Lecture 9

Models of Computation

Cell-probe Model:

- Memory cells have size \( w \)
- We can read/write in cells for free at cost of 1
- Any computation is free
- Not practical but good model to prove lower bounds

Transdichotomous RAM:

- Memory cells have size \( w \) (and we assume \( w \geq \log n \))
- Each operation can modify only \( O(1) \) cells
- Cells can be addressed arbitrarily (RAM)
Variant 1 (Word RAM): O(1) operations:
+ - * / \% \& ^ © \_ <=
(C-style)

Variant 2 (AC⁰-RAM): O(1) operations:
have to be realized by an AC⁰ circuit
(constant depth, poly. size w/ w)
No multiplications!
(Rest of word-EM ops are fine)

Pointer Machine Model: data structure is modeled as directed graph with bounded degree

BST Model: graph is a binary search tree
Fixed Universe assumption: universe is the set of integers from 0,...,(n-1)

Successor/Predecessor in Fixed Universe

Overview (Classic results)

<table>
<thead>
<tr>
<th>DS</th>
<th>time/op</th>
<th>Space</th>
<th>model</th>
</tr>
</thead>
<tbody>
<tr>
<td>balanced trees</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>BST</td>
</tr>
<tr>
<td>van Emde Boas Trees</td>
<td>O(log log n)</td>
<td>O(n)</td>
<td>Word/AC0 PAM</td>
</tr>
<tr>
<td>y-Fast Trees</td>
<td>O(log n) w.h.p.</td>
<td>O(n)</td>
<td>Word RAM</td>
</tr>
<tr>
<td>Fusion Trees</td>
<td>O(log n / log log n)</td>
<td>O(n)</td>
<td>Word RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>A0 PAM</td>
</tr>
</tbody>
</table>

Van Emde Boas Trees

• represent the set S ⊆ U as bit vector

\[ n = 2^{a_1} 3^{a_2} \cdots 7^{a_7} \]

\[ S = 2, 3, 5, 6, 7, 3 \]

=)

The operations insert, find, delete, succ, pred should all need only \( O(\log \log n) \) time.
Idea: Use recursion
\[ S = \begin{array} {c c c c c c} \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array} \]

\[ \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \quad \rightarrow \]

bit vector
shifted in full blocks

We store the information if the blocks are empty
in an auxiliary structure
\[ \text{Summary}(S) \]

\[ \text{Summary} = \text{bit vector of size } \prod_{i} \]

1st approach: \[ \text{successor}(S, x) \]

\[ k = \text{successor}([\text{high}(x), \text{low}(x)]) \]
\[ \text{if } k < \infty \text{ report } k + \text{high}(x) \cdot \prod_{i} \]
\[ \text{else } z = \text{successor}([\text{summary}[\text{high}(x)]) \]
\[ \text{if } z = \infty \text{ return } \infty \]
\[ \text{else } y = \text{successor}([\text{high}(x), -\infty) \]
\[ \text{return } z \cdot \prod_{i} + y \]

\[ \rightarrow \text{ needs } \quad T(n) = 3 \cdot T(\sqrt{n}) + O(1) \quad \text{time} \]

\[ \text{too much! we have to get rid of the } \sqrt{n}. \]
2nd Approach: Store for every substructure $S$ also $\min S$, $\max S$

```
(Successor ($S$, $x$))
  if low$x < \max [\text{Sub}[S], \text{high}$x$]$
    then $j = \text{Successor}(\text{Sub}[S], \text{high}$x$)$
    return high$x$. $\text{Sub}[S] + j$
  else $k = \text{Successor}(\text{Sub}[S], \text{high}$x$)$
    return $\min [\text{Sub}[S], k] + k. \text{TS}$
```

Notice we have to update $\min/\max S$ while

```
\text{INSERT} (S, x)
  if $\min(S) = \emptyset$ then $\min(S) = x$
  if $x < \min(S)$ swap $x, \min(S)$
  if $\min [\text{Sub}[S], \text{high}$x$] = \emptyset$
    then $\min [\text{Sub}[S], \text{high}$x$] = \text{low}$x$
      $\text{INSERT}(\text{Sub}[S], \text{high}$x$)$
    else $\text{INSERT}(\text{Sub}[S], \text{low}$x$)$
  if $x > \text{MAX}[S]$ then $\text{MART}[S] = x$
```

Van Emde Boas Tree need $O(n)$ space!
X-fast Trees

Successor/query: (1) go up until you read "1" predecessor
(2) go down as close as possible

- This gives predecessor or successor, list all leaves to allow to answer all queries
- Store in every node min/max to speed up (2)
- The first "1" we find is the longest prefix for binary (x) in 2 binary (y) if yes?
- Store all prefixes in HasTable
- (perfect, dynamic) O(1)/O(n) w.h.p.
We have stored \( n \) "Paths" of length \( O(\log n) \) in \( O(n \cdot \log n) \) space (too much!)

Accessing Successor / Predecessor

Binary search over prefix length \( O(\log \log n) \)

Updates: we might add \( O(\log n) \) prefixes

\( \Rightarrow O(\log n) \) (too much!)
Updates

• predecessor queues

Algorithms

• store the representatives in an X-fast-tree

all space

• space X-fast-tree = O(1)

Pt1. Let S1 = O(\log n)

• every representative has a pointer to its BST

Pt2. \log n = O(\log \log n)

• update only in Ri (merging Ri)

Pt3. \log n = O(\log \log n)

• move Ri's

Pt4. Split Ri

Pt5. O(\log \log n)

• update x & t

Pt6. O(\log \log n)

• update x & t
\[ |\text{binary}(x)| = \log n \]

- search for the longest prefix (binary search) \[ \log \log n \]

- drawbacks: insert/delete: \( O(\log n) \)
space: \( O(n \log n) \)

**Improvement** \( y \)-fast trees [Willard '85]

**Idea:** we group small subsets of our input together and store them in an additional DS (indirection)

- cluster size: \( \Theta(\log n) \)

\[ S_1, S_2, \ldots, S_k = \text{clusters} \]

- each cluster \( S_i \) is stored in a balanced BST

\( \text{e.g. (2,4) - trees} \) \( R_i \)

- for each cluster \( S_i \) we have a representative \( R_i : \forall S_i \subseteq S_{i+1} \)

\[ S_i \geq R_i > S_{i+1} \]