**LRU Cache Implementation**

#include <unordered\_map>

class LRUCache {

public:

class Node {

public:

int key;

int val;

Node\* next;

Node\* prev;

Node(int key, int val) {

this->key = key;

this->val = val;

}

};

Node\* head = new Node(-1, -1);

Node\* tail = new Node(-1, -1);

int capacity;

unordered\_map<int, Node\*> cache;

LRUCache(int capacity) {

this->capacity = capacity;

head->next = tail;

tail->prev = head;

}

void deleteNode(Node\* node) {

Node\* prevNode = node->prev;

Node\* nextNode = node->next;

prevNode->next = nextNode;

nextNode->prev = prevNode;

}

void addNode(Node\* node) {

node->next = head->next;

node->next->prev = node;

node->prev = head;

head->next = node;

}

int get(int key) {

if (cache.find(key) == cache.end()) {

return -1;

}

Node\* node = cache[key];

int value = node->val;

cache.erase(key);

deleteNode(node);

addNode(node);

cache[key] = head->next;

return value;

}

void put(int key, int value) {

if (cache.find(key) != cache.end()) {

deleteNode(cache[key]);

cache.erase(key);

}

if (cache.size() == capacity) {

cache.erase(tail->prev->key);

deleteNode(tail->prev);

}

Node\* newNode = new Node(key, value);

addNode(newNode);

cache[key] = head->next;

}

};