Nonblocking Concurrent Data Structures

CSC 2/458

March 2026

Michael L. Scott

Why not just use locks?

- In theory: thread failure. If thread *t* dies while holding a lock on object *O*, no thread that needs to use *O* will be able to make progress.
  - » NB: threads that share data usually die together, so this hasn't mattered much in practice. (Recent developments may change that; stay tuned.)
- In practice: (1) preemption. If *t* is preempted while holding *O*'s lock, no thread that needs *O* will be able to make progress until *t* runs again. That can really mess up performance, especially when there are more threads than cores.
- And (2) priority inversion: if a low priority thread holds a lock and is interrupted by an interrupt or signal handler that needs the lock, deadlock can occur.

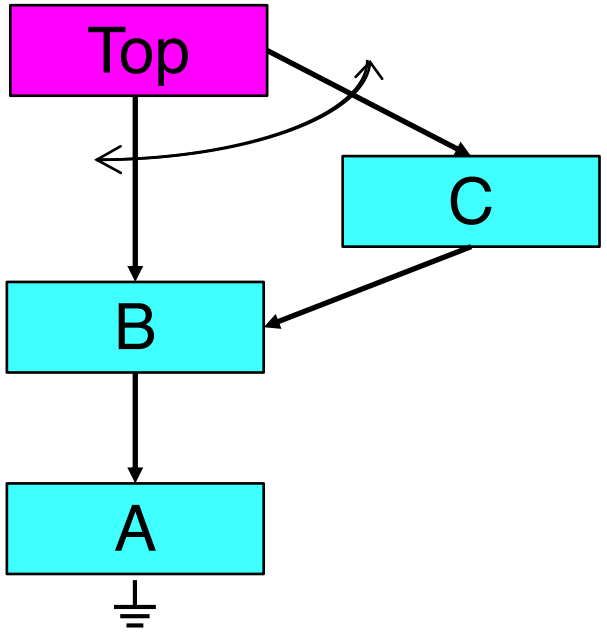
Enter nonblocking structures

- Key idea: every reachable concrete state of the data structure ("object") corresponds to a unique abstract state, and can serve as the starting point for any operation (or the continuation of any operation already underway).
- Typical (but not universal) pattern:
  - Harmless preparation
  - Linearizing instruction
  - Post-op clean-up, which any thread can help

Topics

- Simple objects: stacks, queues, lists
  - The ABA problem
  - Dynamic memory management
- More complex objects: hash tables, skip lists, trees, …
- Advanced topics (as time permits): universal constructions, duals, combining, persistence, …

Recall the Treiber stack



Push

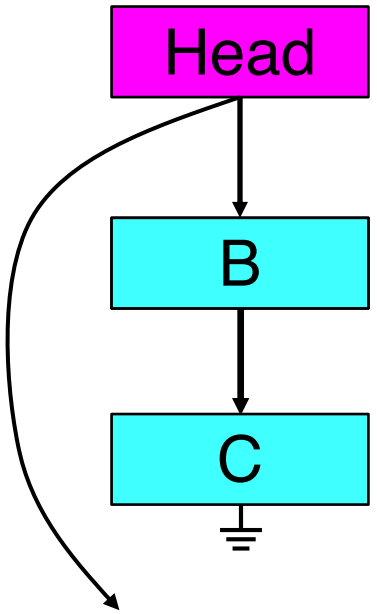
- allocates new node
- sets next pointer
- installs with CAS
- retries as needed

Pop

- reads head→next
- removes with CAS
- retries as needed
- deletes old node

And the ABA problem

- Consider concurrent operations



- Thread 1 starts a pop: reads head→next but hasn't yet tried CAS
- Thread 2 completes
  - pop(A)
  - push(B)
  - push(A) *// same addr*
- Thread 1 does its CAS, which ought to fail but doesn't
- At this point we've lost node B

ABA solutions

- Use LL/SC, if available
- Use *counted pointers* (or other values)
  - ⟨sn, val⟩ pair
  - increment sn on every change
  - requires double-wide CAS
- When (and only when) the problem case includes memory reuse, it can also be solved with *safe memory reclamation* (stay tuned)

Recall correctness

- Safety: “bad things never happen”
  - » ∀ reachable states of the system, invariants hold
  - » *Linearizability* the standard criterion
- Liveness: “good things eventually happen”
  - » ∀ reachable states ∃ a forward execution such that…
  - » Several variants of *nonblocking progress*

Linearizability [Herlihy & Wing 1987]

- An execution history H is linearizable if it is equivalent (same calls and returns, with same argument values) to some sequential execution S that respects
  1. object semantics (e.g., pop returns the most recently pushed value, if any, that has not yet been returned by a pop, or ⊥ if there is none)
  2. "real-time" order (return(A) $<_H$ call(B) $\Rightarrow$ A $<_S$ B)
     (subsumes per-thread program order)
- Equivalently, each operation must appear to happen instantaneously at some point between its call and its return
- An object implementation is linearizable if all realizable histories are linearizable

- Second definition suggests a proof technique: identify a *linearization point* in each operation at which it seems to occur
  - » In simple cases (e.g., atomic counter), this may be statically known
  - » In many other cases (e.g., Treiber stack), it may be dynamically selected based on run-time conditions
  - » In tricky cases, may require retrospective reasoning over histories
- ★ Linearizability is *composable* (*local*): if structures A & B are both linearizable, then all histories using both A & B will be linearizable

Code for the Treiber stack

```
class stack
    <node*, int> top

void stack.push(node* n):
    repeat
        <o, c> := top
        n⟶next := o
    until CAS(&top, <o, c>, <n, c>)
```

```
node* stack.pop():
    repeat
        <o, c> := top
        if o = null return null
        n := o⟶next
    until CAS(&top, <o, c>, <n, c+1>)
    return o
```

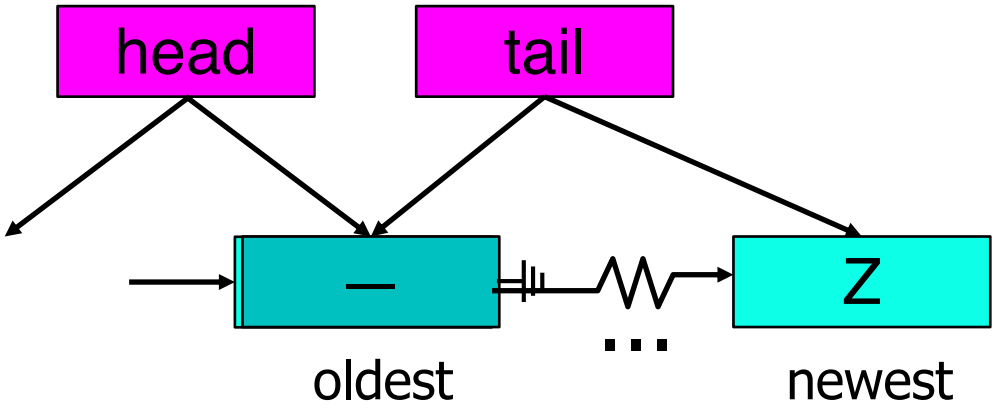
- Push linearizes at the final successful CAS
- Pop linearizes at the read of top if null; otherwise at the final successful CAS

Liveness

- Three widely accepted classes of nonblocking progress guarantee:
  - Wait-free (= starvation-free) : my operation is guaranteed to complete in a bounded number of my steps
  - Lock-free (= livelock-free): *somebody's* operation is guaranteed to complete in a bounded number of my steps
  - Obstruction-free(= [merely] nonblocking): my operation is guaranteed to complete in a bounded number of steps if I get to run all by myself

Michael & Scott queue [1996]

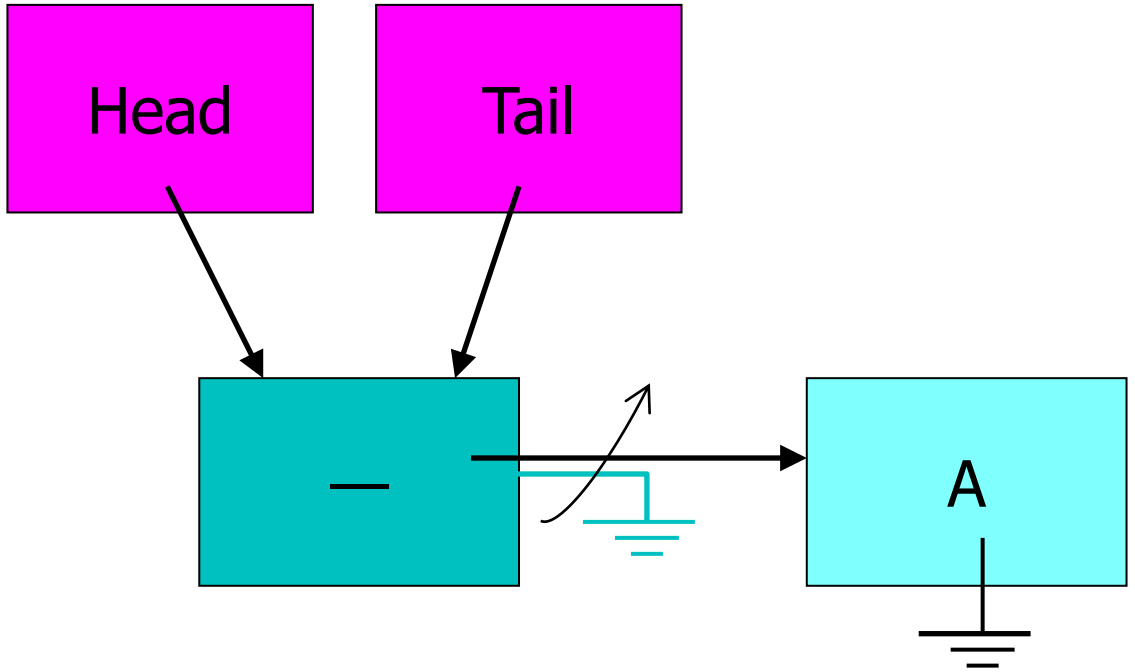
- Empty queue consists of head and tail pointers to a *dummy* node



- If nonempty, first node is still dummy.
- Enqueue adds at tail; dequeue removes at head

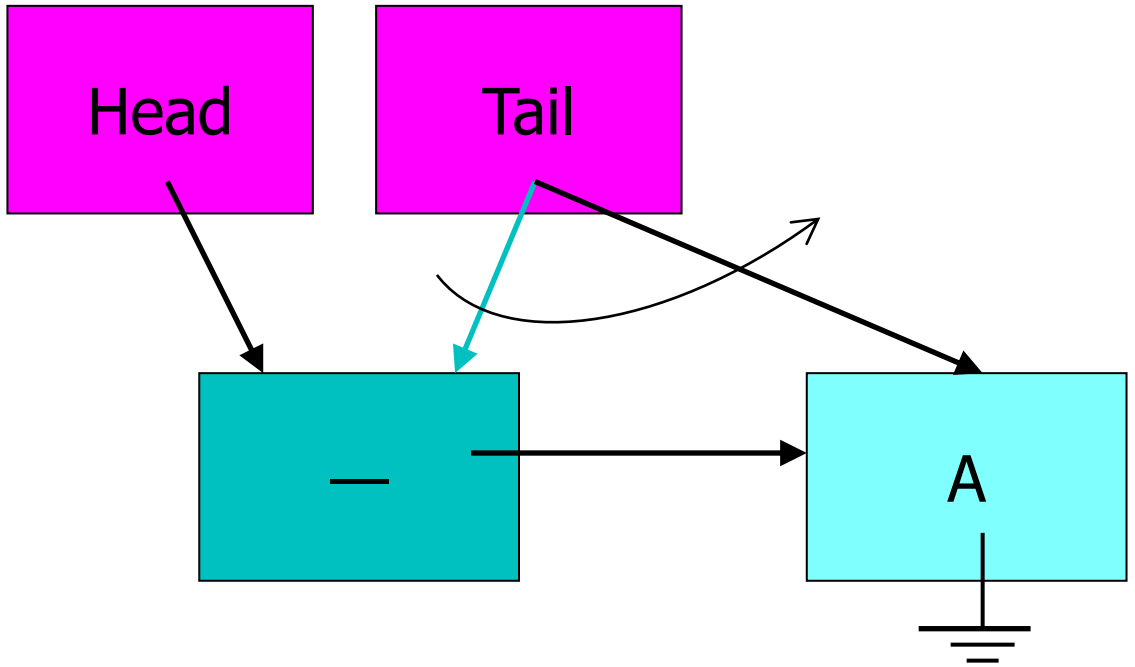
M&S queue: enqueue

1) CAS next pointer of tail node to new node
2) Use CAS to swing tail pointer



M&S queue: enqueue

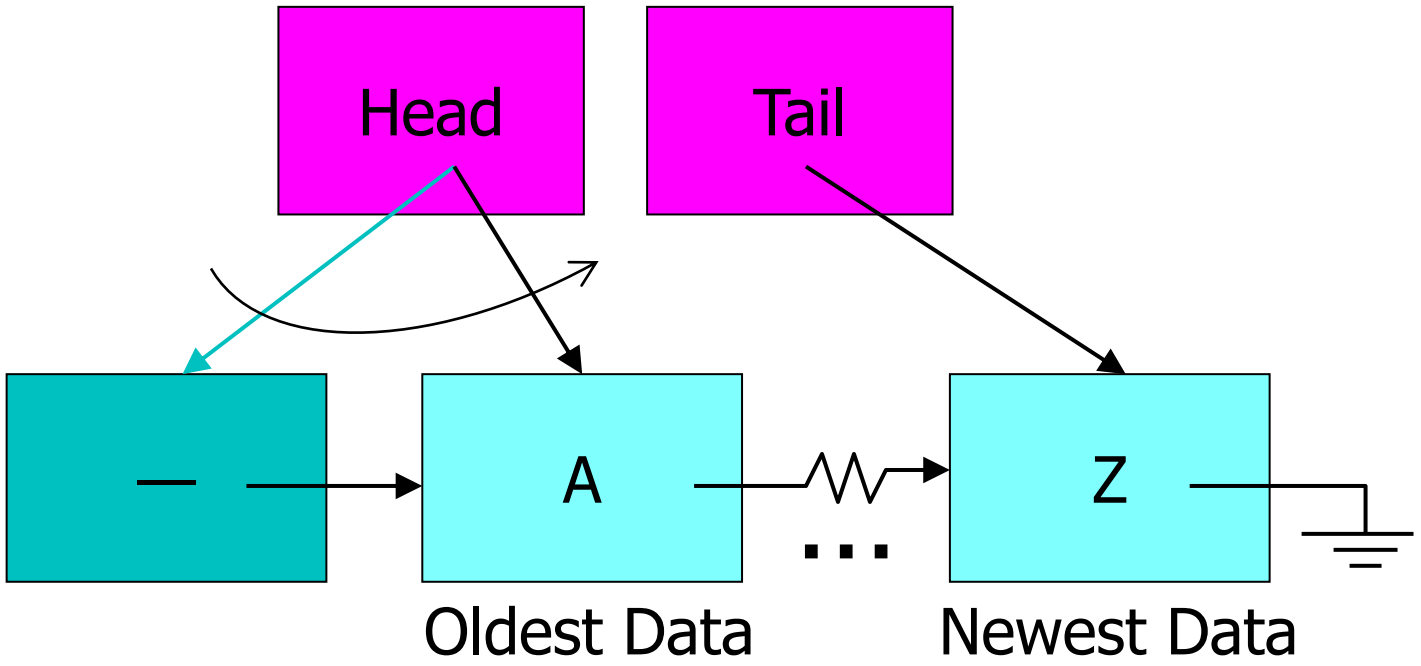
1) CAS next pointer of tail node to new node
2) Use CAS to swing tail pointer (any thread can *help*)



(Helping is a recurring general technique in nonblocking structures.)

M&S queue: dequeue

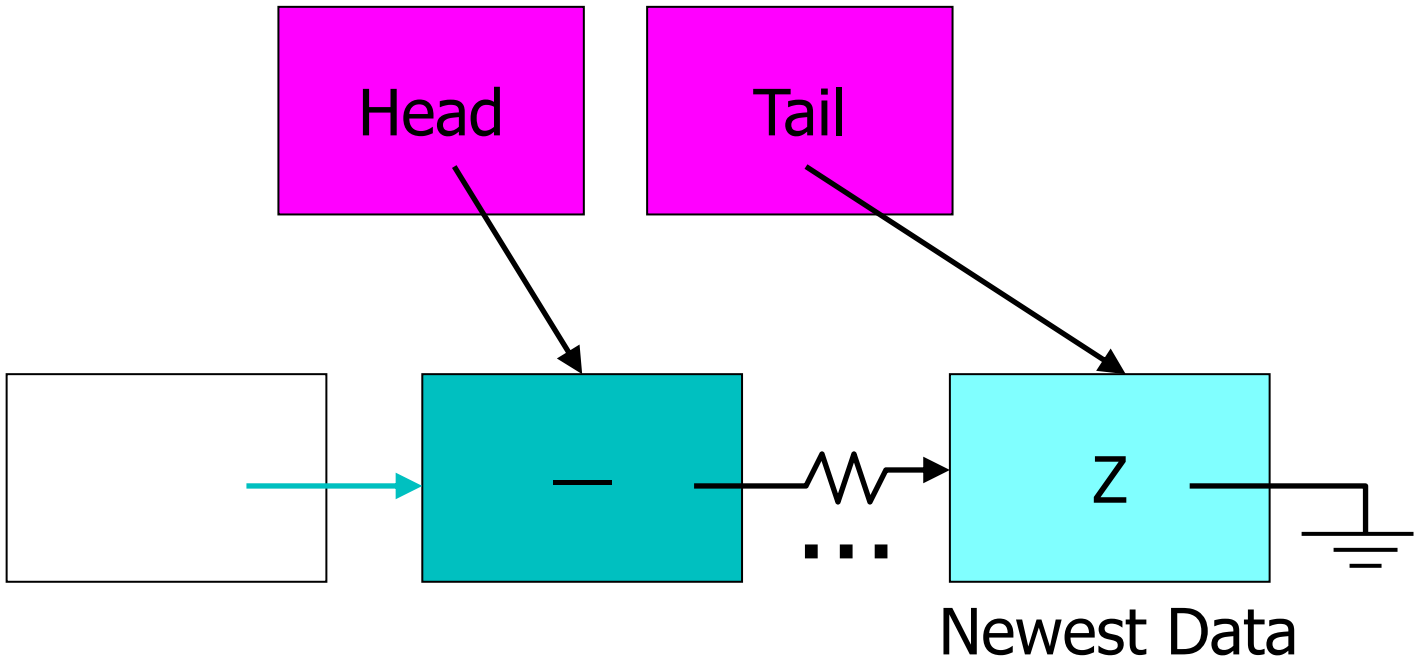
1) Read data in dummy's next node
2) CAS head pointer to dummy's next node



M&S queue: dequeue II

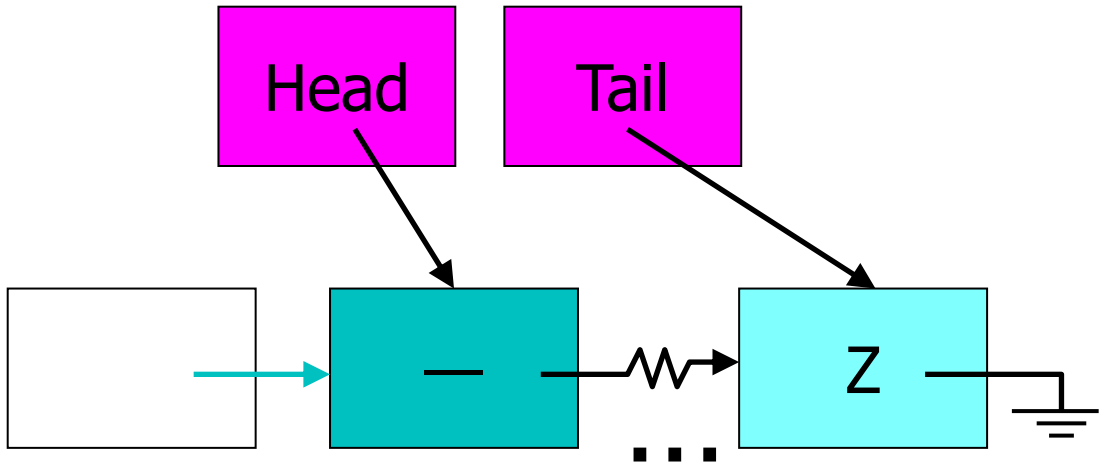
3) Discard old dummy node

4) Node from which we read is new dummy



Problem: freeing nodes

- Suppose
  - » T1 reads Head
  - » T2 removes and frees the dummy node; then reuses it for another purpose
  - » Then T1 tries to read dummy→next
- Could seg-fault or worse

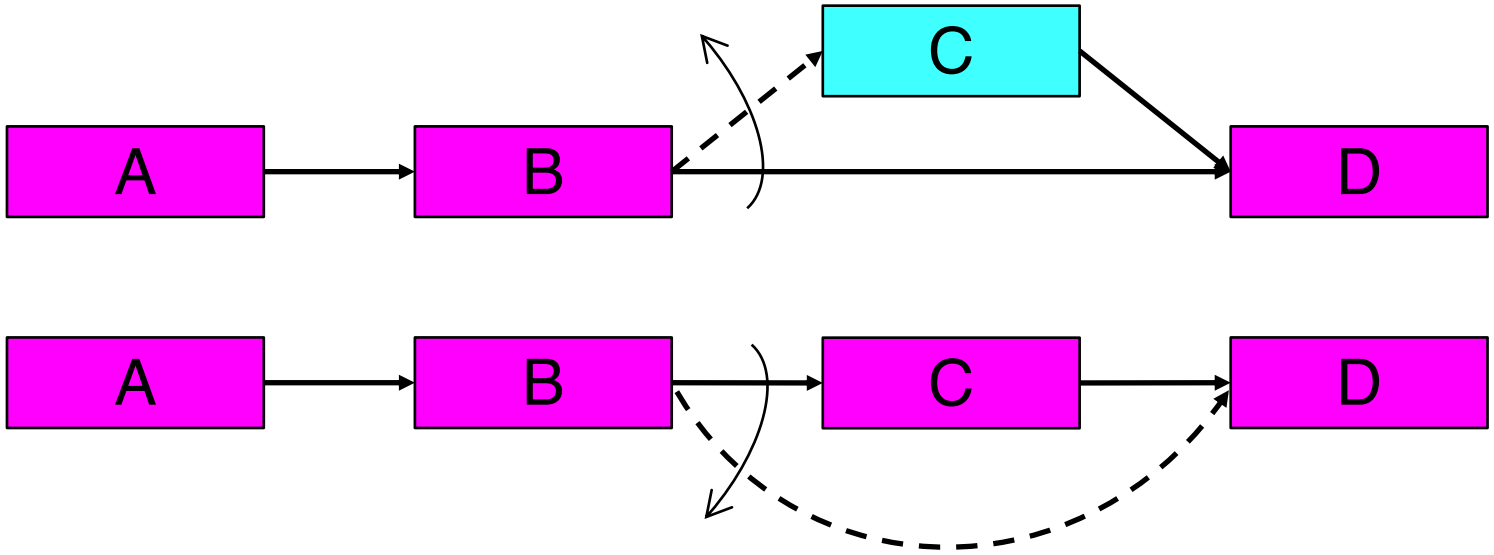


Possible solution

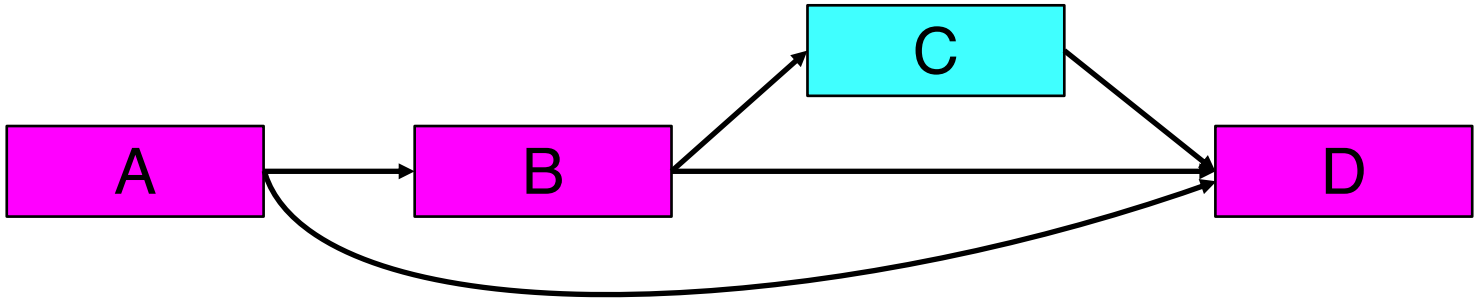
- Head and tail are counted pointers
- Enqueue takes a *consistent snapshot*:
  - » reads head, then head→next, then val
  - » Subsequent CAS works only if head has not changed
  - » head→next and val cannot change without head changing first
- Nodes employ a *type-preserving allocator*
  - » Any memory used for a queue node will never be returned to the operating system or used for any other type
  - » When I read head→next, I know it's a queue node pointer, so I'll never seg-fault; if the node has been deleted, the CAS on head is guaranteed to fail

Harris [2001] & Michael [2002] singly linked lists

- Tempting to think we can insert or remove with a single CAS:



- But what happens with simultaneous insert and remove?



T1: // to insert C

create new node; point at D

read B→next

T2: // to delete B

read A→next

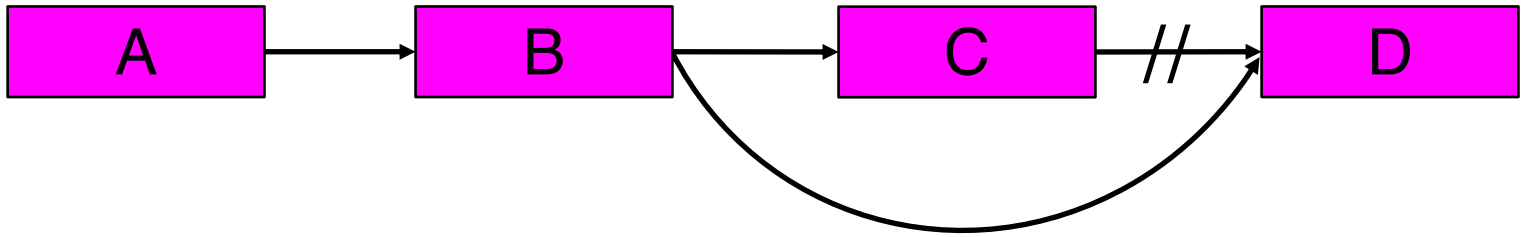
CAS A→next to point to D

CAS B→next to point to C

- Now C has been lost!

Harris's insight

- Delete in *two steps*



  - *Tag* the next pointer of the to-be-deleted node (This is the linearization point)
  - Then remove with CAS (This is cleanup)
- Tag forces a conflict with any concurrent insert
  - Can't insert after C if C is tagged
  - *Any* thread can remove C once it's tagged
  - Can do so *lazily* & *batched* (given garbage collection)

Michael's insight

- We don't need garbage collection; manual reclamation can be made to work
  - » Can employ counted pointers & a type-preserving allocator (as in the M&S queue) or *hazard pointers* (stay tuned)
  - » After reading n→next, double check predecessor to make sure n is still its successor
- Subtlety: can no longer delete lazily—specifically, cannot traverse a deleted (tagged) node, because we might accidentally hop lists

Hazard Pointers [Michael; Herlihy et al. 2002]

- General technique for safe manual reclamation in nonblocking structures
  - » Supported by numerous libraries and systems
- Global array indexed by thread
  - » Every thread publishes the addresses of any nodes on which it's currently working
  - » A thread that deletes a node puts it in a *limbo list* until no hazard pointer points at it
- A HP can be created safely only if the node is still linked into the structure
  - » This is why we can't do lazy reclamation

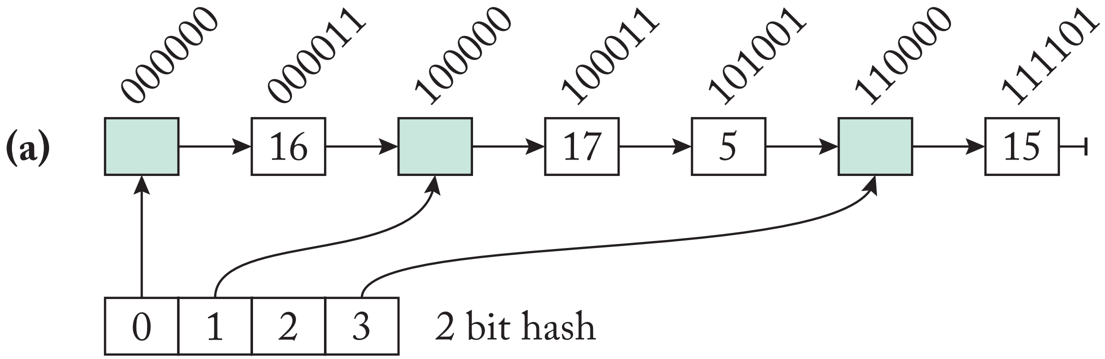
More on memory management

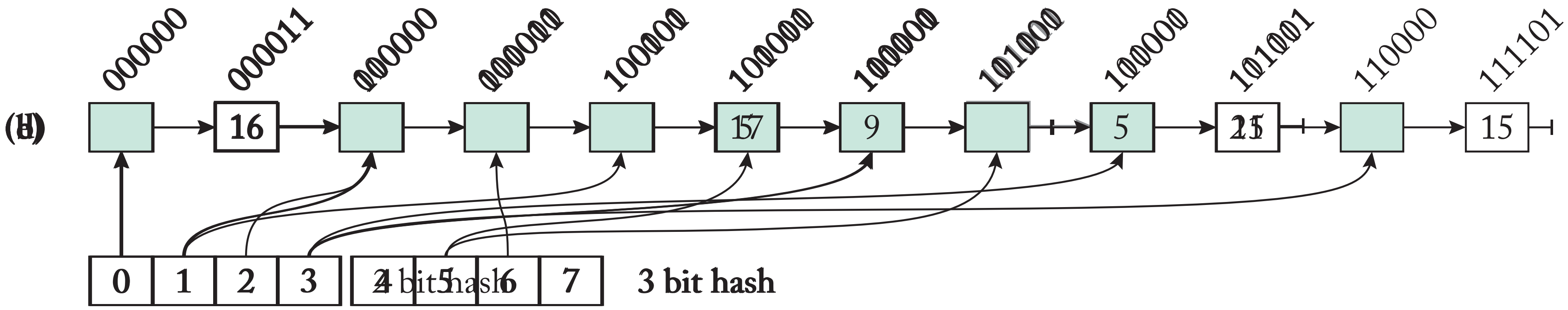
- Reclamation is easy with locks or a tracing garbage collector (which probably needs locks)
- Counted pointers & type-preserving allocation are not general purpose and need wide CAS
- Hazard pointers are general & portable , but
  - awkward for dynamic numbers of threads and pointers
  - expensive: release fence after every HP update
- ∃ several alternatives, inc.
  - Epoch-based reclamation [Fraser 2003; others] — cf. RCU
  - Hazard eras [Ramalhete & Correia, 2017]
  - Interval-based reclamation [Wen et al., 2018]

Hash Tables

- Straightforward to use H&M lists for hash chains of a fixed-size table [Michael, 2002]
- Open-address (no-chain) fixed-size table due to Purcell & Harris [2005]
- Chained resizable table due to Shalev & Shavit [2006]
  - » Single linked list contains all nodes, sorted by bit-reversal of hash code
  - » All ops linearize on list insertion
  - » Resizable index provides hints for where to start searching in list
  - » Clever ordering allows index to be built incrementally; even resizing is O(1)

S&S hash table





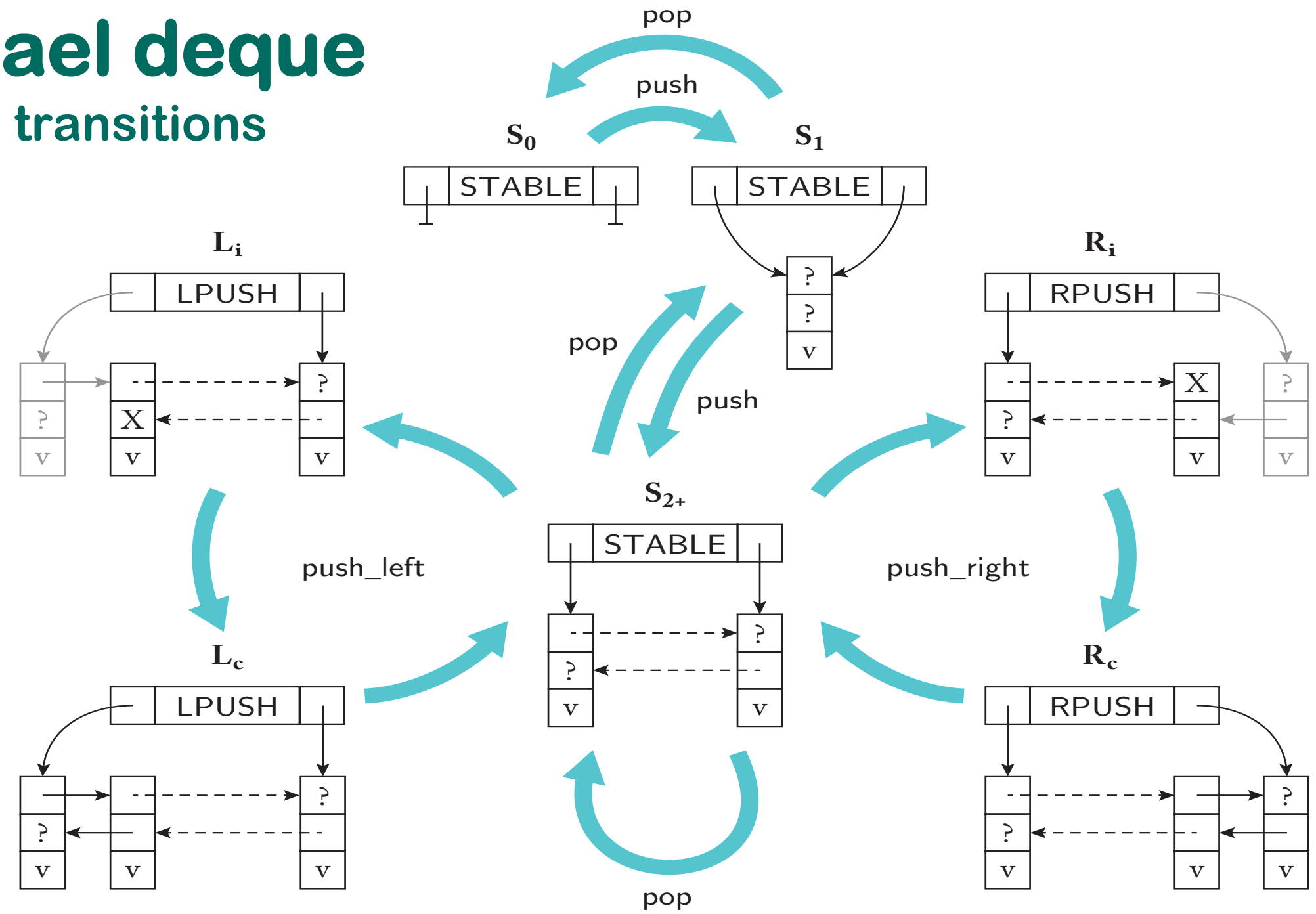
- [illegible]

Double-ended queues

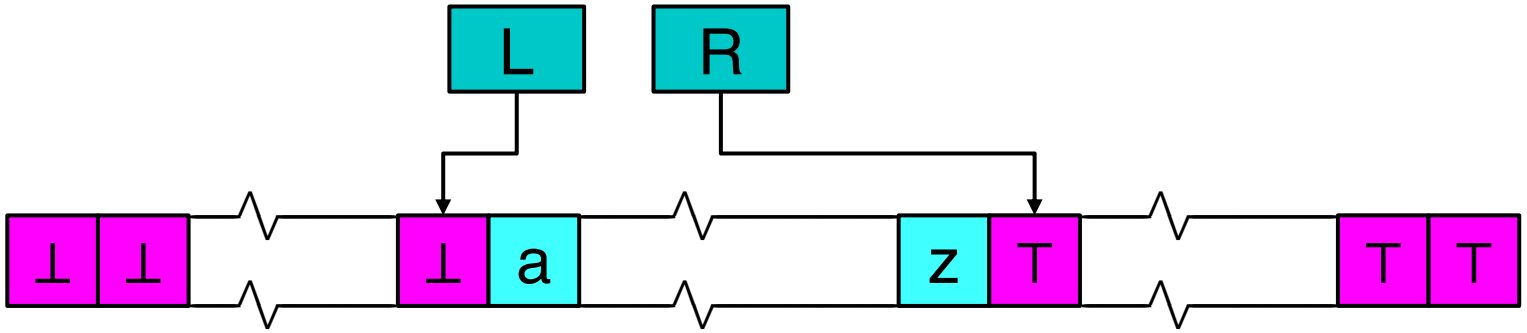
- No known practical applications, but elegant
- Michael [2003]: Lock-free, unbounded, complex
  - Double-wide CAS; 3-CAS updates
- Herlihy, Luchangco, and Moir [2003]
  - Original illustration of obstruction freedom
  - Bounded, elegant
- Graichen, Izraelevitz, and Scott [2016]
  - Natural unbounded extension of HLM
  - Employs a doubly-linked chain of bounded deques
- Work-stealing deques: use 3 of 4 ops

Michael deque

3-stage transitions



HLM deque — obstruction freedom

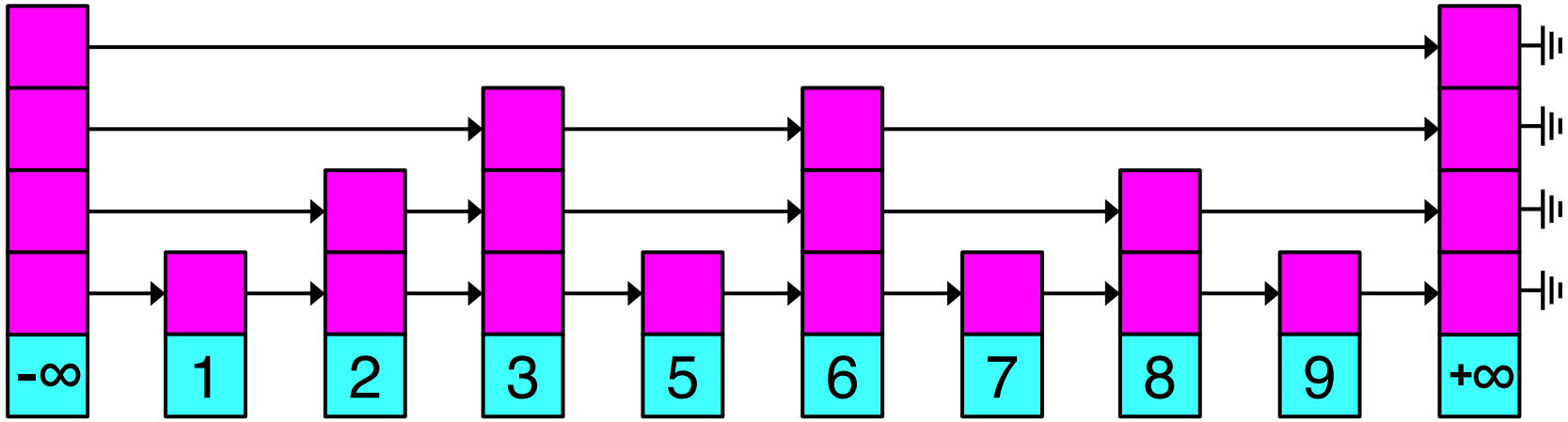


- Circular array, drawn straight here for clarity
  - » Circular version employs 0 or 1 "—" nodes between ⊥ & T
- L and R are hints, lazily updated
- Elements are ⟨count, value⟩ pairs
  - » Each op does 2 CASes: increment neighbor count; update to-be-changed node
  - » Conflicting operations detect each other; may force each other to restart (admitting livelock)

Skip lists

- First nonblocking version due to Fraser [2003]
  - » Based on H&M lists
  - » Refinements by Lea (java.util.concurrent) and by Herlihy, Lev, and Shavit [2007]; sketch the latter here
- Alternative due to Formitchev & Ruppert [2004]
  - » Based on refinement of H&M lists that uses “backlinks” to recover from contention locally, without restarting at the head of the list

Lock-free skip list

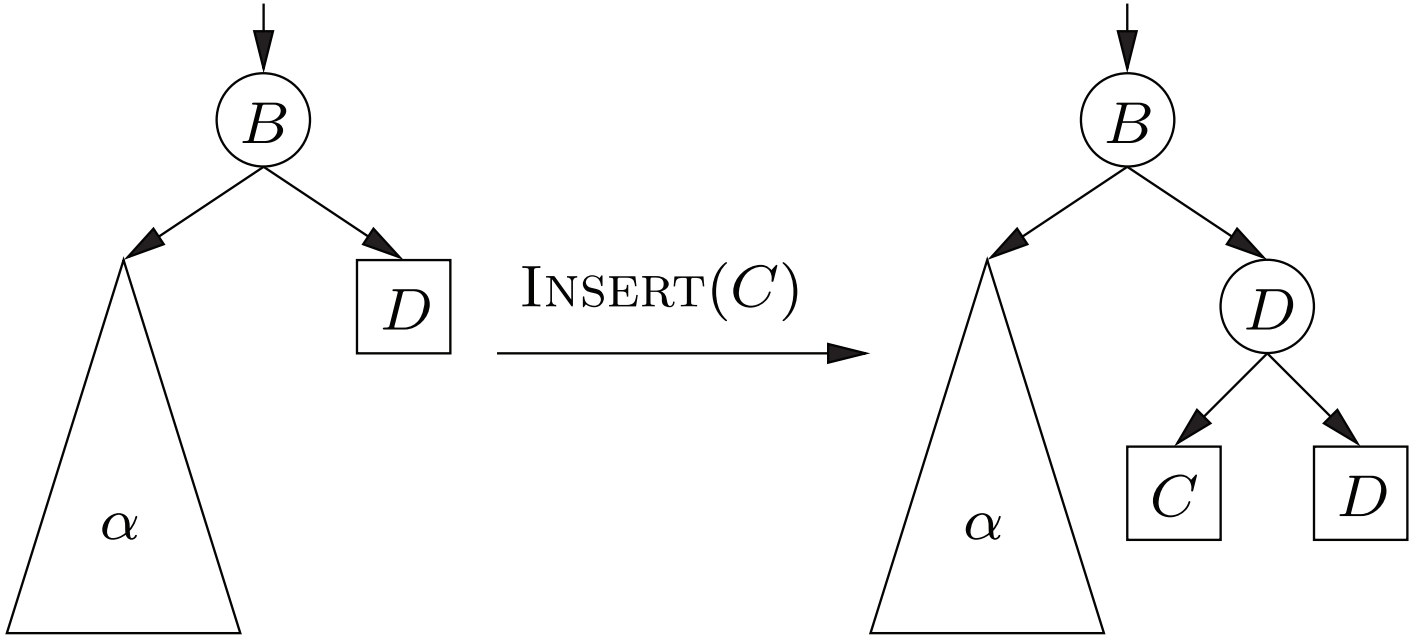


- Each level is an H&M list
  - » Bottom level is authoritative; changes there are linearizing
  - » Upper levels are just hints: they may not always be subsets of lower levels
- Insert works bottom-up; delete works top-down
- Lookup is wait free

Trees

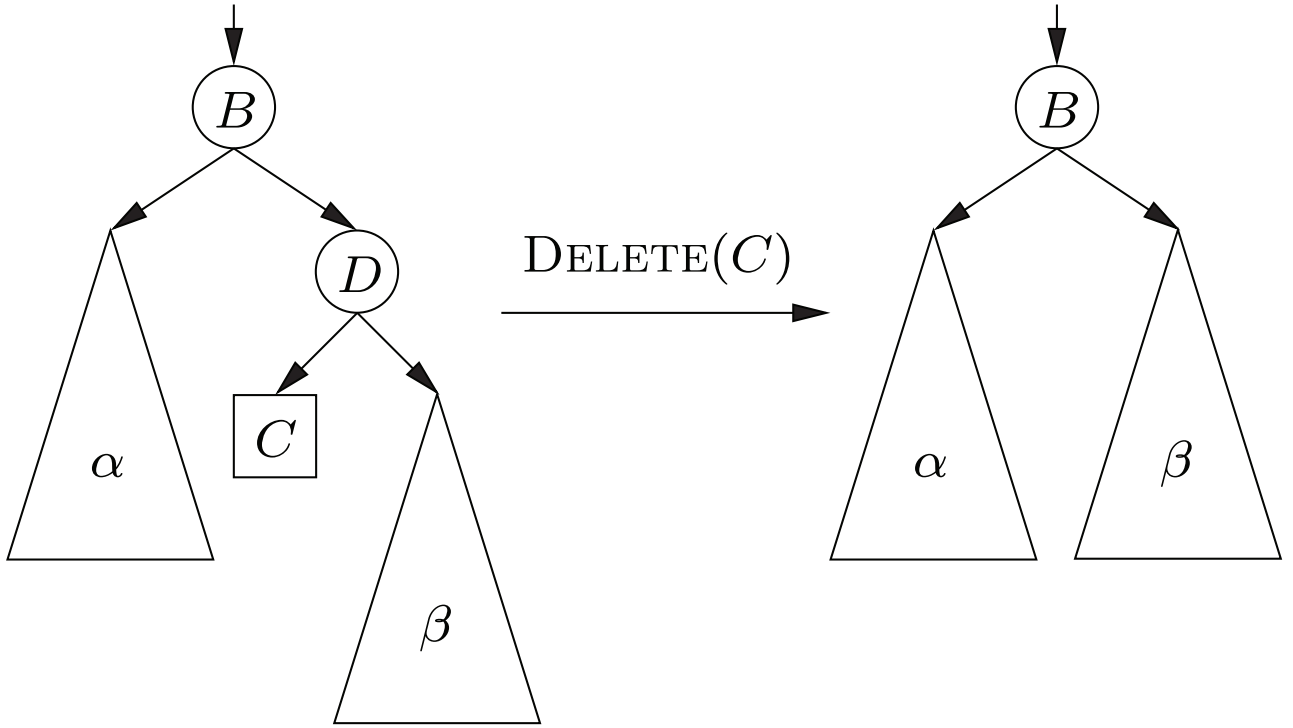
- First practical nonblocking BST due to Ellen et al. [2010]
  - External (all data in leaves), lock-free
  - Extended to k-ary trees by Brown & Helga [2011]
- Internal lock-free BSTs by Howley & Jones [2012]
- Faster external trees by Natarajan & Mittal [2014] (edge marking)
- Wait-free balanced (RB) trees by Natarajan et al. [2013]; lock-free simplification [2013]
- Many other extensions and variants; will sketch the original Ellen et al. algorithm here

Ellen et al.: Insert C



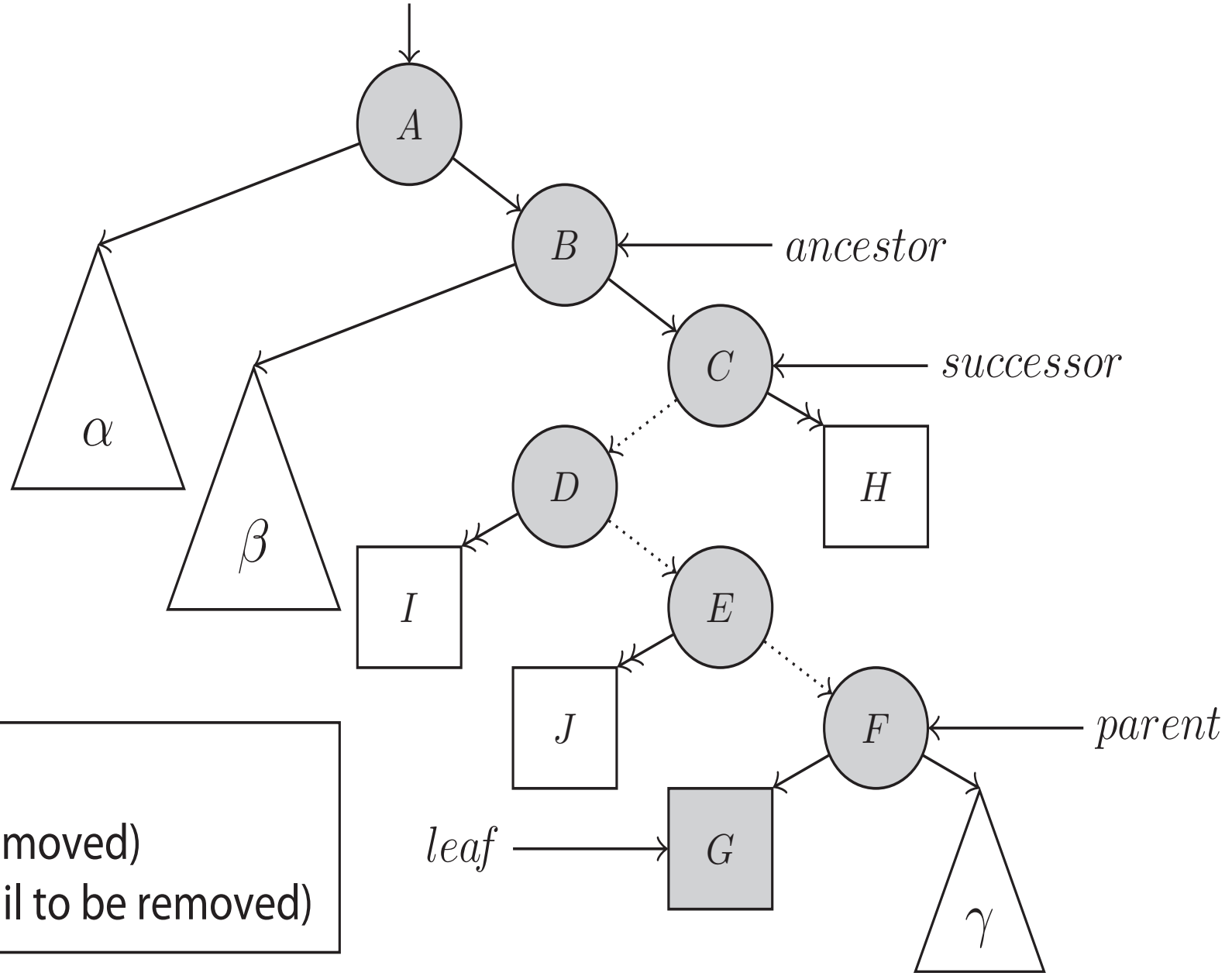
- Three-CAS process
  - » Flag grandparent B as locus of insertion, with an *info record* that explains to other threads how they can help
    - – Operation becomes inevitable at this point
  - » Swap in the new subtree
    - – This is the linearization point
  - » Unflag B

Delete C



- Four-CAS process
  - » Flag grandparent B as locus of deletion, with an *info record*
  - » Mark D as frozen
    - – Back out, help, and retry on failure
  - » Swap in the new subtree β
    - – This is the linearization point
  - » Unflag B

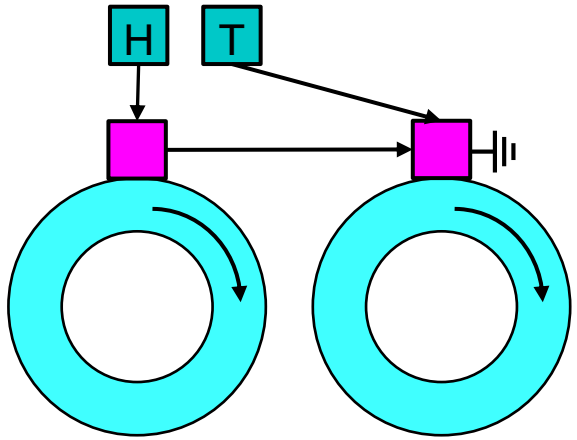
Natarajan & Mittal tree
(edge marking)



unmarked edge
flagged edge (tail to be removed)
tagged edge (head and tail to be removed)

The LCRQ [Morrison & Afek [2013]

- Recall that fetch_and_increment (FAI) always succeeds, while CAS may need to retry
- ∃ circular bounded queues based on FAI that are very fast, but blocking
- Morrison & Afek showed how to make these nonblocking and unbounded by linking together a chain of circular buffers
  - » Very complex — lots of corner cases
  - » ~10x faster than the M&S queue
  - » Dual versions by Izraelevitz & Scott [2017]



Universal Constructions

- Herlihy [1991, 1993] gave automatic techniques to turn any correct sequential object into a correct nonblocking concurrent object
  - » Slow, but not intended to be practical
- Shavit & Touitou [1995] introduced nonblocking *software transactional memory (STM)*
  - » Several subsequent systems provide lock-free or (more commonly) obstruction-free STM; these are not only universal but also *composable*
- Kogan, Petrank, and Timnat [2012, 2014] showed how to turn a lock-free structure into a fast wait-free structure (space overhead remains high)

Duals

- A nonblocking structure (e.g., queue) typically returns null (or ⊥) if the structure is empty
- If you *need* an element you can retry:

```
do {
    t = q.dequeue()
} while (t == ⊥)
```

  But this wastes time and causes contention.

- A better approach is to *insert* a *reservation* and wait on that instead
  - » This can still be nonblocking (see extra slides)

Combining

- In general, combining serves to reduce contention by pairing up matching ops without central coordination
  - » Stacks the canonical example: push & pop can satisfy each other *anywhere* in the sequential history
- *Flat combining* [Hendler et al., 2010] trades concurrency for locality & combining
  - » Threads take turns being the *combiner*, merging other threads' updates & applying to the structure
  - » Performance benefits from reductions in both total work and cache misses on the critical path

Persistence

- Active current topic of research
- Leverages emerging nonvolatile memory
- Addresses the crash consistency problem
  - » Caches are volatile; they lose their content on a crash
  - » They also write back in unpredictable order
  - » What was written back before a crash may not be consistent
- *Durable linearizability* [Izraelevitz et al., 2013] demands that a prefix of happens-before be visible after a crash
  - » Need to add explicit write-back and fence instructions
  - » Lots of special and general-purpose ways to do so

www.cs.rochester.edu/research/synchronization/

www.cs.rochester.edu/u/scott/

- Liveness examples
  - A fetch_and_add-based counter is wait free.
  - A CAS-based counter is lock free, as is the Treiber stack.
  - The seminal obstruction-free structure is the deque of Herlihy, Luchangco, & Moir [2003]. Several software transactional memory systems are also obstruction free.
- In practice:
  - Most published nonblocking structures are lock free.
  - Obstruction-free structures can sometimes be elegantly simple.
  - Wait-free structures are often (but not always!) quite complex. They may be useful for real-time apps, or for structures with large read-only operations.

Dual Data Structures

Michael L. Scott

www.cs.rochester.edu/research/synchronization/

Joint work with William N. Scherer, Doug Lea, and Joseph Izraelevitz

What if a concurrent container—e.g., a queue—is empty?

- With locks, a dequeuing thread can *wait* — tell the scheduler to put it to sleep and release the lock
  - » Some later enqueuing thread, while holding the lock, will tell the scheduler to make the sleeping thread runnable again
- In a nonblocking queue, this wasn't traditionally possible
  - » Does it even make sense to wait (block?) in a nonblocking structure?

The traditional approach

- Dequeue on an empty queue *fails* immediately
- Calling thread must *spin*:

```
do {
    t = q.dequeue()
} while (t == null)
```

- This works but with
  - high contention
  - no guarantee of fairness (waiting threads can succeed out of order)

Dual data structures

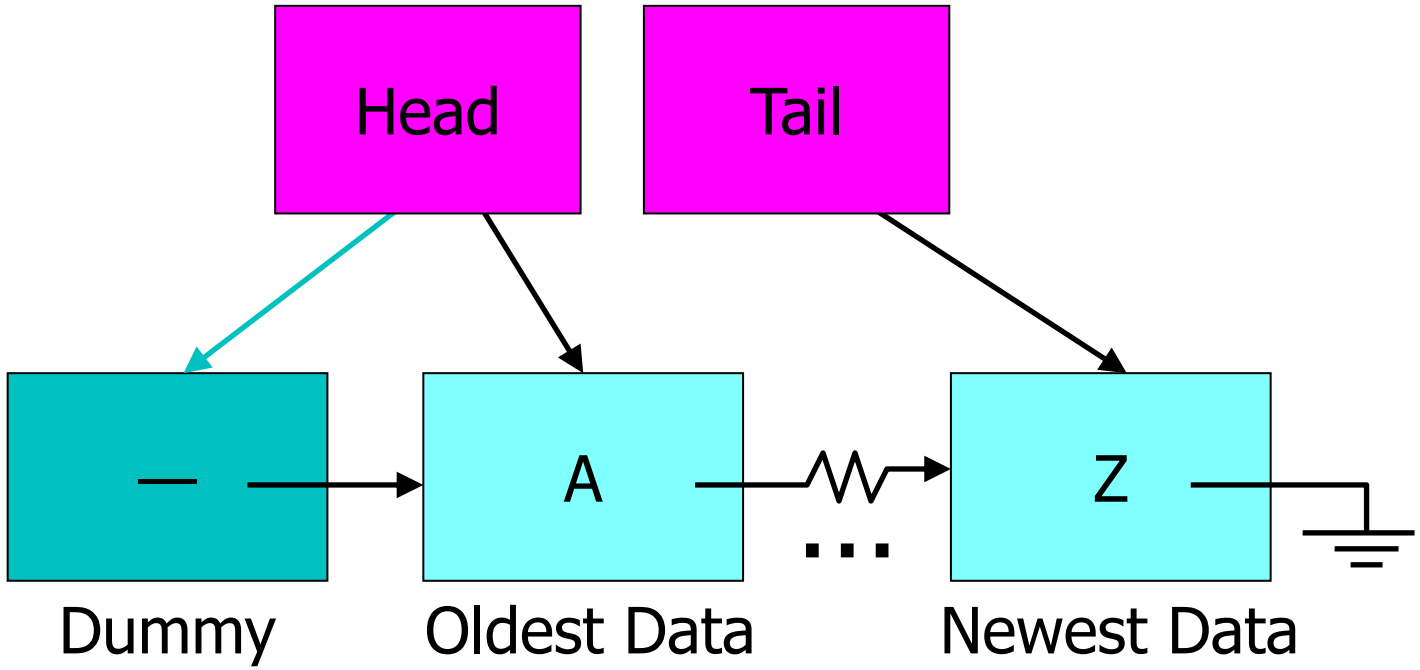
- Data structure holds data or *reservations*
- Dequeuer (in general, consumer) removes data or inserts reservation
- Enqueuer (in general, producer) inserts data or removes and *satisfies* reservation
- Data structure controls which reservation to satisfy — guaranteeing fairness
- Developed dual stack, queue, synchronous variants; exchanger; LCRQ; generic construction
  - » focus here on the queue

The dualqueue

- When trying to dequeue from an empty queue, enqueue a reservation instead
- When enqueuing, satisfy a reservation if present
- Mark pointers to the reservation nodes with a “tag” bit in the pointer
  - Can (mostly) tell queue state from tail pointer
  - Easy in C; requires extra indirection in Java
- Symmetry between enqueues and dequeues
  - Enqueue adds data or removes a reservation
  - Dequeue removes data or adds a reservation

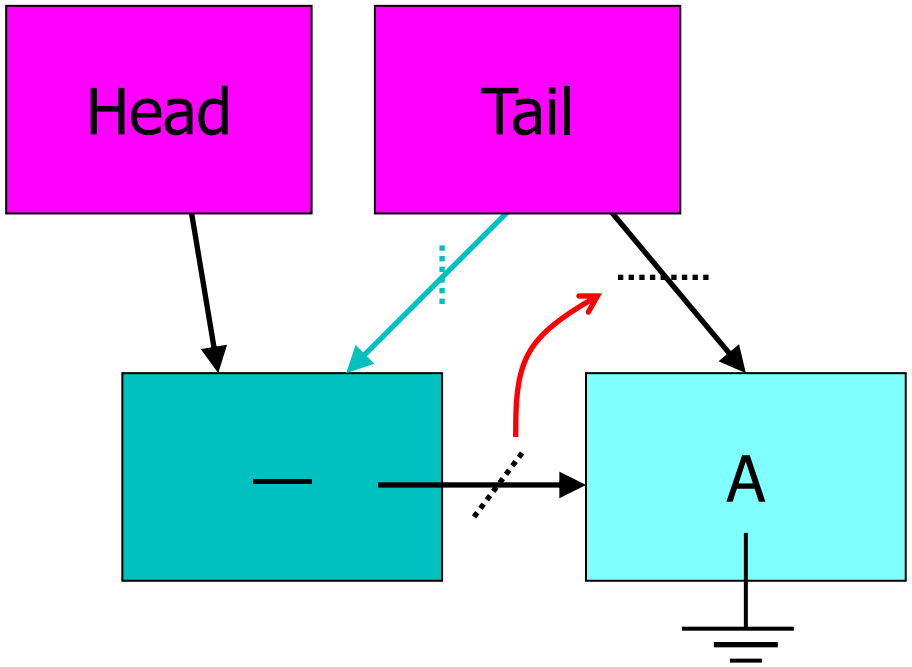
Dualqueue: dequeue

1) Check for queue “empty” or full of reservations
2) If neither, try to dequeue data as before



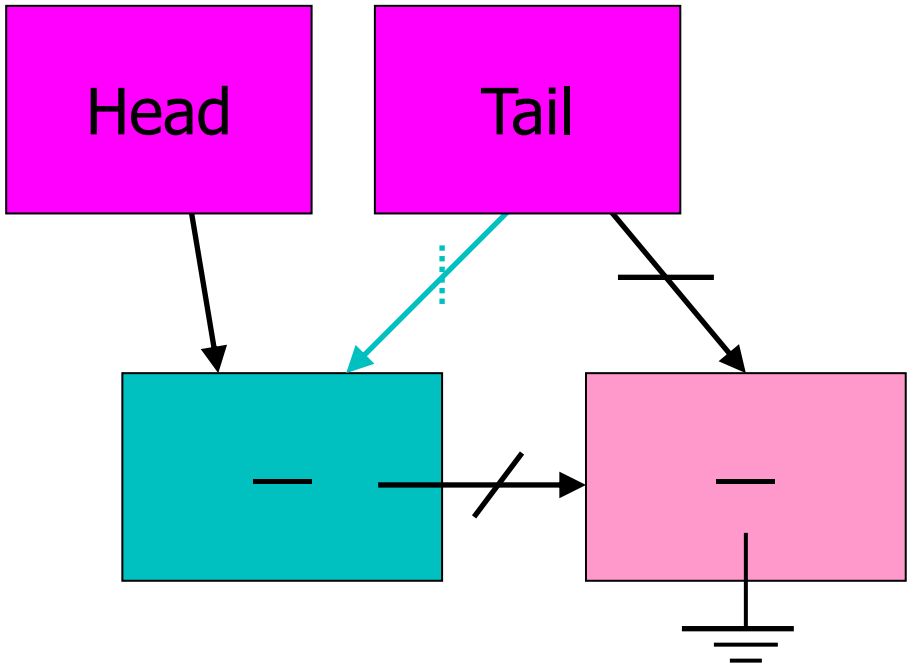
Dualqueue: dequeue II

3) If tail pointer is lagging, swing it and restart. Match tag of tail node's next pointer



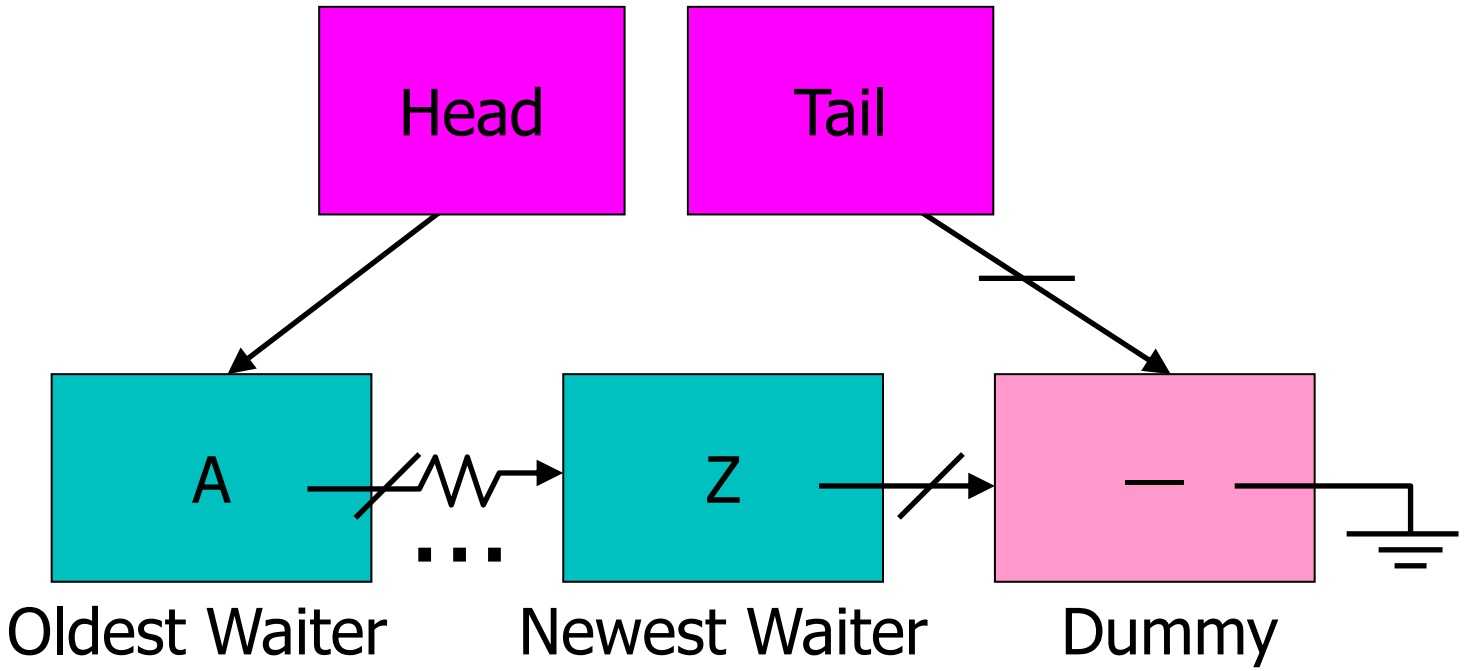
Dualqueue: dequeue III

4) If queue is empty, enqueue a tagged marker node, then swing tail pointer



Dualqueue: dequeue IV

4) Next, spin on the old tail node. Note: when queue holds reservations, dummy node is at *tail* end



Dualqueue: enqueue

1) Read head & tail pointers to see if queue looks empty or has data in it
2) If so, do an enqueue just as in the M&S queue
3) Else, try to satisfy a reservation

Dualqueue: satisfying requests

1) CAS pointer to data node into reservation node, breaking spin

   Alternatively, waiting thread can sleep on a semaphore, to which the awakening thread can post

2) CAS reservation node out of queue (dequeuing thread may help CAS)
3) Dequeuer reads data, frees reservation & data nodes

Synchronous stacks, queues, and exchangers

- Joint work with Doug Lea, chief architect of `java.util.concurrent` libraries
- In synchronous stacks & queues, producer waits for consumer; in exchanger they both wait for the other, then swap values
- Synchronous dualstack 3x faster than previous
- Synchronous dualqueue 14x faster
- Throughput of `Executor` library increased by 2x in “unfair” mode and 10x in “fair” mode
- Standard part of distribution since Java SE 6

Generic duals

- The dualqueue satisfies reservations in FIFO order; the dualstack in LIFO order
- We can write “quacks” and “steues” that use opposite orders for data and reservations
- More generally, we can pair *any* nonblocking container for data (e.g., a priority queue) with almost any nonblocking container for reservations

Almost any?

- Reservation container must provide two special methods
  - ⟨r, k⟩ = peek()
    returns the highest priority reservation and a *key*
  - s = removeConditional(k)
    removes r if it still has highest priority; returns status
- The ability to peek lets us satisfy reservations in a way that is amenable to *helping*
- We also employ a *handshaking* protocol to coordinate the two containers

How fast?

- Along with the generic construction [TOPC 2016] we also presented dual versions of Morrison & Afek's linked concurrent ring queue (LCRQ), which is based on fetch-and-increment (FAI)
- The resulting C code can sustain over 30M ops/s on an 18-core, 3.6GHz Intel Xeon processor
- That's almost 5x the throughput of the original M&S-based dualqueue
- Difficult to add to Java due to extensive pointer tagging

Conclusions/contributions

- Nonblocking operations really can wait
- Dualism improves the performance of
  - » stacks & queues, synchronous stacks & queues, exchangers, and more
  - » nonblocking and lock-based implementations
- Dualism also offers *fairness*: the data structure chooses which waiting thread to satisfy
- Generic construction allows any container for data to be combined with almost any container for reservations

Open questions

- Dual priority queues
- Predicates on remove
  - Give me an element larger than 100?
  - Give me a batch of 4 elements?
  - (What examples arise in practice?)
- Fast specific combinations
  - Priority queue with FIFO or LIFO request satisfaction
- Others?

For more information

- "Nonblocking Concurrent Data Structures with Condition Synchronization." W. N. Scherer III and M. L. Scott. *18th Annual Conf. on Distributed Computing (DISC)*, Oct. 2004.
- "Scalable Synchronous Queues." W. N. Scherer III, D. Lea, and M. L. Scott. *11th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, Mar. 2006; *Communications of the ACM*, May 2009.
- "Generality and Speed in Nonblocking Dual Containers." J. Izraelevitz and M. L. Scott. *ACM Transactions on Parallel Computing (TOPC)*, Mar. 2017.