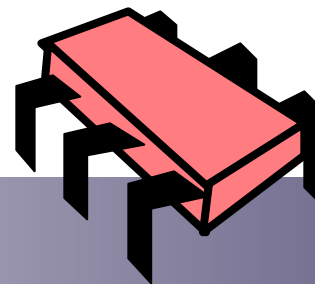

Concurrent Stacks & Elimination

- מרצה: יהודה אפק
- מגיש: ערן שרגיאן



Outline

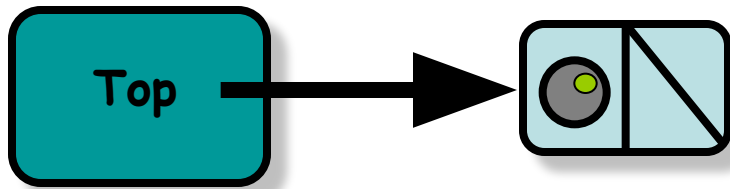
- **Quick reminder of the Stack structure.**
- **The Unbounded Lock-Free Stack.**
- **The Elimination Backoff Stack.**



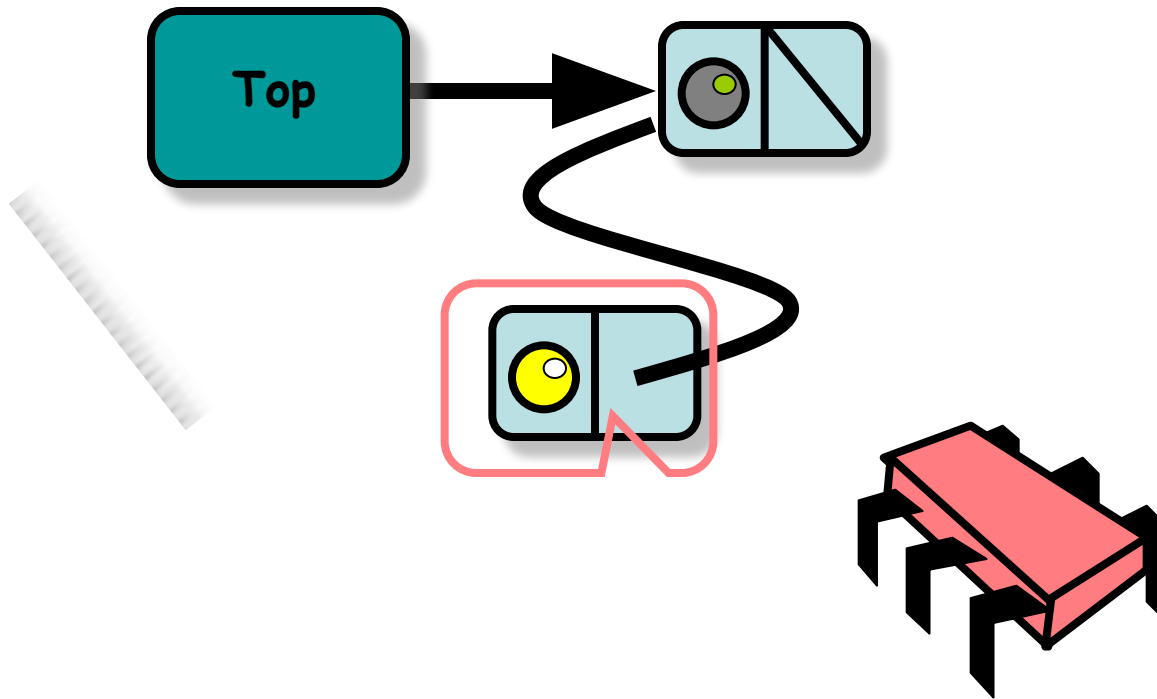
Concurrent Stack

- The `Stack<T>` class is a collection of items (of type `T`) that provides the following methods:
 - `push(x)`
 - `pop()`
- Satisfying the Last-In-First-Out (LIFO) property:
 - The last item pushed is the first popped.

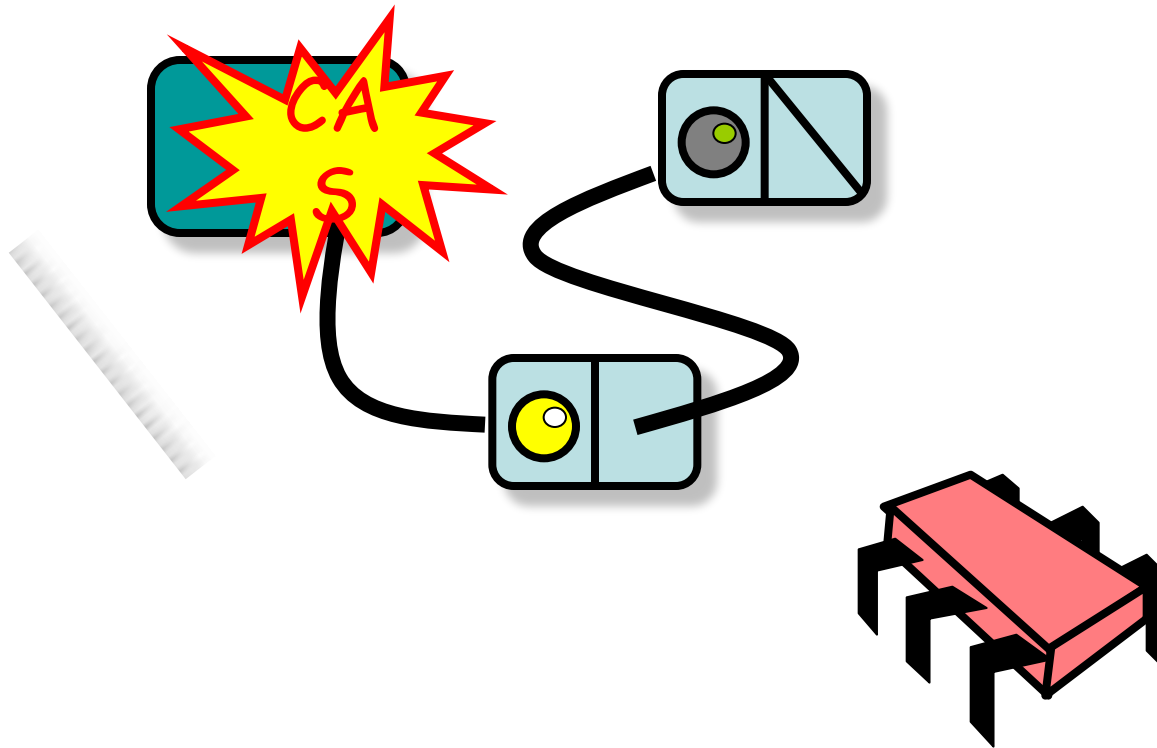
Empty Stack



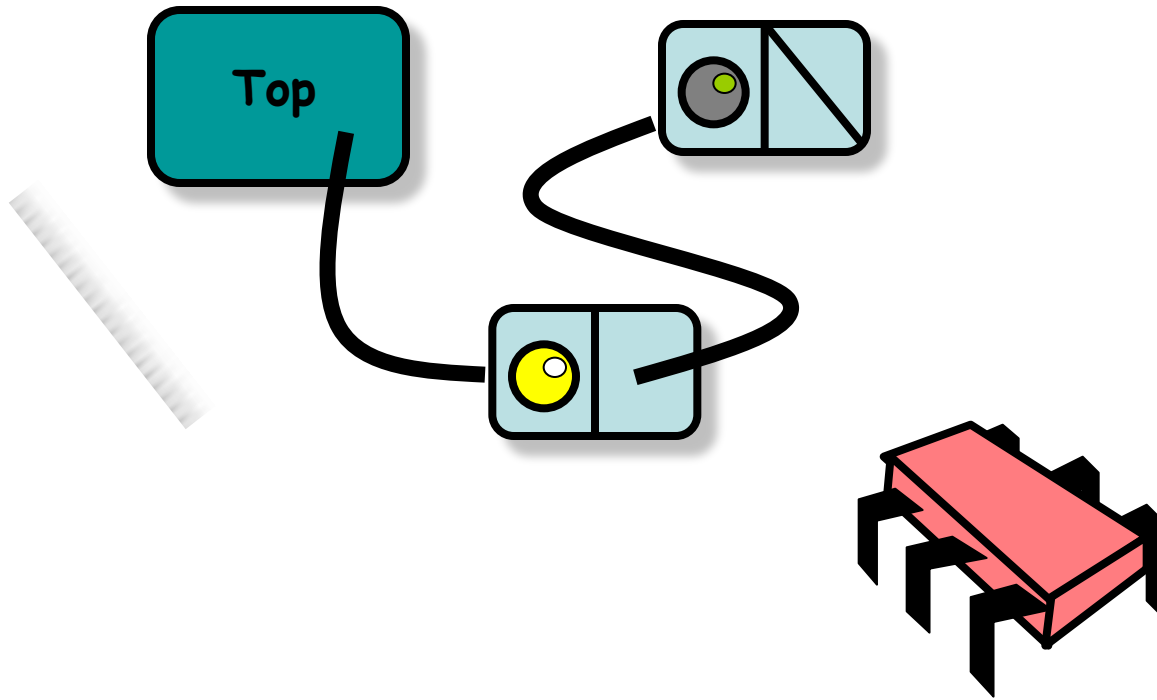
Push



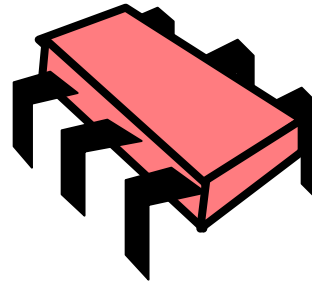
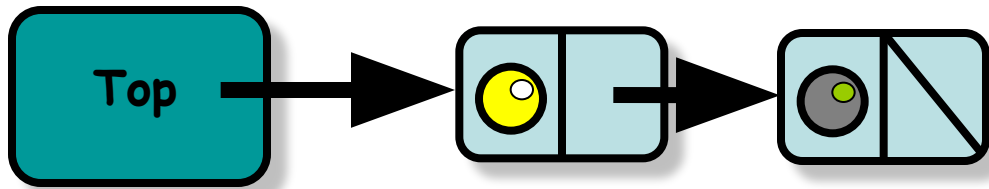
Push



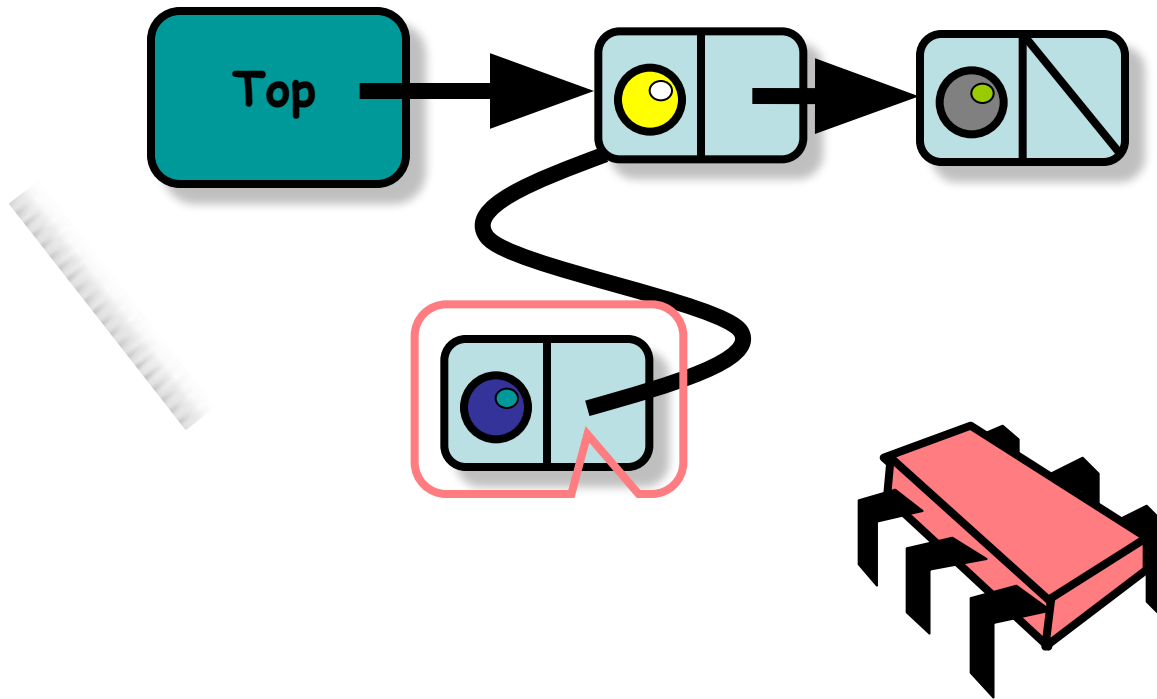
Push



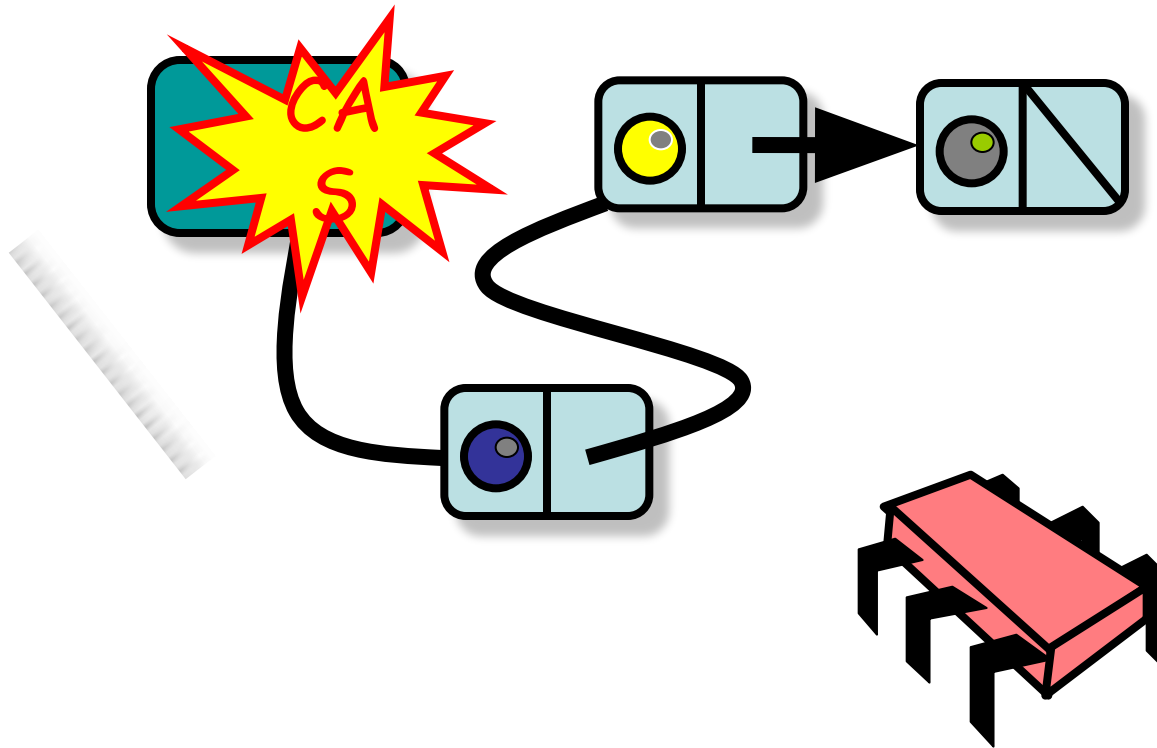
Push



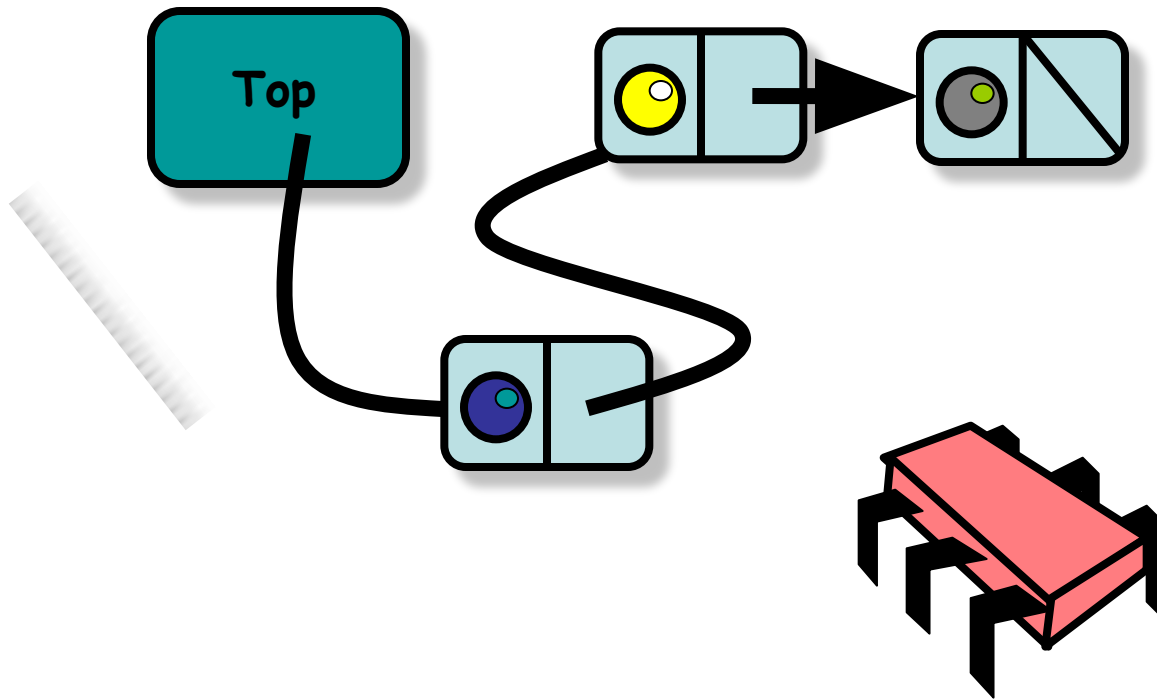
Push



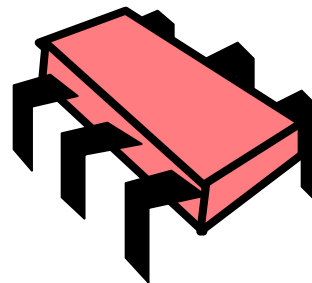
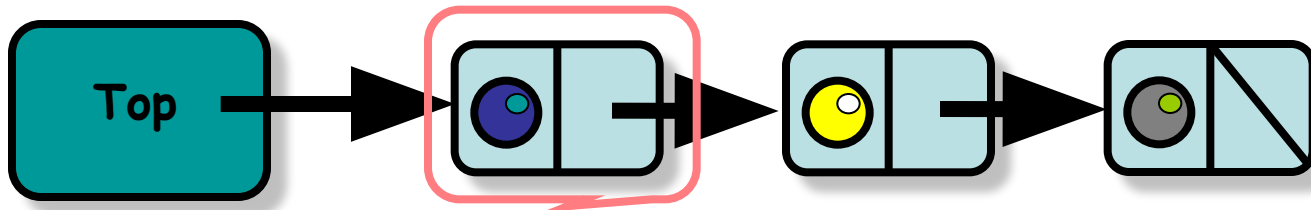
Push



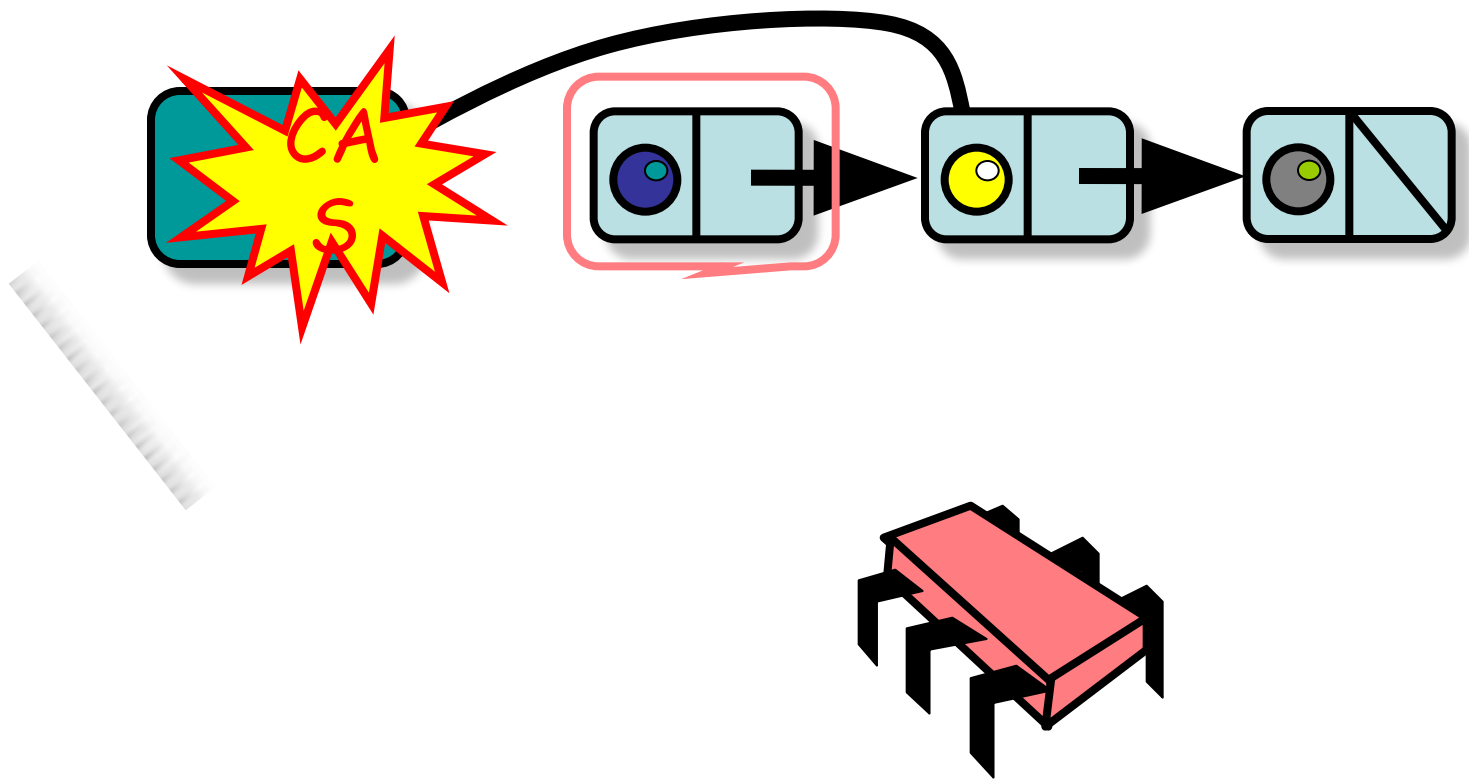
Push



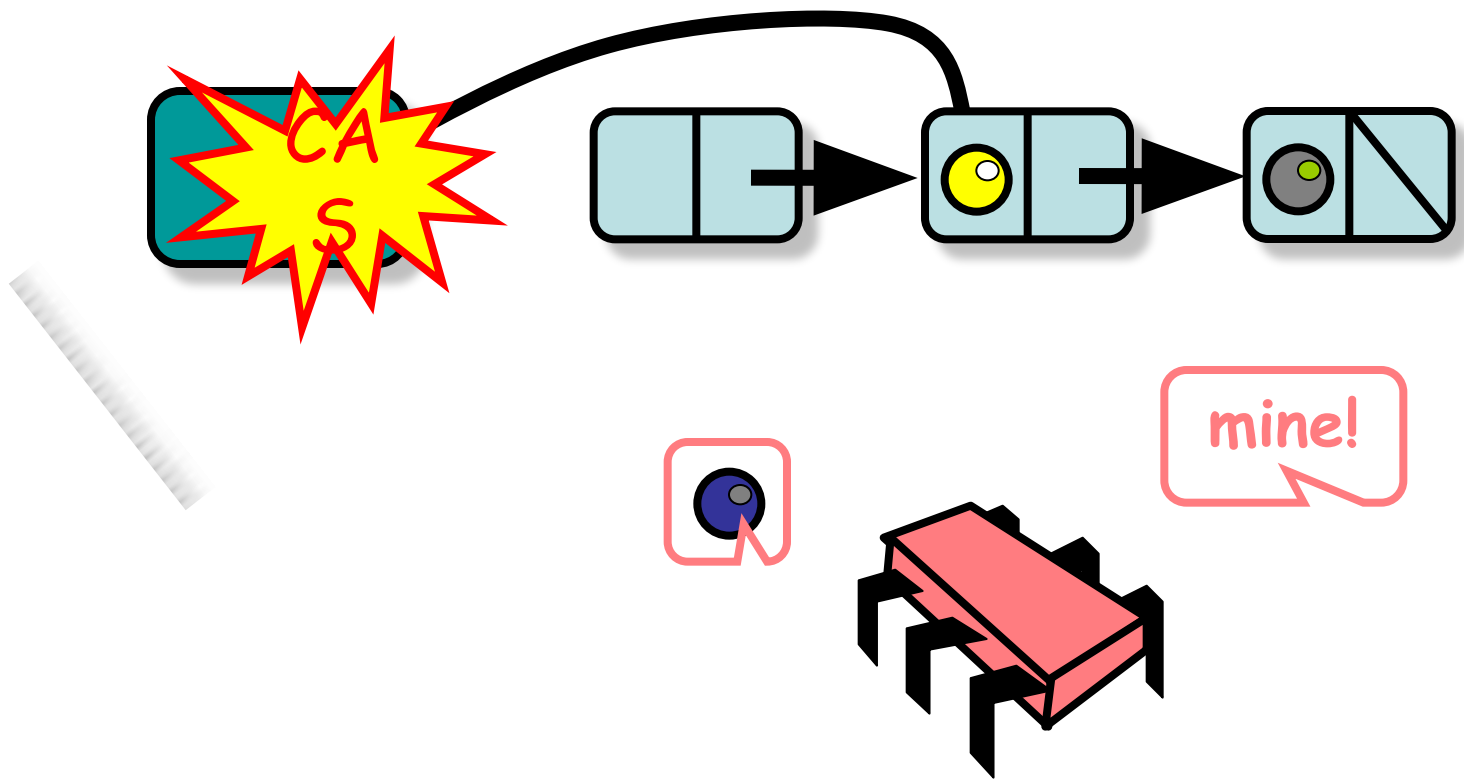
Pop



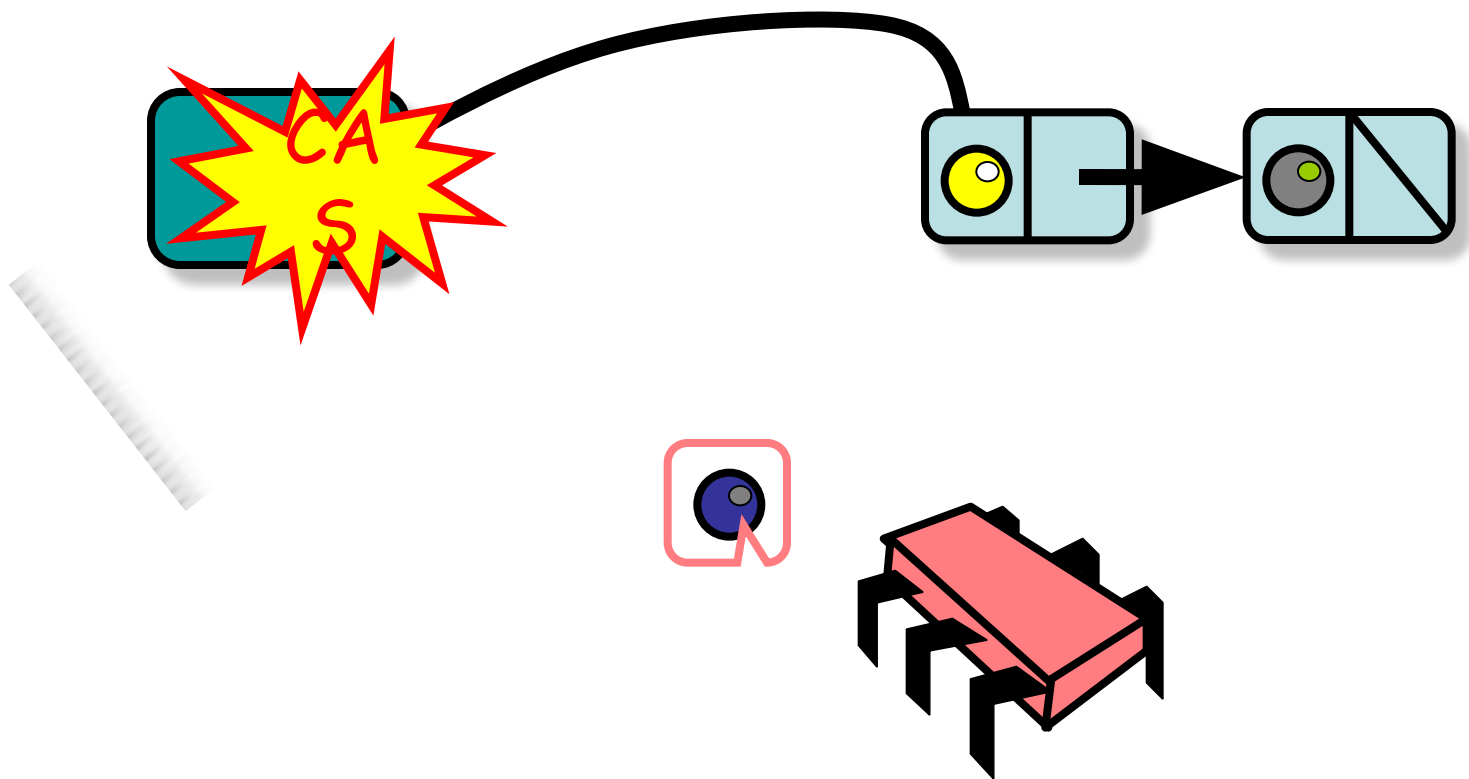
Pop



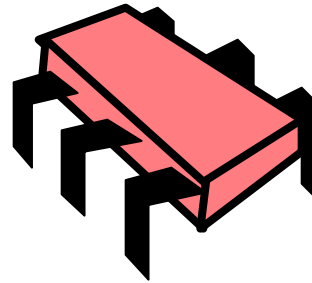
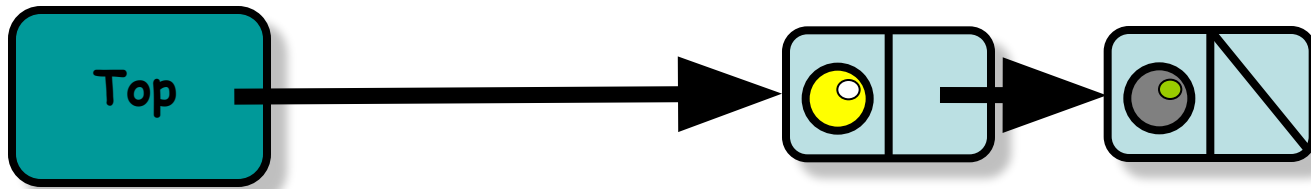
Pop



Pop



Pop



The LockfreeStack class

- The lock-free stack is a linked list, where the top field points to the first node (or null if the stack is empty).
- A pop() call uses compareAndSet() to try to remove the first node from the stack.
- A push() call uses compareAndSet() to try to insert a new node into the top of the stack.

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

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

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

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

Lock-free Stack

- **Good**
 - No locking
- **Bad**
 - huge contention at top
 - No parallelism



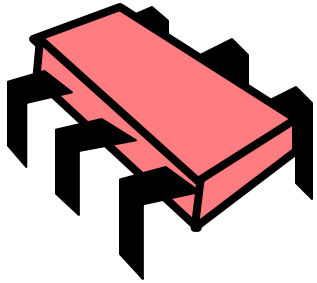
Elimination-Backoff Stack

The LockFreeStack implementation scales poorly, not so much because the stack's top field is a source of contention, but primarily because it is a sequential bottleneck.

Ways to solve it :

- *exponential backoff* (reduces contention but does not solve the bottleneck problem).
- *elimination backoff*

Observation

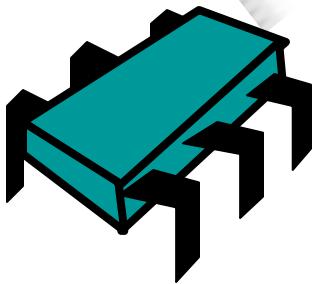


Push(●)

linearizable stack



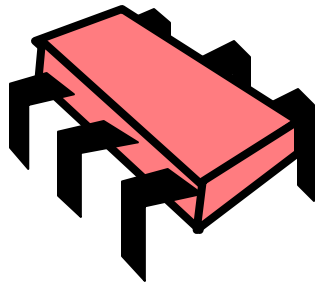
Pop()



Yes!

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

Idea: Elimination Array

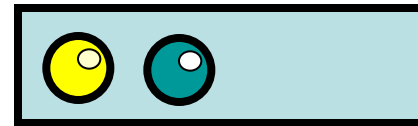


Pick at random

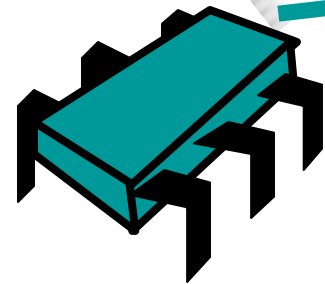
Push(●)



stack



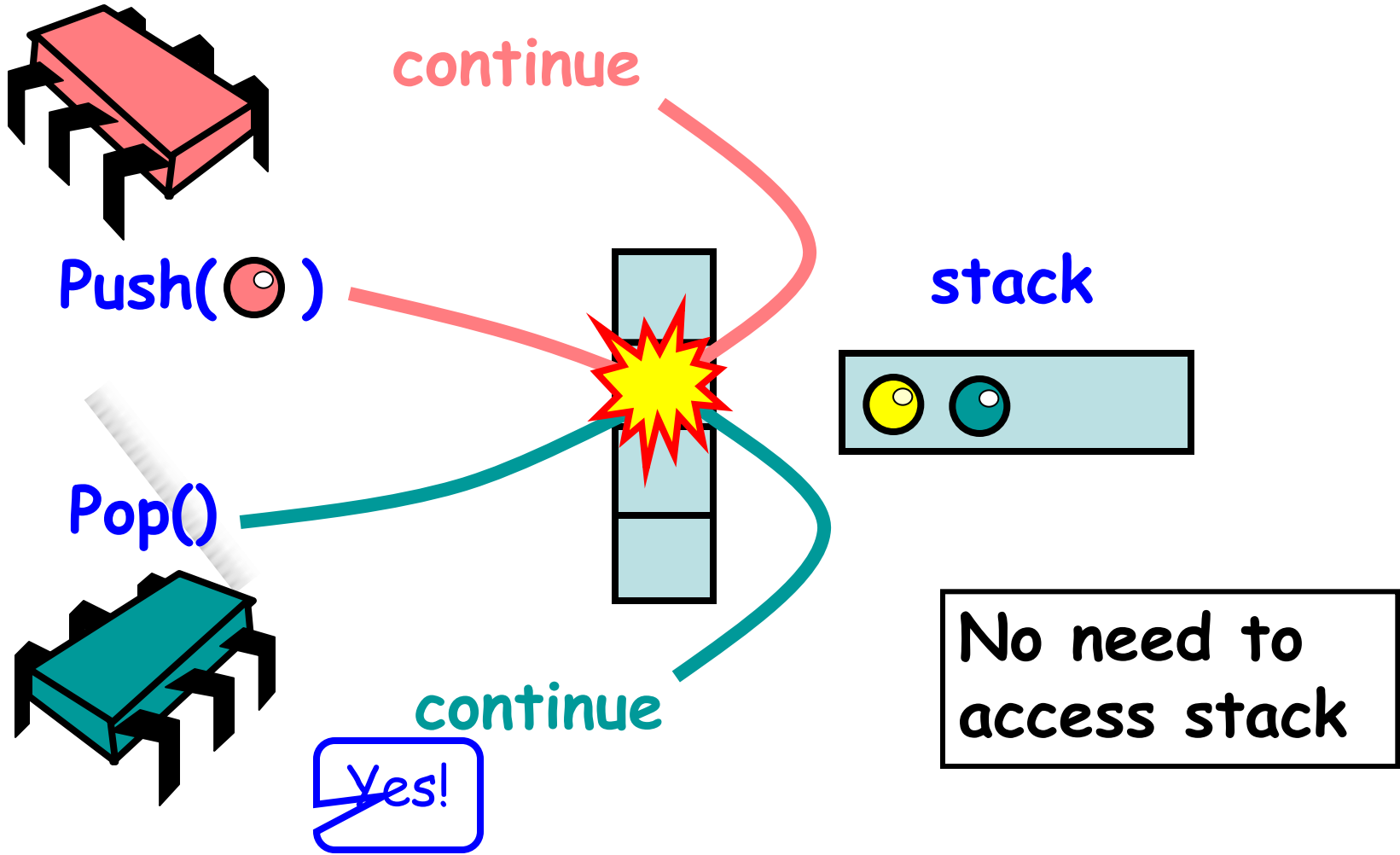
Pop()



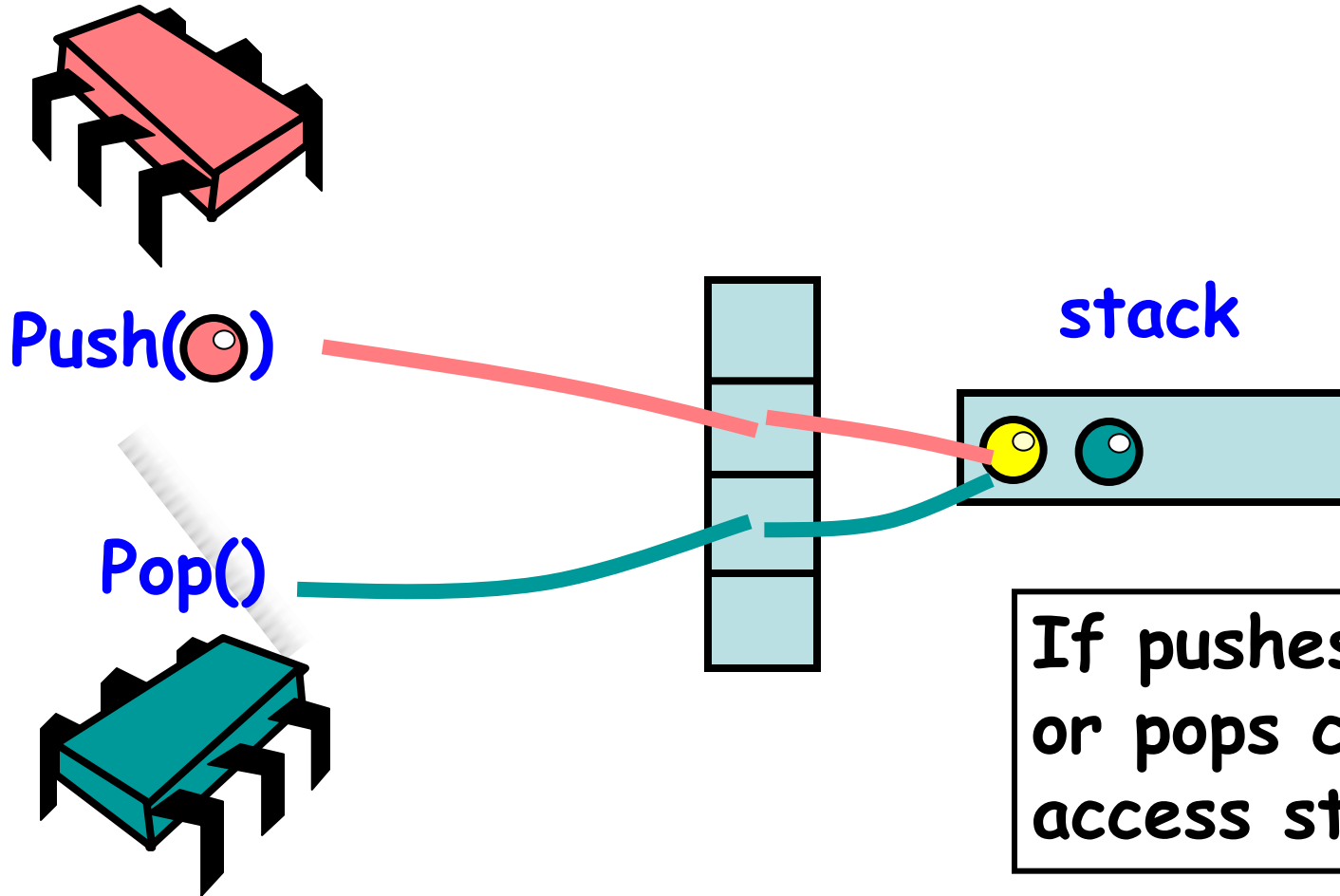
Pick at random

Elimination Array

Push Collides With Pop



No Collision

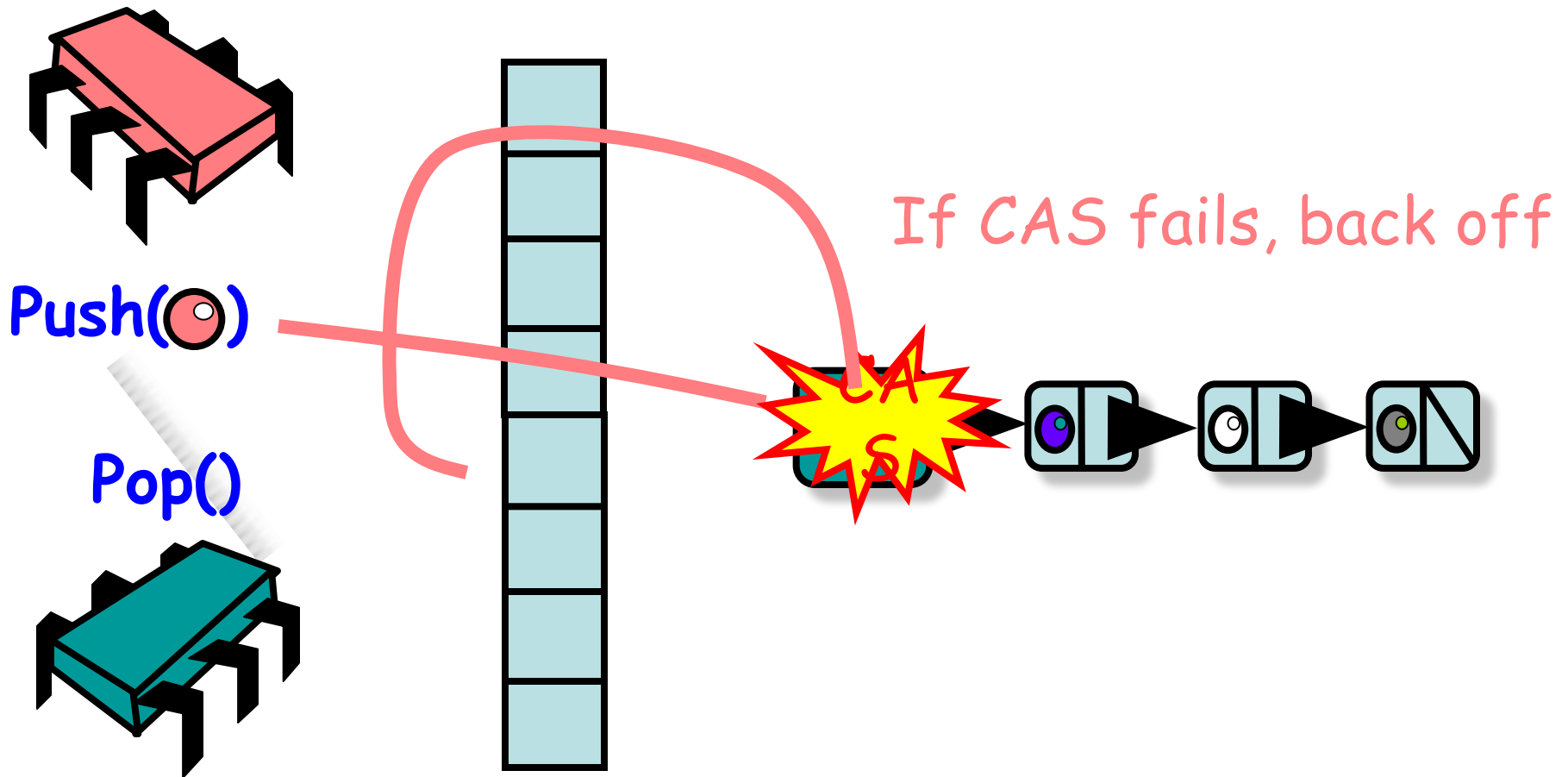


Elimination-Backoff Stack

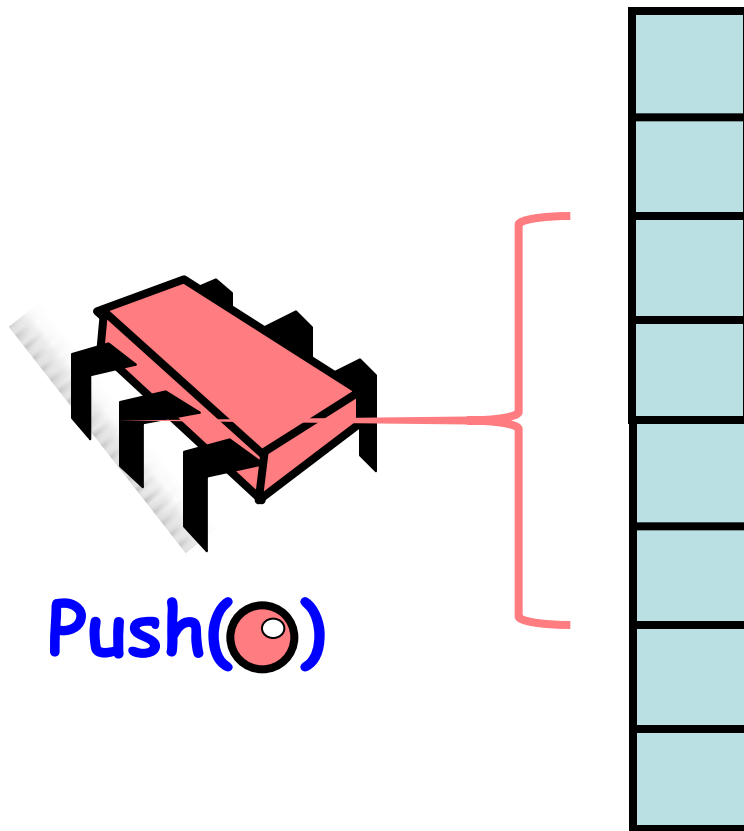
- A union of the LockFreeStack class with the elimination array
- Access Lock-free stack,
 - If uncontended, apply operation
 - if contended, back off to elimination array and attempt elimination



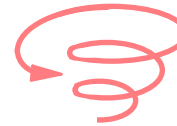
Elimination-Backoff Stack



Dynamic Range and Delay



Push()



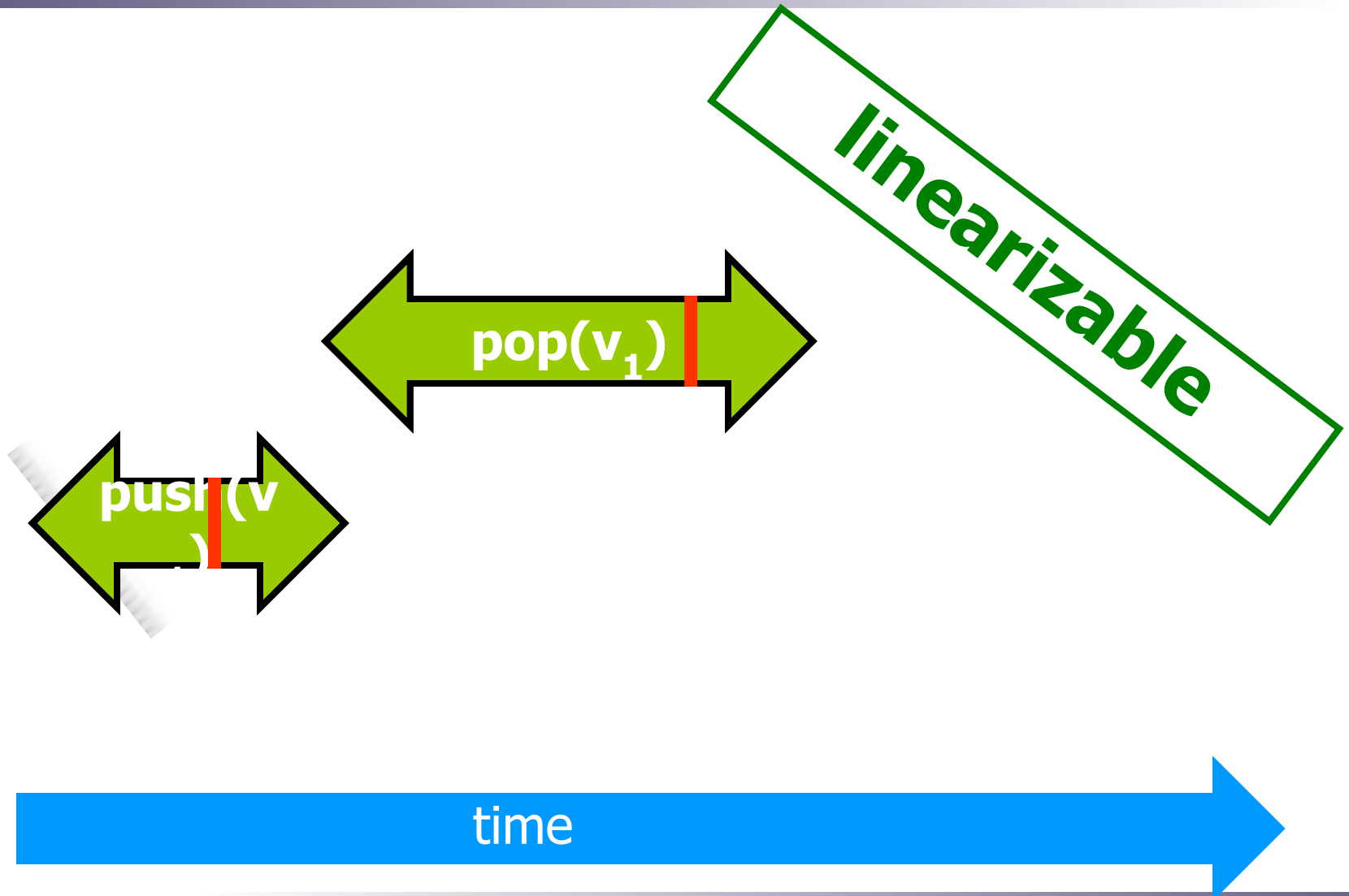
Pick random range and
max time to wait for
collision based on level of
contention encountered

Linearizability

The combined data structure, array, and shared stack, is linearizable because the shared stack is linearizable, and the eliminated calls can be ordered as if they happened at the point in which they exchanged values.

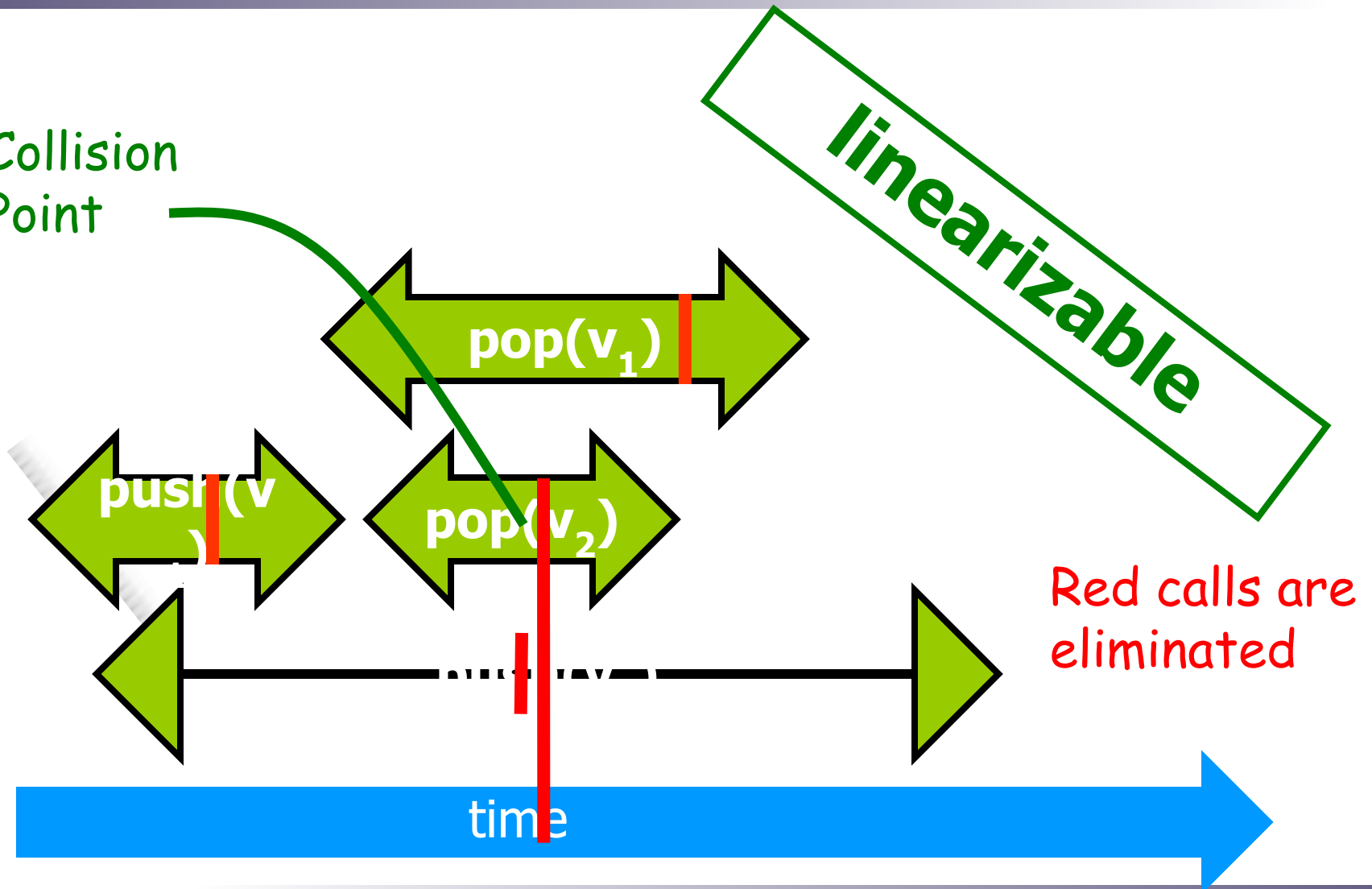
- **Un-eliminated calls**
 - linearized as before
- **Eliminated calls:**
 - linearize pop() immediately after matching push()
- **Combination is a linearizable stack**

Un-Eliminated Linearizability




Eliminated Linearizability

Collision
Point



Backoff Has Dual Effect

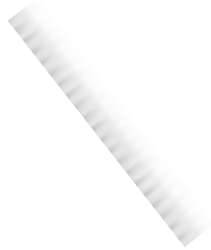
- Elimination introduces parallelism
 - Backoff onto array cuts contention on lock-free stack
 - Elimination in array cuts down total number of threads ever accessing lock-free stack
- 

Elimination Array

```
public class EliminationArray {  
    private static final int duration = ...;  
    private static final int timeUnit = ...;  
    Exchanger<T>[] exchanger;  
    public EliminationArray(int capacity) {  
        exchanger = new Exchanger[capacity];  
        for (int i = 0; i < capacity; i++)  
            exchanger[i] = new Exchanger<T>();  
        ...  
    }  
    ...  
}
```

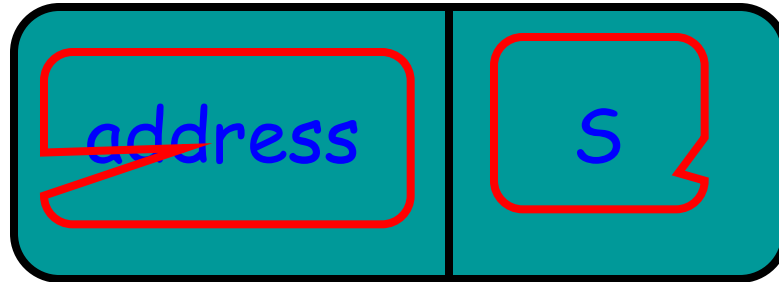
A Lock-Free Exchanger

```
public class Exchanger<T> {  
    AtomicStampedReference<T> slot  
    = new AtomicStampedReference<T>(null, 0);  
}
```



Atomic Stamped Reference

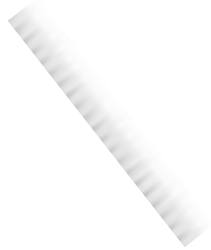
Reference



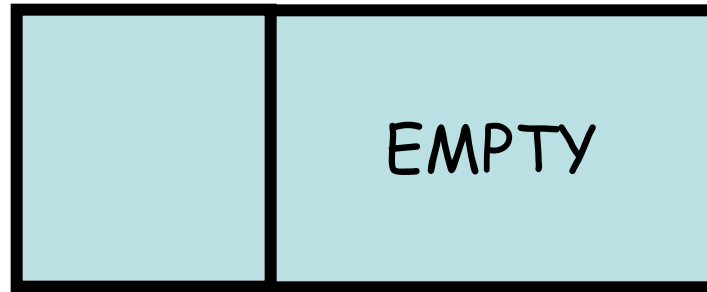
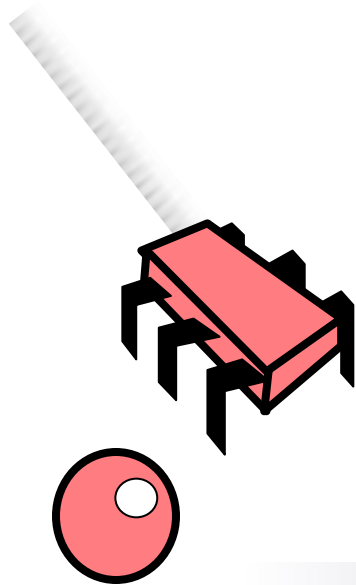
Stamp

Exchanger Status

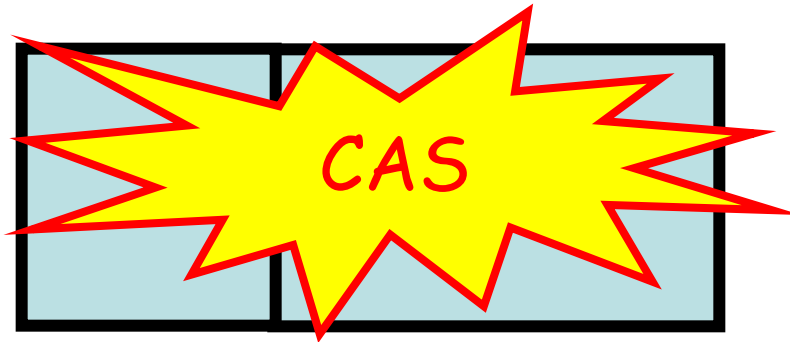
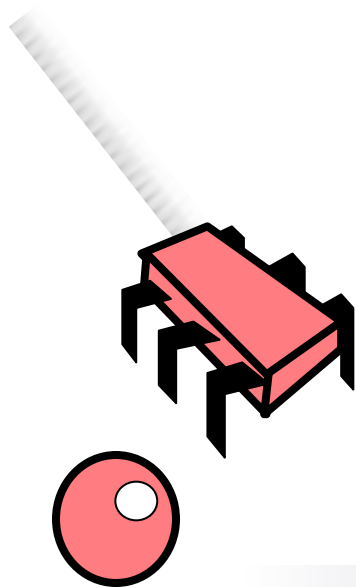
```
enum Status {EMPTY, WAITING, BUSY};
```



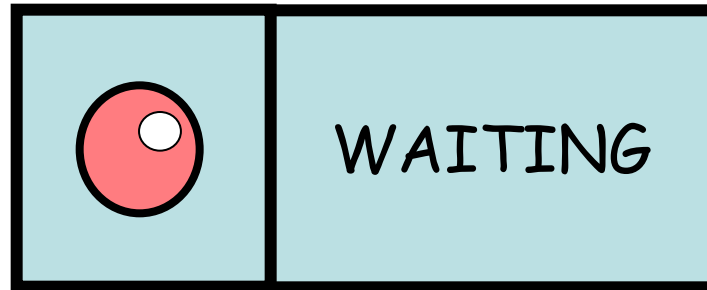
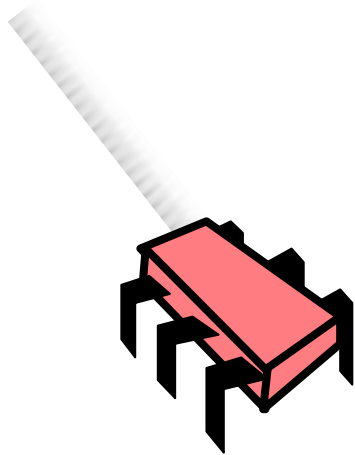
Lock-free Exchanger



Lock-free Exchanger

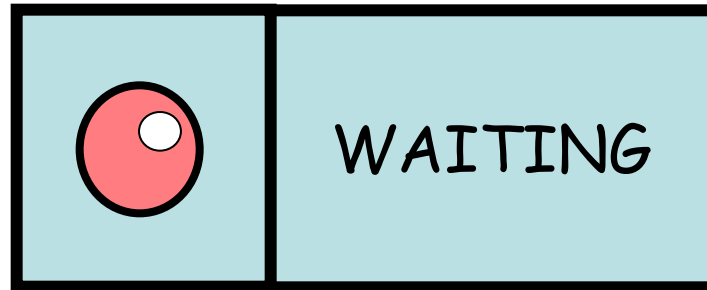
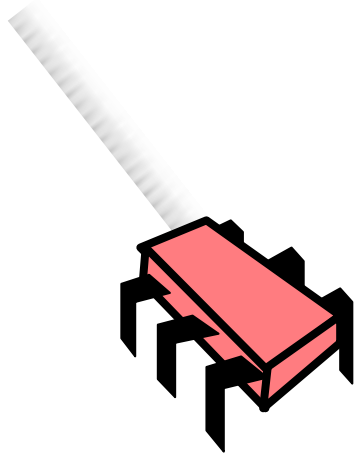


Lock-free Exchanger



Lock-free Exchanger

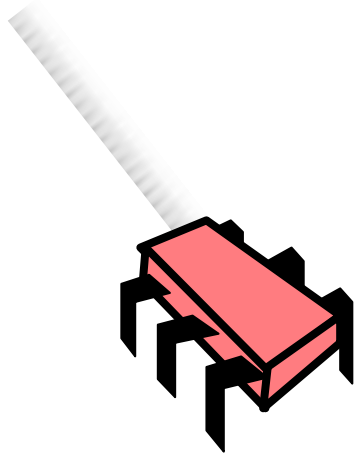
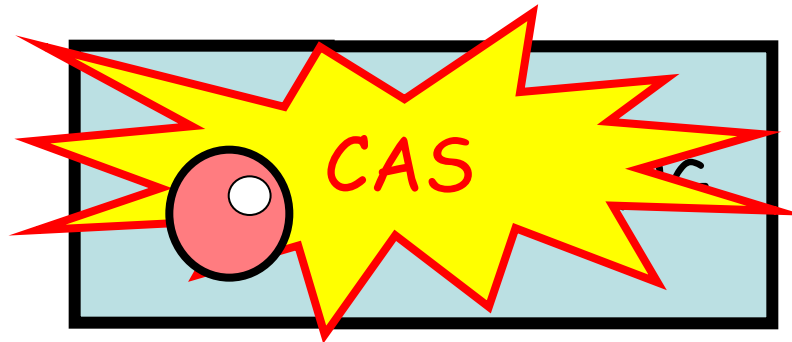
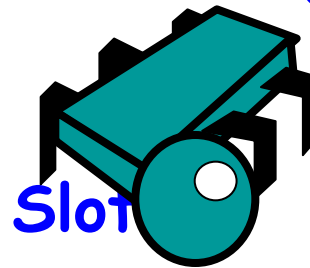
In search of
partner ...



Lock-free Exchanger

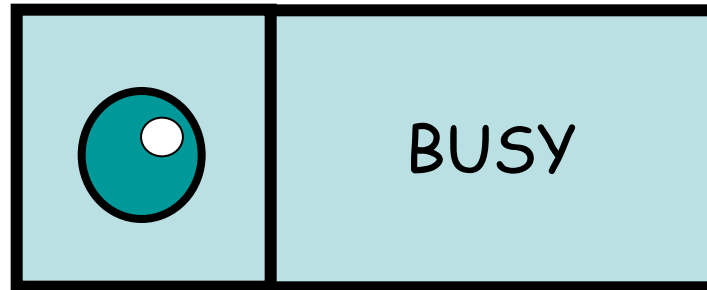
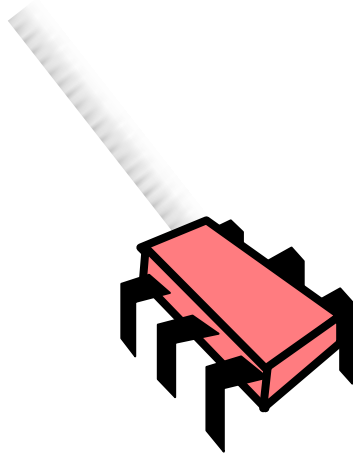
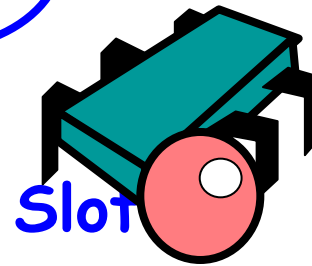
Still waiting ...

Try to
exchange item
and set state
to **BO**SY



Lock-free Exchanger

Partner showed up, take item and reset to EMPTY



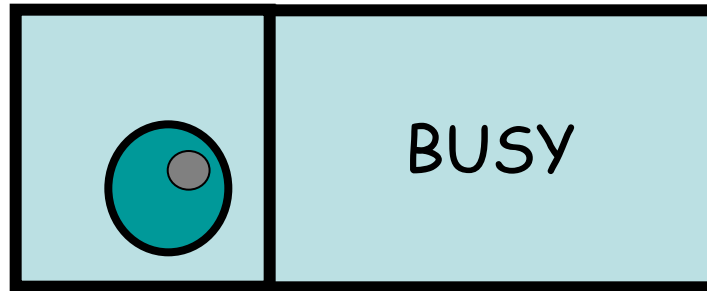
item

stamp/state

Lock-free Exchanger

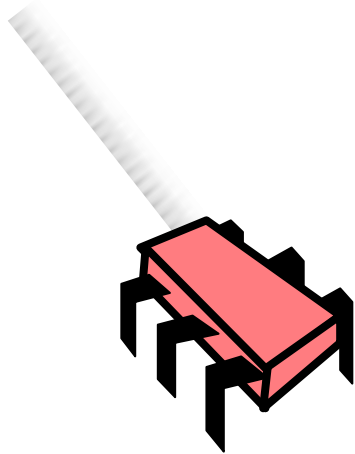
Partner showed up, take item and reset to EMPTY

Slot



item

stamp/state



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



Elimination Array

```
public class EliminationArray {  
    ...  
    public T visit(T value, int Range) throws TimeoutException {  
        int slot = random.nextInt(Range);  
        int nanodur = convertToNanos(duration, timeUnit);  
        return (exchanger[slot].exchange(value, nanodur)  
    }  
}
```

Elimination Stack Push

```
public void push(T value) {  
    ...  
    while (true) {  
        if (tryPush(node)) {  
            return;  
        } else try {  
            T otherValue =  
eliminationArray.visit(value,policy.Range);  
            if (otherValue == null) {  
                return;  
            }  
        }  
    }  
}
```

Elimination Stack Pop

```
public T pop() {  
    ...  
    while (true) {  
        if (tryPop()) {  
            return returnNode.value;  
        } else  
            try {  
                T otherValue =  
                    eliminationArray.visit(null,policy.Range);  
                if (otherValue != null) {  
                    return otherValue;  
                }  
            }  
    }  
}
```

Summary

- Quick reminder of the Stack structure.
- The Unbounded Lock-Free Stack.
- The Elimination Backoff Stack.



תודה רבה על ההקשבה

