



## MODUL PERKULIAHAN

# TFC251 – Praktikum Struktur Data

## Hashing Dalam Python

<b>Penyusun Modul</b>	: Suamanda Ika Novichasari, M.Kom
<b>Minggu/Pertemuan</b>	: 11-12
<b>Sub-CPMK/Tujuan Pembelajaran</b>	: <ol style="list-style-type: none"><li>1. Mahasiswa mampu menerapkan konsep Hashing pada bahasa pemrograman python</li></ol>
<b>Pokok Bahasan</b>	: <ol style="list-style-type: none"><li>1. Struktur Data Hash</li><li>2. Tabel Hash</li><li>3. Fungsi Hash</li><li>4. <i>Collision</i></li></ol>

**Program Studi Teknologi Informasi (S1)**  
**Fakultas Teknik**  
**Universitas Tidar**  
**Tahun 2024**



# Materi 9

## HASHING DALAM PYTHON

---

### Petunjuk Praktikum :

- Cobalah semua contoh penyelesaian kasus dengan kode program yang terdapat pada semua sub bab dalam modul ini menggunakan google colab.
- Dokumentasikan kegiatan dalam bentuk laporan studi kasus sesuai template laporan praktikum yang telah ditentukan.

Peningkatan kebutuhan akan struktur data Hash sejalan dengan pertumbuhan yang cepat dari jumlah data di internet. Meskipun ukuran data dalam kegiatan pemrograman sehari-hari mungkin tidak sebesar itu, namun tetap memerlukan penyimpanan, akses, dan pemrosesan yang efisien. Meskipun Struktur data Array umum digunakan, namun memiliki batasan efisiensi karena walaupun penyimpanannya cepat ( $O(1)$ ), pencariannya memerlukan waktu logaritmik ( $O(\log n)$ ), yang mungkin tidak efisien terutama untuk data yang besar.

Dengan demikian, diperlukan sebuah struktur data baru yang mampu menyimpan dan mencari data dengan cepat dalam waktu konstan ( $O(1)$ ). Inilah alasan mengapa struktur data Hashing sangat penting. Dengan menggunakan Hashing, data dapat disimpan dan diambil dengan mudah serta efisien dalam waktu yang konstan.

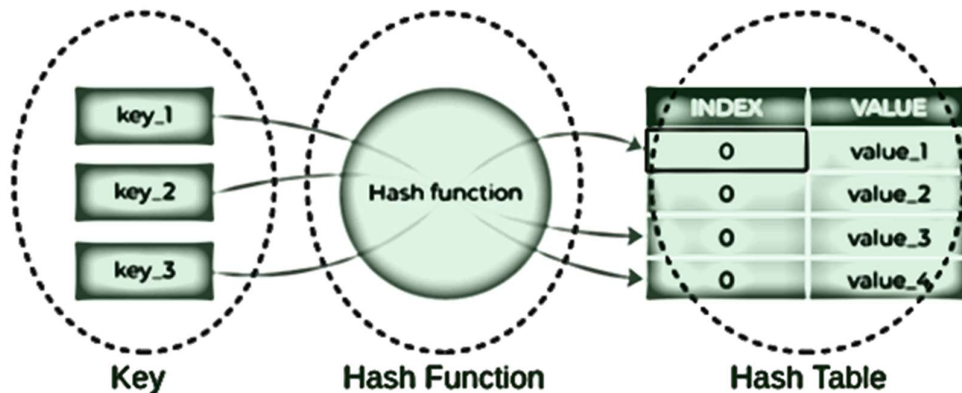
### 9.1 Struktur Data Hash

Hashing adalah prinsip dasar dalam struktur data yang dapat menyimpan dan mengambil data dengan efisien, memungkinkan akses yang cepat. Hashing dilakukan dengan cara memetakan data ke indeks tertentu dalam tabel hash menggunakan fungsi hash, yang memungkinkan pengambilan informasi secara cepat berdasarkan kunci yang diberikan. Teknik ini sering digunakan dalam berbagai aplikasi pemrograman,

termasuk basis data dan sistem caching, untuk meningkatkan efisiensi dalam pencarian dan pengambilan data.

Hashing memiliki tiga komponen utama:

- **Kunci:** Sebuah nilai yang bisa berupa string atau bilangan bulat yang digunakan sebagai input dalam fungsi hash untuk menentukan indeks atau lokasi penyimpanan dalam struktur data.
- **Fungsi Hash:** Fungsi yang menerima kunci input dan mengembalikan indeks elemen dalam tabel hash, yang dikenal sebagai indeks hash.
- **Tabel Hash:** Struktur data yang memetakan kunci ke nilai menggunakan fungsi hash, dengan menyimpan data secara asosiatif dalam array di mana setiap nilai memiliki indeks uniknya sendiri.



Kelebihan Hashing	Kekurangan Hashing
<ul style="list-style-type: none"> <li>• Memberikan sinkronisasi yang lebih baik dibandingkan dengan struktur data lainnya.</li> <li>• Tabel hash lebih efisien daripada pohon pencarian atau struktur data lainnya.</li> <li>• Memberikan waktu yang konstan untuk operasi pencarian, penyisipan, dan penghapusan secara rata-rata.</li> </ul>	<ul style="list-style-type: none"> <li>• Hash menjadi tidak efisien ketika terjadi banyak tabrakan.</li> <li>• Tabrakan hash tidak dapat dihindari untuk kumpulan kunci yang besar.</li> <li>• Hash tidak memperbolehkan nilai null.</li> </ul>

## 9.2 Tabel Hash

Transpose dari sebuah graf berarah  $G$  adalah graf berarah lain pada himpunan titik yang sama dengan semua tepi dibalik dibandingkan dengan orientasi tepi yang sesuai di  $G$ . Artinya, jika  $G$  mengandung tepi  $(u, v)$  maka konvers/transposisi/balik dari  $G$  mengandung tepi  $(v, u)$  dan sebaliknya. Diberikan sebuah graf (diwakili sebagai daftar ketetanggaan), kita perlu menemukan graf lain yang merupakan transpose dari graf yang diberikan.

```
ukuran = 20
class DataItem:
    def __init__(self, data, key):
        self.data = data
        self.key = key
    # Initialize the hash array with None values
    hashArray = [None] * ukuran
    # Create a dummy item to mark deleted cells in the hash table
    dummyItem = DataItem(-1, -1)
    # Variable to hold the item found in the search operation
    item = None

    # Hash function to calculate the hash index for the given key
    def hashCode(key):
        return key % ukuran

    # Function to search for an item in the hash table by its key
    def search(key):
        # Calculate the hash index using the hash function
        hashIndex = hashCode(key)
        # Traverse the array until an empty cell is encountered
        while hashArray[hashIndex] is not None:
            if hashArray[hashIndex].key == key:
                # Item found, return the item
                return hashArray[hashIndex]
            # Move to the next cell (linear probing)
            hashIndex = (hashIndex + 1) % ukuran
        # If the loop terminates without finding the item, it means the item is not present
        return None

    # Function to insert an item into the hash table
    def insert(key, data):
        # Create a new DataItem object
        item = DataItem(data, key)
        # Calculate the hash index using the hash function
        hashIndex = hashCode(key)
        # Handle collisions using linear probing (move to the next cell until an empty cell is found)
        while hashArray[hashIndex] is not None and hashArray[hashIndex].key != -1:
            hashIndex = (hashIndex + 1) % ukuran
        # Insert the item into the hash table at the calculated index
        hashArray[hashIndex] = item

    # Function to delete an item from the hash table
    def deleteItem(item):
        key = item.key
        # Calculate the hash index using the hash function
        hashIndex = hashCode(key)
        # Traverse the array until an empty or deleted cell is encountered
        while hashArray[hashIndex] is not None:
            if hashArray[hashIndex].key == key:
                # Item found, mark the cell as deleted by replacing it with the dummyItem
                temp = hashArray[hashIndex]
                hashArray[hashIndex] = dummyItem
                return temp
            # Move to the next cell (linear probing)
            hashIndex = (hashIndex + 1) % ukuran
        # If the loop terminates without finding the item, it means the item is not present
        return None
```

- Variabel ukuran ditetapkan sebagai 20. Ini adalah ukuran hash table yang akan digunakan.
- Kelas `DataItem` didefinisikan dengan atribut data dan key untuk menyimpan data dan kunci dari item.
- Array `hashArray` diinisialisasi dengan nilai `None` sebanyak ukuran kali. Ini adalah hash table yang akan digunakan untuk menyimpan item.
- Sebuah objek dummy (`dummyItem`) dibuat dengan menggunakan kelas `DataItem`. Objek ini digunakan untuk menandai sel-sel yang telah dihapus dalam hash table.
- Variabel `item` diinisialisasi sebagai `None`. Ini akan digunakan untuk menyimpan item yang ditemukan dalam operasi pencarian.
- Fungsi `hashCode(key)` didefinisikan untuk menghitung indeks hash untuk kunci yang diberikan. Ini menggunakan operasi modulus untuk mendapatkan indeks berdasarkan ukuran hash table.
- Fungsi `search(key)` didefinisikan untuk mencari item dalam hash table berdasarkan kunci. Ini menghitung indeks hash menggunakan fungsi `hashCode(key)` dan melakukan probing linear sampai menemukan item atau menemui sel kosong.
- Fungsi `insert(key, data)` didefinisikan untuk menyisipkan item baru ke dalam hash table. Ini membuat objek `DataItem` baru, menghitung indeks hash, dan menangani tabrakan menggunakan linear probing.
- Fungsi `deleteItem(item)` didefinisikan untuk menghapus item dari hash table. Ini menghapus item dengan kunci yang sesuai dan menandai sel sebagai `dummyItem` jika item ditemukan.
- Fungsi `display()` didefinisikan untuk menampilkan isi dari hash table. Ini mencetak kunci dan data dari setiap item dalam hash table.
- Pada blok `if __name__ == "__main__":`, beberapa item disisipkan ke dalam hash table menggunakan fungsi `insert()`. Setelah itu, pesan "Insertion done" dicetak.
- Setelah itu, isi dari hash table ditampilkan dengan memanggil fungsi `display()`.

```

# Function to display the hash table
def display():
    for i in range(ukuran):
        if hashArray[i] is not None:
            # Print the key and data of the item at the current index
            print("{} {}".format(hashArray[i].key, hashArray[i].data), end=" ")
        else:
            # Print -- for an empty cell
            print("--", end=" ")
        print()

if __name__ == "__main__":
    # Test the hash table implementation Insert some items into the hash table
    insert(1, 20)
    insert(2, 70)
    insert(42, 80)
    insert(4, 25)
    insert(12, 44)
    insert(14, 32)
    insert(17, 11)
    insert(13, 78)
    insert(37, 97)
    print("Insertion done")
    print("Hash Table contents: ")
    # Display the hash table
    display()
    display()
    # Search for an item with a specific key (37)
    item = search(37)

    # Check if the item was found or not and print the result
    if item is not None:
        print("Element found:", item.data)
    else:
        print("Element not found")

    # Delete the item with key 37 from the hash table
    deleteItem(item)

    # Search again for the item with key 37 after deletion
    item = search(37)

    # Check if the item was found or not and print the result
    if item is not None:
        print("Element found:", item.data)
    else:
        print("Element not found")

```

```

Insertion done
Hash Table contents:
-- (1, 20) (2, 70) (42, 80) (4, 25) -- -- -- -- -- (12, 44) (13, 78) (14, 32) -- -- (17, 11) (37, 97) --
-- (1, 20) (2, 70) (42, 80) (4, 25) -- -- -- -- -- (12, 44) (13, 78) (14, 32) -- -- (17, 11) (37, 97) --
Element found: 97
Element not found

```

- Fungsi display():
  - Fungsi ini digunakan untuk menampilkan isi dari tabel hash.
  - Iterasi dilakukan melalui seluruh indeks hashArray, yang ditentukan oleh ukuran.
  - Jika sel pada indeks tertentu tidak kosong (not None), maka kunci (key) dan data dari item di indeks tersebut dicetak.
  - Jika sel pada indeks tertentu kosong, maka cetak -- untuk menandakan sel kosong.
  - Setelah iterasi selesai, baris baru (print()) ditambahkan untuk memisahkan tampilan hash table.

- Blok if `__name__ == "__main__":`:
  - Ini adalah blok kode yang akan dijalankan jika program ini dijalankan sebagai program utama (bukan modul yang diimpor oleh program lain).
  - Beberapa item disisipkan ke dalam tabel hash menggunakan fungsi `insert()`.
  - Setelah semua item disisipkan, pesan "Insertion done" dicetak.
  - Pesan "Hash Table contents: " juga dicetak untuk menandakan bahwa tabel hash akan ditampilkan.
- Panggilan fungsi `display()`:
  - Fungsi `display()` dipanggil dua kali untuk menampilkan isi dari tabel hash setelah penyisipan item.
- Pencarian item dengan kunci tertentu:
  - Fungsi `search()` dipanggil untuk mencari item dengan kunci tertentu (37).
  - Jika item ditemukan, data dari item tersebut dicetak.
  - Jika item tidak ditemukan, pesan "Element not found" dicetak.
- Penghapusan item dengan kunci tertentu:
  - Fungsi `deleteItem()` dipanggil untuk menghapus item dengan kunci tertentu (37).
  - Setelah item dihapus, pencarian kembali dilakukan menggunakan fungsi `search()` untuk memastikan item sudah tidak ada lagi dalam tabel hash.
  - Hasil pencarian setelah penghapusan dicetak, baik jika item ditemukan atau tidak.

### 9.3 Fungsi Hash

Sebuah fungsi hashing harus memenuhi kriteria-kriteria berikut agar efektif dan aman:

- Deterministik — Diberikan input yang sama, fungsi harus selalu mengembalikan output yang sama.
- Efisien — Fungsi harus efisien secara komputasi saat menghitung nilai hash dari input apa pun.
- Tahan tabrakan — Fungsi harus meminimalkan kemungkinan dua input menghasilkan nilai hash yang sama.
- Seragam — Output fungsi harus didistribusikan secara seragam di seluruh rentang nilai hash yang mungkin.
- Tidak dapat dibalik — Seharusnya sulit bagi komputer untuk menghitung nilai input fungsi berdasarkan nilai hash.

- f. Tidak dapat diprediksi — Memprediksi output fungsi harus sulit, dengan diberikan seperangkat input.
- g. Sensitif terhadap perubahan input — Fungsi harus sensitif terhadap perbedaan kecil dalam input. Perubahan kecil harus menyebabkan perbedaan besar dalam nilai hash yang dihasilkan.

Setelah memiliki fungsi hashing yang memadai dengan semua karakteristik tersebut, kemudian dapat diaplikasikan ke berbagai kasus penggunaan. Fungsi hashing bekerja dengan baik untuk:

- a. Penyimpanan kata sandi — Hashing adalah salah satu cara terbaik untuk menyimpan kata sandi pengguna dalam sistem modern. Python menggabungkan berbagai modul untuk meng-hash dan mengamankan kata sandi sebelum menyimpannya dalam database.
- b. Penyimpanan sementara (caching) — Hashing menyimpan output fungsi untuk menghemat waktu saat memanggilnya nanti.
- c. Pengambilan data — Python menggunakan tabel hash dengan struktur data kamus (dictionary) bawaan untuk dengan cepat mengambil nilai berdasarkan kunci.
- d. Tanda tangan digital — Hashing dapat memverifikasi keaslian pesan yang memiliki tanda tangan digital.
- e. Pemeriksaan integritas file — Hashing dapat memeriksa integritas file selama transfer dan pengunduhan.

### 7.3.1 Fungsi Hashing Bawaan Python

Fungsi hashing bawaan Python, `hash()`, mengembalikan nilai integer yang mewakili objek input. Kode kemudian menggunakan nilai hash yang dihasilkan untuk menentukan lokasi objek dalam tabel hash. Tabel hash ini adalah struktur data yang mengimplementasikan kamus (dictionary) dan himpunan (sets). Berikut contoh program menghitung nilai hash dari string "hello world" dan mencetak baik stringnya maupun nilai hash-nya.



```
Python's Built-In Hashing Function

my_string = "hello world"

# Calculate the hash value of the string
hash_value = hash(my_string)

# Print the string and its hash value
print("String: ", my_string)
print("Hash value: ", hash_value)

String: hello world
Hash value: 3519493165036936918
```

### 7.3.2 Keterbatasan Hashing

Meskipun fungsi hash Python menjanjikan untuk berbagai kasus penggunaan, keterbatasannya membuatnya tidak cocok untuk tujuan keamanan. Berikut adalah alasannya:

- Serangan tabrakan — Sebuah tabrakan terjadi ketika dua input yang berbeda menghasilkan nilai hash yang sama. Seorang penyerang bisa menggunakan metode pembuatan input yang sama untuk menghindari langkah-langkah keamanan yang bergantung pada nilai hash untuk otentikasi atau pemeriksaan integritas data.
- Ukuran input terbatas — Karena fungsi hash menghasilkan output berukuran tetap tanpa memperhatikan ukuran input, sebuah input yang lebih besar dari ukuran output fungsi hash dapat menyebabkan tabrakan.
- Prediktabilitas — Sebuah fungsi hash seharusnya bersifat deterministik, memberikan output yang sama setiap kali Anda memberikan input yang sama. Penyerang mungkin memanfaatkan kelemahan ini dengan memprediksi nilai hash untuk banyak input sebelumnya, lalu membandingkannya dengan nilai hash target untuk menemukan kecocokan. Proses ini disebut serangan tabel pelangi (rainbow table attack).

Untuk mencegah serangan dan menjaga data Anda aman, gunakan algoritma hashing yang aman yang dirancang untuk menahan kerentanan seperti ini.

### 7.3.3 Hashlib

Daripada menggunakan `hash()` bawaan Python, gunakan `hashlib` untuk hashing yang lebih aman. Modul Python ini menawarkan berbagai algoritma hash untuk meng-hash data secara aman. Algoritma-algoritma ini termasuk MD5, SHA-1, dan keluarga SHA-2 yang lebih aman, termasuk SHA-256, SHA-384, SHA-512, dan lain-lain.

#### a. MD5

Algoritma kriptografi yang banyak digunakan, MD5, menghasilkan nilai hash 128-bit. Gunakan kode seperti di bawah ini untuk menghasilkan hash MD5 menggunakan konstruktor `md5` dari `hashlib`:

A screenshot of a code editor window titled "MD5". It shows a Python script that imports the hashlib module, defines a text string "Hello World", creates a hash object using hashlib.md5(), and prints the hexadecimal digest. The output of the script is displayed at the bottom of the editor.

```
import hashlib

text = "Hello World"
hash_object = hashlib.md5(text.encode())
print(hash_object.hexdigest())

b10a8db164e0754105b7a99be72e3fe5
```

**Catatan:** Metode `hexdigest()` dalam kode di atas mengembalikan hash dalam format heksadesimal yang aman untuk presentasi non-biner (seperti email).

#### b. SHA-1

Fungsi hash SHA-1 mengamankan data dengan membuat nilai hash 160-bit. Gunakan kode di bawah dengan konstruktor `sha1` untuk hash SHA-1 modul `hashlib`:

A screenshot of a code editor window titled "SHA-1". It shows a Python script that imports the hashlib module, defines a text string "Hello World", creates a hash object using hashlib.sha1(), and prints the hexadecimal digest. The output of the script is displayed at the bottom of the editor.

```
[5] import hashlib

text = "Hello World"
hash_object = hashlib.sha1(text.encode())
print(hash_object.hexdigest())

0a4d55a8d778e5022fab701977c5d840bbc486d0
```

### c. SHA-256

Terdapat berbagai pilihan hash dalam keluarga SHA-2. Konstruktor hashlib SHA-256 menghasilkan versi yang lebih aman dalam keluarga tersebut dengan nilai hash 256-bit.

Para pengembang sering menggunakan SHA-256 untuk kriptografi, seperti tanda tangan digital atau kode otentikasi pesan. Kode di bawah ini menunjukkan bagaimana cara menghasilkan hash SHA-256:

```
✓ SHA-256

04 [6] import hashlib

    text = "Hello World"
    hash_object = hashlib.sha256(text.encode())
    print(hash_object.hexdigest())

a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
```

### d. SHA-384

SHA-384 adalah nilai hash 384-bit. Para pengembang sering menggunakan fungsi SHA-384 dalam aplikasi yang membutuhkan keamanan data lebih lanjut.

Berdasarkan contoh-contoh sebelumnya, Anda mungkin bisa menebak bahwa ini adalah pernyataan yang akan menghasilkan hash SHA-384:

```
✓ SHA-384

04 [8] import hashlib

    text = "Hello World"
    hash_object = hashlib.sha384(text.encode())
    print(hash_object.hexdigest())

99514329186b2f6ae4a1329e7ee6c610a729636335174ac6b740f9028396fcc803d0e93863a7c3d90f86beee782f4f3f
```

#### e. SHA-512

SHA-512 adalah anggota yang paling aman dari keluarga SHA-2. Ini membuat nilai hash 512-bit. Para pengembang menggunakannya untuk aplikasi berkapasitas tinggi, seperti pemeriksaan integritas data. Kode di bawah ini menunjukkan bagaimana cara menghasilkan hash SHA-512 dengan modul hashlib dalam Python:

```
SHA-512
[9] import hashlib

text = "Hello World"
hash_object = hashlib.sha512(text.encode())
print(hash_object.hexdigest())

2c74fd17edaf80e8447b0d46741ee243b7eb74dd2149a0ab1b9246fb30382f27e853d8585719e0e67cbda0daa8f51671064615d645ae27acb15bfb1447f459b
```

### 7.3.4 Penggunaan Hashing untuk kata sandi penyimpanan

Hashing dapat digunakan untuk menyimpan kata sandi dengan aman dalam aplikasi. Idealnya, kata sandi di-hash sebelum disimpan dalam database untuk mencegah akses yang tidak sah. Namun, hanya melakukan hashing saja mungkin tidak cukup melindungi kata sandi dari serangan brute force dan kamus. Untuk meningkatkan keamanan, teknik *salt*ing dapat digunakan. Setiap kali pengguna masuk, aplikasi akan menambahkan *salt* unik ke kata sandi sebelum melakukan hashing. Hal ini membuat serangan menjadi lebih kompleks. Modul rahasia Python dapat digunakan untuk menghasilkan *salt* secara acak dan menyimpan kata sandi dengan aman. Berikut adalah contoh kode yang menggunakan perpustakaan hashlib dan modul rahasia untuk meningkatkan keamanan kata sandi pengguna.

## 9.4 Collisions

*Collision* terjadi saat dua atau lebih kunci hashing menghasilkan lokasi yang sama dalam tabel hash. Dua metode umum untuk menangani *collision* adalah "*Separate Chaining*" dan "*Open Addressing*".

### 9.4.1 Separate Chaining

Dalam metode *Separate Chaining*, setiap sel tabel hash menunjuk ke struktur data terpisah seperti daftar berantai. Saat terjadi *collision*, entri baru disisipkan ke dalam struktur data terpisah tersebut di bawah kunci yang sama. Ini

memungkinkan beberapa kunci hashing untuk menunjuk ke sel yang sama dalam tabel hash.

```

▼ Separate Chaining

class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None

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

    def hash_function(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        if self.table[index] is None:
            self.table[index] = Node(key, value)
        else:
            current = self.table[index]
            while current.next:
                current = current.next
            current.next = Node(key, value)

    def search(self, key):
        index = self.hash_function(key)
        current = self.table[index]
        while current:
            if current.key == key:
                return current.value
            current = current.next
        return None

# Contoh penggunaan
hash_table = SeparateChainingHashTable(10)
hash_table.insert("apple", 10)
hash_table.insert("banana", 20)
hash_table.insert("orange", 30)

print(hash_table.search("apple")) # Output: 10
print(hash_table.search("banana")) # Output: 20
print(hash_table.search("orange")) # Output: 30

10
20
30

```

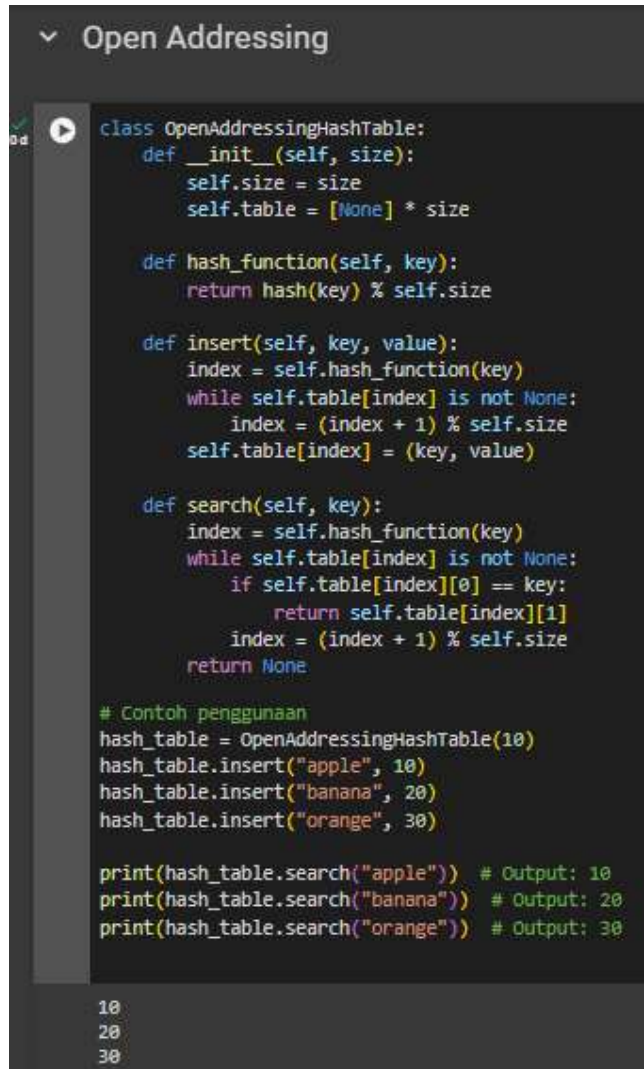
Program di atas merupakan implementasi dari *Separate Chaining* Hash Table dalam Python. Alur program ini dapat dijelaskan sebagai berikut:

- Mendefinisikan kelas Node:  
Kelas Node digunakan untuk merepresentasikan simpul dalam struktur data berantai. Setiap simpul memiliki atribut kunci (key), nilai (value), dan pointer ke simpul berikutnya (next).
- Mendefinisikan kelas SeparateChainingHashTable:
- Metode `__init__()` digunakan untuk menginisialisasi tabel hash dengan ukuran tertentu. Tabel hash diinisialisasi dengan daftar kosong dengan panjang yang sama dengan ukuran yang ditentukan.
- Metode `hash_function()` digunakan untuk menghasilkan indeks dalam tabel hash berdasarkan kunci yang diberikan. Ini dilakukan dengan menggunakan fungsi hash bawaan Python (`hash()`) dan mengambil sisa hasil bagi dengan ukuran tabel hash.
- Metode `insert()` digunakan untuk memasukkan pasangan kunci-nilai ke dalam tabel hash. Jika sel pada indeks hasil hashing masih kosong (`None`), sebuah simpul baru dibuat dan ditempatkan di sel tersebut. Jika sel sudah terisi, simpul baru akan ditambahkan ke ujung rantai yang terhubung dari simpul pertama.
- Metode `search()` digunakan untuk mencari nilai yang terkait dengan kunci yang diberikan. Algoritma mencari dilakukan dengan mengakses sel tabel hash yang sesuai dengan hasil hashing kunci, kemudian menelusuri rantai simpul yang terkait untuk mencari kunci yang cocok. Jika kunci ditemukan, nilai yang terkait dengan kunci tersebut dikembalikan; jika tidak ditemukan, `None` dikembalikan.
- Contoh penggunaan:
  - Membuat objek `hash_table` dari kelas `SeparateChainingHashTable` dengan ukuran tabel hash 10.
  - Memasukkan beberapa pasangan kunci-nilai ke dalam tabel hash menggunakan metode `insert()`.
  - Melakukan pencarian nilai terkait dengan kunci tertentu menggunakan metode `search()` dan mencetak hasilnya.

Program ini menggunakan *Separate Chaining* untuk menangani *collision* dalam tabel hash, di mana setiap sel tabel hash berisi rantai simpul yang terkait.

### 9.4.2 Open Addressing

Dalam metode *Open Addressing*, setiap sel dalam tabel hash dapat menampung maksimal satu entri. Ketika *collision* terjadi, algoritma pencarian akan mencari lokasi selanjutnya yang tersedia dalam tabel hash.



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

    def hash_function(self, key):
        return hash(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

# Contoh penggunaan
hash_table = OpenAddressingHashTable(10)
hash_table.insert("apple", 10)
hash_table.insert("banana", 20)
hash_table.insert("orange", 30)

print(hash_table.search("apple")) # Output: 10
print(hash_table.search("banana")) # Output: 20
print(hash_table.search("orange")) # Output: 30
```

10  
20  
30

Program di atas adalah implementasi dari *Open Addressing* tabel hash dalam Python. Berikut adalah penjelasan alur programnya:

- Mendefinisikan kelas `OpenAddressingHashTable`:
  - Metode `__init__()` digunakan untuk menginisialisasi tabel hash dengan ukuran tertentu. Tabel hash diinisialisasi dengan daftar kosong dengan panjang yang sama dengan ukuran yang ditentukan.

- Metode `hash_function()` digunakan untuk menghasilkan indeks dalam tabel hash berdasarkan kunci yang diberikan. Ini dilakukan dengan menggunakan fungsi hash bawaan Python (`hash()`) dan mengambil sisa hasil bagi dengan ukuran tabel hash.
- Metode `insert()` digunakan untuk memasukkan pasangan kunci-nilai ke dalam tabel hash menggunakan teknik *open addressing*. Jika sel pada indeks hasil hashing sudah terisi, algoritma akan mencari lokasi selanjutnya yang kosong dalam tabel hash dengan menggunakan teknik linear probing (menelusuri tabel dengan penambahan satu indeks). Setelah menemukan lokasi yang kosong, pasangan kunci-nilai baru dimasukkan ke dalam sel tersebut.
- Metode `search()` digunakan untuk mencari nilai yang terkait dengan kunci yang diberikan. Algoritma pencarian dilakukan dengan mengakses sel tabel hash yang sesuai dengan hasil hashing kunci. Jika kunci pada sel tersebut cocok dengan kunci yang dicari, nilai yang terkait dengan kunci tersebut dikembalikan. Jika tidak cocok, algoritma akan mencari kunci berikutnya dengan menggunakan teknik linear probing sampai menemukan kunci yang cocok atau menemui sel kosong.
- Contoh penggunaan:
  - Membuat objek `hash_table` dari kelas `OpenAddressingHashTable` dengan ukuran tabel hash 10.
  - Memasukkan beberapa pasangan kunci-nilai ke dalam tabel hash menggunakan metode `insert()`.
  - Melakukan pencarian nilai terkait dengan kunci tertentu menggunakan metode `search()` dan mencetak hasilnya.

Program ini menggunakan teknik *open addressing* untuk menangani *collision* dalam tabel hash, di mana algoritma mencari lokasi selanjutnya yang kosong untuk menyimpan pasangan kunci-nilai baru jika sel yang dituju sudah terisi.

Dalam contoh-contoh di atas, *Separate Chaining* menggunakan linked list untuk menangani *collision*, sementara *Open Addressing* mencari lokasi kosong dalam tabel hash untuk menempatkan entri baru saat *collision* terjadi.



### 9.4.3 Double Hashing

*Double hashing* adalah teknik penyelesaian *collision* yang digunakan dalam tabel hash. Ini bekerja dengan menggunakan dua fungsi hash untuk menghitung dua nilai hash yang berbeda untuk kunci yang diberikan. Fungsi hash pertama digunakan untuk menghitung nilai hash awal, dan fungsi hash kedua digunakan untuk menghitung langkah ukuran untuk urutan penyelidikan.

*Double hashing* memiliki kemampuan untuk memiliki tingkat tabrakan yang rendah, karena menggunakan dua fungsi hash untuk menghitung nilai hash dan langkah ukuran. Ini berarti bahwa probabilitas terjadinya tabrakan lebih rendah daripada teknik penyelesaian tabrakan lainnya seperti penyelidikan linier atau penyelidikan kuadrat.

Namun, *double hashing* memiliki beberapa kekurangan. Pertama, itu memerlukan penggunaan dua fungsi hash, yang dapat meningkatkan kompleksitas komputasi dari operasi penyisipan dan pencarian. Kedua, itu memerlukan pemilihan yang baik dari fungsi hash untuk mencapai kinerja yang baik. Jika fungsi hash tidak dirancang dengan baik, tingkat tabrakan masih bisa tinggi.

```
# Python3 program to double hashing

from typing import List
import math

MAX_SIZE = 10000001

class DoubleHash:
    def __init__(self, n: int):
        self.TABLE_SIZE = n
        self.PRIME = self.__get_largest_prime(n - 1)
        self.keysPresent = 0
        self.hashTable = [-1] * n

    def __get_largest_prime(self, limit: int) -> int:
        is_prime = [True] * (limit + 1)
        is_prime[0], is_prime[1] = False, False
        for i in range(2, int(math.sqrt(limit)) + 1):
            if is_prime[i]:
                for j in range(i * i, limit + 1, i):
                    is_prime[j] = False
        for i in range(limit, -1, -1):
            if is_prime[i]:
                return i

    def __hash1(self, value: int) -> int:
        return value % self.TABLE_SIZE

    def __hash2(self, value: int) -> int:
```

```

        return self.PRIME - (value % self.PRIME)

    def is_full(self) -> bool:
        return self.TABLE_SIZE == self.keysPresent

    def insert(self, value: int) -> None:
        if value == -1 or value == -2:
            print("ERROR : -1 and -2 can't be inserted in the
table")
            return
        if self.is_full():
            print("ERROR : Hash Table Full")
            return
        probe, offset = self.__hash1(value),
self.__hash2(value)
        while self.hashTable[probe] != -1:
            if -2 == self.hashTable[probe]:
                break
            probe = (probe + offset) % self.TABLE_SIZE
        self.hashTable[probe] = value
        self.keysPresent += 1

    def erase(self, value: int) -> None:
        if not self.search(value):
            return
        probe, offset = self.__hash1(value),
self.__hash2(value)
        while self.hashTable[probe] != -1:
            if self.hashTable[probe] == value:
                self.hashTable[probe] = -2
                self.keysPresent -= 1
                return
            else:
                probe = (probe + offset) % self.TABLE_SIZE

    def search(self, value: int) -> bool:
        probe, offset, initialPos, firstItr =
self.__hash1(value), self.__hash2(value), self.__hash1(value),
True
        while True:
            if self.hashTable[probe] == -1:
                break
            elif self.hashTable[probe] == value:
                return True
            elif probe == initialPos and not firstItr:
                return False
            else:
                probe = (probe + offset) % self.TABLE_SIZE
                firstItr = False
        return False

    def print(self) -> None:
        print(*self.hashTable, sep=', ')

if __name__ == '__main__':
    myHash = DoubleHash(13)

    # Inserts random element in the hash table

```

```

insertions = [115, 12, 87, 66, 123]
for insertion in insertions:
    myHash.insert(insertion)
print("Status of hash table after initial insertions : ",
end="")
myHash.print()

# Searches for random element in the hash table, and prints
them if found.
queries = [1, 12, 2, 3, 69, 88, 115]
n2 = len(queries)
print("\nSearch operation after insertion : ")

for i in range(n2):
    if myHash.search(queries[i]):
        print(queries[i], "present")

# Deletes random element from the hash table.
deletions = [123, 87, 66]
n3 = len(deletions)

for i in range(n3):
    myHash.erase(deletions[i])

print("Status of hash table after deleting elements :
",end='')
myHash.print()

```

Program di atas merupakan implementasi dari struktur data tabel hash dengan metode pencarian double hashing. Berikut adalah langkah-langkahnya:

- Kode dimulai dengan mendefinisikan kelas DoubleHash, yang memiliki beberapa metode untuk mengelola tabel hash.
- Konstruktor kelas DoubleHash digunakan untuk inisialisasi tabel hash dengan ukuran yang ditentukan dan menentukan bilangan prima terbesar yang kurang dari ukuran tabel hash.
- Metode `__get_largest_prime` digunakan secara internal untuk mendapatkan bilangan prima terbesar yang lebih kecil dari batas yang ditentukan.
- Metode `__hash1` dan `__hash2` digunakan untuk menghitung nilai hash berdasarkan dua fungsi hash yang berbeda.
- Metode `is_full` digunakan untuk memeriksa apakah tabel hash penuh atau tidak.
- Metode `insert` digunakan untuk memasukkan nilai ke dalam tabel hash. Jika tabel penuh, akan ditampilkan pesan kesalahan. Jika tidak, nilai akan dimasukkan ke dalam tabel menggunakan metode double hashing.

- Metode erase digunakan untuk menghapus nilai dari tabel hash. Jika nilai tidak ditemukan, metode ini tidak melakukan apa-apa. Jika ditemukan, nilai akan ditandai sebagai nilai yang dihapus dengan mengubahnya menjadi -2.
- Metode search digunakan untuk mencari nilai dalam tabel hash. Metode ini menggunakan double hashing untuk mencari nilai dan mengembalikan True jika nilai ditemukan, dan False jika tidak.
- Metode print digunakan untuk mencetak isi tabel hash.
- Dalam blok `__main__`, sebuah objek `myHash` dari kelas `DoubleHash` dibuat dengan ukuran 13.
- Beberapa nilai dimasukkan ke dalam tabel hash menggunakan metode `insert`.
- Nilai-nilai yang dimasukkan dicetak menggunakan metode `print`.
- Beberapa nilai dicari dalam tabel hash menggunakan metode `search`, dan hasilnya dicetak.
- Beberapa nilai dihapus dari tabel hash menggunakan metode `erase`, dan isi tabel hash setelah penghapusan dicetak kembali.

#### 9.4.4 *Rehashing*

*Rehashing* adalah proses meningkatkan ukuran hashmap dan redistribusi elemen-elemen ke ember baru berdasarkan nilai hash baru mereka. Ini bertujuan meningkatkan kinerja hashmap dan mencegah tabrakan yang disebabkan oleh faktor beban yang tinggi.

Saat hashmap penuh, faktor beban meningkat, yang bisa menyebabkan tabrakan dan kinerja yang buruk. *Rehashing* dapat mengatasi ini dengan mengubah ukuran hashmap dan redistribusi elemen-elemen ke ember baru. Proses ini memakan waktu tetapi penting untuk menjaga efisiensi hashmap.

*Rehashing* diperlukan untuk mencegah tabrakan dan menjaga efisiensi struktur data hashmap. Meskipun mahal dalam hal waktu dan ruang, ini penting untuk menjaga kinerja hashmap.

Proses *rehashing* melibatkan pengecekan faktor beban setiap kali entri baru ditambahkan. Jika faktor beban melebihi batas tertentu, rehashing dilakukan dengan membuat array ember baru dengan ukuran dua kali lipat dari sebelumnya dan menyimpan ulang semua elemen ke dalamnya.

```

# Python3 program to implement Rehashing

class Map:

    class MapNode:
        def __init__(self, key, value):
            self.key=key
            self.value=value
            self.next=None

    # The bucket array where
    # the nodes containing K-V pairs are stored
    buckets=list()

    # No. of pairs stored - n
    size=0

    # Size of the bucketArray - b
    numBuckets=0

    # Default loadFactor
    DEFAULT_LOAD_FACTOR = 0.75

    def __init__(self):
        Map.numBuckets = 5

        Map.buckets = [None]*Map.numBuckets

        print("HashMap created")
        print("Number of pairs in the Map: " + str(Map.size))
        print("Size of Map: " + str(Map.numBuckets))
        print("Default Load Factor : " +
str(Map.DEFAULT_LOAD_FACTOR) + "\n")

    def getBucketInd(self, key):

        # Using the inbuilt function from the object class
        hashCode = hash(key)

        # array index = hashCode%numBuckets
        return (hashCode % Map.numBuckets)

    def insert(self, key, value):

        # Getting the index at which it needs to be inserted
        bucketInd = self.getBucketInd(key)

        # The first node at that index
        head = Map.buckets[bucketInd]

        # First, loop through all the nodes present at that
index
        # to check if the key already exists
        while (head != None):

            # If already present the value is updated

```

```

        if (head.key==key):
            head.value = value
            return
        head = head.next

    # new node with the K and V
    newElementNode = Map.MapNode(key, value)

    # The head node at the index
    head = Map.buckets[bucketInd]

    # the new node is inserted
    # by making it the head
    # and it's next is the previous head
    newElementNode.next = head

    Map.buckets[bucketInd]= newElementNode

    print("Pair(\" {} \", \" {} \") inserted
    successfully.".format(key,value))

    # Incrementing size
    # as new K-V pair is added to the map
    Map.size+=1

    # Load factor calculated
    loadFactor = (1* Map.size) / Map.numBuckets

    print("Current Load factor = " + str(loadFactor))

    # If the load factor is > 0.75, rehashing is done
    if (loadFactor > Map.DEFAULT_LOAD_FACTOR):
        print(str(loadFactor) + " is greater than " +
        str(Map.DEFAULT_LOAD_FACTOR))
        print("Therefore Rehashing will be done.")

    # Rehash
    self.rehash()

    print("New Size of Map: " + str(Map.numBuckets))

    print("Number of pairs in the Map: " + str(Map.size))
    print("Size of Map: " + str(Map.numBuckets))

def rehash(self):

    print("\n***Rehashing Started***\n")

    # The present bucket list is made temp
    temp = Map.buckets

    # New bucketList of double the old size is created
    buckets =(2 * Map.numBuckets)

    for i in range(2 * Map.numBuckets):
        # Initialised to null

```

```

        Map.buckets.append(None)

        # Now size is made zero
        # and we loop through all the nodes in the original
bucket list(temp)
        # and insert it into the new list
        Map.size = 0
        Map.numBuckets *= 2

        for i in range(len(temp)):

            # head of the chain at that index
            head = temp[i]

            while (head != None):
                key = head.key
                val = head.value

                # calling the insert function for each node in
temp
                # as the new list is now the bucketArray
                self.insert(key, val)
                head = head.next

        print("\n***Rehashing Ended***")

    def printMap(self):

        # The present bucket list is made temp
        temp = Map.buckets

        print("Current HashMap:")
        # loop through all the nodes and print them
        for i in range(len(temp)):

            # head of the chain at that index
            head = temp[i]

            while (head != None):
                print("key = \" {} \", val = {}".format(head.key, head.value))

                head = head.next
            print()

if __name__ == '__main__':
    # Creating the Map
    map = Map()

    # Inserting elements
    map.insert(1, "Geeks")
    map.printMap()

    map.insert(2, "forGeeks")

```

```

map.printMap()

map.insert(3, "A")
map.printMap()

map.insert(4, "Computer")
map.printMap()

map.insert(5, "Portal")
map.printMap()

```

Program tersebut merupakan implementasi dari struktur data *hashmap* dengan teknik *rehashing* di dalam bahasa pemrograman Python 3. Berikut adalah langkah-langkah alur program tersebut:

- Program dimulai dengan mendefinisikan sebuah kelas Map yang berisi beberapa metode dan kelas bersarang MapNode yang digunakan untuk merepresentasikan simpul dalam hashmap.
- Di dalam kelas Map, terdapat variabel statis seperti buckets, size, numBuckets, dan DEFAULT\_LOAD\_FACTOR yang digunakan untuk menyimpan informasi tentang hashmap.
- Metode `__init__` digunakan untuk menginisialisasi hashmap dengan ukuran awal 5 bucket dan mengatur beberapa variabel yang diperlukan.
- Metode `getBucketInd` digunakan untuk mengembalikan indeks bucket di mana kunci akan disimpan berdasarkan hash codenya.
- Metode `insert` digunakan untuk memasukkan pasangan kunci-nilai baru ke dalam hashmap. Jika load factor melebihi 0.75, maka dilakukan rehashing.
- Metode `rehash` digunakan untuk melakukan rehashing ketika load factor melebihi batas tertentu. Rehashing dilakukan dengan menggandakan ukuran bucket dan memindahkan semua elemen ke dalam bucket baru.
- Metode `printMap` digunakan untuk mencetak isi hashmap setelah setiap operasi insert.
- Pada bagian akhir program, dilakukan pembuatan objek map dari kelas Map, di mana beberapa elemen dimasukkan ke dalam hashmap menggunakan metode `insert`, dan setiap kali setelah operasi insert dilakukan, isi hashmap dicetak menggunakan metode `printMap`.



## Latihan

---

Bagaimana mengurutkan N elemen menggunakan array hash? Dengan nilai **Masukan** : 8 2 4 6 7 1 dan nilai **Keluaran** : 1 2 4 6 7 8. Implementasikan ke dalam bahasa pemrograman python !

Penjelasan pengurutan menggunakan hash:

- Langkah 1: Buat array hash dengan ukuran (max\_element), karena itulah jumlah maksimum yang kita perlukan
- Langkah 2: Telusuri semua elemen dan hitung jumlah kemunculan elemen tertentu.
- Langkah 3: Setelah menghitung kemunculan semua elemen dalam tabel hash, cukup lakukan iterasi dari 0 hingga max\_element dalam array hash
- Langkah 4: Saat melakukan iterasi dalam array hash, jika kami menemukan nilai yang disimpan pada posisi hash mana pun lebih dari 0, yang menunjukkan bahwa elemen tersebut ada setidaknya satu kali dalam daftar elemen asli.
- Langkah 5: Hash[i] memiliki hitungan berapa kali suatu elemen ada dalam daftar, jadi ketika >0, kami mencetak berapa kali elemen tersebut.

Jika Anda ingin menyimpan elemen, gunakan array lain untuk menyimpannya secara terurut.

Jika kita ingin mengurutkannya dalam urutan menurun, kita cukup menelusuri dari max ke 0 dan ulangi prosedur yang sama.