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 $\{0, 1, 2, \ldots, u - 1\}$ and perform Insert, Delete and Successor operations in $O(\log \log u)$ time.

- If $n = n^{O(1)}$ or $n^{(\log n)^{O(1)}}$, then we have $O(\log \log n)$ time operations
  - Exponentially faster than Balanced Binary Search Trees
  - Cooler queries than hashing

- Application: Network Routing Tables
  - $u = \text{Range of IP Addresses} \rightarrow \text{port to send}$
    $(u = 2^{32} \text{ in IPv4})$

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

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

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: $O(1)$
- Successor/Predecessor: $O(u)$

![Figure 1: Bit vector for $u = 16$. The current set is {1, 9, 10, 15}.](image)

Split Universe into Clusters

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

- $low(x) = x \mod \sqrt{u} = j$
- $high(x) = \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$
- $index(i,j) = i\sqrt{u} + j$

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

- Insert:
  - Set $V.cluster[high(x)][low(x)] = 1$ $O(1)$
- Mark cluster \( \text{high}(x) \) as non-empty \( \mathcal{O}(1) \)
  
- **Successor:**
  - Look within cluster \( \text{high}(x) \) \( \mathcal{O}(\sqrt{u}) \)
  - Else, find next non-empty cluster \( i \) \( \mathcal{O}(\sqrt{u}) \)
  - Find minimum entry \( j \) in that cluster \( \mathcal{O}(\sqrt{u}) \)
  - Return \( \text{index}(i, j) \) 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. \( \text{Insert}(V.cluster[\text{high}(x)], \text{low}[x]) \)
2. \( \text{Insert}(V.summary, \text{high}[x]) \)

So, we get the recurrence:

\[
T(u) = 2T(\sqrt{u}) + \mathcal{O}(1) \\
T'(\log u) = 2T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \\
\implies T(u) = T'(\log u) = \mathcal{O}(\log u)
\]

**SUCCESSOR**(\( V, x \))
1. \( i = \text{high}(x) \)
2. \( j = \text{Successor}(V.cluster[i], j) \)
3. **if** \( j = \infty \)
4. \( i = \text{Successor}(V.summary, i) \)
5. \( j = \text{Successor}(V.cluster[i], -\infty) \)
6. \( \text{return index}(i, j) \)
\[ T(u) = 3T(\sqrt{u}) + \mathcal{O}(1) \]
\[ T'(\log u) = 3T'\left(\frac{\log u}{2}\right) + \mathcal{O}(1) \]
\[ \implies T(u) = T'(\log u) = \mathcal{O}((\log u)^{\log 3}) \approx \mathcal{O}((\log u)^{1.585}) \]

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\((V, x)\)**

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

\[ T(u) = T(\sqrt{u}) + \mathcal{O}(1) \]
\[ \implies T(u) = \mathcal{O}(\log \log u) \]

**Don’t store Min recursively**

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

\[
\text{if } x < V.min : \text{return } V.min \quad \quad (1)
\]
**INSERT**(\(V, x\))

1. if \(V.min == None\)  
2. \(V.min = V.max = x\)  \(\triangleright \mathcal{O}(1)\) 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 \(V.cluster[\text{high}(x)] == None\)  
9. \(\text{Insert}(V.summary, \text{high}(x))\)  \(\triangleright \text{First Call}\)  
10. \(\text{Insert}(V.cluster[\text{high}(x)], \text{low}(x))\)  \(\triangleright \text{Second Call}\)

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

\[
T(u) = T(\sqrt{u}) + \mathcal{O}(1) \\
\Rightarrow T(u) = \mathcal{O}(\log \log u)
\]

**DELETE**(\(V, x\))

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

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

\[
T(u) = T(\sqrt{u}) + \mathcal{O}(1) \\
\Rightarrow T(u) = \mathcal{O}(\log \log u)
\]
Lower Bound [Patrascu & Thorup 2007]

Even for static queries (no Insert/Delete)

• $\Omega(\log\log u)$ time per query for $u = n^{(\log n)^{O(1)}}$

• $O(n \cdot \text{poly}(\log n))$ space

Space Improvements

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

• Only create nonempty clusters
  
  – If $V\text{.min}$ becomes None, deallocate $V$

• Store $V\text{.cluster}$ 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 $O(n \log \log u)$ space (but randomized).

Indirection

We can further reduce to $O(n)$ space.

• Store vEB structure with $n = O(\log \log u)$ using BST or even an array

  $\implies O(\log \log n)$ time once in base case

• We use $O(n/\log \log u)$ such structures (disjoint)

  $\implies O\left(\frac{n}{\log \log u} \cdot \log \log u\right) = O(n)$ space for small

• Larger structures “store” pointers to them

  $\implies O\left(\frac{n}{\log \log u} \cdot \log \log u\right) = O(n)$ space for large

• Details: Split/Merge small structures
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.