Daily Coding Problem #24

Problem

This problem was asked by Google.

Implement locking in a binary tree. A binary tree node can be locked or unlocked only if all of its descendants or ancestors are not locked.

Design a binary tree node class with the following methods:

- `is_locked`, which returns whether the node is locked
- `lock`, which attempts to lock the node. If it cannot be locked, then it should return false. Otherwise, it should lock it and return true.
- `unlock`, which unlocks the node. If it cannot be unlocked, then it should return false. Otherwise, it should unlock it and return true.

You may augment the node to add parent pointers or any other property you would like. You may assume the class is used in a single-threaded program, so there is no need for actual locks or mutexes. Each method should run in O(h), where h is the height of the tree.

Solution

A relatively easy way to implement this would be to augment each node with an `is_locked` attribute as well as a parent pointer. We can then implement the methods in a straightforward manner:

- `is_locked` simply returns the node's attribute
- `lock` searches the node's children and parents for a true `is_locked` attribute. If it is set to true on any of them, then return false. Otherwise, set the current node's `is_locked` to

true and return true.

- `unlock` simply changes the node's attribute to false. If we want to be safe, then we should search the node's children and parents as in `lock` to make sure we can actually unlock the node, but that shouldn't ever happen.

While `is_locked` is O(1) time, `lock` and `unlock` will take O(m + h) time where m is the number of nodes in the node's subtree (since we have to traverse through all its descendants) and h is the height of the node (since we have to traverse through the node's ancestors).

We can improve the performance of `lock` and `unlock` by adding another field to the node that keeps tracks of the count of locked descendants. That way, we can immediately see whether any of its descendants are locked. This will reduce our `lock` and `unlock` functions to only O(h). We can maintain this field by doing the following:

- When locking, if the locking succeeds, traverse the node's ancestors and increment each one's count
- When unlocking, traverse the node's ancestors and decrement each one's count

The code will look something like the following:

```
class LockingBinaryTreeNode(object):
    def __init__(self, val, left=None, right=None, parent=None):
        self.val = val
        self.left = left
        self.right = right

        self.parent = parent
        self.is_locked = False
        self.locked_descendants_count = 0

    def _can_lock_or_unlock(self):
        if self.locked_descendants_count > 0:
            return False

        cur = self.parent
        while cur:
            if cur.is_locked:
                return False
            cur = cur.parent
        return True

    def is_locked(self):
        return self.is_locked
```

```
    def lock(self):
        if self.is_locked:
            return False # node already locked

        if not self._can_lock_or_unlock():
            return False

        # Not locked, so update is_locked and increment count in all ancestors
        self.is_locked = True

        cur = self.parent
        while cur:
            cur.locked_descendants_count += 1
            cur = cur.parent
        return True

    def unlock(self):
        if not self.is_locked:
            return False # node already unlocked

        if not self._can_lock_or_unlock():
            return False

        self.is_locked = False

        # Update count in all ancestors
        cur = self.parent
        while cur:
            cur.locked_descendants_count -= 1
            cur = cur.parent
        return True
```

Now, `is_locked` is still O(1), but `lock` and `unlock` are both O(h) instead of O(m + h).