Van Emde Boas Data Structure

This lecture

- Insert \( O(\log n) \) time
- Min \( O(1) \) time
- Delete \( O(\log \log n) \) time

Assuming all elements are in \( \{1, \ldots, n\} \).

Motivation: Prim / Dijkstra when weights on edges are in \( \{1, \ldots, m\} \).

\( m \) inserts \( \Rightarrow O(n \log \log n) \) time
\( n = \text{min} + \text{delete} \)

(While Kruskal takes \( O(m \cdot \log n) \); see next lecture)
The idea of the data structure is demonstrated nicely by the following problem:

We want to find what's the height (±1 inch) such that if we drop a coconut from this height, it will crack.

When we have one coconut—try 1 in, 2 in, 3 in, ...

What if we have two coconuts? (When one cracks we can continue with the other)

Idea: Divide into blocks of height bin each.

- First coconut will be dropped from 2bin, 3bin, 4bin, ...
  Suppose it breaks at kbin in.
- Second coconut will be dropped from (k-1)bin, (k-1)bin+1, ...

\[ \# \text{drops} \leq \frac{\sqrt{n}}{b} + b \] minimized for \( b = \sqrt{n} \).
van Emde Boas

Idea #1: Represent the set using bit array

\[ A = \begin{array}{c|c|c|c|c} \text{bit} & 1 & 2 & \ldots & n \\ \hline & 0 & 1 & \ldots & 0 \end{array} \]

\[ A[i] = \begin{cases} 1 & \text{i in set} \\ 0 & \text{o/w.} \end{cases} \]

2. Store the min in a separate var.

Insert - \( O(\log n) \) time.

Min - \( O(1) \) time.

Delete - if we delete the min, have to find new min - this can take \( O(n) \) time.
Cocont Idea: Divide n into blocks of size b.

Have another array that stores for each block whether empty.

Insert - \(\mathcal{O}(1)\) time (same as before except may need to update B)

Min - \(\mathcal{O}(1)\) (same as before)

Delete - \(\frac{n}{b} + \mathcal{O}(\log b)\) (\(b = \sqrt{n}\))

For Delete: (Not as good as promised at this point)
Implementation Detail (that will turn out to be crucial)

- Keep the min and max of B separately, don't insert them to B.
- Keep per block the min and max of the block separately, don't insert them to the block.

global min = min of min non-empty block.
global max = max of max non-empty block

$\mathcal{O}(1)$ computation
- Insert -

(1) Either insert to empty block, 
    so need to insert to B, but not to A, 
    only change block's min & max.

(2) Or insert to non-empty block, 
    so don't touch B, just update A; 
    may need to change min/max, may need to insert.

---

- Delete -

(1) Either block will become empty, \( \text{when} \ \text{min-block} = \text{max-block} \), 
    so need to delete from B, but not from A, 
    only change block's min/max.

(2) Or delete from block that won't become empty, 
    so don't touch B, just update A; 
    may need to change min/max, will always need to delete from the block.
Can we get better running time for Delete?

Observations

* Per block in A, have data-structure that needs (only) to support insert, delete, min, max. \( \rightarrow \sqrt{n} \) structures on \( \sqrt{n} \) elements each

* Per B, have data structure that \( \cdots \rightarrow \) one structure on \( \sqrt{n} \) elements.

\[ T_{\text{insert}}(n) = T_{\text{insert}}(\sqrt{n'}) + O(1) \]

\[ T_{\text{delete}}(n) = T_{\text{delete}}(\sqrt{n'}) + O(1) \]

leading constant \( = 1 \) thanks to implementation detail

either delete from B, or from A's block

either insert to B, or to A's block
Remains to solve the recurrence

\[ T(n) = T(\sqrt{n}) + O(1) \]

Idea Substitute \( m = \log n \)

\[ T(\sqrt{m}) = T(2^{m/2}) + O(1) \]

\[ T'(m) = T'(m/2) + O(1) = O(\log m) \]

\[ \Rightarrow T(n) = T'(\log n) = O(\log \log n) \]