**SOLUTION**

class LRUCache {

int capacity;

list< pair<int,int> > lruList; // key, value pair

unordered\_map< int, list<pair<int,int>>::iterator > hash; // key to <key,val> iterator map

void moveToFront( int key, int value ) {

// erase and add a new entry to front

lruList.erase( hash[key] ); // this is O(1) since we are using iterator

lruList.push\_front( make\_pair(key, value) );

hash[key] = lruList.begin();

}

public:

LRUCache( int capacity ) {

this->capacity = capacity;

}

int get( int key ) {

if( hash.find(key) == hash.end() )

return -1;

// move the key, value pair to front

int value = (\*hash[key]).second;

moveToFront( key, value );

return (\*hash[key]).second;

}

void put( int key, int value ) {

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

// when key is already in hash

moveToFront( key, value );

} else {

// add to the cache

lruList.push\_front( make\_pair( key, value ) );

hash[key] = lruList.begin();

if( hash.size() > capacity ) {

// erase

hash.erase( lruList.back().first );

lruList.pop\_back();

}

}

}

};

**get()- O(1)**

**put()- O(1)**