Lecture 4:
Concurrent Queues and Stacks

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
With modifications by Lamont Samuels

pool

- Data Structure similar to Set
  - Does not necessarily provide contains() method
  - Allows the same item to appear more than once
  - get() and set()


```
public interface Pool<T> {
  void put(T item);
  T get();
}
```

Queues & Stacks

- Both: pool of items
- Queue
  - enq() & deq()
  - First-in-first-out (FIFO) order
- Stack
  - push() & pop()
  - Last-in-first-out (LIFO) order

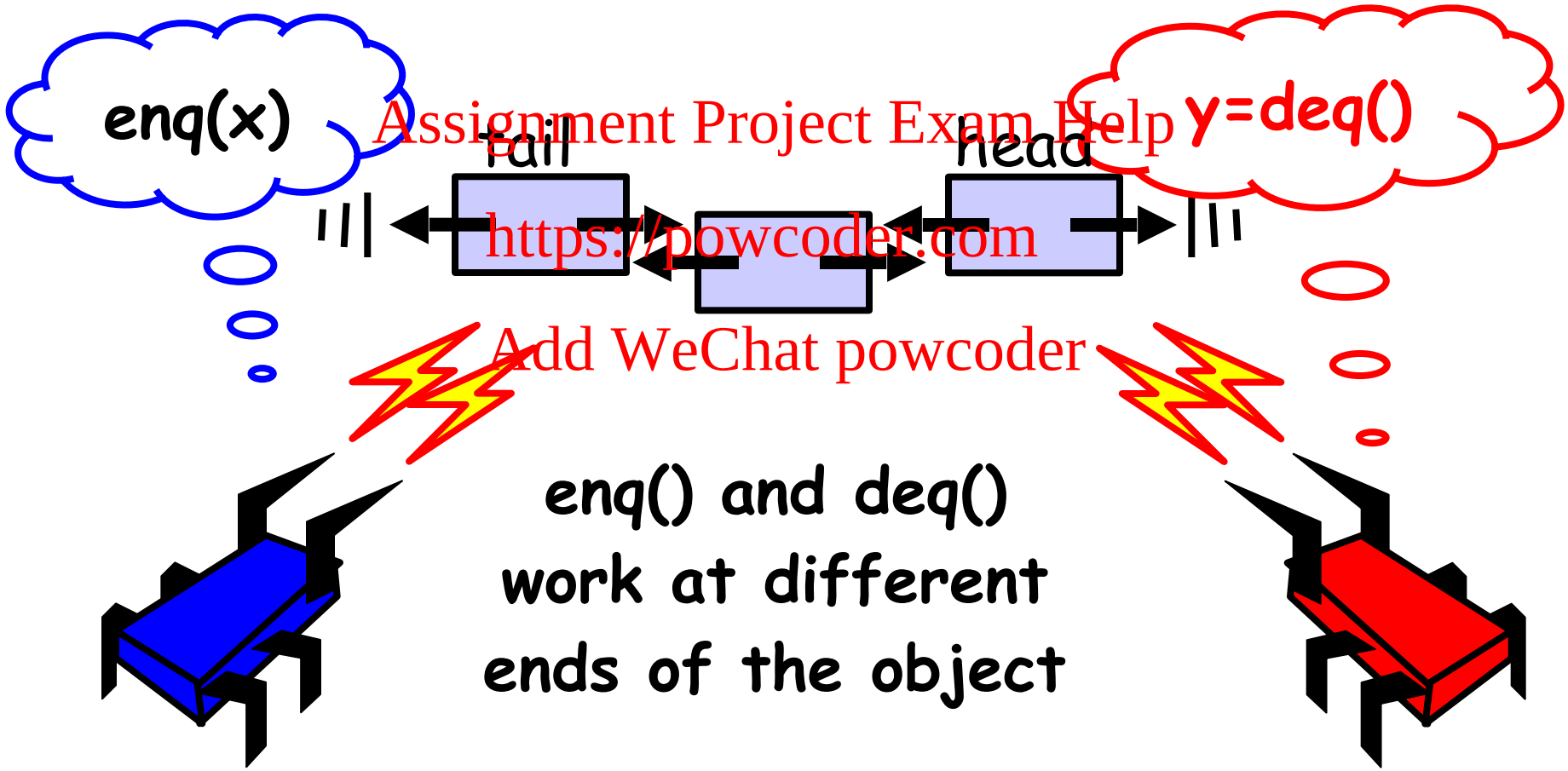
Bounded vs Unbounded

- Bounded
  - Fixed capacity
  - Good when resources an issue
- Unbounded
  - Holds any number of objects

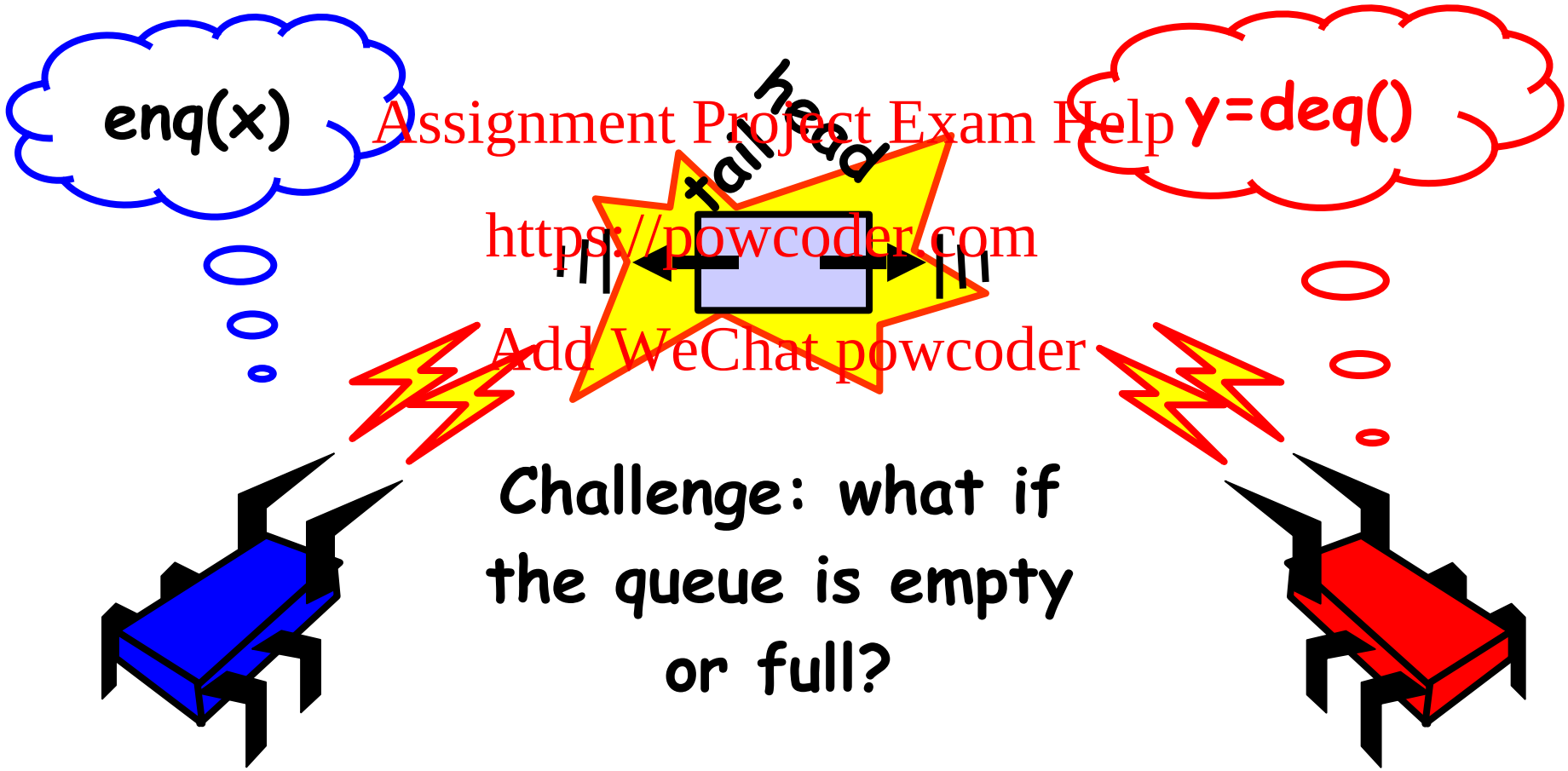
Blocking vs Non-Blocking

- Problem cases:
  - Removing from empty pool
  - Adding to full (bounded) pool
- Blocking
  - Caller waits until state changes
- Non-Blocking
  - Method throws exception

Queue: Concurrency



Concurrency



Challenge: what if the queue is empty or full?

lock

- enqLock/deqLock
  - At most one enqueuer/dequeuer at a time can manipulate the queue's fields
- Two locks
  - Enqueuer does not lock out dequeuer
  - vice versa
- Association
  - enqLock associated with notFullCondition
  - deqLock associated with notEmptyCondition

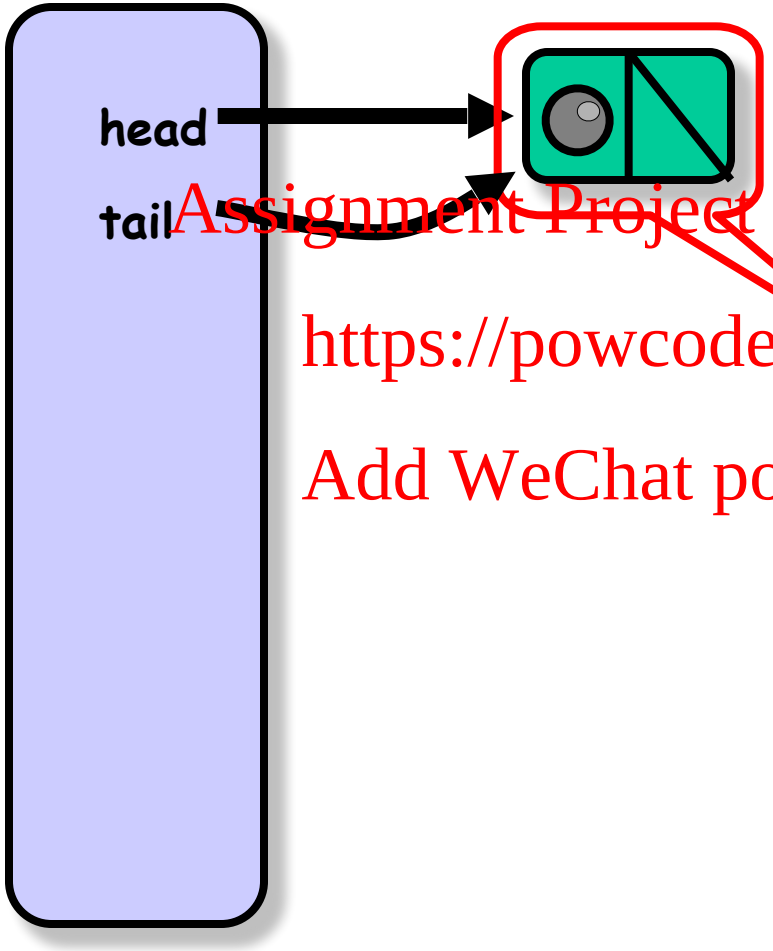
enqueue

1. Acquires enqLock
2. Reads the size field
3. If full, enqueuer must wait until dequeuer makes room
4. enqueuer waits on notFullCondition field, releasing enqLock temporarily, and blocking until that condition is signaled.
5. Each time the thread awakens, it checks whether there is a room, and if not, goes back to sleep
6. Insert new item into tail
7. Release enqLock
8. If queue was empty, notify/signal waiting dequeuers

dequeue

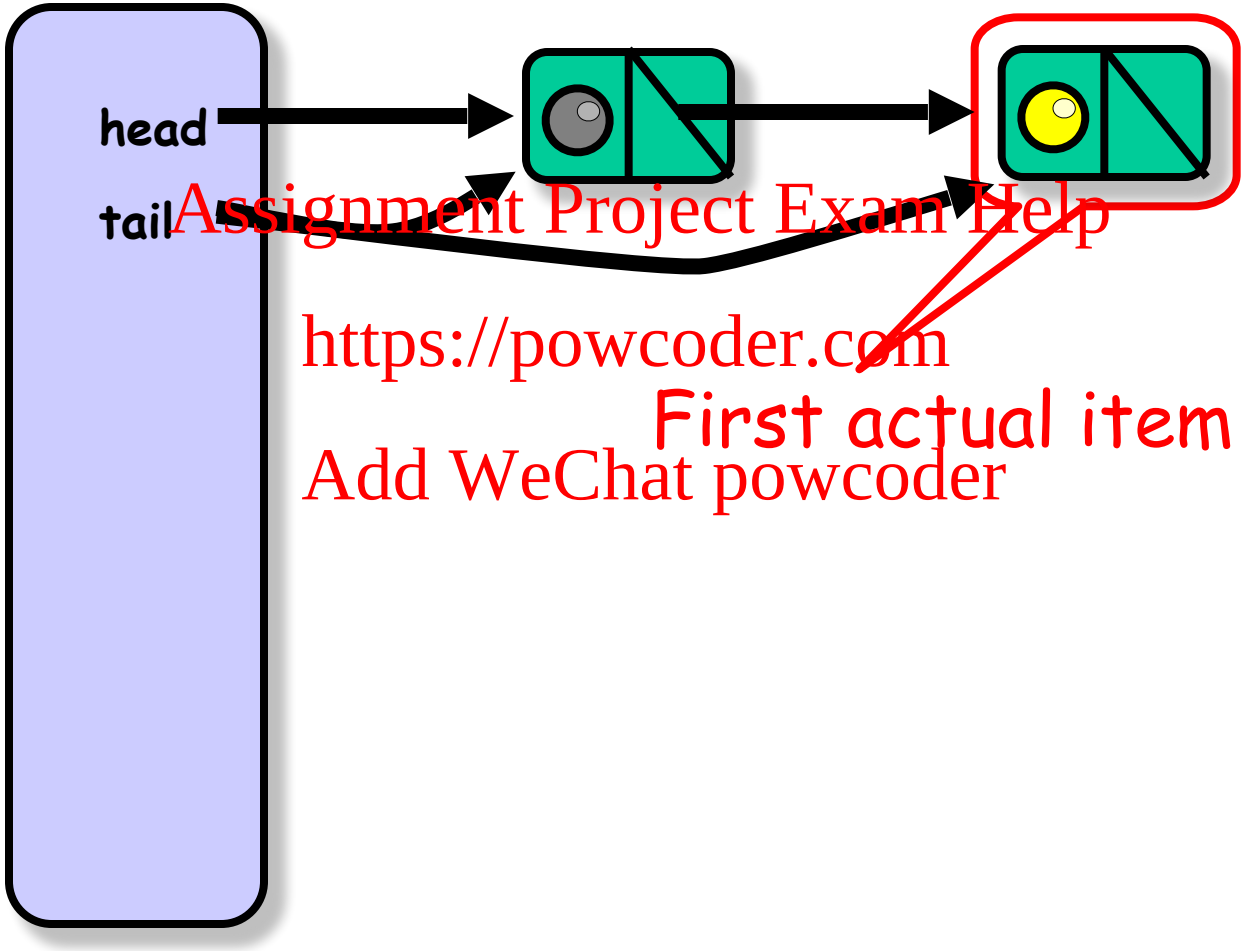
1. Acquires deqLock
2. Reads the size field
3. If empty, dequeuer must wait until item is enqueued
4. dequeuer waits on notEmptyCondition field, releasing deqLock temporarily, and blocking until that condition is signaled.
5. Each time the thread awakens, it checks whether item was enqueued, and if not, goes back to sleep
6. Assigne the value of head's next node to "result" and reset head to head's next node
7. Release deqLock
8. If queue was full, notify/signal waiting enqueuers
9. Return "result"

Bounded Queue

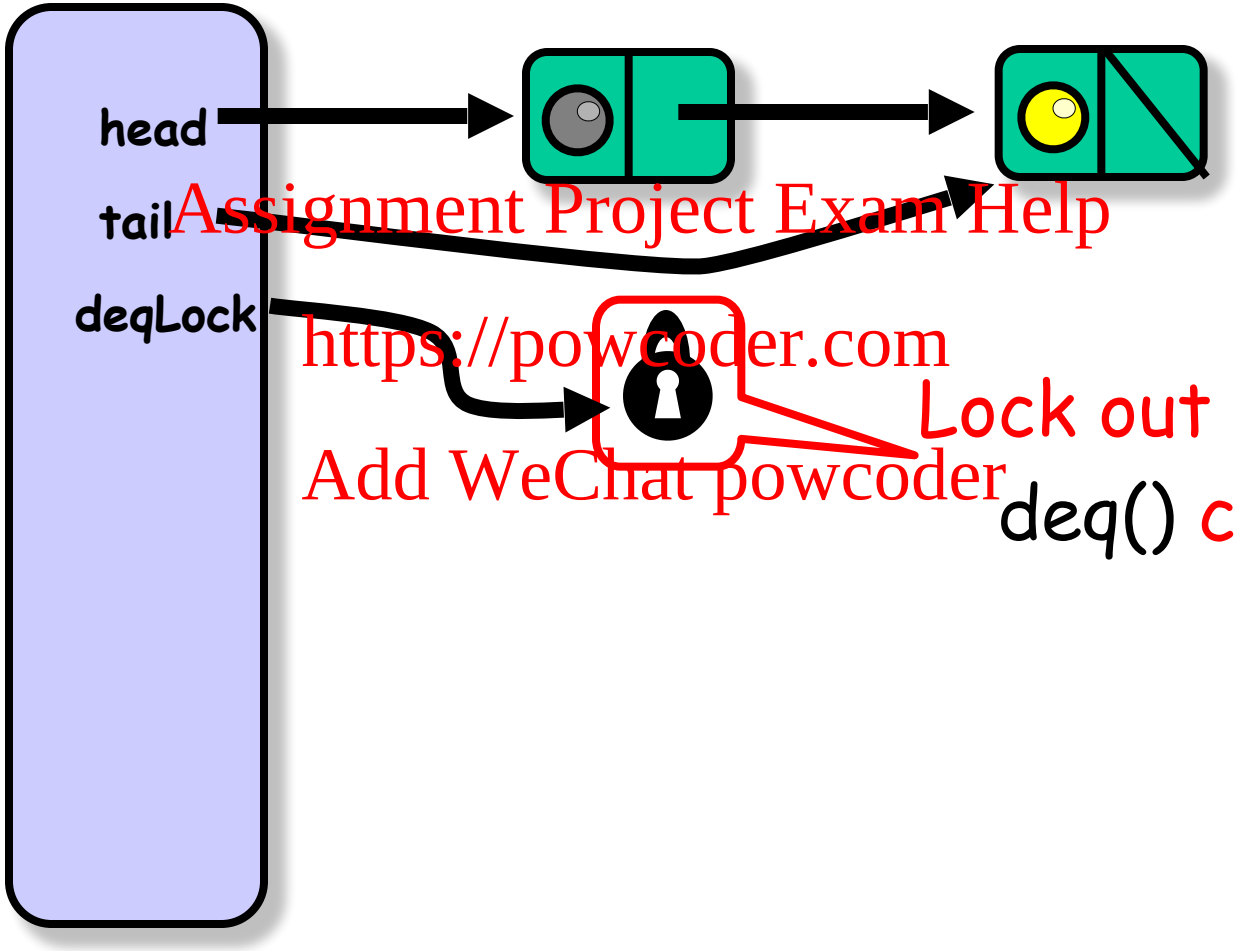


Sentinel

Bounded Queue

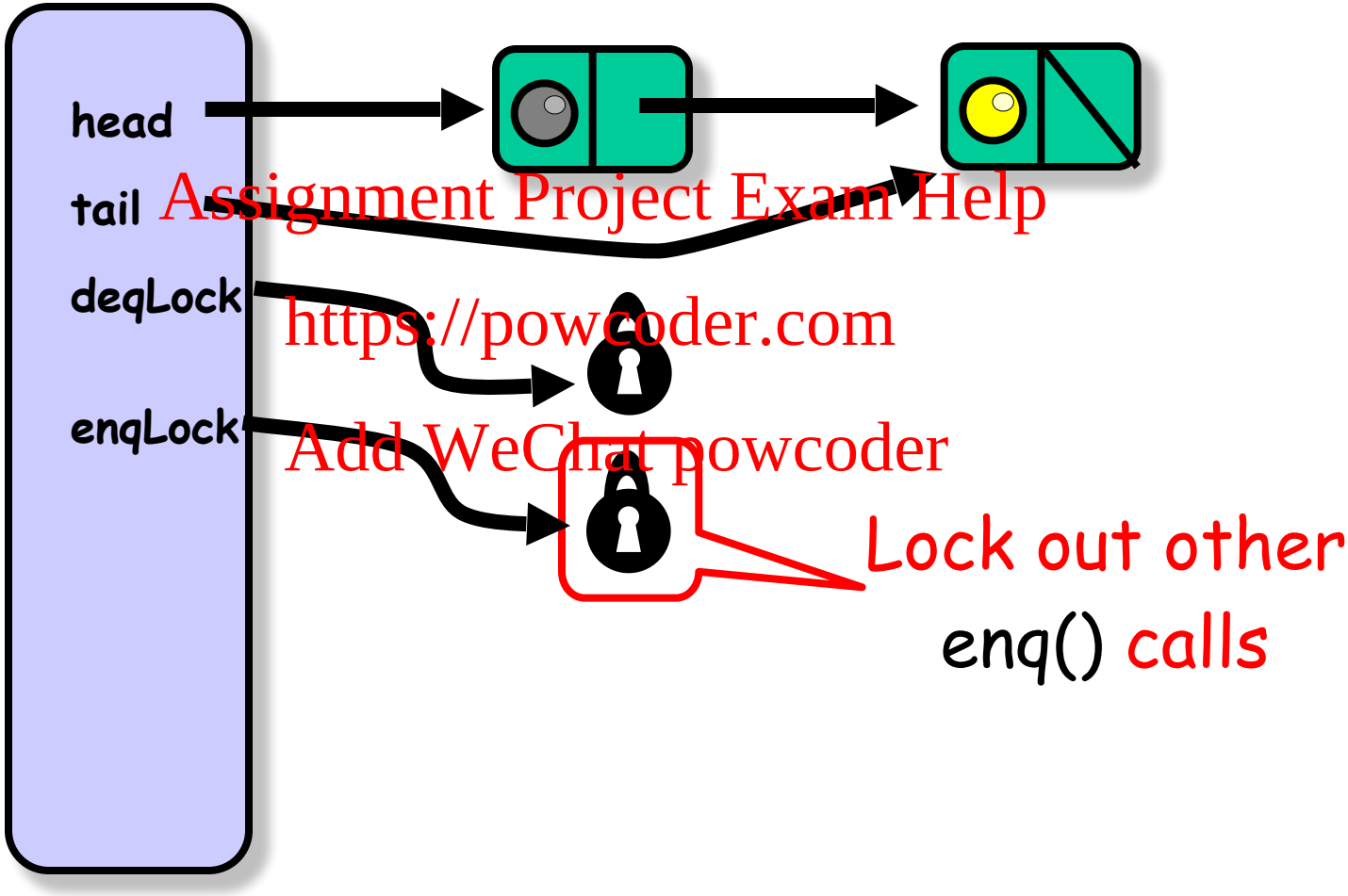


Bounded Queue

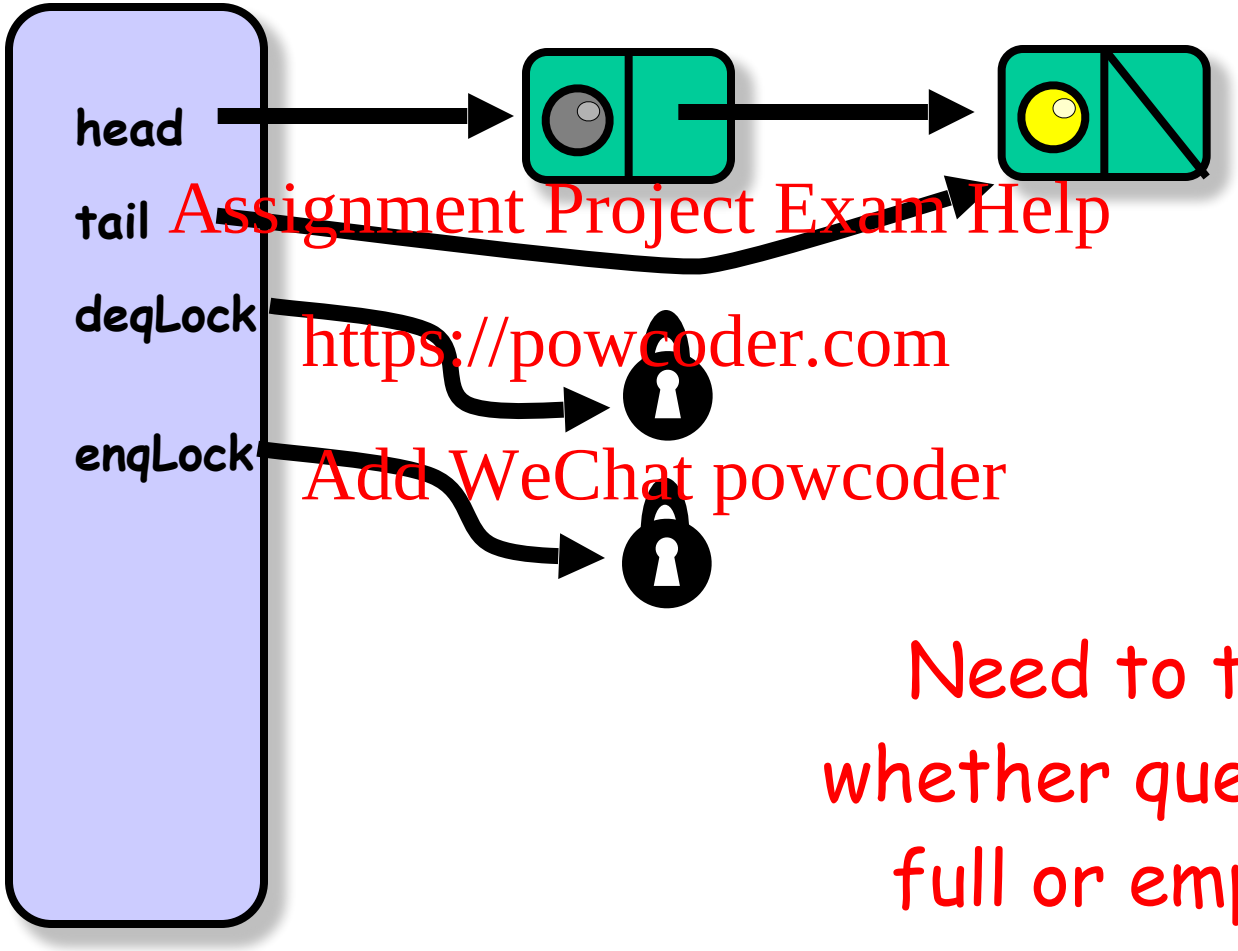


Lock out other deq() calls

Bounded Queue

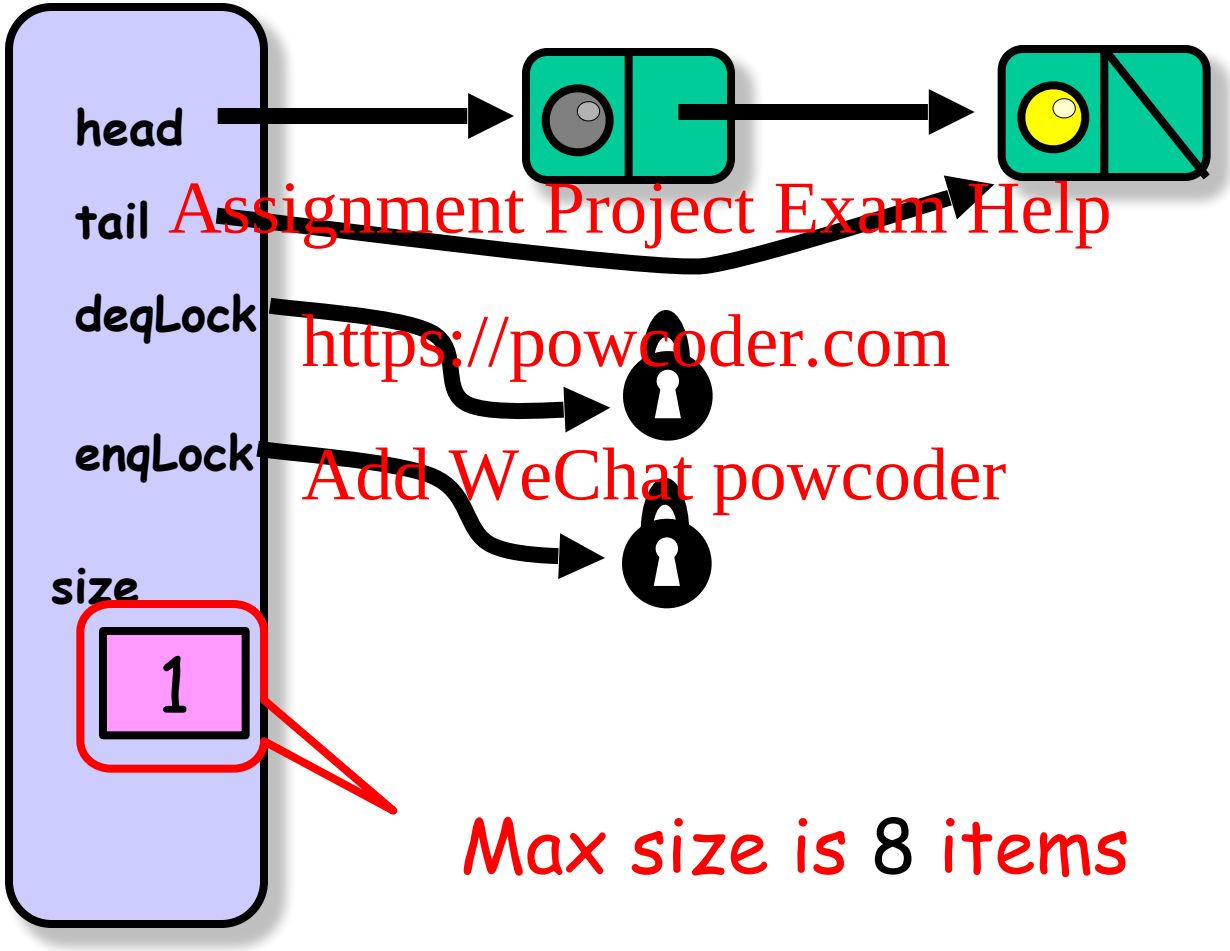


Not Done Yet

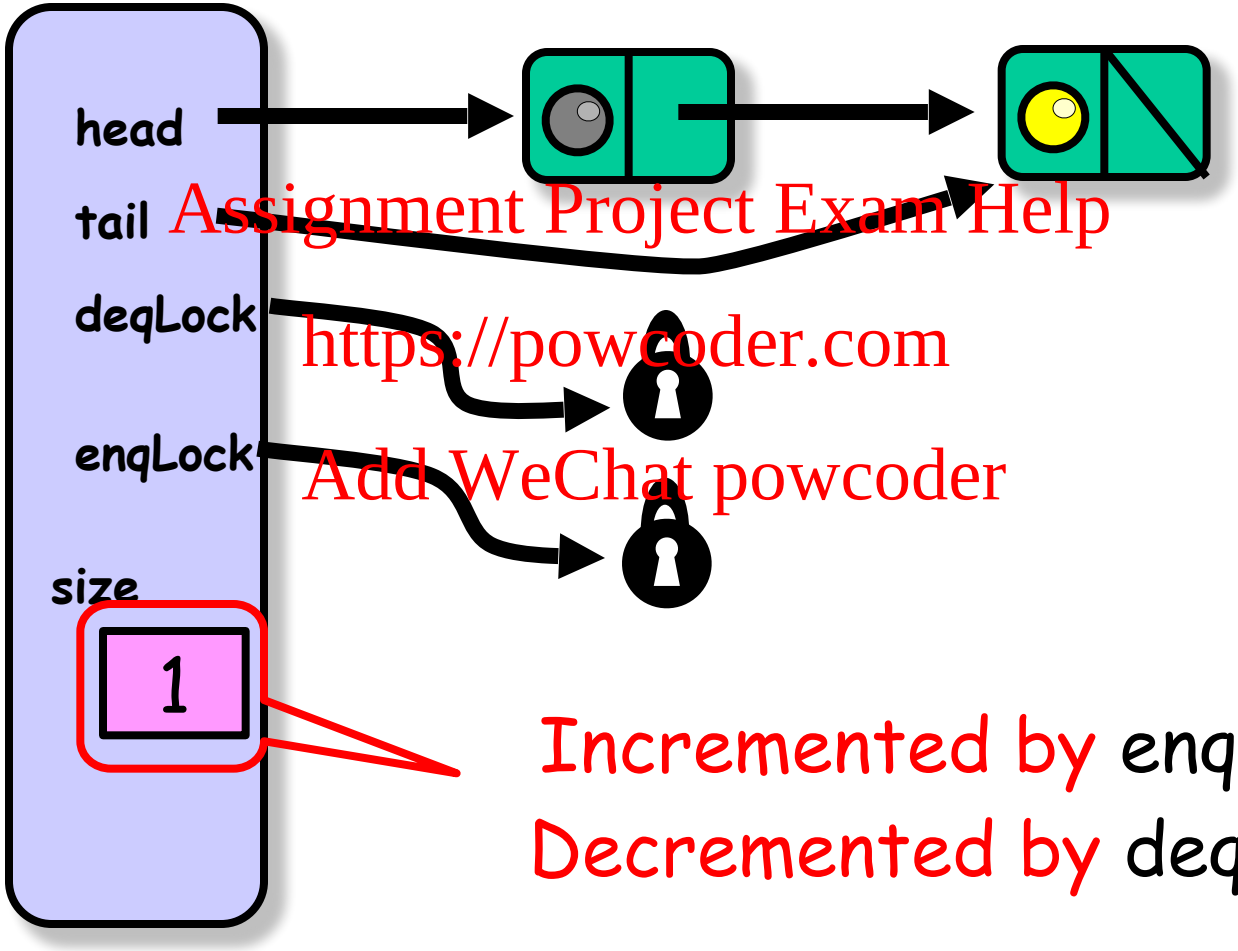


Need to tell whether queue is full or empty

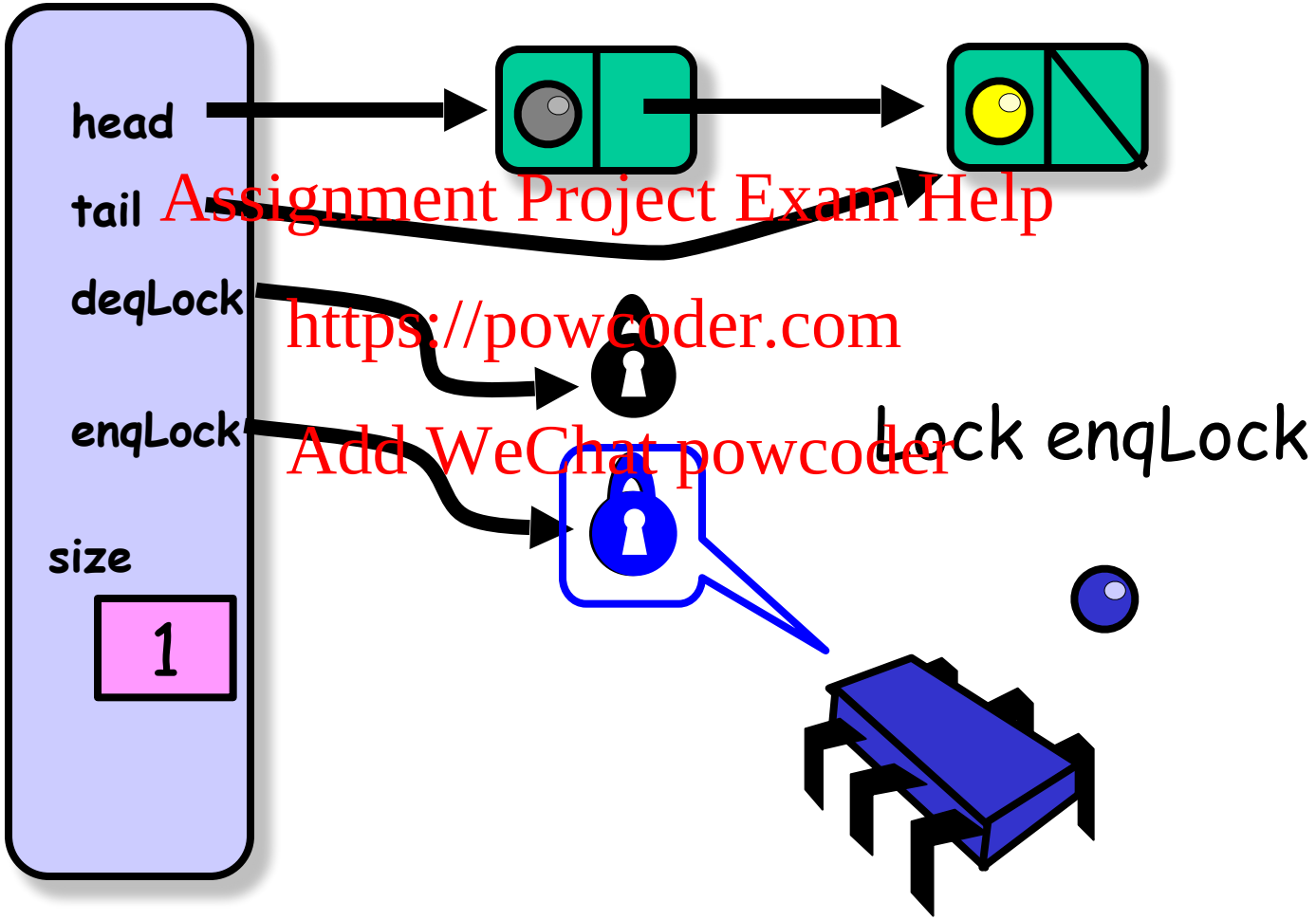
Not Done Yet



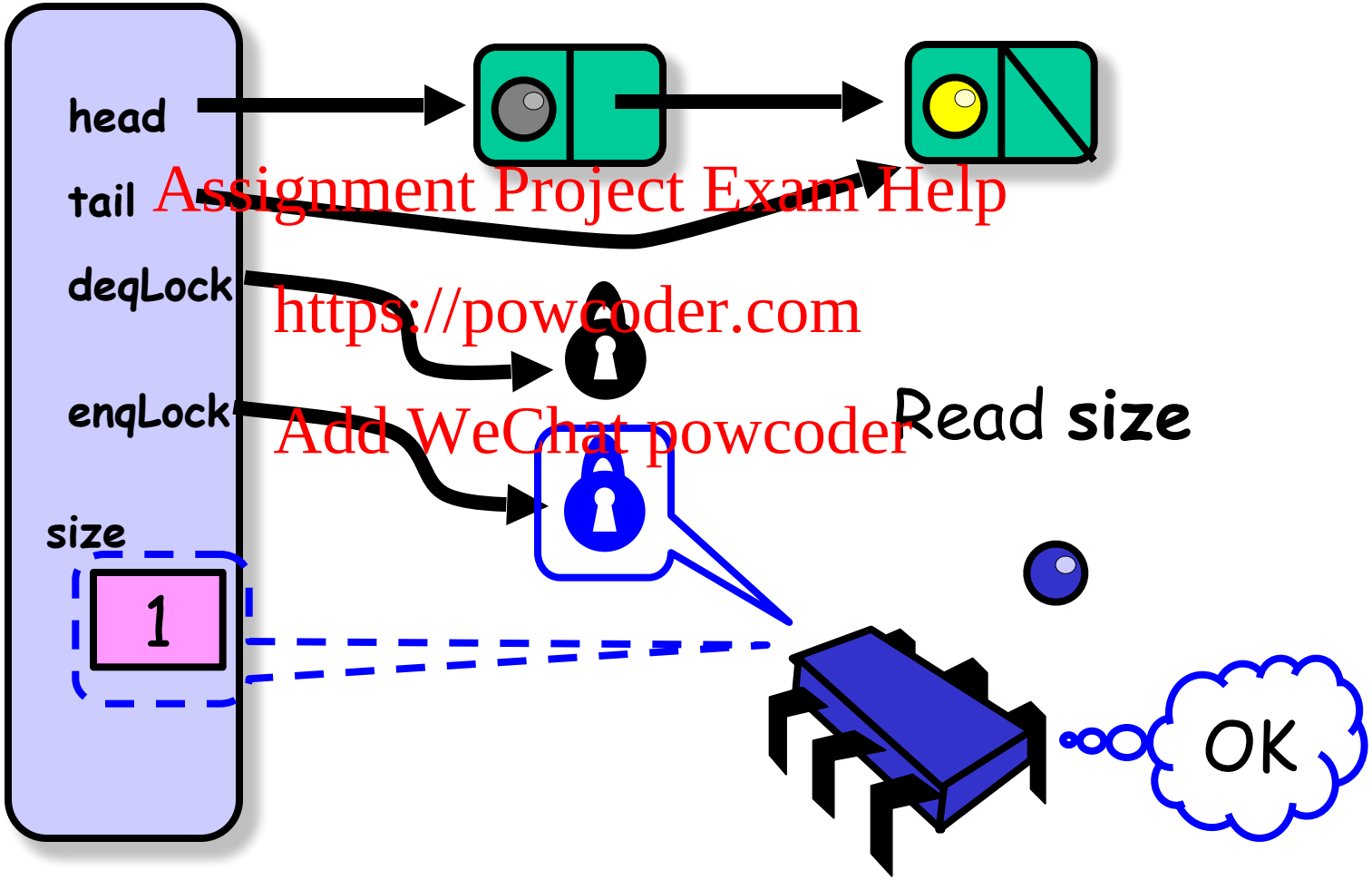
Not Done Yet



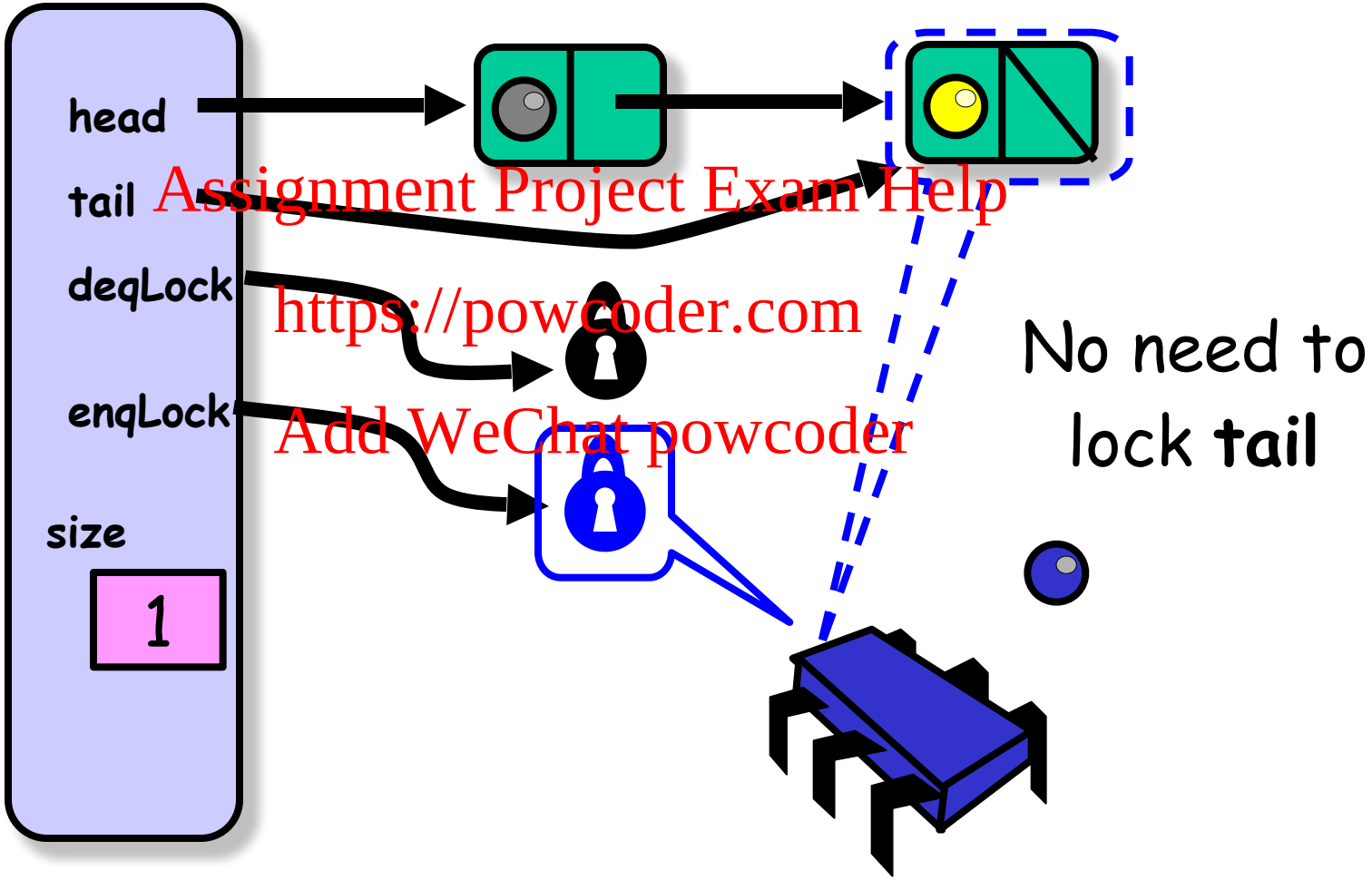
Enqueuer



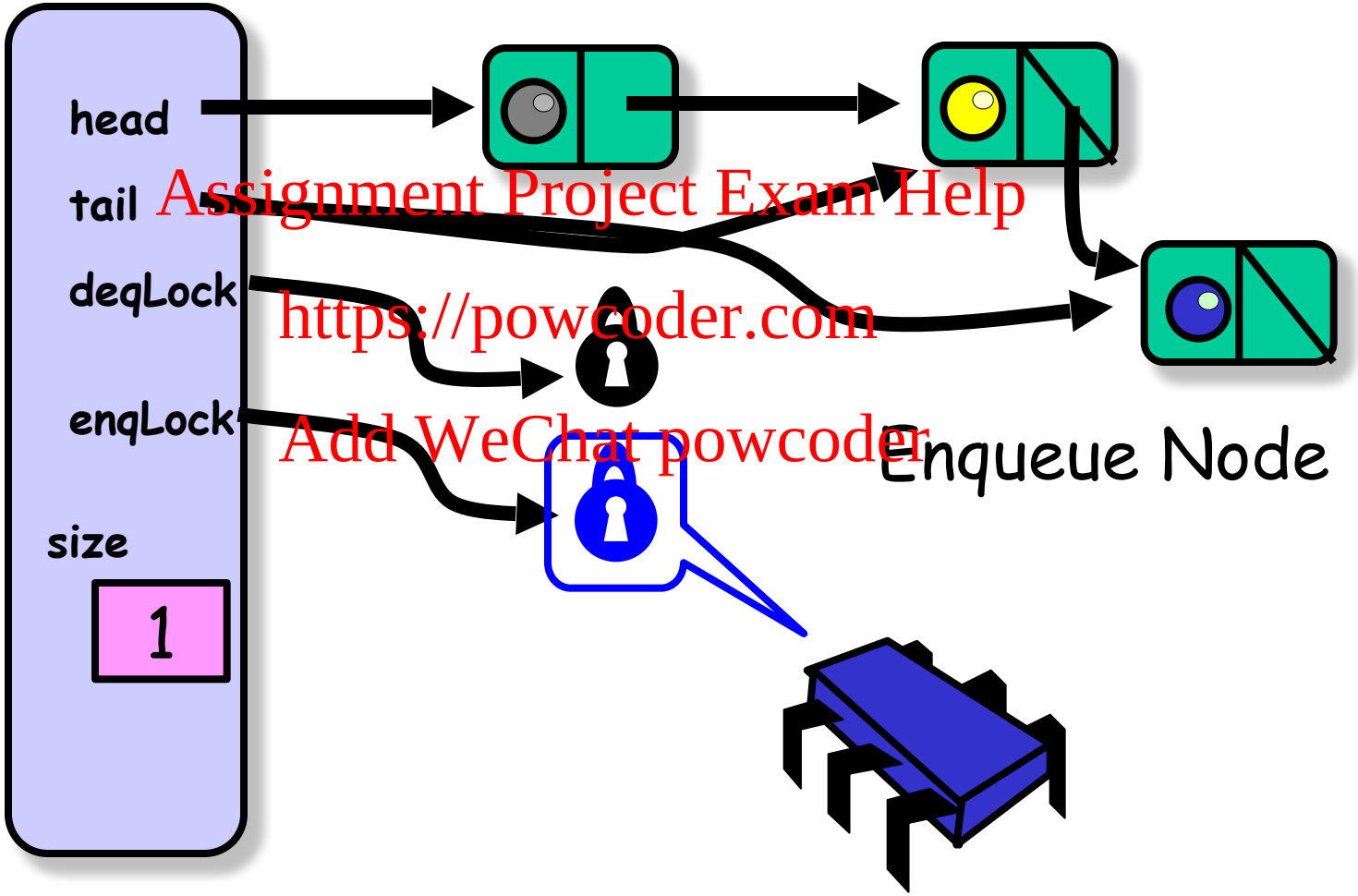
Enqueuer



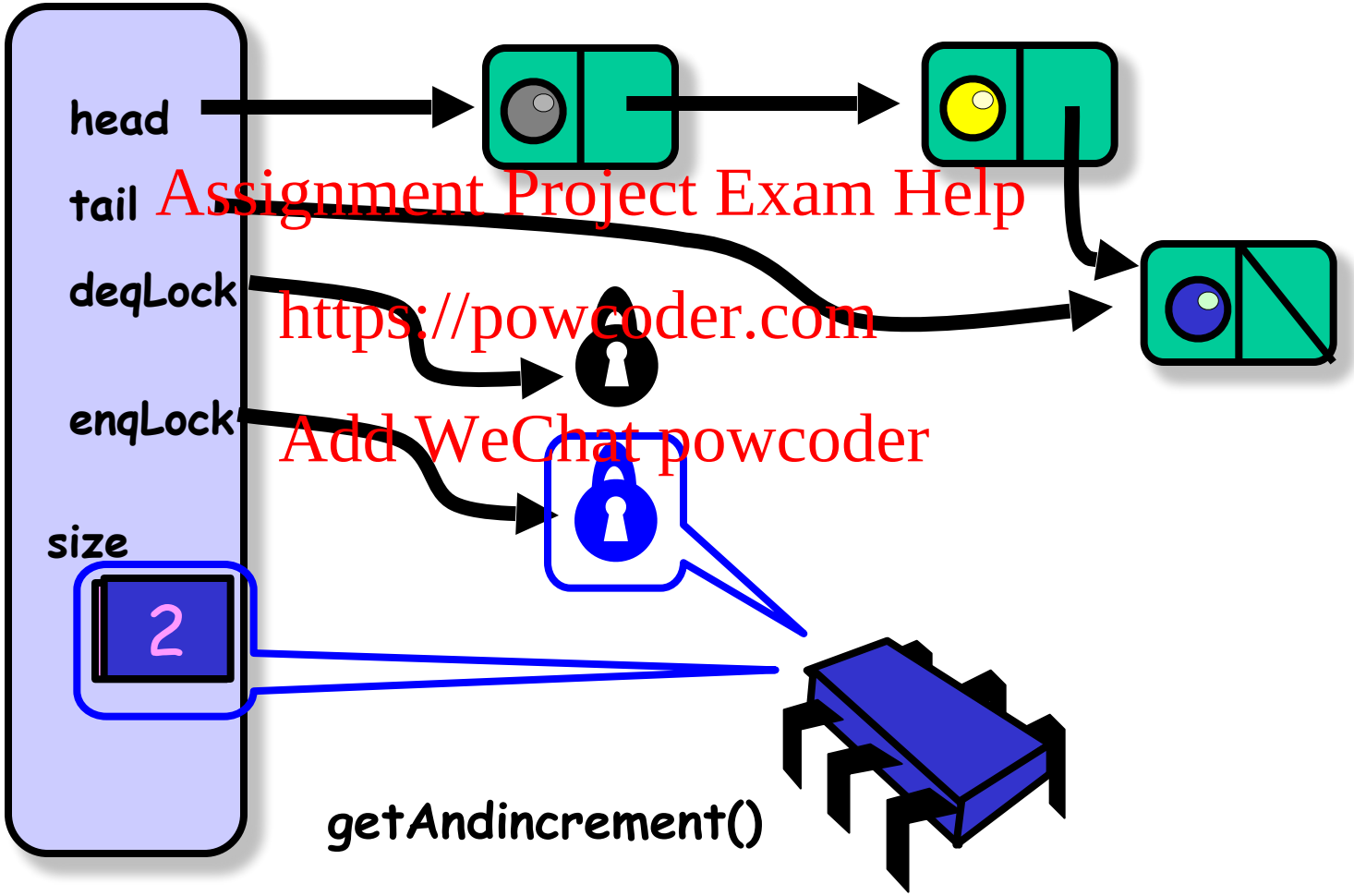
Enqueuer



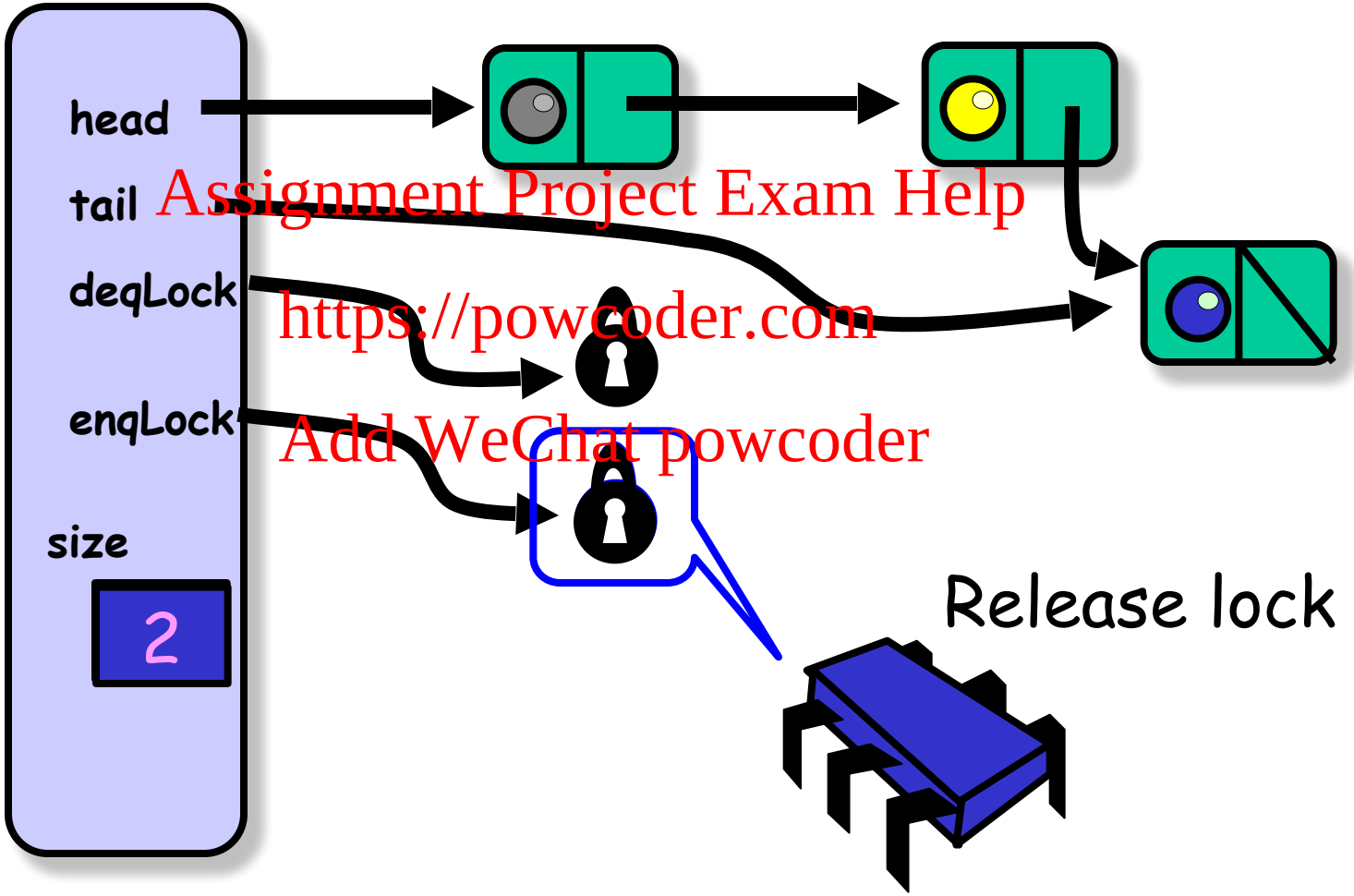
Enqueuer



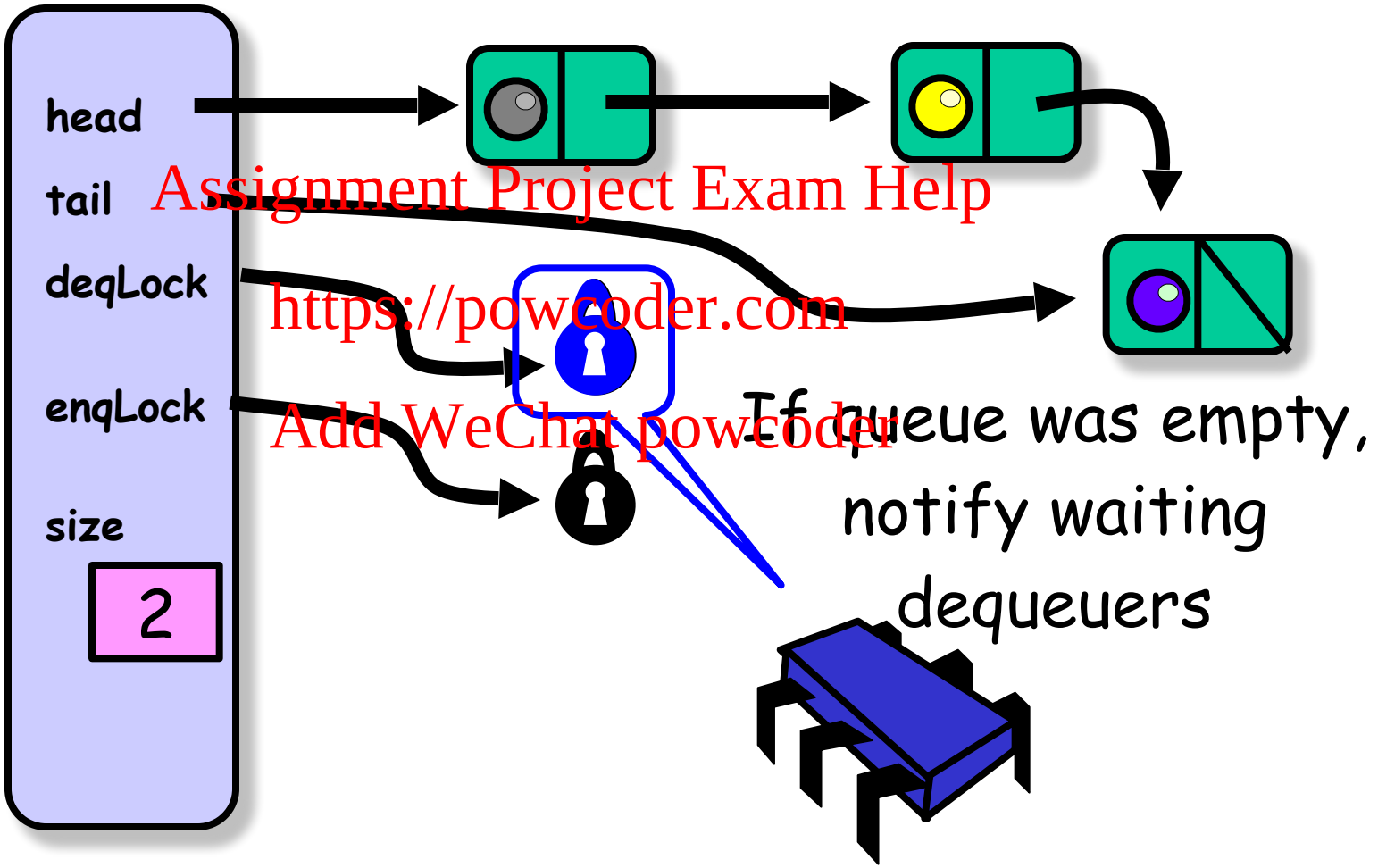
Enqueuer



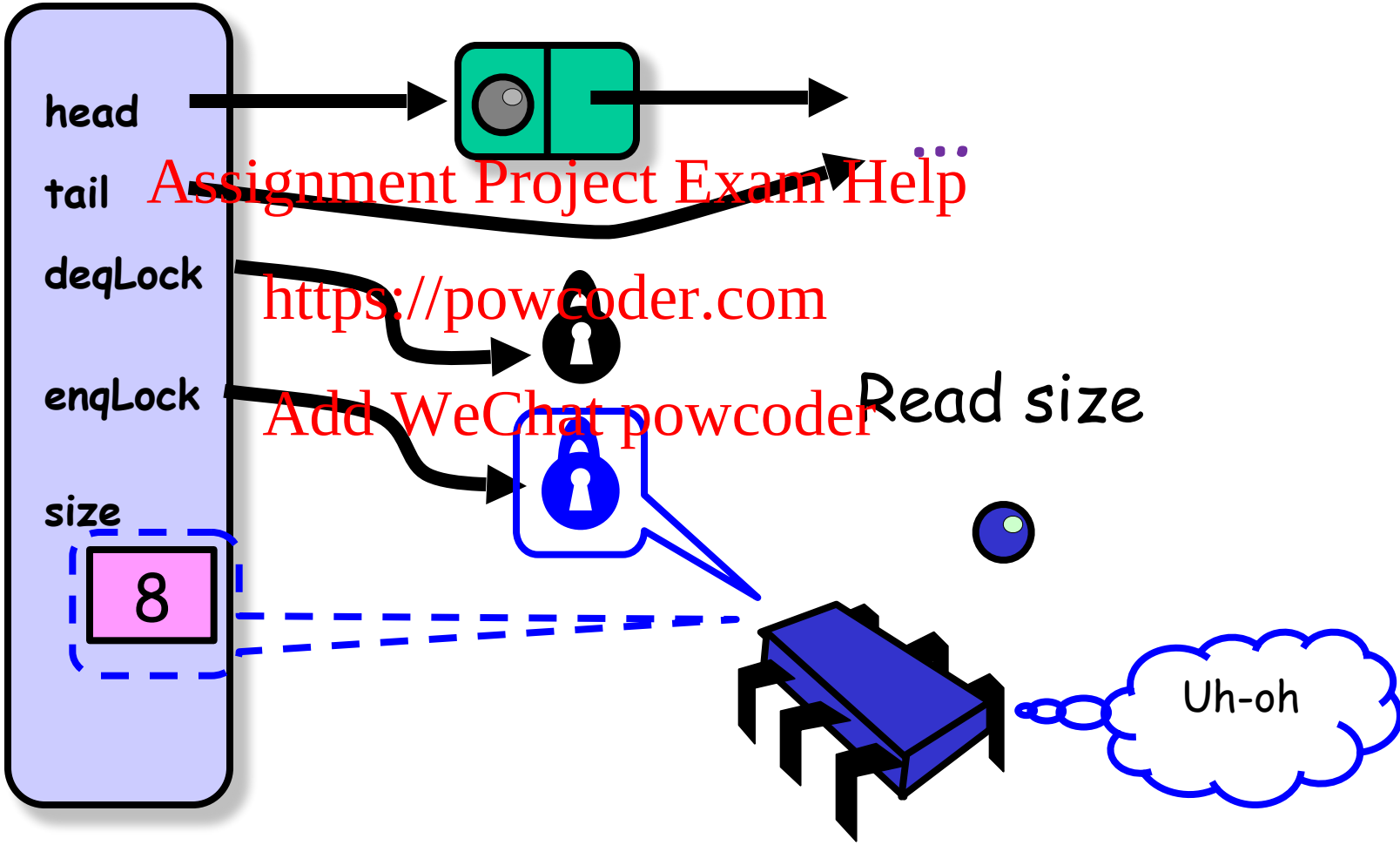
Enqueuer



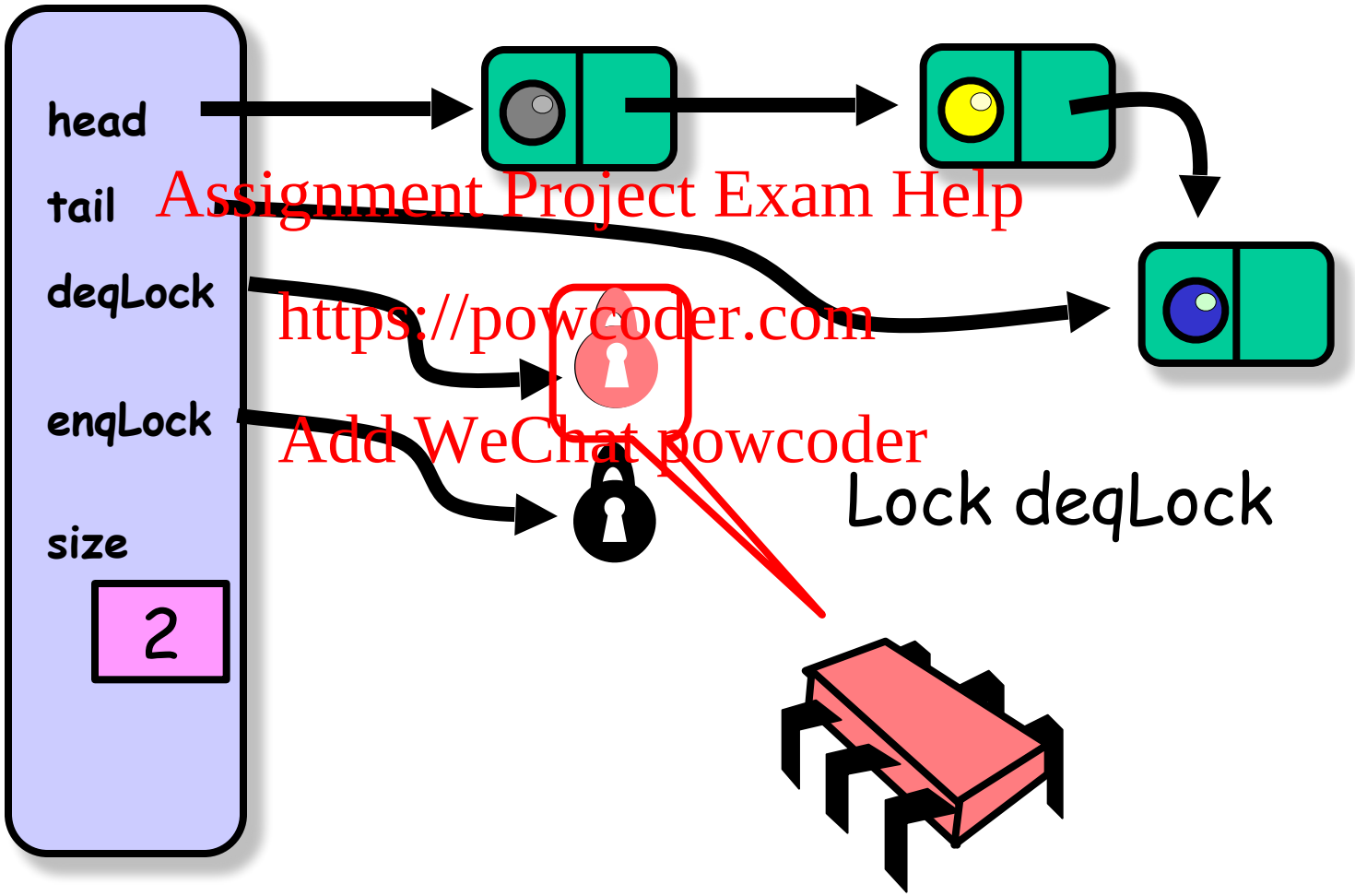
Enqueuer



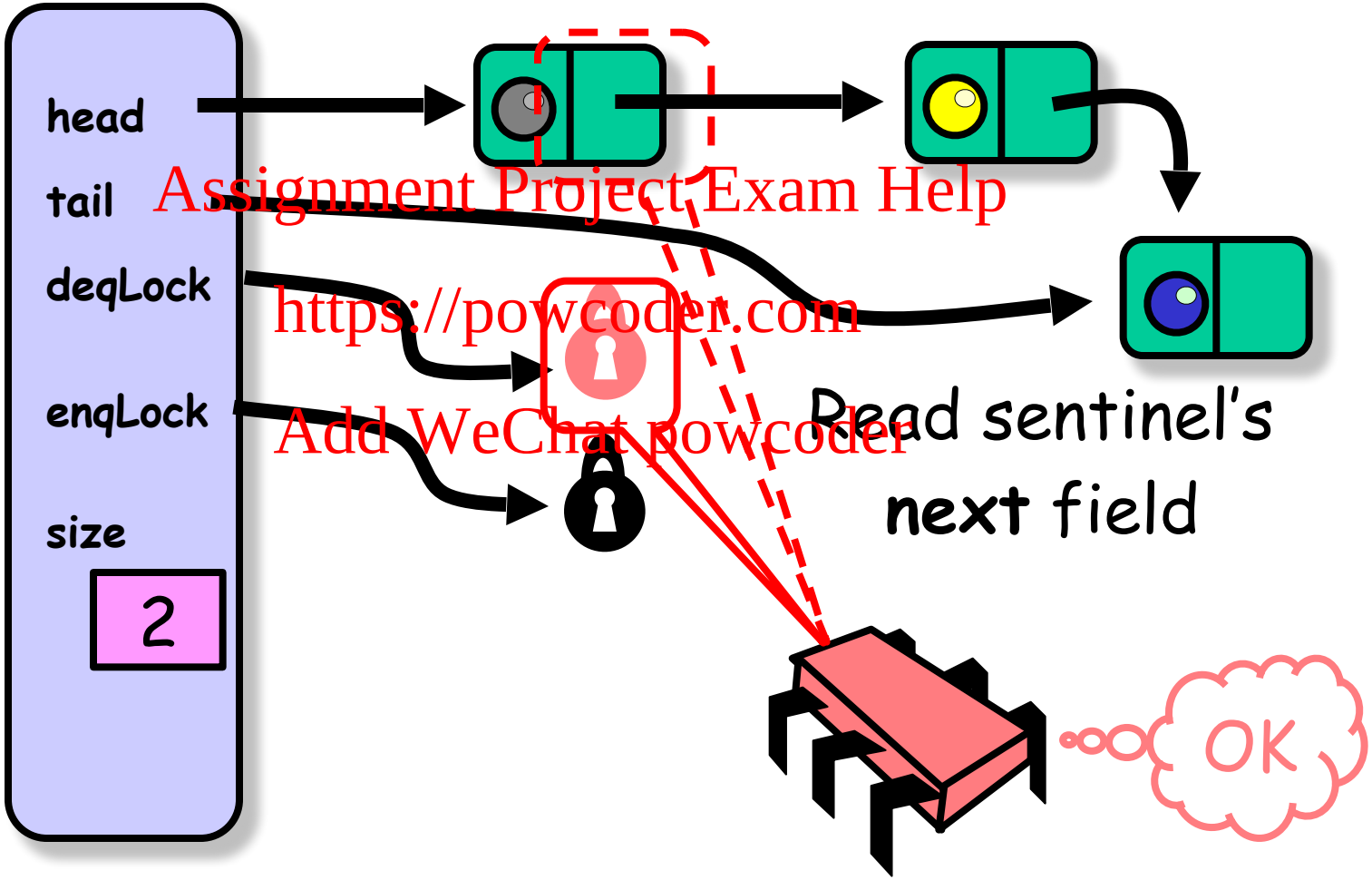
Unsuccesful Enqueuer



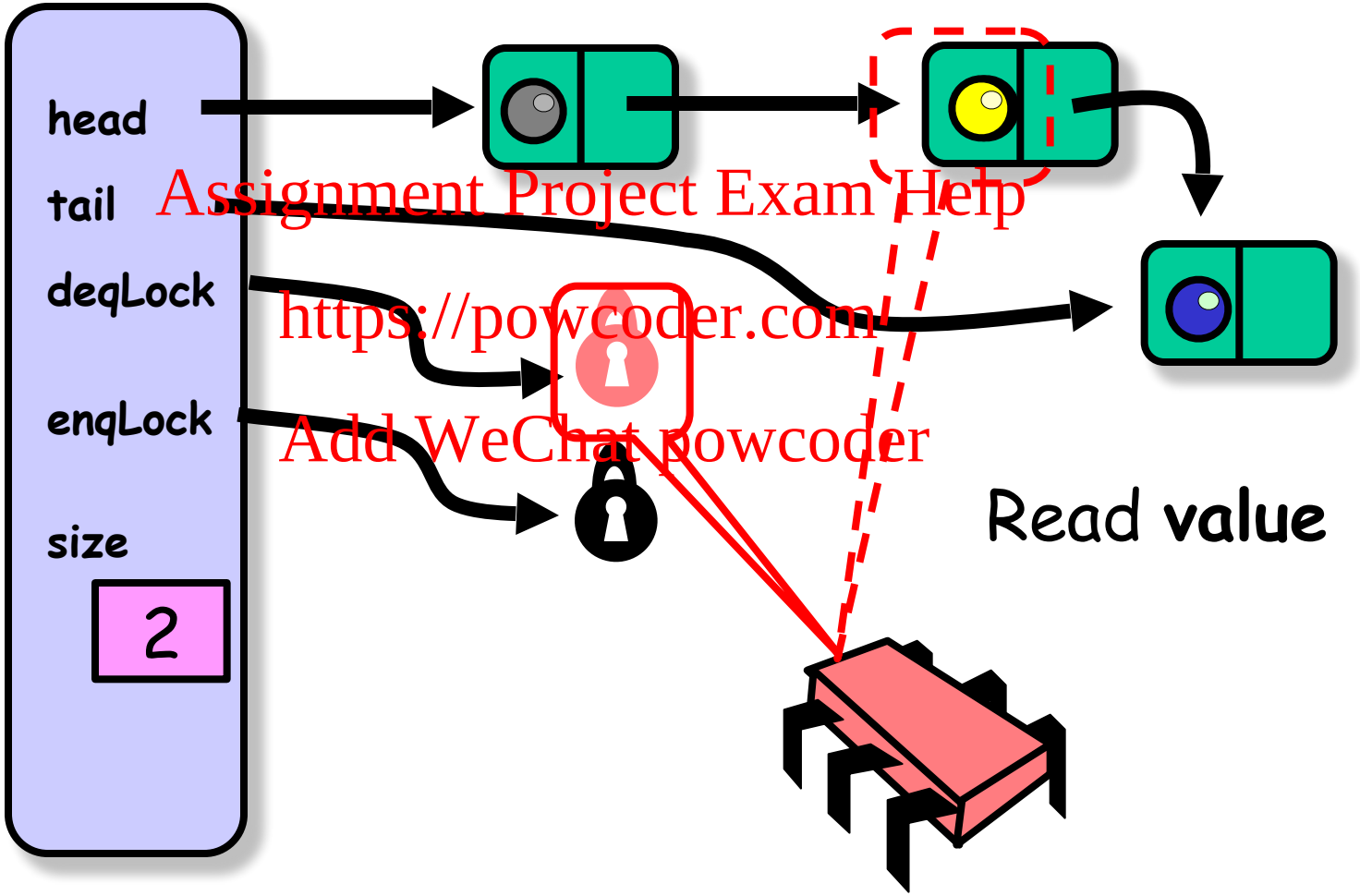
Dequeuer



Dequeuer

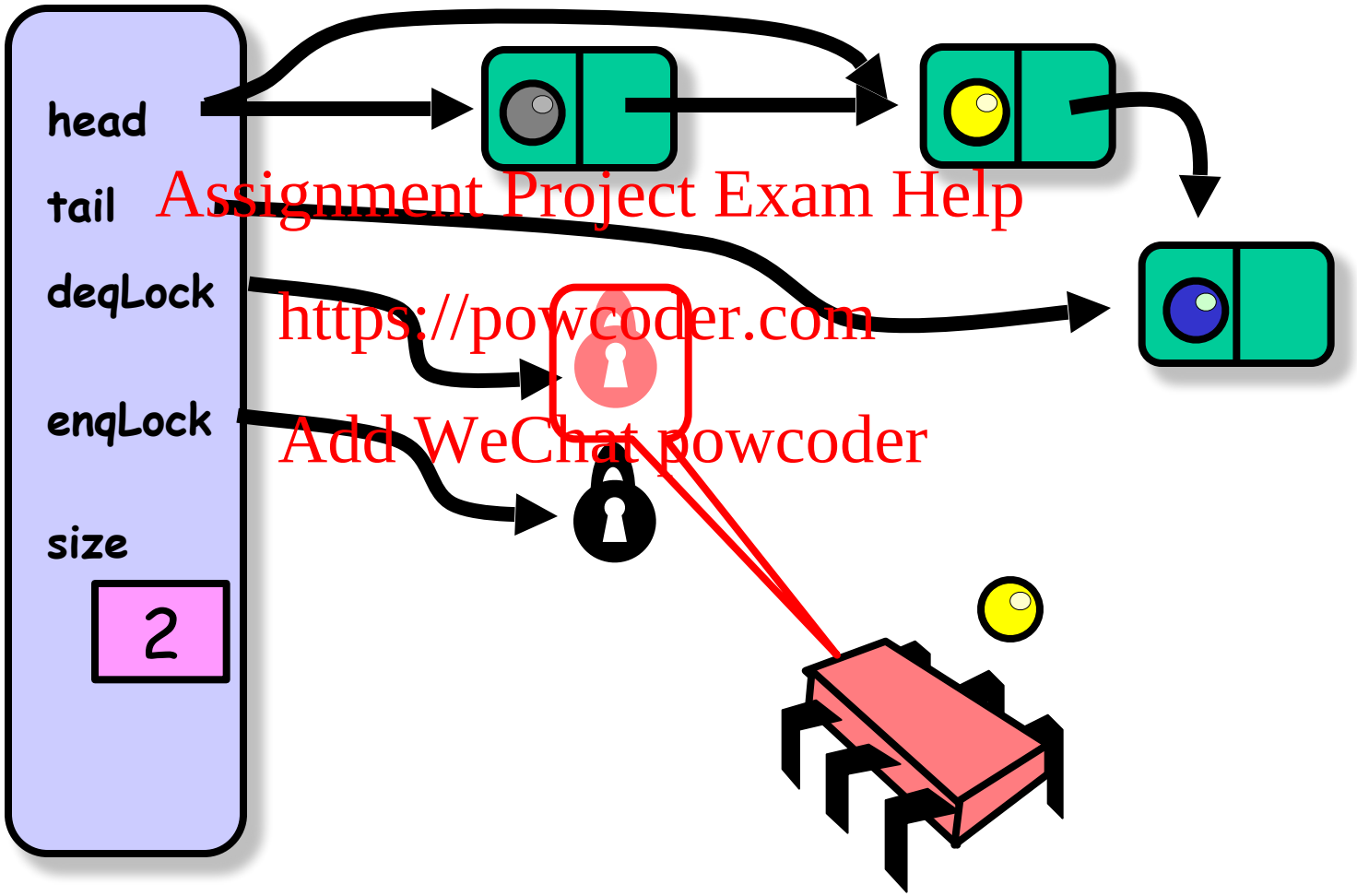


Dequeuer

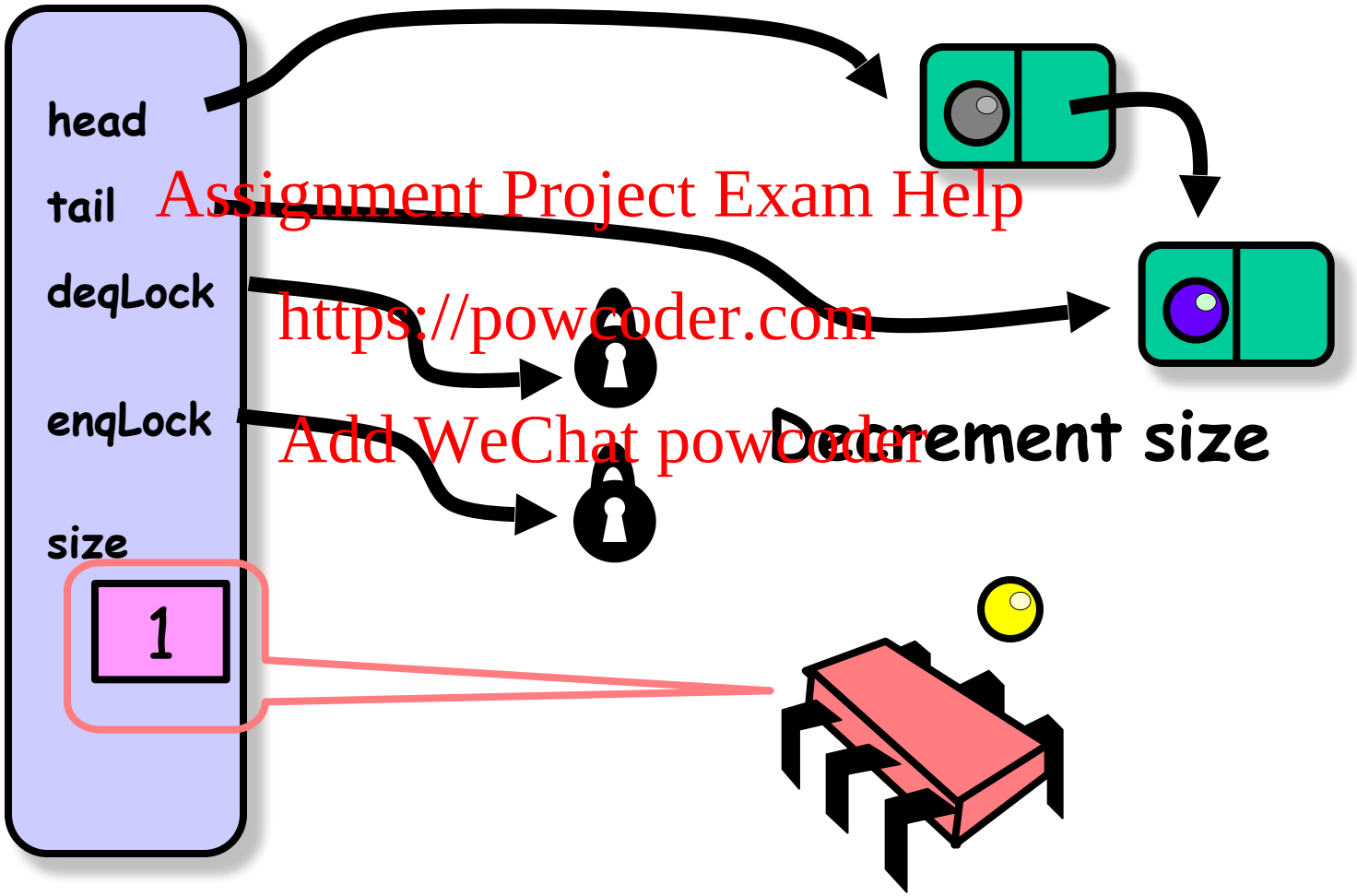


Dequeuer

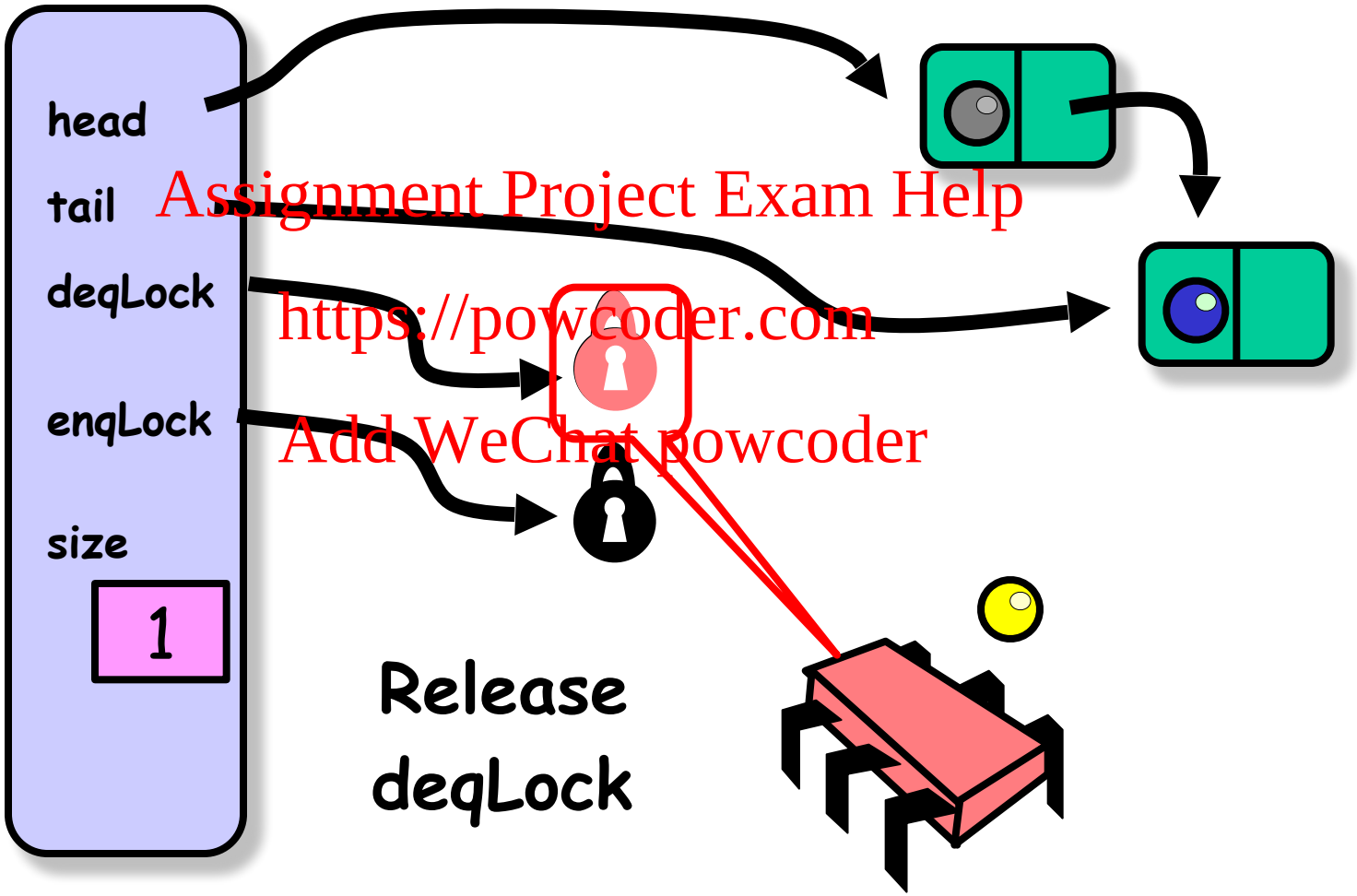
Make first Node new sentinel



Dequeuer



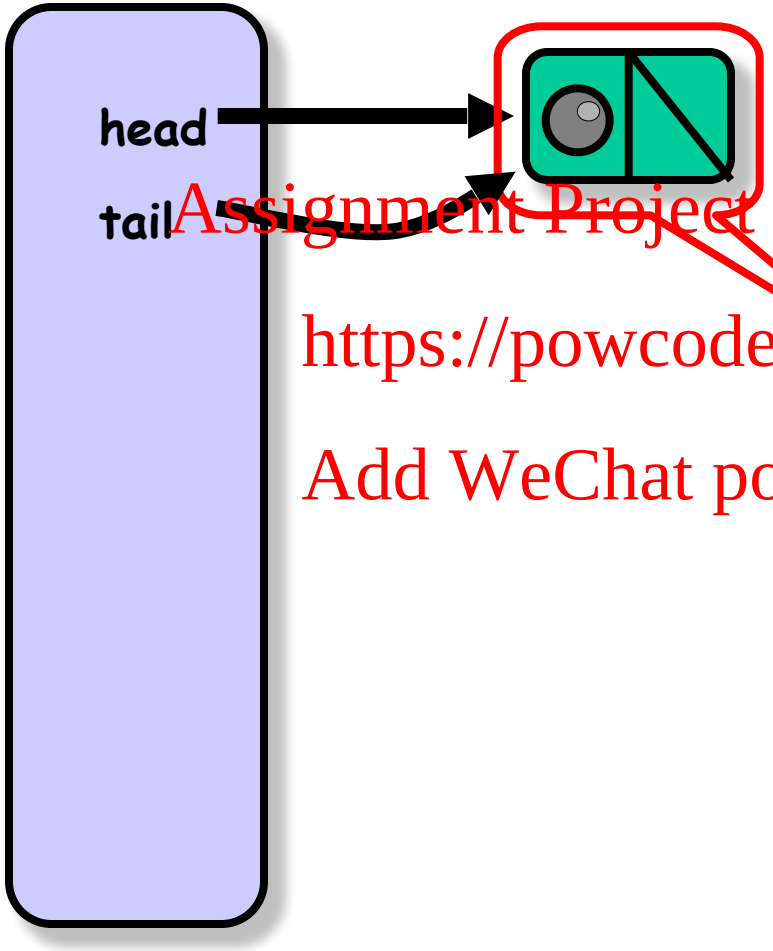
Dequeuer



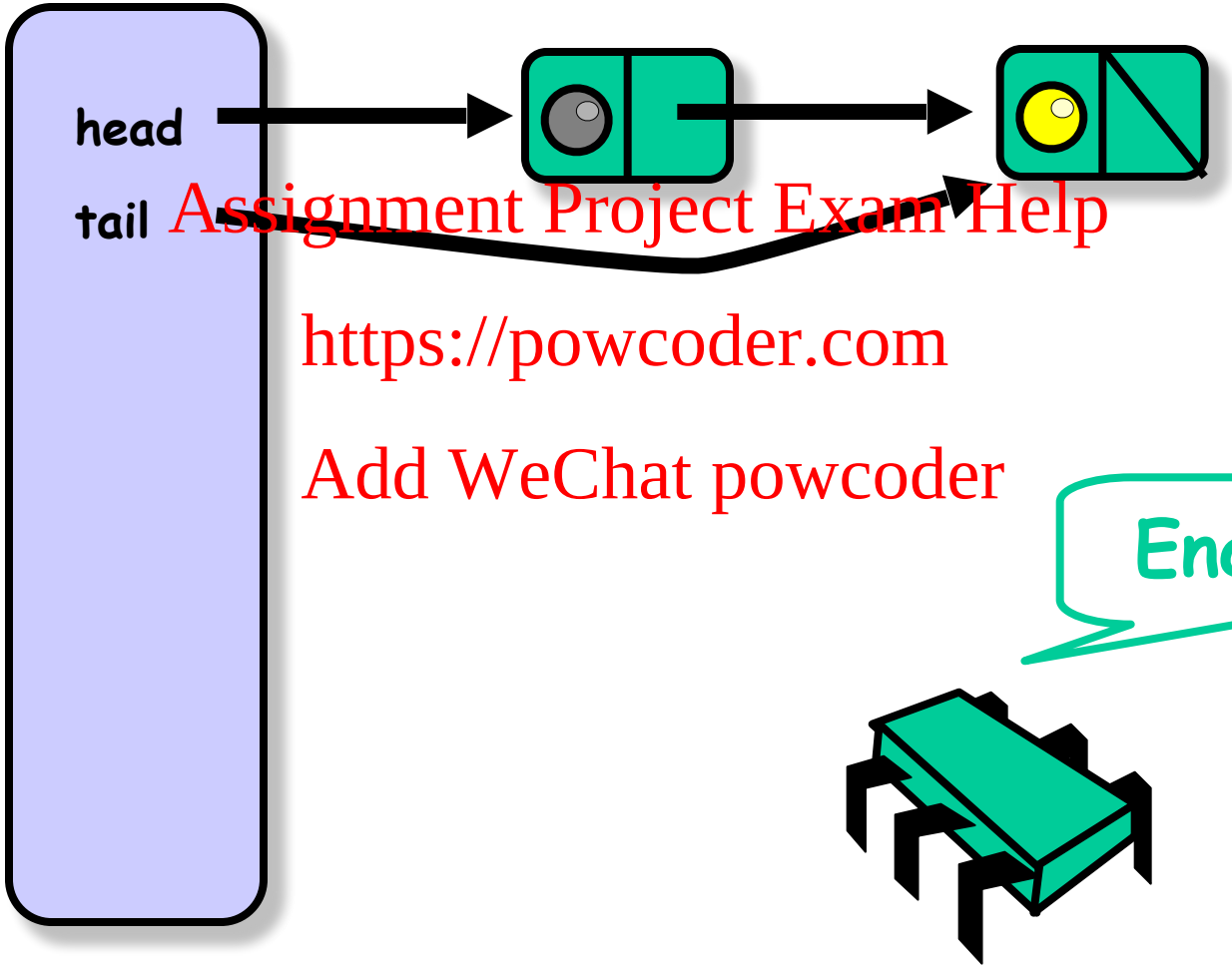
Unbounded Lock-Free Queue (Nonblocking)

- Unbounded
  - No need to count the number of items
- Lock-free
  - Use AtomicReference<V>
    - An object reference that may be updated atomically.
  - boolean compareAndSet(V expect, V update)
    - Atomically sets the value to the given updated value if the current value == the expected value.
- Nonblocking
  - No need to provide conditions on which to wait

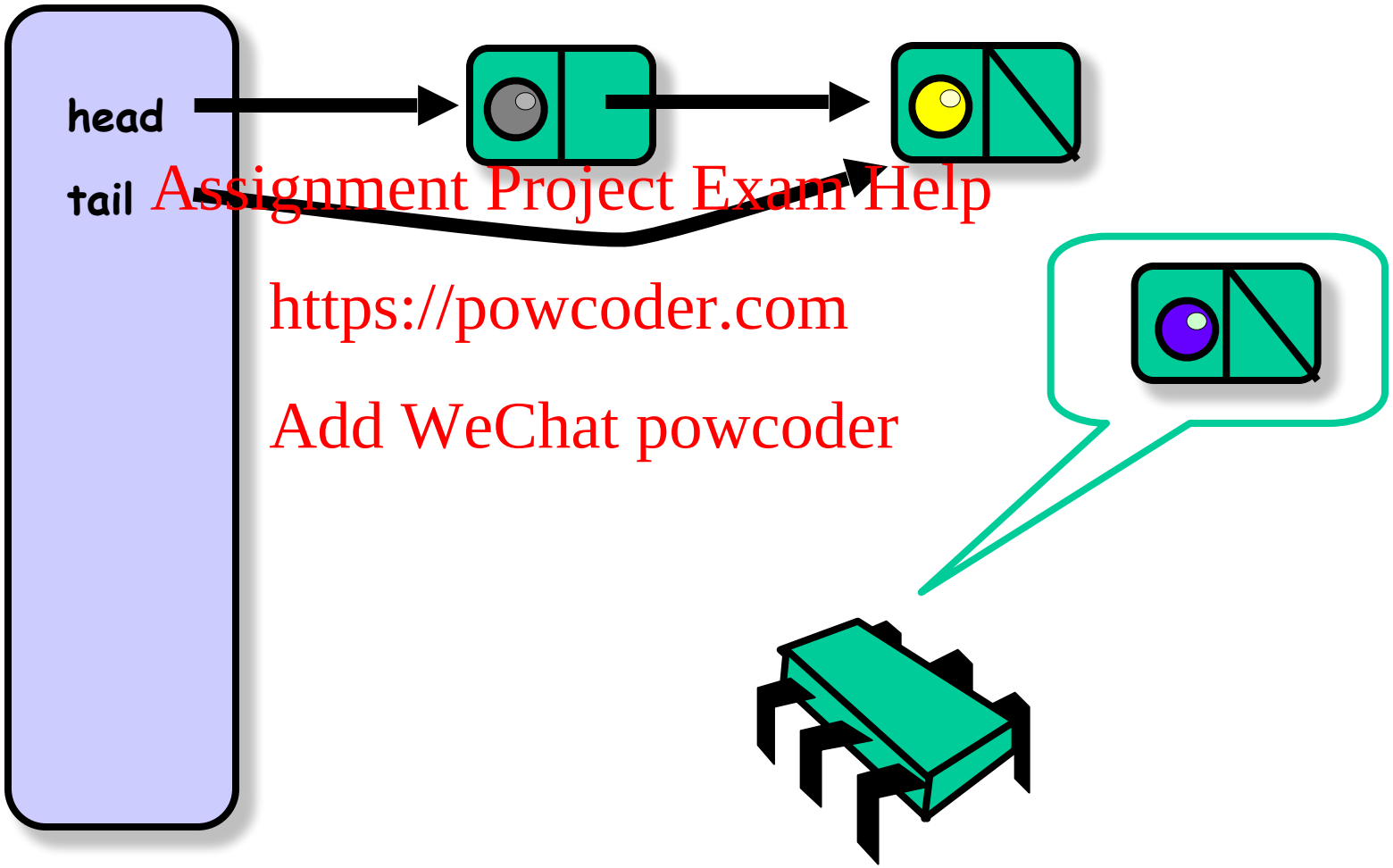
A Lock-Free Queue



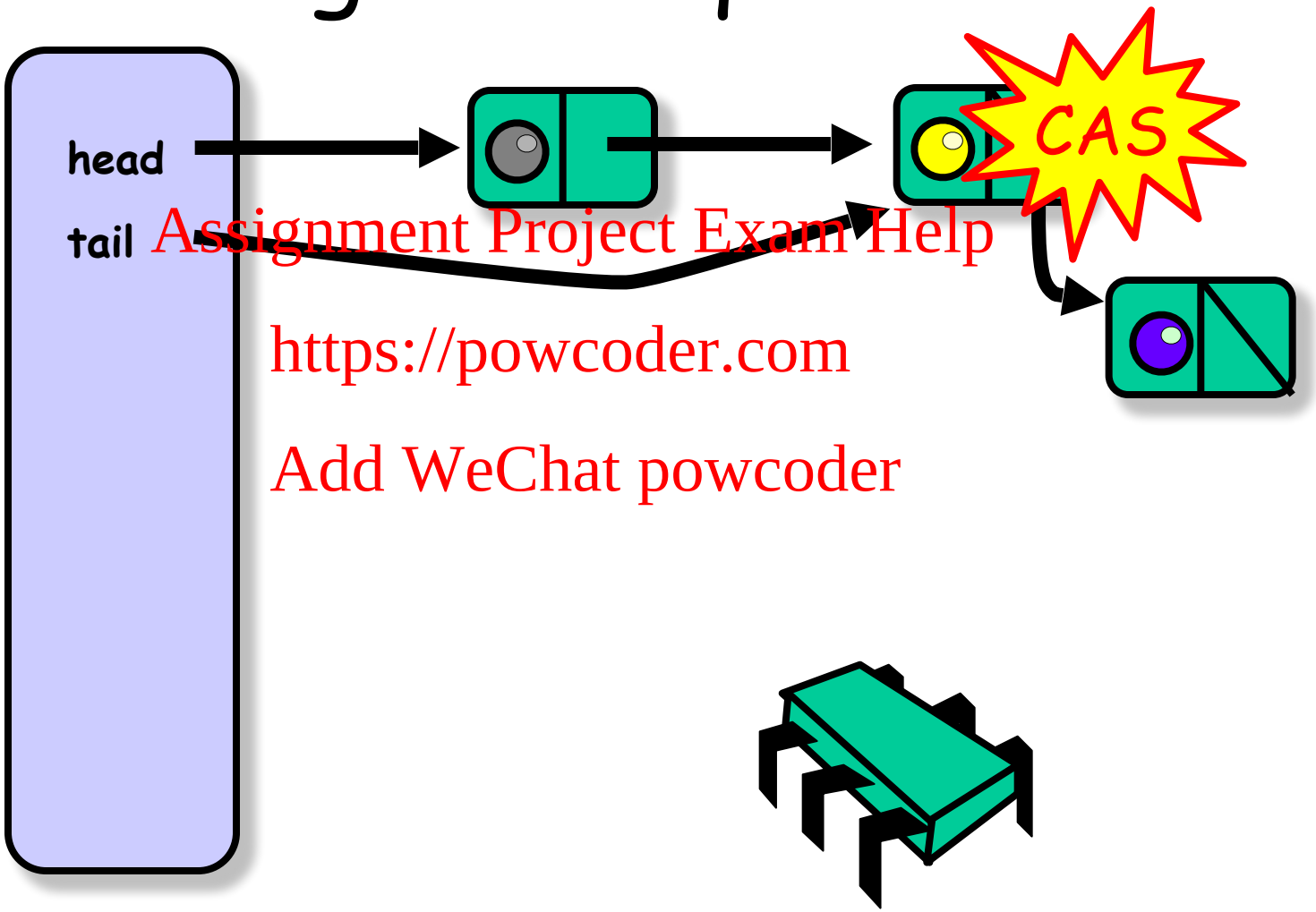
Enqueue



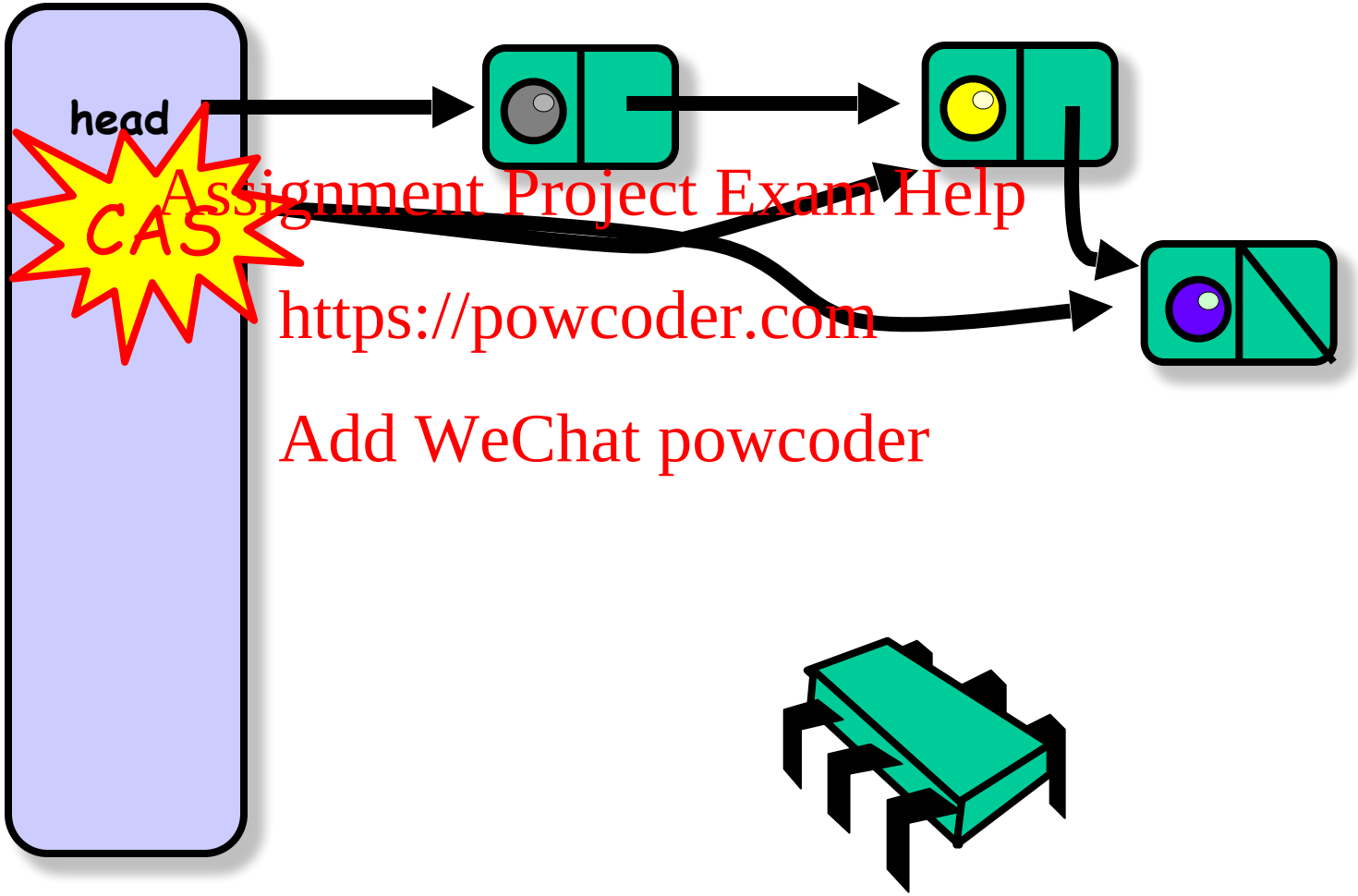
Enqueue



Logical Enqueue



Physical Enqueue



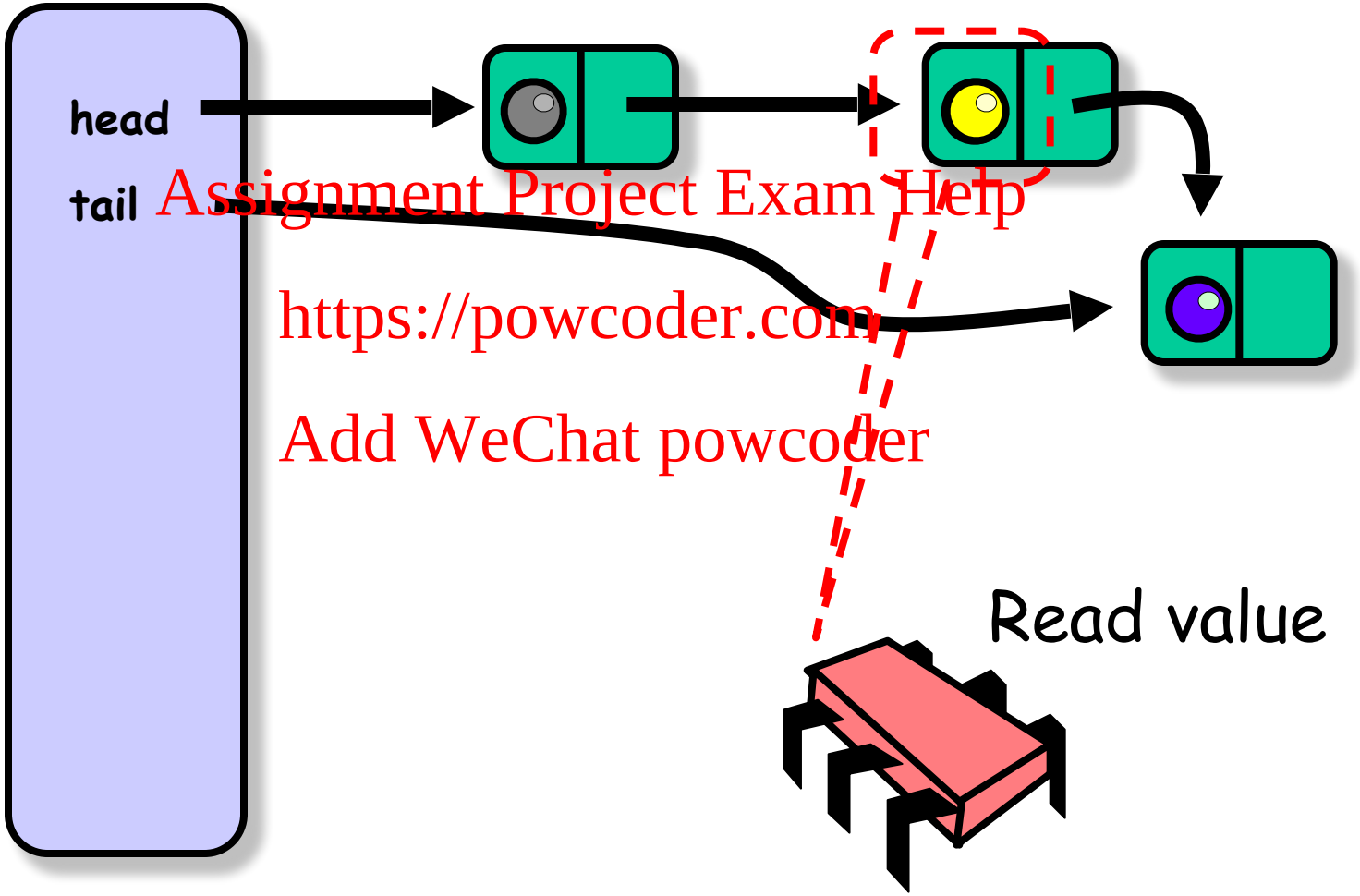
Enqueue

- These two steps are not atomic
- The tail field refers to either
  - Actual last Node (good)
  - Penultimate Node (not so good)
- Be prepared!
- (For you to think about) How could you fix that?

When CASs Fail

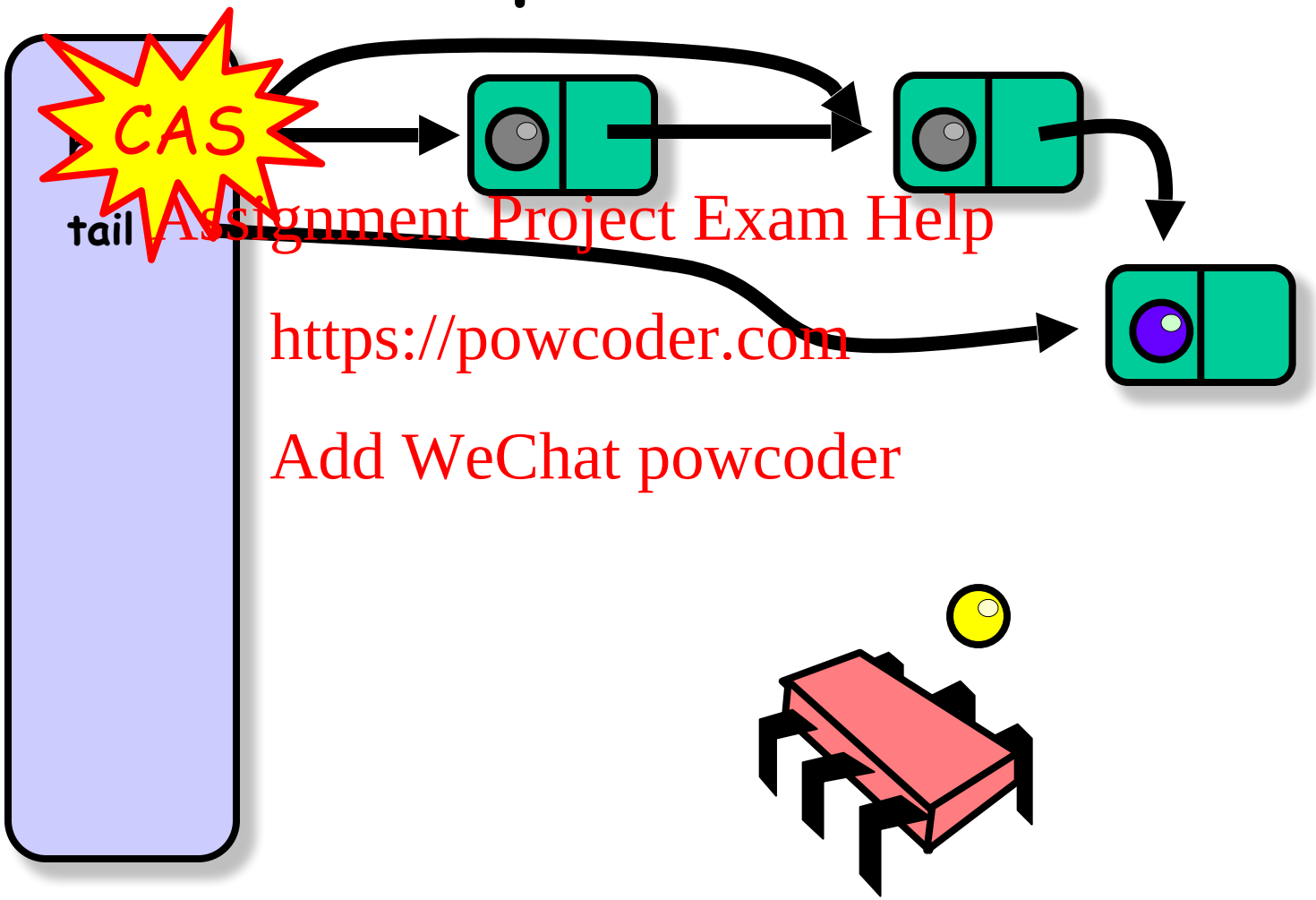
- During logical enqueue
  - Abandon hope, restart
  - Still lock-free (why?)
- During physical enqueue
  - Ignore it (why?)

Dequeuer



Dequeuer

Make first Node new sentinel

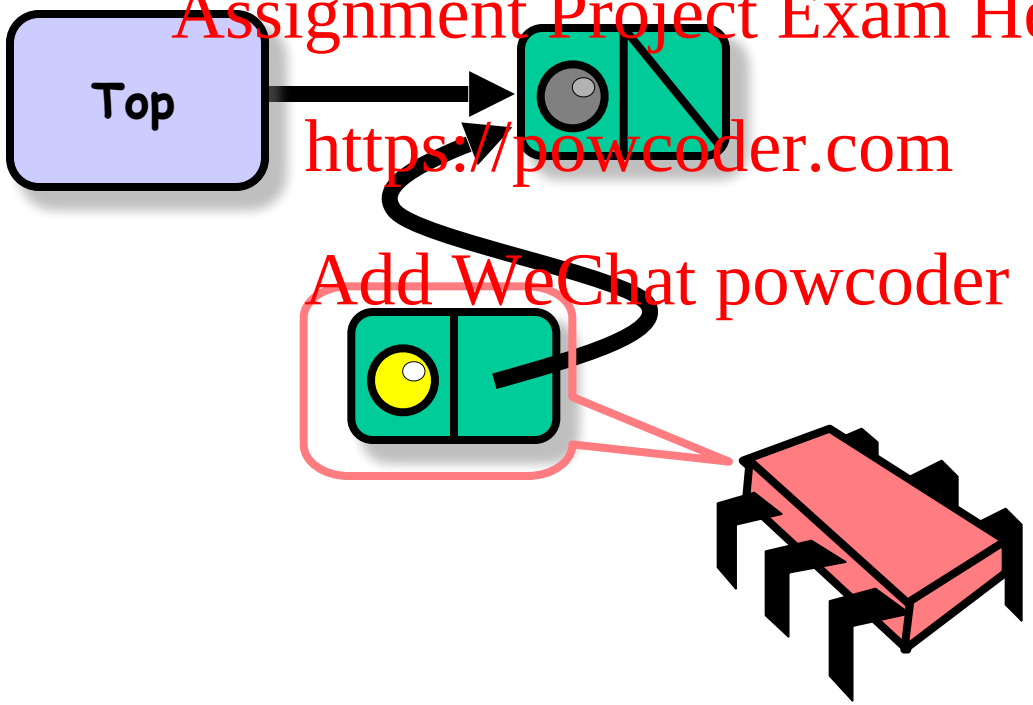


Concurrent Stack

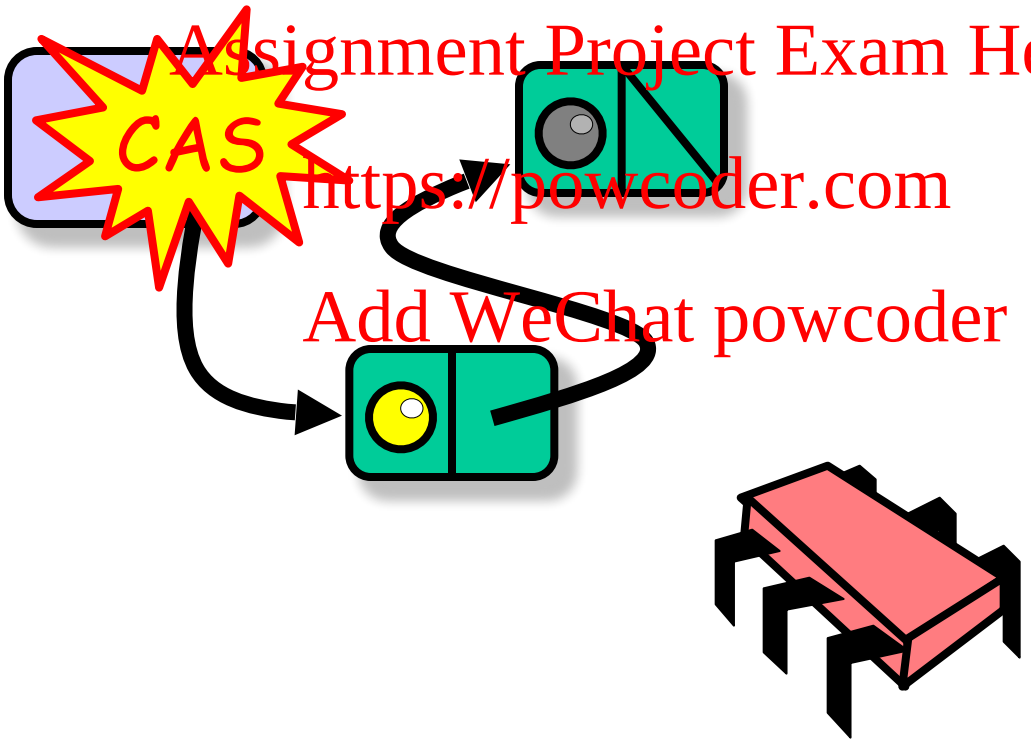
- Methods
  - push(x)
  - pop()
- Last-in, First-out (LIFO) order
- Lock-Free!

Empty Stack

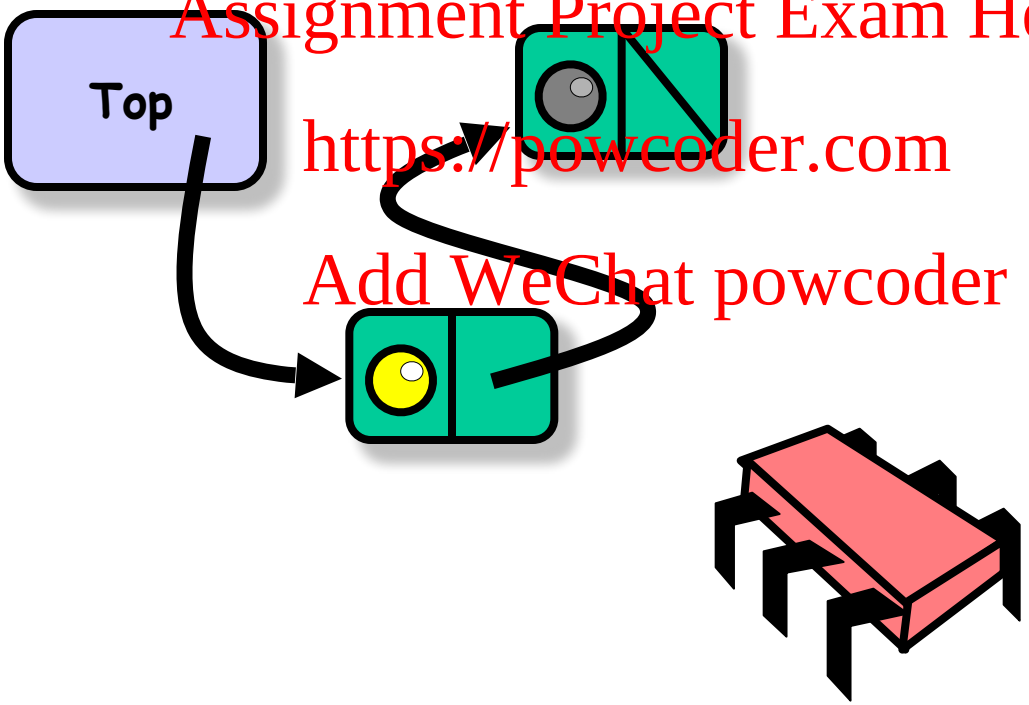
Push



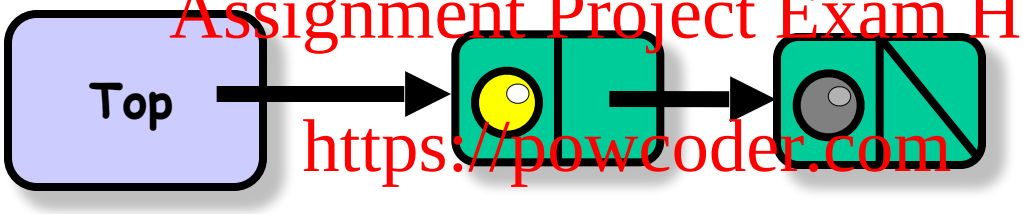
Push

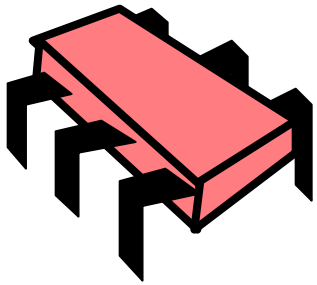


Push

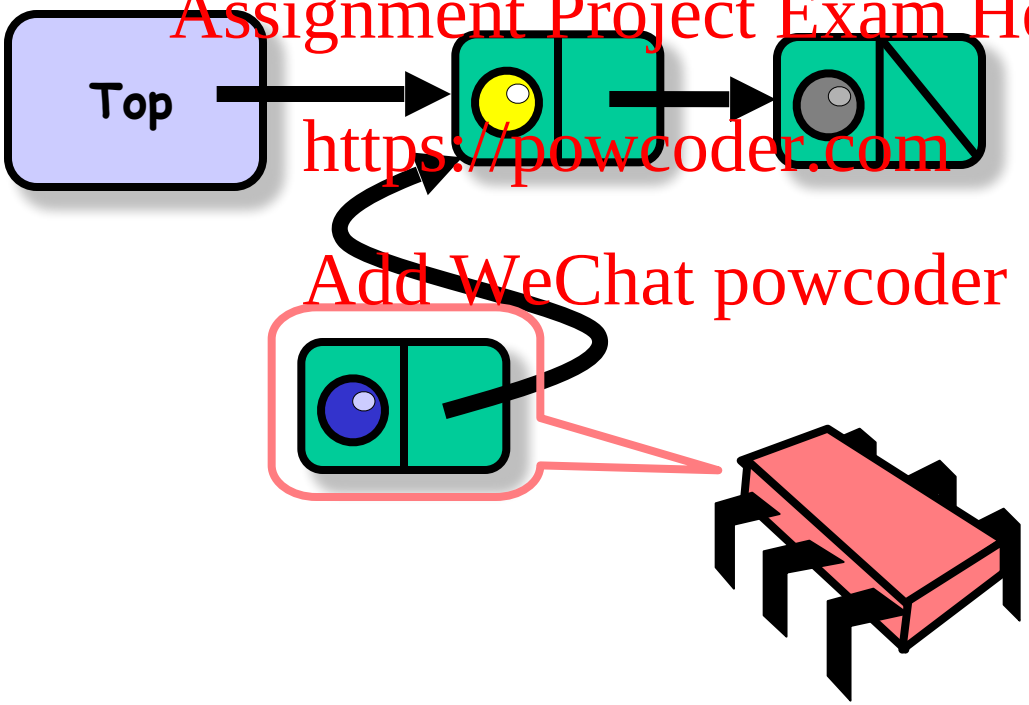


Push

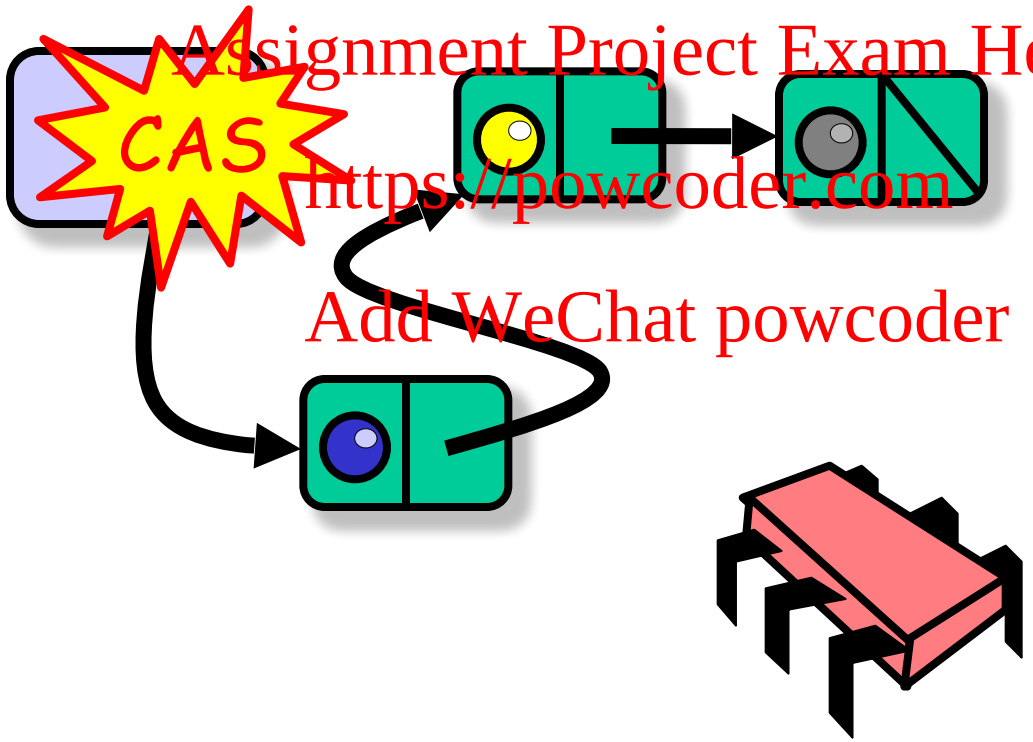




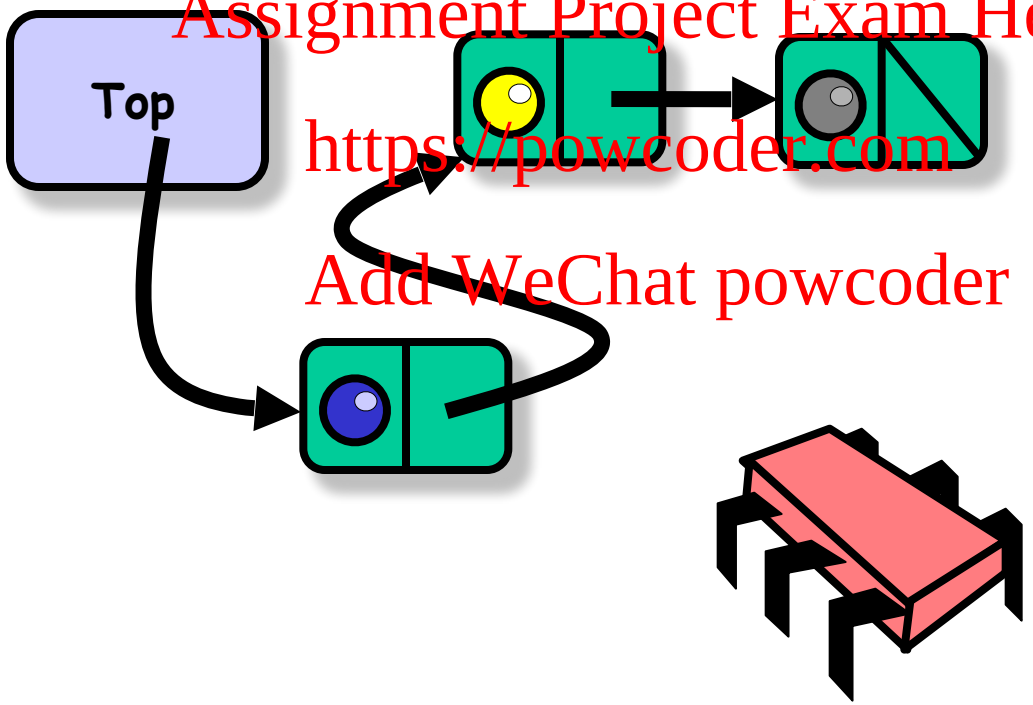
Push



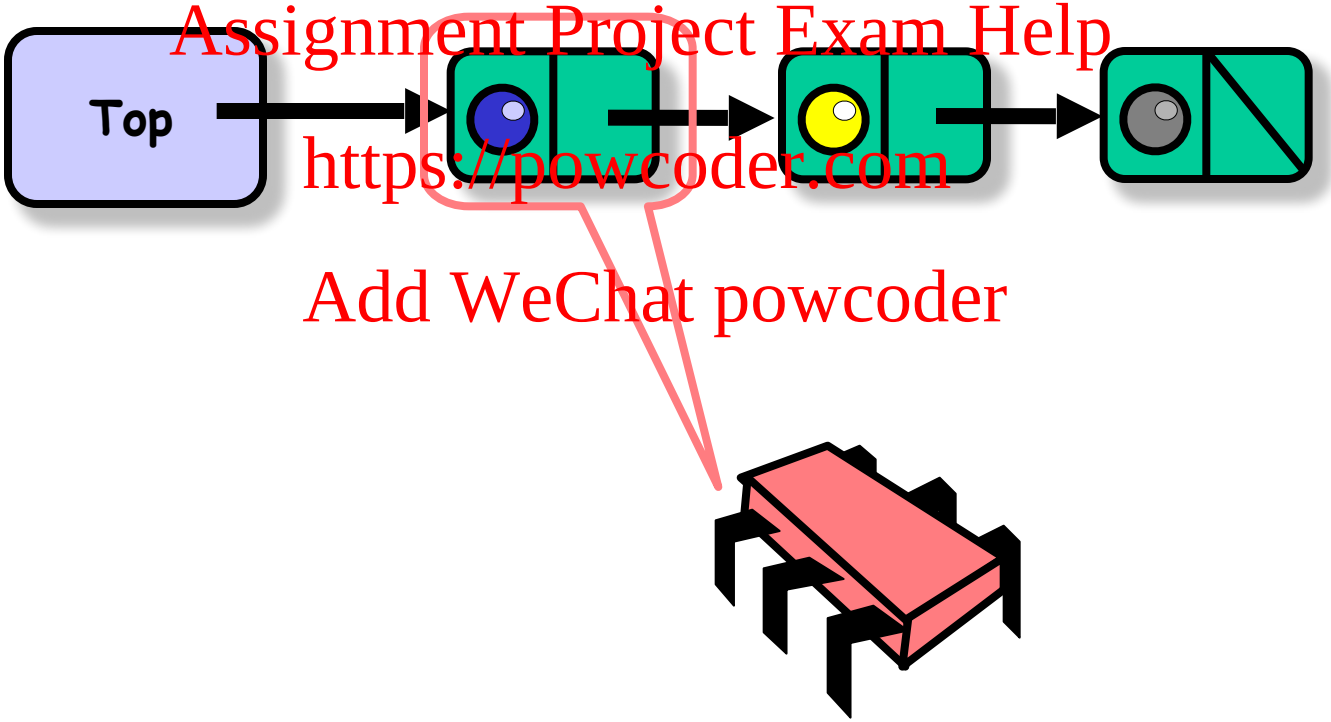
Push



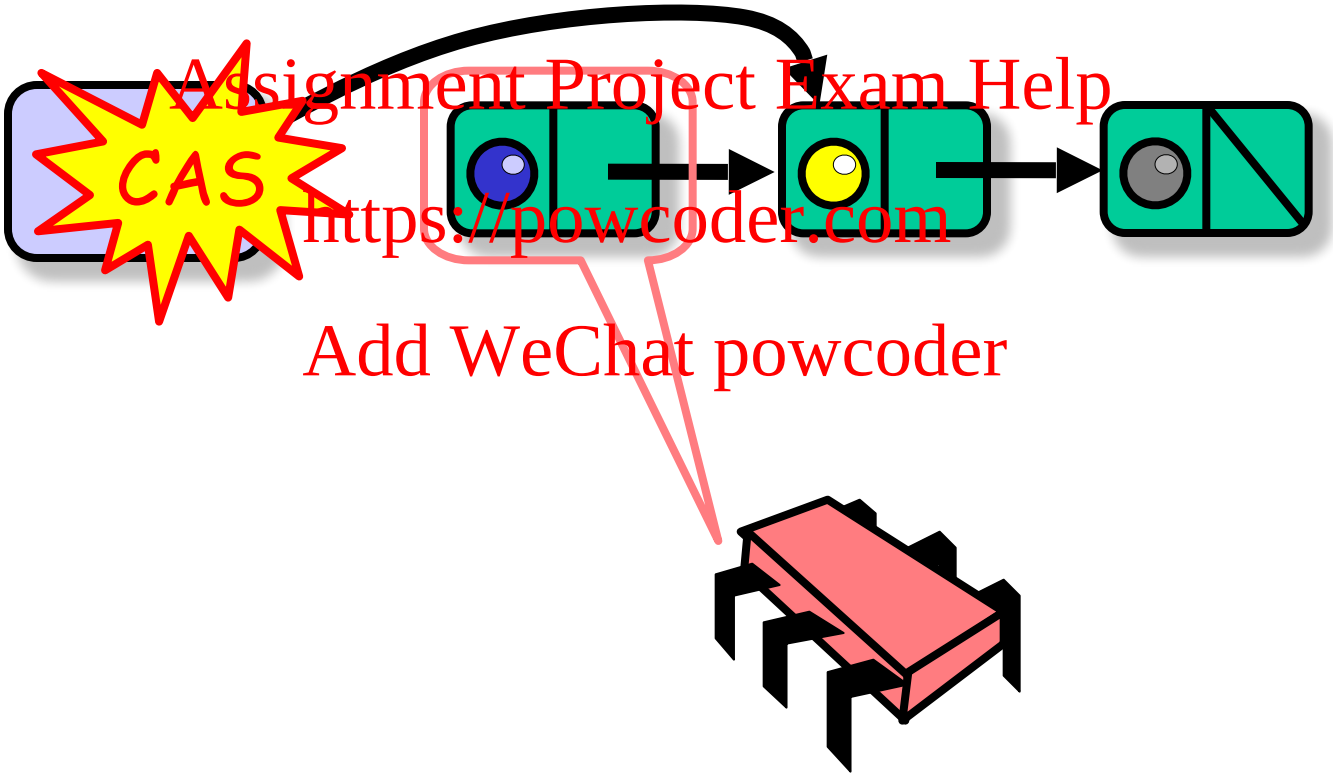
Push



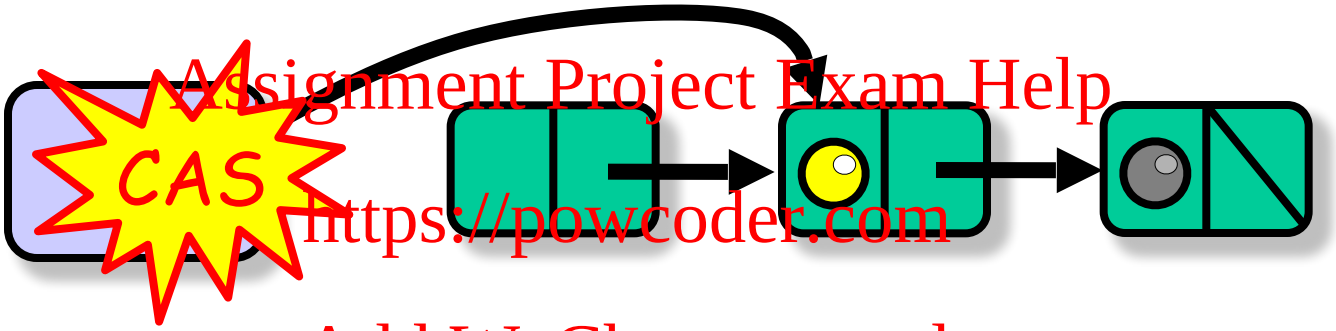
Pop

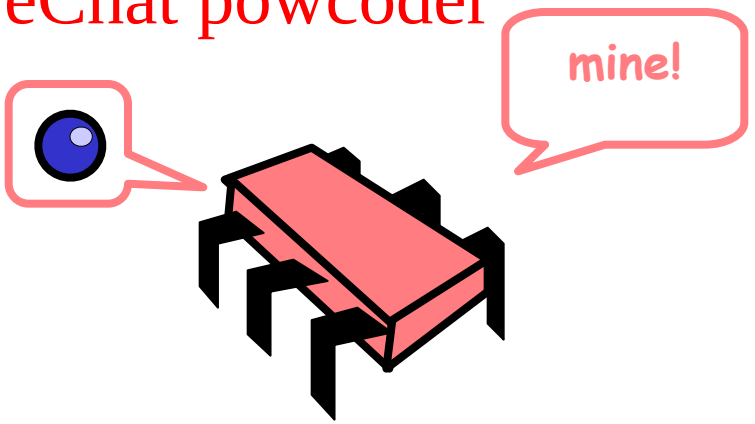


Pop

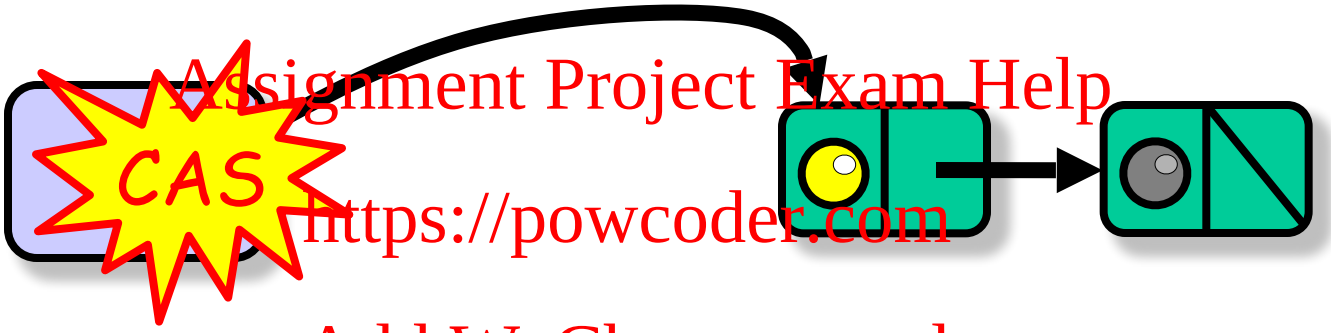


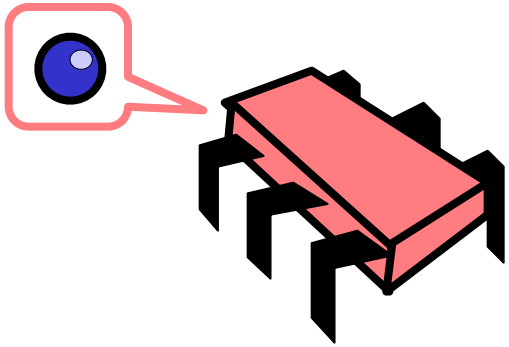
Pop



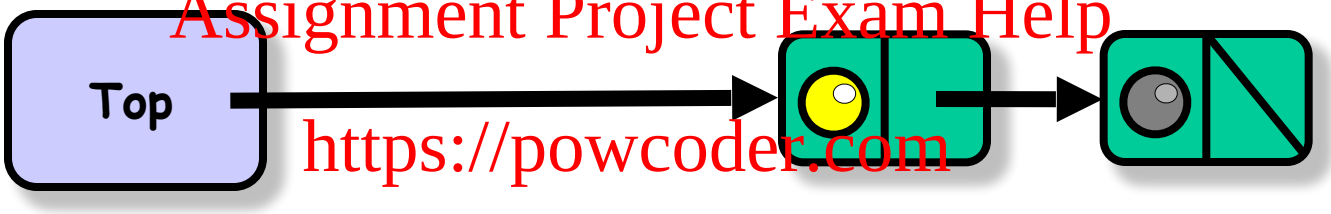


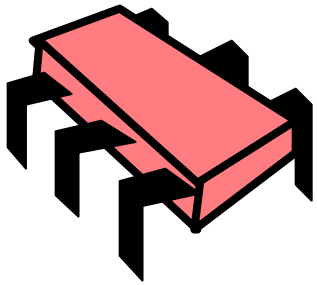
Pop





Pop





Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top =
    new AtomicReference(null);
  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();
  }}
```

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


tryPush attempts to push a node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


Read top value

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


current top will be new node's successor

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


Try to swing top, return success or failure

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


Push calls tryPush

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


Create new node

Lock-free Stack

```
public class LockFreeStack {
  private AtomicReference top = new
AtomicReference(null);
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()
}}
```


If tryPush() fails, back off before retrying

Unbounded Lock-Free Stack

```
protected 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( ); }
  }
}
```

```
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; }
}

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
  - Without GC, Yeah ABA
  - Without backoff, huge contention at top
  - In any case, no parallelism

Question

- Are stacks inherently sequential?
- Reasons why
  - Every **pop()** call fights for top item
- Reasons why not
  - Think about it!