Part A Explanation:
This code implements a **lock management system** for nodes in an **n-ary tree**. Each node can be locked by a user, unlocked, or have its lock status upgraded based on certain conditions. The system tracks whether nodes or their descendants are locked and ensures that locking/unlocking follows the rules. Here's a detailed explanation of each section:

Class N (Node Class): This class represents each node in the tree. It contains:
nm (string): Name of the node.
lBy (int): The ID of the user who locked the node (-1 if it's not locked).
l (bool): Boolean indicating whether the node is locked.
ch (vector<N>) * : A vector of pointers to its children nodes.
par (N)* **:** Pointer to the parent node.
lDesc (set<N>)* **:** A set containing pointers to locked descendants of the current node.

Class LM (Lock Manager): This class manages the locking, unlocking, and upgrading operations on nodes. It contains: *nodes (unordered_map<string, N>):** A map that associates node names with their corresponding node objects.

- **Tree Construction (TB:: buildTree):**
- **Purpose:** Builds an n-ary tree from a list of node names.
- **Logic**: Starting with the root node, it assigns children in batches (chPerN children per node) and builds the tree level by level using a queue.

Core Functions:

Locking Operation (lock):

- **Purpose:** Locks a node if possible.
- **Conditions:**
    1. Node must not already be locked.
    2. Node should not have any locked descendants.
    3. None of its ancestors should be locked.
- **Logic:** The function checks if locking is allowed. It then updates the lBy field to the user ID and marks the node as locked. It also updates the locked descendants (lock Descendent) for the ancestors.

Unlocking Operation (unlock):

- **Purpose:** Unlocks a node.
- **Conditions:**
    1. Node must be locked.
    2. The node must be locked by the same user trying to unlock it.
- **Logic:** The function sets the lock status to false and clears the lBy field. It also updates the ancestors to remove the node from their list of locked descendants.

Upgrade Lock Operation (upgrade):

- **Purpose:** Unlocks all locked descendants of a node and locks the node itself.
- **Conditions:**
    1. Node should not already be locked.
    2. Node should have at least one locked descendant.
    3. All locked descendants must be locked by the same us
- **Logic:** First, the function ensures that all descendants are locked by the same user. Then, it unlocks all of them and locks the current node.

Supporting Functions:

- **updParL(N* p, N* n)**: Updates ancestors to include the node n as one of their locked descendants.
- **updParUnl(N* p, N* n)**: Updates ancestors to remove the node n from their list of locked descendants.

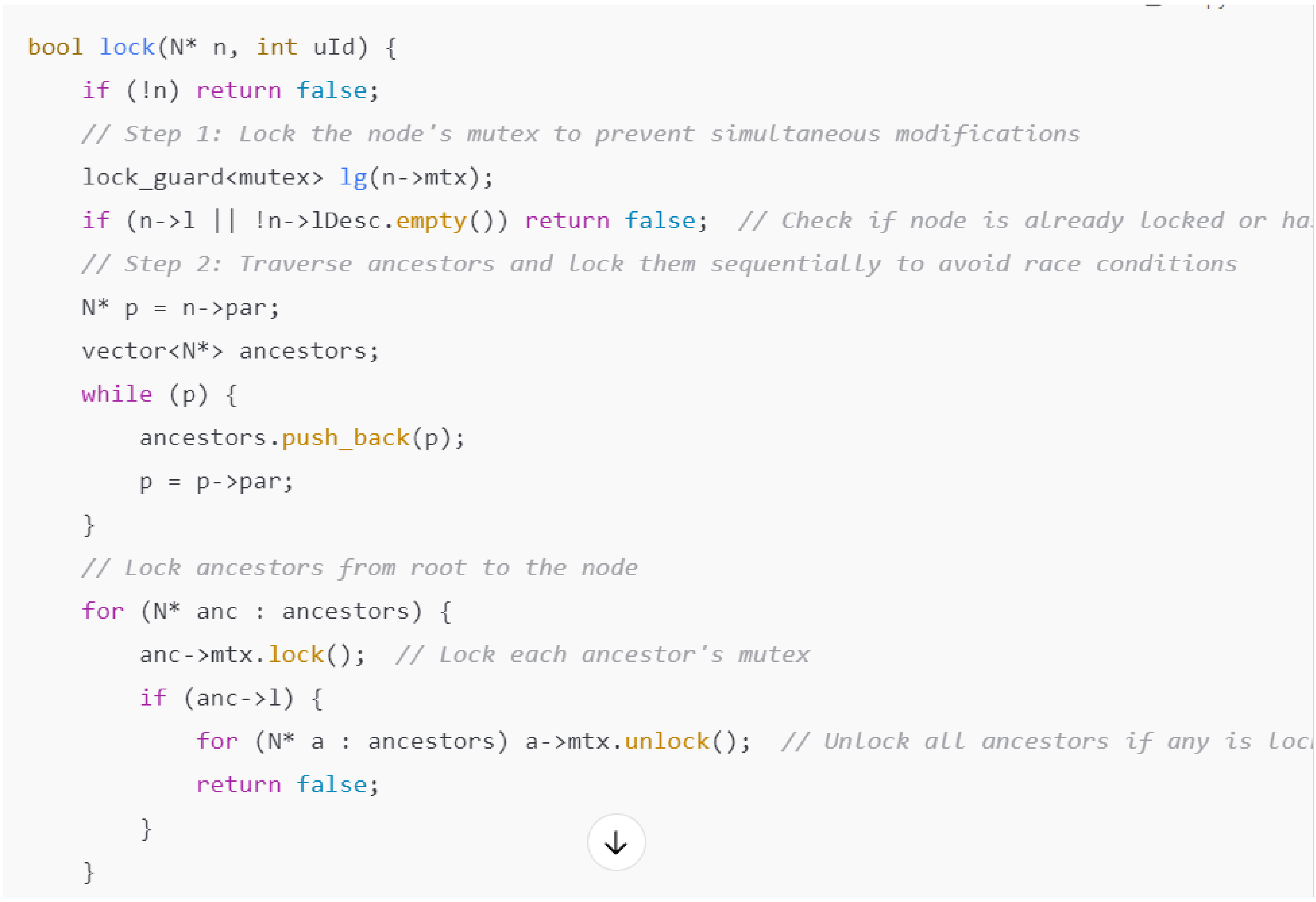
Data Structure Choices:

1. **Vector (ch) for Children Nodes:**
    - A vector is used for storing the children's nodes since the number of children is dynamic but typically bounded by m, and we often need to access them sequentially.
2. **Set (lDesc) for Locked Descendants:**
    - A set is used for lDesc because it provides efficient insertion, deletion, and lookup, which is essential for tracking unique locked descendants without duplicates.
3. **Unordered Map (nodes) for Node Lookup:**
    - An unordered_map is used to map node names to their corresponding node objects for constant time lookup, which is necessary for quick access to nodes by name.

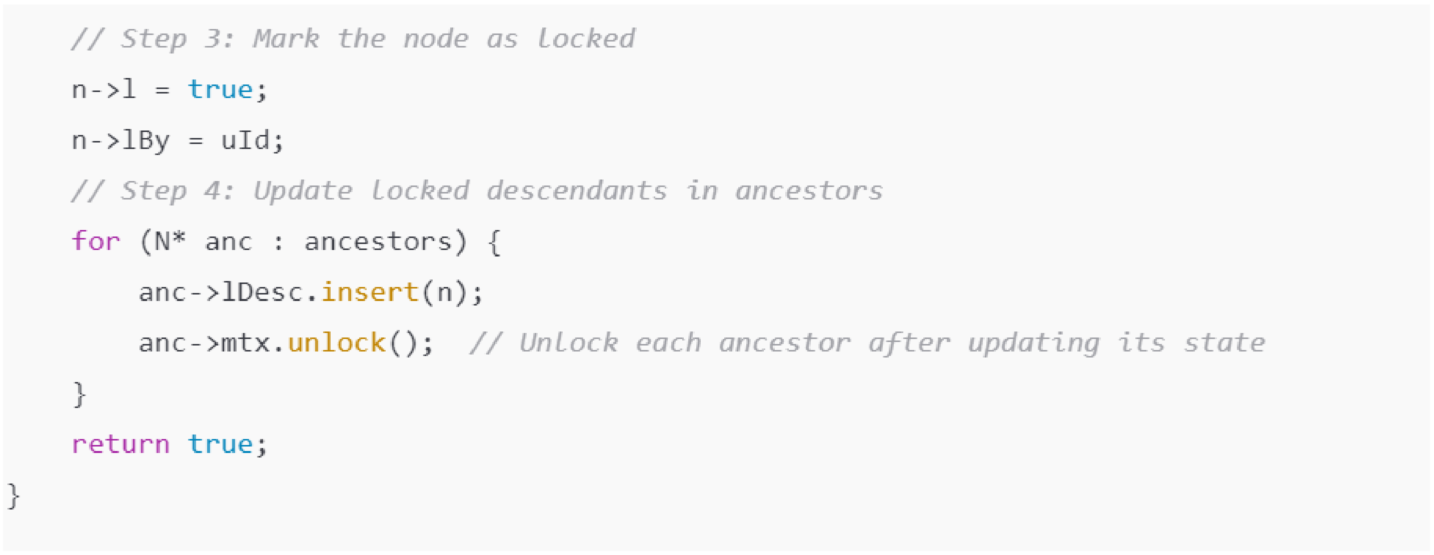
Summary of Constraints

1. **Time Constraints:**
    - Locking: **O(log(n))**.
    - Unlocking: **O(log(n))**.
    - Upgrading: **O(locked_descendants * log(n))**.
2. **Space Complexity: O(n)**.

Multithreading



```cpp
bool lock(N* n, int uId) {
    if (!n) return false;
    // Step 1: Lock the node's mutex to prevent simultaneous modifications
    lock_guard<mutex> lg(n->mtx);
    if (n->l || !n->lDesc.empty()) return false;  // Check if node is already locked or ha
    // Step 2: Traverse ancestors and lock them sequentially to avoid race conditions
    N* p = n->par;
    vector<N*> ancestors;
    while (p) {
        ancestors.push_back(p);
        p = p->par;
    }
    // Lock ancestors from root to the node
    for (N* anc : ancestors) {
        anc->mtx.lock();  // Lock each ancestor's mutex
        if (anc->l) {
            for (N* a : ancestors) a->mtx.unlock();  // Unlock all ancestors if any is Loc
            return false;
        }
    }
```



```cpp
    // Step 3: Mark the node as locked
    n->l = true;
    n->lBy = uId;
    // Step 4: Update locked descendants in ancestors
    for (N* anc : ancestors) {
        anc->lDesc.insert(n);
        anc->mtx.unlock();  // Unlock each ancestor after updating its state
    }
    return true;
}
```

Redundancy

if (!n || n->l || !n->lDesc.empty()) return false; // Ensure node exists and is not locked or has locked descendants

Explanation: !n is checked in every function (lock, unlock, upgrade), so it may not need to be checked here if already verified at the caller level.

vector<N*> descToUnlock(n->lDesc.begin(), n->lDesc.end()); // Copy to avoid modifying while iterating

Explanation: Creating a new vector is redundant here; an iterator-based approach or direct unlocking might be more efficient without needing a copy.

```
if (!n) {
    cout << "false" << endl;  // Ensure the node exists
    continue;  // Skip this iteration if node doesn't exist
}
```

Explanation: This line checks if n is nullptr after every operation, which may be redundant if you check node validity at the top level.

The lDesc set, which holds the locked descendants, should ideally only contain valid (non-null) N* pointers. Since you add nodes to lDesc only through updParL (which always passes a valid node pointer), there should be no nullptr elements in lDesc.

So, this line:

if (!desc || desc->lBy != uId) return false; // Ensure all locked descendants are by the same user

can be simplified to:

if (desc->lBy != uId) return false; // Ensure all locked descendants are by the same user

Final Recommendation:

Remove the !desc check from this line to reduce redundancy. This will make the code more concise without affecting functionality, as lDesc should never contain nullptr.

Race Condition

1. Race Condition in Lock and Unlock Operations

- **Issue**: Multiple threads could attempt to lock or unlock the same node simultaneously. For example, one thread might check that a node is not locked, but before it actually locks the node, another thread might lock it. This would cause inconsistent results, with both threads believing they "own" the lock.
- **Example**: In the lock method:

```
if (!n || n->l || !n->lDesc.empty()) return false;  // Check lock status
```

2. Race Condition in lDesc Updates

- **Issue**: The lDesc set in each node is modified by both updParL and updParUnl. If two threads attempt to lock different nodes that share a common ancestor, they could concurrently modify the lDesc set of that ancestor. This could lead to inconsistent or incorrect tracking of locked descendants.
- **Example**: In updParL and updParUnl:

```
void updParL(N* p, N* n) {
  while (p) {
    p->lDesc.insert(n);
    p = p->par;
  }
}
```

Without thread safety, concurrent modifications to lDesc may result in corrupted data or unexpected behavior.

3. Race Condition in upgrade Method

- **Issue**: The upgrade method first checks if all descendants are locked by the same user. If this check passes, it unlocks all descendants and then locks the current node. Without synchronization, another thread could lock one of the descendants in between these steps, causing an inconsistent state.
- **Example**: This section of code is vulnerable:

```
for (N* desc : n->lDesc) {
    if (desc->lBy != uId) return false;  // Ensure all locked descendants are by the same user
}
// Unlock all locked descendants
vector<N*> descToUnlock(n->lDesc.begin(), n->lDesc.end());
for (N* desc : descToUnlock) {
    unlock(desc, uId);
}
```

A race condition could occur if another thread locks one of the descendants in between the check and unlock steps.

1. Node Attributes in the N Class

- **l (Lock Status)**:

  Race Condition: Without a mutex, one thread could check that l is false (unlocked), but before it sets l to true, another thread could set l to true as well, causing both threads to believe they have successfully locked the node.

- **lBy (User ID that Locked the Node)**:
  Race Condition: If two threads attempt to lock the same node, they might both set lBy to their respective user IDs without synchronization, leading to an inconsistency where two different user IDs believe they own the lock on the same node.

- **lDesc (Locked Descendants Set)**:
  Race Condition: If two threads simultaneously lock or unlock different descendants of the same ancestor, they could both try to add or remove nodes from the ancestor's lDesc set without synchronization, resulting in corrupted or inconsistent data in lDesc.

Detailed Explanation of How Race Conditions Occur

1. **Locking (lock Method)**

```
bool lock(N* n, int uId) {
    if (!n || n->l || !n->lDesc.empty()) return false;
    N* p = n->par;
    while (p) {
        if (p->l) return false;
        p = p->par;
    }
    n->l = true;
    n->lBy = uId;
    updParL(n->par, n);
    return true;
}
```

Race Scenario:

- **Thread A** and **Thread B** both attempt to lock the same node n simultaneously.
- Both threads check n->l and find it false.
- Both proceed to set n->l = true and n->lBy = uId, leading to multiple users thinking they have locked the node

2. Upgrading (upgrade Method)

```
bool upgrade(N* n, int uId) {
    if (!n || n->l || n->lDesc.empty()) return false;
    for (N* desc : n->lDesc) {
        if (desc->lBy != uId) return false;
    }
    vector<N*> descToUnlock(n->lDesc.begin(), n->lDesc.end());
    for (N* desc : descToUnlock) {
        unlock(desc, uId);
    }
    return lock(n, uId);
}
```

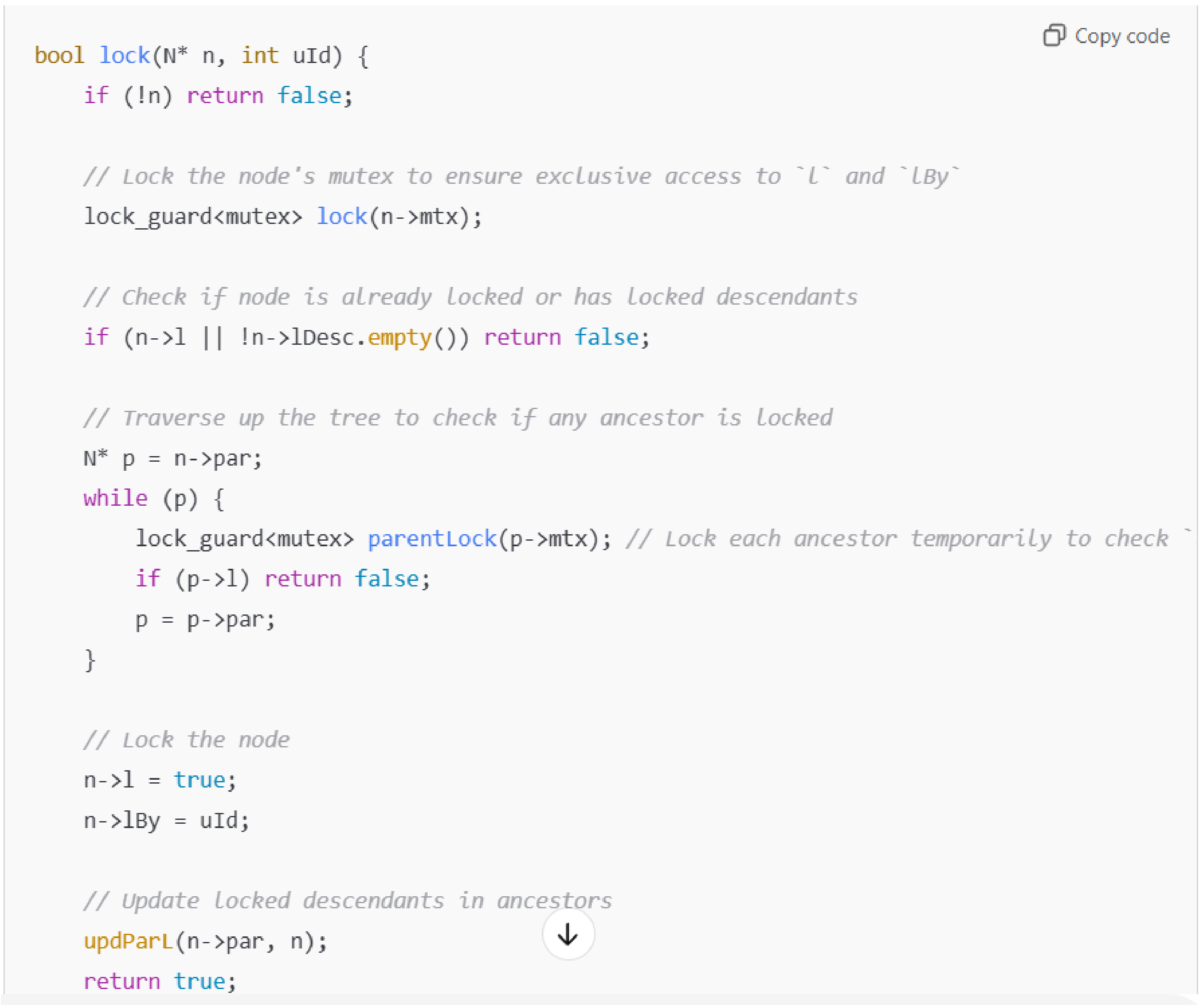
- **Race Scenario**:
    - **Thread A** is upgrading node n, checking that all locked descendants are by uId.
    - **Thread B** concurrently locks one of the descendants in n->lDesc.
    - **Thread A** might pass the initial check but fail to account for the change made by **Thread B**, leading to inconsistent states.

Summary of Variables Causing Potential Race Conditions

1. **N::l** (Locked Status)
2. **N::lBy** (Locked By User ID)
3. **N::lDesc** (Locked Descendants)

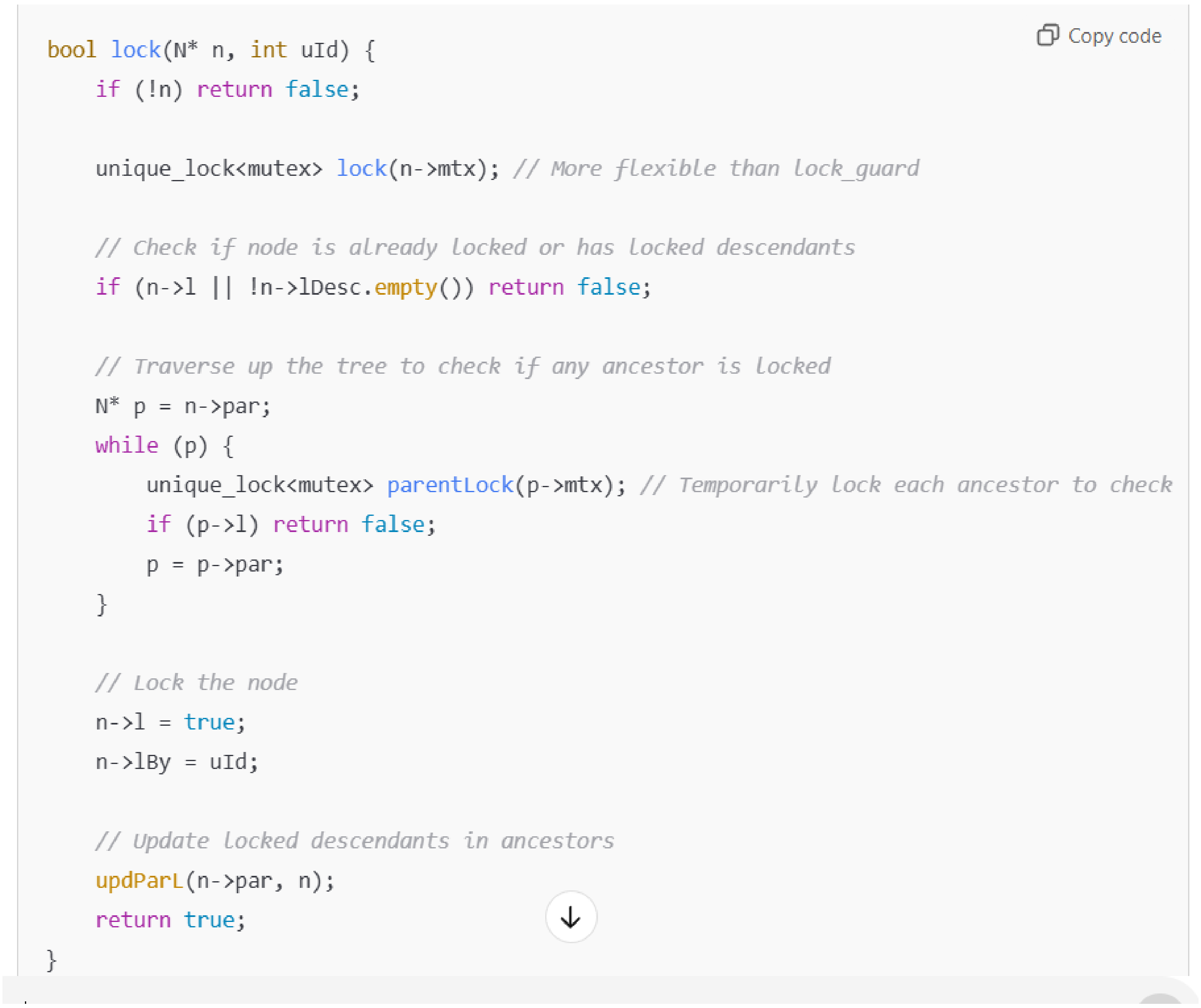
Thread Safe Lock function

To make the lock function thread-safe, you can use a **mutex** to ensure that only one thread can access or modify shared state at a time. Specifically, each node (N) can have its own mutex to protect the l (lock status), lBy (locked by user), and lDesc (locked descendants) fields.



```cpp
bool lock(N* n, int uId) {
    if (!n) return false;

    // Lock the node's mutex to ensure exclusive access to `l` and `lBy`
    lock_guard<mutex> lock(n->mtx);

    // Check if node is already locked or has locked descendants
    if (n->l || !n->lDesc.empty()) return false;

    // Traverse up the tree to check if any ancestor is locked
    N* p = n->par;
    while (p) {
        lock_guard<mutex> parentLock(p->mtx); // Lock each ancestor temporarily to check `
        if (p->l) return false;
        p = p->par;
    }

    // Lock the node
    n->l = true;
    n->lBy = uId;

    // Update locked descendants in ancestors
    updParL(n->par, n);
    return true;
```

➢ **with std::unique_lock**



```cpp
bool lock(N* n, int uId) {
    if (!n) return false;

    unique_lock<mutex> lock(n->mtx); // More flexible than lock_guard

    // Check if node is already locked or has locked descendants
    if (n->l || !n->lDesc.empty()) return false;

    // Traverse up the tree to check if any ancestor is locked
    N* p = n->par;
    while (p) {
        unique_lock<mutex> parentLock(p->mtx); // Temporarily lock each ancestor to check
        if (p->l) return false;
        p = p->par;
    }

    // Lock the node
    n->l = true;
    n->lBy = uId;

    // Update locked descendants in ancestors
    updParL(n->par, n);
    return true;
}
```

Time Complexity and Space Complexity

1. lock(N* n, int uId)

- **Time Complexity**: **O(h)**
  - The lock function has to check all ancestors of the node n to ensure none of them are locked. The time complexity for this is proportional to the height of the tree, which we denote as h.
  - Updating the locked descendants (updParL) also requires traversing up to the root node, again taking O(h) time.
- **Space Complexity: O(1)**
  - The lock function uses a constant amount of extra space since it only checks ancestors and updates pointers without creating new data structures.

2. unlock (N* n, int uId)

- **Time Complexity**: **O(h)**
  - Similar to lock, unlock has to update the lDesc set for all ancestors of node n (using updParUnl). This requires traversing up the tree, taking O(h) time.
- **Space Complexity**: **O(1)**
  - The unlock function also uses a constant amount of extra space, as it only modifies existing pointers and does not allocate additional memory.

3. upgrade (N* n, int uId)

- **Time Complexity**: O(h+d)
  - The upgrade function checks all descendants in n->lDesc to ensure they are locked by the same user (uId). If d is the number of descendants in n->lDesc, this step takes O(d)

    - Then, the function calls unlock on each locked descendant, taking O(d·h) time in the worst case since each unlock requires updating ancestors (which takes O(h) time per unlock call).
    - After unlocking, upgrade calls lock on the current node, which takes O(h) time.
    - Overall, the time complexity for upgrade is **O(d·h)** where d is the number of locked descendants and h is the tree height.
- **Space Complexity**: **O(d)**
    - The function creates a temporary vector (descToUnlock) to store all locked descendants, which has a space complexity of O(d) where ddd is the number of locked descendants.