



WIKIPEDIA  
The Free Encyclopedia

Main page  
Contents  
Featured content  
Current events  
Random article  
Donate to Wikipedia  
Wikipedia store

Interaction  
Help  
About Wikipedia  
Community portal  
Recent changes  
Contact page

Tools  
What links here  
Related changes  
Upload file  
Special pages  
Permanent link  
Page information  
Wikidata item  
Cite this page

Print/export  
Create a book  
Download as PDF  
Printable version

Languages  
Deutsch  
فارسی  
Українська  
Tiếng Việt  
 Edit links

Create account Log in

Article **Talk**

Read **Edit** View history

Search

# Van Emde Boas tree

From Wikipedia, the free encyclopedia



This article **needs attention from an expert in computer science**. The specific problem is: **bug in pseudocode**. See the [talk page](#) for details. [WikiProject Computer science](#) (or its [Portal](#)) may be able to help recruit an expert. *(March 2013)*

A **Van Emde Boas tree** (or **Van Emde Boas priority queue**; Dutch pronunciation: [van ˈɛmdə ˈboːas]), also known as a **vEB tree**, is a **tree data structure** which implements an **associative array** with *m*-bit integer keys. It performs all operations in  $O(\log m)$  time, or equivalently in  $O(\log \log M)$  time, where  $M=2^m$  is the maximum number of elements that can be stored in the tree. The *M* is not to be confused with the actual number of elements stored in the tree, by which the performance of other tree data-structures is often measured. The vEB tree has good space efficiency when it contains a large number of elements, as discussed below. It was invented by a team led by Dutch computer scientist **Peter van Emde Boas** (de) in 1975.<sup>[1]</sup>

## Contents

- 1 Supported operations
- 2 How it works
  - 2.1 FindNext
  - 2.2 Insert
  - 2.3 Delete
  - 2.4 Discussion
- 3 References
  - 3.1 Further reading

## Van Emde Boas tree

|  |  |
|--|--|
| <b>Type</b>                                    | Non-binary <a href="#">tree</a>              |
| <b>Invented</b>                                | 1975   |
| <b>Invented by</b>                             | <b>Peter van Emde Boas</b> <span>(de)</span> |
| <b>Asymptotic complexity in big O notation</b> |  |
| <b>Space</b>                                   | $O(M)$                                       |
| <b>Search</b>                                  | $O(\log \log M)$                             |
| <b>Insert</b>                                  | $O(\log \log M)$                             |
| <b>Delete</b>                                  | $O(\log \log M)$                             |

## Supported operations [\[edit\]](#)

A vEB supports the operations of an *ordered associative array*, which includes the usual associative array operations along with two more *order* operations, *FindNext* and *FindPrevious*.<sup>[2]</sup>

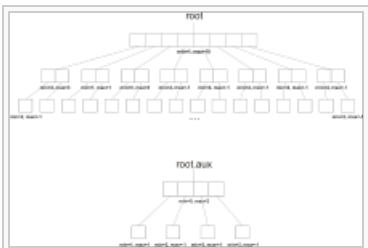
- Insert*: insert a key/value pair with an *m*-bit key
- Delete*: remove the key/value pair with a given key
- Lookup*: find the value associated with a given key
- FindNext*: find the key/value pair with the smallest key at least a given *k*
- FindPrevious*: find the key/value pair with the largest key at most a given *k*

A vEB tree also supports the operations *Minimum* and *Maximum*, which return the minimum and maximum element stored in the tree respectively.<sup>[3]</sup> These both run in  $O(1)$  time, since the minimum and maximum element are stored as attributes in each tree.

## How it works [\[edit\]](#)

For the sake of simplicity, let  $\log_2 m = k$  for some integer *k*. Define  $M=2^m$ . A vEB tree *T* over the universe  $\{0,...,M-1\}$  has a root node that stores an array *T.children* of length  $\sqrt{M}$ . *T.children*[*i*] is a pointer to a vEB tree that is responsible for the values  $\{i\sqrt{M},..., (i+1)\sqrt{M}-1\}$ . Additionally, *T* stores two values *T.min* and *T.max* as well as an auxiliary vEB tree *T.aux*.

Data is stored in a vEB tree as follows: The smallest value currently in the tree is stored in *T.min* and largest value is stored in *T.max*. Note that *T.min* is not stored anywhere else in the vEB tree, while *T.max* is. If *T* is



An example Van Emde Boas tree with dimension 5 and the root's aux structure after 1, 2, 3, 5, 8 and 10 have

empty then we use the convention that  $T.max = -1$  and  $T.min = M$ . Any

other value  $x$  is stored in the subtree  $T.children[i]$  where  $i = \left\lfloor \frac{x}{\sqrt{M}} \right\rfloor$ .

been inserted.

The auxiliary tree  $T.aux$  keeps track of which children are non-empty, so  $T.aux$  contains the value  $j$  if and only if  $T.children[j]$  is non-empty.

## FindNext [\[edit\]](#)

The operation  $FindNext(T, x)$  that searches for the successor of an element  $x$  in a vEB tree proceeds as follows: If  $x \leq T.min$  then the search is complete, and the answer is  $T.min$ . If  $x > T.max$  then the next element does not exist, return  $M$ . Otherwise, let  $i = \lfloor x / \sqrt{M} \rfloor$ . If  $x \leq T.children[i].max$  then the value being searched for is contained in  $T.children[i]$  so the search proceeds recursively in  $T.children[i]$ . Otherwise, We search for the value  $i$  in  $T.aux$ . This gives us the index  $j$  of the first subtree that contains an element larger than  $x$ . The algorithm then returns  $T.children[j].min$ . The element found on the children level needs to be composed with the high bits to form a complete next element.

```
function FindNext(T, x).
    if x ≤ T.min then
        return T.min
    if x > T.max then // no next element
        return M
    i = floor(x/√M)
    lo = x % √M
    hi = x - lo
    if lo ≤ T.children[i].max then
        return hi + FindNext(T.children[i], lo)
    return hi + T.children[FindNext(T.aux, i)].min
end
```

Note that, in any case, the algorithm performs  $O(1)$  work and then possibly recurses on a subtree over a universe of size  $M^{1/2}$  (an  $m/2$  bit universe). This gives a recurrence for the running time of  $T(m) = T(m/2) + O(1)$ , which resolves to  $O(\log m) = O(\log \log M)$ .

## Insert [\[edit\]](#)

The call  $insert(T, x)$  that inserts a value  $x$  into a vEB tree  $T$  operates as follows:

If  $T$  is empty then we set  $T.min = T.max = x$  and we are done.

Otherwise, if  $x < T.min$  then we insert  $T.min$  into the subtree  $i$  responsible for  $T.min$  and then set  $T.min = x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$

Otherwise, if  $x > T.max$  then we insert  $x$  into the subtree  $i$  responsible for  $x$  and then set  $T.max = x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$

Otherwise,  $T.min < x < T.max$  so we insert  $x$  into the subtree  $i$  responsible for  $x$ . If  $T.children[i]$  was previously empty, then we also insert  $i$  into  $T.aux$ .

In code:

```
function Insert(T, x)
    if T.min > T.max then // T is empty
        T.min = T.max = x;
        return
    if T.min == T.max then
        if x < T.min then
            T.min = x
        if x > T.max then
            T.max = x
    if x < T.min then
        swap(x, T.min)
    if x > T.max then
        T.max = x
    i = floor(x / √M)
    Insert(T.children[i], x % √M)
    if T.children[i].min == T.children[i].max then
        Insert(T.aux, i)
end
```

The key to the efficiency of this procedure is that inserting an element into an empty vEB tree takes  $O(1)$  time. So, even though the algorithm sometimes makes two recursive calls, this only occurs when the first recursive call was into an empty subtree. This gives the same running time recurrence of  $T(m)=T(m/2) + O(1)$  as before.

## Delete [\[edit\]](#)

Deletion from vEB trees is the trickiest of the operations. The call *Delete*(*T*, *x*) that deletes a value *x* from a vEB tree *T* operates as follows:

If  $T.min = T.max = x$  then *x* is the only element stored in the tree and we set  $T.min = M$  and  $T.max = -1$  to indicate that the tree is empty.

Otherwise, if  $x = T.min$  then we need to find the second-smallest value *y* in the vEB tree, delete it from its current location, and set  $T.min=y$ . The second-smallest value *y* is either  $T.max$  or  $T.children[T.aux.min].min$ , so it can be found in  $O(1)$  time. In the latter case we delete *y* from the subtree that contains it.

Similarly, if  $x = T.max$  then we need to find the second-largest value *y* in the vEB tree and set  $T.max=y$ . The second-largest value *y* is either  $T.min$  or  $T.children[T.aux.max].max$ , so it can be found in  $O(1)$  time. We also delete *x* from the subtree that contains it.

In case where *x* is not  $T.min$  or  $T.max$ , and *T* has no other elements, we know *x* is not in *T* and return without further operations.

Otherwise, we have the typical case where  $x \neq T.min$  and  $x \neq T.max$ . In this case we delete *x* from the subtree  $T.children[i]$  that contains *x*.

In any of the above cases, if we delete the last element *x* or *y* from any subtree  $T.children[i]$  then we also delete *i* from  $T.aux$

In code:

```
function Delete(T, x)
  if T.min == T.max == x then
    T.min = M
    T.max = -1
    return
  if x == T.min then
    if T.aux is empty then
      T.min = T.max
      return
    else
      x = T.children[T.aux.min].min
      T.min = x
  if x == T.max then
    if T.aux is empty then
      T.max = T.min
      return
    else
      T.max = T.children[T.aux.max].max
  if T.aux is empty then
    return
  i = floor(x /  $\sqrt{M}$ )
  Delete(T.children[i], x %  $\sqrt{M}$ )
  if T.children[i] is empty then
    Delete(T.aux, i)
end
```

Again, the efficiency of this procedure hinges on the fact that deleting from a vEB tree that contains only one element takes only constant time. In particular, the last line of code only executes if *x* was the only element in  $T.children[i]$  prior to the deletion.

## Discussion [\[edit\]](#)

The assumption that  $\log m$  is an integer is unnecessary. The operations  $x/\sqrt{M}$  and  $x\% \sqrt{M}$  can be replaced by taking only higher-order  $\text{ceil}(m/2)$  and the lower-order  $\text{floor}(m/2)$  bits of *x*, respectively. On any existing machine, this is more efficient than division or remainder computations.

The implementation described above uses pointers and occupies a total space of  $O(M) = O(2^m)$ . This can be seen as follows. The recurrence is  $S(M) = O(\sqrt{M}) + (\sqrt{M} + 1) \cdot S(\sqrt{M})$ . Resolving that

would lead to  $S(M) \in (1 + \sqrt{M})^{\log \log M} + \log \log M \cdot O(\sqrt{M})$ . One can, fortunately, also show that  $S(M) = M - 2$  by induction.<sup>[4]</sup>

In practical implementations, especially on machines with *shift-by-k* and *find first zero* instructions, performance can further be improved by switching to a **bit array** once *m* equal to the **word size** (or a small multiple thereof) is reached. Since all operations on a single word are constant time, this does not affect the asymptotic performance, but it does avoid the majority of the pointer storage and several pointer dereferences, achieving a significant practical savings in time and space with this trick.

An obvious optimization of vEB trees is to discard empty subtrees. This makes vEB trees quite compact when they contain many elements, because no subtrees are created until something needs to be added to them. Initially, each element added creates about  $\log(m)$  new trees containing about  $m/2$  pointers all together. As the tree grows, more and more subtrees are reused, especially the larger ones. In a full tree of  $2^m$  elements, only  $O(2^m)$  space is used. Moreover, unlike a binary search tree, most of this space is being used to store data: even for billions of elements, the pointers in a full vEB tree number in the thousands.

However, for small trees the overhead associated with vEB trees is enormous: on the order of  $\sqrt{M}$ . This is one reason why they are not popular in practice. One way of addressing this limitation is to use only a fixed number of bits per level, which results in a **trie**. Alternatively, each table may be replaced by a **hash table**, reducing the space to  $O(n)$  (where *n* is the number of elements stored in the data structure) at the expense of making the data structure randomized. Other structures, including **y-fast tries** and **x-fast tries** have been proposed that have comparable update and query times and also use randomized hash tables to reduce the space to  $O(n)$  or  $O(n \log M)$ .

References [\[edit\]](#)

1.

▲


Peter van Emde Boas

*Preserving order in a forest in less than logarithmic time (Proceedings of the 16th Annual Symposium on Foundations of Computer Science 10: 75-84, 1975)*

2.

▲

Gudmund Skovbjerg Frandsen

*Dynamic algorithms: Course notes on van Emde Boas trees (PDF)*  (University of Aarhus, Department of Computer Science)

3.

▲


Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

*Introduction to Algorithms*, Third Edition. MIT Press, 2009. ISBN 978-0-262-53305-8. Chapter 20: The van Emde Boas tree, pp. 531–560.

4.



▲

Rex, A.



*"Determining the space complexity of van Emde Boas trees"* . Retrieved 2011-05-27.

Further reading [\[edit\]](#)

•

Erik Demaine, Shantonu Sen, and Jeff Lindy. Massachusetts Institute of Technology. 6.897: Advanced Data Structures (Spring 2003). *Lecture 1 notes: Fixed-universe successor problem, van Emde Boas* . *Lecture 2 notes: More van Emde Boas, ...* .

•

van Emde Boas, P.; Kaas, R.; Zijlstra, E. (1976). "Design and implementation of an efficient priority queue" . *Mathematical Systems Theory* **10**: 99–127. doi:10.1007/BF01683268 .

| v · t · e  | Tree data structures   | <span>[hide]</span> |
|--|--|---------------------|
| <b>Search trees</b><br>(dynamic sets/associative arrays) | 2–3 · 2–3–4 · AA · (a,b) · AVL · B · B+ · B* · B <sup>x</sup> · (Optimal) Binary search · Dancing · HTree · Interval · Order statistic · (Left-leaning) Red-black · Scapegoat · Splay · T · Treap · UB · Weight-balanced |                     |
| <b>Heaps</b>   | Binary · Binomial · Fibonacci · Leftist · Pairing · Skew · <b>Van Emde Boas</b>  |                     |
| <b>Tries</b>   | Hash · Radix · Suffix · Ternary search · X-fast · Y-fast   |                     |
| <b>Spatial data partitioning trees</b>                   | BK · BSP · Cartesian · Hilbert R · <i>k</i> -d (implicit <i>k</i> -d) · M · Metric · MMP · Octree · Priority R · Quad · R · R+ · R* · Segment · VP · X   |                     |
| <b>Other trees</b>                                       | Cover · Exponential · Fenwick · Finger · Fusion · Hash calendar · iDistance · K-ary · Left-child right-sibling · Link/cut · Log-structured merge · Mørkle · PQ · Range · SPQR · Top                                      |                     |

Categories: [Priority queues](#) | [Search trees](#)