# Lab Tutorial for Week 6: Hash table Implementation

## Contents

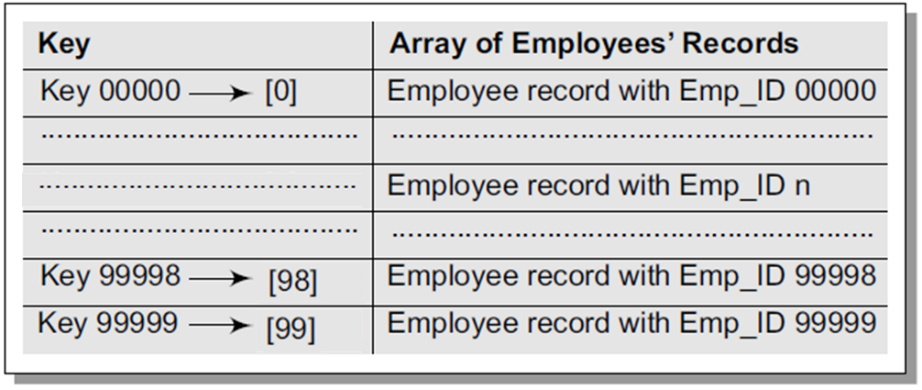
1. Tutorial w6a: Hash table Implementation
2. Tutorial w6b: Application of Hash table

## Tutorial w6a: Hash table Implementation

### Overview:

### You may already know that linear search has a running time proportional to *O*(n), while binary search takes time proportional to *O*(*log n*), where n is the number of elements in the array. What if we want to perform the search operation in time proportional to *O*(1)? In other words, is there a way to search an array in constant time, irrespective of its size?

### Hash table is a data structure in which keys are mapped to array positions by a hash function and in fact, it performs the search operation in time proportional to *O*(1).



A value stored in a hash table can be searched in *O*(1) time by using a hash function which generates an address from the key (by producing the index of the array where the value is stored). A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table (the **home address** or **home position**).The simplest method of hashing an integer *x* module operation, so in this case, the hash function can be given as:

*h*(*x*) = *x* mod *M*.

There are many other methods of hashing, such as multiplication method, mid-square method, folding method, etc. Please search for internet sources if you're curious how these hashing methods work.

There is no perfect has function that can maps input data to address or location one-to-one, instead a hash may produce the same address for different data. In such case, collisions occur and obviously, two records cannot be stored in the same location. Therefore, a collision resolution is required. The two most popular methods of resolving collisions are**:**

### Open addressing: Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position.

### Chaining: In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location.

### Practice:

1. Open the starter file (LabWeek6ClosedHashing) in your IDE.
2. This file contains an implementation of a Hash table using Closed Hashing (a.k.a Open Addressing), but the implementation of delete operations is missing.

class ClosedHashtable

{

private:

int ht[10];

int i;

int found = 0;

int key;

public:

ClosedHashtable();

void insert\_val();

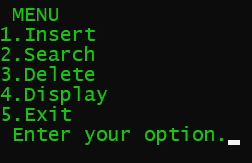
void search\_val();

void delete\_val();

void display();

};

1. Your task is to complete this delete operation. (*hints: use the search function*)
2. Save the completed source file and make sure it’s compiled. When you run it, you will have the following prompt:



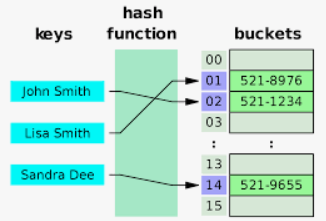
1. Test your Hash table with the following scenarios:
   1. Insert data into the Hash table until the Hash table is full. What would happen with your program?
   2. Delete the data in the Hash table until the Hash table is empty. What would happen with your program?

### Questions:

1. Describe how your code above does collision resolution by printing (cout) what happen in the insert function when collision is occurring.
2. Which method of collision resolution does your code use?
3. Search in internet for a phenomenon called clustering in hash table. Why does this phenomena happen?
4. What is the worst-case situation for a search in hash table?

### Optional Practice: BUCKET HASHING

One implementation of closed hashing groups the hash table into buckets where M slots of the hash table are divided into B buckets. Therefore, each bucket contains M/B slots. When a new record has to be inserted, the hash function computes the home position. If the slot is free, the record is inserted. Otherwise, the bucket’s slots are sequentially searched until an open slot is found. In case, the entire bucket is full, the record is inserted into an overflow bucket. The overflow bucket has infinite capacity at the end of the table and is shared by all the buckets.



1. Open the starter file (LabWeek6BucketHashing) in your IDE.
2. This file contains an implementation of a Hash table using Bucket Hashing Groups, but there some code in the push (insert) function that is missing.

#define MAX\_DATA 10

typedef struct node{

int key;

char data[100];

struct node \*next;

} Node;

typedef struct hashbucket{

int number\_of\_node;

Node \*head;

Node \*tail;

} HashBucket;

class Hash

{

private:

HashBucket\* HASH\_TABLE;

public:

Hash(HashBucket\*);

~Hash();

int sum\_digits(int input, int base);

int folding(char \*input);

int division(char \*input);

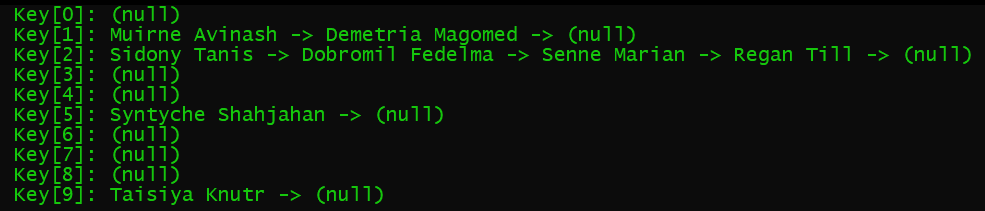
Node\* create\_node(char\* data);

void push\_data(Node \*node);

void view\_data();

};

1. Your task is to complete the missing code.
2. Save the completed source file and make sure it’s compiled. This is the initial display when you run it using tester code.



1. Test your Hash table with the following scenarios:
   1. Insert data into the Hash table until the Hash table is full. What would happen with your program?
   2. Delete the data in the Hash table until the Hash table is empty. What would happen with your program?

## Tutorial w6b: Application of Hash table

### Hash tables are widely used in situations where enormous amounts of data must be accessed to quickly search and retrieve information, for examples:

1. Driving license database
2. Insurance database

### Imagine how big is the database that stores the driving license or insurance for the whole population of a country?

### Tasks:

1. Select one of the above examples.
2. Identify how the unique ID (key) can be generated (*hints: using hashing from data items that never change: date of birth, name, etc. or combination of them*)
3. Use your ADT Hash table in the practice above to build this database.
4. Test your hash table application by adding enough data then retrieve them using the key

---end of Tutorial Week 6---