Hash Maps

Hash maps are a common data structure used to store key-value pairs for efficient retrieval. A value stored in a hash map is retrieved using the (unique) key under which it was stored.

Hashmaps are very useful. They allow for reading and working with large datasets when looking for keys or values and make it faster to search for specific values in 0(1), sometimes in O(n) [worst cases].

Hash tables or hash maps in Python are implemented through the built-in dictionary data type. The keys of a dictionary in Python are generated by a hashing function. The elements of a dictionary are not ordered and they can be changed.

Let's see an example:

```
states ={  'TN': "Tennessee",
          'CA': "California",
          'NY': "New York",
          'FL': "Florida" }
west_coast state = states['CA']
```

Hash function

Hash map data structures use a hash function, which turns a key into an index within an underlying array. The hash function can be used to access an index when inserting a value or retrieving a value from a hash map

Hash map underlying data structure

Hash maps are built on top of an underlying array data structure using an indexing system.

Each index in the array can store one key-value pair. If the hash map is implemented using chaining for collision resolution, each index can store another data structure such as a linked list, which stores all values for multiple keys that hash to the same index.

hash map only one value

Each Hash Map key can be paired with only one value. However, different keys can be paired with the same value.

Example:

```
#This is a valid Hash Map where 2 keys share the same value

correct_hash_map = {"a" : 1,
                    "b" : 3,
                    "c" : 1}


#This Hash Map is INVALID since a key cannot have more than 1 value

incorrect_hash_map = { "a" : 1,
                       "a" : 3,
                       "b" : 2 }
```

This will bring us to … (Collision)

while inserting data into the hash map, if the index for the key is already occupied by another data(key-value), a collision arises. A good hash function is such that, it minimizes the chances of collision.
 # collisions can be avoided by the below methods.

1. Separate chaining (open hashing).

Chaining: If the hash value (index) of the two keys is the same, the data of the second key is added to the end of the first key in a chain fashion and it goes on. The chaining can be done with help of any other data structures like Linked List, List, or Self Balancing BST (AVL Trees, Red-Black Trees). In chaining, usually, the keys are stored separately outside the hashtable. The access time of keys for the following data structure varies from O(1) to O(n) for Linked List & List and O(logn) for Tree.

2. Open addressing (closed hashing)

If the index is already allocated by another key value, it will probe for the next empty slot in the hash map to allocate the new key value. The probing can be done in different ways.

- **Linear probing**: Searching for the next free slot sequentially from the starting index to the index just before it in a circular manner.
  - pros: very good cache performance.
  - cons: primary & secondary **clustering** will occur.

Clustering: The scenario when sequential slots of an index will be allocated very soon for a specific key/Index as the different data(key-value) of the same hash index comes for insertion**.**

- **Quadratic probing**: Searching for the next free slot in a quadratic manner. for (i^2)th slot in ith iteration.
  - pros: No primary clustering.
  - cons: secondary clustering will happen, average cache performance.
- **Double hashing**: use a second hash function to find the next free slot on the index that got from the first hash function.
  - pros: No primary & secondary clustering.
  - cons: poor cache performance.