Data
Structures
and
Algorithms

Hashing

Contents

1. What is Hashing?
Hashing is an algorithm that maps data sets (keys) to smaller data sets of a fixed set length.

1.1 Array Implementation
insert(key, data): arr[key] = data
delete(key): arr[key] = null
find(key): arr[key]

The issues with this is that the key values needs to be **non-negative integer values.**
For example, the keys cannot be 1.13, 13S, CS2040 etc etc.
Range of keys furthermore must be **small** for example 100...

2. Hash Table

Hashing is an algorithm that maps data sets (keys) to smaller data sets of a fixed set length.

2.1 Hash Functions

*Given a function h(key), where it maps large to **smaller** integers, and maps **non-integer** to integer values. Implementation is as followed:*

insert(key, data): arr[h(key)] = data
delete(key): arr[h(key)] = null
find(key): arr[h(key)]

A hash function does not usually guarantee **two different keys** to go into **different** slots.
Thus, a hash function is known as a **many-to-one** mapping.

2.2 Collisions

Collisions occur when two different keys are mapped into the same slot.
For example, given a function where h(key) = key % 11, keys 11 and 22 will collide as
11 % 11 = 22 % 11 = 0

Thus, to rectify collisions, good hash functions must be built.

2.3 Good Hash Functions

It is **fast** to compute, scatters keys **evenly** throughout the hash table and has **less collisions.**

Perfect hash functions only occur if and only if all keys are known, which results in a **one-to-one** mapping, and thus no collisions will occur.

2.4 Division Method

hash(k) = k % m, where m is the number of slots in the hash table.

m should be a **prime number** closed to a power of two.

2.5 Multiplication Method

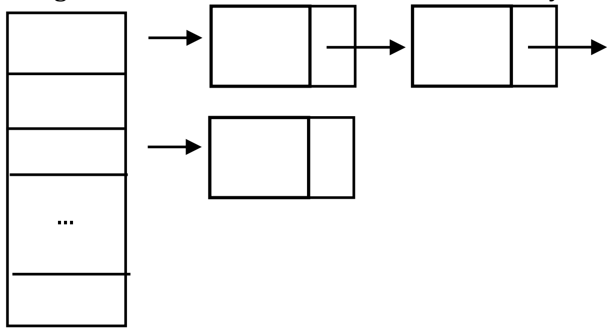
hash(k) = Multiply by a **constant real number** A between 0 and 1, Extract the fractional part, then multiply by m, the hash table size.

3. Collision Resolution

Here are four techniques to solve collisions in hashing.

3.1 Separate Chaining

Using of Linked Lists to store the collided keys



To measure how full is the hash tabled:
use load factor = α = n / m
where n = number of keys and m = number of slots

Insert	O(1 + α)
Delete	O(1 + α/2)
Retrieve	O(1 + α/2) if successful O(1 + α) if unsuccessful

As a result average runtime for Insert, Delete and Retrieve is O(1), given load factor to be a constant.

3.2 Linear Probing

When we get a collision, we find the next empty slot to put the value inside.
For example, hash(k) = k % 7, given 7 slots. Probe sequence is thus
hash(k) = (k + n) % 7, where n is 1, 2, 3... so on and so forth

1) Insert 3: (3 % 7 == 3) -> Check 3; Insert at 3
2) Insert 10: (10 % 7 == 3) -> Check 3; 3 is occupied; Check 4; Insert at 4
3) Insert 17: (17 % 7 == 3) -> Check 3; 3 is occupied; Check 4; 4 is occupied; Check 5; Insert at 5
4) Find 3: (3 % 7 == 3) -> Check 3; Insert at 3
5) Find 31: (31 % 7 == 3) -> Check 3; Not 31; Check 4; Not 31; Check 5; Not 31; Check 6; **Empty**

How to delete though? Do you simply just remove the element?
No! It'll affect your find() or retrieve() function.

Instead, changed the data to null.
6) Delete 17: (17 % 7 == 3) -> Check 3; Not 17; Check 4; Not 17; Check 5; Change 17 to **null**
Hence, when you meet a null when you insert, just replace null with the value

3.2.1 Problems

Primary Clustering: Linear Probing results in **many consecutive occupied slots,** resulting in creased time for find, insert and delete
A way to solve will to be change the hash function from hash(key): (k + n) % m to
hash(key): (k + (n * d)) % m, where d and m are co-prime.

Exercise:

Construct the Hash Tables for Steps 1 to 6 (Answers will not be provided)

3.3 Quadratic Probing
When we get a collision, we find the next empty slot to put the value inside.
For example, hash(k) = k % 7, given 7 slots. Probe sequence is thus
hash(k) = $(k + n^2) \% 7$, where n is 1, 2, 3... so on and so forth

If $\alpha < 0.5$, meaning hash table is less than half full, and m is prime, then we can **always** find an empty slot.

3.3.1 Problems
If two keys have the same initial position, their probe sequences are the same.
This is known as Secondary Clustering.

3.4 Double Hashing
The usage of two hash functions.
$(hash(key) + hash_2(key) * n) \% m$

NOTE: second hash function should **never** evaluate to zero.
So how do we solve this problem?
if $hash_2(key) = k \% m$
change it to $hash_2(key) = m - (k \% m)$, which is always > 0

4. HashMap and HashSet Java API

Note: Use a HashMap when you want to keep track of keys and values.
Note: Use a HashSet when you want to keep track of presence of values only.

https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html
https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

For HashMap:

`boolean`	`contains(Object value)` Tests if some key maps into the specified value in this hashtable.
`boolean`	`containsKey(Object key)` Tests if the specified object is a key in this hashtable.
`boolean`	`containsValue(Object value)` Returns true if this hashtable maps one or more keys to this value.
`Set<Map.Entry<K,V>>`	`entrySet()` Returns a `Set` view of the mappings contained in this map.
`V`	`get(Object key)` Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
`boolean`	`isEmpty()` Tests if this hashtable maps no keys to values.
`Set<K>`	`keySet()` Returns a `Set` view of the keys contained in this map.
`V`	`put(K key, V value)` Maps the specified `key` to the specified `value` in this hashtable.
`V`	`remove(Object key)` Removes the key (and its corresponding value) from this hashtable.
`boolean`	`remove(Object key, Object value)` Removes the entry for the specified key only if it is currently mapped to the specified value.
`V`	`replace(K key, V value)` Replaces the entry for the specified key only if it is currently mapped to some value.
`boolean`	`replace(K key, V oldValue, V newValue)` Replaces the entry for the specified key only if currently mapped to the specified value.
`int`	`size()` Returns the number of keys in this hashtable.
`String`	`toString()` Returns a string representation of this `Hashtable` object in the form of a set of entries, enclosed in braces and separated by the ASCII characters `", "` (comma and space).

For HashSet:

boolean	add(E e) Adds the specified element to this set if it is not already present.
void	clear() Removes all of the elements from this set.
Object	clone() Returns a shallow copy of this HashSet instance: the elements themselves are not cloned.
boolean	contains(Object o) Returns true if this set contains the specified element.
boolean	isEmpty() Returns true if this set contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this set.
boolean	remove(Object o) Removes the specified element from this set if it is present.
int	size() Returns the number of elements in this set (its cardinality).

Important: How to Iterate through a HashMap and a HashSet?

```
for (Map.Entry<Key, Value> entru: hashMap.entrySet()) {
    Key k = entry.getKey();
    Value v = entry.getValue();
}
```

```
Iteratori<Value> it = hashSet.iterator();
while (it.hasNext()) {
    Value v = it.next();
}
```