# Design Data Structure

In this chapter, we will discuss the leetcode design problems. The target of these problem is to design the right data structure to solve the problem in specific time complexity.

The following data structure are based on C++, but you should be able to find the corresponding ones in Java, C# and Python.

1. Vector, the auto growing array, access element on index, delete or insert an element may cause many elements to move. If the value is in order use lower\_bound can do binary search by value.
2. Unordered\_map, the hashtable, access by key in O(1), however calculate hash key is very complicate, so it is normally 10 to 40 times then access vector.
3. Map, the BST based sorted table, access a element is O(log(n)), access the smallest is O(1), please notices that default map can not handle duplicated key, you must use multimap in that case.
4. priority\_queue, The priority queue, which is also called heap, only access largest element at O(1), can handle duplicated value.
5. Stack, last in, first out, can be used either change sequence, handle nested relation, keep increasing or decreasing order.
6. Queue, first in first out, not very useful.
7. Deque, you can push and pop on either side, so it can simulate both queue and stack. It also can be used to maintain a sliding window as two pointers.
8. List, to your surprise, this may be the most useful data structure in C++ for LeetCode design problem. It can simulate the deque behavior, given the iterator it can also access, insert and delete an element in O(1) time, (internally it is a doubly linked list in implementation), but you can not search the element by value. Because the list.begin() will give you the iterator for the first element, so normally you push the new item form front not the back.

## 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;

}

};

## 716. Max Stack

Easy

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;

}

};

## 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;

}

};

## 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;

}

};

## 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);

}

}

};

## 900. RLE Iterator

Medium

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.

### Analysis:

We keep index on which array and the offset in each array.