Hashing → some complexity

Θ(1)

Dictionaries <key, value>

Look up for a key in the dictionary with Θ(1).

values

Word → key that addresses its definitions.

Eg Two-letter words only! 26 alphabets.

26 × 26 = 676 possible different 2-letter words.

Most of the possible combinations are NOT actual words.

≫ than ascii

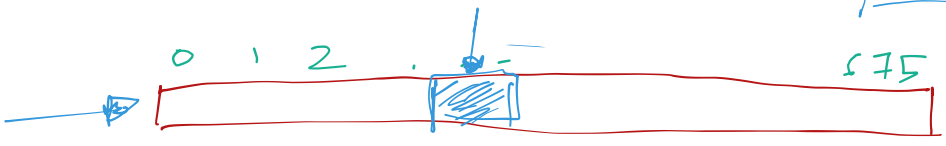
```
class WORD
{
  int hashCode() { returns 26 * (WORDS[0] - 'a') + (WORDS[1] - 'a');
  }
  private: char WORDS[2];
};
```

Encoding

char → int ASCII

0 .. 25

0 .. 675



Definition (Array)

```
class Dictionary {
  public:
    void insert(WORD& w, Value& v) {
      def[w.hashCode()] = v;
    }
    Definition& find(Word& w) {
      return def[w.hashCode()];
    }
  private:
    Definition def;
};
```

Definition of the word.

Θ(1)

Everything is Θ(1) operation

⇒ Looking up is extremely fast.

Problem: Longest word: 45 letters.

26^{45} entries to store all possible words.

~ 700,000 words in the dictionary.

Need to operate on arbitrarily large words!

Hash Tables

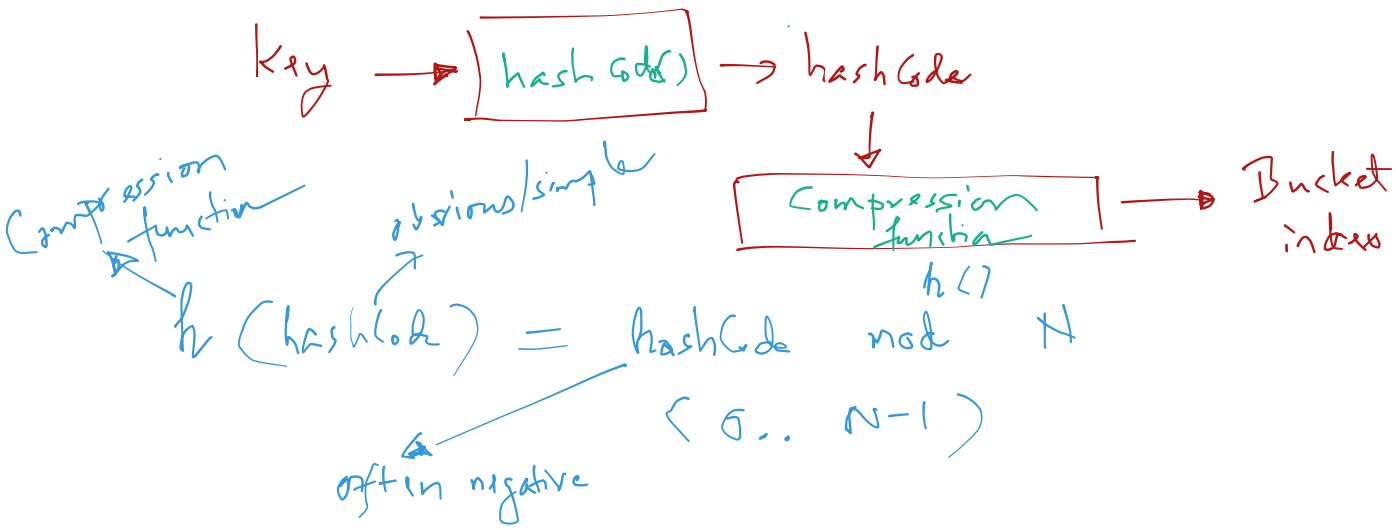
* Insert
* Find
* Remove.

n: Number of keys stored (Actual)

Table of N buckets, where N is a bit larger than n (~20% larger)

Key smaller than the total number of possible keys.

A Hash table maps a huge set of possible keys into N buckets by applying a compression function to every hashCode.



⇒ Likely to have collision

Collision: Several keys hash to the

same bucket.

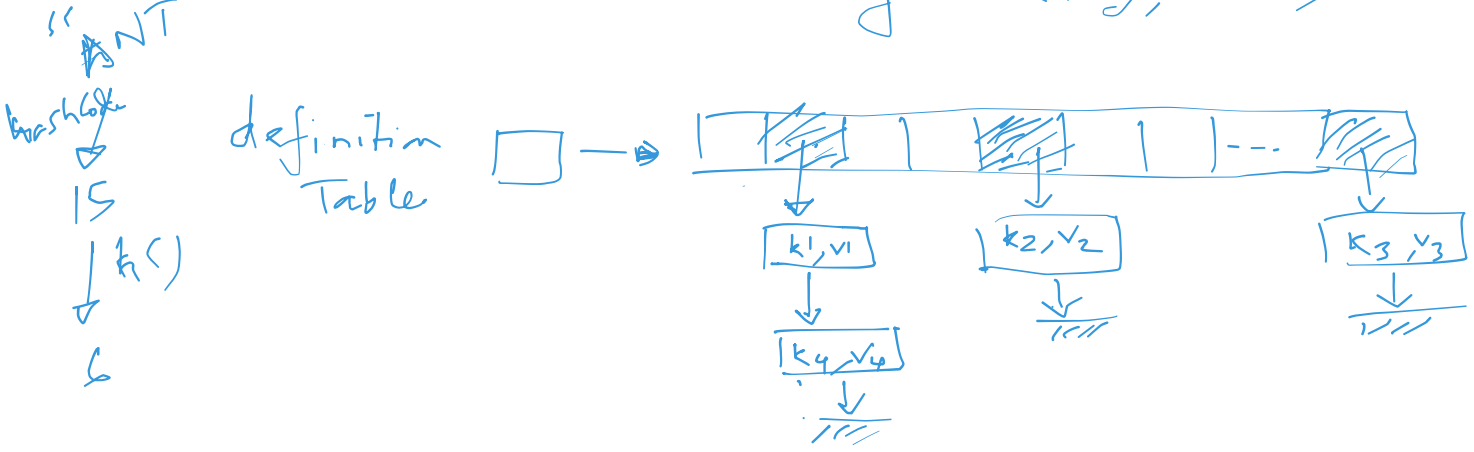
Iff $h(\text{hashCode1}) == h(\text{hashCode2})$

① How to deal with the collision?

Chaining: Each bucket points to a linked list of entries called a chain.

⇒ store both words + definitions in the chain

Entry = <key, value>



Operations:

① Entry* insert (key, value)
- Compute the key's hash code
- Compress to determine the bucket
- Insert the entry into the bucket's chain.

② Entry* find (key)
- Compute the key's hash code & the bucket
- Search the chain for entry with the given key.
- If found, return the entry. Otherwise, return NULL.

③ Entry* remove (key)
- Hash key

std::unordered_map

- Search the chain.
- Remove the entry from the chain.
- Return the entry or NULL.

Multiple copies of the same key?
- Insert both; find can return arbitrarily one of them.
- Replace the old value with the new. → ONLY ONE ENTRY.

Performance of a hash table

Load factor of a hash table = $\frac{n}{N}$ (n → # keys, N → # buckets)

Good load factor ≈ 0.8 or 1.2

If the load factor stays low (close to a small constant) and hash code and compression function are good and no duplicate keys, then the chains are short and each operation takes $O(1)$ time.

If the load factor gets BIG ($n >> N$), (# keys keeps growing) $O(n)$ time.

N is BIG → cost to set up uses a lot of memory.

(2) How to minimize collision? (mod)

(GOOD) Hash Code and Compression function

key → hashCode $\xrightarrow{h()}$ [0..N-1]

17 mod 3 → 2

[mod 3 → 0, 1, 2]

$= 2$

$\frac{N}{3}$ → Take Remainder

$N \bmod 3$ → Remainder when $N \div 3$.

hashCode mod N → $[0..N-1]$

Ideal: Map each key to a random bucket for fewer collisions with each bucket being equally likely.

Bad Compression function

Suppose keys are all ints.

hashCode(i) = i

h(hashCode) = hashCode mod N → 0..N-1

Suppose N = 10000

& keys are divisible by 4.

h() is divisible by 4 too.

⇒ $\left(\frac{3}{4}\right)^{th}$ of the buckets are NEVER used.

Some compression function can be better if N is PRIME.

Better: $h(hashCode) = ((a * hashCode + b) \bmod p) \bmod N$

a, b, p → Positive int.

p is a large prime.

p >> N

mod p → Scrambling of the bits.

mod N → Need not be prime

⇒ N does not need to be a prime.

Good Hash Code for strings

```
int hashCode(String key) {
    int hashVal = 0;
    for (int i = 0; i < key.length(); ++i) {
        hashVal = (127 * hashVal + key[i])
                  % 16908799;
    }
    return hashVal;
}
```

→ 127 ?

⇒ Works well in practice.

Bad hashCode : Creating biases in data

for string keys

(1) Sum ASCII values of characters.

→ Rarely crosses 500.

Most entries mapped into these small set of buckets.

Eg. Anagrams like "pat", "apt", "tap" collide.

(2) First 3 letters of a word with 26^3 buckets.

Biases in English : lot of "pre", "pro", etc.

No word begins with "zxp"...

⇒ Lots of collisions.

(3) Suppose change prime modulus to 127

$(127 * hashVal) \% 127 = 0$

⇒ Only the last char in the

hash Code.

Resizing Hash Tables:

If the load factor $\left(\frac{n}{N}\right)$ gets too big, we lose $O(1)$ time.

- Enlarge the hash table when $\frac{n}{N} > c$ (0.5)
- Allocate new array (at least twice as large) for table
- Walk through the old array, rehash entries into the new hash table for Table.

Cannot simply copy from old chains into the new table.

Compression function changes → Entries go into diff. buckets.

If $\frac{n}{N} < 0.25$, shrink the hash table to free up some memory