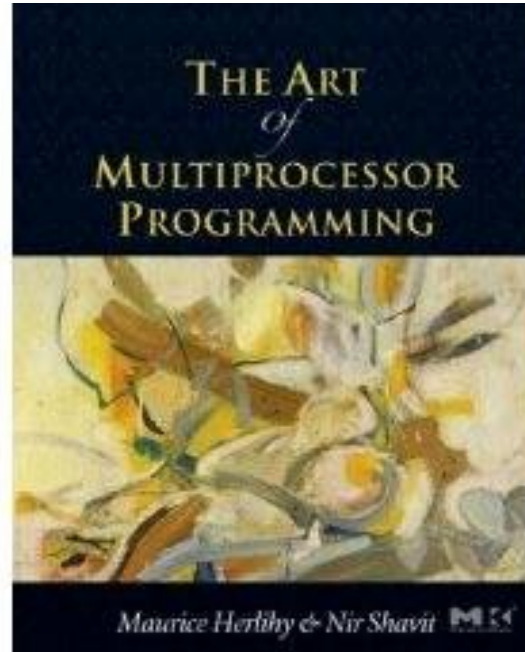


# COS 226

## Chapter 11

## Concurrent Stacks and Elimination

# Acknowledgement



- Some of the slides are taken from the companion slides for “The Art of Multiprocessor Programming” by Maurice Herlihy & Nir Shavit



# Stacks

- Methods

- pop()

- push(x)

- Last-in-First-out (LIFO) order



# Concurrent Stacks

- Do stacks provide opportunity for concurrency?
  - `push()` and `pop()` calls access only the top of the stack
- However, stacks are not inherently sequential



# Lock-Free Stacks

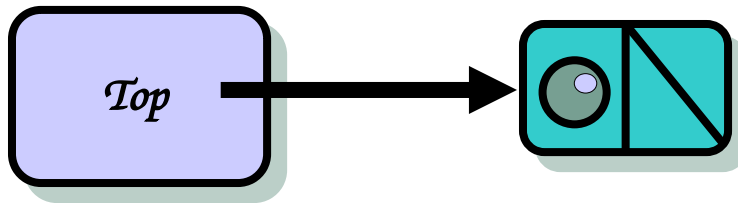
- Linked-list implementation
- top field points to the first node
- pop() from an empty list throws an exception
- push() forms a new node



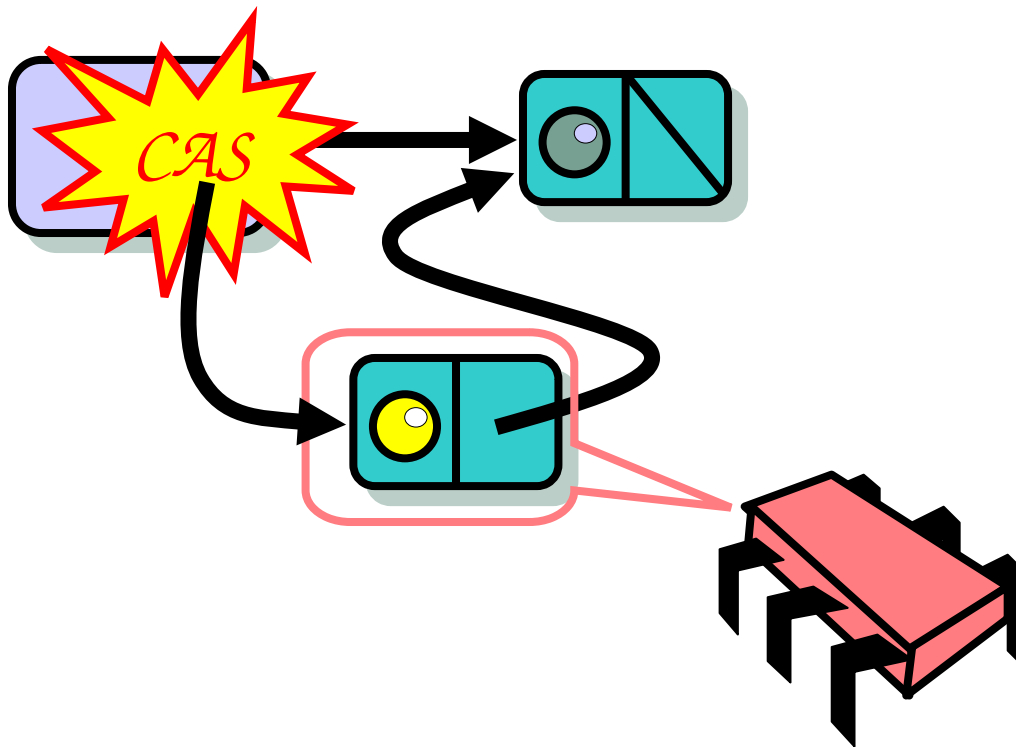
# Lock-free Stacks

- Lock-free = atomic
- Use compareAndSet methods to change the link values in the list

# Empty Stack

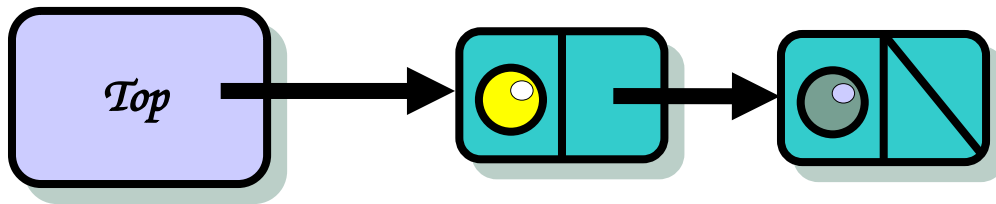


# Push

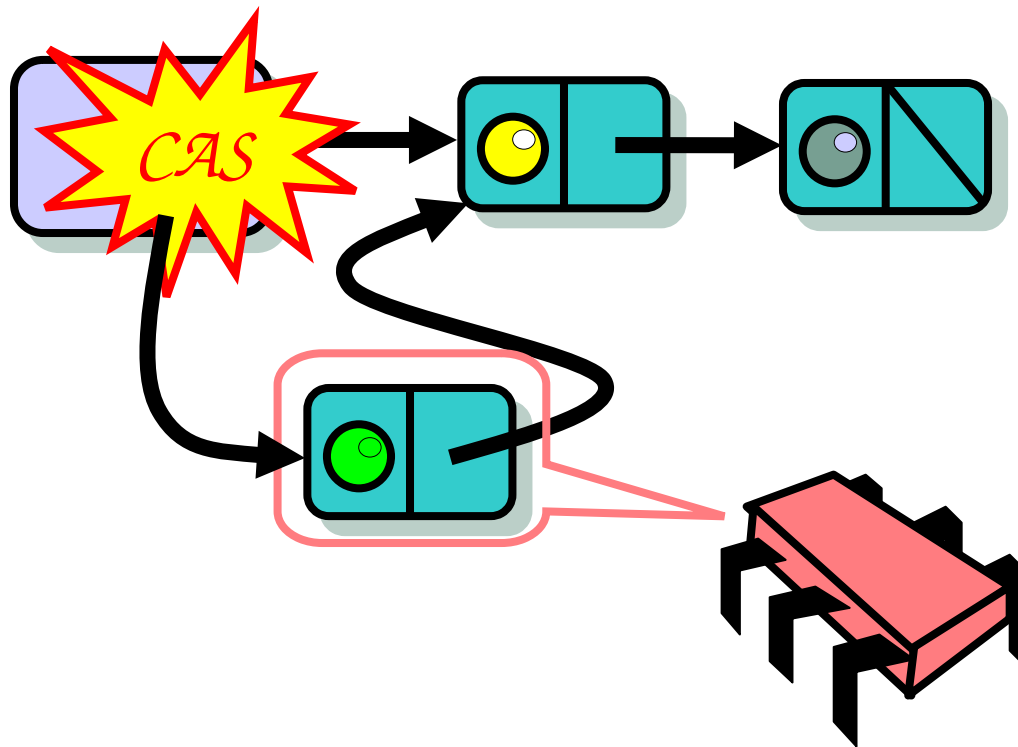




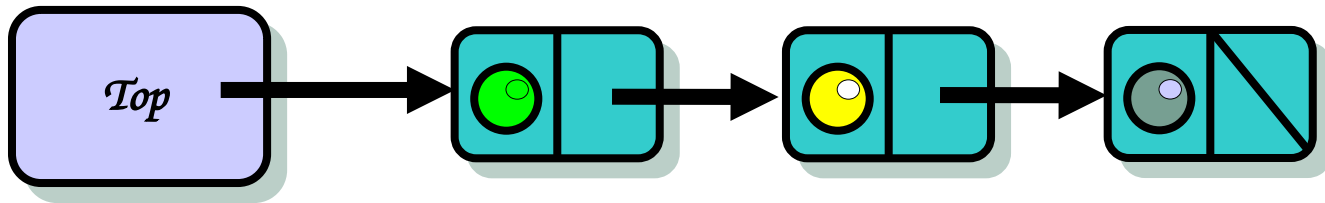
# Push



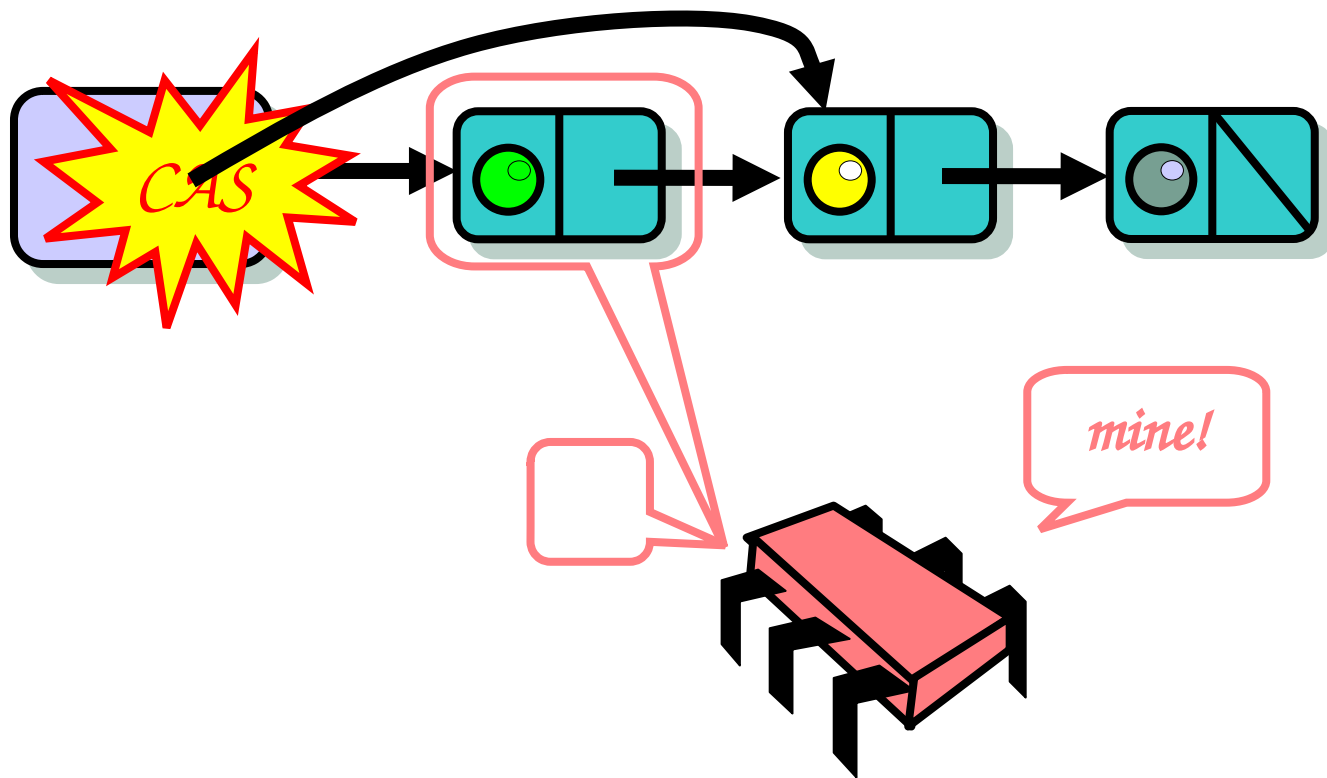
# Push



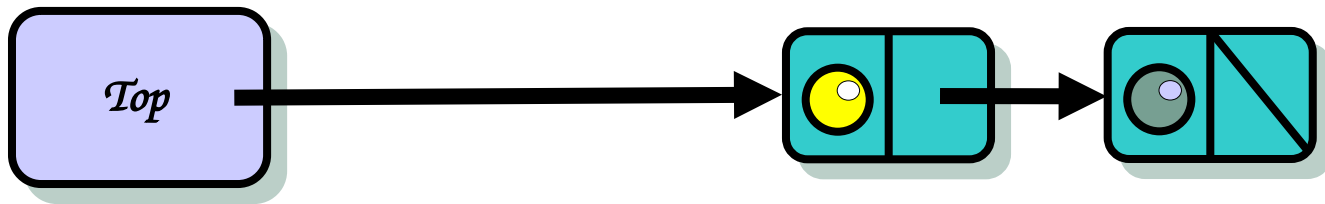
# Stack



# Pop



# Pop



# Lock-free Stack

```
public boolean tryPush(Node node){  
    Node oldTop = top.get();  
    node.next = oldTop;  
    return(top.compareAndSet(oldTop, node))  
}
```

In what scenario will the tryPush  
return false?

*maybe when you push to the top but pop from the top concurrently*



# Lock-free Stack

- If the `compareAndSet` returns a false it means that there is high contention
- In scenarios where there are high contention it is more effective to backoff and try again later

# Lock-free Stack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else backoff.backoff();  
    }  
}
```



# Lock-free Stack

```
public T pop() throws EmptyException {
    while (true) {
        Node returnNode = tryPop();
        if (returnNode != null)
            return returnNode.value;
        else
            backoff.backoff();
    }
}

protected Node tryPop() throws EmptyException {
    Node oldTop = top.get();
    if (oldTop == null)
        throw new EmptyException()
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop, newTop))
        return oldTop;
    else
        return null;
}
```



# Lock-free Stack

- Good

- No locking

- Bad

- Even with backoff, huge contention at top
  - In any case, no real parallelism

# Lock-free Stack

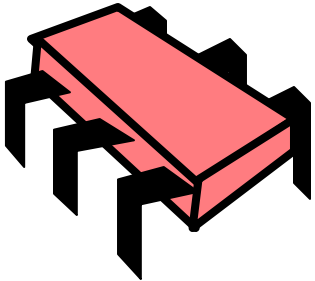
- The LockFreeStack implementation scales poorly
  - Stack's top field is a source of contention
  - Sequential bottleneck
    - Method calls can proceed only one after the other
- Exponential Backoff alleviates contention, but does nothing for bottleneck



# Observation

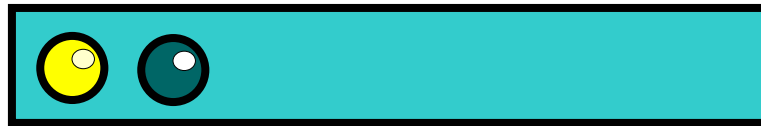
- If a `push()` is immediately followed by a `pop()` the two operations cancel out and the stack does not change
- We can exploit this observation

# Observation

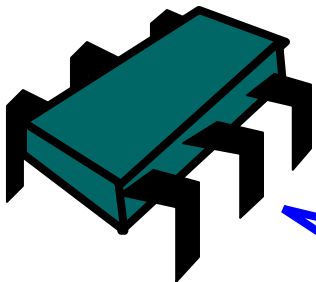


*Push()*

*linearizable stack*



*Pop()*



*Yes!*

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



# Elimination

- Cancel out concurrent pairs of pop() and push() method calls
- Thread calling push() exchanges value with thread calling pop()
- No modification to the stack needed
- The two calls *eliminate* one another

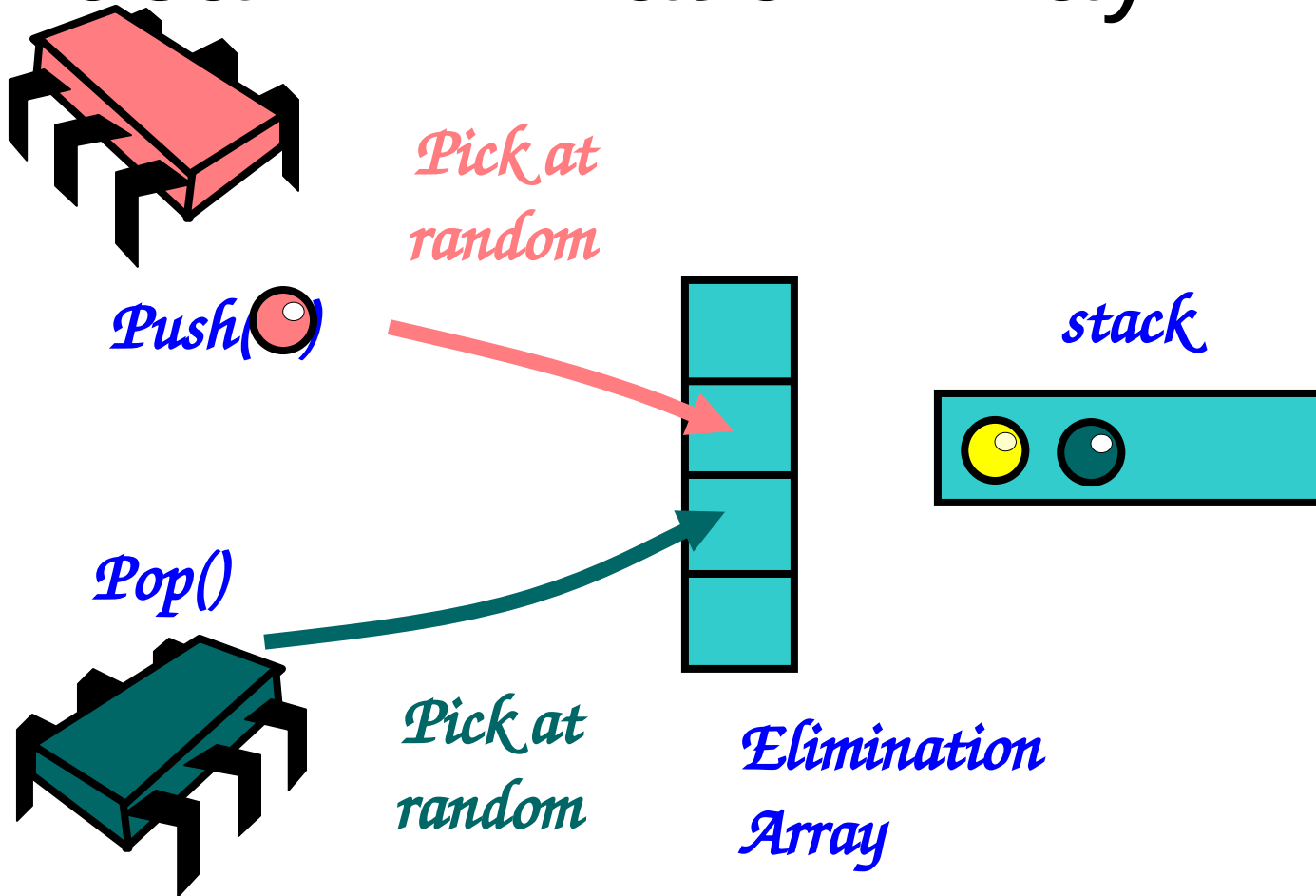


# Elimination

## ■ Idea:

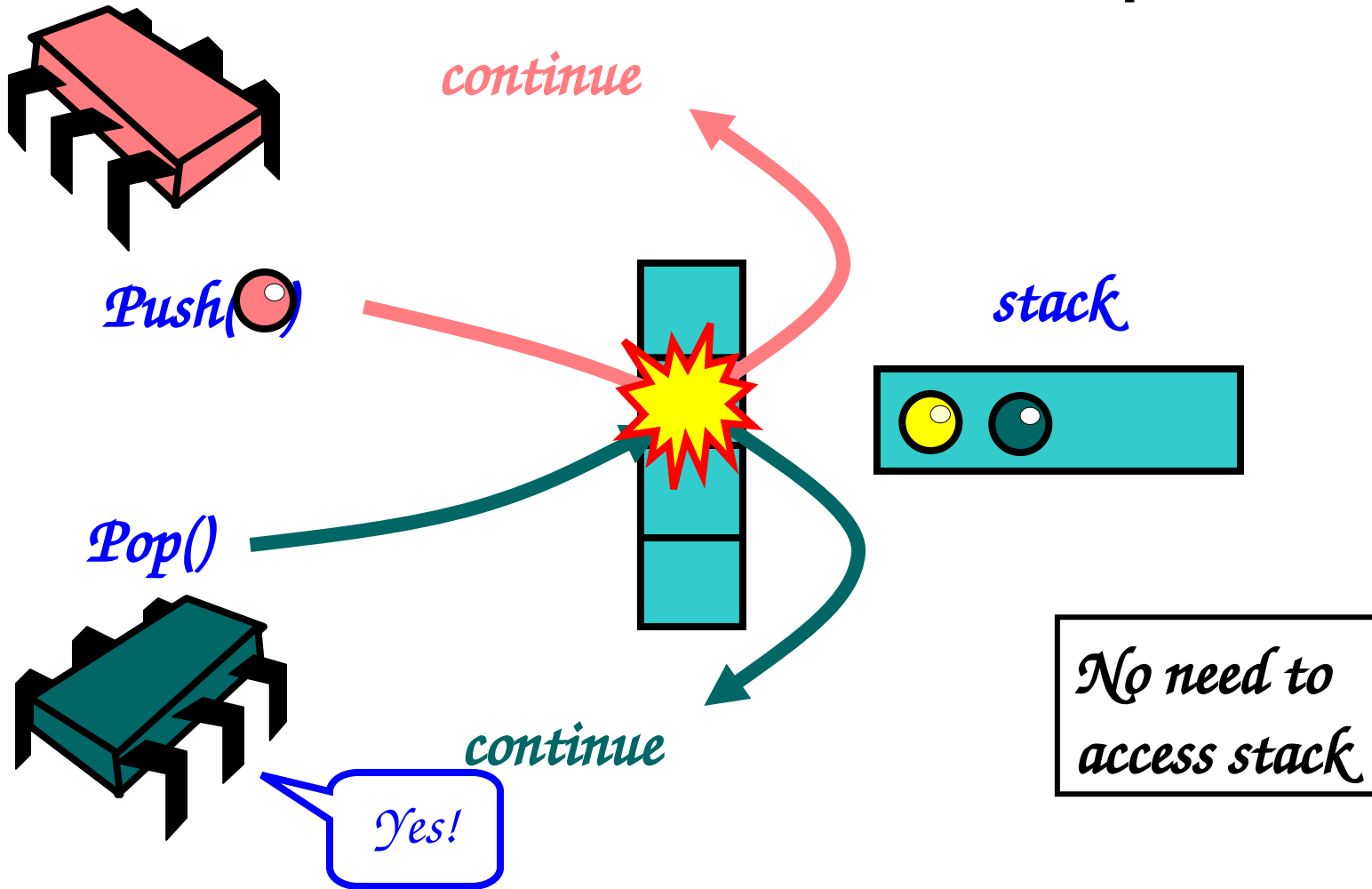
- Threads eliminate one another through an EliminationArray
- Threads pick random array entries to try to meet complementary calls
- Pairs of pop() and push() method calls that pick the same location exchange values and returns

# Idea: Elimination Array





# Push Collides With Pop

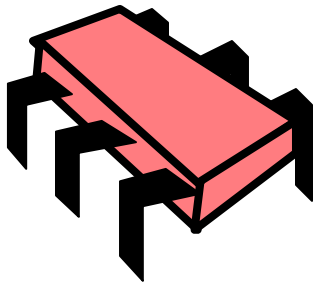




# Elimination

- A thread whose call cannot be eliminated
  - Because it failed to find a partner (picked a wrong location) or
  - Picked a partner of the wrong type (pop() meeting a pop())
- Can either try again to eliminate at a new location or access the stack

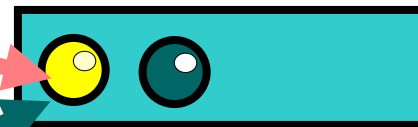
# No Collision



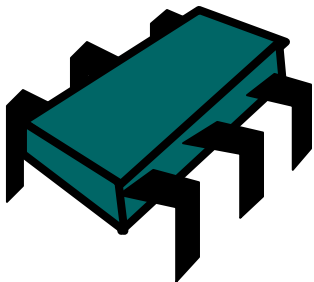
*Push()*



*stack*



*Pop()*



*If pushes collide or pops  
collide access stack*

*access stack*



# Elimination

- The EliminationArray can be used as a backoff scheme:
  - Each thread first access the Stack
  - If it fails (compareAndSet fails) it attempts to eliminate its call using the array instead of simply backing of
  - If it fails to eliminate, it access the Stack again
  - And so on...
- Called an EliminationBackoffStack



# Elimination Backoff Stack

- Threads that push and pop should be allowed to cancel out
- However we must avoid a situation in which a thread can make an offer to more than one thread



# Exchangers

- Object that allow exactly two threads (and no more) to rendezvous and exchange values
- The EliminationArray can be implemented as an array of Exchangers



# LockFreeExchanger

- A `LockFreeExchanger<T>` object allows two threads to exchange values of type `T`
- If A calls `exchange()` with `a` and B calls `exchange()` with `b`, then A should return `b` and vice versa



# LockFreeExchanger

- On a high level:
  - First thread arrives to write its values and spins until a second thread arrives
  - The second detects that the first is waiting, reads its value and signals the exchange
- The first thread may timeout if a second does not show up





# LockFreeExchanger

- The exchanger has three possible states:
  - EMPTY
  - BUSY
  - WAITING
- AtomicStampedReference records the exchanger's state as an int stamp



# LockFreeExchanger

- The exchanger reads the state of the AtomicStampedReference slot and proceeds as follows:



# If EMPTY

- Thread tries to place item in slot and set state to WAITING
- If it was successful it spins waiting for another thread
- If another thread shows up, it will take the item, replace it with its own and set the state to BUSY
- The waiting state will take the item and reset the state to EMPTY

# LockFreeExchanger

```
public class LockFreeExchanger {  
    public T exchange(T myItem) {  
        int[] stampHolder = {EMPTY};  
        while (true) {  
            T yrItem = slot.get(stampHolder);  
            int stamp = stampHolder[0];  
            switch(stamp) {
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        } else {
            yrItem = slot.get(stampHolder);
            slot.set(null, EMPTY);
            return yrItem;
        }
    }
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        }
        return yrItem;
    }
} break;
```

*Slot is free, try to insert myItem and change state to WAITING*

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        }
        Loop while still time left to attempt exchange
        yrItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        }
    }
} break;
```

*Get item from slot and check if state changed to BUSY*



# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        }
        If successful reset slot state to EMPTY
        yrItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, EMPTY);
                return yrItem;
            }
        }
        if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
        } else {
            yrItem = slot.get(stampHolder);
            slot.set(null, EMPTY);
            return yrItem;
        }
    }
} break;
```

*and return item found in slot*

# If EMPTY

*Otherwise we ran out of time, try to  
reset state to EMPTY, if successful  
time out*

```
        em, myItem, EMPTY,
        timeBound){
    yrItem = slot.get(stampHolder);
    if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return yrItem;
    }
    if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
    } else {
        yrItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

# If EMPTY

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

EMPTY,

```
while (System.nanoTime() < timeBound){
    yrItem = slot.get(stampHolder);
    if (stampHolder[0] == BUSY) {
        slot.set(null, EMPTY);
        return yrItem;
    }
    if (slot.compareAndSet(myItem, null, WAITING,
EMPTY)){throw new TimeoutException();
    } else {
        yrItem = slot.get(stampHolder);
        slot.set(null, EMPTY);
        return yrItem;
    }
} break;
```

# If EMPTY

*Set slot to EMPTY and return item found*

```
WAITING)) {  
    while (System.nanoTime() < timeBound){  
        yrItem = slot.get(stampHolder);  
        if (stampHolder[0] == BUSY) {  
            slot.set(null, EMPTY);  
            return yrItem;  
        }  
        if (slot.compareAndSet(myItem, null, WAITING,  
EMPTY)){throw new TimeoutException();  
        } else {  
            yrItem = slot.get(stampHolder);  
            slot.set(null, EMPTY);  
            return yrItem;  
        }  
    }  
} break;
```

# If EMPTY

```
case EMPTY: // slot is free
    if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) {
        while (System.nanoTime() < timeBound){
            yrItem = slot.get(stampHolder);
            if (stampHolder[0] == BUSY) {
                slot.set(null, yrItem);
                return yrItem;
            }
        }
        if (slot.compareAndSet(yrItem, myItem, EMPTY,
WAITING)) { throw new TimeoutException();
        } else {
            yrItem = slot.get(stampHolder);
            slot.set(null, EMPTY);
            return yrItem;
        }
    }
} break;
```

*If initial CAS failed then someone else changed state from EMPTY to WAITING so retry from start*



# If WAITING

- Some thread is waiting and the slot contains the item
- The thread takes the item and tries to replace it with its own by changing state from WAITING to BUSY
- It may fail if another thread succeeds or if the initial thread timeout
- If it does succeed it can return the item

# If WAITING

```
case WAITING:  
    if (slot.compareAndSet(yrItem, myItem, WAITING,  
        BUSY))  
        return yrItem;  
break;
```

- Someone is waiting for an exchange, to try to CAS my item in and change state to BUSY
- If successful return that item, otherwise another thread got it





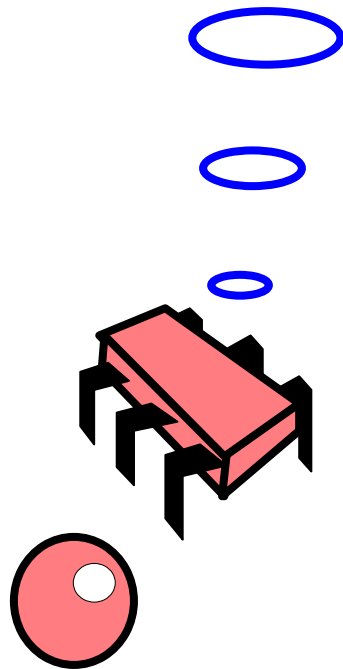
# If BUSY

- If the state is BUSY then two other threads are currently using the slot for an exchange

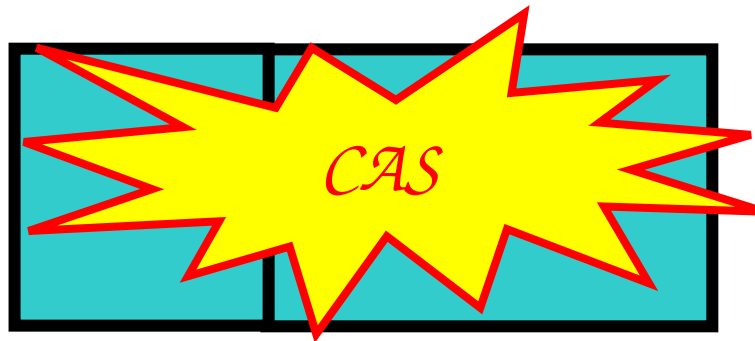
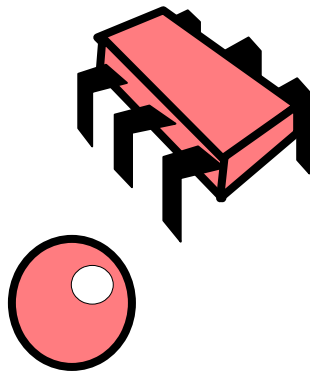
```
case BUSY:  
    break;
```

# Lock-free Exchanger

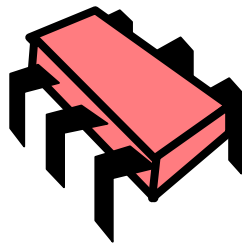
*Sees that slot is empty*



# Lock-free Exchanger

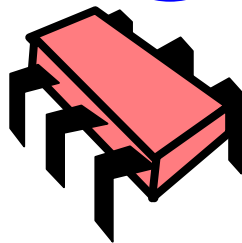


# Lock-free Exchanger



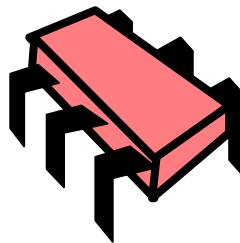
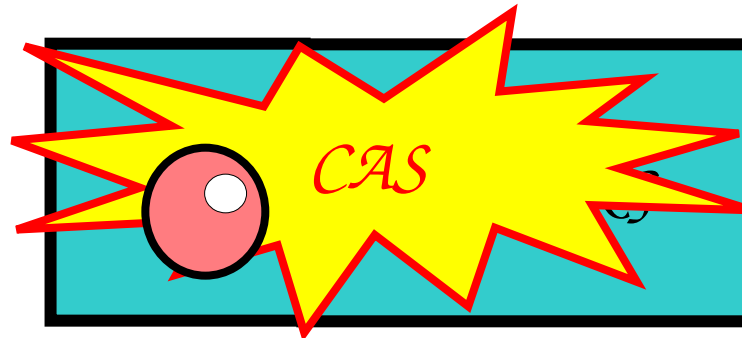
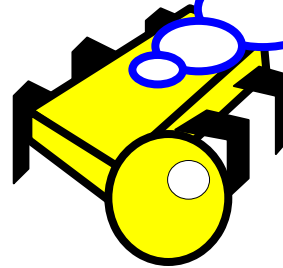
# Lock-free Exchanger

*Waiting for partner ...*



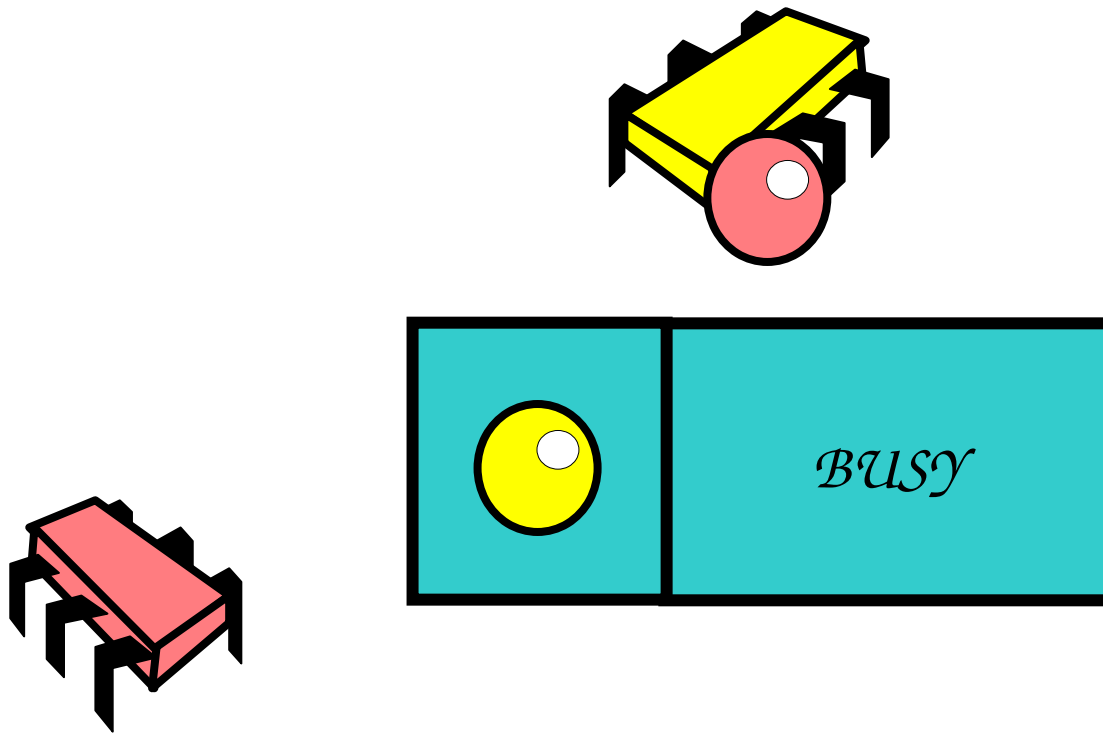
# Lock-free Exchange

*Try to exchange item  
and set state to  
BUSY*



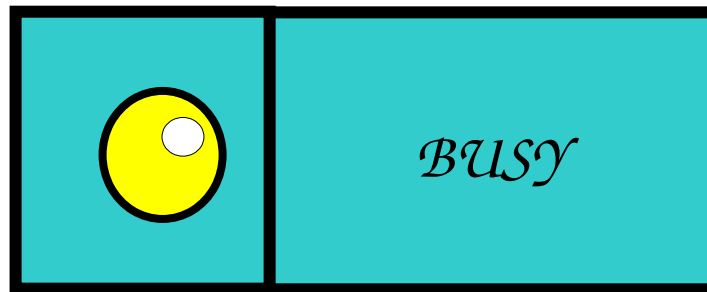
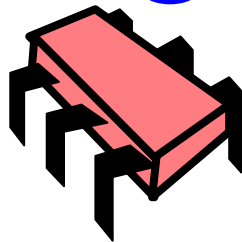
*CAS*

# Lock-free Exchanger



# Lock-free Exchanger

*Partner showed up,  
take item and reset to  
EMPTY*







# Back to EliminationBackoffStack

- An EliminationArray is implemented as an array of Exchanger objects
- A thread attempting to perform an exchange picks an array entry at random and call that entry's exchange() method



# EliminationBackoffStack

- Each thread first access the Stack
- If it fails, it attempts to eliminate its call using the array
  - It chooses a random array entry and calls exchange
- If it fails to eliminate due to:
  - Choosing an entry where the state is BUSY
  - Colliding with an unsuitable method call
- It accesses the Stack again
- And so on...

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*First try to push element onto stack*

# Same method as **Lock-free Stack**

```
public boolean tryPush(Node node){  
    Node oldTop = top.get();  
    node.next = oldTop;  
    return(top.compareAndSet(oldTop, node))  
}
```

*Will return false if another thread succeeded = high contention*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If high contention visit  
Elimination Array and  
try to eliminate call*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If push method returns null, exchange was succesful*

# EliminationBackoffStack

```
public void push(T value) {  
    Node node = new Node(value);  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
                eliminationArray.visit(value);  
            if (otherValue == null)  
                return;  
        } catch (TimeoutException e) {}  
    }  
}
```

*If not, try again*



# EliminationBackoffStack

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

# EliminationBackoffStack

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

*First try to pop element from stack*

# Same method as **Lock-free Stack**

```
protected Node tryPop() throws EmptyException
{
    Node oldTop = top.get();
    if (oldTop == null)
        throw new EmptyException();
    Node newTop = oldTop.next;
    if (top.compareAndSet(oldTop, newTop))
        return oldTop;
    else
        return null;
}
```

*Return null if failed due to high contention*

# EliminationBackoffStack

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

*Attempt to eliminate – add null to elimination  
array*

# EliminationBackoffStack

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

*Successful exchange for pop method if non-null  
value returned*



# Performance

- At low levels of contention the performance of EliminationBackoffStack is comparable to LockFreeStack
- However at high contention the number of successful eliminations will increase, allowing more operations to complete in parallel