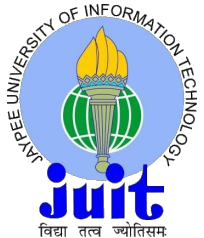


Data Structure

(Course Code: 25B11CI318)

Hashing

Mr. Sandeep Kumar Patel

Assistant Professor

Jaypee University of Information Technology

Waknaghat, Solan, HP-173234

Introduction to Hashing

Definition

Hashing is a technique used to store and retrieve data efficiently by mapping a key to a specific location in a hash table using a **hash function**.

The basic idea is:

$$\boxed{\text{Key} \xrightarrow{\text{Hash Function}} \text{Index}}$$

Example:

$$key = 25$$

If:

$$h(25) = 5$$

then the key is stored at:

$$\boxed{\text{Index } 5}$$

Why Do We Need Hashing?

Consider searching in an array:

$$A = [10,\ 25,\ 35,\ 42,\ 51,\ 68,\ 75]$$

Using Linear Search:

$$O(n)$$

Using Binary Search:

$$O(\log n)$$

if the array is sorted.

Hashing aims to provide:

$$\boxed{O(1) \text{ average search time}}$$

Hash Table

Definition

A **Hash Table** is a data structure consisting of an array of slots or buckets in which records are stored according to the value produced by a hash function.

Suppose the table size is:

$$m = 10$$

The valid indices are:

$$0, 1, 2, \ldots, 9$$

A hash function maps a key into this range:

$$h(key) \in [0, m-1]$$

Basic Hashing Example

Let:

$$m = 10$$

and hash function:

$$\boxed{h(k) = k \bmod 10}$$

Consider the keys:

$$23,\ 45,\ 18,\ 37$$

Their positions are:

$$h(23) = 23 \bmod 10 = 3$$
$$h(45) = 45 \bmod 10 = 5$$
$$h(18) = 18 \bmod 10 = 8$$
$$h(37) = 37 \bmod 10 = 7$$

Therefore:

$$\boxed{23 \to 3, \quad 45 \to 5, \quad 18 \to 8, \quad 37 \to 7}$$

Hash Function

Definition

A **hash function** is a function that converts a key into an index of the hash table.

General form:

$$\boxed{h(k) = \text{index}}$$

where:

- k = key
- $h(k)$ = hash value
- m = hash table size

The hash value should satisfy:

$$\boxed{0 \leq h(k) < m}$$

Properties of a Good Hash Function

A good hash function should:

1. Be easy and fast to compute.
2. Distribute keys uniformly.
3. Minimize collisions.
4. Always produce a valid table index.
5. Produce the same index for the same key.

Goal

A good hash function tries to distribute keys **uniformly** across the table.

Division Method

One of the simplest hash functions is:

$$\boxed{h(k) = k \bmod m}$$

where m is the table size.

Example:

$$m = 11$$

For:

$$k = 45$$

we get:

$$h(45) = 45 \bmod 11$$

Multiplication Method

The multiplication method uses:

$$\boxed{h(k) = \lfloor m(kA \bmod 1) \rfloor}$$

where:

$$0 < A < 1$$

and m is the table size.

The fractional part of kA is multiplied by the table size.

Advantage

The method can provide good distribution without requiring special properties of the table size.

What is a Collision?

Definition

A **collision** occurs when two or more different keys are mapped to the same hash table index.

Suppose:

$$h(k) = k \bmod 10$$

For:

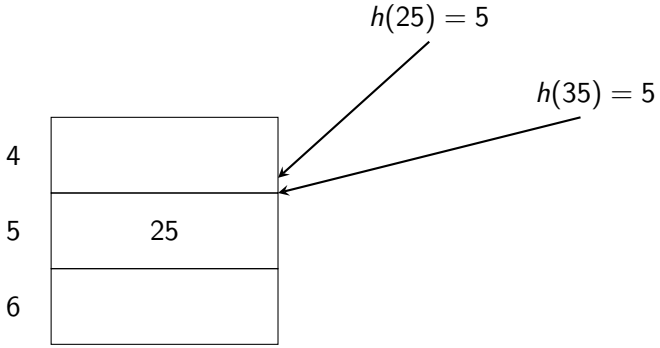
$$k_1 = 25$$

$$h(25) = 5$$

and:

$$k_2 = 35$$

Collision: Visual Example



Collision at index 5

$$h(25) = h(35) = 5$$

Collision Resolution Techniques

When a collision occurs, we need a strategy to store both keys.

Major collision resolution techniques are:

1. **Separate Chaining**
2. **Open Addressing**
   - Linear Probing
   - Quadratic Probing
   - Double Hashing

$$\boxed{\text{Collision} \Rightarrow \text{Collision Resolution}}$$

Separate Chaining

Definition

In **Separate Chaining**, each position of the hash table contains a linked list of all keys that hash to that position.

Example:

$$h(k) = k \bmod 10$$

Keys:

$$25,\ 35,\ 45$$

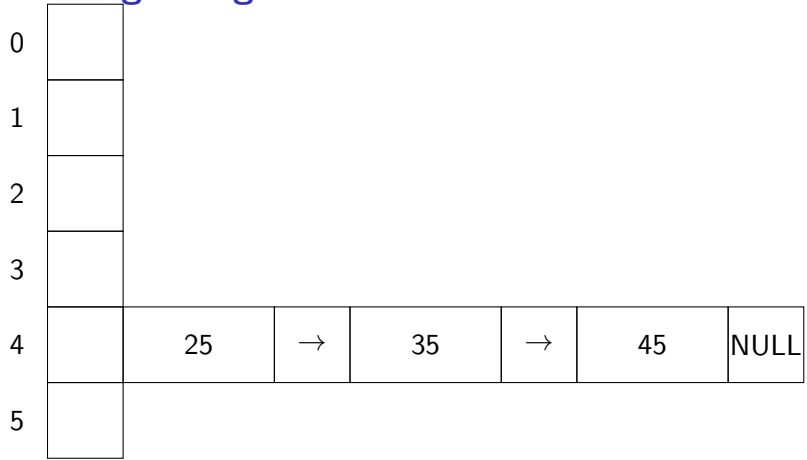
All produce:

$$5$$

Therefore:

$$5 \rightarrow 25 \rightarrow 35 \rightarrow 45$$

Separate Chaining: Diagram



$5 \rightarrow 25 \rightarrow 35 \rightarrow 45 \rightarrow NULL$

Separate Chaining: C Structure

```
struct Node
{
    int key;
    struct Node *next;
};

struct Node *hashTable[SIZE];
```

Each table entry points to the beginning of a linked list.

Hash Table Entry $\longrightarrow$ Linked List

Separate Chaining: Insertion

```
void insert(int key)
{
    int index = key % SIZE;

    struct Node *newNode;
    newNode = malloc(sizeof(struct Node));

    newNode->key = key;

    newNode->next = hashTable[index];
    hashTable[index] = newNode;
}
```

The key is inserted into the linked list corresponding to its hash index.

Open Addressing

Definition

In **Open Addressing**, all elements are stored directly inside the hash table.

When a collision occurs, another empty position is searched according to a probing sequence.

Major techniques:

- Linear Probing
- Quadratic Probing
- Double Hashing

One key occupies one table slot

Linear Probing

Definition

Linear Probing resolves a collision by checking the next consecutive positions until an empty slot is found.

The probing function is:

$$\boxed{h_i(k) = (h(k) + i) \bmod m}$$

where:

$$i = 0, 1, 2, \ldots$$

Linear Probing: Example

Let:

$$m = 10$$

and:

$$h(k) = k \bmod 10$$

Insert:

$$25,\ 35,\ 45$$

First:

$$h(25) = 5$$

Store at index 5. For 35:

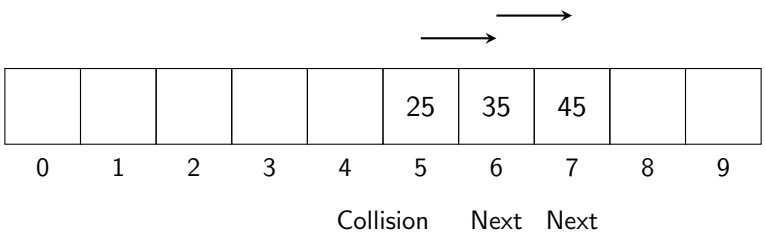
$$h(35) = 5$$

Index 5 is occupied. Check:

$$6$$

Store 35 at index 6.

Linear Probing: Diagram



$5 \rightarrow 6 \rightarrow 7$

Primary Clustering

Primary Clustering

Linear probing tends to create groups of consecutive occupied positions called **primary clusters**.

Example:

$$[\_,\ 10,\ 20,\ 30,\ 40,\ \_,\ 60,\ \_,\ \_]$$

The consecutive occupied positions form a cluster.

Problem

As clusters become larger, the number of probes required for insertion and search can increase.

Quadratic Probing

Definition

Quadratic Probing uses quadratic increments instead of checking consecutive positions.

The probing function is:

$$\boxed{h_i(k) = (h(k) + c_1 i + c_2 i^2) \bmod m}$$

A common simplified form is:

$$\boxed{h_i(k) = (h(k) + i^2) \bmod m}$$

Quadratic Probing: Example

Let:

$$m = 10$$

and:

$$h(k) = k \bmod 10$$

Suppose:

$$h(k) = 5$$

and index 5 is occupied.
The probes are:

$$i = 0 :$$

Quadratic Probing: Advantage

Compared with Linear Probing:

- It does not usually inspect consecutive slots.
- Reduces primary clustering.
- Spreads probes over the table.

Important

Quadratic probing can still suffer from **secondary clustering**.

Double Hashing

Definition

Double Hashing uses a second hash function to determine the probing step size.

The probing function is:

$$\boxed{h_i(k) = (h_1(k) + i\, h_2(k)) \bmod m}$$

where:

$$h_1(k) = \text{Primary hash function}$$

$$h_2(k) = \text{Secondary hash function}$$

Double Hashing: Example

Let:

$$m = 13$$

Primary hash:

$$h_1(k) = k \bmod 13$$

Secondary hash:

$$h_2(k) = 7 - (k \bmod 7)$$

Suppose:

$$k = 26$$

Then:

Why Double Hashing?

Double hashing provides a different probe sequence for different keys.

It generally:

- Reduces clustering.
- Provides better distribution than linear probing.
- Reduces secondary clustering.

Key Idea

The second hash function determines the distance between successive probes.

Collision Resolution: Comparison

Technique	Basic Idea	Clustering	Storage
Chaining	Linked List	No primary clustering	Outside table
Linear Probing	Next slot	Primary	Inside table
Quadratic Probing	Quadratic jump	Secondary	Inside table
Double Hashing	Second hash	Low	Inside table

Remember

- Chaining uses linked lists.
- Open addressing stores everything in the table.
- Linear probing is simplest but suffers from clustering.
- Double hashing generally provides better probe distribution.

Operations on a Hash Table

Main operations:

1. Insertion
2. Searching
3. Deletion

For a good hash function and reasonable load factor:

$$\boxed{\text{Average Search} = O(1)}$$

$$\boxed{\text{Average Insertion} = O(1)}$$

$$\boxed{\text{Average Deletion} = O(1)}$$

Worst case:

Load Factor

The **load factor** measures how full a hash table is.

It is defined as:

$$\boxed{\alpha = \frac{n}{m}}$$

where:

- n = number of stored elements
- m = size of hash table

Example:

$$n = 7, \qquad m = 10$$

Then:

$$\alpha = \frac{7}{10} = 0.7$$

$$\boxed{\alpha = 0.7}$$

Load Factor and Performance

As the load factor increases:

$$\alpha \uparrow$$

the number of collisions generally increases.

Therefore:

$$\boxed{\text{Higher Load Factor} \Rightarrow \text{More Collisions}}$$

and:

$$\boxed{\text{More Collisions} \Rightarrow \text{More Probing/Search}}$$

A hash table may be resized when the load factor becomes too large.

Simple Hash Function in C

```c
#define SIZE 10

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

Example:

```c
hashFunction(25);
```

returns:

$$25 \bmod 10 = 5$$

Therefore:

$$\boxed{25 \rightarrow 5}$$

Linear Probing: C Implementation

```
void insert(int hashTable[], int key)
{
    int index = key % SIZE;

    while(hashTable[index] != -1)
    {
        index = (index + 1) % SIZE;
    }

    hashTable[index] = key;
}
```

The table is searched sequentially until an empty position is found.

Linear Probing: Search

```
int search(int hashTable[], int key)
{
    int index = key % SIZE;
    int start = index;
    while(hashTable[index] != -1)
    {
        if(hashTable[index] == key)
            return index;
        index = (index + 1) % SIZE;
        if(index == start)
            break;
    }

    return -1;
}
```

Complete Example

Let:

$$m = 10$$

and:

$$h(k) = k \bmod 10$$

Insert:

$$23,\ 43,\ 13,\ 27$$

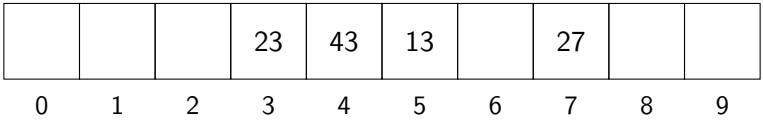
Hash values:

$$h(23) = 3$$
$$h(43) = 3$$
$$h(13) = 3$$
$$h(27) = 7$$

Using Linear Probing:

$$23 \to 3$$
$$43 \to 4$$
$$13 \to 5$$

Complete Example: Table



$23 \rightarrow 3, \quad 43 \rightarrow 4, \quad 13 \rightarrow 5, \quad 27 \rightarrow 7$

Complexity of Hashing

Operation	Average Case	Worst Case
Search	$O(1)$	$O(n)$
Insertion	$O(1)$	$O(n)$
Deletion	$O(1)$	$O(n)$

Why Worst Case $O(n)$?

If many keys collide and form a long chain or long probe sequence, searching may require examining many elements.

Applications of Hashing

Hashing is widely used in:

- Database indexing
- Symbol tables in compilers
- Dictionaries
- Sets and maps
- Caches
- Password lookup systems
- Duplicate detection
- File systems

Fast Key $\rightarrow$ Value Lookup

Advantages of Hashing

- Average-case search is $O(1)$.
- Fast insertion and deletion.
- Efficient for large collections of key-value data.
- Direct access using a calculated index.
- Widely applicable in real-world systems.

Limitations of Hashing

- Collisions are unavoidable in general.
- Performance depends on the hash function.
- Poor hash functions lead to clustering.
- Hash tables require careful management of load factor.
- Data is not naturally stored in sorted order.
- Resizing can be expensive.

Hashing: Key Takeaways

1. Hashing maps a key to a table index.

Hashing: Key Takeaways

1. Hashing maps a key to a table index.
2. A hash function determines the index.

Hashing: Key Takeaways

1. Hashing maps a key to a table index.
2. A hash function determines the index.
3. Two keys mapping to the same index cause a collision.

Hashing: Key Takeaways

1. Hashing maps a key to a table index.
2. A hash function determines the index.
3. Two keys mapping to the same index cause a collision.
4. Collisions can be handled using:
   - Separate Chaining
   - Linear Probing
   - Quadratic Probing
   - Double Hashing

Hashing: Key Takeaways

1. Hashing maps a key to a table index.
2. A hash function determines the index.
3. Two keys mapping to the same index cause a collision.
4. Collisions can be handled using:
   - Separate Chaining
   - Linear Probing
   - Quadratic Probing
   - Double Hashing
5. Good hashing provides average $O(1)$ operations.

Hashing: Key Takeaways

1. Hashing maps a key to a table index.
2. A hash function determines the index.
3. Two keys mapping to the same index cause a collision.
4. Collisions can be handled using:
   - Separate Chaining
   - Linear Probing
   - Quadratic Probing
   - Double Hashing
5. Good hashing provides average $O(1)$ operations.
6. High load factor generally increases collisions.

Remember

Hashing = Calculate the Location Instead of Searching

Thank You

Questions?