CS 224: Advanced Algorithms Spring 2017

Lecture 1 — 24 January 2017

Prof. Jelani Nelson *Scribe: Jerry Ma*

1 Overview

We introduce the word RAM model of computation, which features fixed-size storage blocks ("words") similar to modern integer data types. We also introduce the predecessor problem and demonstrate solutions to the problem that are faster than optimal comparison-based methods.

2 Administrivia

- Personnel: Jelani Nelson (instructor) and Tom Morgan (TF)
- Course site: `people.seas.harvard.edu/~cs224`
- Course staff email: `cs224-s17-staff@seas.harvard.edu`
- Prerequisites: CS 124/125 and probability.
- Coursework: lecture scribing, problem sets, and a final project. Students will also assist in grading problem sets. **No coding**.
- Topics will include those from CS 124/125 at a deeper level of exposition, as well as entirely new topics.

3 The Predecessor Problem

In the predecessor, we wish to maintain a ordered set $S = \{X_1, X_2, ..., X_n\}$ subject to the predecessor query:

$$pred(z) = \max\{x \in S | x < z\}$$

Usually, we do this by representing S in a data structure. In the **static predecessor problem**, S does not change between *pred* queries. In the **dynamic predecessor problem**, S may change through the operations:

$$insert(z) : S \leftarrow S \cup \{z\}$$

$$del(z) : S \leftarrow S \setminus \{z\}$$

For the static predecessor problem, a good solution is to store S as a **sorted array**. *pred* can then be implemented via binary search. This requires $\Theta(n)$ space, $\Theta(\log n)$ runtime for *pred*, and $\Theta(n \log n)$ preprocessing time.

For the dynamic predecessor problem, we can maintain S in a **balanced binary search tree**, or BBST (e.g. AVL tree, red-black tree). This also requires $\Theta(n)$ space, and $\Theta(\log n)$ runtime for *pred*, *insert*, and *del*.

Note that query and insert runtime is asymptotically optimal in a comparison-based model (where the only available operations on element values are comparisons such as $\leq, =, \geq$. This is because a set may be sorted by inserting all n elements and calling predecessor n times, passing in each time the result of the previous call. If query and insert were sub-logarithmic, the $\Omega(n \log n)$ lower bound on comparison-based sorting would be broken. Thus, to optimize beyond BBSTs, we will require a new model of computation: the word RAM model.

4 Word RAM Model

In the **word RAM** model, all elements are integers that fit in a machine word of w bits. So the universe of possible values is $\{0, 1, \ldots, u-1\}$ where $u = 2^w$. Basic C operations (comparison, arithmetic, bitwise) on such words take $\Theta(1)$ time. [1]

Note that we already assume the word RAM model for things like merge sort, since array indices occupy at least $\log n$ bits and thus are not constant size. This is not a radical departure from the word RAM model with $w \geq \log_2 n$.

5 Word RAM Solutions to the Predecessor Problem

Today, we introduce two solutions to the predecessor problem in the word RAM model. The first is the **van Emde Boas (vEB) tree** [1], a data structure with $\Theta(u)$ space requirements and $\Theta(\log w)$ runtime for *pred*, *insert*, and *del*. The second is Willard's **y-fast trie** [2], which has $\Theta(n)$ space requirements and (expected) $\Theta(\log w)$ runtime for *pred*, *insert*, and *del*.

On Thursday, we will introduce Fredman and Willard's **fusion tree** [3], which has $\Theta(n)$ space requirements and $\Theta(\log_w n)$ operation runtime.

With vEB and fusion trees, we can always achieve $O(\min\{\log w, \log_w n\})$ operation runtime for the static predecessor problem. The data structures are asymptotically equal in runtime when $\log w = \log n / \log w$, or $\log w = \sqrt{\log n}$. Patrascu and Thorup [4, 5] have shown that using the better of (slightly tweaked) vEB trees and static fusion trees is essentially optimal for near-linear space.

These results for the static predecessor problem do not necessarily imply faster sorting, since preprocessing can be arbitrarily expensive. However, dynamic fusion trees also exist and suggest an $O(n\sqrt{\log n})$ sorting algorithm. In fact, we can do better with Han's $O(n \log \log n)$ algorithm [6] or Han and Thorup's (expected) $O(n\sqrt{\log \log n})$ algorithm [7]. Can we do even better than this? Who knows.

[1] `https://www.tutorialspoint.com/cprogramming/c_operators.htm` (up to "Bitwise Operators") lists such word operations

5.1 vEB Tree Details

At the heart of the vEB tree is the division of the universe into $\sqrt{u}$ clusters. A word's **cluster** is the half-size word formed by leftmost $\frac{w}{2}$ bits, and a word's **id** is the half-size word formed by the rightmost $\frac{w}{2}$ bits. Our notation for decomposing a word x into cluster index and id is $x = \langle c, i \rangle$.

A vEB_u tree V can operate on values in the universe $[0, 1, ..., u-1]$ (i.e. $\log_2 u$ size words) and has the following members:

- $V.min$: type word. The element stored by this tree itself.
- $V.clusters$: type size-$\sqrt{u}$ array of $vEB_{\sqrt{u}}$. $V.clusters[c]$ stores the ids of all elements in cluster c (except for $V.min$).
- $V.summary$: type $vEB_{\sqrt{u}}$. Stores all non-empty cluster indices (except that of $V.min$).
- $V.max$: type word. The maximum element stored by this tree itself or any of its subtrees.

Of course, a vEB_u tree for $u \leq$ some small constant T can be almost anything (e.g. a dumb old array) – since operations will be $O(T)$ time and T is a constant, we can treat operations on such "trees" as constant time.

5.1.1 Pseudocode

```
def pred(V, x = <c, i>):
    if x <= V.min:
        return null
    if notempty(V.cluster[c]) and i > V.cluster[c].min:
        return <c, pred(V.cluster[c], i)>
    else:
        c’ = pred(V.summary, c)
        return <c’, V.cluster[c’].max>

def insert(V, x = <c, i>):
    if V.min = null:
        V.min = x
        V.max = x
        return
    if x < V.min:
        swap(x, V.min)
    if x > V.max:
        V.max = x
    if v.cluster[c].min = null:
        insert(V.summary, c)
    insert(V.cluster[c], i)

// del can be implemented in a similar manner
```

5.1.2 Analysis: Runtime

pred on a vEB_u tree makes at most 1 recursive call on a $vEB_{\sqrt{u}}$ tree and has constant non-recursive operations. So the runtime recurrence is:

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

This solves to $T(u) = O(\log \log u) = O(\log w)$.

insert on a vEB_u tree appears to make up to two recursive calls. However, if the first recursive call is made, then the second recursive call will be $O(1)$ because *V.cluster*$[c]$ is empty. So we can analyze it as if it had a single recursive call, and the Equation (1) recurrence applies, giving an insertion time of $O(\log \log u) = O(\log w)$.

5.1.3 Analysis: Space

Each vEB_u tree has $\sqrt{u} + 1$ subtrees of type $vEB_{\sqrt{u}}$, and other members comprising constant space. So the space recurrence is:

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + \Theta(1)$$

This solves to $S(u) = \Theta(u)$ – the same space as a lookup table! To improve, we can avoid storing empty subtrees entirely by making *V.clusters* a hashtable instead of an array and constructing new subtrees lazily. The Equation (1) recurrence also applies for the depth of the tree, and we create at most one new subtree at each level upon insertion. Thus each insert creates at most $\log w$ subtrees, and the total space requirement is $O(n \log w)$.

This optimization slightly changes the runtime analysis – since accesses to *V.cluster* are now *expected* constant time rather than worst-case constant time, operations are now *expected* $O(\log w)$ time. Consult [8] for an example of a word-RAM hashtable. [2]

5.2 x-fast Trie Details

The x-fast trie serves as an intermediate step in developing the y-fast trie. We build a "flat tree" array A with $2u - 1$ elements (similar to a binary heap structure). $A[0]$ is the root node, $A[\lfloor \frac{i}{2} \rfloor]$ is the parent node of $A[i]$, and $\{A[2i+1], A[2i+2]\}$ are the child nodes of $A[i]$. The leaf nodes are thus $A[u] \ldots A[2u-1]$ and leaf node $A[i]$ is 1 iff $i - u$ is in the tree. Leaf nodes with value 1 are connected in order via a doubly linked list. Finally, the value of inner node $A[i]$ is recursively defined as the OR of the children.

Insertion and deletion both involve an $O(\log u) = O(w)$ time traversal down the tree.

The *pred* query involves accessing the corresponding leaf node: if it's a 1, just follow the linked list to the predecessor. Otherwise, we can go up the tree until we find the lowest 1-value ancestor, then traverse the tree back down to a neighboring node and follow the linked list if needed.

[2]Hashtable is used loosely as "any solution to the dynamic dictionary problem", which comprises the $find$, *insert*, and *delete* operations.

Naively, this is an $O(\log u) = O(w)$ operation due to the depth of the tree. However, note that the node values are monotone between a leaf and the root. Thus, we can binary search for the smallest 1-value ancestor in $O(\log w)$ time. To traverse the tree back down, we can make the optimization that if a node has a 0-value left child, it will store the lowest 1-value leaf node of the right subtree (and vice-versa for a 0-value right child). Thus, we have optimized *pred* to $O(\log w)$ runtime.

However, the x-fast trie still occupies $O(u)$ space. To correct this, we can again replace the array A with a giant hashtable containing only the 1-value nodes. Since each insert adds $O(w)$ additional 1-values to the trie, the total space requirement becomes $O(nw)$. And as with vEB trees, the operation runtime becomes expected rather than worst-case.

5.3 y-fast Trie Nondetails

The y-fast trie is essentially an x-fast trie that stores BBSTs of words instead of words themselves. Each BBST stores consecutive stored words (in comparison order, not insertion order) of the y-fast trie. The BBSTs have $\Theta(w)$ elements, so the x-fast trie has $\Theta(\frac{n}{w})$ elements. Thus, the space requirement for the x-fast trie is $O(n)$. The extra BBST queries are $O(\log w)$, which does not worsen the query and insertion asymptotics. The full details of the y-fast trie will be presented on Thursday.

References

[1] Peter van Emde Boas. Preserving Order in a Forest in less than Logarithmic Time. *16th Annual Symposium on Foundations of Computer Science*, 75–84, 1975.

[2] Dan Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(n)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

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

[4] Mihai Patrascu, Mikkel Thorup. Time-space trade-offs for predecessor search. *38th Annual Symposium on Theory of Computing*, 232–240, 2006.

[5] Mihai Patrascu, Mikkel Thorup. Randomization does not help searching predecessors. *18th Annual Symposium on Discrete Algorithms*, 555–564, 2007.

[6] Yijie Han. Deterministic sorting in O(nlog log n) time and linear space. *34th Annual Symposium on Theory of Computing*, 602–608, 2002.

[7] Yijie Han, Mikkel Thorup. Integer Sorting in O(n sqrt (log log n)) Expected Time and Linear Space. *43rd Symposium on Foundations of Computer Science*, 135–144, 2002.

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