Advanced Data Structures

Lecture 10: Retroactive Data Structures (cnt.) and Minimal Perfect Hashing

Florian Kurpicz
Recap: Retroactive Data Structures

Operations

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

\text{QUERY} \text{ is only allowed for } t = \infty \text{ now}

Definition: Full Retroactivity

\text{QUERY} \text{ is allowed at any time } t

Definition: Nonoblivious Retroactivity

\text{INSERT, DELETE, and QUERY} \text{ 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>
<tr>
<td>now</td>
<td>time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3/31

Florian Kurpicz | Advanced Data Structures | 10 Retroactive DS (cnt.) & Minimal Perfect Hashing
Institute of Theoretical Informatics, Algorithm Engineering
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
- $\text{INSERT}(t, \text{delete-min}())$
- $\text{INSERT}(t, \text{insert}(i))$

- $\text{INSERT}(t, \text{delete-min}())$ 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
- insert deleted minimum right after deletion
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 $Q_{t'} \subseteq Q_\infty$

what times are bridges? PINGO
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, 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 $Q_\infty$
- $\max\{v, v' \notin Q_\infty : v' \text{ inserted at time } \geq t'\}$

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 (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? 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
Hashing (1/2)

**Definition: Totally Random**
- \( h: \{0, \ldots, u - 1\} \rightarrow \{0, \ldots, m - 1\} \)
- \( n \) objects
- from universe \( U = \{0, \ldots, u - 1\} \)
- hash table of size \( m \) close to \( n \)
- \( m \ll u \)

**Definition: Universal**
- choose \( h \) from family \( H \) with 
  \( \Pr_{h \in H}[h(x) = h(y)] = O(1/m) \) for all 
  \( x \neq y \in U \)
- family is small to enable efficient encoding

\[ h(x) = (ax \mod u) \mod m \] for \( 0 < a < p \) and \( p \) being prime \( > u \)

\[ h(x) = ax \gg (\log u - \log m) \] for \( m, u \) being powers of two

Why is this family easier to store?
### Hashing (2/2)

#### Definition: $k$-wise Independent
- Choose $h$ from family $H$ with
  \[ P[h(x_1) = t_1 \& \ldots \& h(x_k) = t_k] = O(1/m^k) \]
  for distinct $x_1, \ldots, x_k \in U$
- Implies universal
- $h(x) = ((\sum_{i=0}^{k-1} a_ix^i) \mod p) \mod m$ for $0 \leq a_i < p$ and $0 < a_{k-1} < p$
- Pairwise ($k = 2$) independence is stronger than universal
- $h(x) = ((ax + b) \mod u) \mod m$

#### Definition: Simple Tabulation Hashing
- View $x$ as vector $x_1, \ldots, x_c$ of characters
- Totally random hash table $T_i$ for each character
- $h(x) = T_1(x_1) \oplus \ldots \oplus T_c(x_c)$
- Why can we use totally random hash tables?
- $O(cu^{1/c})$ space
- $O(c)$ time to compute
- 3-wise independent
Minimal Perfect Hashing

**Definition: Perfect Hash Function**
- injective hash function
- maps $n$ objects to $m$ slots
- lower space bound for $m = (1 + \epsilon)n$ is
  \[
  \log e - \epsilon \log \frac{1 + \epsilon}{\epsilon}
  \]
- for $m$ close to $n$ there are likely collisions

**Definition: Minimal Perfect Hash Function**
- bijective hash function
- maps $n$ objects to $m = n$ slots
- $h: N \rightarrow [0, n)$
- lower space bound as for PHF with $\epsilon = 0$:
  \[
  \log e \approx 1.44
  \]
- no collisions
- can we make PHF to MPHF?

PINGO

12/31 2024-07-01 Florian Kurpicz | Advanced Data Structures | 10 Retroactive DS (cnt.) & Minimal Perfect Hashing Institute of Theoretical Informatics, Algorithm Engineering
for each object calculate three potential slots \((h_0, h_1, \text{ and } h_2)\)
- for each slot that contains only one object, remove the object from all its other slots
- one slot per object
- if that does not work use other hash functions
- use rank data structure to map slots to \([0, n)\)

how to check if hash function works
- interpret each slot as node in a hypergraph
- objects are edges
- if graph is peelable, we have a feasible mapping

Definition: Peelable
A hypergraph is peelable, if it is possible to obtain a graph without edges by iteratively taking away edges that contain a node with degree 1

1.95 bits per object when \(m = 1.23n\)
Compress, Hash, and Displace [BBD09a]

- partition keys into buckets
- set \( m = (1 + \epsilon)n \)
- sort partitions by size
- starting with largest bucket, find universal hash function mapping all keys to empty slots
- if key mapped to non-empty slot, try next hash function
- for each bucket store universal hash function
- use rank data structure to map slots to \([0, n)\)

- can be used as PHF
- there are a lot of tricks w.r.t. bucket sizes and size distributions
- requires around 2.05 bits per object

example on the board
RecSplit Overview \textbf{[EGV20a]} \\

- partition keys into buckets of size $b$ \\
- for each bucket compute splitting trees \\
- split keys into smaller sets \\
- stop when sets have size $\ell$

- upper aggregation levels have fanout 2 \\
- lower two aggregation levels have fanout \\
  - $\max\{2, \lceil 0.35\ell + 0.55 \rceil \}$ \\
  - $\max\{2, \lceil 0.21\ell + 0.9 \rceil \}$

- last level is leaf level \\
- find bijections
RecSplit Splitting Tree

- tree structure is well defined
- store information for each node in preorder
- store hash function for each splitter
- encode function using Golomb-Rice

- encodings of splitting trees stored in one bit vector
- use Elias-Fano to store
  - size of buckets
  - starting position of bucket in bit vector
Definition: Golomb Code

Given an integer $x > 0$ and a constant $b > 0$, the Golomb code consists of:

- $q = \lfloor \frac{x}{b} \rfloor$
- $r = x - qb = x \% b$
- $c = \lceil \lg b \rceil$

with

$$(x)_{\text{Gol}(b)} = (q)_1(r)_2$$

where $(r)_2$ depends on its size

- $r < 2^{\lfloor \lg b \rfloor - 1}$: $r$ requires $\lfloor \lg b \rfloor$ bits and starts with a 0
- $r \geq 2^{\lfloor \lg b \rfloor - 1}$: $r$ requires $\lfloor \lg b \rfloor$ bits and starts with a 1 and it encodes $r - 2^{\lfloor \lg b \rfloor - 1}$

- $b$ has to be fixed for all codes
- still variable length

- Golomb-Rice is special case where $r$ is power of two

- for $b = 5$, there are 4 remainders: 00, 01, 100, 101, and 110
- $2^{\lfloor \lg 5 \rfloor - 1} = 2$
- 0, 1 < 2: 00 and 01 require 2 bits
- 2, 3, 4 ≥ 2: require 3 bits and encode 0, 1, 2 starting with 1
Comparison of Codes (1/2)

![Comparison of Codes](image-url)

Legend:
- **unary**
- **ternary**
- **Fibonacci**
- **Elias-γ**
- **Elias-δ**
- **Golomb \((b = 5)\)**
- **Golomb \((b = 10^6)\)**
Comparison of Codes (2/2)

![Graph comparing different encoding methods](image)

- Unary
- Ternary
- Fibonacci
- Elias-γ
- Elias-δ
- Golomb ($b = 5$)
- Golomb ($b = 10^6$)
RecSplit Leaves

- find perfect hash function for keys in leaves
- test hash functions brute force
- use hash value modulo $\ell$
- set bit in “bit vector” of length $\ell$
- all bits set indicates bijection
RecSplit Queries

- find bucket
- follow splitting tree
- accumulate number of objects to the left
- use bijection in leaf
- result is sum of
  - objects in previous buckets
  - objects to the left in splitting tree
  - value of bijection
Parallel RecSplit

- based on a Domink Bez’ Master’s thesis
randomly distribute objects in leaf in two sets $A$ and $B$
hash objects in both set
two “bit vectors”: cyclic shift one until all bits are set when $0$Red
store hash function and rotation

Lemma: Rotation Fitting

Let $|A| = A$, $|B| = B$, and $P(R)$ be the probability of finding a bijection using rotation fitting. Let $P(B)$ denote the probability of finding a bijection using RecSplit’s brute force strategy. Then, $P(R) \rightarrow mP(B)$ for $m \rightarrow \infty$. 

![Graph showing expected factor and higher probability](image)

![Graph showing expected space overhead](image)
Proof (Sketch)

- consider number of different injective functions under cyclic shifts
- bit vector of length $m$ with $B$ set bits
- total number of equivalence classes under rotation is $\frac{1}{m} \sum_{d \text{ divides } \gcd(A, B)} \phi(d) \left( \frac{m/d}{B/d} \right)$
- probability of the event $I$ that there is a rotation has the $m$ least significant bits set is

$$P(I) \geq \frac{1}{\sum_{d \text{ divides } \gcd(A, B)} \phi(d) \left( \frac{m/d}{B/d} \right)}$$

- $\phi(i) = |\{j \leq i : \gcd(i, j) = 1\}|$ is Euler’s totient function

Proof (Sketch, cnt.)

- determine the probability $P(R)$ using the events
  - $A$: popcount(a) = $A$
  - $B$: popcount(b) = $B$
  - $B$: found bijection using brute-force
Proof (Sketch, ctn.)

\[
P(R) = P(A)P(B)P(I) \\
\geq \frac{m!}{(m - A)! m^A} \cdot \frac{m!}{(m - B)! m^B} \cdot P(I) = \frac{m!}{m^m} \cdot \frac{m!}{A!B!} \cdot P(I) = P(B) \cdot \frac{m!}{A!B!} \cdot P(I) \\
\geq P(B) \cdot \frac{m!}{A!B!} \cdot \frac{1}{\sum_{d \mid \gcd(A,B)} (\phi(d) (m/d) (A/B))} = P(B) \cdot \frac{m!}{m! + (A!B!) \sum_{d \mid \gcd(A,B), d \neq 1} (\phi(d) (m/d))} \\
= P(B) \cdot \frac{1}{1 + \sum_{d \mid \gcd(A,B), d \neq 1} (\phi(d) (m/d) (A!/B!))} \\
\sim P(B) \cdot \frac{1}{1 + \sum_{d \mid \gcd(A,B), d \neq 1} (\phi(d) (\sqrt{d} \frac{A - A/dB - B/d}{m^m - m/d}))} \\
\rightarrow P(B) \cdot m \text{ for } m \rightarrow \infty
\]
Parallel RecSplit on the GPU

Computing on the GPU

- several streaming multiprocessors (SMs)
- each SM contains many arithmetic logic units (ALUs)
- several threads operate in lock-step (warp)
- to hide latencies, each SM is oversubscribed with more threads than ALUs
- in CUDA, kernels are functions that can be executed on the GPU
- a kernel is executed on a grid of thread blocks

- use GPU to determine splitting and bijections
Experimental Evaluation

- Intel i7 11700 processor with 8 cores (16 hardware threads (HT)), base clock: 2.5 GHz
- AVX-512.
- Ubuntu 22.04 with Linux 5.15.0
- NVIDIA RTX 3090 GPU

- AMD EPYC 7702P processor with 64 cores (128 hardware threads), base clock: 2.0 GHz
- AVX2
- Ubuntu 20.04 with Linux 5.4.0

- GNU C++ compiler v.11.2.0 (-03 -march=native)
Rotation Fitting

![Graph showing Comparison between Brute force and Rotation fitting](image)

- Brute force
- Rotation fitting

Objects/second vs. Bits per object

Speedup vs. Bits per object

Institute of Theoretical Informatics, Algorithm Engineering
## Overview Results

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Method</th>
<th>Bijections</th>
<th>Threads</th>
<th>B/Obj</th>
<th>Constr.</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell = 16, b = 2000$</td>
<td>RecSplit [EGV20b]</td>
<td>Brute force</td>
<td>1</td>
<td>1.560</td>
<td>1175.4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RecSplit</td>
<td>Brute force</td>
<td>16</td>
<td>1.560</td>
<td>206.5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>1</td>
<td>1.560</td>
<td>138.0</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>16</td>
<td>1.560</td>
<td>27.9</td>
<td>42</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.560</td>
<td>1.8</td>
<td>655</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.560</td>
<td>1.0</td>
<td>1173</td>
</tr>
<tr>
<td>$\ell = 18, b = 50$</td>
<td>RecSplit [EGV20b]</td>
<td>Brute force</td>
<td>1</td>
<td>1.707</td>
<td>2942.9</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>RecSplit</td>
<td>Brute force</td>
<td>16</td>
<td>1.713</td>
<td>504.0</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>1</td>
<td>1.709</td>
<td>58.3</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>SIMDRecSplit</td>
<td>Rotation fitting</td>
<td>16</td>
<td>1.708</td>
<td>12.3</td>
<td>239</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.708</td>
<td>5.2</td>
<td>564</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.709</td>
<td>0.5</td>
<td>5438</td>
</tr>
<tr>
<td>$\ell = 24, b = 2000$</td>
<td>GPURecSplit</td>
<td>Brute force</td>
<td>GPU</td>
<td>1.496</td>
<td>2300.9</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>GPURecSplit</td>
<td>Rotation fitting</td>
<td>GPU</td>
<td>1.496</td>
<td>467.9</td>
<td>—</td>
</tr>
</tbody>
</table>
Comparison with Competitors

Comparison of different data structures on various hardware configurations:

- **8-core Intel**
  - 1 Thread
  - 16 Threads

- **64-core AMD**
  - 1 Thread
  - 128 Threads

**Throughput (MObjects/s) vs. Bits/Object**

Data structures considered:

- BBHash [Lim+17]
- CHD [BBD09b]
- PTHash [PT21b]
- PTHash-HEM [PT21a]
- RecSplit [EGV20b]
- SIMDRecSplit
- SicHash [LSW23]
Conclusion and Outlook

This Lecture
- conclusion retroactive data structures
- minimal perfect hash functions

Next Lecture (15.07.2024)
- NO LECTURE ON 08.07.2024
- learned data structures

Oral Exams and Project
- registration exams and project will open this week
Bibliography I


Bibliography II


Bibliography III

