

# HASHING

CC-5 Data Structures

Department of Computer and System Sciences

*Visva Bharati University*

*Shantiniketan – 731325*



Arpan Pal

Bijan Roy

Harsh Sarkar

Vigyan Kumar

# INTRODUCTION

## DEFINITION

- In Computer Science, **hashing** is the process of mapping data of arbitrary size to fixed-size values called hash codes or hashes.

## PURPOSE

- Hashing is used to make hashed data structures like hash tables and hash sets where insertion, searching and deletion ideally take constant time.
- Hashing is used in cryptography to store and send confidential data.
- Hashing is used to check file integrity and detect changes in files.

# HASH TABLE

## DEFINITION

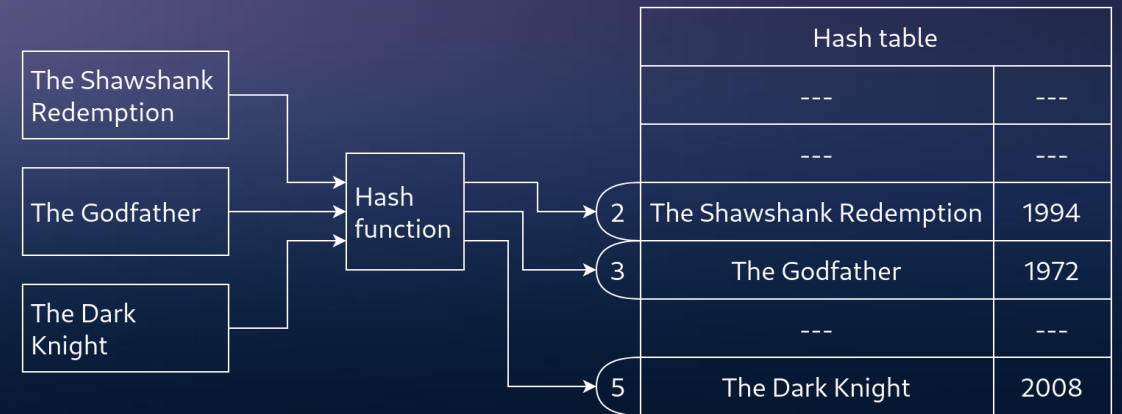
- A hash table is a hashed data structure which stores *key-value pairs* called *items*.
- Items are stored into buckets according to hashes returned by a hash function. This makes operations very fast.

## KEY COMPONENTS

- **Buckets:** Storage locations identified by indices generated from the hash function. Each bucket contains an item.
- **Hash function:** A mathematical function which returns the bucket index where an item should be stored.
- **Collision resolution:** Techniques for handling collisions, i.e. when two keys have the same hash.

## HASH TABLE OPERATIONS

- Insertion
- Deletion
- Searching



The hash table is an array of key-value pairs. Each block of memory can store a key-value pair and is called a bucket.

# HASH FUNCTION

## DEFINITION

- A hash function is a mathematical function which takes a variable length key as an argument and returns a fixed length value called hash.

## PURPOSE

- To return the index where an item has to be stored.
- To ensure even distribution of data across the hash table.

## CHARACTERISTICS

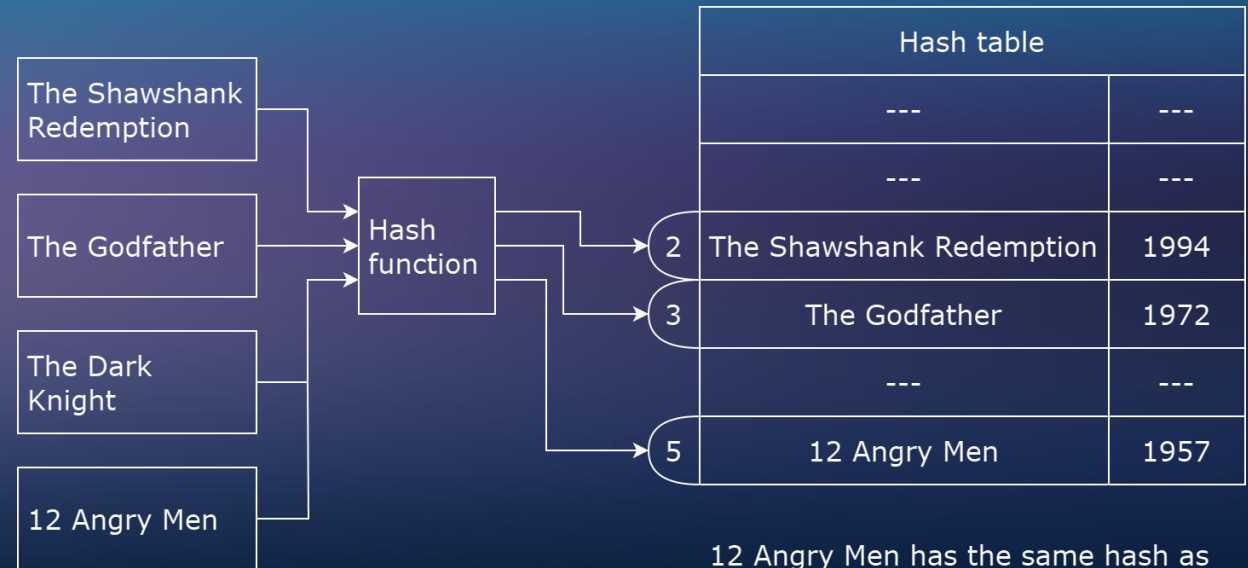
- **Deterministic:** The function must always return the same hash for the same key.
- **Fast:** The hash should not take too long to compute.
- **Minimum collision:** A good function aims at minimizing collision.

```
1  #include <string.h>
2
3  // fast and deterministic, but collision prone
4  int hashFunction1(char *key) {
5      return 5;
6  }
7
8  // fast and deterministic, but still high
9  // chance of collision
10 int hashFunction2(char *key) {
11     return strlen(key);
12 }
13
14 // fast and deterministic, but has much less
15 // chance of collisions than the previous two
16 int hashFunction3(char *key) {
17     int hash = 0;
18     while (*key) hash += *key++;
19     return hash;
20 }
```

# COLLISION

## DEFINITION

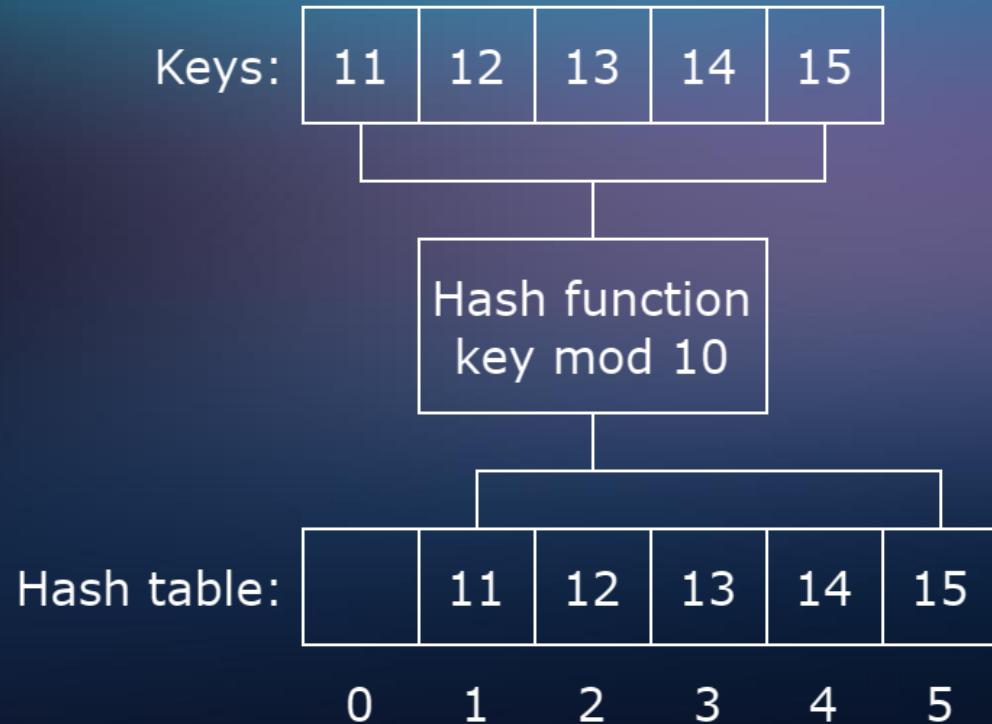
- A collision occurs when two different keys hash to the same index or location in the hash table.
- Collisions are important to address because if not handled properly, they can lead to data being overwritten or mixed up in the same location in the hash table.
- The colliding pair needs to be stored at another memory location. This is called collision resolution.
- Collision resolution techniques can broadly be categorized in two categories:
  - Separate chaining
  - Open addressing



12 Angry Men has the same hash as The Dark Knight and thus replaced the previous entry during collision.

# HASH TABLE

- A hash table is a data structure which stores key-value pairs according to hash codes returned by a hash function of the keys.
- A simple implementation of a hash table is given below.




```
1  int hashFunction(int key) {
2      return key % 10;
3  }
4
5  int main() {
6      int hashTable[6];
7      int keys[] = {11, 12, 13, 14, 15};
8      for (int i = 0; i < 5; ++i) {
9          int key = keys[i];
10         hashTable[hashFunction(key)] = key;
11     }
12 }
```

# HASH FUNCTION

- A hash function is a function which takes a variable length key as an argument and returns a fixed length hash.

## CRITERIA

- A hash function must have the following characteristics:
  - It should be fast to compute.
  - It should always return the same hash for the same key.
  - It should be chosen such that it reduces collision.
  - It should be chosen such that it evenly distributes items across the hash table.



```
1  int simpleHash(char *key) {
2      int hash = 0;
3      // sum of ascii values of characters
4      while (*key) hash += *key++;
5      return hash;
6  }
```

# DIVISION METHOD

- In this method, the hash is the remainder obtained when the key is divided by a constant M.
- M is smaller or equal to the size of the hash table.
- Remainder or modulo operation is used in many hash functions, but if M is divisible by too many numbers, clustering will increase. So M is preferably a prime number.

## Formula

$$H(key) = key \% M$$



```
1  int divisionMethod(int key) {  
2      const int M = 10;  
3      return key % M;  
4  }
```



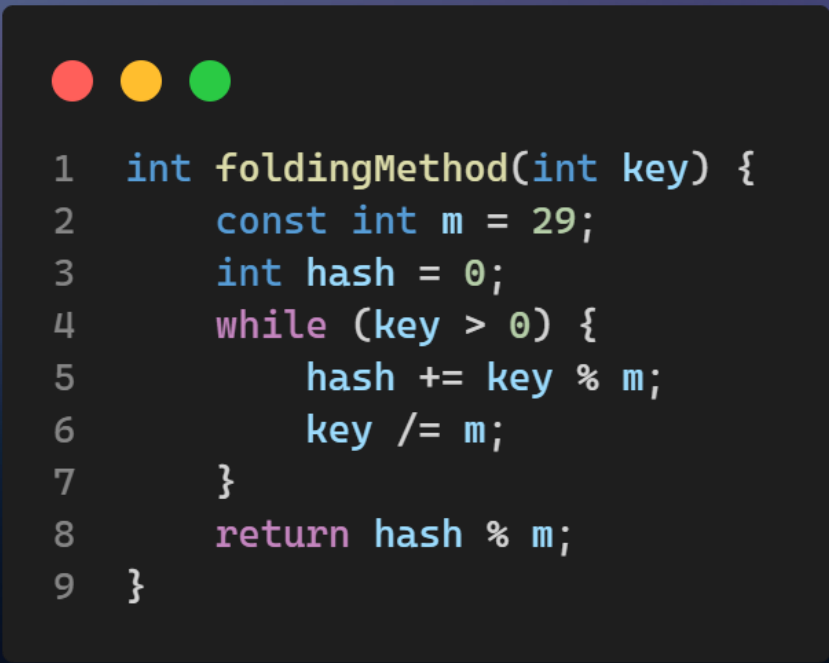
# MID SQUARE METHOD

- In mid-square method, *key* is squared.
- A constant, *r* digits are selected from the middle as the hash.
- The hash can be any combination of *r* digits, so the range of hash is  $[0, 10^r)$ .

```
1  int countDigits(int x);  
2  int sliceInt(int x);  
3  
4  int midSquareMethod(int key) {  
5      const int r = 2;  
6      int sq = key * key;  
7      int ndigits = countDigits(sq);  
8      int mid = (ndigits - r) / 2;  
9      return intSlice(sq, mid, r);  
10 }
```

# DIGIT FOLDING METHOD

- In folding method, a big *key* is broken down into smaller parts.
- The parts are then added to obtain the hash.



```
1  int foldingMethod(int key) {  
2      const int m = 29;  
3      int hash = 0;  
4      while (key > 0) {  
5          hash += key % m;  
6          key /= m;  
7      }  
8      return hash % m;  
9  }
```

# MULTIPLICATION METHOD

- In multiplication method,  $key$  is multiplied by a constant  $A$  and the decimal part is extracted.
- The extracted decimal is multiplied to another constant  $m$  to get the hash. The hash will always be less than  $m$ .
- Statistically, Golden Ratio or  $\Phi$ , 0.6180339887, is the best choice for  $A$ . and

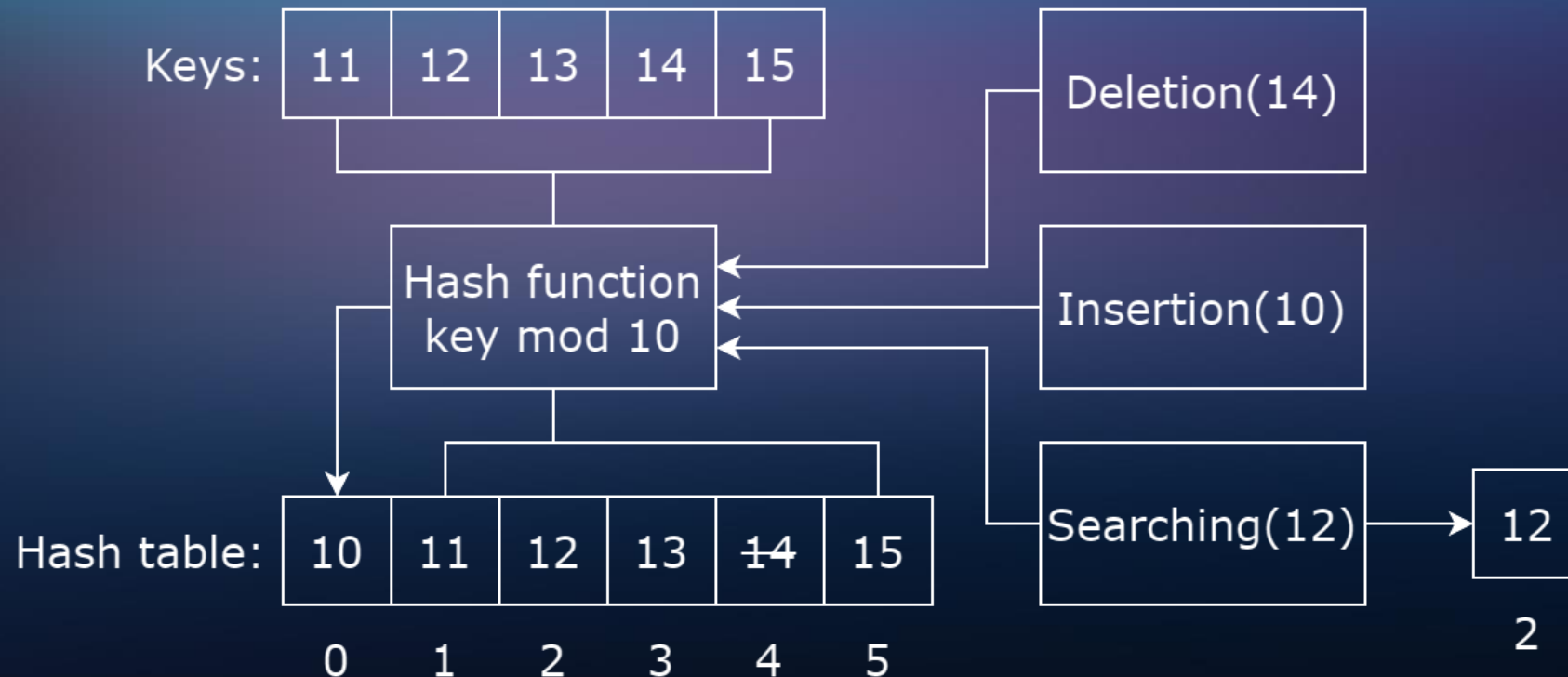
## FORMULA

$$H(key) = \text{floor}(m * (key * A - \text{floor}(key * A)))$$

```
1  int multiplicationMethod(int key) {  
2      const double A = 0.6180339887;  
3      const int m = 29;  
4      double product = key * A;  
5      return floor(m * (product - floor(product)));  
6  }
```

# HASH TABLE OPERATIONS

- **INSERTION** Adds a new key-value pair into the hash table.
- **DELETION** Removes an existing key-value pair from the hash table.
- **SEARCHING** Find and retrieve the value associated with a specific key.



# INSERTION



```
1  typedef Item;
2  typedef HashTable;
3
4  int hashfunction(char *key);
5  Item *create_item(char *key, int value);
6
7  void hash_insert(HashTable *hashtable, Item *item) {
8      int hash = hashfunction(item->key);
9      hashtable->items[hash] = create_item(item->key, item->value);
10 }
```

# DELETION



```
1  typedef Item;  
2  typedef HashTable;  
3  
4  int hashfunction(char *key);  
5  void delete_item(Item *item);  
6  
7  void hash_delete(HashTable *hashtable, char *key) {  
8      int hash = hashfunction(key);  
9      delete_item(hashtable->items[hash]);  
10     hashtable->items[hash] = NULL;  
11 }
```

# SEARCHING

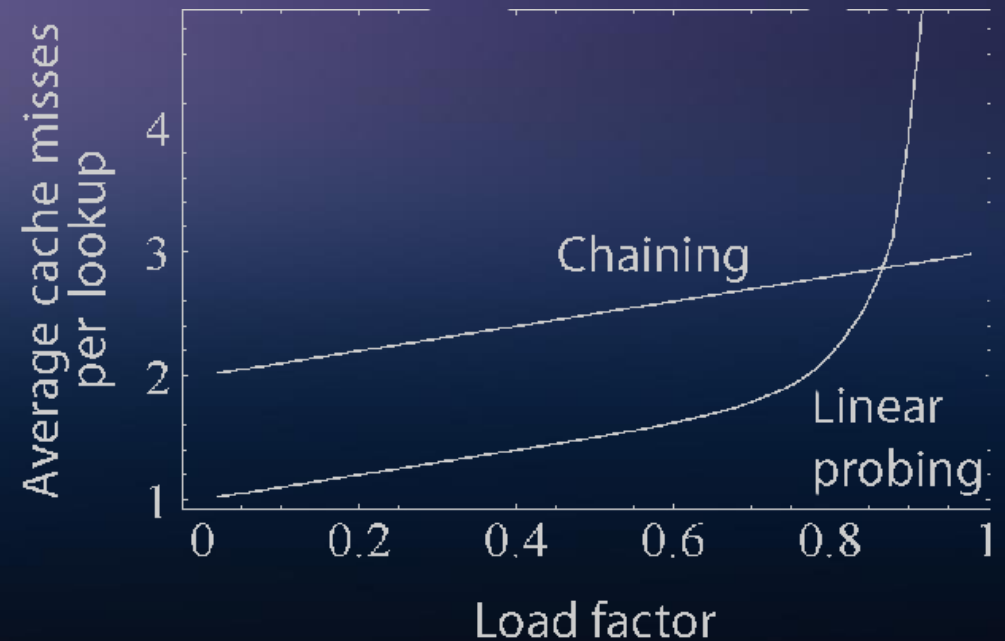


```
1  typedef Item;  
2  typedef HashTable;  
3  
4  int hashfunction(char *key);  
5  
6  int hash_search(HashTable *hashtable, char *key) {  
7      int hash = hashfunction(key);  
8      Item *item = hashtable->items[hash];  
9      return item->value;  
10 }
```

# LOAD FACTOR

- The load factor is defined as the ratio of the number of elements stored in the hash table and the total number of slots or buckets available in the table.
- It is denoted by the symbol  $\lambda$ .
- The efficiency of a hash table decreases as it starts filling up. So a maximum load factor is selected so that a proportion of the hash table is always empty.

$$\lambda = \frac{\text{Number of items}}{\text{Length of hash table}}$$





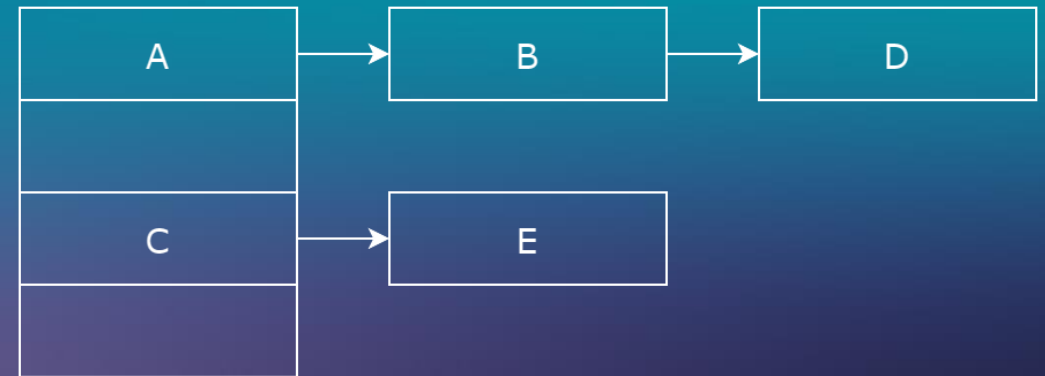
# COLLISION RESOLUTION TECHNIQUES

## DEFINITION

- Collision resolution is crucial for handling situations where multiple keys hash to the same index in a hash table.

## TYPES

- Separate chaining:** In this technique, each bucket is a linked data structure like *linked list* or *tree*. All keys having the same hash are nodes in the structure at the hash'th bucket.
- Open addressing:** In open addressing, no separate chains are created, colliding keys are stored at empty buckets in the hash table at an offset from the hash'th bucket.



Separate chaining using linked list



Open addressing using linear probing

# SEPARATE CHAINING

- In this method, each bucket contains a node. Each node contains an item and a reference to one or more nodes.
- If there are collisions, pointer to the colliding nodes can be stored in the existing node.
- The node can be part of any suitable data structure like linked list, tree or even another hash table.

```
1  typedef Item;
2  typedef Node;
3  typedef HashTable;
4
5  int hashfunction(char *key);
6  void list_insert(Node *head, Item *item);
7  Node *create_node(Item *item);
8
9  void hash_insert(HashTable *hashtable, Item *item) {
10     int hash = hashfunction(item->key);
11     Node *newnode = create_node(item);
12     if (hashtable->items[hash] == NULL)
13         hashtable->items[hash] = newnode;
14     else
15         list_insert(hashtable->items[hash], newnode);
16 }
```

# LINEAR PROBING

- In this method, if an item already exists in a bucket, colliding items are placed at an offset from the hash where there is no item.
- The offset is searched by sequentially checking each bucket one after the other.
- So the hash is  $H + i$ .

```
1  typedef Item;
2  typedef HashTable;
3
4  int hashfunction(char *key);
5
6  void hash_insert(HashTable *hashtable, Item *item) {
7      int hash = hashfunction(item->key);
8      if (hashtable->items[hash] == NULL)
9          hashtable->items[hash] = item;
10     else {
11         while (hashtable->items[++hash] != NULL);
12         hashtable->items[hash] = item;
13     }
14 }
```

# QUADRATIC PROBING

- Like in linear probing, colliding items are placed at an offset from the hash.
- But unlike in linear probing where the offsets are (1, 2, 3, ...), in quadratic probing the offsets are (1, 4, 9, ...).
- So the hash is  $H + i^2$ .

```
1  typedef Item;
2  typedef HashTable;
3
4  int hashfunction(char *key);
5
6  void hash_insert(HashTable *hashtable, Item *item) {
7      int hash = hashfunction(item->key);
8      if (hashtable->items[hash] == NULL)
9          hashtable->items[hash] = item;
10     else {
11         int offset;
12         for (int i = 1; i < hashtable->maxsize; ++i) {
13             offset = i * i;
14             if (hashtable->items[hash+offset] == NULL)
15                 break;
16         }
17         hashtable->items[hash+offset] = item;
18     }
19 }
```

# DOUBLE HASHING

- Double hashing also places colliding items at an offset from their hash, but the offset is obtained by another hash function.
- Generally there are two hash functions  $H_1$  and  $H_2$ . Offset is a multiple of  $H_2$  ( $key$ ).
- So the hash is  $H_1 + i.H_2$ .

```
1  typedef Item;
2  typedef HashTable;
3
4  int hashA(char *key);
5  int hashB(char *key);
6
7  void hash_insert(HashTable *hashtable, Item *item) {
8      int hash = hashA(item->key);
9      if (hashtable->items[hash] == NULL)
10         hashtable->items[hash] = item;
11     else {
12         int offset;
13         for (int i = 1; i < hashtable->maxsize; ++i) {
14             offset = i * hashB(item->key);
15             if (hashtable->items[hash+offset] == NULL)
16                 break;
17         }
18         hashtable->items[hash+offset] = item;
19     }
20 }
```

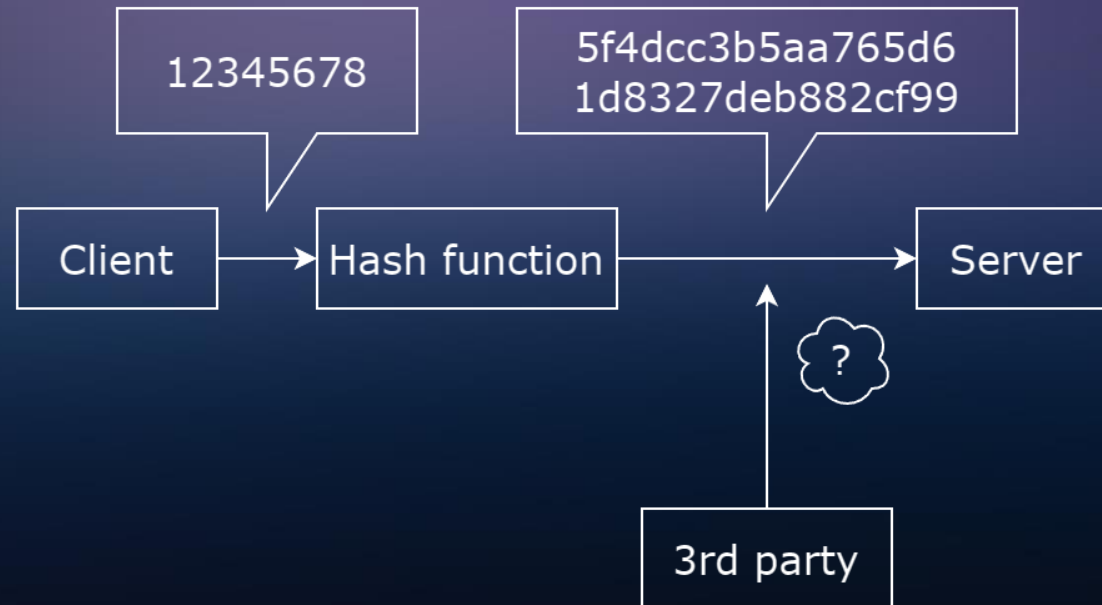
# COALESCED HASHING

- Like separate chaining, colliding items are inserted as nodes in a linked data structure at the hash'th bucket.
- But unlike separate chaining, the nodes of the data structure are stored in empty buckets of the hash table (like in open addressing).
- Since a node at hash position is connected to the colliding nodes which are present in other parts of the hash table, it takes lesser time to find the colliding nodes.

```
1  typedef Item;
2  typedef Node;
3  typedef HashTable;
4
5  int hashfunction(char *key);
6  Node *create_node(Item *item);
7  void list_insert(Node *head, Item *item);
8
9  void hash_insert(HashTable *hashtable, Item *item) {
10     int hash = hashfunction(item->key);
11     if (hashtable->items[hash] == NULL)
12         hashtable->items[hash] = create_node(item);
13     else {
14         int offset = 1;
15         for (; offset < hashtable->maxsize; ++offset)
16             if (hashtable->items[hash+offset] == NULL)
17                 break;
18         Node *newnode = create_node(item);
19         hashtable->items[hash+offset] = newnode;
20         list_insert(hashtable->items[hash], newnode);
21     }
22 }
```

# HASHING FOR SECURITY

- Hashing is widely used in cyber security to safeguard passwords and sensitive data.
- Instead of storing passwords in plain text, websites and applications convert them into hashes before storing them in databases.
- This way, even if the database is breached, attackers would encounter hashes that are extremely difficult to reverse-engineer back into the original passwords.





# CRYPTOGRAPHIC HASH FUNCTIONS

- Cryptography hash functions are mathematical algorithms that take an input (or message) of any length and produce a fixed-size output, commonly referred to as a hash value or hash code. These functions are designed to be *one-way*, meaning it's computationally infeasible to reverse the process and retrieve the original input from the hash value.

## CHARACTERISTICS

- Fixed Output Size
- Irreversibility
- Deterministic
- Collision Resistance

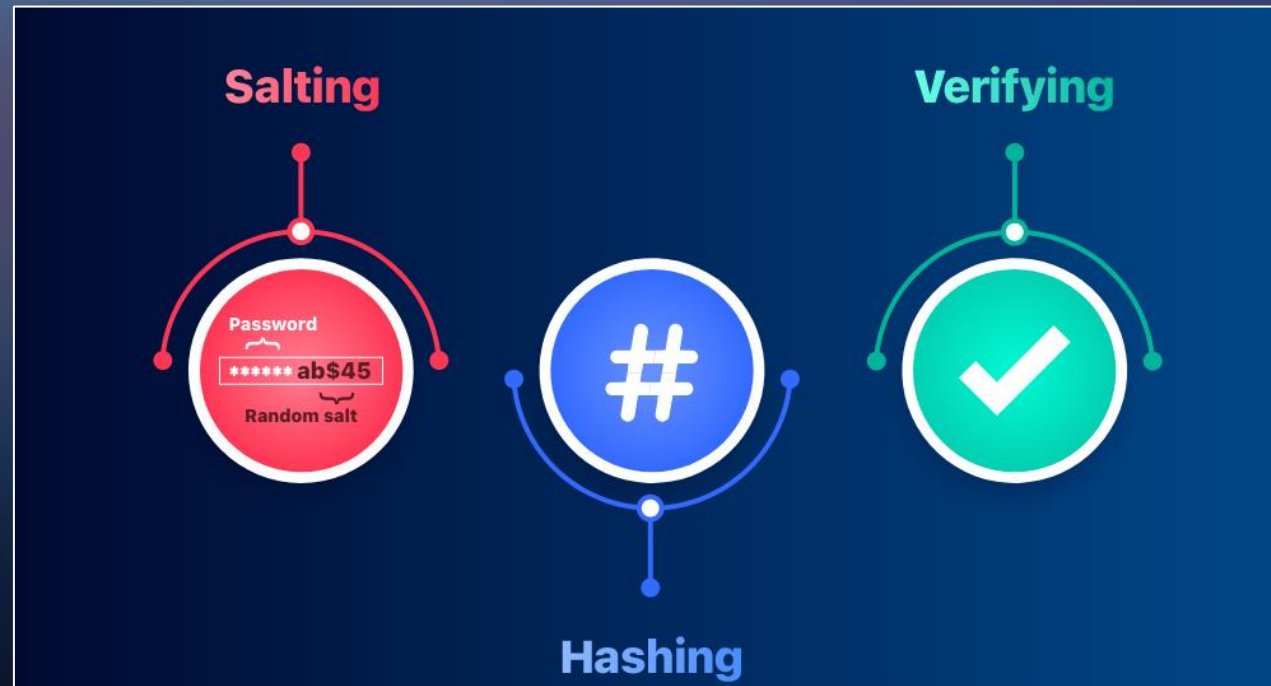
## EXAMPLES

- SHA-256 (Secure Hash Algorithm 256-bit)
- SHA-3
- BLAKE2
- Whirlpool
- RIPEMD-160



# SALTING

- Salting: Addition of a unique random value (salt) to each password before hashing further strengthens security by preventing attackers from using pre computed tables (rainbow tables) to crack passwords efficiently.



# ACKNOWLEDGEMENT

We would like to express our special gratitude to Mrs. Sanchita Pal Choudhuri who gave us the opportunity to do this wonderful presentation on hashing which helped us a lot in doing research on this topic and learn new topics regarding this course.

# REFERENCES

- [https://en.wikipedia.org/wiki/Hash function](https://en.wikipedia.org/wiki/Hash_function)
- [https://en.wikipedia.org/wiki/Open addressing](https://en.wikipedia.org/wiki/Open_addressing)
- [https://en.wikipedia.org/wiki/Cryptographic hash function](https://en.wikipedia.org/wiki/Cryptographic_hash_function)

**THANK YOU**