Lecture 21: van Emde Boas Trees

Sam McCauley

November 26, 2024

Williams College

Admin

- Any questions?

Predecessor and Successor Queries

Problem for today:

- Store a set S of size n (must be comparable items: for any $i, j \in S$ must have $i < j$, $i > j$, or $i = j$).
- Want to answer predecessor and successor queries. On a query q
  - Predecessor: Find the largest $i \in S$ such that $i \leq q$
  - Successor: Find the smallest $i \in S$ such that $i \geq q$
- Also want to be able to insert and delete items
- In CS 136 we saw how to answer this using a balanced binary search tree in $O(\log n)$ time
- This is optimal if all you can do is *compare* items

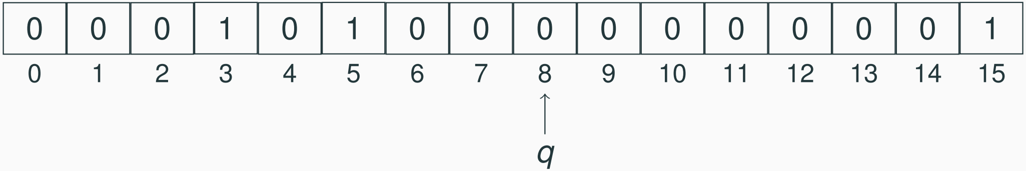
Generalizing the model

- This assumption is often too restrictive! Often we want to perform predecessor queries on integers or strings
- Know much more about the relative values of integers or strings
- Today: let's say that the items of S are taken from a bounded set $\{0, \ldots, M-1\}$
- For example: if the items of S are 64-bit integers, then we have $M = 2^{64}$. If items of S are k-character strings, we have $M = 256^k$.
- In this case, we will show how to get predecessor and successor in $O(\log \log M)$ time.
    - For a w-bit integer, get $O(\log w)$ time
    - For a k-character string, get $O(\log k)$ time

Data structure for today

- Van Emde Boas tree!
- Clever data structure. Very good constants, but still used sometimes in practice
- We'll only look at inserts, successor. Can generalize to predecessor queries and deletes.
- Let's not worry about space today (we'll wind up with $O(M)$ space). Some techniques to achieve $O(n)$ space.
- Also, let's assume that $\log_2 \log_2 M$ is an integer (M is 2 to a power of 2; like 2^8 or 2^{64})

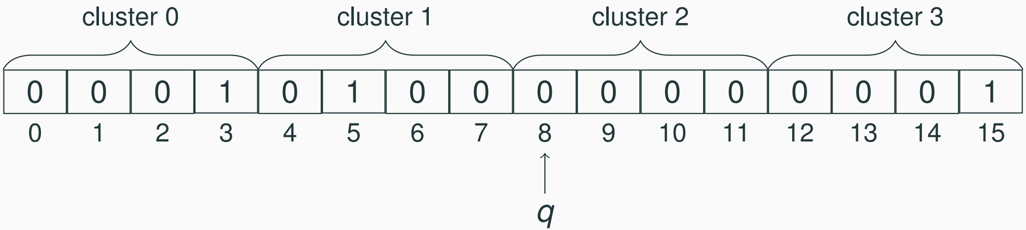
First attempt at Insert, Successor



- Let's keep a bit array A of length M
- $A[i] = 0$ if $i \notin S$, $A[i] = 1$ if $i \in S$
    - Time for insert?
    - $O(1)$
    - Time for successor?
    - $O(M)$
- Insert is really fast. Can we try to speed up successor?

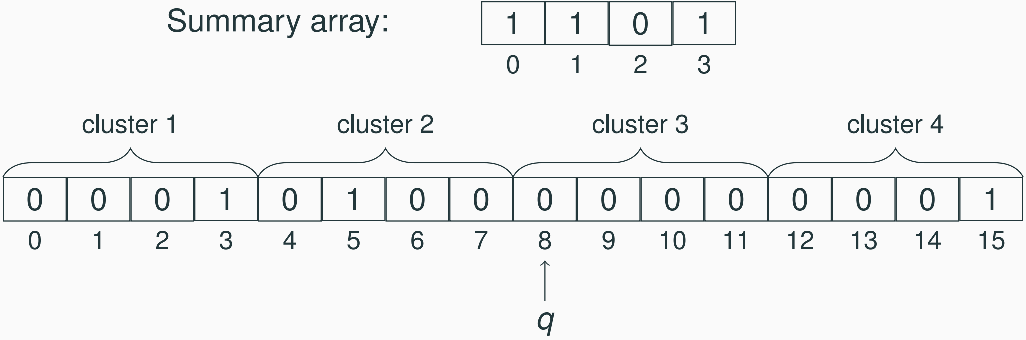
Second attempt at Insert, Successor

- Split our array into "clusters" of $\sqrt{M}$ elements.
- Let's do a "two-level" query for the successor:
    - First, find which cluster q is in
    - If the successor of q is there then we are done ($O(\sqrt{M})$ time)
    - Otherwise, find the next nonempty cluster
    - Then, query within the correct cluster for the minimum element ($O(\sqrt{M})$ time as before)
    - How can we query for minimum using a successor query?
    - How can we find the next nonempty cluster?



Second attempt at Insert, Successor

- We want to find the next nonempty cluster
- That's a successor query!
- Let's create a second, identical data structure to hold whether or not each cluster is empty



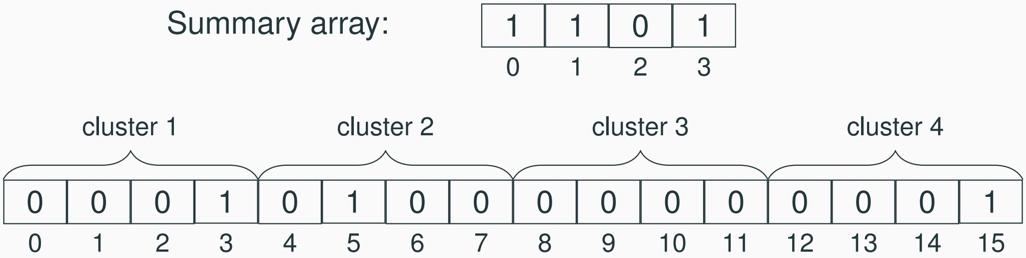
Second attempt at Insert, Successor

$O(1)$ insert, $O(\sqrt{M})$ successor query:

Successor:

- Figure out which cluster q is in (can calculate: $\lfloor q/\sqrt{M} \rfloor$)
- (These are the top $w/2$ bits of q if q is an integer, or the first $k/2$ characters if q is a string.)
- Check for the successor of q in q's cluster
- If it's not found:
    - Find the next nonempty cluster by looking in the summary array ($O(\sqrt{M})$ time)
    - Find the successor of q by looking for the smallest element in that cluster
- $O(\sqrt{M})$ time

Second attempt at Insert, Successor



Second attempt at Insert, Successor

$O(1)$ insert, $O(\sqrt{M})$ successor query:

Insert:

- Set the q bit in the overall array
- Figure out which cluster q is in (can calculate: $\lfloor q/\sqrt{M} \rfloor$)
- (These are the top $w/2$ bits of q)
- Set the cluster bit in the summary array

Where to go from here?

- Insert is still really fast, we want to improve successor.
- Where can we improve?
- All our time is spent doing array scans for successor queries within a cluster...
- But we know how to do better-than-linear successor queries! Let’s recurse.

Recursing: van Emde Boas Tree (almost)

If $M = 1$, just store the array.

Otherwise:

- Store a summary vEB tree of size $\sqrt{|M|}$ to keep track of which clusters are full
- For each cluster, store a vEB tree of size $\sqrt{M}$
- (Keep an array with a pointer to each of these vEB trees)
- Let's draw a picture of it on the board

(almost) vEB Tree Insert

- To insert, we need to recursively insert into the summary vEB tree, and we need to insert into the appropriate cluster
- Recurrence:
- $T(M) = 2T(\sqrt{M}) + O(1)$
- Solves to $O(\log M)$ insert time (too slow!)

(almost) vEB Tree Successor

- To find the successor of q, we need to:
  - Query the main cluster to see if the successor is there
  - If not found, find the next nonempty cluster using a successor query on the summary vEB tree
  - Then query that cluster for the minimum element
- Let's draw what this might look like on the board.
- Recurrence:
- $T(M) = 3T(\sqrt{M}) + O(1)$
- Solves to $O((\log M)^{\log_2 3}) = O(\log^{1.585} M)$ insert time (way too slow!)

The Problem

- Too many recursive calls!
- Can we get rid of some of them? Let's focus on successor

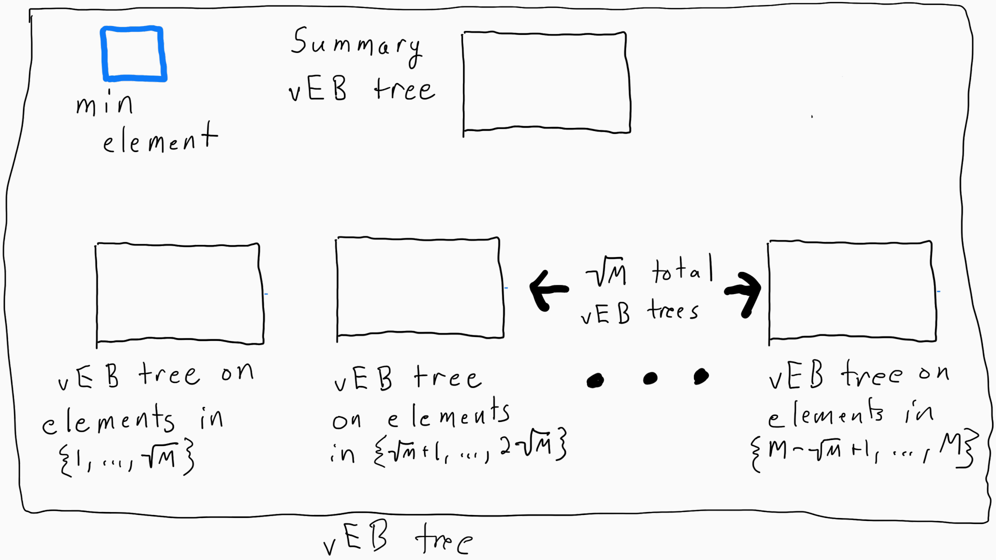
(almost) vEB Tree Successor

- To find the successor of q, we need to:
  - Query the main cluster to see if the successor is there
  - If not found, find the next nonempty cluster using a successor query on the summary vEB tree
  - Then query that cluster for the minimum element
- Finding the minimum element doesn't require a whole successor call! Let's just store the minimum element in each cluster. Then finding the minimum element is $O(1)$.

vEB Tree: Adding Minimum Element

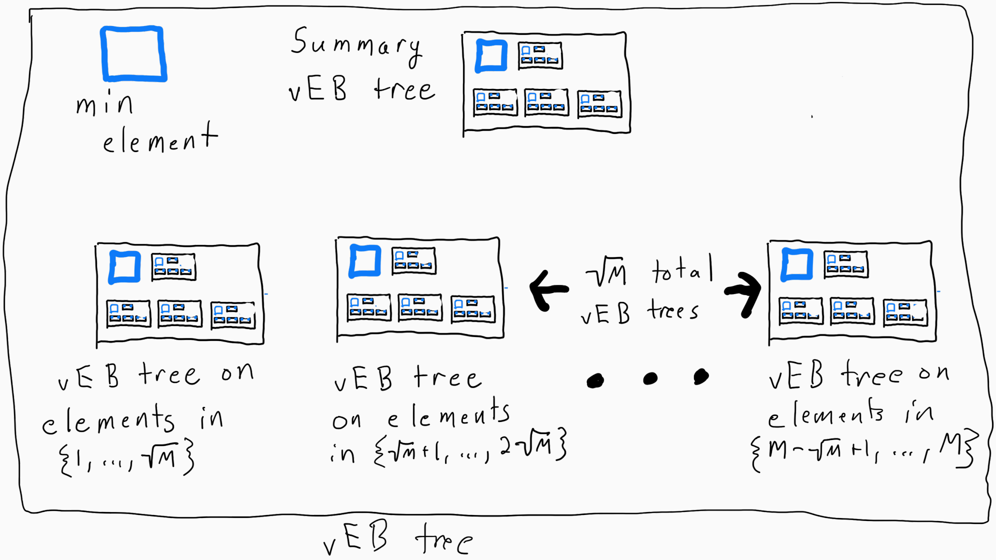
- On insert: proceed like before (insert into summary cluster; insert into the cluster itself). But, every time you insert into a cluster, check to see if the element we're inserting is the new minimum. If so, swap it out.
- Successor: we still query the main cluster. If the successor is not found, use a successor query in the summary vEB tree to find the next nonempty cluster. Return the minimum element in that cluster.
- Recurrence for both: $T(M) = 2T(\sqrt{M}) + O(1)$; solves to $T(M) = \log M$.

vEB Tree



min element
Summary vEB tree
vEB tree on elements in {1, ..., √M}
vEB tree on elements in {√M+1, ..., 2√M}
√M total vEB trees
vEB tree on elements in {M−√M+1, ..., M}
vEB tree

vEB Tree


min element
Summary vEB tree
vEB tree on elements in {1, ..., √M}
vEB tree on elements in {√M+1, ..., 2√M}
√M total vEB trees
vEB tree on elements in {M−√M+1, ..., M}
vEB tree

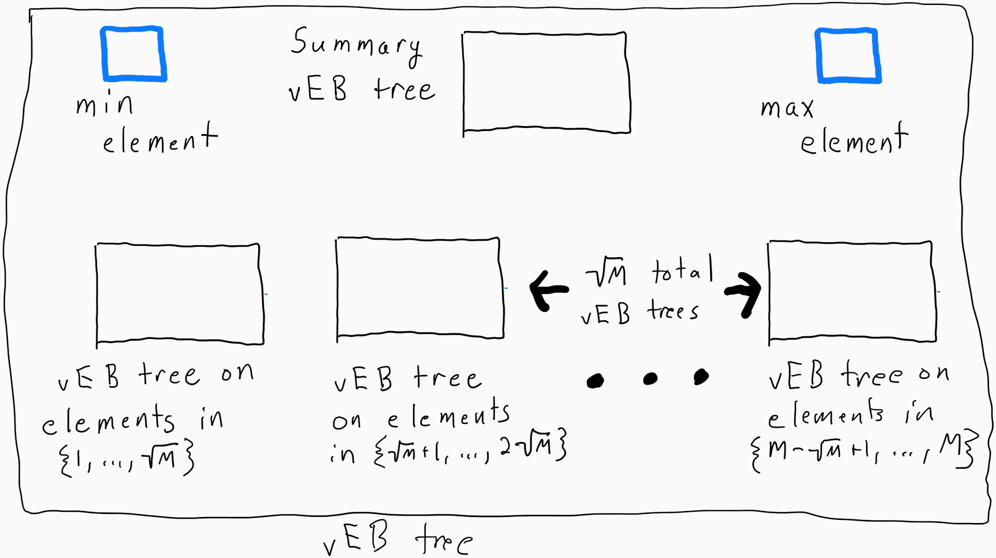
Getting to $\log \log M$

- Target recurrence?
- $T(M) = T(\sqrt{M}) + O(1)$. This solves to $O(\log \log M)$.
- Goal: get rid of second recursive call in insert and successor query
- On query: we still query the main cluster. If the successor is not found, use a successor query in the summary vEB tree to find the next nonempty cluster. Return the minimum element in that cluster.
- How can we make this just one call?
- *Hint:* Can we store something to help us determine if q has a successor in its cluster without a recursive query?
    - Store the *max element* in each cluster!

vEB Tree: Store the Max and Min in each cluster

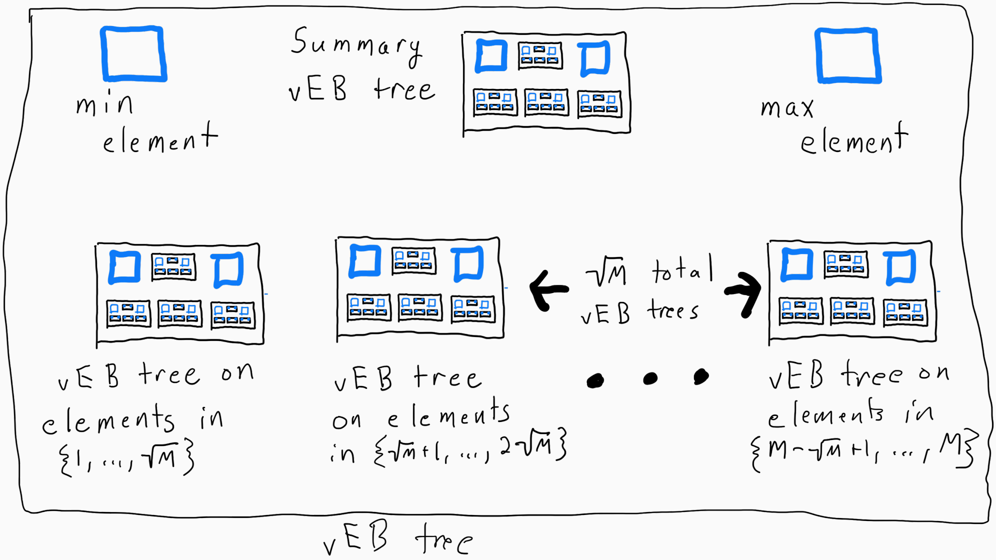
- On query: find q's cluster.
- If q is less than the max, find successor(q) in that cluster and return it
- Otherwise, use a successor query on the summary vEB tree to find the next nonempty cluster
- Return the minimum element in that cluster
- Example on board: store $3, 5, 15$ from universe $\{0, \ldots 15\}$; query for element 8.

vEB Tree



min element
Summary vEB tree
max element
vEB tree on elements in {1, ..., √M}
vEB tree on elements in {√M+1, ..., 2√M}
√M total vEB trees
vEB tree on elements in {M−√M+1, ..., M}
vEB tree

vEB Tree


min element
Summary vEB tree
max element
vEB tree on elements in {1, ..., √M}
vEB tree on elements in {√M+1, ..., 2√M}
√M total vEB trees
vEB tree on elements in {M−√M+1, ..., M}
vEB tree

Speeding up Insert

- Before: insert q in correct cluster; insert cluster into summary data structure
- How can we turn this into one recursive call?
- We only need to insert q into summary data structure if its cluster was empty
- In that case: just store q as min!
- Change to the algorithm: don't store minimum element recursively!
- Only need to recurse on summary data structure

Making sure successor still works

- Does successor still work if the minimum element is not stored recursively?
- No, but it's easy to fix: just check if $q <$ the minimum element. If so, the minimum element is the successor.
- Done!

van Emde Boas Tree Summary

- If $|M| = 1$, just store whether or not the one element is in our set
- Otherwise, have a "summary" vEB tree of size $\sqrt{M}$; and, divide M into $\sqrt{M}$ parts, with one vEB tree for each
- Plus the minimum and maximum elements in our structure, if they exist

van Emde Boas Tree Summary: Insert

To insert an item x:

- Find x's cluster c. If c has no minimum, set the minimum of c to be x, and insert c into the summary data structure.
- Otherwise:
    - Check if x is less than the minimum m.
    - If so, set x to be the minimum, and insert m into x's cluster.
    - Do the same for the maximum.
    - Otherwise, insert x into its cluster.

van Emde Boas Tree Summary: Successor

To find the successor of an item x:

- If x is less than the current minimum element m, return m.
- Find x's cluster c. If x is smaller than the maximum value in that cluster, query vEB tree c for the successor of x.
- Otherwise, query the summary vEB tree for the successor of c; call it c'. Return the minimum element of c'.

Analysis

- Successor does $O(1)$ work and makes one recursive call of size $\sqrt{M}$.
- $T(M) = T(\sqrt{M}) + O(1)$ gives $O(\log \log M)$ query time
- Insert does $O(1)$ work and makes one recursive call of size $\sqrt{M}$; also $O(\log \log M)$ time

Moving forward

- Predecessor queries?
- Pretty much identical
- What's the current space usage? Can we set up a recurrence?
- $S(M) = (\sqrt{M} + 1)S(\sqrt{M}) + O(\sqrt{M})$
- Solves to $O(M)$. Very bad!
- Deletes?
- Can make deletes work pretty easily with what we have.

Smaller space

- We won't go over this
- Basic idea: just use hashing! Only store nonempty clusters
- Can get $O(n)$ space
- Possible to get $O(n)$ space deterministically using another, more complicated data structure (y-fast tries)

Predecessor/Successor data structures

For a set S from $\{0, \ldots, M-1\}$:

- BBSTs: $O(\log n)$
- van Emde Boas trees: $O(\log \log M)$
- Takeaway: unless M is very large or n is very small, vEB trees are quite a lot faster
- But, they're probably a bit more complicated