External Memory Model [AV88]

Definition: External Memory Model

- internal memory of $M$ words
- instances of size $N \gg M$
- unlimited external memory
- transfer blocks of size $B$ between memories

- measure number of blocks I/Os
- scanning $N$ elements: $\Theta(N/B)$
- sorting $N$ elements: $\Theta\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$
External Memory Model [AV88]

**Definition: External Memory Model**
- internal memory of $M$ words
- instances of size $N \gg M$
- unlimited external memory
- transfer blocks of size $B$ between memories
- measure number of blocks I/Os
- scanning $N$ elements: $\Theta(N/B)$
- sorting $N$ elements: $\Theta(N/B \log_{M/B} N)$

**Set of Strings**
- alphabet $\Sigma$ of size $\sigma$
- $k$ strings $\{s_1, \ldots, s_k\}$ over the alphabet $\Sigma$
- total size of strings is $N = \sum_{i=1}^{k} |s_i|$
- queries ask for pattern $P$ of length $m$
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:
- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

**Definition: Trie**

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique
Given a set $S \subseteq \Sigma^*$ of prefix-free strings, we want to answer:

- is $x \in \Sigma^*$ in $S$
- add $x \notin S$ to $S$
- remove $x \in S$ from $S$
- predecessor and successor of $x \in \Sigma^*$ in $S$

### Definition: Trie

Given a set $S = \{S_1, \ldots, S_k\}$ of prefix-free strings, a trie is a labeled rooted tree $G = (V, E)$ with:

1. $k$ leaves
2. $\forall S_i \in S$ there is a path from the root to a leaf, such that the concatenation of the labels is $S_i$
3. $\forall v \in V$ the labels of the edges $(v, \cdot)$ are unique

$S = \{\text{bear, bee, cab, car}\}$
# Theoretical Comparison

<table>
<thead>
<tr>
<th>Representation</th>
<th>Query Time (Contains)</th>
<th>Space in Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays of variable size</td>
<td>$O(m \cdot \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>arrays of fixed size</td>
<td>$O(m)$</td>
<td>$O(N \cdot \sigma)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$O(m)$ w.h.p.</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>$O(m \cdot \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>weight-balanced search trees</td>
<td>$O(m + \lg k)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-levels with weight-balanced search trees</td>
<td>$O(m + \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
### Theoretical Comparison

<table>
<thead>
<tr>
<th>Representation</th>
<th>Query Time (Contains)</th>
<th>Space in Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrays of variable size</td>
<td>$O(m \cdot \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>arrays of fixed size</td>
<td>$O(m)$</td>
<td>$O(N \cdot \sigma)$</td>
</tr>
<tr>
<td>hash tables</td>
<td>$O(m)$ w.h.p.</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>balanced search trees</td>
<td>$O(m \cdot \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>weight-balanced search trees</td>
<td>$O(m + \lg k)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>two-levels with weight-balanced search trees</td>
<td>$O(m + \lg \sigma)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>

[more details in lecture Text Indexing](#)
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

**Definition: Compact Trie**

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
Compact Trie

- tries have unnecessary nodes
- branchless paths can be removed
- edge labels can consist of multiple characters

Definition: Compact Trie

- A compact trie is a trie where all branchless paths are replaced by a single edge.
- The label of the new edge is the concatenation of the replaced edges’ labels.
(Recap) B-Trees

- search tree with out-degree in \([b, 2b)\)
- works well in external memory
- uses separators to find subtree
- can be dynamic
- who knows B-trees?

example on the board

From Atomic Values to Strings

- strings take more time to compare
- load as few strings from disk as possible
String B-Tree [FG99]

- Strings are stored in EM.
- Strings are identified by starting positions.

B-tree layout for sorted suffixes identified by position:
- At least $b = \Theta(B)$ children.
- Tree height $O(\log_B N)$.

Given node $v$ with children $v_0, \ldots, v_k$ with $k \in [b, 2b)$.
- Inner: store separators $L(v_0), R(v_0), \ldots, L(v_k), R(v_k)$.
- Leaf: store strings and link leaves.

Given node $v$:
- $L(v)$ is lexicographically smallest string at $v$.
- $R(v)$ is lexicographically largest string at $v$. 
Search in String B-Tree

- task: find all occurrences of pattern \( P \)
- two traversals of String B-Tree
- identify leftmost/rightmost occurrence
- output all strings in \( O(\text{occ}/B) \)

- at every node with children \( v_0, \ldots, v_k \)
- binary search for \( P \) in \( L(v_0), \ldots, R(v_k) \)
  - if \( R(v_i) < P < L(v_{i+1}) \): not found
  - if \( L(v_i) \leq P \leq R(v_i) \): continue in \( v_i \)

Lemma: String B-Tree

Using a String B-tree, a pattern \( P \) can be found in a set of strings with total length \( N \) in \( O(|P|/B \log N) \) I/Os

Proof (Sketch)

- String B-Tree has height \( \log_B N \)
- load separators of node: \( O(1) \) I/O
- load strings for binary search: \( O(|P|/B) \) I/Os
- total: \( O(\log_B N \cdot \log B \cdot |P|/B) = O(|P|/B \log N) \) I/Os
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{S_0, \ldots, S_{k-1}\}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings $S = \{ S_0, \ldots, S_{k-1} \}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings

How do Patricia tries help?
Improving String B-Tree with Patricia Tries (1/2)

Patricia Trie

- for strings \( S = \{ S_0, \ldots, S_{k-1} \} \)
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size \( O(k) \) for \( k \) strings

- search requires two steps
  - first **blind search** using only trie
  - blind search can result in false matches
  - second a comparison with resulting string
  - use any leaf after matching pattern
Patricia Trie

- for strings $S = \{ S_0, \ldots, S_{k-1} \}$
- a compact trie where only branching characters are stored
- additionally the string depth is stored
- size $O(k)$ for $k$ strings

- search requires two steps
- first blind search using only trie
- blind search can result in false matches
- second a comparison with resulting string
- use any leaf after matching pattern

How do Patricia tries help?
Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf \( w \)
- compute \( L = \text{lcp}(P, w) \)
- let \( u \) be first node on root-to-\( w \) path with \( d \geq L \)
Improving String B-Tree with Patricia Tries (2/2)

- in each inner node build Patricia trie for separators
- if blind search finds leaf $w$
- compute $L = \text{lcp}(P, w)$
- let $u$ be first node on root-to-$w$ path with $d \geq L$

$d = L$

- find matching children $v_i$ and $v_{i+1}$ of $w$ with
- branching characters $c_i < P[L + 1] < c_{i+1}$
- example on the board 📚
Improving String B-Tree with Patricia Tries (2/2)

- In each inner node build Patricia trie for separators
- If blind search finds leaf $w$
- Compute $L = lcp(P, w)$
- Let $u$ be first node on root-to-$w$ path with $d \geq L$

### $d > L$
- Consider next branching character $c$ on path
- If $P[L + 1] < c$ continue in leftmost leaf
- If $P[L + 1] > c$ continue in rightmost leaf

### $d = L$
- Find matching children $v_i$ and $v_{i+1}$ of $w$ with
- Branching characters $c_i < P[L + 1] < c_{i+1}$
- Example on the board
Searching in Improved String B-Tree

- at every node with children \( v_0, \ldots, v_k \)
- load Patricia trie for \( L(v_0), \ldots, R(v_k) \)
- search Patricia trie for \( w \) result of blind search
- load one string and compare with \( P \)
- identify child and continue
Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for $w$ \(\triangleright\) result of blind search
- load one string and compare with $P$
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern $P$ can be found in a set of strings with total length $N$ with $O(|P|/B \log_B N)$ I/Os.
Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for $w$  
  result of blind search
- load one string and compare with $P$
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern $P$ can be found in a set of strings with total length $N$ with $O(|P|/B \log_B N)$ I/Os

Proof (Sketch)

- loading PT: $O(1)$ I/Os
- blind search: no I/Os
- loading one string: $O(|P|/B)$ I/Os
- identify child: no I/Os
- total $O(|P|/B \log_B N)$ I/Os
Searching in Improved String B-Tree

- at every node with children $v_0, \ldots, v_k$
- load Patricia trie for $L(v_0), \ldots, R(v_k)$
- search Patricia trie for $w$ \(\triangleright\) result of blind search
- load one string and compare with $P$
- identify child and continue

Lemma: String B-Tree with PTs

Using a string B-tree with Patricia tries, a pattern $P$ can be found in a set of strings with total length $N$ with $O(|P|/B \log_B N)$ I/Os.

Proof (Sketch)

- loading PT: $O(1)$ I/Os
- blind search: no I/Os
- loading one string: $O(|P|/B)$ I/Os
- identify child: no I/Os
- total $O(|P|/B \log_B N)$ I/Os

How can this be improved even further?

PINGO
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree \( p_0, p_1, p_2, \ldots \)
- in Patricia tries \( PT_{p_i} \), compute \( L = \text{lcp}(P, w) \)
- all strings in \( p_i \) have prefix \( P[0..L) \)
- do not compare previously matched characters
- load only \( |P| - L \) characters at next node
- pass \( L \) down the String B-tree
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree \( p_0, p_1, p_2, \ldots \)
- in Patricia tries \( PT_{p_i} \), compute \( L = lcp(P, w) \)
- all strings in \( p_i \) have prefix \( P[0..L] \)
- do not compare previously matched characters
- load only \( |P| - L \) characters at next node
- pass \( L \) down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern \( P \) can be found in a set of strings with total length \( N \) in \( O(|P|/B + \log_B N) \) I/Os.
Improving Search with LCP-Values

- search for pattern in nodes
- path in String B-tree $p_0, p_1, p_2, \ldots$
- in Patricia tries $PT_{p_i}$ compute $L = \text{lcp}(P, w)$
- all strings in $p_i$ have prefix $P[0..L)$
- do not compare previously matched characters
- load only $|P| - L$ characters at next node
- pass $L$ down the String B-tree

Lemma: String B-Tree with PTs and LCP

Using a String B-tree with Patricia tries and passing down the LCP-value, a pattern $P$ can be found in a set of strings with total length $N$ in $O(|P|/B + \log B N)$ I/Os.

Proof (Sketch)

- passing down LCP-value: no I/Os
- telescoping sum $\sum_{i \leq h} \frac{L_i - L_{i-1}}{B}$
- $h = \log B N$ \(\#\) height of String B-tree
- $L_i$ is LCP-value on Level $i$
- $L_0 = 0$ and $L_h \leq |P|$
- total: $O(|P|/B + \log B N)$ I/Os
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
  - 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

- 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

### Definitions

**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

- **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, 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 \( \text{for simplicity otherwise use order-maintenance data structure} \)
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

\textbf{Definition: Partial Retroactivity}

QUERY is only allowed for \(t = \infty\) \(\bigcirc\) now

\textbf{Definition: Full Retroactivity}

QUERY is allowed at any time \(t\)

\textbf{Definition: Nonoblivious Retroactivity}

INSERT, DELETE, and QUERY at any time \(t\) but also identify changed QUERY results

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

2024-06-17 Florian Kurpicz | Advanced Data Structures | 09 String B-Trees & 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 \(t = \infty \circlearrowleft\) now

Definition: Full Retroactivity

**QUERY** is allowed at any time \(t\)

---

Observations:

- \(insert(7)\) insert(2) insert(3) del-min del-min
- \(queries\)
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 for simplicity otherwise use order-maintenance data structure

**Definition: Partial Retroactivity**

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

<table>
<thead>
<tr>
<th>insert(7)</th>
<th>insert(2)</th>
<th>insert(3)</th>
<th>del-min</th>
<th>del-min</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

queries

now  time
Easy Cases: Partial Retroactivity

- commutative updates
- invertible updates
  - operation $op^{-1}$ such that $op^{-1}(op(\cdot)) = \emptyset$
  - DELETE becomes INSERT inverse operation
- makes partial retroactivity easy
- $\text{INSERT}(t, operation) = \text{INSERT}(\infty, operation)$
- $\text{DELETE}(t, op) = \text{INSERT}(\infty, op^{-1})$
Easy Cases: Partial Retroactivity

- commutative updates
- 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 $A[i]+ = \text{value}$
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: Search Problem
A search problem is a problem on a set \( S \) of objects with operations \( \text{insert}, \ \text{delete}, \ \text{and} \ \text{query}(x, S) \).

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
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

**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

- predecessor and successor search
- range minimum queries
- nearest neighbor
- point location
- ...
**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
- . . .
- 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 problem 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

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: 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 or 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
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, 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

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

- 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
  - \( X = x \)
  - \( Y + = value \)
  - \( Y = X \cdot Y \)
  - \( query \ Y \)
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
  - $X = x$
  - $Y+ = \text{value}$
  - $Y = X \cdot Y$
  - 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
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
  - \( X = x \)
  - \( Y+ = \text{value} \)
  - \( Y = X \cdot Y \)
  - \( \text{query } Y \)

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
  - \( X = x \)
  - \( Y+ = value \)
  - \( Y = X \cdot Y \)
  - 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
- \( Y = a_n \cdot X^n + a_{n-1} X^{n-1} + \ldots + a_0 \)
- 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
  - $Y+ = a_n$
  - $Y = X \cdot Y$
  - $Y+ = a_{n-1}$
  - $Y = X \cdot Y$
  - \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 $t=0, X=x$
- this requires $\Omega(n)$ time [FHM01]
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 $O(\log m)$ 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 $O(\log m)$ 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 $O(\log m)$ 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
  - $\text{INSERT}(t, \text{delete-min}())$
  - $\text{INSERT}(t, \text{insert}(i))$

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

- can we solve $\text{DELETE}(t, \text{delete-min}())$ using $\text{INSERT}(t, \text{insert}(i))$? PINGO
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

can we solve \( \text{DELETE}(t, \text{delete-min}()) \) using
\( \text{INSERT}(t, \text{insert}(i)) \)?
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 $Q_\infty$? 1 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 1 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 INSERT($t$, 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 $t_\infty$

what times are bridges?
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 $t_\infty$

What times are bridges?
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

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

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

Proof (Sketch)

- $\max\{v' \not\in Q_\infty : v' \text{ inserted at time } t' \geq t\} \in \{v' : v' \text{ deleted at time } t' \geq t\}$
  - 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

$$\max\{v': v' \text{ deleted at time } \geq t\} = \max\{v' \not\in Q_{\infty} : v' \text{ inserted at time } \geq t'\}$$

Proof (Sketch)

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

- what values are in $Q_\infty$? partial retroactivity
- what value inserts $\text{INSERT}(t, \text{insert}(v))$ in $Q_\infty$
- $\max\{v, v' \notin Q_\infty : v' \text{ inserted at time } \geq t'\}$
Priority Queues: Partial Retroactivity (5/6)

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

BBST for $Q_{\infty}$ changed for each update
Priority Queues: Partial Retroactivity (5/6)

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

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

- 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
Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log m)$ 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
Priority Queues: Partial Retroactivity (5/6)

- keep track of inserted values
- use balanced binary search trees for $O(\log m)$ 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 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
keep track of inserted values
use balanced binary search trees for $O(\log m)$ 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? 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 $t'$
keep track of inserted values
use balanced binary search trees for $O(\log m)$ overhead

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

- keep track of inserted values
- use balanced binary search trees for $O(\log m)$ 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 $t'$

- 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 $O(\log m)$ overhead per partially retroactive operation.

- requires three BBSTs
- updates need to update all BBSTs
Conclusion and Outlook

This Lecture
- string B-tree
- 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

Institute of Theoretical Informatics, Algorithm Engineering
Conclusion and Outlook

This Lecture
- string B-tree
- retroactive data structures

Next Lecture
- learned 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
Bibliography I

