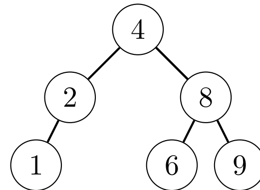
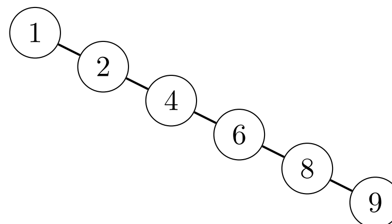
Tao of Trees (treaps)

Input file: `standard input`
Output file: `standard output`
Time limit: 1 second
Memory limit: 256 megabytes

As awesome as vanilla binary trees are, they do have that tiny-bitsy problem with heights. You see, this is what we would like our trees to look like, thick and short:



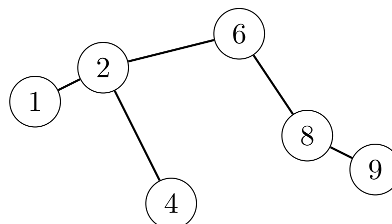
However, for that tree to appear, we need to add elements in the right order. If instead the elements were added in the wrong order, say, in increasing order, we would have obtained this tree:



And this is bad! We did not care before, because we were just asked to print the tree. But we normally want binary search trees to, well, search inside them. For that purpose, thick and short is much better than thin and tall. In the first tree, all nodes are 2 steps away from the root; in the second tree some nodes are much further. And the difference will only become larger: a full, thick tree of height n holds $2^{n+1} - 1$ nodes; a skinny tree of height n can only hold $n + 1$ elements.

The standard solution to this problem (AVL trees, red-black trees) is to do some operations (rotations) that will keep the tree more or less balanced as it grows. Instead, we will teach a different, slightly slower but flexible approach, called treaps.

A treap is a binary search tree where, instead of adding nodes at the leaves, we add nodes into given heights, and we respect those heights no matter the order in which nodes are added. For instance, consider that we add nodes with keys $(1, 2, 4, 6, 8, 9)$ at heights $(23, 19, 95, 13, 25, 71)$. Here height has to be understood not as the exact distance to the root, but as a relative positioning between nodes: a child must always have larger height than a parent. With this additional restriction, you may want to verify that there is only way to represent that treap:

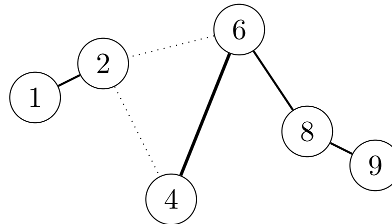


Indeed, 6 is the node with lowest height (13), so it must be the root. This splits all remaining nodes into those with smaller keys, which must appear at the left child, and those with larger keys, right child.

Why are treaps useful? Because if whenever we need to add a new node with key k into the treap, we add it into a random height, it is as if we are placing the node somewhere at random. The resulting tree will very rarely be thin and long. Treaps are, with very high probability, thick and short, no matter in what order the data arrives!

We still need to solve the problem of how to add or remove elements into the treap. For this, we will introduce two treap operations, `SPLIT` T k and `JOIN` T T'.

The operation `SPLIT` T k breaks a treap T into two treaps L and R such that all the nodes with key less or equal than k are on L, and all the nodes with key greater than k are on R, and the same height restrictions are conserved. For instance, if we apply `SPLIT` T 2 to the given example T, it would result in the following two trees L (of root 2) and R (of root 6).



Notie that this operation is far from trivial! The node with key 4 was originally a child of 2, but the splitting force it to change parents and become a child of 6.

The operation `JOIN` T T' receives two treaps such that all the nodes of T have smaller keys than all the nodes of T', and joins them. There is only one way of doing this, so joining the previous two trees should result in T again.

In this problem we ask you to do exactly the same as the previous problem, except that now every node will contain two values: the key k, and a height h. To avoid complications, we guarantee that all keys and heights will be unique.

- `PRINT`: Print the tree recursively: empty string if empty, `k/h` if it is a childless node of key k and height h, and `k/h(L,R)` otherwise, where `L` and `R` are its left and right children.
- `ADD` k h: Add a node K with key k at height h into the current treap T as follows. Execute $(L, R) :=$ `SPLIT` T k onto the current treap T. Then, create the node K, and execute $T :=$ `JOIN` L `(JOIN` K R`)`.
- `DEL` k: Delete the node with key k from the current treap T as follows. First split T twice: $(M, R) :=$ `SPLIT` T k and $(L, K) :=$ `SPLIT` M $k-1$. Then, create a new treap by joining L and R, executing $T :=$ `JOIN` L R. Safely discard the single node treap K.

To make things easier, we will never give you repeated heights or keys, or ask you to add keys that already exist or remove keys that do not exist.

Hint. Splitting and joining trees might look like very complicated operations at first sight. But if you find yourself writing lots of code, stop: they can actually be implemented in very few lines of code, with a single recursive pass into the tree.

Input

Starting with an empty tree, execute a sequence of orders, as described. Integers k will be from 0 to 10^5, heights h will be from 0 to 10^9.

At all times the nodes in the tree will not have repeated keys or heights. We will not ask you to remove nodes that do not exist.

Output

For each command `PRINT`, output a line with the contents of the tree at that time.

Examples

standard input	standard output
ADD 1 23 PRINT ADD 2 19 PRINT ADD 4 95 ADD 6 13 ADD 8 25 ADD 9 71 PRINT DEL 6 PRINT ADD 6 13 PRINT	1/23 2/19(1/23,) 6/13(2/19(1/23,4/95),8/25(,9/71)) 2/19(1/23,8/25(4/95,9/71)) 6/13(2/19(1/23,4/95),8/25(,9/71))
ADD 1 1 ADD 2 2 ADD 3 3 ADD 4 4 ADD 5 5 PRINT DEL 3 DEL 2 DEL 4 DEL 1 DEL 5 PRINT ADD 1 74 ADD 2 23 ADD 3 91 ADD 4 43 ADD 5 17 PRINT	1/1(,2/2(,3/3(,4/4(,5/5)))) 5/17(2/23(1/74,4/43(3/91,)),)