6.854 Advanced Algorithms

Lecture 3: September 15, 2003 Lecturer: David Karger

Scribes: abhi shelat (1999), Andrew Menard (1999), Akshay Patil (2003), Daniel Myers(2006)

Van Emde Boas Queues

We would like to design a queue that only handles integers from $0...C$, but supports all operations in $O(\log \log C)$ time. This idea is first presented in "Design and Implementation of an efficient priority queue", Mathematical Systems Theory 10 (1977). Mikkel Thorup provides some extensions in "On RAM priority queues" in SODA 1996.

This is a fantastic example of an efficient recursive data structure. This structure also exploits the fact that keys in priority queues are most often integers and that computers have efficient operations for manipulating the binary representations of integers (shifts, masks, logical operations).

Each vEB queue maintains the following information:

Field	Notation
The current minimum	$Q.min$
vEB queue on high halfword	$Q.summ$
An array where each entry is a vEB queue on the low halfwords	$Q[x_h]$

NB: 2006 lectures referred to $Q.summ$ as $Q.high$ and $Q[x_h]$ as $Q.low$.

Intuitively, we have the minimum item stored so we can return it quickly. VEB priority queues take the place of the buckets in the multi-level bucket structure that was presented earlier (see Scribe Notes 2). Specifically, we note that the problem of finding the next minimum among the buckets in the previous multi-level structure is actually a priority queue problem, so we will recursively use our own priority queue implementation to solve it!

There are $\sqrt{U}$ elements in the array ($Q[x_h]$ aka $Q.low$), and each element in the array corresponds to a distinct high halfword x_h. (high halfword, x_h = the higher-order $\frac{w}{2}$ bits in the w-bit representation of a number, x; $w = \lg C$. low halfword, x_l = the rest). Thus when looking for an element, we can look for the upper high halfword of the number, then recurse on the low halfword

The purpose of the $summ$ is to quickly determine which low halfword vEB queues actually contain elements. If a low halfword vEB is non-empty, then we store the vEB's address (x_h) in $Q.summ$. Thus $Q.summ$ is a vEB of size $\sqrt{U}$ just like the low halfword vEBs.

This is a rather straightforward approach and requires $O(U)$ space. (a more complex, randomized approach is possible, leading to $O(\sqrt{U})$ space.)

Inserts

Consider the algorithm for inserting an element into the queue.

Algorithm 1 INSERT(x, Q)

```
1: if Q.min = null then
2:    Q.min = x, return
3: end if
4: if Q.min > x then
5:    swap x ↔ Q.min
6: end if
7: (x_h, x_l) ⇐ x {divide x into its high halfword, x_h, and its low halfword, x_l}
8: if Q[x_h].min = null then
9:    INSERT(x_h, Q.summ)
10:   Q[x_h].min = x_l
11: else
12:   INSERT(x_l, Q[x_h])
13: end if
```

First we update all Q.min information. This takes lines 1-5

Line 7 of the algorithm splits x into halfwords. This operation can be done with a bit shift operation and a bitwise-AND operation.

At this point, we have split the problem of inserting x into this queue into a smaller problem of either inserting the high halfword of x, or the low halfword of x. If we have already inserted x_h in a previous operation, then it suffices to insert x_l in the low halfword queue that belongs to x_h. If we have not already seen x_h, then we need to insert it into our high halfword queue. However, this means that we can insert x_l in $O(1)$ time by simply changing $Q[x_h].min$.

The recursions finishes when the queues become small enough to store as arrays (eg, 1-bit queues).

Notice that the function makes only one recursive call to the INSERT function. Because all of the other operations are $O(1)$ time, the recurrence is

$$T(b) = T(b/2) + O(1)$$

where b is the number of bits in x. Hence, $T(b) = O(\log b)$ and since $b = \log C$, the algorithm runs in $O(\log \log C)$ time.

Find-Min

Trivial, return Q.min. This should take $O(1)$ time.

Delete

At somepoint, every element is a minimum element for some vEB. We handle this base case by deleting the min and then recalculating what the minimum element is. We must be mindful to keep $Q.summ$ up to date as well. Otherwise, we merely drill down the levels of vEB until we reach our base case situation.

Once again, the running time for this algorithm is $O(\log \log C)$. The reason why this again results in only one call to a $O(\sqrt{U})$ problem is that even if DELETE is called twice, the second call implies that the first call took only constant time. (if $Q(x_h)$ is now empty, that means that the earlier call to DELETE was deleting that vEBs minimum, and only, element. Such a call takes constant time, so we can now use our $O(\sqrt{U})$ time on updating $Q.summ$).

Algorithm 2 DELETE(x, Q)

```
1: if x < Q.min then
2:    return/die
3: end if
4: if x = Q.min then
5:    first = Q.summ.min
6:    if first = null then
7:       Q.min = null, return
8:    end if
9:    x = Q.min = first|Q[first].min {create element with high halfword = "first", low = Q[first].min }
10: end if
11: DELETE(x_l, Q[x_h])
12: if Q[x_h].min = null then
13:    DELETE(x_h, Q.summ)
14: end if
```

Other supported operations

Note that it is easy to support all of the following operations in $O(\log \log C)$ time

- MAX(). Symmetric to the min.
- FIND(x). Determines whether the element exists in the queue.
- SUCC(x). Finds the smallest value in the queue that is larger than x. Returns nothing if x is the largest.
- PRED(x). Finds the largest value in the queue that is smaller than x. Returns nothing if x is the smallest.
- DELETE-MIN(x). Special case of DELETE.
- DELETE-MAX(). Removes the max element from the queue. Works in the same way that DELETE-MIN works.