**ADVANCED DATA STRUCTURES**

**Problem Statement**

Shashank loves trees and math. He has a rooted tree, consisting of nodes uniquely labeled with integers in the inclusive range [1,N] . The node labeled as 1 is the root node of tree T , and each node in is associated with some positive integer value (all values are initially 0).

Let's define Fk as the kth Fibonacci number. Shashank wants to perform 2 types of operations over his tree, T:

1. **U X k**

Update the subtree rooted at node X such that the node at level 0 in subtree X (i.e., node X) will have Fk added to it, all the nodes at level 1 will have Fk+1 added to them, and so on. More formally, all the nodes at a distance D from node X in the subtree of node X will have (k+D)th the Fibonacci number added to them.

1. **Q X Y**

Find the sum of all values associated with the nodes on the unique path from X to Y. Print your sum modulo 109 + 7 on a new line.

Given the configuration for tree T and a list of M operations, perform all the operations efficiently.

Note: F1 = F2 = 1

**Input Format**

The first line contains 2 space-separated integers, N(the number of nodes in tree T) and M(the number of operations to be processed), respectively.

Each line i of the N-1 subsequent lines contains an integer, P, denoting the parent of the (i+1)th node.

Each of the M subsequent lines contains one of the two types of operations mentioned in the Problem Statement above.

**Constraints**

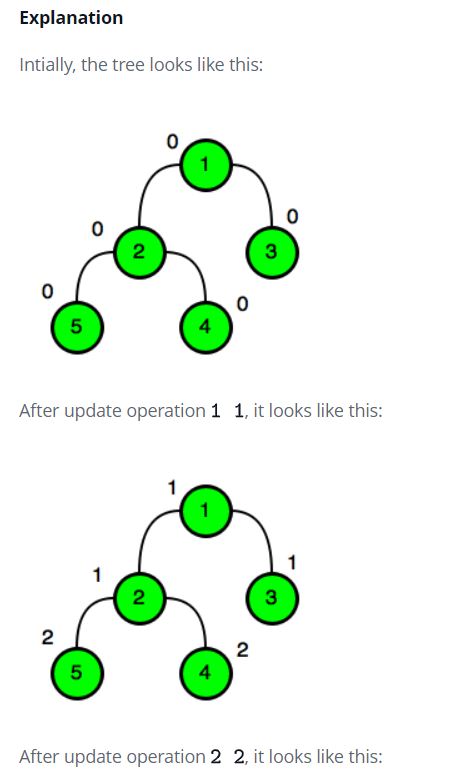
1<=N,M<=105

1<=X,Y<=N

1<=k<=1015

**Output Format**

For each operation of type 2(i.e., Q), print the required answer modulo 109 + 7 on a new line.

**Sample Input**

5 10

1

1

2

2

Q 1 5

U 1 1

Q 1 1

Q 1 2

Q 1 3

Q 1 4

Q 1 5

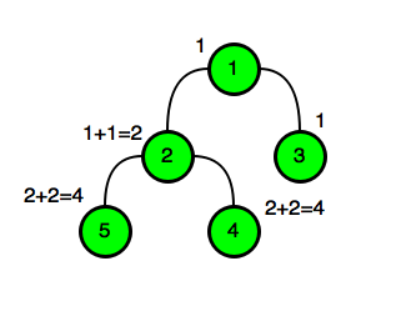
U 2 2

Q 2 3

Q 4 5

**Sample Output:**

0

1

2

2

4

4

4

10

**Code with explanation**

import java.io.\*;

import java.util.\*;

public class Solution {

static final int maxlevels = 17,high = 1000000007;

//maxlevel - For n nodes, there will be max. logn levels

//high - The upper limit is given as 10^9+7

static int check = 0;

//check - used for keeping the visited status

static List<Integer>[] tree1;

//tree1 - stores all the nodes with child as its list

static long[] dep,most;

static int[][] val;

//val - Matrix in which the fibonacci values are updated

//p1,p2 - used for tracking previous and post results.

static int[] p1,p2;

/\*Every node of tree will have a value, parent, bool value for visited status\*/

static class node {

int u,p;

long d;

boolean start = true; //all nodes are initially assigned true and then made false after visiting.

public node(int u, long d, int p) {

this.u = u;

this.d = d;

this.p = p;

}

}

/\*The pathSum() function is used for finding the sum of values from root to node given\*/

static long pathSum(int node) {

Hello h1 = find1(getSum(less, p1[node]+1), dep[node]);

return (h1.first + getSum(most, p1[node]+1)) % high;

}

static void traverse(int u, long d, int p) {

Deque<node> d1 = new LinkedList<>();

//we use a double ended queue for initialization purposes.

d1.add(new node(u, d, p));

//we add the values in obj which is pushed in deque

while (!d1.isEmpty()) {

node node = d1.peekLast(); //the last element inserted is taken

if (node.start) { //if the node bool value is true, the values are assigned correspondingly

dep[node.u] = node.d;

val[0][node.u] = node.p;

p1[node.u] = check++;

//the tree1[] is traversed for every element and if it is not root,

//we move onto that node and make parent as node.u

for (int v: tree1[node.u]) {

if (v != node.p) {

d1.add(new node(v, node.d+1, node.u));

}

}

node.start = false; //once the node is visited, it is marked as false

} else {

p2[node.u] = check;

d1.removeLast(); //if the node is visited, then the node is removed.

}

}

}

/\* This function is used to calculate the sum of values from the given node to all nodes in demo[].

It is used in pathSum() function\*/

static long getSum(long[] demo, int i) {

long s = 0;

for (; i > 0; i &= i-1) {

s = (s + demo[i-1]) % high;

}

return s;

}

/\*This class is defined to change the value of the nodes\*/

static class Hello {

/\*Initialization is done\*/

long first = 0;

long second = 0;

Hello ()

{}

/\*Parametrized constructors are defined\*/

Hello(Hello h1) {

this.first = h1.first;

this.second = h1.second;

}

Hello(long first, long second) {

this.first = first;

this.second = second;

}

/\*This function adds new fibo value to the existing nodes in hi obj\*/

void add(Hello h1) {

if (h1 != null) {

first = (first + h1.first) % high; //The new first value is obtained

second = (second + h1.second) % high; //The new second value is obtained

}

}

}

/\*This function is used to add child nodes to existing child in demo[] array\*/

static void add(Hello[] demo, int n, int i, Hello x) {

for (; i < n; i |= i+1) {

demo[i].add(x); //the child is appended in the list

}

}

/\*This function is used to add fibo values to existing child in demo[] array\*/

static Hello[] less;

static void add(long[] demo, int n, int i, long x) {

for (; i < n; i |= i+1) {

demo[i] = (demo[i]+x)%high;//Values are added and new value is obtained.

}

}

/\*This function is used to find the corresponding fibo value to be added to each node\*/

static Hello find1(Hello x, long n) {

if (n >= 0) {

long sa = 1, sb = 0, a = 0, b = 1;

for (; n > 0; n /= 2) { //as traversing to each child, path becomes n/2

if ((n & 1) > 0) { //true if n is odd

long ta = sa;

sa = (sa\*a+sb\*b)%high; //the 'first' value is sa

sb = (ta\*b+sb\*a+sb\*b)%high;//the 'second' value here is sb

}

long ta = a;

a = (a\*a+b\*b)%high;

b = (2\*ta\*b+b\*b)%high;

}

Hello h1=new Hello((sa\*x.first+sb\*x.second)%high, (sb\*x.first+(sa+sb)\*x.second)%high);

return h1;

}

else // if n(from func) is <0

{

long sa = 1, sb = 0, a = 0, b = 1;

for (n = -n; n > 0; n /= 2) { //as n is negative.

if ((n & 1) > 0) {

//same logic as above

long ta = sa;

sa = (sa\*a+sb\*b)%high;

sb = (ta\*b+sb\*a-sb\*b)%high;

}

long ta = a;

a = (a\*a+b\*b)%high;

b = (2\*ta\*b-b\*b)%high;

}

x.second = (x.second-x.first)%high;

Hello h1= new Hello((sa\*x.first+sb\*x.second)%high, ((sa+sb)\*x.first+sa\*x.second)%high);

return h1;//It returns the corresponding Hello obj with first and second value

}

}

/\*This function is used to find the common parent between u and v nodes \*/

static int lowestcommonancestor(int u, int v) {

if (dep[u] < dep[v]) { //if the count of the first node is less than second node

int tmp = u;

u = v;

v = tmp;

}

/\*This is used to find the parent in each subsequent level.\*/

for (int i = maxlevels; --i >= 0; ) {

if (dep[u]-dep[v] >= 1 << i) {

u = val[i][u];

}

}

/\*if the parent is same, then it returns it.\*/

if (u == v) {

return u;

}

/\*For all levels, the parent of u and v are traced from bottom.\*/

for (int i = maxlevels; --i >= 0; ) {

if (val[i][u] != val[i][v]) {

u = val[i][u];

v = val[i][v];

}

}

return val[0][u];/\*The value in the first row of uth column will have the common ancestor\*/

}

/\*This function is used to give the sum of the nodes of all child. Used in pathSum() function\*/

static Hello getSum(Hello[] demo, int i) {

Hello s = new Hello();

for (; i > 0; i &= i-1) {

s.add(demo[i-1]);/\*The childs are appended to the hello object\*/

}

return s; //The object of hello is returned to the pathSum() function

}

/\*Main function\*/

public static void main(String[] args) throws IOException {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

StringTokenizer st = new StringTokenizer(br.readLine());

/\*Bufferred reader is used for reading the input line by line

String Tokenizer is used to split the values into token with " " delimiter\*/

int n = Integer.parseInt(st.nextToken()); //Number of nodes

int m = Integer.parseInt(st.nextToken()); //Number of queries

tree1 = new List[n]; //Initializing the tree with n list nodes

for (int i = 0; i < n; i++) {

tree1[i] = new LinkedList<>(); //initializing each node with linkedlist

}

for (int i = 1; i < n; i++) { //for traversing through n-1 parents as root node has no parent

st = new StringTokenizer(br.readLine()); //reading the next set of line

int parent = Integer.parseInt(st.nextToken())-1; //the subsequent parent is fetched and -1, because tree value starts from 0.

tree1[parent].add(i); //for every parent(input), the subsequent child is appended as a list to it

}

/\*Initializing all the global variables with size n for 1D and n\*n for 2D\*/

dep = new long[n];

val = new int[maxlevels][n];

p1 = new int[n];

p2 = new int[n];

/\*To initialize the val matrix, we use traverse() and assign the initial values\*/

traverse(0, 0, -1);

/\*loop to traverse through all levels and

replace the parent values level by level\*/

for (int i = 1; i < maxlevels; i++) {

for (int j = 0; j < n; j++) {

if(val[i-1][j] < 0)

{

val[i][j]=-1; //if parent is -1, then that is the root.

}

else

{

val[i][j]=val[i-1][val[i-1][j]]; //all subsequent nodes are replaced by parent values.

}

}

}

/\*Initializing most and less variables which are of type long and class Hello\*/

most = new long[n+1];

less = new Hello[n+1];

for (int i = 0; i <= n; i++) {

less[i] = new Hello(); //every array of less is associated to a object of Hello class.

}

/\*This loop iterates for all queries i.e. m\*/

while (m>0) {

st = new StringTokenizer(br.readLine()); //each query is read as a line

char op = st.nextToken().charAt(0); //every line consists of a character at first (Q or U)

int x = Integer.parseInt(st.nextToken())-1; //The node 'X' is mentioned here.

if (op == 'Q') {

int y = Integer.parseInt(st.nextToken())-1; //if it is Q, then the second parameter is node 'Y'

int z = lowestcommonancestor(x, y); //This function return the common parent of both X & Y nodes

/\*This gives the sum of the path from root to x and from root to y minus the root of common ancestor\*/

long result = pathSum(x) + pathSum(y) - pathSum(z);

if (z > 0) {

/\*//if the common ancestor is not the root of the tree, then the path length of root to z should be subtracted\*/

result -= pathSum(val[0][z]);

}

System.out.println(((result%high+high)%high));

//The final result value is the required value from x to y path. %high is used as the limit is 10^9+7

}

else if(op == 'U')

{

long y = Long.parseLong(st.nextToken()); //The value for 'k' is taken here which is the fibonacci index.

/\*a new obj for hello is created with first two fibo values\*/

Hello t = find1(new Hello(0, 1), y+1);

add(most, n, p1[x], - t.first); //this finds the root and increments its value to F(k)

add(most, n, p2[x], t.first); //this finds the child and increments its values to F(k+1)

t = find1(new Hello(0, 1), y-dep[x]+2);

add(less, n, p1[x], t);

add(less, n, p2[x], new Hello(- t.first, -t.second));

/\* Correspondingly the values for the subsequent child nodes are incremented by passing to the above function\*/

}

m--; //one query is successfully executed.} } }

**Time complexity analysis:**

For the first query i.e. **U X k**, the time complexity for executing each query is : **O(logn \* k)**. Because O(logn) is for find1() function which is used to assign fibonacci values level wise and O(k) is for add() function used to update the values of all subsequent parent and child nodes in the sub tree of X.

For the second query i.e. **Q X Y**, the time complexity for executing each query is : **O(logn \* logn)**. Because O(logn) is for finding the lowest common ancestor node of X and Y. The second O(logn) is for finding pathSum() where the values are added in its path. It is logn because the values are added from bottom level to top.

If there are m1 queries of type 1 and m2 queries of type 2 (such that m1 + m2 = m), then the total time complexity is : **O(m1\*(logn \* k) + m2\*(logn \* logn))**.

**Number of test cases passed:**

All test cases passed i.e. **18** .

**Problem Link:**

https://www.hackerrank.com/challenges/fibonacci-numbers-tree/problem

**Result:**

Thus, the program to implement the fibonacci numbers tree was executed successfully in hackerrank platform.