6.897: Advanced Data Structures		Spring 2003
	Lecture 1 — February 9, 2003	
Prof. Erik Demaine		*Scribe: Shantonu Sen*

1 Fixed-Universe Successor Problem

1.1 Motivation

Frequently, you need to store a dynamic set of n integers such that you can perform fast lookups to determine whether an element is part of the set (i.e. a *membership* test). This problem can be solved in constant time per operation using hashing, as we'll see in the next lecture. Two related pieces of information is the *successor* and *predecessor* of an integer (not necessarily in the set), that is, the two elements in the set that are immediately greater than and less than the integer, respectively. The successor and predecessor of an integer are particularly useful when the integer is not in the set, because they designate where that integer would "fit" if it were in the sorted set.

The classic solution to this problem is to maintain a balanced binary tree of the integers in your set. The membership test for a binary balanced tree can be performed in $O(\lg n)$ via binary search. The predecessor and successor functions can also be computed in $O(\lg n)$ time, by first attempting to check membership of the search element, and then moving up the tree and "leftwards" (as formalized later) for the predecessor, and moving up the tree and "rightwards" for successor.

One way to make the problem easier is to impose that the universe is not the set of all integers, but integers from 0 to $u-1$. This assumption is called the *fixed-universe assumption*. Together with the operations described above, the problem is called the *fixed-universe successor problem* or *fixed-universe predecessor problem*. This problem is also referred to as *Interval Union-Split-Find* [MNA88] and as *priority queues* [vEKZ77].

1.2 Formal definition

We would like to create a data structure with the follow properties:

Goal: Maintain a dynamic subset S of size n from the universe $\mathcal{U} = \{0, 1, 2, 3, \ldots, u-1\}$ of size u

Supported operations:

- Insert($x \in \mathcal{U} \notin S$): add an element to S
- Delete($x \in S$): remove an element from S
- Successor($x \in \mathcal{U}$): find the smallest element $\in S$ that is $> x$
- Predecessor($x \in \mathcal{U}$): find the largest element $\in S$ that is $< x$

Desired performance: Better than $O(\lg n)$ for all operations, with the time bound possibly depending on u. Traditional balanced binary tree operate in $O(\lg n)$ without the fixed-universe assumption. In particular, can we achieve $O(\lg \lg u)$?

1.3 Known results

The standard solution to successor problem uses a balanced binary search trees, with a running time of $O(\lg n)$ per operation, using the comparison model on a pointer machine without the fixed-universe assumption. There are several other solutions, depending on the model of computation we consider.

1.3.1 Models

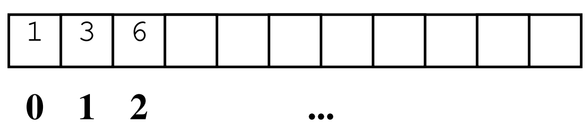
We will consider many different models of computation during this class. For starters, here are the three that have been studied for the fixed-universe successor problem:

Pointer-machine model. On a pointer machine, the data structure is described by a directed graph, where each node stores a constant number of labeled outgoing pointers and a constant number of integers. In other words, you have a constant branching factor. For the fixed-universe successor problem, there is a pointer to each element in the universe $\mathcal{U}$, and the input to an operation is one of these pointers.

Random Access Machine (RAM). In a RAM, memory is laid out as a finite array of slots. If you know the index of an entry, you can jump to its location and do a memory access in $O(1)$ time.

For example, below is a representation of a memory store that holds the numbers 1, 3, and 6 in the first 3 slots. Memory addresses can be loaded from and stored to, and the results of loads can be combined with arithmetic or logical operators.

The cost of an algorithm using a RAM is linear in the number of total instructions. That is, both memory accesses and arithmetic/logical operations cost $O(1)$ time.



Cell-probe model. The cell-probe model is just like a RAM, except that arithmetic/logical computation is "free". In this model, an algorithm's cost is linear in the number of memory accesses. This model is rather unrealistic, but is commonly used for lower bounds; any lower bound on the cell-probe model also applies to the RAM.

1.3.2 Results

Here are several upper and lower bounds for the successor and fixed-universe successor problem, and their models of computation.

- Balanced binary search trees
  - Comparison model using a pointer machine (no fixed-universe assumption)

- $O(\lg n)$ time per operation
- $O(n)$ space

- van Emde Boas [vEKZ77, vEB77] (this lecture)
  - Fixed-universe model on pointer machine (but we will describe the algorithm as working on a RAM)
  - $O(\lg\lg u)$ time per operation
  - $O(u)$ space

- Lower bound by Mehlhorn, Näher, and Alt [MNA88]
  - Pointer machine
  - Lower bound of $\Omega(\lg\lg u)$ time per operation

- y-fast trees [Wil83]
  - Randomized algorithm on a RAM
  - $O(\lg\lg u)$ time per operation
  - $O(n)$ space

- Lower bound by Beame and Fich [BF02]
  - Cell-probe model
  - Static case, no Insert/Delete
  - Lower bound of $\Omega\left(\min\left\{\frac{\lg\lg u}{\lg\lg\lg u}, \sqrt{\frac{\lg n}{\lg\lg n}}\right\}\right)$ time per operation, for any data structure using only $O(n^{O(1)})$ space

- Exponential search trees [AT99]
  - RAM model
  - $O\left(\min\left\{\frac{\lg\lg u}{\lg\lg\lg u}\lg\lg n, \sqrt{\frac{\lg n}{\lg\lg n}}\right\}\right)$ worst-case time per operation

2 $O(\lg\lg u)$ solution: van Emde Boas structure

Our goal for this lecture is to achieve $O(\lg\lg u)$ running time. Based on existing techniques for analyzing asymptotic running time of algorithms, we have some intuition about how we might end up with this type of running time.

One approach of traversing a data structure might involve doing binary search over $O(\lg u)$ things. Because a binary search runs in logarithmic time, this would yield performance of $O(\lg\lg u)$.

Another possibility is creating a recurrence relation whose solution is $O(\lg \lg u)$. For instance, consider the relation:

$$
\begin{aligned}
T(u) &= T(\sqrt{u}) + O(1) \\
T'(\lg u) &= T'(\lg \sqrt{u}) + O(1) \qquad \text{— } \textit{let } T'(\lg v) = T(v) \\
T'(\lg u) &= T'(\frac{1}{2} \lg u) + O(1) \qquad \text{— } \textit{pull out the square root} \\
T'(x) &= T'(\frac{1}{2} x) + O(1) \qquad \text{— } \textit{substitute } x = \lg u \\
T' &\text{ is } O(\lg x) \qquad \text{— } \textit{using Master Method} \\
T' &\text{ is } O(\lg \lg u) \qquad \text{— } \textit{substitute for } x = \lg u
\end{aligned}
$$

The van Emde Boas structure will employ the second technique to achieve $O(\lg \lg u)$ running time. In a certain sense, it will also correspond to binary searching over the $\Theta(\lg u)$ levels of a complete binary tree on u leaves.

2.1 Starting point: precompute answers

One technique for solving the problem is to use an array to store the successor value for every element x in the universe $\mathcal{U}$. Similarly, store the predecessors to support Predecessor queries. For example, if our set contains the elements $\{1, 3, 7\}$, and we wish to store the successors for any possibly query element x, our array would look like:

1	3	3	7	7	7	7	...			
0	1	2	3	4	5	6	7			u-1

This makes Successor and Predecessor queries very fast, $O(1)$ time, using only a single load to fetch the answer. But updates are slow, $\Theta(n)$ time in the worst case, because we may need to update many slots to point to the newly inserted element.

2.2 Starting point: store a bit vector

Another approach is to use a RAM to store a bit vector of the elements present in the set. If the set has the elements $\{2, 3, 6, 8, 9\}$, then the RAM looks like:

0	0	1	1	0	0	1	0	1	1	
0	1	2	3							u-1

Now, queries are slower, $O(n)$ time, requiring a linear search for the next present element. However, updates are now fast, $O(1)$ time, because we just have to modify the slot corresponding to the element being added or removed.

2.3 Improving bit-vector search time with a binary tree

Using the bit-vector representation as a starting point, we build up a binary tree of OR relations. This will let us know at each node whether there are any present elements in a subtree.

To find a successor, move up the tree until you enter a node from the left and there is a 1 on the right branch. Then go down the right branch, staying as close to the left as possible while following 1 branches, until you find an element. The running time is therefore $O(\lg u)$, as it is for updates.

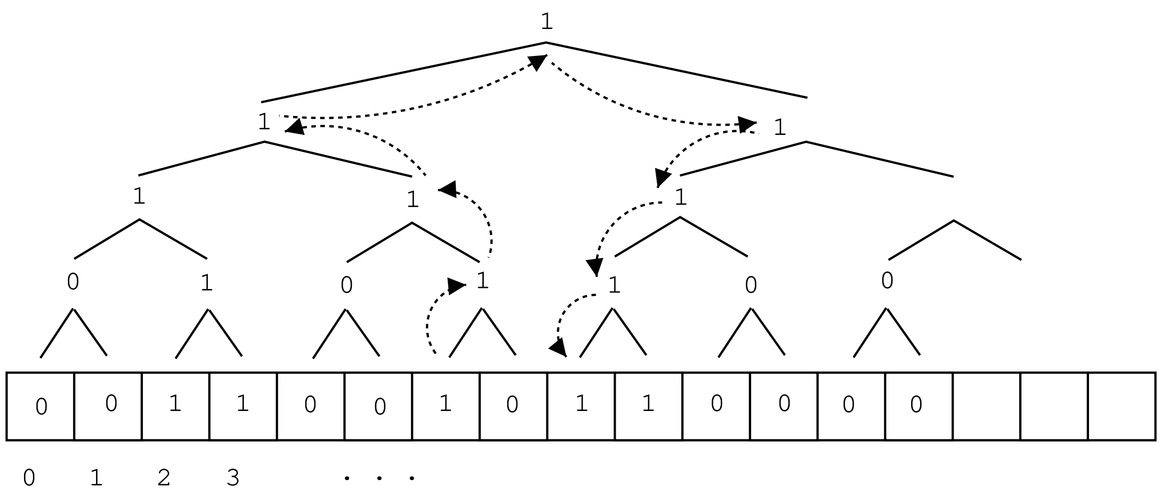


Figure 1: Bit-vector with OR tree

2.4 Variation: Use trees of constant height

Instead of a strictly binary tree, cluster the bit vector into $\sqrt{u}$ groups of size $\sqrt{u}$. Let us call each cluster sub[0], sub[1], $\cdots$, sub[$\sqrt{u}-1$]. Because each cluster has $\sqrt{u}$ elements, sub[i] represents the elements $\{i\sqrt{u}, i\sqrt{u}+1, \cdots, (i+1)\sqrt{u}-1\} \in \mathcal{U}$.

When searching for a successor, start out in the cluster representing your query element. Do a linear search within that cluster for a successor, and if one is not present, look only at the "emptiness" summary bits for the subsequent clusters to find the next non-empty cluster. Once such a cluster is found (if there is a successor at all in the set), do a linear search within that cluster, and we are guaranteed to succeed.

Queries take $O(\sqrt{u})$ time, because we do a linear search of up to $\sqrt{u}$ elements in two clusters, and a linear search of up to $\sqrt{u}$ emptiness summary bits. However, inserts take $O(1)$ time, because they just need to update a bit in a cluster and possibly the emptiness summary bit of that tree.

2.5 Bit manipulation: Helper functions

Going forward, it will be useful to have some tools for manipulating the binary representation of an element $x \in \mathcal{U}$, which uses $\lceil \lg u \rceil$ bits. For example, we might express 55 in a universe of size 256 as 00110111_2. Two functions we will want are:

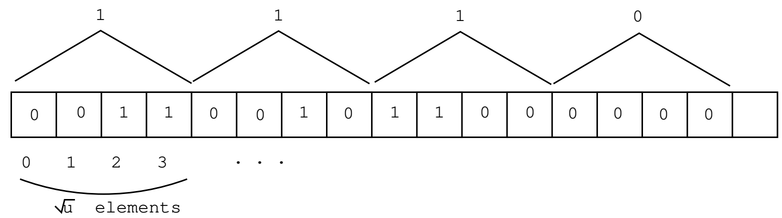


Figure 2: Bit-vector grouped into trees of size $\sqrt{u}$

- high(x) = high-order half of bits (0011_2 for the example)
- low(x) = low-order half of bits (0111_2 for the example)

Expressed more mathematically, for a given value x, these functions are:

- high$(x) = \lfloor x/\sqrt{u} \rfloor$
- low$(x) = x \bmod \sqrt{u}$

If we consider our clusters above, these two functions really give us a way to map an element in the universe into a bucket, and more specifically a location within a given bucket

- high(x) = which of the $\sqrt{u}$ clusters x is in
- low(x) = index within that cluster

2.6 Refined attempt: use recursion

If we can recursively apply the $\sqrt{u}$ solution, we should be able to get down to $O(\lg \lg u)$ time per operation.

View the universe $\mathcal{U}$ as a *structure* of size u. In general, suppose we have a set of elements represented by a structure S of size $|S|$. Split it into $\sqrt{|S|}$ substructures each of size $\sqrt{|S|}$. Each substructure (cluster) is named sub$[S][0]$, sub$[S][1]$, …, sub$[S][\sqrt{|S|}-1]$.

Because we need to store which substructures are empty, we can recursively use another substructure of size $\sqrt{|S|}$ called summary$[S]$. Each element in this summary structure will correspond to the emptiness of one of the $\sqrt{|S|}$ substructures.

Insertion therefore corresponds to two recursive calls, one in the appropriate substructure, and possibly one in the summary structure:

Insert(x, S):
  Insert (low(x), sub$[S]$[high(x)])
  Insert (high(x), summary$[S]$) if sub$[S]$[high(x)] was empty

Recurrence relation for Insert's running time:

$$\begin{aligned}
T(u) &= 2T(\sqrt{u}) + O(1) \\
T'(\lg u) &= 2T'(\lg \sqrt{u}) + O(1) \\
T'(\lg u) &= 2T'(\frac{1}{2}\lg u) + O(1) \\
T'(x) &= 2T'(\frac{1}{2}x) + O(1) \\
T' &\text{ is } O(x) \\
T' &\text{ is } O(\lg u)
\end{aligned}$$

Oops, we have too many recursive calls to get $O(\lg \lg u)$.

Successor is similar to the $\sqrt{u}$ solution, but with recursive calls. First we look in the appropriate substructure. If we don't find the element there, we look in the summary structure for the next next nonempty substructure. The key observation is that this operation is another successor query. Then we find the minimum element in that substructure, which can be viewed as finding the successor of $-\infty$.

Successor(x, S):
  $j \leftarrow$ Successor(low(x), sub[S][high(x)])
  if $j < \infty$
    return $j+$ high (x) $\cdot \sqrt{|S|}$
  $i \leftarrow$ Successor(high(x), summary[S])
  $j \leftarrow$ Successor($-\infty$, sub[S][i])
  return $j + i \cdot \sqrt{|S|}$

Recurrence relation for Successor:

$$\begin{aligned}
T(u) &= 3 * T(\sqrt{u}) + O(1) \\
T'(\lg u) &= 3 * T'(\lg \sqrt{u}) + O(1) \\
T'(\lg u) &= 3 * T'(\frac{1}{2}\lg u) + O(1) \\
T' &\text{ is } O(\lg u)^{\lg 3}
\end{aligned}$$

This is much worse than we want, even worse than logarithmic. The reason is that we are making too many recursive calls. We need to reduce our running time by reducing the number of recursive calls from 2 and 3 down to 1.

2.7 Store min and max of each cluster to reduce recursion

We can reduce the number of recursive calls to Insert or Successor by being mindful of when we can precalculate information without needing to do a full-blown search. One mechanism is to cache the minimum and maximum element contained by each structure S as an additional piece of information associated with the structure S. We will refer to these are min[S] and max[S].

Acccessing the minimum element in a structure S is now $O(1)$. If we can replace one or more of the recursive calls in Insert or Successor with a quick access to the minimum element of the structure, we can reduce the overall running time of the algorithms dramatically.

First, let us assess how this helps Successor:

Successor(x, S):
  if low(x) $<$ max[sub[S][high(x)]]
    $j \leftarrow$ Successor(low(x), sub[S][high(x)])
    return high(x) $\cdot \sqrt{|S|} + j$
  else
    $i \leftarrow$ Successor(high(x), summary[S])
    return min[sub[S][i]] $+ i \cdot \sqrt{|S|}$

We know what cluster x should appear in (high(x)), and if there's an element larger than x in that cluster, we just need to search in that cluster. If there was no such element, we can look at the summary structure for the next non-empty cluster, and return the smallest element in that cluster.

Since the condition of the "if" can be calculated in $O(1)$ time, and either branch makes only one recursive call to Successor on a structure of size $\sqrt{|S|}$, we've succeeded in reducing the amount of recursion to 1. The recurrence relation now looks like:

$$\begin{aligned} T(u) &= T(\sqrt{u}) + O(1) \\ T'(\lg u) &= T'(\lg \sqrt{u}) + O(1) \\ T'(\lg u) &= T'(\frac{1}{2}\lg u) + O(1) \\ T' &\text{ is } O(\lg \lg u) \end{aligned}$$

Even though Successor is now performing well, this strategy does not work well for Insert. Fundementally, Insert would still require 2 recursive calls to Insert—one to Insert into a substructure, and one to update the summary[S] structure. In order to have a running time of $O(\lg \lg u)$, we need to eliminate one of the recusive calls.

2.7.1 Final Solution: van Emde Boas structure

Our solution to this dilemma is to just not recurse unless absolutely needed. We do this by using min[S] as a sort of very-cheap ($O(1)$-time) very-small (single-element) cache for the data structure. If a structure holds only a single element, that element is non-recursively stored in the min[S] slot.

Now, to check whether an entire structure S is empty, we can just check whether min[S] is unset, instead of traversing the summary structure. More importantly, inserting into an empty structure is also a $O(1)$-time operation, because we can just set the min value and be done.

The final forms of our Insert and Successor functions are as follows. Note that both make only a

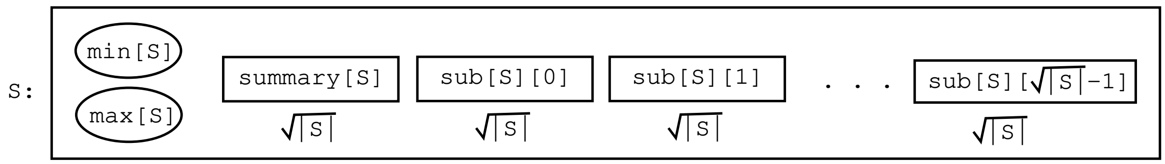


Figure 3: Representation of full van Emde Boas structure

single recursive call to themselves, so using the derivation above for calculating running time, both these functions run in $O(\lg \lg u)$ time.

Insert(x, S):
  if $x <$ min[S] then swap x & min[S]
  if sub[S][high(x)] is empty:
    Insert (high(x), summary[S])
    min[sub[S][high(x)]] $\leftarrow$ low(x)
  else
    Insert (low(x), sub[S][high(x)])
  if $x >$ max[S] then max[S] $\leftarrow x$

Successor(x, S):
  if low(x) <max[sub[S][high(x)]]:
    $j \leftarrow$ Successor(low(x), sub[S][high(x)])
    return high(x) $\cdot \sqrt{|S|} + j$
  else
    $i \leftarrow$ Successor(high(x), summary[S])
    return min[sub[S][i]] $+ i \cdot \sqrt{|S|}$

References

[AT99] Arne Andersson and Mikkel Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32nd Annual ACM symposium on Theory of computing*, pages 335–342. ACM, 1999.

[BF02] Paul Beame and Faith E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, August 2002.

[MNA88] Kurt Mehlhorn, Stefan Näher, and Helmut Alt. A lower bound on the complexity of the union-split-find problem. *SIAM Journal on Computing*, 17(6):1093–1102, December 1988.

[vEB77] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[vEKZ77] Peter van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Systems Theory*, 10:99–127, 1977.

[Wil83] Dan E. Willard. Log-logarithmic worst case range quieries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.