Lecture 12: van Emde Boas trees

Supports dynamic set operations in $O(\log\log u)$ time when elements have values from $\{0, 1, \ldots, u-1\}$, its *universe*.

Direct addressing: We record the element values in a bit vector $A[0..u-1]$, where $A[x] = 1$ iff the value x is in the set. INSERT, DELETE, and MEMBER operations can be performed in $O(1)$ time, but MINIMUM, MAXIMUM, SUCCESSOR, and PREDECESSOR may take $\Theta(u)$ time.

Superimpose a binary tree structure on top of the bit vector. Since the height is $\log u$ and each operation makes at most one pass up the tree and one pass down, the operation time is $O(\log u)$.

Superimpose a tree of constant height: Assume that $u = 2^{2k}$ for some integer k, so that $\sqrt{u}$ is an integer. Then superimpose a tree of degree $\sqrt{u}$ on top of the bit vector. The height of the tree is 2.

The internal nodes at depth 1 we can view as an array *summary*$[0..\sqrt{u}-1]$. Then *summary*$[i] = 1$ iff the subarray $A[i\sqrt{u}..(i+1)\sqrt{u}-1]$ contains a 1. We call this $\sqrt{u}$-bit subarray of A the ith **cluster**.

- To insert x, set $A[x]$ and *summary*$[\lfloor x/\sqrt{u}\rfloor]$ to 1, which takes $O(1)$ time.
- To find the minimum (maximum) value, find the leftmost (rightmost) entry in *summary* that contains a 1, and then do a linear search within that cluster for the leftmost (rightmost) 1.
- To find the successor (predecessor) of x, first search to the right (left) within its cluster. If no 1 is found, continue search to the right (left) within *summary* array from index $\lfloor x/\sqrt{u}\rfloor$.
- To delete x, let $i = \lfloor x/\sqrt{u}\rfloor$. Set $A[x]$ to 0 and *summary*$[i]$ to the logical-or of the bits in the ith cluster.

In each operation we search through at most two clusters plus the *summary* array, so the time is $O(\sqrt{u})$. This approach will be a key idea of van Emde Boas trees.

A recursive structure

We will use the idea of superimposing a tree of degree $\sqrt{u}$ on top of a bit vector, but shrink the universe size recursively by a square root at each tree level. The $u^{1/2}$ items on the first level each hold structures of $u^{1/4}$ items, which hold structures of $u^{1/8}$ items, and so on, down to size 2.

Assume for simplicity now that $u = 2^{2^k}$ for some integer k. Our aim is to achieve time complexity:

$$T(u) = T(\sqrt{u}) + O(1) = T(u^{1/2^k}) + O(k) = O(\log\log u)$$

since $u^{1/2^k} = 2$ implies $2^k = \log u$, which gives $k = \log\log u$.

On the top level of the tree, $\log u$ bits are needed to store the universe size, and each level needs half the bits of the previous level. Define:

high$(x) = \lfloor x/\sqrt{u}\rfloor$, the most significant $(\log u)/2$ bits of x gives the number of x's cluster.

low$(x) = x \bmod \sqrt{u}$, the least significant $(\log u)/2$ bits of x gives x's position within its cluster.

index$(x, y) = x\,\sqrt{u} + y$, builds an element number, where $x =$ index(high(x),low(x)).

Proto van Emde Boas structure or **proto-vEB**(u):

- If $u = 2$ then it contains an array $A[0..1]$ of two bits.
- For $u = 2^{2^k}$ for $k \geq 1$, it contains the attributes:
  - a pointer *summary* to a top *proto-vEB*($\sqrt{u}$) structure
  - an array *cluster*$[0..\sqrt{u} - 1]$ of $\sqrt{u}$ pointers to *proto-vEB*($\sqrt{u}$) structures as leaves

Determining if a value x is in a set V takes $O(\log \log u)$ time:

PROTO-vEB-MEMBER(V, x)
if $V.u = 2$ **then return** $V.A[x]$
else return PROTO-vEB-MEMBER(V.*cluster*[high(x)], low(x))

The value high(x) gives the proto-vEB($\sqrt{u}$) to visit and low(x) gives the element within that structure we are querying.

Finding the minimum element:

PROTO-vEB-MINIMUM(V)
if $V.u = 2$ **then**
  if $V.A[0] = 1$ **then return** 0
  elseif $V.A[1] = 1$ **then return** 1
  else return NIL
else *min-cluster* $\leftarrow$ PROTO-vEB-MINIMUM(V.*summary*)
  if *min-cluster* $=$ NIL **then return** NIL
  else *offset* $\leftarrow$ PROTO-vEB-MINIMUM(V.*cluster*[*min-cluster*])
    return index(*min-cluster, offset*)

If not the base case, find the first cluster that contains an element. If the set is non-empty, get the offset of the minimum element within the cluster.

With two recursive calls in the worst case, the time is

$$T(u) = 2T(\sqrt{u}) + O(1) = 2^k T(u^{1/2^k}) + O(2^{k-1}) = O(\log u)$$

instead of the desired $O(\log \log u)$.

Finding the successor:

PROTO-vEB-SUCCESSOR(V, x)
if $V.u = 2$ **then**
  if $x = 0$ and $V.A[1] = 1$ **then return** 1
  else return NIL
else *offset* $\leftarrow$ PROTO-vEB-SUCCESSOR(V.*cluster*[high(x)], low(x))
  if *offset* $\neq$ NIL **then return** index(high(x), *offset*)
  else *succ-cluster* $\leftarrow$ PROTO-vEB-SUCCESSOR(V.*summary*, high(x))
    if *succ-cluster* $=$ NIL **then return** NIL
    else *offset* $\leftarrow$ PROTO-vEB-MINIMUM(V.*cluster*[*succ-cluster*])
      return index(*succ-cluster, offset*)

If not the base case, search for a successor within x's cluster, assigning the result to *offset*. If there is none, search for the next non-empty cluster. If any, *offset* gives the first element in that cluster.

With possibly two recursive calls plus the call to PROTO-vEB-MINIMUM, the time is

$$T(u) = 2T(\sqrt{u}) + O(\log \sqrt{u}) = 2T(\sqrt{u}) + O(\log u) = 2^k T(u^{1/2^k}) + O(2^{k-1} \log u) = O(\log u \log \log u)$$

Inserting an element:

PROTO-vEB-INSERT(V, x)
if $V.u = 2$ **then** $V.A[x] \leftarrow 1$
else PROTO-vEB-INSERT(V.*cluster*[high(x)], low(x))
PROTO-vEB-INSERT(V.*summary*, high(x))

If not the base case, insert x in the right cluster and set the summary bit for that cluster to 1.

Time is the same as for PROTO-vEB-MINIMUM: $T(u) = 2T(\sqrt{u}) + O(1) = O(\log u)$.

Deleting an element is more complicated since we cannot just reset the appropriate summary bit to 0.

The van Emde Boas tree (vEB tree)

We will now just assume that $u = 2^k$ for some integer k. When $\sqrt{u}$ is not an integer we will divide the $\log u$ bits into the most significant $\lceil (\log u)/2 \rceil$ bits and the least significant $\lfloor (\log u)/2 \rfloor$ bits. We denote $2^{\lceil (\log u)/2 \rceil}$ by $\sqrt[\uparrow]{u}$ (upper square root) and $2^{\lfloor (\log u)/2 \rfloor}$ by $\sqrt[\downarrow]{u}$ (lower square root). Hence, $u = \sqrt[\uparrow]{u} \cdot \sqrt[\downarrow]{u}$.

high(x) $= \lfloor x / \sqrt[\downarrow]{u} \rfloor$

low(x) $= x \bmod \sqrt[\downarrow]{u}$

index(x, y) $= x \sqrt[\downarrow]{u} + y$

Attribute *summary* points to a vEB($\sqrt[\uparrow]{u}$) tree, and array *cluster*$[0..\sqrt[\uparrow]{u} - 1]$ points to $\sqrt[\uparrow]{u}$ vEB($\sqrt[\downarrow]{u}$) trees.

A vEB tree also stores its minimum element as *min* and its maximum as *max*, which help us as follows:

1. MINIMUM and MAXIMUM operations do not need to recurse.
2. SUCCESSOR can avoid a recursive call to determine if the successor of x lies within its high(x), because x's successor lies within its cluster iff $x <$ *max* of its cluster.
3. INSERT and DELETE will be easy if both *min* and *max* are NIL, or if they are equal.
4. If a vEB tree is empty, INSERT takes constant time just by updating its *min* and *max*. Similarly, if it has only one element DELETE takes constant time.

Time will be given by $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1)$, which also solves to $O(\log \log u)$.

Finding minimum and maximum:

vEB-TREE-MINIMUM(V)
return V.*min*

vEB-TREE-MAXIMUM(V)
return V.*max*

Determining if a value x is in set V:

```
vEB-TREE-MEMBER(V, x)
if x = V.min or x = V.max then return TRUE
elseif V.u = 2 then return FALSE
else return vEB-TREE-MEMBER(V.cluster[high(x)], low(x))
```

Finding the successor:

```
vEB-TREE-SUCCESSOR(V, x)
if V.u = 2 then
    if x = 0 and V.max= 1 then return 1
    else return NIL
elseif V.min ≠ NIL and x < V.min then return V.min
else max-low ← vEB-TREE-MAXIMUM(V.cluster[high(x)])
    if max-low ≠ NIL and low(x) < max-low then
        offset ← vEB-TREE-SUCCESSOR(V.cluster[high(x)], low(x))
        return index(high(x), offset)
    else succ-cluster ← vEB-TREE-SUCCESSOR(V.summary, high(x))
        if succ-cluster = NIL then return NIL
        else offset ← vEB-TREE-MINIMUM(V.cluster[succ-cluster])
            return index(succ-cluster, offset)
```

If not the base case and $x \geq$ minimum value, let *max-low* be the maximum in x's cluster. If there is a greater element in the cluster then assign it to *offset* and return the index of the successor. Otherwise we have to search for the next non-empty cluster. If any, *offset* gives the minimum in that cluster.

With just one recursive call, time is $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1) = O(\log \log u)$.

Finding the predecessor is almost symmetric. There is one extra case when x's predecessor, if it exists, does not reside in x's cluster.

Inserting an element:

```
vEB-EMPTY-TREE-INSERT(V, x)
V.min ← x
V.max ← x
```

```
vEB-TREE-INSERT(V, x)
if V.min = NIL then vEB-EMPTY-TREE-INSERT(V, x)
else if x < V.min then exchange x with V.min
    if V.u > 2 then
        if vEB-TREE-MINIMUM(V.cluster[high(x)]) = NIL then
            vEB-TREE-INSERT(V.summary, high(x))
            vEB-EMPTY-TREE-INSERT(V.cluster[high(x)], low(x))
        else vEB-TREE-INSERT(V.cluster[high(x)], low(x))
    if x > V.max then V.max ← x
```

Update the *min* value if necessary. Then, if not the base case and the relevant cluster is empty, insert x's cluster number into the summary and insert x into the empty cluster. If x's cluster was not empty, insert x into it (the summary need not be updated). At the end check if the *max* needs to be updated.

Time is $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1) = O(\log \log u)$, since inserting into an empty tree is $O(1)$.

Deleting an element:

vEB-TREE-DELETE(V, x)
if V.*min* $=$ V.*max* **then** V.*min* $\leftarrow$ V.*max* $\leftarrow$ NIL
elseif $V.u = 2$ **then**
  if $x = 0$ **then** V.*min* $\leftarrow 1$ **else** V.*min* $\leftarrow 0$
  V.*max* $\leftarrow$ V.*min*
else if $x = V$.*min* **then**
  first-cluster $\leftarrow$ vEB-TREE-MINIMUM(V.*summary*)
  $x \leftarrow$ index(*first-cluster*, vEB-TREE-MINIMUM(V.*cluster*[*first-cluster*]))
  V.*min* $\leftarrow x$
  vEB-TREE-DELETE(V.*cluster*[high(x)], low(x))
  if vEB-TREE-MINIMUM(V.*cluster*[high(x)]) $=$ NIL **then**
    vEB-TREE-DELETE(V.*summary*, high(x))
    if $x = V$.*max* **then**
      summary-max $\leftarrow$ vEB-TREE-MAXIMUM(V.*summary*)
      if *summary-max* $=$ NIL **then** V.*max* $\leftarrow$ V.*min*
      else V.*max* $\leftarrow$ index(*summary-max*, vEB-TREE-MAXIMUM(V.*cluster*[*summary-max*]))
  elseif $x = V$.*max* **then** V.*max* $\leftarrow$ index(high(x), vEB-TREE-MAXIMUM(V.*cluster*[high(x)]))

If $|V| \geq 2$ and $u \geq 4$ we have to delete an element from a cluster. This may not be x, because if x equals *min* then after deleting x, another element within one of V's clusters becomes the new *min*, and we have to delete that element from its cluster.

If the cluster now is empty then remove x's cluster number from the summary. If we have deleted *max* we have to find another maximum element. It is equal to *min* if all of V's clusters are empty.

Finally, if x's cluster did not become empty when x was deleted, we may need to update *max*.

Two recursive calls can be made, vEB-TREE-DELETE(V.*cluster*[high(x)], low(x)) and vEB-TREE-DELETE(V.*summary*, high(x)), but the second call is only reached when x's cluster is empty. Then x was the only element in its cluster at the first recursive call, which takes $O(1)$ time to execute. The recurrence is therefore as before: $T(u) \leq T(\sqrt[\uparrow]{u}) + O(1) = O(\log \log u)$.