public boolean add(T item) {
    int key = item.hashCode();
    head.lock();
    Node pred = head;
    try {
        Node curr = pred.next;
        curr.lock();
        try {
            while (curr.key < key) {
                pred.unlock();
                pred = curr;
                curr = curr.next;
                curr.lock();
            }
            if (curr.key == key) {
                return false;
            }
            Node newNode = new Node(item);
            newNode.next = curr;
            pred.next = newNode;
            return true;
        } finally {
            curr.unlock();
        }
    } finally {
        pred.unlock();
    }
}

Figure 1: The FineList class: the add() method uses hand-over-hand locking to traverse the list. The finally blocks release locks before returning.

Problem 1. Explain why the fine-grained list’s add() method is linearizable.
public boolean add(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key <= key) {
            pred = curr; curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key) {
                    return false;
                } else {
                    Node node = new Node(item);
                    node.next = curr;
                    pred.next = node;
                    return true;
                }
            }
        } finally {
            pred.unlock(); curr.unlock();
        }
    }
}
public boolean remove(T item) {
    int key = item.hashCode();
    while (true) {
        Node pred = head;
        Node curr = pred.next;
        while (curr.key < key) {
            pred = curr; curr = curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred, curr)) {
                if (curr.key == key)
                    pred.next = curr.next;
                return true;
            } else {
                return false;
            }
        } finally {
            pred.unlock(); curr.unlock();
        }
    }
}

Figure 3: The OptimisticList class: the remove() method traverses ignoring locks, acquires locks, and validates before removing the node.

Problem 2. In the optimistic list algorithm, show that once tail becomes reachable from a node, then it remains reachable, even if that node is deleted. Recall that tail is the name of the sentinel at the end of the list and that head is the name of the one in the beginning.
**Problem 3.** Explain why the fine-grained locking list algorithm is not subject to deadlock.
Problem 4. Extend the OptimisticList, LazyList, and LockFreeList list algorithms described in pages 205, 208, and 213 of the book so that they represent multisets (lists that can contain duplicates). Since the point of this assignment is to introduce you to the sort of design patterns that arise when creating lock/wait-free implementations of an object’s methods, the stencil code that we give you will consist of only interfaces and a tester class that you will need to implement. (Note that most of the code you will need for this assignment will be from the book, so feel free to use it!)

The stencil code you’ll need to begin this problem can be found at /course/cs1760/pub/duplists.