More about hashing

Hashing has the potential to provide search, insert, and delete all in O(1) time by giving each key its own slot in the array through the hash function h(k)

Note that the hash function must provide a valid array index. To ensure this, we will take the result module the size of the hash table. A[h(k)%m]

However, we will need to resolve collisions.

Collision resolution scheme

- Open addressing

- chaining

All keys are placed in the table typically by placing the new key in a different location.

* Linear probing
* Quadratic probing
* Double hashing

Linear probing: the colliding key is simply placed in the first available slot moving from left to right

h(k,i) = (h(k) + i) % m

i represents the probe number. It starts at 0 for each insert or search and increment each time we have a collision.

When searching for a key, we can stop when we see an empty slot along the probe sequence.

In the example, each key hashed to the same location!! Hashing has degenerated into sequential search, which is O(n)

The problem with linear probing is that it leads to primary clustering. Primary clustering is long runs of filled slots near the hash locations of keys.

In the best base, hashing is O(1)。 In the worst case, it’s O(n).

But we’re concerned with what happens in practice. What do we expect?

Under simple uniform hashing, a key is equally likely to hash to any location. Let’s have @ = n/m denote the load factor of the table ( 0 <= @ <= 1)

Q: how long will insert take?

A: <= 1 + @^1 + @^2 + @^3 + …… = 1/1-@

In array of size m = 1000000, if @ = 0.9, then we expect to 10 tries.

So, the expected insertion cost is O(1)