Chapter 20

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 20.1-1

To modify these structure to allow for multiple elements, instead of just storing a bit in each of the entries, we can store the head of a linked list representing how many elements of that value that are contained in the structure, with a NIL value to represent having no elements of that value.

Exercise 20.1-2

All operations will remain the same, except instead of the leaves of the tree being an array of integers, they will be an array of nodes, each of which stores $x.key$ in addition to whatever additional satellite data you wish.

Exercise 20.1-3

To find the successor of a given key k from a binary tree, call the procedure $SUCC(x, T.root)$. Note that this will return NIL if there is no entry in the tree with a larger key.

Exercise 20.1-4

The new tree would have height k. INSERT would take $O(k)$, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and DELETE would take $O(ku^{1/k})$.

Exercise 20.2-1

See the two algorithms, PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

Exercise 20.2-2

When we delete a key, we need to check membership of all keys of that cluster to know how to update the summary structure. There are $\sqrt{u}$ of these, and each membership takes $O(\lg \lg u)$ time to check. With the recursive calls, recurrence for running time is

Algorithm 1 SUCC(k,x)

```
if k < x.key then
    if x.left == NIL then
        return x
    else
        if SUCC(k, x.left) == NIL then
            return x
        else
            return SUCC(k,x.left)
        end if
    end if
else
    if x.right == NIL then
        return NIL
    else
        return SUCC(k,x.right)
    end if
end if
```

Algorithm 2 PROTO-vEB-MAXIMUM(V)

```
if V.u == 2 then
    if V.A[1] == 1 then
        return 1
    else if V.A[0] == 1 then
        return 0
    else
        return NIL
    end if
else
    max-cluster = PROTO-vEB-MAXIMUM(V.summary)
    if max-cluster ==NIL then
        return NIL
    else
        offset = PROTO − vEB − MINIMUM(V.cluster[max − cluster])
        return index(max − cluster, offset)
    end if
end if
```

Algorithm 3 PROTO-vEB-PREDECESSOR(V,x)

```
if V.u==2 then
    if x==1 and V.A[0] ==1 then
        return 0
    else
        return NIL
    end if
else
    offset = PROTO−vEB−PREDECESSOR(V.cluster[high(x)], low(x))
    if offset ≠ NIL then
        return index(high(x),offest)
    else
        pred-cluster = PROTO-vEB-PREDECESSOR(V.summary,high(x))
        if pred-cluster ==NIL then
            return NIL
        else
            return index(succ-cluster, PROTO-vEB-
MAXIMUM(V.cluster[pred-cluster]))
        end if
    end if
end if
```

$$T(u) = T(\sqrt{u}) + O(\sqrt{u} \lg \lg u).$$

We make the substitution $m = \lg u$ and $S(m) = T(2^m)$. Then we apply the Master Theorem, using case 3, to solve the recurrence. Substituting back, we find that the runtime is $T(u) = O(\sqrt{u} \lg \lg u)$.

Exercise 20.2-3

We would keep the same as before, but insert immediately after the else, a check of whether $n = 1$. If it doesn't continue as usual, but if it does, then we can just immediately set the summary bit to zero, null out the pointer in the table, and be done immediately. This has the upside that it can sometimes save up to $\lg \lg u$. The procedure has the big downside that the number of elements that are in the set could be as high as $\lg(\lg(u))$, in which case $\lg(u)$ many bits are needed to store n.

Exercise 20.2-4

The array A found in a proto van Emde Boas structure of size 2 should now support integers, instead of just bits. All other pats of the structure will remain the same. The integer will store the number of duplicates at that position. The modifications to insert, delete, minimum, successor, etc... will be minor. Only the base cases will need to be updated.

Algorithm 4 PROTO-vEB-DELETE(V, x)

```
if V.u == 2 then
    V.A[x] = 0
else
    PROTO-vEB-DELETE(V.cluster[high(x)], low(x))
    inCluster = False
    for i = high(x) · √u to (high(x) + 1) · √u − 1 do
        if PROTO-vEB-MEMBER(V.cluster[high(x)], i) then
            inCluster = True
            Break
        end if
    end for
    if inCluster == False then
        PROTO-vEB-DELETE(V.summary, high(x))
    end if
end if
```

Exercise 20.2-5

The only modification necessary would be for the u=2 trees. They would need to also include a length two array that had pointers to the corresponding satellite data which would be populated in case the corresponding entry in A were 1.

Exercise 20.2-6

This algorithm recursively allocates proper space and appropriately initializes attributes for a proto van Emde Boas structure of size u.

Algorithm 5 Make-Proto-vEB(u)

```
V = allocate-node()
V.u = u
if u == 2 then
    Allocate-Array A of size 2
    V.A[1] = V.A[0] = 0
else
    V.summary = Make-Proto-vEB(√u)
    for i = 0 to √u − 1 do
        V.cluster[i] = Make-Proto-vEB(√u)
    end for
end if
```

Exercise 20.2-7

For line 9 to be executed, we would need that in the summary data, we also had a NIL returned. This could of either happened through line 9, or 6. Eventually though, it would need to happen in line 6, so, there must be some number of summarizations that happened of V that caused us to get an empty u=2 vEB. However, a summarization has an entry of one if any of the corresponding entries in the data structure are one. This means that there are no entries in V, and so, we have that V is empty.

Exercise 20.2-8

For MEMBER the runtime recurrence is $T(u) = T(u^{1/4}) + O(1)$, whose solution is again $O(\lg \lg(u))$. For MINIMUM, $T(u) = 2T(u^{1/4}) + O(1)$. Making a substitution and applying case 1 of the master theorem, this is $O(\sqrt{\lg u})$. For SUCCESSOR, $T(u) = 2T(u^{1/4}) + O(\lg u)$. By case 3 of the master theorem, this is $O(\lg u)$. For INSERT, $T(u) = 2T(u^{1/4}) + O(1)$. This is the same as MINIMUM, which is $O(\sqrt{\lg u})$. To analyze DELETE, we need to update the recurrence to reflect the fact that DELETE depends on MEMBER. The new recurrence is $T(u) = T(u^{1/4}) + O(u^{1/4} \lg \lg u)$. By case 3 of the master theorem, this is $O(u^{1/4} \lg \lg u)$.

Exercise 20.3-1

To support duplicate keys, for each u=2 vEB tree, instead of storing just a bit in each of the entries of its array, it should store an integer representing how many elements of that value the vEB contains.

Exercise 20.3-2

For any key which is a minimum on some vEB, we'll need to store its satellite data with the min value since the key doesn't appear in the subtree. The rest of the satellite data will be stored alongside the keys of the vEB trees of size 2. Explicitly, for each non-summary vEB tree, store a pointer in addition to min. If min is NIL, the pointer should also point to NIL. Otherwise, the pointer should point to the satellite data associated with that minimum. In a size 2 vEB tree, we'll have two additional pointers, which will each point to the minimum's and maximum's satellite data, or NIL if these don't exist. In the case where min=max, the pointers will point to the same data.

Exercise 20.3-3

We define the procedure for any u that is a power of 2. If $u = 2$, then, just slap that fact together with an array of length 2 that contains 0 in both entries.

If $u = 2^k > 2$, then, we create an empty vEB tree called Summary with $u = 2^{\lceil k/2 \rceil}$. We also make an array called cluster of length $2^{\lceil k/2 \rceil}$ with each

entry initialized to an empty vEB tree with $u = 2^{\lfloor k/2 \rfloor}$. Lastly, we create a min and max element, both initialized to NIL.

Exercise 20.3-4

Suppose that x is already in V and we call INSERT. Then we can't satisfy lines 1, 3, 6, or 10, so we will enter the else case on line 9 every time, causing an infinite loop. Now suppose we call DELETE when x isn't in V. If there is only a single element in V, lines 1 through 3 will delete it, regardless of what element it is. To enter the elseif of line 4, x can't be equal to 0 or 1 and the vEB tree must be of size 2. In this case, we delete the max element, regardless of what it is. Since the recursive call always puts us in this case, we always delete an element we shouldn't. To avoid these issue, keep and updated auxiliary array A with u elements. Set $A[i] = 0$ if i is not in the tree, and 1 if it is. Since we can perform constant time updates to this array, it won't affect the runtime of any of our operations. When inserting x, check first to be sure $A[x] == 0$. If it's not, simply return. If it is, set $A[x] = 1$ and proceed with insert as usual. When deleting x, check if $A[x] == 1$. If it isn't, simply return. If it is, set $A[x] = 0$ and proceed with delete as usual.

Exercise 20.3-5

Similar to the analysis of (20.4), we will analyze:

$$T(u) \leq T(u^{1-1/k}) + T(u^{1/k}) + O(1)$$

This is a good choice for analysis because for many operations we first check the summary vEB tree, which will have size $u^{1/k}$ (the second term). And then possible have to check a vEB tree somewhere in cluster, which will have size $u^{1-1/k}$(the first term). We let $T(2^m) = S(m)$, so the equation becomes

$$S(m) \leq S(m(1 - 1/k)) + S(m/k) + O(1).$$

If $k > 2$ the first term dominates, so by master theorem, we'll have that $S(m)$ is $O(\lg(m))$, this means that T will be $O(\lg(\lg(u)))$ just as in the original case where we took squareroots.

Exercise 20.3-6

Set $n = u/\lg\lg u$. Then performing n operations takes $c(u + n\lg\lg u)$ time for some constant c. Using the aggregate amortized analysis, we divide by n to see that the amortized cost of each operations is $c(\lg\lg u + \lg\lg u) = O(\lg\lg u)$ per operation. Thus we need $n \geq u/\lg\lg u$.

Problem 20-1

a. Lets look at what has to be stored for a vEB tree. Each vEB tree contains one vEB tree of size $\sqrt[+]{u}$ and $\sqrt[+]{u}$ vEB trees of size $\sqrt[-]{u}$. It also is storing three numbers each of order $O(u)$, so they need $\Theta(\lg(u))$ space each. Lastly, it needs to store $\sqrt[+]{u}$ many pointers to the cluster vEB trees. We'll combine these last two contributions which are $\Theta(\lg(u))$ and $\Theta(\sqrt[+]{u})$ respectively into a single term that is $\Theta\sqrt[+]{u}$. This gets us the recurrence

$$P(u) = P(\sqrt[+]{u}) + \sqrt[+]{u}P(\sqrt[-]{u}) + \Theta(\sqrt{u})$$

Then, we have that $u = 2^{2m}$(which follows from the assumption that $\sqrt{u}$ was an integer), this equation becomes

$$P(u) = (1 + 2^m)P(2^m) + \Theta(\sqrt{u}) = (1 + \sqrt{u})P(\sqrt{u}) + \Theta(\sqrt{u})$$

as desired.

b. We recall from our solution to problem 3-6.e (it seems like so long ago now) that given a number n, a bound on the number of times that we need to take the squareroot of a number before it falls below 2 is $\lg(\lg(n))$. So, if we just unroll out recurrence, we get that

$$P(u) \leq \left(\prod_{i=1}^{\lg(\lg(u))} (u^{1/2^i} + 1) \right) P(2) + \sum_{i=1}^{\lg(\lg(u))} \Theta(u^{1/2^i})(u^{1/2i} + 1)$$

The first product has a highest power of u corresponding to always multiplying the first terms of each binomial. The power in this term is equal to $\sum_{i=1}^{\lg(\lg(u))} \frac{1}{2^i}$ which is a partial sum of a geometric series whose sum is 1. This means that the first term is $o(u)$. The order of the ith term in the summation appearing in the formula is $u^{2/2^i}$. In particular, for $i = 1$ is it $O(u)$, and for any $i > 1$, we have that $2/2^i < 1$, so those terms will be $o(u)$. Putting it all together, the largest term appearing is $O(u)$, and so, $P(u)$ is $O(u)$.

c. For this problem we just use the version written for normal vEB trees, with minor modifications. That is, since there are entries in cluster that may not exist, and summary may of not yet been initialized, just before we try to access either, we check to see if it's initialized. If it isn't, we do so then.

d. As in the previous problem, we just wait until just before either of the two things that may of not been allocated try to get used then allocate them if need be.

e. Since the initialization performed only take constant time, those modifications don't ruin the the desired runtime bound for the original algorithms already had. So, our responses to parts c and d are $O(\lg(\lg(n)))$.

f. As mentioned in the errata, this part should instead be changed to $O(n \lg(n))$ space. When we are adding an element, we may have to add an entry to a dynamic hash table, which means that a constant amount of extra space

Algorithm 6 RS-vEB-TREE-INSERT(V,x)

if $V.min == NIL$ **then**
  vEB-EMPTY-TREE-INSERT(V,x)
else
  if $x < V.min$ **then**
    swap V.min with x
  end if
  if $V.u > 2$ **then**
    if $V.summary == NIL$ **then**
      $V.summary = CREATE - NEW - RD - vEB - TREE(\sqrt[+]{V.u})$
    end if
    if $lookup(V.cluster, low(x)) == NIL$ **then**
      insert into V.summary with key high(x) what is returned by $CREATE - NEW - RD - vEB - TREE(\sqrt[-]{V.u})$
    end if
    if $vEB-TREE-MINIMUM(lookup(V.cluster, high(x))) == NIL$ **then**
      vEB-TREE-INSERT(V.summary,high(x))
      vEB-EMPTY-TREE-INSERT(lookup(V.cluster,high(x)),low(x))
    else
      vEB-TREE-INSERT(lookup(V.cluster,high(x)),low(x))
    end if
  end if
  if $x > V.max$ **then**
    $V.max = x$
  end if
end if

Algorithm 7 RS-vEB-TREE-SUCCESSOR(V,x)

if $V.u == 2$ **then**
  if $x == 0$ and $V.max == 1$ **then**
    return 1
  else
    return NIL
  end if
else if $V.min \neq NIL$ and $x < V.min$ **then**
  return V.min
else
  if $lookup(V.cluster, low(x)) == NIL$ **then**
    insert into V.summary with key high(x) what is returned by $CREATE - NEW - RD - vEB - TREE(\sqrt[-]{V.u})$
  end if
  max-low = vEB-TREE-MAXIMUM(lookup(V.cluster,high(x)))
  if $max - low \neq NIL$ and $low(x) < max - low$ **then**
    return$index(high(x), vEB - TREE - SUCCESSOR(lookup(V.summary, high(x)), low(x)))$
  else
    if $V.summary == NIL$ **then**
      $V.summary = CREATE - NEW - RD - vEB - TREE(\sqrt[+]{V.u})$
    end if
    $succ{-}cluster = vEB{-}TREE{-}SUCCESSOR(V.summary, high(x))$
    if succ-cluster==NIL **then**
      return NIL
    else
      return$index(succ - cluster, vEB - TREE - MINIMUM(lookup(V.summary, succ - cluster)))$
    end if
  end if
end if

would be needed. If we are adding an element to that table, we also have to add an element to the RS-vEB tree in the summary, but the entry that we add in the cluster will be a constant size RS-vEB tree. We can charge the cost of that addition to the summary table to the making the minimum element entry that we added in the cluster table. Since we are always making at least one element be added as a new min entry somewhere, this amortization will mean that it is only a constant amount of time in order to store the new entry.

g. It only takes a constant amount of time to create an empty RS-vEB tree. This is immediate since the only dependence on u in CREATE-NEW-RS-vEB-TREE(u) is on line 2 when V.u is initialized, but this only takes a constant amount of time. Since nothing else in the procedure depends on u, it must take a constant amount of time.

Problem 20-2

a) By 11.5, the perfect hash table uses $O(m)$ space to store m elements. In a universe of size u, each element contributes $\lg u$ entries to the hash table, so the requirement is $O(n \lg u)$. Since the linked list requires $O(n)$, the total space requirement is $O(n \lg u)$.

b) MINIMUM and MAXIMUM are easy. We just examine the first and last elements of the associated doubly linked list. MEMBER can actually be performed in $O(1)$, since we are simply checking membership in a perfect hash table. PREDECESSOR and SUCCESSOR are a bit more complicated. Assume that we have a binary tree in which we store all the elements and their prefixes. When we query the hash table for an element, we get a pointer to that element's location in the binary search tree, if the element is in the tree, and NULL otherwise. Moreover, assume that every leaf node comes with a pointer to its position in the doubly linked list. Let x be the number whose successor we seek. Begin by performing a binary search of the prefixes in the hash table to find the longest hashed prefix y which matches a prefix of x. This takes $O(\lg \lg u)$ since we can check if any prefix is in the hash table in $O(1)$. Observe that y can have at most one child in the BST, because if it had both children then one of these would share a longer prefix with x. If the left child is missing, have the left child pointer point to the largest labeled leaf node in the BST which is less than y. If the right child is missing, use its pointer to point to the successor of y. If If y is a leaf node then $y = x$, so we simply follow the pointer to x in the doubly linked list, in $O(1)$, and its successor is the next element on the list. If y is not a leaf node, we follow its predecessor or successor node, depending on which we need. This gives us $O(1)$ access to the proper element, so the total runtime is $O(\lg \lg u)$. INSERT and DELETE must take $O(\lg u)$ since we need to insert one entry into the hash table for each of their bits and update the pointers.

c) The hash table has $\lg u$ entries for each of the $n/\lg u$ groups, so it stores a total of n entries, making it size $O(n)$. There are $n/\lg u$ binary trees of size $\lg u$, so they take $O(n)$ space. Finally, the linked list takes $O(n/\lg u)$ space. Thus, the total space requirement is $O(n)$.

d) For MINIMUM: Starting with the linked list, locate the minimum representative. This is $O(1)$ since we can just look at the start of the doubly linked list. Then use the hash table to find its corresponding binary tree in $O(1)$. Since this binary tree contains $lg(u)$ elements and is balanced, its height is $\lg\lg u$, so we can find its minimum in $O(\lg\lg u)$. The procedure is similar for MAXIMUM.

e) We start by finding the smallest representative greater than or equal to x. To do this, store the representatives in the structure described above, with runtimes given in parts a and b, and call SUCCESSOR(x) to find the proper binary search tree to look in. Since $n \leq u$ we can do this in $O(\lg\lg u)$. Next we search the binary search tree for x. Since its height is $\lg u$, the total runtime is $O(\lg\lg u)$.

f) Again, if we can find the largest representative greater than or equal to x, we can determine which binary tree contains the predecessor or successor of x. To do this, just call PREDECESSOR or SUCCESSOR on x to locate the appropriate tree in $O(\lg\lg u)$. Since the tree has height $\lg u$, we can find the predecessor or successor in $(O\lg\lg u)$.

g) Insertion and deletion into a binary tree of height $\lg u$ is $\Omega(\lg\lg u)$. In addition to this, we may have to update the representatives of the groups which can only increase the running time. representatives.

h) We can relax the requirements and only impose the condition that each group has at least $\frac{1}{2}\lg u$ elements and at most $2\lg u$ elements. If a red-black tree is too big, we split it in half at the median. If a red-black tree is too small, we merge it with a neighboring tree. If this causes the merged tree to become too large, we split it at the median. If a tree splits, we create a new representative. If two trees merge, we delete the lost representative. Any split or merge takes $O(\lg u)$ since we have to insert or delete an element in the data structure storing our representatives, which by part b takes $O(\lg u)$. However, we only split a tree after at least $\lg u$ insertions, since the size of one of the red-black trees needs to increase from $\lg u$ to $2\lg u$ and we only merge two trees after at least $(1/2)\lg u$ deletions, because the size of the merging tree needs to have decreased from $\lg u$ to $(1/2)\lg u$. Thus, the amortized cost of the merges, splits, and updates to representatives is $O(1)$ per insertion or deletion, so the amortized cost is $O(\lg\lg u)$ as desired.