Sample Solution for Homework 6

Problem 1  AMP, p. 220: Exercise 102: Linearization of add (5 Points)

To show that every call to the add method is linerizable, we have to show that there exist linearization points such that: (1) whenever add returns true the given item was not in the list before this point and it is inserted to the list after this point; (2) whenever add returns false, the item must already have been in the list before this point. In both case (1) and case (2), choose the point when control exits the loop on line 14 as the linearization point.

After the while loop terminates, the thread is holding the locks on curr and pred. By the way in which the locks are acquired, the loop ensures that pred must be reachable from head and that pred.next == curr must still hold after the loop. Moreover, the loop ensures pred.key < key. From the loop condition we additionally know that curr.key >= key holds after the loop.

Now, if add returned true, then the condition on line 15 was false, i.e., we must have curr.key > key. From the established properties and the sortedness of the list, it follows that item is not yet in the list. Hence, the insertion of the item is correct.

If on the other hand add returned false, then we have curr.key = key. Since curr is reachable from head and by the uniqueness of keys, it follows that item is already in the list. Again, add returns the correct result.

Note that there are multiple correct ways of choosing the linearization points. Choosing the exit point of the while loop has the advantage that we can use a single linearization point for all cases. However, it is not always possible to find a single linearization point.

Problem 2  AMP, p. 220: Exercise 105: Fine-grained contains (8 Points)

```java
public boolean contains(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();
            }
            return curr.key == key;
        } finally { curr.unlock(); }
    } finally { pred.unlock(); }
```
The contains method scans the list to find the pair of nodes \((\text{pred}, \text{curr})\) reachable from head such that \(\text{pred.next} == \text{curr}, \text{pred.key} < \text{key}\) and \(\text{curr.key} >= \text{key}\). The traversal uses hand-over-hand locking. Hence, the locks for both pred and curr are held when line 15 is reached and \(\text{curr.key} == \text{key}\) is evaluated. Thus, it ensures that the above properties still hold for the two nodes when line 15 is executed. Suppose \(\text{curr.key} == \text{key}\) evaluates to false. If follows that \(\text{curr.key} > \text{key}\). From the sortedness invariant of the list, \(\text{pred.next} == \text{curr}\), and \(\text{pred.key} < \text{key}\) we conclude that item cannot be in the set. Suppose on the other hand that \(\text{curr.key} == \text{key}\) evaluates to true. Then it follows from the uniqueness of keys that \(\text{curr.item} = \text{item}\). Hence, item is in the set.

Note that replacing the hand-over-hand locking implementation with the optimistic locking implementation would still be correct.

**Problem 3** AMP, p. 220: Exercise 109: Non-blocking optimistic contains (7 Points)

The alternative implementation is not linearizable. Consider an interleaved execution of two threads \(T_1\) and \(T_2\) on an empty set instance \(s\). Suppose thread \(T_1\) first executes \(s.add(1)\) and then executes \(s.remove(1)\) up to the beginning of line 34. Now, let \(T_2\) execute \(s.contains(1)\) right up to before it evaluates \(\text{curr.key} == \text{key}\) and returns the result. At this point, since 1 is in the set, we must have \(\text{curr.key} == \text{key}\). Now, let \(T_1\) continue its execution of \(s.remove(1)\). Since \(T_2\) is not holding any locks, \(T_1\) can complete its execution without blocking, removing 1 from the set and returning true indicating successful removal. Now, \(T_2\) continues. Since \(\text{curr.key} == \text{key}\) still holds it returns true, even though 1 is no longer in the set.

**Problem 4** AMP, p. 220: Exercise 112: Fault lazy validation (5 Points)

The new employee forgot about the add method, which may modify \(\text{pred.next}\) without marking \(\text{curr}\). The modified implementation would no longer be linearizable. Specifically, suppose we have an empty set instance \(s\) and a thread \(T_1\) executes \(s.add(1)\) right up to the beginning of line 9. At this point, \(\text{pred}\) and \(\text{curr}\) point to the two sentinel nodes of the list. In particular, \(\text{curr.key} == \text{key}\) does not hold. Now, another thread \(T_2\) executes \(s.add(1)\) until completion, returning true to indicate the successful insertion. Finally, \(T_1\) continues its execution. The call to \(\text{validate}\) returns true since neither of the sentinel nodes has been marked. Moreover, \(\text{curr.key} == \text{key}\) does still not hold. Hence, \(T_1\) reinserts 1 into the list and also returns true, even though 1 was already present in the list.