The Art of Multiprocessor Programming
Solutions to Exercises
Chapter 12

July 14, 2009

Exercise 134. Prove Lemma **??**.

Solution If $y_0, \ldots, y_{w-1}$ is a sequence of nonnegative integers, we must show the following statements are all equivalent:

1. For any $i < j$, $0 \leq y_i - y_j \leq 1$.
2. Either $y_i = y_j$ for all i, j, or there exists some c such that for any $i < c$ and $j \geq c$, $y_i - y_j = 1$.
3. If $m = \sum y_i$, $y_i = \lceil \frac{m-i}{w} \rceil$.

Claim: (1) implies (2). Clearly, (1) implies that every such y_i and y_j differ by either 0 or 1. If y_i and y_j are not all equal, there is a least c such that $y_c - y_{c+1} = 1$. If $i, j < c$, then y_i and y_j must be equal, because otherwise y_i and y_c would differ by more than 1. If $i, j > c$, then y_i and y_j must be equal, because otherwise y_c and y_j would differ by more than 1.

Claim (2) implies (3). If $y_i = y_j$ for all i, j, then $m = wy_0$, $y_i = y_0 = \lceil \frac{wy_0 - i}{w} \rceil$. Otherwise, if for $i < c$ and $j \geq c$, $y_i - y_j = 1$, then $m = wy_0 + c$. For $i < c$, $y_i = y_0 = \lceil \frac{wy_i + c - i}{w} \rceil$. For $i \geq c$, $y_i = y_0 - 1 = \lceil \frac{wy_i + c - i}{w} \rceil$.

Claim (3) implies (1).

Let $m = \sum y_i$. For $i < j$, let $y_i = \lceil \frac{m-i}{w} \rceil$, and $y_j = \lceil \frac{m-j}{w} \rceil$. The difference between since $0 \leq i, j < m$, these two expressions can differ by at most one, and the second cannot exceed the first.

Exercise 135. Implement a *trinary* CombiningTree, that is, one that allows up to three threads coming from three subtrees to combine at a given node. Can you estimate the advantages and disadvantages of such a tree when compared to a *binary* combining tree?

Solution The advantage is that the tree is shallower, so traversing it may take lesstime. The disadvantage is that there is more opportunity for threads to block one anohter. Another disadvantage is the increased code complexity.

Most of the changes concern the Node<> class (see Fig 1). We add a THIRD to indicate that there are three values waiting to be combined. Instead of returning a Boolean, the precombine() method returns a status indicating whether the thread is first, second, or third. It locks the node only after there are two values waiting.

The combine() method uses the node status to determine how many nodes to combine.

The op() method behaves essentially the same if the node is in the FIRST or SECOND states. If the node is in the THIRD state, then it distributes both results to the firstResult and secondResult fields. The node cannot be ulocked until all results have been distributed, so we add an additional drained field to detect when all waiting threads have received their values.

```
1   public enum CStatus3 {
2     IDLE, FIRST, SECOND, THIRD, RESULT, ROOT
3   };
4  public class Node3 {
5     boolean locked;    // is node locked?
6     int drained;       // distributing after a triple combination
7     CStatus3 cStatus;  // combining status
8     int firstValue, secondValue, thirdValue; // values to be combined
9     int result, secondResult, thirdResult;       // results of combining
10    Node3 parent;      // reference to parent
11    /** Creates a root Node */
12    public Node3() {
13      cStatus = CStatus3.ROOT;
14      locked = false;
15    }
16    /** Create a non-root Node */
17    public Node3(Node3 _parent) {
18      parent = _parent;
19      cStatus = CStatus3.IDLE;
20      locked = false;
21    }
22    synchronized CStatus3 precombine() throws InterruptedException {
23      while (locked) {
24        wait();
25      }
26      switch (cStatus) {
27        case IDLE:
28          cStatus = CStatus3.FIRST;
29          return CStatus3.IDLE;
30        case FIRST:
31          cStatus = CStatus3.SECOND;
32          return CStatus3.SECOND;
33        case SECOND:
34          locked = true;
35          cStatus = CStatus3.THIRD;
36          return CStatus3.THIRD;
37        case ROOT:
38          return CStatus3.IDLE;
39        default:
40          throw new PanicException("unexpected Node state " + cStatus);
41      }
42    }
43    synchronized int combine(int combined) throws InterruptedException {
44      while (locked) {
45        wait();
46      }
47      locked = true;
48       firstValue = combined;
49      switch (cStatus) {
50        case FIRST:
51          return firstValue;
52        case SECOND:
53          return firstValue + secondValue;
54        case THIRD:
55          drained = 2; // need to hand out two values later
```

```
1   public int getAndIncrement() throws InterruptedException {
2     CStatus3 myStatus;
3     Stack<Node3> stack = new Stack<Node3>();
4     Node3 myLeaf = leaf[ThreadID.get() / 2];
5     Node3 node = myLeaf;
6     // phase one
7     while ((myStatus = node.precombine()) == CStatus3.IDLE) {
8       node = node.parent;
9     }
10    Node3 stop = node;
11    // phase two
12    node = myLeaf;
13    int combined = 1;
14    while (node != stop) {
15      combined = node.combine(combined);
16      stack.push(node);
17      node = node.parent;
18    }
19    // phase 3
20    int prior = stop.op(combined, myStatus);
21    // phase 4
22    while (!stack.empty()) {
23      node = stack.pop();
24      node. distribute ( prior );
25    }
26    return prior ;
27  }
```

Figure 2: Part of tree class for three-way combining

The Tree class is essentially unchanged (Fig. 2), except that in Phase 1, the thread has to remember whether it was first or second at the node where it stopped.

Exercise 136. Implement a CombiningTree using Exchanger objects to perform the coordination among threads ascending and descending the tree. What are the possible disadvantages of your construction when compared to the CombiningTree class presented in Section **??**?

Solution We can use Exchangers in the operation phase (phase three). We introduce two new Exchanger<Integer> fields: exchangeUp conveys the values to be combined to the thread moving up the tree, and exchangeDn conveys the value to be split to threads moving down the tree.

In this way, we can eliminate the need for the RESULT value, as well as narrowing the use of synchronized blocks.

Exercise 137. Implement the cyclic array based shared pool described in Section **??** using two simple counters and a ReentrantLock per array entry.

Solution Fig. 4 shows an implementation using the java.util.concurrent atomic types. (An alternative implementation using **synchronized** blocks is similar.) Note that when spinning, we employ a kind of test-and-test-and-set strategy. put() repeatedly reads its slot until it observes the slot is empty before trying to swap in an item (and similarly for get()).

Exercise 138. Provide an efficient lock-free implementation of a Balancer.

Solution Represent the balancer as a AtomicBoolean, where *true* represents up, and *false* down. Flip the balancer by reading the current state, and calling compareAndSet$(, t)$o attempt to flip the state. See Fig. 5. If the attempt fails, then some other thread has succeeded.

Exercise 139. (Hard) Provide an efficient wait-free implementation of a Balancer (i.e. not by using the universal construction).

Solution Represent the balancer as an AtomicInteger, where even values represent up, and odd values down. Flip the balancer by incrementing or decrementing the current state. To avoid overflow, threads alternate between incrementing and decrementing. The counter value always lies between $\pm n$, where n is the maximal number of threads.

Exercise 140. Prove that the TREE$[2k]$ balancing network constructed in Section **??** is a counting network, that is, that in any quiescent state, the sequences of tokens on its output wires have the step property.

```
1  public class NodeX {
2    enum CStatus {IDLE, FIRST, SECOND, ROOT };
3    boolean locked;    // is node locked?
4    CStatus cStatus;   // combining status
5    int  firstValue , secondValue; // values to be combined
6    int  result ;          // result of combining
7    NodeX parent;        // reference to parent
8    Exchanger<Integer> sendUp;
9    Exchanger<Integer> sendDn;
10   public NodeX() {
11     cStatus = CStatus.ROOT;
12     locked = false;
13   }
14   public NodeX(NodeX myParent) {
15     parent = myParent;
16     cStatus = CStatus.IDLE;
17     locked = false;
18   }
19   boolean precombine() throws InterruptedException {
20     lock();
21     switch (cStatus) {
22       case IDLE:
23         cStatus = CStatus.FIRST;
24         unlock();
25         return true;
26       case FIRST:
27         cStatus = CStatus.SECOND;
28         return false;
29       case ROOT:
30         return false;
31       default:
32         throw new PanicException("unexpected Node state " + cStatus);
33     }
34   }
35   int combine(int combined) throws InterruptedException {
36     lock();
37      firstValue = combined;
38     switch (cStatus) {
39       case FIRST:
40         return firstValue ;
41       case SECOND:
42         return firstValue + sendUp.exchange(null);
43       default:
44         throw new PanicException("unexpected Node state " + cStatus);
45     }
46   }
47   int op(int combined) throws InterruptedException {
48     switch (cStatus) {
49       case ROOT:
50         synchronized (this) {
51           int oldValue = result;
52           result += combined;
53           return oldValue;
54         }
55       case SECOND:
```

```
1  public class SimplePool<T> {
2    AtomicInteger putCounter, getCounter;
3    AtomicReferenceArray<T> items;
4    final int capacity;
5    public SimplePool(int myCapacity) {
6      capacity = myCapacity;
7      putCounter = new AtomicInteger(0);
8      getCounter = new AtomicInteger(0);
9      items = new AtomicReferenceArray<T>(capacity);
10   }
11   public void put(T x) {
12     int i = putCounter.getAndIncrement() % capacity;
13     while (true) {
14       if (items.get(i) == null) {
15         if (items.compareAndSet(i, null, x)) {
16           return;
17         }
18       }
19     }
20   }
21   public T get() {
22     int i = getCounter.getAndIncrement() % capacity;
23     while (true) {
24       if (items.get(i) != null) {
25         T y = items.getAndSet(i, null);
26         if (y != null) {
27           return y;
28         }
29       }
30     }
31   }
32 }
```

Figure 4: Cyclic array-based shared pool

```
1  public class BalancerLF {
2    AtomicBoolean toggle;
3    public BalancerLF() {
4      toggle = new AtomicBoolean(true);
5    }
6    public int traverse() {
7      while (true) {
8        boolean prior = toggle.get();
9        if (toggle.compareAndSet(prior, !prior)) {
10         return (prior ? 0 : 1);
11       }
12     }
13   }
14   public int antiTraverse() {
15     while (true) {
16       boolean prior = toggle.get();
17       if (toggle.compareAndSet(prior, !prior)) {
18         return (prior ? 1 : 0);
19       }
20     }
21   }
22 }
```

Figure 5: A lock-free balancer

```
1  public class BalancerWF {
2    AtomicInteger toggle;
3    private static final int THRESHOLD = Integer.MAX_VALUE / 2;
4    public BalancerWF() {
5      toggle = new AtomicInteger(0);
6    }
7    public int traverse() {
8      int prior = toggle.getAndIncrement();
9      while (prior > THRESHOLD) {
10       int state = toggle.get();
11       if (state % 2 == 0) {
12         toggle.compareAndSet(state, 0);
13       } else {
14         toggle.compareAndSet(state, 1);
15       }
16     }
17     return (prior % 2 == 0 ? 0 : 1);
18   }
19   public int antiTraverse() {
20     int prior = toggle.getAndIncrement();
21     while (prior > THRESHOLD) {
22       int state = toggle.get();
23       if (state % 2 == 0) {
24         toggle.compareAndSet(state, 0);
25       } else {
26         toggle.compareAndSet(state, 1);
27       }
28     }
29     return (prior % 2 == 0 ? 1 : 0);
30   }
31 }
```

Figure 6: A wait-free balancer

Solution By induction on the depth d of the tree. When $d = 1$, a single balancer's output has the step property. Assume the claim for trees of depth $d-1$. A tree of depth d consists of a root balancer and left and right subtrees of depth $d-1$. For m input values, the root sends $\lceil m/2 \rceil$ values left and $\lfloor m/2 \rfloor$ values right. The leaves of the left-hand child are the even-numbered outputs, and the leaves of the right-hand child the odd-numbered outputs. If m is even, then each subtree has the same number of inputs, and each has the same "step point" c where the outputs decrease by one. For the left-hand subtree, this point occurs at wire $2c$, and for the right-hand subtree, at $2c+1$. If m is odd, the left-hand tree's step point occurs at some $2c$, and the the right-hand tree's at $2c-1$. Either way, the step property is preserved.

Exercise 141. Let $\mathcal{B}$ be a width-w balancing network of depth d in a quiescent state s. Let $n = 2^d$. Prove that if n tokens enter the network on the same wire, pass through the network, and exit, then $\mathcal{B}$ will have the same state after the tokens exit as it did before they entered.

Solution. We argue by induction on d, the network depth.

When the network depth is 1, the tokens affect only one balancer, $s^d = 2$, and any even number of tokens leaves the balancer in the same state.

Assume the claim for networks of depth $d-1$. Consider a quiescent state after 2^{d-1} tokens enter on one wire. By the induction hypothesis, every balancer at depth less than d is in its original state (but not necessarily every balancer at depth d). If, however, 2^{d-1} *more* tokens enter on the same wire, for a total of 2^d tokens, then every balancer at depth less than d is still in its original state, but all balancers at depth d will have been visited by twice the number of tokens that is, an even number, so they will also be in their original states.

In the following exercises, a *k-smooth sequence* is a sequence $y_0, ..., y_{w-1}$ that satisfies

$$\text{if} \qquad i < j \qquad \text{then} \qquad |y_i - y_j| \leq k.$$

Exercise 142. Let X and Y be k-smooth sequences of length w. A *matching* layer of balancers for X and Y is one where each element of X is joined by a balancer to an element of Y in a one-to-one correspondence.

Prove that if X and Y are each k-smooth, and Z is the result of matching X and Y, then Z is $(k+1)$-smooth.

Solution Let x and y be the smallest values in X and Y. Suppose a balancer joins x_i and y_j to produce z_i and z_j. If the balancer is is in a state where it outputs the first token on wire i, then

$$z_i = \left\lceil \frac{x_i + y_j}{2} \right\rceil \qquad \text{and} \qquad z_j = \left\lfloor \frac{x_i + y_j}{2} \right\rfloor.$$

The smallest value that any element in Z can assume is thus

$$\left\lfloor \frac{x+y}{2} \right\rfloor$$

and the largest value is

$$\left\lceil \frac{x+k+y+k}{2} \right\rceil = \left\lceil \frac{x+y}{2} \right\rceil + k \leq \left\lfloor \frac{x+y}{2} \right\rfloor + k + 1.$$

The largest and smallest elements of Z differ by at most $k+1$.

Exercise 143. Consider a BLOCK $[k]$ network in which each balancer has been initialized to an arbitrary state (either *up* or *down*). Show that no matter what the input distribution is, the output distribution is $(\log k)$-smooth.

Hint: you may use the claim in Exercise 142.

Solution By induction on k. The claim is immediate when k is one.

Recall that a BLOCK $[2k]$ network is contructed from two parallel BLOCK $[k]$ networks joined by a final layer matchingtheoutputs of the half-size networks. By the induction hypothesis, the ouputs of the half-size networks are $\log(k)$-smooth, and by Exercise 142 the result of joining those $\log(k)$-smooth outputs is $\log(k)+1 = \log(2k)$-smooth.

Exercise 144. A *smoothing network* is a balancing network that ensures that in any quiescent state, the output sequence is 1-smooth.

Counting networks are smoothing networks, but not vice versa.

A Boolean sorting network is one in which all inputs are guaranteed to be Boolean. Define a *pseudo-sorting balancing network* to be a balancing network with a layout isomorphic to a Boolean sorting network.

Let $\mathcal{N}$ be the balancing network constructed by taking a smoothing network $\mathcal{S}$ of width w, a pseudo-sorting balancing network $\mathcal{P}$ also of width w, and joining the i^{th} output wire of $\mathcal{S}$ to the i^{th} input wire of $\mathcal{P}$.

Show that $\mathcal{N}$ is a counting network.

Solution Take any arbitrary sequence of *false*'s and *true*'s as inputs to the comparison network, and for the balancing network place a token on each *false* input wire and no token on each *true* input wire. We now show that if we run both networks in lock-step, the balancing network will simulate the comparison network, that is, the correspondence between tokens and *false*'s holds.

The proof is by induction on the depth of the network. For level 0 the claim holds by construction. Assuming it holds for wires of a given level k, let us prove it holds for level $k+1$. On every gate where two *false*'s meet in the comparison network, two tokens meet in the balancing network, so one *false* leaves on each wire in the comparison network on level $k+1$, and one token leaves on each line in the balancing network on level $k+1$. On every gate where two *true*'s

meet in the comparison network, no tokens meet in the balancing network, so a *true* leaves on each level $k+1$ wire in the comparison network, and no tokens leave in the balancing network. On every gate where a *false* and *true* meet in the comparison network, the *false* leaves on the lower wire and the *true* on the upper wire on level $k+1$, while in the balancing network the token leaves on the lower wire, and no token leaves on the upper wire.

If the balancing network is a counting network, i.e., it has the step property on its output level wires, then the comparison network must have sorted the input sequence of *false*'s and *true*'s.

Exercise 145. A *3-balancer* is a balancer with three input lines and three output lines. Like its 2-line relative, its output sequences have the step property in any quiescent state. Construct a depth-3 counting network with 6 input and output lines from 2-balancers and 3-balancers. Explain why it works.

Solution See Figure 7. For a complete discussin of this see:

- Aharonson, E. and Attiya, H. 1992. Counting networks with arbitrary fan-out. In Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms (Orlando, Florida, United States). Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics, Philadelphia, PA, 104-113.

- Busch, C. and Herlihy, M. 1999. Sorting and counting networks of small depth and arbitrary width. In Proceedings of the Eleventh Annual ACM Symposium on Parallel Algorithms and Architectures (Saint Malo, France, June 27 - 30, 1999). SPAA '99. ACM, New York, NY, 64-73. DOI= http://doi.acm.org/10.1145/305619.305627

Exercise 146. Suggest ways to modify the BitonicSort class so that it will sort an input array of width w where w is not a power of 2.

Solution The simplest approach is to pick the smallest k such that $2^k > w$, pad the inputs with $2^k - w$ values equal to the maximal integer size, and then discard the $2^k - w$ highest outputs.

Exercise 147. Consider the following w-thread counting algorithm. Each thread first uses a bitonic counting network of width w to take a counter value v. It then goes through a *waiting filter*, in which each thread waits for threads with lesser values to catch up.

The waiting filter is an array filter [] of w Boolean values. Define the phase function

$$\phi(v) = \lfloor (v/w) \rfloor \bmod 2.$$

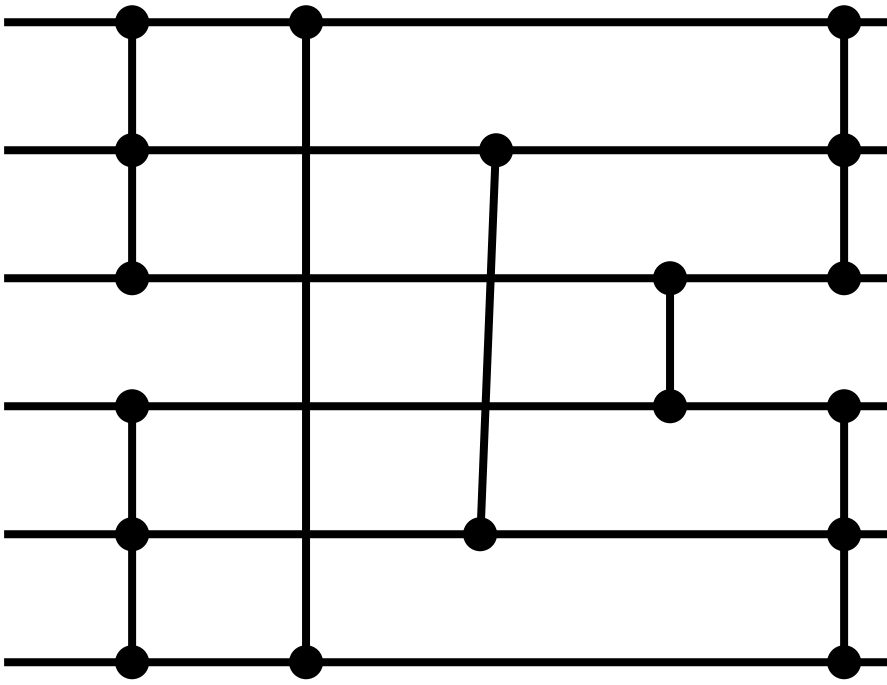

Figure 7: Counting Network using 2 and 3-balancers

A thread that exits with value v spins on filter $[(v-1) \bmod n]$ until that value is set to $\phi(v-1)$. The thread responds by setting filter $[v \bmod w]$ to $\phi(v)$, and then returns v.

1. Explain why this counter implementation is linearizable.
2. An exercise here shows that any linearizable counting network has depth at least w. Explain why the filter [] construction does not contradict this claim.
3. On a bus-based multiprocessor, would this filter [] construction have better throughput than a single variable protected by a spin lock? Explain.

Solution Operations are linearizable in the order the threads stop spinning. If A returns a and B returns b, where $a < b$, then A stopped spinning before B, so A could not have started its operation after B returned.

The $\theta(n)$ lower bound only applies to wait-free networks. Specifically, the proof assumed that one token could pass completely through the network while other tokens were stalled in the middle.

Because threads spin on locally-cached copies, the filter would perform better than a test-and-set lock. The filter is actually a queue lock in disguise.

It is enough to keep track of filter values module 2, because the predecessor is either executing in the same phase (with value $v-1$) or in the previous phase (with value $v-n-1$).

Exercise 148. If a sequence $X = x_0, \ldots x_{w-1}$ is k-smooth, then the result of passing X through a balancing network is k-smooth.

Solution It is enough to show that the result of passing any two elements of X through a single balancer leads to a k-smooth sequence. The result follows by induction on the number of balancers in the network. Let Z be the result of passing two elements x_i and x_j of X through a balancer. Let x and y be the smallest values in X and Y. Suppose a balancer joins x_i and y_j to produce z_i and z_j. Without loss of generality, assume the balancer is in the state where it outputs the first token on wire i.

$$z_i = \left\lceil \frac{x_i + x_j}{2} \right\rceil \qquad \text{and} \qquad z_j = \left\lfloor \frac{x_i + x_j}{2} \right\rfloor .$$

It follows that the smallest element in Z is greater than or equal to the smallest element in X, and the largest element in Z is less than or equal to the largest element in X. Since X is k-smooth, so is Z.

Exercise 149. Prove that the Bitonic$[w]$ network has depth $(\log w)(1+\log w)/2$ and uses $(w \log w)(1 + \log w)/4$ balancers.

Solution Recall that a bitonic network consists of two parallel half-size bitonic networks followed by a merger network. The depth $B(w)$ of a bitonic network is given by the following recurrence:

$$B(2w) = B(w) + M(2W)$$

where $M(w)$ is the depth of the merger network. The merger network consists of two parallel half-size merger networks followed by a single layer:

$$M(2w) = M(W) + 1$$

This recurrence is satisfied by

$$M(w) = \log(w)$$

where the log is base 2. Plugging in these formulas:

$$B(2w) = \frac{\log(2w)(1 + \log(2w))}{2} \tag{0.0.1}$$

$$= \frac{(1 + \log(w))(2 + \log(w))}{2} \tag{0.0.2}$$

$$= \frac{\log(w)(1 + \log(w))}{2} + (1 + log(w)) = B(w) + M(2w). \tag{0.0.3}$$

Exercise 152. Show that the OddEven network in Fig. **??** is a sorting network but not a counting network.

Solution First, we show that the network is a sorting network. By the 0-1 principle, we need to prove only that the network sorts any sequence of 0s and 1s.

We argue by induction on w, the width of the network. Clearly, a single comparator sorts two values.

Assume an odd-even network of with w network sorts w input values. Consider an input sequence X of length $w + 1$. There are two cases to consider. Suppose $X_w = 1$. This input is already in the correct position, and the remaining w gates will sort the remaining inputs correctly. Suppose $X_w = 0$. Every time x_w encounters a comparator, it moves up (by convention, if both inputs are 0). We can eliminate these comparators and cross wires, leading the bottom input wire directly to the top output wire. The remaining wires and comparators form an odd-even network of width w, which sorts its inputs by the induction hypothesis.

To see why the odd-even network is not a counting network, consider a network of width 4. Send three tokens through wire 0. The first token emerges on output wire 0, and the second on output wire 1. The third, however, emerges on wire 0, violating the step property.

```
1  public synchronized int antiTraverse() {
2    try {
3      if (toggle) {
4        return 1;
5      } else {
6        return 0;
7      }
8    } finally {
9      toggle = !toggle;
10   }
11 }
```

Figure 8: The antiTraverse() method.

Exercise 153. Can counting networks do anything besides increments? Consider a new kind of token, called an *antitoken*, which we use for decrements. Recall that when a token visits a balancer, it executes a getAndComplement(): it atomically reads the toggle value and complements it, and then departs on the output wire indicated by the old toggle value. Instead, an antitoken complements the toggle value, and then departs on the output wire indicated by the new toggle value. Informally, an antitoken "cancels" the effect of the most recent token on the balancer's toggle state, and vice versa.

Instead of simply balancing the number of tokens that emerge on each wire, we assign a *weight* of $+1$ to each token and -1 to each antitoken. We generalize the step property to require that the sums of the weights of the tokens and antitokens that emerge on each wire have the step property. We call this property the *weighted step property*.

Fig. 8 shows how to implement an antiTraverse() method that moves an antitoken though a balancer. Adding an antiTraverse() method to the other networks is left as an exercise.

Exercise 154. Let $\mathcal{B}$ be a balancing network in a quiescent state s, and suppose a token enters on wire i and passes through the network, leaving the network in state s'. Show that if an antitoken now enters on wire i and passes through the network, then the network goes back to state s.

Solution We argue by induction on the network depth. If the network depth is 1, then a token followed by an antitoken affects only one balancer, and that balancer is restored to its original state.

Assume the claim for networks of depth d-1. We can split a network of depth d into two parts, the first layer, of depth 1, and a second, of depth $d-1$. Suppose a token entering on wire i exits the first layer and enters the remaining network on wire j. An antitoken entering on wire i) restores the first layer to its

prior state, and also enters the remaining network on wire j. By the induction hypothesis, it restores the remaining network to its prior state.

Exercise 155. Show that if balancing network $\mathcal{B}$ is a counting network for tokens alone, then it is also a balancing network for tokens and antitokens.

Solution This is a hard problem. A full solution appears in "William Aiello, Costas Busch, Maurice Herlihy, Marios Mavronicolas, Nir Shavit, Dan Touitou: Supporting Increment and Decrement Operations in Balancing Networks. Chicago J. Theor. Comput. Sci. 2000: (2000)"

Exercise 156. A *switching network* is a directed graph, where edges are called *wires* and node are called *switches*. Each thread shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire. Here, for simplicity, switches have two input and output wires. Note that switching networks are more powerful than balancing networks, since switches can have arbitrary state (instead of a single bit) and tokens also have state.

An *adding network* is a switching network that allows threads to add (or subtract) arbitrary values.

We say that a token is *in front of* a switch if it is on one of the switch's input wires. Start with the network in a quiescent state q_0, where the next token to run will take value 0. Imagine we have one token t of weight a and n–1 tokens $t_1, \ldots, t_{n-1}$ all of weight b, where $b > a$, each on a distinct input wire. Denote by $\mathcal{S}$ the set of switches that t traverses if it traverses the network by starting in q_0.

Prove that if we run the $t_1, \ldots, t_{n-1}$ one at a time though the network, we can halt each t_i in front of a switch of $\mathcal{S}$.

At the end of this construction, $n-1$ tokens are in front of switches of $\mathcal{S}$. Since switches have two input wires, it follows that t's path through the network encompasses at least $n-1$ switches, so any adding network must have depth at least $n-1$, where n is the maximum number of concurrent tokens. This bound is discouraging because it implies that the size of the network depends on the number of threads (also true for CombiningTrees, but not counting networks), and that the network has inherently high latency.

Solution Let q_0 be the initial state of the switching network, before any tokens have entered. Consider first an execution α_1' starting from q_0 in which t_1 runs by itself ("solo") through the network. Clearly, t_1 takes the value b in α_1'. Assume that t_1 never traverses any switch of $\mathcal{S}$. If we then let t run solo all the way through the network starting from the final state of α_1', it would encounter the same switches in the same internal states as it would starting from q_0, and it would still take value a. But t and t_1 must take either a and $a+b$, or $a+b$

and b. This implies that t_1 traverses at least one switch of $\mathcal{S}$ in α_1'. Take α_1 to be the shortest prefix of α_1' such that t_1 is in front of a switch of $\mathcal{S}$ at the final state q_1 of α_1. Clearly, t_1 does not traverse any switch of $\mathcal{S}$ in α_1.

Consider now an execution fragment α_2' starting from q_1 in which t_2 runs solo all the way through the network. Since t takes2 no steps in $\alpha_1 \cdot \alpha_2'$, t_2 takes either b or $2b$ in α_2'. (Recall that t_1 does not traverse any switch of $\mathcal{S}$ in α_1.) Assume that t_2 never traverses some switch of $\mathcal{S}$ in α_2'. Then, if we let t run solo all the way through the network starting from the final state of α_2', it would encounter the same switches in the same internal states as it would starting from q_0, and it would take value a. However, it is not possible that the three tokens take values a, b, and $2b$. It follows that t_2 traverses at least one switch of $\mathcal{S}$ in α_2'. Take α_2 to be the shortest prefix of α_2' such that t_2 is in front of a switch of $\mathcal{S}$ at the final state q_2 of α_2.

Continuing in this way, we can "park" each t_k in front of a distinct switch in $\mathcal{S}$.

(A formal proof would be expressed in terms of induction on k.)

Exercise 157. Extend the proof of Exercise 156 to show that a *linearizable* counting network has depth at least n.

Solution Consider an n-width counting network, and assume it's linearizable. In the notation of the previous exercise, token t enters on wire 0, and stops. Let $\mathcal{S}$ be the set of balancers that would be visited by t running solo. Let tokens $t_1, \ldots, t_{n-1}$ enter on the remaining wires. By Exercise 156, we can stop each t_1 in front of a unique balancer in $\mathcal{S}$. Since each balancer in $\mathcal{S}$ has a distinct depth, the network must have depth at least $n - 1$.