Alex Kuk Follow

Mar 4, 2020 · 12 min read · Listen

System design: cache eviction policies (with java impl)

In the previous article we defined what cache is, what we need to know about our data and how to implement different writing policies. Show must go on! And now it's time to show how records could be forced out of the cache.

Why do we need it?

As I mentioned earlier, hashtable is the most suitable data structure for building a cache system. Because it has O(1) access for both put and get operations (if you have a good hash function which provide a uniform distribution of hash values). But hashtable has O(n) occupied memory space. Therefore we need to remove some unnecessary entries and keep only those that are in demand.

Eviction policies

There are a lot of eviction policies:

- *FIFO (First In First Out), LIFO (Last In First Out)* — I think it doesn't need to be explained.

- *LRU (Least Recently Used)* — discards items that have not been used for a long time (more than other elements in the cache).

- *TLRU (Time aware Least Recently Used)* — LRU with a valid life time for the elements.

- *MRU (Most Recently Used)* — discards items that have been used latest.

- *RR (Random Replacement)* — randomly selects a candidate for replacement to make space.

- *SLRU (Segmented LRU)* — two segments of the cache — probationary and protected segments. Both are constructed on LRU concepts. A new value adds to the probationary segment first. When this value is requested again, it will be moved to the protected segment.

- *LFU (Least Frequently Used)* — counts how often an item is needed and discards rare elements.

- *LFRU (Least Frequent Recently Used)* — like SLRU, it has two partitions, but one is LFU and the other is LRU (with the most popular content).

- *LIRS (Low Inter-reference Recency Set)* — replaces items based on their recency.

- *ARC (Adaptive Replacement Cache)* — balances between LRU and LFU. It is based on SLRU, but it can dynamically adjust the size of both segments.

Okay, but what about admission policy?

Oh, you can just add values and not worry.. But you can also calculate entries uses, compare a new value and an evict candidate, and decide whether to add a new value.

But now, let's discuss about replacement policies more precisely: LRU, LFU & SLRU.

1. Least Recently Used

In the cache HashTable we store the keys (by which we're gonna find) and the values (what we're gonna find). But we need to know the sequence of using items to evict the oldest one. Linked list could help us! Just use the nodes of linked list as the values in our cache.

Create nested class Node and override toString() method:

```java
private static class Node<K,V> {
    K key;
    V item;
    Node<K,V> next;
    Node<K,V> prev;

    Node(Node<K,V> prev, K key, V element, Node<K,V> next) {
        this.key = key;
        this.item = element;
        this.next = next;
        this.prev = prev;
    }

    @Override
    public String toString() {
        return " " +
                (prev == null ? null : prev.item) + " <<-- " +
                item +
                " -->> " + (next == null ? null : next.item);
    }
}
```

CacheEvictNode.java hosted with ❤️ by **GitHub** **view raw**

Our cache class will look like this:

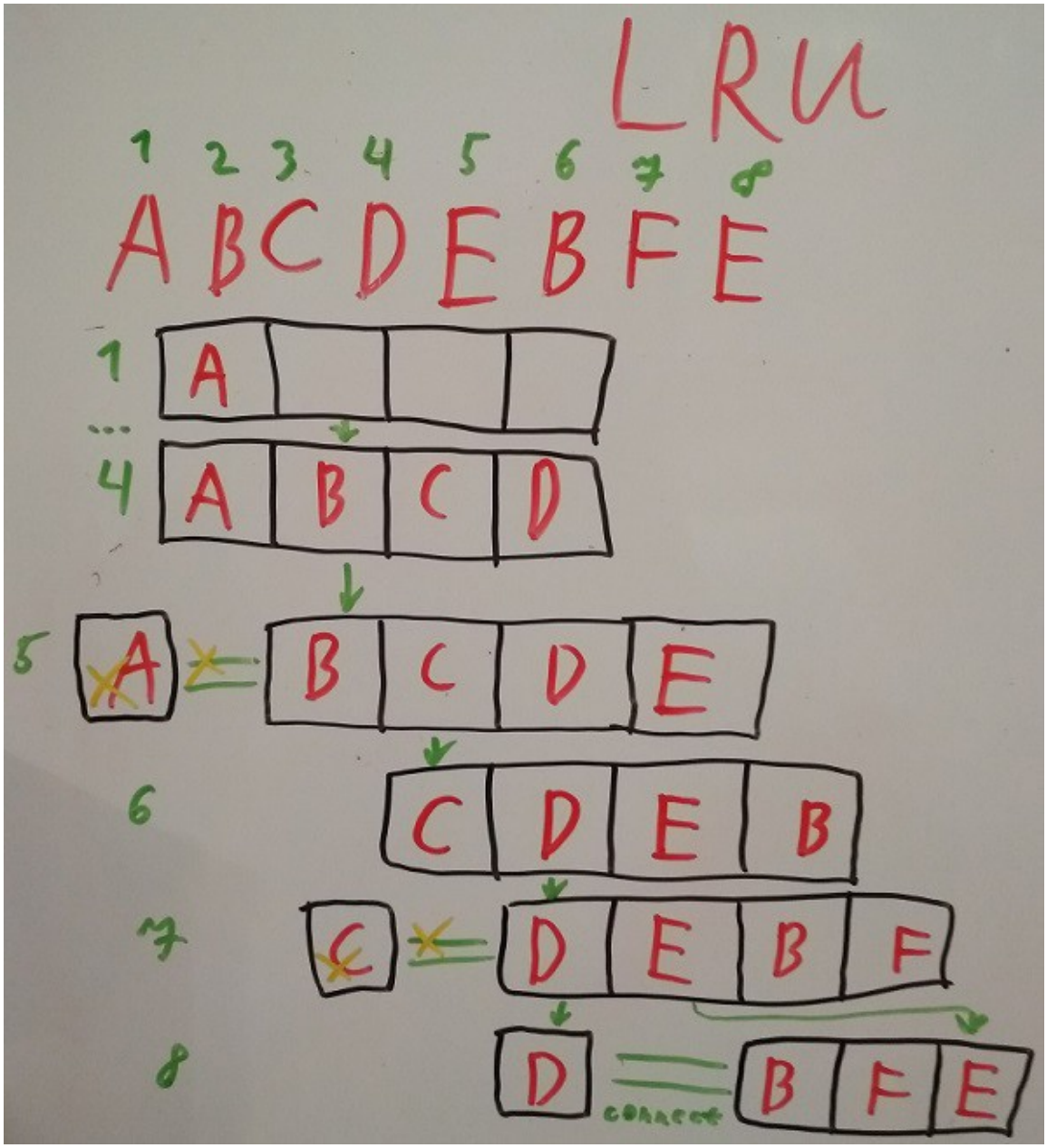
```java
public class LRUCache<K,V> implements Cache<K,V> {

……
    private final int capacity;
    private Node<K,V> head;
    private Node<K,V> tail;
    private final Map<K, Node<K,V>> cache;

    LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
    }
……
}
```

CacheEvictLRU.java hosted with ❤️ by **GitHub** **view raw**

Suppose we have a sequence to set/get values to/from a cache "*A B C D E B F E*" and cache size equals to 4. What will happen?



In 1–4 steps the A,B,C,D will be set.

5. We need to add E to the cache, but the cache size if four, therefore we must evict something. The most unused item is A, so cut it out.

6. Get B from the cache (or set, it doesn't matter). Replace B to the head of our linked list as last used element.

7. Add F. Now C is the most unused item, cut it and add F to the head.

8. Get E. Replace E to the head of the list and connect D and B between each other.

Not so difficult, is it?

Create get() and set() methods:

```java
@Override
public void set(K key, V value) {

    // try to find node with value in the cache
    Node<K,V> node = cache.get(key);

    if (node == null) {
        // create new one at the head
        node = new Node<>(head, key, value, null);
        cache.put(key, node);
    } else {
        node.item = value;
        connectNeighborNodes(node);
        ifThisNodeIsTail(node);
    }
    setHead(node);

    // remove tail
    if (tail != null && cache.size() > capacity) {
        cache.remove(tail.key);
        tail = tail.next;
        if (tail != null)
            tail.prev = null;
    } else if (tail == null) {
        tail = node;
    }

}

@Override
public V get(K key) {

    Node<K,V> node = cache.get(key);
    if (node == null) {
        // cache miss
        return null;
    }

    connectNeighborNodes(node);
    ifThisNodeIsTail(node);
    setHead(node);

    return node.item;
}

```

```
46  private void connectNeighborNodes(Node<K, V> node) {
47      // connect neighbor nodes between themselves
48      if (node.prev != null)
49          node.prev.next = node.next;
50      if (node.next != null)
51          node.next.prev = node.prev;
52  }
53
54  private void ifThisNodeIsTail(Node<K, V> node) {
55      // change the tail
56      if (tail == node)
57          tail = node.next;
58  }
59
60  private void setHead(Node<K, V> node) {
61      // set accessed element as the head
62      node.prev = head;
63      if (head != null)
64          head.next = node;
65      head = node;
66      node.next = null;
67  }
```

CacheEvictLRUGetSet.java hosted with ❤️ by **GitHub** **view raw**

Note that catching of all situations and errors is not our purpose. This code just shows the logic of the cache with LRU eviction policy.

Create a test method and show the cache content with a head and a tail in each step (every node shows like *{previous node item <← current node item →> next node item}*):

```
0 – – – – – – –

head=null
```

{}

tail=null

1 – – – – – – –

head= null <← A →> null

{1= null <← A →> null}

tail= null <← A →> null

2 – – – – – – –

head= A <← B →> null

{1= null <← A →> B, 2= A <← B →> null}

tail= null <← A →> B

3 – – – – – – –

head= B <← C →> null

{1= null <← A →> B, 2= A <← B →> C, 3= B <← C →> null}

tail= null <← A →> B

4 – – – – – – –

head= C <← D →> null

{1= null <← A →> B, 2= A <← B →> C, 3= B <← C →> D, 4= C <← D →> null}

tail= null <← A →> B

5 – – – – – – –

head= D <← E →> null

{2= null <← B →> C, 3= B <← C →> D, 4= C <← D →> E, 5= D <← E →> null}

tail= null <← B →> C

6 – – – – – – –

head= E <← B →> null

```
{2= E <← B →> null, 3= null <← C →> D, 4= C <← D →> E, 5= D <← E →> B}
tail= null <← C →> D
7 – – – – – – –
head= B <← F →> null
{2= E <← B →> F, 4= null <← D →> E, 5= D <← E →> B, 6= B <← F →> null}
tail= null <← D →> E
8 – – – – – – –
head= F <← E →> null
{2= D <← B →> F, 4= null <← D →> B, 5= F <← E →> null, 6= B <← F →> E}
tail= null <← D →> B
```

As you can see every node and item is in their place.

Actually, Java has an implementation of this cache variant in *LinkedHashMap*.

What is the problem with LRU?

With the addition of many new elements, we can replace some items, which were frequently used.

2. Least Frequently Used

Apache did the LFU cache implementation in one of the ways. Let's consider it!

They have a node:

```java
private static class CacheNode<Key, Value> {

    public final Key k;
    public Value v;
    public int frequency;

    public CacheNode(Key k, Value v, int frequency) {
        this.k = k;
        this.v = v;
        this.frequency = frequency;
    }
}
```

CacheEvictLFUNode.java hosted with ❤️ by **GitHub** **view raw**

and a cache with some parameters:

```java
private final Map<Key, CacheNode<Key, Value>> cache;
private final LinkedHashSet[] frequencyList;
private int lowestFrequency;
private int maxFrequency;
private final int maxCacheSize;
```

CacheEvictLFUVars.java hosted with ❤️ by **GitHub** **view raw**

The cache is a hashMap with some object as key and Node as value. The Node stores key&value with the uses frequency.

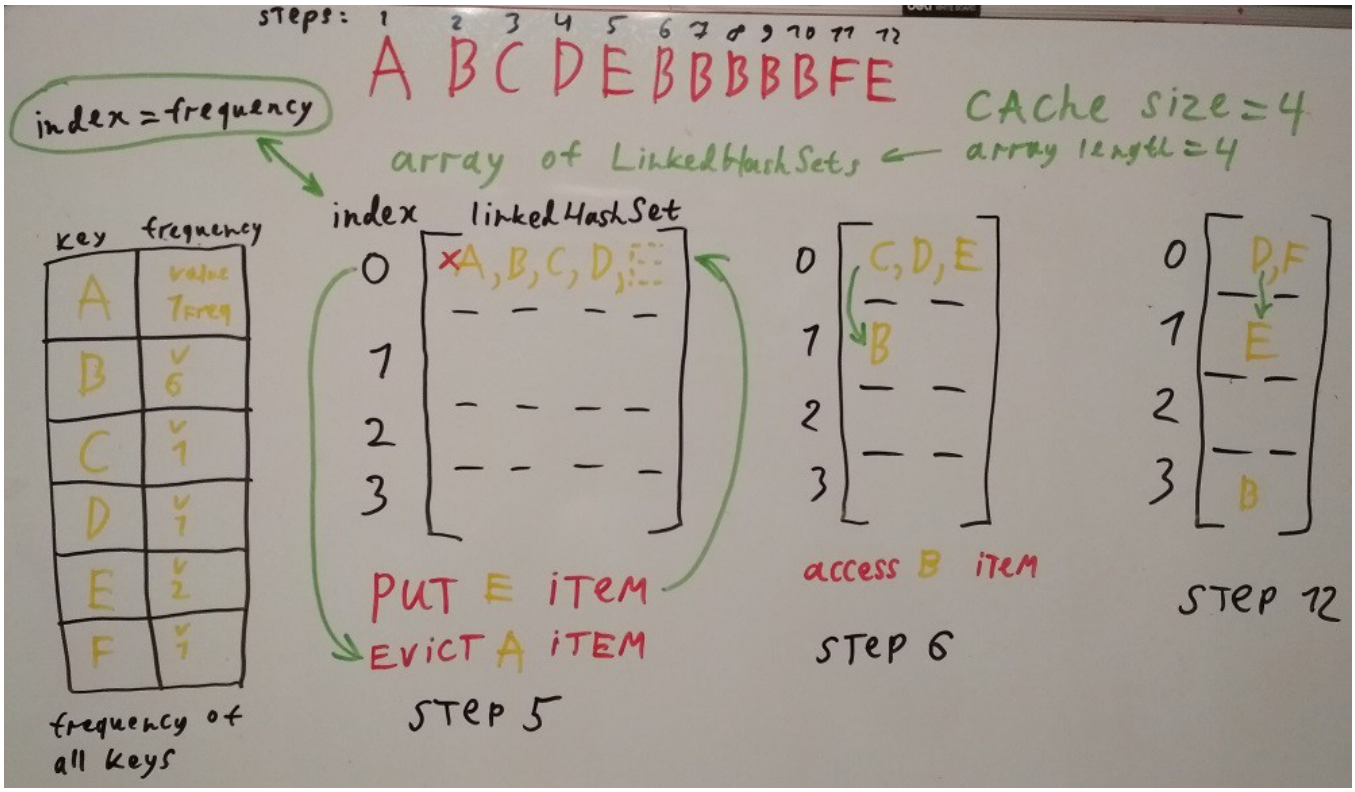
Frequency list is an array of linked sets of the same Nodes as in the cache.

```java
this.cache = new HashMap<Key, CacheNode<Key, Value>>(maxCacheSize);
this.frequencyList = new LinkedHashSet[maxCacheSize];

for (int i = 0; i <= maxFrequency; i++) {
    frequencyList[i] = new LinkedHashSet<CacheNode<Key, Value>>();
}
```

CacheEvictLFUConstruct.java hosted with ❤️ by **GitHub** **view raw**

How does it work and why do they need *lowestFrequency* and *maxFrequency*? Let's do a test with the sequence of "*A B C D E B B B B B F E*" and look at the result:



1–4 steps: put A,B,C,D in the set at 0 position (each was just used once).

Step 5: put E and evict A (E used once, A is the least recently used item).

Step 6: B was accessed twice -> move it to the next set with index 1.

7–10 steps: B was accessed six times and moved to the set at index 5. But cache size is 4, so B was moved to the set at 3 position in the array.

Step 11: put F.

Step 12: E was accessed twice -> move it to the next set with index 1.

The main thing here is: the index of the *frequencyList* array is a frequency of the items inside the set in this position.

```
0 – – – – – –
```

{}

[[], [], [], []]

1 – – – – – –

{1=A=0}

[[A=0], [], [], []]

2 – – – – – –

{1=A=0, 2=B=0}

[[A=0, B=0], [], [], []]

3 – – – – – –

{1=A=0, 2=B=0, 3=C=0}

[[A=0, B=0, C=0], [], [], []]

4 – – – – – –

{1=A=0, 2=B=0, 3=C=0, 4=D=0}

[[A=0, B=0, C=0, D=0], [], [], []]

5 – – – – – –

{2=B=0, 3=C=0, 4=D=0, 5=E=0}

[[B=0, C=0, D=0, E=0], [], [], []]

6,7,8,9,10 – – – – – –

{2=B=3, 3=C=0, 4=D=0, 5=E=0}

[[C=0, D=0, E=0], [], [], [B=3]]

11 – – – – – –

{2=B=3, 4=D=0, 5=E=0, 6=F=0}

[[D=0, E=0, F=0], [], [], [B=3]]

12 – – – – – –

```
{2=B=3, 4=D=0, 5=E=1, 6=F=0}

[[D=0, F=0], [E=1], [], [B=3]]
```

1. The frequency of any item cannot be more than the cache size.

2. All elements in the cache are stored in the linked sets by their frequency: if the item is just put, it will be stored at the end of linked set at zero position in the array. If it is accessed once — it'll be moved to the end of the linked set at position one in the array, and so on. When item is accessed 4 or more times (in our case 4 is a maximum cache size = *maxFrequency*), it'll be put to the latest linked set in the array and the count of uses of this item won't be higher than 3 (positions in the array starts from zero).

3. When we need to evict some element, we take the linked hash set from the array by index = *lowestFrequency*, then remove the first (least) item from it (an *iterator* will help us).

That's it. **We have O(1) time complexity** for get and put operations.

What's the problem?

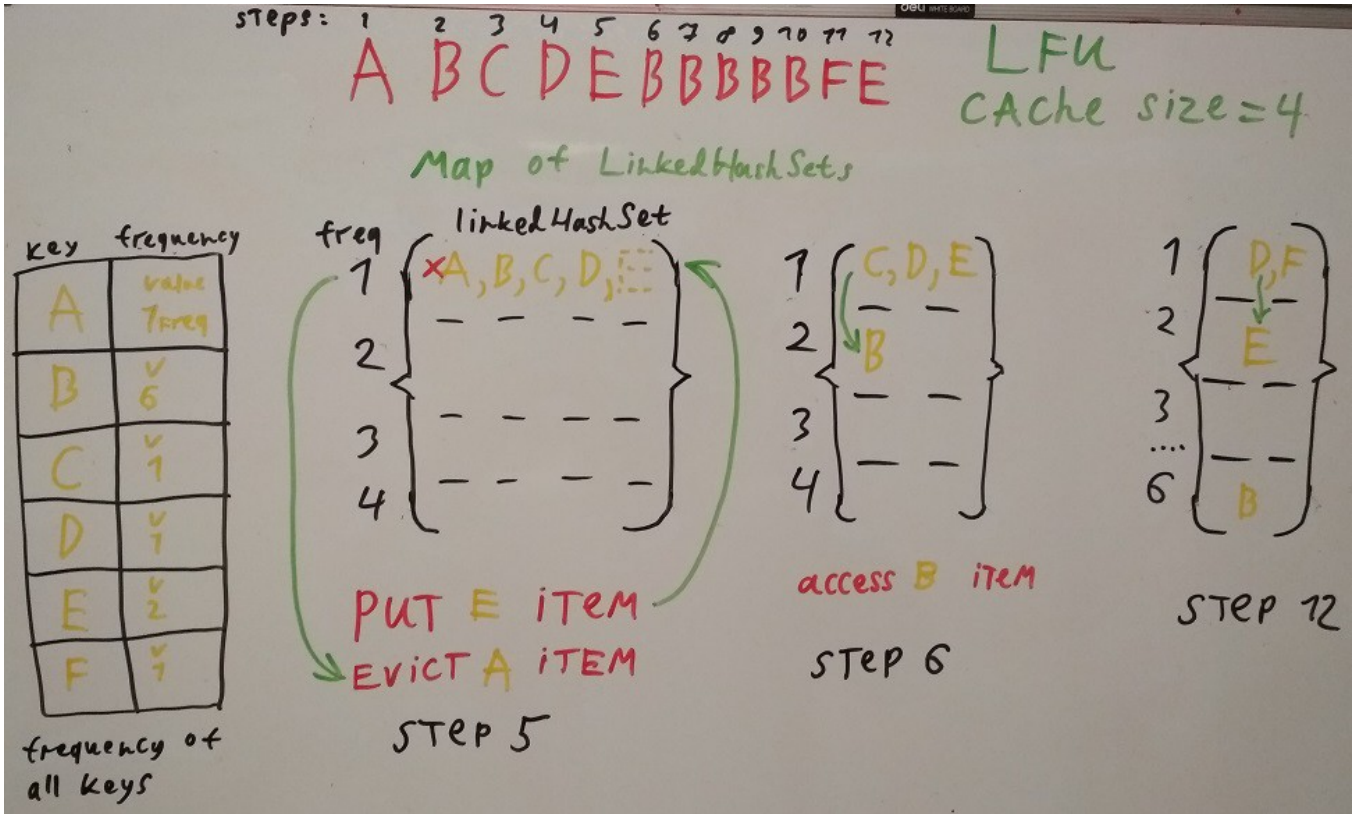
We don't take into account how frequently each item has been accessed during the whole period. Is it possible? Of course, everything is possible, even the flying unicorns!

Use the separate map to store the frequency and one more for the linked sets (instead of using an array):

```
private final HashMap<K, Long> frequency;

private final HashMap<Long, LinkedHashSet<K>> frequencySets;
```

If we don't remove evicting items from this frequency map, we will get actual whole period count of using any element. We don't need a *maxFrequency* variable in this case, 'cause we have a map of sets instead of an array, and we can store any count. If the frequency of some item reaches a maximum of long-type, we need to do something. For example, divide all the frequencies by two, or just drop this item.



7–10 steps: B has been accessed six times and moved to the set at hash of 6. We removed the restriction of the maximum frequency.

You can look closer at this implementation in the class *LFUCacheThreeMaps,* the link to the whole code at github will be presented at the end of the article.

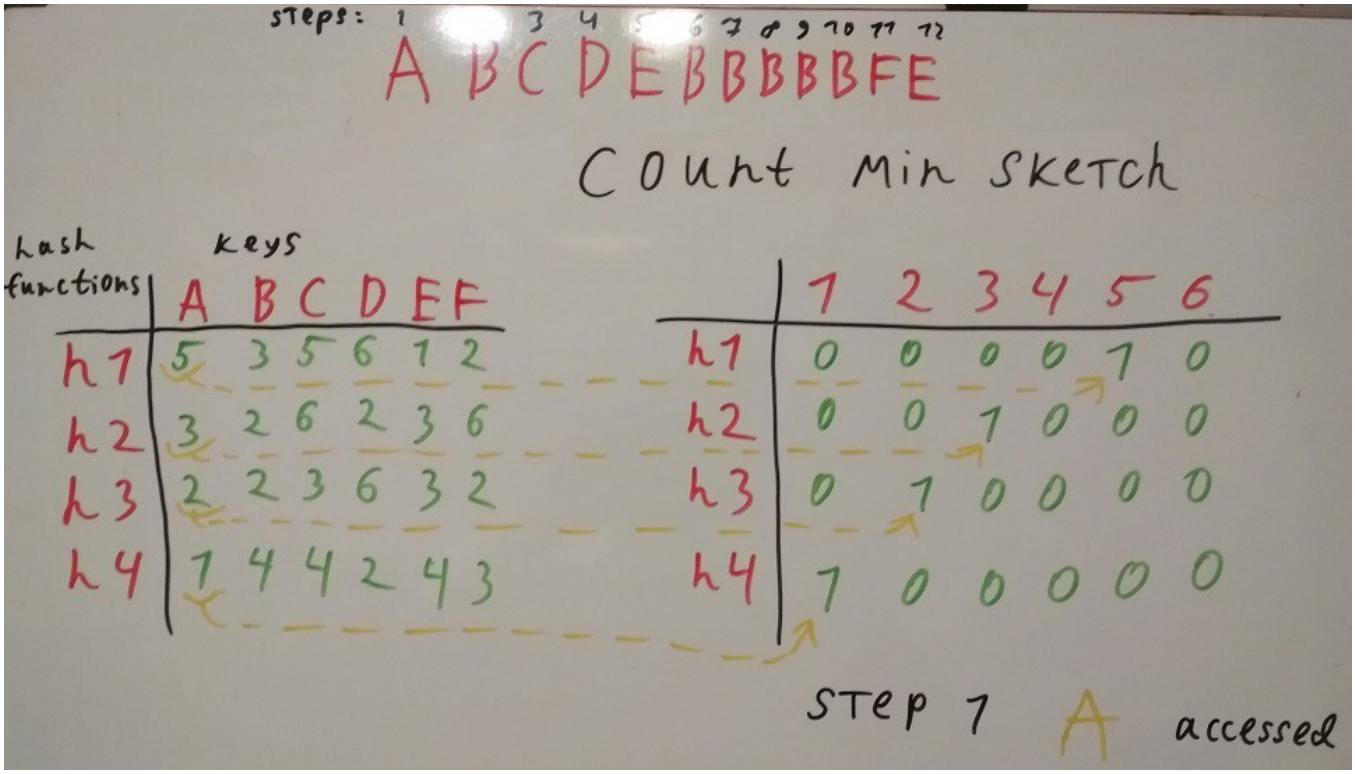
But.. what's the problem?? Of course in the memory! It's not rubber! All keys will be stored forever.. And what can we do?

3. Count Min Sketch

Let's take some sequence again "*A B C D E B B B B B F E*". How often is every letter repeated?

A — 1, B — 6, C — 1, D — 1, E — 2, F — 1. How could we store it without storing the keys? With a hashing.

Suppose we have a four different hash functions with the same distribution of six samples.

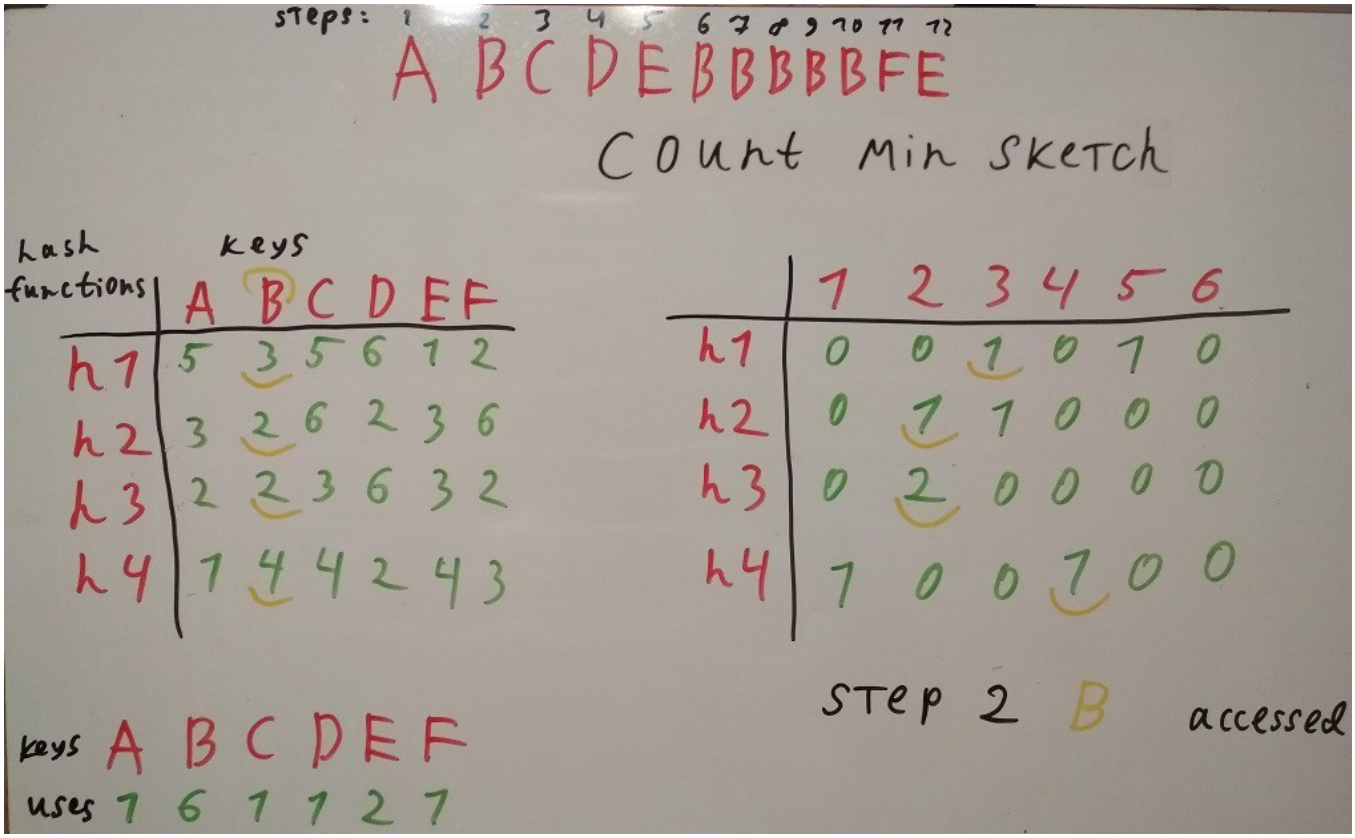


A is accessed (put or get — it doesn't matter).

Hash function 1 on key A gives the result 5. Take current value on position 5 (in the right table) and increase it by 1.

Hash function 2 on key A gives the result 3. Take current value on position 3 (in the right table) and increase it by 1.

Hash 3 gives us 2, and hash 4 returns 1. Increase it.

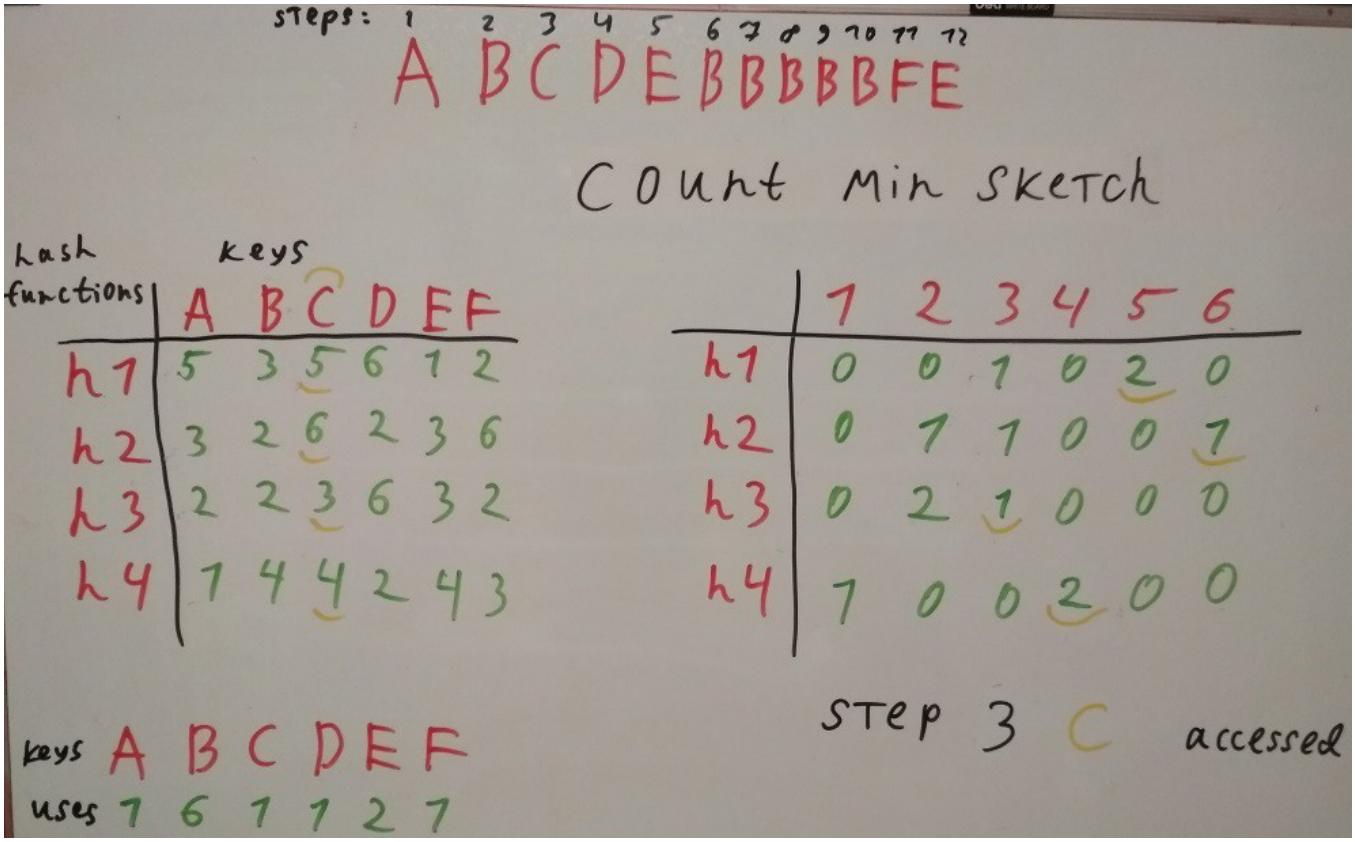


B is accessed.

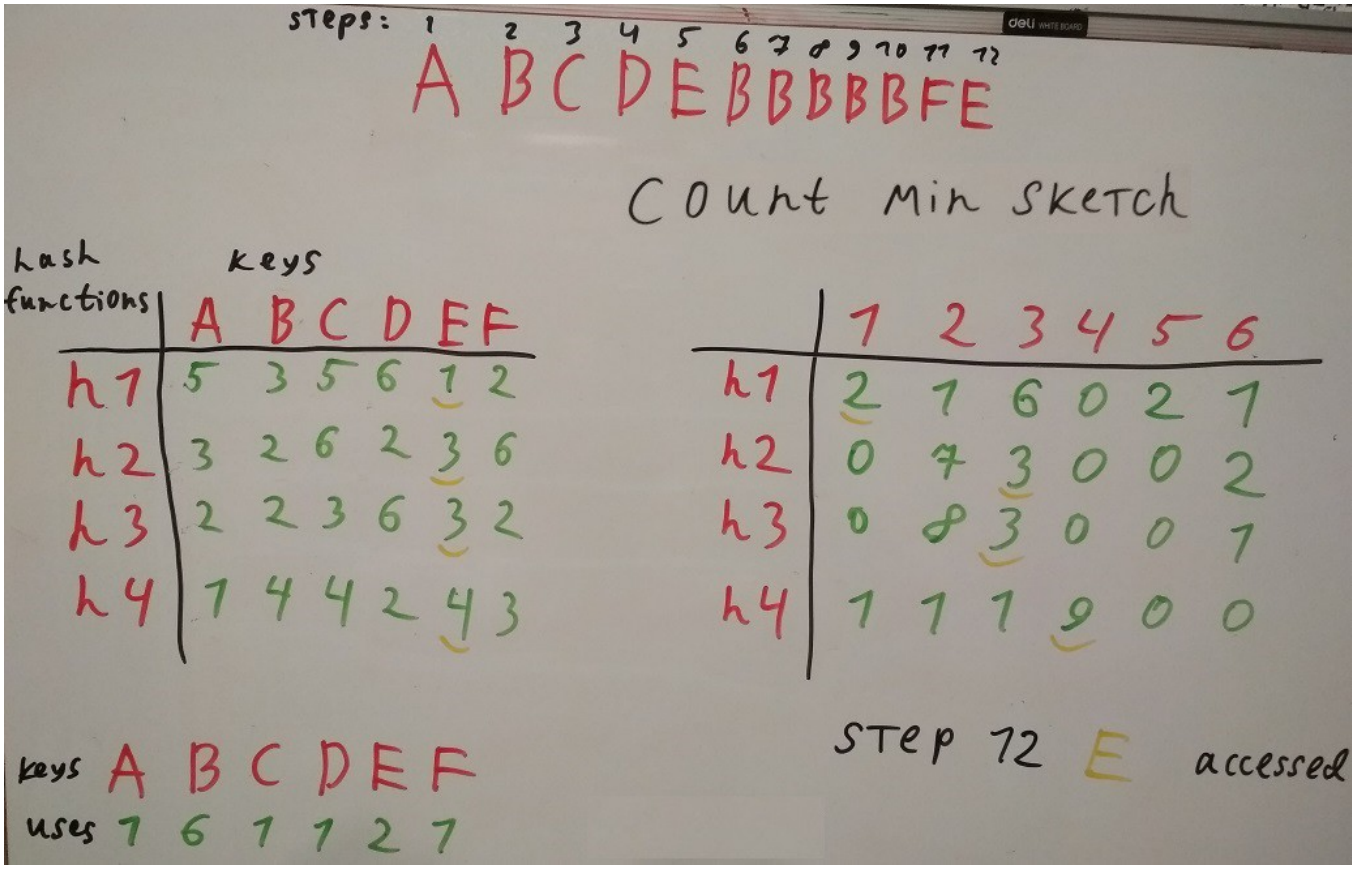
Hash function 1 on key B gives the result 3. Take current value on position 3 (in the right table) and increase it by 1.

Do it for each result of hash functions on B.

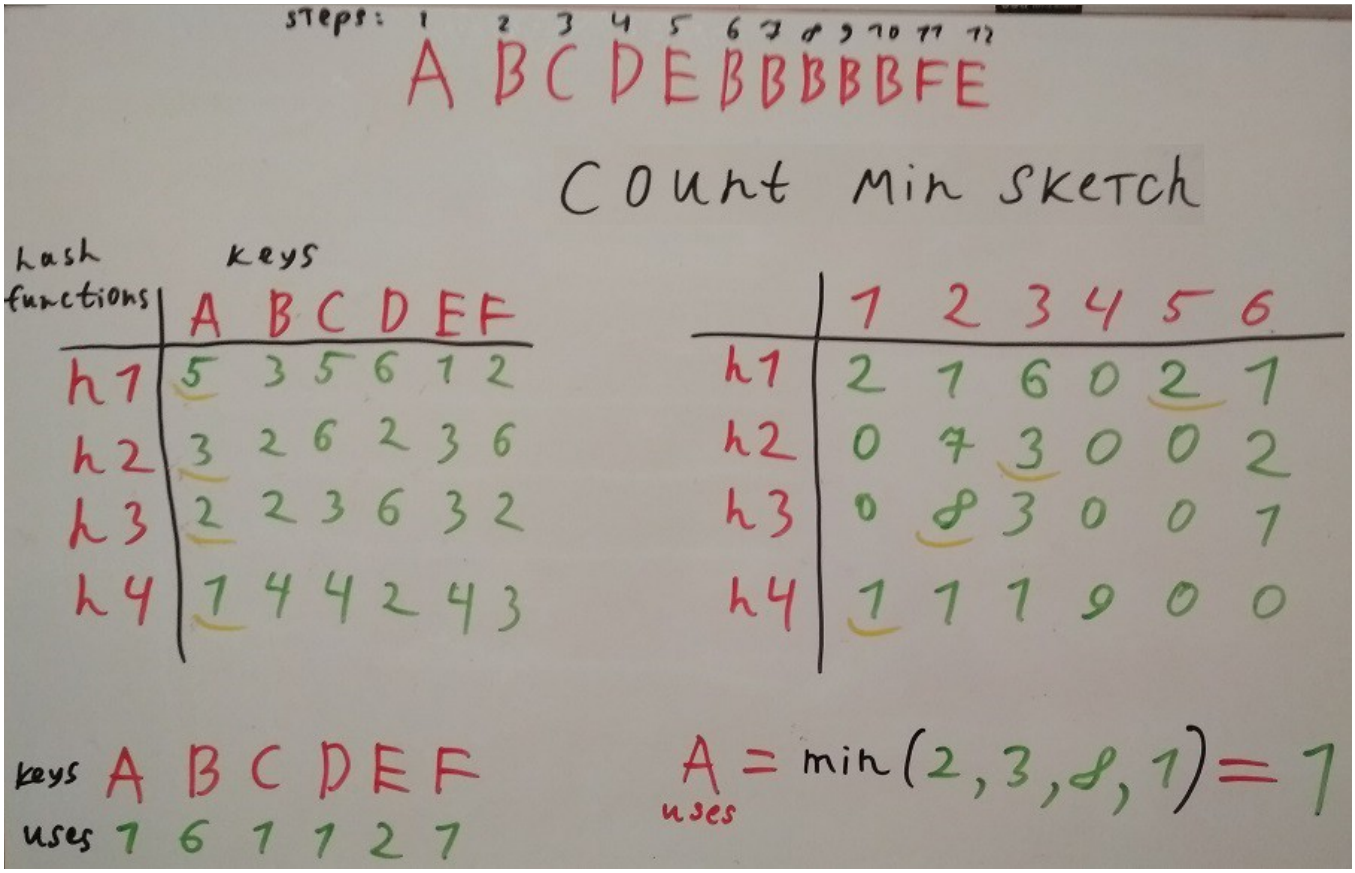
Note that hash3 on key B has a result of 2, like on the key A. It means that after increasing the number in the right table, we will get the 2 uses/accesses.



Do it all for the key C. And so on for every next accessed key.

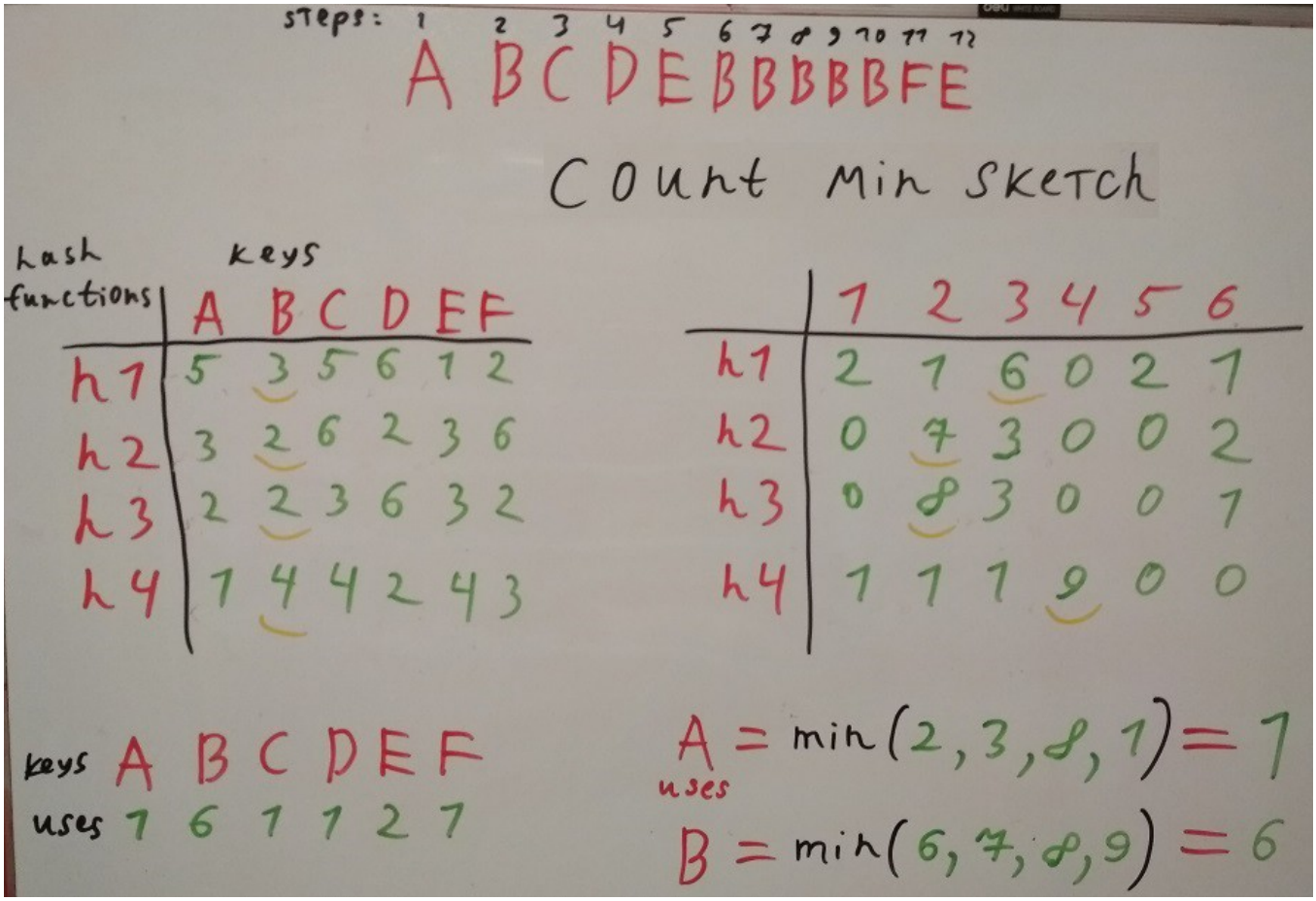


Okay, and how could it help us determine the count of every key use?

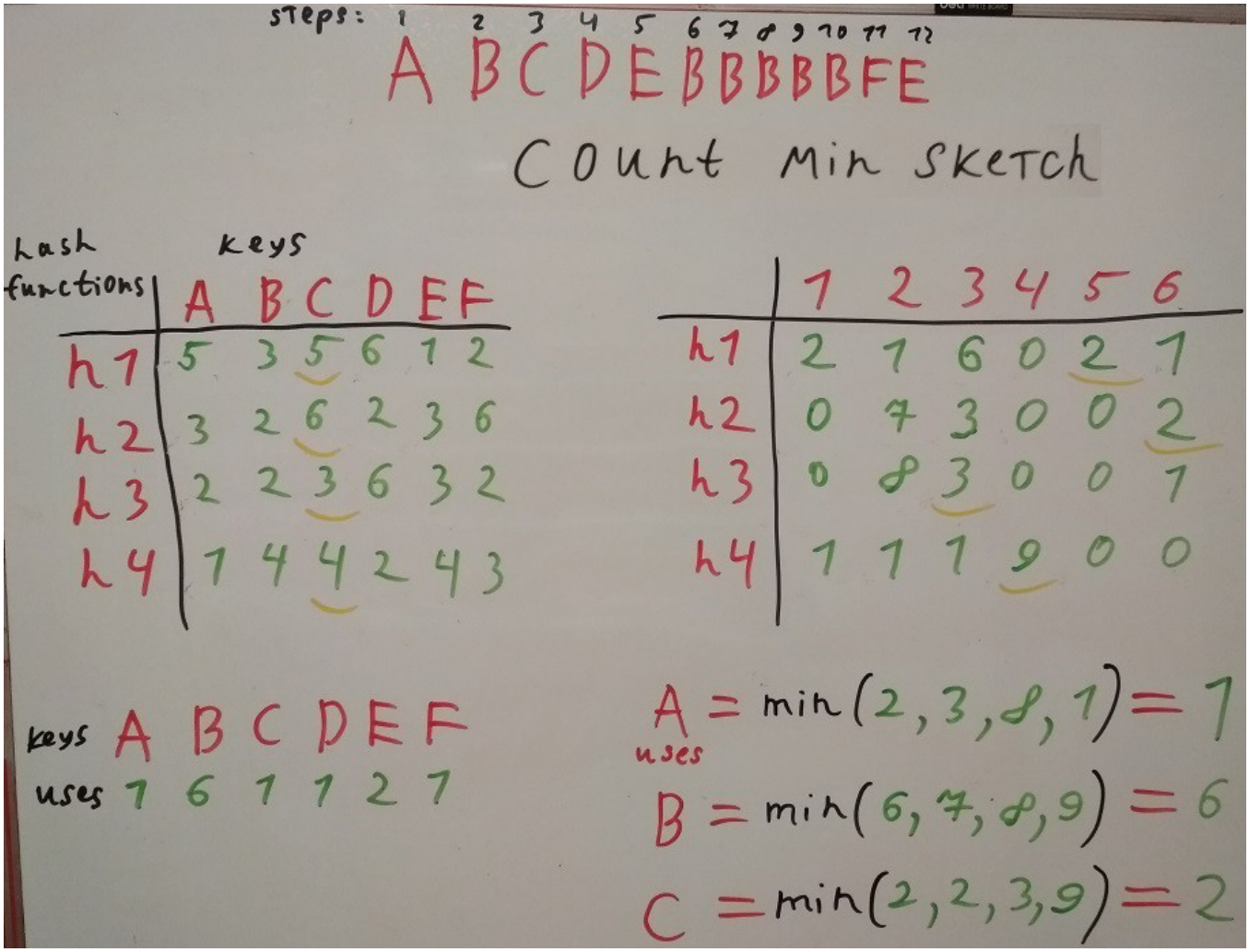


It's simple: take all counted values for the key and determine the minimum of them.

For the key A: hash1 has a result 5. Look at the right table at position hash1–5 and find there the counted value 2. Do it for every hash function. Take the minimum of the results — it'll be the 1. So, A was accessed once.



B has been accessed six times. It's true.



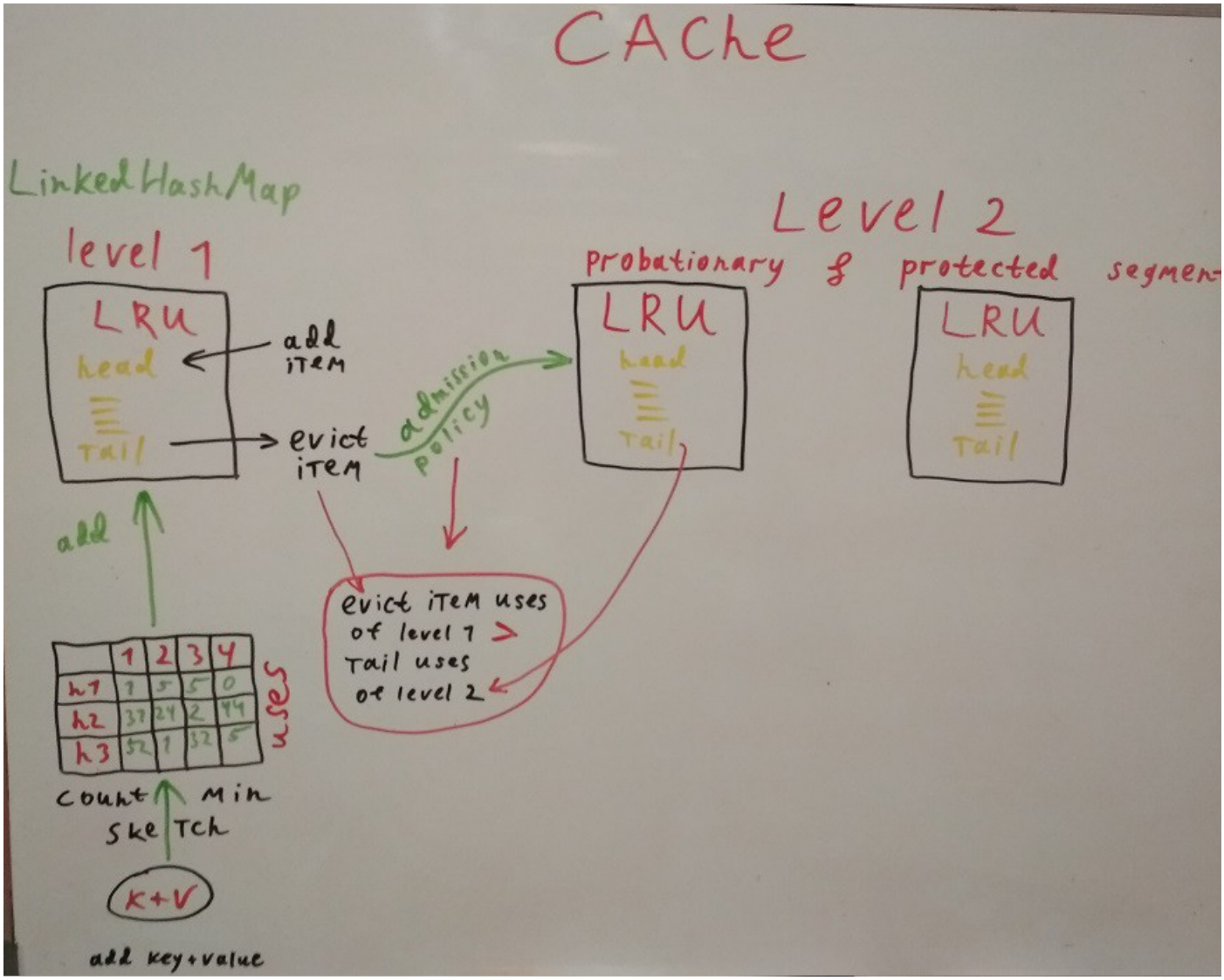
C has been allegedly accessed 2 times. But we know that C was accessed just once. It's not a mistake. With Count Min Sketch you will get the actual number of the item uses or more. But no less.

If you want to increase the precision, you can add more hash functions.

What if we reach the maximum of using variable type in the table of uses? Just divide all the frequencies by two, or drop this item.

4. True cache with blackjack and levels!

How could we use all our new knowledge? Let's build a true cache with blackjack and levels!



What's going on here? We have two levels of cache. First level — is simple LRU cache, contains new items. Second level — is SLRU cache with the two LRU segments. But we have an **admissi policy** for this level: any *new item should have a larger access frequency than the item, which will evicted*.

1. We want to put a key and a value.

2. Drive a key through Count Min Sketch table to count the accesses frequency.

3. Put the item to the head of level 1 cache.

4. Evict the tail item.

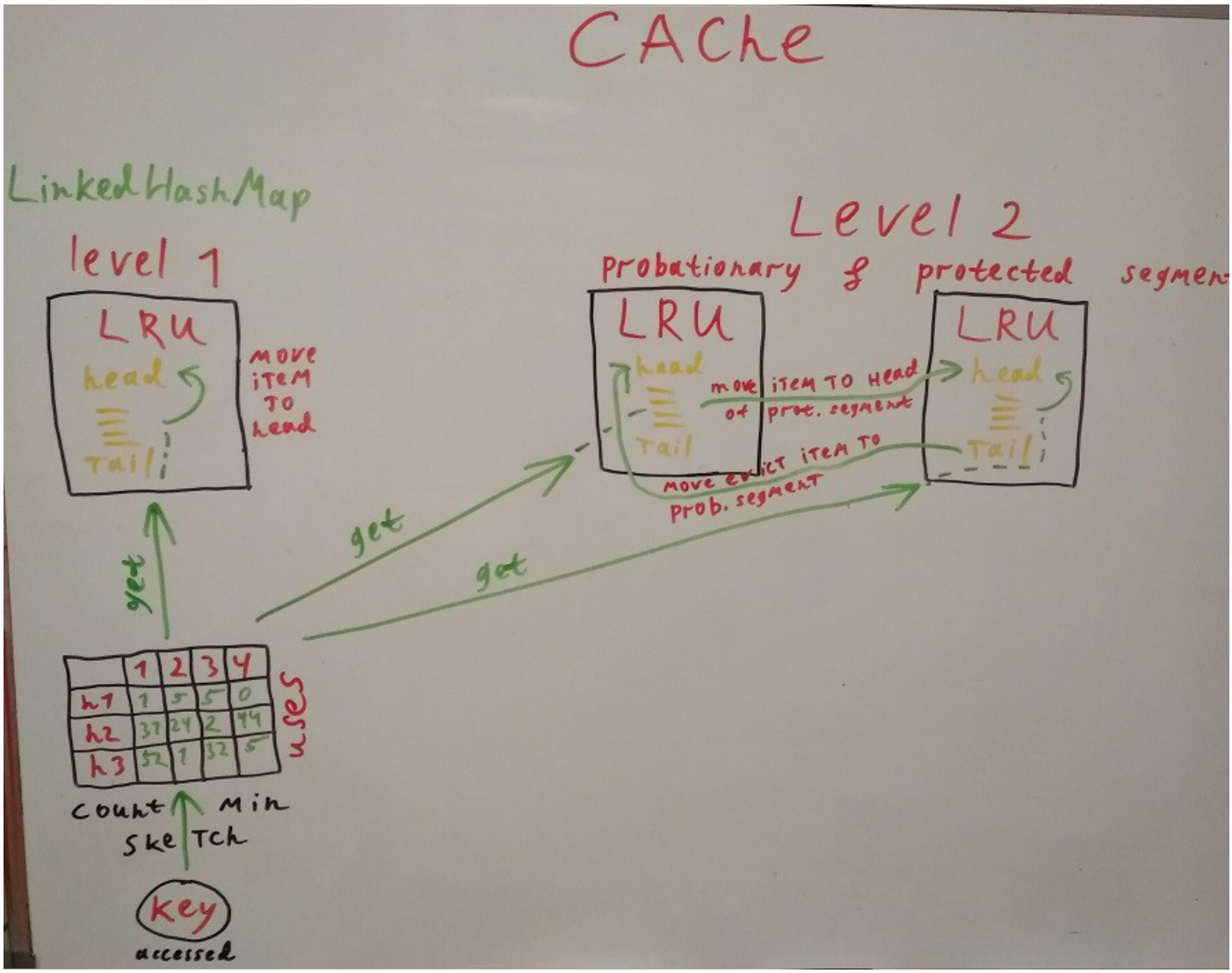
5. This evicted item could be moved to the level 2 cache:

5.1 get the evicted (from the level 1 cache) item frequency (look at the Count Min Sketch table

5.2 if we add a new element to the level 2 cache, then some element will be evicted from there. So, get the tail (from the probationary segment in the level 2 cache) item frequency.

5.3 Compare them.

5.4 If the frequency of the new element is larger than the tail item frequency, replace it. But to t head, of course, it's still the LRU cache after all.



1. We want to access some key.

2. Drive it through Count Min Sketch table to count the accesses frequency.

3. Check it in the level 1 cache and move it to the head if exists.

3.1 if not — check it in the probationary segment in the level 2 cache.

3.2 If it exists, move it to the protected segment.

3.2.1 and move the tail of protected segment back to the head of probationary segment. Just gi it one more chance.

3.3 if not — check it in the protected segment in the level 2 cache and move it to the head if exists.

Congratulations! We created the true cache with two levels and with taking into account the access frequencies of the items.

Conclusions.

Now you know how to implement the different eviction policies of the cache systems, how to make the admission policy and how to take into account the access frequencies of all elements ever requested from the cache.

You can look at the full code on my github project.

Feel free to shake hand on Medium or Telegram or Twitter.