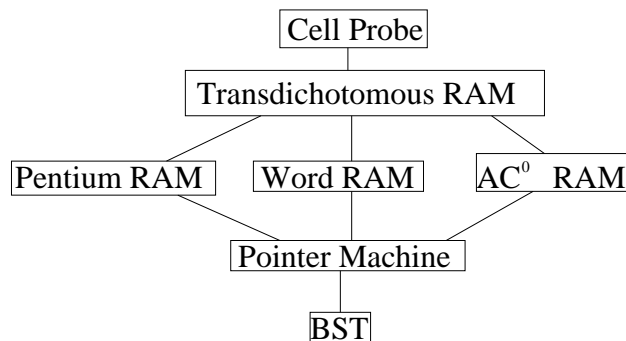


1 Overview

Last lecture we saw how to perform membership queries in $O(1)$ time using hashing. Today we start the topic of integer data structures. We first specify the model of computation we are going to use and then beat the usual $O(\lg n)$ bounds for *insert/delete* and *successor/predecessor* queries using van Emde Boas and y-fast trees. We assume that the elements – inputs, outputs, memory cells – are all w bit integers where w is the word size. We also assume a fixed size universe $\mathcal{U} = \{0, 1, \dots, u - 1\}$ where $u = 2^w$.

2 Models of Computation



Cell Probe Model: This is the strongest model, where we only pay for accessing memory (reads or writes) and any additional computation is free. Memory cells have some size w , which is a parameter of the model. The model is non-uniform, 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.

Transdichotomous RAM Model: This model 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, since 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 a 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, there are several instantiations of the *Transdichotomous RAM Model*:

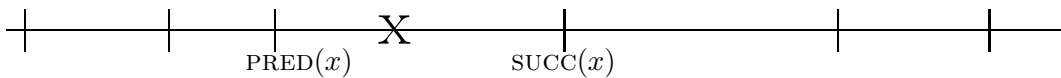
- *Word RAM*: In this model we have *TransdichotomousRAMModel* + $O(1)$ “C-style” operations like $+ - * / \% \& | \wedge \sim \ll \gg$. This is the model we are going to use today.
- *AC⁰ RAM*: The operations in this model must have an implementation by a constant-depth, unbounded fan-in, polynomial-size (in w) circuit. 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.

Pointer Machine Model: In this model the data structure is described by a directed graph with constant branching factor. For the fixed universe there is a pointer to each element in the universe \mathcal{U} . The input to an operation is just one of these pointers.

3 Successor/Predecessor 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, that is, $u = 2^w$. Our data structure must support the following operations:

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



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 . For our model of computation we use the *Word RAM Model* described above.

3.1 Classic Results

	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]

These are the classic solutions to the *successor/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.

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 or $\lg \lg u \geq \sqrt{\lg n}$, 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, or $\lg \lg u \leq \sqrt{\lg n}$. 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

There are two ways to obtain $\lg \lg u$ bound. The most intuitive one is to do a binary search over the bits of the word which results in $O(w) = O(\lg \lg u)$. The other way is to get a recursion of the form $T(u) = T(\sqrt{u}) + O(1)$, which also gives $T(u) = O(\lg \lg u)$. We are going to do a binary search on bits of query word x .

Suppose x has w bits. 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:

$$binary(x) = \underbrace{01100}_{high(x)} \underbrace{11100}_{low(x)} \quad x \in S$$

Low-order bits ($low(x)$) distinguish \sqrt{u} consecutive items. High order bits ($high(x)$) distinguish \sqrt{u} of these clusters.

4.1 van Emde Boas structure

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

- \sqrt{u} substructures: $(S[0], S[1], \dots, S[\sqrt{u} - 1])$. 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} . $S.summary[i] = 1 \Leftrightarrow S[i]$ is non-empty.

INSERT(x, S)

- 1 INSERT($low(x)$, $S[high(x)]$)
- 2 INSERT($high(x)$, $S.summary$)

If we do the analysis of this algorithm we get the following recursion: $T(u) = 2T(\sqrt{u}) + O(1)$. This recursion does not give us the complexity we want, it is only $T = \lg u$. We need to get rid of the factor 2. For that we introduce two additions.

- $S.min$ = 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 there is some value stored in $S.min$.
- $S.max$ = the maximum element of the set; unlike the min, this is stored recursively.

In the next section we see how the algorithm works.

4.2 Algorithm

We give the algorithms SUCCESSOR and INSERT, and show that they can be implemented with only a single recursive call. PREDECESSOR and DELETE work similarly.

For an insertion of element x to $S[i]$, one of the following two is true:

- 1) $S[i]$ was empty and just set $S[i].min = x$
- 2) $S.Summary[i]$ was already 1, no need to update.

```

INSERT( $x, S$ )
1  if  $S.min = \phi$ 
2    then  $S.min \leftarrow x$ 
3    return
4  if  $x < S.min$ 
5    then SWAP( $x, S.min$ )
6  if  $S[high(x)].min = \phi$ 
7    then  $S[high(x)].min \leftarrow low(x)$ 
8        INSERT( $high(x), S.summary$ )
9  else INSERT( $low(x), S[high(x)]$ )
10 if  $x > S.max$ 
11   then  $S.max \leftarrow x$ 
12

```

```

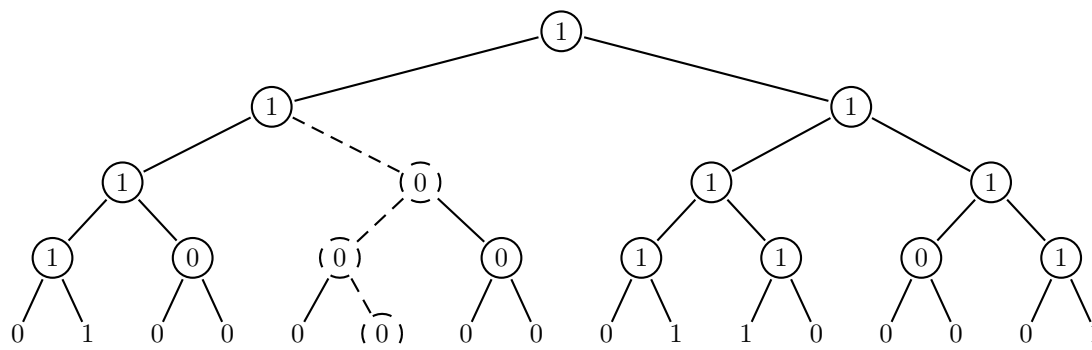
SUCCESSOR( $x, S$ )
1  if  $x < S.min$ 
2    then return  $S.min$ 
3  if  $low(x) < S[high(x)].max$ 
4    then //successor is in  $S[high(x)]$ 
5        return  $high(x) \cdot \sqrt{u} + \text{SUCCESSOR}(low(x), S[high(x)])$ 
6  else  $i \leftarrow \text{SUCCESSOR}(high(x), S.summary)$ 
7    return  $i \cdot \sqrt{|S|} + S[i].min$ 

```

In each step, SUCCESSOR recurses to a substructure 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.

This data structure is very good except the space is $O(u)$. Now, we will see another data structure that is as fast as van Ende Boas (expected) but occupies only $O(n)$ space, which is the size of the set S .

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 $O(\lg u)$ time (they may need to insert $\lg u$ nodes in the hash table), and the hash table takes $O(n \lg u)$ 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 $\Theta(\lg u)$ each
- store elements of each cluster 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 cluster's representative is between the maximum element in the cluster (inclusively) and the minimum element in the next cluster (exclusively).

Since we only have $O(\frac{n}{\lg u})$ 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 $O(\lg \lg u)$ time. Thus, the total time bound is $O(\lg \lg u)$, as before.

Now we discuss insertions and deletions. We first find the cluster where the element should fit, in $O(\lg \lg u)$ time. Then, we insert or delete it from the BST in time $O(\lg \lg u)$. All that remains is to ensure that our clusters stay of size $\Theta(\lg u)$. Whenever a BST grows too large (say, $\frac{2n}{\lg u}$), we split it in two. If a BST gets too small (less than $\frac{n}{4\lg u}$), merge it with an adjacent bucket (and possibly split the resulting BST if it is now too large).

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

6.1 General Indirection

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