

CMSC 433 – Programming Language Technologies and Paradigms

Atomic Variables
Nonblocking Synchronization

java.util.concurrent Performance

- Many java.util.concurrent classes perform better than synchronized alternatives. Why?
 - Atomic variables & nonblocking algorithms
- Atomic variables – lock-free atomic operations
- Nonblocking algorithms are concurrent algorithms that derive their thread safety from low-level atomic hardware primitives (not locks)

Disadvantages of Locking

- When a thread fails to acquire lock it can be suspended
 - Context switching & resumption can be expensive
- When waiting for a lock, thread can't do anything
- If a Thread holding a lock is delayed, no thread that needs that lock can progress
 - Can result in priority inversion: low priority thread has lock needed by a high priority thread
- Caveat: contention, rather than locking, is the real issue. YMMV

Hardware Support

- Locking is pessimistic
 - If contention is infrequent, most locking was unneeded
- In earlier lecture we discussed optimistic trying
 - Proceed with the update
 - Check for collision
 - If update fails, retry
- Processor can use atomic operations to support optimistic trying

Atomic Variables

- Holder classes for scalars, references and fields
- Supports atomic operations
 - Compare-and-set (CAS) (discussed later)
 - Get, set and arithmetic (where applicable)
- Some main classes: {int, long, ref} X {value, field, array}
 - E.g. AtomicInteger useful for counters, sequences, statistics
- Essential for writing efficient code on MPs
 - Nonblocking data structures & optimistic algorithms
 - Reduce overhead/contention updating “hot” fields
- JVM uses best construct available on platform
 - CAS, load-linked/store-conditional, locks

AtomicInteger

- Some methods include:
- `int get()` – returns the current value
- `void set(int newValue)` – sets to the given value
- `int getAndSet(int newValue)` - Atomically sets to the given value and returns the old value
- `boolean compareAndSet(int expected, int update)`
-Atomically sets the value to update if current value == expected

Summary of Atomic Variables

- Generalization of volatile variables
- Allows atomic read-modify-write operations without intrinsic locking
- Scope of contention limited to a single variable
- Faster than locking -- no scheduling impact
- Like volatiles, can't synchronize two atomic vars
- In general, doesn't support atomic check-then-act sequences

A Synchronized Counter

```
public final class SynchronizedCounter {  
    private int c= 0;  
    public synchronized int increment() {  
        return ++c;  
    }  
    public synchronized int decrement() {  
        return --c;  
    }  
    public synchronized int value() {  
        return c;  
    }  
}
```

AtomicCounter

```
class AtomicCounter implements Counter {  
    private AtomicInteger c = new AtomicInteger(0);  
    public void increment() {  
        c.incrementAndGet();  
    }  
    public void decrement() {  
        c.decrementAndGet();  
    }  
    public int value() {  
        return c.get();  
    }  
}
```

Atomic Variables

- See: CounterTest.java

Compare and Swap (CAS)

- CAS has 3 operands
 - Memory location V , expected value A , new value B
- Atomically updates V to new value B , only if current value is the expected value A
- Multiple threads try to update V , only one succeeds
 - But the losers don't get punished with suspension
 - They can just try again

Simulated CAS

```
public class SimulatedCAS { // not implemented this way!
    private int value;

    public synchronized int get() {return currValue;}

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

    public synchronized boolean compareAndSet(int expectedValue, int newValue) {
        return (expectedValue == compareAndSwap(expectedValue, newValue));
    }
}
```

A Nonblocking Counter

```
// demonstrates the use of CAS
public class NonblockingCounter {
    private AtomicInteger value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        } while (!value.compareAndSet(v, v + 1));
        return v + 1;
    }
}
```

Updating Complex Objects

- Suppose we have an invariant over two variables
 - Can't maintain this invariant using volatile fields
- General strategy
 - Turn compound update into single update

CasNumberRange

```
// INVARIANT: lower <= upper
// How do you make this thread-safe?
private static class IntPair {
    final int lower, upper;
    public IntPair(int lower, int upper) {....}
    public void setLower(int i) {....}
    public void setUpper(int i) {....}
}
```

CasNumberRange

```
public class CasNumberRange {  
    // IntPair represents a pair of Integers  
    private final AtomicReference<IntPair> values =  
        new AtomicReference<IntPair>(new IntPair(0, 0));  
  
    public void setLower(int i) {  
        while (true) {  
            IntPair oldv = values.get(); // gets the current value atomically  
            if (i > oldv.upper) throw new IllegalArgumentException();  
            IntPair newv = new IntPair(i, oldv.upper);  
            if (values.compareAndSet(oldv, newv)) return;  
        }  
    }  
    // setUpper() similar to setLower()  
}
```

Performance Comparison

- Two implementations of a psuedo-random number generator (PRNG)
 - One uses locks: `ReentrantLockPseudoRandom.java`
 - One is nonblocking: `AtomicPseudoRandom.java`
- PRNG issues
 - Next value based on last value, so you need to remember last value
- How do lock-based and non-lock-based implementations compare?

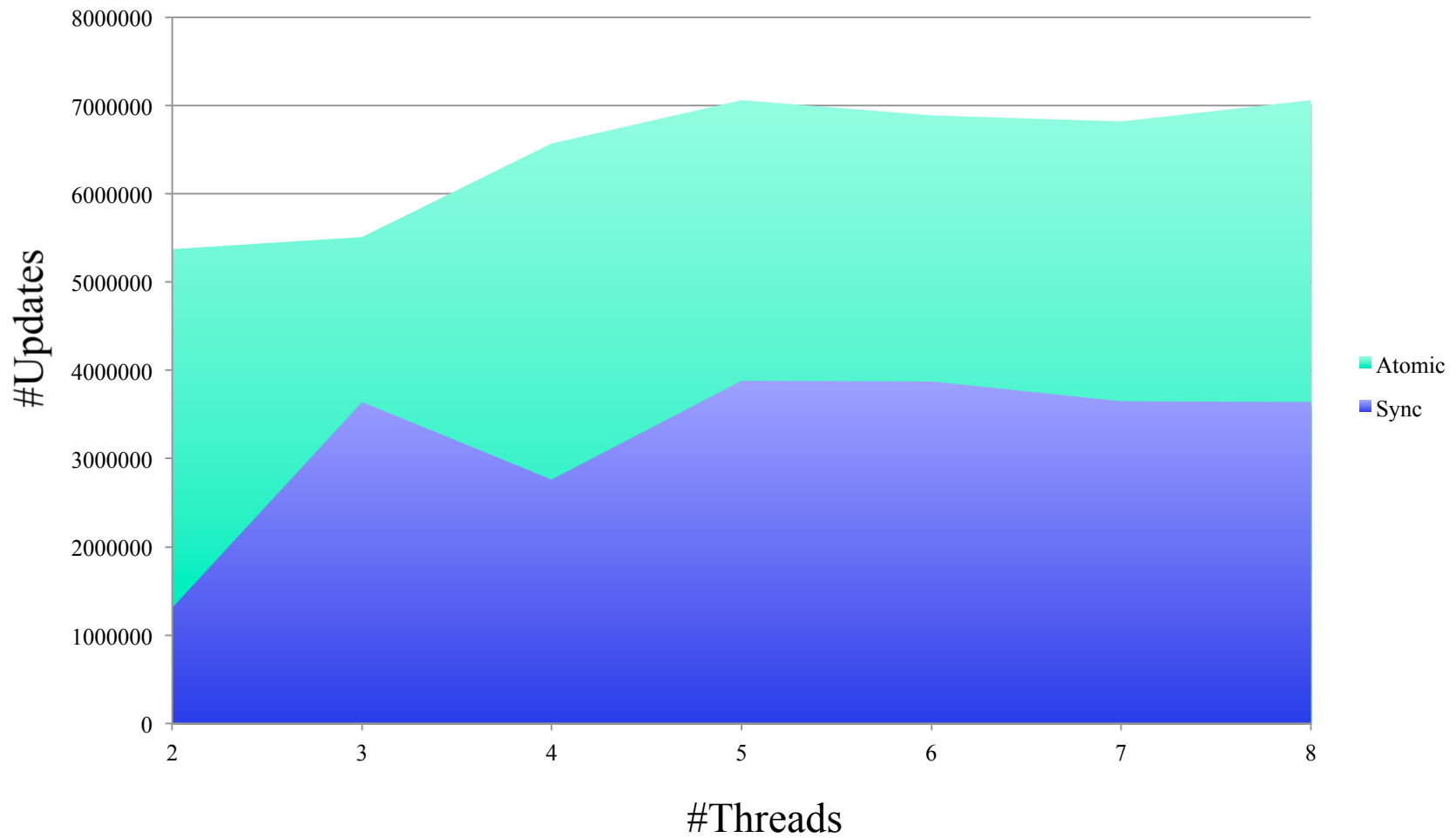
ReentrantLockPseudoRandom

```
public class ReentrantLockPseudoRandom extends PseudoRandom {  
    private final Lock lock = new ReentrantLock(false);  
    private int seed;  
  
    ReentrantLockPseudoRandom(int seed) {this.seed = seed;}  
  
    public int nextInt(int n) {  
        lock.lock();  
        try {  
            int s = seed; seed = calculateNext(s); int remainder = s % n;  
            return remainder > 0 ? remainder : remainder + n;  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

AtomicPseudoRandom

```
public class AtomicPseudoRandom extends PseudoRandom {  
    private AtomicInteger seed;  
  
    AtomicPseudoRandom(int seed) {this.seed = new AtomicInteger(seed);}   
  
    public int nextInt(int n) {  
        while (true) {  
            int s = seed.get();  
            int nextSeed = calculateNext(s);  
            if (seed.compareAndSet(s, nextSeed)) {  
                int remainder = s % n;  
                return remainder > 0 ? remainder : remainder + n;  
            }  
        }  
    }  
}
```

Comparing Performance



MacPro, OS X Snow Leopard, 8 cores, 8 GB of RAM

Nonblocking Algorithms

- No locks
- Stopping one thread will not prevent global progress
 - Immune to deadlock
 - Starvation is possible
- Writing correct nonblocking algorithms is very hard!

Nonblocking Algorithm Flavors

- Wait-Free
 - All threads complete in finite count of steps
 - Low priority threads cannot block high priority threads
- Lock-Free
 - Every successful step makes global progress
 - Individual threads may starve; priority inversion possible
 - No live-lock
- Obstruction-Free
 - A single thread in isolation completes in finite count of steps
 - Threads may block each other; live-lock possible
 - Example: optimistic retry

Nonblocking Stack

```
public class ConcurrentStack <E> {  
    private static class Node <E> {  
        public final E item;  public Node<E> next;  
        public Node(E item) {  
            this.item = item;  
        }  
    }  
    AtomicReference<Node<E>> top =  
        new AtomicReference<Node<E>>();  
}
```

Nonblocking Stack

...

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

Nonblocking Stack

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

push() & pop()

```
public void push(E item) {  
    Node<E> nh= new Node<E>(item);  
    Node<E> oh;  
    do {  
        oh = top.get();  
        nh.next = oh;  
    } while (!top.compareAndSet(oh, nh));  
}
```

```
public E pop() {  
    Node<E> oh; Node<E> nh;  
    do {  
        oh = top.get();  
        if (oh == null)  
            return null;  
        nh = oh.next;  
    } while (!top.compareAndSet(oh, nh));  
    return oh.item;  
}
```

Nonblocking Stack

- See: `ConcurrentStack.java` & `SynchStack.java`

A Nonblocking Queue

- Remember general strategy
 - Limit change to one variable
- Harder for a Queue because we need to update both head and tail
- See: `SynchQueue.java` & `ConcurrentQueue.java`

Overview of Michael & Scott Approach

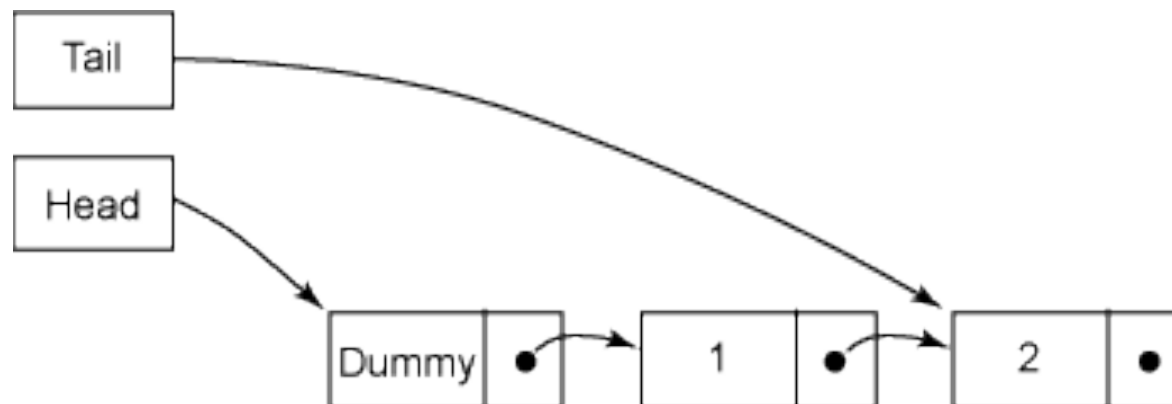
- Make sure queue is always in consistent state
- Threads must be able to detect whether another operation is already in progress
 - Thread B can wait for thread A to finish before starting
- Prevents corruption, but late thread can fail if early thread fails

Overview of Michael & Scott Approach

- If thread B arrives while operation in progress for thread A, let B finish update for A
 - Then B can progress without waiting for A
 - If A finds some of its work done, it doesn't repeat. It just skips doing it itself

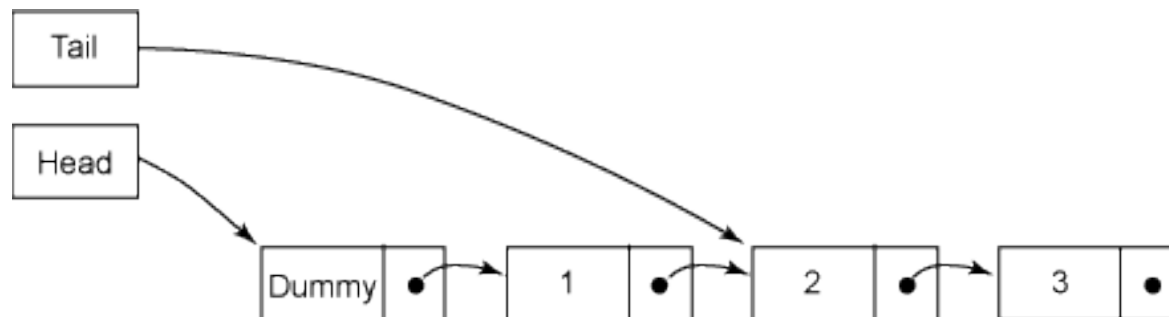
Michael & Scott Nonblocking Queue

- Queue with two elements in quiescent state



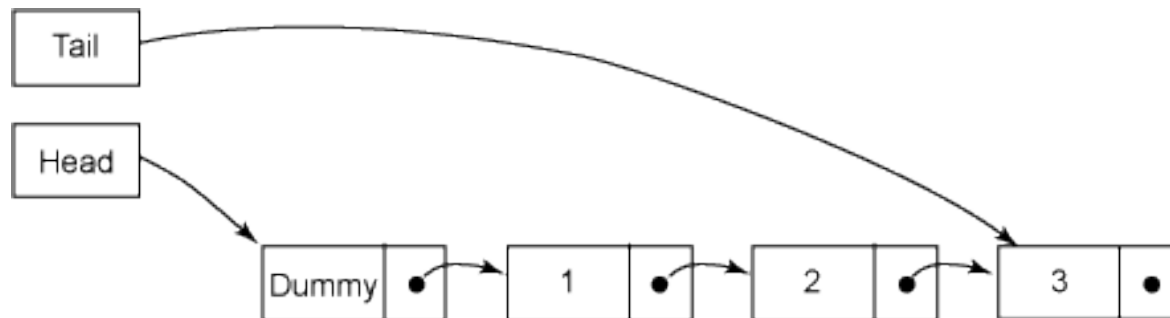
Michael & Scott Nonblocking Queue

- Queue in intermediate state during insertion
 - After the new element is added but before the tail pointer is updated



Michael & Scott Nonblocking Queue

- Queue in quiescent state again after the tail pointer is updated



Michael & Scott Nonblocking Queue

- Observation:
 - If `tail.next` is non-null, then a `put()` operation is in progress
- If a thread finds an `put()` operation in progress, it will try to advance `tail` to return queue to stable state
 - Then it will reload `tail` and repeat process

ConcurrentQueue

```
public class ConcurrentQueue <E> {  
    private static class Node <E> {  
        final E item;  
        final AtomicReference<Node<E>> next;  
        public Node(E item, Node<E> next) {  
            this.item = item;  
            this.next = new AtomicReference<Node<E>>(next);  
        }  
    }  
    private final Node<E> dummy = new Node<E>(null, null);  
    private final AtomicReference<Node<E>> head  
        = new AtomicReference<Node<E>>(dummy);  
    private final AtomicReference<Node<E>> tail  
        = new AtomicReference<Node<E>>(dummy);  
}
```

ConcurrentQueue

```
public boolean put(E item) {
    Node<E> newNode = new Node<E>(item, null);
    while (true) {
        Node<E> curTail = tail.get();
        Node<E> tailNext = curTail.next.get();
        if (curTail == tail.get()) { // did tail change?
            if (tailNext != null) { // Queue in intermediate state, advance tail
                tail.compareAndSet(curTail, tailNext);
            } else { // In quiescent state, try inserting new node
                if (curTail.next.compareAndSet(null, newNode)) {
                    // Insertion succeeded, try advancing tail
                    tail.compareAndSet(curTail, newNode); // will fail if tail already moved
                    return true;
                }
            }
        }
    }
}
```

ConcurrentQueue

```
public E take() {  
    for (;;) {  
        Node<E> oldHead = head.get();           // get current head  
        Node<E> oldTail = tail.get();           // get current tail  
        Node<E> oldHeadNext = oldHead.next.get(); // get current head.next  
        if (oldHead == head.get()) {            // has another take happened?  
            if (oldHead == oldTail) {           // Queue empty or tail being updated  
                if (oldHeadNext == null)        // Is queue empty? If yes,  
                    return null;  
                tail.compareAndSet(oldTail, oldHeadNext); // tail needs update. Try to advance it  
            } else {                             // No need to deal with tail  
                if (head.compareAndSet(oldHead, oldHeadNext))  
                    return oldHeadNext.item;  
            }  
        }  
    }  
}
```

Execution Traces

- 2 threads attempt to put()

Trace 1

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }
18 }
19 }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



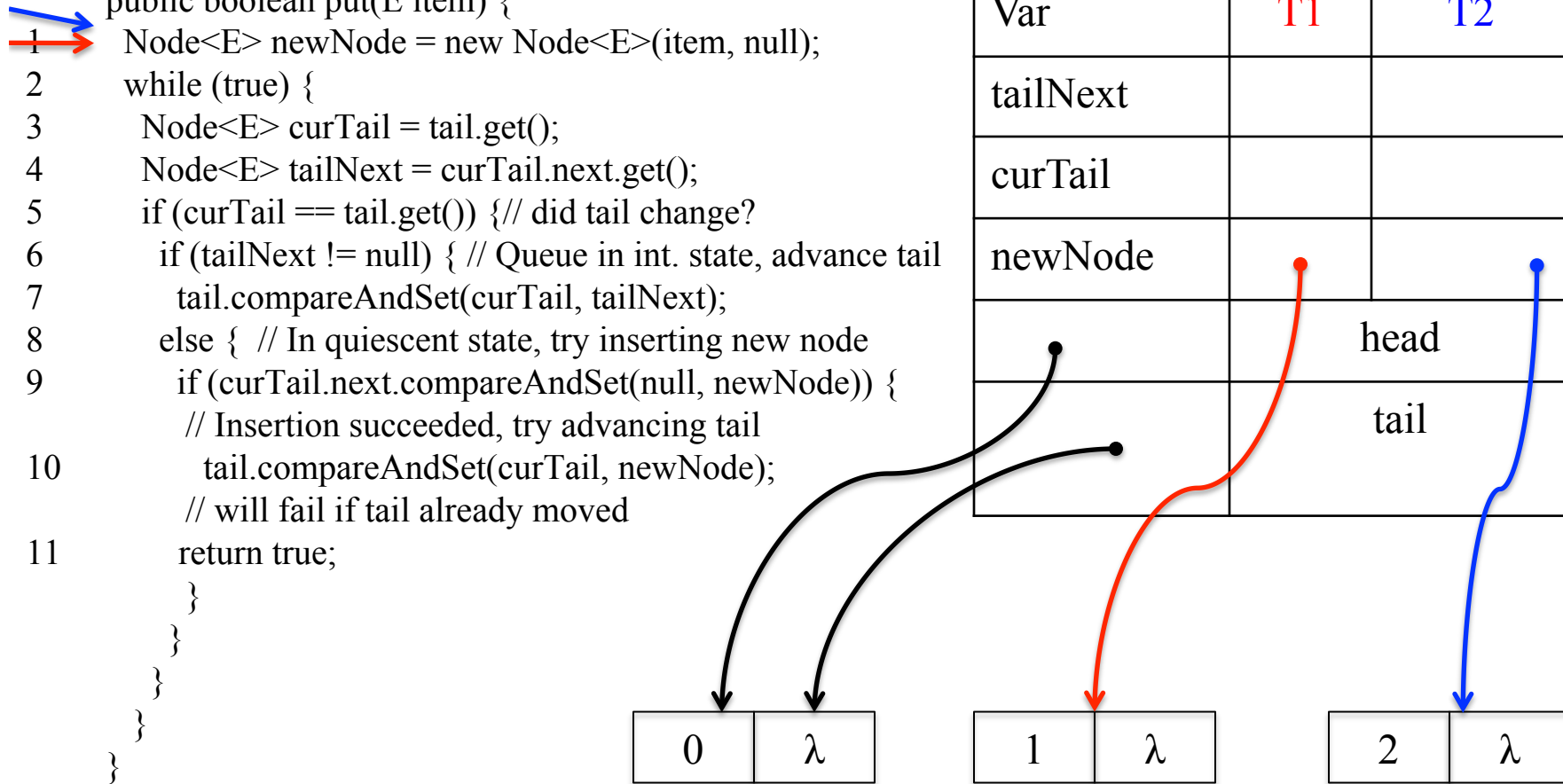
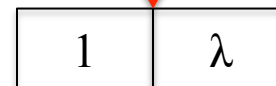
Trace 1

```

1 public boolean put(E item) {
2   Node<E> newNode = new Node<E>(item, null);
3   while (true) {
4     Node<E> curTail = tail.get();
5     Node<E> tailNext = curTail.next.get();
6     if (curTail == tail.get()) { // did tail change?
7       if (tailNext != null) { // Queue in int. state, advance tail
8         tail.compareAndSet(curTail, tailNext);
9       }
10      else { // In quiescent state, try inserting new node
11        if (curTail.next.compareAndSet(null, newNode)) {
12          // Insertion succeeded, try advancing tail
13          tail.compareAndSet(curTail, newNode);
14          // will fail if tail already moved
15          return true;
16        }
17      }
18    }
19  }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



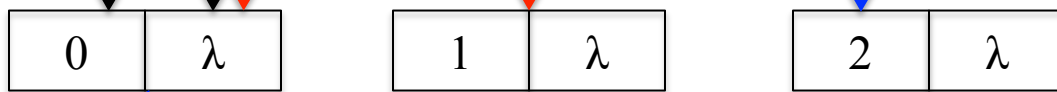
Trace 1

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3  → Node<E> curTail = tail.get();
4  → Node<E> tailNext = curTail.next.get();
5  if (curTail == tail.get()) { // did tail change?
6  if (tailNext != null) { // Queue in int. state, advance tail
7  tail.compareAndSet(curTail, tailNext);
8  else { // In quiescent state, try inserting new node
9  if (curTail.next.compareAndSet(null, newNode)) {
10 // Insertion succeeded, try advancing tail
10 tail.compareAndSet(curTail, newNode);
11 // will fail if tail already moved
    return true;
    }
    }
    }
    }
}

```

Var	T1	T2
tailNext		
curTail		
newNode		
head		
tail		



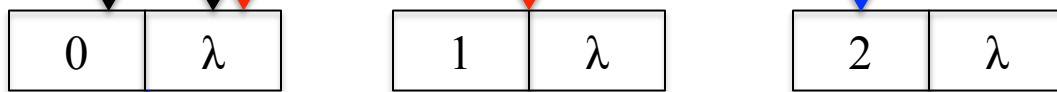
Trace 1

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      } else { // In quiescent state, try inserting new node
9        if (curTail.next.compareAndSet(null, newNode)) {
10         // Insertion succeeded, try advancing tail
11         tail.compareAndSet(curTail, newNode);
12         // will fail if tail already moved
13         return true;
14       }
15     }
16   }
17 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



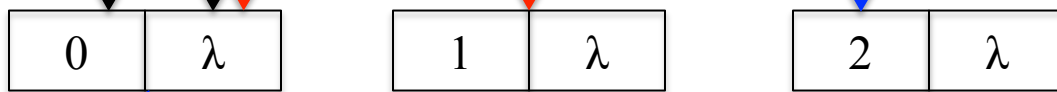
Trace 1

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5 →  if (curTail == tail.get()) { // did tail change?
6 →      if (tailNext != null) { // Queue in int. state, advance tail
7          tail.compareAndSet(curTail, tailNext);
8      } else { // In quiescent state, try inserting new node
9          if (curTail.next.compareAndSet(null, newNode)) {
10             // Insertion succeeded, try advancing tail
11             tail.compareAndSet(curTail, newNode);
12             // will fail if tail already moved
13             return true;
14         }
15     }
16 }
17 }
18 }
19 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



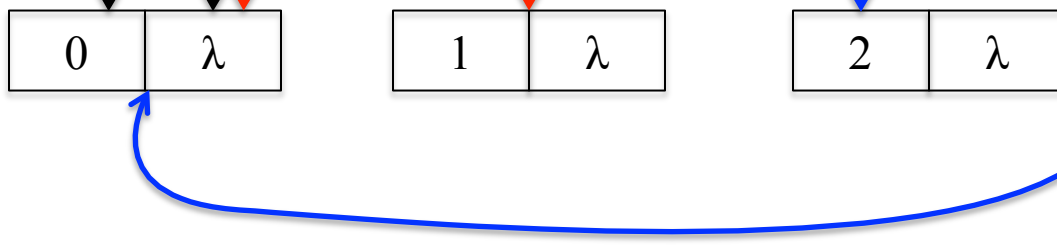
Trace 1

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6 →  if (tailNext != null) { // Queue in int. state, advance tail
7 →      tail.compareAndSet(curTail, tailNext);
8      }
9      else { // In quiescent state, try inserting new node
10     if (curTail.next.compareAndSet(null, newNode)) {
11         // Insertion succeeded, try advancing tail
12         tail.compareAndSet(curTail, newNode);
13         // will fail if tail already moved
14         return true;
15     }
16 }
17 }
18 }
19 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



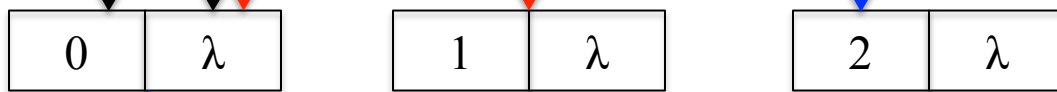
Trace 1: T1 wins

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      }
9      else { // In quiescent state, try inserting new node
10         if (curTail.next.compareAndSet(null, newNode)) {
11           // Insertion succeeded, try advancing tail
12           tail.compareAndSet(curTail, newNode);
13           // will fail if tail already moved
14           return true;
15         }
16       }
17     }
18   }
19 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



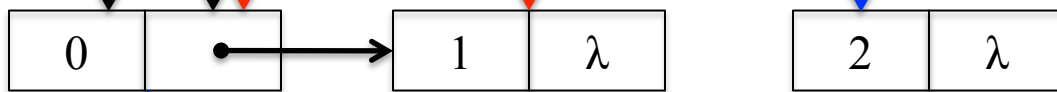
Trace 1: After CAS

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      }
9      else { // In quiescent state, try inserting new node
10         if (curTail.next.compareAndSet(null, newNode)) {
11           // Insertion succeeded, try advancing tail
12           tail.compareAndSet(curTail, newNode);
13           // will fail if tail already moved
14           return true;
15         }
16       }
17     }
18   }
19 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



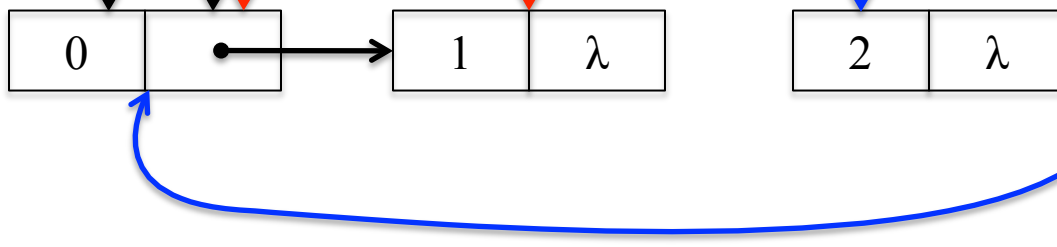
Trace 1: T2 Continues & CAS Fails

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



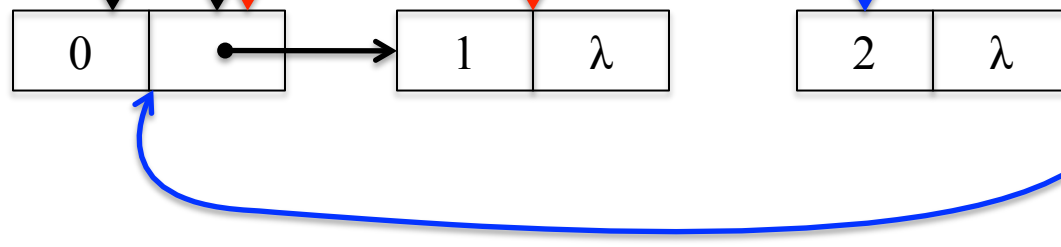
Trace 1: T1 Continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             if (curTail.next.compareAndSet(null, newNode)) {
11                 // Insertion succeeded, try advancing tail
12                 tail.compareAndSet(curTail, newNode);
13                 // will fail if tail already moved
14                 return true;
15             }
16         }
17     }
18 }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



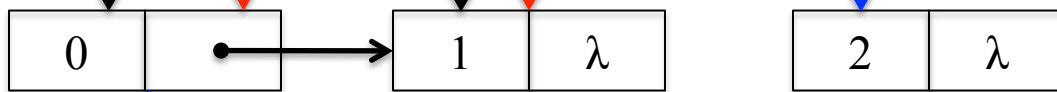
Trace 1: After CAS

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13                  return true;
14              }
15          }
16      }
17  }
18  }
19  }
20  }

```

Var	T1	T2
tailNext	λ	λ
curTail		
newNode		
head		
tail		



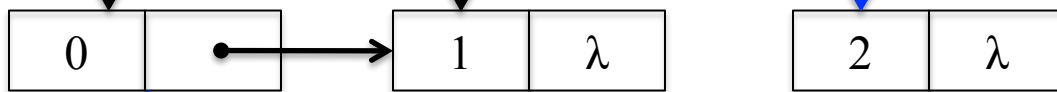
Trace 1: T1 exits

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



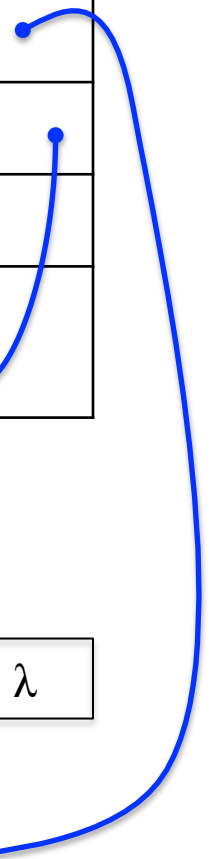
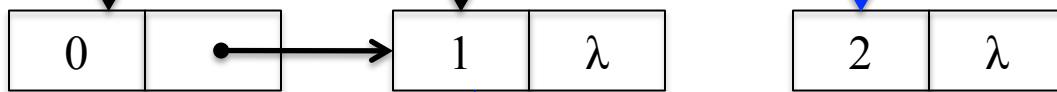
Trace 1: T2 Continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



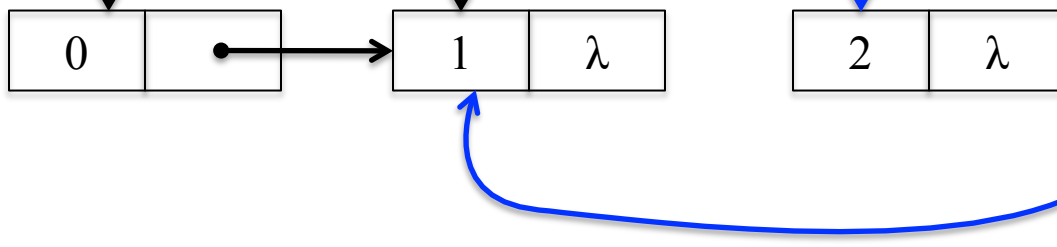
Trace 1: T2 Continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



Trace 1: T2 Continues

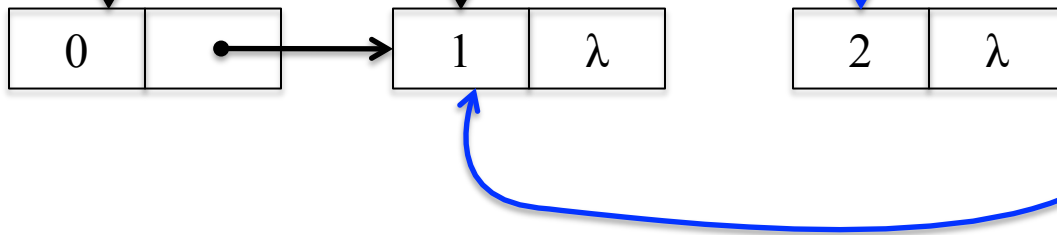
```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13              }
14          }
15      }
16      return true;
17  }
18  }
19  }
20  }
21  }

```



Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



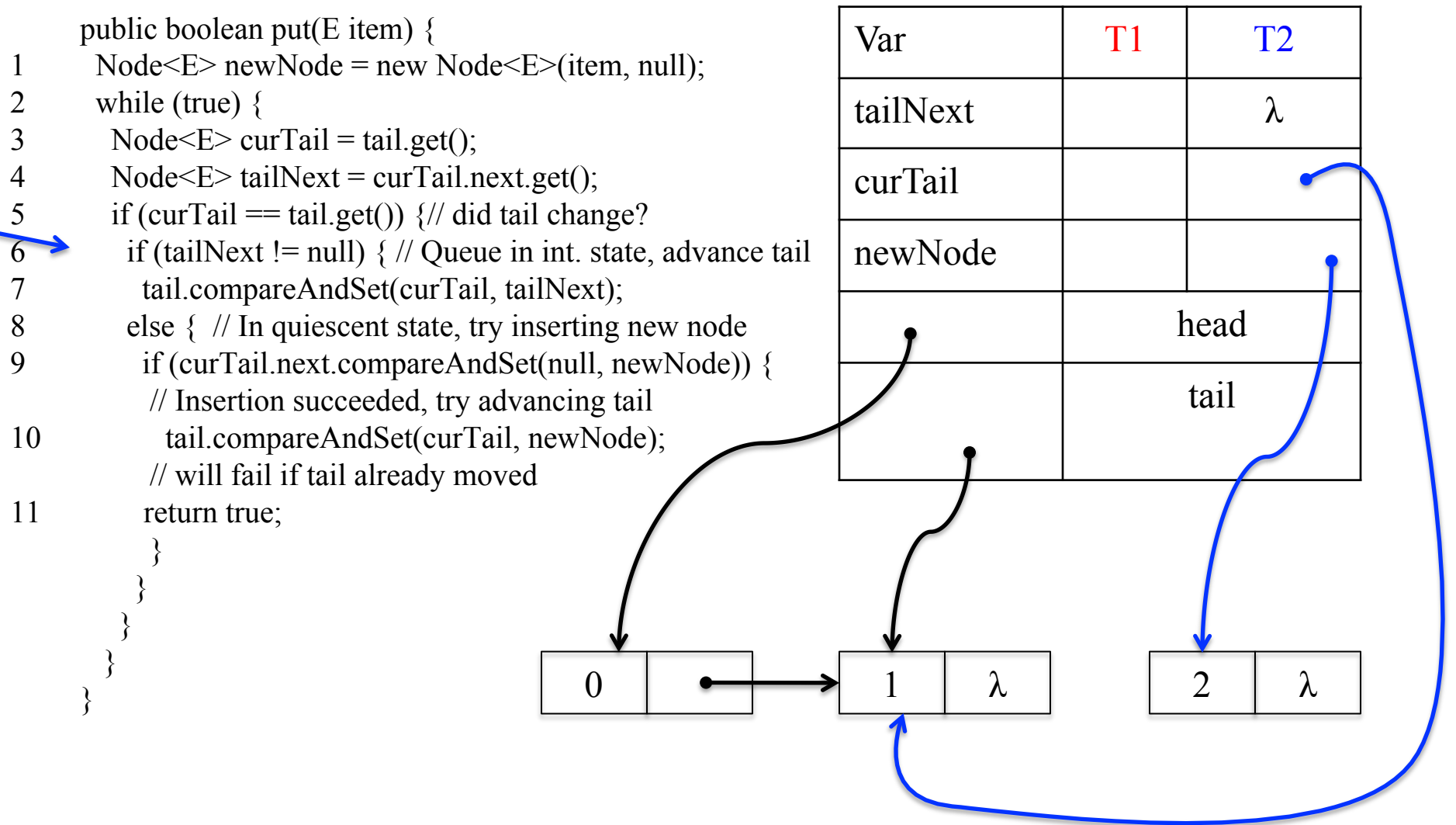
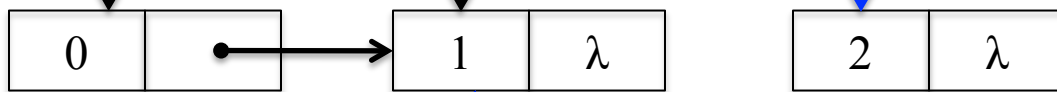
Trace 1: T2 Continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      } else { // In quiescent state, try inserting new node
9        if (curTail.next.compareAndSet(null, newNode)) {
10         // Insertion succeeded, try advancing tail
11         tail.compareAndSet(curTail, newNode);
12         // will fail if tail already moved
13         return true;
14       }
15     }
16   }
17 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



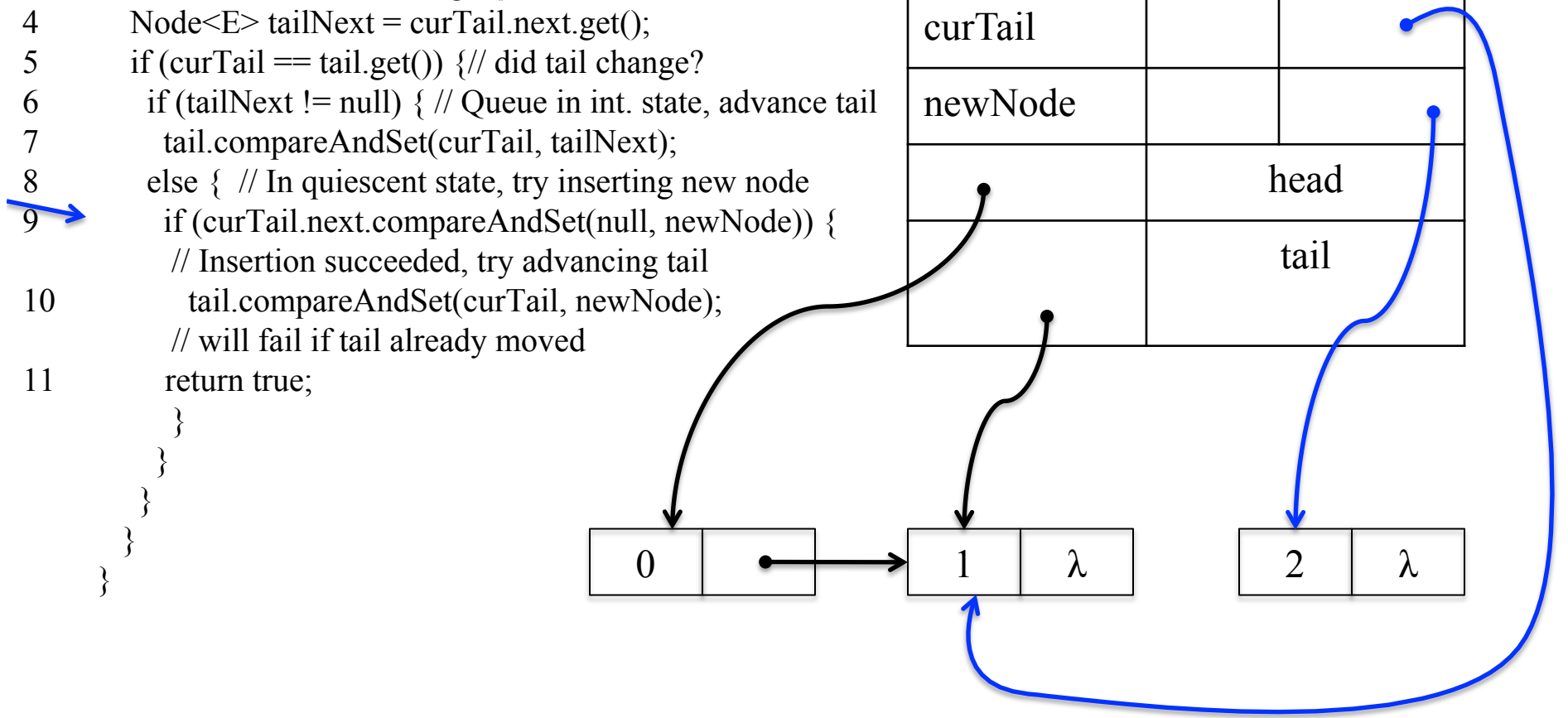
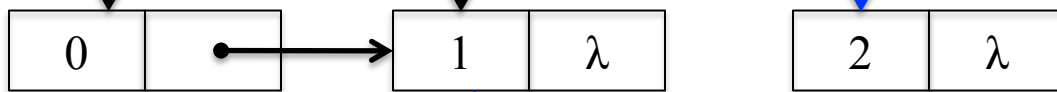
Trace 1: T2 Continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



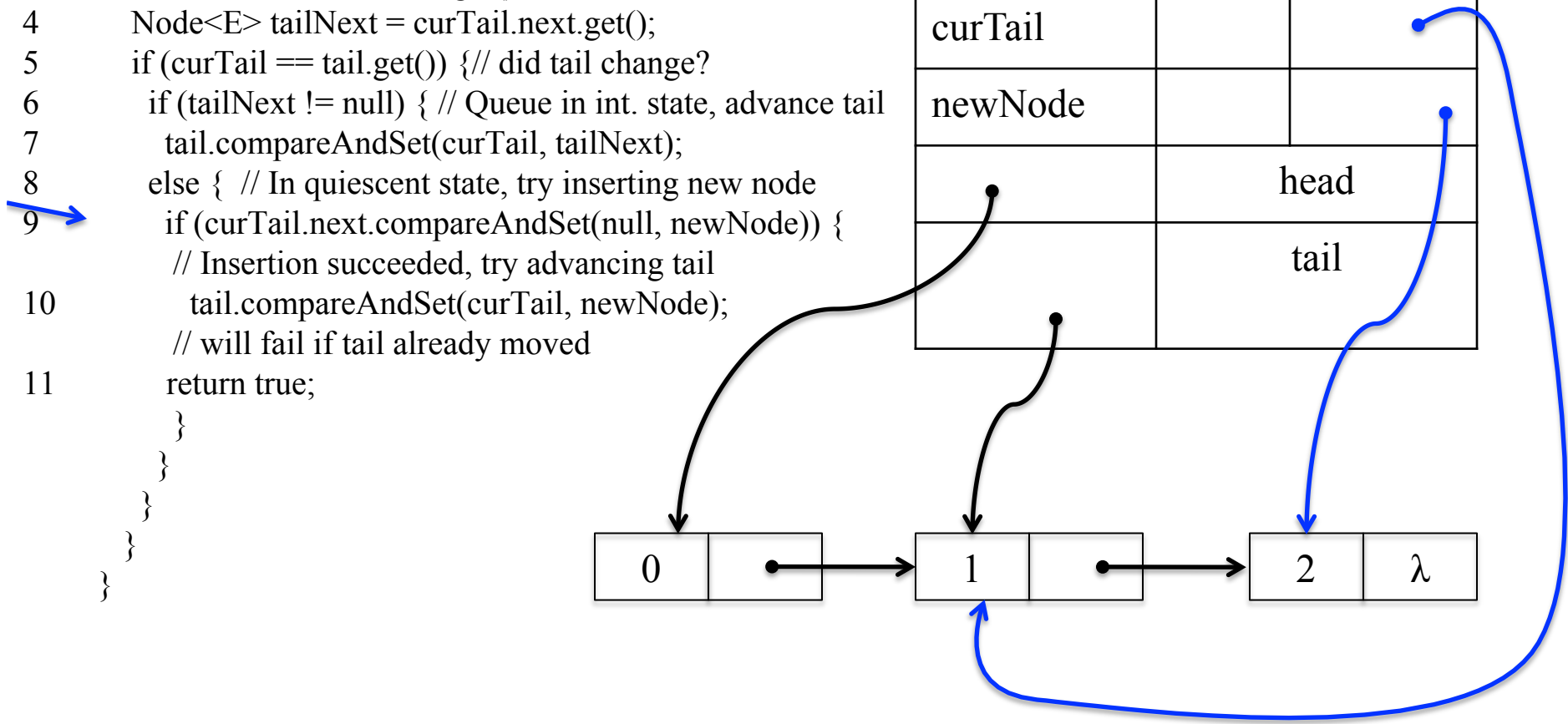
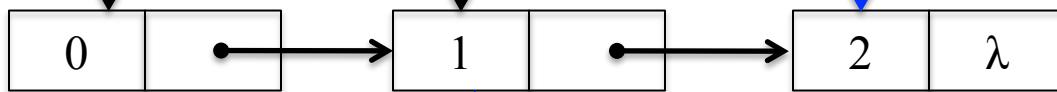
Trace 1: After CAS

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      }
9      else { // In quiescent state, try inserting new node
10         if (curTail.next.compareAndSet(null, newNode)) {
11           // Insertion succeeded, try advancing tail
12           tail.compareAndSet(curTail, newNode);
13           // will fail if tail already moved
14           return true;
15         }
16       }
17     }
18   }
19 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



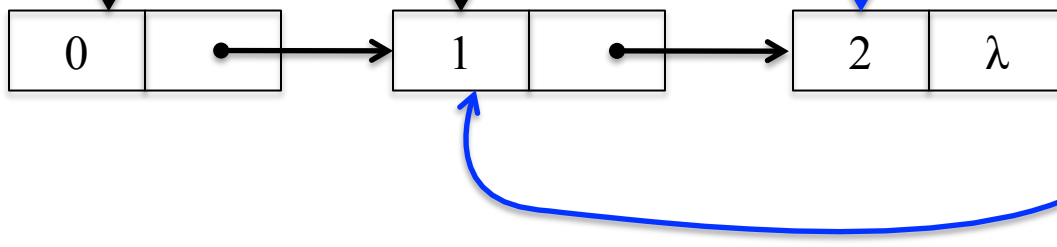
Trace 1: T2 Continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }
18 }
19 }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



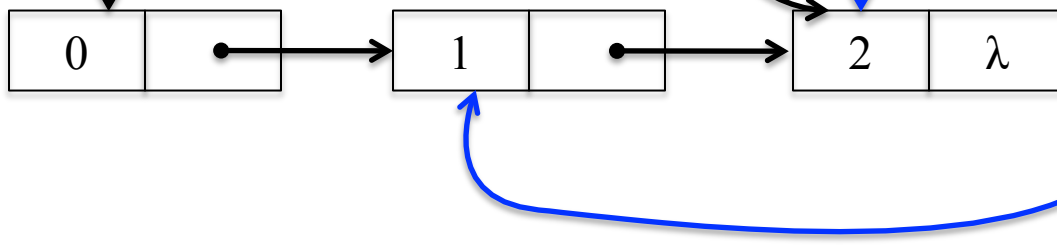
Trace 1: T2 Continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      } else { // In quiescent state, try inserting new node
9        if (curTail.next.compareAndSet(null, newNode)) {
10         // Insertion succeeded, try advancing tail
11         tail.compareAndSet(curTail, newNode);
12         // will fail if tail already moved
13         return true;
14       }
15     }
16   }
17 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	



Trace 1: T2 Exits

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



Trace 2

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



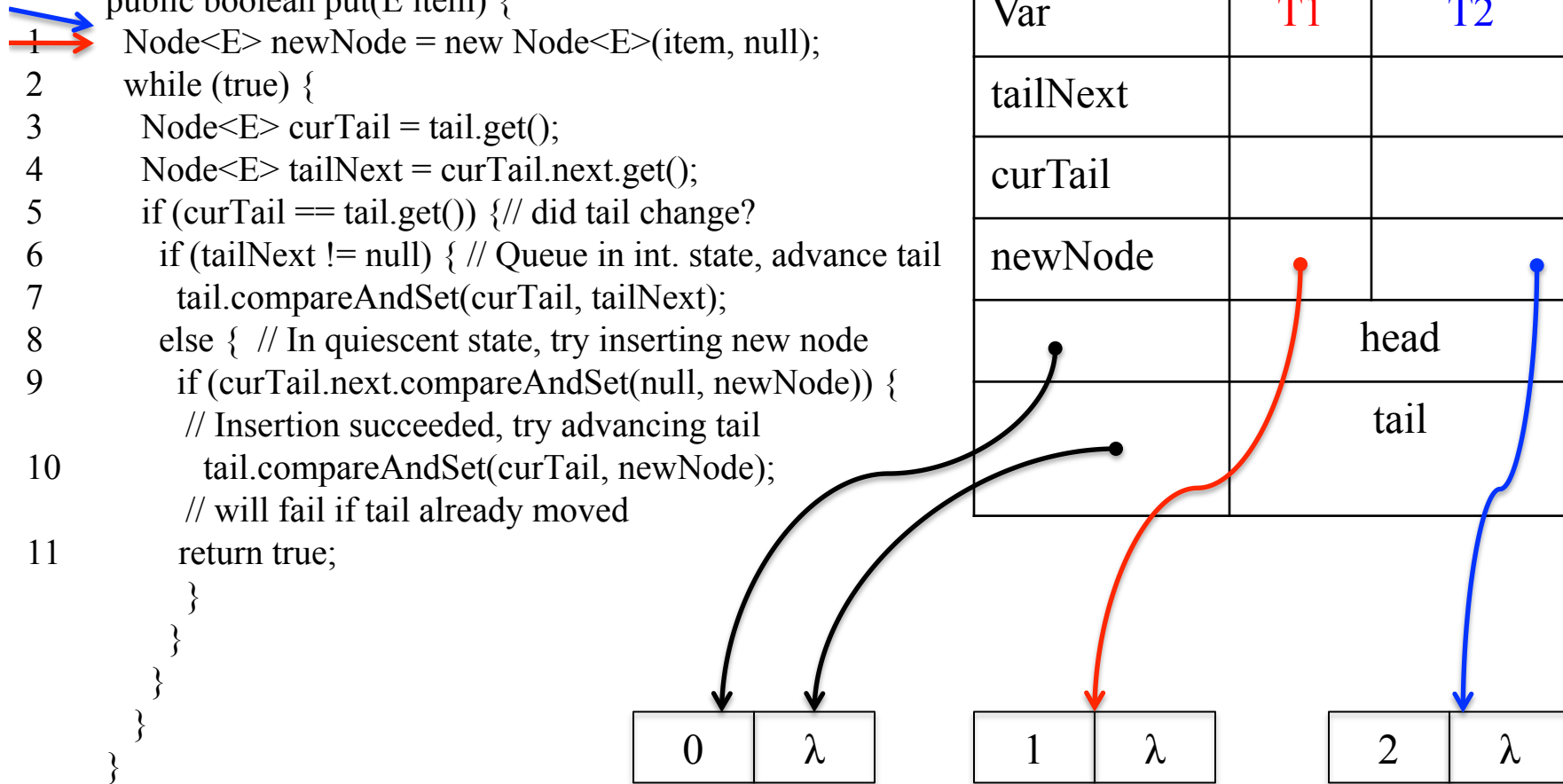
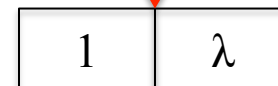
Trace 2

```

1 public boolean put(E item) {
2   Node<E> newNode = new Node<E>(item, null);
3   while (true) {
4     Node<E> curTail = tail.get();
5     Node<E> tailNext = curTail.next.get();
6     if (curTail == tail.get()) { // did tail change?
7       if (tailNext != null) { // Queue in int. state, advance tail
8         tail.compareAndSet(curTail, tailNext);
9       }
10      else { // In quiescent state, try inserting new node
11        if (curTail.next.compareAndSet(null, newNode)) {
12          // Insertion succeeded, try advancing tail
13          tail.compareAndSet(curTail, newNode);
14          // will fail if tail already moved
15          return true;
16        }
17      }
18    }
19  }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



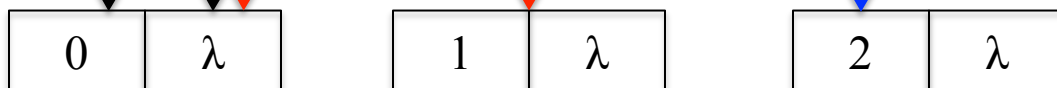
Trace 2

```

1 → public boolean put(E item) {
2   Node<E> newNode = new Node<E>(item, null);
3 →   while (true) {
4     Node<E> curTail = tail.get();
5     Node<E> tailNext = curTail.next.get();
6     if (curTail == tail.get()) { // did tail change?
7       if (tailNext != null) { // Queue in int. state, advance tail
8         tail.compareAndSet(curTail, tailNext);
9       }
10      else { // In quiescent state, try inserting new node
11        if (curTail.next.compareAndSet(null, newNode)) {
12          // Insertion succeeded, try advancing tail
13          tail.compareAndSet(curTail, newNode);
14          // will fail if tail already moved
15          return true;
16        }
17      }
18    }
19  }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
		head
		tail



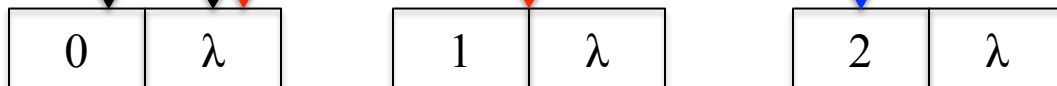
Trace 2

```

1 → public boolean put(E item) {
2   while (true) {
3     Node<E> curTail = tail.get();
4 →   Node<E> tailNext = curTail.next.get();
5     if (curTail == tail.get()) { // did tail change?
6       if (tailNext != null) { // Queue in int. state, advance tail
7         tail.compareAndSet(curTail, tailNext);
8       else { // In quiescent state, try inserting new node
9         if (curTail.next.compareAndSet(null, newNode)) {
10          // Insertion succeeded, try advancing tail
11          tail.compareAndSet(curTail, newNode);
12          // will fail if tail already moved
13          return true;
14        }
15      }
16    }
17  }
18 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
		head
		tail



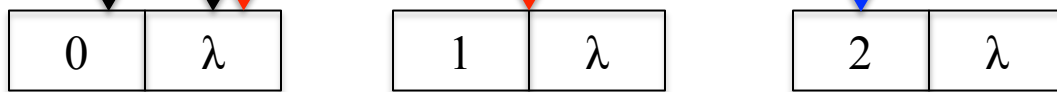
Trace 2

```

1 → public boolean put(E item) {
2   while (true) {
3     Node<E> curTail = tail.get();
4     Node<E> tailNext = curTail.next.get();
5 →   if (curTail == tail.get()) { // did tail change?
6     if (tailNext != null) { // Queue in int. state, advance tail
7       tail.compareAndSet(curTail, tailNext);
8     else { // In quiescent state, try inserting new node
9       if (curTail.next.compareAndSet(null, newNode)) {
10        // Insertion succeeded, try advancing tail
11        tail.compareAndSet(curTail, newNode);
12        // will fail if tail already moved
13        return true;
14      }
15    }
16  }
17 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
		head
		tail



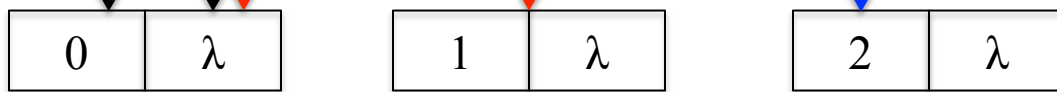
Trace 2

```

1 → public boolean put(E item) {
2   while (true) {
3     Node<E> curTail = tail.get();
4     Node<E> tailNext = curTail.next.get();
5     if (curTail == tail.get()) { // did tail change?
6 →   if (tailNext != null) { // Queue in int. state, advance tail
7     tail.compareAndSet(curTail, tailNext);
8   } else { // In quiescent state, try inserting new node
9     if (curTail.next.compareAndSet(null, newNode)) {
10    // Insertion succeeded, try advancing tail
11    tail.compareAndSet(curTail, newNode);
12    // will fail if tail already moved
13    return true;
14  }
15 }
16 }
17 }
18 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
	head	
	tail	



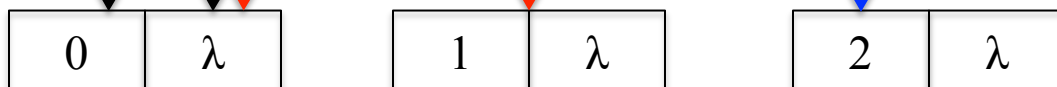
Trace 2

```

1 → public boolean put(E item) {
2   while (true) {
3     Node<E> curTail = tail.get();
4     Node<E> tailNext = curTail.next.get();
5     if (curTail == tail.get()) { // did tail change?
6       if (tailNext != null) { // Queue in int. state, advance tail
7         tail.compareAndSet(curTail, tailNext);
8       }
9       else { // In quiescent state, try inserting new node
10        if (curTail.next.compareAndSet(null, newNode)) {
11          // Insertion succeeded, try advancing tail
12          tail.compareAndSet(curTail, newNode);
13          // will fail if tail already moved
14          return true;
15        }
16      }
17    }
18  }
19 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
	head	
	tail	



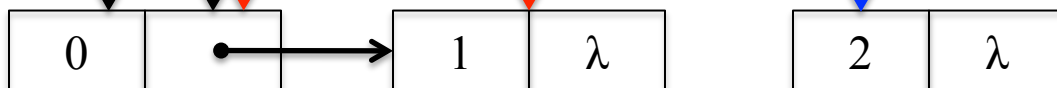
Trace 2: After CAS

```

1 → public boolean put(E item) {
2   while (true) {
3     Node<E> curTail = tail.get();
4     Node<E> tailNext = curTail.next.get();
5     if (curTail == tail.get()) { // did tail change?
6       if (tailNext != null) { // Queue in int. state, advance tail
7         tail.compareAndSet(curTail, tailNext);
8       }
9 →   else { // In quiescent state, try inserting new node
10      if (curTail.next.compareAndSet(null, newNode)) {
11        // Insertion succeeded, try advancing tail
12        tail.compareAndSet(curTail, newNode);
13        // will fail if tail already moved
14        return true;
15      }
16    }
17  }
18 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
	head	
	tail	



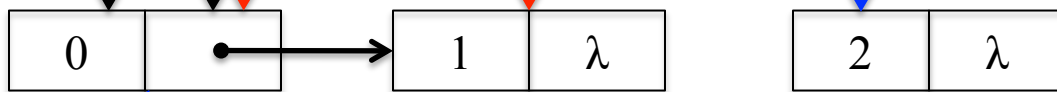
Trace 2: T2 continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             if (curTail.next.compareAndSet(null, newNode)) {
11                 // Insertion succeeded, try advancing tail
12                 tail.compareAndSet(curTail, newNode);
13                 // will fail if tail already moved
14                 return true;
15             }
16         }
17     }
18 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
		head
		tail



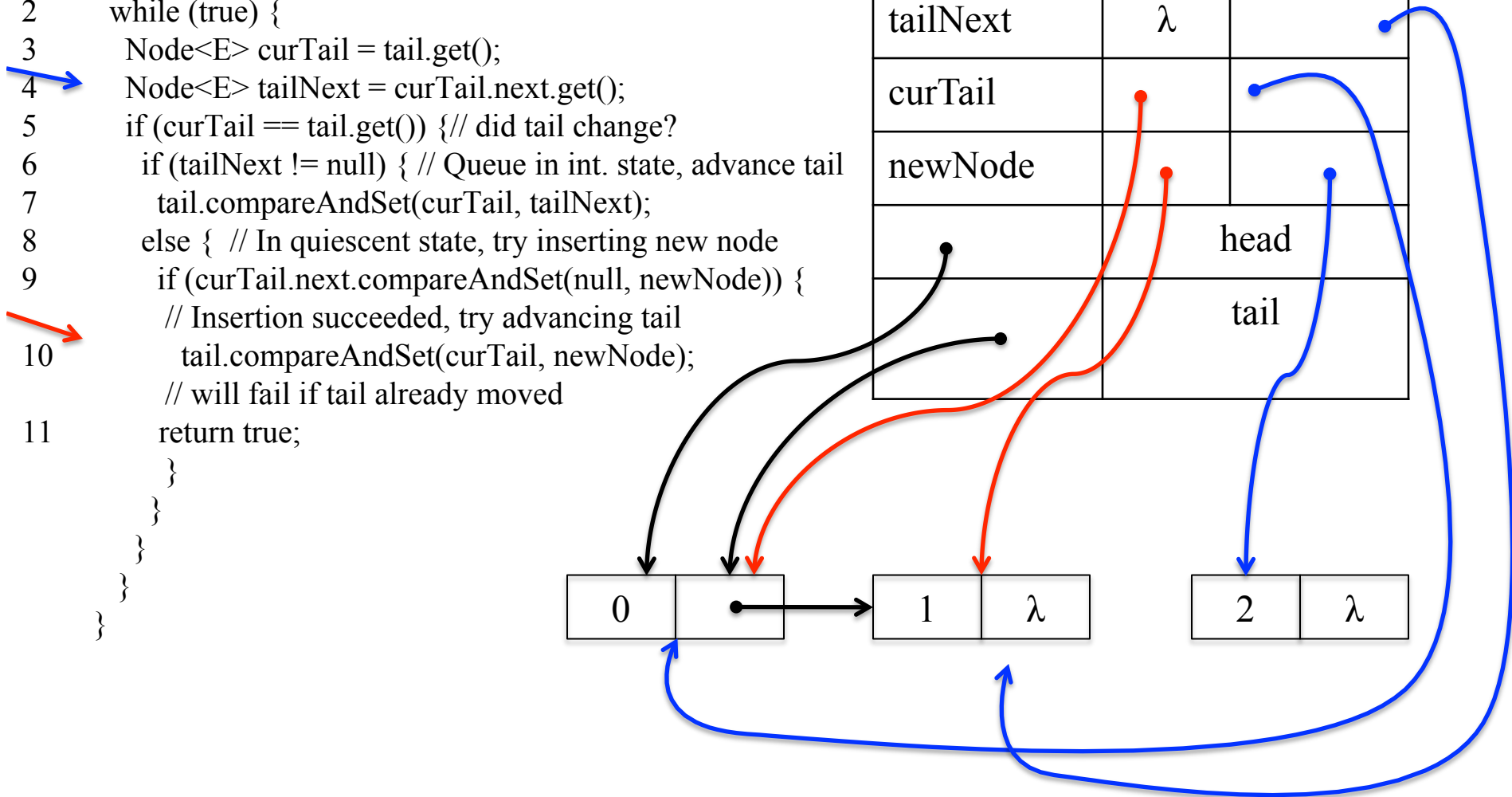
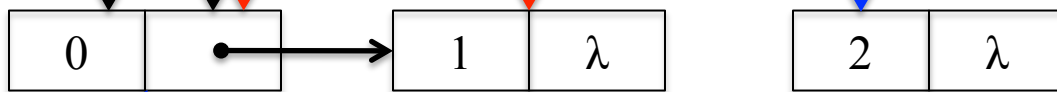
Trace 2

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13             }
14             return true;
15         }
16     }
17 }
18 }
19 }
20 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
head		
tail		



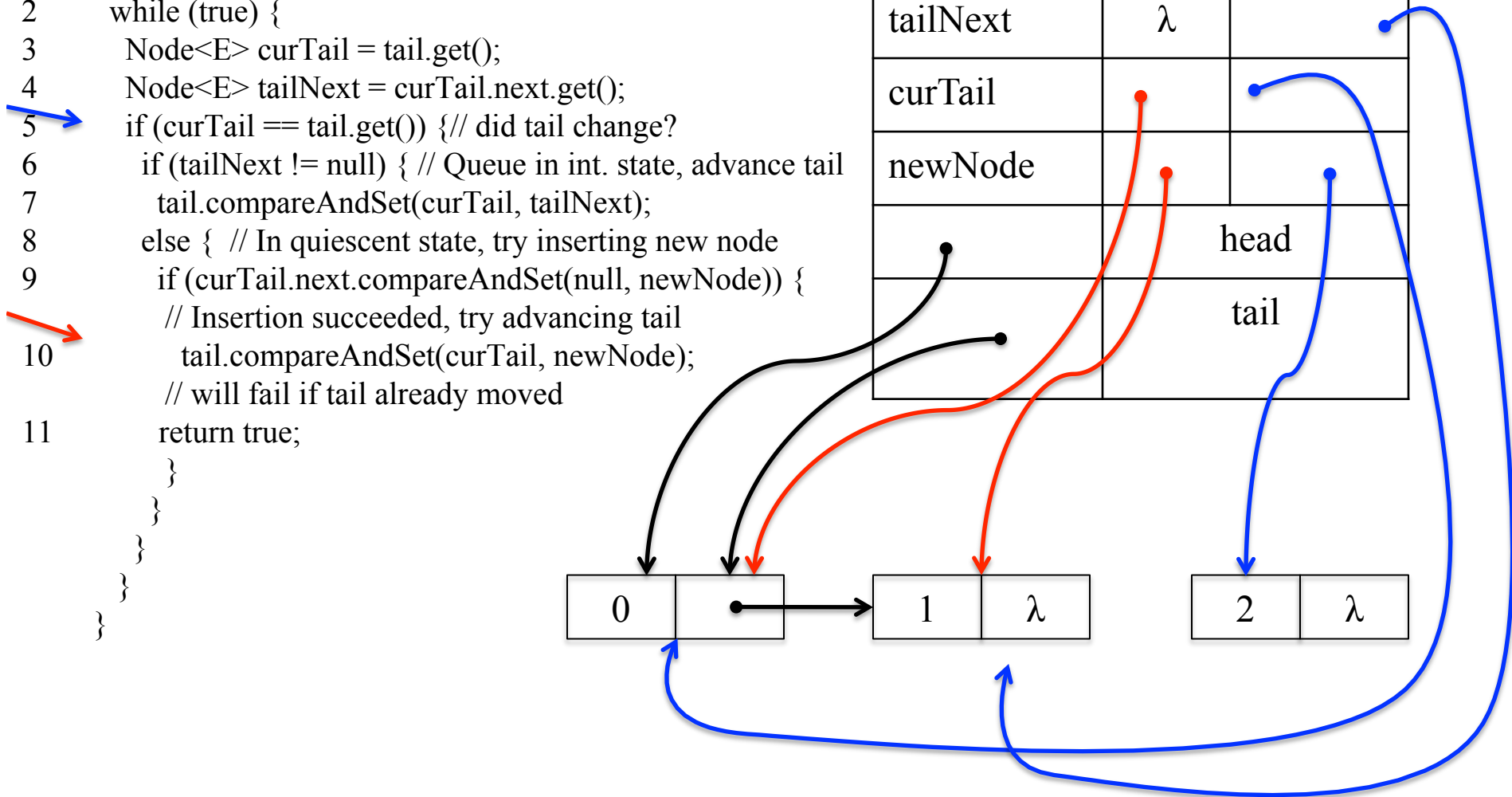
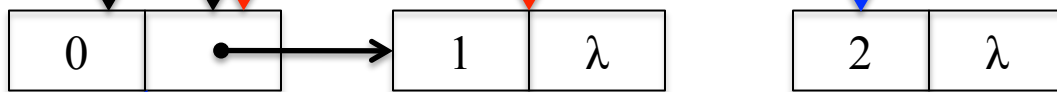
Trace 2

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13             }
14             return true;
15         }
16     }
17 }
18 }
19 }
20 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
	head	
	tail	



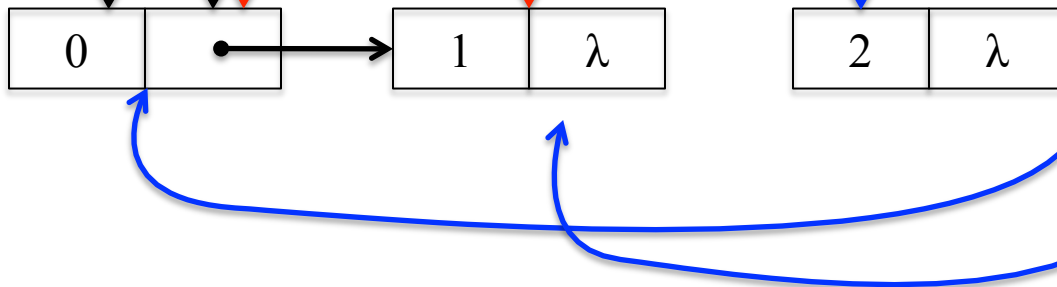
```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13              }
14          }
15      }
16  }
17  }
18  }
19  }
20  }

```

The diagram illustrates a linked list structure. It consists of two rectangular nodes. The first node contains the value '0'. The second node contains a black dot, which represents a pointer to the next node in the list. A blue arrow points from line 6 of the code to the first node, indicating the state of the tail pointer. A red arrow points from line 10 of the code to the second node, indicating the state of the tail pointer after an insertion attempt.

Var	T1	T2
tailNext	λ	
curTail		
newNode		
		head
		tail



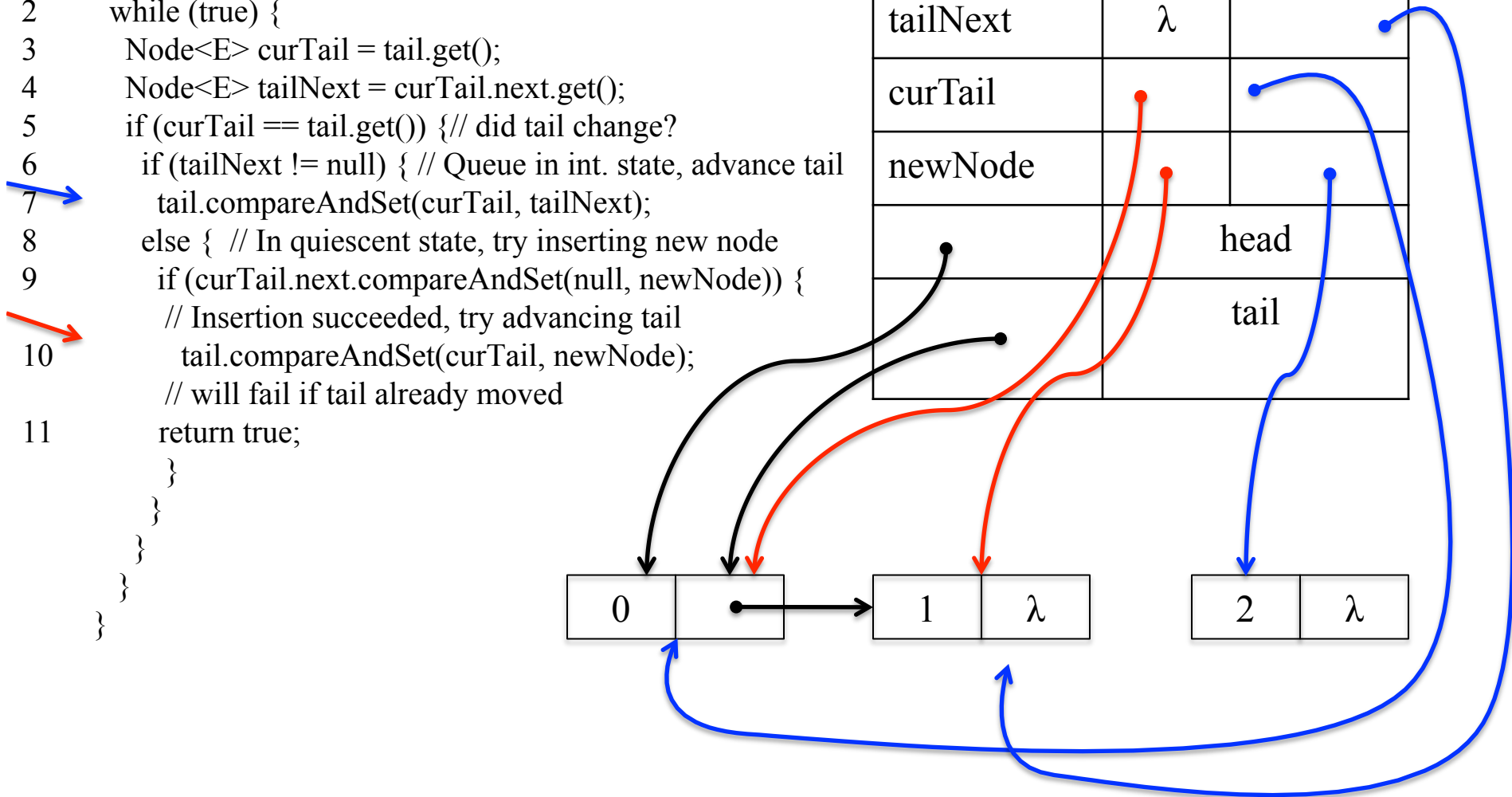
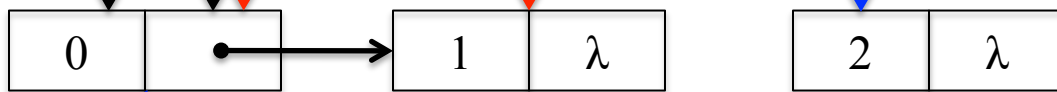
Trace 2: T2 wins

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          }
9          else { // In quiescent state, try inserting new node
10             if (curTail.next.compareAndSet(null, newNode)) {
11                 // Insertion succeeded, try advancing tail
12                 tail.compareAndSet(curTail, newNode);
13                 // will fail if tail already moved
14             }
15             return true;
16         }
17     }
18 }
19 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
head		
tail		



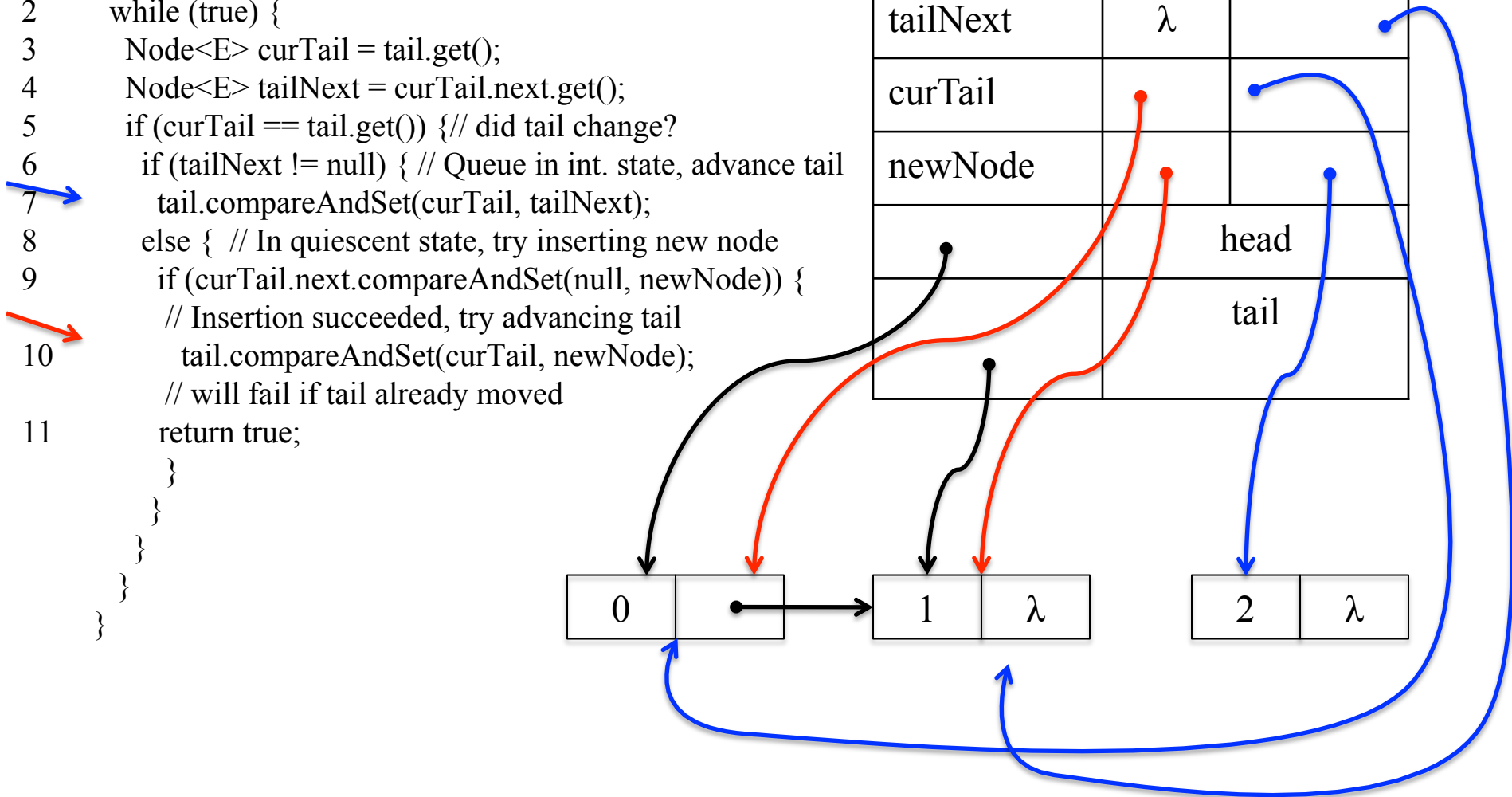
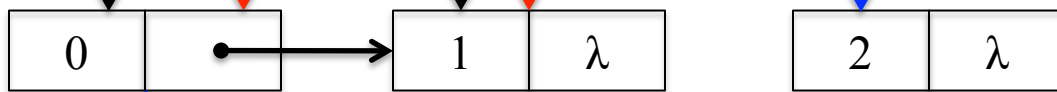
Trace 2: After CAS

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          }
9          else { // In quiescent state, try inserting new node
10             if (curTail.next.compareAndSet(null, newNode)) {
11                 // Insertion succeeded, try advancing tail
12                 tail.compareAndSet(curTail, newNode);
13                 // will fail if tail already moved
14             }
15             return true;
16         }
17     }
18 }
19 }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
head		
tail		



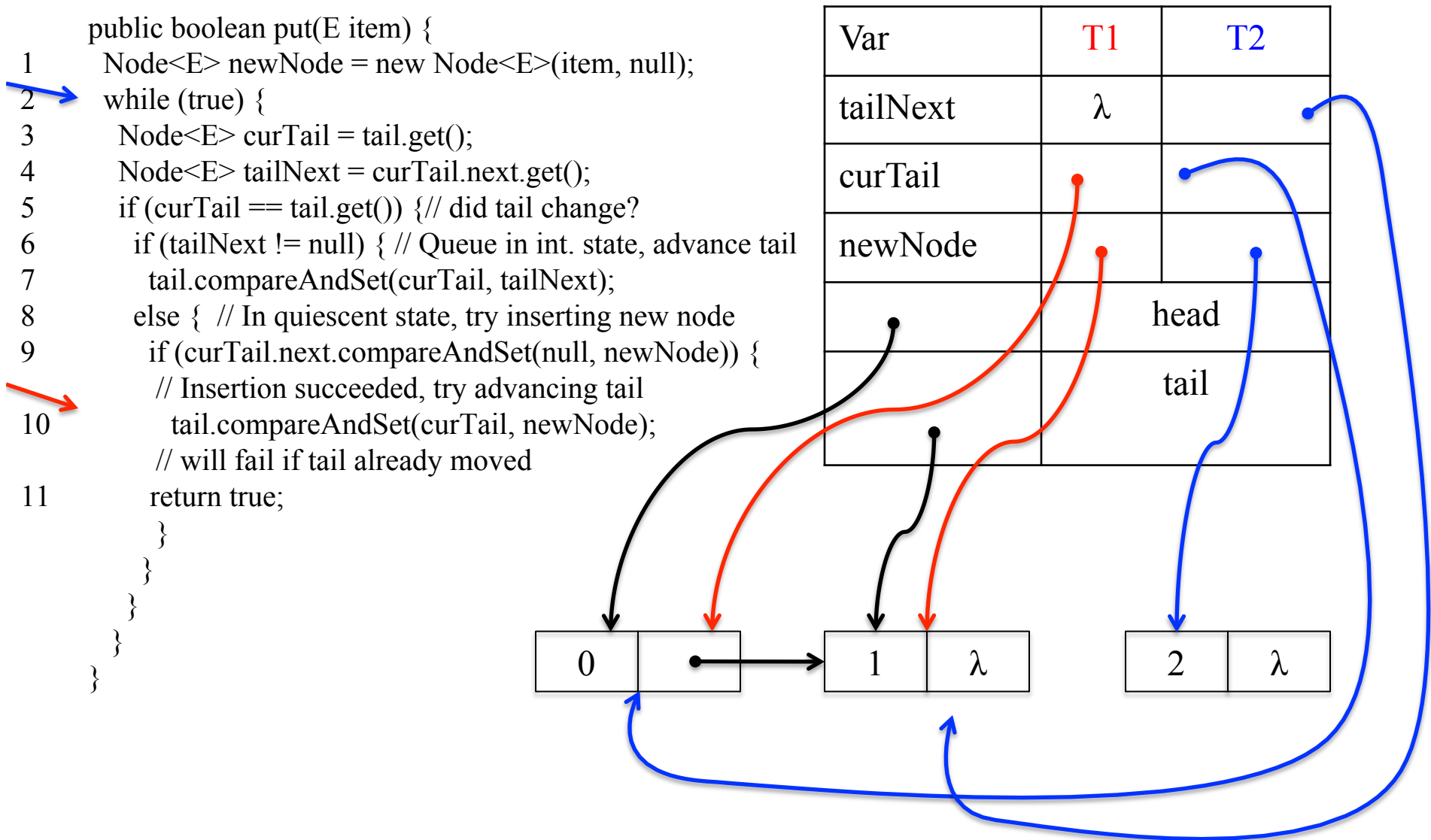
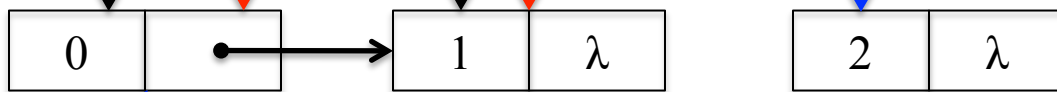
Trace 2: T1 continues: CAS fails

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13                  return true;
14              }
15          }
16      }
17  }
18  }
19  }

```

Var	T1	T2
tailNext	λ	
curTail		
newNode		
head		
tail		



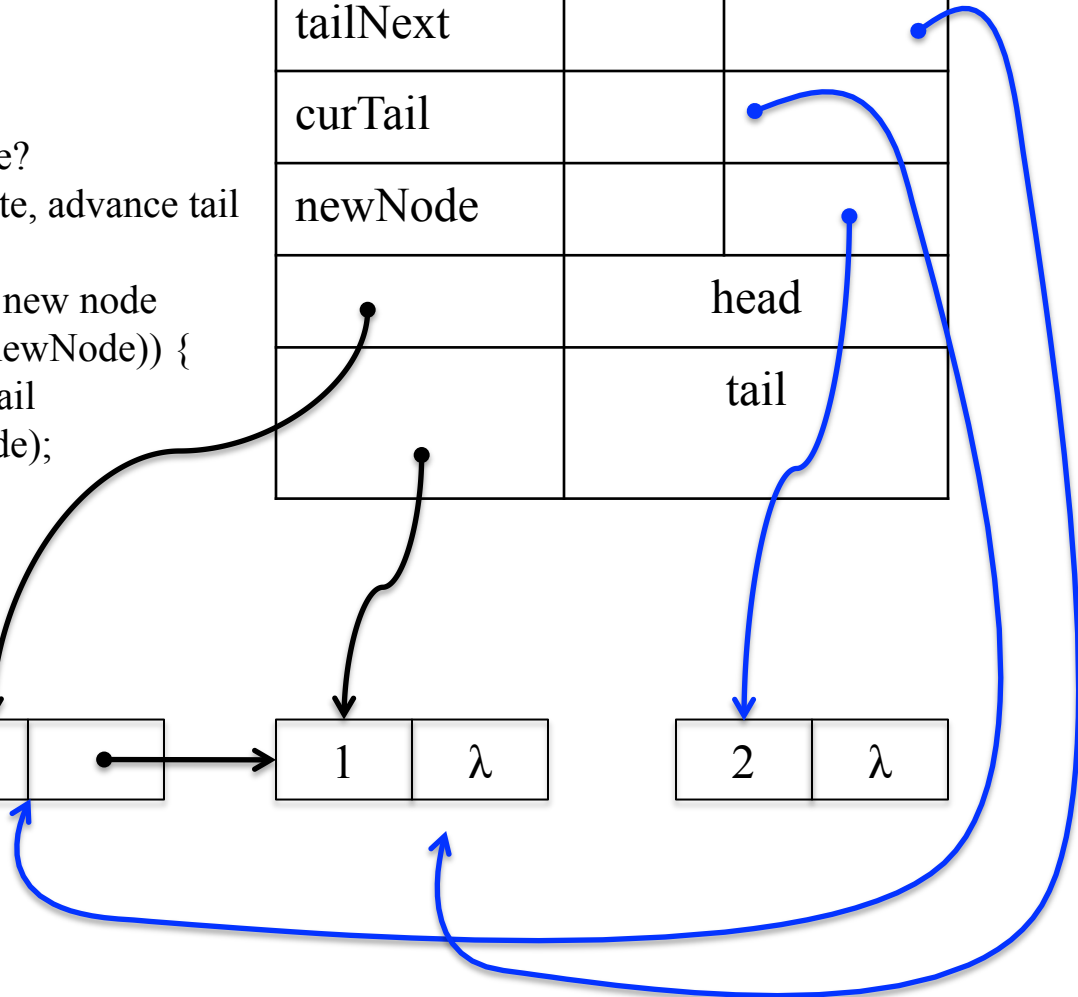
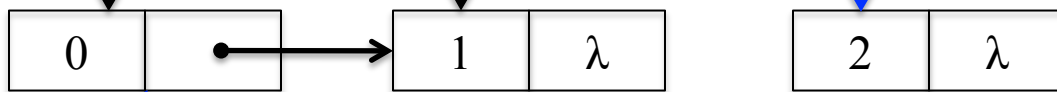
Trace 2: T1 exits

```

1  public boolean put(E item) {
2  →  Node<E> newNode = new Node<E>(item, null);
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



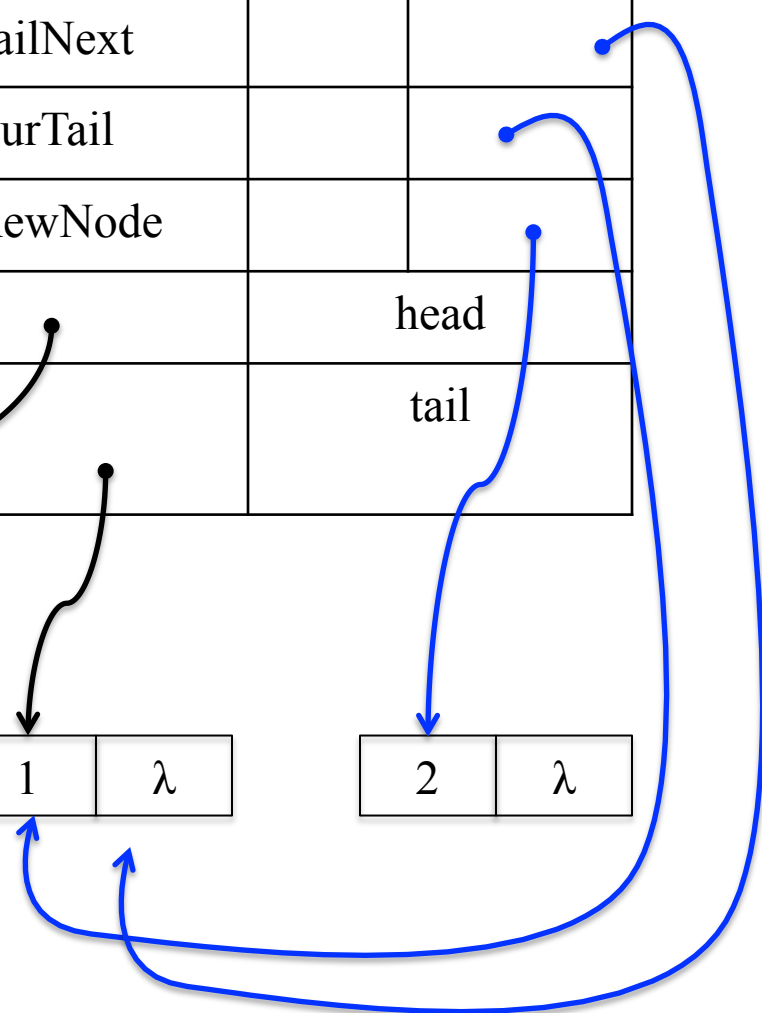
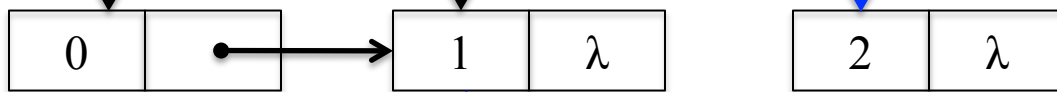
Trace 2: T2 continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



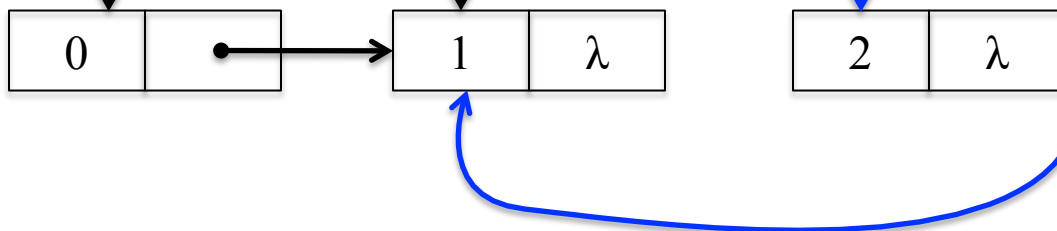
Trace 2: T2 continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



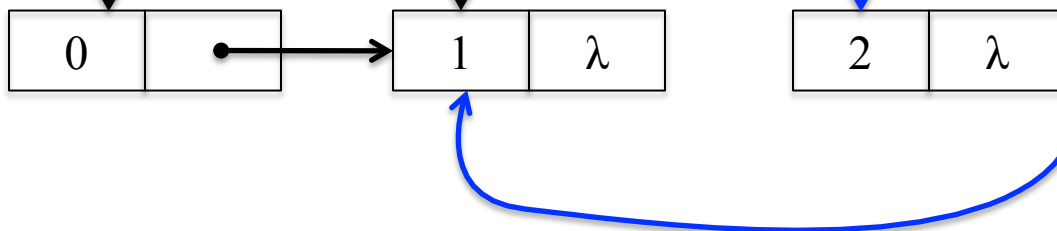
Trace 2: T2 continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }
18 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



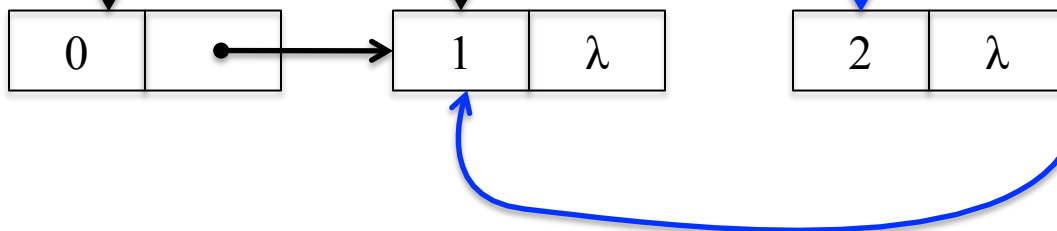
Trace 2: T2 continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3    Node<E> curTail = tail.get();
4    Node<E> tailNext = curTail.next.get();
5    if (curTail == tail.get()) { // did tail change?
6      if (tailNext != null) { // Queue in int. state, advance tail
7        tail.compareAndSet(curTail, tailNext);
8      } else { // In quiescent state, try inserting new node
9        if (curTail.next.compareAndSet(null, newNode)) {
10         // Insertion succeeded, try advancing tail
11         tail.compareAndSet(curTail, newNode);
12         // will fail if tail already moved
13         return true;
14       }
15     }
16   }
17 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



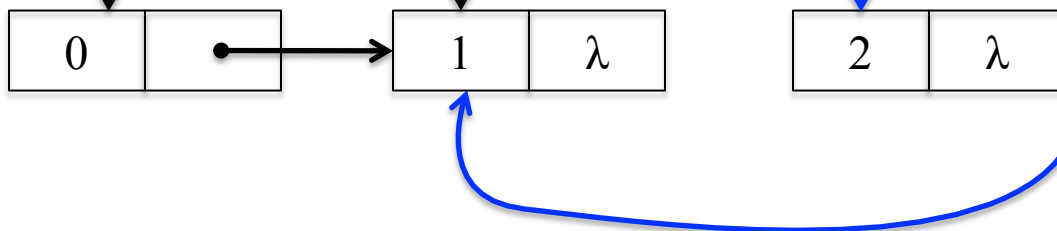
Trace 2: T2 continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



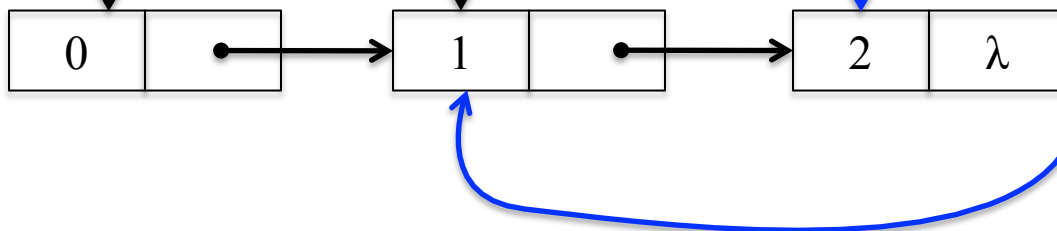
Trace 2: After CAS

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           }
9       } else { // In quiescent state, try inserting new node
10          if (curTail.next.compareAndSet(null, newNode)) {
11              // Insertion succeeded, try advancing tail
12              tail.compareAndSet(curTail, newNode);
13              // will fail if tail already moved
14              return true;
15          }
16      }
17  }
18  }
19  }
20  }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



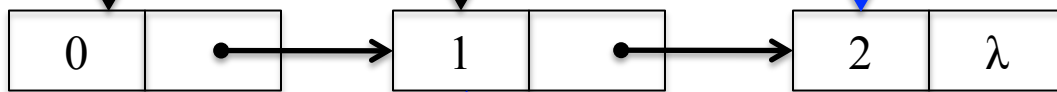
Trace 2: T2 continues

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }
18 }
19 }
20 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



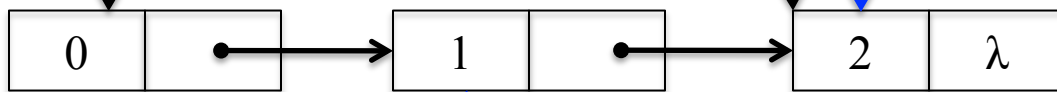
Trace 2: After CAS

```

public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }
18 }

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
head		
tail		



Trace 2: T2 Exits

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

Var	T1	T2
tailNext		
curTail		
newNode		
	head	
	tail	



Execution Traces

- 2 threads attempt to take()

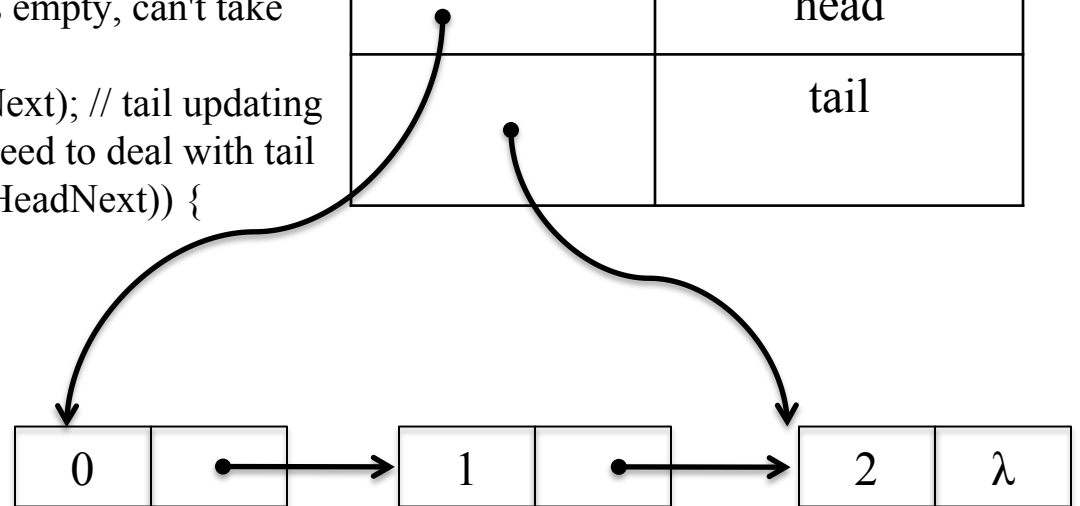
Trace 1

```

public E take() {
  for (;;) {
    1 Node<E> oldHead = head.get();           // current head
    2 Node<E> oldTail = tail.get();           // current tail
    3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    4 if (oldHead == head.get()) {           //head, tail, and next changed?
    5   if (oldHead == oldTail) {             //Queue empty or tail updated?
    6     if (oldHeadNext == null) { // Is queue empty?
    7       return null;                  //Queue is empty, can't take
    8   }
    9   tail.compareAndSet(oldTail, oldHeadNext); // tail updating
   10  } else {                           // No need to deal with tail
      if (head.compareAndSet(oldHead, oldHeadNext)) {
        return oldHeadNext.item;
      }
    }
  }
}

```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
	head	
	tail	



Trace 1

```

public E take() {
    for (;;) {
1 → Node<E> oldHead = head.get();           // current head
2 → Node<E> oldTail = tail.get();           // current tail
3 → Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4 → if (oldHead == head.get()) {           //head, tail, and next changed?
5 →     if (oldHead == oldTail) {           //Queue empty or tail updated?
6 →         if (oldHeadNext == null) { // Is queue empty?
7 →             return null;               //Queue is empty, can't take
            }
8 →         tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else {                           // No need to deal with tail
9 →             if (head.compareAndSet(oldHead, oldHeadNext)) {
10 →                 return oldHeadNext.item;
            }
        }
    }
}
    }
}
    
```

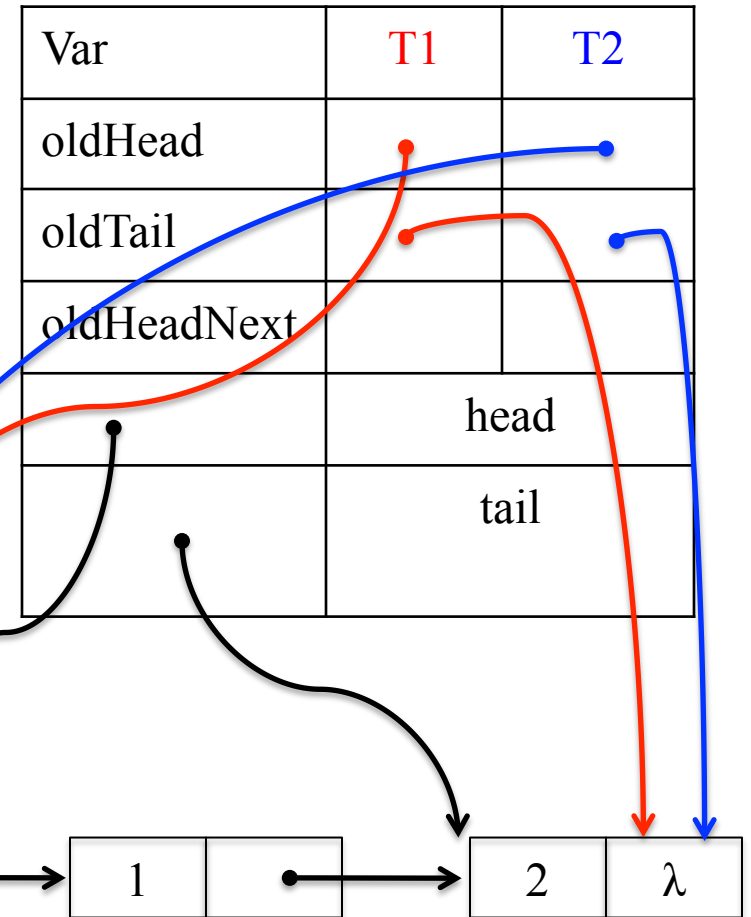
Var	T1	T2
oldHead	●	●
oldTail		
oldHeadNext		
	head	
	tail	



Trace 1

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

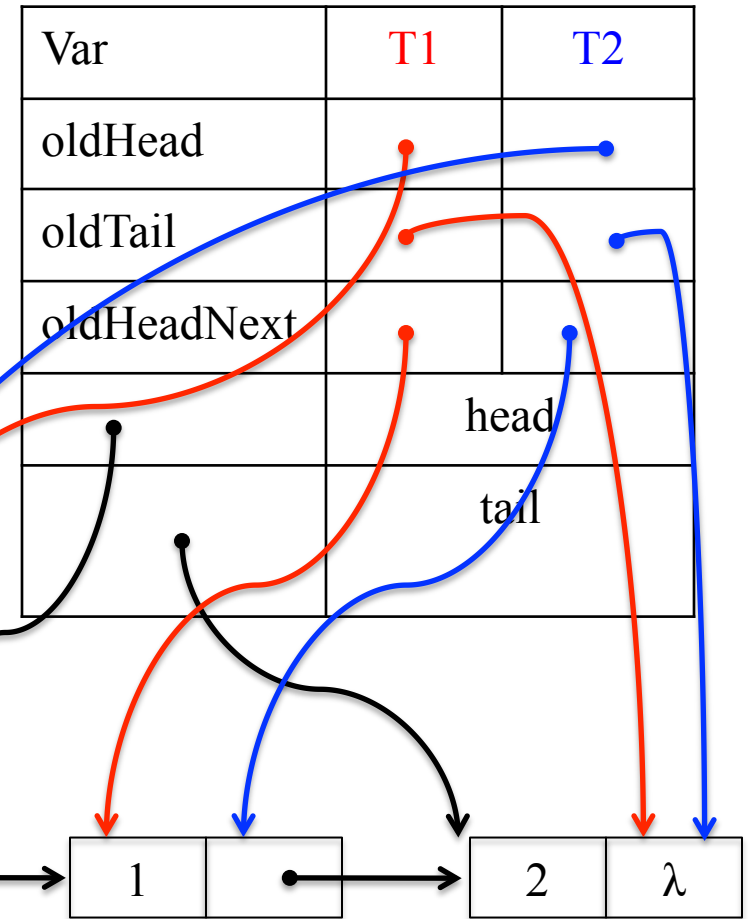


Trace 1

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {             //head, tail, and next changed?
5      if (oldHead == oldTail) {              //Queue empty or tail updated?
6        if (oldHeadNext == null) {           // Is queue empty?
7          return null;                       //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                               // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }

```

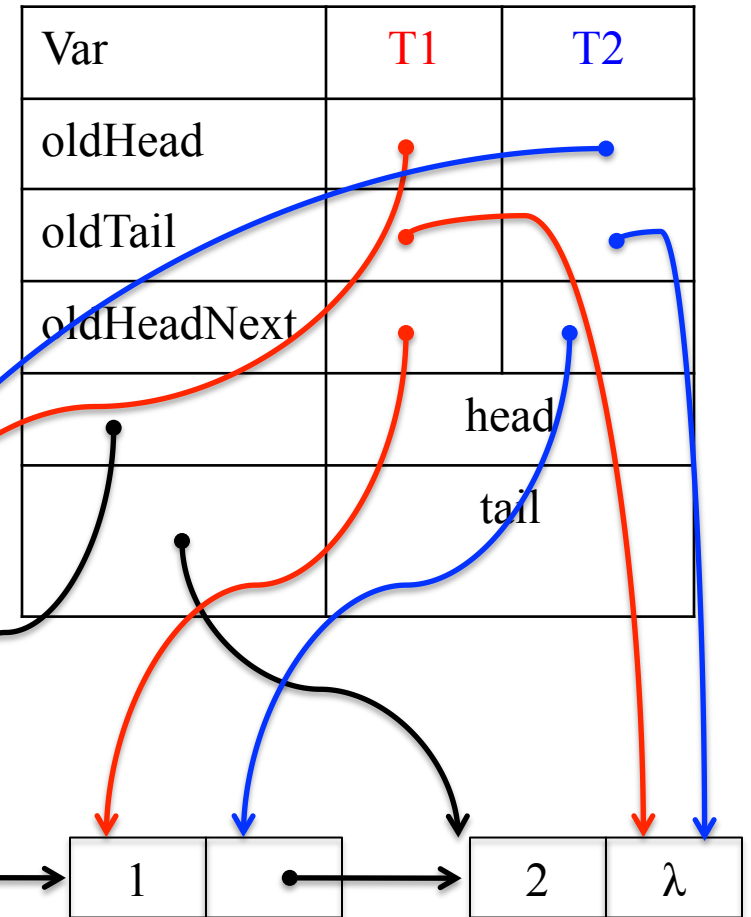


Trace 1

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }

```

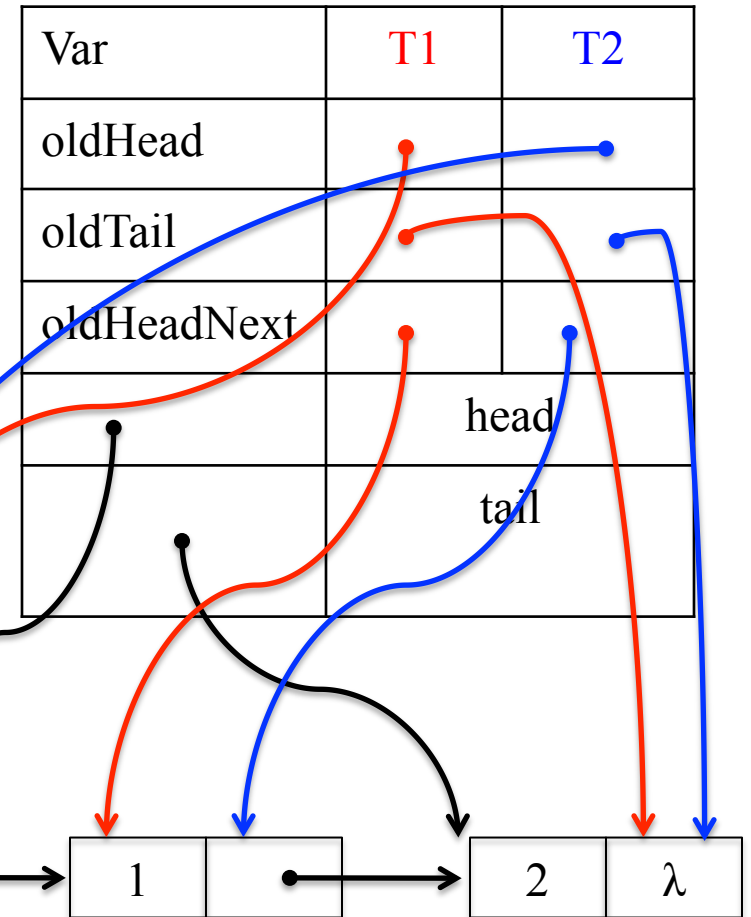


Trace 1

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5    → if (oldHead == oldTail) {           //Queue empty or tail updated?
6      if (oldHeadNext == null) { // Is queue empty?
7        return null;           //Queue is empty, can't take
8      }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                  // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }

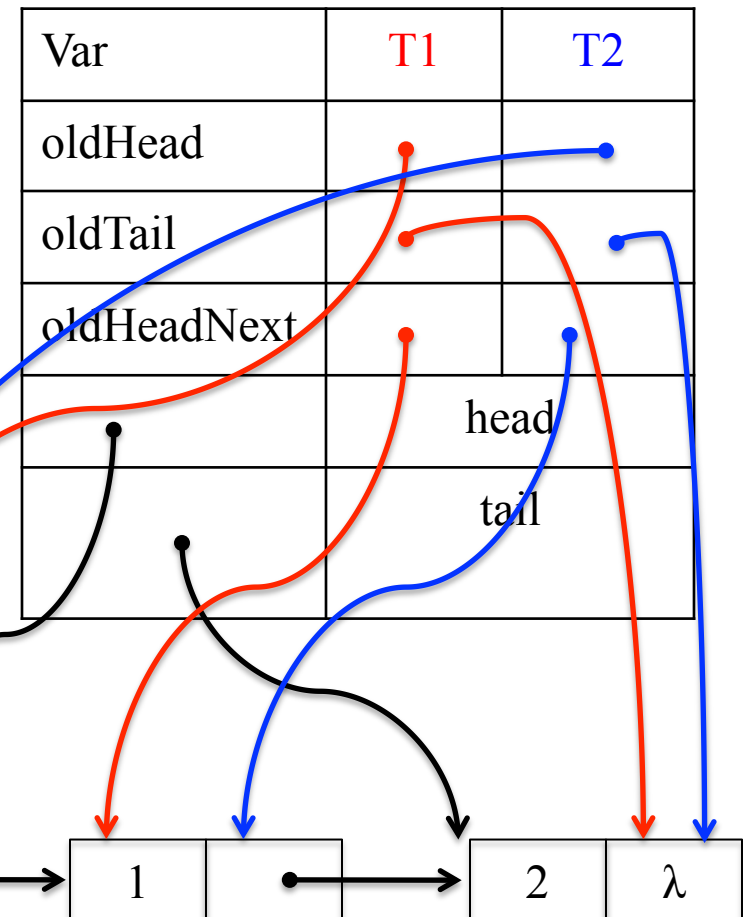
```



Trace 1: T1 wins, CAS succeeds

```

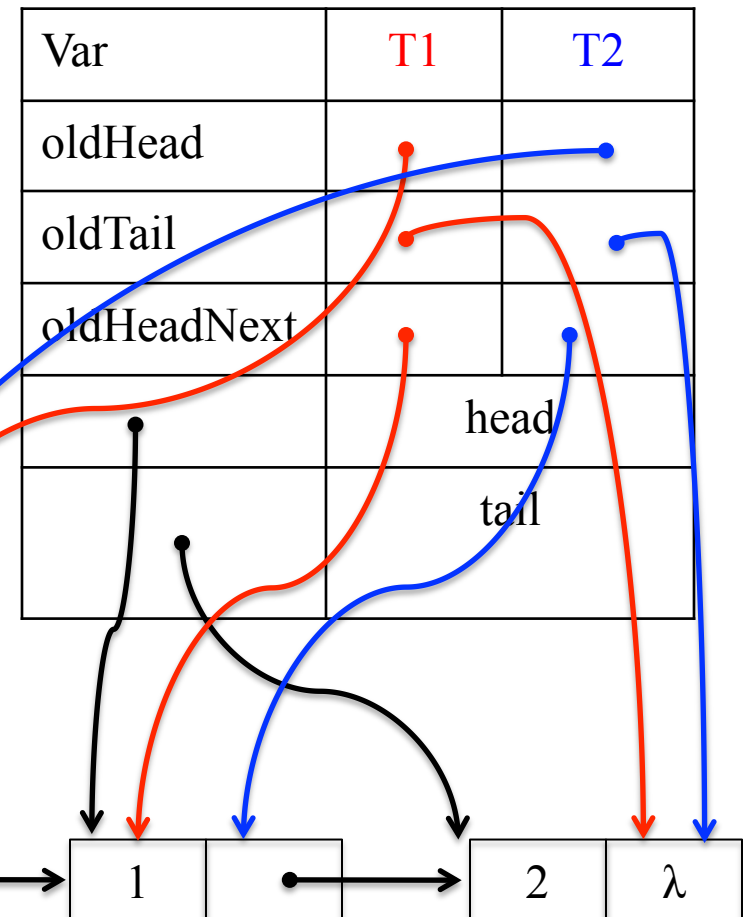
public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {             //head, tail, and next changed?
5      if (oldHead == oldTail) {              //Queue empty or tail updated?
6        if (oldHeadNext == null) {          // Is queue empty?
7          return null;                      //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                               // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: T1 wins - after CAS

```

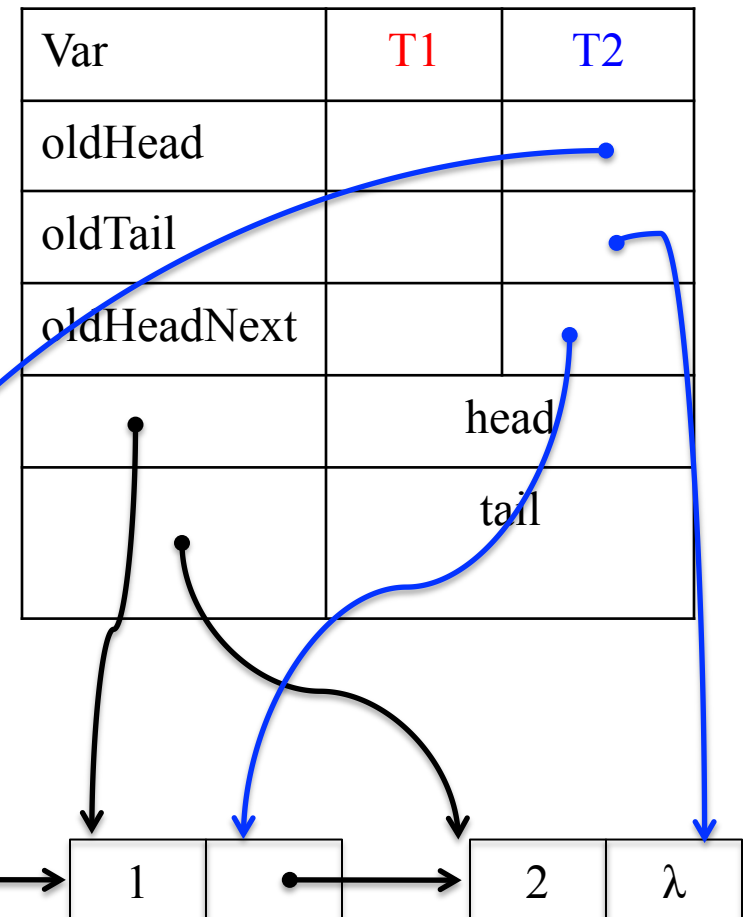
public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: T1 Exits

```

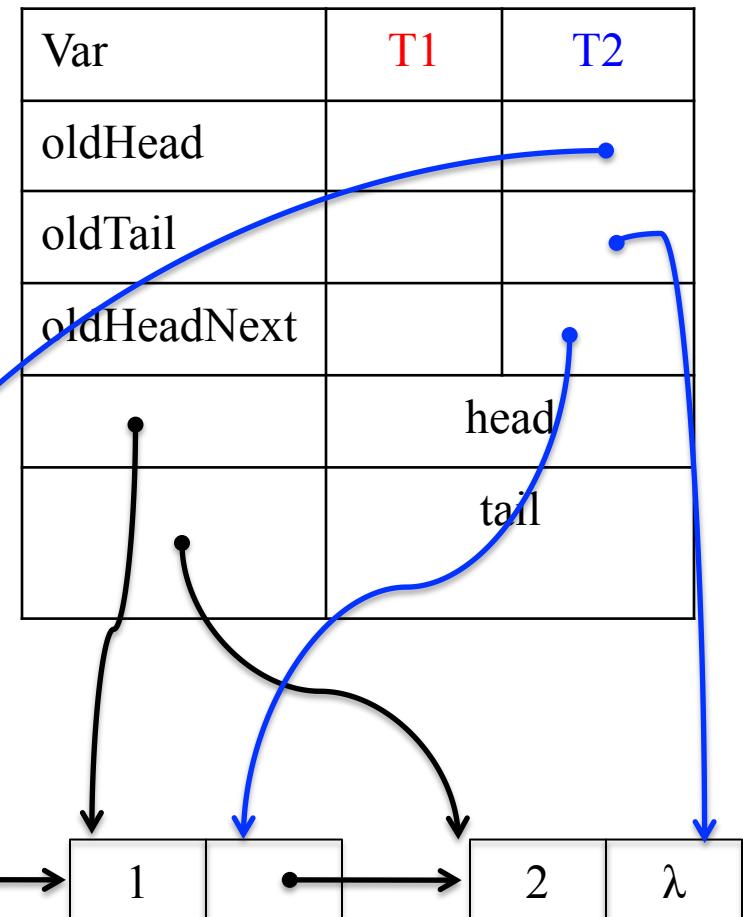
public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: T2 continues, CAS fails

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

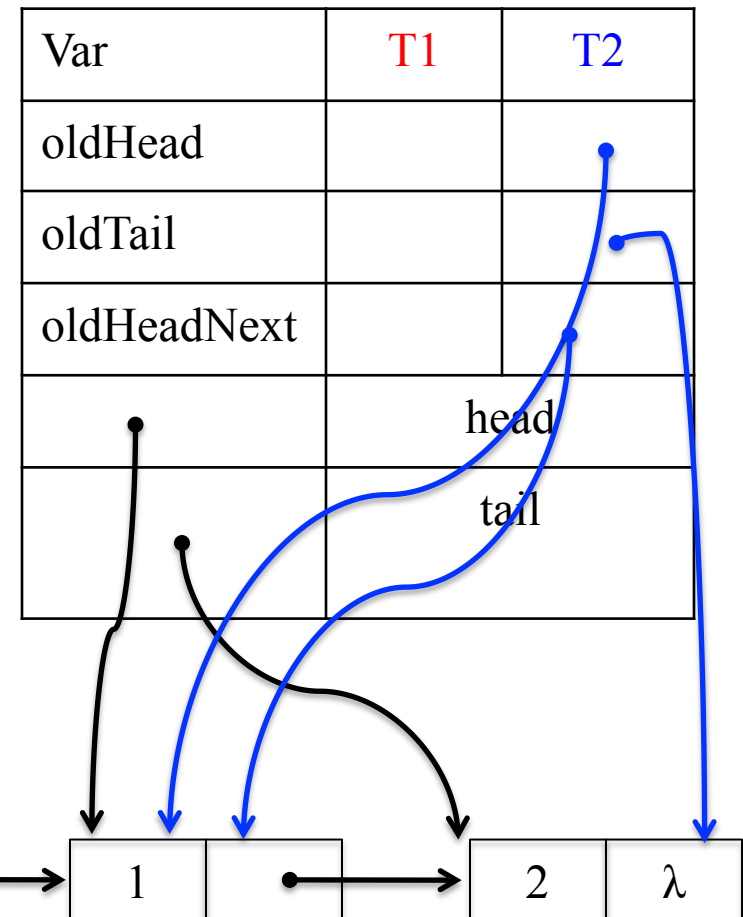


Trace 1: T2 continues

```

public E take() {
    for (;;) {
1 → Node<E> oldHead = head.get();           // current head
2   Node<E> oldTail = tail.get();           // current tail
3   Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4   if (oldHead == head.get()) {             //head, tail, and next changed?
5       if (oldHead == oldTail) {            //Queue empty or tail updated?
6           if (oldHeadNext == null) {        // Is queue empty?
7               return null;                 //Queue is empty, can't take
8           }
9       tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10      } else {                             // No need to deal with tail
11          if (head.compareAndSet(oldHead, oldHeadNext)) {
12              return oldHeadNext.item;
13          }
14      }
15  }
16  }
17  }
18  }
19  }
20  }

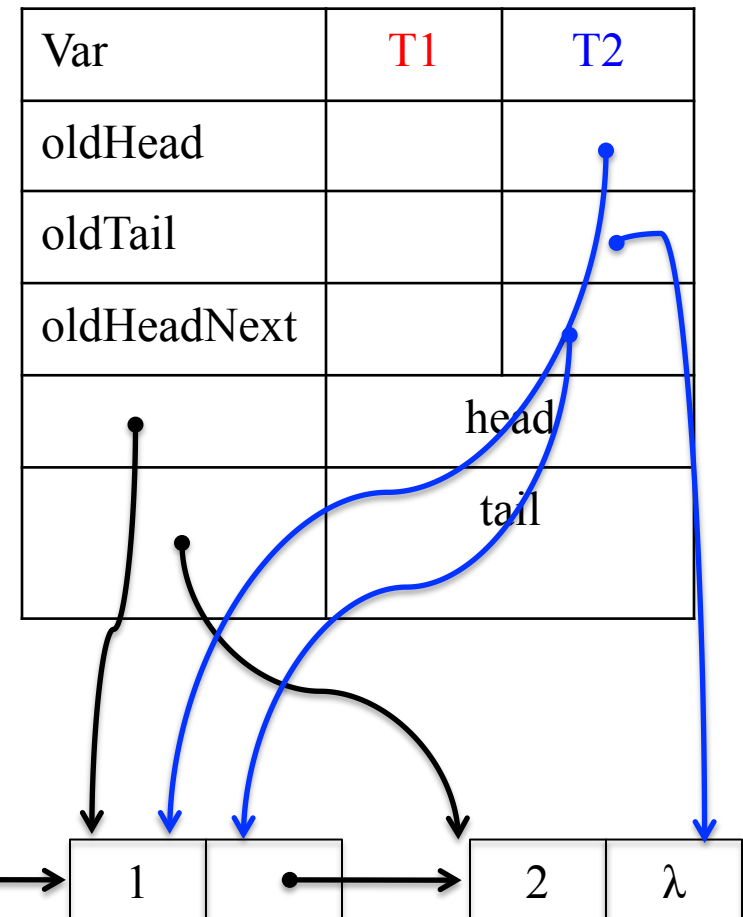
```



Trace 1: T2 continues

```

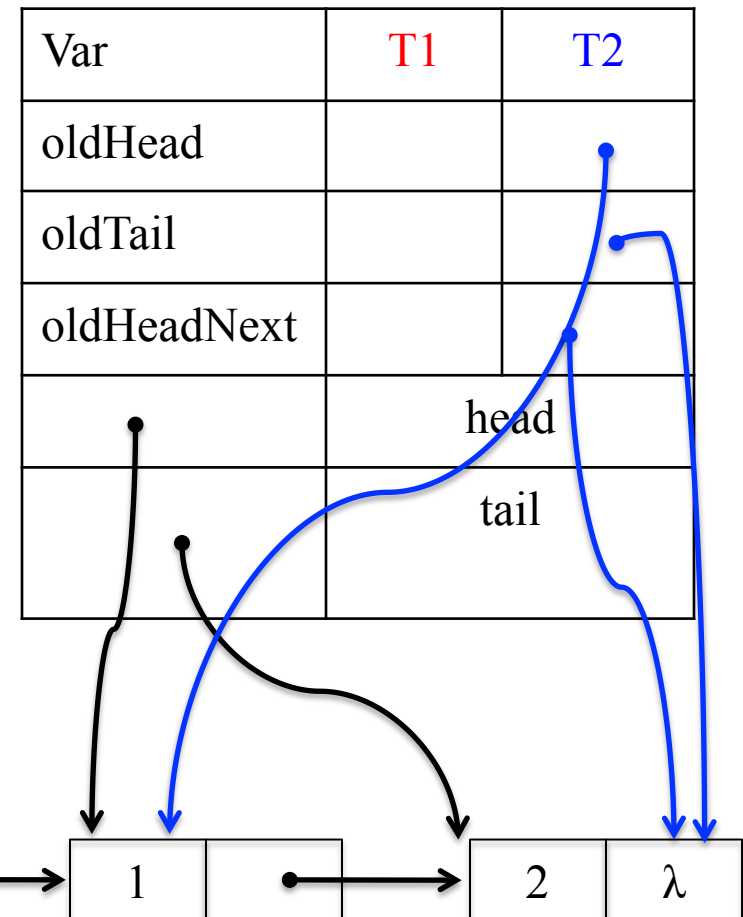
public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: T2 continues

```

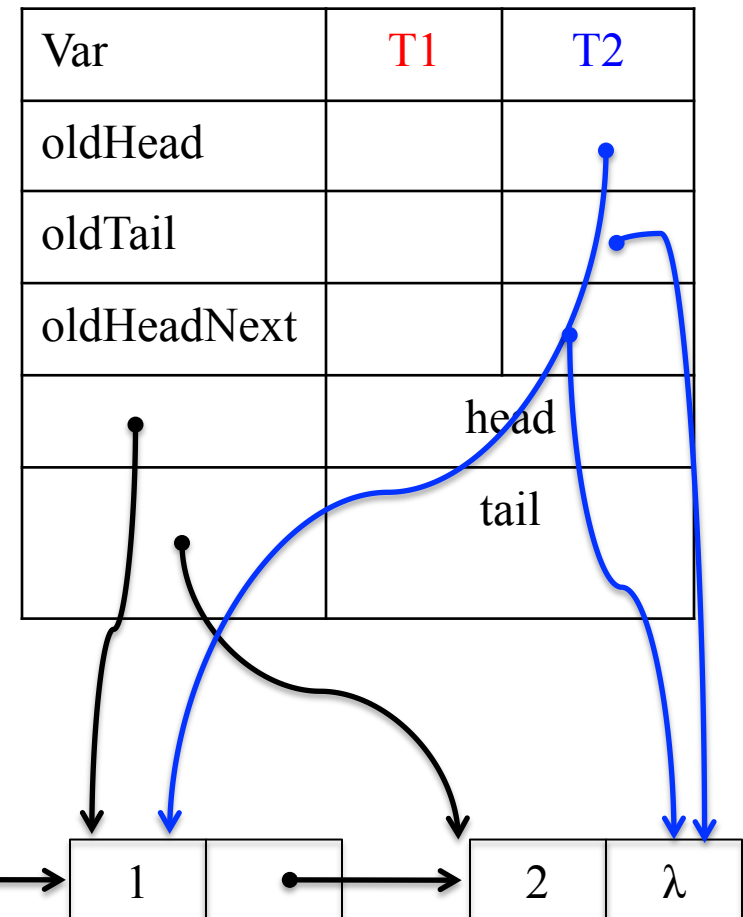
public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```



Trace 1: T2 continues

```

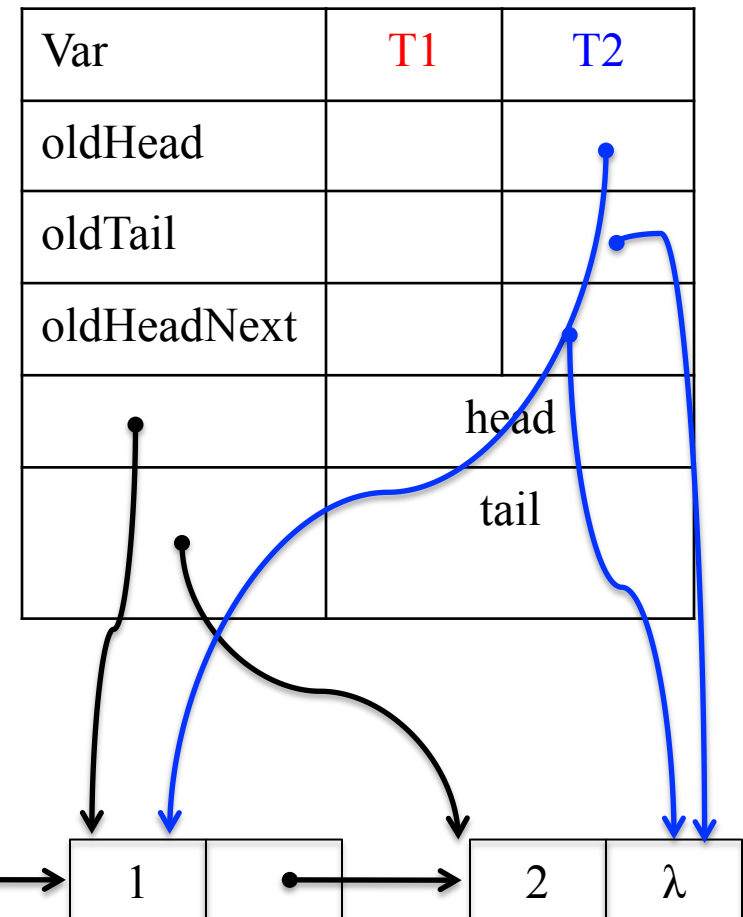
public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {           //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;               //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                         // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```



Trace 1: T2 continues

```

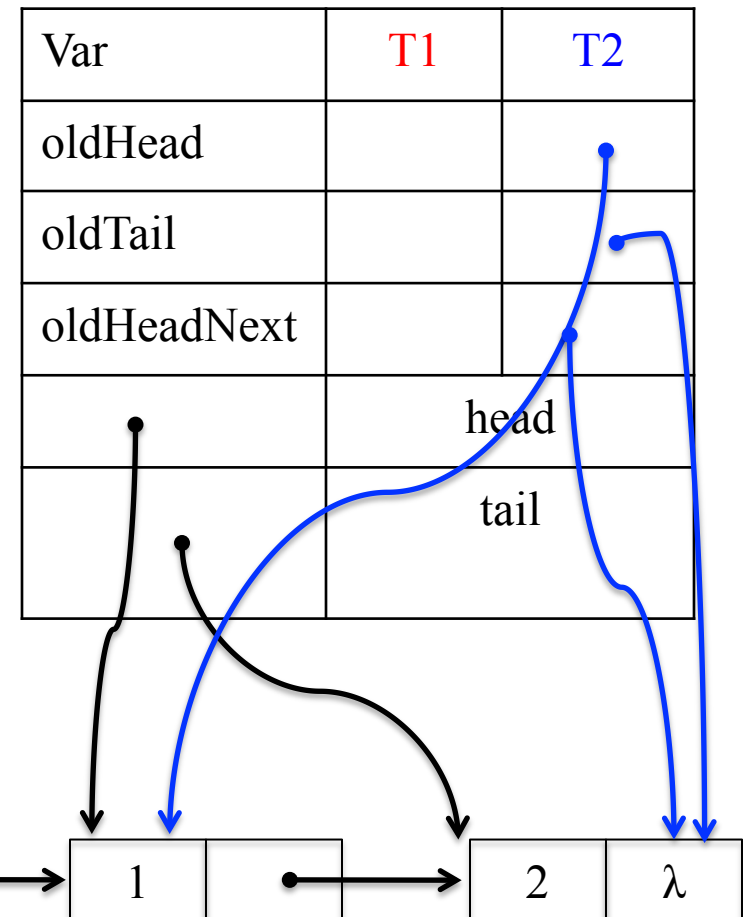
public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {           //head, tail, and next changed?
5       →   if (oldHead == oldTail) {           //Queue empty or tail updated?
6           if (oldHeadNext == null) { // Is queue empty?
7               return null;           //Queue is empty, can't take
            }
8       tail.compareAndSet(oldTail, oldHeadNext); // tail updating
9       } else {                             // No need to deal with tail
10          if (head.compareAndSet(oldHead, oldHeadNext)) {
              return oldHeadNext.item;
          }
      }
    }
}
    
```



Trace 1: T2 continues, CAS succeeds

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

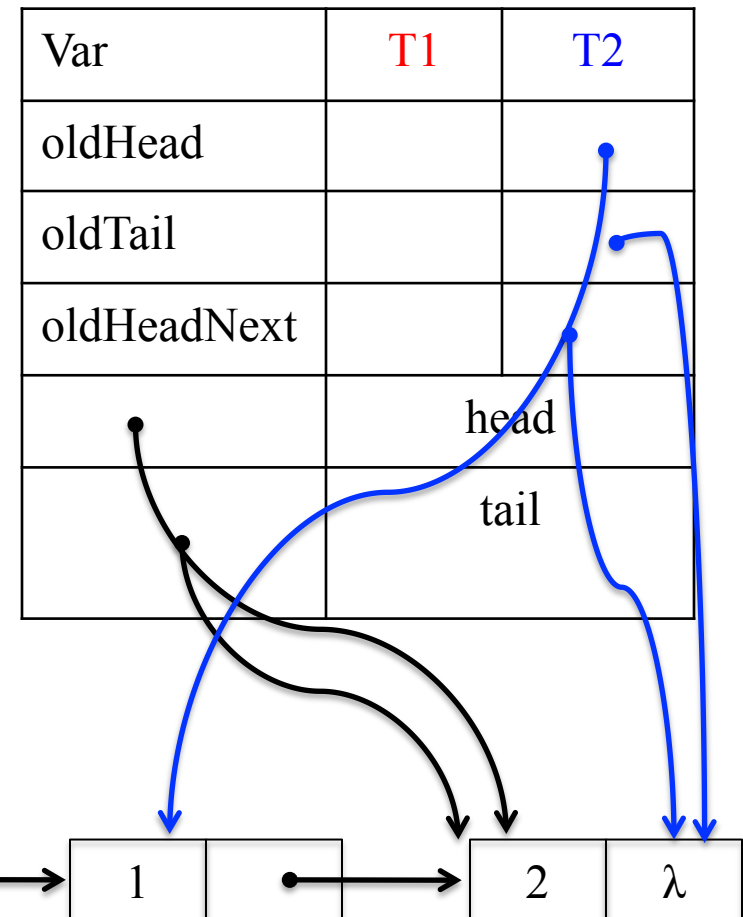


Trace 1: T2 continues, after CAS

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }

```



Trace 1: T2 exits

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }

```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
	head	
	tail	



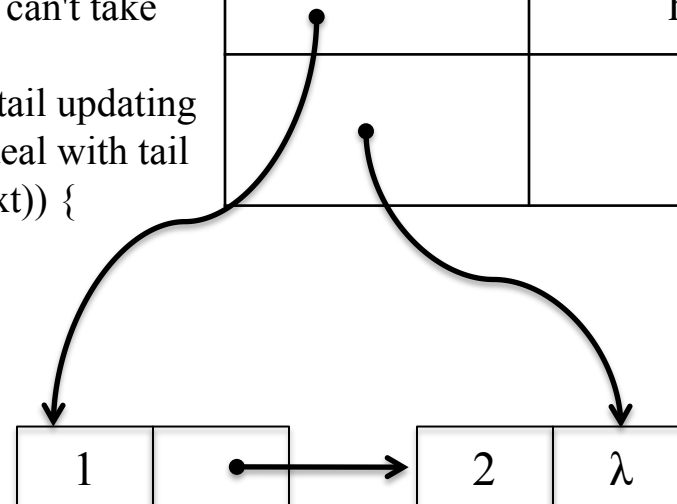
Trace 2

```

public E take() {
  for (;;) {
    1 Node<E> oldHead = head.get();           // current head
    2 Node<E> oldTail = tail.get();           // current tail
    3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    4 if (oldHead == head.get()) {           //head, tail, and next changed?
    5     if (oldHead == oldTail) {           //Queue empty or tail updated?
    6         if (oldHeadNext == null) { // Is queue empty?
    7             return null;               //Queue is empty, can't take
        }
    8     tail.compareAndSet(oldTail, oldHeadNext); // tail updating
    9     } else {                             // No need to deal with tail
    10        if (head.compareAndSet(oldHead, oldHeadNext)) {
            return oldHeadNext.item;
        }
    }
  }
}

```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
	head	
	tail	



Trace 2

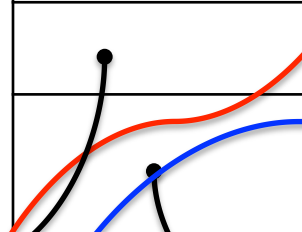
```

public E take() {
    for (;;) {
1 → Node<E> oldHead = head.get();           // current head
2   Node<E> oldTail = tail.get();           // current tail
3   Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4   if (oldHead == head.get()) {             //head, tail, and next changed?
5       if (oldHead == oldTail) {           //Queue empty or tail updated?
6           if (oldHeadNext == null) {      // Is queue empty?
7               return null;                //Queue is empty, can't take
            }
8       tail.compareAndSet(oldTail, oldHeadNext); // tail updating
9   } else {                                 // No need to deal with tail
10      if (head.compareAndSet(oldHead, oldHeadNext)) {
          return oldHeadNext.item;
      }
    }
  }
}

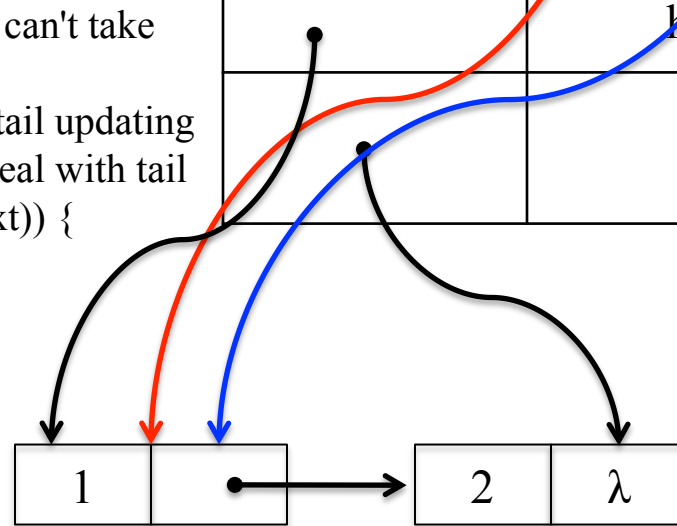
```

The diagram shows a single node in a queue, represented as a box divided into two parts. The left part contains the number '1'. A black arrow points from the 'oldHead' variable in the code to this box. A red arrow points from the 'oldTail' variable in the code to the same box, indicating that both pointers refer to the same node.

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		



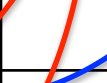
The diagram illustrates the state of pointers `head` and `tail` for two threads, T1 and T2. T1's pointer (red) is at `oldHead`, and T2's pointer (blue) is at `oldHead`. The diagram shows a linked list structure with nodes and pointers.

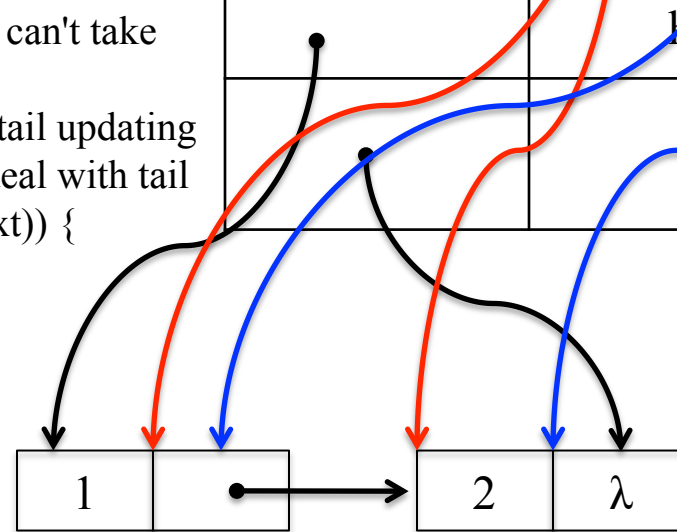


Trace 2

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

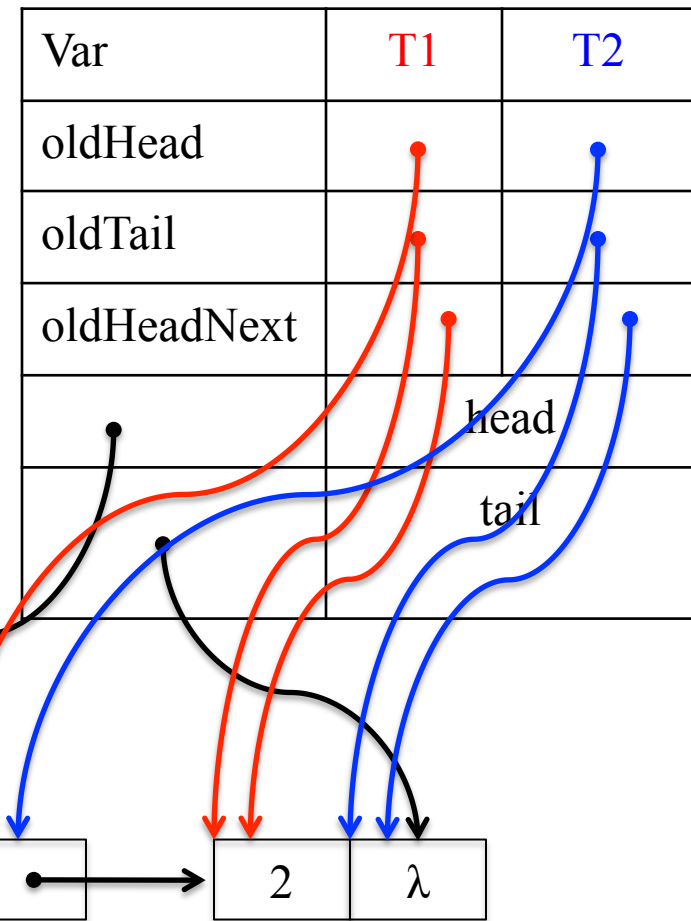
Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
head		
tail		



Trace 2

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

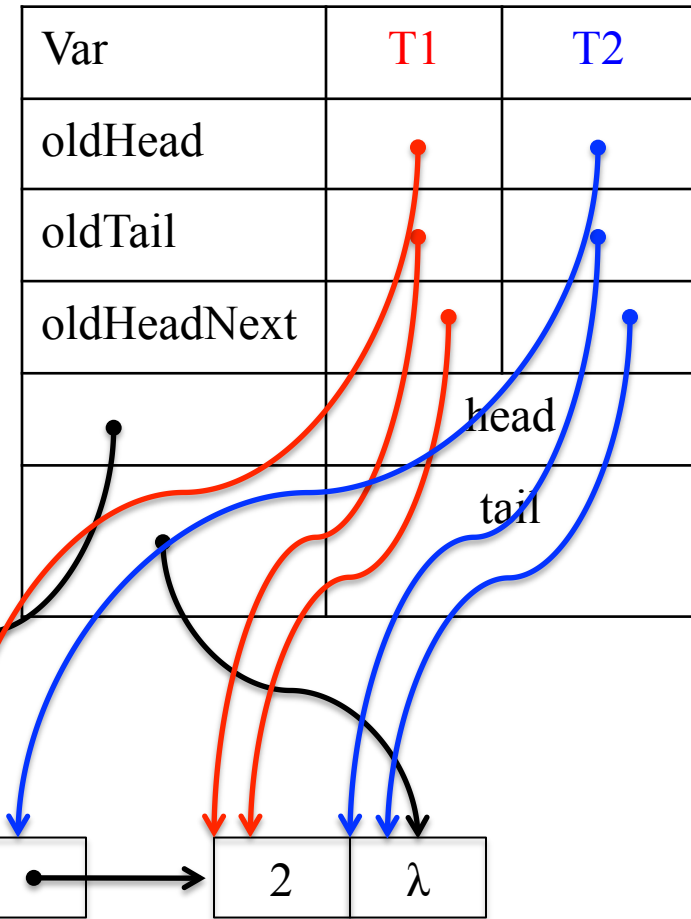


Trace 2

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4 →   if (oldHead == head.get()) {             //head, tail, and next changed?
5       if (oldHead == oldTail) {              //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7         return null;                        //Queue is empty, can't take
8        }
9      }
10     tail.compareAndSet(oldTail, oldHeadNext); // tail updating
11   } else {                                  // No need to deal with tail
12     if (head.compareAndSet(oldHead, oldHeadNext)) {
13       return oldHeadNext.item;
14     }
15   }
16 }
17 }
18 }
19 }
20 }

```

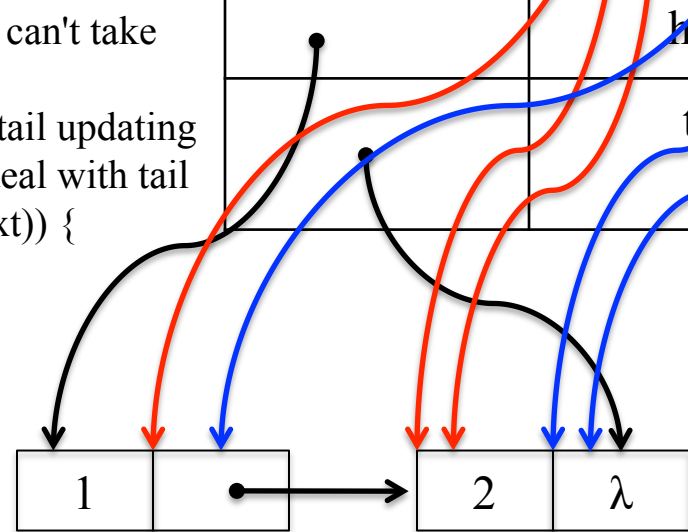


Trace 2

```

public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {             //head, tail, and next changed?
5       → if (oldHead == oldTail) {               //Queue empty or tail updated?
6       → if (oldHeadNext == null) { // Is queue empty?
7           return null;                       //Queue is empty, can't take
8       }
9       tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10      } else {                                // No need to deal with tail
11          if (head.compareAndSet(oldHead, oldHeadNext)) {
12              return oldHeadNext.item;
13          }
14      }
15  }
16  }
17  }
18  }
19  }
20  }
21  }
22  }
23  }
24  }
25  }
26  }
27  }
28  }
29  }
30  }
31  }
32  }
33  }
34  }
35  }
36  }
37  }
38  }
39  }
40  }
41  }
42  }
43  }
44  }
45  }
46  }
47  }
48  }
49  }
50  }
51  }
52  }
53  }
54  }
55  }
56  }
57  }
58  }
59  }
60  }
61  }
62  }
63  }
64  }
65  }
66  }
67  }
68  }
69  }
70  }
71  }
72  }
73  }
74  }
75  }
76  }
77  }
78  }
79  }
80  }
81  }
82  }
83  }
84  }
85  }
86  }
87  }
88  }
89  }
90  }
91  }
92  }
93  }
94  }
95  }
96  }
97  }
98  }
99  }
100 }
    
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
head		
tail		

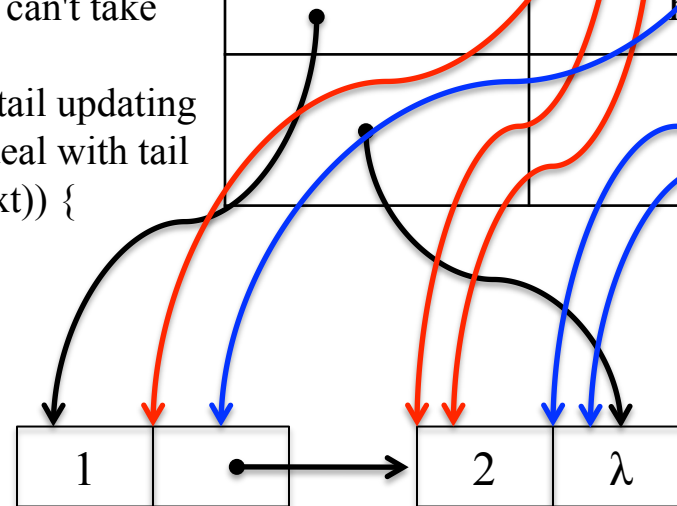


Trace 2: T1 wins, CAS succeeds

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {           //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;               //Queue is empty, can't take
8              }
9          tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10         } else {                             // No need to deal with tail
11             if (head.compareAndSet(oldHead, oldHeadNext)) {
12                 return oldHeadNext.item;
13             }
14         }
15     }
16 }
    
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
head		
tail		



Trace 2: after CAS

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
head		
tail		



Trace 2: T1 exits

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
		
		



Trace 2: T2 continues, CAS fails

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6        if (oldHeadNext == null) { // Is queue empty?
7          return null;                 //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                          // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
head		
tail		



Trace 2: T2 continues, retry

```

public E take() {
  for (;;) {
    1 → Node<E> oldHead = head.get();           // current head
    2   Node<E> oldTail = tail.get();           // current tail
    3   Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    4   if (oldHead == head.get()) {           //head, tail, and next changed?
    5     if (oldHead == oldTail) {           //Queue empty or tail updated?
    6       if (oldHeadNext == null) { // Is queue empty?
    7         return null;                 //Queue is empty, can't take
    8       }
    9       tail.compareAndSet(oldTail, oldHeadNext); // tail updating
   10      } else {                       // No need to deal with tail
        if (head.compareAndSet(oldHead, oldHeadNext)) {
          return oldHeadNext.item;
        }
      }
    }
  }
}

```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
		
		



Trace 2: T2 continues

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {           //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;               //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                         // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
		head
		tail



Trace 2: T2 continues

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

Var	T1	T2
oldHead		•
oldTail		•
oldHeadNext		λ
	head	
	tail	



Trace 2: T2 continues

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {           //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;               //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                         // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

Var	T1	T2
oldHead		•
oldTail		•
oldHeadNext		λ
	head	
	tail	



Trace 2: T2 continues

```

public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {             //head, tail, and next changed?
5       →   if (oldHead == oldTail) {             //Queue empty or tail updated?
6           if (oldHeadNext == null) { // Is queue empty?
7               return null;                //Queue is empty, can't take
            }
8           tail.compareAndSet(oldTail, oldHeadNext); // tail updating
        } else {                             // No need to deal with tail
9           if (head.compareAndSet(oldHead, oldHeadNext)) {
10              return oldHeadNext.item;
            }
        }
    }
}
    
```

Var	T1	T2
oldHead		•
oldTail		•
oldHeadNext		λ
	head	
	tail	



Trace 2: T2 continues

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6      →  if (oldHeadNext == null) { // Is queue empty?
7        return null;                     //Queue is empty, can't take
8      }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                             // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```

Var	T1	T2
oldHead		•
oldTail		•
oldHeadNext		λ
	head	
	tail	



Trace 2: T2 exits

```

public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {             //head, tail, and next changed?
5           if (oldHead == oldTail) {             //Queue empty or tail updated?
6               if (oldHeadNext == null) { // Is queue empty?
7                   return null;                 //Queue is empty, can't take
8               }
9               tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10              } else {                          // No need to deal with tail
11                  if (head.compareAndSet(oldHead, oldHeadNext)) {
12                      return oldHeadNext.item;
13                  }
14              }
15          }
16      }
17  }
18  }
19  }
20  }
    
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
	head	
	tail	



Execution Traces

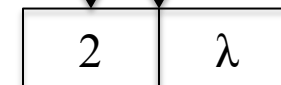
- 1 thread calls take()
- 1 thread calls put()

Trace 1

```

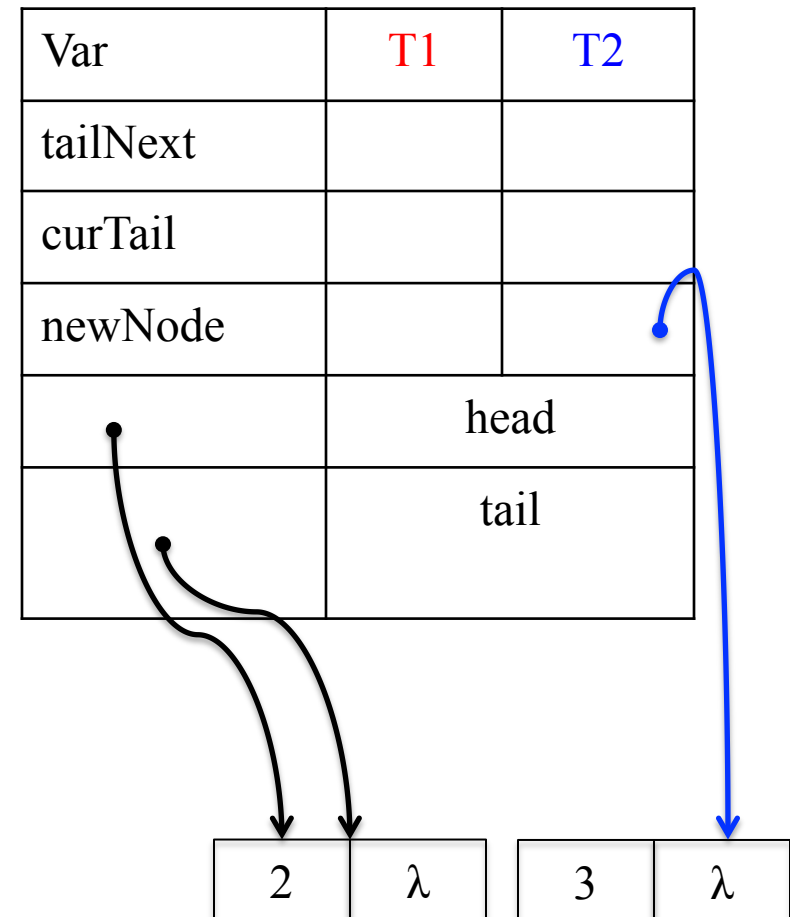
public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                 //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

Var	T1	T2
oldHead		
oldTail		
oldHeadNext		
	head	
	tail	



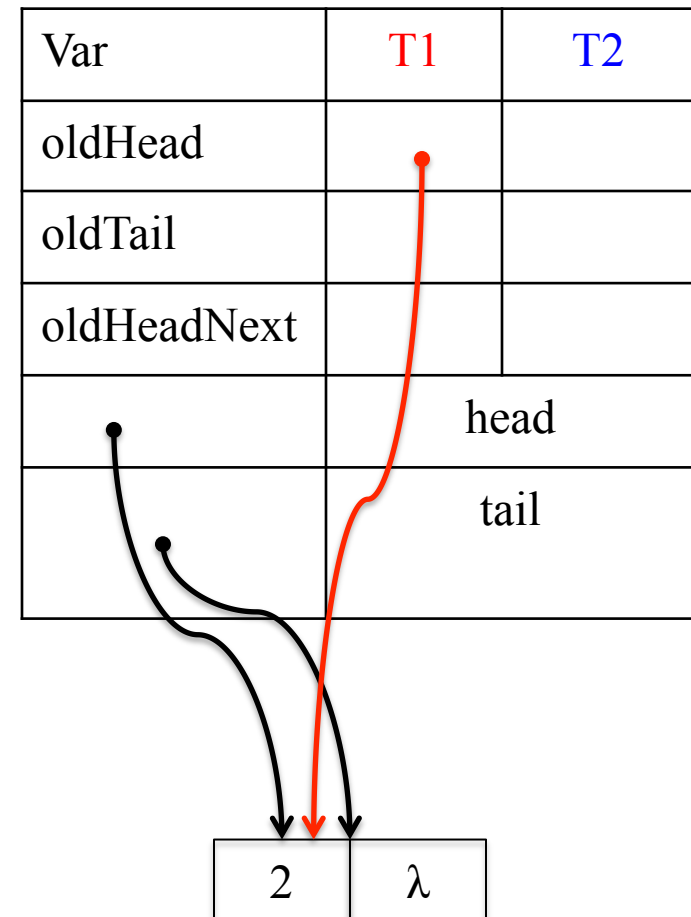
Trace 1

```
public boolean put(E item) {  
1  Node<E> newNode = new Node<E>(item, null);  
2  while (true) {  
3    Node<E> curTail = tail.get();  
4    Node<E> tailNext = curTail.next.get();  
5    if (curTail == tail.get()) { // did tail change?  
6      if (tailNext != null) { // Queue in int. state, advance tail  
7        tail.compareAndSet(curTail, tailNext);  
8      } else { // In quiescent state, try inserting new node  
9        if (curTail.next.compareAndSet(null, newNode)) {  
10         // Insertion succeeded, try advancing tail  
11         tail.compareAndSet(curTail, newNode);  
12         // will fail if tail already moved  
13         return true;  
14       }  
15     }  
16   }  
17 }
```



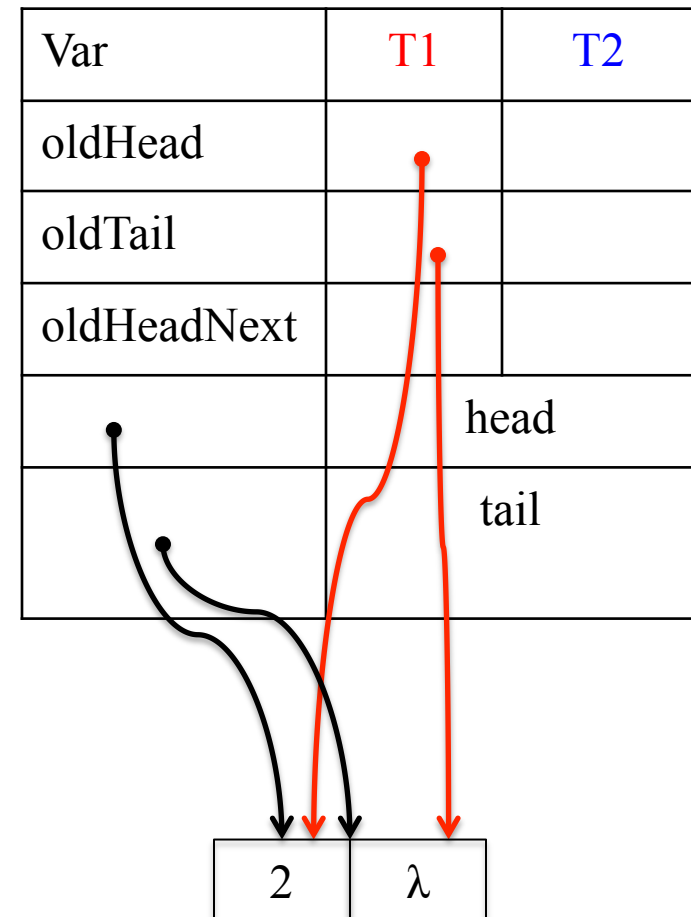
Trace 1: T1 continues

```
public E take() {  
    for (;;) {  
1 → Node<E> oldHead = head.get();           // current head  
2     Node<E> oldTail = tail.get();          // current tail  
3     Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4     if (oldHead == head.get()) {           //head, tail, and next changed?  
5         if (oldHead == oldTail) {          //Queue empty or tail updated?  
6             if (oldHeadNext == null) { // Is queue empty?  
7                 return null;              //Queue is empty, can't take  
8             }  
9             tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
10            } else {                          // No need to deal with tail  
11                if (head.compareAndSet(oldHead, oldHeadNext)) {  
12                    return oldHeadNext.item;  
13                }  
14            }  
15        }  
16    }  
17 }  
18 }
```



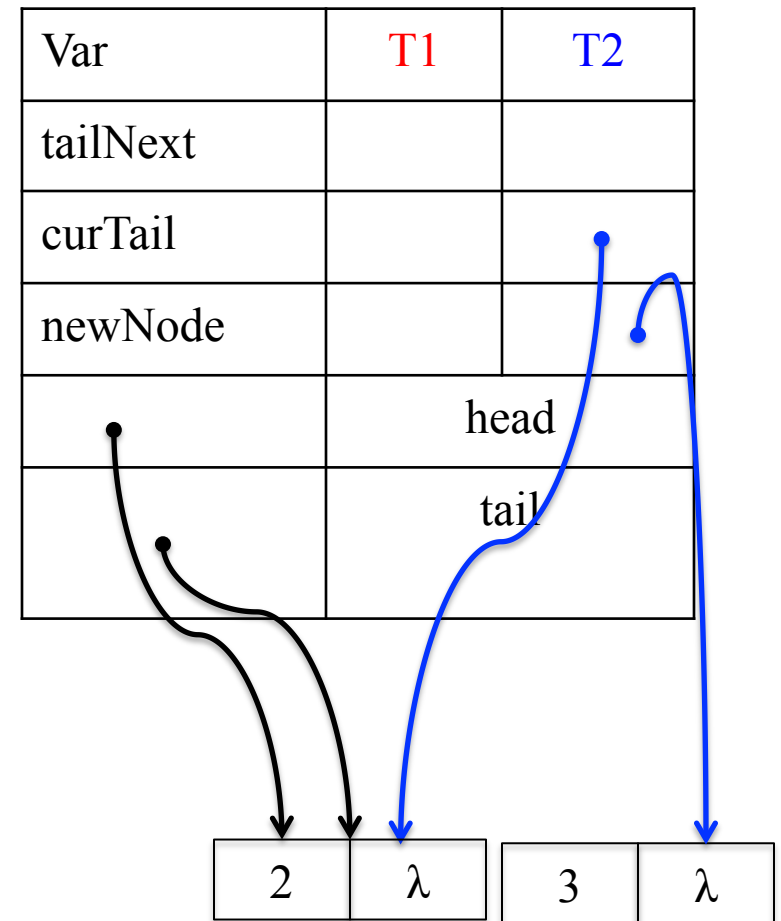
Trace 1: T1 continues

```
public E take() {  
    for (;;) {  
1      Node<E> oldHead = head.get();           // current head  
2      Node<E> oldTail = tail.get();           // current tail  
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4      if (oldHead == head.get()) {           //head, tail, and next changed?  
5          if (oldHead == oldTail) {           //Queue empty or tail updated?  
6              if (oldHeadNext == null) { // Is queue empty?  
7                  return null;               //Queue is empty, can't take  
8              }  
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
10             } else {                          // No need to deal with tail  
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {  
12                     return oldHeadNext.item;  
13                 }  
14             }  
15         }  
16     }  
17 }
```



Trace 1: T2 continues

```
public boolean put(E item) {  
1   Node<E> newNode = new Node<E>(item, null);  
2   while (true) {  
3       Node<E> curTail = tail.get();  
4       Node<E> tailNext = curTail.next.get();  
5       if (curTail == tail.get()) { // did tail change?  
6           if (tailNext != null) { // Queue in int. state, advance tail  
7               tail.compareAndSet(curTail, tailNext);  
8           } else { // In quiescent state, try inserting new node  
9               if (curTail.next.compareAndSet(null, newNode)) {  
10                  // Insertion succeeded, try advancing tail  
11                  tail.compareAndSet(curTail, newNode);  
12                  // will fail if tail already moved  
13                  return true;  
14              }  
15          }  
16      }  
17  }
```

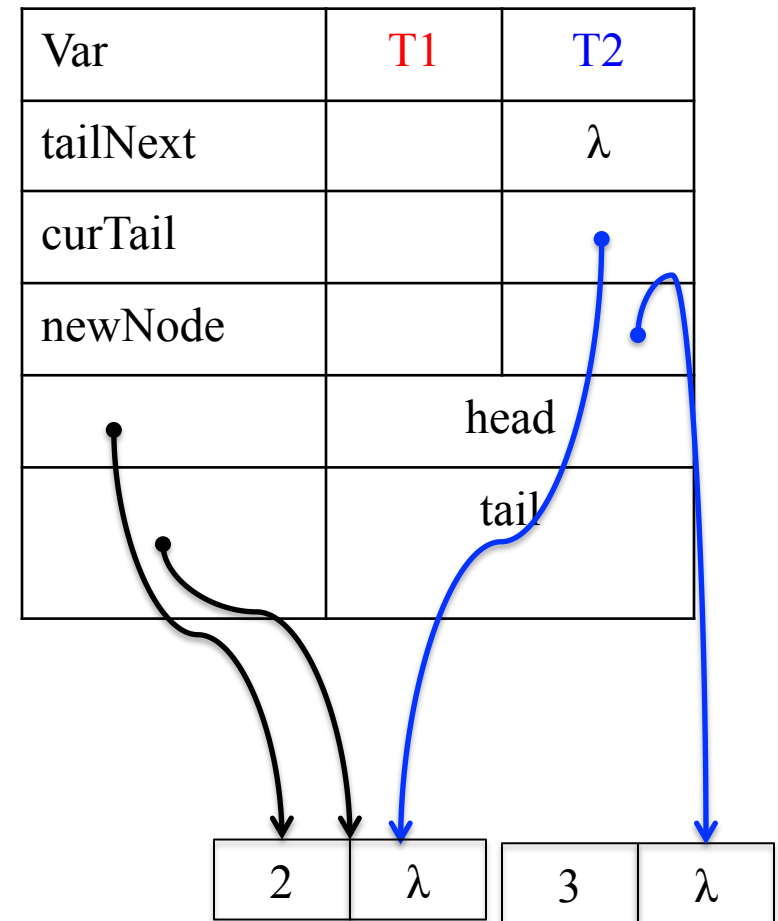


Trace 1: T2 continues

```

1  public boolean put(E item) {
2      Node<E> newNode = new Node<E>(item, null);
3      while (true) {
4          Node<E> curTail = tail.get();
5          Node<E> tailNext = curTail.next.get();
6          if (curTail == tail.get()) { // did tail change?
7              if (tailNext != null) { // Queue in int. state, advance tail
8                  tail.compareAndSet(curTail, tailNext);
9              }
10             else { // In quiescent state, try inserting new node
11                 if (curTail.next.compareAndSet(null, newNode)) {
12                     // Insertion succeeded, try advancing tail
13                     tail.compareAndSet(curTail, newNode);
14                     // will fail if tail already moved
15                     return true;
16                 }
17             }
18         }
19     }
20 }

```

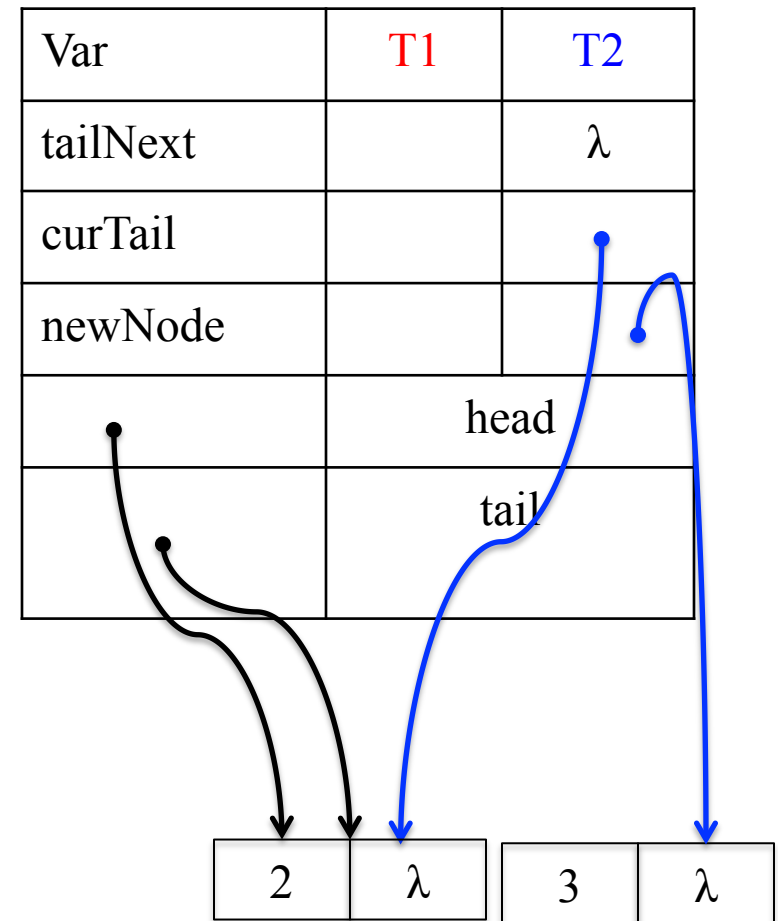


Trace 1: T2 continues

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13                  return true;
14              }
15          }
16      }
17  }
18  }
19  }

```

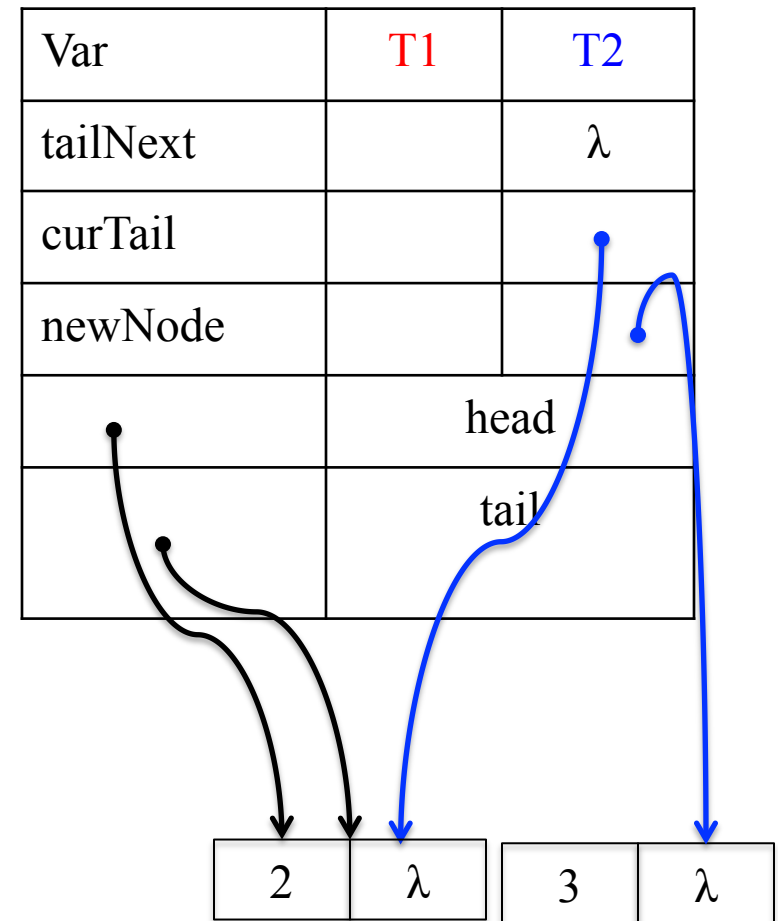


Trace 1: T2 continues

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           } else { // In quiescent state, try inserting new node
9               if (curTail.next.compareAndSet(null, newNode)) {
10                  // Insertion succeeded, try advancing tail
11                  tail.compareAndSet(curTail, newNode);
12                  // will fail if tail already moved
13                  return true;
14              }
15          }
16      }
17  }
18  }
19  }

```



Trace 1: CAS succeeds

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           }
9           else { // In quiescent state, try inserting new node
10              if (curTail.next.compareAndSet(null, newNode)) {
11                  // Insertion succeeded, try advancing tail
12                  tail.compareAndSet(curTail, newNode);
13                  // will fail if tail already moved
14              }
15          }
16      }
17      return true;
18  }
19  }
20  }
21  }
}

```

Var	T1	T2
tailNext		λ
curTail		
newNode		
	head	
	tail	

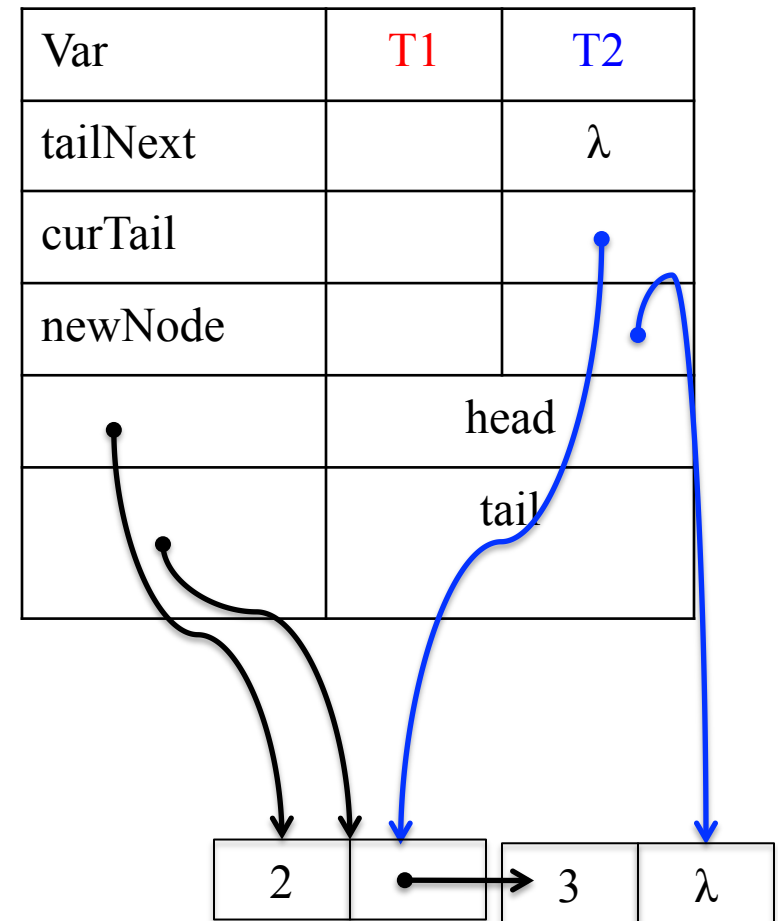
The diagram illustrates the state of two threads, T1 and T2, and their pointers to nodes in a linked list. The table above shows the current values of variables: tailNext is λ , curTail is null, and newNode is null. T1's head points to node 2, and its tail points to node λ . T2's head points to node 3, and its tail points to node λ . The nodes are represented as boxes with values 2, λ , 3, and λ .

Trace 1: after CAS

```

public boolean put(E item) {
1   Node<E> newNode = new Node<E>(item, null);
2   while (true) {
3       Node<E> curTail = tail.get();
4       Node<E> tailNext = curTail.next.get();
5       if (curTail == tail.get()) { // did tail change?
6           if (tailNext != null) { // Queue in int. state, advance tail
7               tail.compareAndSet(curTail, tailNext);
8           }
9           else { // In quiescent state, try inserting new node
10              if (curTail.next.compareAndSet(null, newNode)) {
11                  // Insertion succeeded, try advancing tail
12                  tail.compareAndSet(curTail, newNode);
13                  // will fail if tail already moved
14                  return true;
15              }
16          }
17      }
18  }
19  }
20  }

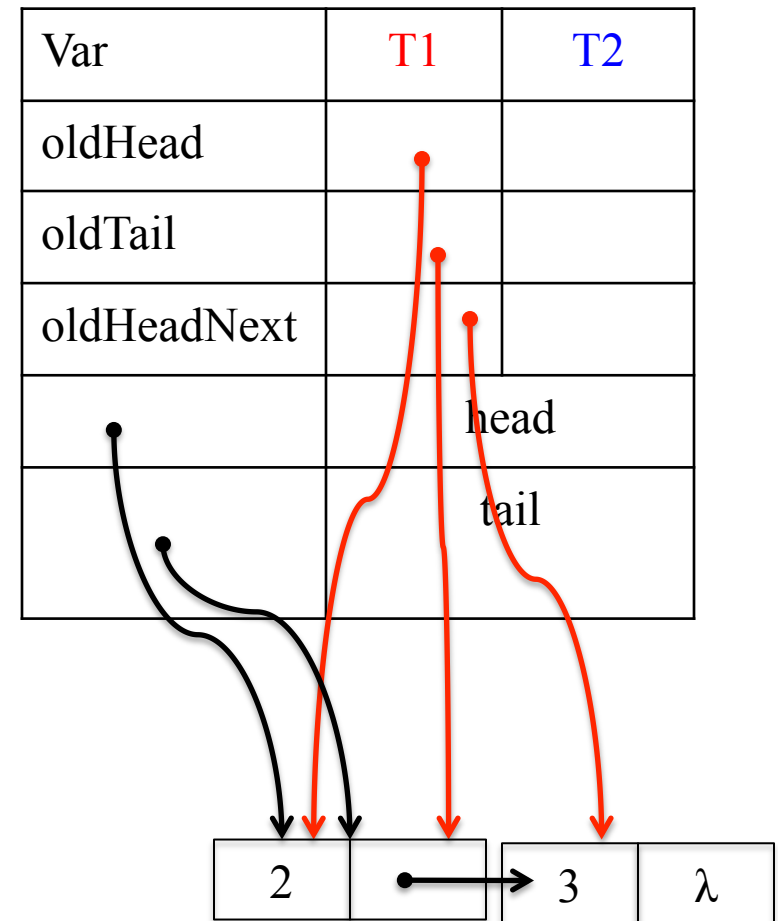
```



Trace 1: T1 continues

```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {         //Queue empty or tail updated?
6              if (oldHeadNext == null) {    // Is queue empty?
7                  return null;              //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                       // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```

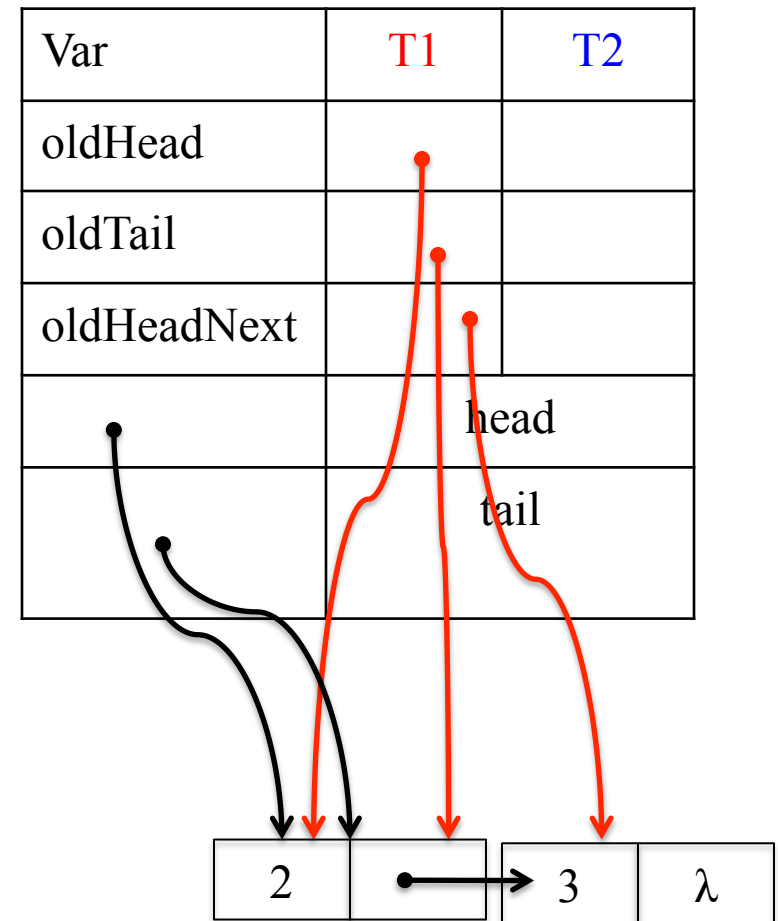


Trace 1: T1 continues

```

public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4 →    if (oldHead == head.get()) {             //head, tail, and next changed?
5         if (oldHead == oldTail) {             //Queue empty or tail updated?
6             if (oldHeadNext == null) { // Is queue empty?
7                 return null;                //Queue is empty, can't take
8             }
9             tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10          } else {                            // No need to deal with tail
11              if (head.compareAndSet(oldHead, oldHeadNext)) {
12                  return oldHeadNext.item;
13              }
14          }
15      }
16  }
17  }
18  }
19  }
20  }

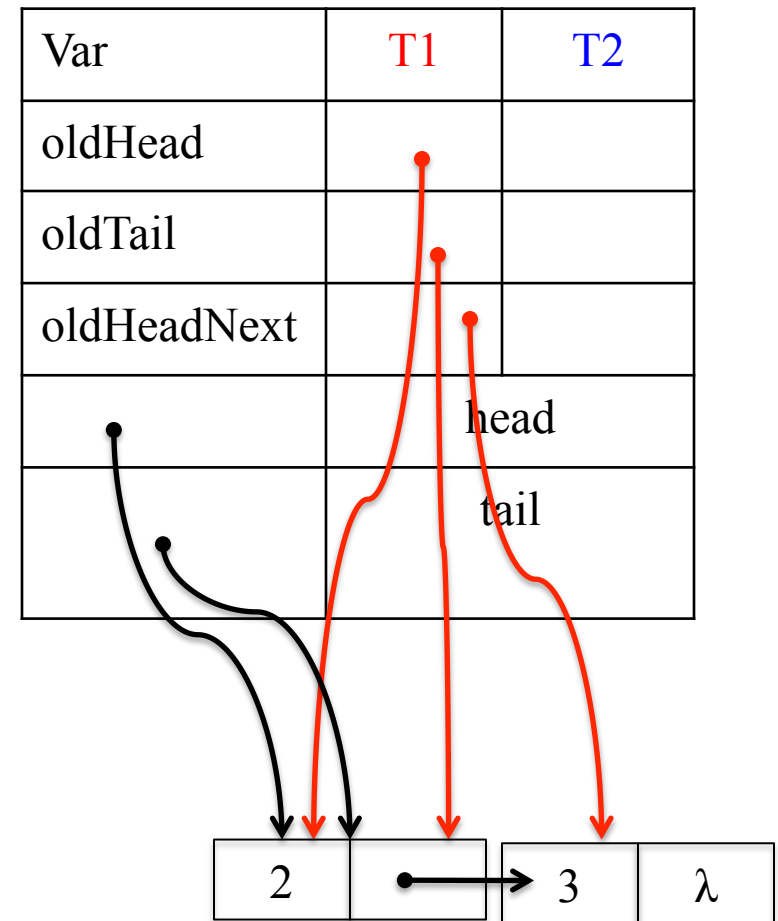
```



Trace 1: T1 continues

```

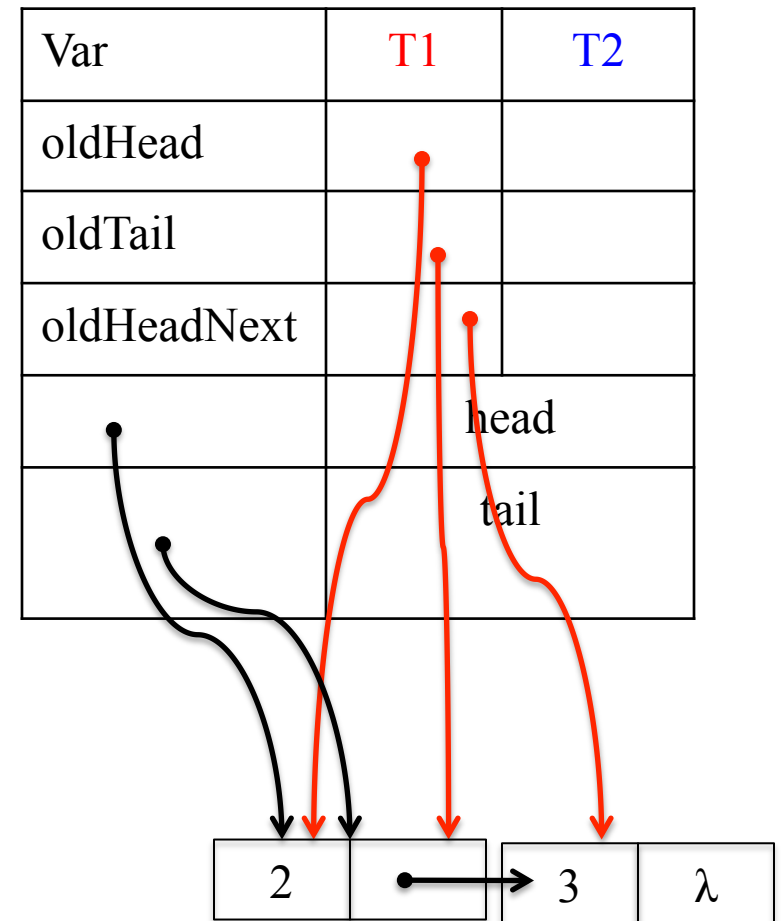
public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {           //head, tail, and next changed?
5       →   if (oldHead == oldTail) {           //Queue empty or tail updated?
6           if (oldHeadNext == null) { // Is queue empty?
7               return null;           //Queue is empty, can't take
            }
8           tail.compareAndSet(oldTail, oldHeadNext); // tail updating
9       } else {                             // No need to deal with tail
10          if (head.compareAndSet(oldHead, oldHeadNext)) {
              return oldHeadNext.item;
          }
      }
    }
}
    
```



Trace 1: T1 continues

```

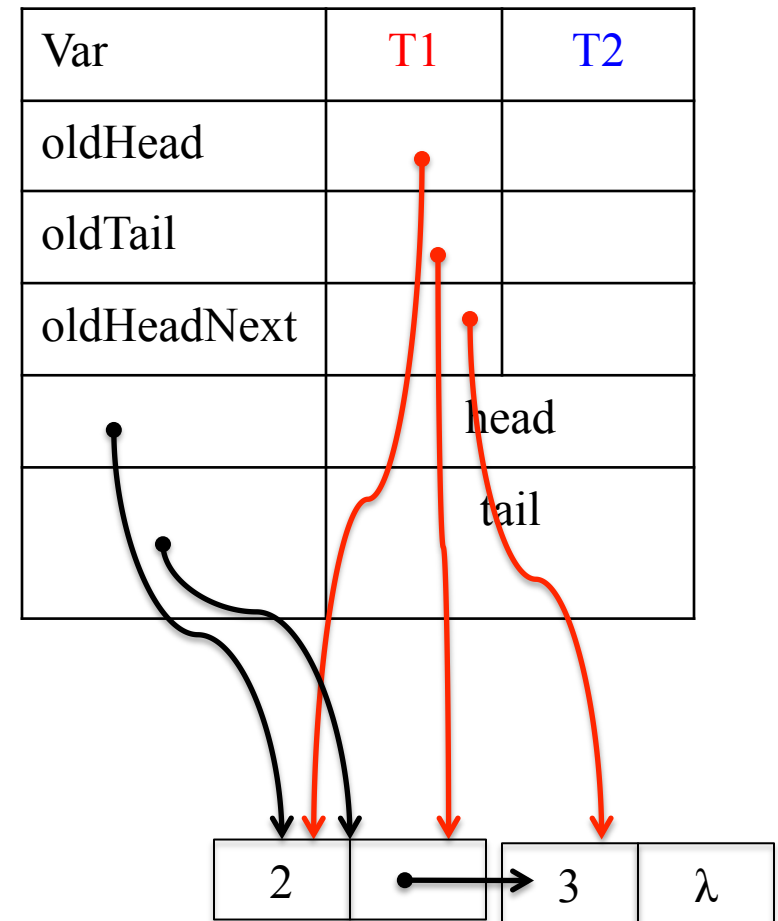
public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {           //head, tail, and next changed?
5      if (oldHead == oldTail) {           //Queue empty or tail updated?
6      →  if (oldHeadNext == null) { // Is queue empty?
7        return null;                     //Queue is empty, can't take
8      }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                             // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: CAS succeeds

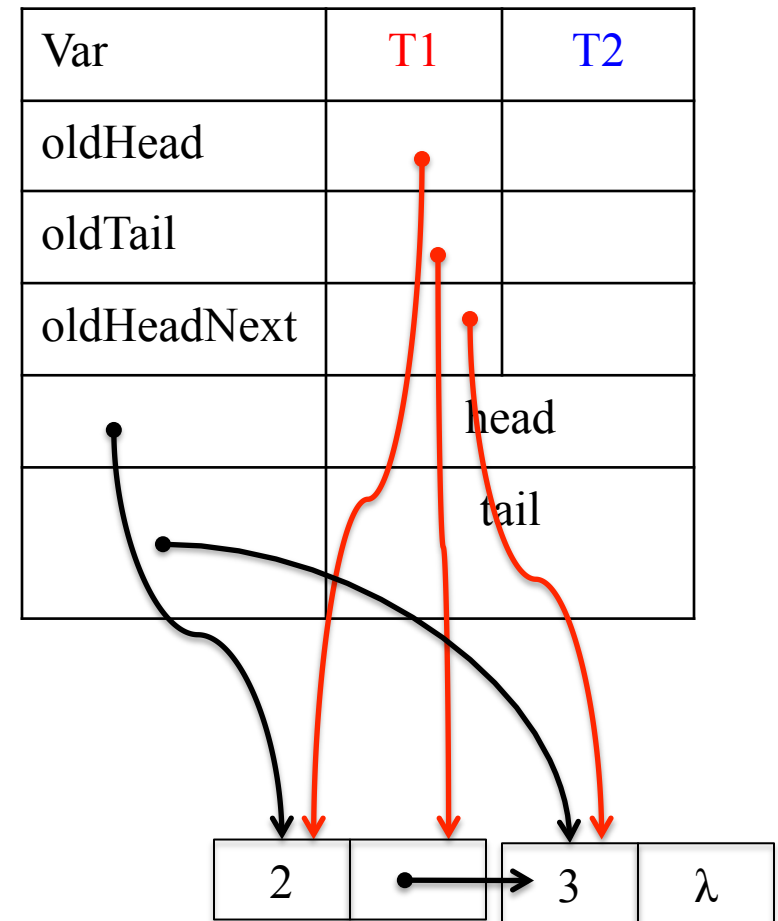
```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {             //head, tail, and next changed?
5          if (oldHead == oldTail) {             //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;                //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                          // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```



Trace 1: after CAS

```
public E take() {  
    for (;;) {  
1      Node<E> oldHead = head.get();           // current head  
2      Node<E> oldTail = tail.get();           // current tail  
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4      if (oldHead == head.get()) {           //head, tail, and next changed?  
5          if (oldHead == oldTail) {           //Queue empty or tail updated?  
6              if (oldHeadNext == null) { // Is queue empty?  
7                  return null;             //Queue is empty, can't take  
            }  
8      → tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
        } else {                             // No need to deal with tail  
9          if (head.compareAndSet(oldHead, oldHeadNext)) {  
10             return oldHeadNext.item;  
        }  
    }  
} } }
```

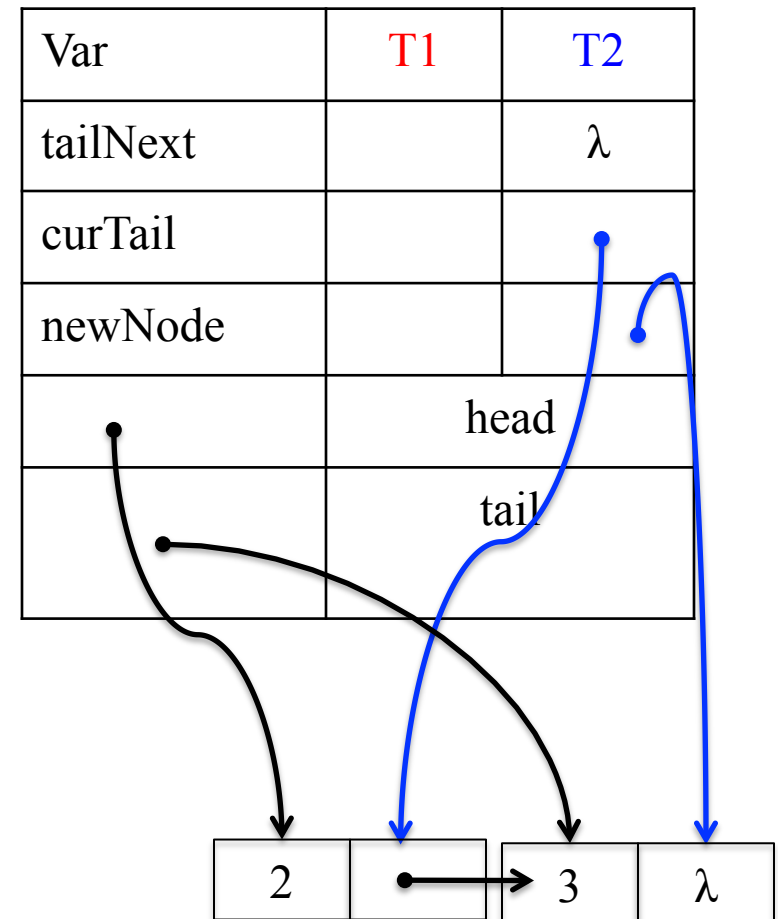


Trace 1: T2 continues & CAS fails

```

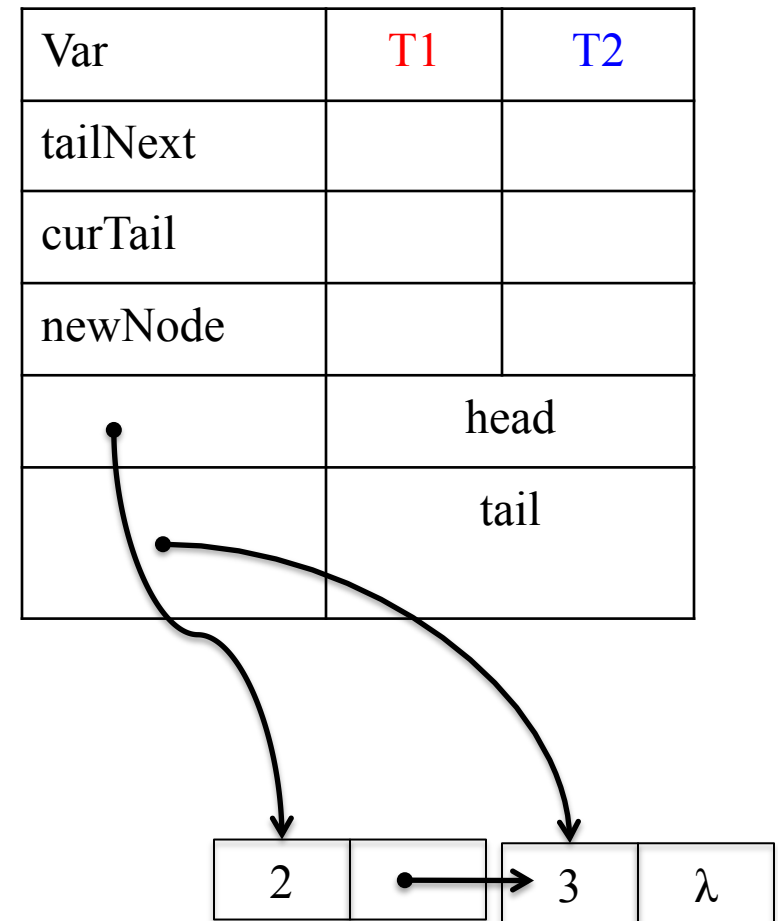
public boolean put(E item) {
1  Node<E> newNode = new Node<E>(item, null);
2  while (true) {
3      Node<E> curTail = tail.get();
4      Node<E> tailNext = curTail.next.get();
5      if (curTail == tail.get()) { // did tail change?
6          if (tailNext != null) { // Queue in int. state, advance tail
7              tail.compareAndSet(curTail, tailNext);
8          } else { // In quiescent state, try inserting new node
9              if (curTail.next.compareAndSet(null, newNode)) {
10                 // Insertion succeeded, try advancing tail
11                 tail.compareAndSet(curTail, newNode);
12                 // will fail if tail already moved
13                 return true;
14             }
15         }
16     }
17 }

```



Trace 1: T2 exits

```
public boolean put(E item) {  
1   Node<E> newNode = new Node<E>(item, null);  
2   while (true) {  
3       Node<E> curTail = tail.get();  
4       Node<E> tailNext = curTail.next.get();  
5       if (curTail == tail.get()) { // did tail change?  
6           if (tailNext != null) { // Queue in int. state, advance tail  
7               tail.compareAndSet(curTail, tailNext);  
8           } else { // In quiescent state, try inserting new node  
9               if (curTail.next.compareAndSet(null, newNode)) {  
10                  // Insertion succeeded, try advancing tail  
11                  tail.compareAndSet(curTail, newNode);  
                  // will fail if tail already moved  
12              }  
13          }  
14      }  
15      return true;  
16  }  
17 }
```

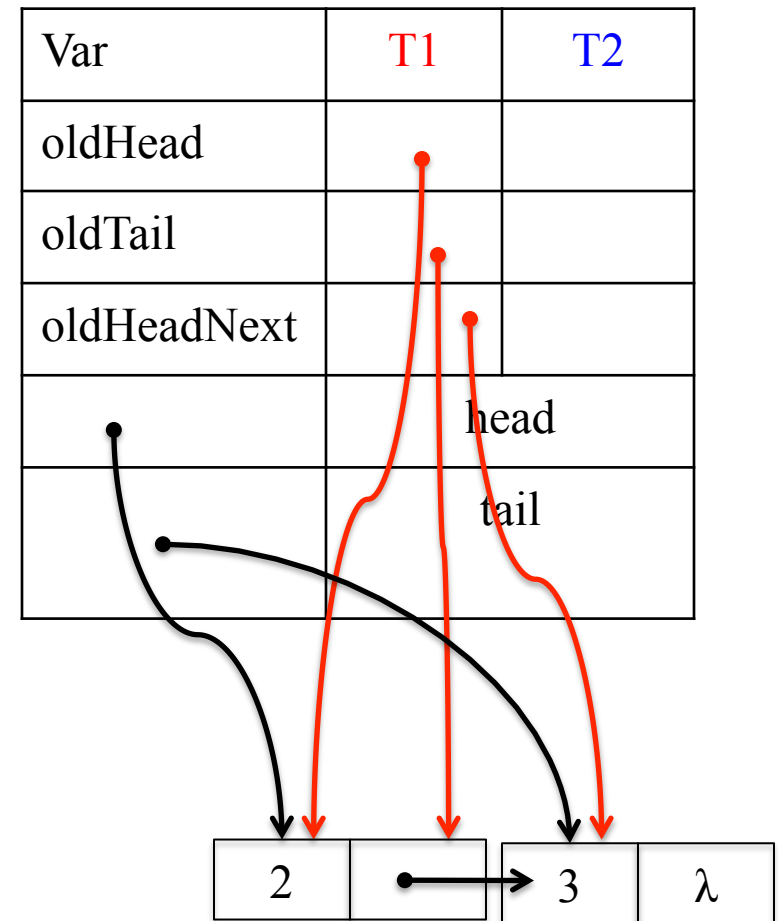


Trace 1: T1 continues

```

public E take() {
  for (;;) {
    1 Node<E> oldHead = head.get();           // current head
    2 Node<E> oldTail = tail.get();           // current tail
    3 Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
    4 if (oldHead == head.get()) {           //head, tail, and next changed?
    5     if (oldHead == oldTail) {           //Queue empty or tail updated?
    6         if (oldHeadNext == null) { // Is queue empty?
    7             return null;               //Queue is empty, can't take
        }
    8     tail.compareAndSet(oldTail, oldHeadNext); // tail updating
    9     } else {                             // No need to deal with tail
    10        if (head.compareAndSet(oldHead, oldHeadNext)) {
            return oldHeadNext.item;
        }
    }
  }
}

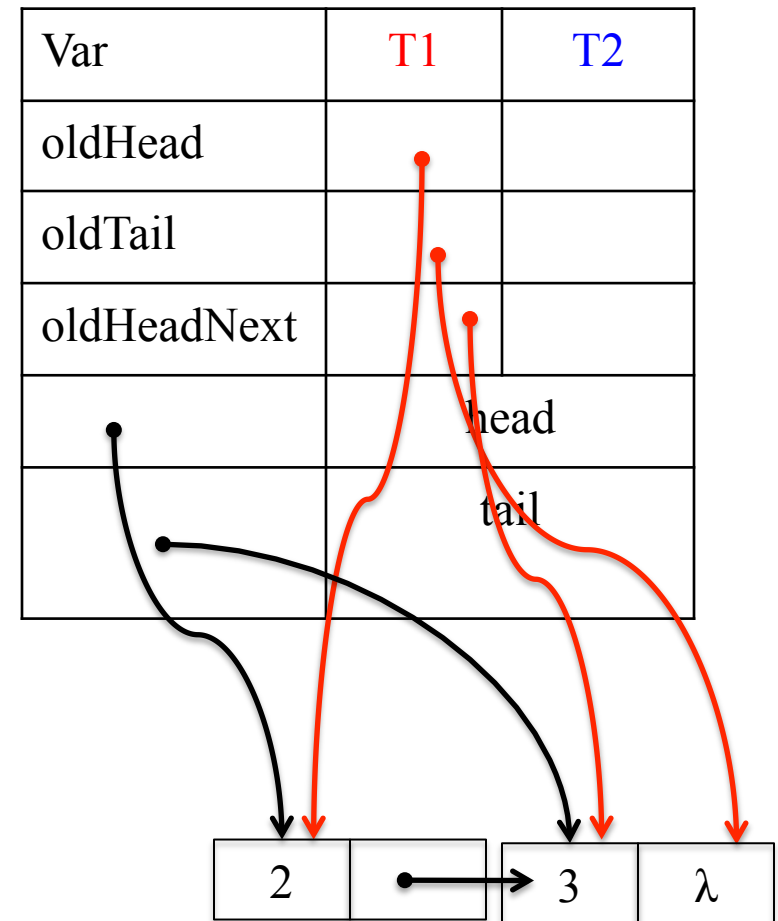
```



Trace 1: T1 continues

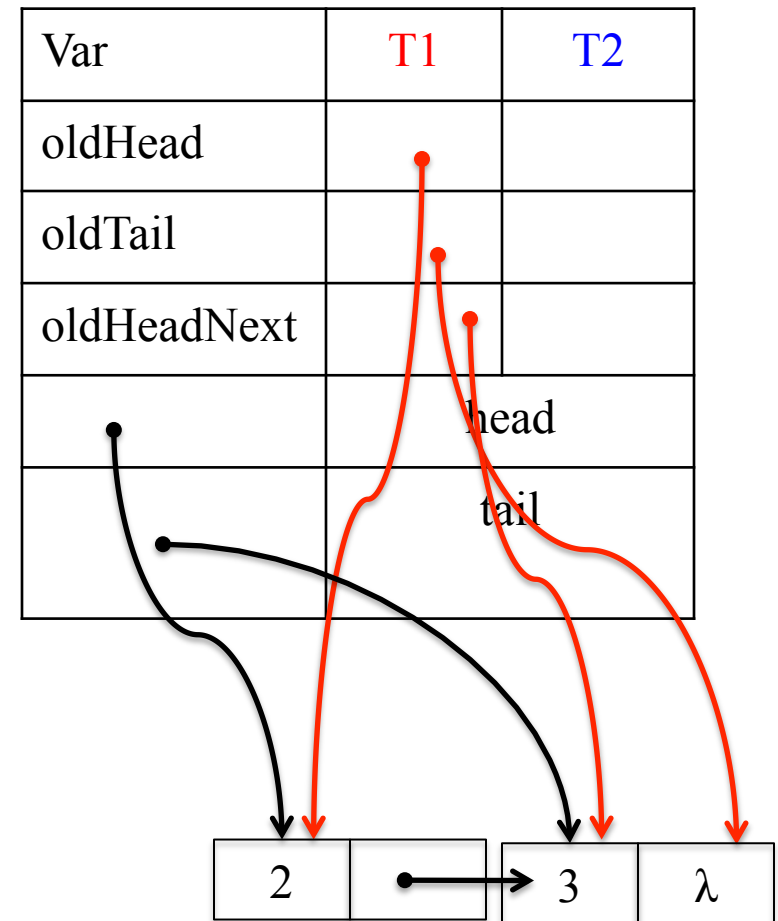
```

public E take() {
    for (;;) {
1      Node<E> oldHead = head.get();           // current head
2      Node<E> oldTail = tail.get();           // current tail
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4      if (oldHead == head.get()) {           //head, tail, and next changed?
5          if (oldHead == oldTail) {           //Queue empty or tail updated?
6              if (oldHeadNext == null) { // Is queue empty?
7                  return null;               //Queue is empty, can't take
8              }
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10             } else {                         // No need to deal with tail
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {
12                     return oldHeadNext.item;
13                 }
14             }
15         }
16     }
17 }
    
```



Trace 1: T1 continues

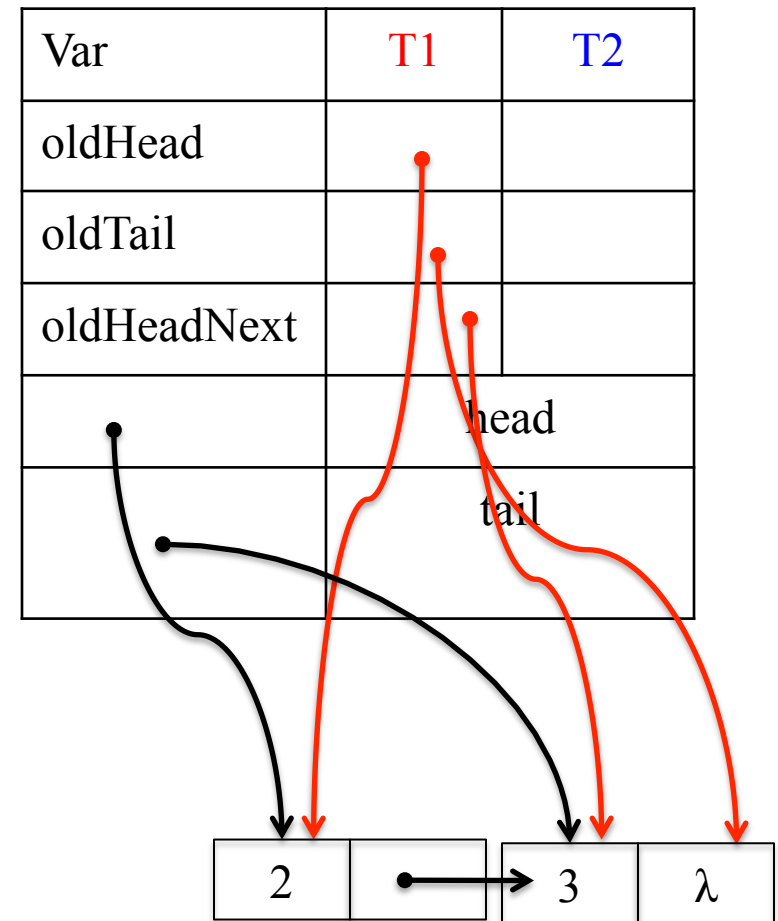
```
public E take() {  
    for (;;) {  
1      Node<E> oldHead = head.get();           // current head  
2      Node<E> oldTail = tail.get();           // current tail  
3 →   Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4      if (oldHead == head.get()) {           //head, tail, and next changed?  
5          if (oldHead == oldTail) {         //Queue empty or tail updated?  
6              if (oldHeadNext == null) { // Is queue empty?  
7                  return null;             //Queue is empty, can't take  
8              }  
9              tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
10             } else {                       // No need to deal with tail  
11                 if (head.compareAndSet(oldHead, oldHeadNext)) {  
12                     return oldHeadNext.item;  
13                 }  
14             }  
15         }  
16     }  
17 }
```



Trace 1: T1 continues

```

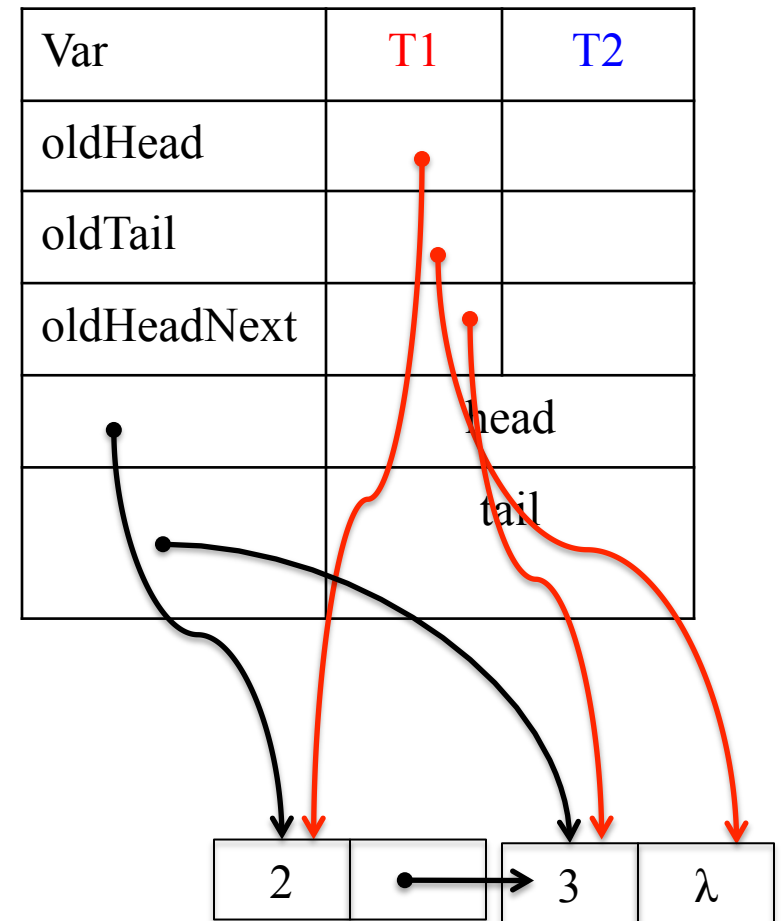
public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4 →    if (oldHead == head.get()) {             //head, tail, and next changed?
5         if (oldHead == oldTail) {             //Queue empty or tail updated?
6             if (oldHeadNext == null) { // Is queue empty?
7                 return null;                 //Queue is empty, can't take
8             }
9             tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10          } else {                             // No need to deal with tail
11              if (head.compareAndSet(oldHead, oldHeadNext)) {
12                  return oldHeadNext.item;
13              }
14          }
15      }
16  }
17  }
18  }
19  }
20  }
    
```



Trace 1: T1 continues

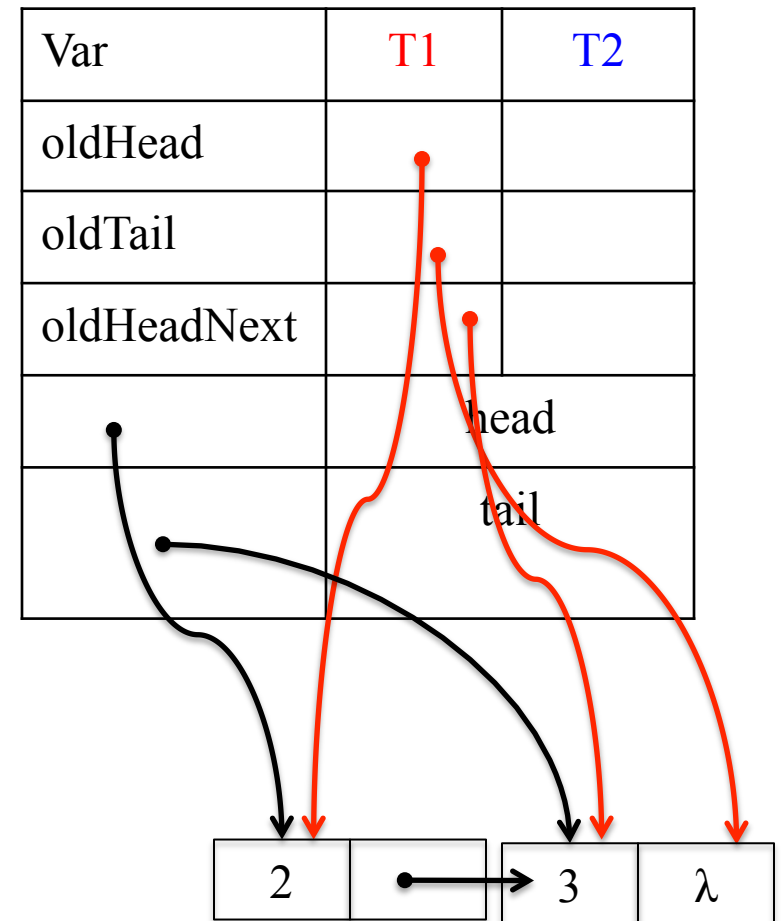
```

public E take() {
    for (;;) {
1       Node<E> oldHead = head.get();           // current head
2       Node<E> oldTail = tail.get();           // current tail
3       Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4       if (oldHead == head.get()) {           //head, tail, and next changed?
5       →   if (oldHead == oldTail) {           //Queue empty or tail updated?
6           if (oldHeadNext == null) { // Is queue empty?
7               return null;           //Queue is empty, can't take
            }
8           tail.compareAndSet(oldTail, oldHeadNext); // tail updating
9       } else {                               // No need to deal with tail
10          if (head.compareAndSet(oldHead, oldHeadNext)) {
              return oldHeadNext.item;
          }
      }
    }
}
    
```



Trace 1: T1 CAS succeeds

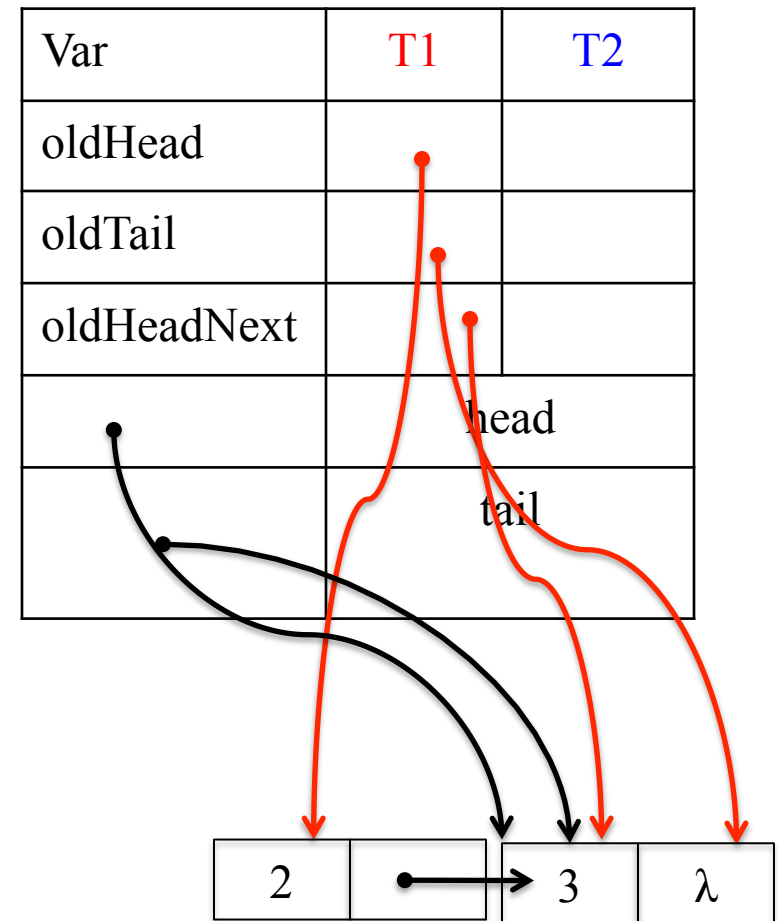
```
public E take() {  
    for (;;) {  
1      Node<E> oldHead = head.get();           // current head  
2      Node<E> oldTail = tail.get();           // current tail  
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4      if (oldHead == head.get()) {           //head, tail, and next changed?  
5          if (oldHead == oldTail) {           //Queue empty or tail updated?  
6              if (oldHeadNext == null) { // Is queue empty?  
7                  return null;           //Queue is empty, can't take  
8              }  
8          tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
9      } else {                               // No need to deal with tail  
10         if (head.compareAndSet(oldHead, oldHeadNext)) {  
11             return oldHeadNext.item;  
12         }  
13     }  
14 }  
15 }
```



Trace 1: T1 after CAS

```

public E take() {
  for (;;) {
1    Node<E> oldHead = head.get();           // current head
2    Node<E> oldTail = tail.get();           // current tail
3    Node<E> oldHeadNext = oldHead.next.get(); // curr head.next
4    if (oldHead == head.get()) {             //head, tail, and next changed?
5      if (oldHead == oldTail) {              //Queue empty or tail updated?
6        if (oldHeadNext == null) {           // Is queue empty?
7          return null;                       //Queue is empty, can't take
8        }
9      tail.compareAndSet(oldTail, oldHeadNext); // tail updating
10     } else {                               // No need to deal with tail
11       if (head.compareAndSet(oldHead, oldHeadNext)) {
12         return oldHeadNext.item;
13       }
14     }
15   }
16 }
  
```



Trace 1: T1 exits

```
public E take() {  
    for (;;) {  
1      Node<E> oldHead = head.get();           // current head  
2      Node<E> oldTail = tail.get();           // current tail  
3      Node<E> oldHeadNext = oldHead.next.get(); // curr head.next  
4      if (oldHead == head.get()) {           //head, tail, and next changed?  
5          if (oldHead == oldTail) {           //Queue empty or tail updated?  
6              if (oldHeadNext == null) { // Is queue empty?  
7                  return null;               //Queue is empty, can't take  
              }  
8              tail.compareAndSet(oldTail, oldHeadNext); // tail updating  
9          } else {                             // No need to deal with tail  
10             if (head.compareAndSet(oldHead, oldHeadNext)) {  
                return oldHeadNext.item;  
            }  
        }  
    }  
}
```

