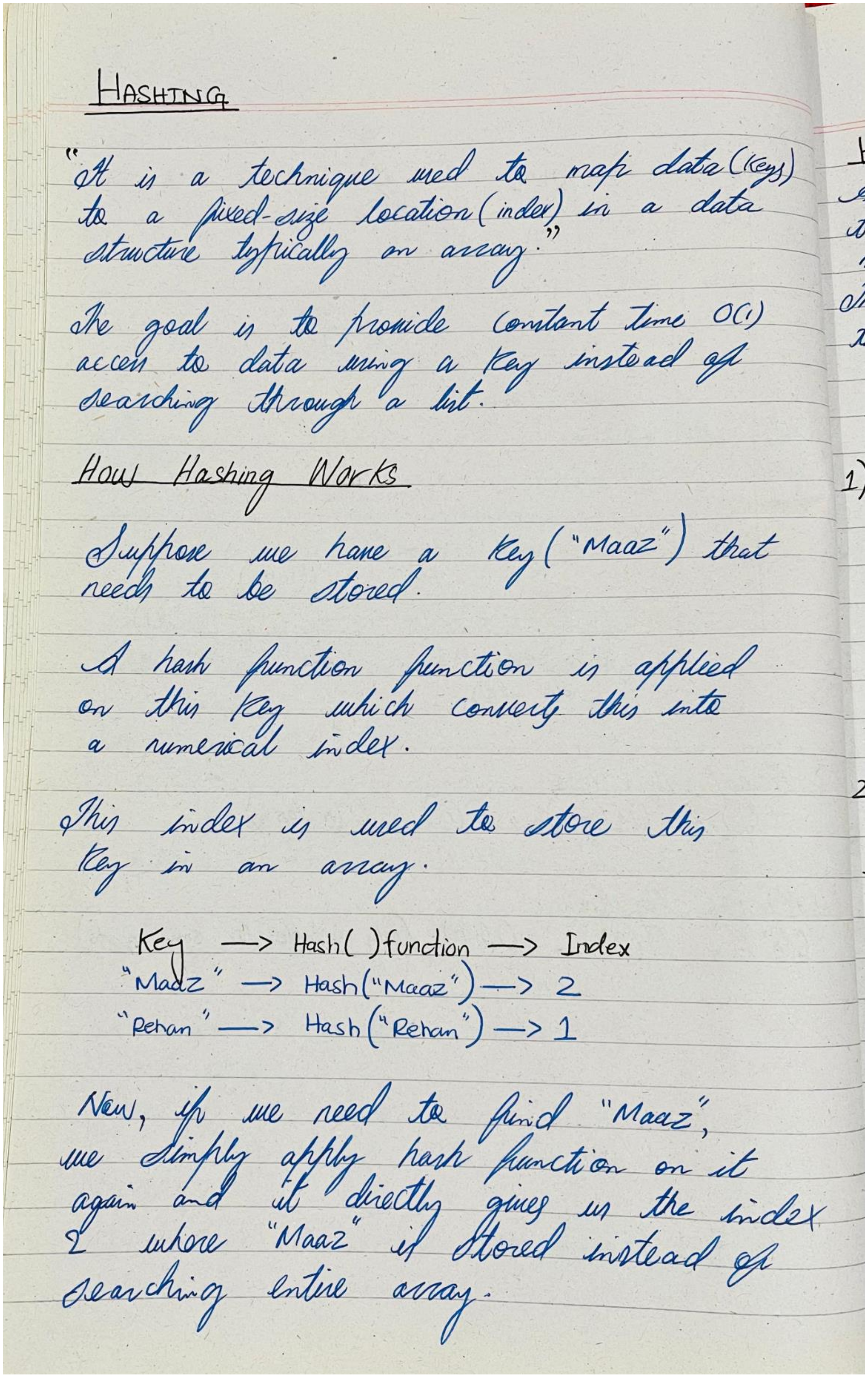
Hashing



HASHING

"It is a technique used to map data (keys) to a fixed-size location (index) in a data structure typically an array."

The goal is to provide constant time O(1) access to data using a key instead of searching through a list.

How Hashing Works

Suppose we have a key ("Maaz") that needs to be stored.

A hash function function is applied on this key which converts this into a numerical index.

This index is used to store this key in an array.

Key —> Hash( )function —> Index
"Maaz" —> Hash("Maaz") —> 2
"Rehan" —> Hash("Rehan") —> 1

Now, if we need to find "Maaz", we simply apply hash function on it again and it directly gives us the index 2 where "Maaz" is stored instead of searching entire array.

Hash Functions

A hash function is the core of hashing. It takes an input (Key) and converts it into a numerical value called a hash code (or hash value) This hash code determines the index where the value is stored in a hash table.

TYPES OF Hash Functions

1) Modulo Hashing (Division Method)

[Key % Table_size = index]

```
def  modulo_hash (Key, table_size):
     hash_code = Sum( ord(char) for char in Key) % table_siz
     return hash_code
```

2) Polynomial Rolling Hash

$$\text{hash}(S) = \left(S[0]\,p^{0} + S[1]\,p^{1} + S[2]\,p^{2} + \cdots + S[n-1]\cdot p^{n-1}\right) \% \, m$$

$$\sum_{i=0}^{n-1} S[i]\cdot p^{i} \ \% \ m$$

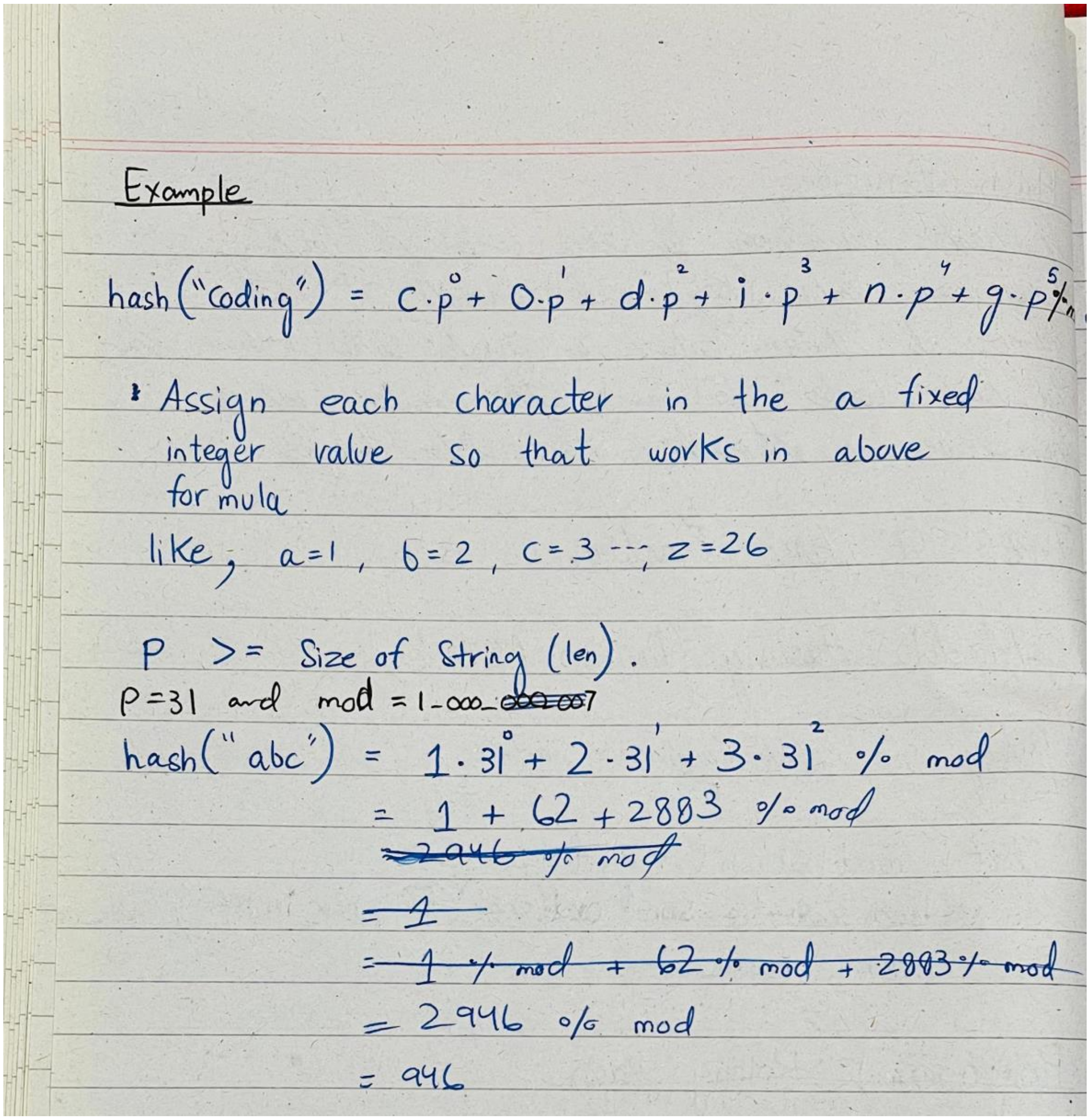
here

i = indices from 0 to $n-1$

p = is the prime

NOTE: choose a prime between 31 and 53

31, 37, 41, 43, 47, 53



Example

hash("coding") = $c \cdot p^0 + o \cdot p^1 + d \cdot p^2 + i \cdot p^3 + n \cdot p^4 + g \cdot p^5 \% m$

- Assign each character in the a fixed integer value so that works in above formula

like, a=1, b=2, c=3 ---, z=26

P >= Size of String (len).

P=31 and mod = 1-000-000007

hash("abc") = $1 \cdot 31^0 + 2 \cdot 31^1 + 3 \cdot 31^2 \% \text{ mod}$

= 1 + 62 + 2883 % mod

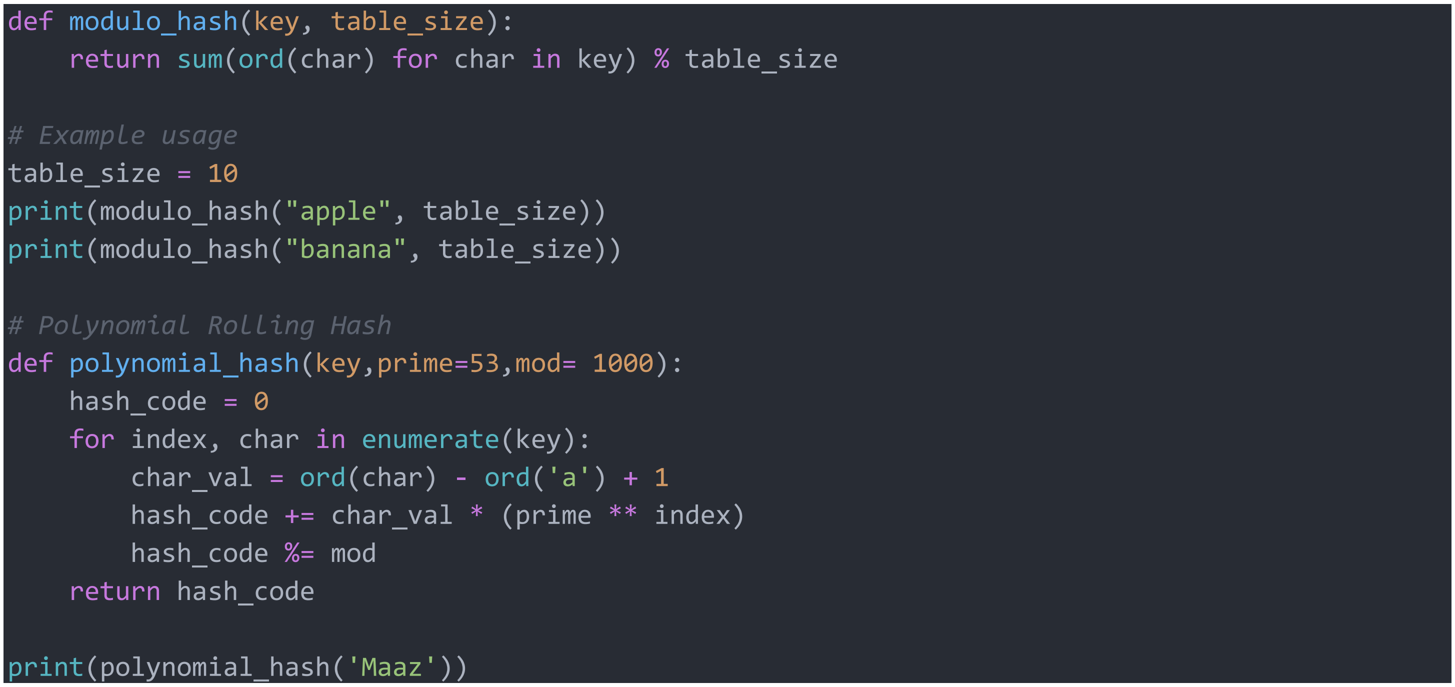
~~= 2946 % mod~~

~~= 1~~

~~= 1 % mod + 62 % mod + 2883 % mod~~

= 2946 % mod

= 946

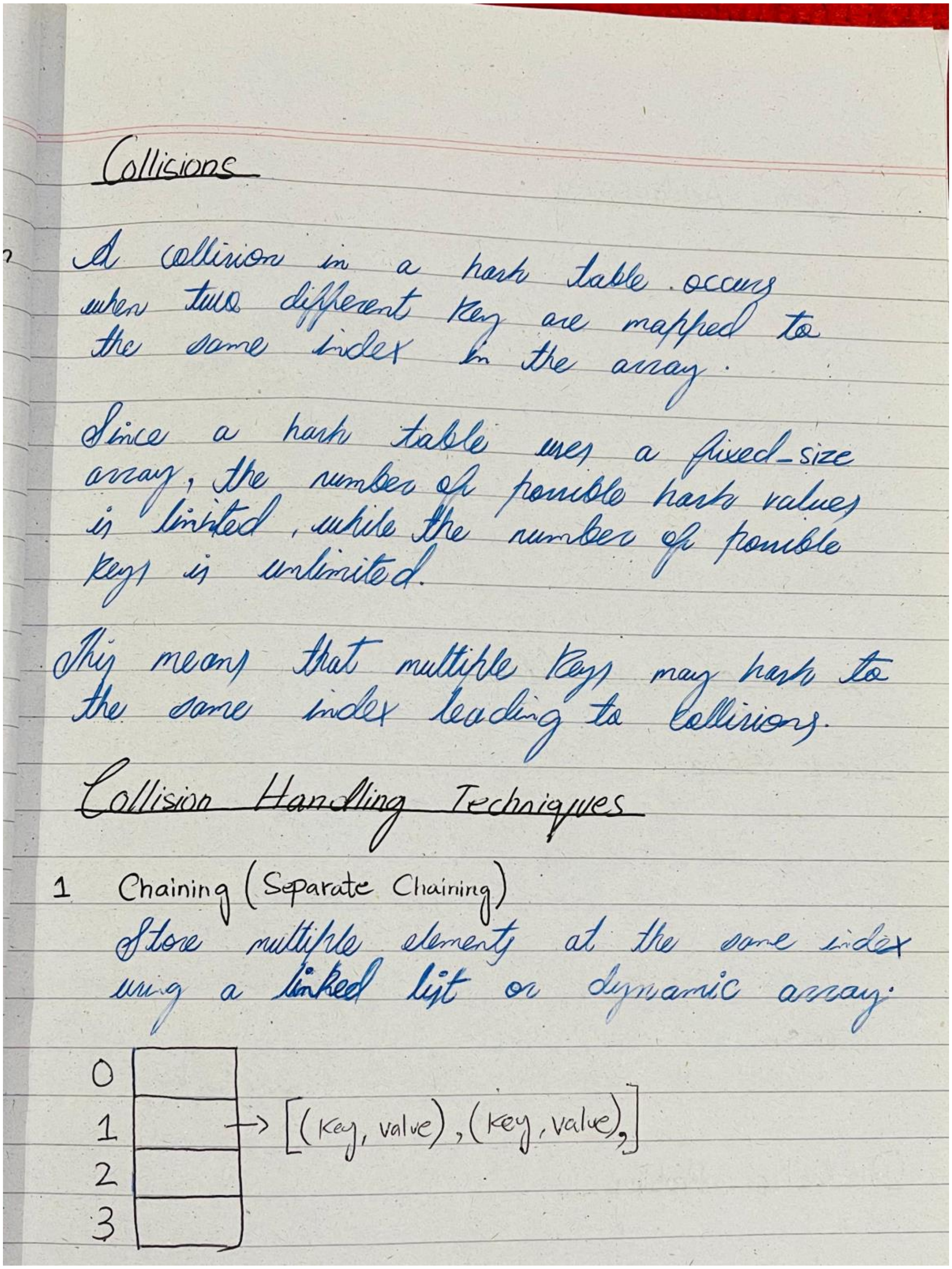


```python
def modulo_hash(key, table_size):
    return sum(ord(char) for char in key) % table_size

# Example usage
table_size = 10
print(modulo_hash("apple", table_size))
print(modulo_hash("banana", table_size))

# Polynomial Rolling Hash
def polynomial_hash(key,prime=53,mod= 1000):
    hash_code = 0
    for index, char in enumerate(key):
        char_val = ord(char) - ord('a') + 1
        hash_code += char_val * (prime ** index)
        hash_code %= mod
    return hash_code

print(polynomial_hash('Maaz'))
```

Simple Hash Table

```
# Simple Hash table without collision handling

class SimpleHashTable:
    def __init__(self,size=10):
        self.size = size
        self.table = [None] * self.size # fixed size array

    # It converts the key into hash code(index)
    def hash_function(self,key):
        return sum(ord(char) for char in key) %  self.size

    def polynomial_hash(self, key, prime=31, mod=10):
        hash_value = 0
        for i, char in enumerate(key):
            char_value = ord(char) - ord('a') + 1
            hash_value += char_value * (prime ** i)
            hash_value %= mod
            return hash_value

    # It inserts a tuple at index after fingding it through hash function

    def simple_insert(self,key,value):
        index = self.hash_function(key)
        self.table[index] = (key,value)

    def insert(self,key,value):
        index = self.polynomial_hash(key)
        self.table[index] = (key,value)


    def simple_search(self,key):
        index = self.hash_function(key)
        if self.table[index] and self.table[index][0] == key:
            return self.table[index][1]

    def search(self,key):
        index = self.polynomial_hash(key)
        if self.table[index] and self.table[index][0] == key:
            return self.table[index][1]


    def display(self):
        for i,item in enumerate(self.table):
            print(f'INDEX-{i} : {item}')
```

```python
hashTable = SimpleHashTable()
hashTable.insert('name','maaz')
hashTable.insert('age',20)

print(hashTable.search('age'))

hashTable.display()
```



Collisions

A collision in a hash table occurs when two different Key are mapped to the same index in the array.

Since a hash table uses a fixed-size array, the number of possible hash values is limited, while the number of possible keys is unlimited.

This means that multiple Keys may hash to the same index leading to collisions.

Collision Handling Techniques

1 Chaining (Separate Chaining)
Store multiple elements at the same index using a linked list or dynamic array.

Chaining (Separate Chaining)

```python
# Hash Table with collision handling ----- Chaining(Seperate Chaining)

class HashTable():
    def __init__(self,size=10):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def hash_function(self,key):
        return sum(ord(char) for char in key) %  self.size

    def insert(self,key,value):
        index = self.hash_function(key)
        self.table[index].append((key,value))

    def search(self,key):
        index = self.hash_function(key)
        for k,v in self.table[index]:
            if k == key:
                return v
        return None

    def display(self):
        for i,item in enumerate(self.table):
            print(f'INDEX-{i} : {item}')

hashTable = HashTable()

hashTable.insert('name','maaz')
hashTable.insert('age',20)

hashTable.insert('ega',100)


hashTable.display()
print(hashTable.search('name'))
```

2) Open Addressing

Unlike Chaining instead of storing multiple values at the same index, it search for the next available slot in the array.

When a collision happens, it find another open slot using a probing method.

Types of Open Addressing

Linear Probing

Search the next available index by moving sequentially.

index = hash_function (Key)

index = (index + 1) % size

issues Leads to clustering (many values grouped together)

Quadratic Probing

Instead of checking the next index use a quadratic sequence.

$$index = (index + i^2) \% size$$

where i starts at 1 and increases by 1 with each subsequent collision.

Double Hashing

Use a second hash function to find a step size. for probing when collision occurs

index (hash1(Key) + i * hash2(Key)) % size

Step size = hash2(Key)

where

```
hash2(Key):
    return 1 + (sum(ord(char) for char in Key) %
                                        (Size-1))
```

Now, next index

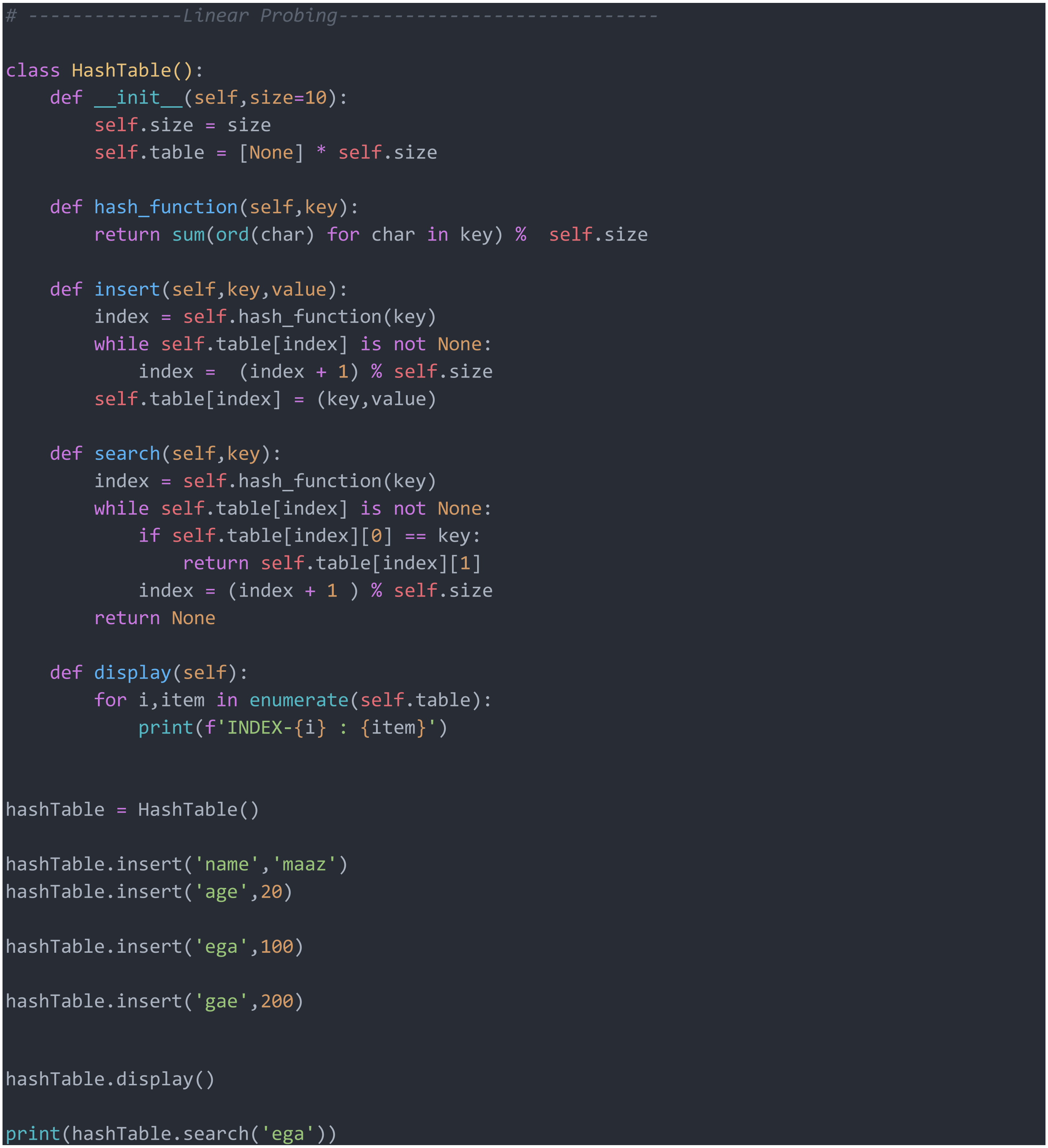
index = (inedex + i * ~~ha~~ Step Size) % Size

What is Hashable

Hashable	Not Hashable
Things that are immutable can be hashed	Things that are mutable can't be hashed
Strings Integers Tuples	Arrays Dictionaries

Open Addressing

1. Linear Probing



```python
# -------------Linear Probing-----------------------------

class HashTable():
    def __init__(self,size=10):
        self.size = size
        self.table = [None] * self.size

    def hash_function(self,key):
        return sum(ord(char) for char in key) %  self.size

    def insert(self,key,value):
        index = self.hash_function(key)
        while self.table[index] is not None:
            index =  (index + 1) % self.size
        self.table[index] = (key,value)

    def search(self,key):
        index = self.hash_function(key)
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + 1 ) % self.size
        return None

    def display(self):
        for i,item in enumerate(self.table):
            print(f'INDEX-{i} : {item}')


hashTable = HashTable()

hashTable.insert('name','maaz')
hashTable.insert('age',20)

hashTable.insert('ega',100)

hashTable.insert('gae',200)


hashTable.display()

print(hashTable.search('ega'))
```

2.Quadratic Probing

```python
# ---------- Quadratic Probing-------------

class HashTable():
    def __init__(self,size=10):
        self.size = size
        self.table = [None] * self.size

    def hash_function(self,key):
        return sum(ord(char) for char in key) % self.size

    def insert(self,key,value):
        index =  self.hash_function(key)
        i=1
        while self.table[index] is not None:
            index = (index + i **2) % self.size # Formula
            i += 1

            if i > self.size:
                print('Hash table limit Exceeds Please increase the size')
                return None

        self.table[index] = (key,value)

    def search(self,key):
        index = self.hash_function(key)
        i = 1
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]

            index =  (index + i **2) % self.size
            i +=1
        return None

    def display(self):
        for i,item in enumerate(self.table):
            print(f'INDEX-{i} : {item}')


hashTable = HashTable(size=20)

hashTable.insert('name','maaz')
hashTable.insert('age',20)
hashTable.insert('ega',100)
hashTable.insert('gae',200)
hashTable.insert('eman','val')
hashTable.insert('mane','val')
hashTable.insert('amne','val')
hashTable.insert('gender','val')
hashTable.insert('dergen','val')
hashTable.insert('ergend','val')



hashTable.display()
```

3.Double Hashing

```
# ---------- Double Hashing-------------

class DoubleHashingHashTable():
    def __init__(self,size=10):
        self.size = size
        self.table = [None] * self.size

    def hash1_function(self,key):
        return sum(ord(char) for char in key) % self.size

    def hash2_function(self,key):
        return 1 + (sum(ord(char) for char in key) % (self.size -1))

    def insert(self,key,value):
        index = self.hash1_function(key)
        step_size = self.hash2_function(key)
        i = 1
        while self.table[index] is not None:
            index = (index + i * step_size) % self.size # FORMULA
            i += 1

            if i > self.size:
                print('Hash table limit Exceeds Please increase the size')
                return None
        self.table[index] = (key,value)

    def search(self,key):
        index = self.hash1_function(key)
        step_size = self.hash2_function(key)
        i = 1
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + i * step_size) % self.size
            i +=1
        return None

    def display(self):
        for i,item in enumerate(self.table):
            print(f'INDEX-{i} : {item}')
```

```
hashTable = DoubleHashingHashTable(size=20)

hashTable.insert('name','maaz')
hashTable.insert('age',20)
hashTable.insert('ega',100)
hashTable.insert('gae',200)
hashTable.insert('eman','val')
hashTable.insert('mane','val')
hashTable.insert('amne','val')
hashTable.insert('gender','val')
hashTable.insert('dergen','val')
hashTable.insert('ergend','val')



hashTable.display()

print(hashTable.search('ggg'))
```