

# Практические построения на списках

Nikita Koval<sup>1</sup>    Roman Elizarov<sup>2</sup>

<sup>1</sup>Researcher, JetBrains  
ndkoval@ya.ru

<sup>2</sup>Kotlin Team Lead, JetBrains  
elizarov@gmail.com

ИТМО 2020

# План

1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# План

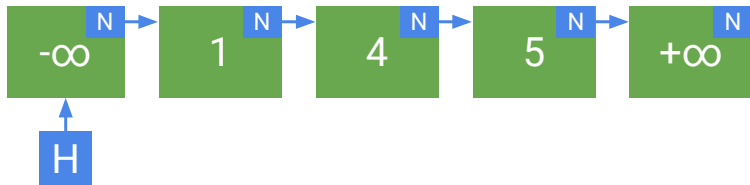
1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Множество

```
interface Set {  
    void add(int key)  
  
    boolean contains(int key)  
  
    void remove(int key)  
}
```

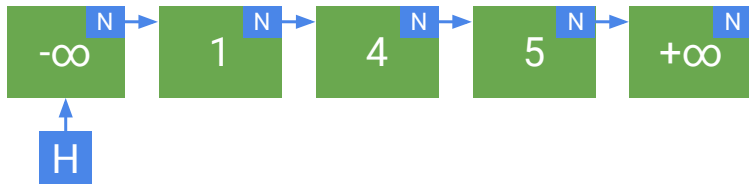
# Односвязный список

Элементы упорядочены по возрастанию

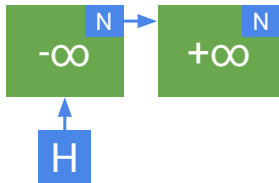


# Односвязный список

Элементы упорядочены по возрастанию



Пустой список состоит из двух граничных элементов



# Односвязный список: алгоритм

- Элементы упорядочены по возрастанию
- Ищем окно  $(cur, next)$ , что  $cur.KEY < k \leq next.KEY$  и  $cur.N = next$
- Искомый элемент будет в  $next$
- Новый элемент добавляем между  $cur$  и  $next$

# Односвязный список: псевдокод

```
class Node(var N: Node, val key: Int)

val head = Node( $-\infty$ , Node( $\infty$ , null))

(Node, Node) findWindow(key) {
  cur := head
  next := cur.N
  while (next.key < key):
    cur = next
    next = cur.N
  return (cur, next)
}
```



# Односвязный список: псевдокод

```
class Node(var N: Node, val key: Int)

val head = Node( $-\infty$ , Node( $\infty$ , null))

(Node, Node) findWindow(key) {
  cur := head
  next := cur.N
  while (next.key < key):
    cur = next
    next = cur.N
  return (cur, next)
}
```

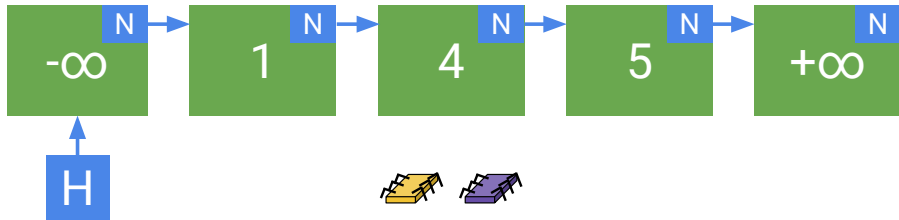
```
boolean contains(key) {
  (cur, next) := findWindow(key)
  return next.key == key
}
```

```
void add(key) {
  (cur, next) := findWindow(key)
  if (next.key != key)
    cur.N = Node(key, next)
}
```

```
void remove(key) {
  (cur, next) := findWindow(key)
  if (next.key == key)
    cur.N = next.N;
}
```

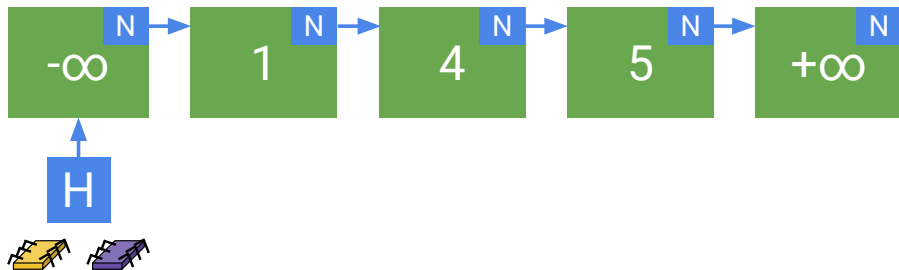
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



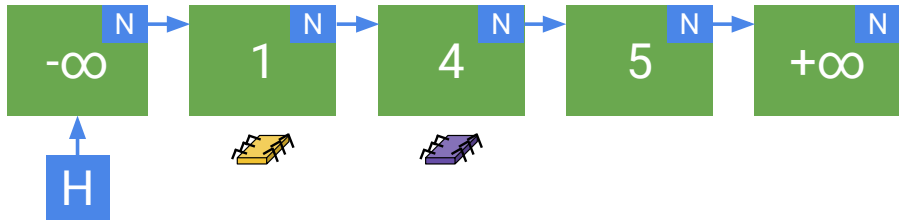
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



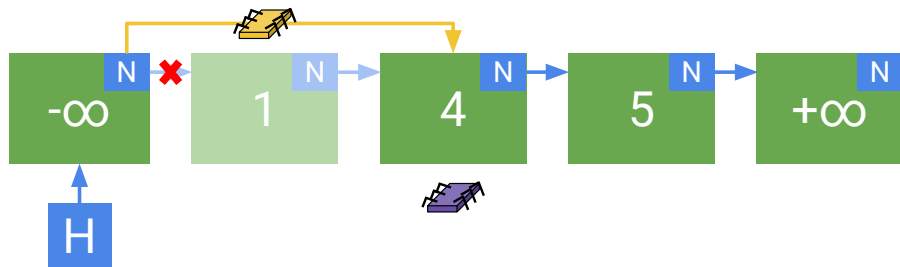
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



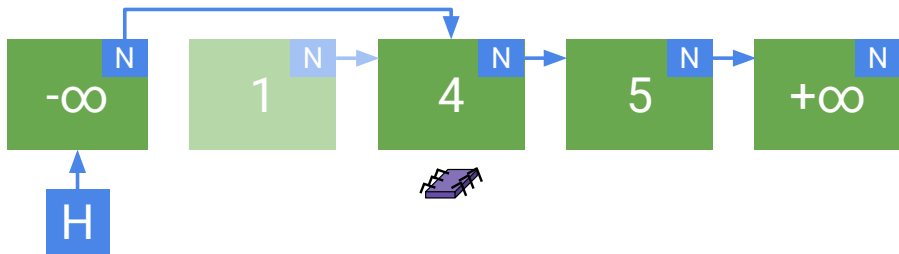
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



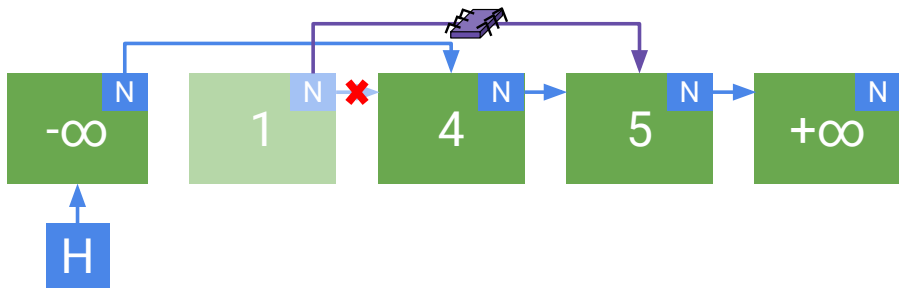
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



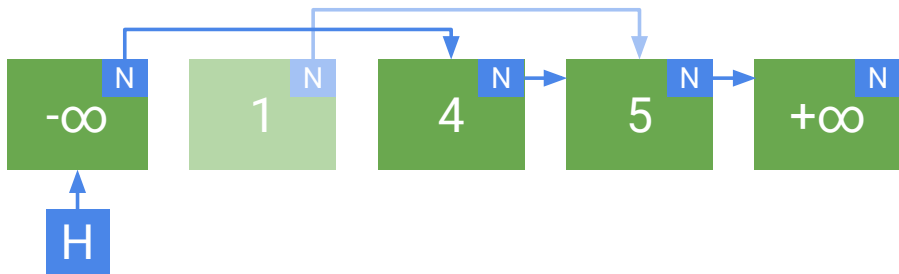
# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



# Проблема

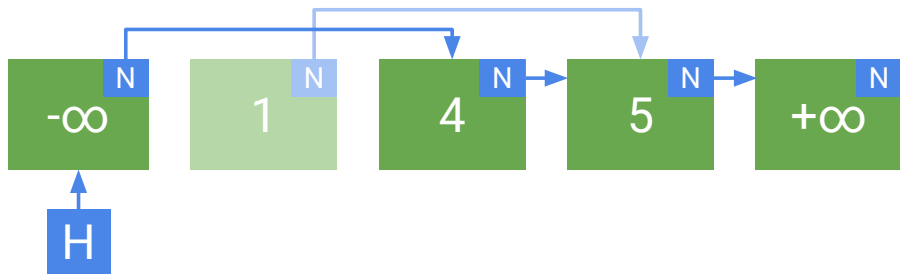
Жёлтый удаляет «1», фиолетовый удаляет «4»





# Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



У фиолетового ничего не вышло!

# План

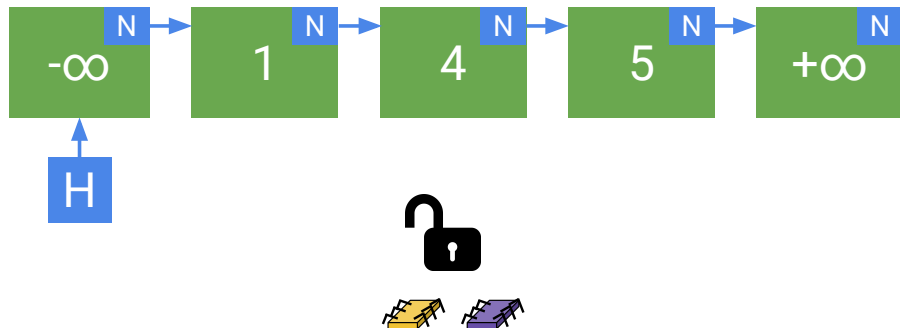
1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Грубая синхронизация

- Coarse-grained locking
- Используем общую блокировку для всех операций
- $\Rightarrow$  обеспечиваем последовательное исполнение
- В Java для этого можно использовать `synchronized` или `j.u.c.locks.ReentrantLock`

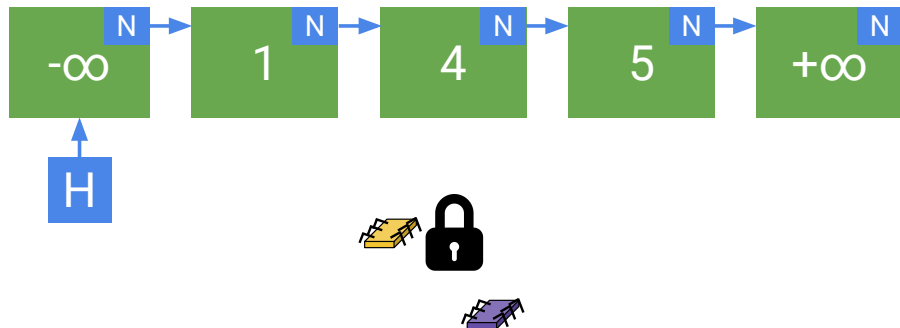
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



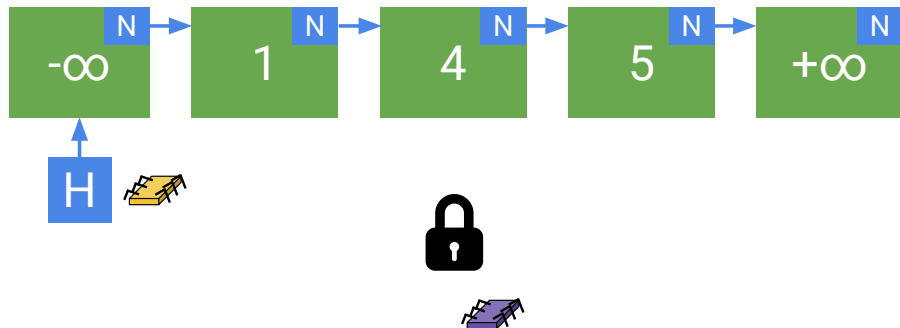
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



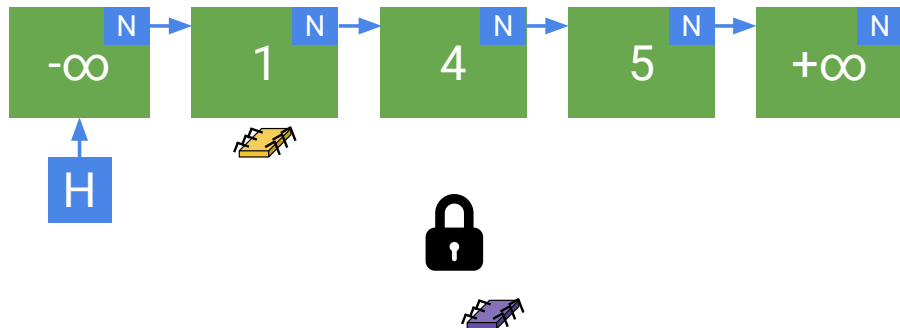
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



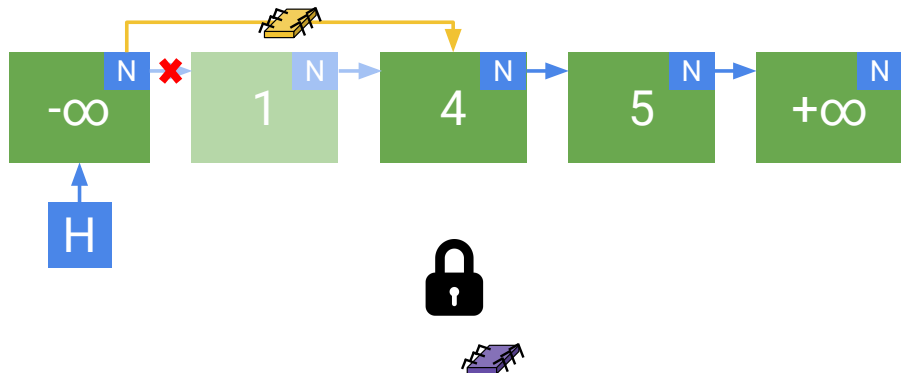
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



# Пример

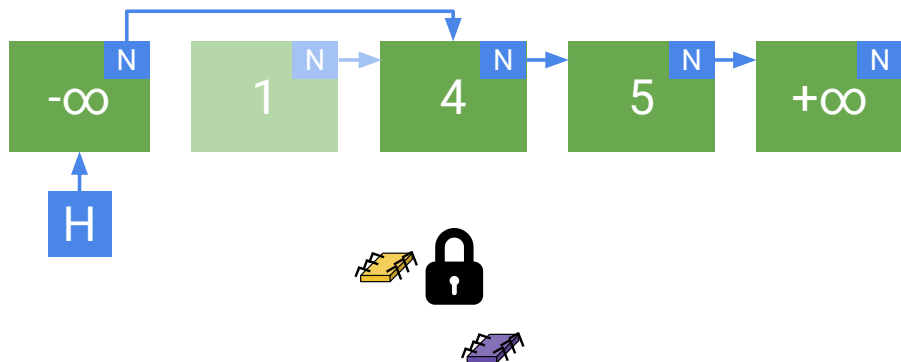
Жёлтый удаляет «1», фиолетовый удаляет «4»





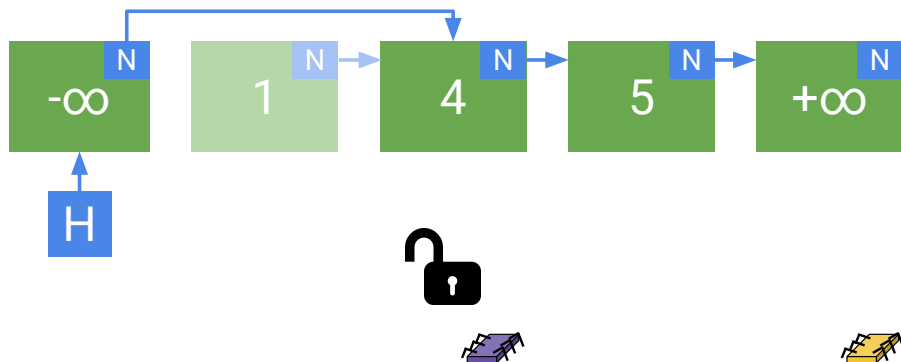
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



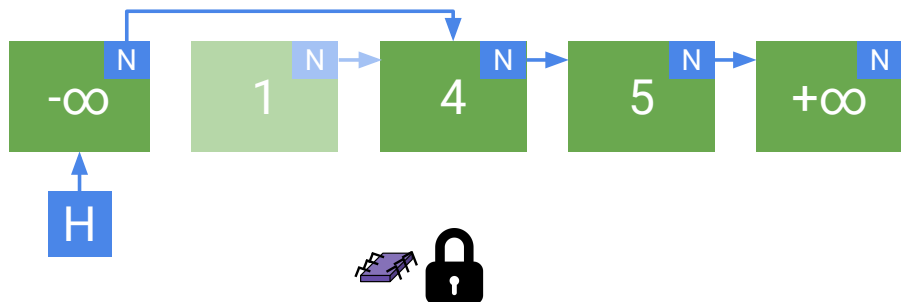
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



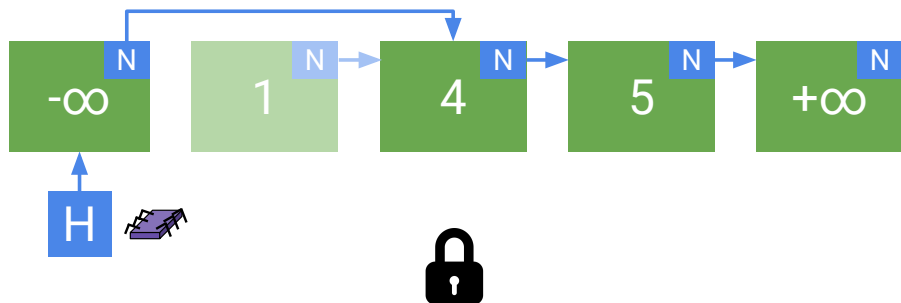
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



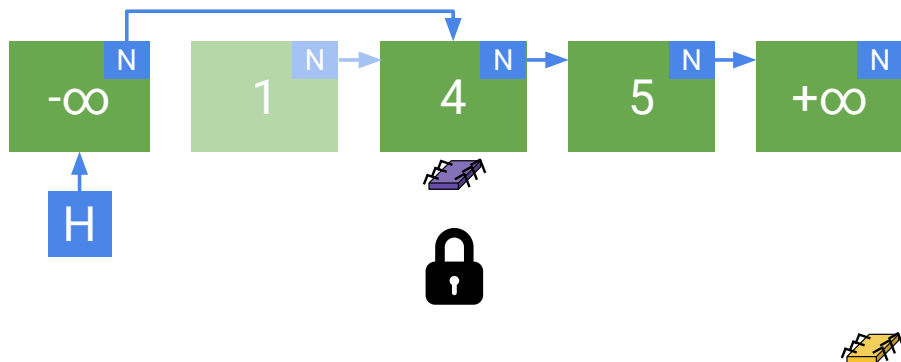
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



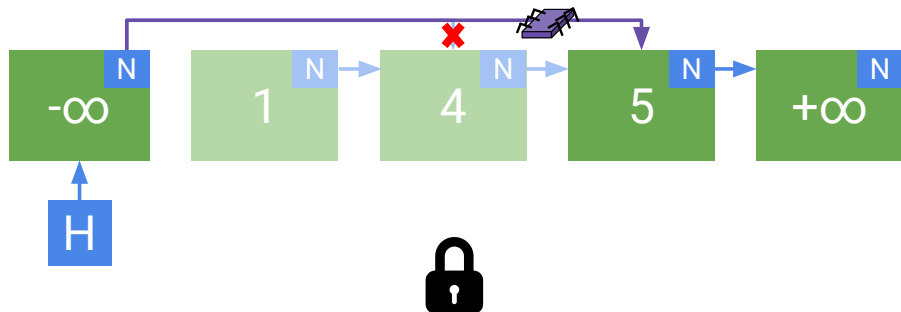
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



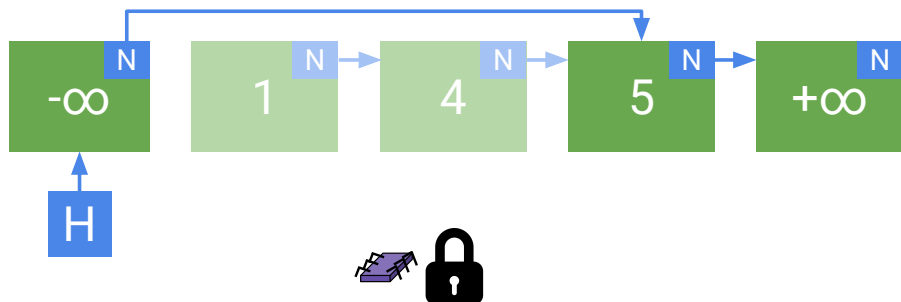
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



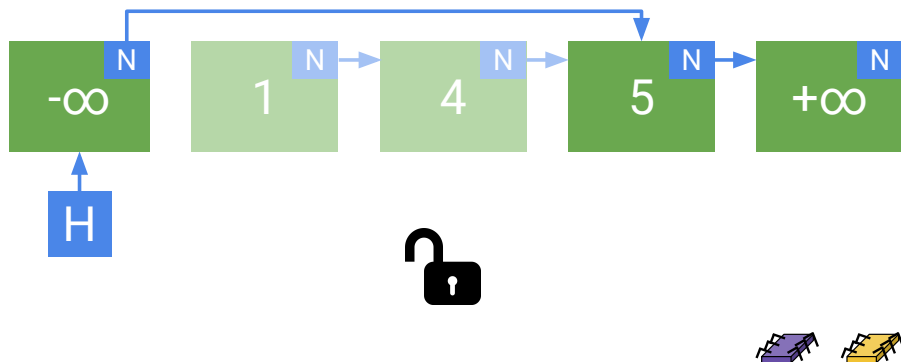
# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



# Пример

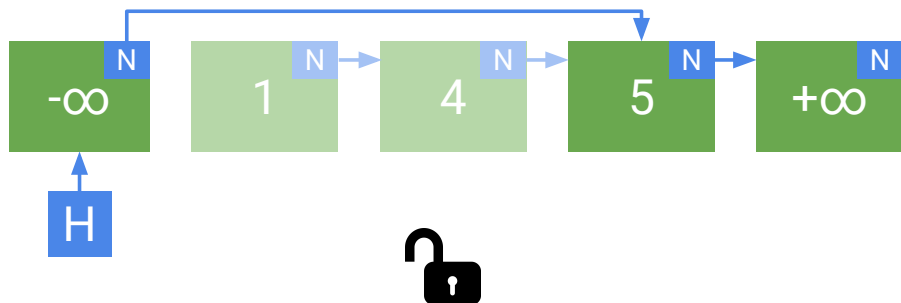
Жёлтый удаляет «1», фиолетовый удаляет «4»





# Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



На этот раз удалили оба элемента!



# Грубая синхронизация: задание

Задание «linked-list-set»: необходимо сделать реализацию множества корректной с помощью грубой блокировки.

# Грубая синхронизация: псевдокод

```
synchronized boolean contains(key) {  
    (cur, next) = findWindow(key)  
    return next.key == key  
}
```

```
synchronized void add(key) {  
    (cur, next) = findWindow(key)  
    if (next.key != key)  
        cur.N = Node(key, next)  
}
```

```
synchronized void remove(key) {  
    (cur, next) = findWindow(key)  
    if (next.key != key) return  
    cur.N = next.N  
}
```

# План

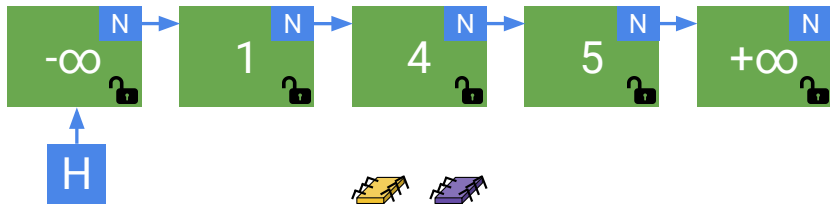
1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Тонкая синхронизация

- Fine-Grained locking
- Своя блокировка на каждый элемент
- При поиске окна держим блокировку на текущий и следующий элементы

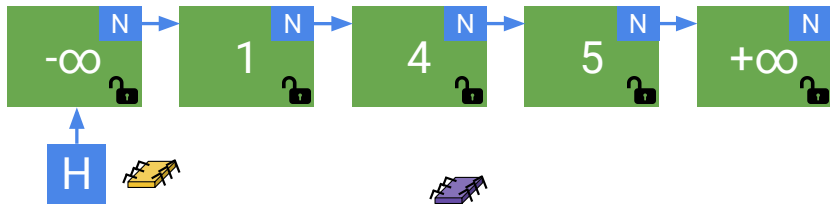
# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



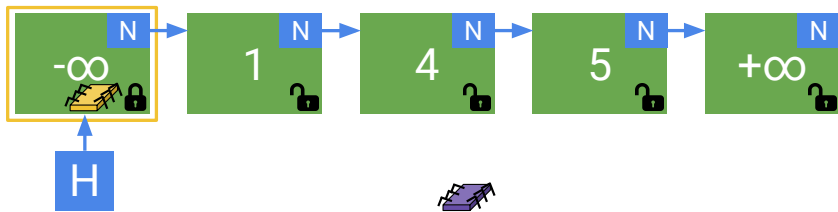
# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»

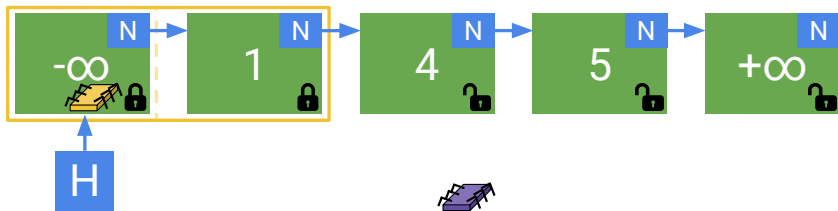


Жёлтый берёт блокировку на голову списка ...



# Пример

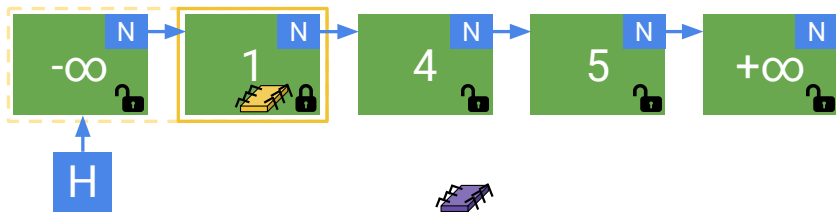
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и на следующий элемент

# Пример

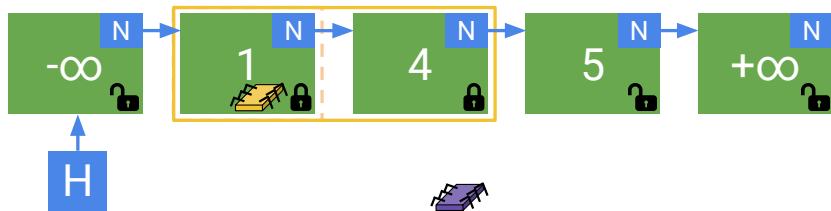
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый отпускает блокировку на голову списка ...

# Пример

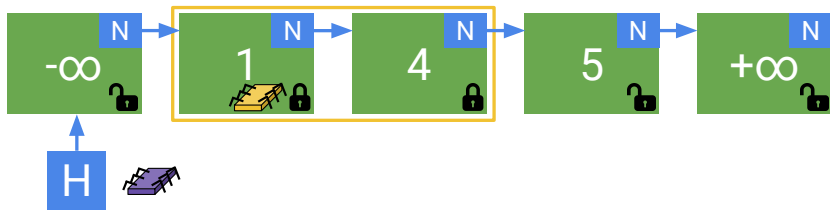
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и берёт блокировку на «4», нашёл окно

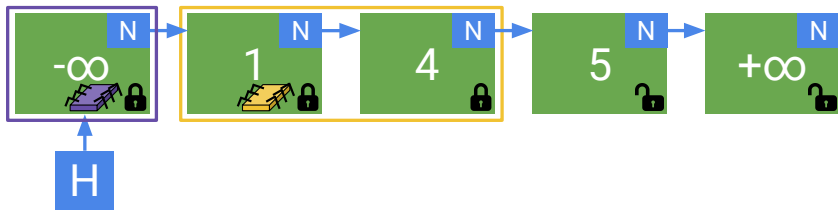
# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



# Пример

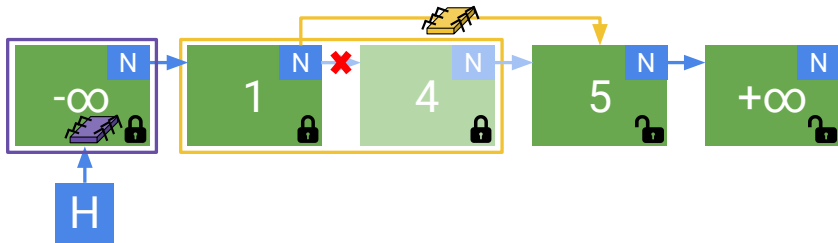
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на голову списка и ждет жёлтого

# Пример

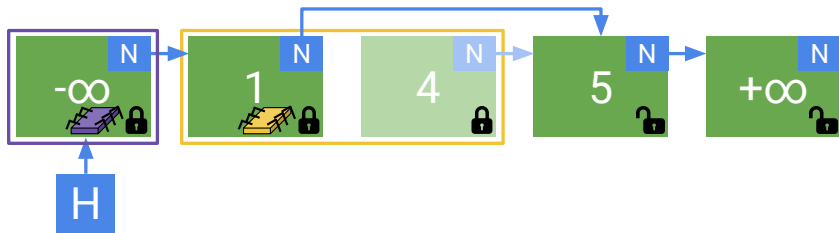
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый удаляет «4»

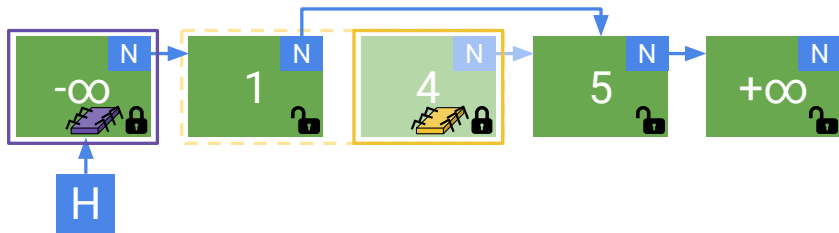
# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»

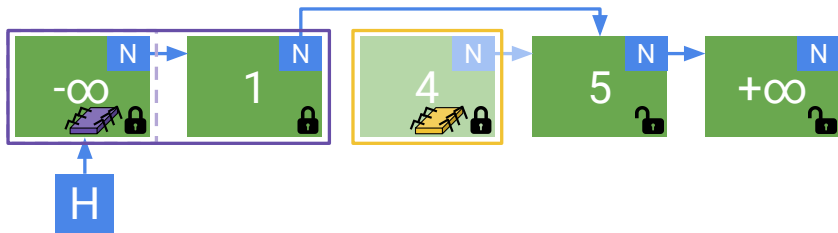


Жёлтый отпускает блокировку на «1»



# Пример

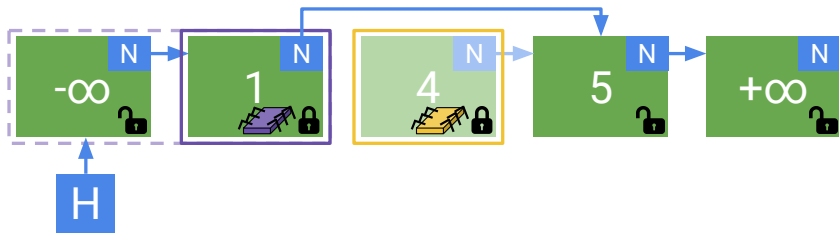
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на «1» ...

# Пример

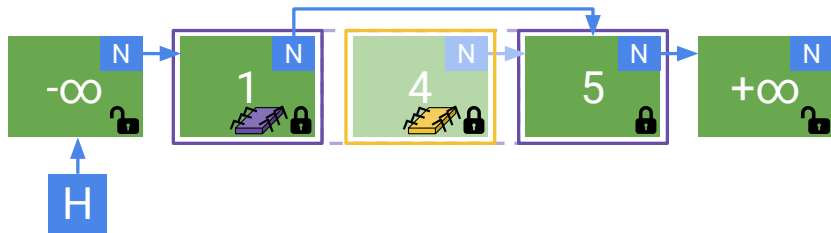
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и отпускает блокировку на голову списка

# Пример

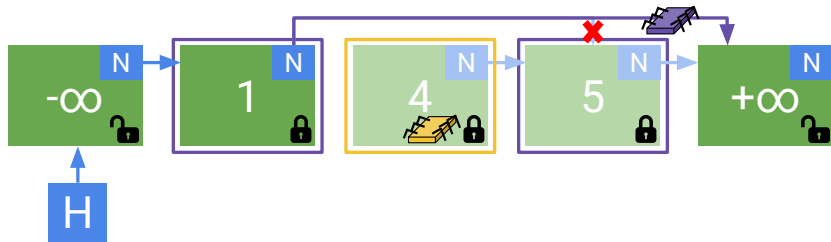
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на «5», нашёл окно.

# Пример

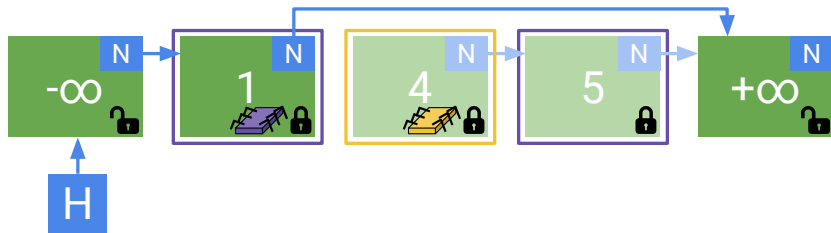
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый удаляет «5» ...

# Пример

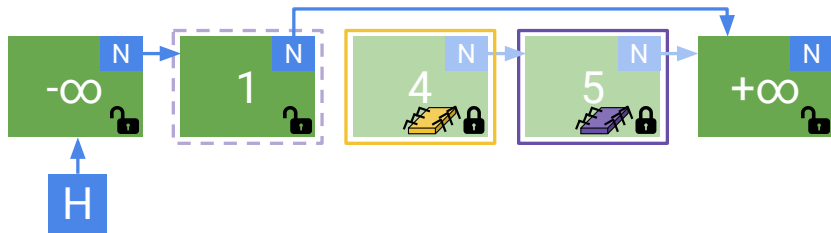
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый удаляет «5» ...

# Пример

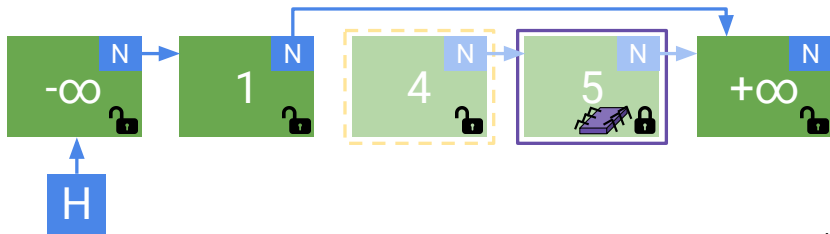
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и отпускает блокировку на «1»

# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



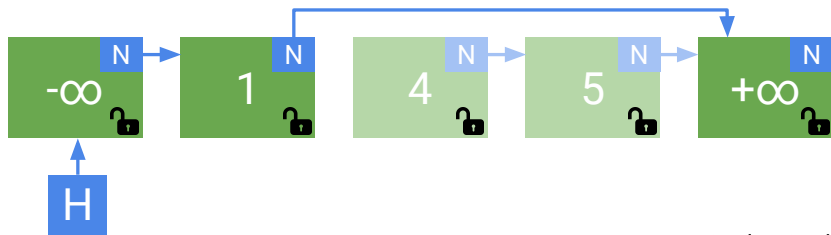
Жёлтый отпускает блокировку на «4»





# Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



Оба элемента удалены корректно

# Тонкая синхронизация: псевдокод

```
(Node, Node) findWindow(key) {  
    var cur = head; cur.lock()  
    var next = cur.N; next.lock()  
    while (next.key < key):  
        cur.unlock(); cur = next  
        next = cur.N; next.lock()  
    return (cur, next)  
}
```

```
boolean contains(key) {  
    (cur, next) = findWindow(key)  
    val res = next.key == key  
    cur.unlock(); next.unlock()  
    return res  
}
```

Остальные операции аналогично

# Корректность

- Поиск окна: запись и чтение  $\text{cur}.N$  не могут происходить параллельно
- Модификация: во время изменения окно защищено блокировками  $\Rightarrow$  атомарно
- $\forall k$  : операции с ключом  $k$  линейризуемы  $\Rightarrow$  всё исполнение линейризуемо
- Операции с ключом  $k$  упорядочены взятием соответствующей блокировки

# План

1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Алгоритм абстрактной операции

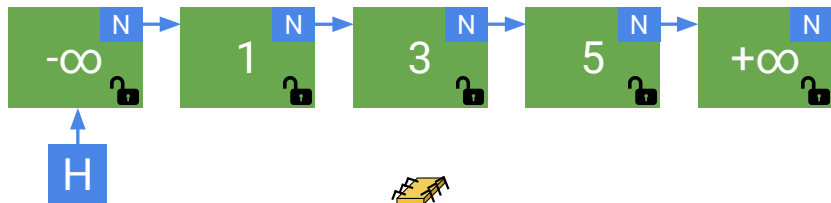
1. Найти окно (`cur`, `next`) без синхронизации
2. Взять блокировки на `cur` и `next`
3. Проверить инвариант `cur.N = next`
4. Проверить, что `cur` не удалён
5. Выполнить операцию (добавить, удалить, ...)
6. При любой ошибке начать заново

## Проверка, что узел не удалён

- Как проверить, что `cur` не удалён?
- Держим блокировку на `cur` и `cur` удален  $\Rightarrow$  не увидим `cur` при проходе
- Попробуем найти `cur` ещё раз за  $O(n)$  и проверим, что `cur.N = next`

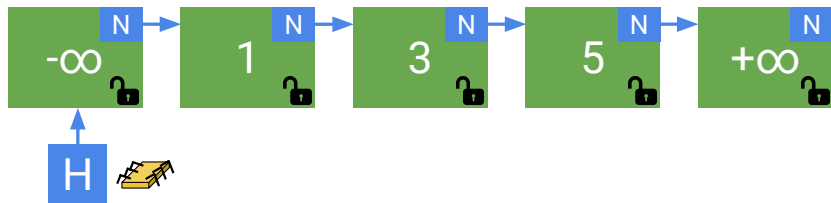
# Пример успешной операции

Жёлтый добавляет «4»



# Пример успешной операции

Жёлтый добавляет «4»

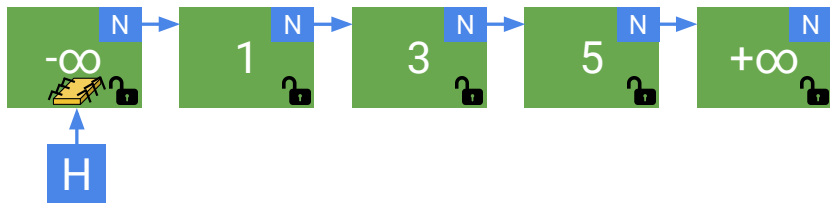


Жёлтый ищет окно



# Пример успешной операции

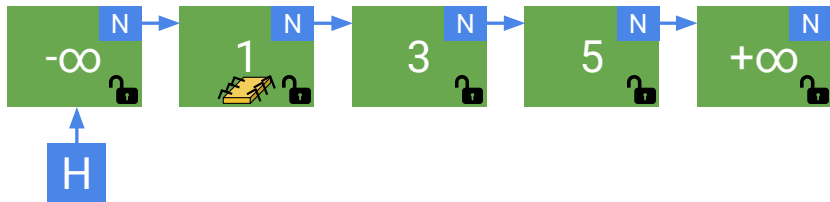
Жёлтый добавляет «4»



Жёлтый ищет окно

# Пример успешной операции

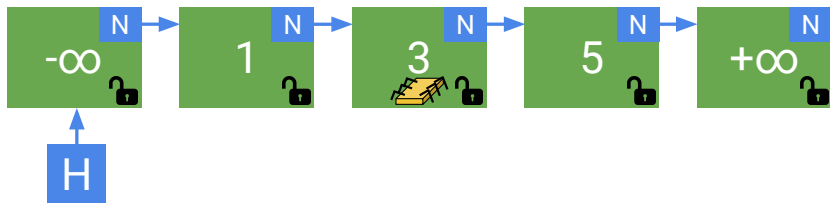
Жёлтый добавляет «4»



Жёлтый ищет окно

# Пример успешной операции

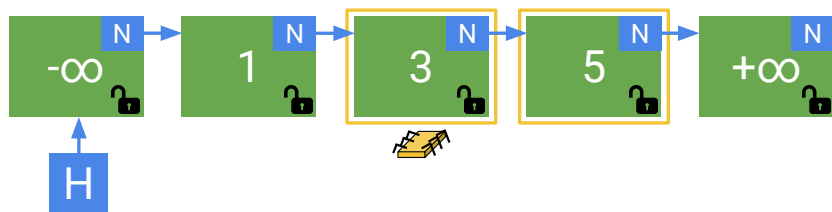
Жёлтый добавляет «4»



Жёлтый ищет окно

# Пример успешной операции

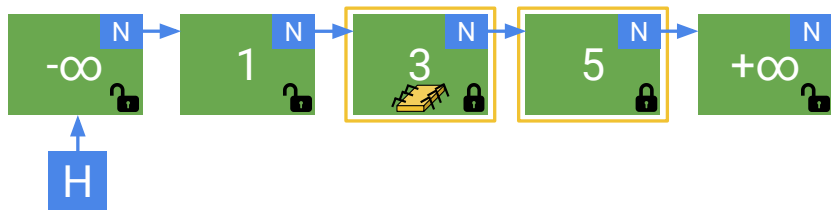
Жёлтый добавляет «4»



Нашёл окно

# Пример успешной операции

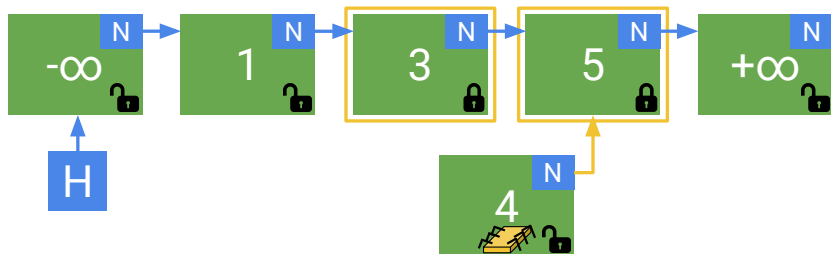
Жёлтый добавляет «4»



Берёт блокировки

# Пример успешной операции

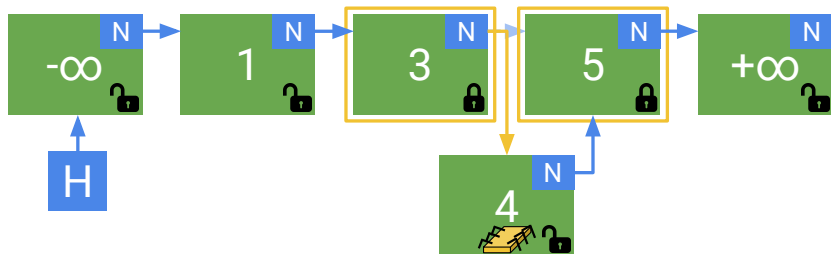
Жёлтый добавляет «4»



Добавляет узел «4»

# Пример успешной операции

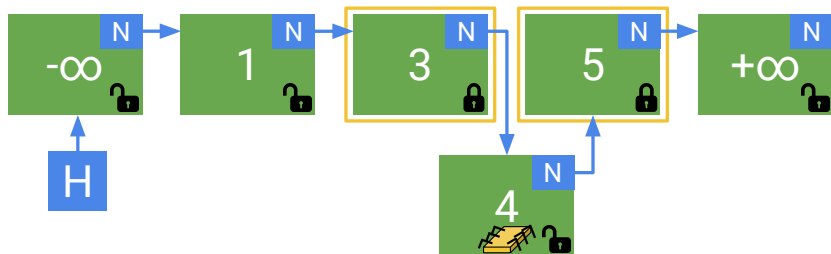
Жёлтый добавляет «4»



Добавляет узел «4»

# Пример успешной операции

Жёлтый добавляет «4»

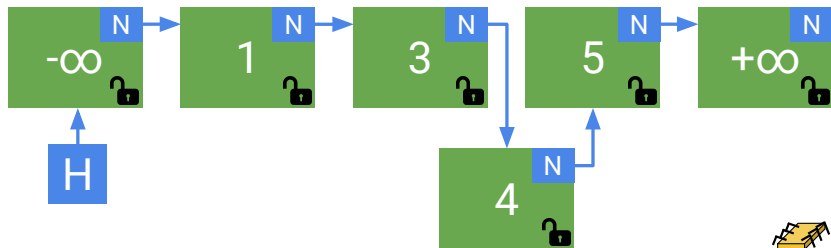


Добавляет узел «4»



# Пример успешной операции

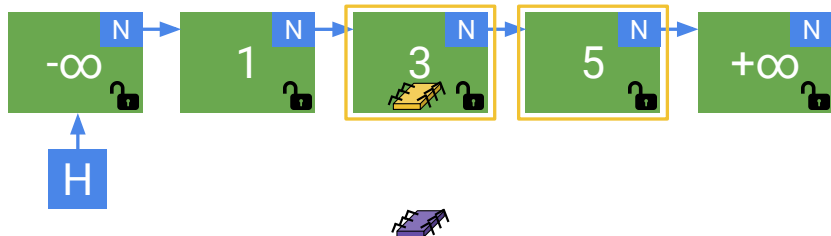
Жёлтый добавляет «4»



Отпускает блокировки и уходит

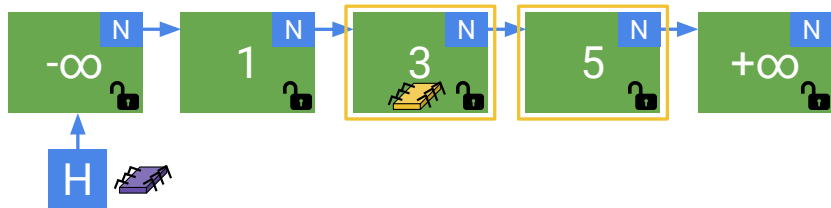
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



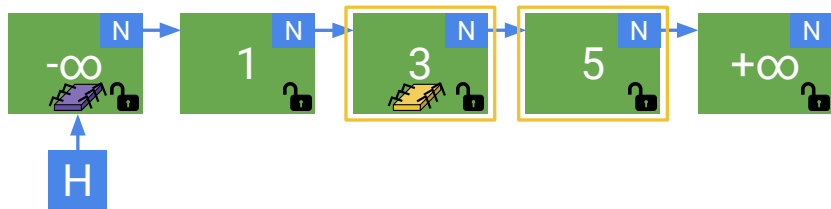
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



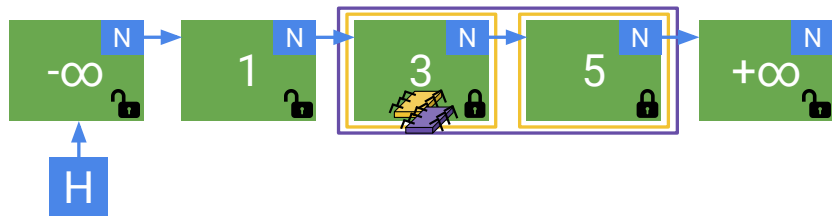
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



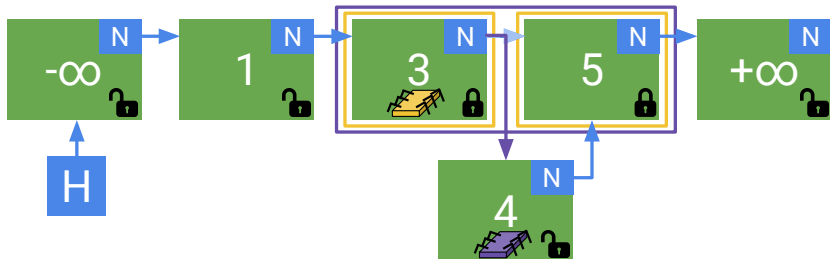
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



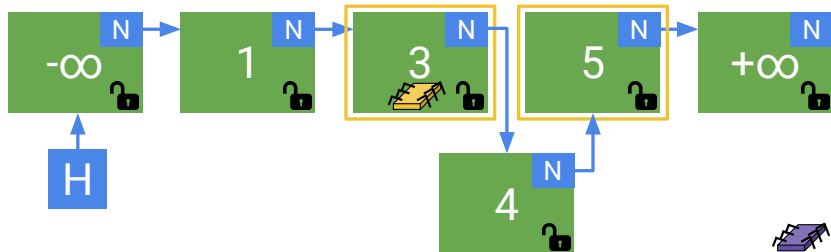
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



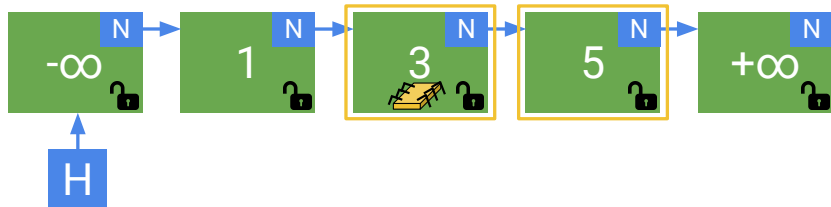
Проблема:  $\text{cur.N} \neq \text{next}$

Оба добавляют «4»



# Проблема: cur уже удалили

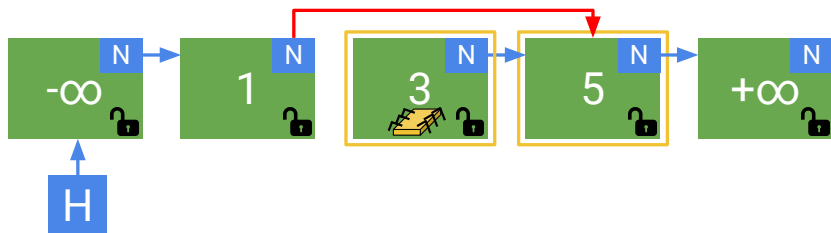
Пока жёлтый «тормозил», cur уже удалили





# Проблема: cur уже удалили

Пока жёлтый «тормозил», cur уже удалили



# Оптимистичный поиск: псевдокод

```
data class Node(  
  @Volatile var N: Node,  
  val key: Int  
)  
  
(Node, Node) findWindow(key) {  
  var cur = head  
  var next = cur.N  
  while (next.key < key):  
    cur = next  
    next = cur.N  
  return (cur, next)  
}
```

```
boolean contains(key) {  
  while (true):  
    (cur, next) = findWindow(key)  
    cur.lock(); next.lock()  
    if (!validate(cur, next)):  
      (cur, next).unlock(); continue  
    val res = next.key == key  
    cur.unlock(); next.unlock()  
    return res  
}
```

# Оптимистичный поиск: псевдокод

```
boolean validate(cur, next) {  
    var node = head  
    while(node.key < cur.key):  
        node = node.N  
    return (cur, next) == (node, node.N)  
}
```

# Корректность

- Поиск: запись и чтение  $cur.N$  связаны отношением «произошло до»
- Можем говорить о линеаризуемости операций над одинаковыми ключами
- Точка линеаризации - взятие блокировки на  $cur$

# План

1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Ленивое удаление: идея

- Добавим в Node поле `boolean removed`
- Удаление в две фазы:
  1. `node.removed = true` - логическое удаление
  2. Физическое удаление из списка

# Ленивое удаление: идея

- Добавим в Node поле `boolean removed`
- Удаление в две фазы:
  1. `node.removed = true` - логическое удаление
  2. Физическое удаление из списка
- Инвариант: все неудаленные вершины в списке
- $\Rightarrow$  теперь не надо проходить по списку в `validate()`

# Ленивое удаление: псевдокод

```
boolean validate(cur, next) {  
    return !cur.removed &&  
           !next.removed &&  
           cur.N == next  
}
```



# План

1. Множество на односвязном списке
2. Грубая синхронизация
3. Тонкая синхронизация
4. Оптимистичная синхронизация
5. Ленивая синхронизация
6. Неблокирующая синхронизация

# Неблокирующий поиск

- Поле  $N$  - volatile
- $\Rightarrow$  на момент чтения поля  $N$  видим состояние на момент записи  $N$  как минимум
- $\Rightarrow$  можем не брать блокировку при поиске

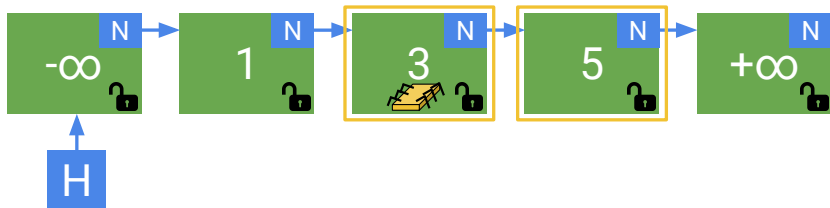
# Неблокирующий поиск

- Поле  $N$  - volatile
- $\Rightarrow$  на момент чтения поля  $N$  видим состояние на момент записи  $N$  как минимум
- $\Rightarrow$  можем не брать блокировку при поиске

```
boolean contains(key) {  
    (cur, next) = findWindow(key)  
    return next.key == key  
}
```

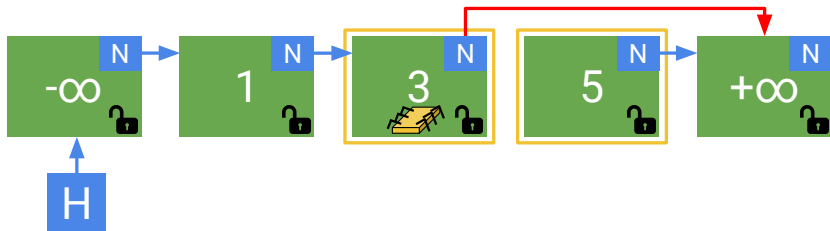
# Пример

Жёлтый ищет «5», но его удаляют параллельно



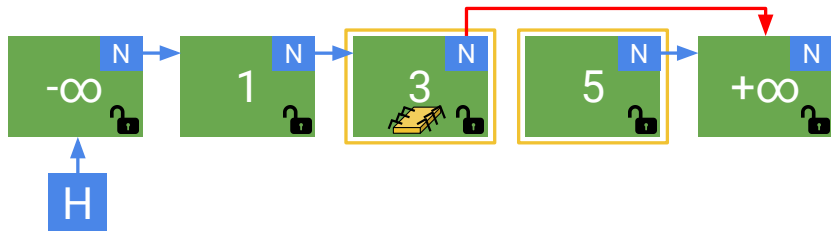
# Пример

Жёлтый ищет «5», но его удаляют параллельно



# Пример

Жёлтый ищет «5», но его удаляют параллельно



Выполняются параллельно  
 $\Rightarrow$  можем упорядочить как угодно

# Compare-and-set

```
class AtomicReg<T> {  
    var x: T  
  
    boolean CAS(expected: T, value: T) {  
        do atomically:  
            val old = x  
            if (old == expected):  
                x = value  
                return true  
            return false  
    }  
}
```

В Java для этого используются AtomicReference, AtomicInteger, AtomicLong, ...

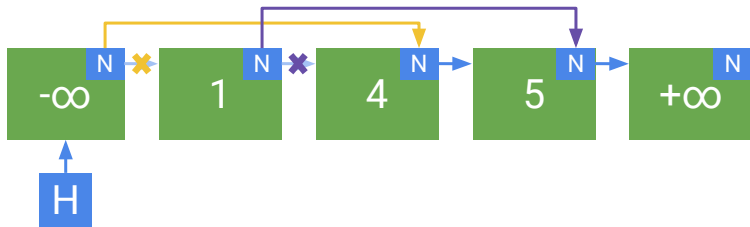
# Неблокирующая модификация

- Вернёмся к множеству на односвязном списке



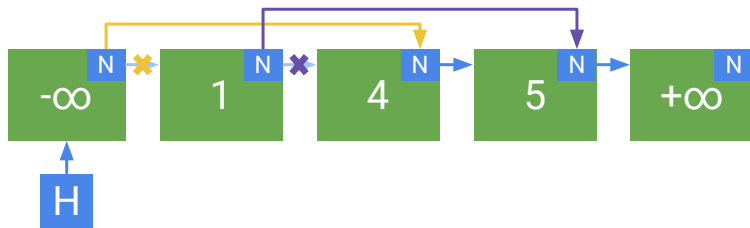
# Неблокирующая модификация

- Вернёмся к множеству на односвязном списке
- Просто CAS недостаточно, не работает remove



# Неблокирующая модификация

- Вернёмся к множеству на односвязном списке
- Просто CAS недостаточно, не работает remove



Всё оттого, что мы не знали, что «1» уже удалили

# Неблокирующая модификация

- Объединим `N` и `removed` в одну переменную, пару `(N, removed)`
- Будем менять `(N, removed)` атомарно
- Каждая операция модификации будет выполняться одним успешным CAS-ом
- В Java для этого есть `AtomicMarkableReference`

- AtomicMarkableReference работает как volatile

```
data class Node(  
    // содержит ссылку на след. узел  
    // и наш флаг removed  
    var N: AtomicMarkableReference,  
    val key: Int  
)
```

# Поиск окна

```
(Node, Node) findWindow(key) {  
  retry: while(true):  
    var cur = head, next = cur.N  
    boolean[] removed = new boolean[1]  
    while (next.key < key):  
      val node = next.N.get(removed)  
      if (removed[0]):  
        // удалим физически  
        if (!cur.N.CAS(next, node, false, false)):  
          continue retry  
      next = node  
    else:  
      cur = next  
      next = cur.N  
    // тут еще проверка, что next не удален  
    return (cur, next)  
}
```

# Поиск

```
boolean contains(key) {  
    (cur, next) = findWindow(key)  
    return next.key == key  
}
```

Поиск может не удалять узлы физически

# Добавление

```
void add(key) {  
  while(true):  
    (cur, next) = findWindow(key)  
    if (next.key == key):  
      return  
    val node = Node(key, next)  
    if (cur.N.CAS(next, node, false, false)):  
      return  
}
```

# Удаление

```
void remove(key) {  
  while(true):  
    (cur, next) = findWindow(key)  
    if (next.key != key)  
      return // false  
    val node = next.N.getReference();  
    if (next.N.CAS(node, node, false, true)):  
      // помогаем findWindow удалить физически  
      cur.N.CAS(next, node, false, false)  
    return // true  
}
```



Спасибо за внимание!