B+Tree Locking Algorithm

This database implements a version of lock crabbing, optimized for each of instrumentation. Since it is a rather unique implementation, we've decided to write up a full explanation of the algorithm.

0.1 The Simple Case

Let's say we want to lock a B+Tree to be thread-safe. Assume for now that we won't change the structure of the B+Tree while we traverse it (i.e. we will not cause a split). Then, the locking scheme is rather simple:

```
lock(current)
while current.child != nil:
    lock(current.child)
    unlock(current)
    current = current.child
current.read()
unlock(current)
```

Critically, we do what is more commonly known as **hand-over-hand locking**, which ensures that we always have at least one point of contact in the database. If we flipped the lock of the child node and the unlock of the current node, there will be a split second where the child could move out from underneath us, making our locking algorithm unsafe. Thus, we always ensure that we are holding at least one lock until we finish.

0.2 The Complex Case

We've just gone over the simple case, which is the case where we can't cause a split. Let's lift this assumption and see what happens as a result.

Say we insert a tuple, Consider the following tree traversal, where each node is marked `x` if it might split (i.e. has `num_keys` keys):

```
[x] => [x] => [x] => [x] => [x]
```

Since every node might split, the node above it might change, meaning that it doesn't make sense to release any locks until after we finish our entire insertion. We can imagine the case where a split in a leaf triggers a split in the root node; therefore, we want to keep the root node so long as its possible that it may have to handle a split or split itself.

Let's go through a few more examples:

```
[x] => [x] => [ ] => [x] => [x]
```

Here, since our 3rd node has no chance of splitting, there is no need for us to continue holding locks on the 1st and 2nd nodes as we traverse further down the tree. So we don't need to hold locks until completion - when we encounter a node that definitely won't split, none of its parent locks need to be help anymore (note that we should lock the child node before unlocking the parent nodes to avoid the problem described in the simple case).

Another:

```
[] => [] => [] => [] => [x]
```

Pause and ponder: Which locks should we be holding when we perform the insertion?

In this case, we should hold on to locks on the last two nodes only. This is because if the last node splits, the second last node will have to shift keys around to accomodate it.

And a last one:

```
[x] => [x] => [x] => [x] => []
```

Pause and ponder: when we get to the leaf node, what can we do?

In this case, when we arrive at the leaf node, we can check that it will never split, meaning that we can unlock all of its parent nodes.

0.3 The Complex Case + Splitting

So far we have an incomplete policy. Here it is in summary:

```
lock(currentNode)
unlockParentsMaybe(currentNode)
recur(child)
```

But we need to handle two more small details.

Firstly, we need to make sure that a leaf node always unlocks itself, since no other node will, and that the parents definitely get unlocked, since if the leaf node doesn't trigger it before the insert ends, nobody will:

```
lock(currentNode)
unlockParentsMaybe(currentNode)
if child != nil:
    recur(child)
else:
    write(currentNode)
    unlock(currentNode)
    unlockParentsDefinitely(currentNode)
```

Secondly, when we receive a split from our child, there are two cases. Either we also split, or we don't. In the latter case, we can definitely unlock all parents; in the former, we can't. In both cases, we know that no child has unlocked us, so we definitely have to unlock ourselves (think about it: if a child had unlocked us, but we recieved a split, this means someone screwed up):

```
lock(currentNode)
unlockParentsMaybe(currentNode)
if child != nil:
    split = recur(child)
    if not split:
        unlockParentsDefinitely(currentNode)
    unlock(currentNode)
else:
    split = write(currentNode)
    if not split:
        unlockParentsDefinitely(currentNode)
    unlock(currentNode)
```

This completes our protocol.

0.4 Implementation Details

There are two final details that are worth considering.

Firstly, if the root node splits, we want to ensure nobody can enter it while it is splitting, However, the root node doesn't have a parent, meaning that other tenants might try to access it or the new splitting node while the root splits. Thus, we define a super-node which holds the parent lock for the root node. Students don't have to worry about it, but know that this is to prevent issues when the root node splits.

Secondly is how we determine which nodes are parents and how many parents we need to unlock. To do this, we do 3 things: 1) When we lock a child, set that child's `parent` pointer to the current node. 2) When we unlock a node, set that node's `parent` pointer to `nil`. 3) When we want to unlock all parents, traverse `parent` pointers until we see `nil`.

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!