**Module** **22.** **Hash Map**

**Learning Objectives**

* Understand the primarily implemented through dictionaries, focuses on understanding their structure, functionality, and applications.
* Define the hash maps store unique key-value pairs, enabling efficient data access through a hash function that determines the index of each key in an array.
* Hash maps offer unique keys, mutability for dynamic data management, and efficient average-case time complexity of O (1) for operations like insertion, deletion, and retrieval.
* Explain hash maps that are indexed data structure and hash function that is the core of implementing a hash map.
* Implementation of hash map for optimizing search, e.g., dictionaries are example of hash maps.

1. **Introduction**

Hash maps are indexed data structures. A hash map makes use of a hash function to compute an index with a key into an array of buckets or slots. Its value is mapped to the bucket with the corresponding index. The key is unique and immutable. Think of a hash map as a cabinet having drawers with labels for the things stored in them. For example, storing user information- consider email as the key, and we can map values corresponding to that user such as the first name, last name etc to a bucket.

Hash function is the core of implementing a Hash Map. It takes in the key and translates it to the index of a bucket in the bucket list. Ideal hashing should produce a different index for each key. However, collisions can occur. When hashing gives an existing index, we can simply use a bucket for multiple values by appending a list or by rehashing.

In Python, dictionaries are examples of hash maps. We’ll see the implementation of hash map from scratch in order to learn how to build and customize such data structures for optimizing search.

The Hash Map design will include the following functions:

A Hash Map is a form of Hash Table data structure that usually holds a large number of entries. Using a Hash Map we can search, add, modify, and remove entries really fast.

* set\_val(key, value): Inserts a key-value pair into the hash map. If the value already exists in the hash map, update the value.
* get\_val(key): Returns the value to which the specified key is mapped, or “No record found” if this map contains no mapping for the key.
* delete\_val(key): Removes the mapping for the specific key if the hash map contains the mapping for the key.

Below is the implementation.

class HashTable:

# Create empty bucket list of given size

def \_\_init\_\_(self, size):

self.size = size

self.hash\_table = self.create\_buckets()

def create\_buckets(self):

return [[] for \_ in range(self.size)]

# Insert values into hash map

def set\_val(self, key, val):

# Get the index from the key

# using hash function

hashed\_key = hash(key) % self.size

# Get the bucket corresponding to index

bucket = self.hash\_table[hashed\_key]

found\_key = False

for index, record in enumerate(bucket):

record\_key, record\_val = record

# check if the bucket has same key as

# the key to be inserted

if record\_key == key:

found\_key = True

break

# If the bucket has same key as the key to be inserted,

# Update the key value

# Otherwise append the new key-value pair to the bucket

if found\_key:

bucket[index] = (key, val)

else:

bucket.append((key, val))

# Return searched value with specific key

def get\_val(self, key):

# Get the index from the key using

# hash function

hashed\_key = hash(key) % self.size

# Get the bucket corresponding to index

bucket = self.hash\_table[hashed\_key]

found\_key = False

for index, record in enumerate(bucket):

record\_key, record\_val = record

# check if the bucket has same key as

# the key being searched

if record\_key == key:

found\_key = True

break

# If the bucket has same key as the key being searched,

# Return the value found

# Otherwise indicate there was no record found

if found\_key:

return record\_val

else:

return "No record found"

# Remove a value with specific key

def delete\_val(self, key):

# Get the index from the key using

# hash function

hashed\_key = hash(key) % self.size

# Get the bucket corresponding to index

bucket = self.hash\_table[hashed\_key]

found\_key = False

for index, record in enumerate(bucket):

record\_key, record\_val = record

# check if the bucket has same key as

# the key to be deleted

if record\_key == key:

found\_key = True

break

if found\_key:

bucket.pop(index)

return

# To print the items of hash map

def \_\_str\_\_(self):

return "".join(str(item) for item in self.hash\_table)

hash\_table = HashTable(50)

# insert some values

hash\_table.set\_val('gfg@example.com', 'some value')

print(hash\_table)

print()

hash\_table.set\_val('portal@example.com', 'some other value')

print(hash\_table)

print()

# search/access a record with key

print(hash\_table.get\_val('portal@example.com'))

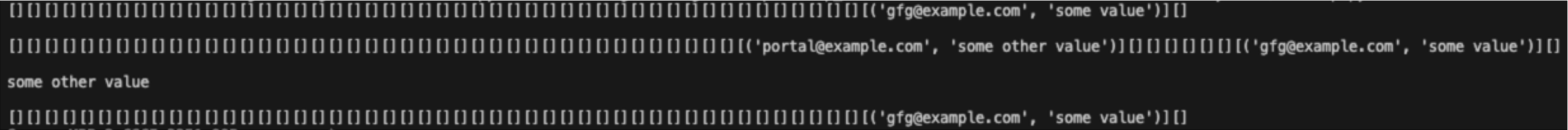
print()

# delete or remove a value

hash\_table.delete\_val('portal@example.com')

print(hash\_table)

Output



**Time Complexity:**

Memory index access takes constant time and hashing takes constant time. Hence, the search complexity of a hash map is also constant time, that is, O(1).

1. **Hash Maps**

Hash Maps are used to find detailed information about something. In the implementation below, people are stored in a Hash Map. A person can be looked up using a person’s unique social security number (the Hash Map key), and then we can see that person’s name (the Hash Map value).

A screenshot of a phone

Description automatically generated

class SimpleHashMap:

def \_\_init\_\_(self, size=100):

self.size = size

self.buckets = [[] for \_ in range(size)] # A list of buckets, each is a list (to handle collisions)

def hash\_function(self, key):

# Sum only the numerical values of the key, ignoring non-numeric characters

numeric\_sum = sum(int(char) for char in key if char.isdigit())

return numeric\_sum % 10 # Perform modulo 10 on the sum

def put(self, key, value):

# Add or update a key-value pair

index = self.hash\_function(key)

bucket = self.buckets[index]

for i, (k, v) in enumerate(bucket):

if k == key:

bucket[i] = (key, value) # Update existing key

return

bucket.append((key, value)) # Add new key-value pair if not found

def get(self, key):

# Retrieve a value by key

index = self.hash\_function(key)

bucket = self.buckets[index]

for k, v in bucket:

if k == key:

return v

return None # Key not found

def remove(self, key):

# Remove a key-value pair

index = self.hash\_function(key)

bucket = self.buckets[index]

for i, (k, v) in enumerate(bucket):

if k == key:

del bucket[i] # Remove the key-value pair

return

def print\_map(self):

# Print all key-value pairs in the hash map

print("Hash Map Contents:")

for index, bucket in enumerate(self.buckets):

print(f"Bucket {index}: {bucket}")

# Creating the Hash Map from the simulation

hash\_map = SimpleHashMap(size=10)

# Adding some entries

hash\_map.put("123-4567", "Charlotte")

hash\_map.put("123-4568", "Thomas")

hash\_map.put("123-4569", "Jens")

hash\_map.put("123-4570", "Peter")

hash\_map.put("123-4571", "Lisa")

hash\_map.put("123-4672", "Adele")

hash\_map.put("123-4573", "Michaela")

hash\_map.put("123-6574", "Bob")

hash\_map.print\_map()

print("\nName associated with '123-4570':", hash\_map.get("123-4570"))

print("Updating the name for '123-4570' to 'James'")

hash\_map.put("123-4570","James")

print("Name associated with '123-4570':", hash\_map.get("123-4570"))

#Python

Output

A computer screen with white text

Description automatically generated

It is easier to understand how Hash Maps work if you first have a look at the two previous pages about Hash Tables and Hash Sets. It is also important to understand the meaning of the words below.

* Entry: Consists of a key and a value, forming a key-value pair.
* Key: Unique for each entry in the Hash Map. Used to generate a hash code determining the entry's bucket in the Hash Map. This ensures that every entry can be efficiently located.
* Hash Code: A number generated from an entry's key, to determine what bucket that Hash Map entry belongs to.
* Bucket: A Hash Map consists of many such buckets, or containers, to store entries.
* Value: Can be nearly any kind of information, like name, birth date, and address of a person. The value can be many different kinds of information combined.

1. **Finding the Hash Code**

A hash code is generated by a hash function.

The hash function in the simulation above takes the numbers in the social security number (not the dash), add them together, and does a modulo 10 operation (% 10) on the sum of characters to get the hash code as a number from 0 to 9.

This means that a person is stored in one of ten possible buckets in the Hash Map, according to the hash code of that person's social security number. The same hash code is generated and used when we want to search for or remove a person from the Hash Map.

The Hash Code gives us instant access as long as there is just one person in the corresponding bucket.

In the simulation above, Charlotte has social security number 123-4567. Adding the numbers together gives us a sum 28, and modulo 10 of that is 8. That is why she belongs to bucket 8.

**Direct Access in Hash Maps**

Searching for Charlotte in the Hash Map, we must use the social security number 123-4567 (the Hash Map key), which generates the hash code 8, as explained above.

This means we can go straight to bucket 8 to get her name (the Hash Map value), without searching through other entries in the Hash Map.

In cases like this we say that the Hash Map has constant time O(1) for searching, adding, and removing entries, which is really fast compared to using an array or a linked list.

But, in a worst case scenario, all the people are stored in the same bucket, and if the person we are trying to find is last person in this bucket, we need to compare with all the other social security numbers in that bucket before we find the person we are looking for.

In such a worst case scenario the Hash Map has time complexity O(n), which is the same time complexity as arrays and linked lists.

To keep Hash Maps fast, it is therefore important to have a hash function that will distribute the entries evenly between the buckets, and to have around as many buckets as Hash Map entries.

Having a lot more buckets than Hash Map entries is a waste of memory, and having a lot less buckets than Hash Map entries is a waste of time.

**Hash Map Implementation**

Hash Maps in Python are typically done by using Python's own [dictionary data type](https://www.w3schools.com/dsa/python/python_dictionaries.asp), but to get a better understanding of how Hash Maps work we will not use that here.

To implement a Hash Map in Python we create a class SimpleHashMap.

Inside the SimpleHashMap class we have a method \_\_init\_\_ to initialize the Hash Map, a method hash\_function for the hash function, and methods for the basic Hash Map operations: put, get, and remove.

We also create a method print\_map to better see how the Hash Map looks like.

class SimpleHashMap:

def \_\_init\_\_(self, size=100):

self.size = size

self.buckets = [[] for \_ in range(size)] # A list of buckets, each is a list (to handle collisions)

def hash\_function(self, key):

# Sum only the numerical values of the key, ignoring non-numeric characters

numeric\_sum = sum(int(char) for char in key if char.isdigit())

return numeric\_sum % 10 # Perform modulo 10 on the sum

def put(self, key, value):

# Add or update a key-value pair

index = self.hash\_function(key)

bucket = self.buckets[index]

for i, (k, v) in enumerate(bucket):

if k == key:

bucket[i] = (key, value) # Update existing key

return

bucket.append((key, value)) # Add new key-value pair if not found

def get(self, key):

# Retrieve a value by key

index = self.hash\_function(key)

bucket = self.buckets[index]

for k, v in bucket:

if k == key:

return v

return None # Key not found

def remove(self, key):

# Remove a key-value pair

index = self.hash\_function(key)

bucket = self.buckets[index]

for i, (k, v) in enumerate(bucket):

if k == key:

del bucket[i] # Remove the key-value pair

return

def print\_map(self):

# Print all key-value pairs in the hash map

print("Hash Map Contents:")

for index, bucket in enumerate(self.buckets):

print(f"Bucket {index}: {bucket}")

# Creating the Hash Map from the simulation

hash\_map = SimpleHashMap(size=10)

# Adding some entries

hash\_map.put("123-4567", "Charlotte")

hash\_map.put("123-4568", "Thomas")

hash\_map.put("123-4569", "Jens")

hash\_map.put("123-4570", "Peter")

hash\_map.put("123-4571", "Lisa")

hash\_map.put("123-4672", "Adele")

hash\_map.put("123-4573", "Michaela")

hash\_map.put("123-6574", "Bob")

hash\_map.print\_map()

# Demonstrating retrieval

print("\nName associated with '123-4570':", hash\_map.get("123-4570"))

print("Updating the name for '123-4570' to 'James'")

hash\_map.put("123-4570","James")

# Checking if Peter is still there

print("Name associated with '123-4570':", hash\_map.get("123-4570"))

Output

A screenshot of a computer

Description automatically generated