Van Emde Boas Trees

"Logarithm has been proven to go to infinity, but has never been observed to do so." — Anonymous

Ordered Dictionary

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST</th>
<th>Array</th>
<th>Array</th>
<th>Van Emde Boas Trees</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>1</td>
</tr>
<tr>
<td>Delete</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
<td>1</td>
</tr>
<tr>
<td>Member</td>
<td>O(log n)</td>
<td>1</td>
<td>O(1)</td>
<td>O(log log U)</td>
</tr>
<tr>
<td>Next</td>
<td>1</td>
<td>O(log n)</td>
<td>O(U)</td>
<td>1</td>
</tr>
<tr>
<td>Prev</td>
<td>1</td>
<td>1</td>
<td>O(U)</td>
<td>1</td>
</tr>
</tbody>
</table>

Today: Elements are all integers in range $[0, 1, 2, \ldots, U-1]$

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 $2^{O(n \log n)}$ lower bound for sorting?

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

\[ A : \begin{array}{ccccccc}
  0 & 1 & 2 & 3 & 4 & \cdots & N-1 \\
  1 & 1 & 1 & 1 & 1 & \cdots & 1 \\
\end{array} \]

Idea: \( A[i] = 1 \iff i \text{ is in the set.} \)
(initially, \( A[i] = 0 \) for all \( i \))

\[
\begin{align*}
\text{O(1) Insert}(i) & : \ A[i] = 1 \\
\text{O(1) Delete}(i) & : \ A[i] = 0 \\
\text{O(1) Member}(i) & : \ \text{return } A[i] \\
\text{O(N) next}(i) & : \ \text{for } j = i, i+1, \ldots, N-1 \\
& \quad \text{if } A[j] = 1 \\
& \quad \quad \text{return } j \\
& \quad \text{return nil} \\
\text{O(N) prev}(i) & - \text{symmetric.}
\end{align*}
\]

Question: next and prev take \( O(N) \) time.
(For contrast, note that \( N < N \ldots \))
How do we improve this?

Idea: Break up range \( 0, 1, 2, \ldots, N-1 \) into smaller pieces
so that only need to search few pieces.
(Next page.)
Bit Array V2.0

Two levels:

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

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

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

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

Element $i$ represented by entry $i \mod \sqrt{U}$ in $A[\lfloor i / \sqrt{U} \rfloor]$.

$0(1)$ Insert $(i)$: $B = A[\lfloor i / \sqrt{U} \rfloor]$

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

$O(1)$ Delete, member - similar

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

if $j \neq \text{nil}$

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

for $k = \lfloor i / \sqrt{U} \rfloor + 1, \lfloor i / \sqrt{U} \rfloor + 2, \ldots, \sqrt{U} - 1$

if $A[k]$.size $\neq 0$

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

end for

return $\text{nil}$

$O(\sqrt{U})$

$Q$: How can we do better for prev/next?

$A$: More levels!
Bit Array V.k.0

**k levels**

1st level: Top \( \log_2 U \) bits

2nd level: next \( \log_2 U \) bits

... 

Last level: Bottom \( \log_2 U \) bits.

Insert, delete, member \( \in O(k) \) time \((O(1) \text{ operations/level})\)

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

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

Answer: \( \min k \cdot \log_2 (k) \) \( k \leq 1 \) \( \iff \) \( \min k \cdot \frac{1}{k} \ln U \).

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

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 \( f(k) \), for which we get

\[ g(k) = g(\ln U) = (\ln U) \cdot \frac{1}{\ln U} = 1 = O(\log_2 U). \]

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

\[ g(k) = g(\log_2 U) = (\log_2 U) \cdot \frac{1}{\log_2 U} = 1 = O(\log_2 U). \]
Taking \( k = \log_2 U \), have arrays of size \( U^k = 2 \).

All operations take \( O(k) = O(\log U) \) time.

\[
\Rightarrow \text{We have re-invented balanced search trees!}
\]

Consider member(i) operation. Runtime given by

\[
T(U) = T\left(\frac{U}{2}\right) + 1 = O(\log U)
\]

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

\[
\text{Our Goal}
\]

Recurrences of the form

\[
T(U) = T\left(\sqrt[2]{U}\right) + 1 = \Theta(\log \log U)
\]

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

Alternative proof - substitution method

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

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

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 \( (U, \sqrt{U}, \sqrt[4]{U}, \ldots) \)
2. Single recursive call.
3. Constant time per recursive level.

Let \( VEB(U) \equiv \text{Van Emde Boas Tree for universe size } U \).

**Insert \( i \):**

\[ B = A[Li/\sqrt{U}] \]

\[ B. \text{insert}(i \mod \sqrt{U}) \]

**Delete, Member — Similarly.**

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

**next \( i \):**

\[ B = A[Li/\sqrt{U}] \]

\[ j = B.\text{next}(i \mod \sqrt{U}) \]

- if \( j \neq \text{nil} \)
  - return \( j + \lceil \frac{\sqrt{U}}{2} \rceil \)
- for \( k = \frac{Li}{\sqrt{U}} + 1, \ldots, \sqrt{U} - 1 \)
  - if \( A[k].\text{size} \neq 0 \)
    - return \( A[k].\text{next}(0) + k.\sqrt{U} \)
  - return \( \text{nil} \)

**Time:** \( T(U) = 2T(\sqrt{U}) + \sqrt{U} \) — due to \( \ast \)

**Fix:**

Keep \( VEB(\sqrt{U}) \) of entries \( k \) of array \( A \) with \( A[k].\text{size} \neq 0 \).

Call this for:

Replace by \( k = \text{top}.\text{next}(\lceil i/\sqrt{U} \rceil + 1) \),
followed by \( \text{if } k = \text{nil}, \text{return } \text{nil} \)
followed by \( \text{return } A[k].\text{next}(0) \).
**Van Emde Boas Trees - Table 2**

**Time:**

- `insert()` now requires 2 recursive cells: one to `A(Li/U)` and one to top.

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

**Proof:** substitution method. Again, \( m = \log U \), \( s(m) = T(2^m) \).

Get \( s(m) = 2 s(\frac{m}{2}) + 1 = \Theta(m) = \Theta(\log U) \).

Even worse, `next(i)` requires 3 recursive cells.

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

**Proof:** as above, this time also relying on the master theorem.
(Alternatively, recursion tree to analyze \( s(m) = 3 s(\frac{m}{2}) + 1 = \Theta(m^{1.52}) \).)

**Question:** How do we decrease the recursive calls?

**Idea:** Also maintaining min and max fields. Use to decrease the recursive calls.
Let us start by implementing `next(i)`, given these fields.

```plaintext
next(i) : B = \( A\left[\lfloor i / \sqrt{U} \rfloor\right]\)

if \( B\).max > \( i \mod \sqrt{U} \)
    return \( B\).next(i mod \( \sqrt{U} \)) + \( \sqrt{U} \cdot \sqrt{U} \)

k = Top.next(\( \lfloor i / \sqrt{U} \rfloor + 1 \))
if k ≠ nil
    return \( A[k]\).min + k \cdot \sqrt{U}

return nil
```

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

\[
\Rightarrow \ T(U) = T(\sqrt{U}) + 1 = \Theta(\log \log U)
\]

What about `insert`?
Here seems we need 2 recursive calls:
One to \( A\left[\lfloor i / \sqrt{U} \rfloor\right] \) and one to `Top`.
In case \( A\left[\lfloor i / \sqrt{U} \rfloor\right] \) was empty before.

Idea: Save recursive call in \( A\left[\lfloor i / \sqrt{U} \rfloor\right] \) if
\( A\left[\lfloor i / \sqrt{U} \rfloor\right] \) was empty before - by
not inserting min/max into recursive structures!
\[ \text{insert}(i): \quad \text{if} \quad \text{size} = 0 \]
\[ \text{min} = \text{max} = i \]
\[ \text{size} = 1 \]
\[ \text{return} \]
\[ \text{if} \quad i < \text{min} \]
\[ \text{swap}(i, \text{min}) \]
\[ \text{if} \quad i > \text{max} \]
\[ \text{swap}(i, \text{max}) \]
\[ \text{B} = A[(L^i) / \sqrt{U}] \]
\[ \text{B.insert}(i \mod \sqrt{U}) \]
\[ \text{if} \quad \text{B.size} = 1 \]
\[ \text{Top.insert}(L^i / \sqrt{U}) \]
\[ \text{size} = \text{size} + 1 \]

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

\[ \Rightarrow \text{Time} = T(U) = T(\sqrt{U}) + O(1) = \Theta(\log \log U). \]

- \( \text{Delete}(i) \) is symmetric.
- \( \text{find}(i) \) also requires only 1 recursive call.

\[ \text{find}(i): \quad \text{if} \quad i = \text{min} \text{ or } i = \text{max} \]
\[ \text{return true} \]
\[ \text{B} = A[(L^i) / \sqrt{U}] \]
\[ \text{return} \quad \text{B.find}(i \mod \sqrt{U}) \]
**Summary**

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

**Takeaways**

- **Design + Analysis** (go head in hand)
  
  (Aimed for recurrence $T(U) = T(\frac{U}{2}) + 1 = O(\log \log U)$)

- Be suspicious of assumptions of lower bounds ($\Omega(n \log n)$ only for comparison-based algo,s)

- If need information often - make it accessible!
  
  (Examples:
  1. **Top** allowed us to quickly find next non-empty recursive VEB($\frac{U}{2}$) to look at - rather than going over all.
  2. **min/max** saved us some recursive calls e.g.: - calls intended to output min/max - calls which find no larger element)

**Next Lesson:**

Dynamic Programming.