The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 14

July 14, 2009

Exercise 163. Recall that a skiplist is a *probabilistic* data structure. Although the expected peformance of a contains() call is $O(\log n)$, where n is the number of items in the list, the worst-case peformance could be $O(n)$. Draw a picture of an 8-element skiplist with worst-case performance, and explain how it got that way.

Solution See Fig. 1. Here, when value i was inserted, it randomly chose height $i + 1$.

Exercise 164. You are given a skiplist with probability p and MAX_LEVEL M. If the list contains N nodes, what is the expected number of nodes at each level from 0 to $M - 1$?

Solution The expected number of nodes at level i or higher is is $p^i \gg N$. The expected number at level i is f$(p^i - p^{i+1}) \cdot N$, for $i < M - 1$.

Exercise 165. Modify the LazySkipList class so find() starts at the level of the highest node currently in the structure, instead of the highest level possible (MAX_LEVEL).

Solution See the LazySkipListLimit class in the code folder. We keep the highest actual level in an AtomicInteger field, which we update whenever we create a node higher than the previous maximum. In this solution, we do not try to decrease the max if the node at the highest level is removed.

Exercise 166. Modify the LazySkipList to support multiple items with the same key.

Solution Simply change add() to return **void**, and delete the code where add() checks whether the value is already present.

Exercise 167. Suppose we modify the LockFreeSkipList class so that at Line **??** of Fig. **??**, remove() restarts the main loop instead of returning *false*.

Is the algorithm still correct? Address both safety and liveness issues. That is, what is an unsuccessful remove() call's new linearization point, and is the class still lock-free?

Solution The method is still correct, but no longer lock-free. The unmodified method had three possible linearization points, corresponding to finding the item, not finding the item, or finding the item marked. The modification simply eliminates the third. The class is no longer lock-free because a thread that halts after marking an item will block any theads looking for that item.

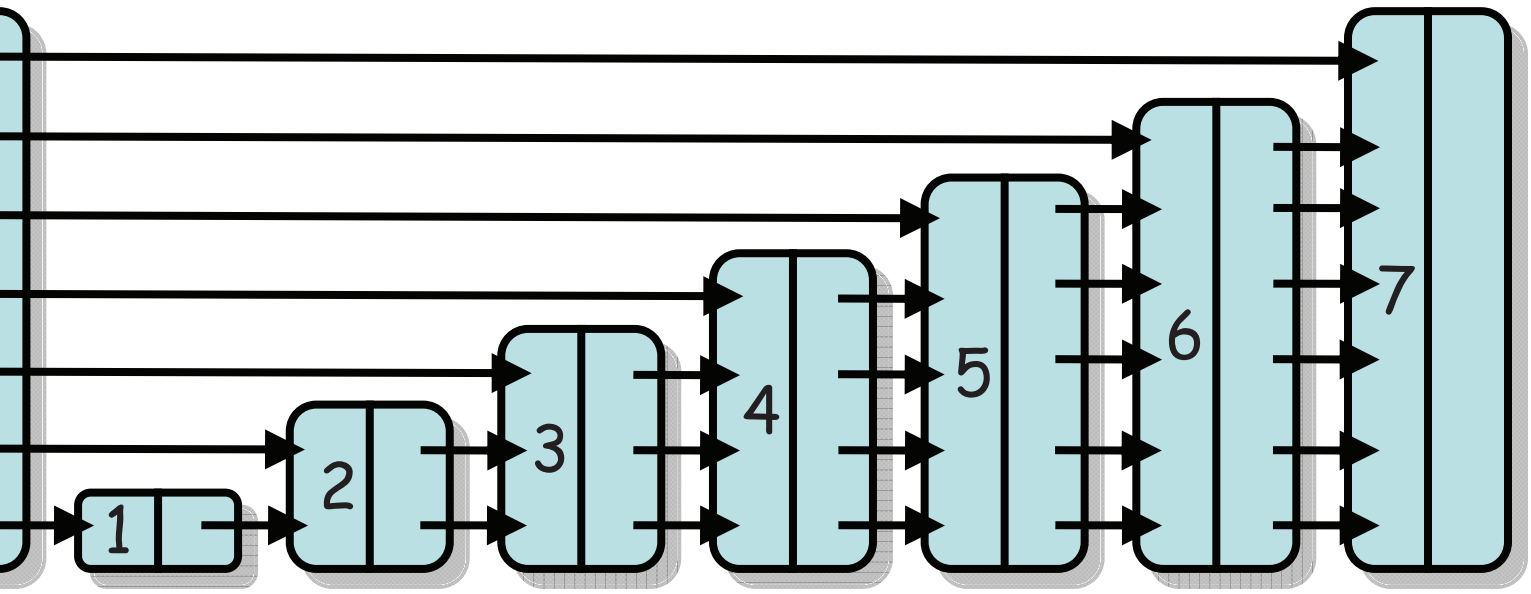

Figure 1: Worst-case skiplist performance

Exercise 168. Explain how, in the LockFreeSkipList class, a node might end up in the list at levels 0 and 2, but not at level 1. Draw pictures.

Solution See Fig. 3. At the start, there are two nodes, 2 and 5, both at level 2.

- Thread A inserts node 3 at level 2 between nodes 2 and 5.
- B removes 3, marking the references from 2 to 3 at levels 0 and 1, but then pauses.
- C inserts 4. It searches at level 2, finds an unmarked link from 3 to 5, so it moves down to level 1. At level 1, it finds a marked reference from 2 to 3, so it physically deletes that node, redirecting it from 2 to 5.

Exercise 169. Modify the LockFreeSkipList so that the find() method snips out a sequence of marked nodes with a single compareAndSet(). Explain why your implementation cannot remove a concurrently inserted unmarked node.

Solution [[[[Nir?]]]]

Exercise 170. Will the add() method of the LockFreeSkipList work even if the bottom level is linked and then all other levels are linked in some arbitrary order? Is the same true for the marking of the next references in the remove() method: the bottom level next reference is marked last, but references at all other levels are marked in an arbitrary order?

```
1  boolean contains(T x) {
2    int bottomLevel = 0;
3    int key = x.hashCode();
4    Node<T> pred = head;
5    Node<T> curr = null;
6    for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
7      curr = pred.next[ level ].getReference();
8      while (curr.key < key ) {
9        pred = curr;
10       curr = pred.next[ level ].getReference();
11     }
12   }
13   return curr.key == key;
14 }
```

Figure 2: The LockFreeSkipList class: an *incorrect* contains().

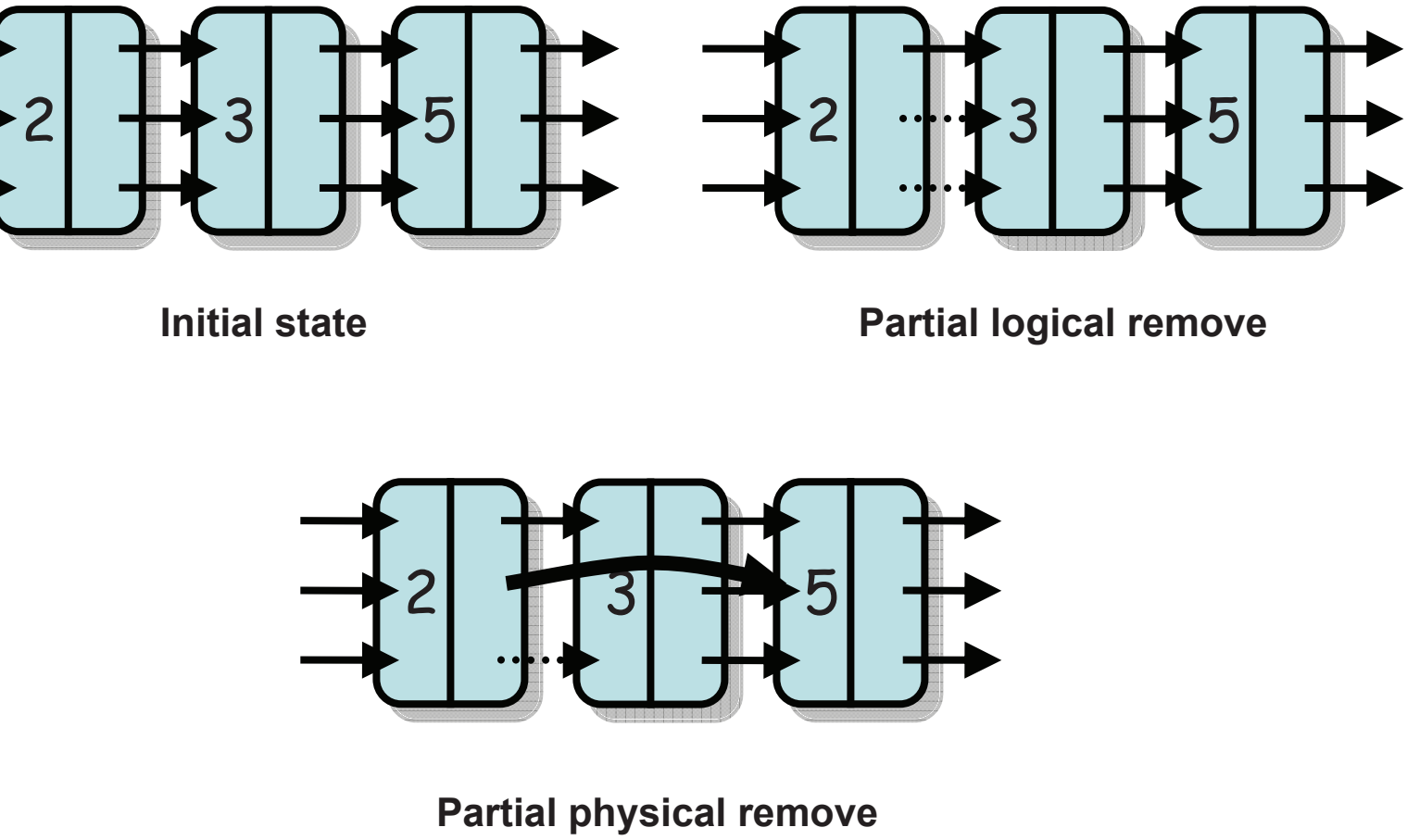

Figure 3: Solution to Exercise 168

Solution [[[[Nir?]]]]

Exercise 171. (Hard) Modify the LazySkipList so that the list at each level is bidirectional, and allows threads to add and remove items in parallel by traversing from either the head or the tail .

Solution [[[[Nir?]]]]

Exercise 172. Fig. 2 shows a buggy contains() method for the LockFreeSkipList class. Give a scenario where this method returns a wrong answer. Hint: the reason this method is wrong is that it takes into account keys of nodes that have been removed.

Solution

Solution 172 Initially the list has height 2, and has the values 1 and 4 in both levels.

1. A call to add(2) links a node with key 2 in the bottom layer, and stalls (Fig. 4).
2. A call to add(2) links a node with key 2 in the bottom layer, and stalls (Fig. 5).
3. A call to remove(2) marks 2's node at the bottom layer (Fig. 6).
4. The add(2) call continues. It fails to link 2's node in the 2nd layer, so it calls find() again, which now returns 1 and 3 as predecessor and successor. 2 is successfully spliced in between them (Fig. 7).

Now a contains(3) call finds 2 and 3 as the predecessor and successor in the second layer, goes down to the bottom layer, and traverses 2's next next reference to 4, and returns that 3 is not in the list. This execution is not linearizable, because the call to add(3) has returned.

Figure 4: Counterexample: Step 1

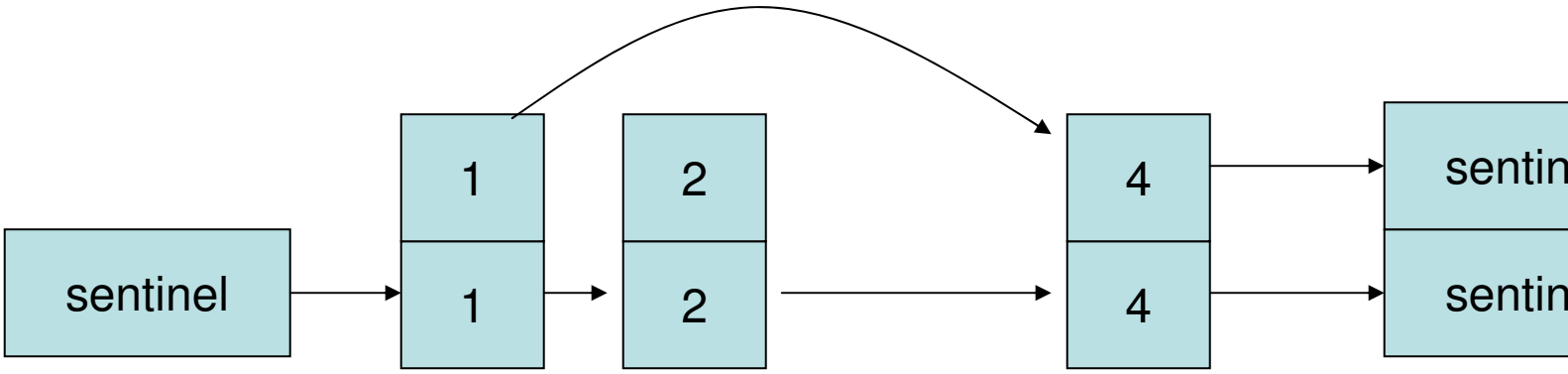

Figure 5: Counterexample: Step 2

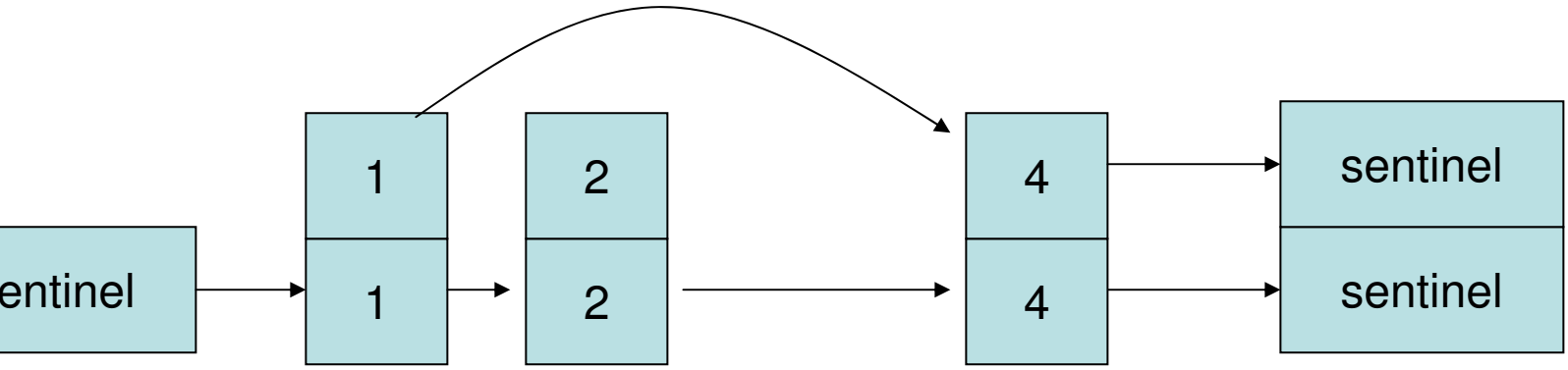

Figure 6: Counterexample: Step 3

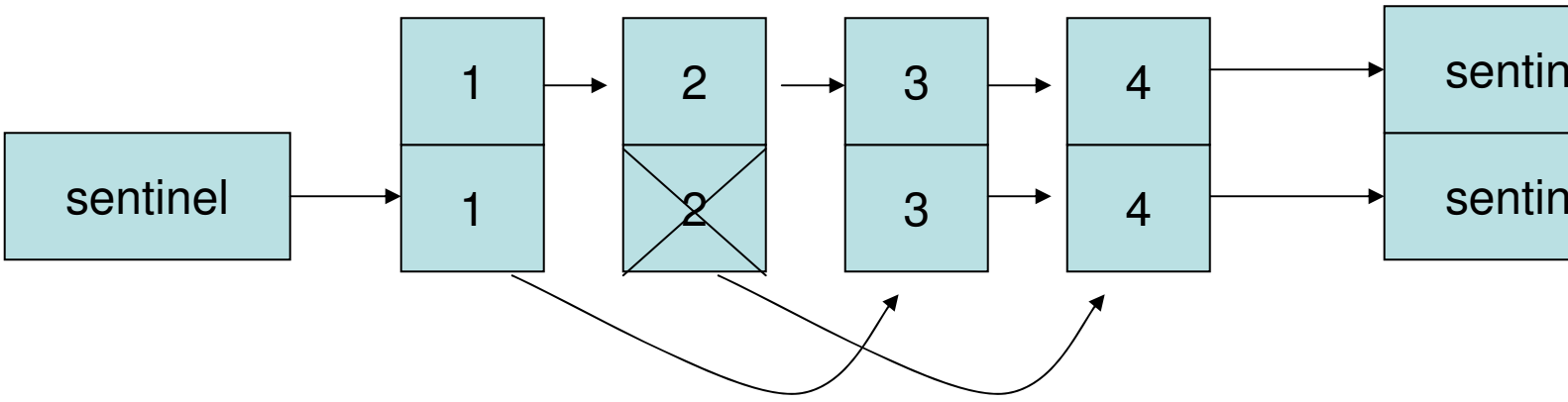

Figure 7: Counterexample: Step 4