Polylogarithmic Fully Retroactive Priority Queues via Hierarchical Checkpointing

Erik D. Demaine, Tim Kaler, Quanquan Liu, Aaron Sidford, Adam Yedidia

Massachusetts Institute of Technology

CSAIL

Priority Queue Definition

- Abstract data type where each element given a priority key.

Priority Queue Definition

- Abstract data type where each element given a priority key.
- Priority queue supports:
  - Updates:

Priority Queue Definition

- Abstract data type where each element given a priority key.
- Priority queue supports:
  - Updates:
    - Insert: Inserts elements.

Priority Queue Definition

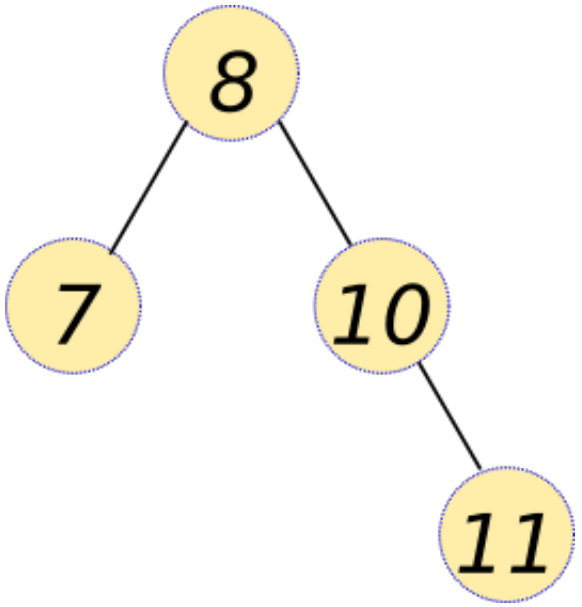
- Abstract data type where each element given a priority key.
- Priority queue supports:
  - Updates:
    - Insert: Inserts elements.
    - Delete-min: Deletes the element with minimum key.

Priority Queue Definition

- Abstract data type where each element given a priority key.
- Priority queue supports:
  - Updates:
    - Insert: Inserts elements.
    - Delete-min: Deletes the element with minimum key.
  - Query:
    - Find-min: Finds the minimum key value element.

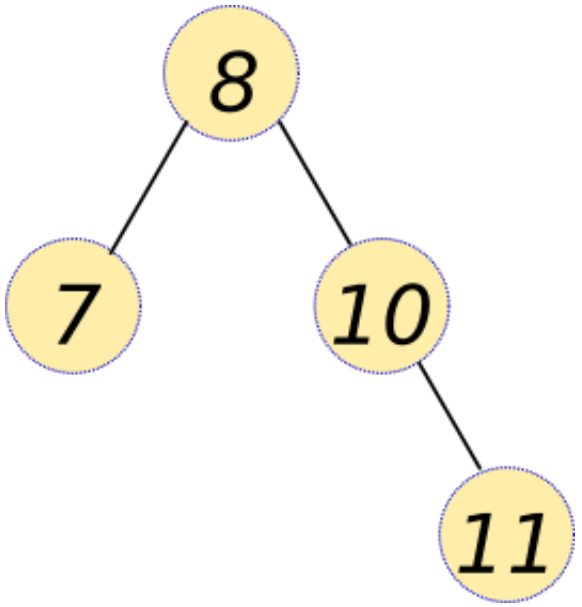
Priority Queue Definition

- Abstract data type where each element given a priority key.
- Priority queue supports:
  - Updates:
    - Insert: Inserts elements.
    - Delete-min: Deletes the element with minimum key.
  - Query:
    - Find-min: Finds the minimum key value element.



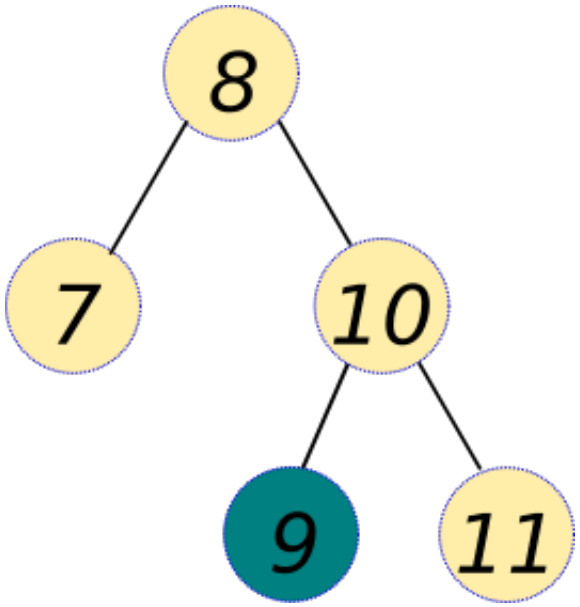
Traditional Priority Queue

- Normal priority queue events occur in the present.



Traditional Priority Queue

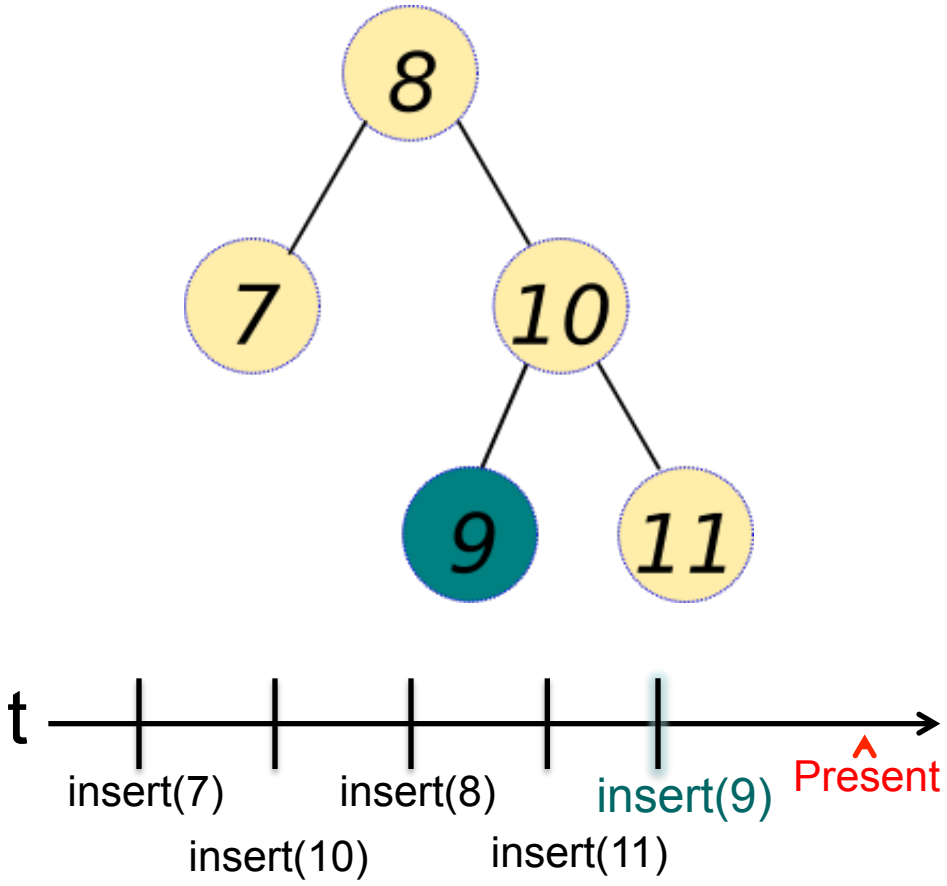
- Normal priority queue events occur in the present.



insert(9)

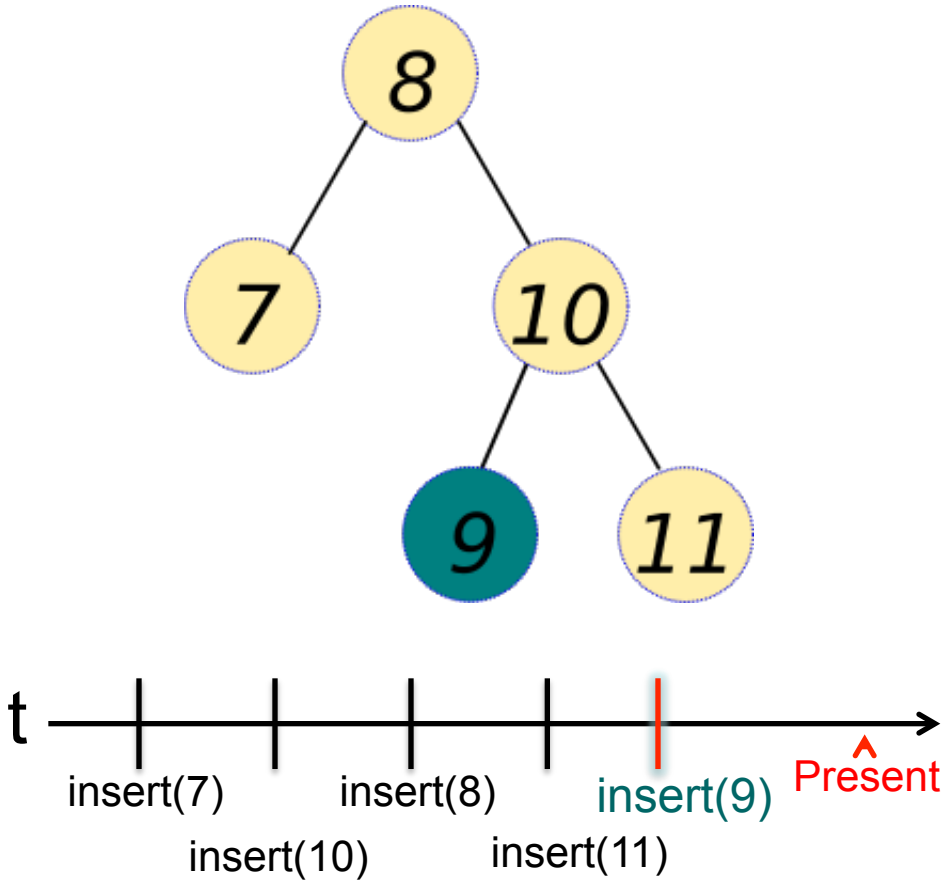
Traditional Priority Queue

- Normal priority queue events occur in the present.
- Timeline of update events maintains history of updates.



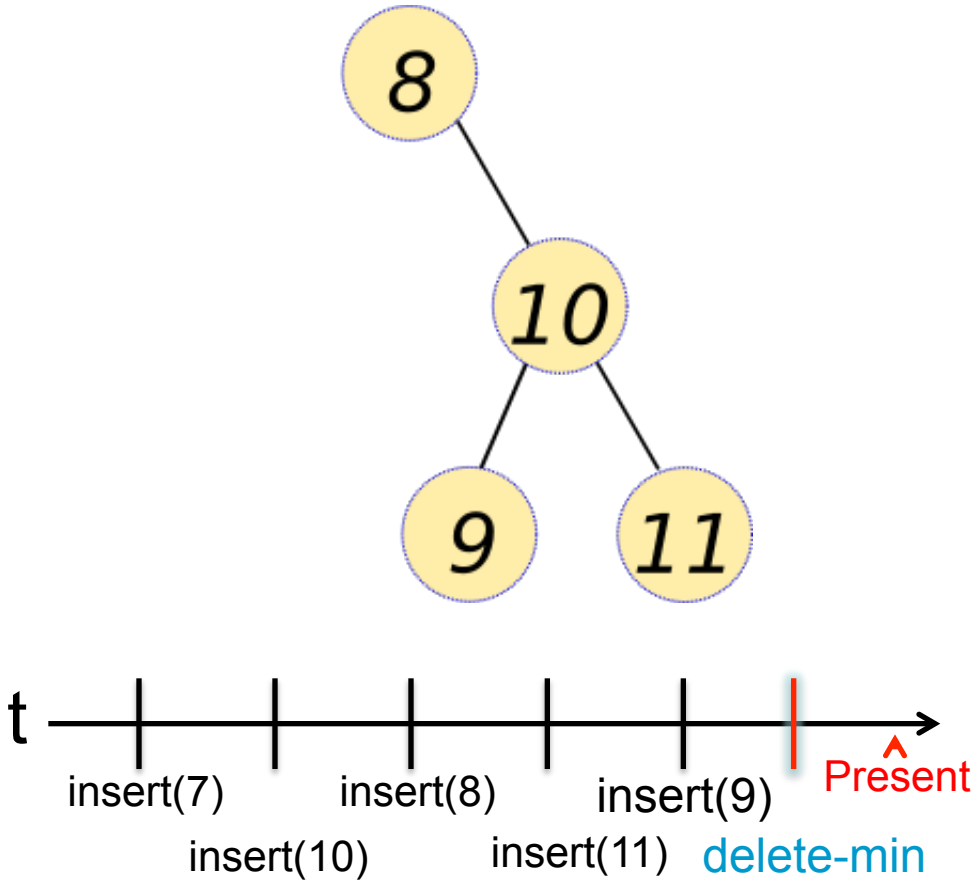
Traditional Priority Queue

- Normal priority queue events occur in the present.
- Timeline of update events maintains history of updates.
- Updates occur in the present and are appended to the end of the timeline.



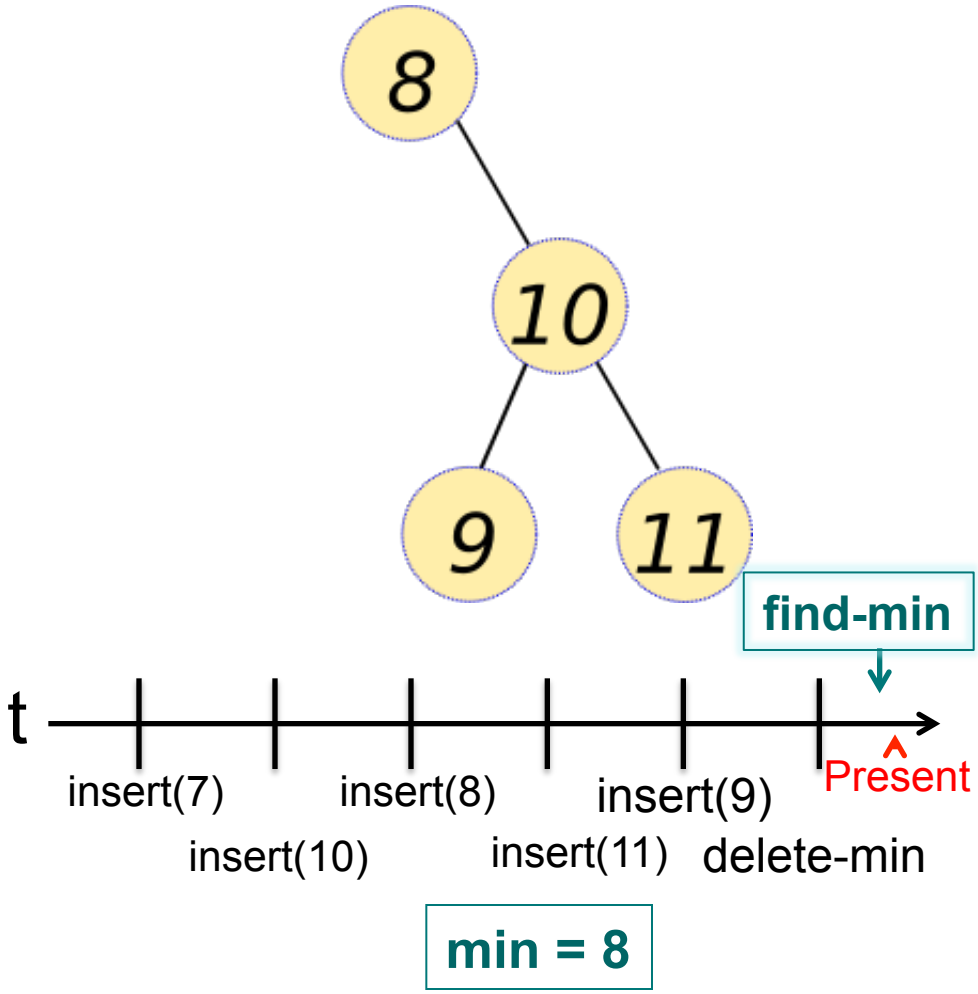
Traditional Priority Queue

- Normal priority queue events occur in the present.
- Timeline of update events maintains history of updates.
- Updates occur in the present and are appended to the end of the timeline.



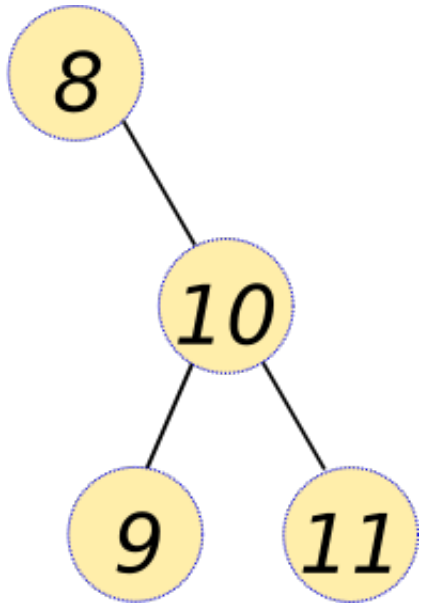
Traditional Priority Queue

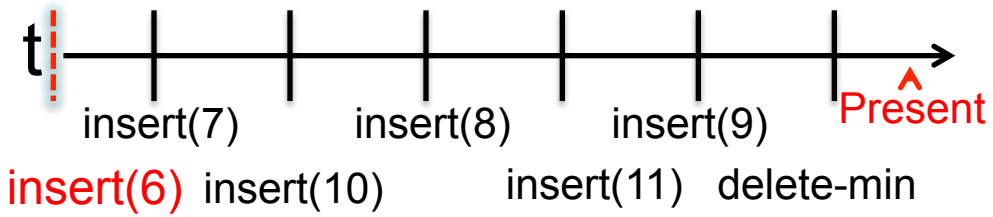
- Normal priority queue events occur in the present.
- Timeline of update events maintains history of updates.
- Updates occur in the present and are appended to the end of the timeline.
- Queries can be made in the present.



Updates in the Past

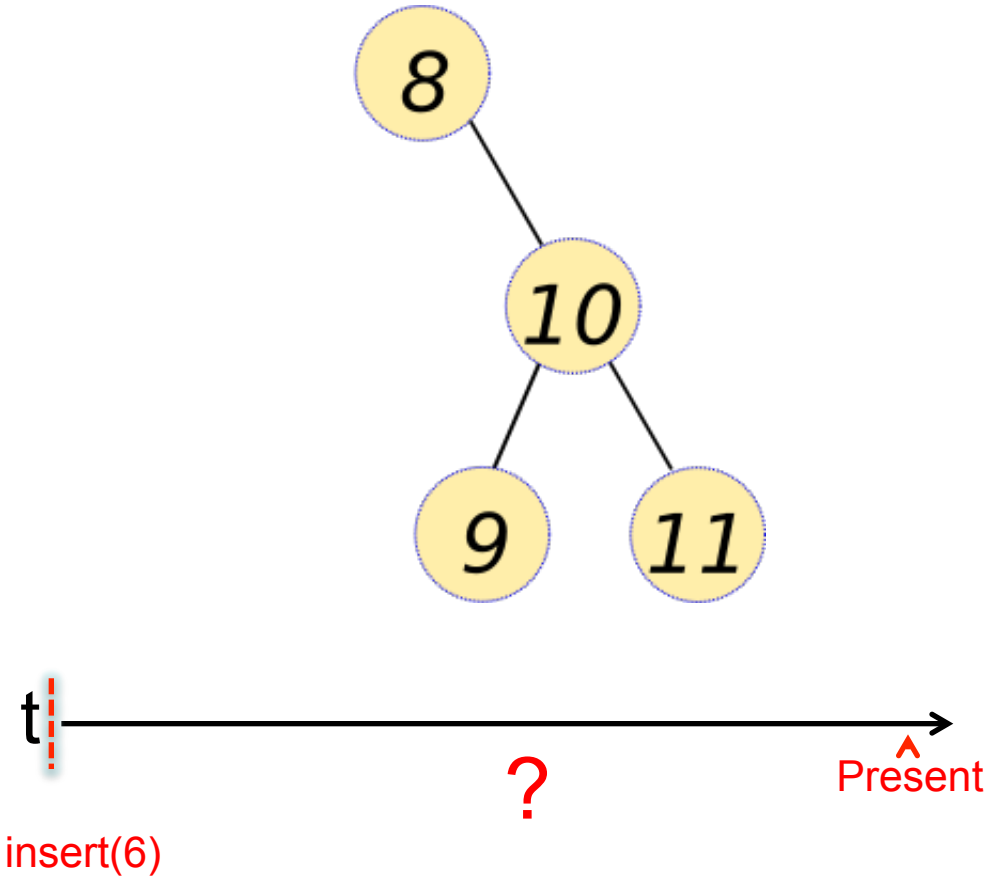
- Would like to insert(6) at the beginning of the timeline.





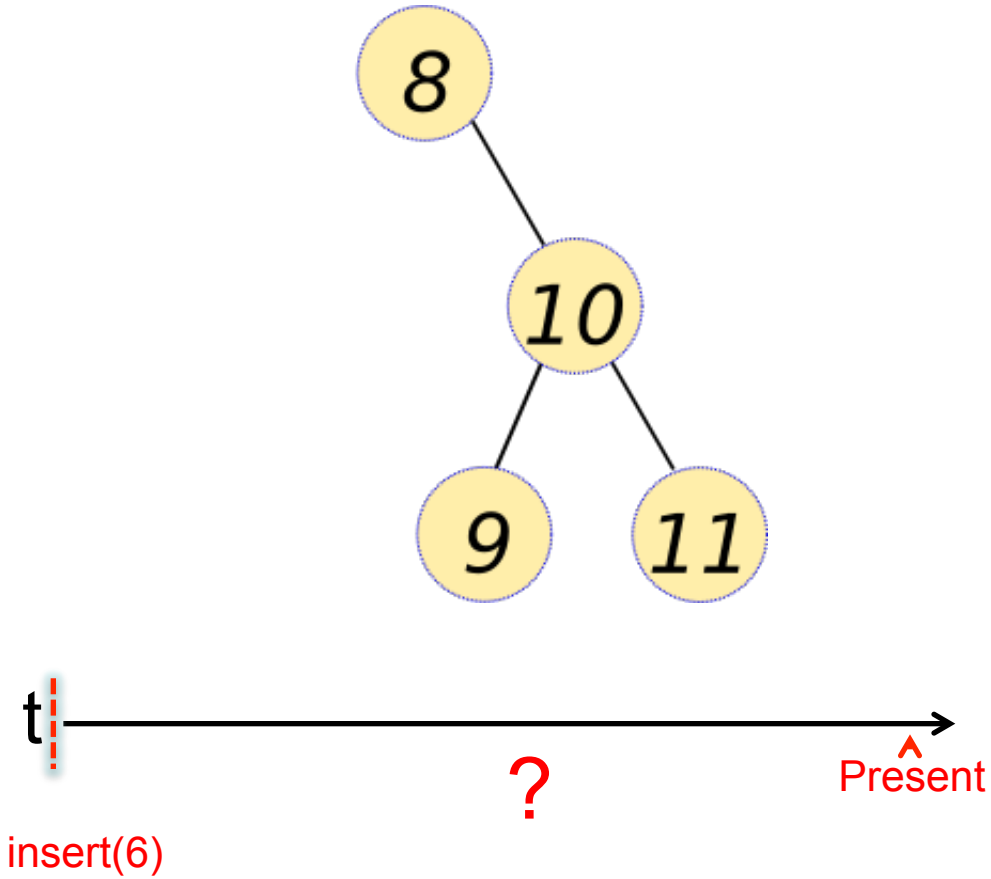
Updates in the Past

- Would like to insert(6) at the beginning of the timeline.
- Cannot insert(6) right now because do not know intervening updates.



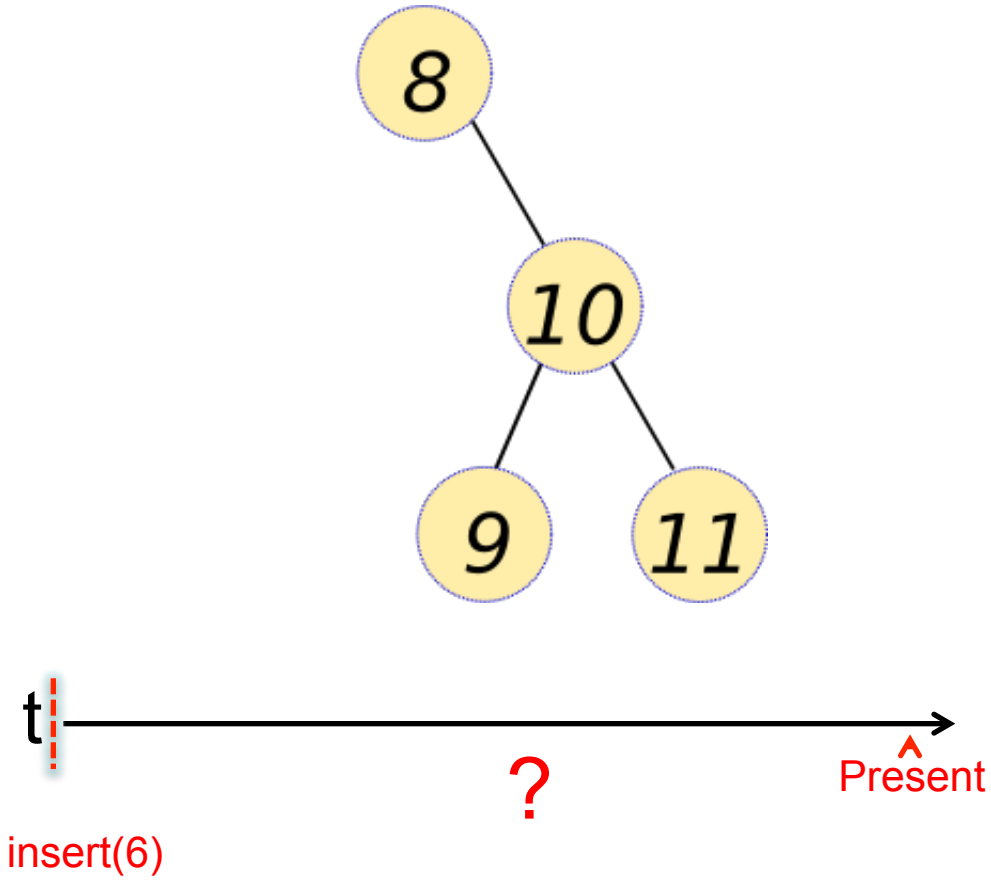
Updates in the Past

- Would like to insert(6) at the beginning of the timeline.
- Cannot insert(6) right now because do not know intervening updates.
- In a normal priority queue, can replay all intervening updates.



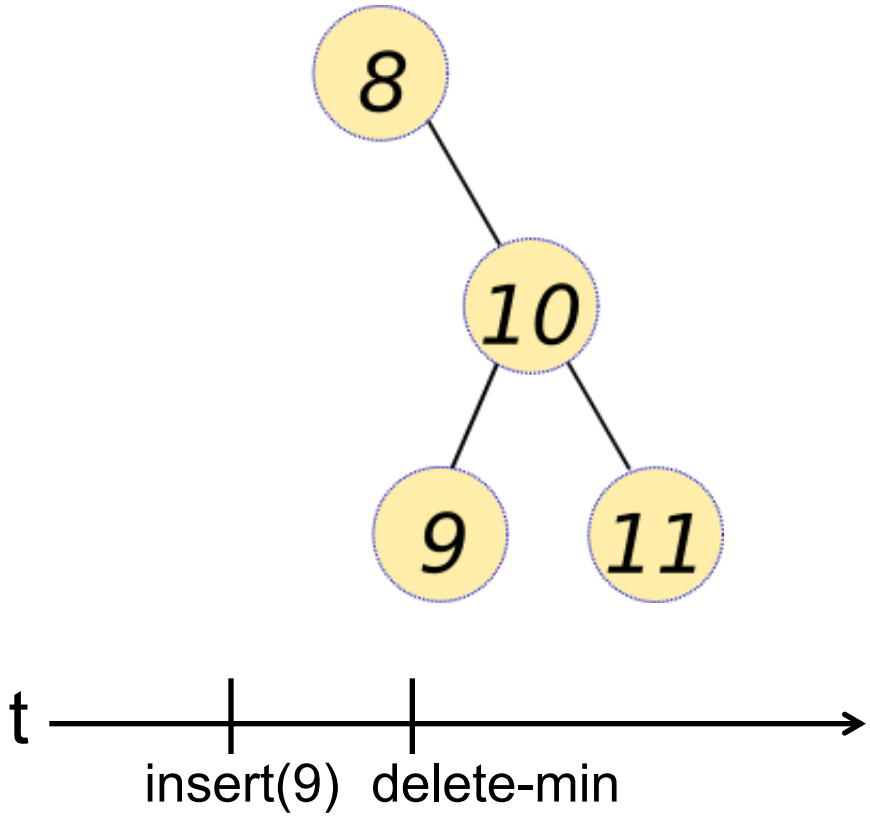
Updates in the Past

- Would like to insert(6) at the beginning of the timeline.
- Cannot insert(6) right now because do not know intervening updates.
- In a normal priority queue, can replay all intervening updates.
- Costly if large number of updates.

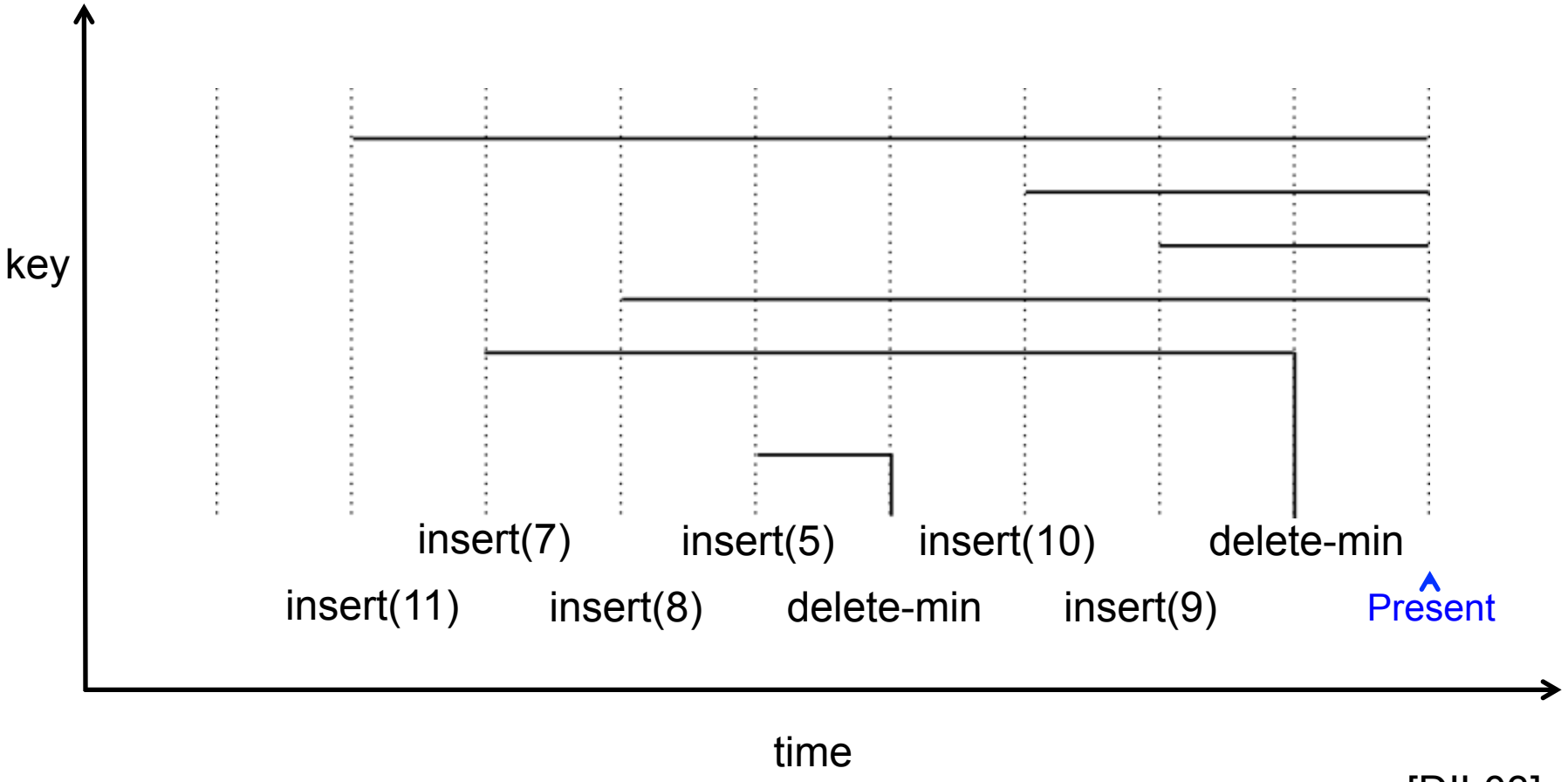


Priority Queue Definition

- Update in the present are appended to the end of the timeline:
  - insert
  - delete-min
- Query the state of the structure in the present:
  - find-min

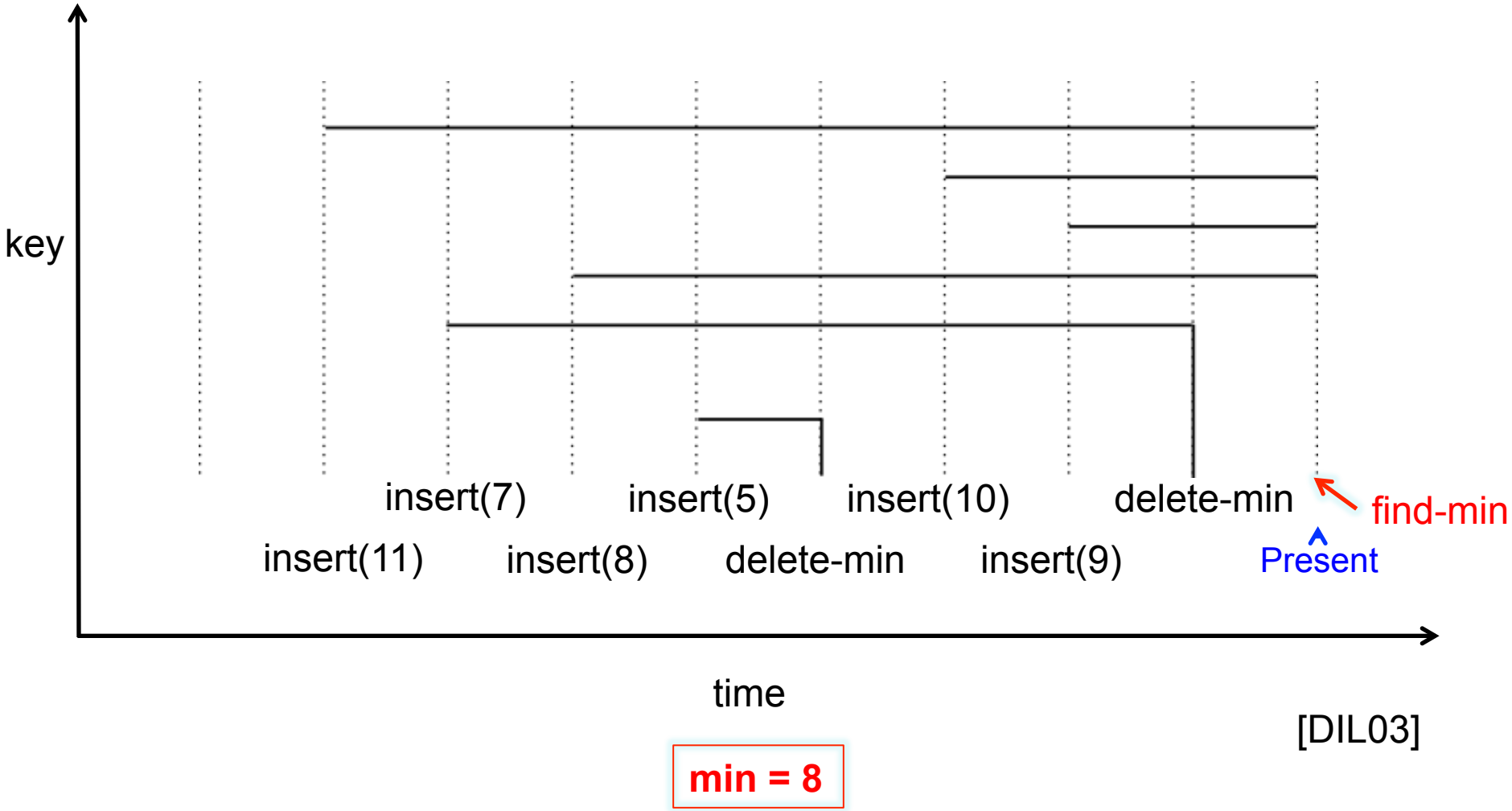


Partially Retroactive Priority Queue



[DIL03]

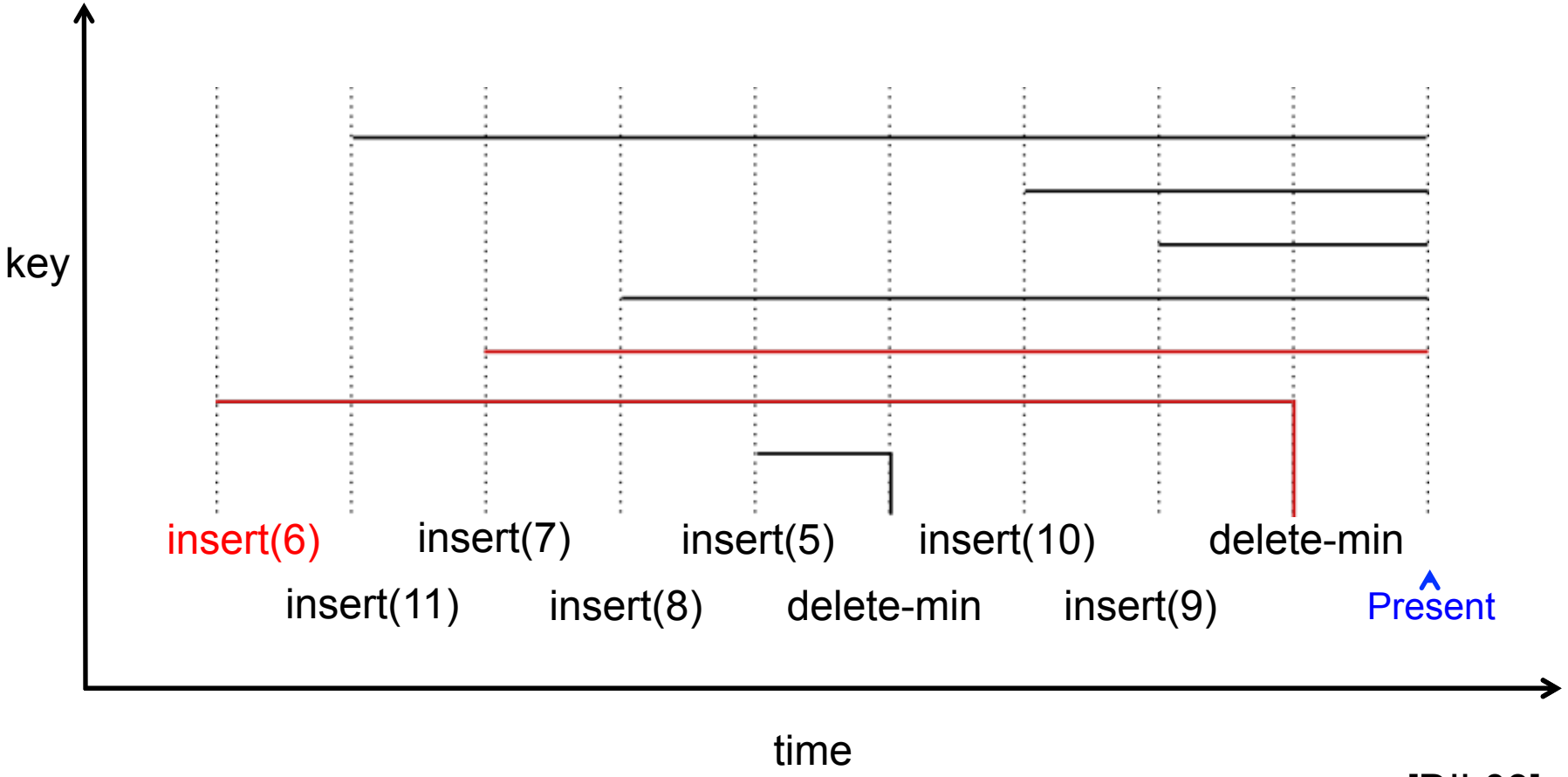
Partially Retroactive Priority Queue



[DIL03]

min = 8

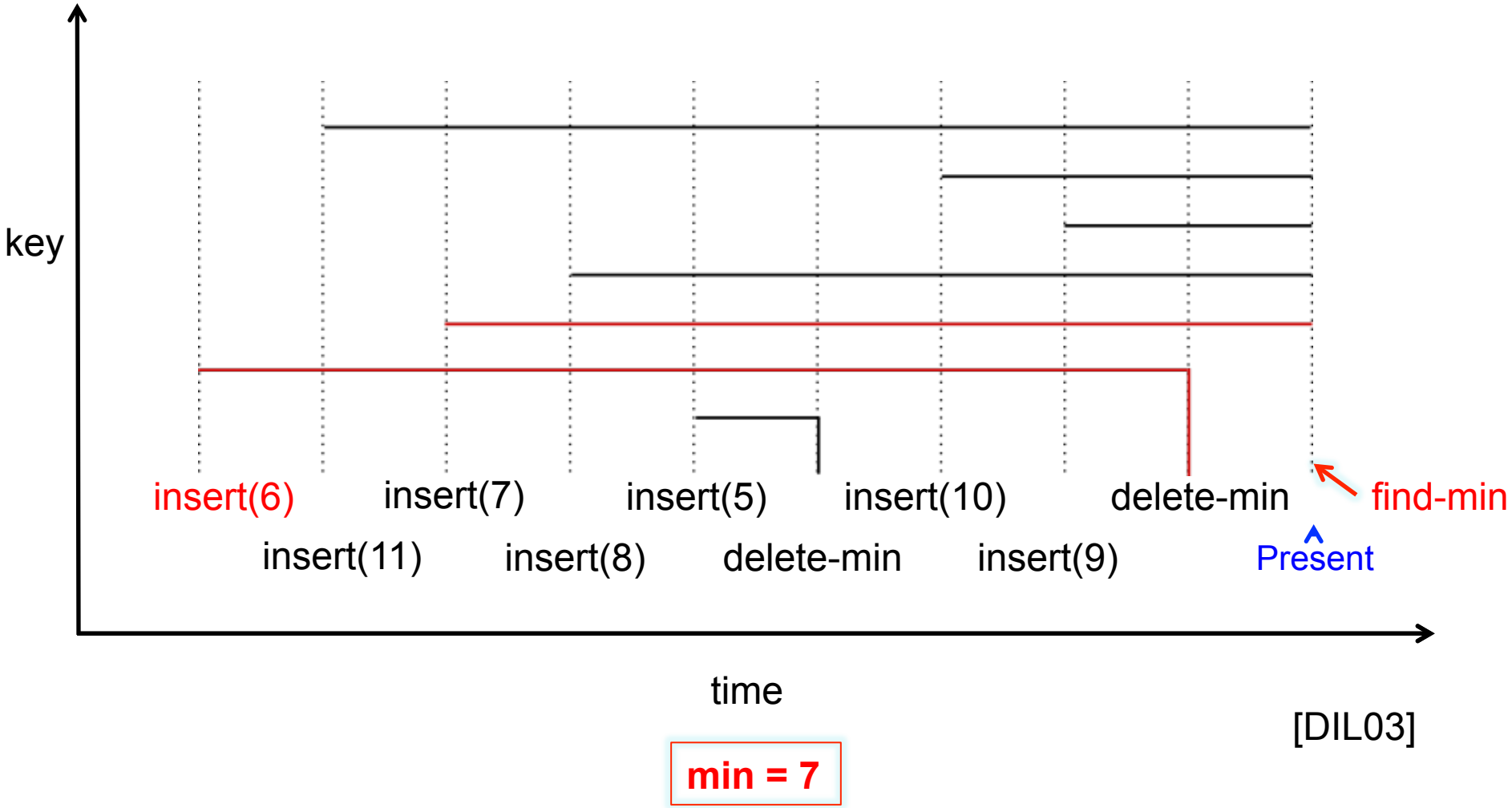
Partially Retroactive Priority Queue: Update



[DIL03]

Insert-Op(0, insert(6))

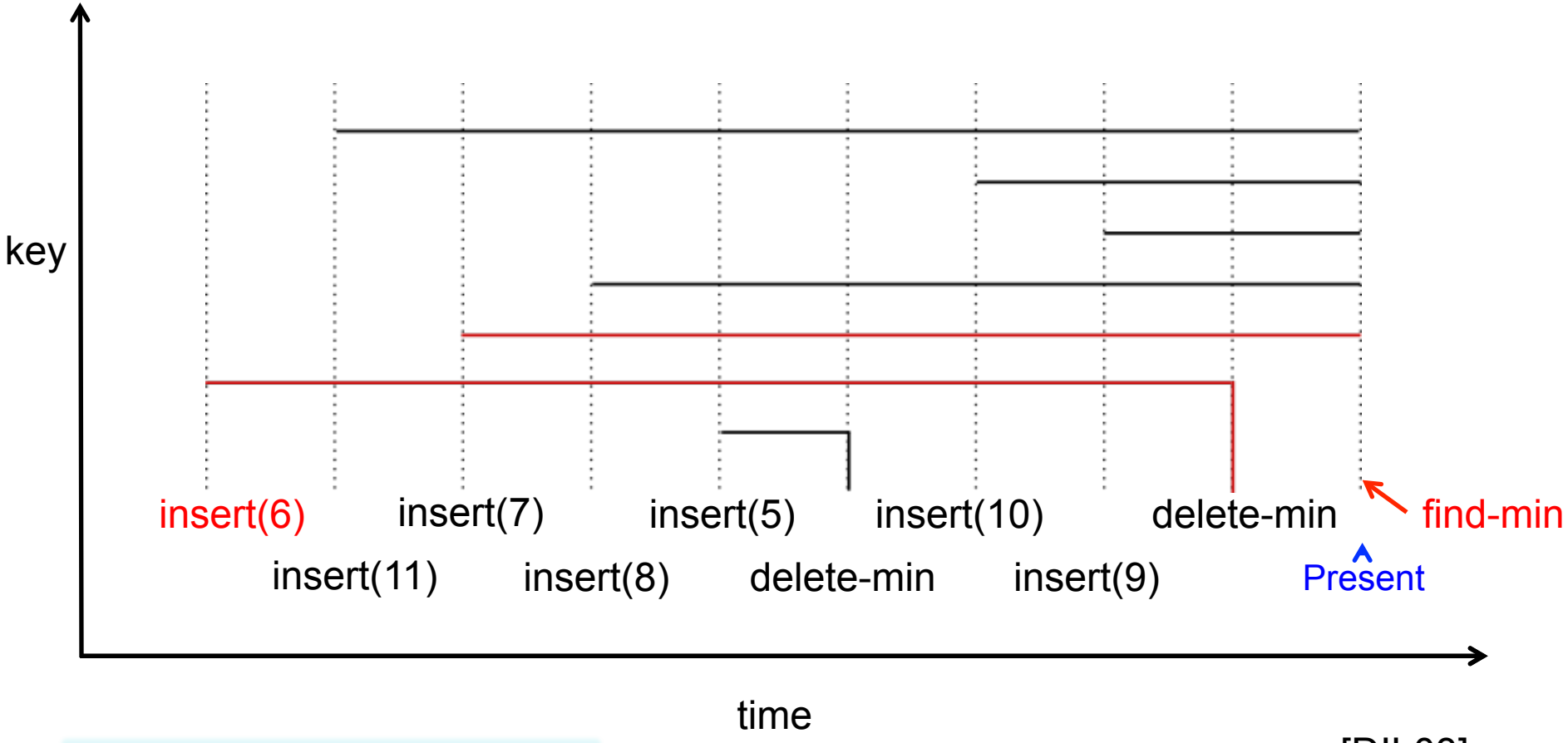
Partially Retroactive Priority Queue: Query



[DIL03]

min = 7

Partially Retroactive Priority Queue: Query



[DIL03]

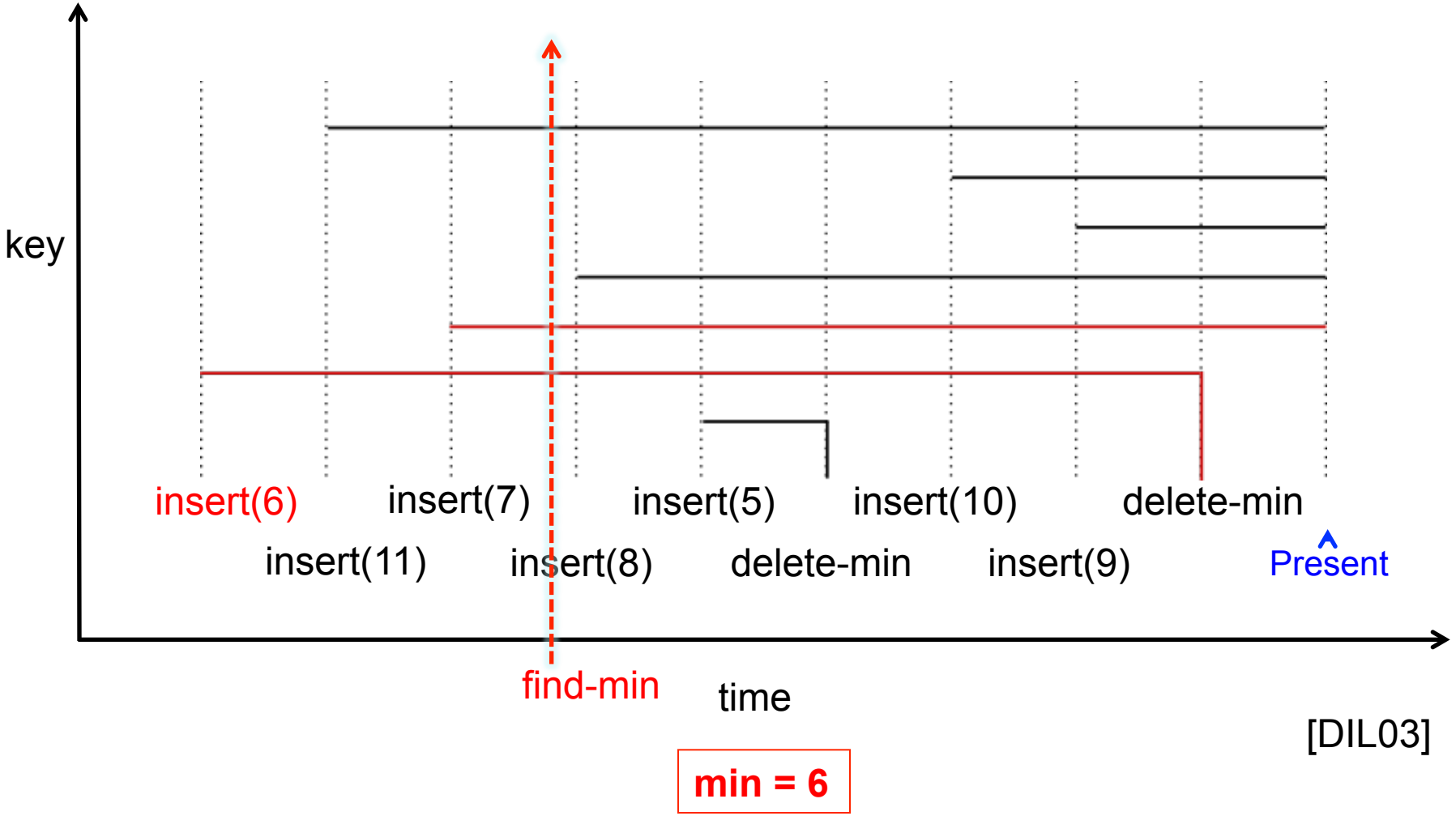
$O(\log m)$ runtime

min = 7

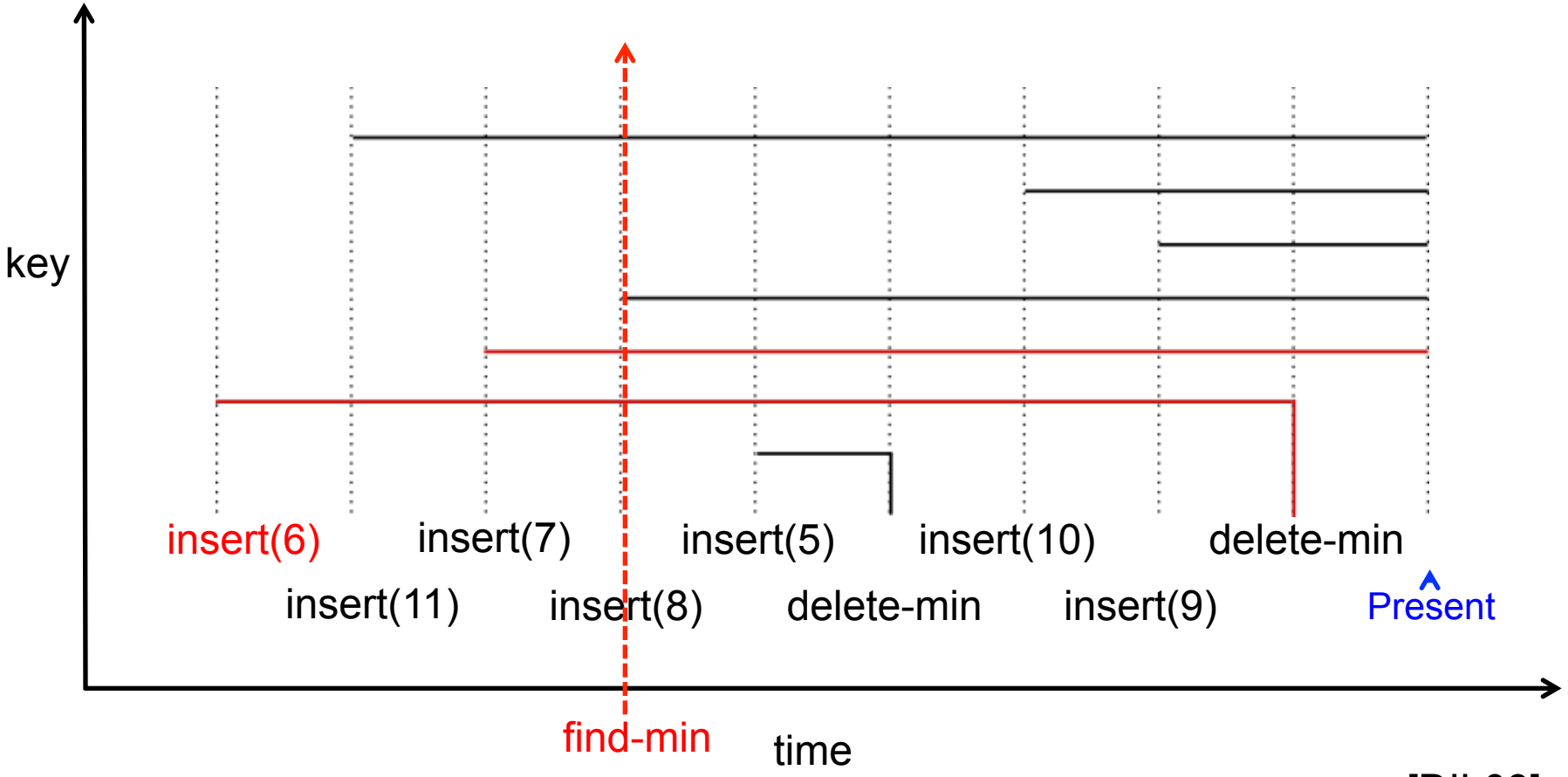
m = number of updates in timeline.

Demaine, Kaler, Liu, Sidford, Yedidia

Fully Retroactive Priority Queue: Query



Previous Result: Fully Retroactive Priority Queue



[DIL03]

$O(\sqrt{m}\log m)$ runtime

min = 6

Demaine, Kaler, Liu, Sidford, Yedidia

Partial vs. Full Retroactivity

- Partial retroactivity: updates can be added or deleted from anywhere in timeline but queries can only be made in the present. [DIL03]
- Full retroactivity: queries can be made at any point in the timeline. [DIL03]
- Not always easy converting from partial to full retroactivity

Partial vs. Full Retroactivity

	Normal	Partially Retroactive	Fully Retroactive
Updates	Present		
Queries	Present		

Partial vs. Full Retroactivity

	Normal	Partially Retroactive	Fully Retroactive
Updates	Present	Past and Present	
Queries	Present	Present	

Partial vs. Full Retroactivity

	Normal	Partially Retroactive	Fully Retroactive
Updates	Present	Past and Present	Past and Present
Queries	Present	Present	Past and Present

Partial vs. Full Retroactivity

	Normal	Partially Retroactive	Fully Retroactive
Updates	Present	Past and Present	Past and Present
Queries	Present	Present	Past and Present

Can we minimize the cost of converting from partial to full retroactivity for retroactive data structures?

Previous Results

[DIL03, GK09]:

Abstract Data Type	Partially Retroactive	Fully Retroactive
Dictionary	$O(\log m)$	$O(\log m)$
Queue	$O(1)$	$O(\log m)$
Stack	$O(\log m)$	$O(\log m)$
Deque	$O(\log m)$	$O(\log m)$
Union-Find	$O(\log m)$	$O(\log m)$
Priority queue	$O(\log m)$	$O(\sqrt{m}\log m)$

Previous Results

[DIL03, GK09]:

Abstract Data Type	Partially Retroactive	Fully Retroactive
Dictionary	$O(\log m)$	$O(\log m)$
Queue	$O(1)$	$O(\log m)$
Stack	$O(\log m)$	$O(\log m)$
Deque	$O(\log m)$	$O(\log m)$
Union-Find	$O(\log m)$	$O(\log m)$
Priority queue	$O(\log m)$	$O(\sqrt{m}\log m)$

$O(\sqrt{m})$ overhead in priority queues obtained through checkpointing

Demaine, Kaler, Liu, Sidford, Yedidia

Our Results

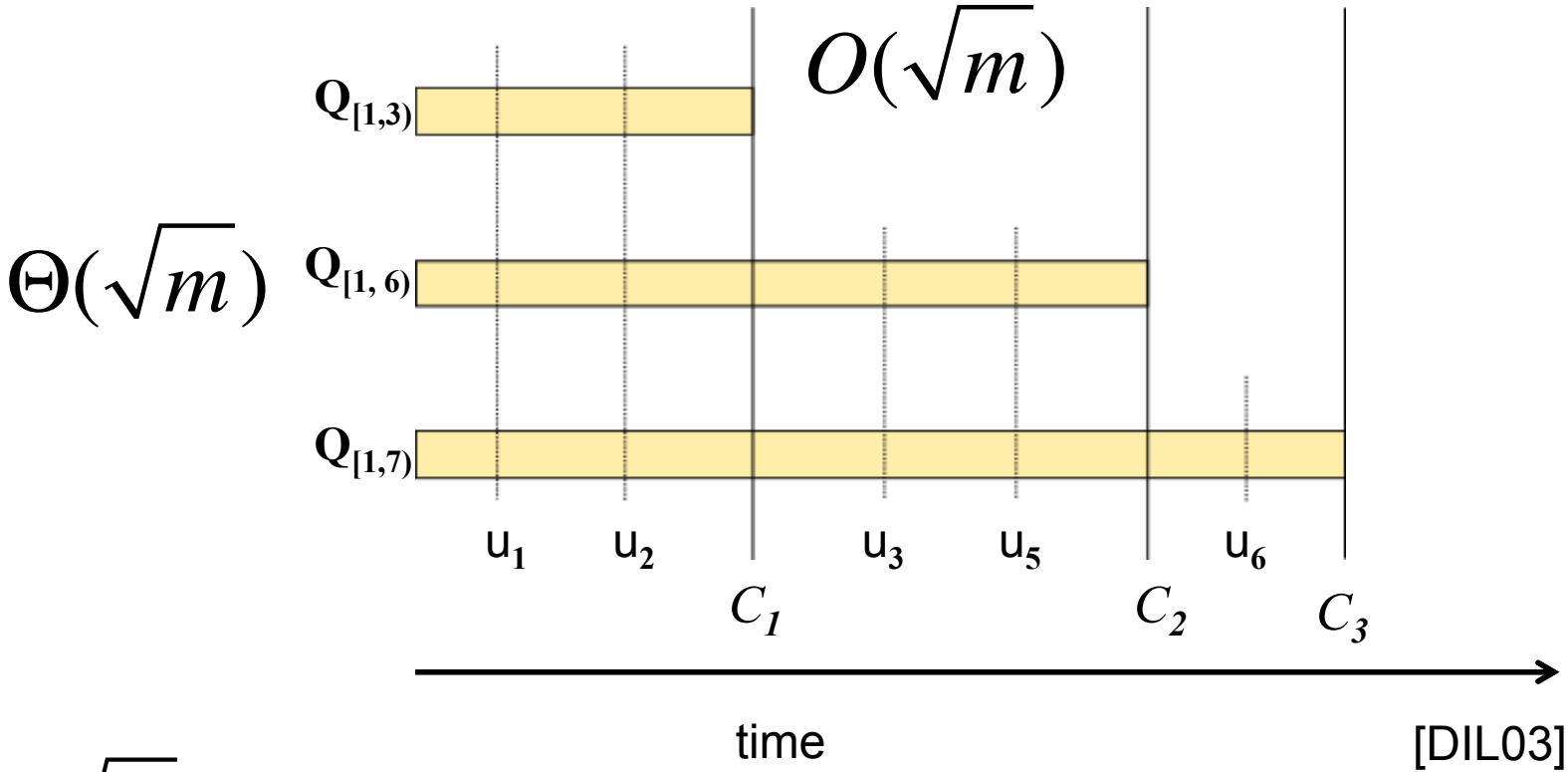
[DIL03, GK09, DKLSY15]:

Abstract Data Type	Partially Retroactive	Fully Retroactive
Dictionary	$O(\log m)$	$O(\log m)$
Queue	$O(1)$	$O(\log m)$
Stack	$O(\log m)$	$O(\log m)$
Deque	$O(\log m)$	$O(\log m)$
Union-Find	$O(\log m)$	$O(\log m)$
Priority queue	$O(\log m)$	$\tilde{O}(\log^2 m)$

$\tilde{O}(\log m)$ overhead in priority queues obtained through hierarchical checkpointing

Demaine, Kaler, Liu, Sidford, Yedidia

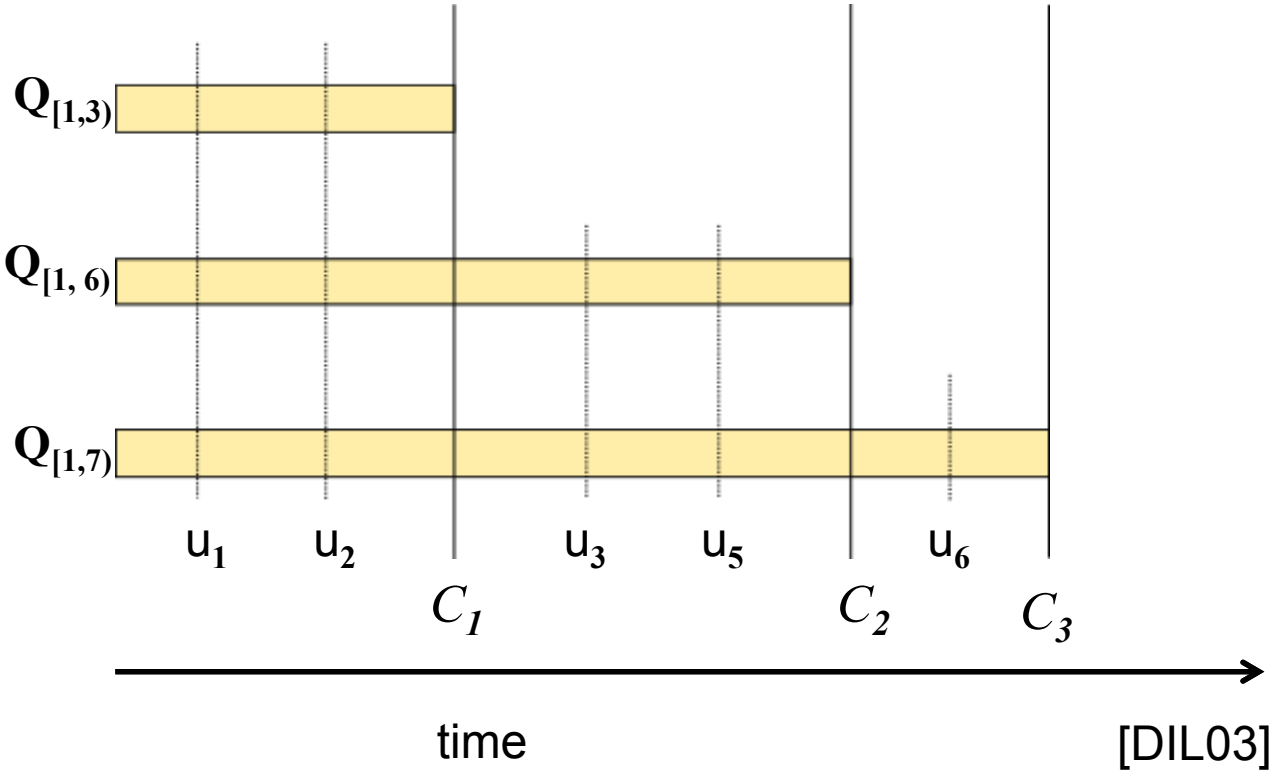
Previous Results
Checkpointing



[DIL03]

$\Theta(\sqrt{m})$ partially retroactive priority queues each spanning a prefix of the timeline.

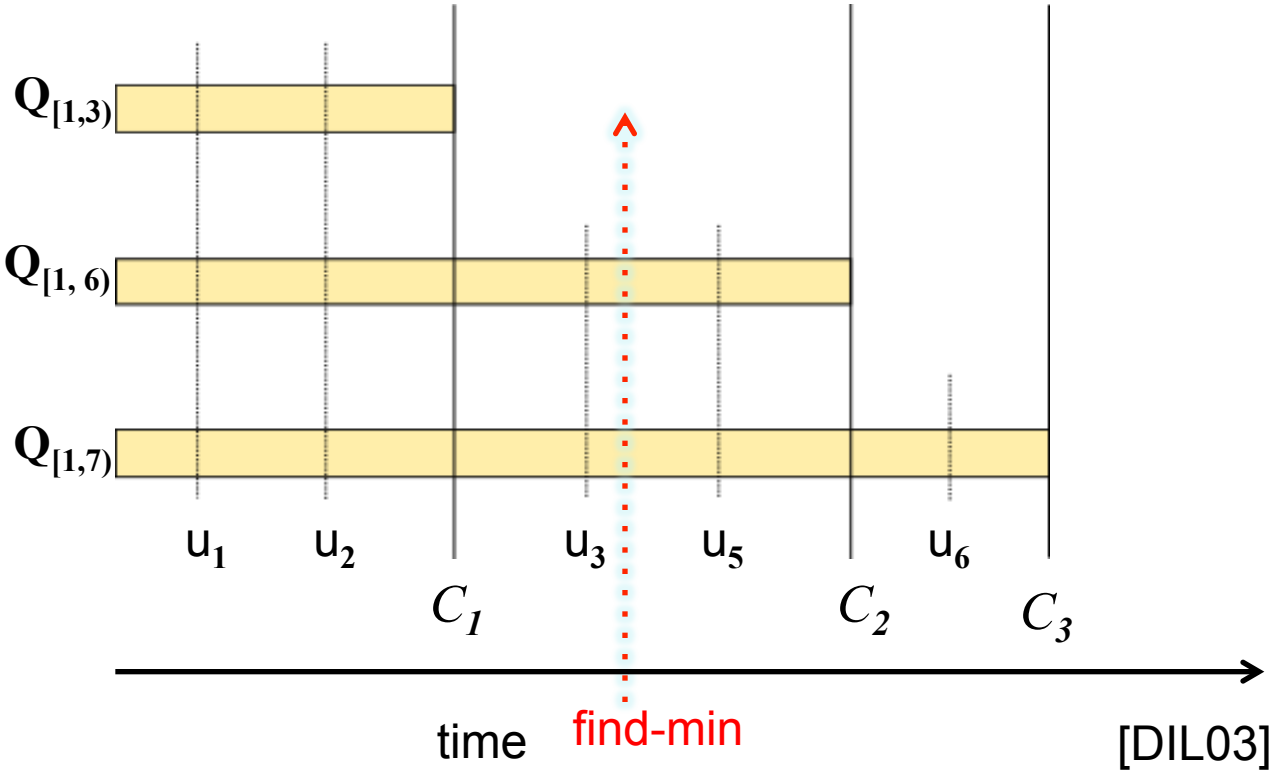
Previous Results
Checkpointing: Query



[DIL03]

Query nearest checkpoint structure containing updates before query time.

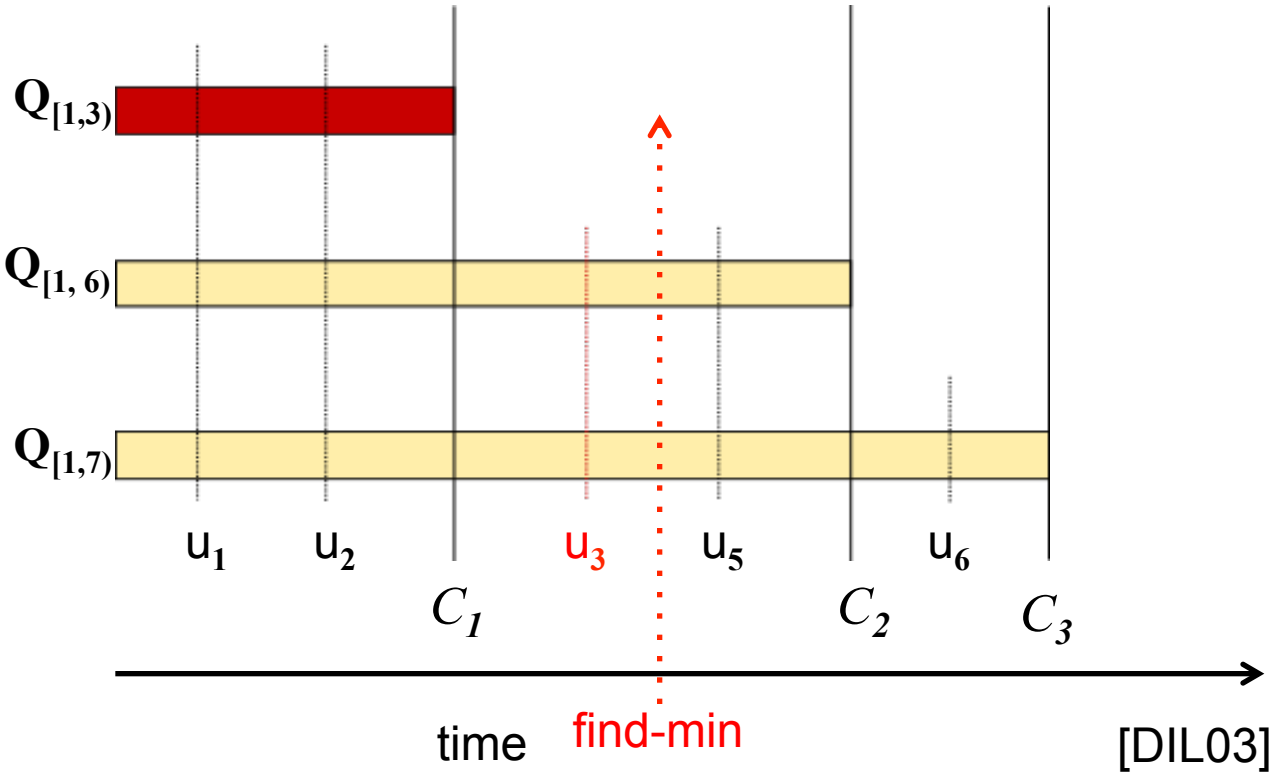
Previous Results
Checkpointing: Query



Query nearest checkpoint structure containing updates before query time.

Demaine, Kaler, Liu, Sidford, Yedidia

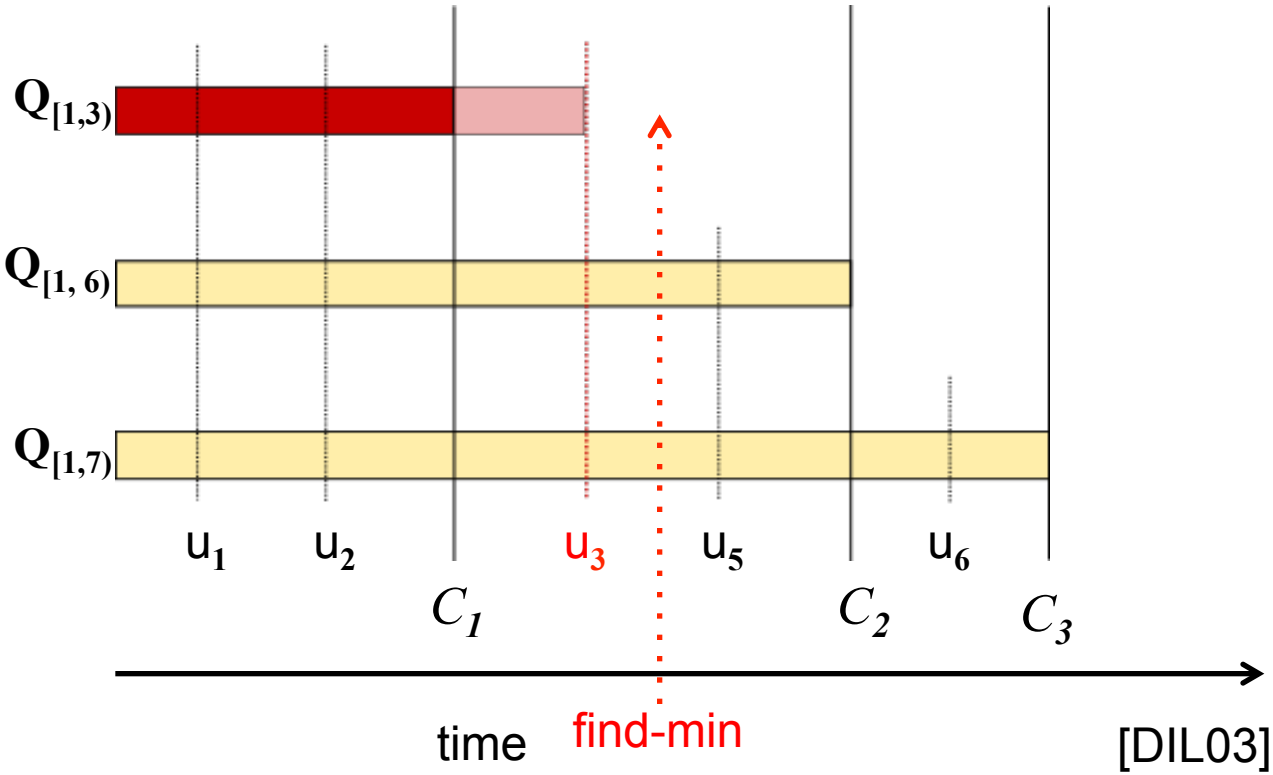
Previous Results
Checkpointing: Query



Query nearest checkpoint structure containing updates before query time.

Demaine, Kaler, Liu, Sidford, Yedidia

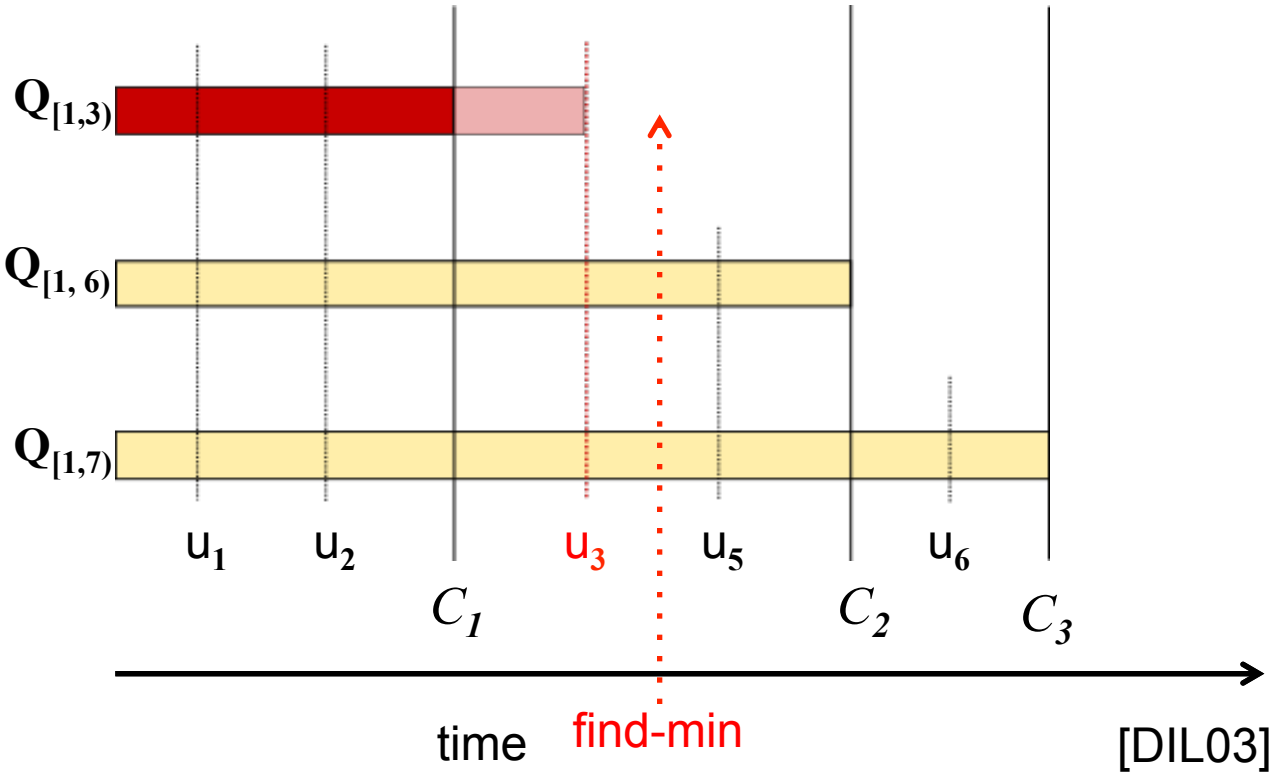
Previous Results
Checkpointing: Query



Temporarily add updates to partially retroactive priority queue.

Demaine, Kaler, Liu, Sidford, Yedidia

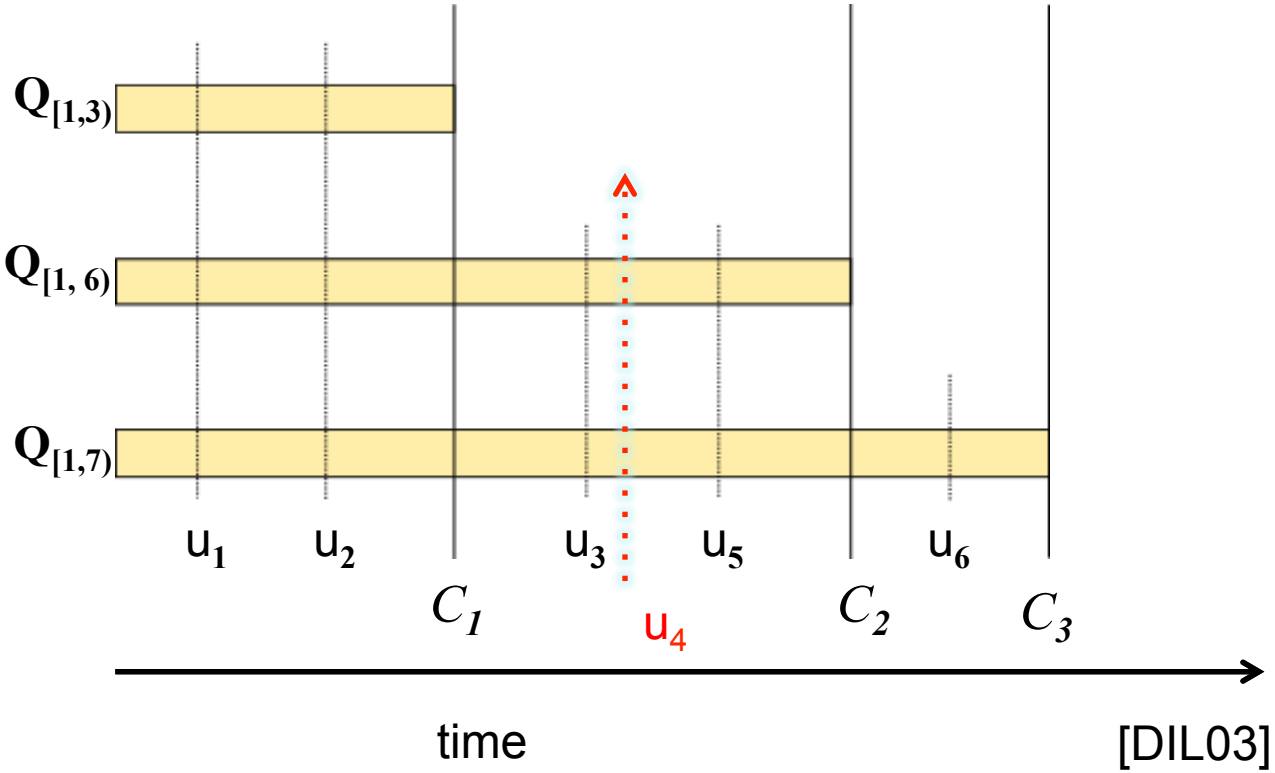
Previous Results
Checkpointing: Query



$O(\sqrt{m})$ overhead for queries

Demaine, Kaler, Liu, Sidford, Yedidia

Previous Results
Checkpointing: Update

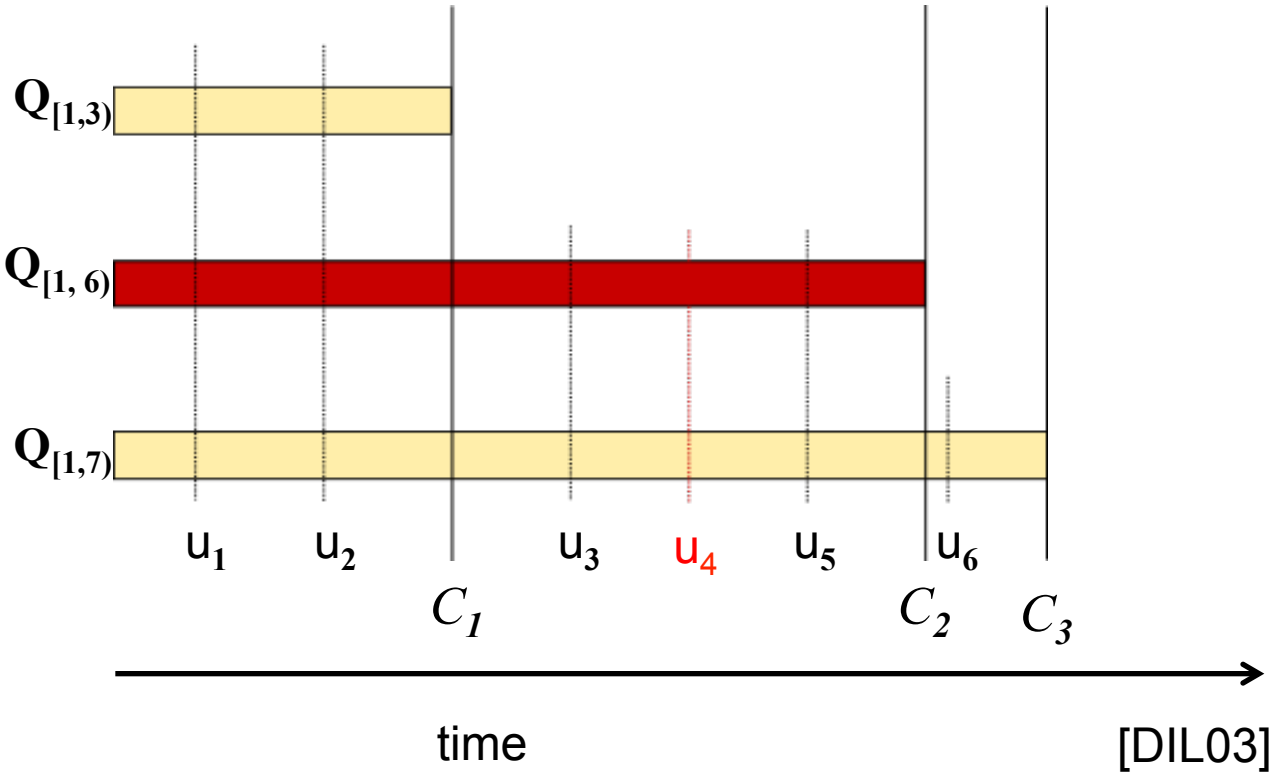


[DIL03]

Insert update into all intersecting partially retroactive priority queues.

Demaine, Kaler, Liu, Sidford, Yedidia

Previous Results
Checkpointing: Update

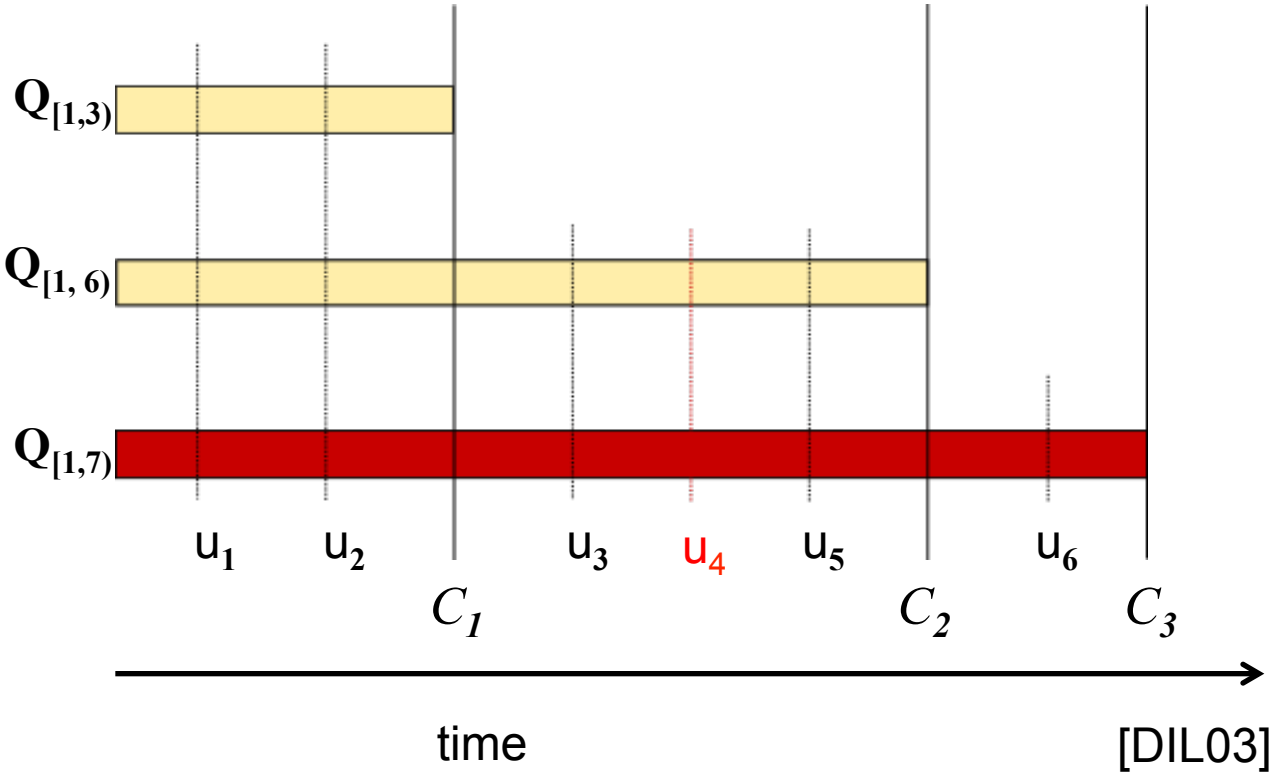


[DIL03]

Insert update into all intersecting partially retroactive priority queues.

Demaine, Kaler, Liu, Sidford, Yedidia

Previous Results
Checkpointing: Update

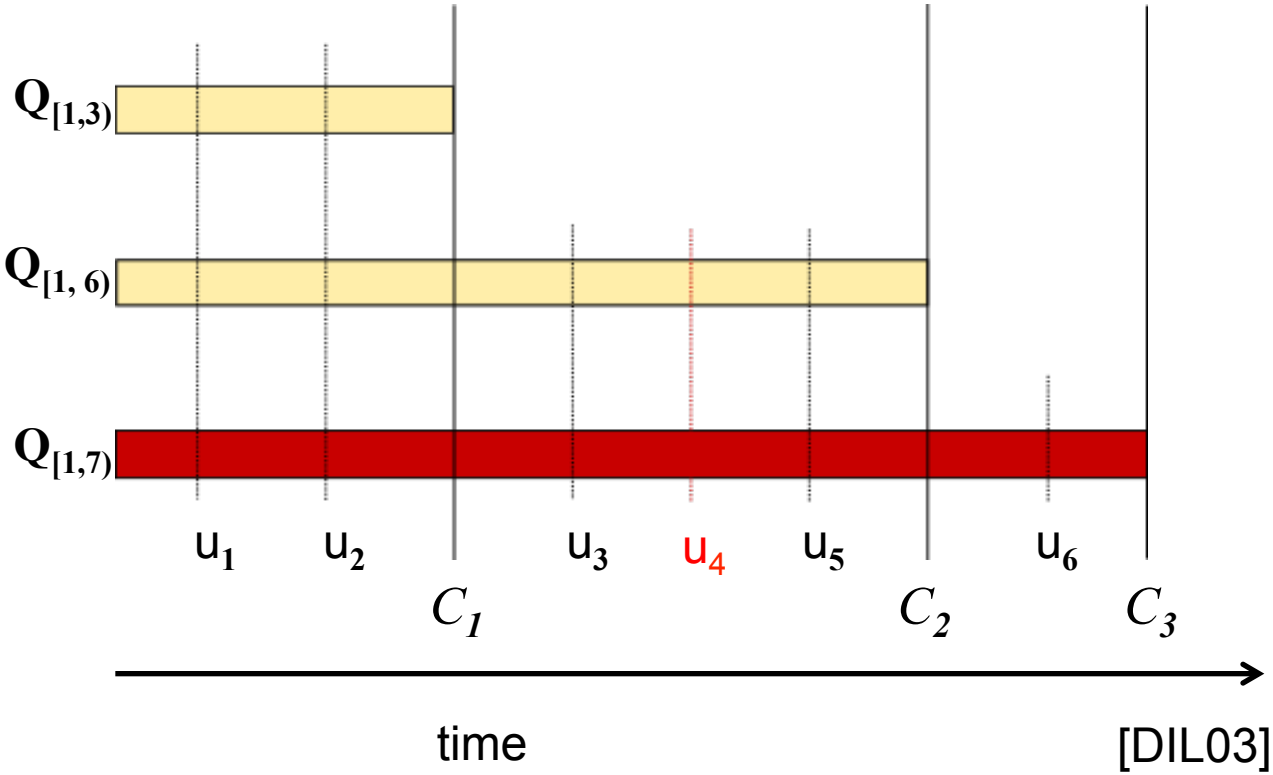


[DIL03]

Insert update into all intersecting partially retroactive priority queues.

Demaine, Kaler, Liu, Sidford, Yedidia

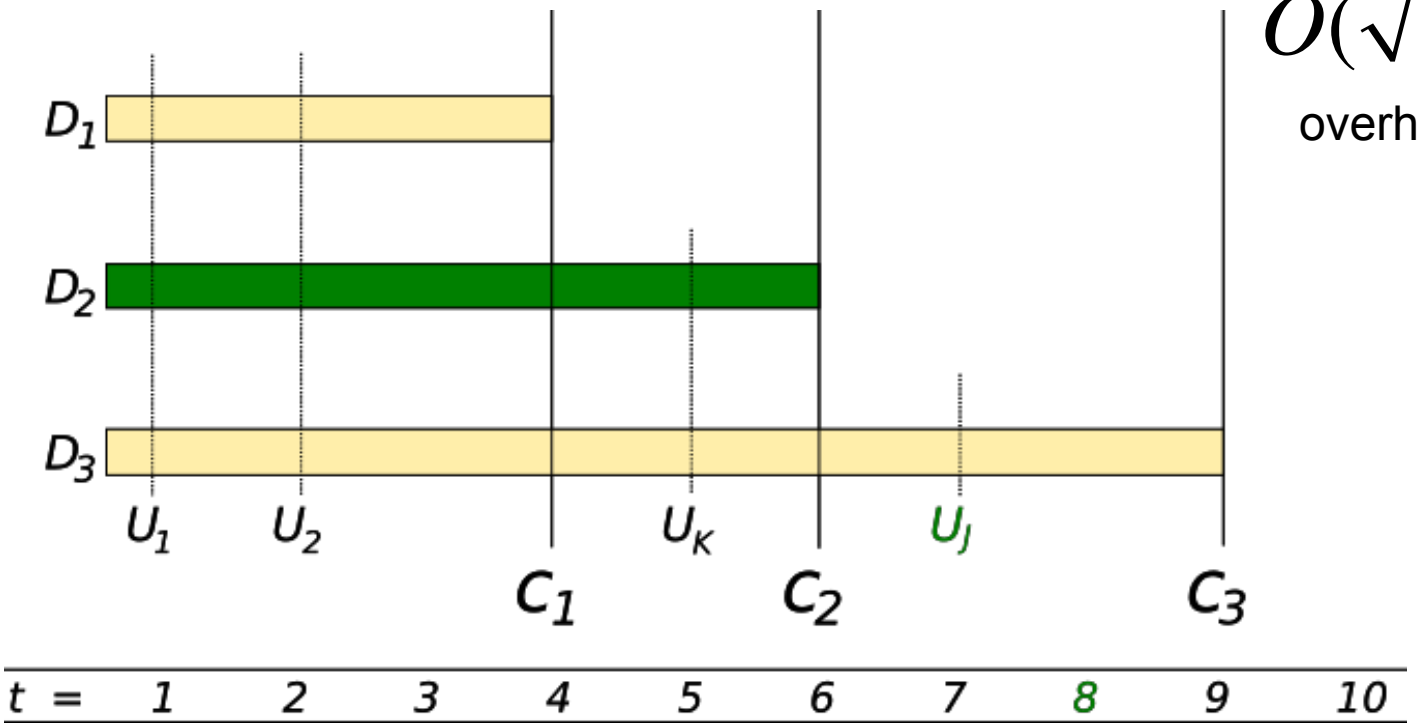
Previous Results
Checkpointing: Update



$O(\sqrt{m})$ overhead on update time

Demaine, Kaler, Liu, Sidford, Yedidia

Previous Results
Checkpointing: Query



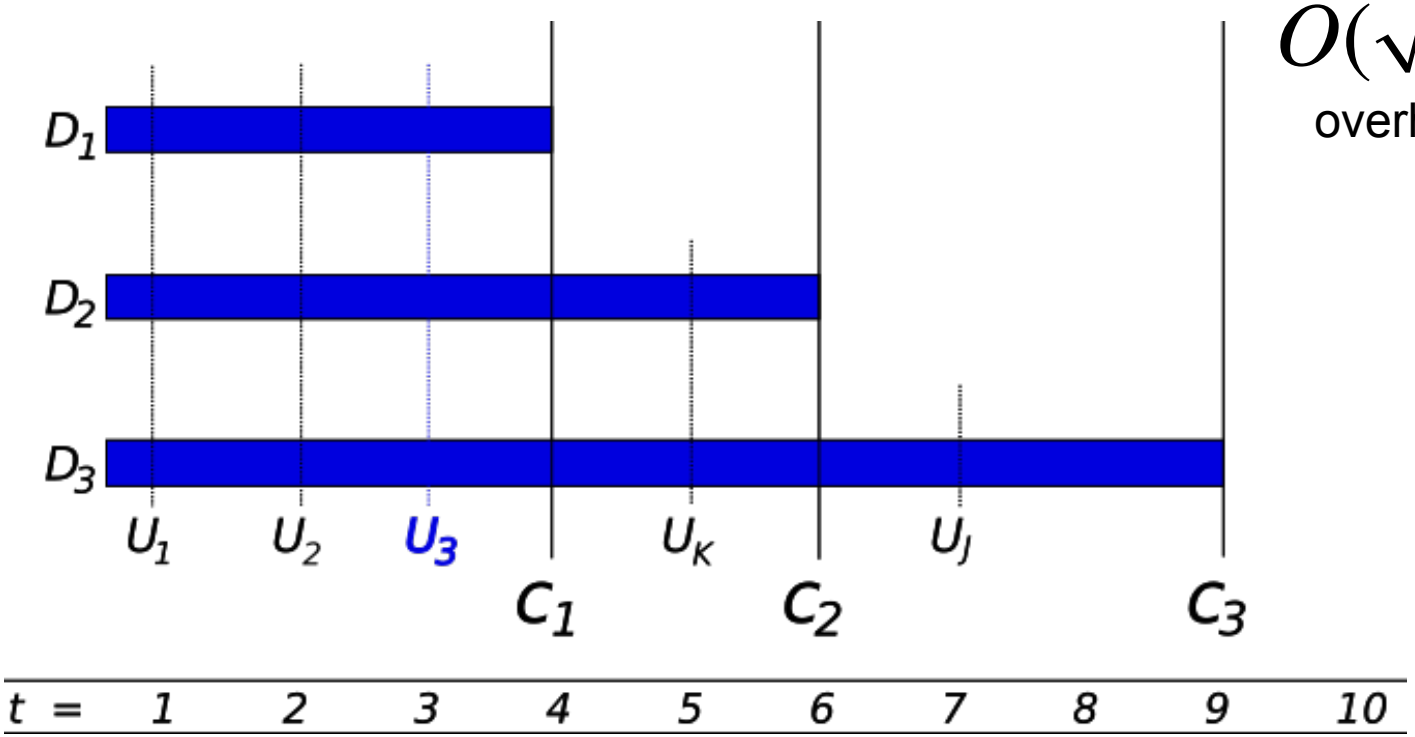
$O(\sqrt{m})$

overhead

Find-Min(8) Demaine, Kaler, Liu, Sidford, Yedidia

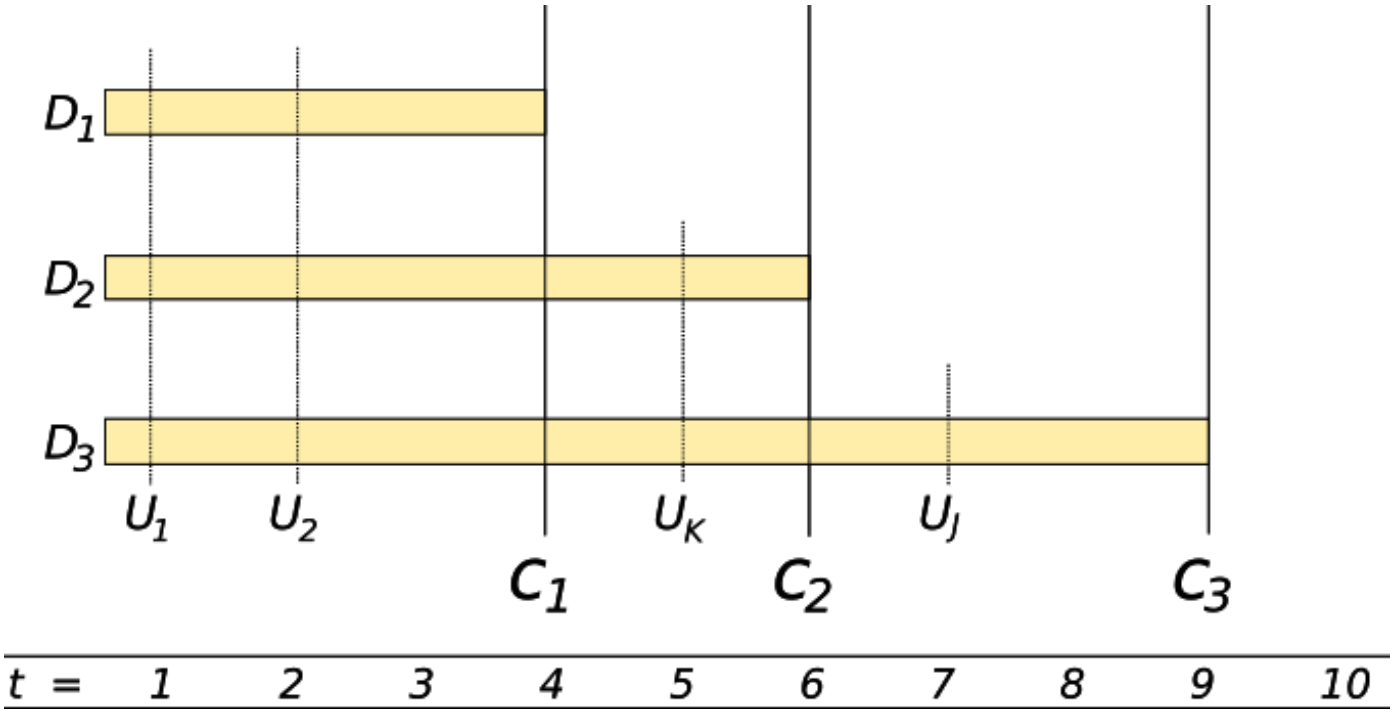
Previous Results
Checkpointing: Update

- May need to update $\sqrt{m}$ partially retroactive priority queues.



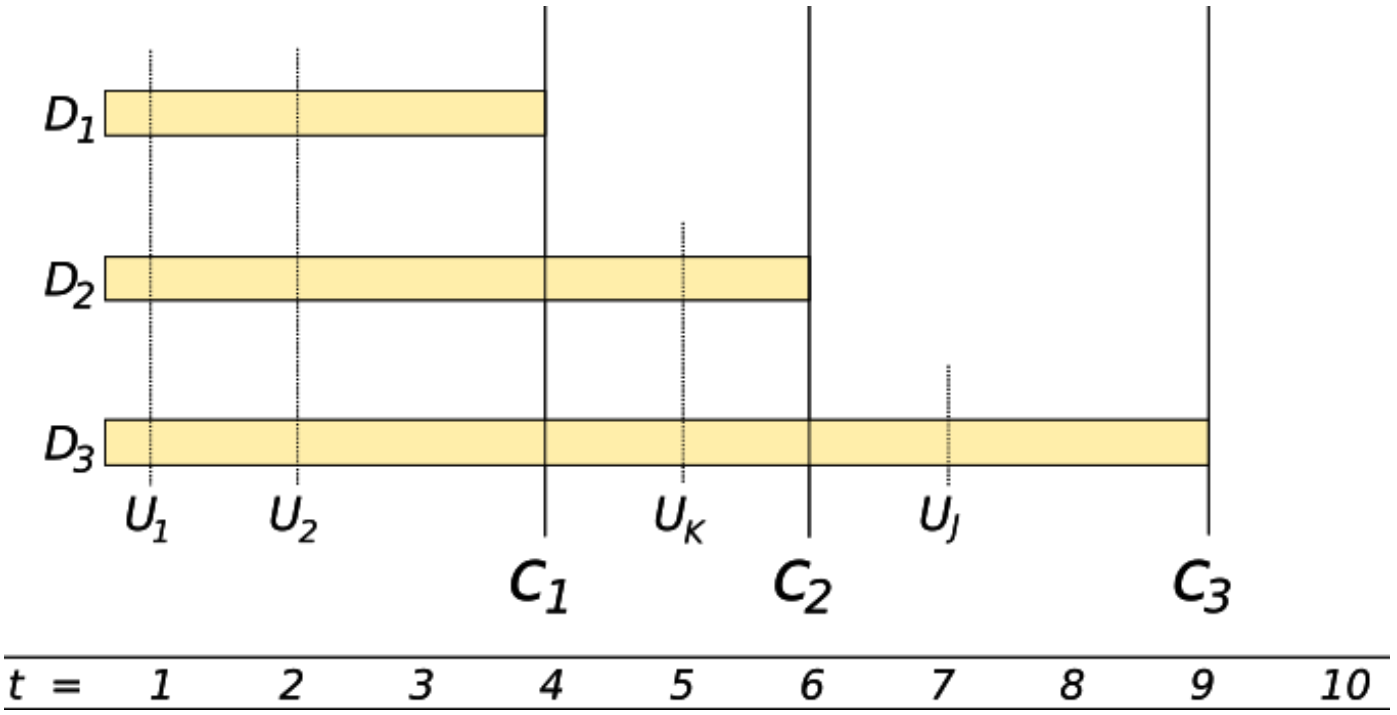
Insert-Op(3, U_3)

Checkpointing

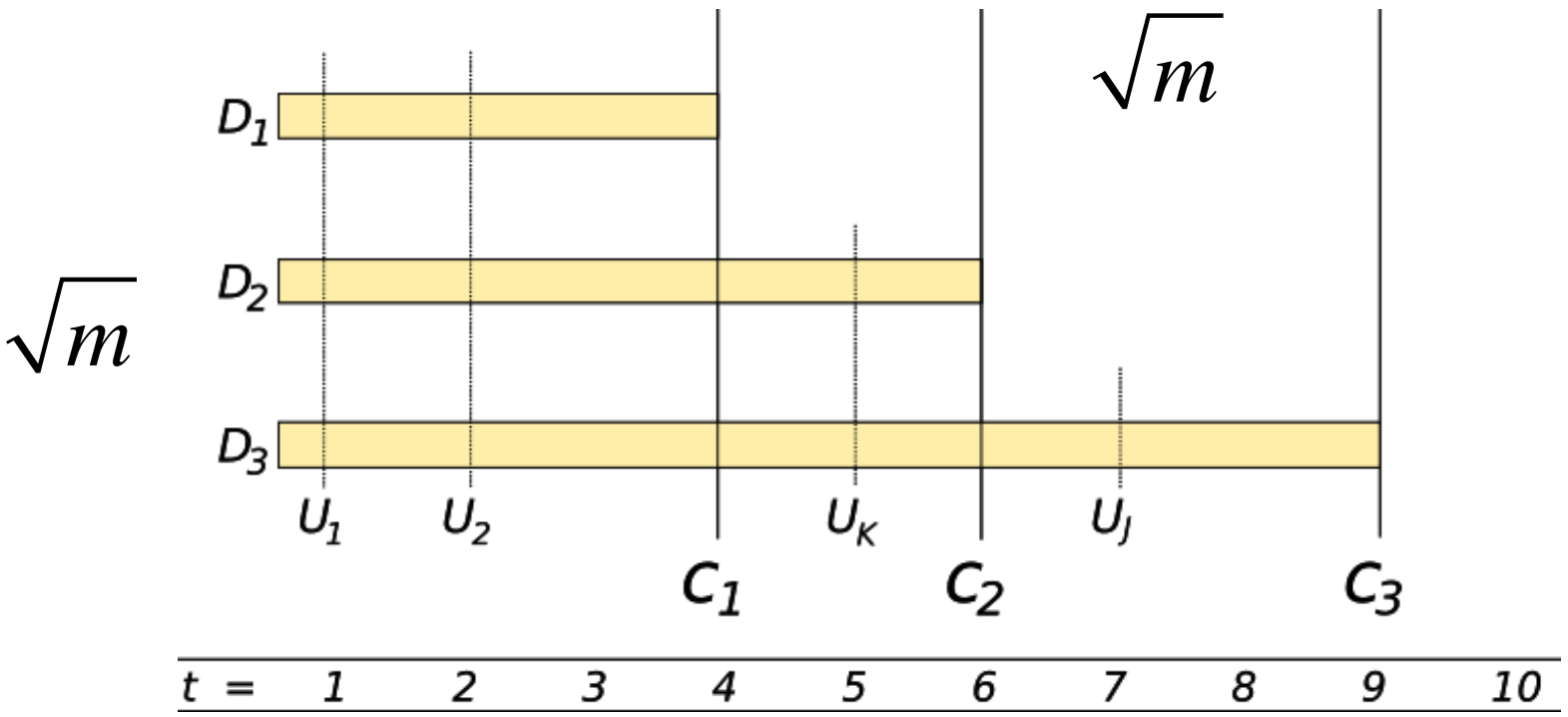


Checkpointing

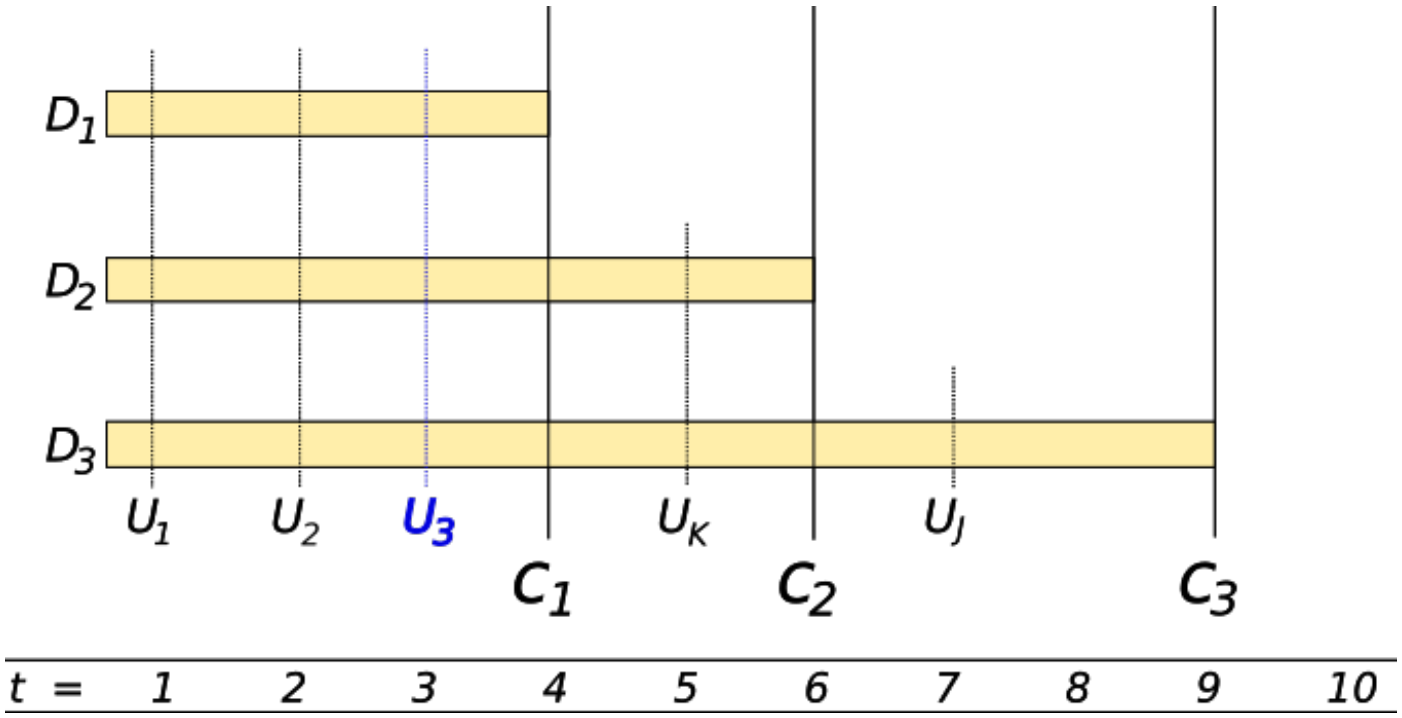
$\sqrt{m}$



Checkpointing

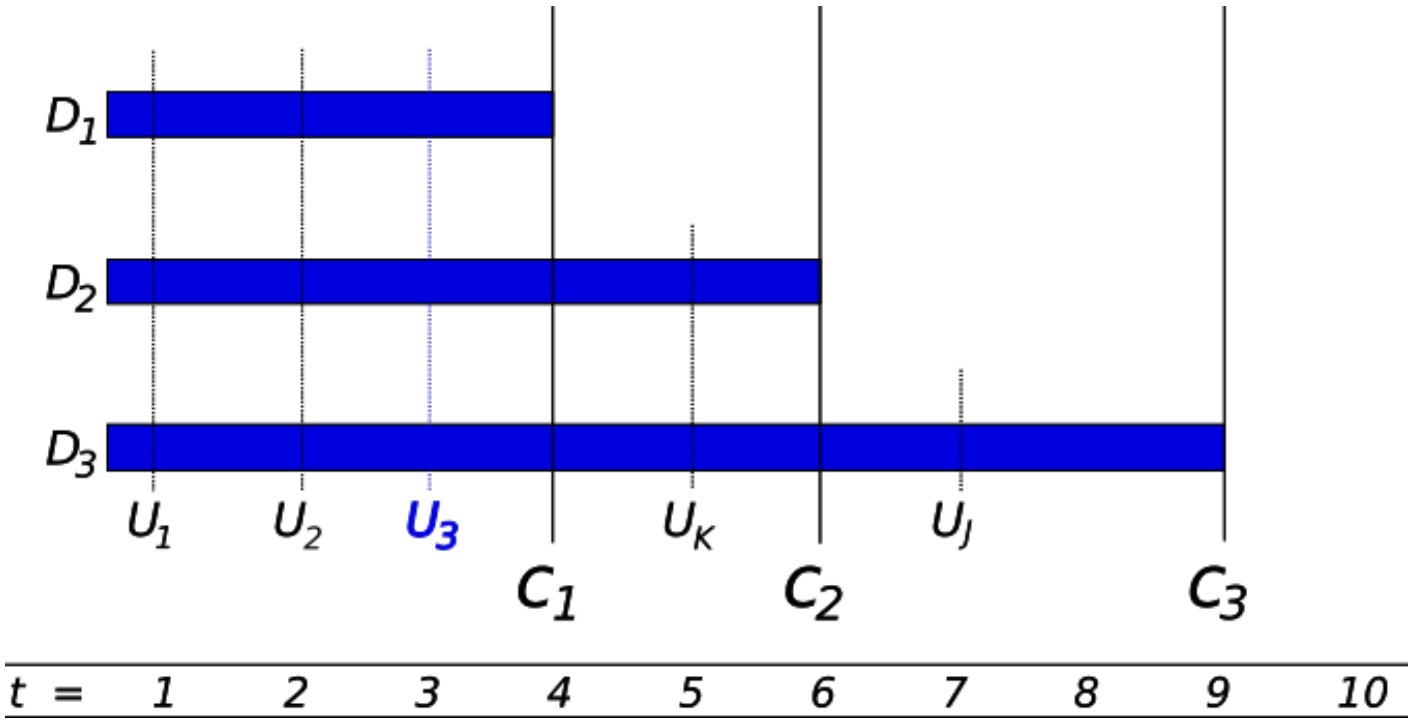


Checkpointing



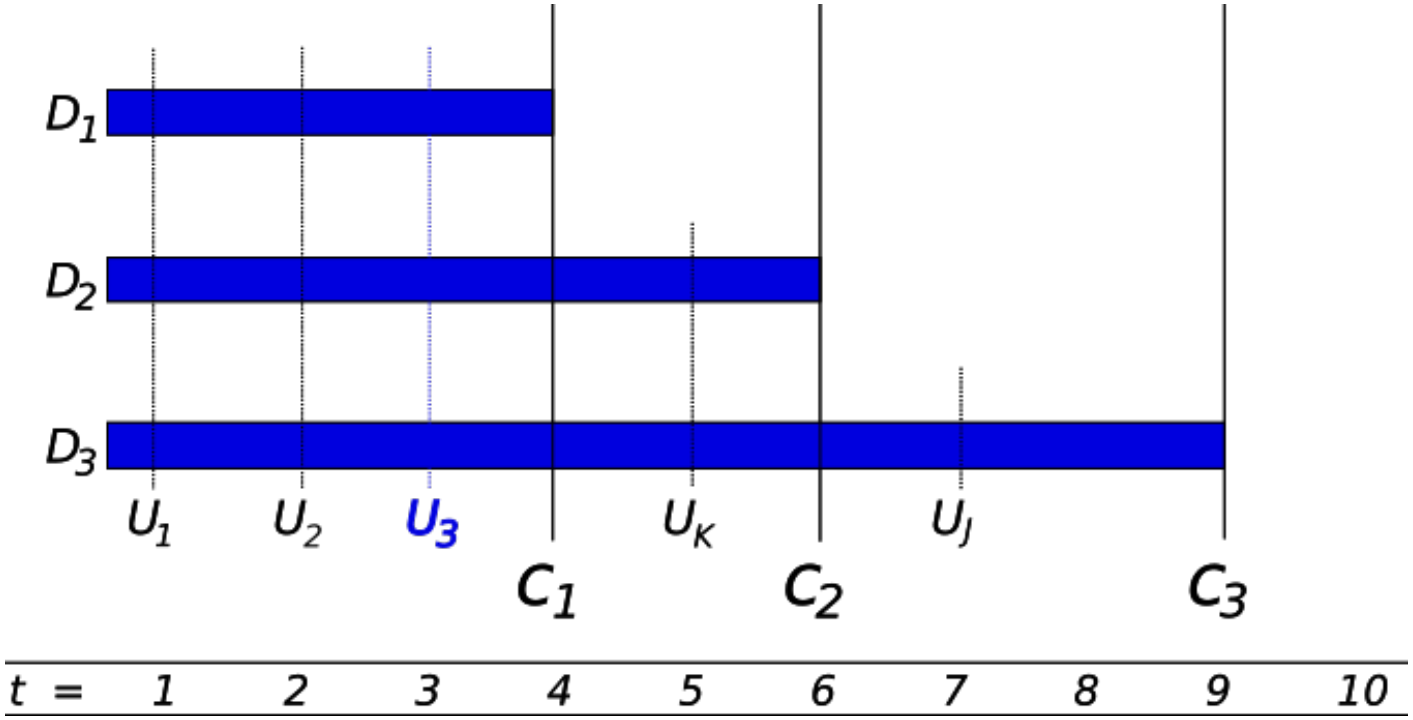
Insert-Op(3, U)

Checkpointing



Insert-Op(3, U)

Checkpointing

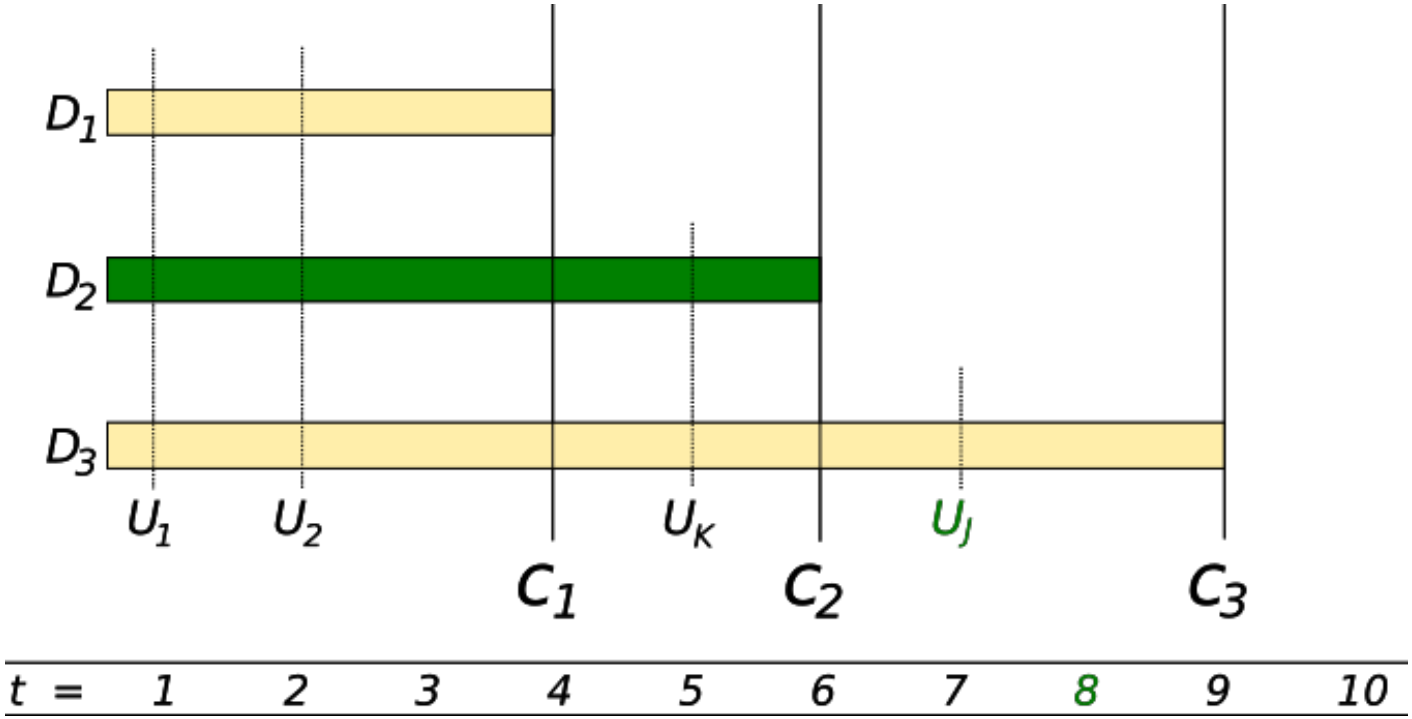


Insert-Op(3, U)

$O(\sqrt{m})$ overhead

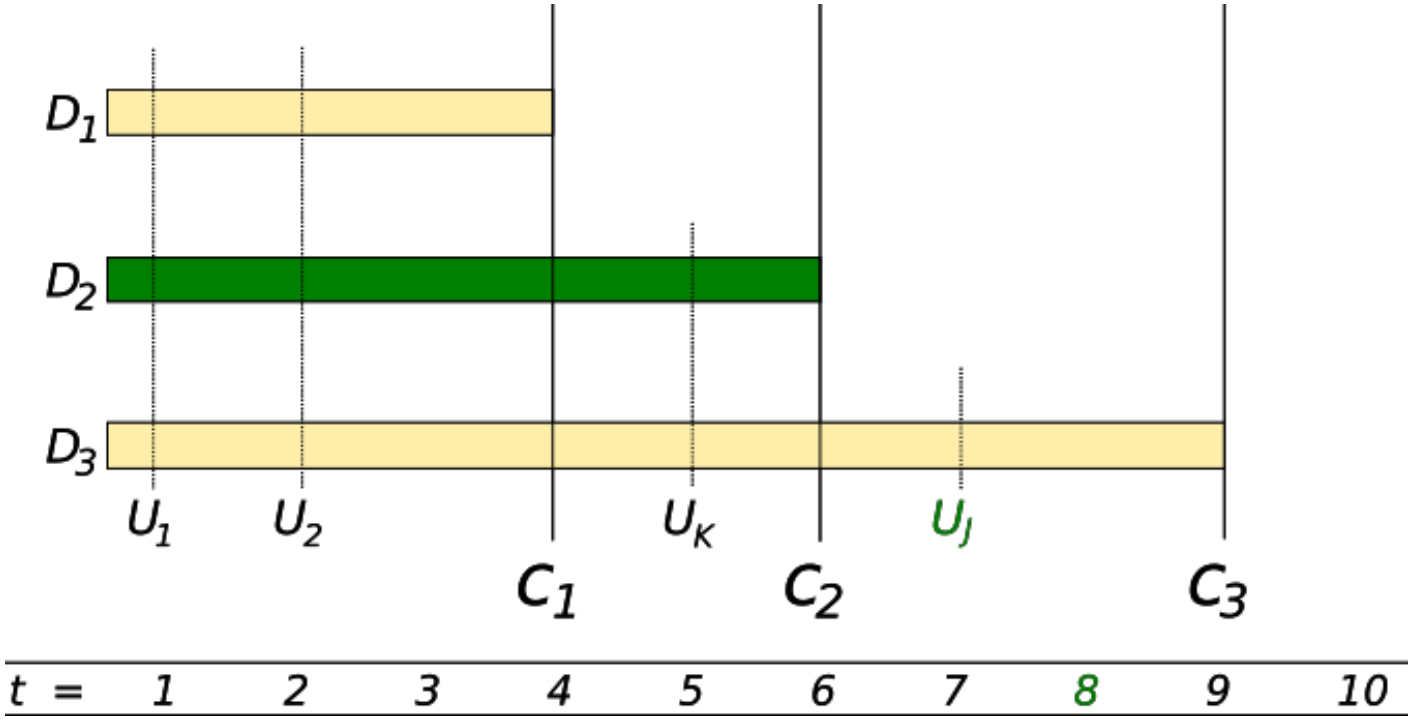
Demaine, Kaler, Liu, Sidford, Yedidia

Checkpointing



Get-View(8)

Checkpointing

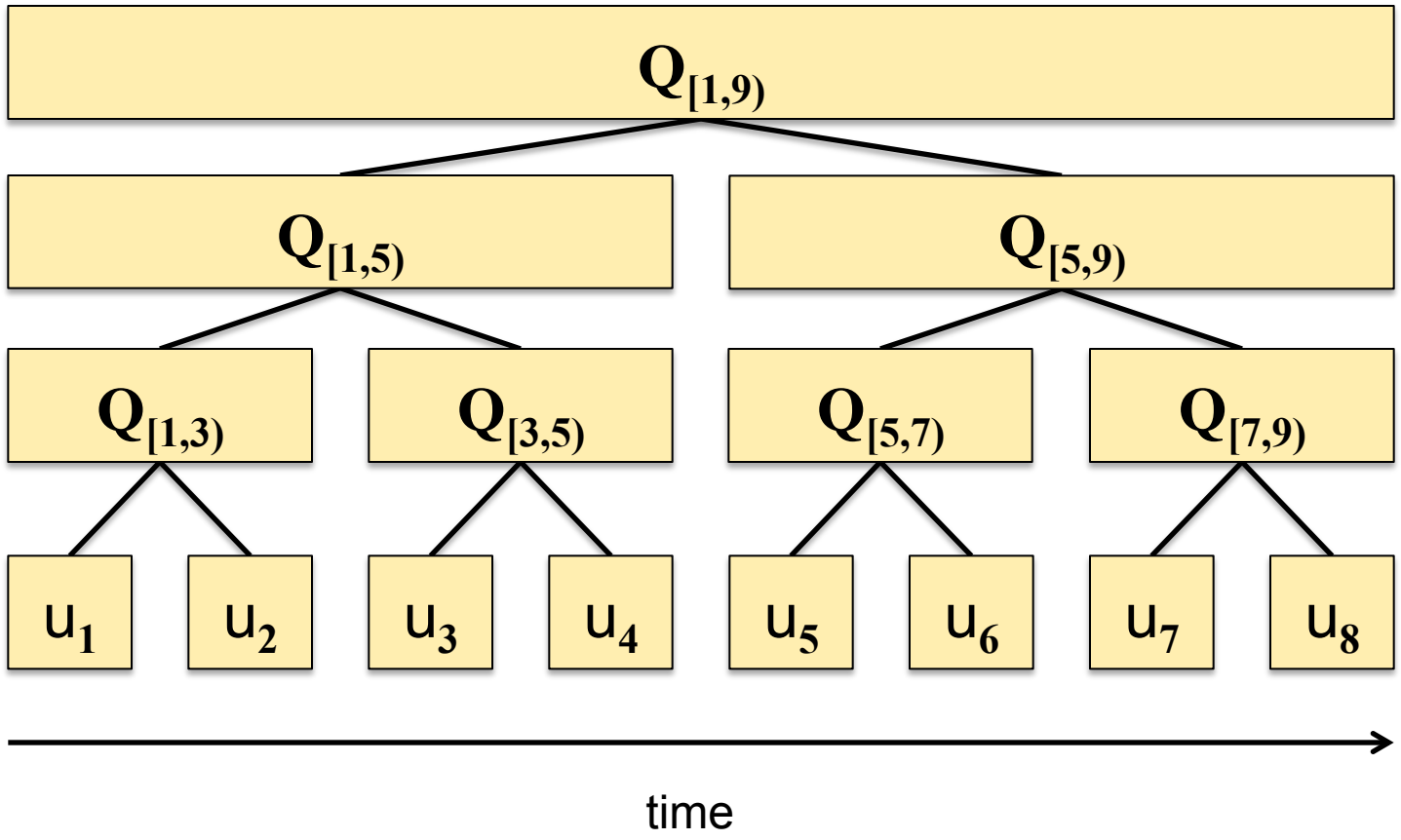


Find-Min(8)

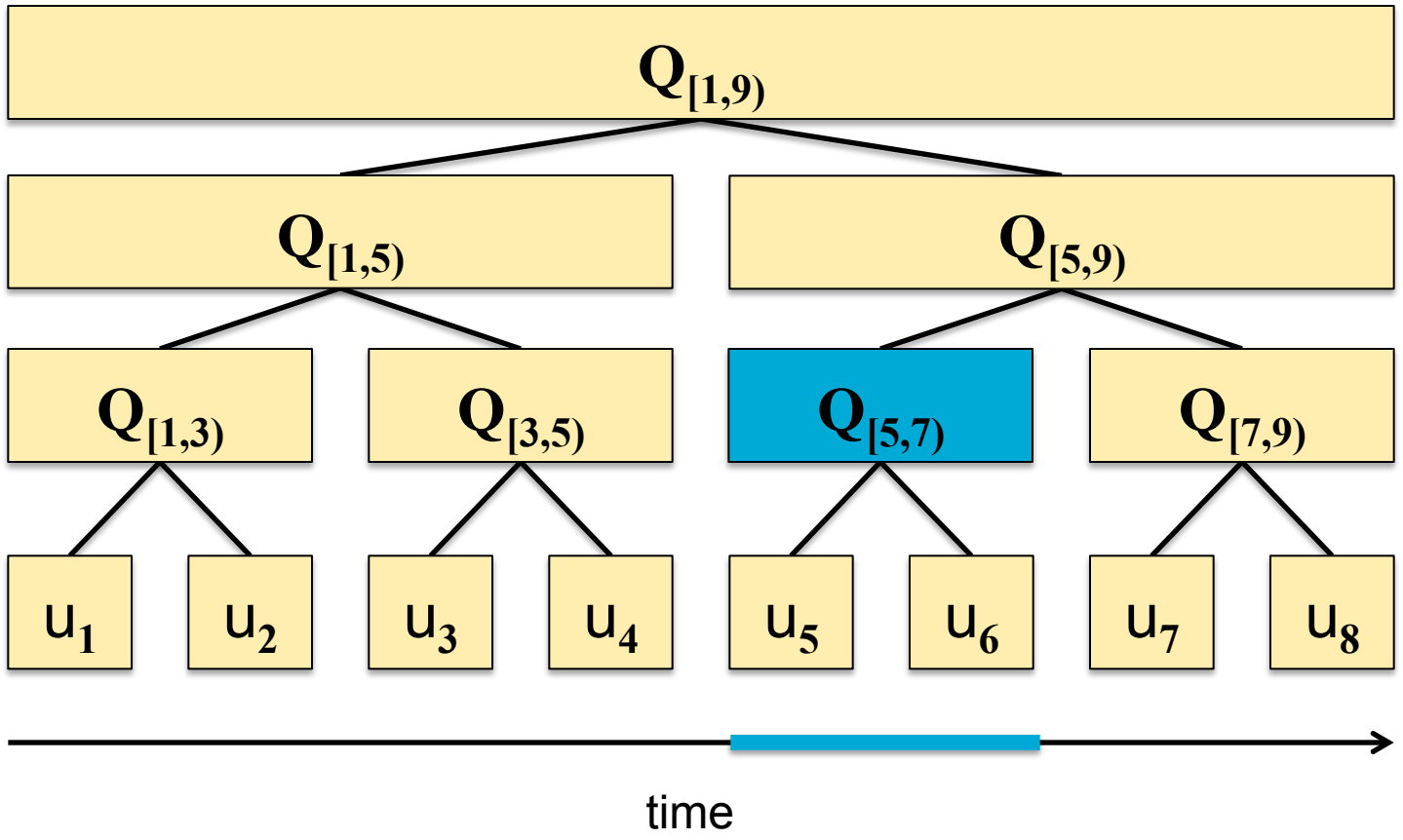
$O(\sqrt{m})$ overhead

Demaine, Kaler, Liu, Sidford, Yedidia

Hierarchical Checkpointing



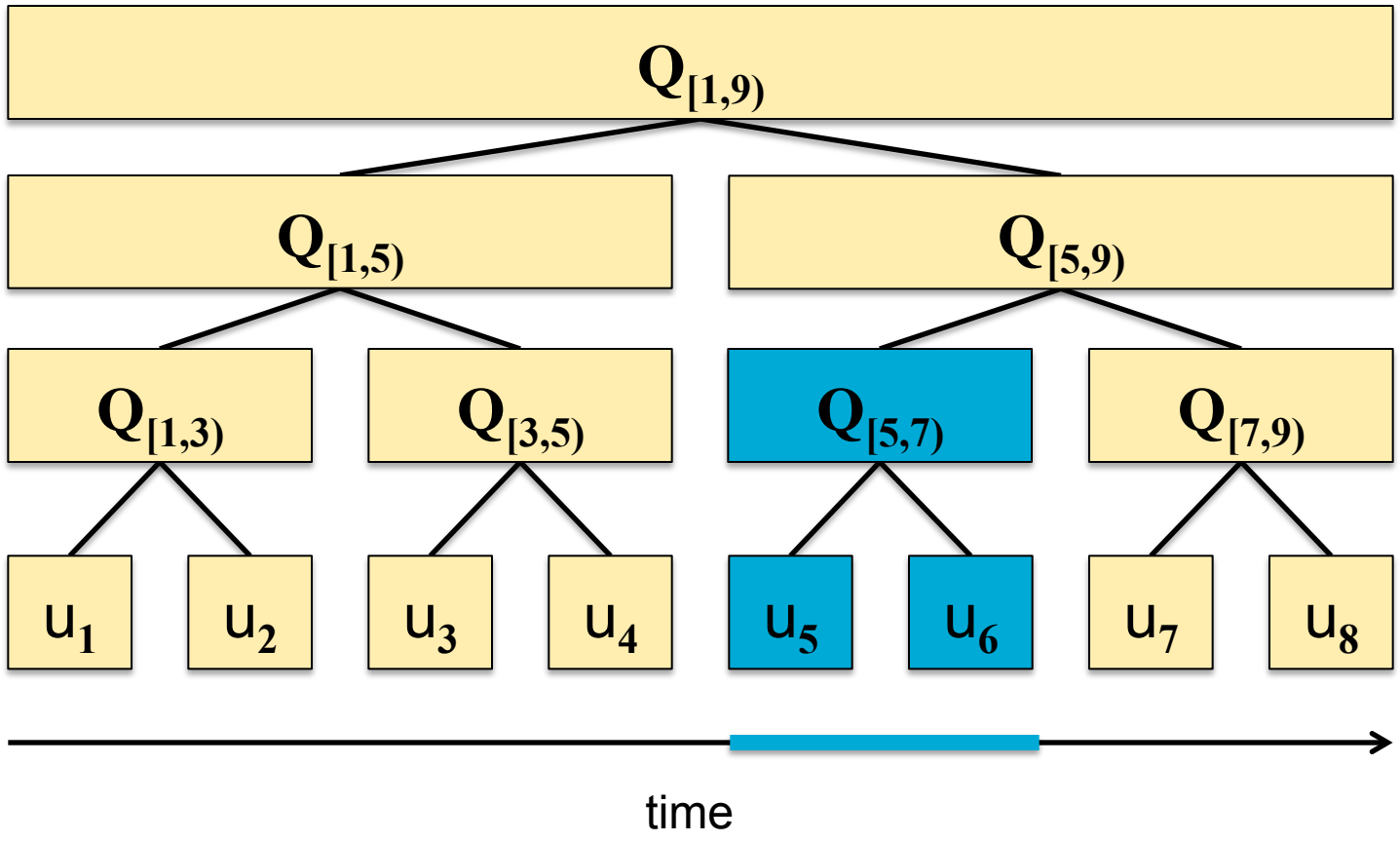
Hierarchical Checkpointing



Inner nodes contains partially retroactive priority queues spanning segment of time.

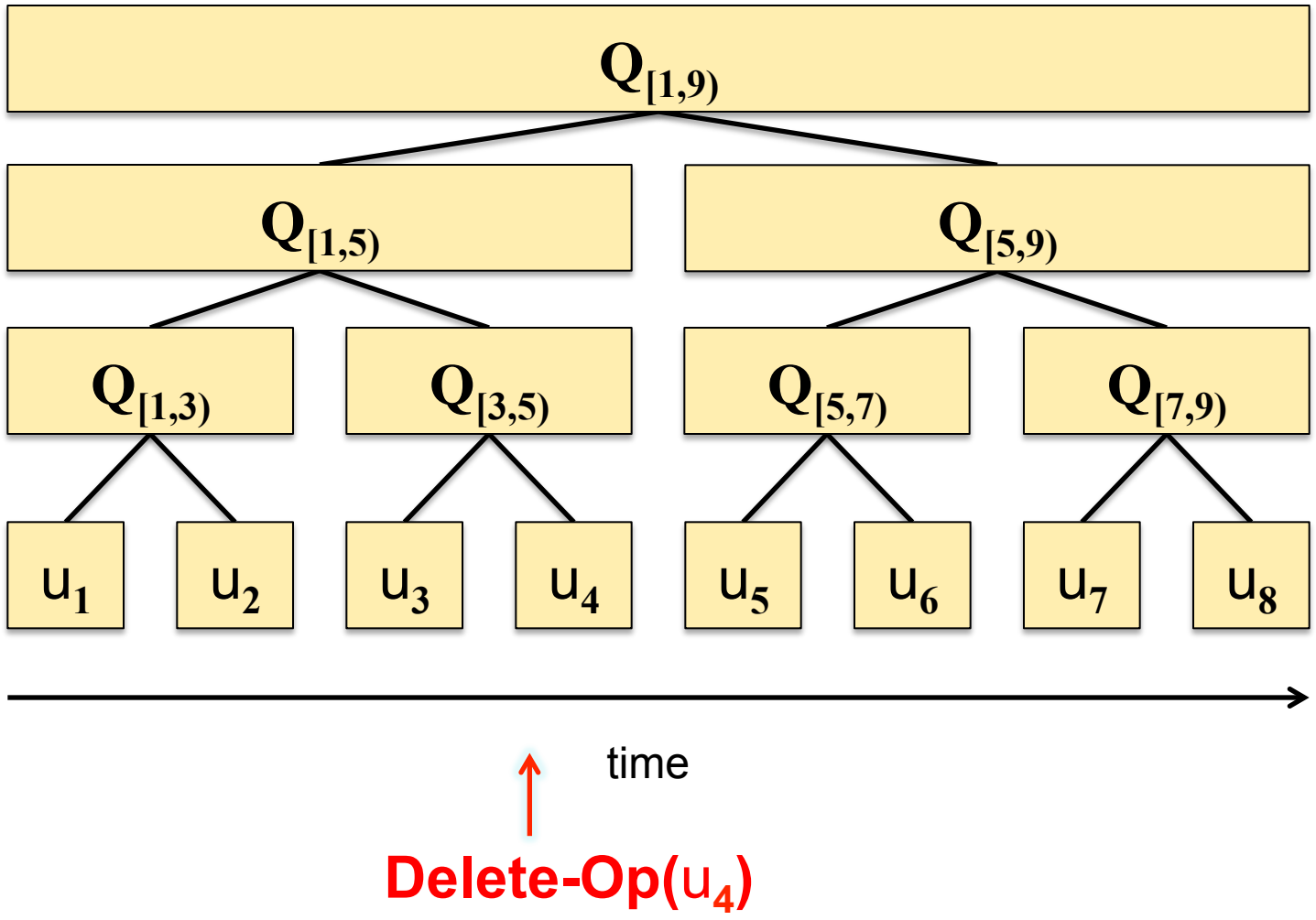
Demaine, Kaler, Liu, Sidford, Yedidia

Hierarchical Checkpointing

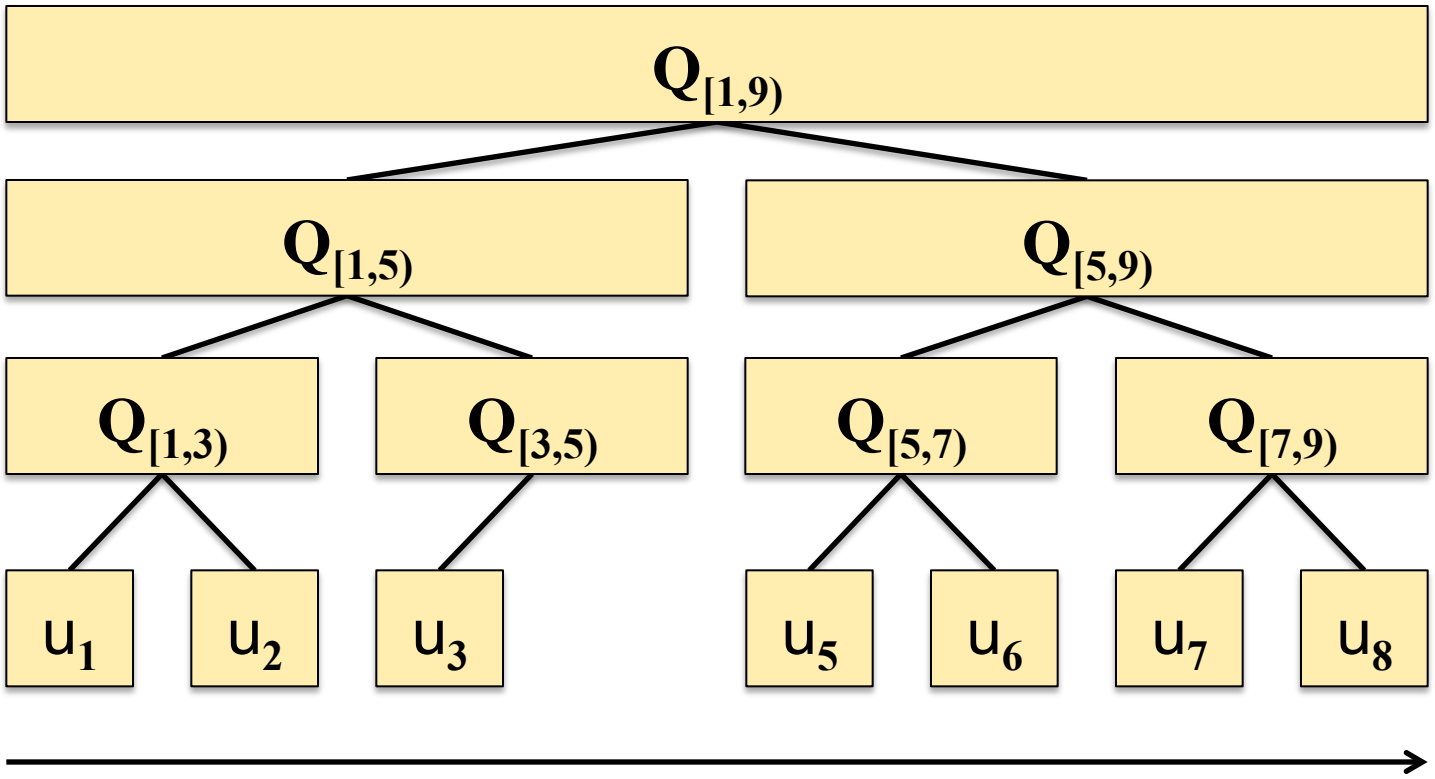


Each node contains updates in leaves of subtree.

Hierarchical Checkpointing: Update

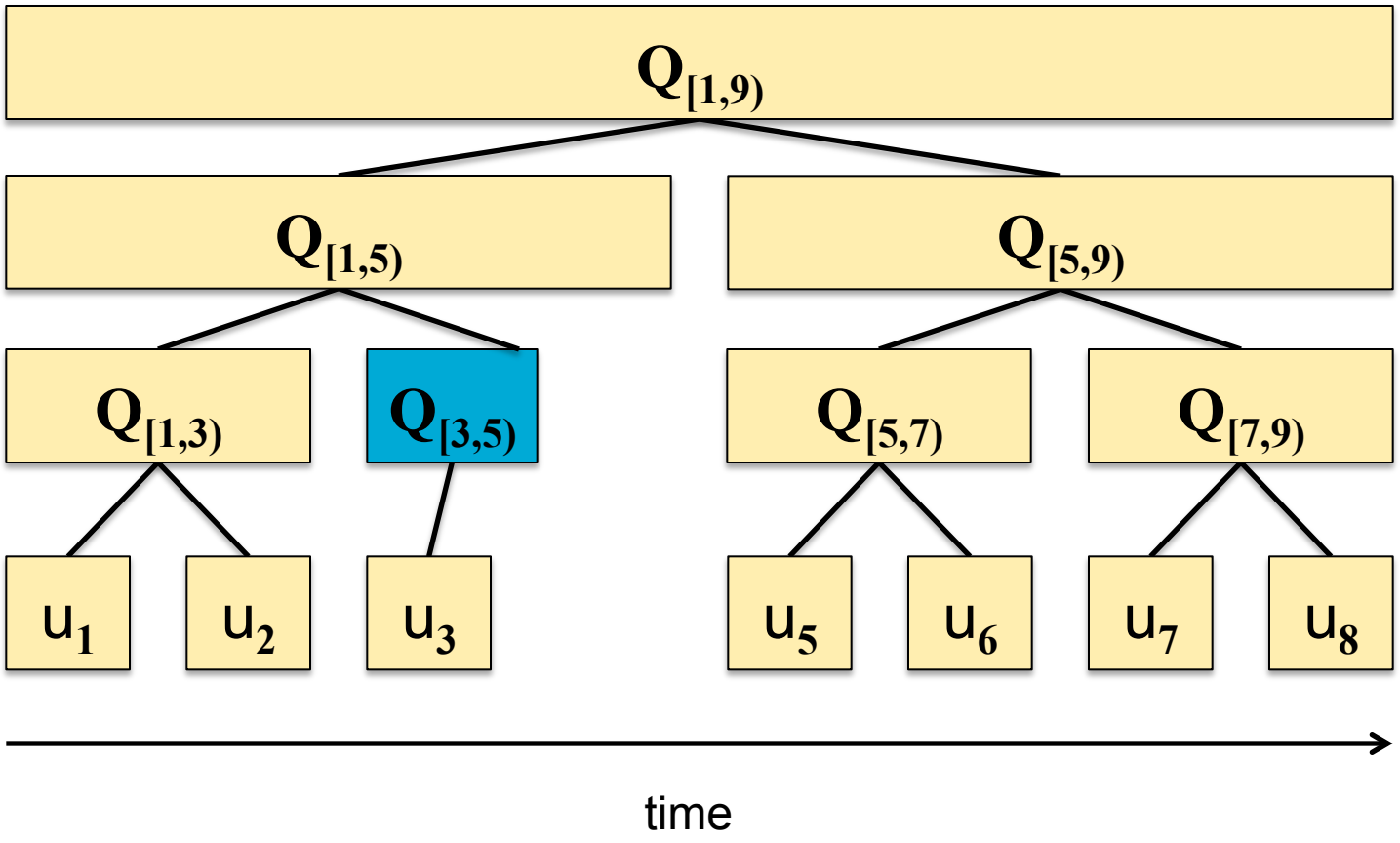


Hierarchical Checkpointing: Update



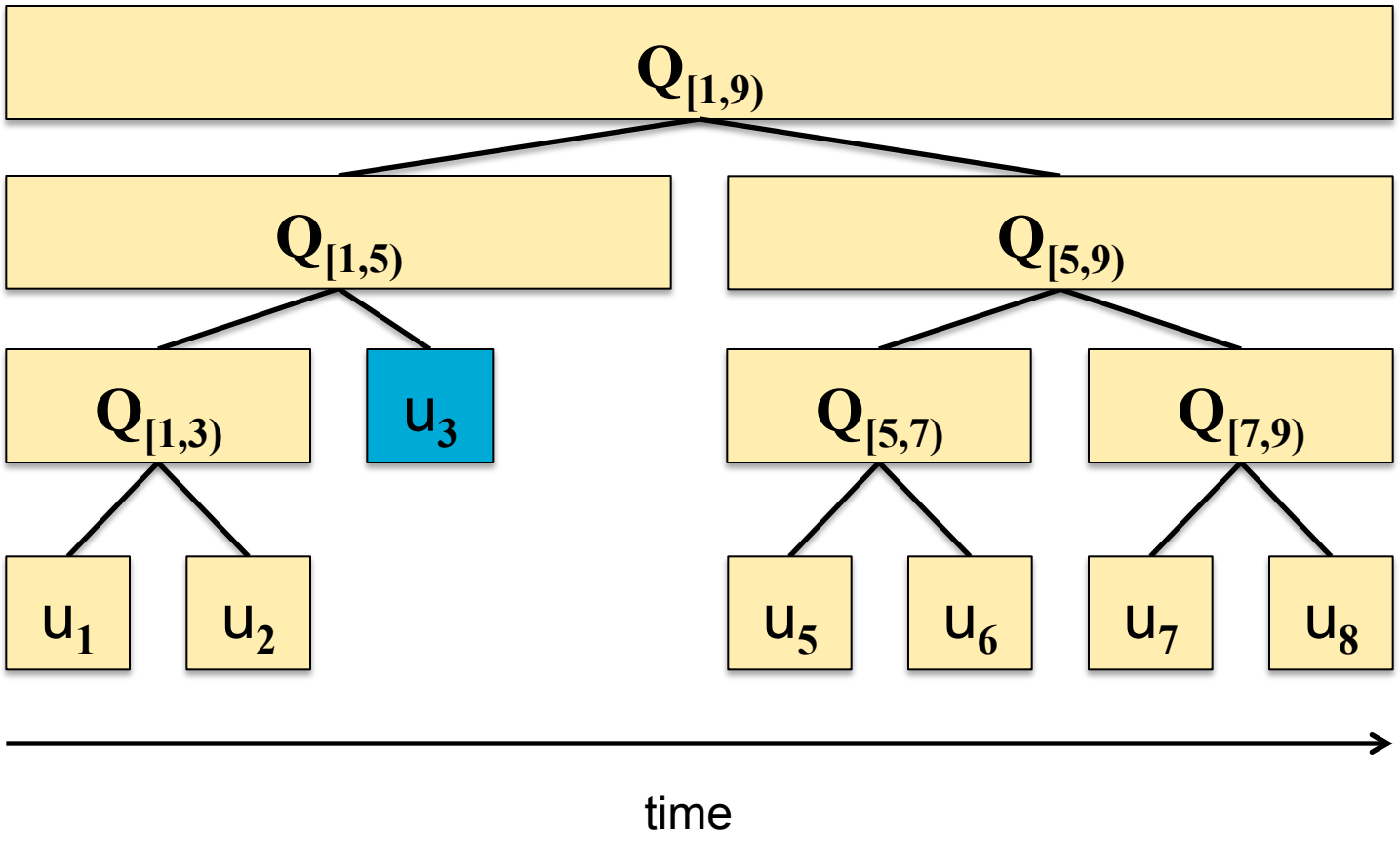
Delete-Op(u_4)

Hierarchical Checkpointing: Update



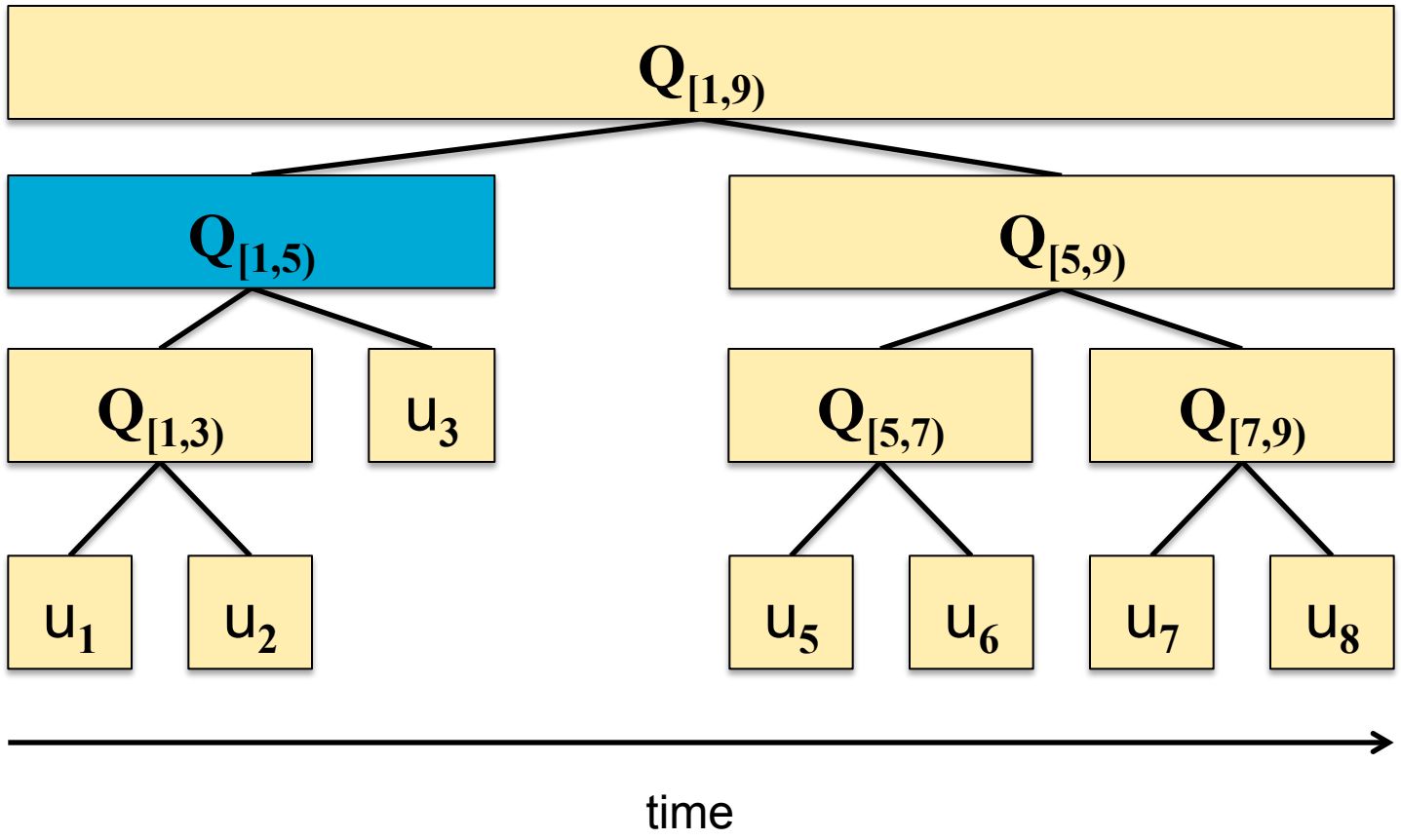
Delete-Op(u_4)

Hierarchical Checkpointing: Update



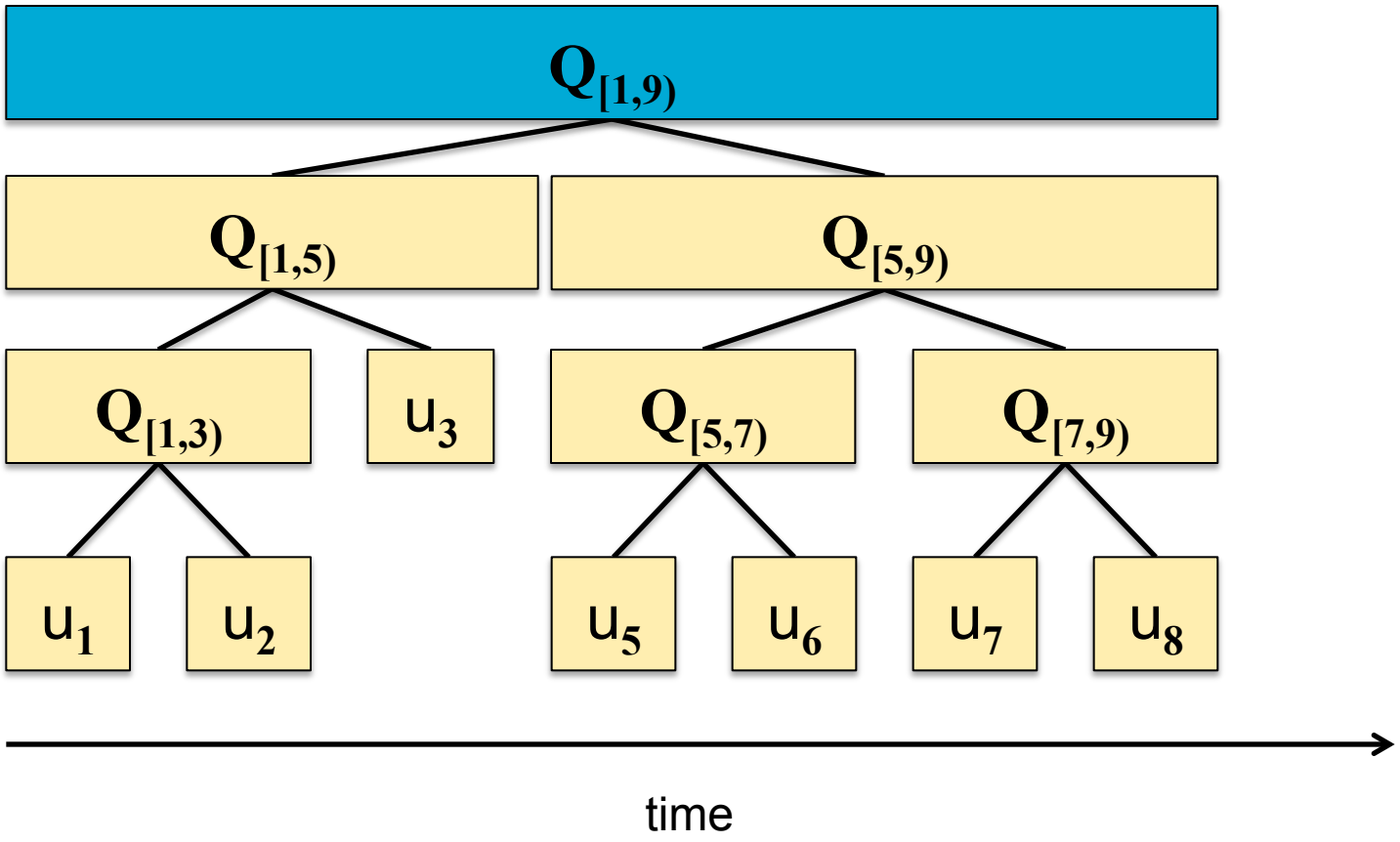
Delete-Op(u_4)

Hierarchical Checkpointing: Update



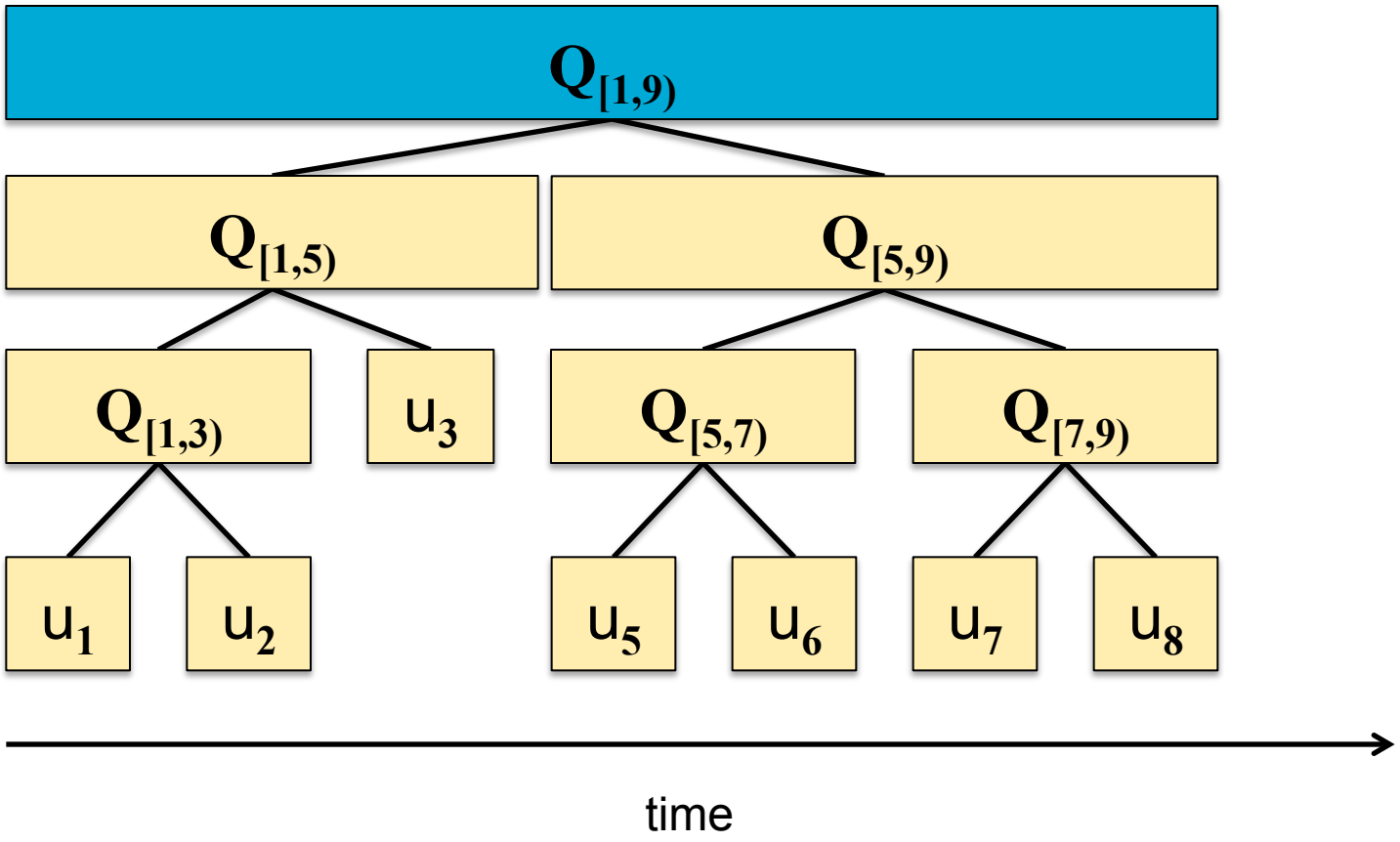
Delete-Op(u_4)

Hierarchical Checkpointing: Update



Delete-Op(u_4)

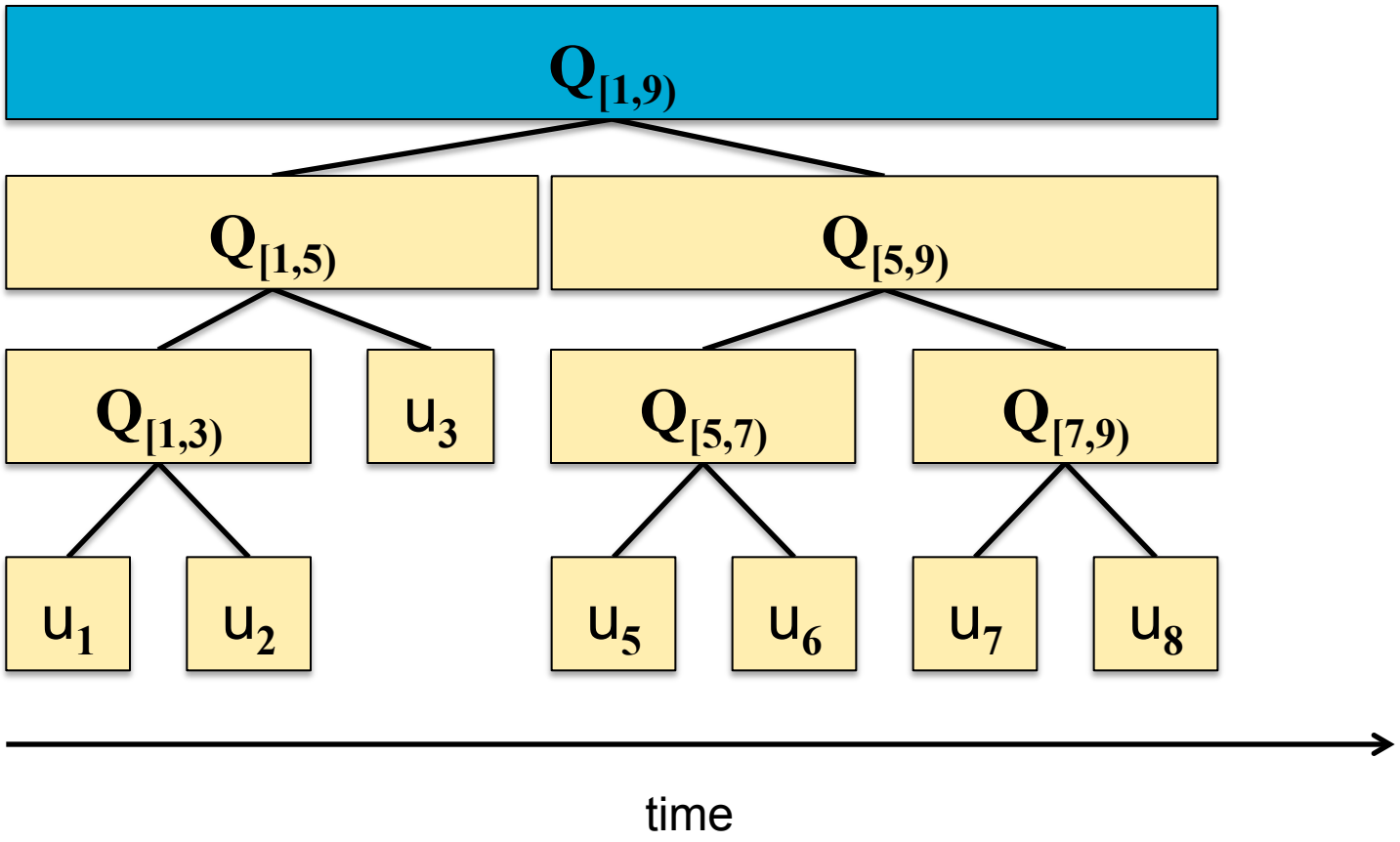
Hierarchical Checkpointing: Update



update $O(\log m)$ partially retroactive priority queue ancestors

Demaine, Kaler, Liu, Sidford, Yedidia

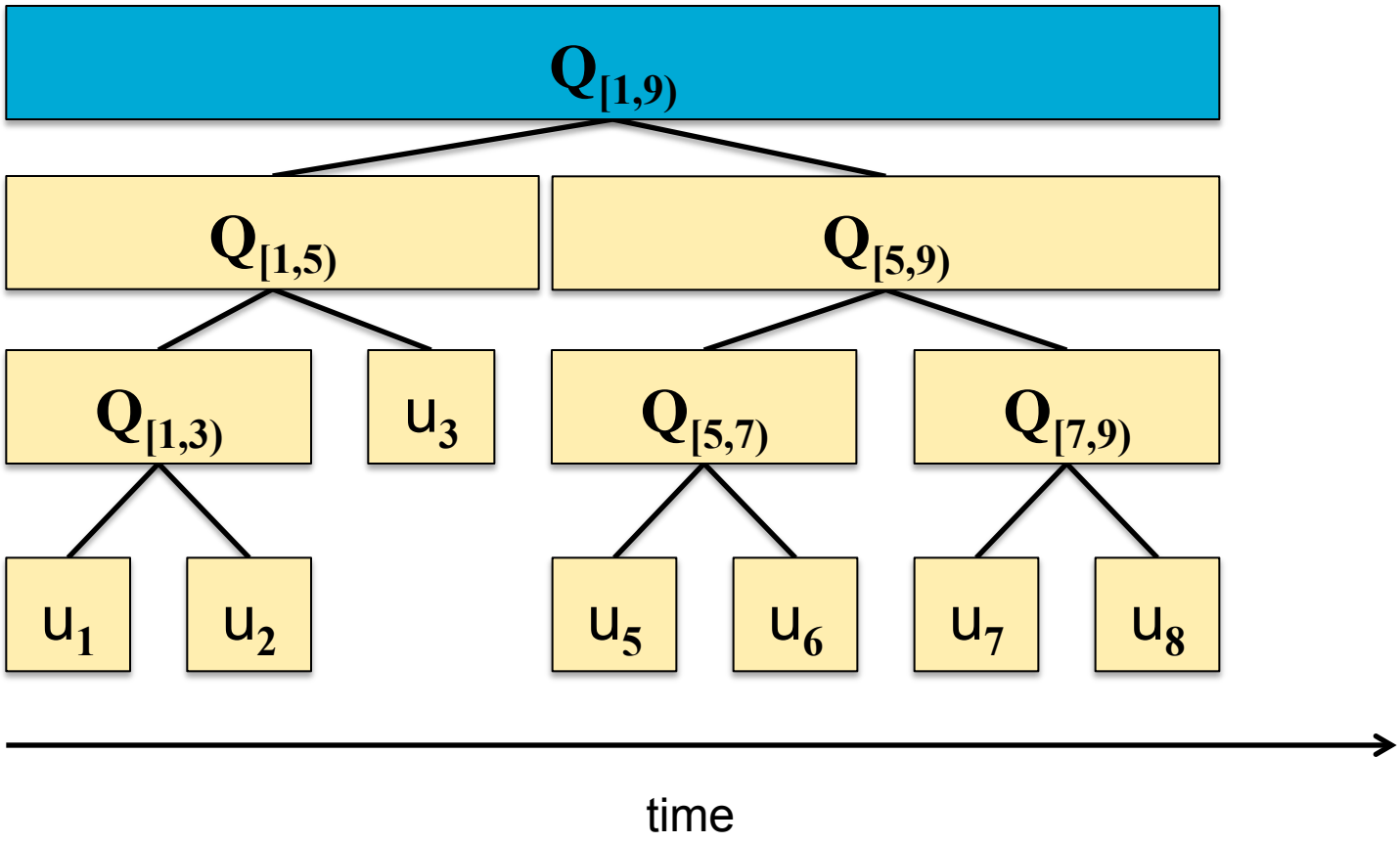
Hierarchical Checkpointing: Update



$O(\log^2 m)$ overhead with extra factor from tree rebuild

Demaine, Kaler, Liu, Sidford, Yedidia

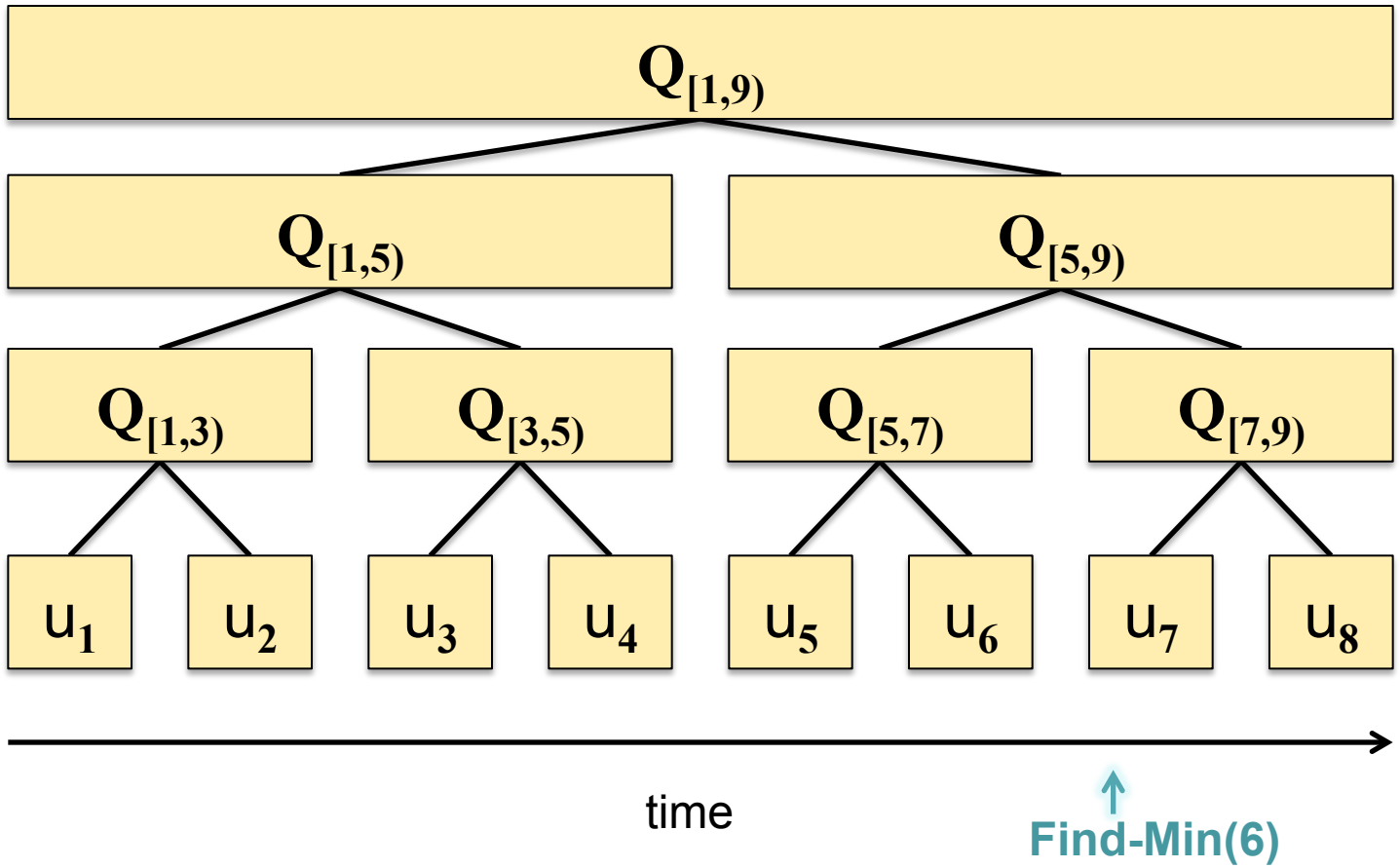
Hierarchical Checkpointing: Update



$O(\log^3 m)$ update time in fully retroactive priority queue

Demaine, Kaler, Liu, Sidford, Yedidia

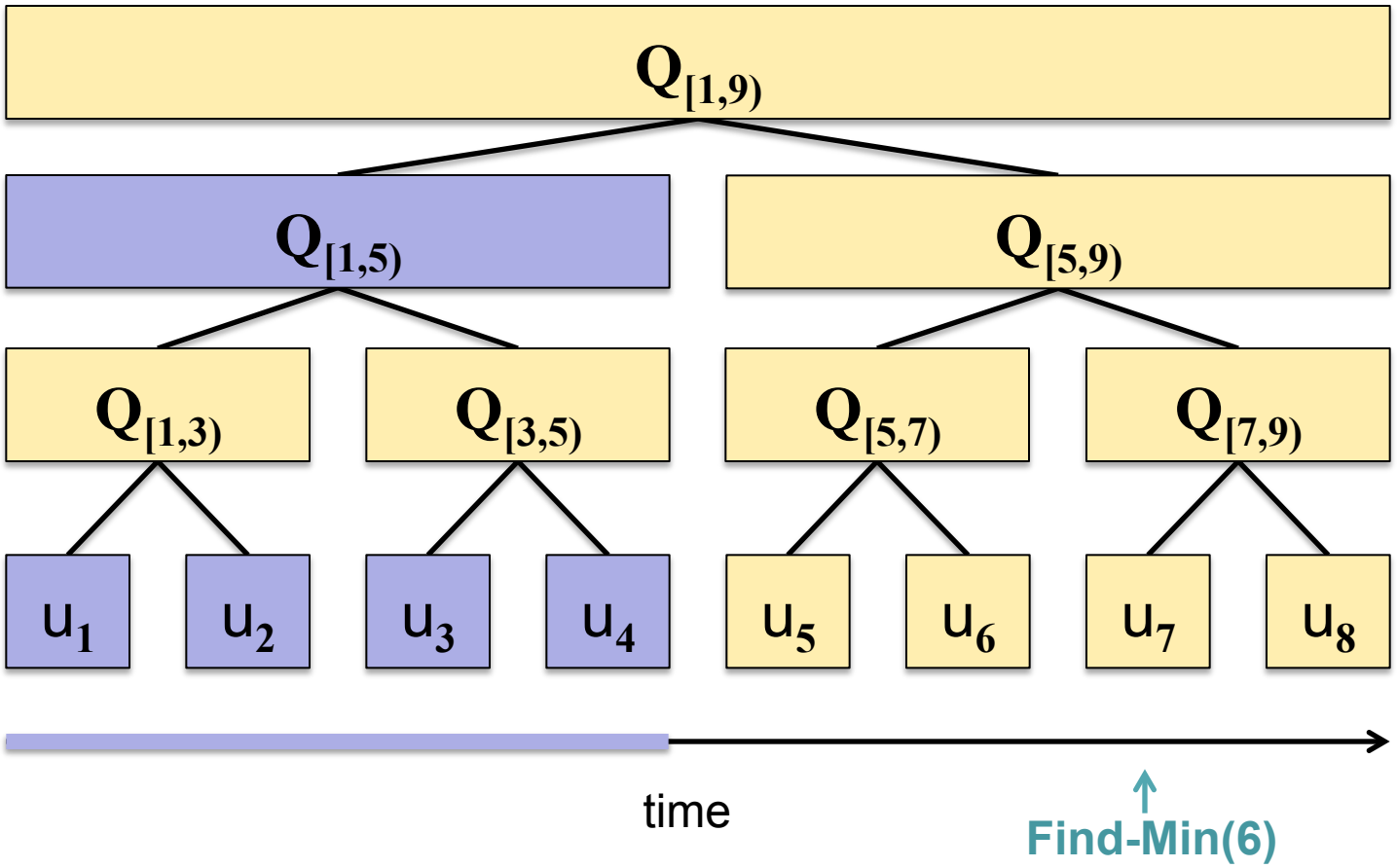
Hierarchical Checkpointing: Query



To query, merge structures that span disjoint, contiguous intervals of time to produce read-only structure.

Demaine, Kaler, Liu, Sidford, Yedidia

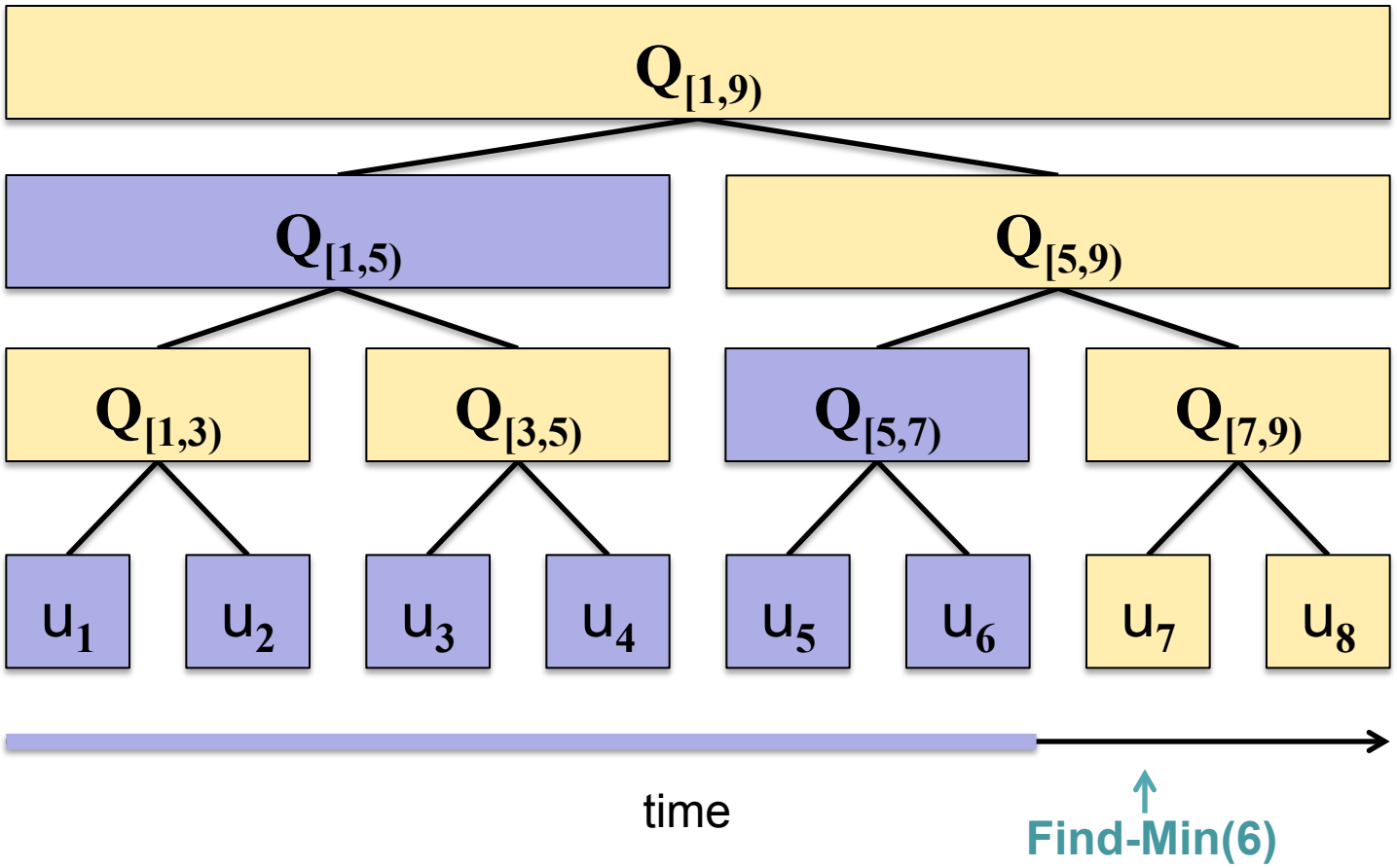
Hierarchical Checkpointing: Query



Merge disjoint, contiguous intervals of time.

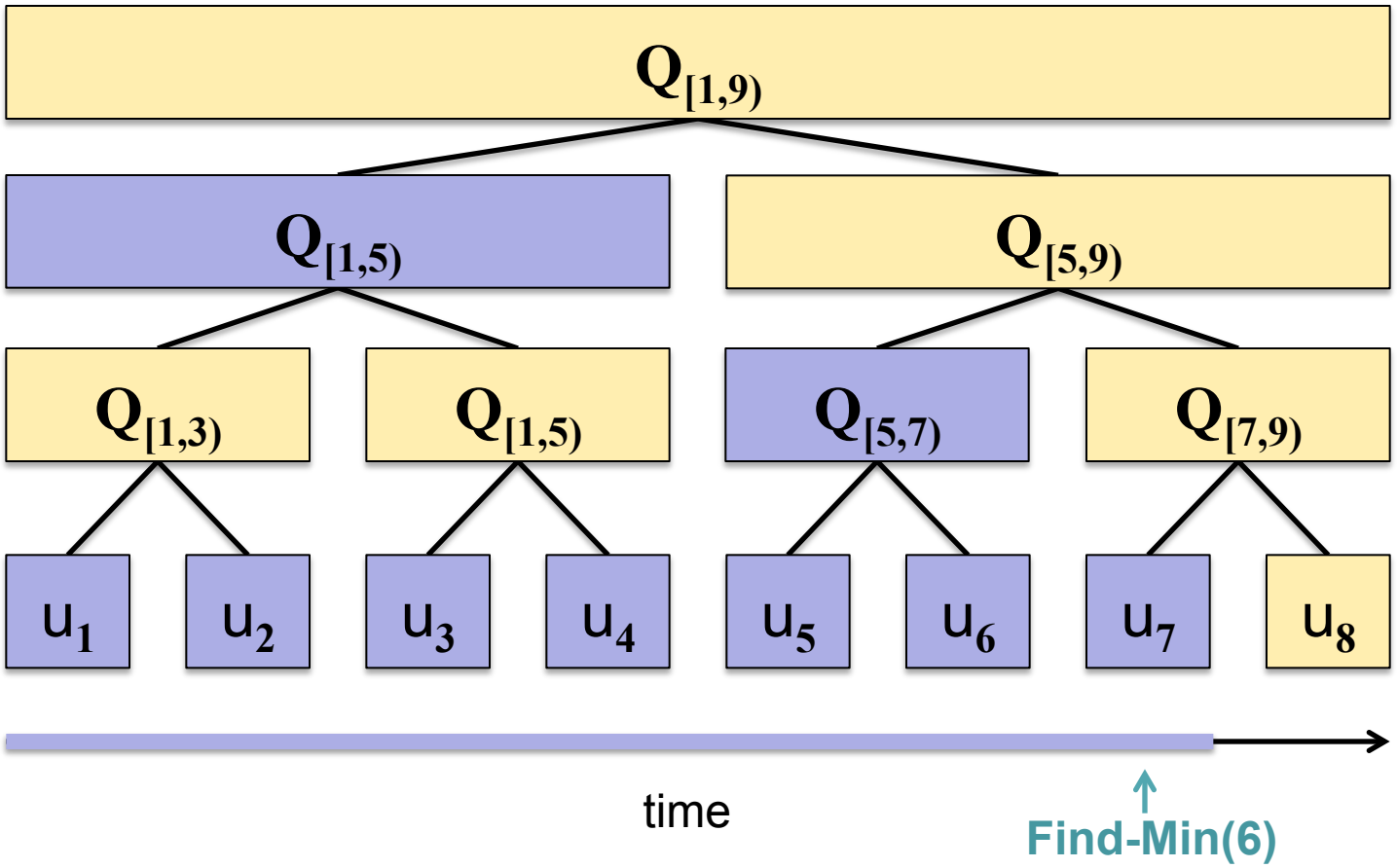
Demaine, Kaler, Liu, Sidford, Yedidia

Hierarchical Checkpointing: Query



Merge disjoint, contiguous intervals of time.

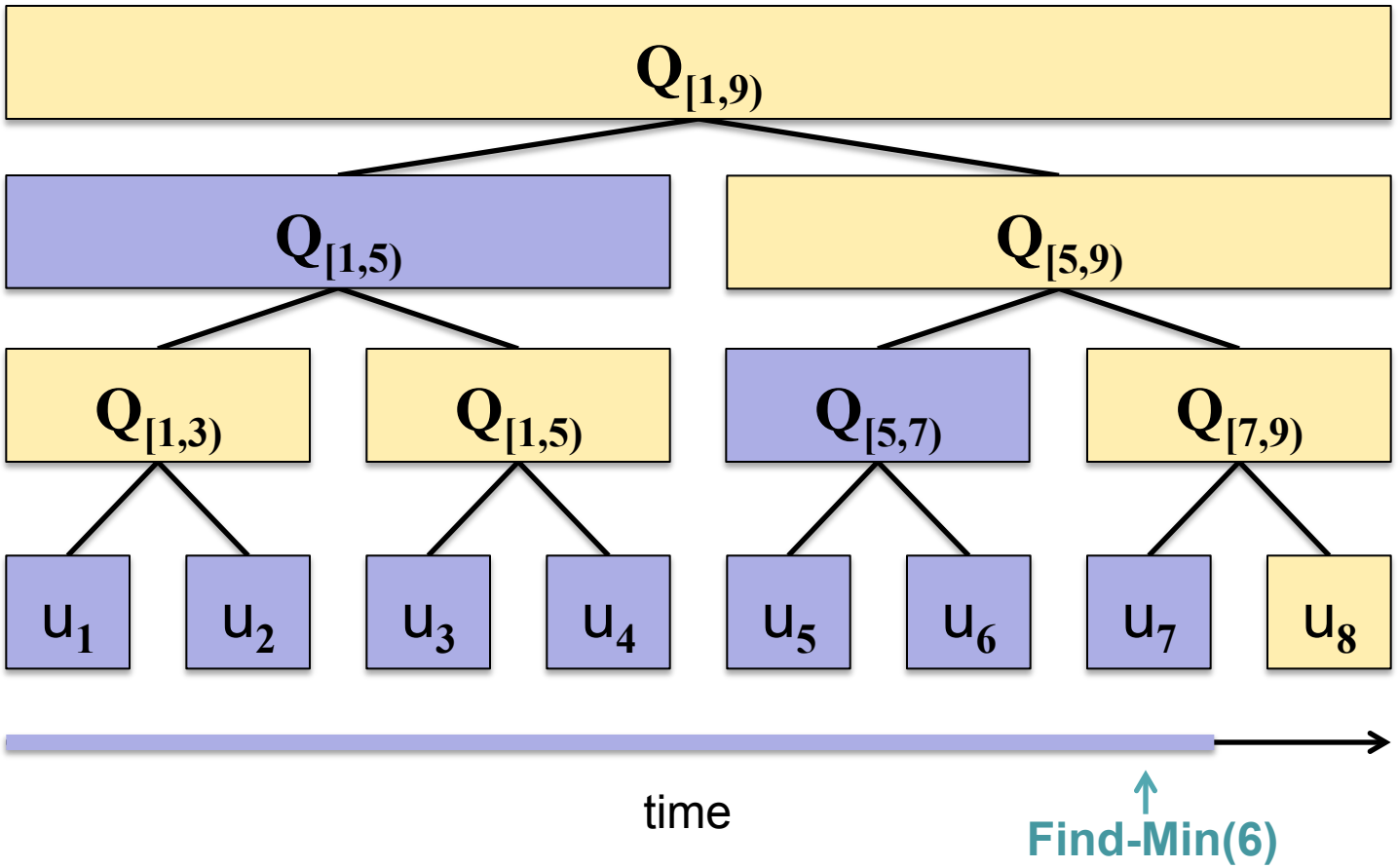
Hierarchical Checkpointing: Query



$O(\log m)$ partially retroactive priority queues to merge

Demaine, Kaler, Liu, Sidford, Yedidia

Hierarchical Checkpointing: Query



Requires ability to merge two partially retroactive priority queues to form a read-only partially retroactive priority queue.

Demaine, Kaler, Liu, Sidford, Yedidia

Checkpointing vs. Hierarchical Checkpointing

1. Checkpointing is a general transformation that can be applied to any partially retroactive data structure.

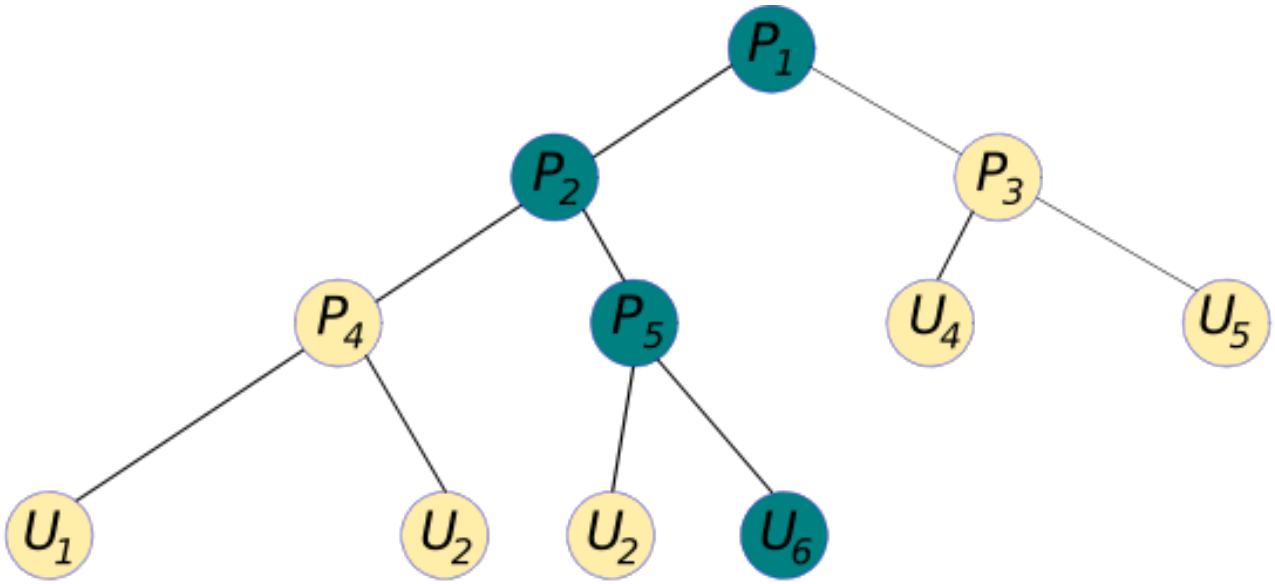
Checkpointing vs. Hierarchical Checkpointing

1. Checkpointing is a general transformation that can be applied to any partially retroactive data structure.
2. Hierarchical checkpointing has a stronger bound on conversion with a stronger assumption on partially retroactive data structure.

Checkpointing vs. Hierarchical Checkpointing

1. Checkpointing is a general transformation that can be applied to any partially retroactive data structure.
2. Hierarchical checkpointing has a stronger bound on conversion with a stronger assumption on partially retroactive data structure.
3. Hierarchical checkpointing requires ability to merge two versions of a structure to produce read-only structure that can be queried.

Hierarchical Checkpointing

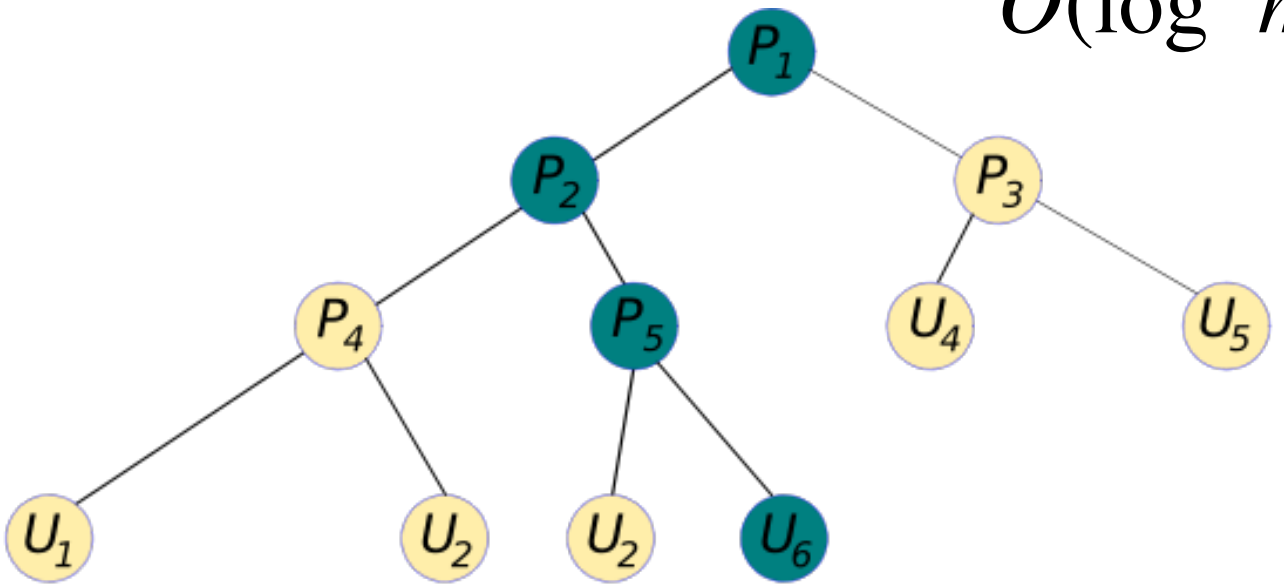


$t =$	1	2	3	4	5	6	7	8	9	10	11

Insert-Op(6, U_6)

Hierarchical Checkpointing

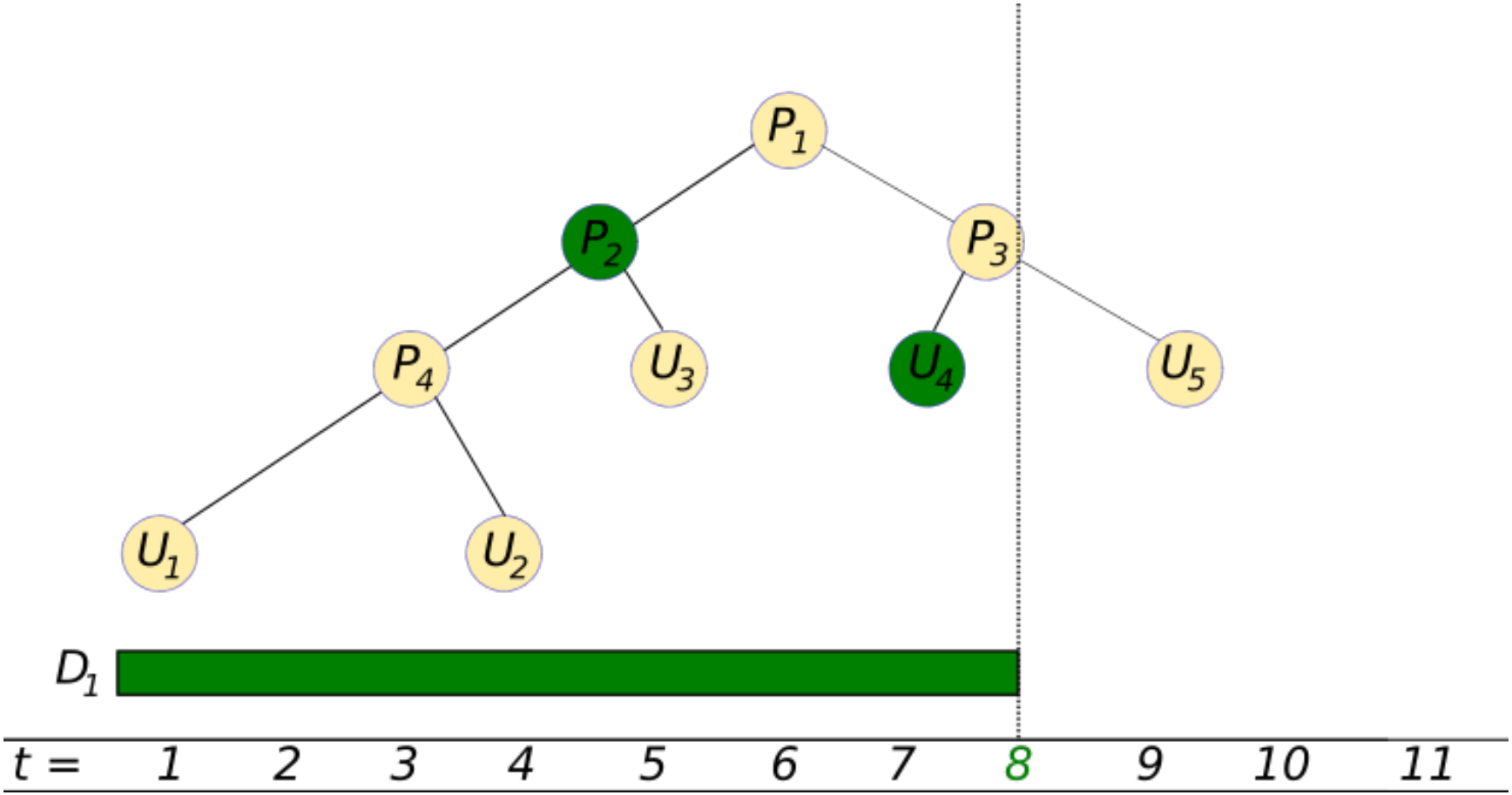
$O(\log^2 m)$ overhead



t =	1	2	3	4	5	6	7	8	9	10	11

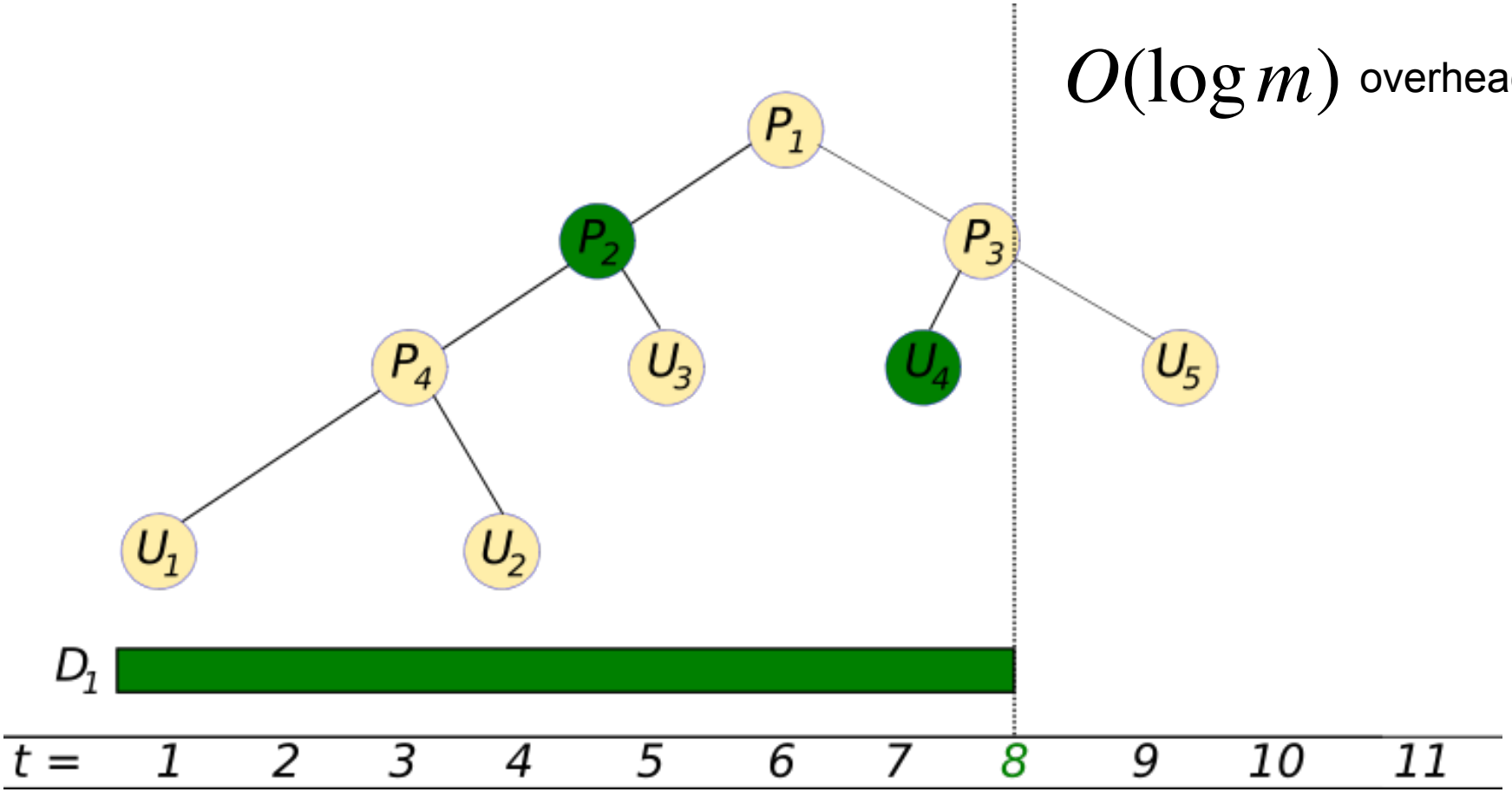
Insert-Op(6, U_6)

Hierarchical Checkpointing



Get-View(8)

Hierarchical Checkpointing



Get-View(8)

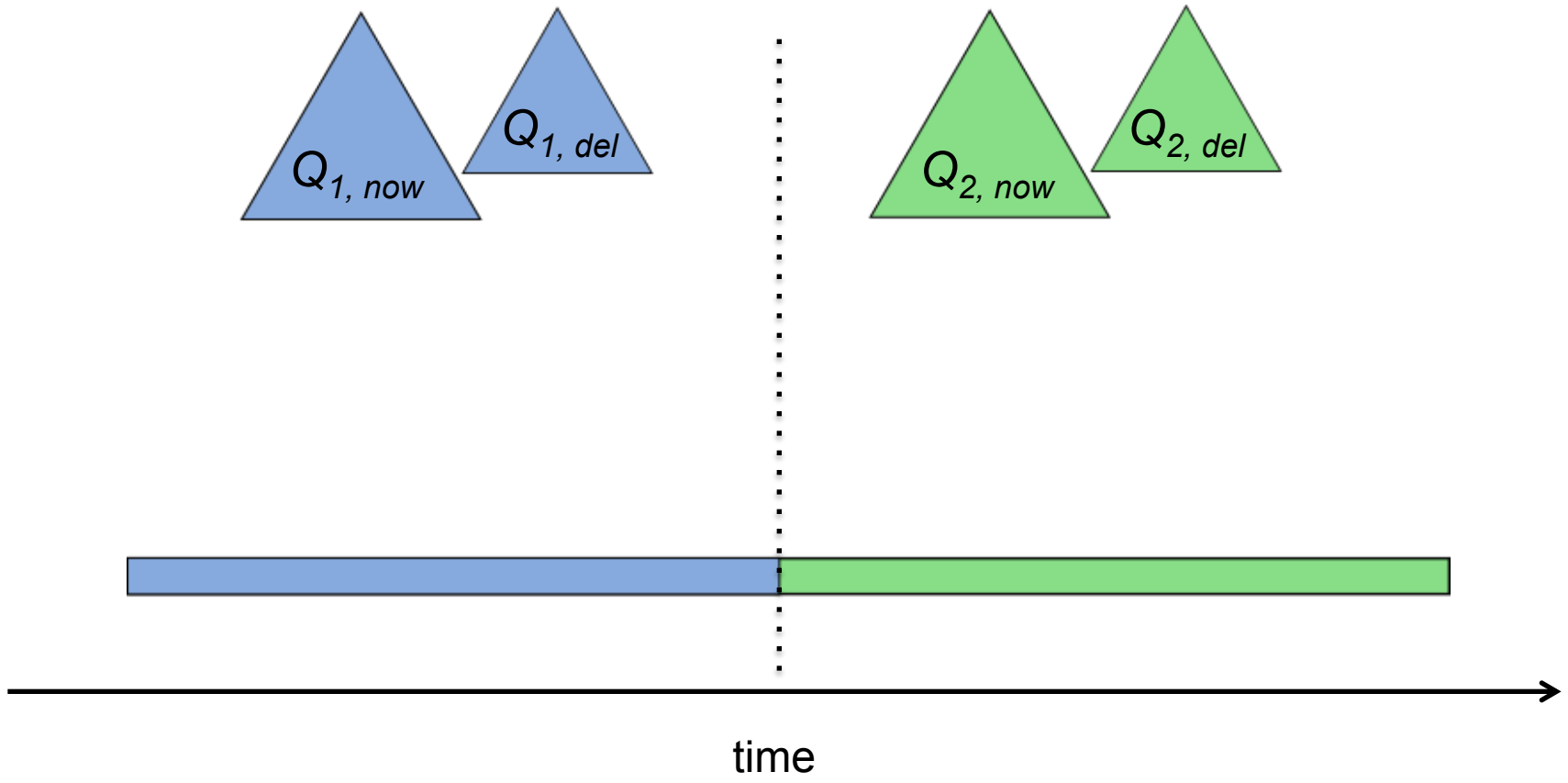
Merging PRPQs

- Q_{now} contains all elements that are in a partially retroactive priority queue
- Q_{del} contains set of elements that were deleted from a partially retroactive priority queue by delete-min operations
- We maintain Q_{now} and Q_{del} as weight-balanced B-trees with balance factor d = 8

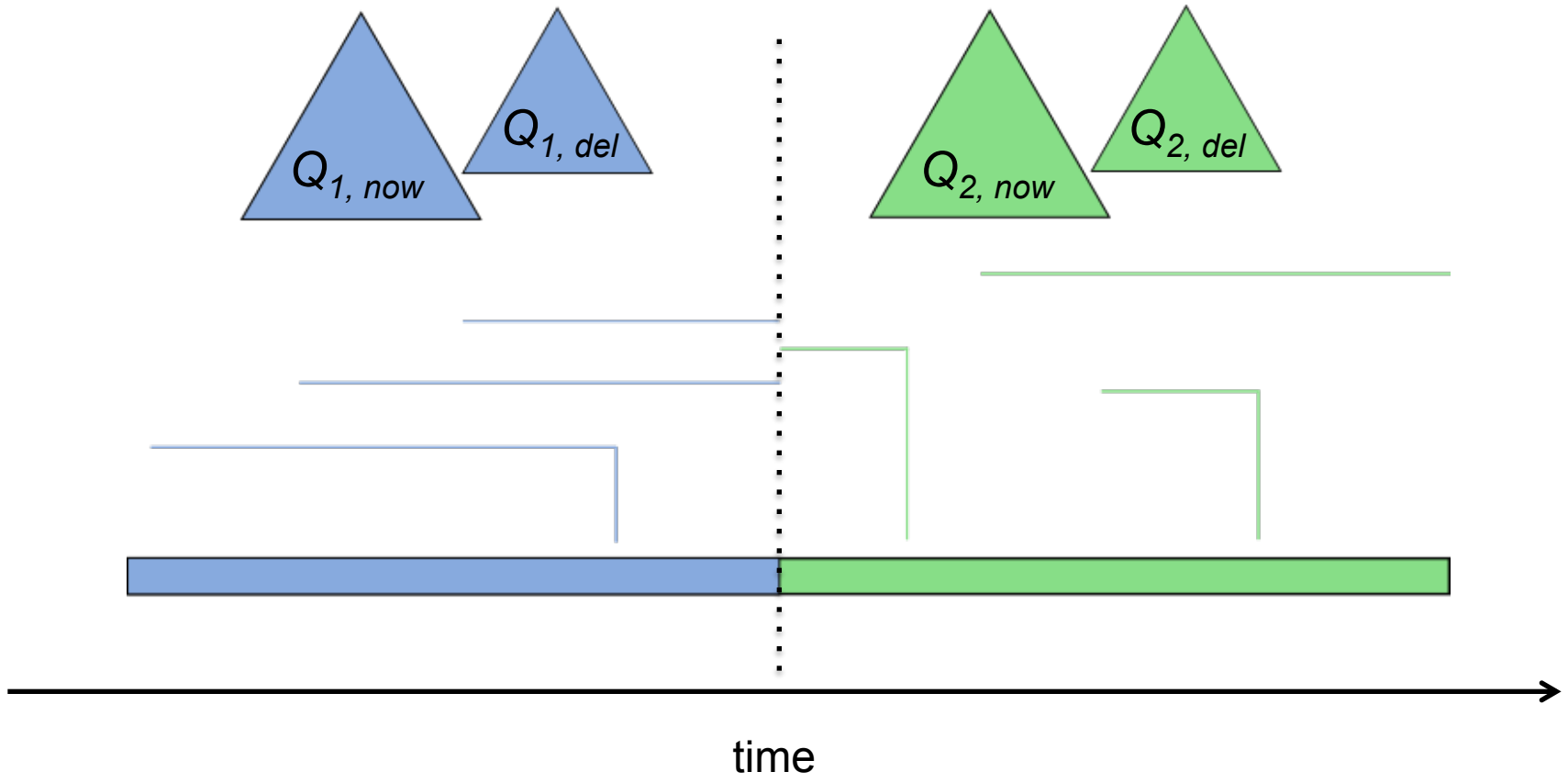
Merging PRPQs

time

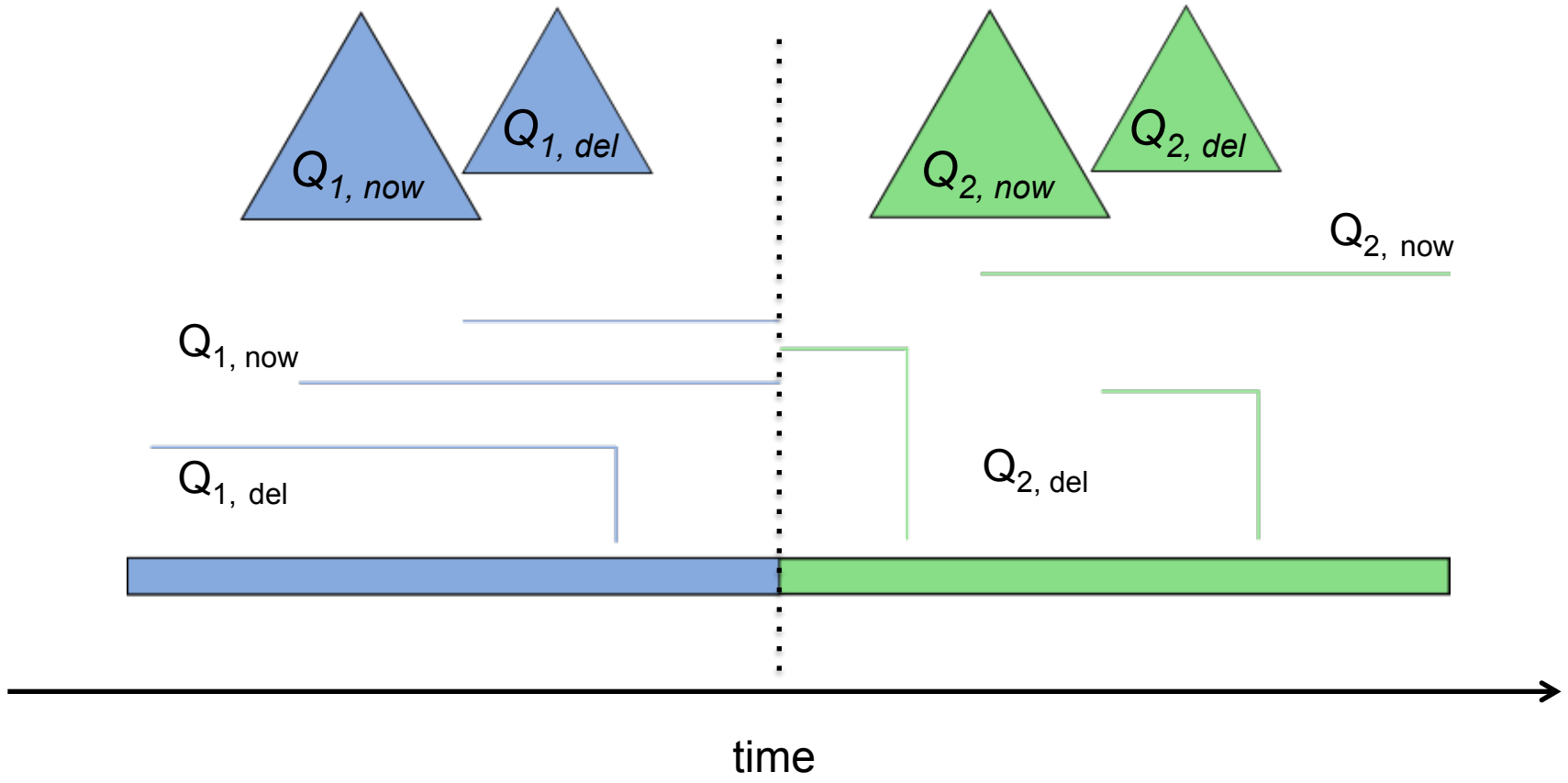
Merging PRPQs



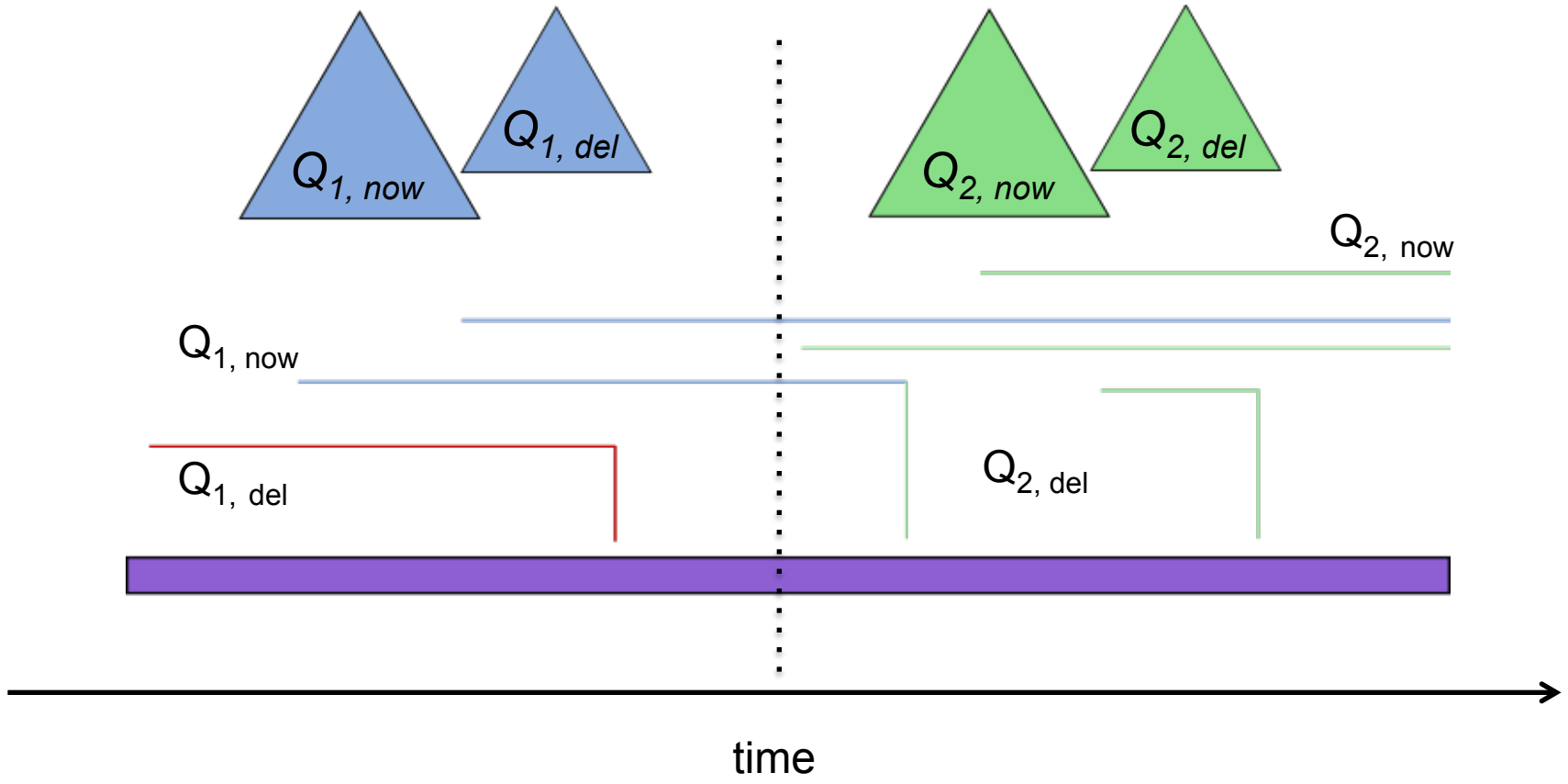
Merging PRPQs



Merging PRPQs

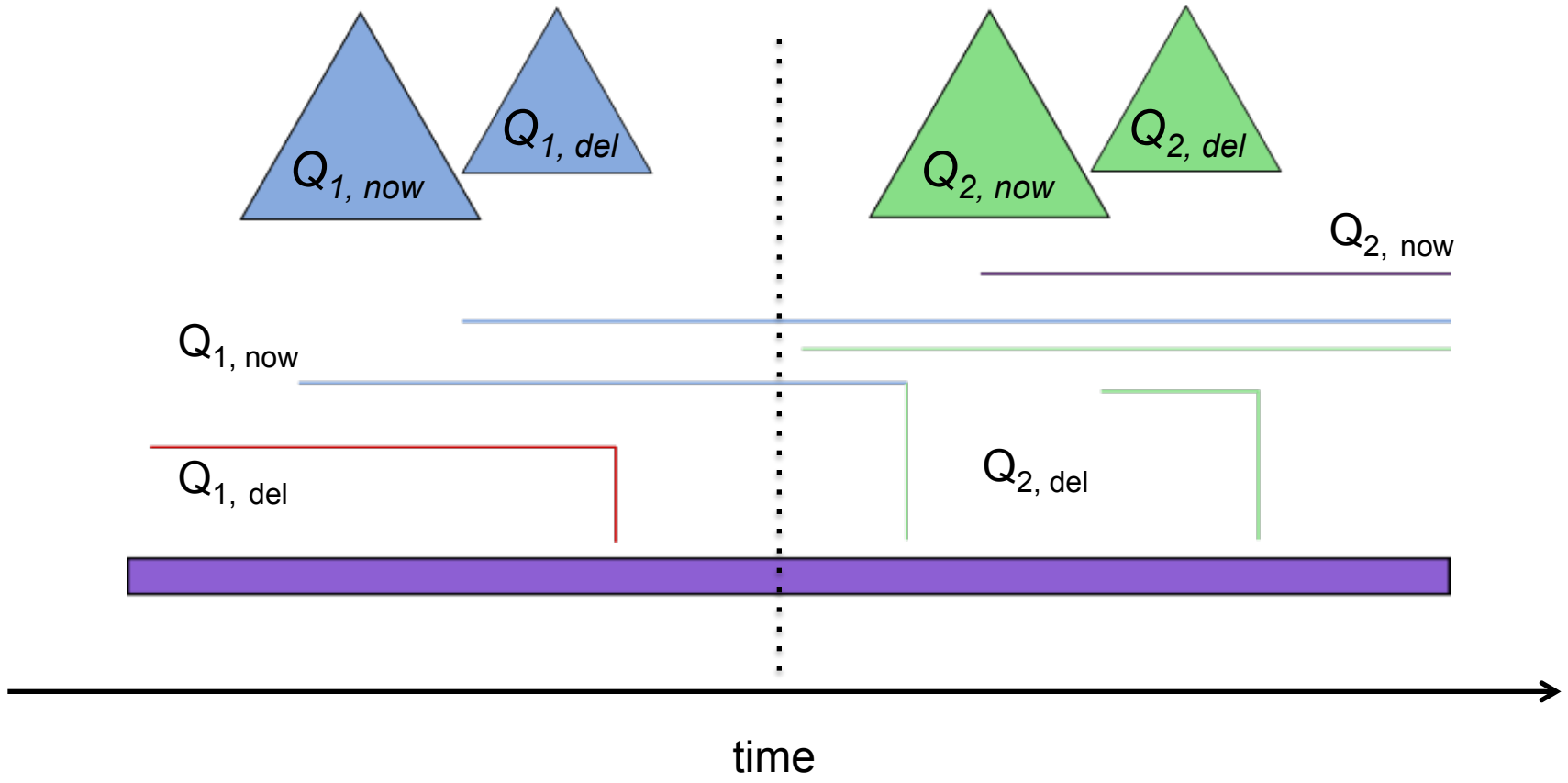


Merging PRPQs



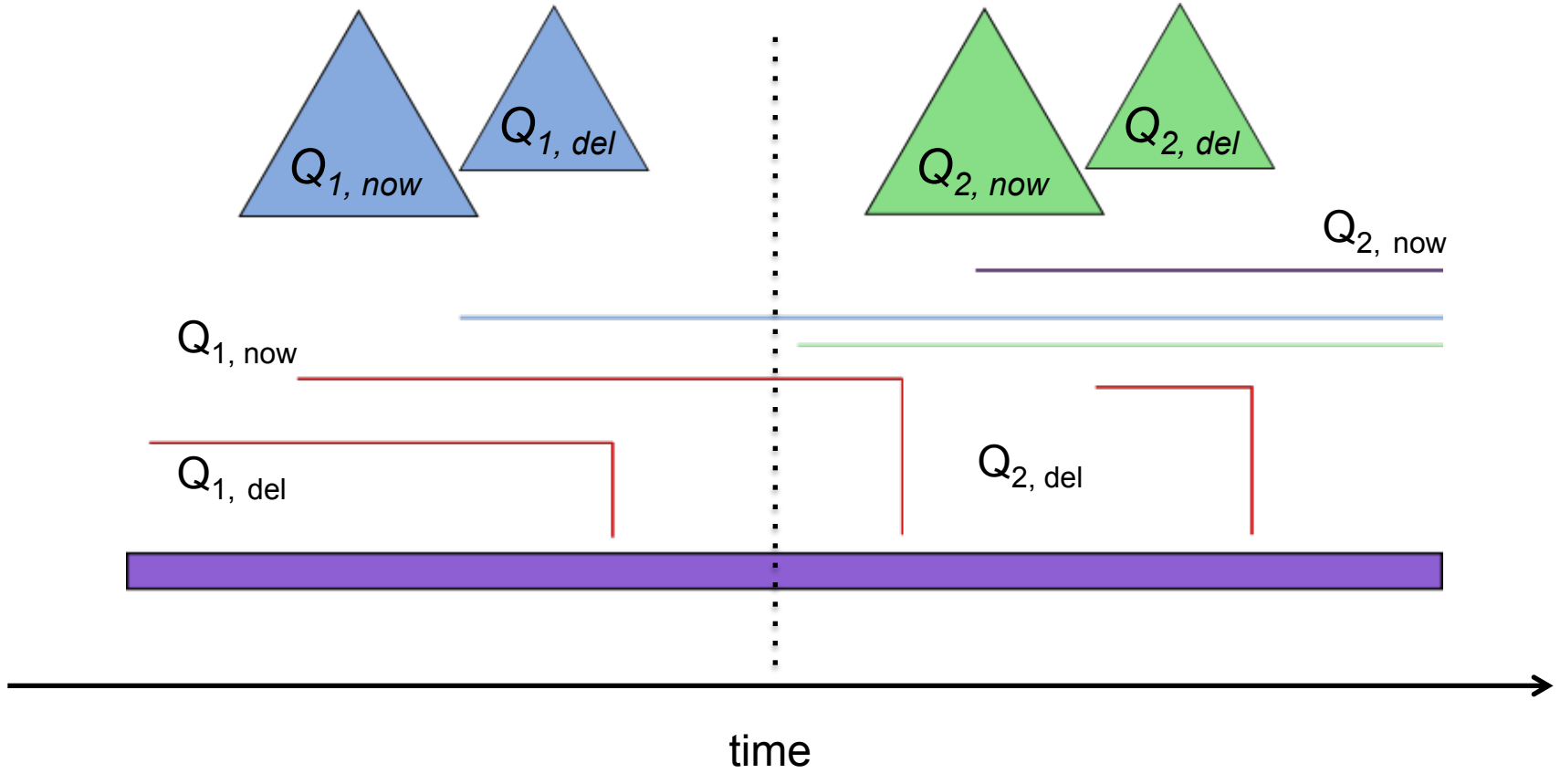
All elements in $Q_{1,\ del}$ deleted.

Merging PRPQs



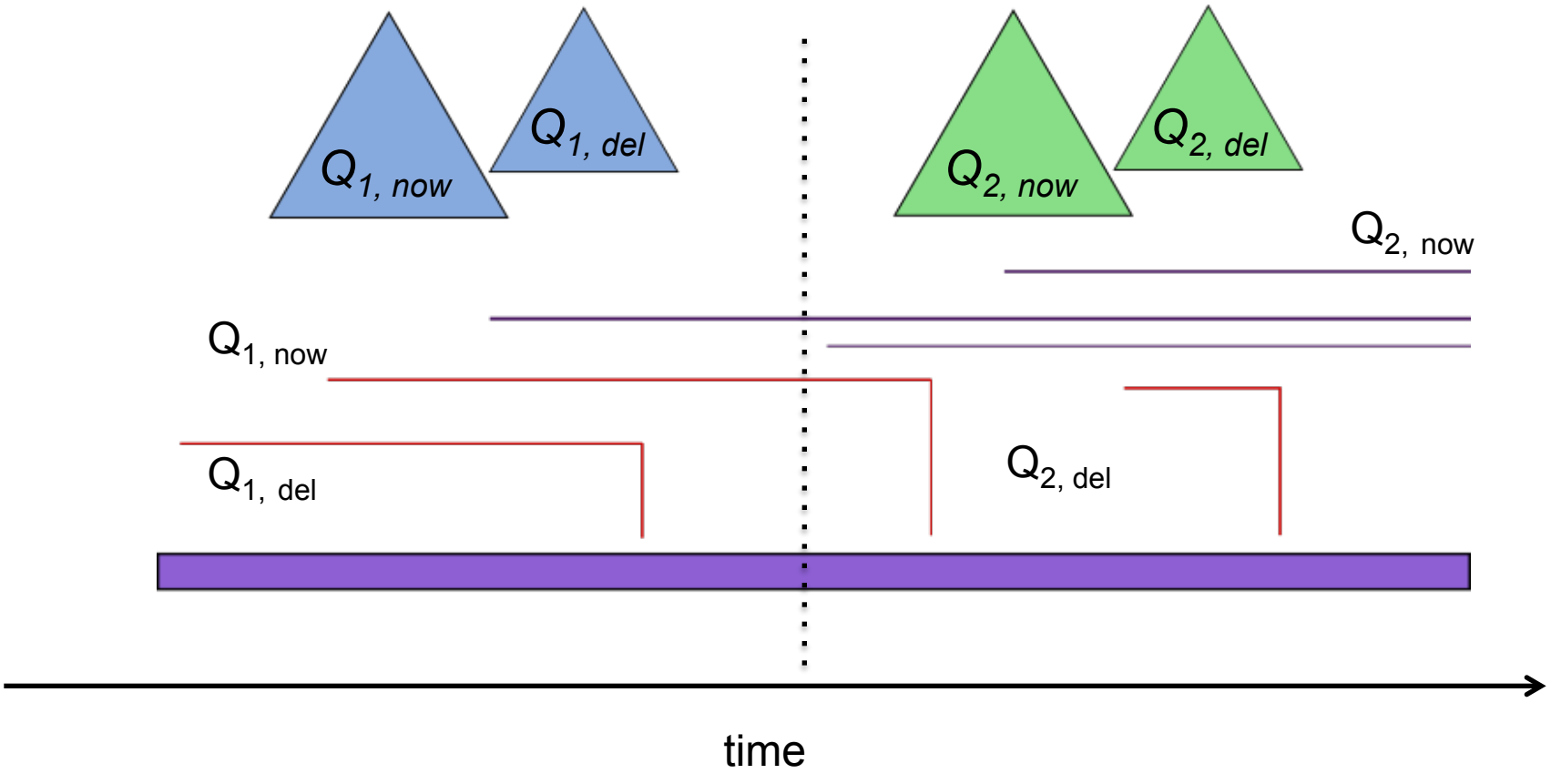
All elements in $Q_{2,\ now}$ survive.

Merging PRPQs



Smallest $D = \left|Q_{2,\ del}\right|$ elements from $Q_{1,\ now} \cup Q_{2,\ del}$ deleted.

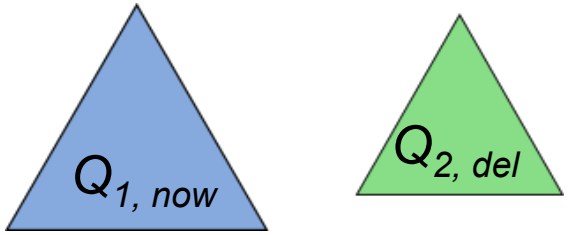
Merging PRPQs



Largest $S = \left|Q_{1,\ now}\right|$ elements from $Q_{1,\ now} \cup Q_{2,\ del}$ survive.

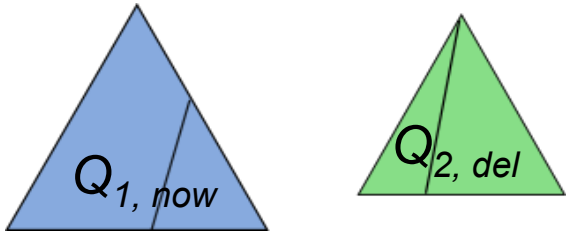
Merge by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.



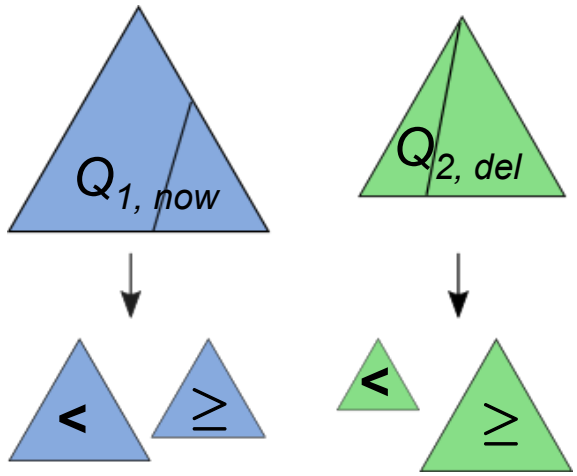
Merge by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
2. Use the element to split each tree into two parts containing surviving and deleted elements.



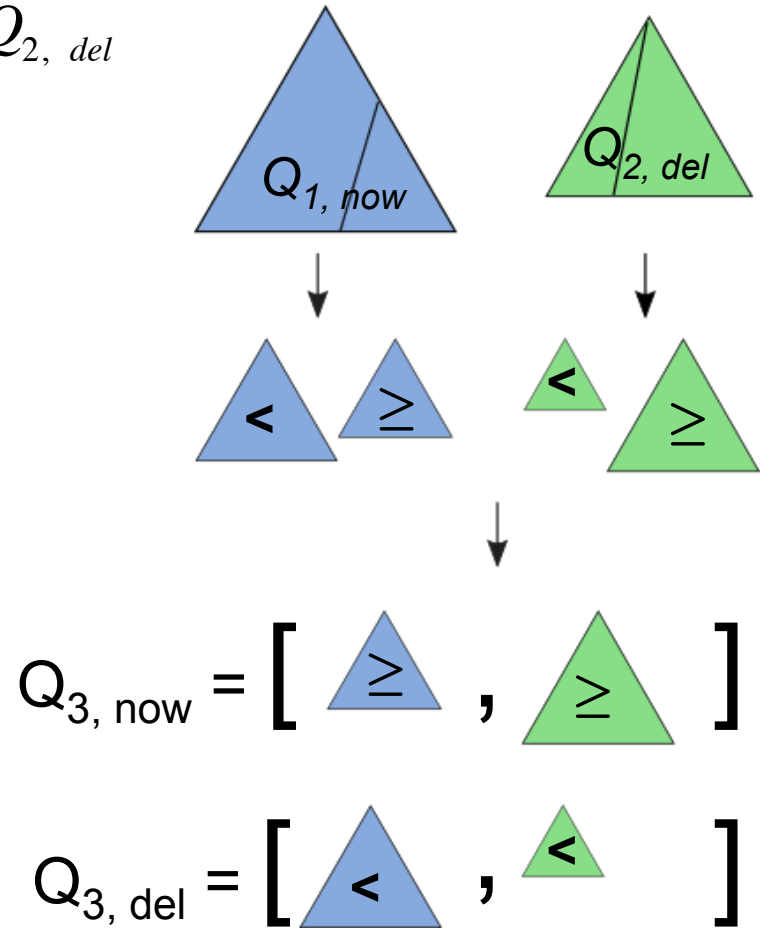
Merge by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
2. Use the element to split each tree into two parts containing surviving and deleted elements.



Merge by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
2. Use the element to split each tree into two parts containing surviving and deleted elements.
3. List of tree pieces comprise merged structure.



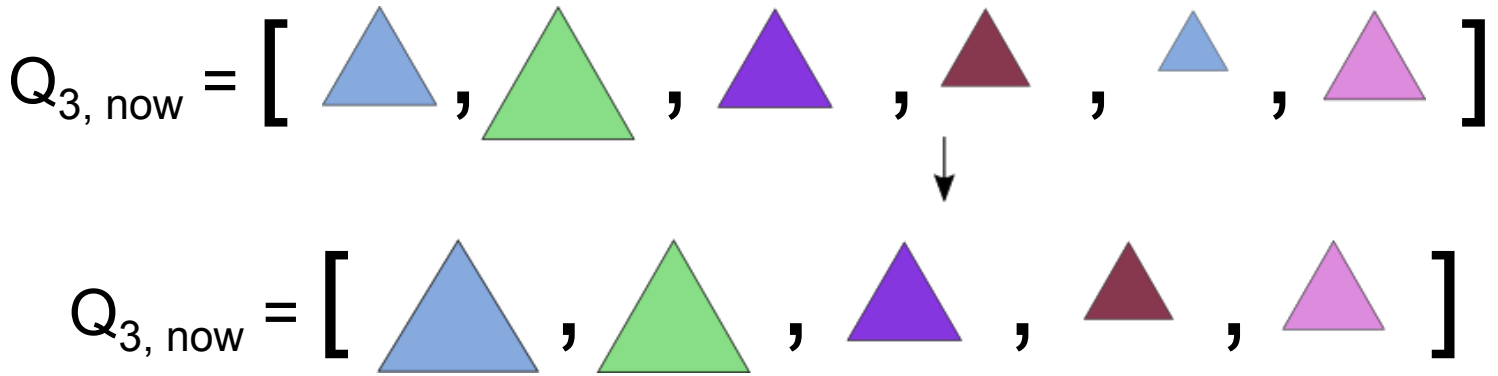
Merge by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.

$Q_{3,\ now}$ = []

Merge by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.



Merge by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.

5. Query the trees in merged Q_{now} to find min element.

$Q_{3,\ now}$ = []

$Q_{3,\ now}$ = []

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
   – Adaption of finding the (D+1)-st smallest element in *k* arrays [RA10] to *k* weight balanced binary search trees.

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
   - Adaption of finding the (D+1)-st smallest element in *k* arrays [RA10] to *k* weight balanced binary search trees.
   - $O(k \log m)$ time where $k = 2 \log m$.

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
   - Adaption of finding the (D+1)-st smallest element in *k* arrays [RA10] to *k* weight balanced binary search trees.
   - $O(k \log m)$ time where $k = 2 \log m$.
2. Use the element to split each tree into two parts containing surviving and deleted elements.

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
   - Adaption of finding the (D+1)-st smallest element in *k* arrays [RA10] to *k* weight balanced binary search trees.
   - $O(k \log m)$ time where $k = 2 \log m$.
2. Use the element to split each tree into two parts containing surviving and deleted elements.
   - $O(\log^2 m)$ assuming list of tree pieces size $O(\log m)$.

Runtime of Merging by Splitting Trees

1. Find element in $Q_{1,\ now} \cup Q_{2,\ del}$ where $D = \left|Q_{2,\ del}\right|$ are less than it.
   - Adaption of finding the (D+1)-st smallest element in *k* arrays [RA10] to *k* weight balanced binary search trees.
   - $O(k \log m)$ time where $k = 2 \log m$.
2. Use the element to split each tree into two parts containing surviving and deleted elements.
   - $O(\log^2 m)$ assuming list of tree pieces size $O(\log m)$.
3. List of tree pieces comprise merged structure.

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.
5. Query the trees in merged Q_{now} to find min element.

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.
5. Query the trees in merged Q_{now} to find min element.
   - $O(\log^2 m)$ time

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.
5. Query the trees in merged Q_{now} to find min element.
   - $O(\log^2 m)$ time

$$O(\log^2 m) \text{ time per merge}$$

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.
5. Query the trees in merged Q_{now} to find min element.
   - $O(\log^2 m)$ time

$O(\log^2 m)$ time per merge

$O(\log m)$ merges

Runtime of Merging by Splitting Trees

4. If two pieces originate from the same tree, concatenate them into one tree.
   - If two pieces in the same list are from the same original tree, then they span disjoint, contiguous intervals.
   - $O(\log m \log\log m)$ time to sort and $O(\log m)$ time to concatenate.
5. Query the trees in merged Q_{now} to find min element.
   - $O(\log^2 m)$ time

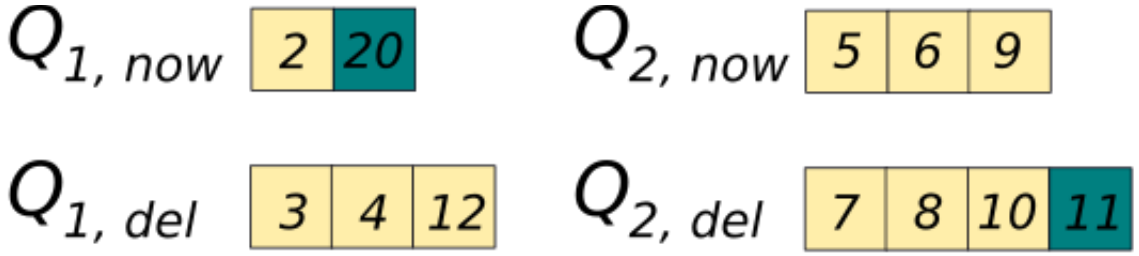
$$O(\log^3 m) \text{ runtime}$$

Splitting Trees

- Merge two partially retroactive priority queues by splitting the Q_{now} and Q_{del} trees to obtain surviving and deleted elements.
- Obtain a set of tree pieces.
- List of tree pieces comprise merged Q_{now} and Q_{del}.
- Query each of the trees in the merged Q_{now} to obtain the minimum element.

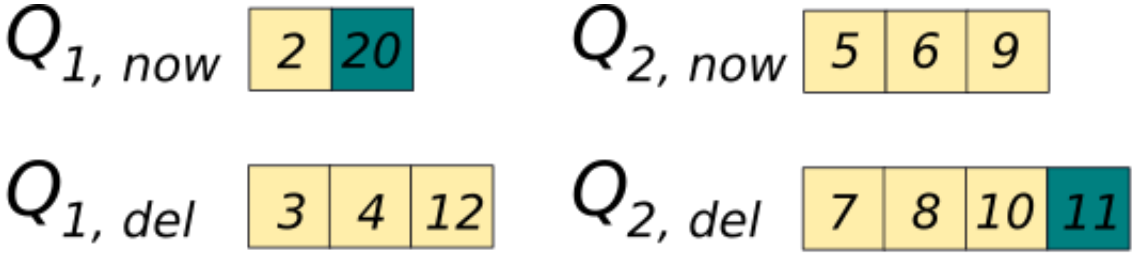
Merging PRPQs Example

Find $|Q_{1,\ del}|$ smallest elements to delete from $Q_{1,\ now} \cup Q_{2,\ del}$



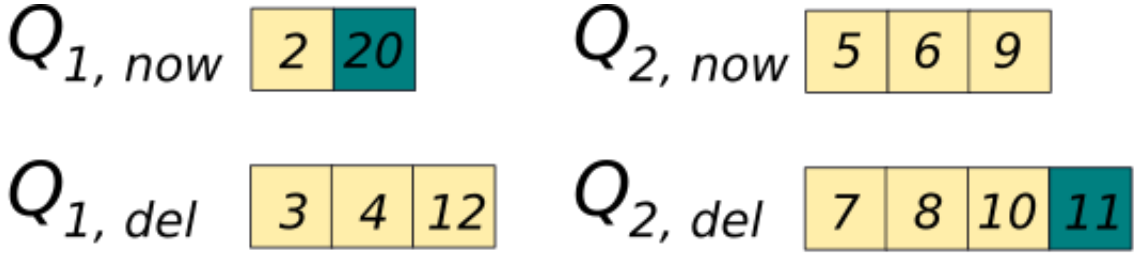
Merging PRPQs Example

Find $|Q_{1,\ del}|$ smallest elements to delete from $Q_{1,\ now} \cup Q_{2,\ del}$

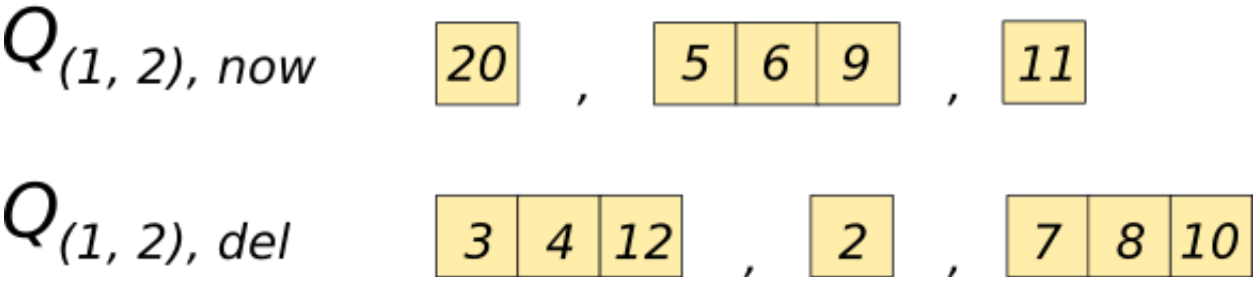


Merging PRPQs Example

Find $|Q_{1,\ del}|$ smallest elements to delete from $Q_{1,\ now} \cup Q_{2,\ del}$



The merged partially retroactive priority queue contains list of tree pieces.



Optimizing the Merge

- Query for the minimum element in list of trees.
- To prevent the list of trees from exceeding $O(\log m)$ trees, we can concatenate trees that originated from the same queue.
- Merging the $O(\log m)$ trees with merge time $T(k) = k \log m$ where $k \leq \log m$ results in a merge time of $O(\log^3 m)$.

Hierarchical Checkpointing for FRPQs

- By applying our checkpointing framework directly to fully retroactive priority queues, we obtain

Operation	Runtime
Updates	~~$O(\log^3 m)$~~ → $O(\log^2 m)$
Queries	$O(\log^2 m \log\log m)$
Finding time of deletion	~~$O(\log^3 m \log\log m)$~~ → $O(\log^2 m)$

Hierarchical Checkpointing for FRPQs

By applying our checkpointing framework directly to fully retroactive priority queues, we obtain

Operation	Hierarchical Checkpointing Runtime	Improved Runtimes
Updates	$O(\log^3 m)$	
Queries	$O(\log^3 m)$	
Finding time of deletion	$O(\log^4 m)$	

Log-Shaving: Updates

Simpler rebuild procedure when a subtree is unbalanced leads to $O(\log^2 m)$ updates.

Operation	Hierarchical Checkpointing Runtime	Improved Runtimes
Updates	$O(\log^3 m)$	$O(\log^2 m)$
Queries	$O(\log^3 m)$	
Finding time of deletion	$O(\log^4 m)$	

Demaine, Kaler, Liu, Sidford, Yedidia

Log-Shaving: Queries

Merging the updates using a binary merge tree results in $O(\log^2 m \log\log m)$ queries.

Operation	Hierarchical Checkpointing Runtime	Improved Runtimes
Updates	$O(\log^3 m)$	$O(\log^2 m)$
Queries	$O(\log^3 m)$	$O(\log^2 m \log\log m)$
Finding time of deletion	$O(\log^4 m)$	

Demaine, Kaler, Liu, Sidford, Yedidia

Log-Shaving: Find-delete-time

Modified binary search that maintains a counter of current surviving elements results in $O(\log^2 m)$ runtime for find-deletion-time.

Operation	Hierarchical Checkpointing Runtime	Improved Runtimes
Updates	$O(\log^3 m)$	$O(\log^2 m)$
Queries	$O(\log^3 m)$	$O(\log^2 m \log\log m)$
Finding time of deletion	$O(\log^4 m)$	$O(\log^2 m)$

Demaine, Kaler, Liu, Sidford, Yedidia

Time-Fusibility

- Time-fusible partially retroactive data structures are structures that allow merging of timelines:
  - Produces a read-only data structure that contains updates from both timelines.
  - Exhibits substring closure.

Time-Fusibility

- Time-fusible partially retroactive data structures are structures that allow merging of timelines:
  - Produces a read-only data structure that contains updates from both timelines.
  - Exhibits substring closure.
- Time-fusible data structures allow an $O(\log^2 m)$ overhead converting from partial to full retroactivity via hierarchical checkpointing.

Conclusion and Open Questions

Conclusion and Open Questions

- Hierarchical checkpointing procedure for time-fusible structures $O(\log^2 m)$ overhead.

Conclusion and Open Questions

- Hierarchical checkpointing procedure for time-fusible structures $O(\log^2 m)$ overhead.
  - Examples of other structures?

Conclusion and Open Questions

- Hierarchical checkpointing procedure for time-fusible structures $O(\log^2 m)$ overhead.
  - Examples of other structures?
- Polylogarithmic fully retroactive priority queue with runtimes:

Operation	Improved Runtimes
Updates	$O(\log^2 m)$
Queries	$O(\log^2 m \log\log m)$
Finding time of deletion	$O(\log^2 m)$

Conclusion and Open Questions

- Hierarchical checkpointing procedure for time-fusible structures $O(\log^2 m)$ overhead.
  - Examples of other structures?
- Polylogarithmic fully retroactive priority queue with runtimes:

Operation	Improved Runtimes
Updates	$O(\log^2 m)$
Queries	$O(\log^2 m \log\log m)$
Finding time of deletion	$O(\log^2 m)$

  - Logarithmic or constant overhead?

Open Questions

- Does there exist logarithmic separation between partial and full retroactivity for priority queues?
- Can we adapt hierarchical checkpointing to a broader class of structures?

PRPQ Fusion

- (Diagram describing finding the split-key procedure)
- We first find the split-key in the PRPQ using a modified median finding algorithm
- Suppose for simplicity $|Q_{now}| = |Q_{del}|$
- (Diagram showing find split key with two priority queues, i.e. two sets of Q_now and Q_del)

Time-Fusibility of PRPQs

- Two partially retroactive priority queues may be fused
- Split-key "splits" elements into elements that are deleted and elements that survive
- All elements less than split-key are deleted in the fused structure
- All elements greater than split-key survive
- (Words above may be said and replaced with a diagram.)

Fusion Runtime

- The runtime of the fusion operation for partially retroactive priority queues is
  $O(\log m \log\log m)$
- (Above could be placed with the diagram of the fusion in the previous slide.)

Substring Integrity

- Deleted element is moved from Q_{now} to Q_{del}
- Delete-min operations that do not move an element from Q_{now} to Q_{del} can be represented by the insertion of a key of ∞ weight