van Emde Boas Trees

光吉　健汰

北海道大学工学部 情報エレクトロニクス学科 情報理工学コース 4 年
情報知識ネットワーク研究室

June 17, 2019

Contents

Contents

こんにちは

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 (以下 vEB 木) は動的集合を扱うデータ構造

操作

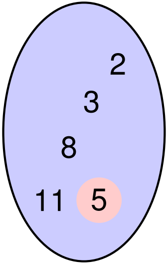
MEMBER(V, x)	V に x が存在するかを返す
MIN(V)	V の要素の最小値を返す
MAX(V)	V の要素の最大値を返す
SUCCESSOR(V, x)	V の x より大きい最小の要素を返す
PREDECESSOR(V, x)	V の x より小さい最大の要素を返す
INSERT(V, x)	V に x を挿入する
DELETE(V, x)	V から x を削除する

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 $\textsc{Member}(V, x)$

- $\textsc{Member}(V, x)$ は, x が集合 V に存在するかを真偽値で返す.

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



- $\textsc{Member}(V, 5) = \textsf{TRUE}$
- $\textsc{Member}(V, 6) = \textsf{FALSE}$

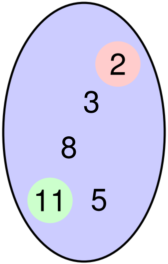
各関数の引数

関数に渡す引数は要素として取りうる値のみとする

操作 $\textsc{Min}(V)$, $\textsc{Max}(V)$

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

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

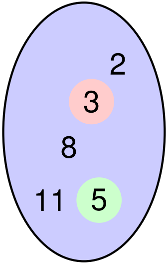


- $\textsc{Min}(V) = \boxed{2}$
- $\textsc{Max}(V) = \boxed{11}$

操作 Successor(V, x), Predecessor(V, x)

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

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



- Successor$(V, 2) = \boxed{3}$
- Predecessor$(V, 7) = \boxed{5}$

操作 Insert(V, x)

- Insert(V, x) は, 集合 V に x を挿入する.

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

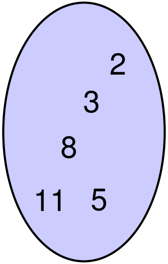
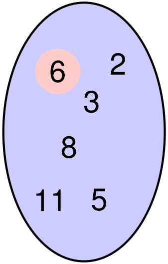


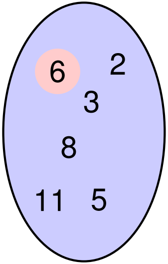
Figure: $V = \{2, 3, 5, \mathbf{6}, 8, 11\}$



操作 Delete(V, x)

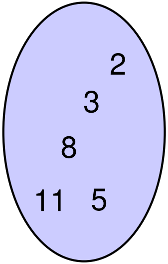
- Delete(V, x) は, 集合 V から x を削除する.

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



$\longrightarrow$

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



Contents

直接アドレス法 (1/2)

空間計算量 $O(u)$ で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

今回考えているデータ構造では付属データは持たないので,
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.

0	0	1	1	0	1	0	0	1	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

直接アドレス法 (2/2)

空間計算量 $O(u)$ で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.

- Member(V, x), Insert(V, x), Delete(V, x) の処理は, 配列のランダムアクセスが定数時間であるので, 時間計算量 $O(1)$.
- Successor(V, x) の処理は, x の次の値から bit が立っている要素まで
  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 $O(u)$.

0	0	1	1	0	1	0	0	1	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

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

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

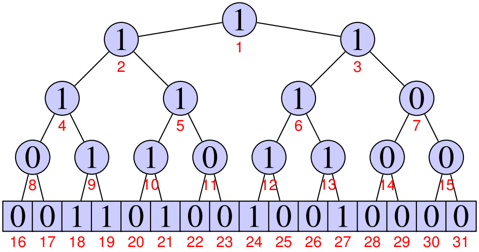
二分木とは, 葉ではない頂点の子が常に 2 個であるような木

- **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)

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.

- 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 bit が立っている場合,
  その頂点の下にある葉の少なくとも 1 頂点は要素が 1.
- 根から深さ毎に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 (後述の操作のために 1-*indexed* とする)

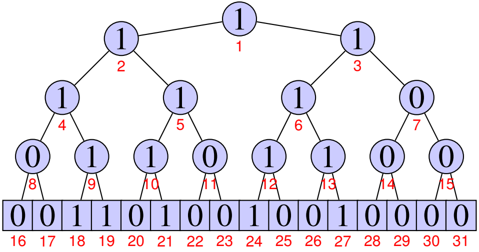


二分木構造 (3/3)

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: $2 \times i + 1$ と $2 \times i + 2$
親: $\lfloor \frac{i}{2} \rfloor$ (以後 $\frac{r}{m}$ が整数でない場合これを切り捨てた値とする)
葉: 集合 V 中の要素 v に対応する葉の添字は $i + u$



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.

Algorithm 1 Min(V)

Require:
    V := 二分木を格納した配列
    u := 全体集合の大きさ

function Min(V)
    $i \leftarrow 1$
    while $i < u$ **do**
        if V$[2 \times i + 1]$ = 1 **then**
            $i \leftarrow 2 \times i + 1$
        else if V$[2 \times i + 2]$ = 1 **then**
            $i \leftarrow 2 \times i + 2$
        else
            return nil
    return $i - u$
end function

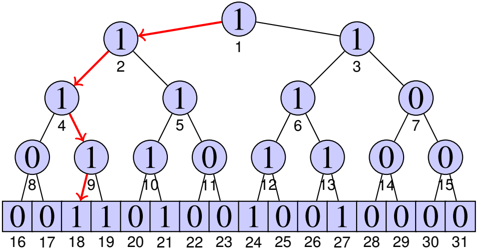


Figure: Min(V), $V = \{2, 3, 5, 8, 11\}$

二分木構造 操作 $\textsc{Successor}(V,x)$, $\textsc{Predecessor}(V,x)$ (1/2)

- $\textsc{Successor}(V,x)$ は, x の親から, 自身が直前の頂点が左の子かつ右の子の bit が立っている頂点になるまで親を辿り, 右の子の最小値を返す.
  - $\textsc{Predecessor}(V,x)$ も同様に処理する.

Algorithm 2 $\textsc{Successor}(V,x)$

```
function SUCCESSOR(V, x)
    i ← (x + u)/2
    prev ← x + u
    while v ≠ root do
        if V[i] = 1 and prev ≠ ODD
          and V[2 × i + 2] = 1      then
            MIN(V) と同様の処理
        else
            prev = v
            v = v.parent
    return NIL
end function
```

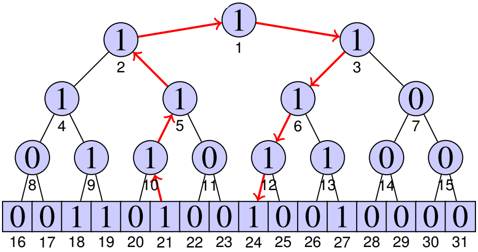


Figure: $\textsc{Successor}(V,5)$, $V = \{2,3,5,8,11\}$

- Successor(V, x) は, x の親から, 自身が直前の頂点が左の子かつ右の子の bit が立っている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.

Algorithm 3 Successor(V, x)

```
function Successor(V, x)
    i ← (x + u)/2
    prev ← x + u
    while v ≠ root do
        if V[i] = 1 and prev ≠ ODD
           and V[2 × i + 2] = 1     then
            Min(V) と同様の処理
        else
            prev = v
            v = v.parent
    return NIL
end function
```

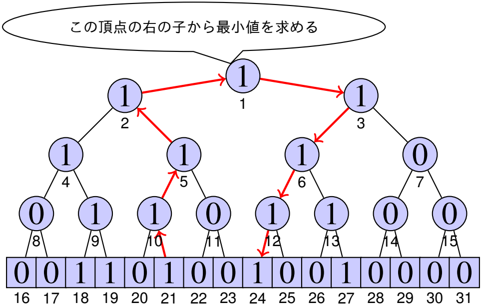


Figure: Successor$(V, 5)$, $V = \{2, 3, 5, 8, 11\}$

二分木構造 操作 $\textsc{Member}(V, x)$

- $\textsc{Member}(V, x)$ は, 対応する葉の値を返す.

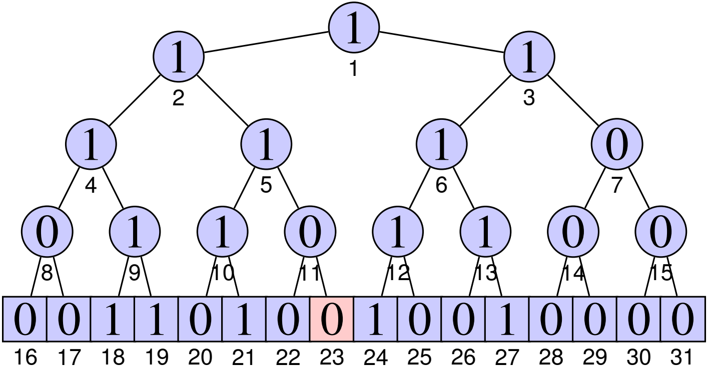


Figure: $\textsc{Member}(V, 7)$, $V = \{2, 3, 5, 8, 11\}$

二分木構造 操作 INSERT(V, x) (1/6)

- INSERT(V, x) は, 対応する葉から親の値を 1 にしながら根まで辿る.

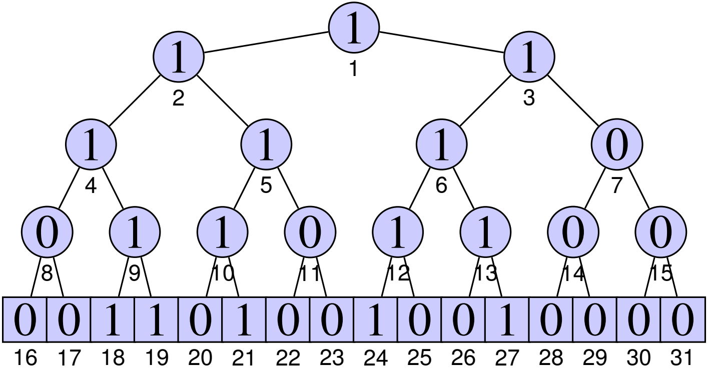


Figure: INSERT$(V, 12), V = \{2, 3, 5, 8, 11\}$

- Insert(V, x) は, 対応する葉から親の値を 1 にしながら根まで辿る.

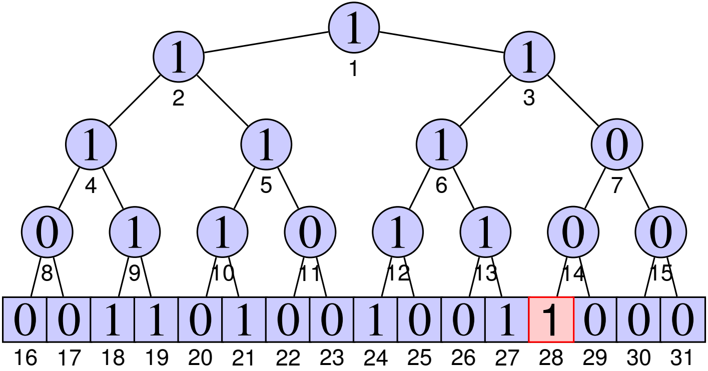


Figure: Insert$(V, 12)$, $V = \{2, 3, 5, 8, 11\}$

- Insert(V, x) は, 対応する葉から親の値を 1 にしながら根まで辿る.

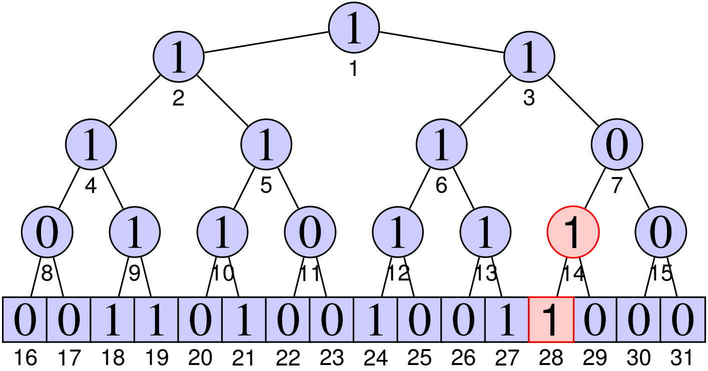


Figure: Insert$(V, 12), V = \{2, 3, 5, 8, 11\}$

- Insert(V, x) は, 対応する葉から親の値を 1 にしながら根まで辿る.

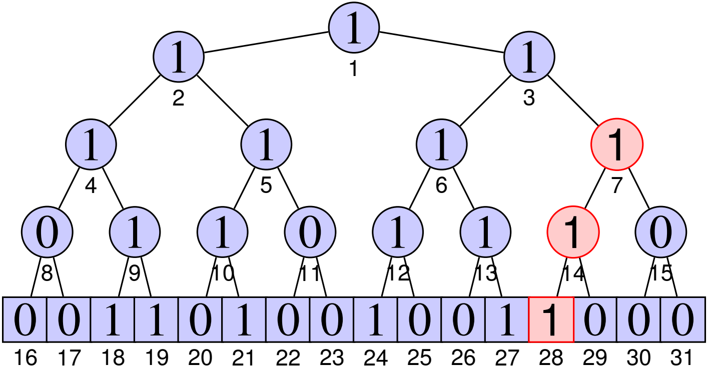


Figure: Insert($V, 12$), $V = \{2, 3, 5, 8, 11\}$

- Insert(V, x) は, 対応する葉から親の値を 1 にしながら根まで辿る.

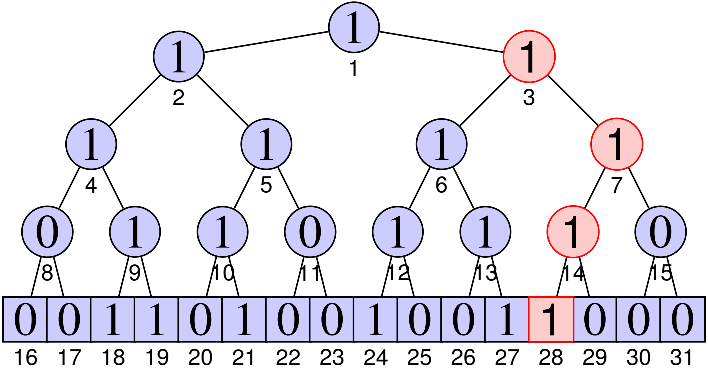


Figure: Insert(V, 12), $V = \{2, 3, 5, 8, 11\}$

二分木構造 操作 $\textsc{Insert}(V, x)$ (6/6)

- $\textsc{Insert}(V, x)$ は, 対応する葉から親の値を 1 にしながら根まで辿る.

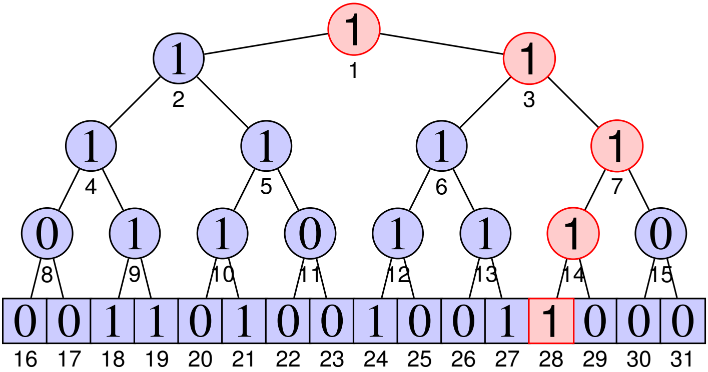


Figure: $\textsc{Insert}(V, 12), V = \{2, 3, 5, 8, 11\}$

二分木構造 操作 Delete(V, x) (1/5)

- Delete(V, x) は, 対応する葉の値を 0 にし親の頂点に遷移し, 以下の操作を繰り返す.
  - 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.

Algorithm 4 Delete(V, x)

---

function Delete(V, x)
    $i \leftarrow (x + u)/2$
    while $V[i]$ の子の値が両方とも 0 **do**
        $V[i] \leftarrow 0$
        $i \leftarrow \frac{i}{2}$
end function

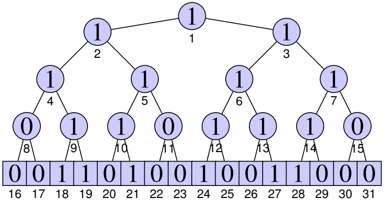


Figure: Delete$(V, 12)$, $V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作 DELETE(V, x) (2/5)

- DELETE(V, x) は, 対応する葉の値を 0 にし親の頂点に遷移し, 以下の操作を繰り返す.
    - 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.

Algorithm 5 DELETE(V, x)

```
function DELETE(V, x)
    i ← (x + u)/2
    while V[i] の子の値が両方とも 0 do
        V[i] ← 0
        i ← i/2
end function
```

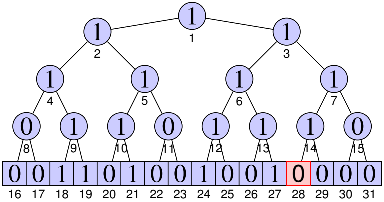


Figure:
DELETE$(V, 12)$, $V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作 $\textsc{Delete}(V, x)$ (3/5)

- $\textsc{Delete}(V, x)$ は, 対応する葉の値を 0 にし親の頂点に遷移し, 以下の操作を繰り返す.
  - 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.

Algorithm 6 $\textsc{Delete}(V, x)$

```
function Delete(V, x)
    i ← (x + u)/2
    while V[i] の子の値が両方とも 0 do
        V[i] ← 0
        i ← i/2
end function
```

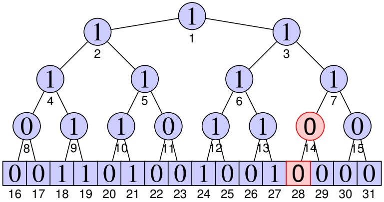


Figure: $\textsc{Delete}(V, 12)$, $V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作 $\textsc{Delete}(V, x)$ (4/5)

- $\textsc{Delete}(V, x)$ は, 対応する葉の値を 0 にし親の頂点に遷移し, 以下の操作を繰り返す.
    - 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.

Algorithm 7 $\textsc{Delete}(V, x)$

```
function Delete(V, x)
    i ← (x + u)/2
    while V[i] の子の値が両方とも 0 do
        V[i] ← 0
        i ← i/2
end function
```

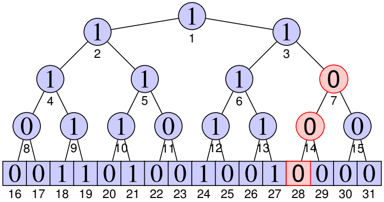


Figure: $\textsc{Delete}(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

二分木構造 操作 $\textsc{Delete}(V, x)$ (5/5)

- $\textsc{Delete}(V, x)$ は, 対応する葉の値を 0 にし親の頂点に遷移し, 以下の操作を繰り返す.
    - 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.

Algorithm 8 $\textsc{Delete}(V, x)$

---

function $\textsc{Delete}(V, x)$
  $i \leftarrow (x + u)/2$
  while $V[i]$ の子の値が両方とも 0 **do**
    $V[i] \leftarrow 0$
    $i \leftarrow \frac{i}{2}$
end function

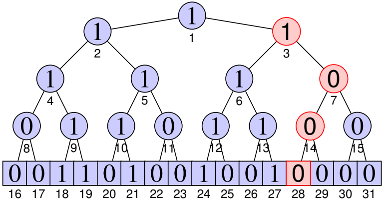


Figure: $\textsc{Delete}(V, 12), V = \{2, 3, 5, 6, 8, 11, 12\}$

- $\textsc{Min}(V)$, $\textsc{Max}(V)$ 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 $\Theta(\log u)$ となる.

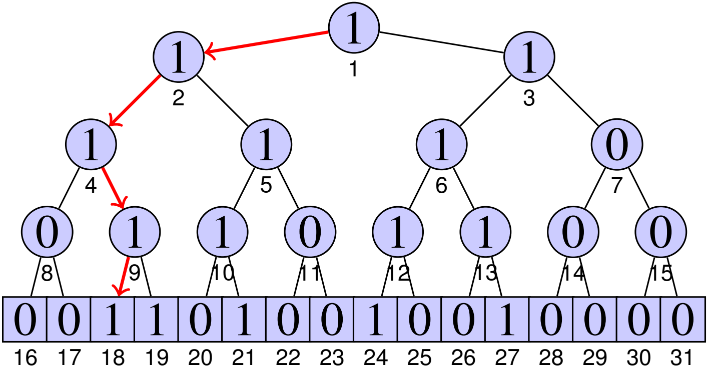


Figure: $\textsc{Min}(V)$, $V = \{2, 3, 5, 8, 11\}$

二分木構造 計算量 (2/3)

- 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 $O(\log u)$ となる.

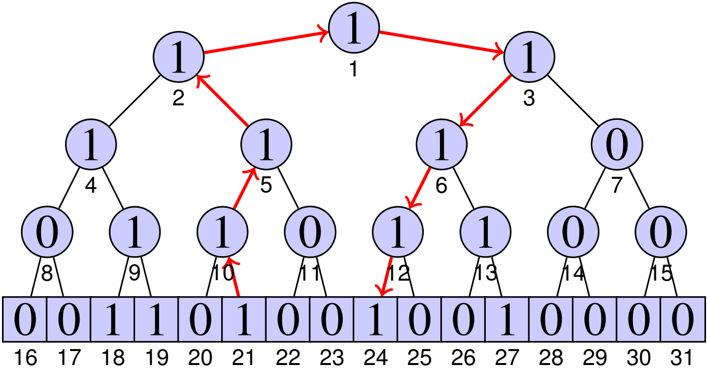


Figure: SUCCESSOR$(V, 5)$, $V = \{2, 3, 5, 8, 11\}$

- $\textsc{Member}(V, x)$ 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 $O(1)$ となる.

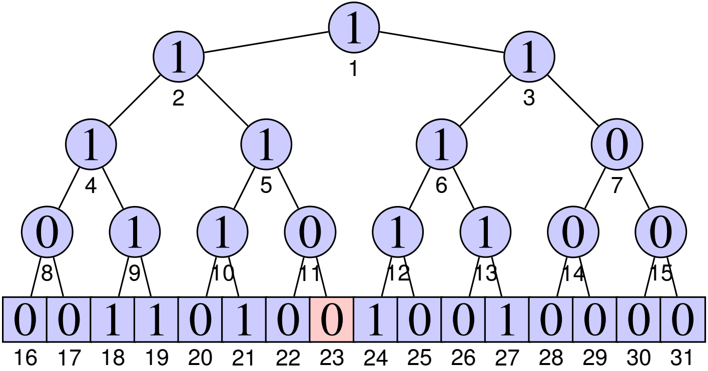


Figure: $\textsc{Member}(V, 7), V = \{2, 3, 5, 8, 11\}$

- 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 $O(\log u)$ となる.

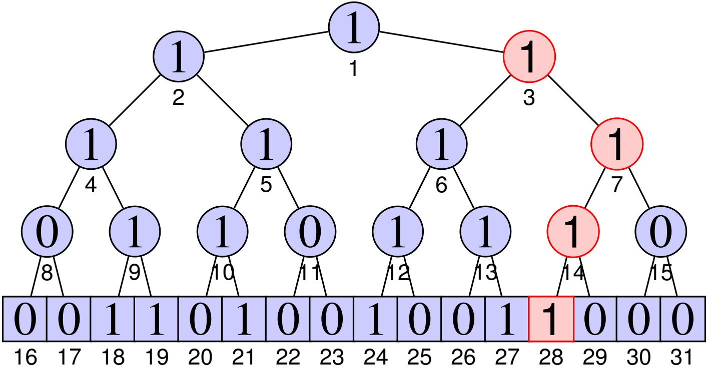
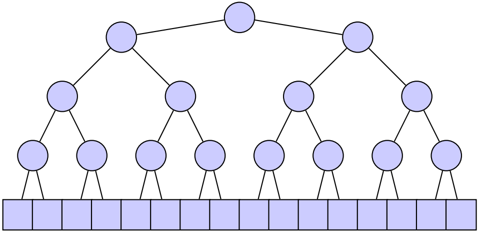


Figure: INSERT$(V, 12)$, $V = \{2, 3, 5, 8, 11\}$

二分木構造 高速化 (1/2)

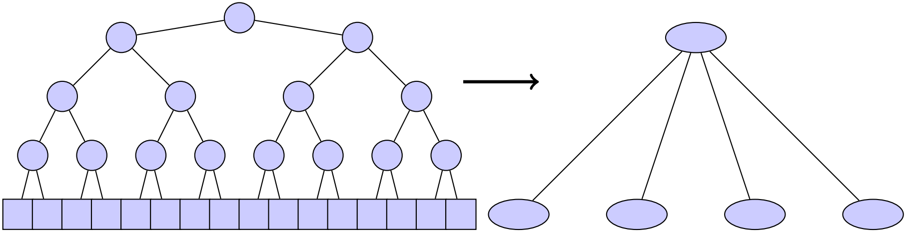
二分木構造では, 各種操作の時間計算量が $O(\log u)$ となる.



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.
$\rightarrow$ 木の高さに時間計算量が依存している.

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.
- 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.

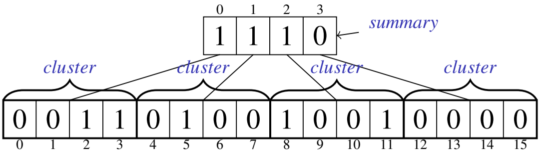


Figure: $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 Min(V), Max(V)

Algorithm 9 Min(V)

Require:
A := 葉の各要素に対応した配列
($\sqrt{u}$ 個に分割したものが *cluster*)

```
function Min(V)
    i ← 0
    while i < √u
and summary[i] ≠ 1 do
        i ← i + 1
    j ← 0
    while j < √u
and A[i × √u + j] ≠ 1 do
        j ← j + 1
end function
```

- 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.

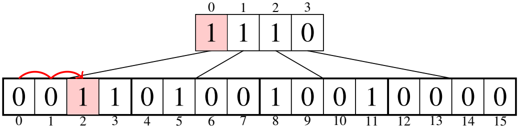


Figure: Min(), $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 Min(V), Max(V)

Algorithm 10

Successor(V, x)

function Successor(V, x)
  $i \leftarrow x$
  while $i < (1 + \frac{x}{\sqrt{u}}) \times \sqrt{u}$ **do**
    if $A[i] = 1$ **then**
      return i
  $j \leftarrow \frac{x}{\sqrt{u}}$
  while $summary[j] \neq 1$ *and* $j < \sqrt{u}$ **do**
    $j \leftarrow j + 1$
  $i \leftarrow j \times \sqrt{u}$
  while $i < (1 + j) \times \sqrt{u}$ **do**
    if $A[i] = 1$ **then**
      return i
  return NIL
end function

- 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.

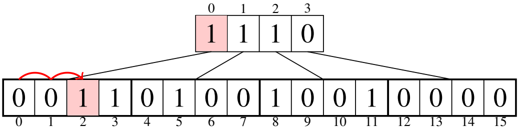


Figure: Min(), $V = \{2, 3, 5, 8, 11\}$

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.

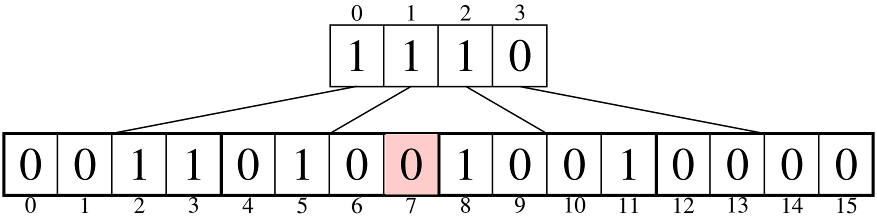


Figure: Member($V, 7$), $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 Insert(V, x) (1/3)

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

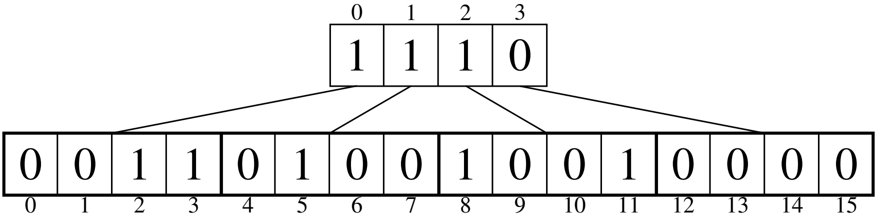


Figure: Insert(V, 12), $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 INSERT(V, x) (2/3)

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

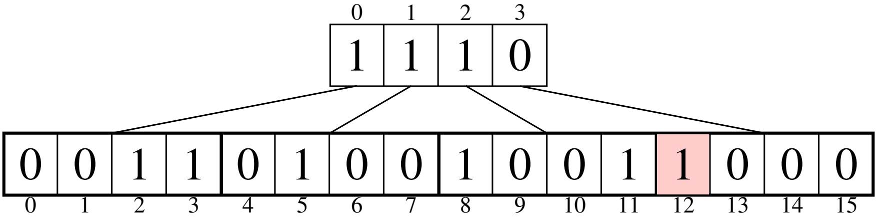


Figure: INSERT$(V, 12)$, $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 $\textsc{Insert}(V, x)$ (3/3)

- $\textsc{Insert}(V, x)$ は, 対応する *summary* の要素と *cluster* の要素を 1 にする.

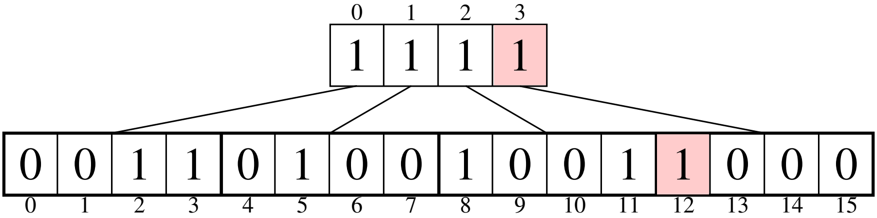


Figure: $\textsc{Insert}(V, 12), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 操作 DELETE(V, x) (1/4)

Algorithm 11 DELETE(V, x)

```
function DELETE(V, x)
    A[i] ← 0
    if i 番目の cluster の全ての要素の値が 0 then
        summary[i/√u] ← 0
end function
```

- DELETE(V, x) は 対応する *cluster* の要素の値を 1 にし, *cluster* の全要素が 0 ならば *summary* の要素を 0 にする.

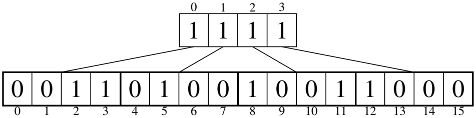


Figure:
DELETE(V, 12), $V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 操作 DELETE(V, x) (2/4)

Algorithm 12 DELETE(V, x)

```
function DELETE(V, x)
    A[i] ← 0
    if i 番目の cluster の全ての要素の値が 0 then
        summary[i/√u] ← 0
end function
```

- DELETE(V, x) は 対応する *cluster* の要素の値を 1 にし, *cluster* の全要素が 0 ならば *summary* の要素を 0 にする.

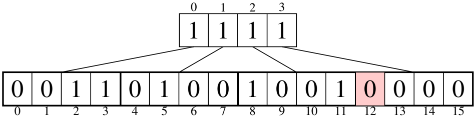


Figure:
DELETE$(V, 12)$, $V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 操作 DELETE(V, x) (3/4)

Algorithm 13 DELETE(V, x)

```
function DELETE(V, x)
    A[i] ← 0
    if i 番目の cluster の全ての要素の値が 0 then
        summary[i/√u] ← 0
end function
```

- DELETE(V, x) は 対応する *cluster* の要素の値を 1 にし, *cluster* の全要素が 0 ならば *summary* の要素を 0 にする.

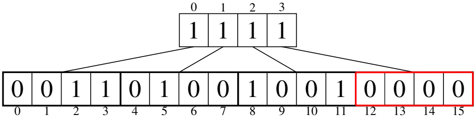


Figure:
DELETE$(V, 12)$, $V = \{2, 3, 5, 8, 11, 12\}$

Algorithm 14 DELETE(V, x)

function DELETE(V, x)
    $A[i] \leftarrow 0$
    if i 番目の *cluster* の全ての要素の値が 0 **then**
        $summary[\frac{i}{\sqrt{u}}] \leftarrow 0$
end function

- DELETE(V, x) は 対応する *cluster* の要素の値を 1 にし, *cluster* の全要素が 0 ならば *summary* の要素を 0 にする.

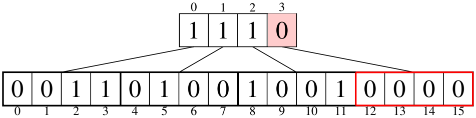


Figure:
DELETE$(V, 12)$, $V = \{2, 3, 5, 8, 11, 12\}$

平方分割木構造 計算量 (1/4)

- $\textsc{Min}(V)$, $\textsc{Max}(V)$ では, *summary* と, *cluster*1 つを線形探索するので, 時間計算量は $O(\sqrt{u})$ となる.
- $\textsc{Successor}(V, x)$, $\textsc{Predecessor}(V, x)$ も $\textsc{Min}(V)$ と同様に時間計算量は $O(\sqrt{u})$ となる.

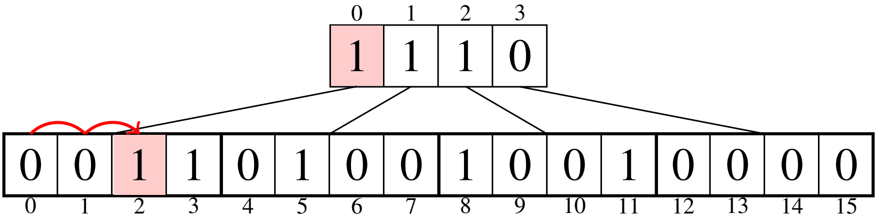


Figure: $\textsc{Min}()$, $V = \{2, 3, 5, 8, 11\}$

平方分割木構造 計算量 (2/4)

- $\textsc{Member}(V, x)$ では,
  対応する *cluster* の要素を取得すればよいので, 時間計算量は
  $O(1)$ となる.

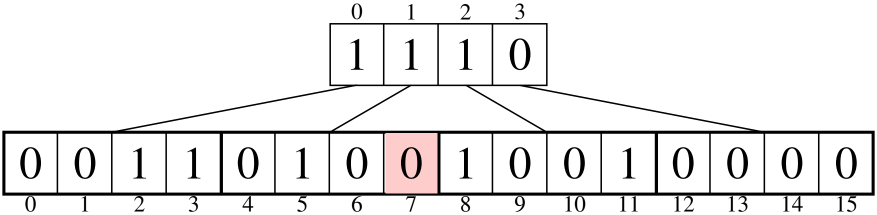


Figure: $\textsc{Member}(V, 7), V = \{2, 3, 5, 8, 11\}$

平方分割木構造 計算量 (3/4)

- $\textsc{Insert}(V, x)$ 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.

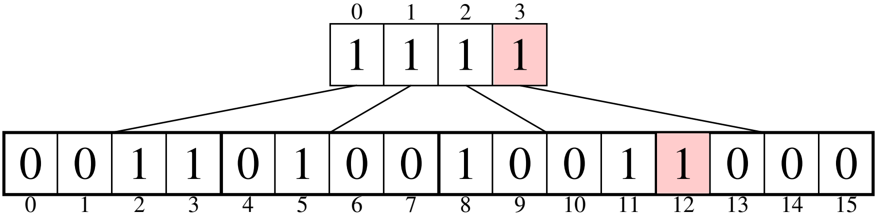


Figure: $\textsc{Insert}(V, 12), V = \{2, 3, 5, 8, 11\}$

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.

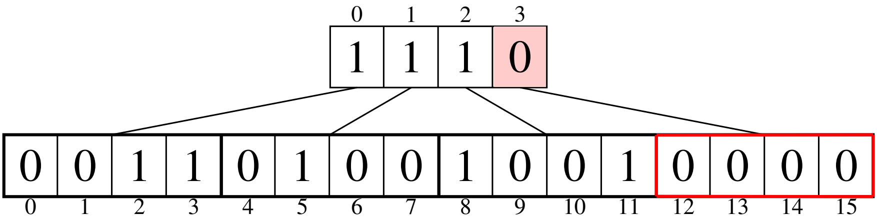


Figure: DELETE$(V, 12), V = \{2, 3, 5, 8, 11, 12\}$

前半のまとめ

- 二分木構造で時間計算量 $O(\log u)$ を達成
- 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, 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
- 次回は, 線形探索を回避するために新たなデータ構造を考える