# Advanced algorithms and data structures

## Lecture 6: van Emde Boas Trees

Jacob Holm (jaho@di.ku.dk)

December 7th 2022

# Today's Lecture

## van Emde Boas Trees

- Predecessor search/ordered sets
- Naive
- Twolevel
- Recursive
- vEB: worst case $\mathcal{O}(\log\log|U|)$ time
- RS-vEB: expected $\mathcal{O}(\log\log|U|)$ time, $\mathcal{O}(n\log\log|U|)$ space
- $R^2$S-vEB: expected $\mathcal{O}(\log\log|U|)$ time, $\mathcal{O}(n)$ space
- Bonus: vEB is optimal for $w = \Theta(\log n)$
- Bonus: Integer sorting in expected $\mathcal{O}(n\log\log|U|)$ time.

# Predecessor search/ordered sets

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

member$(x, S)$: Return $[x \in S]$.

insert$(x, S)$: Add $x$ to $S$ (assumes $x \notin S$).

delete$(x, S)$: Remove $x$ from $S$ (assumes $x \in S$).

empty$(S)$: Return $[S = \emptyset]$.

min$(S)$: Return $\min S$ (assumes $S \neq \emptyset$).

max$(S)$: Return $\max S$ (assumes $S \neq \emptyset$).

predecessor$(x, S)$: Return $\max\{y \in S \mid y < x\}$
(assumes $\{y \in S \mid y < x\}$ is nonempty,
i.e. $S \neq \emptyset$ and $x > \min(S)$).

successor$(x, S)$: Return $\min\{y \in S \mid y > x\}$
(assumes $\{y \in S \mid y > x\}$ is nonempty,
i.e. $S \neq \emptyset$ and $x < \max(S)$).

# Predecessor search/ordered sets

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

member$(x, S)$: Return $[x \in S]$.

insert$(x, S)$: Add $x$ to $S$ (assumes $x \notin S$).

delete$(x, S)$: Remove $x$ from $S$ (assumes $x \in S$).

empty$(S)$: Return $[S = \emptyset]$.

min$(S)$: Return $\min S$ (assumes $S \neq \emptyset$).

max$(S)$: Return $\max S$ (assumes $S \neq \emptyset$).

predecessor$(x, S)$: Return $\max\{y \in S \mid y < x\}$ (assumes $\{y \in S \mid y < x\}$ is nonempty, i.e. $S \neq \emptyset$ and $x > \min(S)$).

successor$(x, S)$: Return $\min\{y \in S \mid y > x\}$ (assumes $\{y \in S \mid y > x\}$ is nonempty, i.e. $S \neq \emptyset$ and $x < \max(S)$).

# Predecessor search/ordered sets

**Problem:**

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

member$(x, S)$: Return $[x \in S]$.

insert$(x, S)$: Add $x$ to $S$ (assumes $x \notin S$).

delete$(x, S)$: Remove $x$ from $S$ (assumes $x \in S$).

empty$(S)$: Return $[S = \emptyset]$.

min$(S)$: Return $\min S$ (assumes $S \neq \emptyset$).

max$(S)$: Return $\max S$ (assumes $S \neq \emptyset$).

predecessor$(x, S)$: Return $\max\{y \in S \mid y < x\}$
(assumes $\{y \in S \mid y < x\}$ is nonempty,
i.e. $S \neq \emptyset$ and $x > \min(S)$).

successor$(x, S)$: Return $\min\{y \in S \mid y > x\}$
(assumes $\{y \in S \mid y > x\}$ is nonempty,
i.e. $S \neq \emptyset$ and $x < \max(S)$).

# 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($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



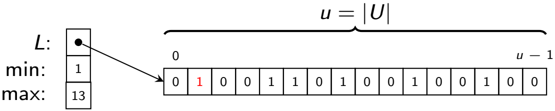

$\{1, 4, 5, 7, 10, 13\}$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



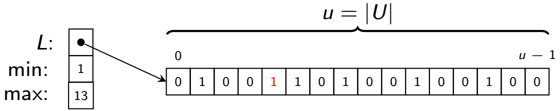

$$\{1, 4, 5, 7, 10, 13\}$$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



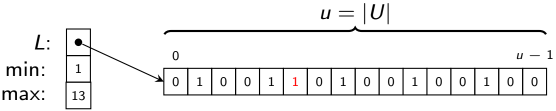

$$\{1, 4, 5, 7, 10, 13\}$$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



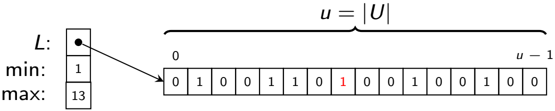

$\{1, 4, 5, 7, 10, 13\}$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



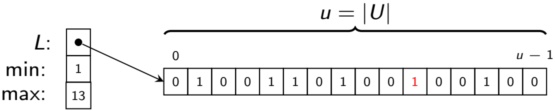

$\{1, 4, 5, 7, 10, 13\}$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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($S$)?

member($x, S$)?

predecessor($x, S$) and successor($x, S$)?

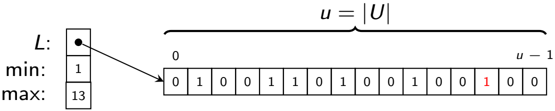
delete($x, S$)?

insert($x, S$)?

# 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.



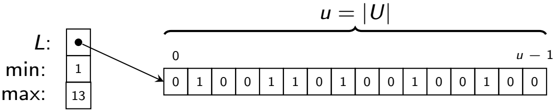

$\{1, 4, 5, 7, 10, 13\}$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



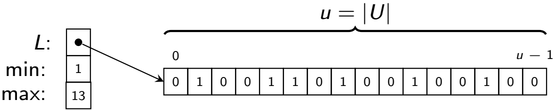

$$\{1, 4, 5, 7, 10, 13\}$$

How fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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($x, S$)?

predecessor($x, S$) and successor($x, S$)?

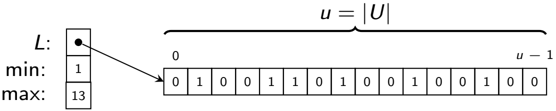
delete($x, S$)?

insert($x, S$)?

# 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($x, S$)?

predecessor($x, S$) and successor($x, S$)?

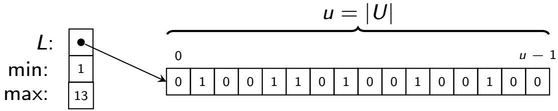
delete($x, S$)?

insert($x, S$)?

# 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.



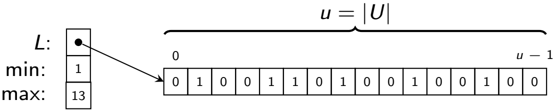

$\{1, 4, 5, 7, 10, 13\}$

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($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



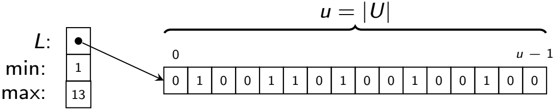

$$\{1, 4, 5, 7, 10, 13\}$$

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($x, S$)?
delete($x, S$)?
insert($x, S$)?

# 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.



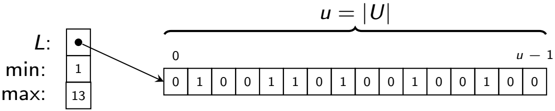

$\{1, 4, 5, 7, 10, 13\}$

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($x, S$)?
insert($x, S$)?

# 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.



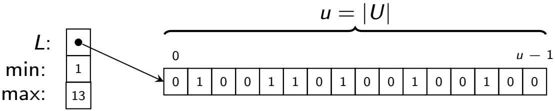

$\{1, 4, 5, 7, 10, 13\}$

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($x, S$)?
insert($x, S$)?

# 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.



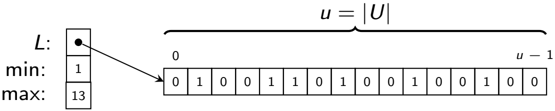

$$\{1, 4, 5, 7, 10, 13\}$$

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($x, S$)?

# 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.



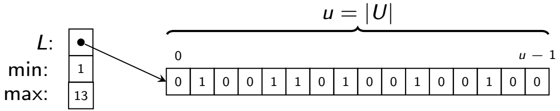

$$\{1, 4, 5, 7, 10, 13\}$$

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($x, S$)?

# 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.

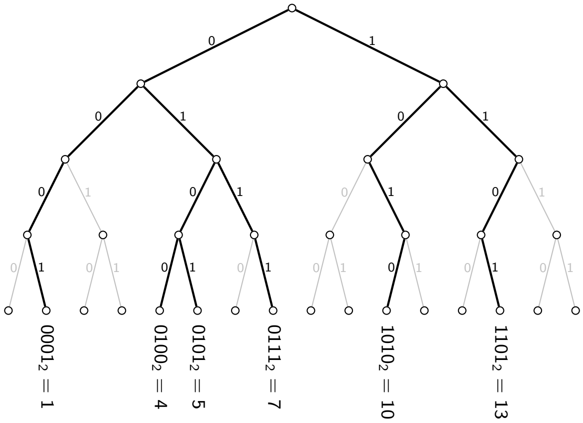



$\{1, 4, 5, 7, 10, 13\}$

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 $\mathcal{O}(1)$.

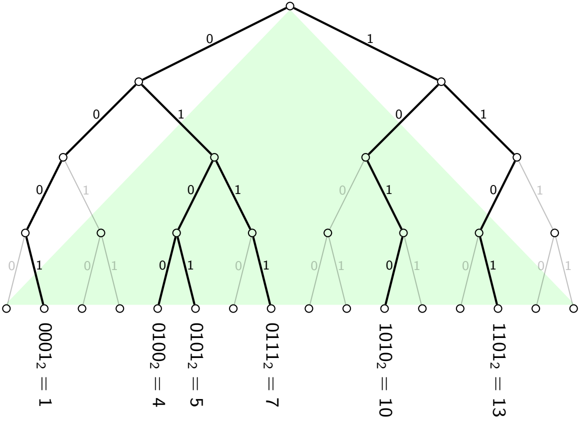
# Bit-Trie

**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?



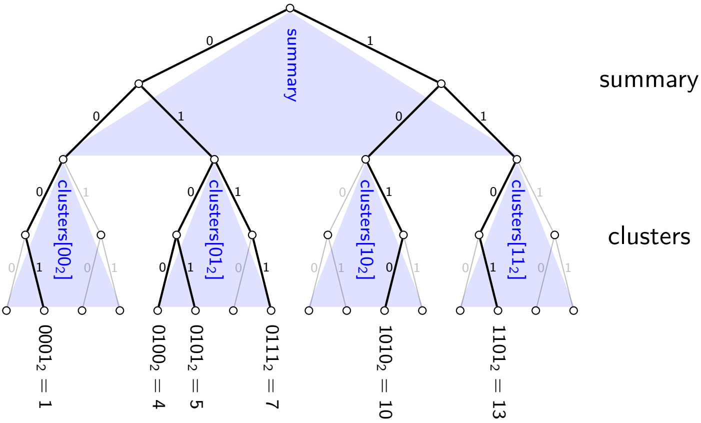
# Bit-Trie

**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?



# 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

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

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

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

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

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




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

Now let the structure directly store the values:

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

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

and use the naive structure to store

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

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

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

# Twolevel

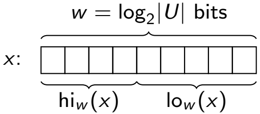
**Idea:** Split each key into *high* and *low* parts. Use naive for each.

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

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

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

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




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

Now let the structure directly store the values:

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

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

and use the naive structure to store

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

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

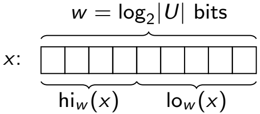
note that $S = \bigcup_{h \in [2^{\lfloor w/2 \rfloor}]} \{\mathrm{index}_w(h, \ell) \mid \ell \in \mathrm{clusters}[h]\}$.

# Twolevel

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

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

$$\mathrm{hi}_w(x) := \left\lfloor \frac{x}{2^{\lceil w/2 \rceil}} \right\rfloor$$
$$\mathrm{lo}_w(x) := x \bmod 2^{\lceil w/2 \rceil}$$
$$\mathrm{index}_w(h, \ell) := h \cdot 2^{\lceil w/2 \rceil} + \ell$$




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

Now let the structure directly store the values:

$$\min := \min(S) \text{ if } S \neq \emptyset, \text{ else } 1$$
$$\max := \max(S) \text{ if } S \neq \emptyset, \text{ else } 0$$

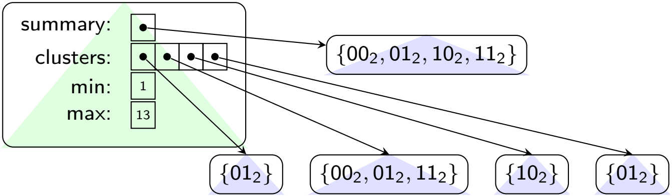
and use the naive structure to store

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

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

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

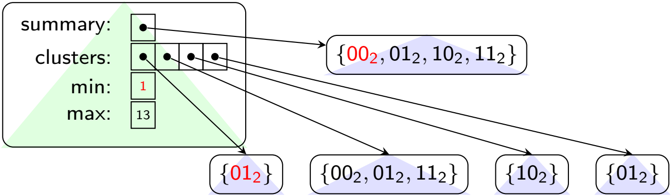
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

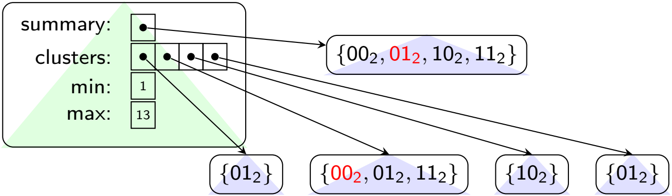
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

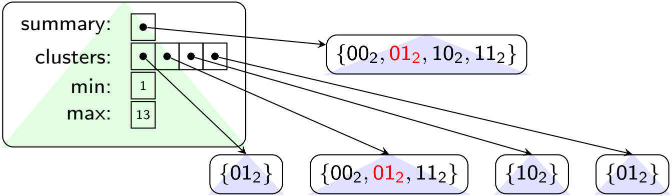
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

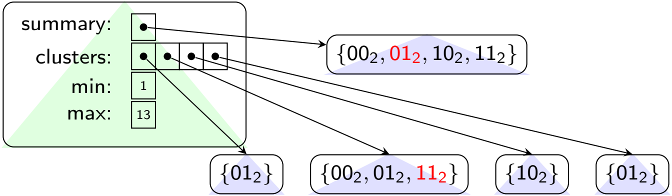
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

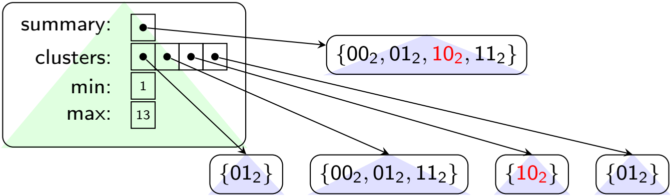
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

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




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

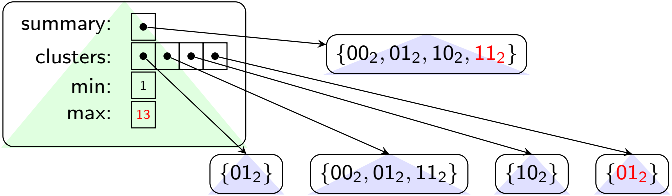
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

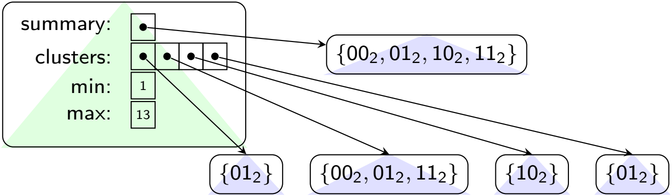
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

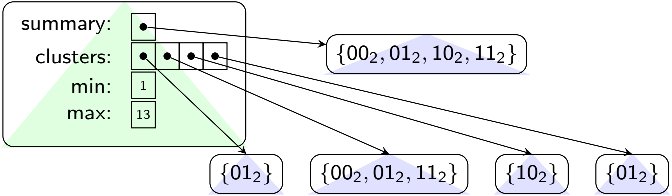
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$

member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

delete($x$, $S$)?

insert($x$, $S$)?

**function** EMPTY($S$)
  **return** $S$.min $>$ $S$.max

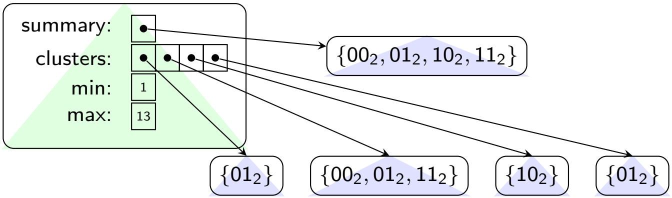
**function** MIN($S$)
  ▷ Assumes $S \neq \emptyset$
  **return** $S$.min

**function** MAX($S$)
  ▷ Assumes $S \neq \emptyset$
  **return** $S$.max

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?

predecessor($x$, $S$), successor($x$, $S$)?

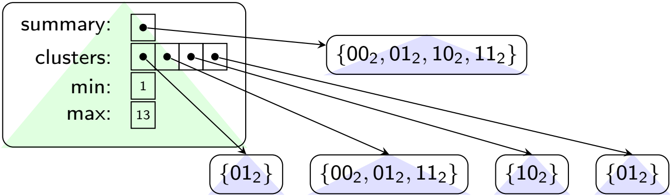
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?

delete($x$, $S$)?

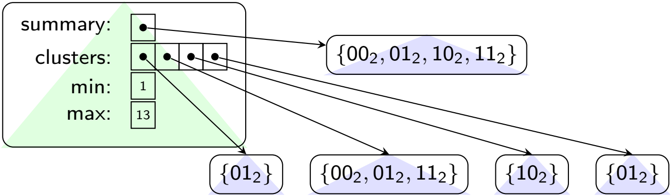
insert($x$, $S$)?

**function** $\text{MEMBER}_w(x, S)$
**return** $\text{MEMBER}_{\lceil w/2 \rceil}(\text{lo}_w(x), S.\text{clusters}[\text{hi}_w(x)])$

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times\text{naive} = \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?

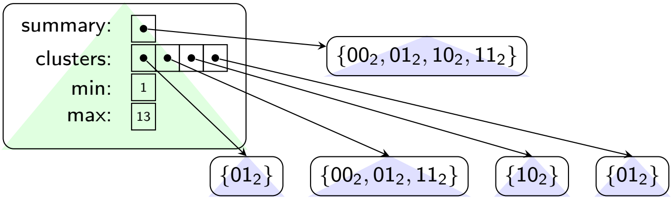
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x, S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x, S$), successor($x, S$)?
$\mathcal{O}(1) + 1\times$naive
$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$
delete($x, S$)?

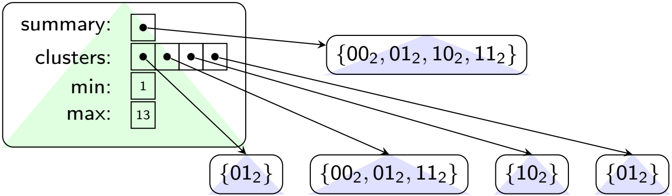
insert($x, S$)?

**function** $\text{PREDECESSOR}_w(x, S)$
  ▷ Assumes $S \neq \emptyset$ and $\min(S) < x$
  **if** $x > S$.max **then**
    **return** $S$.max
  $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$, $C \leftarrow S$.clusters[$p$]
  **if** not $\text{EMPTY}(C)$ and $C$.min $< s$ **then**
    **return** $\text{index}_w(p, \text{PREDECESSOR}_{\lceil w/2 \rceil}(s, C))$
  $p \leftarrow \text{PREDECESSOR}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
  **return** $\text{index}_w(p, S.\text{clusters}[p].\text{max})$

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

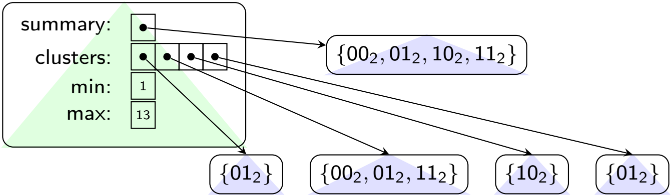
empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive
$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$
delete($x$, $S$)?

insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

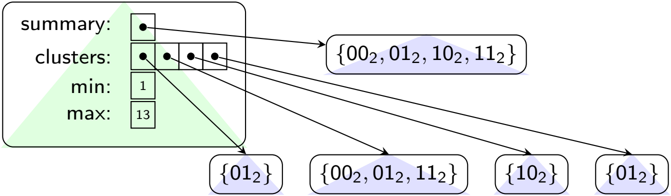
empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive
$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$
delete($x$, $S$)?
$\mathcal{O}(1) + 2\times$naive $= \Theta(\sqrt{|U|})$
insert($x$, $S$)?

**function** $\text{DELETE}_w(x, S)$
  ▷ Assumes $x \in S$
  $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$, $C \leftarrow S.\text{clusters}[p]$
  $\text{DELETE}_{\lceil w/2 \rceil}(s, C)$
  **if** EMPTY(C) **then**
    $\text{DELETE}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
  **if** $S.\text{min} = S.\text{max}$ **then**
    $S.\text{min} \leftarrow 1$, $S.\text{max} \leftarrow 0$
  **else if** $x = S.\text{min}$ **then**
    $p \leftarrow S.\text{summary.min}$, $S.\text{min} \leftarrow \text{index}_w(p, S.\textit{clusters}[p].\text{min})$
  **else if** $x = S.\text{max}$ **then**
    $p \leftarrow S.\text{summary.max}$, $S.\text{max} \leftarrow \text{index}_w(p, S.\textit{clusters}[p].\text{max})$

How can we improve the time further?

# Twolevel

We can draw the 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:



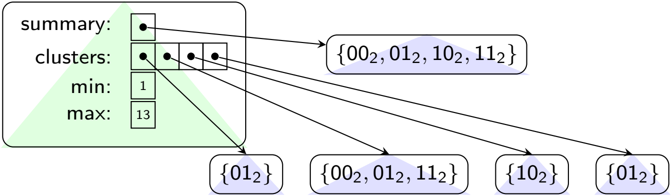

How fast (worst case) is:

empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive
$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$
delete($x$, $S$)?
$\mathcal{O}(1) + 2\times$naive $= \Theta(\sqrt{|U|})$
insert($x$, $S$)?

How can we improve the time further?

# Twolevel

We can draw the 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:




How fast (worst case) is:

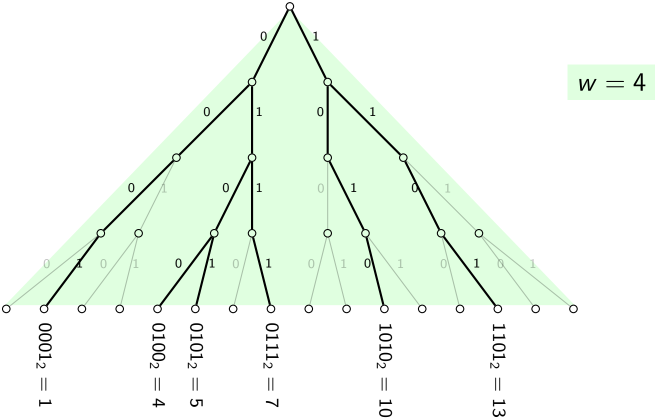
empty($S$), min($S$), max($S$)?
$\mathcal{O}(1)$
member($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive $= \mathcal{O}(1)$
predecessor($x$, $S$), successor($x$, $S$)?
$\mathcal{O}(1) + 1\times$naive
$= \Theta(2^{\lceil w/2 \rceil}) = \Theta(\sqrt{|U|})$
delete($x$, $S$)?
$\mathcal{O}(1) + 2\times$naive $= \Theta(\sqrt{|U|})$
insert($x$, $S$)?
$\mathcal{O}(1) + 2\times$naive $= \mathcal{O}(1)$

**function** $\text{INSERT}_w(x, S)$
    ▷ Assumes $x \notin S$
    **if** $\text{EMPTY}(S)$ **then**
        $S.\text{min} \leftarrow x$, $S.\text{max} \leftarrow x$
    **if** $x < S.\text{min}$ **then** $S.\text{min} \leftarrow x$
    **if** $x > S.\text{max}$ **then** $S.\text{max} \leftarrow x$
    $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$
    **if** $\text{EMPTY}(S.\text{clusters}[p])$ **then**
        $\text{INSERT}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
    $\text{INSERT}_{\lceil w/2 \rceil}(s, S.\text{clusters}[p])$

How can we improve the time further?

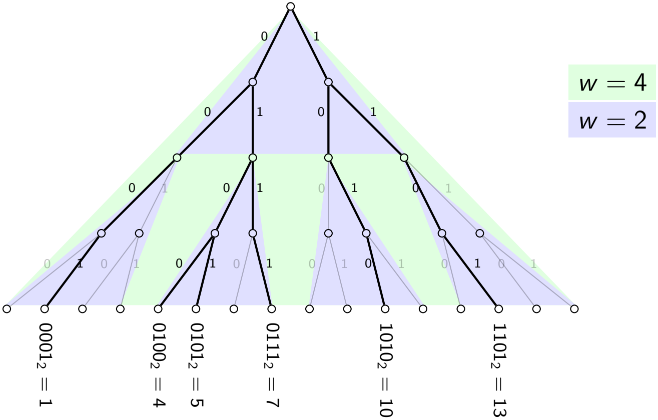
# Recursive

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



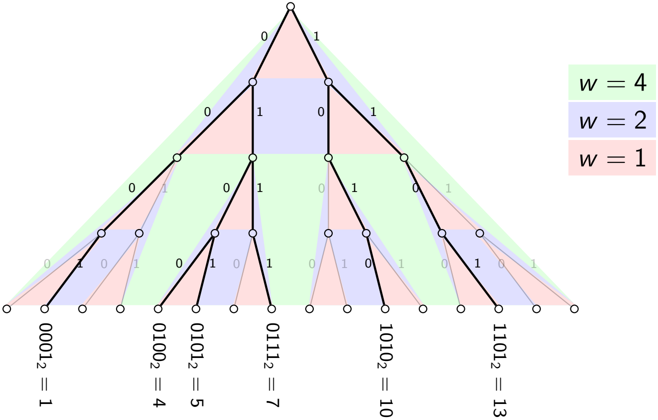
# Recursive

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



# Recursive

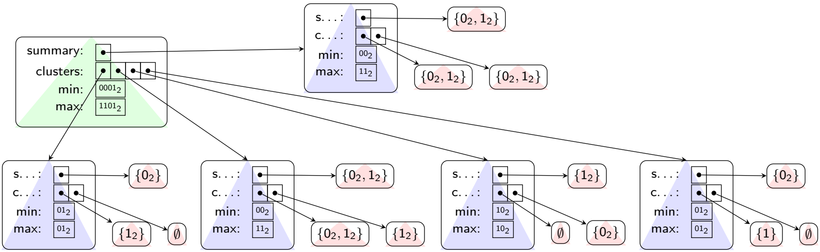
**Idea:** Instead of using the naive structure for summary and clusters[$h$], recursively use the same type of structure (stop recursion when $w = 1$).



# 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:

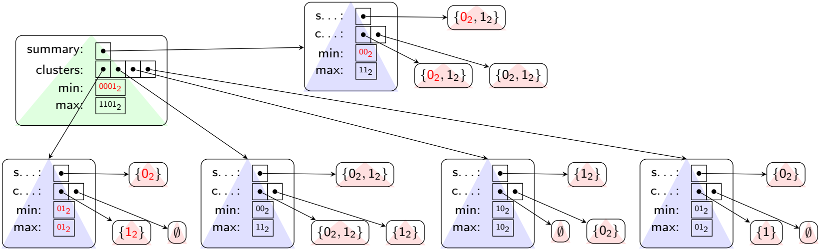



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

# 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:

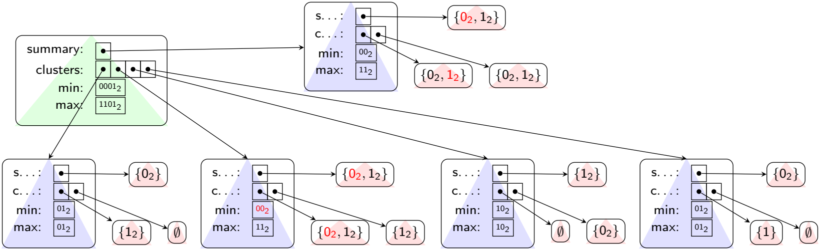



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

# 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:

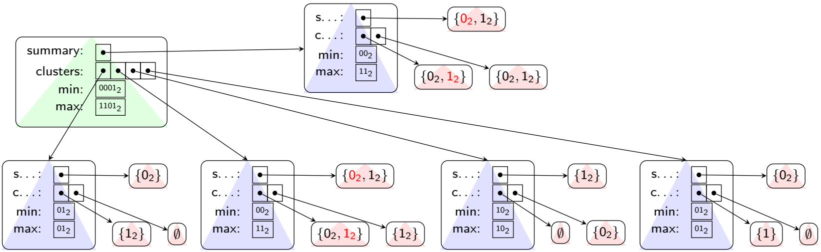



This structure is (essentially) what the book calls “proto-vEB”. Note that the naive structure we started with is completely gone.

# 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:

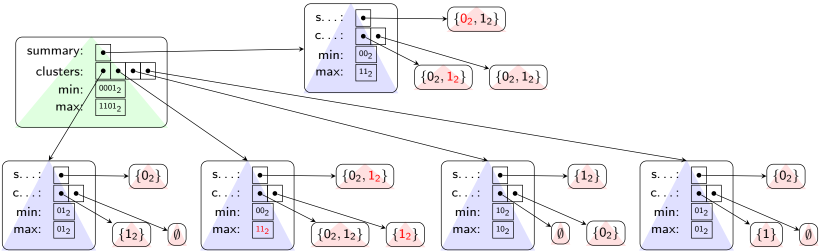



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

# 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:

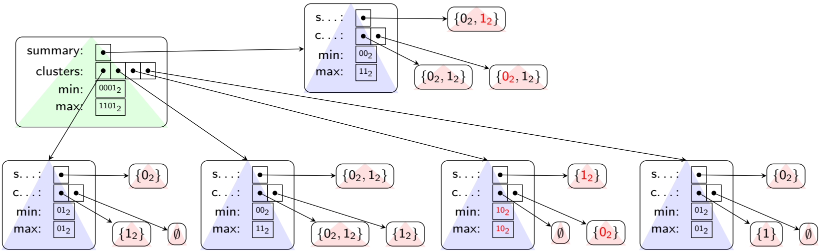



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

# 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:

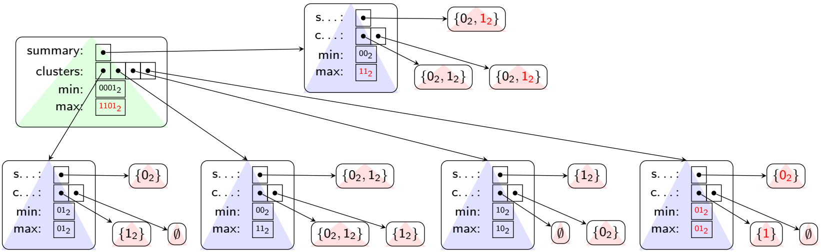



This structure is (essentially) what the book calls “proto-vEB”. Note that the naive structure we started with is completely gone.

# 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:

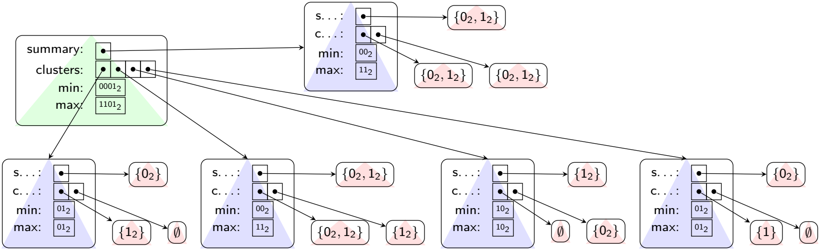



This structure is (essentially) what the book calls "proto-vEB". Note that the naive structure we started with is completely gone.

# 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.

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. □

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# 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 $\lceil \log_2(2w') \rceil = \lceil \log_2(w) \rceil$. $\square$

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)?
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)?
predecessor($x, S$) and successor($x, S$)?

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1 \times \text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log \log |U|)$.
predecessor($x, S$) and successor($x, S$)?

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1\times\text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
predecessor($x, S$) and successor($x, S$)?

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1\times\text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
predecessor($x, S$) and successor($x, S$)? $O(1) + 1\times\text{recursion}$
$= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.

insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1\times$recursion $= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
predecessor($x, S$) and successor($x, S$)? $O(1) + 1\times$recursion
$= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
insert($x, S$) and delete($x, S$)?

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:

empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.

member($x, S$)? $\mathcal{O}(1) + 1\times\text{recursion} = \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.

predecessor($x, S$) and successor($x, S$)? $O(1) + 1\times\text{recursion}$
$= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.

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

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1\times$recursion $= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
predecessor($x, S$) and successor($x, S$)? $O(1) + 1\times$recursion
$= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
insert($x, S$) and delete($x, S$)? $\mathcal{O}(1) + 2\times$recursion
$= \Theta(2^{d(w)}) = \Theta(w) = \Theta(\log|U|)$.

How can we improve the update time?

# Recursive

Using the recursive structure, how fast is:
empty($S$), min($S$), and max($S$)? worst case $\mathcal{O}(1)$.
member($x, S$)? $\mathcal{O}(1) + 1\times$recursion $= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
predecessor($x, S$) and successor($x, S$)? $O(1) + 1\times$recursion
$= \mathcal{O}(d(w)) = \mathcal{O}(\log\log|U|)$.
insert($x, S$) and delete($x, S$)? $\mathcal{O}(1) + 2\times$recursion
$= \Theta(2^{d(w)}) = \Theta(w) = \Theta(\log|U|)$.

How can we improve the update time? Somehow make insert and delete recurse in only one substructure.

# 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:

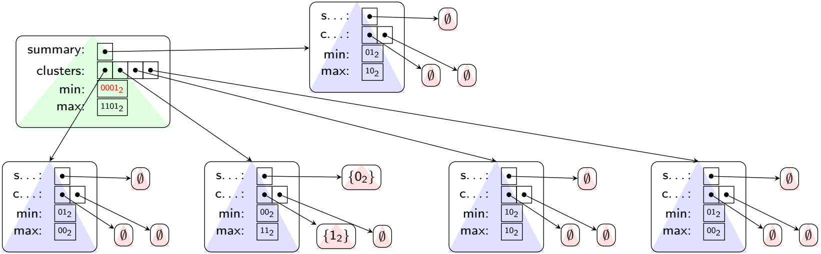
# vEB: worst case $\mathcal{O}(\log\log|U|)$ time

**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:



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

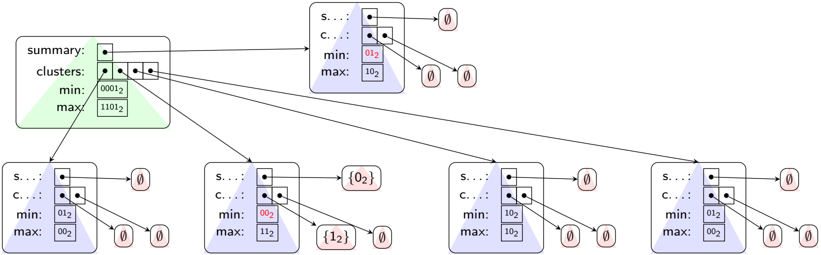
**Idea:** Exclude $\min(S)$ and/or $\max(S)$ from the set of keys stored in summary and clusters. (CLRS excludes only $\min(S)$, I exclude both).

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:



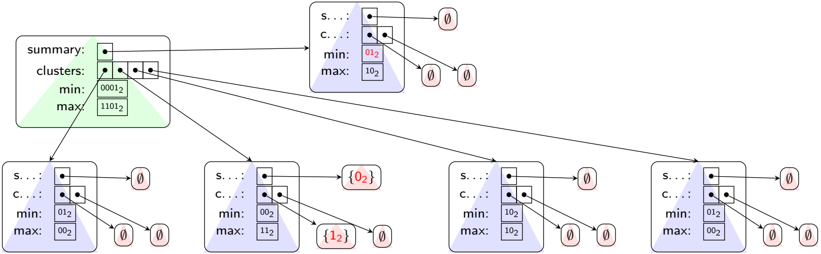
# vEB: worst case $\mathcal{O}(\log\log|U|)$ time

**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:



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

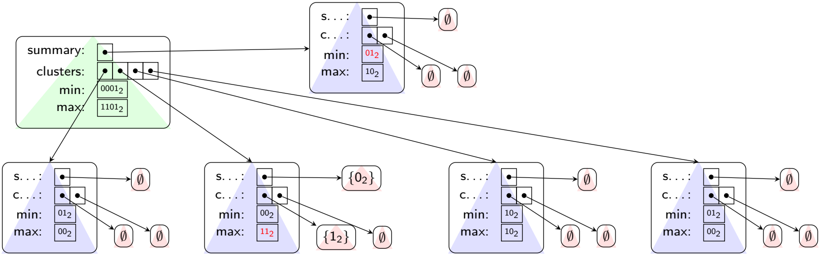
**Idea:** Exclude $\min(S)$ and/or $\max(S)$ from the set of keys stored in summary and clusters. (CLRS excludes only $\min(S)$, I exclude both).

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:



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

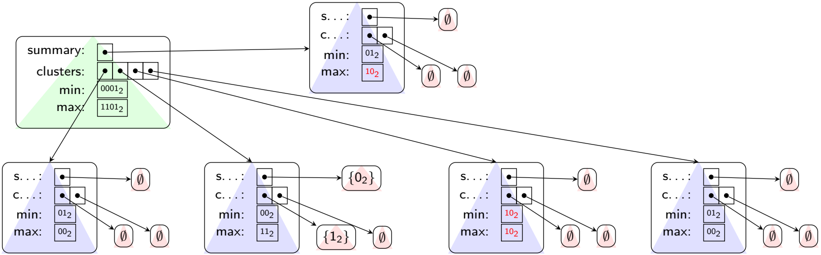
**Idea:** Exclude $\min(S)$ and/or $\max(S)$ from the set of keys stored in summary and clusters. (CLRS excludes only $\min(S)$, I exclude both).

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 $S = \{1, 4, 5, 7, {\color{red}10}, 13\} = \{0001_2, 0100_2, 0101_2, 0111_2, {\color{red}1010_2}, 1101_2\} \subseteq [2^4]$ as:



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

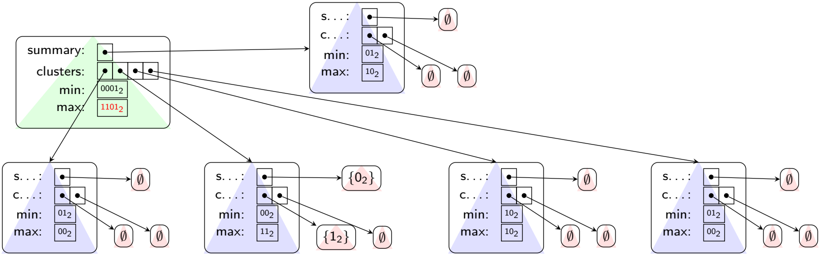
**Idea:** Exclude $\min(S)$ and/or $\max(S)$ from the set of keys stored in summary and clusters. (CLRS excludes only $\min(S)$, I exclude both).

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:



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

**function** $\text{PREDECESSOR}_w(x, S)$ ▷ Assumes $S \neq \emptyset$ and $x > S.\text{min}$
    **if** $x > S.\text{max}$ **then**
        **return** $S.\text{max}$
    **if** $w = 1$ **then**
        **return** $S.\text{min}$
    $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$, $C \leftarrow S.\text{clusters}[p]$
    **if** not $\text{EMPTY}(C)$ and $C.\text{min} < s$ **then**
        **return** $\text{index}_w(p, \text{PREDECESSOR}_{\lceil w/2 \rceil}(s, C))$
    **if** $\text{EMPTY}(S.\text{summary})$ or $p \leq S.\text{summary.min}$ **then**
        **return** $S.\text{min}$
    $p \leftarrow \text{PREDECESSOR}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
    **return** $\text{index}_w(p, S.\text{clusters}[p].\text{max})$

**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. □

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

```
function PREDECESSOR_w(x, S)        ▷ Assumes S ≠ ∅ and x > S.min
    if x > S.max then
        return S.max
    if w = 1 then
        return S.min
    p ← hi_w(x), s ← lo_w(x), C ← S.clusters[p]
    if not EMPTY(C) and C.min < s then
        return index_w(p, PREDECESSOR_⌈w/2⌉(s, C))
    if EMPTY(S.summary) or p ≤ S.summary.min then
        return S.min
    p ← PREDECESSOR_⌊w/2⌋(p, S.summary)
    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. □

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

**function** $\textsc{Insert}_w(x, S)$ ▷ Assumes $x \notin S$
  **if** $\textsc{Empty}(S)$ **then**
    $S.\text{min} \leftarrow x$, $S.\text{max} \leftarrow x$, **return**
  **if** $S.\text{min} = S.\text{max}$ **then**
    **if** $x < S.\text{min}$ **then** $S.\text{min} \leftarrow x$
    **if** $x > S.\text{max}$ **then** $S.\text{max} \leftarrow x$
    **return**
  **if** $x < S.\text{min}$ **then** $S.\text{min} \leftrightarrow x$
  **if** $x > S.\text{max}$ **then** $S.\text{max} \leftrightarrow x$
  $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$
  **if** $\textsc{Empty}(S.\text{clusters}[p])$ **then**
    $\textsc{Insert}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
  $\textsc{Insert}_{\lceil w/2 \rceil}(s, S.\text{clusters}[p])$

## Theorem

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

## Proof.

It makes at most one recursive call on a non-empty substructure, and inserting in an empty substructure takes constant time. □

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

**function** $\text{INSERT}_w(x, S)$ ▷ Assumes $x \notin S$
  **if** $\text{EMPTY}(S)$ **then**
    $S.\text{min} \leftarrow x$, $S.\text{max} \leftarrow x$, **return**
  **if** $S.\text{min} = S.\text{max}$ **then**
    **if** $x < S.\text{min}$ **then** $S.\text{min} \leftarrow x$
    **if** $x > S.\text{max}$ **then** $S.\text{max} \leftarrow x$
    **return**
  **if** $x < S.\text{min}$ **then** $S.\text{min} \leftrightarrow x$
  **if** $x > S.\text{max}$ **then** $S.\text{max} \leftrightarrow x$
  $p \leftarrow \text{hi}_w(x)$, $s \leftarrow \text{lo}_w(x)$
  **if** $\text{EMPTY}(S.\text{clusters}[p])$ **then**
    $\text{INSERT}_{\lfloor w/2 \rfloor}(p, S.\text{summary})$
  $\text{INSERT}_{\lceil w/2 \rceil}(s, S.\text{clusters}[p])$

## Theorem

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

## Proof.

It makes at most one recursive call on a non-empty substructure, and inserting in an empty substructure takes constant time. □

# 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[$\cdot$], 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?

Why does this only use $\mathcal{O}(n \cdot d(w)) = \mathcal{O}(n \log \log |U|)$ space?

## 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[$\cdot$], 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?

Why does this only use $\mathcal{O}(n \cdot d(w)) = \mathcal{O}(n \log\log|U|)$ space?

# 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[$\cdot$], 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?

# 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[$\cdot$], 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?

# 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$[\cdot]$, 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 $\textsc{Insert}_w$ 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

# $R^2$S-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.

# $\mathrm{R}^2$S-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.

# $\mathrm{R}^2$S-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.

# R$^2$S-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.

# $R^2$S-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.

# Bonus: vEB is optimal for $w = \Theta(\log n)$

In fact, $\mathcal{O}(\log w)$ is the optimal query time when $w \in \Theta(\log n)$, or equivalently when $n \in 2^{\Theta(w)}$. This was shown in:

*Time-space trade-offs for predecessor search*,
Mihai Pătrașcu and Mikkel Thorup,
STOC'06: Proceedings of the thirty-eighth annual ACM symposium on Theory of Computing, pages 232–240.
https://doi.org/10.1145/1132516.1132551

# Bonus: Integer sorting in expected $\mathcal{O}(n \log \log |U|)$ time.

**Idea:** Insert all elements in R²S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log \log |U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

QuickSort Takes expected $\mathcal{O}(n \log n)$ time.

CountingSort Takes $\mathcal{O}(n + |U|)$ time.

RadixSort Takes $\mathcal{O}((n + r)\log_r |U|) = \mathcal{O}((n + r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n} \log |U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log\log|U|)$ time.

**Idea:** Insert all elements in R$^2$S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log\log|U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

$\textsc{QuickSort}$ Takes expected $\mathcal{O}(n \log n)$ time.

$\textsc{CountingSort}$ Takes $\mathcal{O}(n + |U|)$ time.

$\textsc{RadixSort}$ Takes $\mathcal{O}((n + r)\log_r|U|) = \mathcal{O}((n + r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n}\log|U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log\log|U|)$ time.

**Idea:** Insert all elements in R$^2$S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log\log|U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

QuickSort Takes expected $\mathcal{O}(n \log n)$ time.

CountingSort Takes $\mathcal{O}(n + |U|)$ time.

RadixSort Takes $\mathcal{O}((n+r)\log_r|U|) = \mathcal{O}((n+r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n}\log|U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log \log |U|)$ time.

**Idea:** Insert all elements in R$^2$S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log \log |U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

- QUICKSORT Takes expected $\mathcal{O}(n \log n)$ time.
- COUNTINGSORT Takes $\mathcal{O}(n + |U|)$ time.
- RADIXSORT Takes $\mathcal{O}((n + r) \log_r |U|) = \mathcal{O}((n + r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n} \log |U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log\log|U|)$ time.

**Idea:** Insert all elements in R²S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log\log|U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

QuickSort Takes expected $\mathcal{O}(n \log n)$ time.

CountingSort Takes $\mathcal{O}(n + |U|)$ time.

RadixSort Takes $\mathcal{O}((n+r)\log_r|U|) = \mathcal{O}((n+r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n}\log|U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log\log|U|)$ time.

**Idea:** Insert all elements in R$^2$S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log\log|U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

QuickSort Takes expected $\mathcal{O}(n \log n)$ time.

CountingSort Takes $\mathcal{O}(n + |U|)$ time.

RadixSort Takes $\mathcal{O}((n+r)\log_r|U|) = \mathcal{O}((n+r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n}\log|U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Bonus: Integer sorting in expected $\mathcal{O}(n \log \log |U|)$ time.

**Idea:** Insert all elements in R$^2$S-vEB structure, use $\textsc{min}(S)$ then repeatedly $\textsc{successor}(x, S)$.

This takes expected $\mathcal{O}(n \log \log |U|) = \mathcal{O}(n \log w)$ time.

Compare this to other famous sorting methods:

$\textsc{QuickSort}$ Takes expected $\mathcal{O}(n \log n)$ time.

$\textsc{CountingSort}$ Takes $\mathcal{O}(n + |U|)$ time.

$\textsc{RadixSort}$ Takes $\mathcal{O}((n + r) \log_r |U|) = \mathcal{O}((n + r)\frac{w}{\log r})$ time for radix $r$, e.g. $\mathcal{O}(\frac{n}{\log n} \log |U|) = \mathcal{O}(\frac{nw}{\log n})$ for $r = n$.

We observe that for $\log n \in \Omega(\log w) \cap \mathcal{O}(\frac{w}{\log w})$, or equivalently $n \in w^{\Omega(1)} \cap 2^{\mathcal{O}(\frac{w}{\log w})}$, sorting using vEB should be better than these alternatives.

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2S$-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using “indirection” ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($\mathrm{R}^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using "indirection" ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness

# Summary

Todays topic was van Emde Boas Trees, which is a data structure for sets of bounded integers. We have covered

- A naive implementation of ordered sets.
- A slightly less naive two-level implementation.
- A recursive implementation with good query time (proto-vEB).
- The actual van Emde Boas Tree (vEB).
- An extension to vEB using hashing to save space (RS-vEB).
- A further space-saving extension using “indirection” ($R^2$S-vEB).
- Some notes about the optimality of vEB and on integer sorting.
- Next time: NP-completeness