LFU Cache

LFUCache (int capacity)
int get (int key)
void put (int key, int value)

Input
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation

```
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is  most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1);   // cache=[1, ], cnt(1)=1
lfu.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1);      // return 1
                 // cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3);   // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
                 // cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4);   // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
                 // cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4);      // return 4
                 // cache=[4,3], cnt(4)=2, cnt(3)=3
```

(2)

Put (1,1)

Put (2,2)

get (1)

Put (3,3)

get (2)

get (3)

put (4,4)

get (1)

get (3)

get (4)

Input

```
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
```

Output

```
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]
```

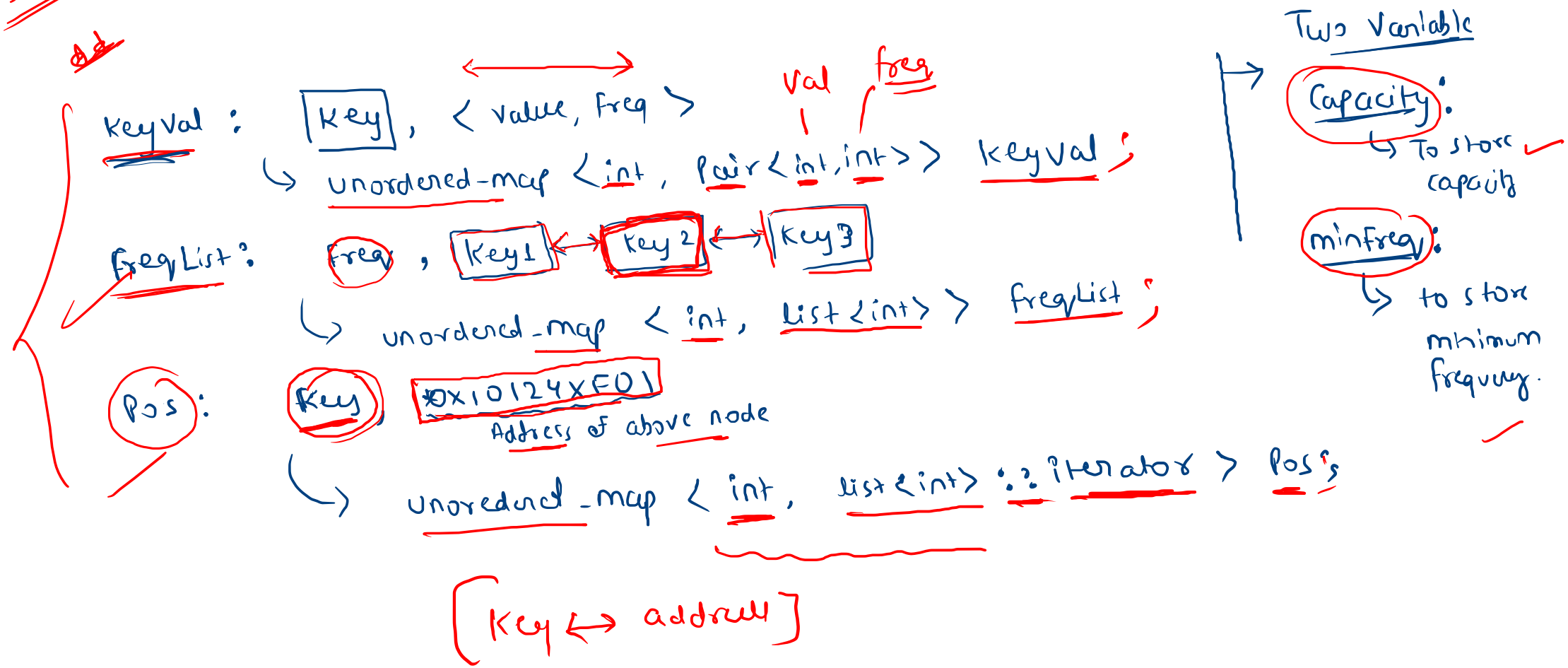
Explanation

```
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is  most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1);   // cache=[1,_], cnt(1)=1
lfu.put(2, 2);   // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1);      // return 1
                 // cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3);   // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
                 // cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4);   // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
                 // cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1);      // return -1 (not found)
lfu.get(3);      // return 3
                 // cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4);      // return 4
                 // cache=[4,3], cnt(4)=2, cnt(3)=3
```

Solution Design

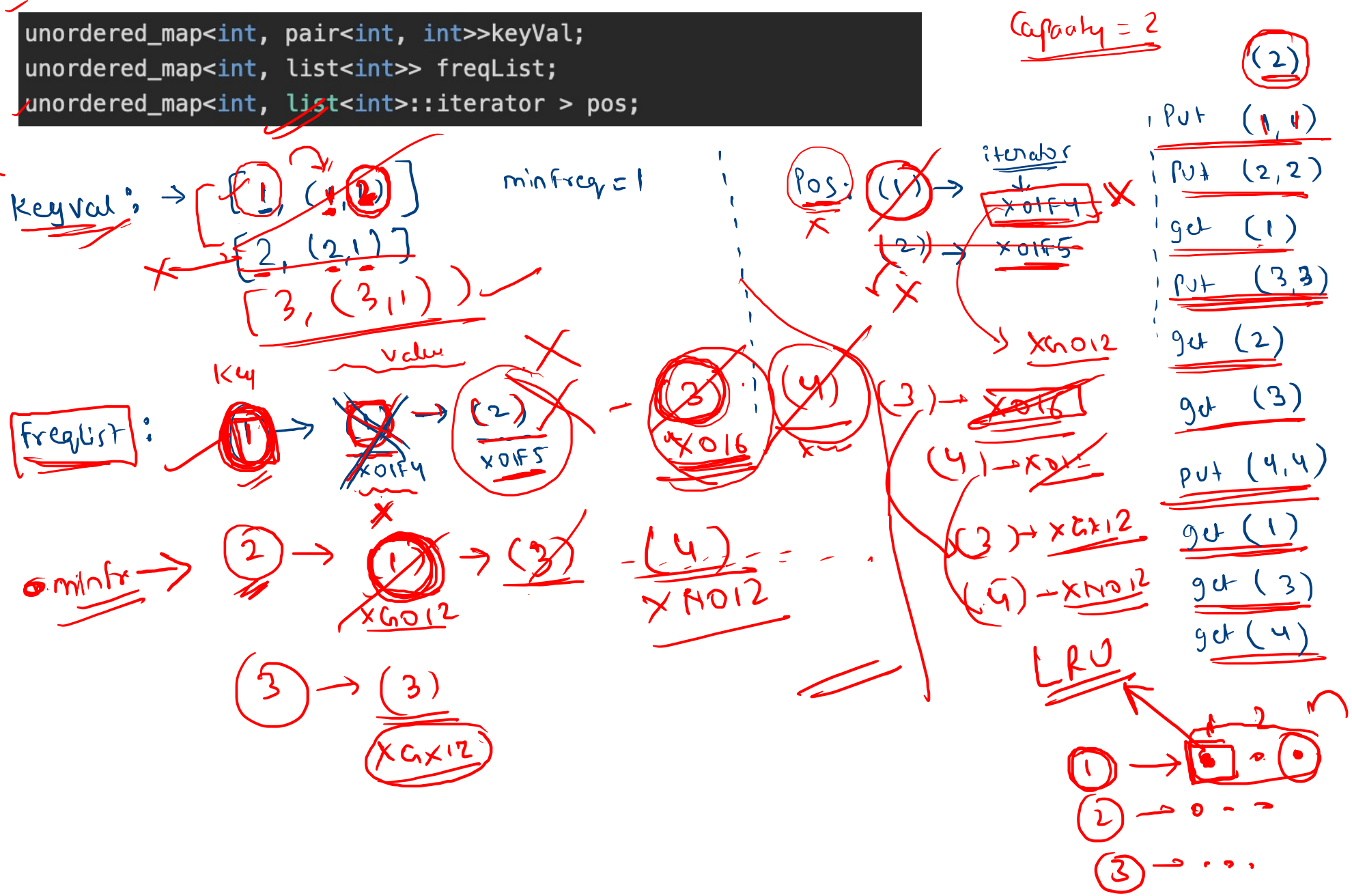
List<int> doubly linked

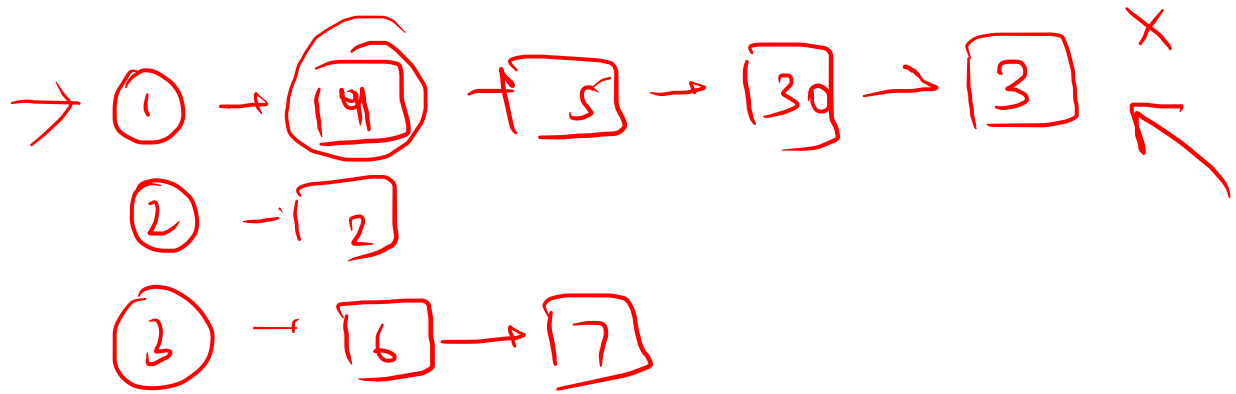
By using 3 Maps

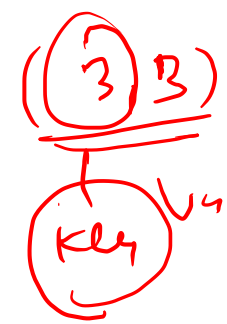


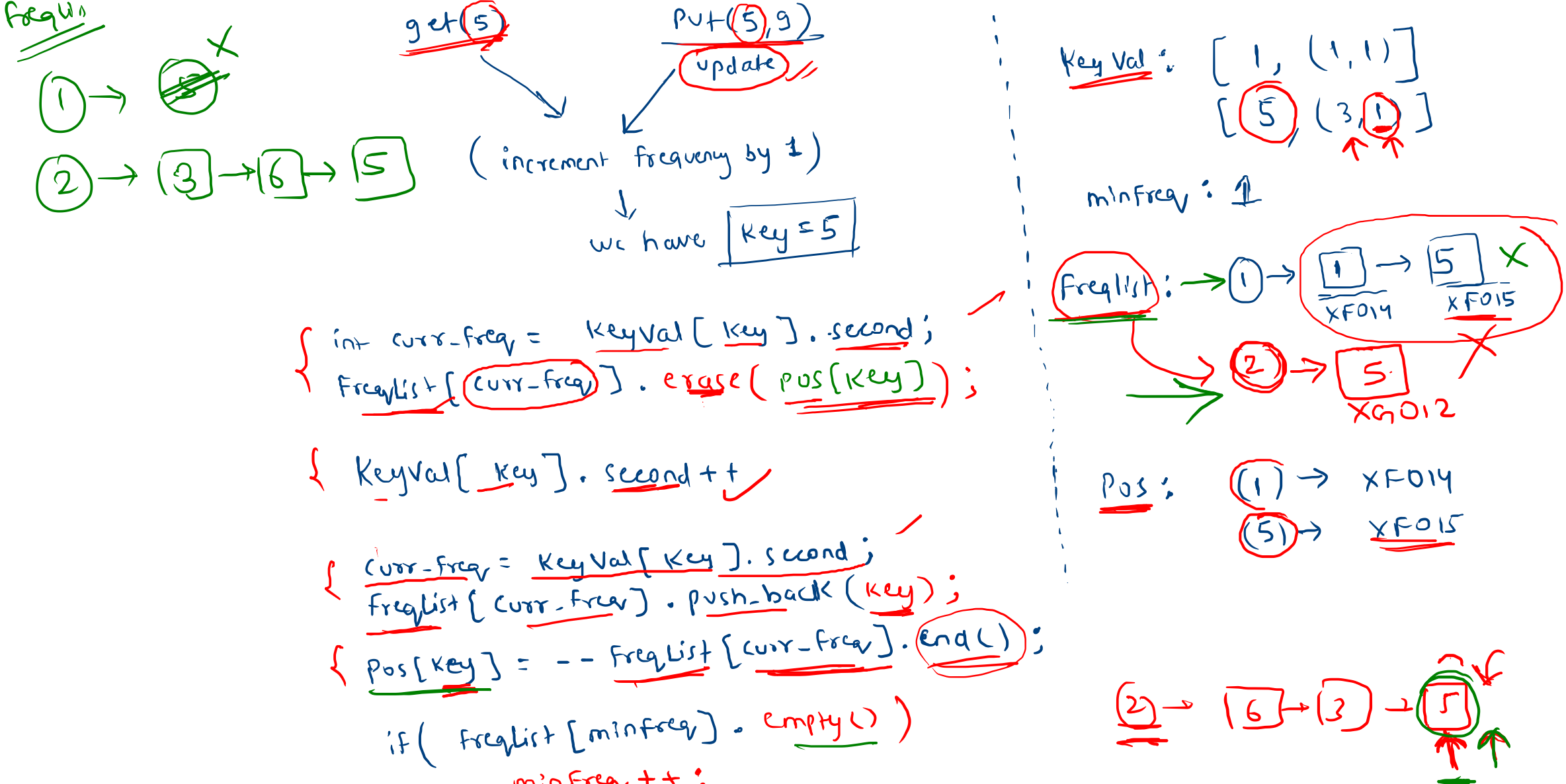
minfreq = 1

```
unordered_map<int, pair<int, int>>keyVal;
unordered_map<int, list<int>> freqList;
unordered_map<int, list<int>::iterator > pos;
```




1
4
5
30
3
2
2
3
6
7


3
3
key



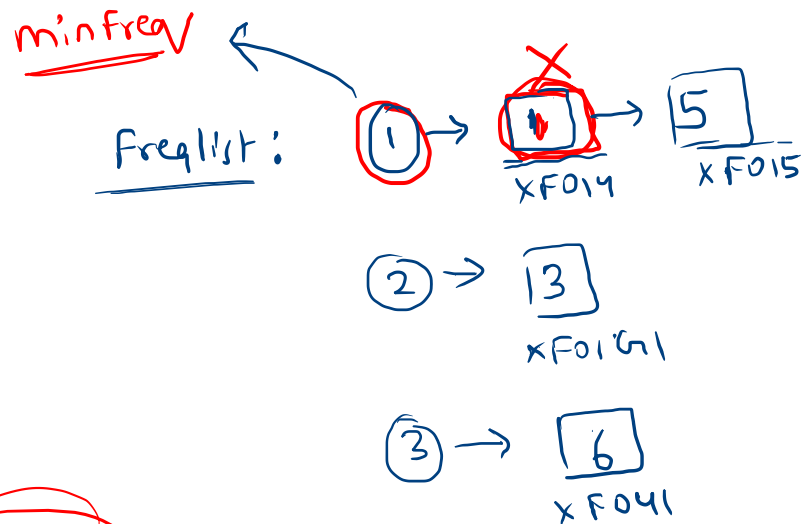

```
{ int curr_freq = KeyVal[key].second;
  Freqlist[curr_freq].erase(pos[key]);

{ KeyVal[key].second++

{ curr_freq = KeyVal[key].second;
  Freqlist[curr_freq].push_back(key);
{ pos[key] = --Freqlist[curr_freq].end();

if( freqlist[minfreq].empty() )
    minFreq++;
```

if capacity is full




```
int delkey = freqlist[minfreq].front();
keyval.erase(delkey)
pos.erase(delkey)
freqlist[minfreq].pop_front();
```

delkey = 1

Insert

```
keyval[key] = {value, 1}
freqlist[1].push_back(key);
pos[key] = --freqlist[1].end();
minfreq = 1
```