van Emde Boas Trees -
Dynamic Predecessors

Uri Zwick
Tel Aviv University
Dictionaries

Hash Tables

\( O(1) \) time

Insert
Delete
Find
Successor / Predecessor
Find-min / Find-max

Binary Search Trees

\( O(\log n) \) time
vEB Trees
\( O(\log w) \) time

\( w \) – word size

Typically \( \log w < \log n \)
Splitting a word

\[ w \text{ bits} \]

\[ \begin{array}{c}
\text{high} \\
\text{low}
\end{array} \]

\[ \begin{array}{c}
w/2 \text{ bits} \\
w/2 \text{ bits}
\end{array} \]

\[ \text{high} = x >> (w/2) \]

\[ \text{low} = x \& ((1 << (w/2)) - 1) \]

\[ O(1) \text{ time} \]
Definitions

\( D \) – the set of keys in the dictionary

\[
Succ(D, x) = \min\{ y \in D \mid y > x \}
\]

\[
High(D) = \{ high(x) \mid x \in D \}
\]

\[
Low(D, a) = \{ low(x) \mid x \in D \land high(x) = a \}
\]
van Emde Boas Trees

A $vEB$ tree representing $D$ consists of:

$w$ – word size

$\text{min}$ – the minimal key in $D$

$max$ – the maximal key in $D$

High – a $vEB$ tree for $\text{High}(D - \{\text{min}, \text{max}\})$

Low – a hash table, giving for every $a \in \text{High}(D - \{\text{min}, \text{max}\})$, a pointer to a $vEB$ tree for $\text{Low}(D - \{\text{min}, \text{max}\}, a)$
van Emde Boas Trees

Are *Top, Bot* better names than *High, Low*?
van Emde Boas Trees

If $D = \emptyset$ then $min = 1$ and $max = 0$

If $D = \{x\}$ then $min = x$ and $max = x$

If $|D| \geq 2$ then $min = \min D < max = \max D$

(Can add a size field)

If $|D| \leq 2$ then $High = null$ and $Low = \emptyset$

If $|D| > 2$ then $High \neq null$ and $Low \neq \emptyset$

Keeping at least one of $min$ and $max$ is essential for the efficiency of the data structure
van Emde Boas Trees

For compactness, we use array inspired notation for manipulating hash tables.

\[
\begin{align*}
\text{Low}[a] & \quad \rightarrow \quad \text{Low}.\text{HashFind}(a) \\
\text{Low}[a] & \leftarrow D' \quad \rightarrow \quad \text{Low}.\text{HashInsert}(a, D') \\
\text{Low}[a] & \leftarrow \text{null} \quad \rightarrow \quad \text{Low}.\text{HashDelete}(a)
\end{align*}
\]

van Emde Boas actually used arrays and not hash tables in his original data structure.

Hash tables introduced later by [Mehlhorn-Näher (1990)]
$$D. Insert(x)$$

The boring part:
If $min > max \ (D = \emptyset)$ then $min, max \leftarrow x$ ; return
If $x = min$ or $x = max$ then return
If $x < min = max$ then $min \leftarrow x$ ; return
If $min = max < x$ then $max \leftarrow x$ ; return

Of some interest:
If $x < min$ then $x \leftrightarrow min$
If $x > max$ then $x \leftrightarrow max$

The interesting part:
Insert $x$ given that $min < x < max$
**D. Insert**(\(x\))

Insert \(x\) given that \(\min < x < \max\)

Split \(x\) into \((\text{high, low})\)

If \(D\) already contains a “middle” item beginning with \(\text{high}\), recursively insert \(\text{low}\) into \(\text{Low}[\text{high}]\)

Otherwise, allocate a new \(vEB\) tree containing \(w/2\)-bit keys and insert \(\text{low}\) into it.

And, recursively insert \(\text{high}\) to the \(vEB\) tree \(\text{High}\).

(Initialize \(\text{High}\) if not initialized yet.)

**Important:** Only one \(recursive\) call at each level
D. Insert(x)

Insert x given that $\text{min} < x < \text{max}$

$$(\text{high}, \text{low}) \leftarrow \text{Split}(x, w)$$

if $\text{Low}[\text{high}] \neq \text{null}$ then:
   $\text{Low}[\text{high}].\text{Insert}(\text{low})$
else:
   $\text{Low}[\text{high}] \leftarrow \text{vEB}(w/2, \text{low})$

if $\text{High} = \text{null}$ then:
   $\text{High} \leftarrow \text{vEB}(w/2, \text{high})$
else:
   $\text{High}.\text{Insert}(\text{high})$
**D. Delete**($x$)

The boring part:

If $min > max$ ($D$ is empty) then return

If $x < min$ or $x > max$ then return

If $x = min = max$ and then

$min \leftarrow 1$ ; $max \leftarrow 0$ ; return

The interesting parts:

Delete $x$ given that $x = min < max$

Delete $x$ given that $min < max = x$

Delete $x$ given that $min < x < max$
**D. Delete** \((x)\)

Delete \(x\) given that \(x = \text{min} < \text{max}\)

If \(|D| = 2\) (\(\text{High}\) is null) then

\[
\text{min} \leftarrow \text{max} \quad \text{return}
\]

Otherwise, find and set the new min of \(D\):

- \textit{high} of new min is \(\text{High}.\text{min}\)
- \textit{low} of new min is \(\text{Low}[\text{high}].\text{min}\)

Delete \((\text{high}, \text{low})\) from \(D\)

The case \(\text{min} < \text{max} = x\) is analogous
D. Delete(x)

Delete \( x \) given that \( x = \text{min} < \text{max} \)

if \( \text{High} = \text{null} \) then:
  \[ \text{min} \leftarrow \text{max} \]
  return

else:
  # Set the new minimum
  \[ \text{high} \leftarrow \text{High} . \text{min} \]
  \[ \text{low} \leftarrow \text{Low}[\text{high}] . \text{min} \]
  \[ \text{min} \leftarrow \text{Combine}(\text{high}, \text{low}) \]

Delete \((\text{high}, \text{low})\) from \( D \)
D. Delete($x$)
Delete $x$ given $\text{min} < x < \text{max}$

Split $x$ into $(\text{high}, \text{low})$.

Recursively delete $\text{low}$ from $\text{Low}[\text{high}]$.

If $\text{Low}[\text{high}]$ is now empty then:
   - Destroy $\text{Low}[\text{high}]$.
   - Delete $\text{high}$ from $\text{Low}$.

Recursively delete $\text{high}$ from $\text{High}$.
    - If $\text{High}$ is empty, destroy it.

Important: Only one recursive call at each level is a “real” non-trivial recursive call.
D. Successor(x)

If \( x < \text{min} \) then return \( \text{min} \)
If \( x \geq \text{max} \) then return 0 (no successor)

Split \( x \) into \( (\text{high}, \text{low}) \)

If \( \text{high} \in \text{Low} \) and \( \text{low} < \text{Low}[\text{high}].\text{max} \) then recursively compute the successor \( \text{low1} \) of \( \text{low} \) in \( \text{Low}[\text{high}] \) and return \( (\text{high}, \text{low1}) \)

If \( \text{High} \neq \text{null} \) then recursively compute the successor \( \text{high1} \) of \( \text{high} \) in \( \text{High} \).

If \( \text{high1} \) exists, return \( (\text{high1}, \text{Low}[\text{high1}].\text{min}) \)
If \( \text{High} = \text{null} \) or \( \text{high1} \) does not exist, return \( \text{max} \)
D. Successor($x$)

if $\text{Low}[\text{high}] \neq \text{null}$ and $\text{low} < \text{Low}[\text{high}].\text{max}$ then:
    return $\text{Combine}(\text{high}, \text{Low}[\text{high}].\text{Successor}(\text{low}))$
else:
    if $\text{High} = \text{null}$ then:
        return \text{max}
    else:
        $\text{high1} \leftarrow \text{High}.\text{Successor}(\text{high})$
        if $\text{high1} = 0$ then:
            return \text{max}
        else:
            return $\text{Combine}(\text{high1}, \text{Low}[\text{high1}].\text{min})$
Time complexity

We assume that each hash table operation takes $O(1)$ time, which holds in expectation.

We sometimes need to double the size of the hash table, or cut it by a factor of 4. This can be taken care of by amortization or background rebuilding.

Each operation on a vEB tree with $w$-bit keys spends $O(1)$ time and then performs at most one non-trivial recursive call on a vEB tree with $(w/2)$-bit keys.
Time complexity

\[ T(w) = O(1) + T\left(\frac{w}{2}\right) \]

\[ T(1) = O(1) \]

\[ T(w) = O(\log w) \]
Exercise: (Trivial) Add a *Find* operation to a vEB tree. How much time does it take? Augment the data structure to support *Find* in $O(1)$ time.

Exercise: Design a variant of vEB trees that does not explicitly maintains the *min* and *max* items. What are the times required by the different operations?

Exercise: Design a variant of vEB trees that maintains *min* but not *max* items. Implement efficiently an operation that finds either the successor or the predecessor. Show how the data structure can be augmented to return successors and predecessors.
Space complexity

Where are items actually stored?

In \textit{min} and \textit{max} fields.

Total space used is proportional to number of vEB structures, which is proportional to total number of \textit{min}’s and \textit{max}’s.

A hash table containing \( k \) items has size \( O(k) \) and it points to \( k \) vEB structures. Thus, the size of the table can be charged to the \textit{min} and \textit{max} items in these \( k \) structures.

Important: We do not keep empty vEB structures.
Space complexity

To be continued…

On the blackboard.