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

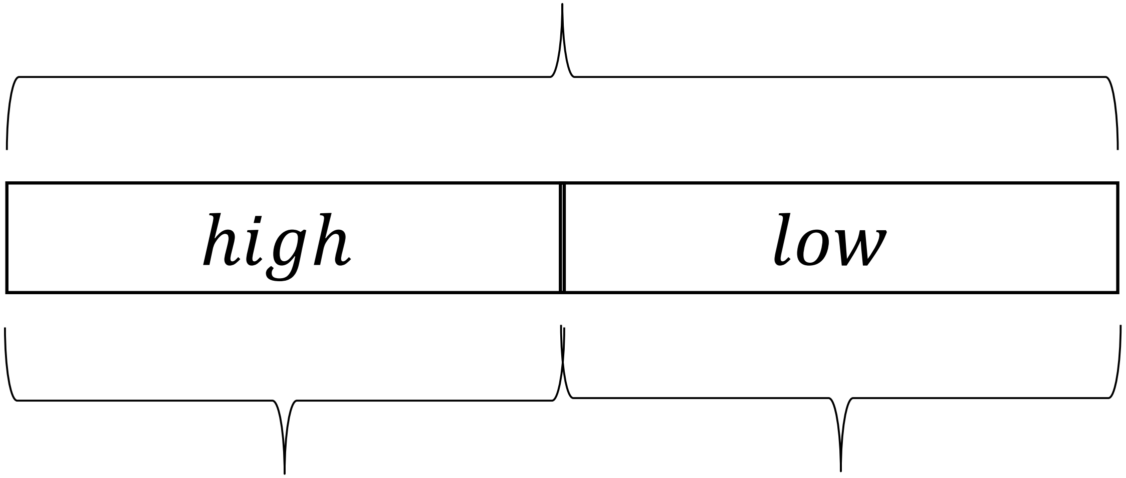
vEB Trees
$O(\log w)$
time

w – word size

Typically $\log w < \log n$

Splitting a word

w bits



$high$ | low

$w/2$ bits $\quad$ $w/2$ bits

$$high = x >> (w/2)$$

$$low = x \,\&\, ((1 << (w/2)) - 1)$$

$O(1)$ time

Definitions

D – the set of keys in the dictionary

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

$$High(D) = \{\, high(x) \mid x \in D \,\}$$

$$Low(D, a) = \{\, low(x) \mid x \in D \wedge high(x) = a \,\}$$

van Emde Boas Trees

A *vEB* tree representing D consists of:

w – word size

min – the minimal key in D

max – the maximal key in D

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

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

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.

$$Low[a] \quad \rightarrow \quad Low.HashFind(a)$$

$$Low[a] \leftarrow D' \quad \rightarrow \quad Low.HashInsert(a, D')$$

$$Low[a] \leftarrow null \quad \rightarrow \quad Low.HashDelete(a)$$

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 $(high, low)$

If D already contains a “middle” item beginning with $high$, *recursively* insert low into $Low[high]$

Otherwise, allocate a new *vEB* tree containing $w/2$-bit keys and insert low into it.

And, *recursively* insert $high$ to the *vEB* tree $High$.
(Initialize $High$ if not initialized yet.)

Important: Only one *recursive* call at each level

$D.Insert(x)$

Insert x given that $min < x < max$

```
$(high, low) \leftarrow Split(x, w)$
if $Low[high] \neq null$ then:
    $Low[high].Insert(low)$
else:
    $Low[high] \leftarrow vEB(w/2, low)$
    if $High = null$ then:
        $High \leftarrow vEB(w/2, high)$
    else:
        $High.Insert(high)$
```

$D.Delete(x)$

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 $x = min < max$

If $|D| = 2$ ($High$ is $null$) then

$min \leftarrow max$; return

Otherwise, find and set the new min of D:

$high$ of new min is $High.min$

low of new min is $Low[high].min$

Delete $(high, low)$ from D

The case $min < max = x$ is analogous

$D.Delete(x)$

Delete x given that $x = min < max$

```
if $High = null$ then:
    $min \leftarrow max$
    return
else:
    # Set the new minimum
    $high \leftarrow High.min$
    $low \quad \leftarrow Low[high].min$
    $min \leftarrow Combine(high, low)$

Delete $(high, low)$ from $D$
```

$D.Delete(x)$

Delete x given $min < x < max$

Split x into $(high, low)$.

Recursively delete low from $Low[high]$.

If $Low[high]$ is now empty then:
Destroy $Low[high]$.
Delete $high$ from Low.
Recursively delete $high$ from $High$.
If $High$ is empty, destroy it.

Important: Only one *recursive* call at each level is a “real” non-trivial *recursive* call.

$D.Successor(x)$

If $x < min$ then return min

If $x \geq max$ then return 0 (no successor)

Split x into $(high, low)$

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

If $High \neq null$ then *recursively* compute the successor $high1$ of $high$ in $High$.

If $high1$ exists, return $(high1, Low[high1].min)$

If $High = null$ or $high1$ does not exist, return max

$D.Successor(x)$

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

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

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

$$T(1) = O(1)$$

$$\Downarrow$$

$$T(w) = O(\log w)$$

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 min and max fields.

Total space used is proportional to number of vEB structures, which is proportional to total number of min’s and 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 min and max items in these k structures.

Important: We do not keep empty vEB structures.

Space complexity

To be continued…

On the blackboard.