Van Emde Boas Tree

November 21, 2021

Swapnil Saurav (201EEB1204) ,
Tanishk Gupta (201EEB1205) ,
Yash Jain (201EEB1208)

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Sarthak Joshi

Summary: This project report highlights the study by our group on Van Emde Boas Tree. The objective of this project was to understand and implement the Van Emde Boas Tree (or simply, VEB Tree). This was further extended to its comparison with other data structures like AVL Trees, B-Trees etc. It was expected and observed that this data structure has lesser time complexity than those data structures. However, in some cases the space complexity is more in VEB Trees as compared to other data structures. Numerous studies and articles are cited in relation to this data structure. They were understood, implemented and desired results were obtained.

---

1. Introduction

This document is based on the understanding, implementation and analysis of VEB Trees. A VEB Tree is a data structure which can perform insert, search, delete, predecessor and successor functions in O(loglog(M)) time complexity (where M is the largest number in the user data). Minimum and Maximum can be obtained in O(1) time. This turns out to be much faster than other data structures like AVL Trees which perform this operation in O(log(N)) time. However, it takes O(M) space to provide such a great time complexity.

Complexity analysis of different functions/algorithms

	VEB Tree	AVL Tree
insert	O(loglog(M))	O(log(N))
delete	O(loglog(M))	O(log(N))
search	O(loglog(M))	O(log(N))
predecessor	O(loglog(M))	O(log(N))
successor	O(loglog(M))	O(log(N))
Min	O(1)	O(log(N))
Max	O(1)	O(log(N))
Space Complexity	O(M)	O(N)

Table 1: Complexity analysis.

From the above table, the following points are observed:

1. This data structure is only highly time efficient when log(M) is less than N. If the number of elements is less but the maximum is very high, this becomes inefficient.

2. This is not space efficient if M is very larger than N. This will result in very high memory consumption and time efficiency will be of no use.
3. For N ≈ M, this is highly efficient and should be used for better productivity.

In this document, we will discuss its basic structure, implementation and result discussion.

2. Basic Structure

The basic node structure of a VEB Tree as explained in the CLRS book [1] is as shown in the figure 1 below.

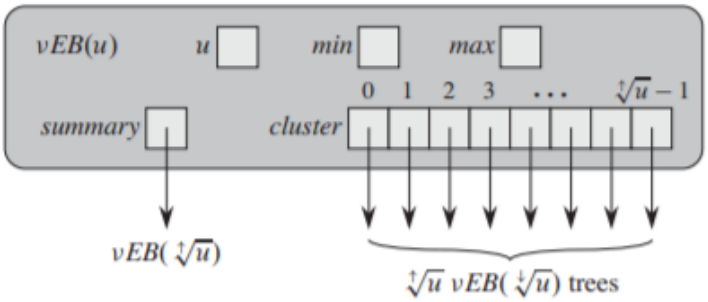

Figure 1: The node structure of VEB Tree.

Each node is defined by a universe size U. Each node stores the min and max of its subtree. In this way, it can provide min and max in O(1) time. The summary is used to check whether a subtree is completely empty or not.

The below example figure 2 from the CLRS book [1] shows an easy understanding of the summary. Note that the below figure doesn't exactly shows the VEB Tree. It is just an example to understand.

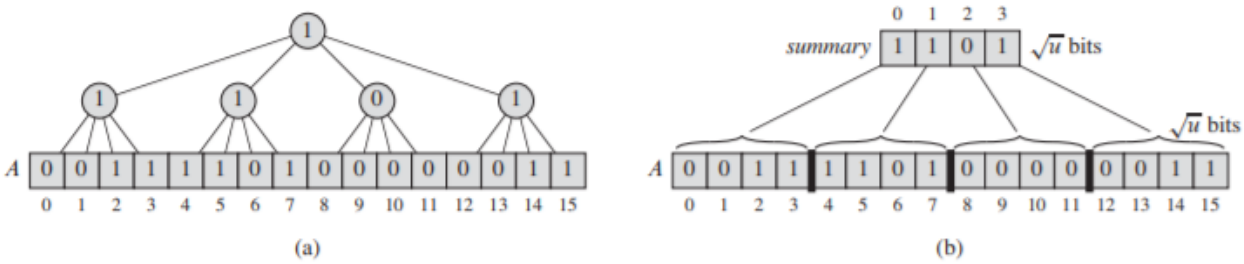

Figure 2: Example of the summary.

The overall tree is a recursive structure of these nodes, shrinking the universe size by $\sqrt{u}$ at each level down to a base size of 2. From the CLRS book [1], the structure is obtained as shown in figure 3.

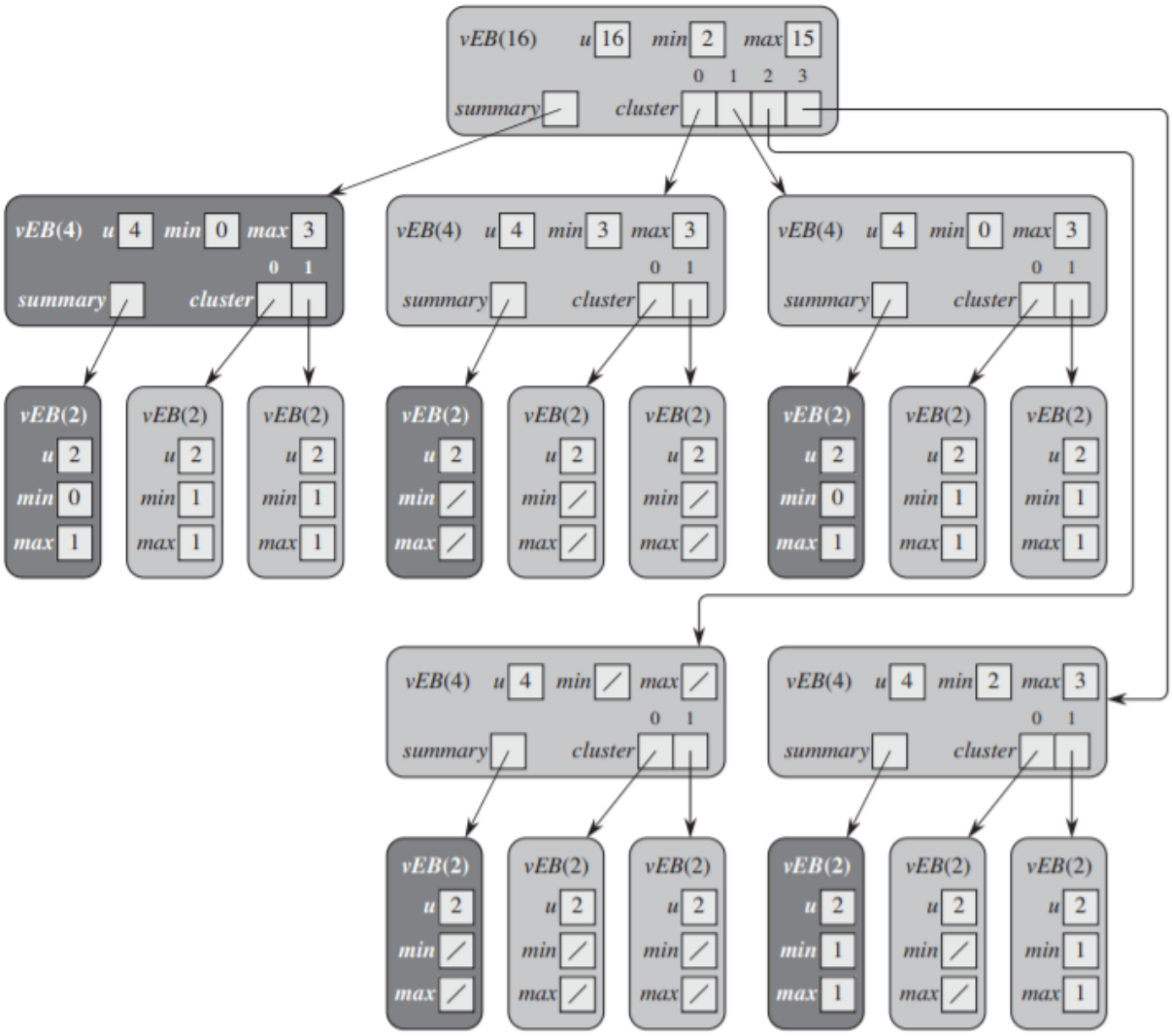

Figure 3: The Tree structure of VEB Tree.

Most of the structure is understood by the CLRS book [1], while some understanding is done with the help of Geeks For Geeks [2] and an article by PUC RIO [3].

3. Algorithms with Pseudo Codes

In the implementation, various functions were implemented. Their pseudo codes are given below. Note that, insert and delete have almost similar algorithms and hence only one of them (i.e. insert) has been mentioned. Similarly, predecessor and successor are almost same and only successor is implemented for the sake of easy understanding.
Most of the algorithms are studied and understood by the CLRS book [1], while some understanding of the function and structures were done by Geeks For Geeks [2].

Algorithm 1 Insert-in-empty(V,x)

```
1: V.min = x
2: V.max = x
```

Algorithm 2 Insert(V,x)

```
k = ceil(√Usize), Low(x) = x%k , High(x)=x/k
if V.min == NIL then
    Insert-in-empty(V,x)
else if x < V.min then
    swap x with V:min
    if V.Usize > 2 then
        if MINIMUM(V.cluster[High(x)]) == NIL then
            Insert(V.summary,High(x))
            Insert-in-empty(V.cluster[High(x)],Low(x))
        else
            Insert(V.cluster[High(x)],Low(x))
        end if
    end if
    if x > V.max then
        V.max = x
    end if
end if
```

Algorithm 3 Search(V,x)

```
k = ceil(√Usize)
if x >= V.Usize or x < 0 then
    Out of bounds
    return false
end if
if x == V.max or x == V.min then
    return true
end if
if V.Usize == 2 then
    return false
end if
return Search(V.clusters[x/k],x mod k)
```

Algorithm 4 Successor(V,x)

```
k = ceil(√Usize), Low(x) = x%k , High(x)=x/k, index(a,b) = a * k + b
if V.Usize == 2 then
    if x == 0 and V.max == 1 then
        return 1
    else
        return -1
    end if
else if V.min != −1 and x < V.min then
    return V.min
else
    max-low = MAXIMUM(V.cluster[High(x)])
    if max − low != −1 and Low(x) < max − low then
        offset = Successor(V.cluster[High(x)],Low(x))
        return index(High(x),offset)
    else
        succ-cluster = Successor(V.summary,High(x))
        if succ − cluster == −1 then
            return -1
        else
            offset = MINIMUM(V.cluster[succ − cluster])
            return index(succ-cluster,offset)
        end if
    end if
end if
```

4. Analysis

After implementing the above data structure and various helper function, its time complexity analysis was done. We have compared it with AVL Tree for a better understanding.
For this, we first created a random array (this may have duplicates) which tells us which numbers are present in the tree. Then using the $< time.h >$ header file, we calculated the time taken for various functions.
For insert function, we know that the AVL Tree takes $log(N)$ time, while the VEB tree takes $log(log(M))$ time. It wasn't possible to plot both of these in the same chart because the difference will be very high at larger inputs. For example, when AVL Tree takes 15000 milliseconds, the VEB Tree will take only 5 milliseconds. Hence, the VEB Tree line will appear zero always.
For a better understanding, we plotted the log of time taken by AVL Tree with the time taken by VEB Tree to show that there two are comparable. The figure 4 shows the chart for the insert function.

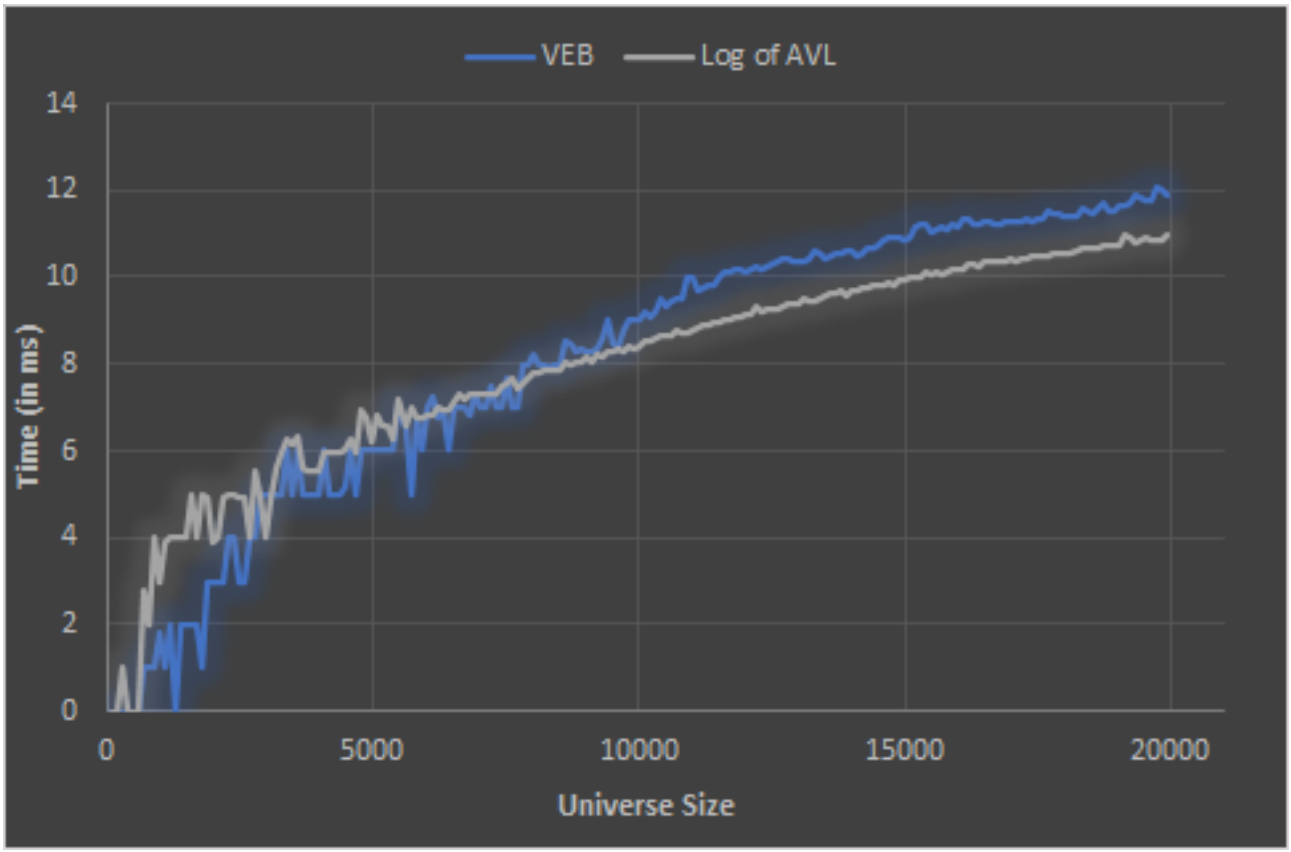

Figure 4: Time complexity analysis for Insert function.

5. Conclusions

The VEB Tree was understood, implemented and analysed and the following conclusions were made in the project:

- The VEB Tree is a very time efficient tree when the Universe Size (the maximum of the data) is very close to the number of elements. Therefore, It can be used in many implementations.
- It was observed from the graph that it is highly faster than AVL Tree. However, it is space inefficient when $M >> N$. It takes up a very huge space in storing the summary and child clusters.
- In future, the developments can be made in optimizing its space complexity. This can be used by using pointers and allocating and freeing their memory every time when we insert and delete an element respectively. This can reduce the space but its implementation would be challenging task.

6. Acknowledgements

We would like to express our special thanks of gratitude and appreciation to our course instructor Dr. Anil Shukla (Computer Science Department, IIT Ropar). Without his teaching, motivation and support, it would be impossible to complete this project.
We would also like to thank all the Teaching Assistants who helped us in this project by explaining us the usage of GitHub, Overleaf, Debugger. Their help in understanding the conceptual side of the project is really appreciable.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, London, England, 3 edition, 2009. The CLRS book.

[2] Geeks for Geeks. Van emde boas tree, basics and construction. *GFG Community*, August 2019.

[3] Eduardo Laber and David Sotelo. Van emde boas tree. *PUC RIO*, 2000.

A. Appendix A

Real Life Examples

- In practice, the van Emde Boas layout (based on van Emde Boas tree) is used in cache-oblivious data structures meaning faster data access.
- It is used as a priority queue in algorithms such as dijkstra.
- It is used in routing applications.

B. Appendix B

Some general terms used

- AVL tree is self balancing binary search tree that performs operations such as insertion , deletion and search in **O(log(n))**.
- **Predecessor(x)** is the largest element is the given set that is strictly smaller than x.
- **Successor(x)** is the smallest element in the given set that is strictly greater than x.

C. Appendix C

Proof of Time Complexity

Solving the recurrence,

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

let's assume, $n = 2^k$

We have, $T(2^k) = T(2^{\frac{k}{2}}) + \mathcal{O}(1)$

Now assume $T(2^k) = S(k)$

$S(k) = S(\frac{k}{2}) + \mathcal{O}(1)$

Using rule 2 of master theorem,

$n \log_b a = n^k$ then $T(n) = \theta(n^k log(n))$

using above, we get

$S(k) = log(k)$

further we have,

$n = 2^k, k = log(n)$

$T(n) = S(log(k)) = log(log(n))$

Therefore the time complexity is $\mathcal{O}(log(log(n))$.