How do you like your Mocha?

Problem Description

Your friend, Juan, is madly in love and needs your help. He is courting Mocha, one of the more famous dancers from your school. Unable to do simple research, Mocha asked Juan to do a take-home machine problem for her. Your broski needs you. You have to help Juan!

The problem is about implementing a **Least Recently Used (LRU) caching algorithm** through a hash map. *Caching* is the process of storing data in a temporary storage area called *cache*. When the *cache* is full, caching algorithms must choose which items to remove to make room for new ones. These algorithms allow the applications we develop to be blazing fast!

Least Recently Used (LRU) algorithm discards the least recently used entry if a new entry is about to be added when the cache is already full. For a hash map, an entry is considered *'used'* when `get` and `put` operations are applied to it.

For example, given a map called `my_cache` with a capacity of `3`, if we `put` the following entries sequentially:

```
my_cache[1] = 1234;
my_cache[3] = 300;
my_cache[9] = 9088;
my_cache[14] = 2184;
```

By the time entry `14` is about to be `put`, `my_cache` is already full. LRU algorithm decides that entry `1` is the *least recently used* since among all the elements, it is the 'oldest' entry that was applied with either `put` or `get`. Thus, entry `1` is discarded, leaving `my_cache` with the following elements:

3	9	14
300	9088	2184

Entry `3` here will be the *LRU* entry. If we apply a `get` operation on entry `3` and another `put` operation on entry `9`:

```
my_cache[3];
my_cache[9] = 787898;
```

Then, entry `3` will not be the *LRU* anymore. Entry `9`'s value will also be updated and it will become the *most* recently used entry. This leaves entry `14` as the *LRU* item.

14 (LRU)	3	9
2184	300	787898

So, putting another new item will remove entry `14` from `my_cache`.

```
my_cache[24] = 143;
```

3	9	24
300	787898	143

Seems straightforward, huh?

Input Format

A problem case starts with a positive integer *$0 < c <= 2000$*, the capacity of the map. This is followed in a separate line by *$0 < p < 16384$*, the number of map operations. This is then followed by *p* lines denoting an operation, each of which can be one of the following:

- `put <k> <v>` - put an entry with *k* as key and *v* as value. Both `k` and `v` are positive integers <= *2,147,483,647*.
- `get <k>` - return the value *v* that corresponds to *k*. Assume that *k* is present in the map.
- `print` - print a list of all *k* in the map

Each problem case is separated by a blank line.

Output Format

The output for each case begins with a line containing the word `CASE` followed by a single space, the case number, and a colon (:). This is followed by the output for the operations performed.

Each operation, **except** `put`, should yield an output on a new line, defined by the following:

- `=<v>` - for every `get <k>` operation, where *v* is the value that corresponds to *k*
- `[k1,k2,k3...,kn]` - for every `print`, enumerate the *k* of each entry in the map, from least recently used to most recently used, each separated by a comma. The entire list is enclosed by square brackets `[]`.

Sample Input

```
3
8
put 1 1234
put 3 300
put 9 9088
put 14 2184
get 3
put 9 787898
put 24 143
print

2
5
put 1993 24
put 6 8
print
get 1993
print
```

Sample Output

```
CASE 1:
=300
[3,9,24]

CASE 2:
[1993,6]
=24
[6,1993]
```