# van Emde Boas trees

2020年度前期・高度データ構造

本日は van Emde Boas tree というデータ構造を紹介します。

# おさらい

- 【前回のデータ構造】
- q-fast trie [Willard 1984]
  - 各クエリ・操作を$O(\sqrt{\log u})$ 時間で実現

  ➔ もっともっと速く！

- 【今週のデータ構造】
- van Emde Boas tree [van Emde Boas et al. 1977]
  - 各クエリ・操作を $O(\mathrm{loglog}\ u)$ 時間で実現

前回の講義では、各クエリ・操作を O(√log u) 時間で実現できる q-fast trie を取り扱いました。ここで、u は全体集合 U のサイズです。

では、クエリ・操作時間をより高速化することは可能なのでしょうか？

今週は、O(log log u) 時間で各クエリ・操作を実現する van Emde Boas tree を取り扱います。

# $O(\log\log u)$への道 [1/2]

- どうやったら $O(\log\log u)$ という計算量が出てきそうか，そこから考えてみる
- よくある再帰的なアルゴリズムの計算量の漸化式

$$T(u) = T(u/2) + O(1)$$

O(log log u) 時間を実現するために、そもそもどんな戦略であれば O(log log u) という計算量が出てきそうか、という方向から考えてみることにします。

まず、よくある再帰的なアルゴリズムの計算量の漸化式
T(u) = T(u / 2) + O(1)
をおさらいしてみましょう。

この式は、一回の試行で問題サイズ u が半分の u/2 になり、かつ一回の試行に O(1) 時間を要する、という意味です。

例えば、2分探索の計算量はこの漸化式で表せます。

（次のスライドに続く）

# $O(\log\log u)$への道 [1/2]

- どうやったら $O(\log\log u)$ という計算量が出てきそうか，そこから考えてみる
- よくある再帰的なアルゴリズムの計算量の漸化式

$$T(u) = T(u/2) + O(1)$$

- 簡単のために $u = 2^h$ とし，上式を $h$ 回適用すると

$$T(u) = T(u/2^h) + O(h) = T(1) + O(\log u)$$
$$= O(\log u)$$

- $O(\log\log u)$ を得るためには，もうひと工夫必要

この漸化式を解いてみます。簡単のため $u = 2^h$ とします。ここで h は非負整数です。

上の式を h 回適用、つまり h 回再帰すると、

T(u)
= $T(u/2^h) + O(h)$
= $T(1) + O(\log u)$
= $O(\log u)$

を得ます。ここで、T(1) は問題サイズ 1 に掛かる時間なので、T(1) = O(1) です。

というわけで、この漸化式で計算量が表されるアルゴリズムの計算量はO(log u) であることがわかりました。
2分探索の計算量が O(log u) であることと確かに一致しています。

しかし、O(log log u) を得るためには、もう一工夫必要そうです。

# $O(\log\log u)$への道 [2/2]

- $\log\log u$ を得るためには，$u$ の値を半分にするのではなく $u$ の次数を半分にすればよい

$$T(u) = T(\sqrt{u}) + O(1)$$

- 簡単のために $u = 2^{2^k}$ とし，上式を $k$ 回適用すると

$$T(u) = T(u^{1/2^k}) + O(k) = T(2) + O(k)$$
$$= O(\log\log u)$$

- というわけで，$O(\log\log u)$ 時間を実現するためには，空間サイズ $u$ を 再帰的に $\sqrt{u}$ に縮小していけば良さそう

先ほどの漸化式では、問題サイズ u を半分にしていっていました。

log log u を得るためには、u の値を半分にするのではなく u の次数を半分にすれば上手く行きます。

つまり、
T(u) = T(√u) + O(1)
という漸化式です。

この漸化式を解いてみます。簡単のために 2^{2^{k}} として、この式を k 回適用すると

T(u)
= T(u^{1/2^k}) + O(k)
= T(2) + O(k)
= O(log log u)

が確かに得られました。

というわけで，O(loglog u) 時間を実現するためには，空間サイズ u を 再帰的に √u に縮小していけば良さそうということが見えてきました。

# 集合$S$のビット列表現

- $U = \{1, 2, ..., 16\}, S = \{2, 10, 11, 16\}$ とする
- このとき，例えば successor$(7, S) = 10$

- 集合 $S$ のビット列表現 **S** を考える
- $S = \{2, 10, 11, 16\}$ に対し **S** = 0100000001100001

- **S** をそのまま用いると，insert/delete は $O(1)$ 時間でできるが，successor/predecessor は $O(u)$ 時間かかる

それでは、van Emde Boas tree の具体的な中身に入っていきます。

全体集合を U = {1, 2, ..., 16} とし、その部分集合 S = {2, 10, 11, 16} を考えます。
このとき、例えば successor(7, S) = 10 です。

このデータ構造では、集合 S のビット列表現 **S** を用います。

S = {2, 10, 11, 16} に対し **S** = 0100000001100001 です。

つまり、**S** は
i ∈ S ならば **S**[i] = 1
i ∉ S ならば **S**[i] = 0
を満たす長さ |S| のビット列です。

このビット列 **S** をそのまま用いても insert/delete は O(1) 時間でできますが，
successor/predecessor には O(u) 時間かかってしまいます。

(例えば、**S** = 0000000001 で、successor(1, S) を探す場合など)

# Sの分割

簡単のために$\sqrt{u}$は整数と仮定

- S をそれぞれサイズ $\sqrt{u}$ の $\sqrt{u}$ 個のブロックに分割する

$$\mathbf{S} = \overbrace{0}^{\mathbf{S}_1}\overbrace{1000}^{\mathbf{S}_2}\overbrace{0000}^{\mathbf{S}_3}\overbrace{1100001}^{\mathbf{S}_4}$$

- $\mathrm{high}(x) = \lceil x/\sqrt{u} \rceil,\ \mathrm{low}(x) = ((x-1) \bmod \sqrt{u}) + 1$ とすると

$$\mathbf{S}[x] = \mathbf{S}_{\mathrm{high}(x)}[\mathrm{low}(x)]$$

$\mathrm{high}(x)$ は $x$ が格納されているブロックを示し，
$\mathrm{low}(x)$ はそのブロック中での $x$ の位置を示している

そこで、次のような工夫をします。

議論の簡単のため、√u は整数であると仮定します。

S をそれぞれサイズ √u の √u 個のブロックに分割します。

このスライドの例では、長さ 16 のビット列 S がそれぞれ長さ 4 の 4 個のブロック $S_1$, $S_2$, $S_3$, $S_4$, に分割されました。

ここで、1 ≦ x ≦ |S| に対して high(x) と low(x) をこのように定義すると、
$S[x] = S_{high(x)}[low(x)]$
が成り立ちます。

つまり、high(x) は x を含むブロックの番号を表し、low(x) はそのブロック中の x の位置を表します。

# Sの分割

簡単のために$\sqrt{u}$は整数と仮定

- S をそれぞれサイズ $\sqrt{u}$ の $\sqrt{u}$ 個のブロックに分割する

$$\mathbf{S} = \overbrace{0}^{\mathbf{S}_1}\overbrace{10000000}^{\mathbf{S}_2}\overbrace{11}^{\mathbf{S}_3}\overbrace{00001}^{\mathbf{S}_4}$$

- $\mathrm{high}(x) = \lceil x/\sqrt{u} \rceil,\ \mathrm{low}(x) = ((x-1) \bmod \sqrt{u}) + 1$ とすると

$$\mathbf{S}[x] = \mathbf{S}_{\mathrm{high}(x)}[\mathrm{low}(x)]$$

- $x = 10$ のとき, $\mathrm{high}(10) = 3, \mathrm{low}(10) = 2$ より

$$\mathbf{S}[10] = \mathbf{S}_3[2] = 1$$

$\mathrm{high}(x)$ は $x$ が格納されているブロックを示し,
$\mathrm{low}(x)$ はそのブロック中での $x$ の位置を示している

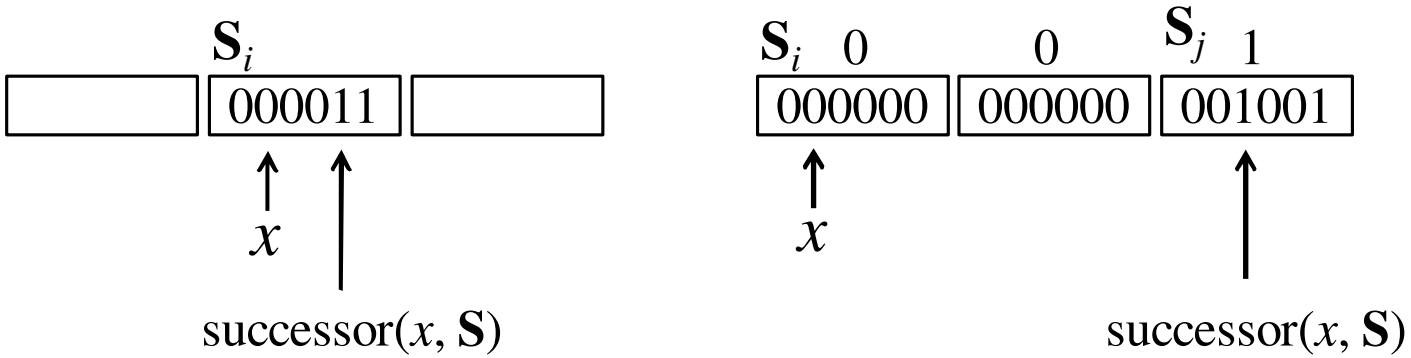
具体例で確認してみます。

x = 10 のとき, high(10) = 3, low(10) = 2 となります。

S[10] = $S_3$[2] = 1 なので、確かに x = 10 は ブロック $S_3$ の 2 番目に格納されていることが確認できました。

## successor はこんな感じでいけそう!?!?

1. $x$ と同じブロック $\mathbf{S}_i$ 中での $x$ の successor を探す
2. Step 1 で successor が見つからなかったら，
   $\mathbf{S}_i$ 以降の最初の空でないブロック $\mathbf{S}_j$ を探す $(j > i)$
3. $\mathbf{S}_j$中で 1が立っている最初のビットを探す




- Step 1, 3 はブロック内での successor query
- Step 2 も 実はブロックに対する successor query

このブロック分割を使って、successor を以下のような方法で正しく計算できます。

1. x と同じブロック $S_i$ 中での x の successor を探す。

2. Step 1 で successor が見つからなかったら，$S_i$ 以降の最初の空でないブロック $S_j$ を探す (j > i)。

3. $S_j$ 中で1が立っている最初（最も左）のビットを探す。

この図を見るとわかる通り、Step 1 と Step 3 はブロック内での successor クエリです。

ここで、1 を含むブロックに 1、1 を含まないブロックに 0 を割り当てると、実は Step 2 も「ブロックに対する successor クエリ」とみなすことができます（図を参照）

# 疑似コード

naive_successor($x$, $\mathbf{S}$):

1. compute naive_successor(low($x$), $\mathbf{S}_{\text{high}(x)}$)
2. **if** a successor is found **then return** it
3. **else**
4. $j \leftarrow$ naive_successor(high($x$), $\mathbf{S}$.summary)
5. **return** naive_successor($-\infty$, $\mathbf{S}_j$)

summaryは非空ブロックと空ブロックを表す配列：

$$\mathbf{S}.\text{summary}[i] = \begin{cases} 1 & \text{if } \mathbf{S}_i \neq \phi \\ 0 & \text{if } \mathbf{S}_i = \phi \end{cases}$$

前ページのアルゴリズムを naive_successor と名付け、疑似コードで記述するとこのようになります。

ここで、summary は非空ブロックと空ブロックを表す配列です。

非空ブロック：1を含むブロック
空ブロック：1を含まないブロック

# naive_successorの時間計算量

- naive_successor の再帰呼び出しは高々3回
- $u = 2^{2^k}$ とすると

$$T(u) = 3T(\sqrt{u}) + O(1)$$
$$= 3^k T(u^{1/2^k}) + O(k) = 3^k T(2) + O(k)$$
$$= O(3^{\log\log u}) = O((\log u)^{\log 3})$$

➔ 再帰呼び出しの回数は高々1回でなければ
時間計算量は $O(\log\log u)$ にならない！

では、naive_successor の時間計算量を解析してみましょう。

前ページの疑似コードを見て分かる通り、naive_successor の再帰呼び出しの回数は高々3回です（1, 4, 5 行目）

よって、時間計算量の漸化式は
$T(u) = 3T(\sqrt{u}) + O(1)$
となります。

$3T(\sqrt{u})$ の 3 が再帰呼び出しの最大回数 3 に対応しています。

この漸化式を解くと、
$T(u) = O((\log u)^{\log 3})$
となってしまい、$O(\log \log u)$ よりも大きな値になってしまっています。

何が悪かったのかというと、2行目の式変形で $3^k$ という項が出てきたのが原因です。

言い換えると、再帰呼び出しの回数は高々1回でなければ、時間計算量は $O(\log\log u)$ にならないということになります。

# $O(\log\log u)$-time successor query

successor($x$, **S**):

1. **if** $x < \mathbf{S}.\min$ **then return** $\mathbf{S}.\min$
2. **if** $\text{low}(x) < \mathbf{S}_{\text{high}(x)}.\max$ **then**
3. **return** $(\text{high}(x)-1)\sqrt{|\mathbf{S}|} + \text{successor}(\text{low}(x), \mathbf{S}_{\text{high}(x)})$
4. **else**
5. $j \leftarrow \text{successor}(\text{high}(x), \mathbf{S}.\text{summary})$
6. **return** $(j-1)\sqrt{|\mathbf{S}|} + \mathbf{S}_j.\min$

$$\mathbf{S}.\min = \min\{k \mid \mathbf{S}[k] = 1\}$$
$$\mathbf{S}.\max = \max\{k \mid \mathbf{S}[k] = 1\}$$

そこで、再帰呼び出しの回数を高々1回で済むように上手く設計したのが、この疑似コードのアルゴリズムです。

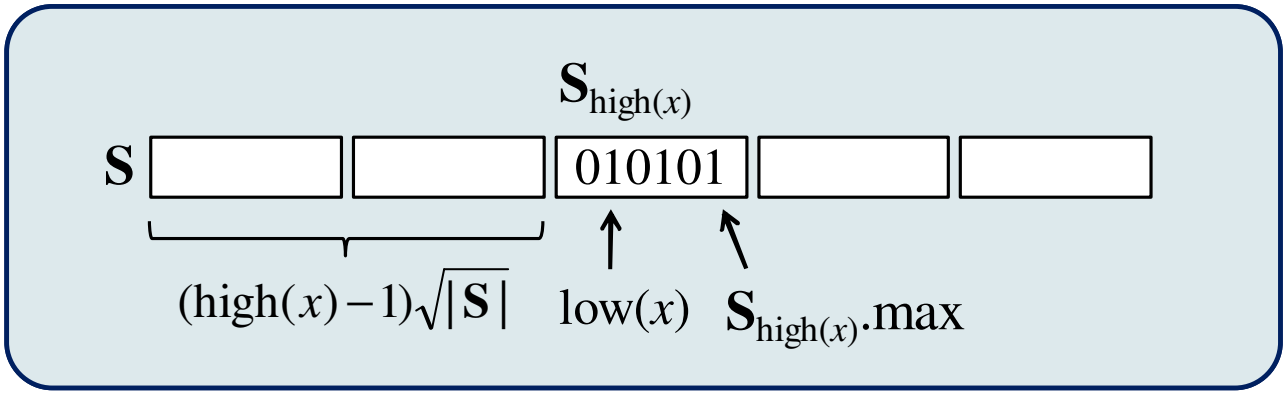
ここで、S.min と S.max はそれぞれ集合 S の最小値と最大値を表します。

1行目で、もし x が S.min よりも小さいのであれば S.min を返せばよいので、そのような処理をしています。

以降、x が S.min 以上である場合について、場合1と場合2に分けて説明します。

# 場合 1

2. **if** $\mathrm{low}(x) < \mathbf{S}_{\mathrm{high}(x)}.\max$ **then**
3. **return** $(\mathrm{high}(x)-1)\sqrt{|\mathbf{S}|} + \mathrm{successor}(\mathrm{low}(x), \mathbf{S}_{\mathrm{high}(x)})$




場合１：疑似コード2行目の条件を満たす場合です。

high(x) は x が含まれるブロック番号、low(x) はそのブロック中の x の位置であることを思い出しましょう。

low(x) が ブロック $\mathbf{S}_{\mathrm{high}(x)}$ 内の最大値より小さい場合を今考えているので、このブロックの中に答えとなる successor(x, S) があることになります。

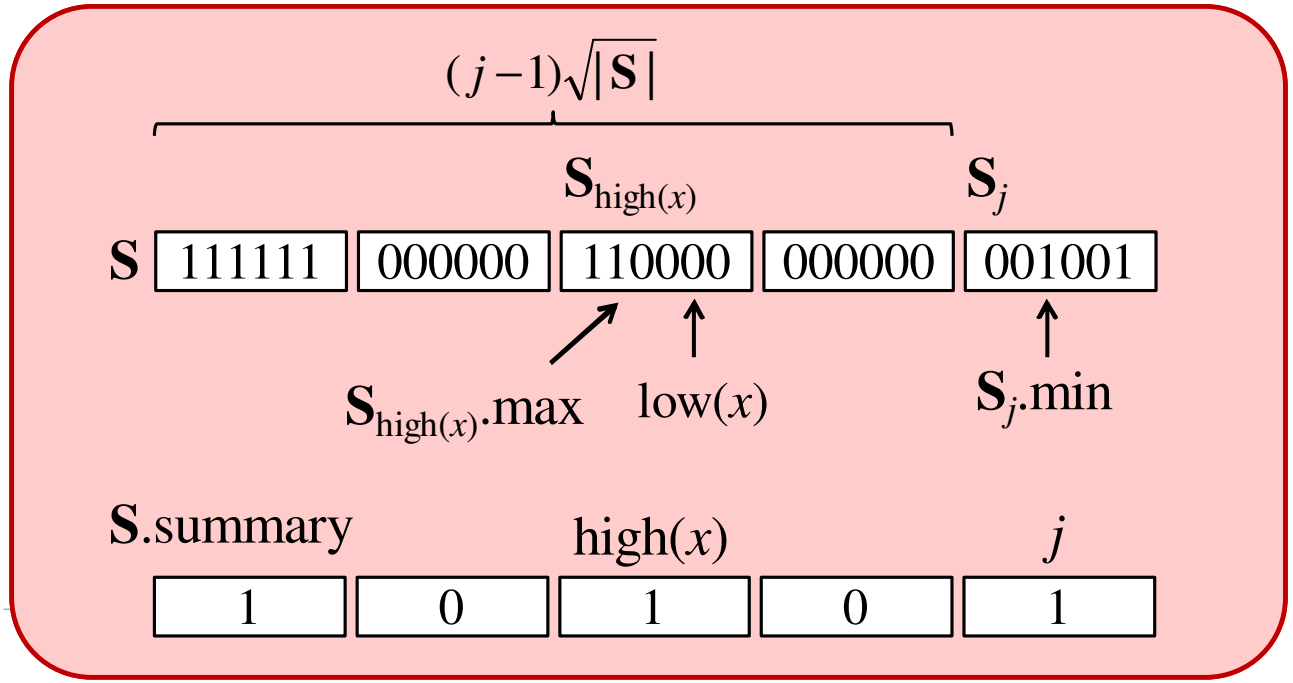
その答えはブロック内 での successor クエリ $\mathrm{successor}(\mathrm{low}(x), \mathbf{S}_{\mathrm{high}(x)})$ を再帰的に求めることで得られます。

ただし、この値に high(x) の前のオフセット値 $(\mathrm{high}(x)-1)\sqrt{|\mathbf{S}|}$ を足しておく必要があります。

以上が場合１です。

# 場合２

4. **else** // $\mathrm{low}(x) \geqq \mathbf{S}_{\mathrm{high}(x)}.\max$ のとき //
5. $j \leftarrow \mathrm{successor}(\mathrm{high}(x), \mathbf{S}.\mathrm{summary})$
6. **return** $(j-1)\sqrt{|\mathbf{S}|} + \mathbf{S}_j.\min$




場合２：疑似コード2行目の条件を満たさない場合です。

このとき、low(x) が ブロック $S_{high(x)}$ 内の最大値以上なので、このブロックの中に答えとなる successor(x, S) は存在しません。

そこで、次に 1 を含むブロック $S_j$ を求めます（j > high(x)）
これは、S.summary に対する high(x) の successor クエリで再帰的に求めることができます。

そして、$S_j$ の最小値にオフセットを足した値を返せばよい、ということになります。

以上が場合２です。

# $O(\text{loglog } u)$-time successor query

successor($x$, **S**):

1. **if** $x < \mathbf{S}.\min$ **then return** $\mathbf{S}.\min$
2. **if** $\text{low}(x) < \mathbf{S}_{\text{high}(x)}.\max$ **then**
3. **return** $(\text{high}(x)-1)\sqrt{|\mathbf{S}|} + \text{successor}(\text{low}(x), \mathbf{S}_{\text{high}(x)})$
4. **else**
5. $j \leftarrow \text{successor}(\text{high}(x), \mathbf{S}.\text{summary})$
6. **return** $(j-1)\sqrt{|\mathbf{S}|} + \mathbf{S}_j.\min$

高々一回の再帰呼び出し
➔ $O(\text{loglog } u)$ 時間

$$\mathbf{S}.\min = \min\{k \mid \mathbf{S}[k] = 1\}$$
$$\mathbf{S}.\max = \max\{k \mid \mathbf{S}[k] = 1\}$$

1行目の条件を満たす場合は、successor の再帰呼び出し回数は0です。

また、場合１、場合２にいずれにおいても、successor の再帰呼び出し回数は1回です。

よって、いずれの場合でも、高々1回の再帰呼び出しで済んでいるので、このアルゴリズムは successor(x, S) を O(log log u) 時間で計算できています。

# 練習問題

- $O(\log\log u)$ 時間で動作する predecessor query の疑似コードを示せ

※ 演習問題 **提出〆切：7月2日（木）23：59**

では、本日の練習問題です。

O(log log u) 時間で動作する predecessor クエリの疑似コードを示してください。

# $O$(loglog $u$)-time insertion [1/3]

【難しさ】

- $\mathbf{S}_i$ と $\mathbf{S}$.summary を両方更新しなければならないが，
  許される再帰呼び出しの回数は高々1回のみ

【考察1】

- 前述の successor アルゴリズムでは，
  各ブロック $\mathbf{S}_i$ に対して $\mathbf{S}_i$.min と $\mathbf{S}_i$.max が
  正しくメンテナンスされていれば十分である

続いて、O(log log u) 時間で insert を行う方法を考えます。

insert においては、最悪時には $S_i$ と S.summary を両方更新しなければなりませんが、前述の通り、O(log log u) 時間を達成するためには、許される再帰呼び出しの回数は高々1回のみです。

ここに insert の難しさがあります。

この難しさを克服するために、いくつかの考察を行います。

まず1つめの考察は、前述の successor アルゴリズムでは，各ブロック $S_i$ に対して $S_i$.min と $S_i$.max が正しくメンテナンスされていれば十分である、ということです。

# $O(\log\log u)$-time successor query（再掲）

successor($x$, **S**):

1. **if** $x < \mathbf{S}.\min$ **then return** $\mathbf{S}.\min$
2. **if** $\text{low}(x) < \mathbf{S}_{\text{high}(x)}.\max$ **then**
3. **return** $(\text{high}(x)-1)\sqrt{|\mathbf{S}|} + \text{successor}(\text{low}(x), \mathbf{S}_{\text{high}(x)})$
4. **else**
5. $j \leftarrow \text{successor}(\text{high}(x), \mathbf{S}.\text{summary})$
6. **return** $(j-1)\sqrt{|\mathbf{S}|} + \mathbf{S}_j.\min$

$$\mathbf{S}.\min = \min\{k \mid \mathbf{S}[k] = 1\}$$
$$\mathbf{S}.\max = \max\{k \mid \mathbf{S}[k] = 1\}$$

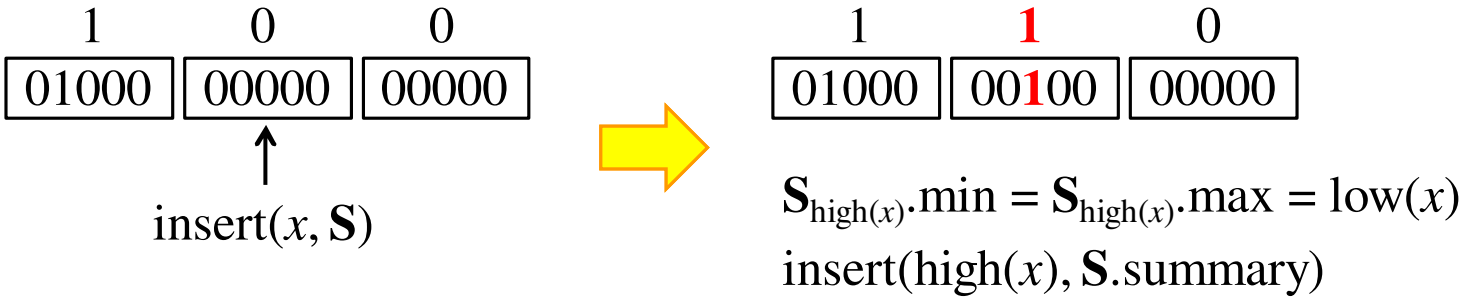
考察1を確認してみます。

確かに、実際に値を比較に使ったり、値を返したりしている部分では、S.min と S.max のみを使っているのがわかると思います。

# $O(\log\log u)$-time insertion [2/3]

【考察2】

- $\mathbf{S}_{\mathrm{high}(x)} = \phi$ のとき $\Rightarrow$ $\mathbf{S}_{\mathrm{high}(x)}$.max と $\mathbf{S}_{\mathrm{high}(x)}$.min を更新
  $\mathbf{S}$.summary を更新




次に考察2です。

$\mathbf{S}_{\mathrm{high}(x)}$ が空の場合を考えます。

この場合、insert(x, S) を行うためには、
(1) $\mathbf{S}_{\mathrm{high}(x)}$.max と $\mathbf{S}_{\mathrm{high}(x)}$.min をそれぞれ 0 から 1 に更新
(2) S.summary を更新
を行えばよいことがわかります。

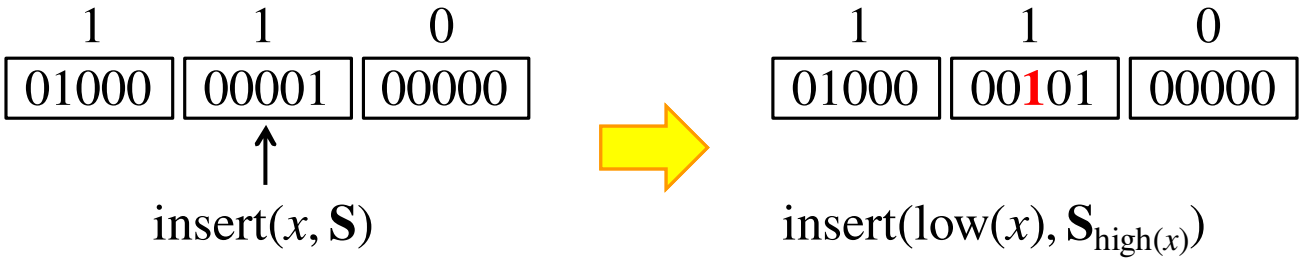
ここで、(1) では再帰呼び出しを行う必要がなく、(2) では S.summary に high(x) を追加する操作を再帰呼び出しすればよいので、この場合における insert の再帰呼び出しの回数は高々1回です。

# $O(\text{loglog } u)$-time insertion [3/3]

【考察3】

- $\mathbf{S}_{\text{high}(x)} \neq \phi$ のとき $\Rightarrow \mathbf{S}_{\text{high}(x)}$ を更新

  S.summary を更新する必要なし




最後に考察3です。

$S_{high(x)}$ が空ではない場合を考えます。

このとき、$S_{high(x)}$ に対応する S.summary の値はすでに 1 になっているので、S.summary を更新する必要はありません。

よって、この場合においては、low(x) を $S_{high(x)}$ に追加するための再帰呼び出し高々1回で済むことがわかります。

# $O$(loglog $u$)-time insertion の疑似コード

insert($x$, **S**):

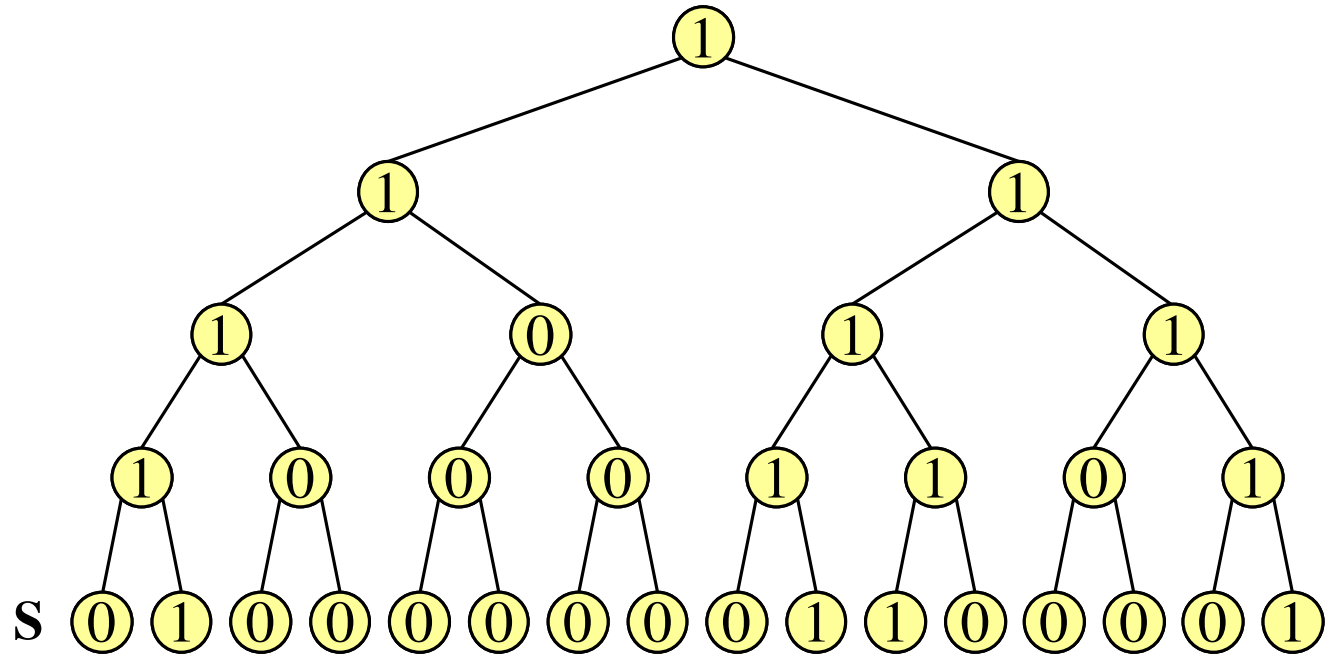
1. **if** **S**.min = nil **then** /* **S** is empty */
2. **S**.min ← $x$
3. **S**.max ← $x$
4. **if** $x$ > **S**.max **then** **S**.max ← $x$
5. **if** $x$ < **S**.min **then** **S**.min ← $x$
6. **if** $\mathbf{S}_{high(x)}$.min = nil **then** // $\mathbf{S}_{high(x)}$ is empty //
7. $\mathbf{S}_{high(x)}$.min ← low($x$)
8. $\mathbf{S}_{high(x)}$.max ← low($x$)
9. insert(high($x$), **S**.summary)
10. **else** insert(low($x$), $\mathbf{S}_{high(x)}$) // $\mathbf{S}_{high(x)}$ is non-empty //

高々一回の再帰呼び出し
➔ $O$(loglog $u$) 時間

以上の考察を元に、insert アルゴリズムを疑似コードで記述するとこのようになります。

いずれの場合も高々 1 回の再帰呼び出しで済んでいるので、このアルゴリズムは insert を O(log log u) 時間で実現しています。

# Tree view of van Emde Boas




$S = \{2, 10, 11, 16\}$

さて、この講義のはじめに、このデータ構造の名前を van Emde Boas “tree” と呼んでいました。

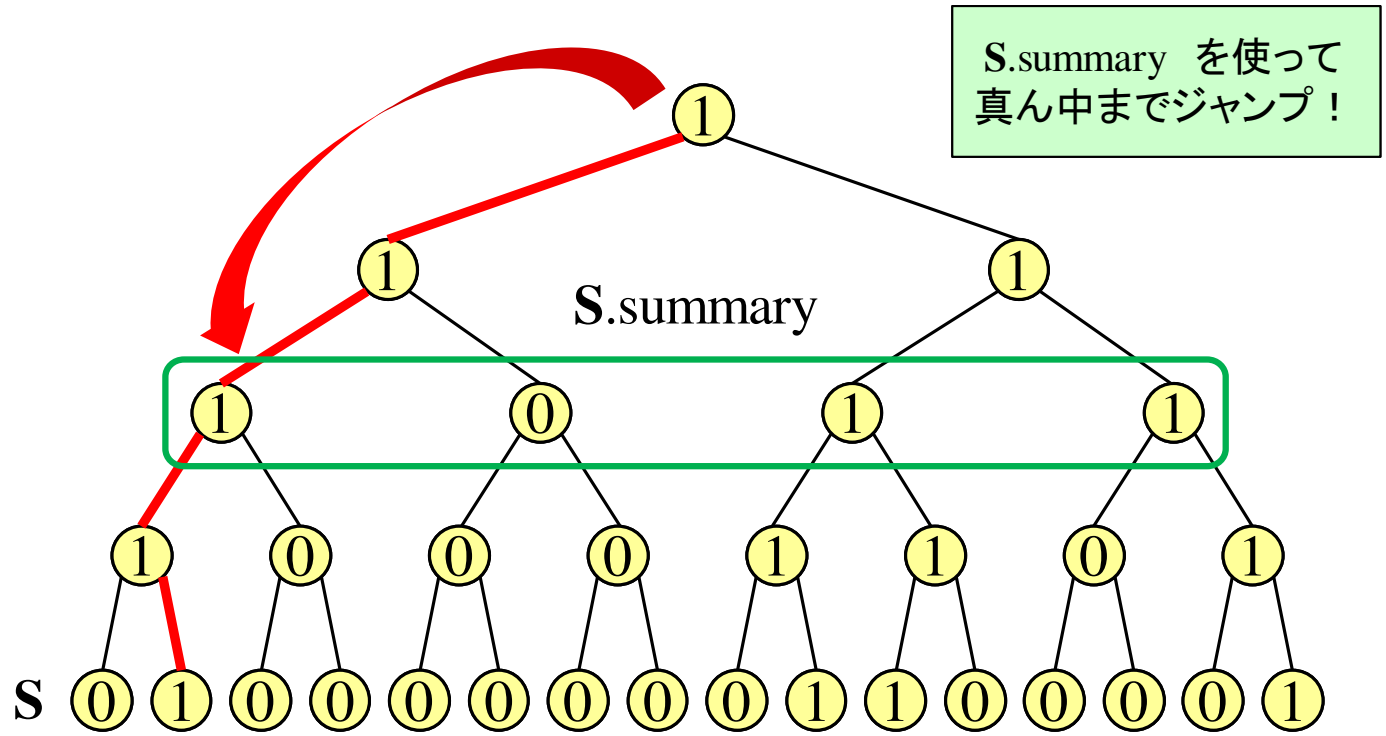
今までの説明では、木構造は陽には出てきていませんでしたが、実は、このデータ構造を以下のような木とみなすことができます。

葉がビット列 S の要素に対応する完全2分木を考えます。葉の数は u 個です。

S の要素に対応する葉は 1, それ以外の葉は 0 を格納しています。

各内部頂点 v は、v を根とする部分木の葉に 1 があるなら 1 を格納し、そうでなければ 0 を格納します。

# Tree view of van Emde Boas




$S = \{2, 10, 11, 16\}$

この木の根から、S 中の要素にアクセスすることを考えます（図の例では 2 にアクセスしようとしています）

普通であれば、根から葉 2 にパス中の辺を1つずつ降りていくわけですが、先ほどのデータ構造では、再帰呼び出しのたびに S.summary を使ってちょうど真ん中の深さの頂点（緑枠の中）にジャンプしていたことになります。

# Tree view of van Emde Boas

- 深さ $h/2$ の頂点数は？
- $u$ 個の葉から1段上がるごとに頂点数が半分になっていき，$h$ 段でちょうど根に到達するので，$\frac{u}{2^h} = 1$ → $u = 2^h$.
- よって深さ $h/2$ の頂点数は $\frac{u}{2^{h/2}} = \frac{2^h}{2^{h/2}} = 2^{h/2} = \sqrt{u}$

  ➔ 確かに深さ $h/2$ が S.summary に対応している

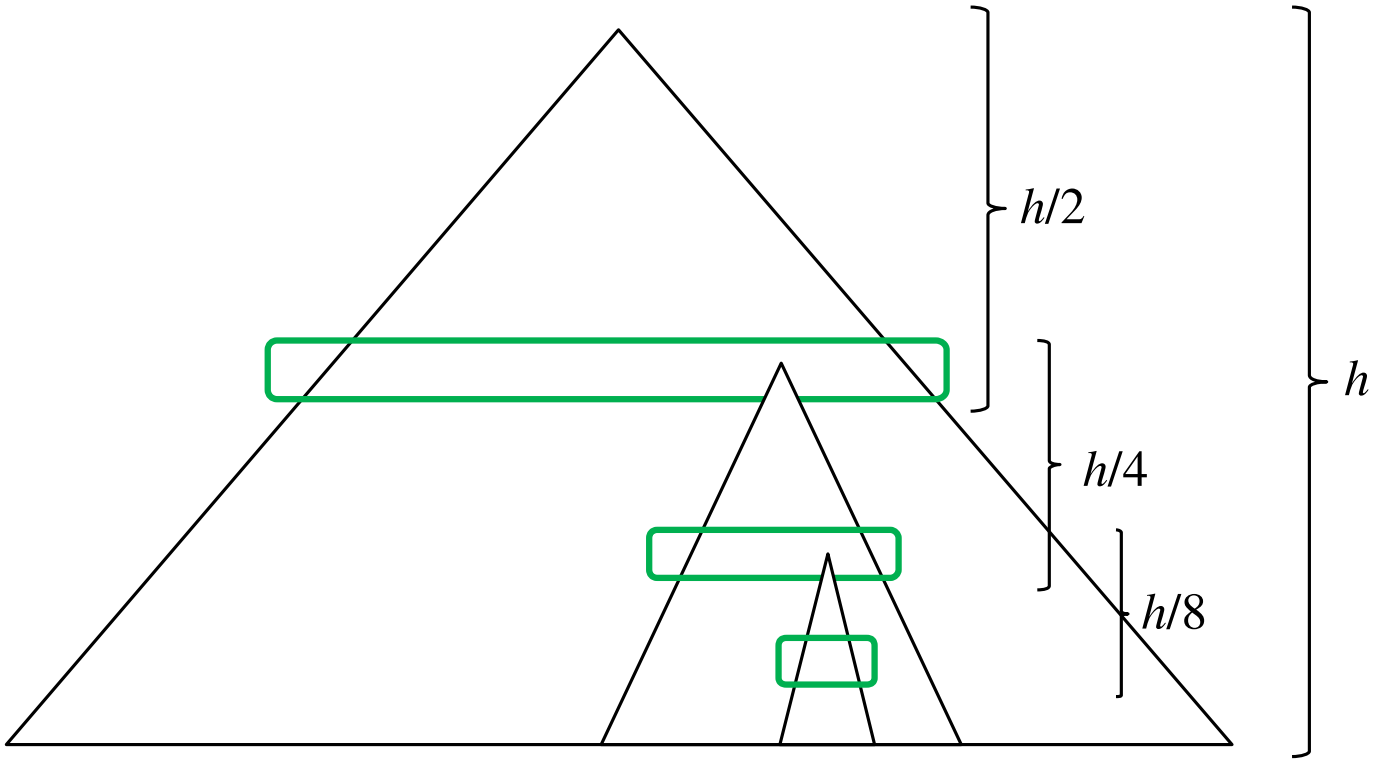
木の高さを h とし、深さ h/2 に頂点が何個あるか考えてみましょう。

u 個の葉から1段上がるごとに頂点数が半分になっていき、h 段でちょうど根に到達するので、$u / 2^h = 1$ すなわち $u = 2^h$ です。

よって、深さ h/2 の頂点数は $u/2^{h/2} = \sqrt{u}$ となり、確かに深さ h/2 がちょうど S.summary に対応していることがわかると思います。

(S.summary のサイズは$\sqrt{u}$ だったことを思い出しましょう)

# Tree view of van Emde Boas




これを再帰的に何度も繰り返していくと、この図のように、再帰呼び出しを行う際の木の高さが h/2、h/4、h/8 … というように減っていき、1 になれば必ず停止します。

# Tree view of van Emde Boas

- 再帰呼び出しする毎に，部分木の高さが半分になっていく
- 全体の木の高さは $h = \log_2 u$ なので，
  再帰呼び出しの回数は $\log\log u$ で抑えられる！

木の高さは $h = \log_2 u$ なので、再帰呼び出しの回数は log log u で抑えられる、ということになります。

これが、木を使った van Emde Boas データ構造の時間計算量の解析です。

# 領域計算量

- van Emde Boas データ構造の領域計算量の漸化式は

$$S(u) = \sqrt{u}\,(S(\sqrt{u}) + O(1))$$

これを解くと $S(u) = O(u)$

最後に、このデータ構造の領域計算量について見てみます。

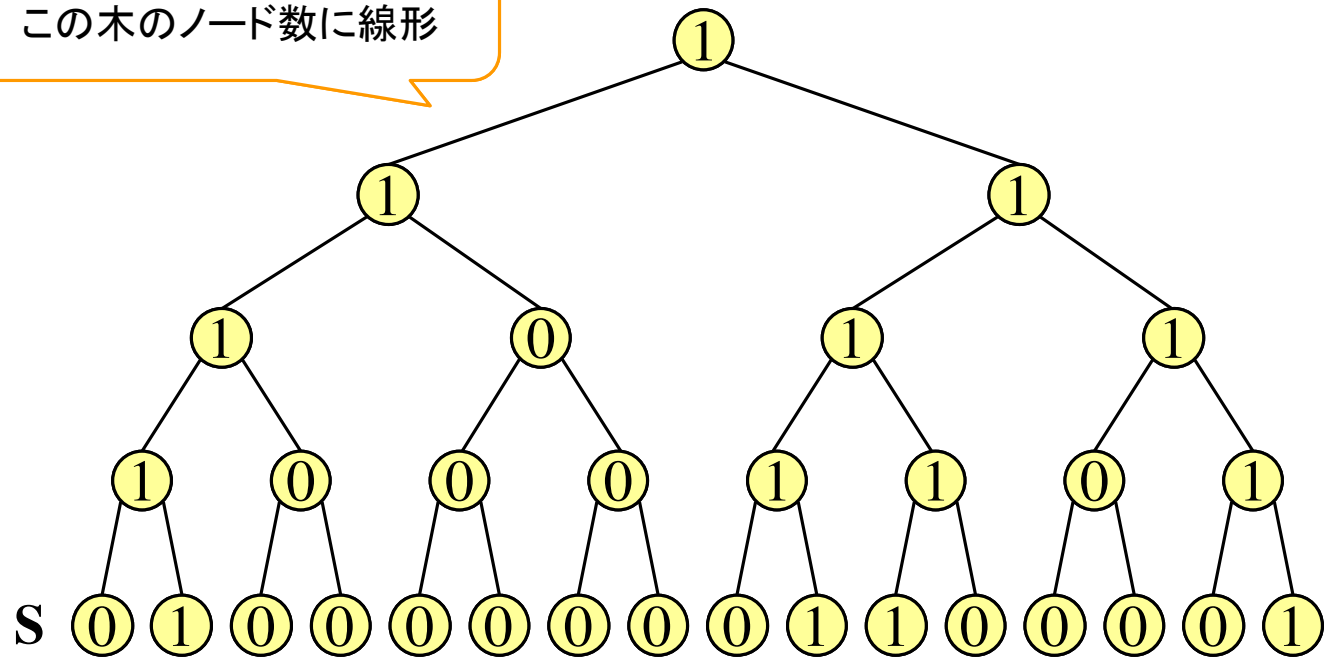
領域計算量は
S(u) = √u (S(√u)) + O(1)
という漸化式で表すことができます。

この漸化式の解き方はやや面倒なので、ここでは割愛しますが、これを解くと S(u) = O(u) が得られます。

つまり、van Emde Boas データ構造は O(u) 領域を使用する、ということです。

# Tree view of van Emde Boas

van Emde Boas の領域は
この木のノード数に線形




$S = \{2, 10, 11, 16\}$

領域計算量についても、木を使った解析を行うことができます。

各頂点が保持する 1 または 0 は、まさに各ブロックが格納していたビット列の値と、S.summary の値に対応しています。

よって、このデータ構造が保持するべき情報は、各頂点内の 0/1 と、各ブロックの max/min だけということになります。

その個数は、明らかにこの木のサイズを超えません。

よって、van Emde Boas データ構造の領域は、この木の頂点数に線形です。

この木は葉が u 個あり、すべての内部頂点は枝分かれしているので、総頂点数は 2u − 1 です。

よって、van Emde Boas データ構造の領域は O(u) ということになります。

以上が木を使った領域の解析です。

# まとめ

**定理**

van Emde Boas tree を用いると，member, predecessor/successor, insert/delete を $O(\log\log u)$ 時間，min/max を O(1) 時間で計算可能である．領域計算量は $O(u)$ である．

本日の内容をまとめた定理です。

van Emde Boas tree を用いると，member, predecessor/successor, insert/delete を O(loglog u) 時間，min/max を O(1) 時間で計算することができます．

delete は insert よりもやや複雑なのですが、O(log log u) 時間で行うことができます。興味のある人は調べてみてください。

領域計算量は、前のスライドでやったように O(u) です．

# まとめ

**定理**

van Emde Boas tree を用いると，member, predecessor/successor, insert/delete を $O(\log\log u)$ 時間，min/max を O(1) 時間で計算可能である．領域計算量は $O(u)$ である．

【欠点】$n << u$ のときは，かなり無駄な領域を使用する

例）$U = \{1, 2, ..., 10000\}, S = \{1, 4000, 9998\}$ のとき

$$\mathbf{S} = \underline{1}0000000...000000\underline{1}000000\ldots0000000\underline{1}00$$

O(log log u) 時間という非常に高速なクエリ・操作を実現できる van Emde Boas データ構造ですが、欠点もあります。

もし集合 S の要素数 n が 全体集合 U のサイズ u よりも遥かに小さい場合には、O(u) はかなり無駄な領域を使用してしまっていることになります。

例えば、この例のように u = 10000 で、n = 3 のような極端な場合を考えてみましょう。

このときビット列 S の長さは 10000 ですが、実際には集合 S は3つしか要素を含まないので、successor などは素朴な方法で行ったほうが高速かつ省領域ということになってしまいます。

よって、S が密な場合には van Emde Boas は非常に有効ですが、疎な場合には有効でないことがあります。

予告

## 次回予告

- predecessor/successor を $O(\log\log u)$ 時間かつ $O(n)$ 領域で実現するデータ構造

そこで次回は、predecessor/successor クエリを O(log log u) 時間 かつ S の要素数 n に線形な O(n) 領域で実現するデータ構造について紹介します。

今回の講義は以上です。