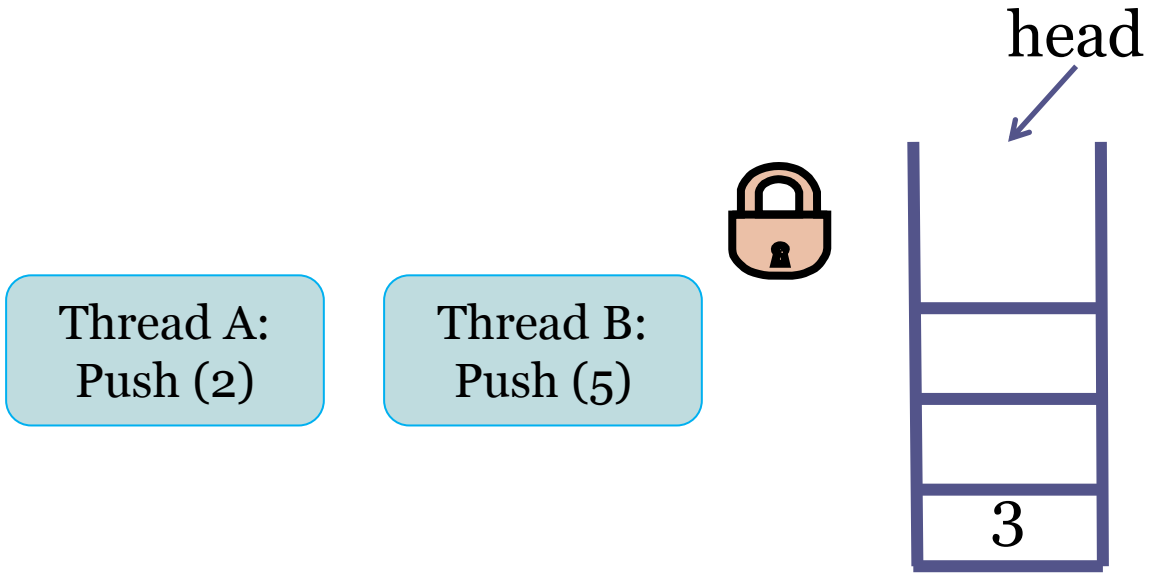
LCD: Local Combining on Demand

Dana Drachsler-Cohen and Erez Petrank
Technion , Israel

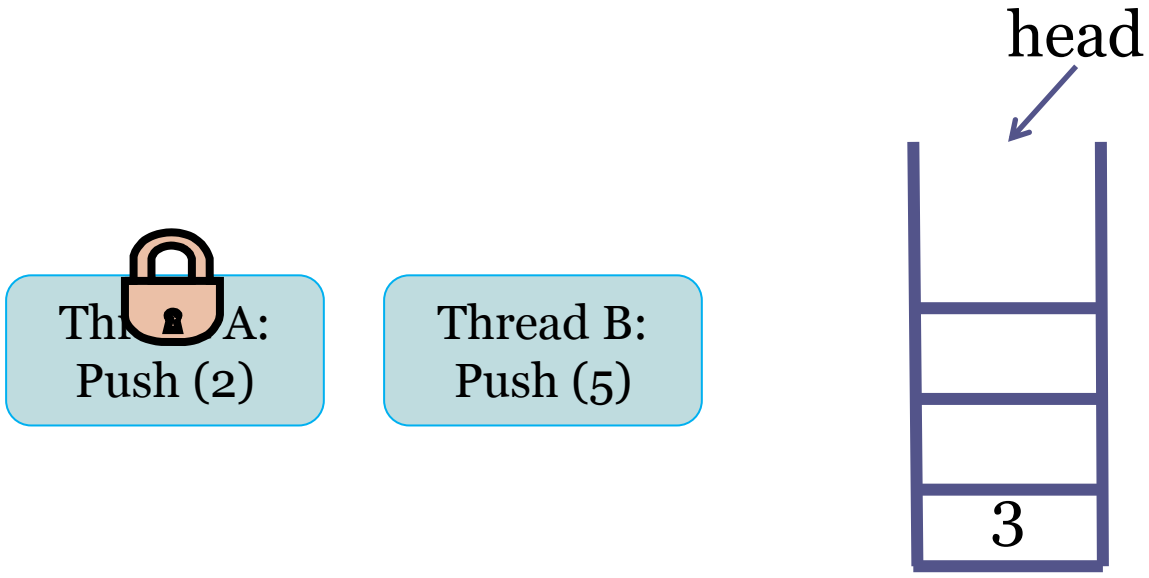
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



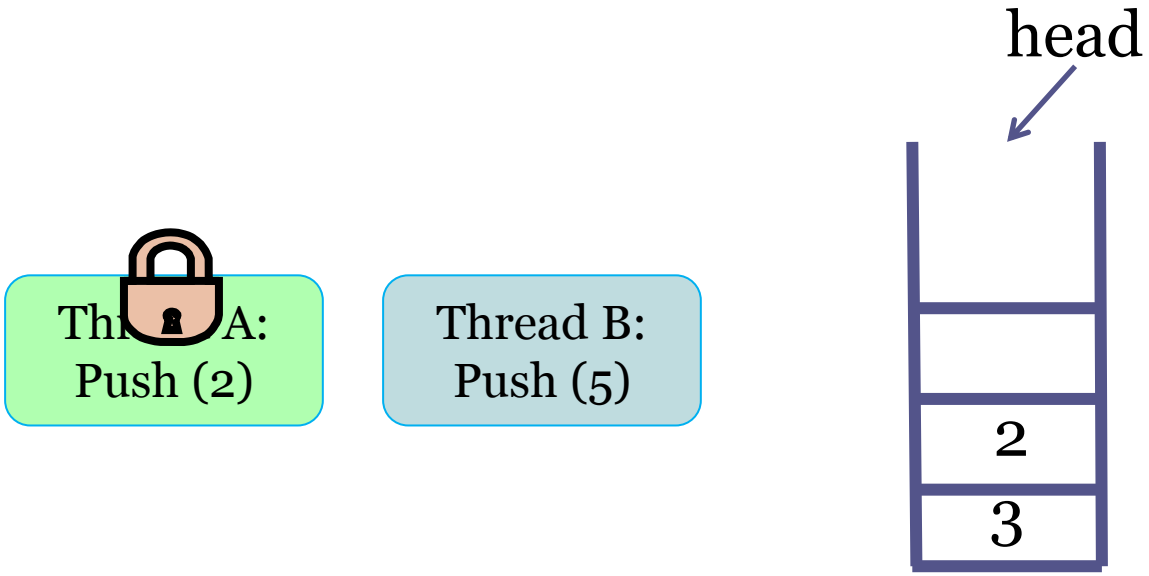
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



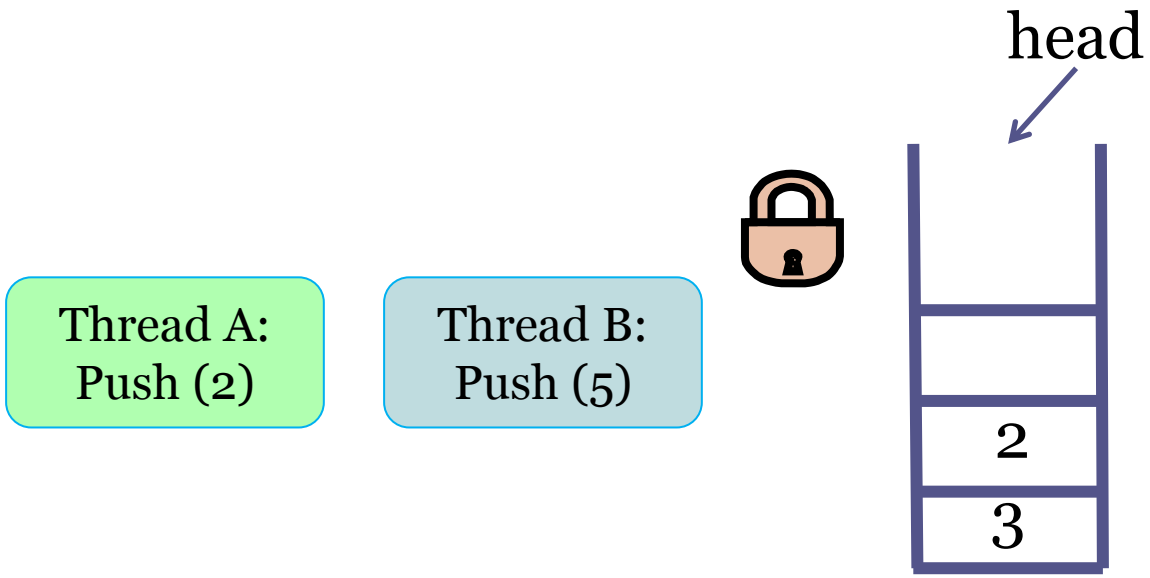
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



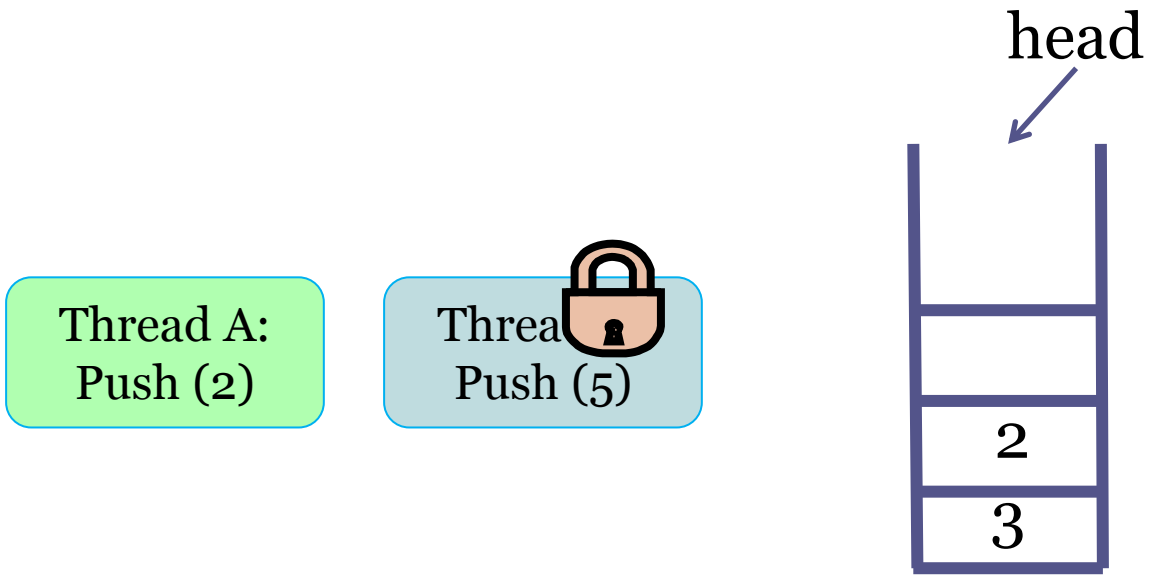
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



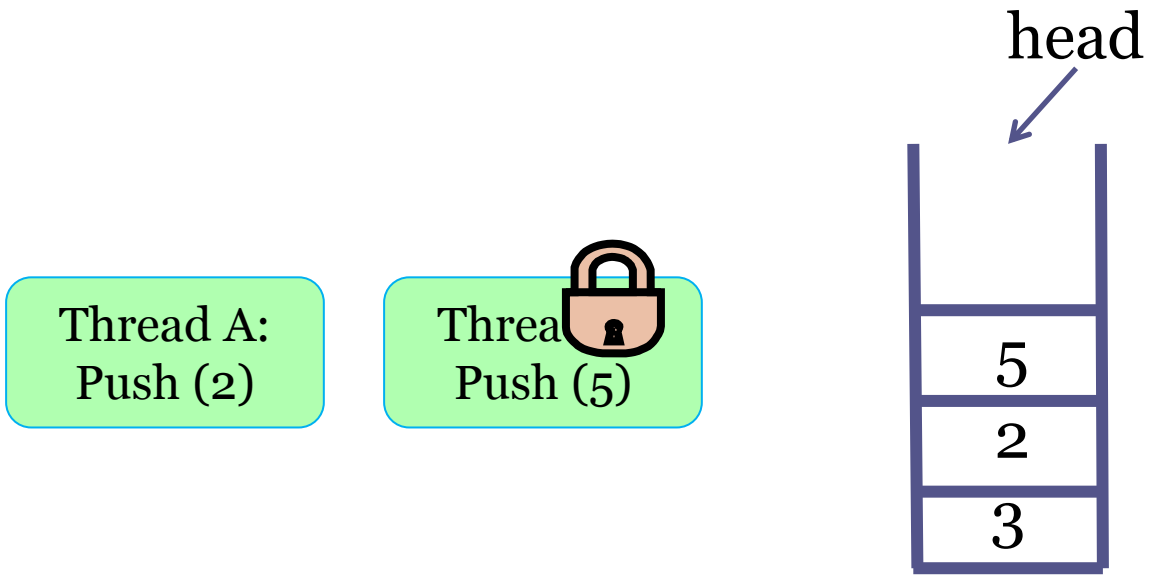
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



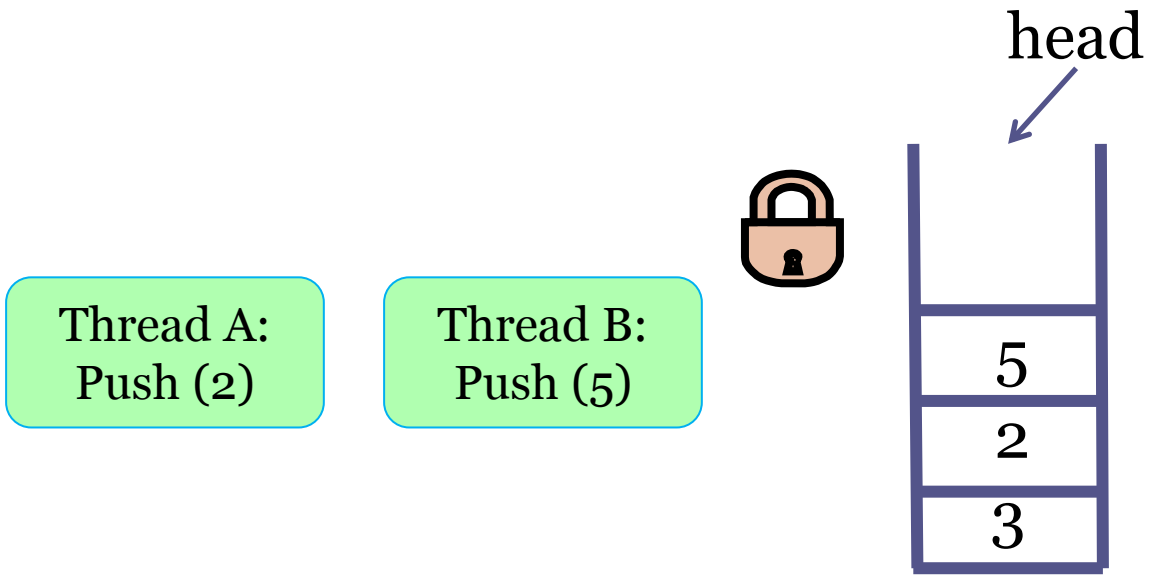
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



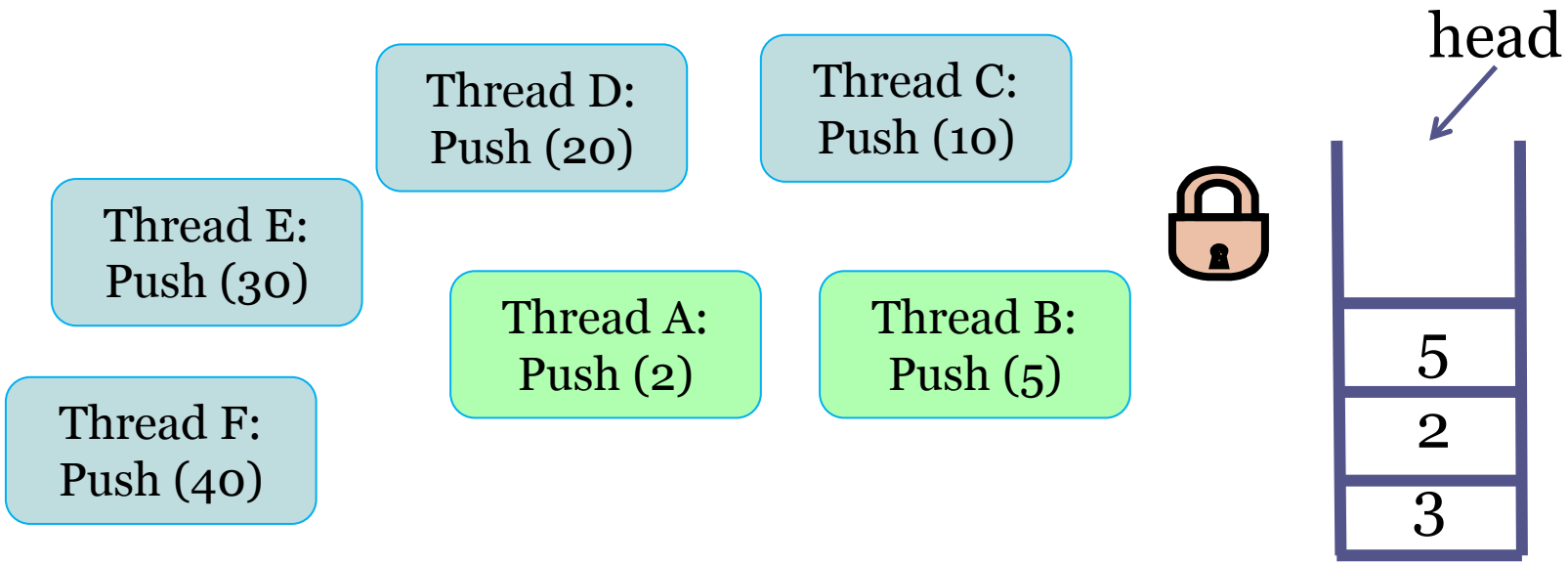
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



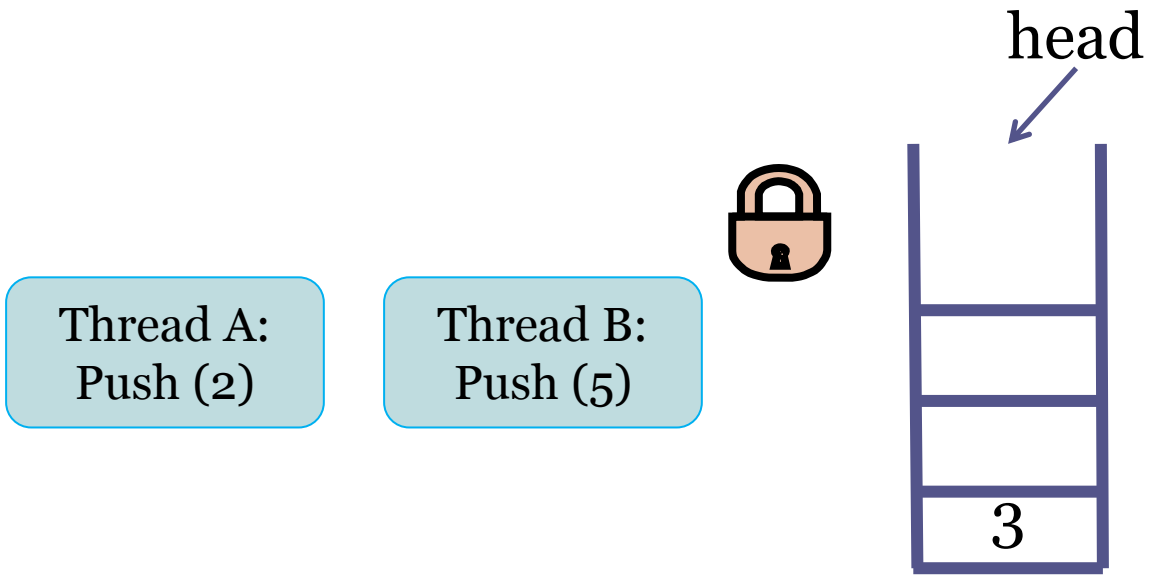
Combining Motivation

- Concurrent data structures are fundamental building blocks in various algorithms
- A common mean of synchronization: locks



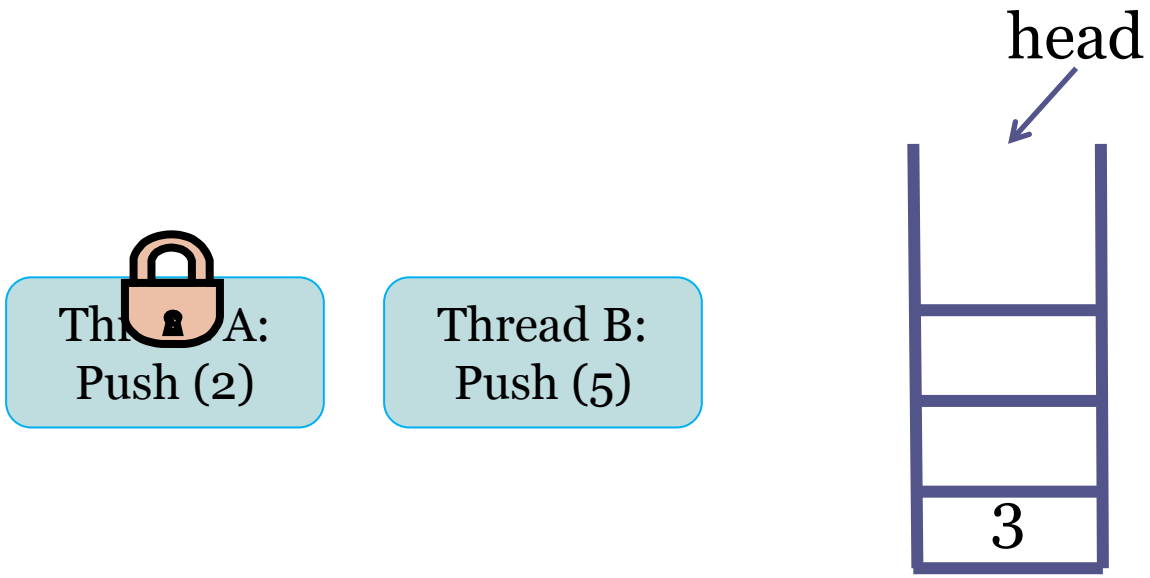
Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads



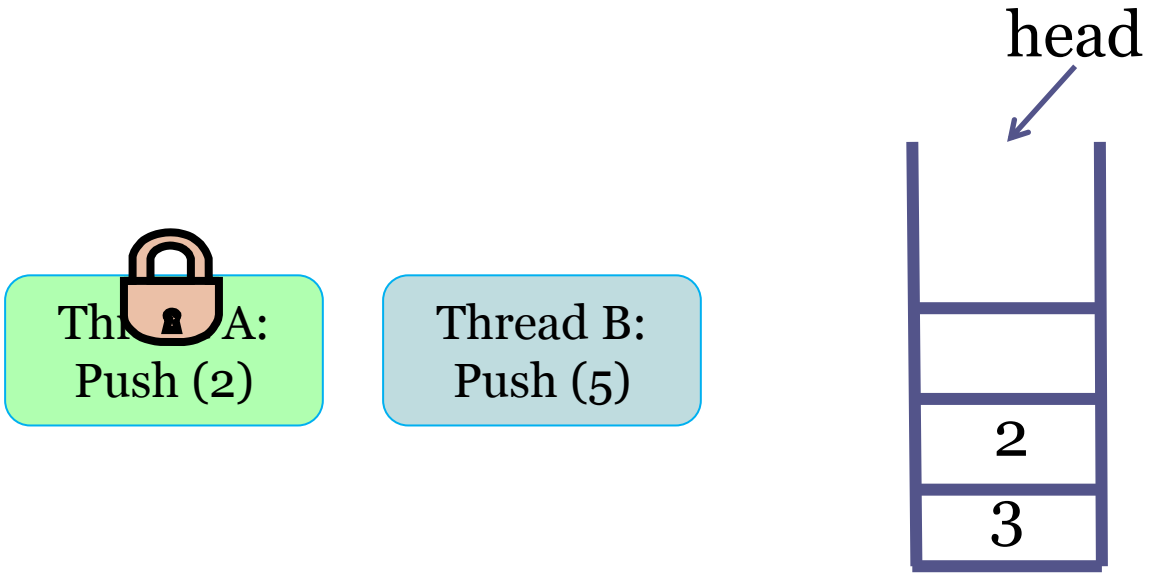
Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads



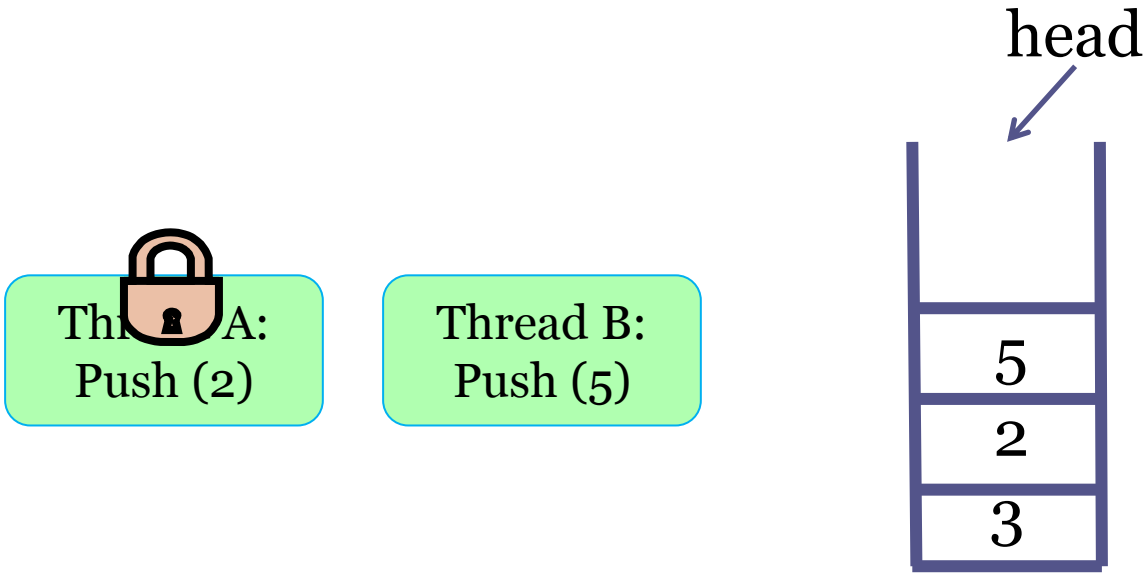
Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads



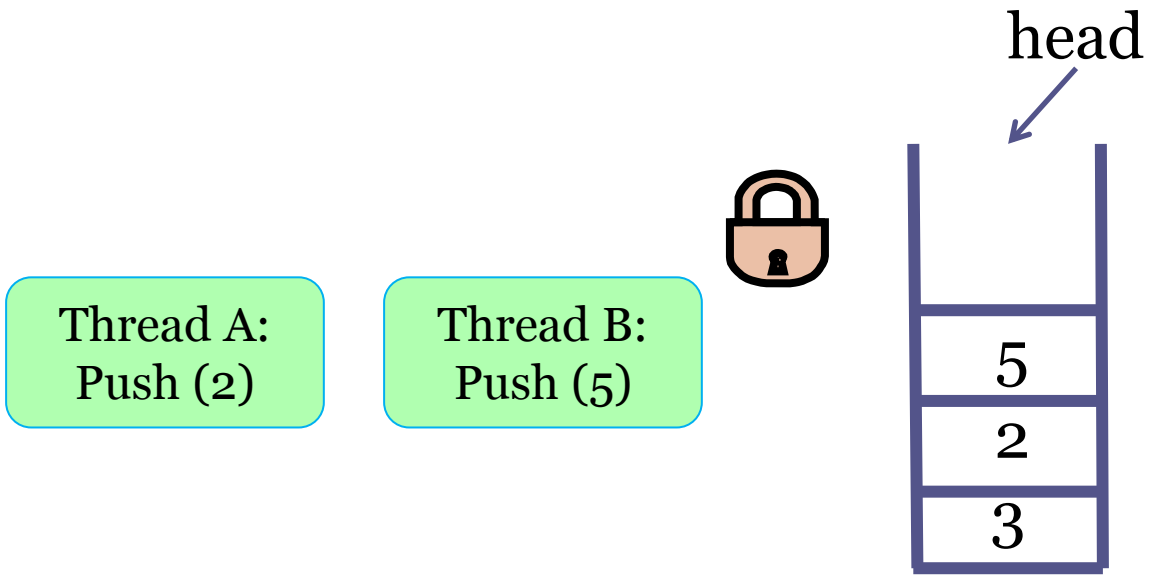
Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads



Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads



Combining Motivation

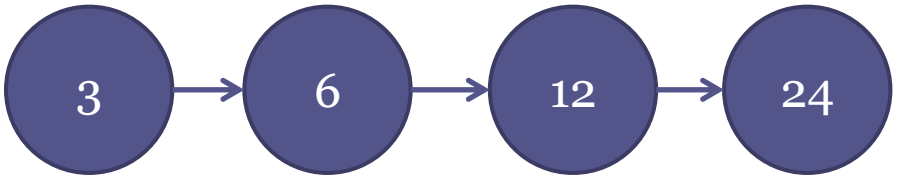
- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads
  - Was shown to improve performance of data structures with 1-2 contention points
    - [Hendler, Incze, Shavit, Tzafrir], [Fatourou, Kallimanis], [Dice, Marathe, Shavit]
- Namely, they were already protected by 1-2 locks
  - The global lock did not reduce scalability

Combining Motivation

- In combining, threads help each other
- Common approach: a thread acquires a global lock and executes operations of contending threads
  - Was shown to improve performance of data structures with 1-2 contention points
    - [Hendler, Incze, Shavit, Tzafrir], [Fatourou, Kallimanis], [Dice, Marathe, Shavit]
- J

What about data structures with unbounded number of contention points (e.g., the linked-list)?

Linked-List

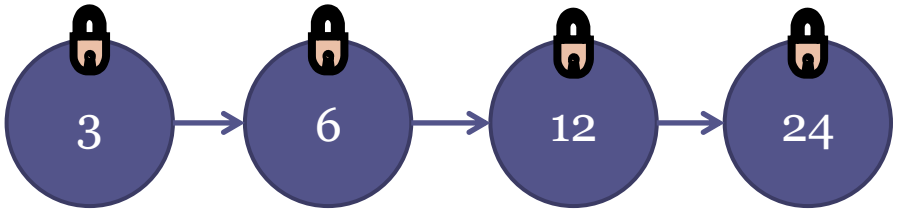


- Consists of nodes each with a unique key
- Sorted by the keys
  - Each node points to its successor in the list
- Supports three operations
  - Insert(k): insert k if it is not yet present
  - Remove(k): remove k if present
  - Contains(k): return true if k is present
- A simple and practical concurrent implementation
  - The Lazy List

The Lazy List

[Heller, Herlihy, Luchangco, Moir, Scherer III, Shavit]

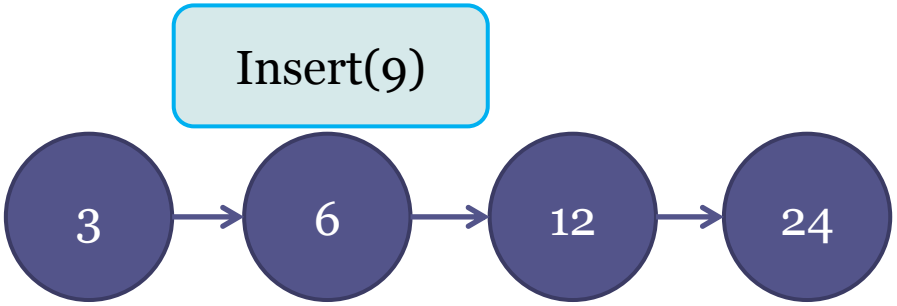
- Lock-based – each node has a lock
  - Insert(k): 1 lock
  - Remove(k): 2 locks
  - Contains(k): no locks



- We focus on combining of locked-based operations
  - Thus, combining only affects insert and remove

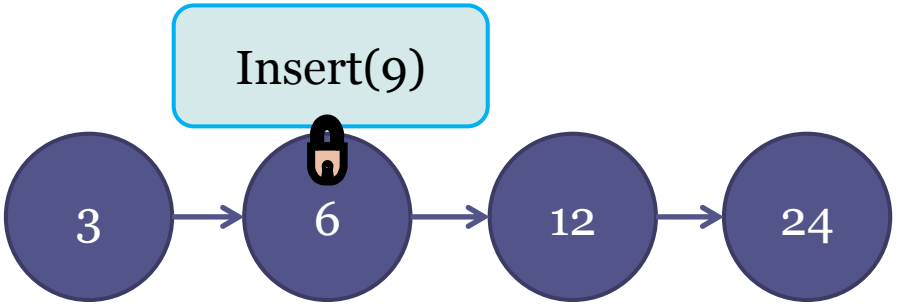
The Lazy List Insert(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$



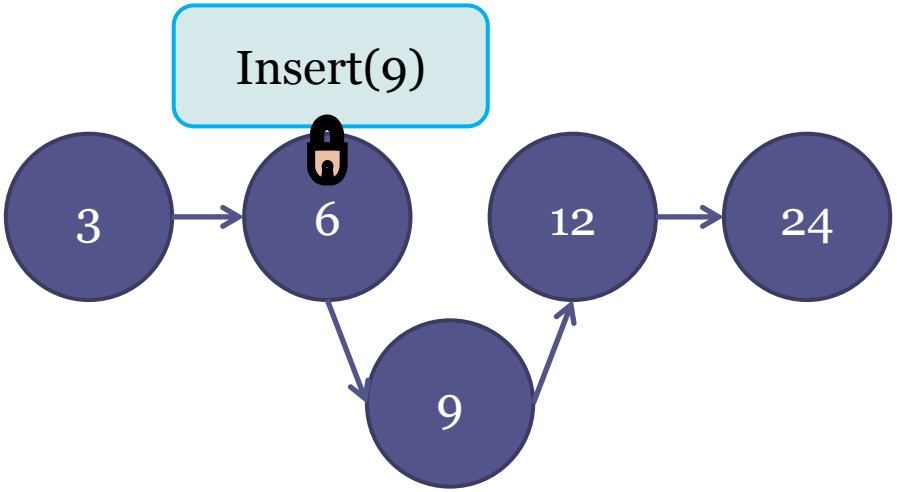
The Lazy List Insert(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p



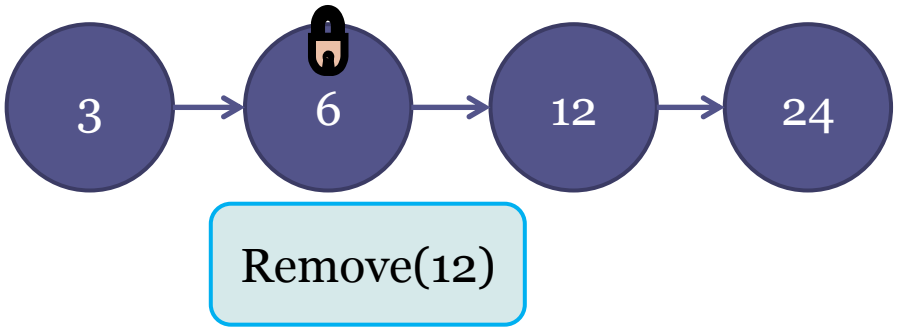
The Lazy List Insert(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p
- If the key of p's successor is k: cannot insert
- Otherwise, add k after p



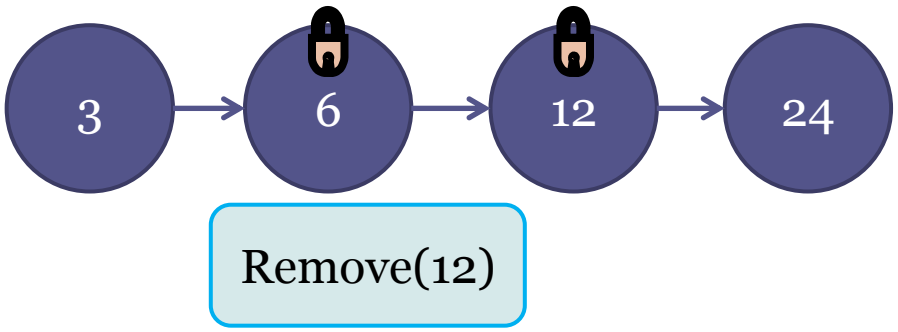
The Lazy List Remove(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p



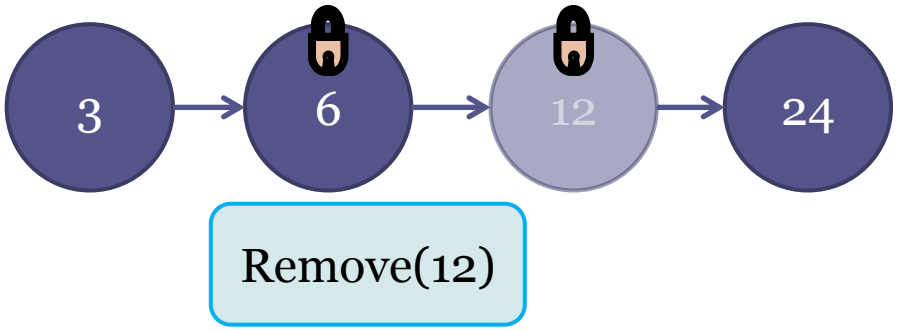
The Lazy List Remove(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p
- If the key of p's successor is not k: cannot remove
- Otherwise
  - Lock p's successor



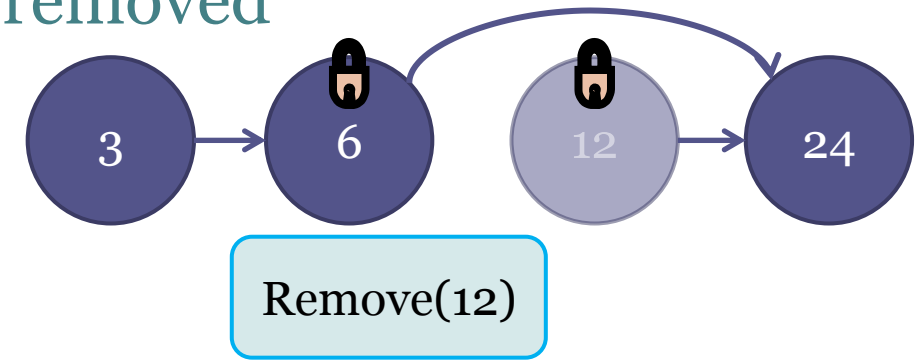
The Lazy List Remove(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p
- If the key of p's successor is not k: cannot remove
- Otherwise
  - Lock p's successor
  - Mark the successor as removed

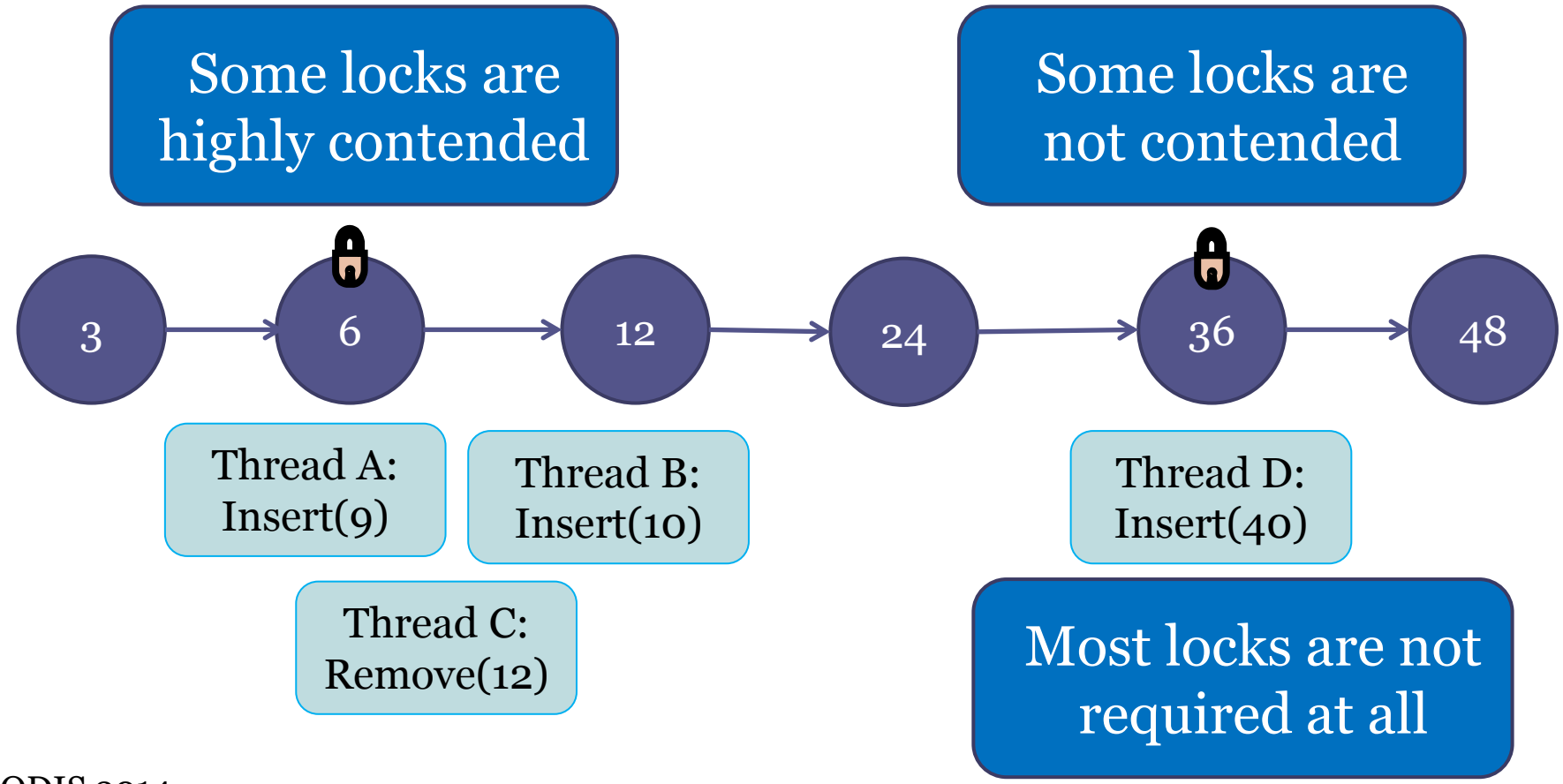


The Lazy List Remove(k)

- Search for a node p s.t.: $p.key < k \leq next(p).key$
- Lock p
- If the key of p's successor is not k: cannot remove
- Otherwise
  - Lock p's successor
  - Mark the successor as removed
  - Update p's successor



Contention in Linked-Lists



Our Contribution: LCD: Local Combining on Demand

- We present a new combining technique for such data structures which is *Local* and *On demand*
  - *Local* – combine for each lock independently
    - No global lock
  - *On demand* – do not introduce overhead if there is no contention
- We demonstrate it on linked-lists
  - The *LCD-list*
    - An extension of the Lazy List

Key Ideas of LCD

- What are the combining types?
- How does on-demand combining work?
- How does the local combining work?
- How to handle operations requiring multiple locks?
- How to integrate LCD with the Java lock?

What are the combining types?

Combining Types

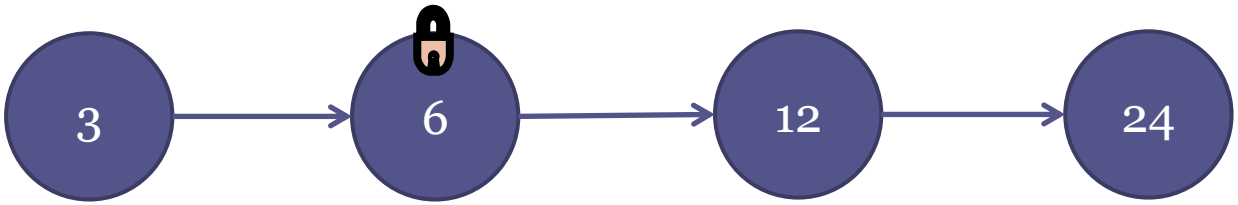
1. Execute other thread operations

Thread A:
Insert(7)

Thread B:
Insert(8)

Thread C:
Insert(9)

Thread D:
Remove(12)



Combining Types

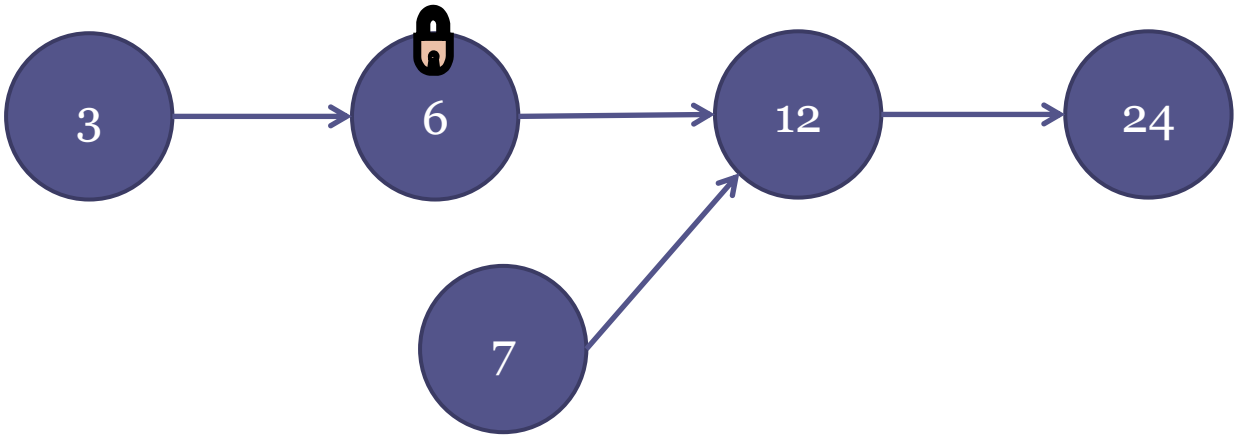
1. Execute other thread operations

Thread A:
Insert(7)

Thread B:
Insert(8)

Thread C:
Insert(9)

Thread D:
Remove(12)



Combining Types

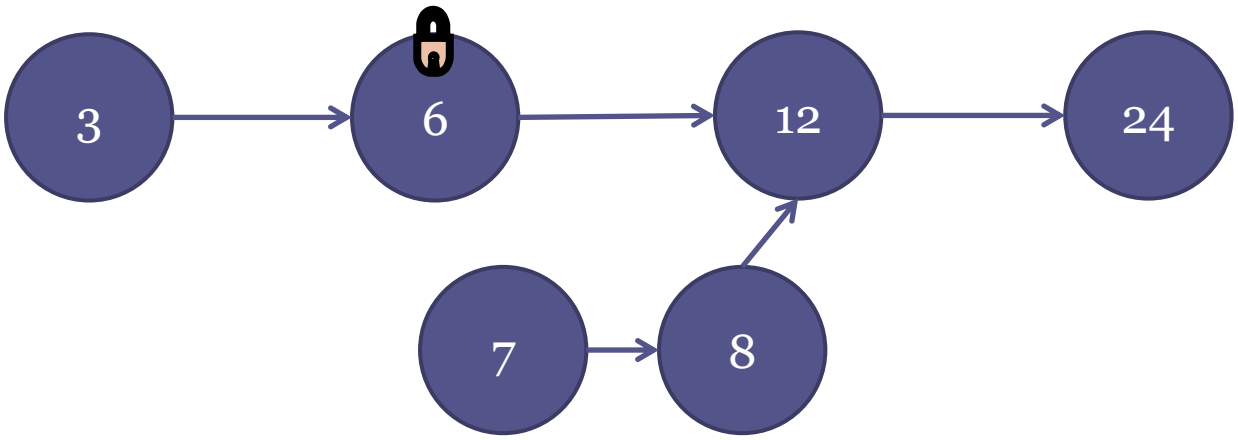
1. Execute other thread operations

Thread A: Insert(7)

Thread B: Insert(8)

Thread C: Insert(9)

Thread D: Remove(12)



Combining Types

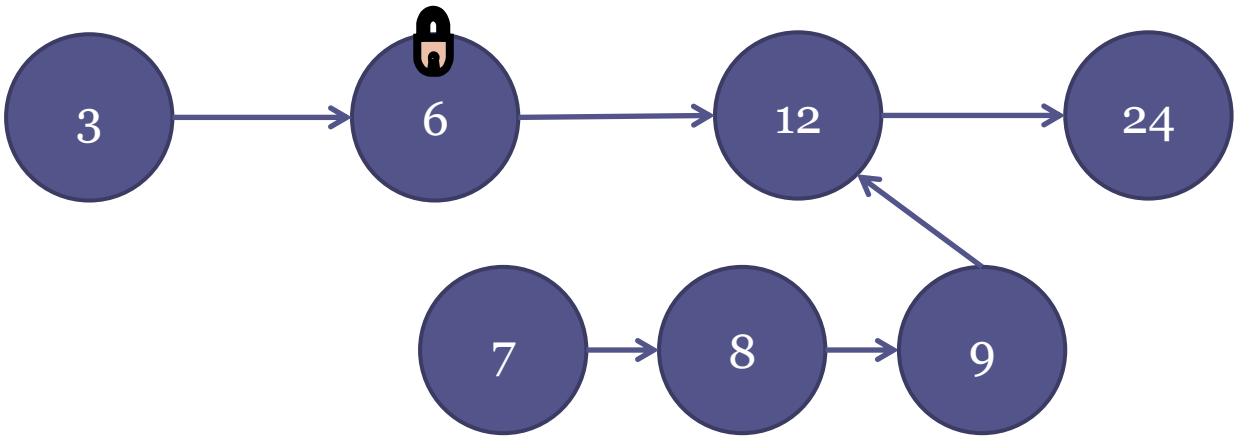
1. Execute other thread operations

Thread A:
Insert(7)

Thread B:
Insert(8)

Thread C:
Insert(9)

Thread D:
Remove(12)



Combining Types

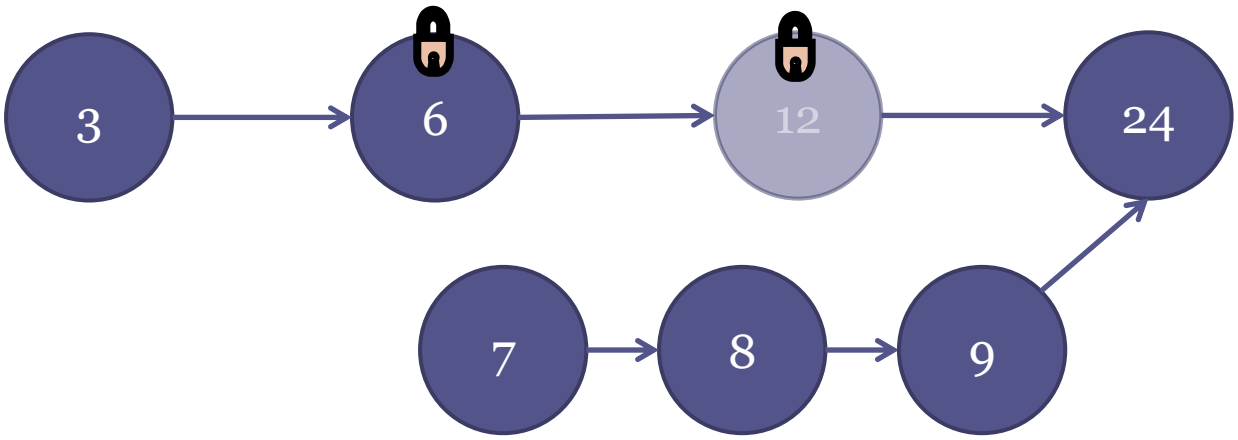
1. Execute other thread operations

Thread A: Insert(7)

Thread B: Insert(8)

Thread C: Insert(9)

Thread D: Remove(12)



Combining Types

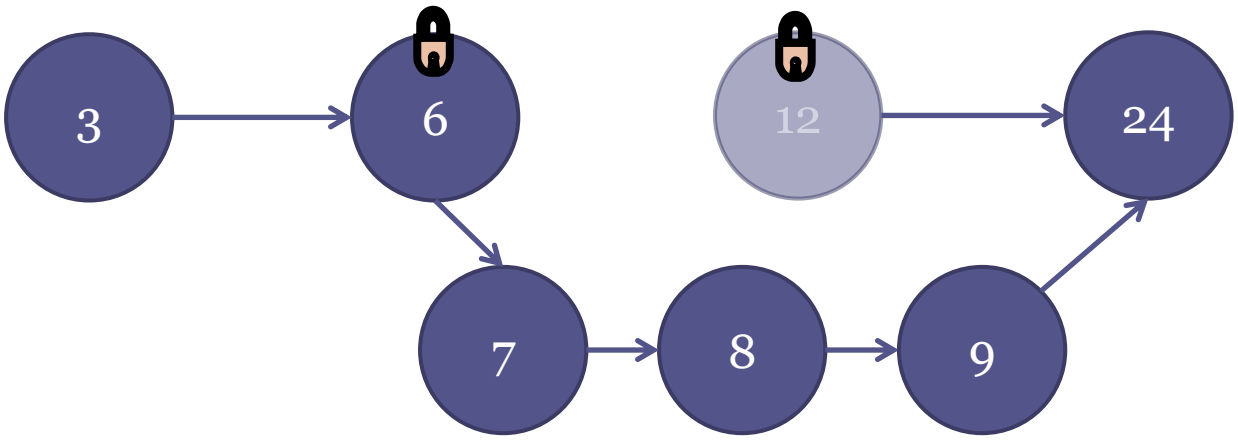
1. Execute other thread operations

Thread A: Insert(7)

Thread B: Insert(8)

Thread C: Insert(9)

Thread D: Remove(12)



Combining Types

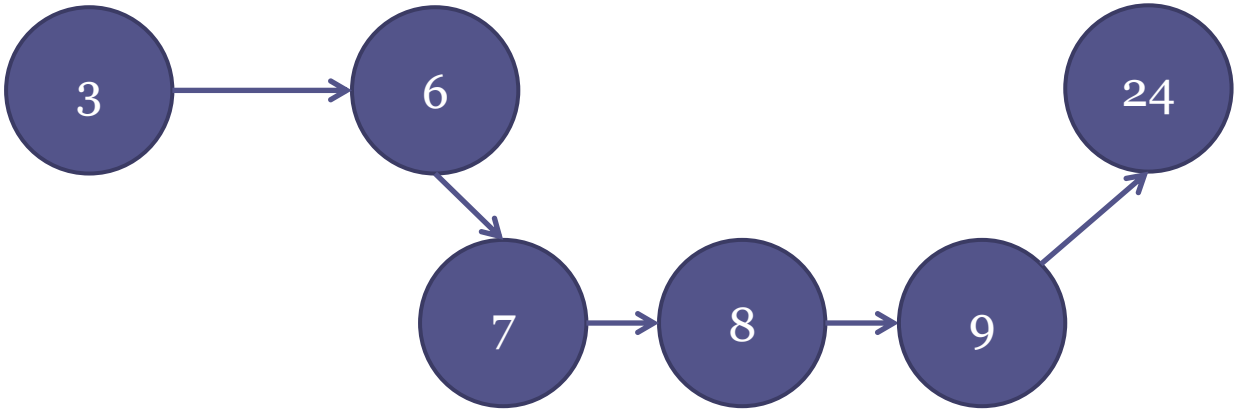
1. Execute other thread operations

Thread A:
Insert(7)

Thread B:
Insert(8)

Thread C:
Insert(9)

Thread D:
Remove(12)

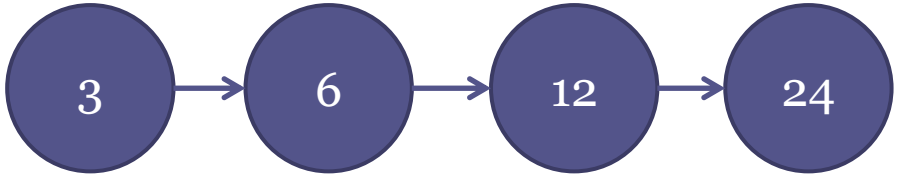


Combining Types

2. Complementary operation elimination

Thread A:
Insert(9)

Thread B:
Remove(9)

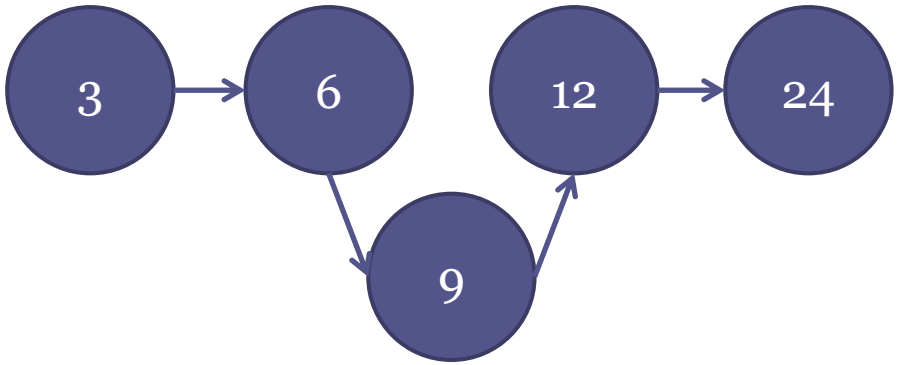


Combining Types

2. Complementary operation elimination

Thread A:
Insert(9)

Thread B:
Remove(9)

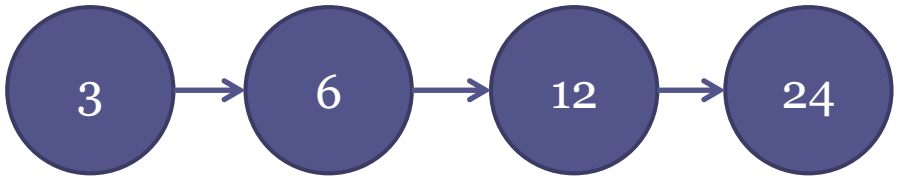


Combining Types

2. Complementary operation elimination

Thread A:
Insert(9)

Thread B:
Remove(9)

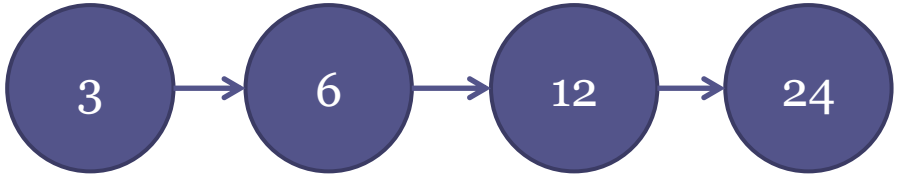


Combining Types

2. Complementary operation elimination

Thread A:
Insert(9)

Thread B:
Remove(9)

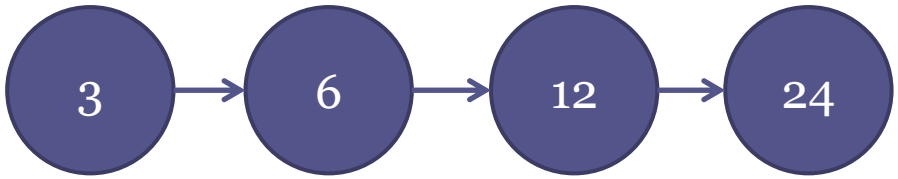


Combining Types

2. Complementary operation elimination

Thread A:
Insert(9)

Thread B:
Remove(9)



Combining Types

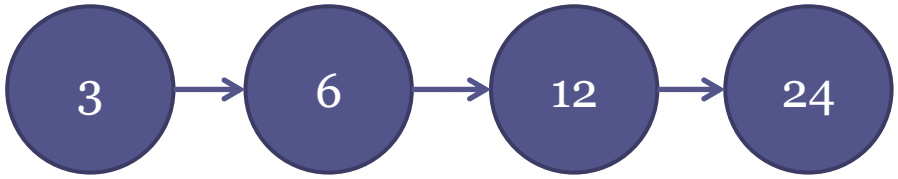
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

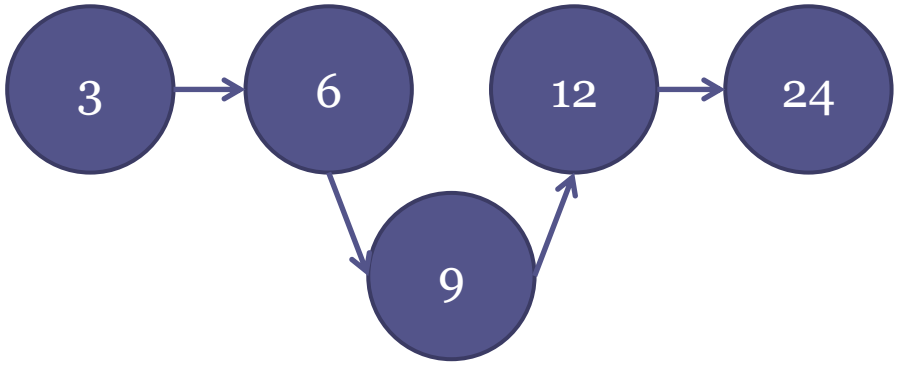
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

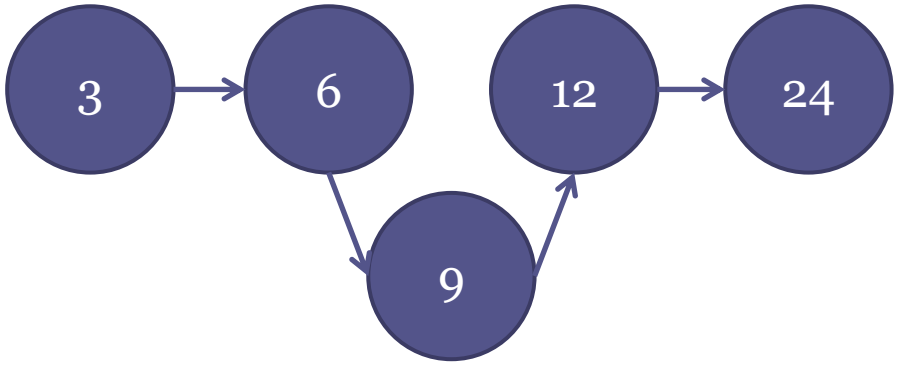
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

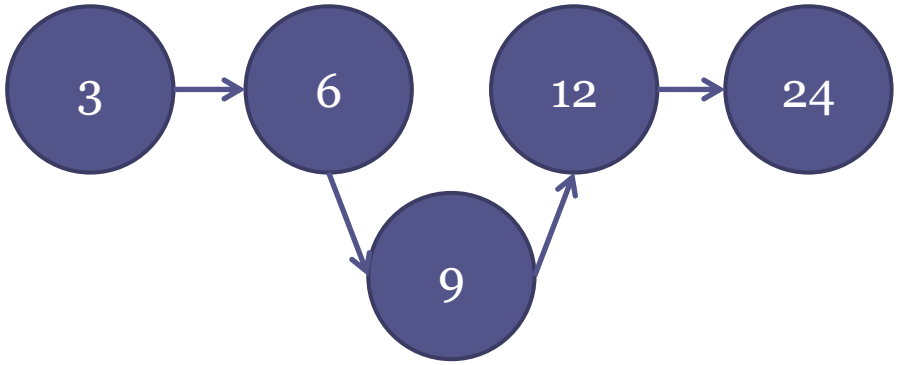
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

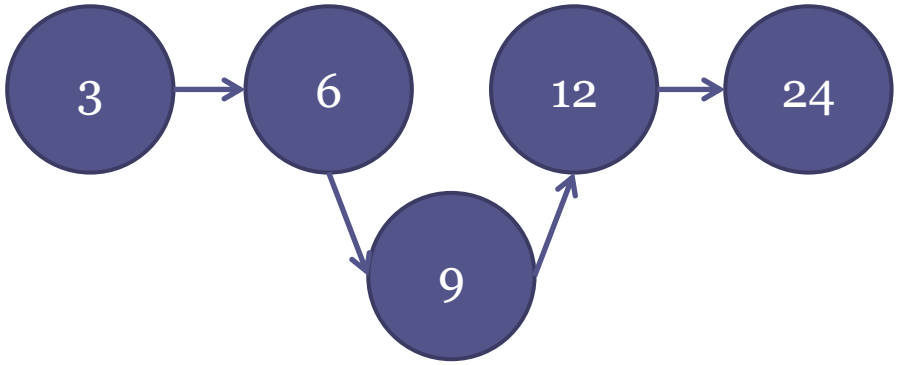
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

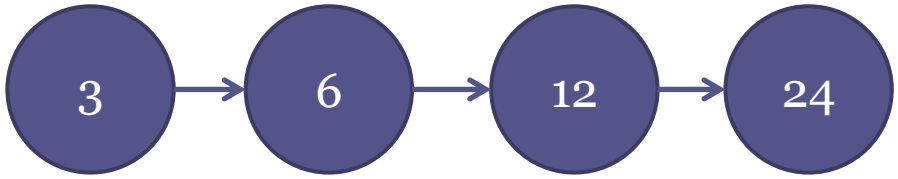
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

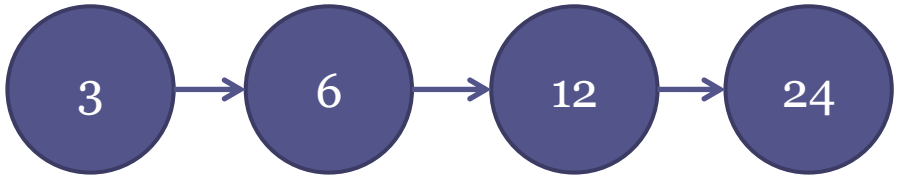
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

Thread D: Insert(9)



Combining Types

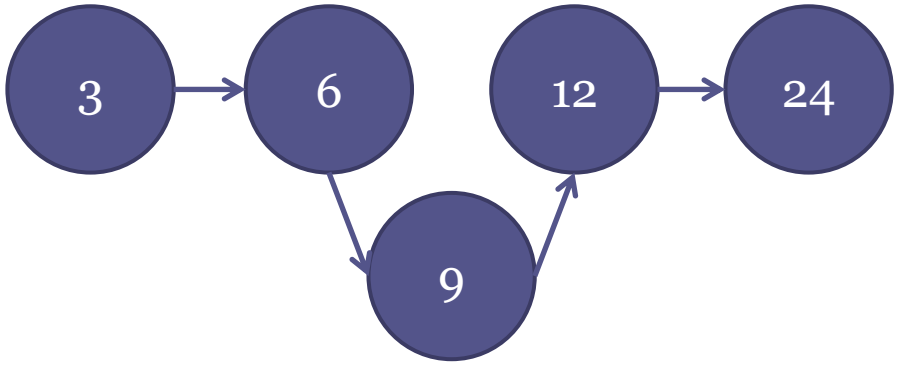
3. Identical operation elimination

Thread A: Insert(9)

Thread B: Insert(9)

Thread C: Insert(9)

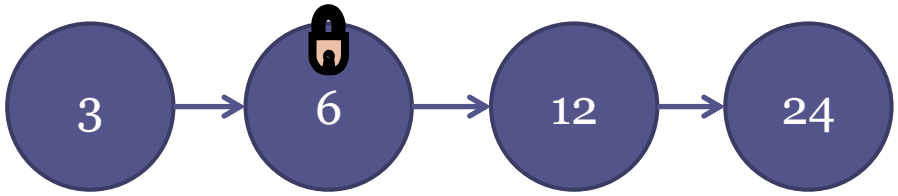
Thread D: Insert(9)



Combining Types

4. Wrong lock notification

Thread A:
Insert(9)



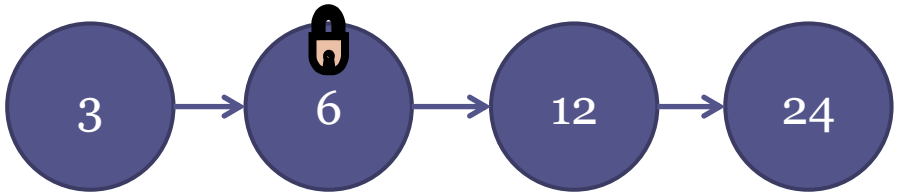
Combining Types

4. Wrong lock notification

Thread A: Insert(9)

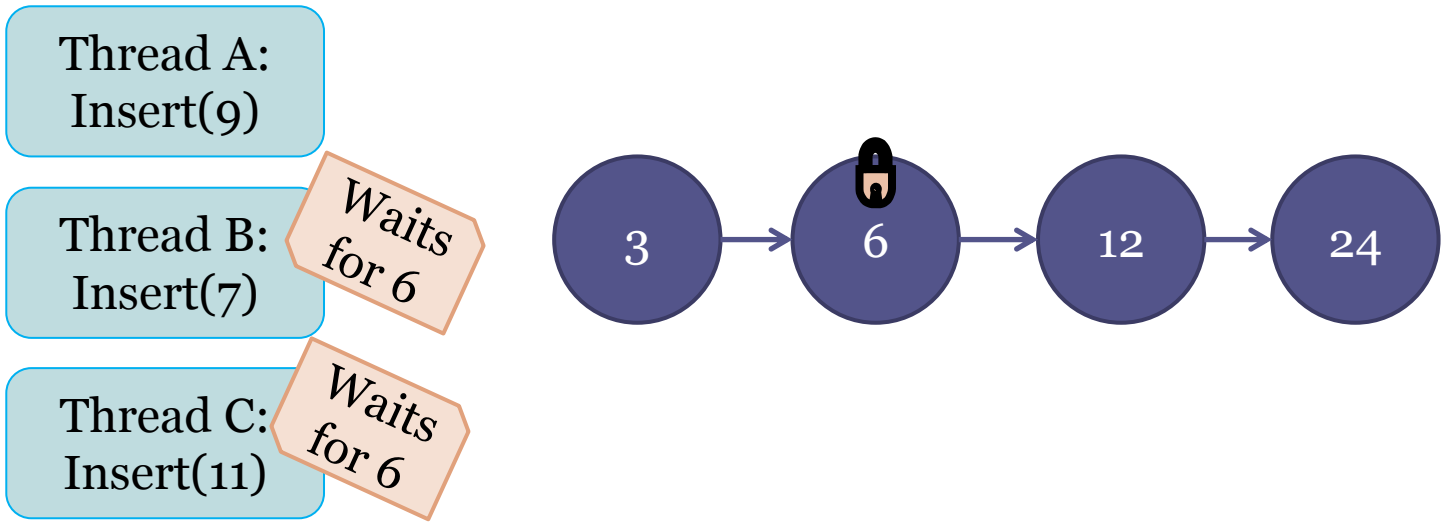
Thread B: Insert(7)

Thread C: Insert(11)



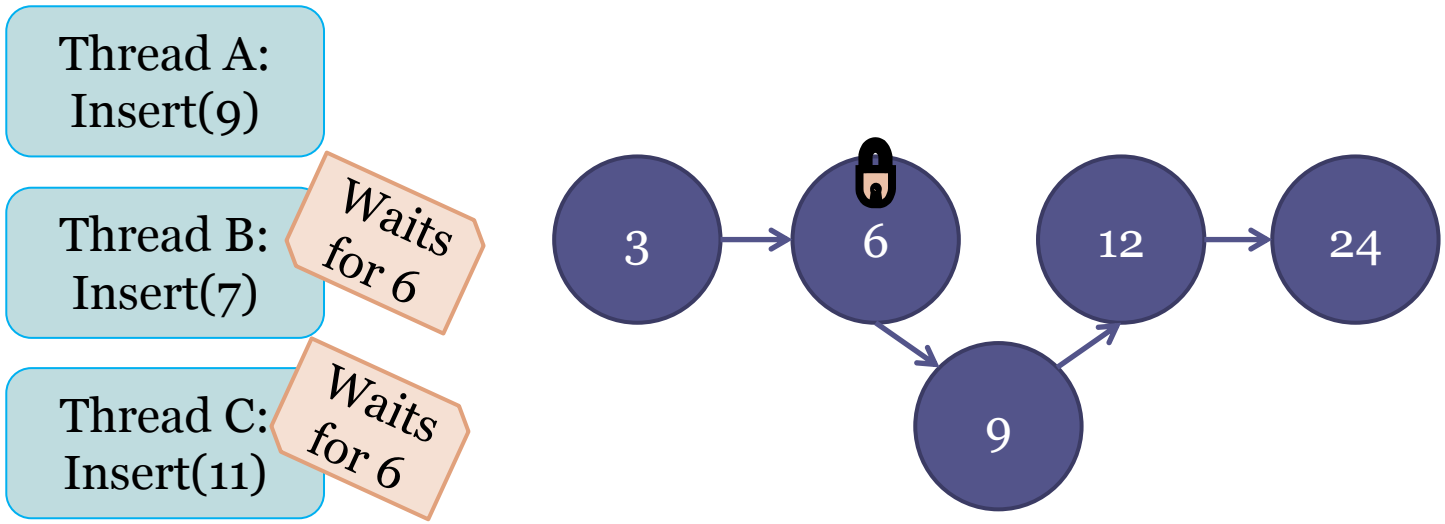
Combining Types

4. Wrong lock notification



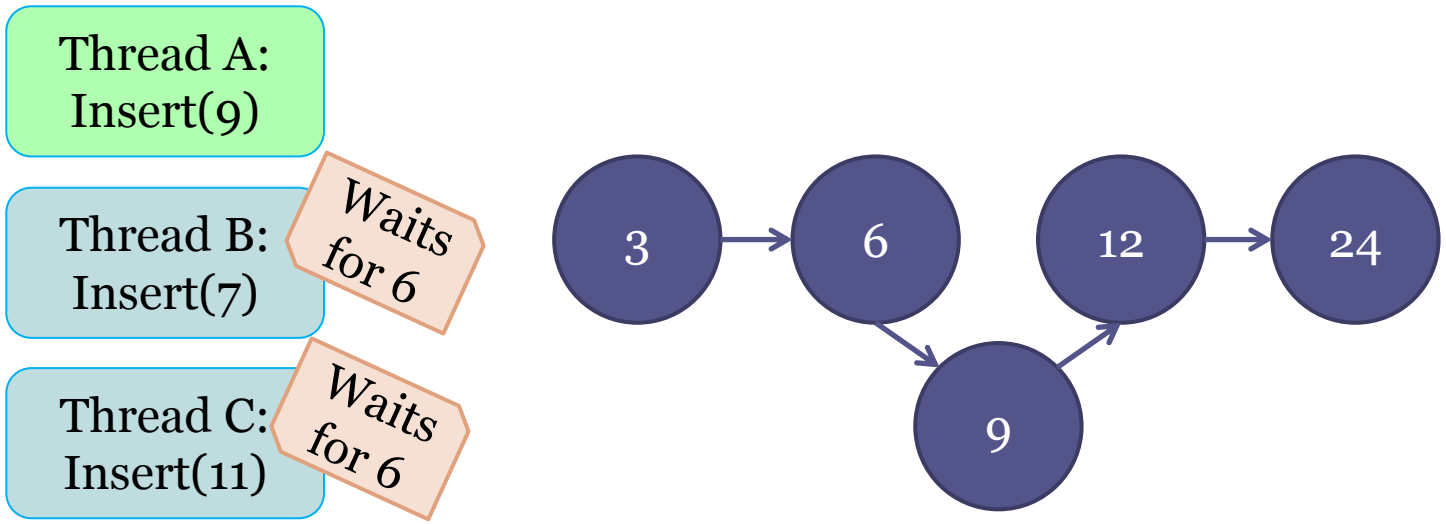
Combining Types

4. Wrong lock notification



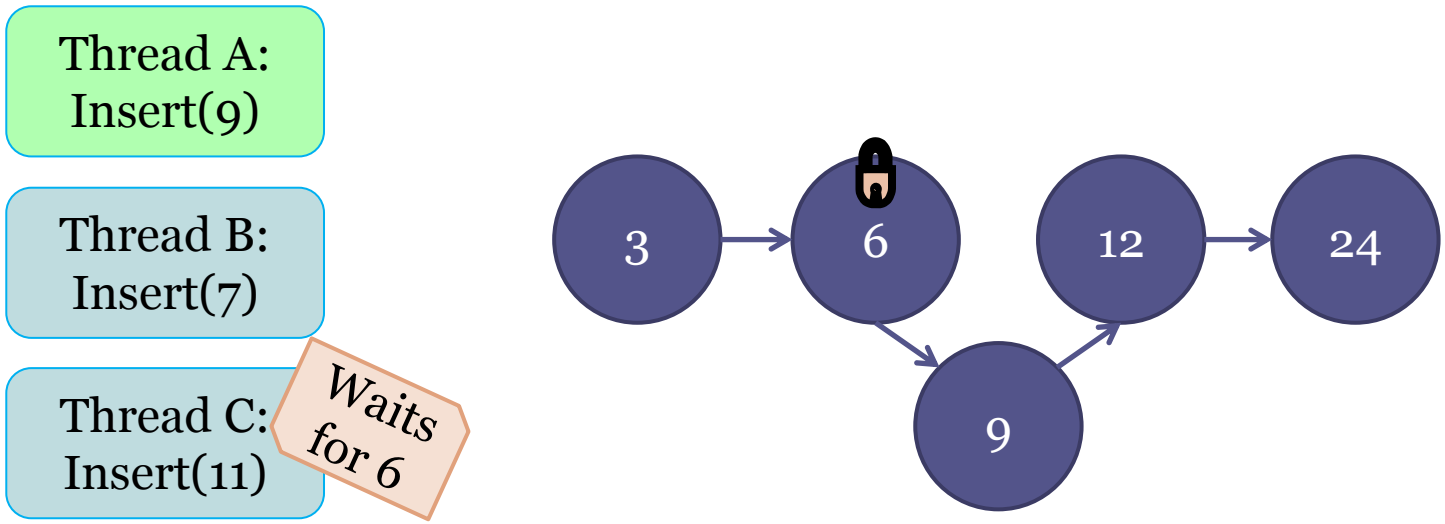
Combining Types

4. Wrong lock notification



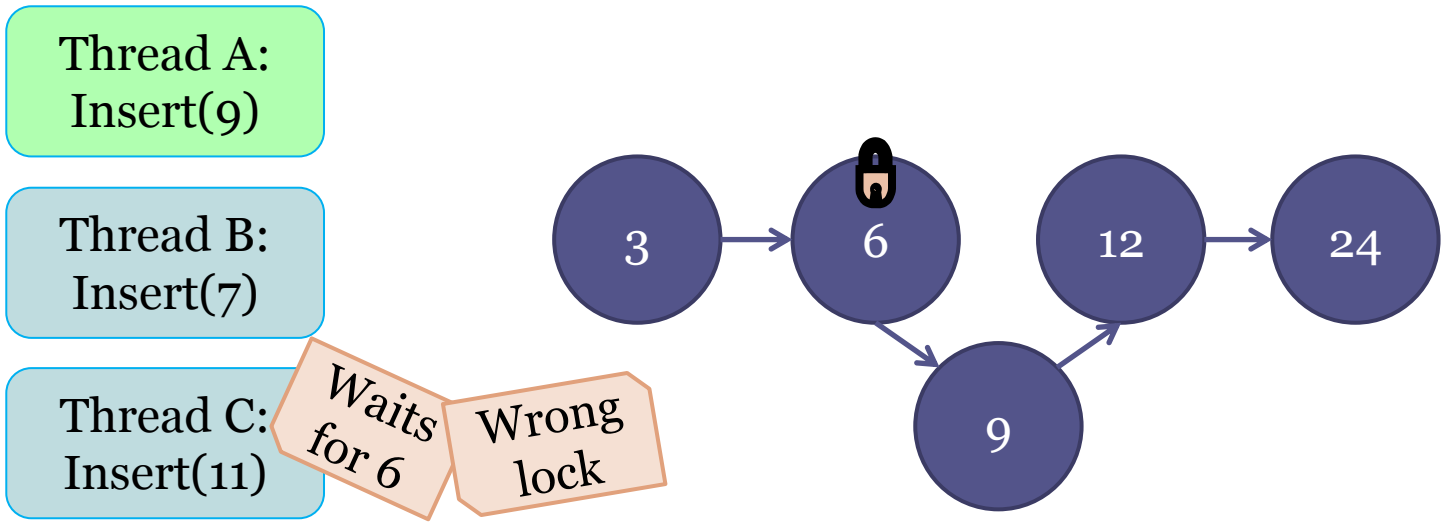
Combining Types

4. Wrong lock notification



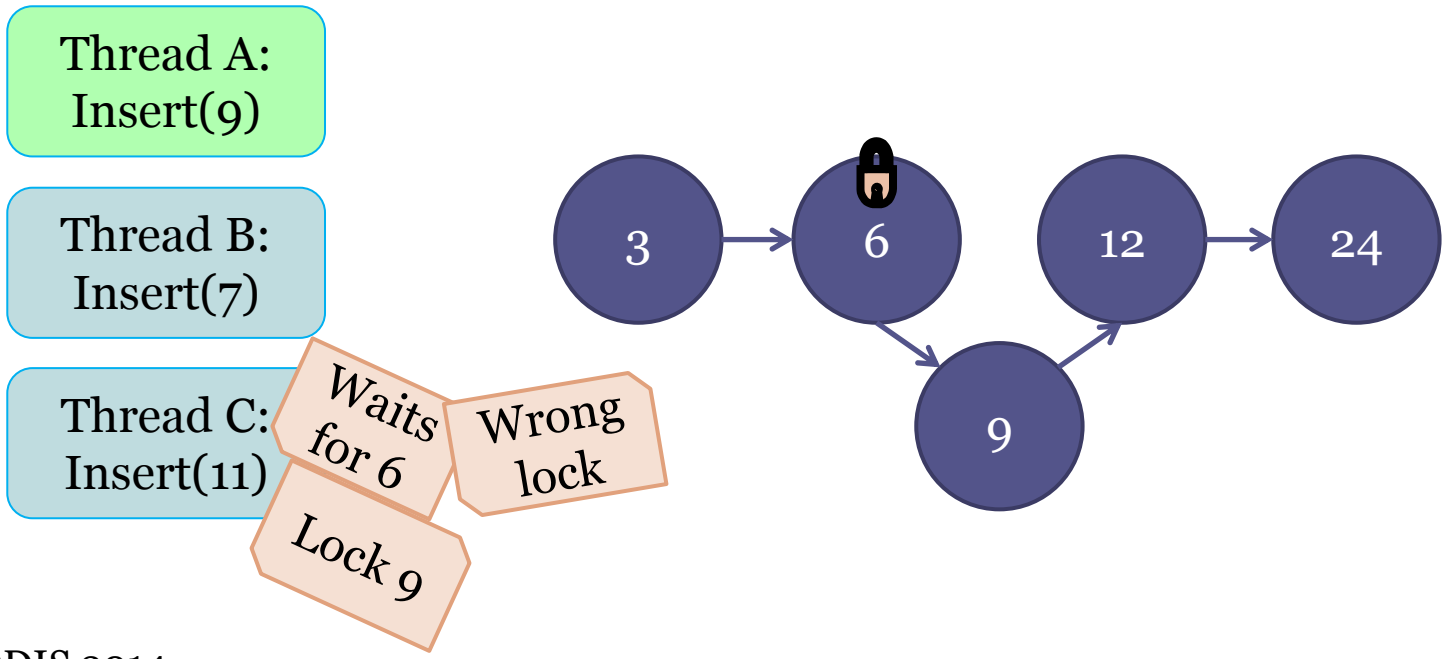
Combining Types

4. Wrong lock notification



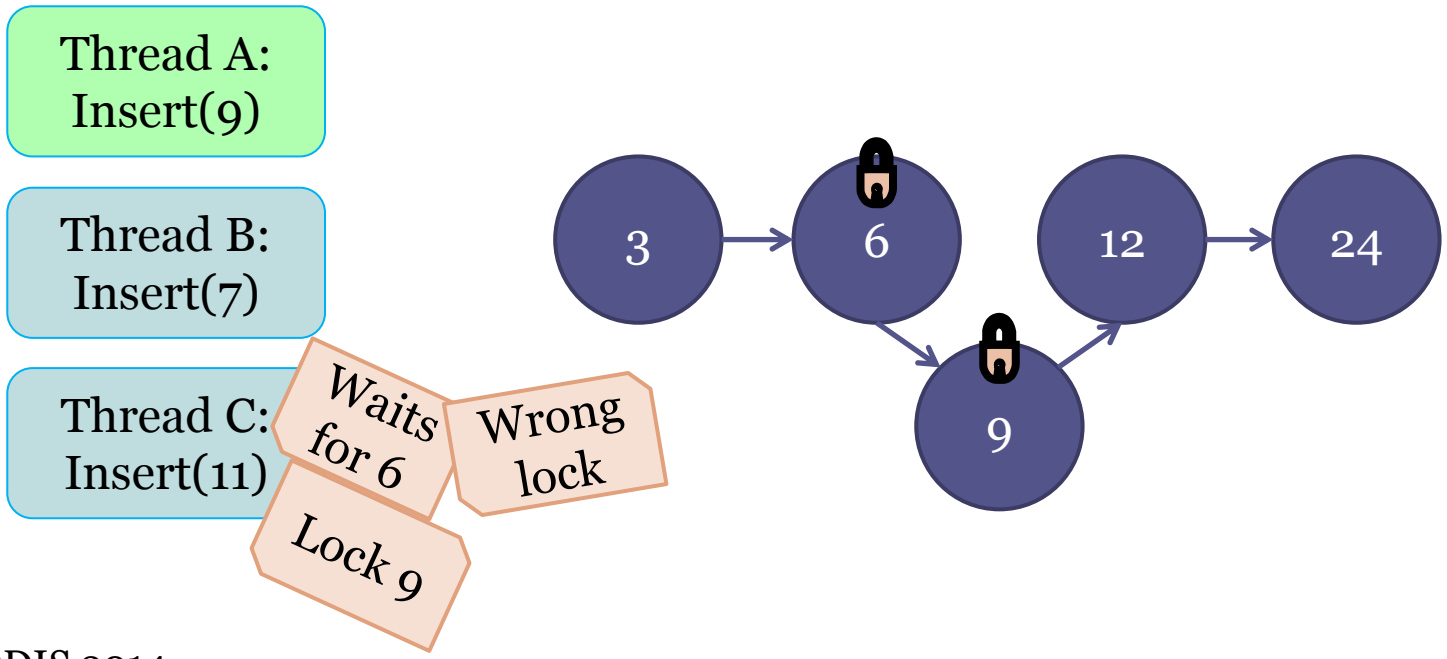
Combining Types

4. Wrong lock notification



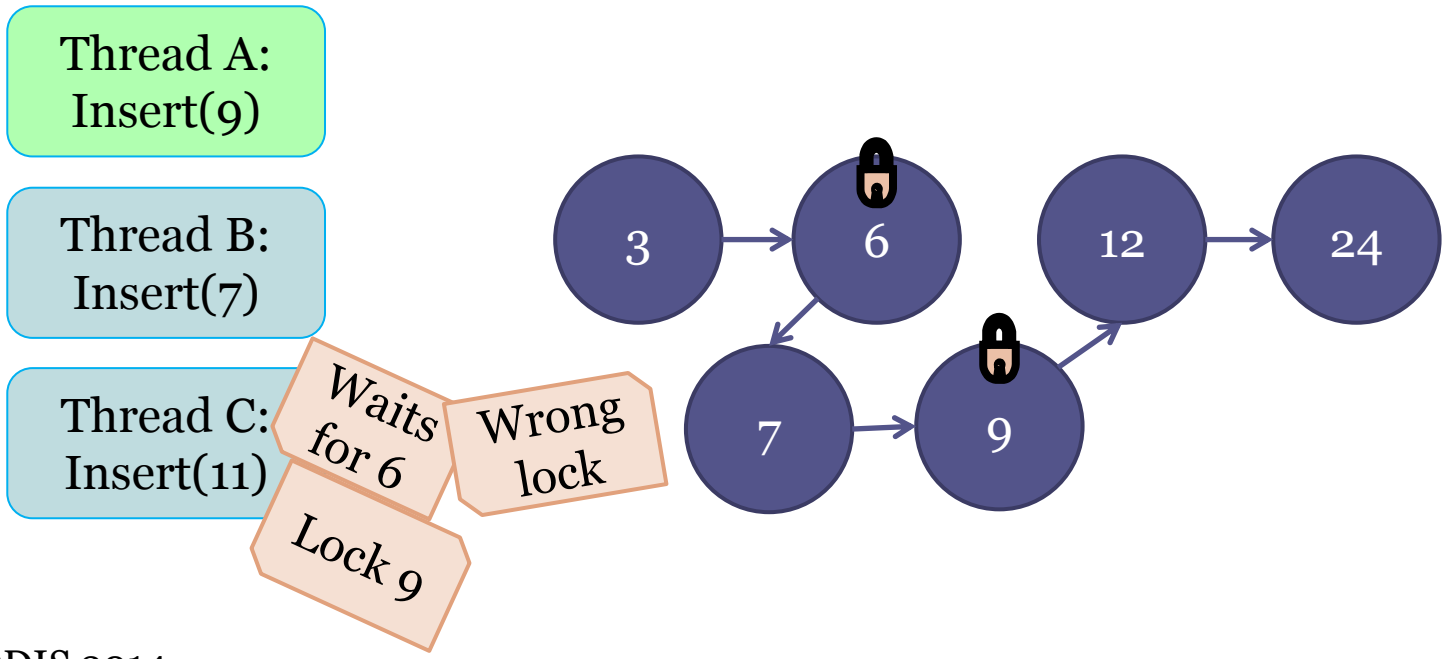
Combining Types

4. Wrong lock notification



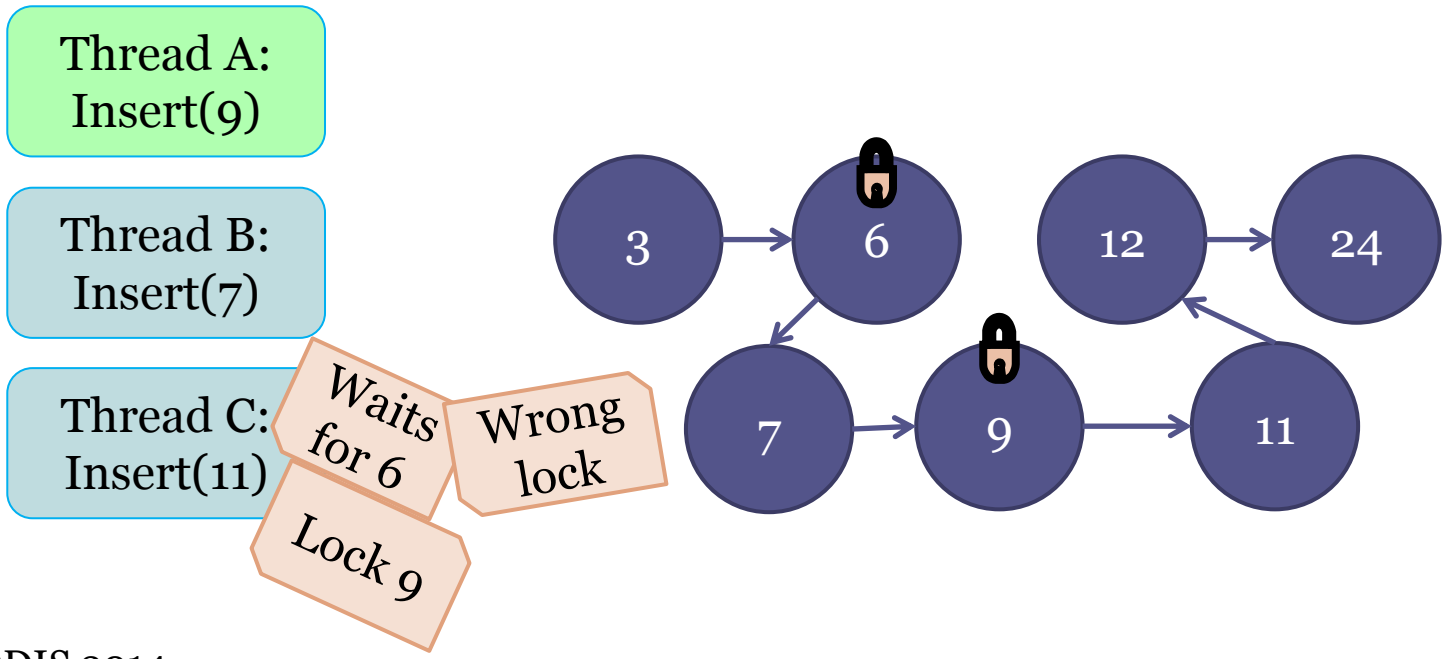
Combining Types

4. Wrong lock notification



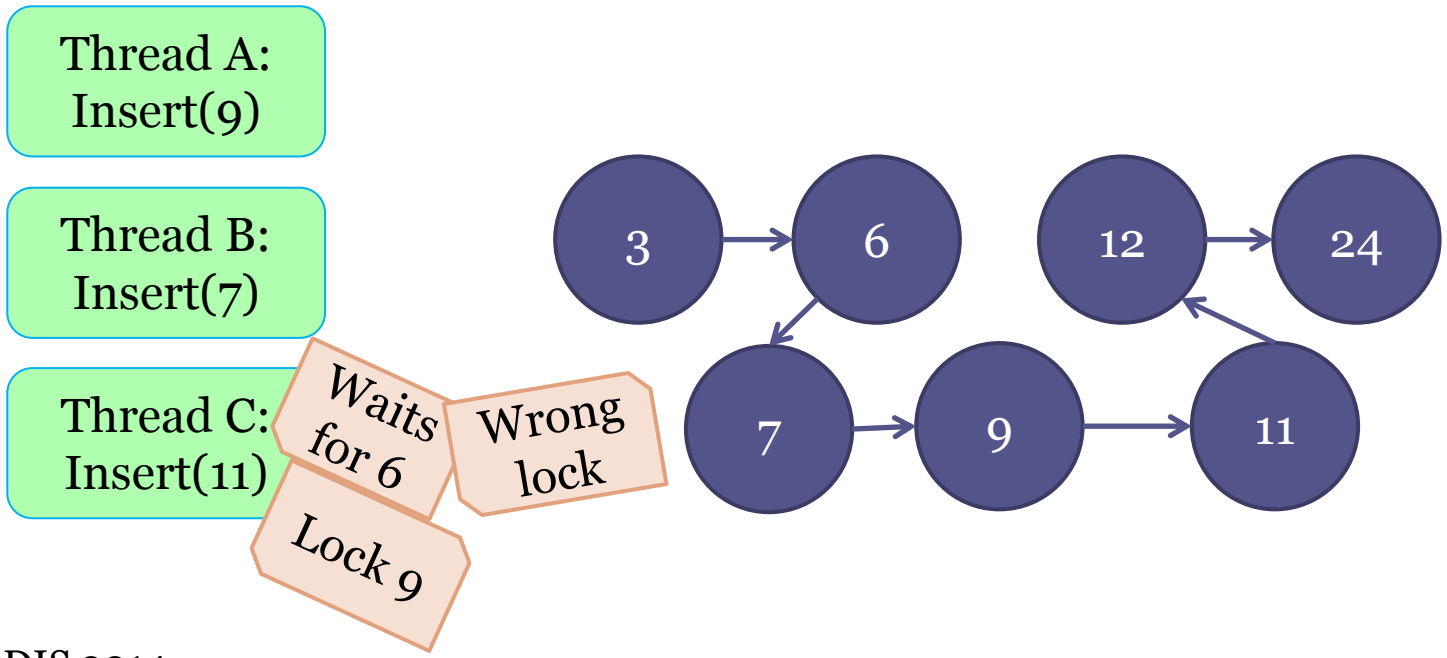
Combining Types

4. Wrong lock notification



Combining Types

4. Wrong lock notification



How does on-demand combining work?

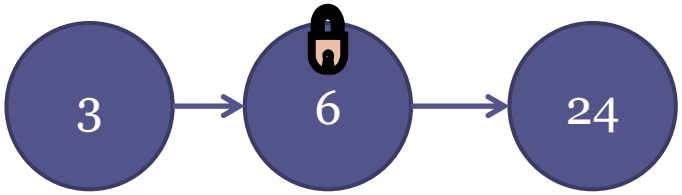
No Combining Demand – No Overhead

- If a thread did not observe contention, it is executed without any combining overhead
  - Thread observes contention if it needs to wait to acquire a lock

No Combining Demand – No Overhead

- If a thread did not observe contention, it is executed without any combining overhead
  - Thread observes contention if it needs to wait to acquire a lock

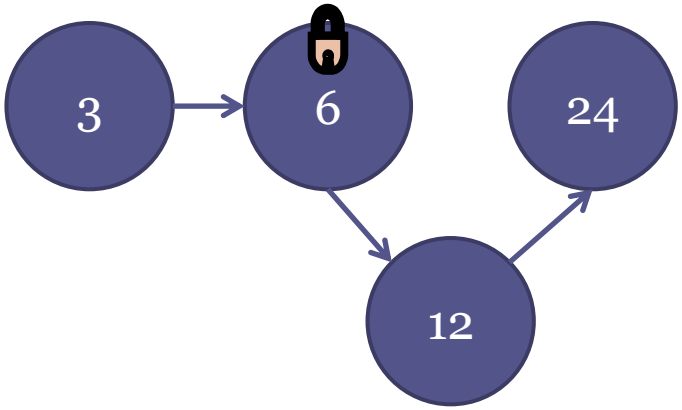
Thread A:
Insert(12)



No Combining Demand – No Overhead

- If a thread did not observe contention, it is executed without any combining overhead
  - Thread observes contention if it needs to wait to acquire a lock

Thread A:
Insert(12)



No Combining Demand - No Overhead

- If a thread did not observe contention, it is executed without any combining overhead
  - Thread observes contention if it needs to wait to acquire a lock

Thread A: Insert(12)

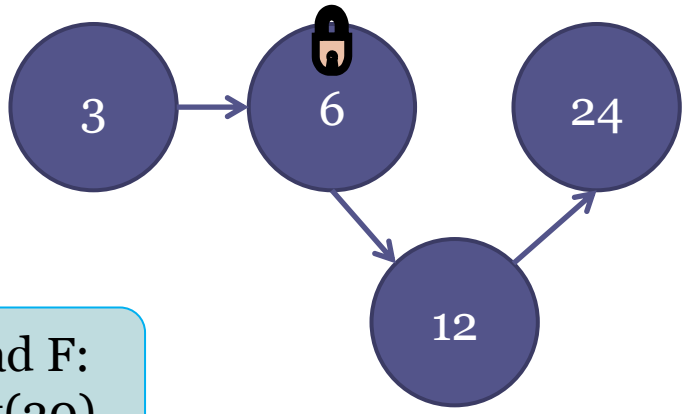
Thread B: Insert(9)

Thread D: Insert(10)

Thread C: Insert(9)

Thread E: Remove(10)

Thread F: Insert(20)



No Combining Demand – No Overhead

- If a thread did not observe contention, it is executed without any combining overhead
  - Thread observes contention if it needs to wait to acquire a lock

Thread A: Insert(12)

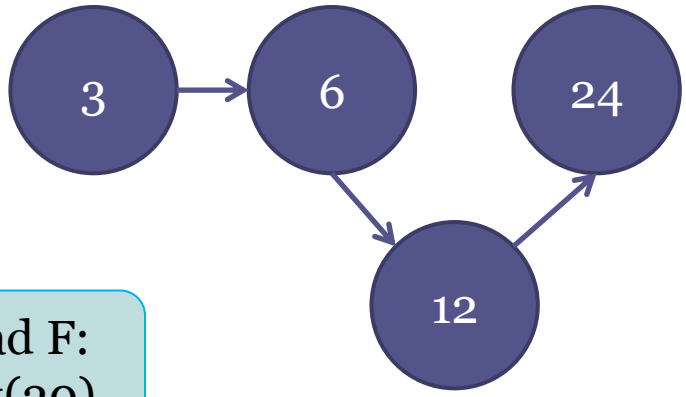
Thread B: Insert(9)

Thread D: Insert(10)

Thread C: Insert(9)

Thread E: Remove(10)

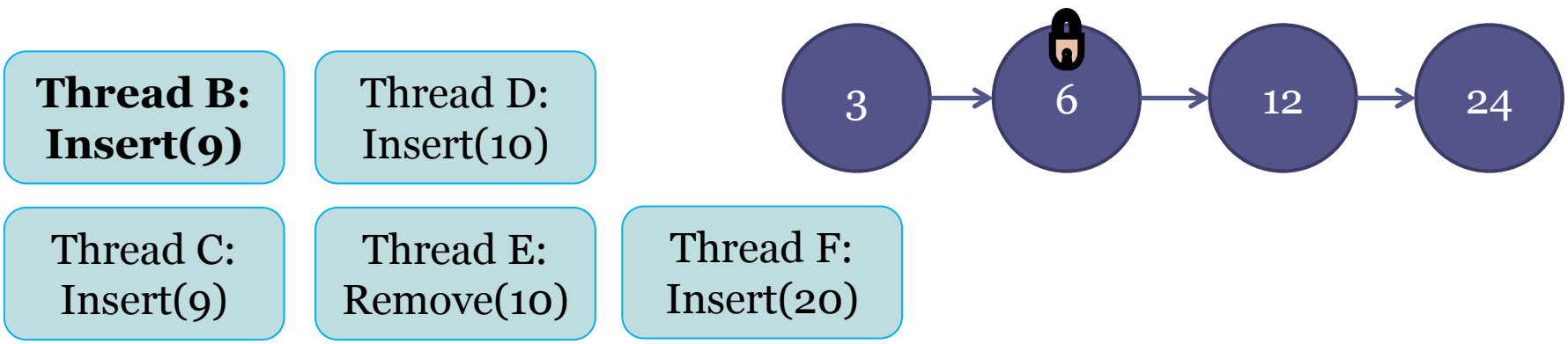
Thread F: Insert(20)



How does the local combining work?

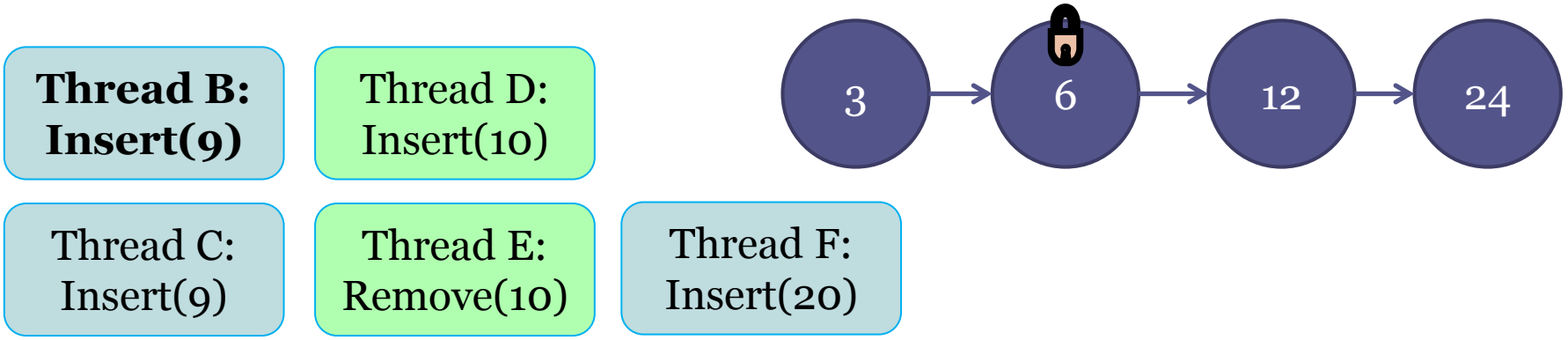
Combine Locally

- A thread that locked after observing contention collects operations of contending threads



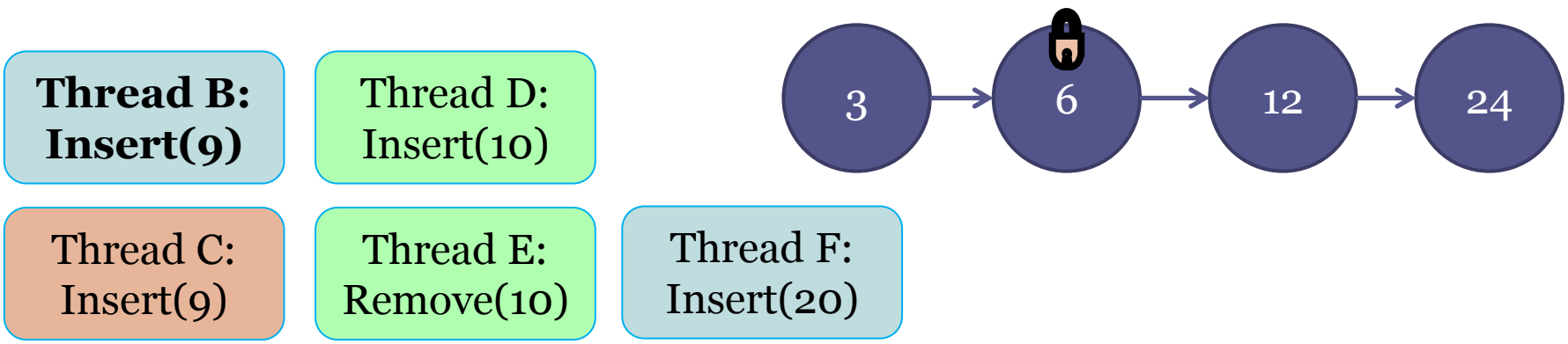
Combine Locally

- A thread that locked after observing contention collects operations of contending threads
  - Complementary operations are eliminated



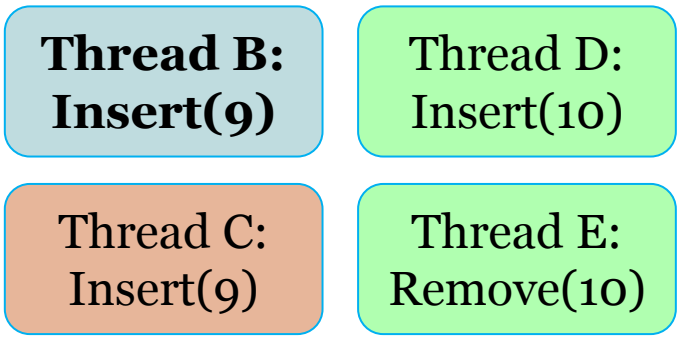
Combine Locally

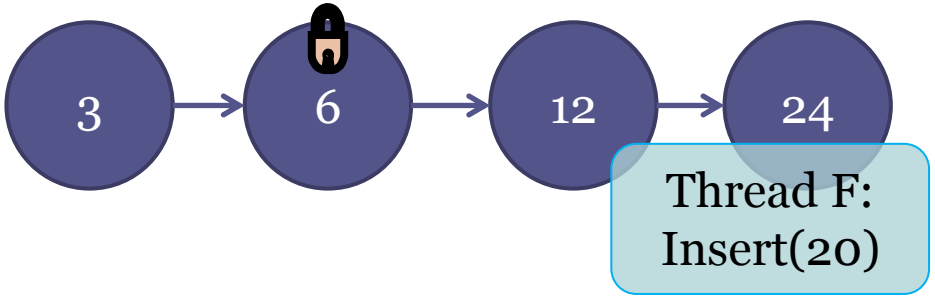
- A thread that locked after observing contention collects operations of contending threads
  - Complementary operations are eliminated
  - Identical operations are eliminated



Combine Locally

- A thread that locked after observing contention collects operations of contending threads
  - Complementary operations are eliminated
  - Identical operations are eliminated
  - Operations requiring a different lock are notified





Combine Locally

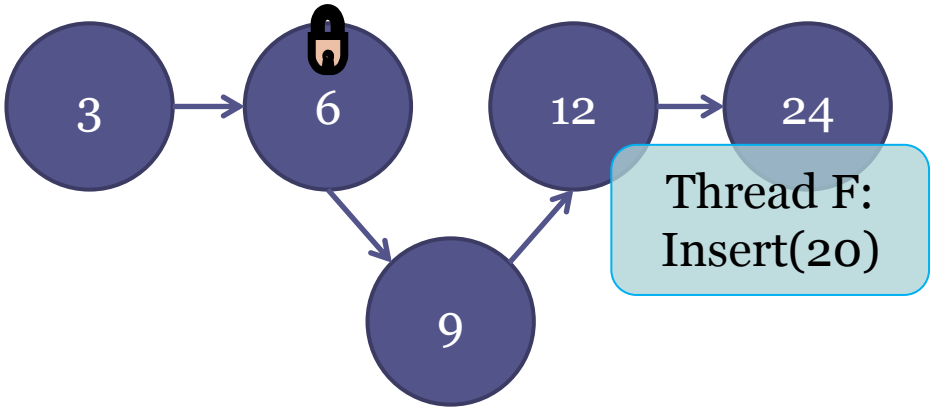
- A thread that locked after observing contention collects operations of contending threads
  - Complementary operations are eliminated
  - Identical operations are eliminated
  - Operations requiring a different lock are notified

Thread B: Insert(9)

Thread D: Insert(10)

Thread C: Insert(9)

Thread E: Remove(10)



Combine Locally

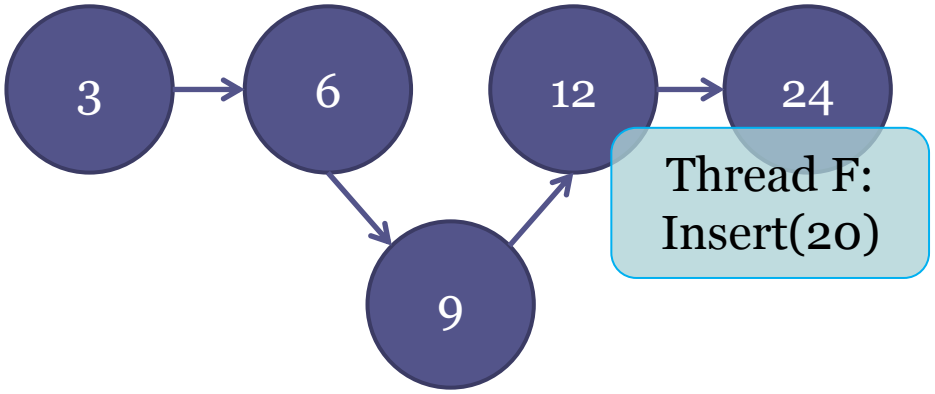
- A thread that locked after observing contention collects operations of contending threads
  - Complementary operations are eliminated
  - Identical operations are eliminated
  - Operations requiring a different lock are notified

Thread B: Insert(9)

Thread D: Insert(10)

Thread C: Insert(9)

Thread E: Remove(10)



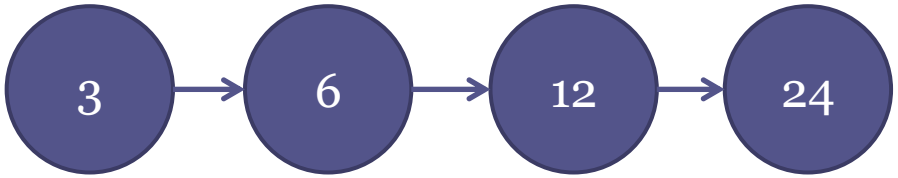
How to handle operations requiring multiple locks?

Operations Requiring Multiple Locks

- LCD is applied independently for each lock
- Operations acquiring multiple locks are split
  - Each sub-operation acquires one lock
  - Each sub-operation may be executed by a different combining thread
- Example: $Remove(k)$

Operations Requiring Multiple Locks

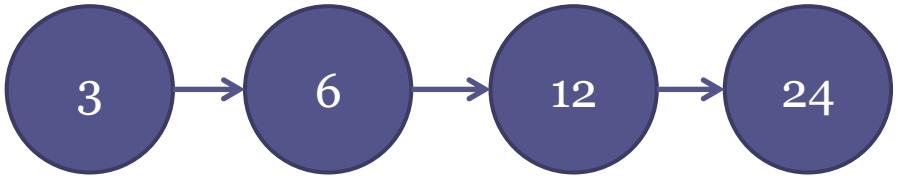
- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



Thread A:
Remove(12)

Operations Requiring Multiple Locks

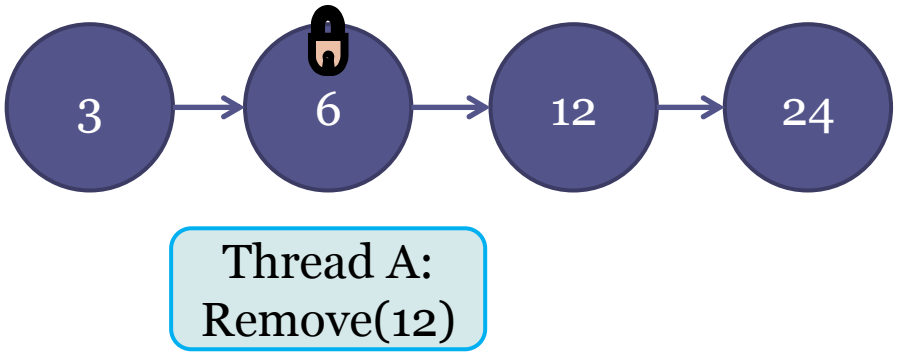
- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



Thread A:
Remove(12)

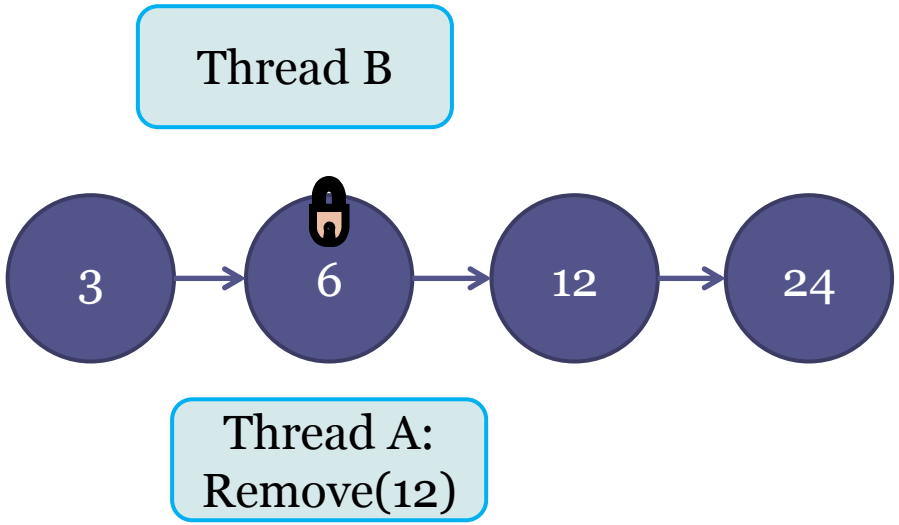
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



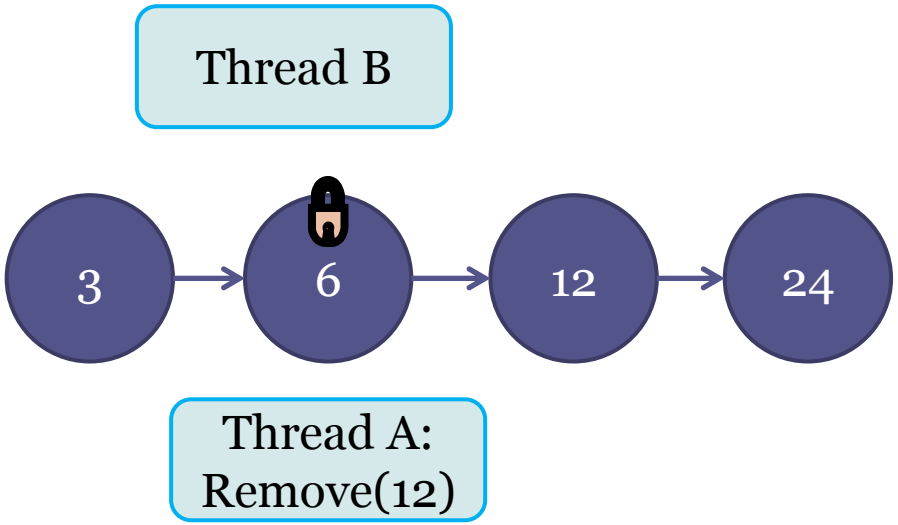
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



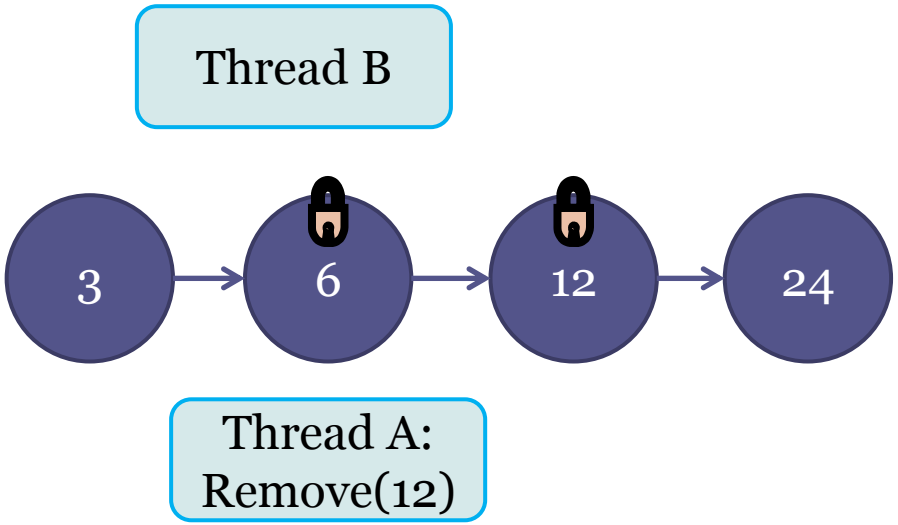
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



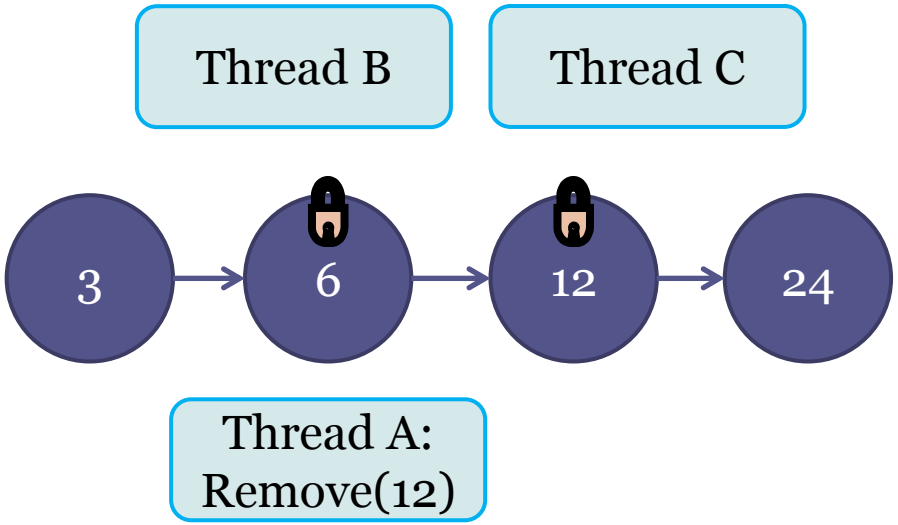
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



Operations Requiring Multiple Locks

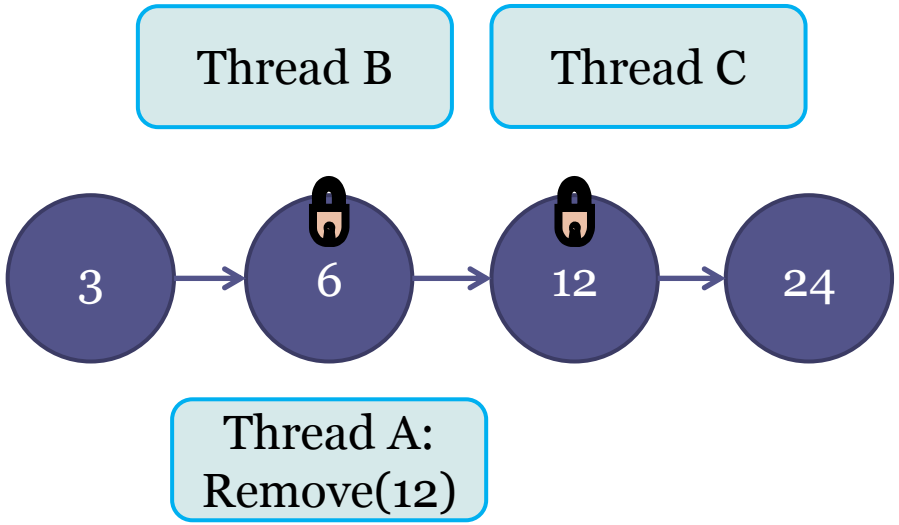
- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor

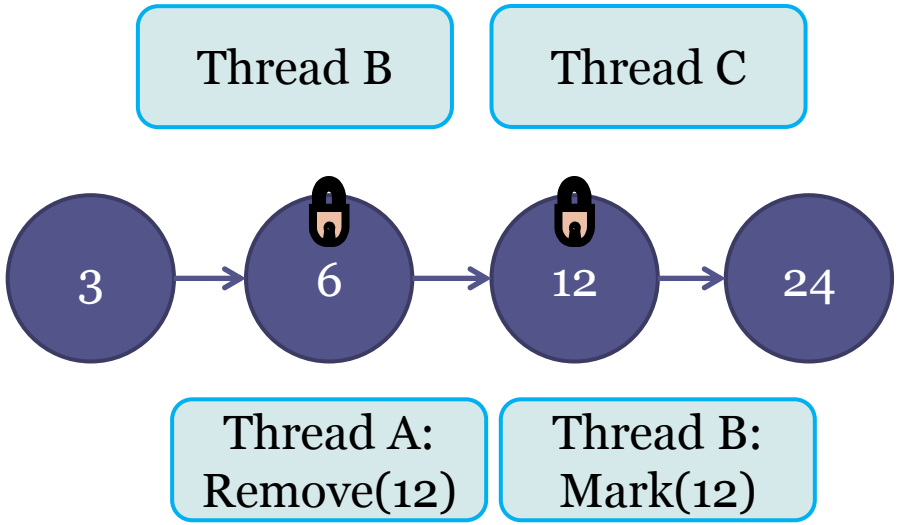
New sub-operation: Mark



Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor

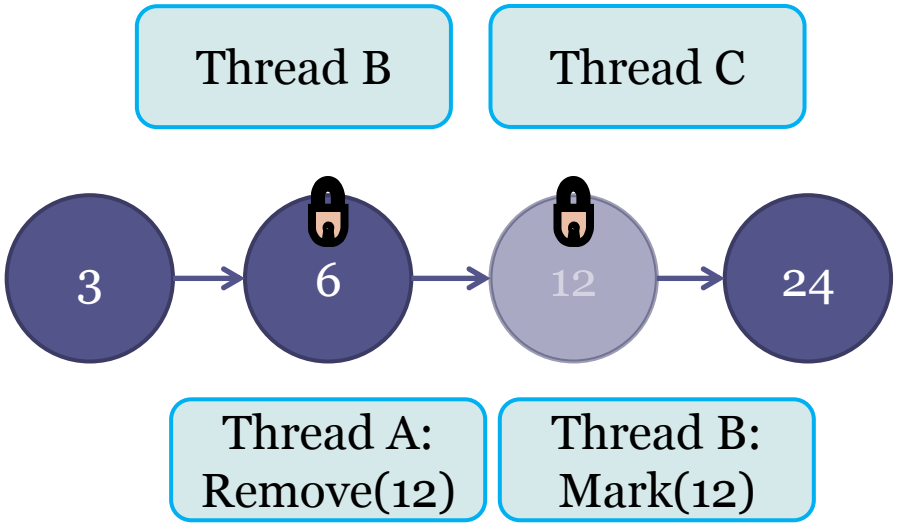
New sub-operation: Mark



Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor

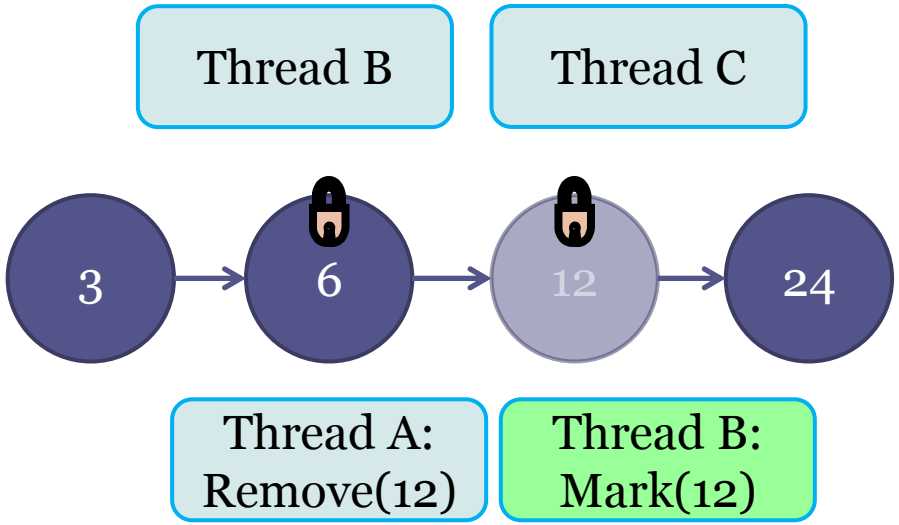
New sub-operation: Mark



Operations Requiring Multiple Locks

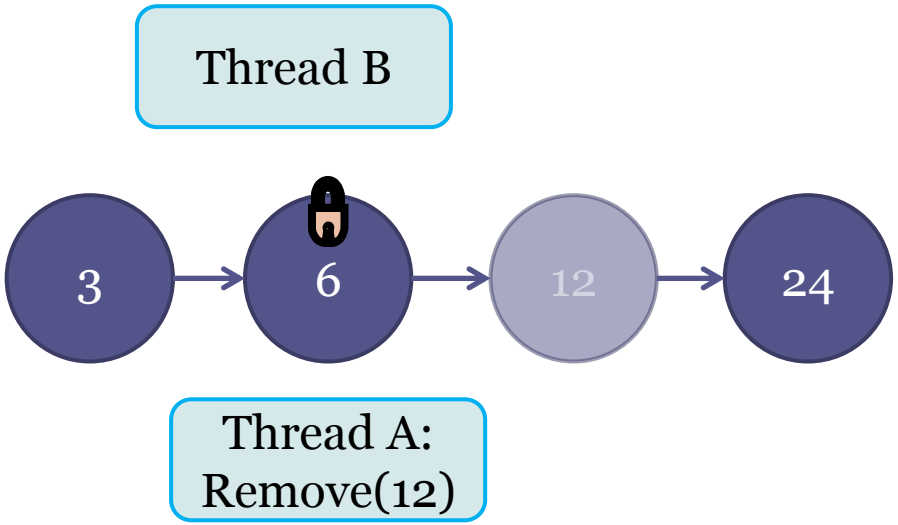
- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor

New sub-operation: Mark



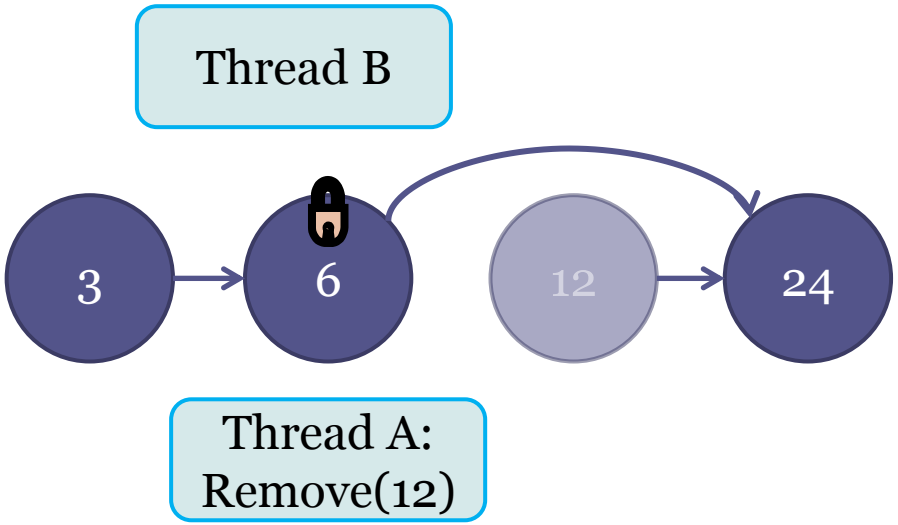
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



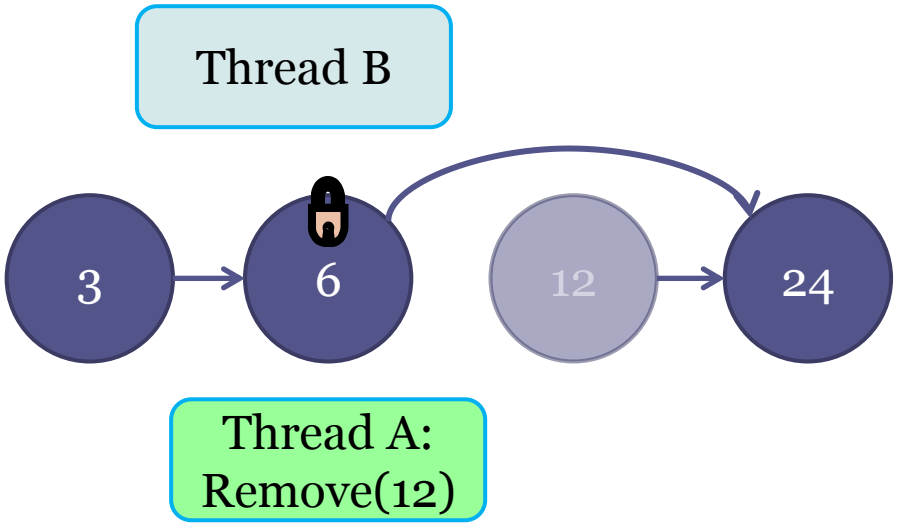
Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



Operations Requiring Multiple Locks

- Example: $Remove(k)$
  - Search for a node p s.t.: $p.key < k \leq next(p).key$
  - Lock p
  - If the key of p's successor is k
    - Lock p's successor
    - Mark the successor
    - Update p's successor



How to integrate LCD with the Java lock?

Java Lock Integration

- Most of the LCD mechanism was integrated into the Java lock
- The Java lock maintains a queue of threads waiting to acquire the lock
- The LCD requires the list of contending threads
  - Which is exactly the list of threads waiting to lock
  - We leverage this queue to the LCD purposes

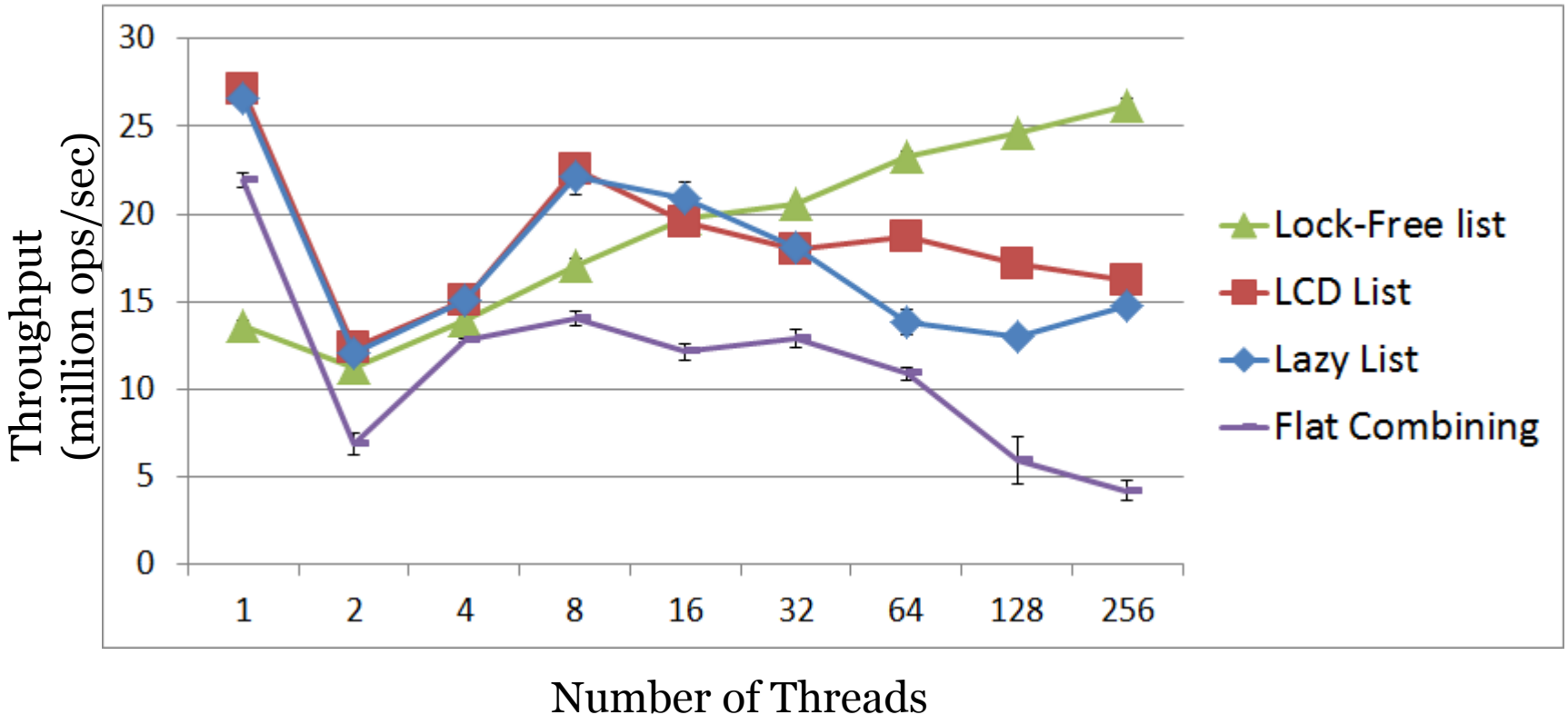
LCD Evaluation

- LCD list was implemented in Java and compared to state-of-the-art algorithms
  - Linked-list algorithms
    - The Lazy List [Heller, Herlihy, Luchangco, Moir, Scherer III, Shavit]
    - The Lock-free List [Harris]
  - Combining algorithms
    - Flat Combining (global lock) [Hendler, Incze, Shavit, Tzafrir]

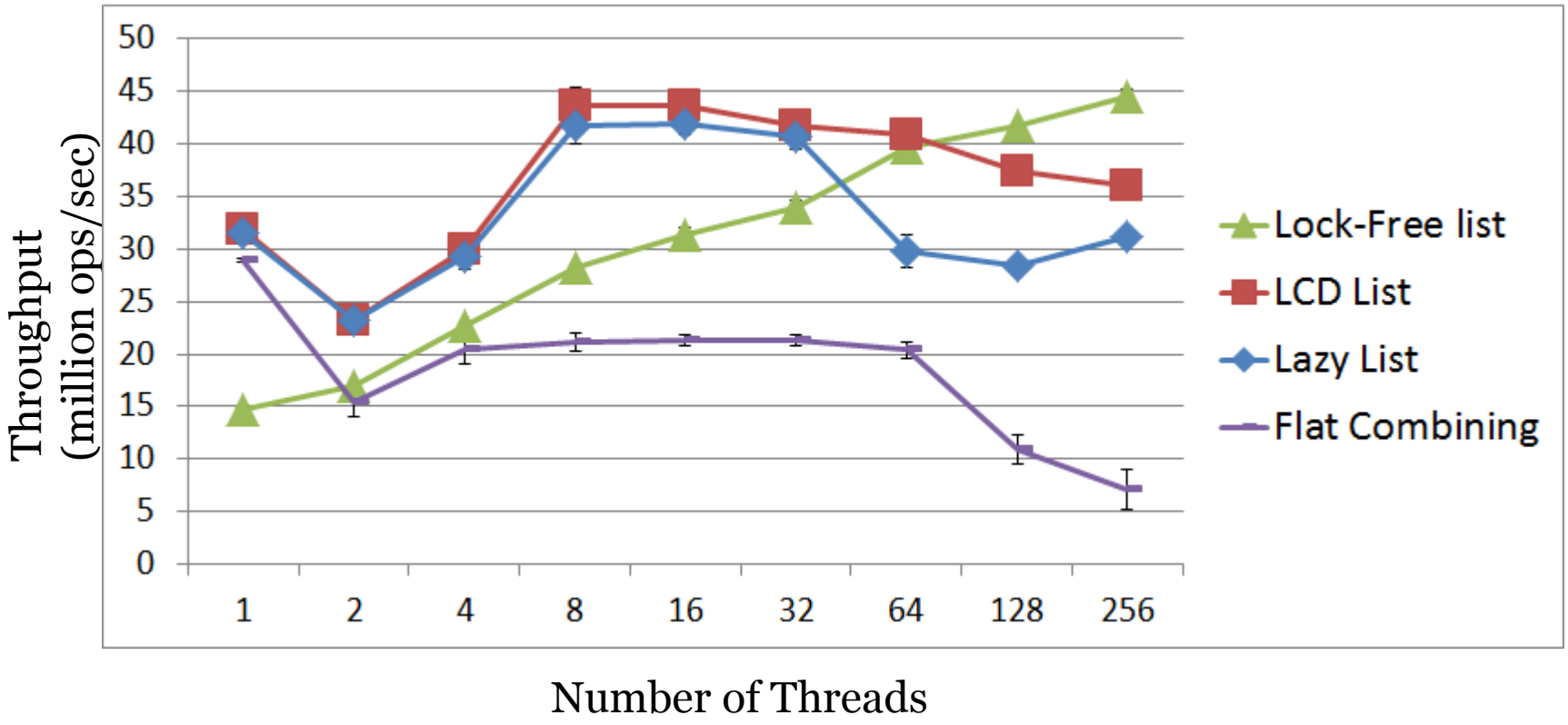
Benchmarks

- Threads randomly chose operation type and key
  - Different workloads for the operation type
    - 0% contains, 50% insert, 50% remove
    - 60% contains, 20% insert, 20% remove
  - Different key ranges
    - 128, 1024
- List was populated to half of its expected size
  - 64, 512
- AMD Opteron machine with 64 h/w threads

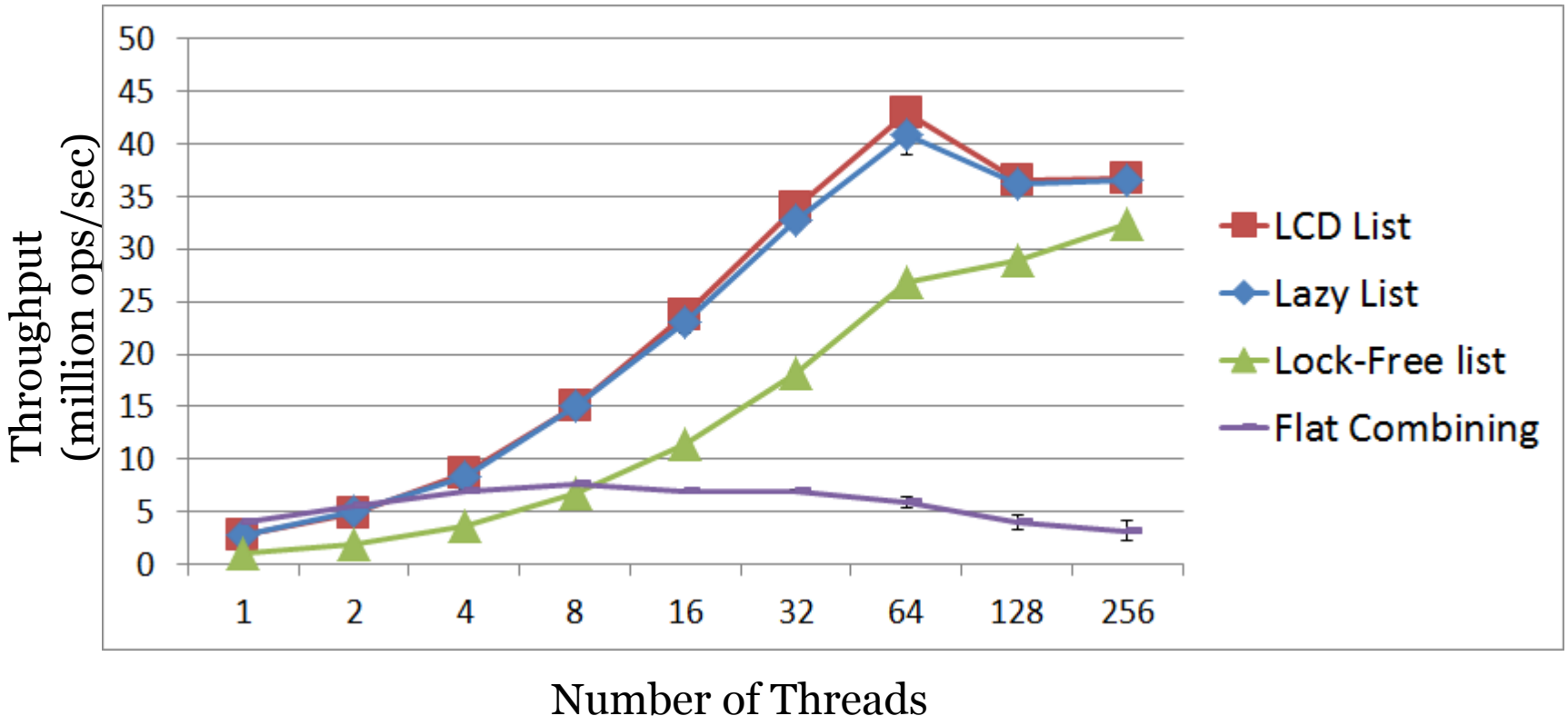
0% contains, 50% insert, 50% remove
Key range: 128



60% contains, 20% insert, 20% remove
Key range: 128



60% contains, 20% insert, 20% remove
Key range: 1024



Summary

- We presented a new combining technique
  - Local
    - Applied independently for contended resources
  - On-demand
    - No overhead if there is no contention
- We demonstrated it on the linked-list
  - And showed it improves performance