Chapter 8

Bounded Universes

Until so far, most of the data structures described in this book have been *comparison based*. Although they've been described as storing real-valued keys, the only operations performed on keys have been comparisons, so they could just as easily store keys from any total order. In this chapter we study the issue of what happens when the keys are integers and we can do operations other than comparisons, like addition, subtraction, integer division, and using the number to index arrays.

More specifically, our keys come from the universe $U = \{0, \ldots, N-1\}$ and we assume a model that allows us to do most common mathematical operations on integers and gives integer results by truncating (taking the floor of) the results of operations when necessary. The results of operations can also be used as indices into arrays.

We are interested in data structures that support insertion, deletion, membership queries and next element queries on keys from U. The meanings of insertion and deletion are obvious. A membership query takes as input a key k and determines whether k is currently stored in the data structure. An next element query takes a key k and returns the smallest value k' stored in the data structure such that $k' \geq k$.

As a first attempt, we might consider using an array $A_1, \ldots, A_N$ of boolean values. On creation, every array entry is initialized to `false`. When an element k is inserted, we simply set A_k to `true`. Similarly, when an element k is deleted we set A_k to `false`. To do a membership query on the key k we just check the value of A_k. Unfortunately, this is where this approach breaks down. The only way to do a next element query on key k is to check the values of A_k, A_{k+1}, A_{k+2}, and so on until we find the first value $A_{k'}$ that is set to true. Thus, with this approach the first three operations can be done in constant time, but the fourth (next element search) operation takes $\Omega(N)$ time in the worst case.

8.1 Van Emde Boas Trees

An interesting (and old) method of doing searches on bounded universes was proposed by Peter Van Emde Boas. It is based on the recurrence

$$T_N = \begin{cases} O(1) + T_{\sqrt{N}} & \text{if } N \geq 2 \\ O(1) & \text{otherwise} \end{cases} \tag{8.1}$$

which solves to $O(\log \log N)$ (because $N^{1/\log n} = 2$). Essentially, the above recurrence says that if we can, in constant time, reduce the search space to the square root of its original size and then recurse we will get a running time of $O(\log \log N)$.

The data structure we use to achieve this is called a VEB-tree. Assume $N = 2^{2^m}$ for some integer m.[1] The root of the VEB-tree for U has $\sqrt{N}$ children that are stored in an array. Each child of the root is also a VEB-tree, and the ith child corresponds to a VEB-tree for elements $i\sqrt{N}, \ldots, (i+1)\sqrt{N} - 1$. The root of the VEB-tree also stores two integer values called min and max, which contain the smallest and largest element currently contained in the tree. If no values are stored in the tree then min and max are set to some value not in U, say -1. A crucial point to remember about a VEB-tree is that the min and max values at the root are only stored at the root. Because of this, we can insert into an initially empty tree or delete the last element from a tree in constant time.

The root of the VEB-tree also contains another VEB-tree—and this is the truly clever part—for the universe $\{0, \ldots, \sqrt{N} - 1\}$. This auxilliary tree is used to keep track of which children of the root contain data. That is, the tree contains the element i if the ith child of the root contains some key.

From this definition, it follows that the storage used by a VEB-tree is given by the recurrence

$$S_N = (\sqrt{N} + 1)S_{\sqrt{N}} + \sqrt{N}$$

which solves to $O(N)$. (The extra $+1$ comes from the auxilliary VEB-tree.)

Although VEB-trees are not extremely complicated, their implementation requires some care. In the following three sections we sketch the implementations of the algorithms for searching, inserting and deleting in VEB-trees, and provide more precise pseudocode. In the pseudocode, W is a VEB-tree node, $W[i]$ is the ith child of W, $\text{child}(k, W)$ is the index of the child of W that stores the key k, and $\text{aux}(W)$ is the auxilliary VEB-tree stored at W.

8.1.1 Searching

To do a next element search for the key k in a VEB-tree, we first check if k is less than the min value stored at the root. If so, then we simply report the min value. Otherwise, we need to determine which child of the root stores the key k. This can be done in constant time since it is the $i = \lfloor k/\sqrt{N} \rfloor$th child and the children are stored in an array. We then inspect the max value for the ith child. If it is larger than k, then we can be sure that the key k' we are looking for is in the subtree rooted at the ith child and we recurse on the ith child.

[1] We only make this assumption so that $N^{1/2^i}$ is an integer for all integers $i < m$. This allows us to avoid the need for floors, ceilings, and special cases. The modifications required when N is not of this form should be clear.

Otherwise, the key we are looking for is contained in the jth child of the root, where j is the smallest value greater than i such that the jth child of the root contains some key. In fact, the key k' we are looking for is the min value the jth child of the root. Therefore, we can use the auxilliary tree at the root to find the value of j and then report the min value in constant time.

In both cases, the algorithm makes one recursive search call and does $O(1)$ work. The recursive search call is on a VEB-tree for a universe of size $\sqrt{N}$. Thus, the running time of the search algorithm is given by the recurrence (8.1) and is $O(\log\log N)$.

```
SUCCESSOR(k, W)
 1: if k < min(W) then
 2:    output min(W) {k is smaller than every element}
 3: else if k > max(W) then
 4:    output ∞ {k is larger than every element}
 5: end if
 6: i ← child(k, W)
 7: if max(W[i]) > k then
 8:    SUCCESSOR(k, W[i]) {k′ is stored in W[i]}
 9: else
10:    j ← SUCCESSOR(i, aux(W)) {k′ is in first non-empty sibling of W[i]}
11:    output min(W[j])
12: end if
```

8.1.2 Inserting

To insert the key k into a VEB-tree we proceed as follows. If the root of T is empty then we simply set the min and max values at the root to be k. Otherwise, we check if k is less than (respectively greater than) the min value (respectively max value) at the root. If so, we swap the values of k and the min value (respectively max value) before continuing. Next, we find the child of the root that should contain the key k using the formula $i = \lfloor k/\sqrt{N} \rfloor$. If the ith child of the root contains no elements then we insert i into the root's auxilliary VEB-tree and insert k into the ith child of the root. Otherwise (the ith child already contains some element) we only insert k into the ith child of the root.

Observe that, because we explicitly check this condition, inserting into an empty VEB-tree takes constant time. This is very important, because the VEB-tree algorithm may make two recursive insertion calls; once to insert k and once to insert i into the auxilliarly VEB-tree. However, in this case, the recursive call to insert k takes only constant time. Thus, no matter what happens, the running time of the insertion algorithm satisfies the "rootish" recurrence (8.1) and therefore runs in $O(\log\log n)$ time.

```
INSERT(k, W)
 1: if min(W) = max(W) = −1 then
 2:    min(W) ← max(W) ← k {tree is empty}
 3: else if min(W) = max(W) then
 4:    min(W) ← min{k, min(W)} {tree contains 1 element}
 5:    max(W) ← max{k, max(W)}
 6: else
 7:    if k < min(W) then
 8:       swap k ↔ min(W) {k is the new min, insert the old min}
```

```
 9:    else if k > max(W) then
10:       swap k ↔ max(W) {k is the new max, insert the old max}
11:    end if
12:    i ← child(k, W)
13:    INSERT(k, W[i])
14:    if max(W[i]) = min(W[i]) = k then
15:       INSERT(i, aux(W)) {W[i] just went from empty to non-empty}
16:    end if
17: end if
```

8.1.3 Deleting

Deleting the key k from a VEB-tree is similar to insertion. If the tree contains only the element k, it is stored as the min and max values of the root and we can delete it in constant time. Otherwise, if k is equal to the min (respectively max) value at the root then we swap k and the min (respectively max) value at the root. We then recursively delete k from the child of the root that contains it and, if this child becomes empty we delete the child's index from the auxiliary VEB-tree.

As with insertion, although there may be two recursive calls, only one of them takes more than constant time. Thus, the running time of the deletion algorithm is given by recurrence (8.1) and runs in $O(\log\log n)$.

```
DELETE(k, W)
 1: if min(W) = max(W) = k then
 2:    min(W) ← max(W) ← −1 { k is the last element }
 3: else if min(aux(W)) = −1 then
 4:    if min(W) = k then
 5:       min(W) ← max(W)
 6:    else
 7:       max(W) ← min(W)
 8:    end if
 9: else if k = min(W) then
10:    j ← min(aux(W))
11:    min(W) ← min(W[j])
12:    k ← min(W)
13: else if k = max(W) then
14:    j ← max(aux(W))
15:    max(W) ← max(W[j])
16:    k ← max(W)
17: end if
18: i ← child(k, W)
19: DELETE(k, W[i])
20: if min(W[i]) = max(W[i]) = −1 then
21:    DELETE(i, aux(W))
22: end if
```

Theorem 23. *Van Emde Boas trees support insertion, deletion, and successor queries for elements in the universe* $U = \{0, \ldots, N-1\}$ *in* $O(\log\log N)$ *time and require* $O(N)$ *storage.*

8.2 Reducing Storage

8.3 Willard's X- and Y-Fast Trees

Luc's notes go here.

8.4 Discussion and References

The Van Emde Boas tree (VEB-tree) was introduced by van Emde Boas [1, 2]. The description here was conveyed to us by Michael Bender. Since then, several variants have been introduced, most with the goal of reducing the storage...

Bibliography

[1] P. van Emde Boas. An $O(n \log n)$ on-line algorithm for the insert-extract min problem. Technical Report TR-74-221, Department of Computer Science, Cornell University, December 1974.

[2] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.