Overview

A van Emde Boas tree (vEB) is a data-structure that supports operations SEARCH, INSERT, DELETE, SUCCESSOR (and several others that are immediate once these are implemented), all in worst-case time $O(\log \log u)$ under the assumption that all keys come from the set $\{0, 1, \ldots, u-1\}$, which we call the *universe*.

We focus only on storing keys (and do not discuss storing values), which is a reasonable simplification since the most common uses of vEB do not use satellite data. Similarly, assume all keys are distinct, as duplicate keys do not fit in with the purpose of vEB.

Motivation

If u is polynomial in n (a realistic assumption), all operations take $O(\log \log n)$ time, an exponential improvement over binary search trees.

The fast SUCCESSOR operation is perhaps the biggest motivation for vEB. There are many applications, for instance network routing, where given some x, we want to find the first key in our data structure that is at least x.

Developing vEB

A bit vector version

Suppose we are only interested in inserting and removing keys. For this we maintain an array $A[0..u-1]$ of u bits, with the property that

$$A[x] = 1 \iff x \text{ is in the set.}$$

To insert/delete x, simply set/clear $A[x]$, in constant time. The problem is SUCCESSOR(x) takes $\Theta(u)$ time: all we can do is linearly search through A starting at x.

The key insight to improving SUCCESSOR is splitting the array A into *clusters*. The intuition is, we can really speed up SUCCESSOR, by skipping large clusters of the array that do not contain any elements.

In this example, if we are searching for the successor of $x = 3$, then we may skip over all of $A[4..7]$ since it does not contain any keys. We need some way of knowing whether a particular interval in A, contains a key: we do this by *superimposing* a binary tree structure.

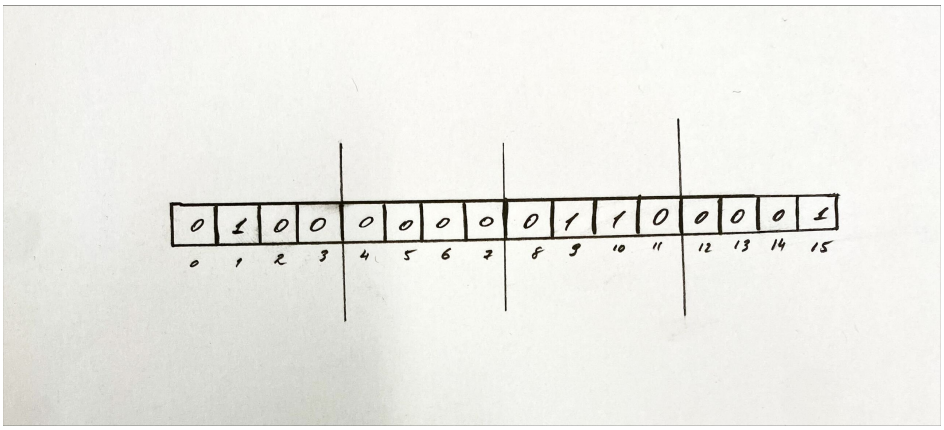

Figure 1: A divided into clusters.

The entries of the bit vector become the leaves of the binary tree, and each internal node contains a 1 if and only if any leaf in its subtree contains a 1. We note that we can compute the value at each internal node with bitwise-OR of the two children.

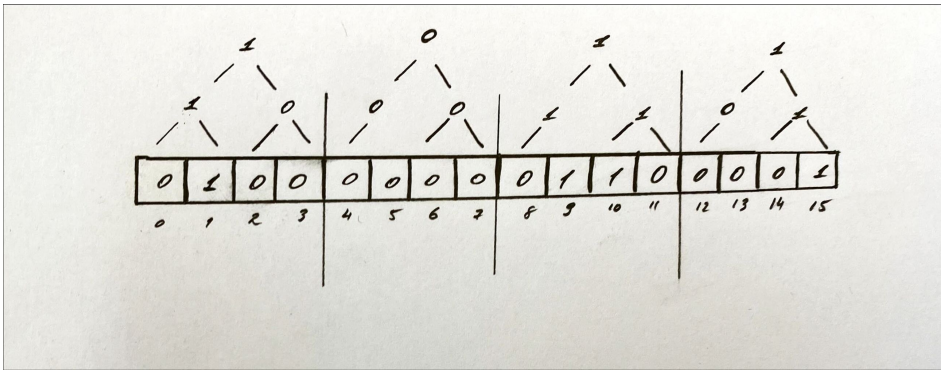

Figure 2: Superimposing a binary tree.

Now SUCCESSOR(3) can skip over the entire cluster $A[4..7]$. More generally, to find the successor of x: (1) we check for any 1s to the right of x in x's cluster; if there are none, (2) we go through the root nodes of clusters to the right of x, stop at the first one that has a 1, and then (3) search that node's cluster from left to right (see Figure 3).

INSERT(x) still takes constant time, set $A[x]$ and set the root node of x's cluster (note: we ignore other internal nodes). DELETE becomes less obvious, we will discuss it later.

We will store the root nodes of each cluster in an array called the *summary* vector. The question still stands, how big should we make the clusters? If we make them bigger, SUCCESSOR spends less time in *summary*, but looking through an individual cluster

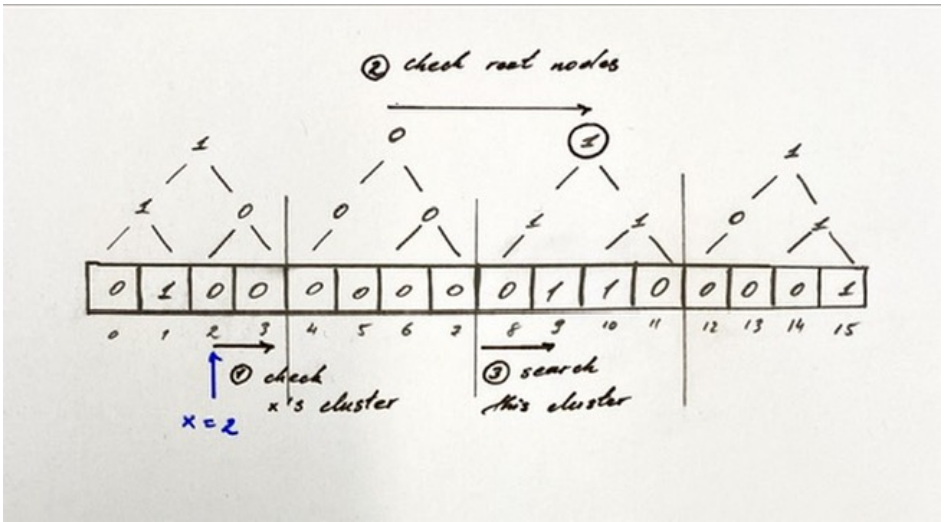

Figure 3: Example of SUCCESSOR with $x = 2$.

takes longer. The same trade-off stands for making clusters smaller. Since the number of clusters and cluster size multiply to u, we make them both $\sqrt{u}$ (as with interpolation search).

Now SUCCESSOR takes $\Theta(\sqrt{u})$, which still is not the desired runtime. The key insight for speeding it up to $\Theta(\log \log u)$ is using vEB recursively.[1]

Algorithm 1: *insert*(x)

1 $A[x] = 1$; $summary[x] = 1$

Algorithm 2: *successor*(x)

1 look in x's cluster;
2 look for next 1 in *summary*;
3 look for first 1 in that cluster

The trick is that we will **recursively represent each of the clusters as a vEB**.

> **Exercise 1.** SUCCESSOR(x) in vEB, in contrast to that in BST, allows x to not be in the set. Give an algorithm to find the successor of x in a BST that works even if x is not stored in the tree.

[1] The classic recurrence that resolves to $\Theta(\log \log u)$ is $T(u) = T(\sqrt{u}) + \Theta(1)$. Our goal now is to make worst-case times of INSERT and SUCCESSOR satisfy this recurrence.

A recursive structure

Looking at the bit vector implementation, a given value x resides in cluster number $\lfloor x/\sqrt{u} \rfloor$. Within its cluster x appears in position $x\%\sqrt{u}$. It is convenient to define helper functions to abstract away this arithmetic:

$$\begin{aligned} hi(x) &:= \lfloor x/\sqrt{u} \rfloor \\ lo(x) &:= x \mod \sqrt{u} \\ index(i,j) &:= i\sqrt{u} + j \end{aligned}$$

The reason for this naming is that if we view x as a $\log u$-bit integer, then the cluster of x is given by the most significant $(\log u)/2$ bits of x; $lo(x)$ is named similarly. Finally, we have the identity $x = index(hi(x), lo(x))$.

Now we are ready to define our vEB recursively: This is not yet the final version but it is almost it. [2]

> A size-u vEB V, for $u \geq 3$, consists of two parts:
>
> an array $V.cluster$ of size-$\sqrt{u}$ vEB's: this is the array of all the clusters, each of which is a vEB. This array has size $\sqrt{u}$.
>
> a size-$\sqrt{u}$ vEB $V.summary$: this is the summary structure. We maintain $V.summary[i]= 1$ if and only if cluster i of V is not empty.
>
> It also has a field u, storing its size.

To insert x into V, we insert into the appropriate cluster, namely $hi(x)$, and then indicate in the summary structure that the cluster $hi(x)$ is not empty.

Algorithm 3: $insert(V, x)$

```
1 insert( V.cluster[hi(x)], lo(x));
2 insert( V.summary, hi(x))
```

For SUCCESSOR, we mimic Algorithm 2 exactly:[3] (see next page)

Did we achieve a $\Theta(\log logu)$ time SUCCESSOR? Unfortunately, SUCCESSOR calls itself recursively three times in the worst-case and we still do not have the nice recurrence $T(u) = T(\sqrt{u}) + 1 \in \Theta(\log \log u)$.

It is very easy to get rid of the recursive call on line 4. The successor of $-\infty$ in a cluster is simply the minimum element. We augment our vEB to also store the minimum element; so line 4 becomes constant time.

[2] [CLRS] calls this recursive data structure a *proto van Emde Boas structure*.

[3] We omit the base case for now. It is trivial to handle, and would just be a distraction for now.

Algorithm 4: $successor(V, x)$

```
1 i = hi(x); j = successor(V.cluster[i], lo(x)) // look in x's cluster
2 if j == ∞ then
      /* did not find in x's cluster                                  */
3     i = successor(V.summary, i) // next 1 in summary
      /* if i == ∞, can return here (ommitted for clarity)            */
4     j = successor(V.cluster[i], −∞) // first 1 in next cluster
5 end
6 return index(i, j)
```

What about line 3? We will only make this recursive call if x is greater than any element in its cluster; or in other words, if x is *greater than the maximum* element of that cluster. With another augmentation (storing the maximum element) we get the following $\Theta(\log \log u)$ time implementation:[4]

Algorithm 5: $successor(V, x)$

```
1 i = hi(x) // x's cluster
2 if x > V.max then
      // successor of x cannot be in this cluster
3     i = successor(V.summary, i) // next 1 in summary
4     j = V.cluster[i].min // first 1 in next cluster
5 else j = successor(V.cluster[i], lo(x)) ;
6 return index(i, j)
```

Of course, we modify INSERT to maintain the minimum and maximum fields:

Algorithm 6: $insert(V, x)$

```
1 if x < V.min then V.min = x;
2 if x > V.max then V.max = x;
  /* ...  and the rest is as before                                  */
```

This is still slower than $\Theta(\log \log u)$.

Exercise 2.[5] Give a tight bound on the runtime of INSERT.

Exercise 3.[6] Give an implementation of DELETE.

[4] Now a size-2 vEB has the three fields: u, min, and max.

[5] Perhaps resolving such a recurrence is a bit harder than what is expected in CS240E.

[6] Worst-case time $\Theta(\sqrt{u})$ is all we want here.

A faster Insert

Again the problem with INSERT(x) is that it makes two recursive calls. We need to get it down to just one. For convenience this is our current implementation, which we need to improve:

Algorithm 7: *insert*(V, x)

```
1 if x < V.min then V.min = x;
2 if x > V.max then V.max = x;
  // and now as before:
3 insert(V.cluster[hi(x)], lo(x));
4 insert(V.summary, hi(x))
```

The trick now is: if the cluster where x will end up **already has an element, then we do not need to update the summary**. So on the first insert into a cluster, we will **only update the summary**, and store x *lazily*. That is to say, we will not do the recursive call on line 3, and only store x as the minimum (and the maximum) element of V. So now the first INSERT makes only one recursive call.

What about a future INSERT? We actually already handled this case! We **no longer need to update the summary**, so we there is just one recursive call here too.

Algorithm 8: *insert*(V, x)

```
1 if V.min is ∅ then V.min = V.max = x; return
2 if x < V.min then swap x with V.min
3 if x > V.max then V.max = x
4 if V.cluster[hi(x)].min is ∅ then insert(V.summary, hi(x))
5 insert(V.cluster[hi(x)], lo(x))
```

Line 1 is the first INSERT, when V is empty. We know that V is empty if and only if $V.min$ is null.

On line 2, we know V is not empty, and thus some element will be inserted into a cluster of V. But this element might not necessarily be the x passed as an argument. If $x < min$ then x needs to become the new min. Since we do not want to lose the original min, we insert *it* into a cluster of V.

Line 3 is just some maintenance.

Line 4 there was previously nothing in the cluster $hi(x)$, so we need to update the summary structure.

Line 4 is exactly line 3 from before. We always insert into x's cluster.

It seems that in the worst-case there are two recursive calls, but we can only have one. How is this better than the old algorithm? We look closely at lines 5-6:

If there was already something in the cluster $hi(x)$, then we do not make the call on line 6; so there is only one call on line 7 (and the runtime is good).

If there was nothing in the cluster $hi(x)$, then the recursive call on line 6 actually exits at line 1! So it takes constant time. So again, we only make one meaningful recursive call to INSERT.

So indeed this implementation has $\Theta(\log \log u)$ worst-case time.

Exercise 4. With this implementation of INSERT, our Algorithm 5 for SUCCESSOR becomes incorrect; though the changes required are minimal. Adapt the SUCCESSOR to this implementation of INSERT.

For completeness, we give the solution to this exercise:

Algorithm 9: *successor*(V, x)

```
1 if V.min is not ∅ and x < V.min then return V.min
  // rest is as before
2 ...
```

We cannot rely on a recursive substructure to store the minimum. However, it is really easy to check if the minimum is the successor of x. It is the successor of x if and only if x is less than it.

Implementing Delete

The intuition for our implementation of DELETE is to look at INSERT, and "reverse" its every possible execution. This is the overview:

Algorithm 10: *delete*(V, x)

```
1 if x == V.min then /* ...  we'll fill this in                    */
2 delete(V.cluster[hi(x)], lo(x)) /* always do this, as insert:line5 */
3 if V.cluster[hi(x)].min is ∅ then delete(V.summary, hi(x))
4 if x == V.min then /* ...  we'll fill this in too                */
```

As with INSERT, the recursive call on line 3 is hard to avoid. If $V.cluster[hi(x)]$ is empty, we need to delete $hi(x)$ from the summary. Our goal is to fill in line 1 so that

if $V.cluster[hi(x)]$ **becomes empty, the recursive call on line 2 is cheap.**[7] We fill in line 1:

Algorithm 11: *delete*(V, x)

```
1 if x == V.min then
2 |   i = V.summary.min
3 |   if i is ∅ then V.min = V.max = ∅; return
4 |   x = V.min = index(i, V.cluster[i].min)
5 end
6 delete(V.cluster[hi(x)], lo(x)) if V.cluster[hi(x)].min is ∅ then
    delete(V.summary, hi(x))
7 if x == V.max then /* ...  we'll fill this in too                */
```

The check on line 3 passes if and only if x was the last item in V. In this case, do not do any recursive calls. This gives us the cheap base case we were hoping for above.

Line 4 happens when we were deleting the minimum, but it was not the only item. So to remove x in this case, it suffices to override it with the new minimum element of V, namely

$$index(\underbrace{i}_{\text{first non-empty cluster by line 2}}, \underbrace{V.cluster[i].min}_{\text{min of that cluster}})$$

It remains to fill in line 7. The intuition here is to imagine we just want reset $V.max$ to the correct value.

Algorithm 12: *delete*(V, x)

```
1 if x == V.min then
2 |   i = V.summary.min
3 |   if i is ∅ then V.min = V.max = ∅; return
4 |   x = V.min = index(i, V.cluster[i].min)
5 end
6 delete(V.cluster[hi(x)], lo(x))
7 if V.cluster[hi(x)].min is ∅ then delete(V.summary, hi(x))
8 if x == V.max then
9 |   i = V.summary.max
10|   if i is ∅ then V.max = V.min
11|   else V.max = index(i, V.cluster[i].max)
12 end
```

Line 9 finds the last non-empty cluster. If it does not exist (line 10), then the maximum

[7] Notice how the last DELETE "reverses" everything done in the first INSERT.

element is the miminum element. Note that this works whether there is just one element in V, or there are no elements in V. In the latter case $V.max$ becomes $\varnothing$.

Otherwise, there is some non-empty cluster. Line 11 simply obtains its maximum elements.

Base cases

For completeness, we give the three operations, that handle all base cases carefully:

Algorithm 13: *successor*(V, x)

```
1  if V.min is not ∅ and x < V.min then return V.min;
2  if V.u == 2 then
3  |  if x == 0 and V.max == 1 then return 1;
4  |  else return ∅;
5  end
6  i = hi(x);
7  if lo(x) > V.cluster[i].max then
8  |  i = successor(V.summary, i) j = V.cluster[i].min
9  else  j = successor(V.cluster[i], lo(x)) ;
10 return index(i, j)
```

Algorithm 14: *insert*(V, x)

```
1  if V.min is ∅ then  V.min = V.max = x; return
2  if x < V.min then  swap x with V.min
3  if x > V.max then  V.max = x
4  if V.u > 2 then
5  |  if V.cluster[hi(x)].min is ∅ then  insert(V.summary, hi(x))
6  |  insert(V.cluster[hi(x)], lo(x))
7  end
```

Algorithm 15: $delete(V, x)$

```
1  if V.min == V.max then V.min = V.max = ∅; return
2  if V.u == 2 then
3  |   if x == V.min then V.min = 1
4  |   else V.min = 0
5  |   V.max = V.min
6  |   return
7  end
8  if x == V.min then
9  |   i = V.summary.min
10 |   if i is ∅ then V.min = V.max = ∅; return
11 |   x = V.min = index(i, V.cluster[i].min)
12 end
13 delete(V.cluster[hi(x)], lo(x))
14 if V.cluster[hi(x)].min is ∅ then delete(V.summary, hi(x))
15 if x == V.max then
16 |   i = V.summary.max
17 |   if i is ∅ then V.max = V.min
18 |   else V.max = index(i, V.cluster[i].max)
19 end
```

Other operations

The Minimum and Maximum are easy constant time operations because we store those values as fields in V:

Algorithm 16: $maximum(V, x)$

```
1  return V.max
```

Algorithm 17: $minimum(V, x)$

```
1  return V.min
```

Searching for an element is now done in $\Theta(\log \log u)$, using that $x = index(hi(x), lo(x))$.

Line 1 checks to see whether x is either the minimum or the maximum element. If it is not, and V can only contain two items (line 2), then x is not in V. Otherwise, we search in the appropriate cluster recursively.

The Predecessor operation is symmetric to Successor.

Algorithm 18: $search(V, x)$

```
1 if x == V.min or x == V.max then return true;
2 if V.u == 2 then return false;
3 return search( V.cluster[hi(x)], lo(x))
```

Algorithm 19: $predecessor(V, x)$

```
1 if V.max is not ∅ and x > V.max then return V.max;
2 if V.u == 2 then
3 |   if x == 1 and V.min == 0 then return 0;
4 |   else return ∅;
5 end
6 i = hi(x) if x < V.min then
7 |   i = predecessor( V.summary, i) j = V.cluster[i].max
8 else  j = predecessor( V.cluster[i], lo(x)) ;
9 return index(i, j)
```

References

[CLRS] Cormen T., Leiserson C., Rivert R., Stein C. *Introduction to Algorithms.* Third edition.

[ED] Demaine E. *Design and analysis of algorithms* lecture notes. Lecture 4.