Advanced Data Structures

Lecture 10: Temporal Data Structures 2

Florian Kurpicz

Organization

Exams

- 12.09.2023, 13.09.2023, and 21.09.2023 (09:00–16:00)
- 22.09.2023 (13:00–16:00)
- write to `blancani@kit.edu`
  - full name
  - Matrikelnummer
  - PO version
  - date
- in person
- 17.07.2022 Q&A during last half of lecture
- registration for project is open

Organization

Exams

- 12.09.2023, 13.09.2023, and 21.09.2023 (09:00–16:00)
- 22.09.2023 (13:00–16:00)
- write to `blancani@kit.edu`
  - full name
  - Matrikelnummer
  - PO version
  - date
- in person
- 17.07.2022 Q&A during last half of lecture
- registration for project is open

Evaluation

- next week

Range Minimum Queries in $O(1)$ Time and $O(n \log n)$ Space

- instead of storing all solutions
- store solutions for intervals of length 2^k for every k
- $M[0..n)[0..\lfloor \log n \rfloor)$

Queries

- query $rmq(A, s, e)$ is answered using two subqueries
- let $\ell = \lfloor log(e - s + 1) \rfloor$
- $m_1 = rmq(A, s, s + 2^\ell - 1)$ and $m_2 = rmq(A, e - 2^\ell + 1, e)$
- $rmq(A, s, e) = \arg\min_{m \in \{m_1, m_2\}} A[m]$

Construction

$$\begin{aligned} M[x][\ell] &= rmq(A, x, x + 2^\ell - 1) \\ &= \arg\min\{A[i] \colon i \in [x, x + 2^\ell)\} \\ &= \arg\min\{A[i] \colon i \in \{rmq(A, x, x + 2^{\ell-1} - 1), \\ &= \qquad rmq(A, x + 2^{\ell-1}, x + 2^\ell - 1)\}\} \\ &= \arg\min\{A[i] \colon i \in \{M[x][\ell - 1], \\ &= \qquad M[x + 2^{\ell-1}][\ell - 1]\}\} \end{aligned}$$

- dynamic programming in $O(n \log n)$ time

PINGO

https://pingo.scc.kit.edu/919720

Recap: Persistent Data Structures

- lecture based on: `http://courses.csail.mit.edu/6.851/spring12/lectures/L01`

Recap: Persistent Data Structures

- lecture based on: `http://courses.csail.mit.edu/6.851/spring12/lectures/L01`

Persistence

- change in the past creates new branch
- similar to version control
- everything old/new remains the same

Recap: Persistent Data Structures

- lecture based on: `http://courses.csail.mit.edu/6.851/spring12/lectures/L01`

Persistence

- change in the past creates new branch
- similar to version control
- everything old/new remains the same

Definition: Partial Persistence

Only the latest version can be updated

Definition: Full Persistence

Any version can be updated

Definition: Confluent Persistence

Like full persistence, but two versions can be combined to a new version

Definition: Functional

Nodes cannot be modified, only new nodes can be created

Recap: Persistent Data Structures

- lecture based on: `http://courses.csail.mit.edu/6.851/spring12/lectures/L01`

Persistence

- change in the past creates new branch
- similar to version control
- everything old/new remains the same

Retroactivity

- change in the past affects future
- make change in earlier version changes all later versions

Definition: Partial Persistence

Only the latest version can be updated

Definition: Full Persistence

Any version can be updated

Definition: Confluent Persistence

Like full persistence, but two versions can be combined to a new version

Definition: Functional

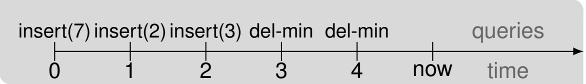
Nodes cannot be modified, only new nodes can be created

Retroactive Data Structures

Operations

- `INSERT`$(t, \mathit{operation})$: insert operation at time t
- `DELETE`(t): delete operation at time t
- `QUERY`(t, query): ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure

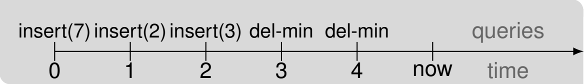


Retroactive Data Structures

Operations

- $\texttt{INSERT}(t, \mathit{operation})$: insert operation at time t
- $\texttt{DELETE}(t)$: delete operation at time t
- $\texttt{QUERY}(t, \mathit{query})$: ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

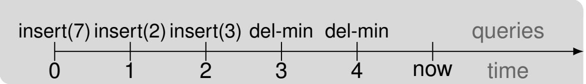
$\texttt{QUERY}$ is only allowed for $t = \infty$ ℹ now

Retroactive Data Structures

Operations

- `INSERT`$(t, \mathit{operation})$: insert operation at time t
- `DELETE`(t): delete operation at time t
- `QUERY`(t, query): ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

`QUERY` is only allowed for $t = \infty$ ℹ now

Definition: Full Retroactivity

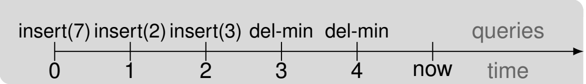
`QUERY` is allowed at any time t

Retroactive Data Structures

Operations

- `INSERT`$(t, \mathit{operation})$: insert operation at time t
- `DELETE`(t): delete operation at time t
- `QUERY`(t, query): ask *query* at time t

- for a priority queue updates are
  - insert
  - delete-min
- time is integer ℹ for simplicity otherwise use order-maintenance data structure



Definition: Partial Retroactivity

`QUERY` is only allowed for $t = \infty$ ℹ now

Definition: Full Retroactivity

`QUERY` is allowed at any time t

Definition: Nonoblivious Retroactivity

`INSERT`, `DELETE`, and `QUERY` at any time t but also identify changed `QUERY` results

Easy Cases: Partial Retroactivity

- invertible updates
    - operation op^{-1} such that $op^{-}1(op(\cdot)) = \emptyset$
    - `DELETE` becomes `INSERT` inverse operation
- makes partial retroactivity easy
- $\texttt{INSERT}(t, operation) = \texttt{INSERT}(\infty, operation)$
- $\texttt{DELETE}(t, op) = \texttt{INSERT}(\infty, op^{-1})$

Easy Cases: Partial Retroactivity

- invertible updates
  - operation op^{-1} such that $op^{-}1(op(\cdot)) = \emptyset$
  - `DELETE` becomes `INSERT` inverse operation
- makes partial retroactivity easy
- $\texttt{INSERT}(t, operation) = \texttt{INSERT}(\infty, operation)$
- $\texttt{DELETE}(t, op) = \texttt{INSERT}(\infty, op^{-1})$

Partial Retroactivity

- hashing
- dynamic dictionaries
- array with updates only ℹ $A[i] += value$

Search Problems

Definition: Search Problem

A search problem is a problem on a set S of objects with operations *insert*, *delete*, and *query*(x, S)

Search Problems

Definition: Search Problem

A search problem is a problem on a set S of objects with operations *insert*, *delete*, and $\mathit{query}(x, S)$

Definition: Decomposable Search Problem

A decomposable search problem is a search problem, with

- $\mathit{query}(x, A \cup B) = f(\mathit{query}(x, A), \mathit{query}(x, B))$
- with f requiring $O(1)$ time

Search Problems

Definition: Search Problem

A search problem is a problem on a set S of objects with operations *insert*, *delete*, and $query(x, S)$

Definition: Decomposable Search Problem

A decomposable search problem is a search problem, with

- $query(x, A \cup B) = f(query(x, A), query(x, B))$
- with f requiring $O(1)$ time

- which decomposable search problem have we seen **PINGO**

Search Problems

Karlsruhe Institute of Technology

Definition: Search Problem

A search problem is a problem on a set S of objects with operations *insert*, *delete*, and $\mathit{query}(x, S)$

Definition: Decomposable Search Problem

A decomposable search problem is a search problem, with

- $\mathit{query}(x, A \cup B) = f(\mathit{query}(x, A), \mathit{query}(x, B))$
- with f requiring $O(1)$ time

- which decomposable search problem have we seen **PINGO**

- predecessor and successor search
- range minimum queries

- nearest neighbor
- point location
- . . .

Search Problems

Definition: Search Problem

A search problem is a problem on a set S of objects with operations *insert*, *delete*, and $query(x, S)$

Definition: Decomposable Search Problem

A decomposable search problem is a search problem, with

- $query(x, A \cup B) = f(query(x, A), query(x, B))$
- with f requiring $O(1)$ time

- which decomposable search problem have we seen **PINGO**

- predecessor and successor search
- range minimum queries

- nearest neighbor
- point location
- . . .

- these types of problems are also “easy”

Decomposable Search Problems: Full Retroactivity

Lemma: Full Retroactivity for DSP

Every decomposable search problems can be made fully retroactive with a $O(\log m)$ overhead in space and time, where m is the number of operations

Decomposable Search Problems: Full Retroactivity

Lemma: Full Retroactivity for DSP

Every decomposable search problems can be made fully retroactive with a $O(\log m)$ overhead in space and time, where m is the number of operations

Proof (Sketch)

- use balances search tree ⓘ segment tree
- each leaf corresponds to an update
- node n corresponds to interval of time $[s_n, e_n]$
- if an object exists in the time interval $[s, e]$, then it appears in node n if $[s_n, e_n] \subseteq [s, e]$ if none of n's ancestors' are $\subseteq [s, e]$
- each object occurs in $O(\log n)$ nodes

Decomposable Search Problems: Full Retroactivity

Lemma: Full Retroactivity for DSP

Every decomposable search problems can be made fully retroactive with a $O(\log m)$ overhead in space and time, where m is the number of operations

Proof (Sketch)

- use balances search tree ℹ segment tree
- each leaf corresponds to an update
- node n corresponds to interval of time $[s_n, e_n]$
- if an object exists in the time interval $[s, e]$, then it appears in node n if $[s_n, e_n] \subseteq [s, e]$ if none of n's ancestors' are $\subseteq [s, e]$
- each object occurs in $O(\log n)$ nodes

Proof (Sketch, cnt.)

- to query find leaf corresponding to t
- look at ancestors to find all objects
- $O(\log m)$ results which can be combined in $O(\log m)$ time

Decomposable Search Problems: Full Retroactivity

Lemma: Full Retroactivity for DSP

Every decomposable search problems can be made fully retroactive with a $O(\log m)$ overhead in space and time, where m is the number of operations

Proof (Sketch)

- use balances search tree ℹ segment tree
- each leaf corresponds to an update
- node n corresponds to interval of time $[s_n, e_n]$
- if an object exists in the time interval $[s, e]$, then it appears in node n if $[s_n, e_n] \subseteq [s, e]$ if none of n's ancestors' are $\subseteq [s, e]$
- each object occurs in $O(\log n)$ nodes

Proof (Sketch, cnt.)

- to query find leaf corresponding to t
- look at ancestors to find all objects
- $O(\log m)$ results which can be combined in $O(\log m)$ time

- data structure is stored for each operation!
- $O(\log m)$ space overhead!

General Full Retroactivity

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

General Full Retroactivity

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

Proof (Sketch)

- two values X and Y
- initially $X = \emptyset$ and $Y = \emptyset$
- supported operations
  - $X = x$
  - $Y += value$
  - $Y = X \cdot Y$
  - $query\ Y$

General Full Retroactivity

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

Proof (Sketch)

- two values X and Y
- initially $X = \emptyset$ and $Y = \emptyset$
- supported operations
  - $X = x$
  - $Y+ = value$
  - $Y = X \cdot Y$
  - $query\ Y$

Proof (Sketch, cnt.)

- perform operations
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n=1}$
  - $Y = X \cdot Y$
  - $\ldots$
  - $Y+ = a_0$
- what are we computing here? **PINGO**

General Full Retroactivity

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

Proof (Sketch)

- two values X and Y
- initially $X = \emptyset$ and $Y = \emptyset$
- supported operations
  - $X = x$
  - $Y+ = value$
  - $Y = X \cdot Y$
  - $query\ Y$

Proof (Sketch, cnt.)

- perform operations
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n=1}$
  - $Y = X \cdot Y$
  - $\dots$
  - $Y+ = a_0$
- what are we computing here? **PINGO**
- $Y = a_n \cdot X^n + a_{n-1}X^{n-1} + \cdots + a_0$

General Full Retroactivity

Karlsruhe Institute of Technology

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

Proof (Sketch)

- two values X and Y
- initially $X = \emptyset$ and $Y = \emptyset$
- supported operations
  - $X = x$
  - $Y += value$
  - $Y = X \cdot Y$
  - $query\ Y$

Proof (Sketch, cnt.)

- perform operations
  - $Y += a_n$
  - $Y = X \cdot Y$
  - $Y += a_{n=1}$
  - $Y = X \cdot Y$
  - $\dots$
  - $Y += a_0$
- what are we computing here? **PINGO**
- $Y = a_n \cdot X^n + a_{n-1}X^{n-1} + \dots + a_0$
- evaluate polynomial at $X = x$ using `t=0,X=x`

General Full Retroactivity

Lemma: Lower Bound

Rewinding m operations has a lower bound of $\Omega(m)$ overhead

- general case

Proof (Sketch)

- two values X and Y
- initially $X = \emptyset$ and $Y = \emptyset$
- supported operations
  - $X = x$
  - $Y+ = value$
  - $Y = X \cdot Y$
  - $query\ Y$

Proof (Sketch, cnt.)

- perform operations
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n=1}$
  - $Y = X \cdot Y$
  - $\dots$
  - $Y+ = a_0$
- what are we computing here? **PINGO**
- $Y = a_n \cdot X^n + a_{n-1}X^{n-1} + \cdots + a_0$
- evaluate polynomial at $X = x$ using `t=0,X=x`
- this requires $\Omega(n)$ time [FHM01]

Priority Queues: Partial Retroactivity (1/6)

- priority queue with
  - insert
  - delete-min
- delete-min makes PQ non-commutative

Lemma: Partial Retroactive PQ

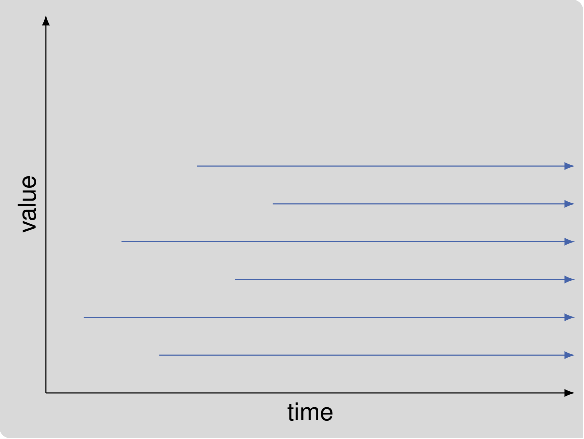
A priority queue can be partial retroactive with only $O(\log n)$ overhead per partially retroactive operation

Priority Queues: Partial Retroactivity (1/6)

- priority queue with
    - insert
    - delete-min
- delete-min makes PQ non-commutative

Lemma: Partial Retroactive PQ

A priority queue can be partial retroactive with only $O(\log n)$ overhead per partially retroactive operation

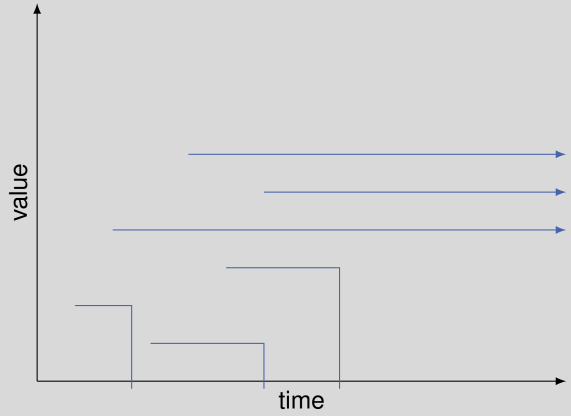


Priority Queues: Partial Retroactivity (1/6)

- priority queue with
    - insert
    - delete-min
- delete-min makes PQ non-commutative

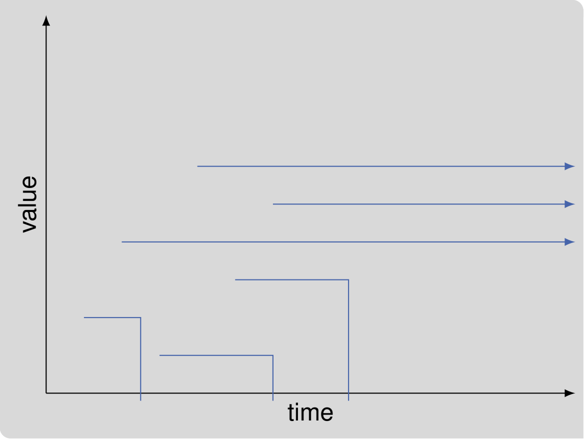
Lemma: Partial Retroactive PQ

A priority queue can be partial retroactive with only $O(\log n)$ overhead per partially retroactive operation



Priority Queues: Partial Retroactivity (2/6)

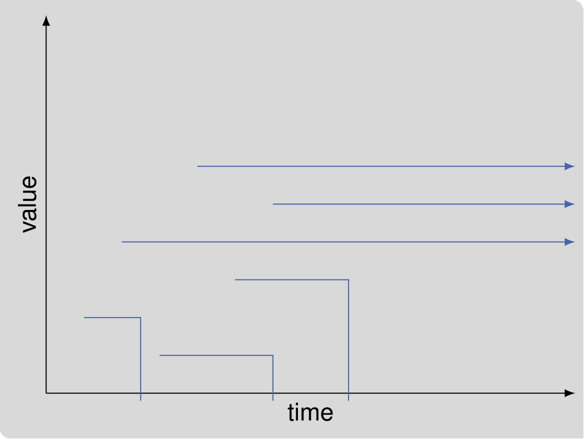
- what is the problem with
  - `INSERT(t,`delete-min()`)`
  - `INSERT(t,`insert`(i))`



Priority Queues: Partial Retroactivity (2/6)

- what is the problem with
  - `INSERT(t,`delete-min()`)`
  - `INSERT(t,`insert`(i))`

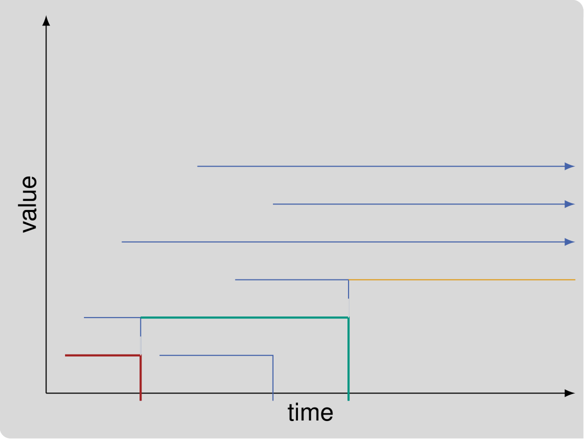
- `INSERT(t,`delete-min()`)` creates chain-reaction
- `INSERT(t,`insert`(i))` creates chain-reaction



Priority Queues: Partial Retroactivity (2/6)

- what is the problem with
  - `INSERT(t,`delete-min()`)`
  - `INSERT(t,`insert`(i))`

- `INSERT(t,`delete-min()`)` creates chain-reaction
- `INSERT(t,`insert`(i))` creates chain-reaction

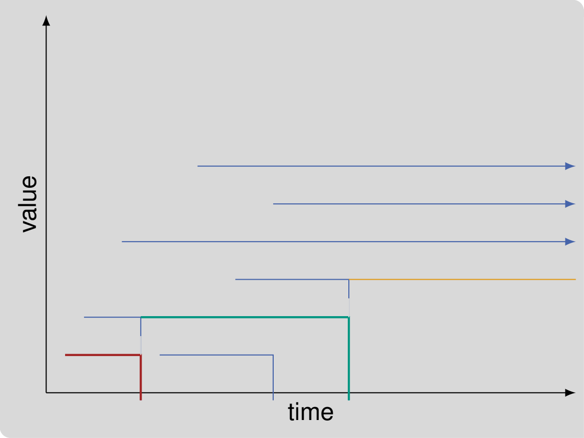


Priority Queues: Partial Retroactivity (2/6)

- what is the problem with
  - `INSERT(t,`delete-min())
  - `INSERT(t,`insert`(i))`

- `INSERT(t,`delete-min()) creates chain-reaction
- `INSERT(t,`insert`(i))` creates chain-reaction

- can we solve `DELETE(t,`delete-min`())` using `INSERT(t,`insert`(i))`? **PINGO**

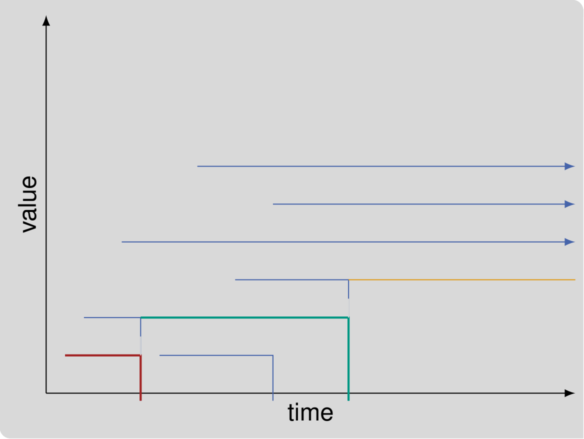


Priority Queues: Partial Retroactivity (2/6)

- what is the problem with
  - `INSERT(t,`delete-min()`)`
  - `INSERT(t,`insert`(i))`

- `INSERT(t,`delete-min()`)` creates chain-reaction
- `INSERT(t,`insert`(i))` creates chain-reaction

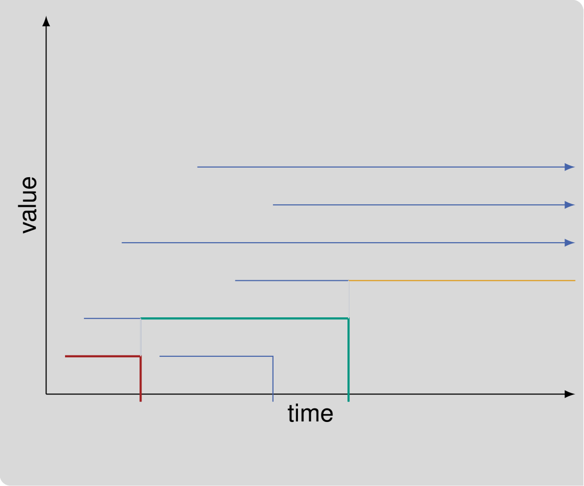
- can we solve `DELETE(t,`delete-min`())` using `INSERT(t,`insert`(i))`? **PINGO**
- insert deleted minimum right after deletion



Priority Queues: Partial Retroactivity (3/6)

- let Q_t be elements in PQ at time t

- what values are in Q_∞? ℹ partial retroactivity
- what value inserts $\texttt{INSERT}(t, \text{insert}(v))$ in Q_∞
- values is $\max\{v, v' : v' \text{ deleted at time} \geq t\}$
- maintaining deleted elements is hard ℹ can change a lot



Priority Queues: Partial Retroactivity (3/6)

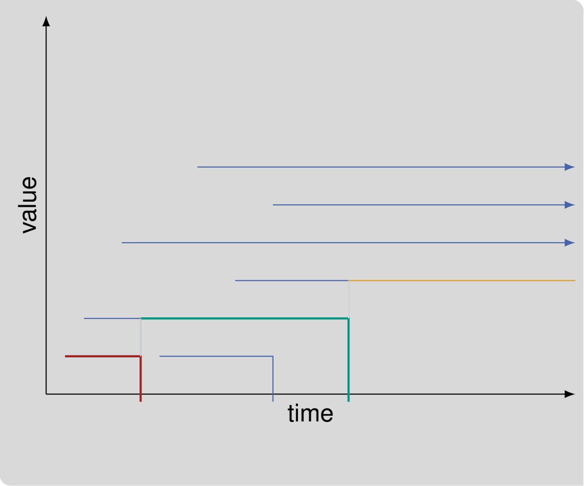
- let Q_t be elements in PQ at time t

- what values are in Q_∞? ℹ partial retroactivity
- what value inserts $\texttt{INSERT}(t, \text{insert}(v))$ in Q_∞
- values is $\max\{v, v' : v' \text{ deleted at time} \geq t\}$
- maintaining deleted elements is hard ℹ can change a lot

Definition: Bridge

A time t' is a bridge if $Q_{t'} \subseteq Q_\infty$

- all elements present at t' are present at t_∞



Priority Queues: Partial Retroactivity (3/6)

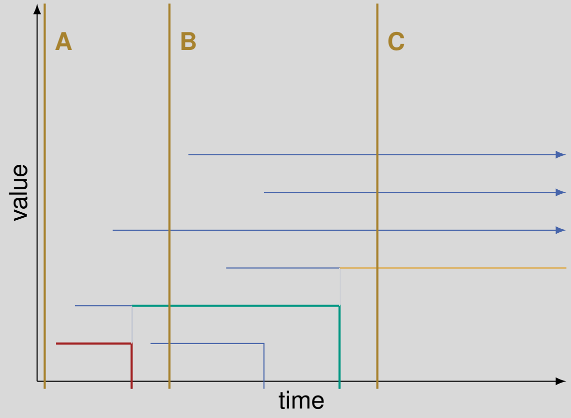
- let Q_t be elements in PQ at time t

- what values are in Q_∞? ℹ partial retroactivity
- what value inserts $\texttt{INSERT}(t, \text{insert}(v))$ in Q_∞
- values is $\max\{v, v' : v' \text{ deleted at time } \geq t\}$
- maintaining deleted elements is hard ℹ can change a lot

Definition: Bridge

A time t' is a bridge if $Q_{t'} \subseteq Q_\infty$

- all elements present at t' are present at t_∞



- what times are bridges? **PINGO**

Priority Queues: Partial Retroactivity (3/6)

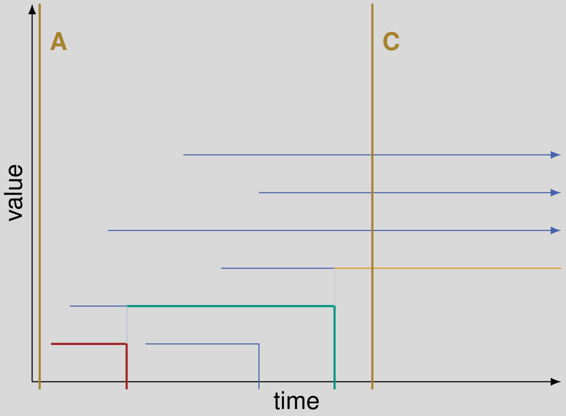
- let Q_t be elements in PQ at time t

- what values are in Q_∞? ⓘ partial retroactivity
- what value inserts $\texttt{INSERT}(t, \text{insert}(v))$ in Q_∞
- values is $\max\{v, v' : v' \text{ deleted at time} \geq t\}$
- maintaining deleted elements is hard ⓘ can change a lot

Definition: Bridge

A time t' is a bridge if $Q_{t'} \subseteq Q_\infty$

- all elements present at t' are present at t_∞



- what times are bridges? **PINGO**

Priority Queues: Partial Retroactivity (4/6)

Lemma: Deletions after Bridges

If time t' is closest bridge preceding time t, then

$$\max\{v' : v' \text{ deleted at time} \geq t\}$$

$$=$$

$$\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$$

Priority Queues: Partial Retroactivity (4/6)

Lemma: Deletions after Bridges

If time t' is closest bridge preceding time t, then

$$\max\{v' : v' \text{ deleted at time} \geq t\}$$

$$=$$

$$\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$$

Proof (Sketch)

- $\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\} \in \{v' : v' \text{ deleted at time} \geq t\}$
  - if maximum value is deleted between t' and t
  - then this time is a bridge
  - contradicting that t' is bridge preceding t

Priority Queues: Partial Retroactivity (4/6)

Lemma: Deletions after Bridges

If time t' is closest bridge preceding time t, then

$$\max\{v' : v' \text{ deleted at time} \geq t\}$$

$$=$$

$$\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$$

Proof (Sketch)

- $\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\} \in \{v' : v' \text{ deleted at time} \geq t\}$
  - if maximum value is deleted between t' and t
  - then this time is a bridge
  - contradicting that t' is bridge preceding t

Proof (Sketch, cnt.)

- $\max\{v' : v' \text{ deleted at time} \geq t\} \in \{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$
  - if v' is deleted at some time $\geq t$
  - then it is not in Q_∞

Priority Queues: Partial Retroactivity (4/6)

Lemma: Deletions after Bridges

If time t' is closest bridge preceding time t, then

$$\max\{v' : v' \text{ deleted at time} \geq t\}$$

$$=$$

$$\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$$

Proof (Sketch)

- $\max\{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\} \in \{v' : v' \text{ deleted at time} \geq t\}$
  - if maximum value is deleted between t' and t
  - then this time is a bridge
  - contradicting that t' is bridge preceding t

Proof (Sketch, cnt.)

- $\max\{v' : v' \text{ deleted at time} \geq t\} \in \{v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$
  - if v' is deleted at some time $\geq t$
  - then it is not in Q_∞

- what values are in Q_∞? ℹ partial retroactivity
- what value inserts $\texttt{INSERT}(t, \text{insert}(v))$ in Q_∞
- $\max\{v, v' \notin Q_\infty : v' \text{ inserted at time} \geq t'\}$

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ℹ changed for each update

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ℹ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty : v'$ inserted in subtree of $x\}$

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ⓘ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty \colon v' \text{ inserted in subtree of } x\}$
- BBST where leaves are all updates ordered by time augmented with
  - leaves store 0 for inserts with $v \in Q_\infty$, 1 for inserts with $v \notin Q_\infty$ and -1 for delete-mins
  - inner nodes store subtree sums

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ⓘ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty : v' \text{ inserted in subtree of } x\}$
- BBST where leaves are all updates ordered by time augmented with
  - leaves store 0 for inserts with $v \in Q_\infty$, 1 for inserts with $v \notin Q_\infty$ and -1 for delete-mins
  - inner nodes store subtree sums

- how can we find bridges? **PINGO**

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ℹ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty : v' \text{ inserted in subtree of } x\}$
- BBST where leaves are all updates ordered by time augmented with
  - leaves store 0 for inserts with $v \in Q_\infty$, 1 for inserts with $v \notin Q_\infty$ and -1 for delete-mins
  - inner nodes store subtree sums

- how can we find bridges? **PINGO**
- use third BBST and find prefix of updates summing to 0
- requires $O(\log n)$ time as we traverse tree at most twice
- this results in bridge t'

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ⓘ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty : v' \text{ inserted in subtree of } x\}$
- BBST where leaves are all updates ordered by time augmented with
  - leaves store 0 for inserts with $v \in Q_\infty$, 1 for inserts with $v \notin Q_\infty$ and -1 for delete-mins
  - inner nodes store subtree sums

- how can we find bridges? **PINGO**
- use third BBST and find prefix of updates summing to 0
- requires $O(\log n)$ time as we traverse tree at most twice
- this results in bridge t'

- use second BBST to identify maximum value not in Q_∞ on path to t'
- since BBST is augmented with these values, this requires $O(\log n)$ time

Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log n)$ overhead

- BBST for Q_∞ ℹ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node x store $\max\{v' \notin Q_\infty : v' \text{ inserted in subtree of } x\}$
- BBST where leaves are all updates ordered by time augmented with
  - leaves store 0 for inserts with $v \in Q_\infty$, 1 for inserts with $v \notin Q_\infty$ and -1 for delete-mins
  - inner nodes store subtree sums

- how can we find bridges? **PINGO**
- use third BBST and find prefix of updates summing to 0
- requires $O(\log n)$ time as we traverse tree at most twice
- this results in bridge t'

- use second BBST to identify maximum value not in Q_∞ on path to t'
- since BBST is augmented with these values, this requires $O(\log n)$ time

- update all BBSTs in $O(\log n)$ time

Priority Queues: Partial Retroactivity (6/6)

Lemma: Partial Retroactive PQ

A priority queue can be partial retroactive with only $O(\log n)$ overhead per partially retroactive operation

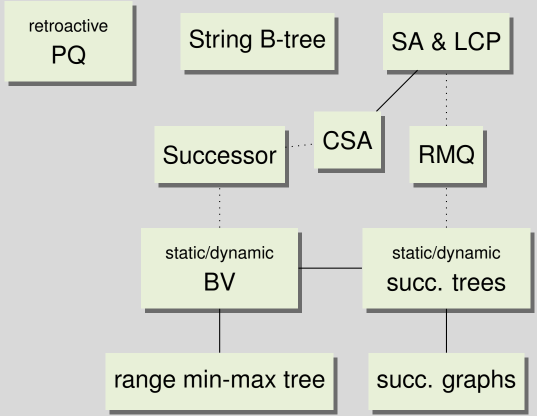
- requires three BBSTs
- updates need to update all BBSTs

Conclusion and Outlook

This Lecture

- retroactive data structures

Advanced Data Structures



Conclusion and Outlook

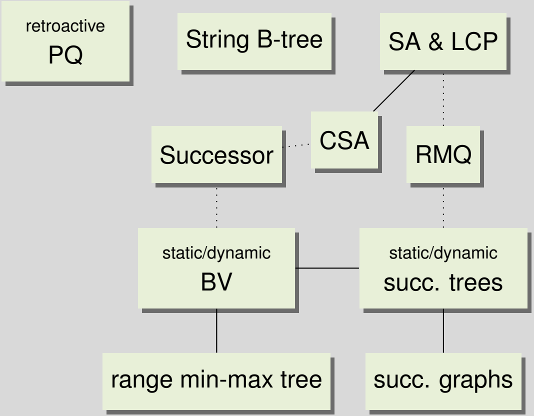
This Lecture

- retroactive data structures

Next Lecture

- (minimal) perfect hashing

Advanced Data Structures



Bibliography I

[FHM01] Gudmund Skovbjerg Frandsen, Johan P. Hansen, and Peter Bro Miltersen. “Lower Bounds for Dynamic Algebraic Problems”. In: *Inf. Comput.* 171.2 (2001), pages 333–349. DOI: `10.1006/inco.2001.3046`.