A Skiplist-based Concurrent Priority Queue with Minimal Memory Contention

Jonatan Lindén and Bengt Jonsson

Uppsala University, Sweden

December 18, 2013

Contributions

Motivation: Improve performance of concurrent Discrete Event Simulator.
Outcome: New lock-free skiplist-based priority queue.

- New representation for logically deleted nodes.
- Minimizes the contention.
- Improved performance over existing algorithms by 30 – 80% on multiprocessors.
- Linearizable.

Outline

Priority Queues

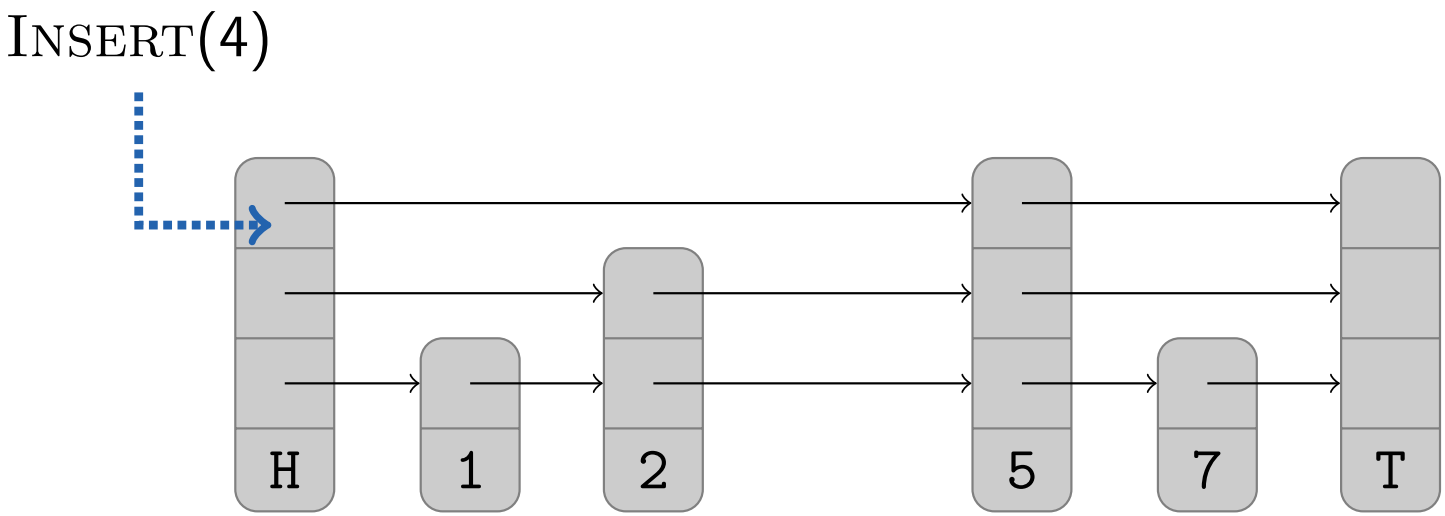
- Priority Queue - A set of (key,value) pairs with two operations:
  - Insert(*key*, *value*)
  - DeleteMin()
- Applications: Discrete Event Simulation, Numerical algorithms.

Priority Queues

- Priority Queue - A set of (key,value) pairs with two operations:
  - INSERT(*key*, *value*)
  - DELETEMIN()
- Applications: Discrete Event Simulation, Numerical algorithms.
- Implementations:
  - Traditionally, implemented on top of heaps or tree data structures.
  - Skiplists [Pugh:1990] have been used for several parallel implementations.

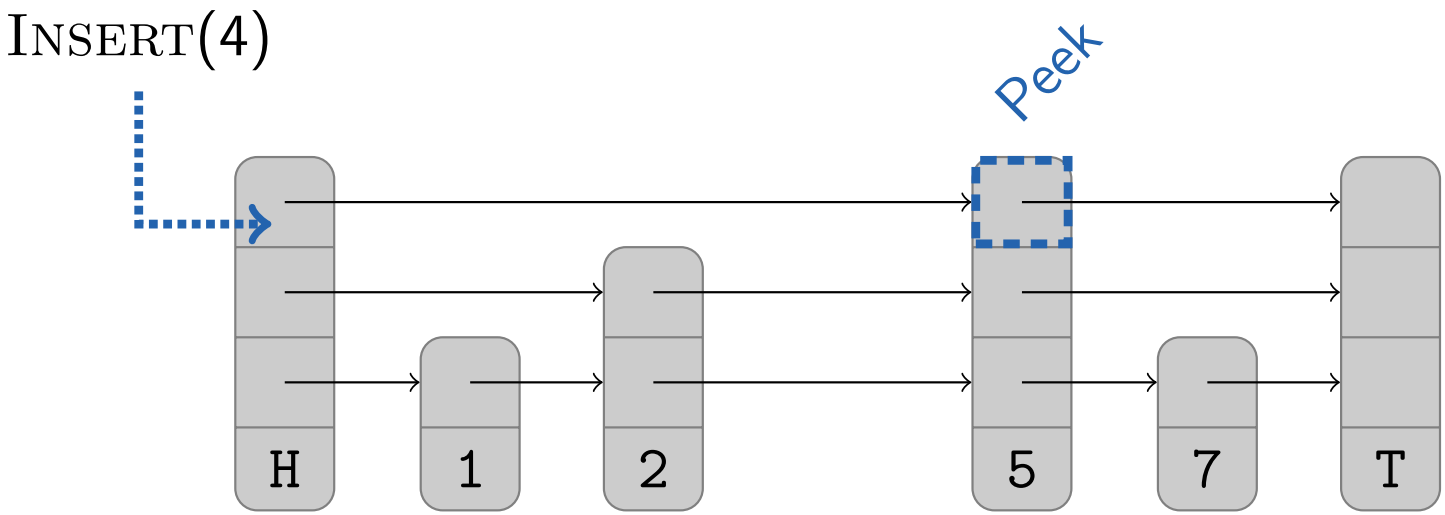
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



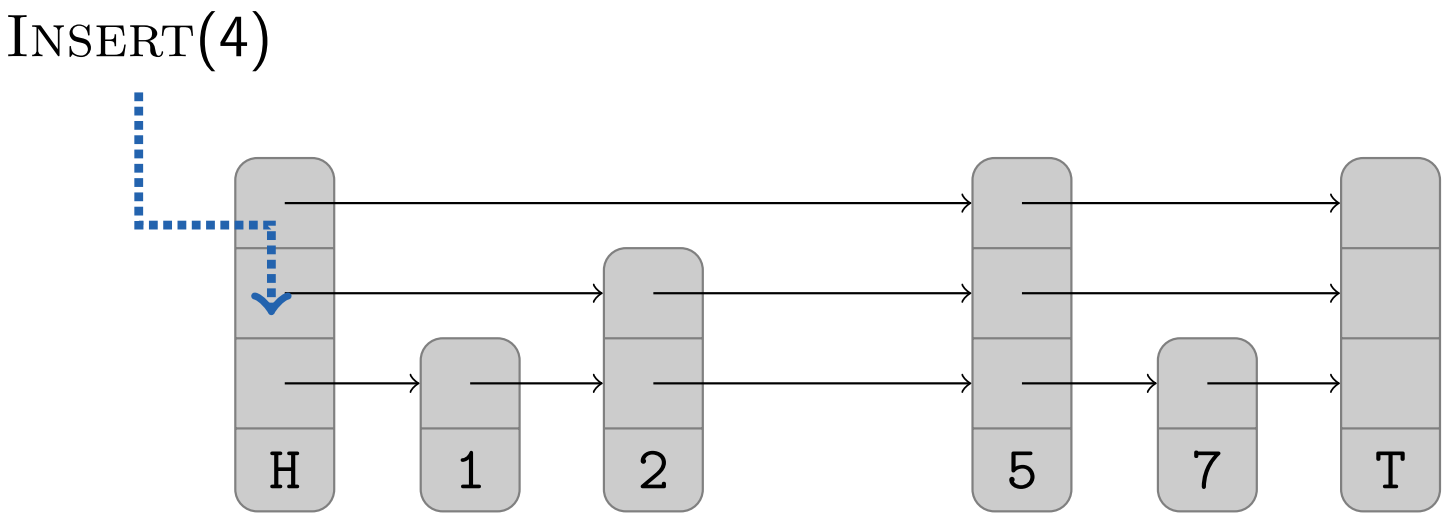
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



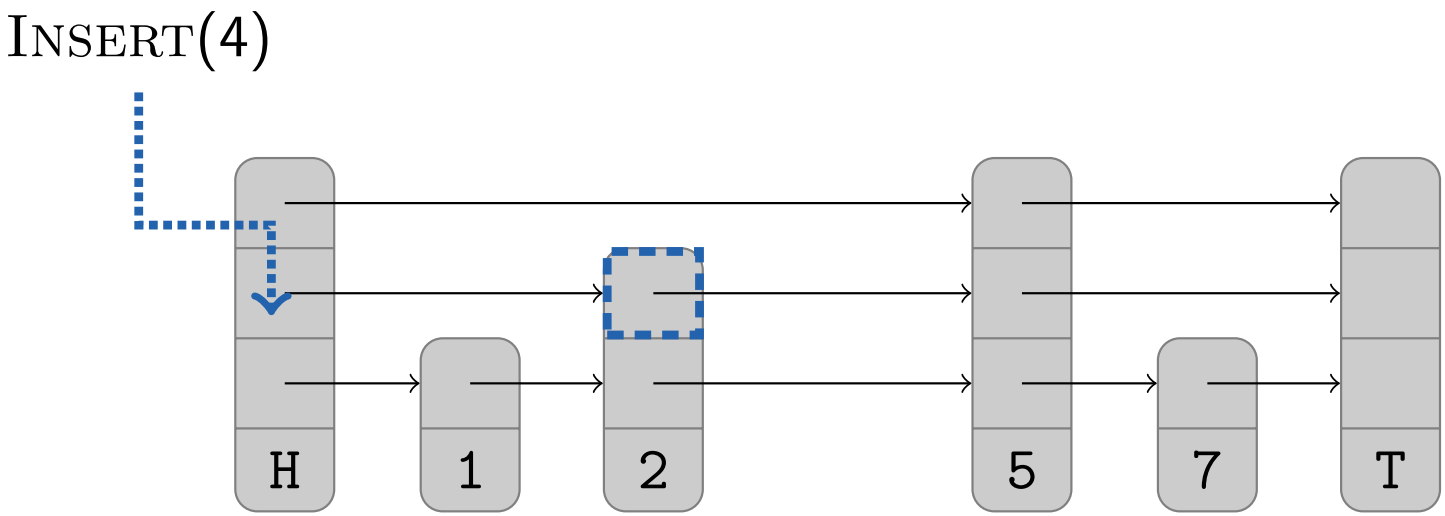
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



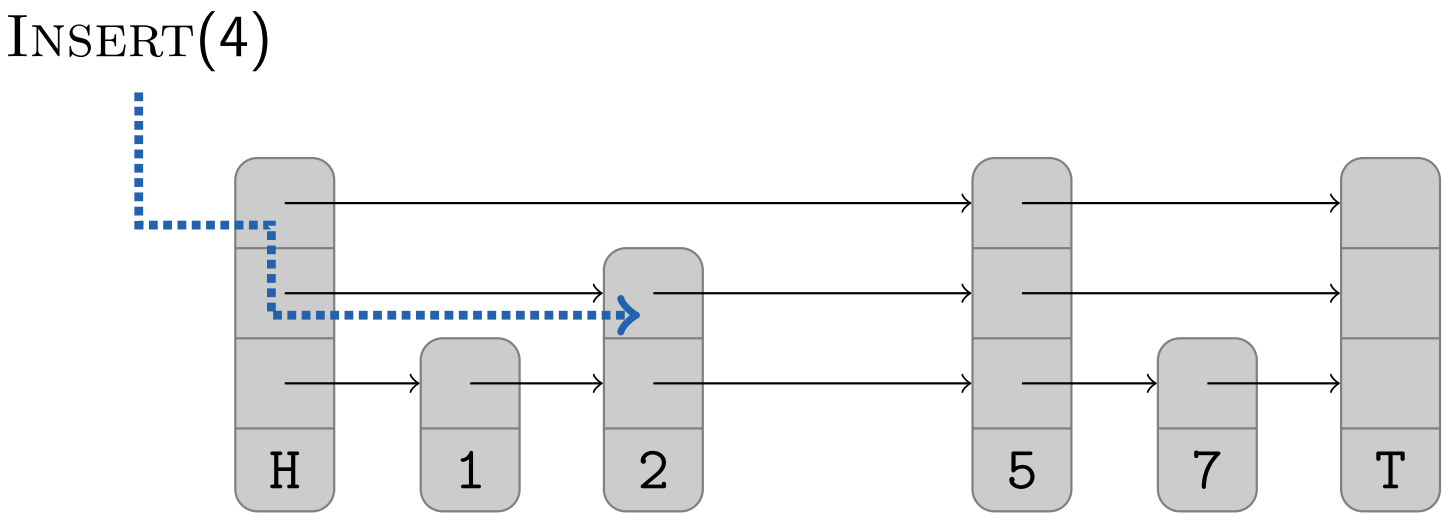
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



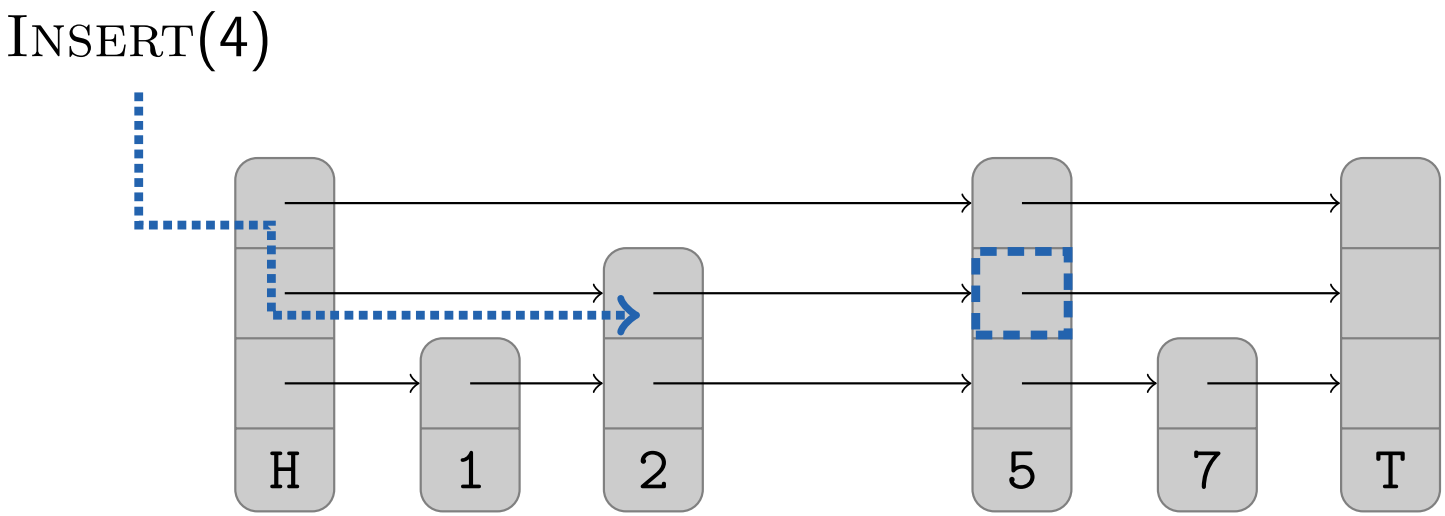
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



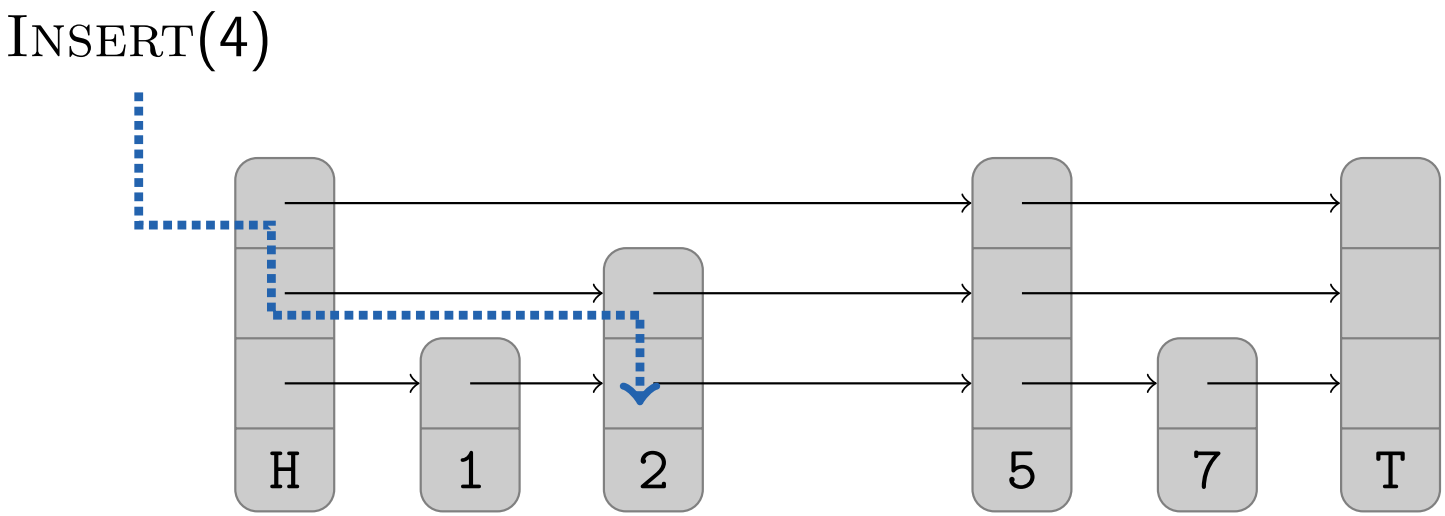
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



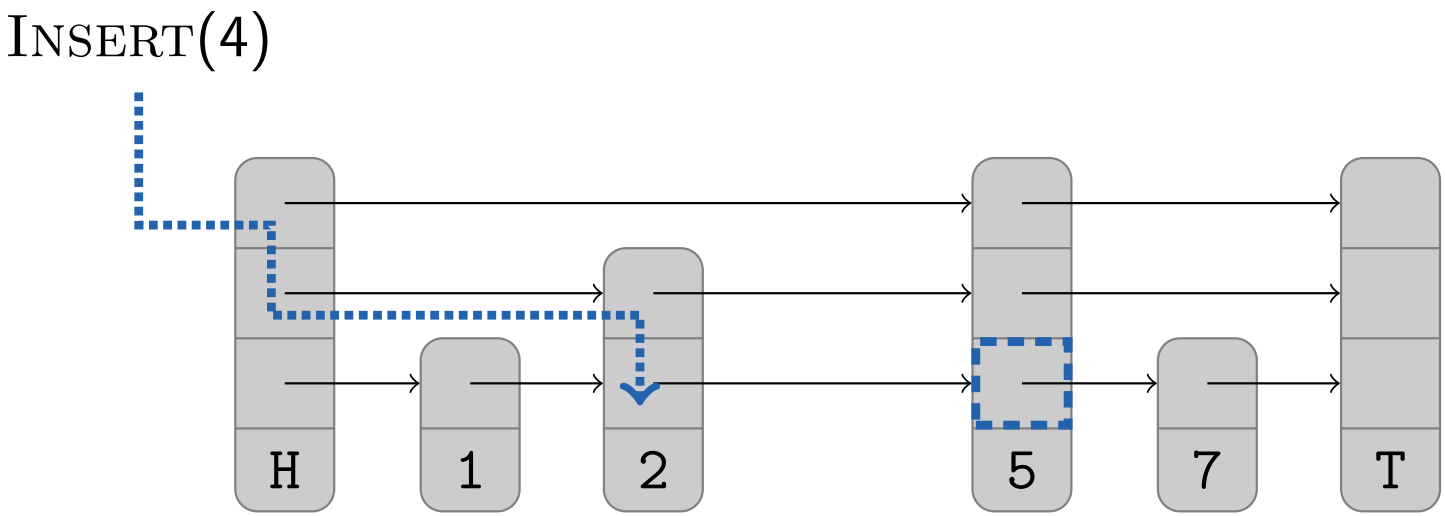
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



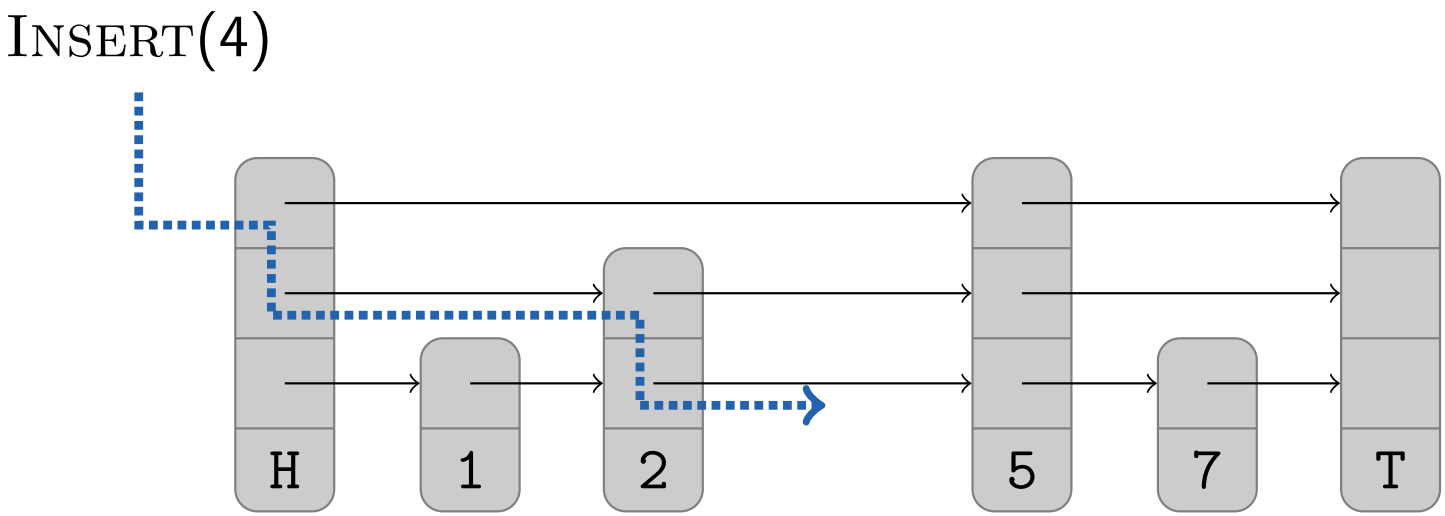
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



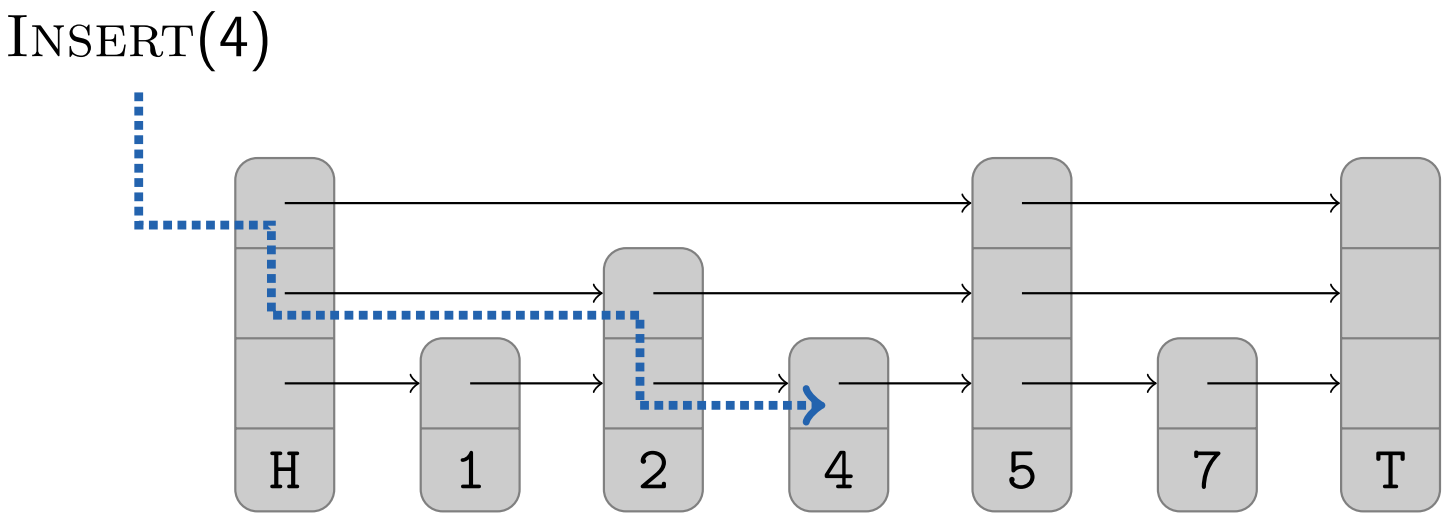
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



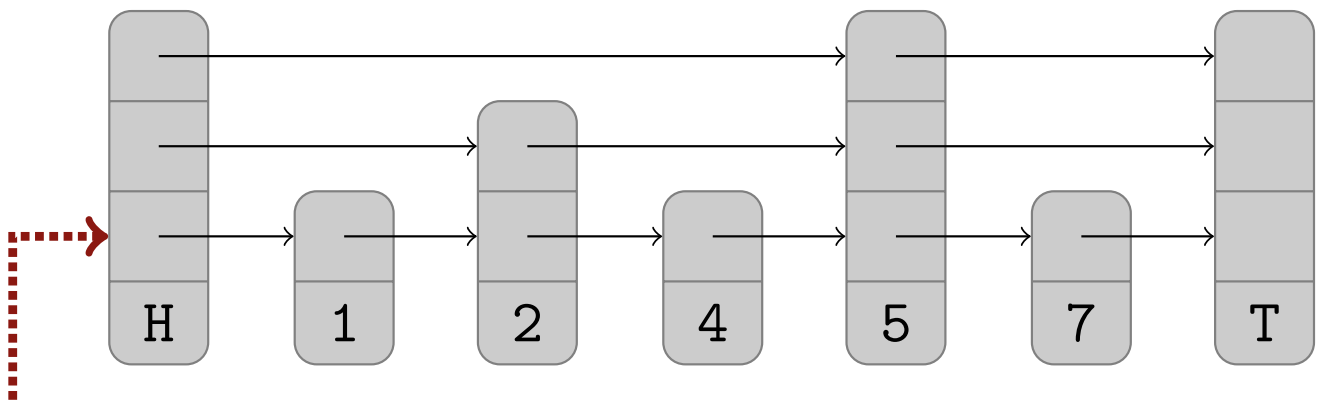
Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time



Skiplist

- layered linked list
- lowest-level list defines logical state, ordered
- higher-level lists are shortcuts
- probabilistic guarantee of logarithmic search time
- Smallest element at the beginning of the lowest-level list.



DeleteMin entry point

Concurrent skiplist-based priority queues

- Skiplists are easy to make concurrent
- Skiplists scale well when concurrent threads access different parts of the structure.

Concurrent skiplist-based priority queues

- Skiplists are easy to make concurrent
- Skiplists scale well when concurrent threads access different parts of the structure.

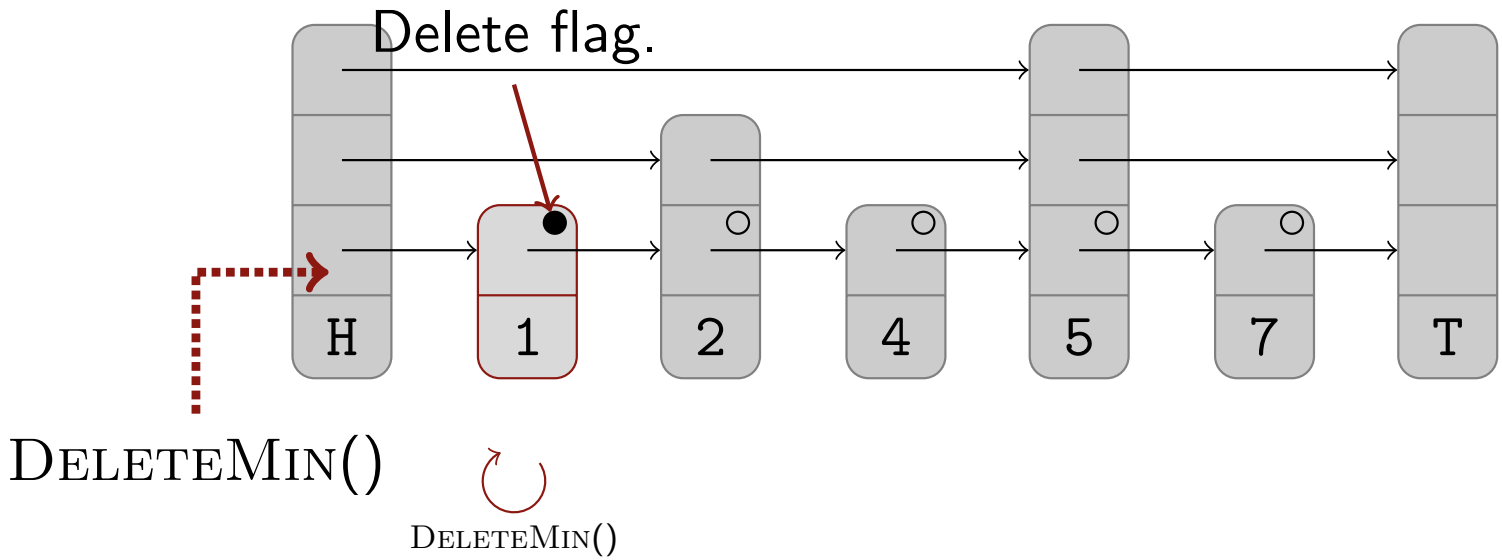
Concurrent skiplist-based priority queues

- Skiplists are easy to make concurrent
- Skiplists scale well when concurrent threads access different parts of the structure.
- Bottleneck: concurrent DeleteMin operations in priority queues try to remove the same element.

Standard solution

Standard solution to increase concurrency:

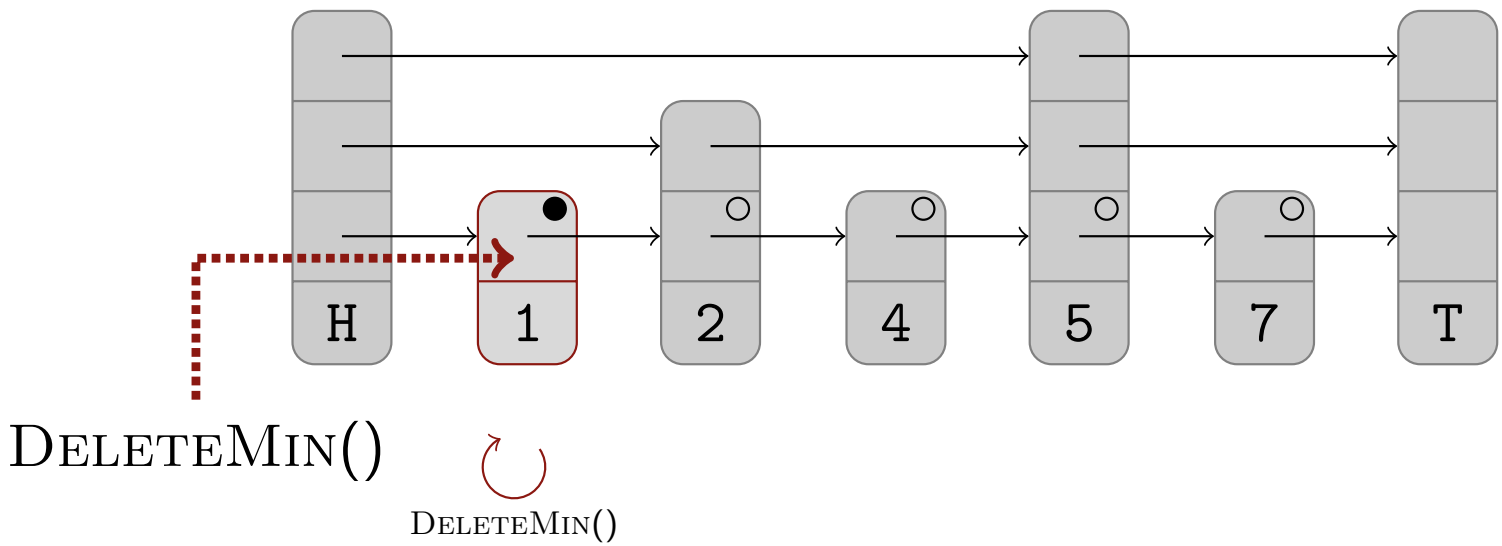
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

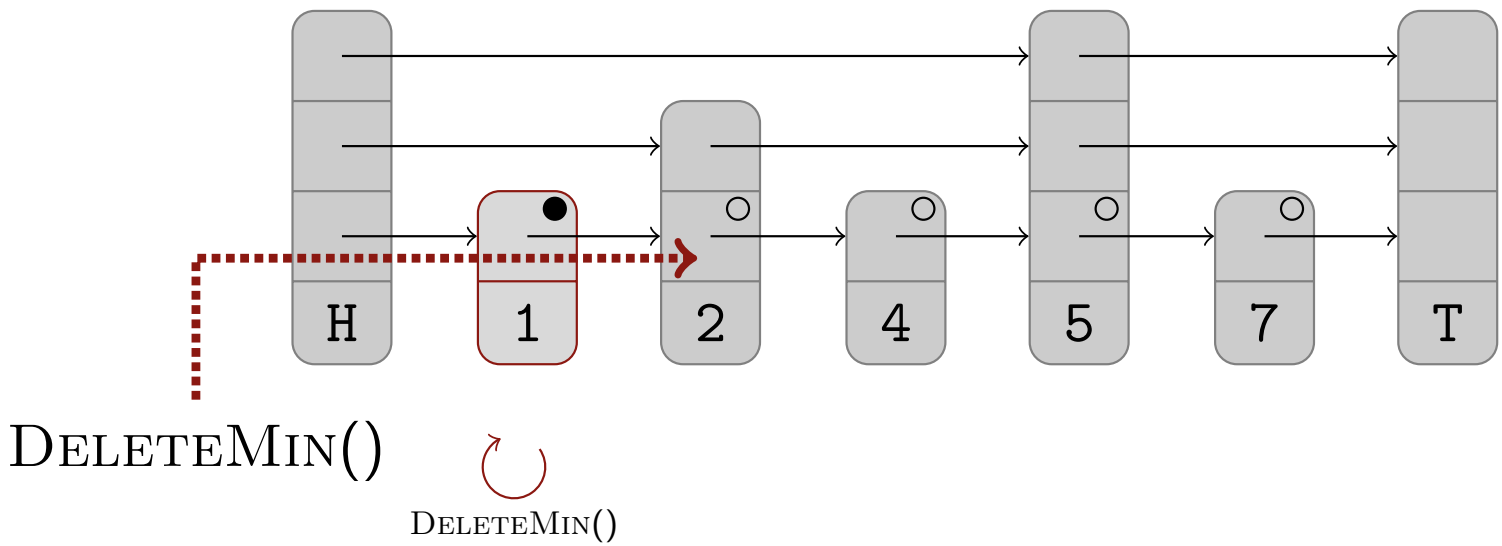
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

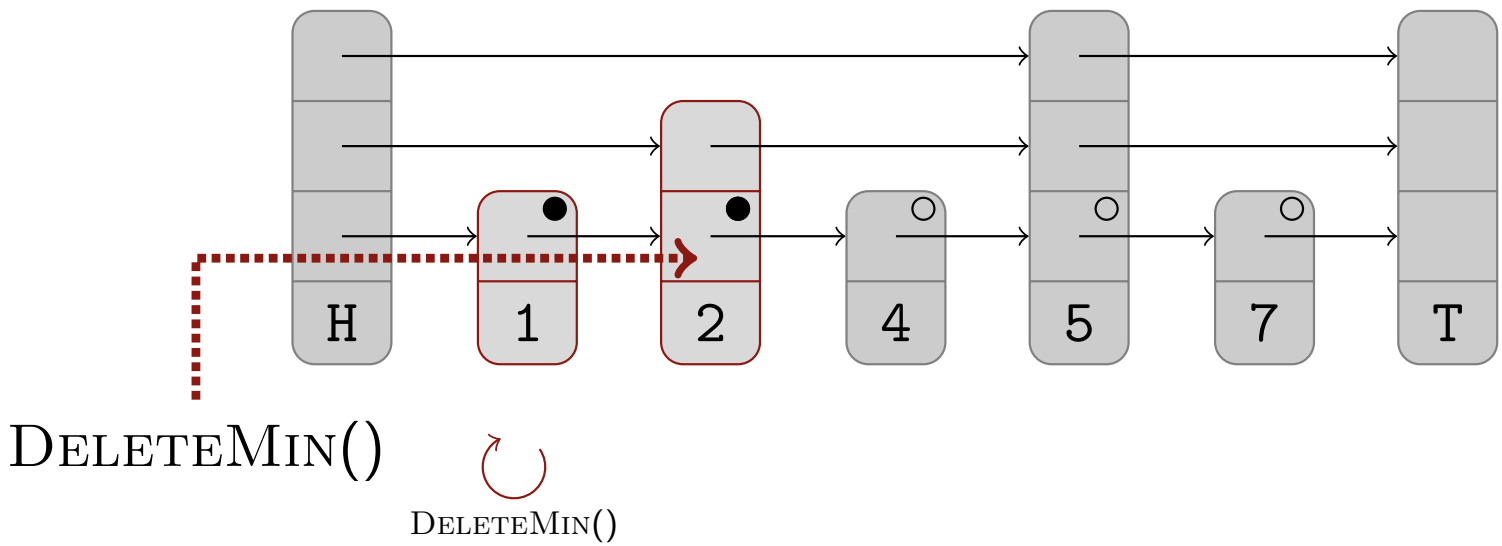
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

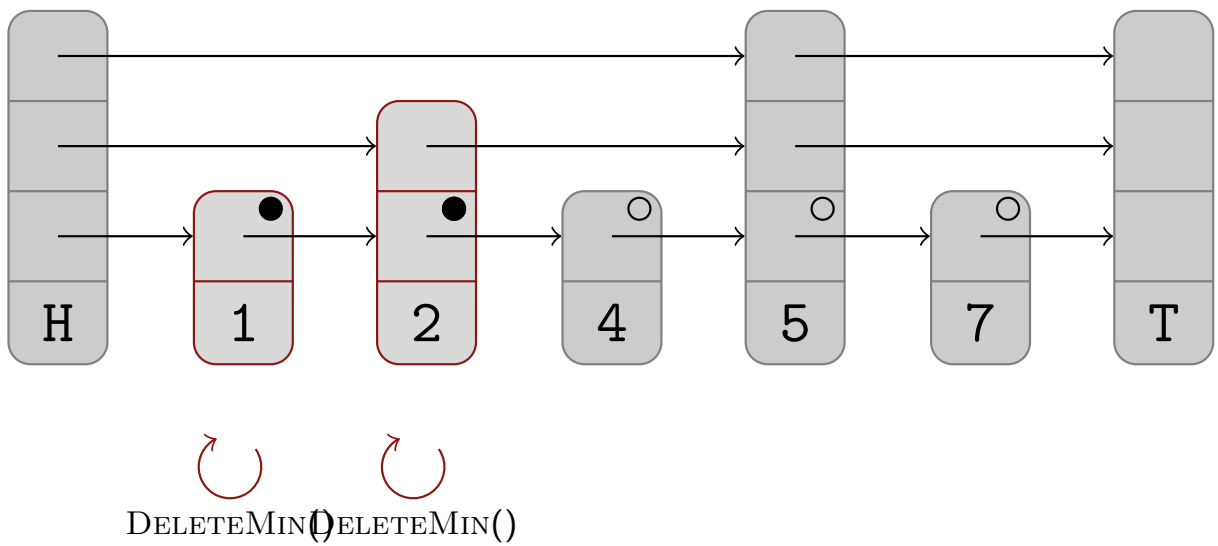
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

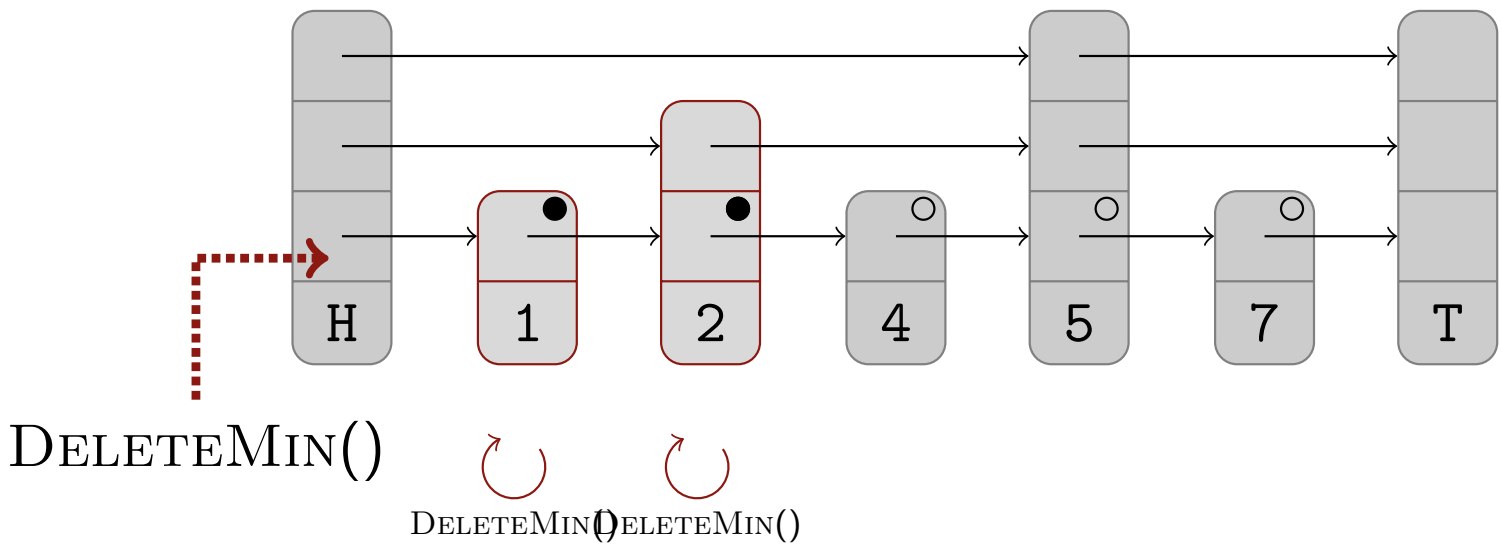
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

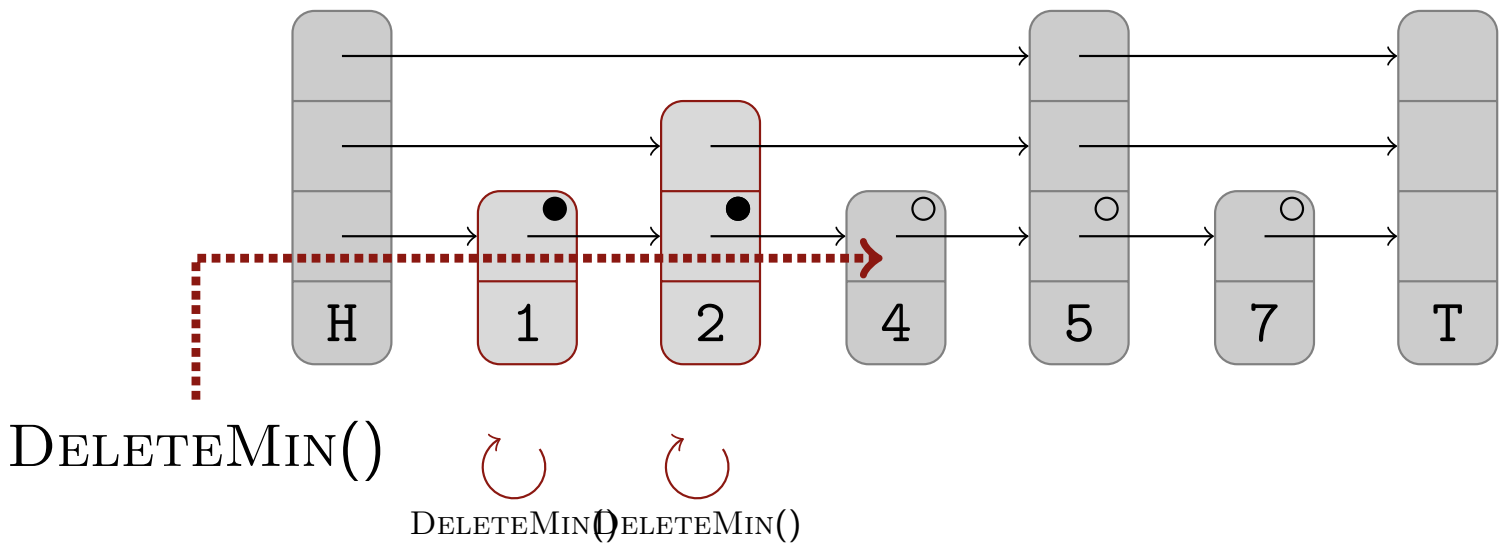
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

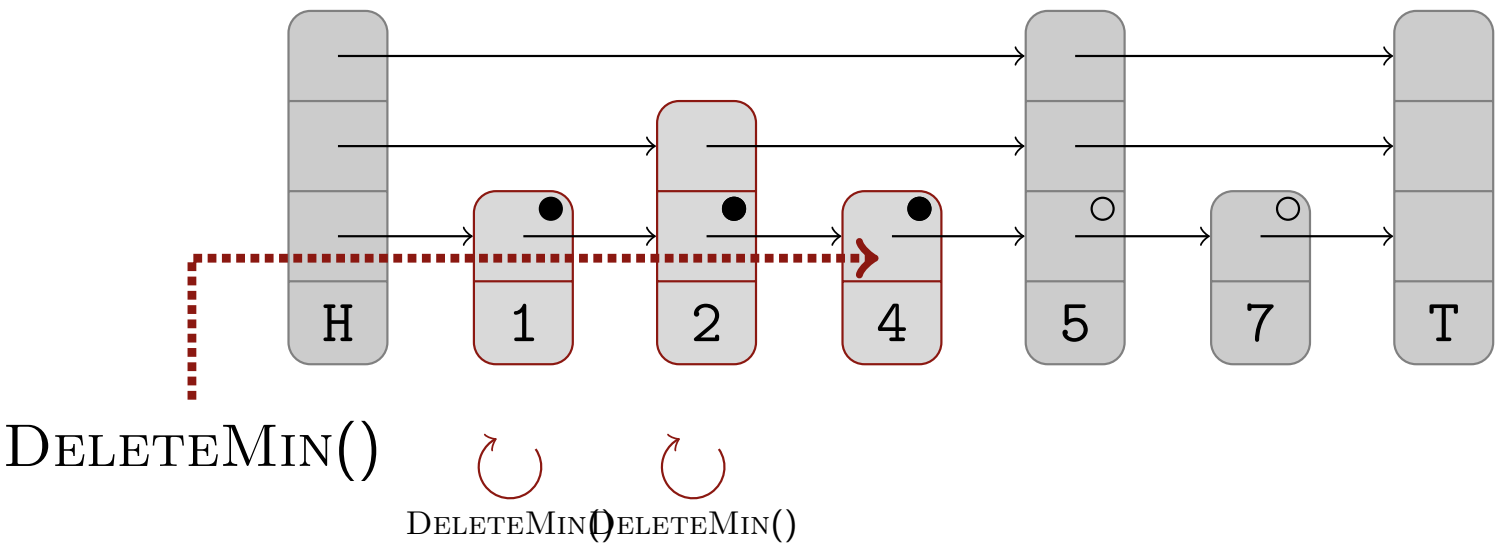
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.



Standard solution

Standard solution to increase concurrency:

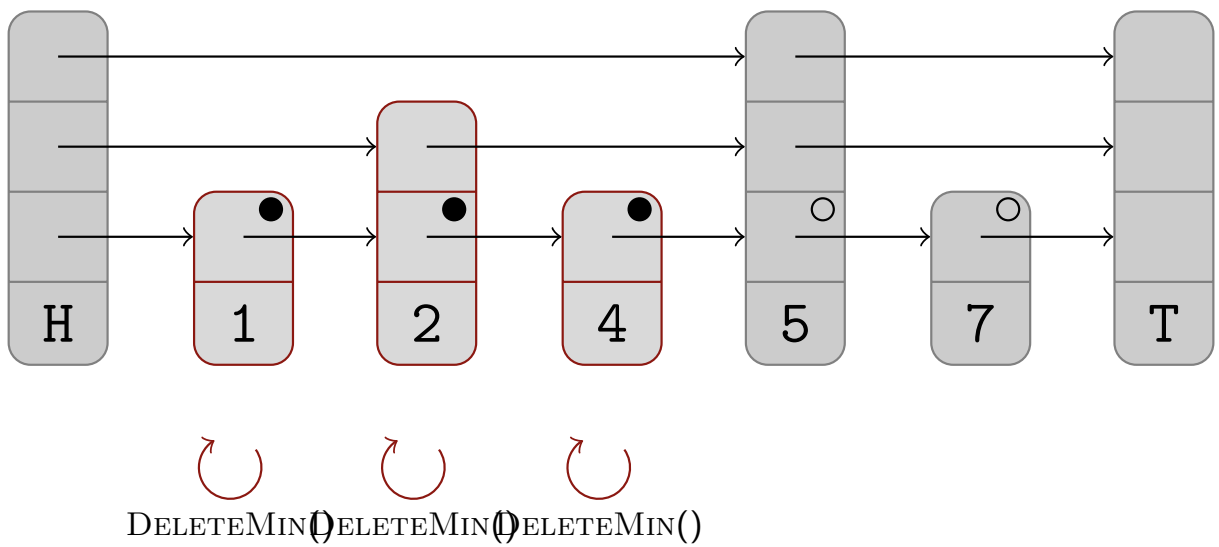
- *logical deletion* by setting a delete flag.
- *physical deletion* unlinks the node after the logical deletion has succeeded.
- Lock-free:
  - Perform all writes using Compare-and-Swap (CAS)
  - Colocate delete flag together with each next pointer, e.g., in the lowest order bit [Harris 2001], to make physical deletion safe.



Contention

Bottleneck: CAS in DeleteMin

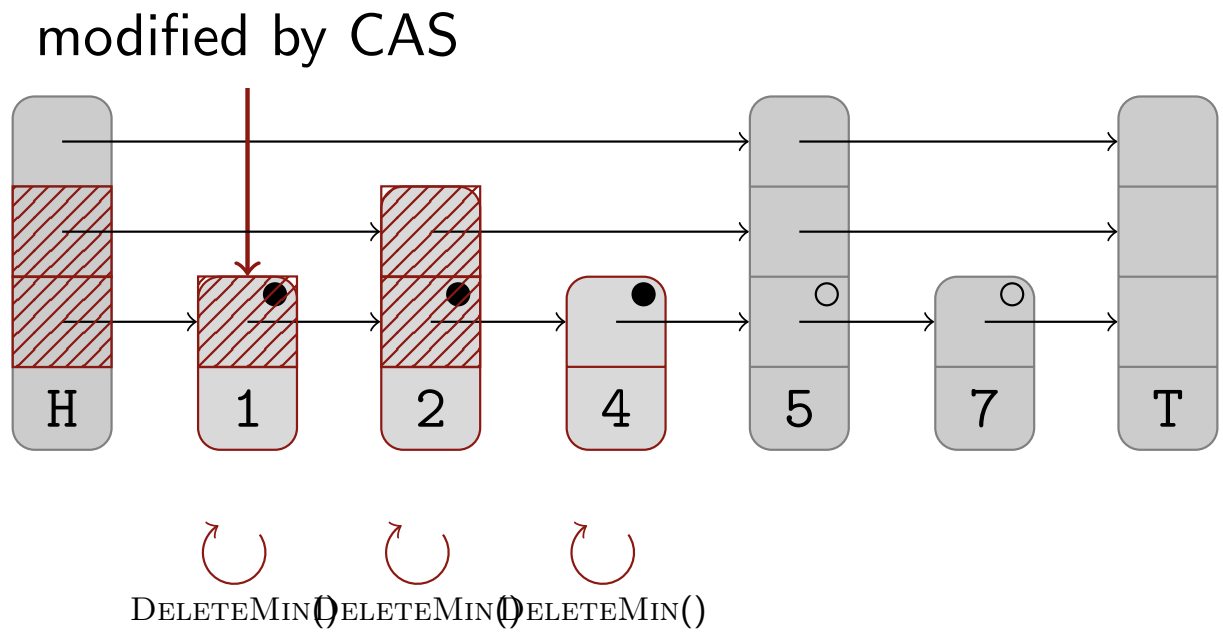
- Several types of contention:
  - (i) Multiple CASes compete



Contention

Bottleneck: CAS in DeleteMin

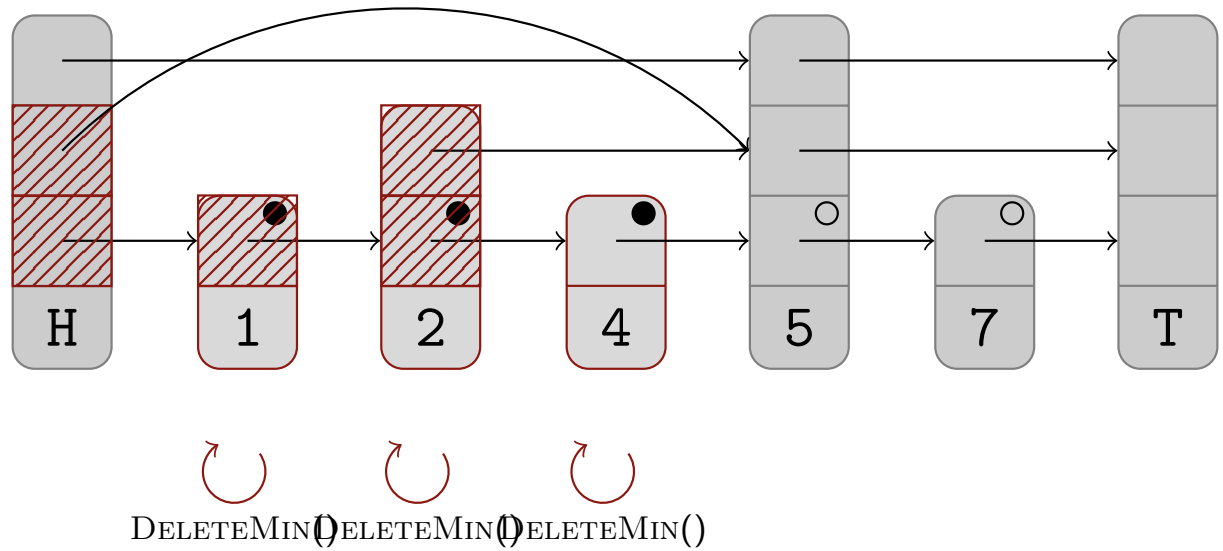
- Several types of contention:
  - (i) Multiple CASes compete



Contention

Bottleneck: CAS in DeleteMin

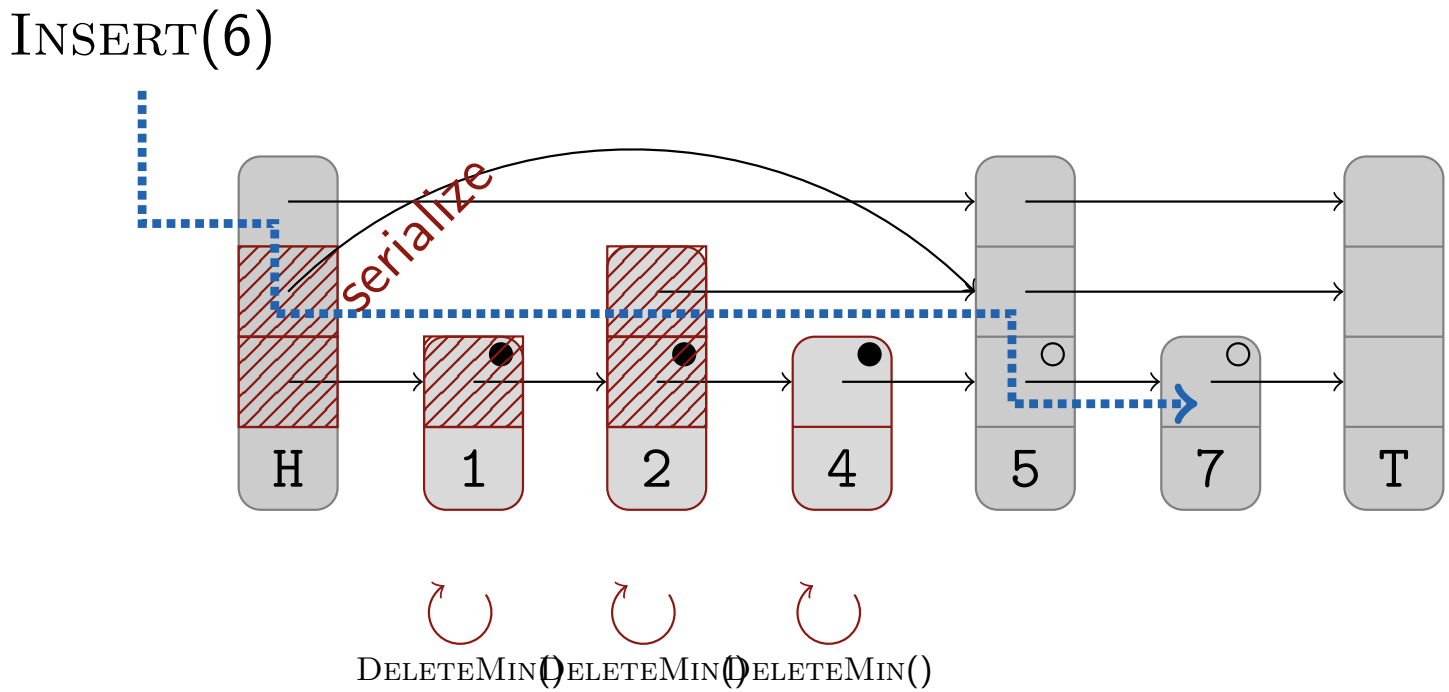
- Several types of contention:
  - (i) Multiple CASes compete
  - (ii) Updates must be propagated to reads



Contention

Bottleneck: CAS in DeleteMin

- Several types of contention:
  - (i) Multiple CASes compete
  - (ii) Updates must be propagated to reads

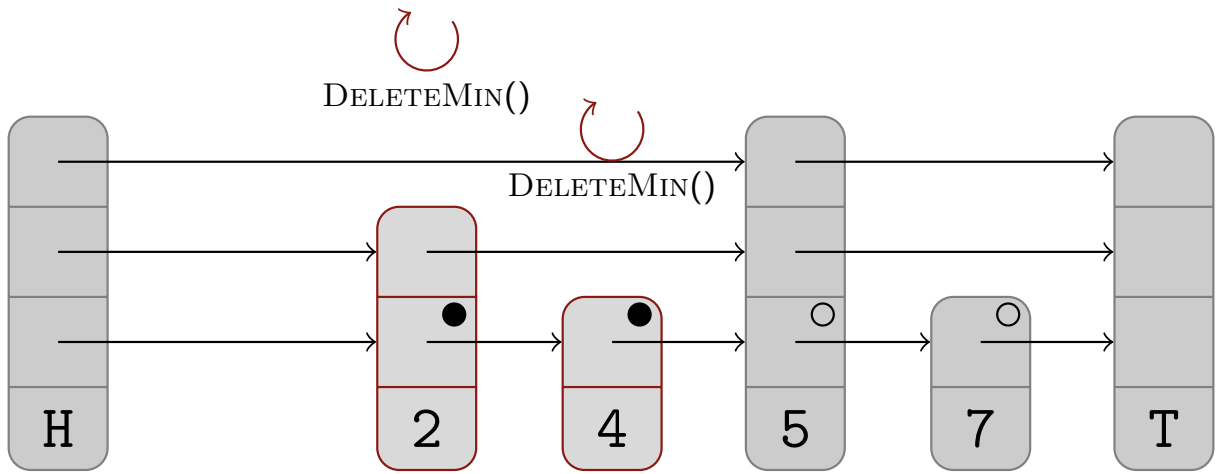


Our algorithm

Our solution

Key idea: No physical deletion after logical deletion!

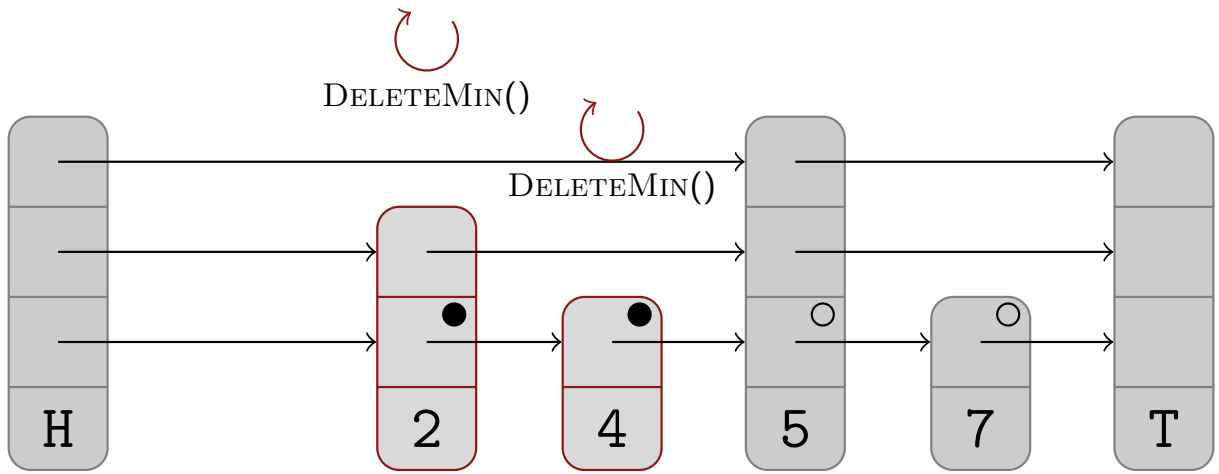
- Instead, delete nodes in batches.



Our solution

Key idea: No physical deletion after logical deletion!

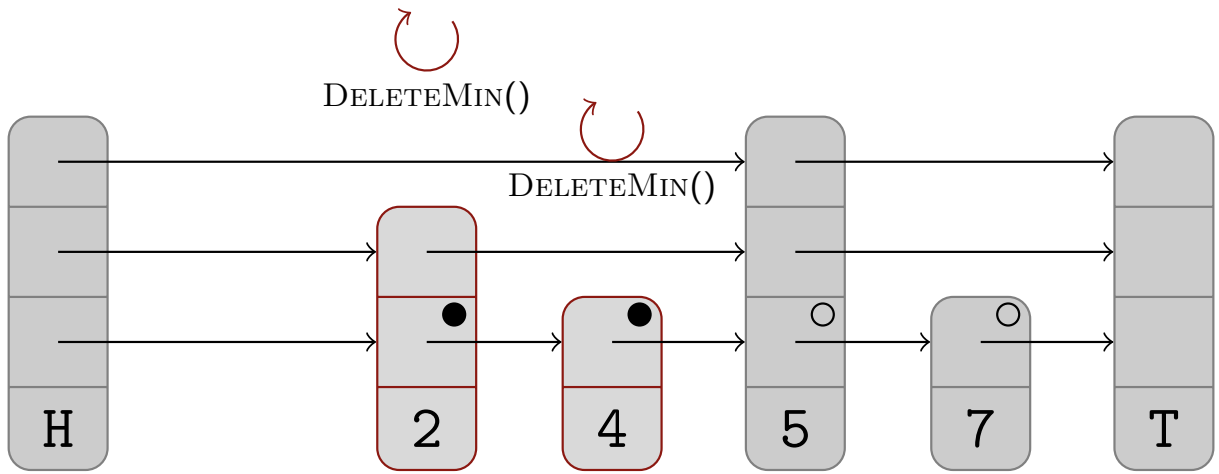
- Instead, delete nodes in batches.
- By updating the pointers in head node.



Our solution

Key idea: No physical deletion after logical deletion!

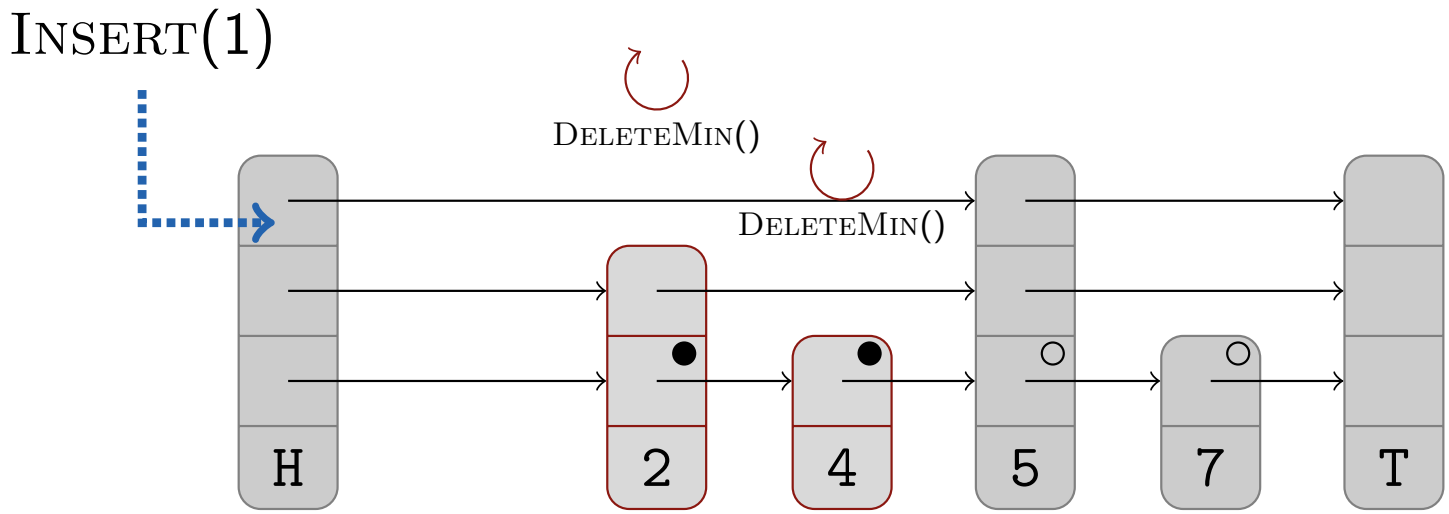
- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.



Our solution

Key idea: No physical deletion after logical deletion!

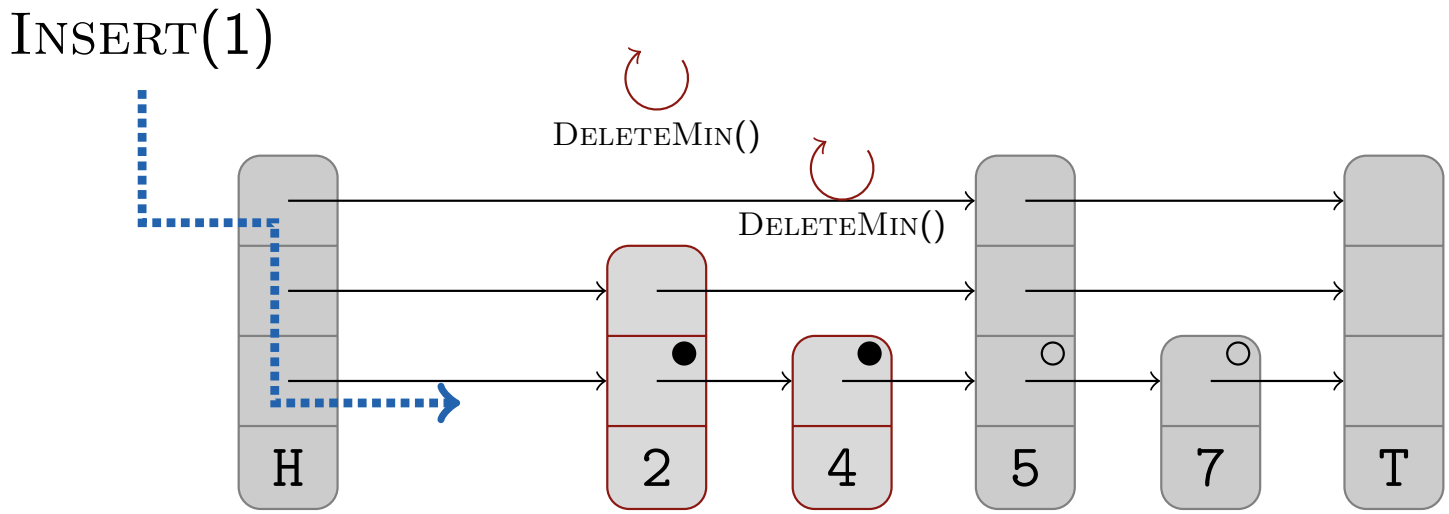
- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.



Our solution

Key idea: No physical deletion after logical deletion!

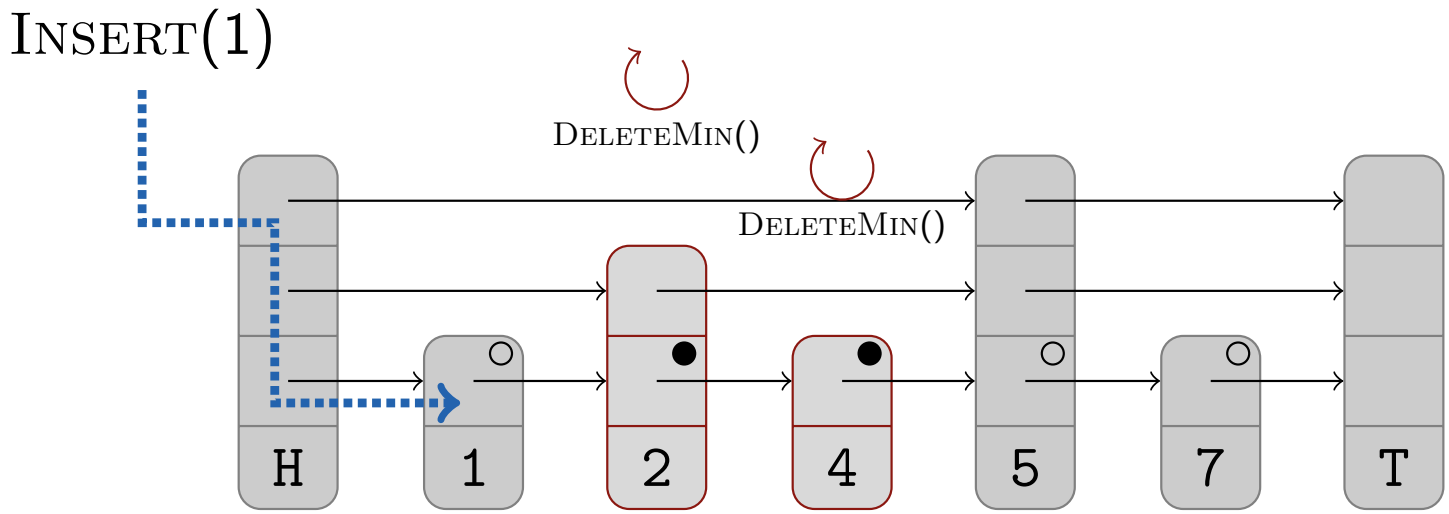
- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.



Our solution

Key idea: No physical deletion after logical deletion!

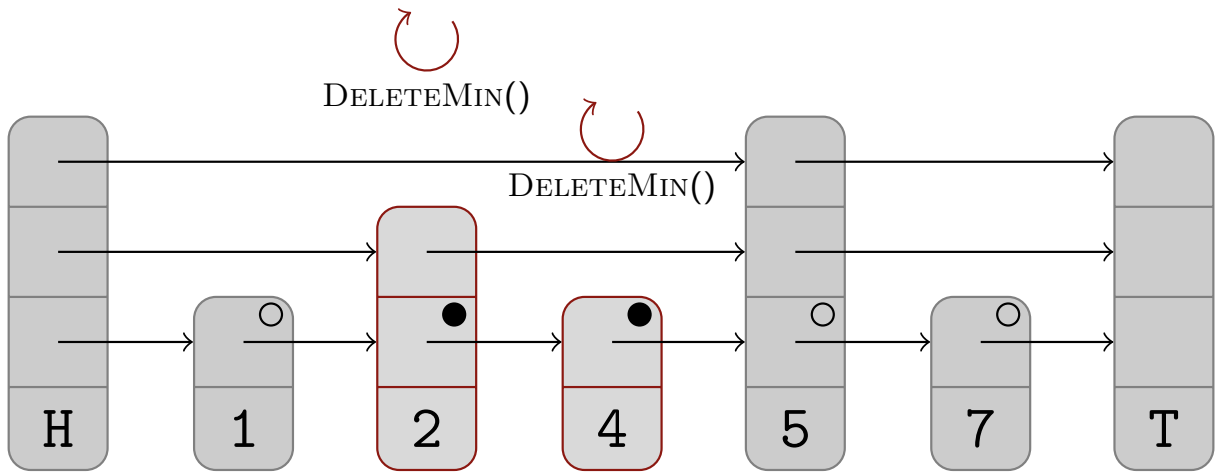
- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.



Our solution

Key idea: No physical deletion after logical deletion!

- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.

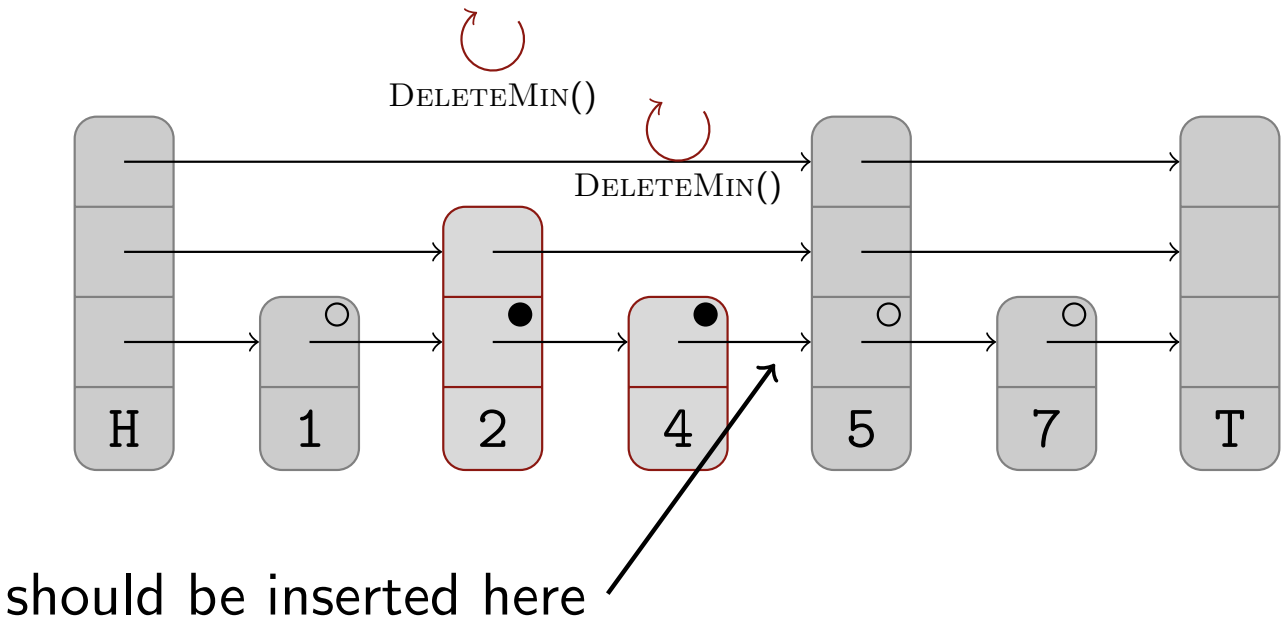


Does not work!

Our solution

Key idea: No physical deletion after logical deletion!

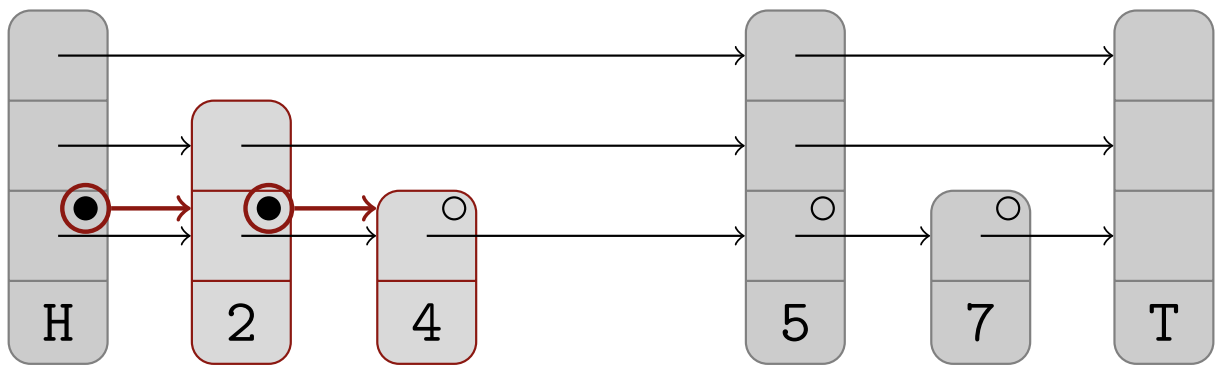
- Instead, delete nodes in batches.
- By updating the pointers in head node.
- Requires that logically deleted nodes form a prefix.



Our solution

To guarantee prefix property,

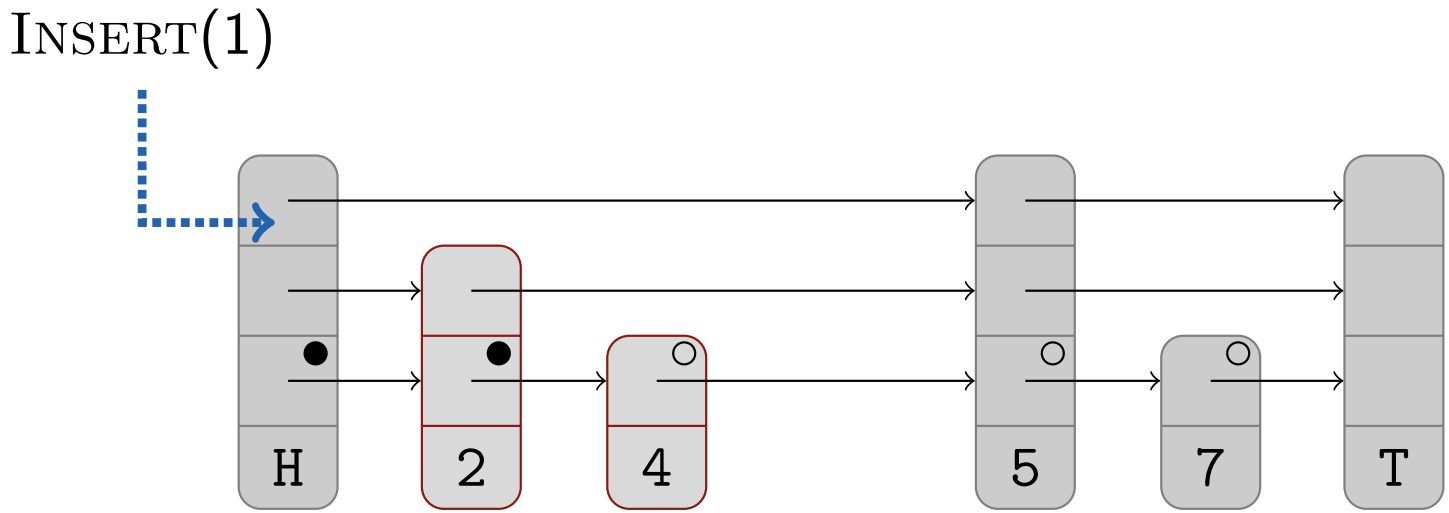
- Store delete flag together with the *predecessor's* next pointer.



Our solution

To guarantee prefix property,

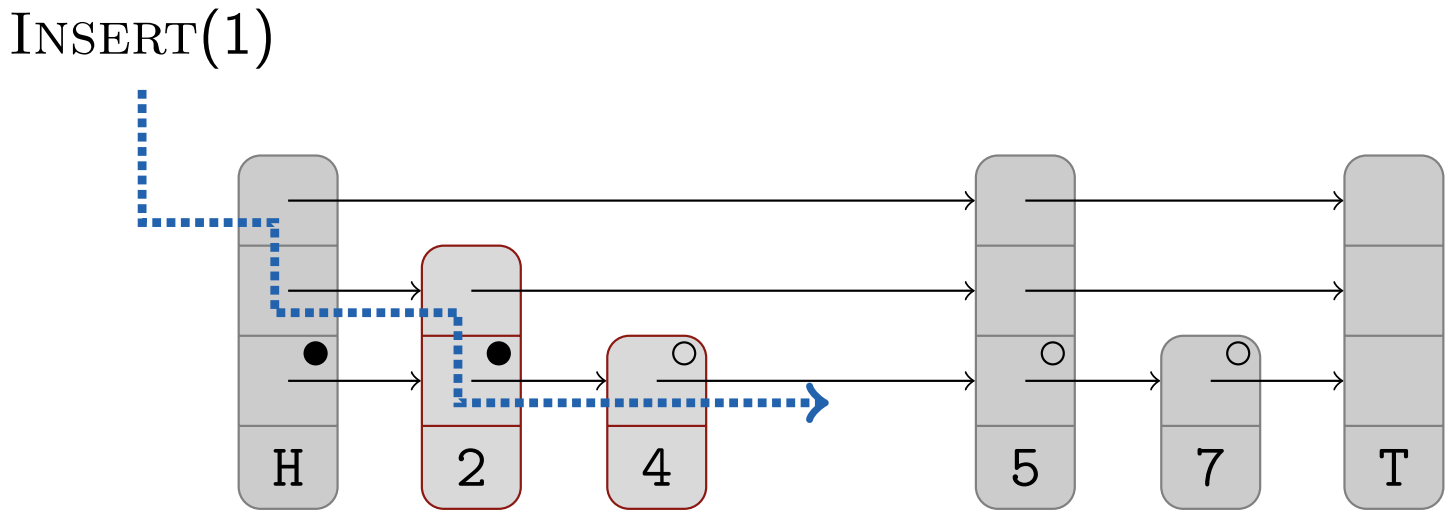
- Store delete flag together with the *predecessor's* next pointer.



Our solution

To guarantee prefix property,

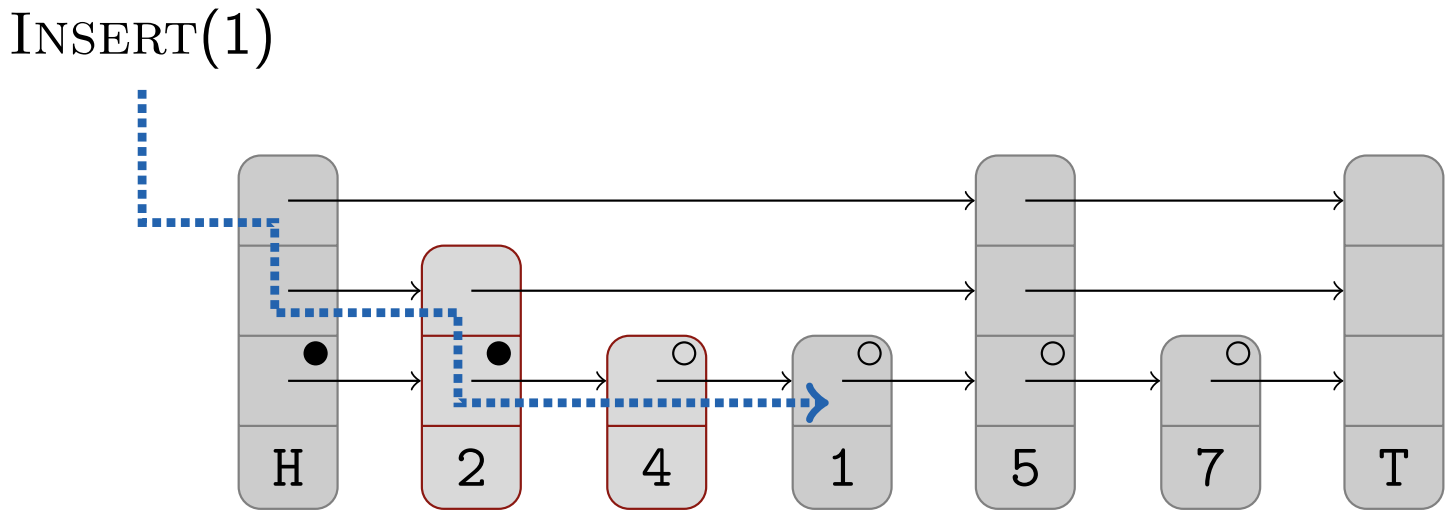
- Store delete flag together with the *predecessor's* next pointer.



Our solution

To guarantee prefix property,

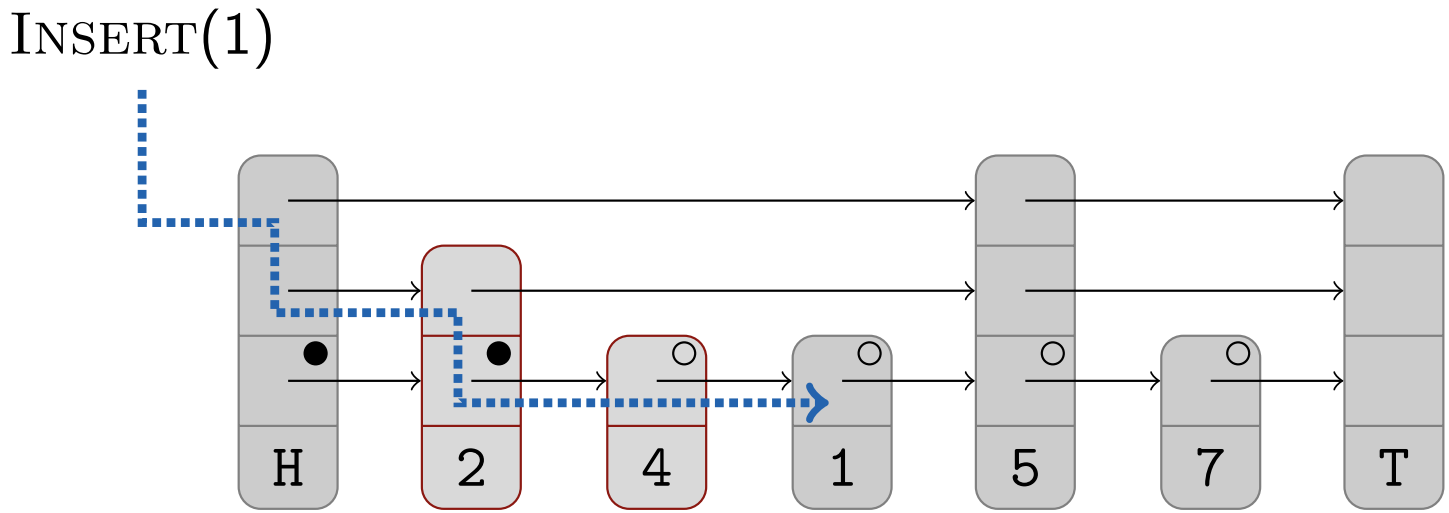
- Store delete flag together with the *predecessor's* next pointer.



Our solution

To guarantee prefix property,

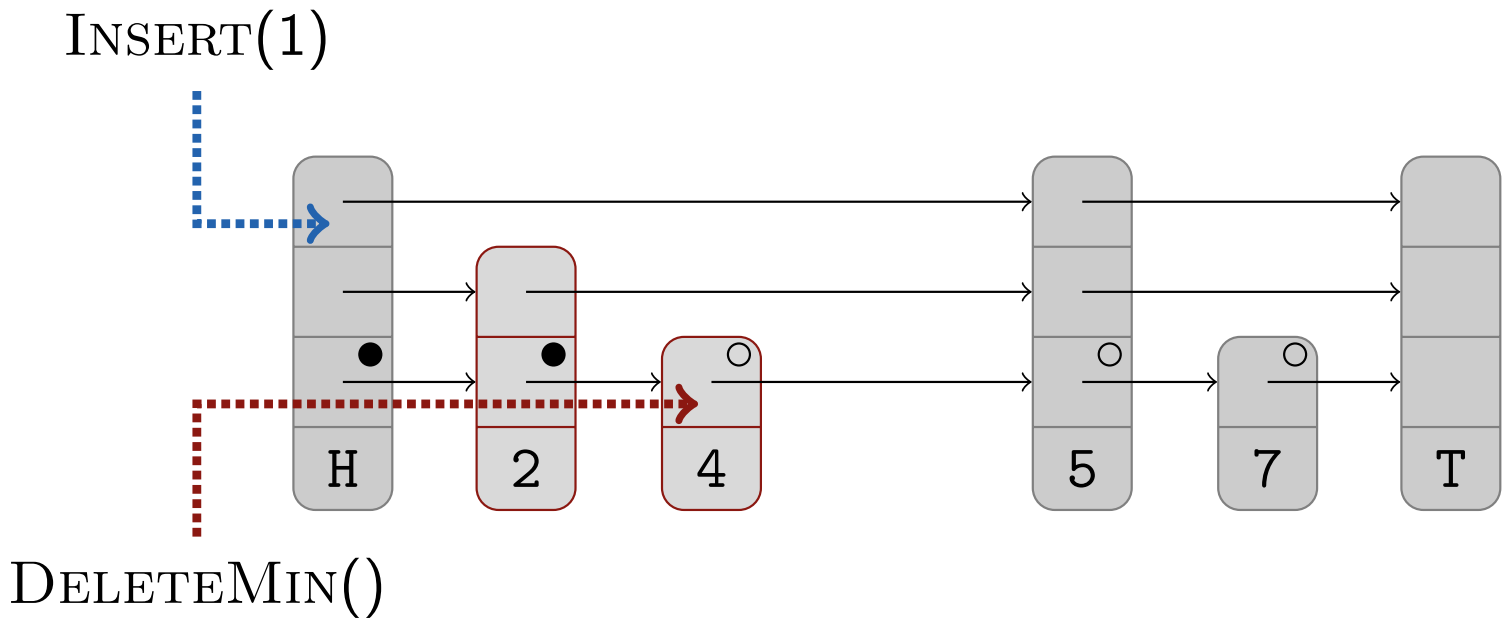
- Store delete flag together with the *predecessor's* next pointer.



Still a prefix!

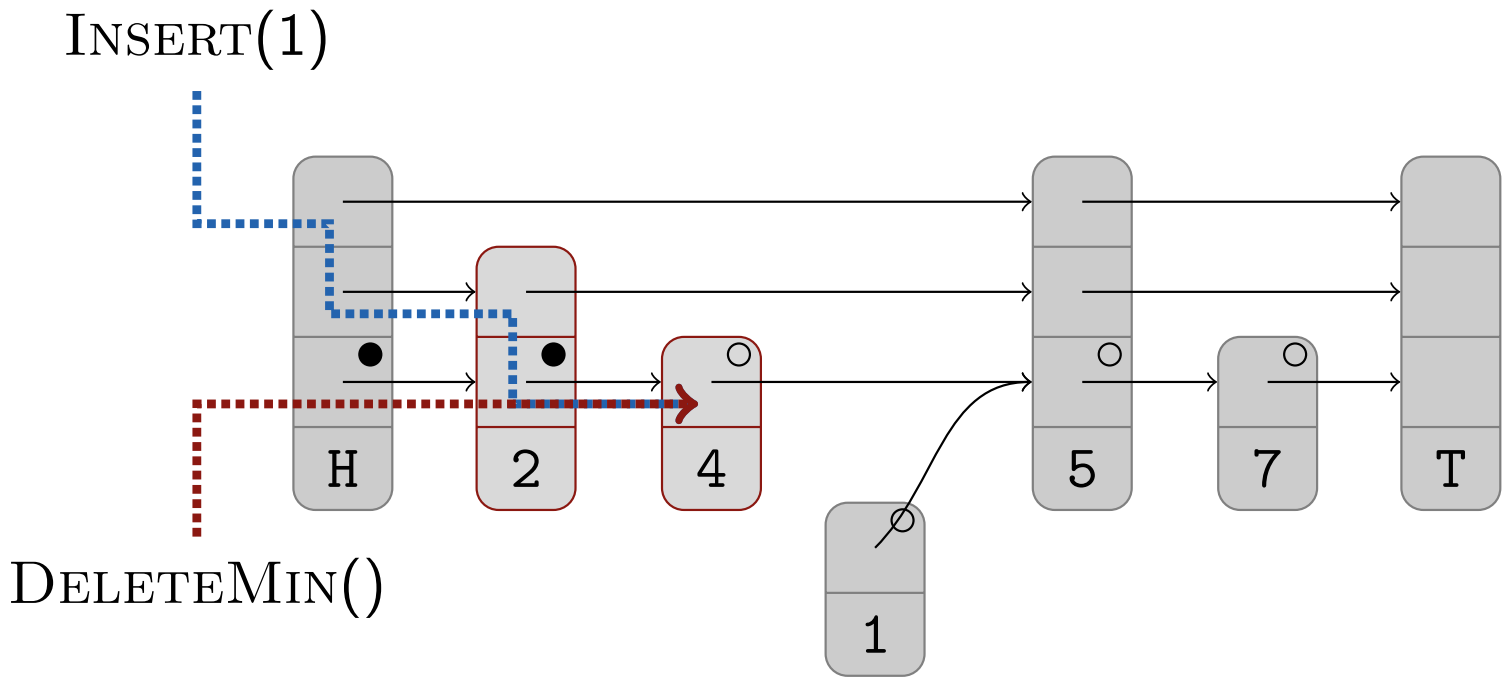
Our solution

Resolving conflicts between INSERT and DELETEMIN.



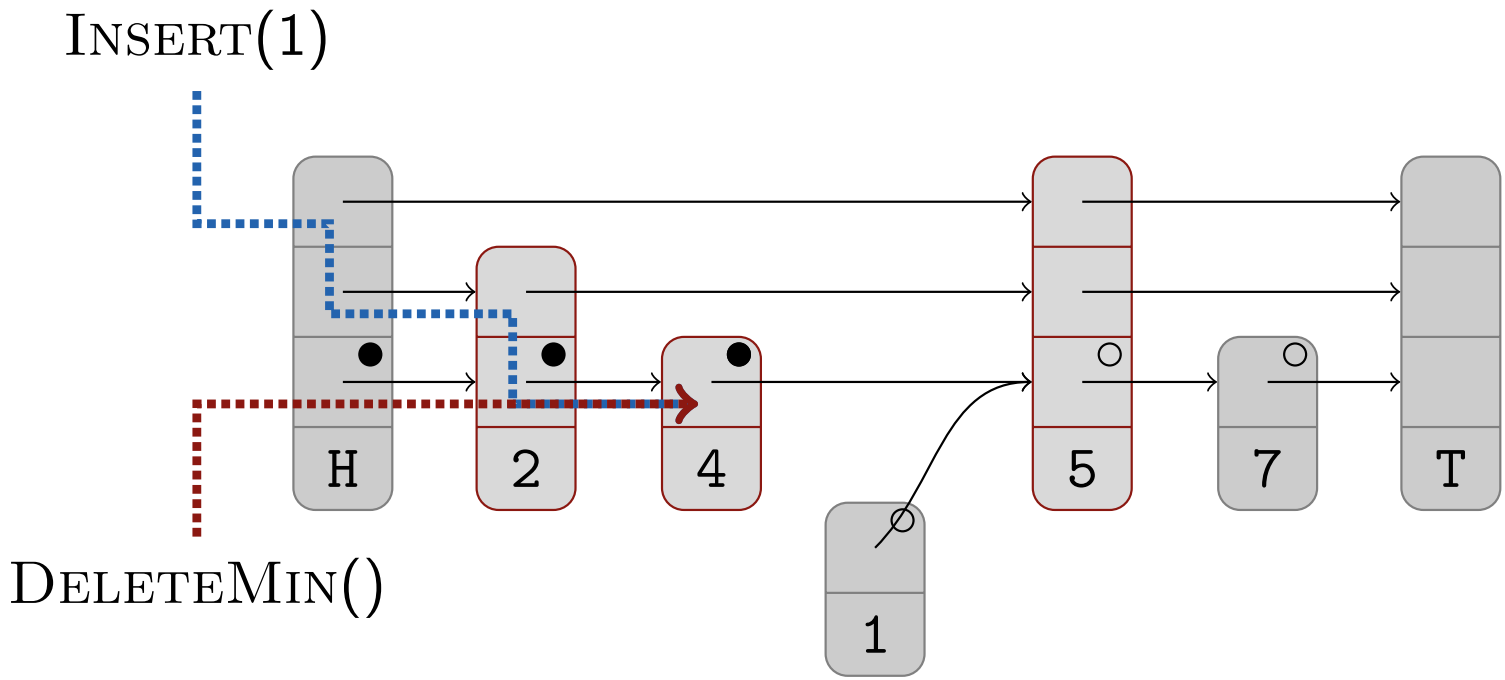
Our solution

Resolving conflicts between Insert and DeleteMin.



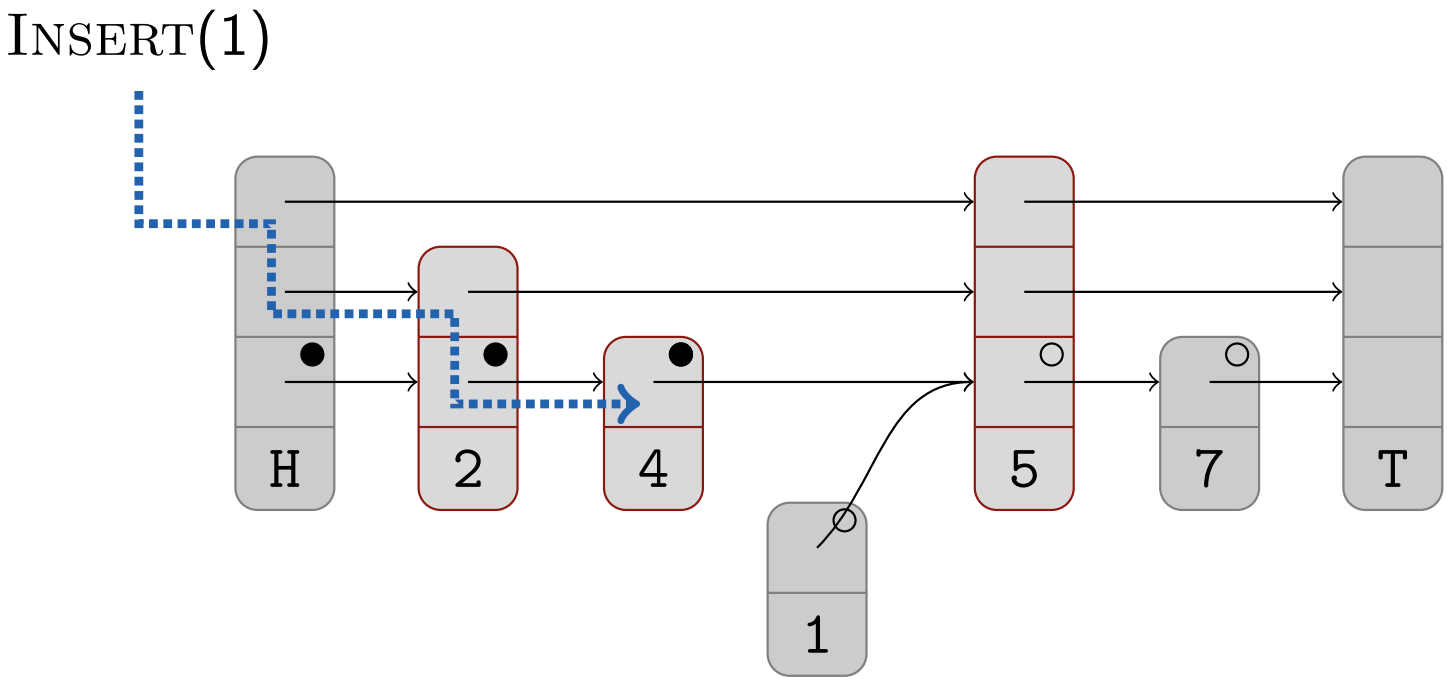
Our solution

Resolving conflicts between Insert and DeleteMin.



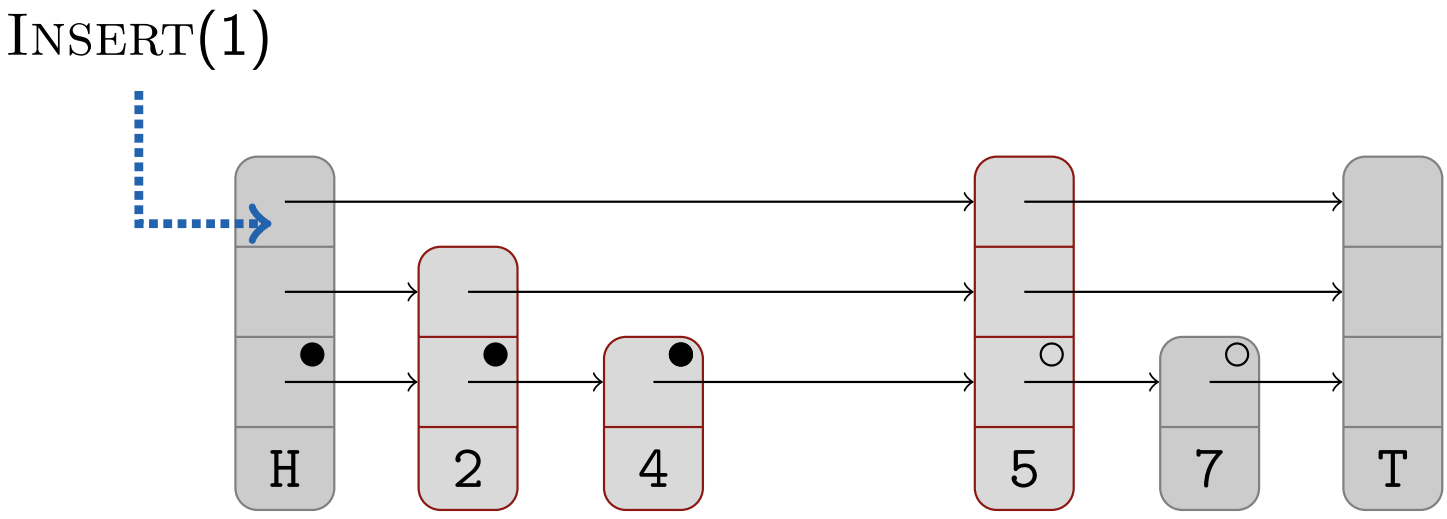
Our solution

Resolving conflicts between INSERT and DELETEMIN.



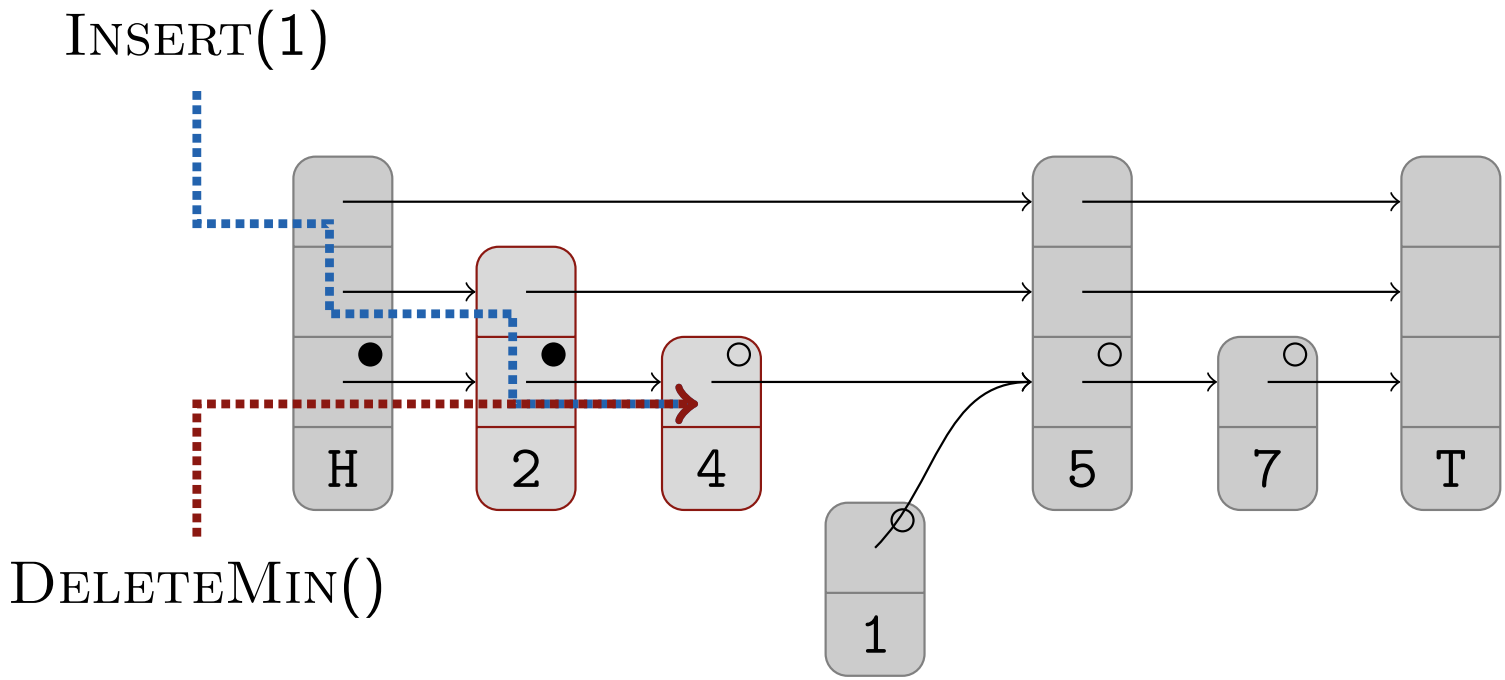
Our solution

Resolving conflicts between INSERT and DELETEMIN.



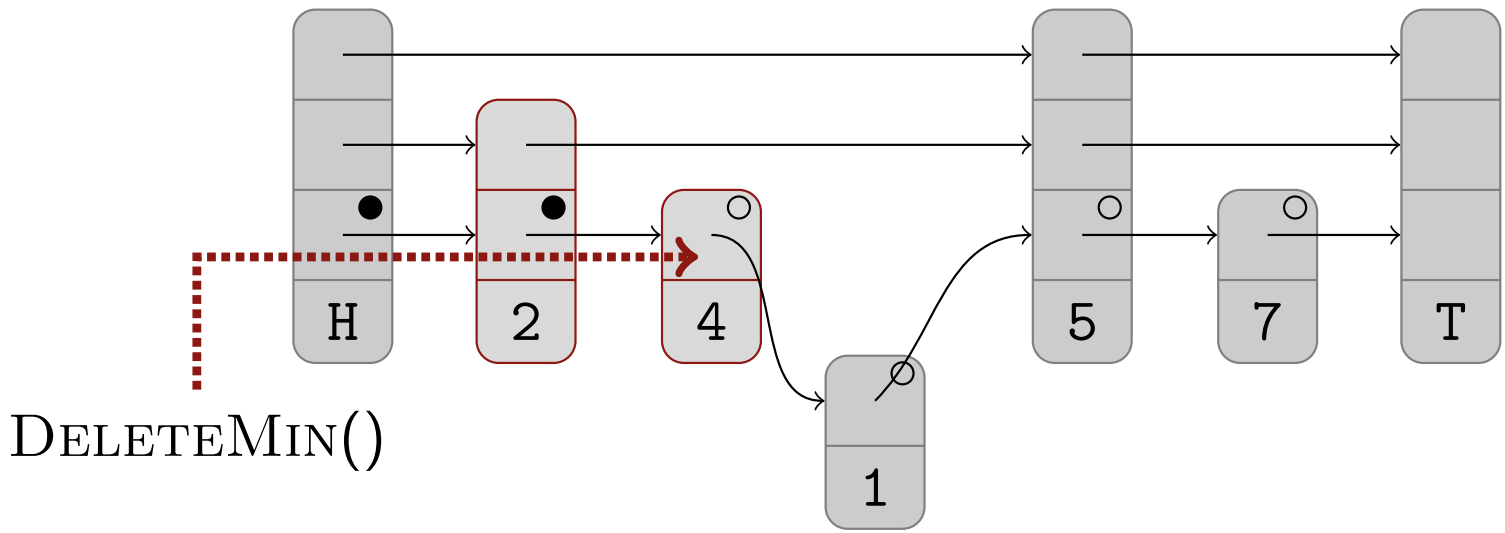
Our solution

Resolving conflicts between Insert and DeleteMin.



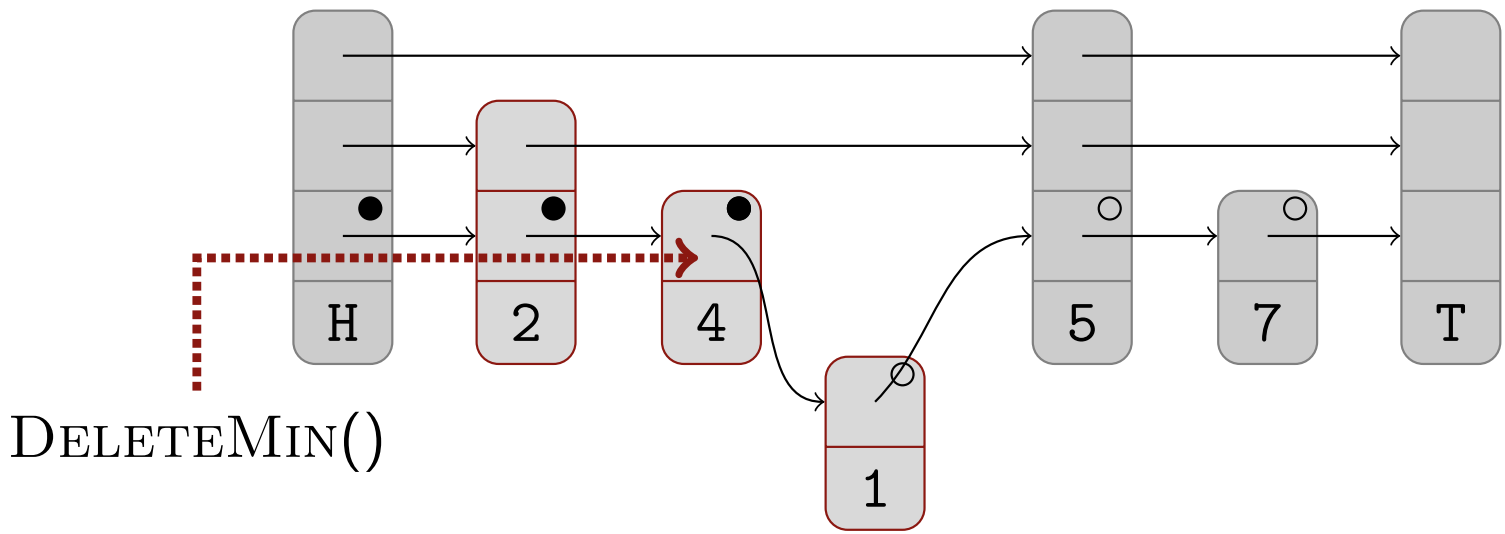
Our solution

Resolving conflicts between INSERT and DELETEMIN.



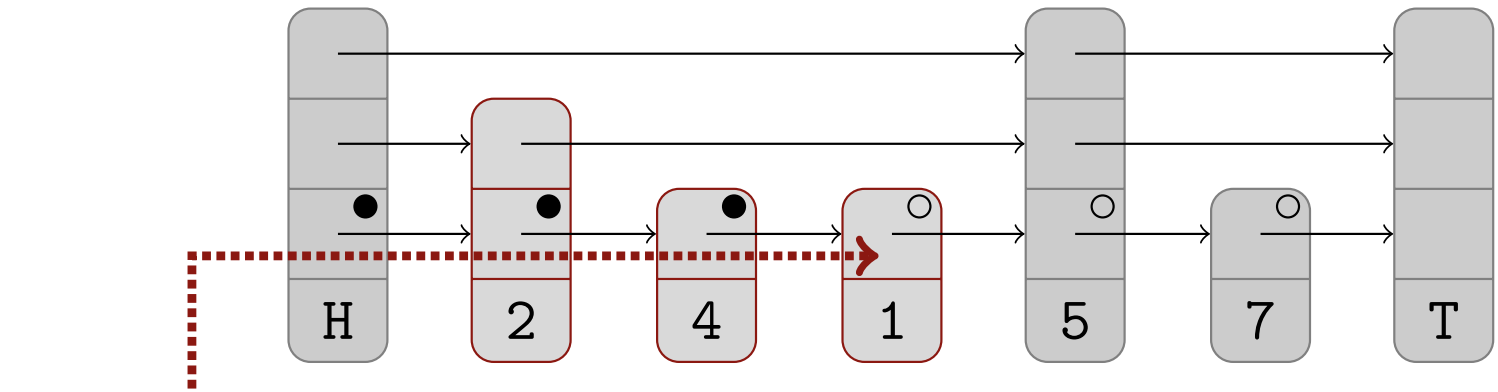
Our solution

Resolving conflicts between INSERT and DELETEMIN.



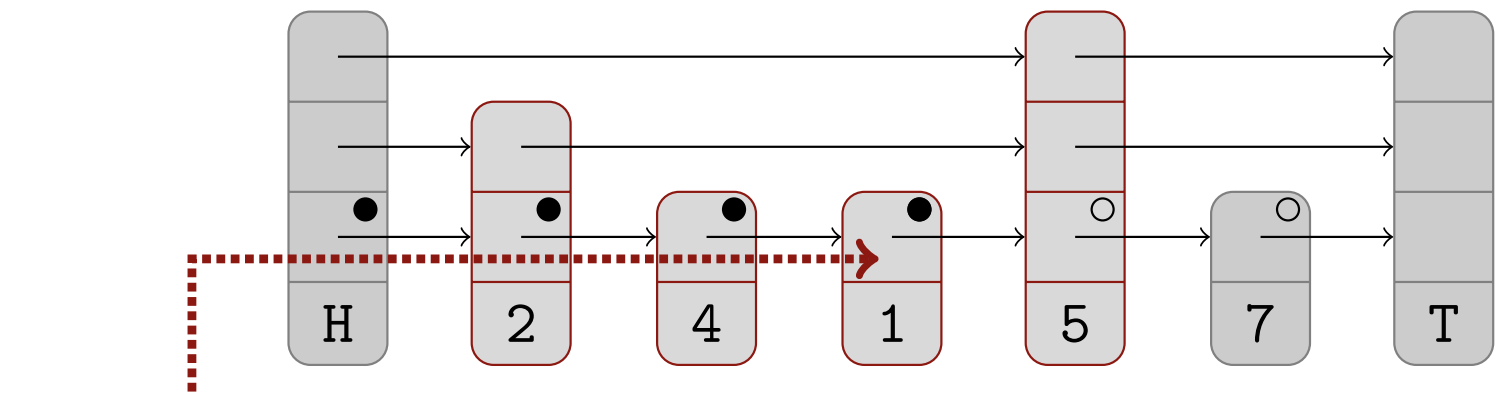
Physical batch deletion

- Physical batch deletion: update pointers in the head
- Done by DeleteMin, when the prefix of deleted nodes exceeds a threshold



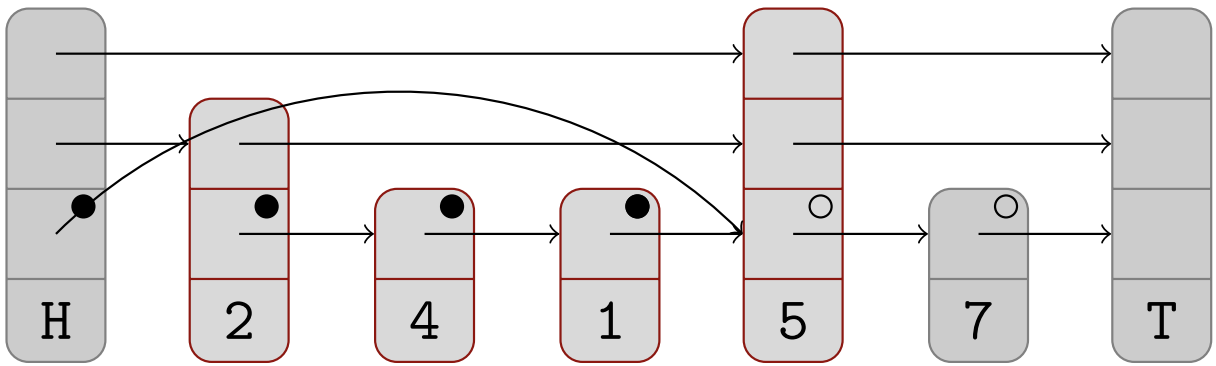
Physical batch deletion

- Physical batch deletion: update pointers in the head
- Done by DeleteMin, when the prefix of deleted nodes exceeds a threshold



Physical batch deletion

- Physical batch deletion: update pointers in the head
- Done by DeleteMin, when the prefix of deleted nodes exceeds a threshold



Correctness

- linearizable concurrent priority queue
- Correctness proof based on assertional reasoning in the paper
- Follows rather easily after establishing that the lowest level list consists of a deleted prefix followed by a sorted list which defines the logical state of the queue.
- We have also modeled the algorithm in SPIN and performed extensive state-space exploration

Evaluation

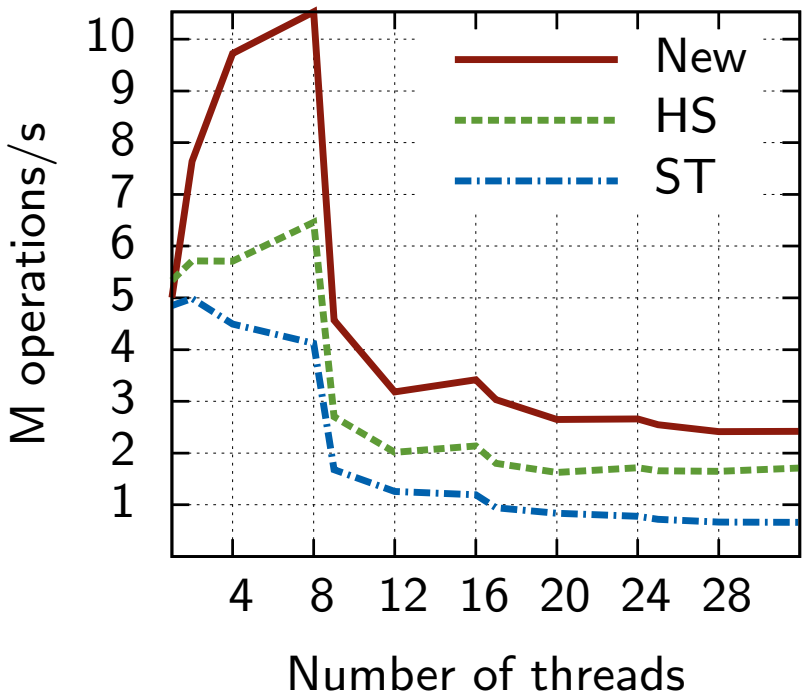
Comparison of maximal throughput

Compared against other lock-free skiplist-based priority queues:

- Sundell & Tsigas (*ST*): Only a single logically deleted node allowed in the lowest-level list.
- Herlihy & Shavit (*HS*): Lock-free adaptation of [Lotan & Shavit]. Logically deleted nodes need not form a prefix.

Comparison of maximal throughput

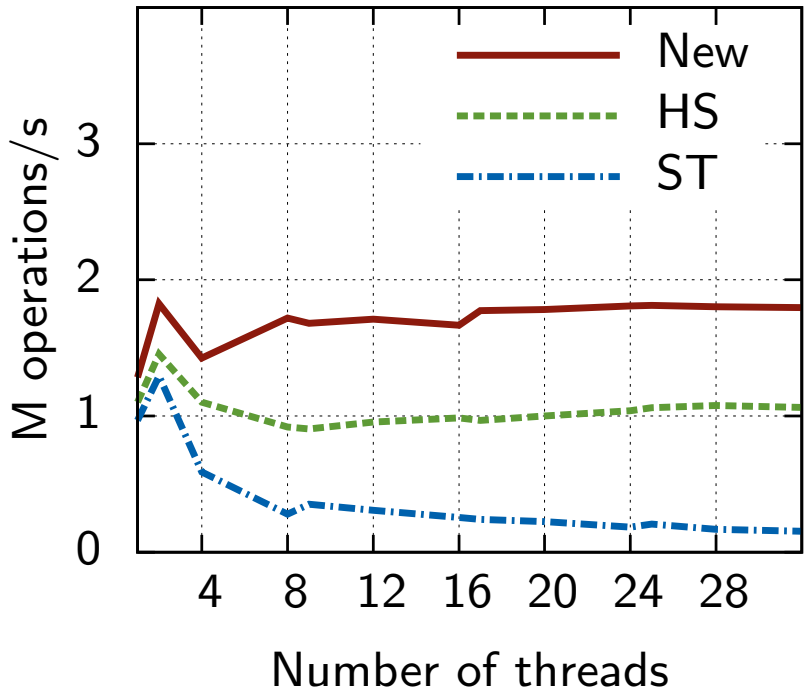
Benchmark: 50% Insert, 50% DeleteMin, 4 socket Intel sandybridge machine.



- 30 – 80% improvement in comparison to *HS*
- 1-8 cores: single socket (shared L3 cache)

Evaluation

Benchmark: DES workload, 4-socket AMD bulldozer machine.



- 30 – 80% improvement in comparison to *HS*
- 1 – 2 cores: shared L2 cache

More resources

- `http://user.it.uu.se/~jonli208/priorityqueue`
  - BSD licensed implementation
  - SPIN model
  - extended technical report
- Performance bug in the version in the proceedings: in the RESTRUCTURE algorithm which updates the head pointers – please see the extended technical report.

Conclusions

- a new linearizable, lock-free, skiplist-based priority queue algorithm.
- new representation of logical deletion.
- This reduces the number of CASes to critical shared memory locations
- 30 – 80 % performance improvement over existing such algorithms.

Conclusions

- a new linearizable, lock-free, skiplist-based priority queue algorithm.
- new representation of logical deletion.
- This reduces the number of CASes to critical shared memory locations
- 30 – 80 % performance improvement over existing such algorithms.

Thank you.