Построение многопоточных алгоритмов с различными типами синхронизации

Никита Коваль

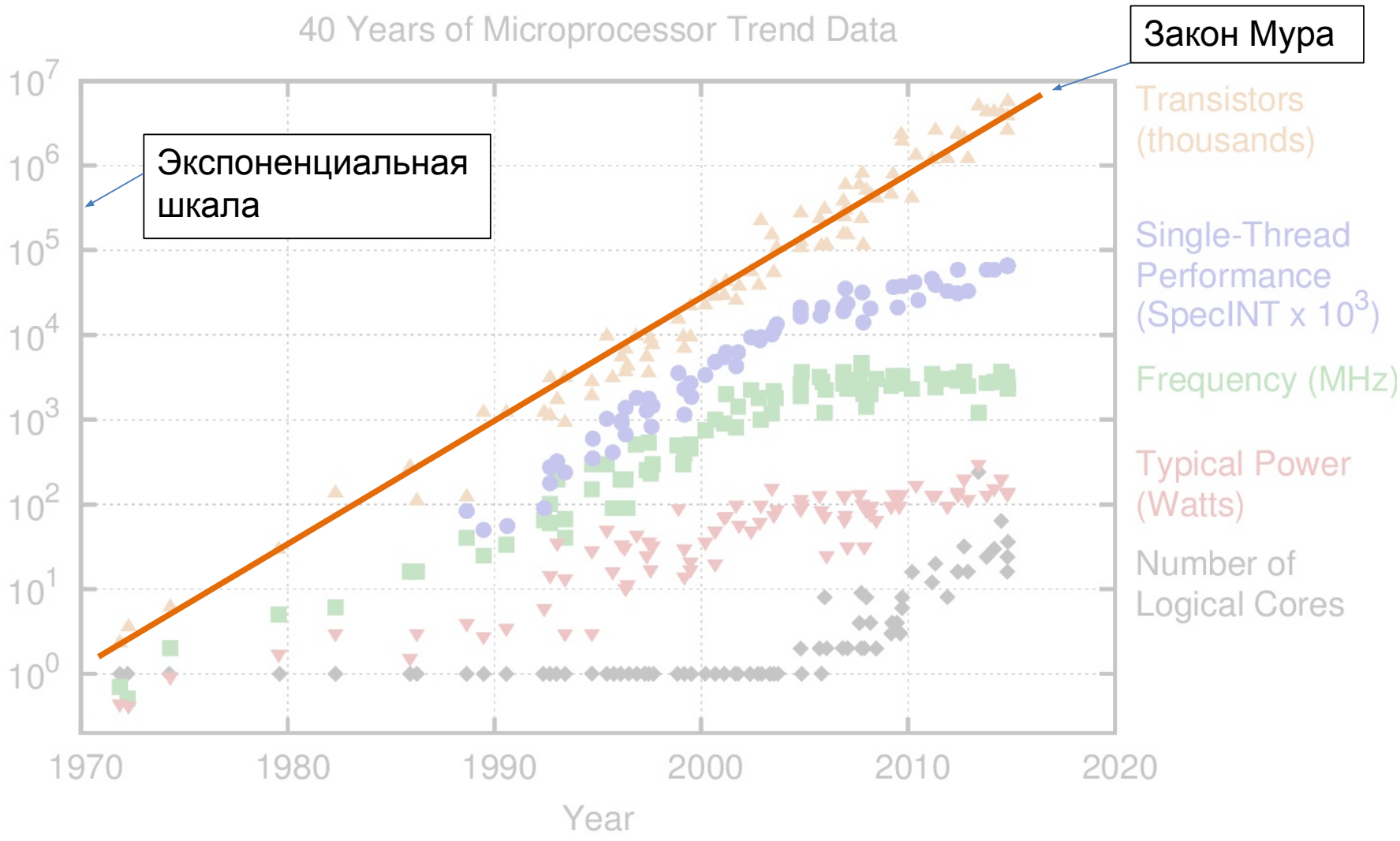
Research engineer, Devexperts
ndkoval@ya.ru

Пермь, 2017

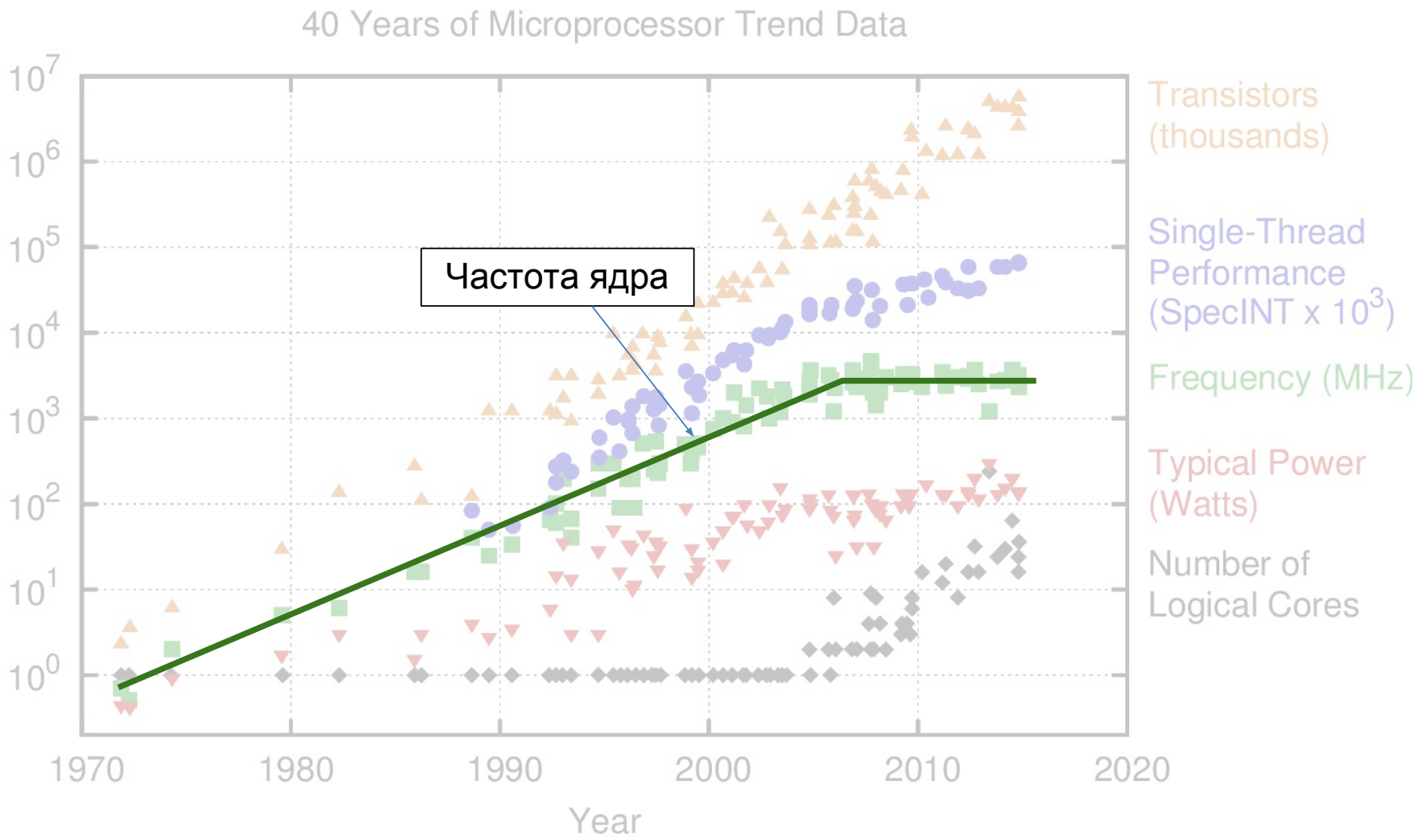
План

1. Мотивация
2. Блокировки
3. Множество на односвязном списке
4. Грубая синхронизация
5. Тонкая синхронизация
6. Оптимистичная синхронизация
7. Ленивая синхронизация
8. Неблокирующая синхронизация
9. Подведём итоги

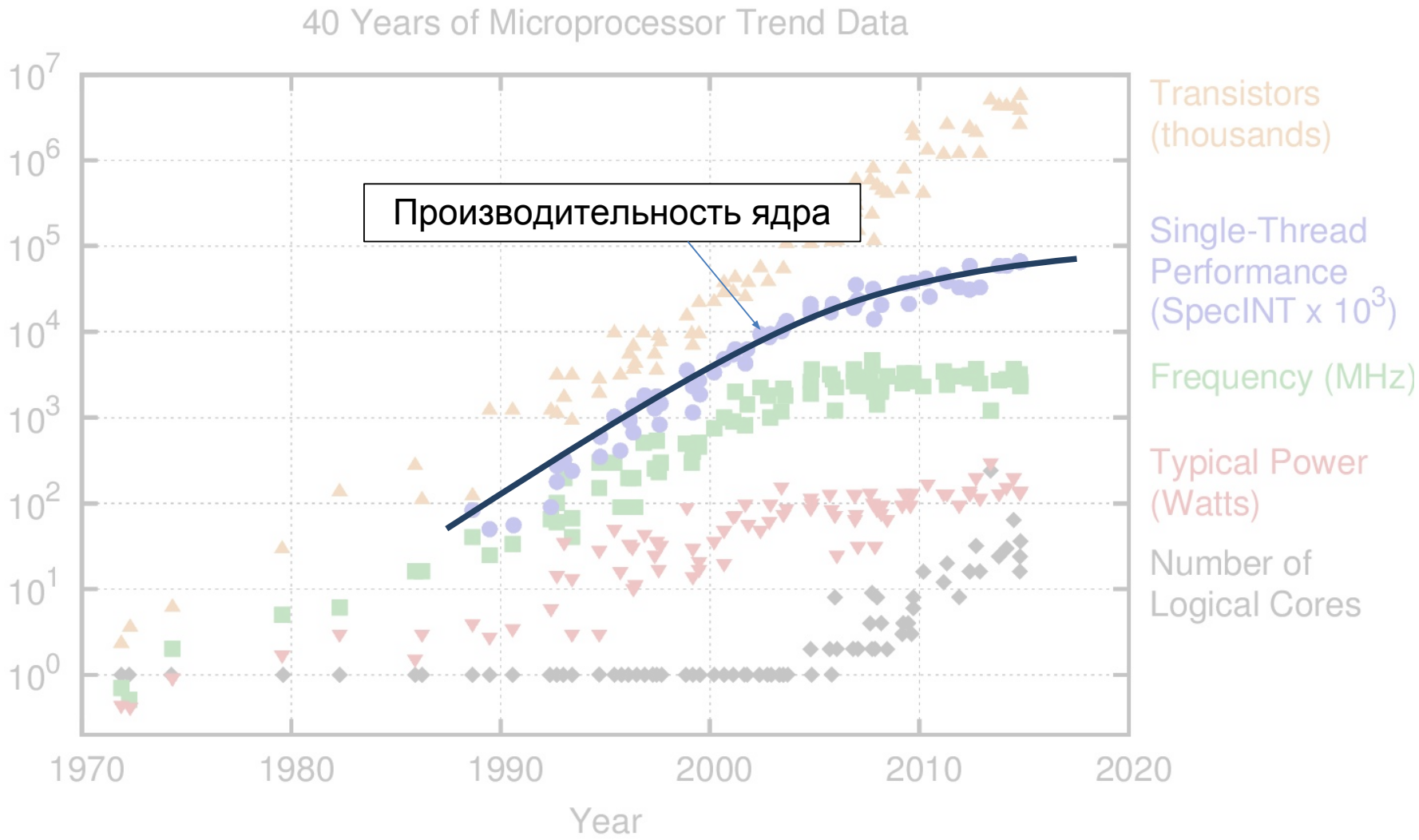
Закон Мура и "The free lunch is over"



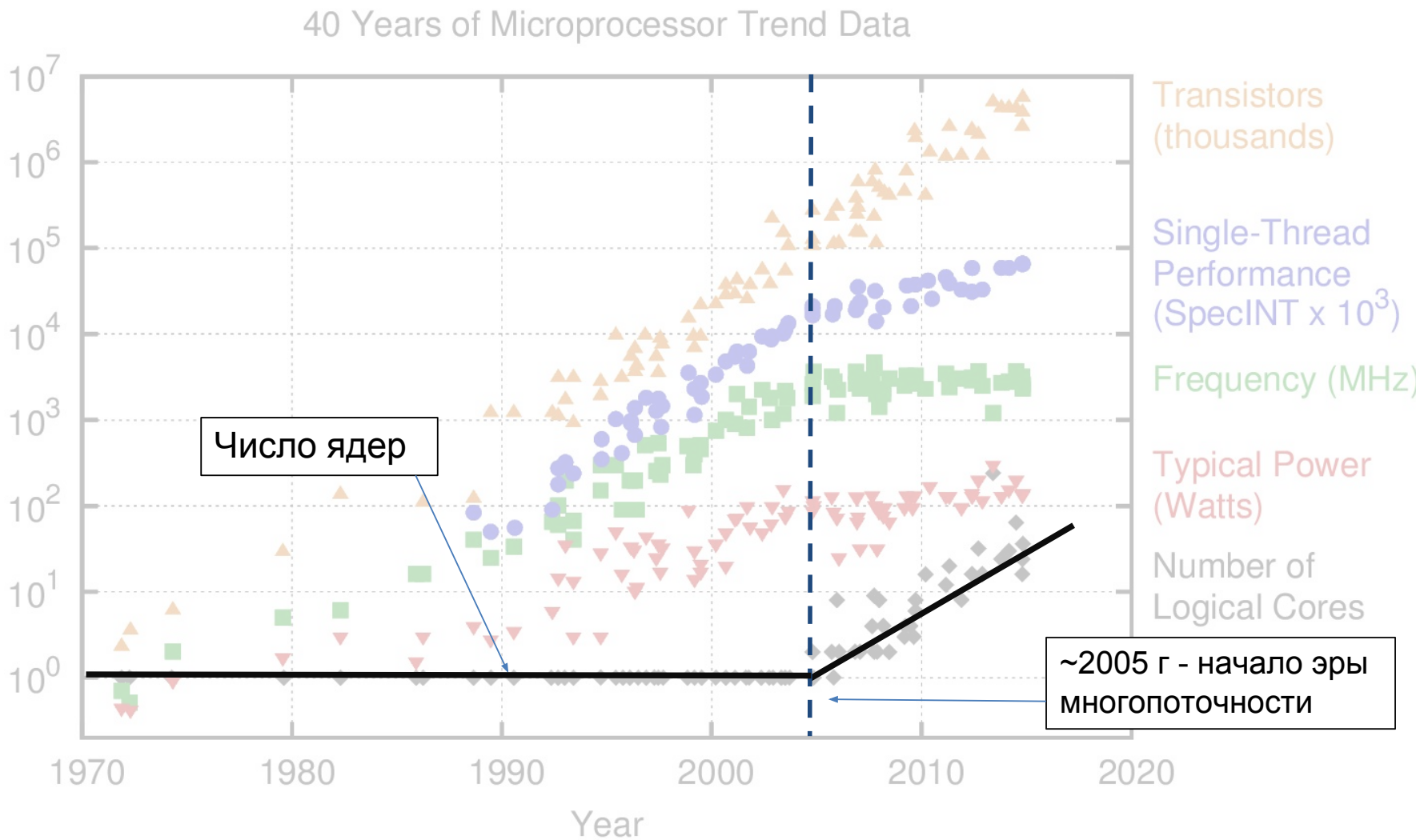
Закон Мура и “The free lunch is over”



Закон Мура и “The free lunch is over”



Закон Мура и “The free lunch is over”



Закон Амдала

- S - ускорение программы
- N - количество ядер
- P - доля параллельного кода

$$\text{Ускорение } S = \frac{1}{1-P+\frac{P}{N}}$$

- $1 - P$ - последовательная часть
- $\frac{P}{N}$ - параллельная часть

Закон Амдала: предел

Количество ядер $N \to \infty$

Макс. ускорение $S = \frac{1}{1-P}$

Закон Амдала: предел

Количество ядер $N \to \infty$
Доля параллельного кода $P = 60\%$

Макс. ускорение $S = \frac{1}{1-P} = 2.5$

Закон Амдала: предел

Количество ядер $N \to \infty$
Доля параллельного кода $P = 95\%$

Макс. ускорение $S = \frac{1}{1-P} = 20$

Закон Амдала: предел

Количество ядер $N \to \infty$
Доля параллельного кода $P = 99\%$

Макс. ускорение $S = \frac{1}{1-P} = 100$

Для масштабируемости нужно больше **параллелизма**

План

1. Мотивация
2. Блокировки
3. Множество на односвязном списке
4. Грубая синхронизация
5. Тонкая синхронизация
6. Оптимистичная синхронизация
7. Ленивая синхронизация
8. Неблокирующая синхронизация
9. Подведём итоги

Построение линеаризуемых объектов

```
class Bank {
  int[] amounts = new int[N]

  void deposit(int id, int amount) { amounts[id] += amount }

  void transfer(int idFrom, int idTo, int amount) {
    amounts[idFrom] -= amount
    amounts[idTo] += amount
  }

  int getAmount(int id) { return amounts[id] }
}
```

Вопрос дня

Как построить корректный многопоточный объект из последовательного?

Взаимное исключение

Защитим операцию специальным объектом **блокировка** (lock)

```
l.lock();
  // Critical section: only
  // one thread can be here
l.unlock()
```

Взаимное исключение

Защитим операцию специальным объектом **блокировка** (lock)

```
l.lock();
  // Critical section: only
  // one thread can be here
l.unlock()
```

Защитив все операции, будет работать как последовательный код

Грубая блокировка

Получаем код:

```
class Bank {
  int[] amounts = new int[N]

  synchronized void deposit(int id, int amount) { amounts[id] += amount }

  synchronized void transfer(int idFrom, int idTo, int amount) {
    amounts[idFrom] -= amount
    amounts[idTo] += amount
  }

  synchronized int getAmount(int id) { return amounts[id] }
}
```

Такая блокировка называется **грубой**

Тонкая блокировка: идея

Будем блокировать не весь объект целиком, а только **отдельные необходимые** части

Тонкая блокировка: попытка

```
void deposit(int id, int amount) {
  locks[id].lock()
    amounts[id] += amount
  locks[id].unlock;
}

void transfer(int idFrom, int idTo, int amount) {
  locks[idFrom].lock()
    amounts[idFrom] -= amount
  locks[idFrom].unlock()
  locks[idTo].lock()
    amounts[idTo] += amount
  locks[idTo].unlock()
}
```

Тонкая блокировка: запустим

Bank bank = new Bank(N)

bank.transfer(a, b, 100)	bank.get(a) bank.get(b)

Тонкая блокировка: запустим

Bank bank = new Bank(N)

bank.transfer(a, b, 100):	bank.get(a):
1: locks[a].lock()	4: locks[a].lock()
2: amounts[a] -= 100	5: return amounts[a] // -100
3: unlock[a].unlock()	6: unlock[a].unlock()
	bank.get(b)
	7: locks[b].lock()
	8: return amounts[b] // 0
	9: unlock[b].unlock()
10 ...	

Двухфазная блокировка

- Каждому объекту сопоставлена своя блокировка
- Алгоритм 2-Phase Locking:
  1. Взять блокировки на необходимые объекты
  2. Выполнить операцию
  3. Отпустить блокировки
- Брать и отпускать блокировки можно в любом порядке

Тонкая блокировка: исправляем

```
void deposit(int id, int amount) {
  locks[id].lock()
    amounts[id] += amount
  locks[id].unlock;
}

void transfer(int idFrom, int idTo, int amount) {
  // Lock phase
  locks[idFrom].lock()
  locks[idTo].lock()
    amounts[idFrom] -= amount
    amounts[idTo] += amount
  // Unlock phase
  locks[idFrom].unlock()
  locks[idTo].unlock()
}
```

Проблема: deadlock

Что будет, если параллельно выполнять transfer(to, from, ...) и transfer(from, to, ...)?

Проблема: deadlock

Что будет, если параллельно выполнять transfer(to, from, ...) и transfer(from, to, ...)?

Bank bank = new Bank(N)

bank.transfer(a, b, ...):	bank.transfer(b, a, ...):
a.lock()	
	b.lock()
b.lock()	a.lock()
...	...

Иерархия блокировок

- Упорядочим блокировки каким-то способом (например, по id объекта)

Иерархия блокировок

- Упорядочим блокировки каким-то способом (например, по id объекта)
- Будем брать блокировки только в этом порядке
- $\Rightarrow$ не будет «встречных» взятий блокировок

Использование блокировок

Любой объект можно сделать корректным:

- Грубая блокировка
  - Блокируем всю операцию целиком
- Тонкая блокировка
  - Блокируем только операции над отдельными внутренними объектами
  - Используем двухфазную блокировку, чтобы обеспечить корректность
  - Соблюдаем иерархию блокировок, чтобы не попасть в дедлок

Задание «bank»: необходимо сделать реализацию банка корректной с помощью тонкой блокировки.

План

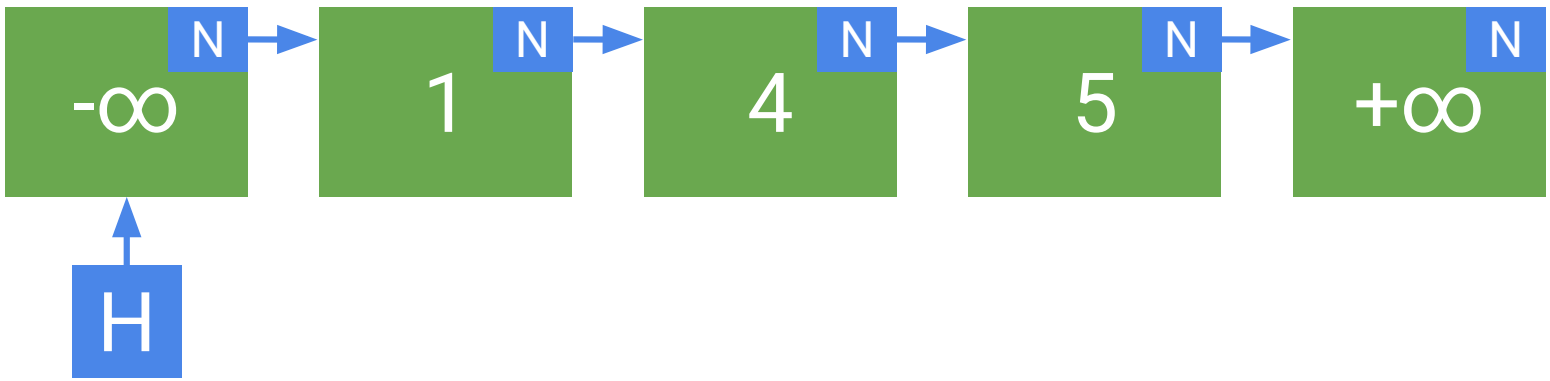
1. Мотивация
2. Блокировки
3. Множество на односвязном списке
4. Грубая синхронизация
5. Тонкая синхронизация
6. Оптимистичная синхронизация
7. Ленивая синхронизация
8. Неблокирующая синхронизация
9. Подведём итоги

Множество

```
interface Set {

  // Adds the specified key to this set if it is not already present.
  void add(int key);

  // Returns true if this set contains the specified key
  boolean contains(int key);

  // Removes the specified element from this set if it is present.
  void remove(int key);
}
```

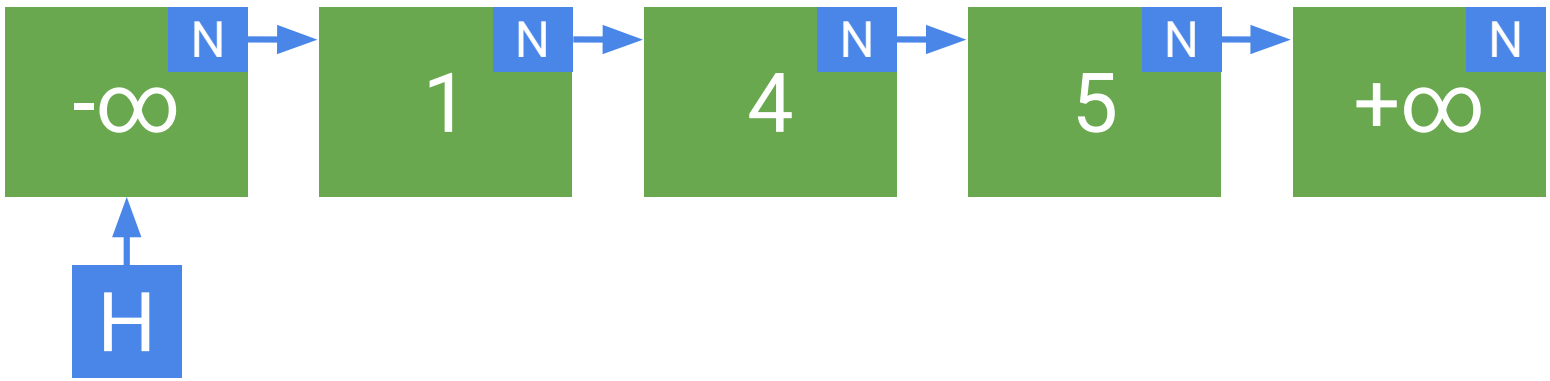
Односвязный список

Элементы упорядочены по возрастанию

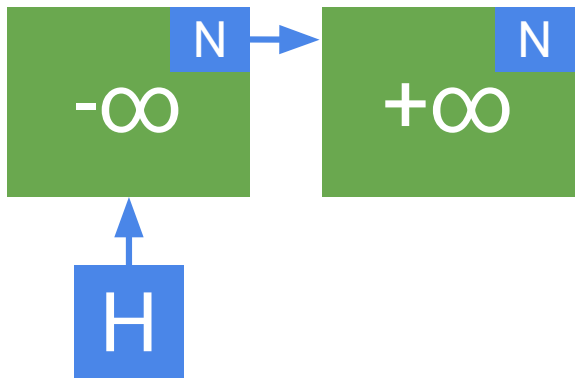


Односвязный список

Элементы упорядочены по возрастанию



Пустой список состоит из двух граничных элементов



Односвязный список: алгоритм

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

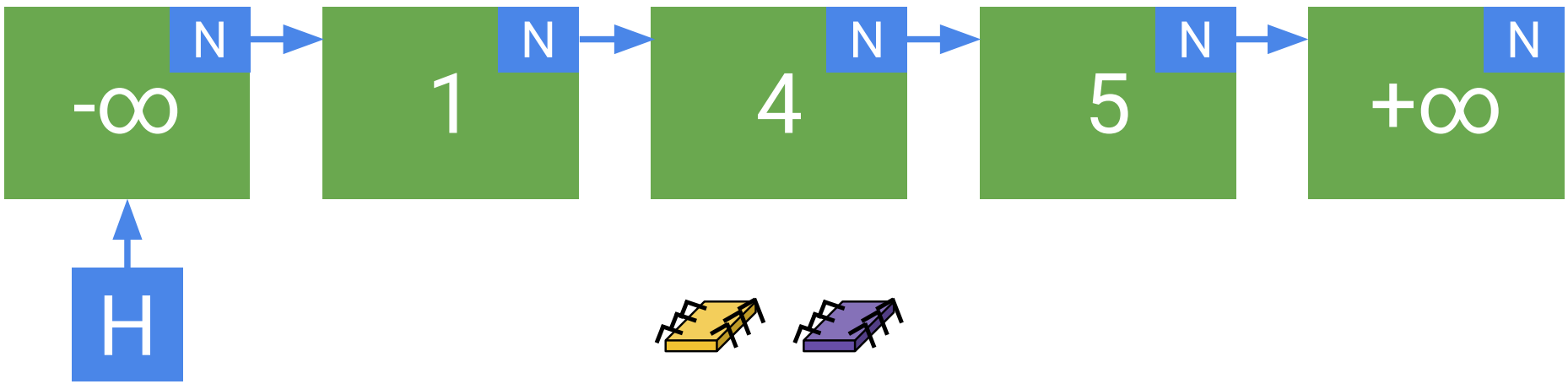
Односвязный список: псевдокод

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

val head = Node(−∞, Node(∞, null))

(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) {
  (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;
}
```

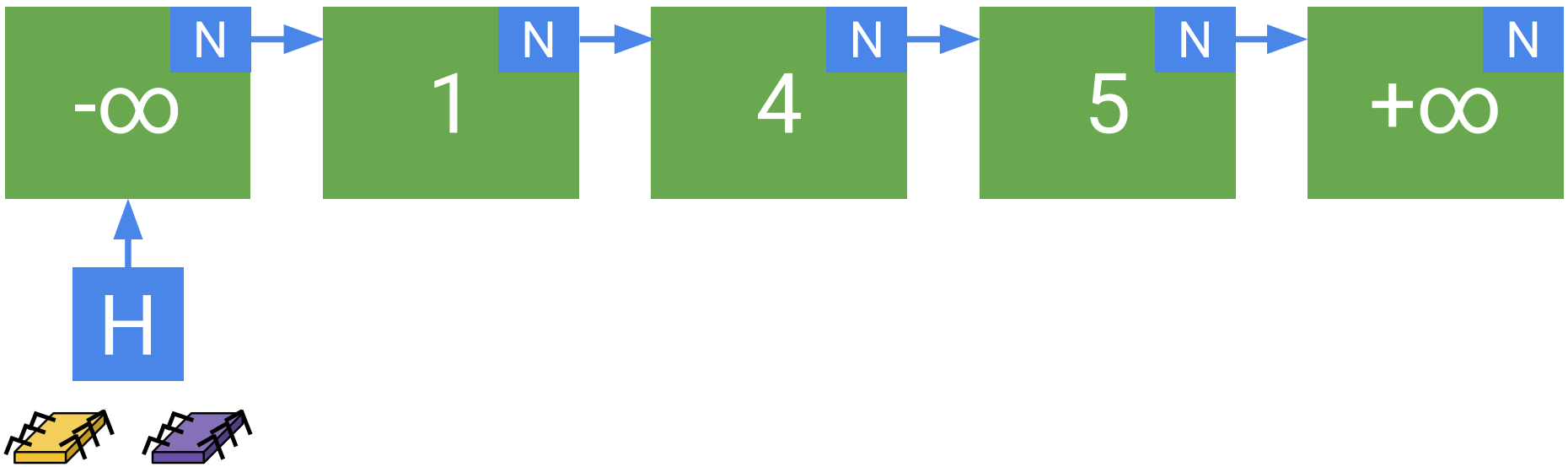
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



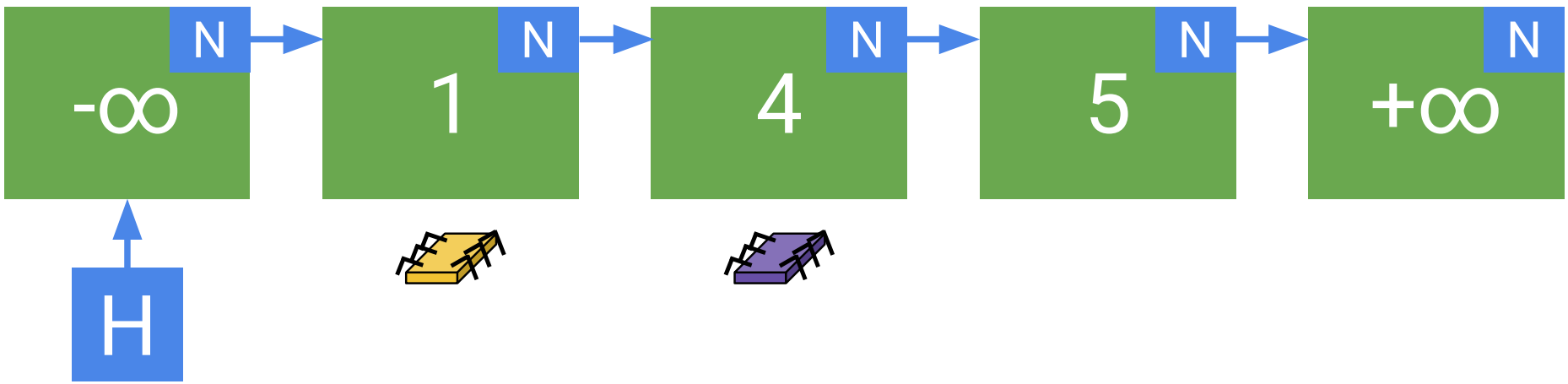
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



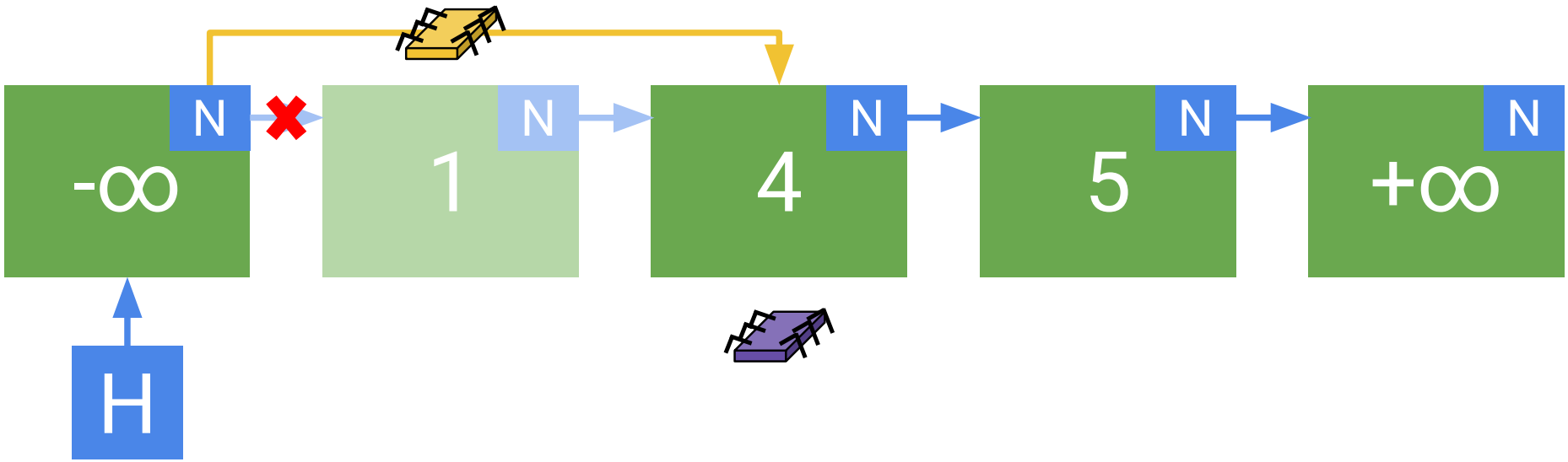
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



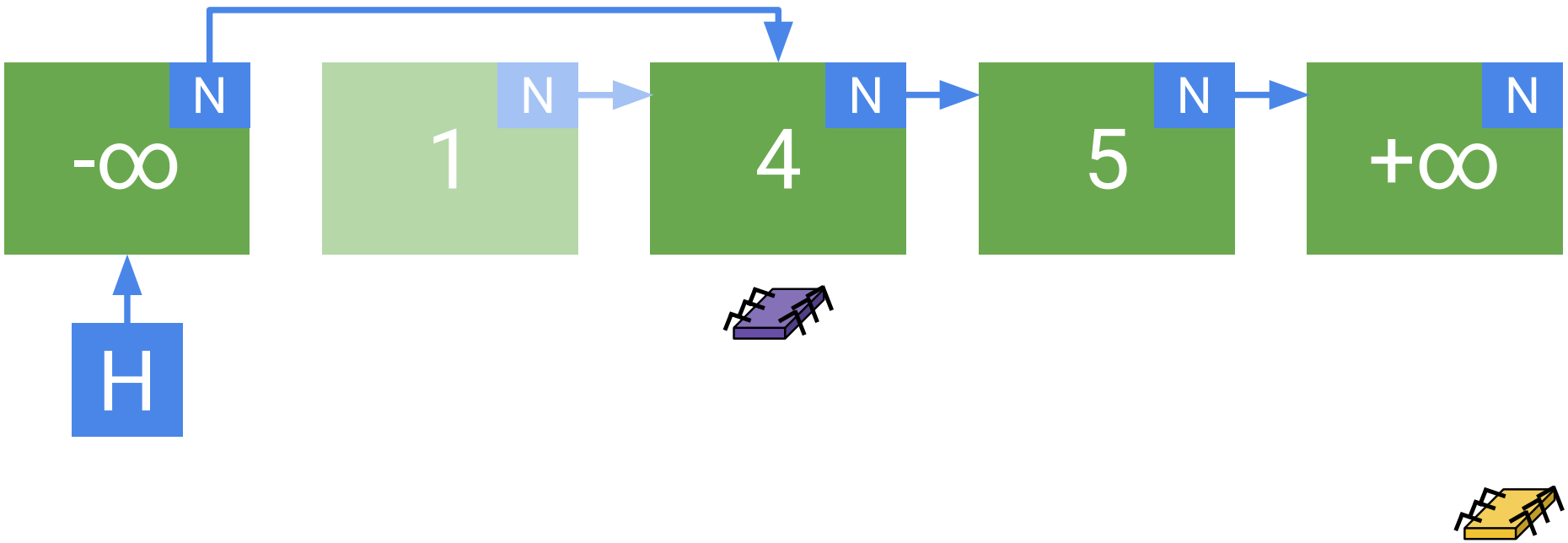
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



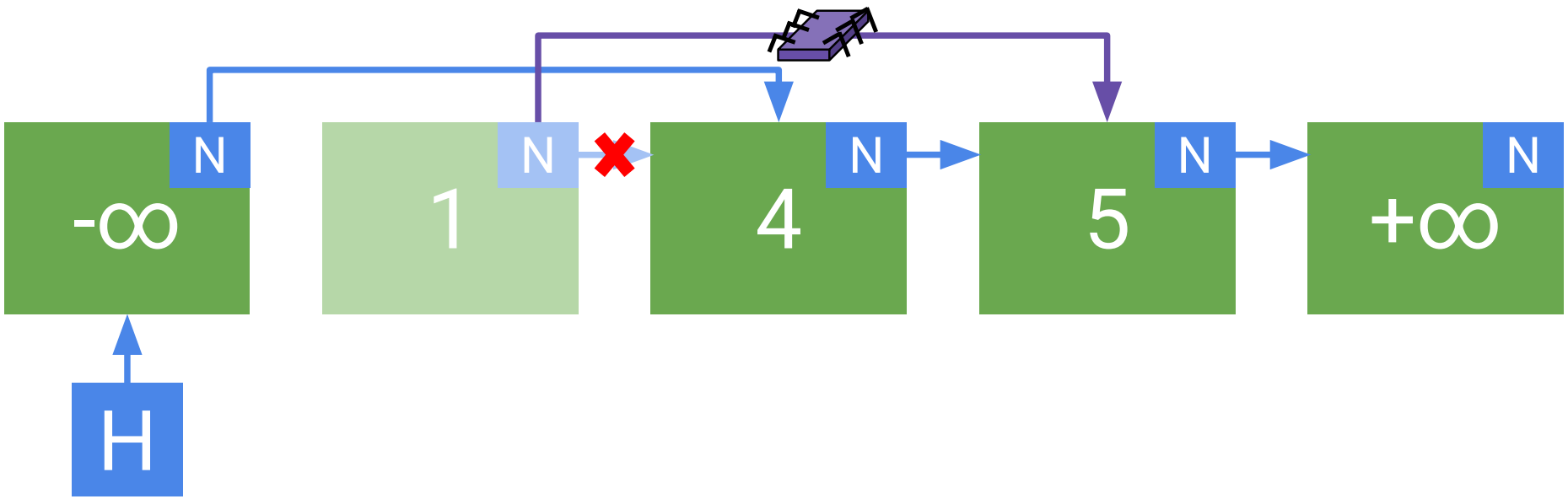
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



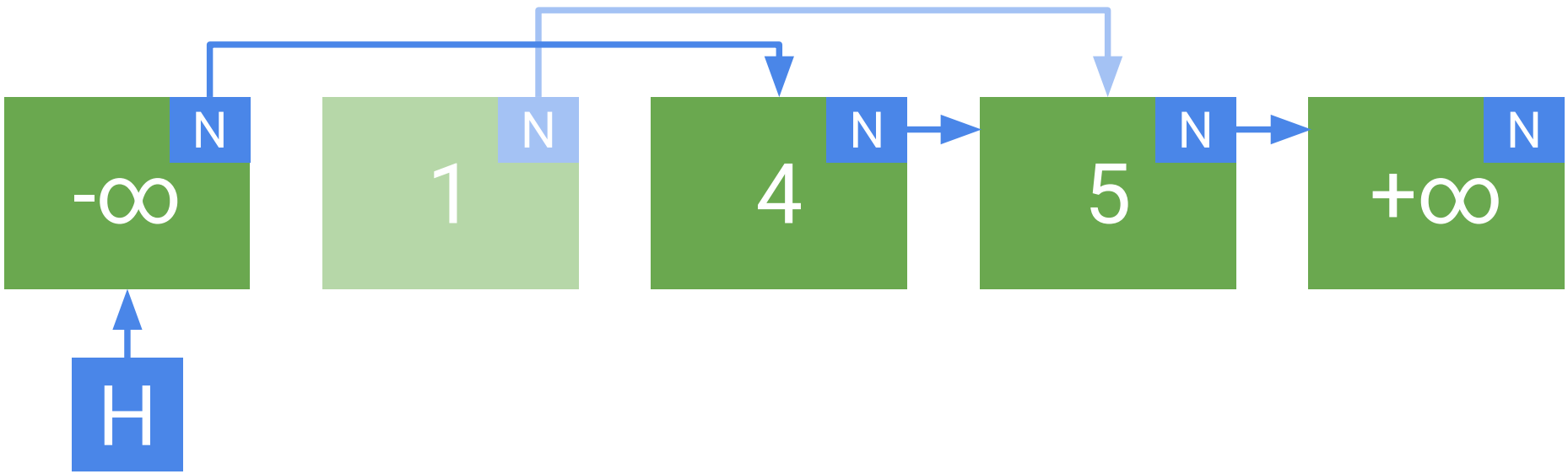
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



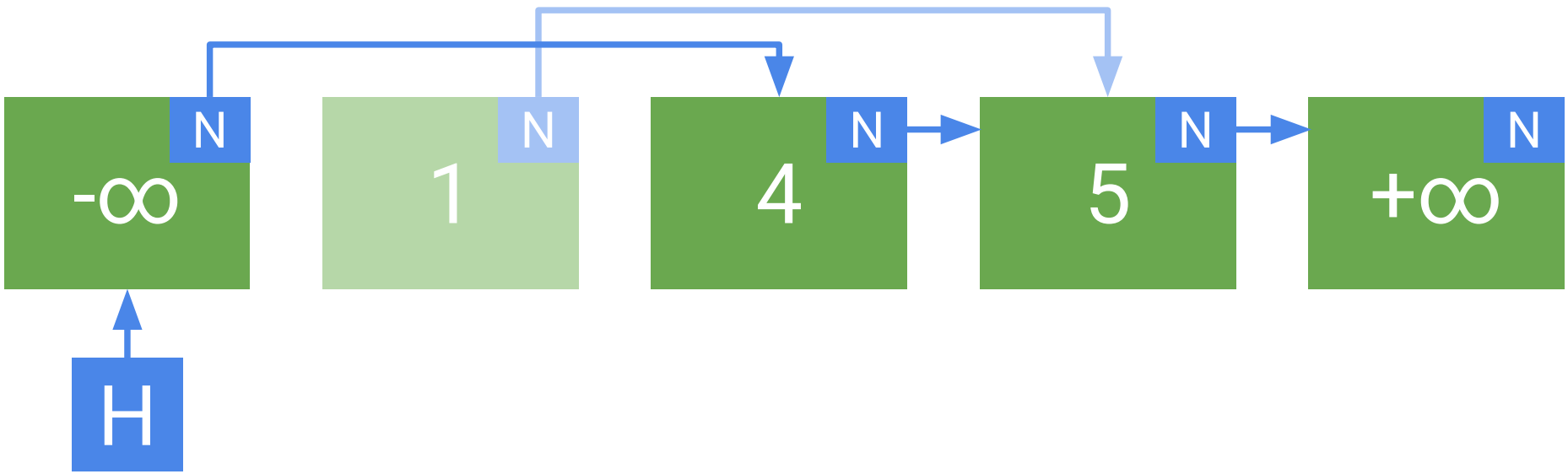
Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



Проблема

Жёлтый удаляет «1», фиолетовый удаляет «4»



У фиолетового ничего не вышло!

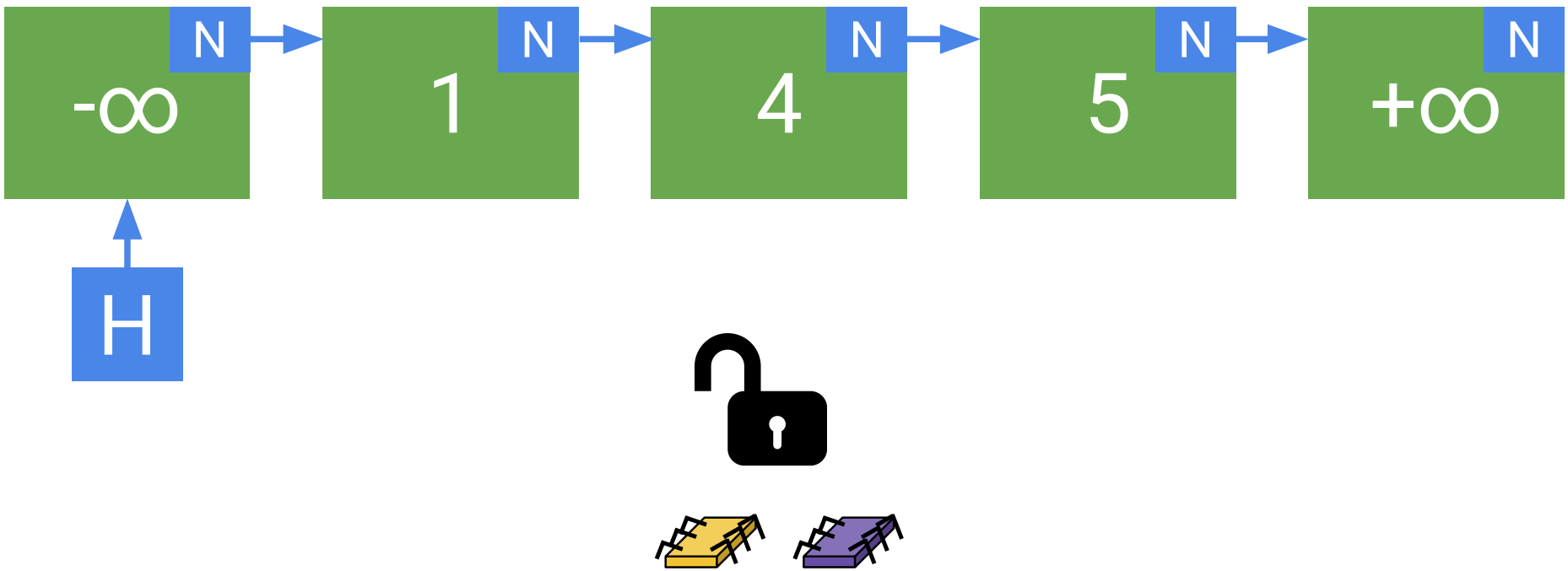
План

Грубая синхронизация

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

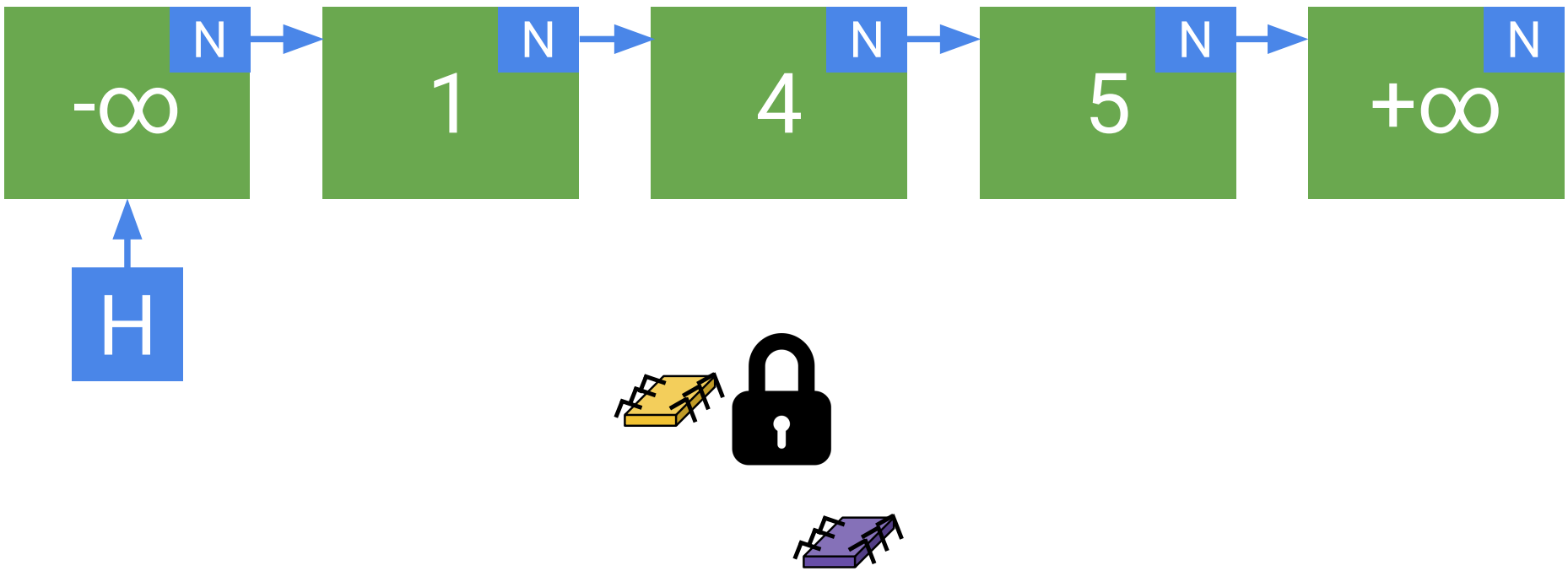
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



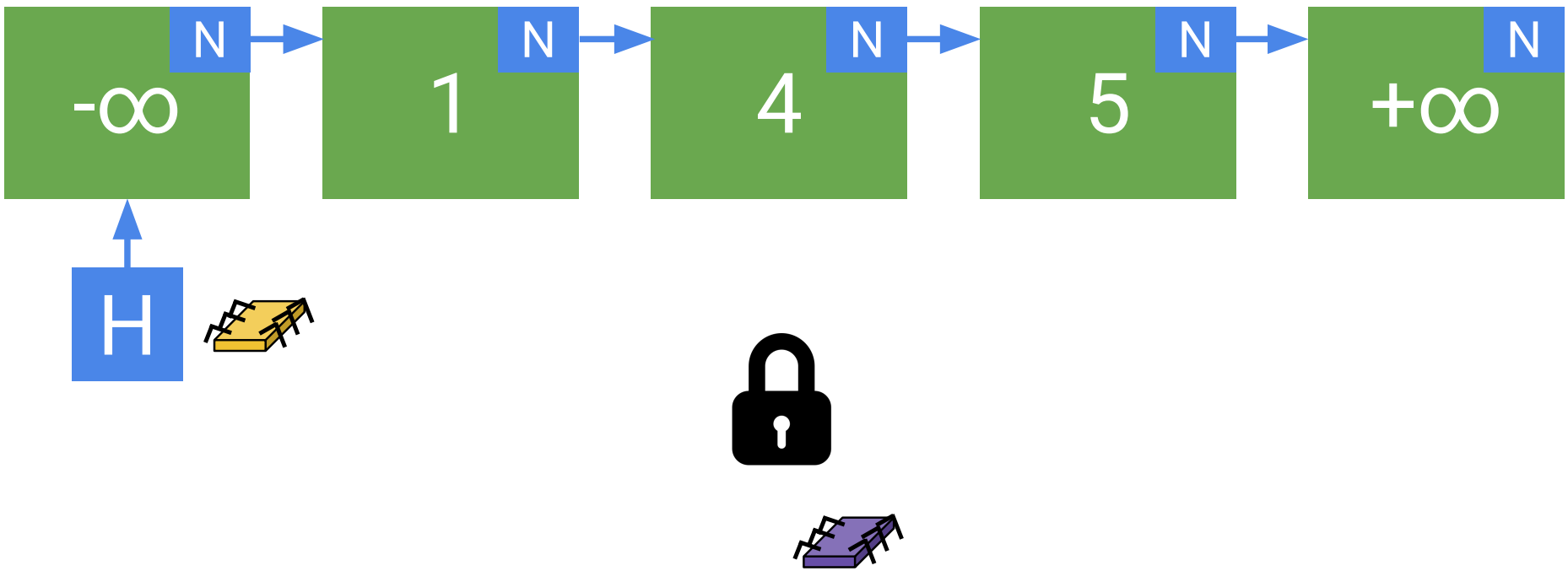
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



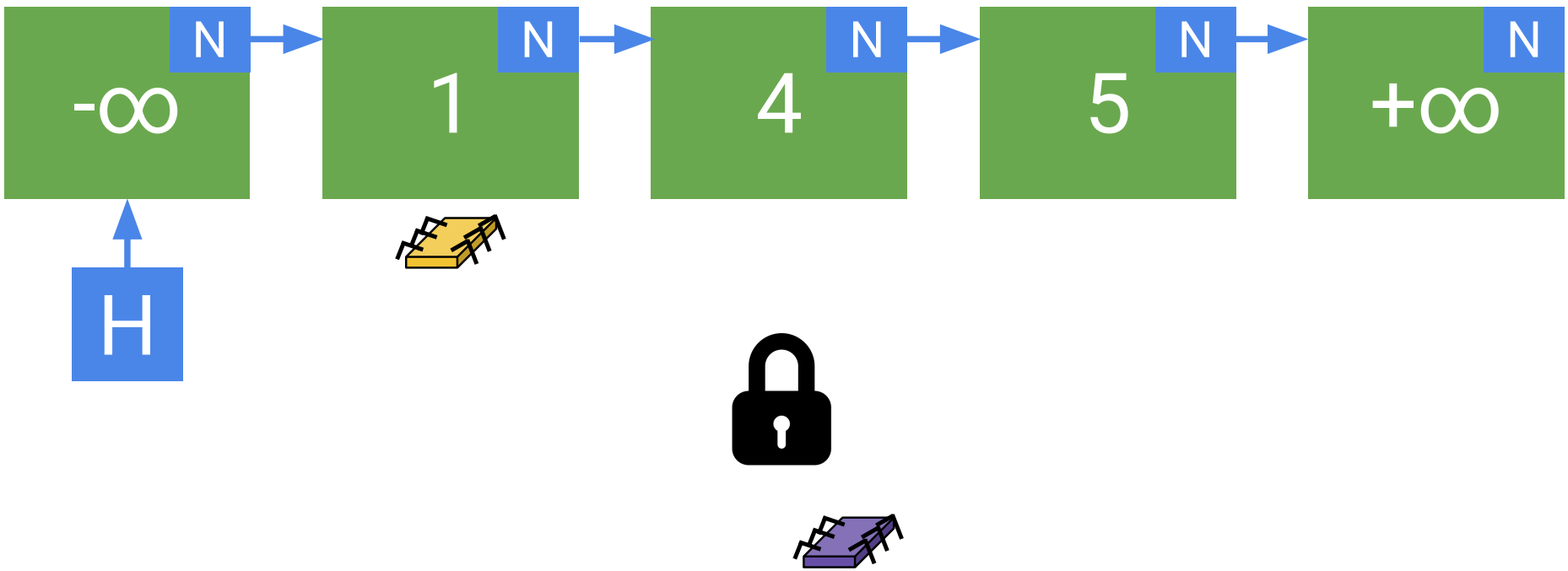
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



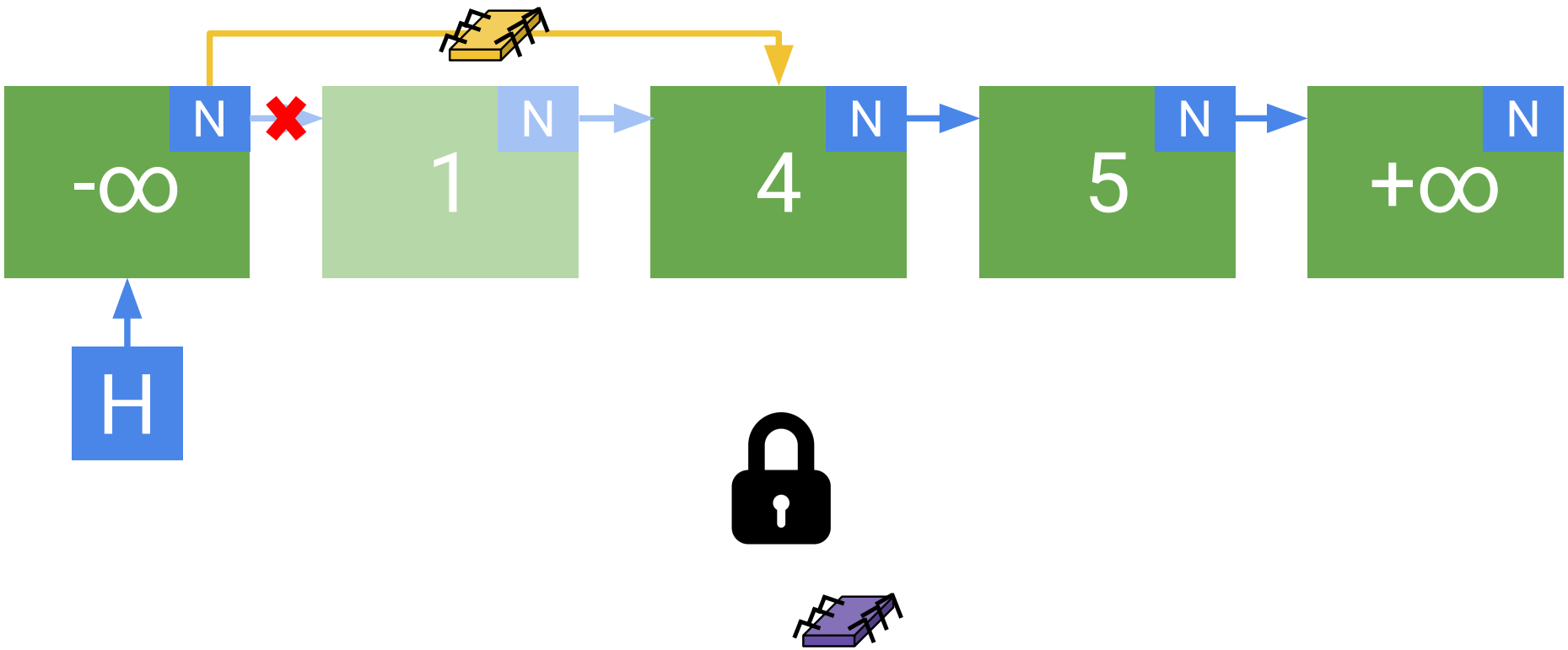
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



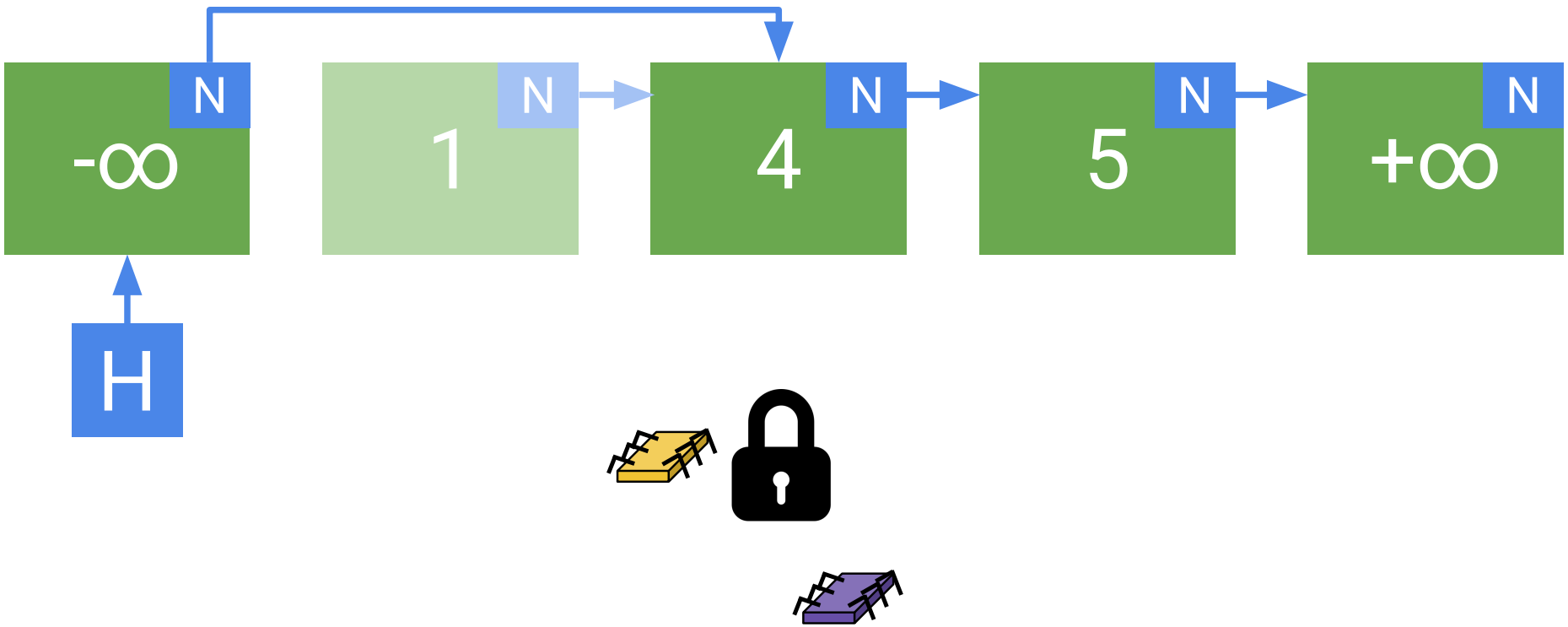
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



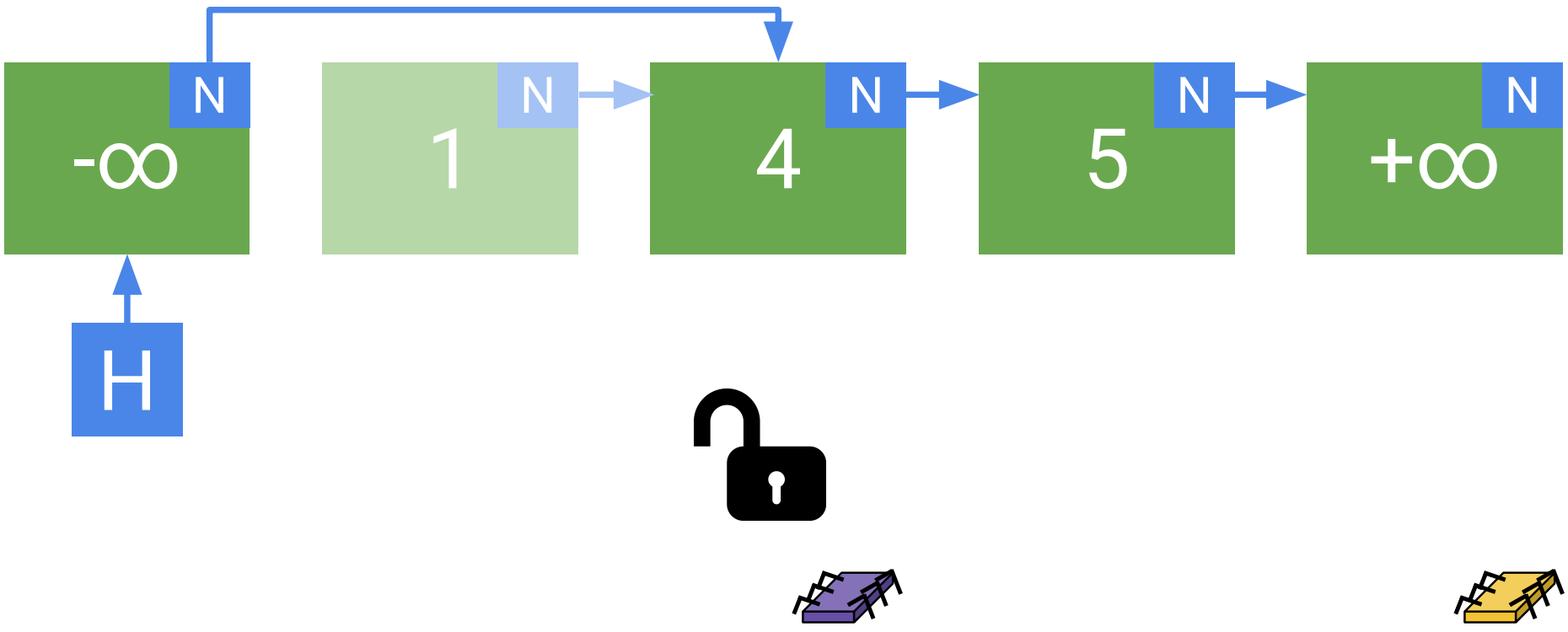
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



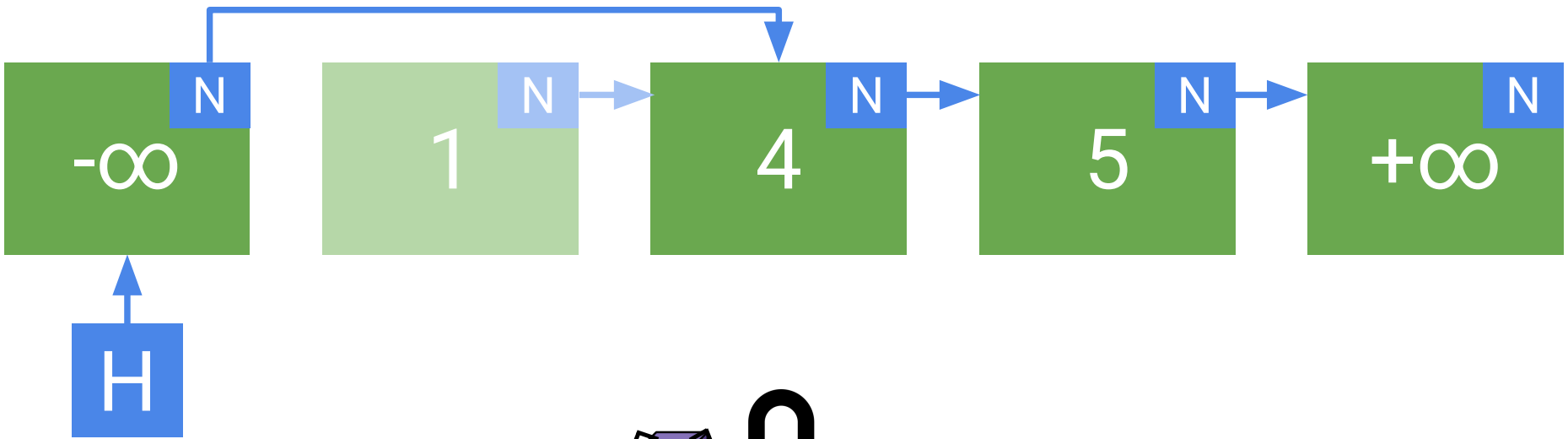
Пример

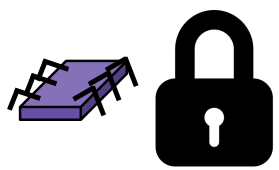
Жёлтый удаляет «1», фиолетовый удаляет «4»



Пример

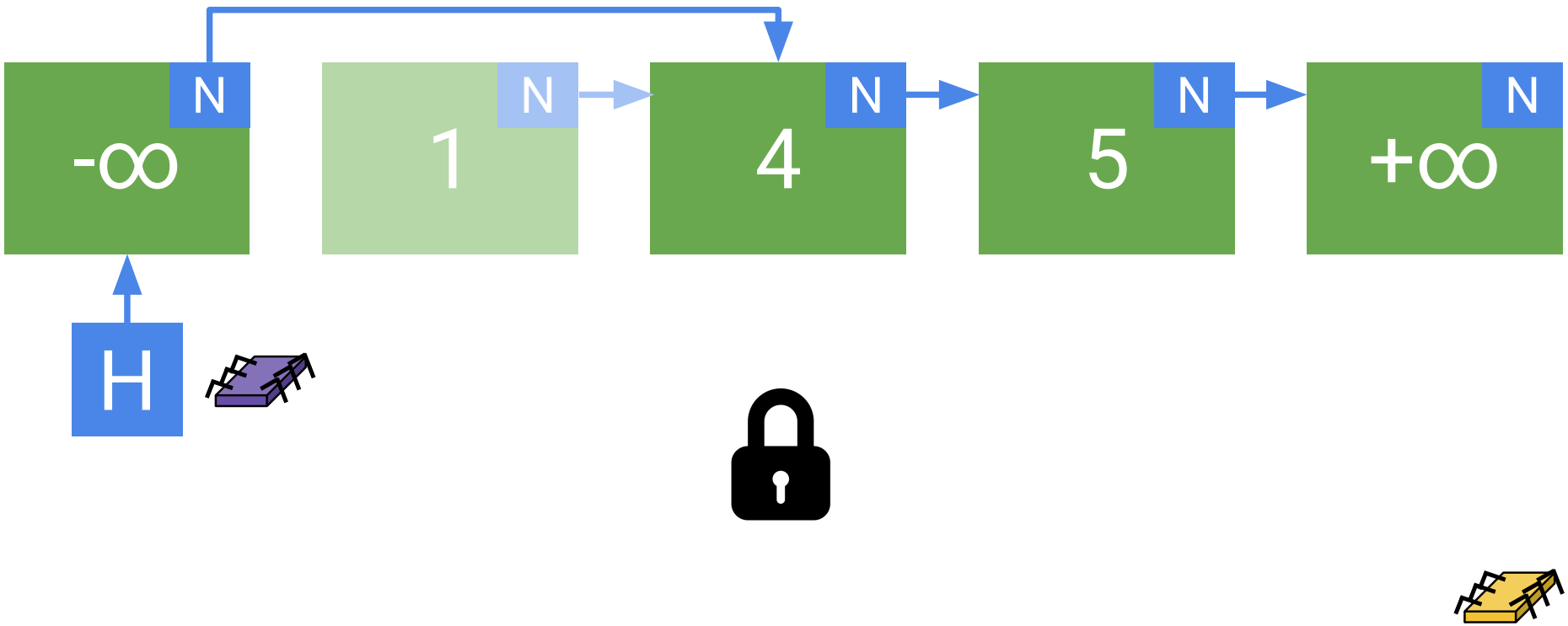
Жёлтый удаляет «1», фиолетовый удаляет «4»





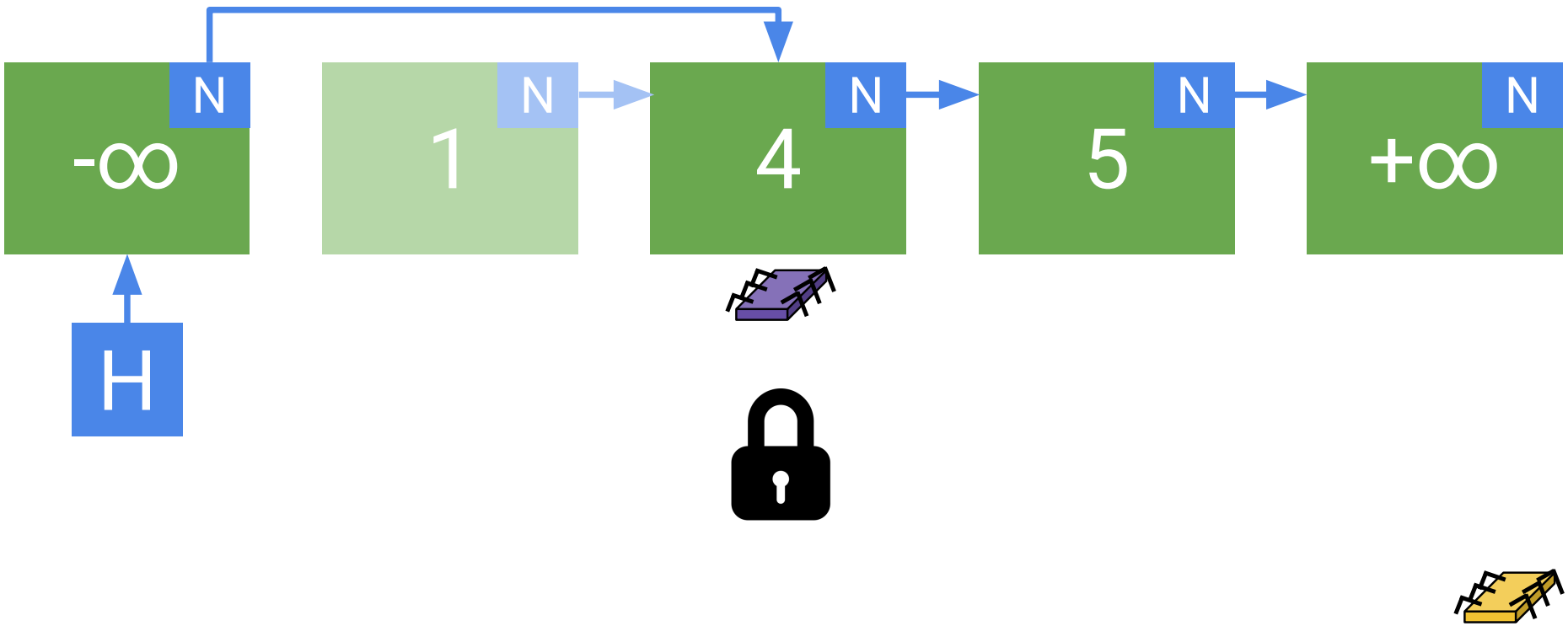
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



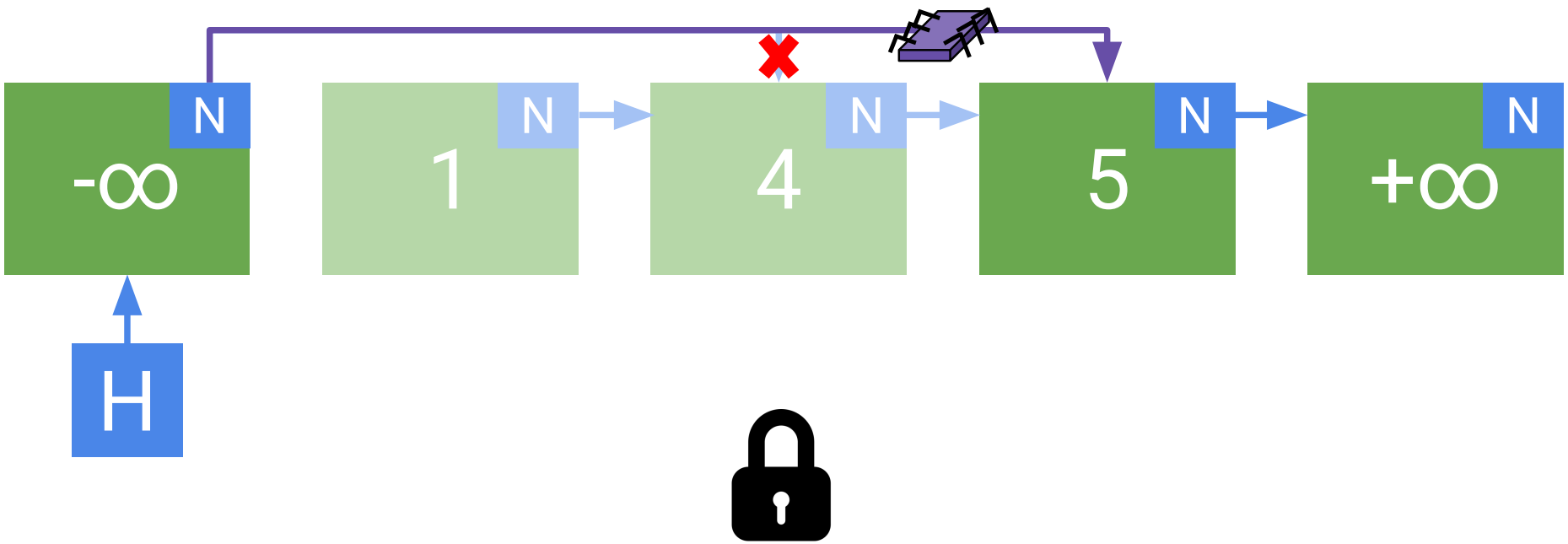
Пример

Жёлтый удаляет «1», фиолетовый удаляет «4»



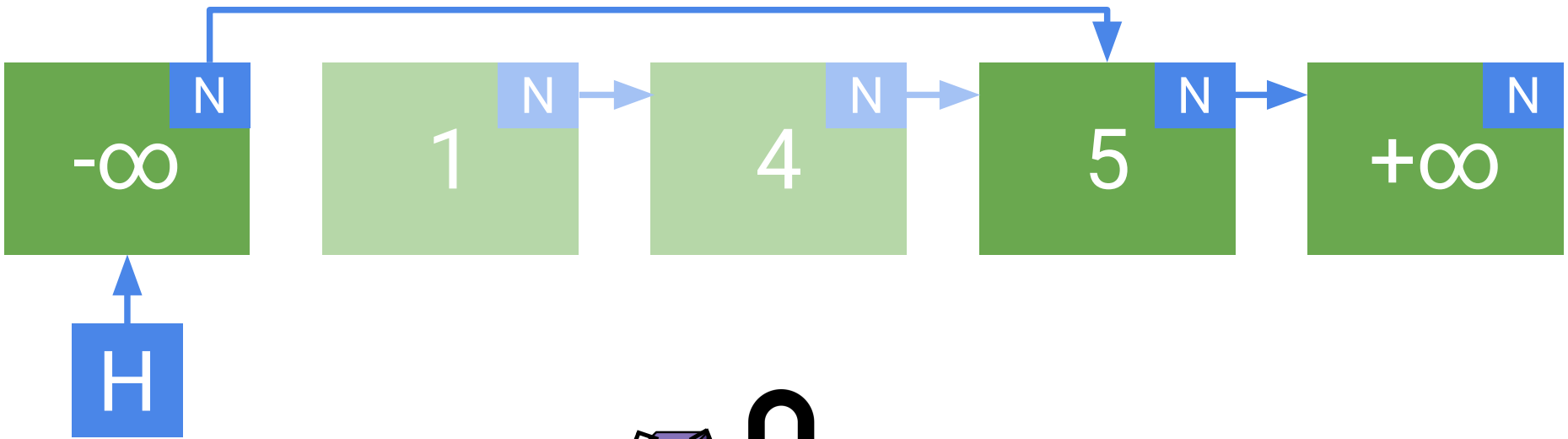
Пример

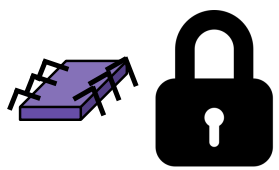
Жёлтый удаляет «1», фиолетовый удаляет «4»



Пример

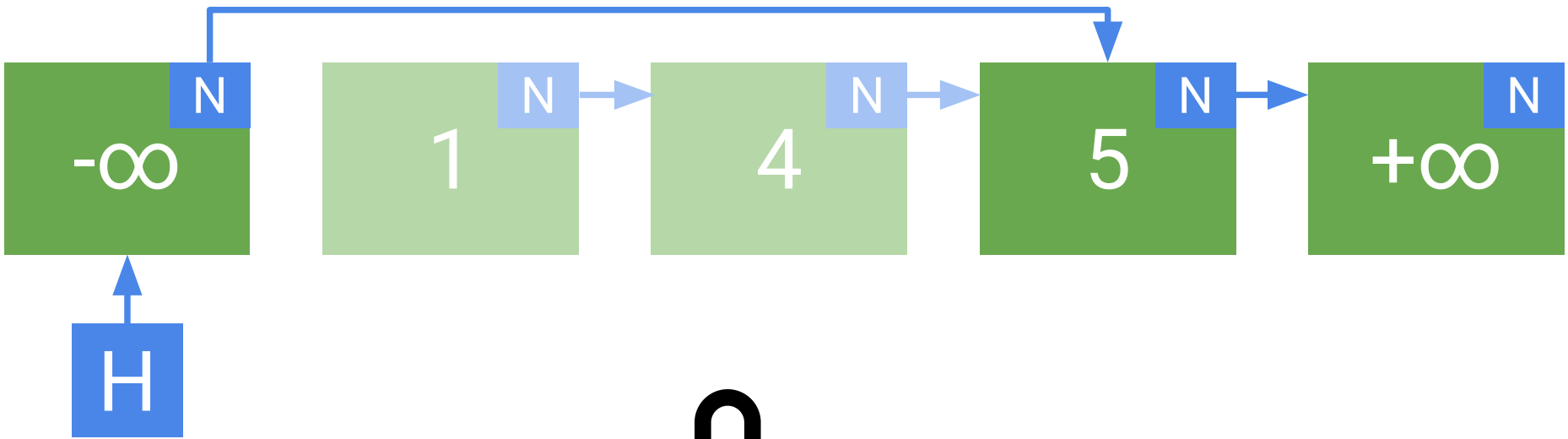
Жёлтый удаляет «1», фиолетовый удаляет «4»





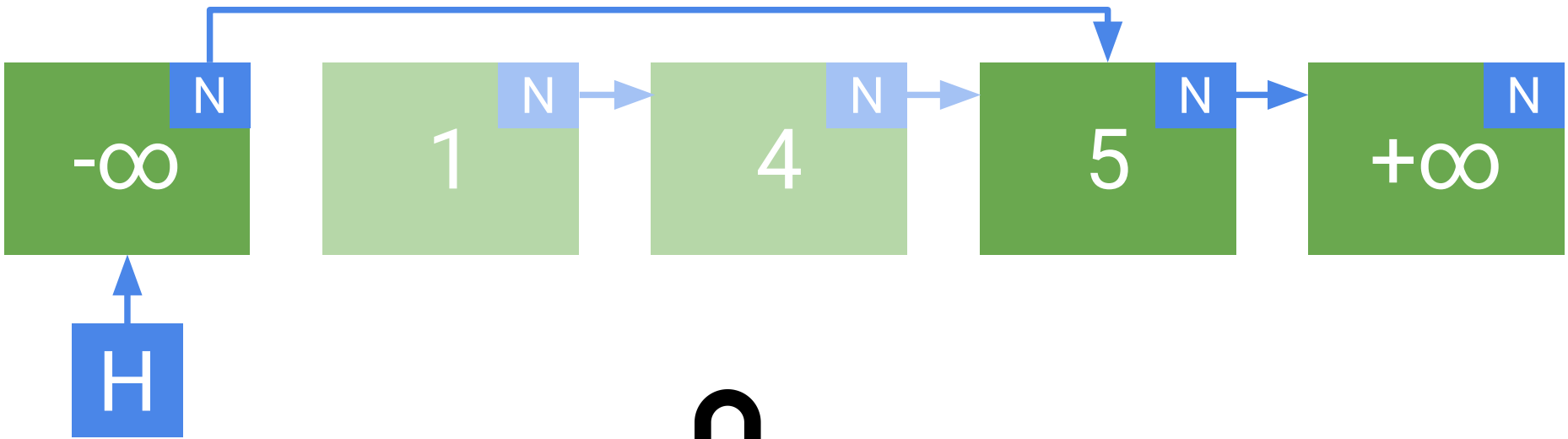
Пример

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)
  cur.N = Node(key, next)
}

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

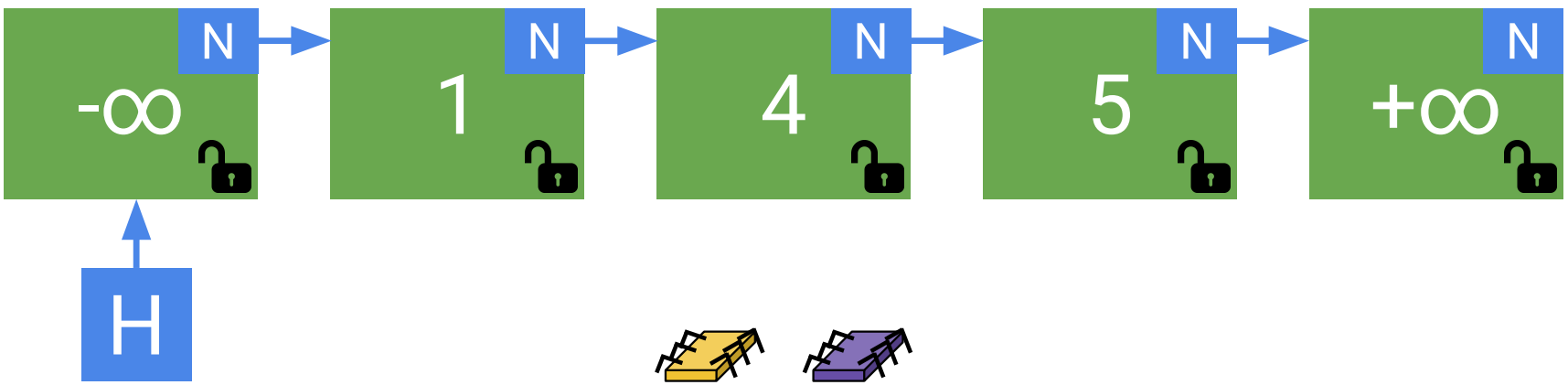
План

Тонкая синхронизация

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

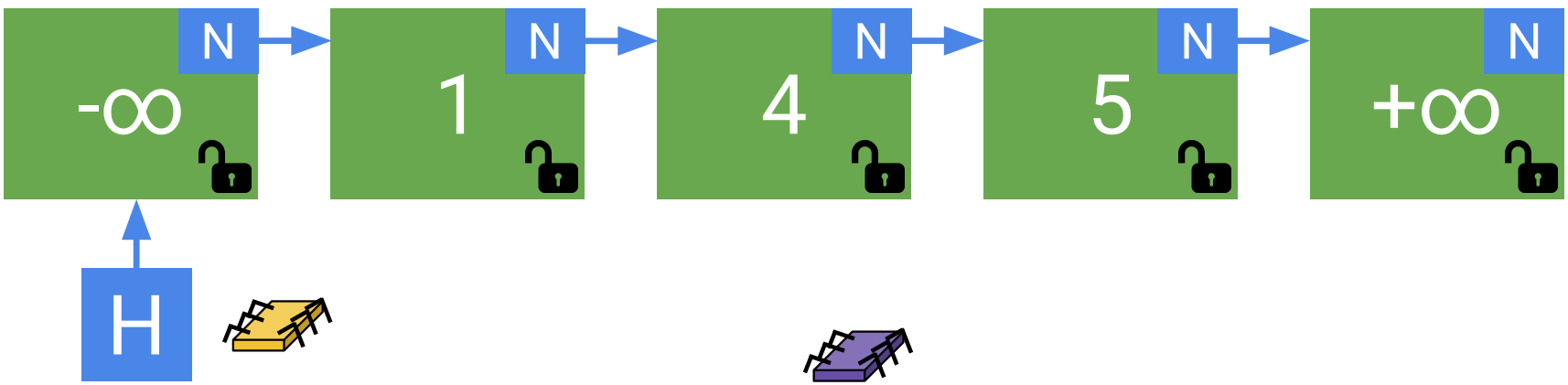
Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



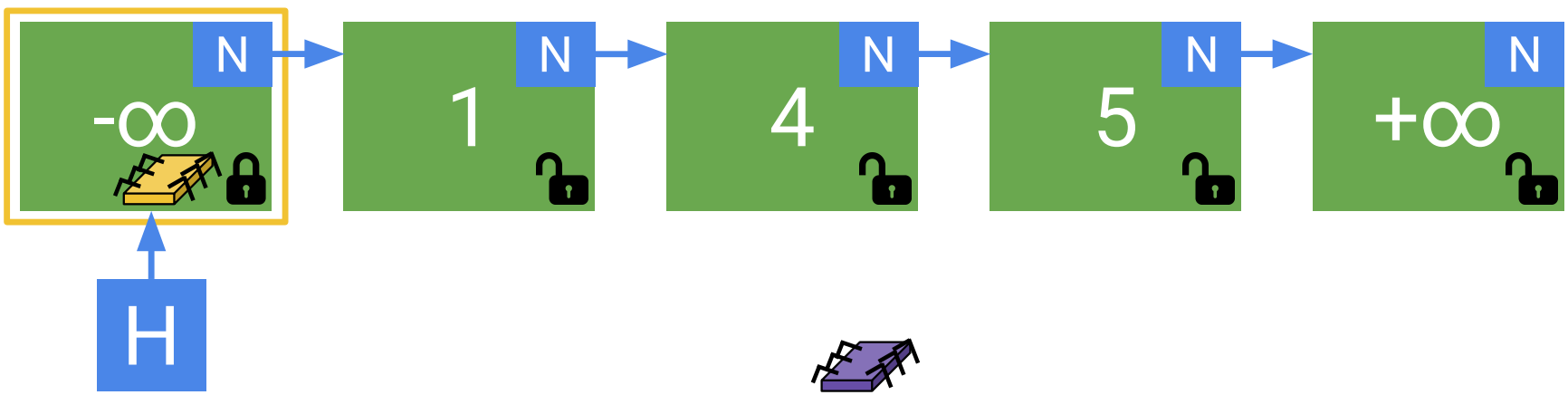
Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



Пример

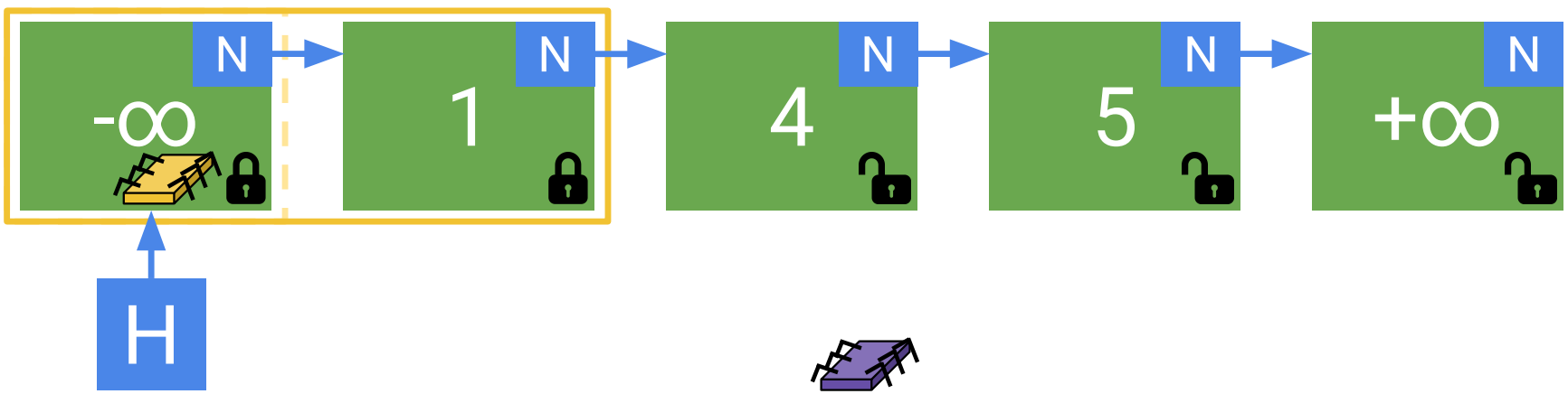
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый берёт блокировку на голову списка ...

Пример

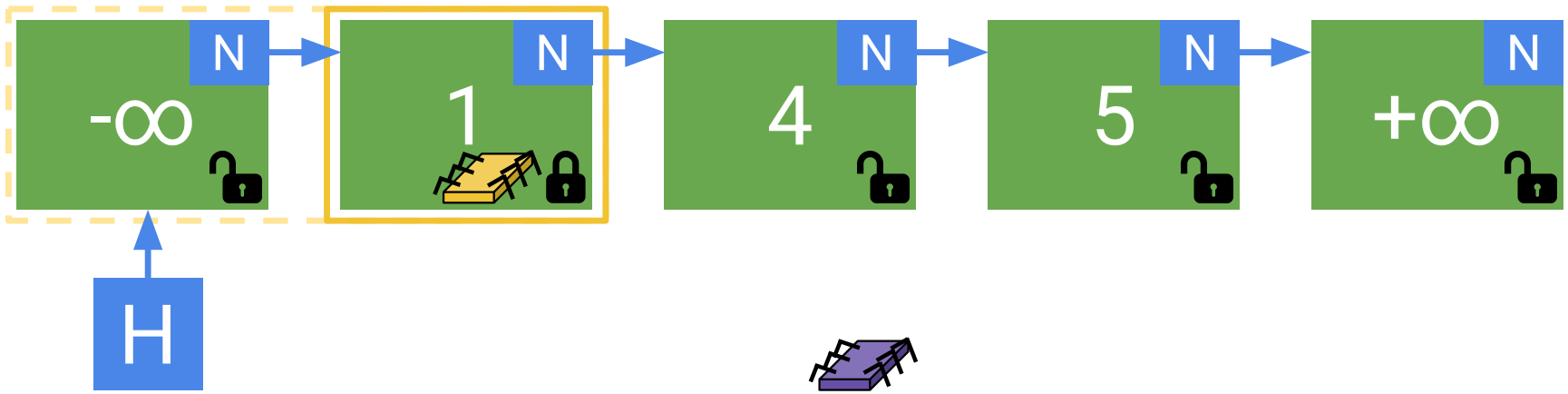
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и на следующий элемент

Пример

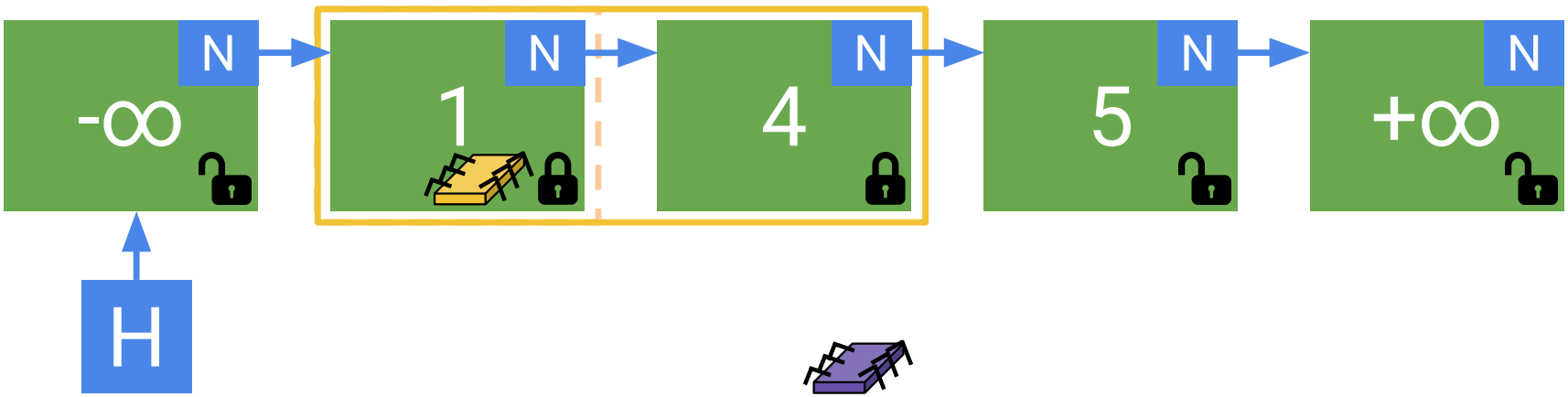
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый отпускает блокировку на голову списка ...

Пример

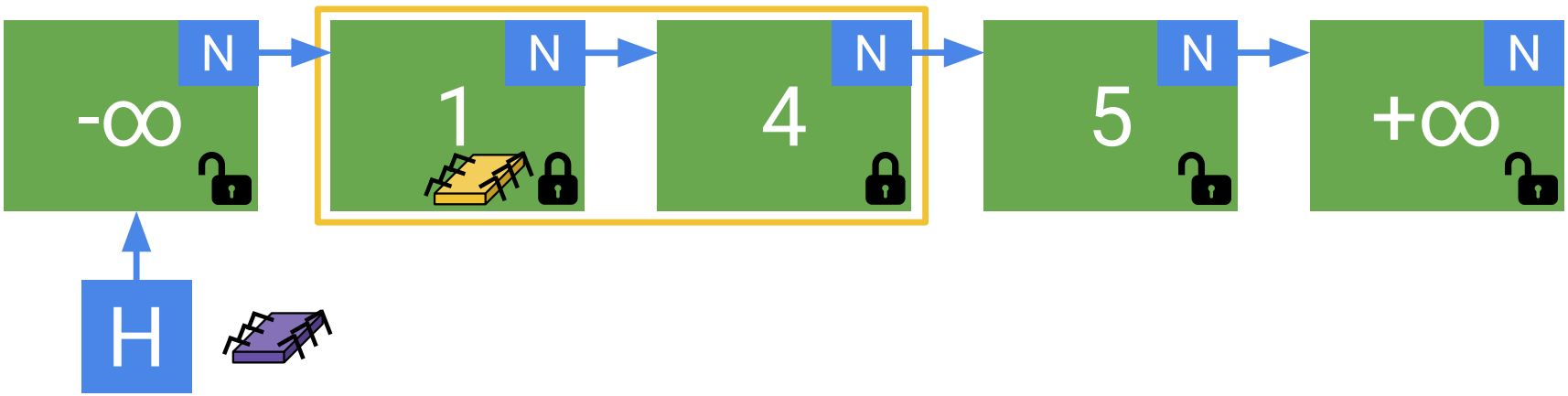
Жёлтый удаляет «4», фиолетовый удаляет «5»



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

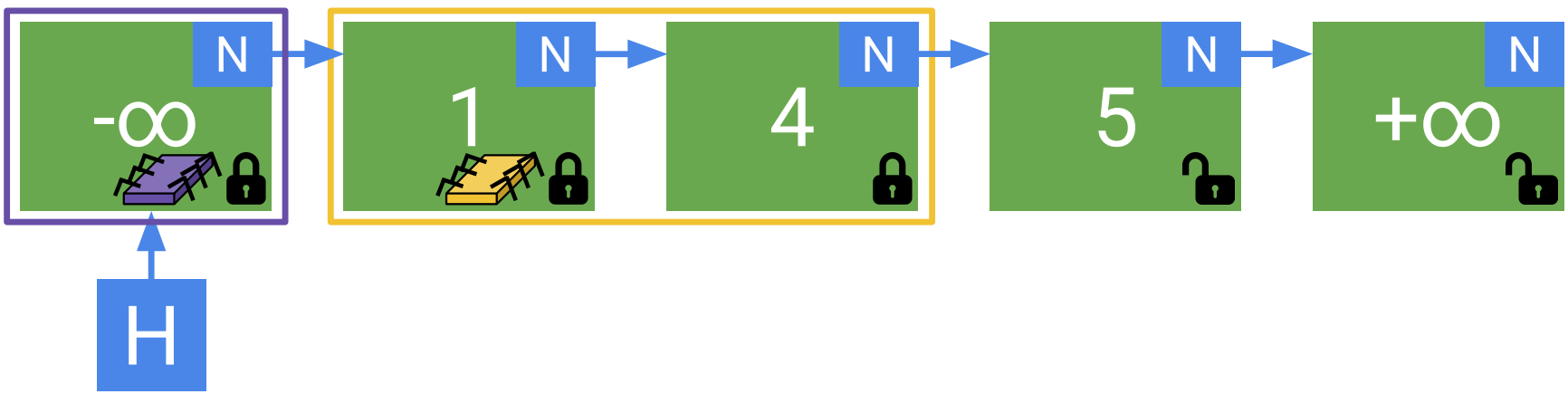
Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



Пример

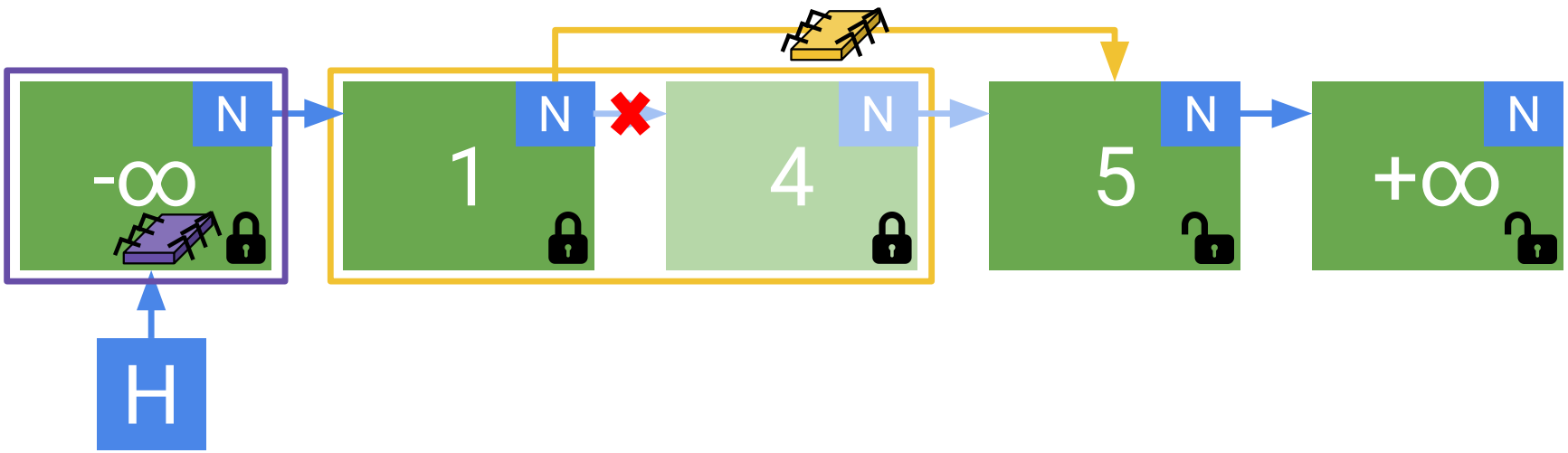
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на голову списка и ждет жёлтого

Пример

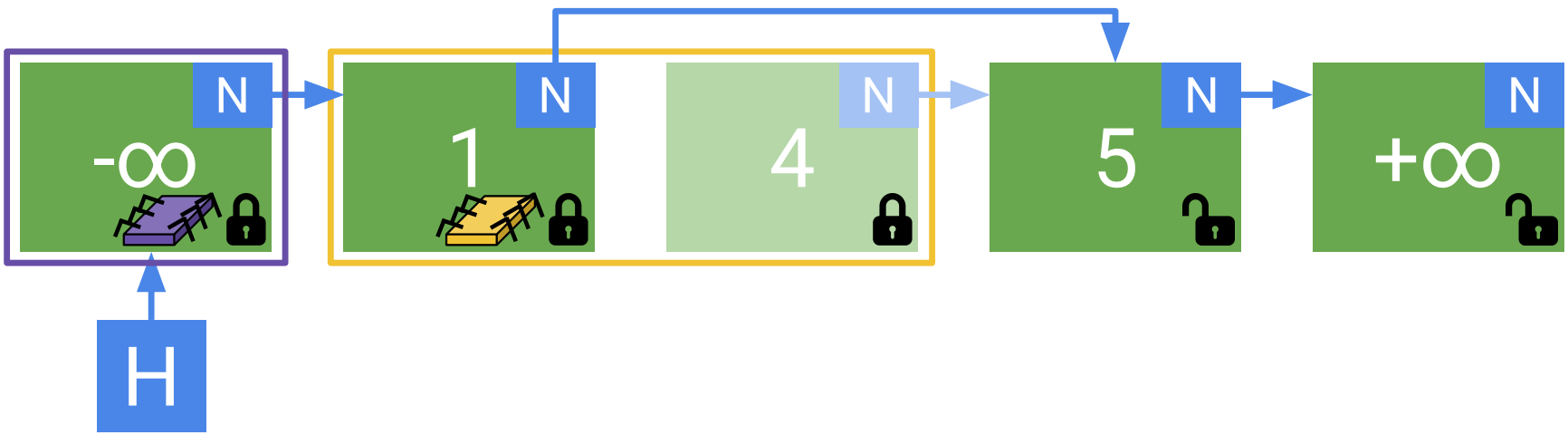
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый удаляет «4»

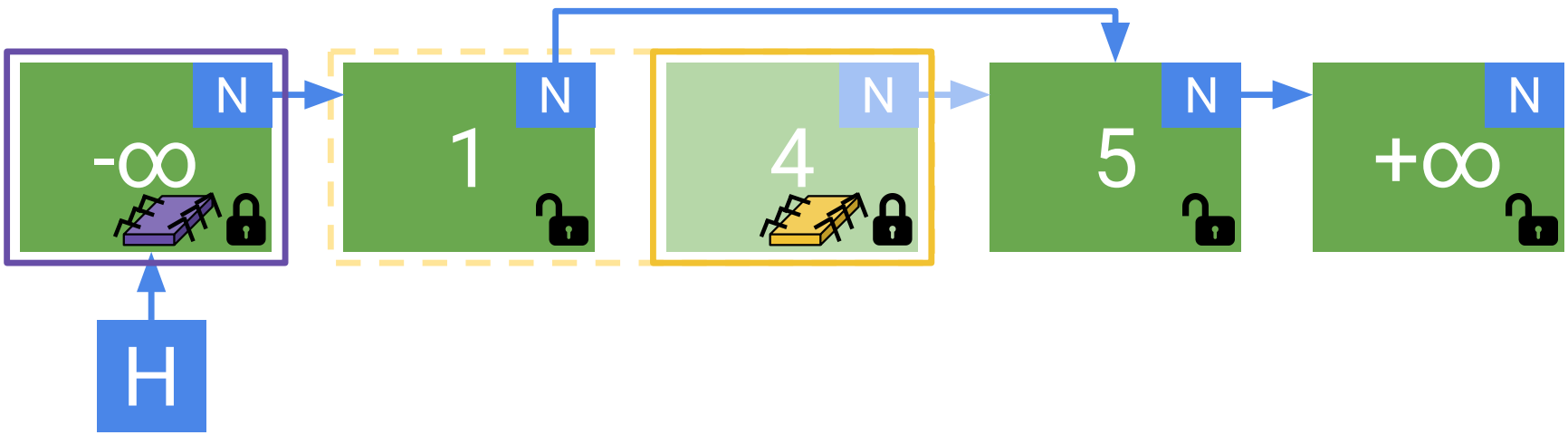
Пример

Жёлтый удаляет «4», фиолетовый удаляет «5»



Пример

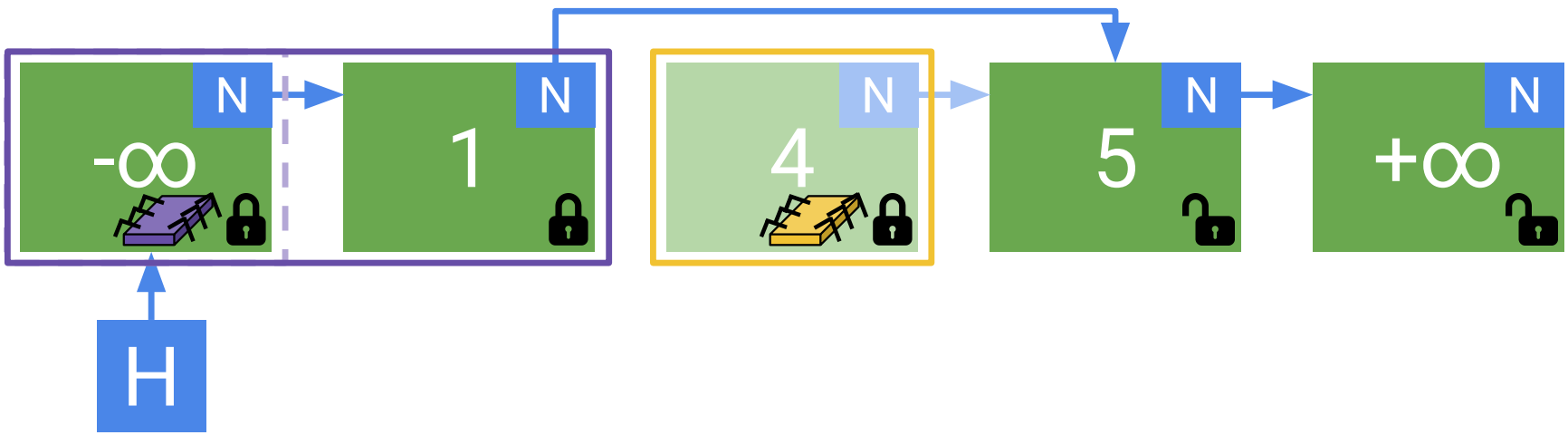
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый отпускает блокировку на «1»

Пример

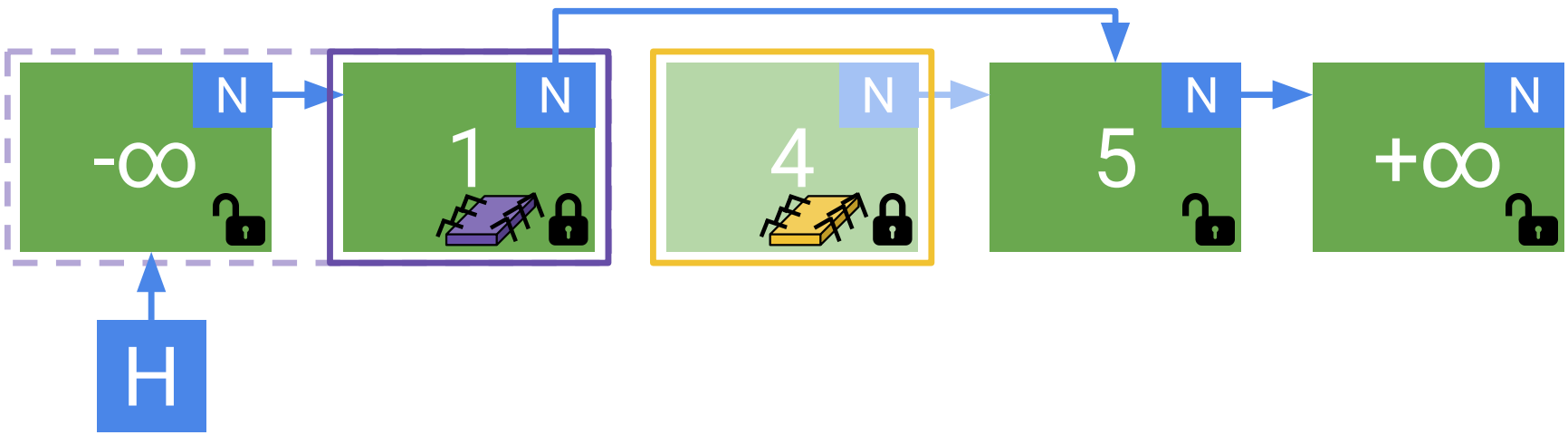
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на «1» ...

Пример

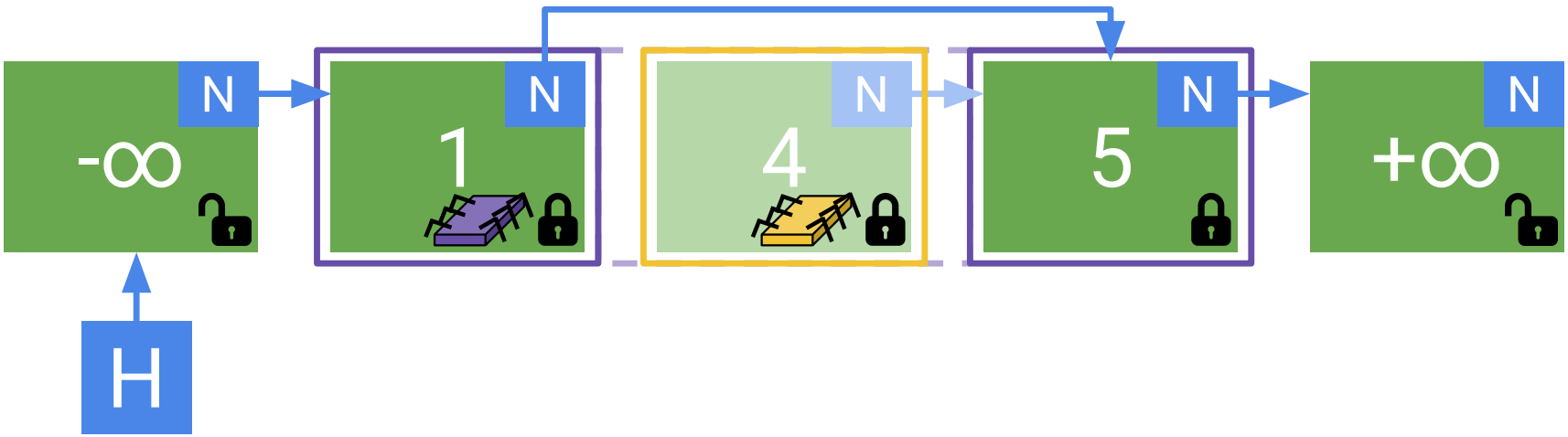
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и отпускает блокировку на голову списка

Пример

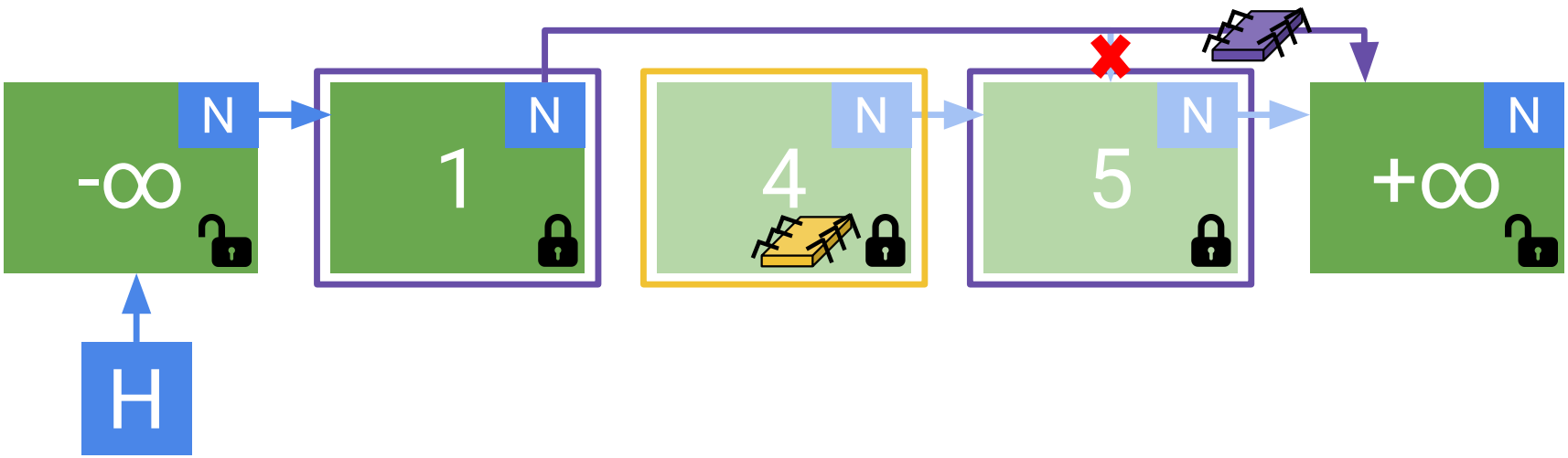
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый берёт блокировку на «5», нашёл окно.

Пример

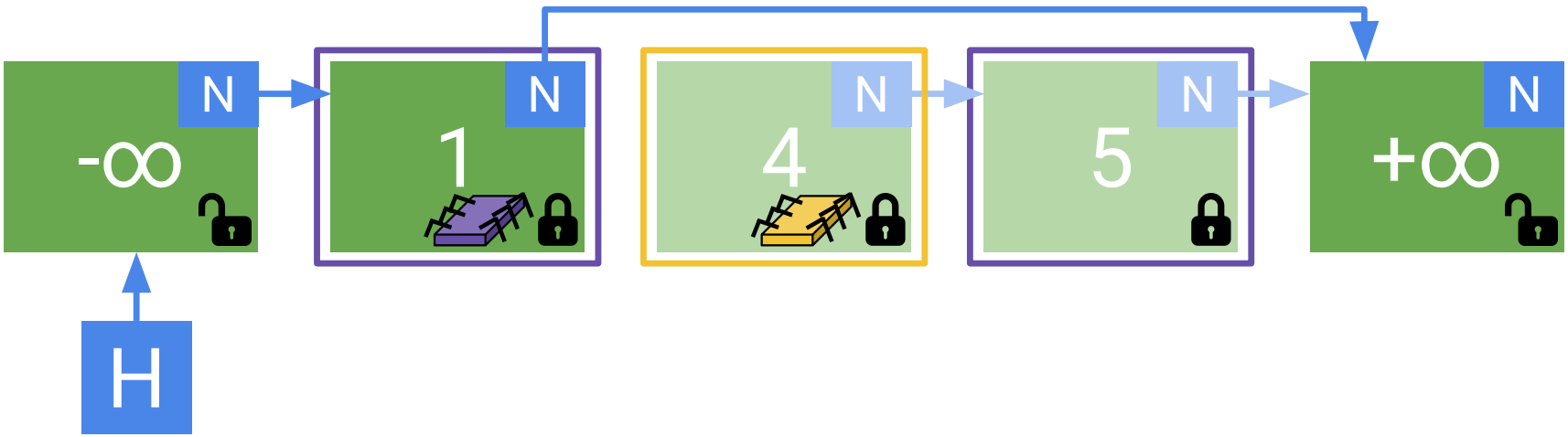
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый удаляет «5» ...

Пример

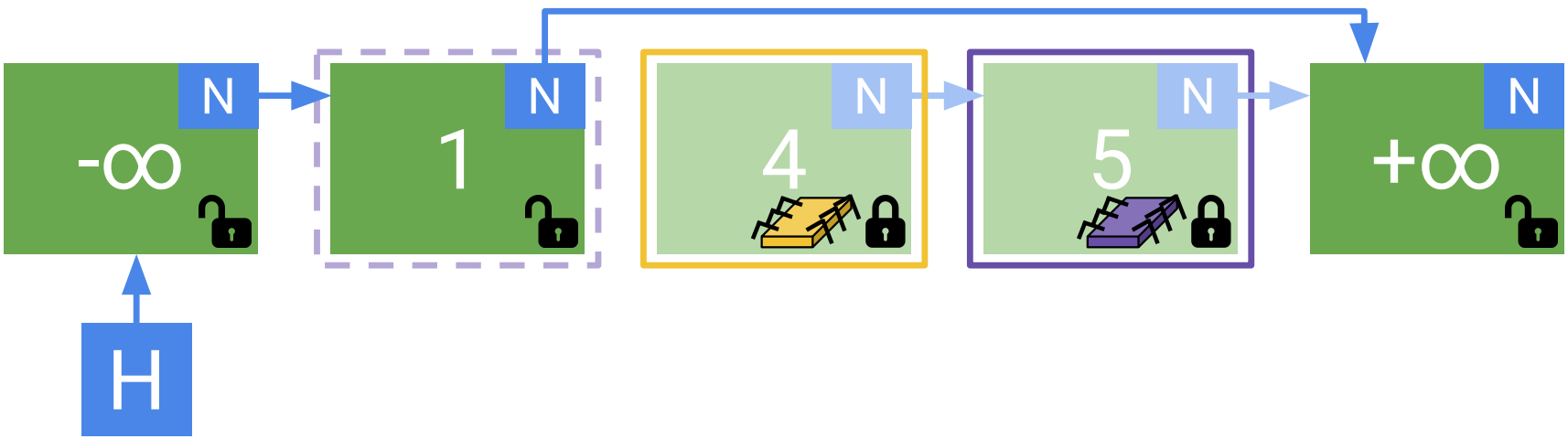
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый удаляет «5» ...

Пример

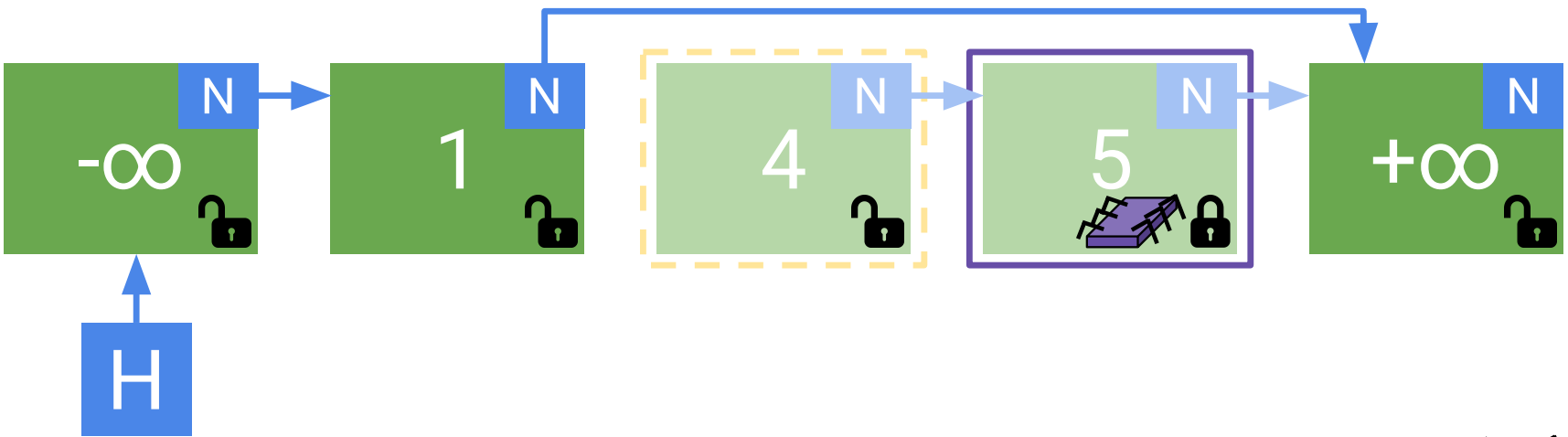
Жёлтый удаляет «4», фиолетовый удаляет «5»



... и отпускает блокировку на «1»

Пример

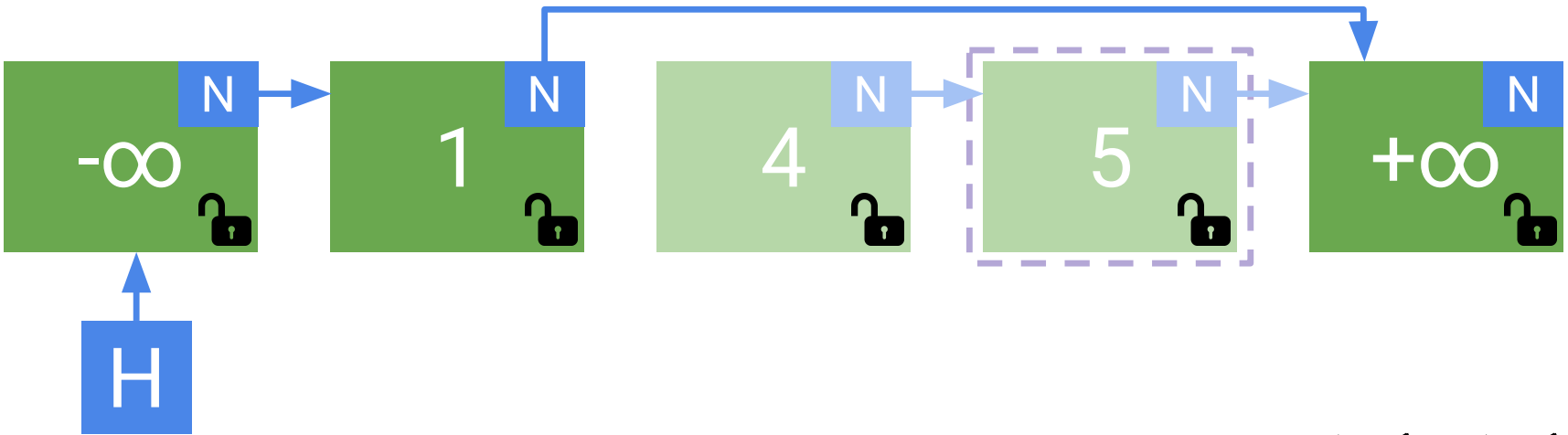
Жёлтый удаляет «4», фиолетовый удаляет «5»



Жёлтый отпускает блокировку на «4»

Пример

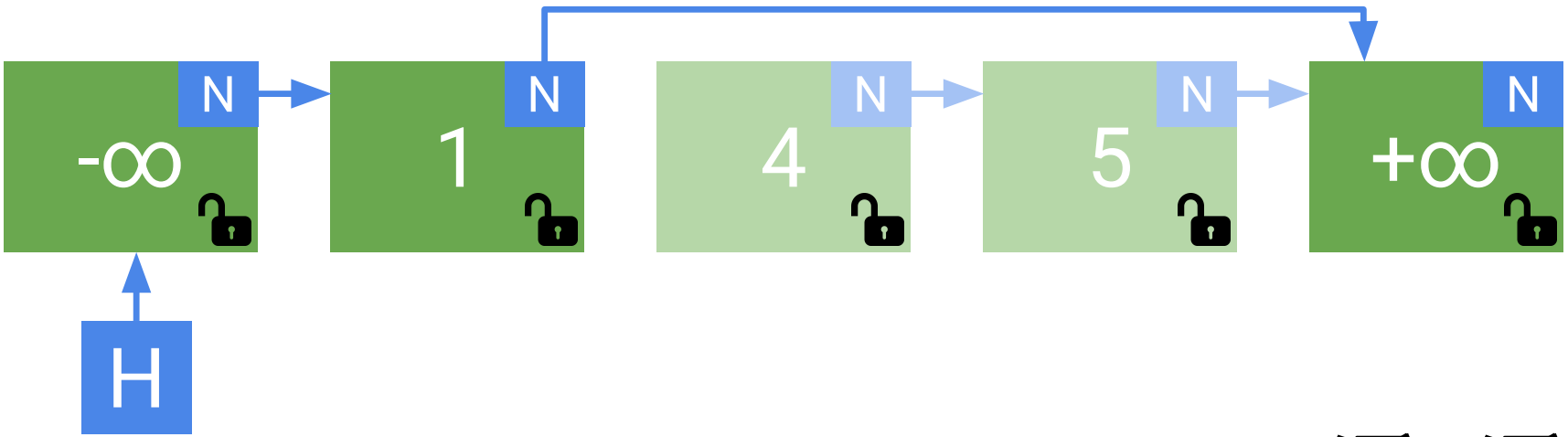
Жёлтый удаляет «4», фиолетовый удаляет «5»



Фиолетовый отпускает блокировку на «5»

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

Тонкая синхронизация: задание

Задание «linked-list-set»: теперь используем тонкую блокировку

Корректность

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

План

Алгоритм абстрактной операции

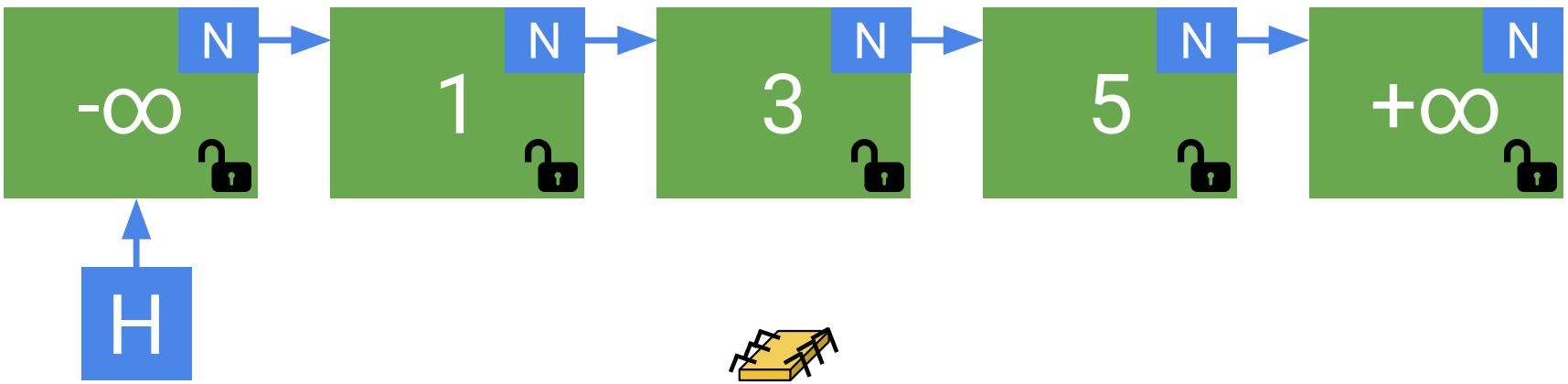
1. Найти окно $(\texttt{cur}, \texttt{next})$ без синхронизации
2. Взять блокировки на $\texttt{cur}$ и $\texttt{next}$
3. Проверить инвариант $\texttt{cur}.N = \texttt{next}$
4. Проверить, что $\texttt{cur}$ не удалён
5. Выполнить операцию (добавить, удалить, ...)
6. При любой ошибке начать заново

Проверка, что узел не удалён

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

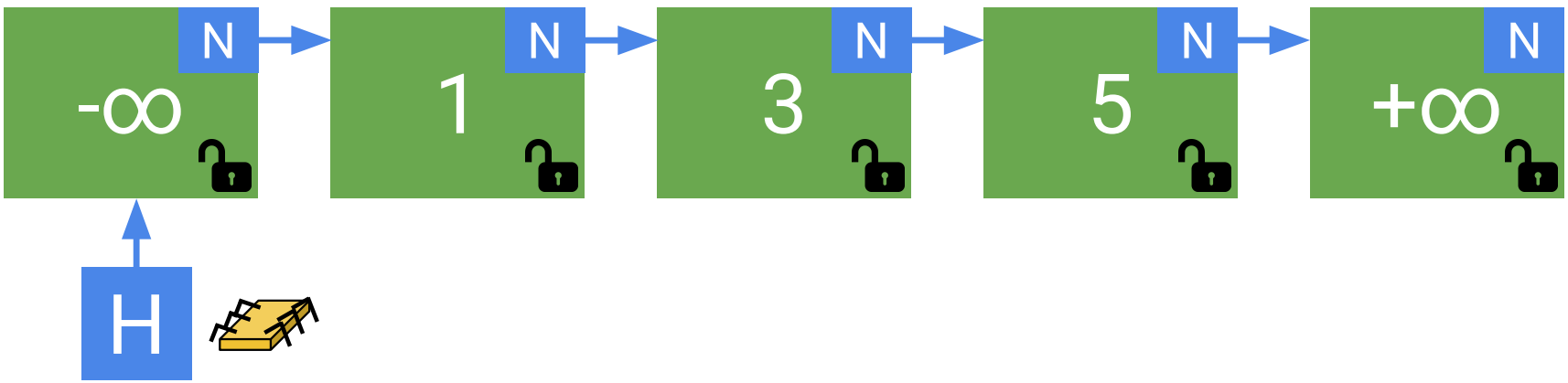
Пример успешной операции

Жёлтый добавляет «4»



Пример успешной операции

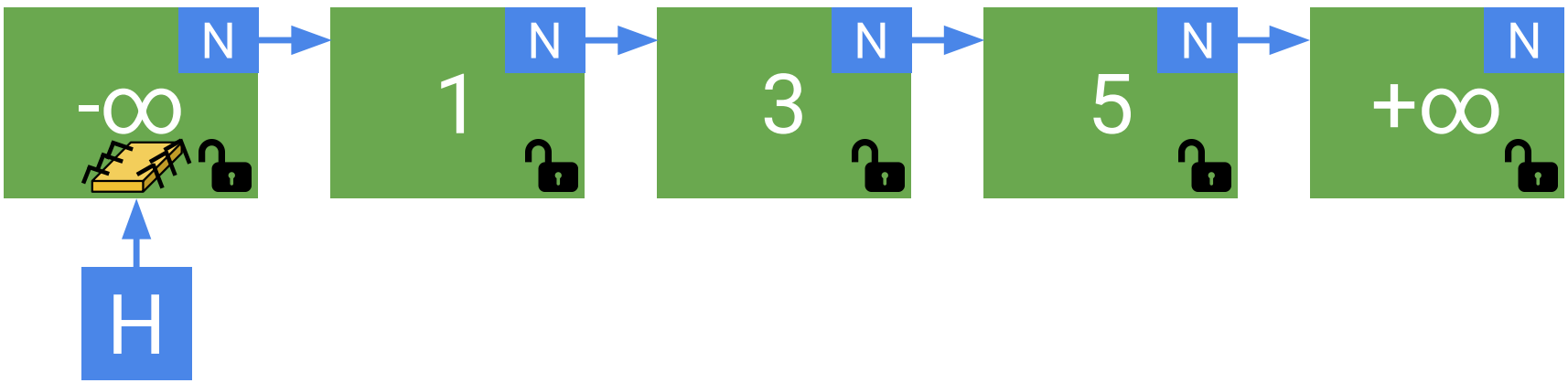
Жёлтый добавляет «4»



Жёлтый ищет окно

Пример успешной операции

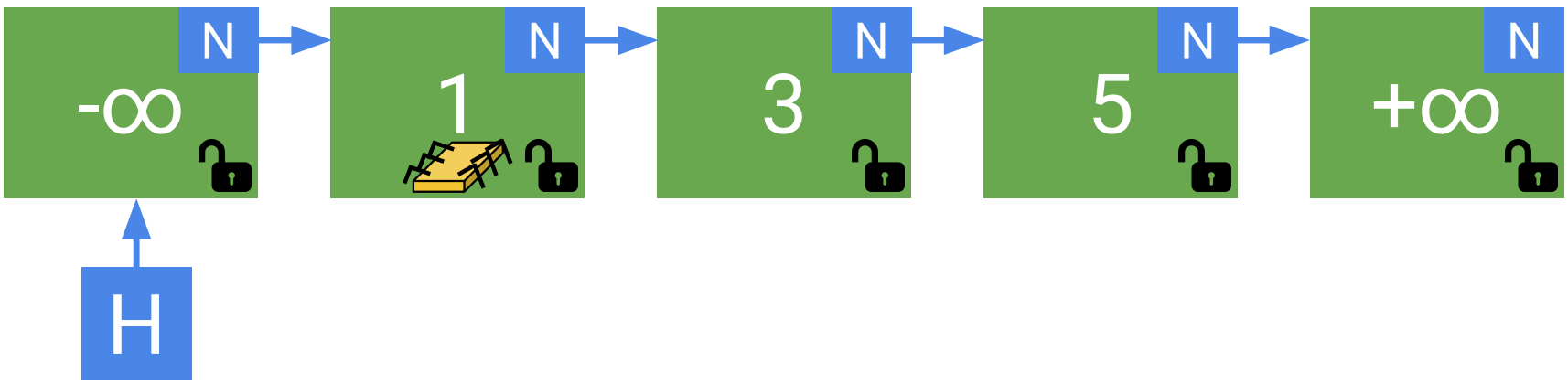
Жёлтый добавляет «4»



Жёлтый ищет окно

Пример успешной операции

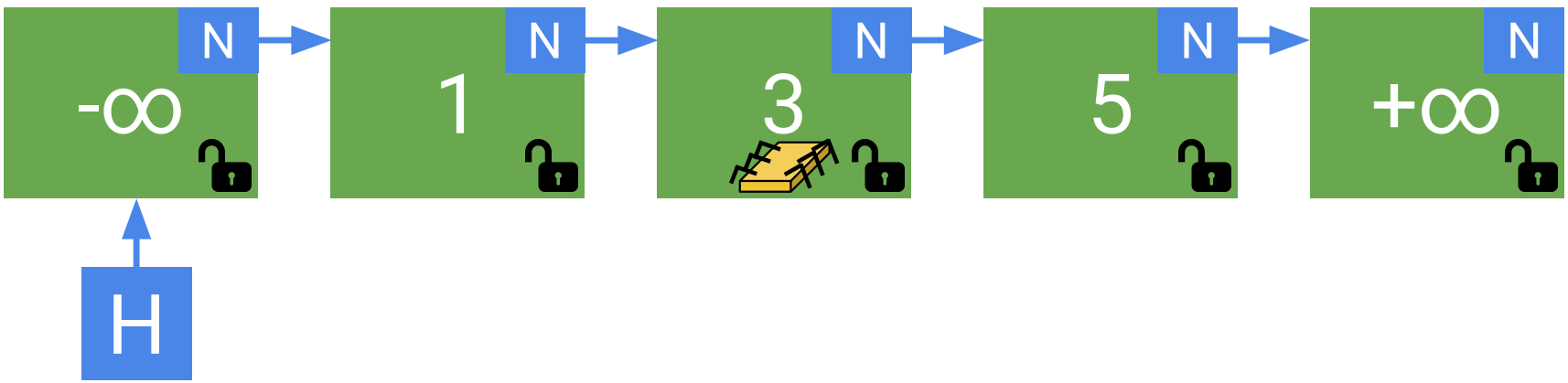
Жёлтый добавляет «4»



Жёлтый ищет окно

Пример успешной операции

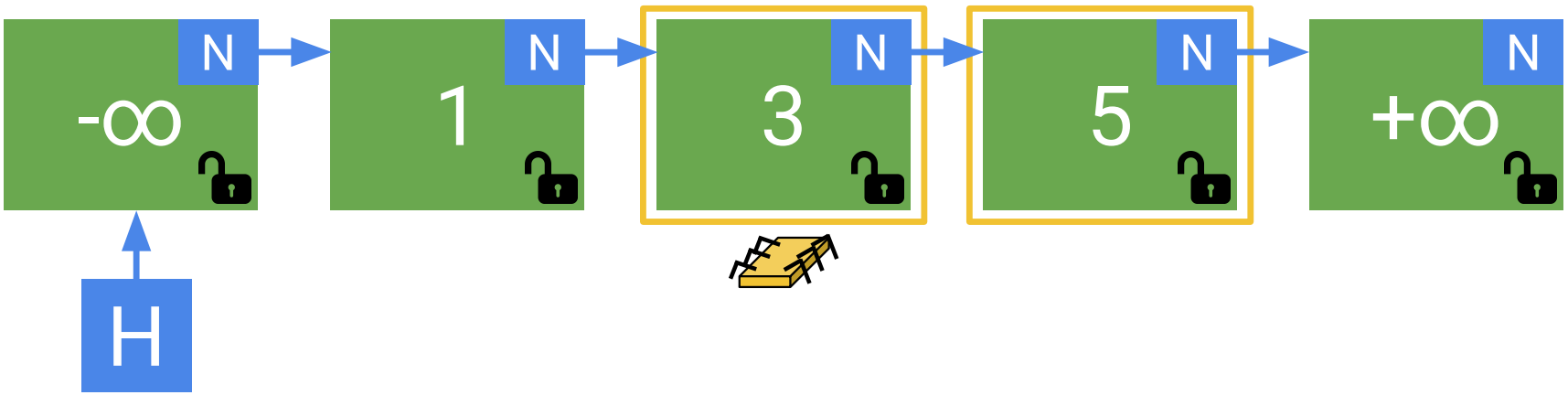
Жёлтый добавляет «4»



Жёлтый ищет окно

Пример успешной операции

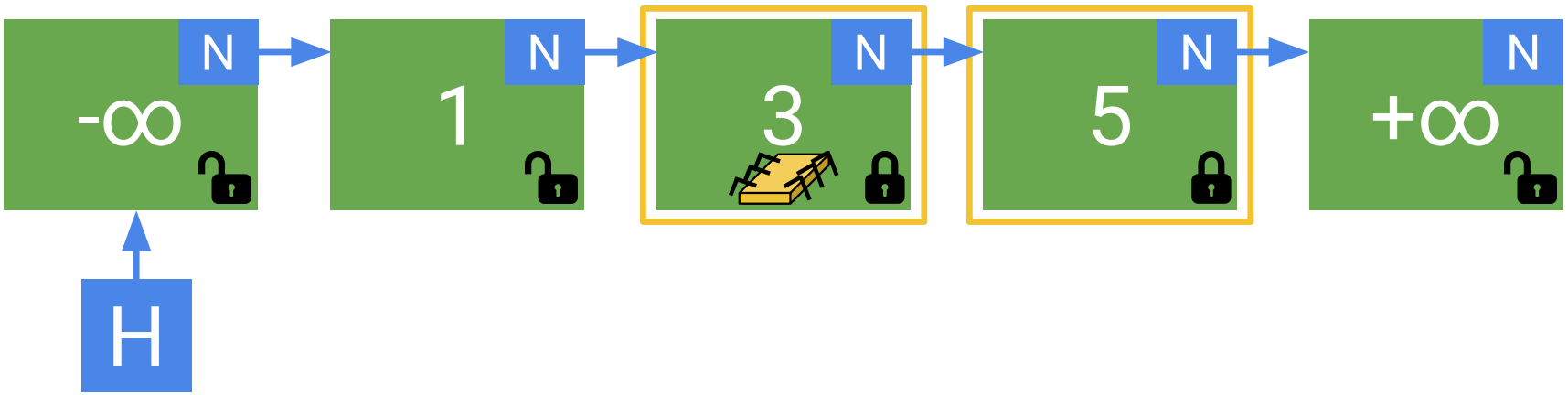
Жёлтый добавляет «4»



Нашёл окно

Пример успешной операции

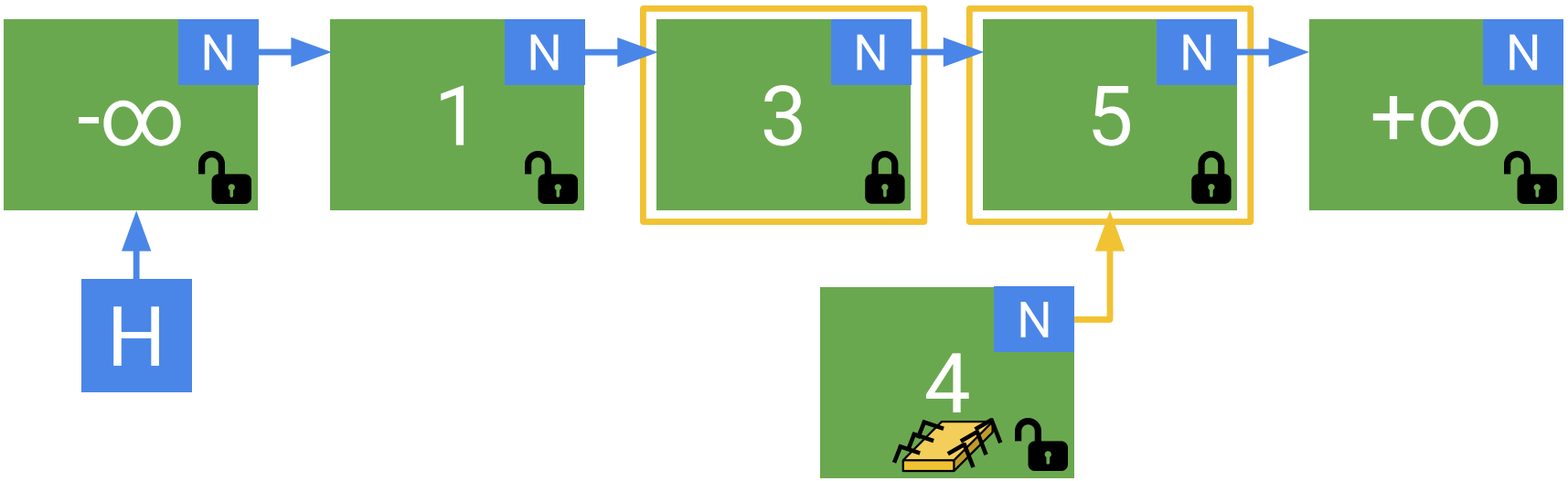
Жёлтый добавляет «4»



Берёт блокировки

Пример успешной операции

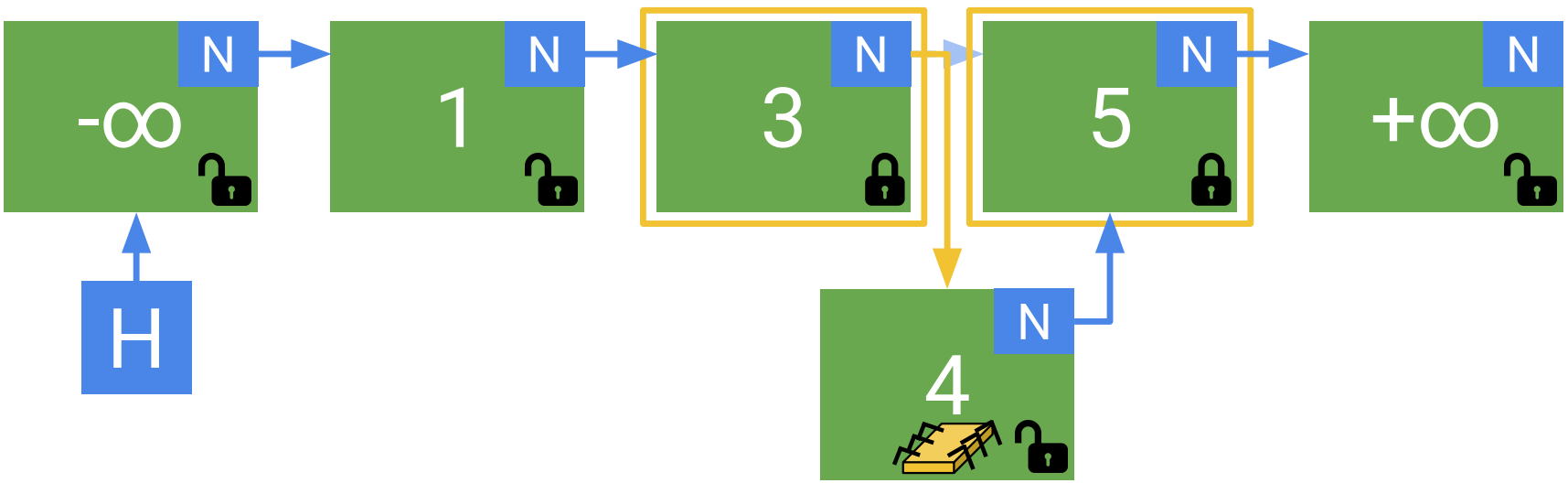
Жёлтый добавляет «4»



Добавляет узел «4»

Пример успешной операции

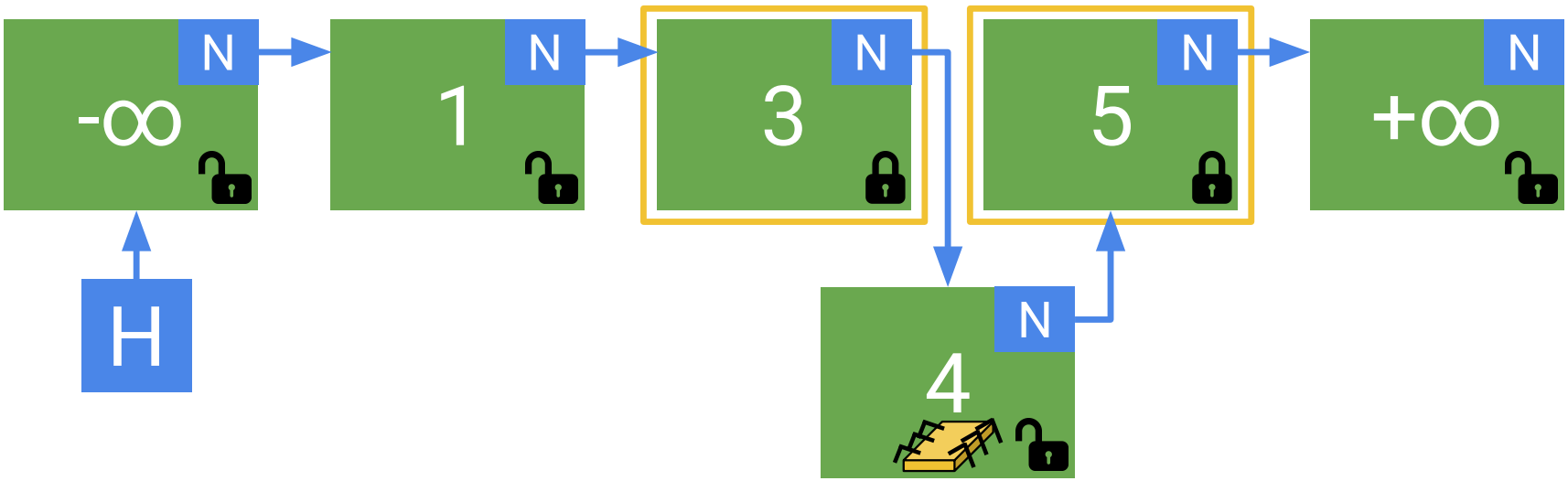
Жёлтый добавляет «4»



Добавляет узел «4»

Пример успешной операции

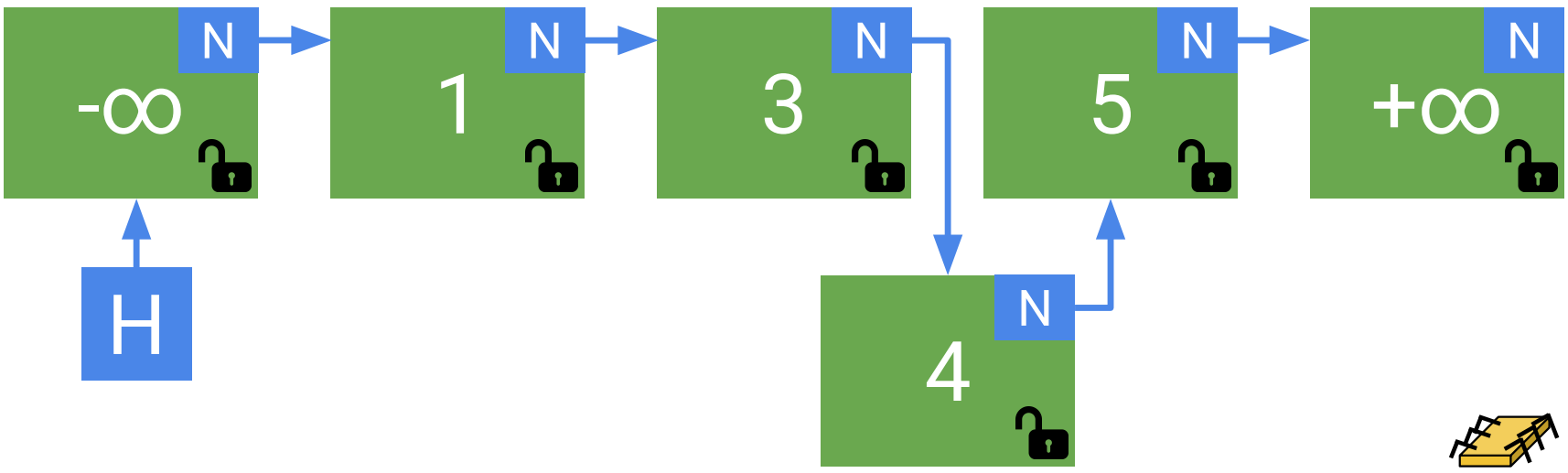
Жёлтый добавляет «4»



Добавляет узел «4»

Пример успешной операции

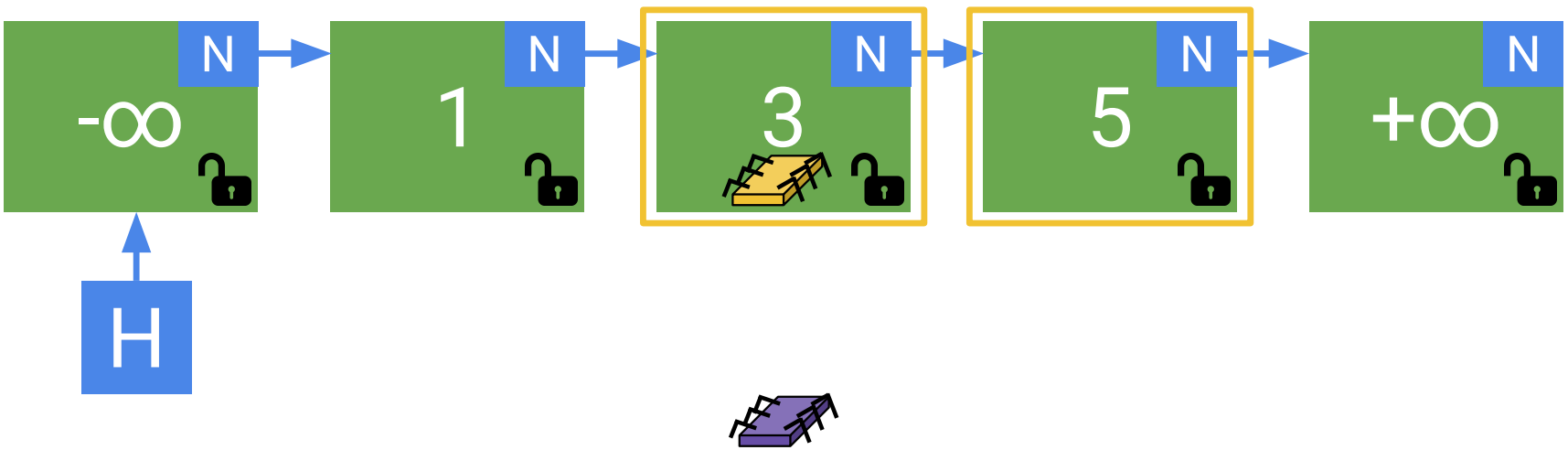
Жёлтый добавляет «4»



Отпускает блокировки и уходит

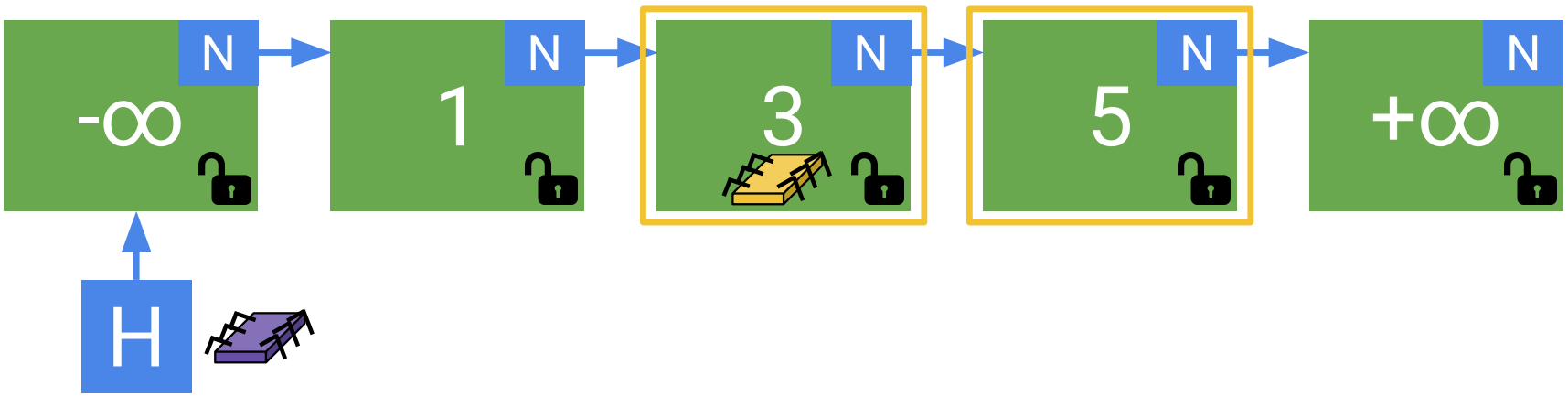
Проблема: cur.N != next

Оба добавляют «4»



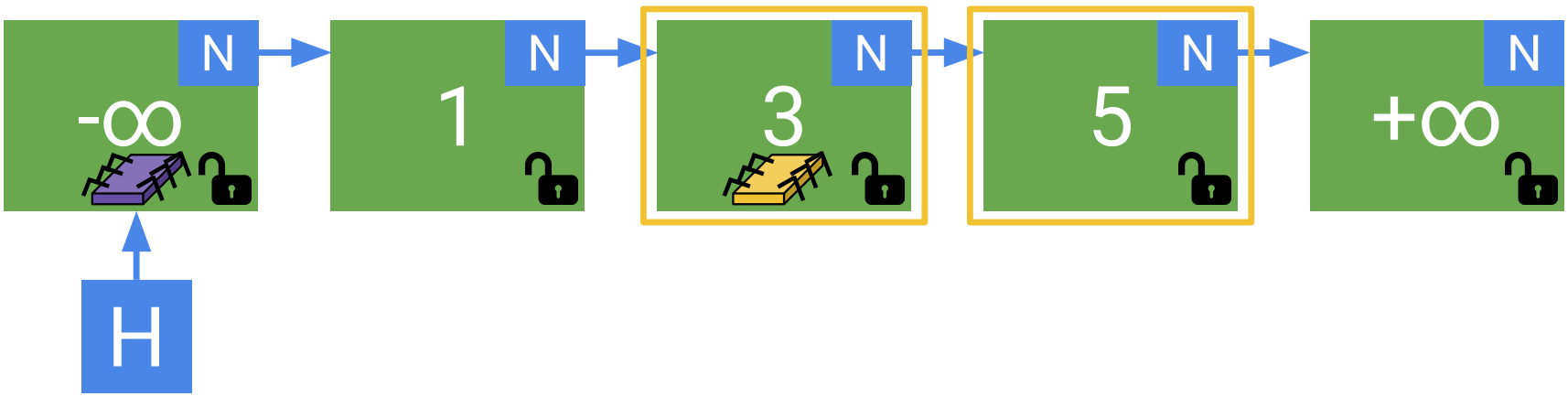
Проблема: cur.N != next

Оба добавляют «4»



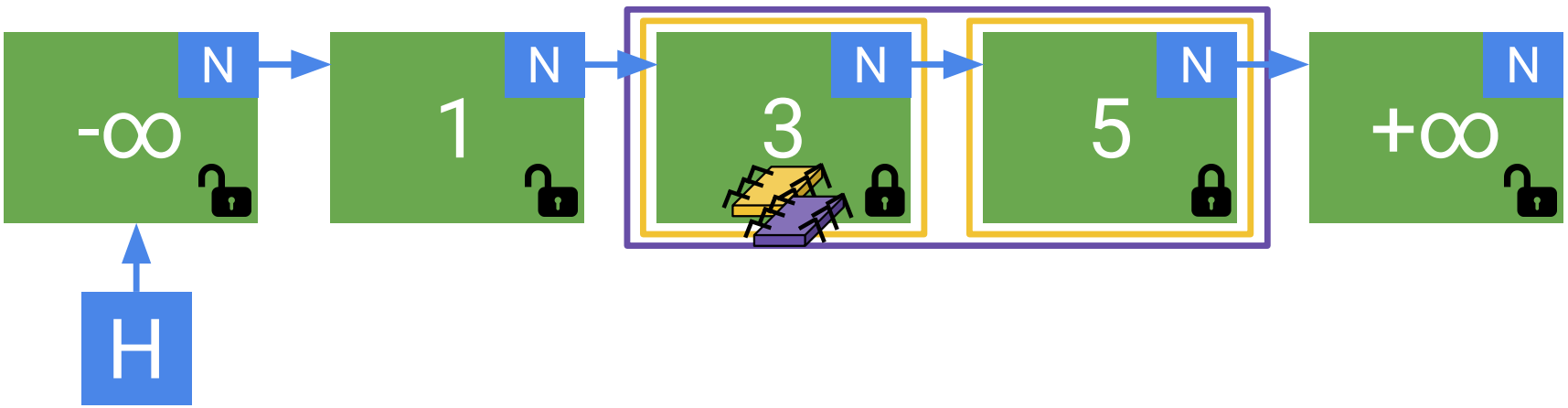
Проблема: cur.N != next

Оба добавляют «4»



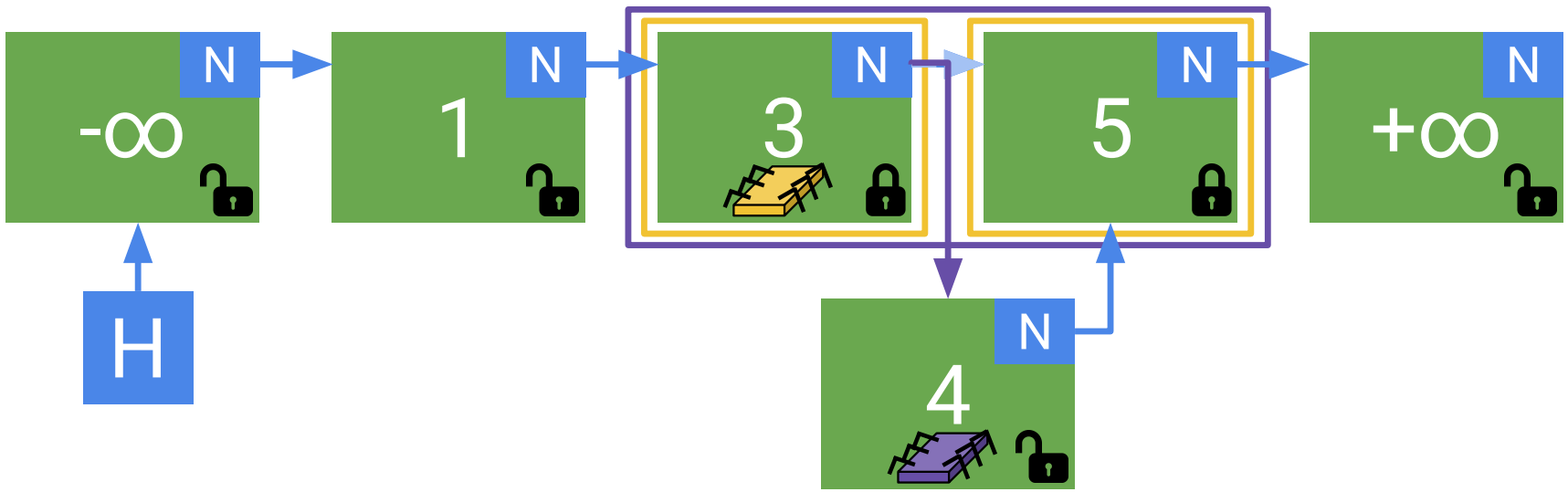
Проблема: cur.N != next

Оба добавляют «4»



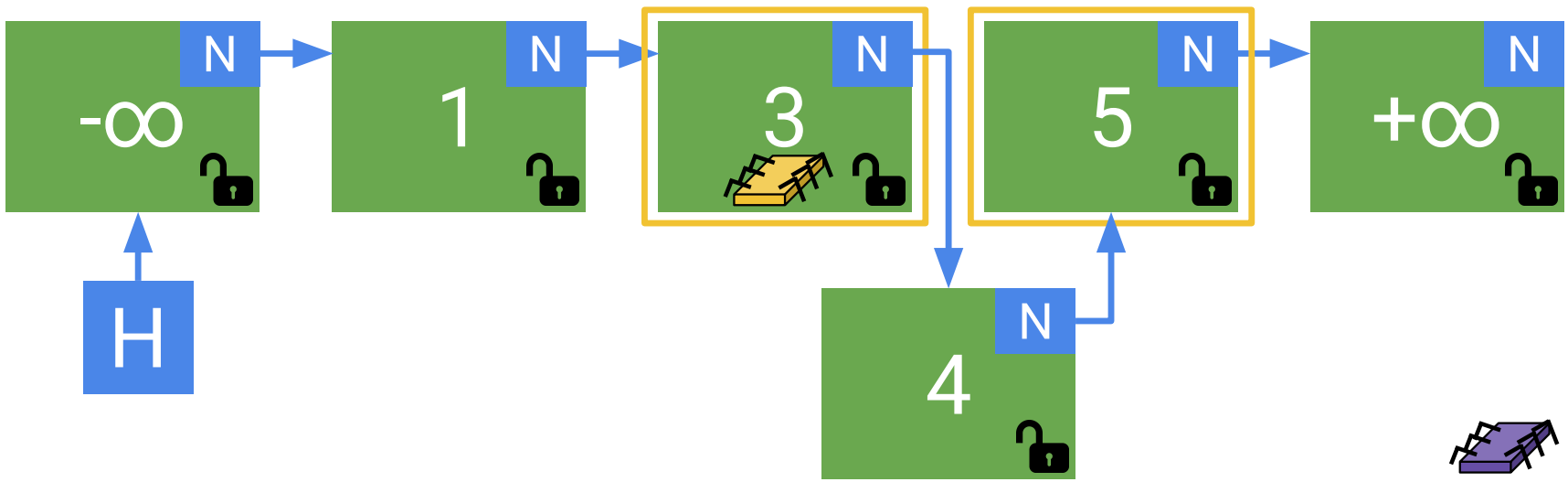
Проблема: cur.N != next

Оба добавляют «4»



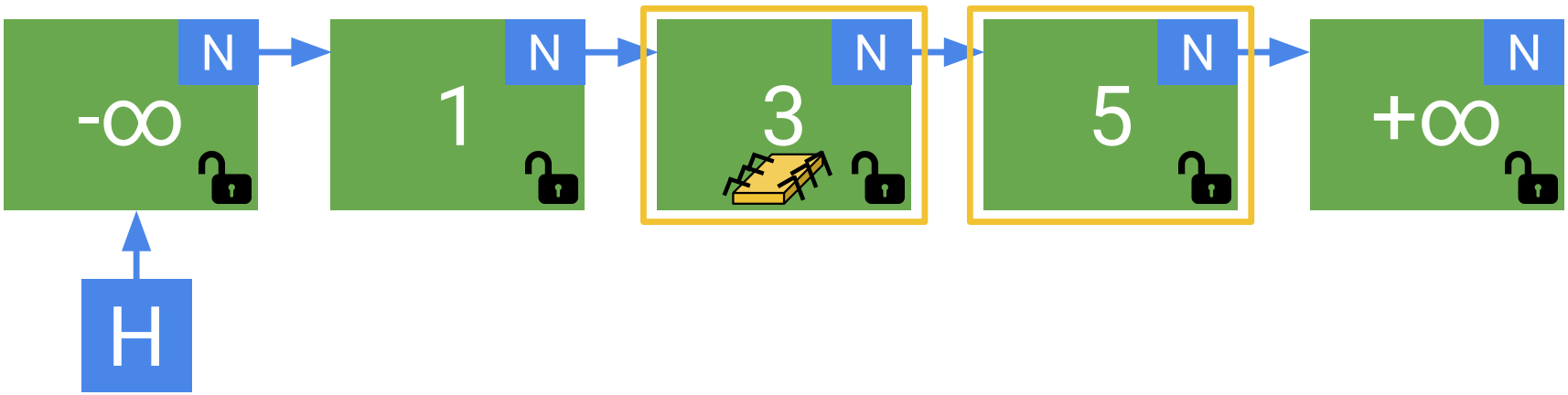
Проблема: cur.N != next

Оба добавляют «4»



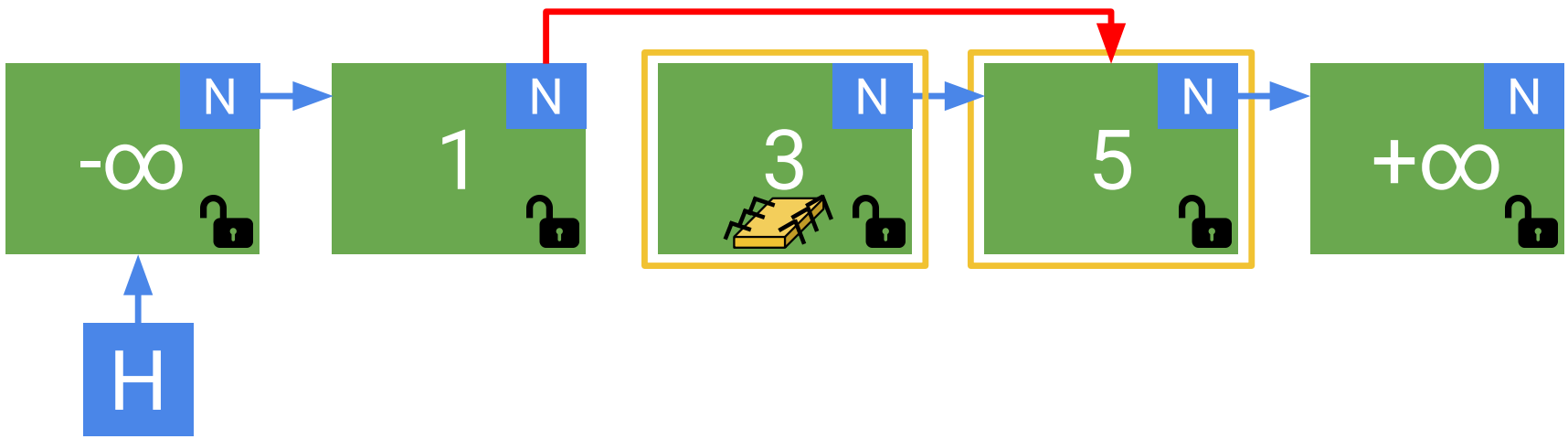
Проблема: cur уже удалили

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)
}
```

Оптимистичная синхронизация: задание

Задание «linked-list-set»: используем оптимистичную синхронизацию для поиска окна

Корректность

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

План

Ленивое удаление: идея

- Добавим в Node поле $\mathrm{boolean\ removed}$
- Удаление в две фазы:
  1. $\mathrm{node.removed} = \mathrm{true}$ - логическое удаление
  2. Физическое удаление из списка

Ленивое удаление: идея

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

Ленивое удаление: псевдокод

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

План

Неблокирующий поиск

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

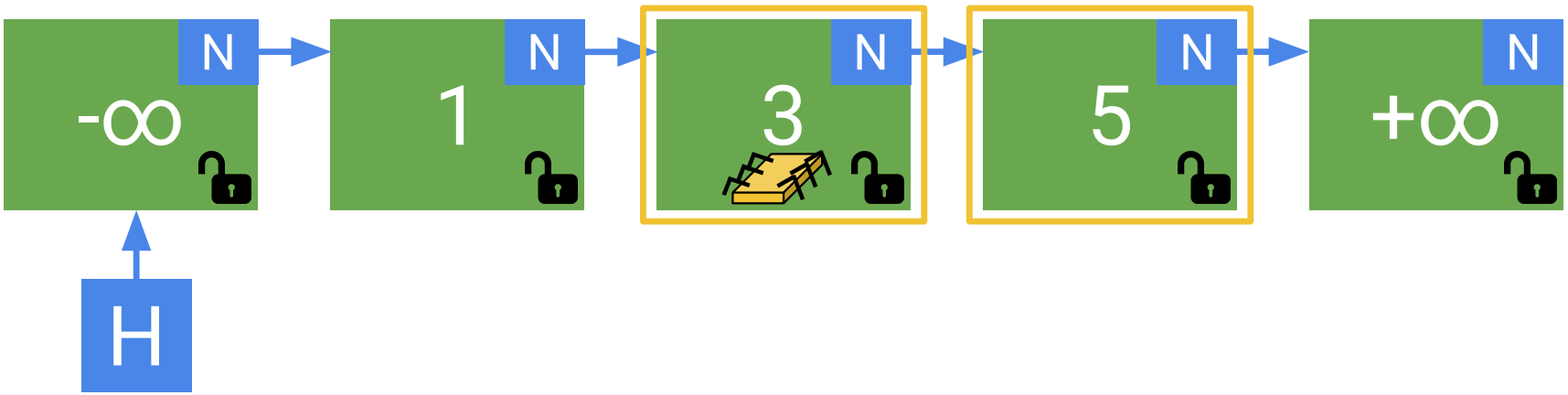
Неблокирующий поиск

- 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
}
```

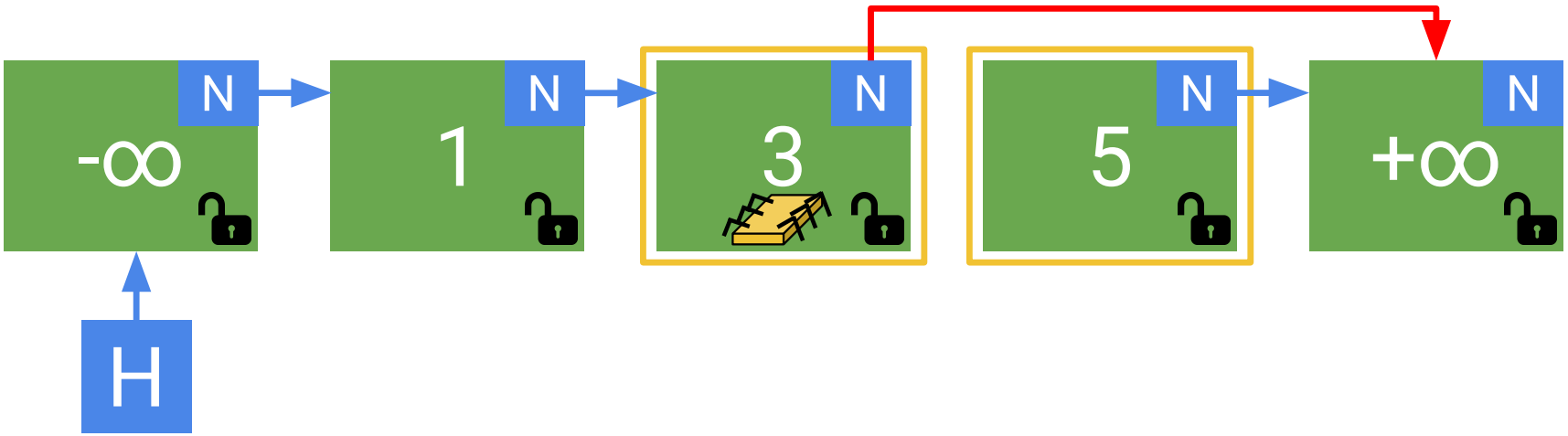
Пример

Жёлтый ищет «5», но его удаляют параллельно



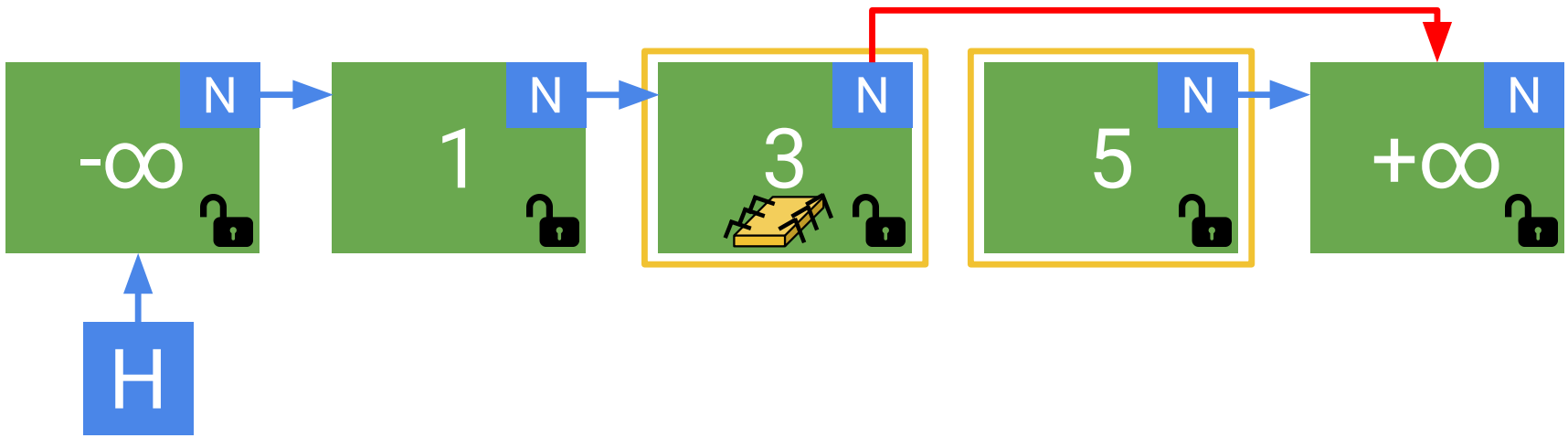
Пример

Жёлтый ищет «5», но его удаляют параллельно



Пример

Жёлтый ищет «5», но его удаляют параллельно



Выполняются параллельно

$\Rightarrow$ можем упорядочить как угодно

Ленивая синхронизация + неблокирующий поиск: задание

Задание «linked-list-set»: будем удалять лениво, добавьте флажок $\mathrm{removed}$ в Node. Не забудьте сделать next volatile, чтобы обеспечить корректную публикацию. Заодно сделаем поиск неблокирующим.

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, ...

CAS: универсальная конструкция

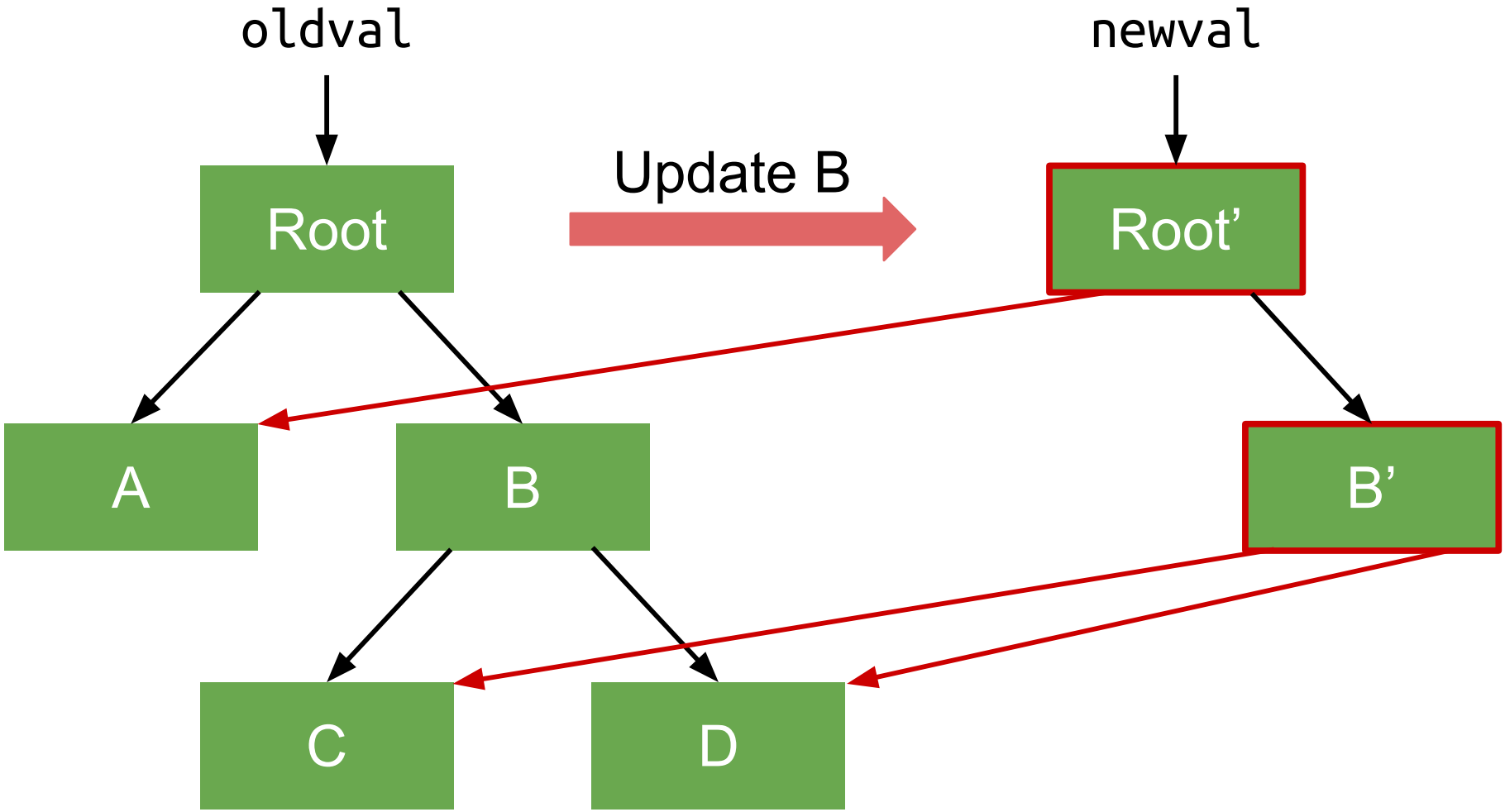
- r хранит указатель на данные
- deepCopy делает полную копию объекта
- получаем выполнение операции без блокировки

```
class ConcurrentXXX {
  var r: XXX

  fun concurrentOperationYYY(args) {
    while (true):
      val old = r
      val upd = old.deepCopy()
      val res = upd.operationYYY(args)
      if (r.CAS(old, upd)): // ничего не поменялось
        return res
  }
}
```

CAS: деревья

- Структура представлена в виде дерева
- Тогда операции можно реализовать в виде одного CAS, который заменяет указатель на root дерева
- Неизменившуюся часть дерева можно использовать в новой версии, т.е. не нужно копировать всю СД
- Это т.н. персистентные структуры данных

Персистентная структура данных



CAS: задание

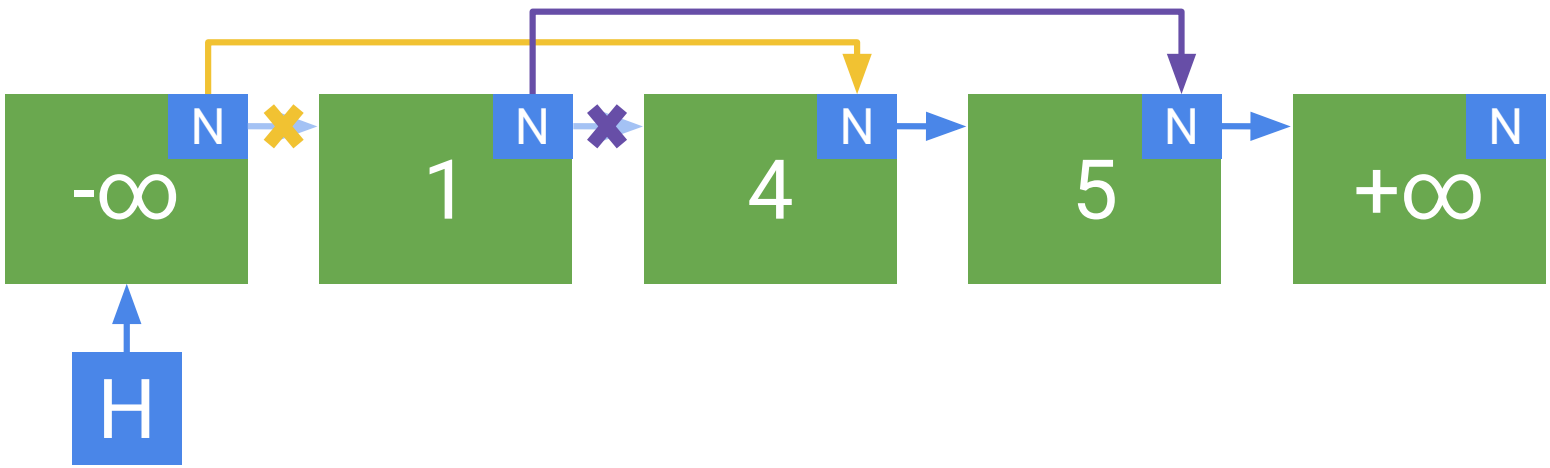
Задание «stack»: сделайте реализацию стека lock-free с использованием AtomicReference

Неблокирующая модификация

- Вернёмся к множеству на односвязном списке

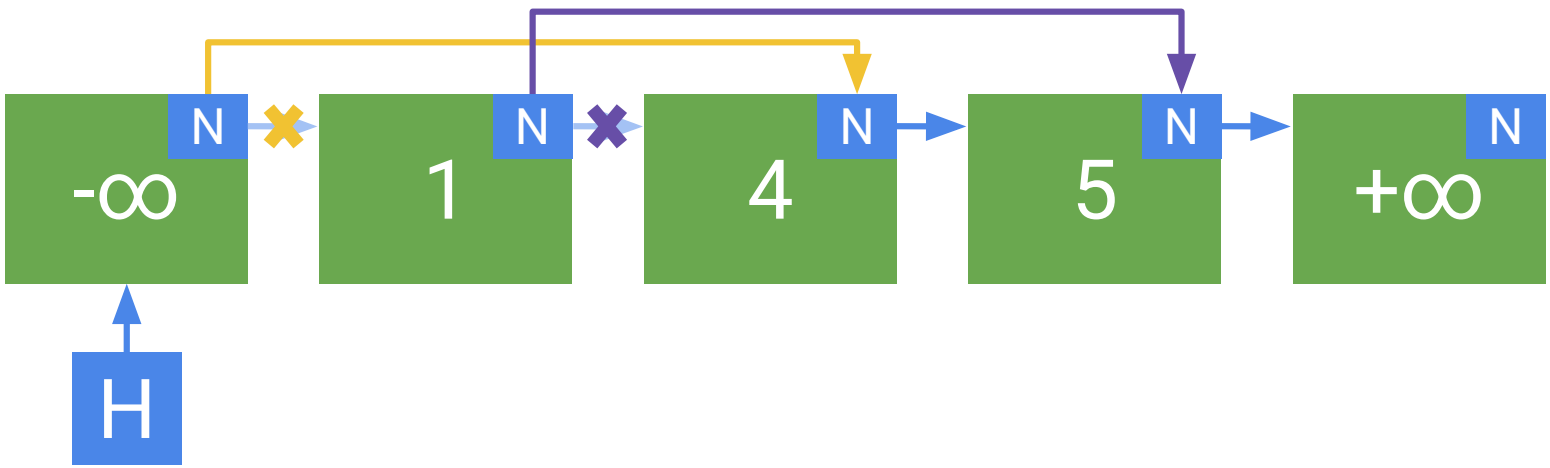
Неблокирующая модификация

- 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 $\mathrm{removed}$ в одну переменную, пару $(\mathrm{N}, \mathrm{removed})$
- Будем менять $(\mathrm{N}, \mathrm{removed})$ атомарно
- Каждая операция модификации будет выполняться одним успешным CAS-ом
- В Java для этого есть $\mathrm{AtomicMarkableReference}$

Узел

- AtomicMarkableReference работает как volatile

```
data class Node(
  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
}
```

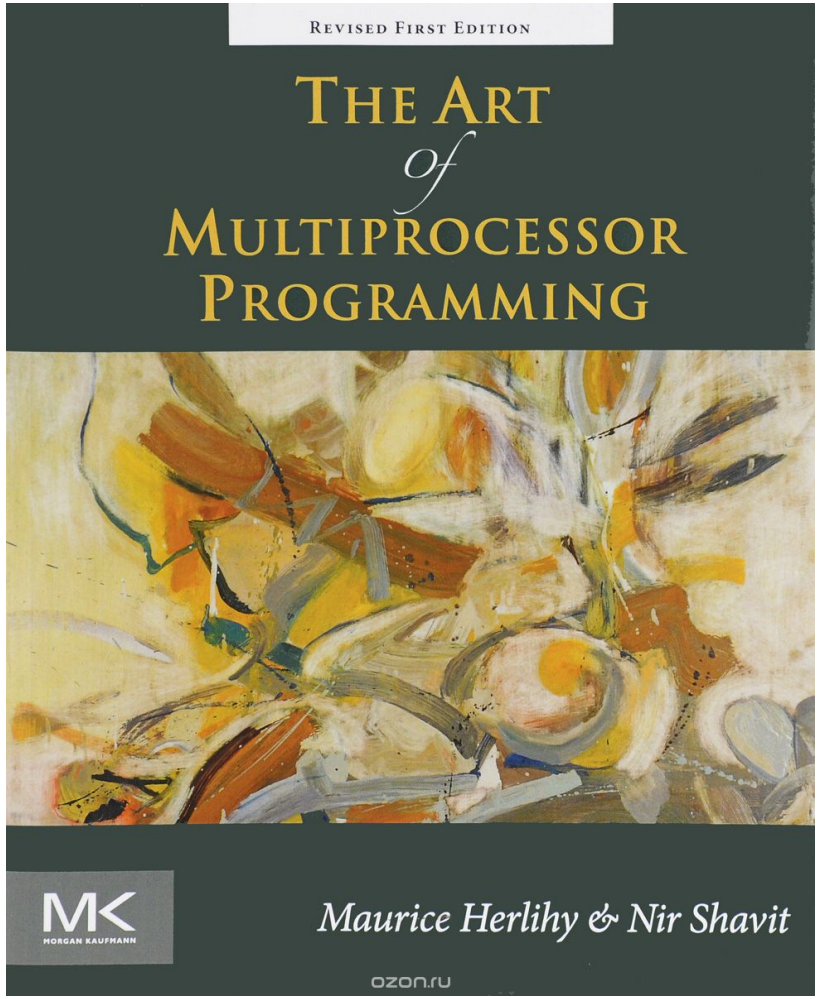
Удаление

```
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
}
```

План

1. Мотивация
2. Блокировки
3. Множество на односвязном списке
4. Грубая синхронизация
5. Тонкая синхронизация
6. Оптимистичная синхронизация
7. Ленивая синхронизация
8. Неблокирующая синхронизация
9. Подведём итоги

Литература



The Art of Multiprocessor Programming
yeby M. Herlihy and N. Shavit

Домашнее задание

- Lock-free очередь by Michael & Scott
- Можно найти в книге «The Art of Multiprocessor Programming»
- В этом алгоритме потоки помогают друг другу
- Разберитесь с алгоритмом, реализуйте и протестируйте (см. тесты из заданий)

Спасибо за внимание!

Построение многопоточных алгоритмов с различными типами синхронизации

Никита Коваль

Research engineer, Devexperts
ndkoval@ya.ru

Пермь, 2017