HASH TABLE

Here is our Team:

LIM KIMHOUNG	e20200339
EK VONGPANHARITH	e20200887
HENG PAVISAL	e20200291
HIN BUNRA	e20201287
KHEANGTONGHEANG	e20200472
HONG KIMLENG	e20200766

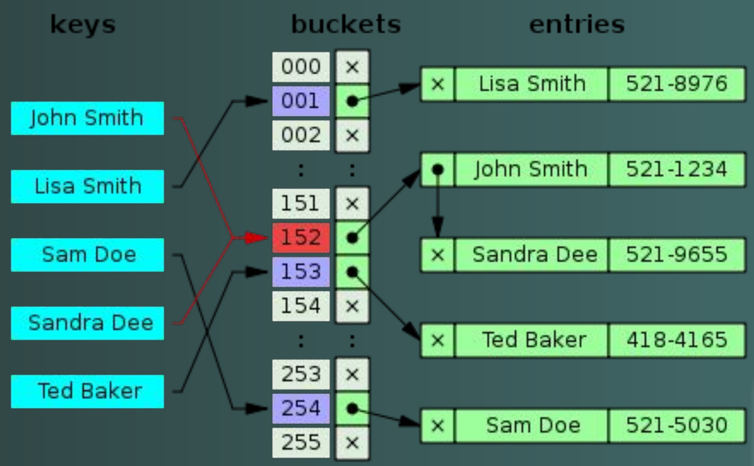
1 ABOUT HASH TABLE

WHAT IS HASH TABLE?

- It is a data structure that you can use to store data in key-value format with direct access to its items in constant time.

- Has Table are said to be associative, which means that for each key, data occurs at most once. Hash table let us implement things like phone books or dictionaries

- We can use hash table to store retrieve and delete data uniquely based on their unique key.



1. ABOUT HASH TABLE

WHY USE HASH TABLE?

- A hash table offers very fast insertion and searching, almost O(1).
- Relatively easy to program as compared to trees
- Based on arrays, hence difficult to expand.
- No convenient way to visit the items in a hash table in any kind of order.
- A range of key values can be transformed into a range of array index values. A simple array can be used where each record occupies one cell of the array and the index number of the cell is the key value for that record. But keys may not be well arranged.

1. ABOUT HASH TABLE

EMPLOYEE DATABASE EXAMPLE

- A small company with, say, 1,000 employees, every employee has been given a number from 1 to 1,000, etc.

What sort of data structure should you use in this situation?

Index Numbers As Keys

- One possibility is a simple array. Each employee record occupies one cell of the array, and the index number of the cell is the employee number for that record.
- O(1) time to perform *insert, remove, search*

Not Always So Orderly

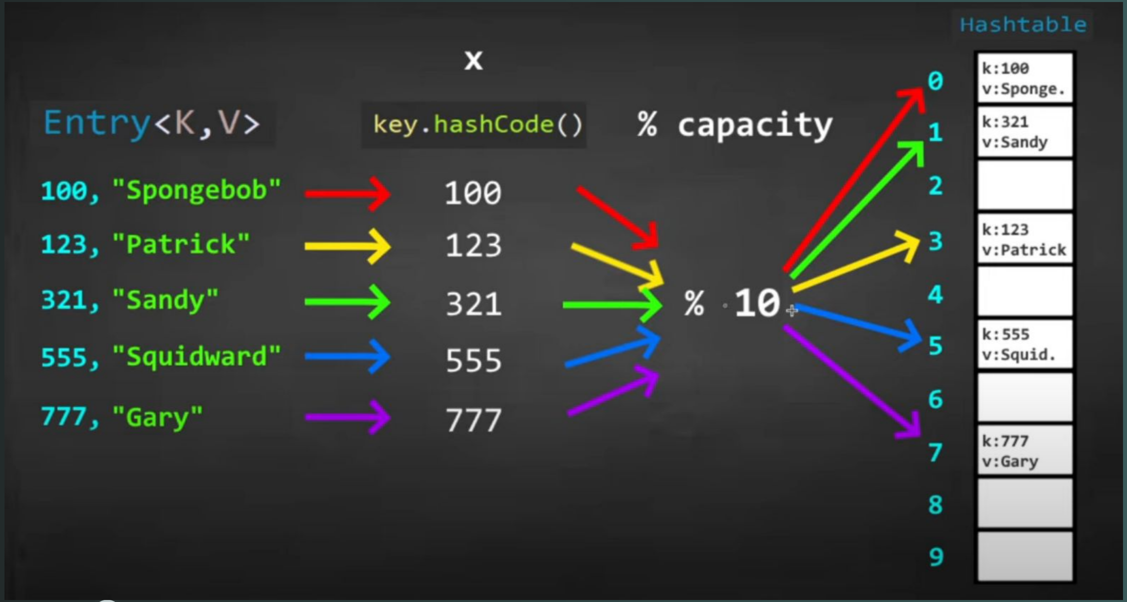
- speed and simplicity of data access make this very attractive. However, this example works only because the keys are unusually well organized.
- Ideal case is unrealistic!

2. HOW DO HASH TABLE WORK?

There are 4 distinct aspects to discuss how hash table work :

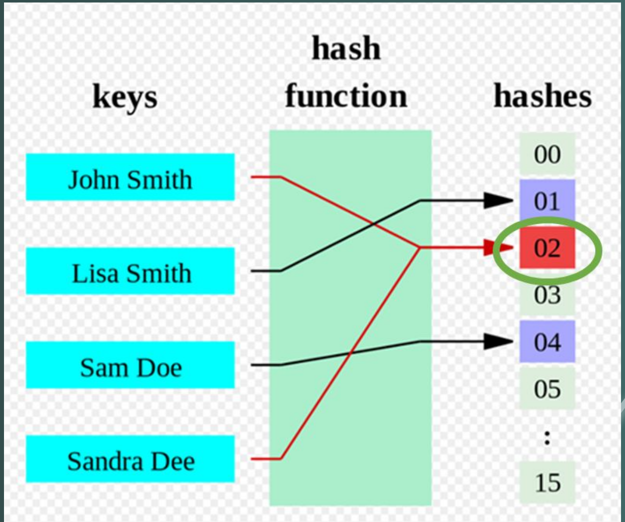
- **Storage:** It use an array as the storage.
- **Key-Value Pair:** It has an unique key to a value.
- **Hash Function:** It will use the key to determine what index the value will store in the array.
- **Table Operation:** It should be able to perform:
  - Add - add a key-value pair
  - Get - get a value by key
  - Remove - remove a value by key
  - List - get all the keys
  - Count - count the number of items in table

2. HOW DO HASH TABLE WORK?



3 Collision in Hashing

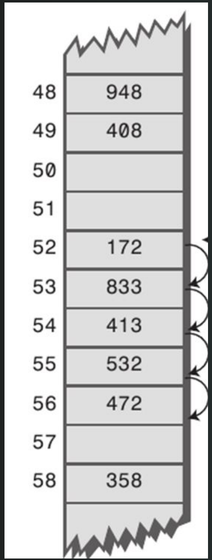
- The Hash function maps multiple keys to the same location
- More than will fit in the hash bucket
- A collision or clash occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure(hash table).

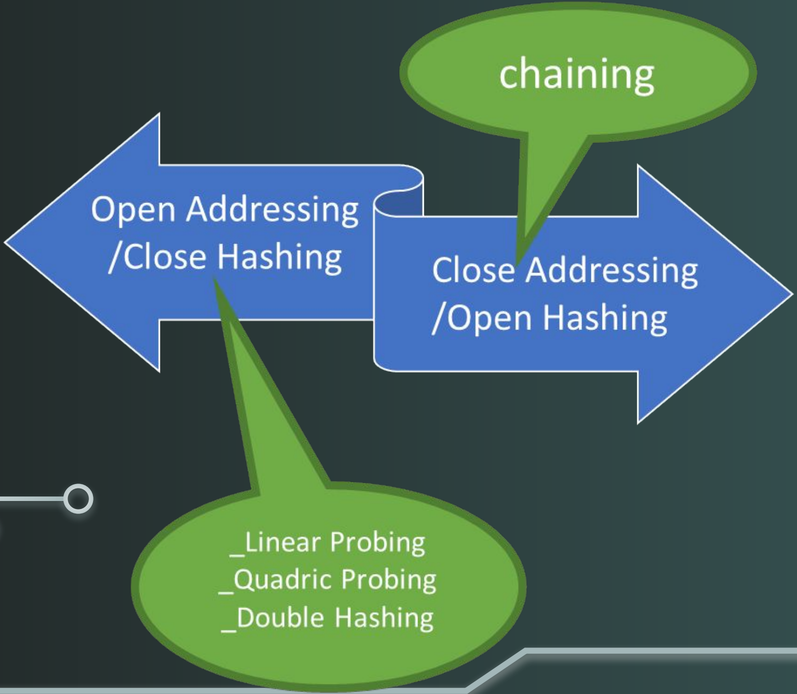


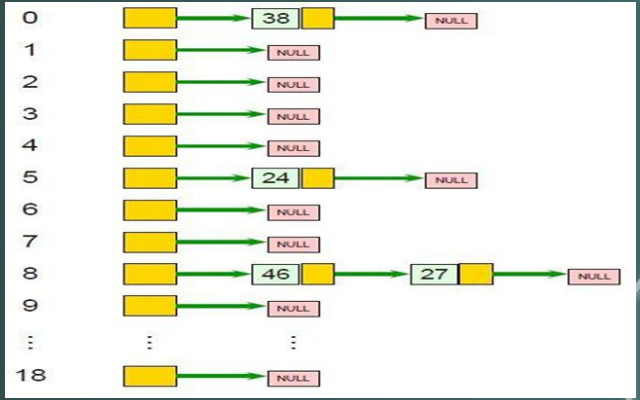
3 Collision in Hashing

Two of the Most common strategy of Collision are

Open Addressing and Chaining.







4 Hashing Technique

h(k) = 2k + 3 (Hashing Function)
hashing method: Linear Probing

Insert k(i) at first free location from (u+i)%m where i = 0,1, n-1
Example:
(5 + 0)/10 = 5
(5 + 1)/10 = 6

Since index 5 is not free but 6 is free so the key value 11 will store in index 6.

0	13
1	9
2	12
3	
4	
5	6
6	11
7	2
8	7
9	3

Hash Table

Key	Location
3	[(2X3)+3]%10 = 9
2	[(2X3)+2]%10 = 7
9	[(2X9)+3]%10 = 1
6	[(2X3)+6]%10 = 5
11	[(2X11)+3]%10 = **5**
13	[(2X3)+13]%10 = **9**
7	[(2X7)+3]%10 = **7**
12	[(2X12)+3]%10 = **7**

4 Hashing Technique

h(k) = 2k + 3 (Hashing Function)
hashing method: **Quadric Probing**

Insert k(i) at first free location from (u+i^2)%m where i = 0,1, m-1

Example:

(7 + 0)/10 = 5
(7 + 1)/10 = 8
(7 + 4)/10 = 1
(7 + 9)/10 = 6
(7 + 16)/10 = 3

Since index 5 is not free but until index 3 is free so the key value 12 will store in index 3.

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

Hash Table

Key	Location (u)
3	[(2X3)+3]%10 = 9
2	[(2X3)+2]%10 = 7
9	[(2X9)+3]%10 = 1
6	[(2X3)+6]%10 = 5
11	[(2X11)+3]%10 = 5
13	[(2X3)+13]%10 = 9
7	[(2X7)+3]%10 = 7
12	[(2X12)+3]%10 = 7

4 Hashing Technique

h1(k) = 2k + 3 (Hashing Function1)
h2(k) = 3k + 1 (Hushing Function 2)
hashing method: **Double Hushing**

Insert k(i) at first free location from (u + v * i) % m where i = 0,1,...,m-1

0	13
1	9
2	
3	11
4	
5	6
6	
7	2
8	
9	3

Hash Table

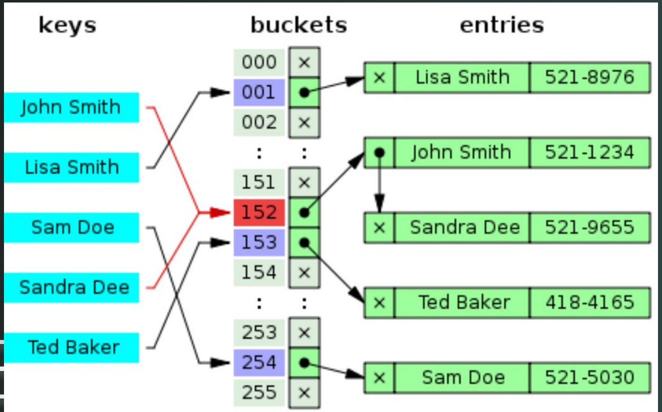
Key	Location (u)
3	[(2X3)+3]%10 = 9
2	[(2X3)+2]%10 = 7
9	[(2X9)+3]%10 = 1
6	[(2X3)+6]%10 = 5
11	[(2X11)+3]%10 = 5
13	[(2X3)+13]%10 = 9
7	[(2X7)+3]%10 = 7
12	[(2X12)+3]%10 = 7

Key	Location (v)
3	[(3X3)+1]%10 = 0
2	...
9	..
6	..
11	...
13	..
7	...
12	..

How are we different?

HASH TABLE

- It also store key value pair but it generate the key by hush function to determine which index the value will be store.



ASSOCIATIVE ARRAY

- It store key value pair.

HUSHING IMPLEMENTATION

<u>Seperate Chaining Method</u>

```cpp
#include <iostream>
#include <list>

int TABLE_SIZE = 128;

class HashNode
{
public:
    int key;
    int value;
    HashNode* next;
    HashNode(int k, int v)
    {
        this->key = k;
        this->value = v;
        this->next = nullptr;
    }
};
class HashMap
{
private:
    HashNode** hashTable;
public:
    HashMap()
    {
        hashTable = new HashNode * [TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i += 1)
        {
            hashTable[i] = nullptr;
        }
    }
```

CLASSES OF HASH TABLE NODE AND HASH TABLE MAP.

```cpp
int HashFunc(int key)
{
    return key % TABLE_SIZE;
}

void _Insert(int key, int value)
{
    int Index = HashFunc(key);
    HashNode* prev = nullptr;
    HashNode* entry = hashTable[Index];

    while (entry != nullptr)
    {
        prev = entry;
        entry = entry->next;
    }
    if (entry == nullptr)
    {
        entry = new HashNode(key, value);

        if (prev == nullptr)
        {
            hashTable[Index] = entry;
        }
        else
        {
            prev->next = entry;
        }
    }
    else
    {
        entry->value = value;
    }
}
```

INSERT FUNCTION AND FUNCTION TO GENERATE INDICES OF HASH TABALE

```cpp
void _Search(int key)
{
    int Index = HashFunc(key);

    HashNode* entry = hashTable[Index];

    while (entry != nullptr)
    {
        if (entry->key == key)
        {
            std::cout << "\n\t\t" << entry->key << "---->" << entry->value << " ";
        }
        entry = entry->next;
    }
}

void Display()
{
    HashNode* temp;
    for (int i = 0; i < TABLE_SIZE; i += 1)
    {
        temp = hashTable[i];
        while (temp != nullptr)
        {
            std::cout << "\n\t\t" << i << " ----> " << temp->value << std::endl;
            break;
        }
    }
}
```

SEARCH FUNCTION TO SEARCH FOR KEY ELEMENT AND DISPLAY ALL ELEMENT IN HASH TABLE.

HUSHING IMPLEMENTATION

Linear Probing Method

```cpp
#include <iostream>

using std::cout;
using std::endl;
using std::cin;

const int Table_Size = 15;

class Hash_Table
{
public:
    int key , value;
    Hash_Table(int k , int v)
    {
        this->key = k;
        this->value = v;
    }
};

class Delete_Node:public Hash_Table
{
private:
    static Delete_Node *entry;

    Delete_Node(): Hash_Table(-1 , -1) {};
public:
    static Delete_Node *init_node()
    {
        if (entry == nullptr)
        {
            entry = new Delete_Node();
        }
        return entry;
    }
};
```

```cpp
Delete_Node *Delete_Node::entry = nullptr;

class HashMapTable
{
private: Hash_Table** hash_t;
public:
    HashMapTable()
    {
        hash_t = new Hash_Table* [Table_Size];
        for (int i = 0 ; i < Table_Size ; i++)
        {
            hash_t[i] = nullptr;
        }
    }

    int getHash(int key)
    {
        return key % Table_Size;
    }
```

```
void Insert_(int k , int v)
{
    int Index = getHash(k);
    int init = -1;
    int del_index = -1;

    while(Index != init && (hash_t[Index]) == Delete_Node::init_node() || hash_t[Index] != nullp
                && hash_t[Index]->key != k )
    {
        if (init == -1)
        {
            init = Index;
        }
        if (hash_t[Index] == Delete_Node::init_node())
        {
            del_index = Index;
            Index = getHash(Index + 1);
        }
    }
    if (hash_t[Index] == nullptr || Index == init)
    {
        if (del_index != -1)
        {
            hash_t[del_index] = new Hash_Table (k , v);
        }
        else
        {
            hash_t[Index] = new Hash_Table(k , v);
        }
    }
```

```
if(init != Index)
{
    if (hash_t[Index] !+ Delete_Node::init_node())
    {
        if (hash_t[Index] != nullptr)
        {
            if (hash_t[Index] -> key == k)
            {
                hash_t[Index] -> value = v;
            }
        }
    }
    else
    {
        hash_t[Index] = new Hash_Table(k , v);
    }
}
```

HUSHING IMPLEMENTATION
<u>Quadratic Probing</u>

```cpp
#include <iostream>
#define MAX_TABLE_SIZE 10

using std::cout;
using std::cin;
using std::endl;

enum EntryType { Legit , Empty , Deleted };

struct HashNode
{
    int element;
    enum EntryType info;
};

struct HashTable
{
    int size;
    HashNode *table;
};
```

DATA STRUCTURE OF OUR HASH TABLE.

```
int HashFunc(int key , int size)
{
    return key % size;
}
```

FUNCTION TO GENERATE INDEX FOR HASH TABLE.

FUNCTION TO INITIALIZE HASH TABLE WITH SIZE PARAMETER

```cpp
HashTable* init_Table(int size)
{
    HashTable *hash_t;

    if (size < MAX_TABLE_SIZE)
    {
        cout << "\n\t\tTable size is too small!" << endl;
        return nullptr;
    }

    hash_t = new HashTable();

    if (hash_t == nullptr)
    {
        return nullptr;
    }

    hash_t -> size = nextPrime(size);
    hash_t -> table = new HashNode [hash_t -> size];

    if (hash_t -> table == nullptr)
    {
        std::cout << "\n\t\tTable size is too small!" << endl;
        return nullptr;
    }

    for(int i = 0 ; i < hash_t -> size ; i += 1)
    {
        hash_t->table[i].info = Empty;
        hash_t->table[i].element = 0;
    }
    return hash_t;
}
```

```
int Search(int key , HashTable* h_t)
{
    int Index = HashFunc(key , h_t -> size);
    int collision = 0;

    while (h_t -> table[Index].info != Empty && h_t ->table[Index].element != key)
    {
        Index = Index + 2 * ++collision - 1;
        if (Index >= h_t -> size)
        {
            Index = Index - h_t -> size;
        }
    }
    return Index;
}
```

SEARCH FUNCTION FOR HASH TABLE.

```c
void Insert (int key , HashTable* h_t)
{
    int Index = Search(key , h_t);

    if (h_t -> table[Index].info != Legit)
    {
        h_t -> table[Index].info = Legit;
        h_t -> table[Index].element = key;
    }
}

HashTable *Rehash(HashTable *h_t)
{
    int size = h_t -> size;
    HashNode* table = h_t -> table;
    h_t = init_Table(2 * size);
    for (int i = 0; i < size ; i++)
    {
        if (table[i].info == Legit)
        {
            Insert(table[i].element , h_t);
        }
    }
    free(table);
    return h_t;
}
```

FUNCTION TO INSERT KEY AND VALUE TO HASH TABLE AND A FUNCTION TO REHASH THE HASH TABLE IN CASE OUR HASH TABLE IS FULL IN SIZE.

```cpp
void display(HashTable* h_t)
{
    for (int i = 0 ; i < h_t -> size ; i += 1)
    {
        int value = h_t -> table[i].element;
        char null = '/';

        if (value)
        {
            std::cout << "\n\t\tIndex: " << i + 1 << " ---> Element: " << value << std::endl;
        }
        else
        {
            std::cout << "\n\t\tIndex: " << i + 1 << " ---> Element: " << null << std::endl;
        }
    }
}
```

A FUNCTION TO DISPLAY ALL KEYS AND VALUES IN HASH TABLE.

THANK YOU!