Hashing

Introduction

- **Hashing** is designed to solve the problem of efficiently finding or storing an item in a collection.
- It involves using a function known as Hash Function to map object data to some representative integer value known as **Hash Code** or **Hash Value**.
- A table known as a **Hash Table** is used for saving all the key and value pairs. Keys are used for finding the needed data from the table.
- Hash tables have to support three operations: **insert(key, value), get(key)** and **delete(key)**.

Click here for Hashing Key Terms

Collision Handling

Let's understand the concept of collisions with an example.

Suppose that we want to map a list of string keys to string values (for example, map a list of countries to their capital cities). Now we have to store this information in a hash table.

We take our Hash function to store the information as simply the length of the country's name.

So if we have to perform insert("France", "Paris"), since "France" has six characters in it, the information is stored at the sixth index in the table. Following this pattern let's add a few entries to the table.

Position	Country (Key)	Capital (Value)
1		
2		
3		
4	Fiji	Suva

5	India	New Delhi
6	France	Paris
7	Tunisia	Tunis
8		

Now let's say that the next entry to be made is for "Spain". This entry has a hash value of 5 (length of string being 5). But as we can see in the table above, 5th position is already occupied by "India" which also happens to have the same hash value. This has resulted in what is known as **Collision**.

A very important part of hashing is to handle the collisions so that the hash table can store all values. This is what is known as **Collision Handling**.

Hashing Techniques

No matter how good a hash function is collisions are bound to happen. The target is minimize them as much as possible. There are two methods to handle collisions:

- Separate Chaining
- Open Addressing

1. Separate Chaining

Make each slot of hash table point to a linked list of records that have the same hash value.

Performance Analysis: Hashing performance can be evaluated under the assumption that each key is equally likely and uniformly hashed to any slot of hash table.

- Consider table_size as number of slots in hash table and n as number of keys to be saved in the table.
- Then: We have Load factor α = n/table_size
- Time complexity to search/delete = O(1 + α)
- Time complexity for insert = O(1)
- Hence, overall time complexity of search, insert and delete operation will be O(1) if α is O(1)

Advantages:

- Simple implementation.

- No concern about hash table filling up.
- Less sensitive of hash functions and load factors.
- Used when we do not know exactly how many and how frequently the keys would be inserted or deleted.

Disadvantages:

- In the worst case scenario, it might happen that all the keys to be inserted belong to a single bucket. This would result in a linked list structure and the search time would be O(n).
- Some parts of hash table might never be used.
- Extra memory needed for creating and maintaining links.

2. Open Addressing

In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Following are the techniques of Open Addressing:

- Linear probing
- Quadratic probing
- Double hashing

Read more about the techniques mentioned above here