Lock-free Treiber Stack and Michael-Scott Queue

Nikita Koval[1] Roman Elizarov[2]

[1]Researcher, JetBrains
ndkoval@ya.ru

[2]Kotlin Team Lead, JetBrains
elizarov@gmail.com

ITMO 2020

План

1. Неблокирующиеся алгоритмы
2. Treiber Lock-Free Stack
3. Michael-Scott Lock-Free Queue

План

1. Неблокирующиеся алгоритмы
2. Treiber Lock-Free Stack
3. Michael-Scott Lock-Free Queue

Mutual exclusion

- Aka *mutex* и *lock*
- Только один поток может “держать” блокировку

Mutual exclusion

- Aka *mutex* и *lock*
- Только один поток может “держать” блокировку

```
l.lock()
// critical section here
l.unlock()
```

Atomic counter: mutex

```
val l = Lock()
var Counter = 0

fun incAndGet(): Int {
    l.lock()
    try {
        return ++Counter
    } finally {
        l.unlock()
    }
}
```

Atomic counter: mutex

```
val l = Lock()
var Counter = 0

fun incAndGet(): Int {
    l.lock()
    try {
        return ++Counter
    } finally {
        l.unlock()
    }
}
```

Нет гарантии прогресса...

Lock-freedom

- *Lock-freedom* гарантирует прогресс в системе
- Базовый "кирпичик": *Compare-And-Set* (CAS)
  - $\mathrm{CAS}(\mathrm{p}, \mathrm{old}, \mathrm{new})$ атомарно проверяет, что значение по адресу p совпадает с old и заменяет его на new

Lock-freedom

- *Lock-freedom* гарантирует прогресс в системе
- Базовый "кирпичик": *Compare-And-Set* (CAS)
  - CAS(p, old, new) атомарно проверяет, что значение по адресу p совпадает с old и заменяет его на new

```
var Counter = 0

fun incAndGet(): Int {
  while (true) { // CAS loop
    cur := Counter
    if (CAS(&Counter, cur, cur + 1))
      return cur + 1
  }
}
```

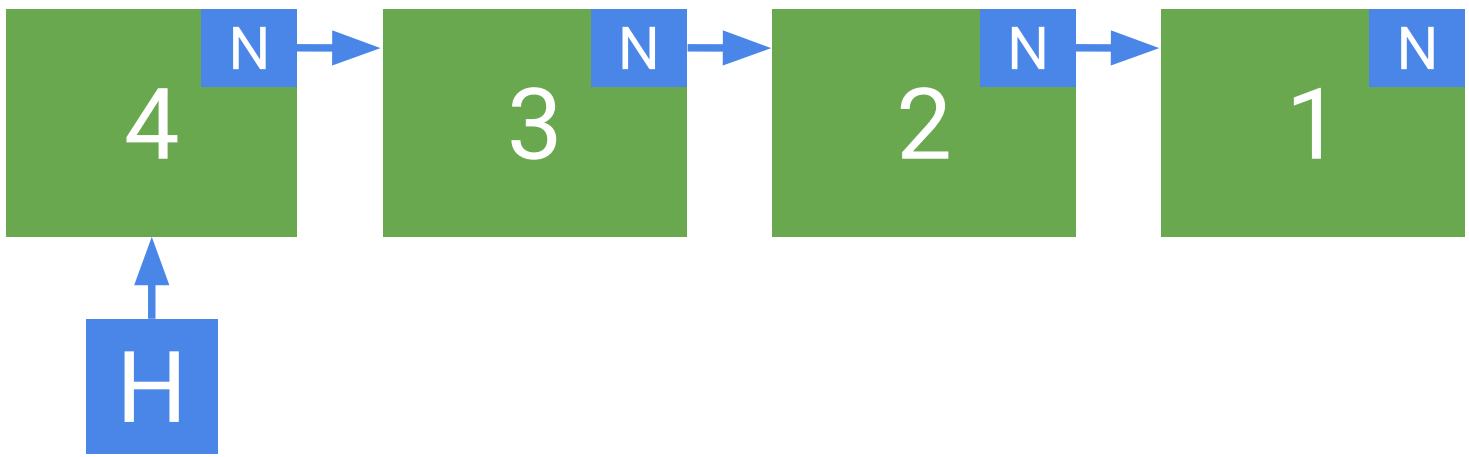
Неудачный CAS $\Rightarrow$ кто-то выполнил incAndGet()

План

1. Неблокирующиеся алгоритмы
2. Treiber Lock-Free Stack
3. Michael-Scott Lock-Free Queue

Структура стека

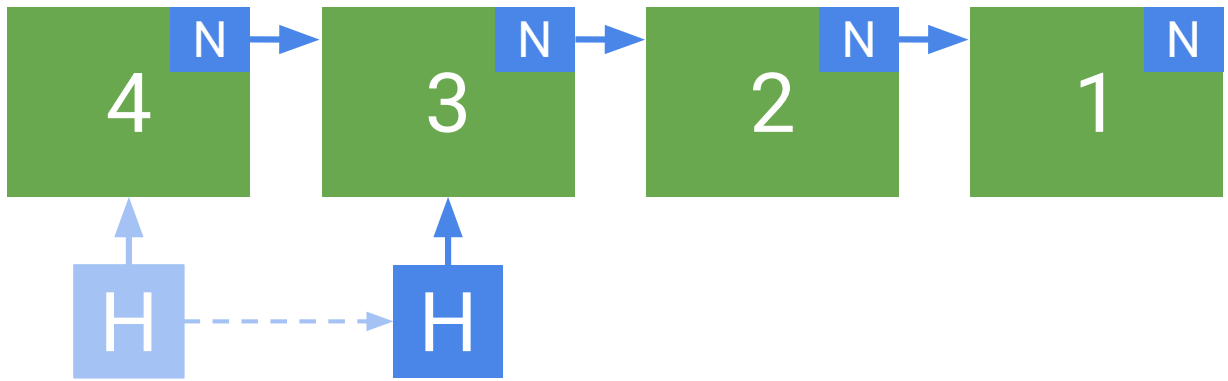
Указатель на голову стека H (изменяется CAS-ом)



Очередь пуста $\Leftrightarrow$ H указывает на null

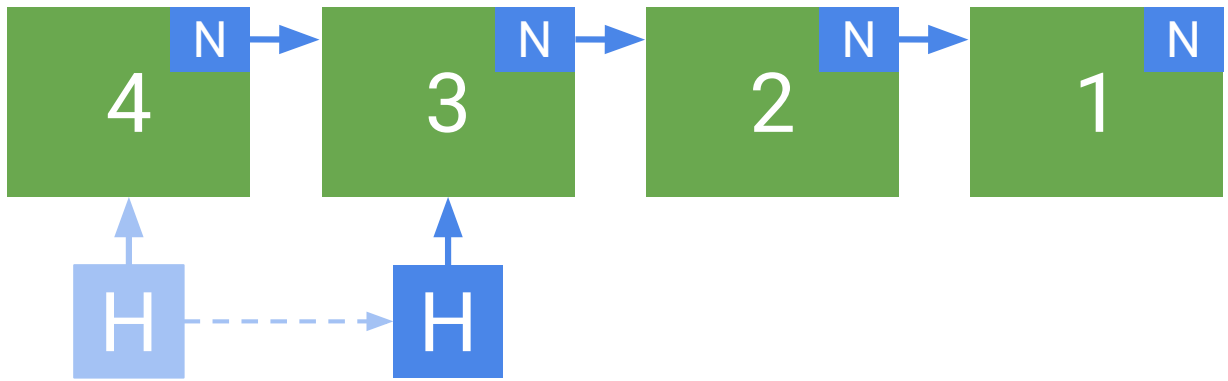
Удаление из стека

Перемещаем указатель на голову стека вперёд



Удаление из стека

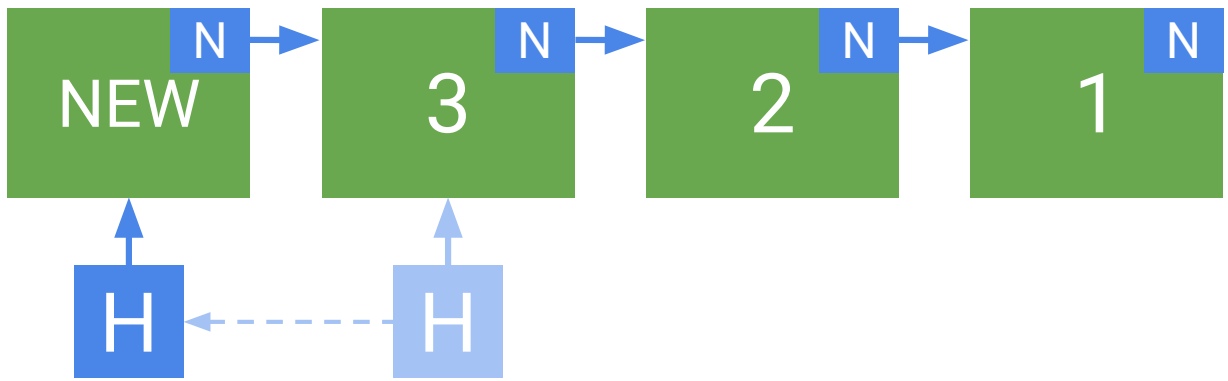
Перемещаем указатель на голову стека вперёд




```
fun pop(): Int {
    while (true) { // CAS loop
        head := H
        if (CAS(&H, head, head.N))
            return head.Value
    }
}
```

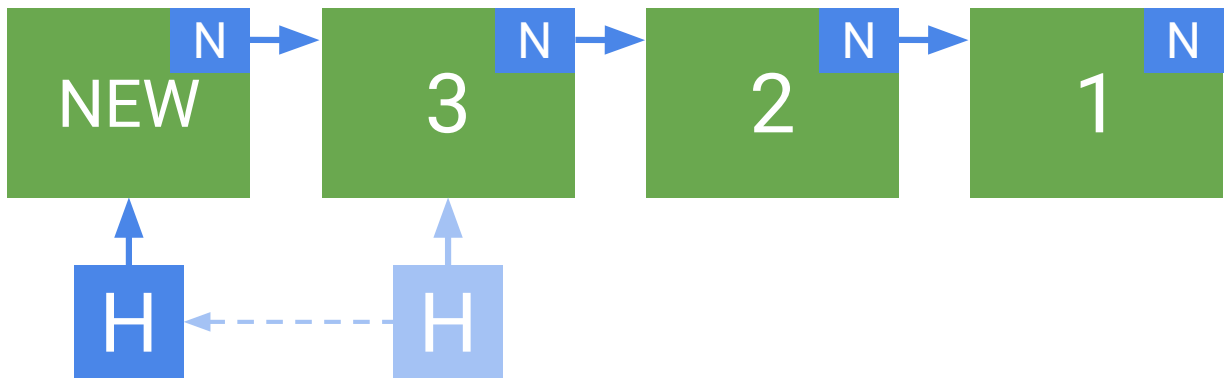
Добавление в стек

Создаем вершину с $\mathrm{N} = \mathrm{H}$ и обновляем H



Добавление в стек

Создаем вершину с N = H и обновляем H




```
fun push(x: Int) {
    while (true) { // CAS loop
        head := H
        newHead := Node {Value: x, N: head}
        if (CAS(&H, head, newHead))
            return
    }
}
```

Задачка

Как реализовать метод top, который возвращает значение с головы стека (не удаляя его)?

Задачка

Как реализовать метод top, который возвращает значение с головы стека (не удаляя его)?

Решение: максимально прямолинейно

```
fun top(): Int? {
    head := H
    if head == null: return null
    return head.Value
}
```

Elimination

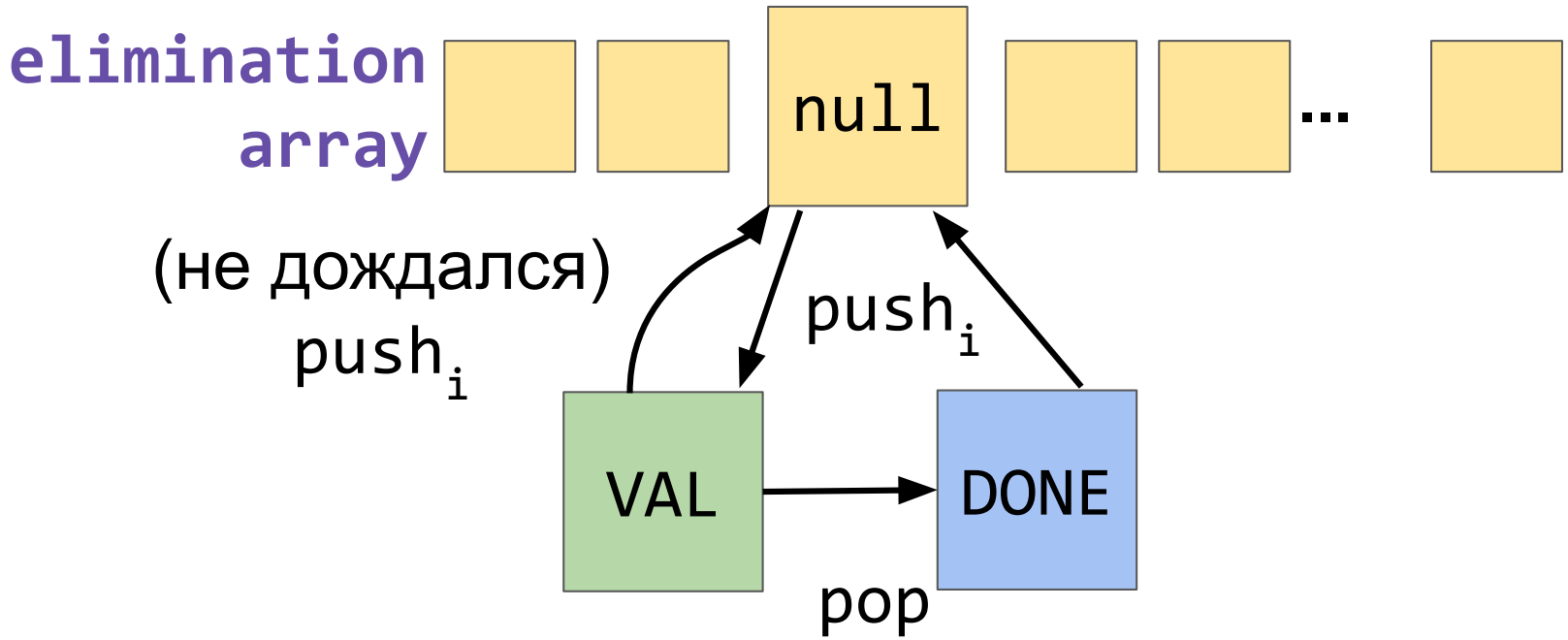
Хорошо ли масштабируется этот алгоритм?

Elimination

Хорошо ли масштабируется этот алгоритм?

Если параллельно выполняются много push и pop (нагрузка сбалансирована), почему бы им не встретиться на нейтральной территории

Elimination for Stack



Используем предыдущий алгоритм, если оптимизация не получилась

Elimination for Stack

Обе операции при выборе случайной ячейки пробуют посмотреть на соседние при неудачи; это “дешево”, т.к. они скорее всего закешированы

На практике, elimination должен быть адаптивным, чтобы не тратить время на эту “оптимизацию” впустую и выбрать наиболее оптимальный размер массива

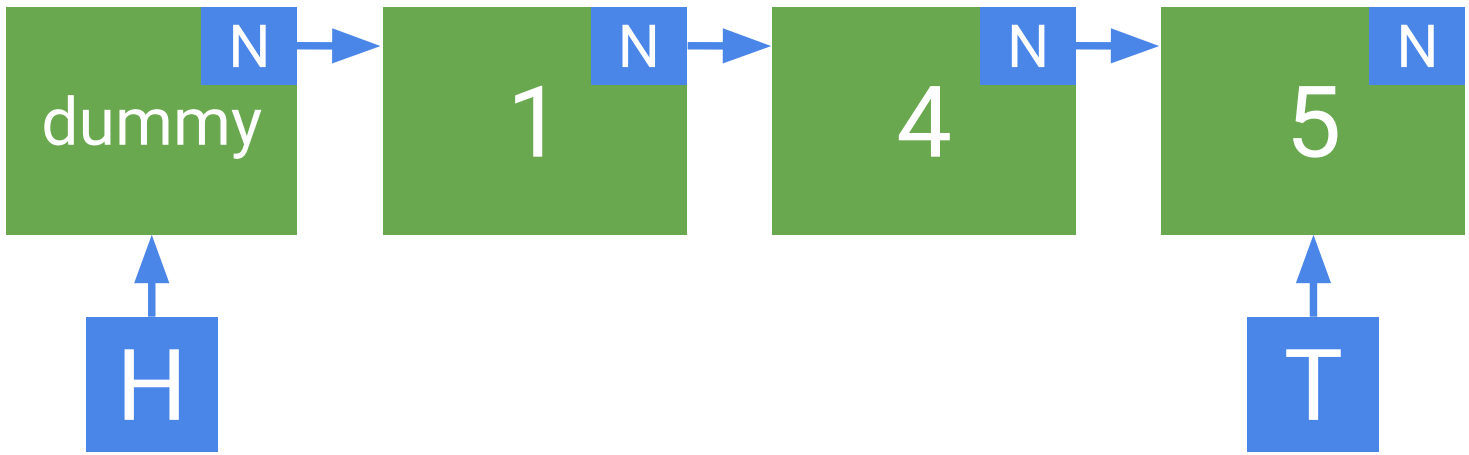
План

1. Неблокирующиеся алгоритмы
2. Treiber Lock-Free Stack
3. Michael-Scott Lock-Free Queue

Lock-free queue by Maged M. Michael and Michael L. Scott, PODC'96

Структура очереди

Два указателя: на голову очереди H и на хвост T

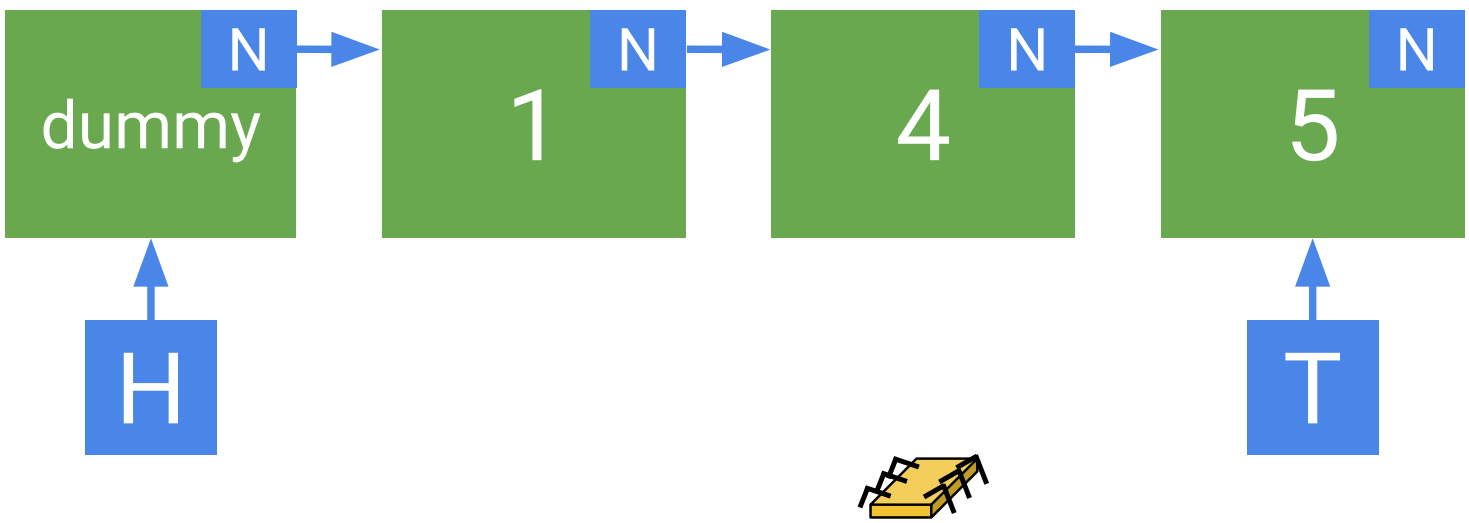


Очередь пуста $\Leftrightarrow$ голова и хвост указывают на один элемент*

*Изначально состоит из одного фиктивного элемента (dummy)

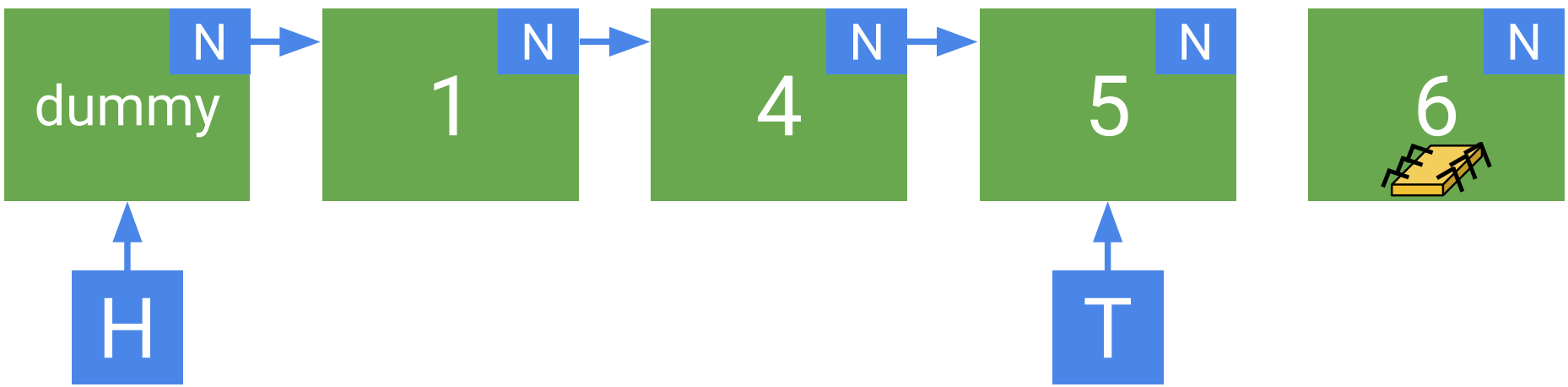
Добавление в очередь

1. Создадим новый хвост
2. Поменяем указатель следующего элемента у хвоста: $\mathrm{T.N} = \mathrm{newTail}$
3. Заменим хвост на новый: $\mathrm{T} = \mathrm{newTail}$



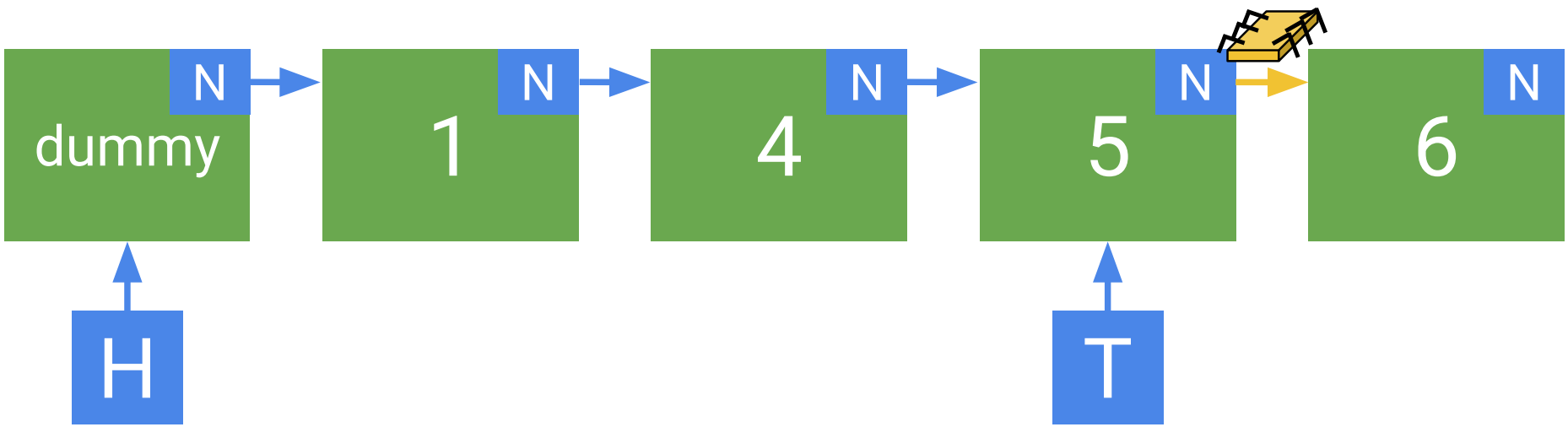
Добавление в очередь

1. Создадим новый хвост
2. Поменяем указатель следующего элемента у хвоста: $\mathrm{T.N} = \mathrm{newTail}$
3. Заменим хвост на новый: $\mathrm{T} = \mathrm{newTail}$



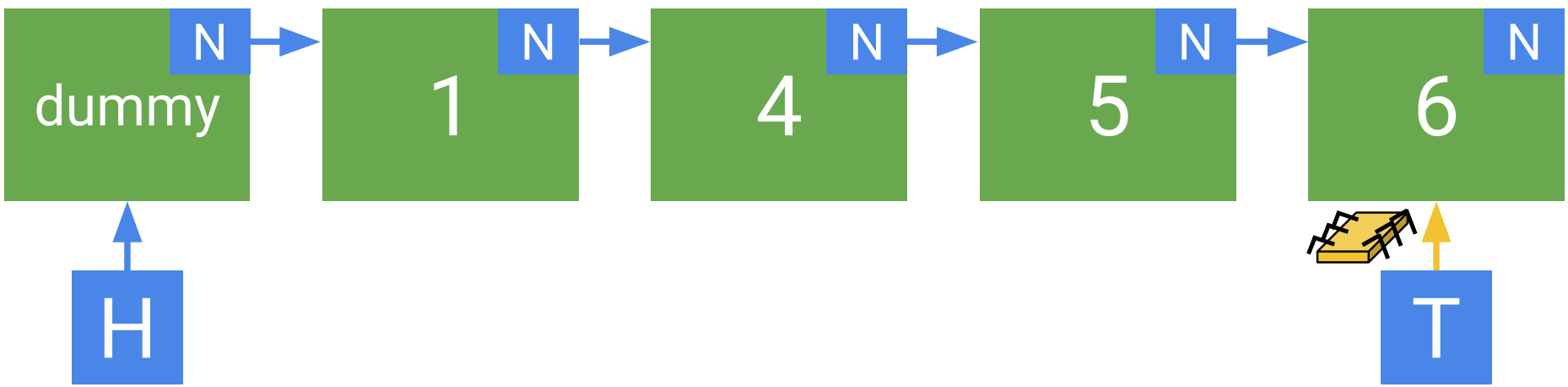
Добавление в очередь

1. Создадим новый хвост
2. Поменяем указатель следующего элемента у хвоста: $\mathrm{T.N} = \mathrm{newTail}$
3. Заменим хвост на новый: $\mathrm{T} = \mathrm{newTail}$



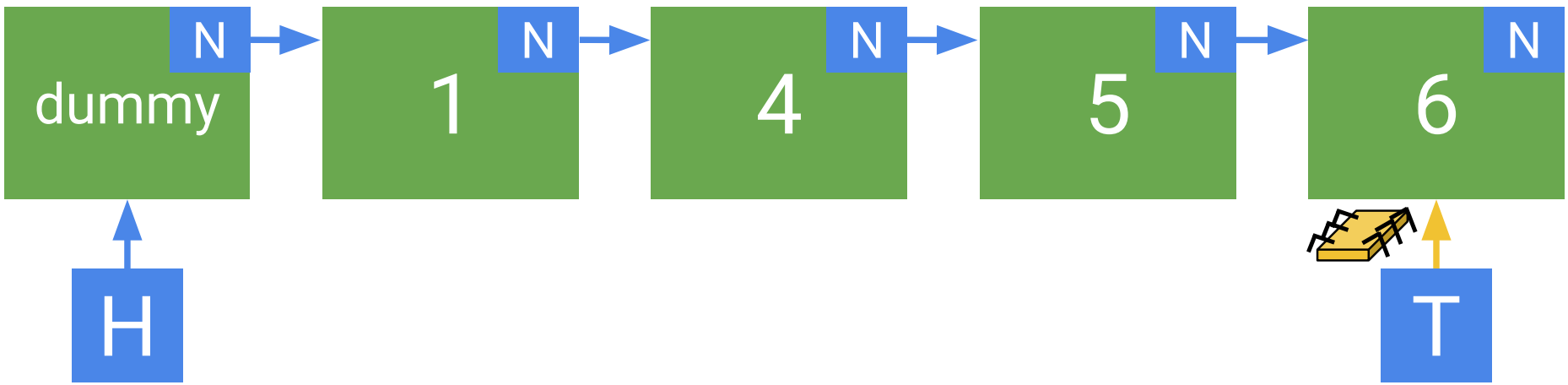
Добавление в очередь

1. Создадим новый хвост
2. Поменяем указатель следующего элемента у хвоста: $\mathrm{T.N} = \mathrm{newTail}$
3. Заменим хвост на новый: $\mathrm{T} = \mathrm{newTail}$



Добавление в очередь

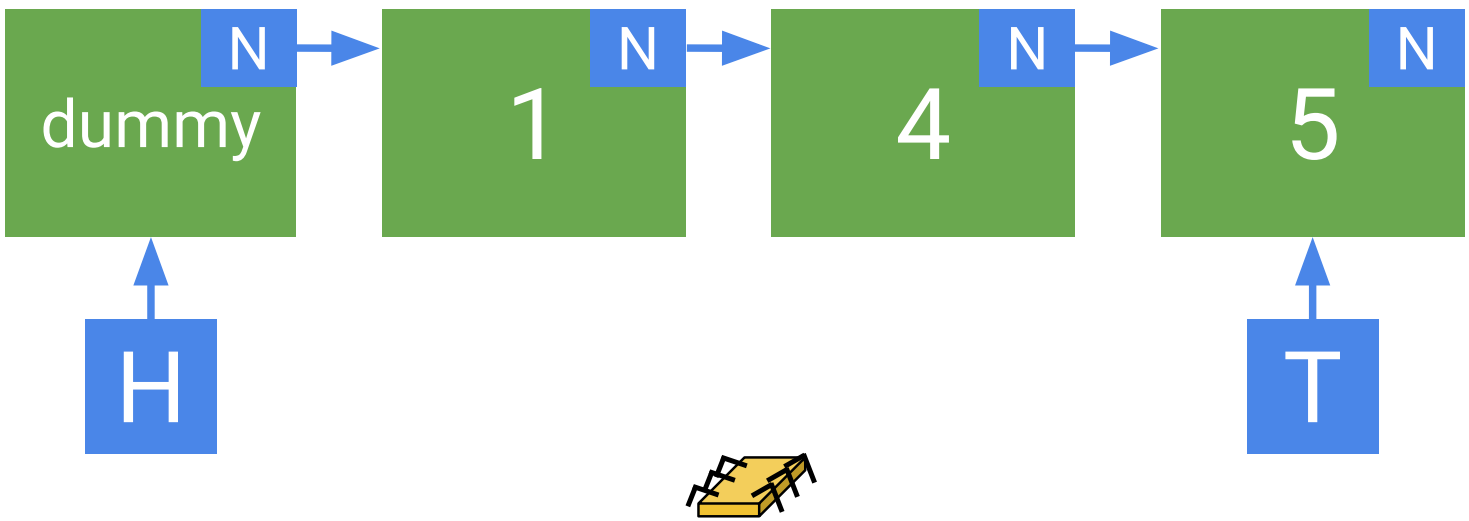
1. Создадим новый хвост
2. Поменяем указатель следующего элемента у хвоста: $\mathrm{T.N} = \mathrm{newTail}$
3. Заменим хвост на новый: $\mathrm{T} = \mathrm{newTail}$



Хочется делать 2 и 3 атомарно

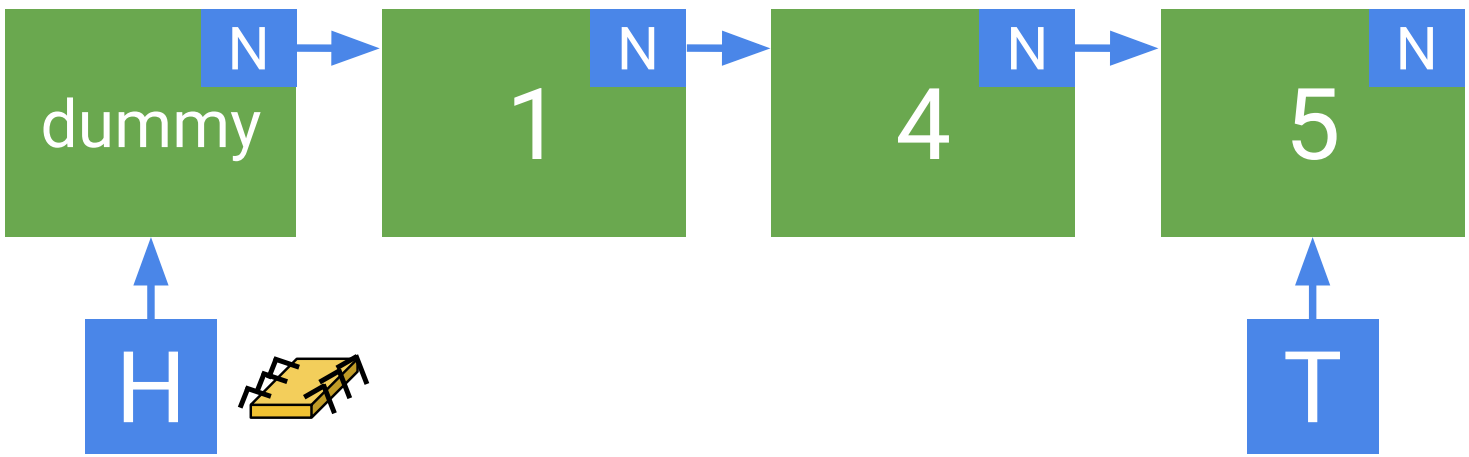
Удаление из очереди

1. Перенесём указатель на голову вправо: $\mathrm{H} = \mathrm{H.N}$



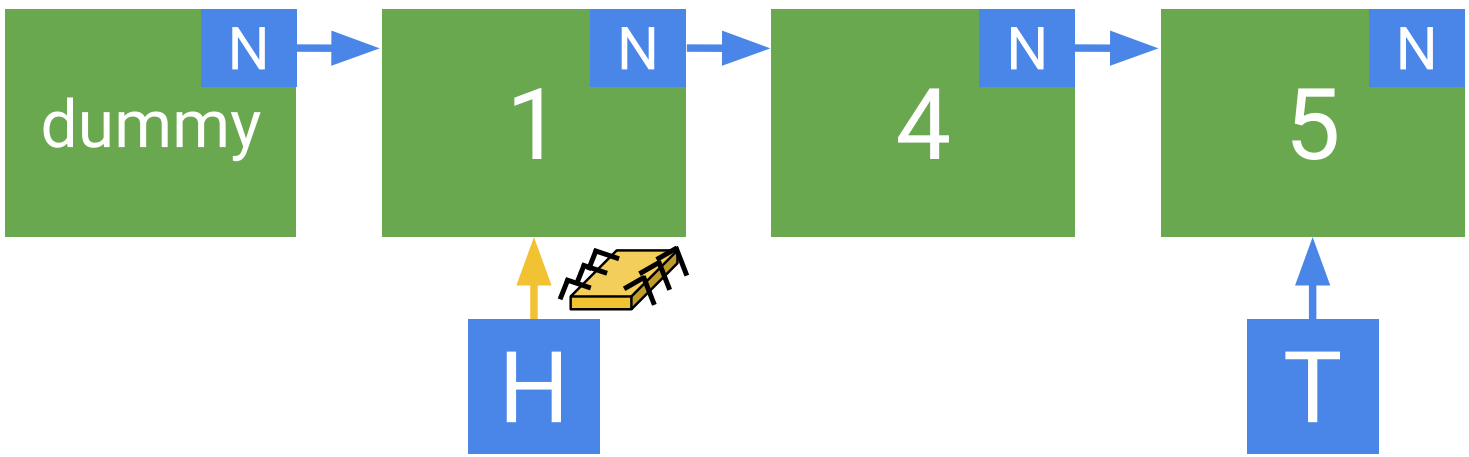
Удаление из очереди

1. Перенесём указатель на голову вправо: $\mathrm{H} = \mathrm{H.N}$



Удаление из очереди

1. Перенесём указатель на голову вправо: $\mathrm{H} = \mathrm{H.N}$



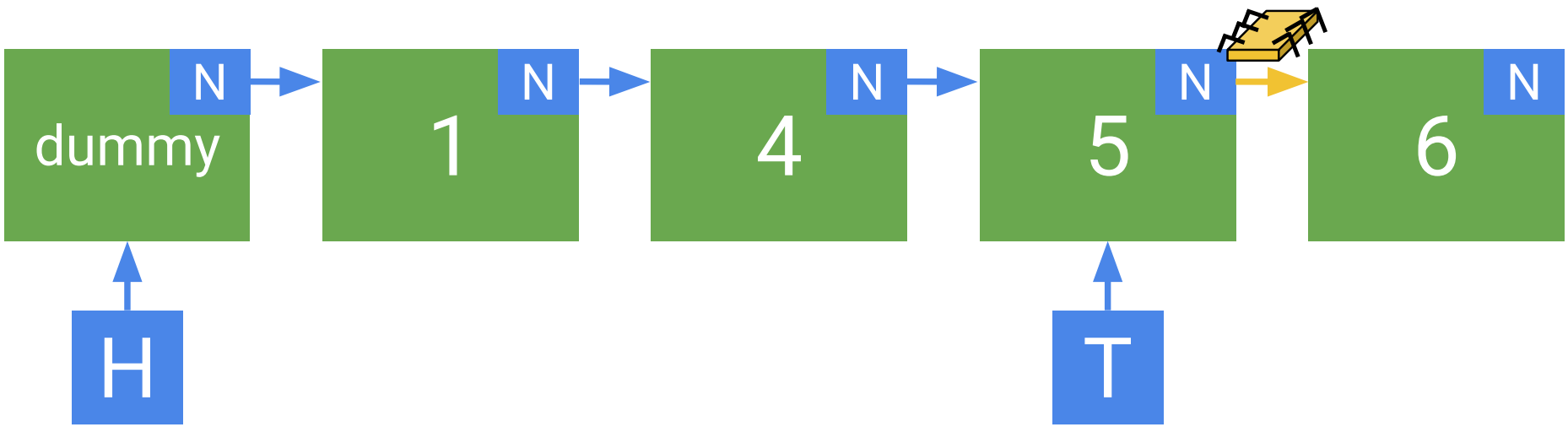
Возвращаем «1», теперь это новый «фиктивный» элемент

Многопоточность: используем CAS

- Будем на всех переходах использовать CAS

Многопоточность: используем CAS

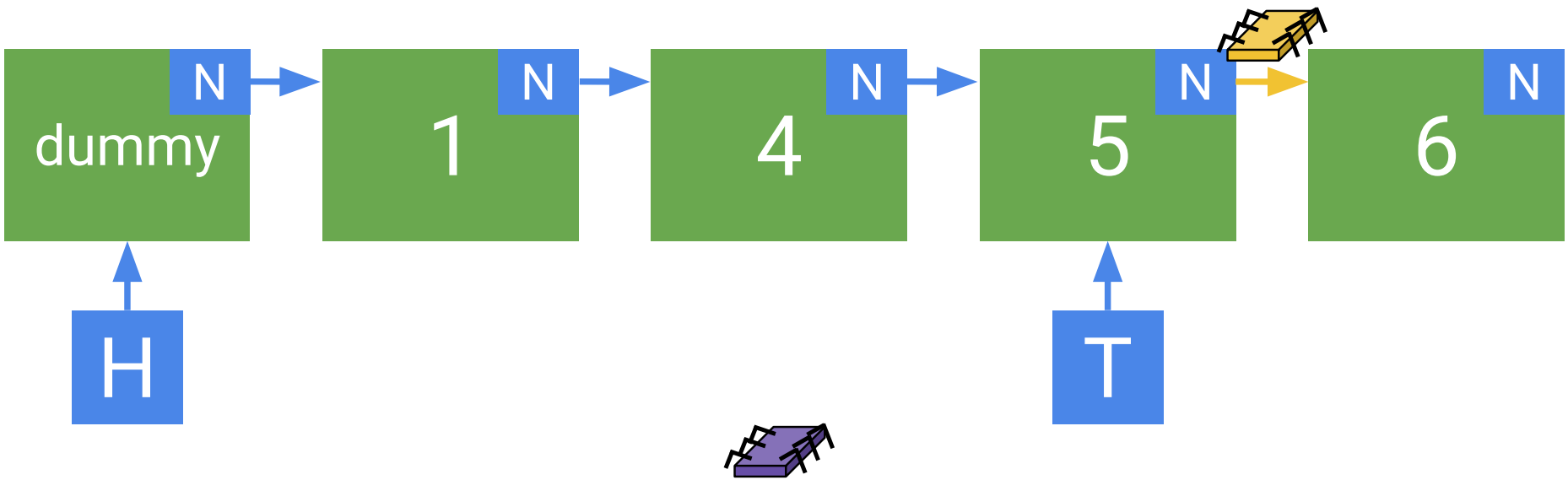
- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Жёлтый поменял $\mathrm{T.N}$, но не успел поменять указатель на хвост

Многопоточность: используем CAS

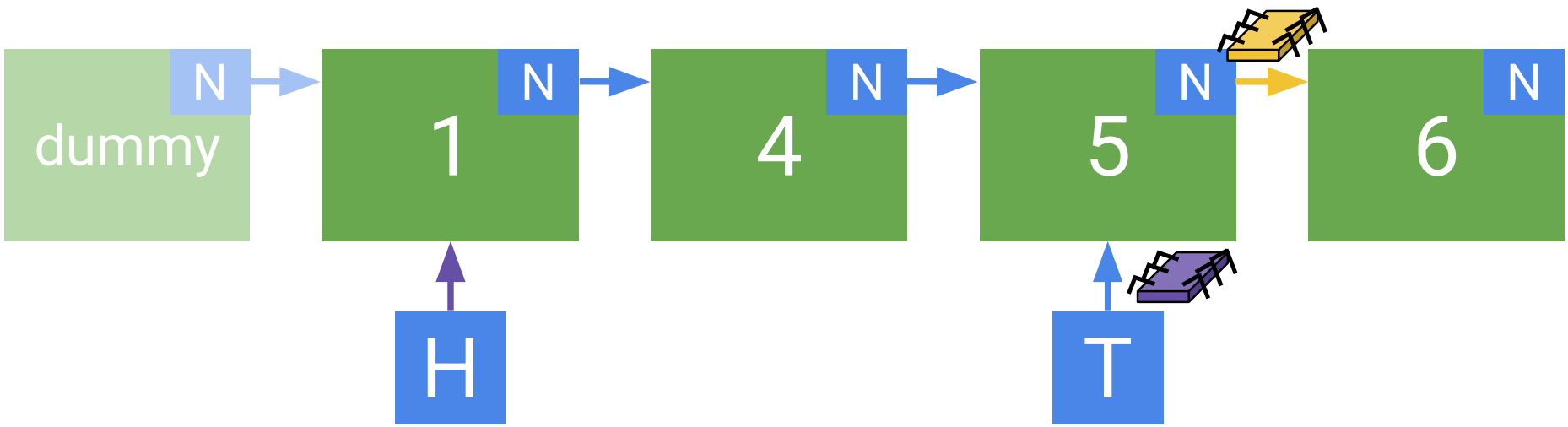
- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Пришёл фиолетовый поток, тоже хочет добавить элемент

Многопоточность: используем CAS

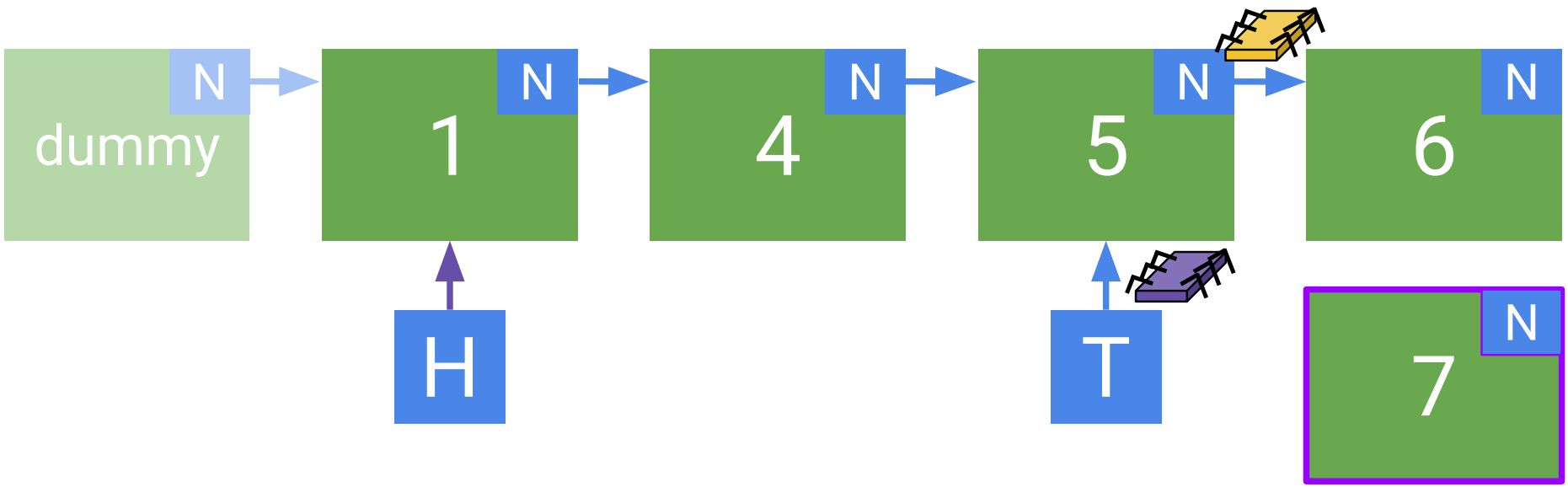
- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Создаёт новую вершину

Многопоточность: используем CAS

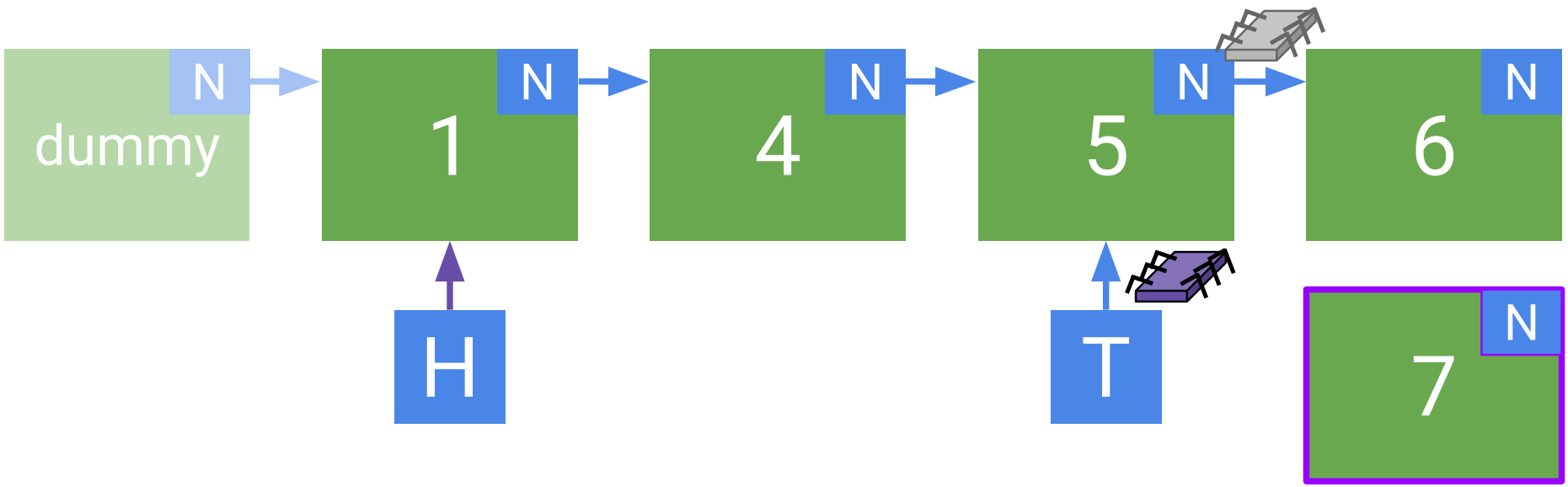
- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Но не может поменять $\mathrm{T.N}$, т.к. следующий элемент уже записан!

Многопоточность: используем CAS

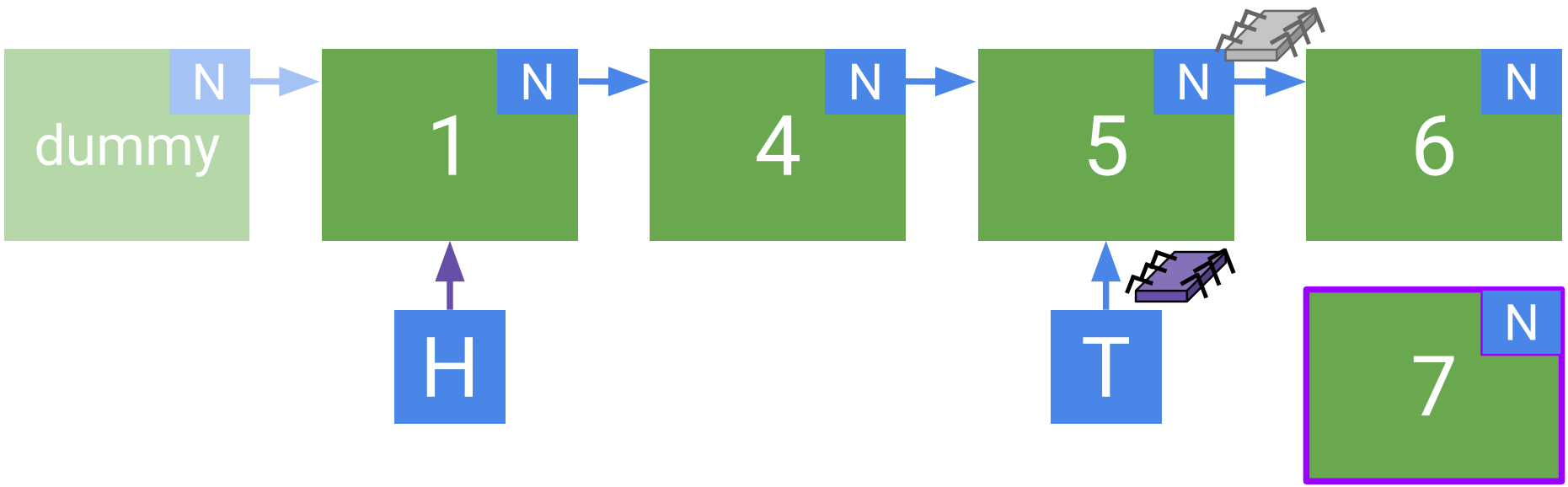
- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Получили блокирующийся алгоритм...

Многопоточность: используем CAS

- Будем на всех переходах использовать CAS
- Проблема: enqueue все равно не атомарен.
- Жёлтый добавляет «6», фиолетовый — «7»



Так жить нельзя!

Michael-Scott Lock-Free Queue

Основная идея

- Нельзя просто так взять и выполнить $\mathrm{T.N} = \mathrm{newTail}$ и $\mathrm{T} = \mathrm{newTail}$ атомарно

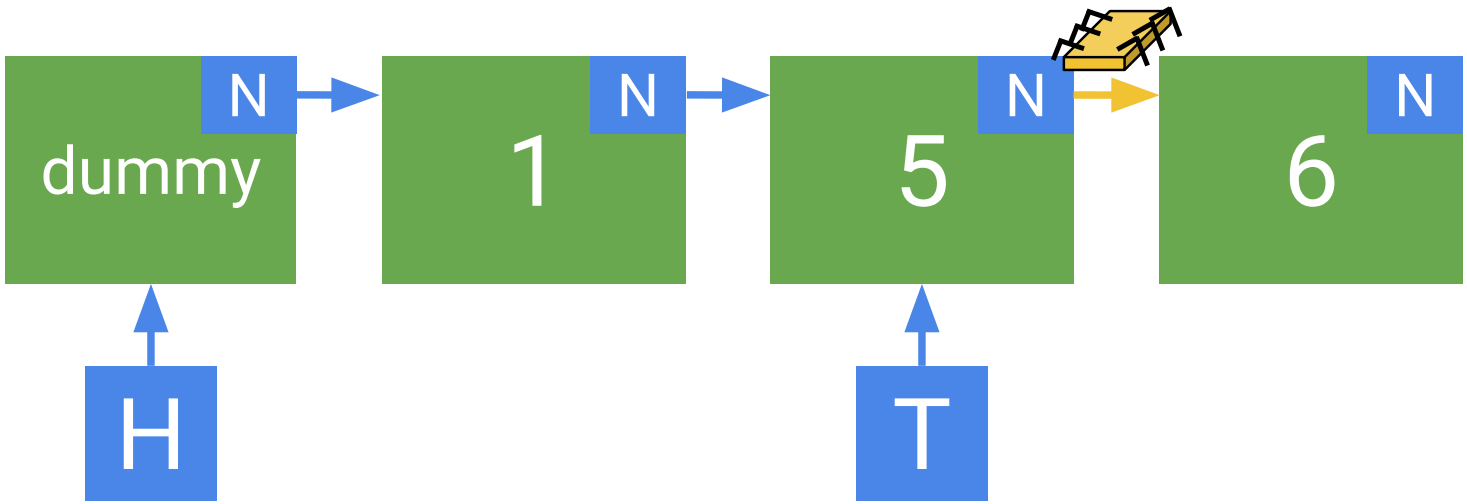
Michael-Scott Lock-Free Queue

Основная идея

- Нельзя просто так взять и выполнить $\mathrm{T.N} = \mathrm{newTail}$ и $\mathrm{T} = \mathrm{newTail}$ атомарно
- Пусть другие потоки помогают перенести ссылку на хвост, если видят непустой $\mathrm{T.N}$

Починим проблему

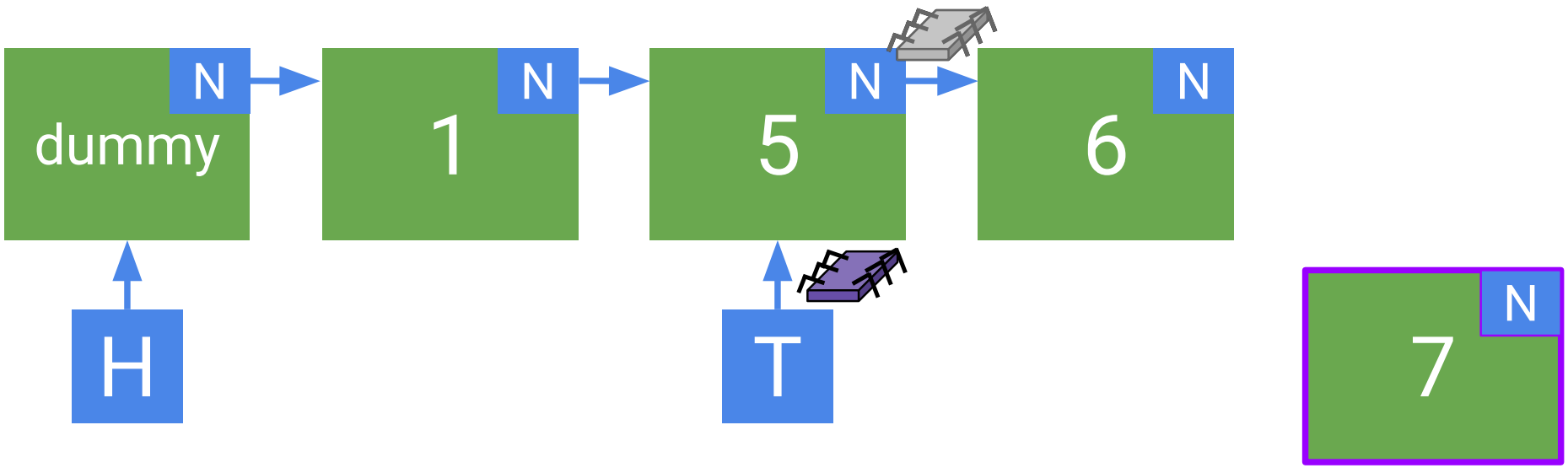
Жёлтый добавляет «6», фиолетовый — «7»



Жёлтый поменял $\mathrm{T.N}$, но не успел поменять указатель на хвост

Починим проблему

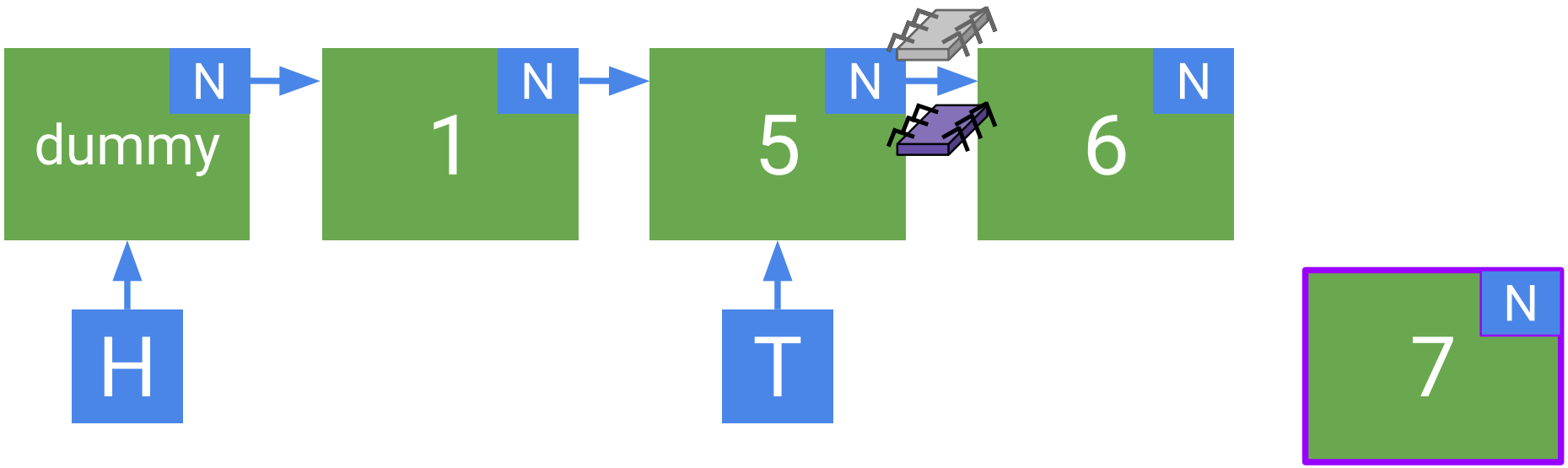
Жёлтый добавляет «6», фиолетовый — «7»



Фиолетовый увидел ещё старый хвост и создал новую вершину

Починим проблему

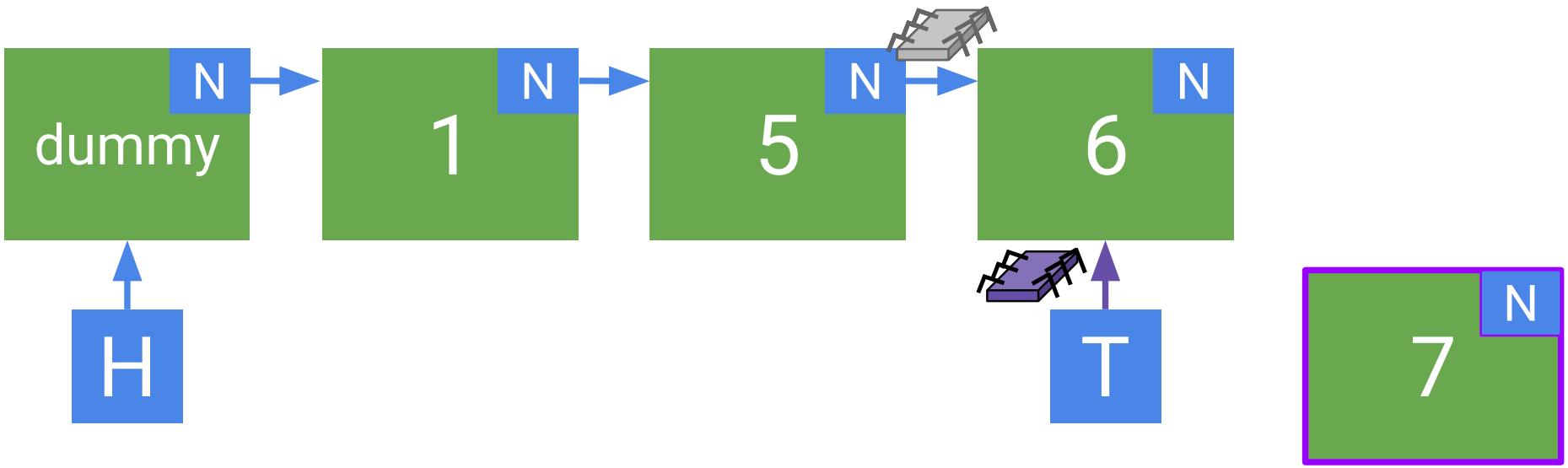
Жёлтый добавляет «6», фиолетовый — «7»



Неудачно делает $\mathrm{CAS}(\&\mathrm{tail.N}, \mathrm{null}, \mathrm{newTail}) \Rightarrow$ кто-то не до конца добавил вершину

Починим проблему

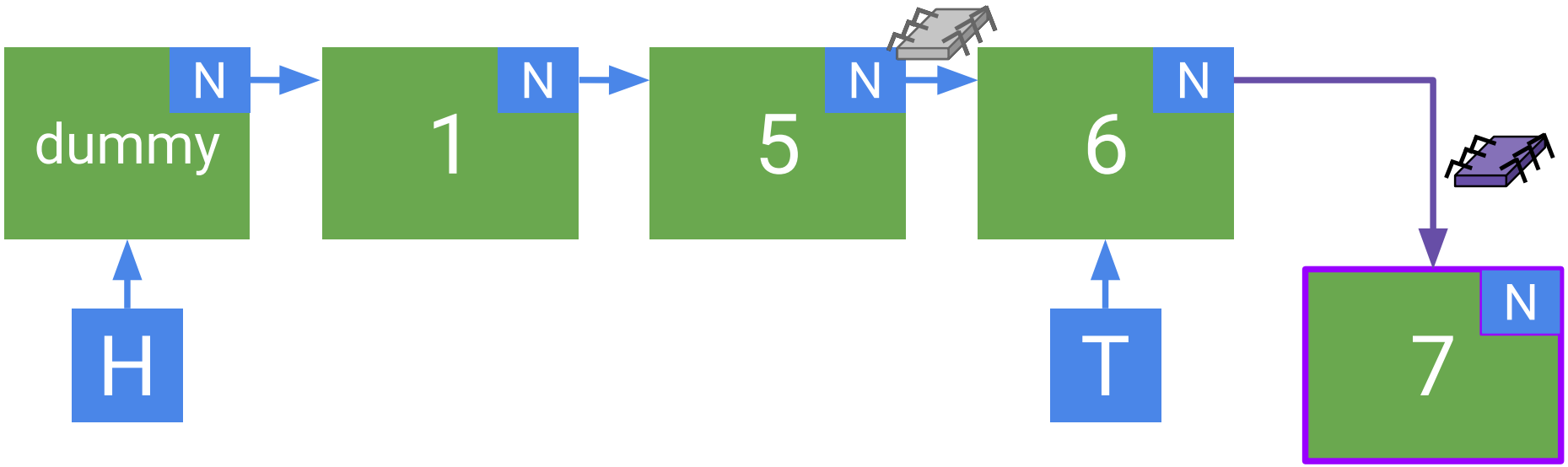
Жёлтый добавляет «6», фиолетовый — «7»



Помогает перенести указатель на хвост, делая $\mathrm{CAS}(\&\mathrm{T}, \mathrm{tail}, \mathrm{tail.N})$

Починим проблему

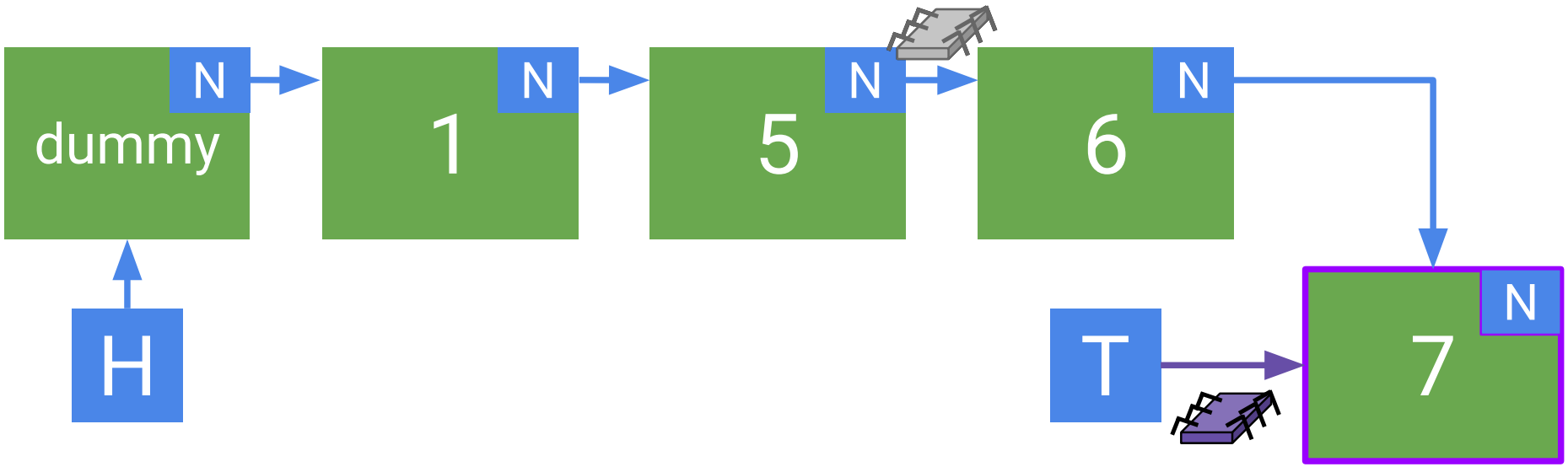
Жёлтый добавляет «6», фиолетовый — «7»



Далее добавляет как обычно

Починим проблему

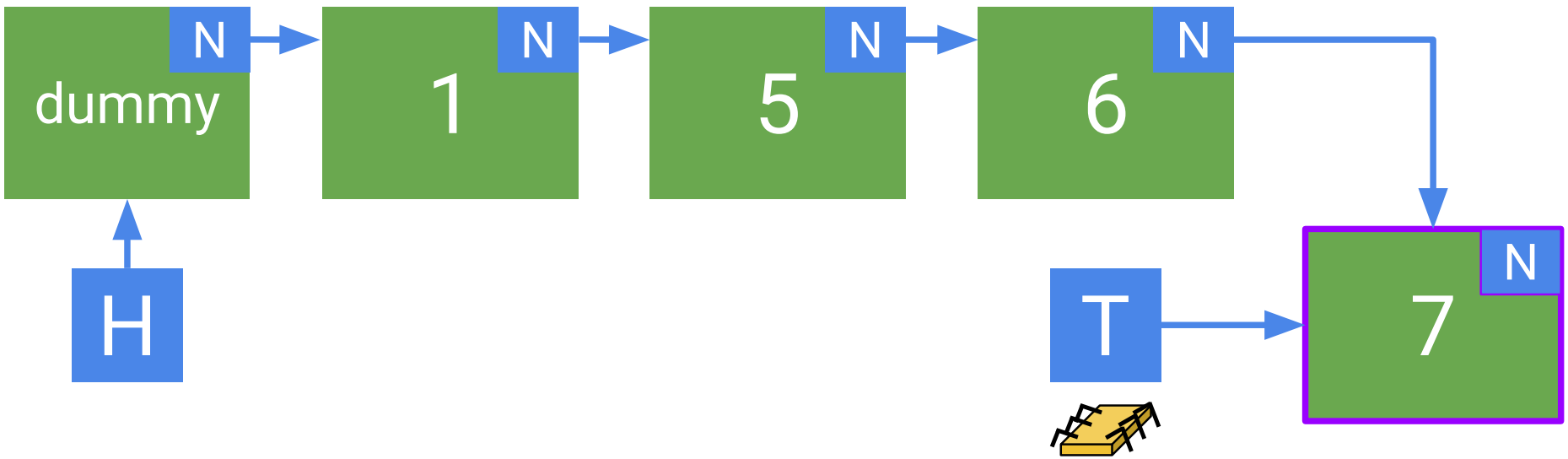
Жёлтый добавляет «6», фиолетовый — «7»



Далее добавляет как обычно

Починим проблему

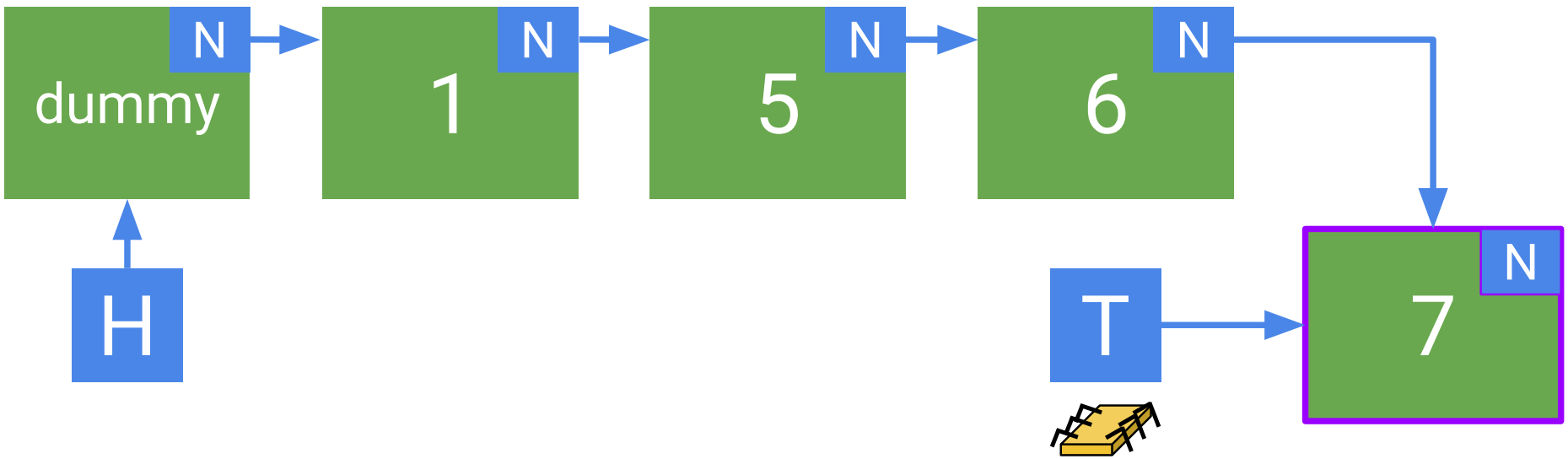
Жёлтый добавляет «6», фиолетовый — «7»



Когда жёлтый проснулся, его CAS для перемещения хвоста не пройдёт — уже перенесли

Починим проблему

Жёлтый добавляет «6», фиолетовый — «7»



Ура, починили!

Псевдокод: удаление

```
fun dequeue(): Int {
  while (true) { // CAS loop
    head := H
    headNext := head.N
    if CAS(&H, head, headNext):
      return headNext.Value
  }
}
```

Псевдокод: добавление

```
fun enqueue(x: Int) {
  newTail := Node { Value = x, N = null }
  while (true) { // CAS loop
    tail := T
    if CAS(&tail.N, null, newTail) {
      // ‘newTail‘ just added, move the tail forward
      CAS(&T, tail, newTail)
      return
    } else {
      // help other enqueue operations
      CAS(&T, tail, tail.N)
    }
  }
}
```

Задачка

Пропустив в очередной раз пары по многопоточному программированию, Антоха решил «соптимизировать» алгоритм очереди и удалить перенос хвоста очереди из nqueue. Остался ли алгоритм корректен?

```
fun enqueue(x: Int) {
  newTail = Node(val = x, next = null)
  while (true): // CAS loop
    curTail = T
    if T.N.CAS(null, newTail):
      T.CAS(curTail, newTail)
      return
}
```

Elimination for Queues

Можно ли использовать тот же алгоритм elimination-a, что и для стека?

Как корректно реализовать метод $\mathrm{size}()$?

Вопросы