CS 224: Advanced Algorithms Fall 2013

Lecture 1 — September 2nd, 2013

Prof. Jelani Nelson *Scribe: Marcus Schorow*

1 Overview

In this lecture, we went over class logistics. We also discussed the word RAM model and Van Ende Boas trees as a data structure solution to the static predecessor problem. Towards the end of class we talked about y-fast tries as a similar approach.

2 Course logistics

The course staff can be emailed at css224-f14-staff@seas.harvard.edu.

2.1 Grading

- *Scribing **10%*** – Every student will be responsible for taking scribe notes for 1 (possibly 2) lectures. Scribe notes are due by **9 PM** the following day.
- *Psets **60%*** – In addition to turning in psets, students may be responsible for grading them in groups. At most, each student will be responsible for one pset.
- *Project **30%*** – There will be a final project at the end of the semester. The project proposal will be due **October 30th** and the project report will be due on the last day of reading period.

2.2 Goals

Main takeaways from this class should include:

- Increased ability to analyze and create algorithms
- Understanding of models and the inspiration behind models for which to analyze algorithms

3 Static and Dynamic Predecessor Introduction

In this problem, we desire a data structure that represents a set of items $S = \{x_1, \ldots, x_n\}$ that can satisfy predecessor queries that are of the form $pred(z) = \max\{x \in S : x < z\}$. We wish for a data structure that uses low space and has a fast query. For the static problem, we do not support insertions and deletions, whereas we allow these for the dynamic version of the problem.

One solution for the static version is to store elements in sorted order and do a binary search for each query. For the dynamic version, we can get $O(\log n)$ query by using a balanced BST like a red-black tree. This will take $O(n)$ space. Note that this provides for a sorting algorithm using the dynamic solution: find a max by going through the entire data and then continuously find the predecessor. This will yield an $O(nlog(n))$ sort. With more complicated data structures, we will see that we can do better than this.

4 Word RAM Model

In the word RAM model, items are integers in $\{0, 1, \ldots, 2^w - 1\}$, where the word size is w. The universe u will then have size $u = 2^w - 1$. We assume that all pointers can fit in a word, and that we have enough space to fit everything, so $u \geq n$ and $w \geq logn$. All normal operations such as $+ - / * \ ^| \& >><<$ are supported in constant time on integers that fit in a word.

5 Proposed Solutions to the Predecessor Problem

We will cover two data structures as solutions:

1. *Van Emde Boas tree [1]* – Update and query are $\Theta(\log w)$ time. However, the naive version uses $\Theta(u)$ space. This can be made $\Theta(n)$ with randomization. A very similar structure is the y-fast trie which has the same bounds, and is due to Willard [2].
2. *Fusion trees, from a paper by Fredman and Willard [3]* – Query in $\Theta(\log_w n)$ and linear space. For simplicity, this class will only consider these for the static version of the predecessor problem.

By choosing the better of these two, we can achieve $\min(\log w, \log_w n)$ which is at least as good as $\sqrt{\log n}$. This implies that with dynamic trees, we can have $O(n\sqrt{\log n})$ sorting. This won't be fully covered, as we will not go into dynamic fusion trees.

5.1 Aside on Sorting

We can actually get faster sorting. In particular, $O(n \log \log n)$ deterministically was demonstrated by Han [4], and $O(n\sqrt{\log \log n})$ randomized was provided by Han and Thorup soon afterwards [5]. It is an open problem if you can get linear within this model.

6 van Emde Boas Trees (vEB trees)

The basic idea is divide-and-conquer and requires bit manipulation. Let us define vEB trees recursively. Let vEB_u denote the vEB tree of size u. It will store the minimum ($V.min$) and maximum ($V.max$) elements, a top $vEB_{\sqrt{u}}$ sub-tree ($V.summary$) and $\sqrt{u}$ bottom $vEB_{\sqrt{u}}$ sub-trees ($V.cluster$).

Let us express the element x in binary and divide it into a lower half and an upper half of bits $\langle c, i \rangle$. This is basically writing x in base $\sqrt{u}$. We will store i in the c-th cluster. The summary will track which clusters are non-empty, so we will insert c into the summary if the c-th cluster was empty before.

Queries: If what we are looking for is below the minimum of the corresponding cluster or the cluster is empty, we find the predecessor to that cluster in the summary and get the maximum element from that cluster. Otherwise, we know the predecessor is in the cluster, and we make a recursive call. Pseudocode:

```
pred(V, x = <c, i>)
    if x > V.max: return V.max
    else if V.cluster[c].min < x:
        return pred(V.cluster[c], i)
    else:
        c' = pred(V.summary, c)
        return V.cluster[c'].max
```

To do insert, we use the trick that the minimum of a tree will not be stored in its clusters and only in $V.min$. You can think of the min as a buffer. To check if a tree is empty, we can just check if its min is set.

```
insert(V, x = <c, i>):
    if x > V.max:
        V.max = x
    if V is empty:
        V.min = x;
        return;
    if x < V.min: swap(x, V.min)
    if V.cluster[c].min == null:
        insert(V.summary, c)
    insert(V.cluster[c], i)
```

When we call the query, we make one recursive call which sets up the recurrence $T(u) = T(\sqrt{u}) + O(1)$, which then implies that $T(u) = O(\log \log u) = O(\log w)$.

For insertion, it may look like we can make 2 recursive calls in the worst case (when the cluster is empty), but in this case, inserting i will be constant time since the cluster was empty. Thus, we have that $T(u) = T(\sqrt{u}) + O(1)$. Then, $T(u) = O(\log \log u)$.

In terms of space of vEB trees, we have $S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + O(1)$, which implies that $S(u) = \Theta(u)$. This is poor.

To improve space, in a vEB data structure, we take advantage of the fact that almost all of the clusters are empty. Let us have a hash table instead. A key is a cluster ID c and its value is a pointer to the corresponding non-empty cluster. We claim that this vEB with the hash table uses $\Theta(n)$ space.

Proof: If we charge the cost of storing (c, ptr to cluster c) to the minimum element of cluster c, each

minimum is charged exactly once, so the amount of space will be linear in the number of elements we store. ***

6.1 Aside on the Dictionary Problem

In the dictionary problem we want to store (key, value) pairs. We want to support:

- query(k): returns the value associated with key k (or null if k is not associated with any values)
- insert(k, v): associates value v with key k

It turns out that a dynamic dictionary is possible with $\Theta(n)$ space, $\Theta(1)$ worst case query, and $\Theta(1)$ expected insertion and deletion. This was by Dietzfelbinger et al in 1994 [6]

7 Y-Fast Tries

We will now cover a slightly different approach that yields the same bounds.

Another solution is using a bit array of length u. Bits will be set to 1 if the corresponding element is in the set and 0 otherwise. This naively uses $O(u)$ time. To speed this up, we also build a perfect binary tree, where each internal node stores the logical or of its two children. Store all the leaf 1's in a doubly linked list. To get a predecessor of a 1, just go backwards in the linked list. To get a predecessor of a 0, go up the tree and find the nearest 1. Then, we follow the 1's to find the 1 in the other branch. If we hit the ancestor 1 from the right branch, we find the maximum and this yields the predecessor. Otherwise, we have find the minimum which is the successor 1 and we can step backwards in our doubly linked list to find the predecessor. Naively, this takes $O(logu)$ time to find the first ancestor which is a 1. However, we can use the fact that all paths to the top are monotone and can do a binary search to find where it changes from 0 to 1. If we store the tree as an array, we can find the k-th ancestor simply by right shifting the index by k bits. This means we can find the ancestor is $O(loglog(u))$

To improve the space usage of the tree in a similar way as before,we can only store the 1's in a hash table, where the key would be the index of the node. Now when we search, we only check whether the key exists. This will require $\Theta(nw)$ space since each level has at most n 1's, and is called an x-fast trie.

To improve to a y-fast trie which will use $\Theta(n)$ space, we use a technique called indirection. We have an x-fast trie on n/w items, where each item is a balanced BST that has roughly w sub-items whose smallest element is the representative element for the top level x-fast trie. This will now take $O(n)$ space. Each BST will contain an interval of elements in order. We use the top level trie to find which BST to use. Searching in this BST will take $O(log(w))$ time. We allow each BST to have between $w/2$ and $2w$ elements, and so only need to modify the trie at most every $\Theta(w)$ steps, which will cost $logw$ and thus will not affect our amortized performance. Therefore, insertion (and deletion) can still be done in $O(log(w))$ time.

References

[1] Peter van Emde Boas. Preserving Order in a Forest in Less than Logarithmic Time. FOCS 1975: 75-84

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

[4] Yijie Han: Deterministic sorting in $O(n \log\log n)$ time and linear space. STOC 2002: 602-608

[5] Yijie Han, Mikkel Thorup. Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space. FOCS 2002: 135-144

[6] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, Robert Endre Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. SIAM J. Comput. 23(4): 738-761 (1994)