

Lecture 4: Divide and Conquer: van Emde Boas Trees

- Series of Improved Data Structures
- Insert, Successor
- Delete
- Space

This lecture is based on personal communication with Michael Bender, 2001.

Goal

We want to maintain n elements in the range $\{0, 1, 2, \dots, u - 1\}$ and perform Insert, Delete and Successor operations in $\mathcal{O}(\log \log u)$ time.

- If $n = n^{\mathcal{O}(1)}$ or $n^{(\log n)^{\mathcal{O}(1)}}$, then we have $\mathcal{O}(\log \log n)$ time operations
 - Exponentially faster than Balanced Binary Search Trees
 - Cooler queries than hashing
- Application: Network Routing Tables
 - u = Range of IP Addresses \rightarrow port to send ($u = 2^{32}$ in IPv4)

Where might the $\mathcal{O}(\log \log u)$ bound arise ?

- Binary search over $\mathcal{O}(\log u)$ elements
- Recurrences
 - $T(\log u) = T\left(\frac{\log u}{2}\right) + \mathcal{O}(1)$
 - $T(u) = T(\sqrt{u}) + \mathcal{O}(1)$

Improvements

We will develop the van Emde Boas data structure by a series of improvements on a very simple data structure.

Bit Vector

We maintain a vector V of size u such that $V[x] = 1$ if and only if x is in the set. Now, inserts and deletes can be performed by just flipping the corresponding bit in the vector. However, successor/predecessor requires us to traverse through the vector to find the next 1-bit.

- Insert/Delete: $\mathcal{O}(1)$
- Successor/Predecessor: $\mathcal{O}(u)$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1

Figure 1: Bit vector for $u = 16$. The current set is $\{1, 9, 10, 15\}$.

Split Universe into Clusters

We can improve performance by splitting up the range $\{0, 1, 2, \dots, u - 1\}$ into \sqrt{u} clusters of size \sqrt{u} . If $x = i\sqrt{u} + j$, then $V[x] = V.Cluster[i][j]$.

$$low(x) = x \bmod \sqrt{u} = j$$

$$high(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$$

$$index(i, j) = i\sqrt{u} + j$$

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	1
$V.Cluster[0]$				$V.Cluster[1]$				$V.Cluster[2]$				$V.Cluster[3]$			

Figure 2: Bit vector ($u = 16$) split into $\sqrt{16} = 4$ clusters of size 4.

- Insert:
 - Set $V.cluster[high(x)][low(x)] = 1$ $\mathcal{O}(1)$

- Mark cluster $high(x)$ as non-empty $\mathcal{O}(1)$
- Successor:
 - Look within cluster $high(x)$ $\mathcal{O}(\sqrt{u})$
 - Else, find next non-empty cluster i $\mathcal{O}(\sqrt{u})$
 - Find minimum entry j in that cluster $\mathcal{O}(\sqrt{u})$
 - Return $index(i, j)$ Total = $\mathcal{O}(\sqrt{u})$

Recurse

The three operations in Successor are also Successor calls to vectors of size \sqrt{u} . We can use recursion to speed things up.

- $V.cluster[i]$ is a size- \sqrt{u} van Emde Boas structure ($\forall 0 \leq i < \sqrt{u}$)
- $V.summary$ is a size- \sqrt{u} van Emde Boas structure
- $V.summary[i]$ indicates whether $V.cluster[i]$ is nonempty

INSERT(V, x)

- 1 $Insert(V.cluster[high(x)], low[x])$
- 2 $Insert(V.summary, high[x])$

So, we get the recurrence:

$$\begin{aligned}
 T(u) &= 2T(\sqrt{u}) + \mathcal{O}(1) \\
 T'(\log u) &= 2T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \\
 \implies T(u) &= T'(\log u) = \mathcal{O}(\log u)
 \end{aligned}$$

SUCCESSOR(V, x)

- 1 $i = high(x)$
- 2 $j = Successor(V.cluster[i], j)$
- 3 **if** $j == \infty$
- 4 $i = Successor(V.summary, i)$
- 5 $j = Successor(V.cluster[i], -\infty)$
- 6 **return** $index(i, j)$

$$\begin{aligned}
T(u) &= 3T(\sqrt{u}) + \mathcal{O}(1) \\
T'(\log u) &= 3T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \\
\implies T(u) &= T'(\log u) = \mathcal{O}((\log u)^{\log 3}) \approx \mathcal{O}((\log u)^{1.585})
\end{aligned}$$

To obtain the $\mathcal{O}(\log \log u)$ running time, we need to reduce the number of recursions to one.

Maintain Min and Max

We store the minimum and maximum entry in each structure. This gives an $\mathcal{O}(1)$ time overhead for each *Insert* operation.

SUCCESSOR(V, x)

```

1   $i = \text{high}(x)$ 
2  if  $\text{low}(x) < V.\text{cluster}[i].\text{max}$ 
3       $j = \text{Successor}(V.\text{cluster}[i], \text{low}(x))$ 
4  else  $i = \text{Successor}(V.\text{summary}, \text{high}(x))$ 
5       $j = V.\text{cluster}[i].\text{min}$ 
6  return  $\text{index}(i, j)$ 

```

$$\begin{aligned}
T(u) &= T(\sqrt{u}) + \mathcal{O}(1) \\
\implies T(u) &= \mathcal{O}(\log \log u)
\end{aligned}$$

Don't store Min recursively

The *Successor* call now needs to check for the min separately.

if $x < V.\text{min}$: **return** $V.\text{min}$ (1)

INSERT(V, x)

```

1  if  $V.min == None$ 
2       $V.min = V.max = x$       ▷  $\mathcal{O}(1)$  time
3      return
4  if  $x < V.min$ 
5      swap( $x \leftrightarrow V.min$ )
6  if  $x > V.max$ 
7       $V.max = x$ 
8  if  $V.cluster[high(x)] == None$ 
9      Insert( $V.summary, high(x)$ )    ▷ First Call
10 Insert( $V.cluster[high(x)], low(x)$ )  ▷ Second Call

```

If the **first call** is executed, the **second call** only takes $\mathcal{O}(1)$ time. So

$$T(u) = T(\sqrt{u}) + \mathcal{O}(1)$$

$$\implies T(u) = \mathcal{O}(\log \log u)$$

DELETE(V, x)

```

1  if  $x == V.min$       ▷ Find new min
2       $i = V.summary.min$ 
3      if  $i = None$ 
4           $V.min = V.max = None$       ▷  $\mathcal{O}(1)$  time
5          return
6       $V.min = index(i, V.cluster[i].min)$     ▷ Unstore new min
7  Delete( $V.cluster[high(x)], low(x)$ )    ▷ First Call
8  if  $V.cluster[high(x)].min == None$ 
9      Delete( $V.summary, high(x)$ )    ▷ Second Call
10 ▷ Now we update  $V.max$ 
11 if  $x == V.max$ 
12 if  $V.summary.max = None$ 
13 else
14      $i = V.summary.max$ 
15      $V.max = index(i, V.cluster[i].max)$ 

```

If the **second call** is executed, the **first call** only takes $\mathcal{O}(1)$ time. So

$$T(u) = T(\sqrt{u}) + \mathcal{O}(1)$$

$$\implies T(u) = \mathcal{O}(\log \log u)$$

Lower Bound [Patrascu & Thorup 2007]

Even for static queries (no Insert/Delete)

- $\Omega(\log \log u)$ time per query for $u = n^{(\log n)^{\mathcal{O}(1)}}$
- $\mathcal{O}(n \cdot \text{poly}(\log n))$ space

Space Improvements

We can improve from $\Theta(u)$ to $\mathcal{O}(n \log \log u)$.

- Only create nonempty clusters
 - If $V.min$ becomes *None*, deallocate V
- Store $V.cluster$ as a hashtable of nonempty clusters
- Each insert may create a new structure $\Theta(\log \log u)$ times (each empty insert)
 - Can actually happen [Vladimir Čunát]
- Charge pointer to structure (and associated hash table entry) to the structure

This gives us $\mathcal{O}(n \log \log u)$ space (but randomized).

Indirection

We can further reduce to $\mathcal{O}(n)$ space.

- Store vEB structure with $n = \mathcal{O}(\log \log u)$ using BST or even an array
 - $\implies \mathcal{O}(\log \log n)$ time once in base case
- We use $\mathcal{O}(n / \log \log u)$ such structures (disjoint)
 - $\implies \mathcal{O}(\frac{n}{\log \log u} \cdot \log \log u) = \mathcal{O}(n)$ space for small
- Larger structures “store” pointers to them
 - $\implies \mathcal{O}(\frac{n}{\log \log u} \cdot \log \log u) = \mathcal{O}(n)$ space for large
- Details: Split/Merge small structures

MIT OpenCourseWare
<http://ocw.mit.edu>

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2015

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.