CSE 613: Parallel Programming

Lecture 15
(Concurrent Data Structures: Queues and Stacks)

Rezaul A. Chowdhury
Department of Computer Science
SUNY Stony Brook
Spring 2019

Desirable Properties of Concurrent Objects

Safety Property

— Nothing bad ever happens

— Needed for correctness

Liveness Property

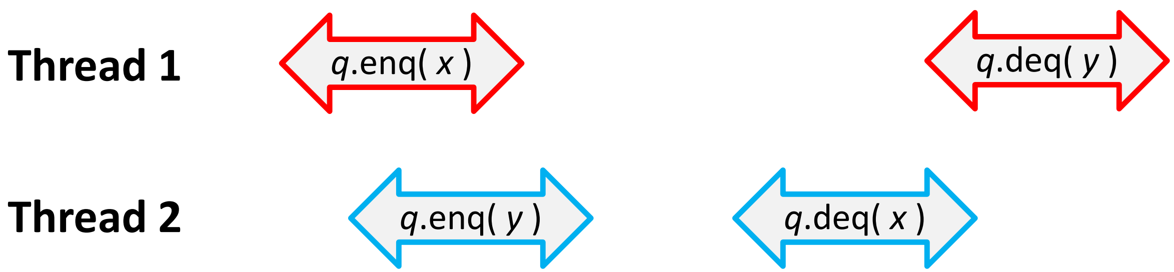
— Something good eventually happens

— Needed for progress (e.g., no deadlock)

Correctness Properties

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order

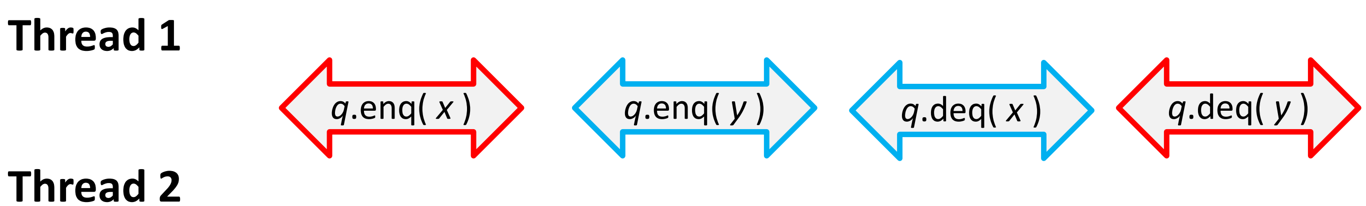


Correctness Properties

Sequential Consistency

— Method calls should appear to happen in a one-at-a-time sequential order

— For each thread method calls should appear to take effect in program order



Sequentially Consistent
(one way)

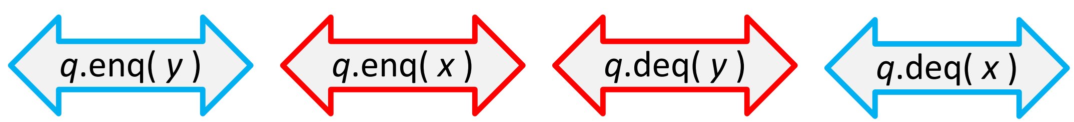
Correctness Properties

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order

Thread 1

Thread 2



Sequentially Consistent
(another way)

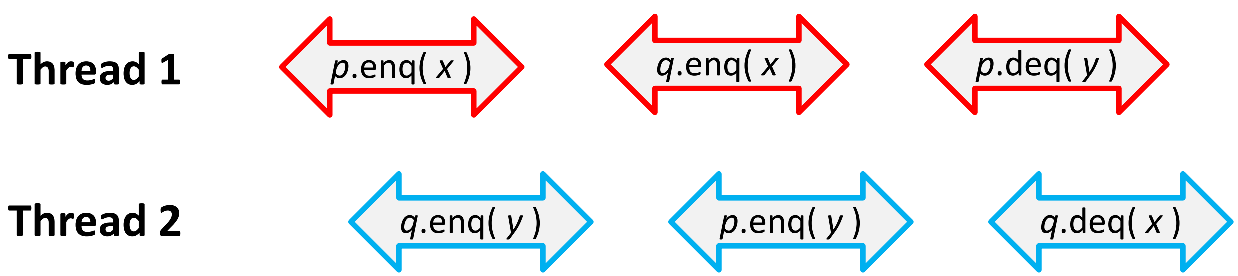
Correctness Properties

Sequential Consistency

— Method calls should appear to happen in a one-at-a-time sequential order

— For each thread method calls should appear to take effect in program order

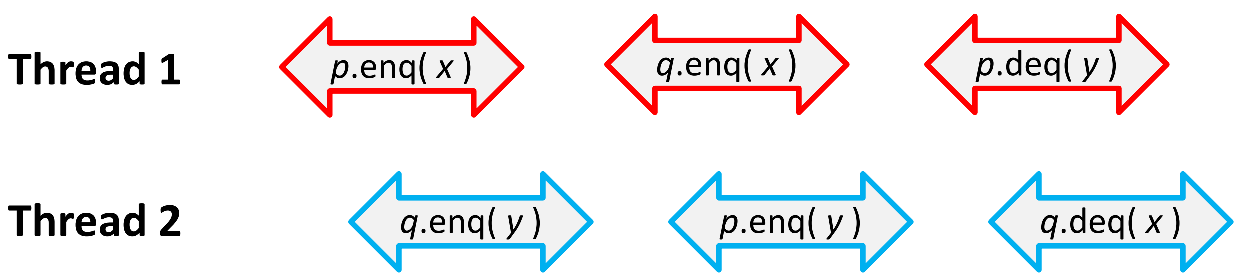
— Sequential Consistency is not compositional



Correctness Properties

Sequential Consistency

- Method calls should appear to happen in a one-at-a-time sequential order
- For each thread method calls should appear to take effect in program order
- Sequential Consistency is not compositional

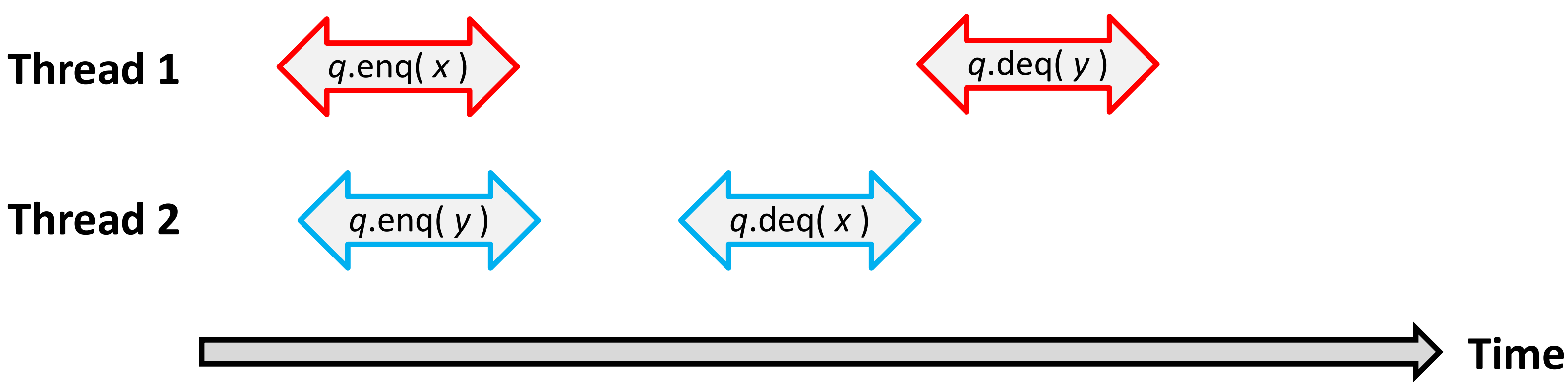


p and *q* are independently sequentially consistent, but their composition is not

Correctness Properties

Linearizability

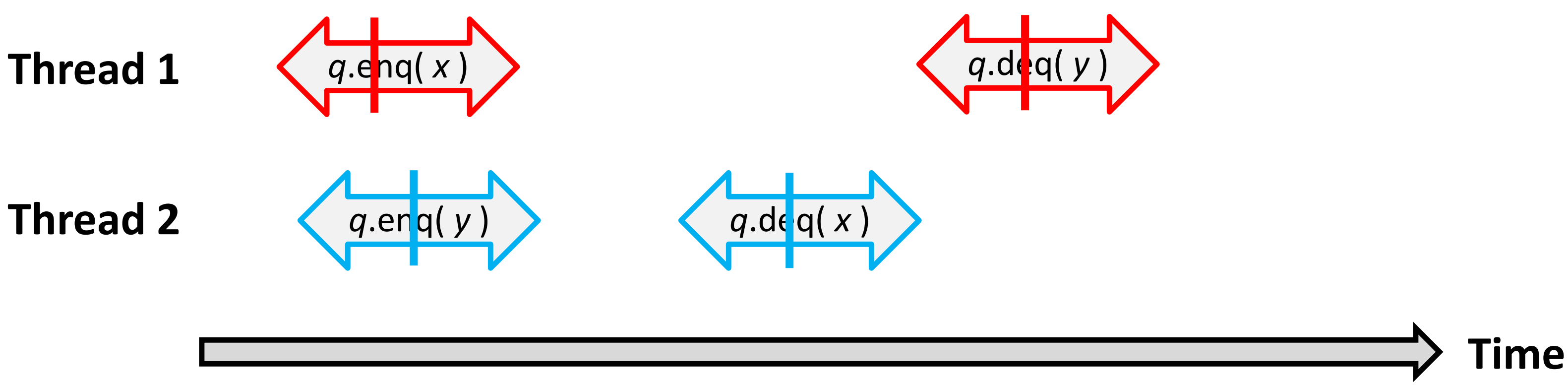
- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional



Correctness Properties

Linearizability

- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional

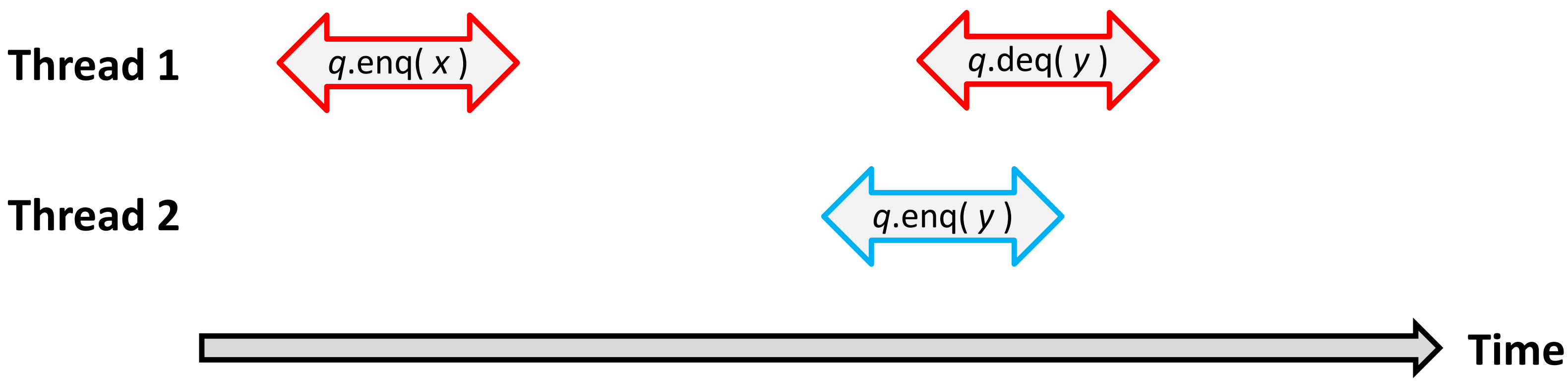


Linearizable

Correctness Properties

Linearizability

- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional

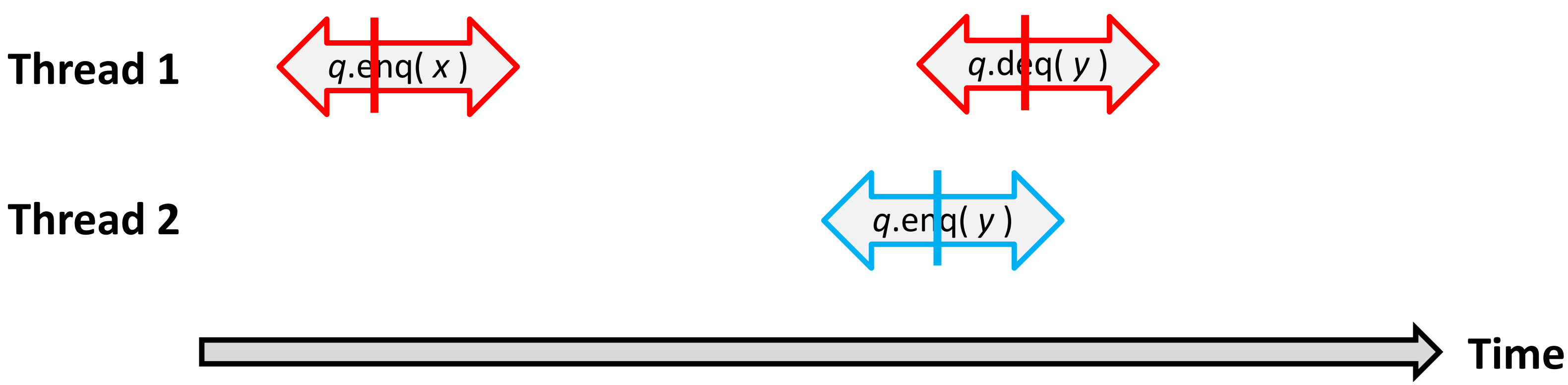


Correctness Properties

Linearizability

— Each method call should appear to take effect instantaneously at some moment between its invocation and response

— Compositional

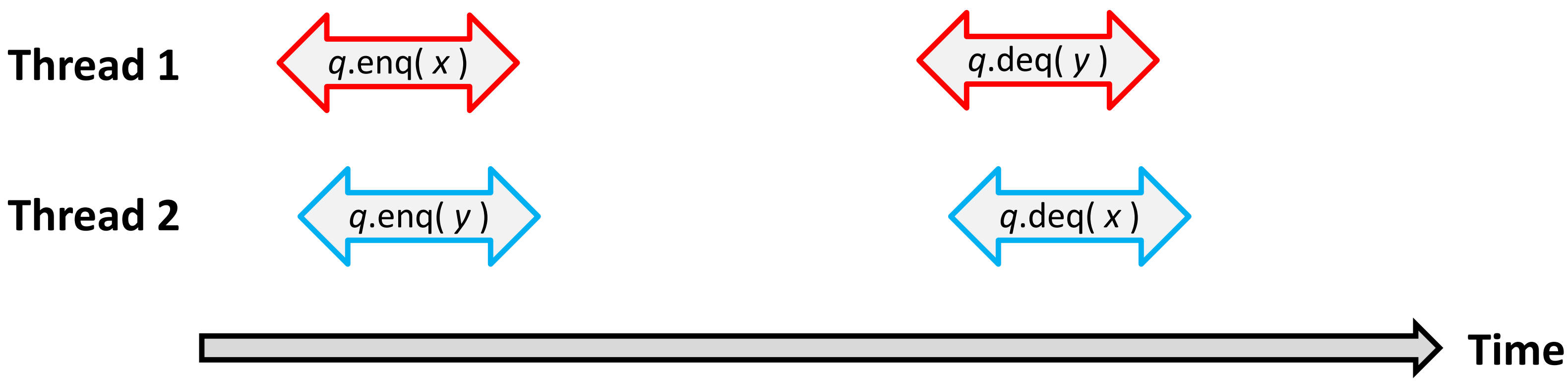


Not Linearizable

Correctness Properties

Linearizability

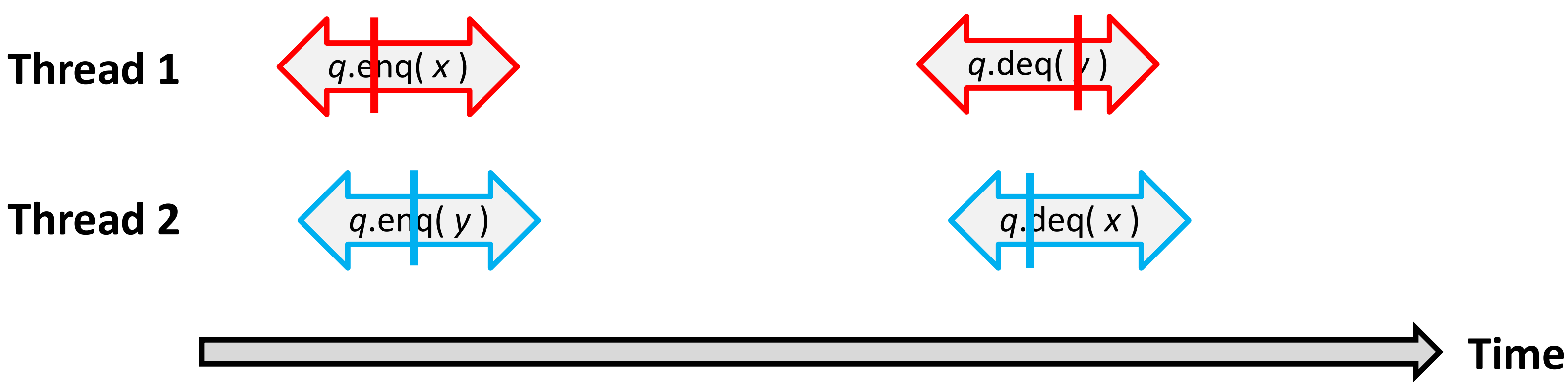
- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional



Correctness Properties

Linearizability

- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional

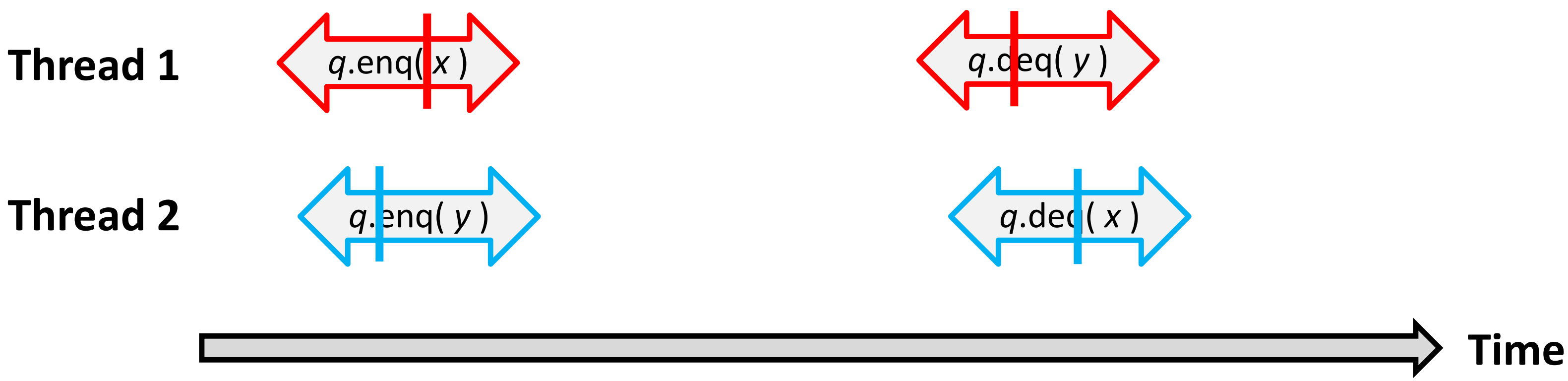


Linearizable
(one way)

Correctness Properties

Linearizability

- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional

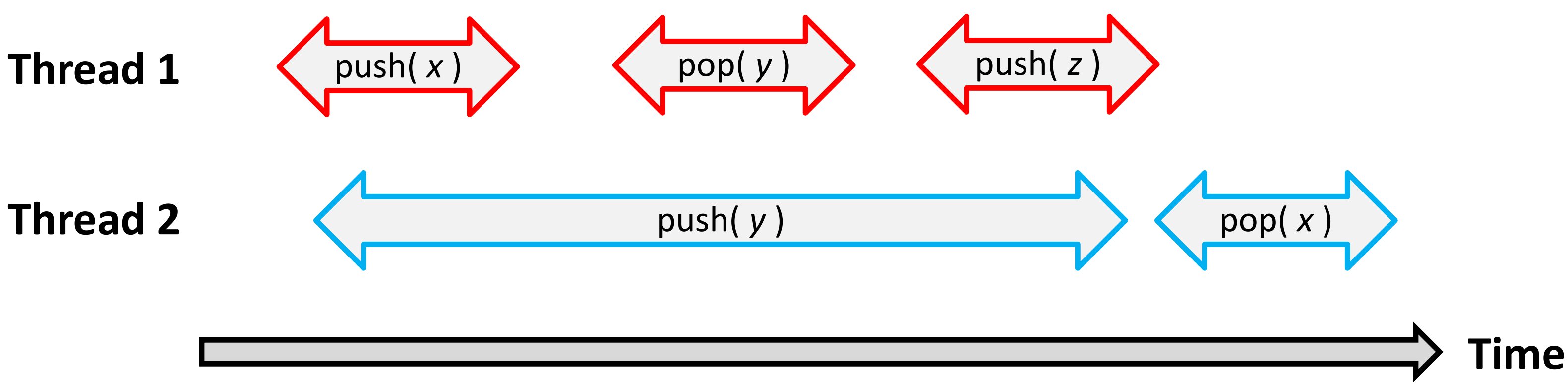


Linearizable
(another way)

Correctness Properties

Linearizability

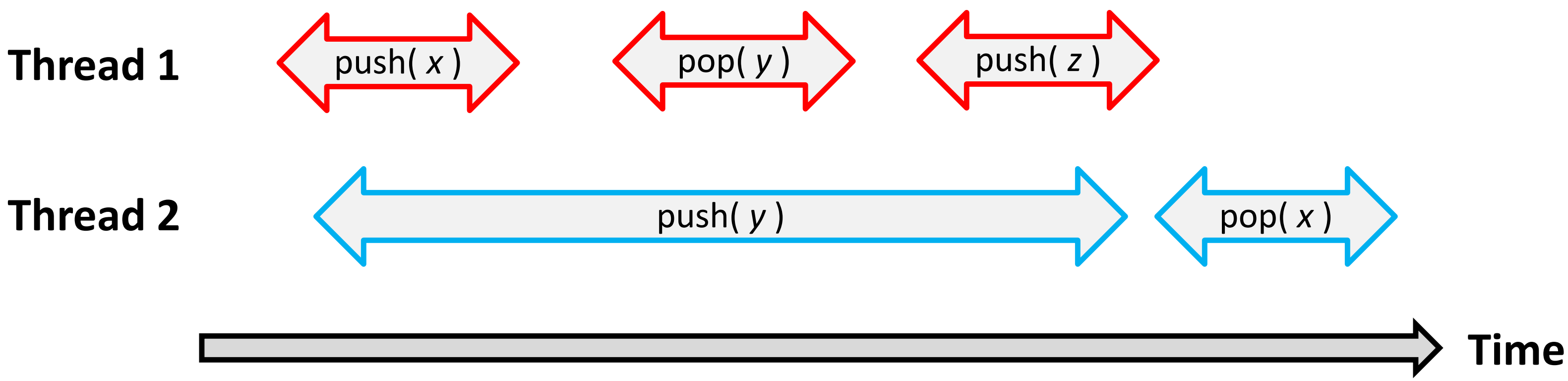
- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional



Correctness Properties

Linearizability

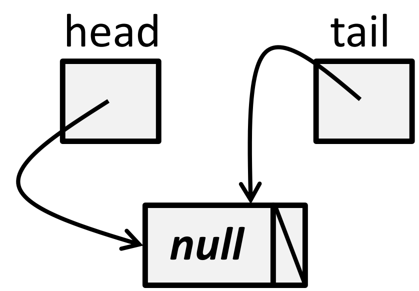
- Each method call should appear to take effect instantaneously at some moment between its invocation and response
- Compositional



Not Linearizable

A Bounded Lock-Based Queue

```java
public class BoundedQueue< T > {
    ReentrantLock enqLock, deqLock;
    Condition notEmptyCondition, notFullCondition;
    AtomicInteger size;
    Node head, tail;
    int capacity;
    public BoundedQueue( int _capacity ) {
        capacity = _capacity;
        head = new Node( null );
        tail = head;
        size = new AtomicInteger( 0 );
        enqLock = new ReentrantLock( );
        notFullCondition = enqLock.newCondition( );
        deqLock = new ReentrantLock( );
        notEmptyCondition = deqLock.newCondition( );
    }
```



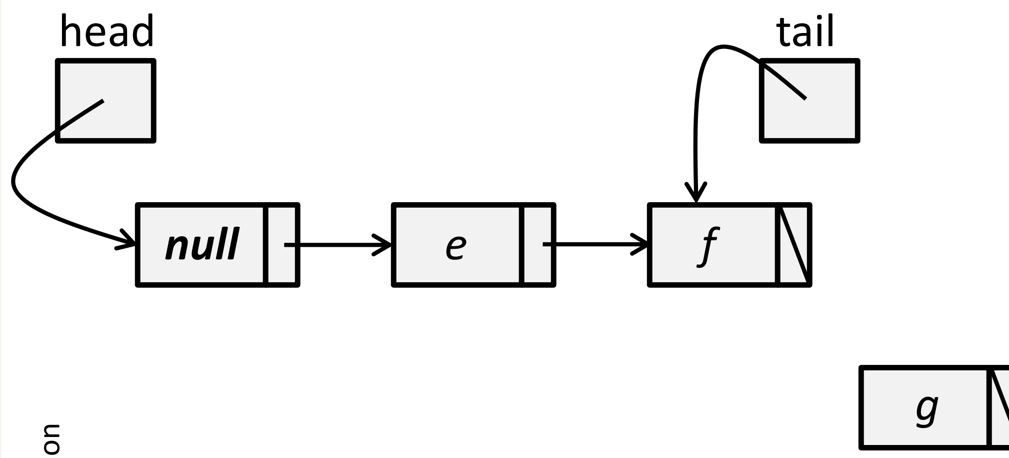

```java
protected class Node {
    public T value;
    public Node next;
    public Node( T x ) {
        value = x;
        next = null;
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

A Bounded Lock-Based Queue: Enqueue

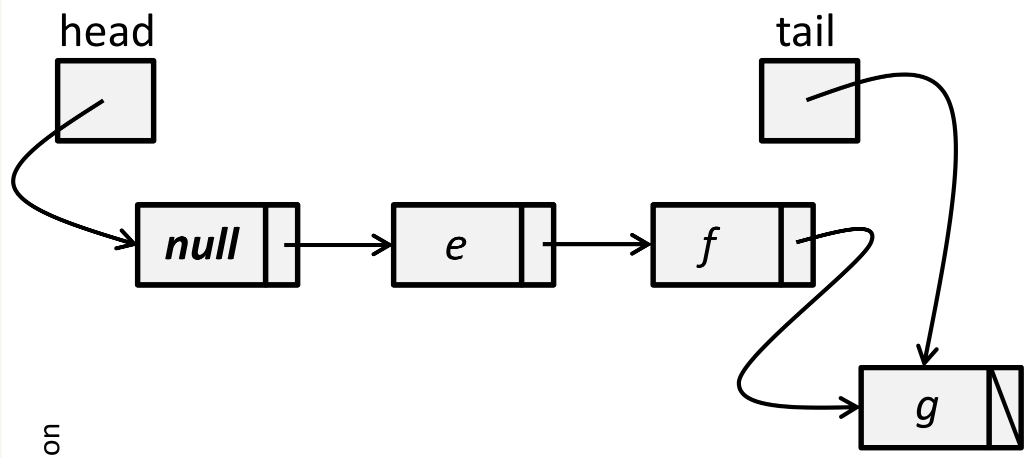
```
public void enq( T x ) {
    boolean mustWakeDequeuers = false;
    enqLock.lock( );
    try {
      while ( size.get( ) == capacity )
          notFullCondition.await( );
      Node e = new Node( x );
      tail.next = tail = e;
      if ( size.getAndIncrement( ) == 0 )
          mustWakeDequeuers = true;
    } finally {
      enqLock.unlock( );
    }
    if ( mustWakeDequeuers ) {
      deqLock.lock( );
      try {
          notEmptyCondition.signalAll( );
      } finally {
          deqLock.unlock( );
      }
    }
}
```

Source: Herlihy & Shavit.,
"The Art of Multiprocessor Programming", 1st Edition



A Bounded Lock-Based Queue: Enqueue

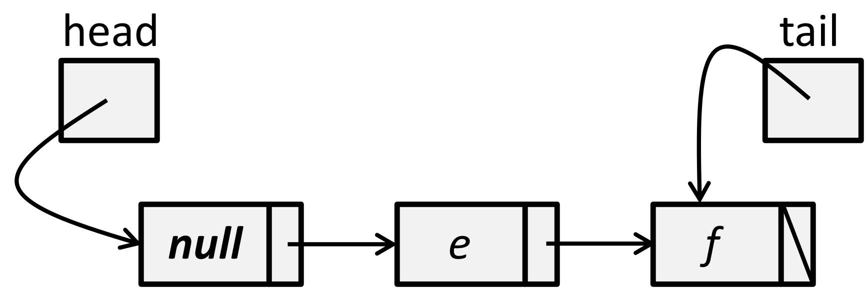
```
public void enq( T x ) {
    boolean mustWakeDequeuers = false;
    enqLock.lock( );
    try {
      while ( size.get( ) == capacity )
          notFullCondition.await( );
      Node e = new Node( x );
      tail.next = tail = e;
      if ( size.getAndIncrement( ) == 0 )
          mustWakeDequeuers = true;
    } finally {
      enqLock.unlock( );
    }
    if ( mustWakeDequeuers ) {
      deqLock.lock( );
      try {
        notEmptyCondition.signalAll( );
      } finally {
        deqLock.unlock( );
      }
    }
}
```



Source: Herlihy & Shavit.,
"The Art of Multiprocessor Programming", 1st Edition

A Bounded Lock-Based Queue: Dequeue

```
public T deq( ) {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock( );
    try {
       while ( size.get( ) == 0 )
           notEmptyCondition.await( );
       result = head.next.value;
       head = head.next;
       if ( size.getAndIncrement( ) == capacity ) {
          mustWakeEnqueuers = true;
       }
    } finally {  deqLock.unlock( ); }
    if ( mustWakeEnqueuers ) {
       enqLock.lock( );
       try {
          notFullCondition.signalAll( );
       } finally { enqLock.unlock( ); }
    }
    return result;
}
```

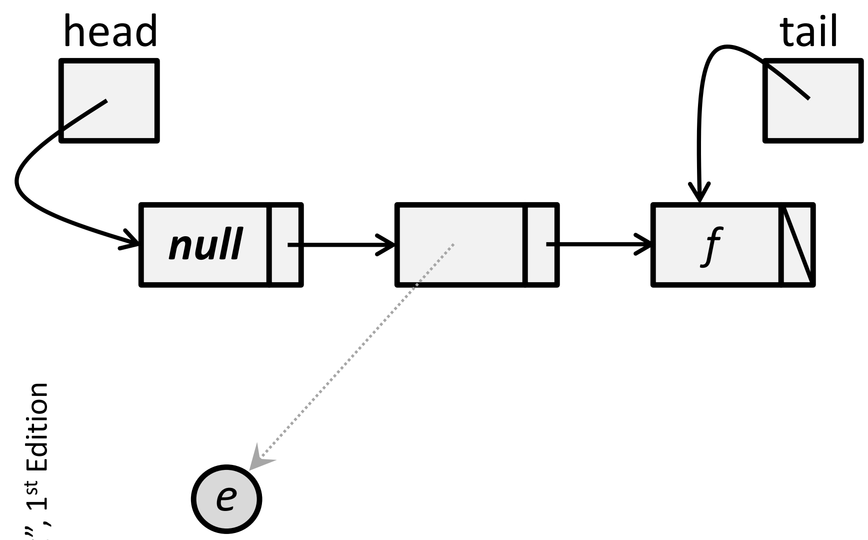


Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition

A Bounded Lock-Based Queue: Dequeue

```
public T deq( ) {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock( );
    try {
        while ( size.get( ) == 0 )
            notEmptyCondition.await( );
        result = head.next.value;
        head = head.next;
        if ( size.getAndIncrement( ) == capacity ) {
            mustWakeEnqueuers = true;
        }
    } finally {  deqLock.unlock( ); }
    if ( mustWakeEnqueuers ) {
        enqLock.lock( );
        try {
            notFullCondition.signalAll( );
        } finally { enqLock.unlock( ); }
    }
    return result;
}
```

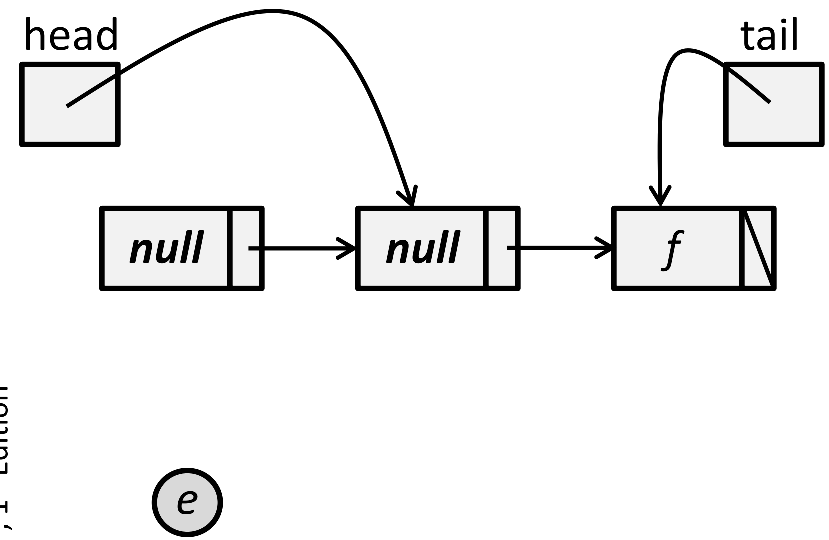
Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition



A Bounded Lock-Based Queue: Dequeue

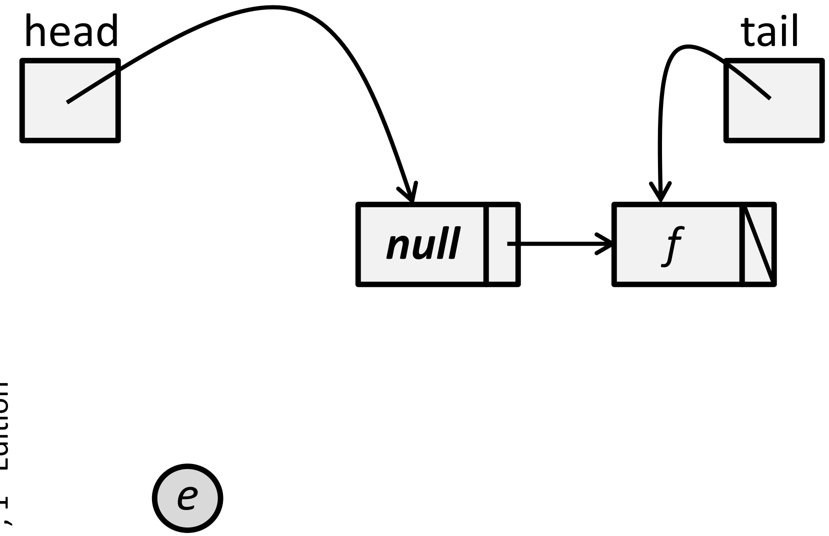
```
public T deq( ) {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock( );
    try {
        while ( size.get( ) == 0 )
            notEmptyCondition.await( );
        result = head.next.value;
        head = head.next;
        if ( size.getAndIncrement( ) == capacity ) {
            mustWakeEnqueuers = true;
        }
    } finally {  deqLock.unlock( ); }
    if ( mustWakeEnqueuers ) {
        enqLock.lock( );
        try {
            notFullCondition.signalAll( );
        } finally { enqLock.unlock( ); }
    }
    return result;
}
```

Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition



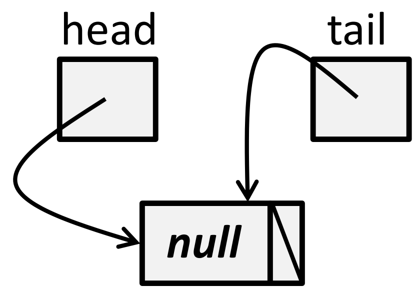
A Bounded Lock-Based Queue: Dequeue

```
public T deq( ) {
    T result;
    boolean mustWakeEnqueuers = true;
    deqLock.lock( );
    try {
        while ( size.get( ) == 0 )
            notEmptyCondition.await( );
        result = head.next.value;
        head = head.next;
        if ( size.getAndIncrement( ) == capacity ) {
            mustWakeEnqueuers = true;
        }
    } finally {  deqLock.unlock( ); }
    if ( mustWakeEnqueuers ) {
        enqLock.lock( );
        try {
            notFullCondition.signalAll( );
        } finally { enqLock.unlock( ); }
    }
    return result;
}
```



Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition

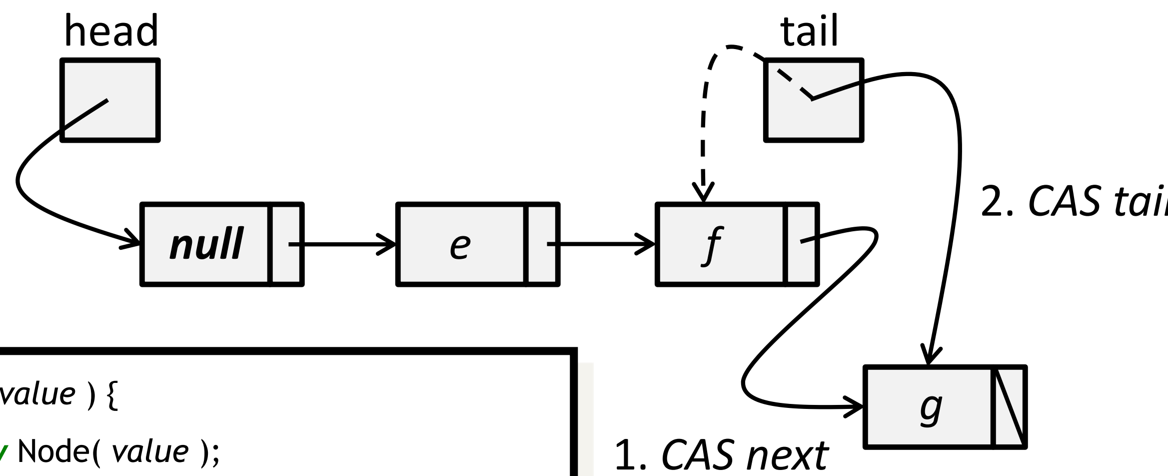
An Unbounded Lock-Free Queue




```
public class Node {
    public T value;
    public AtomicReference< Node > next;
    public Node( T _value ) {
        value = _value;
        next = new AtomicReference< Node >( null );
    }
}
```

Source: Herlihy & Shavit.,
"The Art of Multiprocessor Programming", 1st Edition

An Unbounded Lock-Free Queue: Enqueue




```
public void enq( T value ) {
    Node node = new Node( value );
    while ( true ) {
        Node last = tail.get( );
        Node next = last.next.get( );
        if ( last == tail.get( ) ) {
            if ( next == null ) {
                if ( last.next.compareAndSet( next, node ) ) {
                    tail.compareAndSet( last, node );
                    return;
                }
            } else { tail.compareAndSet( last, next ); }
        }
    }
}
```

Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition

An Unbounded Lock-Free Queue: Dequeue

```
public T deq( ) throws EmptyException {
    while ( true ) {
        Node first = head.get( );
        Node last = tail.get( );
        Node next = first.next.get( );
        if ( first == head.get( ) ) {
          if ( first == last ) {
            if ( next == null ) {
              throw new EmptyException( );
            }
            tail.compareAndSet( last, next );
          } else {
            T value = next.value;
            if ( head.compareAndSet( first, next ) )
               return value;
          }
        }
    }
}
```

Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition

Exponential Backoff

```
public class Backoff {
    final int minDelay, maxDelay;
    int limit;
    final Random rand;
    public Backoff( int min, int max ) {
        minDelay = min;
        maxDelay = min;
        limit = minDelay;
        rand = new Random( );
    }
    public void backoff( ) throws InterruptedException {
        int delay = rand.nextInt( limit );
        limit = Math.min( maxDelay, 2 * limit );
        Thread.sleep( delay );
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

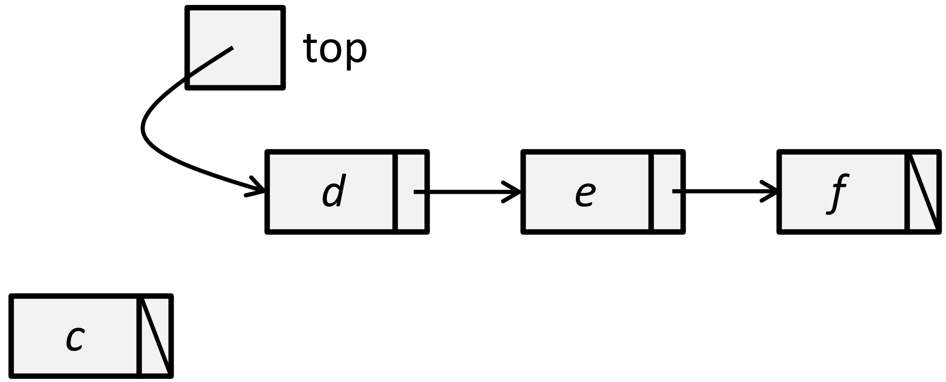
An Unbounded Lock-Free Stack

```
public class LockFreeStack< T > {
    AtomicReference< Node > top = new AtomicReference< Node >( null );
    static final int MIN_DELAY = ...;
    static final int MAX_DELAY = ...;
    Backoff backoff = new Backoff( MIN_DELAY, MAX_DELAY );

    protected class Node {
        public T value;
        public Node next;
        public Node( T _value ) {
            value = _value;
            next = null;
        }
    }
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

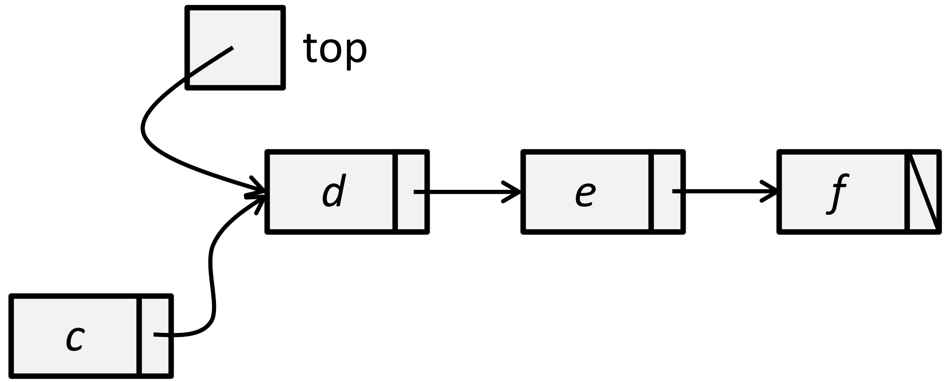
An Unbounded Lock-Free Stack: Push




```
protected boolean tryPush( Node node ) {
    Node oldTop = top.get( );
    node.next = oldTop;
    return ( top.compareAndSet( oldTop, node ) );
}
public void push( T value ) {
    Node node = new Node( value );
    while ( true ) {
        if ( tryPush( node ) ) { return; }
        else { backoff.backoff( ); }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

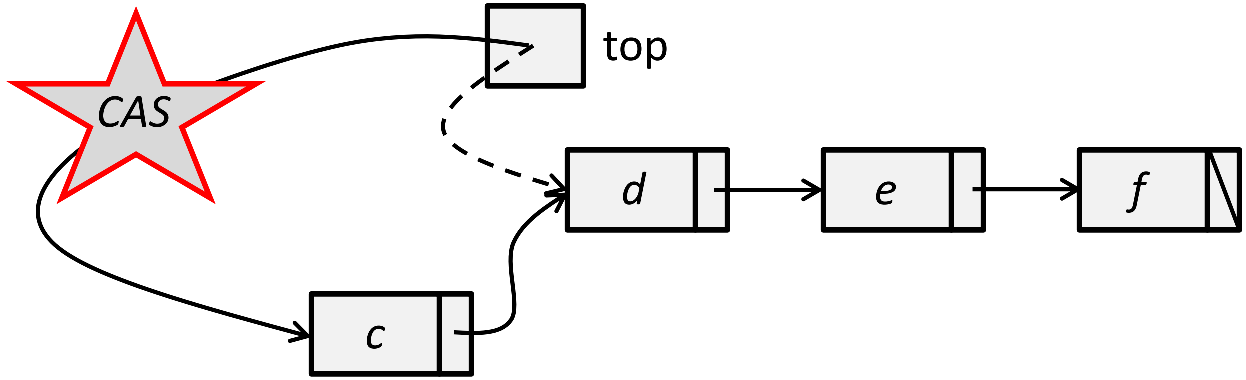
An Unbounded Lock-Free Stack: Push




```
protected boolean tryPush( Node node ) {
    Node oldTop = top.get( );
    node.next = oldTop;
    return ( top.compareAndSet( oldTop, node ) );
}
public void push( T value ) {
    Node node = new Node( value );
    while ( true ) {
        if ( tryPush( node ) ) { return; }
        else { backoff.backoff( ); }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

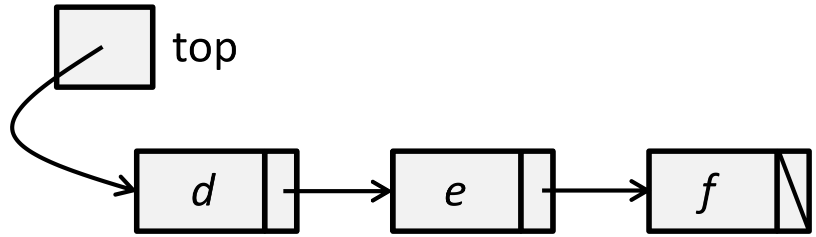
An Unbounded Lock-Free Stack: Push




```
protected boolean tryPush( Node node ) {
    Node oldTop = top.get( );
    node.next = oldTop;
    return ( top.compareAndSet( oldTop, node ) );
}
public void push( T value ) {
    Node node = new Node( value );
    while ( true ) {
        if ( tryPush( node ) ) { return; }
        else { backoff.backoff( ); }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

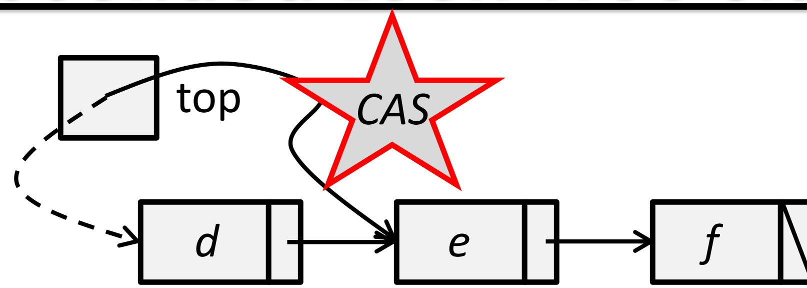
An Unbounded Lock-Free Stack: Pop




```
protected Node tryPop( ) throws EmptyException {
    Node oldTop = top.get( );
    if ( oldTop == null ) {
        throw new EmptyException( );
    }
    Node newTop = oldTop.next;
    if ( top.compareAndSet( oldTop, newTop ) ) { return oldTop; }
    else { return null; }
}
public T pop( ) throws EmptyException {
    while ( true ) {
        Node returnNode = tryPop( );
        if ( returnNode != null ) { return returnNode.value; }
        else { backoff.backoff( ); }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

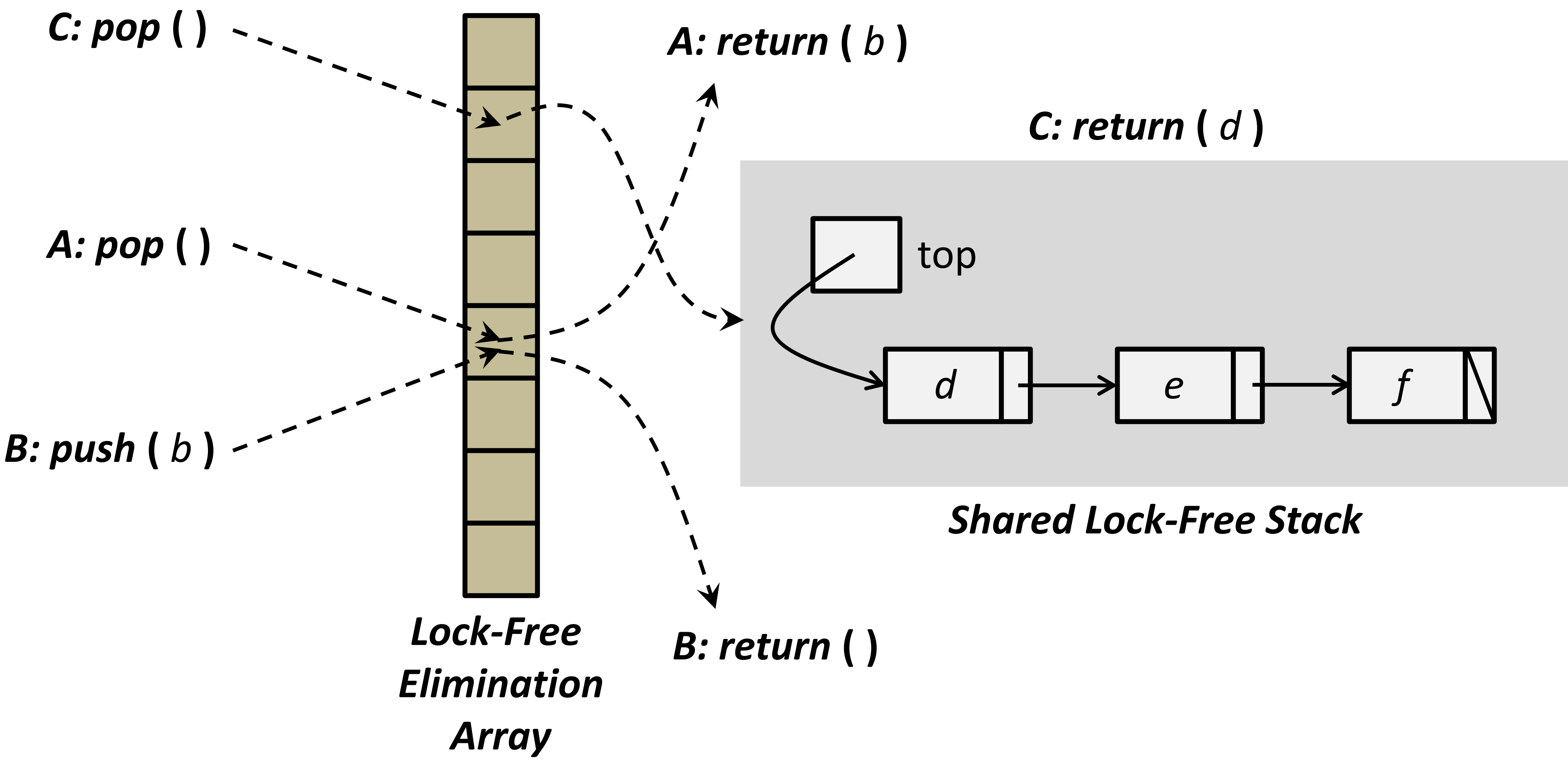
An Unbounded Lock-Free Stack: Pop




```
protected Node tryPop( ) throws EmptyException {
    Node oldTop = top.get( );
    if ( oldTop == null ) {
        throw new EmptyException( );
    }
    Node newTop = oldTop.next;
    if ( top.compareAndSet( oldTop, newTop ) ) { return oldTop; }
    else { return null; }
}
public T pop( ) throws EmptyException {
    while ( true ) {
        Node returnNode = tryPop( );
        if ( returnNode != null ) { return returnNode.value; }
        else { backoff.backoff( ); }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

Elimination-Backoff Stack



Elimination Array

```
public class EliminationArray< T > {
    private static final int duration = ...;
    LockFreeExchanger< T >[ ] exchanger;
    Random rand;
    public EliminationArray( int capacity ) {
        exchanger = ( LockFreeExchanger< T >[ ] ) new LockFreeExchanger[ capacity ];
        for ( int i = 0; i < capacity; i++ ) {
            exchanger[ i ] = new LockFreeExchanger< T >( );
        }
        rand = new Random( );
    }
    public T visit( T value, int range ) throws TimeoutException {
        int slot = rand.nextInt( range );
        return ( exchanger[ slot ].exchange( value, duration, TimeUnit.MILLISECONDS ) );
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition

An Unbounded Lock-Free Elimination-Backoff Stack

```
public class EliminationBackoffStack< T > extends LockFreeStack< T > {
    static final int capacity = ...;
    EliminationArray< T > eliminationArray = new EliminationArray< T >( capacity );
    static int range =  ...;

    public void push( T value ) {
        Node node = new Node( value );
        while ( true ) {
            if ( tryPush( node ) ) { return; }
            else try {
                T otherValue = eliminationArray.visit( value, range );
                if ( otherValue == null ) { return; } // exchanged with pop
            }  catch ( TimeoutException ex ) { }
        }
    }
```

Source: Herlihy & Shavit., "The Art of Multiprocessor Programming", 1st Edition
(modified)

An Unbounded Lock-Free Elimination-Backoff Stack

```
public T pop( ) throws EmptyException {
    while ( true ) {
        Node returnNode = tryPop( );
        if ( returnNode != null ) { return returnNode.value; }
        else try {
            T otherValue = eliminationArray.visit( null, range );
            if ( otherValue != null ) { return otherValue; }
        } catch ( TimeoutException ex ) { }
    }
}
```

Source: Herlihy & Shavit., “The Art of Multiprocessor Programming”, 1st Edition (modified)