Exams

- 10.08.2022 and 29.09.2022
- write to blancani@kit.edu
  - full name
  - Matrikelnummer
  - PO version
  - date
- online or in person depending on situation/personal preferences
- 18.07.2022 Q&A during last half of lecture
Organization

Exams
- 10.08.2022 and 29.09.2022
- write to blancani@kit.edu
  - full name
  - Matrikelnummer
  - PO version
  - date
- online or in person depending on situation/personal preferences
- 18.07.2022 Q&A during last half of lecture

Evaluation
- now
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
  - 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
Recap: Persistent Data Structures

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

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

- **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

### Definitions

- **Partial Persistence**
  - Only the latest version can be updated

- **Full Persistence**
  - Any version can be updated

- **Confluent Persistence**
  - Like full persistence, but two versions can be combined to a new version

- **Functional**
  - Nodes cannot be modified, only new nodes can be created
Retroactive Data Structures

Operations

- INSERT\((t, operation)\): 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 \(\dagger\) for simplicity otherwise use order-maintenance data structure

0 1 2 3 4 now time

insert(7) insert(2) insert(3) del-min del-min queries
Retroactive Data Structures

**Operations**

- **INSERT**(t, 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** for simplicity otherwise use order-maintenance data structure

---

**Definition: Partial Retroactivity**

**QUERY** is only allowed for $t = \infty$ now

---

**Legend**

- insert
- delete-min

- now
- time

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>now</td>
</tr>
</tbody>
</table>

5/17 2022-07-04 Florian Kurpicz | Advanced Data Structures | 08 Temporal Data Structures 2

Institute of Theoretical Informatics, Algorithm Engineering
Retroactive Data Structures

### Operations
- **INSERT**\((t, operation)\): 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 for simplicity otherwise use order-maintenance data structure

### Definition: Partial Retroactivity
**QUERY** is only allowed for \(t = \infty\)

### Definition: Full Retroactivity
**QUERY** is allowed at any time \(t\)

<table>
<thead>
<tr>
<th>insert(7)</th>
<th>insert(2)</th>
<th>insert(3)</th>
<th>del-min</th>
<th>del-min</th>
<th>queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>now time</td>
</tr>
</tbody>
</table>
Retroactive Data Structures

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

For a priority queue updates are
- insert
- delete-min

Time is integer \(\triangleright\) for simplicity otherwise use order-maintenance data structure

**Definition: Partial Retroactivity**

QUERY is only allowed for \(t = \infty \triangleright\) 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

- commutative operations
  - insert and delete-min are not commutative
  - insert and delete are commutative
- invertible updates
  - operation $op^{-1}$ such that $op^{-1}(op(\cdot)) = \emptyset$
  - DELETE becomes INSERT inverse operation
- makes partial retroactivity easy
- $\text{INSERT}(t, \text{operation}) = \text{INSERT}(\infty, \text{operation})$
- $\text{DELETE}(t, op) = \text{INSERT}(\infty, op^{-1})$
Easy Cases: Partial Retroactivity

- Commutative operations
  - Insert and delete-min are not commutative
  - Insert and delete are commutative
- Invertible updates
  - Operation $op^{-1}$ such that $op^{-1}(op(\cdot)) = \emptyset$
  - Delete becomes insert inverse operation
- Makes partial retroactivity easy
- $\text{INSERT}(t, \text{operation}) = \text{INSERT}(\infty, \text{operation})$
- $\text{DELETE}(t, op) = \text{INSERT}(\infty, op^{-1})$

Partial Retroactivity

- Hashing
- Dynamic dictionaries
- Array with updates only $A[i]+ = \text{value}$
Definition: Search Problem

A search problem is a problem on a set $S$ of objects with operations $\text{insert}$, $\text{delete}$, and $\text{query}(x, S)$.
Definition: Search Problem
A search problem is a problem on a set $S$ of objects with operations \textit{insert}, \textit{delete}, and \textit{query}(x, S)

Definition: Decomposable Search Problem
A decomposable search problem is a search problem, with
- \textit{query}(x, A \cup B) = f(\textit{query}(x, A), \textit{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 \textit{insert}, \textit{delete}, and \textit{query}(x, S)

Definition: Decomposable Search Problem

A decomposable search problem is a search problem, with

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

which decomposable search problem have we seen PINGO
Search Problems

Definition: Search Problem
A search problem is a problem on a set $S$ of objects with operations $\text{insert}$, $\text{delete}$, and $\text{query}(x, S)$.

Definition: Decomposable Search Problem
A decomposable search problem is a search problem, with

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

- 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 $\text{query}(x, S)$

Definition: Decomposable Search Problem
A decomposable search problem is a search problem, with
- $\text{query}(x, A \cup B) = f(\text{query}(x, A), \text{query}(x, B))$
- with $f$ requiring $O(1)$ time

- predecessor and successor search
- range minimum queries
- nearest neighbor
- point location
- ...
- these types of problems are also “easy”

- which decomposable search problem have we seen PINGO
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.
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
- 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 all node $n$ if $[s_n, e_n] \subseteq [s, e]$ if non of $n$'s ancestors’ are $\subseteq [s, e]$
- each object occurs in $O(\log n)$ nodes
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
- 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 all node $n$ if $[s_n, e_n] \subseteq [s, e]$ if non 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
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
- 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 all node $n$ if $[s_n, e_n] \subseteq [s, e]$ if non of $n$'s ancestors’ are $\subseteq [s, e]$
- each object occurs in $O(\log n)$ nodes

Proof (Sketch, cont.)

- 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!
Lemma: Lower Bound
Rewinding $m$ operations has a lower bound of $\Omega(m)$ overhead

- general case
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
  - \( X = x \)
  - \( Y+ = \text{value} \)
  - \( Y = X \cdot Y \)
  - query \( Y \)
Lemma: Lower Bound

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

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
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n-1}$
  - $Y = X \cdot Y$
  - ...$
  - $Y+ = a_0$
- what are we computing here? PINGO
Lemma: Lower Bound

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

Proof (Sketch)

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

$Y = a_n \cdot X^n + a_{n-1}X^{n-1} + \cdots + a_0$
Lemma: Lower Bound
Rewinding \( m \) operations has a lower bound of \( \Omega(m) \) overhead

Proof (Sketch)
- two values \( X \) and \( Y \)
- initially \( X = \emptyset \) and \( Y = \emptyset \)
- supported operations
  - \( X = x \)
  - \( Y+ = \text{value} \)
  - \( Y = X \cdot Y \)
  - 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
- \( Y = a_n \cdot X^n + a_{n-1}X^{n-1} + \cdots + a_0 \)
- evaluate polynomial at \( X = x \) using \( t=0, X=x \)
Lemma: Lower Bound

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

Proof (Sketch)

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

Proof (Sketch, cnt.)

- perform operations
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n-1}$
  - $Y = X \cdot Y$
  - ...$
  - $Y+ = a_0$
- 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
what is the problem with
- INSERT(t, delete-min())
- INSERT(t, insert(i))

Can we solve DELETE(t, delete-min()) using INSERT(t, insert(i))?
what is the problem with
- \text{INSERT}(t, \text{delete-min}())
- \text{INSERT}(t, \text{insert}(i))

- \text{INSERT}(t, \text{delete-min}()) creates chain-reaction
- \text{INSERT}(t, \text{insert}(i)) creates chain-reaction
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
what is the problem with
- \text{INSERT}(t, \text{delete-min}())
- \text{INSERT}(t, \text{insert}(i))

\text{INSERT}(t, \text{delete-min}()) \text{ creates chain-reaction}
\text{INSERT}(t, \text{insert}(i)) \text{ creates chain-reaction}

can we solve \text{DELETE}(t, \text{delete-min}()) \text{ using} \text{INSERT}(t, \text{insert}(i))? \text{ PINGO}
what is the problem with
  - \texttt{INSERT(t, delete-min())}
  - \texttt{INSERT(t, \text{insert}(i))}

\texttt{INSERT(t, delete-min())} creates chain-reaction
\texttt{INSERT(t, \text{insert}(i))} creates chain-reaction

can we solve \texttt{DELETE(t, delete-min())} using
\texttt{INSERT(t, \text{insert}(i))}? PINGO
insert deleted minimum right after deletion
let $Q_t$ be elements in PQ at time $t$

what values are in $Q_\infty$? \piring partial retroactivity

what value inserts $\text{INSERT}(t, \text{insert}(v))$ in $Q_\infty$

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

maintaining deleted elements is hard \piring can change a lot

Priority Queues: Partial Retroactivity (3/6)
let $Q_t$ be elements in PQ at time $t$

what values are in $Q_\infty$? partial retroactivity
what value inserts \text{INSERT}(t, \text{insert}(v)) in $Q_\infty$
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 $t_\infty$
Priority Queues: Partial Retroactivity (3/6)

- let $Q_t$ be elements in PQ at time $t$
- what values are in $Q_\infty$? partial retroactivity
- what value inserts $\text{INSERT}(t, \text{insert}(v))$ in $Q_\infty$
- 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 $t_\infty$
Let $Q_t$ be elements in PQ at time $t$.

- What values are in $Q_\infty$? (partial retroactivity)
- What value inserts $\text{INSERT}(t, \text{insert}(v))$ in $Q_\infty$?
- 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 $t_\infty$
Lemma: Deletions after Bridges

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

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

= 

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

- If maximum value is deleted between $t'$ and $t$
- Then this time is a bridge
- Contradicting that $t'$ is bridge preceding $t$
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 $Q_\infty$
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)

- \( \max \{ v' \notin Q_\infty : v' \text{ inserted at time } \geq t' \} \in \{ v' \notin Q_\infty : v' \text{ inserted at time } \geq t' \} \)
  - 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 \( Q_\infty \)

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

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 $t'_1$

- use second BBST to identify maximum value not in $Q_\infty$ on path to $t'_1$
- 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 (5/6)

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

- BBST for $Q_\infty$ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node $x$ store $\max\{v' \notin Q_\infty : v' \text{ inserted in subtree of } x\}$
Priority Queues: Partial Retroactivity (5/6)

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

- BBST for $Q_\infty$ 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 $x\}$
- 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?
- Use a 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 $Q_\infty$ 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 (5/6)

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

- BBST for $Q_\infty$ 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 $x\}$
- 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 $Q_\infty$ 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 $Q_\infty$ changed for each update
- BBST where leaves are inserts ordered by time augmented with
  - for each node $x$ store $\max\{v' \not\in Q_\infty : v' \text{ inserted in subtree of } x\}$
- 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 $v \not\in Q_\infty$ and $-1$ for delete-mins
  - inner nodes store subtree sums

- how can we find bridges?
- 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 $Q_\infty$ on path to $t'$
- since BBST is augmented with these values, this requires $O(\log n)$ time
keep track of inserted values
use balanced binary search trees for $O(\log n)$ overhead

BBST for $Q_\infty$ changed for each update
BBST where leaves are inserts ordered by time augmented with
  - for each node $x$ store $\max\{v' \not\in Q_\infty : v' \text{ inserted in subtree of } x\}$
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 $v \not\in Q_\infty$ and $-1$ for delete-mins
  - inner nodes store subtree sums

how can we find bridges?
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 $Q_\infty$ 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
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
Nonoblivious Retroactivity

- priority queue with
  - insert
  - delete
  - min

- identify queries that are now incorrect
  - using ray shooting 🎨
Conclusion and Outlook

This Lecture
- retroactive data structures

Advanced Data Structures

- retroactive PQ
- String B-tree
- SA & LCP
- Successor
- CSA
- RMQ
- static/dynamic BV
- static/dynamic succ. trees
- range min-max tree
- succ. graphs
Conclusion and Outlook

This Lecture
- retroactive data structures

Next Lecture
- geometric data structures

Advanced Data Structures

- retroactive
- String B-tree
- SA & LCP
- PQ
- CSA
- RMQ
- static/dynamic
- BV
- static/dynamic
- succ. trees
- range min-max tree
- succ. graphs