Problem 1. Simple Concurrency with B-Trees

Note: Real databases and file systems use very fancy fine-grained synchronization for B-Trees such as “hand-over-hand locking” (which we did not discuss), but this problem considers some relatively simpler approaches.

Suppose we have a B Tree supporting operations insert and lookup. A simple way to synchronize threads accessing the tree would be to have one lock for the entire tree that both operations acquire/release.

(a) Suppose instead we have one lock per node in the tree. Each operation acquires the locks along the path to the leaf it needs and then at the end releases all these locks. Explain why this allegedly more fine-grained approach provides absolutely no benefit.

(b) Now suppose we have one readers/writer lock per node and lookup acquires a read lock for each node it encounters whereas insert acquires a write lock for each node it encounters. How does this provide more concurrent access than the approach in part (a)? Is it any better than having one readers/writer lock for the whole tree (explain)?

(c) Now suppose we modify the approach in part (b) so that insert acquires a write lock only for the leaf node (and read locks for other nodes it encounters). How would this approach increase concurrent access? When would this be incorrect? Explain how to fix this approach without changing the asymptotic complexity of insert by detecting when it is incorrect and in (only) those cases, starting the insert over using the approach in part (b) for that insert. Why would reverting to the approach in part (b) be fairly rare?

Problem 2. Concurrent/Parallel Graph Traversal

(a) In Java or pseudocode, define an unbounded stack with operations push and pop that can be used by threads concurrently. Make the element type generic and use a linked list for the underlying implementation. A pop should not raise an error. Instead it should wait until the stack is not empty. Use a condition variable. (Java detail: The wait method throws an exception that Java requires you catch, but it doesn’t matter if you include this detail in your solution.)

(b) The method traverseFrom in the code below uses your answer to part (a) to apply the doIt method to every node reachable from some node in a graph. The order doIt is applied to nodes does not matter. The code uses 4 threads to try to improve performance. All you need to do is write the run method (probably around 10 lines) for GraphWalker using the other code. Requirements:

- No node should have doIt applied to it more than once. You can assume the visited field of each node is initially false for this purpose.
- Use the stack to hold nodes that still need to be processed. That way threads not busy completing a doIt call can find useful work.
- Avoid all synchronization errors. (Hint: Your code does not need any explicit synchronization since it can call other code that already performs synchronization.)
- Because doIt may be expensive, do not hold any locks while calling it.

Note that after all reachable nodes have been processed, all threads will be blocked forever waiting for the stack to become non-empty. That is fine for a homework problem. Also note that you may or may not find isVisited helpful.
Problem 3. Minimum Spanning Trees

(a) Weiss, problem 9.15(a). For Prim's algorithm, start with vertex A, show the resulting table (see Table 9.55 as an example), and indicate the order in which vertices are added. For Kruskal's algorithm, produce a table, similar to Table 9.56. Ties may be broken arbitrarily.

(b) Weiss, problem 9.15(b).