van Emde Boas Trees

- **Intro => *Ordered Set***

Given a universe $U = [u]$ where $u = 2^w$ maintain subset $S \subseteq U, |S| = n$ under:

`member(x, S)`

`insert(x, S)`

`delete(x, S)`

`empty(S)`

`min(S)`

`max(S)`

`predecessor(x, S)`

`successor(x, S)`

- ***Naive***

If we are willing to spend $O(|U|)$ space

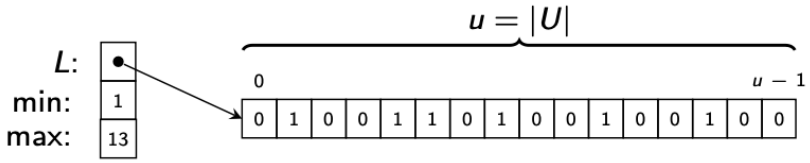
Store S as a bit-array L of length $|U|$ such that $L[x] = [x \in S]$, and keep track of the min and max values explicitly.

用一组大小为 u 的数列存 $0, 1$ 值， 0 表示该位置所表达的值不在 S 中，若为 1， 则 $x \in S$。

Naive

Idea: If we are willing to spend $\mathcal{O}(|U|)$ space...

Store S as a bit-array L of length $|U|$ such that $L[x] = [x \in S]$, and keep track of the min and max values explicitly.



$\{1, 4, 5, 7, 10, 13\}$

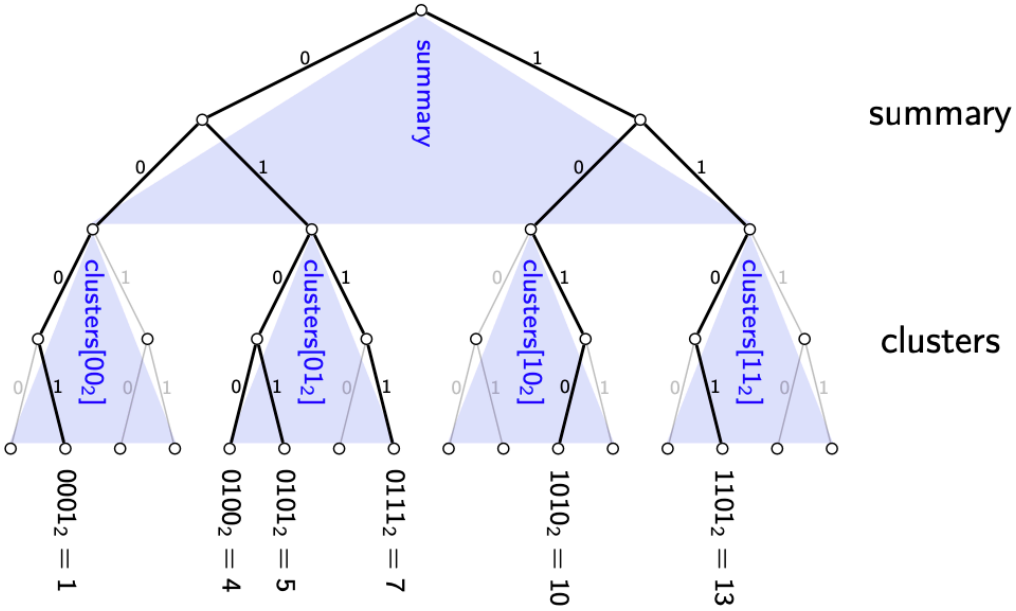
How fast is:
empty(S), min(S), and max(S)? worst case $\mathcal{O}(1)$.
member(x, S)? worst case $\mathcal{O}(1)$.
predecessor(x, S) and successor(x, S)? worst case $\Theta(|U|)$.
delete(x, S)? worst case $\mathcal{O}(|U|)$.
insert(x, S)? worst case $\mathcal{O}(1)$.

For `predecessor` and `successor`, the worst case is the predecessor (successor) is in the head (end) of the array. `Delete` takes $O(|U|)$ time for the worst case as it needs to maintain the min and max. Go through the array to find the min and max. While `insert` only takes $O(1)$. The step is to set the corresponding value to 1, and compare it with the min and max in the data structure.

- ***Bit-Trie***

Bit-Trie

Idea: Think of each key as a w-bit string describing a path in a binary *trie* (= a special kind of tree). The naive structure ignores all intermediate branches and jump directly to the leaves. What if we split each key into a high and a low part?



- ***Twolevel***

Twolevel

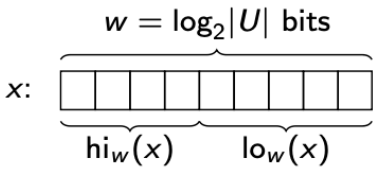
Idea: Split each key into *high* and *low* parts. Use naive for each.

Recall that $U = [2^w]$, and define:

$$\mathsf{hi}_w(x) := \left\lfloor \frac{x}{2^{\lceil w/2 \rceil}} \right\rfloor$$

$$\mathsf{lo}_w(x) := x \bmod 2^{\lceil w/2 \rceil}$$

$$\mathsf{index}_w(h, \ell) := h \cdot 2^{\lceil w/2 \rceil} + \ell$$



note that $x = \mathsf{index}_w(\mathsf{hi}_w(x), \mathsf{lo}_w(x))$.

Now let the structure directly store the values:

$$\mathsf{min} := \mathsf{min}(S) \text{ if } S \neq \emptyset, \text{ else } 1$$

$$\mathsf{max} := \mathsf{max}(S) \text{ if } S \neq \emptyset, \text{ else } 0$$

and use the naive structure to store

$$\mathsf{summary} := \{\mathsf{hi}_w(x) \mid w \in S\}$$

$$\mathsf{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \mathsf{index}_w(h, \ell) \in S\} \qquad \forall h \in [2^{\lfloor w/2 \rfloor}]$$

note that $S = \bigcup_{h \in [2^{\lfloor w/2 \rfloor}]} \{\mathsf{index}_w(h, \ell) \mid \ell \in \mathsf{clusters}[h]\}$.

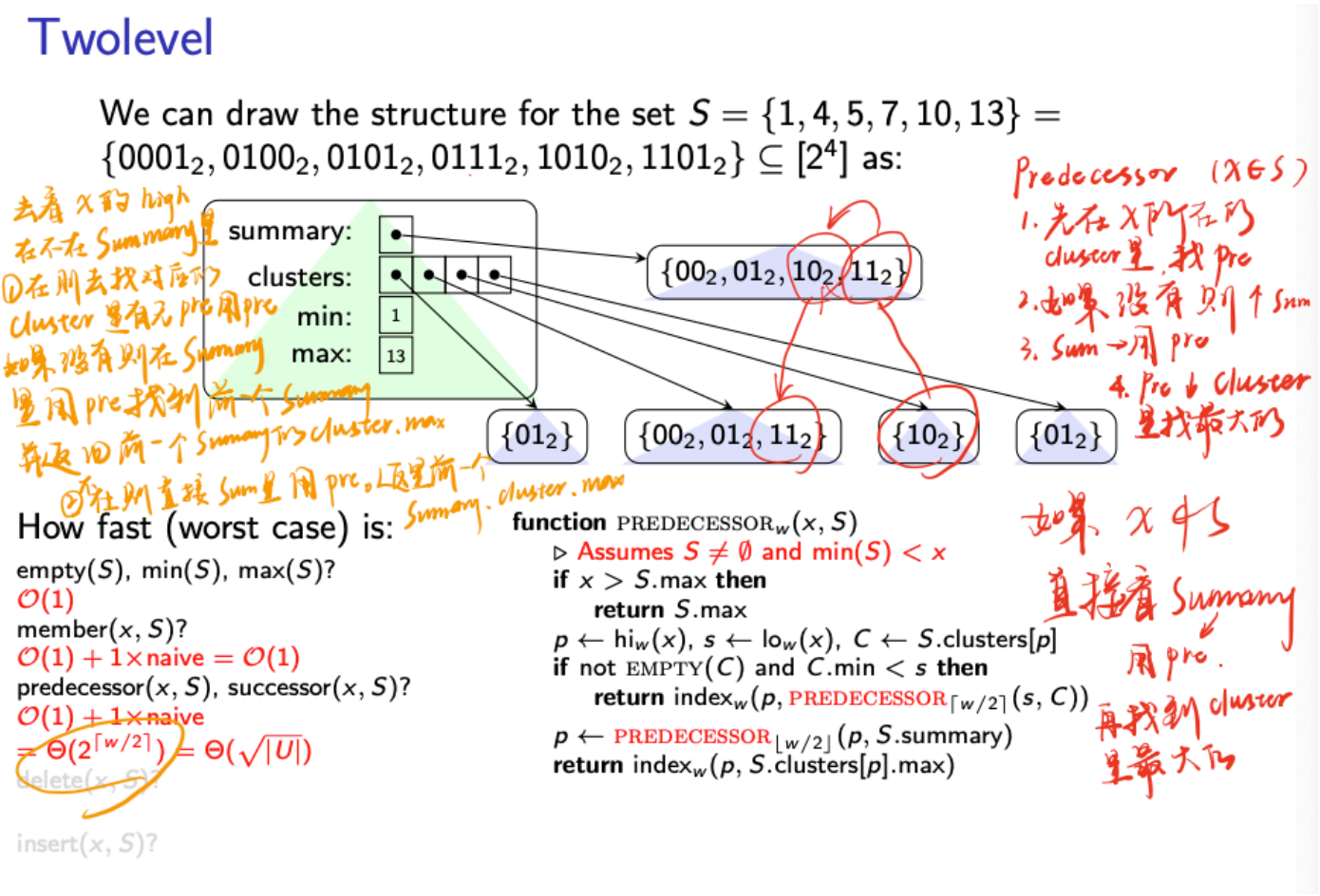
此时的高位是 Summary，低位是 Cluster 且每一个 Cluster 接收其对应的高位作为参数，如果对应的 h 没有的话，比如没有1101，则summary里只剩下$00, 01, 10$，则对应的cluster则存空集。

The worst case of the running time

`empty(S)`, `min(S)`, `max(S)` => $O(1)$

`member(x, S)` => $O(1) + 1 \times naive = O(1)$. 根据 h 找到对应的 $cluster$，在 $cluster$ 里调用 $member$ 找（$naive$）。

`predecessor(x, S)`, `successor(x, S)` => $O(1) + 1 \times naive = \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$.



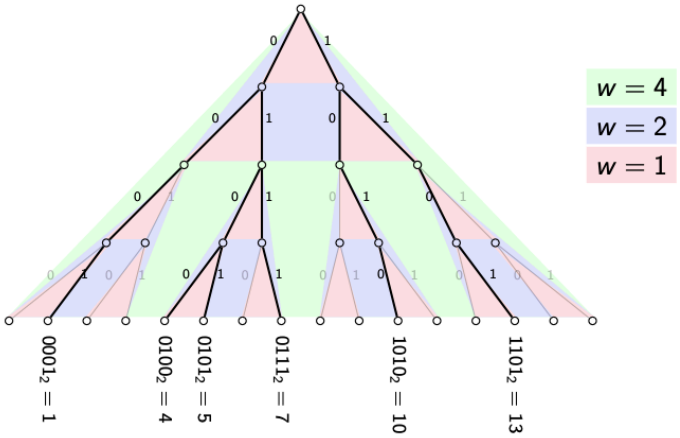
最差的情况一定会调用一次predecessor，而且predecessor是用在$[w/2]$上的。

delete(x, S) => $O(1) + 2 \times naive = \Theta(\sqrt{|U|})$. 因为得删两个 substructure 里的数据，所以是 $2 \times naive = 2 \times \Theta(2^{\lceil w/2 \rceil})$。

insert(x, S) => $O(1) + 2 \times naive = O(1)$. 同样是要插入两次，所以也是$2 \times naive$。

- ***Recursive***

Instead of using the naive structure for summary and clusters[h], recursively use the same type of structure (stop recursion when w = 1).

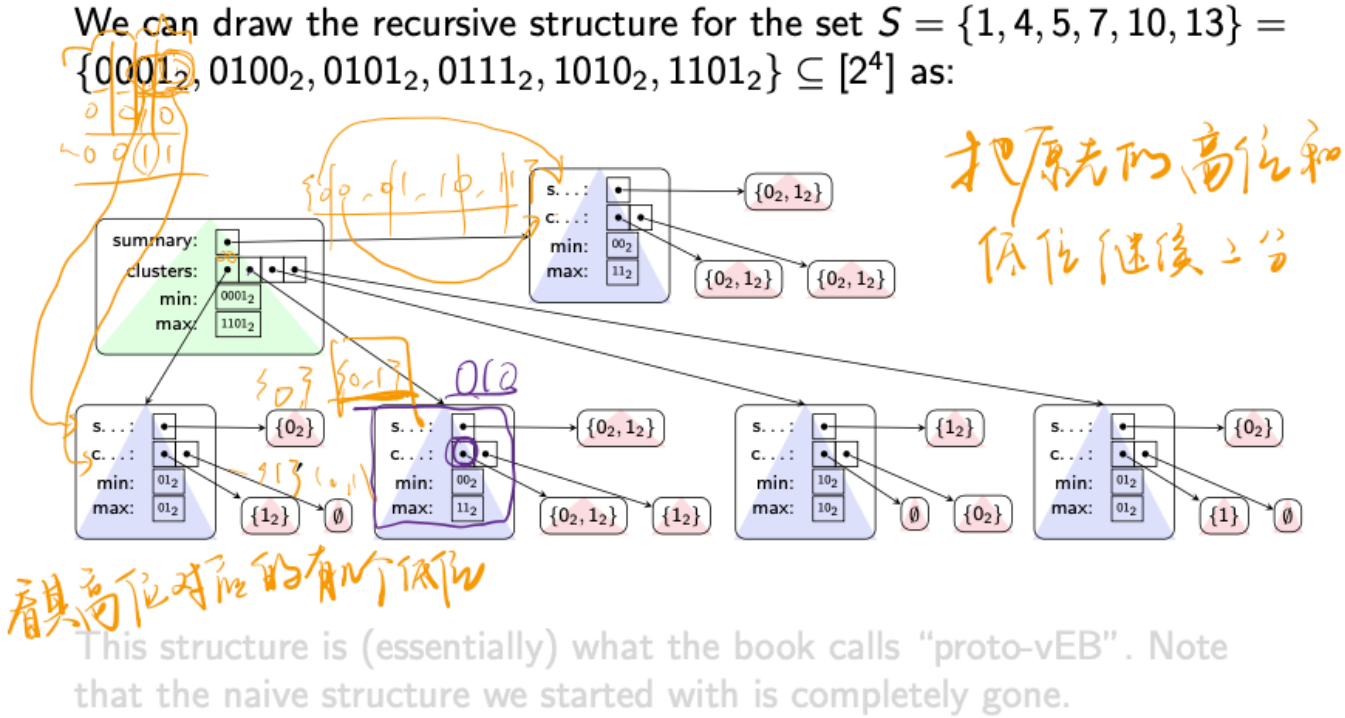


From my perspective, we recursively divide the original part, including the original high part and low part, by 2 until the number of the bit equals to 1. In the original case, the summary has converted into a substructure with high part and low part.

Recursive

Idea: Instead of using the naive structure for summary and clusters[h], recursively use the same type of structure (stop recursion when $w = 1$).

We can draw the recursive structure for the set $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$ as:



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

And this new structure is called ***"proto-vEB"***.

- ***Recursive - Theorem => Theorem 1***

The recursion depth of this structure, when used on the universe $U = [2^w]$ ***is*** $\lceil \log_2 w \rceil = O(\log \log |U|)$.

Recursive

Theorem

The recursion depth of this structure, when used on the universe $U = [2^w]$ *is* $\lceil \log_2 w \rceil = \mathcal{O}(\log \log |U|)$.

Proof.

Let $d(w)$ be the recursion depth when working with a universe of size 2^w. We will prove by induction that $d(w) = \lceil \log_2 w \rceil$.
The base case is $w = 1$ where there is no recursion, so $d(1) = 0 = \lceil \log_2(1) \rceil$.
For the induction case, suppose $w > 1$ and that $d(w') = \lceil \log_2(w') \rceil$ for all $w' \in [w]_+$. Now let $w' = \lceil w/2 \rceil \in [w]_+$, then the largest universe size used in the recursion is $2^{w'}$, and
$d(w) = 1 + d(w') = 1 + \lceil \log_2(w') \rceil = \lceil \log_2(2w') \rceil$.
If w is even, $2w' = w$ and we are done.
Otherwise w is odd and $w \geq 3$ so the smallest integer $k = \lceil \log_2(w) \rceil$ such that $2^k \geq w$ must also be the smallest integer $k = \lceil \log_2(2w') \rceil$ such that $2^k \geq 2w' = w + 1$ and therefore $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

Running Time

`empty(S)`, `min(S)` and `max(S)` => $O(1)$.

`member(x, S)` => $O(1) + 1 \times recursion = O(d(w)) = O(\log \log |U|)$. 如果该数存在，那么一定要遍历整个结构找到最底层所存的数。

`predecessor(x, S)`, and `successor(x, S)` =>
$O(1) + 1 \times recursion = O(d(w)) = O(\log \log |U|)$.

`insert(x, S)` and `delete(x, S)` =>
$O(1) + 2 \times recursion = \Theta(2^{d(w)}) = \Theta(w) = \Theta(\log |U|)$.

If w is even, the depth of the summary and the depth of each cluster is the same.

对于这两个方法，一次迭代是为了在cluster的最底层中做更新，另一次迭代是为了在summary里更新min和max。

- ***vEB: worst case*** $O(\log \log |U|)$ ***time***

vEB: worst case $\mathcal{O}(\log\log|U|)$ time

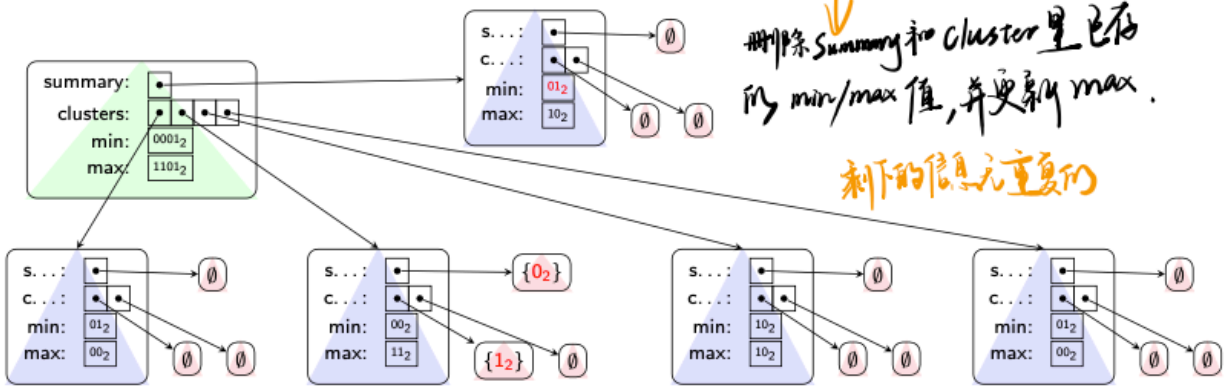
Idea: Exclude $\min(S)$ and/or $\max(S)$ from the set of keys stored in summary and clusters. (CLRS excludes only $\min(S)$, I exclude both).

Specifically, redefine summary and clusters to recursively store the sets:

$$\text{summary} := \{\text{hi}_w(x) \mid w \in S \setminus \{\min, \max\}\}$$

$$\text{clusters}[h] := \{\ell \in [2^{\lceil w/2 \rceil}] \mid \text{index}_w(h, \ell) \in S \setminus \{\min, \max\}\} \quad \forall h \in [2^{\lfloor w/2 \rfloor}]$$

We can draw the van Emde Boas tree for the set $S = \{1, 4, 5, 7, 10, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, 1010_2, 1101_2\} \subseteq [2^4]$ as:



The basic idea is that each substructure do not have the redundent data. *For example, 0001 as the min in the set, the substructure of the summary has already stores the minimum 0001, therefore, there is no need to store this value in the other substructures. The reason that the first cluster's substructure is empty (max > min) is because the corresponding summary 00 related value 0001 already stored in the top level structure.*

Running Time

`PREDECESSORw(x, S)`

vEB: worst case $\mathcal{O}(\log\log|U|)$ time, $\mathcal{O}(|U|)$ space

```
1:  function PREDECESSOR_w(x, S)        ▷ Assumes S ≠ ∅ and x > S.min
2:      if x > S.max then
3:          return S.max
4:      if w = 1 then
5:          return S.min
6:      p ← hi_w(x), s ← lo_w(x), C ← S.clusters[p]
7:      if not EMPTY(C) and C.min < s then
8:          return index_w(p, PREDECESSOR_⌈w/2⌉(s, C))
9:      if EMPTY(S.summary) or p ≤ S.summary.min then
10:         return S.min
11:     p ← PREDECESSOR_⌊w/2⌋(p, S.summary)
12:     return index_w(p, S.clusters[p].max)
```

Theorem

$\text{PREDECESSOR}_w(x, S)$ *takes worst case* $\mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$ *time.*

Proof.

It makes at most one recursive call. □

1. First, exclude the case that $x > S.max$.
2. Then, exclude the case that $w = 1$.
3. Calculate the high part and low part. Otherwise, we are in a case where we may have to do recursion.

   After line 7, $S.min < x \leq S.max$.
4. If the corresponding cluster is not empty and the cluster.min < s, then return the value by doing a recursive call to the `PREDECESSOR`.
5. If the summary is empty or $p \leq S.summary$ which means this value is the second minimum in the set. Return $S.min$.
6. Otherwise, call the `PREDECESSOR(p, S.summary)` to find the previous p and return the corresponding cluster's maximum.

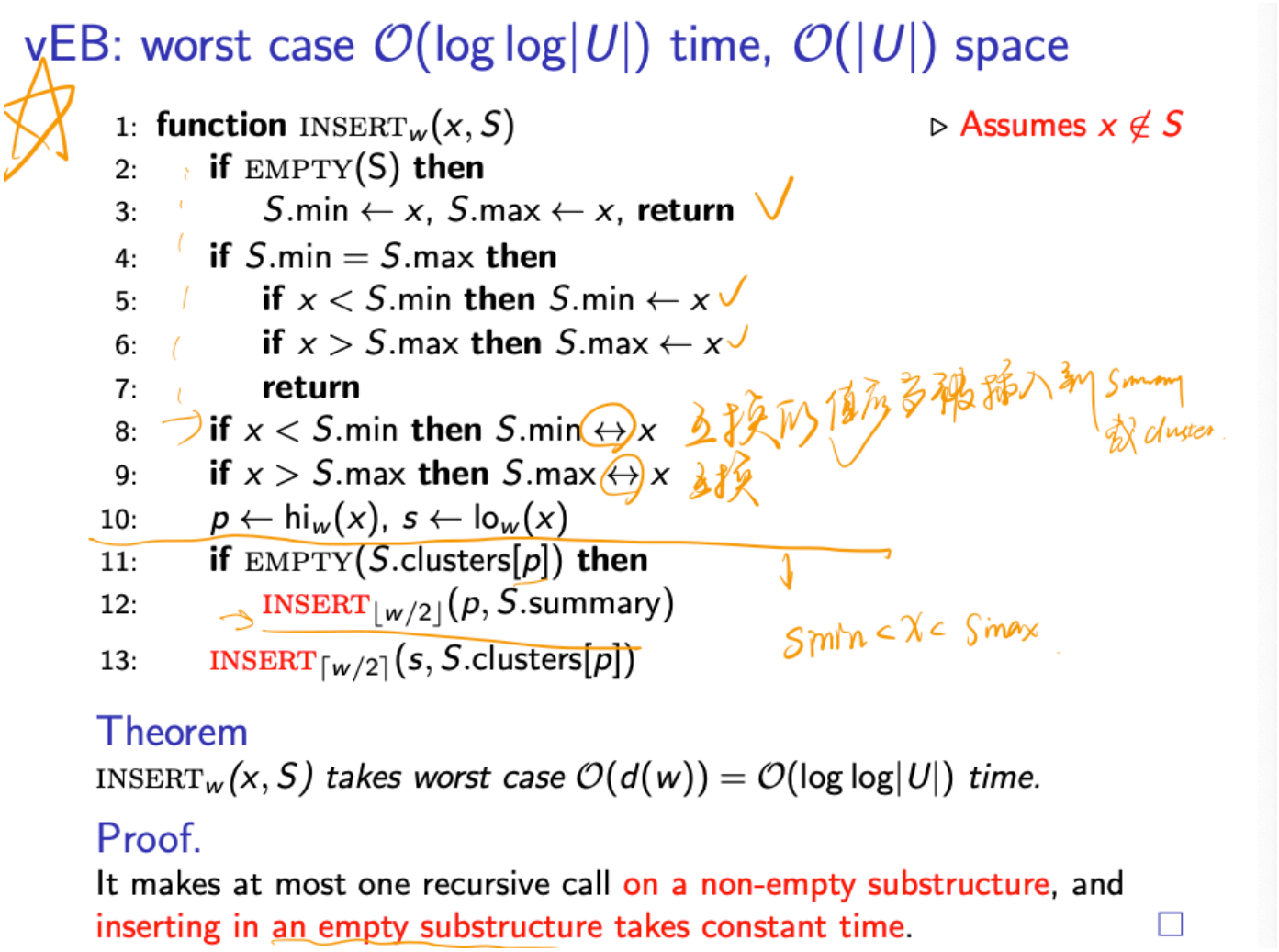
Theorem - 2

`PREDECESSORw(x,S)` takes worst case $O(d(w)) = O(\log\log|U|)$ time.

Proof

By observing the pseudocode, we know that the recursion depth is a $d(w)$ which is $\log\log|U|$ and this function does at most one recursive call and everything else takes constant time.

`INSERTw(x, S)`



- ***RS-vEB***

RS-vEB: expected $\mathcal{O}(\log\log|U|)$ time, $\mathcal{O}(n\log\log|U|)$ space

Idea: Use a hash table instead of an array to store clusters[·], and don't store empty substructures.

Why does this change updates from worst case $\mathcal{O}(\log\log|U|)$ to expected $\mathcal{O}(\log\log|U|)$ time? Because updates to a hash table take expected $\mathcal{O}(1)$ time rather than worst case.

Why does this only use $\mathcal{O}(n \cdot d(w)) = \mathcal{O}(n\log\log|U|)$ space? Because the empty structure uses $\mathcal{O}(1)$ space and INSERT_w only creates or updates $\mathcal{O}(d(w))$ substructures in the worst case. Each of these costs at most an additional $\mathcal{O}(1)$ space

- ***R^2S-vEB***

R^2S-vEB: expected $\mathcal{O}(\log\log|U|)$ time, $\mathcal{O}(n)$ space

Idea: Partition S into "chunks" of size $\Theta(\min\{n, \log\log|U|\})$, and store only one element representing each chunk in the RS-vEB structure. I.e. RS-vEB stores only $\mathcal{O}(\max\{1, n/\log\log|U|\})$ elements, using $\mathcal{O}(n)$ space.

We can store each chunk as a sorted linked list, and keep a hash table mapping each representative to its chunk. This also takes $\mathcal{O}(n)$ space.

PREDECESSOR and SUCCESSOR uses the RS-vEB structure to find the nearest two representatives in $\mathcal{O}(\log\log|U|)$ time, and can then spend linear time in the size of the two chunks to find the result.

INSERT may have to split a chunk that becomes too large and insert a new representative in the RS-vEB structure. Splitting the chunk can take linear time in the size of the chunk, and together with inserting the new representative into the RS-vEB structure, this still only takes expected $\mathcal{O}(\log\log|U|)$ time.

Similarly, DELETE may have to join two chunks and delete a representative, but again this only takes expected $\mathcal{O}(\log\log|U|)$ time.