6.851: Advanced Data Structures Spring 2012

Lecture 11 — March 22, 2012

Prof. Erik Demaine

1 Overview

In this lecture, we will be discussing integer data structures that can be used to solve the predecessor problem. We will start by describing the various models we can use, and then introduce two integer data structures, Van Emde Boas and Y-fast trees.

2 Integer Data Structures

Before presenting any integer data structures, we present some models for computations with integers (both to precisely state what we are looking for, and as a framework to prove lower bounds).

In each model, memory is arranged as a collection of words of some size (and potentially a small amount of working memory), and a fixed cost is charged for each operation. We also always make the *fixed universe assumption*. We assume that each data structure needs to answer queries with respect to a fixed set of w-bit integers $\mathcal{U} = \{0, 1, 2, \ldots, u-1\}$.

2.1 Trandichotomous RAM

- Memory is divided into an array of size S made up of cells of size w. It is practical to require w to be greater than $\log(S)$, as otherwise, you won't be able to access the whole array. If we let n be the problem size, then this also implies that $w \geq log(n)$.
- There is some fixed set of operations, each of which modifies $O(1)$ memory cells.
- Words serve as pointer, so cells can be addressed arbitrarily.

2.2 word-RAM

In word-RAM, the processor may use the $O(1)$ fixed operations $+, -, *, /, \%, \&, |, ' >>, <<, <, >$, you would expect as primitives in a high-level language such as C.

2.3 The Cell-Probe Model

- Memory is divided into cells of size w, where w is a parameter of the model.

- Reading to or writing from a memory cell costs 1 unit.
- Other operations are free

This model does not realistically reflect a modern processor, but its simplicity and generality make it appropriate for proving lower bounds.

These models vary in how powerful they are. In particular, the cell probe model is stronger than the trandichotomous RAM model which is in turn stronger than the word RAM model, followed by the pointer machine and the BST.

3 Successor / Predecessor Queries

We are interested in data structures which maintain a set of integers S and which are able to insert an integer into S, delete an integer from S, report the smallest element of S greater than some query x, or report the largest element of S less than some query x.

We sill start by presenting the current known bounds for such a data structure on these various models.

3.1 BST Model

In the BST model, this problem is known to require $\Theta(\log(n))$ time per query.

3.2 Word RAM model

In the word RAM model, we have a few data structures. Using the Van Emde Boas data structure, we can achieve $O(\log(w))$ time per query, and $\Theta(u)$ space. However, we can decrease the amount of space using modified version of the Van Emde Boas data structure, $O(\log(w))$ time per query with high probability, and $\Theta(n)$ space. Y-fast trees also achieve the same bounds.

There are also fusion trees, which can achieve $O\left(\log_w(n)\right)$ time with high probability, and $\Theta(u)$ space. We will be covering these in lecture 12.

Note that the query time of a fusion tree may or may not be better than that of a Y-fast tree, depending on our values of w and n. If we chose the optimal data structure depending on these values, then it turns out that the query time is $O\left(\sqrt{\log(n)}\right)$ time with high probability, with $\Theta(u)$ space. This can be shown by a small calculation.

3.3 Cell Probe Model

The lower bounds with the cell probe model are $O(n \operatorname{poly}\log(n))$ space, and $\Omega\left(\min\{\log_w(n), \frac{\log(w)}{\log(\frac{\log(w)}{\log\log(n)})}\right)$ time. The latter bound takes the optimal of van Emde Boad and fusion trees. Specifically, van Emde Boas is optimal when $w = O(\operatorname{poly}\log(n))$, while fusion trees are optimal for when $w = 2^{\Omega\left(\sqrt{\log(n)}\log\log(n)\right)}$

3.4 Pointer machine model

Here, our word is specified by a pointer. Using van Emde Boas we can use $O(\log(\log(u)))$ time per operation, and $\Theta(u)$ space. There also exists a lower bound of $O(\log(\log(u)))$ time per operation, and $\Omega(u)$ space

4 Van Emde Boas Trees

Now we will describe the structure of a Van Emde Boas. The goal of this data structure is to be able to bound the running time of all operations by $T(u) \leq T(\sqrt{u}) + O(1)$. This way, we get using Master's theorem that $T(u) = O(\log \log u)$.

To start, if we are given a binary string x write high(x) for the first $|x|/2$ digits and low(x) for the last $|x|/2$ digits. When x is an integer with binary representation bin(x), we write c for the number whose binary representation is high(bin(x)) and i for the number whose binary representation is low(bin(x)). Then given an integer x, it is stored in block V.cluster[c] and within that block it is at position i.

A VEB, V, of size u with word size w consists of $\overline{(u)}$ VEB's of size $\overline{(u)}$ and word size $w/2$, each denoted by V.cluster[c]. If $x = \langle c, i \rangle$ is contained in V (and x is not the minimum), then i is contained in V.cluster[c]. In addition, we maintain a summary VEB V.summary of size $\overline{(u)}$ and word size $w/2$, which contains all integers c such that V.cluster[c] is not empty. Finally, each structure contains the minimum element, V.min (or None if it is empty), and a copy of the maximum element, V.max.

To answer a query Successor$(V, x = \langle c, i \rangle)$, we first check if x is less than or greater the minimum and maximum elements respectively. In this case, the minimum element is the successor, or there is no successor. Otherwise, we check to see if i is less than the maximum of V.cluster[c]. In this case, there exists an element greater than x in the same cluser as x, so we can just recursively search for the sucessor in that cluster. Otherwise, the successor of x is the min of V.cluster[c'], where c' is the successor of c in V.summary. In this case, we recursively find the successor of c in V.summary. Note that in both cases, we require a single search in a VEB of size $O(\overline{(u)})$. The query to find the predecessor works in a very similar fashion.

For the query Insert$(V, x = \langle c, i \rangle)$, we again start by checking the minimum and the maximum. If V is empty, (V.min is None), then we let both the min and the max be x, and we are done. If x is less than V.min, then we let V.min be x, and insert the old minimum. If x is greater than V.max then we replace this with a copy of x in V.max. Now, we check to see if V.cluster[c] is empty, or if there is no minimum. In this case, we recursively insert c into V.summary, and then insert i into V.cluster[c]. Inserting i will take constant time in this case because V.cluster[c] is empty at this point. Otherwise, we just recursively insert i into V.cluster[c].

Finally, to preform deletions, Delete$(V, x = \langle c, i \rangle)$, we start with the case in which x is the minimum element. If the max is also x, then we can just delete this element by setting V.min and V.max to None. Otherwise, we have to find the new min, which is $\langle$V.summary.min, V.cluster[V.summary.min].min$\rangle$ (note that the minimum is not stored elsewhere), delete it, and set it as the new V.min. To delete x if x is not V.min, we first recursively delete i from V.cluster[c].i. If V.cluster[c] is now empty, then this would have taken constant time, and now we must recursively delete c from V.summary. Finally, we must update V.max. If V.summary is empty, then there is only one element in V, and V.max must be set to V.min. Otherwise, V.max is now ⟨V.summary.max, V.cluster[V.summary.max].max⟩).

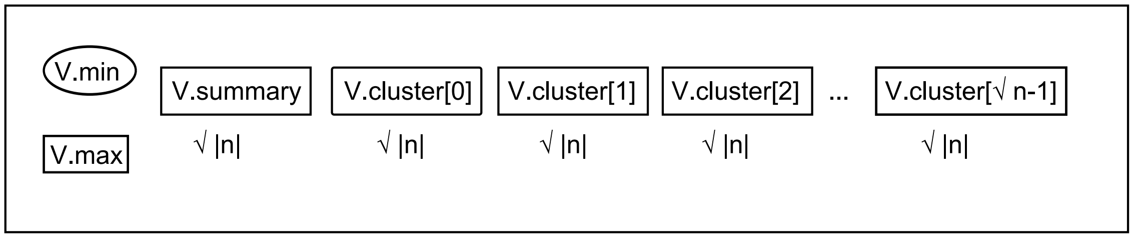

Figure 1: A visual representation of the recursive structure of a VEB. Note that at each level, the minimum is stored directly, and not recursively.

Theorem 1. *A VEB preforms the operations sucessor, predecessor, insertion and deletion in* $O(\log\log u)$ *time.*

Proof. The running time T in all cases satisfies $T(u) \leq T(\sqrt{u}) + O(1)$. Note that specifically, we only recurse once. It follows by the Master's theorem, that $T(u) = O(\log\log u)$. □

5 Binary tree view

One can also interpret the van Emde Boas structure as a binary tree built over the bit vector v where $v[i] = 1$ if i is in the structure and $v[i] = 0$ otherwise. This is how van Emde Boas presented the structure in his original paper [2]. At a given node, we store the OR of its two children, thus the value of a node determines if there exists in the structure some integer in the interval of that node. We will also store in every node the minimum and maximum of all integers stored in its interval. In addition, we can compute the k^{th} level node in the path to any given integer x in $O(1)$ time because that node corresponds to the k high-order bits of x.

This view gives us an algorithm to quickly compute the predecessor or successor of any given integer using the tree structure, assuming that our set is static. The key observation is that if we look at the values of the nodes along the path to a given integer x, the values form a monotone sequence of contiguous ones followed by zeros. Thus, we can do a binary search to find the $1 \to 0$ transition in the sequence, so let the final 1-value node be u and the first 0-value node be v. In the case that no transition exists, then we either have no integers in the structure or x is in the structure, both of which are trivial cases. If v is the left child of u, then we know that the successsor of x must lie in the right child of u and that the right child of u contains at least one integer, thus we can simply return the minimum stored at the right child of u. Similarly, if v is the right child of u, then we can obtain the predecessor of x by looking at the maximum stored in the left child of u. Finally, to obtain the successor from the predecessor or vice versa, we can store all elements of the structure in a sorted linked list.

5.1 Analysis

Because the binary tree has height w and the query time is dominated by the binary search on the path, this gives us an $O(\lg w)$ or $O(\lg\lg u)$ query time bound. However, to implement the binary search on a pointer machine, a stratified tree is required, thus storing at each node pointers to its ancestor of height 2^i for $i = 0, 1, ..., \lg w$, thus because we have $O(u)$ nodes we obtain space bound $O(u \lg w)$. In addition, updating this structure is very expensive, as we have to update all nodes along the path, for $O(\lg u)$ update time. However, as van Emde Boas notes [2], the same trick of not storing the minimum recursively can reduce the update time to $O(\lg\lg u)$, and this bound is optimal on a pointer machine.

5.2 Indirection

We now present a second way to reduce the update time on this tree structure through indrection. Suppose that instead of having our tree structure be over an set of integers, suppose we instead use a set of binary search trees, each of size $\Theta(w)$. In other words, we partition the set of integers in the structure into $\Theta(\frac{n}{w})$ parts and store each part in a binary tree. For a given BST, we can arbitrarily choose any integer stored in the tree as its representative element in the larger tree structure.

We allow the BST structures to have size in $[1/2w, 2w]$. Once a BST gets too large, we split it into two BSTs. Once a BST gets too small, we merge it with one of the adjacent BSTs and then potentially split the result. These splits and merges, as well as updating the top-level structure can all be done in $O(w)$ time, so we just need to show that this happens in at most one out of every $O(w)$ updates. However, after a split we can verify that a binary search tree has between $2/3w$ and $3/2w$ vertices, so does not need to split or merged again for another $O(w)$ operations. When we merge two trees, we pay in advance for the next time we will have to split the merged tree. Thus the total amortized time is $O(1)$ for each modification, so the time bound is dominated by the bottom-level update time, which is $O(\lg w) = O(\lg\lg u)$, as desired.

6 Reducing space

All structures given so far require $\Omega(u)$ space, but we wish to achieve the optimal $\Theta(n)$ space bound. We present two methods for achieving this bound as well as $O(\lg\lg u)$ update time with high probability through the use of perfect hashing.

6.1 Hash tables in vEB

Suppose in our original van Emde Boas structure, we do not store empty clusters and replace child arrays by dynamic perfect hash tables. By charging every hash table entry to the minimum in that cluster, every integer in the structure is charged at most twice by either its unique parent or entry in a summary structure, thus we obtain the optimal space bound of $\Theta(n)$. In addition, if we use a hash table with $O(1)$ query and update with high probability, this allows us to obtain query and update bounds of $O(\lg\lg u)$ with high probability.

6.2 X-fast trees

Both x-fast trees and y-fast trees are due to Willard [3]. While X-fast trees do not achieve the desired bounds, they are an intermediate step towards y-fast trees which do. The main idea is to only store the 1-value nodes in the simple tree view structure. For a given 1-value node in the tree, we can compute a corresponding bitstring by looking at the path of left-child and right-child steps from the root to the given node, with a left-child step corresponding to a 0 in the string and a right-child step corresponding to a 1. An x-fast tree consists a perfect hash table of all such bitstrings for all 1-value nodes in the tree, or all prefixes of bitstrings of integers in the structure. This table allows us to map a bitstring to the minimum and maximum values stored at that node in $O(1)$ time.

Note that we can look up the value of any given node in the tree by checking whether its bitstring is in the hashtable. Thus, we can still perform the binary search in $O(\lg w)$ time with high probability and achieve the same query bound. However, to perform an update, we still need to update all nodes along the path from the root, thus we have an $O(w)$ update bound. In addition, we store information for all w prefixes of every integer in the structure, for a worst case $O(nw)$ space bound.

6.3 Y-fast trees

To obtain Y-fast trees, we use the same indirection trick. We maintain a top-level x-fast tree structure of size $\Theta(\frac{n}{w})$ with the bottom level structures being binary search trees of size $\Theta(w)$. The $O(\lg w)$ query bound is maintained, but the top-level update can now be charged to the $\Theta(w)$ bottom-level updates, thus the update time is dominated by the $O(\lg w)$ time for the bottom-level BST structures. In addition, the space required for the top level structure is now $O(n)$, thus we achieve total space bound $O(n)$, as desired.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2009.

[2] P. van Emde Boas, *Preserving Order in a Forest in less than Logarithmic Time*, FOCS, 75-84, 1975.

[3] Dan E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space* $\Theta(n)$, Inf. Process. Lett. 17(2): 81-84 (1983)