20 van Emde Boas Trees

20.1 Preliminary approaches

20.1-1

> Modify the data structures in this section to support duplicate keys.

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.

20.1-2

> Modify the data structures in this section to support keys that have associated satellite data.

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 $x.\text{key}$ in addition to whatever additional satellite data you wish.

20.1-3

> Observe that, using the structures in this section, the way we find the successor and predecessor of a value x does not depend on whether x is in the set at the time. Show how to find the successor of x in a binary search tree when x is not stored in the tree.

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

20.1-4

> Suppose that instead of superimposing a tree of degree $\sqrt{u}$, we were to superimpose a tree of degree $u^{1/k}$, where $k > 1$ is a constant. What would be the height of such a tree, and how long would each of the operations take?

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

20.2 A recursive structure

20.2-1

> Write pseudocode for the procedures PROTO-vEB-MAXIMUM and PROTO-vEB-PREDECESSOR.

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

20.2-2

> Write pseudocode for PROTO-vEB-DELETE. It should update the appropriate summary bit by scanning the related bits within the cluster. What is the worst-case running time of your procedure?

```
PROTO-vEB-DELETE(V, x)
    if V.u == 2
        V.A[x] = 0
    else
        PROTO-vEB-DELETE(V.cluster[high(x)], low(x))
        inCluster = false
        for i = 0 to sqrt(u) - 1
```

```
            if PROTO-vEB-MEMBER(V.cluster[high(x)], low(i))
                inCluster = true
                break
        if inCluster == false
            PROTO-vEB-DELETE(V.summary, high(x))
```

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

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

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

20.2-3

> Add the attribute n to each proto-vEB structure, giving the number of elements currently in the set it represents, and write pseudocode for PROTO-vEB-DELETE that uses the attribute n to decide when to reset summary bits to 0. What is the worst-case running time of your procedure? What other procedures need to change because of the new attribute? Do these changes affect their running times?

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 0, 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.

20.2-4

> Modify the proto-vEB structure to support duplicate keys.

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.

20.2-5

> Modify the proto-vEB structure to support keys that have associated satellite data.

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.

20.2-6

> Write pseudocode for a procedure that creates a proto-vEB(u) structure.

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

```
MAKE-PROTO-vEB(u)
    allocate a new vEB tree V
    V.u = u
    if u == 2
        let A be an array of size 2
        V.A[1] = V.A[0] = 0
    else
        V.summary = MAKE-PROTO-vEB(sqrt(u))
        for i = 0 to sqrt(u) - 1
            V.cluster[i] = MAKE-PROTO-vEB(sqrt(u))
```

20.2-7

> Argue that if line 9 of PROTO-vEB-MINIMUM is executed, then the proto-vEB structure is empty.

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.

20.2-8

> Suppose that we designed a proto-vEB structure in which each *cluster* array had only $u^{1/4}$ elements. What would the running times of each operation be?

There are $u^{3/4}$ clusters in each proto-vEB.

- **MEMBER/INSERT:**

$$T(u) = T(u^{1/4}) + O(1) = \Theta(\lg \log_4 u) = \Theta(\lg \lg u).$$

- **MINIMUM/MAXIMUM:**

$$T(u) = T(u^{1/4}) + T(u^{3/4}) + O(1) = \Theta(\lg u).$$

- **SUCCESSOR/PREDECESSOR/DELETE:**

$$T(u) = T(u^{1/4}) + T(u^{3/4}) + \Theta(\lg u^{1/4}) = \Theta(\lg u \lg \lg u).$$

20.3 The van Emde Boas tree

20.3-1

> Modify vEB trees to support duplicate keys.

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.

20.3-2

> Modify vEB trees to support keys that have associated satellite data.

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.

20.3-3

> Write pseudocode for a procedure that creates an empty van Emde Boas tree.

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 $u = 2k > 2$, 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.

20.3-4

> What happens if you call VEB-TREE-INSERT with an element that is already in the vEB tree? What happens if you call VEB-TREE-DELETE with an element that is not in the vEB tree? Explain why the procedures exhibit the behavior that they do. Show how to modify vEB trees and their operations so that we can check in constant time whether an element is present.

- [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.

20.3-5

> Suppose that instead of $\sqrt{u}$ clusters, each with universe size $\sqrt{u}$, we constructed vEB trees to have $u^{1/k}$ clusters, each with universe size $u^{1-1/k}$, where $k > 1$ is a constant. If we were to modify the operations appropriately, what would be their running times? For the purpose of analysis, assume that $u^{1/k}$ and $u^{1-1/k}$ are always integers.

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

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

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

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

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.

20.3-6

> Creating a vEB tree with universe size u requires $O(u)$ time. Suppose we wish to explicitly account for that time. What is the smallest number of operations n for which the amortized time of each operation in a vEB tree is $O(\lg \lg u)$?

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 $n \ge u/\lg\lg u$.

Problem 20-1 Space requirements for van Emde Boas trees

> This problem explores the space requirements for van Emde Boas trees and suggests a way to modify the data structure to make its space requirement depend on the number n of elements actually stored in the tree, rather than on the universe size u. For simplicity, assume that $\sqrt{u}$ is always an integer.
>
> **a.** Explain why the following recurrence characterizes the space requirement $P(u)$ of a van Emde Boas tree with universe size u:
>
> $$P(u) = (\sqrt{u} + 1)P(\sqrt{u}) + \Theta(\sqrt{u}). \qquad (20.5)$$
>
> **b.** Prove that recurrence (20.5) has the solution $P(u) = O(u)$.
>
> In order to reduce the space requirements, let us define a ***reduced-space van Emde Boas tree***, or ***RS-vEB tree***, as a vEB tree V but with the following changes:
>
> - The attribute $V.\text{cluster}$, rather than being stored as a simple array of pointers to vEB trees with universe size $\sqrt{u}$, is a hash table (see Chapter 11) stored as a dynamic table (see Section 17.4). Corresponding to the array version of $V.\text{cluster}$, the hash table stores pointers to RS-vEB trees with universe size $\sqrt{u}$. To find the ith cluster, we look up the key i in the hash table, so that we can find the ith cluster by a single search in the hash table.

- The hash table stores only pointers to nonempty clusters. A search in the hash table for an empty cluster returns NIL, indicating that the cluster is empty.
- The attribute $V.\text{summary}$ is NIL if all clusters are empty. Otherwise, $V.\text{summary}$ points to an RS-vEB tree with universe size $\sqrt{u}$.

Because the hash table is implemented with a dynamic table, the space it requires is proportional to the number of nonempty clusters.

When we need to insert an element into an empty RS-vEB tree, we create the RS-vEB tree by calling the following procedure, where the parameter u is the universe size of the RS-vEB tree:

```
CREATE-NEW-RS-vEB-TREE(u)
    allocate a new vEB tree V
    V.u = u
    V.min = NIL
    V.max = NIL
    V.summary = NIL
    create V.cluster as an empty dynamic hash table
    return V
```

c. Modify the VEB-TREE-INSERT procedure to produce pseudocode for the procedure $\text{RS-VEB-TREE-INSERT}(V, x)$, which inserts x into the RS-vEB tree V, calling $\text{CREATE-NEW-RS-VEB-TREE}$ as appropriate.

d. Modify the $\text{VEB-TREE-SUCCESSOR}$ procedure to produce pseudocode for the procedure $\text{RS-VEB-TREE-SUCCESSOR}(V, x)$, which returns the successor of x in RS-vEB tree V, or NIL if x has no successor in V.

e. Prove that, under the assumption of simple uniform hashing, your RS-VEBTREE-INSERT and $\text{RS-VEB-TREE-SUCCESSOR}$ procedures run in $O(\lg \lg u)$ expected time.

f. Assuming that elements are never deleted from a vEB tree, prove that the space requirement for the RS-vEB tree structure is $O(n)$, where n is the number of elements actually stored in the RS-vEB tree.

g. RS-vEB trees have another advantage over vEB trees: they require less time to create. How long does it take to create an empty RS-vEB tree?

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

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

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 $\sum_{i=1}^{\lg \lg u}$ 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 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 $\text{CREATE-NEW-RSvEB-TREE}(u)$ 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 y-fast tries

This problem investigates D. Willard's "y-fast tries" which, like van Emde Boas trees, perform each of the operations MEMBER, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on elements drawn from a universe with size u in $O(\lg \lg u)$ worst-case time. The INSERT and DELETE operations take $O(\lg \lg u)$ amortized time. Like reduced-space van Emde Boas trees (see Problem 20-1), y-fast tries use only $O(n)$ space to store n elements. The design of y-fast tries relies on perfect hashing (see Section 11.5).

As a preliminary structure, suppose that we create a perfect hash table containing not only every element in the dynamic set, but every prefix of the binary representation of every element in the set. For example, if $u = 16$, so that $\lg u = 4$, and $x = 13$ is in the set, then because the binary representation of 13 is 1101, the perfect hash table would contain the strings 1, 11, 110, and 1101. In addition to the hash table, we create a doubly linked list of the elements currently in the set, in increasing order.

a. How much space does this structure require?

b. Show how to perform the MINIMUM and MAXIMUM operations in $O(1)$ time; the MEMBER, PREDECESSOR, and SUCCESSOR operations in $O(\lg \lg u)$ time; and the INSERT and DELETE operations in $O(\lg u)$ time.

To reduce the space requirement to $O(n)$, we make the following changes to the data structure:

- We cluster the n elements into $n/\lg u$ groups of size $\lg u$. (Assume for now that $\lg u$ divides n.) The first group consists of the $\lg u$ smallest elements in the set, the second group consists of the next $\lg u$ smallest elements, and so on.
- We designate a "representative" value for each group. The representative of the ith group is at least as large as the largest element in the ith group, and it is smaller than every element of the $(i+1)$st group. (The representative of the last group can be the maximum possible element $u - 1$.) Note that a representative might be a value not currently in the set.
- We store the $\lg u$ elements of each group in a balanced binary search tree, such as a red-black tree. Each representative points to the balanced binary search tree for its group, and each balanced binary search tree points to its group's representative.

> The perfect hash table stores only the representatives, which are also stored in a doubly linked list in increasing order.
>
> We call this structure a y-***fast trie***.
>
> **c.** Show that a y-fast trie requires only $O(n)$ space to store n elements.
>
> **d.** Show how to perform the MINIMUM and MAXIMUM operations in $O(\lg \lg u)$ time with a y-fast trie.
>
> **e.** Show how to perform the MEMBER operation in $O(\lg \lg u)$ time.
>
> **f.** Show how to perform the PREDECESSOR and SUCCESSOR operations in $O(\lg \lg u)$ time.
>
> **g.** Explain why the INSERT and DELETE operations take $\Omega(\lg \lg u)$ time.
>
> **h.** Show how to relax the requirement that each group in a y-fast trie has exactly $\lg u$ elements to allow INSERT and DELETE to run in $O(\lg \lg u)$ amortized time without affecting the asymptotic running times of the other operations.

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 NIL 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 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 doubly linked list has less than n elements, while the binary search trees contains n nodes, thus a y-fast trie requires $O(n)$ space.

d. MINIMUM: Find the minimum representative in the doubly linked list in $\Theta(1)$, then find the minimum element in the binary search tree in $O(\lg \lg u)$.

e. Find the smallest representative greater than k with binary searching in $\Theta(\lg \lg u)$, find the element in the binary search tree in $O(\lg \lg u)$.

f. 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. Same as ***e***, we need to find the cluster in $\Theta(\lg \lg u)$, then the operations in the binary search tree takes $O(\lg \lg u)$.

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.