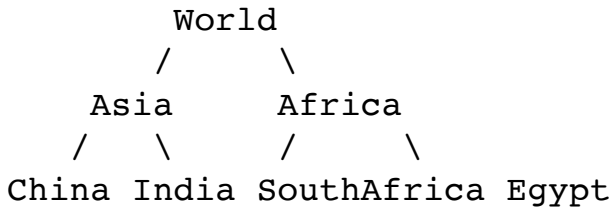
1. Problem Understanding

You're given a tree structure (specifically a complete **m-ary tree**) representing a **world map** hierarchy :

For example :



On this tree, three operations are defined :

1. Lock(NodeName, UserID)

Locks the node **only if**:

- It is not already locked.
- **No ancestor or descendant** of this node is locked.

2. Unlock(NodeName, UserID)

Unlocks the node **only if**:

- The node is currently locked **by the same user**.

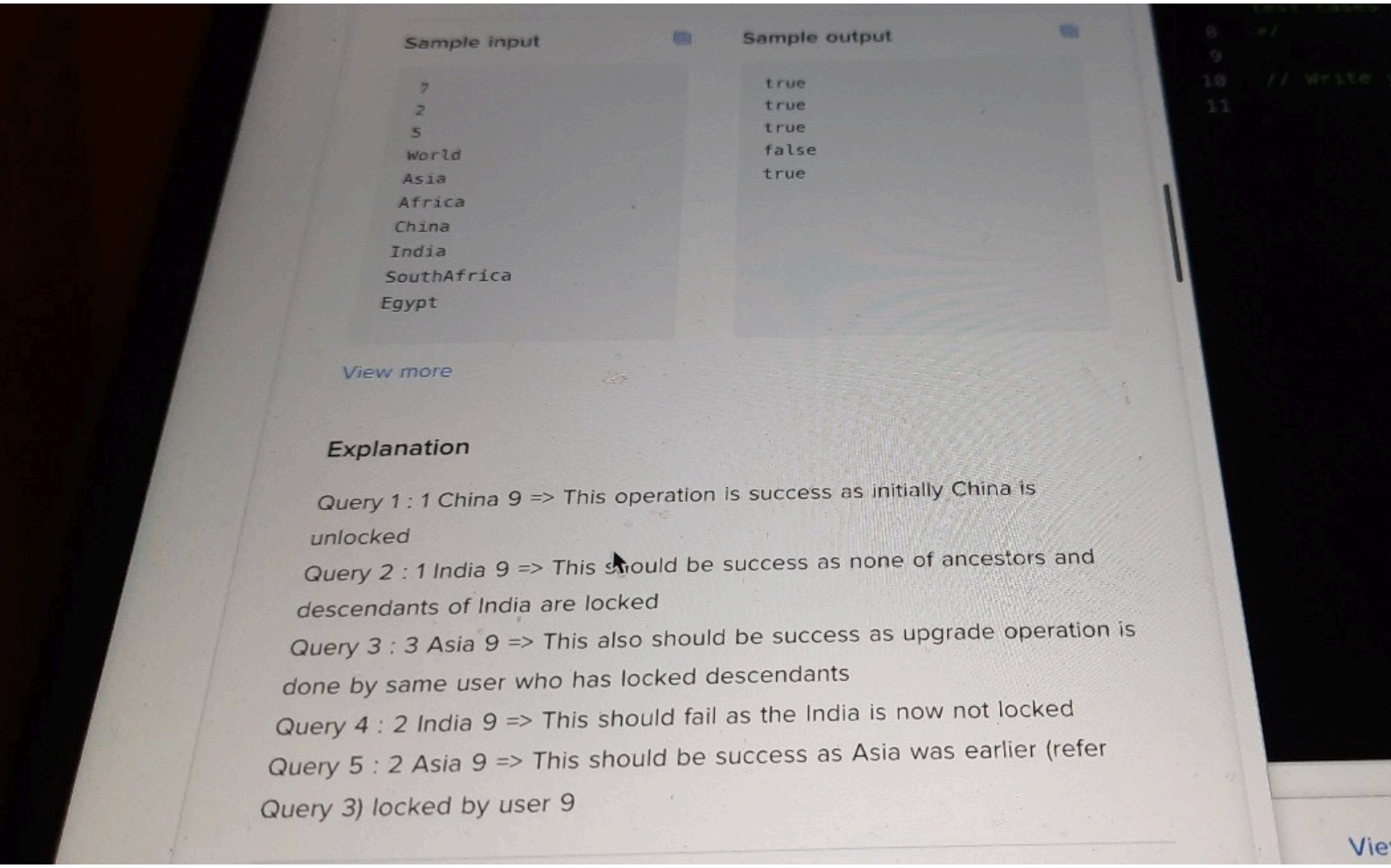
3. UpgradeLock(NodeName, UserID)

This is a special operation:

- You want to **lock** the given node by replacing all its locked descendants (if any).
- Conditions for success:
    - The node itself is not locked.
    - At least one **descendant is locked**.
    - **All locked descendants are by the same user (UID)**.
    - After the upgrade, the **node gets locked**, and all descendant locks are **removed**.

Input:

1. The first line contains the number of Nodes in the tree (N).
2. The second line contains number of children per node (value m in m-ary Tree).
3. The third line contains number of queries (Q).
4. Next N lines contains the NodeName (string) in the m-Ary
5. Next fires contains queries which are in Format OperationType NodeName Userld
6. OperationType->
   - 1 for Lock
   - 2 for unlock
   - 3 for upgradeLock
7. NodeName
8. Name of any node (unique) in m-Ary Tree.
9. Userld Integer value representing a unique user.



Code:

```
#include <iostream>
#include <unordered_map>
#include <vector>
using namespace std;

// Structure to represent each node in the tree
struct Node {
    string name;                    // Unique name of the node
    Node* parent = nullptr;         // Pointer to parent node
    vector<Node*> children;         // List of children nodes
    bool isLocked = false;          // Lock status of this node
    int lockedBy = -1;              // User ID who locked the node
    int lockedDescendants = 0;      // Number of locked descendants
};

// Global map for quick access to nodes by name
unordered_map<string, Node*> nodeMap;

/**
 * Checks if the node and its ancestors are free to lock or unlock.
 * Returns false if any ancestor is locked.
 */
bool canLockOrUnlock(Node* node) {
```

```
    Node* curr = node->parent;
    while (curr) {
        if (curr->isLocked) return false;
        curr = curr->parent;
    }
    return true;
}

/**
 * Updates the lockedDescendants count for all ancestors of the node.
 * Used when a lock or unlock operation occurs.
 */
void updateLockedDescendants(Node* node, int change) {
    Node* curr = node->parent;
    while (curr) {
        curr->lockedDescendants += change;
        curr = curr->parent;
    }
}

/**
 * Recursively checks if the node has any locked descendants.
 * Only descendants locked by the same user (uid) are considered valid.
 * Also collects such nodes to later unlock them if upgrade is allowed.
```

```
 */
bool hasLockedDescendants(Node* node, int uid, vector<Node*>& toUnlock)
{
    if (node->isLocked) {
        if (node->lockedBy != uid) return false;
        toUnlock.push_back(node);
    }
    for (Node* child : node->children) {
        if (!hasLockedDescendants(child, uid, toUnlock)) return false;
    }
    return !toUnlock.empty();
}


/**
 * Attempts to lock a node.
 * Succeeds only if the node is not already locked, has no locked descendants,
 * and no ancestors are locked.
 */
bool lock(string name, int uid) {
    Node* node = nodeMap[name];
    if (node->isLocked || node->lockedDescendants > 0 || !canLockOrUnlock(node)) return false;
    node->isLocked = true;
    node->lockedBy = uid;
```

```
    updateLockedDescendants(node, 1);
    return true;
}

/**
 * Attempts to unlock a node.
 * Succeeds only if the node is currently locked by the same user.
 */
bool unlock(string name, int uid) {
    Node* node = nodeMap[name];
    if (!node->isLocked || node->lockedBy != uid) return false;
    node->isLocked = false;
    node->lockedBy = -1;
    updateLockedDescendants(node, -1);
    return true;
}

/**
 * Attempts to upgrade the lock to a parent node.
 * All descendants must be locked by the same user.
 * If successful, all descendant locks are removed and the parent is locked.
 */
bool upgrade(string name, int uid) {
    Node* node = nodeMap[name];
```

```
    if (node->isLocked || node->lockedDescendants == 0) return false;

    vector<Node*> toUnlock;
    if (!hasLockedDescendants(node, uid, toUnlock)) return false;

    for (Node* n : toUnlock) {
        n->isLocked = false;
        n->lockedBy = -1;
        updateLockedDescendants(n, -1);
    }

    node->isLocked = true;
    node->lockedBy = uid;
    updateLockedDescendants(node, 1);
    return true;
}

/**
 * Main function to initialize the m-ary tree and process queries.
 */
int main() {
    int N, m, Q;
    cin >> N >> m >> Q;
```

```
// Read node names and create node objects
vector<string> names(N);
for (int i = 0; i < N; ++i) {
    cin >> names[i];
    nodeMap[names[i]] = new Node{names[i]};
}

// Build a fully balanced m-ary tree
for (int i = 1; i < N; ++i) {
    int parentIndex = (i - 1) / m;
    Node* parent = nodeMap[names[parentIndex]];
    Node* child = nodeMap[names[i]];
    child->parent = parent;
    parent->children.push_back(child);
}

// Process each query: lock, unlock, or upgrade
while (Q--) {
    int type, uid;
    string name;
    cin >> type >> name >> uid;

    if (type == 1)
        cout << (lock(name, uid) ? "true" : "false") << endl;
```

```
        else if (type == 2)
            cout << (unlock(name, uid) ? "true" : "false") << endl;
        else
            cout << (upgrade(name, uid) ? "true" : "false") << endl;
    }

    return 0;
}
```