Efficient Lock-Free Durable Sets

SPLASH 2019, OOPSLA TRACK

YOAV ZURIEL, MICHAL FRIEDMAN, GALI SHEFFI, NACHSHON COHEN, AND EREZ PETRANK

Highly Efficient, Lock-Free, Durable, Scalable Sets

Unique keys

insert, remove and contains

Highly Efficient, Lock-Free, Durable, Scalable **Sets**

Unique keys

insert, remove and contains

Recoverable on Persistent Memory

Highly Efficient, Lock-Free, **Durable**, Scalable Sets

Unique keys

insert, remove and contains

Recoverable on Persistent Memory

Highly Efficient, Lock-Free, Durable, **Scalable** Sets

Up to 3.3x Faster than Existing Implementations

Unique keys

insert, remove and contains

Recoverable on Persistent Memory

Highly Efficient, **Lock-Free**, Durable, Scalable Sets

Up to 3.3x Faster than Existing Implementations

Progress Guarantee

Unique keys

insert, remove and contains

Recoverable on Persistent Memory

Highly Efficient, Lock-Free, Durable, Scalable Sets

Tailored Lightweight Memory Management

Up to 3.3x Faster than Existing Implementations

Progress Guarantee

Sets Are Everywhere

druid

redis

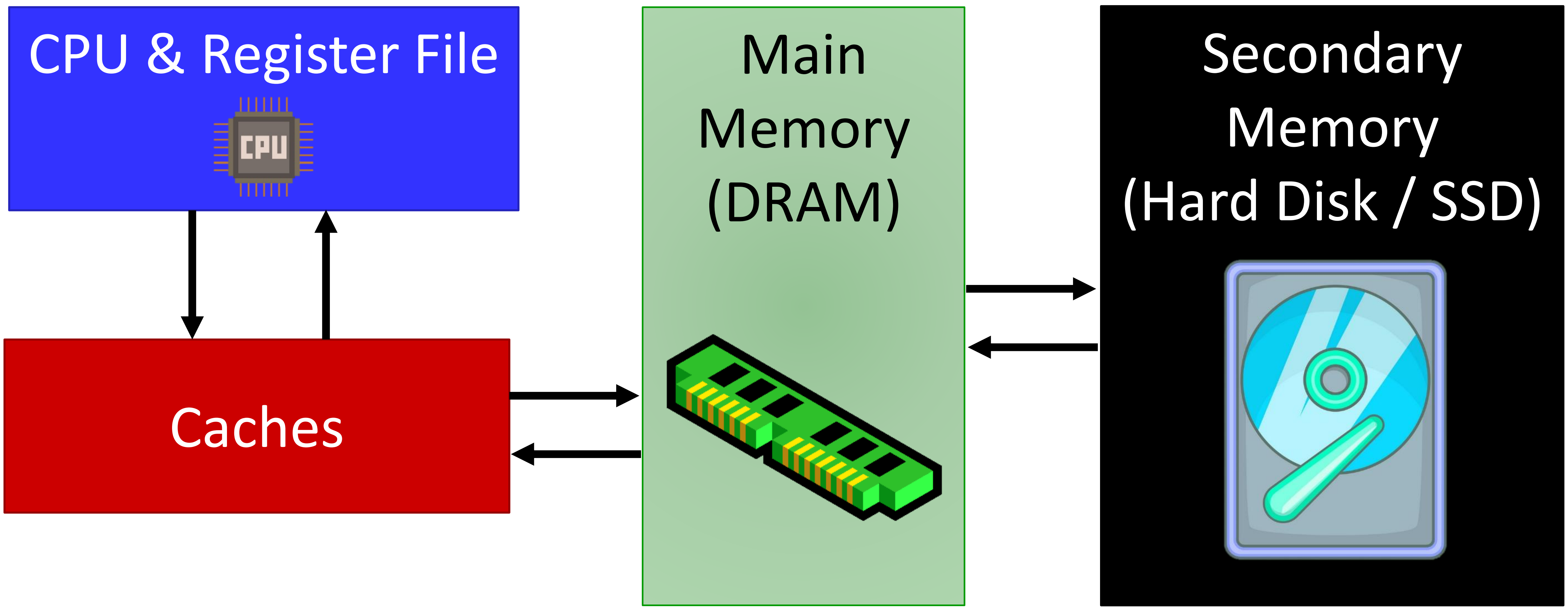
mongoDB®

MEMCACHED

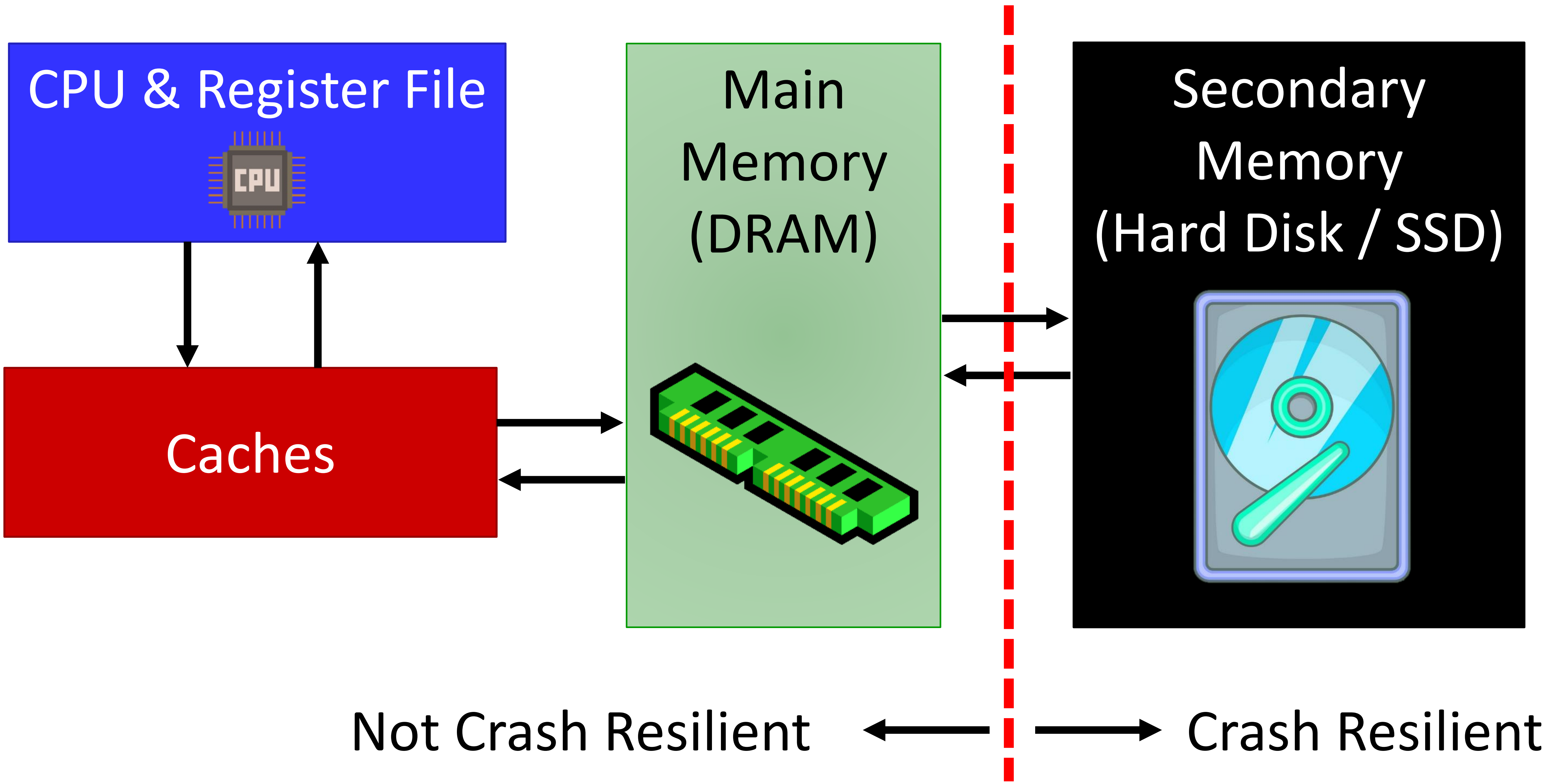
LEVELDB

RocksDB

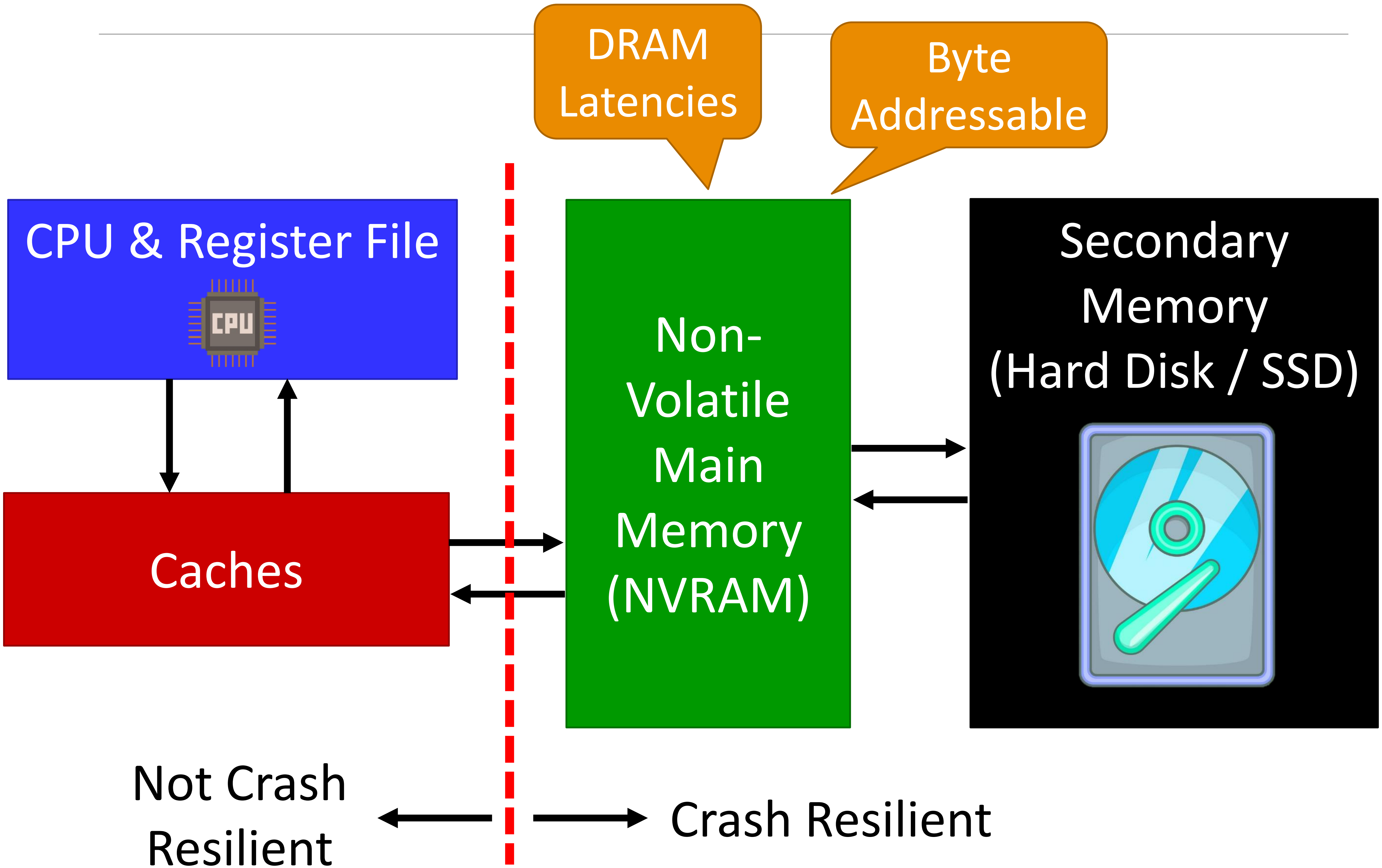
Once Upon a Time...



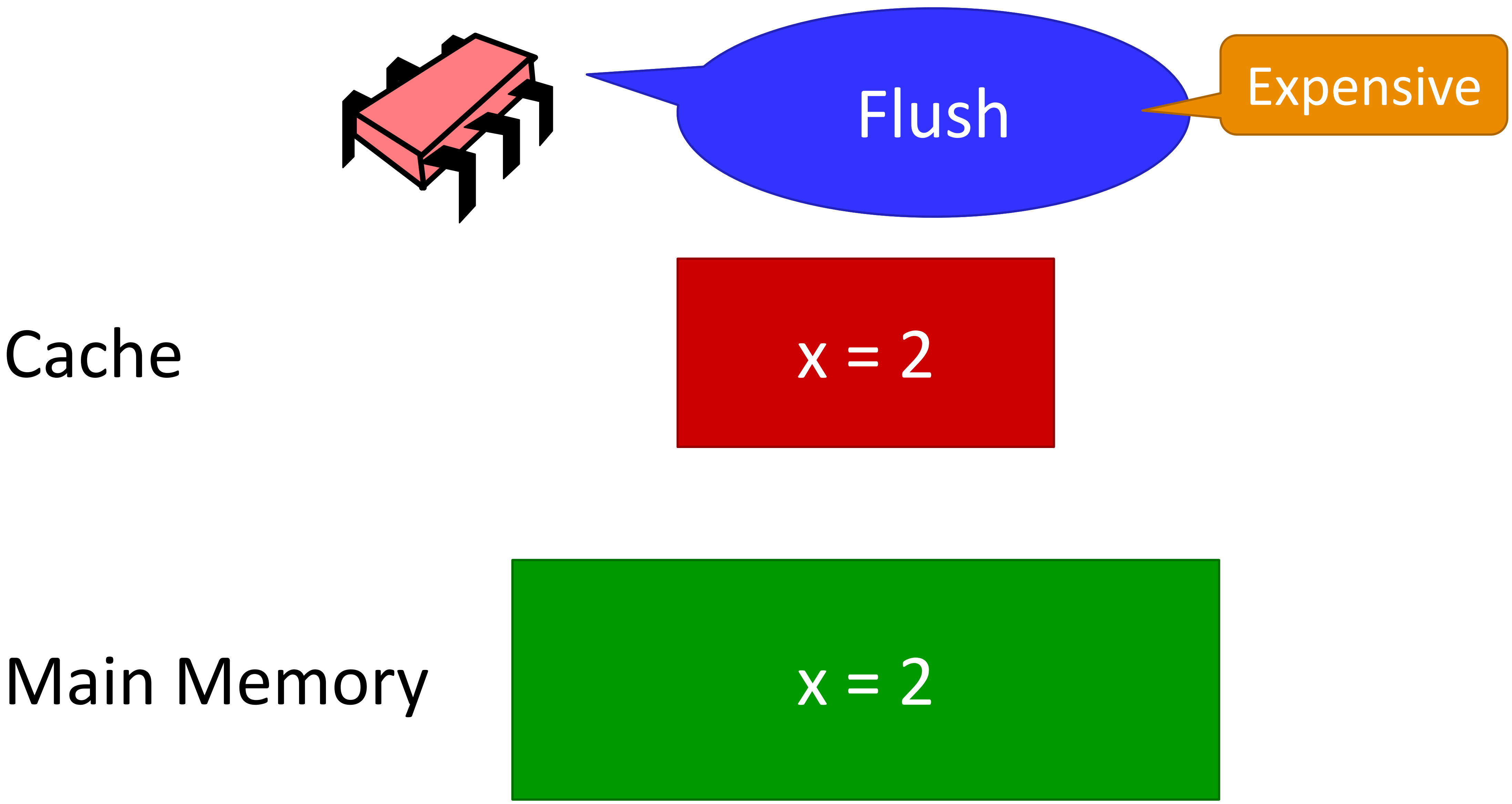
Once Upon a Time...



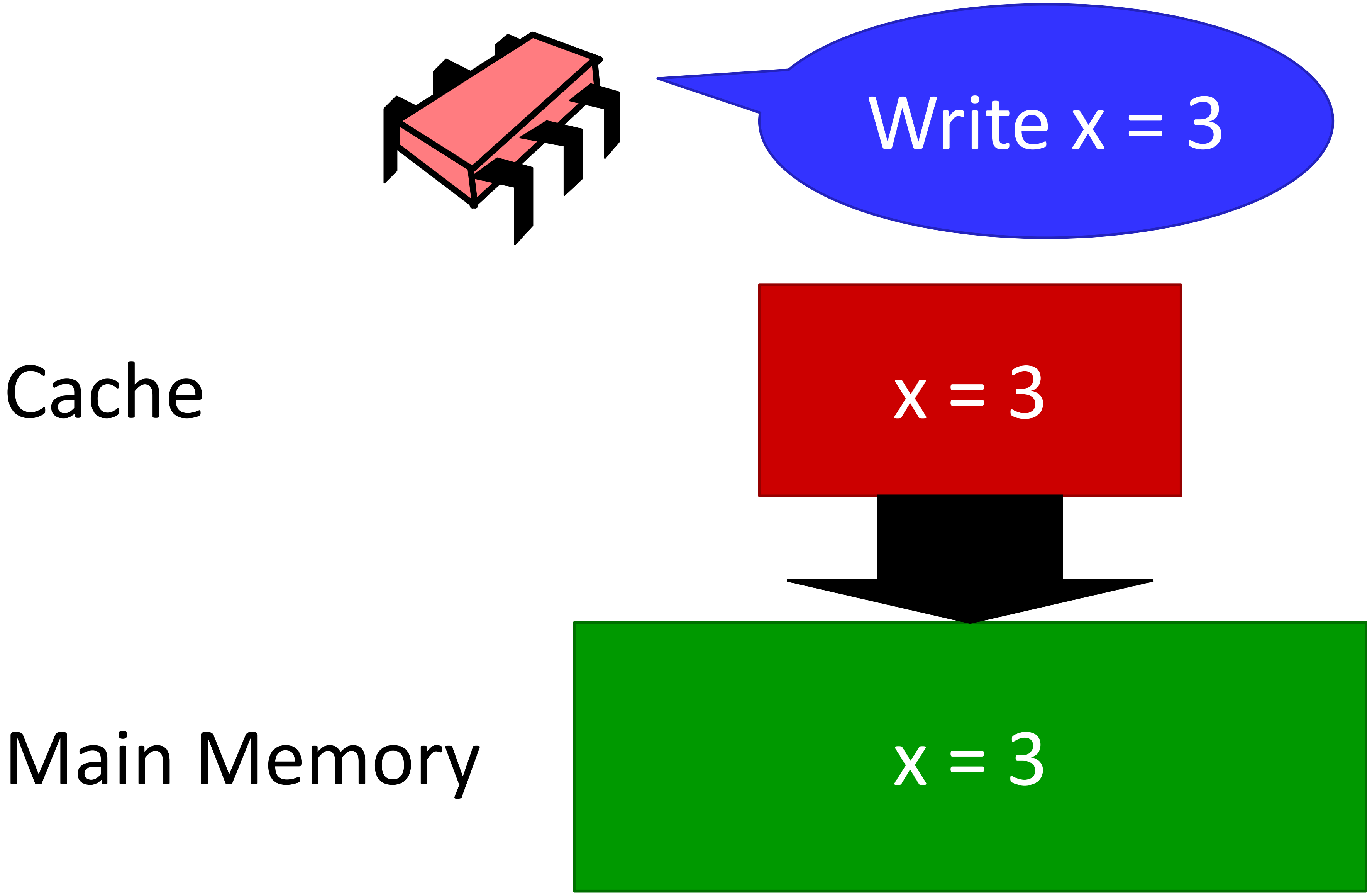
Now


DRAM Latencies
Byte Addressable
CPU & Register File
CPU
Caches
Non-Volatile Main Memory (NVRAM)
Secondary Memory (Hard Disk / SSD)
Not Crash Resilient
Crash Resilient

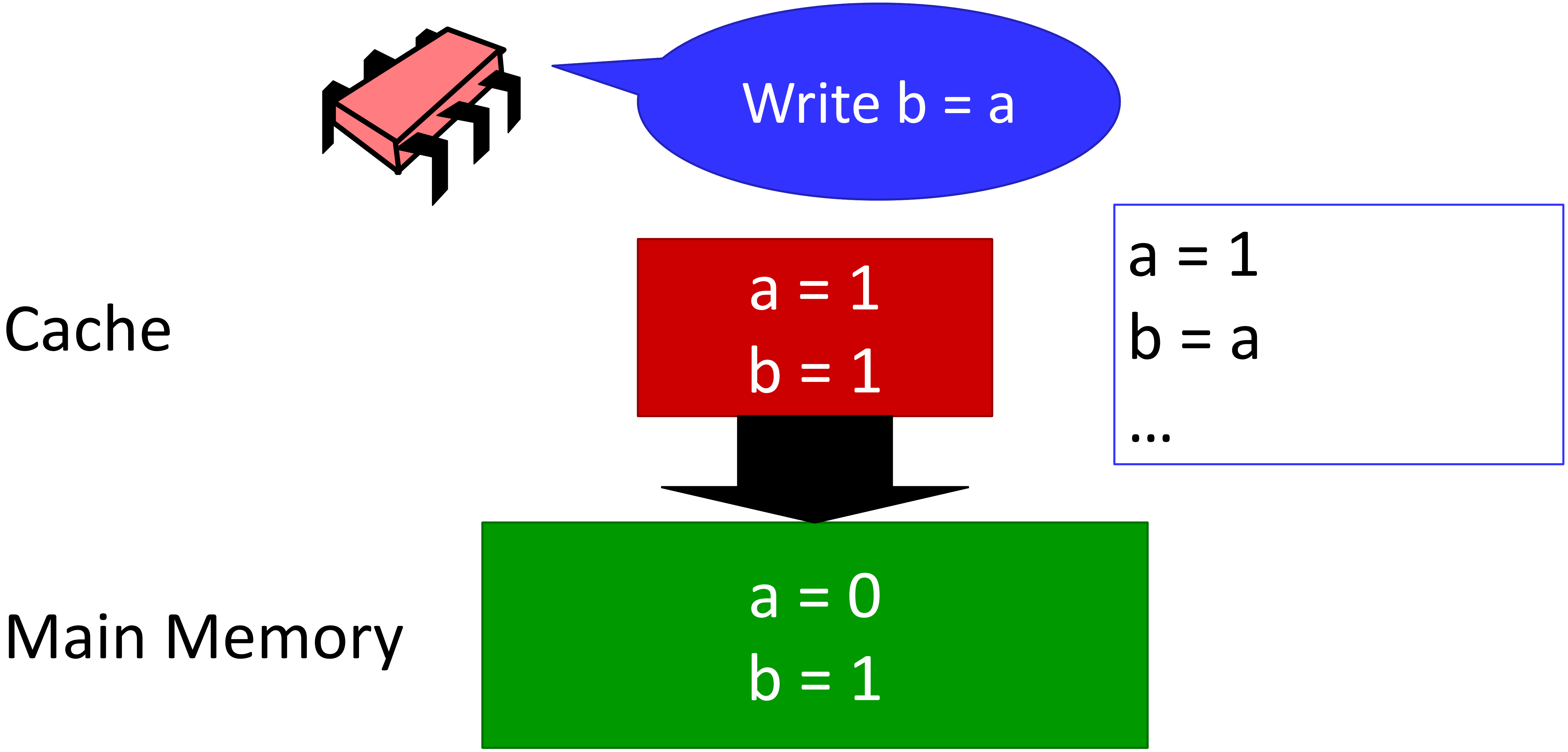
Dataflow: Explicit Flush



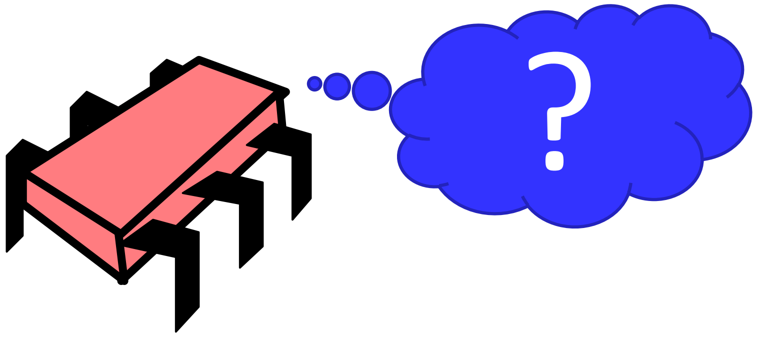
Dataflow: Implicit Flush



Consistency Problem?



Consistency Problem?



Cache

LOST

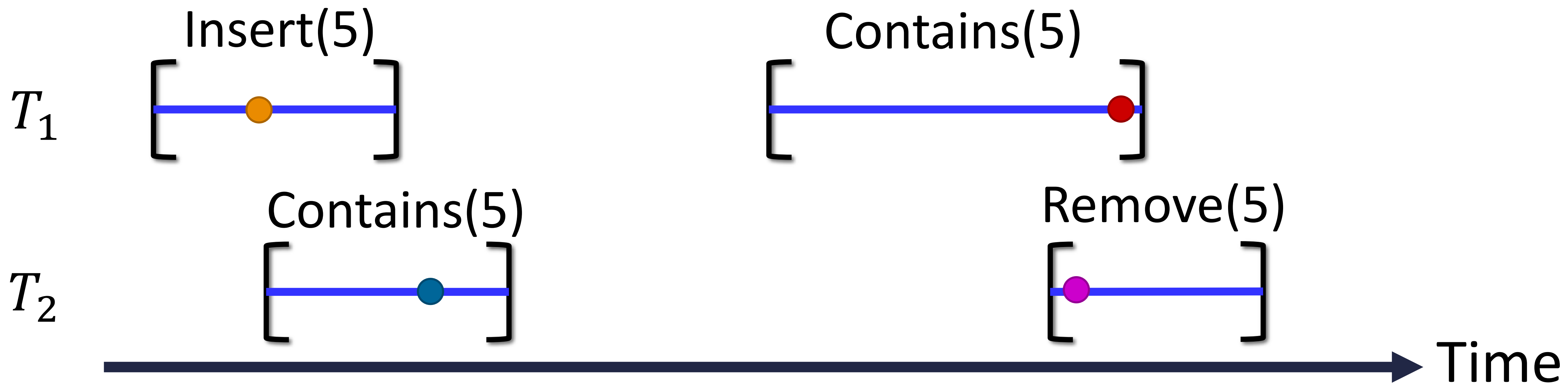
a = 1
b = a
...

Main Memory

a = 0
b = 1

Defining what is “correct” in a concurrent crash-able execution is challenging

For non-crashable execution: Linearizability

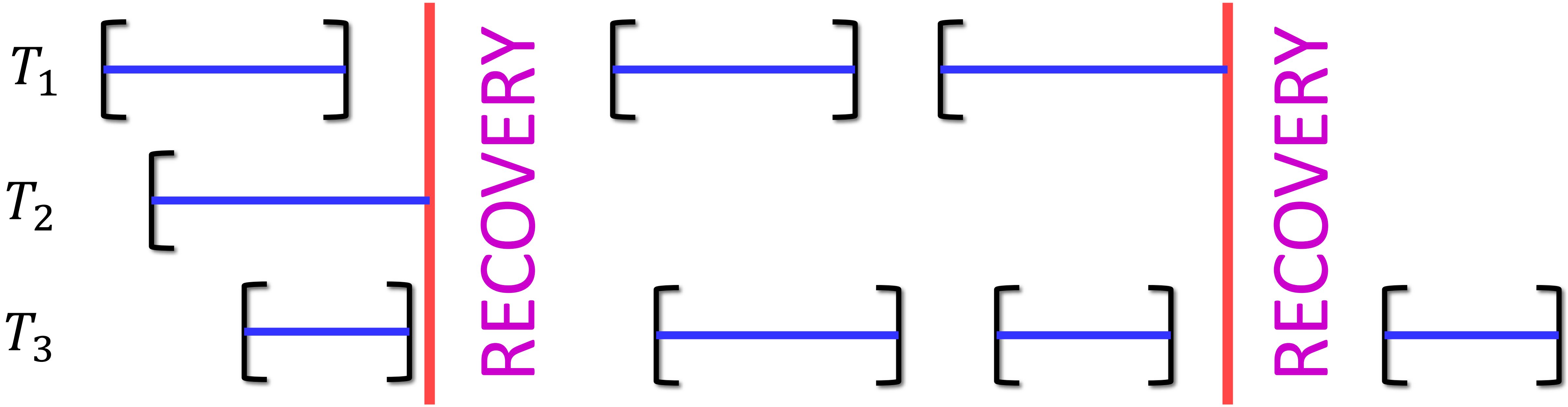


Defining what is "correct" in a concurrent crash-able execution is challenging

For non-crashable execution: Linearizability

For crashable execution: Durable Linearizability

The whole system crashes together

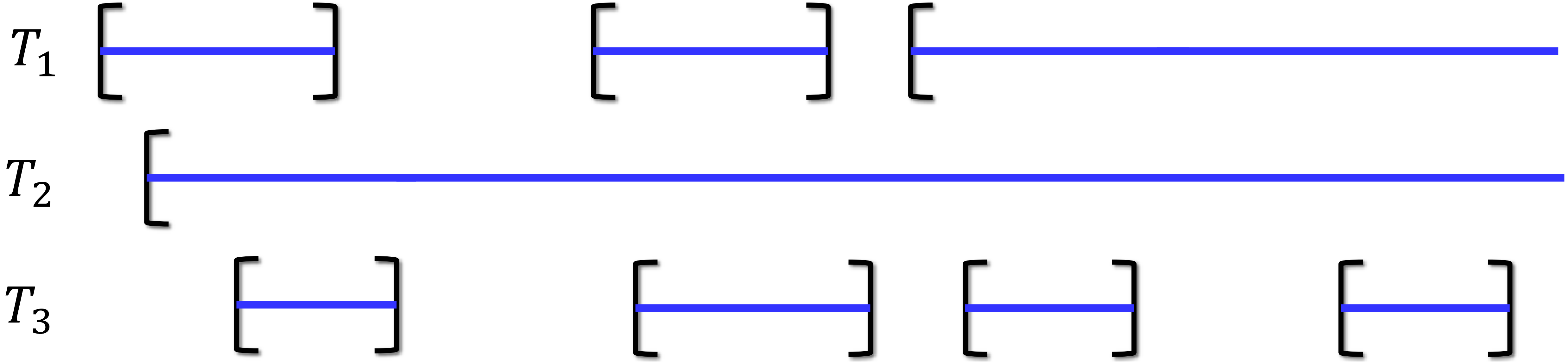


Defining what is "correct" in a concurrent crash-able execution is challenging

For non-crashable execution: Linearizability

For crashable execution: Durable Linearizability

Linearizable after removing crashes



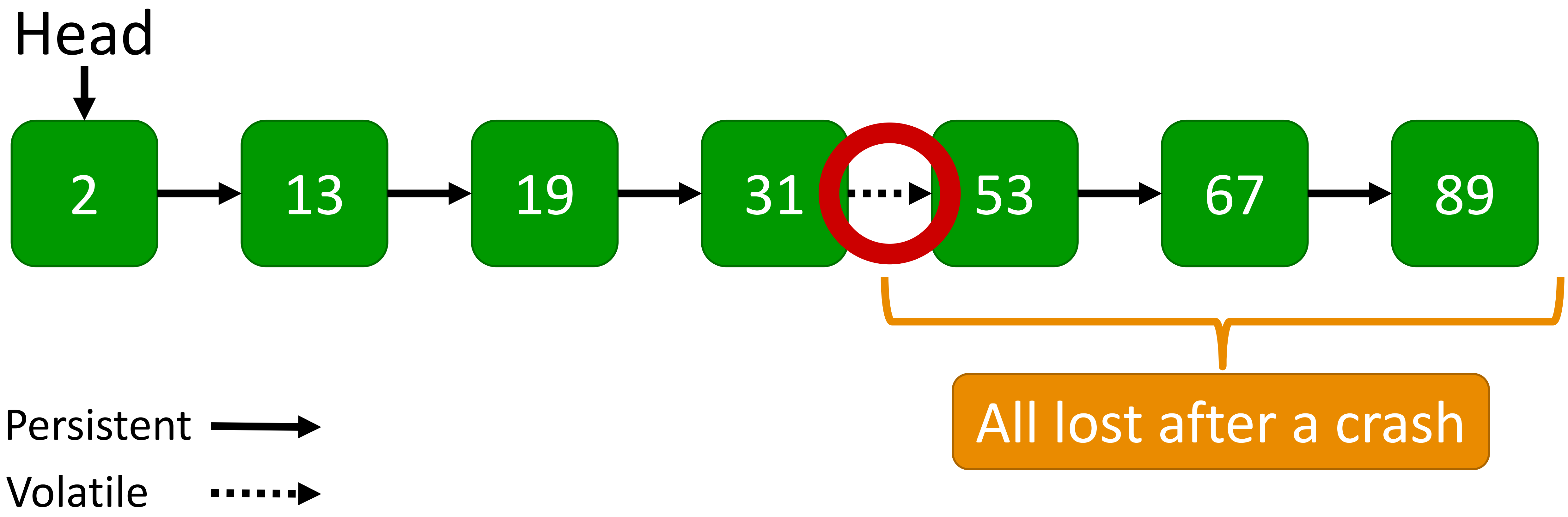
Defining what is "correct" in a concurrent crash-able execution is challenging

For non-crashable execution: Linearizability

For crashable execution: Durable Linearizability

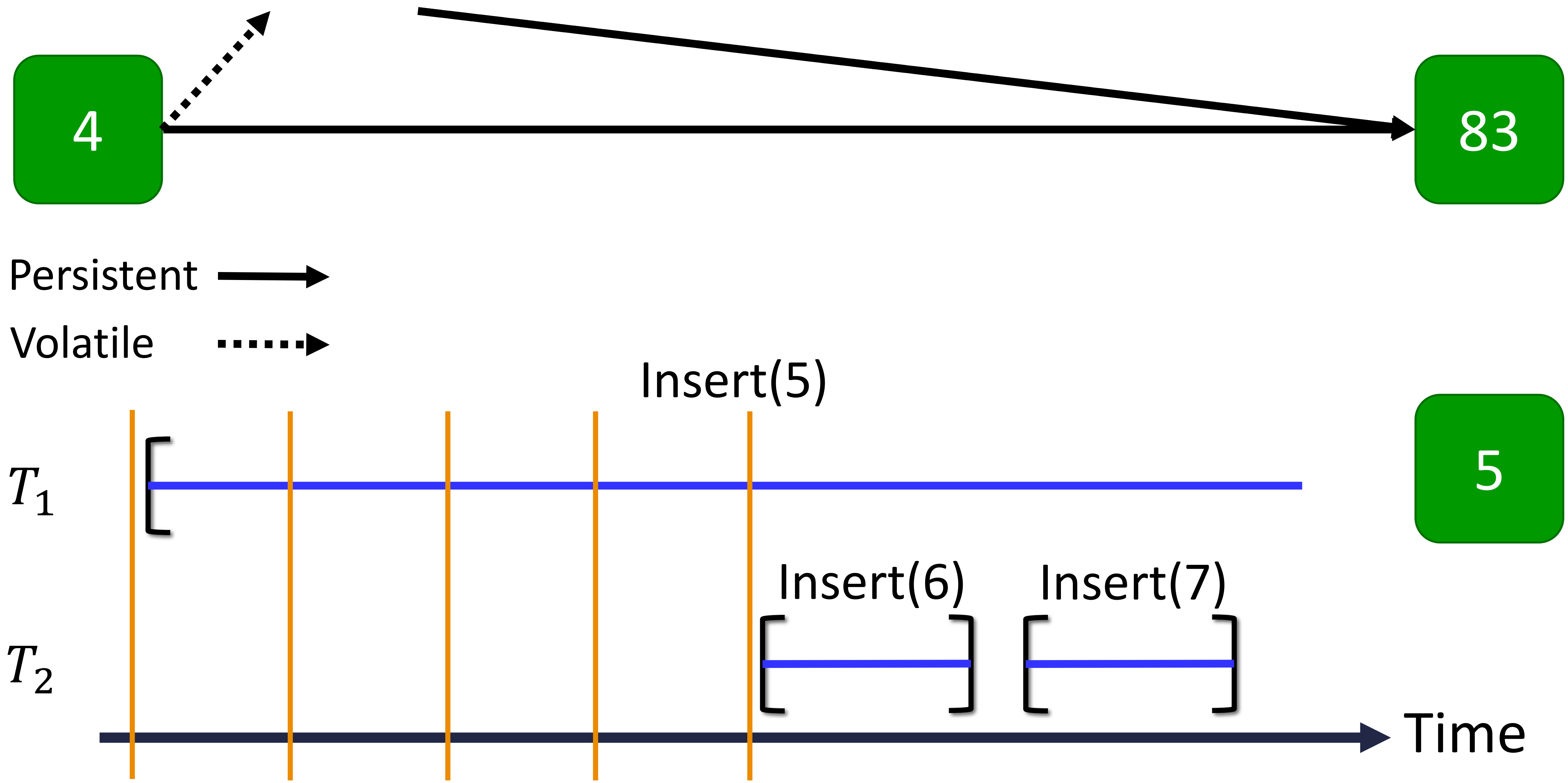
Our designs satisfy Durable Linearizability

The Persistence Challenge

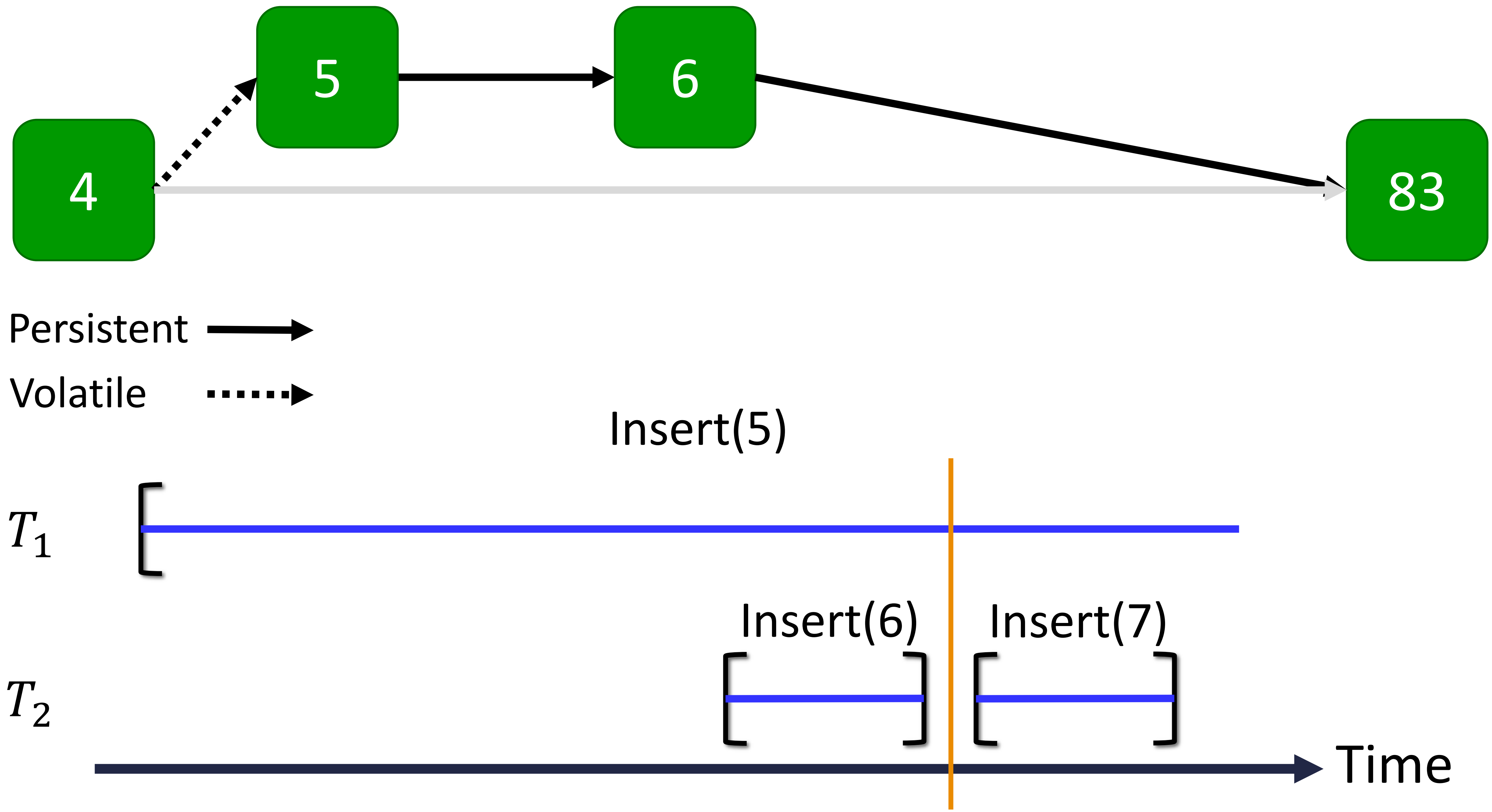


- Explicit flushes are expensive
- Goal: Design data structure which remain consistent after a crash at a minimal cost.

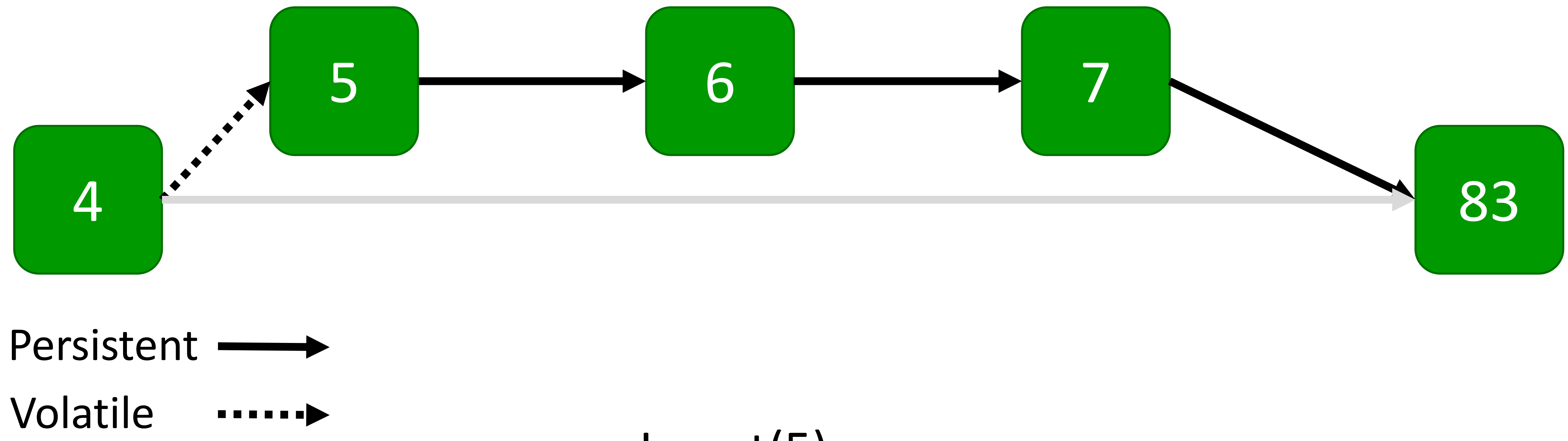
Dependency Issues

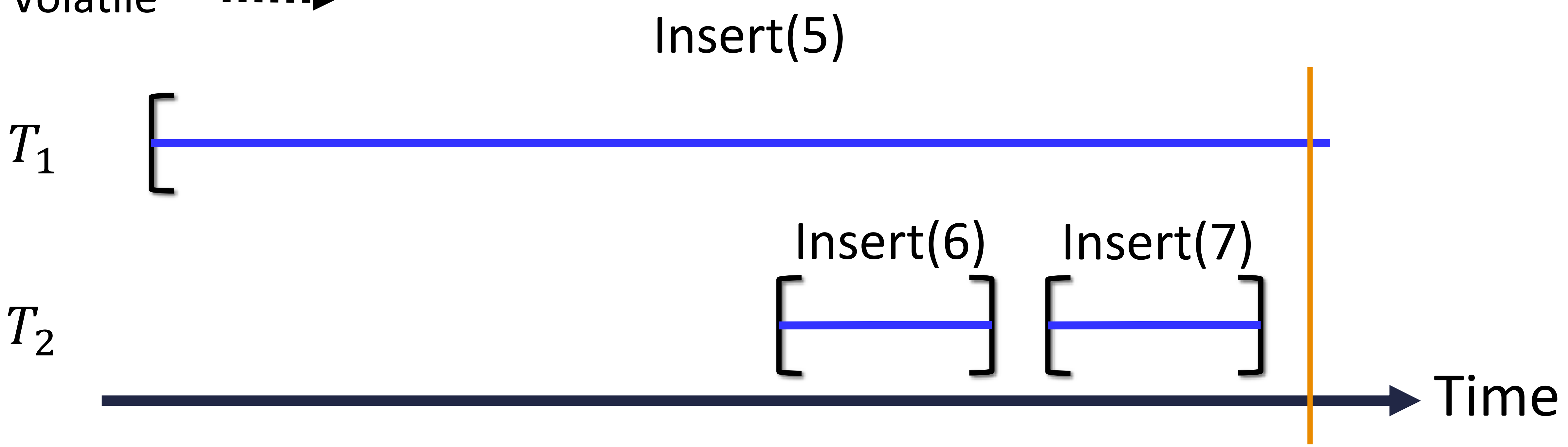


Dependency Issues

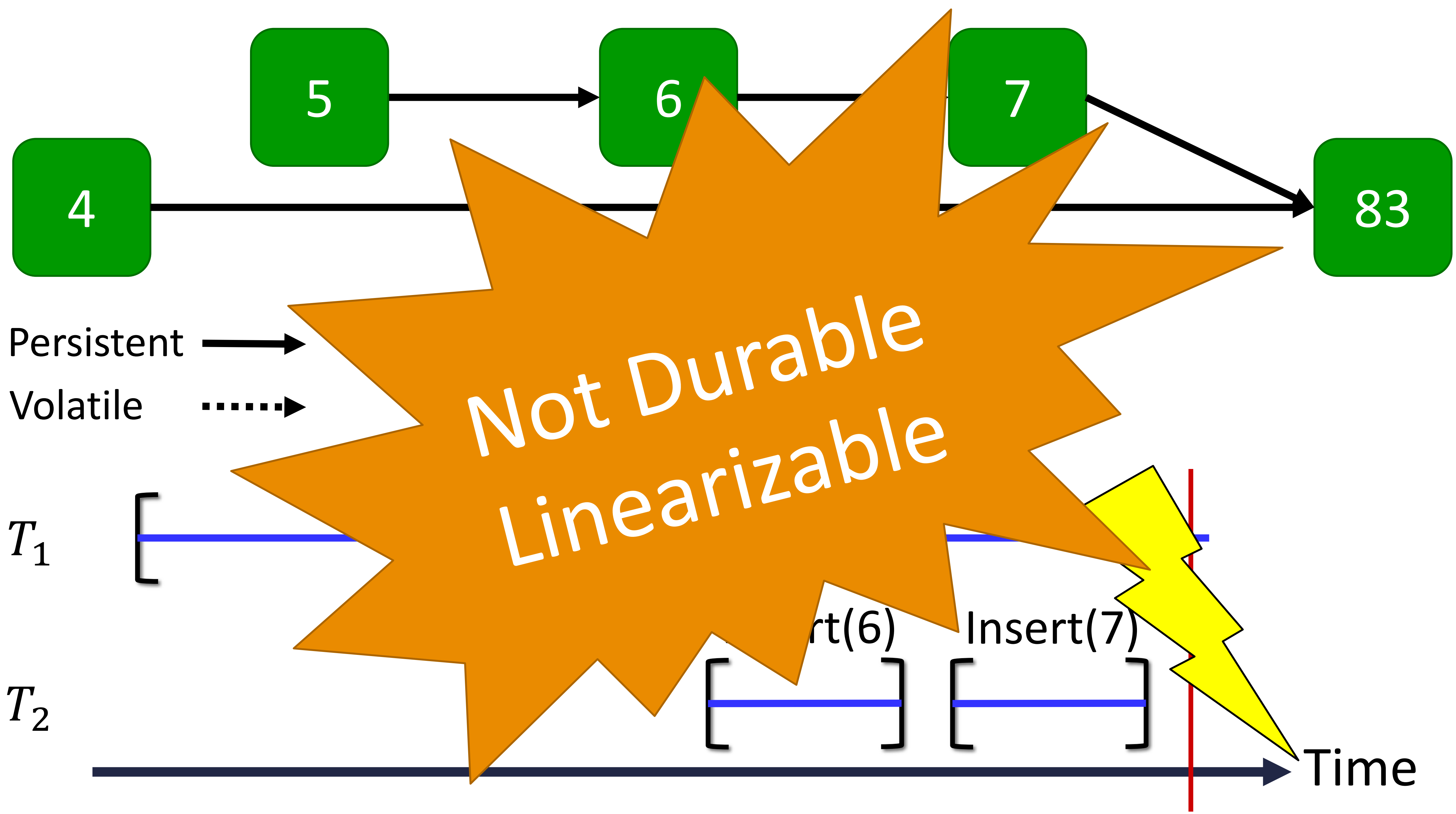


Dependency Issues

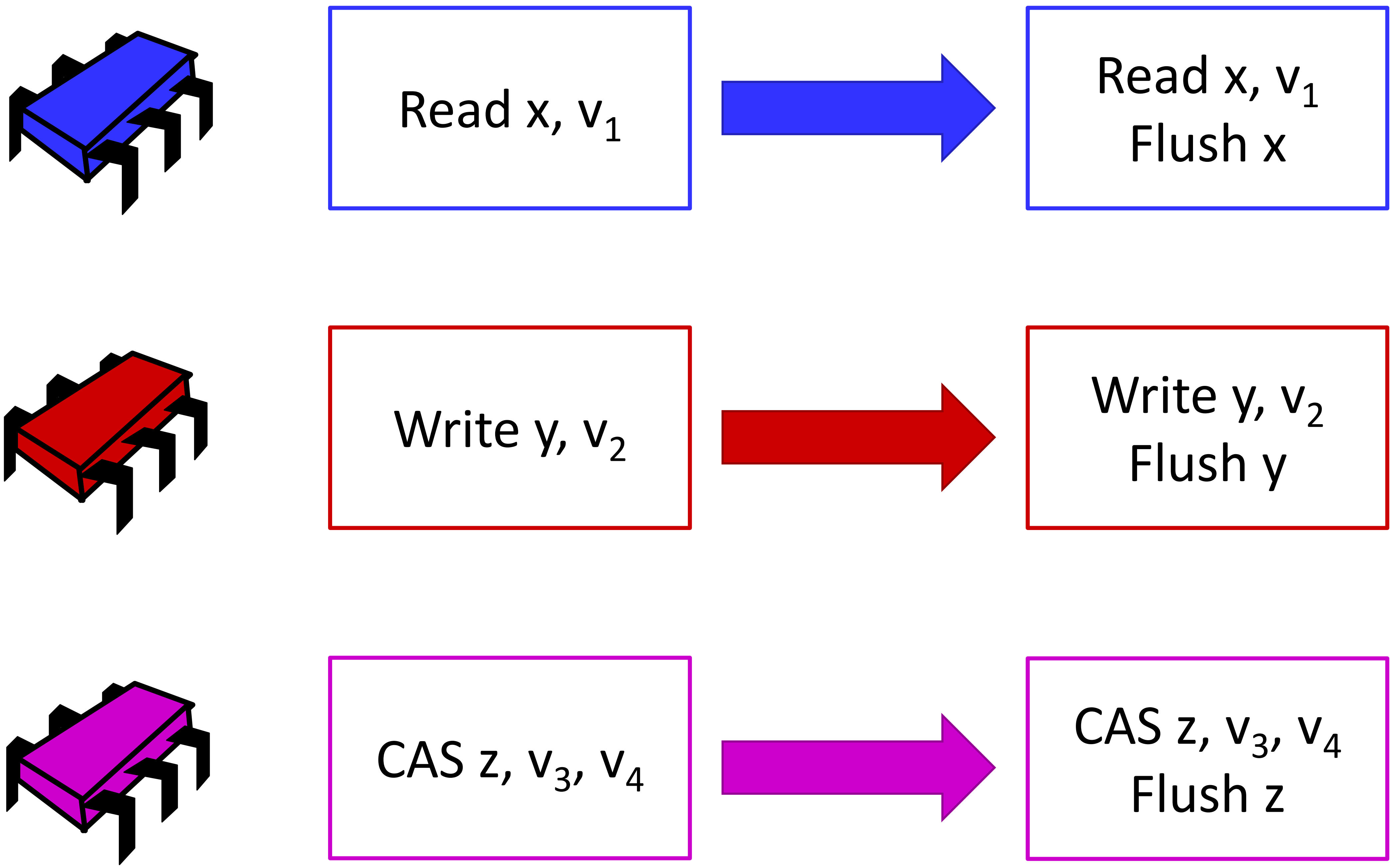




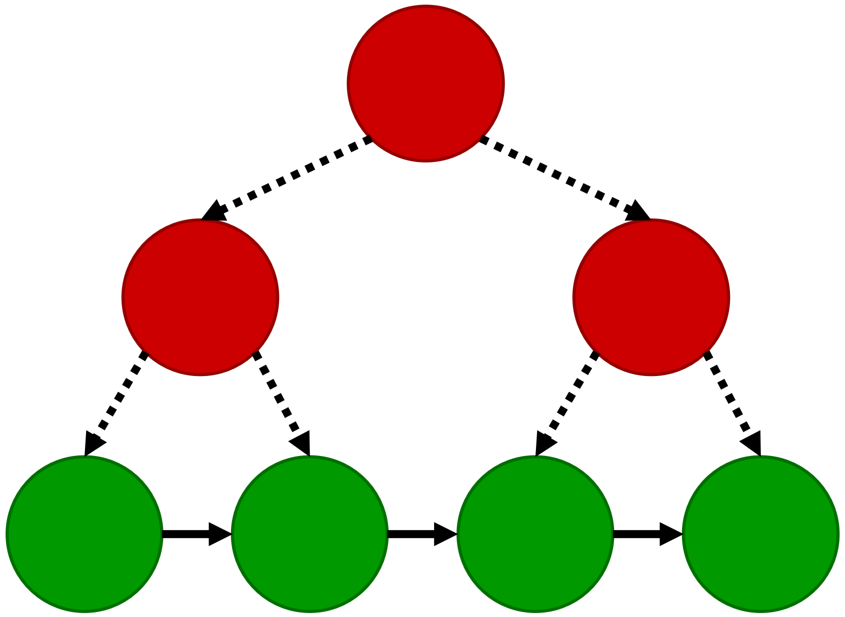
Dependency Issues



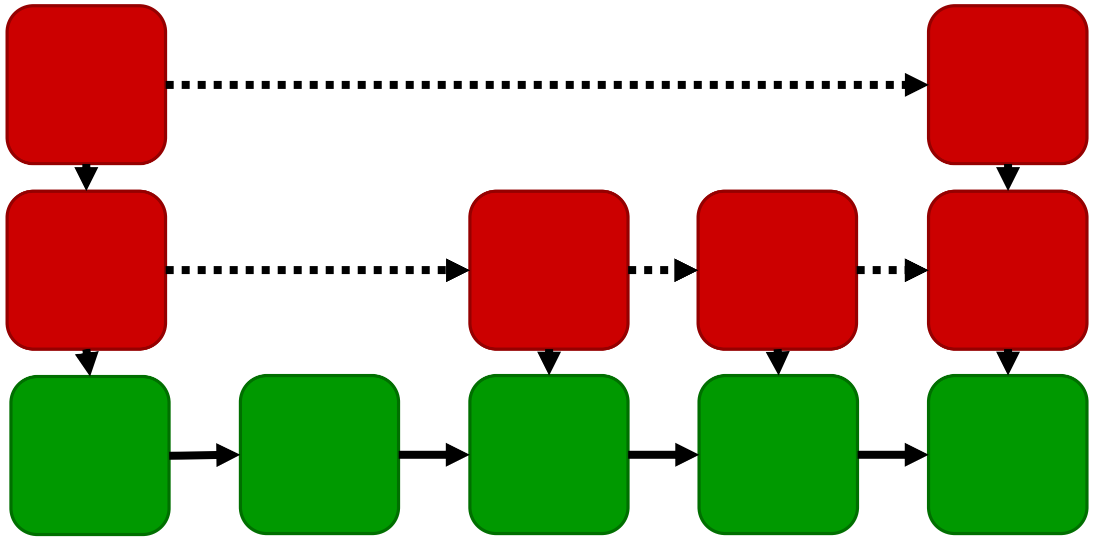
Trivial Solution



Previous Work

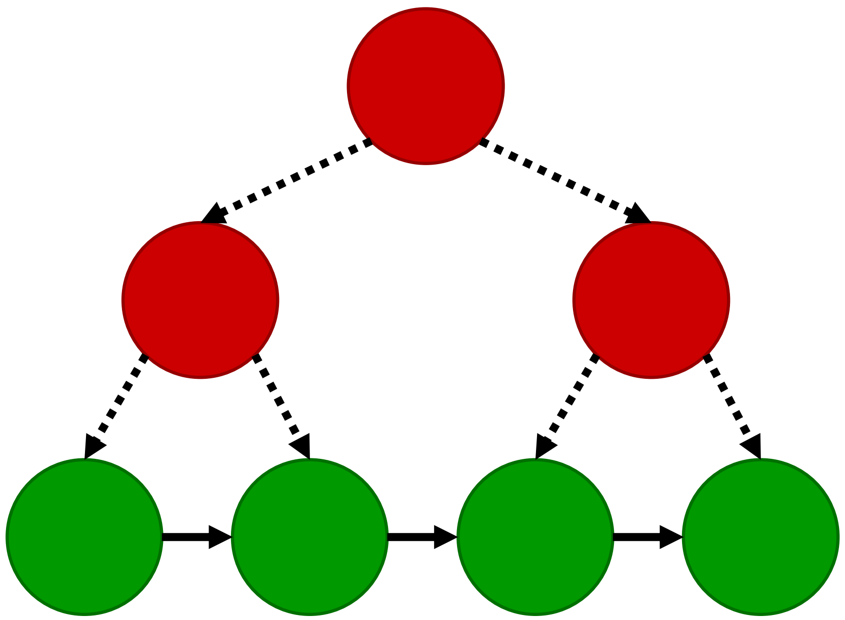


NV-Tree

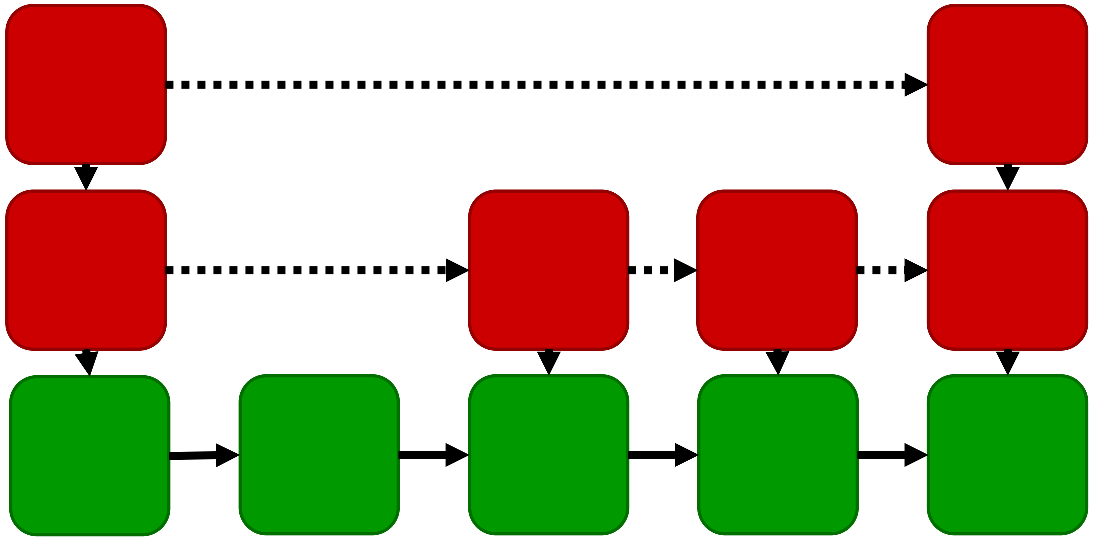


NV-Skiplist

Recovery After a Crash



NV-Tree

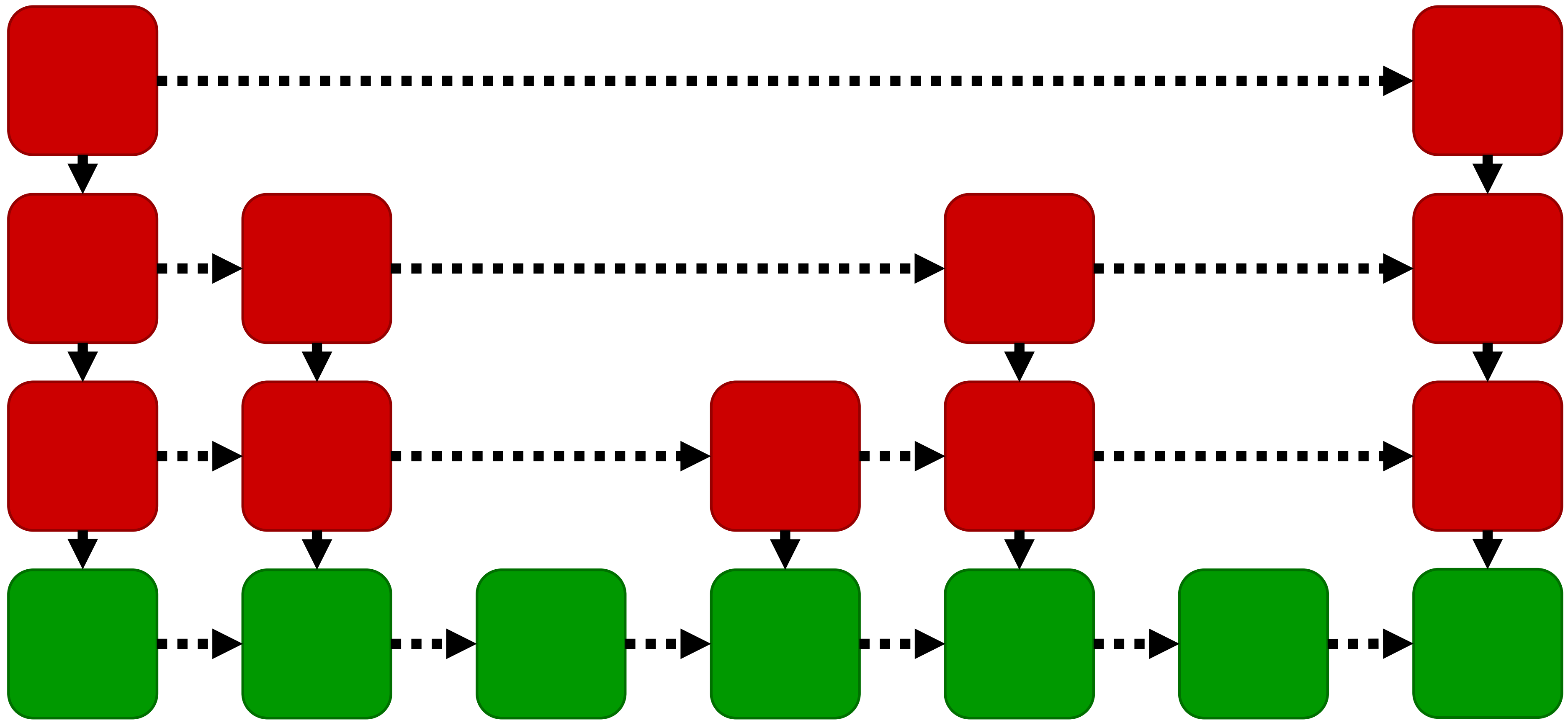


NV-Skiplist

Our Work

No links are persisted

Eliminate inter-key dependencies

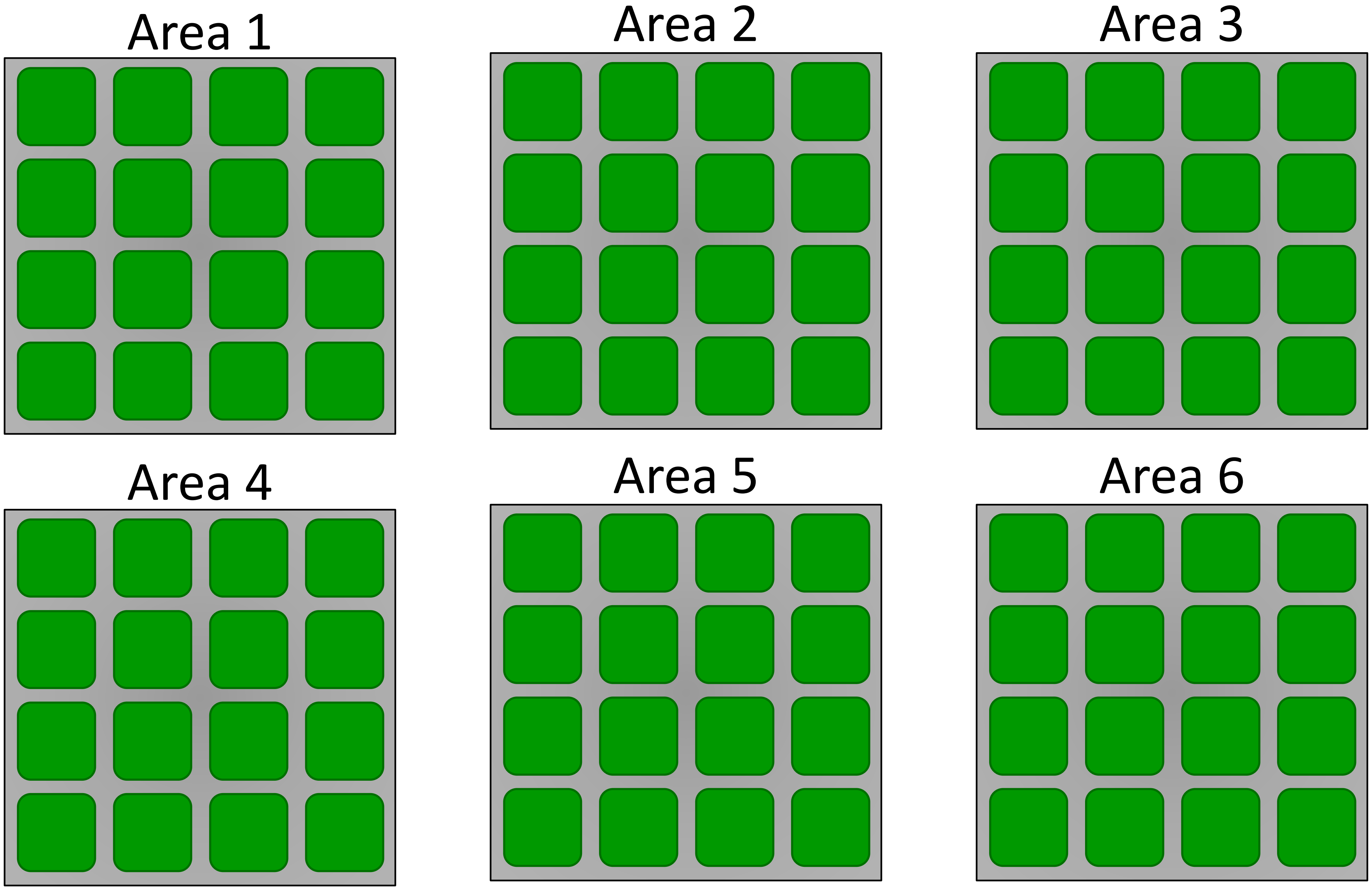


Our Work

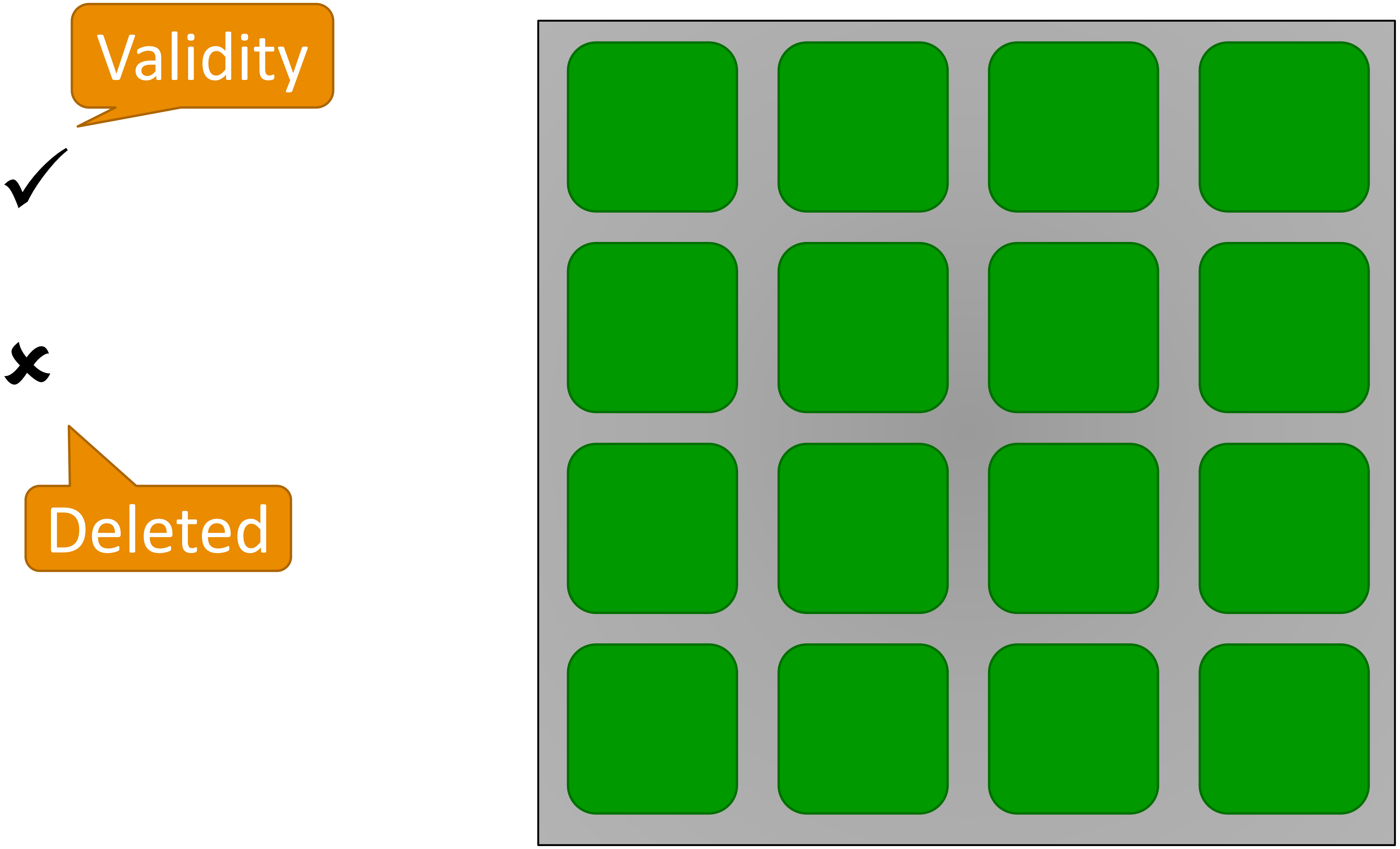
Link-Free: Simpler and natural implementation

SOFT: Complicated but achieves theoretical bound

Durable Areas of Nodes

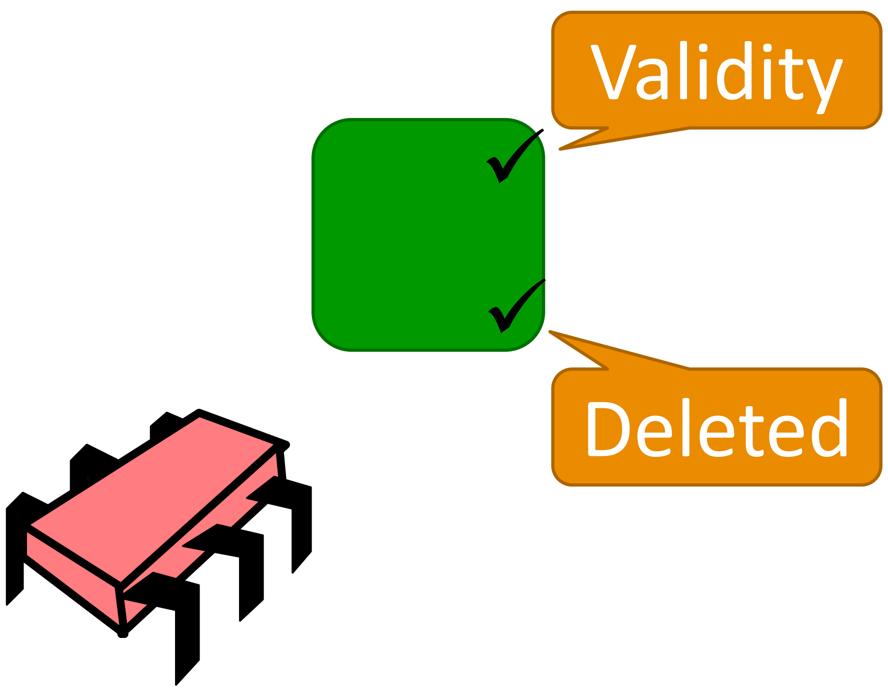


Durable Areas of Nodes



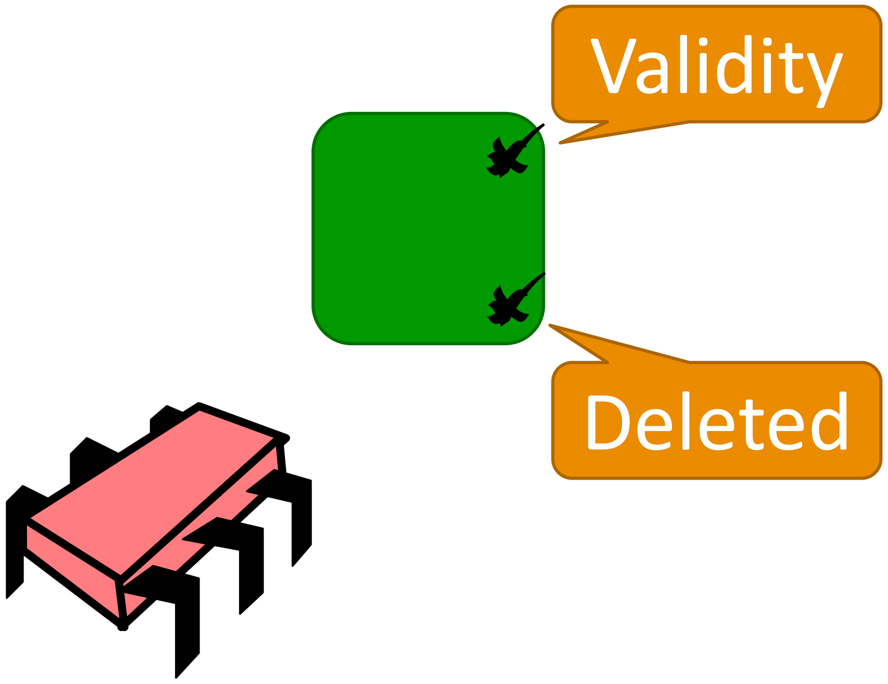
Link-Free: Insert (33)

- Find predecessor and successor
- Allocate from durable area

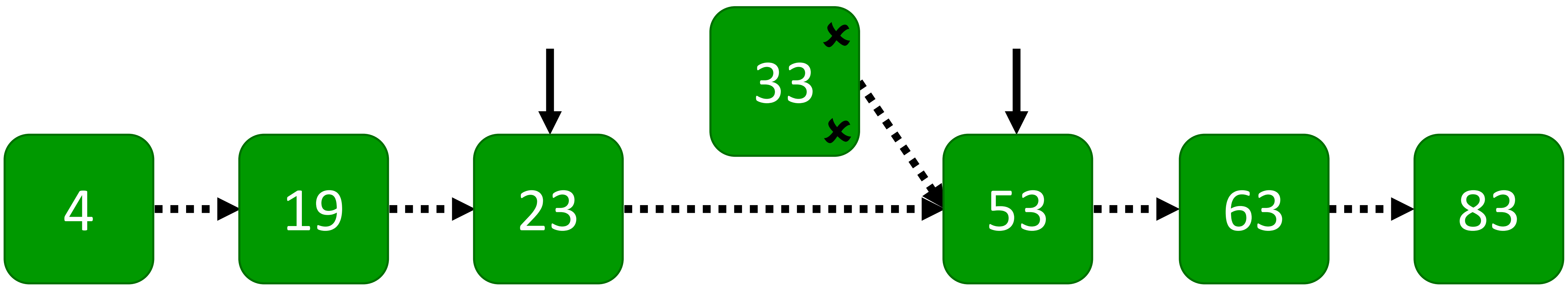


Link-Free: Insert (33)

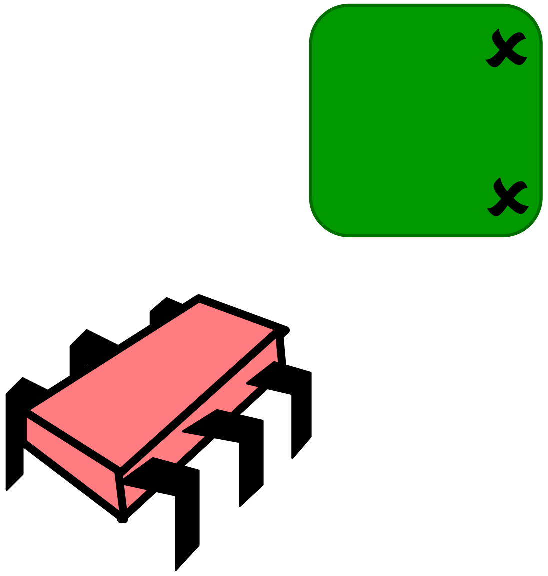
- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted



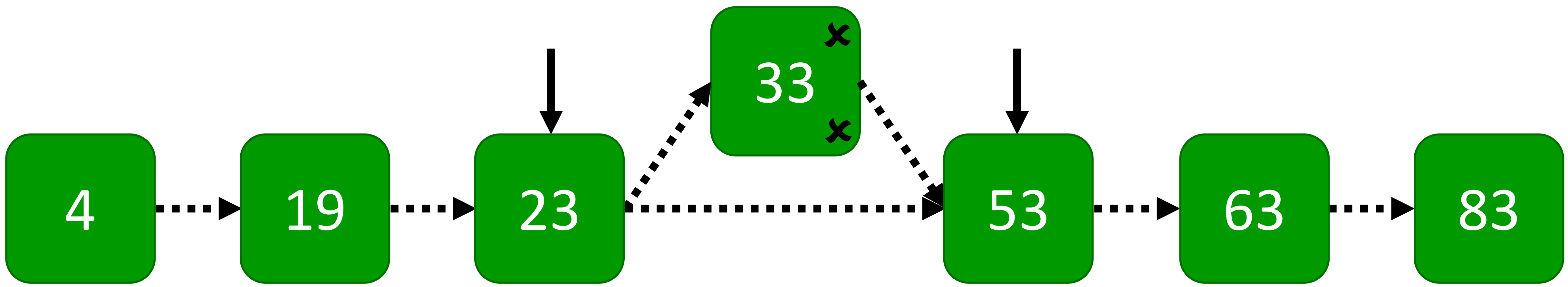
Link-Free: Insert (33)



- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node’s fields

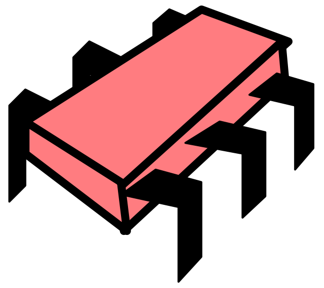


Link-Free: Insert (33)

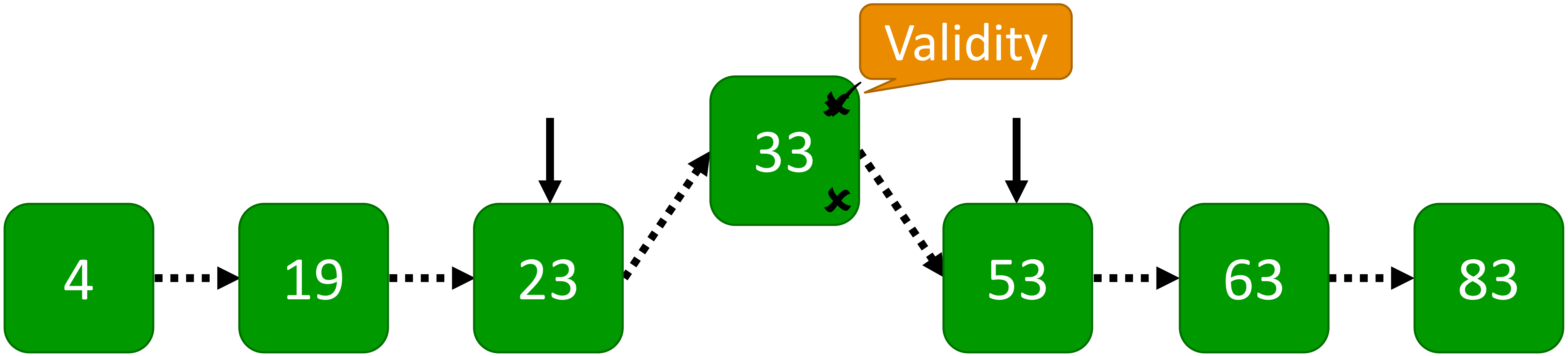


- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node's fields
- CAS node after predecessor

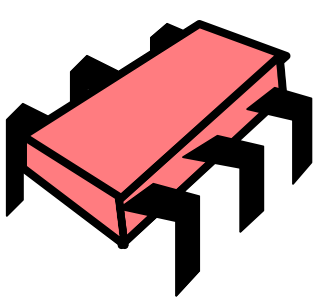
33 not yet in the list



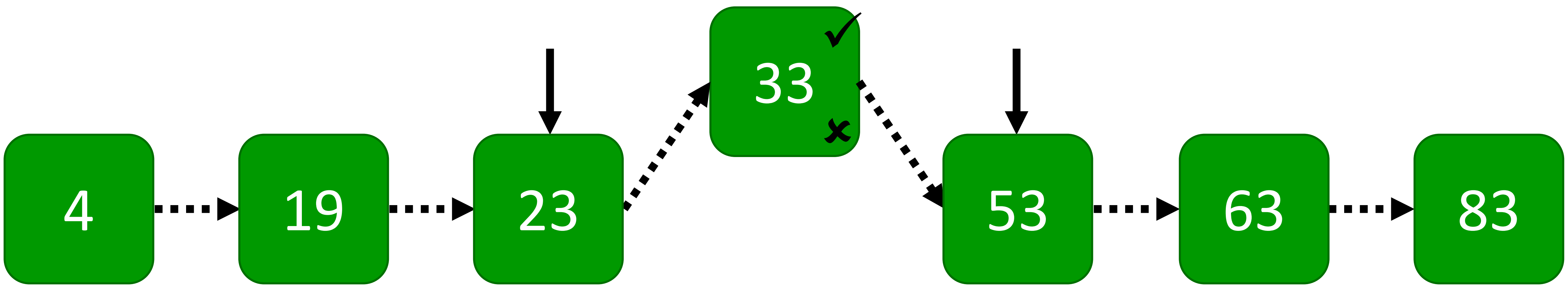
Link-Free: Insert (33)



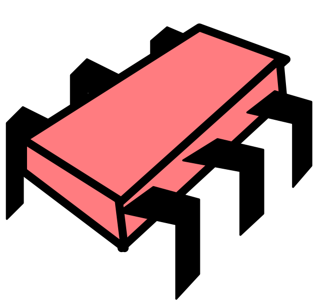
- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node's fields
- CAS node after predecessor
- Make node valid



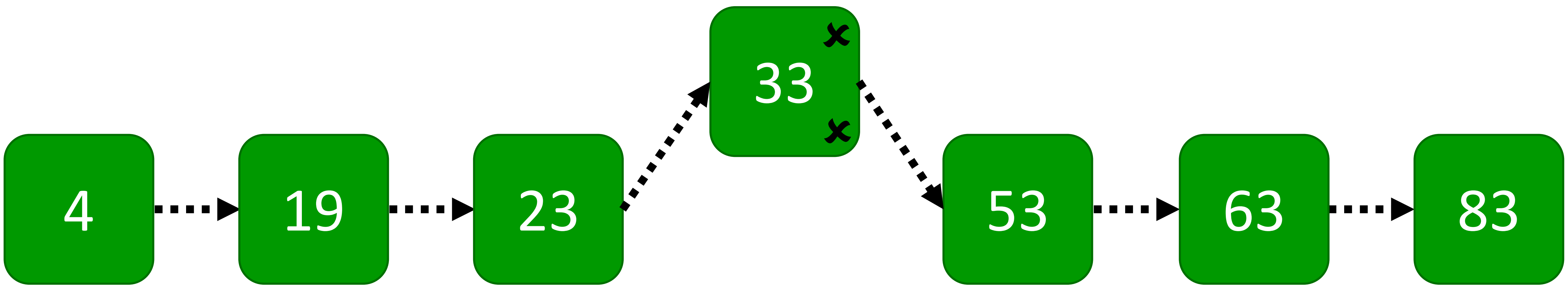
Link-Free: Insert (33)



- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node's fields
- CAS node after predecessor
- Make node valid
- Flush node to NVRAM

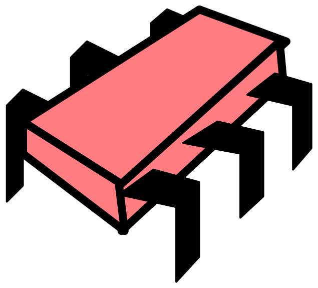


Main Idea

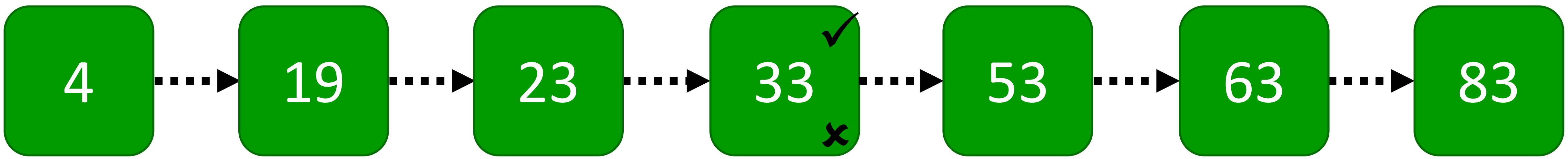


Instability: all subsequent operations on 33 help

- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node’s fields
- CAS node after predecessor
- Make node valid
- Flush node to NVRAM

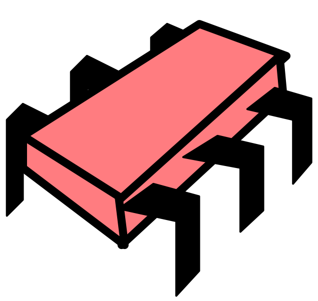


Main Idea



Ensuring 33 survives no pointer flushing

- Find predecessor and successor
- Allocate from durable area
- Make node invalid and not deleted
- Initialize node’s fields
- CAS node after predecessor
- Make node valid
- Flush node to NVRAM



Our Work

Link-Free: Simpler and natural implementation

SOFT: Complicated but achieves theoretical bound

Theoretical Bounds*

Number of flushes per update

Lower Bound:
At least one flush

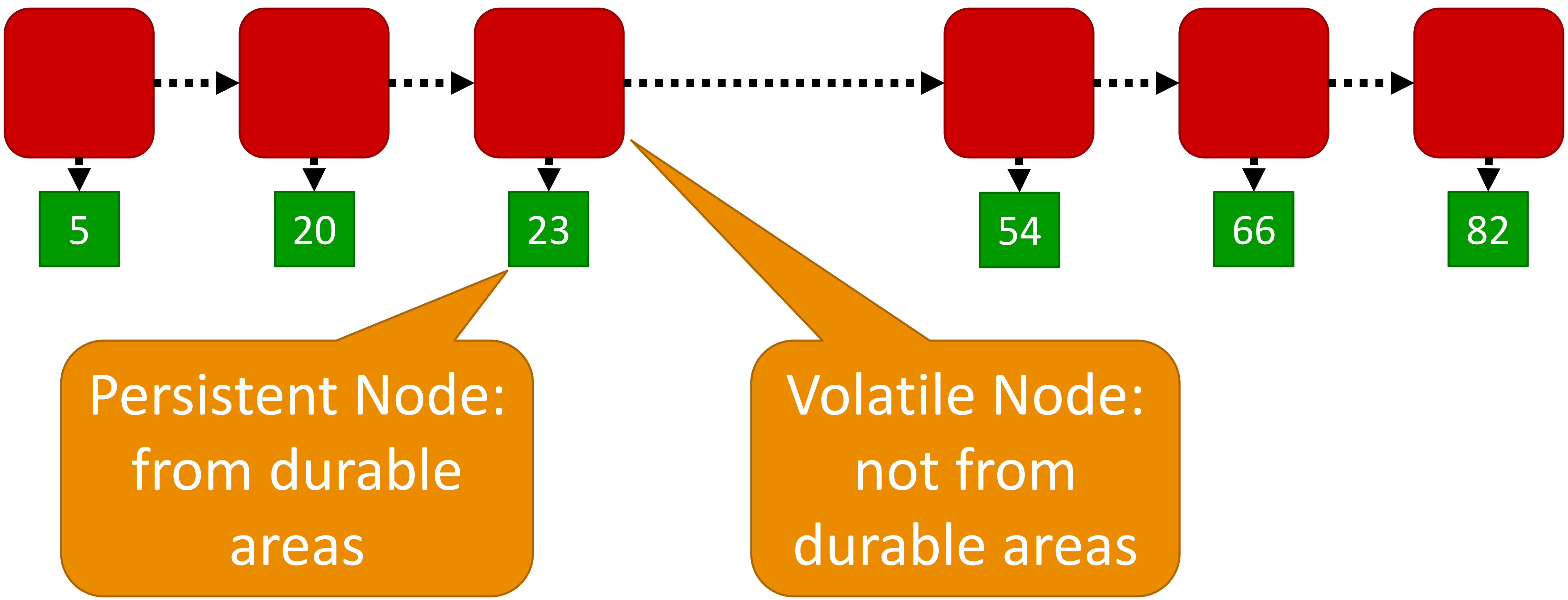
Number of flushes per read-only operation

Lower Bound:
Zero flushes!

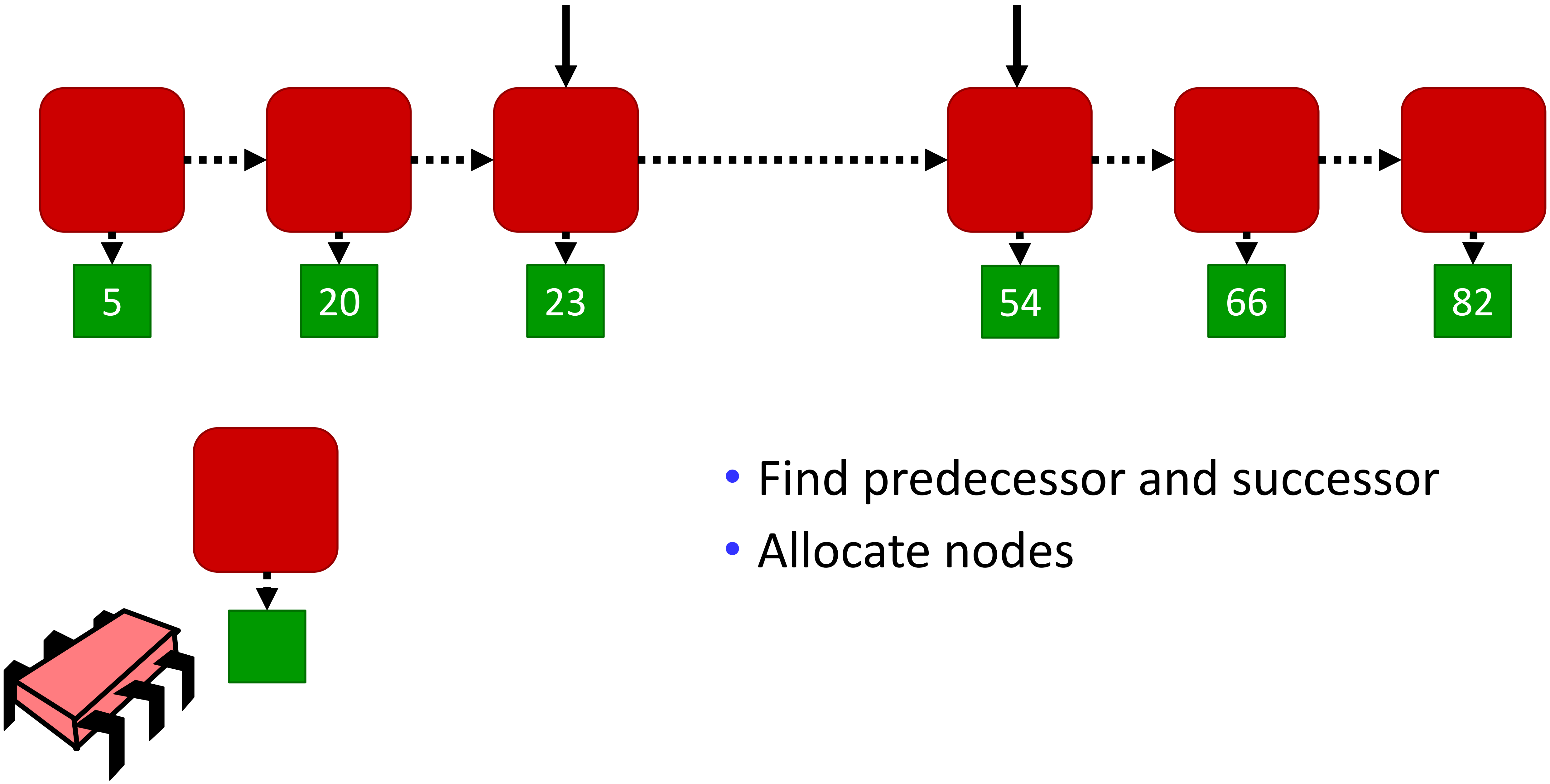
How Does it Work?

- Link-Free: Need to flush what is read
  - To depend on an op, make it survive
  - Cannot meet lower bound
- SOFT: Persist before linearization
  - No need to flush what is read

SOFT: Sets with Optimal Flushing Technique

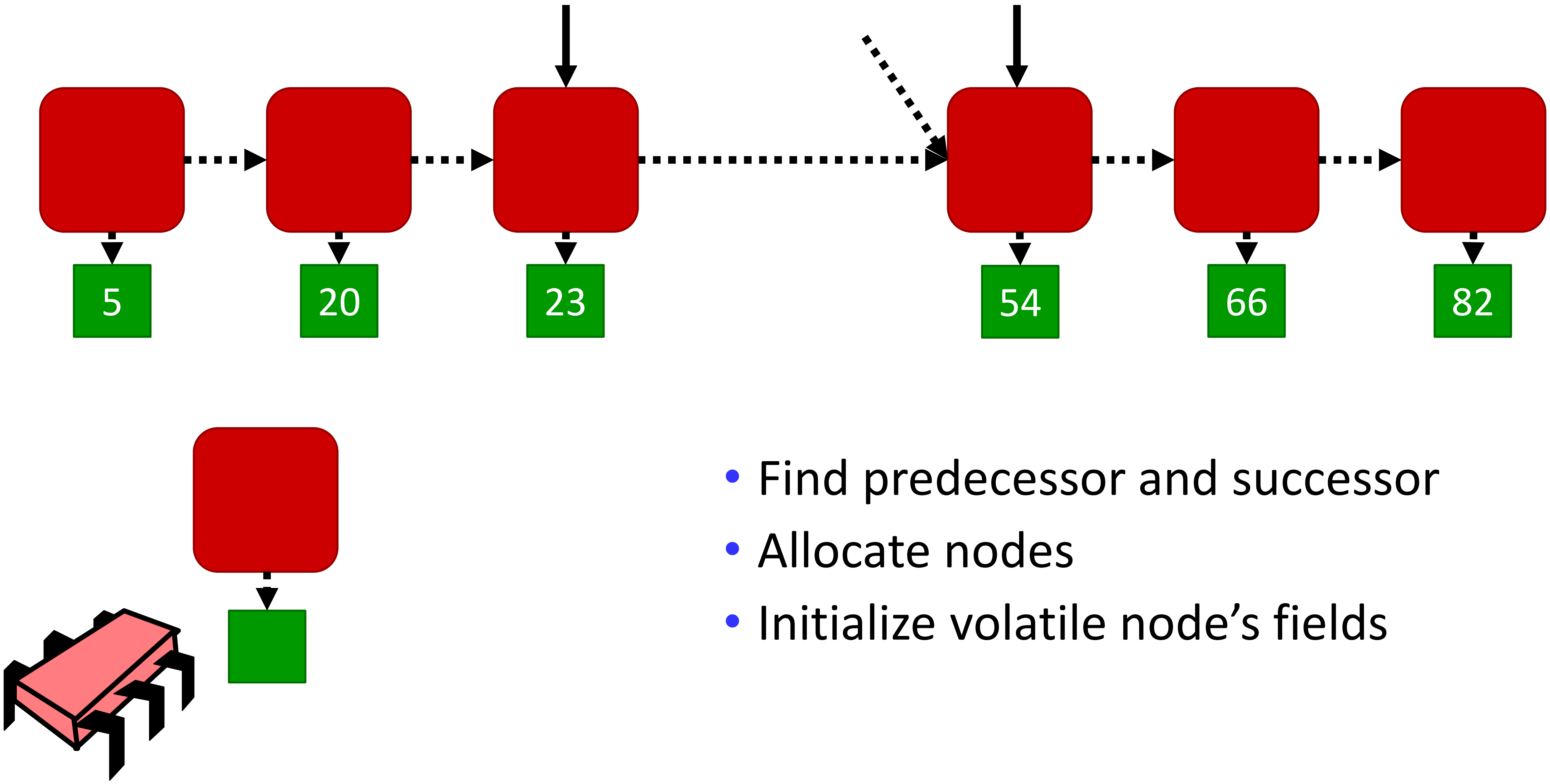


SOFT: Successful Insert of 43



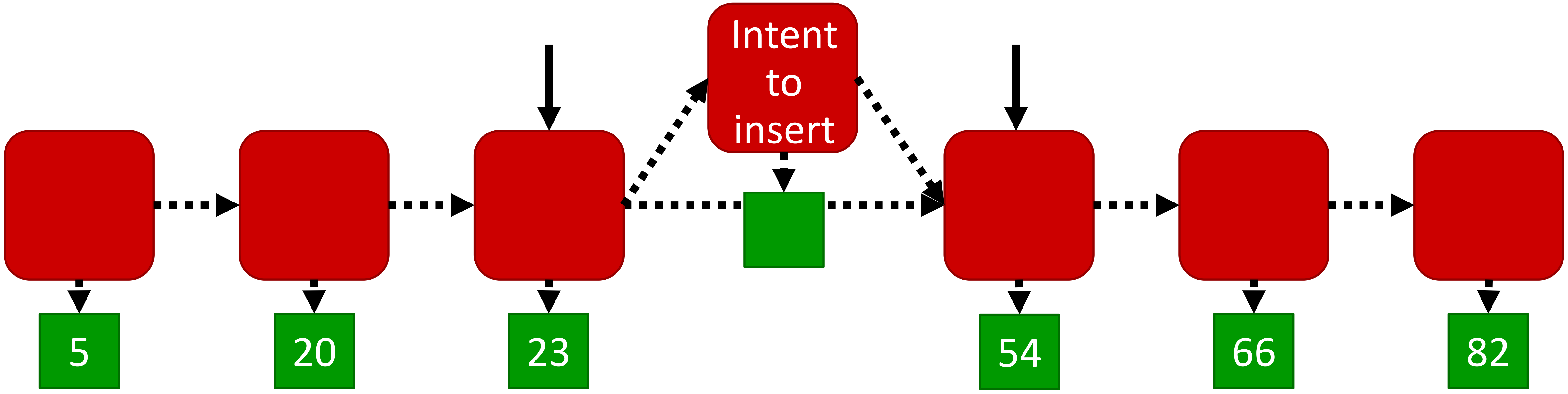
- Find predecessor and successor
- Allocate nodes

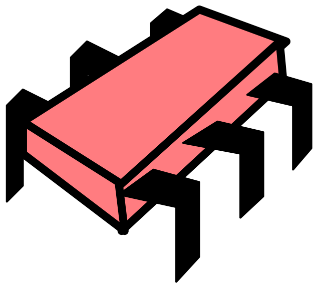
SOFT: Successful Insert of 43



- Find predecessor and successor
- Allocate nodes
- Initialize volatile node's fields

SOFT: Successful Insert of 43

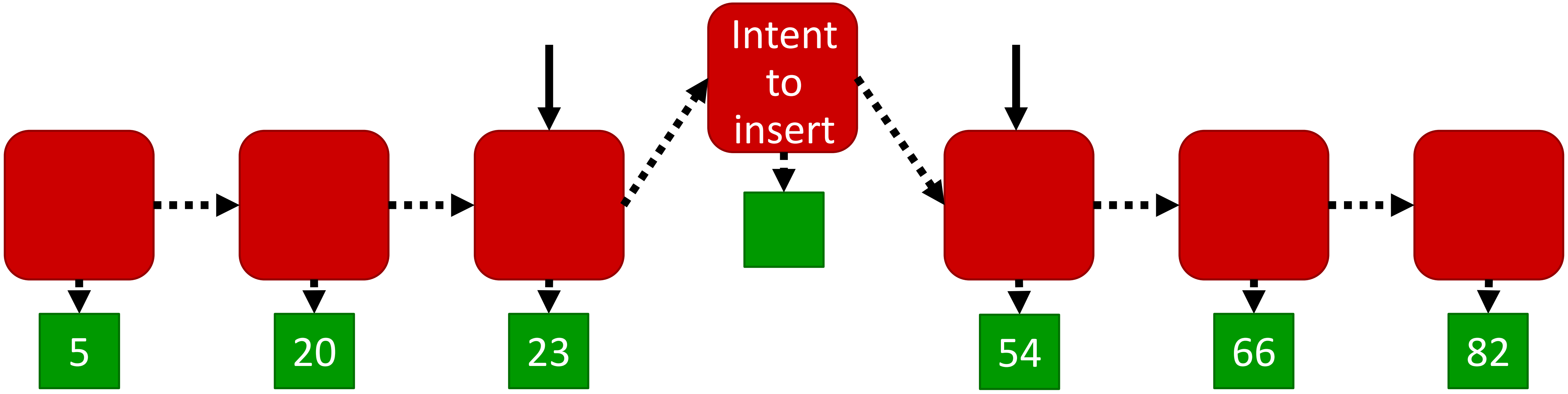




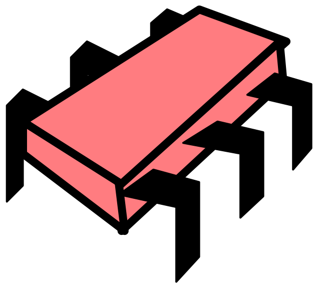
Determine result

- Find predecessor and successor
- Allocate nodes
- Initialize volatile node’s fields
- CAS node after predecessor

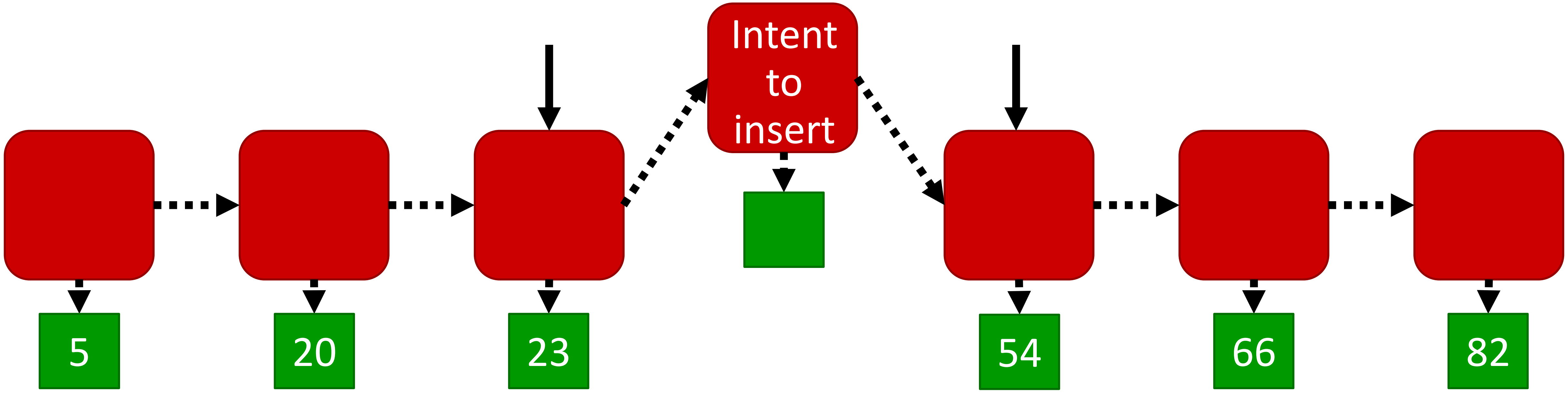
SOFT: Successful Insert of 43



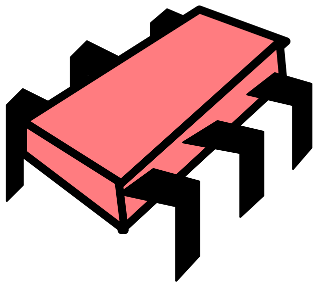
- Find predecessor and successor
- Allocate nodes
- Initialize volatile node's fields
- CAS node after predecessor

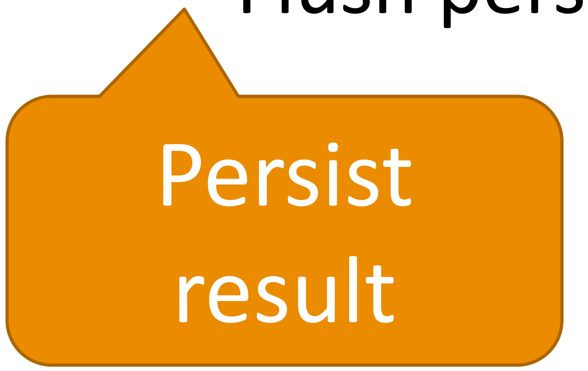


SOFT: Successful Insert of 43

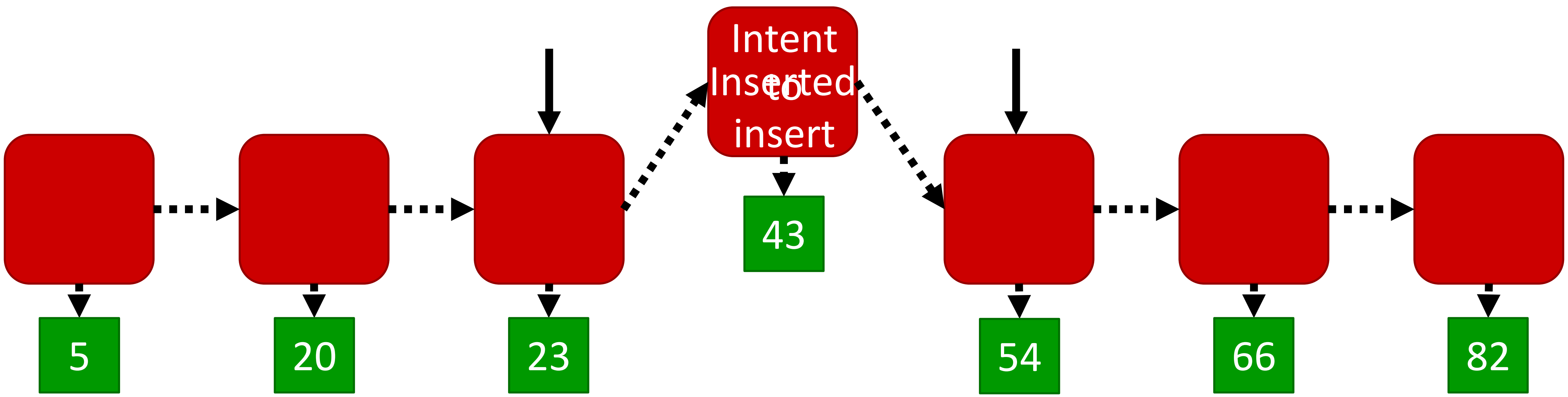


- Find predecessor and successor
- Allocate nodes
- Initialize volatile node's fields
- CAS node after predecessor
- Initialize persistent node
- Flush persistent node

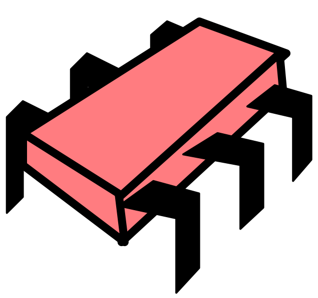




SOFT: Successful Insert of 43

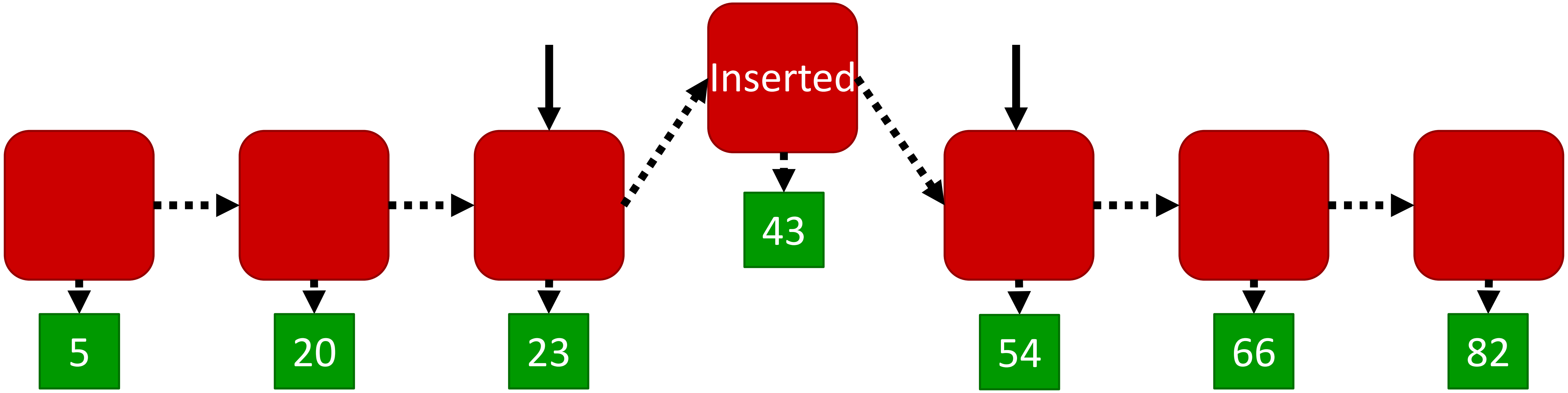


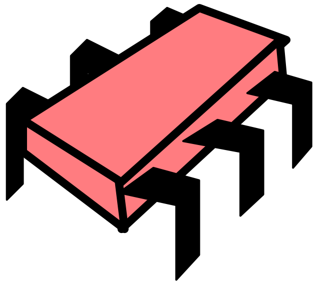
- Find predecessor and successor
- Allocate nodes
- Initialize volatile node's fields
- CAS node after predecessor



- Initialize persistent node
- Flush persistent node
- Change volatile node to "inserted" state

SOFT: Successful Insert of 43





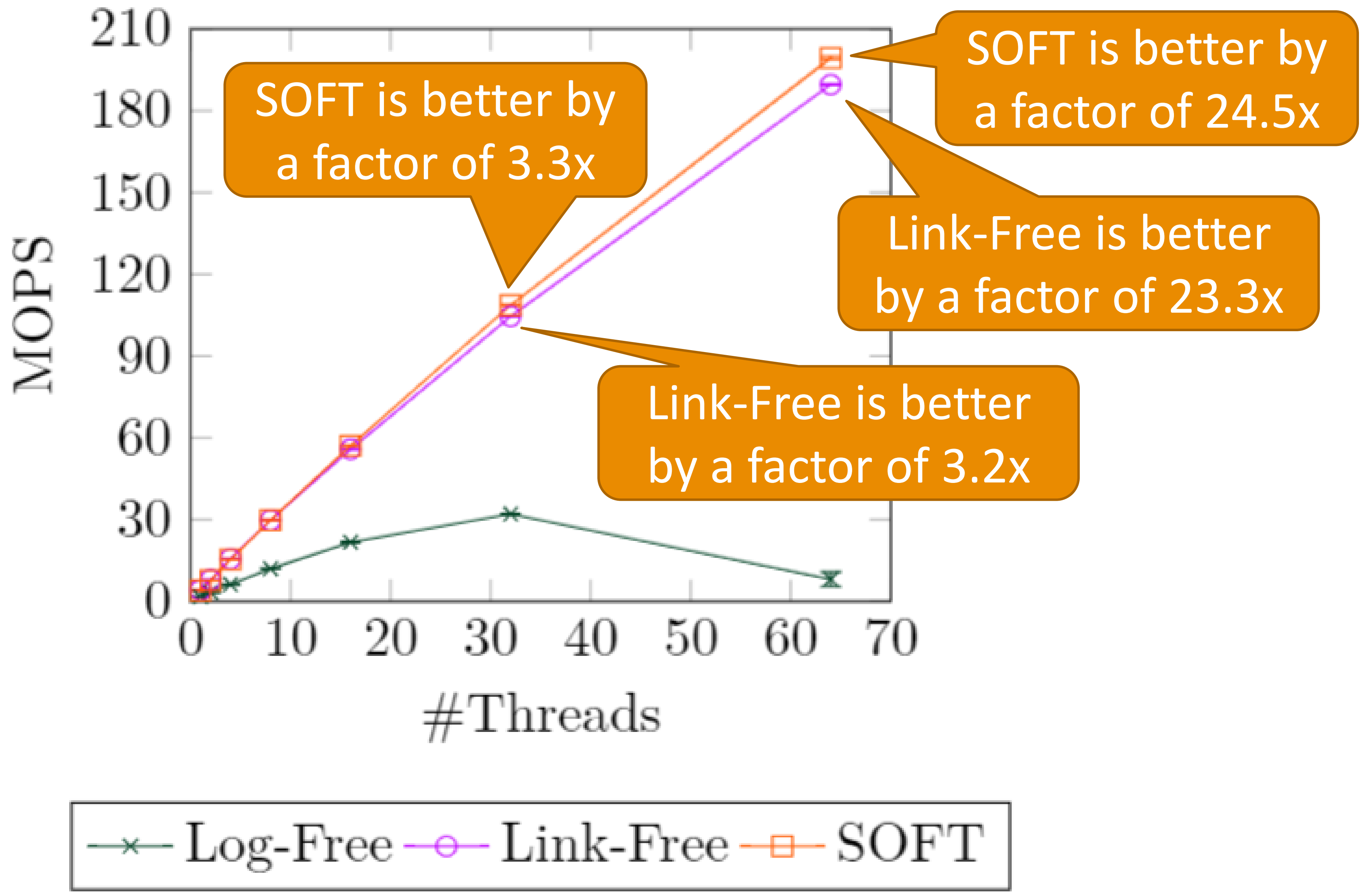
- Find predecessor and successor
- Allocate nodes
- Initialize volatile node's fields
- CAS node after predecessor
- Initialize persistent node
- Flush persistent node
- Change volatile node to "inserted" state

Linearize

Evaluation

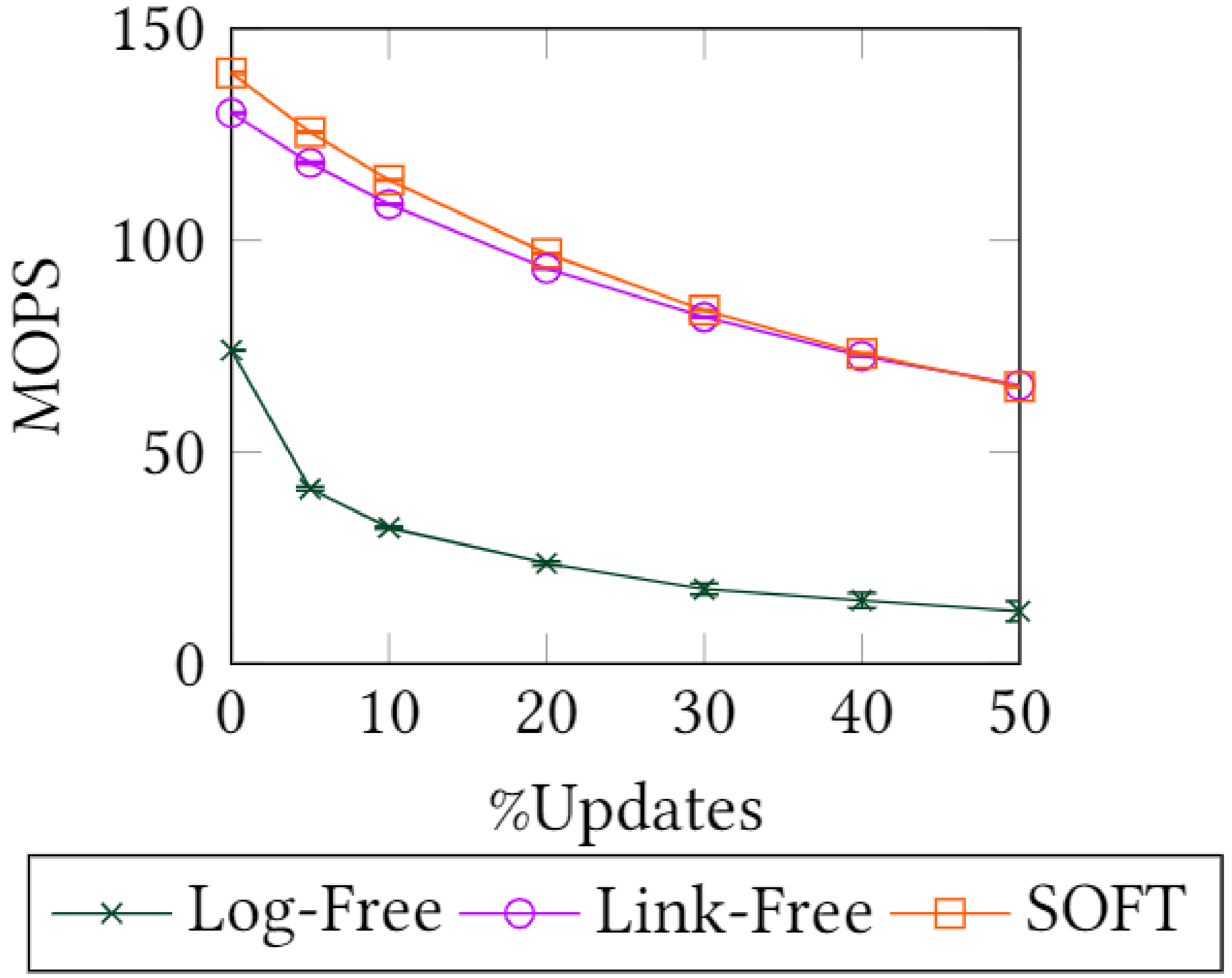
- Log-Free data structures [David et al. 2018] are state-of-the-art durable sets:
  - Linked-list, hash table, skip-list and binary search tree
- Compared Link-Free, SOFT and Log-Free Hash tables.
- Platform:
  - 64 cores (4 AMD Opteron(TM) 6376 2.3GHz processors)
  - Ubuntu 16.04.6 LTS (kernel version 4.4.0)
  - g++ version 8.3.0

Hash Table Scalability

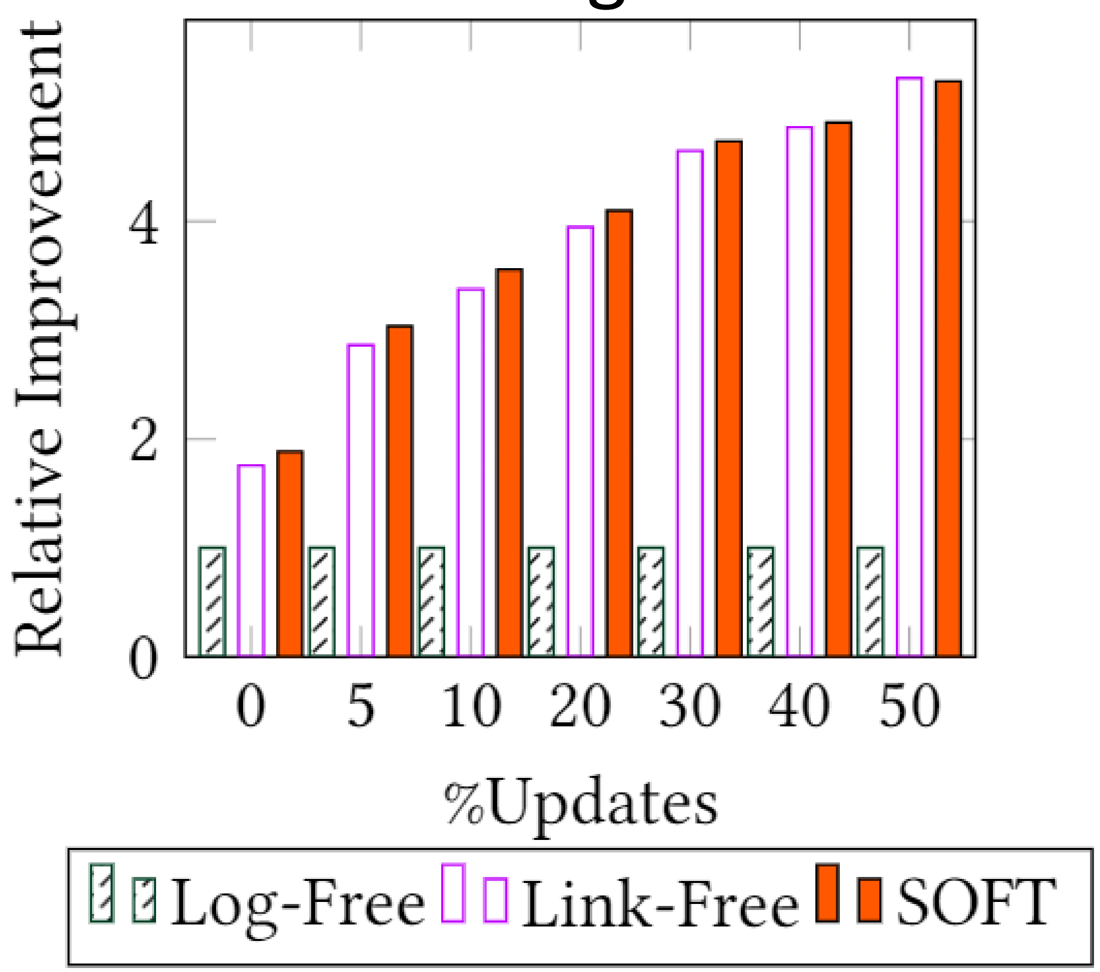


With 500K keys

Hash Table Throughput Varying Update Frequency



Relative Improvement Over Log-Free



With 500K keys

Related Work

- Linearizability [Herlihy and Wing 1990]
- Lock-Free Linked-List [Harris 2001]
- NV-Tree [Yang et al. 2015]
- Durable Linearizability [Izraelevitz et al. 2016]
- Efficient Durable Logging [Cohen et al. 2017]
- Theoretical Bound [Cohen et al. 2018]
- NV-Skiplist [Chen and Yeom 2018]
- Log-Free Data Structures [David et al. 2018]
- …

Conclusion

- Best performing durable sets for non-volatile memory
  - Two algorithms with similar, good performance
  - 3.3x over state-of-the-art with 32 threads
  - SOFT achieves theoretical bound
  - Link-Free good for long lists
- Main idea: do not persist links, only nodes
- Implementation:
  - Keep designated node areas,
  - Node state signifies membership,
  - Careful interplay between linearizability and durability
- Code is public and Proof is available