6.897: Advanced Data Structures Spring 2003

Lecture 2 — February 12, 2003

Prof. Erik Demaine *Scribe: Jeff Lindy*

1 Overview

In the last lecture we considered the successor problem for a bounded universe of size u. We began looking at the van Emde Boas [3] data structure, which implements Insert, Delete, Successor, and Predecessor in $O(\lg \lg u)$ time per operation.

In this lecture we finish up van Emde Boas, and improve the space complexity from our original $O(u)$ to $O(n)$. We also look at perfect hashing (first static, then dynamic), using it to improve the space complexity of van Emde Boas and to implement a simpler data structure with the same running time, y-fast trees.

2 van Emde Boas

2.1 Pseudocode for vEB operations

We start with pseudocode for Insert, Delete, and Successor. (Predecessor is symmetric to Successor.)

Insert(x, S)

  if $x <$min[S]: swap x and min[S]

  if min[sub[S][high(x)]] = nil : // was empty

    Insert(low(x), sub[S][high(x)])

    min[sub[S][high(x)]] $\leftarrow$ low(x)

  else:

    Insert(high(x), summary[S])

  if $x >$ max[S]: max[S] $\leftarrow x$

Delete(x, S)

  if min[S] = nil or $x <$ min[S]: return

  if min[S] = x:

    $i \leftarrow$ min[summary[S]]

$x \leftarrow i\sqrt{|S|} + \min[\text{sub}[S][i]]$

$\min[S] \leftarrow x$

Delete(low(x, sub[W][high(x)])

if min[sub[S][high(x)]] = nil : // now empty

Delete(high(x), summary[S])

// in this case, the first recursive call was cheap

Successor(x, S)

if $x < \min[S]$: return min[S]

if low(x) < max[sub[S][high(x)]]

return high(x) $\sqrt{|S|}$ + Successor(low(x), sub[S][high(x)])

else:

$i \leftarrow$ Successor(high(x), summary[S])

return $i\sqrt{|S|}$ + min[sub[S][i]]

2.2 Tree view of van Emde Boas

The van Emde Boas data structure can be viewed as a tree of trees.

The upper and lower "halves" of the tree are of height $\frac{1}{2} \lg u$, that is, we are cutting our tree in halves by level. The upper tree has $\sqrt{u}$ nodes, as does each of the subtrees hanging off its leaves. (These subtrees correspond to the *sub*[S] data structures in Section 2.1.)

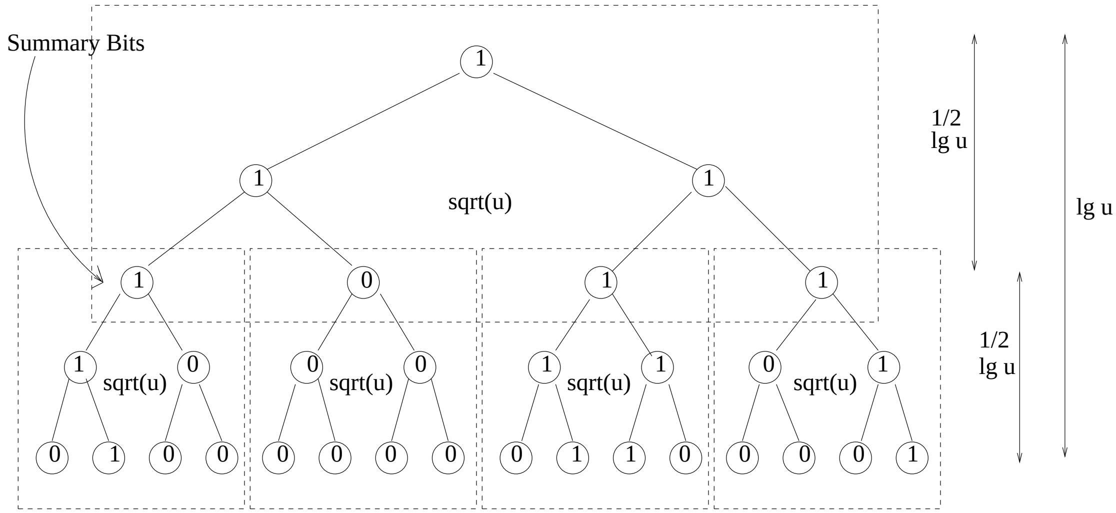

Figure 1: In this case, we have the set 1, 9, 10, 15.

We can consider our set as a bit vector (of length 16) at the leaves. Mark all ancestors of present leaves. The function high(x) tells you how to walk through the upper tree. In the same way, low(x) tells you how to walk through the appropriate lower tree to reach x.

For instance, consider querying for 9. $9_{10} = 1001_2$ which is equivalent to "right left left right"; high(9) $= 10_2$ ("right left"), low(9) $= 01_2$ ("left right").

2.3 Reducing Space in vEB

It is possible that this tree is wasting a lot of space. Suppose that in one of the subtrees there was nothing whatsoever?

Instead of storing all substructures/lower trees explicitly, we can store sub[S] as a dynamic hash table and store only nonempty substructures.

We perform an amortized analysis of space usage:

1. *Hash* – Charge space for hashtable to nonempty substructures, each of which stores an element as min.
2. *Summary* – Charge elements in the summary structure to the mins of corresponding substructures.

When a substructure has just one element, stored in the min field, it does not store a hashtable, so that it can occupy just $O(1)$ space. In all, the tree as described above uses $O(n)$ space, whereas van Emde Boas takes more space, $O(u)$, as originally considered [3]. For this approach to work, though, we need the following hashtable black box:

BLACKBOX for hash table:

Insert	in $O(1)$ time, amortized and expected
Delete	in $O(1)$ time, amortized and expected
Search	in $O(1)$ time, worse case
Space	is $O(n)$

3 Transdichotomous Model

There are some issues with using the comparison model (too restrictive) and the standard unit cost RAM (too permissive, in that we can fit arbitrarily large data structures into single words). Between the two we have the transdichotomous model.

This machine is a modified RAM, and is also called a *word RAM*, with some set size word upon which you can perform constant time arithmetic.

We want to bridge problem size and machine model. So we need a word that is big enough ($\lg\ n$) to index everything we are talking about. (This is a reasonable model since, say, a 64 bit word can index a truly enormous number of integers, 2^{64}.) With this model, we can manipulate a constant number of words in a constant time.

4 Perfect Hashing

Perfect hashing is a way to implement hash tables such that the expected number of collisions is minimized. We do this by building a hash table of hash tables, where the choice of hash functions for the second level can be redone to avoid clobbering elements through hash collision.

We'll start off with the static case; we already have all the elements, and won't be inserting new ones.

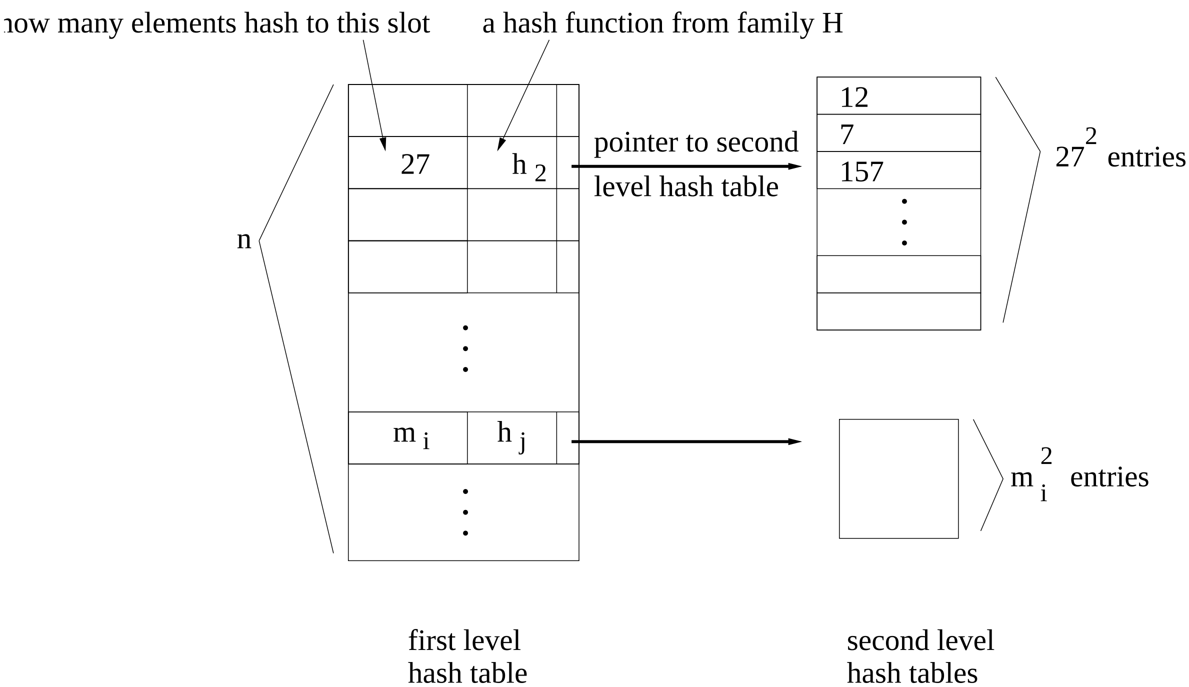

Figure 2: We're storing n elements from our universe U. h_1 maps some $u \in U$ into the first level hash table which is of size n. Each entry in the first level hash table has a count of how many elements hash there, as well as a second hash function h_i which takes elements to a spot in the appropriate second level hash table.

A set H of hash functions of the form $h : U \to \{0, 1, \ldots, m-1\}$ is called *universal* if

$$|\{h \in H : h(x) = h(y)\}| = |H|/m \qquad \text{for all } x, y \in U,\ x \neq y$$

That is, the probability of a hash collision is equal to the size of the universe over the size of the table.

Universal sets of hash functions exist. For instance, take key k, and write it in base m:

$$k = \langle k_0, k_1, \ldots, k_r \rangle$$

Choose some other vector of this size at random:

$$a = \langle a_0, a_1, \ldots, a_r \rangle$$

Then the hash function

$$h_a = \sum_{i=0}^{r} a_i k_i \bmod m$$

is essentially weighting each k_i, via a dot product. So letting the vector a vary, we have a family of hash functions, and it turns out to be universal.

Suppose m_i elements hash to bucket i (a slot in the first hash table).

The expected number of collisions in a bucket is

$$E[\# \text{ of collisions in bucket } i] = \sum_{x \neq y} Pr\{x \text{ and } y \text{ collide}\}$$

Because the second table is of size m_i^2, this sum is equal to

$$\binom{m_i}{2} \frac{1}{m_i^2} < \frac{1}{2} \qquad \text{(by Markov)}$$

The expected total size of all second-level tables is

$$E[\text{space}] = E\left[\sum_{i=0}^{n-1} m_i^2\right]$$

Because the total number of conflicting pairs in the first level can be computed by counting pairs of elements in all second-level tables, this sum is equal to

$$n + 2\ E\left[\sum_{i=0}^{n-1} \binom{m_i}{2}\right] = n + \frac{2\binom{n}{2}}{n} < 2n$$

So our expected space is about $2n$. More specifically, the probability $Pr[\text{space} > 4n] < \frac{1}{2}$.

If we clobber an element, we pick some other hash function for that bucket. We will need to choose new hash functions only an expected constant number of times per bucket, because we have a constant probability of success each time.

4.1 Dynamic Perfect Hashing

Making a perfect hash table dynamic is relatively straightforward, and can be done by keeping track of how many elements we have inserted or deleted, and rebuilding (shrinking or growing) the entire hash table. This result is due to Dietzfelbinger et al. [4].

Suppose at time t there are n^* elements.

Construct a static perfect hash table for $2n^*$ elements. If we insert another n^* elements, we rebuild again, this time a static perfect hash table for $4n^*$ elements.

As before, whenever there's a collision at the second-level hash table, just rebuild the entire second-level table with a new choice of hash function from our family H. This won't happen very often, so amortized the cost is low.

If we delete elements until we have $n^*/4$, we rebuild for n^* instead of $2n^*$. (If we delete enough, we rebuild at half the size.)

All operations take amortized $O(1)$ time.

5 y-fast Trees

5.1 Operation Costs

y-fast trees [1, 2] accomplish the same running times as van Emde Boas—$O(\lg \lg u)$ for Insert, Delete, Successor, and Predecessor—but they are simple once you have dynamic perfect hashing in place.

We can define correspondences $u \in U \longleftrightarrow$ bit string $\longleftrightarrow$ path in a balanced binary search tree.

For all x in our set, we will store a dynamic perfect hash table of all prefixes of these strings. Unfortunately, this structure requires $O(n \lg u)$ space, because there are n elements, each of which have $\lg u$ prefixes. Also, insertions and deletions cost $\lg u$. We'll fix these high costs later.

What does this structure do for Successor, though? To find a successor, we binary search for the lowest "filled" node on the path to x. We ask, "Is half the path there?" for successively smaller halves. It takes $O(\lg \lg u)$ time to find this node.

Suppose we have each node store the min and max element for its subtree (in the hash table entry). This change adds on some constant cost for space. Also, we can maintain a linked list of the entire set of occupied leaves. Then we can find the successor and the predecessor of x in $O(1)$ time given what we have so far.

Using this structure, we can perform Successor/Predecessor in $O(\lg \lg u)$ time, and Insert/Delete in $O(\lg u)$ time, using $O(n \lg u)$ space.

5.2 Indirection

We cluster the n present elements into consecutive groups of size $\Theta(\lg u)$.

- Within a group, we use a balanced binary search tree to do Insert/Delete/Successor/Predecessor in $O(\lg \lg u)$ time.
- We use hash table as above to store one representative element per group $\Rightarrow O(n)$ space
- If a search in the hashtable narrows down to 2 groups, look in both.
- Insert and Delete are delayed by the groups, so it works out to be $O(1)$ amortized to update the hash table, and $O(\lg \lg u)$ total.

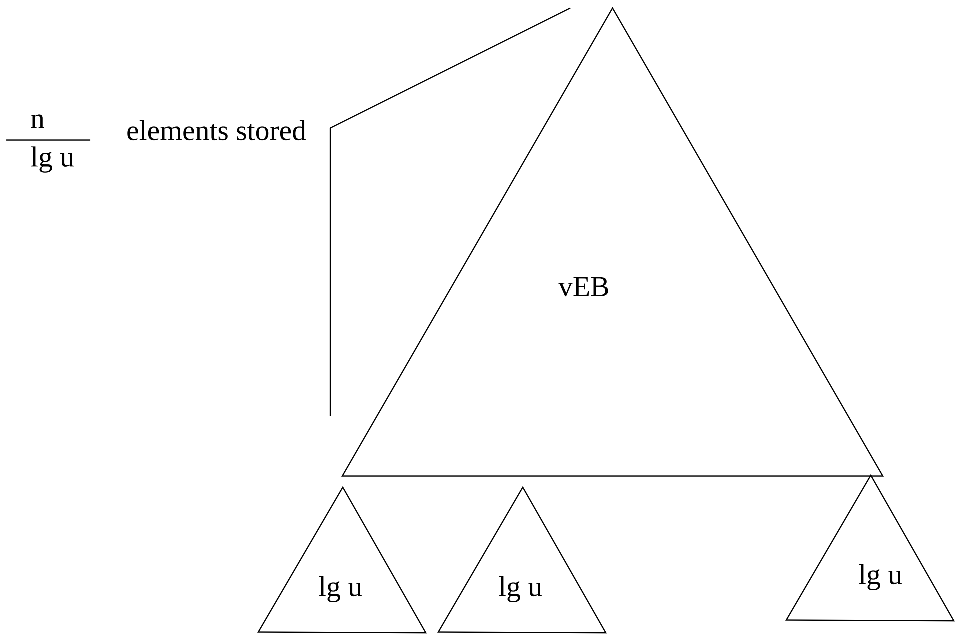

Figure 3: Using indirection in y-fast trees.

References

[1] Dan E. Willard, "Log-logarithmic worst-case range queries are possible in space $\Theta(n)$," *Information Processing Letters*, 17:81–84. 1984.

[2] Dan E. Willard, "New trie data structures which support very fast search operations," *Journal of Computer and System Sciences*, 28(3):379–394. 1984.

[3] P. van Emde Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Information Processing Letters*, 6:80–82, 1977.

[4] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan, "Dynamic perfect hashing: upper and lower bounds", *SIAM Journal on Computing*, 23:738–761, 1994. `http://citeseer.nj.nec.com/dietzfelbinger90dynamic.html`.