COMS E6998-8: Advanced Data Structures (Spring'19) February 14, 2019

Lecture 4: Predecessor Search on the RAM

Instructor: *Omri Weinstein* Scribes: *Alan Du, Alexandre Lamy*

1 Introduction

In this lecture we are still interested in the predecessor search problem.

Recall (Predecessor Search): In the predecessor search problem, one has keys $S = \{x_i\}_{i=1}^n$ where $x_i \in [u]$ ($[u] = \{1, 2, \ldots, u\}$) and $u = \text{poly}(n)$. Queries are of the form $\text{pred}(y) = \max\{x \in S \mid x \leq y\}$. The goal is to prepare and maintain a data structure which is able to answer such queries quickly, and be able to update quickly when a new key is added to the set. We denote the data structure's worse case (potentially amortized) runtime on queries by t_q and on updates as t_u.

Recall (Predecessor search bounds seen so far): In lecture 1, we quickly saw that using self-balanced BSTs, we can achieve a worst-case running time of $t_q = t_u = O(\log n)$. In lectures 2 and 3 we showed that on some access sequences we are able to do better. At the end of lecture 3, we showed that the worst case $O(\log n)$ bound was tight for BSTs. I.e. there exists an access sequence (the bit reversal sequence) on which any BST needs at least $\Omega(\log n)$ time per access/update.

The goal of today's lecture is to achieve running times of $o(\log n)$ by using a stronger model of computation than BSTs: the word-RAM (transdichotomous) model.

Recall (The word-RAM or transdichotomous model): This model assumes that memory is a c-style, finite array which is divided into w-bit words. It further assumes that most bit-wise operations (+, -, *, /, %, &, |, <<, >>, etc.) can be performed on w-bit words in a single operation (constant time). Time is then the number of such operations performed and space is the number of w-bit words used (denoted by s). Finally, the transdichotomous assumption is that $w = \Theta(\log n)$. In practice, this is very reasonable. Indeed, consider today's computers which have 64 bit words. For the assumption to be violated we would need $n \geq 2^{64} > 10^{19}$. Note that the key benefits of the word-RAM model over BSTs is the random access and the ability to encode data arbitrarily.

Goal of lecture: The main goal of this lecture is to show how to do predecessor search on the word RAM in $t_u = t_q = O(\min\{\log w, \log_w n\})$ and space $s = O(n)$. The $O(\log w)$ running time will be achieved using Van Emde Boas Trees and the $O(\log_w n)$ running time will be achieved using Fusion Trees. In lecture 5 we will show that there are more or less matching lower bounds, even in the cell probe model.

2 Y-fast Tries and Van Emde Boas Trees (vEB trees) [van Emde Boas, 1977] and [Willard, 1983]

Van Emde Boas trees and Y-fast tries are used to store w-bit keys (or integers in the universe with $U = 2^w$) that can execute a predecessor search in $O(\lg w) = O(\lg \lg U)$ time.

This is particularly useful when we are storing many keys in our possible key space (e.g. $U = n^{O(1)}$, because this translates to $O(\lg \lg n)$ lookup time, an exponential speed-up over binary search trees).

Bitwise Tries

We will start by trying to get the $O(\lg w)$ search time and worry about space later. The key data structure we will use is a bitwise trie. A trie is a tree that stores a set of strings, where each node in the trie is associated with a string prefix.

A bitwise trie is just a trie that operates over bitstrings:

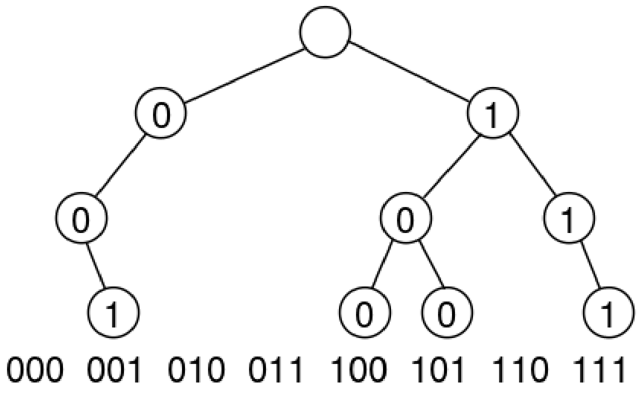

Figure 1: A bitwise trie that stores $\{1, 4, 5, 7\} = \{001, 100, 101, 111\}$

We add two additional types of pointers to a bitwise trie: first, we will add leaf pointers between all the leaf nodes so that they form a linked list. This will allow us to go from a leaf node to its predecessor and successor in $O(1)$ time. We will also add *descendant pointers*, for any non-leaf node missing a child. If any non-leaf node doesn't have a left child, we make its left child point to the smallest leaf node in its subtree ($k_{\min}$). Likewise, if it doesn't have a right cihld, then we add a pointer to its largest leaf ($k_{\max}$).

Now, suppose we want to find pred(q). Then, we can walk down the trie until we find where q "falls off" the trie (this takes $O(w)$ time, because the trie has depth w). But once we're there, we can follow the descendant pointers and the child pointers to find the pred(q) in $O(1)$ time.

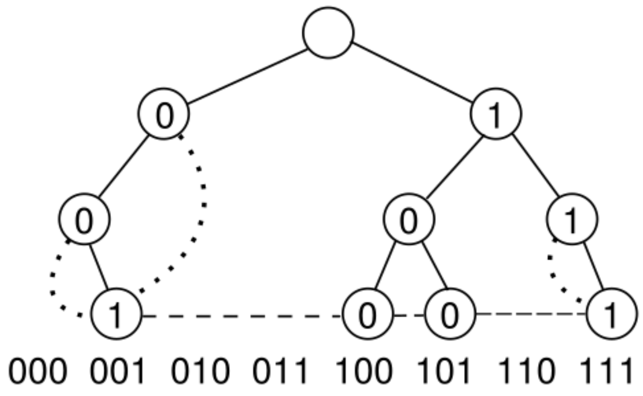

Figure 2: The trie with descendant pointers (finely dashed lines) and child pointers (dashed lines)

Hashing: X-fast Trie

The key observation to make this faster is that instead of doing an $O(w)$ linear search down the trie, we can actually do a binary search on the prefix length (trie level). If, for each level, we can check whether a prefix is in the trie or not in $O(1)$ time, then this gives us $O(\lg w)$ time, which is an exponential speed-up.

How to get constant lookup time? The answer is to construct a *level-search structure* using a hash table (we can either use one hash table per level, or large hash table) to get us $O(1)$ lookup time.

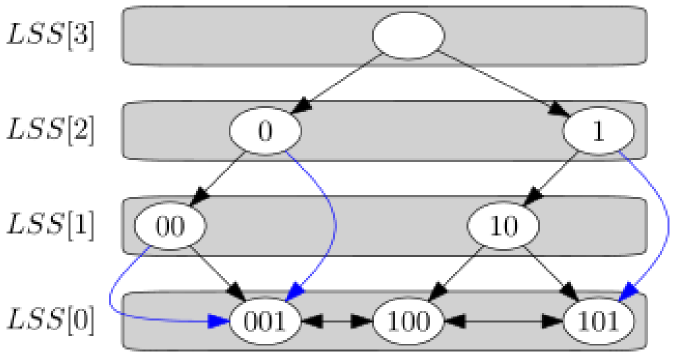

Figure 3: An X-fast trie. Image from [Commons, 2009]

This data structure is called an X-fast trie, and allows us to have $t_u = O(\lg w) = O(\lg \lg U)$ predecessor search time. Each key that we store adds at most w nodes to our trie, so our space complexity $S = O(nw) = O(n \lg U)$.

The downside, however, is that both updates and queries take $O(w)$ time – for each insertion/deletion, we have to the trie from root to leaf to update descendant pointers and add nodes to our hash table. Can we do better?

Indirection: Y-fast Trie

The answer is yes, using a trick called indirection from [Willard, 1983]. Indirection will allow us to get $t_u = O(\lg w)$ and $t_q = O(\lg w)$ amortized while also reducing our space complexity to $O(n)$ instead of $O(nw)$.

The basic idea is to group our keys into buckets of $Theta(w)$ consecutive elements. In total, we should have thus have $\frac{n}{w}$ buckets. From each bucket B_k, we choose a *representative* r_k, which we can choose arbitrarily.

We then construct an X-fast trie based on the $\frac{n}{w}$ representatives, and within each bucket store the keys as any balanced binary search tree (e.g. AVL tree or red-black tree):

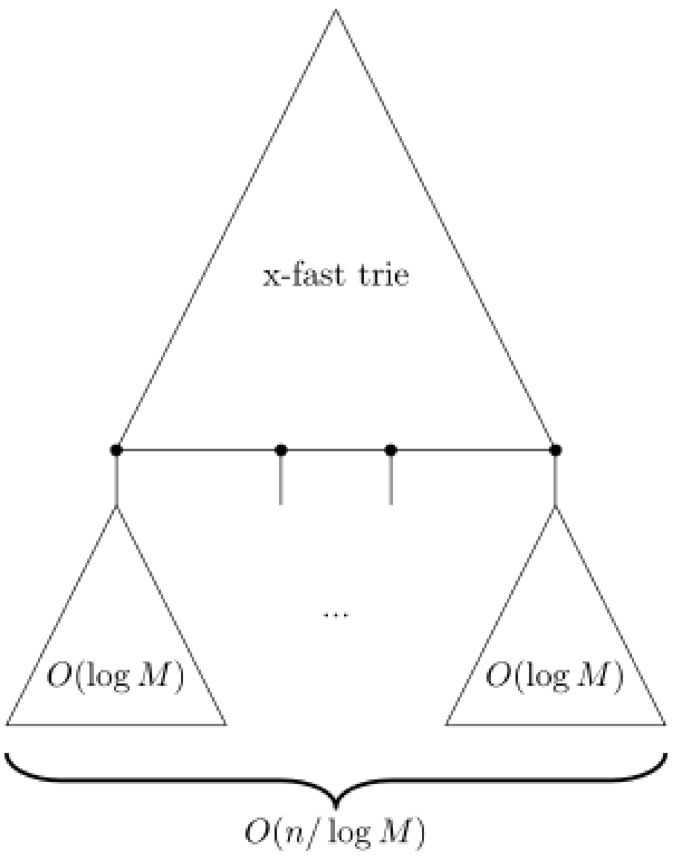

Figure 4: An Y-fast trie, with an x-fast trie at the top and groups of balanced binary search trees at the bottom. Here, $w = \lg M = \lg U$. Image from [Commons, 2011]

Now, to find pred(x), we first find the *predecessor representative* of x in the x-fast trie, which takes $O(\lg w)$ time. Then, we follow the linked list of leaf nodes to find the successor representative of x in $O(1)$ time. By construction of our buckets, the predecessor of x must reside in one of the two representatives' corresponding BSTs, so we execute a predecessor search on both BSTs. Because each bucket has w elements in it, searching each bucket takes $O(\lg w)$ time too, so the total query time is still $t_q = O(\lg w) = O(\lg \lg U)$.

Now, to insert a key x into a y-fast trie, we first find the bucket that contains x (which takes $O(\lg w)$ time) and then insert x into the corresponding BST (which also takes $O(\lg w)$ time). Now, to ensure our BSTs are still sized $\Theta(w)$, whenever our BST gets to size $2w$, we split the BST into two BSTs of size w. Splitting the BSTs takes $O(w)$ time (because we must insert a new representative into the x-fast trie), but we only have to do this every $O(w)$ insertions into the same bucket. The cost of updating the x-fast trie is $O(1)$ amortized, and insertions are $O(\lg w)$ amortized.

Deletions follow similarly, which gets us $t_u = O(\lg w) = O(\lg \lg U)$ amortized.

The last neat thing about indirection is that this also reduces our space complexity! The x-fast trie stores $O(\frac{n}{w})$ keys, which takes $O(n)$ space. But the BSTs also only store n nodes and thus also have $O(n)$ space, so the total space is $O(n)$.

In summary, we have:

	x-fast trie	y-fast-trie
t_q	$O(\lg \lg U)$	$O(\lg \lg U)$
t_u	$O(\lg U)$	$O(\lg \lg U)$ amortized
S	$O(n \lg U)$	$O(n)$

Figure 5: Summary of complexity bounds

Van Emde Boas Trees

Van Emde Boas Trees do this binary search on the string length in a different way. The key idea behind the Van Emde Boas tree is to divide our universe of w-length bitstrings into chunks of size 2^l. Then the first l bits of a key determine which chunk the key belongs to, and we only care about the last $w - l$ bits within each chunk.

Now, suppose we want to find pred(x). Then, if the first l bits of x match any of the chunks, we can find the pred(x) by running a predecessor search on the last $w - l$ bits of x within that cluster. If x is not in the chunk, then we need to find the predecessor chunk of x (which is a predecessor search on the first l bits) and get the maximum value of that chunk.

Let $f(w)$ be the cost of a predecessor search on w bits. Then, if we use a hash table to determine whether the first l bits of x are in any chunk and have each chunk contain its maximum element, we can establish a recurrence relationship:

$$f(w) = \Theta(1) + \max\{f(l), f(w - l)\} = \Theta(1) + f(\frac{w}{2})$$

where we get the second equality by choosing $l = \frac{w}{2}$. If we can establish this recurrence relation, then we know that $f(w) = O(\lg w)$.

Formally, a Van Emde Boas Tree consists of:

1. The minimum and maximum value in the tree
2. The universe size $U = 2^w$.
3. A "summary" or auxiliary Van Emde Boas Tree, for doing a predecessor search on the $\frac{w}{2}$ length prefixes defining the buckets / children.
4. A hash table, mapping $\frac{w}{2}$ length bitstrings to the children Van Emde Boas Trees

Predecessor searches are done by first looking up in the hash table which child Van Emde Boas Tree the query belongs to. If it's found, then you recurse into the child tree. Otherwise, you do a predecessor search on the summary Van Emde Boas Tree to find the predecessor prefix, which you then lookup in the hash table and get the maximum value of.

3 Fusion Trees [Fredman and Willard, 1993]

Goal: We want to show that using this data structure we can achieve $t_q = t_u = O(\log_w n)$ using $O(n)$ space.

Note that $\log_w n = \frac{\log n}{\log w}$ may not seem like much of an improvement over $O(\log n)$ (and isn't if $w = O(1)$), but under the transdichotomous assumption we have $w = O(\log n)$. Thus $\log_w n = \frac{\log n}{\log \log n}$ which is a slight improvement. More interestingly, in practice as mentioned previously w is significantly larger than $\log n$. Indeed in current 64 bit machines, $w = \log n \implies n > 10^{19}$ which is more than unlikely. Hence, in current machines it is often reasonable to assume $w = O(n^\epsilon)$ in which case $\log_w n = \frac{\log n}{\log(n^\epsilon)} = O(1)$!

3.1 B-trees

The main idea behind these Fusion Trees is to use a B-tree instead of a standard binary search tree. B-trees are simply search trees (the children are ordered) with branching factor B (i.e. each node has up to B children). See Figure 6 for an example. Thus note that binary search trees are simply B-trees with $B = 2$. The reason one might be interested in using B-trees is that the depth of the tree is then $\log_B(n)$. Naively then, one might think that we can trivially achieve the $O(\log_w(n))$ bounds by simply taking $B = w$. This is not so since, at each node, we basically have to do a predecessor search in a set of size $O(B)$ which takes time $\log(B)$. In this case the search time in a B-tree would be (time per node) $\times$ depth $= \log(B) \log_B(n) = \log(n)$ (so we get no improvement compared to a regular BST).

A Fusion Tree is a B-tree "done right". More specifically, a Fusion tree **is** a B-tree but where, by exploiting the power of the word-RAM, the tree is able to spend constant $O(1)$ time at each node rather than $\log(B)$ time. This immediately implies that the total search time is $\log_B(n)$ rather than $\log(n)$ as desired. This is only possible for relatively small B values: we need to have $B \leq w^{1/5}$. Note that this

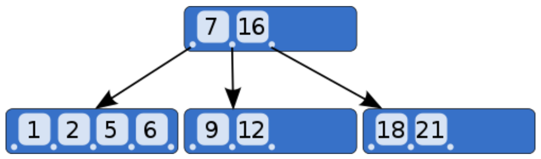

Figure 6: Example B-tree for $B = 5$. Image from [Commons, 2016].

isn't an issue since

$$\log_{w^{1/5}}(n) = \frac{\log n}{\log(w^{1/5})} = 5\frac{\log n}{\log w} = 5\log_w(n) = O(\log_w(n))$$

So all we need to do is look at a single Fusion Tree node and make sure that we are able to spend only constant time at the node. The rest of the data structure then follows.

3.2 Fusion Nodes

In a given fusion node we have B children (keys) and we need to find the predecessor of a new value in $O(1)$ time. The core idea is to use a **trie** and **sketching**.

3.2.1 Sketching a trie

As seen in section 2 using hashing and dynamization via indirection a trie on B nodes takes B space (i.e. Bw bits). But now consider the **branching levels** of the trie (these are defined as the levels at which at least one node has that both its children are nonempty). In a very real sense, only the bits corresponding to those levels (remember that each level corresponds to one bit of the keys) matter. Hence we define the following.

Definition 1. Letting $b_1, b_2, \ldots, b_k$ be the bits corresponding to the branching levels of a given trie, we define the **sketch of x** as $s(x) = x_{b_1}x_{b_2}\ldots x_{b_k}$.

But now note that clearly each additional key can add at most one branching level to the tree. So if there are B keys, there will be at most B branching levels. This means that the sketches of the keys will each take at most B bits. Thus, if we make a trie using the sketches of the keys rather than the actual keys, the whole trie can be made to take space $B \cdot B$ only. Hence if $B = \sqrt{w}$ we could fit the entire trie in a single word. The word-RAM model would then allow us to load up and do certain operations on the whole trie in constant time.

There are a few issues. The main issue is making sure that it is still possible to find the true predecessor of a new query using only the sketched trie. The second is making sure that it is possible to do this using only a constant number of word-RAM operations. We deal with the former issue in the next subsection. The latter issue will be dealt with in lecture 5.

3.2.2 Desketchifying

Clearly sketching is order preserving for the set of keys in the trie. So if a predecessor query seeks exactly one of the keys of the trie, then one can simply run predecessor search on the sketched trie. However, sketching is not order preserving for other values. Indeed if a new value "falls" off the paths leading to keys of the trie at a non-branching level, then that information will be lost in the sketched trie.

Suppose that we were able to find the node at which a given query fell off the key paths. Then if it fell to the right one would just find the predecessor of the largest value in the left subtree (in the sketched trie). If it fell to the left one would simply find the successor of the smallest value in the right subtree (that would give the successor of the query rather than the predecessor, but it is easy to have an ordered doubly linked list of the keys in order to move from successor to predecessor). Specifically, we use the following procedure to find the predecessor of a query.

(1) Find $i \in [B]$ s.t. $s(x_i) \leq s(y) \leq s(x_{i+1})$. We will show in lecture 5 that this can be done in $O(1)$ time.

(2) Let v_y be the prefix corresponding to the node at which the path to y diverges from the paths to any of the keys (i.e. the node at which y fell off). Then note that $v_y = \text{lcp}(\text{lca}(y, x_i), \text{lca}(y, x_{i+1}))$ (where lcp is the longest common prefix and lca is the lowest common ancestor). Note that from the previous step we know $s(x_i)$ and $s(x_{i+1})$, we can then get x_i and x_{i+1} by simply storing a look-up table on the side. We claim that finding the lca and lcp can be done in $O(1)$ time (we will show this in lecture 5).

(3) If y fell to the right let $y' = v_y 0111...1$, if y fell to the left let $y' = v_y 1000...0$.

(4) If y fell to the right find $\text{pred}(s(y'))$, if it fell to the left find $\text{succ}(s(y'))$ and then find the predecessor using the ordered doubly linked list between keys. In either case the obtained value is $\text{pred}(y)$.

References

[Commons, 2009] Commons, W. (2009). File:xfast$_t rie_e xample.svg - - - wikimediacommons, thefreemediarepository. [Online; accessed 21 - February - 2019]$.

[Commons, 2011] Commons, W. (2011). File:y-fast$_t rie.svg - - - wikimediacommons, thefreemediarepository. [Online; accessed 21 - February - 2019]$.

[Commons, 2016] Commons, W. (2016). File:b-tree.svg — wikimedia commons, the free media repository. [Online; accessed 20-February-2019].

[Fredman and Willard, 1993] Fredman, M. L. and Willard, D. E. (1993). Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences*, 47(3):424–436.

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

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