Van Emde Boas Trees

15-750
2/4/2019

"loglog n has been proven to go to infinity, but has never been observed to do so." – Anonymous.

Ordered Dictionary

- Insert (x) ≡ add new element x
- Delete (x) ≡ remove x from set
- Member (x) ≡ output true iff x in the set
- Next (x) ≡ output smallest $y \geq x$ in the set (nil if none)
- Prev (x) ≡ output largest $y \leq x$ in the set (nil if none)
- max ≡ output largest element in the set (nil if empty)
- min ≡ output smallest element in the set (nil if empty)

Will focus on these ops

Applications

- Priority queues (and their applications)
- Sorting
- sorted key value store (By adding satellite data)
  (will not discuss)

Examples + Running Times

	Balanced BST	Sorted Array	Bit Array	Van Emde Boas Trees
Insert	$O(\log n)$	$O(n)$	$O(1)$	$O(\log\log U)$
Delete	$O(\log n)$	$O(n)$	$O(1)$	$O(\log\log U)$
Member	$O(\log n)$	$O(\log n)$	$O(1)$	$O(\log\log U)$
Next	$O(\log n)$	$O(\log n)$	$O(U)$	$O(\log\log U)$
Prev	$O(\log n)$	$O(\log n)$	$O(U)$	$O(\log\log U)$

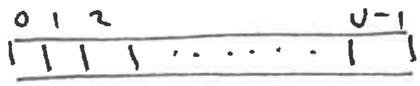
Today: Elements are all integers in range $\{0, 1, 2, \dots, U-1\}$

Note: By at most doubling U, may assume U is an integer power of 2. That is, $\log_2 U$ is an integer.

Question: Too good (fast) to be true?
Can sort in $O(n \log\log U)$ time?
What about $\Omega(n \log n)$ lower bound for sorting?

Answer: NOT comparison based (so lower bound doesn't apply).

Bit Array

A : 

<u>Idea</u>: $A[i] = 1 \iff i$ in the set.
(initially, $A[i] = 0$ for all i)

$O(1)$ Insert(i): $A[i] = 1$

$O(1)$ Delete(i): $A[i] = 0$

$O(1)$ Member(i): return $A[i]$

$O(U)$ next(i):
```
for j = i, i+1, ..., U-1
    if A[j] == 1
        return j
return nil
```

$O(U)$ prev(i) – symmetric.

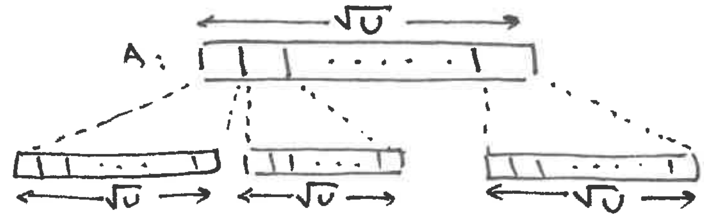
<u>Question</u>: next and prev take $O(U)$ time.
(For contrast, note that $U \geq n$ ...)
How do we improve this?

<u>Idea</u>: Break up range $\{0, 1, 2, \ldots, U-1\}$ into smaller pieces so that only need to search few pieces.

(<u>Next page</u>.)

Bit Array V2.0

Two levels:



A is array of size $\sqrt{U}$ of pointers to bit arrays for universe of size $\sqrt{U}$ (with count fields $\equiv$ num elements, initially equal zero)

Idea: $A[0]$ corresponds to range $\{0, 1, \ldots, \sqrt{U}-1\}$

$A[1]$ —— " —— $\{\sqrt{U}, \sqrt{U}+1, \ldots, 2\sqrt{U}-1\}$

$\vdots$

$A[\sqrt{U}-1]$ —— " —— $\{U-\sqrt{U}, U-\sqrt{U}+1, \ldots, U-1\}$

Element i represented by entry $\underbrace{i \bmod \sqrt{U}}_{\text{bottom } \frac{\log_2 U}{2} \text{ bits of } i.}$ in $A[\underbrace{\lfloor i/\sqrt{U} \rfloor}_{\text{top } \frac{\log_2 U}{2} \text{ bits of } i.}]$.

i $\underbrace{\boxed{\lfloor i/\sqrt{U} \rfloor \,\vdots\, i \bmod \sqrt{U}}}_{\log_2 U}$

$O(1)$ Insert(i): $B = A[\lfloor i/\sqrt{U} \rfloor]$

B.insert($i \bmod \sqrt{U}$)

$O(1)$ Delete, member – similar

$O(\sqrt{U})$ next(i):

$B = A[\lfloor i/\sqrt{U} \rfloor]$
j = B.next($i \bmod \sqrt{U}$) ← $O(\sqrt{U})$ (bit array of range size $\sqrt{U}$)
if $j \neq$ **nil**
  return $j + \lfloor i/\sqrt{U} \rfloor \cdot \sqrt{U}$
for $k = \lfloor i/\sqrt{U} \rfloor + 1, \lfloor i/\sqrt{U} \rfloor + 2, \ldots, \sqrt{U}-1$ — $O(\sqrt{U})$ $\times$ $O(1)$
  if $A[k]$.size $\neq 0$
    return $A[k]$.next$(0) + k\sqrt{U}$ — $+ O(\sqrt{U})$
return **nil**

$= O(\sqrt{U})$

Question: How can we do better for prev/next?

Answer: More levels!

Bit Array Vk.0

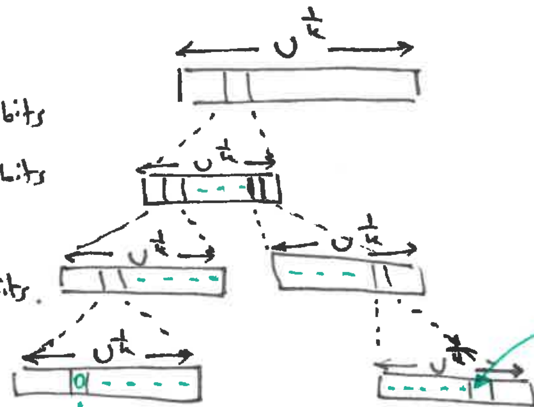
k levels *

1st level: Top $\frac{\log_2 U}{k}$ bits

2nd level: next $\frac{\log_2 U}{k}$ bits

⋮

last level: Bottom $\frac{\log_2 U}{k}$ bits.



location of next(i)

where i should be (if in the set)

* divisibility issue if k does not divide $\log_2 U$. We'll ignore this f

Insert, ~~delete~~ delete, member → $O(k)$ time ($O(1)$ operations / level)

next, prev $\leq O(1)\cdot 2k\cdot U^{\frac{1}{k}} = O(kU^{\frac{1}{k}})$ (see green text in illustration of ~~next(i)~~ above)

$O(1)$ → # scans/level, # levels, time for scan

Question: Best choice of k? (Want to minimize $kU^{\frac{1}{k}}$).

Answer: $\underset{k\geq 1}{\text{minimizing}}\ g(k) = kU^{\frac{1}{k}} \iff \underset{k\geq 1}{\text{minimizing}}\ \log_e(kU^{\frac{1}{k}})$.

Let $f(k) := \ln(kU^{\frac{1}{k}}) = \ln k + \frac{1}{k}\ln U$.

Deriving with respect to k, get:

$$f'(k) = \frac{1}{k} - \frac{1}{k^2}\ln U.$$

So $f'(k) = 0 \iff k = \ln U$.

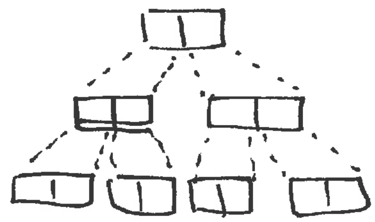
This is in fact a minimum of $f(k)$, and therefore $g(k)$, for which we get

$$g(k) = g(\ln U) = (\ln U)\cdot U^{\frac{1}{\ln U}} = (\ln U)\cdot e = O(\log U).$$

Another choice of k which achieves this asymptotic value for $g(k)$ is $k = \log_2 U$, for which we have

$$g(k) = g(\log_2 U) = (\log_2 U)\cdot U^{\frac{1}{\log_2 U}} = (\log_2 U)\cdot 2 = O(\log U).$$

Taking $k = \log_2 U$,
have arrays of size
$U^{\frac{1}{k}} = 2$.



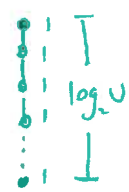
All operations take $O(k) = O(\log U)$ time
or $O(kU^{\frac{1}{k}}) = O(\log U)$ time.

$\Rightarrow$ We have re-invented balanced search trees!

Consider member(i) operation. Runtime given by

$$T(U) = T\left(\frac{U}{2}\right) + 1 = \Theta(\log U)$$

(Divide universe size by constant (2) every recursive call, which costs 1 per call)



Our Goal

Recurrences of the form

$$T(U) = T(\sqrt{U}) + 1 = \Theta(\log\log U)$$

(Divide exponent of input size by constant (2) every recursive call, which costs 1 per call)

Alternative proof - substitution method

Let $m := \log_2 U$ and $S(m) := T(2^m)$.

Then $S(m) = T(2^m) = T(U) = T(\sqrt{U}) + 1 = T(2^{\frac{m}{2}}) + 1$
$= S\left(\frac{m}{2}\right) + 1$

So $S(m) = S\left(\frac{m}{2}\right) + 1 = \Theta(\log m)$.

$\Rightarrow T(U) = T(2^m) = S(m) = \Theta(\log m) = \Theta(\log\log U)$.

Van Emde Boas Trees - Take 1

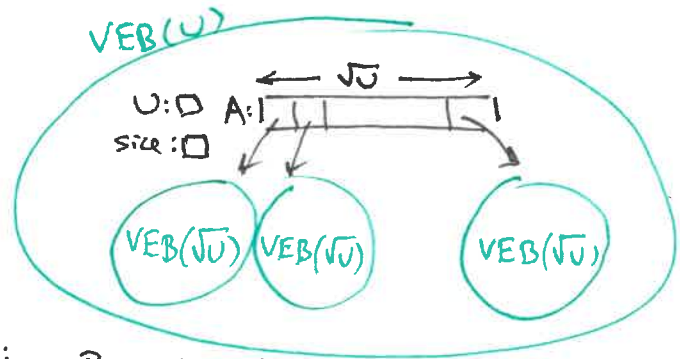
Takeaways from target recurrence, $T(U) = T(\sqrt{U}) + 1$:

(1) Different (universe size) structures at different levels $(U, \sqrt{U}, \sqrt[4]{U}, \sqrt[8]{U}, \ldots)$

(2) single recursive call.

(3) constant time per recursive level.

Let $VEB(U) \equiv$ Van Emde Boas Tree for universe size U.



Insert(i) : $B = A[\lfloor i/\sqrt{U} \rfloor]$

$B.\text{insert}(i \bmod \sqrt{U})$

delete, member - similarly.

Time : $T(U) = T(\sqrt{U}) + 1 = \Theta(\log \log U)$

next(i) : $B = A[\lfloor i/\sqrt{U} \rfloor]$

$j = B.\text{next}(i \bmod \sqrt{U})$

if $j \neq$ **nil**

return $j + \lfloor i/\sqrt{U} \rfloor \cdot \sqrt{U}$

for $k = \lfloor i/\sqrt{U} \rfloor + 1, \ldots, \sqrt{U} - 1$ ⊛

if $A[k].\text{size} \neq 0$

return $A[k].\text{next}(0) + k \cdot \sqrt{U}$

return **nil**

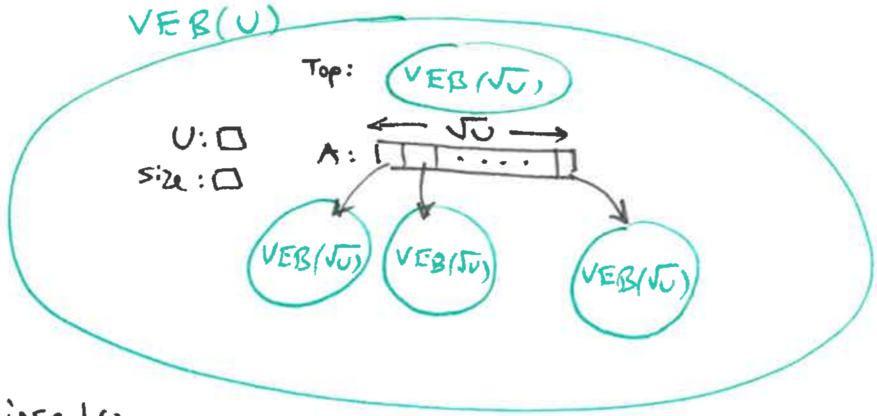
Time: $T(U) = 2T(\sqrt{U}) + \sqrt{U}$! ← due to ⊛

Fix: keep $VEB(\sqrt{U})$ of entries k of array A with $A[k].\text{size} \neq 0$. Call this Top.

Replace ⊛ by $k = \text{Top}.\text{next}(\lfloor i/\sqrt{U} \rfloor + 1)$,

followed by { if $k ==$ nil return **nil**; return $A[k].\text{next}(0)$

Van Emde Boas Trees - Take 2



Time: insert(i) now requires $\underline{2}$ recursive calls:
One to $A[\lfloor i/\sqrt{U}\rfloor]$ and one to top.

$$T(U) = 2T(\sqrt{U}) + 1 = \Theta(\log U)$$

Proof: substitution method. Again, $m := \log U$, $S(m) := T(2^m)$.
Get $S(m) = 2S(\frac{m}{2}) + 1 = \Theta(m) = \Theta(\log U)$.

Even worse, next(i) requires $\underline{3}$ recursive calls.

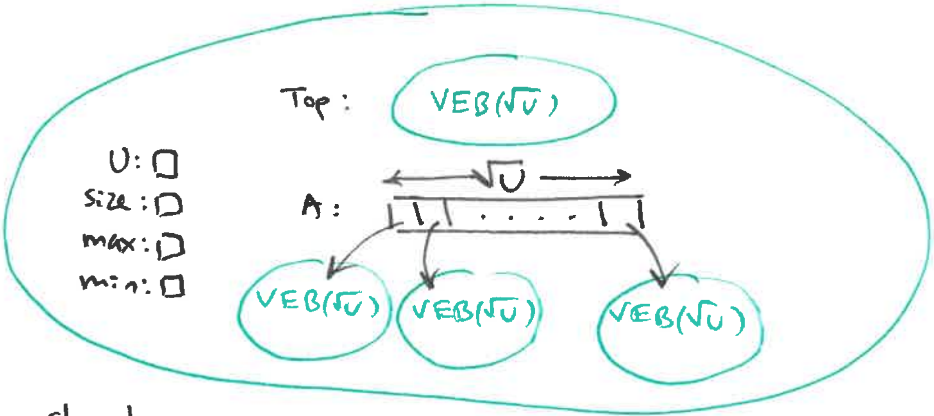
$$T(U) = 3T(\sqrt{U}) + 1 = \Theta((\log U)^{\log_2 3}) \approx \Theta((\log U)^{1.58}).$$

Proof: As above, this time also relying on the master theorem.
(Alternatively, recursion tree to analyze $S(m) = 3S(\frac{m}{2}) + 1 = \Theta(m^{\log_2 3})$.)

Question: How do we decrease #recursive calls?

Idea: Also maintain min and max fields.
Use to decrease # recursive calls.

Van Emde Boas Trees



Let us start by implementing next(i), given these fields.

next(i) :

$B = A[\lfloor i/\sqrt{U} \rfloor]$
if $B.max \geq i \bmod \sqrt{U}$
  return $B.next(i \bmod \sqrt{U}) + \lfloor i/\sqrt{U} \rfloor \cdot \sqrt{U}$ } in previous implementation always recursed here.
$k = Top.next(\lfloor i/\sqrt{U} \rfloor + 1)$
if $k \neq$ **nil**
  return $A[k].min + k \cdot \sqrt{U}$
return **nil**

in previous implementation, this was $A[k].next(0)$

Only 1 recursive call !
Either in $A[\lfloor i/\sqrt{U} \rfloor]$ or in Top.

$$\Rightarrow \quad T(U) = T(\sqrt{U}) + 1 = \Theta(\log\log U).$$

What about insert?
Here seems we need 2 recursive calls:
One to $A[\lfloor i/\sqrt{U} \rfloor]$ and one to Top, in case $A[\lfloor i/\sqrt{U} \rfloor]$ was empty before.

<u>Idea</u>: Save recursive call in $A[\lfloor i/\sqrt{U} \rfloor]$ if $A[\lfloor i/\sqrt{U} \rfloor]$ was empty before - by not inserting min/max into recursive structures!

Van Emde Boas (insert)

```
insert(i) :   if   size == 0
                   min = max = i
                   size = 1
                   return
              if   i < min
                   swap (i, min)
              if   i > max
                   swap (i, max)
```

$B = A(\lfloor i/\sqrt{U} \rfloor)$

B.insert $(i \bmod \sqrt{U})$

if B.size == 1

  Top.insert $(\lfloor i/\sqrt{U} \rfloor)$

size = size + 1

If B.size == 1 after B.insert$(i \bmod \sqrt{U})$, then B.insert$(i \bmod \sqrt{U})$ takes $O(1)$ time.

$\Longrightarrow$ Time $= T(U) = T(\sqrt{U}) + O(1) = \Theta(\log\log U)$.

- Delete(i) is symmetric.
- find(i) also requires only 1 recursive call.

```
find(i) :   if   i == min  or  i == max
                 return true
            B = A[⌊i/√U⌋]
            return B.find(i mod √U)
```

Summary

We saw an ordered dictionary with $O(\log\log U)$ time for all operations.

Takeaways

- Design + Analysis (go hand in hand)
  (Aimed for recurrence $T(U) = T(\sqrt{U}) + 1 = \Theta(\log\log U)$)
- Be suspicious of assumptions of lower bounds
  ($\Omega(n \log n)$ only for comparison-based algo.s)
- If need information often - make it easily accessible!
  (Examples:
  ① Top allowed us to quickly find next non-empty recursive VEB($\sqrt{U}$) to look at
  - rather than going over all.
  ② min/max saved us some recursive calls
  e.g. : - ~~calls~~ calls intended to output min/max
  - calls which find no larger element)

Next Lesson:

Dynamic Programming.