535. Encode and Decode TinyURL **String** Medium Design Hash Table **Hash Function**

Problem Description

is similar to services like TinyURL that make long URLs more manageable and easier to share. The problem specifically requires implementing a class with two methods: one to encode a URL and one to decode a previously encoded URL. Intuition

The problem asks us to design a system that can encode a long URL into a short URL and decode it back to the original URL. This

Encoding:

To solve this problem, we need to establish a system that can map a long URL to a unique short identifier and be able to retrieve the original URL using that identifier. The core idea behind the solution is to use a hash map (or dictionary in Python) to keep

• Each time we want to encode a new URL, we increment an index that acts as a unique identifier for each URL.

Here's the step-by-step reasoning for arriving at the solution:

track of the association between the encoded short URLs and the original long URLs.

• Then, we add an entry to our hash map where the key is the string representation of the current index and the value is the long URL. • The encoded tiny URL is generated by concatenating a predefined domain (e.g., "https://tinyurl.com/") with the index. **Decoding:**

• To decode, we can extract the index from the end of the short URL. This index is the key to our hash map. • We then use this key to look up the associated long URL in our hash map and return it.

Solution Approach The implementation uses a simple yet effective approach, based on a hash map and an incremental counter to correlate long

URLs with their tiny counterparts.

• Hash Map (defaultdict in Python): A hash map is used to store and quickly retrieve the association between the unique identifier (idx) and the original long URL.

Data Structures:

Algorithm:

• A hash map self.m is initialized to store the mapping between a short URL suffix (a unique index) and the original long URL. • self.idx is initialized to 0 which is used as a counter to create unique identifiers for each URL.

• Increment the self.idx counter to generate a new unique identifier for a new long URL. • Store the long URL in the hash map with the string representation of the incremental index as the key.

The codec class is implemented in Python with the following methods:

• Generate the tiny URL by concatenating the predefined domain self.domain with the current index. • Return the full tiny URL. The encode function can be articulated with a small formula where longUrl is mapped to "https://tinyurl.com/" +

Initialization (__init__):

Encode Method (encode):

Decode Method (decode):

• Return the long URL.

return self.m[idx]

str(self.idx). self.idx += 1

self.m[str(self.idx)] = longUrl return f'{self.domain}{self.idx}'

This process can be described as retrieving self.m[idx], where idx is the last part of shortUrl. idx = shortUrl.split('/')[-1]

Extract the unique identifier from the short URL by splitting it at the '/' and taking the last segment.

• The identifier is then used to find the original long URL from the hash map.

Let's demonstrate the encoding and decoding process with a simple example:

self.domain = "https://tinyurl.com/"

Map the current index to the long URL

Extract the identifier from the URL

self.m[str(self.idx)] = longUrl

idx = shortUrl.split('/')[-1]

Retrieve the original long URL

Increment the index to create a new identifier

def encode(self. longUrl):

def decode(self, shortUrl):

return self.m[idx]

Solution Implementation

from collections import defaultdict

self.url mapping = defaultdict()

def encode(self, longUrl: str) -> str:

"""Encodes a URL to a shortened URL."""

return f'{self.domain}{self.index}'

short url = codec.encode("https://www.example.com")

private String domain = "https://tinyurl.com/";

* @param longUrl The original long URL to be encoded

* Encodes a URL to a shortened URL.

public String encode(String longUrl) {

* @return The encoded short URL

urlMap[key] = longUrl;

return domain + key;

// Decodes a shortened URL to its original URL.

// Import necessary components for working with maps

// Declare a counter to generate unique keys for each URL encoded

// Find the position of the last '/' in the short URL using URL class

// Use the key to retrieve the original long URL from the map

const longUrl: string | undefined = urlMap.get(key);

const key: string = shortUrlObj.pathname.substring(1); // Remove the leading '/'

const urlMap = new Map<string, string>();

// Encodes a URL to a shortened URL.

function encode(longUrl: string): string {

// Define the base domain for the shortened URL

const domain: string = "https://tinyurl.com/";

// Decodes a shortened URL to its original URL.

const shortUrlObj = new URL(shortUrl);

function decode(shortUrl: string): string {

import { URL } from "url";

let counter: number = 0;

if (longUrl) {

return longUrl;

// Find the position of the last '/' in the short URL

std::string decode(std::string shortUrl) {

def decode(self, shortUrl: str) -> str:

self.url mapping[str(self.index)] = longUrl

def init (self):

self.index += 1

Python

class Codec:

self.idx += 1

• Unique Identifier: By using a simple counter, each URL gets a unique identifier which essentially works as a key, preventing collisions between different long URLs. DbSetti does not rely on hashing functions or complex encoding schemes, reducing overhead and complexity.

Example Walkthrough

Patterns:

and decoding functions. This straightforward approach is easy to understand and implement, requiring only basic data manipulation. It does not involve

any complex hash functions, avoids collisions, and ensures consistent O(1) performance for basic operations.

After we initiate our codec class, it might look something like this: class Codec: def init (self): $self.m = \{\}$ self.idx = 0

Imagine we have the following URL to encode: https://www.example.com/a-very-long-url-with-multiple-sections.

• Direct Mapping: The system relies on direct mappings from unique identifiers to original URLs, allowing O(1) time complexity for both encoding

Generate and return the shortened URL return f'{self.domain}{self.idx}'

We take the long URL https://www.example.com/a-very-long-url-with-multiple-sections.

Let's go through the actual encoding and decoding steps with our example URL:

Initialize a dictionary to store the long URL against unique indexes

self.domain = 'https://tinyurl.com/' # The domain prefix for the short URL

self.index = 0 # A counter to create unique keys for URL

Increment the index to get a unique key for a new URL

Return the domain concatenated with the unique index

Store the long URL in the dictionary with the new index as key

// Domain to prepend to the unique identifier creating the shortened URL

// Increment the indexCounter to get a unique key for this URL

// Associate the kev with the original long URL in the hashmap

// Construct the short URL by appending the key to the predefined domain

Since self.idx starts at 0, after encoding our first URL, it will become 1.

Encoding the URL:

```
    We add the long URL to the hash map with the key '1'.

   • The method encode returns a tiny URL, which will be "https://tinyurl.com/1".
   Decoding the URL:

    Now, when we want to access the original URL, we take the tiny URL "https://tinyurl.com/1".

   • The method decode will extract the identifier '1' which is the last segment after splitting the URL by '/'.
   ○ It will then look up this index in our hash map to find the original URL, which is https://www.example.com/a-very-long-url-with-multiple-
     sections.
   • The decode method will return this long URL.
By following this simple example, we've seen how the unique identifier helps in associating a long URL with a shortened version,
and how easy it becomes to retrieve the original URL when needed. Each encode operation generates a new, unique tiny URL,
and each decode operation precisely retrieves the corresponding original URL using this mechanism.
```

"""Decodes a shortened URL to its original URL.""" # Extract the index from the short URL by splitting on '/' index = shortUrl.split('/')[-1] # Use the index to retrieve the corresponding long URL from the dictionary return self.url_mapping[index] # Example of Usage: # codec = Codec()

public class Codec { // Map to store the index-to-URL mappings private Map<String, String> indexToUrlMap = new HashMap<>(); // Counter to generate unique keys for shortened URLs

private int indexCounter = 0;

import java.util.HashMap;

import java.util.Map;

print(codec.decode(short_url))

Java

/**

```
String key = String.valueOf(++indexCounter);
        // Store the long URL with the generated key in the map
        indexToUrlMap.put(key, longUrl);
        // Return the complete shortened URL by appending the key to the domain
        return domain + key;
    /**
     * Decodes a shortened URL to its original URL.
     * @param shortUrl The shortened URL to be decoded
     * @return The original long URL
     */
    public String decode(String shortUrl) {
        // Find the position just after the last '/' character in the shortened URL
        int index = shortUrl.lastIndexOf('/') + 1;
        // Extract the key from the short URL and look it up in the map to retrieve the original URL
        String key = shortUrl.substring(index);
        return indexToUrlMap.get(key);
// The Codec class may be used as follows:
// Codec codec = new Codec();
// String shortUrl = codec.encode("https://www.example.com");
// String longUrl = codec.decode(shortUrl);
C++
#include <string>
#include <unordered_map>
class Solution {
public:
    // Encodes a URL to a shortened URL.
    std::string encode(std::string longUrl) {
        // Convert the current counter value to a string to create a unique key
        std::string key = std::to_string(++counter);
```

```
std::size_t lastSlashPos = shortUrl.rfind('/') + 1;
        // Extract the key from the short URL based on the position of the last '/'
        // and use it to retrieve the original long URL from the hashmap
        return urlMap[shortUrl.substr(lastSlashPos, shortUrl.size() - lastSlashPos)];
private:
    // Hashmap to store the association between the unique key and the original long URL
    std::unordered_map<std::string, std::string> urlMap;
    // Counter to generate unique keys for each URL encoded
    int counter = 0;
    // The base domain for the shortened URL
    std::string domain = "https://tinyurl.com/";
};
// Usage example:
// Solution solution;
// std::string shortUrl = solution.encode("https://example.com");
// std::string longUrl = solution.decode(shortUrl);
TypeScript
```

```
// Convert the current counter value to a string to create a unique key
counter++:
const key: string = counter.toString();
// Associate the kev with the original long URL in the map
urlMap.set(key, longUrl);
// Construct the short URL by appending the key to the predefined domain
return domain + key;
```

// Create a Map to store the association between the unique key and the original long URL

```
} else {
        throw new Error("Short URL does not correspond to a known long URL");
// Note: Usage example is not included as we are defining things in the global scope
from collections import defaultdict
class Codec:
   def init (self):
       # Initialize a dictionary to store the long URL against unique indexes
       self.url mapping = defaultdict()
       self.index = 0 # A counter to create unique keys for URL
       self.domain = 'https://tinyurl.com/' # The domain prefix for the short URL
   def encode(self, longUrl: str) -> str:
       """Encodes a URL to a shortened URL."""
       # Increment the index to get a unique key for a new URL
       self.index += 1
       # Store the long URL in the dictionary with the new index as key
       self.url mapping[str(self.index)] = longUrl
       # Return the domain concatenated with the unique index
       return f'{self.domain}{self.index}'
```

Time Complexity encode: The encode method has a time complexity of 0(1) because it only performs simple arithmetic incrementation and

short url = codec.encode("https://www.example.com")

def decode(self, shortUrl: str) -> str:

index = shortUrl.split('/')[-1]

return self.url_mapping[index]

Example of Usage:

print(codec.decode(short_url))

Time and Space Complexity

codec = Codec()

"""Decodes a shortened URL to its original URL."""

Extract the index from the short URL by splitting on '/'

Use the index to retrieve the corresponding long URL from the dictionary

one assignment operation, neither of which depend on the size of the input.

constant time operation since the URL length is fixed ("https://tinyurl.com/" part), and a dictionary lookup, which is generally considered constant time given a good hash function and well-distributed keys.

decode: The decode method has a time complexity of 0(1) because it performs a split operation on a URL which is a

Space Complexity • The space complexity of the overall Codec class is 0(N) where N is the number of URLs encoded. This is because each newly encoded URL adds one additional entry to the dictionary (self.m), which grows linearly with the number of unique long URLs processed.