van Emde Boas Trees

rsy3244

July 1, 2019

Contents

Contents

van Emde Boas tree とは (1/2)

van Emde Boas tree (以下 vEB 木) は動的集合を扱うデータ構造

動的集合

動的集合とは, 集合に対して後述の MEMBER(V, x), INSERT(V, x), DELETE(V, x) が行えるようなデータ構造

- 今回扱う要素は非負整数とする.
- vEB 木が保持しうる要素の集合を全体集合 U とし, その大きさを u とする.
- vEB 木が現在保持している集合を V とし, その大きさを n とする.

van Emde Boas tree とは (2/2)

van Emde Boas tree は以下の操作を行うことができる.

操作

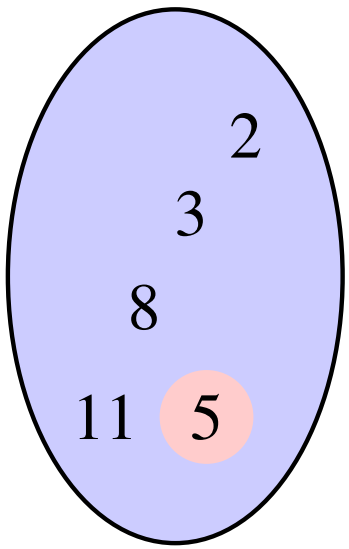
$\textsc{Member}(V, x)$	: V に x が存在するかを返す.
$\textsc{Min}(V)$	: V の要素の最小値を返す.
$\textsc{Max}(V)$	: V の要素の最大値を返す.
$\textsc{Successor}(V, x)$	: V の x より大きい最小の要素を返す.
$\textsc{Predecessor}(V, x)$	: V の x より小さい最大の要素を返す.
$\textsc{Insert}(V, x)$	: V に x を挿入する.
$\textsc{Delete}(V, x)$	: V から x を削除する.

vEB 木では, これらの操作が最悪時間計算量 $O(\log \log u)$ で実行可能

操作: MEMBER(V, x)

- MEMBER(V, x) は, x が集合 V に存在するかを真偽値で返す.

$V = \{2, 3, 5, 8, 11\}$



- MEMBER($V, 5$) = TRUE
- MEMBER($V, 6$) = FALSE

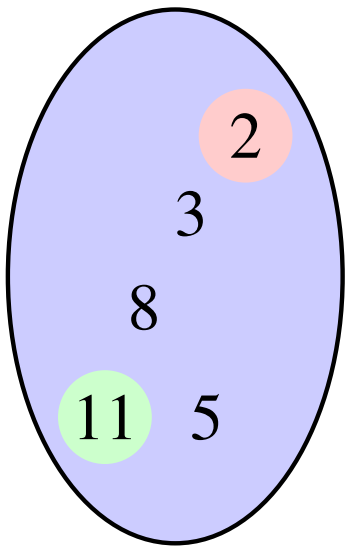
各関数の引数

関数に渡す引数は U の要素のみ

操作: Min(V), Max(V)

- Min(V) は, 集合 V の要素の最小値を返す.
- Max(V) は, 集合 V の要素の最大値を返す.

$V = \{2, 3, 5, 8, 11\}$

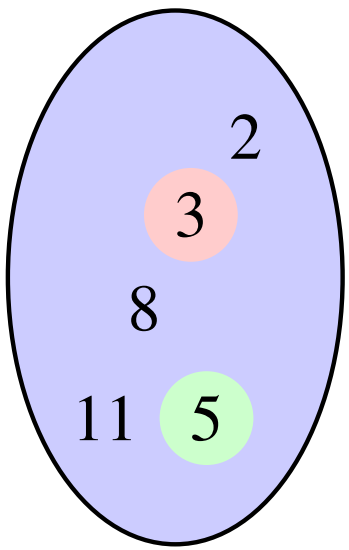


- Min(V) = 2
- Max(V) = 11

操作: Successor(V, x), Predecessor(V, x)

- Successor(V, x) は, 集合 V の x より大きい最小の要素を返す.
- Predecessor(V, x) は, 集合 V の x より小さい最大の要素を返す.

$V = \{2, 3, 5, 8, 11\}$

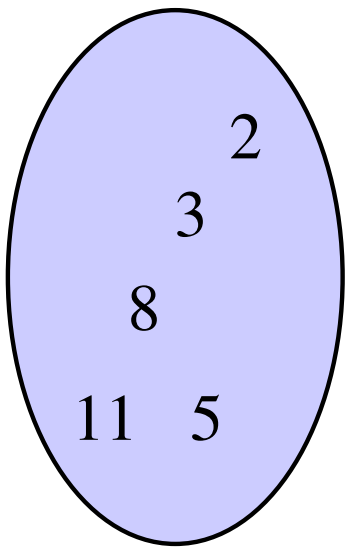


- Successor($V, 2$) $= 3$
- Predecessor($V, 7$) $= 5$

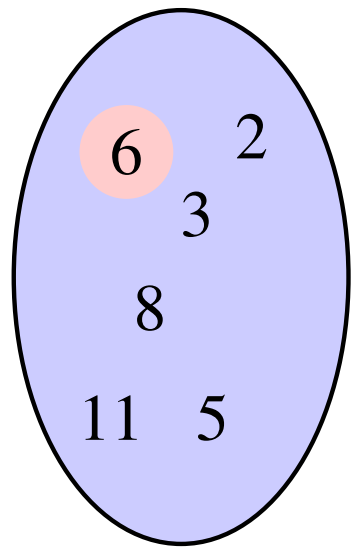
操作: INSERT(V, x)

- INSERT(V, x) は, 集合 V に x を挿入する.
  - $V \leftarrow V \cup \{x\}$

$V = \{2, 3, 5, 8, 11\}$



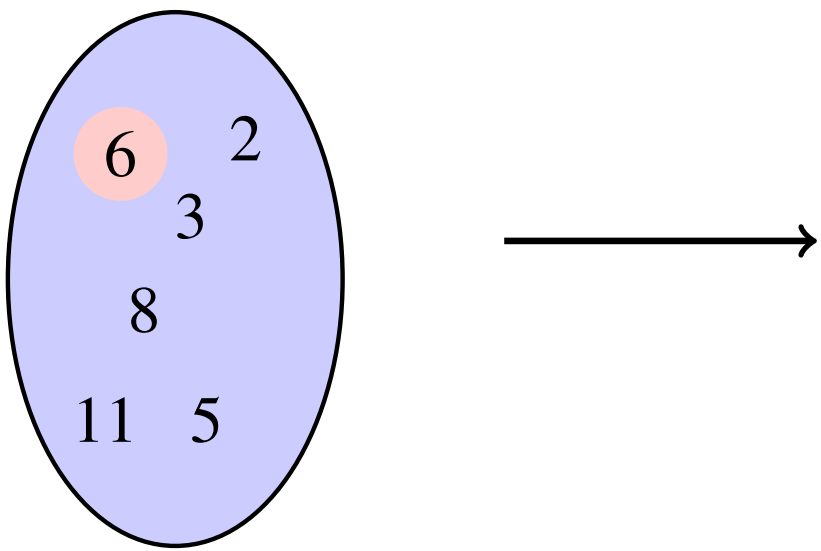
$V = \{2, 3, 5, \mathbf{6}, 8, 11\}$



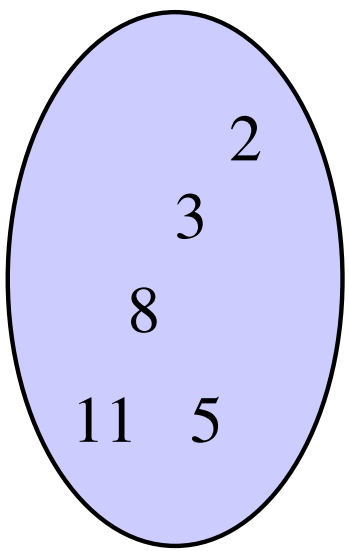
操作: DELETE(V, x)

- DELETE(V, x) は, 集合 V から x を削除する.
  - $V \leftarrow V \setminus \{x\}$

$V = \{2, 3, 5, \mathbf{6}, 8, 11\}$



$V = \{2, 3, 5, 8, 11\}$



Contents

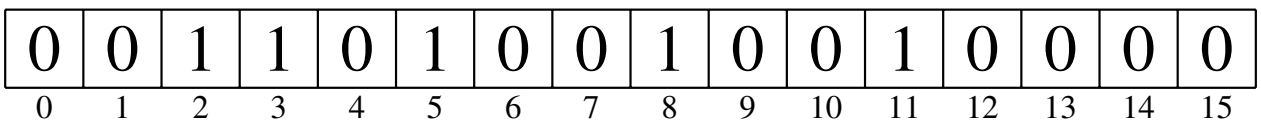
直接アドレス法 (1/2)

動的集合を保持する手段として, 直接アドレス法を考える.

直接アドレス法

要素の値を配列の添字として利用し, データを保持するテクニック

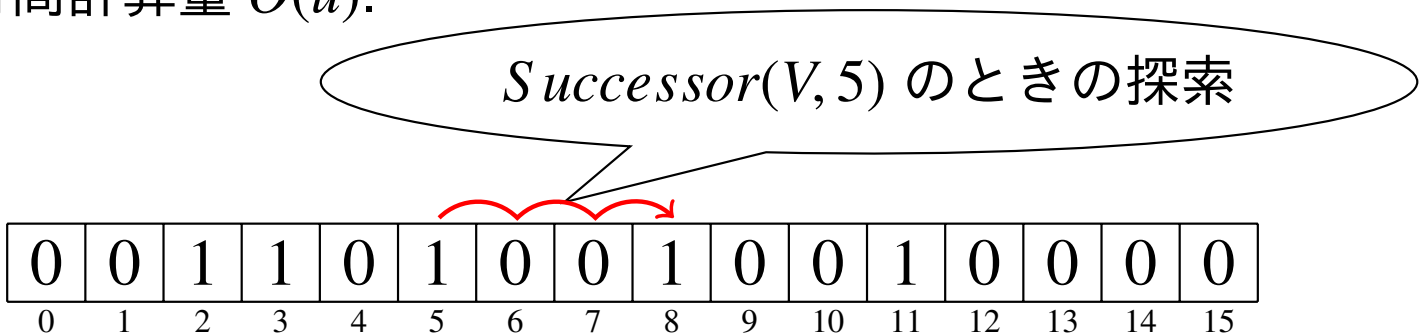
今回のデータ構造ではハッシュ表のような付属データはないので,
各配列の要素には要素を保持しているかを bit で格納する.



$$V = \{2, 3, 5, 8, 11\}$$

直接アドレス法 (2/2)

- MEMBER(V, x), INSERT(V, x), DELETE(V, x) は,
  配列のランダムアクセスが定数時間なので, 時間計算量 $O(1)$.
- SUCCESSOR(V, x) は, x の次の値が 1 の要素まで
  最悪 $\Theta(u)$ 回探索する必要があるので, 時間計算量 $O(u)$.
  - PREDECESSOR(V, x), MIN(V), MAX(V) も同様の処理を行うため,
    時間計算量 $O(u)$.



$$V = \{2, 3, 5, 8, 11\}$$

二分木構造 (1/3)

Successor(V, x) で必要な探索回数を小さくするために,
配列の各要素を葉とした二分木構造を考える.

二分木

二分木とは, 葉ではない頂点の子が常に 2 個であるような木

木 : 閉路を持たない, 連結なグラフ

根 : 木の頂点のうちの 1 つを定義する

子 : ある頂点について, 隣接する頂点のうち根から遠いもの

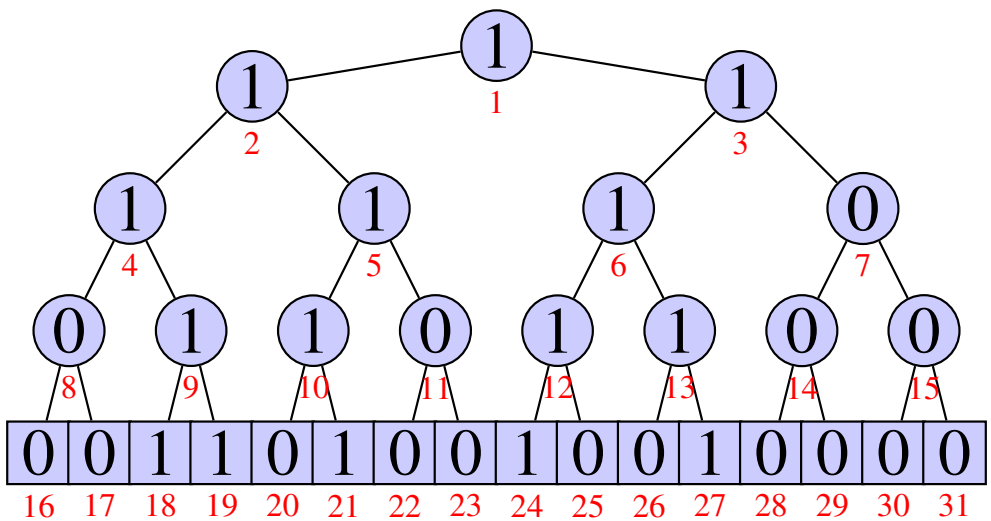
親 : ある頂点について, 隣接する頂点のうち根に近いもの

葉 : 木の頂点のうち, 子を持たないもの

二分木構造 (2/3)

葉ではない各頂点には, 子の値の論理和を格納する.

- ある頂点の値が 1 の場合,
  その頂点の下にある葉の少なくとも 1 つの頂点は値が 1.
- 根から深さ毎に左から配列に詰めて保持することで
  各頂点にランダムアクセスが可能となる.
  - 赤字は格納されている配列の添字

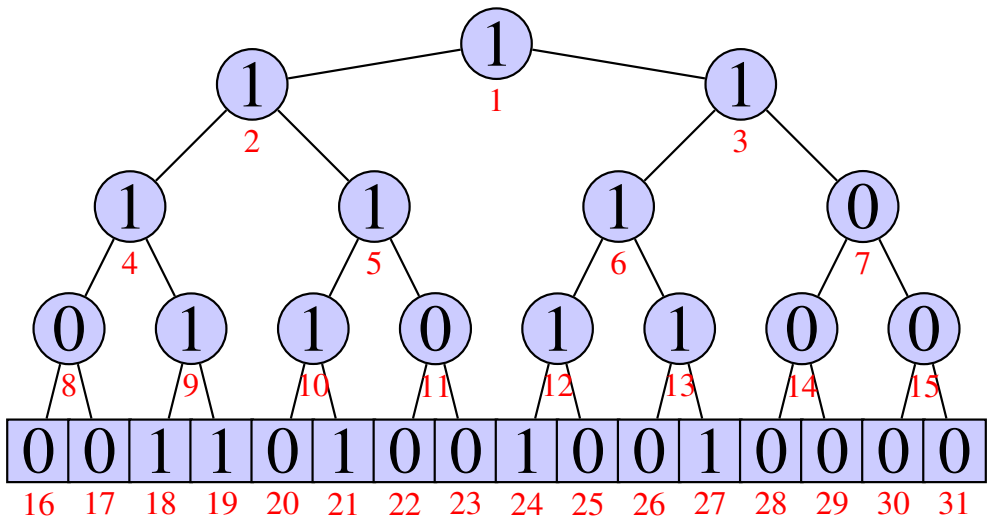


$$V = \{2, 3, 5, 8, 11\}$$

二分木構造 (3/3)

二分木中のある頂点の添字を i とすると,
以下のように他の頂点の添字を求めることができる.

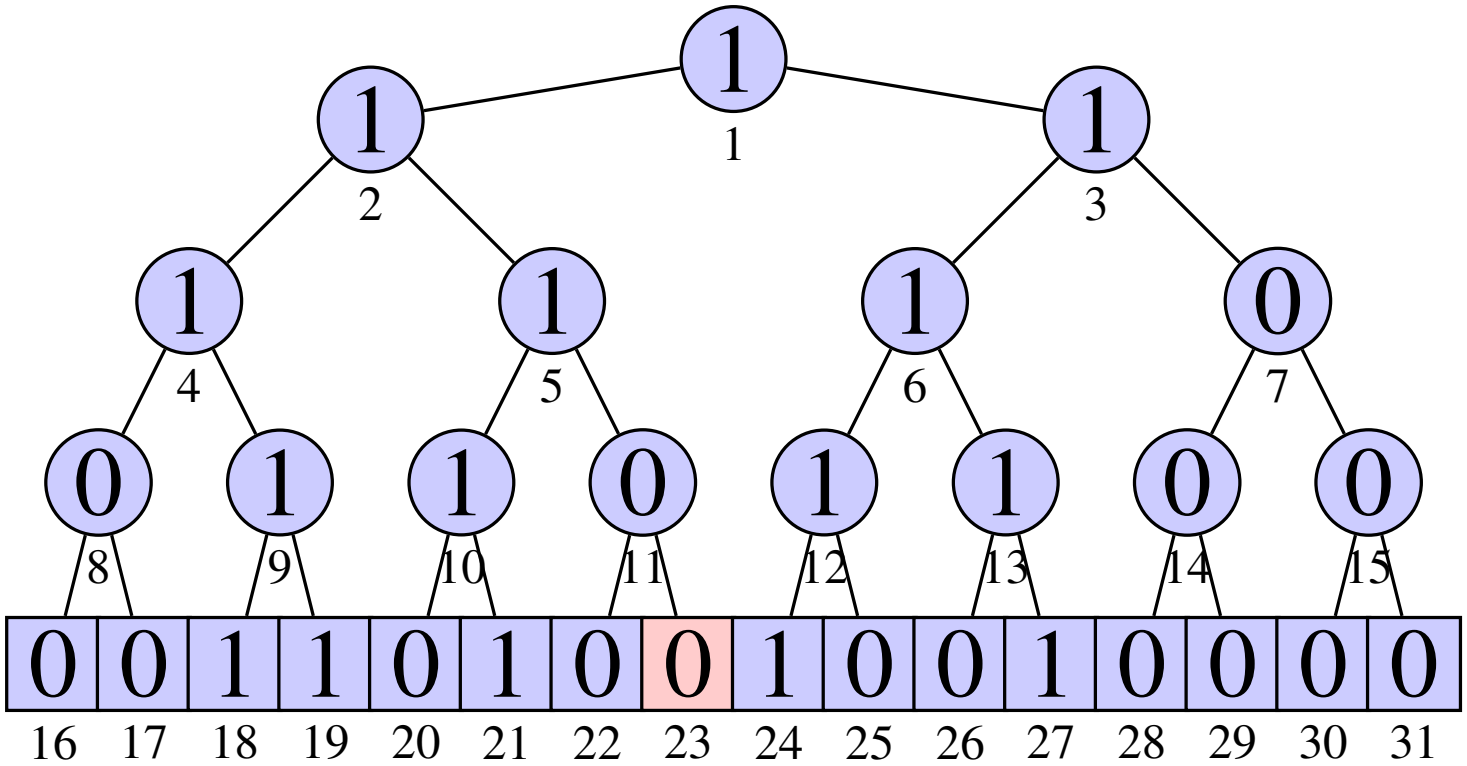
- 根 ：先頭なので 1
- 子 ：$2 \times i$ と $2 \times i + 1$
- 親 ：$\lfloor \frac{i}{2} \rfloor$
- 葉 ：集合 U の要素 v に対応する葉の添字は $i + u$



$$V = \{2, 3, 5, 8, 11\}$$

二分木構造 操作: MEMBER(V, x)

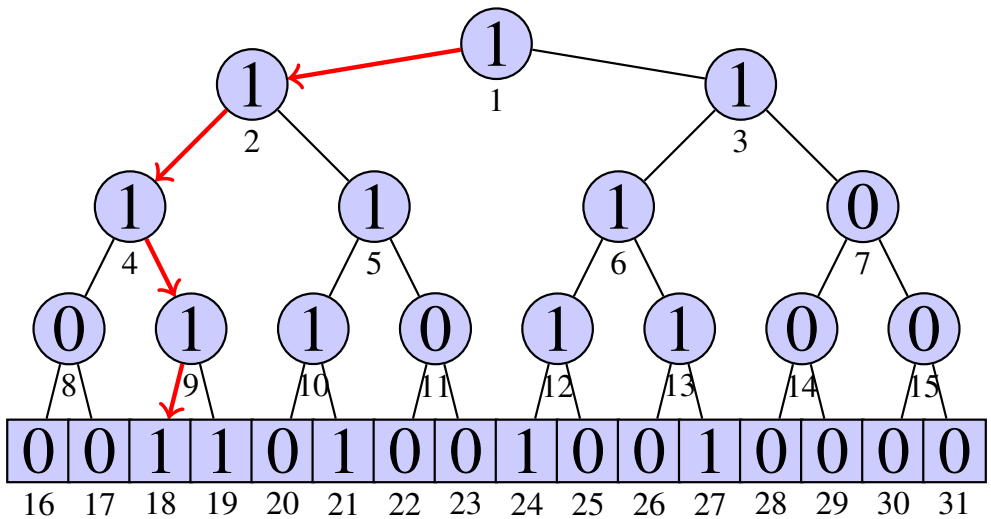
- MEMBER(V, x) は, 対応する葉の値を返す.



MEMBER$(V, 7), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: Min(V), Max(V)

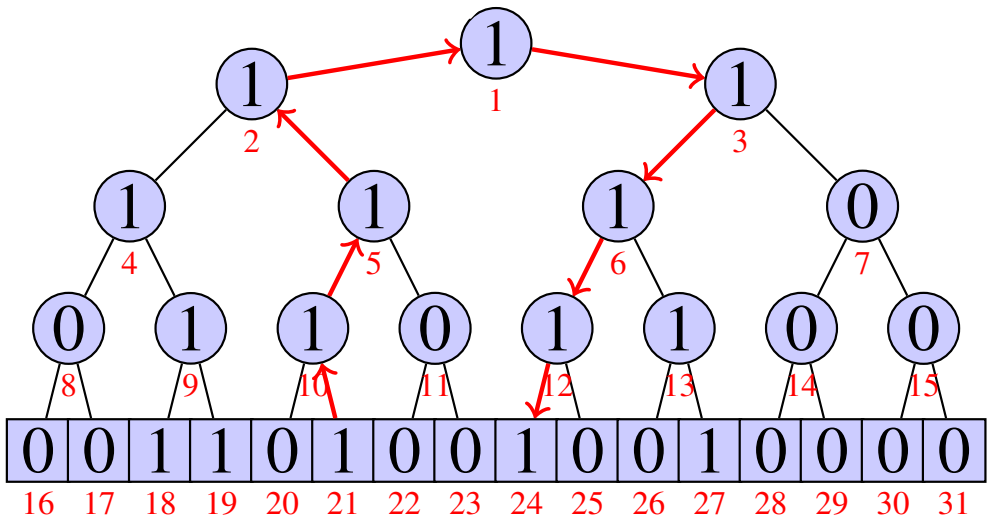
- Min(V) は, 根から左の子が 1 であれば左の子へ,
  そうでなければ右の子へ降りる操作を再帰的に行う.
  - Max(V) も同様に処理する.



Min(V), $V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: Successor(V, x), Predecessor(V, x)

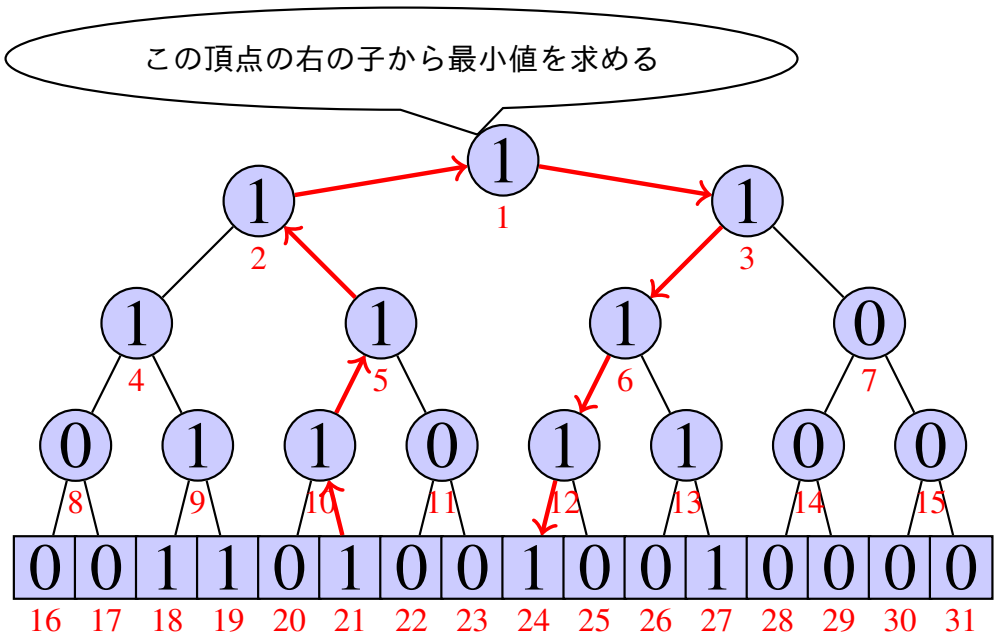
- Successor(V, x) は, x の親から, 直前の頂点が左の子であり, 右の子の値が 1 の頂点になるまで親を辿り, 右の子の最小値を返す.
  - Predecessor(V, x) も同様に処理する.



Successor$(V, 5), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: Successor(V, x), Predecessor(V, x)

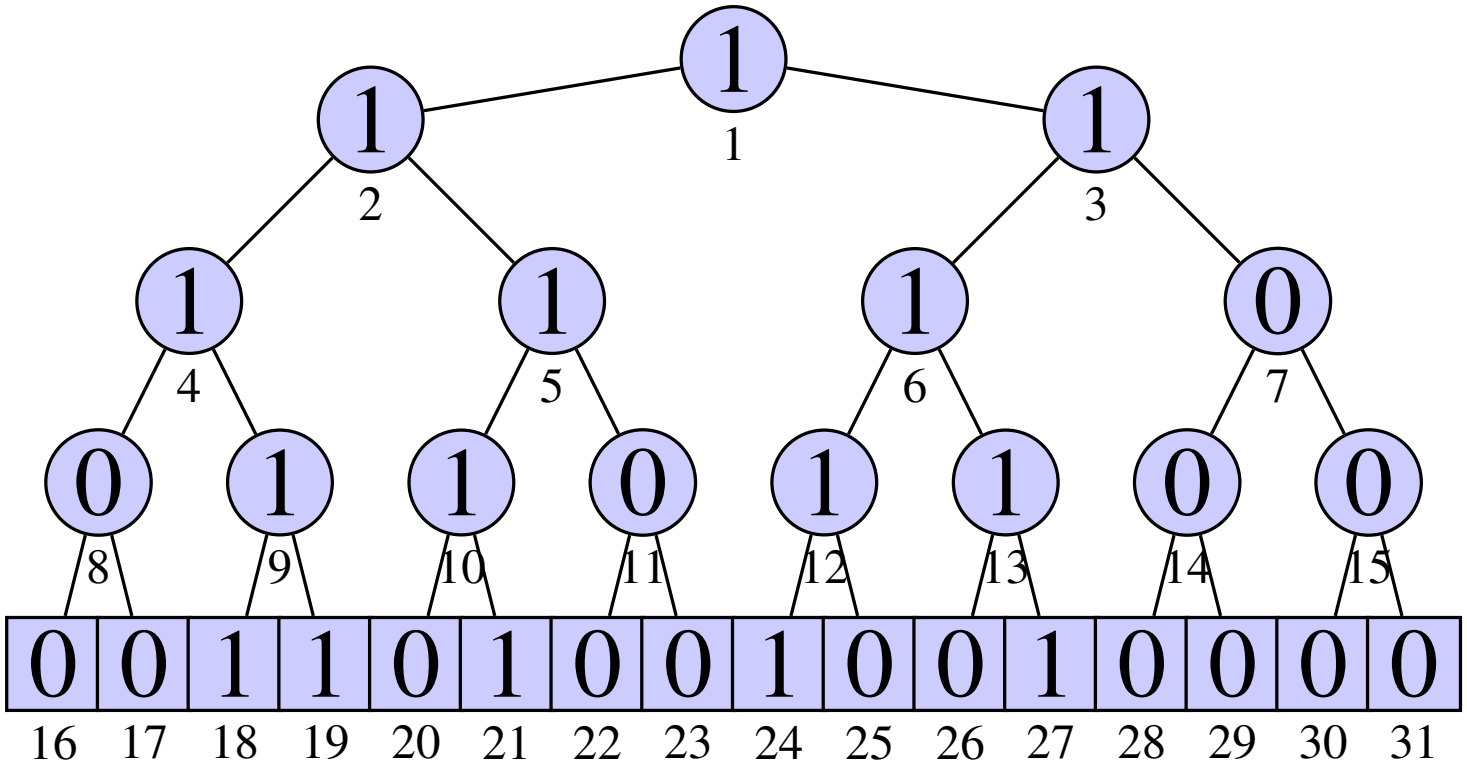
- Successor(V, x) は, x の親から, 直前の頂点が左の子であり, 右の子の値が 1 の頂点になるまで親を辿り, 右の子の最小値を返す.
  - Predecessor(V, x) も同様に処理する.



Successor$(V, 5), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

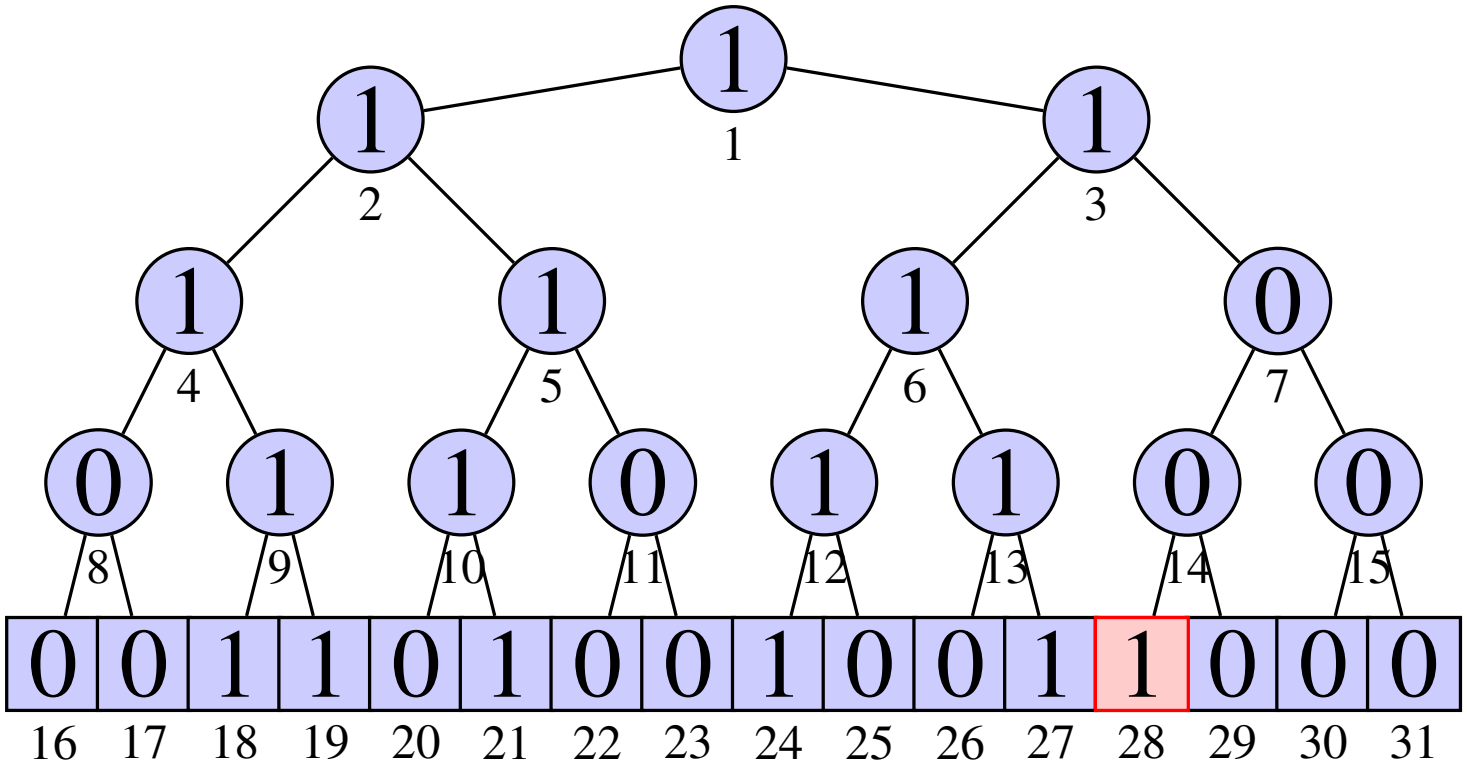
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

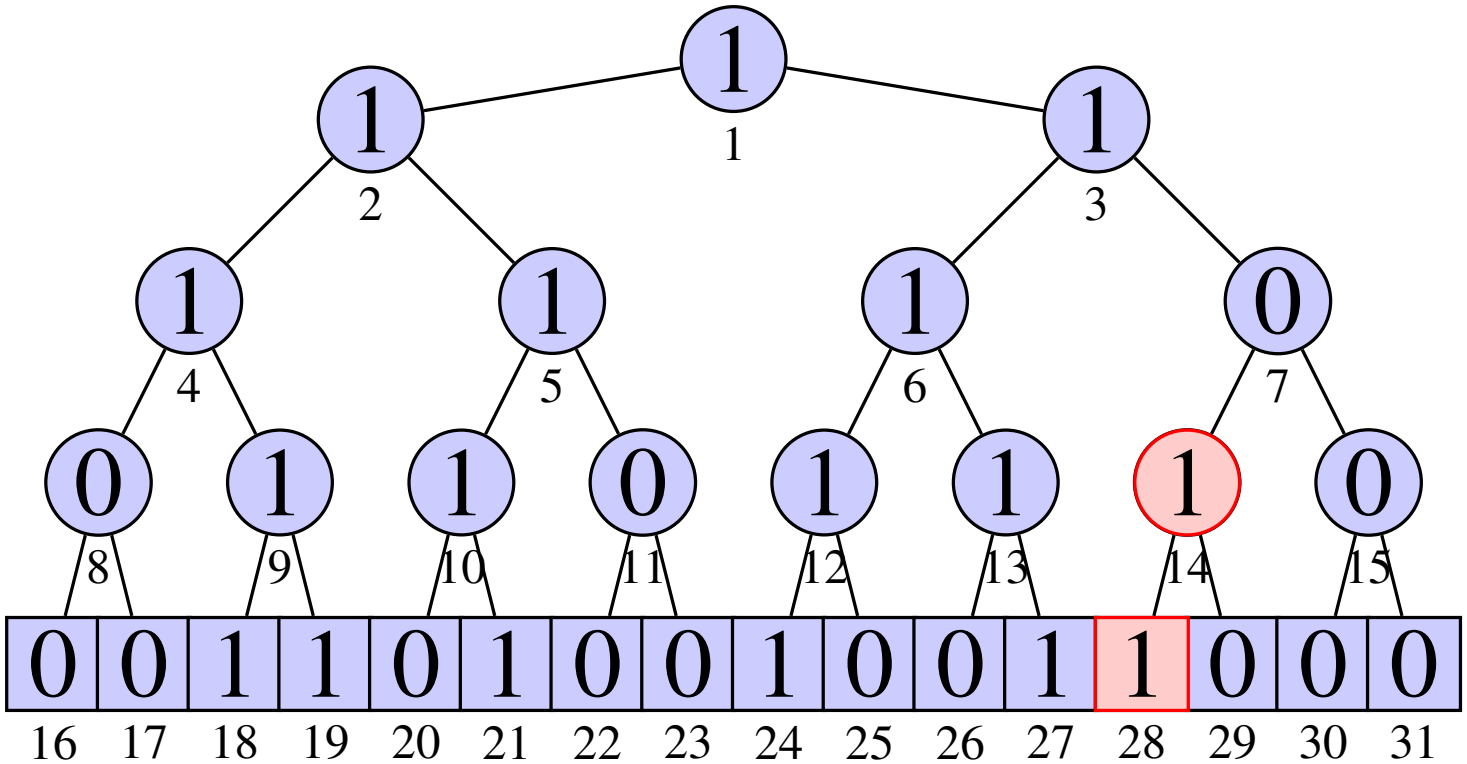
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

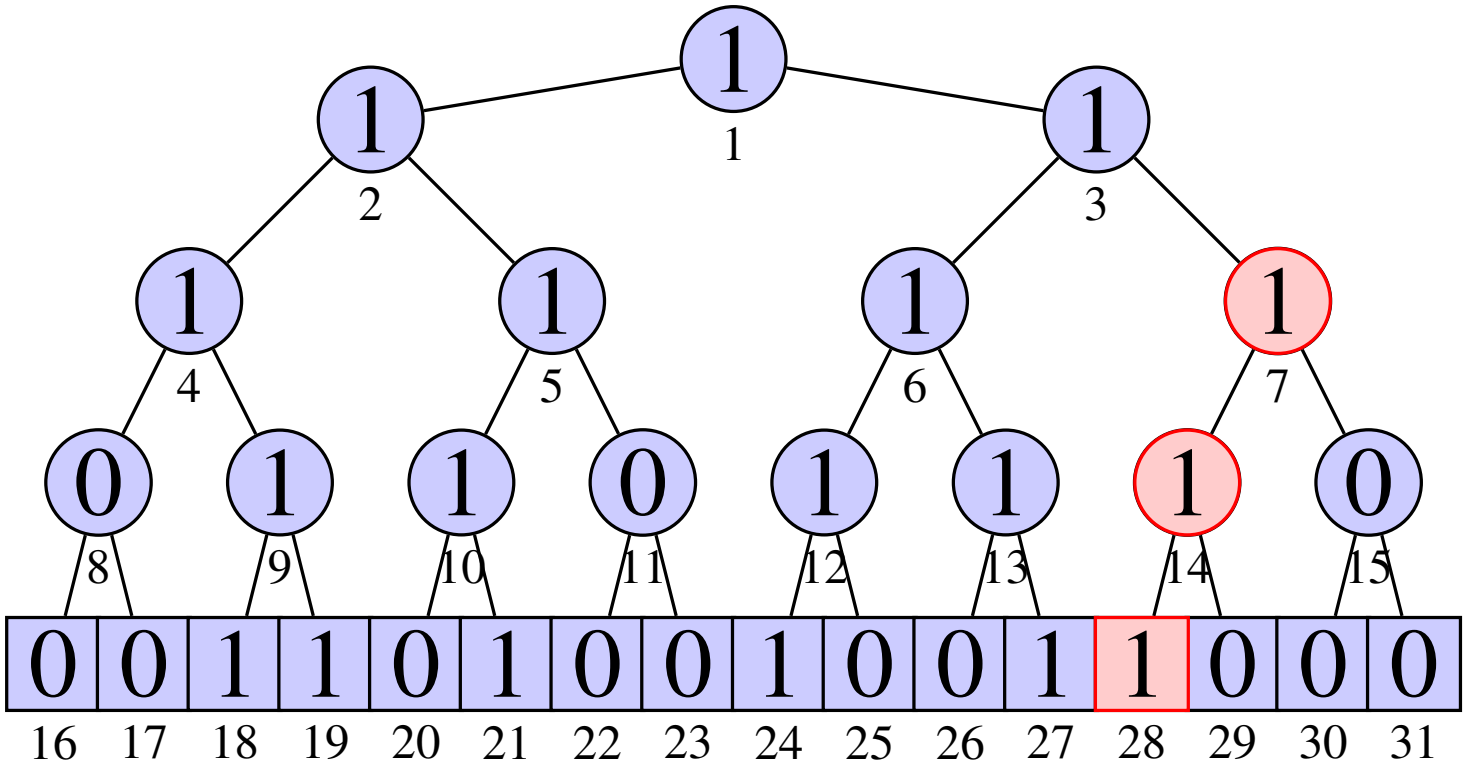
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

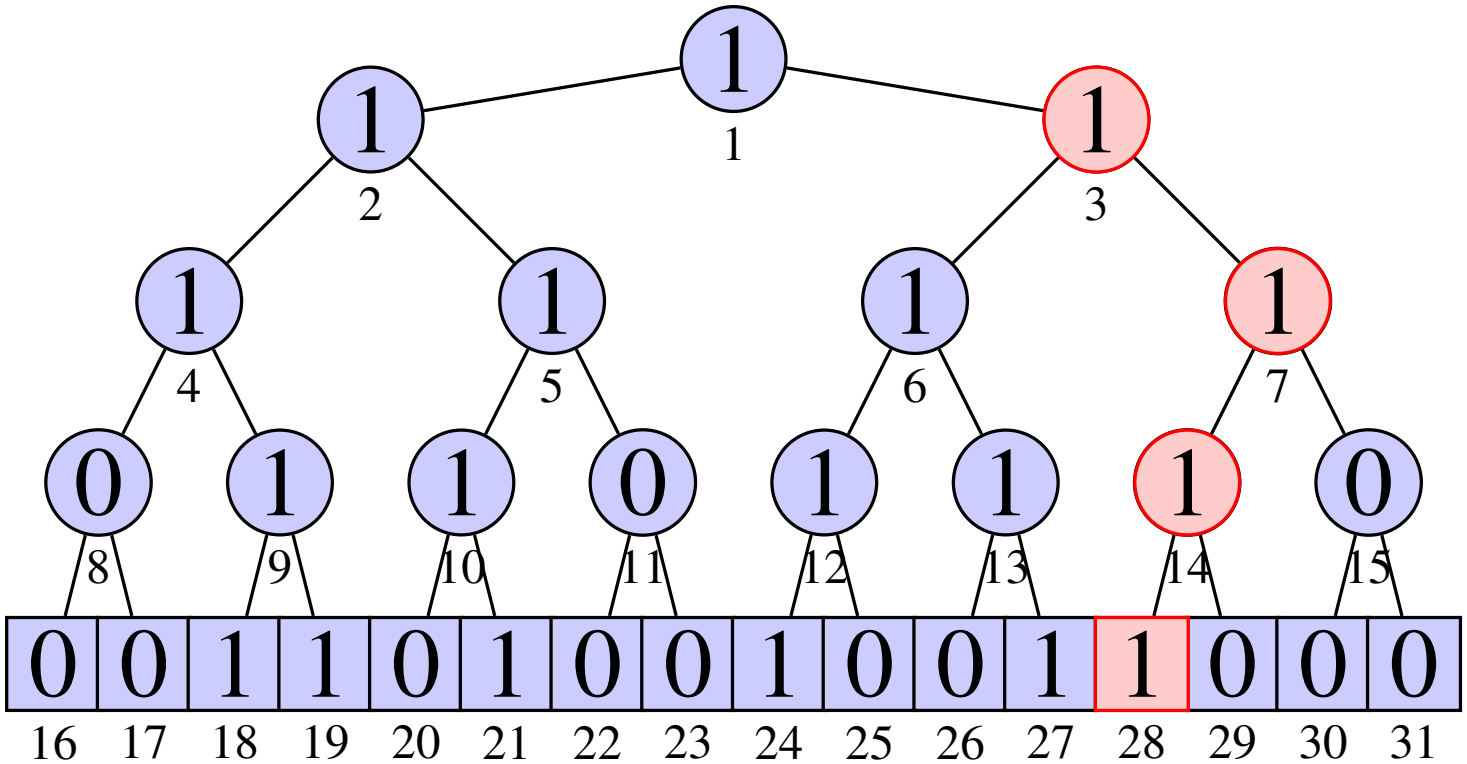
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

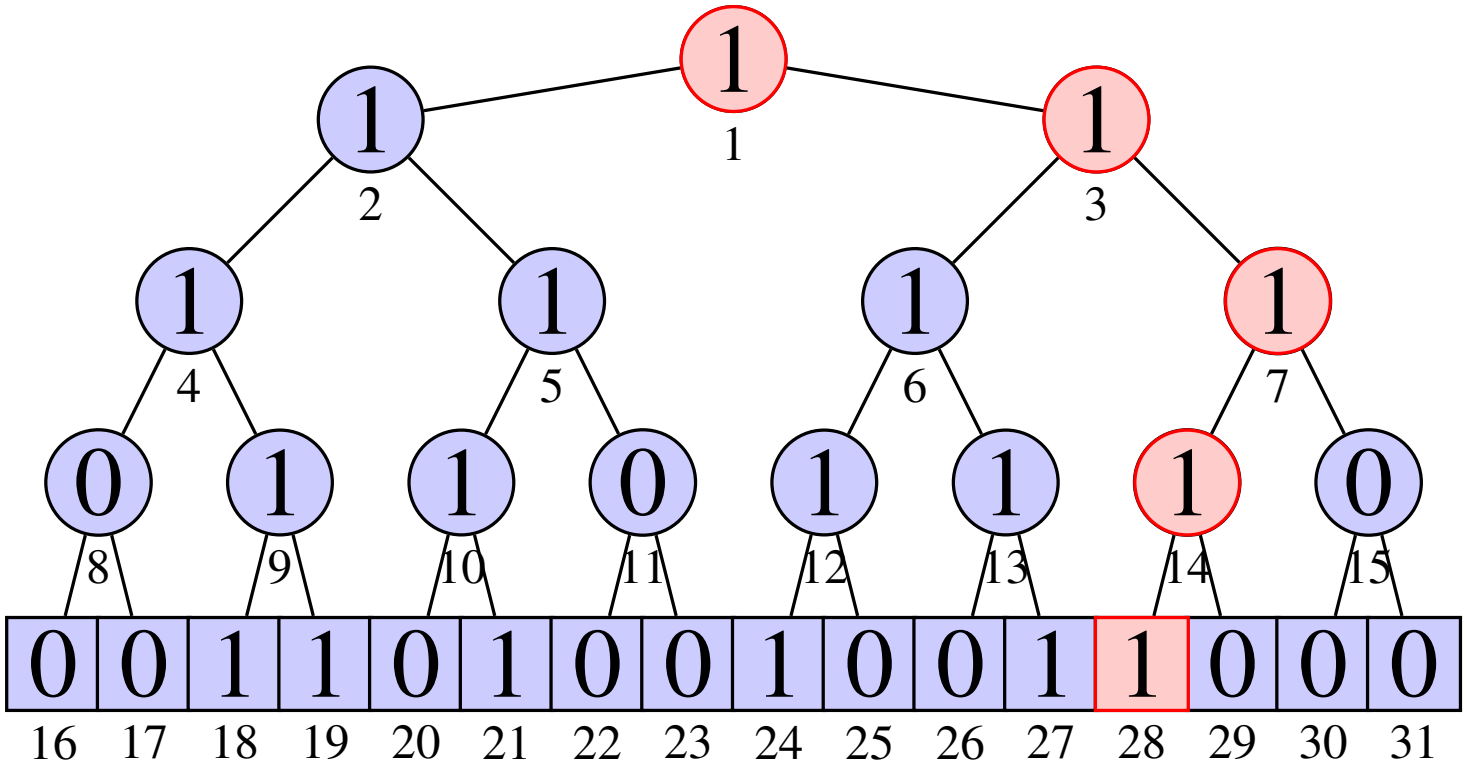
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: INSERT(V, x)

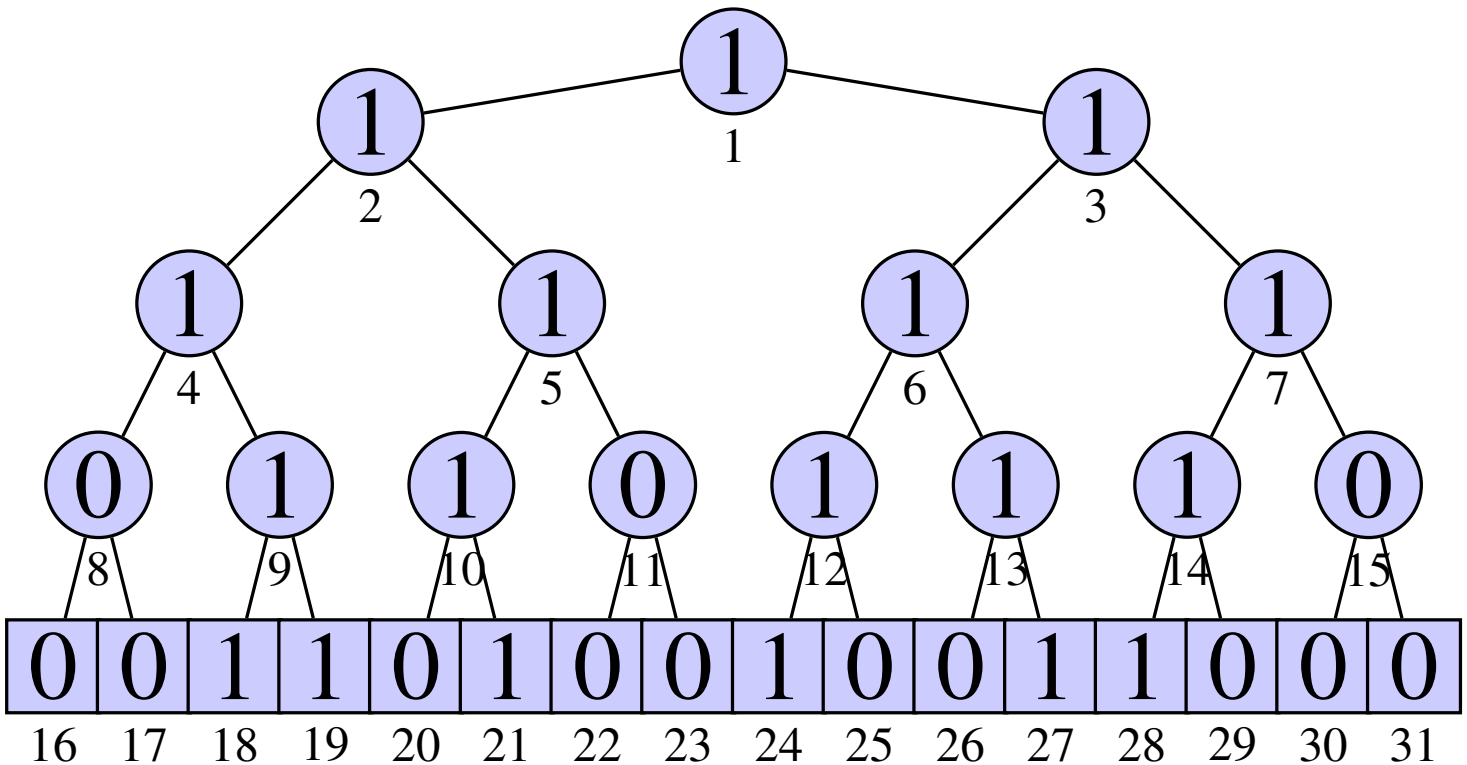
- INSERT(V, x) は, 対応する葉から親の値を 1 にしつつ根まで辿る.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作: DELETE(V, x)

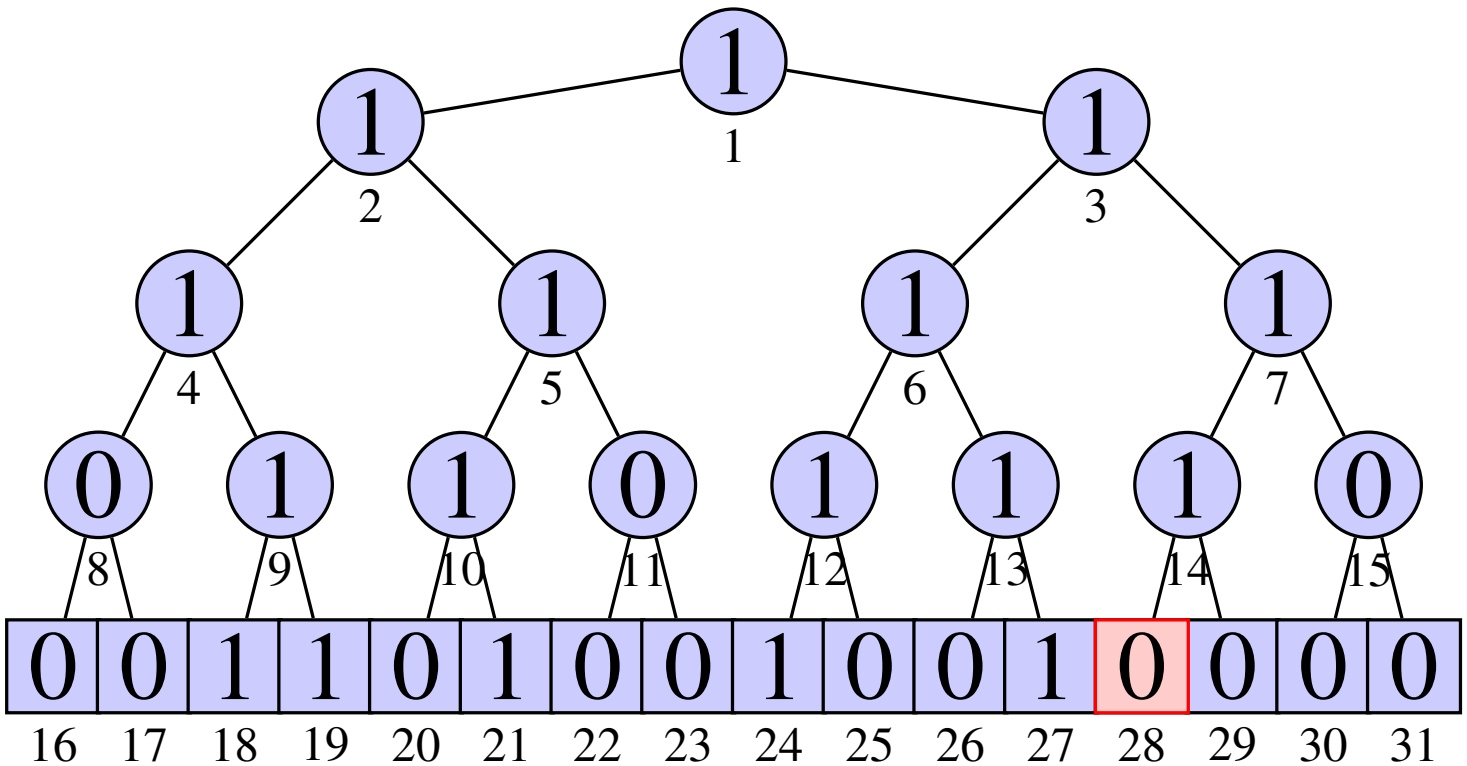
- DELETE(V, x) は, 対応する葉の値を 0 にした後,
  以下の操作を行いながら根まで辿る.
  - 子が両方 0 ならば, 自身の値を 0 にし, 親の頂点に遷移する.



DELETE$(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作: DELETE(V, x)

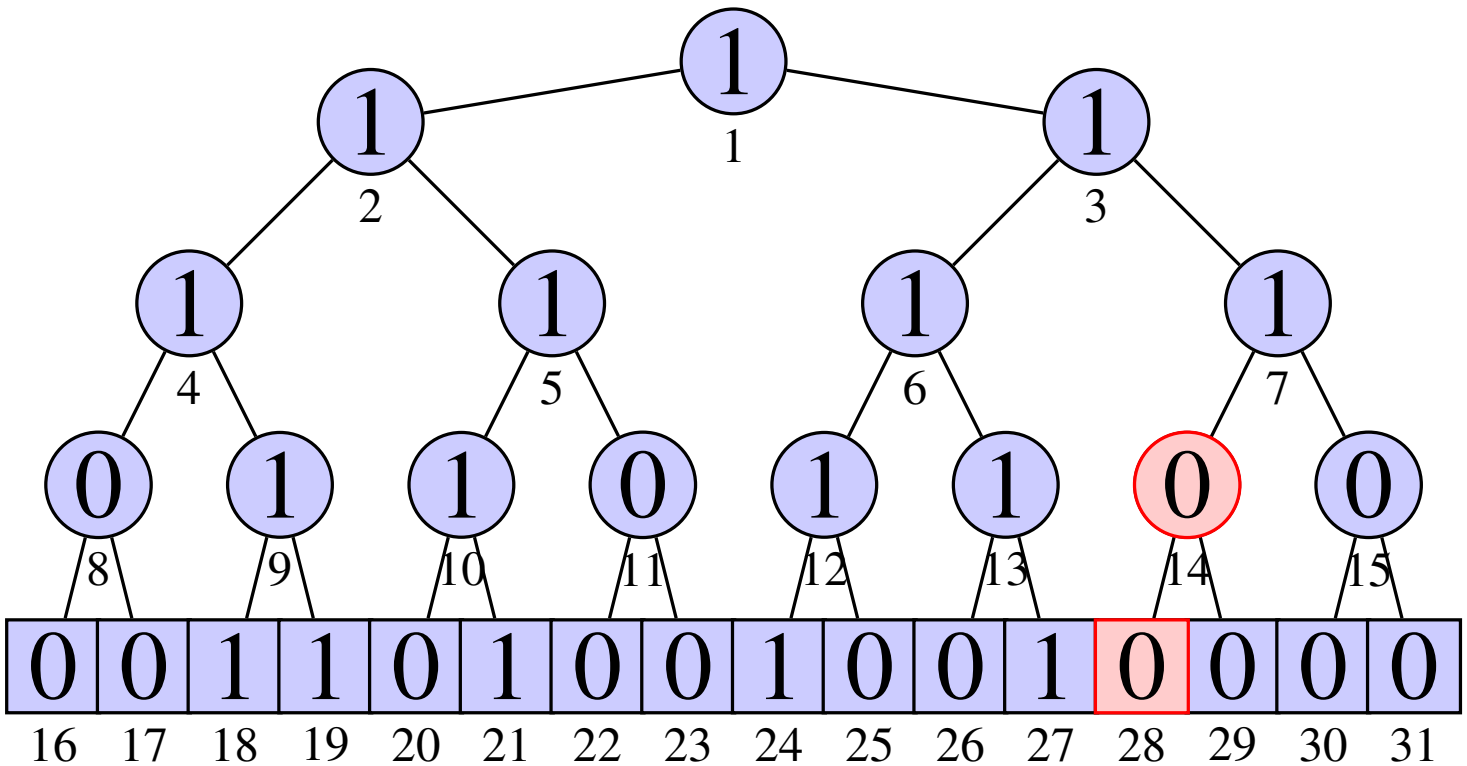
- DELETE(V, x) は, 対応する葉の値を 0 にした後,
  以下の操作を行いながら根まで辿る.
  - 子が両方 0 ならば, 自身の値を 0 にし, 親の頂点に遷移する.



DELETE$(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作: DELETE(V, x)

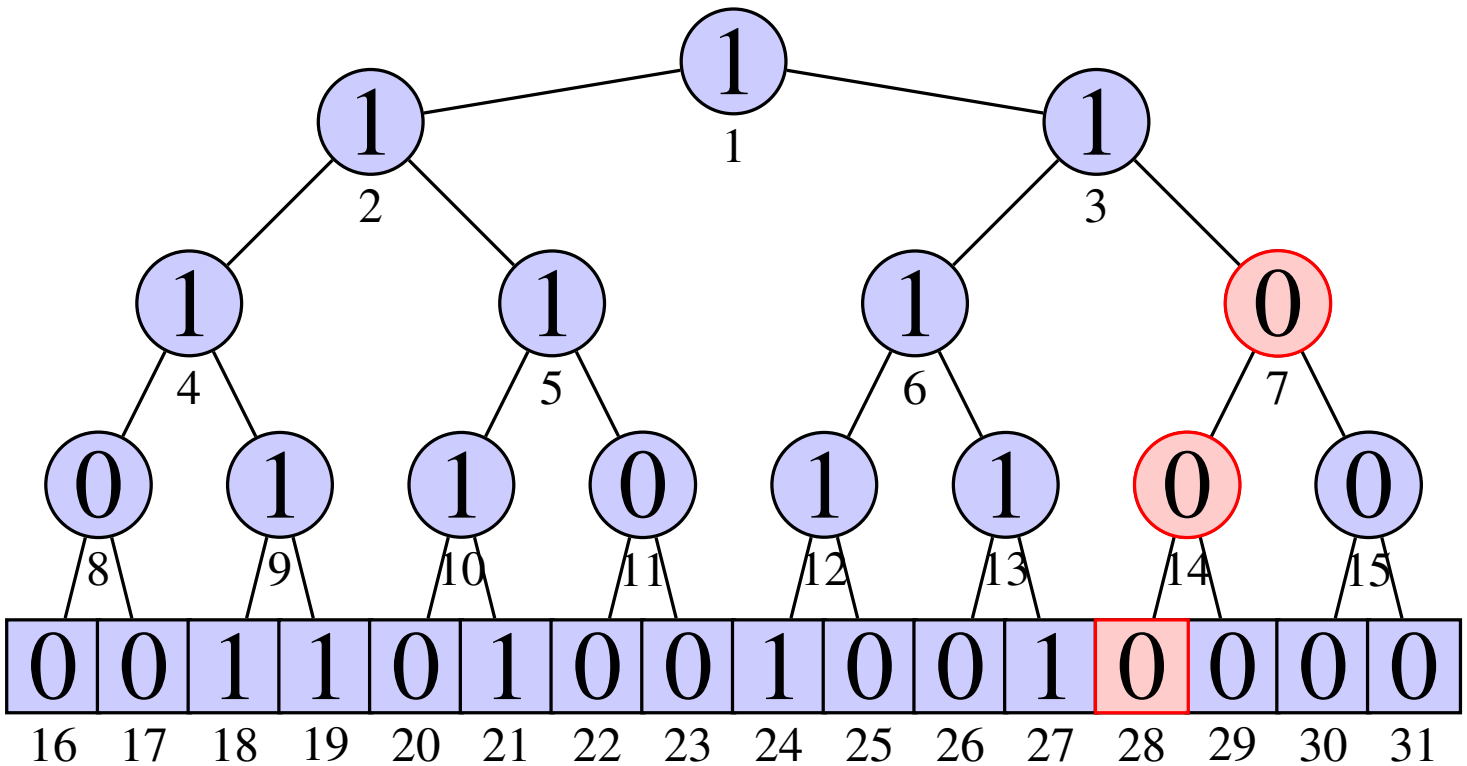
- DELETE(V, x) は, 対応する葉の値を 0 にした後,
  以下の操作を行いながら根まで辿る.
  - 子が両方 0 ならば, 自身の値を 0 にし, 親の頂点に遷移する.



DELETE$(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作: DELETE(V, x)

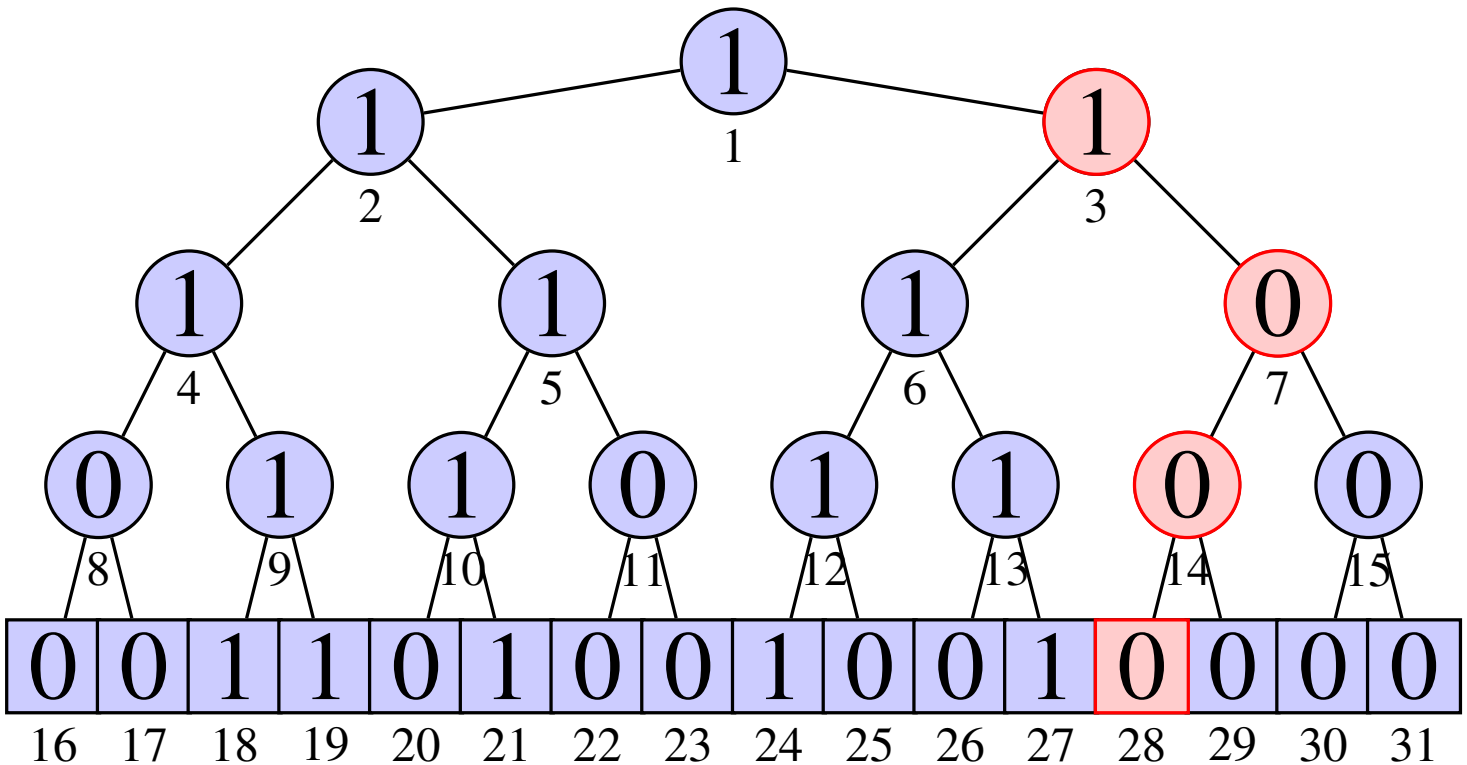
- DELETE(V, x) は, 対応する葉の値を 0 にした後,
  以下の操作を行いながら根まで辿る.
  - 子が両方 0 ならば, 自身の値を 0 にし, 親の頂点に遷移する.



DELETE$(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作: DELETE(V, x)

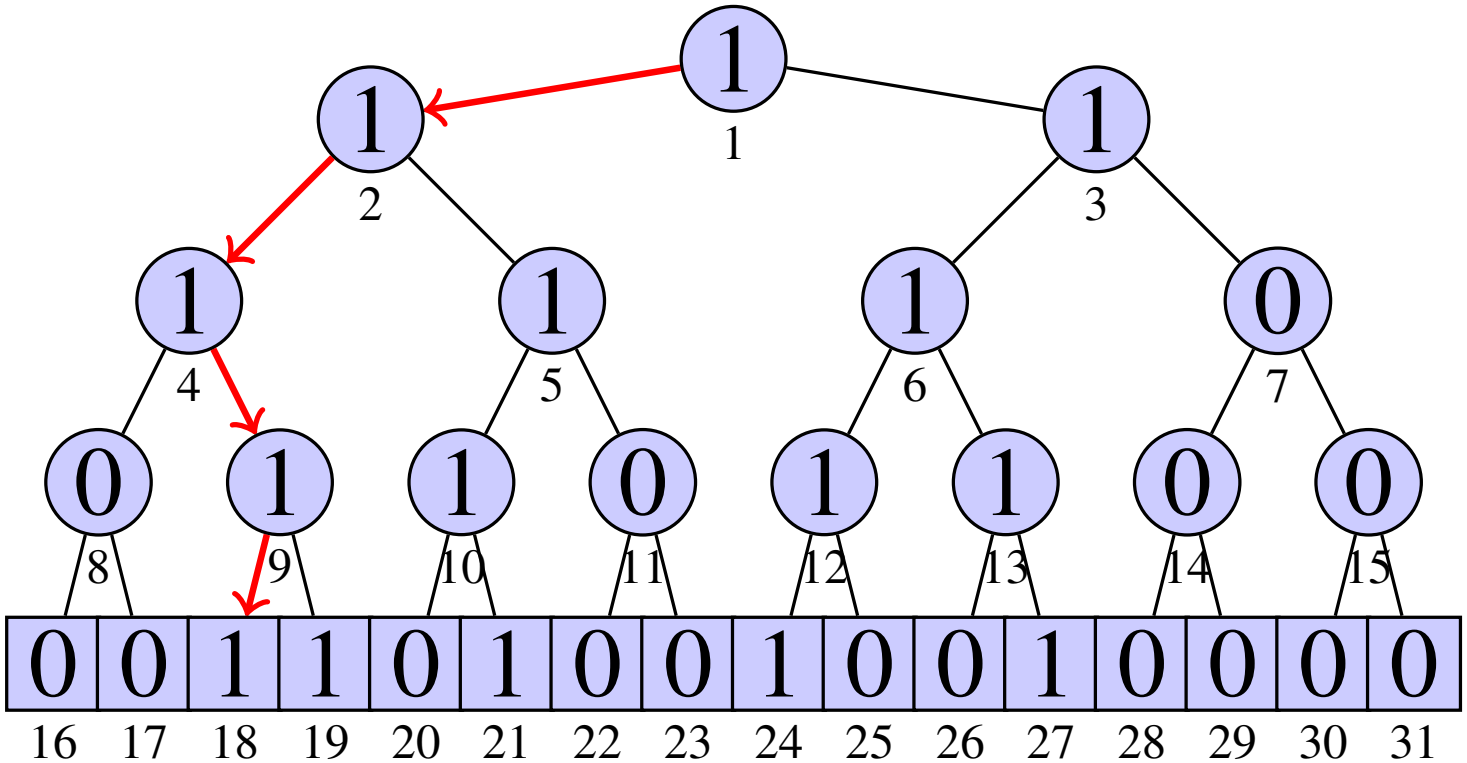
- DELETE(V, x) は, 対応する葉の値を 0 にした後,
  以下の操作を行いながら根まで辿る.
  - 子が両方 0 ならば, 自身の値を 0 にし, 親の頂点に遷移する.



DELETE$(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 計算量 (1/4)

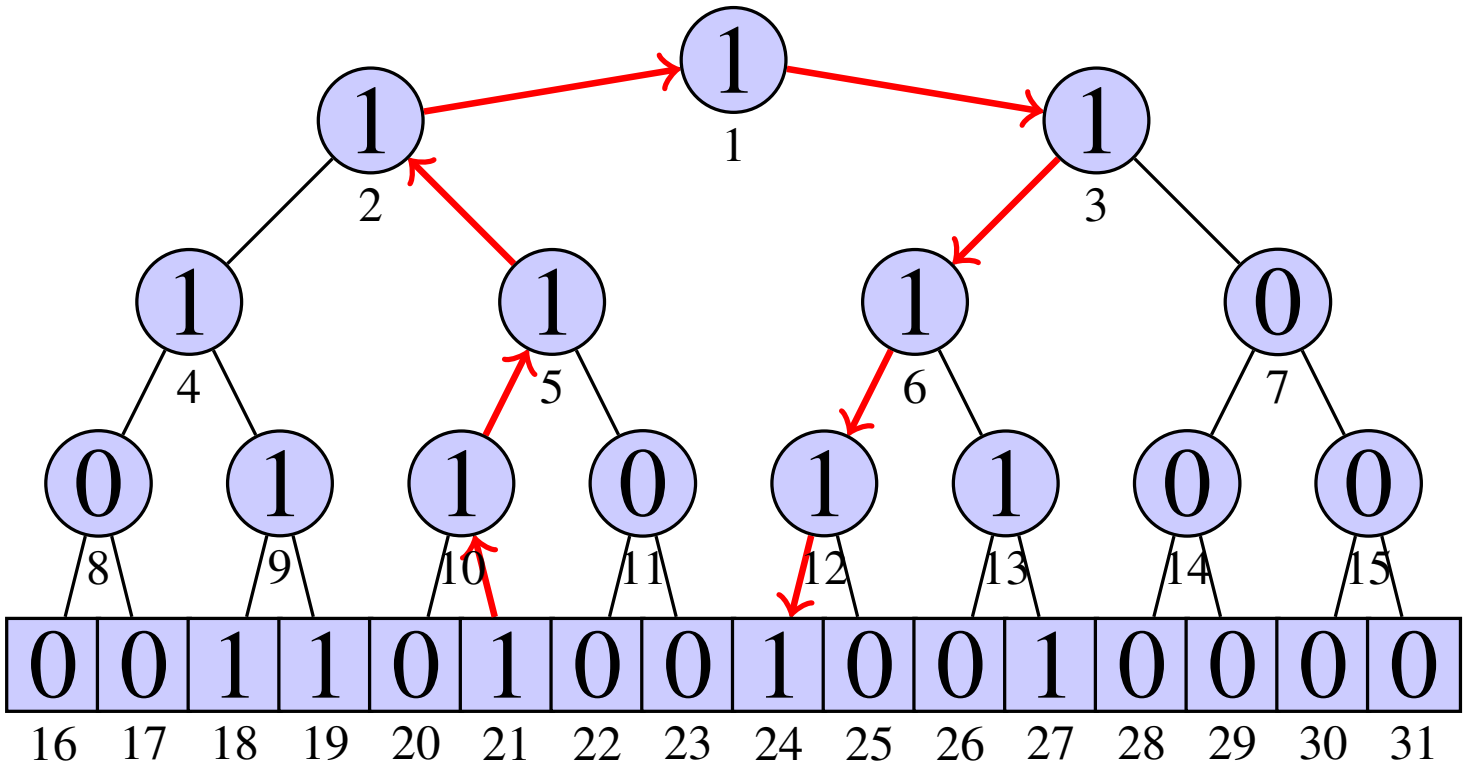
- $\textsc{Min}(V)$, $\textsc{Max}(V)$ では, 根から最小値となる葉まで頂点を探索するので, 時間計算量は $\Theta(\log u)$.



$\textsc{Min}(V)$, $V = \{2, 3, 5, 8, 11\}$

二分木構造 計算量 (2/4)

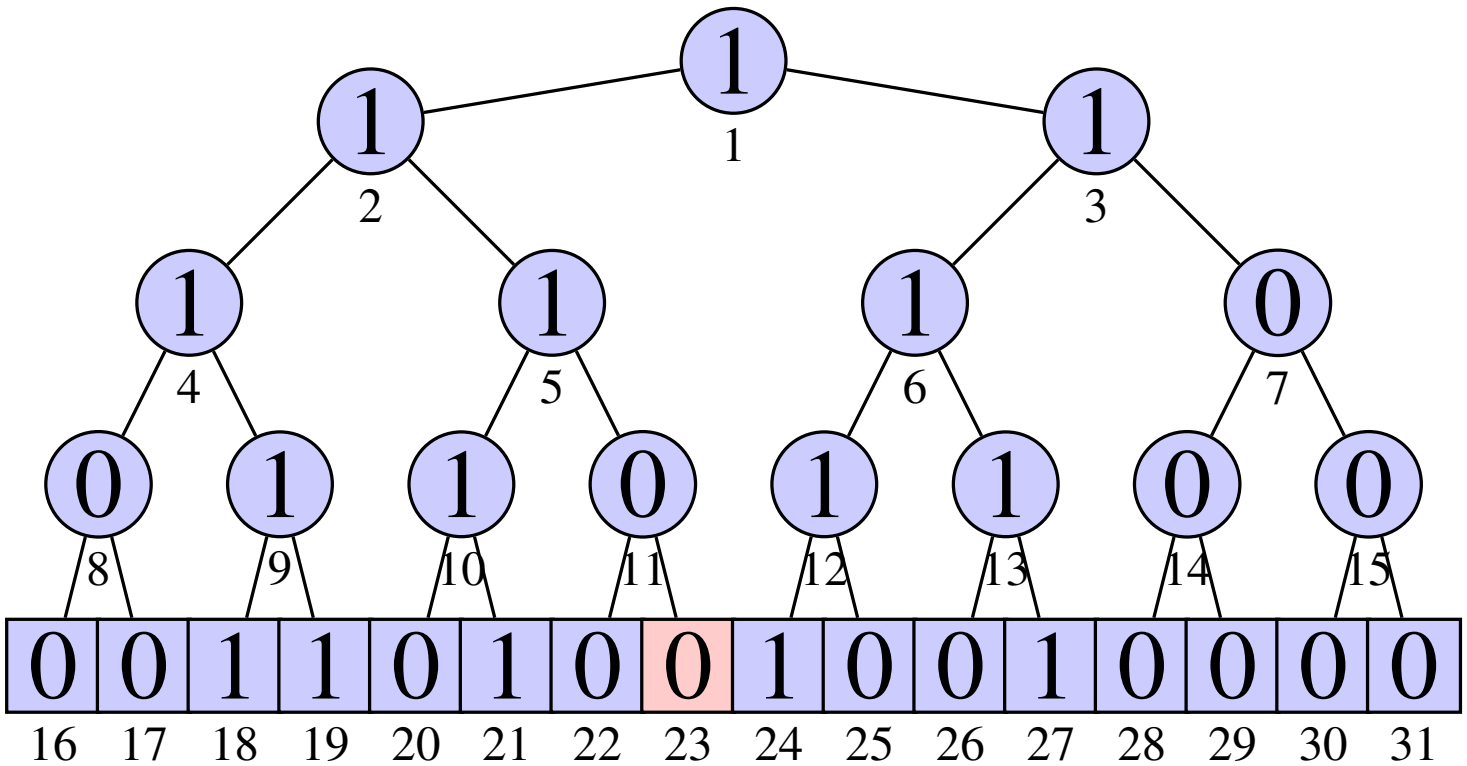
- SUCCESSOR(V, x), PREDECESSOR(V, x) では, 葉から最悪根まで辿り, その後, MIN(V) (または MAX(V)) と同様の処理を行うことから, 時間計算量は $O(\log u)$.



SUCCESSOR$(V, 5), V = \{2, 3, 5, 8, 11\}$

二分木構造 計算量 (3/4)

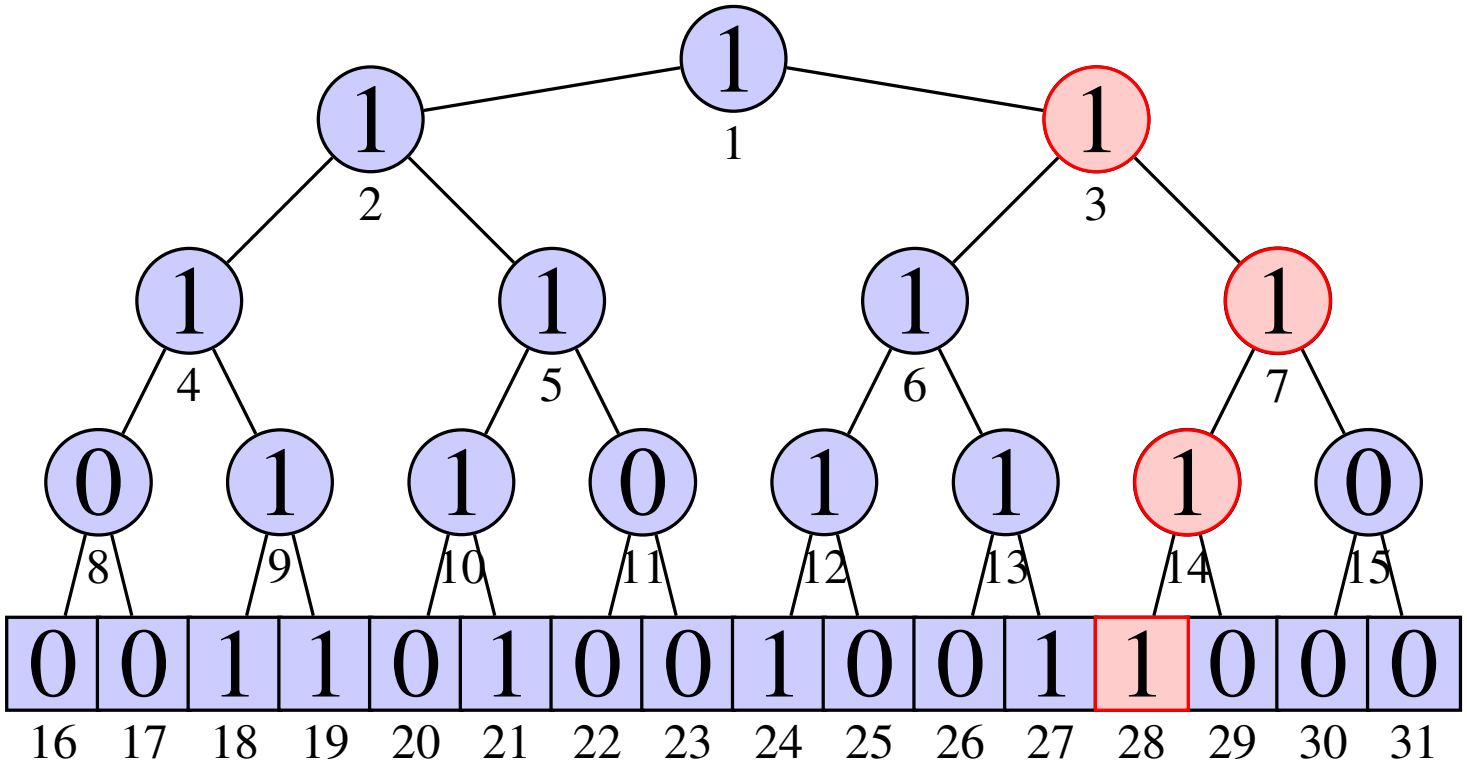
- MEMBER(V, x) では, 対応する葉にアクセスし, 値を取得する, もしくは更新するので, 時間計算量は $O(1)$.



MEMBER$(V, 7), V = \{2, 3, 5, 8, 11\}$

二分木構造 計算量 (4/4)

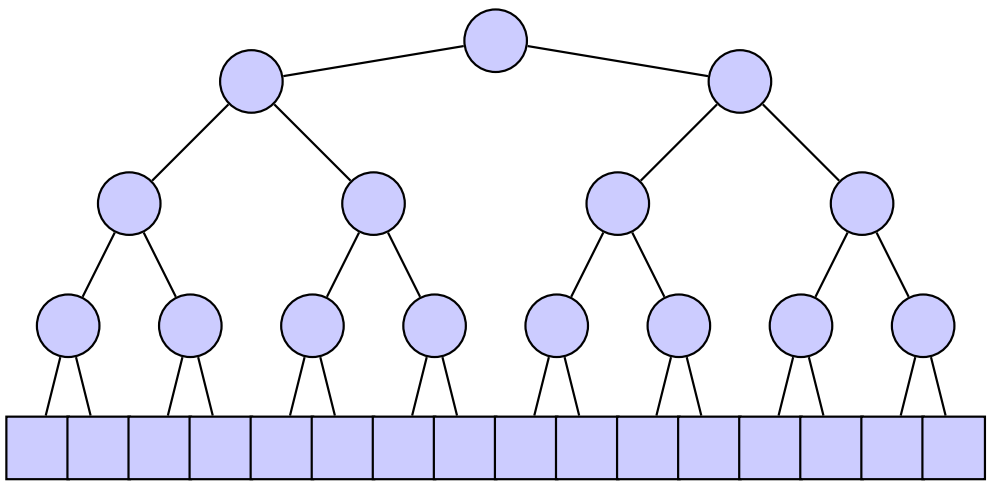
- INSERT(V, x), DELETE(V, x) では,
  葉から根まで辿る処理を行うので, 時間計算量は $\Theta(\log u)$.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

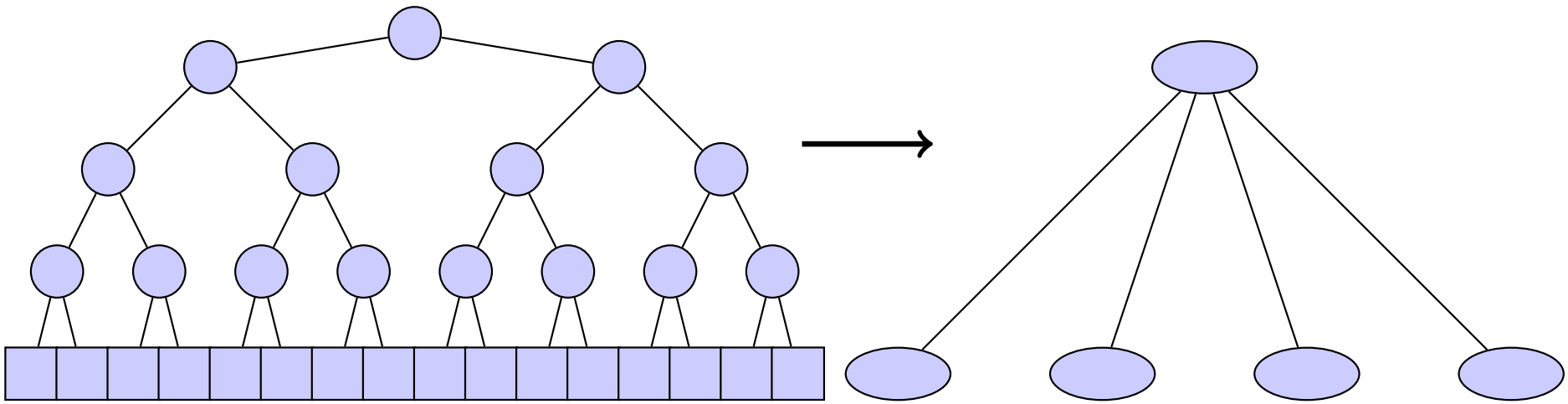
二分木構造 高速化 (1/2)

二分木構造では, 各種操作の時間計算量が $O(\log u)$ となる.



二分木構造 高速化 (2/2)

二分木構造では, 各種操作の時間計算量が $O(\log u)$ となる.
→ 時間計算量が木の高さに依存している.
頂点あたりの子を増やし, 木の高さを小さくする.



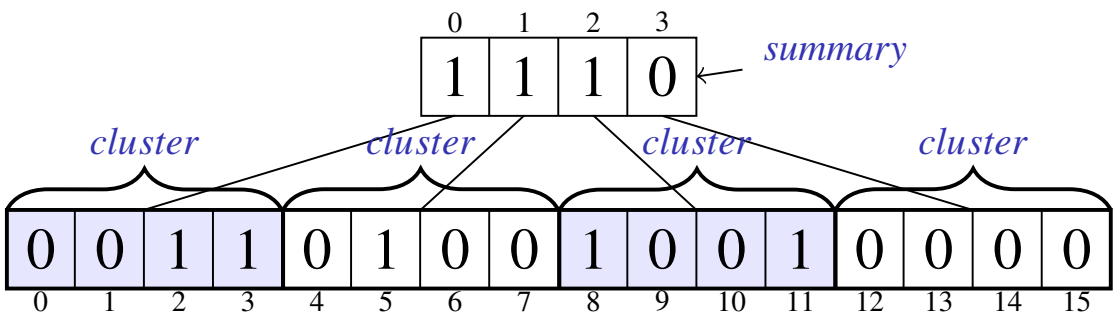
平方分割木

葉の数を $\sqrt{u}$ 個 とした平方分割木 (仮称) を考える.

平方分割木

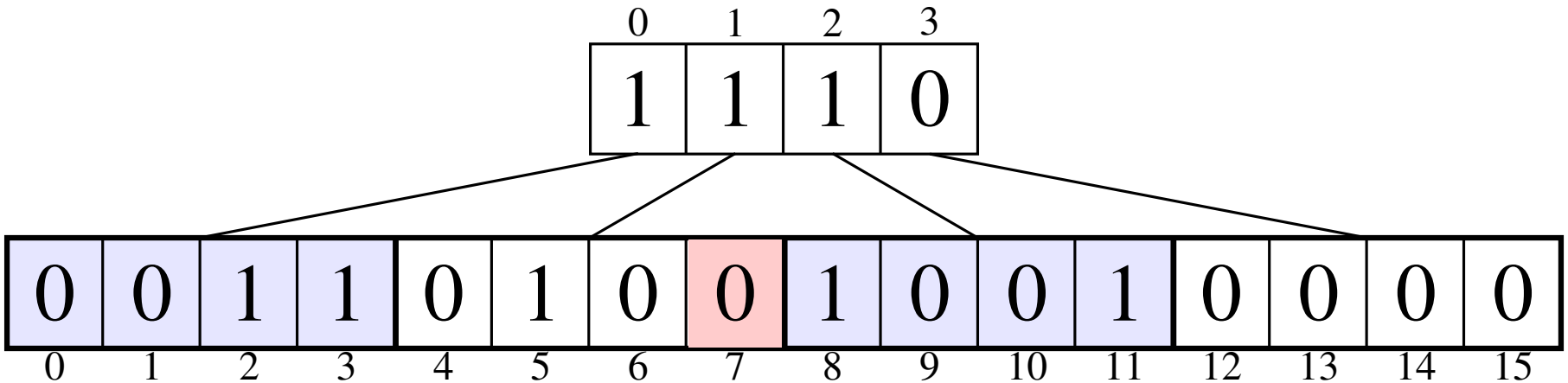
平方分割木とは, 葉を $\sqrt{u}$ 個持ち, 高さが 2 の,
以下のような特徴を持つデータ構造である.

- 要素の全体集合の大きさ u を $u = 2^{2k}$(k は非負整数) とする.
- 配列を $\sqrt{u}$ 個に分割し, それぞれを *cluster* とする.
- *cluster* は大きさが $\sqrt{u}$ の配列であり, 対応する葉の値をそれぞれ保持する.
- 根は大きさ $\sqrt{u}$ の配列を持ち (これを *summary* とする) ,
  それぞれの要素は各 *cluster* の要素の論理和を保持する.



平方分割木構造 操作: MEMBER(V, x)

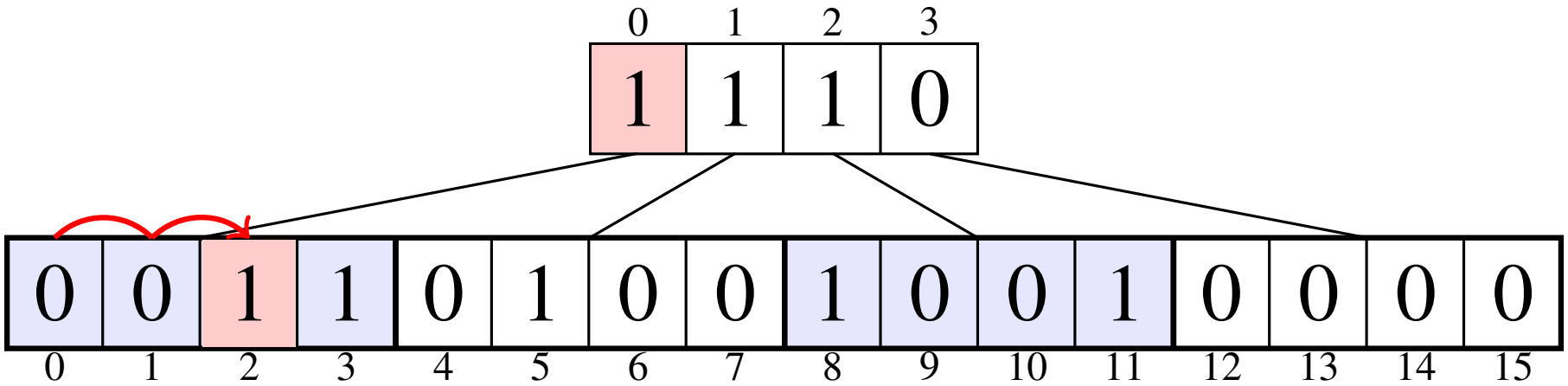
- MEMBER(V, x) は, 対応する配列の要素の値を返す.



MEMBER$(V, 7), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: Min(V), Max(V)

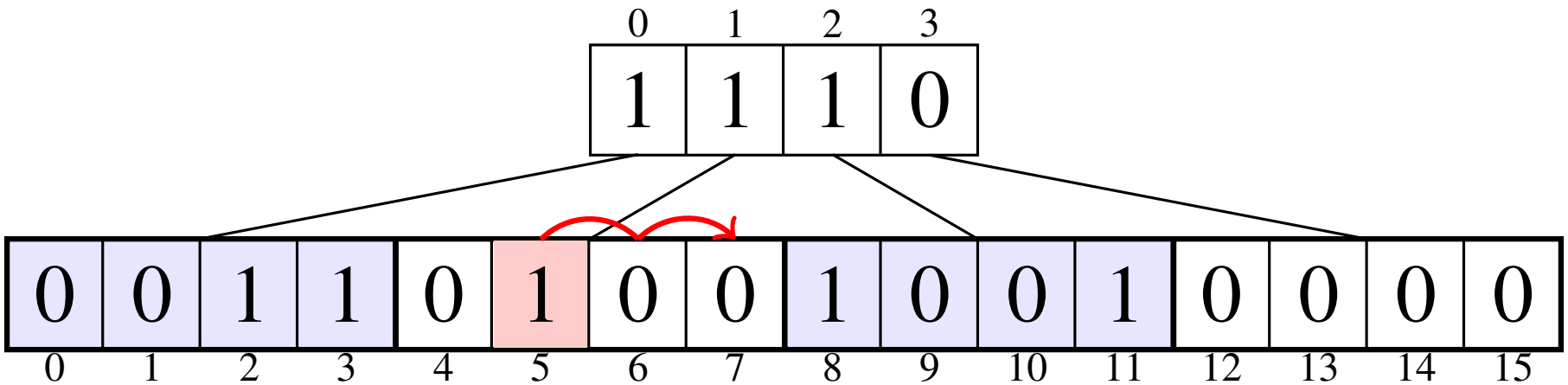
- Min(V) は, *summary* で値が 1 の左端の要素を探し, 対応する *cluster* で最小値を左端から探す.
  - Max(V) も同様に処理する.



$\text{Min}(V), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: Successor(V, x), Predecessor(V, x)

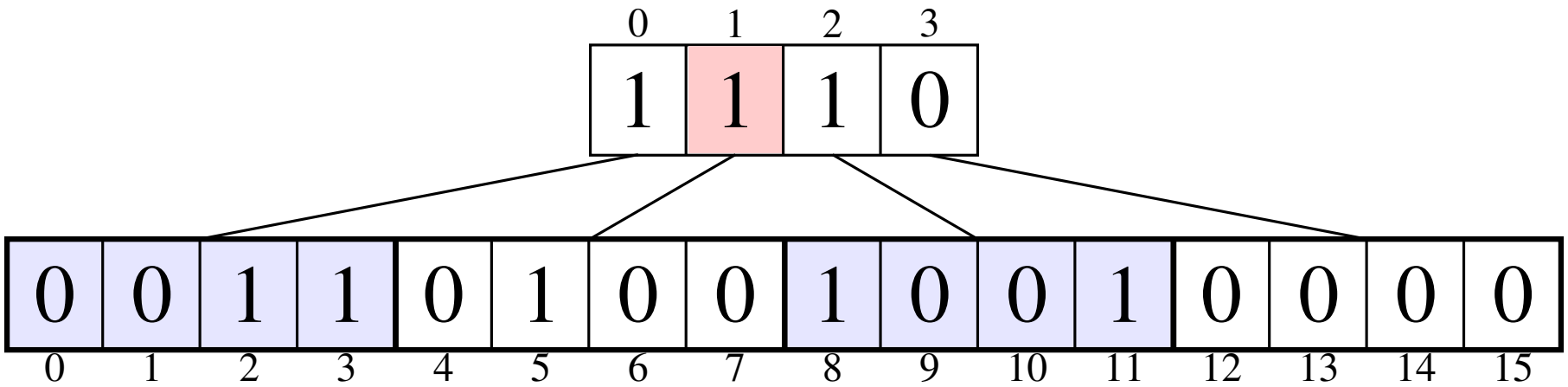
- Successor(V, x) は, x の *cluster* 内で x より左の要素を探索し, もしなければ, *summary* で $\frac{x}{\sqrt{u}}$ より右で 1 である要素を探す. その後, 1 の値である *cluster* 内の最小値を返す.
  - Predecessor(V, x) も同様に処理する.



Successor$(V, x), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: SUCCESSOR(V, x), PREDECESSOR(V, x)

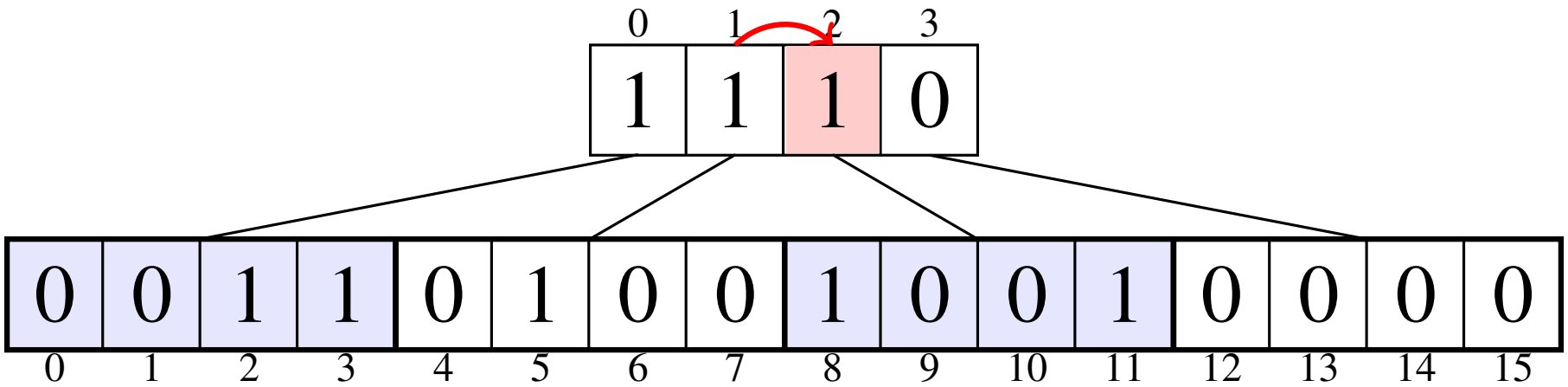
- SUCCESSOR(V, x) は, x の *cluster* 内で x より左の要素を探索し, もしなければ, *summary* で $\frac{x}{\sqrt{u}}$ より右で 1 である要素を探す. その後, 1 の値である *cluster* 内の最小値を返す.
  - PREDECESSOR(V, x) も同様に処理する.



SUCCESSOR$(V, x), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: Successor(V, x), Predecessor(V, x)

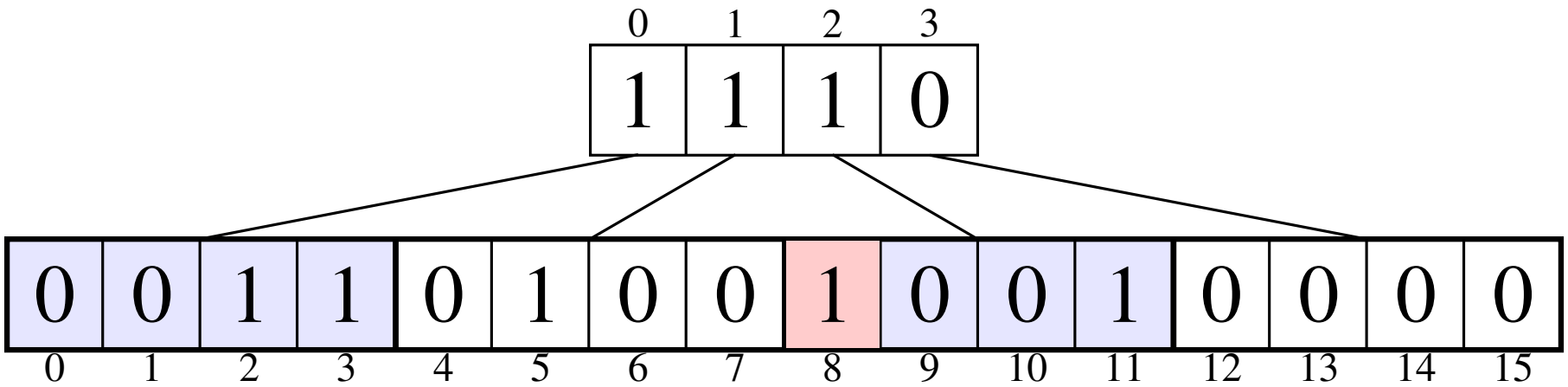
- Successor(V, x) は, x の *cluster* 内で x より左の要素を探索し, もしなければ, *summary* で $\frac{x}{\sqrt{u}}$ より右で 1 である要素を探す. その後, 1 の値である *cluster* 内の最小値を返す.
  - Predecessor(V, x) も同様に処理する.



Successor$(V, x), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: SUCCESSOR(V, x), PREDECESSOR(V, x)

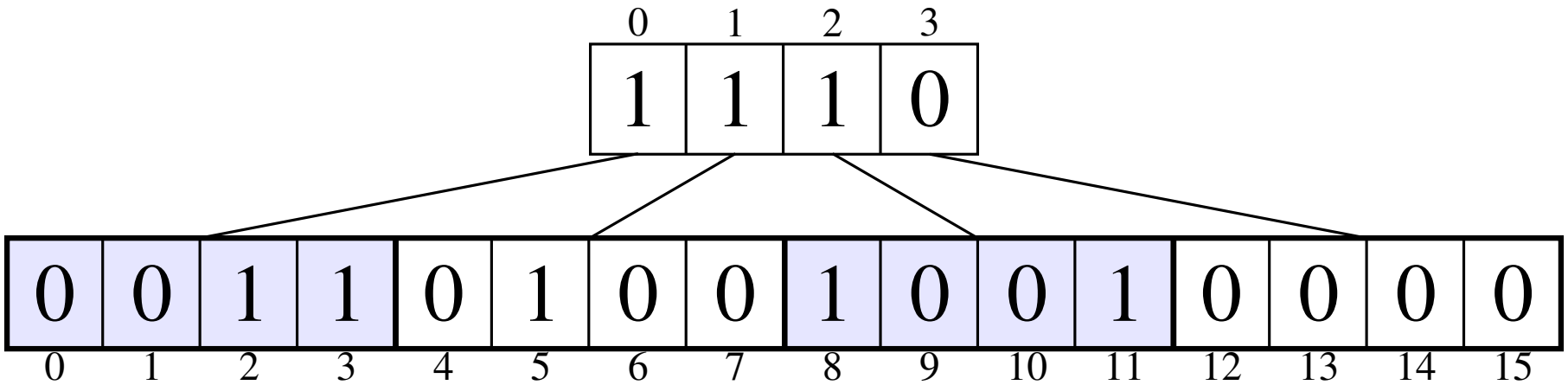
- SUCCESSOR(V, x) は, x の *cluster* 内で x より左の要素を探索し, もしなければ, *summary* で $\frac{x}{\sqrt{u}}$ より右で 1 である要素を探す. その後, 1 の値である *cluster* 内の最小値を返す.
  - PREDECESSOR(V, x) も同様に処理する.



SUCCESSOR$(V, x), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: INSERT(V, x)

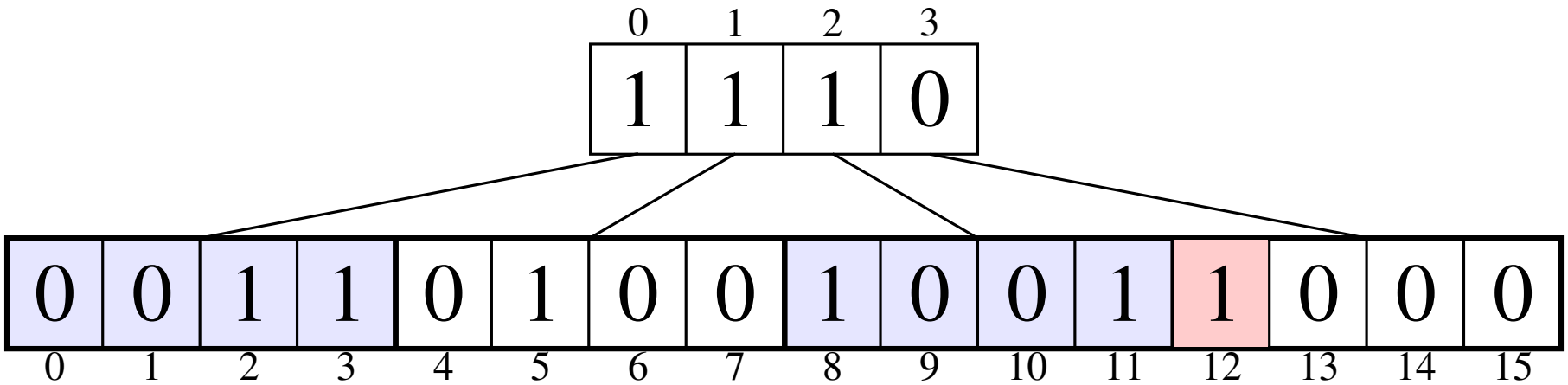
- INSERT(V, x) は, 対応する *summary*, *cluster* の要素を 1 にする.



INSERT($V, 12$), $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: INSERT(V, x)

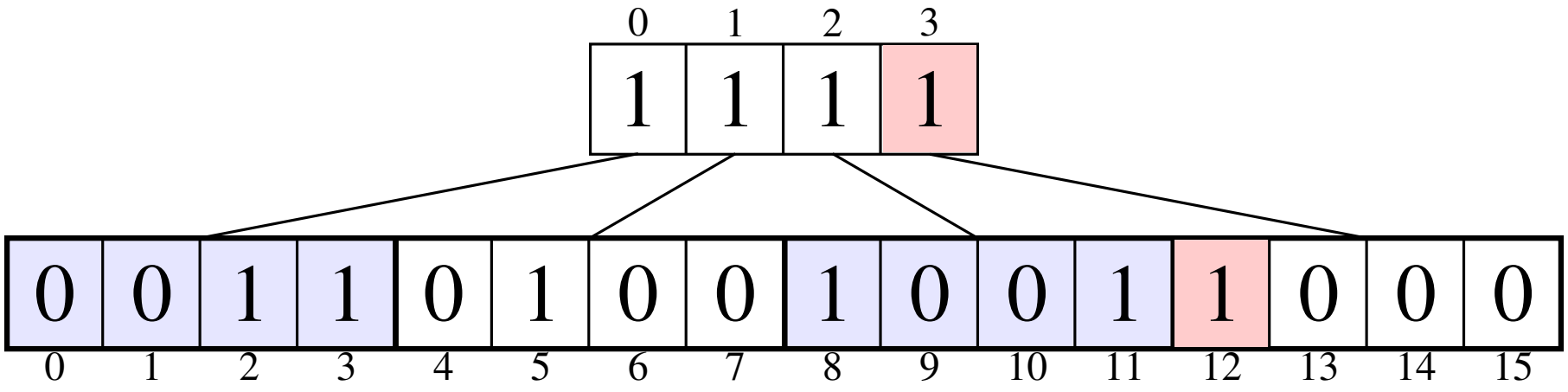
- INSERT(V, x) は, 対応する *summary*, *cluster* の要素を 1 にする.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: INSERT(V, x)

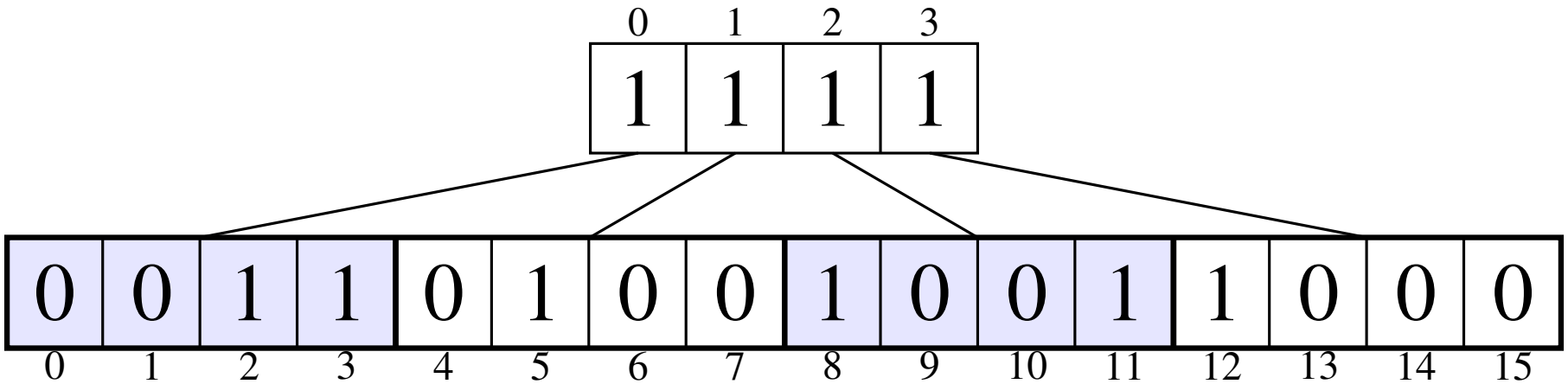
- INSERT(V, x) は, 対応する *summary*, *cluster* の要素を 1 にする.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作: DELETE(V, x)

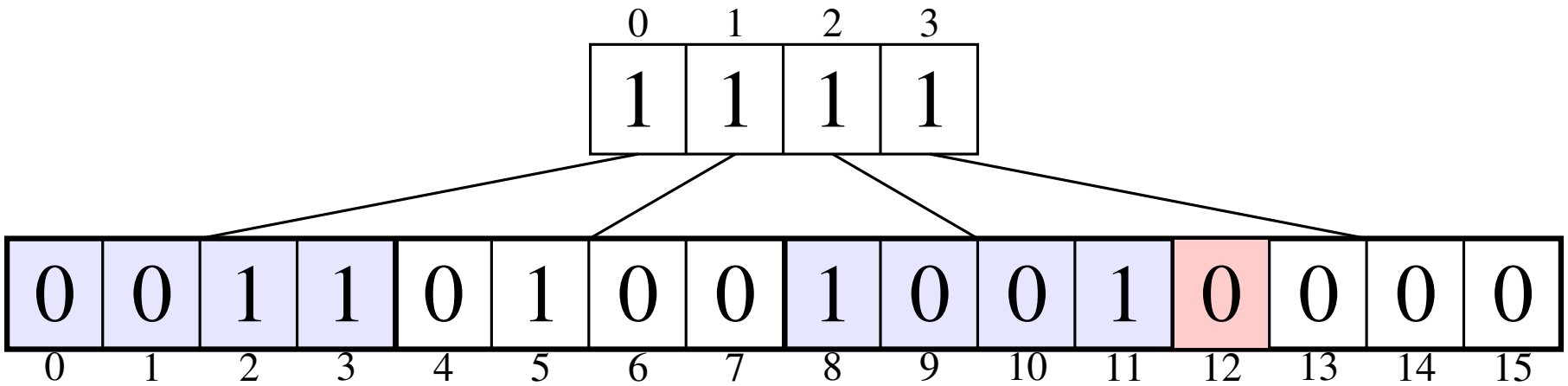
- DELETE(V, x) は 対応する *cluster* の要素の値を 0 にし, *cluster* の全要素が 0 ならば, *summary* の要素を 0 にする.



DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 操作: DELETE(V, x)

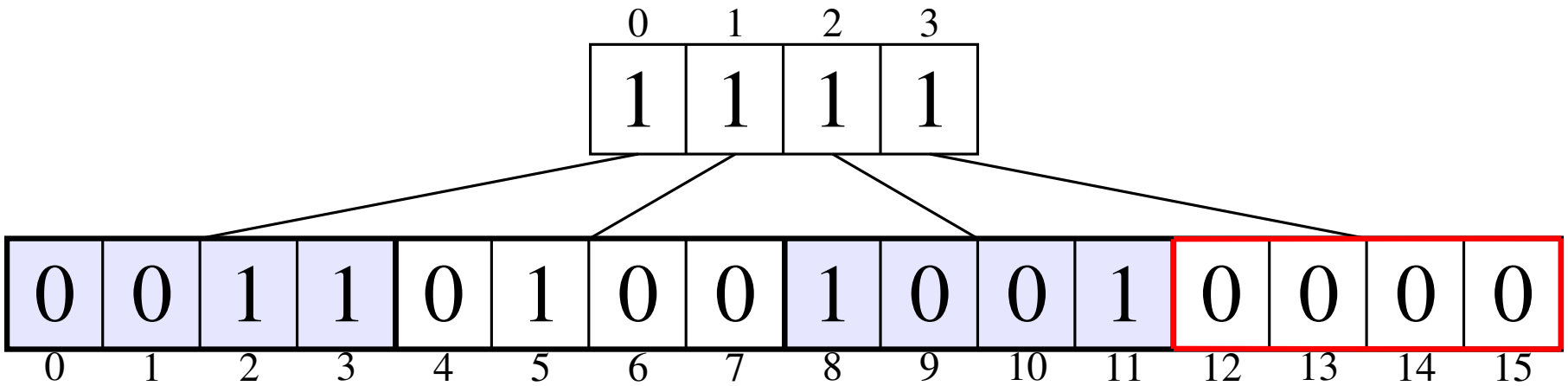
- DELETE(V, x) は 対応する *cluster* の要素の値を 0 にし, *cluster* の全要素が 0 ならば, *summary* の要素を 0 にする.



DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 操作: DELETE(V, x)

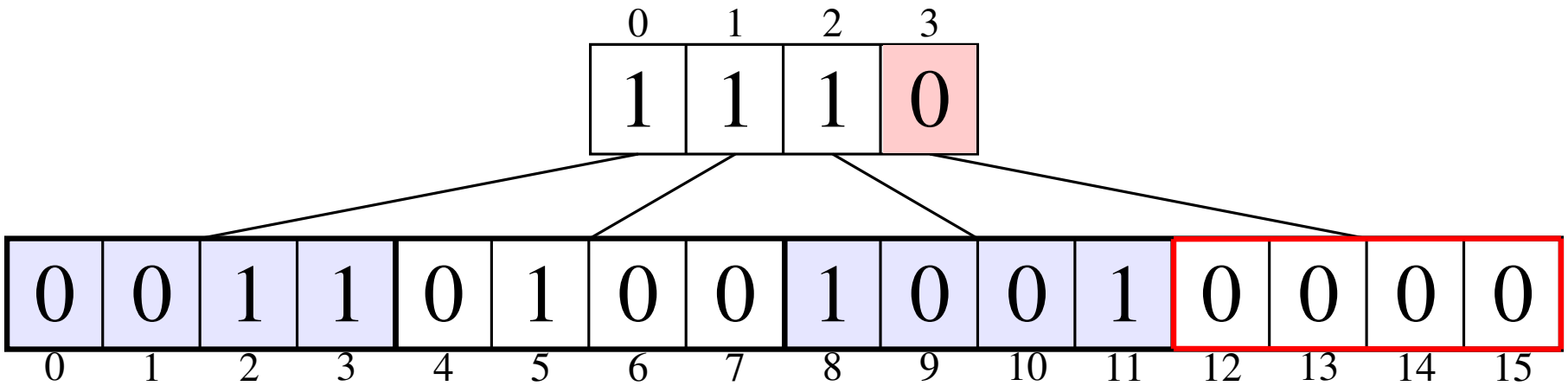
- DELETE(V, x) は 対応する *cluster* の要素の値を 0 にし, *cluster* の全要素が 0 ならば, *summary* の要素を 0 にする.



DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 操作: DELETE(V, x)

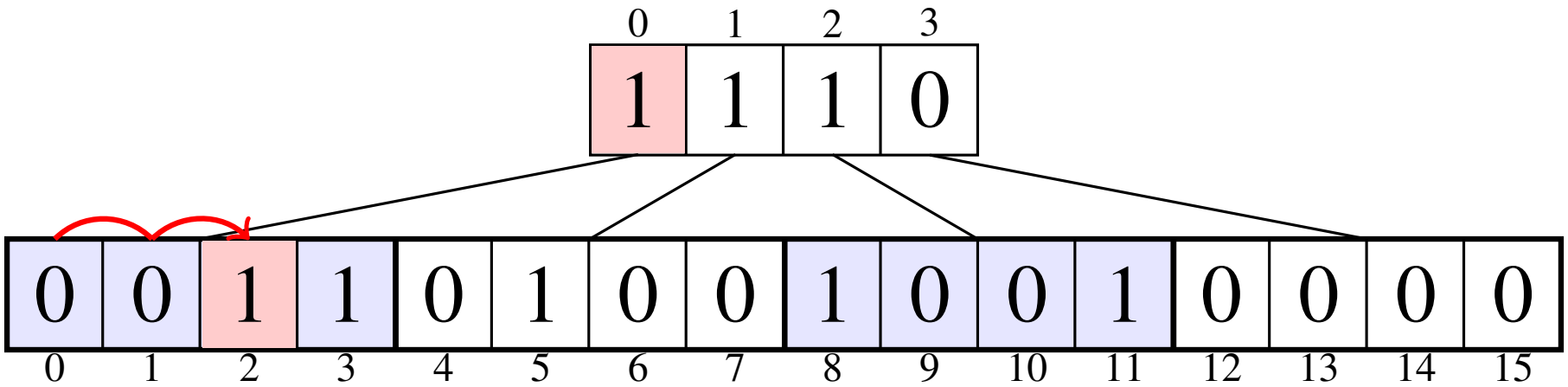
- DELETE(V, x) は 対応する *cluster* の要素の値を 0 にし, *cluster* の全要素が 0 ならば, *summary* の要素を 0 にする.



DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造: 計算量 (1/4)

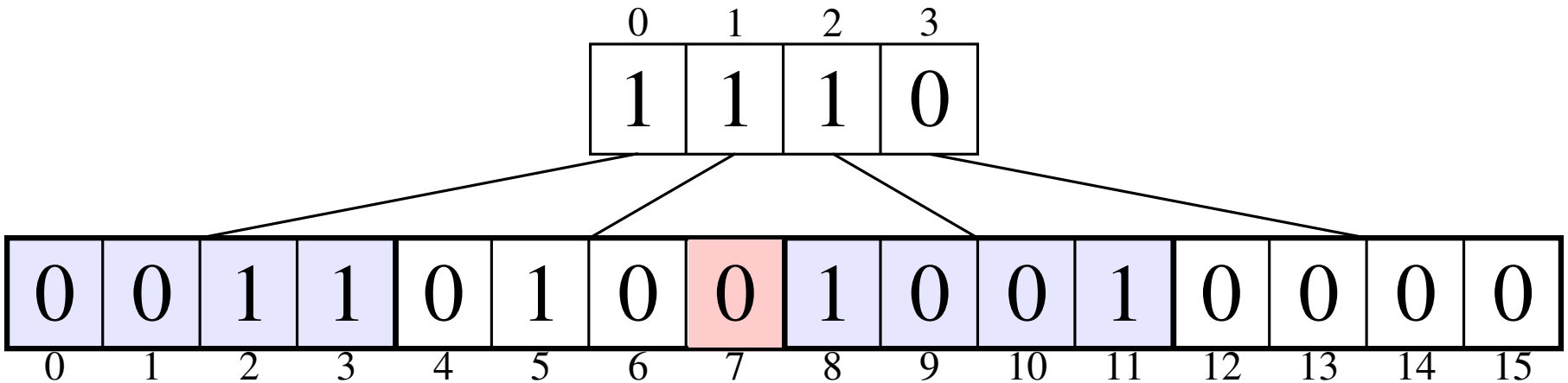
- MIN(V), MAX(V) では, *summary* と *cluster* を線形探索するので, 時間計算量は $O(\sqrt{u})$ となる.
- SUCCESSOR(V, x), PREDECESSOR(V, x) も MIN(V) と同様に 時間計算量は $O(\sqrt{u})$ となる.



$\text{MIN}(V), V = \{2, 3, 5, 8, 11\}$

平方分割木構造: 計算量 (2/4)

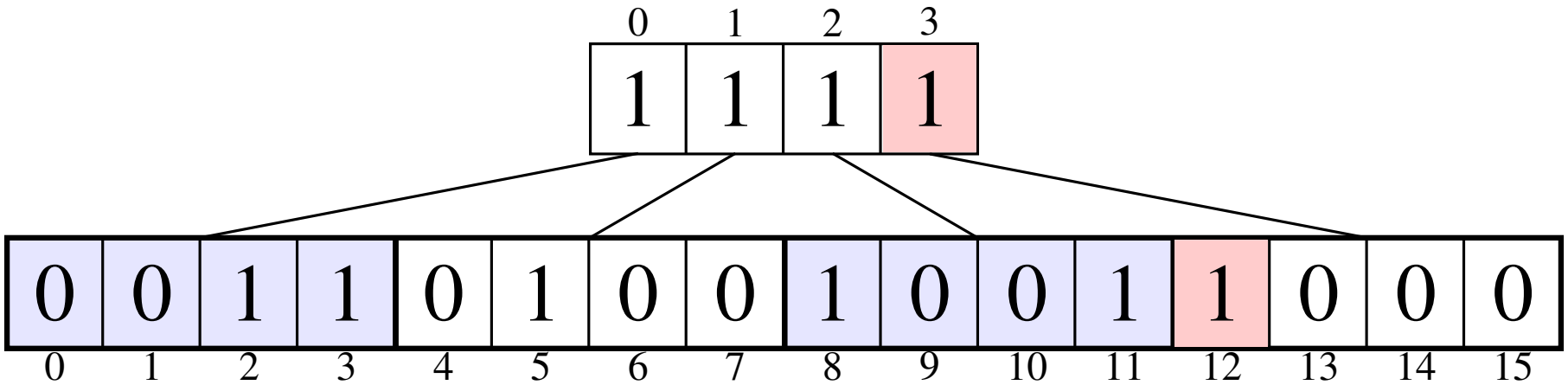
- MEMBER(V, x) では, 二分木構造と同様に, 時間計算量は $O(1)$ となる.



MEMBER$(V, 7), V = \{2, 3, 5, 8, 11\}$

平方分割木構造: 計算量 (3/4)

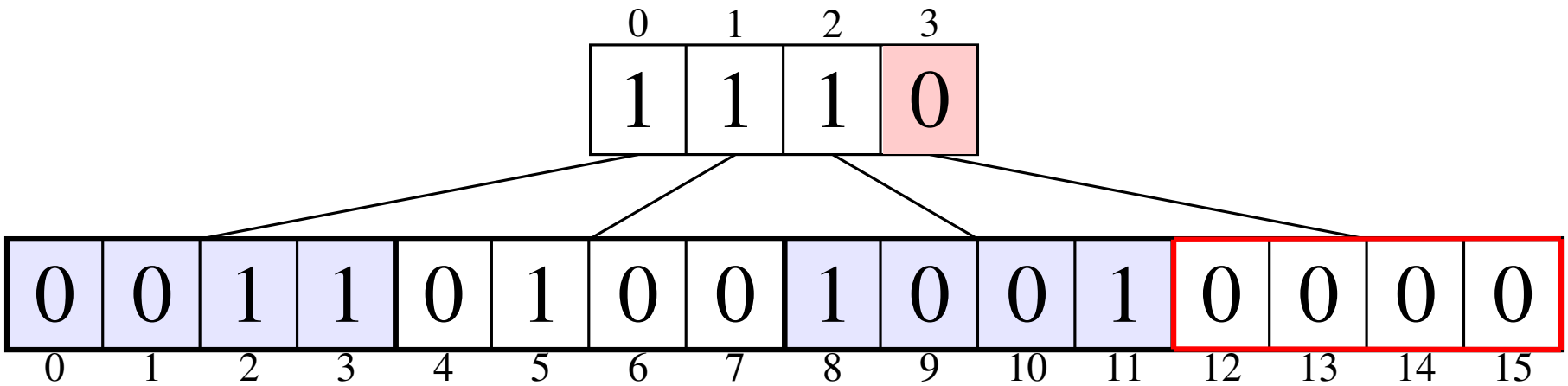
- INSERT(V, x) では, 対応する *cluster*, *summary* の要素を更新すればよいので, 時間計算量は $O(1)$ となる.



INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

平方分割木構造: 計算量 (4/4)

- DELETE(V, x) では, 対応する *cluster* を線形探索するので, 時間計算量は $O(\sqrt{u})$ となる.



DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

前半のまとめ

- vEB 木を定義した.
- 二分木構造で各種操作の時間計算量は下表となった.
- 平方分割木によって二分木構造の木の高さを小さくしたが, 一部操作の最悪時間計算量が $O(\sqrt{u})$ と悪化
  - *summary*, *cluster* の線形探索がボトルネック

	二分木	平方分割木
MIN(V)	$\Theta(\log u)$	$O(\sqrt{u})$
MAX(V)	$\Theta(\log u)$	$O(\sqrt{u})$
MEMBER(V, x)	$O(1)$	$O(1)$
SUCCESSOR(V, x)	$O(\log u)$	$O(\sqrt{u})$
PREDECESSOR(V, x)	$O(\log u)$	$O(\sqrt{u})$
INSERT(V, x)	$O(\log u)$	$O(1)$
DELETE(V, x)	$O(\log u)$	$O(\sqrt{u})$