Problem ID: deadlock
Avoiding Deadlock

The ecosystem of a computer typically consists of files and processes. *Files* allow for temporary/permanent data storage and *processes* encapsulate functionality related to executing compiled code. In order for operating systems to make it seem like many processes are running simultaneously a technique called *time slicing* is used. A simplistic view of time slicing says that every 20 milliseconds a new process is given 100 milliseconds of time on the CPU (that is, a new process gets to run for 100 milliseconds every 20 milliseconds). Processes on most systems run independently from one another and are therefore not allowed to peek into each other's address space (where program data such as variables are stored). Sometimes, however, it is important for processes to work together to fulfill a common goal and this is where files become very useful.

If process A wants to send the message "Hello World" to process B, A just has to open a file and place the message "Hello World" inside of it. Then process B can simply read the message, clear the file, and wait for another message to be written! Unfortunately there is a huge problem with this method of inter-process communication: when does process B know that process A has finished writing to the file? If process A can only write "Hello W" to the file during its 100 millisecond time slice, then process B won't get the full message! This is where the concept of a *lock* comes into play. A lock is essentially a data structure that supports two operations: $lock(L)$ and $unlock(L)$.

Suppose we have some lock named L. If we call $lock(L)$ for the first time, the process that made this call *acquires* the lock. A call to $unlock(L)$ by the same process will *free* the lock. If a call such as $lock(L)$ is made while L is already acquired, then the process that made the call will wait until the lock is freed. With locks we can now pass the "Hello World" message safely.

Not all processes that use locks work correctly. Suppose there is a process A that has lock L_1 and process B that has lock L_2; after acquiring those locks, the processes then try and acquire the others. At this point, neither process can proceed because they're waiting on locks that can't ever be released! This situation is known as *deadlock*. Deadlock can occur with more than two processes; more formally, deadlock occurs when there is a set of waiting processes $\{P_1, P_2, ..., P_n\}$ such that P_1 is waiting for a lock held by P_2, P_2 is waiting for a lock held by P_3 and so on until P_n is waiting for a lock held by P_1.

Given a set of processes, the locks they're currently holding, and the locks they're waiting on, can you determine if a deadlock is occurring?

Input

The input will begin with a line containing a single positive integer, t, representing the number of test cases to process. Each test case will begin with two space-separated integers N and M ($1 \leq N, M \leq 1,000$): the number of processes running and the number of locks available. Following will be a line containing an integer K. The next K lines will be either of the form "L i j" or "H i j" ($1 \leq i \leq N$, $1 \leq j \leq M$). "L i j" says that process P_i is waiting on lock L_j and "H i j" says that process P_i has lock L_j. A process can have multiple locks but can only wait on a single lock. A process will never wait on a lock that it owns. A lock can only be owned by a single process.

Output

For each test case print "Yes" if the processes are in a deadlock, otherwise "No" (on its own line).

Sample Input	Sample Output
2 3 3 6 H 1 1 H 2 2 H 3 3 L 1 2 L 2 3 L 3 1 3 2 4 H 1 1 H 2 2 L 3 2 L 1 2	Yes No