Hashtable is a dictionary

put
get
remove

constant time! (?)

table of hashes

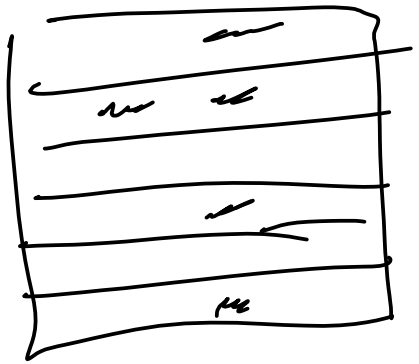
Key $\xrightarrow{\text{hashing}}$ int $\xrightarrow[\%]{\text{fit table}}$ int

key = int

```
int hash(int key) {
    return key;
}
```

```
int hash(bool key) {
    if (key) {
        return 1;
    }
    return 0;
}
```

```
int hash(Pair<A, B> p) {
  int x = hash(p.get_first());
  int y = hash(p.get_second());
  return x + y * 31;
}
```



collision

```
int hash(T* ptr) {
  return hash(*ptr);
}
```

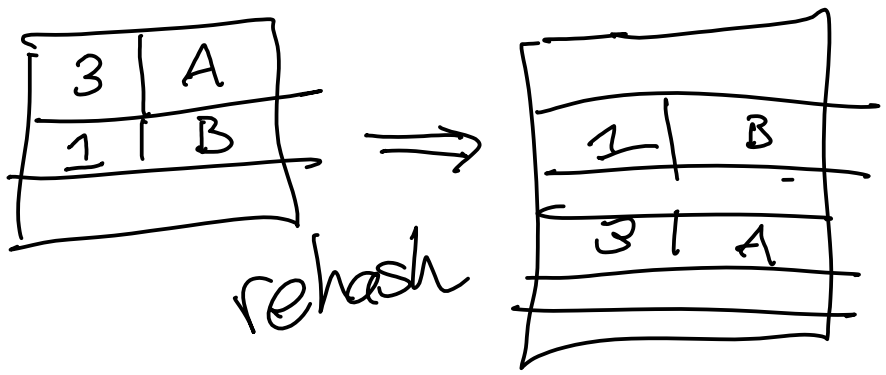
int

Pair<K, V>*

ensure_capacity

load factor < max load factor

(0.6, 0.75, ...)



Triple<K, V, bool>*/ Probing

2 ⇐ -1 ∘ 3 => -1

probing — chaining

Method put(K key, V value):

ensure_capacity(1)

h ← hash(key)

index ← h mod capacity (in c++, fixup)

While contents[index].valid:

If contents[index].key = key:

contents[index].value ← value

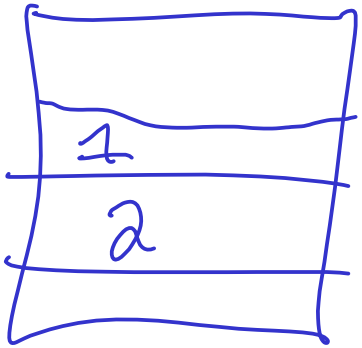
Return

index ← (index+1) mod capacity

contents[index].key ← key

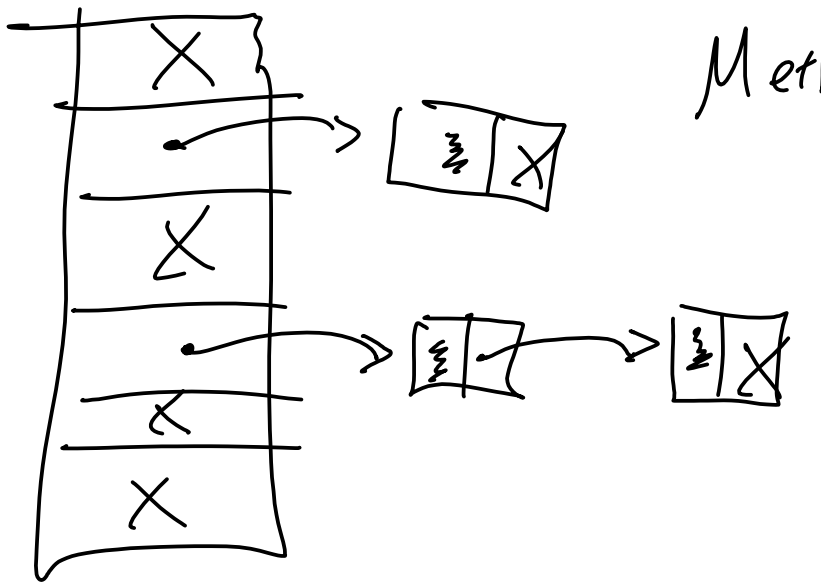
.value ← value

.valid ← true



Chaining

linked list< Pair<K, V> >*



Method put(key, value):

ensure_capacity()

h ← hash(key)

index ← h mod cap

if key appears in contents[i]:

update that pair

Return

contents[i].insert_at_front(

Pair(key, value))

ensure_capacity

if we exceed max load factor:

make new array

rehash & reposition everything

3/5 ⇒ 0

!!

load factor = size / capacity

((float)3)/5 ⇒ 0.6

Method ensure_capacity():

lf ← $\frac{\text{size}+1}{\text{capacity}}$

If lf > mlf:

new_contents ← new array of size 2*cap

For i ← 0...cap-1:

For item in contents[i]:

h ← hash(item.get_first())

index ← h mod (2*cap)

new_contents[index].add(item)

cap ← 2*cap

contents ← new_contents

remember

delete!