

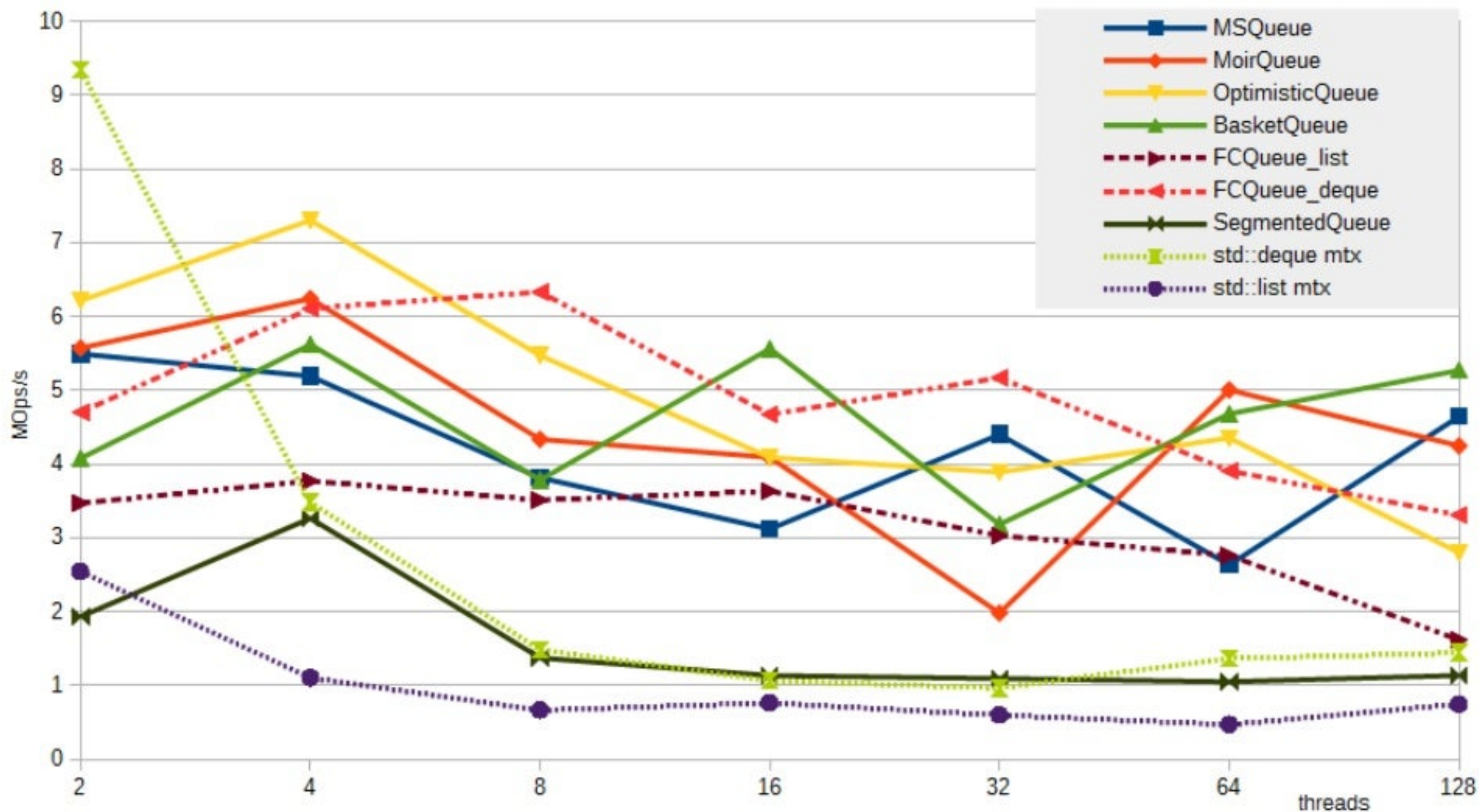


# **Lock-free maps изнутри**

# Queue

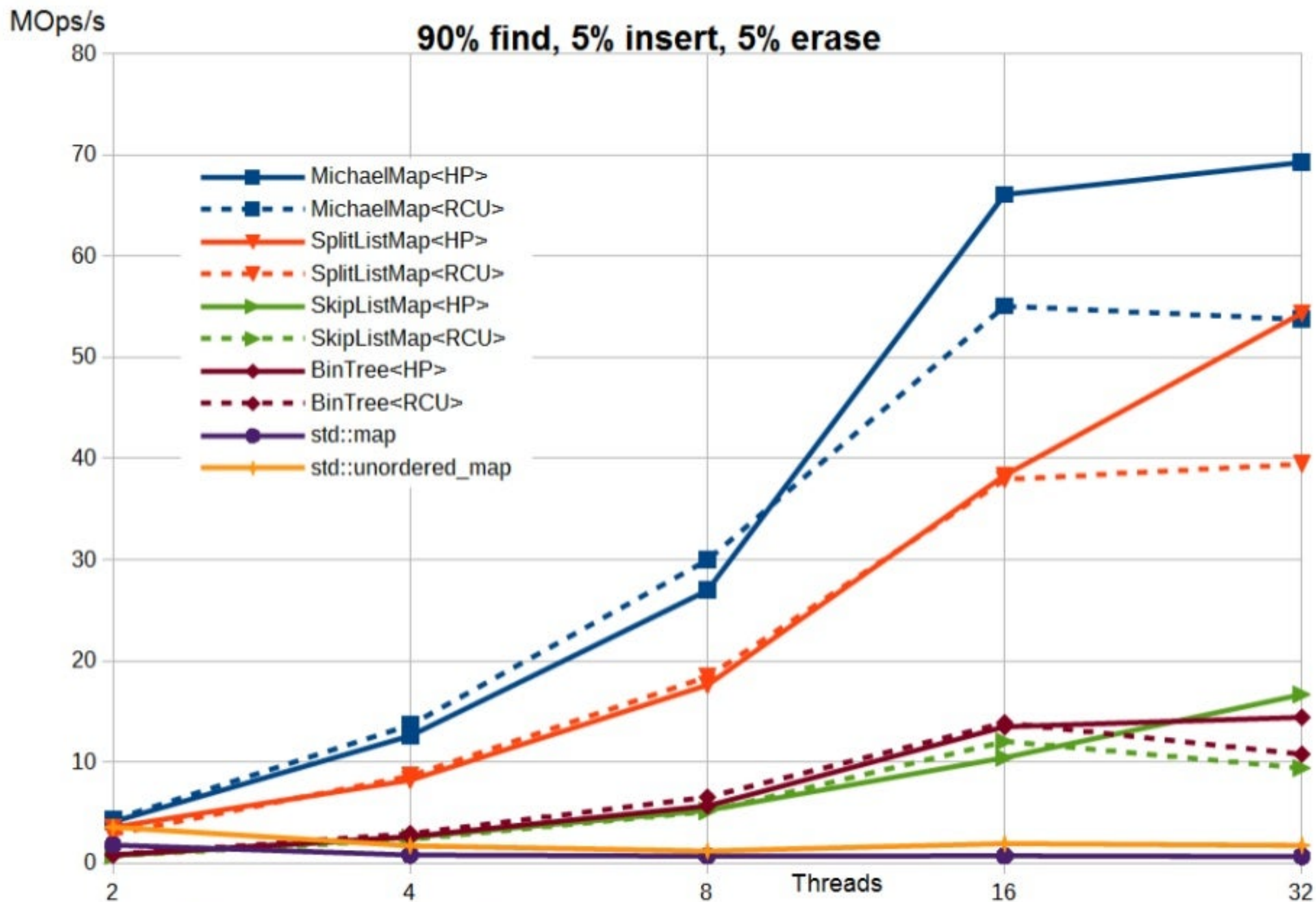
Consumer-producer, std queue

Linux IBM Power8 2x10x8 3.42 GHz





# Map



# Lock-free ordered list



## Операции:

- `insert( node )`
- `erase( key )`
- `find( key )`

```
template <class T>
struct node {
    std::atomic<node*> next_;
    T data_;
};
```

## Lock-free примитивы:

- `atomic load/store`
- `atomic compare-and-swap (CAS)`

# CAS — compare-and-swap

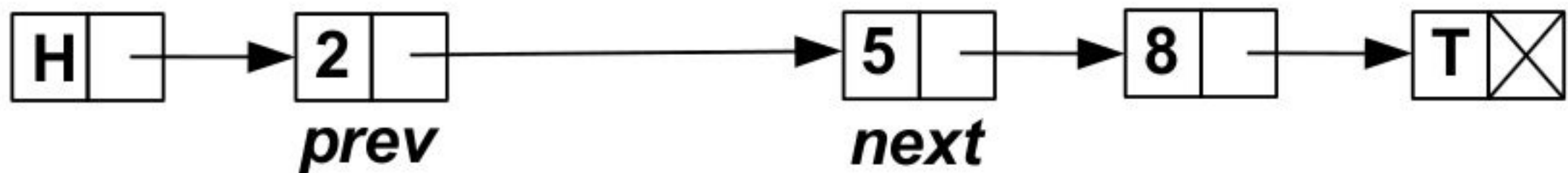
```
template <typename T>
bool CAS( T * pAtomic, T expected, T desired )
atomically {
    if ( *pAtomic == expected ) {
        *pAtomic = desired;
        return true;
    }
    else
        return false;
};
```

```
bool std::atomic<T>::compare_exchange(
    T& expected, T desired );
```

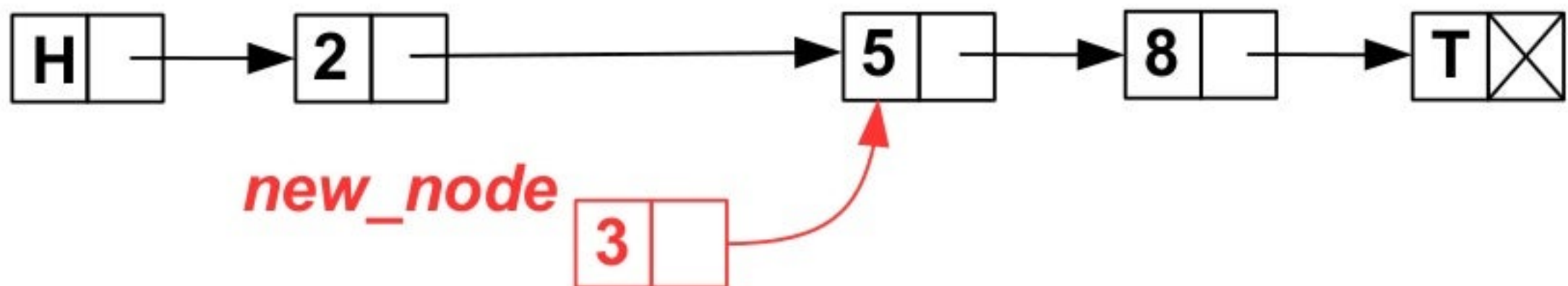


# Lock-free list: insert

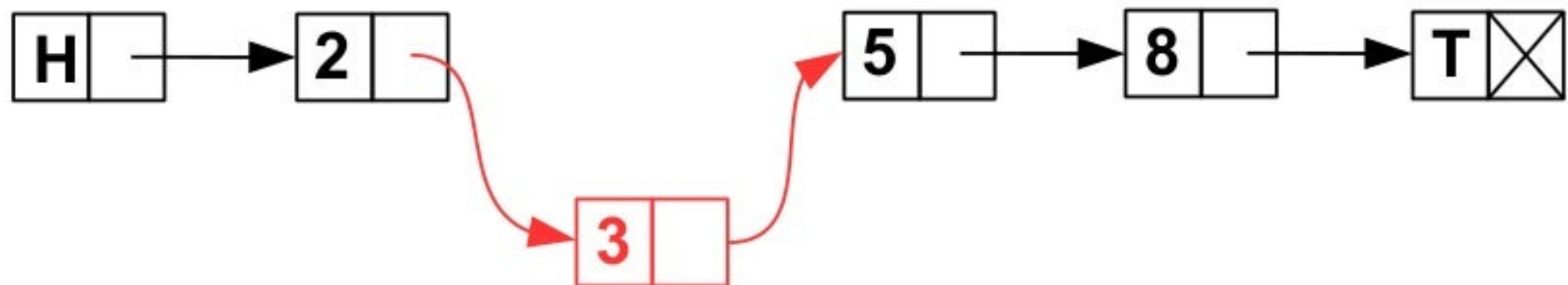
1. find insert position for key 3



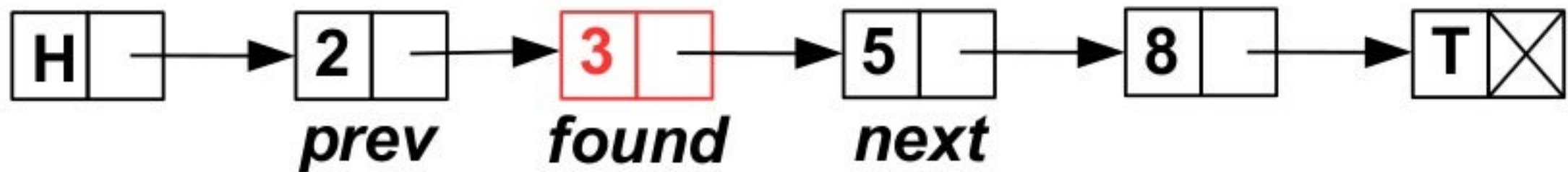
2. `new_node.next_.store( next )`



3. `prev->next_.CAS( next, new_node )`



# Local vars

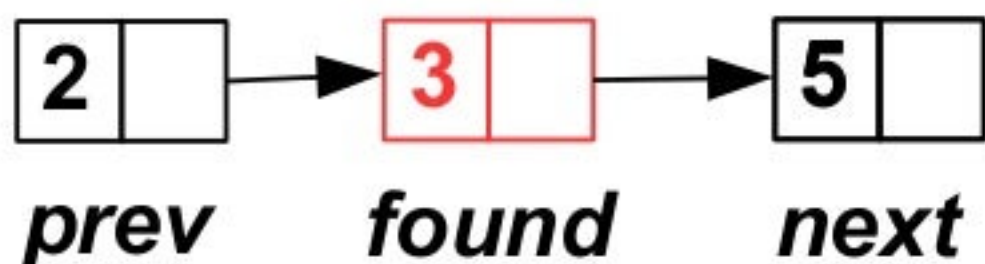


```
erase( Key k ) {  
  retry:  
    node * prev = Head;  
    do {  
      node * found = prev->next_.load();  
      if ( found->key == k ) {  
        node * next = found->next_.load();  
        if ( prev->next_.CAS( found, next ) ) {  
          delete found;  
          return true;  
        }  
        else  
          goto retry;  
      }  
      prev = found;  
    } while ( found->key < k );  
    return false;  
}
```

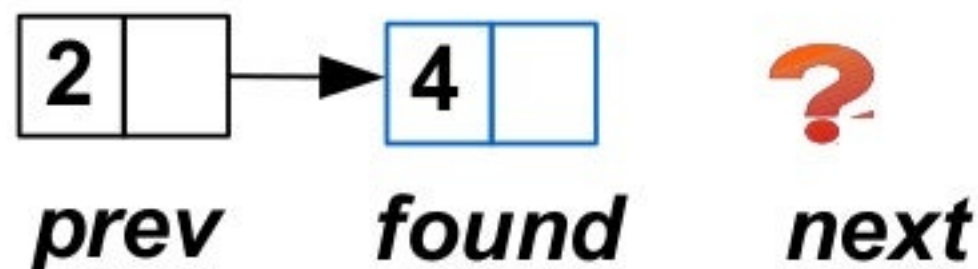
**Проблема: локальные ссылки**

# ABA-проблема

Thread A: erase(3)



*preempted...*

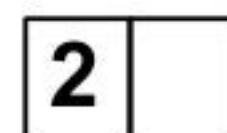


prev->next\_.CAS( found, next ) - **success!!!**



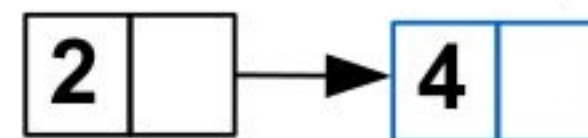
Thread B

erase(3); erase(5)



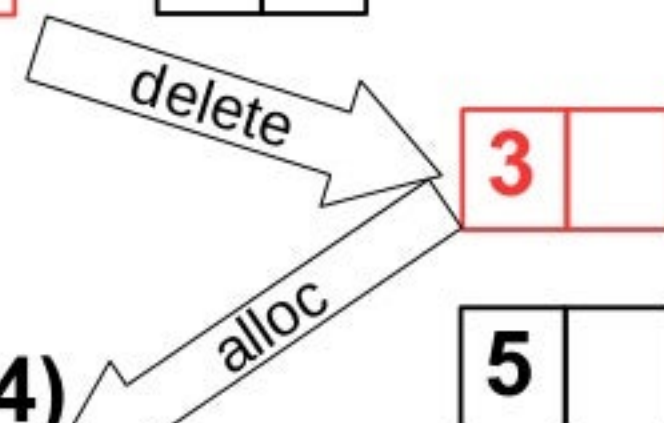
insert(4)

new node(4)



***addr(3) == addr(4)***

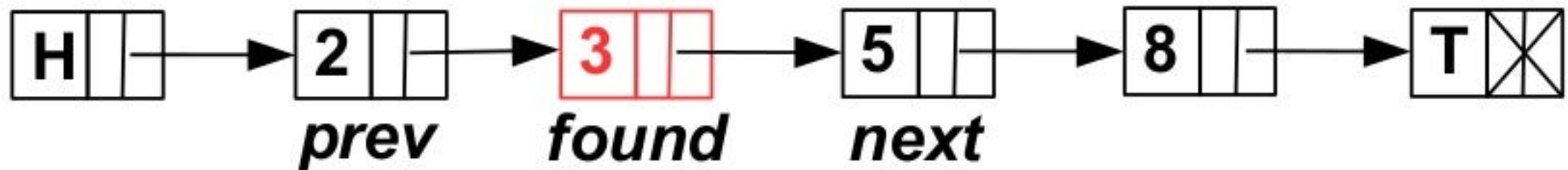
Heap





# Tagged pointers

```
template <class T>
struct tagged_ptr {
    T * ptr;
    uintptr_t tag;
};
```

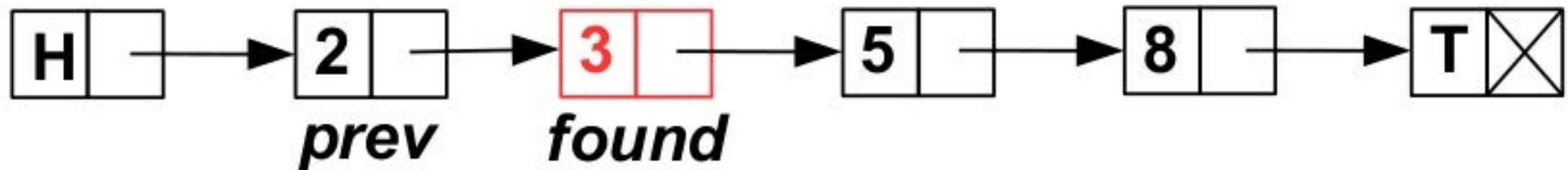


`prev->next_.dwCAS( found, {next.ptr, prev->next_.tag + 1} )`

- ✗ Требуется **dwCAS** — не везде есть
- ✗ Решает только АВА-проблему
  - ✗ Освободить память **нельзя**, нужен *free-list*

[ *boost.lock-free* ]

# Hazard pointers



```
erase( Key k ) {  
    hp_guard hp1 = get_guard();  
    hp_guard hp2 = get_guard();
```

*Распределяем HP (TLS)*

```
retry:
```

```
    node * prev = Head;
```

```
    do {
```

```
        node * found = hp2.protect( prev->next_ );
```

*Защищаем элемент*

```
        if ( found->key == k )
```

```
            if (prev->next_.CAS( found, found->next_.load() )) {
```

```
                hp_retire( found );
```

*Отложенное удаление*

```
                return true;
```

```
            }
```

```
            else
```

```
                goto retry;
```

```
        }
```

```
        hp1 = hp2;
```

```
        prev = found;
```

```
    } while ( found->key < k );
```

```
    return false;
```

```
}
```



# Hazard Pointers

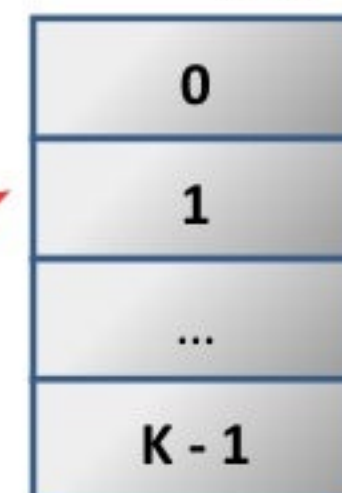
## Объявление Hazard Pointer'a – защита локальной ссылки

```
class hp_guard {  
    void * hp;  
    // ...  
};
```

```
T * hp_guard::protect(std::atomic<T*>& what) {  
    T * t;  
    do {  
        hp = t = what.load();  
    } while (t != what.load());  
    return t;  
}
```

*thread-local*

• HP[K]



0
1
...
K - 1

*Другой поток может  
изменить what, поэтому - цикл*



# Hazard Pointers

## Удаление элемента

```
void hp_retire( T * what ) {  
    push what to current_thread.Retired array  
    if ( current_thread.Retired is full )  
        hp.Scan( current_thread );  
}
```

HP[K]	Retired[R]
0	0
1	1
...	2
K - 1	...
	R - 1

*thread-local*

// сердце hazard pointer

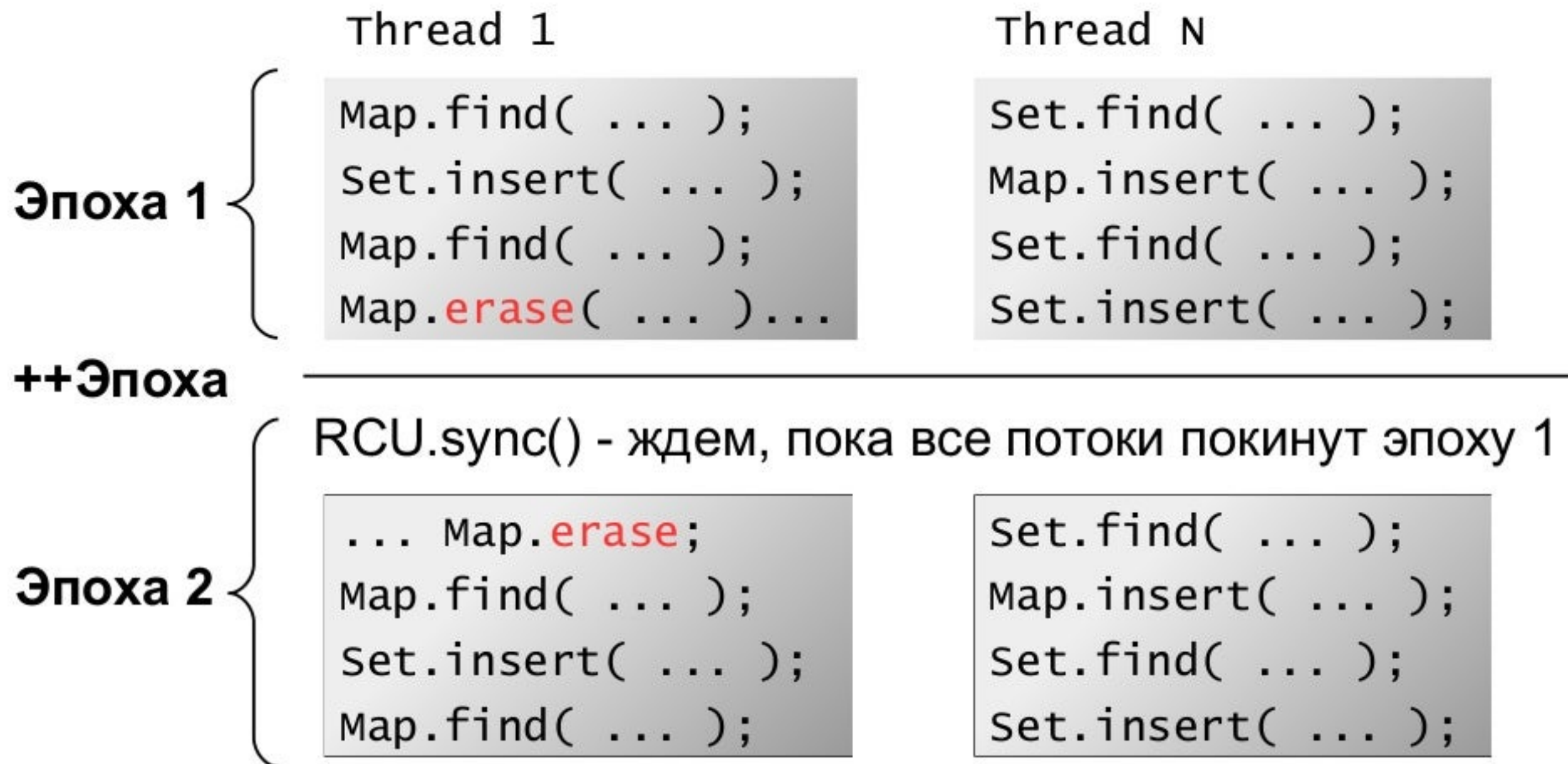
```
void hp::Scan( current_thread ) {  
    void * guarded[K*P] = union HP[K] for all P thread;  
    foreach ( p in current_thread.Retired[R] )  
        if ( p not in guarded[] )  
            delete p;  
}
```

# Hazard pointers

- ✓ решает АВА-проблему
- ✓ Физическое удаление элементов
- ✓ Использует только атомарные чтение/запись
- ◆ Защищает только **локальные** ссылки
- ✓ Размер массива отложенных (готовых к удалению) элементов **ограничен сверху**
- ◆ Перед работой с указателем его следует объявить как **hazard**



# Epoch-based SMR





# Lock-free list

## Atomic примитивы

- atomic load/store
- atomic CAS



Safe memory  
reclamation schema  
(Hazard Pointers, uRCU)

## Решили:

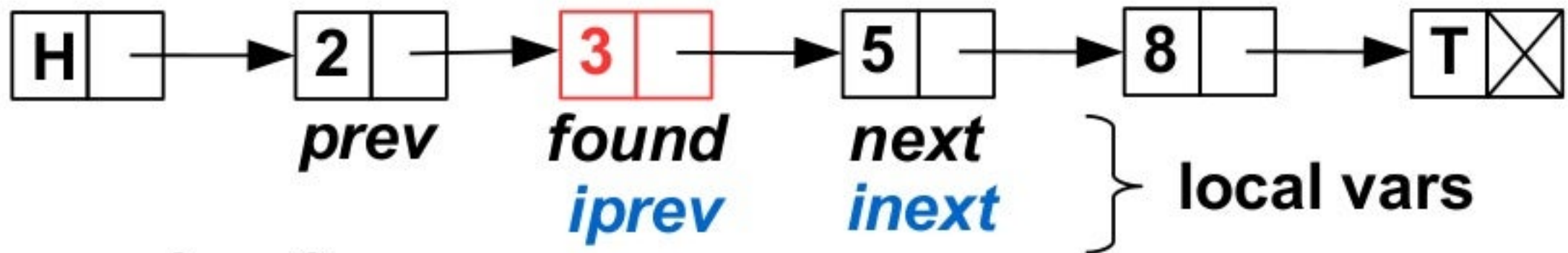
- Проблему удаления узлов (delete)
- АВА-проблему

**Открытая проблема: параллельный insert и erase**

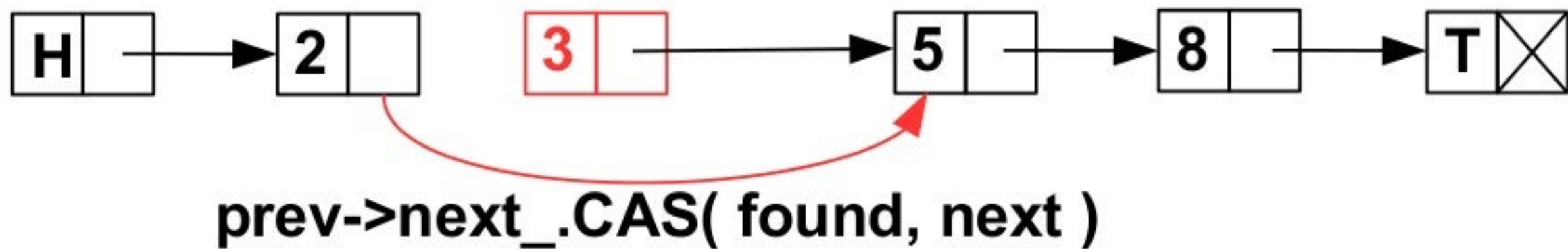
# Lock-free list: insert/erase

A: find key 3

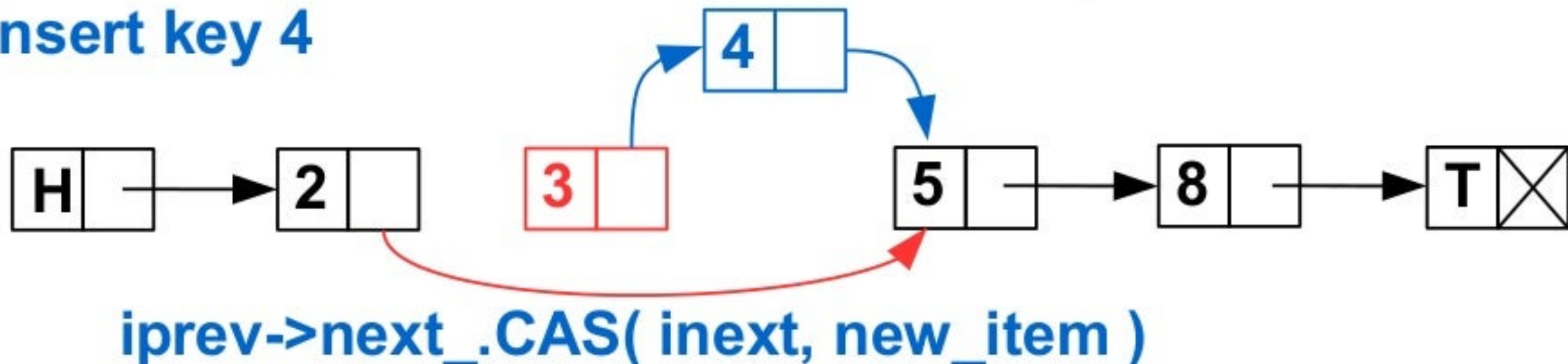
B: find insert pos for key 4



A: erase key 3



B: insert key 4



# Marked pointer

[ T.Harris, 2001 ]

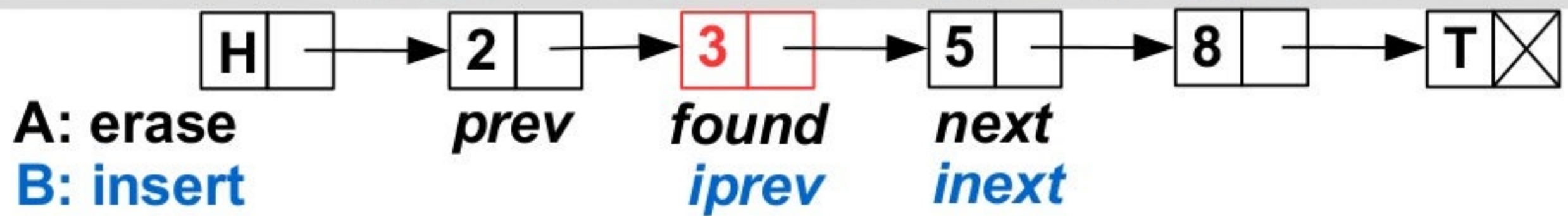
**Двухфазное удаление:**

- *Логическое* удаление — помечаем элемент
- *Физическое* удаление — исключаем элемент

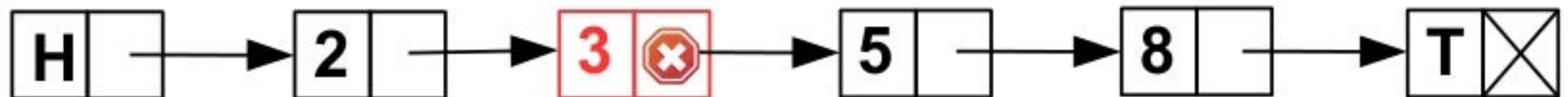
**В качестве метки используем младший бит указателя**



# Lock-free list: marked pointer



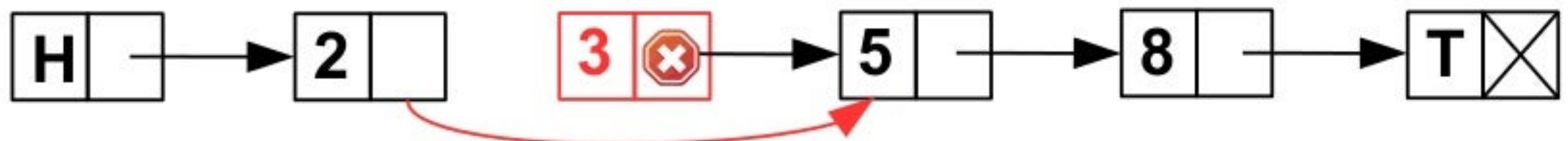
**A: Logical deletion - mark item *found***



**found** → **next\_.CAS( next, next | 1 )**

**B: iprev** → **next\_.CAS( inext, new\_item )** - **failed!!!**

**A: Physical deletion - unlink item *found***



**prev** → **next\_.CAS( found, next )**

# Итераторы

## Требования и контраргументы

```
for (auto it = list.begin(); it != list.end(); ++it)  
    it->do_something();
```

**1** Выдача итератора наружу — фактически,  
ссылки на элемент списка

**но - элемент в любой момент может быть  
удален**



# guarded\_ptr

```
template <typename T>
struct guarded_ptr {
    hp_guard hp;    // защищает элемент
    T * ptr;    // элемент списка
    guarded_ptr(std::atomic<T *>& p) {
        ptr = hp.protect( p ); }
    ~guarded_ptr() { hp.clear(); }
    T * operator ->() const { return ptr; }
    explicit operator bool() const
        { return ptr != nullptr; }
};
```



// Пример

```
guarded_ptr gp = list.find( key );
if ( gp ) {
    // можно безопасно обращаться к полям T через gp
}
```



# Итераторы

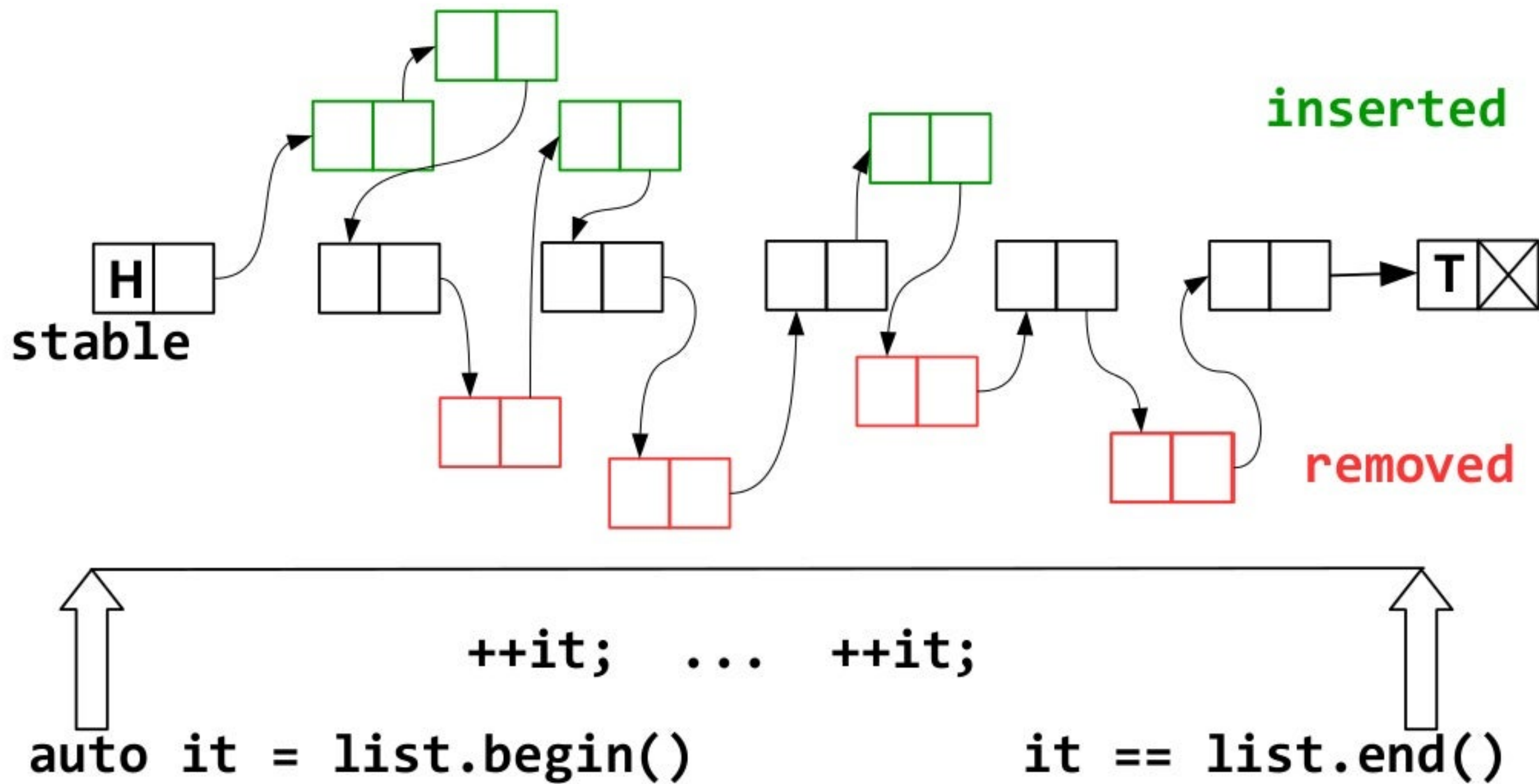
## Требования и контраргументы

```
for (auto it = list.begin(); it != list.end(); ++it)  
    it->do_something();
```

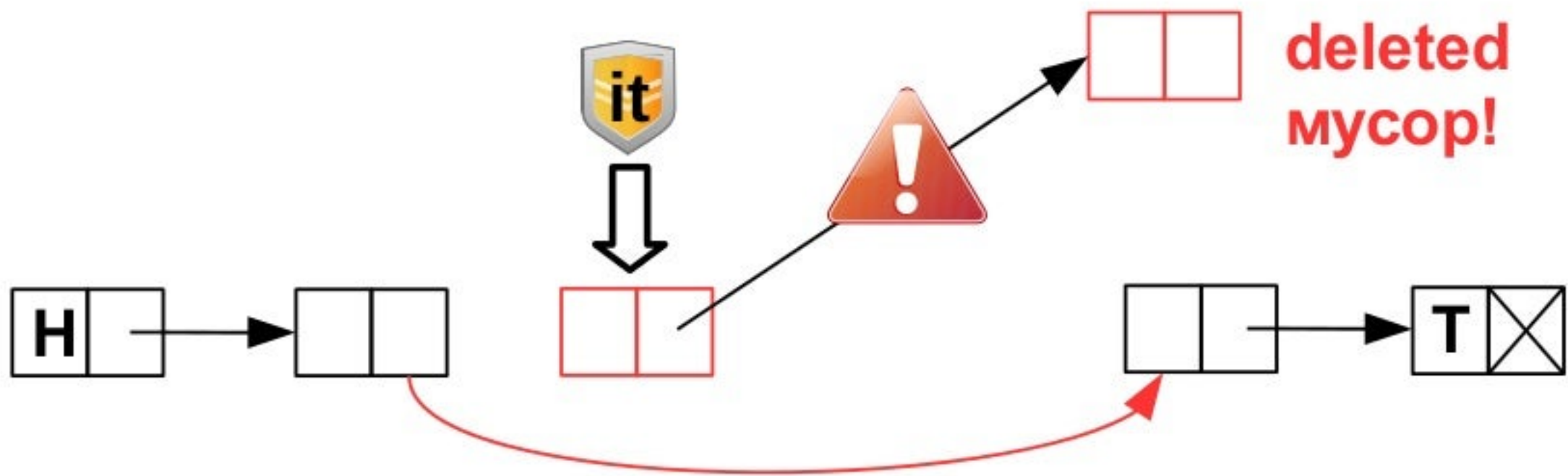
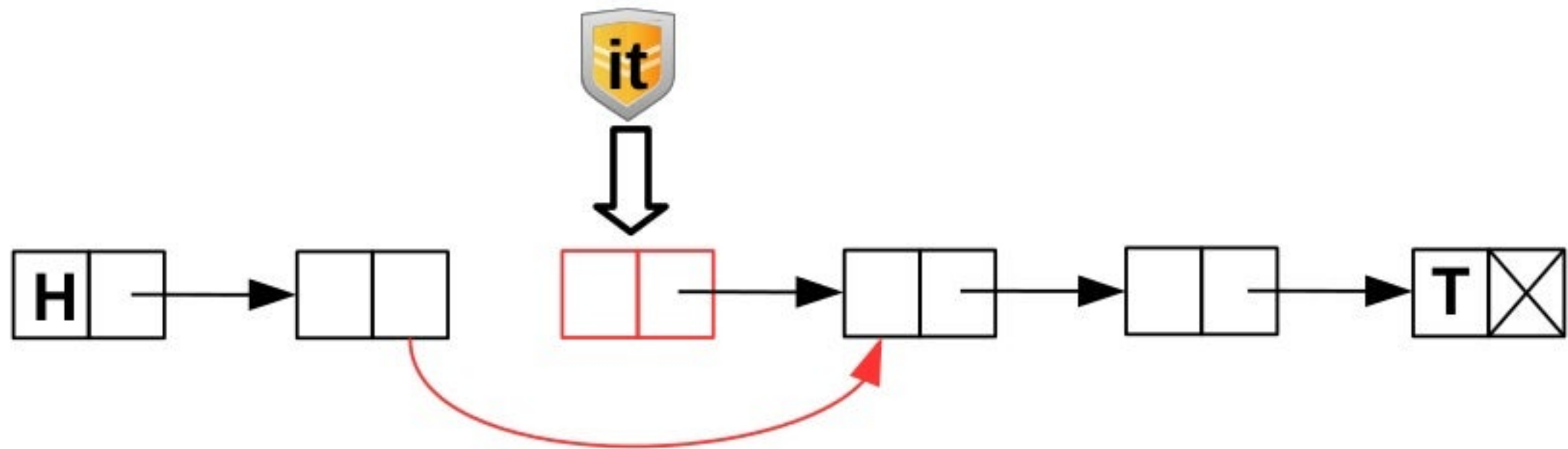
## 2 Обход всех элементов списка

**НО - СПИСОК ПОСТОЯННО ИЗМЕНЯЕТСЯ**

# Итераторы

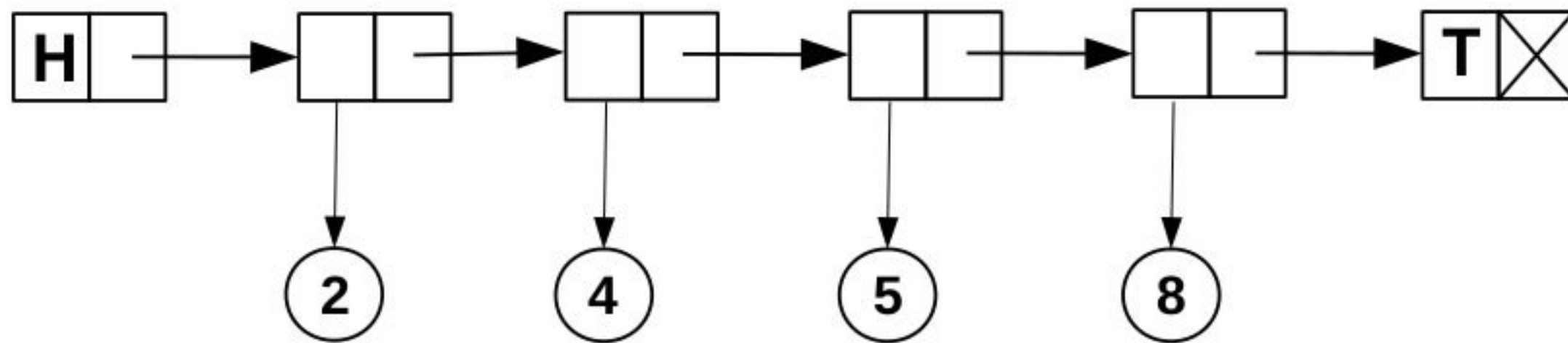


# Итераторы



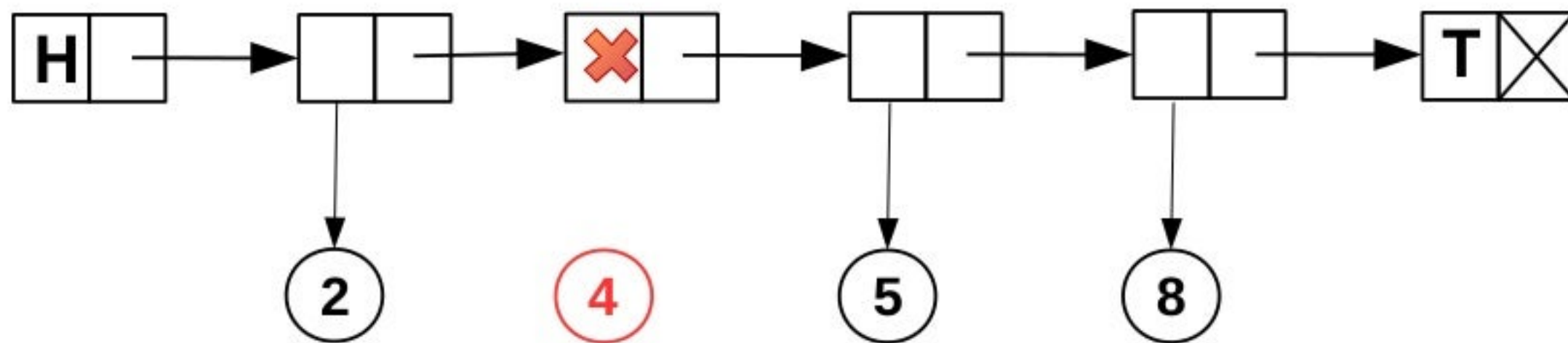


# Iterable lock-free list




Элементы списка хранят **указатели** на данные

При удалении ключа обнуляется указатель, сам **элемент** списка **остается**



```
struct node {  
    std::atomic<node *> next;  
    std::atomic<T *> data;  
};
```

# Iterable lock-free list

```
class iterator {  
    node *    node;  
    guarded_ptr<T> data; // текущий элемент   
public:  
    iterator& operator++() {  
        while ( node ) {  
            node = node->next.load(); // не требует защиты  
            if ( !node ) break;  
            data.ptr = data.hp.protect(node->data.load());  
            if ( data.ptr ) break;  
        }  
        return *this;  
    }  
    T* operator ->() { return data.ptr; }  
    T& operator *()  { return *data.ptr; }  
};
```



**Спасибо за внимание!**

**<https://github.com/khizmax/libcds>**