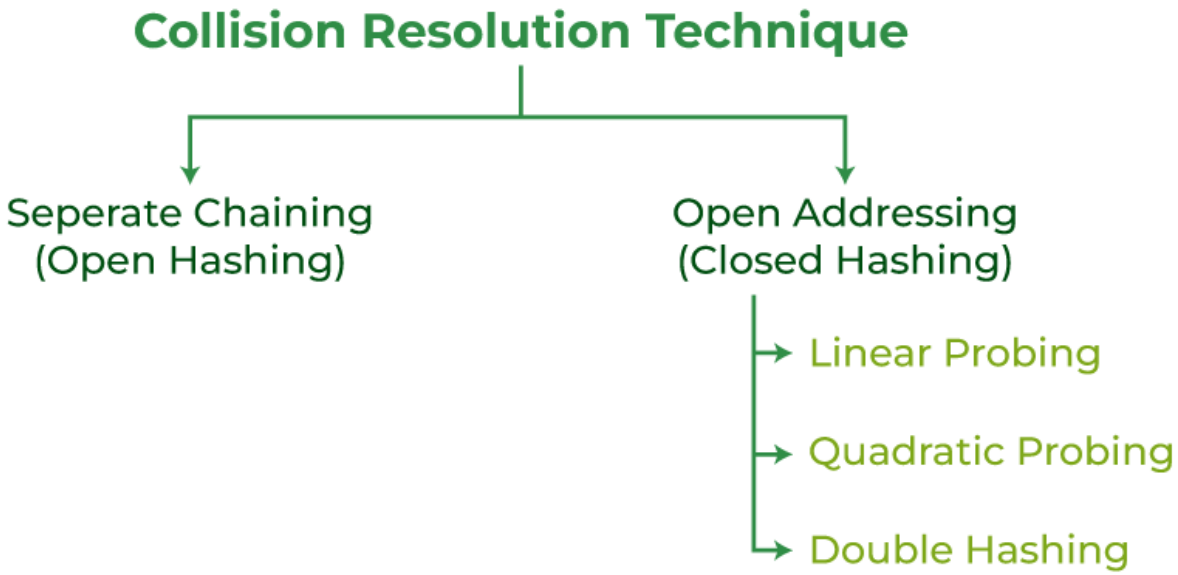
Collision in Hashing:

The hashing process generates a small number for a big key, so there is a possibility that **two keys could produce the same value**. The situation where the newly inserted key maps to an already occupied, and it must be handled using some collision handling technology.



Methods of handling Collisions:

There are mainly two methods to handle collision:



Load Factor in Hashing:

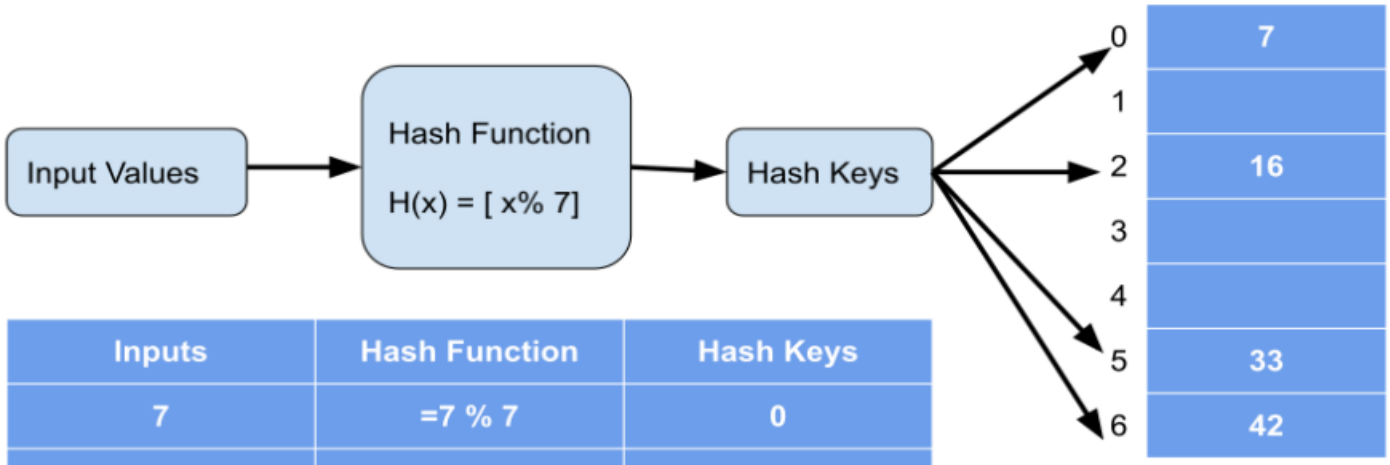
The load factor of the hash table can be defined as the number of items the hash table contains divided by the size of the hash table. The load factor is the decisive parameter that is used when we **want to rehash** the previous hash function or **want to add more elements** to the existing hash table.

It helps us in **determining the efficiency** of the hash function i.e. it tells whether the hash function which we are using is **distributing the keys uniformly or not** in the hash table. The concept of a load factor is applicable to all types of open-addressing collision resolution techniques, including linear probing, quadratic probing, and double hashing.

Load Factor (λ) = Total elements in hash table / Size of hash table

Rehashing in Hashing Data Structure:

As the name suggests, rehashing means hashing again. Basically, when the **load factor increases to more than its predefined value** (the default value of the load factor is 0.75), the **complexity increases**. So, to overcome this, the size of the **array is increased** (doubled) and all the values are hashed again and stored in the new double-sized array to maintain a low load factor and low complexity.



Inputs	Hash Function	Hash Keys
7	=7 % 7	0
16	=16 % 7	2
33	=33 % 7	5
42	=42 % 7	6

How does Hashing work?

Suppose we have a set of strings {“ab”, “cd”, “efg”} and we would like to store it in a table. Our main objective here is to search or update the values stored in the table quickly in O(1) time and we are not concerned about the ordering of strings in the table. So, the given set of **strings can act as a key** and the **string itself will act as the value of the string** but how to store the value corresponding to the key?

- **Step 1:** We know that hash functions (which is some mathematical formula) are used to calculate the hash value which acts as the index of the data structure where the value will be stored.
- **Step 2:** So, let’s assign
  - “a” = 1,
  - “b”=2, .. etc, to all alphabetical characters.
- **Step 3:** Therefore, the numerical value by summation of all characters of the string:
- *“ab” = 1 + 2 = 3,*
- *“cd” = 3 + 4 = 7 ,*
- *“efg” = 5 + 6 + 7 = 18*
- **Step 4:** Now, assume that we have a table of size 7 to store these strings. The hash function that is used here is the sum of the characters in **key mod Table size.** We can compute the location of the string in the array by taking the **sum(string) mod 7.**
- **Step 5:** So, we will then store
  - “ab” in 3 mod 7 = 3,
  - “cd” in 7 mod 7 = 0, and
  - “efg” in 18 mod 7 = 4.

0	1	2	3	4	5	6
cd			ab	egf		

Basic Operations:

Following are 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.

Hash Table

```
class HashTableEntry {
public:
   int k;
   int v;
   HashTableEntry(int k, int v) : k(k), v(v) {}
};


class HashMapTable {
private:
   static const int TABLE_SIZE = 10;  // Adjust the size as needed
   HashTableEntry **t;

public:
   HashMapTable() {
      t = new HashTableEntry *[TABLE_SIZE];
      for (int i = 0; i < TABLE_SIZE; ++i) {
         t[i] = nullptr;
      }
   }

   int HashFunc(int k) {
      return k % TABLE_SIZE;
   }

   void Insert(int k, int v) {
      int hash = HashFunc(k);
      if (t[hash] == nullptr) {
         t[hash] = new HashTableEntry(k, v);
      } else {
         // Handle collisions (e.g., chaining or open addressing)
         // You can implement a collision resolution strategy here
      }
   }

   int SearchKey(int k) {
      int hash = HashFunc(k);
      if (t[hash] != nullptr && t[hash]->k == k) {
         return t[hash]->v;
      } else {
         // Handle search miss
         return -1;  // Or any suitable value indicating not found
      }
```

```
    }

    void Remove(int k) {
        int hash = HashFunc(k);
        if (t[hash] != nullptr) {
            delete t[hash];
            t[hash] = nullptr;
        } else {
            // Handle removal of non-existent key
        }
    }

    ~HashMapTable() {
        for (int i = 0; i < TABLE_SIZE; ++i) {
            delete t[i];
        }
        delete[] t;
    }
};
```

Hash Functions

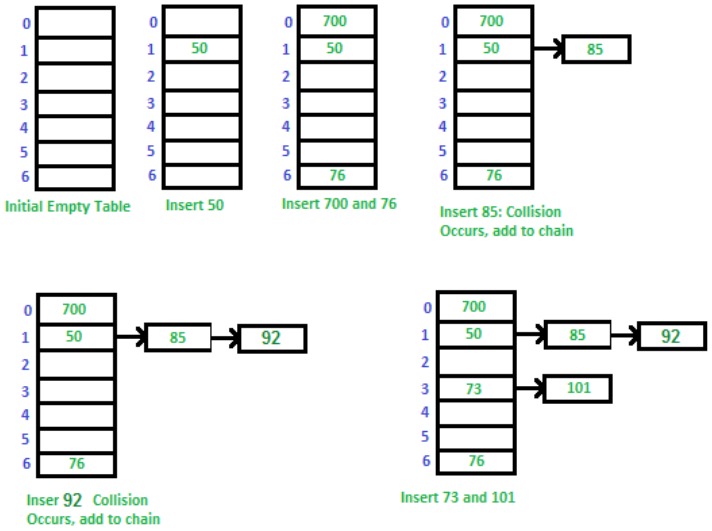
```
int K_Mod_N(int key, int N) {
    return key % N;
}

int middle_value(int value) {
    // Implement a function to extract the middle digits from the squared value
    // This depends on the requirements of your application
    return value;  // Replace with actual implementation
}

int mid_square_hash(int key) {
    int value = key * key;
    int middle_value_result = middle_value(value);
    return middle_value_result;
}

int Folding_hash(int value1, int value2, int value3) {
    return value1 + value2 + value3;
}
```

Ideal Hashing:

Separate Chaining

The linked list data structure is used to implement this technique. So, what happens is, when multiple elements are hashed into the same slot index, then these elements are inserted into a singly-linked list which is known as a chain.

```
class HashNode {
public:
   int key;
   int value;
   HashNode* next;
   HashNode(int key, int value) : key(key), value(value), next(nullptr) {}
};
class HashMap {
private:
   HashNode** htable;
public:
   HashMap() {
      htable= new HashNode*[TABLE_SIZE];
      for (int i = 0; i < TABLE_SIZE; i++)
         htable [i] = nullptr;
   }
   ~HashMap() {
      // Add code to deallocate memory for each linked list in the array
      // and then deallocate the array itself
   }
   // Add methods like insert, search, and remove here
};
```



Formula:

$$h(x) = x \% 10$$

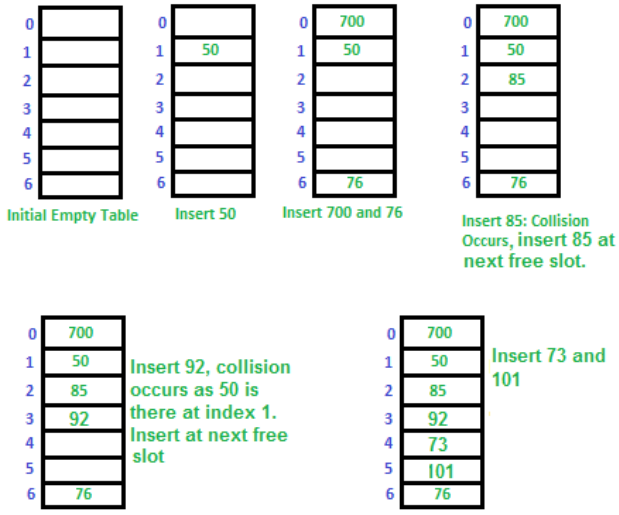
Example:

16 12 25 39 6 122 5 68 75

Linear Probing

In linear probing, the hash table is searched sequentially that starts from the original location of the hash. If in case the location that we get is already occupied, then we check for the next location.

```
void Table::insert(const RecordType& entry)
{
   bool alreadyThere;
   int index;

   assert(entry.key >= 0);

   findIndex(entry.key, alreadyThere, index);
   if (!alreadyThere)
   {
      assert(size() < CAPACITY);
      used++;  // Increment used only when a new element is inserted.
   }
   data[index] = entry;
}
```



Process:

Let hash(x) be the slot index computed using a hash function and S be the table size
If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S
If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

Formula:

h'(x) = (h(x) + f(i)) % 10, where f(i) = i (index) and h(x) = x % 10

Example:

26 30 45 23 25 43 74

Quadratic Probing

The interval between probes will increase proportionally to the hash value. This method is also known as the **mid-square** method. In this method, we look for the **i2'th** slot in the **ith** iteration. We always start from the original hash location. If only the location is occupied then we check the other slots.

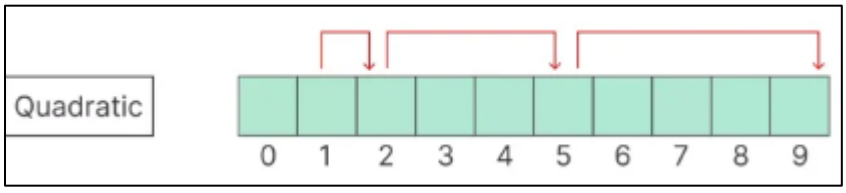
```
function hashing(table, tsize, arr, n)
   // Iterate through each element of the array
   for i from 0 to n-1
      // Compute the hash value
      hv = arr[i] % tsize

      // If no collision, insert the value
      if table[hv] == -1
         table[hv] = arr[i]
      else
         // If collision occurs, check for quadratic probing
         for j from 1 to tsize
            // Compute the new hash value using quadratic probing
            t = (hv + j * j) % tsize
            if table[t] == -1
               // Insert the value and break out of the loop
               table[t] = arr[i]
               break
         end for
      end if
   end for

   // Call function to print the array
   printArray(table, tsize)
end function
```

Process:

let hash(x) be the slot index computed using hash function.
If slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3) % S

Formula:

$$h'(x) = (h(x)+f(i)) \bmod m, \text{ where } f(i) = i^2 \text{ (index) and } h(x) = x \% 10$$



Double Hashing

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Double hashing can be done using:

(hash1(key) + i * hash2(key)) % TABLE_SIZE

Here hash1() and hash2() are hash functions and TABLE_SIZE is size of hash table. (We repeat by increasing i when collision occurs)

First hash function:

hash1(key) = key % TABLE_SIZE

Second hash function:

hash2(key) = PRIME – (key % PRIME) where PRIME is a prime smaller than the TABLE_SIZE.

```
int doubleHash(int key, int i, int TABLE_SIZE) {
   int hash1 = key % TABLE_SIZE;
   int hash2 = PRIME - (key % PRIME);
   return (hash1 + i * hash2) % TABLE_SIZE;
}
```

Process:

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Formula:

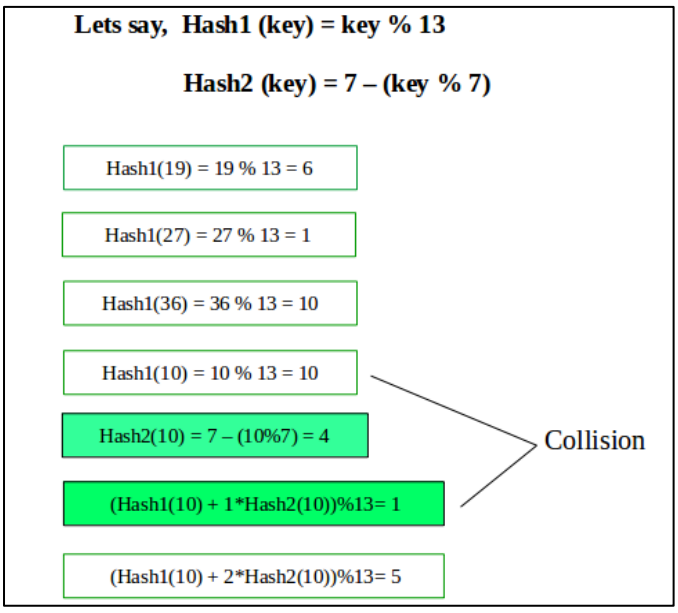
h1(x)=x % 10 //here 10 can be n

h2(x)=7 - (x % 7) //Can be any value but the most common taken is 7

h'(x) = (h1(x) + i * h2(x)) % 10 //again here 10 can be n

Example:

5 25 15 35 95



Tasks

01. Write a Program to implement Hash table and implement the following task with arrays and linked list as well by using any hashing calculation method.

Keys = (20,34,45,70,56)

- **a.** Insert element into the table
- **b.** Search element from the key
- **c.** Delete element at a key

02. Given an array of N integers, and an integer K, find the number of pairs of elements in the array whose sum is equal to K. Use Hashing (time complexity should not be more than N worst case)

Input:
N = 4, K = 6
arr[] = {1, 5, 7, 1}
Output:
2
Explanation:
arr[0] + arr[1] = 1 + 5 = 6
and arr[1] + arr[3] = 5 + 1 = 6.

03. Given an array arr[] of n integers. Check whether it contains a triplet that sums up to zero and time complexity should not exceed(n^2). Use hashing with any method

Note: Return 1, if there is at least one triplet following the condition else return 0.
Input:
n = 5, arr[] = {0, -1, 2, -3, 1}
Output:
1
Explanation:
0, -1 and 1 forms a triplet with sum equal to 0.

04. Given a set of N nuts of different sizes and N bolts of different sizes. There is a one-one mapping between nuts and bolts. Match nuts and bolts efficiently.
Comparison of a nut to another nut or a bolt to another bolt is not allowed. It means nut can only be compared with bolt and bolt can only be compared with nut to see which one is bigger/smaller.
The elements should follow the following order ! # $ % & * @ ^ ~ .

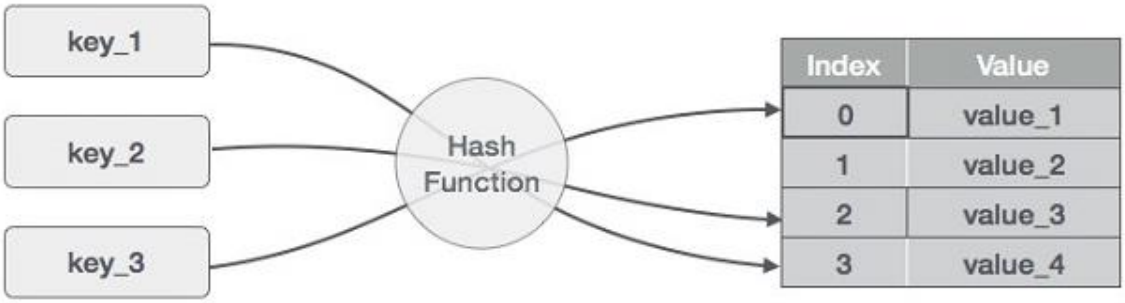
Input:
N = 5
nuts[] = {@, %, $, #, ^} //You can use any symbols they are not exclusive to this
bolts[] = {%, @, #, $ ^}
Output:
$ % @ ^
$ % @ ^

Note: Might sound easy but the max allowed time complexity is N*log(N) with hashes.

References:

1. Hash Table

A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. It is an array into which data is inserted using a hash function.



2. Hash Functions

i. Division Method

This is the easiest method to create a hash function. The hash function can be described as h(k) = k mod n. Here, h(k) is the hash value obtained by dividing the key value k by size of hash table n using the remainder. It is best that n is a prime number as that makes sure the keys are distributed with more uniformity.

ii. Multiplication Method

The hash function used for the multiplication method is h(k) = floor(n(kA mod 1)) Here, k is the key and A can be any constant value between 0 and 1. Both k and A are multiplied and their fractional part is separated. This is then multiplied with n to get the hash value.

iii. Mid Square Value Method

In Mid square, the key is squared and the address is selected from the middle of the result.

iv. Folding Method

Divide the key into several parts with same length (except the last part) • Then sum up these parts (drop the carries) to get the hash address Two method of folding:

Shift folding — add up the last digit of all the parts with alignment

Boundary folding — each part doesn't break off, fold to and fro along the boundary of parts, then add up these with alignment, the result is a hash address

v. Radix Method

Regard keys as numbers using another radix then convert it to the number using the original radix. Pick some digits of it as a hash address, usually choose a bigger radix as converted radix, and ensure that they are inter-prime

3. Collisions

The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Since a hash

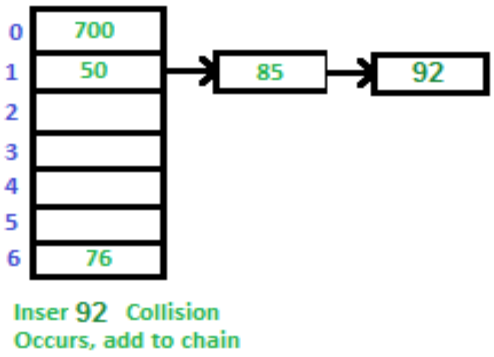
function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value.

4. Linear Probing

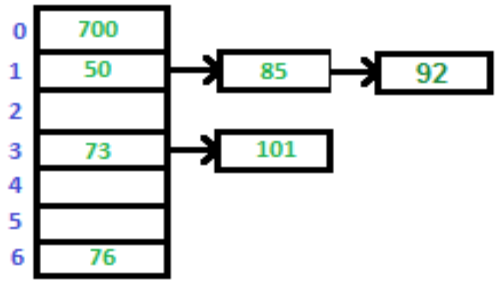
A hash table in which a collision is resolved by putting the item in the next empty place in the array following the occupied place. The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by h(k), it means collision occurred then we do a sequential search to find the empty location. Here the idea is to place a value in the next available position. Because in this approach searches are performed sequentially so it's known as linear probing. Here array or hash table is considered circular because when the last slot reached an empty location not found then the search proceeds to the first location of the array.

5. Chaining

A chained hash table fundamentally consists of an array of linked lists. Each list forms a bucket in which we place all elements hashing to a specific position in the array. To insert an element, we first pass its key to a hash function in a process called hashing the key. This tells us in which bucket the element belongs. We then insert the element at the head of the appropriate list. To look up or remove an element, we hash its key again to find its bucket, then traverse the appropriate list until we find the element we are looking for. Because each bucket is a linked list, a chained hash table is not limited to a fixed number of elements. However, performance degrades if the table becomes too full.



Inser 92 Collision
Occurs, add to chain



Insert 73 and 101