Fast Bounded-Concurrency Hash Tables

Samy Al Bahra / sbahra@backtrace.io
@0xF390

Backtrace

Introduction

A general mechanism for achieving non-blocking progress guarantees cheaply.

- Support for any open-addressed collision resolution mechanism.
- No atomic operations, memory barriers or memory allocation on the fast path (for TSO).
- Writers never block.
- Readers only block for a small subset of executions.
- Dead simple.

Motivation

- Single-writer multi-reader (SWMR) workload.
- Constant stream of read operations but bursts of millions of writes every few minutes.
- Firm real-time requirements on read-side.
- Starvation-freedom on write-side.
- Systems were memory-constrained so memory efficiency is important.

Motivation

State-of-the-art entailed significant trade-off in complexity and performance.

Motivation

State-of-the-art entailed significant trade-off in complexity and performance.

Chaining

- Memory management.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

Motivation

State-of-the-art entailed significant trade-off in complexity and performance.

Chaining

- Memory management.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

Open Addressing

- Complex object re-use constraints in absence of key duplication.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

Motivation

State-of-the-art entailed significant trade-off in complexity and performance.

Chaining

- Memory management.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

Open Addressing

- Complex object re-use constraints in absence of key duplication.
- Reliance on expensive operations.
- Limitations on collision resolution mechanism.

What does specialization get us?

Constraints

- Termination-safety is not a requirement.
- SWMR allows us to rely on less complex instructions.
- Primarily executing on x86 processors.
  - Load to load ordering.
  - Store to store ordering.

Implementation

- **ck_pr_store_X**(a, b)
  atomically { *a = b }
- **ck_pr_load_X**(a)
  atomically { *a }
- **ck_pr_fence_store**()
  smp_wb()
- **ck_pr_fence_load**()
  smp_rb()

Implementation

```
1  struct slot {
2      /*
3       * The key field may be the key value or a reference to
4       * the key value.
5       */
6      void *key;
7
8      /*
9       * The value field may be the value value or a reference to
10      * the value value.
11      */
12     void *value;
13 };
```

Implementation

delete

```
1  ck_pr_store_ptr(&slot->key, T);
2  ck_pr_fence_store();
```

Implementation

delete

```
1  ck_pr_store_ptr(&slot->key, T);
2  ck_pr_fence_store();
```

insert

```
1  if (ck_pr_load_ptr(&slot->key) == T) {
2      ck_pr_store_uint(&D, D + 1);
3      ck_pr_fence_store();
4  }
5
6  ck_pr_store_ptr(&slot->value, value);
7  ck_pr_fence_store();
8  ck_pr_store_ptr(&slot->key, key);
```

Implementation

get

```
1  get(struct slot *result, void *key)
2  {
3    unsigned int d, d_p;
4
5    do {
6        d = ck_pr_load_uint(&D);
7        ck_pr_fence_load();
8        search(result, key);
9        ck_pr_fence_load();
10       d_p = ck_pr_load_uint(&D);
11   } while (d != d_p);
12
13   return;
14 }
```

Implementation

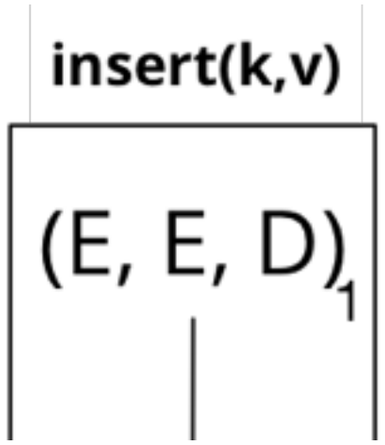
get

```
1   get(struct slot *result, void *key)
2   {
3     unsigned int d, d_p;
4
5     do {
6         d = ck_pr_load_uint(&D);
7         ck_pr_fence_load();
8         search(result, key);
9         ck_pr_fence_load();
10        d_p = ck_pr_load_uint
11    } while (d != d_p);
12
13    return;
14  }
```

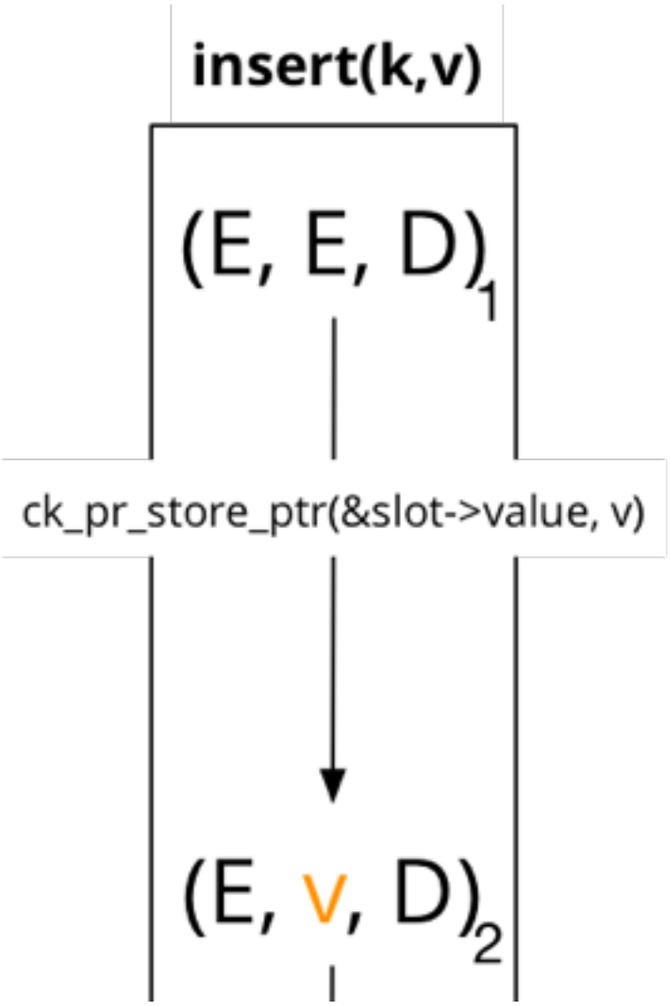
search

```
1   search(struct slot *result, void *key)
2   {
3   [...]
4
5       result->key = ck_pr_load_ptr(&slot->key);
6       ck_pr_fence_load();
7       result->value = ck_pr_load_ptr(&slot->value);
8
9   [...]
10  }
```

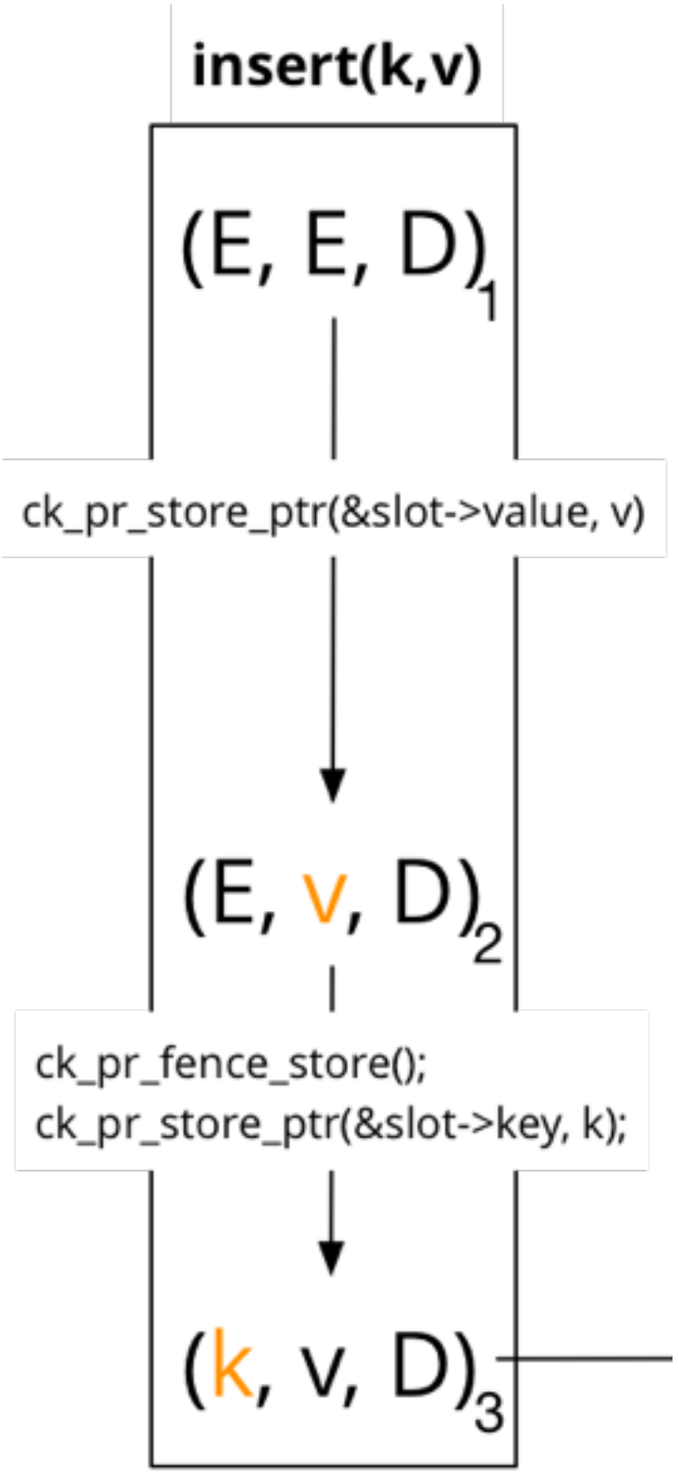
Correctness



Correctness

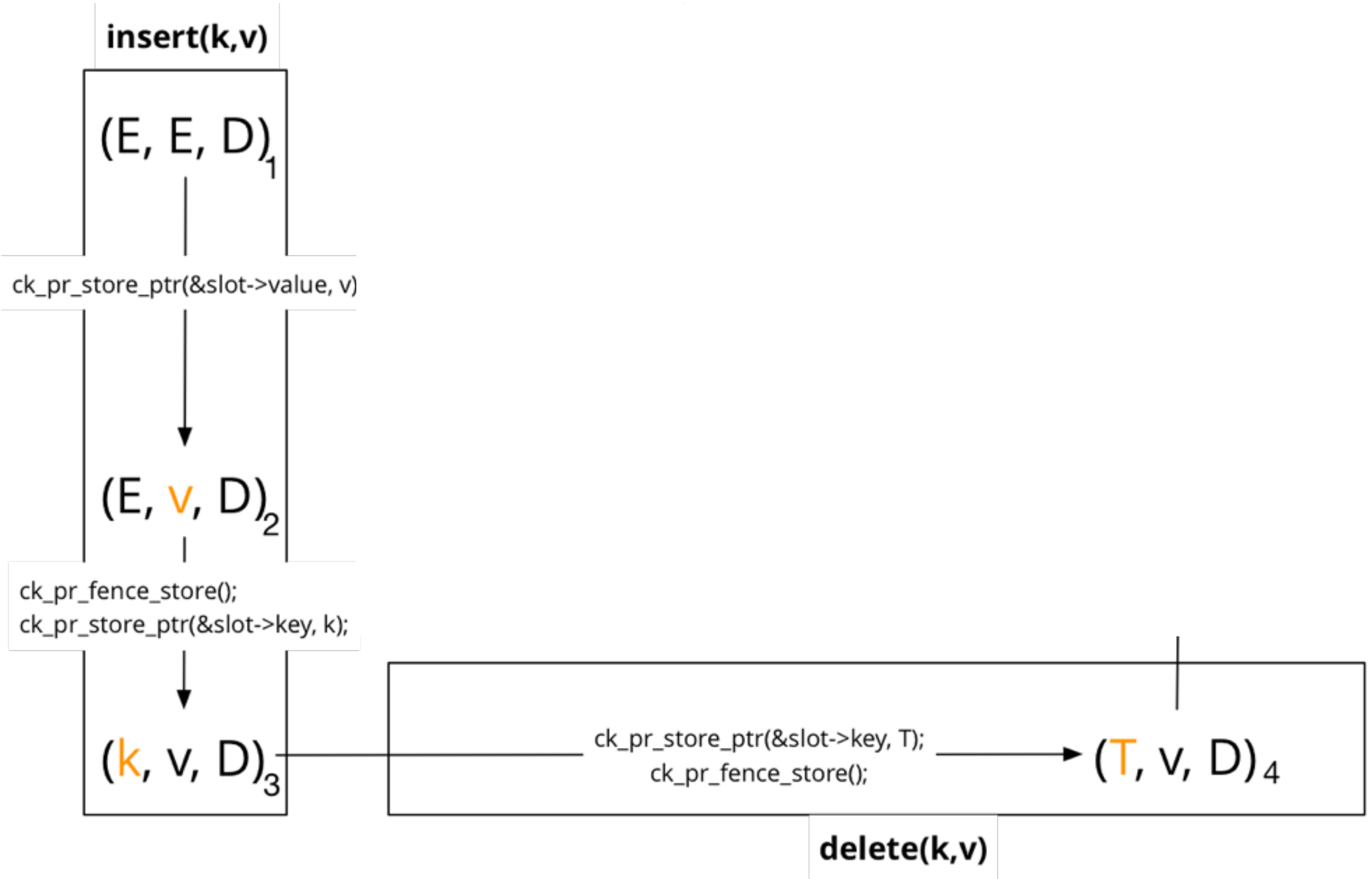


Correctness



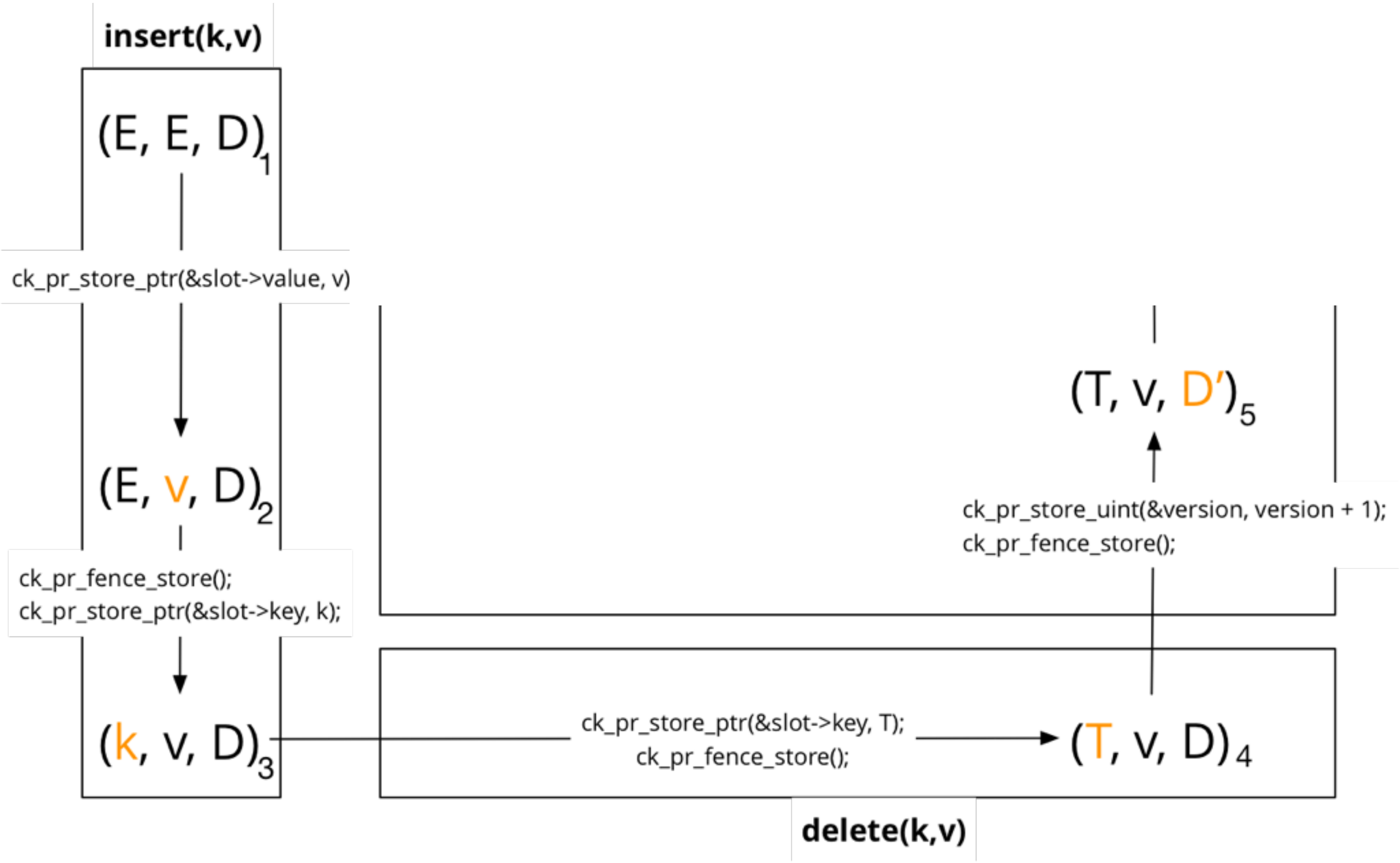
Backtrace

Correctness



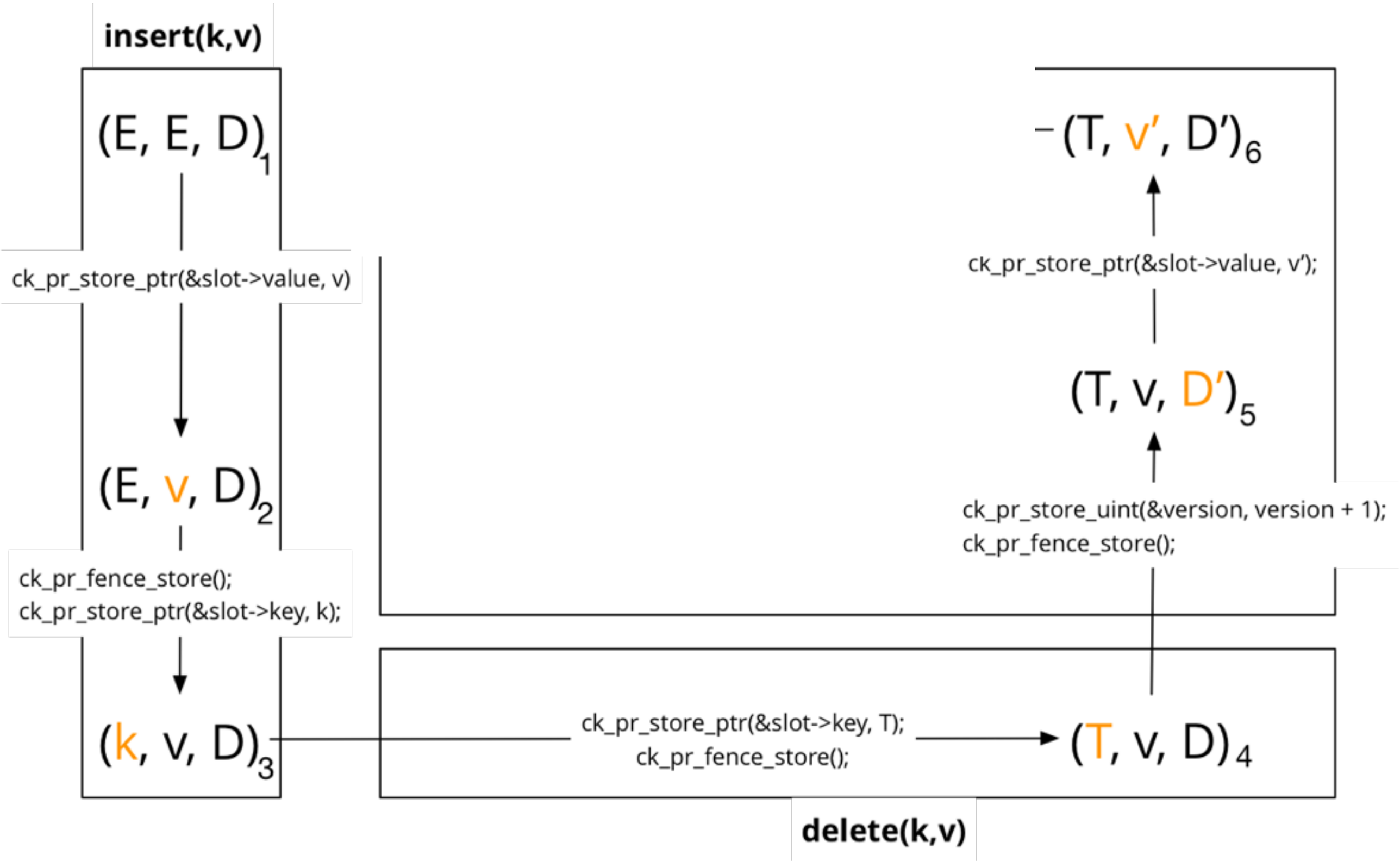
Backtrace

Correctness



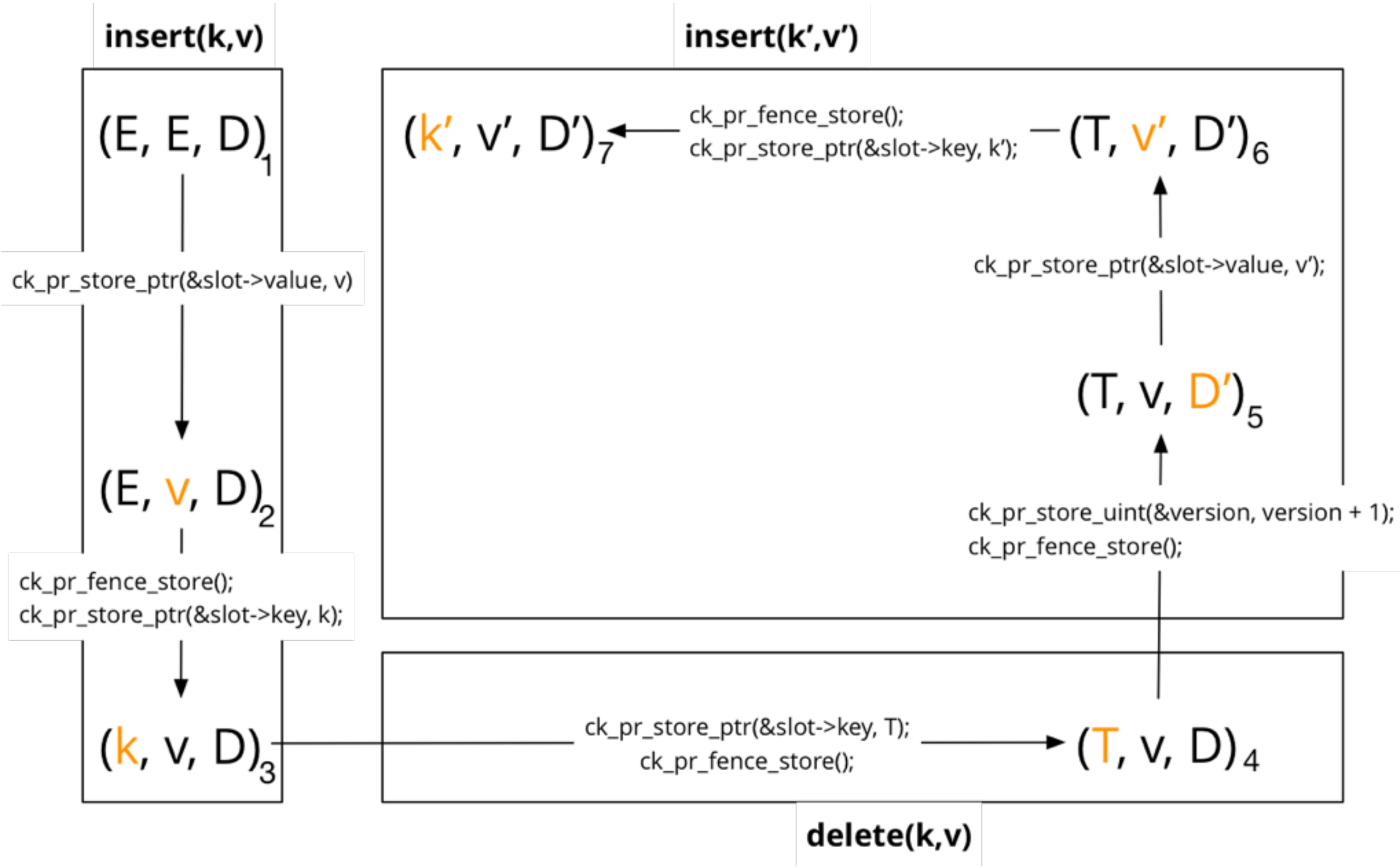
Backtrace

Correctness



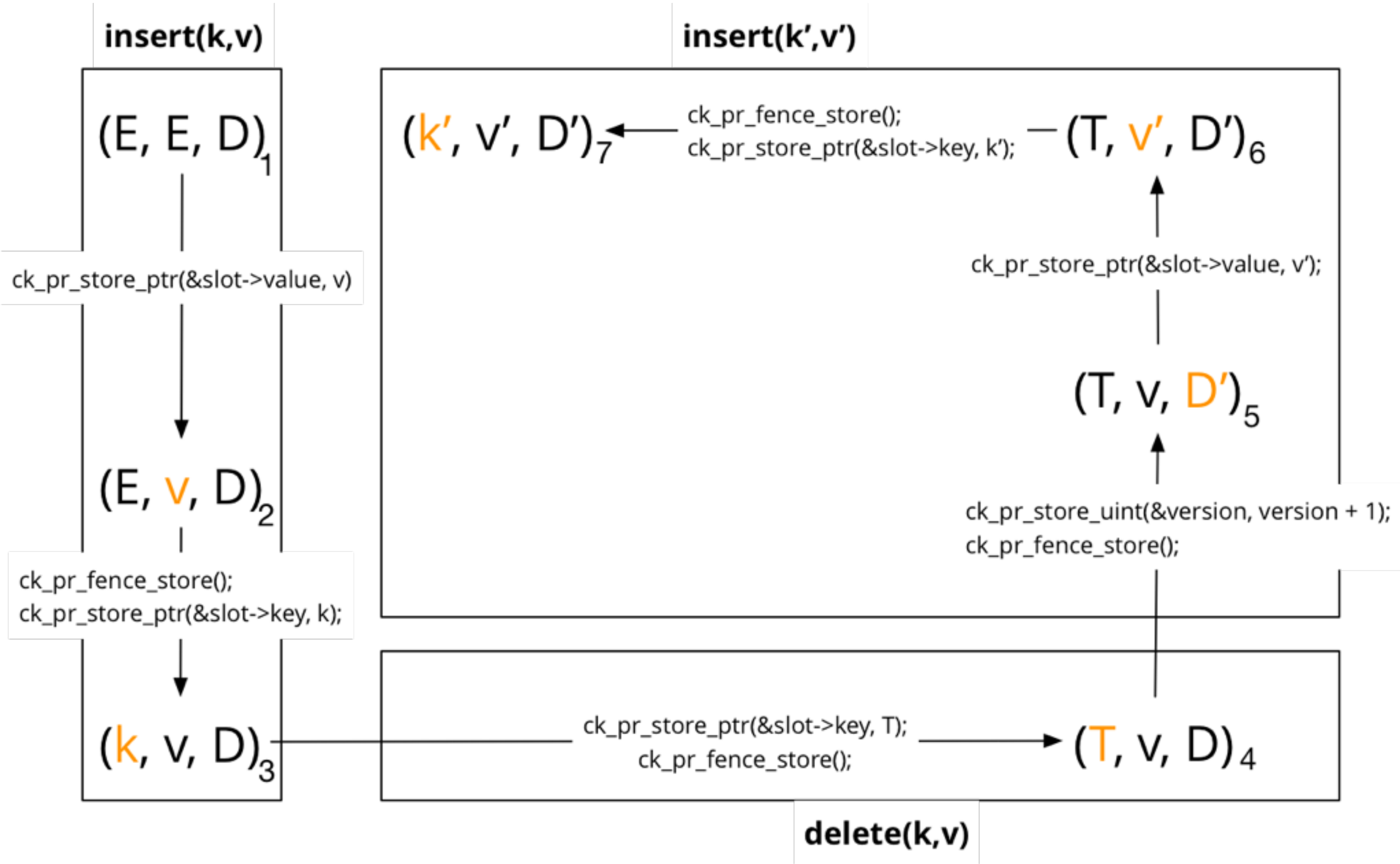
Backtrace

Correctness



Backtrace

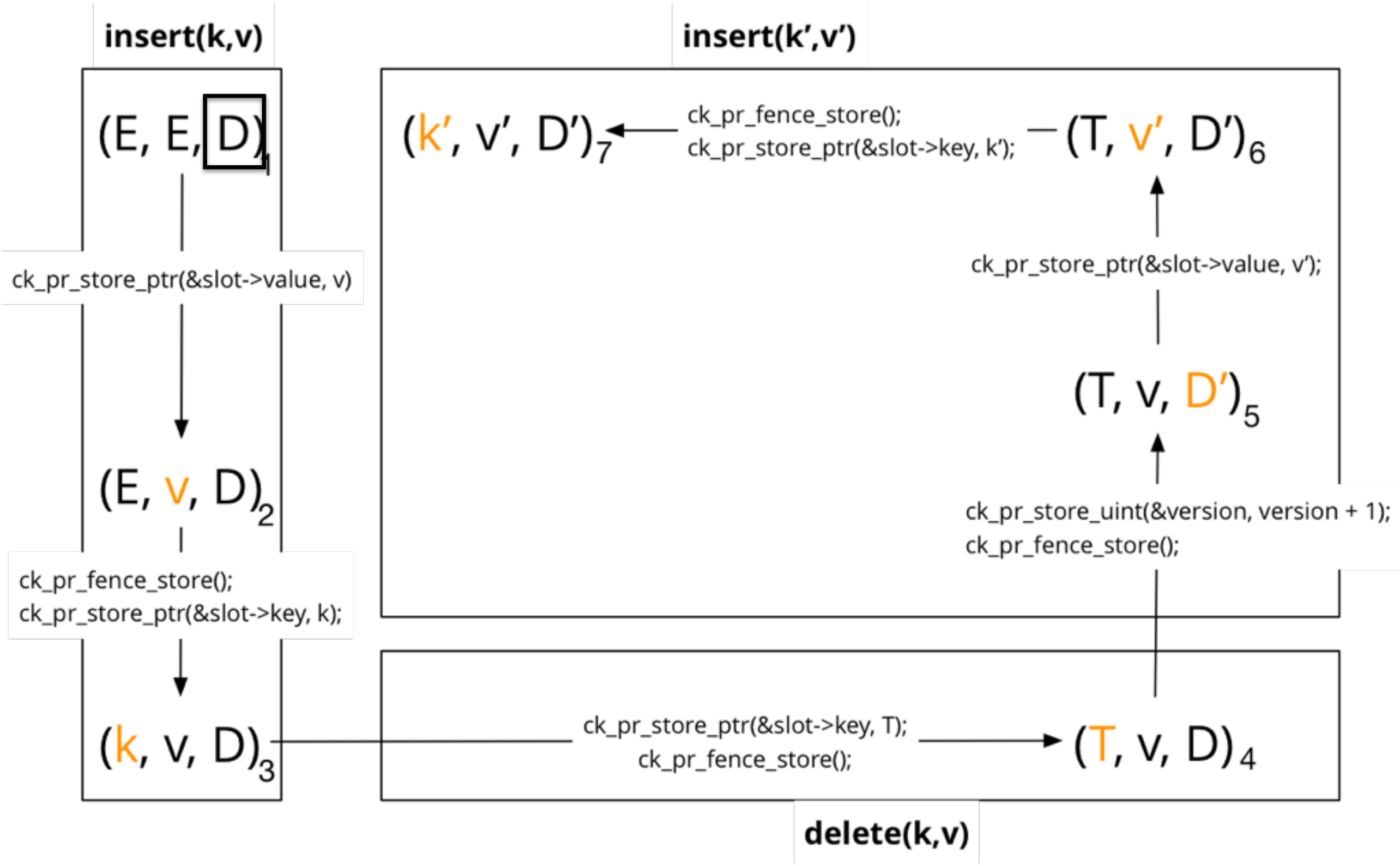
Correctness



Observed (D, E, v, D)

Backtrace

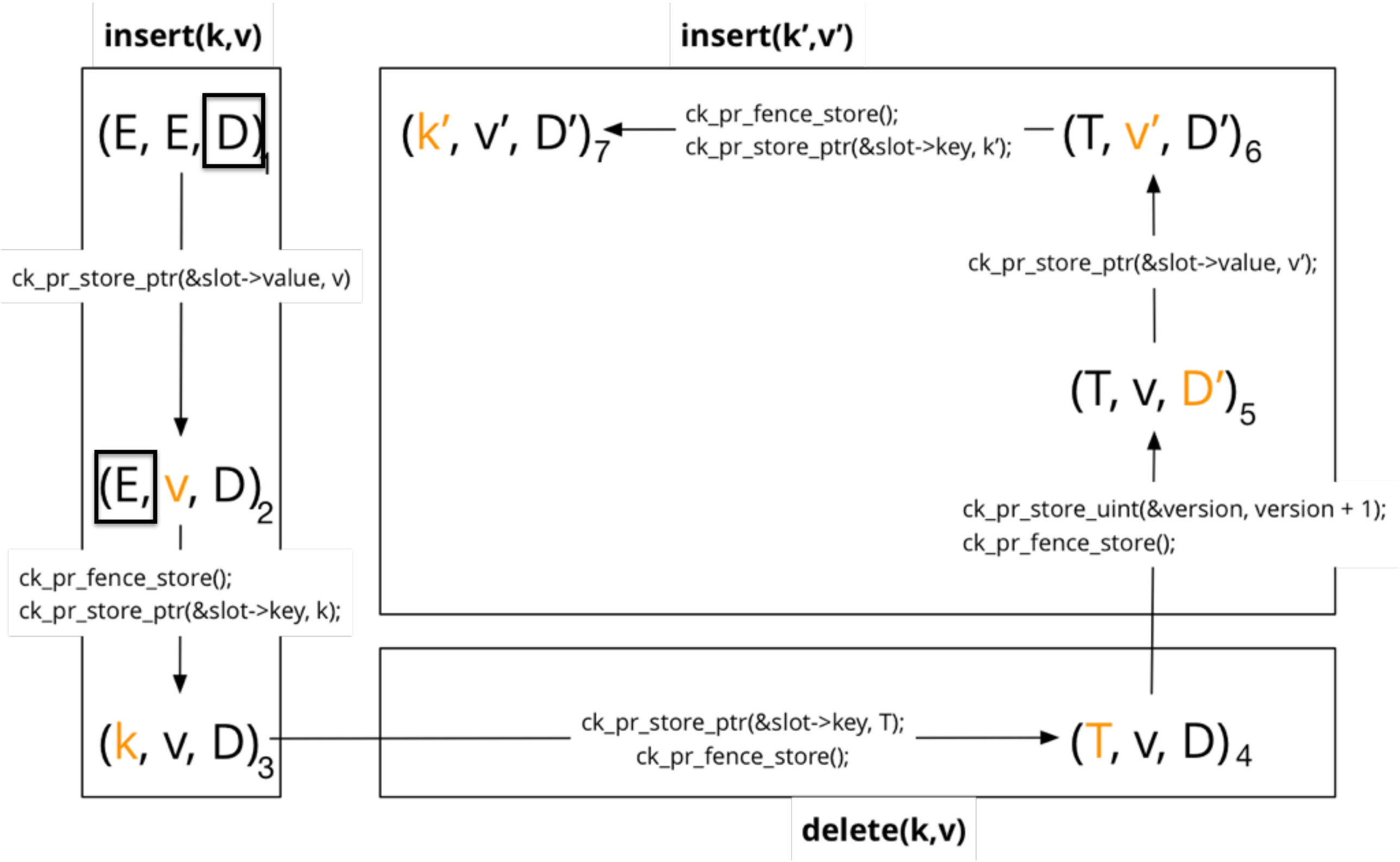
Correctness



Observed (D, E, v, D)

Backtrace

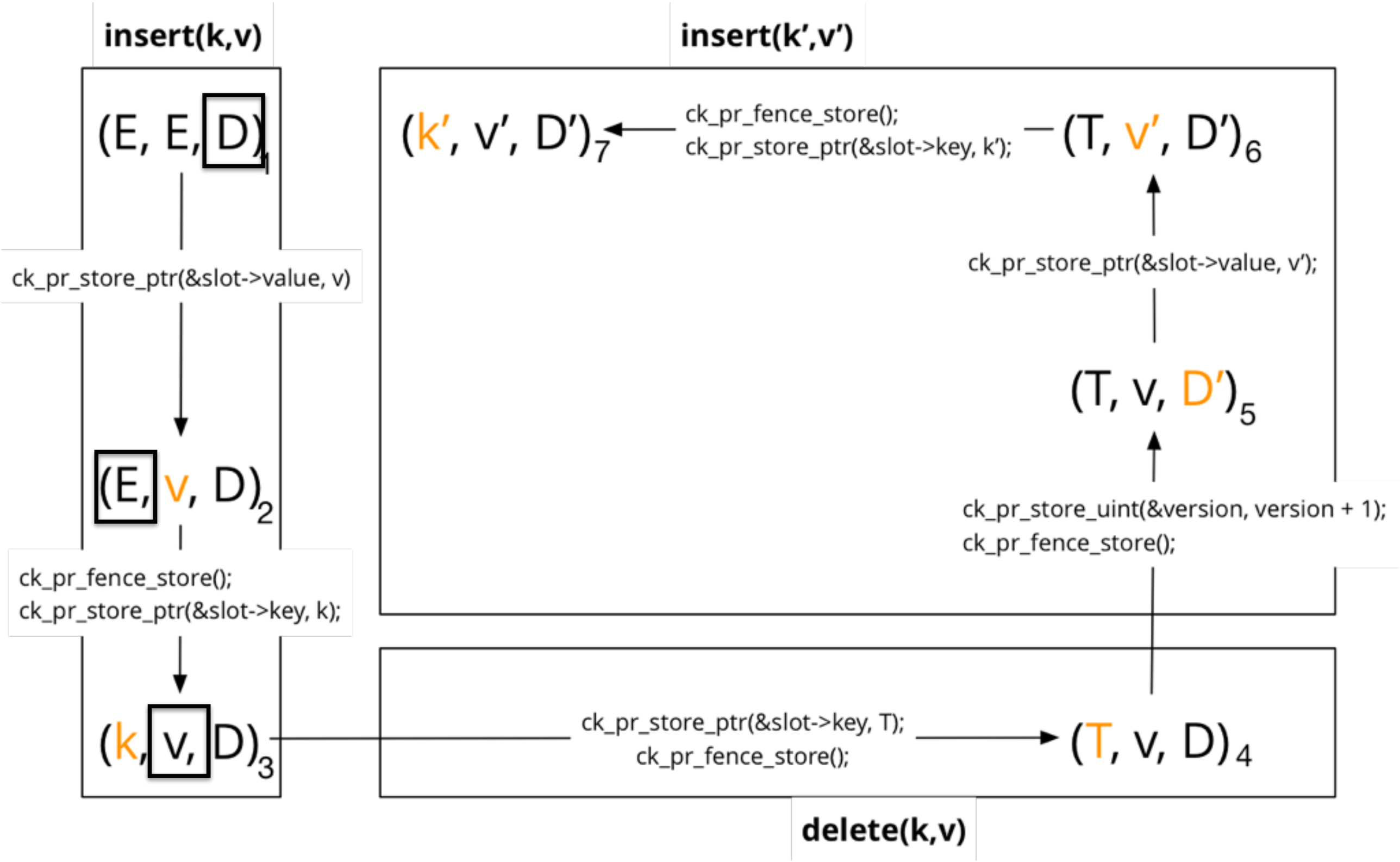
Correctness



Observed (D, E, v, D)

Backtrace

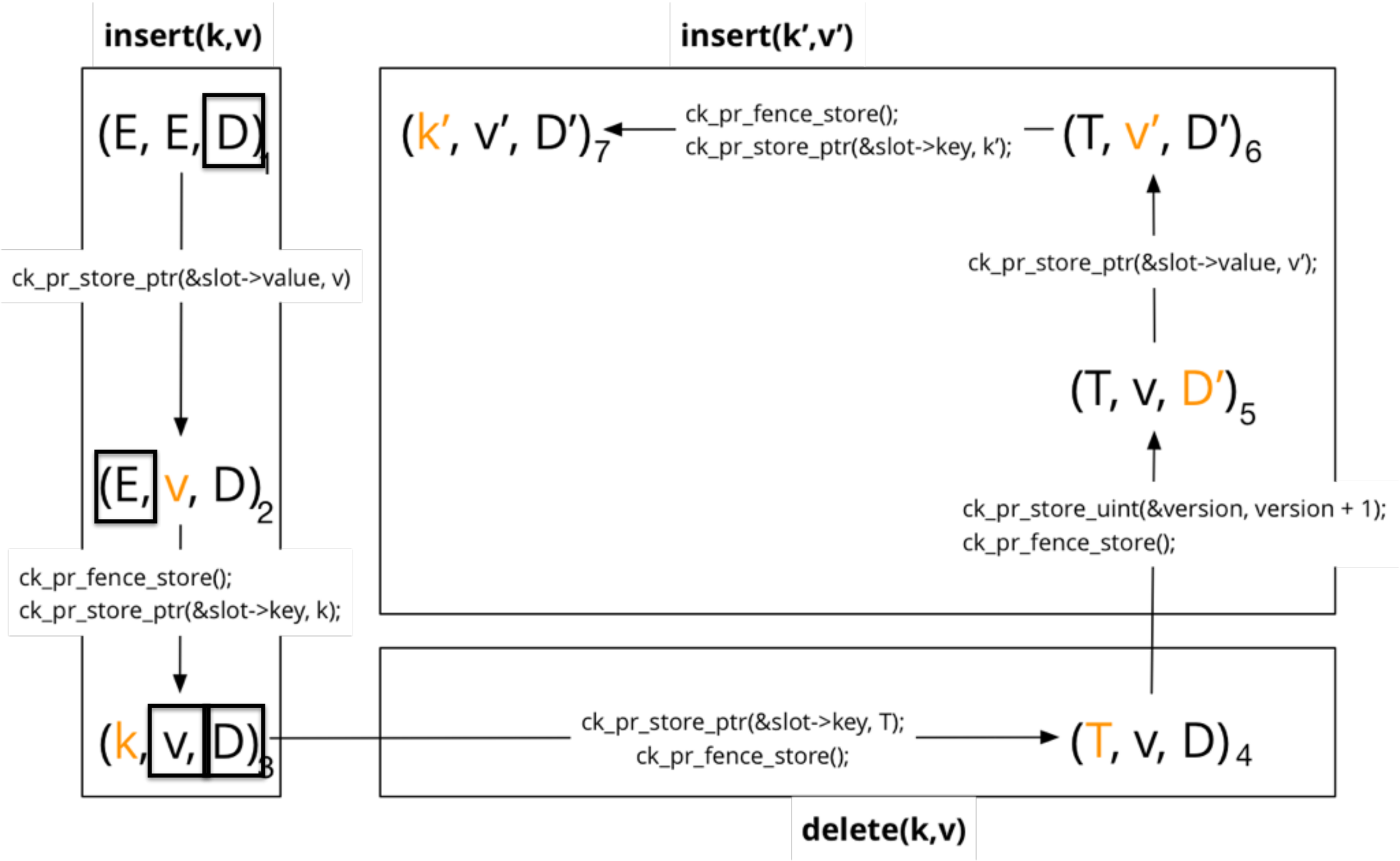
Correctness



Observed (D, E, v, D)

Backtrace

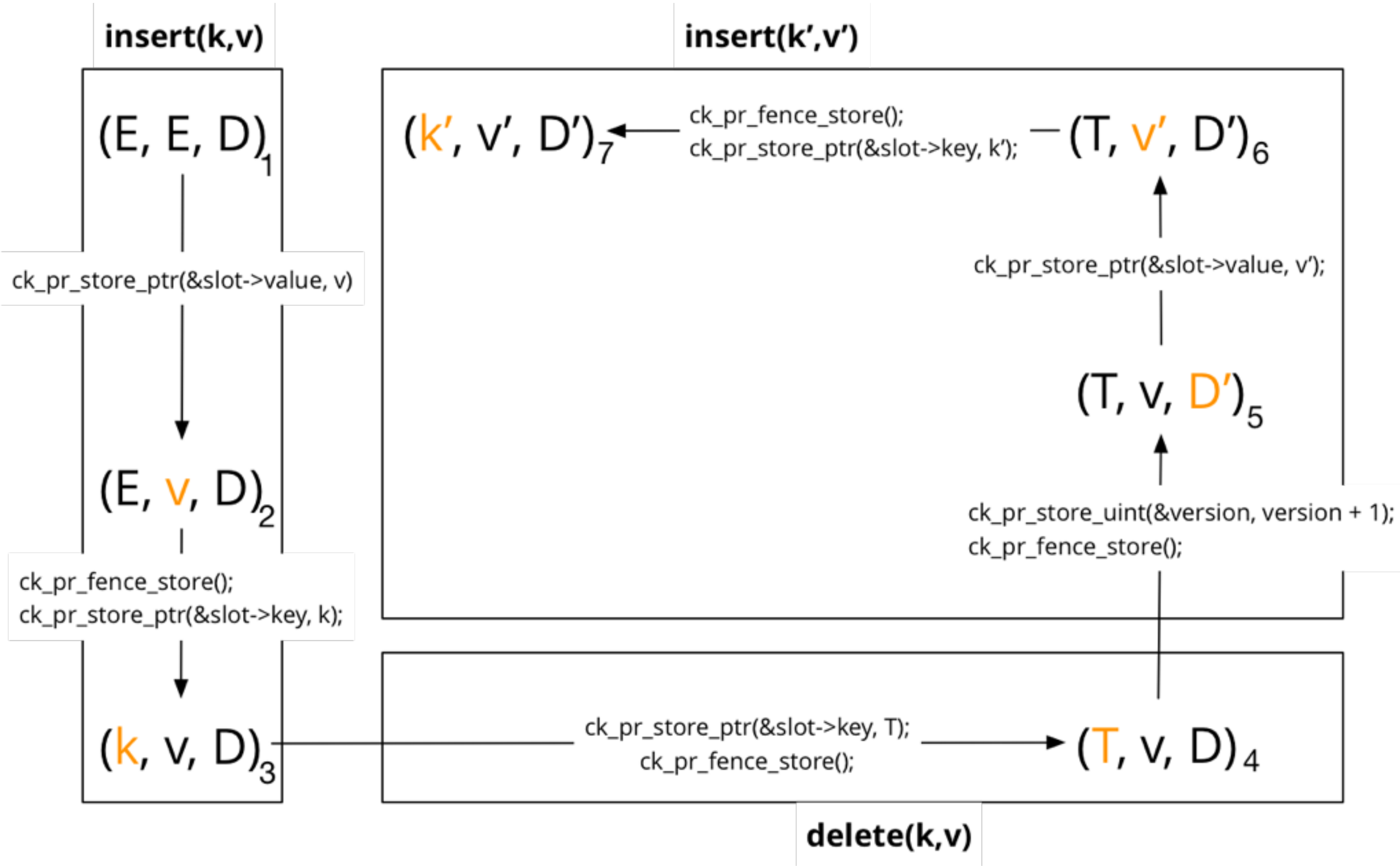
Correctness



Observed (D, E, v, D)

Backtrace

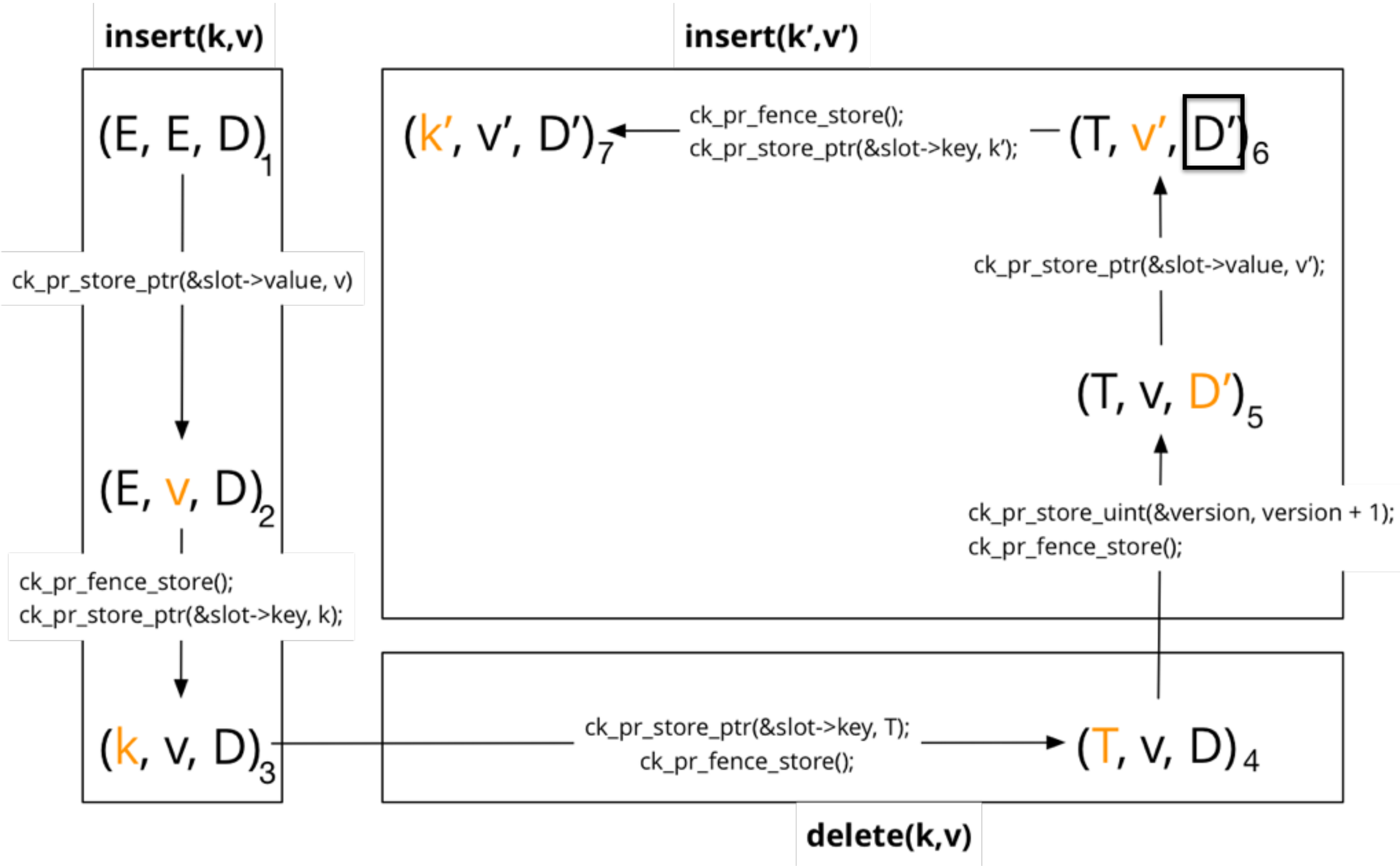
Correctness



Cannot observe (D', E, v, D)

Backtrace

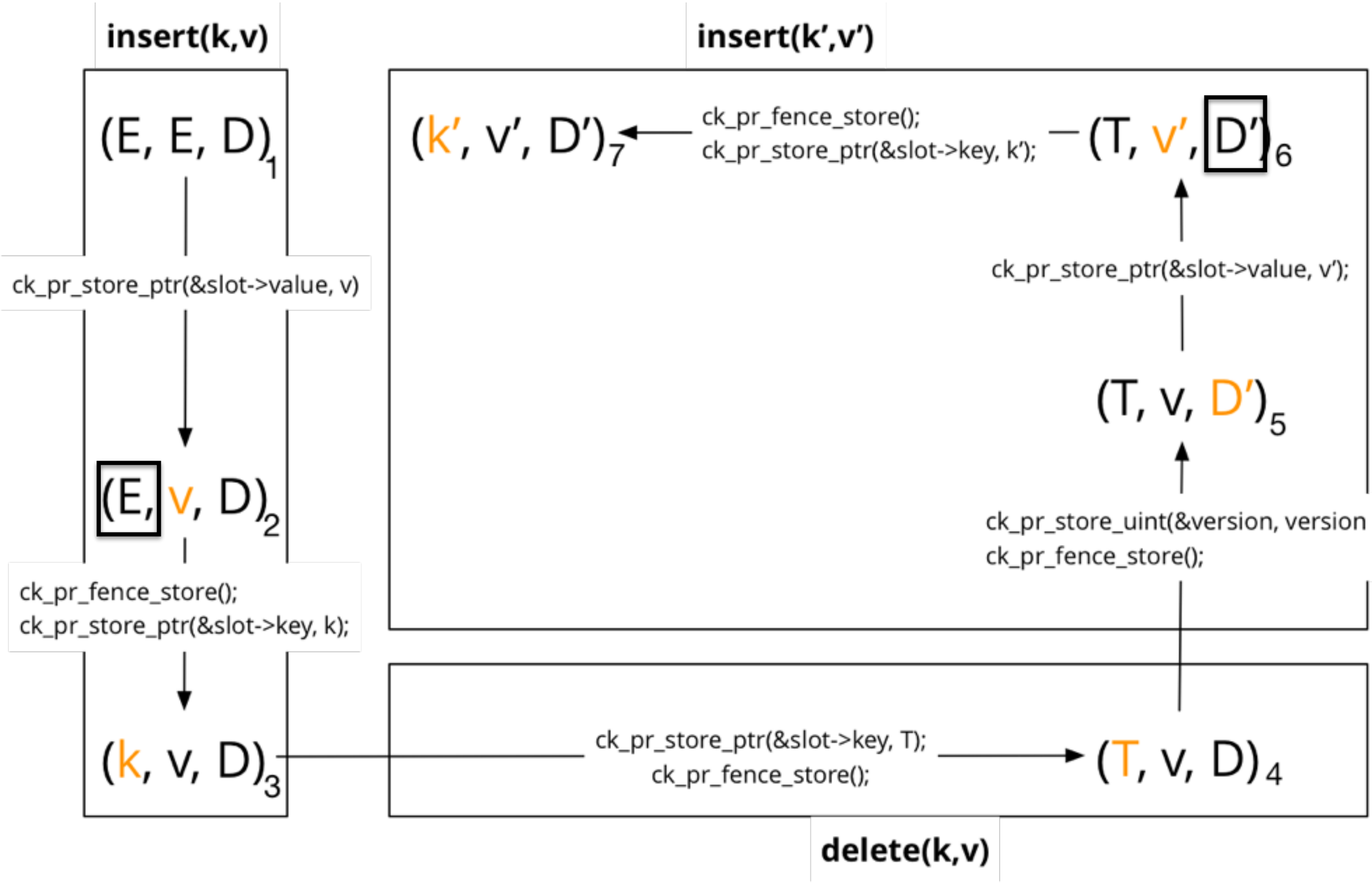
Correctness



Cannot observe (D', E, v, D)

Backtrace

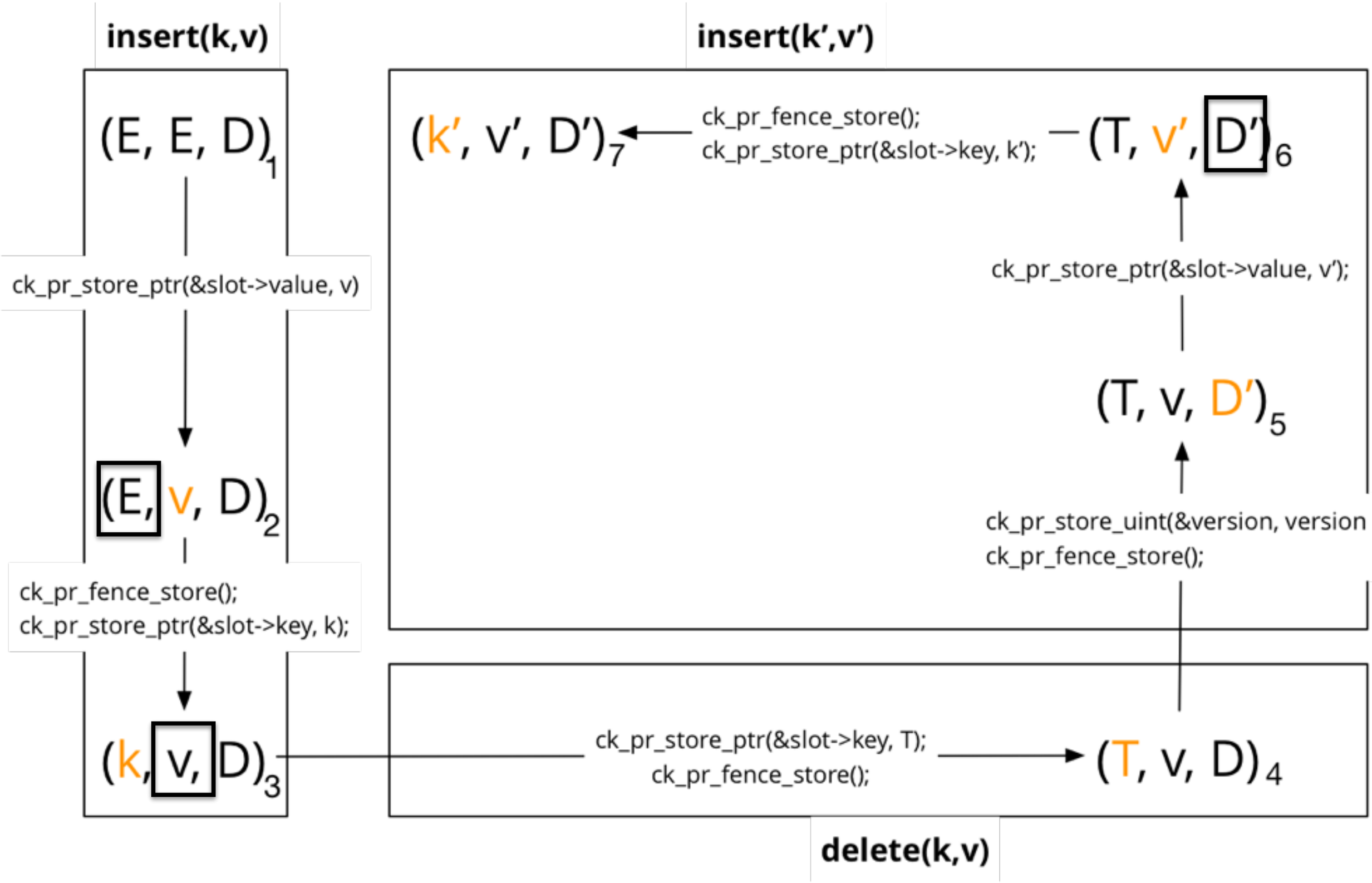
Correctness



Cannot observe (D', E, v, D)

Backtrace

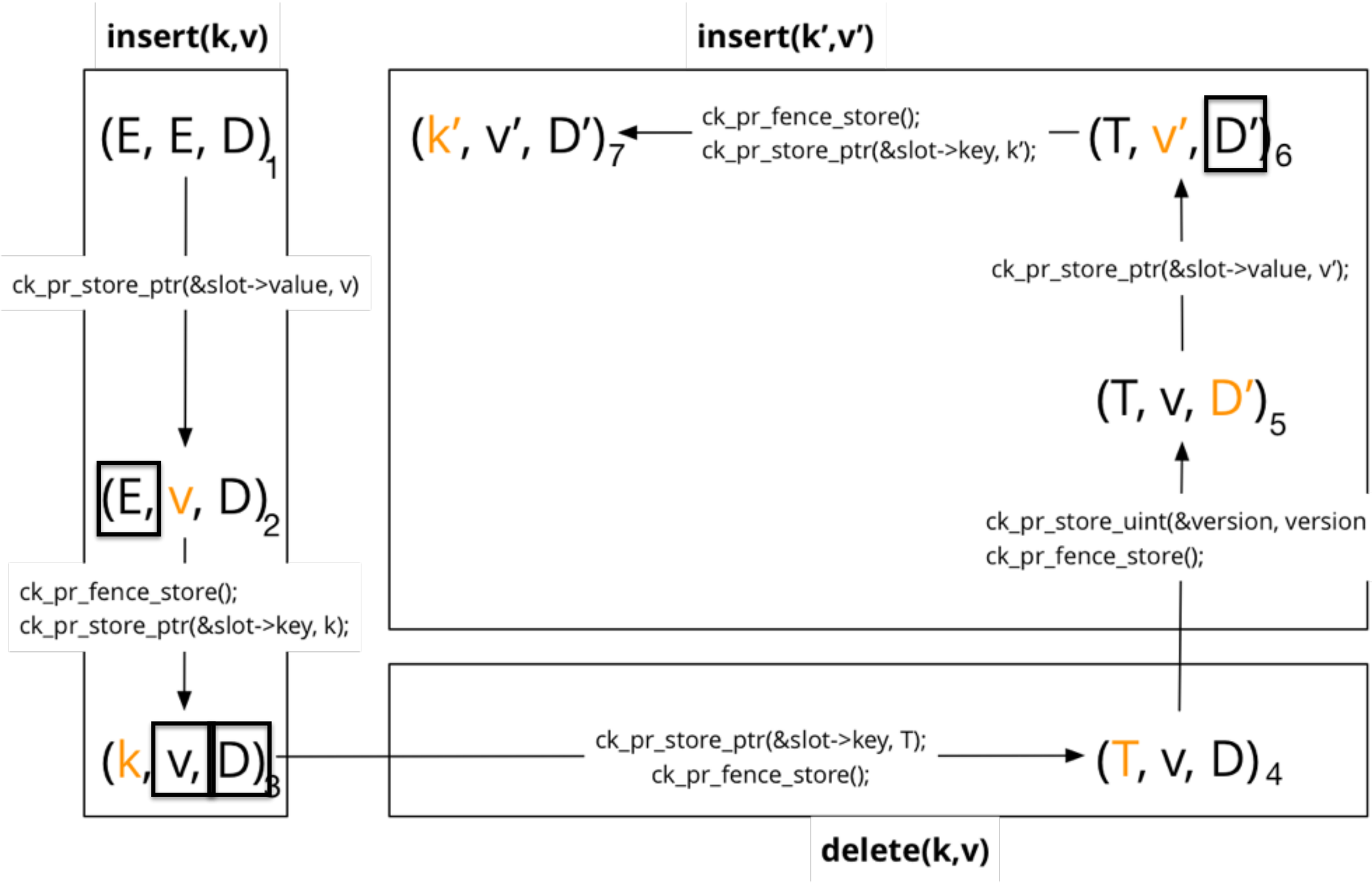
Correctness



Cannot observe (D’, E, v, D)

Backtrace

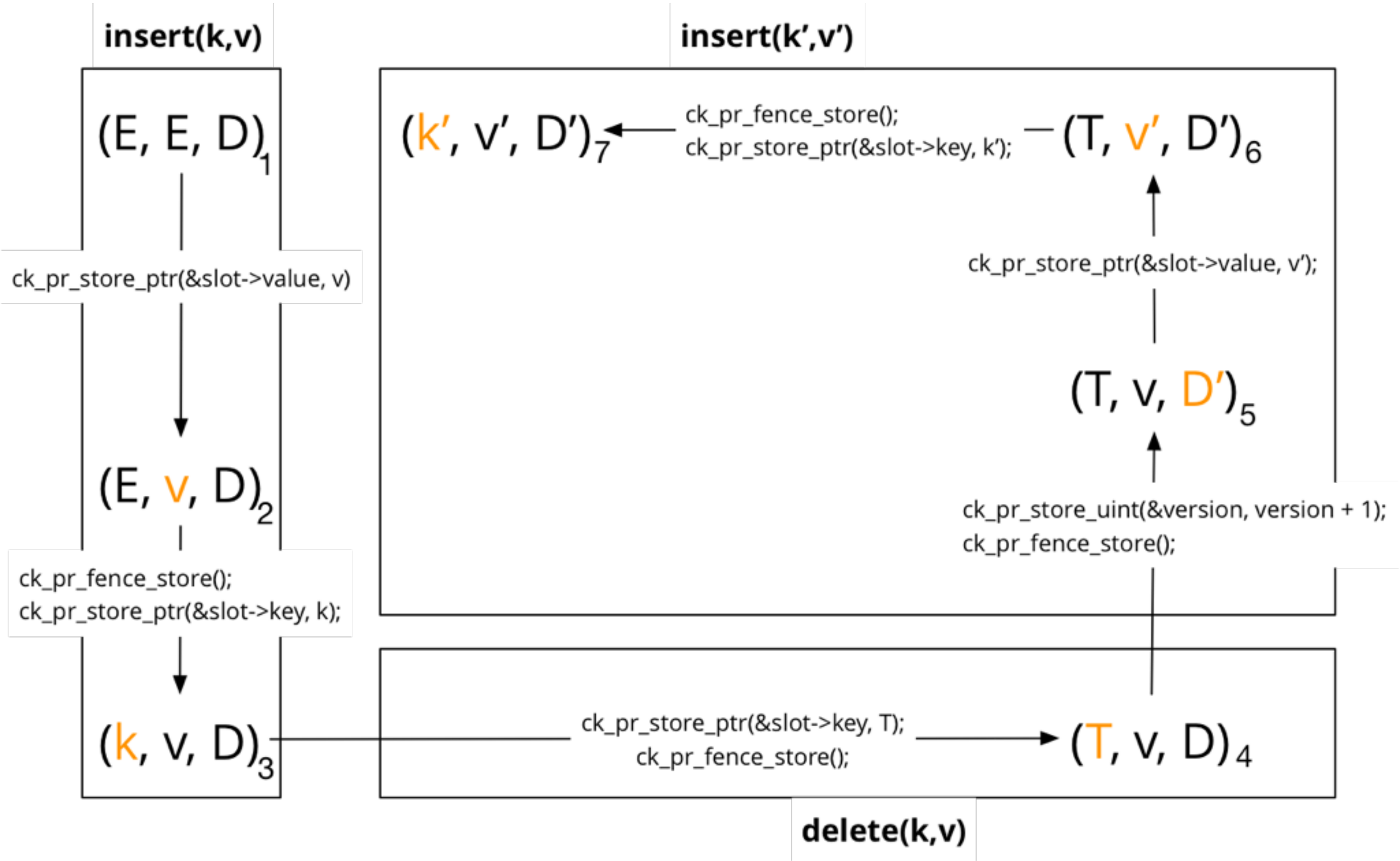
Correctness



Cannot observe (D', E, v, D)

Backtrace

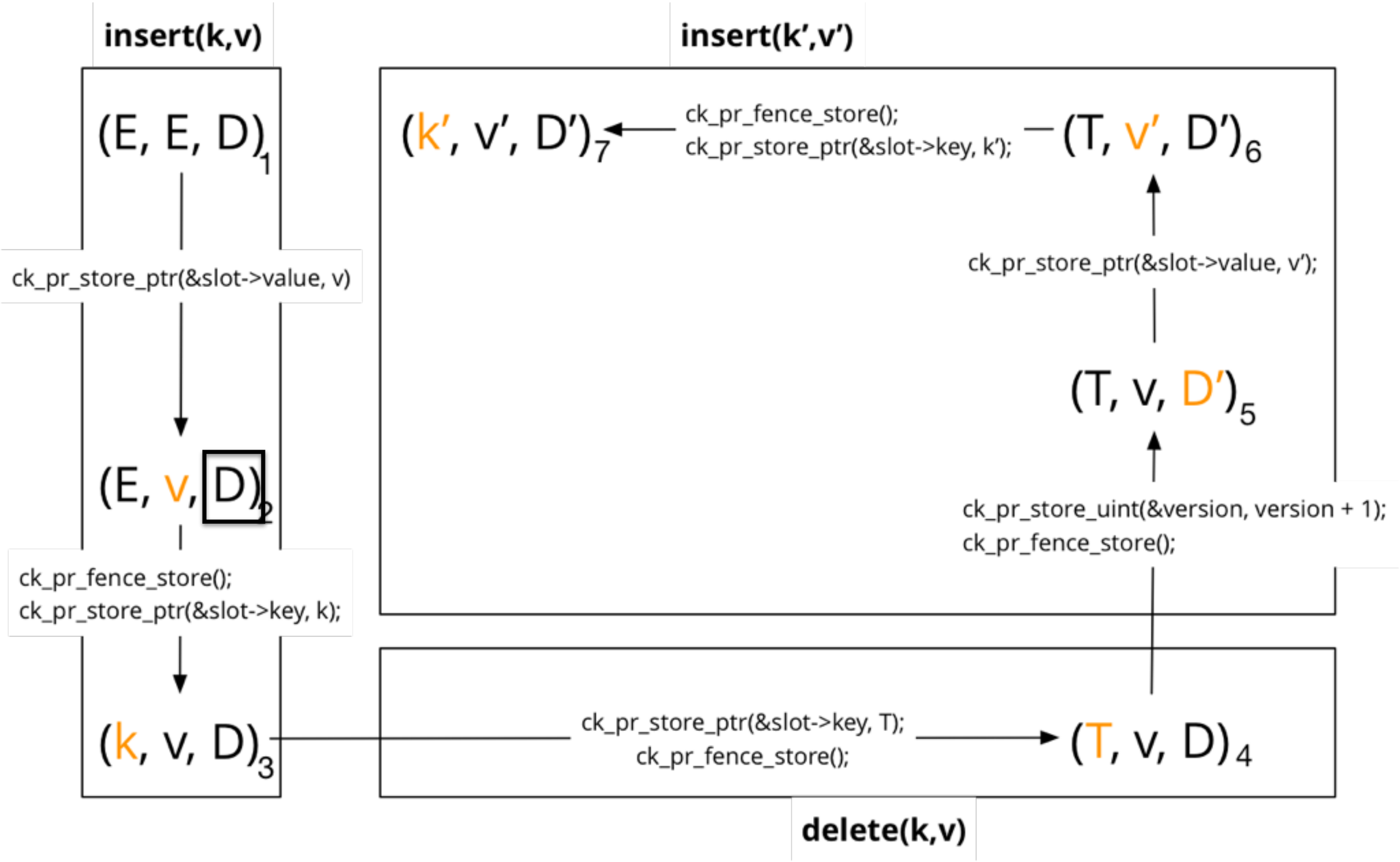
Correctness



If (k, v') is observed, then so is D and D'.

Backtrace

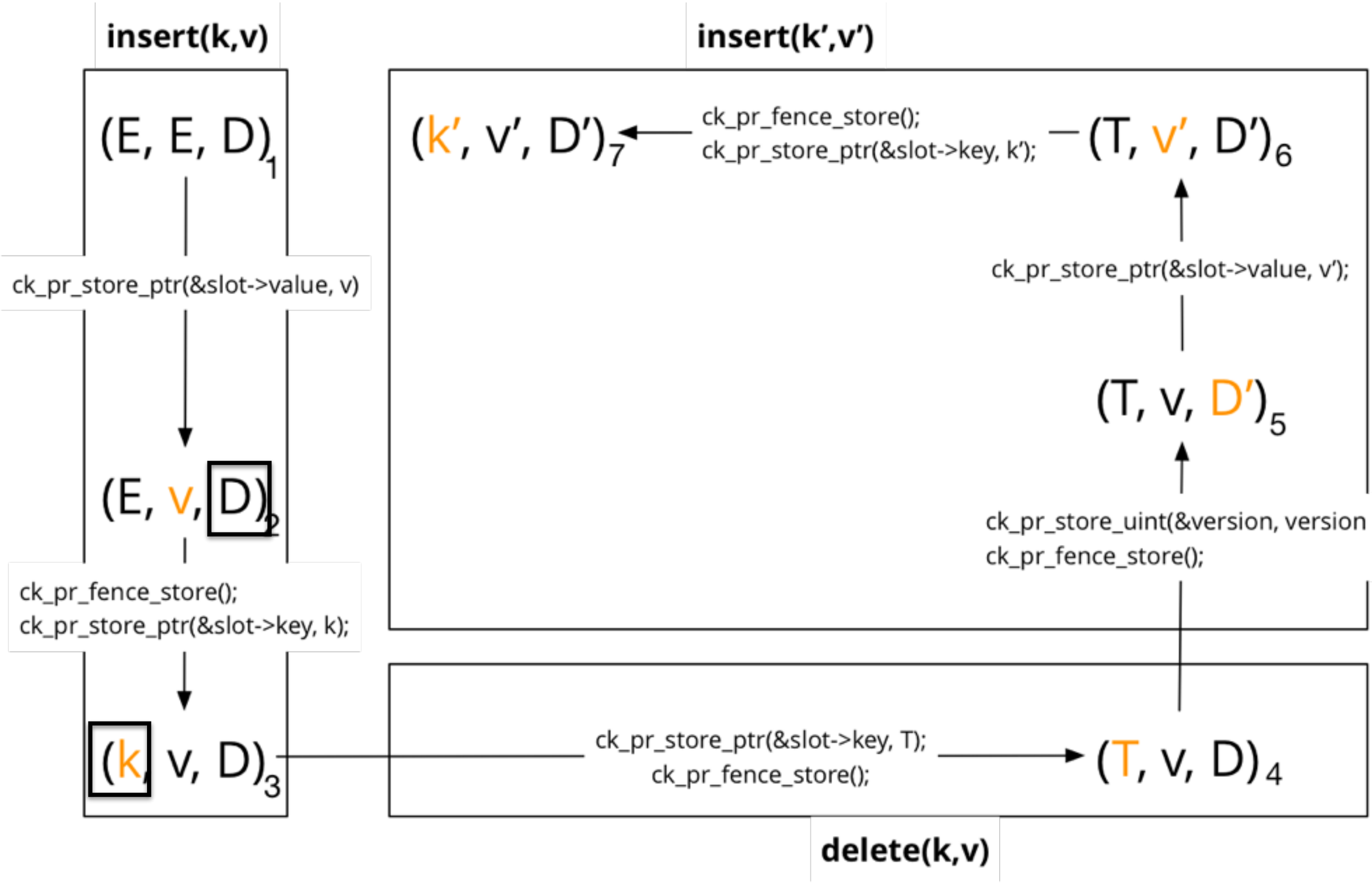
Correctness



If (k, v') is observed, then so is D and D'.

Backtrace

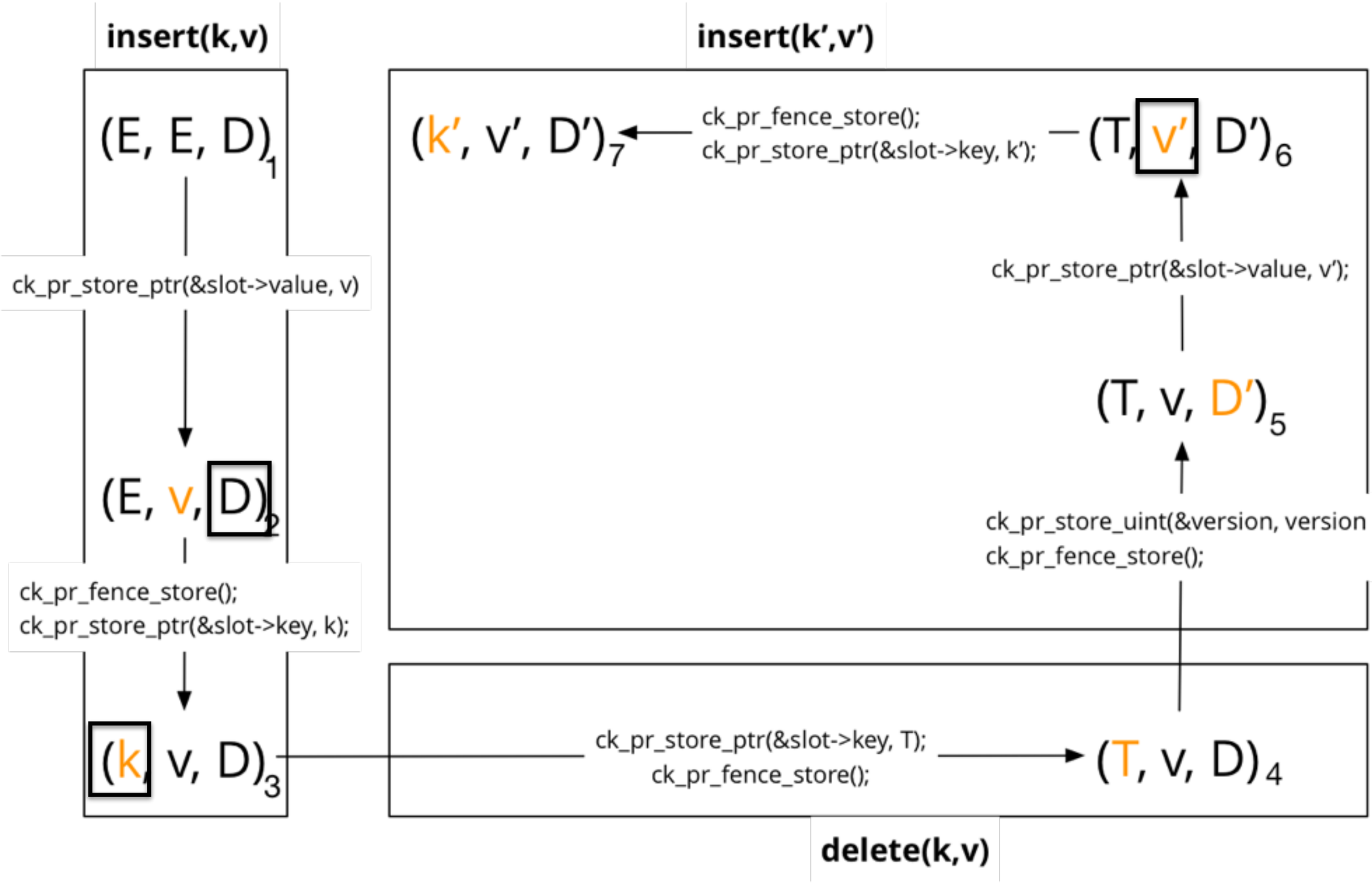
Correctness



If (k, v') is observed, then so is D and D'.

Backtrace

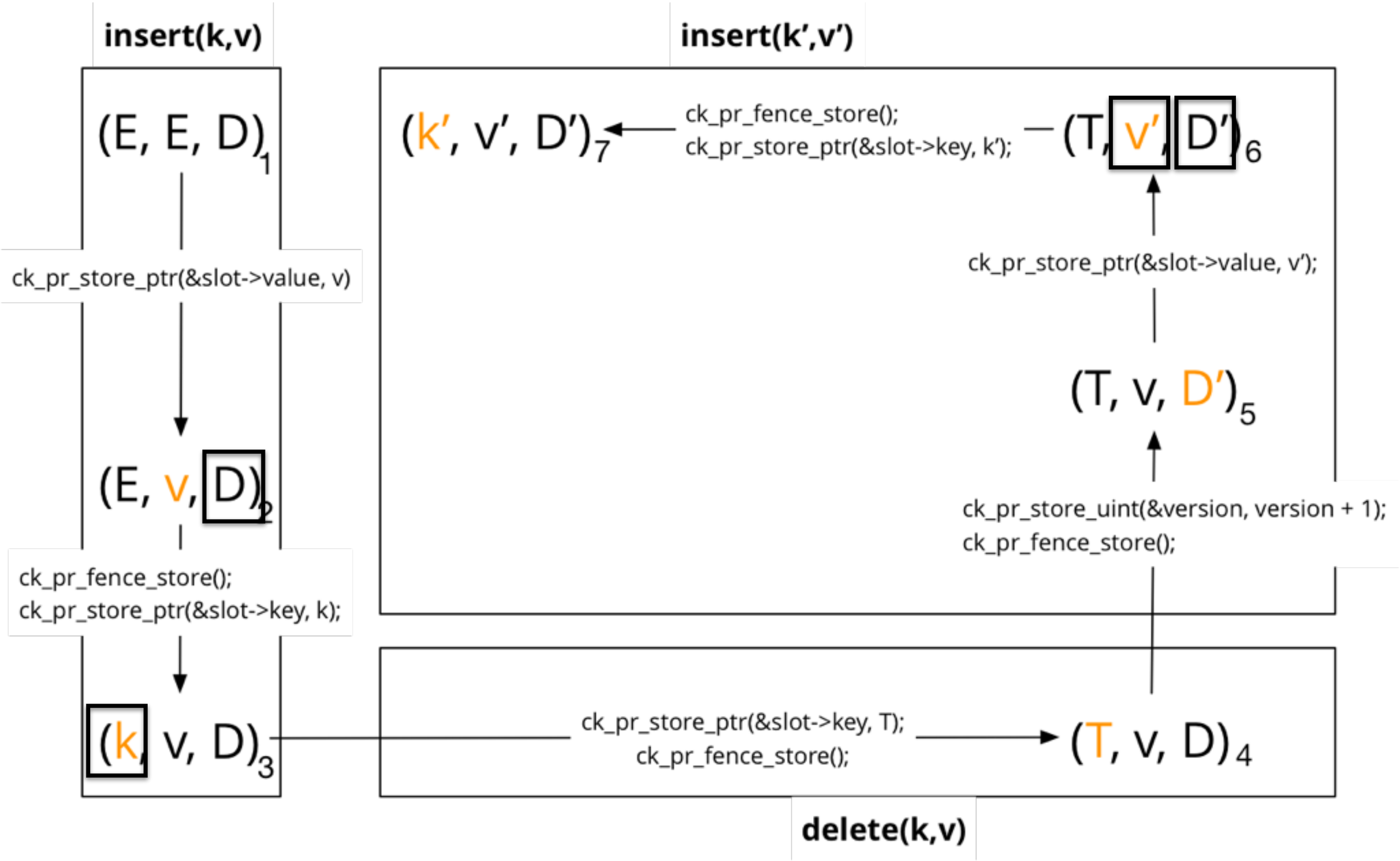
Correctness



If (k, v') is observed, then so is D and D'.

Backtrace

Correctness

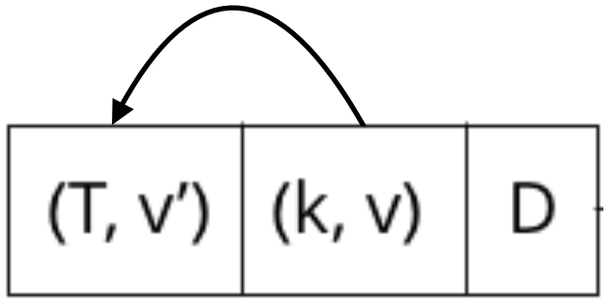


If (k, v') is observed, then so is D and D'.

Backtrace

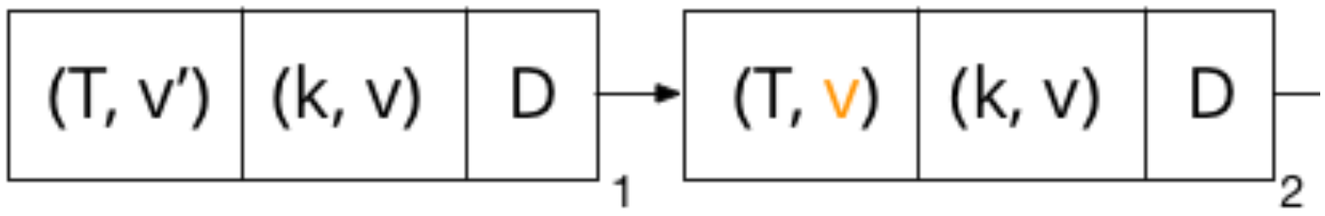
What about more sophisticated collision resolution techniques?

Probe Sequence Mutation



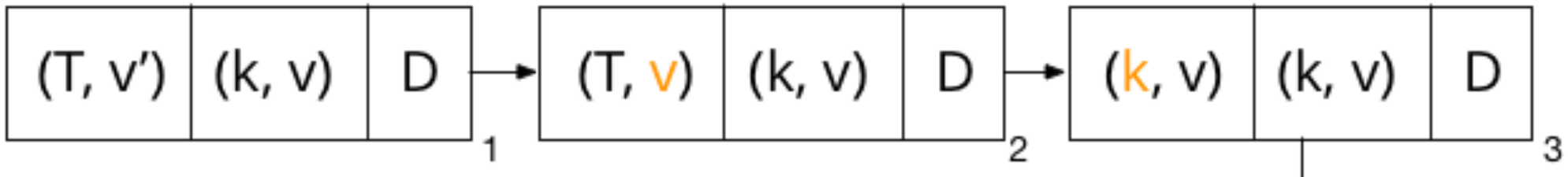
Probe Sequence Mutation

Insert new key-value pair.



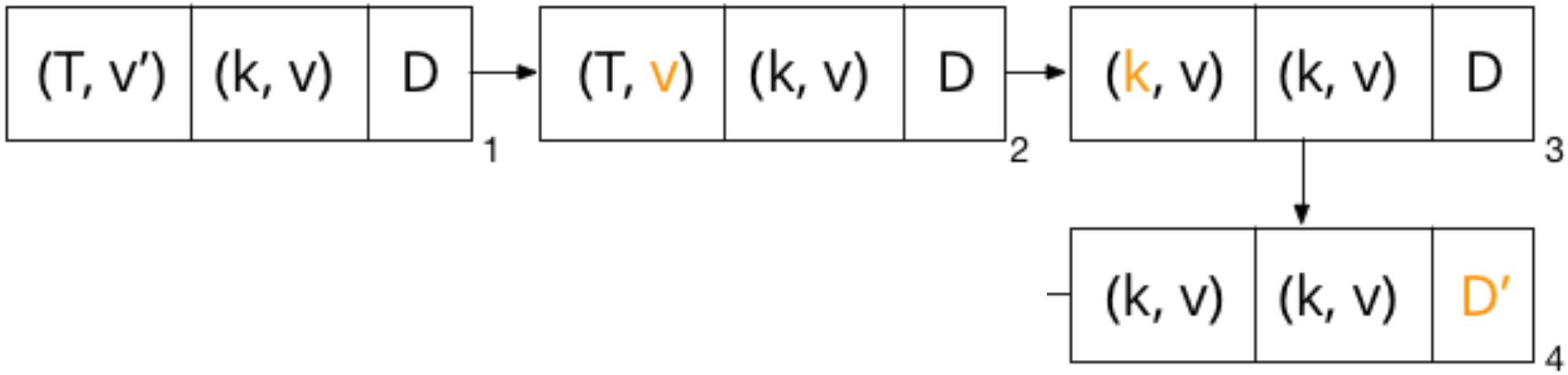
Probe Sequence Mutation

Insert new key-value pair.



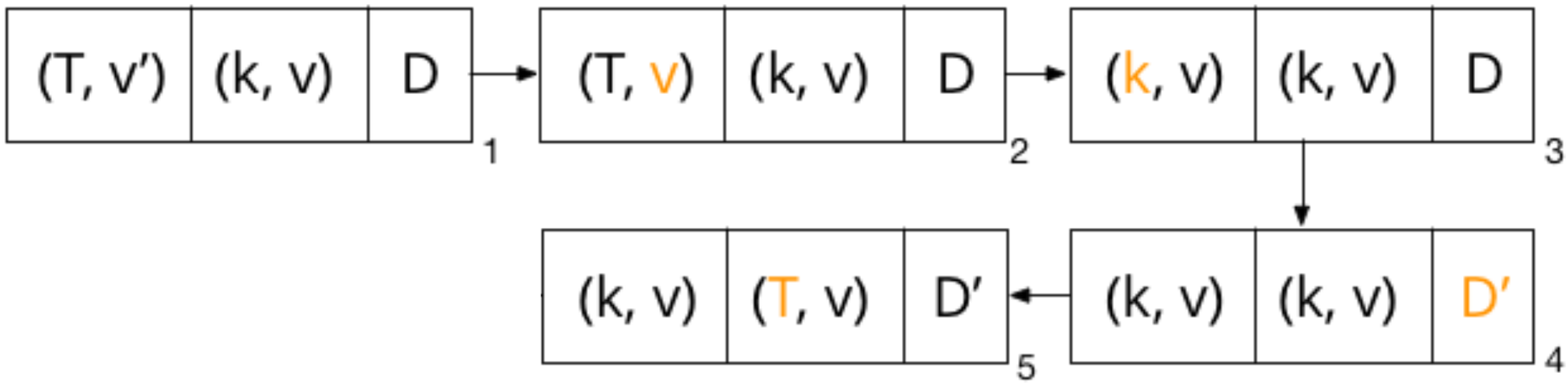
Probe Sequence Mutation

Delete old key-value pair.



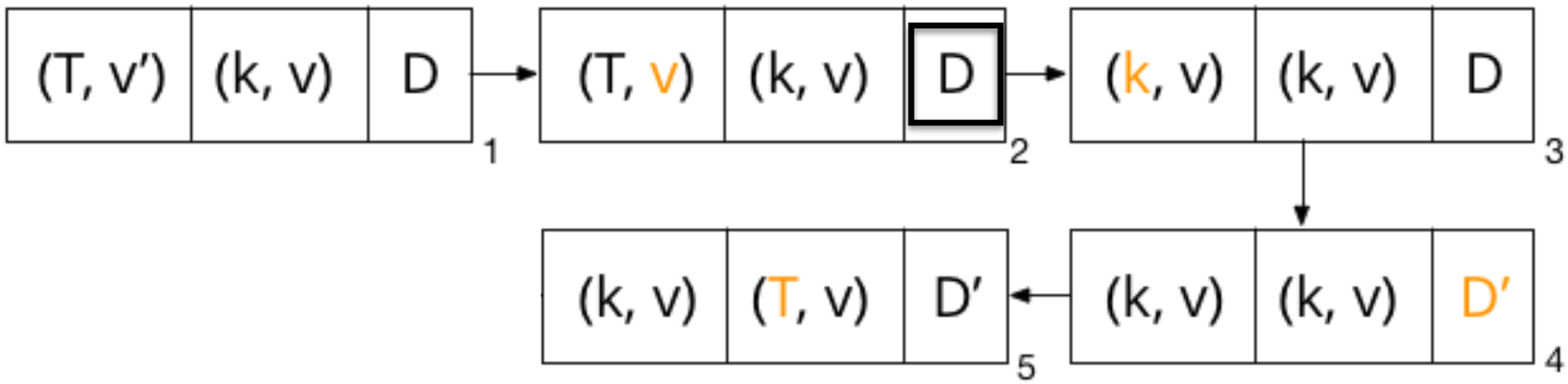
Probe Sequence Mutation

Delete old key-value pair.



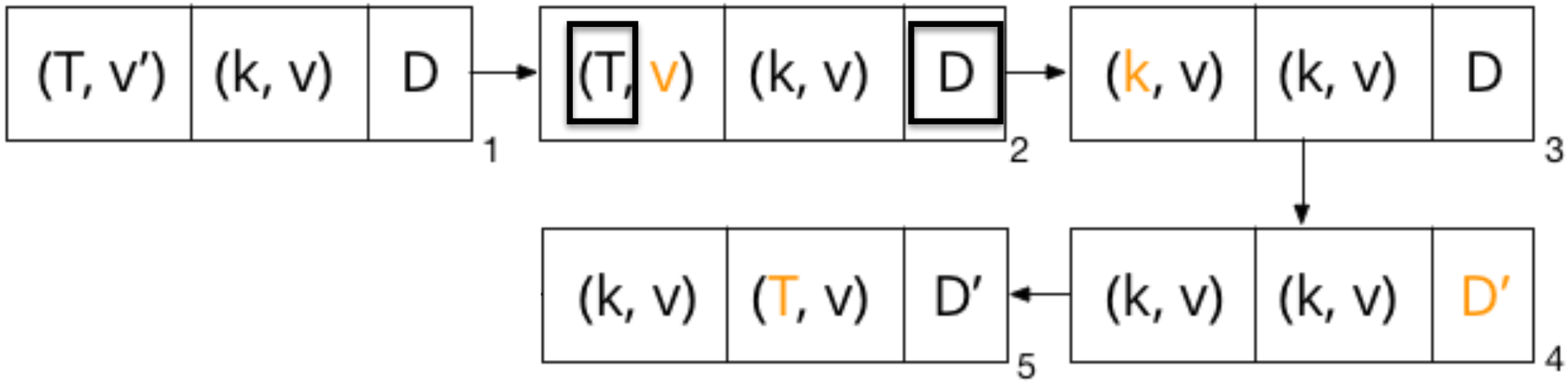
Probe Sequence Mutation

Delete old key-value pair.



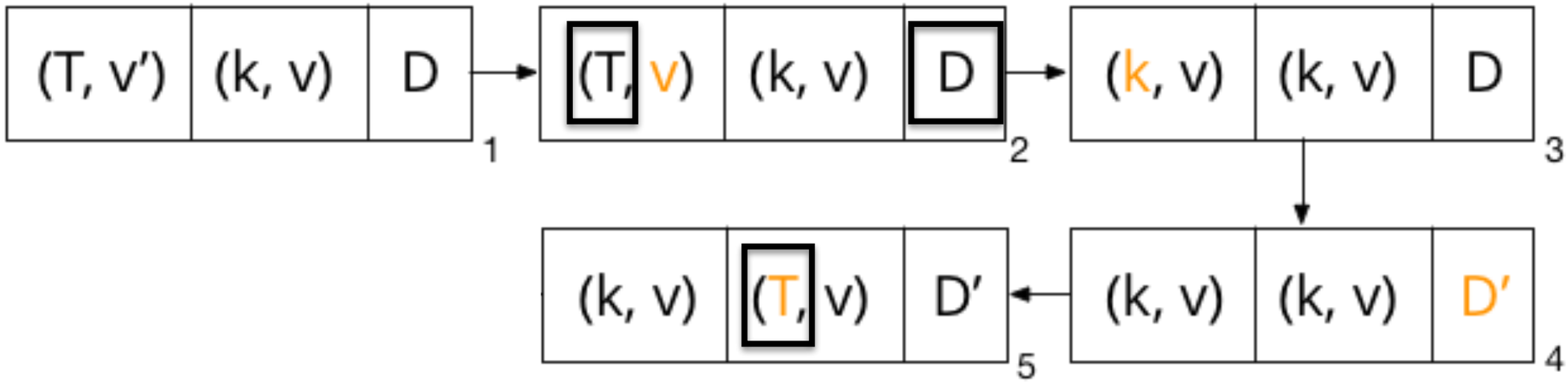
Probe Sequence Mutation

Delete old key-value pair.



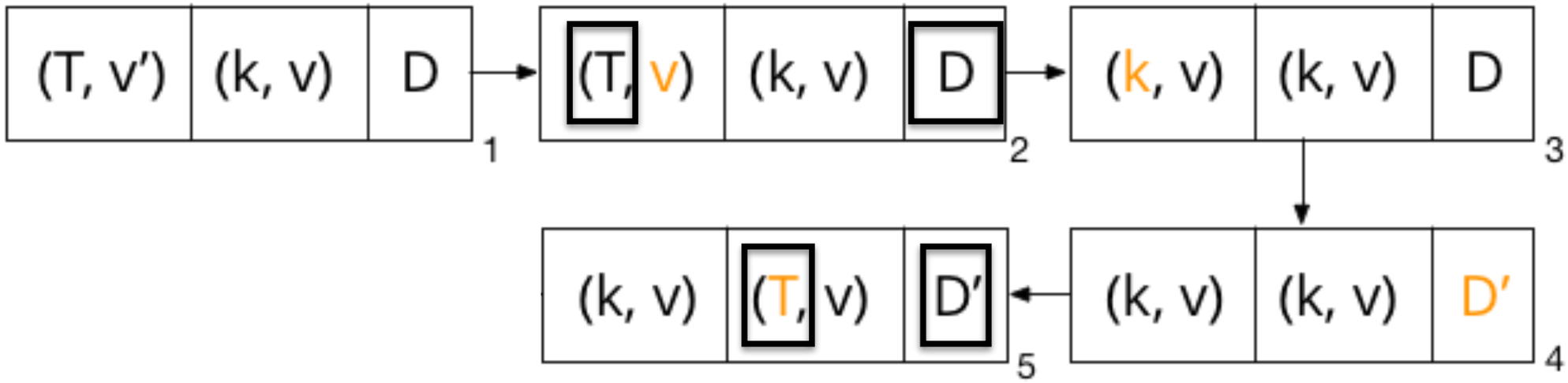
Probe Sequence Mutation

Delete old key-value pair.



Probe Sequence Mutation

Delete old key-value pair.



Further Specialization

Sacrifice write-side progress guarantees for read-side wait-freedom.

Recent Updates

Technical Report

Designing ASCY-compliant Concurrent Search Data Structures

A hash table and a binary search tree

Tudor David Rachid Guerroui Che Tong Vasileios Trigonakis *
Distributed Programming Lab (LPD), EPFL
name.surname@epfl.ch

Algorithmic Improvements for Fast Concurrent Cuckoo Hashing

Xiaozhou Li[1], David G. Andersen[2], Michael Kaminsky[3], Michael J. Freedman[1]

[1]*Princeton University,* [2]*Carnegie Mellon University,* [3]*Intel Labs*

The End

Write-up at:

http://backtrace.io/blog

Implementations at:

http://concurrencykit.org

Thanks to Maged Michael, Mathieu Desnoyers, Olivier Houchard, Paul Khuong, Paul McKenney, Theo Schlossnagle, Wez Furlong and the Backtrace team for feedback on article.