CS 224: Advanced Algorithms Spring 2026

Lecture Lecture 2: Van Emde Boas Tree — 01/12/2025

Prof. Jelani Nelson *Scribe: Woodward Galbriath*

1 Overview

In the last lecture we talked about logistics and how to do well in the course. In this lecture we defined the Van Emde Boas Tree, a efficient tree based data structure for storing integers. We went through several increasingly efficient version before ultimately arriving at one for which we were able to achieve $\mathcal{O}(log(log(U))$ time complexity for insertion deletion and successor search.

2 Motivation

Goal: Our goal is develop a data structure which can maintain n elements among $\{0, 1 \cdots U - 1\}$ subject to the insertion, deletion and successor search operations.

2.1 Operations

We would like the data structure to have the following operations

1. **Insertion:** Adding an item to the tree.
2. **Deletion:** removing an item from the tree
3. **Successor query:** Searching the tree for element $x \in U$, and if x is not in the tree return the smallest element of the tree greater than x

We would like our data structure to have time complexity $\mathcal{O}(log(log(U))$ for all the above operations and space complexity $\mathcal{O}(U)$

2.2 Naive data structures

Some data structures we already have are shown bellow, we can see that they are much less time efficient our goal.

2.2.1 Sorted array

If we store our n elements in a sorted array we will have

- **Insertion time:** $\mathcal{O}(n)$ in the worst case we need to move all elements.

- **Deletion time:** $\mathcal{O}(n)$ in the worst case we need to move all elements.
- **Successor time:** $\mathcal{O}(log(n))$, can binary search
- **Space complexity:** $\mathcal{O}(n)$, we only need to hold n elements

2.2.2 Balanced binary search tree

Could try doing a balanced binary search tree like a red-black tree.

- **Insertion time:** $\mathcal{O}(log(n))$, may need to edit $log(n)$ branches.
- **Deletion time:** $\mathcal{O}(log(n))$, may need to edit $log(n)$ branches.
- **Successor time:** $\mathcal{O}(log(n))$, may need to check $log(n)$ branches.
- **Space complexity:** $\mathcal{O}(n)$, need to store n elements

2.3 How do we get there?

How can we achieve an algorithm that is $\mathcal{O}(log(log(U))$? Recall that by Master theorem we can see the recurrence relation for binary search is

$$T(n) = log(\frac{n}{2}) + \mathcal{O}(1) = \mathcal{O}(log(n))$$

Using a substitution, and Master theorem we can see that an algorithm that is $\mathcal{O}(log(log(U))$ should have a recurrence relation of the form

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

This argument is laid out very nicely on slide 107 of the Stanford CS166 lecture notes on van Emde Boas Trees. So in other words we are going to want an algorithm that instead of dividing our search space in half at each level (as is done in binary search), divides our search space by the square root at each level.

2.4 When is this useful

This is useful if our universe U is much larger than the number of items we want to store n. So for instance if

$$U = n^{\mathcal{O}(1)}$$

that is our universe is growing polynomial with input size. we have

$$log(log(U)) = log(log(n^{\mathcal{O}(1)})) = log(\mathcal{O}(1)log(n)) = \mathcal{O}(log(log(n)))$$

This also holds if $U = n^{log(\mathcal{O}(1))n}$. Some applications of this would be storing IP addresses in networking contexts.

3 Key idea

Binary search on each level of the tree.

3.1 Definitions

- V The VEB tree
- n number of elements in the tree
- U the universe elements can come from
- $x \in U$ a given element.
- $high(x) = \lfloor \frac{x}{\sqrt{U}} \rfloor = i$
- $low(x) = x \mod \sqrt{U} = j$
- $index(i, j) = i\sqrt{U} + j$

4 Version 1: Bit array

We can store our n elements in a bit array of size U. That is we can have a vector of length U such that is element i of the vector is one, if we want to store the number and otherwise the element is zero.

- **Insertion time:** $O(1)$, to add element i to a bit vector we just need to set the ith index of the bit vector to one. This is constant.
- **Deletion time:** $O(1)$, to remove element i from a bit vector we just need to set the ith index of the bit vector to zero. This is constant.
- **Successor time:** $\mathcal{O}(U)$, suppose we have a bit vector that only stores $U - 1$ and we are looking for 0, in this case we need to look at all U elements
- **Space complexity:** $\mathcal{O}(U)$, we need to store U elements.

5 Version 2: Split the universe into clusters

Idea: Split the universe U into $\sqrt{U}$ clusters each of size $\sqrt{U}$

5.1 Structure

- **Summary Vector:** Week keep a summary bit vector where element i of the summary vector indicates if cluster i is non-empty.
- **Clusters:** $\sqrt{U}$ bit arrays of size $\sqrt{U}$. Cluster i bit j represents the $\sqrt{U} * i + j$ element.

5.2 operations

- **Insertion:** $\mathcal{O}(1)$ to insert an element into the array we at most need to modify one element in the summary vector and one element in a cluster.
- **Deletion:** $\mathcal{O}(\sqrt{U})$, need to go up the tree and check if each or operation has changed. So that is $\sqrt{U}$ operations to check.
- **Successor:** $\mathcal{O}(\sqrt{U})$
  - See algorithm 1
  - This requires in the worst case checking three bit arrays.

Algorithm 1 Version 2, successor

```
Require: target $x \in U$, Tree $V$
Ensure: $x$ or smallest tree element larger than $x$
  $e \leftarrow \mathbb{1}(V.cluster[i, j] = 1)$,                ▷ check if x is in the tree
  if e then
      return x
  else
      $k \leftarrow$ next cluster in summary vector that is non-empty
      $o \leftarrow$ the minimum non-empty element of cluster k.
      return o
  end if
```

6 Version 3: Recursive structure

- Version 2 had just two levels, here we want to recurse on that to have more levels.

6.1 Structure

- Let V be a VEB tree of size U.
- V has two components.
  1. *V.cluster*, a list of $\sqrt{U}$ clusters such that each cluster is a VEB tree of size $\sqrt{U}$
  2. *V.summary* a size $\sqrt{U}$ VEB tree

6.2 A note on base cases

Here we do not explicitly define a base case, but the basic idea is we want to keep splitting until we have broken our search space down to a size that we can brute force search in constant time. That basically means as long as we can fit it all in RAM it is fine.

Algorithm 2 Version 3, insert

```
Require: target: $x \in U$ , VEB tree $V$
1: $insert(V.cluster[high(x)], low(x))$        ▷ Recursively update the clusters
2: $insert(v.summary, high(x))$        ▷ Recursively update the summary
```

6.3 Insert

- See Algorithm 2
- here we need to traverse both the cluster and the summary Trees. Thus this will have $T(U) = 2T(\sqrt{U}) + \mathcal{O}(1) = O(log(U))$
- so this is still slower than we want.

6.4 Successor

Algorithm 3 Version 3, successor

```
Require: target: $x \in U$ , VEB tree $V$
Ensure: index of successor
1: $i \leftarrow high(x)$        ▷ Get the cluster position of $x$
2: $j \leftarrow succ(V.cluster[i], low(x))$        ▷ recursively check that cluster
3: if $j = \emptyset$ then        ▷ If can not find a successor in cluster
4:     $i \leftarrow succ(v.Summary, i)$        ▷ Recursively find the next smallest non-empty cluster
5:     $j \leftarrow succ(V.cluster[i], -\infty)$        ▷ Recursively find the smallest non-empty element in cluster $i$
6: end if
7: return $index(i, j)$        ▷ Return the corresponding tree index
```

- See Algorithm 3
- For similar reasons as the insertion this will have runtime $T(U) = 3T(\sqrt{U}) + \mathcal{O}(1) = O(log(U))$

7 Version 4: Store min and max

- Now suppose that we keep the index of the first (min) and last (max) non-zero bit in each cluster, as well as the summary.

7.1 Insert

- See Algorithm 4
- We still need to do two recursions, so our runtime remains unchanged. $T(n) = \mathcal{O}(log(U))$

Algorithm 4 Version 4, insert

```
Require: target: $x \in U$ , VEB tree $V$
1: if $x < V.min$ then        ▷ update the minimum if x is smaller than current min
2:     $V.min \leftarrow x$
3: end if
4: if $x > V.max$ then        ▷ update the max if x is larger than current max
5:     $V.max \leftarrow x$
6: end if
7: $insert(V.cluster[high(x)], low(x))$
8: $insert(V.summary, high(x))$
```

Algorithm 5 Version 4 Successor

```
Require: VEB $V$, target $x \in U$
Ensure: index of targets successor.
  if $x < V.min$ then
      return $V.min$
  end if
  $i \leftarrow high(x)$
  if $low(x) < V.cluster[i].max$ then
      $j \leftarrow succ(V.summary, low(x))$
  else
      $i \leftarrow succ(V.summary, i)$
      $j \leftarrow V.cluster[i].min$
  end if
  return $index(i, j)$
```

7.2 Successor

- See Algorithm 5
- Using this max/min strategy we at most need to do one recursion per call, thus we will have run time $T(n) = T(\sqrt{N}) + \mathcal{O}(1) = \mathcal{O}(log(log(U))$. So we have achieved the right speed on successor.

8 Version 5: Do not store the min recursively

We can finally speed up the insertion and deletion algorithm, by storing the min and the max directly for each node, instead of having them stored recursively.

8.1 Insert

- See algorithm 6
- Only need to make two recursive calls when a cluster is empty. Otherwise only do one. Thus have time complexity $T(U) = O(\sqrt{U}) + \mathcal{O}(1) = \mathcal{O}(log(log(U))$

Algorithm 6 Version 5 Insert

```
Require: Tree $V$, target $x$
  if $V.min = \emptyset$ then
      $V.min = x = V.max$                    ▷ Store min and max directly for empty tree
      return
  end if
  if $x < V.min$ then                        ▷ Update the min but old min needs insertion
      $a \leftarrow x$
      $x \leftarrow V.min$
      $V.min \leftarrow a$
  end if
  if $x > V.max$ then                        ▷ Update max
      $V.max \leftarrow x$
  end if
  if $V.cluster[high(x)].min = \emptyset$ then   ▷ Update summary
      $insert(V.cluster[high(x)], low(x))$
  end if
  $insert(v.cluster[high(x)], low(x))$       ▷ Update clusters
```

8.2 Delete

We did not cover this so I will not include it, but it was covered in the oficial lecture notes.

References

[1] Peter van Emde Boas Preserving order in a forest in less than logarithmic time In 16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975, pages 75–84. IEEE Computer Society, 1975.

[2] Stanford CS166 Slides, link