**Problem Name:**LFU Cache

**Topics:** Hash Table, Linked list, Design, Doubly linked list

**Companies:** Amazon, Microsoft, Google, Linkedin, Salesforce, Snapchat, Facebook, Bloomberg, Walmart, Arcesium.

**Level: Hard**

**Language:** C++

**Problem Statement:** Design and implement a data structure for a [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the LFUCache class:

* LFUCache(int capacity) Initializes the object with the capacity of the data structure.
* int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
* void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** key would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to 1 (due to the put operation). The **use counter** for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

**Input Format:**

First line of input contains integer value capacity

Second line of input contains integer value n (total no of “put” + “get” operation)

Next n line contains:

1st line is string “put” or “get” representing function to call

If it’s “put” then next line contains two integers key and value

If it’s “get” then next line contains integer value key

Ex:

2

10

put

1 1

put

2 2

get

1

put

3 3

get

2

get

3

put

4 4

get

1

get

3

get

4

**Output Format:** Print output from function get whenever it is called. Ex for above input output would be

1 -1 3 -1 3 4d

**Constraints:**

* 0 <= capacity <= 104
* 0 <= key <= 105
* 0 <= value <= 109
* At most 2 \* 105 calls will be made to get and put.

**Examples:**

**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

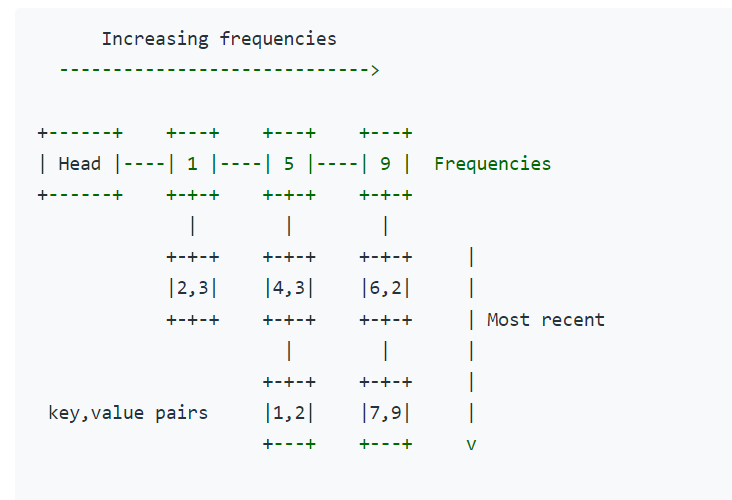
// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

**Approach one Solution:**

**Explanation:**



Similar to bucket sort, we place key,value pairs with the same frequency into the same bucket, within each bucket, the pairs are sorted according to most recent used, i.e., the one that is most recently used (set,get) is at the bottom of each bucket.

**Code:**

#include <bits/stdc++.h>

using namespace std;

class LFUCache

{

 public:

  struct LRUNode

  {

    int freq;

    list<pair<int, int> > vals;

    LRUNode(int f = 0) : freq(f) { }

  };

  typedef list<LRUNode>::iterator iptr;

  typedef list<pair<int, int> >::iterator jptr;

  LFUCache(int capacity)

  {

    capacity\_ = capacity;

  }

  int get(int key)

  {

    int val = -1;

    if (kv\_.find(key) != kv\_.end()) {

      kv\_[key] = promote(key);

      val = kv\_[key].second->second;

    }

    return val;

  }

  void set(int key, int value)

  {

    if (capacity\_ <= 0) return;

    if (kv\_.find(key) == kv\_.end()) {

      if (kv\_.size() == capacity\_) evict();

      kv\_[key] = insert(key, value);

    } else {

      kv\_[key] = promote(key, value);

    }

  }

 private:

  pair<iptr, jptr> promote(int key, int val = -1)

  {

    iptr i; jptr j;

    tie(i, j) = kv\_[key];

    iptr k = next(i);

    if (val < 0) val = j->second;

    int freq = i->freq + 1;

    i->vals.erase(j);

    if (i->vals.empty())

      cache\_.erase(i);

    if (k == cache\_.end() || k->freq != freq)

      i = cache\_.insert(k, LRUNode(freq));

    else i = k;

    j = i->vals.insert(i->vals.end(), {key, val});

    return {i, j};

  }

  void evict()

  {

    iptr i = cache\_.begin();

    jptr j = i->vals.begin();

    kv\_.erase(j->first);

    i->vals.erase(j);

    if (i->vals.empty())

      cache\_.erase(i);

  }

  pair<iptr, jptr> insert(int key, int val)

  {

    iptr i = cache\_.begin();

    if (i == cache\_.end() || i->freq != 1)

      i = cache\_.insert(i, LRUNode(1));

    jptr j = i->vals.insert(i->vals.end(), {key, val});

    return {i, j};

  }

 private:

  list<LRUNode> cache\_;

  int capacity\_;

  unordered\_map<int, pair<iptr, jptr> > kv\_;

};

int main() {

    int n, capacity, key, value, temp;

    string task;

    cin>>capacity;

    cin>>n;

    vector<int> result;

    LFUCache \*obj = new LFUCache(capacity);

    while(n--){

        cin>>task;

        if(task == "put"){

            cin>>key>>value;

            obj->set(key, value);

        }

        else if(task=="get"){

            cin>>key;

            temp = obj->get(key);

            result.push\_back(temp);

        }

    }

    for(int i=0; i<result.size(); i++){

        cout<<result[i]<<" ";

    }

    return 0;

}

**Time Complexity**: O(1)

**Space Complexity:** O(n)

**Optimized Solution:**

**Explanation:**

* The idea is straightforward. We maintain a min-PriorityQueue with the least frequently used element at the top. However, when we access any existing element using get() or set(), its usage frequency should be increased by one, which forces us to change its position in the priority queue (re-heapify). Unfortunately, native STL PriorityQueue does not support this operation. Therefore, we need to create our own priority queue and modify it. I found the so-called Index Priority Queue is suitable for this. It maintains not only a priority queue but also a hash map, "indexMap", that maps the key of an element to its position (index) in the priority queue. Now, we can quickly access any element in the priority queue and re-heapify the PriorityQueue, when the usage frequency of an element changes.
* Another tricky point is that when we need to evict an element, but multiple elements have the same (minimum) usage frequency, we need to evict the least recently used (the oldest) element. To handle this, I maintain a time-stamp variable for each element in the LFU Cache, which indicates the latest time stamp when we access it. Therefore, we maintain the following invariant in the priority queue: **When two elements have the same usage frequency, the least recently used one will always be closer to root node**. When there are multiple least frequently used elements, we always retrieve the one closer to the root.

**Code:**

#include <bits/stdc++.h>

using namespace std;

class LFUCache {

public:

    struct Node {

        int key; // key of the element.

        int val; // value of the ement.

        int fre; // usage frequency

        int timeStamp; // the latest time stamp when this element is accessed.

        Node(): key(-1), val(-1), timeStamp(-1), fre(0) {}

        Node(int k, int v, int ts): key(k), val(v), timeStamp(ts), fre(1) {}

    };

    LFUCache(int capacity) {

        Cap = capacity;

        Node\* dummy = new Node();

        pq.push\_back(dummy); // The pq start from pq[1].

        ts = 0;

    }

    int get(int key) {

        if(!mp.count(key)) return -1;

        int index = mp[key];

        int val = pq[index]->val;

    pq[index]->fre++;

    pq[index]->timeStamp = ++ts;

        sink(index);

        return val;

    }

    void set(int key, int value) {

        if(Cap <= 0) return;

    if(mp.count(key)) {

       int index = mp[key];

       pq[index]->val = value;

       get(key);

    }

    else {

        if(pq.size() - 1 == Cap) {

            int oldKey = pq[1]->key;

        mp.erase(oldKey);

        Node\* newnode = new Node(key, value, ++ts);

        pq[1] = newnode;

        mp[key] = 1;

        sink(1);

        }

        else {

            Node\* newnode = new Node(key, value, ++ts);

        pq.push\_back(newnode);

        mp[key] = pq.size() - 1;

        swim(pq.size() - 1);

        }

    }

    }

private:

    vector<Node\*> pq; // A priority queue, with the least usage frequency and least recently used element at the top.

    unordered\_map<int, int> mp; // A mapping from the key of the element to its index in the priority queue.

    int Cap; // Capcity of the cache

    int ts; // time-stamp: indicate the time stamp of the latest operation of an element. According to the requirement of LFU cache, when we need to evict an element from the cache, but there are multiple elements with the same minimum frequency, then the least recently used element should be evicted.

    /\*

     \* Recursively sink a node in priority queue. A node will be sinked, when its frequency is larger than any of its

     \* children nodes, or the node has the same frequency with a child, but it is recently updated.

     \*/

    void sink(int index) {

        int left = 2 \* index, right = 2 \* index + 1, target = index;

        if(left < pq.size() && pq[left]->fre <= pq[target]->fre) // If the left child has the same frequency, we probably need to swap the parent node and the child node, because the parent node is recently accessed, and the left child node was accessed at an older time stamp.

               target = left;

            if(right < pq.size()) {

                if(pq[right]->fre < pq[target]->fre || (pq[right]->fre == pq[target]->fre && pq[right]->timeStamp < pq[target]->timeStamp)) // If right child has the same frequency and an older time stamp, we must swap it.

                     target = right;

        }

        if(target != index) {

            myswap(target, index);

                sink(target);

        }

    }

    /\*a

     \* Recursively swim a node in priority queue. A node will be swimmed, when its frequency is less than its

     \* parent node. If the node has the same frequency with its parent, it is not needed to be swimmed, because

     \* it is recently accessed.

     \*/

    void swim(int index) {

        int par = index / 2;

        while(par > 0 && pq[par]->fre > pq[index]->fre) {

            myswap(par, index);

        index = par;

        par /= 2;

        }

    }

    void myswap(int id1, int id2) {

        swap(pq[id1], pq[id2]);

        mp[pq[id1]->key] = id1;

        mp[pq[id2]->key] = id2;

    }

};

int main() {

    int n, capacity, key, value, temp;

    string task;

    cin>>capacity;

    cin>>n;

    vector<int> result;

    LFUCache \*obj = new LFUCache(capacity);

    while(n--){

        cin>>task;

        if(task == "put"){

            cin>>key>>value;

            obj->set(key, value);

        }

        else if(task=="get"){

            cin>>key;

            temp = obj->get(key);

            result.push\_back(temp);

        }

    }

    for(int i=0; i<result.size(); i++){

        cout<<result[i]<<" ";

    }

    return 0;

}

**Time Complexity**: O(1)

**Space Complexity:** O(n)