Space and Time Bounded Multiversion Garbage Collection

Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert,
Yihan Sun, and Yuanhao Wei

Introduction

- Multiversioning widely used:
  - Database systems
  - Software Transactional Memory [Fernandes et al. PPoPP'11] [Lu et al. DISC'13]
  - Concurrent data structures [Fatourou et al. SPAA'19] [Wei et al. PPoPP'21]

- High space usage ⇒ obsolete versions must be reclaimed
  - Multiversion garbage collection problem (MVGC)
  - Observed to be a bottleneck in modern database systems [Lee et al. SIGMOD'16] [Böttcher et al. VLDB'19]

Research Question

How do you garbage collect efficiently for multiversioning?

Main results

A **general** MVGC scheme with:

- **Progress: wait-free**
- **Time**: O(1) per reclaimed version, on average
- **Space**: **constant factor** more versions than needed, plus an additive term

Previous solutions either use:

- **unbounded space** **[Wei et al. PPoPP'21] [Fernandes et al. PPoPP'11]** , or
- **O(P)** time per reclaimed version **[Lu et al. DISC'13] [Böttcher et al. VLDB'19]**
  - **P**: number of processes

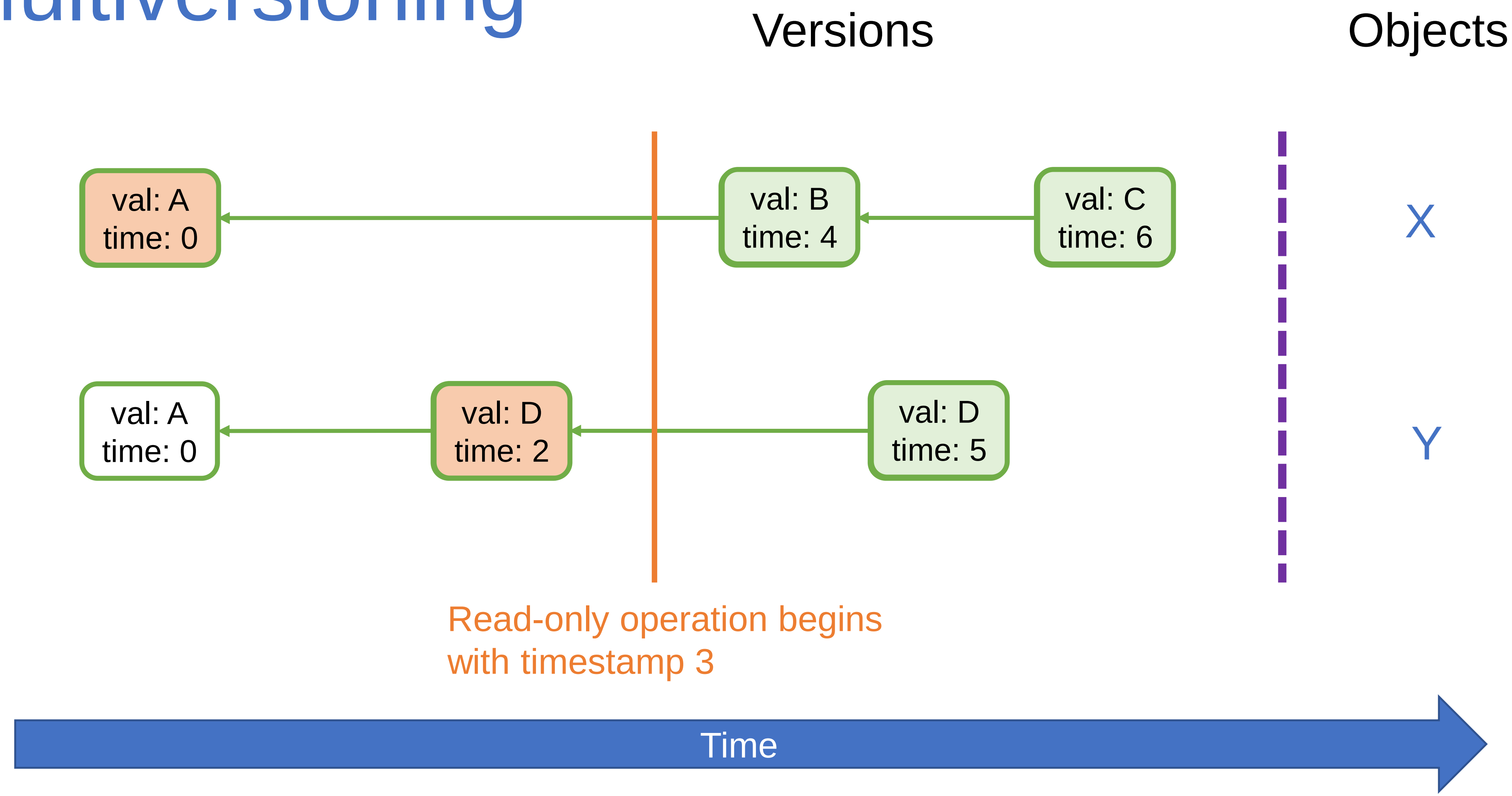
Main results

A **general** MVGC scheme with:

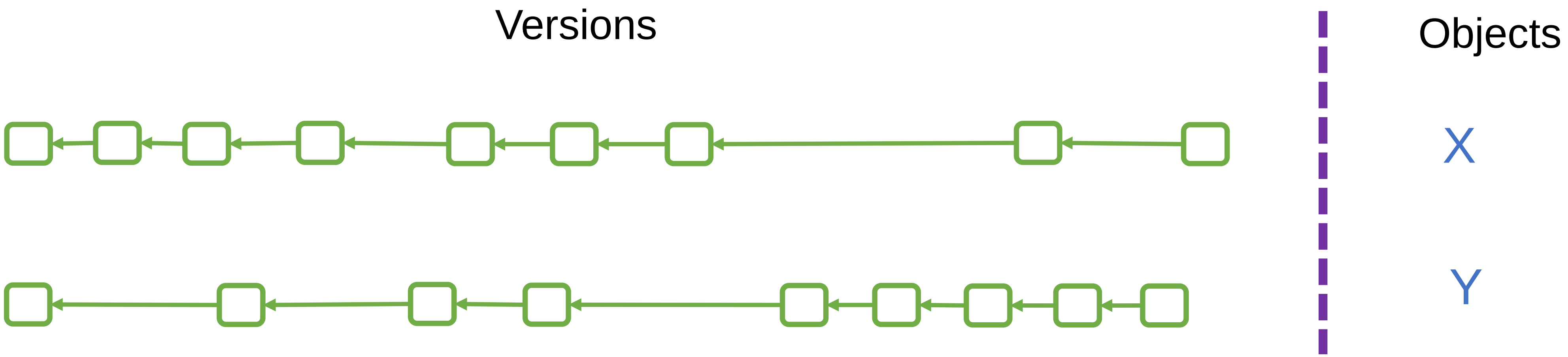
- **Progress: wait-free**
- **Time**: O(1) per reclaimed version, on average
- **Space**: **constant factor** more versions than needed, plus an additive term

- Components of independent interest:
  - Range tracking data structure [for identifying obsolete versions]
  - Concurrent doubly-linked-list [for removing obsolete versions]

Multiversioning



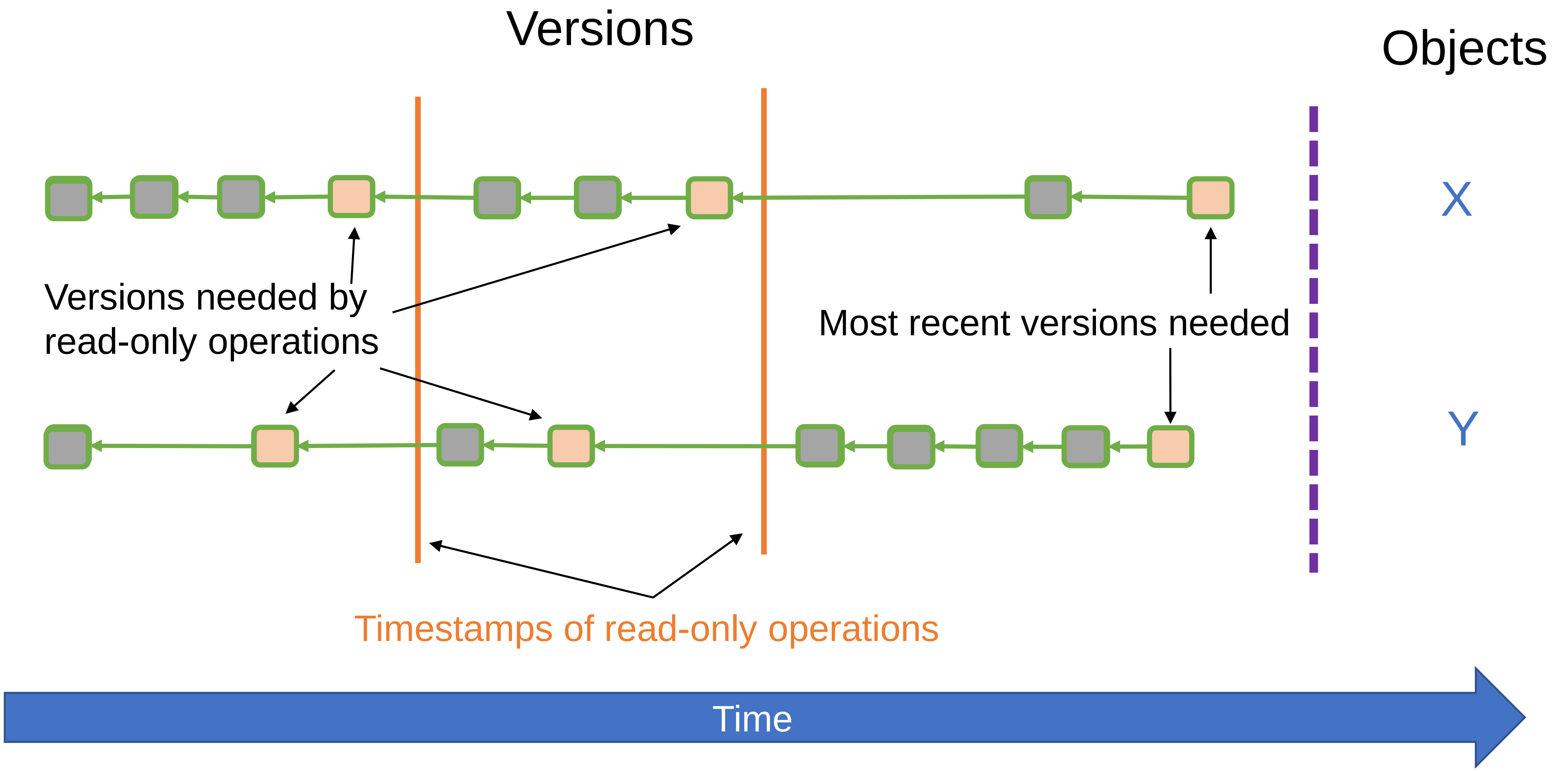
Multiversion Garbage Collection (MVGC)



Maintaining all old versions ⇒ high memory usage

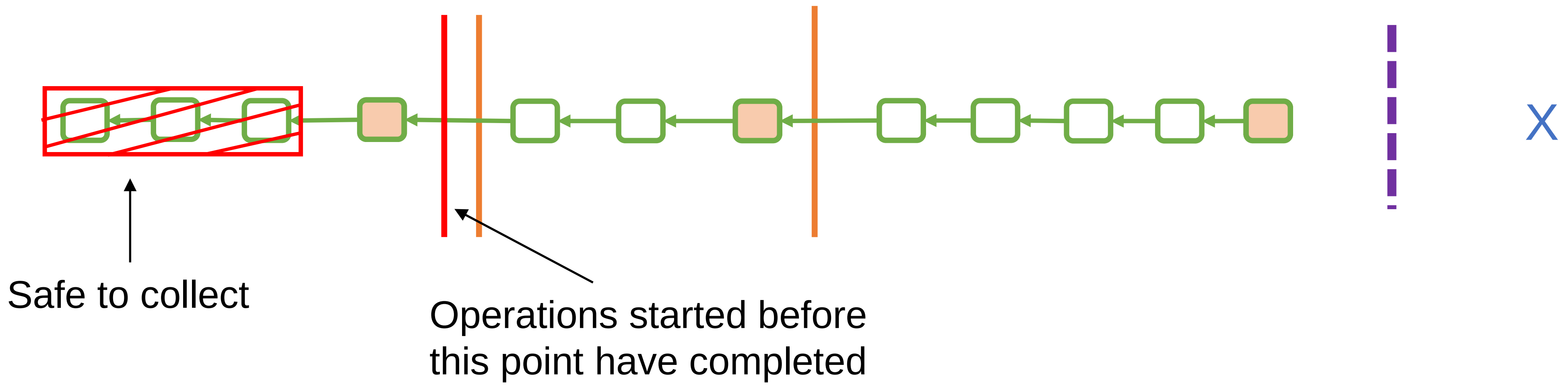
- How do we know which versions obsolete?
- How do we safely reclaim them?

Which Versions are Needed?



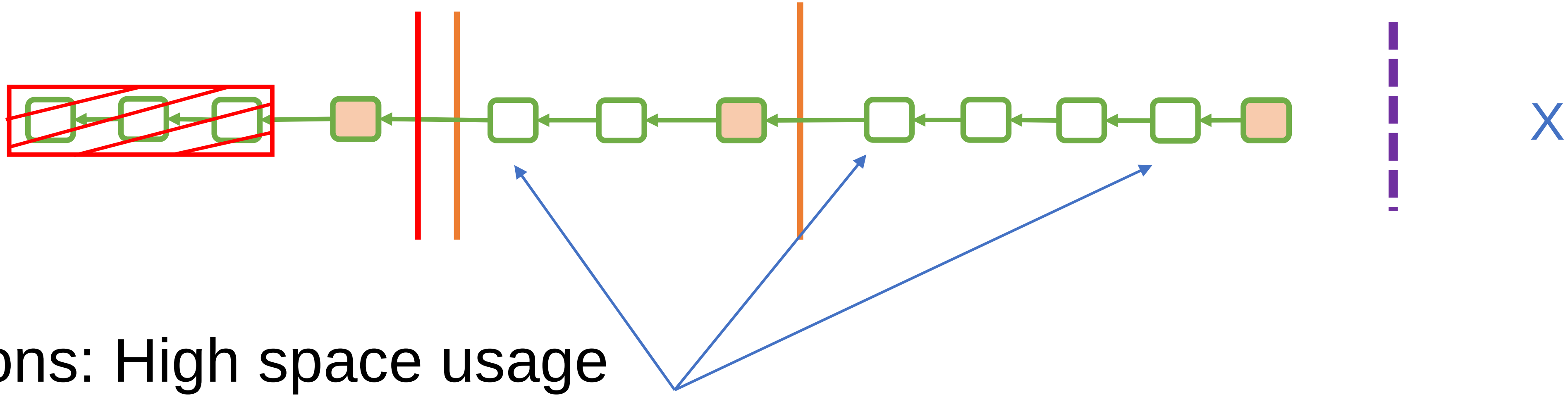
Related Work – Epoch-Based Solutions

- Reclaim versions overwritten before the start of the oldest read-only operation
- Most commonly used



- Pros: Fast, easy to implement

Related Work – Epoch-Based Solutions



- Cons: High space usage
  - Unable to collect newer obsolete versions
  - Particularly bad with long read-only operations
    - E.g. database scans, large range queries
  - Paused process can lead to unbounded space usage

Related Work – Other Solutions

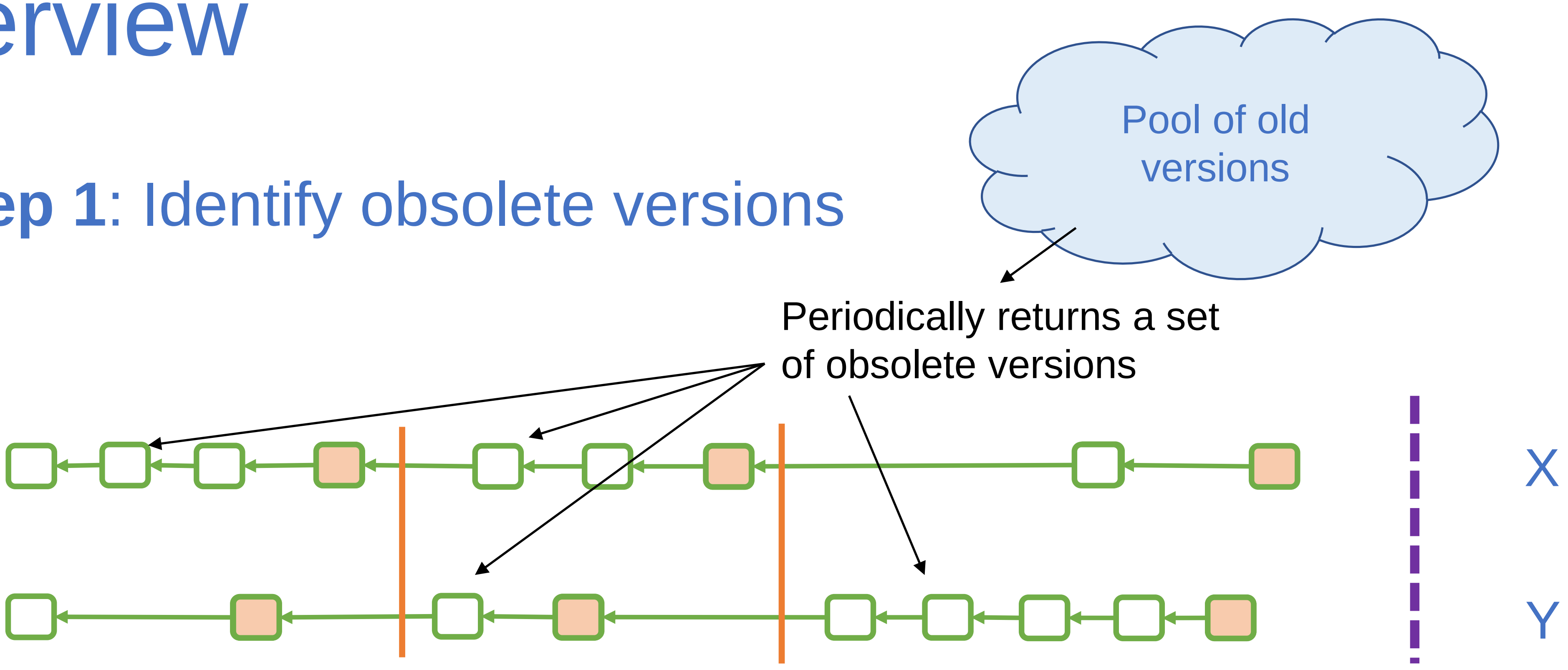
- Techniques have been developed to address shortcomings of epoch-based solutions
  - GMV **[Lu et al. DISC’13]**, Hana **[Lee et al. SIGMOD’16]**, Steam **[Böttcher et al. VLDB’19]**
  - Require Ω(P) time, on average, to collect each version in worst case executions,
  - Keep up to P times more versions than necessary
    - P: number of processes

Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list
Step 3: Reclaim memory of unlinked versions

Overview

Range tracking data structure



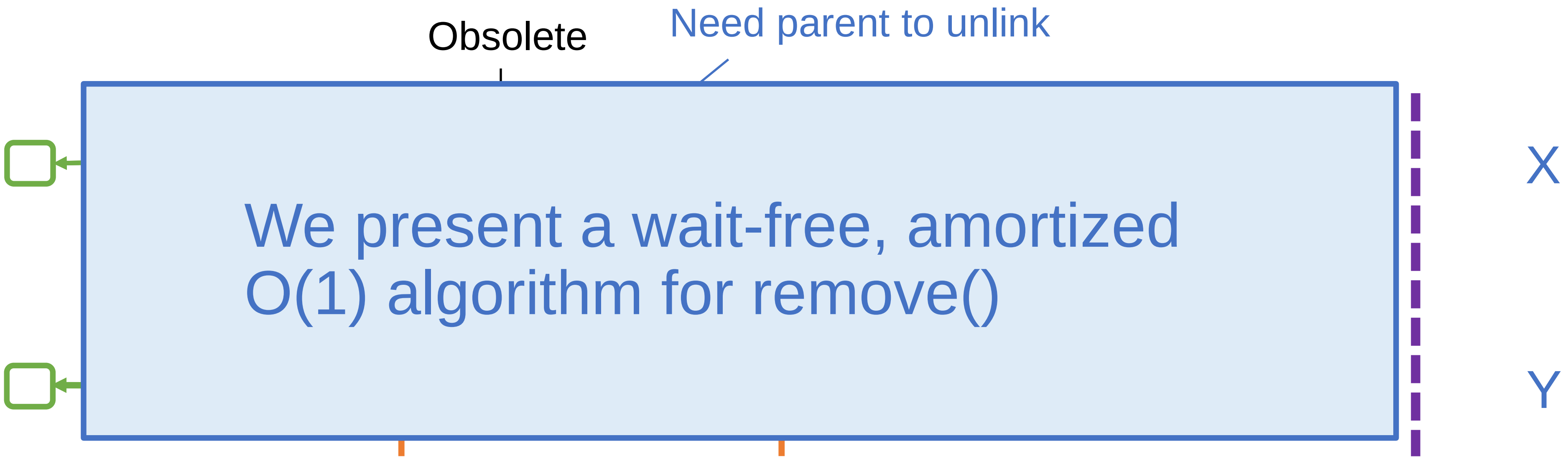
Step 1: Identify obsolete versions

Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions

Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list

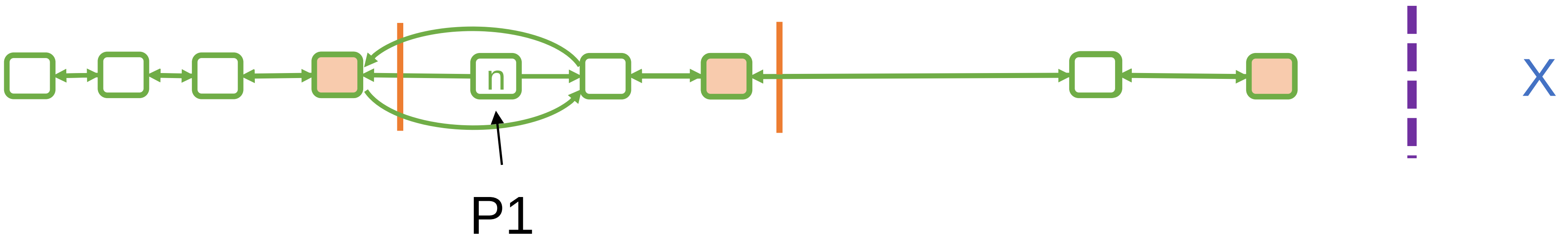


We present a wait-free, amortized O(1) algorithm for remove()

Step 3: Reclaim memory of unlinked versions

Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list
Step 3: Reclaim memory of unlinked versions

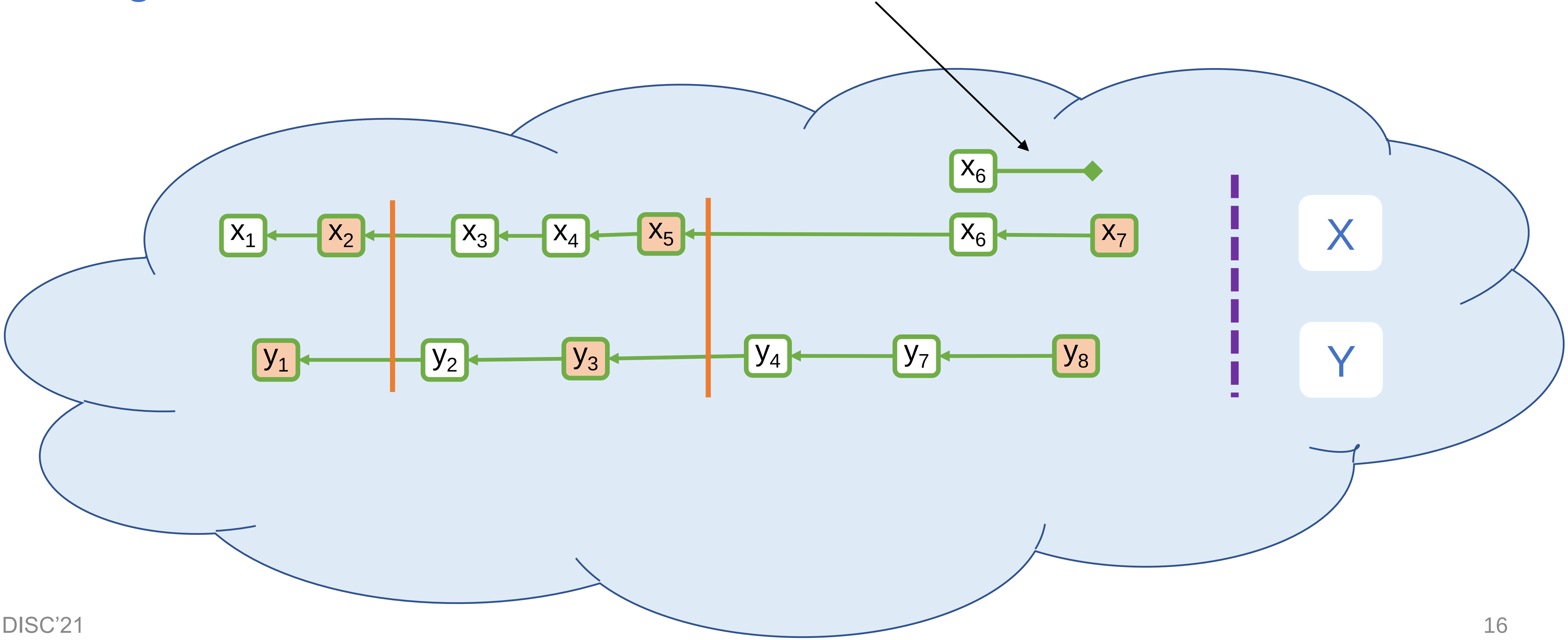


- n is not safe to reclaim right away because a process (P1) could be paused on it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
  - HP sacrifices wait-freedom
  - CRC sacrifices space bounds
- We design a new safe reclamation scheme specifically for our doubly linked version list

Step 1: Identifying Obsolete Versions

Range tracker:

Triplet [version, beginTS, endTS]



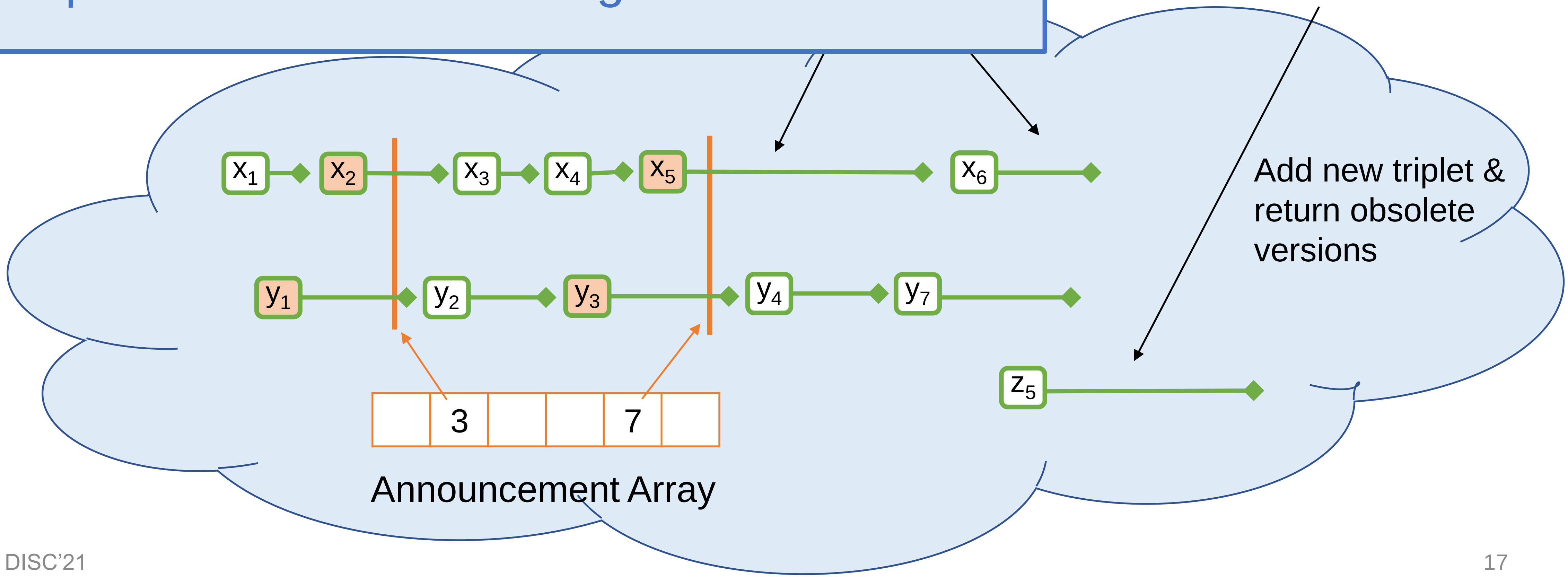
Step 1: Identifying Obsolete Versions

Observation: Triplets added by the same process have increasing endTS.

endTS]

Amortized O(1) time

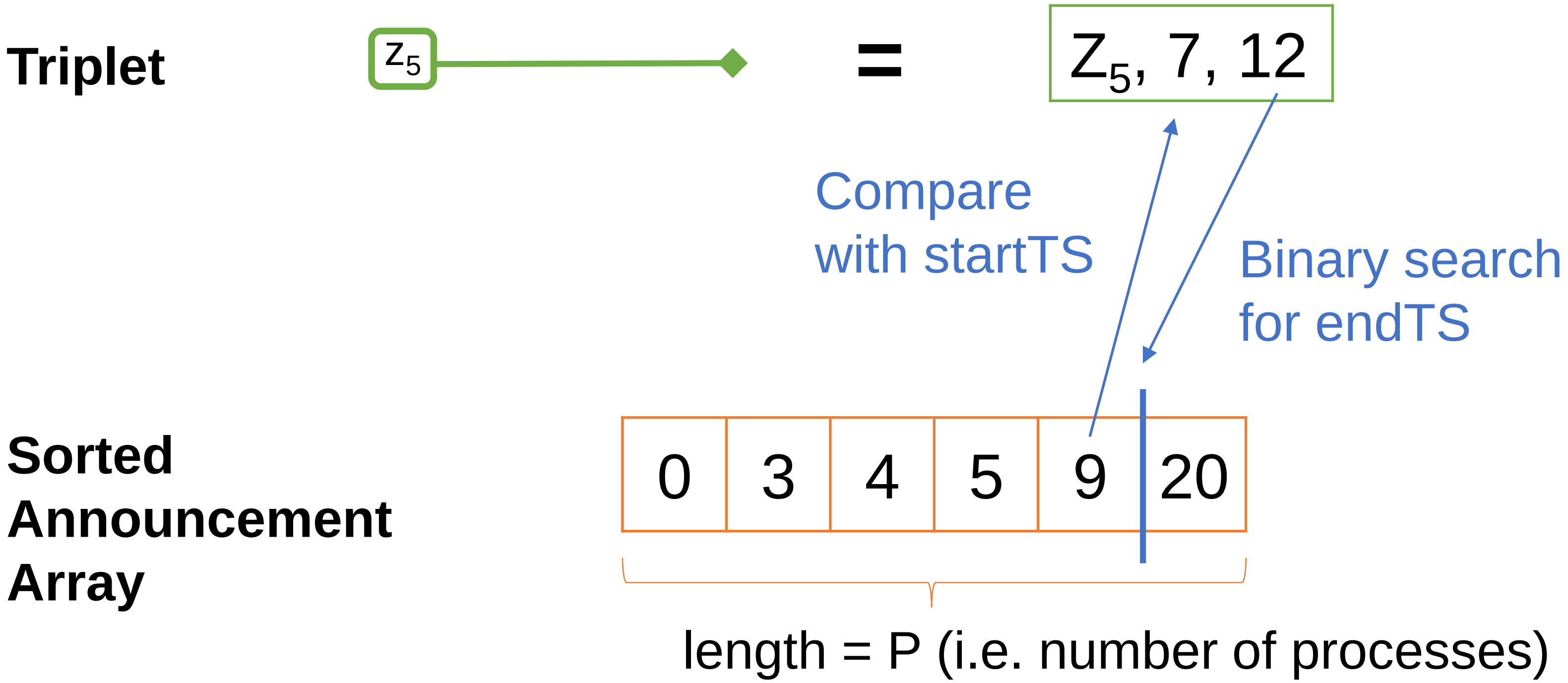
deprecate(z_5, startTS, endTS)



Step 1: Identifying Obsolete Versions

Question: Given a triplet T and a sorted announcement array, how long does it take to check if T is obsolete?

O(log P) time



Step 1: Identifying Obsolete Versions

O(P log P)

Question: What if you were given a batch of ~~O(P)~~ triplets sorted by end timestamp?

O(P log P) time

O(P) time

Batch of O(P log P) Triplets

Batch of O(P) Triplets

Obsolete

Z_5, 2, 8	X_3, 7, 12	…	Y_7, 10, 18

Binary search for all end points using merge()

Takes O(P log P) time

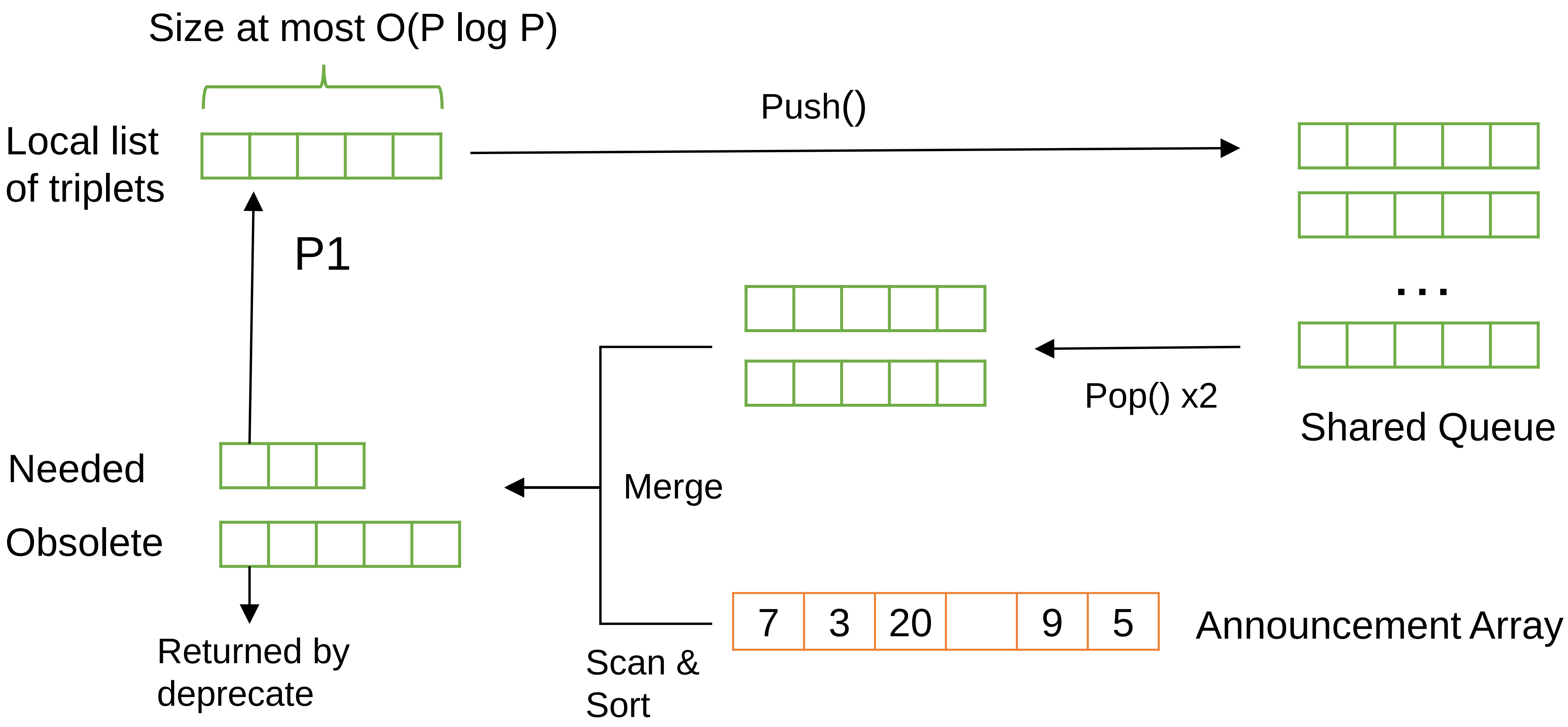
Takes O(P) time

Sorted Announcement Array

1	3	4	5	9	20

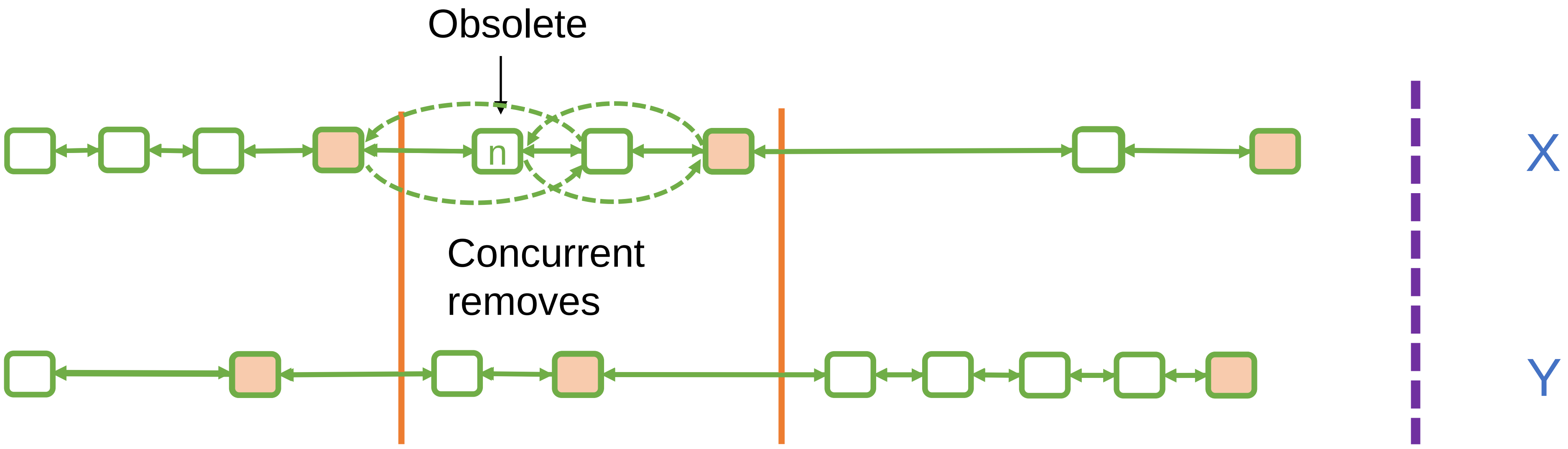
length = P (i.e. number of processes)

Range Tracker: Implementation



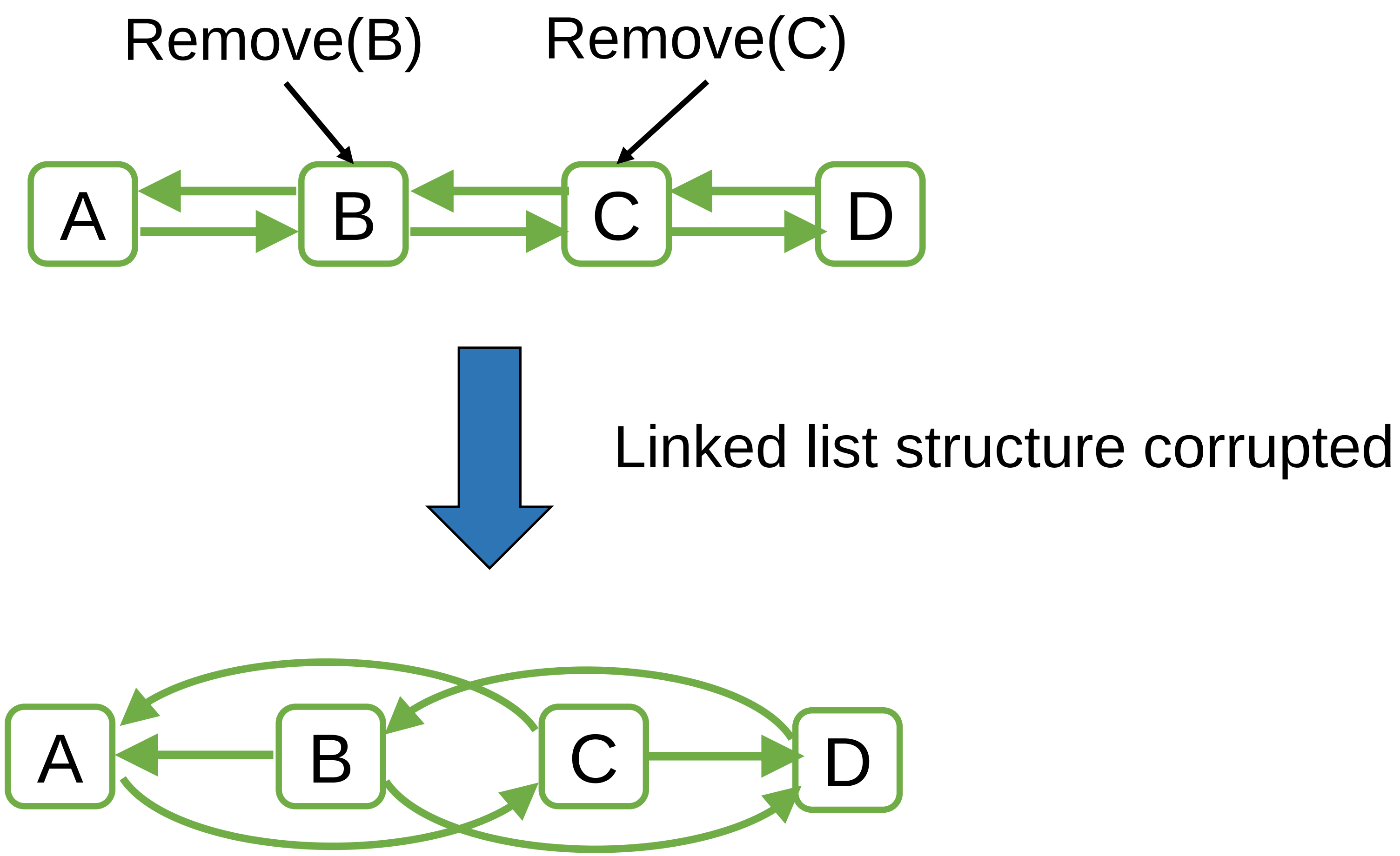
Overview

Step 1: Identify obsolete versions
Step 2: Unlink from version list



Step 3: Reclaim memory of unlinked versions

Concurrent Removes

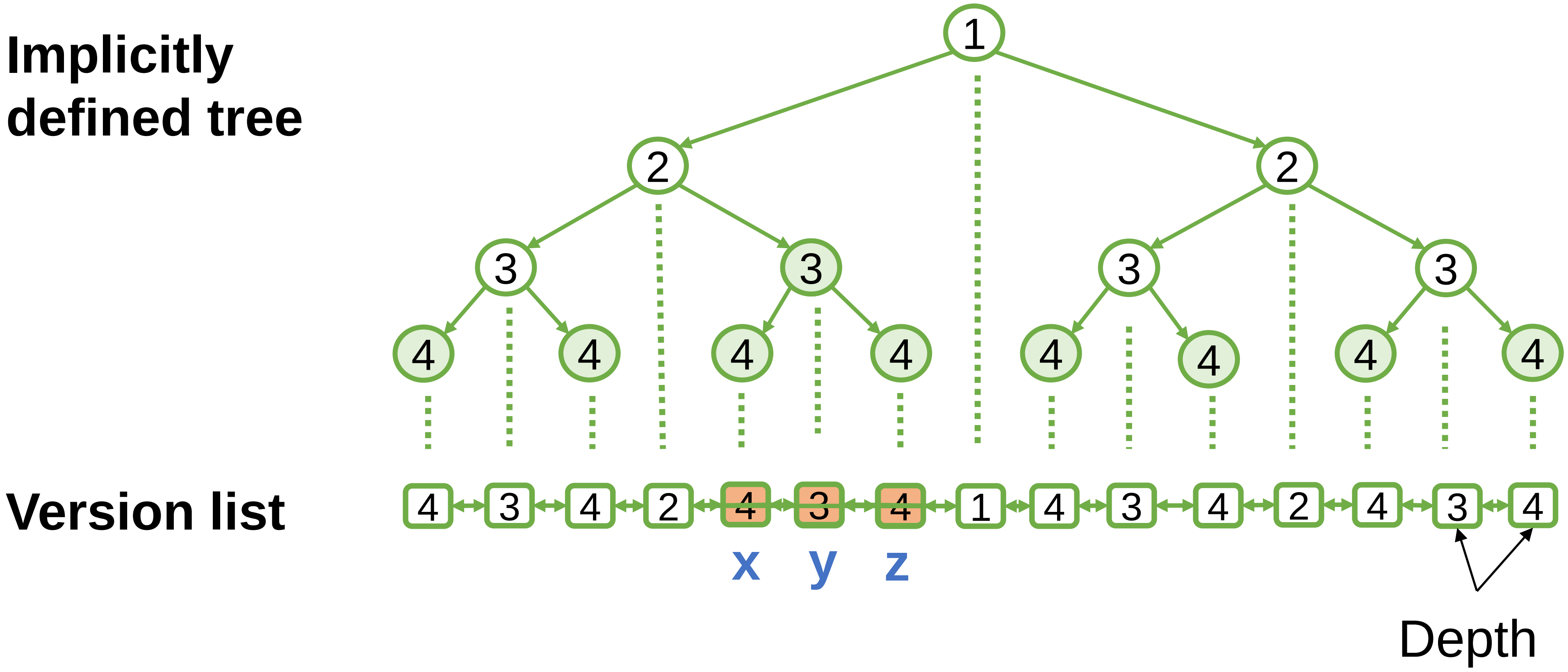


Linked List Remove

Amortized O(1) time

Worst case O(log L) time

L = # of nodes appended to version list



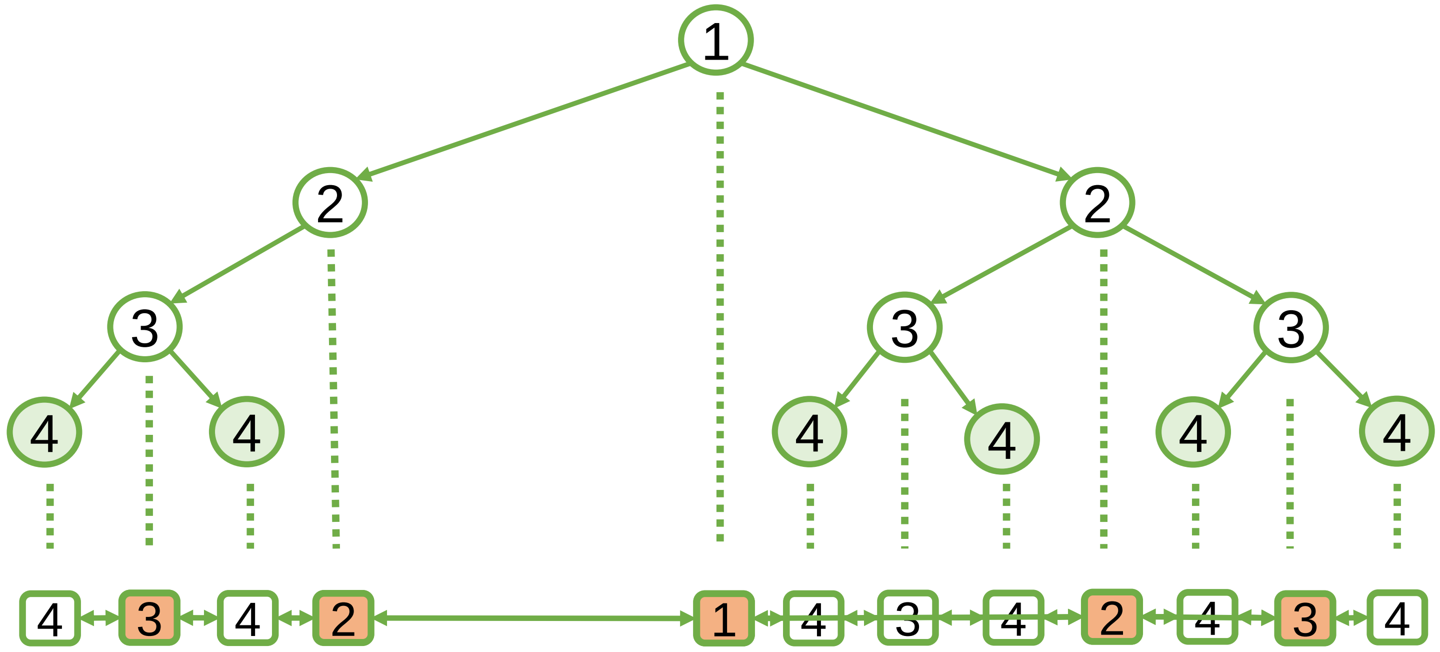
Space Overhead

O(log L) factor space overhead!

L = # of nodes appended to version list

Implicitly defined tree

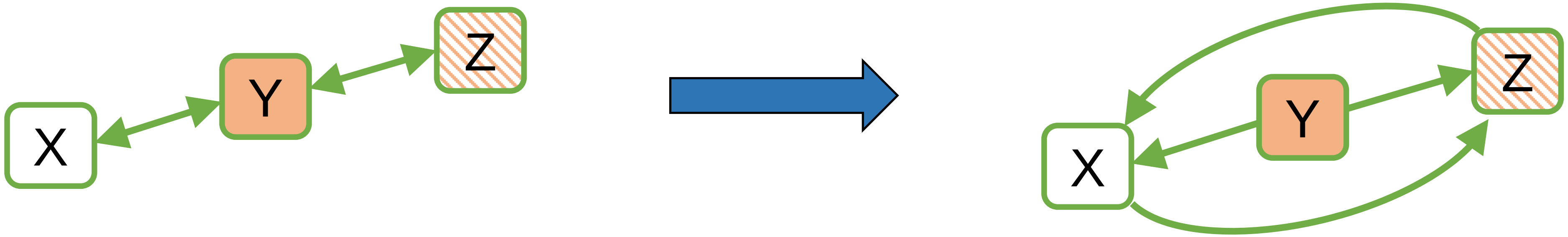
Version list



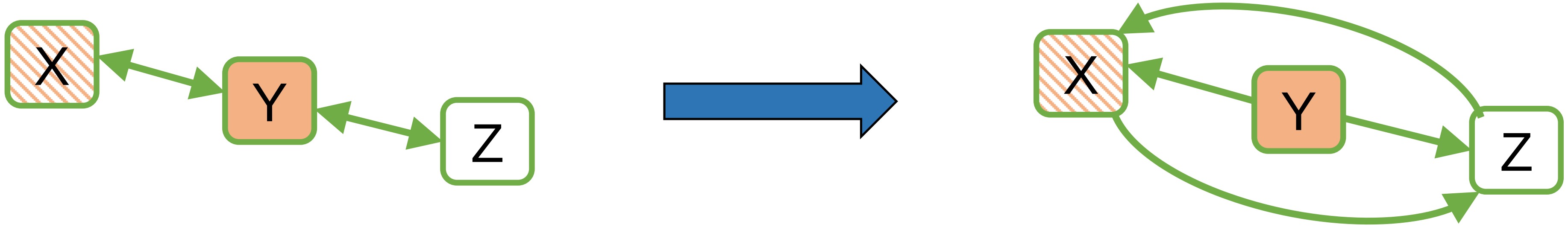
Splicing Out Internal Nodes

May or may not be marked

SpliceUnmarkedLeft(Y): requires **X > Y > Z** and **X unmarked**



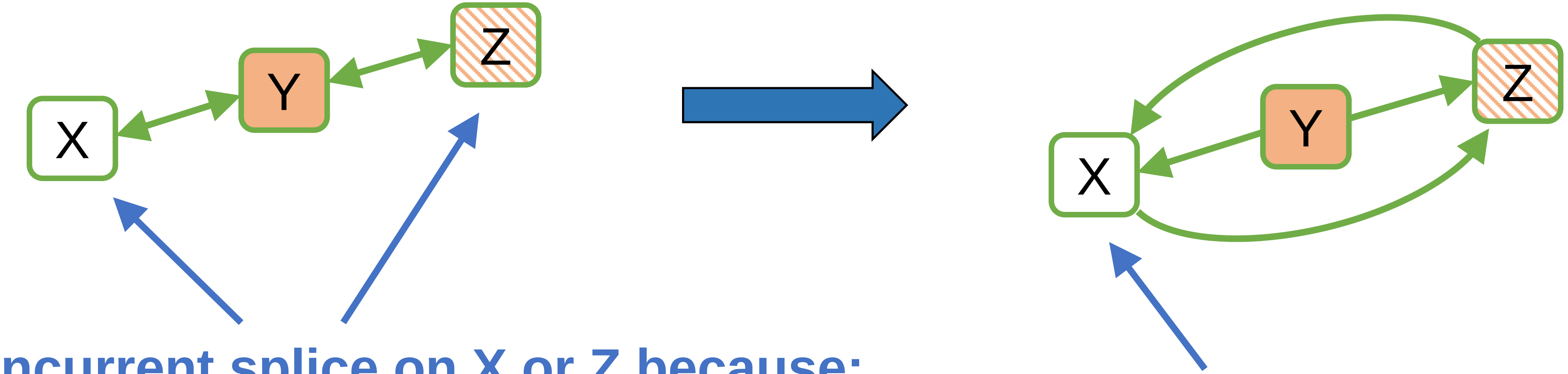
SpliceUnmarkedRight(Y): requires **X < Y < Z** and **Z unmarked**



Splicing Out Internal Nodes

May or may not be marked

SpliceUnmarkedLeft(Y): requires **X > Y > Z** and **X unmarked**

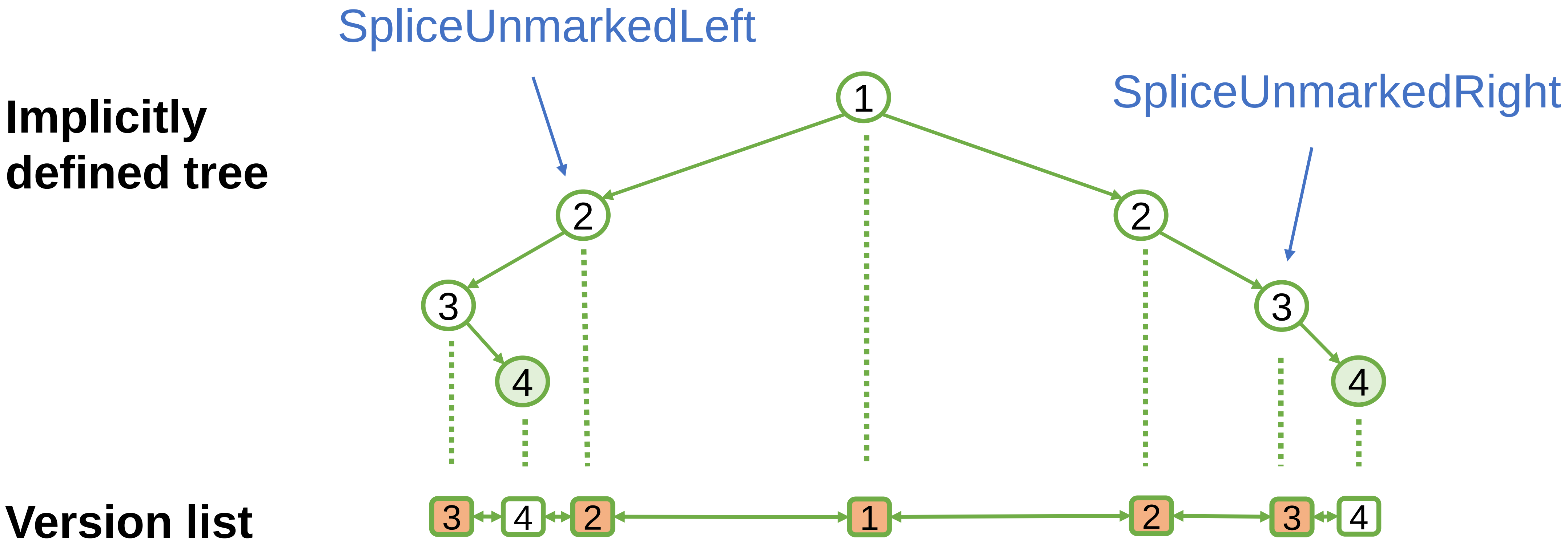


No concurrent splice on X or Z because:

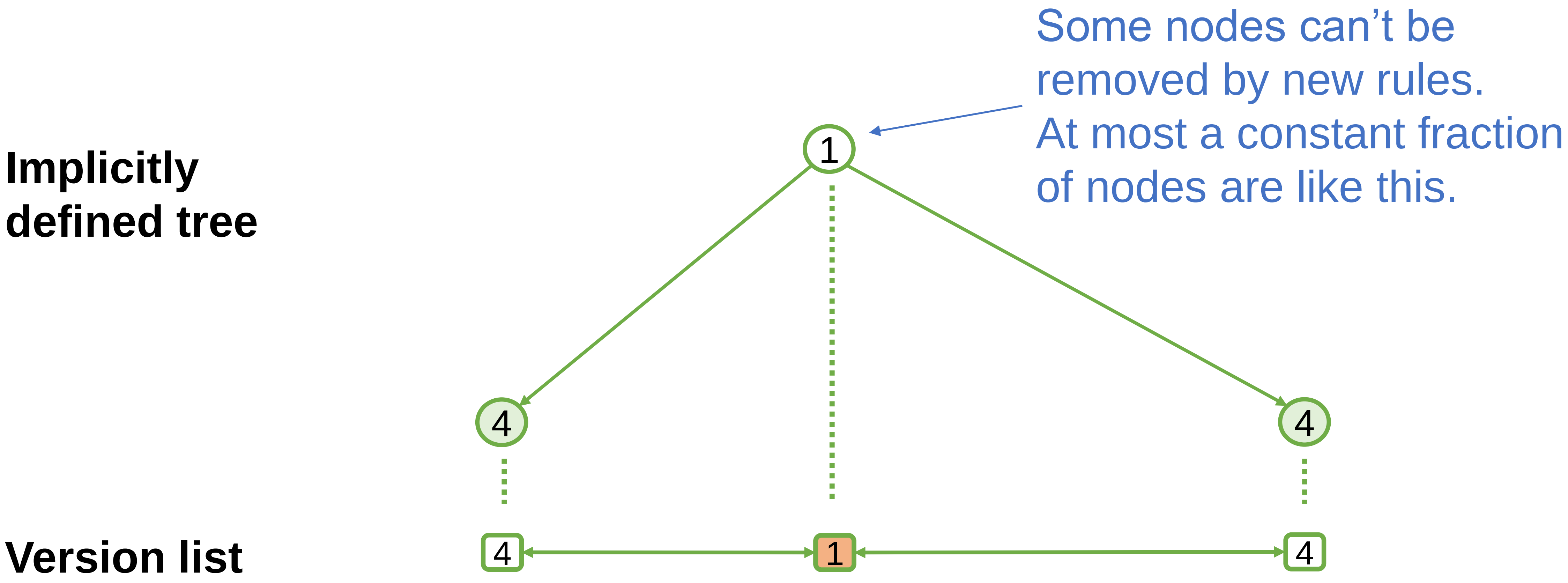
- X is not marked
- Z is an internal node and Y is marked

X cannot be marked for entire duration of the splice on Y

Splicing Out Internal Nodes



Splicing Out Internal Nodes



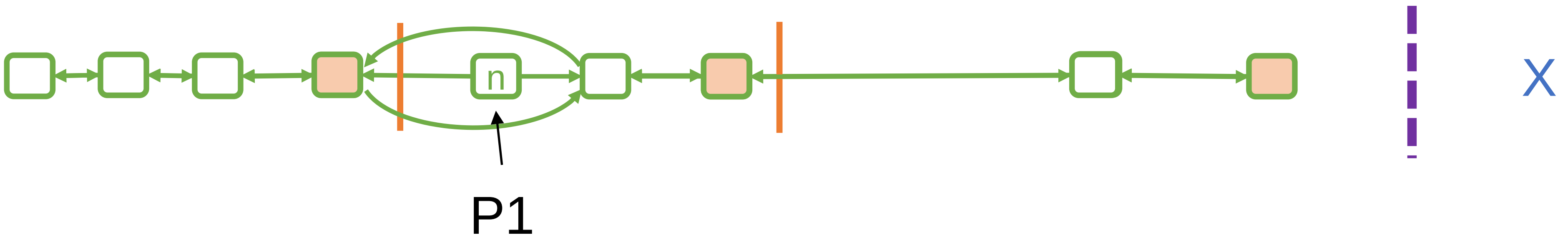
Doubly Linked List

	TryAppend (worst-case)	Remove (amortized)	Remove (worst-case)	Space
Our Results	O(1)	O(1)	O(log L), Wait-free	O(S + c log L)

L = # of successful TryAppends
S = # of nodes appended but not removed
c = # of ongoing remove operations

Overview

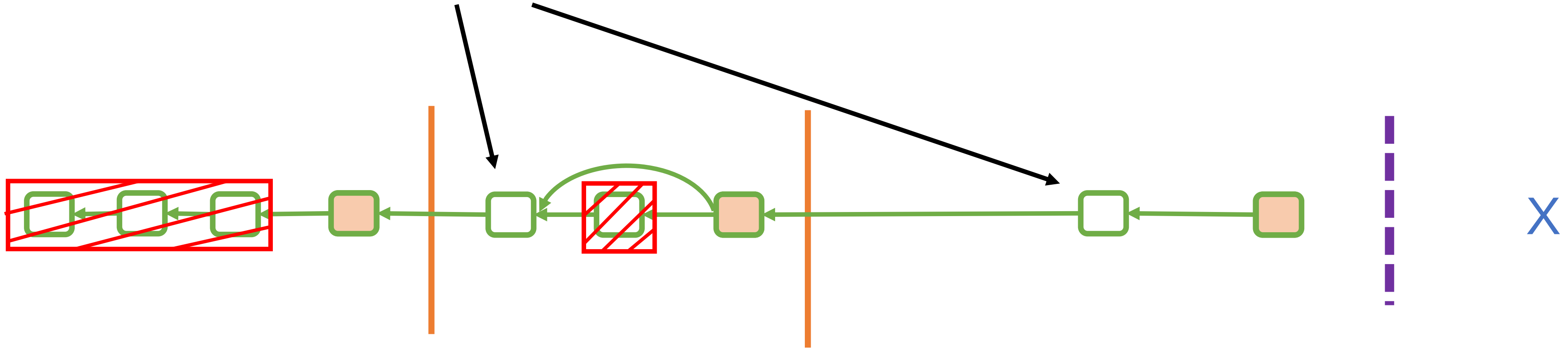
Step 1: Identify obsolete versions
Step 2: Unlink from version list
Step 3: Reclaim memory of unlinked versions



- n is not safe to reclaim right away because a process (P1) could be paused on it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
  - HP sacrifices wait-freedom
  - CRC has bad worst case space bounds
- We design a new safe reclamation scheme specifically for our doubly linked version list

Overall Results

- Time bounds:
  - O(1) time, on average, to identify, remove, and reclaim a version
  - Wait-free
- Space bounds:
  - Number of unreclaimed versions ∊ O(# required versions) + additive term



Space Bounds

- Number of unreclaimed versions $\in O(N + P^2 \log P + P \log L)$
  - N: high watermark number of needed versions throughout execution
  - P: number of processes
  - L: maximum number of versions added to a single version list
- In large data structures, $N > P^2 \log P + P \log L$

Conclusion

- We present a theoretically efficient solution to the MVGC problem
- Developed new techniques for all 3 steps:
  1. Identify obsolete versions
  2. Unlink from version list
  3. Reclaim memory of unlinked versions

- Currently working on a practical version of this algorithm, preliminary results look promising