6.897: Advanced Data Structures	Spring 2003

Lecture 3 — February 19, 2003

Prof. Erik Demaine — *Scribe: Sayan Mitra*

1 Topics Covered

1. Review of y-fast trees
2. RAMBO model
3. Finding the least significant 1 bit in a word
4. Van Emde Boas in $O(1)$
5. Decision-tree model, lower bound

2 Review of y-fast trees

The y-fast tree [5] is a data structure for the fixed-universe successor problem. It uses indirection as shown in Figure 1. Each bottom structure is a balanced binary search tree (e.g., AVL tree or red-black tree) storing $\Theta(\lg u)$ elements, where u is the size of the universe. The top (main) structure is a dynamic perfect hash table storing all $\lg u$ prefixes of each of the $\Theta(n/\lg u)$ representative elements (one per bottom tree), where n is the number of elements in the structure.

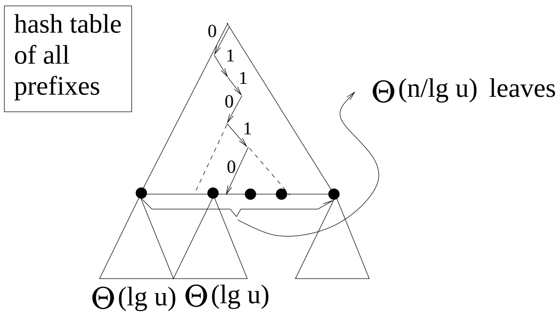

Figure 1: y-fast trees.

To find the successor and predecessor of x in a y-fast tree, we proceed as follows:

- Binary search on the length of a prefix match between x and a representative element, in $O(\lg\lg u)$ time.
- The hash table stores with that prefix the minimum and maximum elements of subtree rooted at the corresponding node in a binary tree.

- The prefix match narrows the search down to two bottom trees (because we learned that x is between two representative elements).
- Search in the two bottom trees in $O(\lg \lg u)$ time.

Updates work as follows:

- Update the bottom tree in $O(\lg \lg u)$ time.
- If a bottom tree grows to $2 \lg u$ elements, split it in half.
- If a bottom tree shrinks to $\frac{1}{4} \lg u$ elements, merge it with a neighbor (and then possibly resplit in half).
- In either case, $O(1)$ representative elements change, and the cost to update the top structure is $O(\lg u)$. We can charge this cost to the $\Theta(\lg u)$ insertions or deletions that caused the grow or shrink.
- Amortized cost of update is $O(\lg \lg u)$.

3 RAMBO model

The RAMBO model was introduced by Fredman and Saks [3]. RAMBO stands for Random Access Machine with Bit Overlap. Like the Transdichitomous RAM [4], a RAMBO has words of length $\geq \lg u$ bits, and the words can be manipulated in $O(1)$ time. Unlike the RAM, however, a RAMBO can have some of its words partially overlapping, i.e., sharing some bits. Figure 2 shows a simple example in which half of the words are rows of a $\lg u \times \lg u$ matrix, half of the words are columns of the matrix, and every row word and column word share exactly one bit.

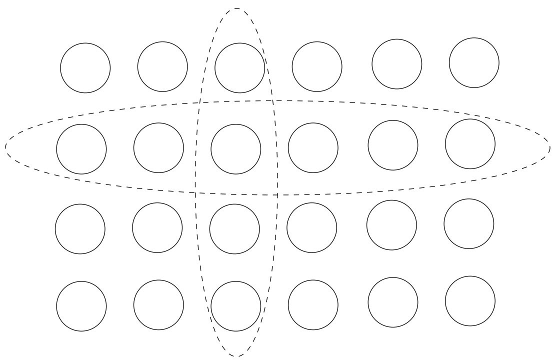

Figure 2: Matrix RAMBO.

This model will be used to solve the fixed-universe successor problem in constant time per operation.

4 Finding the least-significant 1 bit

Problem: Given a binary word w, find the index of the least-significant 1 bit.

Solutions: All of the following solutions take $O(1)$ time.

- An all-powerful RAM can do this by defining it as an instruction.
- This operation is an instruction on many real chips like Cray and Pentium.
- We can get a word containing just the least-significant 1 bit by computing $a = (w \texttt{ AND } w - 1) \texttt{ XOR } w$. For example:

$$\begin{aligned} w &= 01011101000 \\ w-1 &= 01011100111 \\ w \texttt{ AND } (w-1) &= 01011100000 \\ a = (w \texttt{ AND } (w-1)) \texttt{ XOR } w &= 00000001000 \end{aligned}$$

  The word a captures the information regarding the position of the least significant 1 bit in w. This computation can be done in $O(1)$ on a word of length $\lg u$.

  Instead of $a = 00000001000$, if we want the actual index 3 (11 in binary), then the problem can still be solved with $O(1)$ arithmetic and bitwise operations [1].

- For an easier, general solution to problems like this one, it is reasonable to build a lookup table.

  A First Attempt: There are u possible bit strings of length $\lg u$. For each such input, there are $\lg u$ possible indexes for the answer. Thus, we can store an answer using $O(\lg \lg u)$ bits. Therefore, a simple lookup table storing answers for all u inputs would take $O(u \lg \lg u)$ space, which is too much (superlinear).

  Solution: The space requirement can be dramatically reduced by a two-step lookup, that is, searching for a 1 in the lower half bits of the word and then in the higher half bits. Let $\mathrm{LSB}(x)$ be the function which returns the least significant 1 bit in the word x. Let $\mathrm{LSB}[\cdot]$ denote a lookup table access. We denote the lower and the higher half words of x by $\mathrm{low}(x)$ and $\mathrm{high}(x)$, respectively. Then,

$$\mathrm{LSB}(x) = \begin{cases} \mathrm{LSB}[\mathrm{low}(x)] & \text{if } \mathrm{low}(x) \neq 0 \\ \mathrm{LSB}[\mathrm{high}(x)] + \lg u/2 & \text{otherwise} \end{cases}$$

  Each call to $\mathrm{LSB}(\cdot)$ involves one lookup which can be done by a single random access. Now it is sufficient to have a table of answers for all possible half words with $(\lg u)/2$ bits. Such a table has $2^{\lg u/2} = \sqrt{u}$ entries. Therefore the space requirement is now only $O(\sqrt{u} \lg \lg u) = o(u)$ bits. The lookup still takes $O(1)$ time.

5 Van Emde Boas in O(1) in the RAMBO model

Brodnik, Carlsson, Karlsson, and Munro [2] showed how to solve the fixed-universe successor problem in $O(1)$ time in the RAMBO model. There are $2u$ bits of RAMBO storage, one for each node in a complete binary tree (see Figure 3).[1] The words in the RAMBO consist of the bits corresponding to a root-to-leaf path in the binary tree.

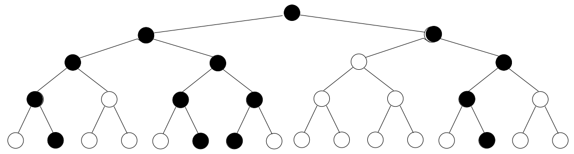

Figure 3: Binary tree with the dark nodes representing present elements and their ancestors.

Suppose we set a bit to a 1 if the corresponding node is an ancestor of a present element, as in Figure 3. As described in the previous section, given a query word $w \in \mathcal{U}$, we can find the lowest set ancestor of w in $O(1)$ time. In order to get the successor, we could maintain a pointer from each node to the min and the max elements in the subtree rooted at that node, as in Figure 4. But then each leaf may have upto $2 \lg u$ pointers to it, and therefore Insert and Delete would require $\Omega(\lg u)$ pointer updates.

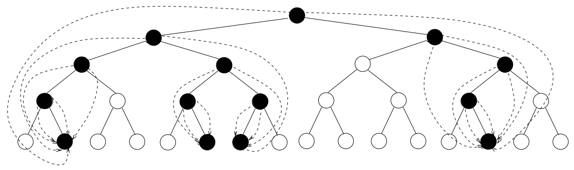

Figure 4: Binary tree with nodes storing pointers to min and max in their subtress.

Instead, we store left and right pointers in a node only if the pointers are different from its parent. More precisely, we store pointers at each *branching node*, i.e., a node with elements present in both the left and right subtrees. Each branching node stores a pointer to the maximum element in its left subtree and the minimum element in its right subtree. See Figure 5.

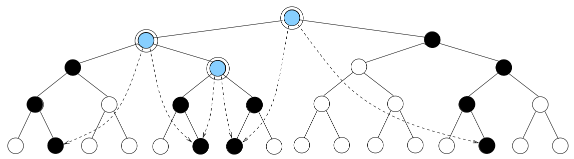

Figure 5: The same binary tree with pointers only from the branching nodes (light nodes with an outer circle).

Now the predecessor and successor of an element of the universe can be found by the two pointers stored at the lowest branching ancestor. The data structure takes only u bits of RAMBO space and $2n$ words of RAM for the pointers.

[1] In fact, there are only u bits of RAMBO storage, because the leaves do not have to store their bit.

One detail remains: how to map a node in the RAMBO tree to the two corresponding pointers stored in RAM space. This mapping can be done by storing the RAM pointers in a dynamic perfect hash table, keyed by the position in the tree. A position in the tree can be represented by a one-word bit string to specify the root-to-node path together with the length of that path. Such a position representation can be computed in constant time, and hence the hash table does the job.

Incidentally, this RAMBO structure has been implemented in real hardware as an SDRAM memory module.

6 Outside fixed universes: Decision-tree model

In this model, the algorithm is viewed as a tree. The nodes represent any computation except for branching, and the branches emerging from a node represent the decisions. Each node is allowed to have only $O(1)$ branches. The length of a root-to-leaf path is the running time of a particular instance of the algorithm. The worst-case running time of the algorithm is the height of the tree.

This model is similar to the comparison model, except that the latter allows only comparison operations in the nodes. Thus, the comparison model is strictly weaker than the decision-tree model.

What kind of computation is not captured by this model? Random access. With random access, the computation can perform multi-way branching based on the input read, whereas this model allows only $O(1)$-way branching.

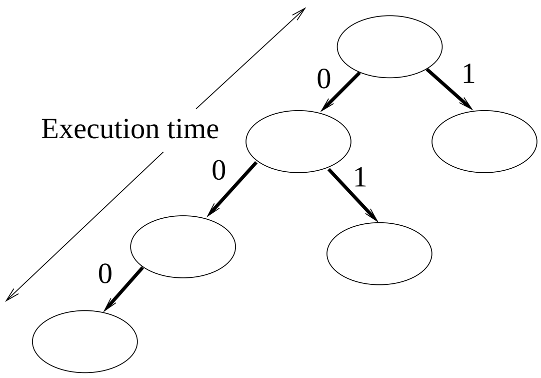

Figure 6: Decision tree model. The nodes represent any finite computation.

6.1 Lower bound

Theorem 1 *Any data structure implementing Insert and Successor requires $\Omega(\lg n)$ amortized time in the worst case, in the decision tree model.*

Proof: The proof is by reduction from $Sort(a_1, a_2, \dots, a_n)$:

$$Insert(a_1)$$

$Insert(a_2)$
$\dots$
$Insert(a_n)$
$print\ x = Successor(-\infty)$
$print\ x = Successor(x)$
$\dots$
$print\ x = Successor(x)$

The n *Successor* calls produce the sorted list of the elements $a_1, a_2, \dots, a_n$. Therefore $Sort \leq Insert\&Successor$. For sorting n elements, there are $n!$ possible permutations, so there are at least $n!$ leaves of the tree. Thus, the height of the tree is at least $\sim n \lg n$, so sorting requires $\Omega(n \lg n)$ time.

7 Introduction to fusion trees

Once again the desired operations on the data structure are *Insert, Delete, Successor* and *Predecessor*, but now we consider the case that the universe $\mathcal{U}$ is very big, so $\Theta(\lg \lg u)$ is too large. We want a running time of $o(\lg n)$.

The fusion tree [4] is a B-tree with branching factor of $\Theta(\lg n)^{1/5}$. It is able to visit a node, and decide which branch to follow, in constant time. Therefore, the time to perform a search is proportional to the height of the tree, $O\left(\frac{\lg n}{\lg \lg n}\right)$.

References

[1] Andrej Brodnik. Computation of the least significant set bit. In *Proceedings of the 2nd Electrotechnical and Computer Science Conference*, Portorož, Slovenia, 1993.

[2] Andrej Brodnik, Svante Carlsson, Johan Karlsson, and J. Ian Munro. Worst case constant time priority queue. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, Washington, DC, 2001, pages 523–528.

[3] Michael L. Fredman and Michael E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the 21st ACM Symposium on Theory of Computing*, Seattle, Washington, 1989, pages 345–354.

[4] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, 1993.

[5] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.