# LeetCode Training Day 20 Design

Today we will discuss the general data structure design problems. Normally the data structure is defined in the name space of collection. Specially in C++, it is defined in STL. First Let’s review the common data structures we know.

|  |  |
| --- | --- |
| Data Structure | Description |
| HashTable | Access by key with O(1), key is normally primitive and value can be any thing, including a pointer to a class or a collection.  Access, and remove is in O(1) |
| HashSet | Sort distinct value in a collection, does not allow duplicate. If you need to store duplicate, use hashtable with count.  Access and remove is O (LOG(N)) |
| TreeMap (SortedDictionary) | Sorted collection with access to key as O(LOG(N)). You can access or remove by key. The internal implementation is sorted tree, for example, it can be a red-black tree. |
| Array | A list with fixed size, can not grow or shrink. e.g. int arr[100]; |
| ArrayList | A list with dynamic size, can grow at end or can shrink. However, remove a element in the middle will cause other elements to move, therefore remove cost it is O(N). |
| LinkedList | A double linked list, elements can be pushed and popped from both ends. However, it normally does not allow to access the element in random position.  A list can be served as Deque. |
| Queue | First in first out, you can push an element to the end, and pop an element from front. You can only access the item in the front.  It is not possible to access an element at random position. |
| Stack | Last in first out, you can push an element to the end, access the element to the end and pop up an element from the end. |
| Deque | Push and Pop the elements from both ends, access the elements at both ends. |
| String | Each element is a character, you can compare, or extract substring. |
| Tuple | A mix of multiple elements with different types. It will avoid you to construct your own class.  In C++, you may use pair (only bundle two), in Java you may have your own class. |

The following tables discuss the common operations in collection.

|  |  |
| --- | --- |
| Operation | Description |
| Iteration | Iteration every element in the collection. From the first one to the last one, please remember the order you see the elements not necessary the order you push the elements.  Stack and Queue does not allow you to iterate.  For example, you can use foreach to iterate a list in C# |
| Binary Search | If the data sture stored the values in order, it may provide a method for you to do binary search. The goal of binary search is not necessary find the exact value you specify. You may want the values no less than a specific value.  In C++, it is called as lower\_bound() and upper\_bound()  In Java and C# ,it is BinarySearch() |
| Modify | You can add, update or remove the elements in the collection based on the interfaces each type of class provides. However, you should NEVER modify the collection inside the loop which you iterate the collection. |

## Language Features and Impact

Assume we use C++, Java or C#. We need to understand the difference between languages and their impact.

|  |  |
| --- | --- |
| Operation | What is good and what is bad |
| C++ | **Good:**   * STL almost defines everything we need to solve the algorithm problems. The consistent interfaces on containers provide us a very easy way to access, operate elements in the container. such begin(), end(), pev(), next() and size(). The iterator type also allow us to easy access or modify the elements. * If the lower\_bound() and upper\_bound() is defined in every sorted data structure.   **Bad:**   * String methods are very limited, to parse a word, you have to do your own tokenization. * Copy the objects may cause extra time in some case, so in many case referenced type is required. * Memory allocation and clean up is explicit for Trie or Tree problems, you may need to do clean up (delete objects) at the end (if you allocate any objects). |
| C# | **Good:**   * You have most of the data types defined. * A LINQ syntax may make your life easy and code short. * Class object referenced as poiner by default, so you can change it in any level in the recusion calls. * String operation is complete.   **Bad:**   * The interfaces in different container are not consistent, for example in hashtable, it may return the key, in ArrayList, it returns the index of element. * Some names are confusion, for example List is ArrayList by default, not LinkedList * Some method is missing in obvious place, for example there is no BinarySearch in a SortedDictionary. You have to convert to List or a collection with IList interface implemented. * LINQ syntax is attractive, but only when you do not care about performance. The internal time complexity of LINQ query is always linear. * It does not have the concept of iterator for existing data structure, so you have to implement your own linked list for LRU problem. |
| Java | Java is very close to C#, with some more interfaces support. |

## 933. Number of Recent Calls

Easy

You have a RecentCounter class which counts the number of recent requests within a certain time frame.

Implement the RecentCounter class:

* RecentCounter() Initializes the counter with zero recent requests.
* int ping(int t) Adds a new request at time t, where t represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range [t - 3000, t].

It is **guaranteed** that every call to ping uses a strictly larger value of t than the previous call.

**Example 1:**

**Input**

["RecentCounter", "ping", "ping", "ping", "ping"]

[[], [1], [100], [3001], [3002]]

**Output**

[null, 1, 2, 3, 3]

**Explanation**

RecentCounter recentCounter = new RecentCounter();

recentCounter.ping(1); // requests = [1], range is [-2999,1], return 1

recentCounter.ping(100); // requests = [1, 100], range is [-2900,100], return 2

recentCounter.ping(3001); // requests = [1, 100, 3001], range is [1,3001], return 3

recentCounter.ping(3002); // requests = [1, 100, 3001, 3002], range is [2,3002], return 3

**Constraints:**

* 1 <= t <= 109
* Each test case will call ping with **strictly increasing** values of t.
* At most 104 calls will be made to ping.

### Analysis:

1. You can not iterate from previous time to current time, it is too slow.
2. Due to that it is incremental, you can store accumulated number of calls with time stamp.
3. Use binary search to find t-3000. In C# you can store series of time in the List.

/// <summary>

/// Leet code #933. Number of Recent Calls

///

/// Write a class RecentCounter to count recent requests.

///

/// It has only one method: ping(int t), where t represents some time in

/// milliseconds.

///

/// Return the number of pings that have been made from 3000 milliseconds

/// ago until now.

///

/// Any ping with time in [t - 3000, t] will count, including the current ping.

///

/// It is guaranteed that every call to ping uses a strictly larger value of

/// t than before.

///

/// Example 1:

///

/// Input: inputs = ["RecentCounter","ping","ping","ping","ping"],

/// inputs = [[],[1],[100],[3001],[3002]]

/// Output: [null,1,2,3,3]

///

///

/// Note:

///

/// Each test case will have at most 10000 calls to ping.

/// Each test case will call ping with strictly increasing values of t.

/// Each call to ping will have 1 <= t <= 10^9.

/// </summary>

class RecentCounter {

private:

map<int, int> m\_PingCount;

int m\_Total;

public:

RecentCounter()

{

m\_Total = 0;

}

int ping(int t)

{

m\_Total++;

m\_PingCount[t] = m\_Total;

auto itr = m\_PingCount.lower\_bound(t - 3000);

int prev = 0;

if (itr == m\_PingCount.begin()) prev = 0;

else

{

itr--;

prev = itr->second;

}

return m\_Total - prev;

}

};

## 900. RLE Iterator

Medium

We can use run-length encoding (i.e., **RLE**) to encode a sequence of integers. In a run-length encoded array of even length encoding (**0-indexed**), for all even i, encoding[i] tells us the number of times that the non-negative integer value encoding[i + 1] is repeated in the sequence.

* For example, the sequence arr = [8,8,8,5,5] can be encoded to be encoding = [3,8,2,5]. encoding = [3,8,0,9,2,5] and encoding = [2,8,1,8,2,5] are also valid **RLE** of arr.

Given a run-length encoded array, design an iterator that iterates through it.

Implement the RLEIterator class:

* RLEIterator(int[] encoded) Initializes the object with the encoded array encoded.
* int next(int n) Exhausts the next n elements and returns the last element exhausted in this way. If there is no element left to exhaust, return -1 instead.

**Example 1:**

**Input**

["RLEIterator", "next", "next", "next", "next"]

[[[3, 8, 0, 9, 2, 5]], [2], [1], [1], [2]]

**Output**

[null, 8, 8, 5, -1]

**Explanation**

RLEIterator rLEIterator = new RLEIterator([3, 8, 0, 9, 2, 5]); // This maps to the sequence [8,8,8,5,5].

rLEIterator.next(2); // exhausts 2 terms of the sequence, returning 8. The remaining sequence is now [8, 5, 5].

rLEIterator.next(1); // exhausts 1 term of the sequence, returning 8. The remaining sequence is now [5, 5].

rLEIterator.next(1); // exhausts 1 term of the sequence, returning 5. The remaining sequence is now [5].

rLEIterator.next(2); // exhausts 2 terms, returning -1. This is because the first term exhausted was 5,

but the second term did not exist. Since the last term exhausted does not exist, we return -1.

**Constraints:**

* 2 <= encoding.length <= 1000
* encoding.length is even.
* 0 <= encoding[i] <= 109
* 1 <= n <= 109
* At most 1000 calls will be made to next.

### Analysis:

1. Keep on deducting n and until it is 0.
2. On each position, if the count is less than remaining n, move to next position.
3. If count is more than remaining n, track unused count as offset (or offset can start from start of current position), for next use.

/// <summary>

/// Leet code #900. RLE Iterator

///

/// Write an iterator that iterates through a run-length encoded sequence.

///

/// The iterator is initialized by RLEIterator(int[] A), where A is a

/// run-length encoding of some sequence. More specifically, for all even i,

/// A[i] tells us the number of times that the non-negative integer value

/// A[i+1] is repeated in the sequence.

///

/// The iterator supports one function: next(int n), which exhausts the next

/// n elements (n >= 1) and returns the last element exhausted in this way.

/// If there is no element left to exhaust, next returns -1 instead.

///

/// For example, we start with A = [3,8,0,9,2,5], which is a run-length

/// encoding of the sequence [8,8,8,5,5]. This is because the sequence

/// can be read as "three eights, zero nines, two fives".

///

/// Example 1:

/// Input: ["RLEIterator","next","next","next","next"], [[[3,8,0,9,2,5]],

/// [2],[1],[1],[2]]

/// Output: [null,8,8,5,-1]

/// Explanation:

/// RLEIterator is initialized with RLEIterator([3,8,0,9,2,5]).

/// This maps to the sequence [8,8,8,5,5].

/// RLEIterator.next is then called 4 times:

///

/// .next(2) exhausts 2 terms of the sequence, returning 8. The remaining

/// sequence is now [8, 5, 5].

///

/// .next(1) exhausts 1 term of the sequence, returning 8. The remaining

/// sequence is now [5, 5].

///

/// .next(1) exhausts 1 term of the sequence, returning 5. The remaining

/// sequence is now [5].

///

/// .next(2) exhausts 2 terms, returning -1. This is because the first term

/// exhausted was 5,

/// but the second term did not exist. Since the last term exhausted does not

/// exist, we return -1.

///

/// Note:

///

/// 1. 0 <= A.length <= 1000

/// 2. A.length is an even integer.

/// 3. 0 <= A[i] <= 10^9

/// 4. There are at most 1000 calls to RLEIterator.next(int n) per test case.

/// 5. Each call to RLEIterator.next(int n) will have 1 <= n <= 10^9.

/// </summary>

class RLEIterator

{

private:

int m\_index;

int m\_offset;

vector<pair<int, int>> m\_data;

public:

RLEIterator(vector<int> A)

{

m\_index = 0;

m\_offset = 0;

for (size\_t i = 0; i < A.size(); i += 2)

{

m\_data.push\_back(make\_pair(A[i], A[i + 1]));

}

}

int next(int n)

{

int result = 0;

while (n > 0)

{

if (m\_index == m\_data.size())

{

result = -1;

break;

}

else if (m\_data[m\_index].first - m\_offset < n)

{

n -= (m\_data[m\_index].first - m\_offset);

m\_offset = 0;

m\_index++;

}

else if (m\_data[m\_index].first - m\_offset == n)

{

n -= (m\_data[m\_index].first - m\_offset);

result = m\_data[m\_index].second;

m\_offset = 0;

m\_index++;

break;

}

else

{

m\_offset += n;

result = m\_data[m\_index].second;

break;

}

}

return result;

}

};

## 355. Design Twitter

Medium

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user, and is able to see the 10 most recent tweets in the user's news feed.

Implement the Twitter class:

* Twitter() Initializes your twitter object.
* void postTweet(int userId, int tweetId) Composes a new tweet with ID tweetId by the user userId. Each call to this function will be made with a unique tweetId.
* List<Integer> getNewsFeed(int userId) Retrieves the 10 most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user themself. Tweets must be **ordered from most recent to least recent**.
* void follow(int followerId, int followeeId) The user with ID followerId started following the user with ID followeeId.
* void unfollow(int followerId, int followeeId) The user with ID followerId started unfollowing the user with ID followeeId.

**Example 1:**

**Input**

["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet", "getNewsFeed", "unfollow", "getNewsFeed"]

[[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]]

**Output**

[null, null, [5], null, null, [6, 5], null, [5]]

**Explanation**

Twitter twitter = new Twitter();

twitter.postTweet(1, 5); // User 1 posts a new tweet (id = 5).

twitter.getNewsFeed(1); // User 1's news feed should return a list with 1 tweet id -> [5]. return [5]

twitter.follow(1, 2); // User 1 follows user 2.

twitter.postTweet(2, 6); // User 2 posts a new tweet (id = 6).

twitter.getNewsFeed(1); // User 1's news feed should return a list with 2 tweet ids -> [6, 5]. Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.

twitter.unfollow(1, 2); // User 1 unfollows user 2.

twitter.getNewsFeed(1); // User 1's news feed should return a list with 1 tweet id -> [5], since user 1 is no longer following user 2.

**Constraints:**

* 1 <= userId, followerId, followeeId <= 500
* 0 <= tweetId <= 104
* All the tweets have **unique** IDs.
* At most 3 \* 104 calls will be made to postTweet, getNewsFeed, follow, and unfollow.

### Analysis:

1. Due to the performance constraint, you can not use push mode which push the twitter to the receivers. Pull twitter from followee is the right choice.
2. We need to get the latest twitters so each twitter should have a time stamp, you can use sequence id as time stamp.
3. To get latest 10 twitter from the followee, simply put the latest post from each of them and put in SortedSet or PriorityQueue and Iterate them.
4. User simply follow himself and can not be unfollowed.
5. Each user only need to keep 10 post, so you only have maximum 5000 posts for 500 users.

/// <summary>

/// LeetCode #355. Design Twitter

/// Design a simplified version of Twitter where users can post tweets,

/// follow/unfollow another user and

/// is able to see the 10 most recent tweets in the user's news feed.

/// Your design should support the following methods:

/// 1.postTweet(userId, tweetId): Compose a new tweet.

/// 2.getNewsFeed(userId): Retrieve the 10 most recent tweet ids in the user's

/// news feed.

/// Each item in the news feed must be posted by users who the user followed

/// or by the user herself.

/// Tweets must be ordered from most recent to least recent.

/// 3.follow(followerId, followeeId): Follower follows a followee.

/// 4.unfollow(followerId, followeeId): Follower unfollows a followee.

/// Example:

/// Twitter twitter = new Twitter();

/// User 1 posts a new tweet (id = 5).

/// twitter.postTweet(1, 5);

/// User 1's news feed should return a list with 1 tweet id -> [5].

/// twitter.getNewsFeed(1);

/// User 1 follows user 2.

/// twitter.follow(1, 2);

/// User 2 posts a new tweet (id = 6).

/// twitter.postTweet(2, 6);

/// User 1's news feed should return a list with 2 tweet ids -> [6, 5].

/// Tweet id 6 should precede tweet id 5 because it is posted after tweet id 5.

/// twitter.getNewsFeed(1);

///

/// User 1 unfollows user 2.

/// twitter.unfollow(1, 2);

/// User 1's news feed should return a list with 1 tweet id -> [5],

/// since user 1 is no longer following user 2.

/// twitter.getNewsFeed(1);

/// </summary>

class Twitter

{

private:

long m\_TimeTicks;

unordered\_map<int, vector<pair<long, int>>> m\_TwitterList;

unordered\_map<int, unordered\_set<int>> m\_FollowList;

public:

/// <summary>

/// Constructor an empty Twitter cache

/// </summary>

/// <returns></returns>

Twitter()

{

m\_TimeTicks = 0;

}

/// <summary>

/// Destructor of an Twitter

/// </summary>

/// <returns></returns>

~Twitter()

{

}

/// <summary>

/// Compose a new tweet

/// </summary>

/// <returns></returns>

void postTweet(int userId, int tweetId)

{

m\_TimeTicks++;

long now = m\_TimeTicks;

m\_TwitterList[userId].push\_back(make\_pair(m\_TimeTicks, tweetId));

}

/// <summary>

/// Retrieve the 10 most recent tweet ids in the user's news feed.

/// Each item in the news feed must be posted by users who the user

/// followed or by the user herself.

/// Tweets must be ordered from most recent to least recent.

/// </summary>

/// <returns></returns>

vector<int> getNewsFeed(int userId)

{

vector<int> result;

priority\_queue <pair<long, pair<int, int>>> candidate\_list;

if (!m\_TwitterList[userId].empty())

{

int clock = m\_TwitterList[userId].back().first;

int index = (int)m\_TwitterList[userId].size() - 1;

candidate\_list.push(make\_pair(clock, make\_pair(userId, index)));

}

for (int followee : m\_FollowList[userId])

{

if (!m\_TwitterList[followee].empty())

{

int clock = m\_TwitterList[followee].back().first;

int index = (int)m\_TwitterList[followee].size() - 1;

candidate\_list.push(make\_pair(clock, make\_pair(followee, index)));

}

}

for (size\_t i = 0; i < 10; i++)

{

if (candidate\_list.empty())

{

break;

}

pair<long, pair<int, int>> tweet\_itr = candidate\_list.top();

candidate\_list.pop();

int user\_id = tweet\_itr.second.first;

int index = tweet\_itr.second.second;

int tweet\_id = m\_TwitterList[user\_id][index].second;

result.push\_back(tweet\_id);

// push the next tweet of this user to priority queue

if (index > 0)

{

index--;

int clock = m\_TwitterList[user\_id][index].first;

candidate\_list.push(make\_pair(clock, make\_pair(user\_id, index)));

}

}

return result;

}

/// <summary>

/// Follower follows a followee. If the operation is invalid, it should

/// be a no-op.

/// </summary>

/// <param name="followerId">The follower id</param>

/// <param name="followeeId">The followee id</param>

/// <returns></returns>

void follow(int followerId, int followeeId)

{

if (followerId != followeeId)

{

m\_FollowList[followerId].insert(followeeId);

}

}

/// <summary>

/// Follower unfollows a followee. If the operation is invalid, it should

/// be a no-op.

/// </summary>

/// <param name="followerId">The follower id</param>

/// <param name="followeeId">The followee id</param>

/// <returns></returns>

void unfollow(int followerId, int followeeId)

{

m\_FollowList[followerId].erase(followeeId);

}

};

## 981. Time Based Key-Value Store

Medium

Create a timebased key-value store class TimeMap, that supports two operations.

1. set(string key, string value, int timestamp)

* Stores the key and value, along with the given timestamp.

2. get(string key, int timestamp)

* Returns a value such that set(key, value, timestamp\_prev) was called previously, with timestamp\_prev <= timestamp.
* If there are multiple such values, it returns the one with the largest timestamp\_prev.
* If there are no values, it returns the empty string ("").

**Example 1:**

**Input:** inputs = ["TimeMap","set","get","get","set","get","get"], inputs = [[],["foo","bar",1],["foo",1],["foo",3],["foo","bar2",4],["foo",4],["foo",5]]

**Output:** [null,null,"bar","bar",null,"bar2","bar2"]

**Explanation:**

TimeMap kv;

kv.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp = 1

kv.get("foo", 1); // output "bar"

kv.get("foo", 3); // output "bar" since there is no value corresponding to foo at timestamp 3 and timestamp 2, then the only value is at timestamp 1 ie "bar"

kv.set("foo", "bar2", 4);

kv.get("foo", 4); // output "bar2"

kv.get("foo", 5); //output "bar2"

**Example 2:**

**Input:** inputs = ["TimeMap","set","set","get","get","get","get","get"], inputs = [[],["love","high",10],["love","low",20],["love",5],["love",10],["love",15],["love",20],["love",25]]

**Output:** [null,null,null,"","high","high","low","low"]

**Note:**

1. All key/value strings are lowercase.
2. All key/value strings have length in the range [1, 100]
3. The timestamps for all TimeMap.set operations are strictly increasing.
4. 1 <= timestamp <= 10^7
5. TimeMap.set and TimeMap.get functions will be called a total of 120000 times (combined) per test case.

### Analysis:

This is a typical problem where you should apply binary search.

/// <summary>

/// Leet code #981. Time Based Key-Value Store

///

/// Create a timebased key-value store class TimeMap, that supports two

/// operations.

/// 1. set(string key, string value, int timestamp)

/// Stores the key and value, along with the given timestamp.

/// 2. get(string key, int timestamp)

/// Returns a value such that set(key, value, timestamp\_prev) was called

/// previously, with timestamp\_prev <= timestamp.

/// 3. If there are multiple such values, it returns the one with the largest

/// timestamp\_prev.

/// 4. If there are no values, it returns the empty string ("").

///

/// Example 1:

/// Input:

/// inputs = ["TimeMap","set","get","get","set","get","get"],

/// inputs = [[],["foo","bar",1],["foo",1],["foo",3],["foo","bar2",4],

/// ["foo",4],["foo",5]]

/// Output: [null,null,"bar","bar",null,"bar2","bar2"]

/// Explanation:

/// TimeMap kv;

/// kv.set("foo", "bar", 1); // store the key "foo" and value "bar" along with

/// timestamp = 1

/// kv.get("foo", 1); // output "bar"

/// kv.get("foo", 3); // output "bar" since there is no value corresponding to

/// foo at timestamp 3 and timestamp 2, then the only value is at timestamp 1

/// ie "bar"

/// kv.set("foo", "bar2", 4);

/// kv.get("foo", 4); // output "bar2"

/// kv.get("foo", 5); //output "bar2"

///

/// Example 2:

/// Input: inputs = ["TimeMap","set","set","get","get","get","get","get"],

/// inputs = [[],["love","high",10],["love","low",20],["love",5],["love",10],

/// ["love",15],["love",20],["love",25]]

/// Output: [null,null,null,"","high","high","low","low"]

///

/// Note:

/// 1. All key/value strings are lowercase.

/// 2. All key/value strings have length in the range [1, 100]

/// 3. The timestamps for all TimeMap.set operations are strictly increasing.

/// 4. 1 <= timestamp <= 10^7

/// 5. TimeMap.set and TimeMap.get functions will be called a total of 120000

/// times (combined) per test case.

/// </summary>

class TimeMap {

private:

unordered\_map<string, map<int, string>> m\_dictionary;

public:

/\*\* Initialize your data structure here. \*/

TimeMap()

{

}

void set(string key, string value, int timestamp)

{

m\_dictionary[key][-timestamp] = value;

}

string get(string key, int timestamp)

{

auto itr = m\_dictionary[key].lower\_bound(-timestamp);

if (itr == m\_dictionary[key].end())

{

return "";

}

else

{

return itr->second;

}

}

};

## 911. Online Election

Medium

In an election, the i-th vote was cast for persons[i] at time times[i].

Now, we would like to implement the following query function: TopVotedCandidate.q(int t) will return the number of the person that was leading the election at time t.

Votes cast at time t will count towards our query.  In the case of a tie, the most recent vote (among tied candidates) wins.

**Example 1:**

**Input:** ["TopVotedCandidate","q","q","q","q","q","q"], [[[0,1,1,0,0,1,0],[0,5,10,15,20,25,30]],[3],[12],[25],[15],[24],[8]]

**Output:** [null,0,1,1,0,0,1]

**Explanation:**

At time 3, the votes are [0], and 0 is leading.

At time 12, the votes are [0,1,1], and 1 is leading.

At time 25, the votes are [0,1,1,0,0,1], and 1 is leading (as ties go to the most recent vote.)

This continues for 3 more queries at time 15, 24, and 8.

**Note:**

1. 1 <= persons.length = times.length <= 5000
2. 0 <= persons[i] <= persons.length
3. times is a strictly increasing array with all elements in [0, 10^9].
4. TopVotedCandidate.q is called at most 10000 times per test case.
5. TopVotedCandidate.q(int t) is always called with t >= times[0].

### Analysis:

Because the time is strictly increasing, which make our life easy, we only need to track the most voted person so far, and record it along with the time stamp. When query we can do binary search in time with the time after the query time, and then move a step back and get the answer.

/// <summary>

/// Leet code #911. Online Election

///

/// In an election, the i-th vote was cast for persons[i] at time times[i].

///

/// Now, we would like to implement the following query function:

/// TopVotedCandidate.q(int t) will return the number of the person that was

/// leading the election at time t.

///

/// Votes cast at time t will count towards our query. In the case of a tie,

/// the most recent vote (among tied candidates) wins.

///

/// Example 1:

///

/// Input: ["TopVotedCandidate","q","q","q","q","q","q"], [[[0,1,1,0,0,1,0],

/// [0,5,10,15,20,25,30]],[3],[12],[25],[15],[24],[8]]

/// Output: [null,0,1,1,0,0,1]

/// Explanation:

/// At time 3, the votes are [0], and 0 is leading.

/// At time 12, the votes are [0,1,1], and 1 is leading.

/// At time 25, the votes are [0,1,1,0,0,1], and 1 is leading (as ties go to

/// the most recent vote.)

/// This continues for 3 more queries at time 15, 24, and 8.

///

/// Note:

///

/// 1. 1 <= persons.length = times.length <= 5000

/// 2. 0 <= persons[i] <= persons.length

/// 3. times is a strictly increasing array with all elements in [0, 10^9].

/// 4. TopVotedCandidate.q is called at most 10000 times per test case.

/// 5. TopVotedCandidate.q(int t) is always called with t >= times[0].

/// </summary>

class TopVotedCandidate

{

private:

unordered\_map<int, int> m\_persons;

vector<int> m\_timestamp;

vector<int> m\_vote;

public:

TopVotedCandidate(vector<int> persons, vector<int> times)

{

int max\_person = -1;

for (size\_t i = 0; i < persons.size(); i++)

{

m\_persons[persons[i]]++;

if (max\_person == -1)

{

max\_person = persons[i];

m\_timestamp.push\_back(times[i]);

m\_vote.push\_back(max\_person);

}

else if (m\_persons[persons[i]] >= m\_persons[max\_person])

{

max\_person = persons[i];

m\_timestamp.push\_back(times[i]);

m\_vote.push\_back(max\_person);

}

}

}

int q(int t)

{

auto pos = upper\_bound(m\_timestamp.begin(), m\_timestamp.end(), t);

pos--;

return m\_vote[pos - m\_timestamp.begin()];

}

};

## 1146. Snapshot Array

Medium

Implement a SnapshotArray that supports the following interface:

* SnapshotArray(int length) initializes an array-like data structure with the given length.  **Initially, each element equals 0**.
* void set(index, val) sets the element at the given index to be equal to val.
* int snap() takes a snapshot of the array and returns the snap\_id: the total number of times we called snap() minus 1.
* int get(index, snap\_id) returns the value at the given index, at the time we took the snapshot with the given snap\_id

**Example 1:**

**Input:** ["SnapshotArray","set","snap","set","get"]

[[3],[0,5],[],[0,6],[0,0]]

**Output:** [null,null,0,null,5]

**Explanation:**

SnapshotArray snapshotArr = new SnapshotArray(3); // set the length to be 3

snapshotArr.set(0,5); // Set array[0] = 5

snapshotArr.snap(); // Take a snapshot, return snap\_id = 0

snapshotArr.set(0,6);

snapshotArr.get(0,0); // Get the value of array[0] with snap\_id = 0, return 5

**Constraints:**

* 1 <= length <= 50000
* At most 50000 calls will be made to set, snap, and get.
* 0 <= index < length
* 0 <= snap\_id < (the total number of times we call snap())
* 0 <= val <= 10^9

### Analysis:

We track on every index with value for different versions in a sorted maner, when we search a value for specific version we do binary search, keep decreasing version help us to fit for the lower\_bound() in C++, but if we do not do so, we just need to move one step back on upper\_bound().

Here we use sorted map not vector, it is because if we have multiple set in same snapshot id, we will let it simply overwrite.

/// <summary>

/// Leet code #1146. Snapshot Array

///

/// Implement a SnapshotArray that supports the following interface:

///

/// SnapshotArray(int length) initializes an array-like data structure

/// with the given length. Initially, each element equals 0.

/// void set(index, val) sets the element at the given index to be equal

/// to val.

/// int snap() takes a snapshot of the array and returns the snap\_id:

/// the total number of times we called snap() minus 1.

/// int get(index, snap\_id) returns the value at the given index, at the

/// time we took the snapshot with the given snap\_id

///

///

/// Example 1:

/// Input: ["SnapshotArray","set","snap","set","get"]

/// [[3],[0,5],[],[0,6],[0,0]]

/// Output: [null,null,0,null,5]

/// Explanation:

/// // set the length to be 3

/// SnapshotArray snapshotArr = new SnapshotArray(3);

/// snapshotArr.set(0,5); // Set array[0] = 5

/// snapshotArr.snap(); // Take a snapshot, return snap\_id = 0

/// snapshotArr.set(0,6);

/// // Get the value of array[0] with snap\_id = 0, return 5

/// snapshotArr.get(0,0);

///

/// Constraints:

/// 1. 1 <= length <= 50000

/// 2. At most 50000 calls will be made to set, snap, and get.

/// 3. 0 <= index < length

/// 4. 0 <= snap\_id < (the total number of times we call snap())

/// 5. 0 <= val <= 10^9

/// </summary>

class SnapshotArray

{

private:

vector<map<int, int>> m\_arr;

int snapshot\_id;

public:

SnapshotArray(int length)

{

snapshot\_id = 0;

m\_arr = vector<map<int, int>>(length);

}

void set(int index, int val)

{

m\_arr[index][-snapshot\_id] = val;

}

int snap()

{

int result = snapshot\_id;

snapshot\_id++;

return result;

}

int get(int index, int snap\_id)

{

auto itr = m\_arr[index].lower\_bound(-snap\_id);

return itr->second;

}

};

## 901. Online Stock Span

Medium

Write a class StockSpanner which collects daily price quotes for some stock, and returns the *span* of that stock's price for the current day.

The span of the stock's price today is defined as the maximum number of consecutive days (starting from today and going backwards) for which the price of the stock was less than or equal to today's price.

For example, if the price of a stock over the next 7 days were [100, 80, 60, 70, 60, 75, 85], then the stock spans would be [1, 1, 1, 2, 1, 4, 6].

**Example 1:**

**Input:** ["StockSpanner","next","next","next","next","next","next","next"], [[],[100],[80],[60],[70],[60],[75],[85]]

**Output:** [null,1,1,1,2,1,4,6]

**Explanation:**

First, S = StockSpanner() is initialized. Then:

S.next(100) is called and returns 1,

S.next(80) is called and returns 1,

S.next(60) is called and returns 1,

S.next(70) is called and returns 2,

S.next(60) is called and returns 1,

S.next(75) is called and returns 4,

S.next(85) is called and returns 6.

Note that (for example) S.next(75) returned 4, because the last 4 prices

(including today's price of 75) were less than or equal to today's price.

**Note:**

1. Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.
2. There will be at most 10000 calls to StockSpanner.next per test case.
3. There will be at most 150000 calls to StockSpanner.next across all test cases.
4. The total time limit for this problem has been reduced by 75% for C++, and 50% for all other languages.

### Analysis:

To track how many prices are less than today, you can use a stack to track the prices, for any new price comes we should pop up the prices in the stack which are less than or equal to current, and record the accumulated counters there.

/// <summary>

/// Leet code #901. Online Stock Span

///

/// Write a class StockSpanner which collects daily price quotes for some

/// stock, and returns the span of that stock's price for the current day.

///

/// The span of the stock's price today is defined as the maximum number of

/// consecutive days (starting from today and going backwards) for which the

/// price of the stock was less than or equal to today's price.

///

/// For example, if the price of a stock over the next 7 days were [100, 80,

/// 60, 70, 60, 75, 85], then the stock spans would be [1, 1, 1, 2, 1, 4, 6].

///

/// Example 1:

///

/// Input: ["StockSpanner","next","next","next","next","next","next","next"],

/// [[],[100],[80],[60],[70],[60],[75],[85]]

/// Output: [null,1,1,1,2,1,4,6]

/// Explanation:

/// First, S = StockSpanner() is initialized. Then:

/// S.next(100) is called and returns 1,

/// S.next(80) is called and returns 1,

/// S.next(60) is called and returns 1,

/// S.next(70) is called and returns 2,

/// S.next(60) is called and returns 1,

/// S.next(75) is called and returns 4,

/// S.next(85) is called and returns 6.

///

/// Note that (for example) S.next(75) returned 4, because the last 4 prices

/// (including today's price of 75) were less than or equal to today's price.

///

///

/// Note:

///

/// 1. Calls to StockSpanner.next(int price) will have 1 <= price <= 10^5.

/// 2. There will be at most 10000 calls to StockSpanner.next per test case.

/// 3. There will be at most 150000 calls to StockSpanner.next across all test

/// cases.

/// 4. The total time limit for this problem has been reduced by 75% for C++,

/// and 50% for all other languages.

/// </summary>

class StockSpanner

{

private:

vector<pair<int, int>> m\_prices;

public:

StockSpanner()

{

}

int next(int price)

{

int result = 1;

while (!m\_prices.empty())

{

pair<int, int> previous = m\_prices.back();

if (previous.first > price)

{

break;

}

m\_prices.pop\_back();

result += previous.second;

}

m\_prices.push\_back(make\_pair(price, result));

return result;

}

};

## 716. Max Stack

Medium

Design a max stack that supports push, pop, top, peekMax and popMax.

1. push(x) -- Push element x onto stack.
2. pop() -- Remove the element on top of the stack and return it.
3. top() -- Get the element on the top.
4. peekMax() -- Retrieve the maximum element in the stack.
5. popMax() -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

**Example 1:**

MaxStack stack = new MaxStack();

stack.push(5);

stack.push(1);

stack.push(5);

stack.top(); -> 5

stack.popMax(); -> 5

stack.top(); -> 1

stack.peekMax(); -> 5

stack.pop(); -> 1

stack.top(); -> 5

**Note:**

1. -1e7 <= x <= 1e7
2. Number of operations won't exceed 10000.
3. The last four operations won't be called when stack is empty.

### Analysis:

Because you want to remove the maximum value, and it is not necessary at the top of stack, so you should use list as stack and use sorted map to keep track on all the value and their position, the the position is stored in a stack (can be also a list) as the value of the sorted map.

/// <summary>

/// Leet code #716. Max Stack

///

/// Design a max stack that supports push, pop, top, peekMax and popMax.

///

/// push(x) -- Push element x onto stack.

/// pop() -- Remove the element on top of the stack and return it.

/// top() -- Get the element on the top.

/// peekMax() -- Retrieve the maximum element in the stack.

/// popMax() -- Retrieve the maximum element in the stack, and remove it.

/// If you find more than one maximum elements, only remove the top-most

/// one.

/// Example 1:

/// MaxStack stack = new MaxStack();

/// stack.push(5);

/// stack.push(1);

/// stack.push(5);

/// stack.top(); -> 5

/// stack.popMax(); -> 5

/// stack.top(); -> 1

/// stack.peekMax(); -> 5

/// stack.pop(); -> 1

/// stack.top(); -> 5

/// Note:

/// 1. -1e7 <= x <= 1e7

/// 2. Number of operations won't exceed 10000.

/// 3. The last four operations won't be called when stack is empty.

/// </summary>

class MaxStack {

private:

map<int, stack<list<int>::iterator>> m\_ValueMap;

list<int> m\_List;

public:

/// <summary>

/// Constructor an empty max stack

/// </summary>

MaxStack()

{

}

void push(int x)

{

// Add to stack stack, last one first

m\_List.push\_front(x);

// Add to value map, biggest first

m\_ValueMap[-x].push(m\_List.begin());

}

int pop()

{

// take first item from m\_StackMap, which is the last item based on index

auto itr = m\_List.begin();

int value = \*itr;

m\_List.pop\_front();

// take out the top index from specific value

m\_ValueMap[-value].pop();

if (m\_ValueMap[-value].empty()) m\_ValueMap.erase(-value);

return value;

}

int top()

{

// take first item from m\_StackMap, which is the last item based on index

auto itr = m\_List.begin();

int value = \*itr;

return value;

}

int peekMax()

{

// take first item from m\_ValueMap, and the first one in the queue

auto itr = m\_ValueMap.begin();

int value = -itr->first;

return value;

}

int popMax()

{

// take first item from m\_ValueMap, and the first one in the queue

auto itr = m\_ValueMap.begin();

int value = -itr->first;

auto list\_itr = itr->second.top();

m\_List.erase(itr->second.top());

itr->second.pop();

if (itr->second.empty()) m\_ValueMap.erase(itr->first);

return value;

}

};

## 146. LRU Cache

Medium

Design and implement a data structure for [Least Recently Used (LRU) cache](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU). It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  
put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

The cache is initialized with a **positive** capacity.

**Follow up:**  
Could you do both operations in **O(1)** time complexity?

**Example:**

LRUCache cache = new LRUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.put(4, 4); // evicts key 1

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

### Analysis:

You can use the list to keep track the key value pair, and use hashtable to keep the key to the position in list, so you can move element or delete it in list.

/// <summary>

/// Leet code #146. LRU Cache

///

/// Design and implement a data structure for Least Recently Used (LRU) cache.

/// It should support the following operations: get and put.

/// get(key) - Get the value (will always be positive) of the key if the key

/// exists in the cache, otherwise return -1.

/// put(key, value) - Set or insert the value if the key is not already

/// present. When the cache reached its capacity, it should invalidate the

/// least recently used item before inserting a new item.

/// The cache is initialized with a positive capacity.

/// Follow up:

/// Could you do both operations in O(1) time complexity?

/// Example:

/// LRUCache cache = new LRUCache( 2 /\* capacity \*/ );

///

/// cache.put(1, 1);

/// cache.put(2, 2);

/// cache.get(1); // returns 1

/// cache.put(3, 3); // evicts key 2

/// cache.get(2); // returns -1 (not found)

/// cache.put(4, 4); // evicts key 1

/// cache.get(1); // returns -1 (not found)

/// cache.get(3); // returns 3

/// cache.get(4); // returns 4

/// </summary>

class LRUCache

{

private:

size\_t m\_Capacity;

list<pair<int, int>> m\_List;

unordered\_map<int, list<pair<int, int>>::iterator> m\_map;

public:

/// <summary>

/// Constructor an empty LRU cache

/// </summary>

/// <param name="capacity">capacity</param>

/// <returns></returns>

LRUCache(int capacity)

{

m\_Capacity = capacity;

}

/// <summary>

/// Destructor of an LRUCache

/// </summary>

/// <returns></returns>

~LRUCache()

{

}

/// <summary>

/// Set the key value pair in the LRU cache.

/// </summary>

/// <param name="key">The key</param>

/// <param name="value">The value</param>

/// <returns></returns>

void put(int key, int value)

{

if (m\_map.count(key) == 0)

{

m\_List.push\_front(make\_pair(key, value));

if (m\_List.size() > m\_Capacity)

{

pair<int, int> pair = m\_List.back();

m\_map.erase(pair.first);

m\_List.pop\_back();

}

m\_map[key] = m\_List.begin();

}

else

{

// take out old value, insert new one

m\_List.erase(m\_map[key]);

m\_List.push\_front(make\_pair(key, value));

m\_map[key] = m\_List.begin();

}

}

/// <summary>

/// Get the value(will always be positive) of the key if the key exists

/// in the cache, otherwise return -1.

/// </summary>

/// <returns>the value</returns>

int get(int key)

{

if (m\_map.find(key) == m\_map.end())

{

return -1;

}

list<pair<int, int>>::iterator iterator = m\_map[key];

pair<int, int> pair = \*iterator;

m\_List.erase(iterator);

m\_List.push\_front(pair);

m\_map[key] = m\_List.begin();

return pair.second;

}

};

## 2166. Design Bitset

Medium

A **Bitset** is a data structure that compactly stores bits.

Implement the Bitset class:

* Bitset(int size) Initializes the Bitset with size bits, all of which are 0.
* void fix(int idx) Updates the value of the bit at the index idx to 1. If the value was already 1, no change occurs.
* void unfix(int idx) Updates the value of the bit at the index idx to 0. If the value was already 0, no change occurs.
* void flip() Flips the values of each bit in the Bitset. In other words, all bits with value 0 will now have value 1 and vice versa.
* boolean all() Checks if the value of **each** bit in the Bitset is 1. Returns true if it satisfies the condition, false otherwise.
* boolean one() Checks if there is **at least one** bit in the Bitset with value 1. Returns true if it satisfies the condition, false otherwise.
* int count() Returns the **total number** of bits in the Bitset which have value 1.
* String toString() Returns the current composition of the Bitset. Note that in the resultant string, the character at the ith index should coincide with the value at the ith bit of the Bitset.

**Example 1:**

**Input**

["Bitset", "fix", "fix", "flip", "all", "unfix", "flip", "one", "unfix", "count", "toString"]

[[5], [3], [1], [], [], [0], [], [], [0], [], []]

**Output**

[null, null, null, null, false, null, null, true, null, 2, "01010"]

**Explanation**

Bitset bs = new Bitset(5); // bitset = "00000".

bs.fix(3); // the value at idx = 3 is updated to 1, so bitset = "00010".

bs.fix(1); // the value at idx = 1 is updated to 1, so bitset = "01010".

bs.flip(); // the value of each bit is flipped, so bitset = "10101".

bs.all(); // return False, as not all values of the bitset are 1.

bs.unfix(0); // the value at idx = 0 is updated to 0, so bitset = "00101".

bs.flip(); // the value of each bit is flipped, so bitset = "11010".

bs.one(); // return True, as there is at least 1 index with value 1.

bs.unfix(0); // the value at idx = 0 is updated to 0, so bitset = "01010".

bs.count(); // return 2, as there are 2 bits with value 1.

bs.toString(); // return "01010", which is the composition of bitset.

**Constraints:**

* 1 <= size <= 105
* 0 <= idx <= size - 1
* At most 105 calls will be made **in total** to fix, unfix, flip, all, one, count, and toString.
* At least one call will be made to all, one, count, or toString.
* At most 5 calls will be made to toString.

### Analysis:

You need to return the result in string, so keep it in string, any possible O(N) call such as flip, count, all, zero and to\_string() will cause TLE. So we need to keep all of them in O(1). To make slip as O(1) we simply keep original string and inverse string (flip 0 to 1) as opposite. When flip, keep flip times, if even use origin, if odd use inverse.

/// <summary>

/// Leet Code 2166. Design Bitset

///

/// Medium

///

/// A Bitset is a data structure that compactly stores bits.

///

/// Implement the Bitset class:

/// Bitset(int size) Initializes the Bitset with size bits, all of which

/// are 0.

/// void fix(int idx) Updates the value of the bit at the index idx to 1.

/// If the value was already 1, no change occurs.

/// void unfix(int idx) Updates the value of the bit at the index idx

/// to 0. If the value was already 0, no change occurs.

/// void flip() Flips the values of each bit in the Bitset. In other w

/// ords, all bits with value 0 will now have value 1 and vice versa.

/// boolean all() Checks if the value of each bit in the Bitset is 1.

/// Returns true if it satisfies the condition, false otherwise.

/// boolean one() Checks if there is at least one bit in the Bitset

/// with value 1. Returns true if it satisfies the condition, false

/// otherwise.

/// int count() Returns the total number of bits in the Bitset which have

/// value 1.

/// String toString() Returns the current composition of the Bitset. Note

/// that in the resultant string, the character at the ith index should

/// coincide with the value at the ith bit of the Bitset.

///

/// Example 1:

/// Input

/// ["Bitset", "fix", "fix", "flip", "all", "unfix", "flip", "one",

/// "unfix", "count", "toString"]

/// [[5], [3], [1], [], [], [0], [], [], [0], [], []]

/// Output

/// [null, null, null, null, false, null, null, true, null, 2, "01010"]

///

/// Explanation

/// Bitset bs = new Bitset(5); // bitset = "00000".

/// bs.fix(3); // the value at idx = 3 is updated to 1, so

/// bitset = "00010".

/// bs.fix(1); // the value at idx = 1 is updated to 1, so

/// bitset = "01010".

/// bs.flip(); // the value of each bit is flipped, so

/// bitset = "10101".

/// bs.all(); // return False, as not all values of the bitset are 1.

/// bs.unfix(0); // the value at idx = 0 is updated to 0, so

/// bitset = "00101".

/// bs.flip(); // the value of each bit is flipped, so

/// bitset = "11010".

/// bs.one(); // return True, as there is at least 1 index with

/// value 1.

/// bs.unfix(0); // the value at idx = 0 is updated to 0, so

/// bitset = "01010".

/// bs.count(); // return 2, as there are 2 bits with value 1.

/// bs.toString(); // return "01010", which is the composition of bitset.

///

/// Constraints:

/// 1. 1 <= size <= 10^5

/// 2. 0 <= idx <= size - 1

/// 3. At most 10^5 calls will be made in total to fix, unfix, flip, all,

/// one, count, and toString.

/// 4. At least one call will be made to all, one, count, or toString.

/// 5. At most 5 calls will be made to toString.

/// </summary>

class Bitset

{

private:

string m\_bits;

string m\_reverse;

int m\_flip;

size\_t m\_count;

public:

Bitset(int size)

{

m\_bits = string(size, '0');

m\_reverse = string(size, '1');

m\_count = 0;

m\_flip = 0;

}

void fix(int idx)

{

if (m\_flip == 0)

{

if (m\_bits[idx] == '0')

{

m\_bits[idx] = '1';

m\_reverse[idx] = '0';

m\_count++;

}

}

else

{

if (m\_reverse[idx] == '0')

{

m\_reverse[idx] = '1';

m\_bits[idx] = '0';

m\_count--;

}

}

}

void unfix(int idx)

{

if (m\_flip == 0)

{

if (m\_bits[idx] == '1')

{

m\_bits[idx] = '0';

m\_reverse[idx] = '1';

m\_count--;

}

}

else

{

if (m\_reverse[idx] == '1')

{

m\_reverse[idx] = '0';

m\_bits[idx] = '1';

m\_count++;

}

}

}

void flip()

{

m\_flip = 1 - m\_flip;

}

bool all()

{

if (m\_flip == 0)

{

return (m\_count == m\_bits.size());

}

else

{

return (m\_count == 0);

}

}

bool one()

{

if (m\_flip == 0)

{

return (m\_count > 0);

}

else

{

return (m\_count < m\_bits.size());

}

}

int count()

{

if (m\_flip == 0)

{

return m\_count;

}

else

{

return m\_bits.size() - m\_count;

}

}

string toString()

{

if (m\_flip == 0)

{

return m\_bits;

}

else

{

return m\_reverse;

}

}

};

## 460. LFU Cache

Hard

Design and implement a data structure for [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache. It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  
put(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least **recently** used key would be evicted.

Note that the number of times an item is used is the number of calls to the get and put functions for that item since it was inserted. This number is set to zero when the item is removed.

**Follow up:**  
Could you do both operations in **O(1)** time complexity?

**Example:**

LFUCache cache = new LFUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.get(3); // returns 3.

cache.put(4, 4); // evicts key 1.

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

### Analysis:

For this problem we should track every key with its frequency and increase the frequency when they are accessed. Every frequency keeps its own list as a queue, first in first out, we remember the global minimum frequency so we can push out it when capacity is full.

/// <summary>

/// Leet code #460. LFU Cache

/// Design and implement a data structure for Least Frequently

/// Used (LFU) cache. It should support the following operations: get and put.

///

/// get(key) - Get the value (will always be positive) of the key if the key

/// exists in the cache, otherwise return -1.

/// put(key, value) - Set or insert the value if the key is not already present.

/// When the cache reaches its capacity, it should invalidate

/// the least frequently used item before inserting a new

/// item. For the purpose of this problem, when there is a

/// tie (i.e., two or more keys that have the same frequency),

/// the least recently used key would be evicted.

///

/// Follow up:

/// Could you do both operations in O(1) time complexity?

/// Example:

/// LFUCache cache = new LFUCache( 2 /\* capacity \*/ );

/// cache.put(1, 1);

/// cache.put(2, 2);

/// cache.get(1); // returns 1

/// cache.put(3, 3); // evicts key 2

/// cache.get(2); // returns -1 (not found)

/// cache.get(3); // returns 3.

/// cache.put(4, 4); // evicts key 1.

/// cache.get(1); // returns -1 (not found)

/// cache.get(3); // returns 3

/// cache.get(4); // returns 4

/// </summary>

class LFUCache

{

private:

size\_t m\_capacity;

int m\_minFreq;

// map key to the frequency list position

unordered\_map<int, pair<int, list<pair<int, int>>::iterator>> m\_keyMap;

// map frequency to the value list

unordered\_map<int, list<pair<int, int>>> m\_freqMap;

public:

LFUCache(int capacity)

{

m\_capacity = capacity;

m\_minFreq = 0;

}

int get(int key)

{

if (m\_keyMap.count(key) == 0)

{

return -1;

}

pair<int, list<pair<int, int>>::iterator> freq\_itr = m\_keyMap[key];

int frequency = freq\_itr.first;

list<pair<int, int>>::iterator key\_val\_itr = freq\_itr.second;

pair<int, int> key\_val = \*key\_val\_itr;

// key\_val\_itr become invalid

m\_freqMap[frequency].erase(key\_val\_itr);

if (m\_freqMap[frequency].empty())

{

m\_freqMap.erase(frequency);

if (m\_minFreq == frequency) m\_minFreq++;

}

frequency++;

m\_freqMap[frequency].push\_front(key\_val);

// new key value pair iterator

key\_val\_itr = m\_freqMap[frequency].begin();

// assign back to kay map

m\_keyMap[key] = make\_pair(frequency, key\_val\_itr);

return key\_val.second;

}

void put(int key, int value)

{

if (m\_capacity == 0) return;

list<pair<int, int>>::iterator key\_val\_itr;

if (get(key) != -1)

{

key\_val\_itr = m\_keyMap[key].second;

key\_val\_itr->second = value;

}

else

{

if (m\_keyMap.size() == m\_capacity)

{

// erase LFU key from frequency map

pair<int, int> key\_val = m\_freqMap[m\_minFreq].back();

m\_freqMap[m\_minFreq].pop\_back();

if (m\_freqMap[m\_minFreq].empty())

{

m\_freqMap.erase(m\_minFreq);

}

// erase LFU key from key map

m\_keyMap.erase(key\_val.first);

}

m\_minFreq = 1;

m\_freqMap[m\_minFreq].push\_front(make\_pair(key, value));

// new key value pair iterator

key\_val\_itr = m\_freqMap[m\_minFreq].begin();

// assign back to kay map

m\_keyMap[key] = make\_pair(m\_minFreq, key\_val\_itr);

}

}

};

## 895. Maximum Frequency Stack

Hard

Implement FreqStack, a class which simulates the operation of a stack-like data structure.

FreqStack has two functions:

* push(int x), which pushes an integer x onto the stack.
* pop(), which **removes** and returns the most frequent element in the stack.
  + If there is a tie for most frequent element, the element closest to the top of the stack is removed and returned.

**Example 1:**

**Input:**

["FreqStack","push","push","push","push","push","push","pop","pop","pop","pop"],

[[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

**Output:** [null,null,null,null,null,null,null,5,7,5,4]

**Explanation**:

After making six .push operations, the stack is [5,7,5,7,4,5] from bottom to top. Then:

pop() -> returns 5, as 5 is the most frequent.

The stack becomes [5,7,5,7,4].

pop() -> returns 7, as 5 and 7 is the most frequent, but 7 is closest to the top.

The stack becomes [5,7,5,4].

pop() -> returns 5.

The stack becomes [5,7,4].

pop() -> returns 4.

The stack becomes [5,7].

**Note:**

* Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.
* It is guaranteed that FreqStack.pop() won't be called if the stack has zero elements.
* The total number of FreqStack.push calls will not exceed 10000 in a single test case.
* The total number of FreqStack.pop calls will not exceed 10000 in a single test case.
* The total number of FreqStack.push and FreqStack.pop calls will not exceed 150000 across all test cases.

### Analysis:

We use a hashtable to track the frequency on each value, and a vector to store the value for every frequency. When we push a new value we increase its frequency and store the value in the corresponding slot (index = frequency -1), and keep all values in the slot as stack, last in first out.

/// <summary>

/// Leet code #895. Maximum Frequency Stack

///

/// Implement FreqStack, a class which simulates the operation of a stack-like

/// data structure.

///

/// FreqStack has two functions:

///

/// push(int x), which pushes an integer x onto the stack.

/// pop(), which removes and returns the most frequent element in the stack.

/// If there is a tie for most frequent element, the element closest to the

/// top of the stack is removed and returned.

///

/// Example 1:

///

/// Input:

/// [

/// "FreqStack","push","push","push","push","push","push","pop","pop","pop",

/// "pop"

/// ],

/// [[],[5],[7],[5],[7],[4],[5],[],[],[],[]]

/// Output: [null,null,null,null,null,null,null,5,7,5,4]

///

/// Explanation:

/// After making six .push operations, the stack is [5,7,5,7,4,5] from bottom

/// to top. Then:

///

/// pop() -> returns 5, as 5 is the most frequent.

/// The stack becomes [5,7,5,7,4].

///

/// pop() -> returns 7, as 5 and 7 is the most frequent, but 7 is closest to

/// the top.

/// The stack becomes [5,7,5,4].

///

/// pop() -> returns 5.

/// The stack becomes [5,7,4].

///

/// pop() -> returns 4.

/// The stack becomes [5,7].

/// Note:

/// 1. Calls to FreqStack.push(int x) will be such that 0 <= x <= 10^9.

/// 2. It is guaranteed that FreqStack.pop() won't be called if the stack has

/// zero elements.

/// 3. The total number of FreqStack.push calls will not exceed 10000 in a

/// single test case.

/// 4. The total number of FreqStack.pop calls will not exceed 10000 in a

/// single test case.

/// 5. The total number of FreqStack.push and FreqStack.pop calls will not

/// exceed 150000 across all test cases.

/// </summary>

class FreqStack

{

private:

unordered\_map<int, int> m\_Frequency;

vector<stack<int>> m\_stack;

public:

FreqStack()

{

}

void push(int x)

{

if (m\_Frequency.count(x) == 0)

{

m\_Frequency[x] = 0;

}

else

{

m\_Frequency[x]++;

}

if (m\_stack.size() == m\_Frequency[x])

{

m\_stack.push\_back(stack<int>());

}

m\_stack[m\_Frequency[x]].push(x);

}

int pop()

{

int result = m\_stack[m\_stack.size() - 1].top();

m\_stack[m\_stack.size() - 1].pop();

if (m\_stack[m\_stack.size() - 1].empty())

{

m\_stack.pop\_back();

}

m\_Frequency[result]--;

if (m\_Frequency[result] == -1)

{

m\_Frequency.erase(result);

}

return result;

}

};