Leet Code Design

This paper is targeted for LeetCode Design Problems

# STD library

The standard library in C++ implement all the basic data structure, which provides significant convinence for you to write the algorithm in C++, so it is a must to be studied and remembered.

Basically, you should have the following include and name space defined in your header file.

#include <functional>

#include <algorithm>

#include <unordered\_map>

#include <unordered\_set>

#include <map>

#include <stack>

#include <set>

#include <queue>

#include <vector>

using namespace std;

In the following table I will explain them in detail.

|  |  |  |
| --- | --- | --- |
| Data Structure | Descriptions | Code example |
| vector<T> | An array which can hold any object type.  It can auto grow, you can refer to a item in the vector by index in O(1).  Because it is push\_back() and pop\_back(), you can also use vector as a stack. | vector<int> array;  array.push\_back(2);  array[2] = 3;  array.back();  array.size();  array.pop\_back();  if (array.empty()) {};  array.clear(); |
| list<T> | single list, you can push to both ends, travel by single direction iterator, forward or backward, you can also go to previous item. | list<int> l;  l.push\_front(25);  l.push\_back(13);  l.pop\_back();  l.pop\_front();  int v = l.front();  int v = l.back();  auto itr = l.begin();  l.erase(itr); |
| queue<T> | A standard queue, which allow you to push to the back and pop from front. FIFO. The items can be any object type as long as constructors are complete (think big 3).  Please make sure to check if the queue is empty before peeking the item (front()). | queue<int> myQueue;  myQueue.push(2);  int v = myQueue.front();  myQueue.pop();  myQueue.empty(); |
| stack<T> | A standard stack, which allow you to push to back and pop up from back, LIFO. It can hold any object.  Please make sure to check if the stack is empty before peeking the item (top()). | stack<int> myStack;  myStack.push(2);  myStack.top();  myStack.pop();  myStack.empty(); |
| unordered\_map<T1, T2> | A hashtable, which hold the key, value pair. The key must implement a default hash function. You can assume accessing by key is O(1), but do not expect the keys are in order. The value can be a simple type such as integer or string, it also can be a container such as a vector, set or another map.  To test if a key is in the hashtable you can use find() (return iterator which is end if not found) or count(key)(return 0 if not found);  Please notice access the map by a key which does not exist, can cause a new key insert with a default value. | unordered\_map<int, int> myMap;  myMap[3] = 2;  if (myMap.count(2) == 0) {};  myMap.erase(2);  myMap.empty();  myMap.find(2);  // the following will have a 0 returned if the key of 3 does not exist.  int v = myMap[3]; |
| unordered\_set<T> | A hashtable based set, duplication are removed.  Access the key is O(1). | unordered\_set<int> mySet;  mySet.insert(3);  if (mySet.count(2) == 0) {};  mySet.erase(2);  mySet.empty(); |
| map<T1, T2> | It is implemented as binary search tree, in most cases, it is self-balanced tree such as red-black tree.  The object type used as key must have a default comparator implemented.  You can assume that you can access any key by O(log(n)).  The other rules are same as unordered\_map.  Please notice for pair<int, int> no hash function is implemented, so it is not supported in unordered\_map, but you can have it in map because the comparator is implemented. | map<int, int> myMap;  myMap[3] = 2;  if (myMap.count(2) == 0) {};  myMap.erase(2);  myMap.empty();  myMap.find(2);  // the following will have a 0 returned if the key of 3 does not exist.  int v = myMap[3]; |
| set<T> | A binary search tree based set, duplication are removed.  Access the key is O(log(n)) | set<int> mySet;  mySet.insert(3);  if (mySet.count(2) == 0) {};  mySet.erase(2);  mySet.empty(); |
| deque<T> | A double direction queue which you can push or pop from both ends, front and back.  In C++, the deque has far more comprehensive methods such as insert, erase which access the item by any position, but for algorithm please ignore them first. | deque<int> myQueue;  myQueue.puh\_back(2);  myQueue.puh\_front(3);  myQueue.pop\_back();  myQueue.pop\_front();  int value = myQueue.back();  int value = myQueue.front();  myQueue.empty(); |
| priority\_queue<T> | A sorted queue which always has the maximum value on the top. You can push any value to the right position, and you can peek or pop the maximum value from queue.  Push to or Pop from the queue is O(log(n)), while peek the top value is O(1).  A priority\_queue is not de-duplicated, so it allows multiple items with same value in the queue.  A priority\_queue can have its customized comparator, so it can defined its own "highest priority value". However if you just want a smallest value at top you can either make the value as negative or have a default comparator such as great()). | priority\_queue<int> myQueue;  myQueue.push(2);  myQueue.pop();  myQueue.top();  myQueue.empty();  myQueue.size();  priority\_queue<int, vector<int>, greater<int>> m\_Large; |
| multiset<T1> | A binary search tree based set which allows duplicated value, please notice that erase a value will erase all the items with the specific value, so you must erase at iterator. | multiset<int> low\_half;  low\_half.insert(value);  low\_half.erase(low\_half.find(\*low\_half.rbegin())); |
| pair<T1, T2> | This is a key-value pair which is equivalent to a vector<T>(2); We normally use it to represent a interval, a position in a 2-D plane or a key value pair. | pair<int, int> interval;  interval.first = 3;  interval.end = 5;  int distance = interval.second – interval.first; |

# Design Problem

There is a group of problems marked as design problem, basically you are asked to implement some data structure which can efficiently fulfill the required operation. There is no very tricky algorithm for such problems, but you should design the best fit data structure for your algorithm.

To resolve such problems, you should be very familiar with the language built-in data structure, for example in C++ it is STL, for Java and C#, it should be Collections. The most common data structures are HashMap (Set), TreeMap (Tree), List, then PriorityQueue, MultiSet, Queue, Stack and Vector (ArrayList). In C++ you can also use pair, in Java you can use ArrayList.

The above data structures are all known as container, to iterate them you should use iterator, being familiar with the iterator for these containers is also important.

Sometimes in the LRU or LFU question, you are asked to store and fetch a top item in O(1) complexity, so you are not allowed to use TreeMap in this case, but you can use List as alternative, and remember the iterator in the list in the hashtable and move (which means erase and insert to front) the list item to top when it is touched.

/// <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 set.

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

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

/// set(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.

/// </summary>

class LRUCache

{

private:

size\_t m\_Capacity;

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

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 set(int key, int value)

{

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

{

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

{

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;

}

/// <summary>

/// Remove a key in the LRU cache.

/// </summary>

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

/// <returns>true, if found</returns>

bool remove(int key)

{

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

{

return false;

}

else

{

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

m\_List.erase(iterator);

m\_map.erase(key);

return true;

}

}

};

/// <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:

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

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

/// 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\_Stack;

public:

/// <summary>

/// Constructor an empty max stack

/// </summary>

MaxStack()

{

}

void push(int x)

{

// Add to stack stack, last one first

m\_Stack.push\_front(x);

// Add to value map, biggest first

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

}

int pop()

{

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

auto itr = m\_Stack.begin();

int value = \*itr;

m\_Stack.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\_Stack.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();

itr->second.pop();

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

m\_Stack.erase(list\_itr);

return value;

}

};

/// <summary>

/// Leet code #251. Flatten 2D Vector

///

/// Implement an iterator to flatten a 2d vector.

/// For example,

/// Given 2d vector =

/// [

/// [1,2],

/// [3],

/// [4,5,6]

/// ]

/// By calling next repeatedly until hasNext returns false, the order of

/// elements returned by next should be: [1,2,3,4,5,6].

/// Hint:

/// 1.How many variables do you need to keep track?

/// 2.Two variables is all you need. Try with x and y.

/// 3.Beware of empty rows. It could be the first few rows.

/// 4.To write correct code, think about the invariant to maintain. What is it?

/// 5.The invariant is x and y must always point to a valid point in the 2d

/// vector. Should you maintain your invariant ahead of time or right when

/// you need it?

/// 6.Not sure? Think about how you would implement hasNext(). Which is more

/// complex?

/// 7.Common logic in two different places should be refactored into a common

/// method.

///

/// Follow up:

/// As an added challenge, try to code it using only iterators in C++ or

/// iterators in Java.

/// </summary>

class Vector2D

{

private:

size\_t m\_Row;

size\_t m\_Col;

vector<vector<int>> m\_Vector;

public:

Vector2D(vector<vector<int>>& vec2d)

{

m\_Vector = vec2d;

m\_Row = 0;

m\_Col = 0;

}

int next()

{

int result = m\_Vector[m\_Row][m\_Col];

m\_Col++;

return result;

}

bool hasNext()

{

while ((m\_Row < m\_Vector.size()) && (m\_Col >= m\_Vector[m\_Row].size()))

{

m\_Row++;

m\_Col = 0;

}

if (m\_Row == m\_Vector.size()) return false;

else return true;

}

};

/// <summary>

/// Leet code #497. Random Point in Non-overlapping Rectangles

///

/// Given a list of non-overlapping axis-aligned rectangles rects, write

/// a function pick which randomly and uniformily picks an integer point

/// in the space covered by the rectangles.

///

/// Note:

///

/// An integer point is a point that has integer coordinates.

/// A point on the perimeter of a rectangle is included in the space

/// covered by the rectangles.

/// ith rectangle = rects[i] = [x1,y1,x2,y2], where [x1, y1] are the

/// integer coordinates of the bottom-left corner, and [x2, y2] are the

/// integer coordinates of the top-right corner.

/// length and width of each rectangle does not exceed 2000.

/// 1 <= rects.length <= 100

/// pick return a point as an array of integer coordinates [p\_x, p\_y]

/// pick is called at most 10000 times.

/// Example 1:

///

/// Input:

/// ["Solution","pick","pick","pick"]

/// [[[[1,1,5,5]]],[],[],[]]

/// Output:

/// [null,[4,1],[4,1],[3,3]]

/// Example 2:

///

/// Input:

/// ["Solution","pick","pick","pick","pick","pick"]

/// [[[[-2,-2,-1,-1],[1,0,3,0]]],[],[],[],[],[]]

/// Output:

/// [null,[-1,-2],[2,0],[-2,-1],[3,0],[-2,-2]]

/// Explanation of Input Syntax:

///

/// The input is two lists: the subroutines called and their arguments.

/// Solution's constructor has one argument, the array of rectangles

/// rects. pick has no arguments. Arguments are always wrapped with a

/// list, even if there aren't any.

/// </summary>

class RandomRectanglePoint

{

private:

vector<vector<int>> m\_rectangles;

vector<int> m\_sum;

public:

RandomRectanglePoint(vector<vector<int>> rects)

{

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

{

m\_rectangles.push\_back(rects[i]);

int area = (rects[i][2] - rects[i][0] + 1) \*

(rects[i][3] - rects[i][1] + 1);

if (m\_sum.empty())

{

// we start from 0, so it is a point

m\_sum.push\_back(area);

}

else

{

m\_sum.push\_back(m\_sum.back() + area);

}

}

srand((unsigned int)time(0));

}

vector<int> pick()

{

vector<int> result = { 0, 0 };

if (m\_sum.empty()) return result;

int rand\_num = rand() % m\_sum.back() + 1;

int index = lower\_bound(m\_sum.begin(), m\_sum.end(), rand\_num)

- m\_sum.begin();

int remaining = (index == 0 ? rand\_num: rand\_num - m\_sum[index-1]);

int x = m\_rectangles[index][0] + (remaining - 1) %

(m\_rectangles[index][2] - m\_rectangles[index][0] + 1);

int y = m\_rectangles[index][1] + (remaining - 1) /

(m\_rectangles[index][2] - m\_rectangles[index][0] + 1);

result = { x, y };

return result;

}

};

/// <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);

}

}

};