# Lecture 15

# van Emde Boas Data Structure

*Supplemental reading in CLRS: Chapter 20*

Given a fixed integer $n$, what is the best way to represent a subset $S$ of $\{0,\ldots,n-1\}$ in memory, assuming that space is not a concern? The simplest way is to use an $n$-bit array $A$, setting $A[i]=1$ if and only if $i \in S$, as we saw in Lecture 10. This solution gives $O(1)$ running time for insertions, deletions, and lookups (i.e., testing whether a given number $x$ is in $S$).[1] What if we want our data structure to support more operations, though? Perhaps we want to be able not just to insert, delete and lookup, but also to find the minimum and maximum elements of $S$. Or, given some element $x \in S$, we may want to find the *successor* and *predecessor* of $x$, which are the smallest element of $S$ that is greater than $x$ and the largest element of $S$ that is less than $x$, respectively. On a bit array, these operations would all take time $\Theta(|S|)$ in the worst case, as we might need to examine all the elements of $S$. The **van Emde Boas (vEB) data structure** is a clever alternative solution which outperforms a bit array for this purpose:

| Operation | Bit array | van Emde Boas |
|---|---|---|
| INSERT | $O(1)$ | $\Theta(\lg\lg n)$ |
| DELETE | $O(1)$ | $\Theta(\lg\lg n)$ |
| LOOKUP | $O(1)$ | $\Theta(\lg\lg n)$ |
| MAXIMUM, MINIMUM | $\Theta(n)$ | $O(1)$ |
| SUCCESSOR, PREDECESSOR | $\Theta(n)$ | $\Theta(\lg\lg n)$ |

We will not discuss the implementation of SUCCESSOR and PREDECESSOR; those will be left to recitation.

## 15.1 Analogy: The Two-Coconut Problem

The following riddle nicely illustrates the idea of the van Emde Boas data structure. I am somewhat embarrassed to give the riddle because it shows a complete misunderstanding of materials science and botany, but it is a standard example and I can't think of a better one.

**Problem 15.1.** There exists some unknown integer $k$ between 1 and 100 (or in general, between 1 and some large integer $n$) such that, whenever a coconut is dropped from a height of $k$ inches or

[1] This performance really is unbeatable, even for small $n$. Each operation on the bit array requires only a single memory access.

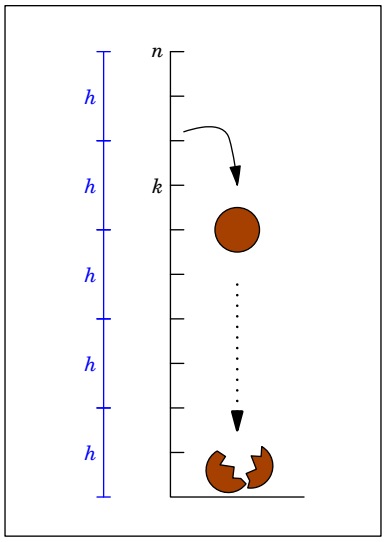


**Figure 15.1.** One strategy for the two-coconut problem is to divide the integers $\{1,\ldots,n\}$ into blocks of size $h$, and then use the first coconut to figure out which block $k$ is in. Once the first coconut breaks, it takes at most $h-1$ more drops to find the value of $k$.

more, the coconut will crack, and whenever a coconut is dropped from a height of less than $k$ inches, the coconut will be completely undamaged and it will be as if we had not dropped the coconut at all. (Thus, we could drop the coconut from a height of $k-1$ inches a million times and nothing would happen.) Our goal is to find $k$ with as few drops as possible, given a certain number of test coconuts which cannot be reused after they crack. If we have one coconut, then clearly we must first try 1 inch, then 2 inches, and so on. The riddle asks, what is the best way to proceed if we have two coconuts?

An approximate answer to the riddle is given by the following strategy: Divide the $n$-inch range into $b$ blocks of height $h$ each (we will choose $b$ and $h$ later; see Figure 15.1). Drop the first coconut from height $h$, then from height $2h$, and so on, until it cracks. Say the first coconut cracks at height $b_0 h$. Then drop the second coconut from heights $(b_0-1)h+1$, $(b_0-1)h+2$, and so on, until it cracks. This method requires at most $b+h-1=\frac{n}{h}+h-1$ drops, which is minimized when $b=h=\sqrt{n}$.

Notice that, once the first coconut cracks, our problem becomes identical to the one-coconut version (except that instead of looking for a number between 1 and $n$, we are looking for a number between $(b_0-1)h+1$ and $b_0 h-1$). Similarly, if we started with three coconuts instead of two, then it would be a good idea to divide the range $\{1,\ldots,n\}$ into equally-sized blocks and execute the solution to the two-cononut problem once the first coconut cracked.

**Exercise 15.1.** *If we use the above strategy to solve the three-coconut problem, what size should we choose for the blocks? (Hint: it is not $\sqrt{n}$.)*

## 15.2 Implementation: A Recursive Data Structure

Just as in the two-coconut problem, the van Emde Boas data structure divides the range $\{0,\ldots,n-1\}$ into blocks of size $\sqrt{n}$, which we call *clusters*. Each cluster is itself a vEB structure of size $\sqrt{n}$. In addition, there is a "summary" structure that keeps track of which clusters are nonempty (see Figure

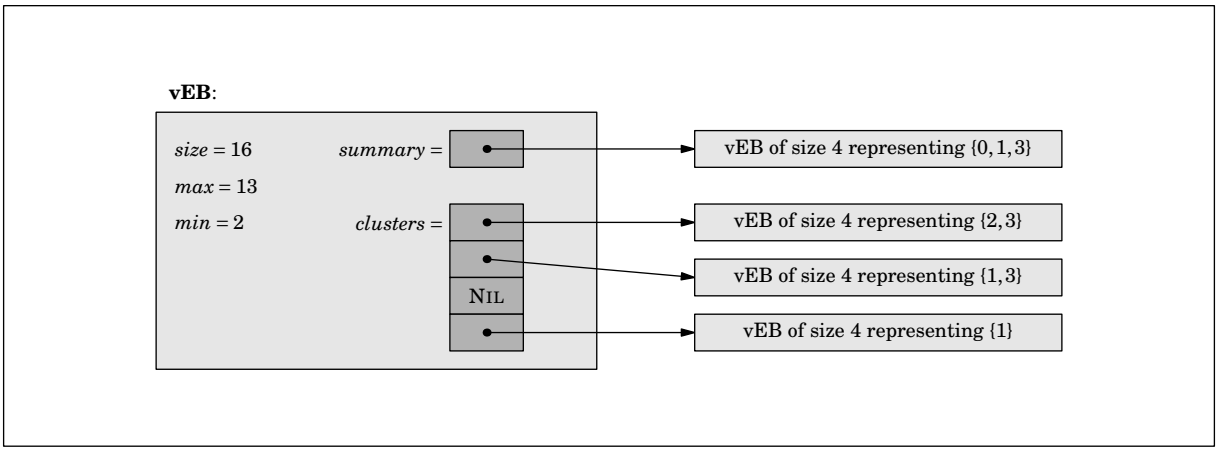


**Figure 15.2.** A vEB structure of size 16 representing the set $\{2,3,5,7,13\} \subseteq \{0,\ldots,15\}$.

15.2). The summary structure is analogous to the first coconut, which told us what block $k$ had to lie in.

### 15.2.1 Crucial implementation detail

The following implementation detail, which may seem unimportant at first, is actually crucial to the performance of the van Emde Boas data structure:

> Do not store the minimum and maximum elements in clusters. Instead, store them as separate data fields.

Thus, the data fields of a vEB structure $V$ of size $n$ are as follows:

- $V.size$ – the size of $V$, namely $n$
- $V.max$ – the maximum element of $V$, or NIL if $V$ is empty
- $V.min$ – the minimum element of $V$, or NIL if $V$ is empty
- $V.clusters$ – an array of size $\sqrt{n}$ which stores the clusters. For performance reasons, the value stored in each entry of $V.clusters$ will initially be NIL; we will wait to build each cluster until we have to insert something into it.
- $V.summary$ – a van Emde Boas structure of size $\sqrt{n}$ that keeps track of which clusters are nonempty. As with the entries of $V.clusters$, we initially set $V.summary \leftarrow$ NIL; we do not build the recursive van Emde Boas structure referenced by $V.summary$ until we have to insert something into it (i.e., until the first time we create a cluster for $V$).

### 15.2.2 Insertions

To simplify the exposition, in this lecture we use a model of computation in which it takes constant time to initialize any array (setting all entries equal to NIL), no matter how big the array is.[2] Thus,

[2] Of course, this is cheating; real computers need an initialization time that depends on the size of the array. Still, there are use cases for which this assumption is warranted. We can preload our vEB structure by creating all possible vEB

it takes constant time to create an empty van Emde Boas structure, and it also takes constant time to insert the first element into a van Emde Boas structure:

**Algorithm:** vEB-FIRST-INSERTION($V, x$)
1 $V.min \leftarrow x$
2 $V.max \leftarrow x$

Using this fact, we will show that the procedure $V$.INSERT($x$) has only one non–constant-time step. Say $V.clusters[i]$ is the cluster corresponding to $x$ (for example, if $n = 100$ and $x = 64$, then $i = 6$). Then:

- If $V.clusters[i]$ is NIL, then it takes constant time to create a vEB structure containing only the element corresponding to $x$. (For example, if $n = 100$ and $x = 64$, then it takes constant time to create a vEB structure of size 10 containing only 4.) We update $V.clusters[i]$ to point to this new vEB structure. Thus, the only non–constant-time operation we have to perform is to update $V.summary$ to reflect the fact that $V.clusters[i]$ is now nonempty.
- If $V.clusters[i]$ is empty[3] (we can check this by checking whether $V.clusters[i].min = \text{NIL}$), then it takes constant time to insert the appropriate entry into $V.clusters[i]$. So again, the only non–constant-time operation we have to perform is to update $V.summary$ to reflect the fact that $V.clusters[i]$ is now nonempty.
- If $V.clusters[i]$ is nonempty, then we have to make the recursive call $V.clusters[i]$.INSERT($x$). However, we do not need to make any changes to $V.summary$.

In each case, we find that the running time $T$ of INSERT satisfies the recurrence

$$T(n) = T\left(\sqrt{n}\right) + O(1). \tag{15.1}$$

As we will see in §15.3 below, the solution to this recurrence is $T_{\text{INSERT}} = \Theta(\lg\lg n)$.

### 15.2.3 Deletions

Similarly, the procedure $V$.DELETE($x$) requires only one recursive call. To see this, let $i$ be as above. Then:

- First suppose $V$ has no nonempty clusters (we can check this by checking whether $V.summary$ is either NIL or empty; the latter happens when $V.summary.min = \text{NIL}$).
  - If $x$ is the only element of $V$, then we simply set $V.min \leftarrow V.max \leftarrow \text{NIL}$. Thus, *deleting the only element of a single-element vEB structure takes constant time*; we will use this fact later.
  - Otherwise, $V$ contains only two elements including $x$. Let $y$ be the other element of $V$. If $x = V.min$, then $y = V.max$ and we set $V.min \leftarrow y$. If $x = V.max$, then $y = V.min$ and we set $V.max \leftarrow y$.

substructures up front rather than using NIL for the empty ones. This way, after an initial $O(n)$ preloading time, array initialization never becomes an issue. Another possibility is to use a dynamically resized hash table for $V.clusters$ rather than an array. Each of these possible improvements has its share of extra details that must be addressed in the running time analysis; we have chosen to ignore initialization time for the sake of brevity and readability.

[3] This would happen if $V.clusters[i]$ was once nonempty, but became empty due to deletions.

- Next suppose $V$ has some nonempty clusters.
  - If $x = V.min$, then we will need to update $V.min$. The new minimum takes only constant time to calculate, though: it is $y$, where

    $$y \leftarrow V.clusters[V.summary.min].min;$$

    in other words, it's the smallest element in the lowest nonempty cluster of $V$. After making this update, we need to make a recursive call to

    $$V.clusters[V.summary.min].\textsc{Delete}(y)$$

    to remove the new value of $V.min$ from its cluster. At this point, there are two possibilities:
    * $y$ is not the only element in its cluster. Then $V.summary$ does not need to be updated.
    * $y$ is the only element in its cluster. Then we must make a recursive call to

      $$V.summary.\textsc{Delete}(V.summary.min)$$

      to reflect the fact that $y$'s cluster is now empty. However, since $y$ was the only element in its cluster, it took constant time to remove $y$ from its cluster: as we said above, it takes only constant time to remove the last element from a one-element vEB structure.

    Either way, there is only one non–constant-time step in $V.\textsc{Delete}$: a recursive call to $\textsc{Delete}$ on a vEB structure of size $\sqrt{n}$.
  - By entirely the same argument (perhaps in mirror-image), we find that the case $x = V.max$ has identical running time to the case $x = V.min$.
  - If $x$ is neither $V.min$ nor $V.max$, then we must delete $x$ from its cluster and, if this causes $x$'s cluster to become empty, make a recursive call to $V.summary.\textsc{Delete}$ to reflect this update. As above, the recursive call to $V.summary.\textsc{Delete}$ will only happen when the deletion of $x$ from its cluster took constant time.

In each case, the only non–constant-time step in $\textsc{Delete}$ is a single recursive call to $\textsc{Delete}$ on a vEB structure of size $\sqrt{n}$. Thus, the running time $T$ for $\textsc{Delete}$ satisfies (15.1), which we repeat here for convenience:

$$T(n) = T\left(\sqrt{n}\right) + O(1). \qquad \text{(copy of 15.1)}$$

Again, the solution is $T_{\textsc{Delete}} = \Theta(\lg\lg n)$.

### 15.2.4 Lookups

Finally, we consider the operation $V.\textsc{Lookup}(x)$, which returns $\textsc{True}$ or $\textsc{False}$ according as $x$ is or is not in $V$. The implementation is easy: First we check whether $x = V.min$, then whether $x = V.max$. If neither of these is true, then we recursively call $\textsc{Lookup}$ on the cluster corresponding to $x$. Thus, the running time $T$ of $\textsc{Lookup}$ satisfies (15.1), and we have $T_{\textsc{Lookup}} = \Theta(\lg\lg n)$.

**Exercise 15.2.** *Go through this section again, circling each step or claim that relies on the decision not to store $V.min$ and $V.max$ in clusters. Be careful—there may be more things to circle than you think!*

## 15.3 Solving the Recurrence

As promised, we now solve the recurrence (15.1), which we repeat here for convenience:

$$T(n) = T\left(\sqrt{n}\right) + O(1). \qquad \text{(copy of 15.1)}$$

### 15.3.1 Base case

Before we begin, it's important to note that we have to lay down a *base case* at which recursive structures stop occurring. Mathematically this is necessary because one often uses induction to prove solutions to recurrences. From an implementation standpoint, the need for a base case is obvious: how could a vEB structure of size 2 make good use of smaller vEB substructures? So we will lay down $n = 2$ as our base case, in which we simply take $V$ to be an array of two bits.

### 15.3.2 Solving by descent

The equation (15.1) means that we start on a structure of size $n$, then pass to a structure of size $\sqrt{n} = n^{1/2}$, then to a structure of size $\sqrt{\sqrt{n}} = n^{1/4}$, and so on, spending a constant amount of time at each level of recursion. So the total running time should be proportional to the number of levels of recursion before arriving at our base case, which is the number $\ell$ such that $n^{1/2^\ell} = 2$. Solving for $\ell$, we find

$$\ell = \lg\lg n.$$

Thus $T(n) = \Theta(\lg\lg n)$.

### 15.3.3 Solving by substitution

Another way to solve the recurrence is to make a substitution which reduces it to a recurrence that we already know how to solve. Let

$$T'(m) = T\left(2^m\right).$$

Taking $m = \lg n$, (15.1) can be rewritten as

$$T'(m) = T'(m/2) + O(1),$$

which we know to have solution $T'(m) = \Theta(\lg m)$. Substituting back $n = 2^m$, we get

$$T(n) = \Theta(\lg\lg n).$$

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012