Practical Concurrent and Parallel Programming VII

Lock-Free Data Structures

Raúl Pardo

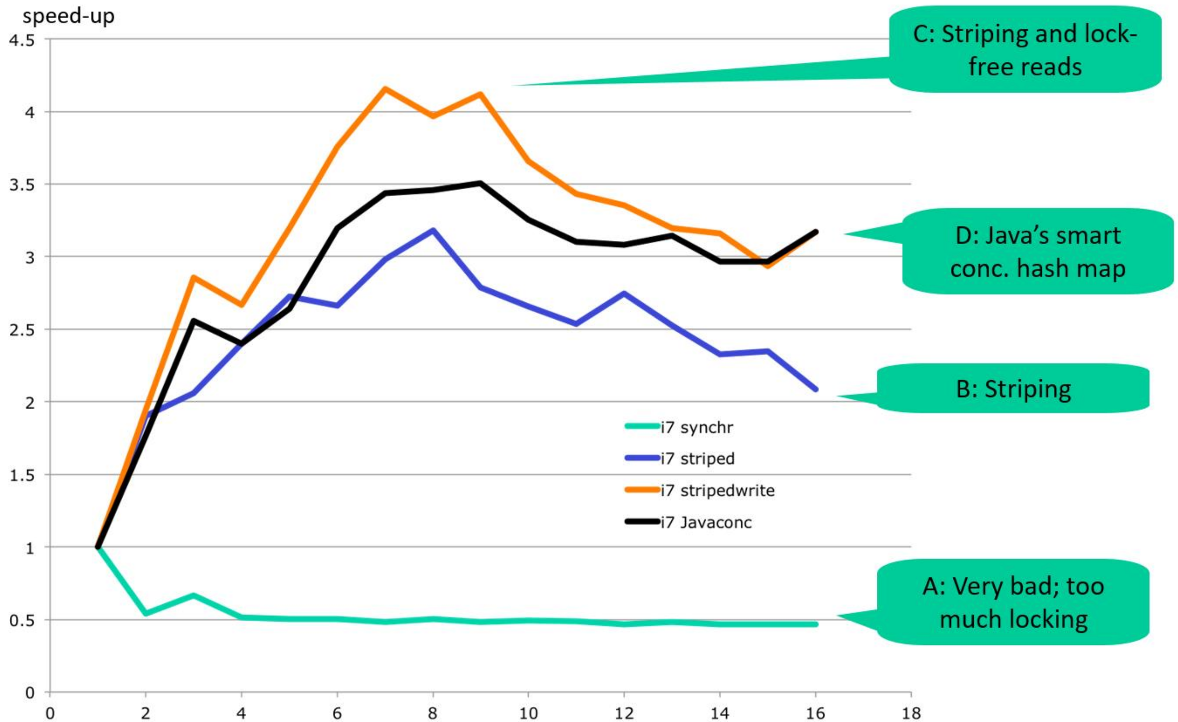
Based on slides of PCPP 2019

Troubles running Gradle?

https://www.menti.com/aljogfb2pj76

Previously on PCPP…

Reducing locking

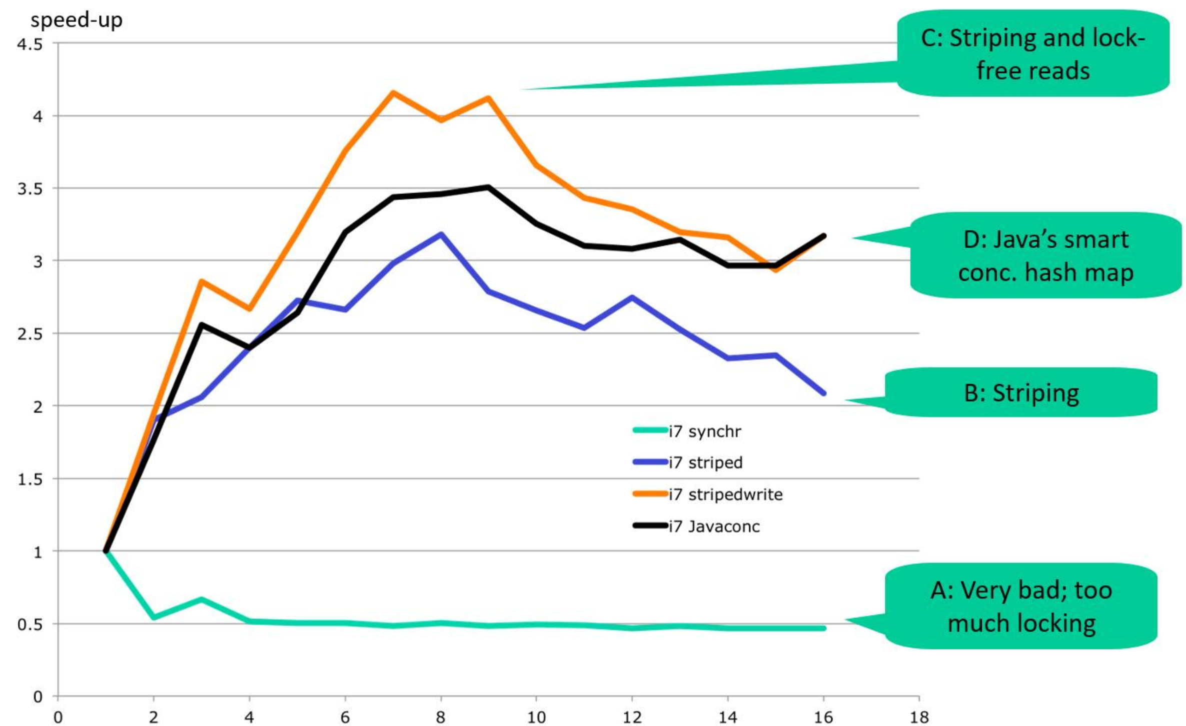


IT UNIVERSITY OF COPENHAGEN

© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Previously on PCPP…

Reducing locking



This lecture:
Can we get rid of locks altogether?

IT UNIVERSITY OF COPENHAGEN
© Raúl Pardo Jimenez and Jørgen Staunstrup – F2023

Agenda

- Compare-And-Swap (CAS)
  - Lock-free atomic integer
  - Lock-free number range
  - Atomic libraries
  - CAS based lock implementation
- Lock-free stack
- ABA problem
- Lock-free queue

HW support for atomic compound operations

- Early processors had atomic compound operations to implement mutexes
  - *test-and-set*
  - Not suitable for implementing advanced lock-free data structures

See Herlihy Sections 5.1 – 5.8
(outside the scope of the course)

- Modern processors provide special instructions to for managing concurrent access to shared variables
  - *store-conditional*, ***compare-and-swap*** **(CAS)**

HW support for Compare-And-Swap (Intel)

- Intel® 64 and IA-32 Architectures Software Developer's Manual
  - *"The CMPXCHG […] If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register).*
    […]
    For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically."

HW support for Compare-And-Swap (Intel)

- Intel® 64 and IA-32 Architectures Software Developer's Manual
  - *"The CMPXCHG [...] If the values contained in the destination operand and the EAX register are equal, the destination operand is replaced with the value of the other source operand (the value not in the EAX register).*
    [...]
    For multiple processor systems, CMPXCHG can be combined with the LOCK prefix to perform the compare and exchange operation atomically."
  - *Intel 64 and IA-32 processors provide a LOCK# signal [...] While this output signal is asserted, requests from other processors or bus agents for control of the bus are blocked [...]*

Compare-And-Swap (CAS)

- At the programming language level, most languages support an operation: **`val.compareAndSwap(a,b)`**
  - Compares the value of `val` and `a`, and, if they are equal `val` is set to `b`, otherwise it does nothing. In either case, it returns the current value in `val`, i.e., the value when CAS was executed
  - This instruction is translated to CMPXCHG (or similar) by the compiler or runtime environment

CAS pseudo-code


```
class MyAtomicInteger {
    private int value;    // Visibility ensured by locking

    public synchronized int compareAndSwap(int oldValue, int newValue) {
        int valueInRegister = this.value;
        if (this.value == oldValue)
            this.value = newValue;
        return valueInRegister;
    }

    public synchronized boolean compareAndSet(int oldValue, int newValue) {
        return oldValue == this.compareAndSwap(oldValue,newValue);
    }

    public synchronized int get() { return this.value; }
...
}
```

Illustrative implementation of CAS

Common alternative.
Also abbreviated as CAS
(optional exercise: implement natively compareAndSet)

- Only to *illustrate* CAS semantics
    - CAS is implemented via a HW operation (e.g., CMPXCHG in Intel architectures)
    - In fact, locks are implemented using CAS

AtomicInteger operations via CAS

- Standard AtomicInteger operations can now be implemented using the CAS operations *without blocking*

- This is an example of *optimistic* concurrency (or non-blocking computation)
    - In a nutshell, trying several times until the operation succeeds

```
class MyAtomicInteger {
…
  public int addAndGet(int delta) {
     int oldValue, newValue;
     do {
         oldValue = get();
         newValue = oldValue + delta;
     } while (!compareAndSet(oldValue, newValue));
     return newValue;
  }

  public int getAndAdd(int delta) {
     int oldValue, newValue;
     do {
         oldValue = get();
         newValue = oldValue + delta;
     } while (!compareAndSet(oldValue, newValue));
     return oldValue;
  }
…
}
```

AtomicInteger operations via CAS

- Standard AtomicInteger operations can now be implemented using the CAS operations *without blocking*

- This is an example of *optimistic* concurrency (or non-blocking computation)
    - In a nutshell, trying several times until the operation succeeds

```
class MyAtomicInteger {
…
  public int addAndGet(int delta) {
      int oldValue, newValue;
      do {
          oldValue = get();
          newValue = oldValue + delta;
      } while (!compareAndSet(oldValue, newValue));
      return newValue;
  }

  public int getAndAdd(int delta) {
      int oldValue, newValue;
      do {
          oldValue = get();
          newValue = oldValue + delta;
      } while (!compareAndSet(oldValue, newValue));
      return oldValue;
  }
…
}
```

Is this the same as busy-wait?

A note on busy-wait

Déjà vu from lecture 2

- *Busy-wait* is an *alternative to blocking* a thread to wait until some condition holds or to enter the critical section
- The main difference with **`lock()`** or **`await()`** is that the thread does not transition to the "blocked" state
- Generally, busy-wait is a bad idea,
  - Threads may consume computing resources to check a condition that has not been updated
  - In this course, we will never ask you to use busy-wait
  - *Exercise solutions using busy-wait will be rejected*
- Very rarely busy-wait may be preferred over blocking the thread
  - When the thread waits for a very short time it might be more efficient to use busy-wait
  - However, as we have discussed, reasoning about how it takes for a thread to do anything is pointless in concurrency

```
…
// state variables
int i = 0;
Lock l = new ReentrantLock();
…

// method example
public void method(…) {
     l.lock()
     try{
          // busy-wait
          while(i>0) {
               // do nothing
          }
     }
     catch (InterruptedException e) {…}
     finally {l.unlock();}
}
```

Progress conditions for non-blocking DS

There exist 3 main notions of progress in non-blocking data structures/computation

1. *Wait-free*: A method of an object implementation is wait-free if every call finishes its execution in a finite number of steps
   - My operations are guaranteed to complete in a bounded number of steps (no matter what other threads do)

Herlihy, Section 3.8 | slide from Michael Scott's presentation: https://www.youtube.com/watch?v=9XAx279s7gs

Progress conditions for non-blocking DS

There exist 3 main notions of progress in non-blocking data structures/computation

1. *Wait-free*: A method of an object implementation is wait-free if every call finishes its execution in a finite number of steps
   - My operations are guaranteed to complete in a bounded number of steps (no matter what other threads do)

2. *Lock-free*: A method of an object implementation is lock-free if executing the method guarantees that some method call (including concurrent) finishes in a finite number of steps
   - Somebody's operations are guaranteed to complete in a bounded number of my steps
   - Most non-blocking data structures are lock-free, e.g., Treabor's stack (see next slides)

Herlihy, Section 3.8 | slide from Michael Scott's presentation: https://www.youtube.com/watch?v=9XAx279s7gs

Progress conditions for non-blocking DS

There exist 3 main notions of progress in non-blocking data structures/computation

1. *Wait-free*: A method of an object implementation is wait-free if every call finishes its execution in a finite number of steps
   - My operations are guaranteed to complete in a bounded number of steps (no matter what other threads do)

2. *Lock-free*: A method of an object implementation is lock-free if executing the method guarantees that some method call (including concurrent) finishes in a finite number of steps
   - Somebody's operations are guaranteed to complete in a bounded number of my steps
   - Most non-blocking data structures are lock-free, e.g., Treabor's stack (see next slides)

3. *Obstruction-free*: A method of an object implementation is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps;
   - My operations are guaranteed to complete in a bounded of steps (if I get to execute them)

Herlihy, Section 3.8 | slide from Michael Scott's presentation: https://www.youtube.com/watch?v=9XAx279s7gs

Progress conditions for non-blocking DS

There exist 3 main notions of progress in non-blocking data structures/computation

1. *Wait-free*: A method of an object implementation is wait-free if every call finishes its execution in a finite number of steps
   - My operations are guaranteed to complete in a bounded number of steps (no matter what other threads do)

2. *Lock-free*: A method of an object implementation is lock-free if executing the method guarantees that some method call (including concurrent) finishes in a finite number of steps
   - Somebody's operations are guaranteed to complete in a bounded number of my steps
   - Most non-blocking data structures are lock-free, e.g., Treabor's stack (see next slides)

3. *Obstruction-free*: A method of an object implementation is obstruction-free if, from any point after which it executes in isolation, it finishes in a finite number of steps;
   - My operations are guaranteed to complete in a bounded of steps (if I get to execute them)

$$wait_{free} \Rightarrow lock_{free} \Rightarrow obstruction_{free}$$

Herlihy, Section 3.8 | slide from Michael Scott's presentation: https://www.youtube.com/watch?v=9XAx279s7gs

Why non-blocking ≠ busy-wait?

- Threads cannot wait forever because other thread finished incorrectly
  - Even in obstruction-free, completion must be guaranteed when the thread runs in isolation
- All progress definitions for non-blocking programs are part of an algorithm composed by multiple threads
  - Threads collaborate with each other

AtomicInteger operations via CAS

- Standard AtomicInteger operations can now be implemented using the CAS

What type of non-blocking progress applies to this AtomicInteger?

  - In a nutshell, trying several times until the operation succeeds

```
class MyAtomicInteger {
…
  public int addAndGet(int delta) {
      int oldValue, newValue;
      do {
          oldValue = get();
          newValue = oldValue + delta;
      } while (!compareAndSet(oldValue, newValue));
      return newValue;
  }

  public int getAndAdd(int delta) {
      int oldValue, newValue;
      do {
          oldValue = get();
          newValue = oldValue + delta;
      } while (!compareAndSet(oldValue, newValue));
      return oldValue;
  }
…
}
```

CAS implementation in Java

Most programming languages have similar lock-free libraries

- In Java, CAS and similar operations are accessed through **`AtomicXX`** objects
  - These classes support atomic lock-free operations for single variables
  - See package [java.util.concurrent.atomic]

- **`AtomicXX`** CAS operations have the same memory semantics as volatile read/write
  - *"The memory effects for accesses and updates of atomics generally follow the rules for volatiles"* [java.util.concurrent.atomic]
  - Unlike in `AtomicXX`, recall that volatile reads/writes are not atomic!

A note on correctness and thread-safety

- The volatile visibility semantics of `AtomicXX` objects allow us to reason about safe-publication
- However, operations on `AtomicXX` are not explicitly part of the Java memory model
  - We cannot use them to reason about correctness or thread-safety
- In this lecture (and exercises), we use specifications to establish correctness (*à la* week 4)
- In two weeks, we will see a finer-grained definition of correctness for concurrent objects, Linearizability

What if we have to update multiple fields?

```
public class NumberRange {

  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public int getLower() { return lower.get();}
  public int getUpper() { return upper.get();}

  public void setLower(int i) {
    if (i > getUpper()) {
      throw new IllegalArgumentException(
          "Can't set lower to " + i +" > upper");
    }
    lower.set(i);
  }

 public void setUpper(int i) {
    if (i < getLower()) {
      throw new IllegalArgumentException(
          ("Can't set upper to " + i +" < lower");
    }
    upper.set(i);
  }
}
```

__Specification__: For any concurrent execution of setLower() and setUpper(), it must always hold that **__lower ≤ upper__**

What if we have to update multiple fields?

```
public class NumberRange {

  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public int getLower() { return lower.get();}
  public int getUpper() { return upper.get();}

  public void setLower(int i) {
    if (i > getUpper()) {
      throw new IllegalArgumentException(
          "Can't set lower to " + i +" > upper");
    }
    lower.set(i);
  }

 public void setUpper(int i) {
    if (i < getLower()) {
      throw new IllegalArgumentException(
          ("Can't set upper to " + i +" < lower");
    }
    upper.set(i);
  }
}
```

Specification: For any concurrent execution of setLower() and setUpper(), it must always hold that **lower ≤ upper**

Does this implementation of setLower and setUpper satisfy the specification?

What if we have to update multiple fields?

```java
public class NumberRange {

  private final AtomicInteger lower = new AtomicInteger(0);
  private final AtomicInteger upper = new AtomicInteger(0);

  public int getLower() { return lower.get();}
  public int getUpper() { return upper.get();}

  public void setLower(int i) {
    if (i > getUpper()) {
      throw new IllegalArgumentException(
          "Can't set lower to " + i +" > upper");
    }
    lower.set(i);
  }

 public void setUpper(int i) {
    if (i < getLower()) {
      throw new IllegalArgumentException(
          ("Can't set upper to " + i +" < lower");
    }
    upper.set(i);
  }
}
```

Specification: For any concurrent execution of setLower() and setUpper(), it must always hold that **lower ≤ upper**

Goetz p. 67, Don't do this

What if we have to update multiple fields?

```java
public class CasNumberRange {

  private final AtomicReference<IntPair> values =
    new AtomicReference<IntPair>(new IntPair(0,0));

  public int getLower() { return values.get().lower;}
  public int getUpper() { return values.get().upper;}

  public void setLower(int i) {
    while(true) {
      IntPair oldv = values.get();
      if (i > oldv.upper) {
        throw new IllegalArgumentException(
          "Can't set lower to " + i +" > upper");
      }
      IntPair newv = new IntPair(i, oldv.upper);
      if(values.compareAndSet(oldv,newv))
        return;
    }
  }

  … // Similarly for setUpper();
}
```

```java
// Immutable class
private static class IntPair {
  private final int lower;
  private final int upper;

  public IntPair(int lower,
                 int upper) {
    this.lower = lower;
    this.upper = upper;
  }
}
```

Specification: For any concurrent execution of setLower() and setUpper(), it must always hold that **lower ≤ upper**

What if we have to update multiple fields?


```
public class CasNumberRange {

  private final AtomicReference<IntPair> values =
    new AtomicReference<IntPair>(new IntPair(0,0));

  public int getLower() { return values.get().lower;}
  public int getUpper() { return values.get().upper;}

  public void setLower(int i) {
    while(true) {
      IntPair oldv = values.get();
      if (i > oldv.upper) {
        throw new IllegalArgumentException(
          "Can't set lower to " + i +" > upper");
      }
      IntPair newv = new IntPair(i, oldv.upper);
      if(values.compareAndSet(oldv,newv))
        return;
    }
  }

  … // Similarly for setUpper();
}
```

```
// Immutable class
private static class IntPair {
  private final int lower;
  private final int upper;

  public IntPair(int lower,
                 int upper) {
    this.lower = lower;
    this.upper = upper;
  }
}
```

Why is it important that IntPair is immutable?

Specification: For any concurrent execution of setLower() and setUpper(), it must always hold that **lower ≤ upper**

CAS based Lock implementation

Implementing Locks using CAS

1. Simple TryLock
   - Non-blocking tryLock, the lock may be acquired only once
   - Regular unlock
2. Reentrant TryLock
   - Non-blocking tryLock, the lock may be acquired repeatedly by the same thread
   - Regular (reentrant) unlock
3. Simple Lock
   - Blocking lock, the lock may be acquired only once
   - Regular unlock
4. Reentrant Lock
   - Block lock, the lock may be acquired repeatedly by the same thread
   - Regular (reentrant) unlock

See **`TestCasLocks.java`**

1. SimpleTryLock, non-blocking

```
class SimpleTryLock {

    // Refers to holding thread, null iff unheld
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();

    public boolean tryLock() {
      final Thread current = Thread.currentThread();
      return holder.compareAndSet(null, current);
    }


    public void unlock() {
      final Thread current = Thread.currentThread();
      if (!holder.compareAndSet(current, null))
          throw new RuntimeException("Not lock holder");
    }


}
```

If the lock is free (holder *value* equals null), takes it and return true. Otherwise, holder is unmodified and returns false.

Sets holder *value* to null. If CAS returns false throws an exception indicating that this thread is not holding the lock.

1. SimpleTryLock, non-blocking

```
class SimpleTryLock {

    // Refers to holding thread, null iff unheld
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();

    public boolean tryLock() {
      final Thread current = Thread.currentThread();
      return holder.compareAndSet(null, current);
    }


    public void unlock() {
      final Thread current = Thread.currentThread();
      if (!holder.compareAndSet(current, null))
          throw new RuntimeException("Not lock holder");
    }

}
```

If the lock is free (holder *value* equals null), takes it and return true. Otherwise, holder is unmodified and returns false.

How is CAS used here to determine that the lock is not held by the thread executing unlock?

Sets holder *value* to null. If CAS returns false throws an exception indicating that this thread is not holding the lock.

2. ReentrantTryLock, non-blocking


```
class ReentrantTryLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    private volatile int holdCount = 0;

    public boolean tryLock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount++;
          return true;
      } else if (holder.compareAndSet(null, current)) {
          holdCount = 1;
          return true;
      }
      return false;
    }

    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount--;
          if (holdCount == 0)
            holder.compareAndSet(current, null);
          return;
      } else
          throw new RuntimeException("Not lock holder");
    }
}
```

2. ReentrantTryLock, non-blocking

```
class ReentrantTryLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    private volatile int holdCount = 0;

    public boolean tryLock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount++;
          return true;
      } else if (holder.compareAndSet(null, current)) {
          holdCount = 1;
          return true;
      }
      return false;
    }

    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount--;
          if (holdCount == 0)
            holder.compareAndSet(current, null);
          return;
      } else
          throw new RuntimeException("Not lock holder");
    }
}
```

If the calling thread already holds the lock, we increase the counter

If not, we try to acquire the lock and set the count to 1

2. ReentrantTryLock, non-blocking

```
class ReentrantTryLock {
    private                                          ew AtomicReference<Thread>();
    private

    public k
      final Thread cu          Thread.currentThread();
      if (holder.get() == current) {
          holdCount++;
          return true;
      } else if (holder.compareAndSet(null, current)) {
          holdCount = 1;
          return true;
      }
      return false;
    }

    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount--;
          if (holdCount == 0)
            holder.compareAndSet(current, null);
          return;
      } else
          throw new RuntimeException("Not lock holder");
    }
}
```

Do we have a race condition in this branch of the if-statement?

If the calling thread already holds the lock, we increase the counter

If not, we try to acquire the lock and set the count to 1

2. ReentrantTryLock, non-blocking

```
class ReentrantTryLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    private volatile int holdCount = 0;

    public boolean tryLock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount++;
          return true;
      } else if (holder.compareAndSet(null, current)) {
          holdCount = 1;
          return true;
      }
      return false;
    }

    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current) {
          holdCount--;
          if (holdCount == 0)
            holder.compareAndSet(current, null);
          return;
      } else
          throw new RuntimeException("Not lock holder");
    }
}
```

If the calling thread already holds the lock, we increase the counter

If not, we try to acquire the lock and set the count to 1

If the calling thread holds the hold we decrement the counter

If the counter is 0 then we release the lock

If the calling thread does not hold the lock we throw an exception

3. SimpleLock, blocking

```
class SimpleLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    // The FIFO queue of threads waiting for this lock
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
      final Thread current = Thread.currentThread();
      waiters.add(current);
      while (waiters.peek() != current || !holder.compareAndSet(null, current)) {
          LockSupport.park(this);
      }
      waiters.remove();
    }




    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.compareAndSet(current, null))
          LockSupport.unpark(waiters.peek());
      else
          throw new RuntimeException("Not lock holder");
    }
}
```

3. SimpleLock, blocking


```
class SimpleLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    // The FIFO queue of threads waiting for this lock
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
     final Thread current = Thread.currentThread();
     waiters.add(current);
     while (waiters.peek() != current || !holder.compareAndSet(null, current)) {
         LockSupport.park(this);
     }
     waiters.remove();
    }


    public void unlock() {
     final Thread current = Thread.currentThread();
     if (holder.compareAndSet(current, null))
         LockSupport.unpark(waiters.peek());
     else
         throw new RuntimeException("Not lock holder");
    }
}
```

First we add the thread to a waiting queue

We check whether the current thread is at the head of the queue and try to take the lock

If not successful, we put the thread to wait (see Javadoc LockSupport)

Finally, we remove the thread from the waiting list

3. SimpleLock, blocking


```
class SimpleLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    // The FIFO queue of threads waiting for this lock
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
      final Thread current = Thread.currentThread();
      waiters.add(current);
      while (waiters.peek() != current || !holder.compareAndSet(null, current)) {
          LockSupport.park(this);
      }
      waiters.remove();
    }

    public void unlock() {
      final Thread current = Thread.currentThread();
      if (holder.compareAndSet(current, null))
          LockSupport.unpark(waiters.peek());
      else
          throw new RuntimeException("Not lock holder");
    }
}
```

First we add the thread to a waiting queue

We check whether the current thread is at the head of the queue and try to take the lock

If not successful, we put the thread to wait (see Javadoc LockSupport)

Finally, we remove the thread from the waiting list

We simply try to release the lock

If successful, we wake up one of the waiting threads, if any

As before, we throw an exception if the calling thread is not the thread holding the lock

3. SimpleLock, blocking (interruptions)


```
class SimpleLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    // The FIFO queue of threads waiting for this lock
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();

    public void lock() {
      final Thread current = Thread.currentThread();
      boolean wasInterrupted = false;
      waiters.add(current);
      while (waiters.peek() != current || !holder.compareAndSet(null, current)) {
         LockSupport.park(this);
         if (Thread.interrupted())
          wasInterrupted = true;
     }
     waiters.remove();
     if (wasInterrupted)
         current.interrupt();
    }
...
}
```

We use a flag to record whether the thread has been interrupted during execution

Immediately after taking the lock, we check if the thread was interrupted

If the thread was interrupted, we call interrupt to propagate the interruption.

4. SimpleReentrantLock, blocking


```
class MyReentrantLock {
    private final AtomicReference<Thread> holder = new AtomicReference<Thread>();
    private final Queue<Thread> waiters = new ConcurrentLinkedQueue<Thread>();
    private volatile int holdCount = 0;

    public void lock() {
      final Thread current = Thread.currentThread();
      if (holder.get() == current)
          holdCount++;
      else {
          boolean wasInterrupted = false;
          waiters.add(current);
          while (waiters.peek() != current || !holder.compareAndSet(null, current)) {
            LockSupport.park(this);
            if (Thread.interrupted())
                wasInterrupted = true;
          }
          holdCount = 1;
          waiters.remove();
          if (wasInterrupted)
            current.interrupt();
      }
    }
…
}
```

Here we simply combine what we have seen in threads 2 and 3

Scalability of CAS

Scalability of CAS

- Pros
  - A CAS operation is faster than acquiring a lock
  - An unsuccessful CAS operation does not cause thread de-scheduling (blocking)

- Cons
  - CAS operations result in high memory overhead

Will CAS based implementations always scale well?
(that is, better than lock based)

Scalability of PRNG


```
// Lock-based implementation
class LockingRandom implements MyRandom {
  private long seed;
  …
  // Recipe from java.util.Random.next(int bits) documentation
  public synchronized int nextInt() {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> 16);
  }
}



// CAS based implementation
class CasRandom implements MyRandom {
  private final AtomicLong seed;
  …
  public int nextInt() {
    long oldSeed, newSeed;
    do {
      oldSeed = seed.get();
      newSeed = (oldSeed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    } while (!seed.compareAndSet(oldSeed, newSeed));
    return (int)(newSeed >>> 16);
  }
}
```

- We use a different implementations of a simple Pseudo-Random Number Generator (PRNG) to test the scalability of CAS vs Locks vs Thread Local (see next slide)

- Example execution with seed = 10
  - 3847489, 1334288366, 1486862010, 711662464, -1453296530, -775316920, 1157481928, 294681619, …

Scalability of PRNG

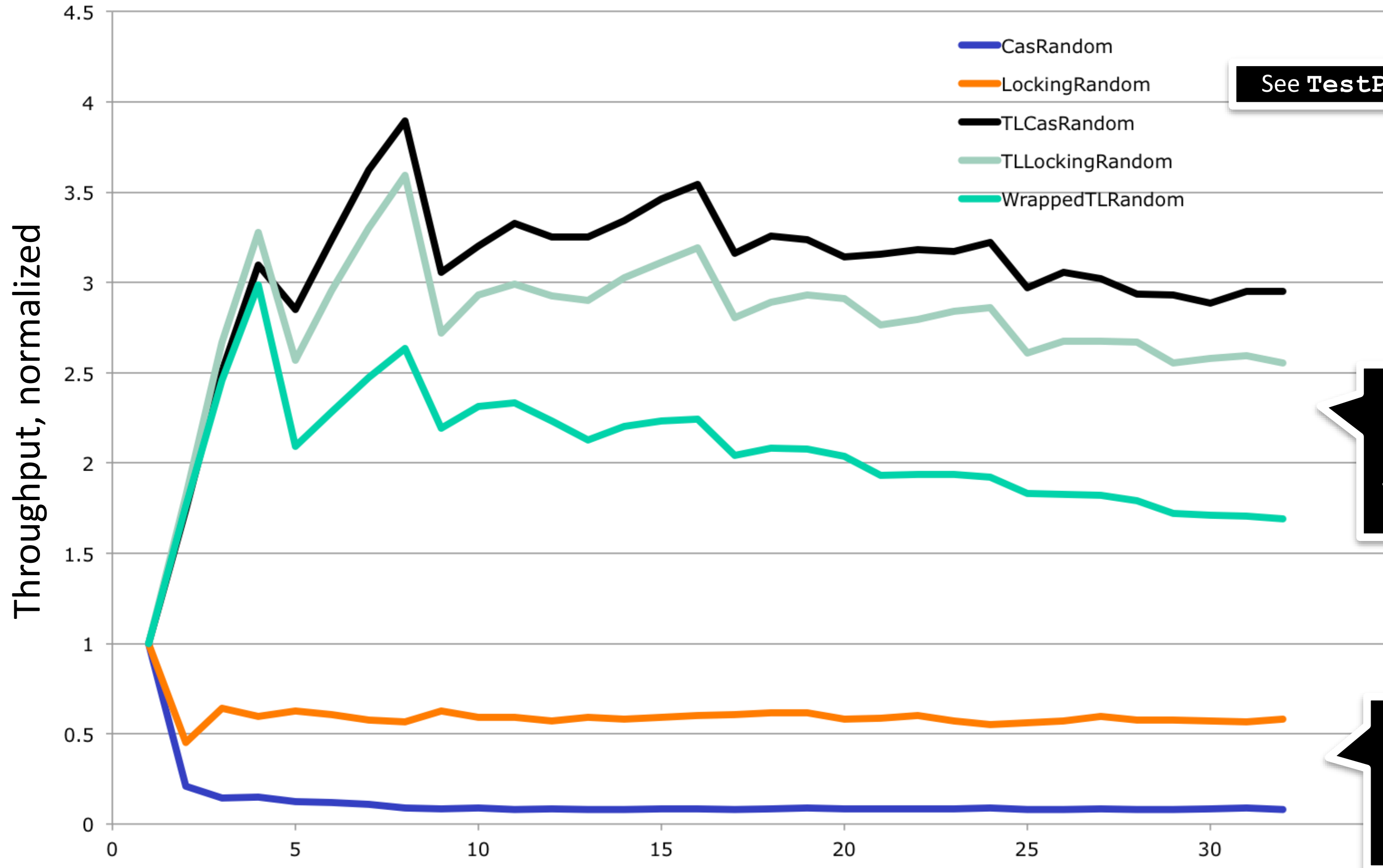

```
// Lock-based implementation
class LockingRandom implements MyRandom {
  private long seed;
  …
  // Recipe from java.util.Random.next(int bits) documentation
  public synchronized int nextInt() {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int)(seed >>> 16);
  }
}



// CAS based implementation
class CasRandom implements MyRandom {
  private final AtomicLong seed;
  …
  public int nextInt() {
    long oldSeed, newSeed;
    do {
      oldSeed = seed.get();
      newSeed = (oldSeed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    } while (!seed.compareAndSet(oldSeed, newSeed));
    return (int)(newSeed >>> 16);
  }
}
```

- We use a different implementations of a simple Pseudo-Random Number Generator (PRNG) to test the scalability of CAS vs Locks vs Thread Local (see next slide)
- Example execution with seed = 10
  - 3847489, 1334288366, 1486862010, 711662464, -1453296530, -775316920, 1157481928, 294681619, …

Could we have an implementation using volatile?

Scalability of PRNGs (unrealistic contention)

See `TestPseudoRandom.java`



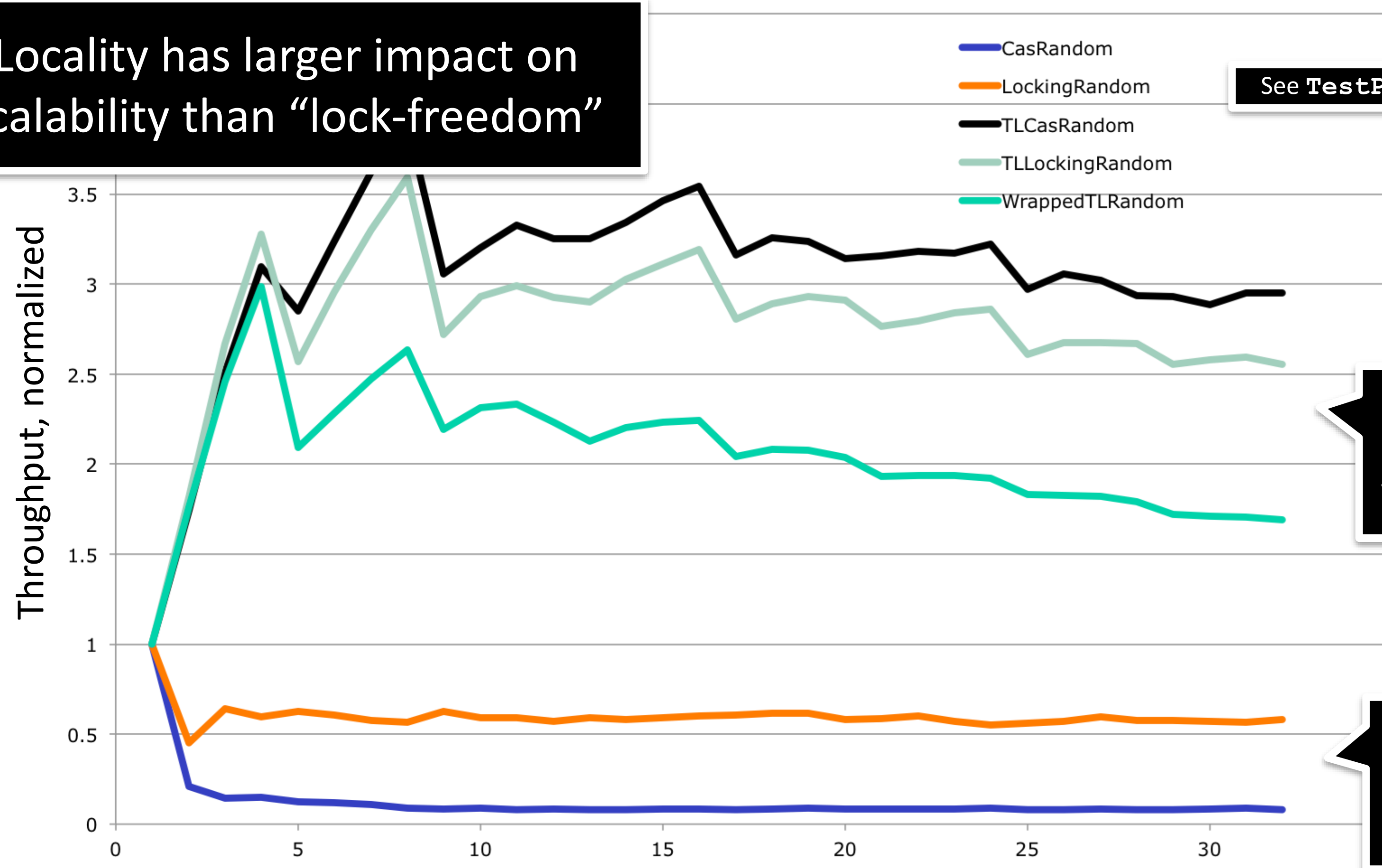
Throughput of computing on ThreadLocal objects

Throughput of computing on shared objects

Scalability of PRNGs (unrealistic contention)

Locality has larger impact on scalability than “lock-freedom”

See `TestPseudoRandom.java`



Throughput of computing on ThreadLocal objects

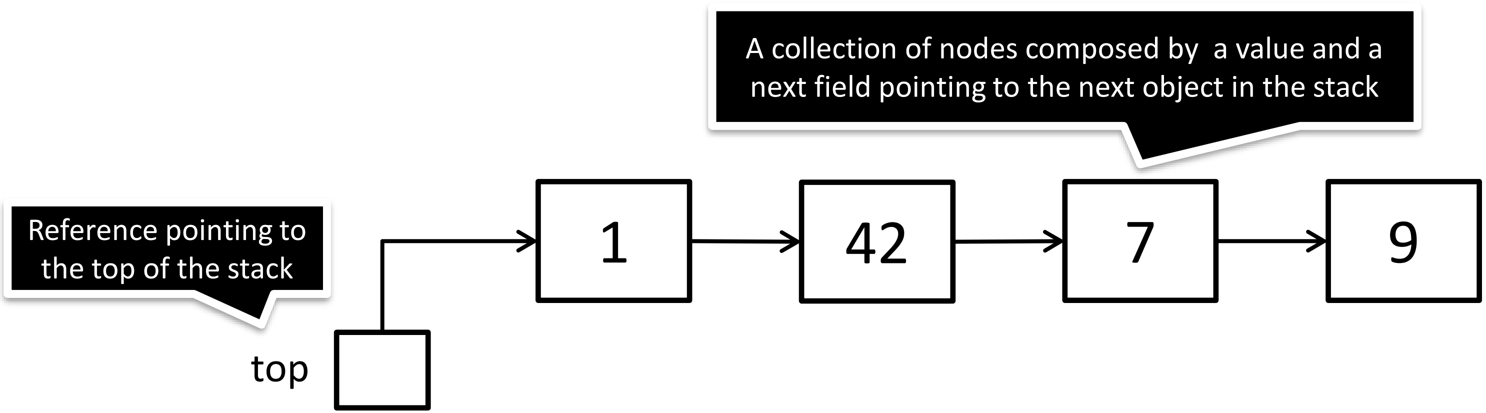
Throughput of computing on shared objects

Lock-free data structures: Stack

Stacks

- A stack is a data structure following a LIFO *(last-in-first-out)* policy
  - push() – adds an element to the top of the stack
  - pop() – removes the top of the stack

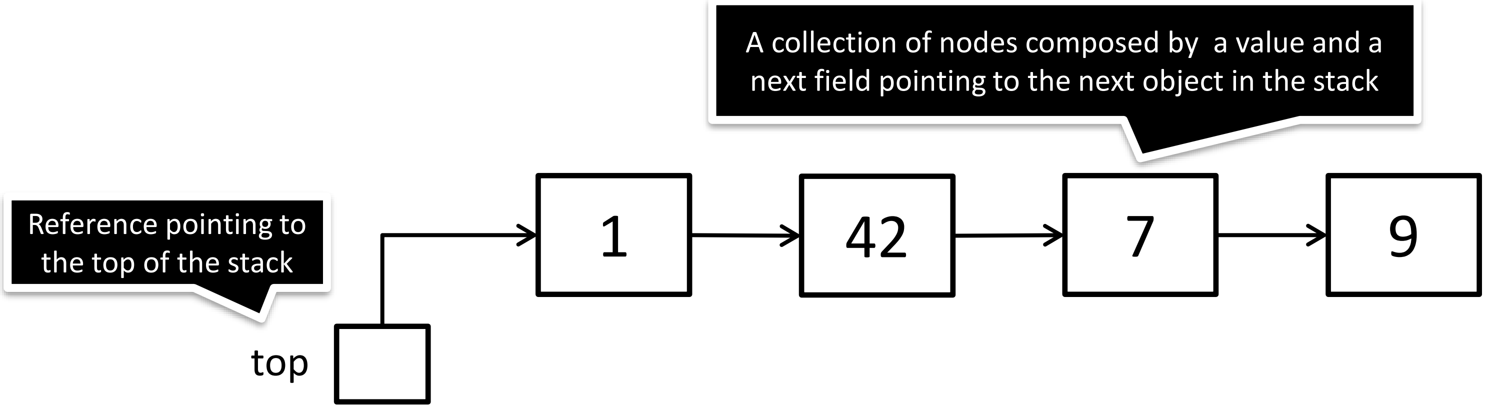
- It is typically implemented as a linked list



How do we model an empty stack?

- A stack is a data structure following a LIFO *(last-in-first-out)* policy
  - push() – adds an element to the top of the stack
  - pop() – removes the top of the stack

- It is typically implemented as a linked list



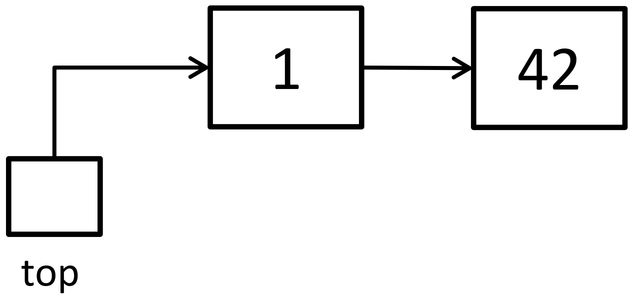
Lock-free Stack

Introduced by Treiber in 1986

```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
        new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }

    public T pop() {
        Node<T> newHead;
        Node<T> oldHead;
        do {
         oldHead = top.get();
         if(oldHead == null) { return null; }
         newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead,newHead));
        return oldHead.value;
    }
}
```

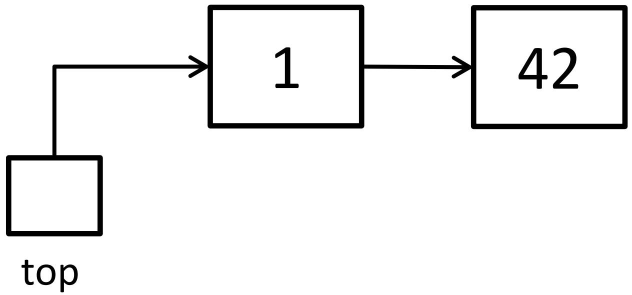
See `TestLockFreeStack.java`

```
private static class Node {
  private final T item;
  private Node<T> next;

  public Node(T item) {
    this.item = item;
    this.next = null;
  }
}
```



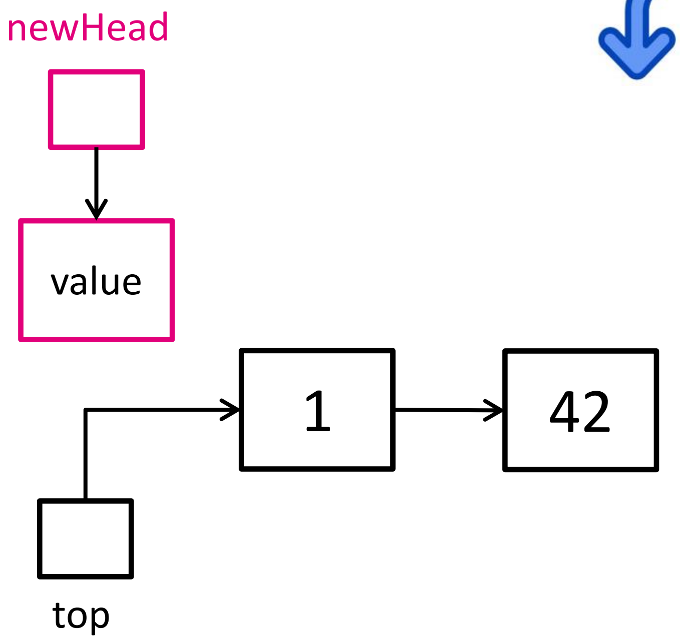
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
   …
}
```



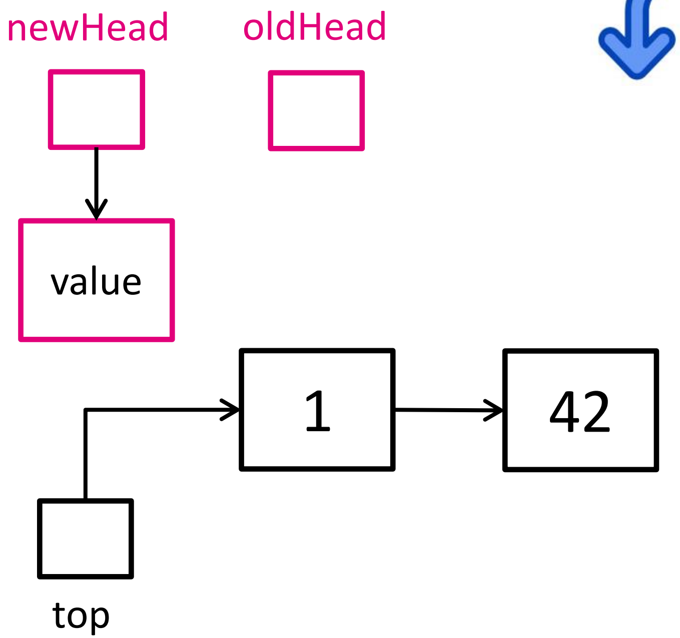
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
   …
}
```



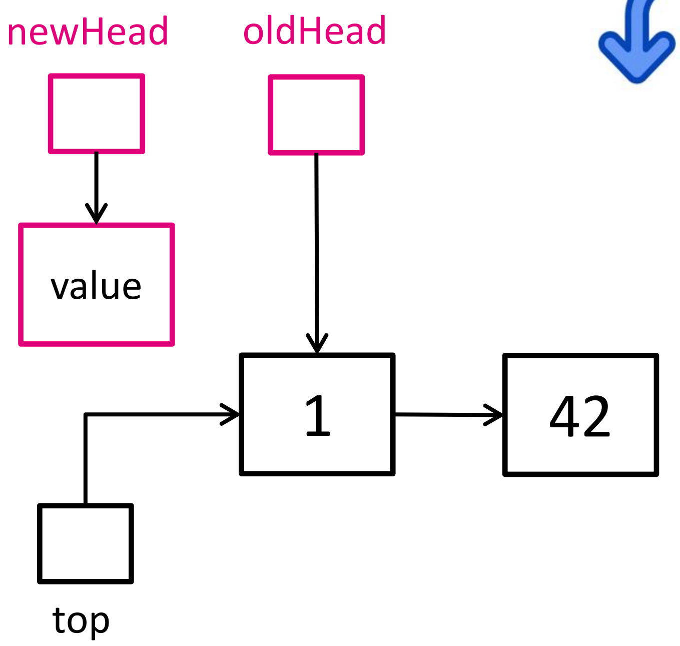
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
  …
}
```



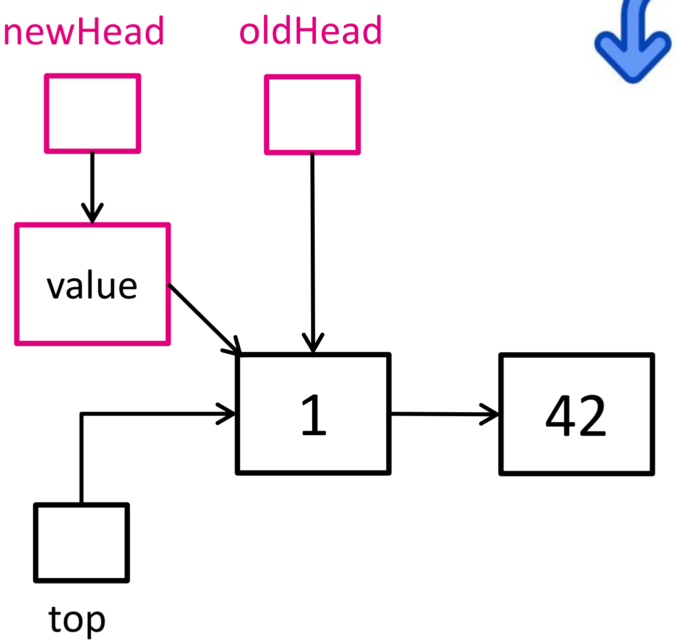
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
  …
}
```



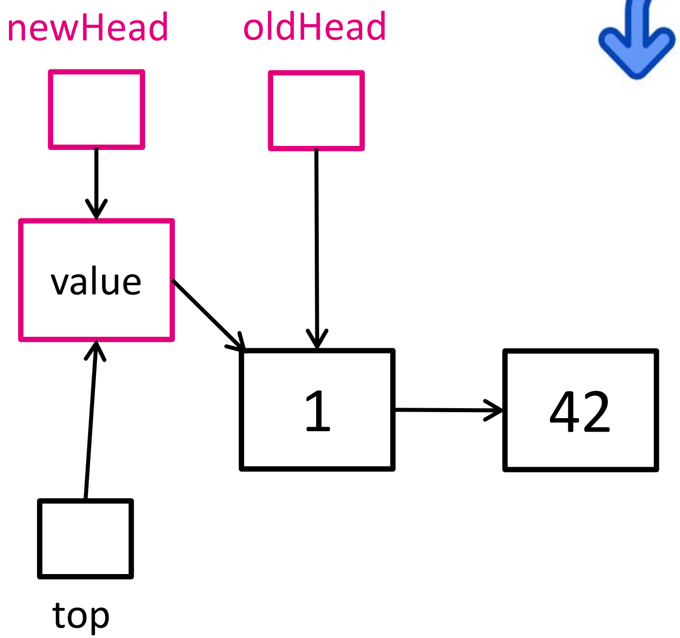
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
  …
}
```



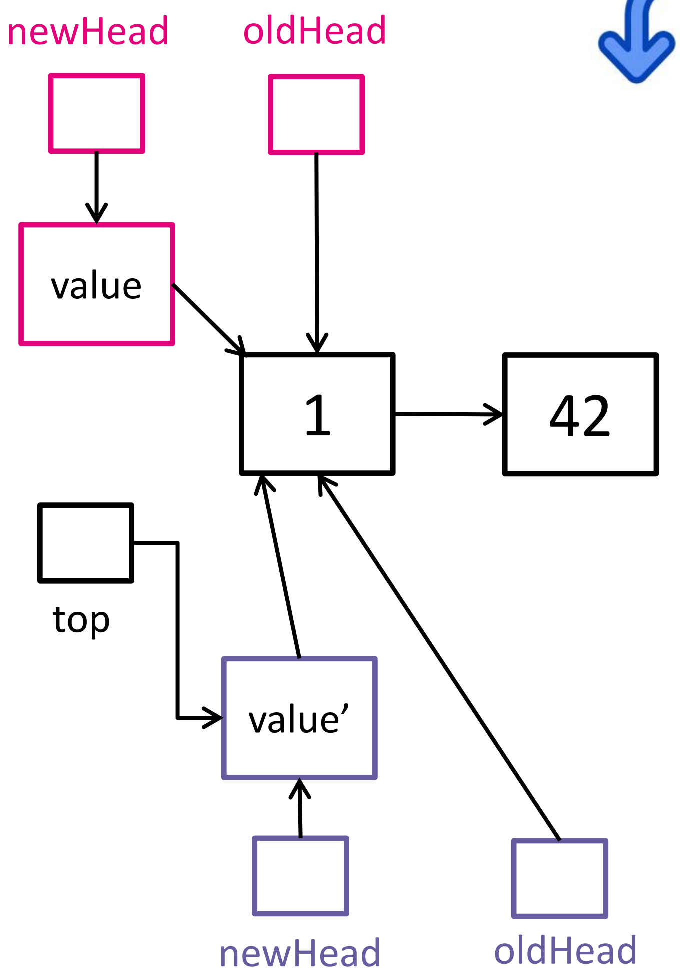
Lock-free Heap | push()


```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
  …
}
```



Lock-free Heap | push()

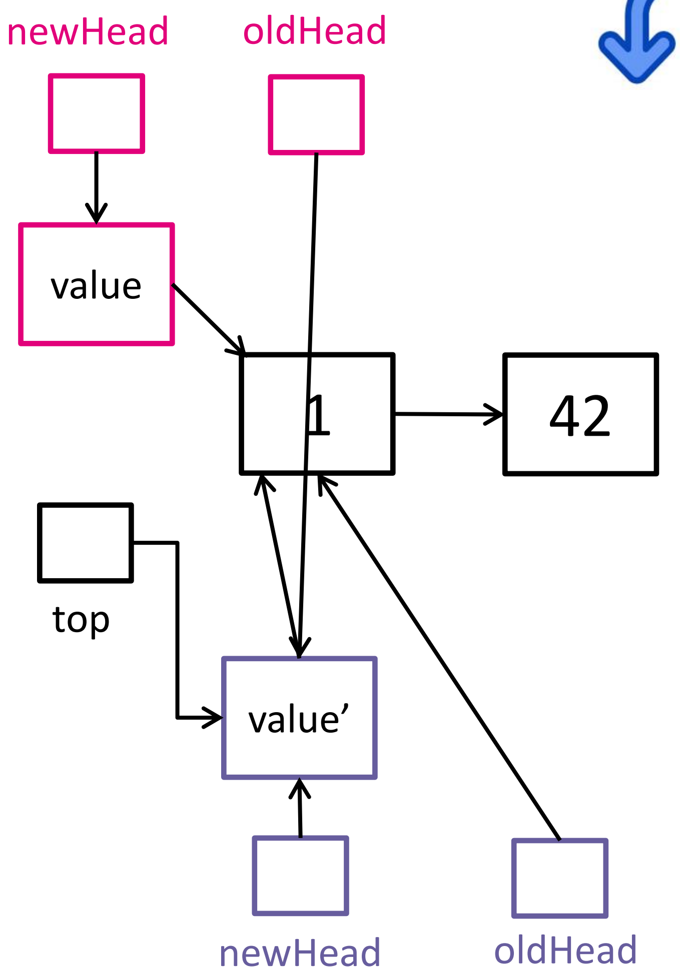

```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
   …
}
```



If before the pink thread executes CAS another thread had pushed concurrently, then CAS fails and push restarts

Lock-free Heap | push()

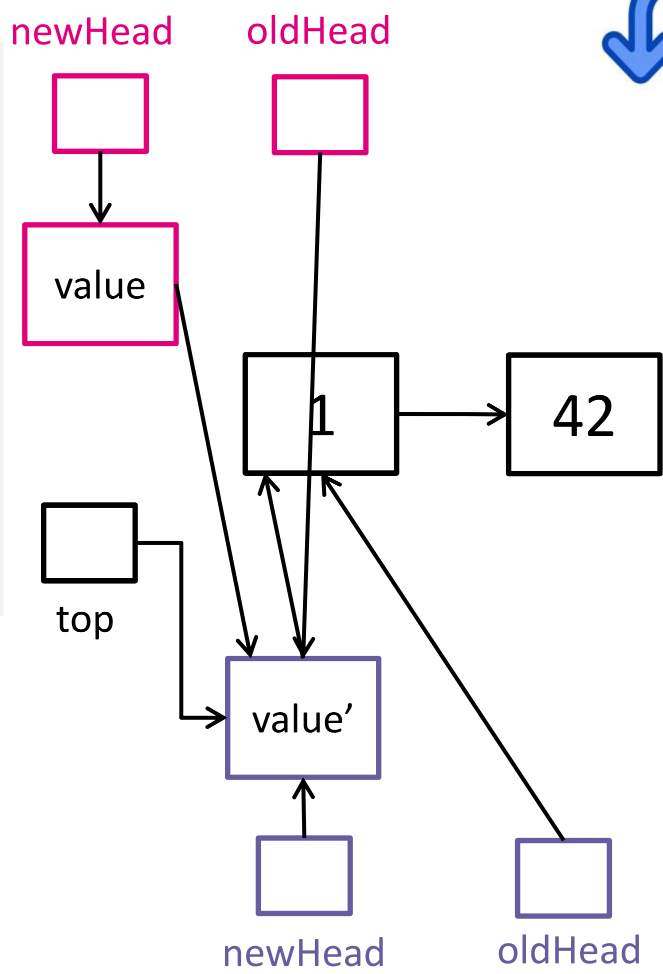

```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
   …
}
```



If before the pink thread executes CAS another thread had pushed concurrently, then CAS fails and push restarts

Lock-free Heap | push()

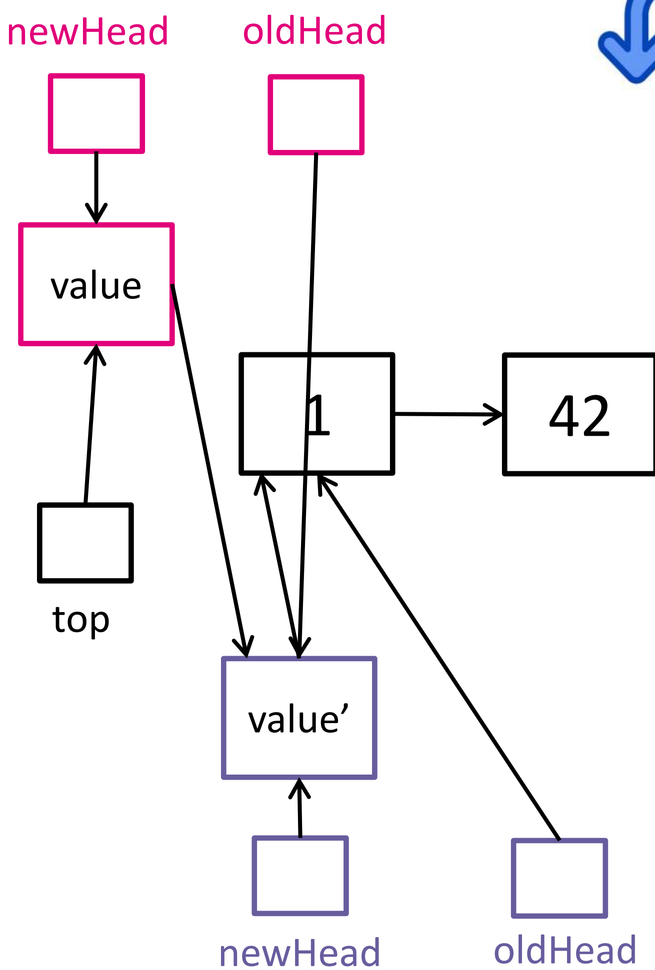
```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
  …
}
```



If before the pink thread executes CAS another thread had pushed concurrently, then CAS fails and push restarts

Lock-free Heap | push()

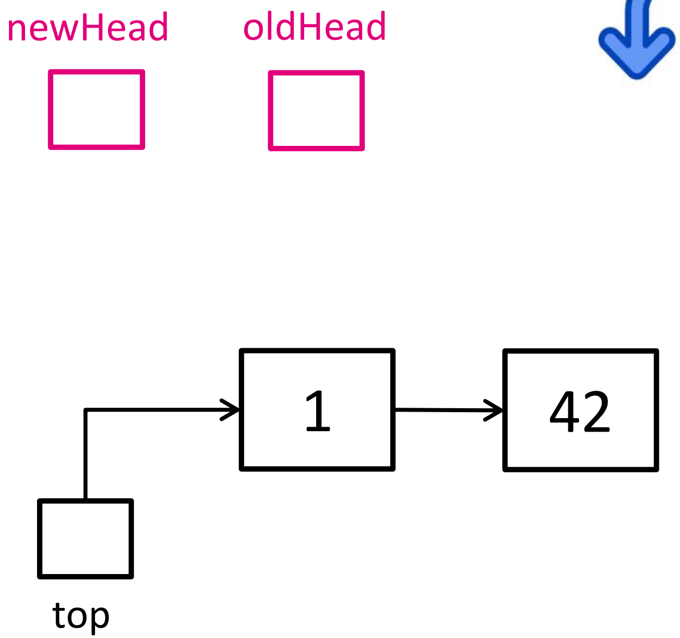

```
class LockFreeStack<T> {
    AtomicReference<Node<T>> top =
         new AtomicReference<Node<T>>();

    public void push(T value) {
        Node<T> newHead = new Node<T>(value);
        Node<T> oldHead;
        do {
         oldHead      = top.get();
         newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead,newHead));
    }
   …
}
```



If before the pink thread executes CAS another thread had pushed concurrently, then CAS fails and push restarts

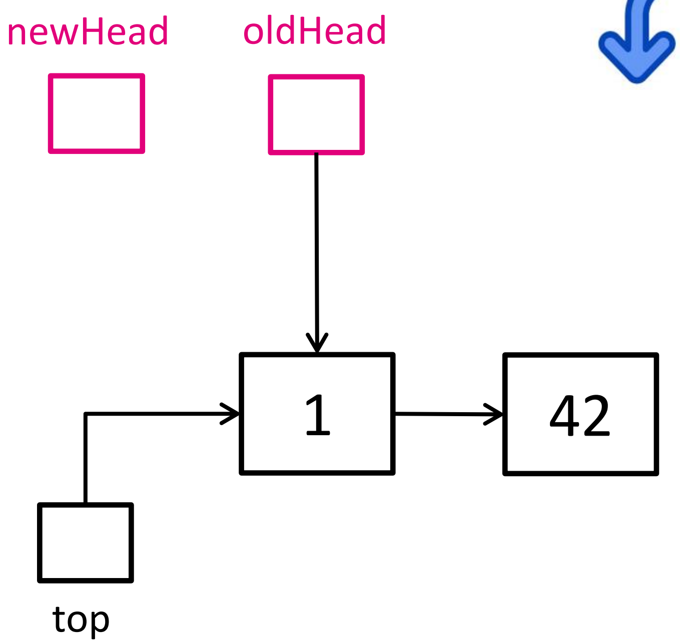
Lock-free Heap | pop()


```
class LockFreeStack<T> {
…
     public T pop() {
          Node<T> newHead;
          Node<T> oldHead;
          do {
           oldHead = top.get();
           if(oldHead == null) { return null; }
           newHead = oldHead.next;
          } while (!top.compareAndSet(oldHead,newHead));
          return oldHead.value;
     }
 }
```



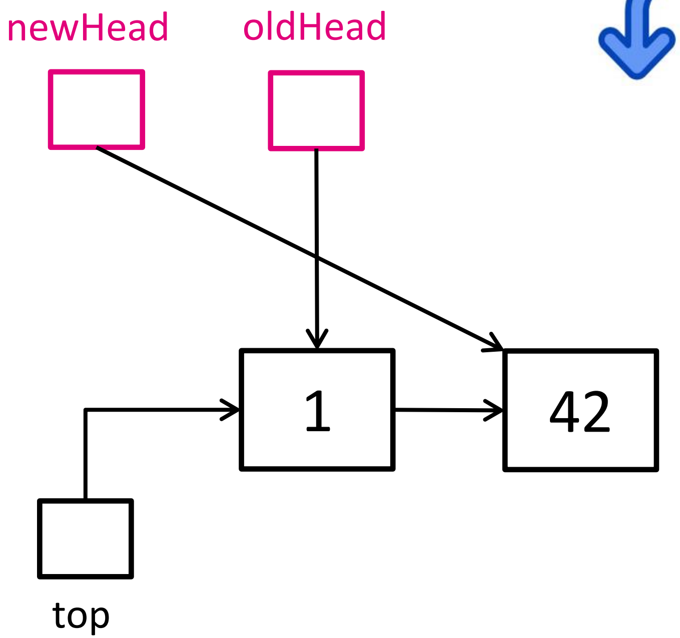
Lock-free Heap | pop()


```
class LockFreeStack<T> {
…
     public T pop() {
          Node<T> newHead;
          Node<T> oldHead;
          do {
           oldHead = top.get();
           if(oldHead == null) { return null; }
           newHead = oldHead.next;
          } while (!top.compareAndSet(oldHead,newHead));
          return oldHead.value;
     }
 }
```



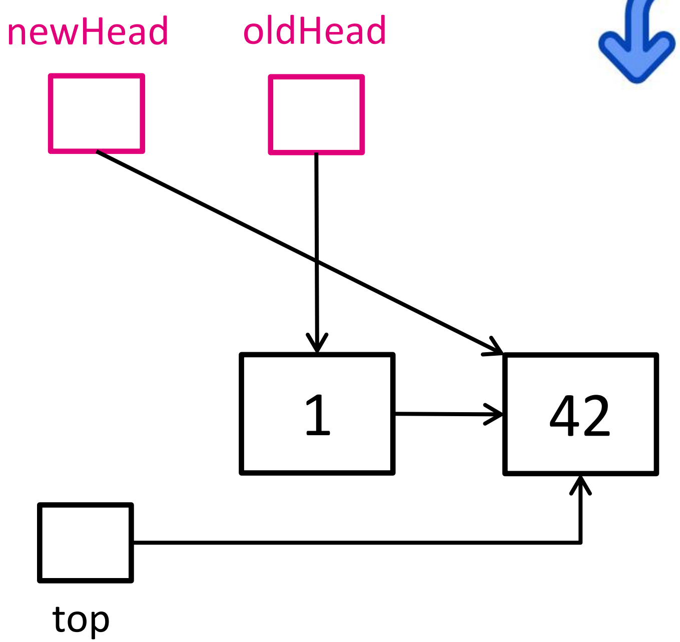
Lock-free Heap | pop()

```
class LockFreeStack<T> {
…
     public T pop() {
          Node<T> newHead;
          Node<T> oldHead;
          do {
           oldHead = top.get();
           if(oldHead == null) { return null; }
           newHead = oldHead.next;
          } while (!top.compareAndSet(oldHead,newHead));
          return oldHead.value;
     }
 }
```



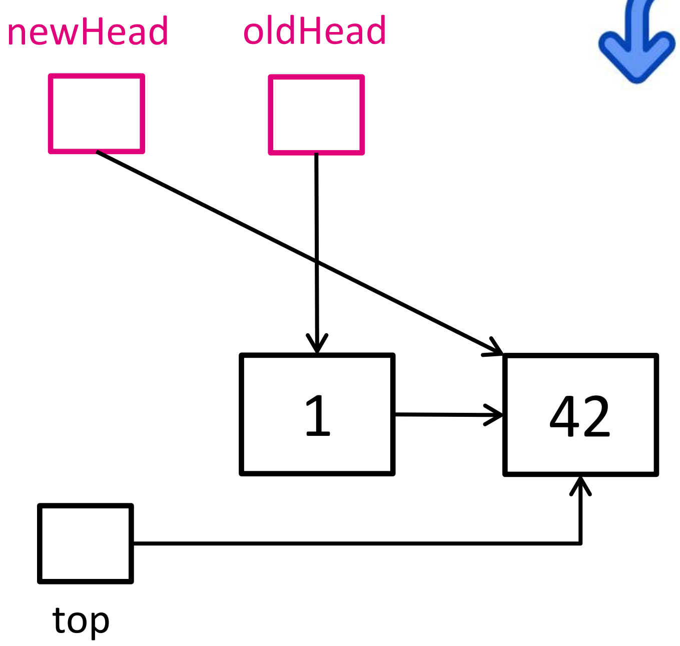
Lock-free Heap | pop()


```
class LockFreeStack<T> {
…
     public T pop() {
          Node<T> newHead;
          Node<T> oldHead;
          do {
           oldHead = top.get();
           if(oldHead == null) { return null; }
           newHead = oldHead.next;
          } while (!top.compareAndSet(oldHead,newHead));
          return oldHead.value;
     }
 }
```



Lock-free Heap | pop()


```
class LockFreeStack<T> {
…
     public T pop() {
          Node<T> newHead;
          Node<T> oldHead;
          do {
           oldHead = top.get();
           if(oldHead == null) { return null; }
           newHead = oldHead.next;
          } while (!top.compareAndSet(oldHead,newHead));
          return oldHead.value;
     }
 }
```

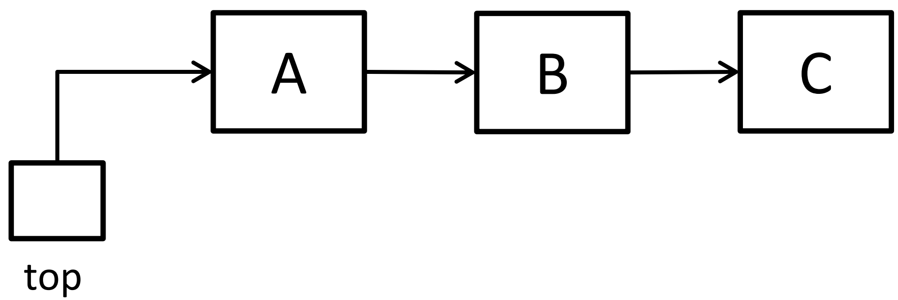


Would there be any problem if another thread is executing pop() concurrently?

The ABA Problem

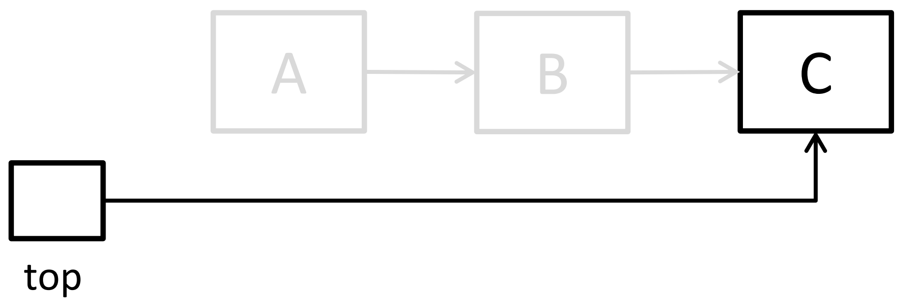
ABA Problem

1. Thread 1 *starts* popping A



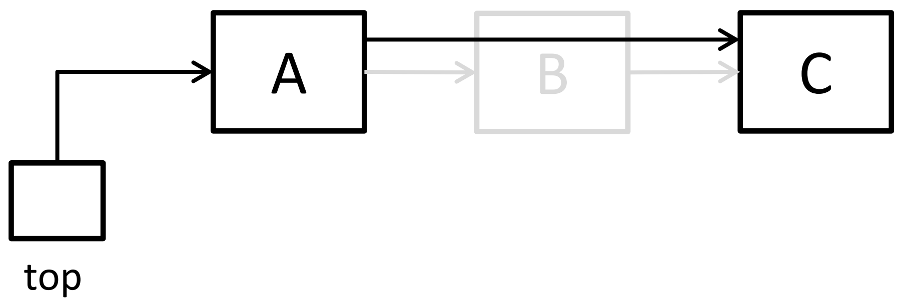
ABA Problem

1. Thread 1 *starts* popping A
2. Before thread 1 finishes, thread 2 pops A and B



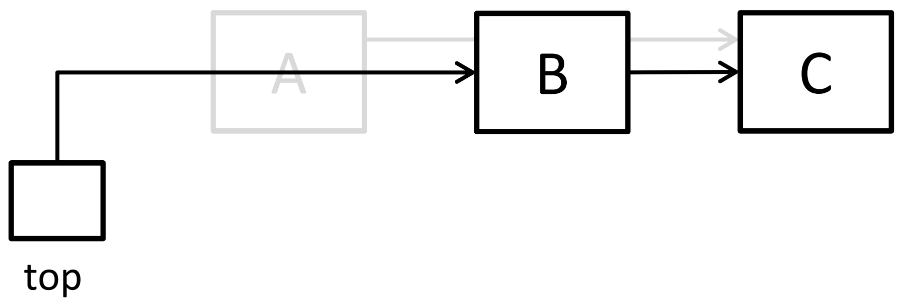
ABA Problem

1. Thread 1 *starts* popping A
2. Before thread 1 finishes, thread 2 pops A and B
3. Thread 2 pushes A back *// recovered from memory (same as thread 1 was popping)*



ABA Problem

1. Thread 1 *starts* popping A
2. Before thread 1 finishes, thread 2 pops A and B
3. Thread 2 pushes A back *// recovered from memory (same as thread 1 was popping)*
4. Thread 1 finishes popping A
    1. Incorrectly, as it thinks that A is the same as before thread 2 operations



ABA Problem

- It is a memory allocation issue which affects mainly languages without garbage collection (e.g., C and C++)

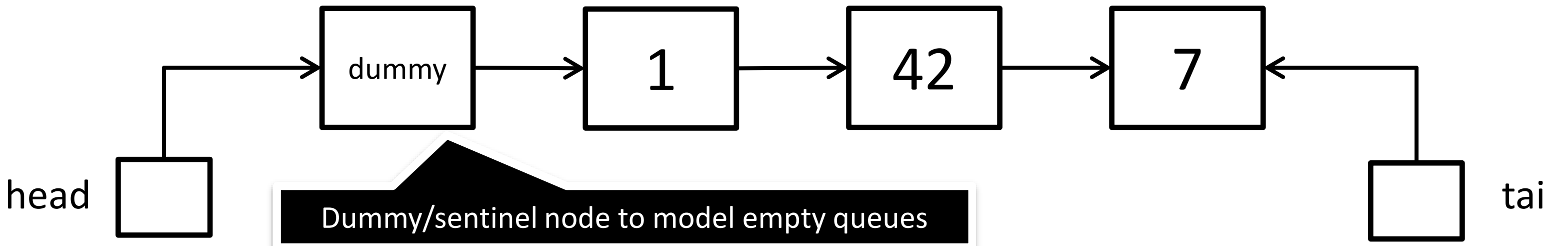
- Languages such as Java do not have this problem because garbage collection ensures that newly created objects are fresh
  - Step 3 in the previous slides would have created a new A object

Lock-free data structures: Queue

Queues

- A queue is a data structure following a FIFO *(first-in-first-out)* policy
  - enqueue() – adds an element to the tail of the queue
  - dequeue() – removes an element from the head of the queue

- It is typically implemented as a linked list



Lock-free queue

- Michael-Scott lock free queue, introduced in 1996 (see optional readings)
- Implemented in ConcurrentLinkedQueue in java.concurrent.* by Doug Lea et. al. (see here)
  - The version on the right is not the JDK implementation

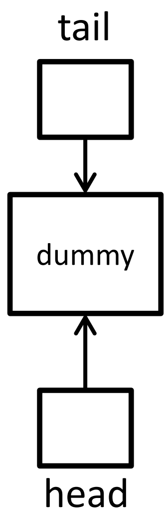
```
private static class Node<T> {
  final T item;
  final AtomicReference<Node<T>> next;

  public Node(T item, Node<T> next) {
    this.item = item;
    this.next = new AtomicReference<Node<T>>(next);
  }
}
```

```
class MSQueue<T> implements UnboundedQueue<T> {
    private final AtomicReference<Node<T>> head, tail;

    public MSQueue() {
        Node<T> dummy = new Node<T>(null, null);
        head = new AtomicReference<Node<T>>(dummy);
        tail = new AtomicReference<Node<T>>(dummy);
    }

    public void enqueue(T item) {
        Node<T> node = new Node<T>(item, null);
        while (true) {
            Node<T> last = tail.get(), next = last.next.get();
            if (last == tail.get()) {
                if (next == null)  {
                    // In quiescent state, try inserting new node
                    if (last.next.compareAndSet(next, node)) {
                        // Insertion succeeded, try advancing tail
                        tail.compareAndSet(last, node);
                        return;
                    }
                } else
                    // Queue in intermediate state, advance tail
                    tail.compareAndSet(last, next);
            }
        }
    }

    public T dequeue() {
        while (true) {
            Node<T> first = head.get(), last = tail.get(), next = first.next.get();
            if (first == head.get()) {
                if (first == last) {
                    if (next == null)
                        return null;
                    else
                        tail.compareAndSet(last, next);
                } else {
                    T result = next.item;
                    if (head.compareAndSet(first, next))
                        return result;
                }
            }
        }
    }
...
```

See **`TestMSQueue.java`**

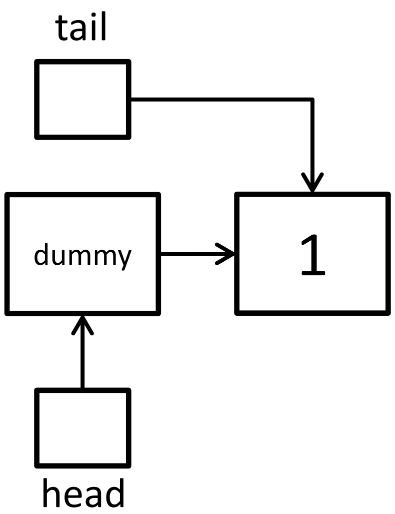
Lock-free queue | initialization


```
class MSQueue<T> implements UnboundedQueue<T> {
    private final AtomicReference<Node<T>> head, tail;

    public MSQueue() {
      Node<T> dummy = new Node<T>(null, null);
      head = new AtomicReference<Node<T>>(dummy);
      tail = new AtomicReference<Node<T>>(dummy);
    }
...
}
```



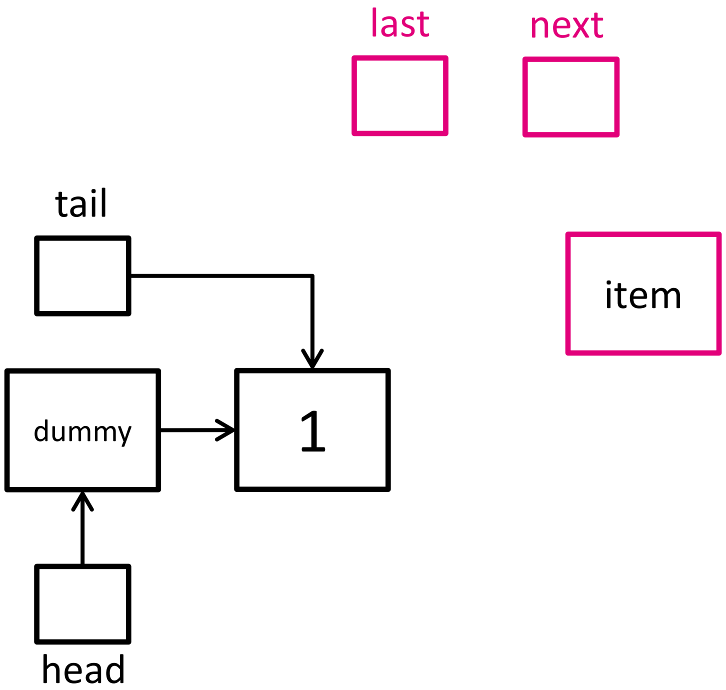
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



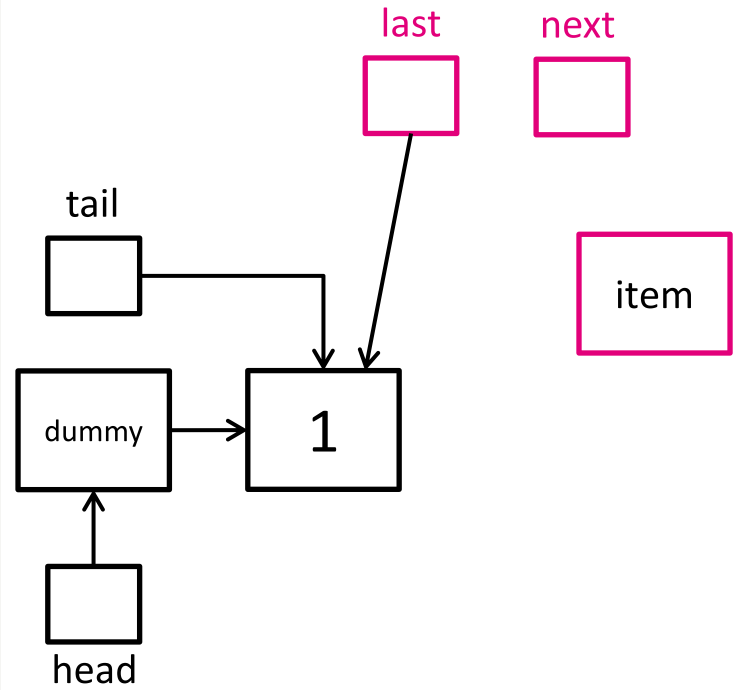
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



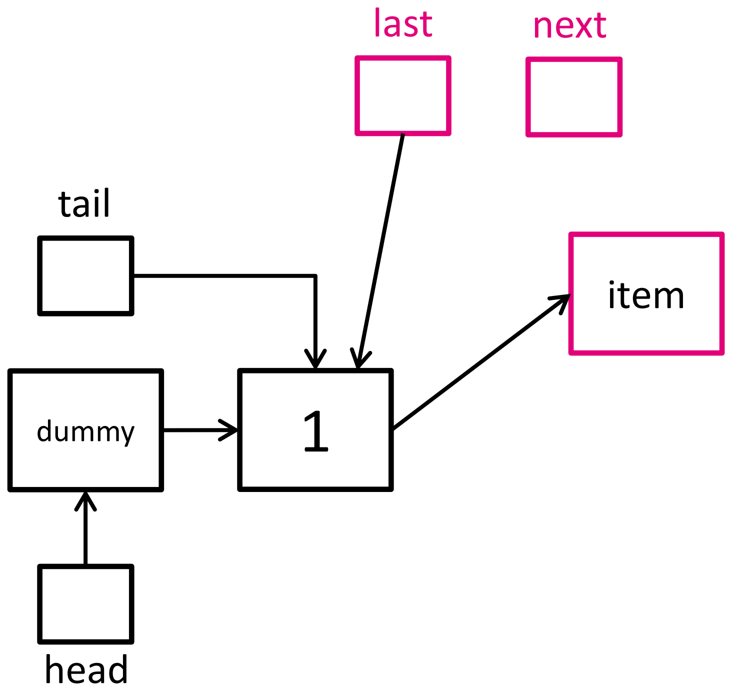
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)   {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



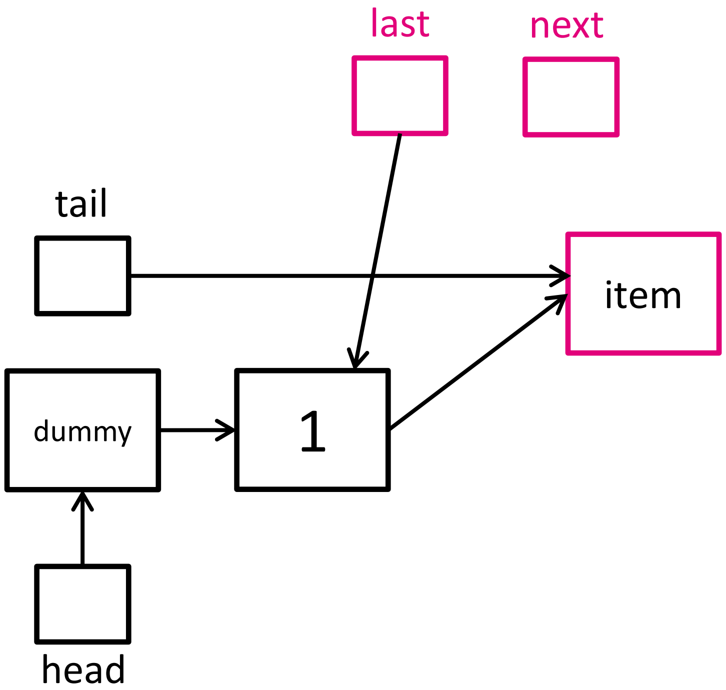
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                  // In quiescent state, try inserting new node
                  if (last.next.compareAndSet(next, node)) {
                    // Insertion succeeded, try advancing tail
                    tail.compareAndSet(last, node);
                    return;
                  }
             } else
                  // Queue in intermediate state, advance tail
                  tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



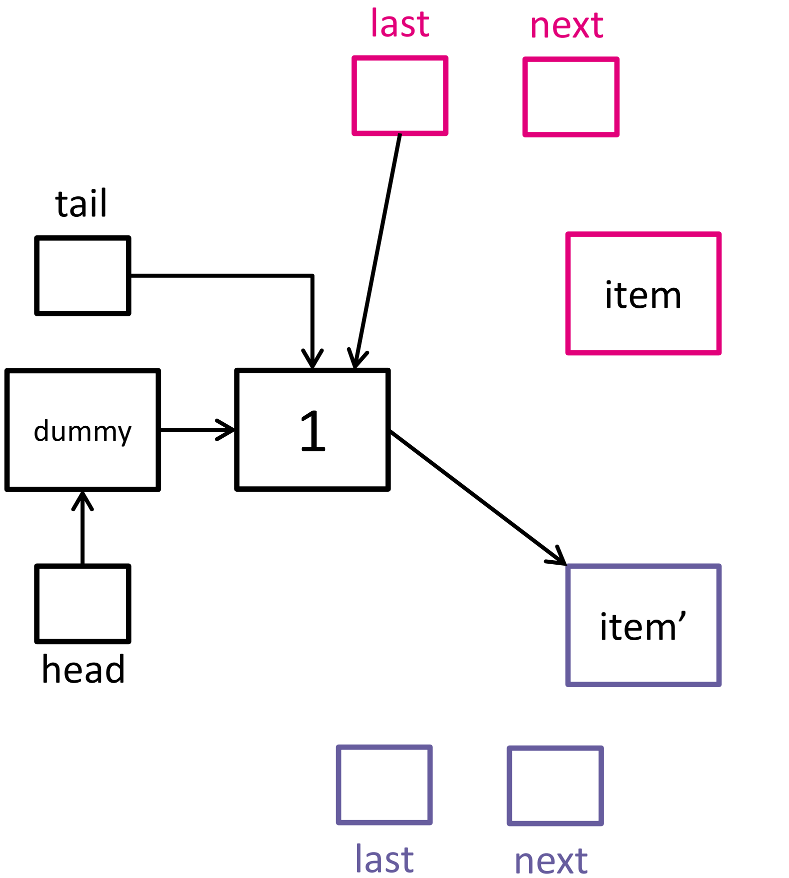
Lock-free queue | enqueue

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



Lock-free queue | enqueue

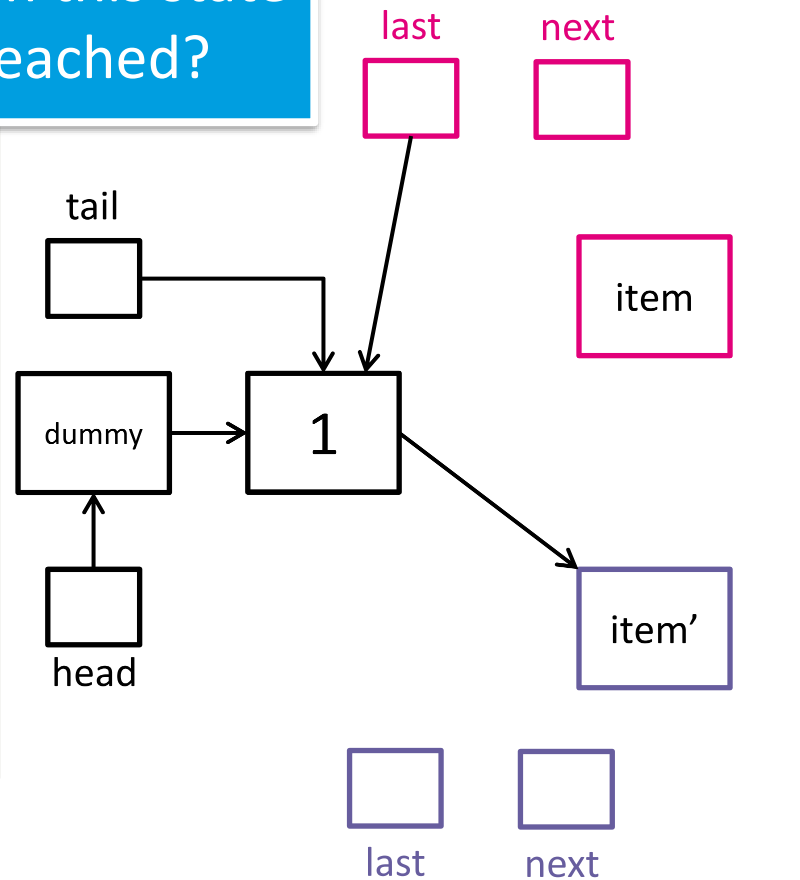

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



Lock-free queue | enqueue

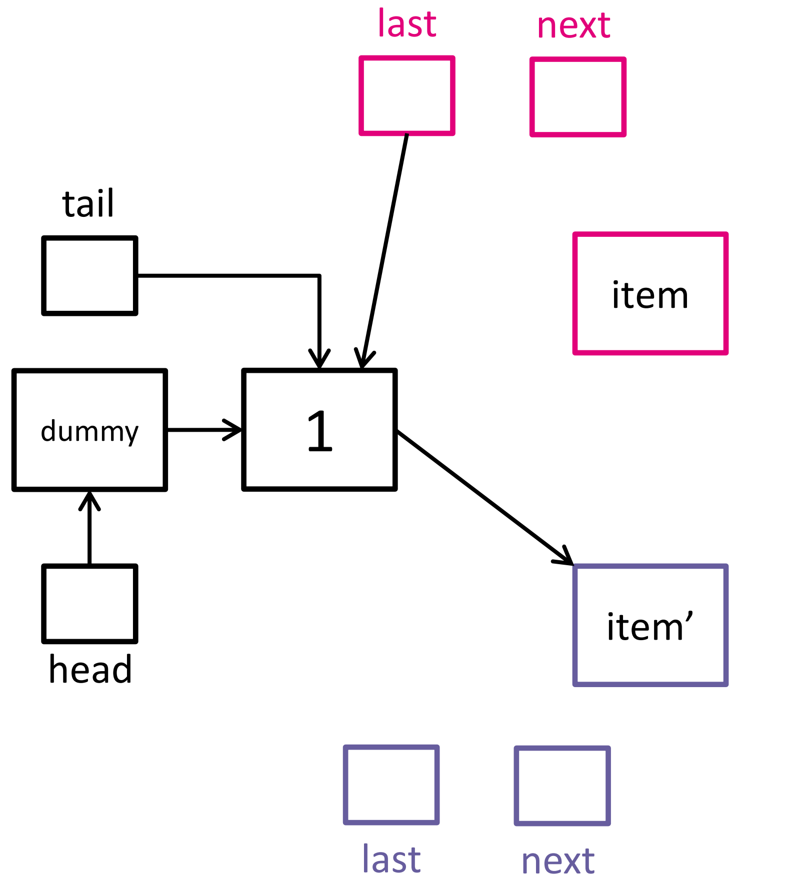

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```

How can this state be reached?



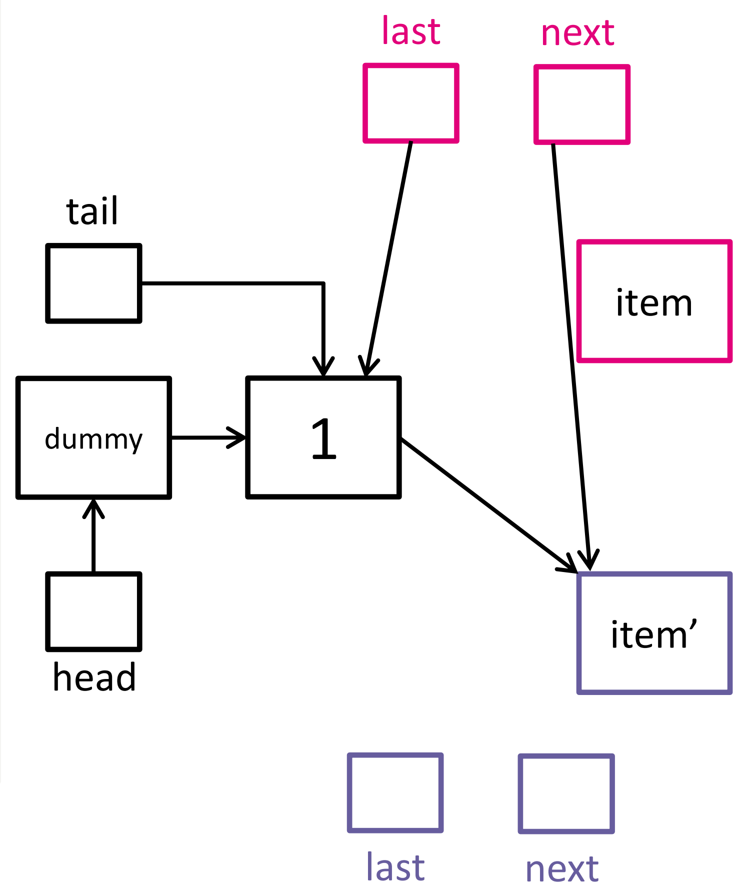
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



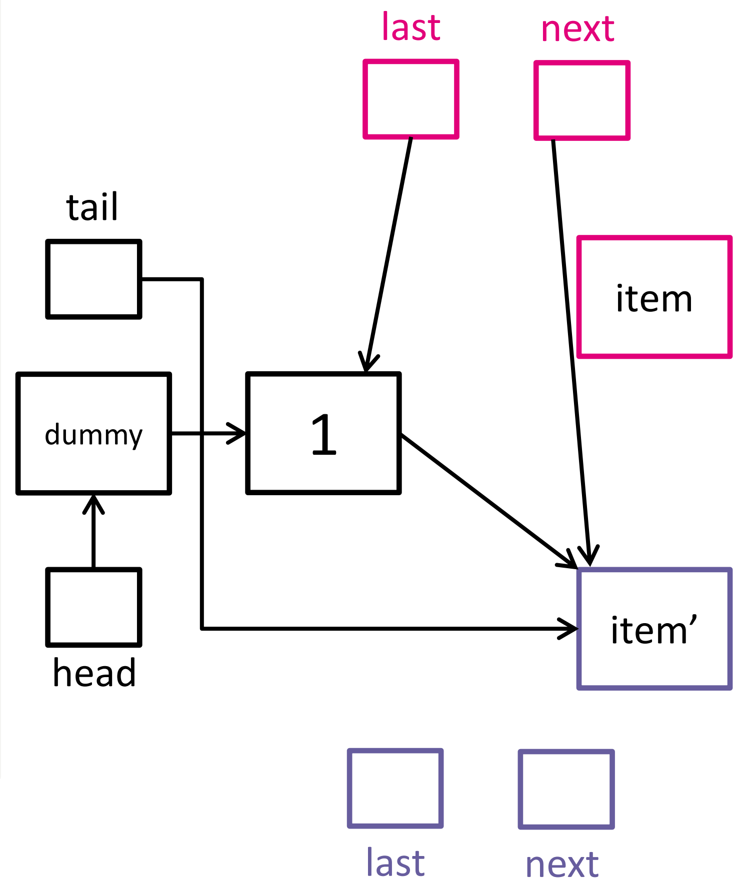
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)   {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



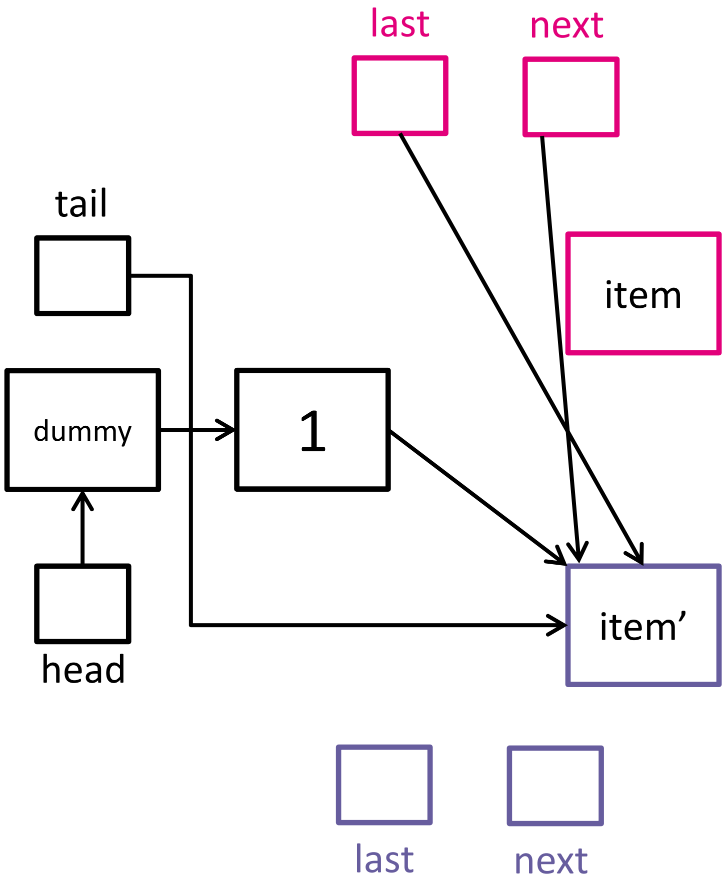
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



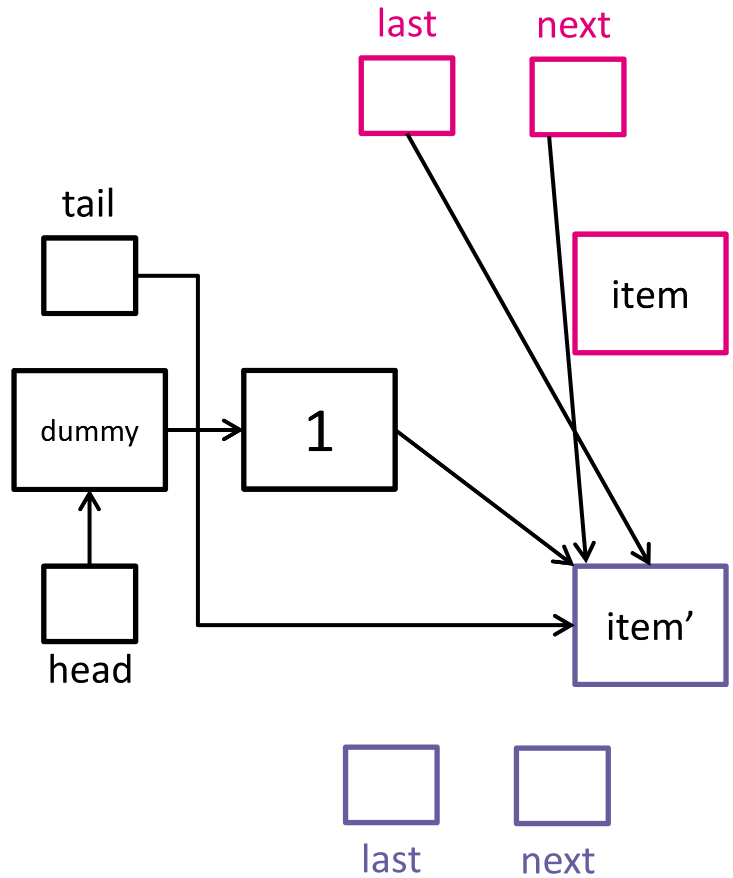
Lock-free queue | enqueue


```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
           Node<T> last = tail.get();
           Node<T> next = last.next.get();
           if (last == tail.get()) {
             if (next == null)  {
                 // In quiescent state, try inserting new node
                 if (last.next.compareAndSet(next, node)) {
                   // Insertion succeeded, try advancing tail
                   tail.compareAndSet(last, node);
                   return;
                 }
             } else
                 // Queue in intermediate state, advance tail
                 tail.compareAndSet(last, next);
           }
      }
    }
…
}
```



Lock-free queue | enqueue

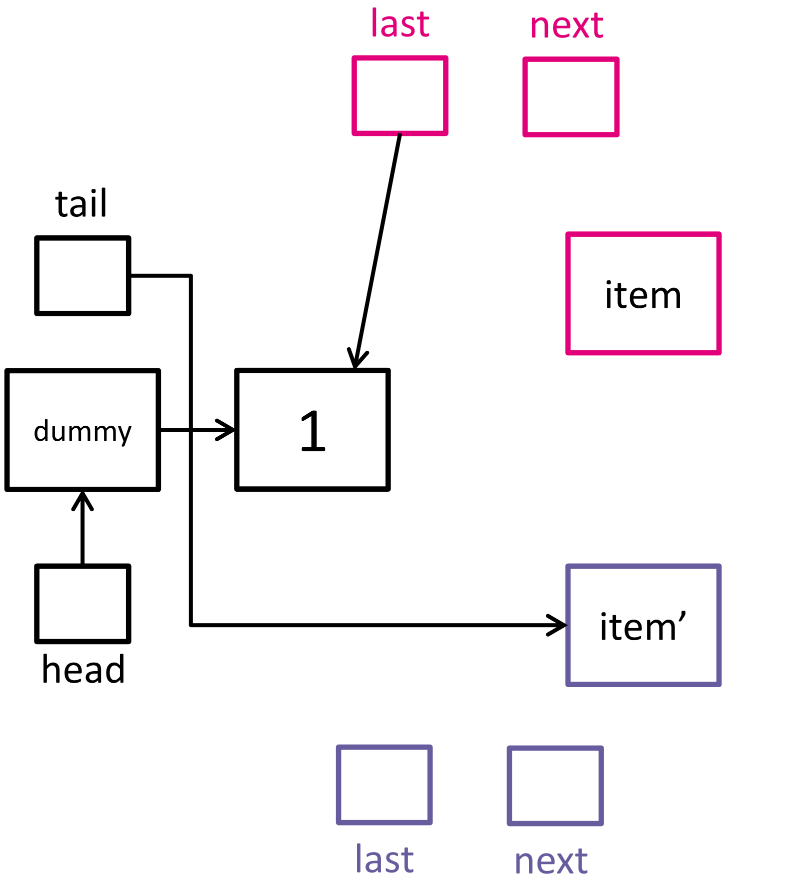

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



In case another thread is enqueuing, and didn't update the tail, the current thread helps by advancing the tail

Lock-free queue | enqueue

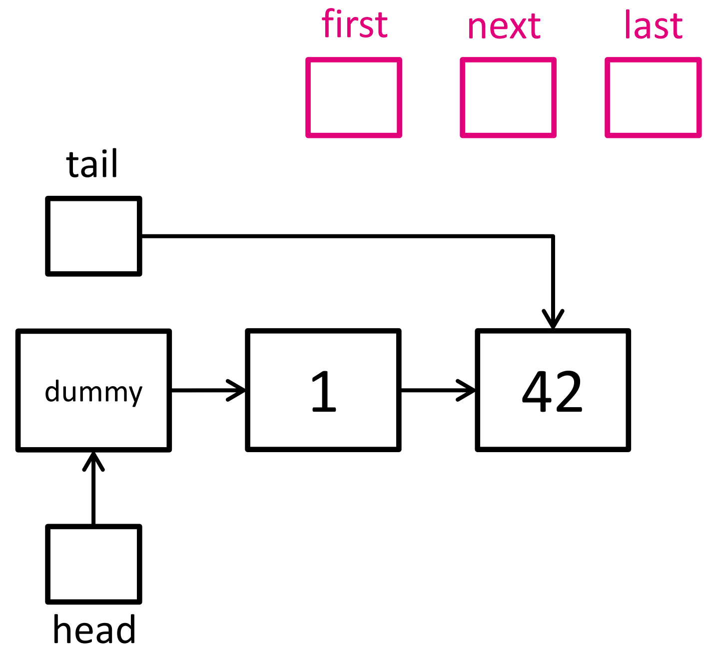

```
class MSQueue<T> implements UnboundedQueue<T> {
…
 public void enqueue(T item) {
      Node<T> node = new Node<T>(item, null);
      while (true) {
          Node<T> last = tail.get();
          Node<T> next = last.next.get();
          if (last == tail.get()) {
            if (next == null)  {
                // In quiescent state, try inserting new node
                if (last.next.compareAndSet(next, node)) {
                  // Insertion succeeded, try advancing tail
                  tail.compareAndSet(last, node);
                  return;
                }
            } else
                // Queue in intermediate state, advance tail
                tail.compareAndSet(last, next);
          }
      }
    }
…
}
```



If the tail has been changed, then the thread restarts right away

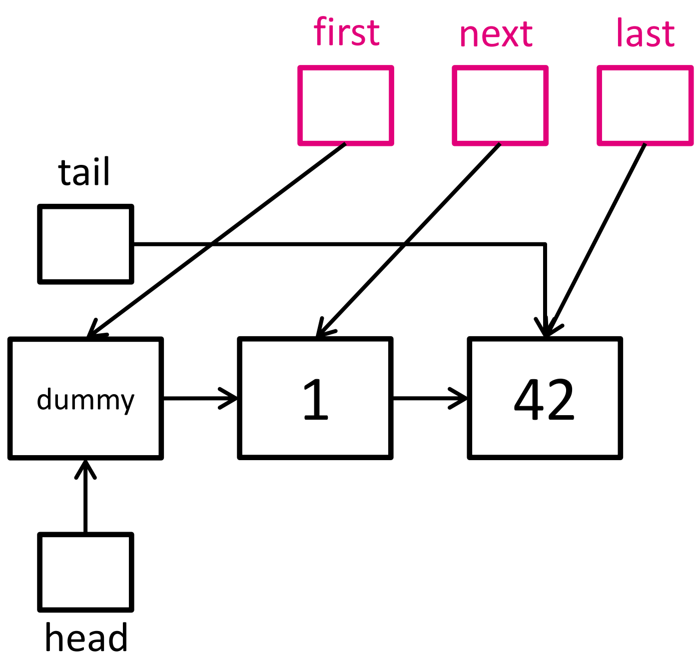
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



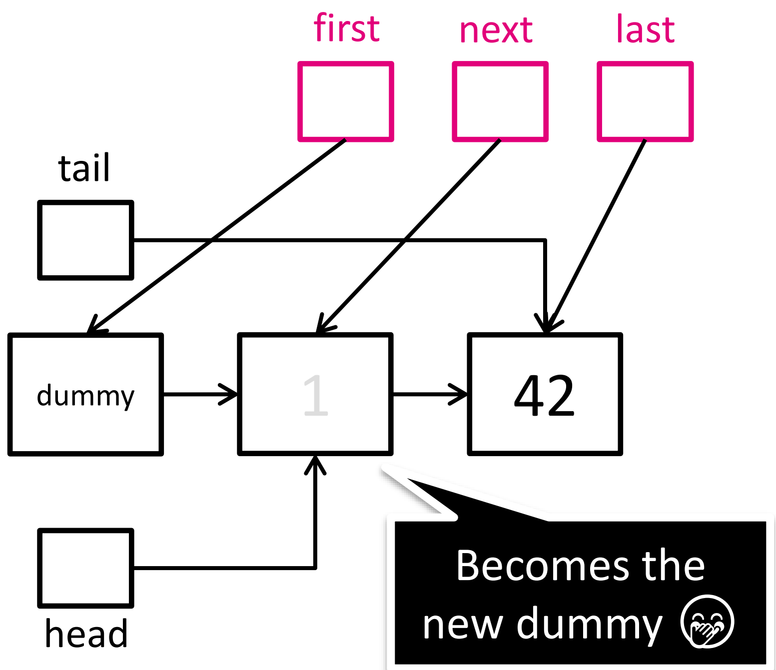
Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                 return null;
               else
                 tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                 return result;
           }
          }
      }
    }
…
}
```



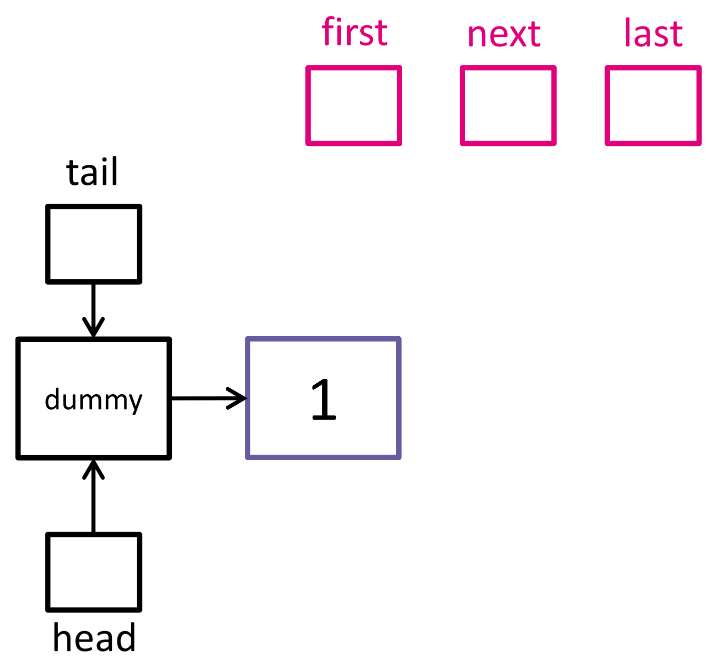
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
          Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



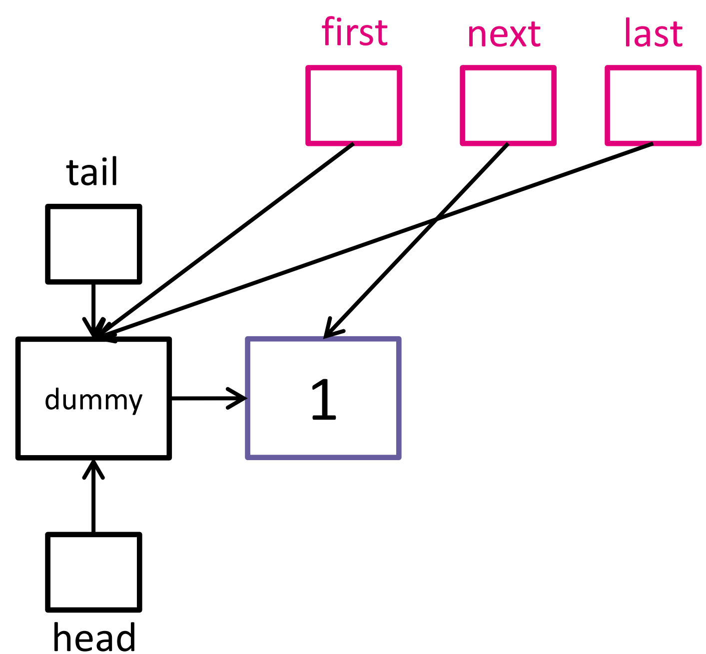
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
                if (next == null)
                  return null;
                else
                  tail.compareAndSet(last, next);
            } else {
                T result = next.item;
                if (head.compareAndSet(first, next))
                  return result;
            }
          }
      }
    }
…
}
```



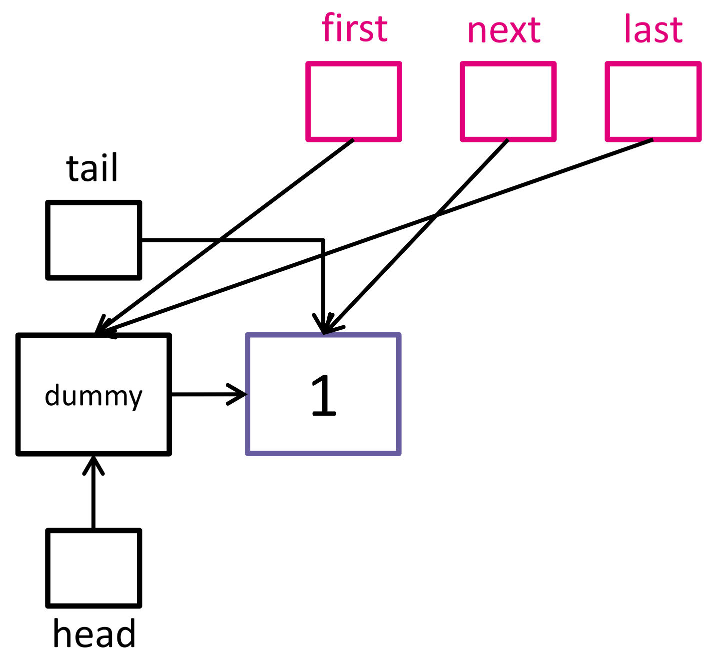
Lock-free queue | dequeue


```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
          Node<T> next = first.next.get();
          if (first == head.get()) {
            if (first == last) {
              if (next == null)
                return null;
              else
                tail.compareAndSet(last, next);
            } else {
              T result = next.item;
              if (head.compareAndSet(first, next))
                return result;
            }
          }
      }
    }
…
}
```



Lock-free queue | dequeue

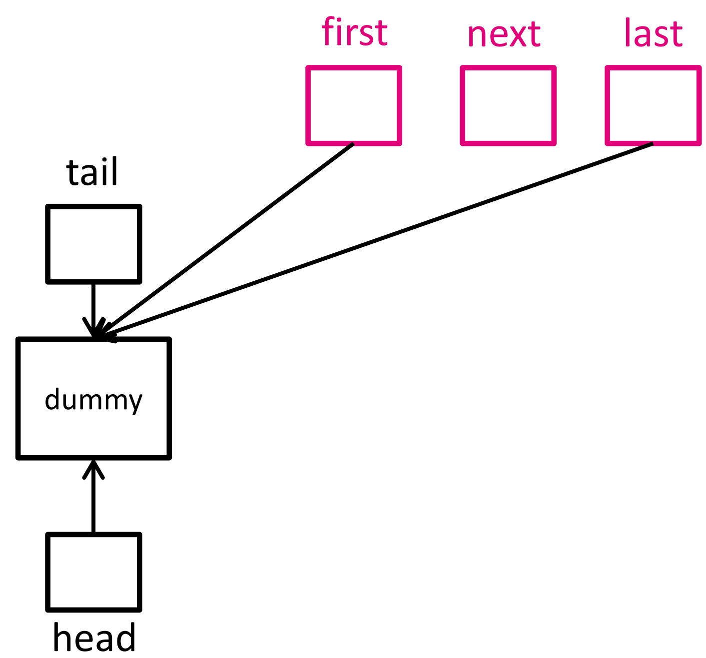
```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                 return null;
               else
                 tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                 return result;
           }
          }
       }
    }
…
}
```



If the next field of the head is not null (because another thread push in the meantime), then the calling thread helps advancing the tail and tries to pop again.

Lock-free queue | dequeue

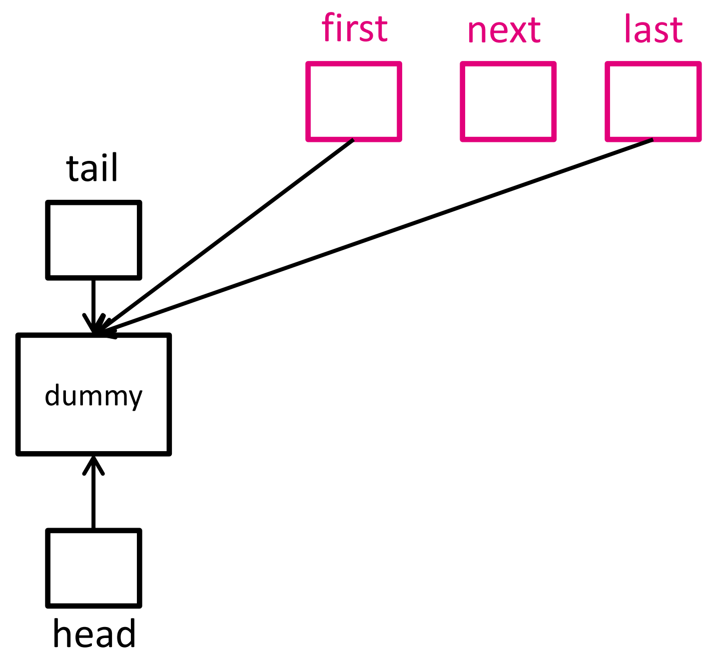

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                 return null;
               else
                 tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                 return result;
           }
          }
      }
    }
…
}
```



Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
      while (true) {
          Node<T> first = head.get();
          Node<T> last = tail.get();
         Node<T> next = first.next.get();
          if (first == head.get()) {
           if (first == last) {
               if (next == null)
                return null;
               else
                tail.compareAndSet(last, next);
           } else {
               T result = next.item;
               if (head.compareAndSet(first, next))
                return result;
           }
          }
      }
…
}
```

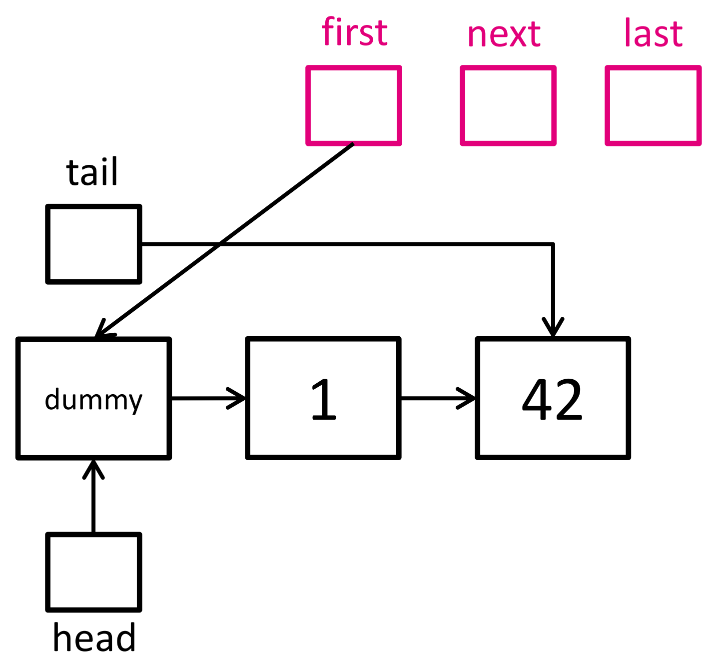
If the next field of the head is pointing to null, then we return null. Why is this correct?



Lock-free queue | dequeue

```
class MSQueue<T> implements UnboundedQueue<T> {
…

    public T dequeue() {
     while (true) {
         Node<T> first = head.get();
         Node<T> last = tail.get();
        Node<T> next = first.next.get();
         if (first == head.get()) {
          if (first == last) {
              if (next == null)
                return null;
              else
                tail.compareAndSet(last, next);
          } else {
              T result = next.item;
              if (head.compareAndSet(first, next))
                return result;
          }
…
}
```

How can a state where first != head.get() be reached?



What about correctness?

- We have seen several implementations of lock-free data structures today
- However, we have not seen any techniques to reason about their correctness
  - In two weeks -> Linearizability

No class next week

Enjoy a well-deserved Autumn break!

Agenda

- Compare-And-Swap (CAS)
  - Lock-free atomic integer
  - Lock-free number range
  - Atomic libraries
  - CAS based lock implementation
- Lock-free stack
- ABA problem
- Lock-free queue