RRR-SMR: Reduce, Reuse, Recycle: Better Methods for Practical Lock-Free Data Structures

Md Amit Hasan Arovi, arovi@psu.edu, The Pennsylvania State University, USA

Ruslan Nikolaev, rnikola@psu.edu, The Pennsylvania State University, USA

Lock-Free Data Structures

1. Allow multiple threads to operate without locks

2. Ensure system-wide progress (at least one thread always makes progress)

3. Avoid deadlocks and improve scalability

Lock-Free Data Structures

1. Lock-Free data structures are more difficult to use

2. In particular: not easy to move objects from one data structure to another

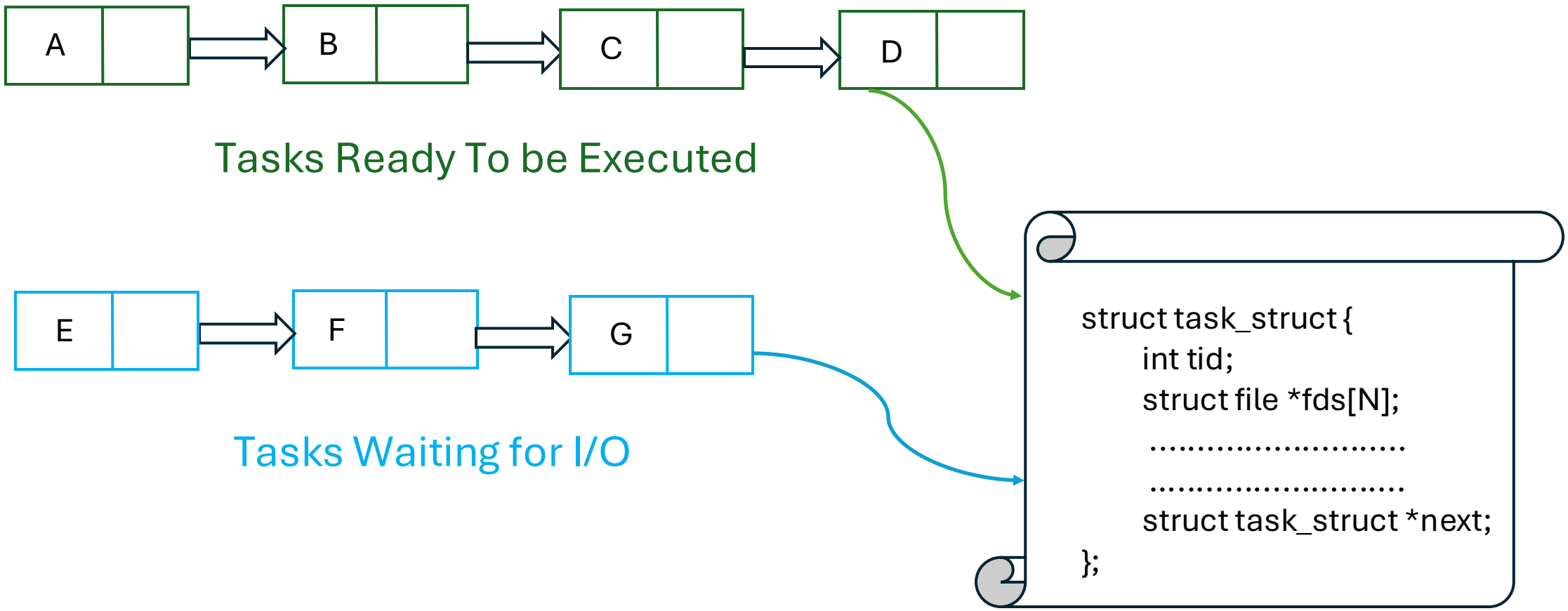
Lock-Free Data Structures

1. Lock-Free data structures are more difficult to use

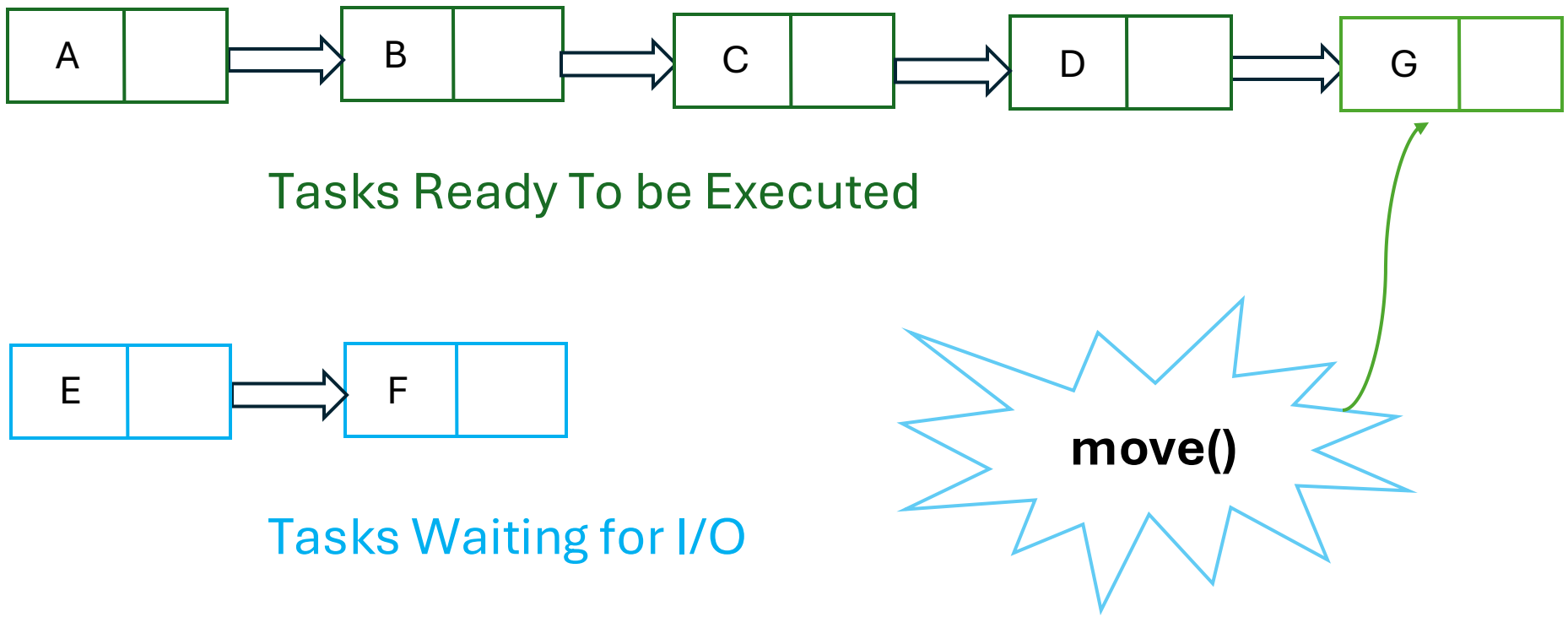
2. In particular: not easy to move objects from one data structure to another

In this paper we address this problem

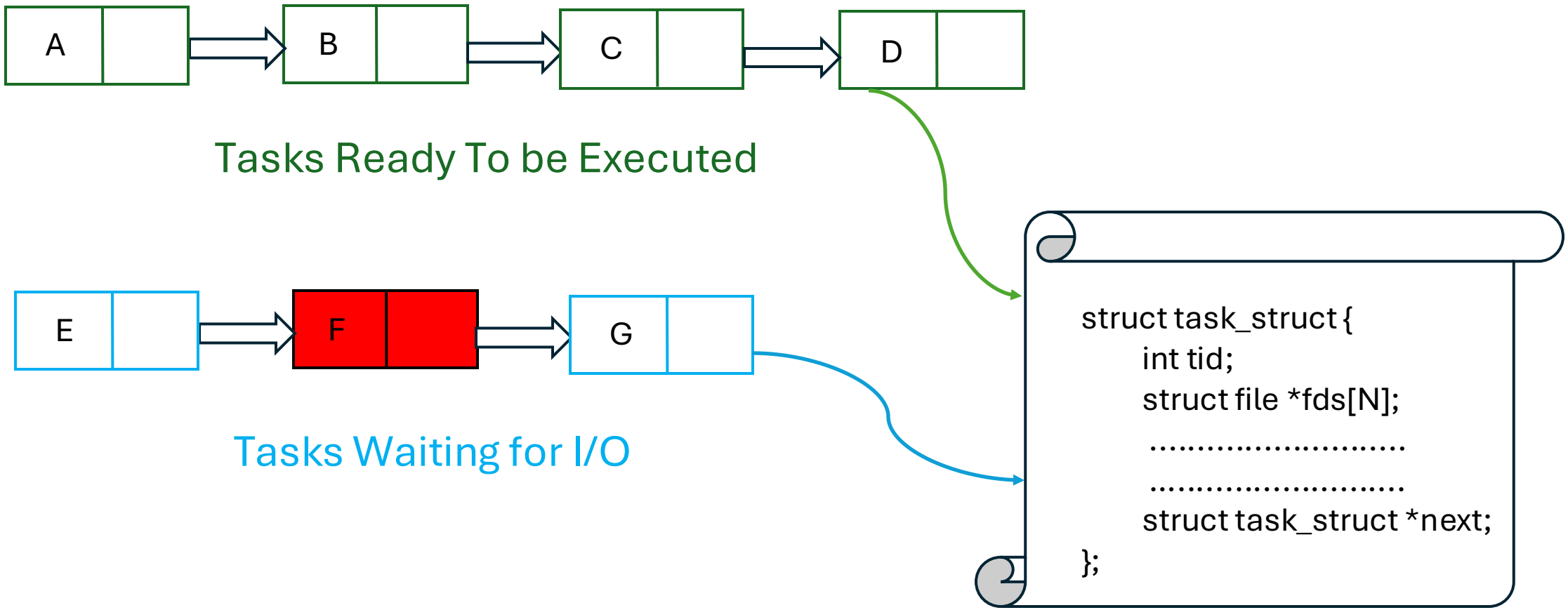
Example


A
B
C
D
Tasks Ready To be Executed
E
F
G
Tasks Waiting for I/O
struct task_struct {
int tid;
struct file *fds[N];
.....
.....
struct task_struct *next;
};

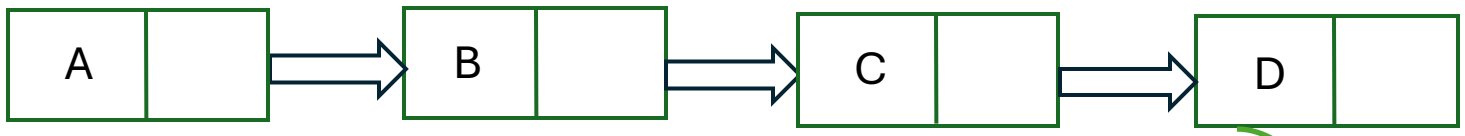
Example


A
B
C
D
G
Tasks Ready To be Executed
E
F
Tasks Waiting for I/O
move()

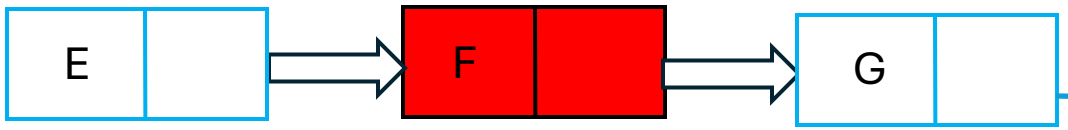
Example


A
B
C
D
Tasks Ready To be Executed
E
F
G
Tasks Waiting for I/O
struct task_struct {
int tid;
struct file *fds[N];
.....
.....
struct task_struct *next;
};

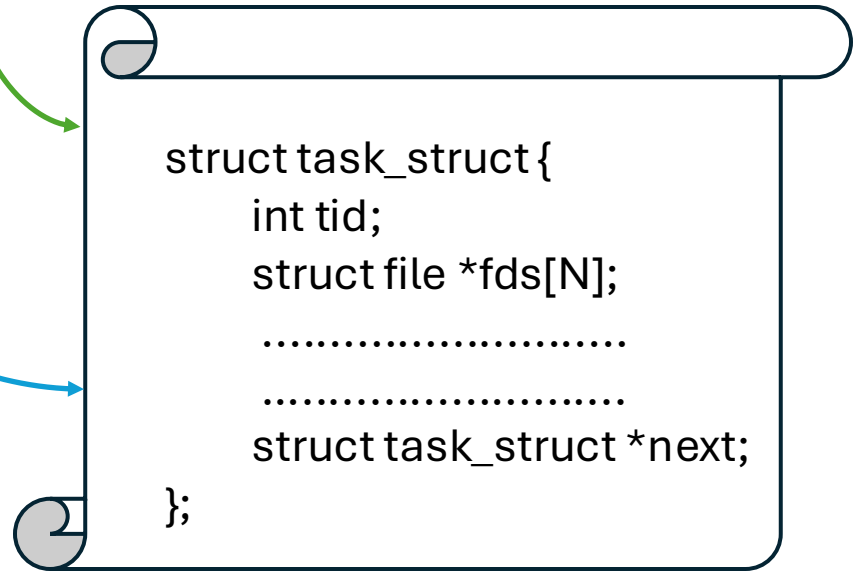
Example



Tasks Ready To be Executed



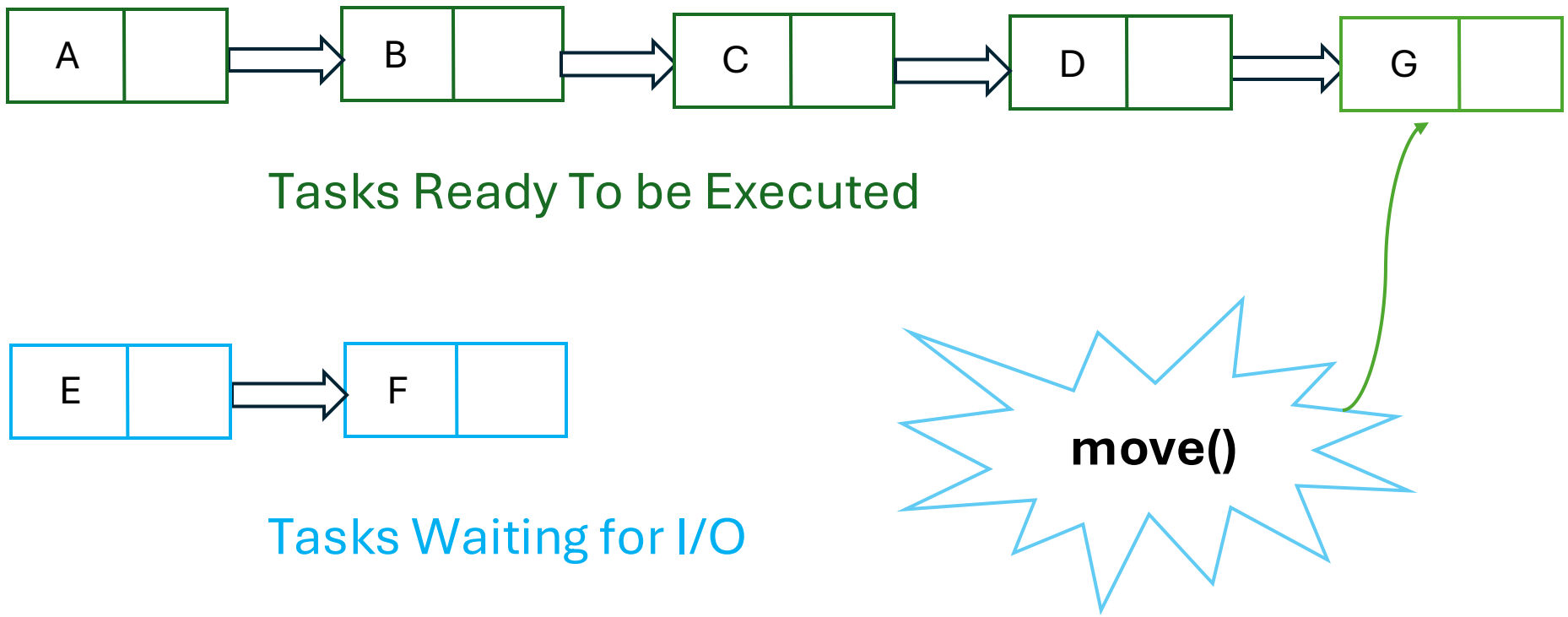
Tasks Waiting for I/O



Epoch-Based Reclamation (EBR)

Hazard Pointers (HP) [TPDS '04]

Example

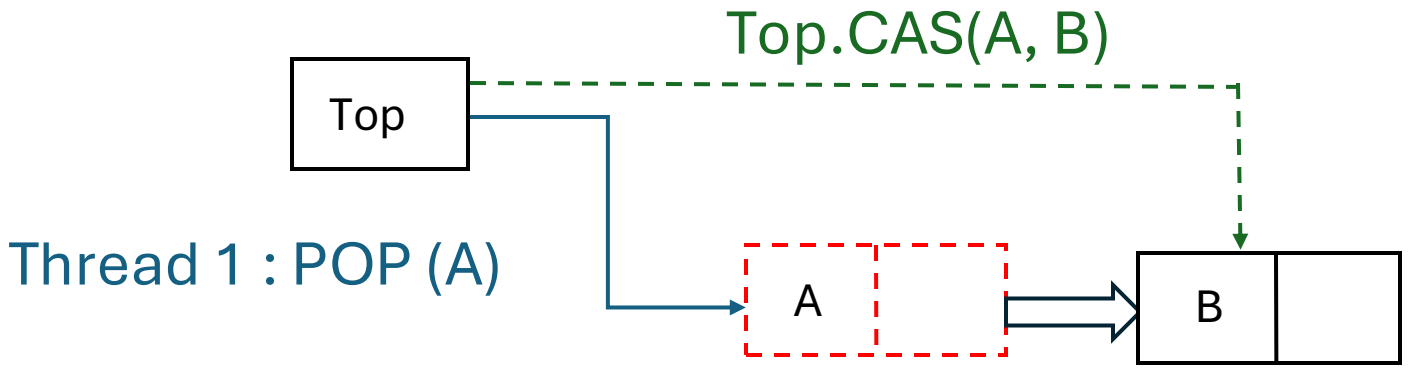


1. Can work around this problem by allocating a new object and copying

2. Comes with copying overheads and sometimes infeasible (interrupts)

Treiber's Stack

Treiber's Stack is a classic lock-free data structure that uses a singly-linked list and the compare-and-swap (CAS) primitive to push and pop elements from the top.

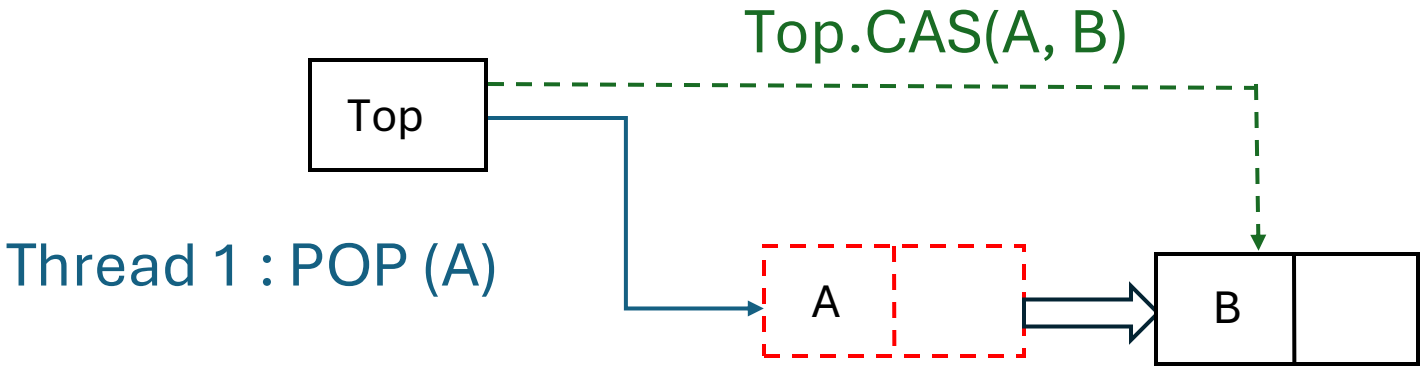


Treiber's Stack

CAS: Compare-And-Swap (CAS) is an atomic instruction that compares a memory location's current value to an expected value and updates it to a new value *only* if they match. It returns *true* if the update succeeds, or *false* otherwise.

Treiber's Stack

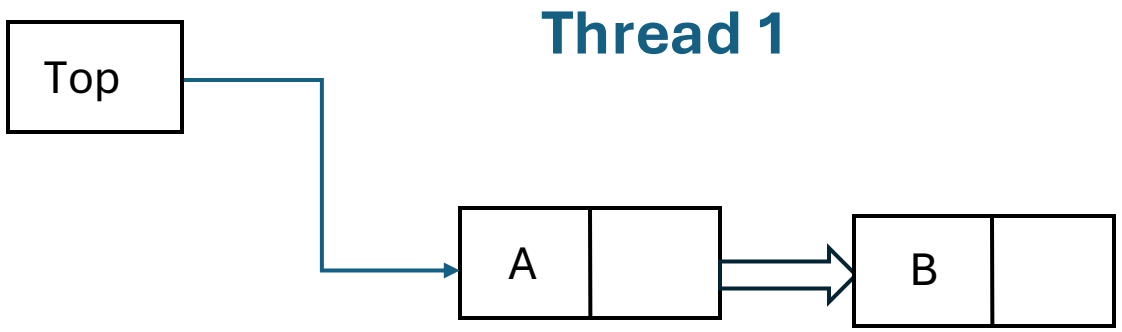
Treiber's Stack is a classic lock-free data structure that uses a singly-linked list and the compare-and-swap (CAS) primitive to push and pop elements from the top.



Treiber's Stack

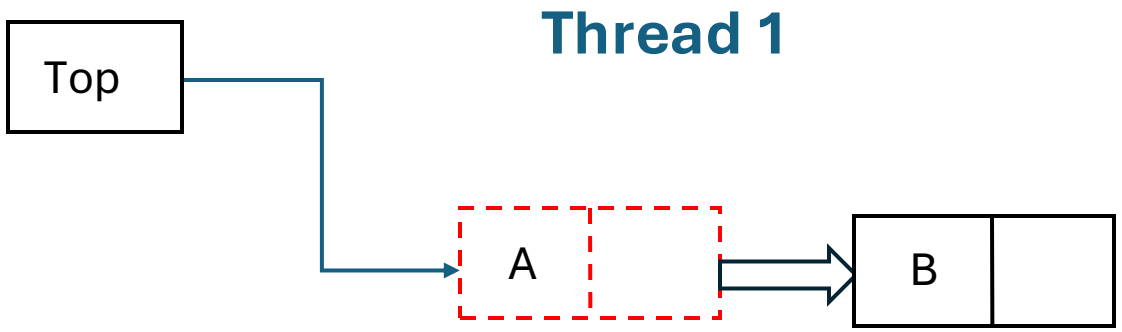
Can we directly move objects from one stack to another one?

Treiber's Stack: The ABA Problem



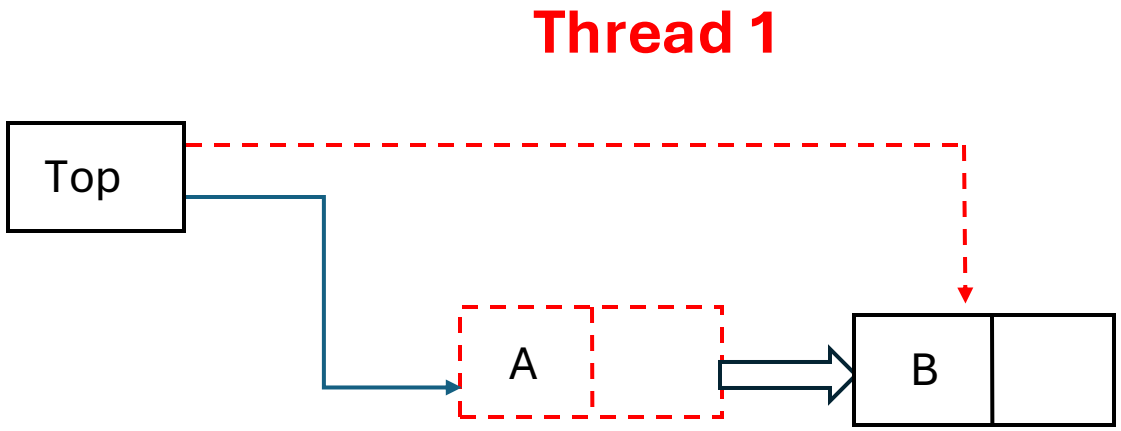
Thread 1 begins its operation.....

Treiber's Stack: The ABA Problem



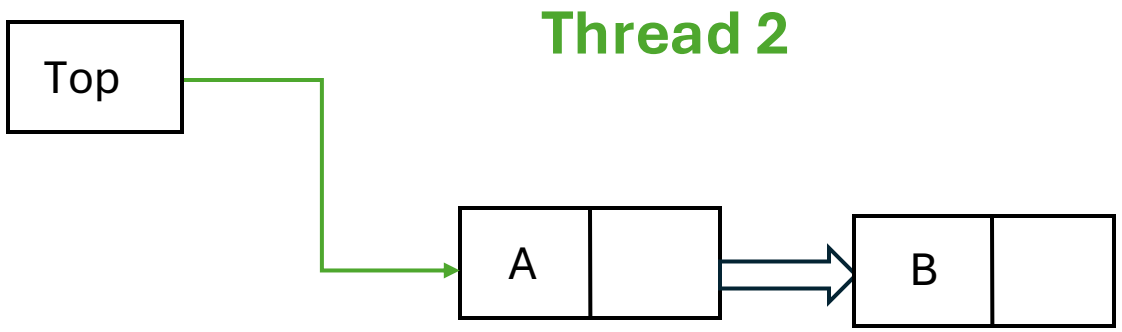
Thread 1 calls POP

Treiber's Stack: The ABA Problem



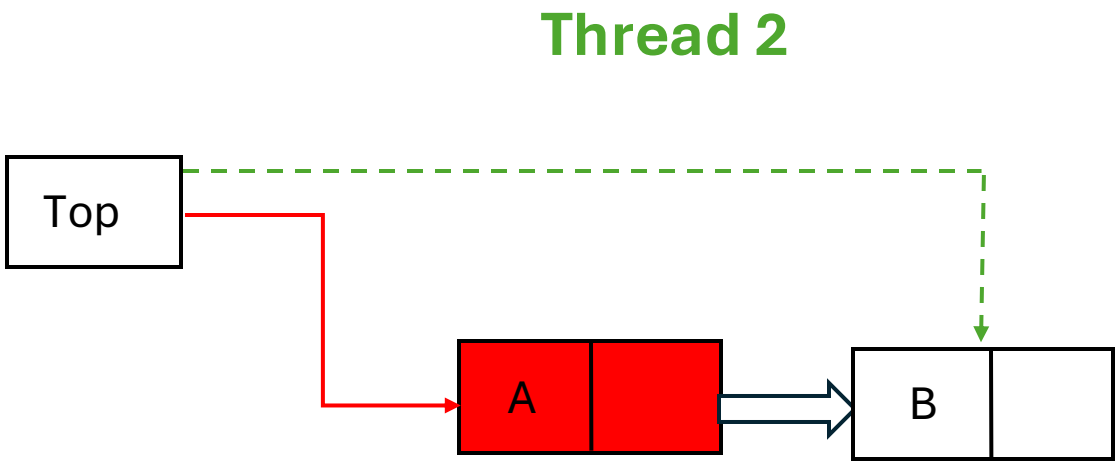
Thread 1 gets preempted before doing Top.CAS (A, B)

Treiber's Stack: The ABA Problem



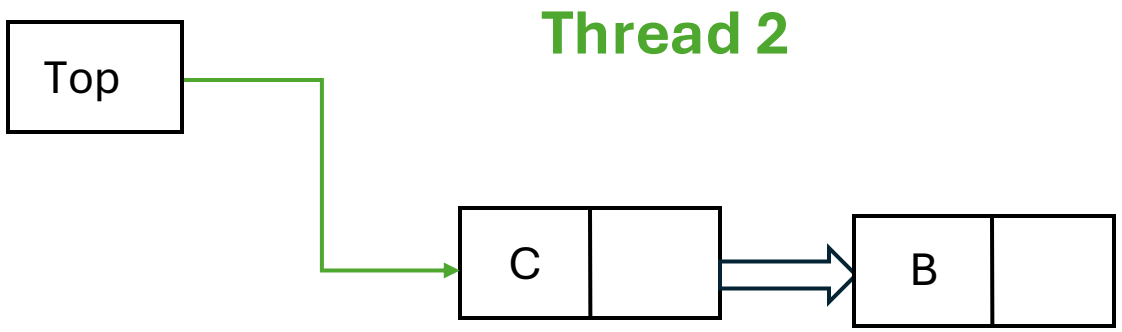
Thread 2 steps in……

Treiber's Stack: The ABA Problem



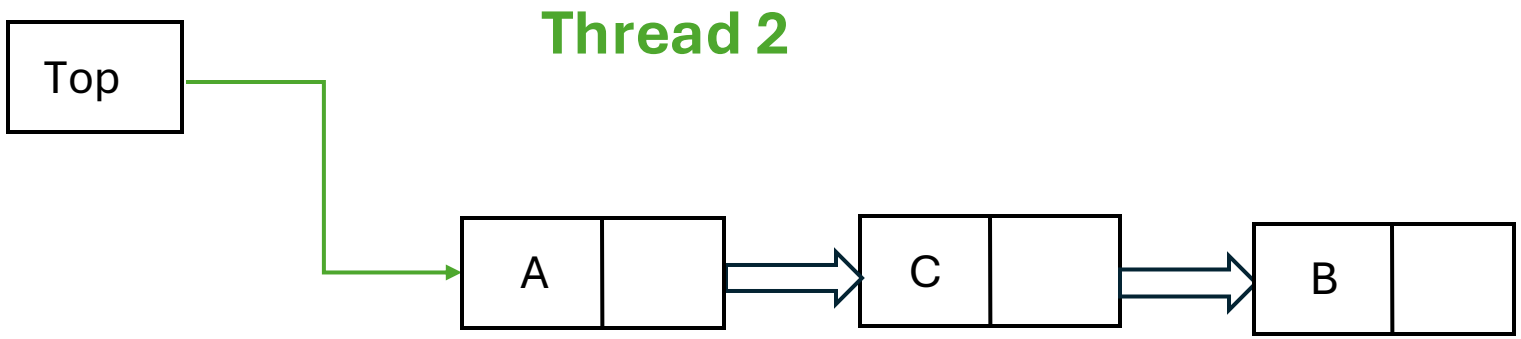
Thread 2 removes A (POP) and changes the top to B (CAS)

Treiber's Stack: The ABA Problem



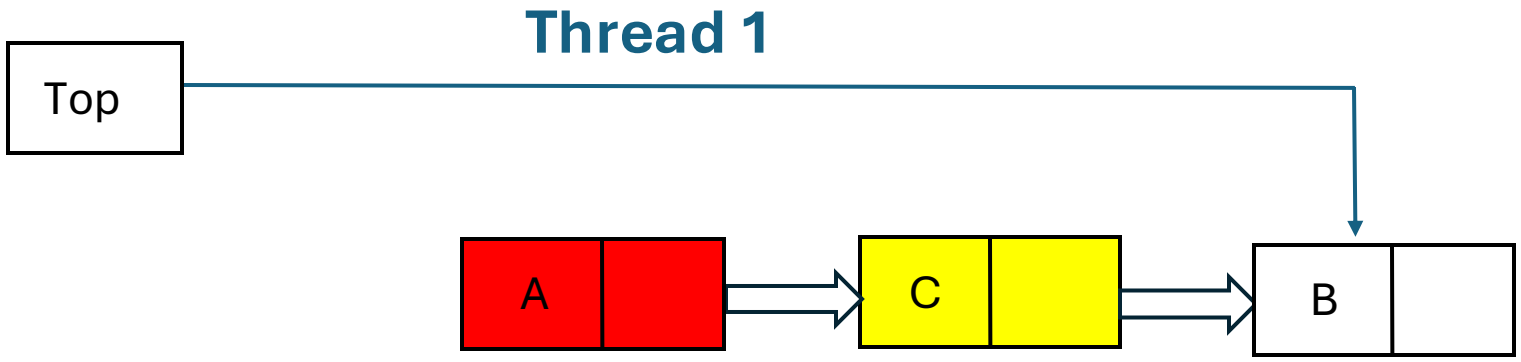
Thread 2 pushes C (new Top)

Treiber's Stack: The ABA Problem



Thread 2 pushes A (reusing the same memory address) and do CAS (Top, A)

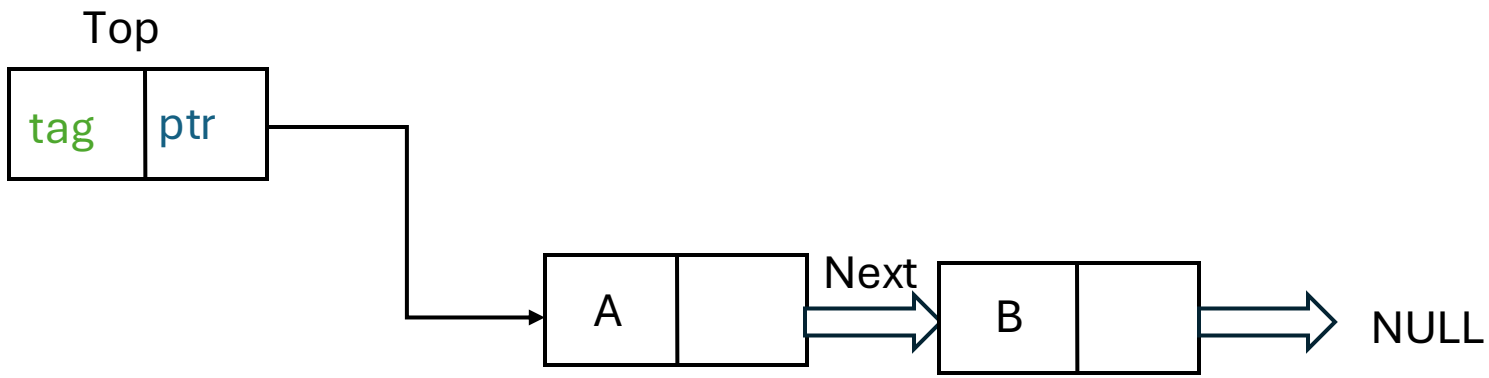
Treiber's Stack: The ABA Problem



Thread 1 resumes and completes its Top.CAS (A, B) leaving C hanging

False Positive!

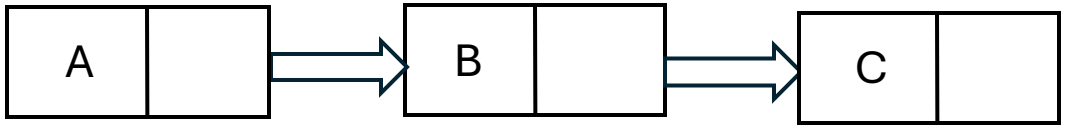
Treiber's Stack: Solution for ABA



Wide CAS (W-CAS) & CMPXCHG16B: W-CAS (Wide Compare-And-Swap) refers to atomic operations on larger-than-pointer-sized data (e.g., 128 bits) and essential for lock-free data structures that use tagged pointers (e.g., pointer + version/tag). On x86-64, the instruction CMPXCHG16B performs an atomic 16-byte (128-bit) compare-and-swap.

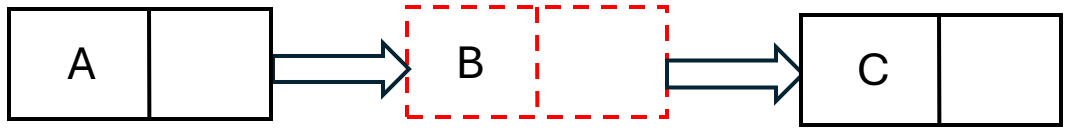
More complex Data Structures: Linked List

Harris' Lock-Free Linked List [DISC '01]



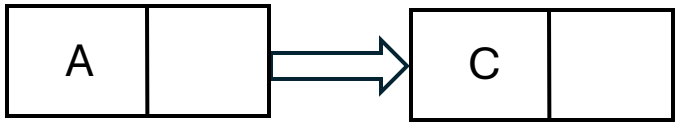
Nodes in a linked list can be removed *arbitrarily*

More complex Data Structures: Linked List



B is marked for deletion (logical)

More complex Data Structures: Linked List

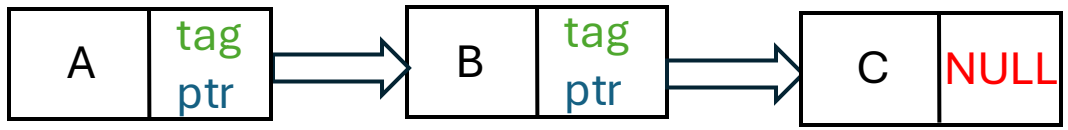


B is physically unlinked from the list

What needs to be solved for RRR Linked List?

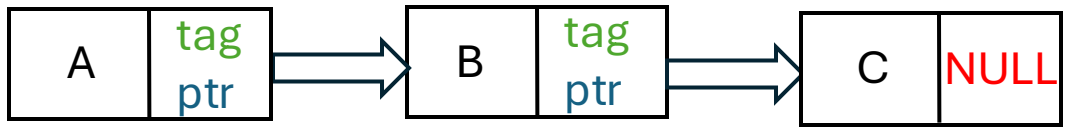
- ABA safety
- Strict ownership
- Safe traversal

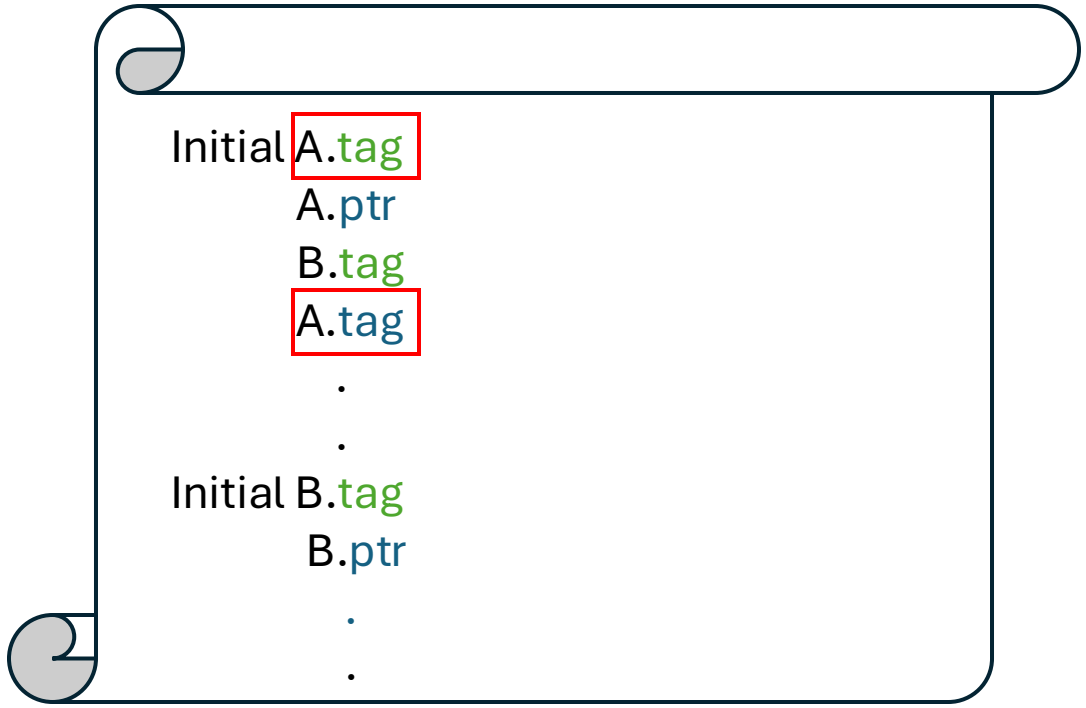
Problem 1 Solution (Linked List)



Initial A.tag
A.ptr
B.tag
A.tag
.
.
Initial B.tag
B.ptr
.
.

Problem 1 Solution (Linked List)

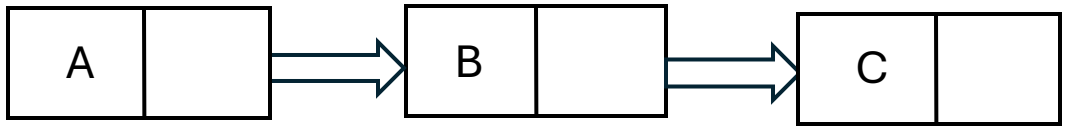




Problem 2 Solution (Linked List)

Strong Ownership (RRR-SMR Model)

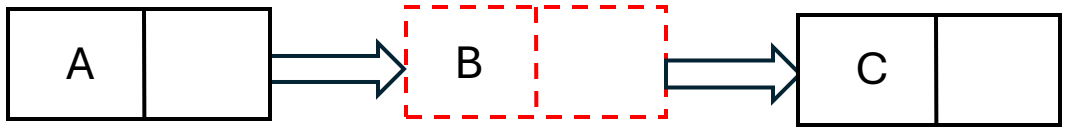
- Logical deletion thread takes ownership of the node
- The thread is responsible for both:
  - Making sure that the node is physically removed
  - Reclaiming memory



Problem 2 Solution (Linked List)

Strong Ownership (RRR-SMR Model)

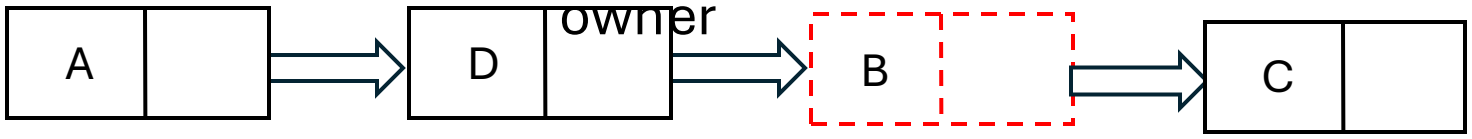
- Logical deletion thread takes ownership of the node
- The thread is responsible for both:
  - Making sure that the node is physically removed
  - Reclaiming memory
- After logical deletion, the thread attempts physical removal



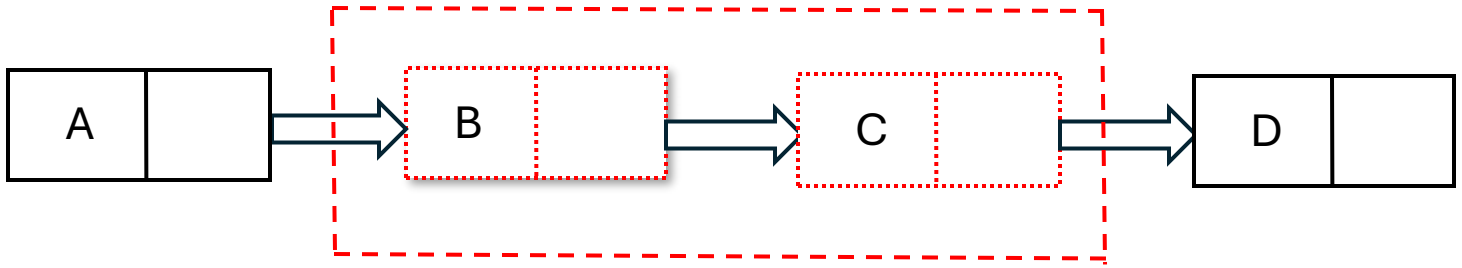
Problem 2 Solution (Linked List)

Strong Ownership (RRR-SMR Model)

- Logical deletion thread takes ownership of the node
- The thread is responsible for both:
  - Making sure that the node is physically removed
  - Reclaiming memory
- After logical deletion, the thread attempts physical removal
- Physical removal can fail due to:
  - Node was already removed (harmless)
  - The list state has changed
- We introduce a pruning method (Do_Prune):
  - It searches for the node later
  - Ensures safe physical removal by the owner

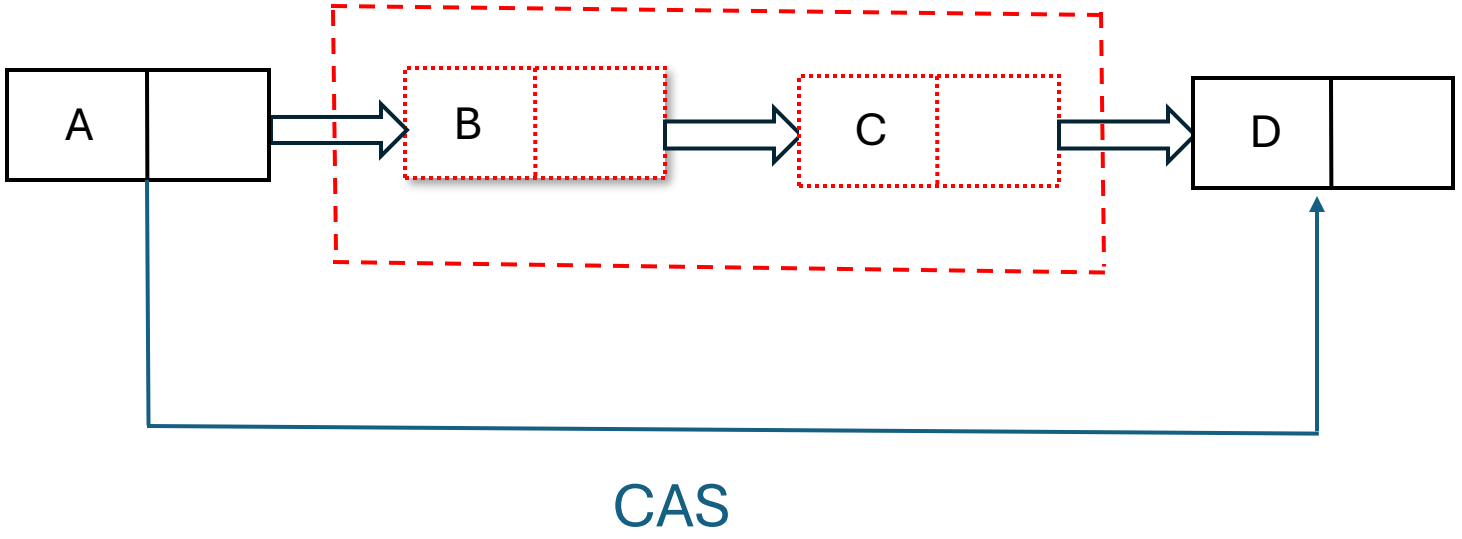


Problem 3 Solution (Linked List)



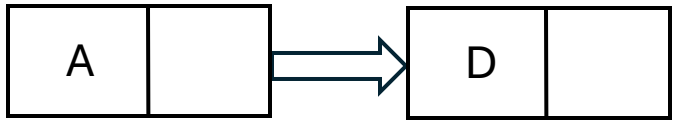
Two Adjacent Node B and C marked for deletion

Problem 3 Solution (Linked List)



CAS to physically unlink B and C

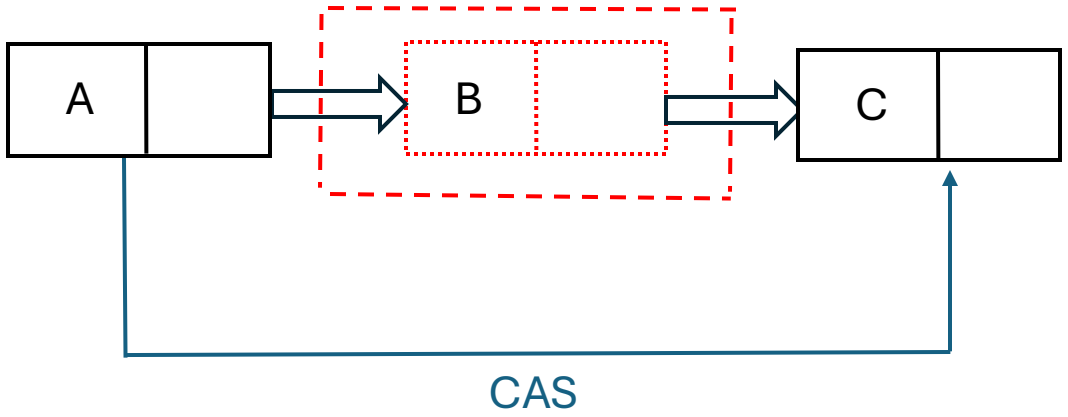
Problem 3 Solution (Linked List)



CAS successful, B and C removed from the list

Problem 3 Solution (Linked List)

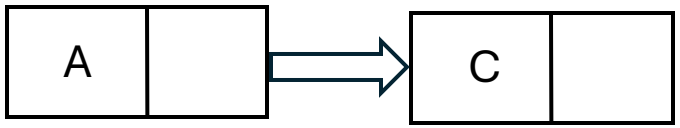
Michael's Modification [SPAA '02]



A thread sees B is marked for logical deletion and attempts to do CAS to remove it from the list

Problem 3 Solution (Linked List)

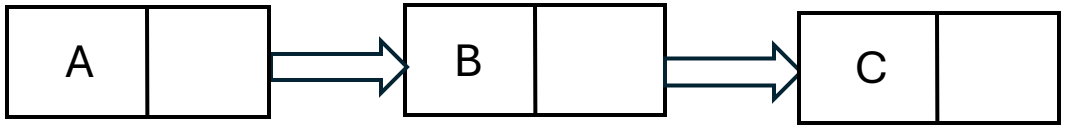
Michael's Modification [SPAA '02]



B is removed!

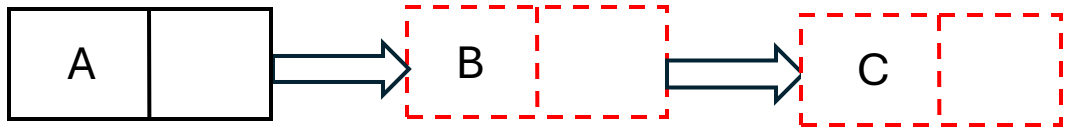
Problem 3 Solution (Linked List)

List 1:



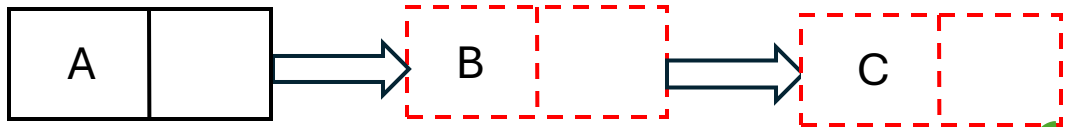
Problem 3 Solution (Linked List)

List 1:

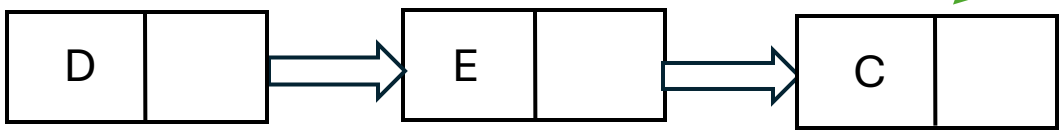


Problem 3 Solution (Linked List)

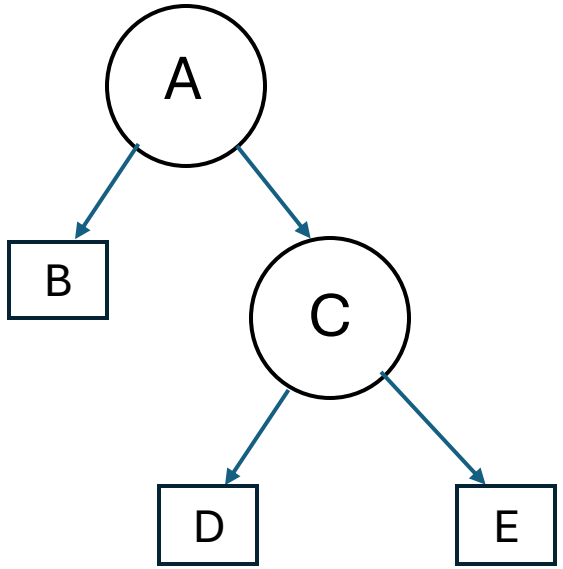
List 1:



List 2:



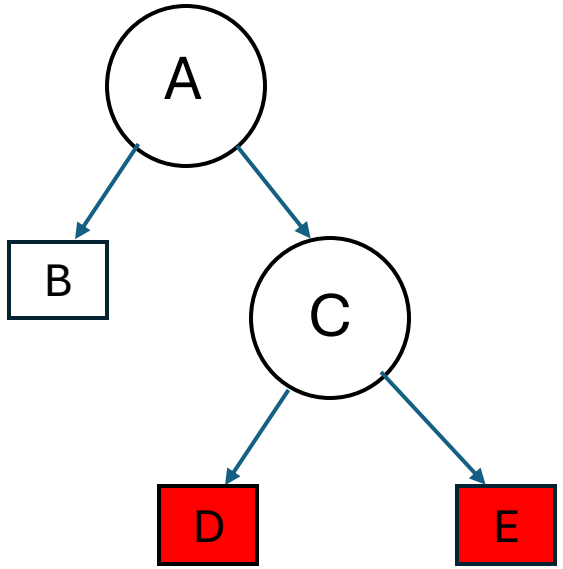
Natarajan-Mittal Tree



Leaf nodes contain actual keys

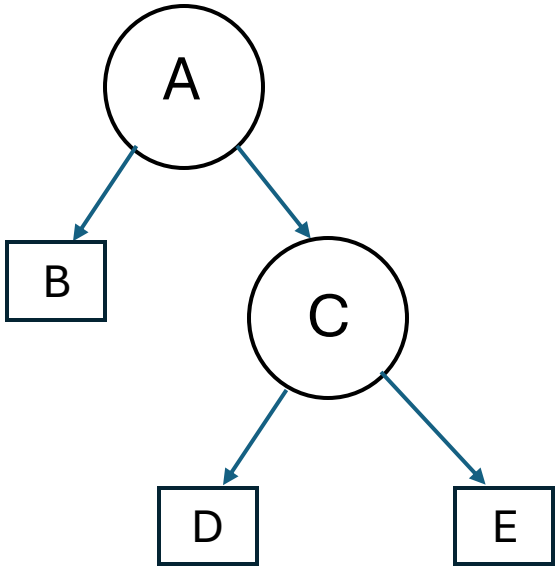
Keys in Internal nodes are used for traversal

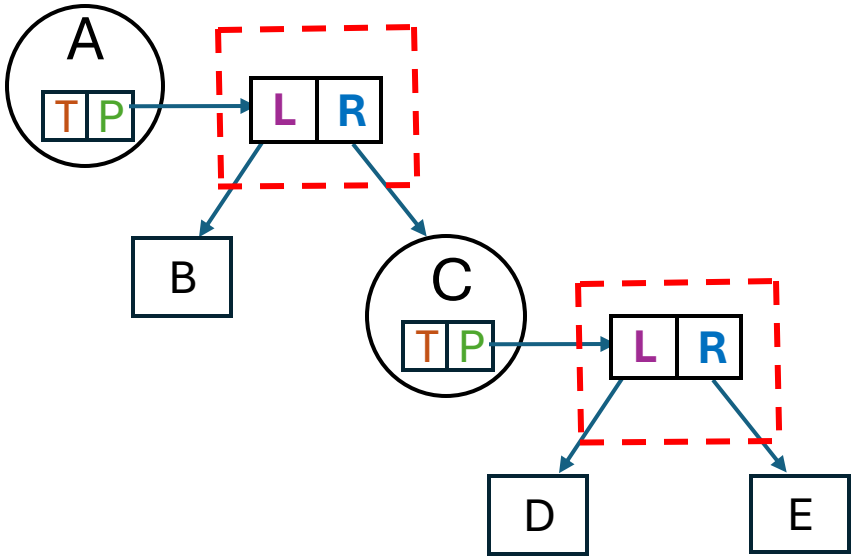
Natarajan-Mittal Tree

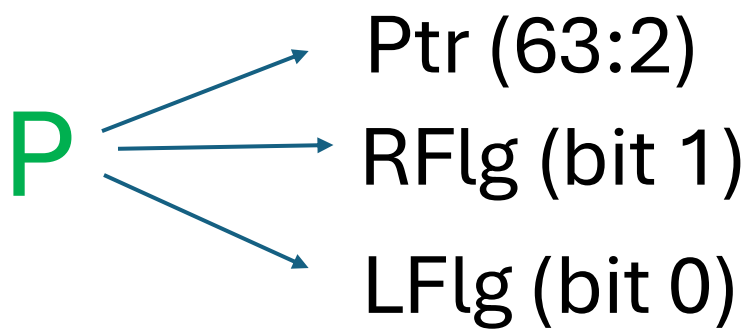


Two concurrent remove() operations

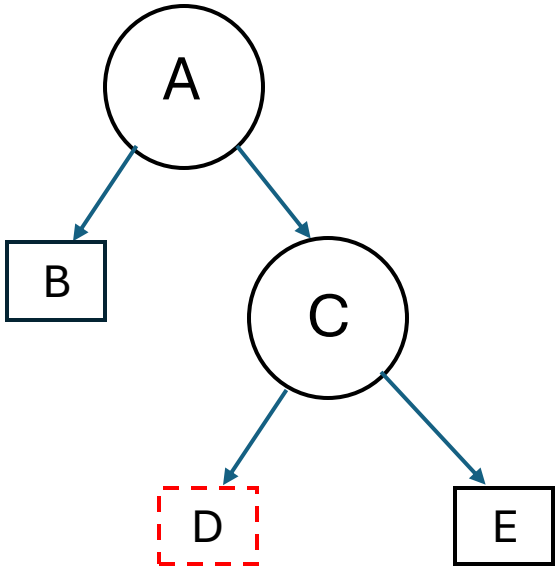
Natarajan-Mittal Tree

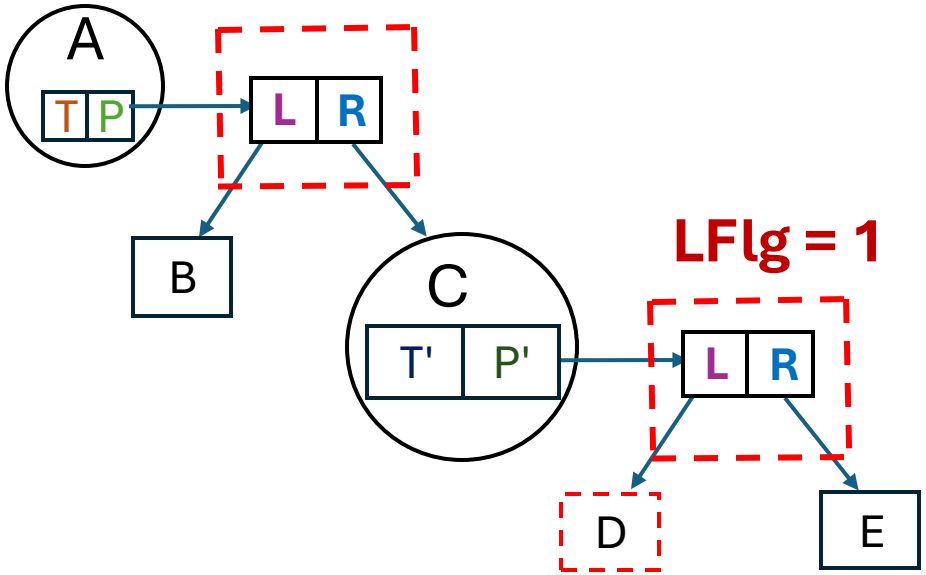






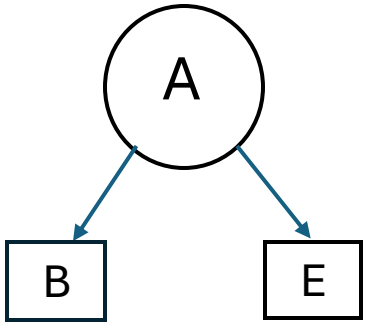
Natarajan-Mittal Tree

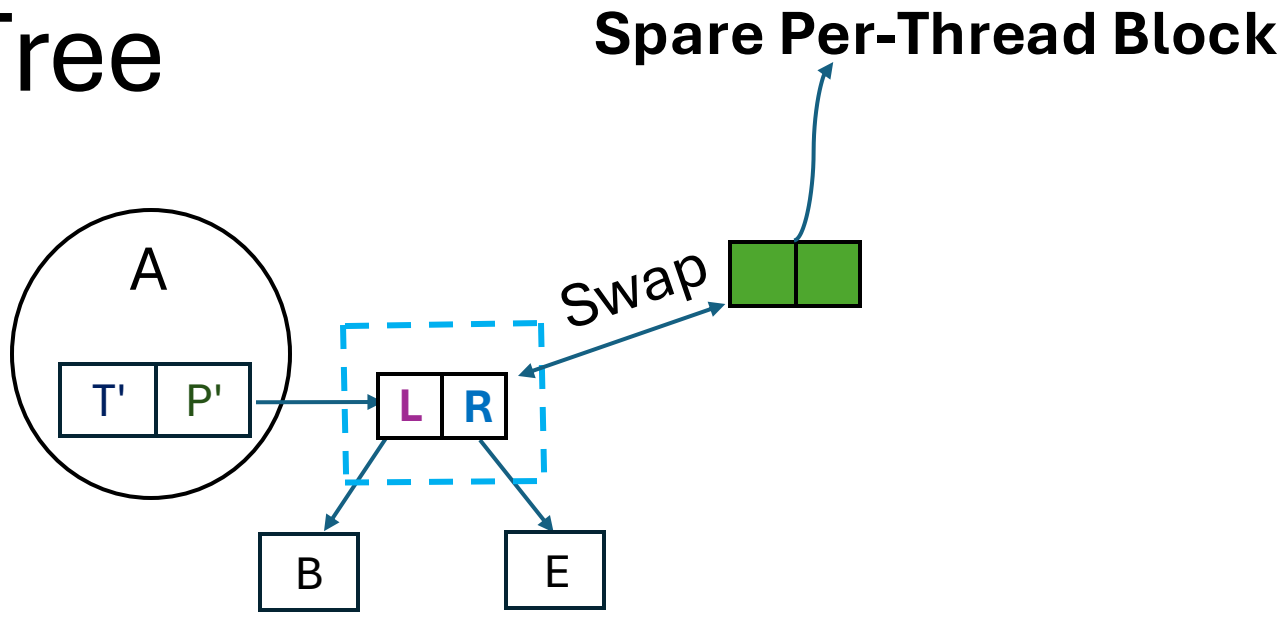




LFlg is set to mark D for logical deletion

Natarajan-Mittal Tree

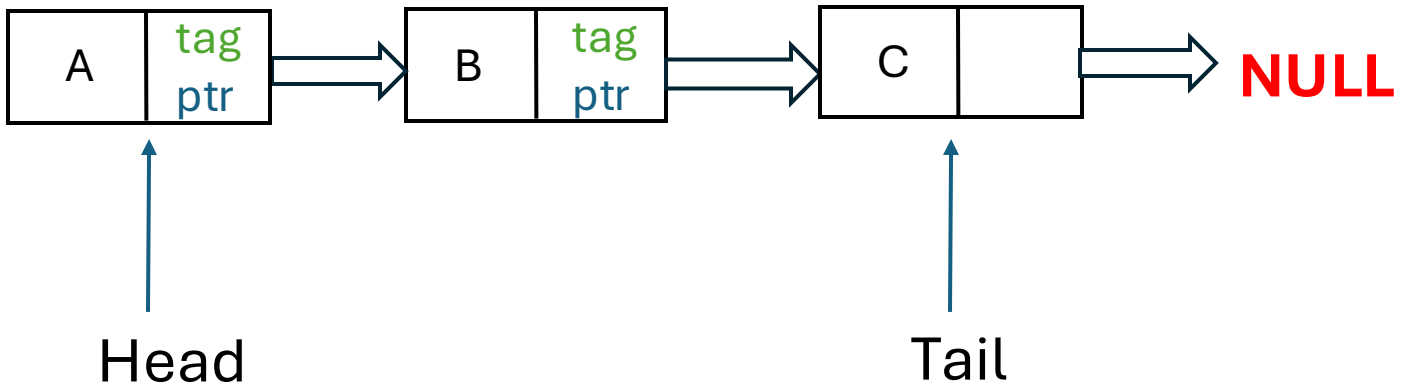




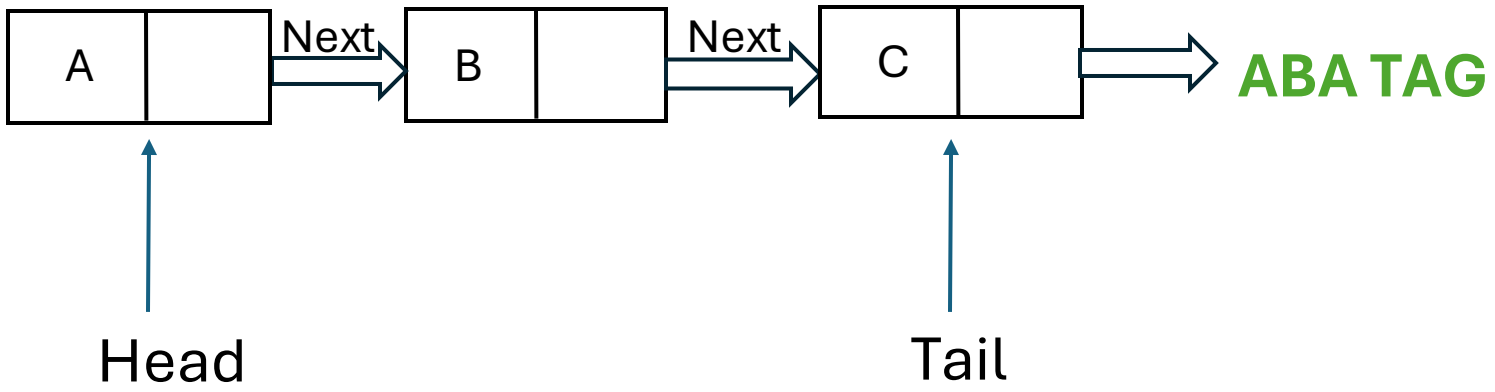
D is unlinked along with its parent C

E moves to the previous position of C

Michael & Scott Queue



Modified Michael & Scott Queue



Last node contains tag instead of ptr

Only Head and Tail contain **both** tag and ptr and need W-CAS

Evaluation Setup

- Platform: AMD EPYC 9754, 128-core, 256 threads, 384 GiB of RAM.
- Payloads: 128-byte (list, tree), 64KiB (queue, list, tree).
- SMR Schemes: Epoch-Based Reclamation (EBR), Hazard Pointers (HP).
- Benchmarks include Queue, Linked List, and Tree variants.
- Measured throughput and memory waste across various configurations.

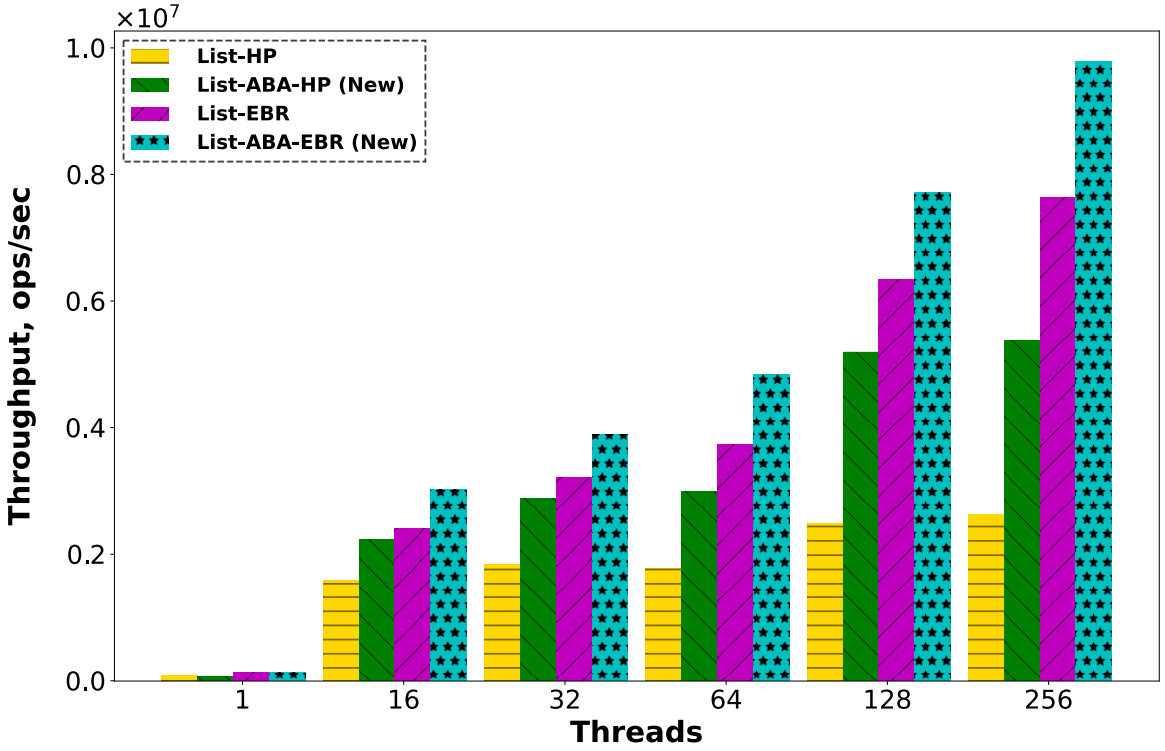
Evaluation Setup

Recycling Percentages

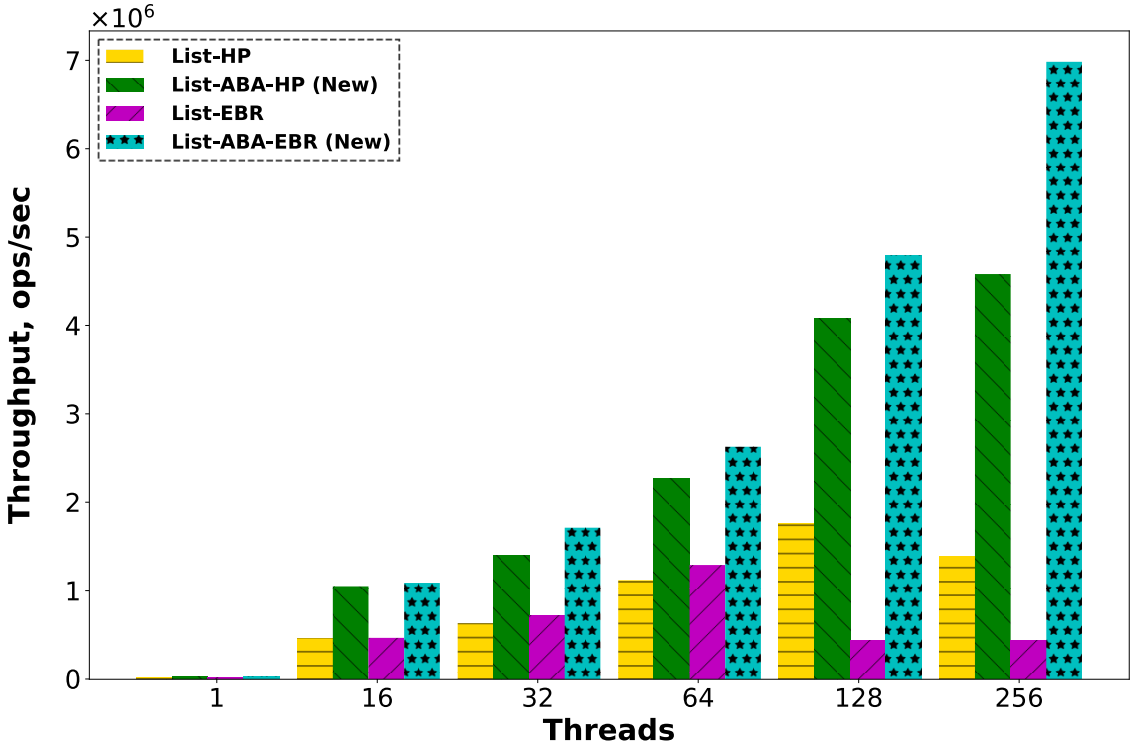
- 50% Recycle:
    - Nodes are moved between structures 50% of the time
    - The other 50% of operations are regular insert and remove

- 90% Recycle:
    - Most operations (90%) move nodes between structures
    - Only 10% involve insertions and deletions

Evaluation: Linked List (Throughput)

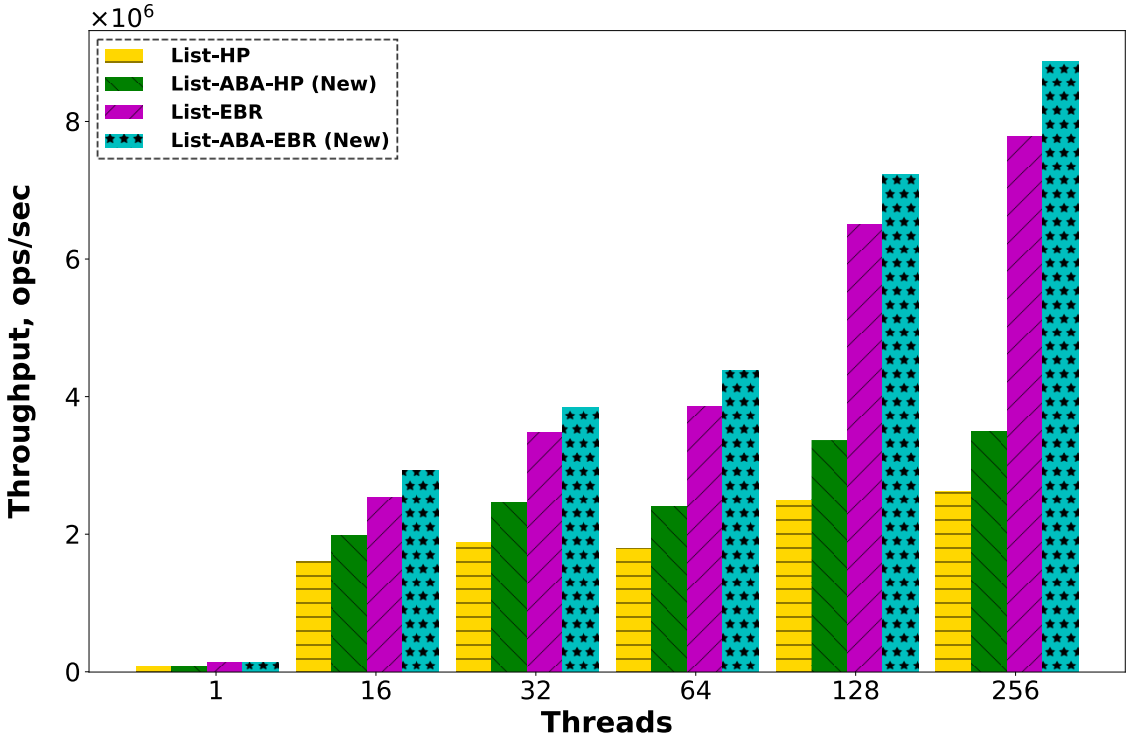


128B Payload, 90% recycle

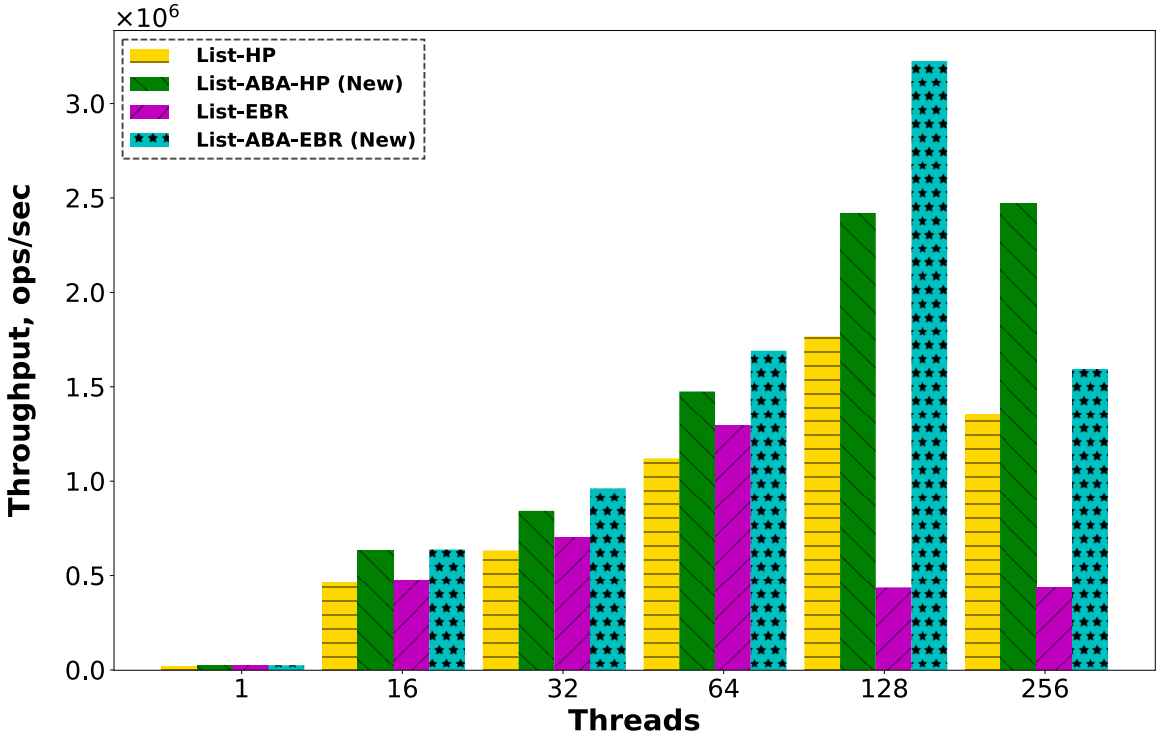


64KiB Payload, 90% recycle

Evaluation: Linked List (Throughput)

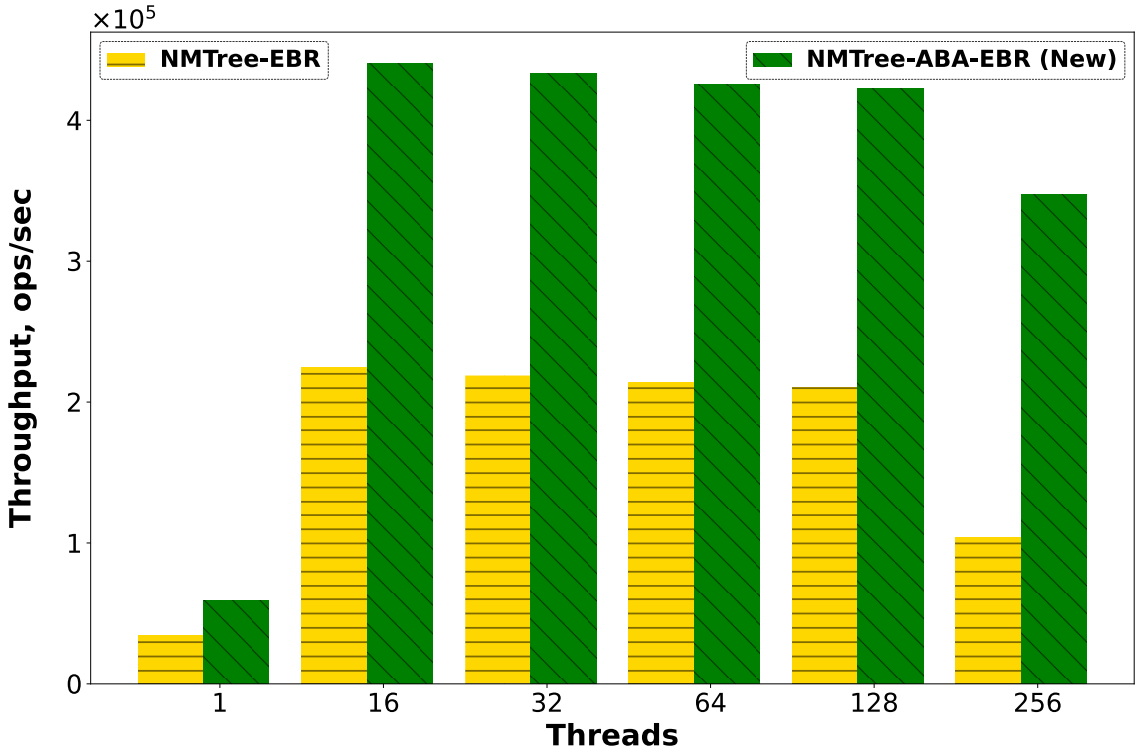


128B Payload, 50% recycle

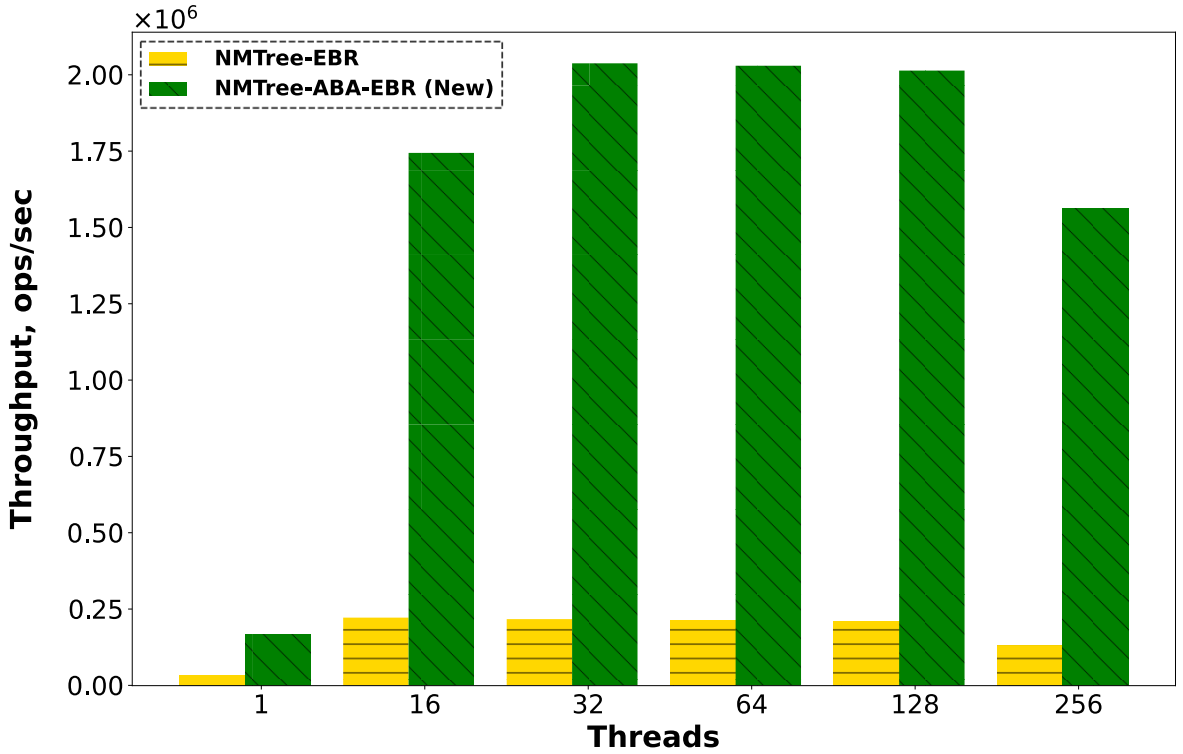


64KiB Payload, 50% recycle

Evaluation: Natarajan-Mittal Tree (Throughput)



64KiB Payload, 50% recycle

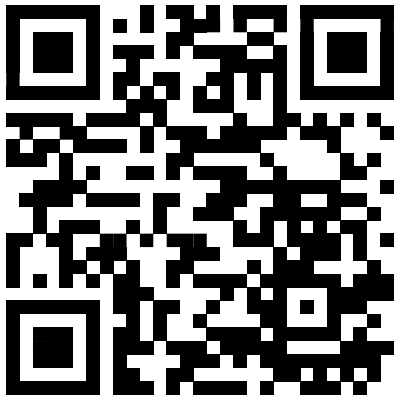


64KiB Payload, 90% recycle

Availability

- Code is open-source and available at:

  https://github.com/rusnikola/rrr-smr



Availability

- Code is open-source and available at:

  https://github.com/rusnikola/rrr-smr

THANK YOU!

QUESTIONS?