Predecessor Data Structures

Philip Bille
Outline

- Predecessor problem
- First tradeoffs
- Simple tries
- x-fast tries
- y-fast tries
Predecessor Problem
The predecessor problem: Maintain a set $S \subseteq U = \{0, \ldots, u-1\}$ under operations

- predecessor($x$): return the largest element in $S$ that is $\leq x$.
- successor($x$): return the smallest element in $S$ that is $\geq x$.
- insert($x$): set $S = S \cup \{x\}$
- delete($x$): set $S = S - \{x\}$

We often ignore successor since we can get it from predecessor.
Applications

• What applications do we know?

• Simplest version of *nearest neighbor* problem.

• Several applications in other algorithms and data structures.

• Probably most practically solved problem in the world: Out all computational resources globally most are used used to solve the predecessor problem!
Routing IP-Packets

• Where should we forward the packet to?
• To address matching the longest prefix of 192.110.144.123.
• Equivalent to predecessor problem.
• Best practical solutions based on advanced predecessor data structures [Degermark, Brodnik, Carlsson, Pink 1997]
Solutions to the Predecessor Problem

• Which solutions do we know?
  • Linked list
  • Balanced binary search trees.
  • Bitvectors
First Tradeoffs
First Tradeoffs

• Assumption: We use space $O(u)$.

• What tradeoffs can we get for predecessor queries vs. updates (inserts and deletes)?
Fast Predecessor Queries

• How fast predecessor queries can we get if we ignore time for updates?
• Hint: Remember you can use $O(u)$ space.
Fast Predecessor Queries

• Solution: Precompute and store all predecessor queries.
• Example: $S = \{0, 2, 8, 11, 14\}$.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
</tbody>
</table>

• $O(1)$ time for predecessor.
• $O(u)$ time for updates.
Fast Updates

• How fast updates can we get if we ignore time for predecessor?

• Hint: Remember you can use $O(u)$ space.
Fast Updates

• Solution: A bitvector

• Example: $S = \{0,2,8,11,14\}$.

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
\]

• $O(u)$ time for predecessor.

• $O(1)$ time for updates.
Fast Updates and not so Slow Predecessors

• Can we improve the bitvector solution to get $O(1)$ updates and $o(u)$ predecessor?

• Hint: Try to come of with a two-level solution!
Fast Updates and not so Slow Predecessors

• Solution: Partition bitvector into \( u^{1/2} \) subvectors of length \( u^{1/2} \).
  • Maintain a summary bitvector of length \( u^{1/2} \) where index \( i \) is 1 iff subvector \( i \) contains a 1.

• Example: \( S = \{0, 2, 8, 11, 14\} \).

\[
\begin{array}{cccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15
\end{array}
\]

• \text{predecessor}(x): Search for predecessor of \( x \) in subvector for \( x \), if not found, find predecessor in summary and then in corresponding subvector. \( O(u^{1/2}) \) time.

• \( O(1) \) for updates (\( \leq 2 \) bitvectors to update).
van Emde Boas Data Structure [van Emde Boas 1975]

- Extend two level idea recursively to $O(\log \log u)$ levels. Combine with some additional tricks.
- => Predecessor data structure with $O(\log \log u)$ time per operation and $O(u)$ space. ($O(n)$ space with hashing)
- Intuition: $T(u) = T(u^{1/2}) + O(1) = O(\log \log u)$
- We show simpler but (essentially) equivalent solution.
Overview

• Simplifying assumption: We ignore updates (inserts and deletes)

• Goal: Achieve $O(\log \log u)$ time queries with $O(n)$ space.

• Solution in 3 steps:
  • Simple trie: Slow and too much space.
  • x-fast trie: Fast but too much space.
  • y-fast trie: Fast and little space.
Simple Trie
Tries

- Trie of $S = \text{tree of prefixes of binary representation of values in } S \text{ (the dark tree above)}$

- Depth is $\log u = \text{number of bits for value in } U$.

- Number of nodes = $O(n \log u)$.

$S = \{0, 2, 8, 11, 14\} = \{0000_2, 0010_2, 1000_2, 1011_2, 1110_2\}$
• Predecessor using tries?

• For each node in trie store pointer to nearest leaf to the left.

• predecessor(x): Find longest prefix of x in trie. Report nearest leaf to the left as predecessor.
Simple trie solution:

- Store trie as binary tree.

- predecessor(x): search top-down in trie to find longest prefix matching x. Report nearest left leaf.

- => $O(\log u)$ for predecessor query and $O(n \log u)$ space.
X-Fast Tries
X-Fast Tries: Data Structure

- Store the set of prefixes for each depth in (perfect) hash table:
  - $d_1 = \{0,1\}$, $d_2 = \{00, 10, 11\}$, $d_3 = \{000, 001, 100, 101, 111\}$, $d_4 = S$
  - $S = \{0, 2, 8, 11, 14\} = \{0000_2, 0010_2, 1000_2, 1011_2, 1110_2\}$

=> $O(n \log u)$ space.
X-Fast Tries: Searching

To find longest prefix of $x$ do a *binary search* over the depth.

Example $x = 9 = 1001_2$:

- Lookup $10_2$ in $d_2$ => exists so we continue in bottom 1/2 of tree.
- Lookup $100_2$ in $d_3$ => exists so we continue in bottom 1/4 of tree.
- Lookup $1001_2$ in $d_4$ => does not exists => $100_2$ is longest prefix.
X-Fast Tries: Searching

Lookup most significant half of x in hash table for depth $\log u/2$ and recurse on top or bottom.

- Binary search of depth of $\log u$.
  - Each step takes $O(1)$ time (lookup in perfect hash table for depth)
  - $\Rightarrow O(\log \log u)$ time for predecessor.
Summary

• Theorem: We can solve the *static* predecessor problem in
  • $O(n \log u)$ space
  • $O(\log \log u)$ time for predecessor.

• How do we get linear space?
Y-Fast Tries
Y-Fast Tries: The Data Structure

- Partition $S$ into $O(n / \log u)$ groups of $\log u$ consecutive values.
- Compute $S' = \text{set of split values between groups. } (|S'| = O(n / \log u).)$
- Data structure: $x$-fast trie over $S'$ + balanced search trees for each group.
- Space:
  - $x$-fast trie $O(|S'| \log u) = O(n / \log u \cdot \log u) = O(n)$
  - balanced search trees: $O(n)$ total.
Y-Fast Tries: Searching

- predecessor(x):
  - Find predecessor s of x in S’.
  - Find predecessor of x in group left or right of s.
- Time:
  - x-fast trie: $O(\log \log u)$
  - balanced search tree: $O(\log |G|) = O(\log \log u)$.
Y-Fast Tries

• Theorem: We can solve the static predecessor problem in
  • O(n) space
  • O(log log u) time for predecessor.

• What about updates?

• Theorem: We can solve the dynamic predecessor problem in
  • O(n) space
  • O(log log u) expected time for predecessor and updates.

From dynamic hashing
Summary

• Predecessor problem
• First tradeoffs
• Simple tries
• x-fast tries
• y-fast tries
References

- P. van Emde Boas, Preserving Order in a Forest in less than Logarithmic Time, FOCS, 1975
- Dan E. Willard, Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(n)$, Inf. Process. Lett., 1983
- Mihai Patrascu, Mikkel Thorup, Time-space trade-offs for predecessor search. STOC 2006
- Scribe notes from MIT