Maps, Hash tables, and Hash functions

Syed Eqbal Alam

University of New Brunswick
Fredericton, New Brunswick, Canada

Overview

1. Maps
2. Hash tables
3. Hash functions
   - Hash code
   - Compression function
4. Collision
   - Collision avoidance techniques
     - Separate chaining
     - Open addressing
   - Probing: Linear and Quadratic
5. Load factor
   - Table doubling

Maps

- A map is an Abstract Data Type; it stores a collection of keys and their corresponding values.
- A Map is also referred as *associative array*.
- Keys in a map can be numeric, alphabetic, or alphanumeric, etc.
- A Key is unique to a value. The (key, value) pair is called an entry.

Example (Map)

Student Id and students' data. Here, the Student Id is the key that is associated with a particular student's data.

Student ID	Student's data
CS 4122	Student 1
CS 4123	Student 2
CS 4124	Student 3

Maps

In Java:

- A Map is an interface in java.util package.
- The Map interface is replaced by the Dictionary class, which was an abstract class, refer `https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`.
- The Map can be implemented as a HashMap class, TreeMap class, etc.

Some of the methods supported by Map are:

(i) isEmpty(): Returns whether a Map object is empty or not (true or false).

(ii) size(): Returns the number of entries in the map.

(iii) put(key, value): Inserts the key and its value in a map.

(iv) get(key): Returns the value corresponding to the key.

Maps

A few more methods supported by Map are:

(v) remove(*key*, value): Deletes the entry associated with the *key*.

(vi) replace(key, old value, new value): Replaces the old value with the new value associated with the key.

(vii) values(): Returns all entries' values in the map.

(viii) keySet(): Returns set of keys of the entries.

(ix) entrySet(): Returns the set of entries, keys and values.

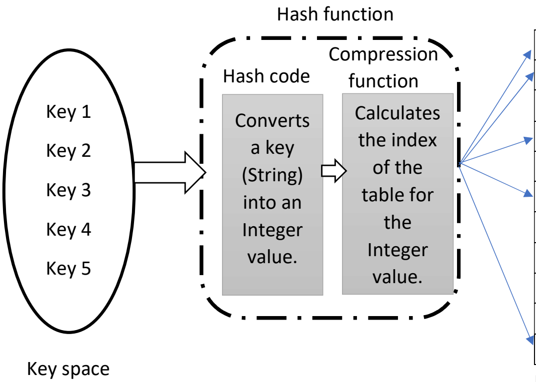
(x) containsValue(value): Checks whether the value exists or not in the map. It returns true or false.

(xi) containsKey(key): Checks whether the key exists or not in the map. It returns true or false.

Hash tables or Hash map

- Hash table or Hash map is an abstract data type that efficiently implements a map to store keys and their corresponding values.
- As the keys can be alphabetic or alphanumeric, we need ways to convert keys to integer values and then to indices of the table.
- Hash table uses hash functions to do this. Hash functions calculate an integer value for a key (String) and then an index of the table from the integer value.
- get(), put(key, value), remove(key, value) can be implemented in $O(1)$—conditional, we will see more details.
- Hash tables are used in database indexing.
- Hashing is used in pattern matching in strings.
- Hashing is also used in compiler design, cryptography, and many other places.

Keys to Hash table mapping



0	Key 1	Data 1
1	Key 2	Data 2
2		
3	Key 5	Data 5
4		
5	Key 4	Data 4
6		
7		
8		
9		
10	Key 3	Data 3

Index

Table

Hash functions

- The hash function is used to find indices of a table corresponding to keys. Then the (key, value) is entered in the Hash table at that index.
- Hash function consists of two components:
  - Function to generate hash code
  - Compression function

Remark

Keys are used to find corresponding integer values, called the hash code.

Example (In Java)

String key = "CS 2383";
System.out.println("Hash code of CS 2383: "+key.hashCode());
Output: Hash code of CS 2383: 1740517036

Remark

The compression function is then used to find the index for the Hash table corresponding to the hash code obtained.

Hash code

Techniques to find out hash code:

(i) Bit-wise representation of keys as integer values:

- Find out the bit-wise integer values of the key and then sum them up to obtain the hash code.

Remark

This technique is not suitable for Strings. It provides the same integer values if the string consists of the same characters; for example, TOY, YOT, OTY, etc.

Hash code generation techniques

(ii) Polynomial hash code:

- This technique considers the relative positions of characters in a string. Let the string consists of $(y_1, y_2, \ldots, y_n)$ characters in order, then its polynomial hash code is calculated as follows, for an integer $x \neq 1$:

$$hashcode = y_1 x^n + y_2 x^{n-1} + \ldots + y_n.$$

(iii) Cyclic bit-wise shift:

- In this technique initial few bits of the key are shifted to the last bits to generate different hash codes.

Example (Method in Java to obtain binary value of a string)

String binary_val = Integer.toBinaryString(key.hashCode());

Example (Cyclic bit-wise shift)

In 0100 1111 1001 1110 0000 0001 1010 0011, the initial four bits are shifted to the right to obtain 1111 1001 1110 0000 0001 1010 0011 0100.

Compression function

- The hash code obtained could be very long, so it is not used directly as an index of the Hash table.
- Therefore, we need techniques to calculate indices suitable for a Hash table's length.

Compression techniques: Let the capacity of the Hash table is N and the hash code for a key be $h(key)$.

(i) The division method: To find the index corresponding to hash code h(key), we perform the following basic task

$$index = h(key) \mod N. \tag{1}$$

- This technique can generate one index for multiple keys. This is called *Collision*.

Example

$\{2, 3, 5, 8, 17\} \mod 5 = \{2, 3, 0, 3, 2\}$. Here, 2 and 17 point to the same index 2 and 3 and 8 point to the same index 3.

Compression function

Compression techniques: Let the capacity of the Hash table is N and the hash code for a key be $h(key)$.

(i) The multiply, add and divide method: For a prime number $q > N$ and $0 < x < q$ and $0 \leq y < q$, we calculate the index as follows:

$$index = ((h(key) \times x + y) \mod q) \mod N. \tag{2}$$

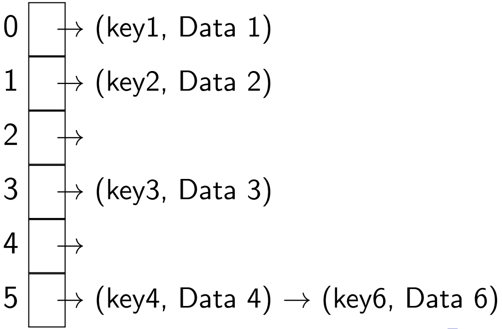
Collision

Collision:

- Hash function producing the same index values for two or multiple keys.
- A good hash function minimizes collision.

Example (Collision)

Suppose key4 and key6 get index 5. Which entry should go at index 5?



Collision avoidance techniques

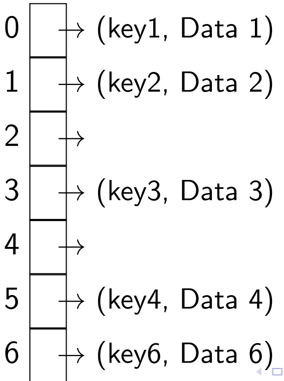
(i) Separate chaining: If the same index is obtained for multiple keys, then they will be inserted in the same bucket of the Hash table (at the same index). As index 5 of the table.

0	$\rightarrow$	(key1, Data 1)
1	$\rightarrow$	(key2, Data 2)
2	$\rightarrow$	
3	$\rightarrow$	(key3, Data 3)
4	$\rightarrow$	
5	$\rightarrow$	(key4, Data 4) $\rightarrow$ (key6, Data 6)

Collision avoidance techniques: Open addressing

(ii) Open addressing:

- One entry per bucket.
- If the allocated index is occupied, look for empty buckets adjacent to the allocated index.
- Key 4 and 6 have the same index value 5. Index 5 is occupied by the entry of *key*4; the adjacent index 6 was empty so entry is made at index 6.



Probing: Linear and Quadratic

In linear probing:

- If two keys are assigned the same index, then check the next/adjacent index, if it is occupied check the next, repeat this until an unoccupied index is found then insert the entry:

$$\text{insert at } \left((h(key) + j) \mod N\right), \quad j = 0, 1, 2, \ldots.$$

- Linear probing creates accumulation of several entries adjacent to each other but other places may be empty, called *clustering*.

Quadratic probing:

- The table is probed for an empty bucket based on the following rule:

$$\text{insert at } \left((h(key) + j^2) \mod N\right), \quad j = 0, 1, 2, \ldots.$$

- Clustering created by Quadratic probing is called *secondary clustering*.

Open addressing: Double hashing

- In double hashing, a new hash function is created. Let us denote it by $h_{new}(key)$. For a prime number $q < N$, we have

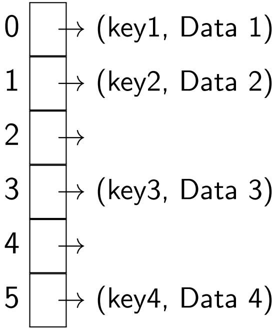
  $$\text{insert at index}\,((h(key) + j \times h_{new}(key)) \mod N)\,, \quad j = 0, 1, 2, \ldots,$$

  where

  $$h_{new}(key) = (q - (key \mod q)).$$

Load factor

- Load factor is defined as the ratio of the number of entries in the map and the capacity (or size) of the Hash table.
- Let N denote the capacity of the table, and let m denote the total number of entries in the table. The load factor is defined as $\lambda = \frac{m}{N}$.
- Load factor of the following Hash table is $4/6 = 0.6667$.



Load factor and Table doubling

- Separate chaining works well when load factor $\lambda < 0.9$ [1].
- Open addressing works well when load factor $\lambda < 0.5$ [1].

Table doubling:

- Increase the capacity of the Hash table when λ is above the threshold.
- Table doubling: Increase the capacity of the Hash table to $2N$.
- For the new Hash table, we need to find out the indices corresponding to the keys.
- Although the hash code will be the same for the keys, but we need to calculate the compression values based on the new capacity N of the Hash table.

Michael T. Goodrich and Roberto Tamassia and Michael H. Goldwasser, *Data Structures and Algorithms in Java*, 6th, 2014, Wiley.