In [ ]:

```
class TreeNode:
    def __init__(self, name):
        self.name = name
        self.locked = False
        self.locked_by = None
        self.locked_descendants = 0
        self.parent = None
        self.children = []

class M_ary_Tree:
    def __init__(self, m):
        self.nodes = {}
        self.m = m

    def add_node(self, node_name):
        node = TreeNode(node_name)
        self.nodes[node_name] = node
        return node

    def set_parent(self, child_name, parent_name):
        child_node = self.nodes[child_name]
        parent_node = self.nodes[parent_name]
        child_node.parent = parent_node
        parent_node.children.append(child_node)

    def can_lock_or_unlock(self, node):
        if node.locked_descendants > 0:
            return False
        temp = node.parent
        while temp:
            if temp.locked:
                return False
            temp = temp.parent
        return True

    def lock(self, node_name, uid):
        node = self.nodes[node_name]
        if node.locked or not self.can_lock_or_unlock(node):
            return False
        node.locked = True
        node.locked_by = uid
        temp = node.parent
        while temp:
            temp.locked_descendants += 1
            temp = temp.parent
        return True

    def unlock(self, node_name, uid):
        node = self.nodes[node_name]
        if not node.locked or node.locked_by != uid:
            return False
        node.locked = False
        node.locked_by = None
        temp = node.parent
        while temp:
            temp.locked_descendants -= 1
            temp = temp.parent
        return True
```

```
    def upgrade_lock(self, node_name, uid):
        node = self.nodes[node_name]
        if node.locked or not node.locked_descendants > 0:
            return False
        # Check if all locked descendants are locked by the same uid
        queue = [node]
        locked_descendants = []
        while queue:
            current_node = queue.pop(0)
            if current_node.locked:
                if current_node.locked_by != uid:
                    return False
                locked_descendants.append(current_node)
            queue.extend(current_node.children)

        # Perform the upgrade
        for locked_node in locked_descendants:
            locked_node.locked = False
            locked_node.locked_by = None
            temp = locked_node.parent
            while temp:
                temp.locked_descendants -= 1
                temp = temp.parent

        node.locked = True
        node.locked_by = uid
        temp = node.parent
        while temp:
            temp.locked_descendants += 1
            temp = temp.parent

        return True

# Function to process the input and perform operations
def process_input():
    n = int(input().strip())
    m = int(input().strip())
    q = int(input().strip())

    tree = M_ary_Tree(m)
    node_list = []
    for _ in range(n):
        node_name = input().strip()
        node_list.append(node_name)
        tree.add_node(node_name)

    for i in range(1, n):
        parent_index = (i - 1) // m
        tree.set_parent(node_list[i], node_list[parent_index])

    results = []
    for _ in range(q):
        query = input().strip().split()
        operation_type = int(query[0])
        node_name = query[1]
        uid = int(query[2])

        if operation_type == 1:
            result = tree.lock(node_name, uid)
```

```
        elif operation_type == 2:
            result = tree.unlock(node_name, uid)
        elif operation_type == 3:
            result = tree.upgrade_lock(node_name, uid)
        else:
            continue

        results.append("true" if result else "false")

    print("\n".join(results))

# Example usage (you can replace this with input())
process_input()
```