LRU CACHE

BRUTE FORCE:

Use array and implement in naive way.

Optimized Approach :

We are given capacity of cache and cannot store elements more than that , two function needs to be implemented get and put. get( ) basically checks whether element is present or not if it is present we simply return the element and as this element was accessed recently it becomes the most recently used so we need to update it to most recently used. put( ) checks whether the key is present or not if it is present we simply update the value and again make it the most recently used one, if it is not present it adds that element.

To implement such a structure we need some data structure where the insertion and deletion is possible in O(1) as the most optimized approach for this problem is O(1). The data structure to be used is Doubly linked list.

We basically maintain the LRU elements at the tail and most recently used at the head. If the cache is full then we remove the element from the tail and add the new element at head. Similarly when an element is already present and is recently accessed we need to search the element remove it from its current position and push it at the head of the double linked list.

For making searching efficient we will make use of unordered map where the average time complexity is O(1) considering there are minimal collisions.

Map will store the key and the address to the node that stores the key.

A class Node , and LRU cache class will be used. LRU cache will store the whole linked list where we maintain two pointers head and tail to avoid writing NULL conditions and node class for creating nodes in the linked list. head→next=tail and tail→prev=head.

Two additional functions will be required addNode and deleteNode.

addNode - adds a node at the head of the linked list.

deleteNode - deletes a node by searching the node from the map.

If a node is to be deleted its entry from the map also needs to be deleted and if a node is added its entry into map is also necessary.

Code :

```
#include <bits/stdc++.h>
class Node
{
    public:
    int key,val;
    Node* next;
    Node* prev;
    Node(int key,int val)
    {
        this->key=key;
        this->val=val;
        prev=NULL;
        next=NULL;
    }
};
class LRUCache
{
public:
    Node* head=new Node(-1,-1);
    Node* tail=new Node(-1,-1);
    unordered_map<int,Node*> m;
    int cap;
    LRUCache(int capacity)
    {
        // Write your code here
        cap=capacity;
        head->next=tail;
        tail->prev=head;
    }
    void addNode(Node* newNode)
    {
        Node* temp=head->next;
        newNode->prev=head;
        newNode->next=temp;
        temp->prev=newNode;
        head->next=newNode;
    }
    void deleteNode(Node* delNode)
    {
        Node* delprev=delNode->prev;
        Node* delnext=delNode->next;
        delprev->next=delnext;
        delnext->prev=delprev;
    }
    int get(int key)
    {
        // Write your code here
        if(m.find(key)!=m.end())
        {
            Node *found=m[key];
            int res=found->val;
            deleteNode(found);
            addNode(found);
            return res;
        }
        return -1;
    }

    void put(int key, int value)
    {
        // Write your code here
        if(m.find(key)!=m.end())
        {
            Node *found=m[key];
            m.erase(key);
            deleteNode(found);
        }
        if(cap==m.size())
        {
            m.erase(tail->prev->key);
            deleteNode(tail->prev);
        }
        addNode(new Node(key,value));
        m[key]=head->next;
    }
};
```

- Time Complexity : O(1)
- Space Complexity : O(2*capacity)