Advanced Data Structures Jan-Apr 2012

Lecture 9 — 16 Feb, 2012

Lecturer: Venkatesh Raman *Scribe: Sudeshna Kolay*

1 Overview

In the last few lectures, we have looked at **FKS Hashing** and **Cuckoo Hashing** and various operations that the two data structures support. However, if we want to find the predecessor or the succesor of some number x, there is no efficient way of finding it by either of the above hashing methods.

In this lecture, we look at data structures that efficiently support operators *pred()* and *succ()* along with the usual membership, insertion and deletion operators.

2 The problem

Firstly, we fix the universe U of queries to $[u]$, for some $u \in N$. Let $S \subseteq U$. S need not be predetermined. We want to use the following functions:

1. $membership(x)$: To check if $x \in S$.
2. $insert(x)$: To insert x in the set S.
3. $delete(x)$: To delete x from S.
4. $pred(x) = max\ \{y \in S | y \leq x\}$
5. $succ(x) = min\ \{y \in S | y \geq x\}$

Notice that if S was a static set then we could store the predecessor and successor of each $x \in S$ explicitly, so that look-up time for these operations would be $O(1)$. However, if $|S| = m$, this would require extra storage space of $O(m \log m)$ bits.

2.1 First attempt using binary trees

Consider the universe as a bit vector in $\{0,1\}^u$, where the positions corresponding to elements in S are set to 1 and all other positions are set to 0. Then we build a complete binary tree on this vector (Figure 1). The size of the binary tree is bounded by $O(u)$ and therefore the height of the tree is bounded by $O(\log u)$. Each internal node of the tree are set to the OR of its children.

1. $membership(x)$ requires $O(1)$ time - We can simply check in the bit vector.
2. $insert(x)$ or $delete(x)$ takes $O(\log u)$ time - First, in the bit vector we have to flip the bit at the position corresponding to x. Then we have to reset the values of the internal nodes in the x-to-root path.

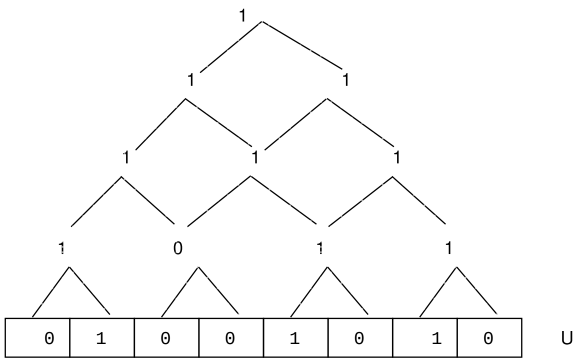


Figure 1: Binary tree built over the bit representation of subset S in universe U

3. $pred(x)$ or $succ(x)$ takes $O(\log u)$ time - For $pred(x)$ we move towards the root. If an internal node is set to 1 but the child that is not on the x-to-root path is set to 0 then we continue towards the root. If the non-path child is set to 1, then from the child we move along the rightmost path where all internal nodes are set to 1. The leaf where this path ends is the predecessor of x. We do something similar for $succ(x)$. In the worst case, we could be traversing the longest path in the binary tree, the length of which is bounded by $O(\log u)$.

If S is much smaller than U, then this data structure may not be efficient as we cannot bound the running time of any of the operations in terms of the size of S.

We could use a $\sqrt{u}$-ary tree of height 2, constructed in the same way as the binary tree. This makes the $membership()$ and $insert()$ operations constant time. We have to do a little more work for $delete()$, since resetting the values of each internal nodes on the leaf-to-root path, therefore the entire operation, may take as much as $O(\sqrt{u})$ time. For $pred()$ or $succ()$, searching for a suitable child at an internal node can also take as much as $O(\sqrt{u})$ time.

2.1.1 van Emde Boas Trees [1]

Here, we look at a data structure that supports all the desired operations in $O(\log \log u)$ time. Instead of representing S and U as a bit vector, we think of U as the set of all binary bit vectors of length $\log u$. Again, we can break up a bit vector x into the most significant $(\log u)/2$ bits (or $high(x)$) and the least significant $(\log u)/2$ bits (or $low(x)$). The universe Top of all $high(x)$, where $x \in U$, is of size $\sqrt{u}$. For each vector v in Top, we can concatenate $\sqrt{u}$ low vectors to get a vector in U for each concatenation.

Using this idea we build a recursive structure (Figure 2). At each node, we store a set of high vectors of a universe. At each high vector, we recursively store all low vectors associated with the high vectors in a subset of the universe. At the node we store the minimum and maximum vectors of the subset. Also, at each high node, we store the minimum and maximum of the subset of vectors formed by concatenating the high vector with each of the low vectors of the universe stored in the recursive structure below it. In case no low vectors are associated with a high node, then it points to NULL, and the minimum and maximum are set to NIL. The low of the minimum vector

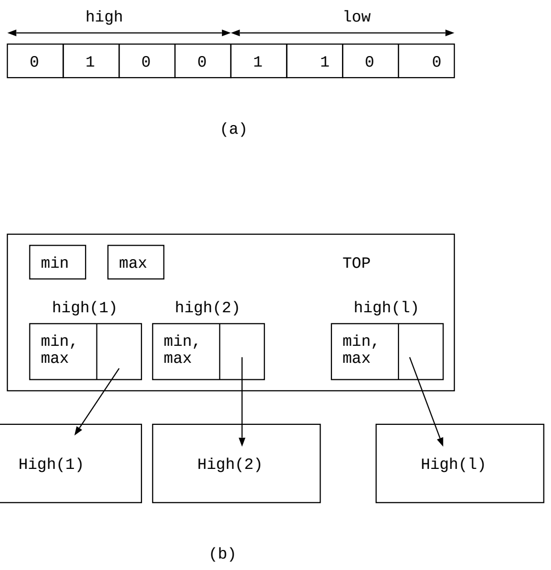


Figure 2: (a)high and low of a vector; (b)van Emde Boas trees

is not stored again in the recursively structure below. This ensures that we can access a minimum element in constant time.

For example, the root of this tree is the set of all high vectors of $S \in U$. For each high vector, we recursively store the set of low vectors such that the concatenation of the high vector with each low vector gives a vector in S. At the root, we store the minimum and maximum vector in S. For a high vector h, let $S_h = \{v \in S | high(v) = h\}$. For each high vector h, we store the minimum and maximum vectors in S_h. Given a structure T, we describe how to do the desired operations. We denote by $High(x)$ the recursive structure pointed at by $high(x)$.

- *membership*(T, x) :

```
if $x < T$.minimum
  then return NO;
if $high.minimum = x$
  then return YES;
if $high.minimum = NIL$
  then return NO;
  else return $membership(High(x), low(x))$
```

- *insert*(T,x) :

```
if $x < T.minimum$
  then $swap(x, minimum)$;
if $high(x).minimum =$ NIL
  then $high(x) = x$; return
if $x < high(x).minimum$
  then $swap(x, high.minimum)$
$insert(High(x),low(x))$
```

- *pred*(T,x) :

```
if $T.minimum = x$ return NIL;
if $High(x).minimum > x$
  then return $pred(T,high(x)).maximum$
```

```
    else return $pred(High(x), low(x))$;
```

- *succ*(x) : Same as *pred*(x) after replacing *pred* by *succ*, minimum by maximum and "<" by ">".
- *delete*(T,x):

```
if $T.minimum > x$ then return
if $high(x).minimum = x$ and $High(x)$ = NULL
then $high(x).minimum$ = NIL; return
if $high(x).minimum = x$
  then    { $y = succ(T,x)$;
            $delete(High(x), low(y))$
            $minimum = y$;}
  else $delete(High(x), low(x))$;
```

Each operation makes one recursive call. So the time complexity of each operation can be calculated as follows :

$$T(u) \leq T(\sqrt{u}) + O(1)$$

.

Replacing u by 2^t and writing the recursion in terms of t, this recursion solves to $T(t) \leq \log t$. Therefore,

$$T(u) \leq \log \log u$$

.

The space complexity, however, has the following recursion :

$T(u) \leq (\sqrt{u} + 1)T(\sqrt{u}) + O(\sqrt{u})$

This solves to $O(u)$ cells.

2.2 y-Fast Tries [2]

y-Fast tries bring down the space to $O(|S|)$ for a $S \in U$, while still performing all the required operations in $O(\log \log u)$ time. Let $|S| = n$.

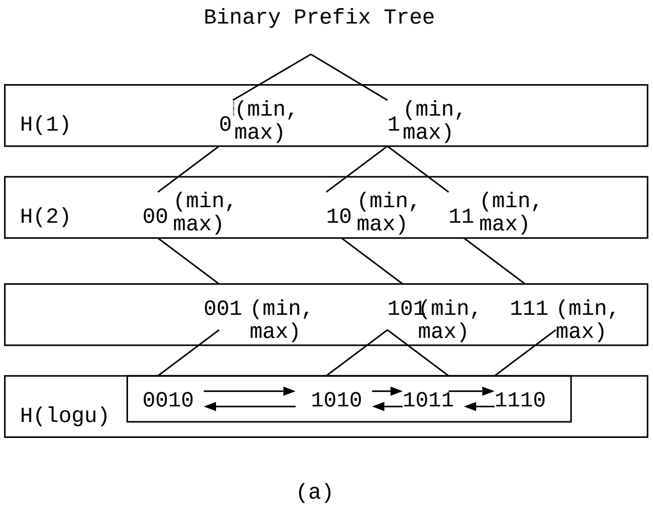
To begin with, we describe a simpler structure that takes more space, but maintains the time complexity of each operation. Again, we consider the elements as $\log u$ length binary vectors. For each vector in S and each $i \in [\log u]$, we define $S_i = \{v : |v| = i, \exists x \in S, x(i) = v\}$, where $x(i)$ represents the vector restricted to the most significant i positions .

For each S_i we build a hash table H_i, using cuckoo hashing. We maintain a prefix tree for S. At each node of the prefix tree we stor the minimum and maximum vectors in the subtree rooted at the node. We also maintain a doubly linked list for the leaves of the prefix tree. Refer to Figure 3 for the representation of the structure.

Total number of prefixes in S is $O(n \log u)$. So, total space for the structure is also $O(n \log u)$.

- *membership*(x):

```
if $H_{\log u}(x)$ is empty
  then return NO
  else return YES
```



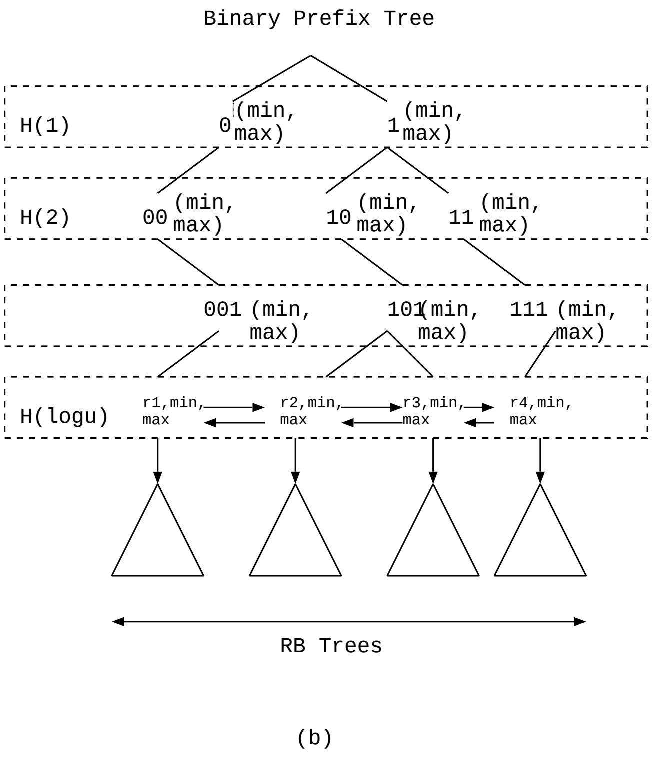


Figure 3: (a) A binary prefix tree; A hash table is maintained for the set of nodes at every level; The leaves (elements of S) are doubly linked. (b) A *y-Fast* trie where the leaves are representatives; groups are attached as RB trees to representatives.

As $|H_{\log u}| < \log u$, we can check for membership in $\log\log u$ time by binary search in the table.

- *insert*(x) or *delete*(x): Same as inserting or deleting in each of the hash tables. For each hash table, deletion is $O(1)$ and insertion is $O(1)$ expected amortised. So total expected amortised time is $O(\log u)$.

- *succ*(x) or *pred*(x): We do a nested binary search to find the longest common subsequence with x. In the inner binary search we search for $x(i)$ in hash table H_i, while in the outer binary search we search for the longest common subsequence with x. On finding $x(i)$ in H_i we go to the corresponding node in the prefix tree and look at the minimum and maximum of that node. If $minimum(maximum) = x$, then the predecessor(successor) has a shorter common sequence. So we search in the shorter prefix hash tables. Otherwise, the predecessor(successor) has a longer common subsequence with x, and we search in the hash tables for the longer subsequences. Once we get the longest common subsequence at node v, then x,if present, should lie in the right(left) subtree of v then $v \rightarrow left.maximum$ $(v \rightarrow right.minimum)$ is the predecessor(successor) of x. Once we find predecessor or successor, we can find the other by using the doubly linked list. The size of each hash table is $O(\log\log u)$ and so is the height of the prefix tree. So these operations takes $O(\log\log u)$ time.

In order to improve the space complexity we break up the set S into $n/\log u$ groups according to the ordering of U, each group being of size $\log u$. Then we pick a representative from each of these groups and form the hash tables and prefix tree on these representatives. Lastly, we build RB trees out of the rest of the elements for every group and attach these trees to the leaves of the prefix tree by pointers. We store the minimum and maximum vectors of each group at the respective leaves. Total space for the whole data structure becomes $O(n)$.

- *membership*(x): We find the representative of x by comparing x with the minimum and maximum stored at the leaves. Then we search for x in the RB tree pointed at by the representative. Finding the representative takes $O(\log u.\log n)$ time while searching in the RB tree takes $O(\log\log u)$ time. So the total time is bounded by $O(\log\log u)$.

- *pred*(x) or *succ*(x): Again we find the representative for the group that should contain x. In the degenerate cases where x lies between the minimum(maximum) of one representative and maximum(minimum) of the previous(next) representative then the predecessor(successor) of x is the maximum(minimum) of the previous(next) representative. Otherwise, on finding the representative, we search in the corresponding RB tree for the *pred*(x) or *succ*(x). Again finding the representative and then searching in the RB tree takes $O(\log\log u)$ in total.

- *delete*(x): We find x and if it is there then delete it. If x is the minimum, maximum or the representative of a group we find the *succ*(x), *pred*(x) and any leaf vector of the RB tree and update respective pointers for minimum, maximum or representative. If, on deletion of x, the group becomes empty then we spend $O(\log u)$ deleting nodes on the x-to-root path that are not used by any other representative. Note that this operation is done only after all $\log u$ elements of a group is deleted. Therefore the amortised time for a deletion is $O(\log\log u)$.

- *insert*(x): We find the group where x should belong and till the group becomes of size more than $(2\log u)$ we simply insert x in the RB tree for that group. If the size of the RB tree exceeds $(2\log u)$ then we split the group into 2 halves according to the ordering in U. Each part has at most $\log u + 1$ nodes. we keep the old representative for one of the parts and fix a new representative for the other part. Now we need to insert the new representative in the $\log u$ hash tables. Insertion into each hash table is $O(1)$ expected amortised. So the total time for introducing the new representative and the reshuffling is $O(\log u)$. But again this only happens in every $\log u$ insertions to a group and so the amortised expected insertion time is $O(\log \log u)$.

Hence, *y-Fast* tries take linear space and log-logarithmic (amortised expected in some cases) time per operation.

References

[1] Peter van Emde Boas: Preserving order in a forest in less than logarithmic time, *Proceedings of the 16th Annual Symposium on Foundations of Computer Science 10*: 75-84, 1975.

[2] Dan E. Willard: Log-logarithmic worst-case range queries are possible in space (N), *Information Processing Letters* (Elsevier) 17 (2): 81-84, 1983.