Supporting more operations

In a dynamic dictionary data structure we store (key, value)-pairs such that for any key there is at most one pair (key, value) in the dictionary.

Three operations are supported:
- **add** 
  Add the pair \((x, v)\) where \(x \in U\), the universe
- **lookup** 
  Return \(v\) if \((x, v)\) is in dictionary, or NULL otherwise.
- **delete** 
  Remove pair \((x, v)\) (assuming \((x, v)\) is in the dictionary).

What happens if we add more operations?

We also want our data structure to support:
- **predecessor** \(k\) - returns the (unique) element \((x, v)\) in the dictionary with the largest key, \(x\) such that \(x \leq k\)
- **successor** \(k\) - returns the (unique) element \((x, v)\) in the dictionary with the smallest key, \(x\) such that \(x \geq k\)

What could we use instead?

We could use a self-balancing binary tree... like a 2-3-4 tree, a red-black tree or an AVL tree

van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree which stores a set \(S\) of integer keys from a universe \(U = \{1, 2, 3, \ldots, u\}\) s.t. \(u = |U|\).

Five operations will be supported:
- **add** \(x\) - Insert the integer \(x\) into \(S\) (where \(x \in U\))
- **lookup** \(x\) - Return \(v\) if \((x, v)\) is in \(S\), or NULL otherwise.
- **delete** \(x\) - Remove \(x\) from \(S\)
- **predecessor** \(k\) - Return the largest integer \(x\) in \(S\) such that \(x \leq k\)
- **successor** \(k\) - Return the smallest integer \(x\) in \(S\) such that \(x \geq k\)

Warning: As stated the operations do not store any data (values) with the integers (keys). It is straightforward to extend the van Emde Boas tree to store (key, value) pairs when the keys are integers from \(U\).

(but I think it’s easier to think about like this)
van Emde Boas Trees

In this lecture, we will see the van Emde Boas (vEB) tree which stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots n\}$ (i.e. $n = |U|$).

Five operations will be supported:

- **add($x$)** Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)** Return yes if $x$ is in $S$, or no otherwise.
- **delete($x$)** Remove $x$ from $S$
- **predecessor($k$)** Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)** Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations will take $O(\log \log n)$ worst-case time and it is a deterministic data structure.

Examples: if $U = \{1, 2, 3, 4 \ldots 100 \cdot n\}$, you get $O(\log \log n)$ time and $O(n)$ space.

### Attempt 1: a big array

Build an array of length $u$...

$A[i] = 1$ if $i$ is in $S$

```
0 1 1 1 0 1 0 0 1 0
```

The operations **add**, **delete** and **lookup** all take $O(1)$ time.

. . . looks good so far!

The **predecessor** and **successor** operations take $O(u)$ time.

. . . not so good!

### Attempt 2: a balanced binary tree

(on top of a big array)

Build a balanced binary tree on top of the array...

Each node is 1 if either child is 1

[i.e. the subtree contains a 1]

The operations **add** and **delete** take $O(\log u)$ time.

The **lookup** operation still takes $O(1)$ time (simply look at $A[x]$)

The operations **predecessor** and **successor** take $O(\log u)$ time.

### Attempt 3: a constant height tree

(on top of a big array)

$C$ is called the summary of $A$

this is 1 if any bit in the child block is 1

```
1 1 0
```

Split $A$ into $\sqrt{u}$ blocks each containing $\sqrt{u}$ bits.

The **lookup** and **add** operations take $O(1)$ time.

The operations **delete**, **predecessor** and **successor** take $O(\sqrt{u})$ time.

### An abstract view

Split the universe $U$ into $\sqrt{u}$ blocks each associated with $\sqrt{u}$ elements

we can think of each block as a 'little' universe of size $\sqrt{u}$

there is a whole lot more universe in here

```
\sqrt{u} \sqrt{u} \sqrt{u} \sqrt{u} \sqrt{u}
```

For block $i$, we build a data structure $B[i]$

which stores elements from $\{1, 2, 3 \ldots \sqrt{u}\}$

$x$ is stored in $B[i]$ if $x + (i-1)\sqrt{u} \in S$

We also build a summary data structure $C$

which stores elements from $\{1, 2, 3 \ldots \sqrt{u}\}$

$x$ is stored in $C$ if $B[i]$ is non-empty
Eventually (after some more work), this will lead to an $O(\log \log n)$ time solution.

How efficient are the operations?

The add operation makes up to two recursive calls and the predecessor operation makes up to three. How should we build $B[1], B[2], \ldots, B[\sqrt{\mathcal{U}}]$ and $C$?

Each $B[i]$ has universe $\{1, 2, 3, \ldots, \sqrt{\mathcal{U}}\}$.

We recursively split this into $\sqrt{\mathcal{U}}$ blocks each associated with $\sqrt{\mathcal{U}}$ elements...

...eventually (after some more work), this will lead to an $O(\log \log n)$ time solution.

To perform add$(x)$:

1. Determine which $B[i]$ the element $x$ belongs in.
2. If $B[i]$ is empty, add $i$ to $C$.
3. Add $x$ to $B[i]$ (suitably adjusting the offset from the start of $B[i]$).

To perform predecessor$(x)$:

1. Determine which $B[i]$ the element $x$ belongs in.
2. Compute the predecessor of $x$ in $B[i]$ (suitably adjusting the offset from the start of $B[i]$).
3. If $x$ has no predecessor in $B[i]$, compute $j = \text{pred}(i)$ in $C$.

A closer look at predecessor

Observation 1: if $x$ has a predecessor in $B[i]$, we only make one recursive call.

The operations \( \text{lookup}, \text{delete}\) and \( \text{successor}\) can all also be defined in a similar, recursive manner.
To perform add(x):

Step 0 If \( x < \min \) then swap \( x \) and \( \min \)
Step 1 Determine which \( B[i] \) the element \( x \) belongs in
Step 2 If \( B[i] \) is empty, add \( x \) to \( C \)
Step 3 If \( B[i] \) is not empty, add \( x \) to \( B[i] \)

Step 4 Update the max

we need to get rid of one of these recursive calls

Finally: van Emde Boas Trees

So that we can find the min/max quickly we store them separately...

Remember that each \( B[i] \) and \( C \) are also vEB (van Emde Boas) trees over the universe \( \{1, 2, 3, \ldots, \sqrt{\pi}\} \)

In particular \( B[i] \) also stores its min/max elements separately so recovering the minimum or maximum in \( B[i] \) (or \( C \) ) takes \( O(1) \) time.

There is one more important thing, the minimum is not also stored in \( B[i] \), this allows us to avoid making multiple recursive calls when adding an element.

We have seen that the operations add and predecessor can be defined so that they make only one recursive call

The operations lookup, delete and successor can all also be defined in a similar, recursive manner so that they make only one recursive call

How long do the operations take?
Time Complexity

Let $T(u)$ be the time complexity of the predecessor operation (where $u$ is the universe size)

We have that, $T(u) = T(\sqrt{u}) + O(1)$

Using substitution and the master method you can show that... $T(u) = O(\log \log u)$

This holds for all the operations

Space Complexity

Let $Z(u)$ be the space used by a vEB tree over a universe of size $u$

We have that, $Z(u) = (\sqrt{u} + 1) \cdot Z(\sqrt{u}) + O(1)$

If you solve this you get that... $Z(u) = O(u)$

van Emde Boas Trees

The van Emde Boas (vEB) tree stores a set $S$ of integer keys from a universe $U = \{1, 2, 3, 4 \ldots u\}$ (i.e. $u = |U|$).

Five operations are supported:

- **add($x$)**: Insert the integer $x$ into $S$ (where $x \in U$)
- **lookup($x$)**: Return yes if $x$ is in $S$, or no otherwise.
- **delete($x$)**: Remove $x$ from $S$
- **predecessor($k$)**: Return the largest integer $x$ in $S$ such that $x \leq k$
- **successor($k$)**: Return the smallest integer $x$ in $S$ such that $x \geq k$

All operations take $O(\log \log u)$ worst case time and the space used is $O(u)$

The space can be improved to $O(n)$ using hashing (see y-fast trees)