**6.897: Advanced Data Structures** Spring 2005

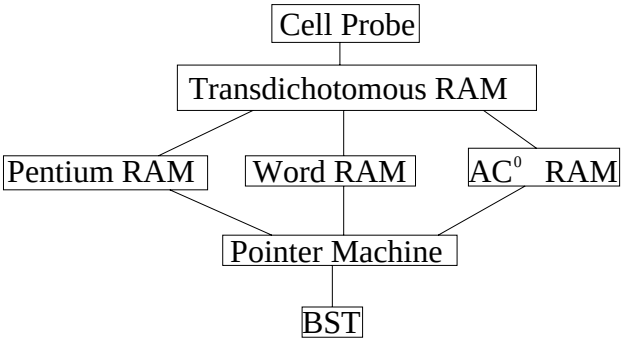
Lecture 9 — March 3, 2005

*Prof. Erik Demaine* *Scribe: Michael Lieberman*

# 1 Overview

Today we start the topic of integer data structures. First we really need to specify our model of computation. Then we will be able to beat the usual $O(\lg n)$ bounds for searching, by using van Emde Boas and y-fast trees.

# 2 Models of Computation




The *cell probe model* at the top is the strongest model, where we only pay for accessing memory (reads or writes). Memory cells have some size $w$, which is a parameter of the model. The model is nonuniform, and allows memory reads or writes to depend arbitrarily on past cell probes. Though not realistic to implement, it is good for proving lower bounds.

The *transdichotomous RAM* tries to model a realistic computer. We assume $w \geq \lg n$; this means that the "computer" changes with the problem size. However, this is actually a very realistic assumption: we always assume words are large enough to store pointers and indices into the data, or otherwise, we cannot even address the input. Also, the algorithm can only use a finite set of operations to manipulate data. Each operation can only manipulate $O(1)$ cells at one time. Cells can be addresses arbitrarily; "RAM" stands for Random Access Machine, which differentiates the model from classic, but uninteresting computation models such as a tape-based Turing machine.

Depending on which operations we allow, we have several instantiations of the transdichotomous RAM:

- *word RAM* – Only "C-style" operators, like: `+ - * / % & | ^ ~ << >>`.
- *$AC^0$ RAM* – The operations must have an implementation by a constant-depth, unbounded fan-in, polynomial-size (in $w$) circuit. This is the most restrictive definition of what "unit

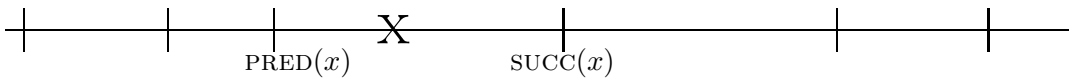
time" can mean. Practically, it allows all the operations of word RAM except for multiplication in constant time.

- *Pentium RAM* – While interesting practically, this model is of little theoretical interest as it tends to change over time.

# 3 Predecessor problem

Also known as the *successor problem*, the goal is to maintain a set $S$ of $n$ items from an ordered universe $\mathcal{U}$ of size $u$. The elements are integers that fit in a machine word, so $u = 2^w$. We will use the Word RAM model, and our data structure must support the following operations:

- INSERT$(x)$, $\quad x \in \mathcal{U}$
- DELETE$(x)$, $\quad x \in S$
- SUCCESSOR$(x)$, $\quad x \in \mathcal{U}$
- PREDECESSOR$(x)$, $\quad x \in \mathcal{U}$




As you can see, we have some universe $\mathcal{U}$ (the entire line) and some set $S$ of points. It is important to note that the PREDECESSOR and SUCCESSOR functions can take any element in $\mathcal{U}$, not just elements from $S$.

## 3.1 Classic Results

These are the classic solutions to the predecessor problems up to about 1993. Today we present van Emde Boas and y-fast trees. Next lecture we will cover fusion trees. In the following lectures, we will discuss more recent results not mentioned in this table.

| | data structure | time/op | space | model |
|---|---|---|---|---|
| 1962 | balanced trees | $O(\lg n)$ | $O(n)$ | BST |
| 1975 | van Emde Boas [1] | $O(\lg w) = O(\lg \lg u)$ | $O(u)$ | word / $AC^0$ RAM |
| 1984 | y-fast trees [2] | $O(\lg w)$ w.h.p. | $O(n)$ | word RAM |
| 1993 | fusion trees [3] | $O(\lg_w n) = O\left(\frac{\lg n}{\lg \lg u}\right)$ | $O(n)$ | word RAM; also $AC^0$ RAM [4] |

## 3.2 Combination of classic results

Fusion trees work well when the size of the universe is much bigger than the size of the data ($u >> n$), while van Emde Boas and y-fast trees work well when the size of the universe is much closer to the size of the data. We can therefore choose which data structure to use depending on

the application. They are equivalent around when $\Theta(\lg w) = \Theta\left(\frac{\lg n}{\lg w}\right)$, so that is the worst case in which we would use either data structure. In this worst case, we notice that $\lg w = \Theta(\sqrt{\lg n})$, so we can always achieve $O(\sqrt{\lg n})$, which is significantly better than the simpler BST model.

# 4 van Emde Boas

The $\lg w$ bound intuitively suggests a binary search on the bits of a number, and that is essentially what van Emde Boas does. Suppose $x$ has $w$ bits. Then we split $x$ into 2 parts: high($x$) and low($x$), each with $\frac{w}{2}$ bits. For example: let $x = 0110011100$. Then we have something that looks like this:

| high(x) | low(x) |
|---|---|
| 01100 | 11100 |

## 4.1 van Emde Boas structure

To handle a universe of size $u$, we will create $\sqrt{u}+1$ substructures. These are recursively constructed in the same way (they are van Emde Boas structures themselves).

- $\sqrt{u}$ substructures: $(S[0], S[1], \ldots, S[\sqrt{u}-1])$. Each substructure handles a range of size $\sqrt{u}$ from the universe. A key $x$ is stored in $S[high(x)]$; once we're down in this structure, we only care about $low(x)$.
- A single substructure $S.summary$ of size $\sqrt{u}$. For every $i$ such that $S[i]$ is nonempty, we store $i$ in the summary structure.
- $S.min$ = the minimum element of the set. This is not stored recursively (i.e. we ignore the minimum element when constructing the substructures). Note that we can test to see if $S$ is empty by simply checking to see if some value is stored in $S.min$.
- $S.max$ = the maximum element of the set; unlike the min, this is also stored recursively.

## 4.2 Algorithm

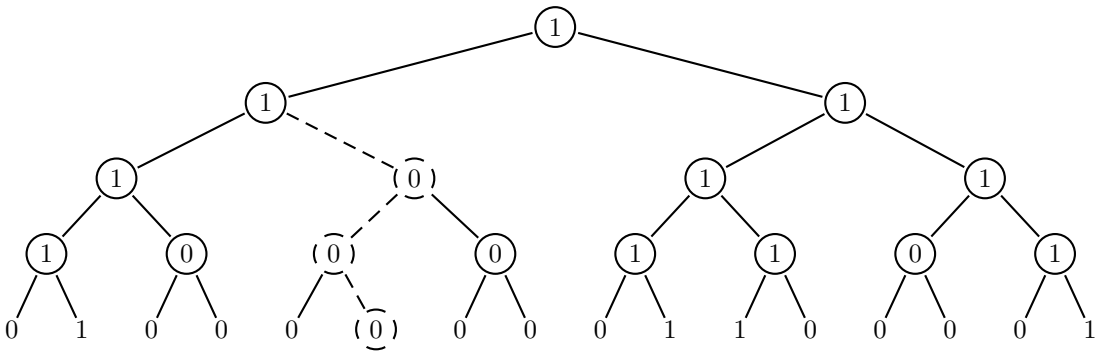
We will work through SUCCESSOR and INSERT, to show that they can be implemented with only a single recursive call. PREDECESSOR and DELETE work similarly.

```
SUCCESSOR(x, S)
1  if x < S.min
2     then return S.min
3  if low(x) < S[high(x)].max
4     then return high(x) · √u + SUCCESSOR(low(x), S[high(x)])
5     else  i ← SUCCESSOR(high(x), S.summary)
6           return i · √u + S[i].min
```

```
INSERT(x, S)
 1  if S.min = φ
 2     then S.min ← S.max ← x
 3          return
 4  if x > S.max
 5     then S.max ← x
 6  if x < S.min
 7     then SWAP(x, S.min)
 8  if S[high(x)].min = φ
 9     then S[high(x)].min ← S[high(x)].max ← low(x)
10          INSERT(high(x), S.summary)
11     else INSERT(low(x), S[high(x)])
```

In each step, SUCCESSOR recurses to a subtructure in which integers have half as many bits. Thus, it takes $O(\lg w)$ time. INSERT accomplishes the same goal, but in a more subtle way. The crucial observation is that either $S[high(x)]$ was already nonempty, so nothing in the summary changes and we just recurse in $S[high(x)]$, or $S[high(x)]$ was empty, and we need to insert $high(x)$ recursively in the summary, but insertion into $S[high(x)]$ is trivial (so we don't need a second recursive call, which would blow up the complexity). Note that it is essential that we don't store the min recursively, to make inserting into an empty structure trivial.

# 5 y-fast trees




The above tree represents a trie over a universe of size $u = 16$. Each element is viewed as a root-to-leaf path, which gives the bits of the number in order. Nodes which are on an active root-to-leaf path (corresponding to some element in $S$) are marked by 1. Alternatively, a node is set to 1 iff a leaf under it corresponds to an element in $S$.

Intuitively, if we are looking for the predecessor or successor of an element, all we have to do is walk up the tree (as shown by the dashed lines above) until we hit a 1. It is then simple to walk back down, and if we ever have a choice, we take the path closer to where we started.

After we find either the predecessor or the successor this way, we can simply maintain a sorted doubly linked list of elements of S to find the other one.

Starting from this intuition, y-fast trees do the following:

- store the location of all of the 1 bits of the tree in a hash table (using dynamic perfect hashing) i.e. store all prefixes of the binary representation of $x \in S$
- binary search to find the longest prefix of $x$ in the hash table i.e. the deepest node with a 1 that is on

the path from $x$ to the root.

- look at the min or max of the other child to find either the successor or predecessor.
- use a linked list on $S$ to find the other.

Unfortunately, the above implementation of y-fast trees leaves us with the problem that updates take $O(w)$ time (they may need to insert $w$ nodes in the hash table), and the hash table takes $O(nw)$ space. To fix this, we use a method called indirection or bucketing, which gives us a black box reduction in space and update time for the predecessor problem.

# 6 Indirection

- cluster elements of $S$ into consecutive groups of size $\Theta(w)$ each
- store elements of a clusted in a balanced BST.
- Maintain a set of *representative* elements (one per cluster) stored in the y-fast structure described above. These elements are not necessarily in $S$, but they separate the clusters. A clusters's representative is between the maximum element in the cluster (inclusively) and the minimum element in the next cluster (exclusively).

Since we only have $O(\frac{n}{w})$ representative elements to keep track of, the space of the y-fast structure is $O(n)$. The space taken by the BSTs is $O(n)$ in total.

To perform a predecessor search, we first query the y-fast structure to find the predecessor among representatives. Given the representative, there are only 2 clusters to search through to find the real predecessor (either the one containing the representative, or the succeeding one). We can now search through these BSTs in $O(\lg w)$ time. Thus, the total time bound is $O(\lg w)$, as before.

Now we discuss insertions and deletions. We first find the cluster where the element should fit, in $O(\lg w)$ time. Then, we insert or delete it from the BST in time $O(\lg w)$. All that remains is to ensure that our clusters stay of size $\Theta(w)$. Whenever a bucket grows too large (say, $\frac{2n}{w}$), we split it in two. If a bucket gets too small (less than $\frac{n}{2w}$), merge it with an adjacent bucket (and possibly split the resulting bucket if it is now too large).

Any time we split or merge we have to change a constant number of representative elements. This takes $O(w)$ time – the update time in the y-fast structure. In addition, we need to split or merge two trees of size $\Theta(w)$, so the total case is $O(w)$. However, merging or splitting only happens after $\Omega(w)$ operations that touch the cluster, so it is $O(1)$ time amortized.

## 6.1 General Indirection Result

Note that our indirection result did not depend on any particular propert of the y-fast structure, and it is a general black box transformation. Given a solution to the predecessor problem with $O(\lg w)$ query time, $n \cdot w^{O(1)}$ space, and $w^{O(1)}$ update time, we obtain a solution with the same query time, $O(n)$ space and $O(\lg w)$ update time.

# References

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

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

[3] M. Fredman, D. E. Willard, *Surpassing the Information Theoretic Bound with Fusion Trees*, J. Comput. Syst. Sci, 47(3):424-436, 1993.

[4] A. Andersson, P. B. Miltersen, M. Thorup, *Fusion Trees can be Implemented with* $AC^0$ *Instructions Only*, Theor. Comput. Sci, 215(1-2): 337-344, 1999.