CS 270: Combinatorial Algorithms and Data Structures Spring 2023

Lecture 8 — February 9, 2023

Prof. Jelani Nelson *Scribe: Kishan Jani, Wilson Wu*

1 Overview

The next few weeks we will be talking about data structures. Today and Tuesday will be about Word RAM model. We want to talk about the predecessor problem.

2 The Predecessor Problem

The predecessor problem has the following parameters and objectives:

- We are given a database of $(\mathsf{key}, \mathsf{val})$ pairs, with keys ranging in $\{0, 1, 2, \ldots, u-1\}$.
- The goal is to efficiently (both in time and space) compute $\mathsf{pred}(x)$, which returns $\max\{z \in S : z < x\}$, where S is the set of keys.
- There are two variants of the problem: *static* and *dynamic*. For the static case, we are given all the keys from the start and no changes to the keys are allowed. For the dynamic case, we are also allowed to define $\mathsf{insert}(x)$ and $\mathsf{delete}(x)$ queries, doing exactly what the name suggests.
- For the static case, given the array of keys, simply sort the keys. Then for each query $\mathsf{pred}(x)$, binary search for the predecessor of x. This takes time $O(n \log n)$ for sorting and $O(\log n)$ for each query $\mathsf{pred}(x)$ using binary search, where n is the size of our database. We can achieve similar runtime bounds for dynamic case, where we use a binary search tree instead of a sorted array. For example, Red-Black trees and AVL trees give $O(\log n)$ time for all three queries.

How do you break ties: what if two items have the same key, and for $\mathsf{pred}(x)$, either item works? There are ways to resolve this, but we will not worry about this issue right now. The second thing to note here is that the above data structures and algorithms rely solely on comparison and sorting of keys. However, this is not the most reasonable case: typically, we are allowed many operations in $O(1)$ time. As we will see, exploiting these operations can lead to faster algorithms.

3 Word RAM Model

Approximate moral: Constant time operations in C (the programming language) are constant time in the Word RAM Model. Following are the assumptions of the model:

1. $u = 2^w$ is the largest supportable integer under bitwise operations. We take keys as integers between 0 and $u - 1$.

2. Our CPU has some built-in operations: LOAD, STORE, $+, -, \times, /$, bitshift, etc. These operate on "words" of w bits in O(1) time.

These assumptions end up leading to significant improvements in the Predecessor problem:

- [1] gave the van Emde Boas (vEB) tree, which achives $O(\log \log u)$ time per query and takes space $O(u)$. This can, however, be improved to $O(n)$ time, where n is the size of our input.
- [8] gets the same bounds using y-fast tries.
- [2] developed fusion trees (and came up with the coolest paper name ever), which achieve $O(\log_w n)$ queries for the static case.
- [7] came up with dynamic fusion trees to deal with the dynamic case of the problem, also in $O(\log_w n)$.

Proposition 3.1. *If you have a good dynamic solution to* pred *problem, you have a good sorting algorithm.*

Proof. How do we use dynamic pred to sort? Given as input the array to be sorted, we insert keys into the data structure one by one and keep track of the max. Then starting at the max, we keep calling $\mathsf{pred}(\cdot)$ to get elements in descending order. There are a total of n insertions and $n-1$ calls to the pred query, giving a runtime of $O\left(n\left[T_{\mathsf{insert}}(n) + T_{\mathsf{pred}}(n)\right]\right)$. □

Using this approach, vEB trees imply $O(n \log w)$ time sorting algorithm, and Fusion trees imply $O(n\frac{\log n}{\log w})$ time sorting. Thus you can get $O(n \min\{\log w, \frac{\log n}{\log w}\})$ by using the faster algorithm based on n and w. The worst case is when both are equally fast, which occurs for $\log w = \sqrt{\log n}$, yielding a runtime of $O(n\sqrt{\log n})$.

Assuming the word RAM model, other fast sorting algorithms not using predecessor data structures are also possible.

- [3] showed that you can sort in $O(n \log \log n)$ time deterministically.
- [4] sorted in $O(n\sqrt{\log \log n})$ with a randomized algorithm.
- Sorting in $O(n)$ time under Word RAM is still an open problem.

Unfortunately, the runtime of predecessor data structures is lower bounded.

Theorem 3.2 ([6],[5])**.** *For predecessor, we cannot do better than* $\min\{T_{vEB}, T_{fusion}\}$.

Now let us begin taking a closer look at these improvements under the word RAM model.

4 vEB Trees

4.1 Motivation

B-trees are a common data structure used in databases. Compared to binary trees, they have a general branching factor of B rather than 2. The depth then is $\log_B n$ instead of $\log n$. Although they achieve the same asymptotic runtime as binary search trees, they run a lot faster empirically. The reason is that it fits better on disks. For non-flash drive disks, there is a spinning magnetic

disk. To read something, you have to wait until the disk spins to where the reading portion is. If this takes a lot of time, in an amortized sense, it is a good idea to wait and read some adjacent data instead. B-trees optimize for this latency by balancing the extra time it takes to read adjacent data with the time it takes the disk to spin to the desired page. By taking into account this disc access model, B-trees are able to perform much better than binary search trees. Similarly, by taking into account the word RAM model, vEB trees allow for faster queries.

4.2 Construction

The basic idea here is divide and conquer. These are parametrized by the size of the universe, which is initially u (with $0 \leq \mathsf{key} \leq u-1$). They are recursively defined. Fields stored by the node vEB_u:

- max, which is the largest $\mathsf{key} \in S$.
- min, which is the smallest $\mathsf{key} \in S$.
- $\mathsf{cluster}[0, 1, \ldots, \sqrt{u}-1]$, an array of pointers to $\sqrt{u}$ total $\mathsf{vEB}_{\sqrt{u}}$ trees.
- summary, pointer to a $\mathsf{vEB}_{\sqrt{u}}$ which stores the indices of the non-empty trees in the cluster

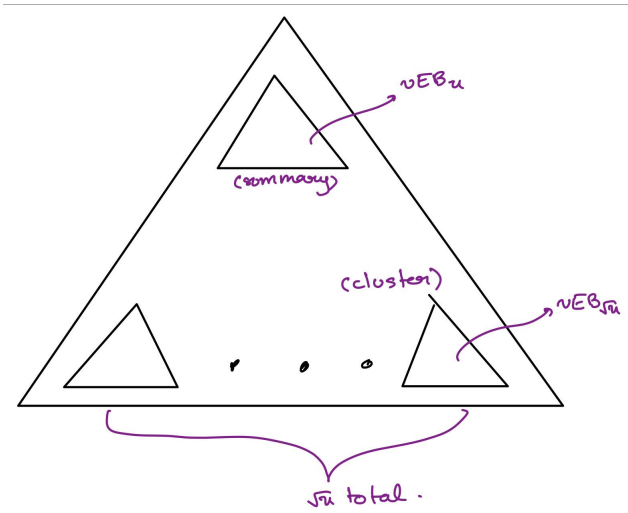

Figure 1: vEB tree

Now, we outline the process:

1. Treat $x \in \{0, \ldots, u-1\}$ (now denoted via $[N] = \{0, 1, \ldots n-1\}$) as $x = \langle c, i \rangle$, with both $c, i \in [\sqrt{u}]$. That is, we split x into the left c and right i halves of its bit representation.
2. If x is in the database, we store i recursively in $V.\mathsf{cluster}[c]$.
3. c is inserted into $V.\mathsf{summary}$ if S contains any key that starts with c. Keep all trees empty when you start.
4. If you have only one element in a tree, store it in min or max field.

4.3 Predecessor

Algorithm 1 $\mathsf{pred}(x)$

Require: x, which is a key from our database.
Ensure: $\mathsf{pred}(x) = \max\{z \in S : z < x\}$

1: Find the decomposition $x = \langle c, i\rangle$ in $O(1)$ time (under the Word RAM model, extractable in constant time by reading two halves via bit-shifts).
2: **if** $x > V.\max$ **then**
3: **return** $V.\max$
4: **if** $V.\mathsf{summary}$ contains c **then**
5: **if** $i > V.\mathsf{cluster}[c].\min$ **then**
6: **return** $\langle c, V.\mathsf{cluster}[c].\mathsf{pred}(i)\rangle$ *The predecessor is in cluster[c]*
7: **else**
8: $c' \leftarrow \mathsf{pred}(V.\mathsf{summary}.(c))$ *Find which cluster c', that the predecessor is in*
9: **return** $\langle c', V.\mathsf{cluster}[c'].\max\rangle$

Runtime analysis: There is a recursive step in line $6 + 8$ on $\mathsf{vEB}_{\sqrt{u}}$, while rest the operations are constant time. Hence the recurrence for runtime is

$$T(u) = T(\sqrt{u}) + O(1), \text{ thus } T(w) = T(w/2) + O(1).$$

By the master theorem, we have $T = O(\log w) = O(\log\log u)$. Notice that the coefficient of $T(\sqrt{u})$ is 1 because the recursive steps of lines 6 and 8 happen under disjoint conditions.

4.4 Insert

Algorithm 2 $\mathsf{insert}(V, x = \langle c, i\rangle)$

Require: $x = \langle c, i\rangle$, which is a key from our database, and a cluster.
Ensure: x is inserted into the cluster.

1: **if** $V.\min = \text{null}$ **then**
2: $V.\min, V.\max \leftarrow x$, **return** *base case: empty tree*
3: **if** $x < V.\min$ **then**
4: $\text{swap}(x, V.\min)$ *set V.min to x and insert the original V.min into a child tree*
5: **if** $x > V.\min$ **then**
6: $V.\max \leftarrow x$
7: **if** $V.\mathsf{cluster}[c] = \text{null}$ **then**
8: $V.\mathsf{summary}.\mathsf{insert}(c)$.
9: $\mathsf{insert}(V.\mathsf{cluster}[c], i)$

Runtime analysis: I have to do two recursive inserts, hence

$$T(u) = 2T(\sqrt{u}) + O(1), \quad T(w) \le 2T(w/2) + O(1),$$

which is time $O(w)$; this is bad. However, this is sloppy, we can are only ever actually making one recursive insert call through the if statements, hence it should be the same recurrence (thus same runtime) as before: since $T(u) = T(\sqrt{u}) + O(1)$, we have $T(w) = O(\log \log w)$. What is the space complexity? We have

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

for the $\sqrt{u} + 1$ clusters to deal with recursively. Furthermore, $O(\sqrt{u})$ is for all the cluster min and max values. Replacing u with w and setting $S(u) = S(2^{w/2}) = S'(w)$, we get

$$S'(w) \leq 2^{w/2} S'(w/2) + 2^{w/2},$$

the branching factors go by $2^{w/2}, 2^{w/4}$, and so on. The time then ends up being $O(2^w) = O(u)$. This is bad news.

We want to improve this, which is accomplished by storing cluster as a hash table instead of as an array. When we say hash table, we mean a solution to the dynamic dictionary problem. We have randomized solutions to hashing with linear space and expected constant time operations. We will talk about other non-170 ways to make good dictionaries later, but let's just accept it and move on for now. We have

$$V.\text{cluster} = \text{hashtable mapping } c \text{ to a pointer to child tree}$$

Space is now proportional to number of clusters rather than u. Memory proportional to sum of sizes of non-empty VEB trees throughout recursion At each level, since you either recurse into the summary or recurse into the cluster, the time is based on depth which yields $O(n \log w)$ memory.

5 x-fast tries

Here is a really simple way to solve pred. The database say is $\{0, 2, 3, 7, 8, 9, 12, 13\}$. Here $w = 4$ and $u = 16$. Let us store this as a bit array:

$$1011000111001100$$

Build a perfect binary tree where these bits are the leaves, and each internal node is the logical OR of two children bits. See figure 2.

What is pred of an element? Store the 1 positions in a w-linked list so that we can jump between w in $O(1)$ time. If someone asks what the predecessor of 6 is, we walk up the tree; we were its left child, which means if I go for the smallest in the right sub-tree, this gives the successor and jumping back to the previous 1 gives me the predecessor via my linked list. However, this process takes $O(w)$ time, based on tree depth.

Notice though the path to the root is monotone due to us using OR, so we can binary search on the ancestry of p.

- Store tree in a length $2u - 1$ array as for a binary heap.
- For a node indexed at x (indexed from 1), the left child is at $2x$ while the right child is at $2x + 1$. Thus in order to find parent, all I have to do is $\lfloor \cdot/2 \rfloor$, which is a bit-shift. Thus to find the kth ancestor, all I need is k bit-shifts, which takes time $O(k)$ under Word RAM. This gives $O(\log w) = O(\log \log u)$ time.

Queries are fast while inserts are slow. y-fast tries, that we will see next time, make fast insertion over linear space.

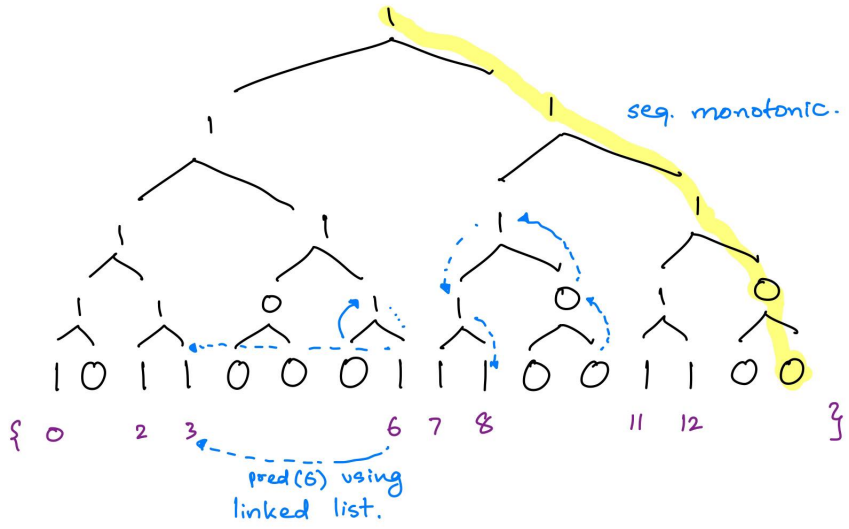

Figure 2: x-fast trie

References

[1] Peter van Emde Boas. "Preserving order in a forest in less than logarithmic time". In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. IEEE. 1975, pp. 75–84.

[2] Michael L Fredman and Dan E Willard. "Blasting through the information theoretic barrier with fusion trees". In: *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*. 1990, pp. 1–7.

[3] Yijie Han. "Deterministic sorting in O (n log log n) time and linear space". In: *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. 2002, pp. 602–608.

[4] Yijie Han and Mikkel Thorup. "Integer sorting in O (n/spl radic/(log log n)) expected time and linear space". In: *The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.* IEEE. 2002, pp. 135–144.

[5] Mihai Patrascu and Mikkel Thorup. "Planning for fast connectivity updates". In: *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. IEEE. 2007, pp. 263–271.

[6] Mihai Pătraşcu and Mikkel Thorup. "Time-space trade-offs for predecessor search". In: *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*. 2006, pp. 232–240.

[7] Rajeev Raman. "Priority queues: Small, monotone and trans-dichotomous". In: *Algorithms—ESA'96: Fourth Annual European Symposium Barcelona, Spain, September 25–27, 1996 Proceedings 4*. Springer. 1996, pp. 121–137.

[8] Dan E Willard. "Log-logarithmic worst-case range queries are possible in space Θ (N)". In: *Information Processing Letters* 17.2 (1983), pp. 81–84.