y-fast tries

2020年度前期・高度データ構造

本日は y-fast trie という
データ構造を取り扱います。

省領域な整数データ構造

【前回のおさらい】

- van Emde Boas trees は 各クエリ・操作を $O(\log\log u)$ 時間，$O(u)$ 領域で実現．
  u は全体集合のサイズ．
  - 欠点：集合 S のサイズ n に関わらず，常に $O(u)$ 領域を必要としてしまう

【今週のデータ構造】

- x-fast tries [Willard 1983]：$O(n \log u)$ 領域
- y-fast tries [Willard 1983]：$O(n)$ 領域
- 各クエリに要する時間はどちらも $O(\log\log u)$

前回の講義では、各クエリ・操作を O(log log u) 時間で実現する van Emde Boas tree を取り扱いました。ここで u は全体集合 U のサイズです。

van Emde Boas tree の欠点として、集合 S の要素数 n に関わらず、常に O(u) 領域を必要としてしまうことがあります。

そこで今週は、より省領域で O(log log u) 時間クエリを実現するデータ構造を扱います。

まず、最初のステップとして O(n log y)領域の x-fast trie を紹介します。
次に、その省領域版である y-fast trie を紹介します。y-fast trie の領域は O(n) です。

省領域な整数データ構造

【前回のおさらい】

- van Emde Boas trees は 各クエリ・操作を $O(\log\log u)$ 時間，$O(u)$ 領域で実現．
  u は全体集合のサイズ．
  - 欠点：集合 S のサイズ n に関わらず，常に $O(u)$ 領域を必要としてしまう

【今週のデータ構造】

- x-fast tries [Willard 1983]：$O(n \log u)$ 領域
- y-fast tries [Willard 1983]：$O(n)$ 領域
- 各クエリに要する時間はどちらも $O(\log\log u)$

【余談】

似たような名前のデータ構造に、この講義の前半で扱った p-fast trie と q-fast trie がありました。

x-fast trie と y-fast trie の関係は、まさに p-fast trie と q-fast trie の関係と同様です（つまり、ほぼ同じアイディアを使って省領域化を実現します）

なお、これら4つのデータ構造はすべて同じ著者（D. Willard）によって発表されました。

x-fast trie

- $U = \{1, 2, 3, ..., u\}$ とする.
- $S \subseteq U$ の x-fast trie は以下を満たす2分木.
  - 高さ $h = \log_2 u$
  - 各葉は S の各要素に対応する.
  - 葉は双方向連結リストで連結されている.
  - 高さ j の各頂点 v は，ある整数 i について $[(i\text{-}1)2^j\text{+}1, i \cdot 2^j] \cap S$ に対応する．このとき，$\mathrm{id}(v) = i$ と書く.

それでは、x-fast trie を解説していきます。

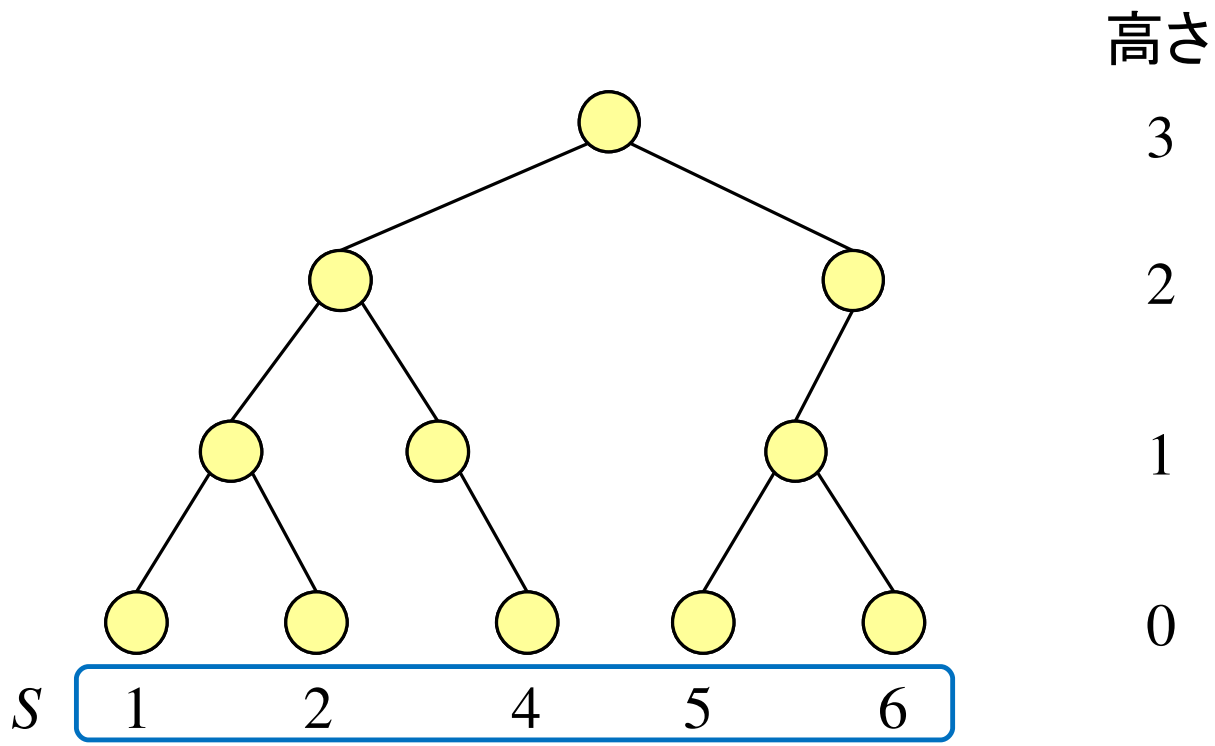
いつものように、全体集合を U = {1, 2, 3, ..., u} とします。

U の部分集合 S の x-fast trie は以下を満たす2分木です。
- 高さ h = log_2 u である。
- 各葉は S の各要素に対応する。
- 葉は双方向連結リストで連結されている。
- 高さ j の各頂点 v は，ある整数 i について [(i-1)2j+1, i・2j] ∩ S に対応する．このとき，id(v) = i と書く.

4つ目の条件、すなわち id(v) の定義については、次のページの例を使って説明します。

x-fast trie

- $U = \{1, 2, 3, ..., 8\}$, $S = \{1, 2, 4, 5, 6\}$



全体集合を U = {1, 2, 3, ..., 8} とし、
その部分集合 S = {1, 2, 4, 5, 6} を考えます。

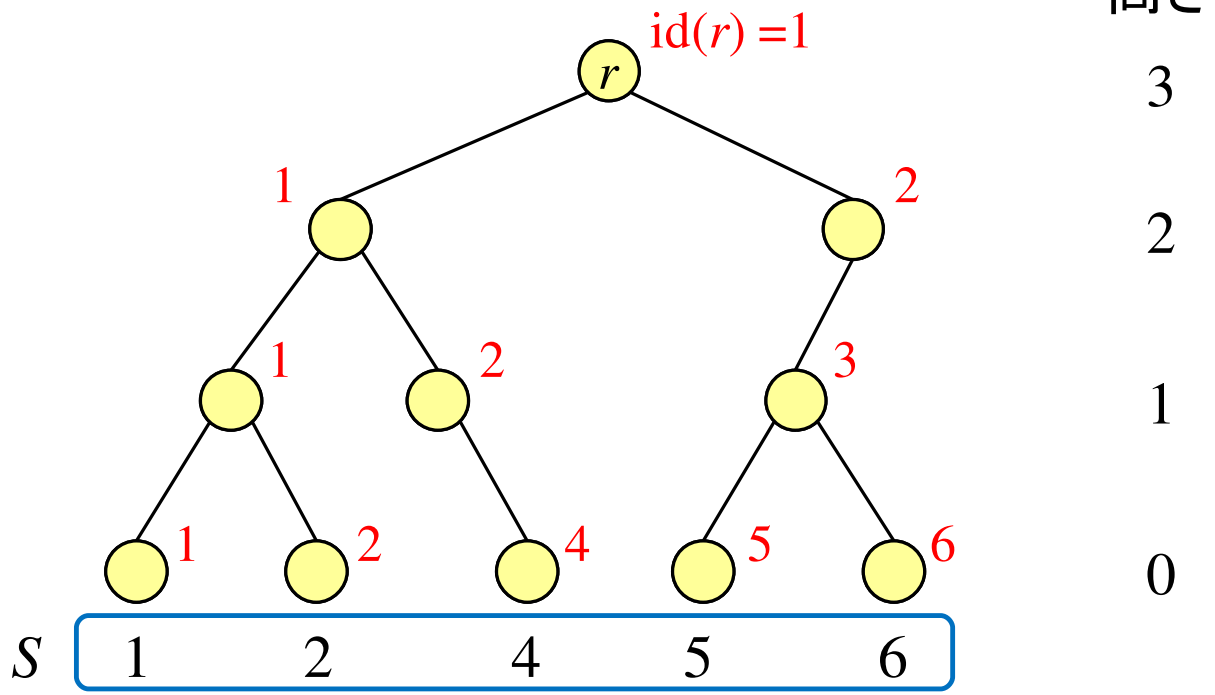
この S に対する x-fast trie はこのような木構造から成ります。

まず、高さは log_2(8) = 3 です。

葉はそれぞれ S の要素 1, 2, 4, 5, 6 に対応しています。

id関数

- $U = \{1, 2, 3, ..., 8\}$, $S = \{1, 2, 4, 5, 6\}$



x-fast trie の各頂点 v について id(v) の値は，この例だとこのようになります。

直感的には、以下の id(v) は以下のような値です。

全体集合 U に対する（想像上の）完全2分探索木を仮想的に考えてみましょう。

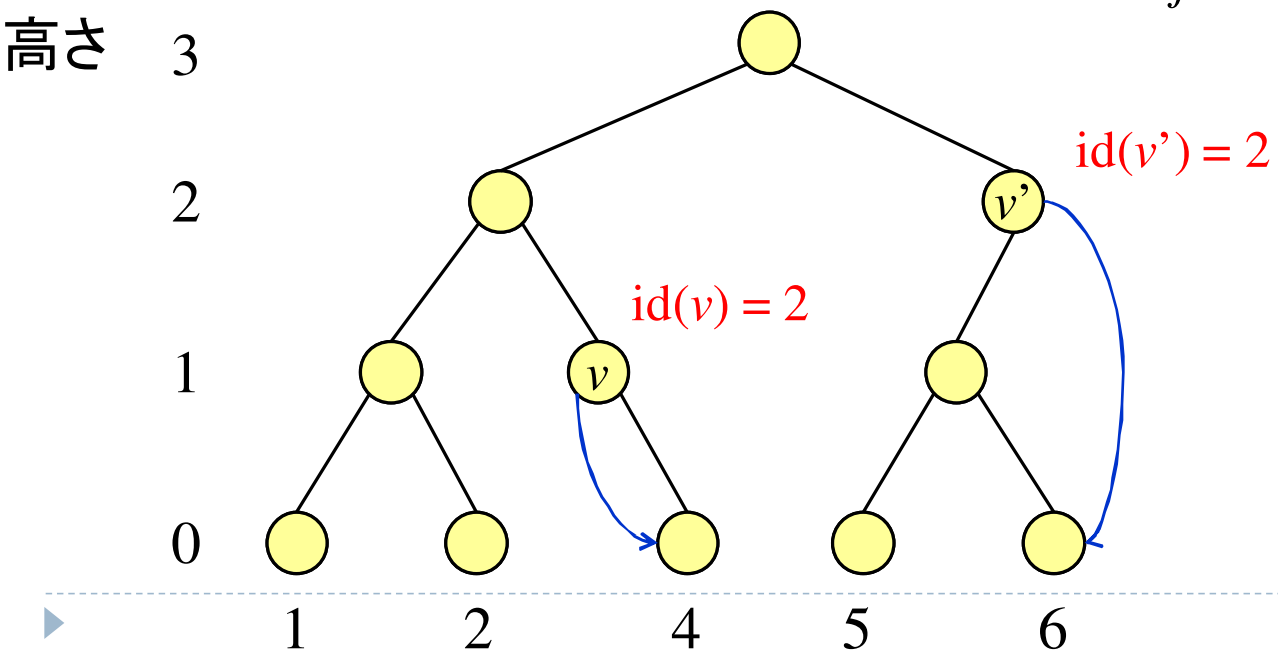
この完全2分探索木の上に x-fast trie を重ね合わせたとき、x-fast trie の頂点が各高さで左から何番目にあるか、というのが id の値です。

例えば、高さ 0 (葉の列) については、id の値は左から 1, 2 ときて、3 番目の葉はないので飛ばして、次に 4, 5, 6, となり、7 番目と 8 番目の葉はない、という感じです。

descendant関数

$$\mathrm{descendant}(v) = \begin{cases} \min([(\mathrm{id}(v)-1)2^j+1,\ \mathrm{id}(v)2^j] \cap S) & v \text{ が左の子を持たないとき} \\ \max([(\mathrm{id}(v)-1)2^j+1,\ \mathrm{id}(v)2^j] \cap S) & v \text{ が右の子を持たないとき} \end{cases}$$

ただし j は v の高さ

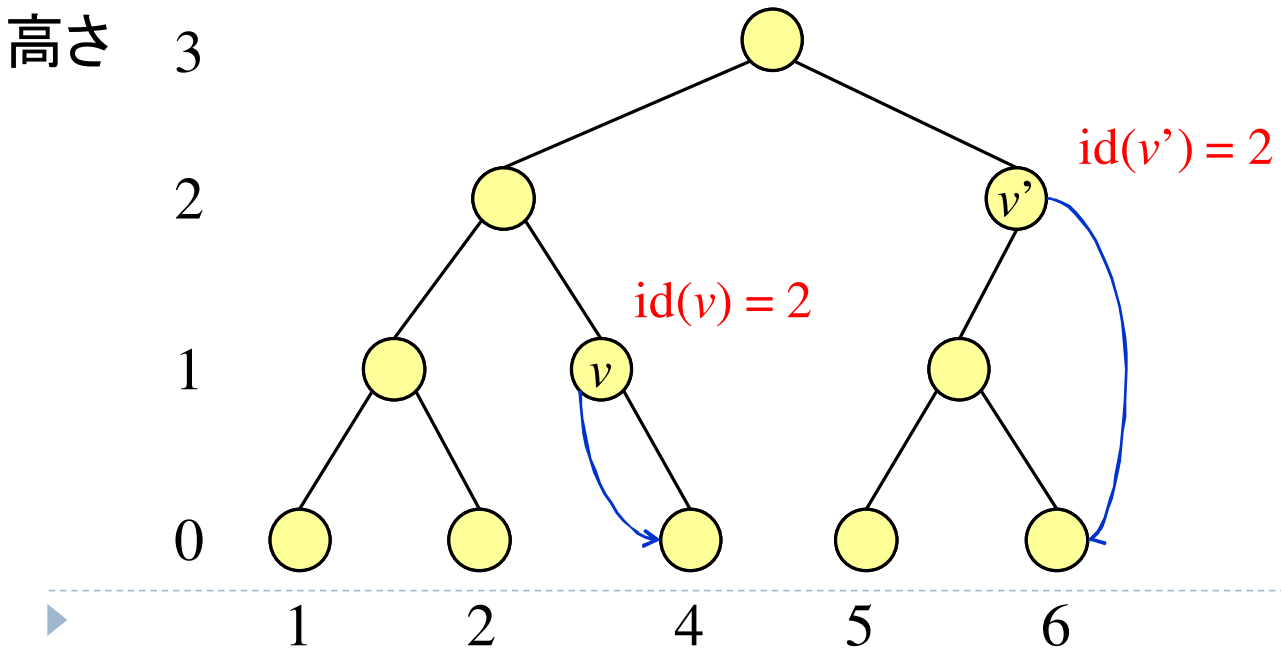


次に、descendant 関数を導入します。

各頂点 v について、descendant(v) は数学的にはこのように定義されますが、式だけ見ると辛いので、次のページで直感的に説明します。

descendant関数

- 言い換えると，v が左の子を持たないとき，descendant(v) は v を根とする部分木の葉の最小値，v が右の子を持たないときは，v を根とする部分木の葉の最大値である．



descendant 関数の直感的な説明です。

v が左の子を持たないとき，descendant(v) は v を根とする部分木の葉の最小値
v が右の子を持たないとき，descendant(v) は v を根とする部分木の葉の最大値

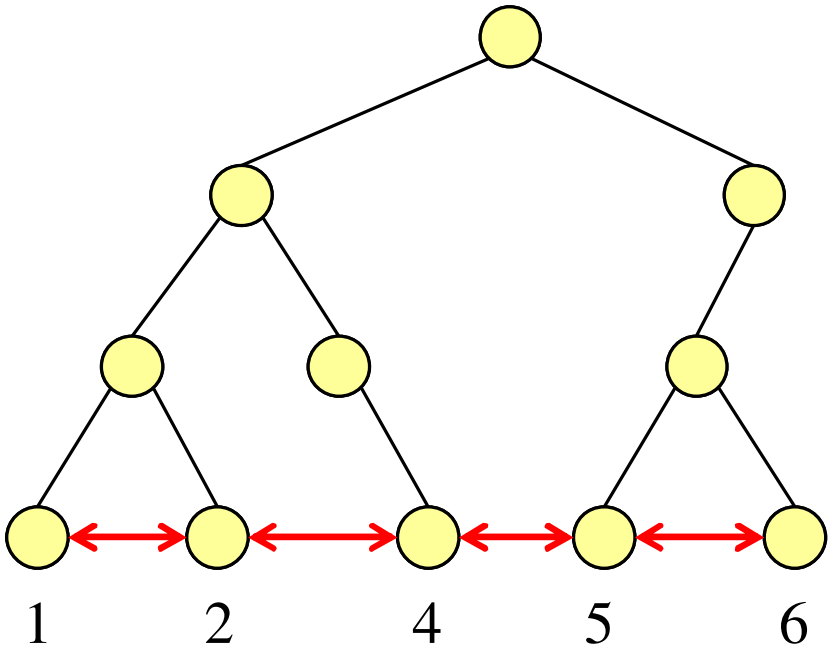
例えば、この図中の v は左の子を持ちません。v を根とする部分木の葉の最小値は 4 なので、descendant(v) = 4（青のリンク）となります。

一方、図中の v' は右の子を持ちません。v' を根とする部分木の葉の最大値は 6 なので、descendant(v') = 6（青のリンク）となります。

これを数学的に記述すると、前ページの式のようになります。

葉の双方向連結リスト

- 任意の葉から隣の葉に $O(1)$ 時間でアクセス可能

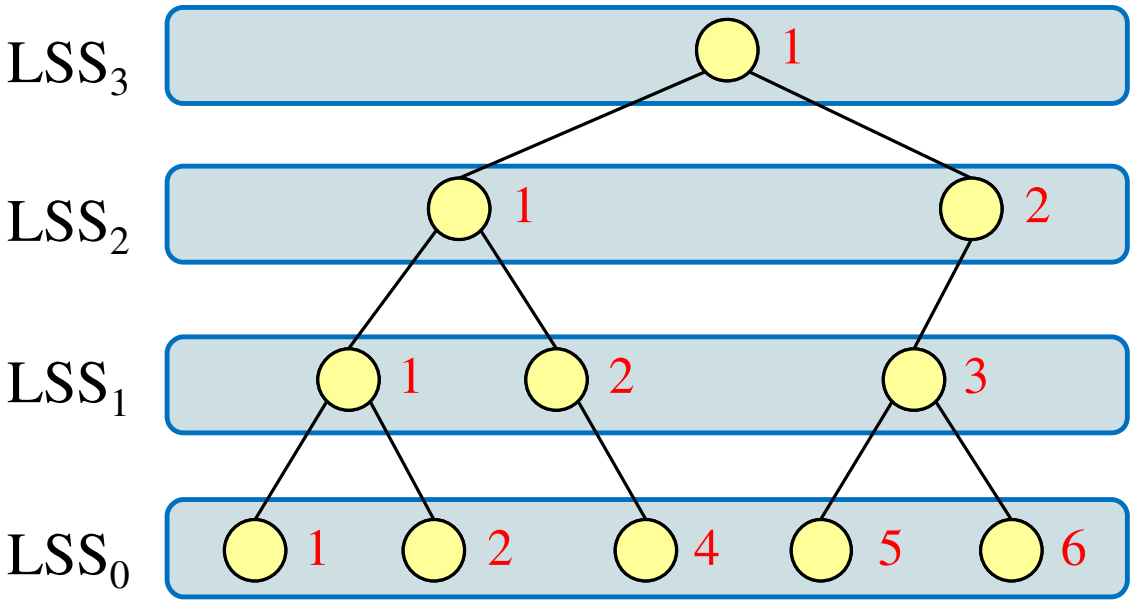


x-fast trie の葉を双方向連結リストに格納しておきます。

こうすることで、任意の葉から隣の葉に定数時間でアクセス可能になります。

level search structure (LSS)

- LSS_j: 高さ j の頂点の id のリスト

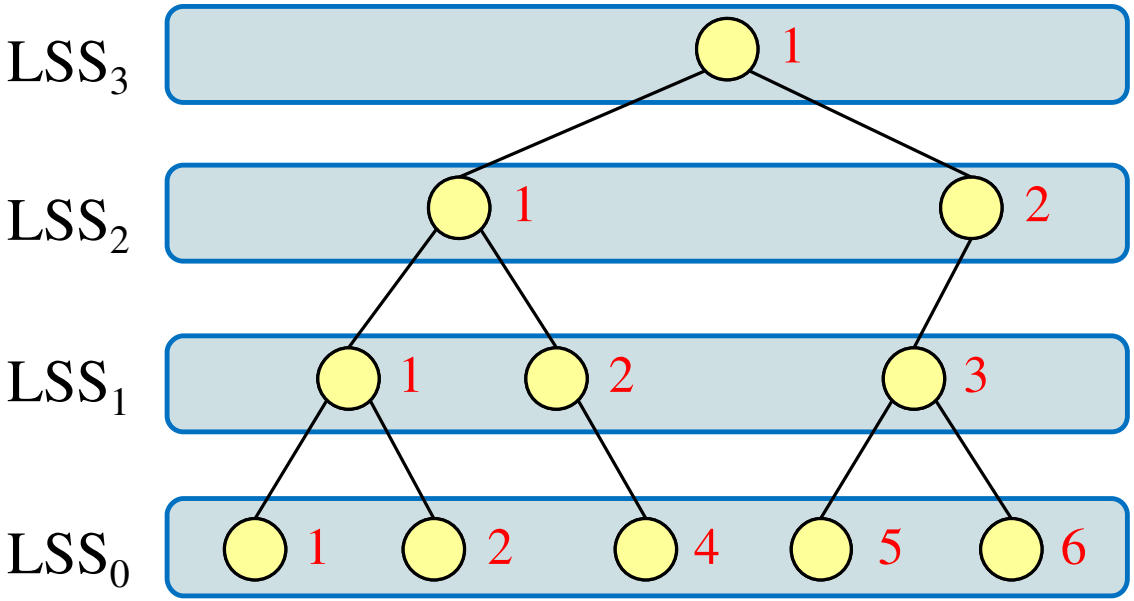


次に、level search structure (LSS) とう補助データ構造を導入します。

各高さ j について、LSS_j は高さ j の頂点の id のリストです。
この図の例では
LSS_0 は 1, 2, 4, 5, 6 のリスト
LSS_1 は 1, 2, 3 のリスト
LSS_2 は 1, 2 のリスト
LSS_3 は 1 のリスト
です。

level search structure (LSS)

- LSS_j: 高さ j の頂点の id のリスト



ひとまず，各 LSS_j に対する member クエリを
$O(1)$ 時間で計算できると仮定する（詳細は後述）.

ここで、ひとまず、各 LSS_j に対する member クエリを定数時間で計算できると仮定しましょう。

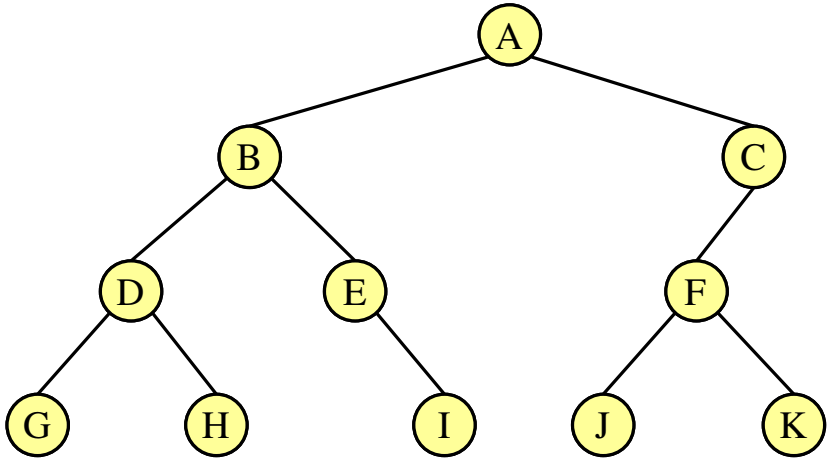
この過程に関する詳細は、この講義の最後で述べます。

bottom関数

- 高さ j の頂点 v が 整数 $x \in U$ の祖先である

$$\Leftrightarrow \quad \left\lceil \frac{x}{2^j} \right\rceil = \mathrm{id}(v) \qquad \text{（注：} x \text{ は } S \text{ の要素とは限らない）}$$

- $\mathrm{bottom}(x) = x$ の最も低い祖先



- 1 2 4 5 6

$\mathrm{bottom}(1) = \mathrm{G}$
$\mathrm{bottom}(2) = \mathrm{H}$
$\mathrm{bottom}(3) = \mathrm{E}$
$\mathrm{bottom}(4) = \mathrm{I}$
$\mathrm{bottom}(5) = \mathrm{J}$
$\mathrm{bottom}(6) = \mathrm{K}$
$\mathrm{bottom}(7) = \mathrm{C}$
$\mathrm{bottom}(8) = \mathrm{C}$

最後に、bottom という関数を導入します。
数学的な定義は上に書いた通りですが、直感的には以下の通りです。

整数 x に対して、bottom(x) は x の最も低い祖先です。x が S の要素でないときは、仮に x がそこにあったとした場合の最も低い祖先を返します。

この例でいうと、1 は S の要素なので、1 に対応する葉 (G) があります。よって、bottom(x) = G です。

一方、3 は S の要素ではないです。ここで、仮に 3 が 2 と 4 の間にあったとすると、そこから木を上がっていって最初に当たる頂点は E です。よって、bottom(3) = E となります。

S の他の要素についても同様です。確認してみてください。

bottom関数の計算 [1/3]

補題

任意の整数 $x \in U$ に対して，
$\mathrm{bottom}(x)$ を $O(\log\log u)$ 時間で計算できる．

【アルゴリズムの概要】

- x-fast trie の高さの2分探索を行う．
- 高さ j の LSS_j に対して $\left\lceil \frac{x}{2^j} \right\rceil$ に関する member クエリを行う．

この bottom 関数について、次の補題が成立します。
U の任意の要素 x に対して、bottom(x) を O(log log u) 時間で計算できます。

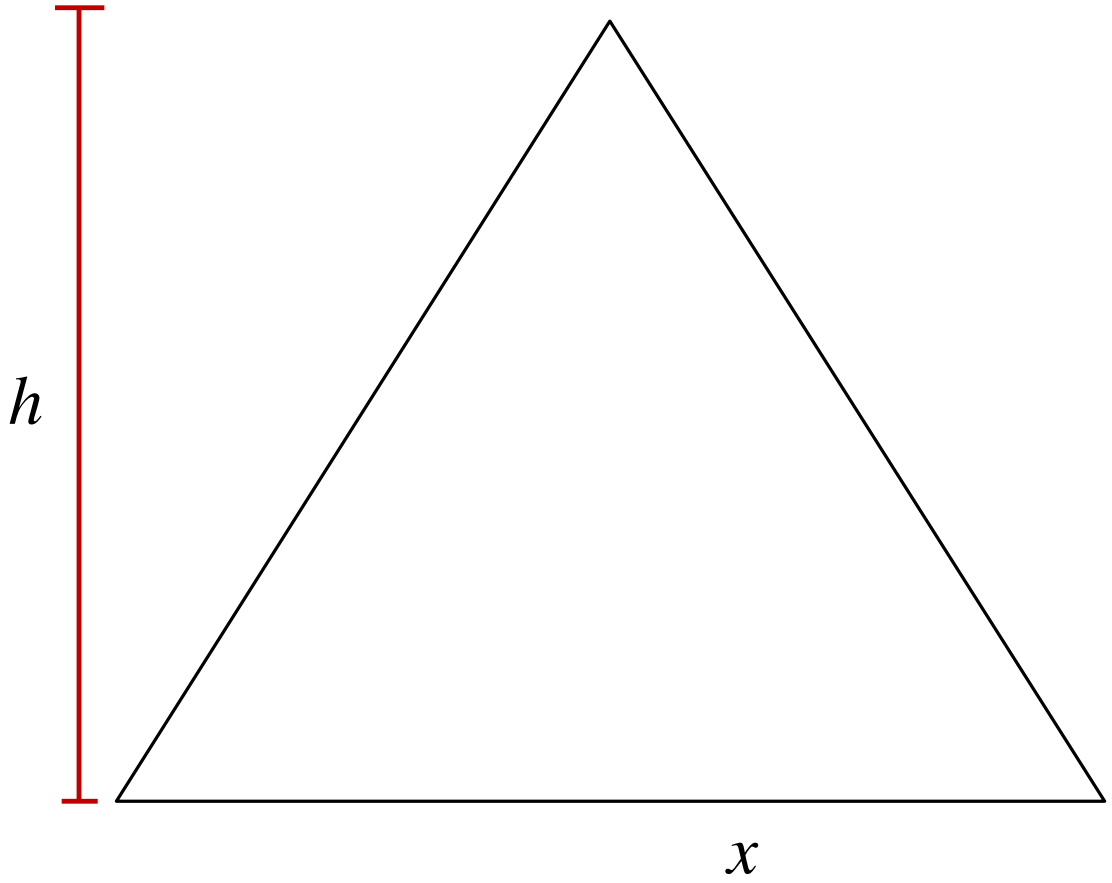
計算方法の概要は以下の通りです。

x-fast trie の高さの2分探索を行います（これは前回の van Emde Boas データ構造を木としてみなしたときの解析と似た話です）

2分探索で訪れた各高さ j について、LSS_j に対して x/2^j を繰り上げた値に関する member クエリを行います。

次のスライドでイメージ図を示します。

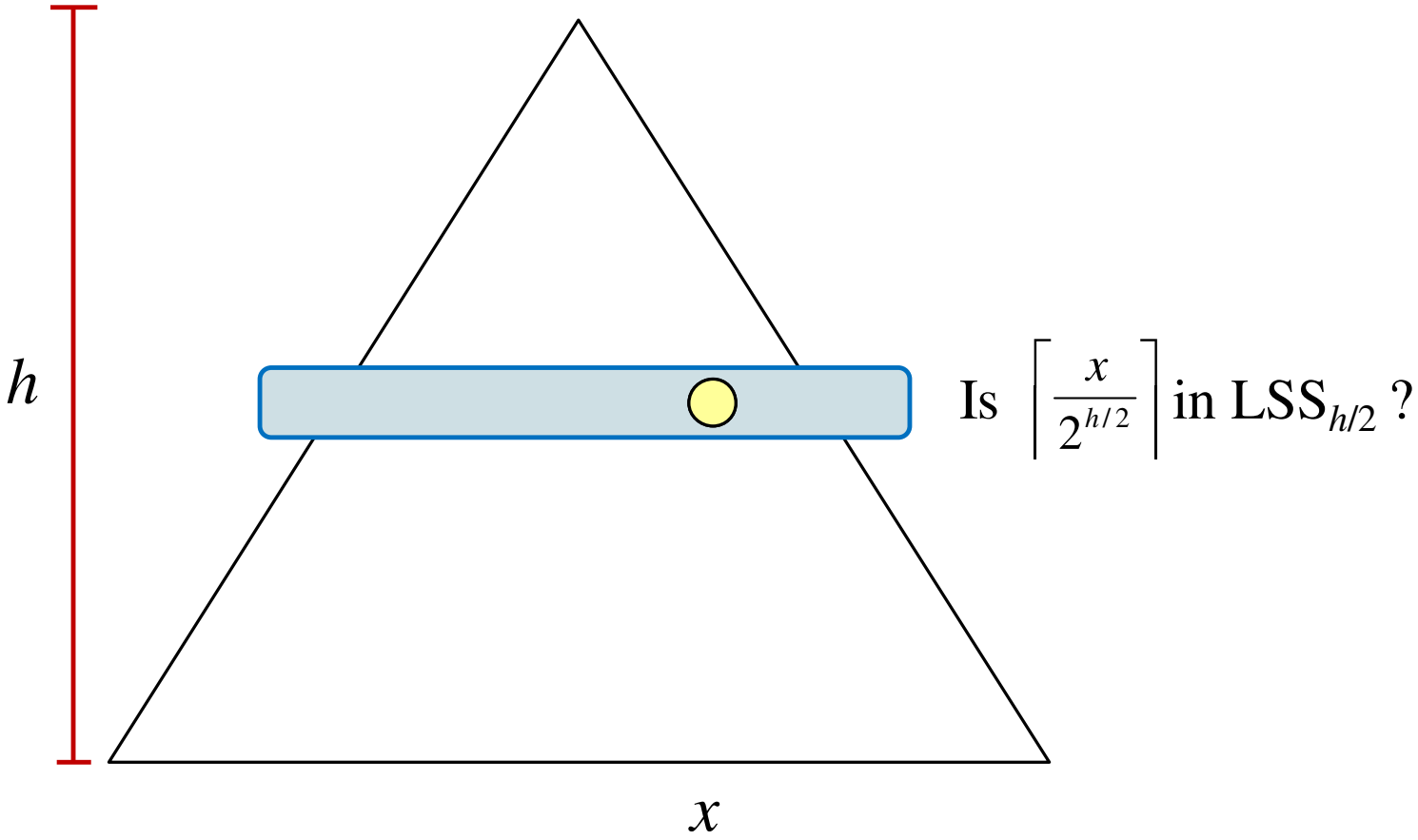
bottom関数の計算 [2/3]



この三角形は x-fast trie の模式図です。その高さを h とします。

いま、与えられた整数 x に対して bottom(x) を計算しようとしています。

bottom関数の計算 [2/3]

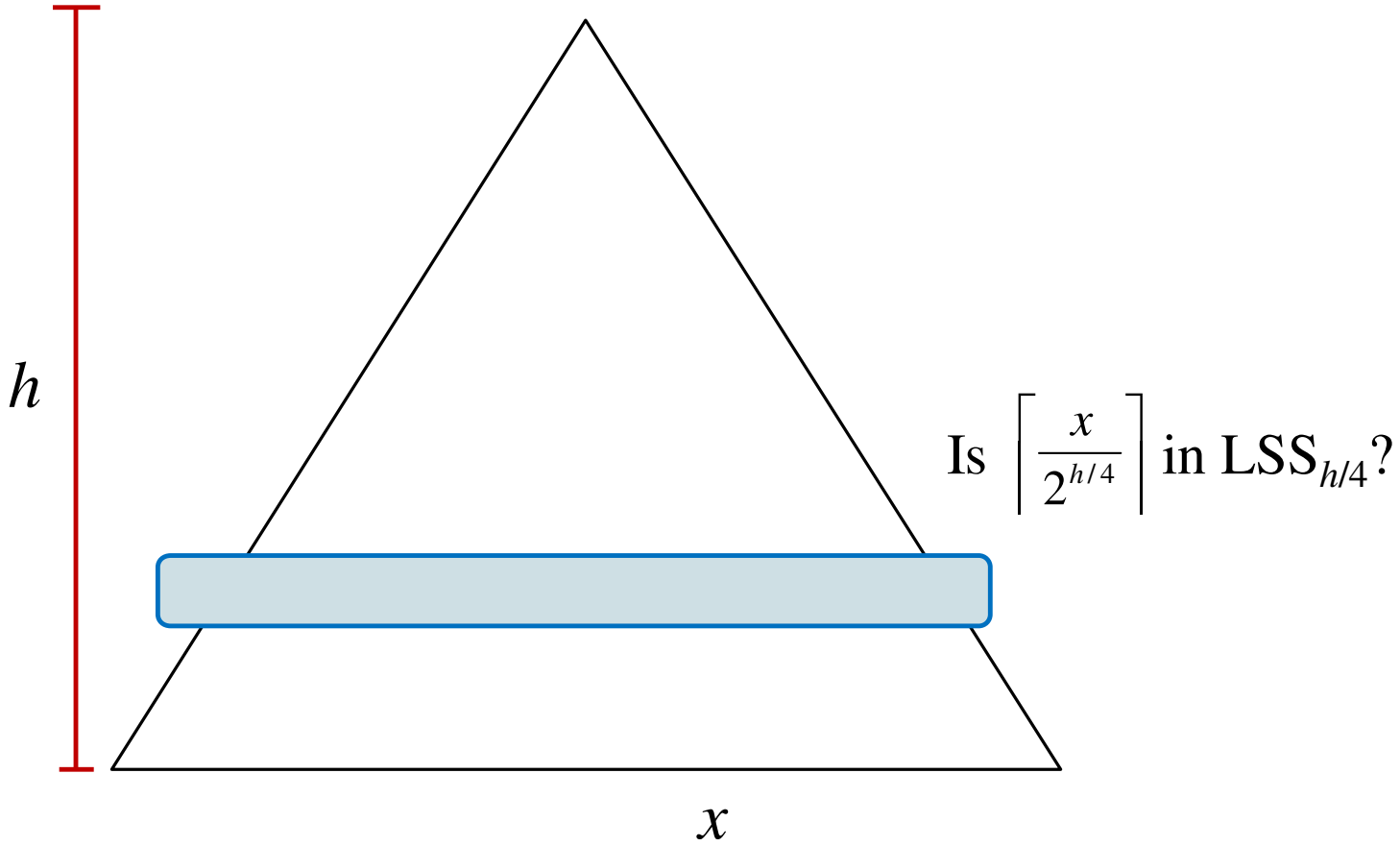


高さの2分探索を行うので、まず半分の高さ h/2 にアクセスし、LSS_{h/2} における x/2^{h/2} を繰り上げた値の member クエリを行います。

member クエリの答えが「YES」だったとします（つまり、図の黄色の○が x/2^{h/2} を繰り上げた値）

bottom(x) はより下にある可能性があるので、2分探索を継続します。

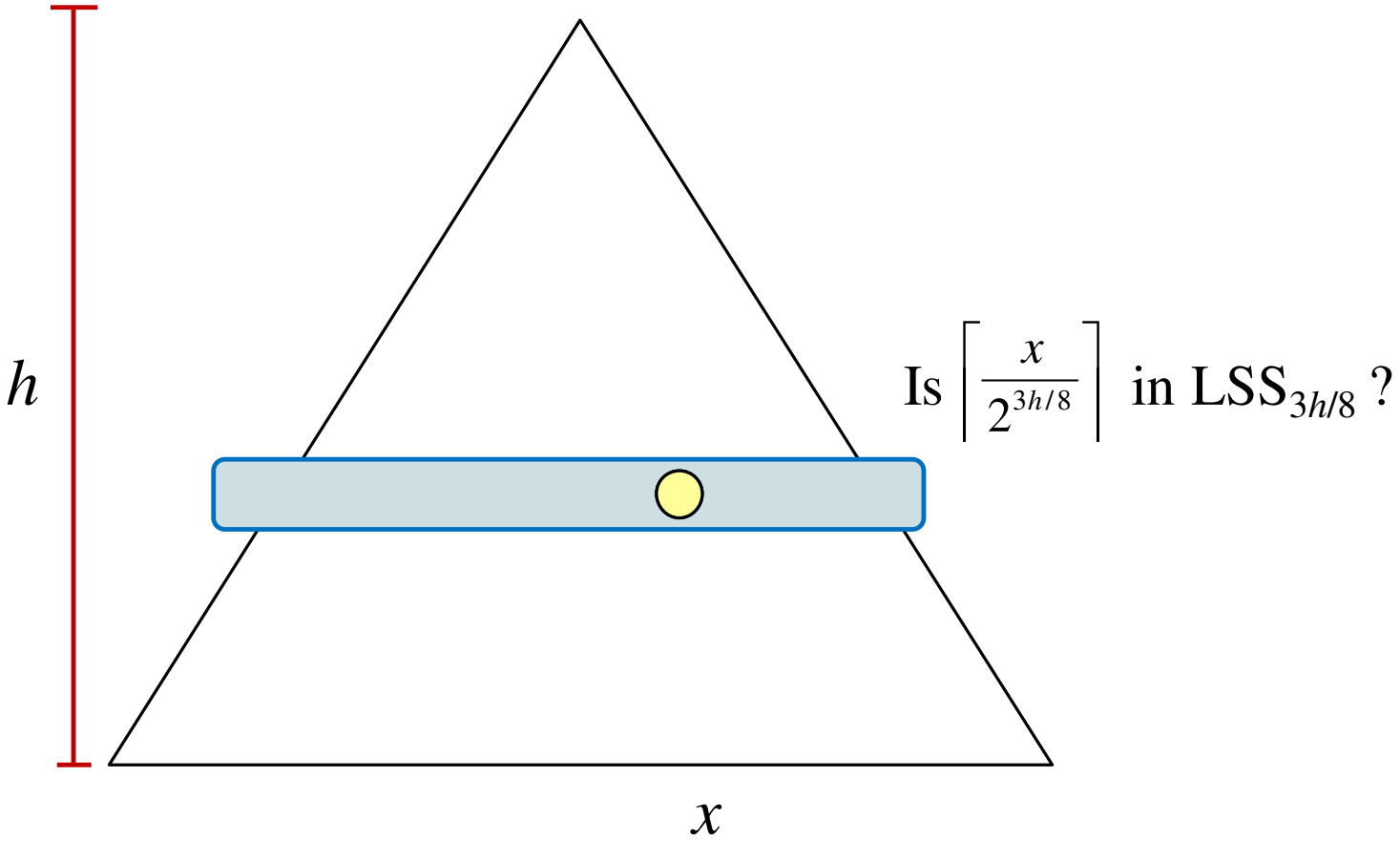
bottom関数の計算 [2/3]



高さ h/4 に対して同様に LSS_{h/4} に member クエリを行ったところ、答えが「NO」だったとしましょう。

bottom(x) はもっと上にあるはずなので、2分探索を継続します。

bottom関数の計算 [2/3]



次に高さ 3h/8 で同様の member クエリを行います。

このようにしていくことで、x の最も低い祖先 bottom(x) を見つけることができます。

bottom関数の計算 [3/3]

補題

任意の整数 $x \in U$ に対して，
bottom(x) を $O(\log\log u)$ 時間で計算できる．

【証明の概要】

- 2分探索の探索回数 ➔ $O(\log\log u)$（trie の高さは $O(\log u)$）
- LSS に対する member クエリ ➔ $O(1)$ 時間（仮定より）
- 全体で $O(\log\log u) \times O(1) = O(\log\log u)$ 時間

この補題の時間計算量を検証します。

x-fast trie の高さは h = log_2 u なので、2分探索のステップ数は O(log h) = O(log log u) です。

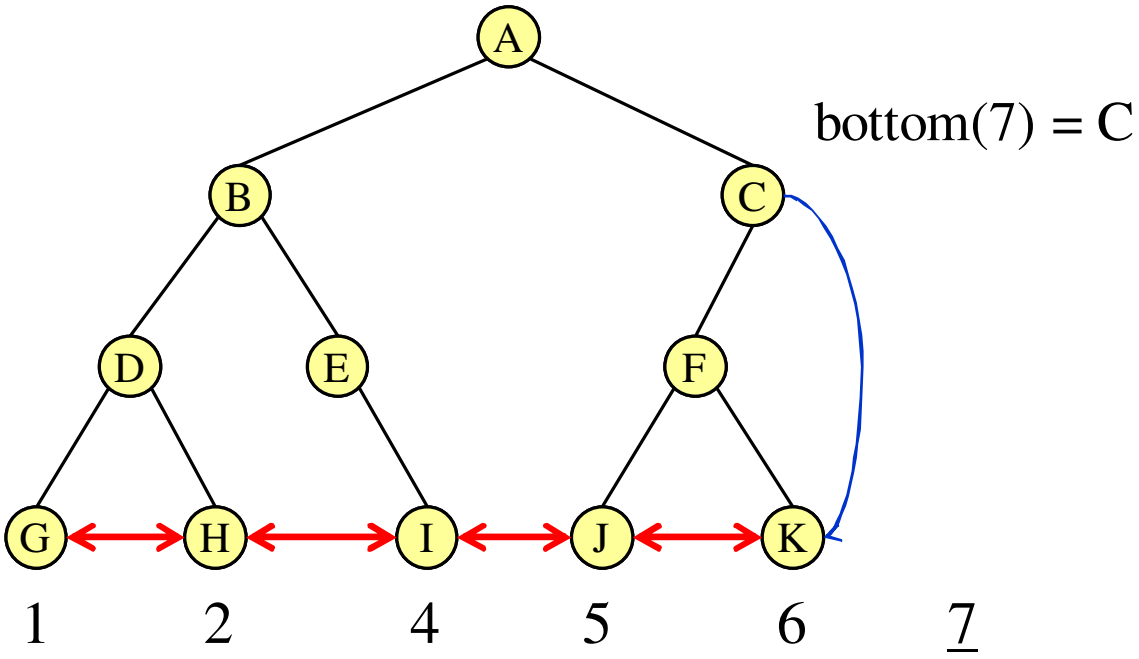
仮定より、LSS に対する毎回の member クエリに要する時間は O(1) です。

よって、合計 O(log log u) 時間で bottom(x) を計算することができます。

以上で証明終わりです。

predecessor クエリwith x-fast trie

1. bottom(x) が右の子を持たないとき



それでは、x-fast trie を用いた predecessor(x, S) の計算方法に入っていきます。

x が与えられたら、まず前述の補題の方法を用いて bottom(x) を計算します。

次のステップでは、場合分けが3つあります。

場合１: bottom(x) が右の子を持たない場合、を考えます。

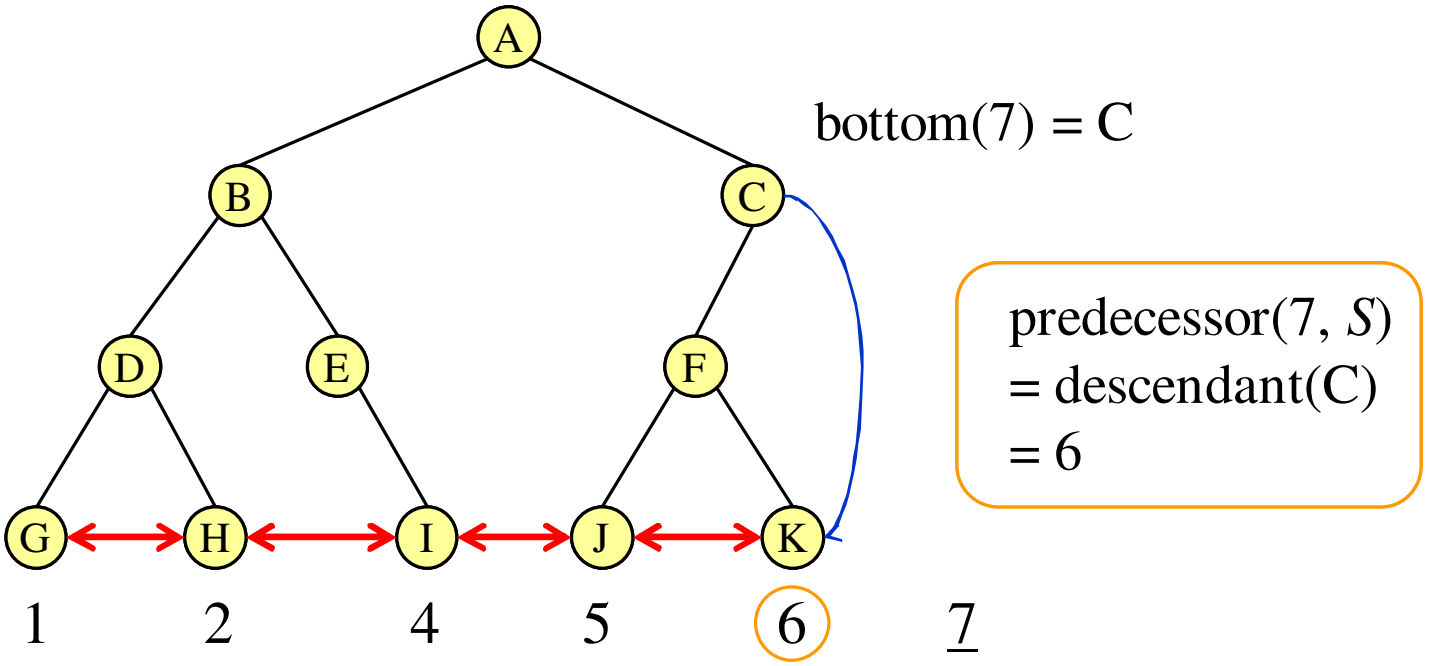
この例では、predecessor(7, S) を探しています。
bottom(7) = C であり、C は右の子を持ちません。

このとき、descendant(C) のリンクを辿って、6 を格納する葉 K にアクセスします。

（次ページに続く）

predecessor クエリwith x-fast trie

1. bottom(x) が右の子を持たないとき

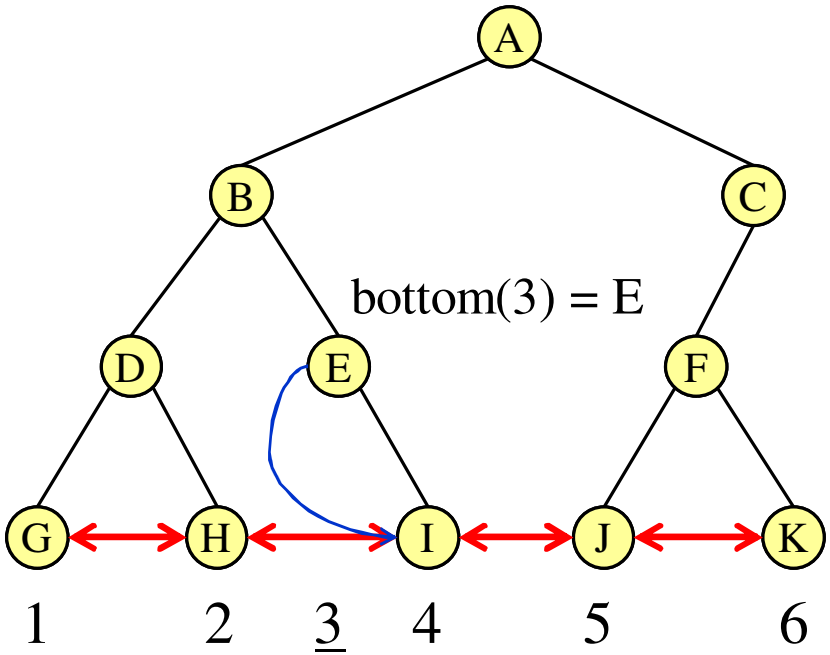


すると、ここが
precedessor(7, S) の答えに
なっています。

以上が場合1です。

predecessor クエリwith x-fast trie

2. bottom(x) が左の子を持たないとき



場合２：bottom(x) が左の子を持たないとき、を考えます。

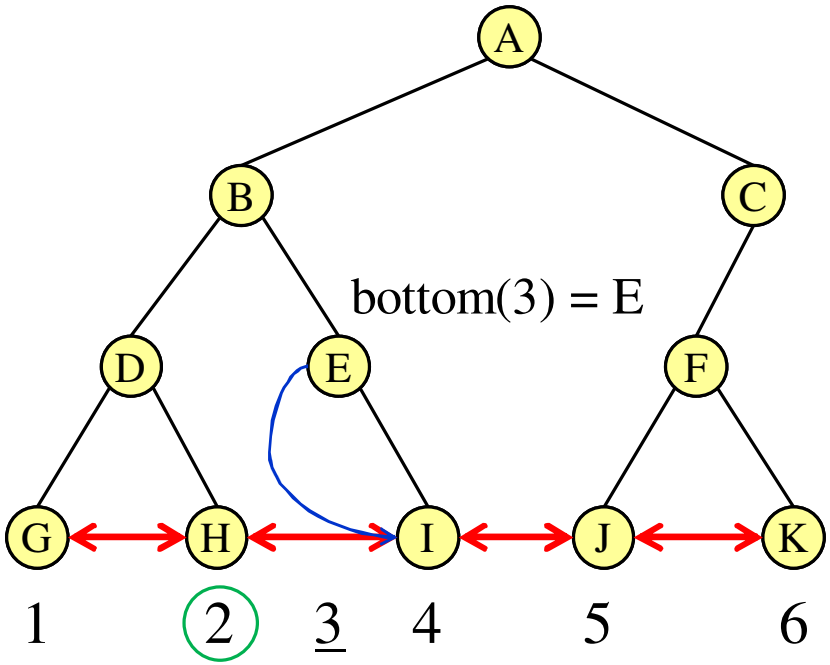
いま、predecessor(3, S) を探しているとしましょう。

まず bottom(3) = E を求めます。E は左の子を持ちません。

（次ページに続く）

predecessor クエリwith x-fast trie

2. bottom(x) が左の子を持たないとき



predecessor(3, S)
= descendant(E)の左隣り
= 4の左隣り
= 2

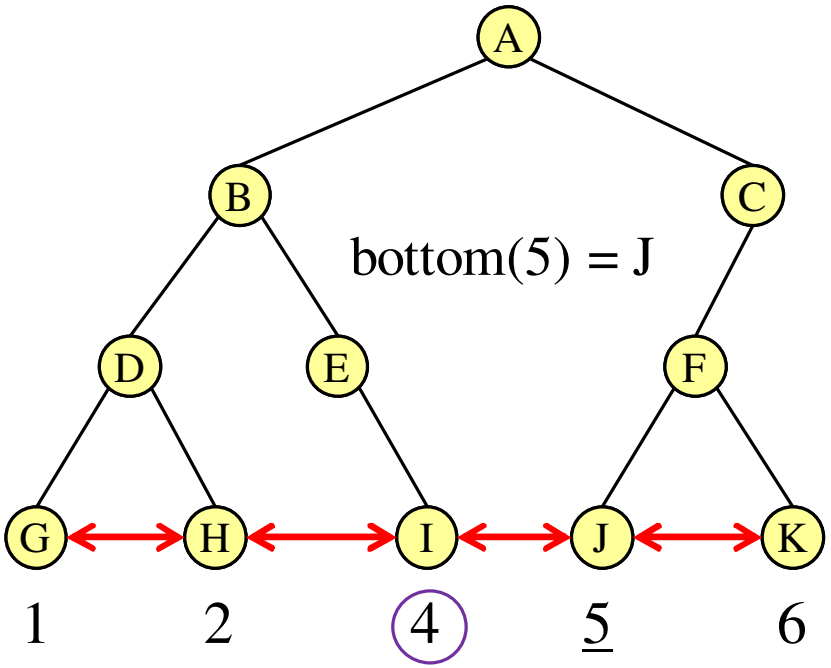
そこで、descendant(E) を辿り、まず 4 を格納する葉 I にアクセスします。

続いて、葉の双方向連結リストを辿って、4 の左隣りの葉 H に移動します。ここに格納されている 2 が predecessor(3, S) の答えです。

以上が場合2です。

predecessor クエリwith x-fast trie

3. bottom(x) が子を持たないとき $\Leftrightarrow$ $x \in S$ のとき



predecessor(5, S)
= 5の左隣り
= 4

場合３：bottom(x) が子を持たないとき、を考えます。

木の頂点の中で、子を持たないのは葉のみです。よって、場合3は bottom(x) が葉である、つまり x が S の要素である場合です。

よって、bottom(x) に対応する葉（図の例では 4）から双方向連結リストを用いて、左隣りの葉に移動すればよいことになります。

場合3は以上です。

predecessor クエリの時間計算量

定理

x-fast trie を用いることにより，
predecessor(x, S) を $O(\log\log u)$ 時間で計算できる

【証明の概要】

- bottom(x) の計算 ➔ $O(\log\log u)$ 時間（補題より）
- 各頂点 v に対する descendant(v) の計算 ➔ $O(1)$ 時間
- 隣の葉への移動 ➔ $O(1)$ 時間（双方向リスト）
- よって，全体で $O(\log\log u) + O(1) + O(1) = O(\log\log u)$ 時間

以上をまとめて、この定理を得ます。

定理：x-fast trie を用いることによって、predecessor(x, S) を O(log log u) 時間で計算できます。

証明の概要です。

補題より、bottom(x) を O(log log u) 時間で計算できます。

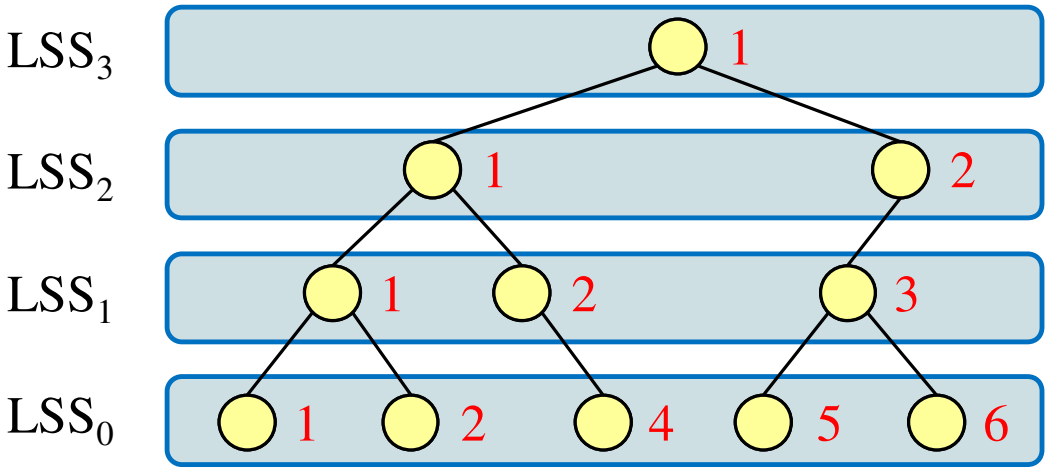
各頂点 v に対する descendant は、リンクを辿るだけなので定数時間です。

隣の葉への移動も、双方向連結リストを辿って定数時間で行えます。

よって、全体で O(log log u) 時間となります。

x-fast trie の領域計算量 [1/2]

- ひとまず，要素数 k の整数集合に対して，
  $O(1)$ 時間で member クエリに答えることができる
  $O(k)$ 領域の LSS データ構造があると仮定（詳細は後述）



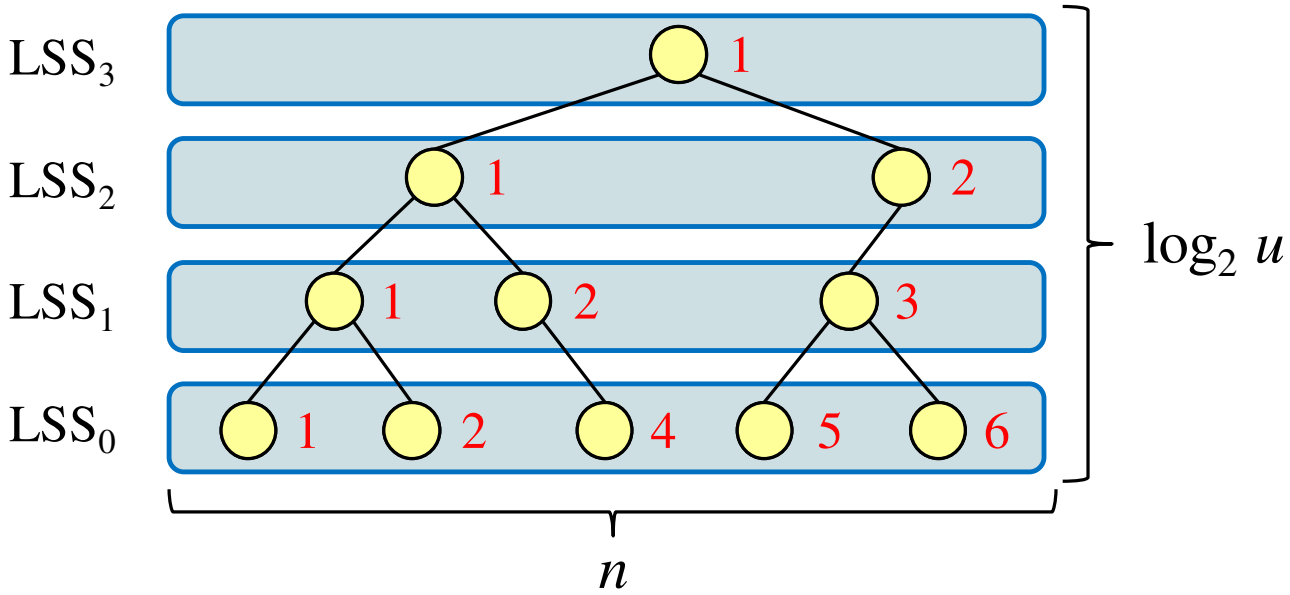
次に、x-fast trie の領域について考察します。

ひとまず，要素数 k の整数集合に対して，O(1) 時間で member クエリに答えることができる O(k) 領域の LSS データ構造があると仮定して話を進めていきましょう。

x-fast trie の領域計算量 [2/2]

定理

x-fast trie の領域計算量は $O(n \log u)$



この仮定のもとで、x-fast trie の領域計算量は O(n log u) で抑えられることが簡単に示せます。

下の図を見てみましょう。

葉の個数は明らかに n = |S| です。また、x-fast trie の高さは log_2 u です。

葉から1段ずつ高さを上がっていくにつれて、LSS に格納する頂点数は明らかに単調非減少します。

よって、x-fast trie の頂点数は O(n log u) で上から抑えられます。

前ページの仮定より、x-fast trie の領域は O(n log u) で抑えれれます。

証明は以上です。

演習問題

- 前ページの図を見ると，高さが上がるにつれて，LSSの要素数は少なくなっている．
- もしかすると，x-fast trie の領域計算量の見積り $O(n \log u)$ は緩いのかもしれない．
- 【問】上記の $O(n \log u)$ という領域計算量の見積りは厳密だろうか？
  厳密であればその理由を示し，
  そうでなければ，より厳密なオーダーを示せ．

※ 演習問題 **提出〆切：7月16日（木）23：59**

では、今週の演習問題です。

前ページで解説したように、高さが上がるにつれて、LSS の要素数は単調非減少していき、いつかは1になります（根の高さには1つしか頂点がないため）

この観察によれば、前ページの x-fast trie の領域の見積もり O(n log u) は緩い上界なのかもしれません。

では、ここで演習問題です。

上記の O(n log u) という領域計算量の見積りは厳密でしょうか？
厳密であればその理由を示し、そうでなければ より厳密なオーダーを示してください。

演習の提出〆切は2週間後の7／16です。

y-fast trie [1/2]

- y-fast trie：x-fast trie の領域計算量を $O(n \log u)$ から $O(n)$ に改良したもの.
  - 第４回/第５回で紹介した p-fast trie と q-fast trie の関係と同じ.
- S をそれぞれ $\Theta(\log u)$ サイズの部分集合に分割する.
- 各部分集合 S_i について，平衡２分探索木を作る.
- 平衡２分探索木の頂点の集合 T に対する x-fast trie を作る.
- S に対する y-fast trie は，この x-fast trie と，各部分集合 S_i に対する平衡２分探索木で構成される.

では、いよいよ本日の本題である y-fast trie を解説していきます。

前述したように、y-fast trie は、x-fast trie の領域計算量を O(n log u) から O(n) に改良したデータ構造で、第４回/第５回で紹介した p-fast trie と q-fast trie の関係と同様のアイディアを用います。

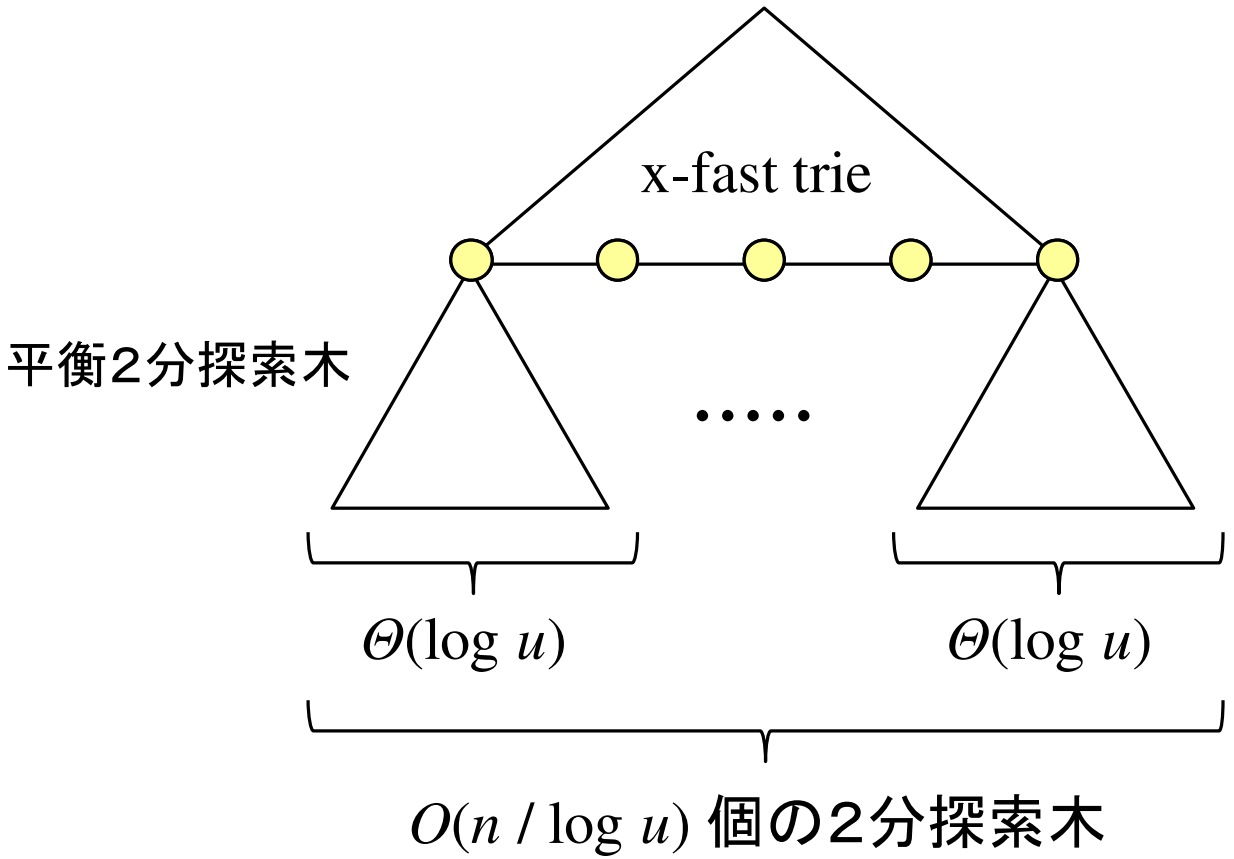
まず、S をそれぞれ Θ(log u) サイズの部分集合に分割します。

次に、各部分集合 S_i について、平衡2分探索木を作ります。

この平衡2分探索木の頂点の集合 T に対する x-fast trie を作ります。

S に対する y-fast trie は、この x-fast trie と、各部分集合 S_i に対する平衡2分探索木から成ります。

y-fast trie [2/2]



y-fast trie を図示するとこのような感じになります。

各2分探索木は Θ(log u) 個の要素を含むので，2分探索木の個数は O(n / log u) です。

よって、トップにある x-fast trie が管理する集合 T の要素数（この図の黄色い頂点の数）も O(n / log u) です。

y-fast trie の領域計算量

定理

y-fast trie の領域計算量は $O(n)$

【証明の概要】

- サイズ $\Theta(\log u)$ の 平衡2分探索木 × $O(n / \log u)$ 個
  $= O(n)$ 領域
- サイズ $O(n / \log u)$ の集合に対する x-fast trie
  $= O((n / \log u) \log u) = O(n)$ 領域
- 全体で $O(n)$ 領域

この定理にあるように、y-fast trie の領域は O(n) であることが示せます。

Θ(log u) サイズの平衡2分探索木が O(n / log u) 個あるので、これらの平衡2分探索木の合計領域は O(n) です。

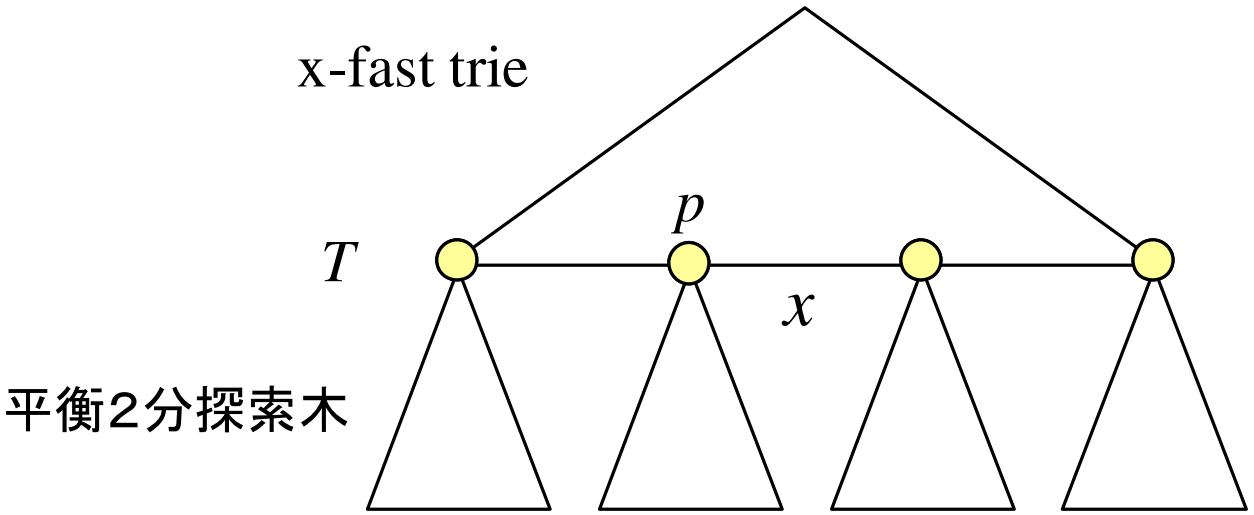
要素数 O(n / log u) の集合に対する x-fast trie のサイズは O((n / log u) log u) = O(n) となります。

よって、全体で O(n) 領域です。

証明は以上です。

predecessor クエリ with y-fast trie

1. x-fast trie を用いて p = predecessor(x, T) を求める

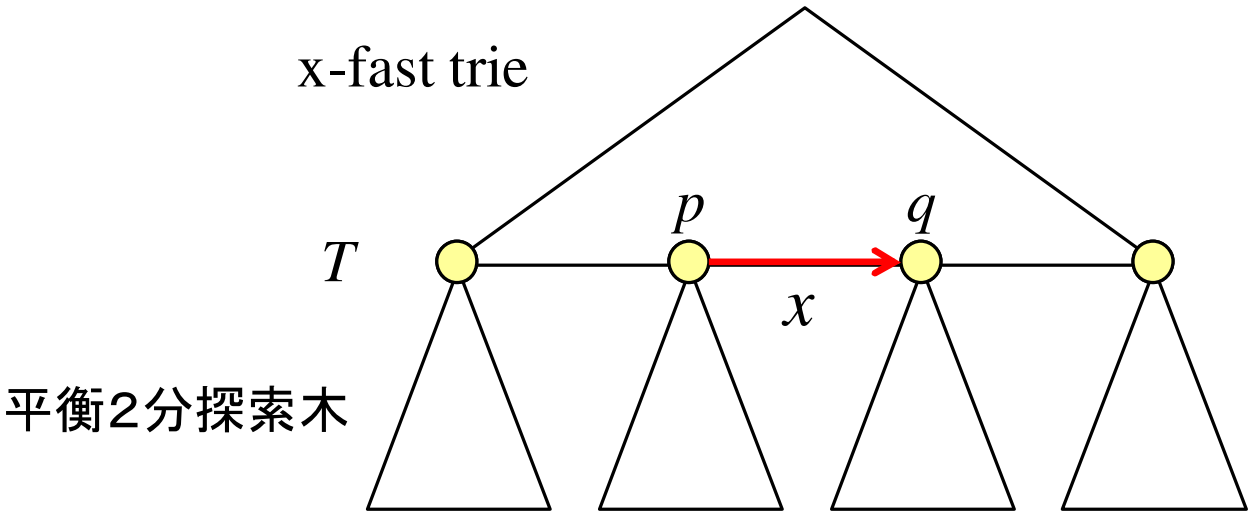


続いて、y-fast trie を用いた predecessor(x, S) の計算方法を解説します。

まず、トップにある x-fast trie を用いて predecessor(x, T) を求め、その答えを p とします。

predecessor クエリ with y-fast trie

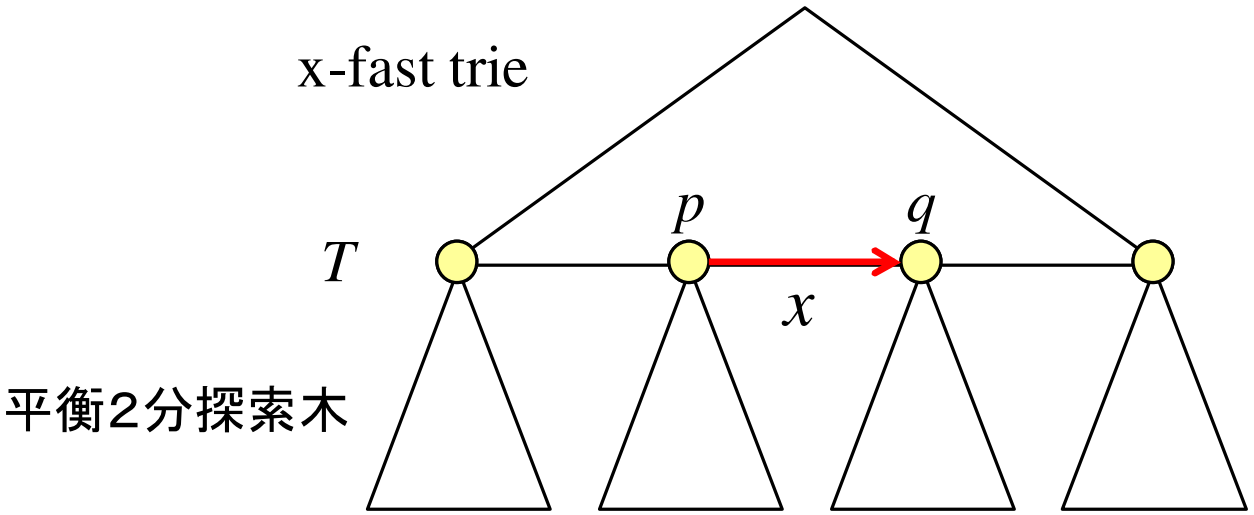
2. 連結リストを用いて q = successor(p, T) を求める



次に、x-fast trie の葉の双方向連結リストを用いて successor(p, T) を求め、これを q とします。

predecessor クエリ with y-fast trie

3. p と q を根とする 平衡2分探索木 から predecessor(x, S) を求める



最後に、p と q を根とする平衡2分探索木から、predecessor(x, S) を求めます。

クエリのアルゴリズムは以上です。

predecessor クエリの時間計算量

定理

y-fast trie を用いることにより,
predecessor(x, S) を $O(\log\log u)$ 時間で計算できる.

1. x-fast trie を用いて p = predecessor(x, T) を求める.
   - $O(\log\log u)$ 時間
2. 連結リストを用いて q = successor(p, T) を求める.
   - $O(1)$時間
3. p と q を根とする 平衡2分探索木 から
   predecessor(x, S) を求める.
   - $O(\log\log u)$ 時間（高さ $O(\log\log u)$ の2分木の探索）

この方法で、predecessor クエリに O(log log u) 時間で応答できます。

x-fast trie の predecessor 探索時間は O(log log u) です。

葉の連結リストを定数時間で辿ったあと、2つの平衡2分探索木から predecessor(x, S) を求めます。
各2分探索木の要素数は O(log u) なので、その高さは O(log log u) です。よって、この処理も O(log log u) 時間で完了します。

以上をすべて合計して、O(log log u) 時間となります。

証明は以上です。

LSSの実現

- とりあえず，member クエリの計算を
  $O(1)$ 時間＆$O(n)$ 領域と仮定していたが・・・
- 配列：member クエリ $O(1)$ 時間，$O(u)$ 領域
- リスト：member クエリ $O(n)$ 時間，$O(n)$ 領域

と、ここまで議論を進めてきましたが、これまでの定理はあくまでも LSS における仮定の上で成り立っていました。

集合の要素数を n とすると、member クエリの計算を O(1) 時間かつ O(n) 領域で実現できる LSS データ構造がある、という仮定です。

もし、スタンダードな配列を LSS に使ったとすると、member クエリは O(1) 時間で実現できますが、配列の領域は全体集合のサイズ u の線形 O(u) になってしまいます。

リストを LSS に使ったとすると、領域は O(n) で収まりますが、member クエリには O(n) 時間を要してしまいます。

LSSの実現

- とりあえず，member クエリの計算を
  $O(1)$ 時間＆$O(n)$ 領域と仮定していたが・・・
- 配列：member クエリ $O(1)$ 時間，$O(u)$ 領域
- リスト：member クエリ $O(n)$ 時間，$O(n)$ 領域
- cuckoo hash（Pagh & Rodler, 2001）：
  member クエリ $O(1)$ 時間，$O(n)$ 領域

そこで、次回は member クエリを O(1) 時間で実現する O(n) 領域のデータ構造として、cuckoo hash というデータ構造を紹介します。

これを LSS に使用することで、前ページまでの定理が成立します。