Session 3

Hashing and Matrix

pandiyanalex1000@gmail.com

CONTENTS

pandiyanalex1000@gmail.com

01

Introduction to Hashing

pandiyanalex1000@gmail.com

Definition and Purpose of Hashing

01

Hashing is a process of converting input data (of any size) into a fixed- length value, typically called a hash code or hash value.

02

To enable fast data retrieval, comparison, or verification.

03

Ensures that data is mapped efficiently for storage and access.

pandiyanalex1000@gmail.com

Real world Applications of Hashing

01

Password Storage Securely stores passwords by hashing them so that the original password is not directly saved.

02

Data Indexing Quickly locates records in large databases using hash tables.

03

Blockchain Ensures data integrity and links blocks in a secure manner.

04

Check sums Verifies file integrity during transfers or downloads.

pandiyanalex1000@gmail.com

02

Hash Function

Hash Functions

Hash Function :

A mathematical function that transforms input into a fixed-size string or number.

Characteristics :

- Deterministic: The same input always produces the same hash.
- Uniform Distribution: Distributes data evenly to minimize collisions.
- Fast Computation: Efficiently computes the hash value for large datasets.
- Avalanche Effect: A small change in input results in a significant change in the hash output.
- Pre-image Resistance: Difficult to reverse-engineer the original input from the hash.

Example of Hash Function

★ MD5: Fast but insecure; used in non-critical checksums.

★ SHA-1: Deprecated due to vulnerabilities but widely used historically.

★ SHA-256: Part of the SHA-2 family; secure and widely used in cryptography.

03

Applications of Hashing

Applications of Hashing

Data Structures:

- **Hash Tables:** Store key-value pairs for fast lookups, like in dictionaries or associative arrays.
  - Example: HashMap in Java, dict in Python.

Cryptographic Hashing:

- Ensures data integrity and security.
- Used in:
  - **Password Hashing**: Stores hashed passwords with added salts.
  - **Digital Signatures**: Verifies the authenticity of data.

File Integrity Verification:

- Hashes verify that files have not been altered during transfers or downloads.
  - Example: Using SHA-256 checksums for software installations.

Applications of Hashing

Blockchain and Cryptocurrency:

- ➢ Hashing links blocks securely in a blockchain.
- ➢ Ensures immutability and data integrity.

Load Balancing:

- ➢ Hashing distributes requests to servers in a balanced way.
  - Example: Consistent hashing in distributed systems.

Caching Mechanisms:

- ➢ Hashes identify cached items efficiently in web applications.

04

Hash Collisions

Hash Collisions

What Are Hash Collisions?

- Occurs when two different inputs produce the same hash value.
- This is an inherent limitation of hash functions since the input space is infinite, but the output space is finite.

Why Do They Occur?

- Poor design of hash functions.
- The fixed size of hash output.

Methods to Resolve Hash Collision

- **Chaining :** Uses linked lists to store multiple values in the same bucket.
- **Open Addressing :**
  - **Linear Probing:** Searches the next available slot.
  - **Quadratic Probing:** Uses a quadratic formula to find the next slot.
  - **Double Hashing:** Uses a secondary hash function to find the next slot.

05

Types of Hashing

Types of Hashing

1. **Static Hashing:**

   - The size of the hash table is fixed.
   - Simple but may lead to wasted space or excessive collisions.

Advantages:

- Simple to implement.
- Predictable memory usage.

Disadvantages:

- May result in **wasted space** if the table is too large.
- Can lead to **excessive collisions** if the table is too small.

Types of Hashing

2. Dynamic Hashing :

- The hash table **dynamically grows or shrinks** as the number of elements changes.
- Useful for applications where the data size is unpredictable.

Advantages :

- Reduces wasted space.
- Handles **load factor** issues dynamically.

Disadvantages :

- Slightly more complex to implement.
- Overhead of resizing or maintaining additional structures.

Types :

- **Extendible Hashing:** Dynamically adjusts the table size using directory pointers.
- **Linear Hashing:** Increases table size incrementally without rebuilding the entire structure

06

Hashing in Data Structures

priyanalex1000@gmail.com

Hash Tables and their Efficiency

- A **hash table** is a data structure that uses a **hash function** to map keys to indices in an array for efficient data storage and retrieval.
- **Efficiency:**
  - **Average Case:** O(1) time for insert, search, and delete operations, thanks to direct indexing using hash codes.
  - **Worst Case:** O(n) when many keys collide (mapped to the same bucket) and are stored in a linked list or tree structure.

Example (Java):

Suppose we store student IDs as keys and names as values.

```java
HashMap<Integer, String> studentMap = new HashMap<>();
studentMap.put(101, "Alice");
studentMap.put(102, "Bob");
studentMap.put(103, "Charlie");
```

To find "Alice," the hash function computes an index based on 101, allowing instant access.

HashMap

What is a Map?

- A data structure for storing **key-value pairs**.
- **Key**: Unique identifier.
- **Value**: Data or details associated with the key.

Key Features

- **Fast Lookups**: Average time complexity **O(1)** for insertion, deletion, and search.
- **Unique Keys**: No duplicate keys; values can repeat.
- **Dynamic Size**: Grows or shrinks as entries are added or removed.

Properties

- Keys CANNOT repeat, has to be unique
- INSERT not possible if the key already exists
- UPDATE Value for a Key
- DELETE based on Key
- POSITION doesn't matter
- SEARCH based on Keys (generally, this is more optimal) or values
- SORT - may be sorted based on Key

HashMap

1. **In Java**
   - ➢ Implements a hash table for storing **key-value pairs**.
   - ➢ Keys must be unique, and values can be duplicate.
   - ➢ Handles collisions using **chaining** (linked lists or trees in modern Java).

Example:

```java
HashMap<String, Integer> ageMap = new HashMap<>();
ageMap.put("John", 25);
ageMap.put("Jane", 30);
System.out.println(ageMap.get("John")); // Output: 25
```

1. **In Python**
   - ➢ **dict:** A dictionary for key-value storage.

Example:

```python
age_map = {"John": 25, "Jane": 30}
print(age_map["John"])  # Output: 25
```

HashSet

1. **In Java :**
   - Internally uses a HashMap to store **unique elements**.
   - Does not allow duplicate values.

   Example:

```java
HashSet<String> colors = new HashSet<>();
colors.add("Red");
colors.add("Blue");
colors.add("Red"); // Duplicate, ignored
System.out.println(colors); // Output: [Red, Blue]
```

1. **In Python :**

   set: A collection for unique elements.

   Example

```python
colors = {"Red", "Blue", "Red"}  # Duplicate ignored
print(colors)  # Output: {'Red', 'Blue'}
```

Custom Hash Functions

- Custom hash functions are useful when keys are **complex objects** (e.g., a combination of multiple fields).
- A good hash function:
  - Ensures uniform distribution of hash values.
  - Avoids collisions as much as possible.

CRUD Operations

- set(key, value): Adds a new key-value pair to the map or updates the value of an existing key.
- get(key): Returns the value associated with the specified key, or undefined if the key is not found in the map.
- has(key): Returns a boolean indicating whether the specified key is present in the map.
- delete(key): Removes the key-value pair associated with the specified key from the map.
- clear(): Removes all key-value pairs from the map.

07

Activity 1

pandiyanalex1000@gmail.com

Activity 1

</> Intersection of Two Arrays - In Session

</> Single Number

</> Find Distinct Elements

</> Union of Arrays with Duplicates