6.046 Recitation 9: van Emde Boas

April 23, 2012

1 Announcements

- Takehomes will be returned at the end of recitation today.
- Problem set 7 is released, due next Wednesday.
- There is a survey and a link has been posted to Piazza; please respond. It's completely anonymous and we'd love to get feedback so we can improve the course for the last month or so of term.

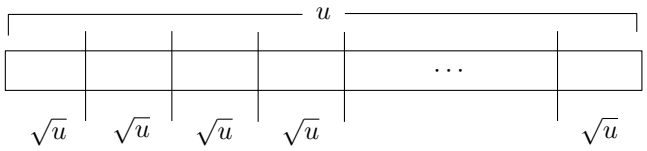
2 Review: The van Emde Boas structure

vEB data structure stores a set S of integers, which is a subset of $U = \{0, 1, \ldots, u-1\}$. It supports the following operations:

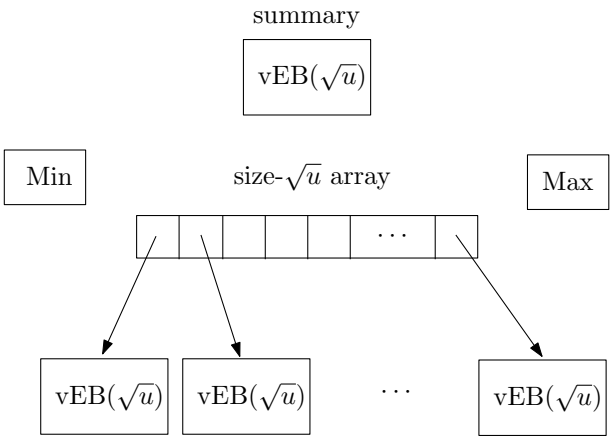
- INSERT(S, x), DELETE(S, x), PREDECESSOR(S, x), SUCCESSOR(S, x) in $O(\lg \lg u)$ time
- MIN(S, x) in constant time

2.1 Overall idea

- Split the range into $\sqrt{u}$ chunks of size $\sqrt{u}$ each; each chunk is governed by a smaller vEB structure on the range of size $\sqrt{u}$.



- Store the overall minimum and maximum outside of the rest of the structure (not stored in any subclusters).
- "Summary" structure is a vEB structure of on a range of $\sqrt{u}$ where $S.summary[c]$ is 1 when cluster c is nonempty.



2.2 Pseudocode

In this section, we will generally use $x = c\sqrt{u} + i$, where $c, i < \sqrt{u}$. c is the cluster number of x, and i is the corresponding index into that cluster.

Each vEB structure S has the fields $S.min$, $S.max$, $S.cluster$ (an array of pointers to the individual $\sqrt{u}$ sub-clusters), and $S.summary$.

2.2.1 Predecessor and Successor

SUCCESSOR(S, x):

```
1  if S is empty: return NIL
2  if x < S.min: return S.min
3  if x < S.cluster[c].max:
4      return c√u + SUCCESSOR(S.cluster[c], i)
5  else
6      c' = SUCCESSOR(S.summary, c)
7      if c' is NIL: return S.max
8      else
9          return c'√u + S.cluster[c'].min
```

Intuition for SUCCESSOR: in the big picture, there are three cases – the successor of x, if it exists, is either the minimum of the structure, in the same cluster as x, or in the next cluster to the right. Overall, the pseudocode is basically handling each of these cases individually.

PREDECESSOR is identical to SUCCESSOR, but switching minimums and maximums as appropriate.

The recurrence for this procedure is $T(u) = T(\sqrt{u}) + O(1)$, which is $O(\lg \lg u)$ as seen in lecture.

2.2.2 Insert

INSERT(S, x):

```
1  if S is empty: S.min = S.max = x
2  if x < S.min: swap x and S.min and continue.
3  if x ≥ S.max: swap x and S.max and continue.
4  if S.cluster[c] is empty:
5      INSERT(S.summary, c)
6  INSERT(S.cluster[c], i)
```

There is at most one recursive call for any given input: if we ever need to call line 5, then line 6 is guaranteed to take constant time. Therefore the recurrence for this is likewise $T(u) = T(\sqrt{u}) + O(1)$.

2.2.3 Delete

DELETE(S, x):

```
 1  if this makes S empty: set S.min = S.max = NIL
 2  if x = S.min:
 3      c' = S.summary.min
 4      if c' is NIL: set S.min = S.max
 5      else
 6          S.min = S.cluster[c'].min
 7          DELETE(S.cluster[c'], S.min)
 8          if S.cluster[c'].min = NIL: // Did we make the cluster empty?
 9              DELETE(S.summary, c')
10  if x = S.max, do the same as the above, but with max instead of min
11  else
12      DELETE(S.cluster[c], i), where x = c√u + i
13      if S.cluster[c].min = NIL:
14          DELETE(S.summary, c)
```

DELETE is slightly more complicated than the others; we must consider the cases:

- where the deleted element is the only element of the vEB structure, thus leaving it empty,
- where it is the minimum element but not the only one, meaning we have to find the smallest remaining element after x is deleted,
- an analogous case where the maximum element is deleted,
- and the usual case where we need to delete x from the cluster containing it.

The only cases where there are two recursive calls to DELETE are in each of the cases in which the deletion makes the cluster containing the deleted element empty; however, we know that that takes constant time, so the recurrence is still $T(u) = T(\sqrt{u}) + O(1)$.

3 Space complexity of vEB

We want to make sure that this recursive structure doesn't take up too much space. In this portion we will show that the vEB data structure takes $O(u)$ space.

To figure out how much space the data structure takes up, we can look at how much space each of the components of the van Emde Boas data structure takes.

- The separately stored minimum and maximum elements, which take $O(1)$ space (we assume that each integer fits in a single word of memory).
- The sub-clusters, of which there are $\sqrt{u}$, each of which takes up the amount of space taken by a vEB structure on a range of $\sqrt{u}$. In other words, the total space taken by these is $\sqrt{u}S(\sqrt{u})$.
- The summary structure, which is a vEB structure of on a range of $\sqrt{u}$; this takes $S(\sqrt{u})$ space.
- An array storing pointers to each of the $\sqrt{u}$ clusters, which takes up $\sqrt{u}$ space.

Summing all of these items together, we get the space recurrence $S(u) = (\sqrt{u}+1)S(\sqrt{u}) + \sqrt{u}$.

We can show that this is linear by substitution. Suppose that $S(k) < c_1 k - c_2$ for all $k < u$.

Then

$$\begin{aligned} S(u) &< (\sqrt{u}+1)(c_1\sqrt{u} - c_2) + \sqrt{u} \\ &= c_1 u - c_2\sqrt{u} + c_1\sqrt{u} - c_2 + \sqrt{u} \\ &= c_1 u - c_2 - \sqrt{u}(c_2 - c_1 - 1) \end{aligned}$$

We can pick c_1 and c_2 appropriately so that $c_2 - c_1 - 1 > 0$ and so that the base case is satisfied, and therefore the space taken is $O(u)$.

Likewise, we can show that the space is $\Omega(u)$ by assuming $S(k) > ck$ for all $k < u$.

Then

$$\begin{aligned} S(u) &> (\sqrt{u}+1)(c\sqrt{u}) + \sqrt{u} \\ &= cu + (c+1)\sqrt{u} \\ &> cu \end{aligned}$$

4 Reducing the space taken

Though $O(u)$ space is good, in the case where the universe is large and the size of S is small, we can probably do better than to take up size of space equal to the size of the whole universe.

Idea: reduce the amount of space taken up by the structure by making the following changes:

- Don't store the empty clusters; create clusters on demand instead, so the only clusters taking up space are the ones that acutally have elements in them.
- Instead of storing *S.cluster* as an array of pointers, store it as a dynamic hash table whose keys are the cluster numbers (and whose values are pointers to the clusters).

The dynamic hash table is implemented with table doubling, as described in CLRS 17.4. We assume simple uniform hashing in this case for an expected running time of $O(1)$ for all operations.

The new method of storing things requires some modifications to the procedures we had before.

4.1 Modifications

4.1.1 Predecessor and Successor

The pseudocode and process for getting the predecessor and successor of some element basically remains exactly the same; the only difference is that when we used to reference $S.cluster[c]$ as a pointer into some portion of an array, all such references are now hash table lookups.

4.1.2 Insert

The key difference for INSERT is that when we are trying to insert into either a vEB structure or one of its clusters and the relevant structure is empty, rather than simply inserting it we need to create a new structure.

Overall, if S is empty before x is inserted, we need to create a new empty vEB structure with all fields set to NIL except for $S.min$ and $S.max$, which are set to x once x is inserted.

In the case where x is inserted into a previously empty cluster, we also need to create a new cluster containing only x; the additional step required here is also to insert a pointer to the new cluster into the dynamic hash table $S.cluster$.

4.1.3 Delete

Likewise with DELETE, we need to delete the relevant cluster every time the deletion causes some cluster to become empty; we also need to remove the relevant key from the hash table. This could happen to the cluster we are deleting the element from, the summary cluster of the top-level structure, or the entirety of the vEB structure S.

4.2 Running Time

Most of the time taken by operations is the same as it was in the previous version of the vEB structure. There are a few changes:

- In the predecessor and successor problems, array lookups are replaced with hash table lookups.
- In each of INSERT and DELETE, there is also a constant number of hash table operations added to the modified procedures.

Each of these changes adds at most an expected constant time additional overhead, meaning the running time is now expected $O(\lg \lg u)$ for each of these operations.

4.3 Space Complexity

Analogously to what was mentioned earlier, the components of the vEB structure that take up space in this new condensed vEB structure are:

- Separately stored minimum and maximum elements, which takes $O(1)$ space.
- The smaller vEB clusters, all of which are known to be nonempty.
- The summary structure, which is also a smaller vEB structure.
- The hash table storing pointers to the nonempty clusters.

Now we can account for how much space is taken up by each of these components. As before, the minimum and maximum take up $O(1)$ space.

Because we are only storing clusters that are nonempty, there are at most as many "descendant" clusters (vEB structures of size less than u) as there are elements in S.

We can "charge" the space cost of each descendant cluster to one of the elements in S, the amount of space taken is $O(n)$, if $n = |S|$; now we just need to account for the summary and the hash tables.

The sum total size of all hash tables is proportional to the number of nonempty clusters that exist, and so this only adds a constant amount of overhead to the space required; for a similar reason, the summary structures add only constant overhead.

6.046J / 18.410J Design and Analysis of Algorithms
Spring 2012