

# On vEB Trees

David Keisar Schmidt

February 11, 2022

## Abstract

A Predecessor algorithm defines a generic sorting algorithm, that is, find the elements set maximum and repeatedly find its predecessor. Thus, the elements number is  $n$  and the algorithm complexity is  $O(f(n))$ , the resulted sorting algorithm complexity is  $O(n \cdot f(n))$ . It is known that the sorting complexity has a spell bound of  $n \log n$ , so without any additional assumptions, it follows that the time complexity of any predecessor algorithm is bounded by  $\log n$  since otherwise we could achieve sorting time better than  $n \log n$ . However, by bounding the size of the largest element  $u$ , we can achieve sorting in  $O(u)$  and when  $u = O(n)$ , it becomes a linear time sorting.<sup>1</sup> Hence, a question arises - can we achieve better predecessor algorithms by bounding  $u$ ?

---

<sup>1</sup>See the [counting sort algorithm](#)

## Contents

<b>1</b>	<b>A constant height tree</b>	<b>3</b>
<b>2</b>	<b>A recursive data structure</b>	<b>4</b>
<b>3</b>	<b>Proto vEB Trees</b>	<b>5</b>
3.1	Determining Membership . . . . .	6
3.2	Finding The Minimum Element . . . . .	7
3.3	Finding The Successor . . . . .	8
3.4	Insertion . . . . .	10
3.5	Deletion . . . . .	10
<b>4</b>	<b>The van Emde Boas tree</b>	<b>11</b>
<b>5</b>	<b>Bibliography</b>	<b>12</b>

## 1 A constant height tree

Let  $u$  be an upper bound of an array  $M$ , which means the array contains elements in the range  $\{0, \dots, u-1\}$ . We define an array  $A[x] = \mathbb{1}_{x \in M}$ , that is,  $A$  is a bitfield array containing 1 in the index  $x$  iff  $x \in M$  and 0 otherwise.

In addition, we define a summary array of size  $\sqrt{u}$ , where  $\text{summary}[i]$  tells us if there is a 1 in the range  $A[i\sqrt{u}, \dots, (i+1)\sqrt{u}-1]$ , thus,  $\text{summary}[i] =$

$\bigvee_{j \in A[i\sqrt{u}, \dots, (i+1)\sqrt{u}-1]} j$ , the logical or of the array  $A[i\sqrt{u}, \dots, (i+1)\sqrt{u}-1]$ .

The  $\sqrt{u}$  sized sub-array  $A[i\sqrt{u}, \dots, (i+1)\sqrt{u}-1]$  is called the  $i^{\text{th}}$  cluster of  $A$ . Besides, we add the logical or of summary - containing 1 iff the array is not empty.

Now, in order to find minimum and maximum we can do the following operations:

- Minimum: Find the left most index  $i$  in the summary with value 1, run a linear scan in the  $i^{\text{th}}$  cluster to find the left most positive value, which is, the minimum.

Besides, now, given an element  $x$ , we know that  $x$  is in cluster number  $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$ , thus, in order to delete  $x$ , we update  $A[x] = 0$ , and recompute  $\text{summary}\left[\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor\right]$ , and the logical or of summary. a similar operation is required for insertion.

In both operations we needed to scan an array of size  $\sqrt{u}$ , so the time complexity if both cases is  $O(\sqrt{u})$ , that is, however, extremely poor, since AVL, Red&Black trees achieve better results -  $O(\log u)$ . Though we shall see we can develop it and achieve better running time, since this design will turn out as a key design.

## 2 A recursive data structure

Instead of storing only two tree levels, we create a recursive data structure. The array “summary” remains the same, however,  $\text{summary}[i]$  contains also a smaller structure of size  $\sqrt{u}$ , of  $i^{\text{th}}$  cluster elements, with the same property. Which means, the first level of the tree contains  $u$  elements, the second level contains  $u^{\frac{1}{2}}$  structures each containing  $u^{\frac{1}{2}}$  elements, the third contains  $u^{\frac{1}{4}}$  of  $u^{\frac{1}{4}}$  elements and so on. We should assume just for know that  $u = 2^{2^k}$  so in each level we get an integer.

Consider the following recurrence  $T(u) = T(\sqrt{u}) + O(1)$  that is, in each step we shrink the recursion by a factor of  $\sqrt{u}$ , substituting  $m = \lg u$  we get  $u = 2^m$  thus  $T(2^m) = T(2^{\frac{m}{2}}) + O(1)$  thus the recurrence  $S(m) = T(2^m) = S(\frac{m}{2}) + O(1)$  has the solution  $S(m) = O(\lg m)$  this  $T(u) = O(\lg m) = O(\lg \lg u)$ , there we would like to design our tree to behave in a similar way - it requires a tree height of  $\sqrt{u}$ .

Now, consider the previous structure with constant height, given a value  $x$ , it is stored in cluster number  $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor = i$  and in  $x$ 's cluster, the elements are  $\sqrt{u}i, \dots, \sqrt{u}(i+1) - 1$  and  $x$  between them, thus its index is  $x \bmod \sqrt{u}$ ,

Note that since  $x < u = 2^{2^k}$ ,  $x$  is represented by at most  $\lg u$  bits and if we view  $x$  as a  $\lg u$  binary number,  $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$  is represented by the most significant  $\frac{\lg u}{2}$  bits of  $x$ , and  $x \bmod \sqrt{u}$  is given by the least  $\frac{\lg u}{2}$  bits of  $x$ . There is a connection between those two expressions:

$$x = x \bmod \sqrt{u} + \sqrt{u} \cdot \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$$

Thus, we define the following functions:

$$\begin{aligned}\text{high}(x) &= \left\lfloor \frac{x}{\sqrt{u}} \right\rfloor \\ \text{low}(x) &= x \bmod \sqrt{u} \\ \text{index}(x, y) &= x\sqrt{u} + y\end{aligned}$$

and it follows that  $\text{index}(\text{high}(x), \text{low}(x)) = x$ , and that is how we can generate  $x$  back just from those two simple functions.

This discussion leads us to the vEB-prototype.

**Definition 1.** The universe size in each level in the tree is  $u^{\frac{1}{2^k}}$  where  $k$  is the height of the tree ( $k = 0$  is the leaves level).

### 3 Proto vEB Trees

We will now define the prototype of vEB tree, which we Denote as *proto-vEB*( $u$ ) where  $u$  is its universe size, as follows:

- if  $u = 2$  this is the base case - we contain only an array of two bits  $A[0 \dots 1]$
- otherwise  $u = 2^{2^k}$  for some integer  $k \geq 1$ , thus,  $u \geq 4$ . In addition to the universe size, proto-vEB contains the following attributes:

- ◁ a pointer named **summary** to a *proto-vEB*( $\sqrt{u}$ ) structure.
- ◁ an array **cluster**[ $0 \dots \sqrt{u} - 1$ ] of  $\sqrt{u}$  pointers, each points to a *proto-vEB*( $\sqrt{u}$ ) structure.

For a given element  $x \in \{0, \dots, u - 1\}$ ,  $x$  is recursively stored in cluster number  $\text{high}(x)$  as the  $\text{low}(x)$  number.

As not as the simple structure we described in the previous section, the array summary does not contain explicit results, but allows us to compute the summary bit recursively. In particular, now, summary contains the indices of clusters that contains any array elements. Since the naive structure's summary at some index  $i$  contained 1 iff there was an array element in the range  $\{i\sqrt{u}, \dots, (i+1)\sqrt{u} - 1\}$ , now in the prototype, we have the same property, but it is achieved recursively.

Now, our goal is to describe algorithms for the following tasks:

1. Query operations - Member, Minimum
2. Successor - does not change the structure.
3. Insert and Delete.

Maximum and predecessor can be achieved in a similar manner. All those operations receive an element  $x$  and assume its validity, that is  $0 \leq x < u$ .

### 3.1 Determining Membership

In order to find  $x$ , we need to split the task into two cases:

- if  $u = 2$ , we only to check if  $A[x] = 1$ .
- otherwise, we need to search for  $\text{low}(x)$  in cluster  $[\text{high}(x)]$ , a recursive process.

Assume the data structure is  $V$ , we get the following algorithm:

---

**Algorithm 1** Proto-vEB-Member( $V, x$ )

---

- 1: **if**  $V.u == 2$  **then**
  - 2:     **return**  $V.A[x]$
  - 3: **return** Proto-vEB-Member( $V.\text{cluster}[\text{high}(x)], \text{low}(x)$ )
-

The running time of this algorithm is

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

, as we have already analyzed, the solution is  $T(u) = O(\lg \lg u)$ , which is far better than  $O(\sqrt{u})$  and  $O(\lg u)$ .

### 3.2 Finding The Minimum Element

In previous sections, we described a naive solution - scanning the summary array from the left till we find 1, and then scanning the corresponding cluster from the left to find the minimum. Now, when summary is not an array but a proto-vEB, we can achieve better results. Firstly, denote that the minimum element appears in the cluster with minimum index, thus, we can search for minimum in summary and get  $min - cluster = \text{high}(x)$  index, and then again in the cluster to find the *offset* which is  $\text{low}(x)$ . Thus, we shall return  $\text{index}(\text{high}(x), \text{low}(x))$ . Secondly, the base case is again when  $u = 2$ , then we just scan the array. Our algorithm returns NIL when the structure is empty, that is, there is not minimum.

**Algorithm 2** Proto-vEB-Minimum( $V$ )

---

```

1: if  $V.u == 2$  then
2:   if  $V.A[0] == 1$  then
3:     return 0
4:   else if  $V.A[1] == 1$  then
5:     return 1
6:   else return NIL
7: else min-cluster=Proto-vEB-Minimum( $V.summary$ )
8:   if min-cluster==NIL then
9:     return NIL
10:  else offset=Proto-vEB-Minimum( $V.cluster[min - cluster]$ )
11:    return index(min-cluster, offset)

```

---

The algorithm in the worst case performs two recursive calls and some constant operations, so the running time is

$$T(u) = 2T(\sqrt{u}) + O(1)$$

To solve that we define  $m = \lg u$  thus  $u = 2^m$  and the formula  $S(m) = T(2^m) = 2T(2^{\frac{m}{2}}) + O(1) = 2S(\frac{m}{2}) + O(1)$ , hence, from the master theorem,  $S(m) = \Theta(m) = \Theta(\lg u)$ . That is not better than a balanced bst, and we will see we can improve it.

### 3.3 Finding The Successor

In order to find the successor, we need, again to split to some cases.

1. Base case: if  $u = 2$ , if  $x = 1$ , there is no successor, and if  $x = 0$  we return 1 if  $A[1] = 1$  and 0 NIL otherwise.
2. We do know the cluster of  $x$  which is  $\text{high}(x)$ , so we can  $\text{low}(x)$  successor in the  $\text{high}(x)$  cluster and if we receive an index  $i$  we can return



$\text{index}(\text{high}(x), i)$ . otherwise, we move to the next cluster till we find the successor.

3. If now successor was found, we return NIL.

However, since the summary array is a proto-veB structure itself, if no successor was found in cluster number  $\text{high}(x)$  we can find the successor of  $\text{high}(x)$  in summary, and then returns its minimum, if exists. This yields the following algorithm:

---

**Algorithm 3** Proto-veB-Successor( $V, x$ )

---

```

1: if  $V.u == 2$  then
2:   if  $x == 0$  and  $V.A[0] == 1$  then
3:     return 1
4:   else return NIL
5: else  $\text{offset} = \text{Proto-veB-Successor}(V.\text{cluster}[\text{high}(x)], \text{low}(x))$ 
6:   if  $\text{offset} \neq \text{NIL}$  then
7:     return  $\text{index}(\text{high}(x), \text{offset})$ 
8:   else  $\text{succ-cluster} = \text{Proto-veB-Successor}(V.\text{summary}, \text{high}(x))$ 
9:     if  $\text{succ-cluster} == \text{NIL}$  then
10:      return NIL
11:     else  $\text{offset} = \text{Proto-veB-Minimum}(V.\text{cluster}[\text{succ-cluster}])$ 
12:     return  $\text{index}(\text{succ-cluster}, \text{offset})$ 

```

---

In the worst case, our algorithm does two recursive calls and a call to Proto-veB-Minimum, thus the recurrence is

$$T(u) = 2T(\sqrt{u}) + \Theta(\lg u)$$

we can simply prove that  $T(u) = \Theta(\lg u \lg \lg u)$ .

### 3.4 Insertion

in order to insert an element we need to insert  $\text{low}(x)$  to the cluster number  $\text{high}(x)$  and insert  $\text{high}(x)$  to summary.

Thus we get the following algorithm:

---

**Algorithm 4** Proto-vEB-Insert( $V, x$ )

---

```

1: if  $V.u == 2$  then
2:    $V.A[x] = 1$ 
3: else Proto-vEB-Insert( $V.\text{cluster}[\text{high}(x)], \text{low}(x)$ )
4:   Proto-vEB-Insert( $V.\text{summary}, \text{high}(x)$ )

```

---

The recurrence is  $T(u) = 2T(\sqrt{u}) + O(1)$  so as we proved in previous sections,  $T(u) = \Theta(\lg u)$ . not good enough.

### 3.5 Deletion

The current structure requires a linear scan over all  $\sqrt{u}$  in  $x$ 's cluster, or, the addition of an element *count* that counts the number of elements in the structure.

Assuming  $n$  exists:

---

**Algorithm 5** Proto-vEB-Delete( $V, x$ )

---

```

1: if  $V.u == 2$  then
2:    $V.A[x] = 0$ 
3:    $V.\text{count} = V.\text{count} - 1$ 
4: else Proto-vEB-Delete( $V.\text{cluster}[\text{high}[x]], \text{low}(x)$ )
5:   if  $V.\text{cluster}[\text{high}[x]].\text{count} == 1$  then
6:     Proto-vEB-Delete( $V.\text{summary}, \text{high}(x)$ )

```

---

The recurrence in the worst case is  $T(u) = 2T(\sqrt{u}) + O(1)$ , thus  $T(u) = O(\lg u)$ .

## 4 The van Emde Boas tree

Our prototype suffers from an impractical assumption -  $u = 2^{2^k}$ . Our structure needs to be able to deal with more “flexible” numbers, we will assume they are just powers of two  $u = 2^m$ . if  $m$  is odd, that is  $m = 2k + 1$ , we divide the bits of  $u$  to  $\left\lfloor \frac{\lg u}{2} \right\rfloor$  most significant bits and  $\left\lfloor \frac{\lg u}{2} \right\rfloor$  least significant bits. It follows that  $\sqrt[2]{u} \leq \sqrt{u} \leq \sqrt[2]{u}$  and  $\sqrt[2]{u} \cdot \sqrt[2]{u} = u$ .

For simplicity we denote  $\sqrt[2]{u} = 2^{\lfloor \frac{\lg u}{2} \rfloor}$ ,  $\sqrt[2]{u} = 2^{\lceil \frac{\lg u}{2} \rceil}$ . Thus, the fundamental methods we have already defined become:

$$\begin{aligned} \text{high}(x) &= \left\lfloor \frac{x}{\sqrt[2]{u}} \right\rfloor \\ \text{low}(x) &= x \bmod \sqrt[2]{u} \\ \text{index}(x, y) &= x \cdot \sqrt[2]{u} + y \end{aligned}$$

We denote our structure as *vEB tree*, which is a modification to the Proto-vEB-tree we defined in the previous section. A vEB tree with universe size  $u$  is denoted as  $vEB(u)$ , and contains the following attributes:

- summary - points to a  $vEB(\sqrt[2]{u})$  tree.
- cluster[0...  $\sqrt[2]{u} - 1$ ] - points to  $\sqrt[2]{u} \cdot vEB(\sqrt[2]{u})$  trees.
- min - stores the minimum element in the vEB tree.
- max - stores the maximum element in the vEB tree.

*Remark 1.* min is stored only in the base vEB tree, and not in its recursive sub trees, thus, all the attributes stores in  $V$  is  $min$  plus all the elements recursively stored in the sub trees.

can be improved, for instance, saving two more attributes  $min, max$ , and maintaining their value, can improve the running time of Minimum and Maximum to  $O(1)$ .

## 5 Bibliography

This scribe is mostly based on the book “Introduction to algorithms” by Thomas H. Cormen, without him, it would have been impossible.

We encourage you to read Cormen’s vEB chapter for more examples and explanations.