1 Summary

2 vEB trees

2.1 Universe size

The vEB tree structure is used to store keys in a universe U of possible keys with size u. The size of the universe can be any exact power of 2. When $\sqrt{u}$ is not an integer, then u is an odd power of 2 say $u = 2^{2k+1}$for some $k > 0$, and we will divide the lg$u$ bits needed to store a universe key into two components - $\lceil (\lg u)/2 \rceil$ number of most significant bits and $\lfloor (\lg u)/2 \rfloor$ number of least significant bits.

For some notation, we define $2^{\lceil (\lg u)/2 \rceil}$ as the upper square root of u by $\sqrt[\uparrow]{u}$. Then we define$2^{\lfloor (\lg u)/2 \rfloor}$, the lower square root of u by the notation $\sqrt[\downarrow]{u}$. This has the effect that

$$\sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u} = 2^{\lceil (\lg u)/2 \rceil} \cdot 2^{\lfloor (\lg u)/2 \rfloor} = 2^{\lceil (\lg u)/2 \rceil + \lfloor (\lg u)/2 \rfloor} = 2^{\lg u} = u$$

and when u is an even power of u the we have $\sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u} = \sqrt{u}$ since taking the floor and ceil does nothing.

The point in vEB trees is to quickly perform operations on the universe keys by considering the bits that make up the keys in the universe. This is why we need to both combine the upper and lower half bits to a new number as well as extract the upper an lower half bits.

2.2 Helper functions

We define the following helper functions

$$\begin{aligned} high(x) &= \lfloor x / \sqrt[\downarrow]{u} \rfloor \\ low(x) &= x \mod \sqrt[\downarrow]{u} \\ index(x, y) &= x \sqrt[\downarrow]{u} + y \end{aligned}$$

The function $high(x)$ gives us the number of the upper half bits. More precisely we need $\lg u$ bits to represent a key, so we get the upper half $\lg u$ bits. Why? Bit-shifting right once is the same as dividing by 2 and removes the least significant bits. We want to remove about $(\lg u)/2$ bits. So how many times do we need to divide by 2? Exactly $\lceil (\lg u)/2 \rceil$ times. So we need to divide by $\sqrt[\downarrow]{u} = 2^{\lfloor (\lg u)/2 \rfloor}$ and floor the number to get the integer number.

The function $low(x)$ gives us the lower half $\lg u$ bits. How? When we take $\mod k$ of a number we get the remainder left after dividing by k.

The function $index(x, y)$ works by treating x as the upper half bits, by bit-shifting(multiplying by 2) left about $(\log u)/2$times and y as the lower half bits and returns a new number. We actually have that $index(high(x), low(x)) = x$ since it combines the lower half and upper half bits.

2.3 The tree structure

The vEB tree is a recursive tree structure and vEB(u) denotes a (sub)tree of universe size u. When $u > 2$ the tree has the following attributes

1. Summary pointer
   This is a pointer to another tree of size vEB($\sqrt[\uparrow]{u}$).

2. Universe size u

3. min that stores the minimum element in the tree

4. max that stores the maximum element in the stree

5. cluster array of $\sqrt[\uparrow]{u}$ pointers to vEB($\sqrt[\downarrow]{u}$) trees. Note the following

   (a) We have about $\sqrt{u}$ trees each with about $\sqrt{u}$ keys for a total of u keys stored.

   (b) We can determine cluster index of a universe key by $high(x)$ - this gives us the upper half bits

Important to note is that the element in *min* does not appear in any of the trees in the cluster. The keys stored by the tree is therefore $V.min$ plus the *min* of each the recursive $cluster[0, 1, ..., vEB(\sqrt[\downarrow]{u}) - 1]$ trees. The *max* attribute can appear in a cluster, but if the tree size is 1 then *min=max* then *max* will not appear in the tree.

For a base tree of size vEB(2) does not need the cluster array, since we can use *min* and *max* to determine the elements of the tree.

Note about figure 20.6 in CLRS. The tree vEB(16).min=2, and vEB(16).cluster[0].min=3. Furthermore vEB[16].cluser[0] contains two cluster points to two vEB(2) tree that both have nil points for the *min* and *max*. This means that the tree vEB(16).cluster[0] which is a vEB(4) tree only contain keys 2 and 3, since the *min* is never stored in the tree.

2.3.1 Clusters

To explain clusters, we have to understand the idea behind keys in a vEB tree. As we have confirmed, a universe size of u keys requires us to use $\lg u$ bits to store the keys. We can the split the bits of the keys into an upper and lower half with the $high, low$ utility functions. Using the upper half bits of keys gives us $(\lg u)/2$ bits we can use for clusters. How many numbers can we make with this many bits

$$2^{(\lg u)2} = (2^{\lg u})^{1/2} = u^{1/2} = \sqrt{u}$$

so we can for a universe size u, we have $\sqrt{u}$ different clusters. So each key $x \in u$ is given a cluster by its upper half bits. We can now use the lower half bits, given by $low(x)$, to differentiate between all the keys within a cluster, and we can store x in the cluster using the lower half bits of x. We can call these lower half bits the offset of x into its cluster. There must also be $\sqrt{u}$ lower offsets into a cluster.

Additionally, we can reconstruct any number x given its cluster number and the offset into the cluster using the $index$ function - it just merges the upper bits of the cluster number with the lower half bits of the offset.

Note however, that for universe sizes that are not an exact power of two, $u = 2^{2k+1}$they will be split into an upper square root number of clusters, and then we user lower square root bits as offsets into the clusters.

In short, clusters are vEB-tree that share the same upper half bits. Since they are vEB tree, they themselves have recursive vEB tree using the upper and lower half of its bits all the way down to a base case of 1 bits - this can only store two keys.

2.3.2 Summary

While the clusters are just vEB trees that share the same upper-half bits, using the lower half bits as keys/offset in the cluster, the summary is a vEB tree using the upper half bits.

A summary pointer is an attribute in a vEB tree of size u. Following the summary pointer gives us a tree of size $\sqrt[\uparrow]{u}$. We have also $\sqrt[\uparrow]{u}$ pointers in the cluster, so the summary in a tree of size u actually stores a summary of each cluster tree. But what is the summary? The min attribute is the minimum cluster num that is not empty, and max is similar for maximum. This structure is then recursive.

Lets say we have a vEB tree of size 16. Then we have $\sqrt{16} = 4$ clusters and thus $V.summary.u = 4$. This is a summary over the four clusters of the $V.u = 16$ vEB tree. The summary then has two base-case tree of size 2. The first of which might have min=1, max=1 saying that cluster 0 is empty. It might however still have a min attribute, since we do not store min elements.

2.4 vEB recurrence

The recurrence describing each of the recursive tree operations can be described by the recurrence

$$T(u) \le T(\sqrt[\uparrow]{u}) + O(1)$$

This can be solved in the following way. Let $m = \lg u$. Then we can write

$$\begin{aligned} T(u) = T(2^m) \le T(\sqrt[\uparrow]{u}) + O(1) = T(u) \le T(2^{\lceil (\lg u)/2 \rceil}) + O(1) \\ = T(2^{\lceil m/2 \rceil}) + O(1) \end{aligned}$$

and since $\lceil m/2 \rceil \le 2m/3$ for all $m \ge 2$ we get

$$T(2^m) \le T(2^{2m/3}) + O(1)$$

and if we let $S(m) = T(2^m)$ then we get

$$S(m) \le S(2m/3) + O(1)$$

and using part 2 of the mater theorem $T(n) = aT(n/b) + f(n)$

$$\text{if} f(n) = \Theta(n^{\log_b a}) \text{ then } T(n) = \Theta(n^{\log_b a} \lg n)$$

with $b = \frac{3}{2}$ and $a = 1$ we get that $n^{\log_{3/2} 1} = n^0 = 1$ so we do have $f(n) = O(1) = \Theta(1)$ which gives us solution

$$S(m) = \Theta(\lg m)$$

and substituting back $T(u) = T(2^m) = S(m)$ and $m = \lg u$ we get

$$T(u) = \lg \lg u$$

2.5 Maximum and minimum

These run in $O(1)$ time and are easy to implement since we just take the tree, V, and return $V.min$ or $V.max$.

2.6 vEB membership

The pseudo code looks like this

```
1. if x == V.min or x ==V.max
    (a) return ture
2. elif V.u==2
    (a) false
3. else return vEB-TREE-MEMBER(V.cluster[high(x), low(x))
```

How does it work?

Well if the key we are looking for is either the minimum or maximum, then we are done.

If we however have recursed to the base-case of depth 2, then hitting line 2 means they key was neither the *min* or the *max*, so the key cannot be there.

If we have not yet found the key, the recurse in one of the subtrees. How do we determine the subtree? We use the structural bit-pattern of the universe keys. We take the upper half bits and use these to figure out which of the $^{\uparrow}\sqrt[u]{u}$ different subtrees the key might be present in. Then we use the lower half bits to represent they key. This works since if the key is in the tree, then it is stored as *min* or *max* and the subtree number of the stored key is found by recursively stripping away the upper bits until we hit the base case, and on the way using the lower half bits as the key value.

Running time

Each element is stored using $\lg u$ bits,and if we search until we hit the base-case then we half the bits using $low(x)$ each time. We can at most half $\lg u$ bits $\lg u$ times, so it is $\lg \lg u$ running time.

2.7 Tree successor

Recall the successor of x is the smallest element larger than x.

```
1. if V.u == 2
    (a) if x == 0 and V.max == 0
        i. return 1
    (b) else return NIL
2. elseif V.min != NIL and x < V.min
    (a) return V.min
3. else max-low = vEB-TREE-MAXIMUM(V.cluser[high(x)])
    (a) if max-low != NIL and low(x) < max-low
        i. offset = vEB-TREE-SUCCESSOR(v.cluser[high(x)], low(x))
        ii. return index(high(x), offset)
    (b) else succ-cluster = vEB-TREE-SUCCESSOR(V.summary, high(x))
        i. if succ-cluster == NIL
            A. return NIL
        ii. else offset = vEB-TREE-Minimum(V.cluster[succ-cluster])
            A. return index(succ-cluster, offset)
```

The algorithm will now be explained in detail. Each bullet corresponds to a bullet in the algorithm.

1. This line indicates we have the base case
    (a) When the universe has size 2, we can only have $U = \{0, 1\}$. So the only element that has a successor is 0 whith successor 1. Therefore this case returns successor 1 if $x = 0$ and the max element is 1. Remember, if the max was 0, then $min = max = 0$ and so there is no successor.
    (b) Otherwise there can be no successor
2. If we are not in the base case, and the minimum element in the tree $V.min \neq NIL$ then we have a non-empty tree. If the minimum element of V is larger than x, then this is the smallest element larger than x and is therefore the successor.

3. Okay, so the successor is not immediately obvious from the $V.min$ attribute in the current tree. Therefore we first get the cluster number of x from $high(x)$. This is a vEB tree, so we can take the max element in this tree. This gives us the largest element in the cluster where x resides.

   (a) First we check if if $max - low$ is NIL, in which case the cluster of x is an empty tree (the successor is not the x cluster but another clusterl. The function $low(x)$ gives us the bits used to represent x in the recursive subtree $V.cluster[high(x)]$. $max - low$ is represented in as many bits as $low(x)$ so we can check whether $low(x) < max - low$. In which case we know that the successor of x is in its cluster

      i. Therefore we search for x with $low(x)$ bits in x cluster in this line

      ii. Since all elements in a cluster share the same upper half bits, we can use $high(x)$ as the upper bits for the found number. Offset is the "representation" of x in the lower half bits, so we can combine the upper half bits, $high(x)$, that are shared between x and $offset$ with the lower half bits given by $offset$ to get the successor and we build this new number with $index$ and return it

   (b) This case covers when the successor is not in the cluster of x. It must be a higher cluster than x, so we first have to figure out which cluster the successor is in. The $summary$ of the current tree V contains information of each $\sqrt[\uparrow]{u}$ clusters, so we can use this to find next nonempty cluster, since this will contain the successor. Let us strip off the upper half bits of x with $high(x)$ to find the cluster number of x. Now we can just recursively look for the successor to $high(x)$, since this gives us the next nonempty cluster.

      One thing to note, when we are in this case, then whatever "successor" cluster, that is the next nonempty cluster, we find, the smallest element in this cluster will be the successor.

      i. If $suc - cluster == NIL$ then there are either no more clusters after the current cluster or the rest of the clusters are all empty, so we must return NIL.

      ii. Otherwise, we found the next nonemptu cluster. We can now find the smallest element in this cluster by calling the minimum on $V.cluster[succ - cluster]$. This gives us the smallest number represented on $V.u/2$ bits.

         A. Since we know the cluster number of the successor and the "offset" (the representation of the number using $V.u/2$ bits) we have everything to build the successor using the $index$ helper function.

2.7.1 Running time

The only recursive calls are those of 3.i.a and 3.b. Each of them are exclusive, so we only do a single recursive call at each iteration. How many recursive calls can we do? Well each time we about half the $\lg u$ bits until a single bit remain and we hit the base case. How many times can we half the bits? $\lg$ the number of bits times

$$O(\lg \lg u)$$

2.8 The predecessor

Recall this is the largest element smaller than x. The pseudocode is almost identical to successor

```
1. if V.u == 2
    (a) if x == 1 and V.min = 0
        i. return 0
    (b) else return NIL
2. elseif V.max != NIL and x > V.max
    (a) return V.min
3. else min-low = vEB-TREE-MINIMUM(V.cluser[high(x)])
    (a) if min-low != NIL and low(x) > min-low
        i. offset = vEB-TREE-PREDECESSOR(v.cluster[high(x)], low(x))
        ii. return index(high(x), offset)
    (b) else pred-cluster = vEB-TREE-PREDECESSOR(V.summary, high(x))
        i. if pred-clust == NIL
            A. if V.min != NIL and x > V.min then return V.min
            B. else return NIL
        ii. else offset = vEB-TREE-MAXIMUM(V.cluster[pred-cluster])
            A. return index(pred-cluster, offset)
```

Each step of the algorithm is explained below

1. We are in the base case
    (a) We can have two elements in the base case since we only have (log$V.u)/2 = 1$ bits to represent the number. Therefore, if$x = 1$ in this tree then the predecessor can only be 0, requiring $V.min = 0$
        i. We return 0 in this case since this can be the only predecessor to $x = 1$
    (b) In case $x = 0$ or $V.min! = 0$ then there can be no predecessor
2. If the tree is not empty and x is larger than the max element of the tree then this element must be the largest number smaller than x.
    (a) Hence we can return $V.max$

3. We hit a case where the predecessor is not obvious from the current tree V. We start by finding the smallest element, $min - low$, in the cluster of x.
    (a) First we do a NIL check to ensure the cluster is not empty. In case the cluster of x is not empty, then this is an obvious place to find the predecessor of x, since these are all the numbers that have the same upper half bits and differ in their lower half bits. We check with $low(x)$ whether x represented with $V.u/2$ bits (the same bits as is in $min - low$) is larger than the smallest number in x's cluster by $low(x) > min - low$. In case it is, then we know the predecessor of x is within its clsuter
        i. We know the predecessor is in the cluster of x, so we recursively call the predecessor routing on the cluster of x using the lower half bits representing x. The results, $offset$, we get is the offset into the cluster the element resides - stated in other words it is the lower $V.u/2$ half bits of the predecessor. The other upper half bits are the cluster number
        ii. Therefore we combine the lower and upper half bits and return this
    (b) This case hits when we detect that the predecessor is not within the cluster, in other words, x is already the min in its vEB tree.
        i. We use the summary tree to find the first previous non-empty tree $pred - cluster$. This is where the predecessor must lie
            A. We check if there is no predecessor cluster by the NILcheck

               Since we do not represent the min in a tree, then if the predecessor is $V.min$ then it will not be in any tree, so the the cluster is NIL. We check this by ensuring $x > V.min$ and return $V.min$

               Otherwise there is no predecessor.
            B. Okay so there were a nonempty smaller numbered cluster. We can find the predecessor as the maximum in this cluster and assign it to $offset$

               Now we use the upper half bits of the cluster number again and the lower half offset to get the actual predecessor.

2.8.1 Running time

Symmetric to the successor analysis

2.9 Insert

Inserting into an empty tree is trivial since we just set the $V.min = V.max = x$ and the rest are NIL pointers. This holds because x will be the only element in the cluster. Inserting into a nonempty vEB tree is a bit more complicated. For the analysis it is assumed that x is not already in the tree. We can without changing the running time break this assumption by calling vEB-TREE-MEMER first in $O(\lg\lg u)$ time and if the element is present return the same tree and otherwise do the following routing

1. if $V.min == NIL$
   (a) vEB-EMPTY-TREE-INSERT(V,X)
2. else
   (a) if $x < V.min$
      i. exchange x with $V.min$ (insert x as $V.min$ and continue with $x = V.min$)
   (b) if $V.u > 2$
      i. if vEB-TREE-MINIMUM($V.clusters[high(x)]$) $== NIL$
         A. vEB-TREE-INSERT($V.summary, high(x)$)
         B. vEB-EMPTY-TREE-INSERT($V.cluster[high(x)], low(x)$)
      ii. else vEB-TREE-INSERT($V.cluster[high(x)], low(x)$)
   (c) if $x > V.max$
      i. $V.max = x$

Now the details of the algorithm is explained

1. Inserting into the empty tree is trivial
2. Inserting into a nonempty tree is a bit harder
   (a) If the element to insert is smaller than the current smallest element, then we need to make $V.min = x$ and the old $V.min$ needs to be x that is to be inserted. We continue the algorithm afterwards
   (b) This case covers when we are not in the base case
      i. We want to check whether the cluster to which x belongs has a minimum - in other words, whether it is empty. **If** it is empty we need to do two things
         A. Insert the key into the current trees summary. This can take $O(\lg\lg u)$ time.
         B. Do the insert into the empty tree in $O(1)$ time.
      ii. Okay so the cluster to which x belongs is not empty. Therefore we recursively try to insert x into the cluster in which it belongs. How? Again, all keys in the cluster share the same upper half bits, so we find the cluster by $high(x)$ and use $lower(x)$ as an offset into the cluster, or as to represent x in the cluster. The recursive call might go on into an empty tree is found in 1 or in 2.b.i or in 2.b.iii

iii. This case is hit when we need to update the max of a tree. We can also hit this case when we insert the key in a base-case tree. This works even if we insert the new min into the base case tree since line 2.a handles exchanging roles so we can insert the key as the max.

2.9.1 Running time

We only do constant time operations or 1 recursive call stripping away half the bits to work on. Making the running-time $O(\lg \lg u)$.

2.10 Delete

Here is the delete Algorithm. It assumes x is already in the tree. Again, we can overcome this restriction by just using the membership operation and it will not break the asymptotic runtime.

```
1. If $V.min == V.max$
   (a) $V.min = NIL$
   (b) $V.max = NIL$
2. elseif $V.u = 2$
   (a) if $x = 0$
       i. $V.min = 1$
   (b) else $V.min = 0$
   (c) $V.max = V.min$
3. else
   (a) if $x == V.min$
       i. $first - cluster$ =vEV-TREE-MINIMUM($V.summary$)
       ii. $x = index(first-cluster$,vEB-TREE-MINIMUM($v.cluset[first-cluster]$)
       iii. $V.min$ =x
   (b) vEB-TREE-DELETE($V.cluster[high(x)], low(x)$)
   (c) if vEB-TREE-MINIMUM($V.cluster[high(x)]$) $== NIL$
       i. vEB-TREE-DELETE($V.summary, high(x)$)
       ii. if $x == V.max$
           A. $summary - max$ =vEB-TREE-MAXIMUM($V.summary$)
           B. if $summary - max == NIL$
              then $V.max = V.man$
```

```
            C. else
               $V.max$ =index($summary - max$,
               vEB-TREE-MAXIMUM($v.cluster[summary - max]$)
    (d) elseif $x == V.max$
          i. $V.max$= index($high(x)$,
             vEB-TREE-MAXIMUM($V.cluster[high(x)]$))
```

Now each step is explained in detail

1. This tests whether the tree contains only 1 element. This can easily be deleted by replacing the max and min attributes with NIL pointers.
2. Else if we hit a base-case tree then we know, due to the previous case, that it has two different elements. Depending on whether we delete the min or max we have to update the attributs properly
   - (a) Deleting the min
     - i. Do so by setting the min to the max
   - (b) Otherwise we delete the max, so we just update $V.min = 0$ so the next case happens for both cases
   - (c) Update the $V.max$ and $V.min$ to be equal.
3. Otherwise we are not in a base case and a tree with a single element.
   - (a) We start by testing whether the key to delete is the minimum key of the tree
     - i. In case it is we find the cluster which belongs to the minimum key, $first - cluster$. We can use the summary tree here, since the minimum in the summary cluster gives us the lowest nonempty cluster number.