

# YSC3248: Parallel, Concurrent and Distributed Programming

Concurrent Stacks and Elimination

# Last and This Lectures

- Queue
  - Bounded, blocking, lock-based
  - Unbounded, non-blocking, lock-free
- Stack
  - Unbounded, non-blocking lock-free
  - Elimination-backoff algorithm

# Last and This Lectures

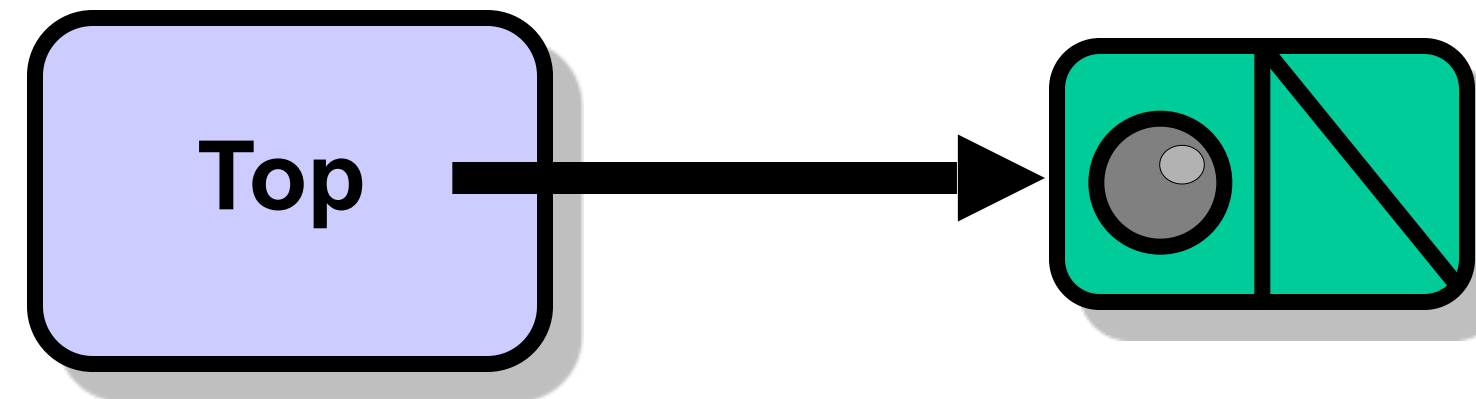
- Queue
  - Bounded, blocking, lock-based
  - Unbounded, non-blocking, lock-free
- Stack
  - Unbounded, non-blocking lock-free
  - Elimination-backoff algorithm

Warm-Up:  
Implementing and Testing Lock-Based Stack

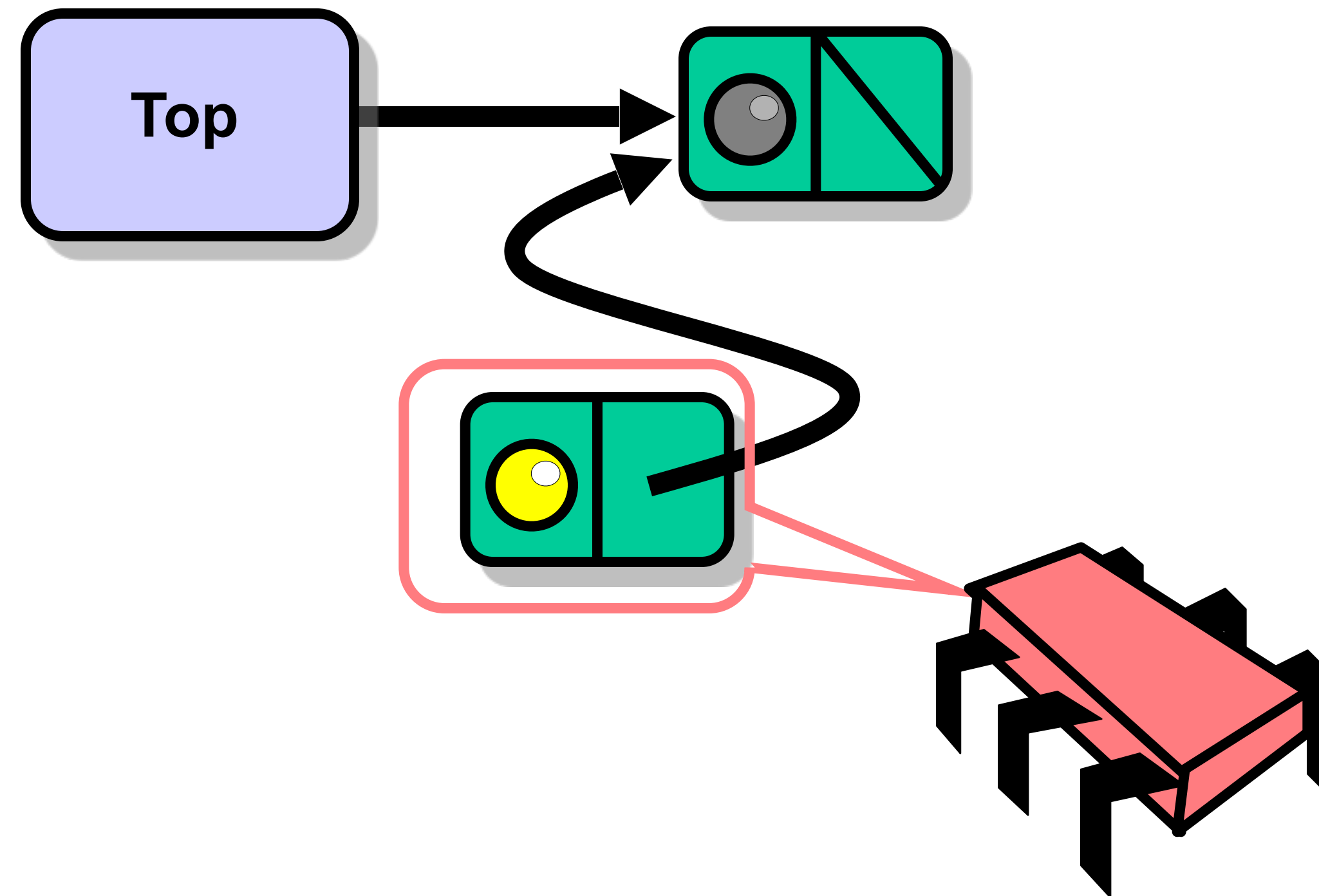
# Concurrent Stack

- Methods
  - **push(x)**
  - **pop()**
- Last-in, First-out (LIFO) order
- Lock-Free!

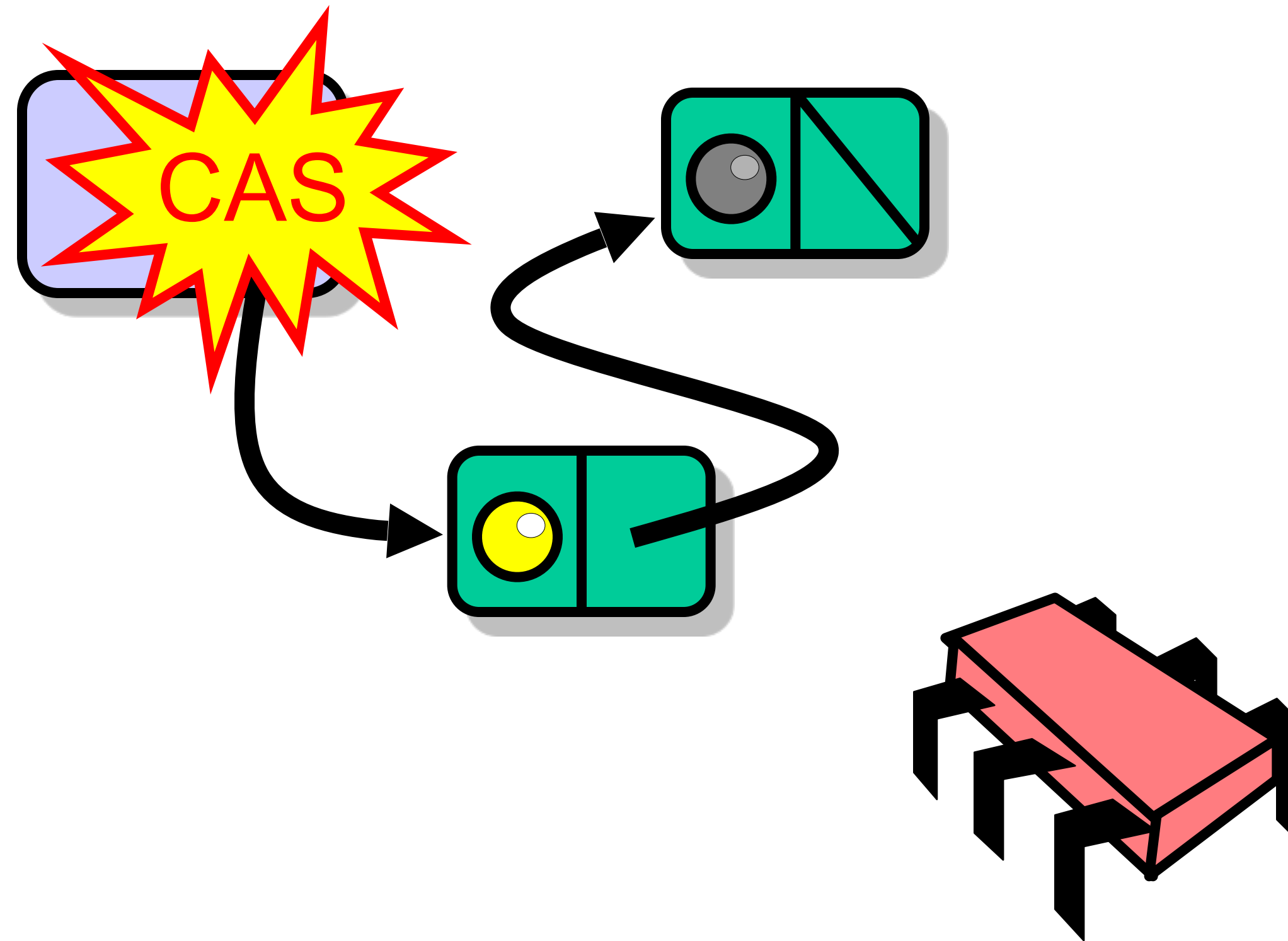
# Empty Stack



# Push

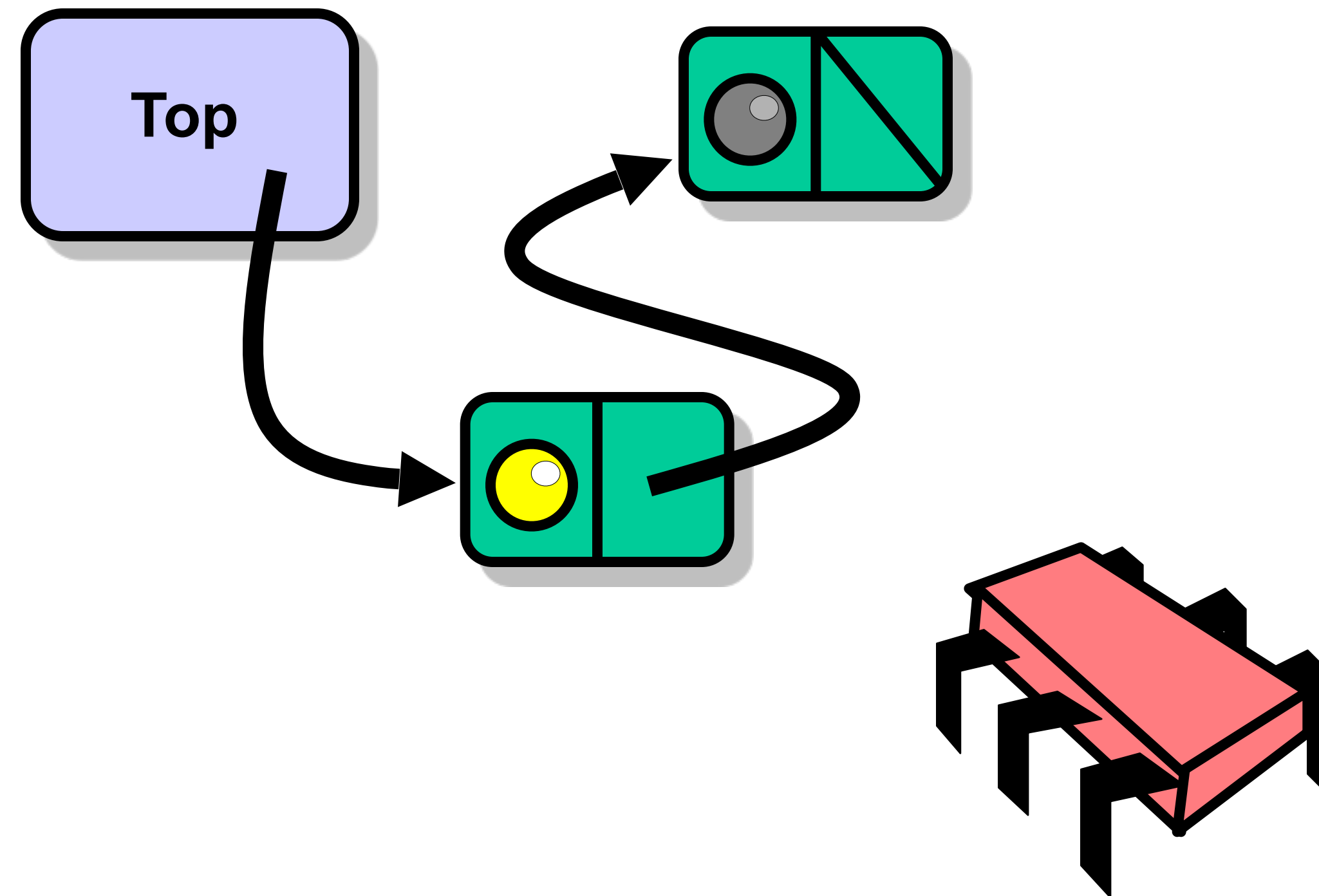


# Push

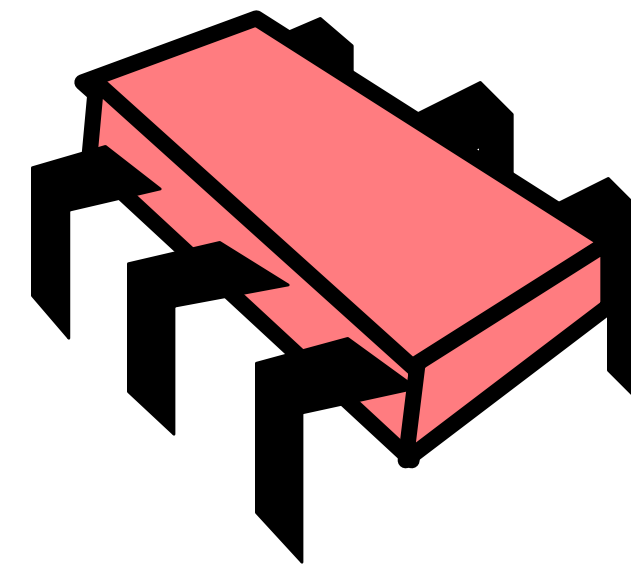
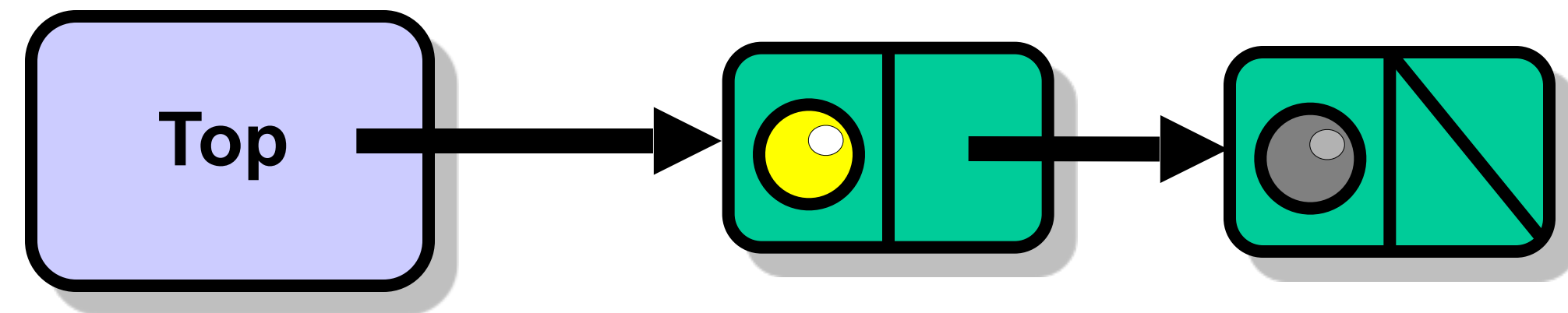




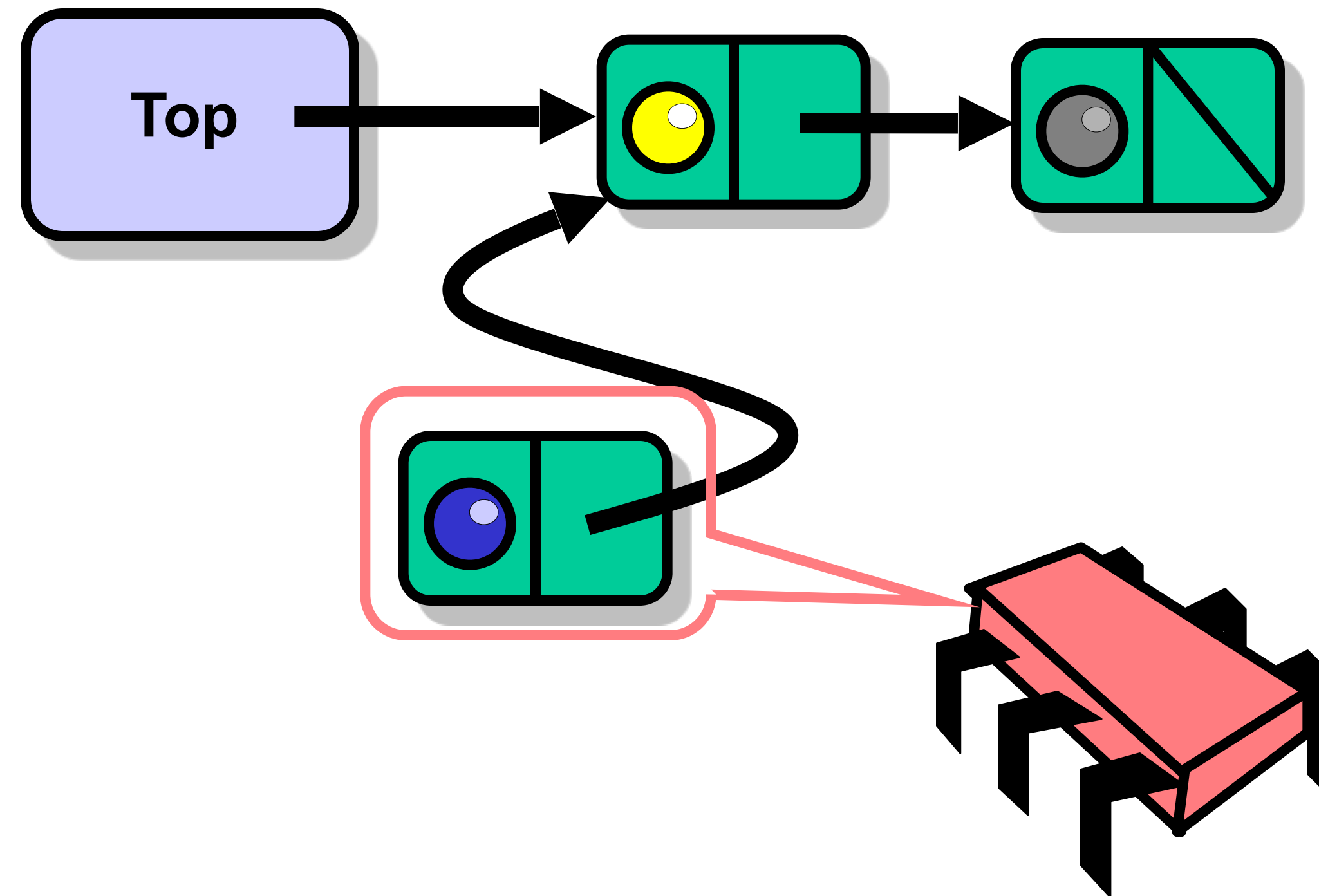
# Push



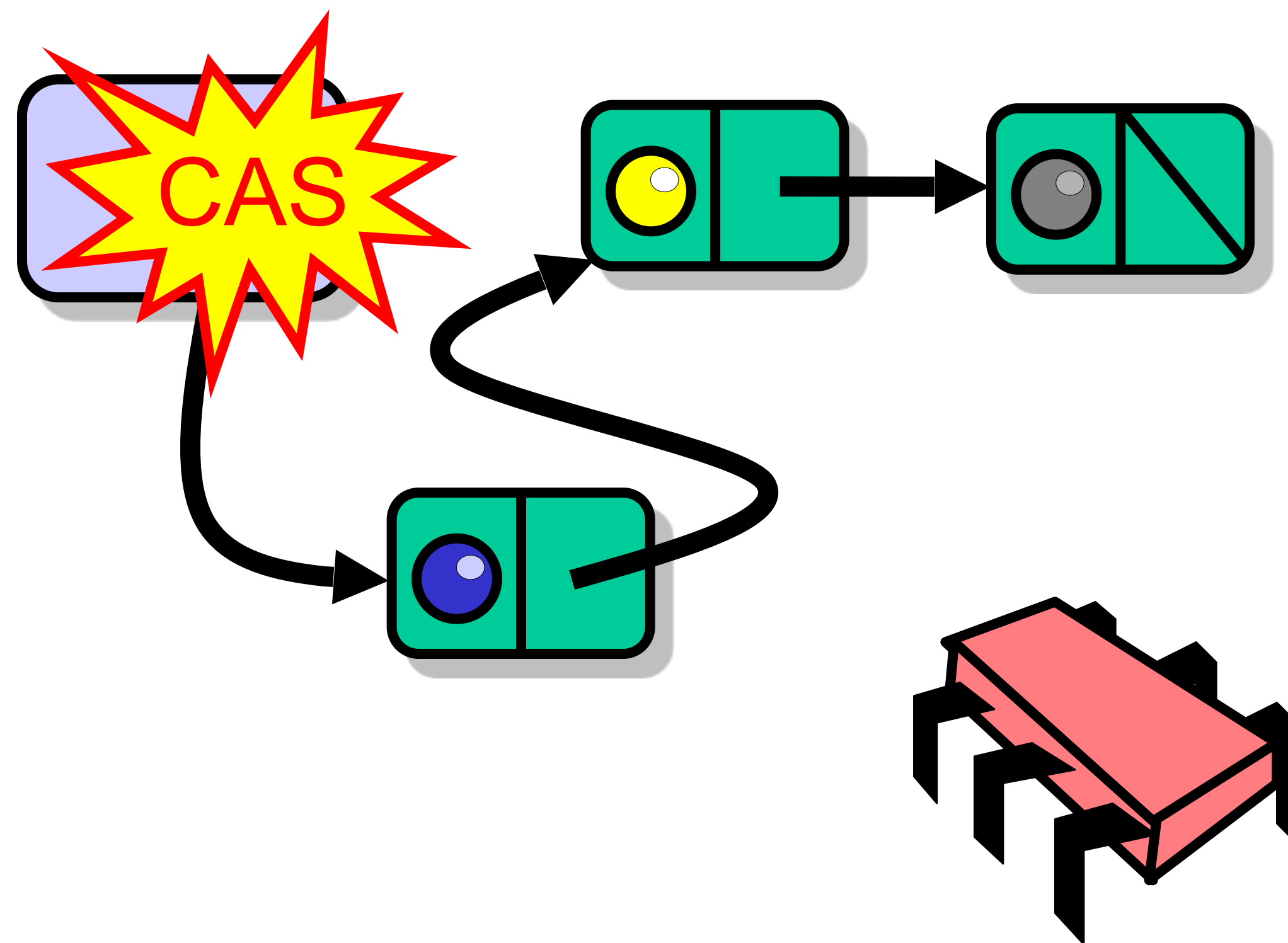
# Push



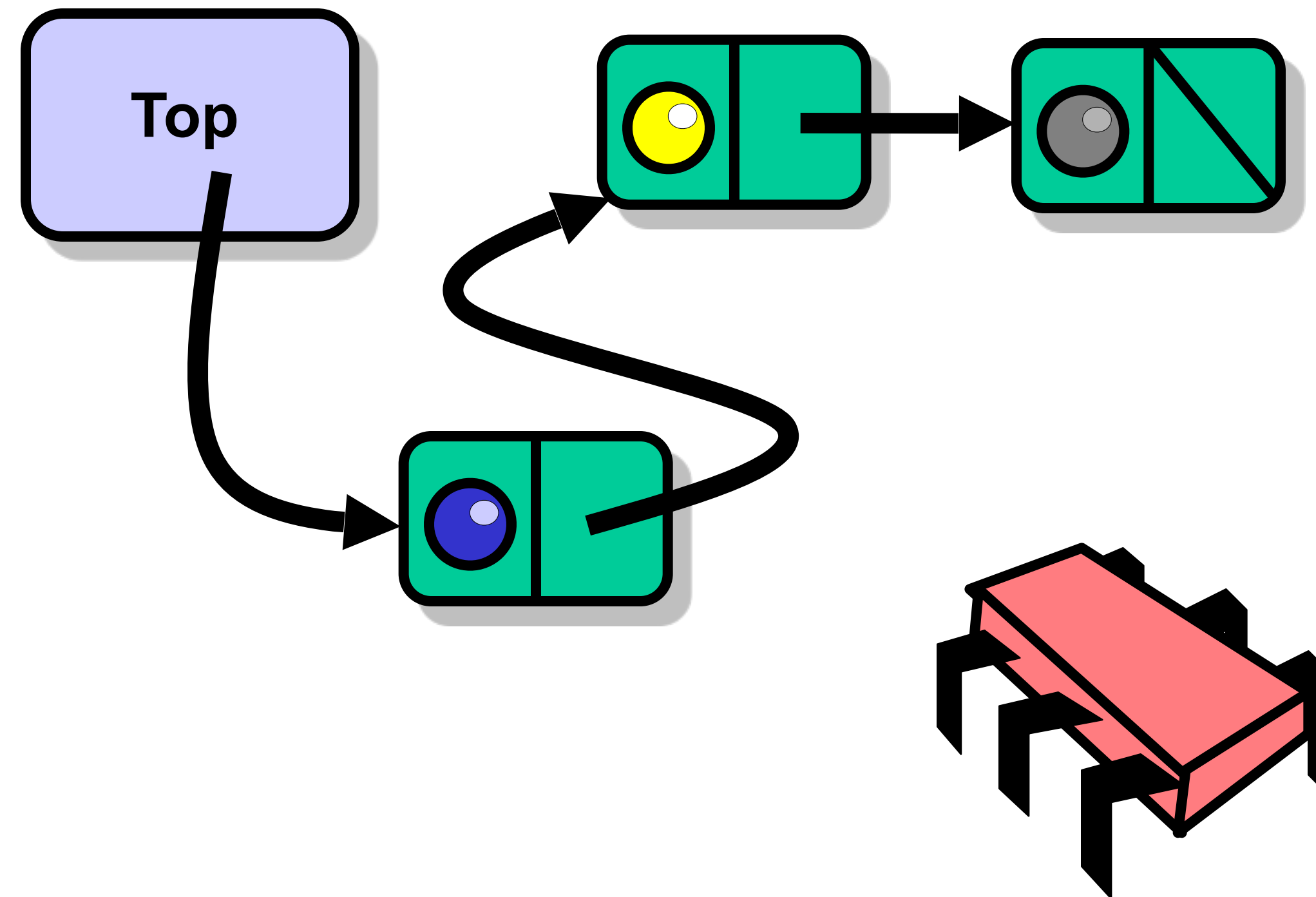
# Push



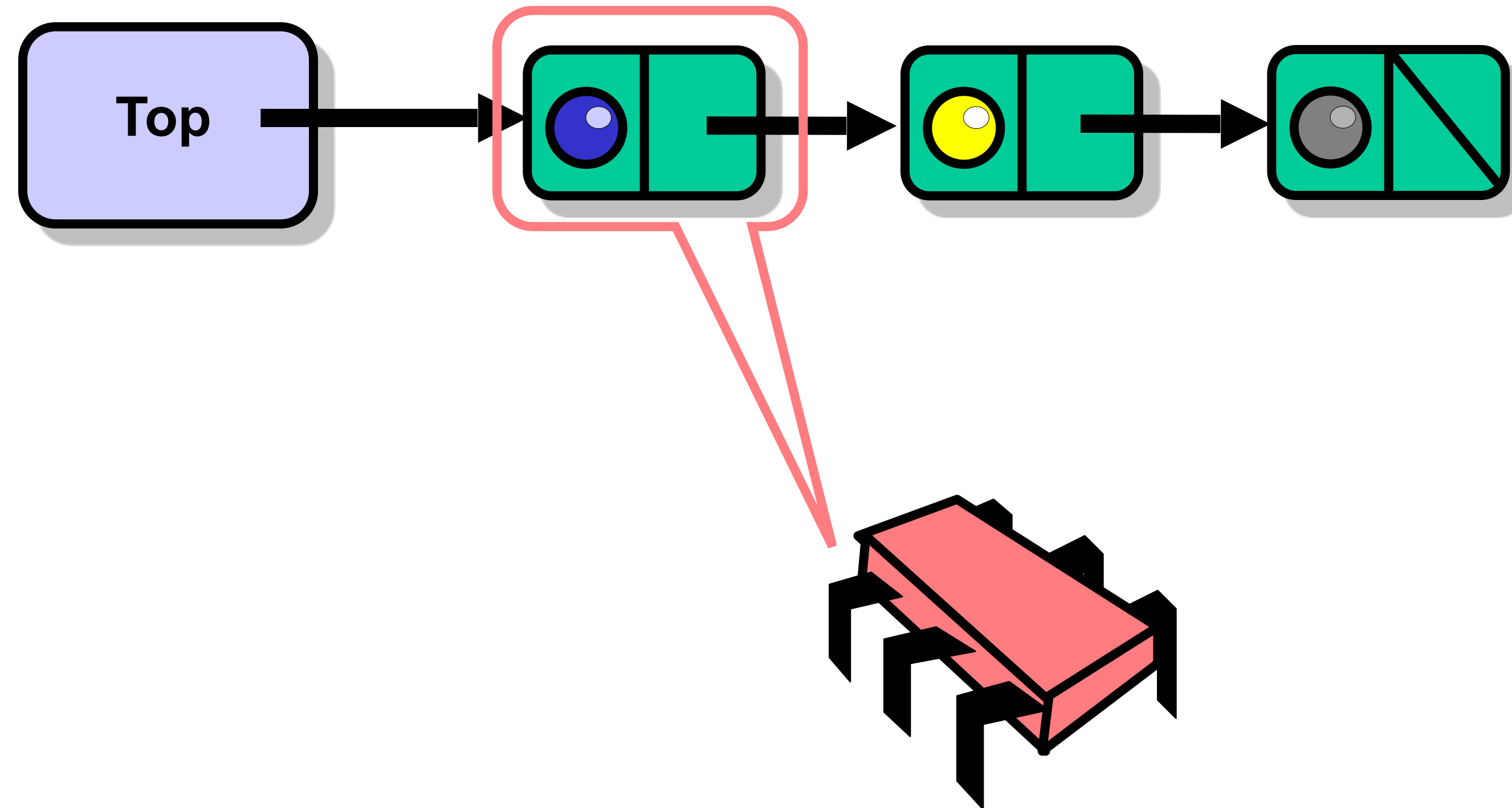
# Push



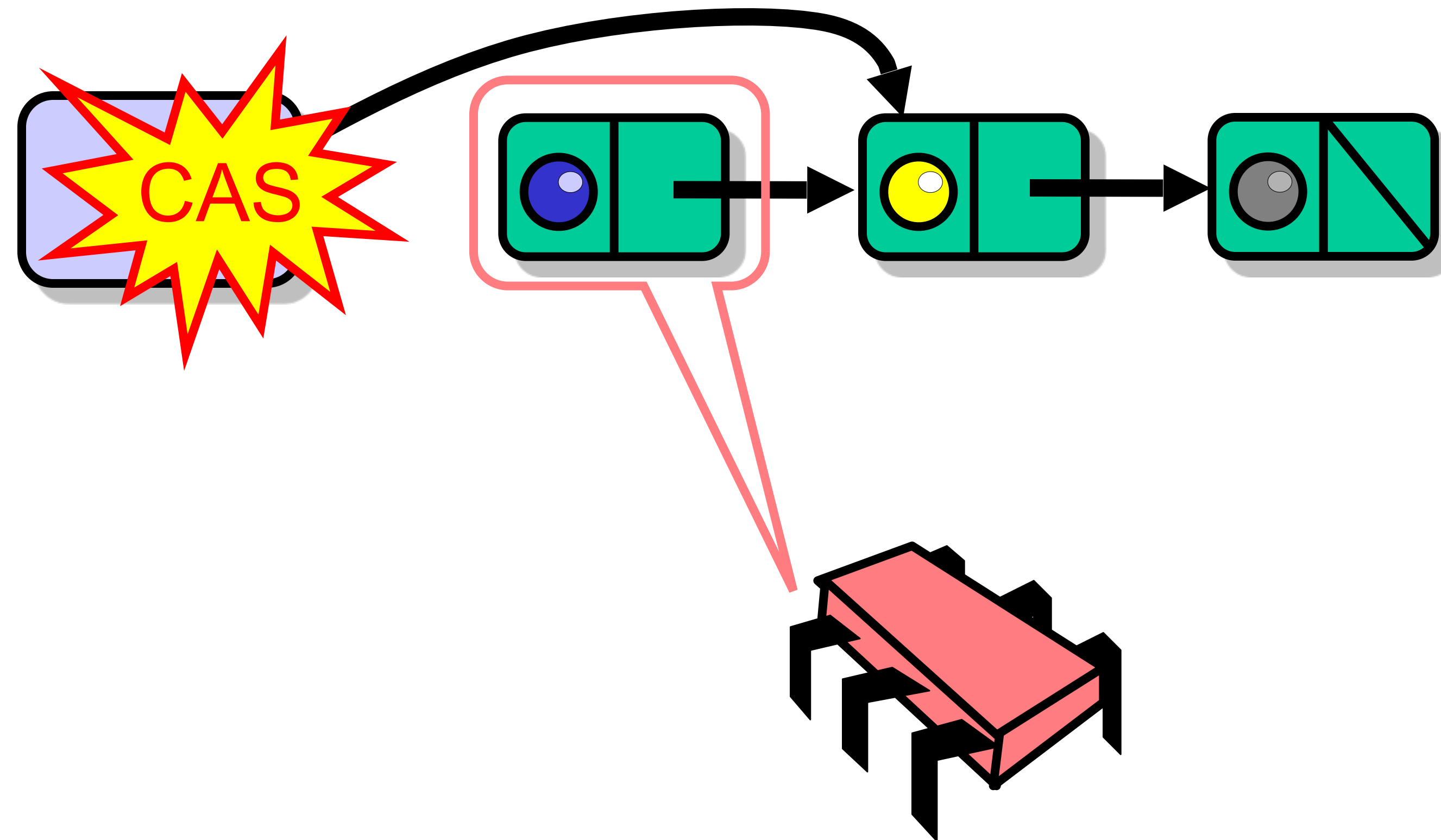
# Push



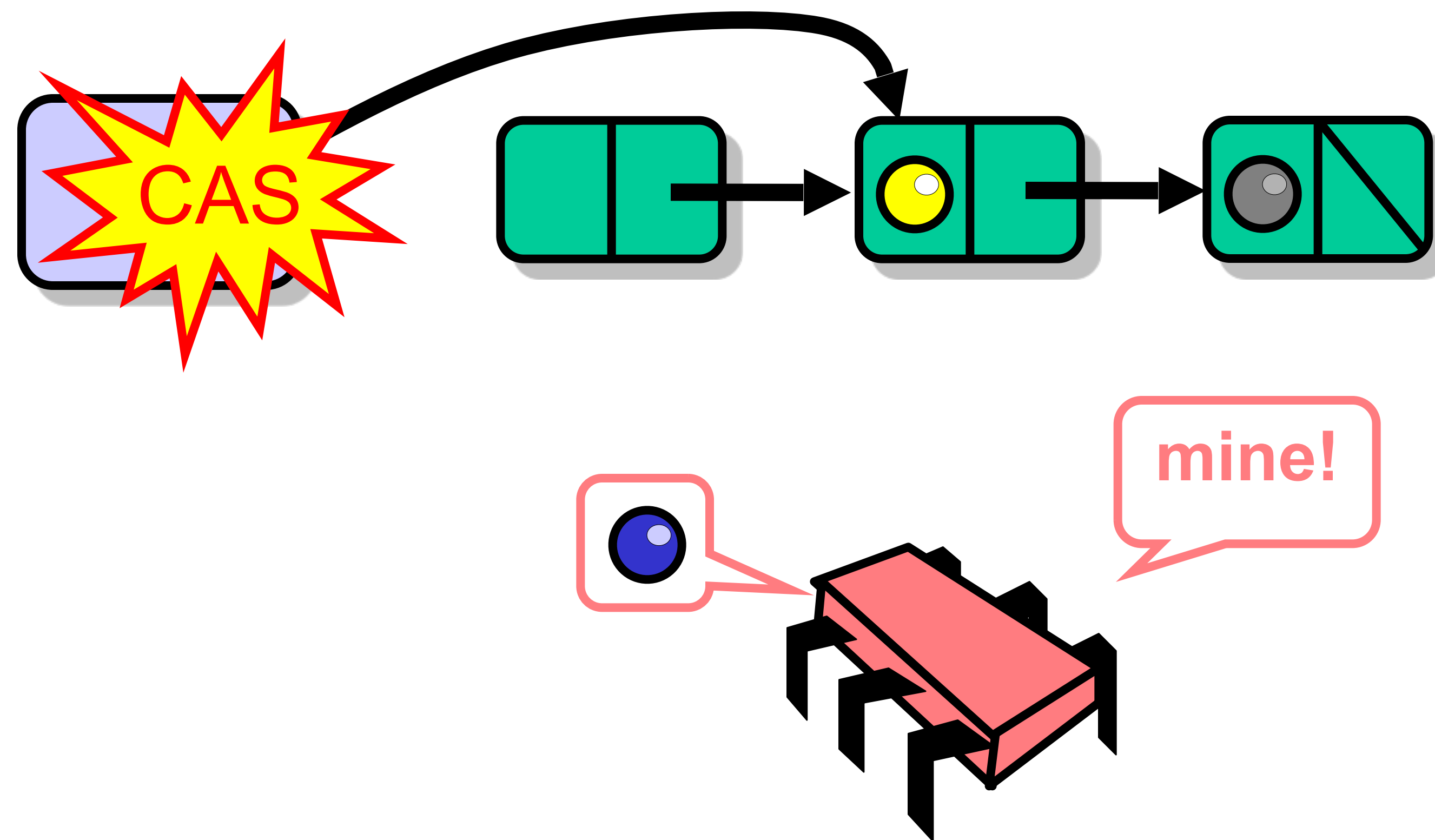
# Pop



# Pop

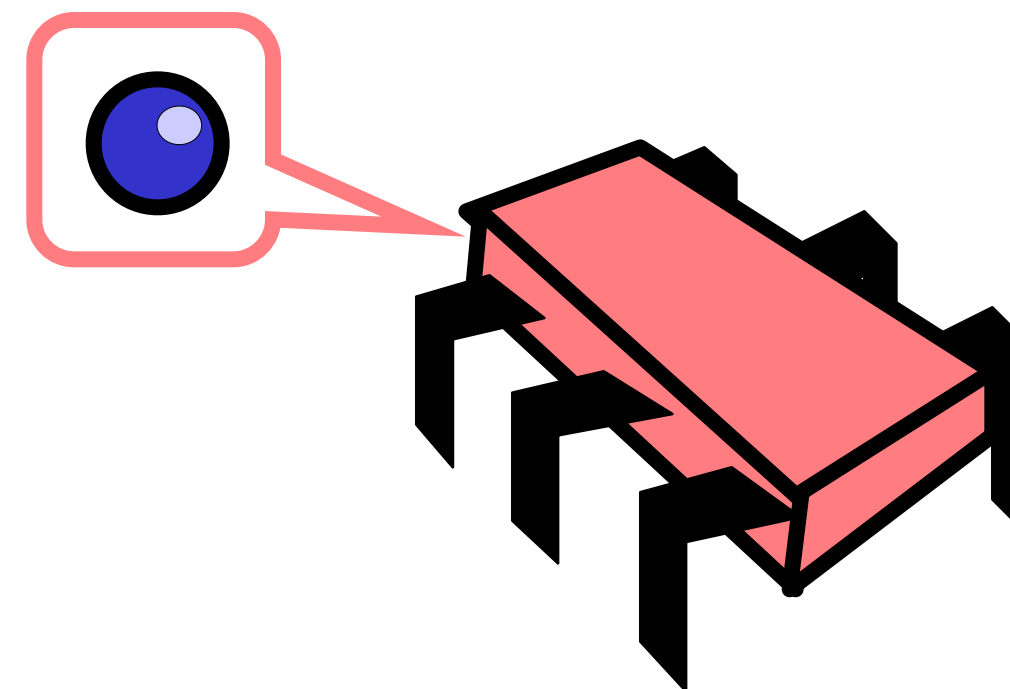
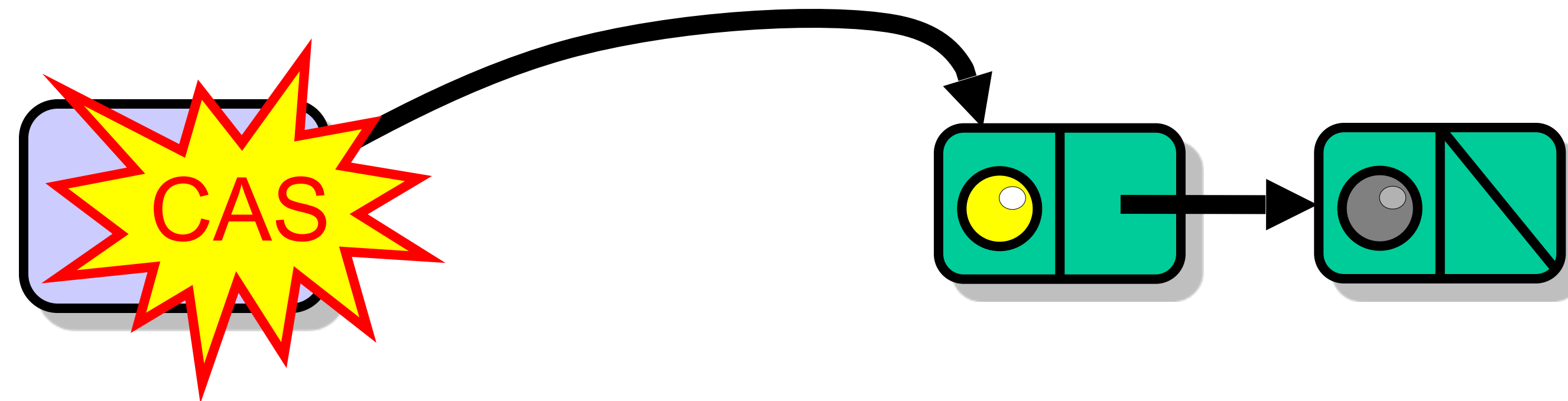


# Pop

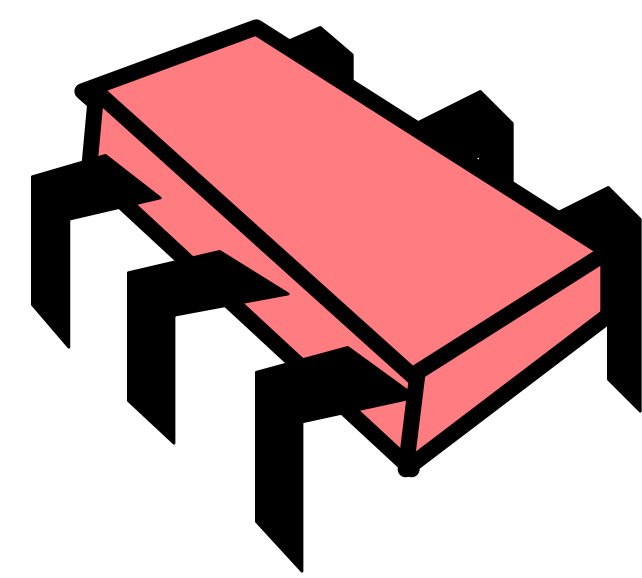
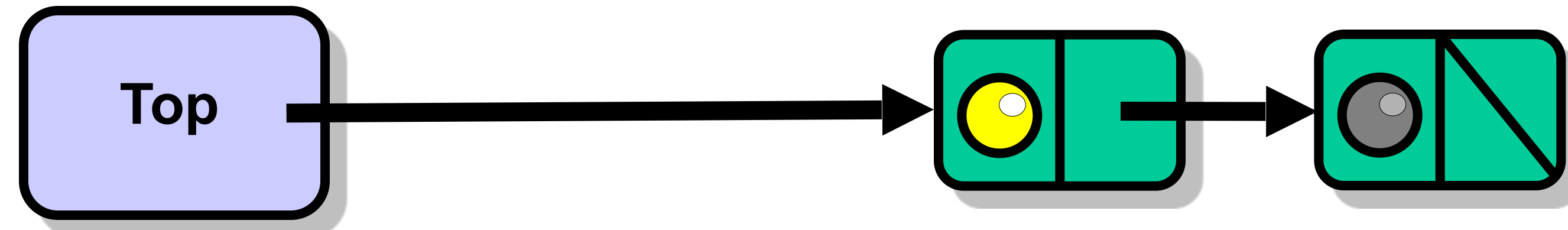




# Pop



# Pop



# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node] (null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.backoff()  
    }  
  }
```

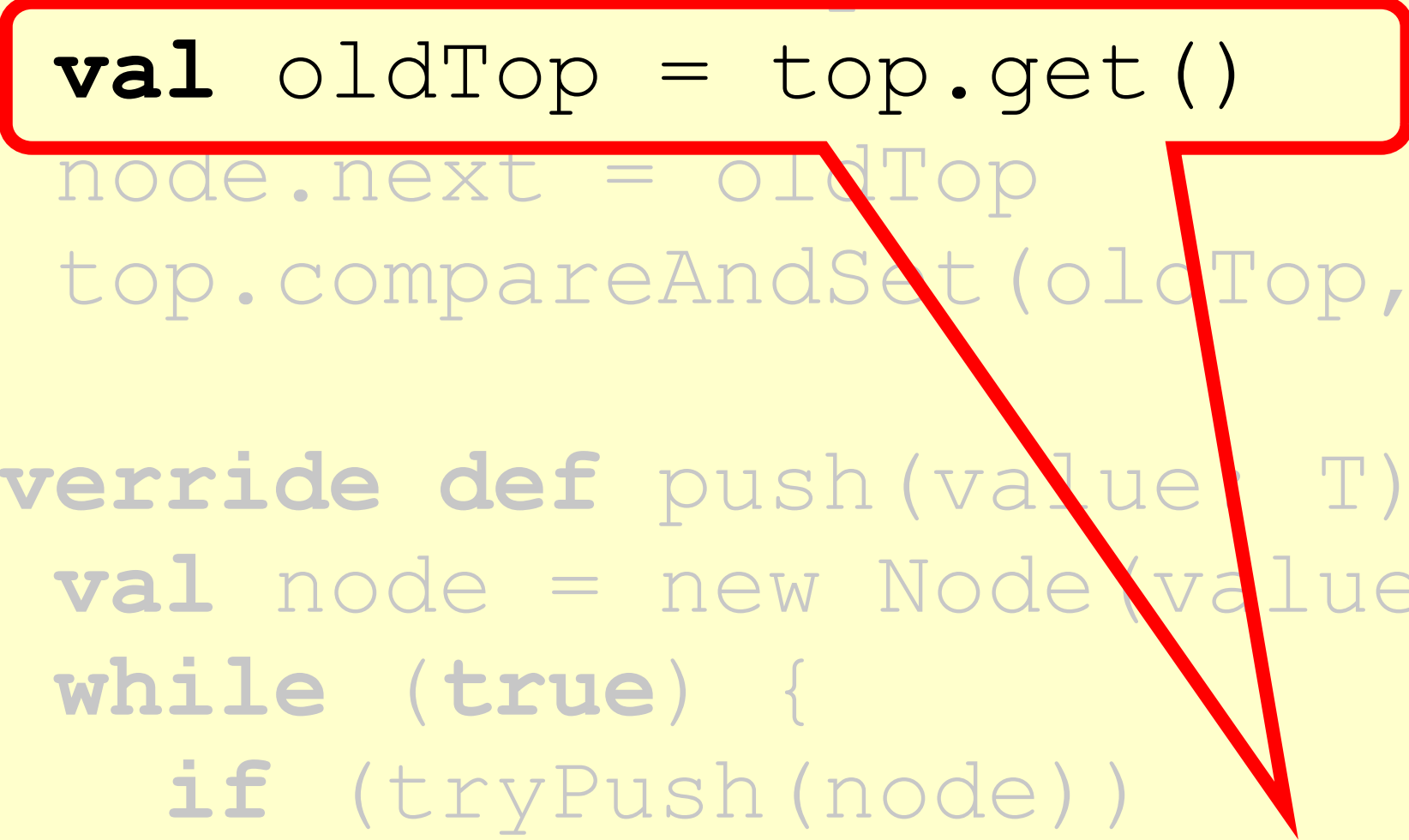
# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
    }  
  }  
}
```

**tryPush attempts to push a node**

# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.  
    }  
  }  
}
```



**Read top value**

# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.backoff()  
    }  
  }  
}
```

**current top will be new node's successor**

# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
    }  
  }  
}
```

**Try to swing top, return success or failure**

# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.backoff()  
    }  
  }  
}
```

**Push calls tryPush**



# Lock-free Stack

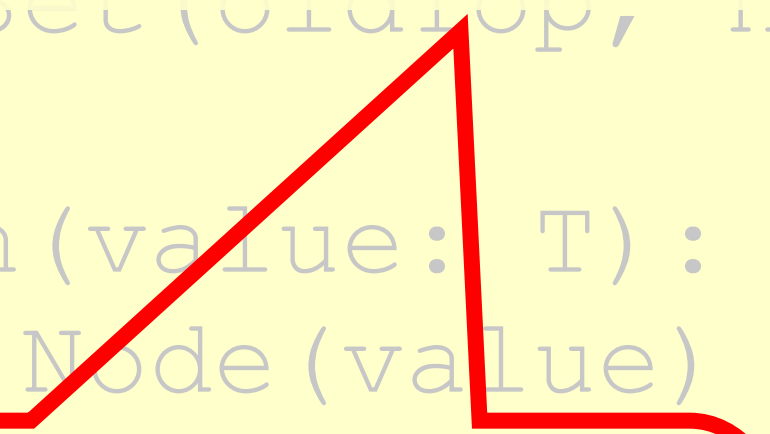
```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.backoff()  
    }  
  }  
}
```

**Create new node**

# Lock-free Stack

```
class LockFreeStack[T] extends ConcurrentStack[T] {  
  val top = new AtomicReference[Node](null)  
  protected def tryPush(node: Node): Boolean = {  
    val oldTop = top.get()  
    node.next = oldTop  
    top.compareAndSet(oldTop, node)  
  }  
  override def push(value: T): Unit = {  
    val node = new Node(value)  
    while (true) {  
      if (tryPush(node))  
        return  
      else backoff.backoff()  
    }  
  }  
}
```

**If tryPush() fails,  
back off before retrying**



# Lock-free Stack

- Good
  - No locking
- Bad
  - Without GC, fear ABA
  - Without backoff, huge contention at top
  - In any case, no parallelism

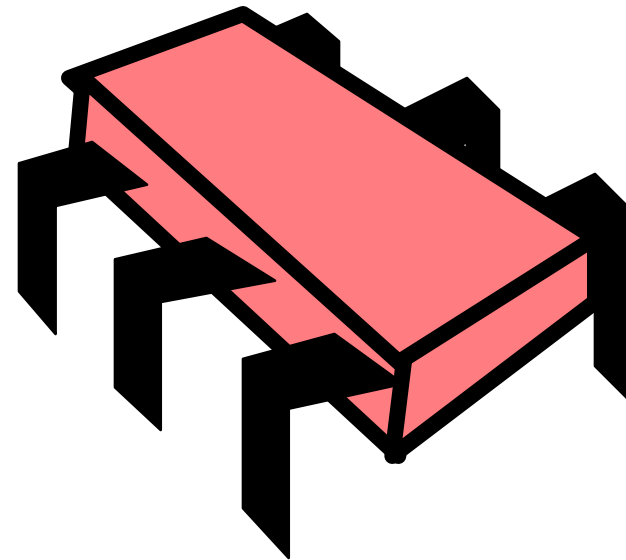
# Big Question

- Are stacks *inherently* sequential?
- Reasons why
  - Every **pop()** call fights for top item
- Reasons why not
  - Stay tuned ...

# Elimination-Backoff Stack

- How to
  - “turn contention into parallelism”
- Replace familiar
  - **exponential backoff**
- With alternative
  - **elimination-backoff**

# Observation

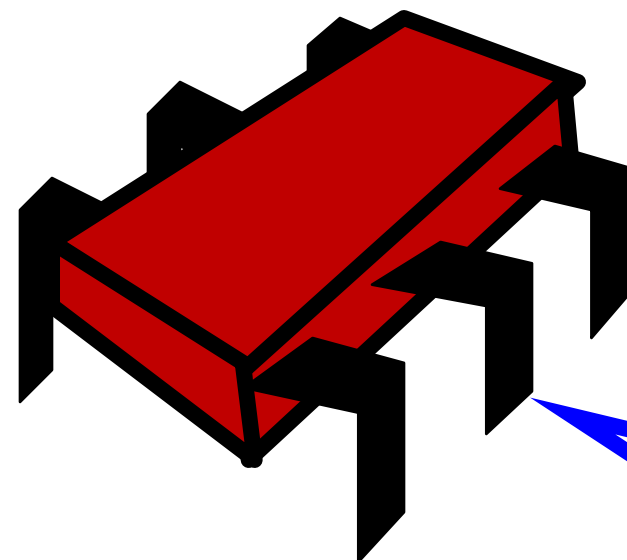


**Push()**

**linearizable stack**



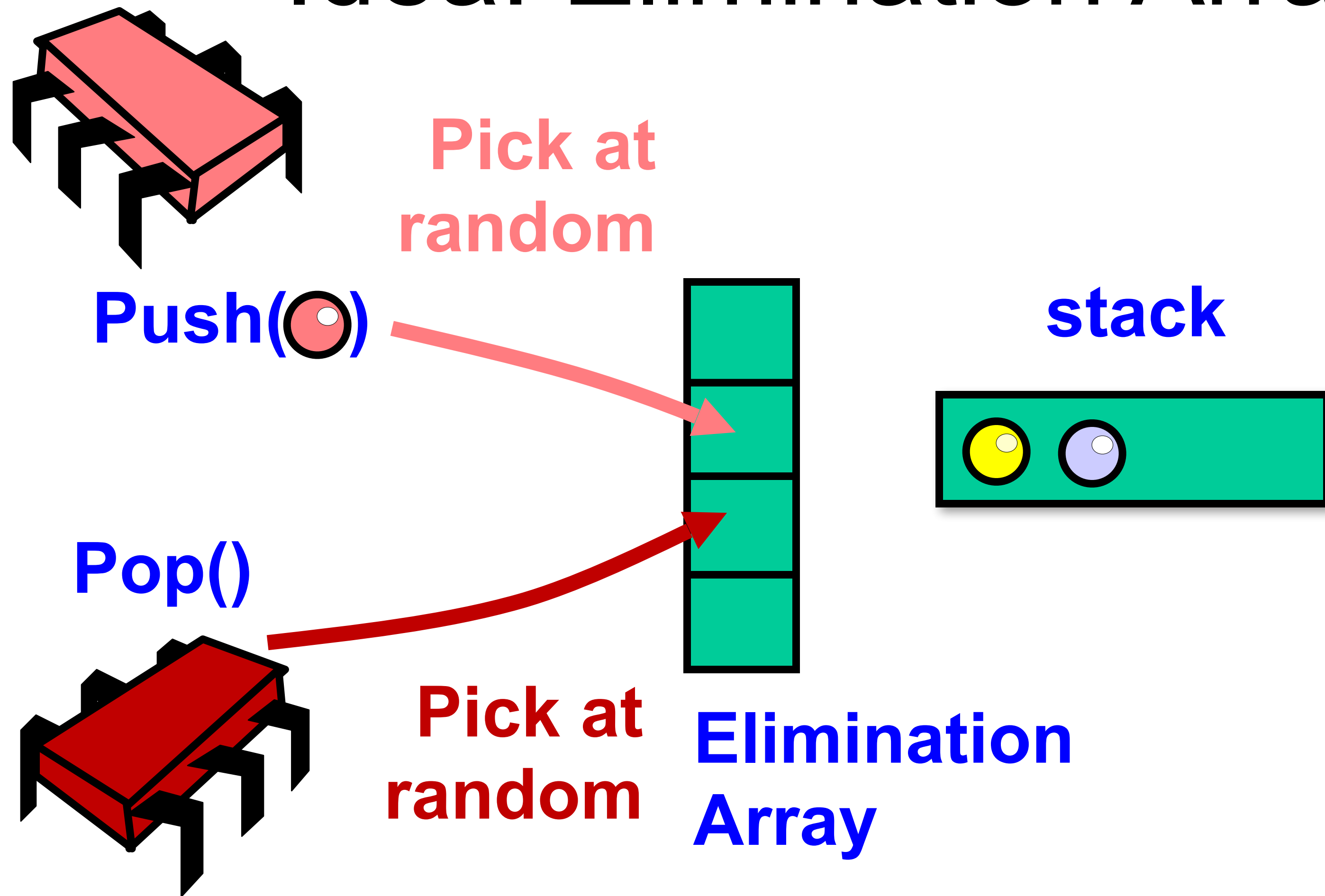
**Pop()**



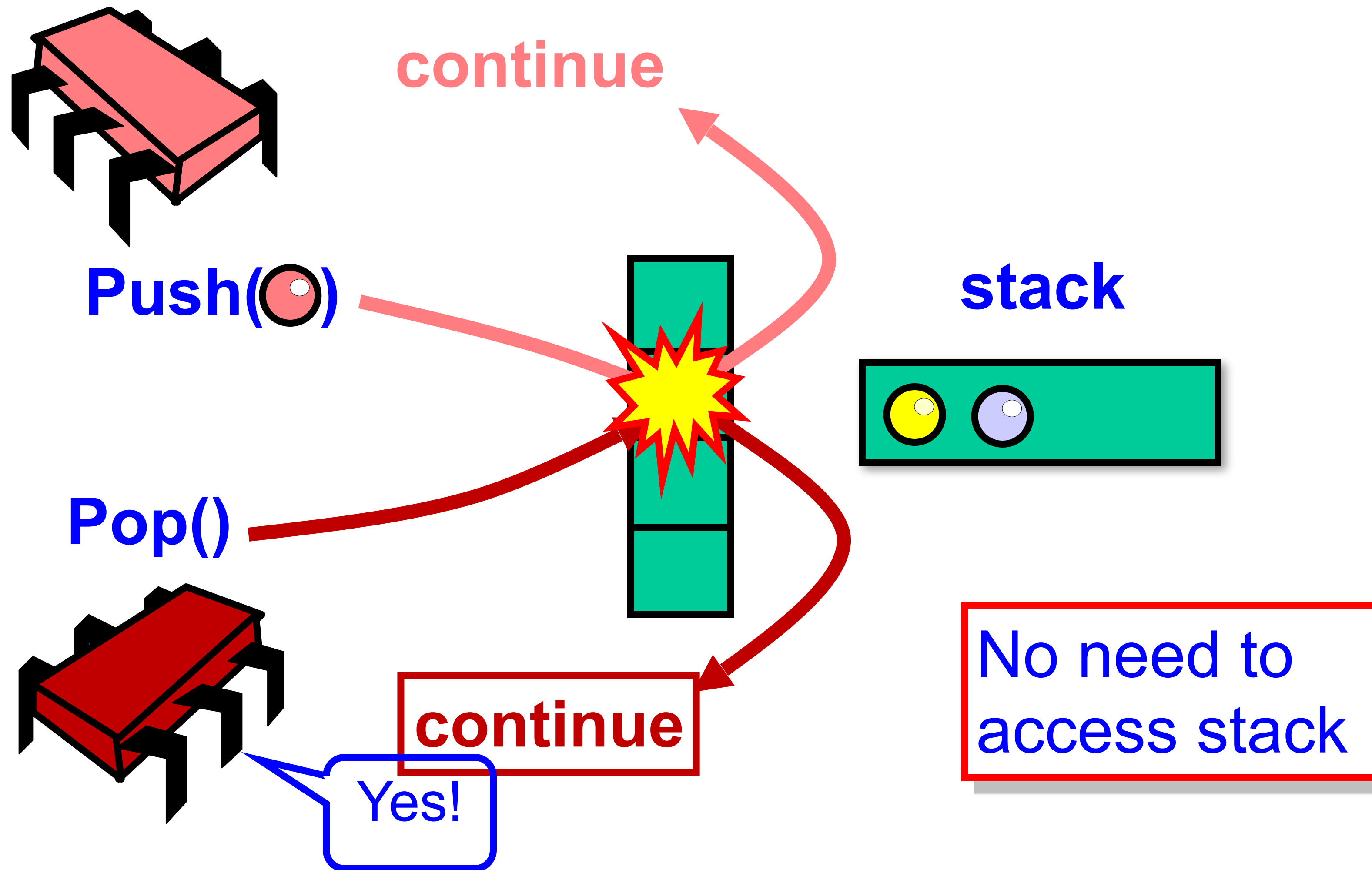
Yes!

After an equal number  
of pushes and pops,  
stack stays the same

# Idea: Elimination Array

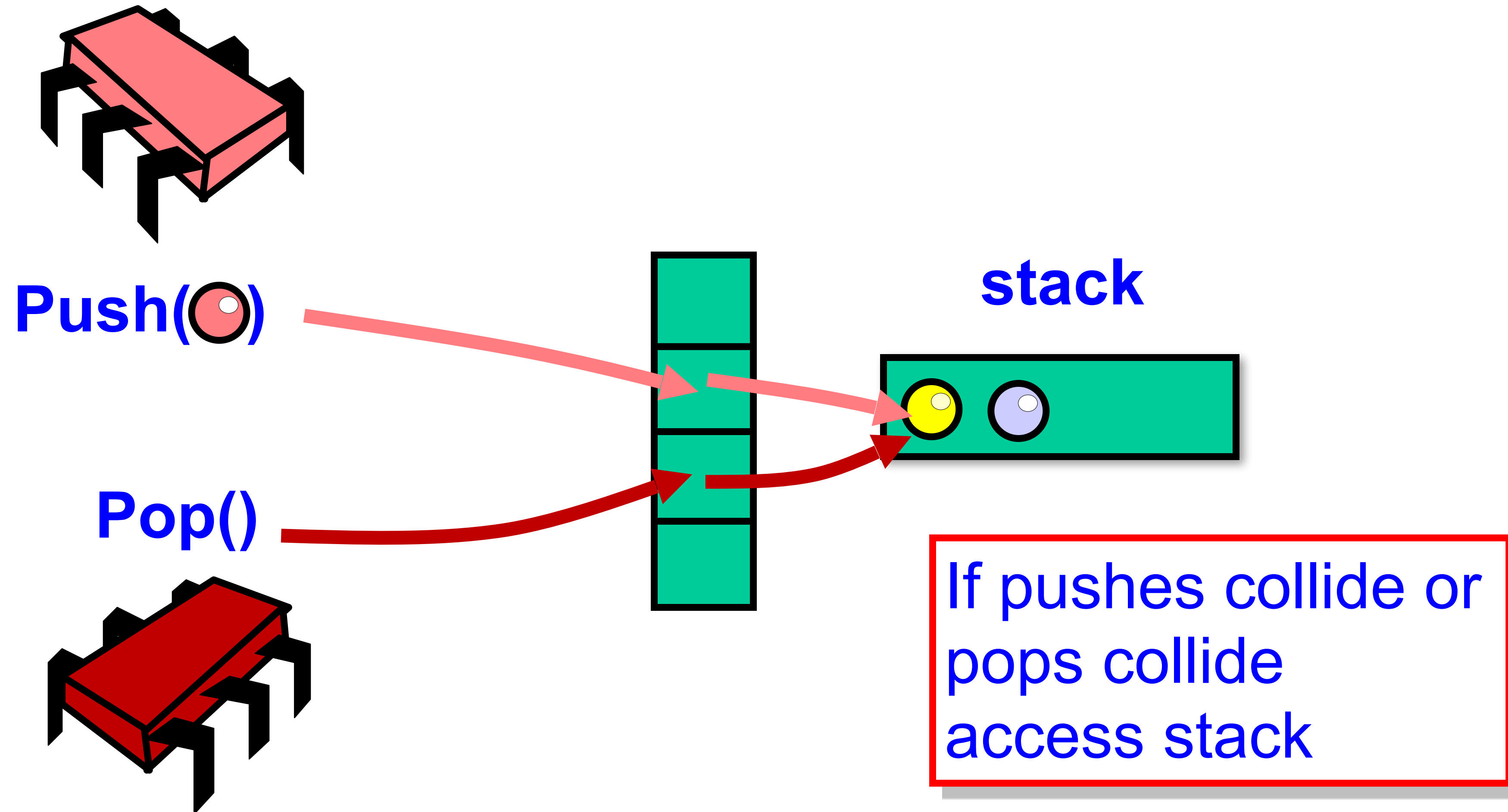


# Push Collides With Pop





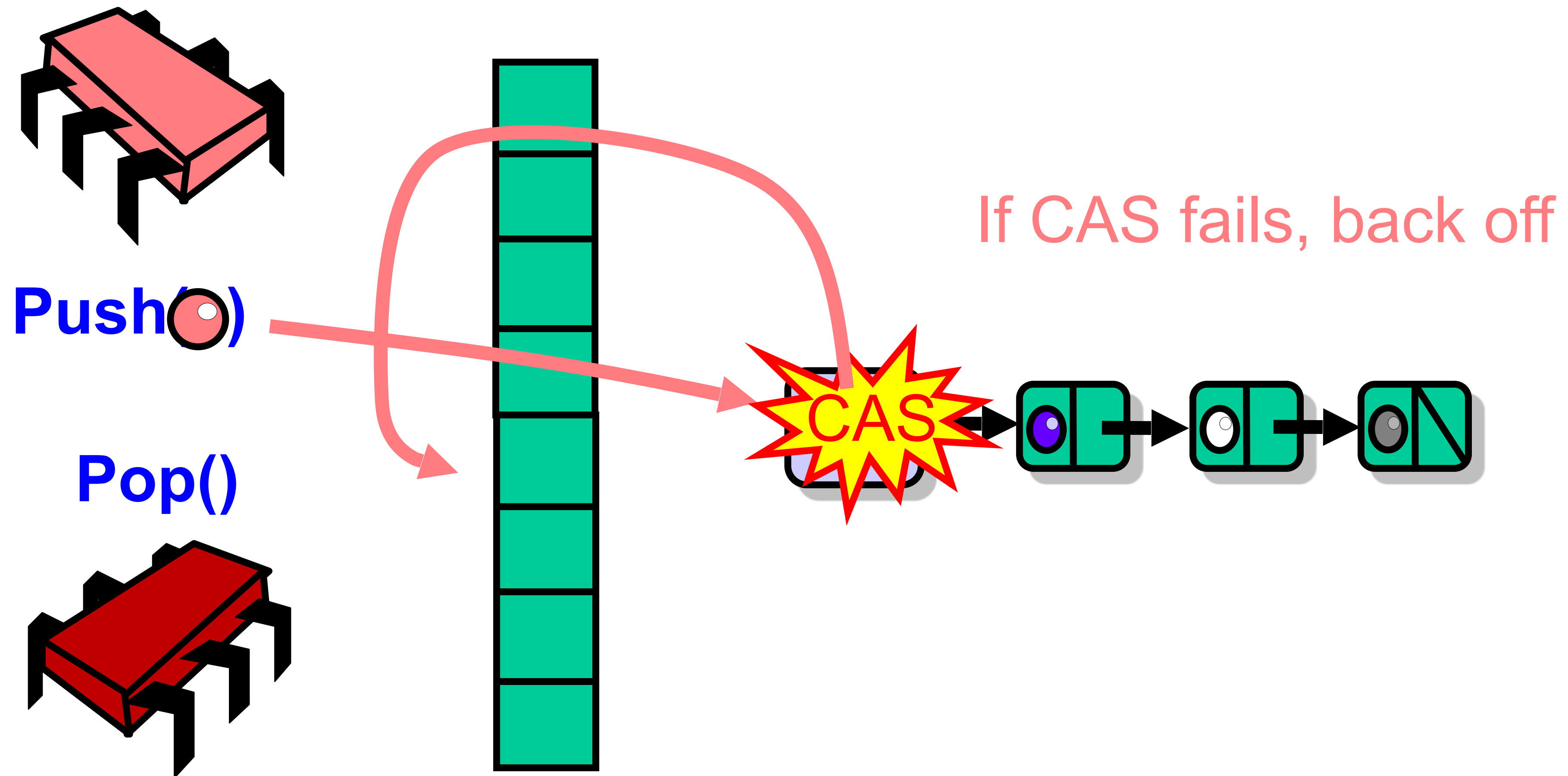
# No Collision



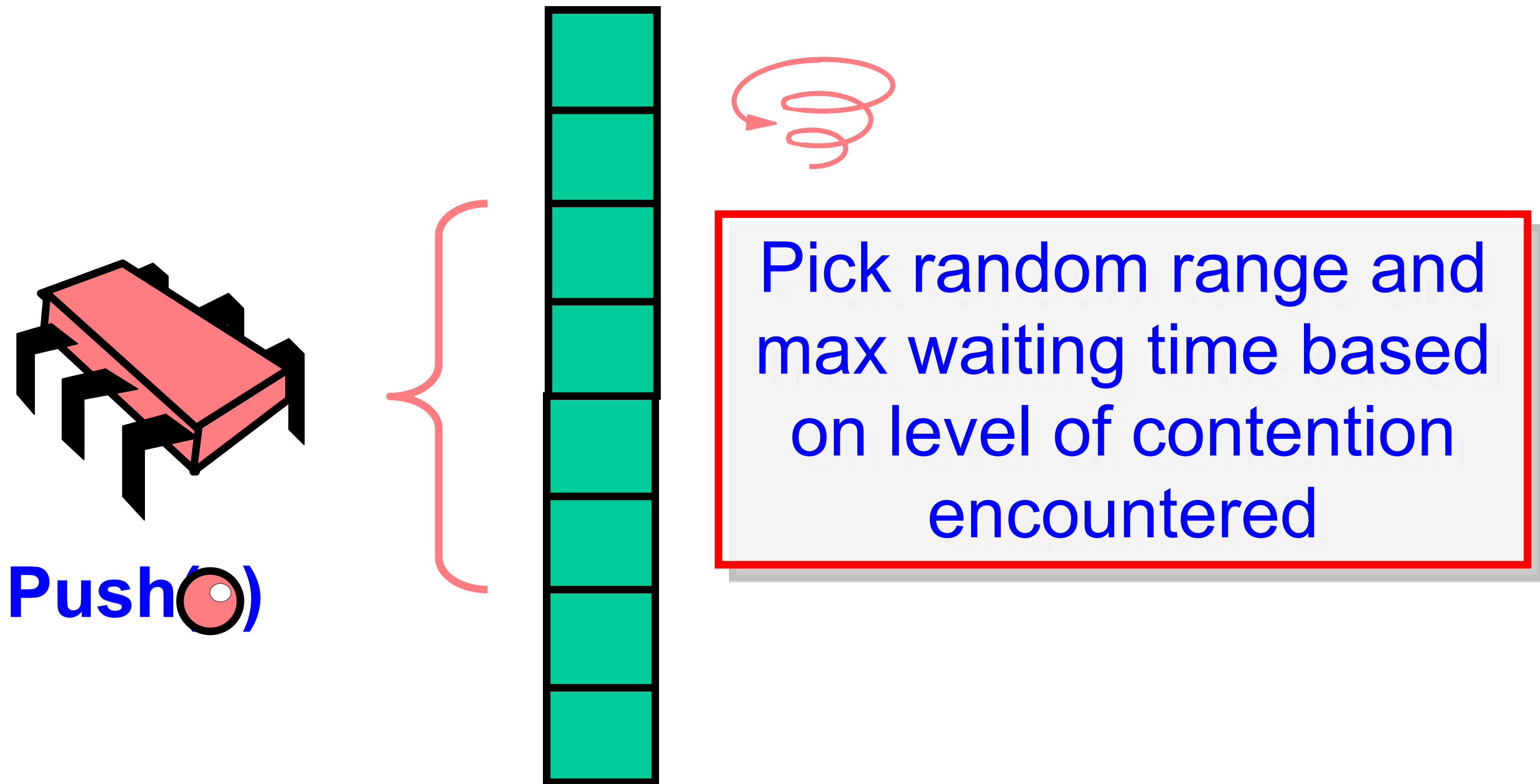
# Elimination-Backoff Stack

- Lock-free stack + elimination array
- Access Lock-free stack,
  - If **uncontended**, apply operation
  - if **contended**, back off to elimination array and attempt elimination

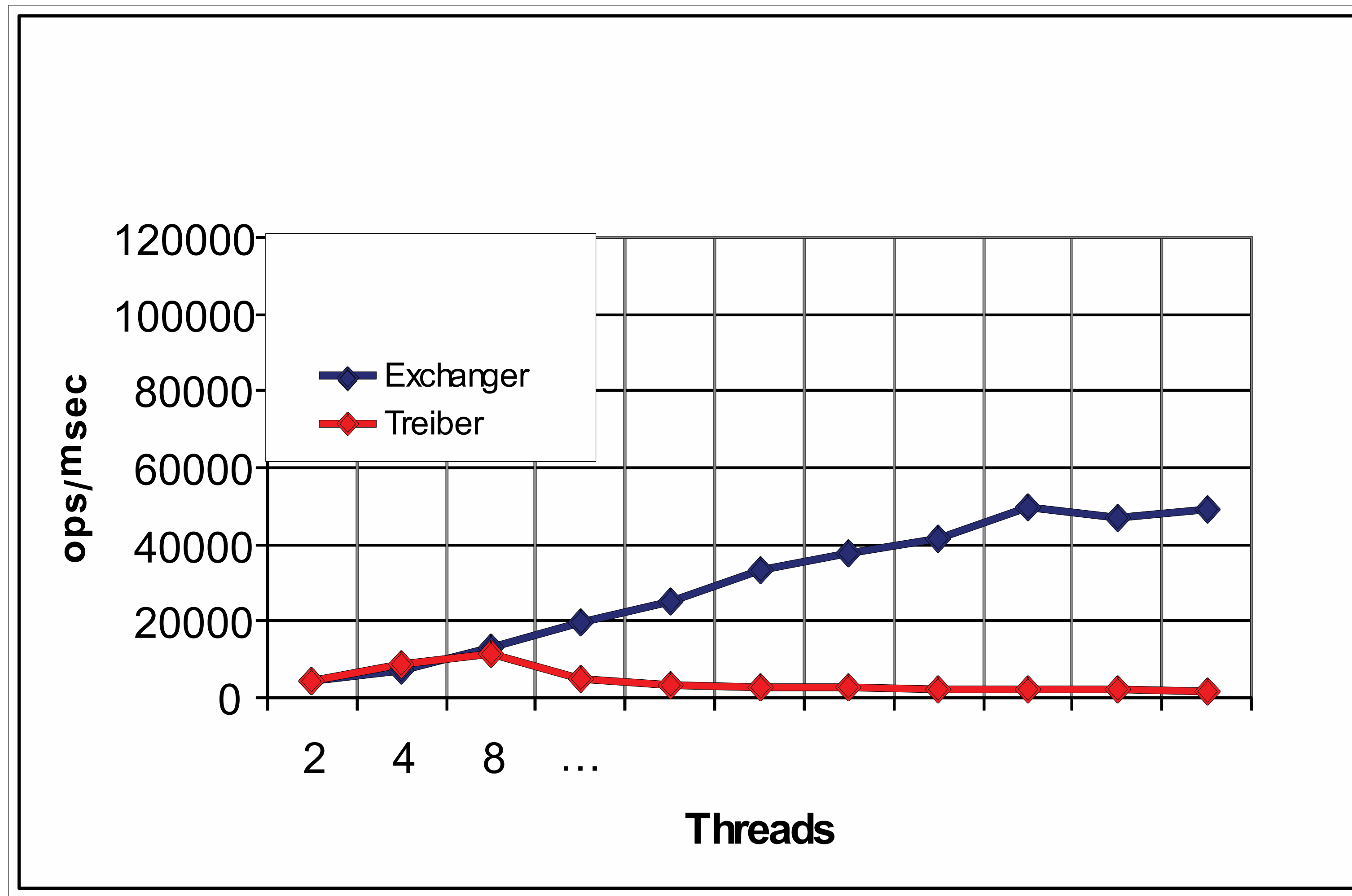
# Elimination-Backoff Stack



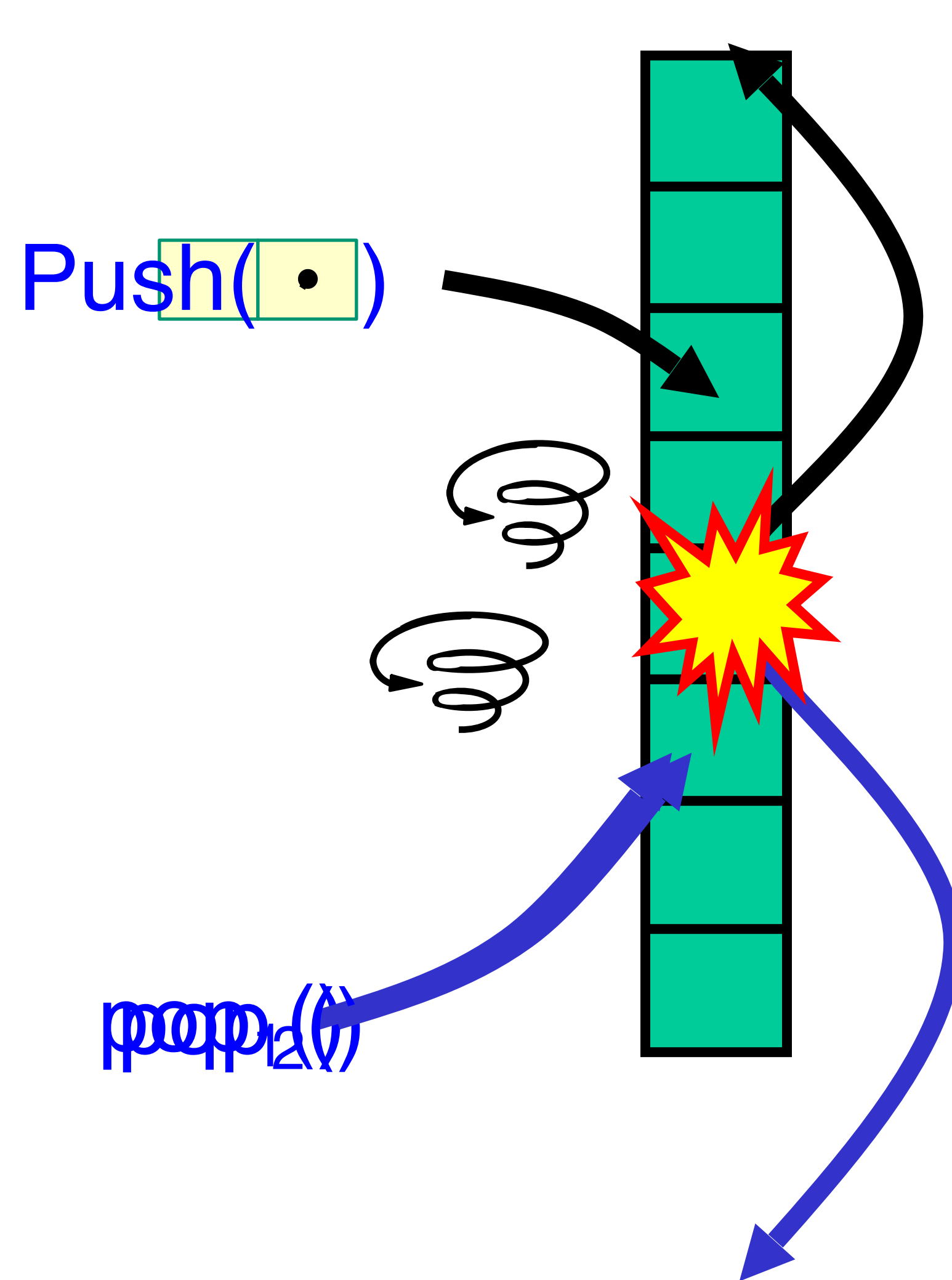
# Dynamic Range and Delay



# 50-50, Random Slots

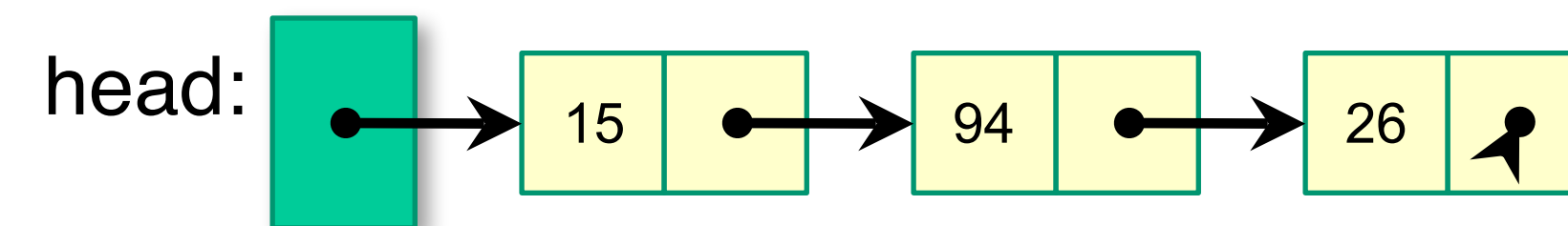


# Asymmetric Rendezvous

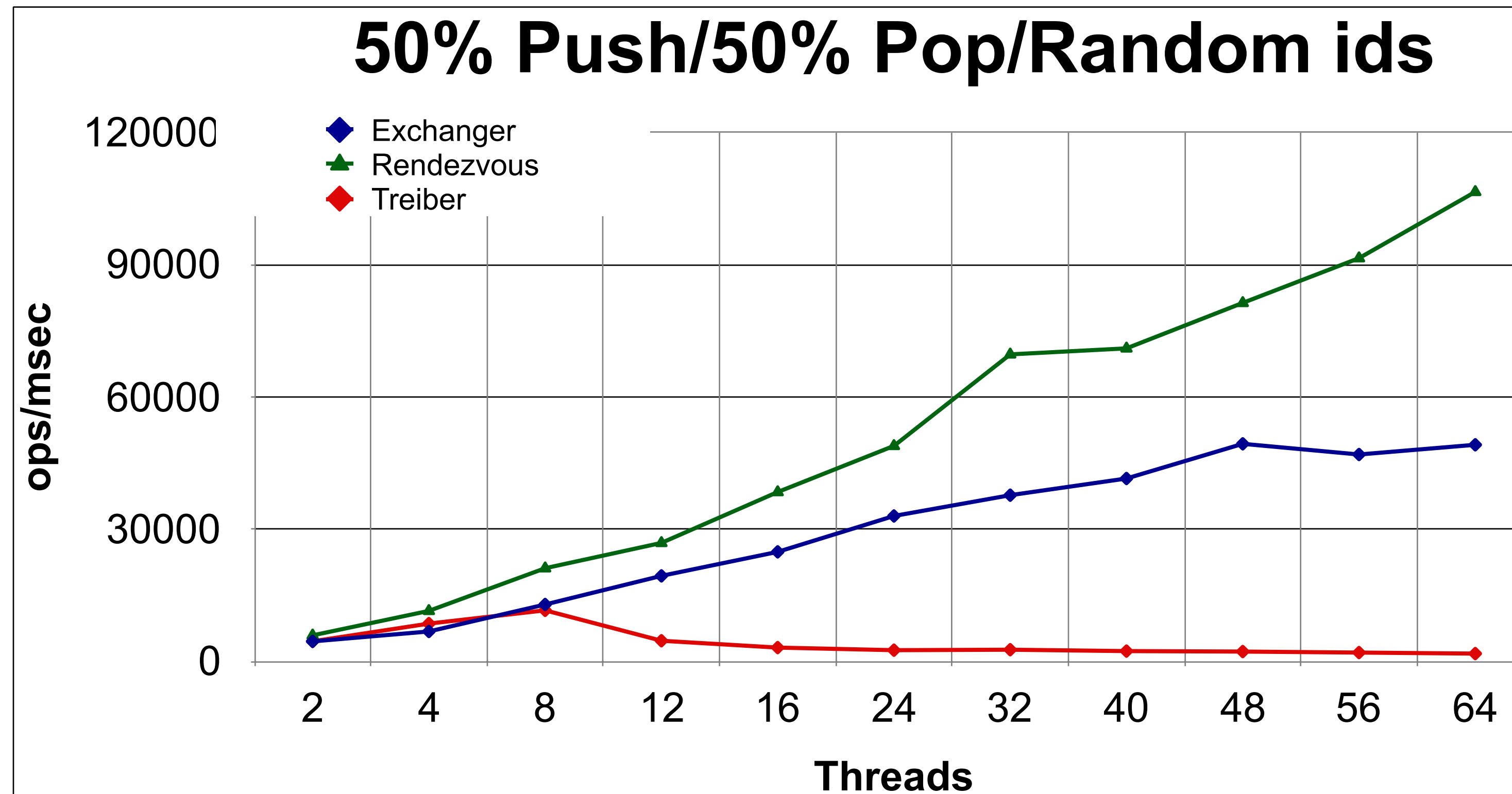


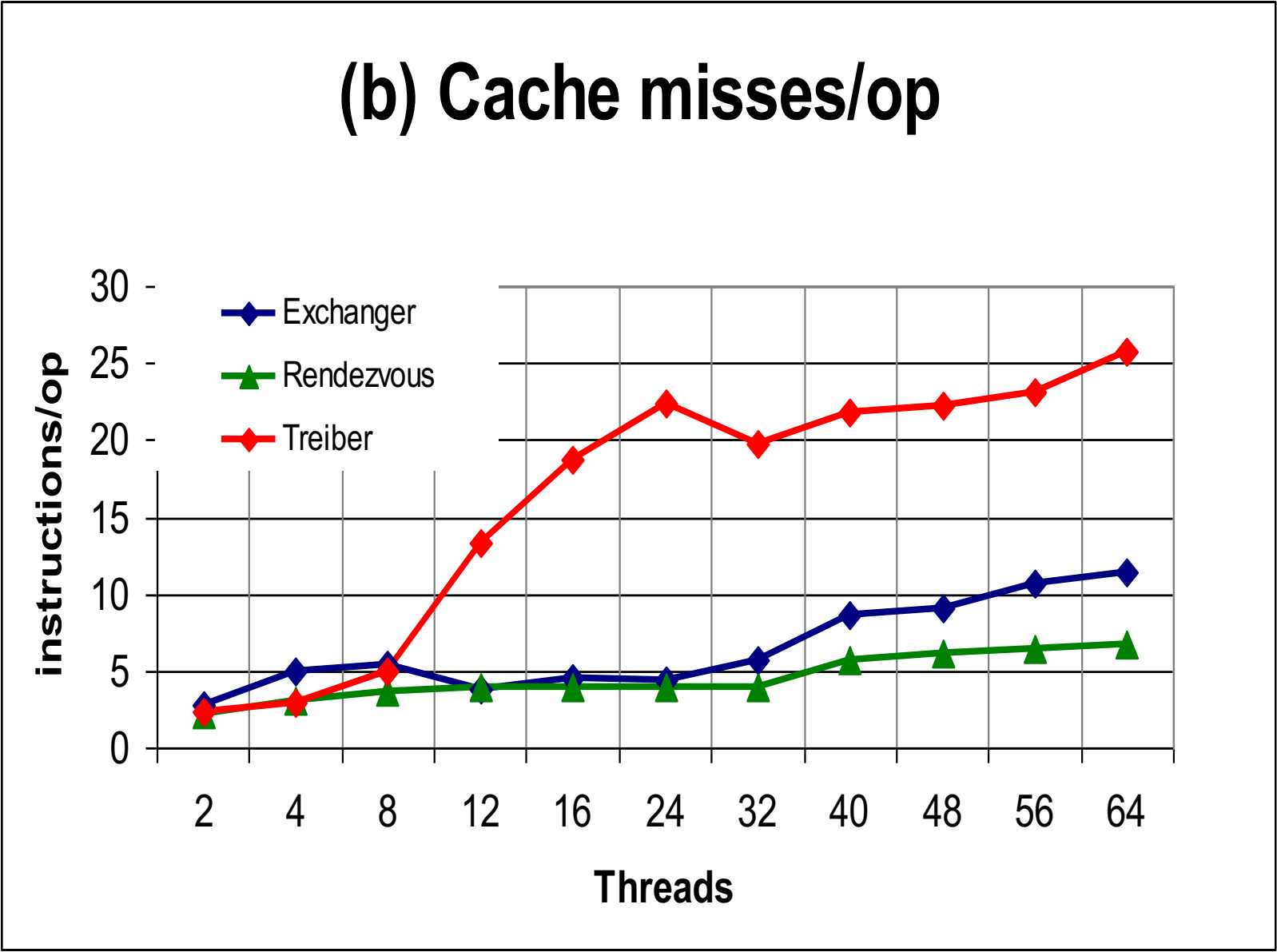
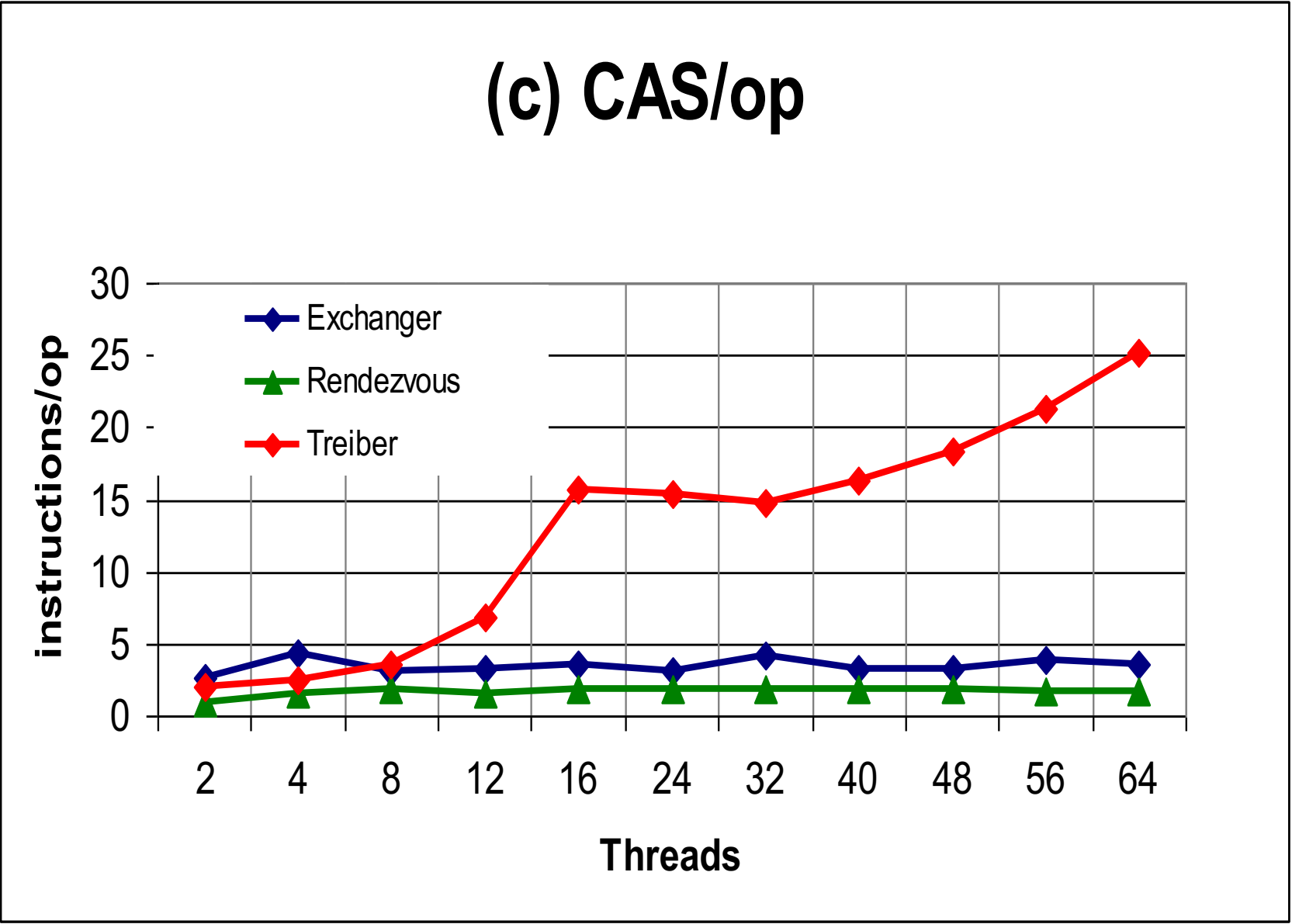
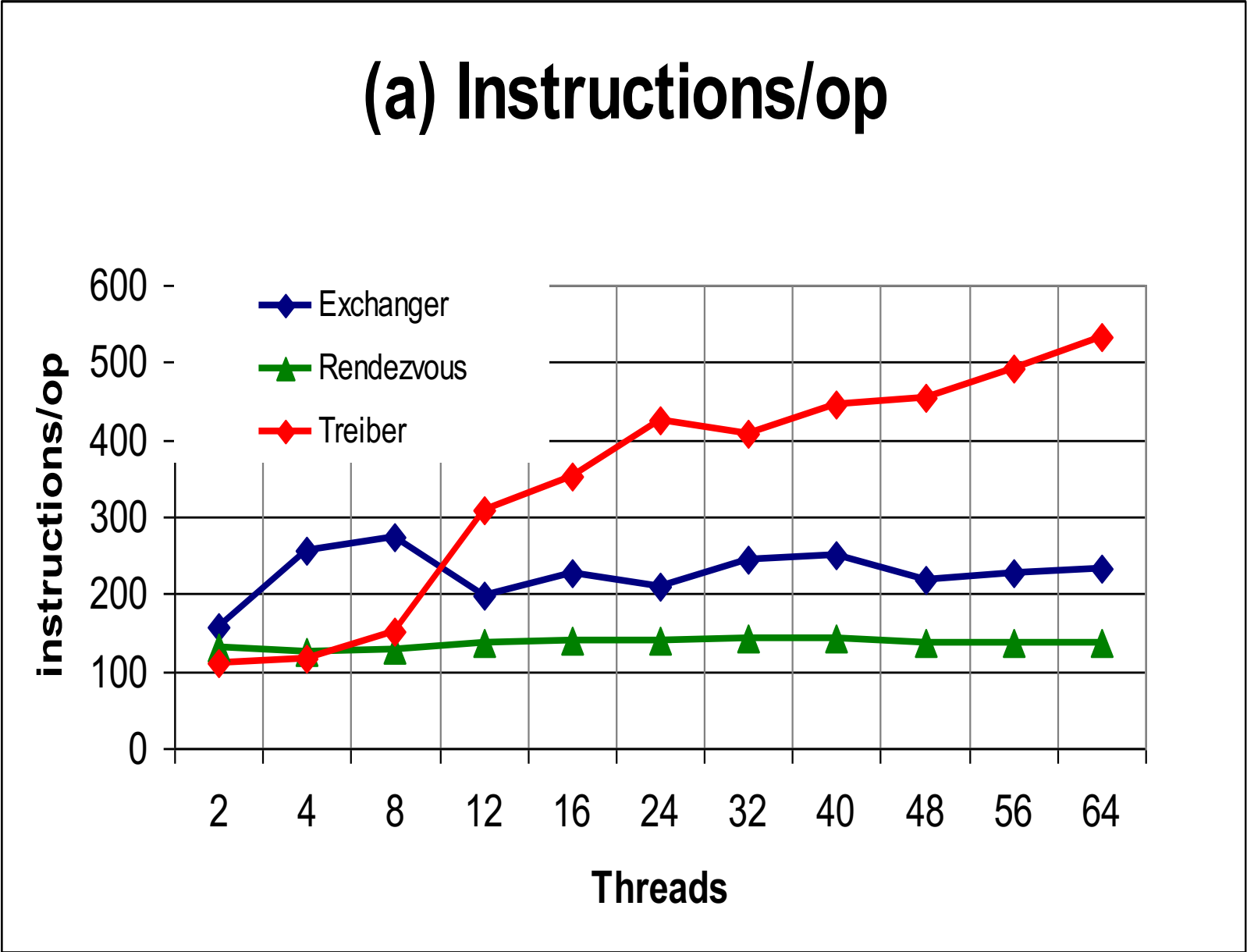
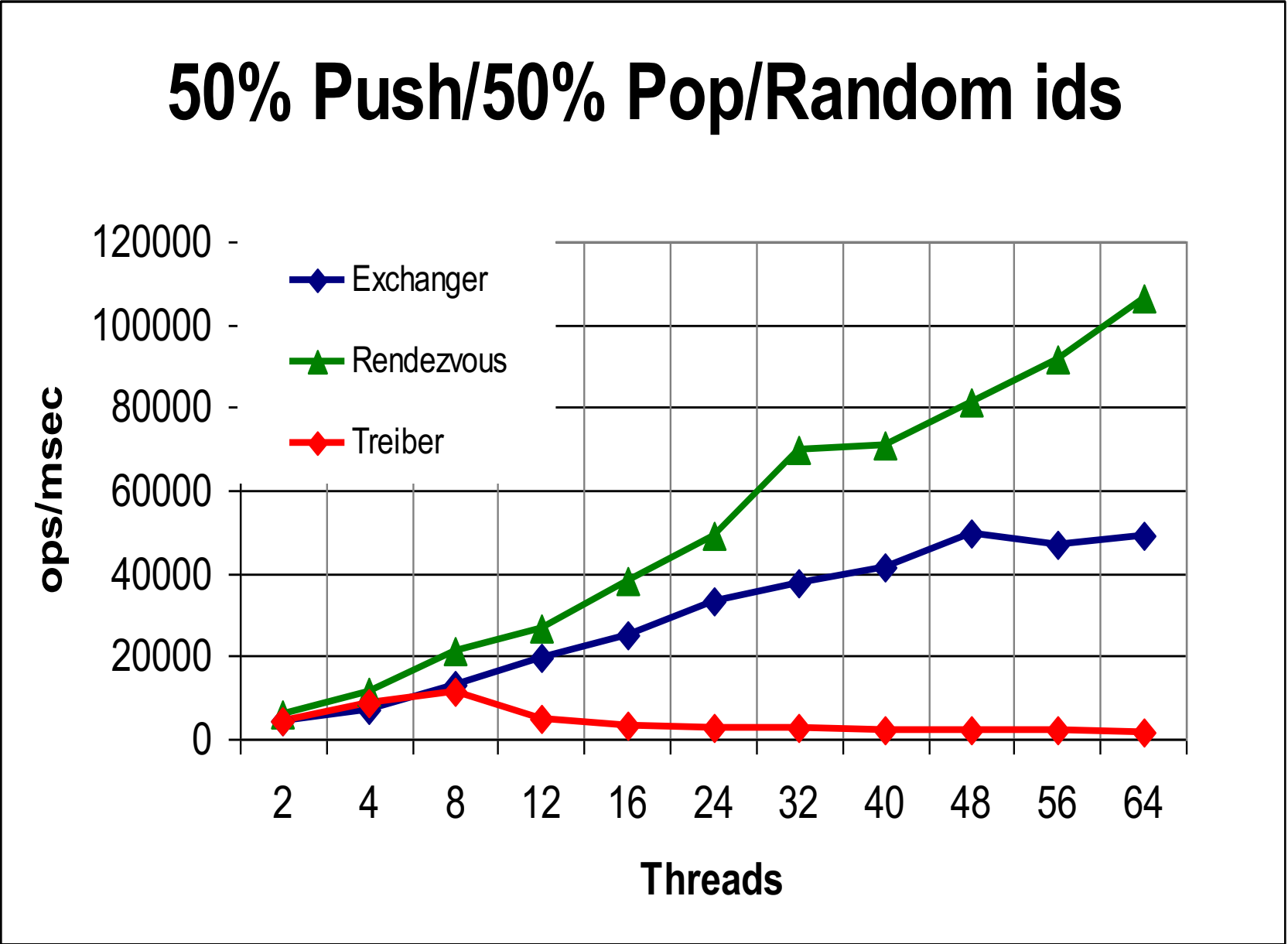
Pops find first vacant slot and spin.

Pushes hunt for pops.



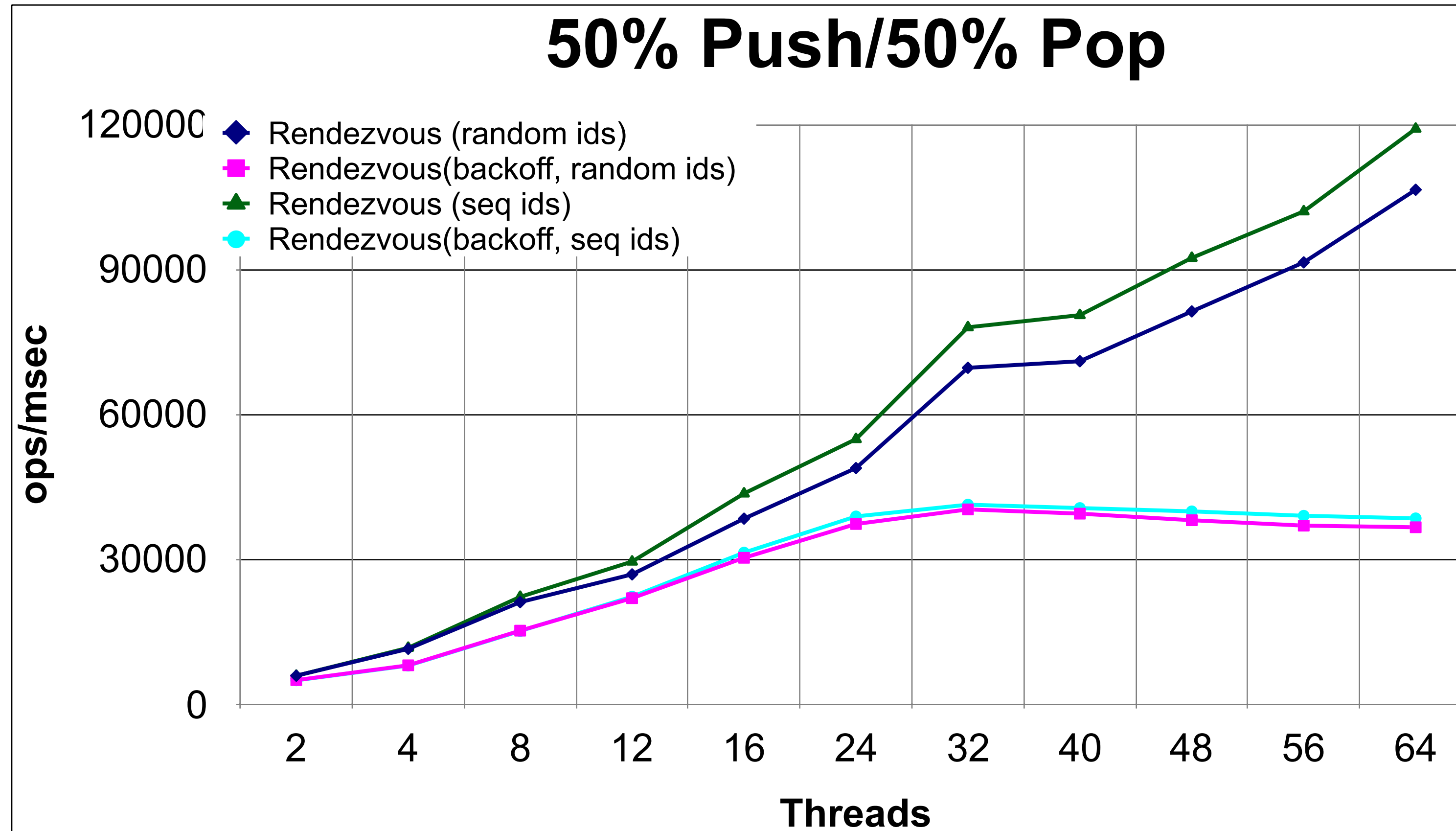
# Asymmetric vs. Symmetric





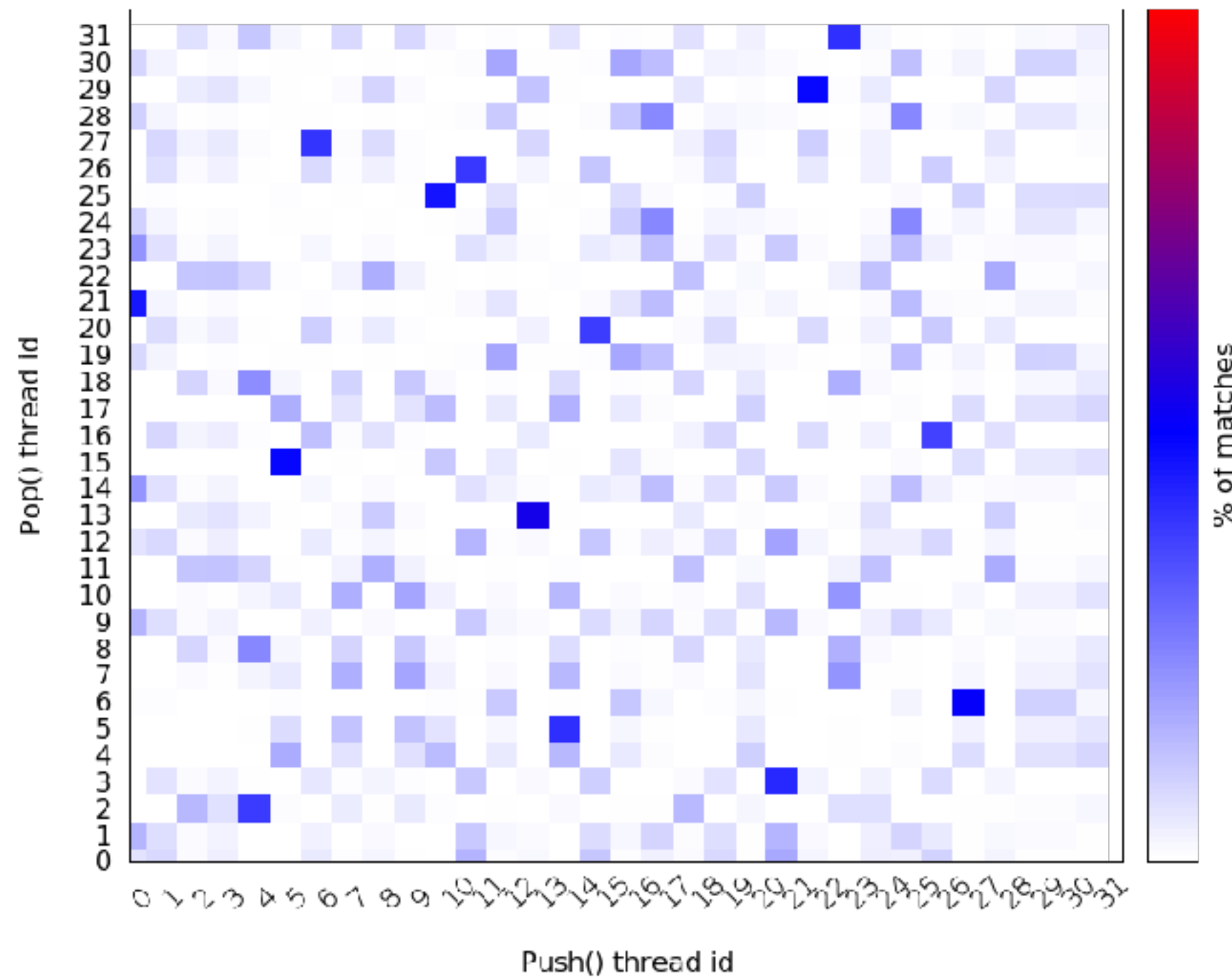


# Effect of Backoff and Slot Choice

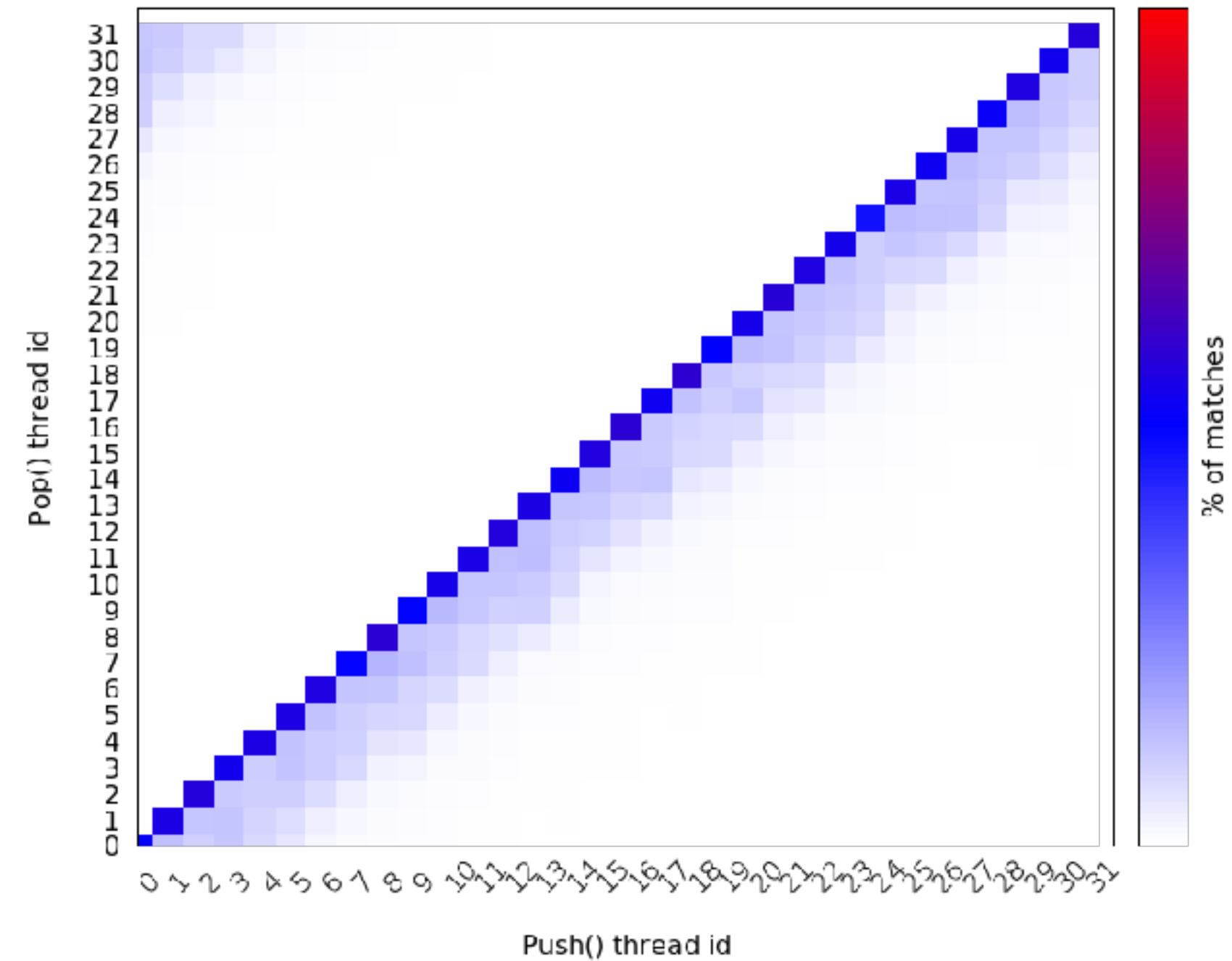


# Effect of Slot Choice

Random choice



Sequential choice



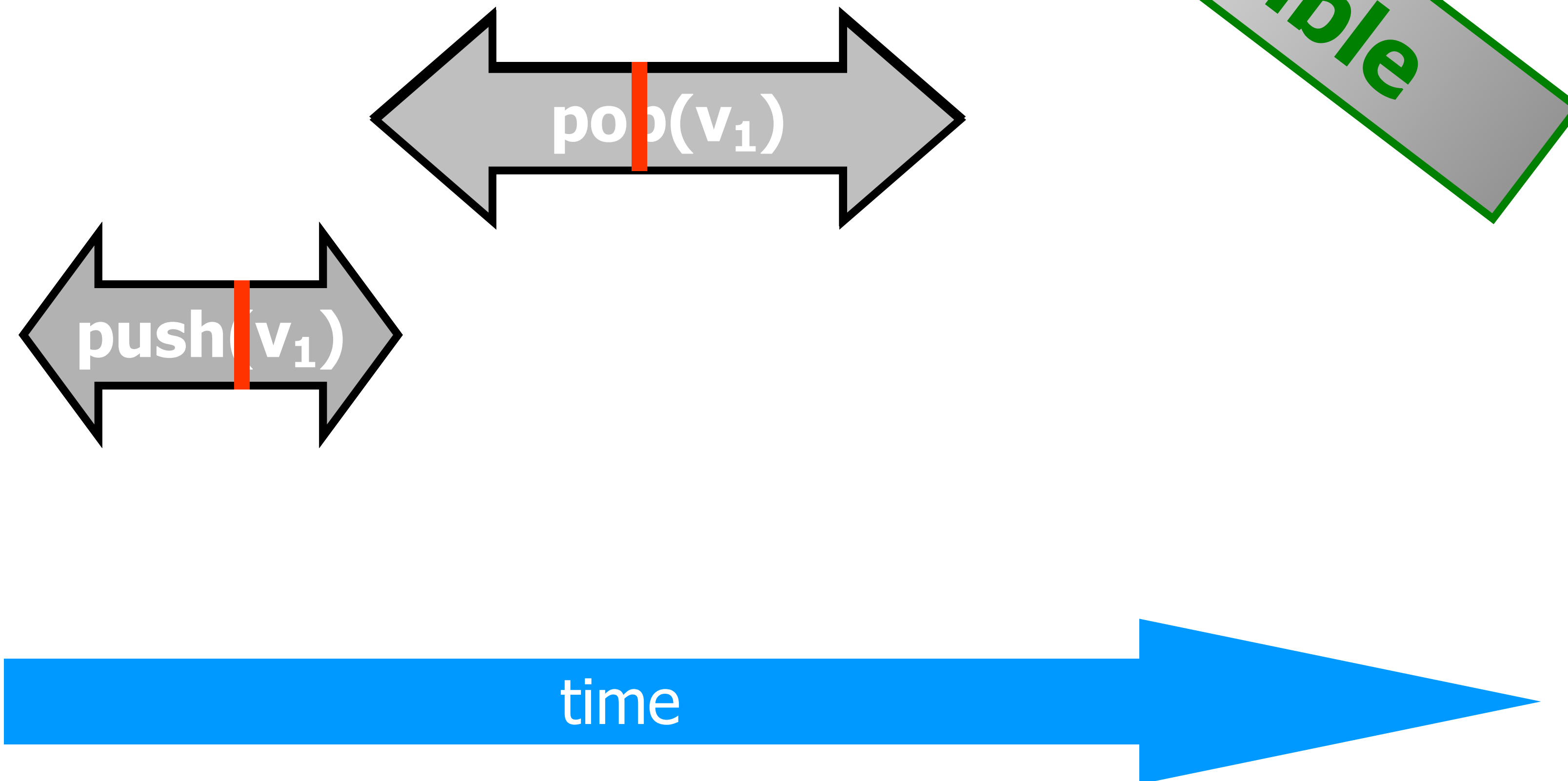
Darker shades mean more exchanges

# Linearizability

- **Un-eliminated calls**
  - **linearized as before**
- **Eliminated calls:**
  - **linearize pop() immediately after matching push()**
- **Combination is a linearizable stack**

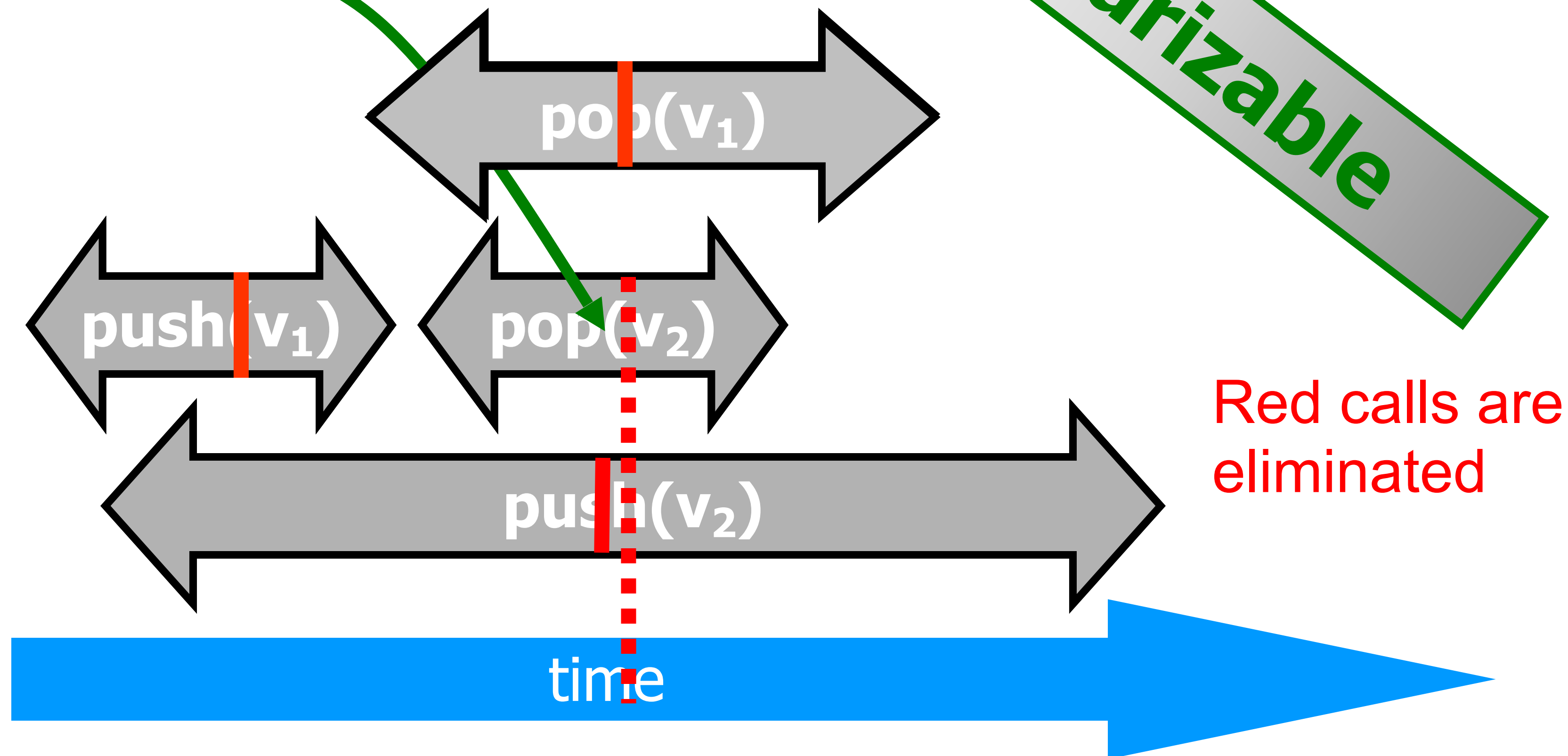
# Un-Eliminated Linearizability

linearizable



# Eliminated Linearizability

Collision  
Point



# Backoff Has Dual Effect

- Elimination introduces parallelism
- Backoff to array cuts contention on lock-free stack
- Elimination in array cuts down number of threads accessing lock-free stack

# Elimination Array

```
class EliminationArray[T: ClassTag] {  
  val duration = 10  
  private val size = ...  
  val exchangers = Array.fill(size) (new Exchanger[T])  
  val random = new Random()  
  
  def visit(value: T, range: Int): T = {  
    val slot = random.nextInt(range)  
    exchangers(slot).exchange(value, duration)  
  }  
}
```

# Elimination Array

```
class EliminationArray[T: ClassTag] {  
  val duration = 10  
  private val size = ...  
  val exchangers = Array.fill(size) (new Exchanger[T])  
  val random = new Random()  
  
  def visit(value: T,  
    val slot = random.nextInt(range)  
    exchangers(slot).exchange(value, duration)  
  }  
}
```

An array of *Exchangers*



# Digression: A Lock-Free Exchanger

```
class Exchanger[T] {  
  
    val slot = new AtomicStampedReference[T] (null, 0)  
  
    def exchange(myItem: T,  
                timeout: Long,  
                unit: TimeUnit): T = { ... }  
}
```

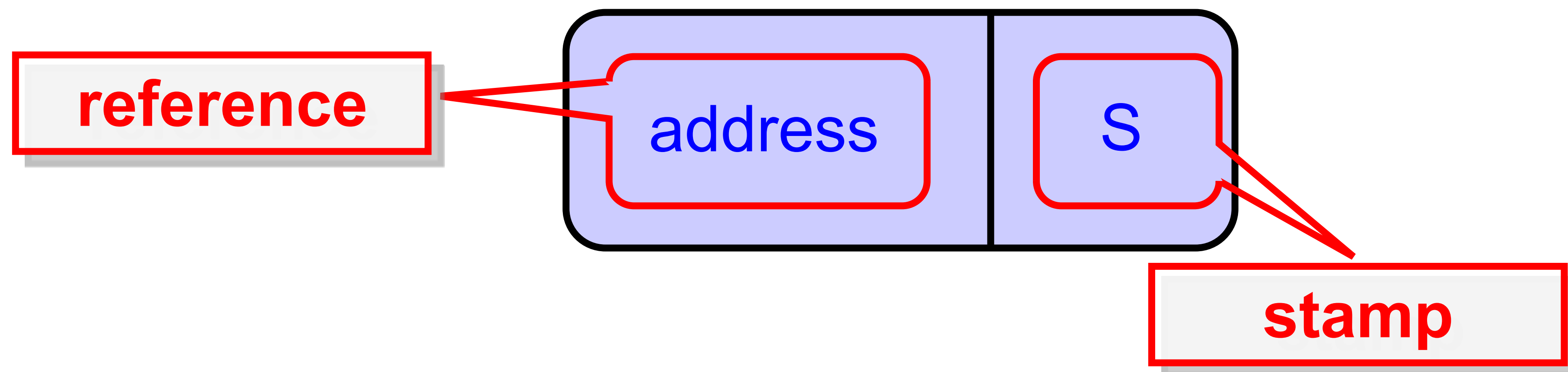
# A Lock-Free Exchanger

```
class Exchanger[T] {  
    val slot = new AtomicStampedReference[T] (null, 0)  
  
    def exchange(myItem: T,  
                timeout: Long,  
                unit: TimeUnit): T = { ... }  
}
```

Atomically modifiable  
reference + status

# Atomic Stamped Reference

- AtomicStampedReference **class**
  - Java.util.concurrent.atomic **package**
- In C or C++:



# Extracting Reference & Stamp

```
def get(stampHolder: Array[Int]): T
```

# Extracting Reference & Stamp

```
def get (stampHolder: Array[Int]) : T
```

Returns stamp at  
array index 0

Returns reference to  
object of type T

# Exchanger Status


```
val EMPTY = 0
```

```
val WAITING = 1
```

```
val BUSY = 2
```

# Exchanger Status

```
val EMPTY = 0  
val WAITING = 1  
val BUSY = 2
```



Nothing yet

# Exchange Status

```
val EMPTY = 0  
val WAITING = 1  
val BUSY = 2
```

Nothing yet

One thread is waiting  
for rendez-vous



# Exchange Status

```
val EMPTY = 0  
val WAITING = 1  
val BUSY = 2
```

Nothing yet

One thread is waiting  
for rendez-vous

Other threads busy  
with rendez-vous

# The Exchange

```
def exchange(myItem: T, timeout: Long): T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...         // someone waiting for me  
      case BUSY => ...            // others exchanging  
    }  
  }  
}
```

```
def exchange(myItem: T, timeout: Long): T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...      // slot is free  
      case WAITING => ...    // someone waiting for me  
      case BUSY => ...       // others exchanging  
    }  
  }  
}
```

# The Exchange

```
def exchange(myItem: T, timeout: Long) : T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound) Item and timeout  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...         // someone waiting for me  
      case BUSY => ...            // others exchanging  
    }  
  }  
}
```

# The Exchange

```
def exchange(myItem: T, timeout: Long): T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...        // someone waiting for me  
      case BUSY => ...           // others exchanging  
    }  
  }  
}
```

Array holds status

# The Exchange

```
def exchange(myItem: T, timeout: Long): T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...         // someone waiting for me  
      case BUSY => ...            // others exchanging  
    }  
  }  
}
```

Loop until timeout

# The Exchange

```
def exchange(myItem: T, timeout: Long): T = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {  
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)  
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...         // someone waiting for me  
      case BUSY => ...            // others exchanging  
    }  
  }  
}
```

Get other's item and status

# The Exchange

**An *Exchanger* has three possible states**

```
def exchangeItem(slot: AtomicReference[A]): Unit = {  
  val timeBound = System.nanoTime() + timeout  
  val stapmholder = Array(EMPTY)  
  while (true) {
```

```
    if (System.nanoTime() > timeBound)  
      throw new TimeoutException  
    var yrItem = slot.get(stapmholder)
```

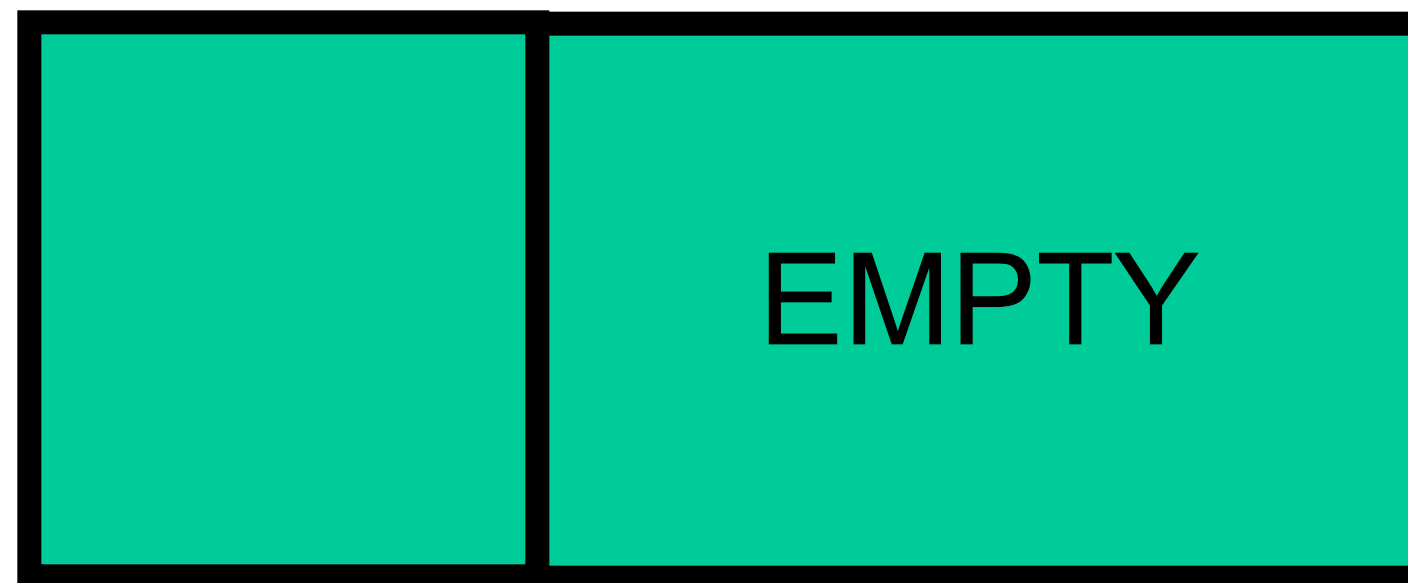
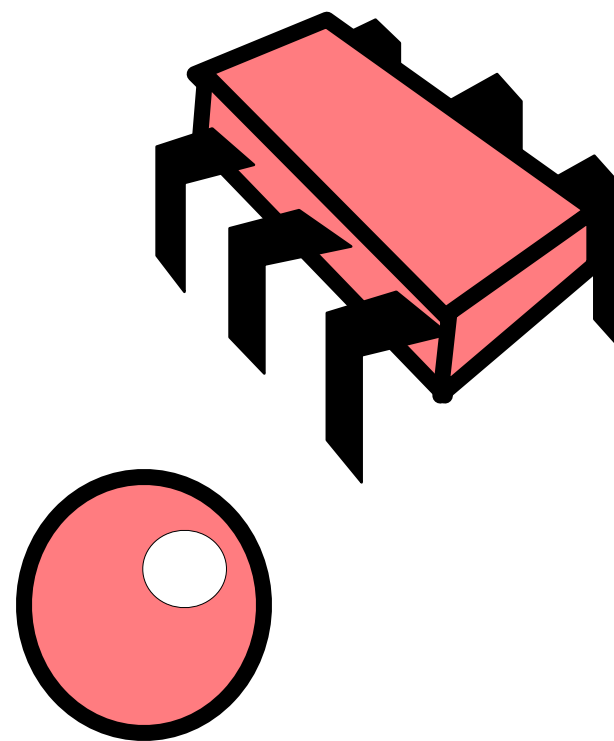
```
    stapmholder(0) match {  
      case EMPTY => ...           // slot is free  
      case WAITING => ...         // someone waiting for me  
      case BUSY => ...            // others exchanging  
    }
```

```
  }
```

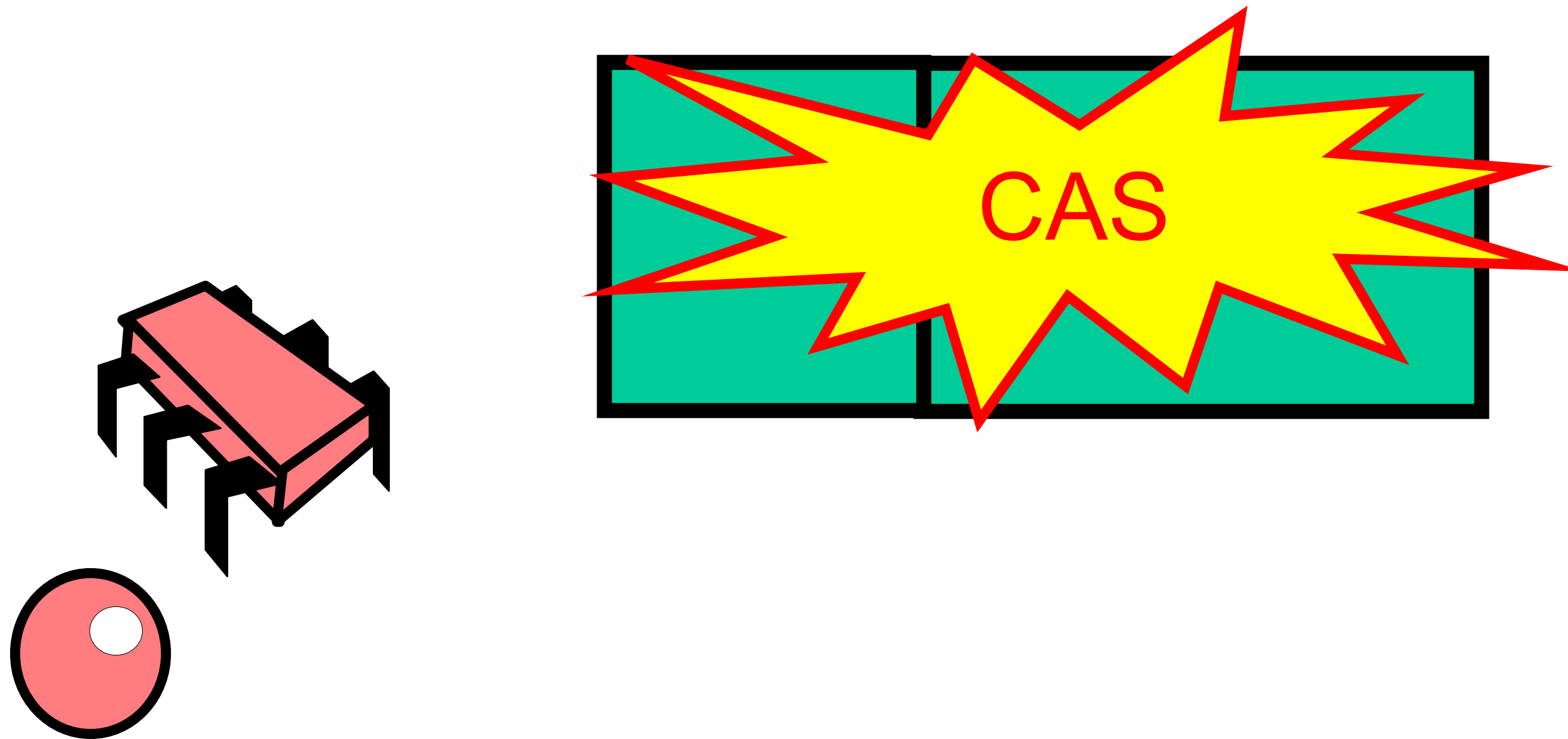
```
}
```



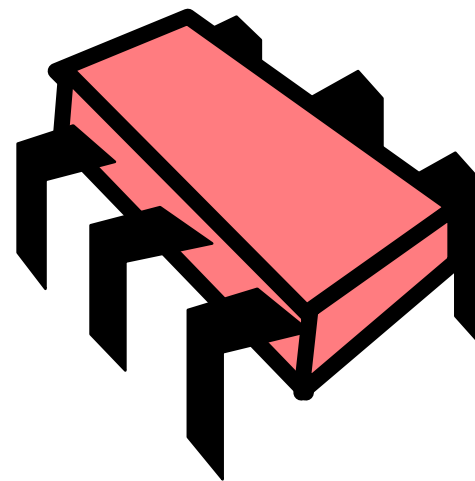
# Lock-free Exchanger



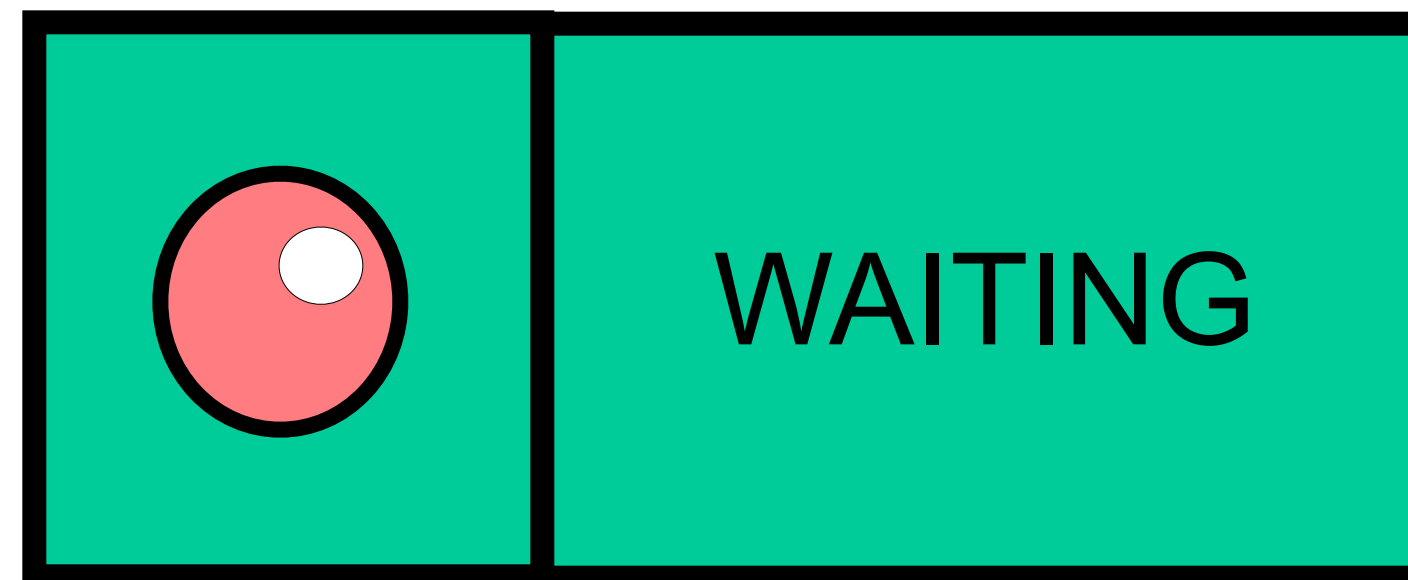
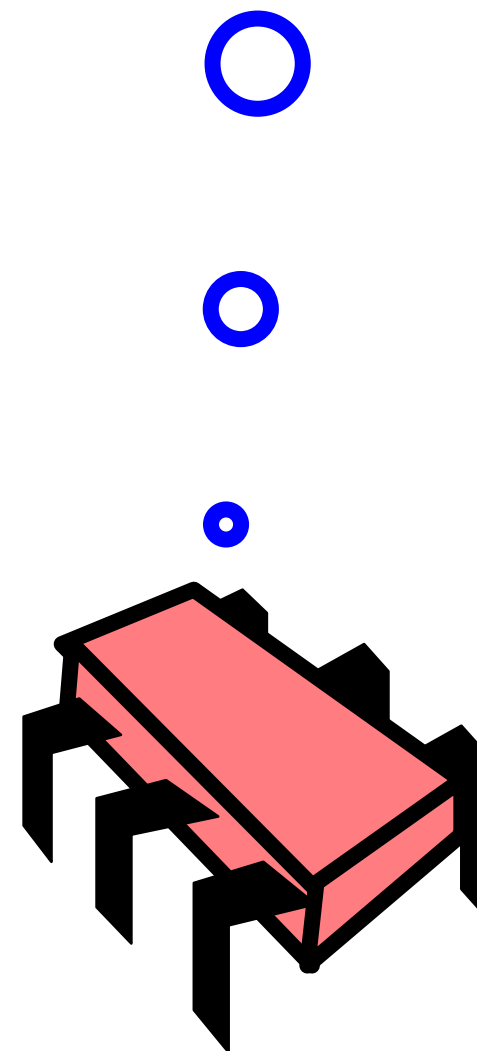
# Lock-free Exchanger



# Lock-free Exchanger



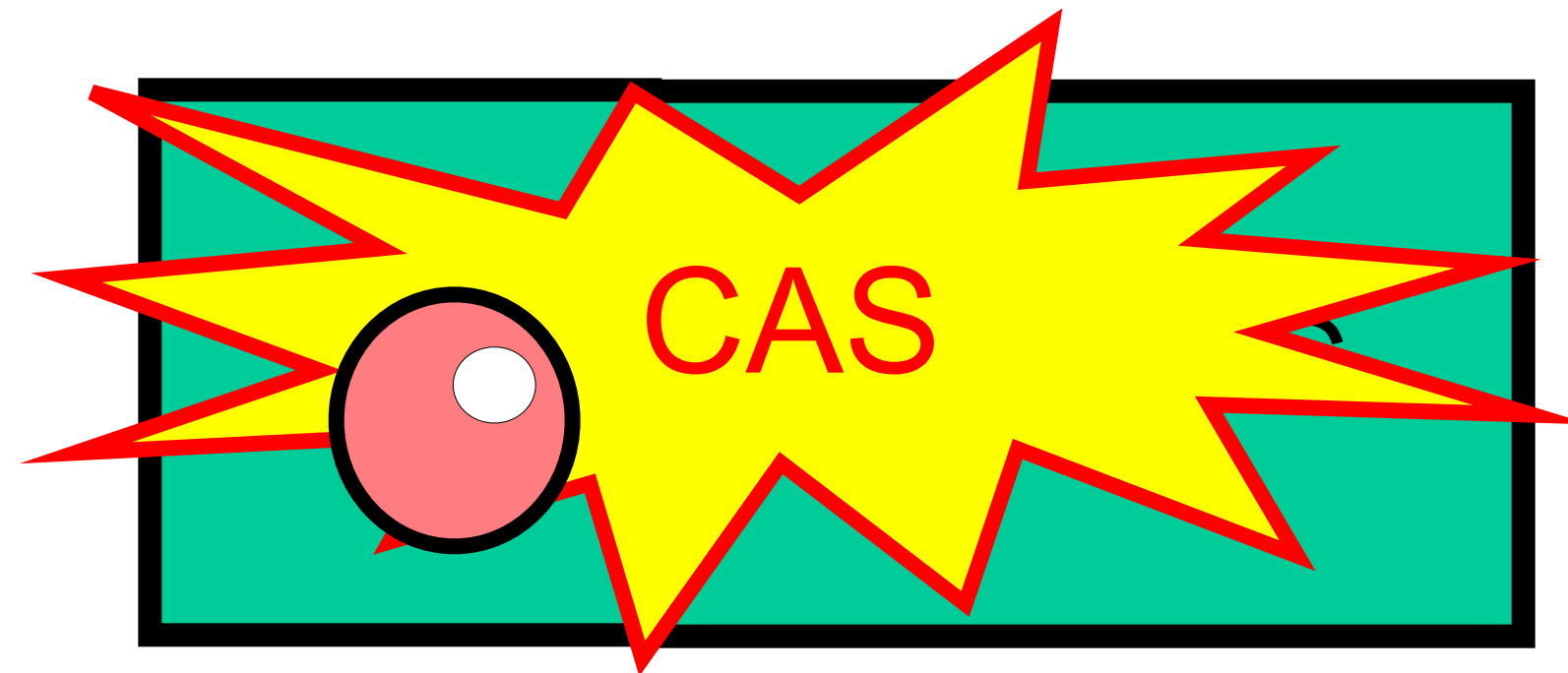
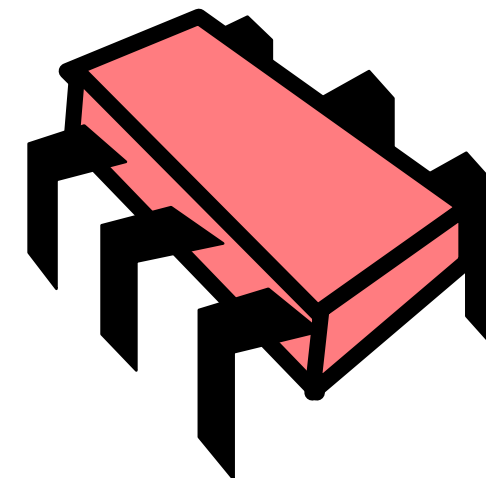
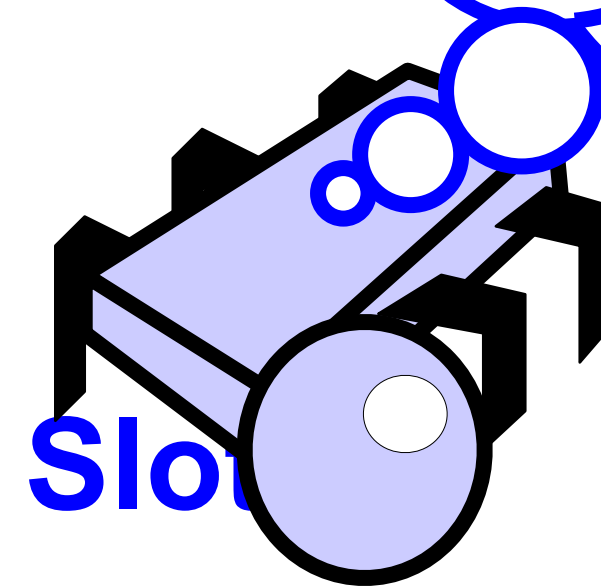
# Lock-free Exchanger



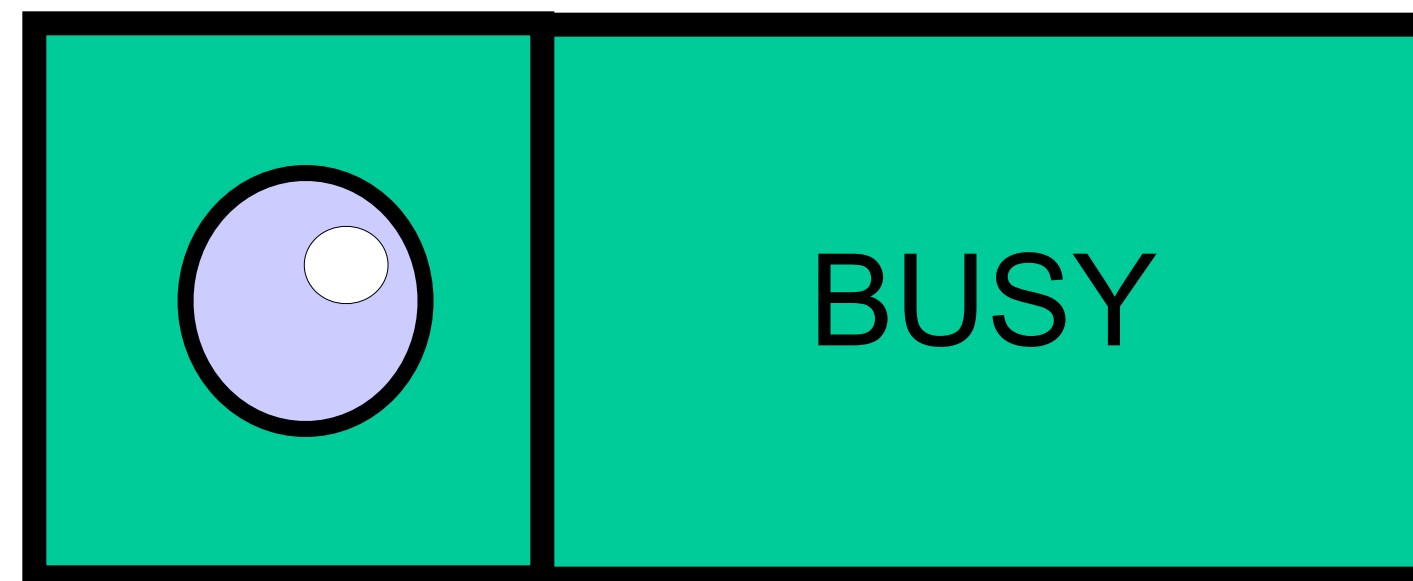
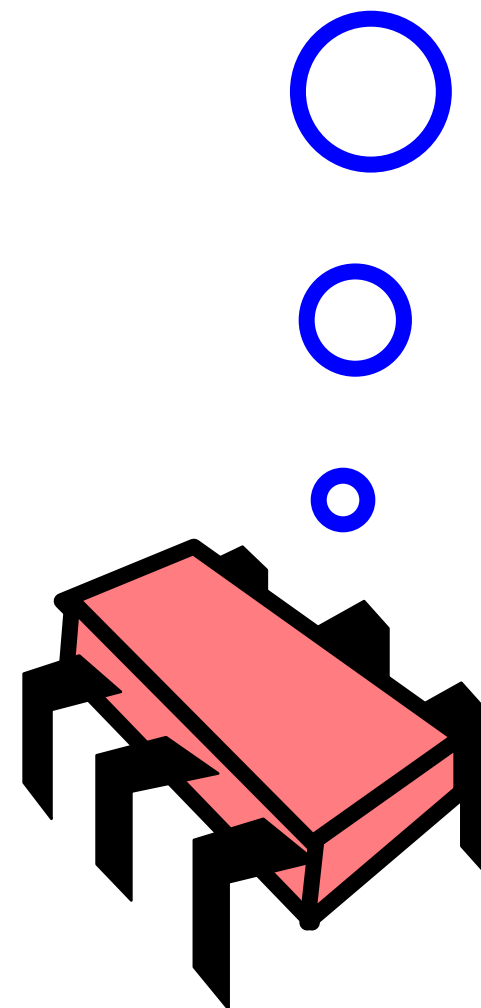
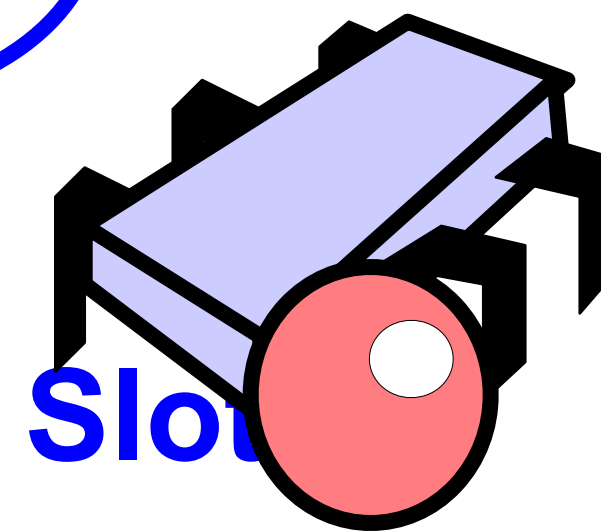
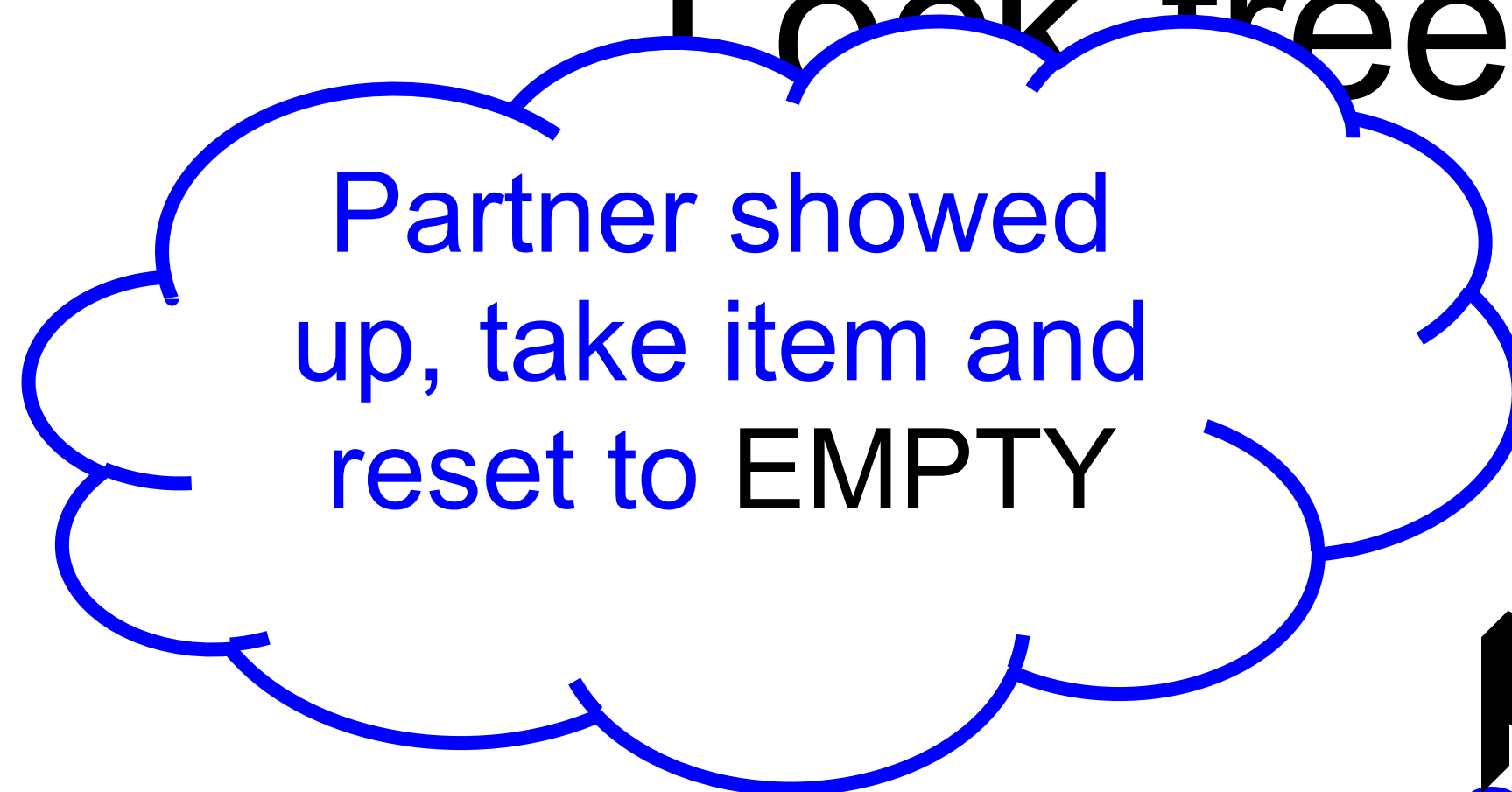
# Lock-free Ex

Still waiting ...

Try to exchange  
item and set  
status to **BUSY**



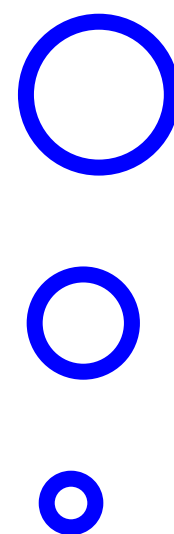
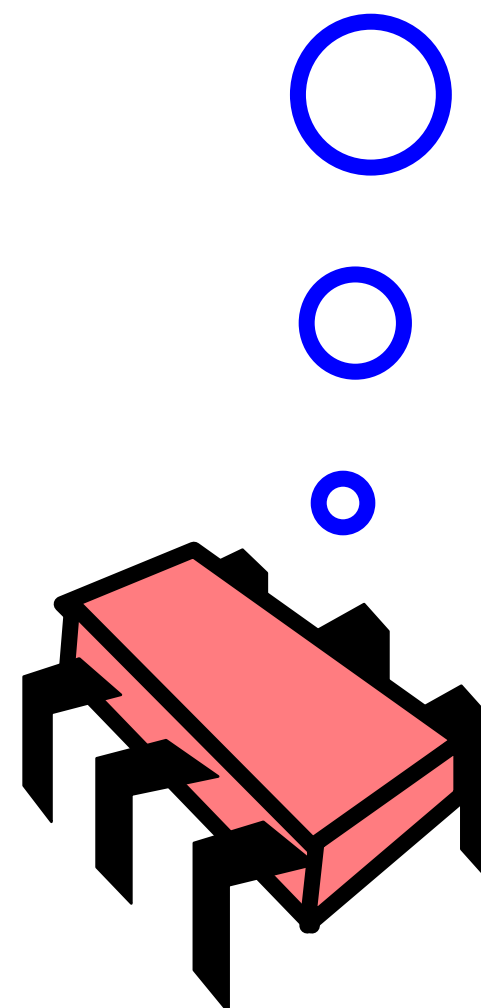
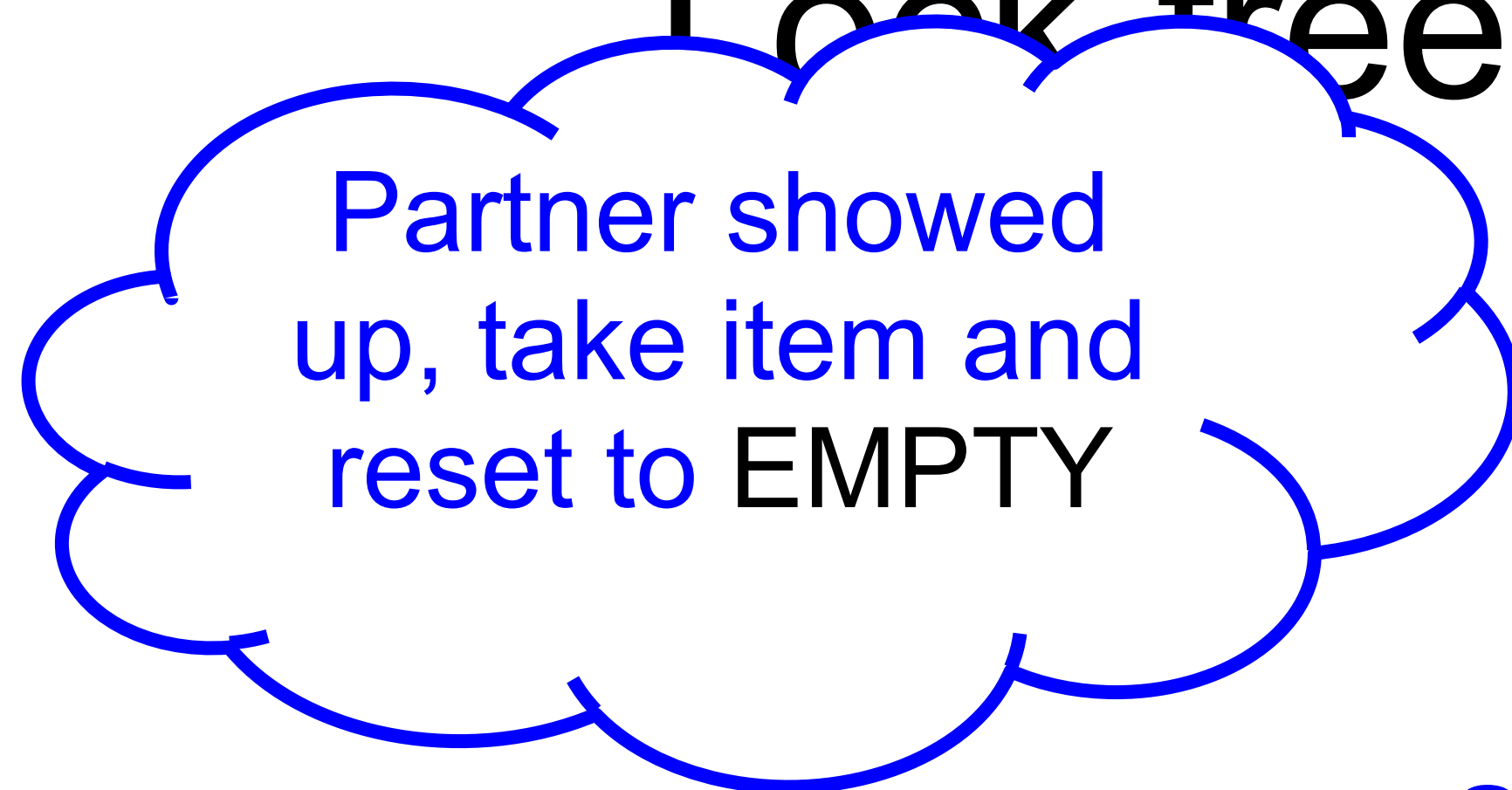
# Look-free Exchanger



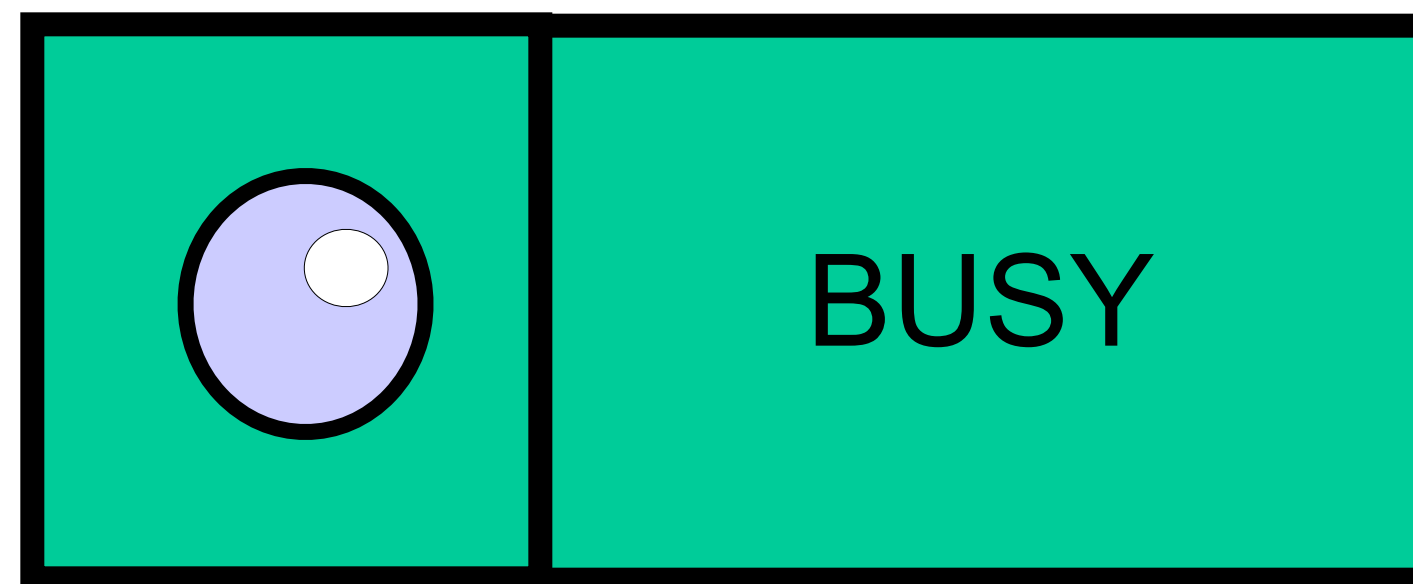
item

status

# Look-free Exchanger



Slot



item

status

# Exchanger State EMPTY

```
case EMPTY =>
    if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
        while (System.nanoTime() < timeBound) {
            yrItem = slot.get(stapmholder)
            if (stapmholder(0) == BUSY) {
                slot.set(null, EMPTY)
                return yrItem
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
            throw new TimeoutException
        } else {
            yrItem = slot.get(stapmholder)
            slot.set(null, EMPTY)
            return yrItem
        }
    }
```



# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound) {
      yrItem = slot.get(stapmholder)
      if (stapmholder(0) == BUSY) {
        slot.set(null, EMPTY)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new TimeoutException
    } else {
      yrItem = slot.get(stapmholder)
      slot.set(null, EMPTY)
      return yrItem
    }
  }
}
```

Try to insert *myItem* and  
change state to *WAITING*

# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(vrItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound) {
      yrItem = slot.get(stapmholder)
      if (stapmholder(0) == BUSY) {
        slot.set(null, EMPTY)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new InterruptedException()
    } else {
      yrItem = slot.get(stapmholder)
      slot.set(null, EMPTY)
      return yrItem
    }
  }
}
```

Spin until either  
*myItem* is taken or timeout

# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound) {
      yrItem = slot.get(stapmholder)
      if (stapmholder(0) == BUSY) {
        slot.set(null, EMPTY)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new TimeoutException
    } else
      yrItem = slot.get(stapmholder)
      slot.set(null, yrItem)
      return yrItem
    }
  }
```

*myItem was taken,  
so return yrItem  
that was put in its place*

# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(myItem, EMPTY, WAITING)) {
    while (system.nanoTime() < timeBound) {
      if (stapmholder() == BUSY) {
        slot.set(myItem, EMPTY)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new TimeoutException
    } else {
      yrItem = slot.get(stapmholder)
      slot.set(null, EMPTY)
      return yrItem
    }
  }
```

Otherwise we ran out of time,  
try to reset status to EMPTY  
and time out

if (slot.compareAndSet(myItem, **null**, WAITING, EMPTY)) {  
 **throw new** TimeoutException

# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound) {
      yrItem = slot.get(stapmholder)
      if (stapmholder(0) == BUSY) {
        slot.set(yrItem, WAITING)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new TimeoutException()
    }
    else {
      yrItem = slot.get(stapmholder)
      slot.set(null, EMPTY)
      return yrItem
    }
  }
}
```

If reset failed,  
someone showed up after all,  
so take that item

# Exchanger State EMPTY

```
case EMPTY =>
  if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
    while (System.nanoTime() < timeBound) {
      yrItem = slot.get(stapmholder)
      if (stapmholder(0) == BUSY) {
        slot.set(null, EMPTY)
        return yrItem
      }
    }
    if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
      throw new TimeoutException
    } else {
      yrItem = slot.get(stapmholder)
      slot.set(null, EMPTY)
      return yrItem
    }
  }
```

Clear slot and take that item

# Exchanger State EMPTY

```
case EMPTY =>
    if (slot.compareAndSet(yrItem, myItem, EMPTY, WAITING)) {
        while (System.nanoTime() < timeBound) {
            yrItem = slot.get(stapmholder)
            if (stapmholder(0) == BUSY) {
                slot.
                retur
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING, EMPTY)) {
            throw new TimeoutException()
        } else {
            yrItem = slot.get(stapmholder)
            slot.set(null, EMPTY)
            return yrItem
        }
    }
}
```

If initial CAS failed,  
then someone else changed status  
from EMPTY to WAITING,  
so retry from start

# States WAITING and BUSY

```
case WAITING =>
    if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY)) {
        return yrItem
    }
case BUSY =>
case x =>
    throw new Exception("Cannot happen")
```



```
case WAITING =>
    if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY)) {
        return yrItem
    }
case BUSY =>
case x =>
    throw new Exception("Cannot happen")
```

# States WAITING and BUSY

```
case WAITING =>
  if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY)) {
    return yrItem
  }
case BUSY =>
case x =>
  throw new Exception("Cannot happen")
```

someone is waiting to exchange,  
so try to CAS my item in  
and change state to BUSY

# States WAITING and BUSY

```
case WAITING =>  
    if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY)) {  
        return yrItem  
    }  
case BUSY =>  
case x =>  
    throw new Exception("Cannot happen")
```

If successful, return other's item,  
otherwise someone else took it,  
so try again from start

# States WAITING and BUSY

```
case WAITING =>
  if (slot.compareAndSet(yrItem, myItem, WAITING, BUSY)) {
    return yrItem
  }
case BUSY =>
case x =>
  throw new Exception("cannot happen")
```

If BUSY,  
other threads exchanging,  
so start again

# The Exchanger Slot

- Exchanger is lock-free
- Because the only way an exchange can fail is if others repeatedly succeeded or no-one showed up
- The slot we need does not require symmetric exchange

# Back to the Stack: the Elimination Array

```
class EliminationArray[T: ClassTag] {  
  ...  
  def visit(value: T, range: Int): T = {  
    val slot = random.nextInt(range)  
    exchanger(slot).exchange(value, duration)  
  }  
}
```

# Elimination Array

```
class EliminationArray[T: ClassTag] {  
  ...  
  def visit(value: T, range: Int): T = {  
    val slot = random.nextInt(range)  
    exchanger(slot).exchange(value, duration)  
  }  
}
```

visit the elimination array  
with fixed value and range

# Elimination Array

```
class EliminationArray[T: ClassTag] {  
  ...  
  def visit(value: T, range: Int): T = {  
    val slot = random.nextInt(range)  
    exchanger(slot).exchange(value, duration)  
  }  
}
```

Pick a random array entry



# Elimination Array

```
class EliminationArray
...
def visit(value: T, range: Int): T = {
  val slot = random.nextInt(range)
  exchanger(slot).exchange(value, duration)
}
}
```

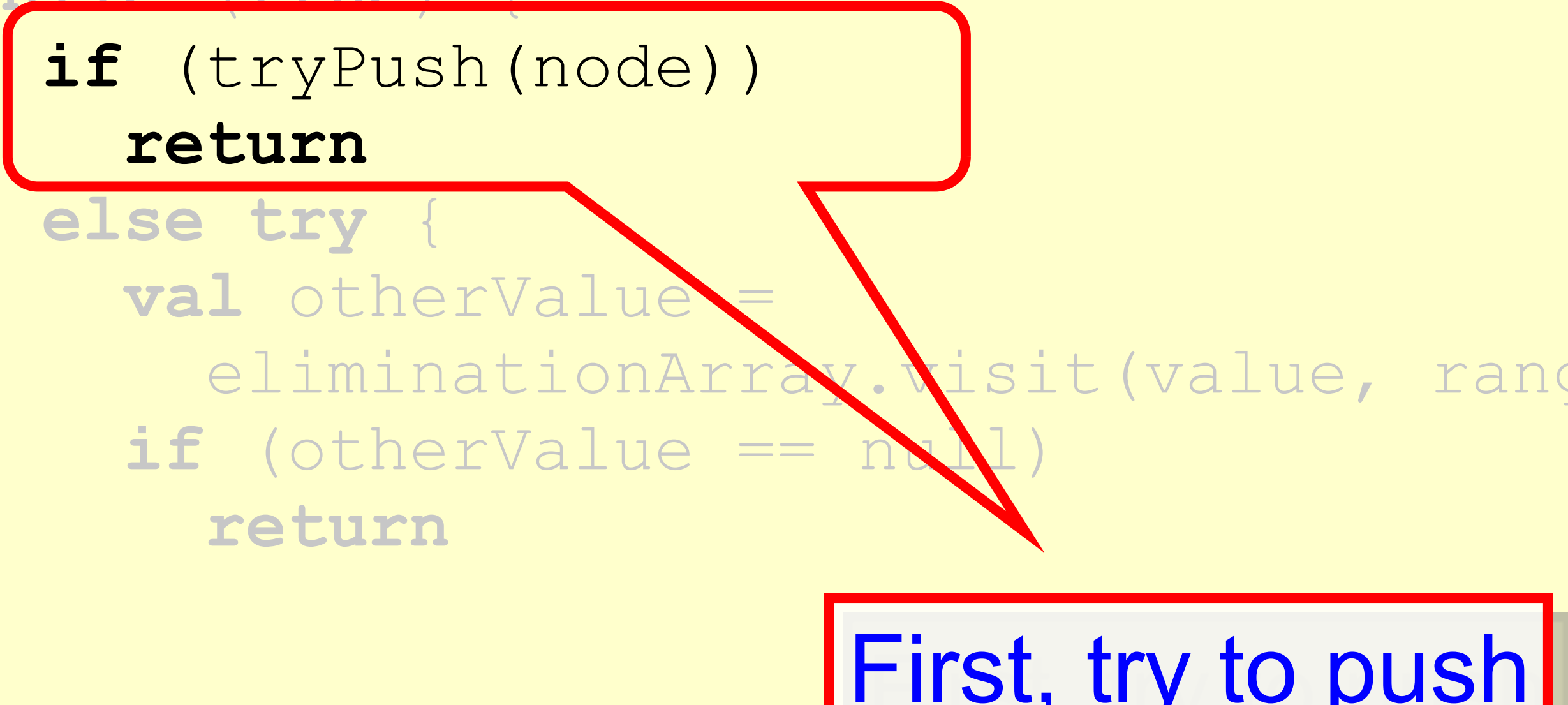
Exchange value or time out

# Elimination Stack Push

```
override def push(value: T): Unit = {  
    ...  
    while (true) {  
        if (tryPush(node))  
            return  
        else try {  
            val otherValue =  
                eliminationArray.visit(value, rangePolicy.getRange)  
            if (otherValue == null)  
                return  
        }  
    }  
}
```

# Elimination Stack Push

```
override def push(value: T): Unit = {  
  ...  
  while (true) {  
    if (tryPush(node))  
      return  
    else try {  
      val otherValue =  
        eliminationArray.visit(value, rangePolicy.getRange)  
      if (otherValue == null)  
        return  
    }  
  }  
}
```



First, try to push

# Elimination Stack Push

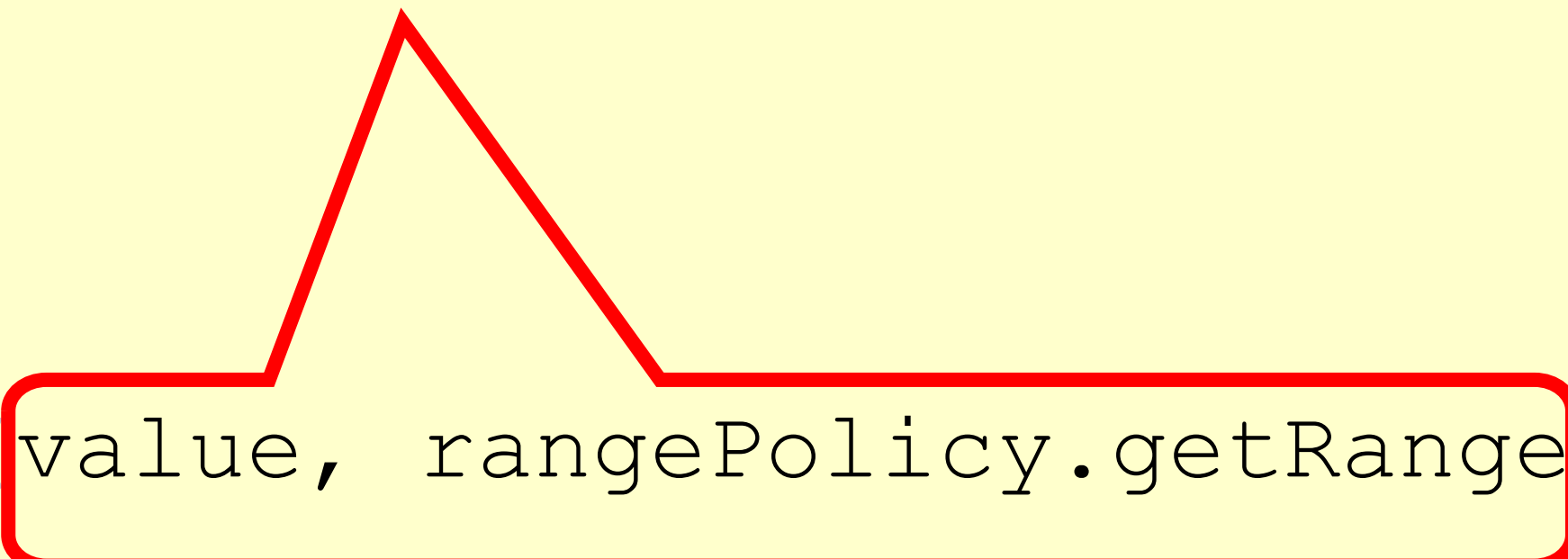
```
override def push(value: T): Unit = {  
  ...  
  while (true) {  
    if (tryPush(node))  
      return  
    else try {  
      val otherValue =  
        eliminationArray.visit(value, rangePolicy.getRange)  
      if (otherValue == null)  
        return  
    }  
  }  
}
```

If I failed, backoff & try to eliminate

# Elimination Stack Push

```
override def push(value: T): Unit = {  
  ...  
  while (true) {  
    if (tryPush(node))  
      return  
    else try {  
      val otherValue =  
        eliminationArray.visit(value, rangePolicy.getRange)  
      if (otherValue == null)  
        return  
    }  
  }  
}
```

Value pushed and range to try



# Elimination Stack Push

```
override def push(value: T): Unit = {  
  ...  
  while (true) {  
    if (tryPush(value)) {  
      return  
    } else try {  
      val otherValue =  
        eliminationArray.visit(value, rangePolicy.getRange)  
      if (otherValue == null) {  
        return  
      }  
    }  
  }  
}
```

Only **pop ()** leaves null,  
so elimination was successful

# Elimination Stack Push

```
override def push(value: T): Unit = {  
  ...  
  while (true) {  
    if (tryPush(node))  
      return  
    else try {  
      val otherValue =  
        eliminationArray.visit(value, rangePolicy.getRange)  
      if (otherValue == null)  
        return  
    }  
  }  
}
```

Otherwise, retry **push()** on lock-free stack

}

# Elimination Stack Pop

```
override def pop(): T = {  
  while (true) {  
    val returnNode = tryPop()  
    if (returnNode != null) {  
      return returnNode.value  
    } else try {  
      val otherValue =  
        eliminationArray.visit(null, rangePolicy.getRange)  
      if (otherValue != null) {  
        return otherValue  
      }  
    }  
  }  
}
```



# Elimination Stack Pop

```
override def pop(): T = {  
  while (true) {  
    if (returnNode != null) {  
      return returnNode.value  
    } else try {  
      val otherValue =  
        eliminationArray.visit(null, rangePolicy.getRange)  
      if (otherValue != null) {  
        return otherValue  
      }  
    }  
  }  
}
```

If value not null, other thread is a **push()**,  
so elimination succeeded

# Summary

- We saw both lock-based and lock-free implementations of
- queues and stacks
- Don't be quick to declare a data structure inherently sequential
  - Linearizable stack is not inherently sequential (though it is in the worst case)
- ABA is a real problem, pay attention



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.