Data Structures Assignment-3

Q1. Vladimir worked for years making matrioshkas, those nesting dolls that certainly represent truly Russian craft. A matrioshka is a doll that may be opened in two halves, so that one finds another doll inside. Then this doll may be opened to find another one inside it. This can be repeated several times, till a final doll -that cannot be opened- is reached.

Recently, Vladimir realized that the idea of nesting dolls might be generalized to nesting toys. Indeed, he has designed toys that contain toys but in a more general sense. One of these toys may be opened in two halves and it may have more than one toy inside it. That is the new feature that Vladimir wants to introduce in his new line of toys.

Vladimir has developed a notation to describe how nesting toys should be constructed. A toy is represented with a positive integer, according to its size. More precisely: if when opening the toy represented by m we find the toys represented by n1, n2, ..., nr, it must be true that n1 + n2 + ... + nr < m. And if this is the case, we say that toy m contains directly the toys n1, n2, ..., nr . It should be clear that toys that may be contained in any of the toys n1, n2, ..., nr are not considered as directly contained in the toy m.

A generalized matrioshka is denoted with a non-empty sequence of non zero integers of the form:

a1 a2 ... aN
such that toy k is represented in the sequence with two integers - k and k, with the negative one occurring in the sequence first that the positive one.
For example, the sequence

-9 -7 -2 2 -3 -2 -1 1 2 3 7 9
represents a generalized matrioshka conformed by six toys, namely, 1, 2 (twice), 3, 7 and 9. Note that toy 7 contains directly toys 2 and 3. Note that the first copy of toy 2 occurs left from the second one and that the second copy contains directly a toy 1. It would be wrong to understand that the first -2 and the last 2 should be paired.
On the other hand, the following sequences do not describe generalized matrioshkas:

-9 -7 -2 2 -3 -1 -2 2 1 3 7 9
because toy 2 is bigger than toy 1 and cannot be allocated inside it.
-9 -7 -2 2 -3 -2 -1 1 2 3 7 -2 2 9
because 7 and 2 may not be allocated together inside 9.
-9 -7 -2 2 -3 -1 -2 3 2 1 7 9
because there is a nesting problem within toy 3.
Your problem is to write a program to help Vladimir telling good designs from bad ones.

Input

The input file contains several test cases, each one of them in a separate line. Each test case is a sequence of non zero integers, each one with an absolute value less than 107.

Output

Output texts for each input case are presented in the same order that input is read.

For each test case the answer must be a line of the form

:-) Matrioshka!

if the design describes a generalized matrioshka. In other case, the answer should be of the form

:-(Try again.

Sample Input

```
-9 -7 -2 2 -3 -2 -1 1 2 3 7 9
-9 -7 -2 2 -3 -1 -2 2 1 3 7 9
-9 -7 -2 2 -3 -1 -2 3 2 1 7 9
-100 -50 -6 6 50 100
-100 -50 -6 6 45 100
-10 -5 -2 2 5 -4 -3 3 4 10
-9 -5 -2 2 5 -4 -3 3 4 9
```

Sample Output

```
:-) Matrioshka!
:-( Try again.
:-( Try again.
:-) Matrioshka!
:-( Try again.
:-) Matrioshka!
:-( Try again.
```

Q2. Family Tree

A family tree is a chart representing family relationships in a conventional tree structure. The more detailed family trees used in medicine, genealogy, and social work are known as genograms. Several genealogical numbering systems have been widely adopted for presenting family trees. Some of them are ascending numbering systems.

Ahnentafel, also known as the Eytzinger Method, Sosa Method, and Sosa-Stradonitz Method, allows for the numbering of ancestors beginning with a descendant. This system allows one to derive an ancestor's number without compiling the list, and allows one to derive an ancestor's relationship based on their numbers.

The number of a person's father is the double of his own number, and the number of a person's mother is the double of his own plus one. For instance, if the number of John Smith is 10, his father is 20, and his mother is 21.

The first 15 numbers, identifying individuals in 4 generations, are as follows:

```
(First Generation)
 1  Subject

(Second Generation)
 2  Father
 3  Mother

(Third Generation)
 4  Father's father
 5  Father's mother
 6  Mother's father
 7  Mother's mother
```

(Fourth Generation)
 8 Father's father's father
 9 Father's father's mother
10 Father's mother's father
11 Father's mother's mother
12 Mother's father's father
13 Mother's father's mother
14 Mother's mother's father
15 Mother's mother's mother

The Problem

We have built our complete family tree, including until the N-th generation. We want to find out the first common descendant of two given people in the family tree.

The Input

The first line of the input contains an integer T indicating the number of test cases. For each test case, there is a line with three positive integer numbers N, A and B, separated by blanks, (4<=N<=20), where N is the number of generations, and A and B are the persons for which we want to find the first common descendant.
The Output

For each test case, the output should consist of one integer indicating the first common descendant of A and B. In case A(or B) is descendant of B(or A), print the integer indicating the descendant.(See test case 3 for this)

Sample Input
3
4 4 12
4 8 11
5 7 31

Sample Output
1
2
7

Q3. Cuckoo Hashing

One of the most fundamental data structure problems is the dictionary problem: given a set D of words you want to be able to quickly determine if any given query string q is present in the dictionary D or not. Hashing is a well-known solution for the problem. The idea is to create a function h :∑* -> [0..n - 1] from all strings to the integer range 0, 1,..., n - 1 , i.e. you describe a fast deterministic program which takes a string as input and outputs an integer between 0 and n - 1 . Next you allocate an empty hash table T of size n and for each word w in D , you set T[h(w)] = w . Thus, given a query string q , you only need to calculate h(q) and see if T[h(q)] equals q , to determine if q is in the dictionary. Seems simple enough, but aren't we forgetting something? Of course, what if two words in D map to the same location in the table? This phenomenon, called collision, happens fairly often (remember the Birthday paradox: in a class of 24 pupils there is more than 50% chance that two of them share birthday). On average you will only be able to put roughly √n-sized dictionaries into the table without getting collisions, quite poor space usage!

A stronger variant is Cuckoo Hashing (Cuckoo Hashing was suggested by the danes R. Pagh and F. F. Rödler in 2001). The idea is to use two hash functions h1 and h2 . Thus each string maps to two positions in the table. A query string q is

now handled as follows: you compute both h1(q) and h2(q) , and if T[h1(q)] = q , or T[h2(q)] = q , you conclude that q is in D . The name ``Cuckoo Hashing" stems from the process of creating the table. Initially you have an empty table. You iterate over the words d in D , and insert them one by one. If T[h1(d)] is free, you set T[h1(d)] = d . Otherwise if T[h2(d)] is free, you set T[h2(d)] = d . If both are occupied however, just like the cuckoo with other birds' eggs, you evict the word r in T[h1(d)] and set T[h1(d)] = d . Next you put r back into the table in its alternative place (and if that entry was already occupied you evict that word and move it to its alternative place, and so on). Of course, we may end up in an infinite loop here, in which case we need to rebuild the table with other choices of hash functions. The good news is that this will not happen with great probability even if D contains up to n/2 words!

Input

On the first line of input is a single positive integer 1<=t<=50 specifying the number of test cases to follow. Each test case begins with two positive integers 1<=m<=n<=10000 on a line of itself, m telling the number of words in the dictionary and n the size of the hash table in the test case. Next follow m lines of which the i :th describes the i :th word di in the dictionary D by two non-negative integers h1(di) and h2(di) less than n giving the two hash function values of the word di . The two values may be identical.

Output

For each test case there should be exactly one line of output either containing the string ``successful hashing" if it is possible to insert all words in the given order into the table, or the string ``rehash necessary" if it is impossible.

Sample Input

```
2
3 3
0 1
1 2
2 0
5 6
2 3
3 1
1 2
5 1
2 5
```

Sample Output

```
successful hashing
rehash necessary
```

Question 4)

Write a program to perform the following operations on AVL-trees: a) Insertion. b) Deletion.

Input:

On the first line of the input is a single positive integer 1<=t<=50 specifying the number of test cases to follow. Each test case has a positive integer n, which denotes the number of queries to follow.

For any integer k, each query is in one of the following formats:

1. i k insert number k into avl tree.
2. d k delete number k from avl tree, if it is present.

3. p print the preorder traversal of the tree formed till then.(print a blank line,if tree is empty)

Also, all the elements to be inserted in the tree are made sure to be unique. (for 2 different insert queries i k1, i k2, k1 != k2)

Output:

For each print query (p), print the preorder traversal of the tree formed till then. Print a blank line,if tree is empty.

Input Format:

```
2
7
i 10
i 20
i 30
i 40
i 50
i 25
p
13
p
i 9
i 1
i 0
i -1
i 5
i 2
i 6
i 10
i 11
p
d 10
p
```

Output Format:

```
30 20 10 25 40 50



9 1 0 -1 5 2 6 10 11

1 0 -1 9 5 2 6 11
```

Question 5)

Given n numbers, construct the binary search tree by inserting them in the given order.
Assume that all the nodes of the tree have unique integer data. i.e. no two node contains same data.

Also following two integers are given to you.
1. myNode: An integer value which is present in the given tree.
2. k : An integer

Now you have to find whether an immediate right node exists for myNode. Lets call this right node IRNode.
If it exists, print the value of IRNode. If it doesn't, print $.

Also find the kth minimum element (kMinElement) for the sub-tree which is hanging from IRNode. [IRNode is the root node for this sub-tree]
Print kMinElement.
Print $ in place of kMinElement in following cases:
case 1: IRNode doesn't exist.
case 2: kMinElement doesn't exist. [for example there are less than k elements in the subtree under consideration]

Input format:
```
3
5
1 2 3 4 5
3 2
8
34 67 12 36 90 23 56 57
12 3
6
34 67 12 36 90 23
12 20
```

Explanation of the input format:
First line of the input represents the number of test cases. Here the number of test cases are 3.
Every test case consumes three lines.
First line: number of elements in the tree.
Second line: Elements of tree separated by space.(level order)
Third line: myNode and k (separated by space).

For example,
for the first test case, there are 5 elements in the the tree. The elements are 1, 2, 3, 4, 5. myNode is 3 and k is 2.
for the second test case, there are 8 elements in the the tree. The elements are 34, 67, 12, 36, 90, 23, 56, 57. myNode is 12 and k is 3.
for the second test case, there are 6 elements in the the tree. The elements are 34, 67, 12, 36, 90, 23. myNode is 12 and k is 20.

Output format:
```
$ $
67 57
67 $
```

Sample output explanation:
Output corresponding to each test case contains 2 values. First is the value of IRNode and second is value of k-th minimum node of the subtree hanging from IRNode.
Test Case 1:
For the node 3, no IRNode exists, so output $. Also in this case there is no kth minimum node, so output $.

Test Case 2:
For the node 12, IRNode is 67, so output 67. Also in this case there is k=3, so output 3rd minimum element in the subtree which is 57.

Test Case 3:
For the node 12, IRNode is 67, so output 67. Also in this case there is k=20, so output 20th minimum element. Since there are less than k elements in the subtree, So output $.

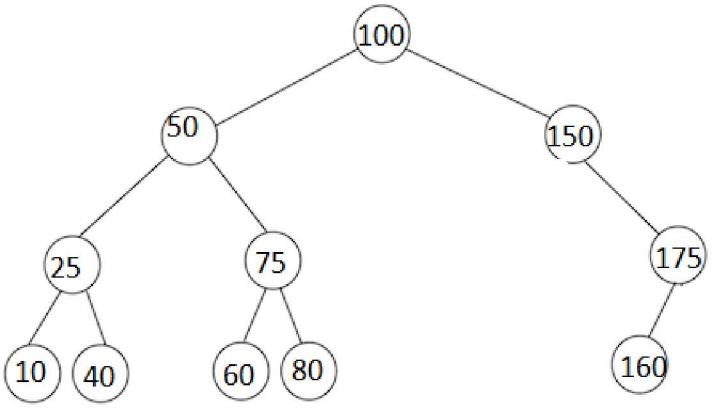
Explanation:

Consider the following example:

Elements are inserted into a BST in following order.

100, 50, 150, 25, 75, 175, 10, 40, 60, 80, 160

The resulting tree is



For Node 75, the immediate right node is 175. And the for k=1, kth minimum element (in the sub-tree hanging from 175) is 160.

For Node 150, immediate right node doesn't exist.

For Node 50, immediate right node is 150. And the for k=3, kth minimum element (in the sub-tree hanging from 150) is 175.

For Node 160, immediate right node doesn't exist.