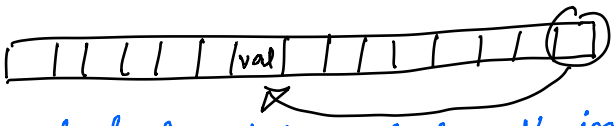
Design Data Structure

Insert, Delete, GetRandom :-

Data Structure: vector<int> arr.
and
unordered map<int, int> indexmap
{val, index}

getRandom(): arr[rand() % (arr.size())]

delete(val):



move last element to val element's index
and update last element's index in indexMap.

insert(val): arr.pushback(val)
mp[val] = arr.size() - 1;

*If we want index in the actual array;
unordered map <int, pair<int, int>> mp.
val → actual, imaginary index

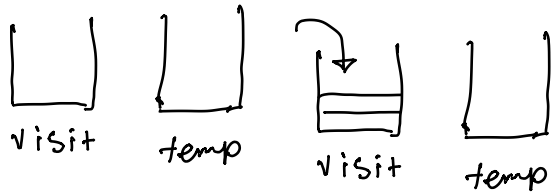
✓ now to have imaginary Indexs;
imIdx = arr.size() + deleteCount; → this integer variable increments at deletion.

Browser History

functions: visit(url) forward() backward()

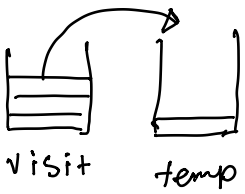
use 2 stacks

visit



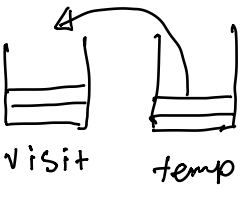
visit.push
empty the temp

backward



Store previous
to temp stack

Forward



restore in
visit

at the end of each function visit.top() is the visible website.

Design Underground System

void checkin( ), checkout( ) param: station-name, time

when a passenger enters a station and exits another station

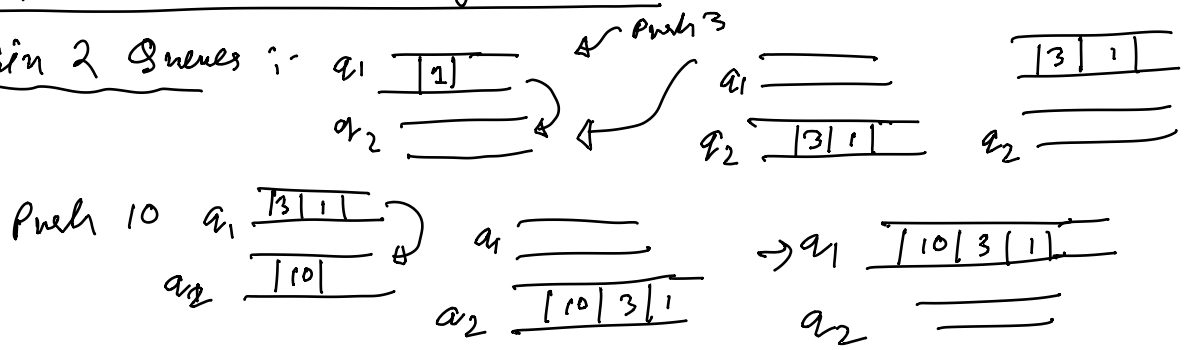
int getAverageTime(stationA, stationB)

Data Structures :- unordered_map < String , pair < int, int >>

stationA + "_" + stationB Total time of all passenger no. of passengers

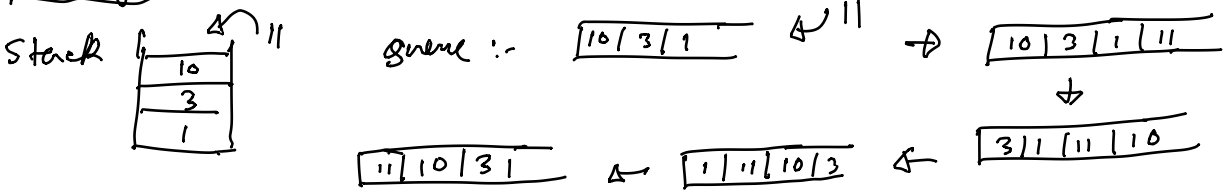
Implement Stack Using Queue :-

Using 2 Queues :-



* at every step have q_1 as in stack formation.

Using 1 Queue :-



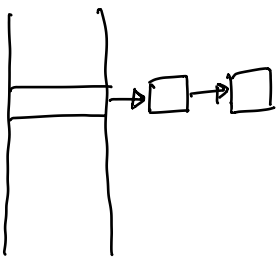
while (n > 1)
{ temp = q.front q.pop q.push(temp) n-- }

Design Hash Set / Hash Table :-

Hash table = Arr of size m

h(key) = key % m

for collision use List / vector



#C++STL

#**&&**

#LeetCode/Hard

LRU Cache (Least Recently Used)

makeLRU(size) put(key, value) get(key);

(least recently used key value pair gets deleted)

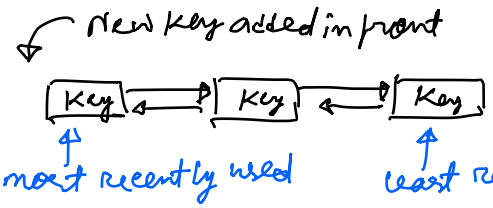
Data Structure :- List<int> dll. (For storing order of key)

unordered map < int , pair < list<int>::iterator , int >>

int → Key

list<int>::iterator → address of the key node in dLL

int → value of the key



map

key	addr, value
key 1	{ addr , value...

✓ To make a node the first node in the dLL

```
void makeRecent(int key) {
    dll.erase(mp[key].first)
    dll.push_front(key)
    mp[key].first = dll.begin()
}
```

LFU Cache

✓ least frequently used
✓ if there's a tie use LRU

Data Structure:-

① map < int , list > freq

ordered list such that least frequency is at the top.

int : key

list : list of {key, value, frequency}

↳ maybe this can get eliminated

② unordered_map < int , address of list node > mp

freq

no. of occurrence	List (dLL)
1	1,1,1
2	3,4,2
3	

Most Recent ← → Least Recent.

※ int & nodeValue = (* (mp[key]))

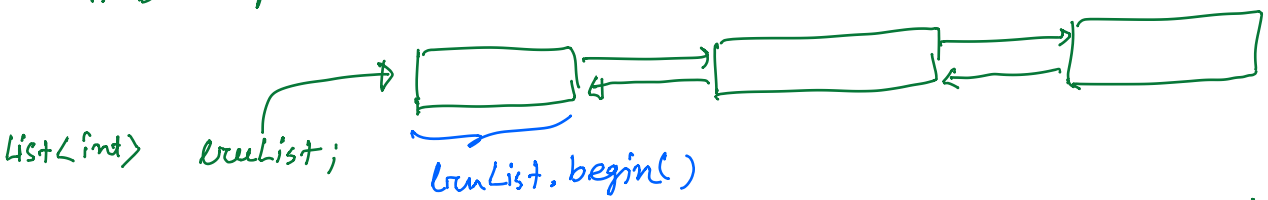
mp[key] = address of key's List node.

nodeValue[0] Stores key } redundant

[1] " value

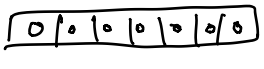
[2] " frequency.

* Both for LRU & FRU;



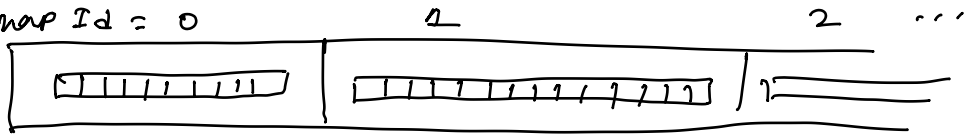
✓ accessing the last node is difficult hence node for recently accessed key,val pair gets appended at the front of LRU (in O(1)) and address of that is lruList.begin().

Snapshot Array :-

snap id = 0
all the array is 0 

at future point the array evolves and sometime it's asked to take a snapshot of the array at a specific snap id.

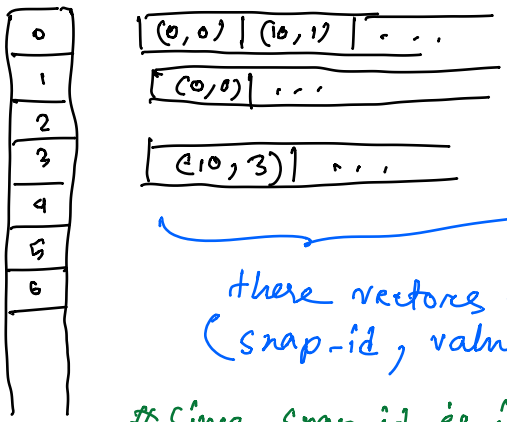
get(index, snap_id) .

Brute force; 

just store all the array's as is at a perticular snap.

↳ This will give Memory Limit Exceeded.

Optimal ;

The array → 

there vectors are storing (snap_id, value at that index)

#algorithm/binary search

Since snap_id is increasing these arrays can be exploited with Binary Search.

Data Structure :

vector < vector< pair<int,int>>> arr(n, {{0,0}});

↳ initiated with 0,0 i.e. at beginning arr was all 0.