```
1  wCQ * LHead = <empty wCQ>, LTail = LHead;
2  void Enqueue_Unbounded(void * p)
3    wCQ * ltail = hp.protectPtr(HPTail, Load(&LTail));
     // Enqueue_Ptr() returns false if full or finalized
4    if ( !ltail->next and ltail->Enqueue_Ptr(p, finalize=True) )
5      hp.clear();
6      return;
7    wCQ * cq = alloc_wCQ();                                  // Allocate wCQ
8    cq->init_wCQ(p);                                       // Initialize & put p
9    enqueuers[TID] = cq;                                 // == Slow path (CRTurn) ==
10   for i = 0 .. NUM_THRDS-1 do
11     if ( enqueuers[TID] = null )
12       hp.clear();
13       return;
14     wCQ * ltail = hp.protectPtr(HPTail, Load(&LTail));
15     if ( ltail ≠ Load(&LTail) ) continue;
16     if ( enqueuers[ltail->enqTid] = ltail )
17       CAS(&enqueuers[ltail->enqTid], ltail, null);
18     for j = 1 .. NUM_THRDS do
19       cq = enqueuers[(j + ltail->enqTid) mod NUM_THRDS];
20       if ( cq = null ) continue;
21       finalize_wCQ(ltail);                  // Duplicate finalize is OK since
22       CAS(&ltail->next, null, cq);              // cq or another node follows
23       break;
24     wCQ * lnext = Load(&ltail->next);
25     if ( lnext ≠ null )
26       finalize_wCQ(ltail);                  // Duplicate finalize is OK since
27       CAS(&LTail, ltail, lnext);             // lnext or another node follows
28   enqueuers[TID] = null;
29   hp.clear();
30 bool dequeue_rollback(wCQ * prReq, wCQ * myReq)
31   deqself[TID] = prReq;
32   giveUp(myReq, TID);
33   if ( deqhelp[TID] ≠ myReq )
34     deqself[TID] = myReq;
35     return False;
36   hp.clear();
37   return True;
```

```
38 void finalize_wCQ(wCQ * cq)
39   OR(&cq->Tail, { .Value=0, .Finalize=1 });
40 void * Dequeue_Unbounded()
41   wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
     // skip_last modifies the default behavior for Dequeue on
     // the last element in aq, as described in the text.
42   void * p = lhead->Dequeue_Ptr(skip_last=True);
43   if ( p ≠ last )
44     if ( p ≠ null or lhead->next = null )
45       hp.clear();
46       return p;
47   wCQ * prReq = deqself[TID];                  // == Slow path (CRTurn) ==
48   wCQ * myReq = deqhelp[TID];
49   deqself[TID] = myReq;
50   for i = 0 .. NUM_THRDS-1 do
51     if ( deqhelp[TID] != myReq ) break;
52     wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
53     if ( lhead ≠ Load(&LHead) ) continue;
54     void * p = lhead->Dequeue_Ptr(skip_last=True);
55     if ( p ≠ last )
56       if ( p ≠ null or lhead->next = null )
57         if ( !dequeue_rollback(prReq, myReq) ) break;
58         return p;
59       Store(&lhead->aq.Threshold, 3n − 1);
60       p = lhead->Dequeue_Ptr(skip_last=True);
61       if ( p ≠ last and p ≠ null )
62         if ( !dequeue_rollback(prReq, myReq) ) break;
63         return p;
64     wCQ * lnext = hp.protectPtr(HPNext, Load(&lhead->next));
65     if ( lhead ≠ Load(&LHead) ) continue;
66     if ( searchNext(lhead, lnext) ≠ NOIDX ) casDeqAndHead(lhead, lnext);
67   wCQ * myCQ = deqhelp[TID];
68   wCQ * lhead = hp.protectPtr(HPHead, Load(&LHead));
69   if ( lhead = Load(&LHead) and myCQ = Load(&lhead->next) )
70     CAS(&LHead, lhead, myCQ);
71   hp.clear();
72   hp.retire(prReq);
73   return myCQ->Locate_Last_Ptr();
```

Figure 13: Adapting CRTurn to an unbounded wCQ-based queue design (high-level methods).

A APPENDIX: UNBOUNDED QUEUE

LSCQ and LCRQ implement unbounded queues by using an outer layer of M&S lock-free queue which links ring buffers together. Since operations on the outer layer are very rare, the cost is dominated by ring buffer operations. wCQ can follow the same idea.

Although the outer layer does not have to be performant, it still must be wait-free with bounded memory usage. However, M&S queue is only lock-free. The (non-performant) CRTurn wait-free queue [37, 39] does satisfy the aforementioned requirements. Moreover, CRTurn already implements wait-free memory reclamation by using hazard pointers in a special way. wCQ and CRTurn combined together would yield a fast queue with bounded memory usage.

Because CRTurn's design is non-trivial and is completely orthogonal to the wCQ presentation, its discussion is beyond the scope of this paper. We refer the reader to [37, 39] for more details about CRTurn. Below, we sketch expected changes to CRTurn (assuming prior knowledge of CRTurn) to link ring buffers rather than individual nodes. In Figure 13, we present pseudocode (assuming that entries are pointers) with corresponding changes to enqueue and dequeue operations in CRTurn. For convenience, we retain the same variable and function names as in [37] (e.g., *giveUp* that is not shown here). Similar to [37], we assume memory reclamation API based on hazard pointers (the *hp* object).

The high-level idea is that we create a wait-free queue (list) of wait-free ring buffers, where *LHead* and *LTail* represent corresponding head and tail of the list. *Enqueue_Unbounded* will first attempt to insert an entry to the last ring buffers as long as *LTail* is already pointing to the last ring buffer. Otherwise, it allocates a new ring buffer and inserts a new element. It then follows CRTurn's procedure to insert the ring buffer to the list. The only difference is that when helping to insert the new ring buffer, threads will make sure that the previous ring buffer is finalized.

Dequeue_Unbounded will first attempt to fetch an element from the first ring buffer. wCQ's Dequeue for **aq** needs to be modified to detect the very last entry in a *finalized* ring buffer. (Note that it can only be done for finalized ring buffers, where no subsequent entries can be inserted.) Instead of returning the true entry, *Dequeue_Ptr* returns a special **last** value. This approach helps to retain CRTurn's wait-freedom properties as every single ring buffer contains at least one entry. Helper methods must also be modified accordingly.