Abstract Classes - Polymorphism

Abstract base classes in C++ can only be used as base classes. Thus, they are allowed to have virtual member functions without definitions.

A cache is a component that stores data so future requests for that data can be served faster. The data stored in a cache might be the results of an earlier computation, or the duplicates of data stored elsewhere. A cache hit occurs when the requested data can be found in a cache, while a cache miss occurs when it cannot. Cache hits are served by reading data from the cache which is faster than recomputing a result or reading from a slower data store. Thus, the more requests that can be served from the cache, the faster the system performs.

One of the popular cache replacement policies is: "least recently used" (LRU). It discards the least recently used items first.

For example, if a cache with a capacity to store 5 keys has the following state(arranged from most recently used key to least recently used key) -

```
5 3 2 1 4
```

Now, If the next key comes as 1(which is a cache hit), then the cache state in the same order will be -

```
1 5 3 2 4
```

Now, If the next key comes as 6(which is a cache miss), then the cache state in the same order will be -

```
6 1 5 3 2
```

You can observe that 4 has been discarded because it was the least recently used key and since the capacity of cache is 5, it could not be retained in the cache any longer.

Given an abstract base class *Cache* with member variables and functions:

mp - Map the key to the node in the linked list
cp - Capacity
tail - Double linked list tail pointer
head - Double linked list head pointer
set() - Set/insert the value off the key, if present, otherwise add the key as the most recently used key. If the cache has reached its capacity, it should replace the least recently used key with a new key.
get() - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1. Then make that key as the most recently used key.

You have to write a class *LRUCache* which extends the class *Cache* and uses the member functions and variables to implement an LRU cache.

Input Format

First line of input will contain the N number of lines containing get or set commands followed by the capacity M of the cache.
The following N lines can either contain get or set commands.

An input line starting with get will be followed by a key to be found in the cache. An input line starting with set will be followed by the key and $value$ respectively to be inserted/replaced in the cache.

Constraints

$1 <= N <= 500000$
$1 <= M <= 1000$
$1 <= key <= 20$
$1 <= value <= 2000$

Output Format

The code provided in the editor will use your derived class *LRUCache* to output the value whenever a get command is encountered.

Sample Input

```
3 1
set 1 2
get 1
get 2
```

Sample Output

```
2
-1
```

Explanation

Since, the capacity of the cache is 1, the first *set* results in setting up the key 1 with it's value 2. The first *get* results in a cache hit of key 1, so 2 is printed as the value for the first *get*. The second *get* is a cache miss, so -1 is printed.