6.854 Advanced Algorithms

Lecture 3: September 16, 1999 Lecturer: David Karger

Scribes: abhi shelat, Andrew Menard

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	min
Number of items in the queue	n
vEB queue on high halfword	$summ$
For each $x_h \in summ$, a vEB queue on the low halfwords	$L(x_h)$

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). We maintain the number of items in a queue so that we can quickly check whether a queue has just one item. This will be important during the delete step.

The purpose of the HASH table is to allow constant time access to elements in the high halfword queue (high halfword, x_h = the higher-order $\frac{w}{2}$ bits in the w-bit representation of a number, x; $w = \lg C$). We use x_h as the key, and $L(x_h)$ as the value in HASH. Implementing a hashtable to be truly constant time in the complexity analysis is tricky. Multiply operations, for example, are not constant time bit operations. The hashtable is only used to save space. The more straightforward indexed array approach requires $\sqrt{U}$ space.

Inserts

Consider the algorithm for inserting an element into the queue.

INSERT(x, S) [1] $(x_h, x_l) \Leftarrow x$ divide x into its high halfword, x_h, and its low halfword, x_l $x_h \in$ HASH INSERT$(x_l, L(x_h))$ $L_(x_h)$ is stored as the key to x_h in HASH INSERT(x_h, H) $L(x_h) \Leftarrow$ MAKEVEB(x_l) insert x_h into hashtable, $HASH$ $x < min(S)$ $min(S) \Leftarrow x$ $n \Leftarrow n + 1$

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

Line 1 of the algorithm splits x into halfwords. This operation can be done with a bit shift operation and a bitwise-AND operation. To check whether x_h is in H, we query $HASH$ in constant time.

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 creating a queue of size 1.

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.

A sample queue with the elements $15, 3, 9$ would look like this:

Figure 3.1: A sample VEB queue.

Delete-Min

In order to delete an element, we return the current minimum and find its replacement. In order to find the replacement, there can be two cases. We first look in the low halfword queue for the current minimum, that is $L(m_h)$ where m_h is the high halfword of $min(S)$. If this queue has other elements, we recursively delete the minimum from this queue, and read off the new minimum.

If the low halfword queue only has one element, then we need to find the next smallest high halfword. We do this by recursively deleting the minimum from H, and then reading off the appropriate values for our new minimum.

DELETE-MIN(S) [1] $(m_h, x_l) \Leftarrow min(S)$ divide the current minimum into high and low halfwords $L(m_h)$ has one element DELETE-MIN(H) $x'_h =$ MIN(H) $x'_l =$ MIN($L(x'_h)$) DELETE-MIN($L(x_h)$) $x'_l \Leftarrow$ MIN($L(m_h)$) $x'_h \Leftarrow m_h$ $n \Leftarrow n - 1$ oldmin $\Leftarrow min(S)$ $min(S) \Leftarrow x'_h(2^{b/2}) + x'_l$ return oldmin

Once again, the running time for this algorithm is $O(\log \log C)$ since all of the comparisons are $O(1)$ time and only one recursive call on $b/2$ bits is made.

Queue Creation

In order to create a queue in $O(1)$ time, we have to be careful (lazy). The naive creation of a queue of size b bits requires an insertion into the $b/2$ bit high halfword queue, and the creation of a $b/2$ bit low halfword queue. Since the high halfword queue might need to be created before we can insert into it, the recurrence describing the amount of work required is $T(b) = 2T(b/2) + O(1) = O(b)$. This is too expensive. To get around this problem, we employ *lazy* creation.

When a queue is first created, we defer the creation of the high and low halfword queues. We simply record the first value that we are inserting as the queue's minimum value. The other pointers in the queue are set to NULL so that we can differentiate this queue during access/modification operations. Since we no longer make recursive calls to create smaller queues, We have reduced the amount of

work required to create a queue to $O(1)$. For example, suppose we have inserted 9 into the four-bit queue from above. Our structure would be

Figure 3.2: A 4 bit VEB queue after inserting 9.

We must argue that the lazy creation does not affect any of the other queue operations. Let us consider what happens to the queue after subsequent inserts. The next insert into the newly minted queue will either be a key that is larger than the minimum or smaller than the minimum. In either case, we will be expanding only the larger element. Let y be the larger element. The insert requires us to create a high halfword queue for y_h, to allocate a hashtable, to create a low halfword queue for y_l, and to insert the low halfword queue in the hashtable. Both of the queue creations that were required are lazy creates as described above. Hence, the amount of time spent for queue creation is again $O(1)$. At this point, there are two elements in the top queue.

Figure 3.3: A 4 bit VEB queue after inserting 9,13.

On the third insert, z, we might have to expand the high halfword queue if $y_h \neq z_h$. This involves one more lazy queue create. As before, we create a queue for z_l and insert it into the hashtable. If $y_h = z_h$, then we only expand the low halfword queue $L(y_h)$. Each insertion requires at most two lazy queue creations.

The key point is that the minimum value of a queue is never inserted into the queue. It is simply recorded as the minimum. This allows the rest of the algorithms to work correctly and at the same time facilities $O(1)$ time queue creation.

Other supported operations

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

- MIN(). Can be done in $O(1)$.
- 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(x). Deletes the element x from the queue.
- DELETE-MAX(). Removes the max element from the queue. Works in the same way that DELETE-MIN works.