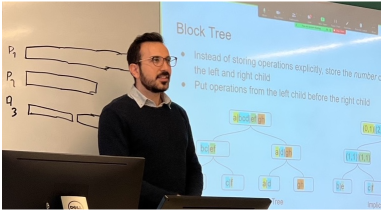
A Wait-Free Queue with Polylogarithmic Step Complexity

Hossein Naderibeni   Eric Ruppert

June 21, 2023



Queues: Breaking Linear-Time Bottleneck

Problem: implement linearizable, lock-free FIFO queue

- shared by p processes
- use single-word CAS (reasonable-sized words)
- support multiple enqueuers, dequeuers

Many previous solutions for this problem.

All require $\Omega(p)$ steps per operation
$\rightarrow$ Real obstacle to scalability

Our New Queue

- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE ($q =$ size of queue)
- wait-free

Queues: Breaking Linear-Time Bottleneck

Problem: implement linearizable, lock-free FIFO queue

- shared by p processes
- use single-word CAS (reasonable-sized words)
- support multiple enqueuers, dequeuers

Many previous solutions for this problem.

All require $\Omega(p)$ steps per operation
$\rightarrow$ Real obstacle to scalability

Our New Queue

- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE ($q =$ size of queue)
- wait-free

Queues: Breaking Linear-Time Bottleneck

Problem: implement linearizable, lock-free FIFO queue

- shared by p processes
- use single-word CAS (reasonable-sized words)
- support multiple enqueuers, dequeuers

Many previous solutions for this problem.

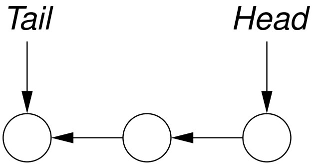
All require $\Omega(p)$ steps per operation
$\rightarrow$ Real obstacle to scalability

Our New Queue

- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE ($q =$ size of queue)
- wait-free

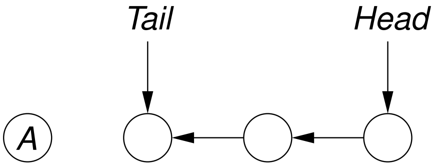
Lock-Free Queue using CAS

Michael and Scott Queue [PODC 1996]



Lock-Free Queue using CAS

Michael and Scott Queue [PODC 1996]

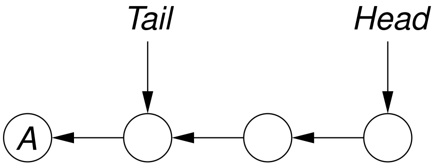


ENQUEUE(*A*):

1. create new node

Lock-Free Queue using CAS

Michael and Scott Queue [PODC 1996]

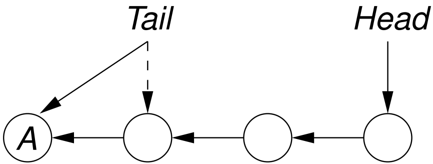


ENQUEUE(A):

1. create new node
2. CAS next pointer

Lock-Free Queue using CAS

Michael and Scott Queue [PODC 1996]

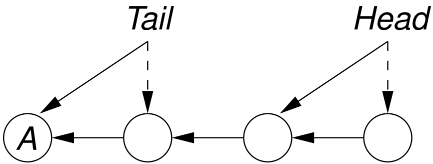


ENQUEUE(*A*):

1. create new node
2. CAS next pointer
3. advance *Tail*

Lock-Free Queue using CAS

Michael and Scott Queue [PODC 1996]



ENQUEUE(*A*):

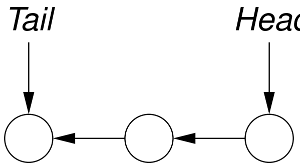
1. create new node
2. CAS next pointer
3. advance *Tail*

DEQUEUE:

1. CAS *Head*

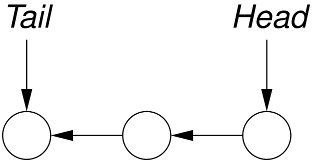
CAS Retry Problem

Suppose p processes want to enqueue simultaneously.



CAS Retry Problem

Suppose *p* processes want to enqueue simultaneously.



ENQUEUE(*A*) (*A*)

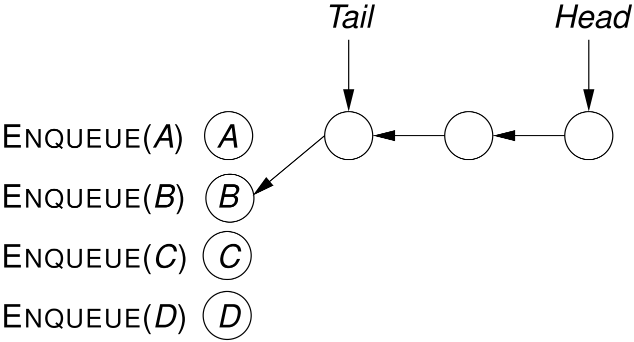
ENQUEUE(*B*) (*B*)

ENQUEUE(*C*) (*C*)

ENQUEUE(*D*) (*D*)

CAS Retry Problem

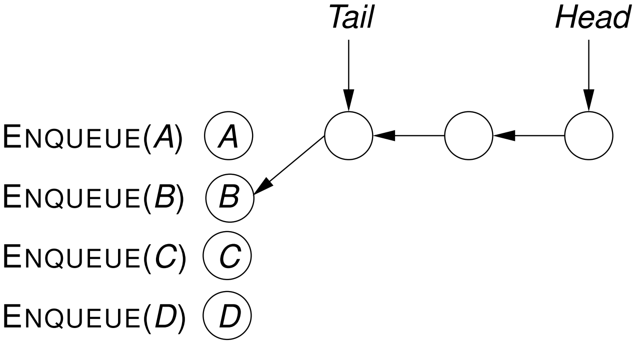
Suppose p processes want to enqueue simultaneously.



- one CAS of next pointer succeeds.
- other ENQUEUES must try again.
- $\Rightarrow$ starvation and $\Omega(p)$ steps per operation (amortized)

CAS Retry Problem

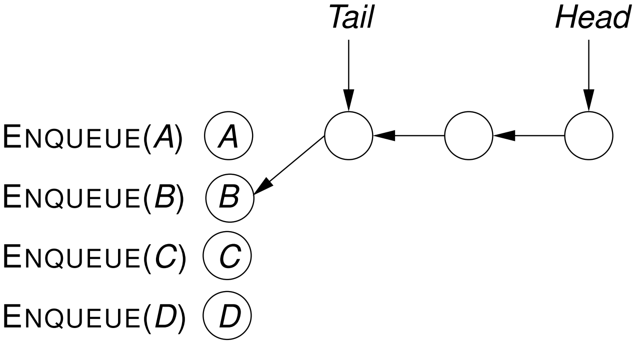
Suppose p processes want to enqueue simultaneously.



- one CAS of next pointer succeeds.
- other ENQUEUEs must try again.
- $\Rightarrow$ starvation and $\Omega(p)$ steps per operation (amortized)

CAS Retry Problem

Suppose p processes want to enqueue simultaneously.



- one CAS of next pointer succeeds.
- other ENQUEUES must try again.
- $\Rightarrow$ starvation and $\Omega(p)$ steps per operation (amortized)

Other Lock-Free Queues

Other list-based queues

- add elimination array [Moir et al. 2005]
- baskets queue [Hoffman, Shalev, Shavit 2007]
- doubly-linked list + optimism [Ladan-Mozes, Shavit 2008]
- fast path, slow path [Kogan, Petrank 2012]
- futures [Kogan, Herlihy 2014]

Other Lock-Free Queues

Other list-based queues

- add elimination array [Moir et al. 2005]
- baskets queue [Hoffman, Shalev, Shavit 2007]
- doubly-linked list + optimism [Ladan-Mozes, Shavit 2008]
- fast path, slow path [Kogan, Petrank 2012]
- futures [Kogan, Herlihy 2014]

All have CAS retry problem [Morrison, Afek 2013]

Other Lock-Free Queues

Other list-based queues

- add elimination array [Moir et al. 2005]
- baskets queue [Hoffman, Shalev, Shavit 2007]
- doubly-linked list + optimism [Ladan-Mozes, Shavit 2008]
- fast path, slow path [Kogan, Petrank 2012]
- futures [Kogan, Herlihy 2014]

All have CAS retry problem [Morrison, Afek 2013]

So do array-based queues

All* previous queues take amortized $\Omega(p)$ steps per operation

YORK UNIVERSITÉ UNIVERSITY

*Exceptions: Sublinear Time Queues

Restricted queues

- 1 enqueuer, multiple dequeuers [David 2004]
- 1 dequeuer, multiple enqueuers [Jayanti, Petrovic 2005]

Other primitives

- $O(\sqrt{p})$ using unusual double-word RMW instructions [Khanchandani, Wattenhofer 2018]

Universal constructions

- $O(\log p)$ using huge words [Afek, Dauber, Touitou 1995; Jayanti 1998]
- $\Omega(p)$ with reasonably-sized words

*Exceptions: Sublinear Time Queues

Restricted queues

- 1 enqueuer, multiple dequeuers [David 2004]
- 1 dequeuer, multiple enqueuers [Jayanti, Petrovic 2005]

Other primitives

- $O(\sqrt{p})$ using unusual double-word RMW instructions [Khanchandani, Wattenhofer 2018]

Universal constructions

- $O(\log p)$ using huge words [Afek, Dauber, Touitou 1995; Jayanti 1998]
- $\Omega(p)$ with reasonably-sized words

*Exceptions: Sublinear Time Queues

Restricted queues

- 1 enqueuer, multiple dequeuers [David 2004]
- 1 dequeuer, multiple enqueuers [Jayanti, Petrovic 2005]

Other primitives

- $O(\sqrt{p})$ using unusual double-word RMW instructions [Khanchandani, Wattenhofer 2018]

Universal constructions

- $O(\log p)$ using huge words [Afek, Dauber, Touitou 1995; Jayanti 1998]
- $\Omega(p)$ with reasonably-sized words

Lower Bound

All previous multi-enqueuer, multi-dequeuer queues take $\Omega(p)$ steps per operation.

For many data structures, fastest lock-free operations take $O(\textit{sequential complexity} + \textit{contention})$ steps

Lower Bound [Attiya, Fouren 2017]

- Amortized step complexity for any bag is $\Omega(\textit{contention})$.
- But lower bound holds only if *contention* is $O(\log \log p)$

Lower Bound

All previous multi-enqueuer, multi-dequeuer queues take $\Omega(p)$ steps per operation.

For many data structures, fastest lock-free operations take $O(\textit{sequential complexity} + \textit{contention})$ steps

Lower Bound **[Attiya, Fouren 2017]**

- Amortized step complexity for any bag is $\Omega(\textit{contention})$.
- But lower bound holds only if *contention* is $O(\log \log p)$

Lower Bound

All previous multi-enqueuer, multi-dequeuer queues take $\Omega(p)$ steps per operation.

For many data structures, fastest lock-free operations take $O(\textit{sequential complexity} + \textit{contention})$ steps

Lower Bound **[Attiya, Fouren 2017]**

- Amortized step complexity for any bag is $\Omega(\textit{contention})$.
- But lower bound holds only if *contention* is $O(\log \log p)$

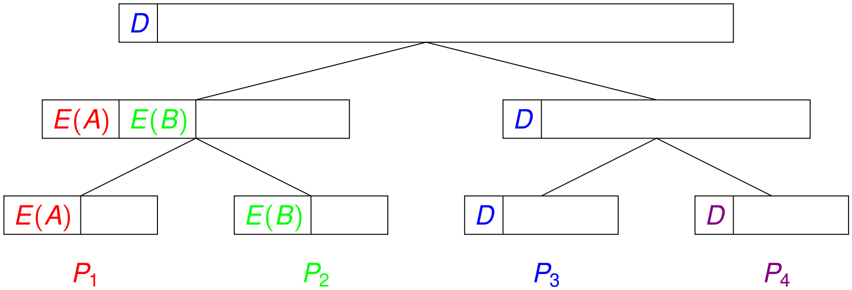
Breaking Linear-Time Bottleneck

Our New Queue

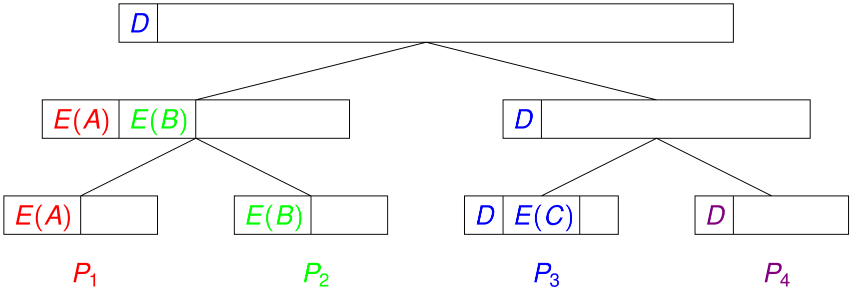
- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE
- wait-free
- uses CAS on reasonable-size words
- bounded space version: $O(\log p \log (p + q))$ amortized steps per operation (relies on safe GC)

$p = \#$ processes
$q = \#$ elements in queue

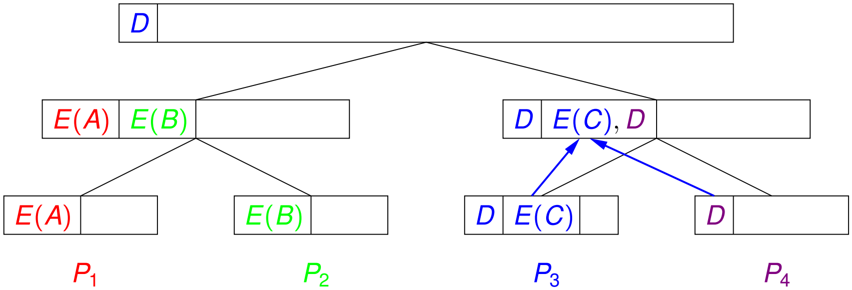
Ordering Tree



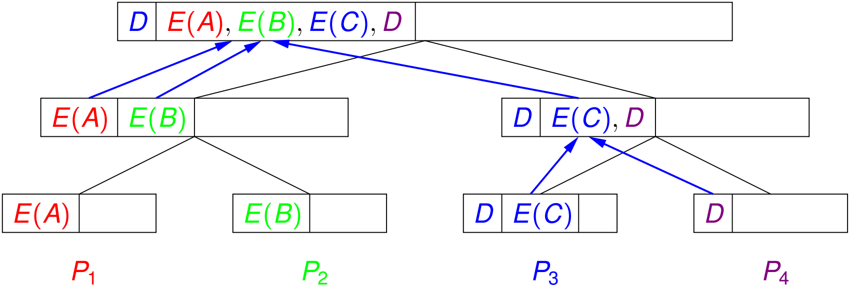
Ordering Tree



Ordering Tree

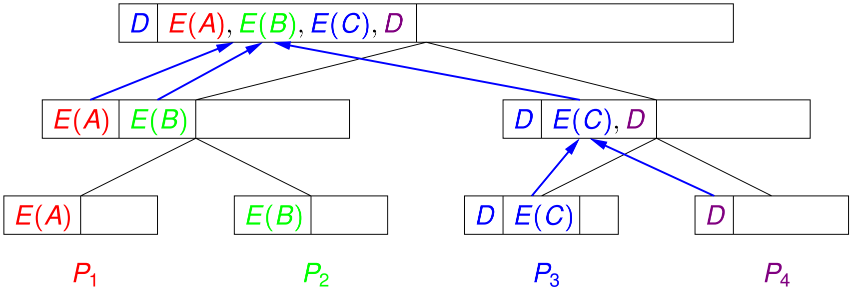


Ordering Tree



Ordering Tree

Use ordering in root as linearization



Propagating Operations to the Root

1. Append operation to your leaf
2. At each node v on path to root refresh *twice*:
   (a) Read unpropagated operations in both of v's children
   (b) CAS them into v

Double Refresh

If your CAS on v fails twice, then
another process has propagated your operation to v.

Avoids CAS retry problem.

Propagating Operations to the Root

1. Append operation to your leaf
2. At each node *v* on path to root refresh *twice*:
   - (a) Read unpropagated operations in both of *v*'s children
   - (b) CAS them into *v*

Double Refresh

If your CAS on *v* fails twice, then
another process has propagated your operation to *v*.

Avoids CAS retry problem.

Propagating Operations to the Root

1. Append operation to your leaf
2. At each node *v* on path to root refresh *twice*:
   (a) Read unpropagated operations in both of *v*'s children
   (b) CAS them into *v*

Double Refresh

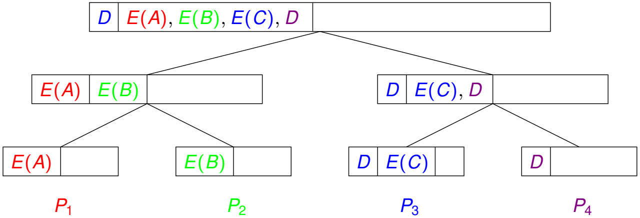
If your CAS on *v* fails twice, then
another process has propagated your operation to *v*.

Avoids CAS retry problem.

Main Challenge

Refresh may have to propagate up to p operations
$\Rightarrow$ need an implicit representation

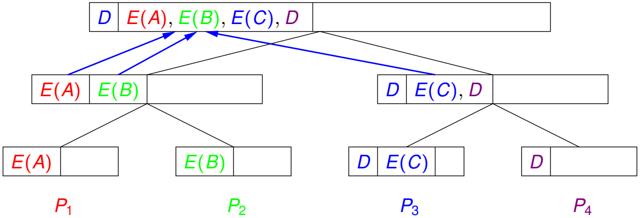
Requirements for Implicit Representation



Must support following in polylog time

- Refresh: promote batch of ops from children to parent
- Find my DEQUEUE in root
- Check if DEQUEUE returns null, or otherwise determine rank of DEQUEUE among non-null DEQUEUES
- Find ENQUEUE of given rank (and its argument)

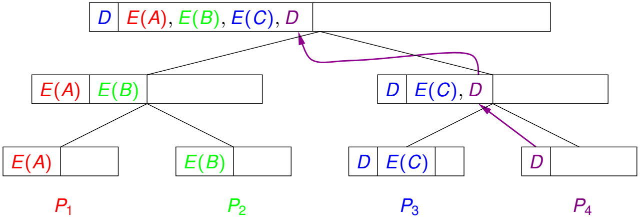
Requirements for Implicit Representation



Must support following in polylog time

- Refresh: promote batch of ops from children to parent
- Find my DEQUEUE in root
- Check if DEQUEUE returns null, or otherwise determine rank of DEQUEUE among non-null DEQUEUES
- Find ENQUEUE of given rank (and its argument)

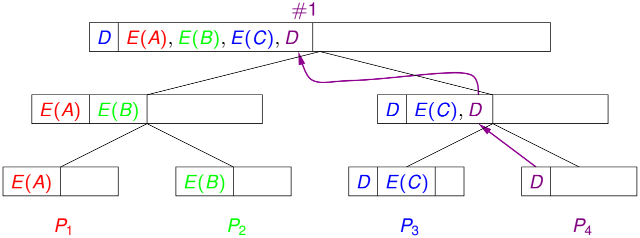
Requirements for Implicit Representation



Must support following in polylog time

- Refresh: promote batch of ops from children to parent
- Find my DEQUEUE in root
- Check if DEQUEUE returns null, or otherwise determine rank of DEQUEUE among non-null DEQUEUES
- Find ENQUEUE of given rank (and its argument)

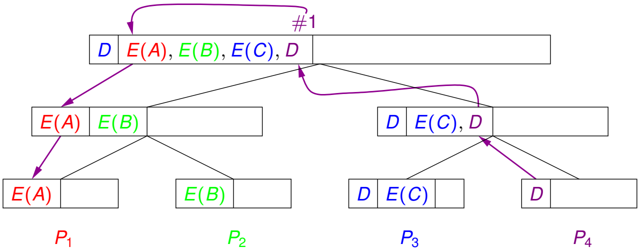
Requirements for Implicit Representation



Must support following in polylog time

- Refresh: promote batch of ops from children to parent
- Find my DEQUEUE in root
- Check if DEQUEUE returns null, or otherwise determine rank of DEQUEUE among non-null DEQUEUES
- Find ENQUEUE of given rank (and its argument)

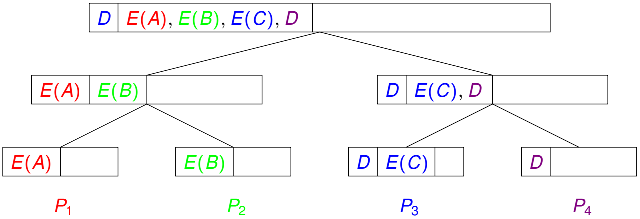
Requirements for Implicit Representation



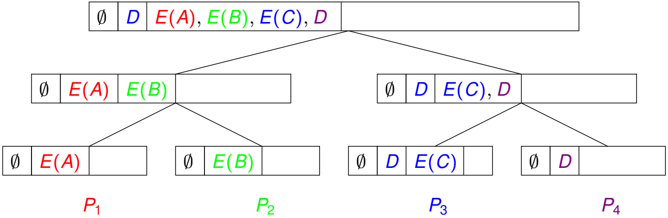
Must support following in polylog time

- Refresh: promote batch of ops from children to parent
- Find my DEQUEUE in root
- Check if DEQUEUE returns null, or otherwise determine rank of DEQUEUE among non-null DEQUEUES
- Find ENQUEUE of given rank (and its argument)

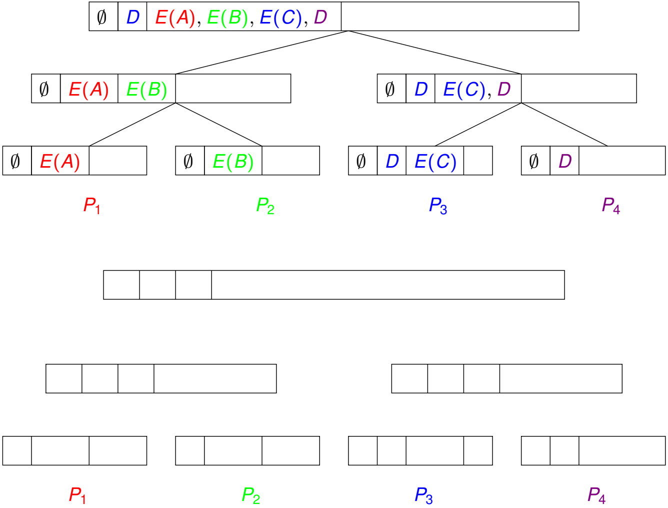
Implicit Representation: Blocks



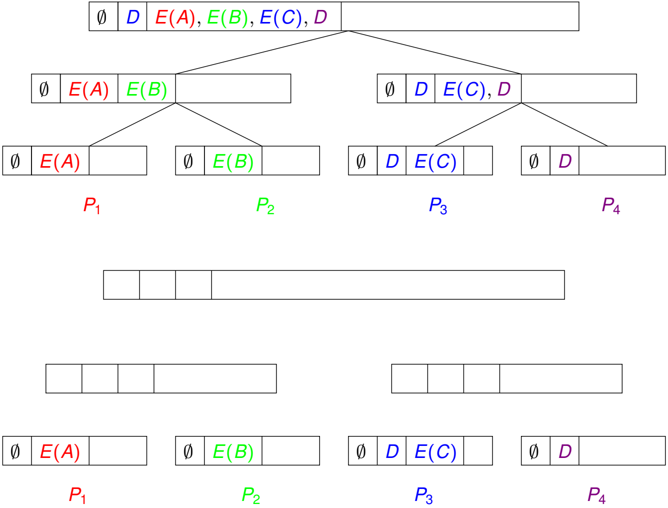
Implicit Representation: Blocks



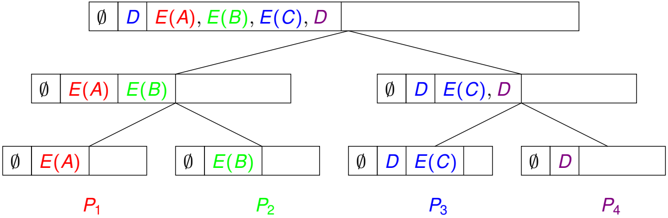
Implicit Representation: Blocks

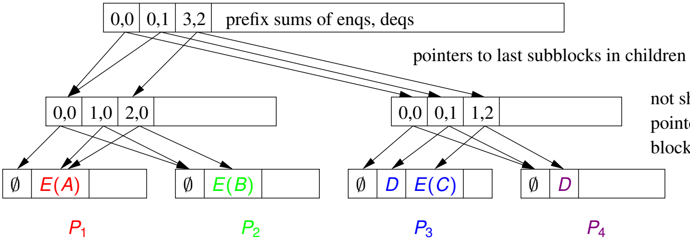


Implicit Representation: Blocks



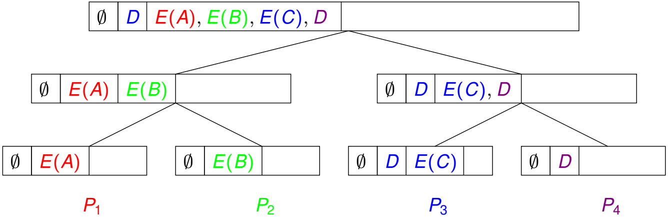
Implicit Representation: Blocks

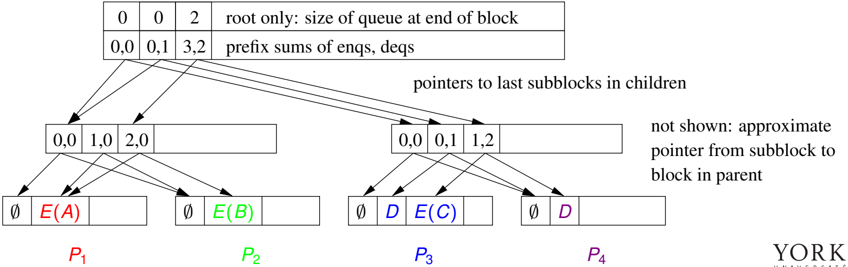




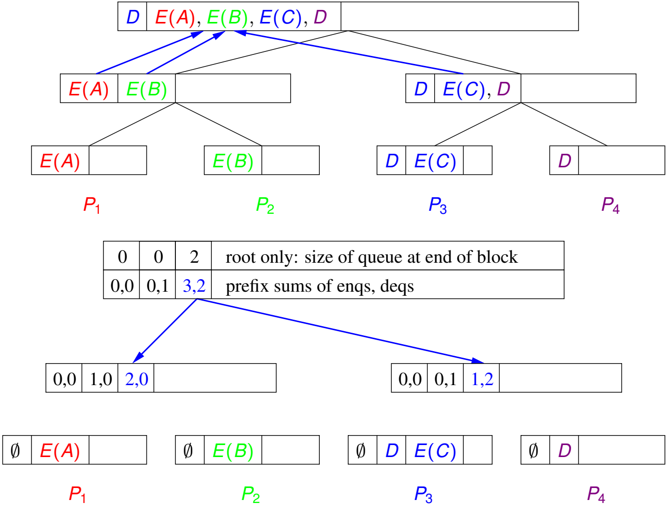
not shown: approximate pointer from subblock to block in parent

Implicit Representation: Blocks

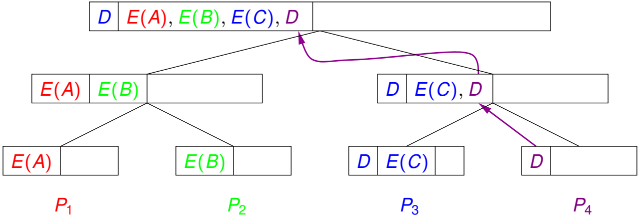


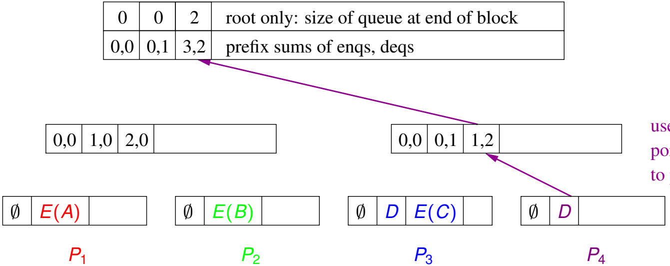


Adding a Block for a Refresh



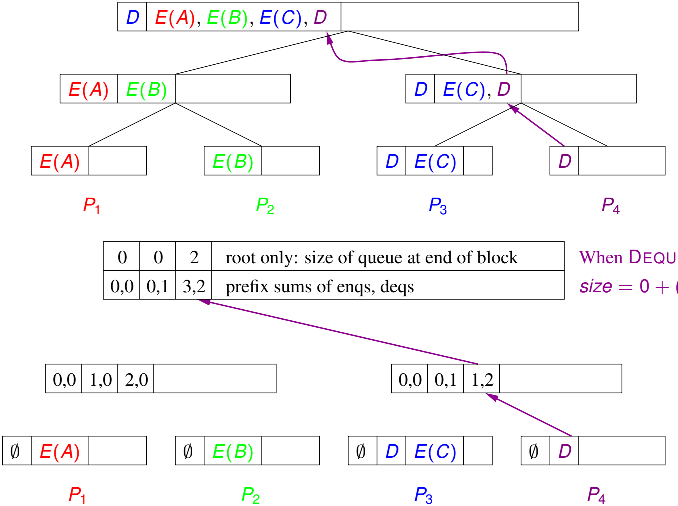
Tracing a DEQUEUE to the Root





use approximate
pointers from sublocks
to parent's block

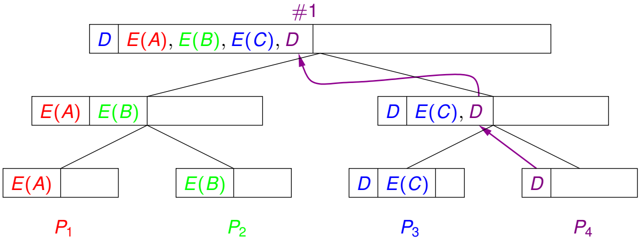
Check if DEQUEUE Returns Null



When DEQUEUE occurs,

$size = 0 + (3 - 0) > 0$

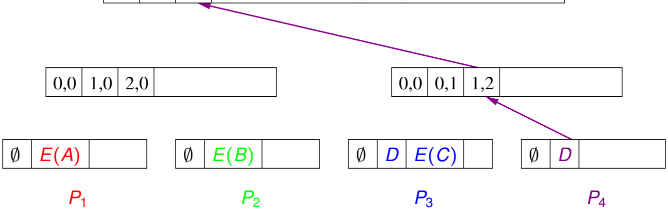
Rank of DEQUEUE Among Non-Null DEQUEUES



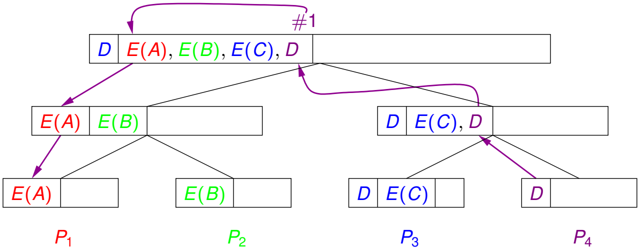
0	0	2	root only: size of queue at end of block
0,0	0,1	3,2	prefix sums of enqs, deqs

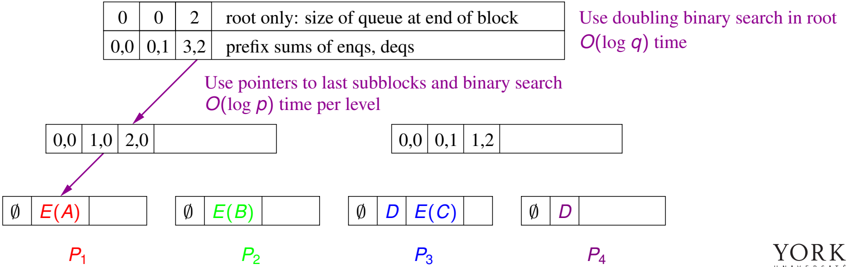
use *size* = #enqs − # non-null deqs

⇒ #non-null deqs = #enqs − *size*



Find ENQUEUE of Given Rank





Our Result

New Wait-Free Queue

- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE
- $O(\log p)$ CAS steps per DEQUEUE
- Unbounded space

$p = \#$ processes
$q = \#$ elements in queue

Our Result

New Wait-Free Queue

- $O(\log p)$ steps per ENQUEUE
- $O(\log^2 p + \log q)$ steps per DEQUEUE
- $O(\log p)$ CAS steps per DEQUEUE
- Unbounded space

$p = \#$ processes
$q = \#$ elements in queue

Bounding Space

- Replace each array of blocks with a red-black tree of blocks
- Periodically split RBT and discard obsolete blocks
- Processes help one another to ensure blocks are obsolete

Bounded-Space Queue

- Amortized $O(\log p \log (p+q))$ steps per operation
- $O(pq + p^3 \log p)$ space
- Still wait-free

$p = \#$ processes
$q = \#$ elements in queue

Future Directions

- Practical implementation
  (perhaps slow path of fast path slow path method)
- Extend technique to other data structures
  (stacks and deques are recently done)
- Close gap between
  $\Omega(\log \log p)$ lower bound [Attiya Fouren 2017]
  $O(\log^2 p + \log q)$ upper bound [this work]