Fine Grained Locking

For Fine grained locking I added a pthread_rwlock to each node, then locked and unlocked it as appropriate to avoid deadlocks and race conditions.

1. db_fine.c no longer include “db.h” but instead inlines the node definition.
    a. This is so that I can add a pthread_rwlock to the node without redefining the other DB’s (Or so I didn’t have to refactor node in the code)
2. node_create initializes the rwlock for each node
3. node_destroy destroys the rwlock whenever the node is destroyed
    a. Note, we don’t unlock the node when deleting it, we just destroy the lock
4. Since we are not locking the whole tree I didn’t change interpret_command at all for db_fine.
    a. any locking is done in the functions operating on the database directly

In summary, to lock the tree I realized that most of the nodes are accessed through search, so the nodes that we care about for add, query, or xremove have to be locked by search as it recurses through the tree.

5. query
    a. I assume that search locks all relevant nodes as it searches though them, and it unlocks them if they are not what it is searching for
    b. I also assume it read locks the thread if it finds it, so query must unlock the node if it finds it
    c. Therefore query doesn’t need to worry about locking the tree, search will do so
6. Add
    a. I assume here also that search locks any nodes it touches.
    b. Here I ask search to write lock anything it touches instead of just read locking it, since I don’t know if I found the item or not until I actually find it
        i. Therefore I can’t just read lock the node, since I’d then have to change that read lock to a write lock
    c. If It finds the node already exists it unlocks the parent and the target
    d. If it adds to the node it unlocks the parent after adding the node
7. xremove
    a. Here everything gets more complicated
    b. First it finds the node to remove
        i. If it doesn’t find it, it unlocks the parent of where it should be
    c. If it can delete the node as it is, it does so
        i. It doesn’t need to unlock the node since delete does so
        ii. The parent gets unlocked before the function returns
    d. If it needs to do a swap while deleting the node things get complicated
        i. It write locks the children as it searches for the final child it is swapping with

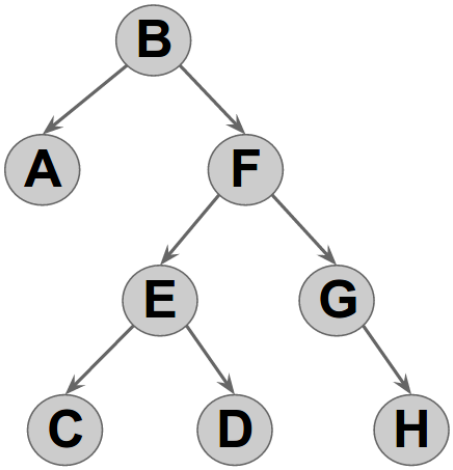
1. It unlocks the previous child when it grabs the next write lock

ii. Once it finds the child to swap with it, it has write locks on the child, the child's parent, and the original position

iii. Then it swaps, deletes the leaf that was swapped, and unlocks the original node and the parent of the leaf

iv. All of this locking only holds up to three write locks at a time

8. Search
    a. Added a field indicating if search should read or write lock items as it traverses the tree
        i. This is so that read only searches only read lock
        ii. This is so that searches for objects to write to don't have to convert a read lock to a write lock once it finds it
            1. I'd argue that it is not feasible to grab a read lock only while searching for the item to remove, then converting it to a write lock. Here's why



                a. In this image
                b. In the situation where we tried to convert a read lock to a write lock
                c. If we are trying to remove B & C, we could have a deadlock if:
                    i. Thread 1 searches for C, finds it, but only has a read lock
                    ii. Thread 2 searches for B, finds it, grabs the write lock, then tries to grab the write lock for C.
                    iii. Context switch back to thread1, and C gets deleted
                    iv. Thread 2 no longer finds C, error!
            2. This is why we must write lock as we search while running xremove
    b. Since search is recursive it is quite hard to keep track of where to lock / unlock nodes, but it is logically correct in my code
        i. I lock both the current node and next node, then unlock every node as we find that it is not the correct one
        ii. I also depend upon the fact that write locks won't block if the current thread already holds the write lock
    c. Search returns both the target and parent node as locked (write or read as specified)