p-fast tries

2020年度前期　高度データ構造

本日は、p-fast trie というデータ構造を取り扱います。

比較モデルにおける insert の計算限界

- 【第2回参照】AVL tree を用いて，整数集合 S 上の各クエリ・操作を $O(\log n)$ 時間で実現できる（$n = |S|$）.
- 2分探索木を用いた insert は $O(\log n)$ が最適時間である（つまり，これ以上速くできない）.
- なぜ？
  - S の2分探索木を構築するために，n 回の insert 操作が必要.
  - 比較ソートの時間計算量の下界は $\Omega(n \log n)$ である.
  - AVL tree を深さ優先探索すると，S のソート列が得られるので，AVL tree の構築には少なくとも $\Omega(n \log n)$ 時間が必要.
  - よって，1回の insert 操作につき $\Omega(\log n)$ 時間が必要.

第2回の講義において、整数集合 S 上の各クエリ・操作を AVL tree を用いて $O(\log n)$ 時間で実現できることを示しました。ここで n は S の要素数です。

AVL tree は要素の大小比較に基づく2分探索木です。一方、大小比較に基づくいかなる2分探索木を用いても、insert を O(log n) より速くすることはできません。

つまり、比較モデルにおいては、AVL tree の O(log n) 時間 insert が最適だということになります。

なぜでしょう？ 以下に理由を説明します（次スライドへ）

比較モデルにおける insert の計算限界

- 【第2回参照】AVL tree を用いて，整数集合 S 上の各クエリ・操作を $O(\log n)$ 時間で実現できる（$n = |S|$）.
- 2分探索木を用いた insert は $O(\log n)$ が最適時間である（つまり，これ以上速くできない）.
- なぜ？
  - S の2分探索木を構築するために，n 回の insert 操作が必要.
  - 比較ソートの時間計算量の下界は $\Omega(n \log n)$ である.
  - AVL tree を深さ優先探索すると，S のソート列が得られるので，AVL tree の構築には少なくとも $\Omega(n \log n)$ 時間が必要.
  - よって，1回の insert 操作につき $\Omega(\log n)$ 時間が必要.

集合 S の2分探索木を構築するためには、n 回の insert 操作が必要です。

一方、比較ソートの時間計算量の下界は Ω(n log n) であることが広く知られています。

AVL tree を深さ優先探索すると、S のソート列が得られるので、AVL tree の構築には少なくとも Ω(n log n) 時間が必要ということになります。

よって，1回の insert 操作につき Ω(log n) 時間が必要ということになります。

高速な整数データ構造

$O(\log n)$ よりも高速なクエリ・操作を実現するために，
word RAM モデル上でのデータ構造を考える．
→ word RAM では，配列やビット演算が使える（第1回参照）．

【今回のデータ構造】

- 全体集合を $U = \{0, 1, 2, ..., u\text{-}1\}$ とし，$S \subseteq U$ とする
- u: U の要素数, n: S の要素数
- p-fast tries [Willard 1984]: $O(n\sqrt{\log u}\,2^{\sqrt{\log u}})$ 領域
- q-fast tries [Willard 1984]: $O(n)$ 領域
- 各クエリ・操作に要する時間はどちらも $O(\sqrt{\log u})$
  - S が密なとき（e.g., $n = \Theta(u)$ のとき），AVL tree よりも高速

というわけで、要素の大小比較だけを用いたデータ構造では、insert などの操作やクエリを O(log n) 時間よりも速くすることは理論的に不可能です。

そこで、O(log n) よりも高速なクエリ・操作を実現するために、word RAM モデル上でのデータ構造を考えます。

word RAM では、配列やビット演算が使えるため（第1回参照）、これらを利用して高速化を目指すというわけです。

（次スライドへ）

高速な整数データ構造

$O(\log n)$ よりも高速なクエリ・操作を実現するために，
word RAM モデル上でのデータ構造を考える．
→ word RAM では，配列やビット演算が使える（第1回参照）．

【今回のデータ構造】

- 全体集合を $U = \{0, 1, 2, \ldots, u\text{-}1\}$ とし，$S \subseteq U$ とする
- u: U の要素数, n: S の要素数
- p-fast tries [Willard 1984]: $O(n\sqrt{\log u}\,2^{\sqrt{\log u}})$ 領域
- q-fast tries [Willard 1984]: $O(n)$ 領域
- 各クエリ・操作に要する時間はどちらも $O(\sqrt{\log u})$
  - S が密なとき（e.g., $n = \Theta(u)$ のとき），AVL tree よりも高速

全体集合 U を 0 から u−1 までの自然数の集合とし、S を U の部分集合とします。
ここで、u は U の要素数です。また n を S の要素数とします。

今回の講義では、p−fast trie と呼ばれるデータ構造を紹介します。このデータ構造の領域はこのようになっています。

（なお次回の講義では、さらにこれを、要素数 n に線形な O(n) 領域に改善した q−fast trie を紹介します）

各クエリ・操作に要する時間は、どちらのデータ構造も O(√log u) です。
部分集合 S が密なときには、AVL tree よりも高速です。

トライ

- S の各要素が葉に対応するトライを考える
- $U = \{0, 1, 2, \ldots, u\text{-}1\}$
- $S \subseteq U$ に対するトライのサイズを (h, b) で表す
  - h : 高さ
  - b : 各頂点の子の最大個数

では、まず準備としてトライというデータ構造（根付き木）を導入します。

ここでは、S の各要素が葉に対応するトライを考えます。

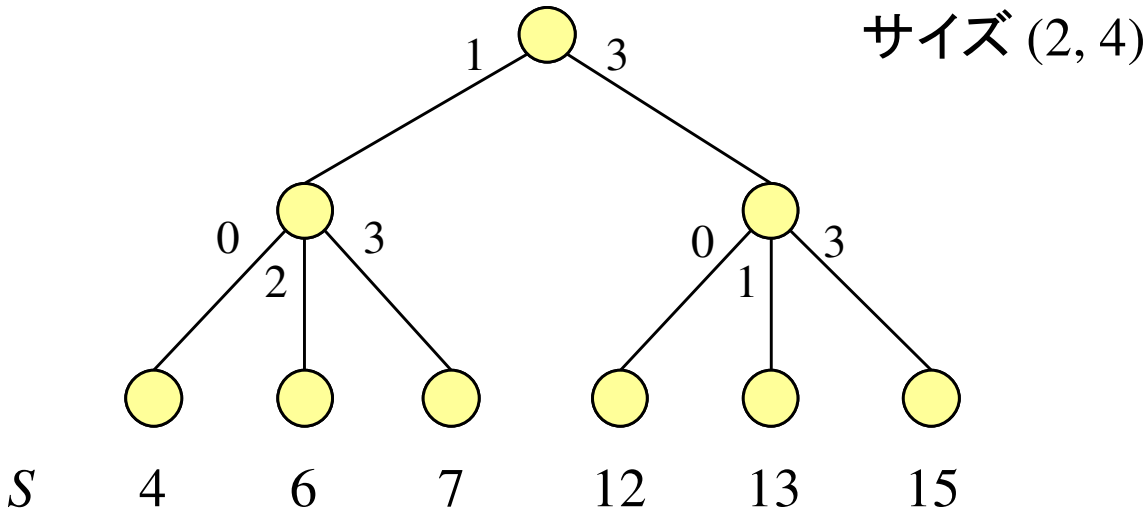
U = {0, 1, 2, …, u−1} に対して、U の部分集合 S に対するトライのサイズを (h, b) で表します。

ここで、h はトライの高さ、bはトライの各頂点の子の最大個数です。

次のスライドに具体例を示します。

(2, 4)トライ

- $U = \{0, 1, 2, 3, ..., 15\}$, $S = \{4, 6, 7, 12, 13, 15\}$



全体集合を U = {0, 1, 2, 3, ..., 15} とし、その部分集合 S = {4, 6, 7, 12, 13, 15} を考えます。

この S に対するサイズ (2, 4) トライはこのようになります。高さ h = 2、子の最大数 b = 4 です。

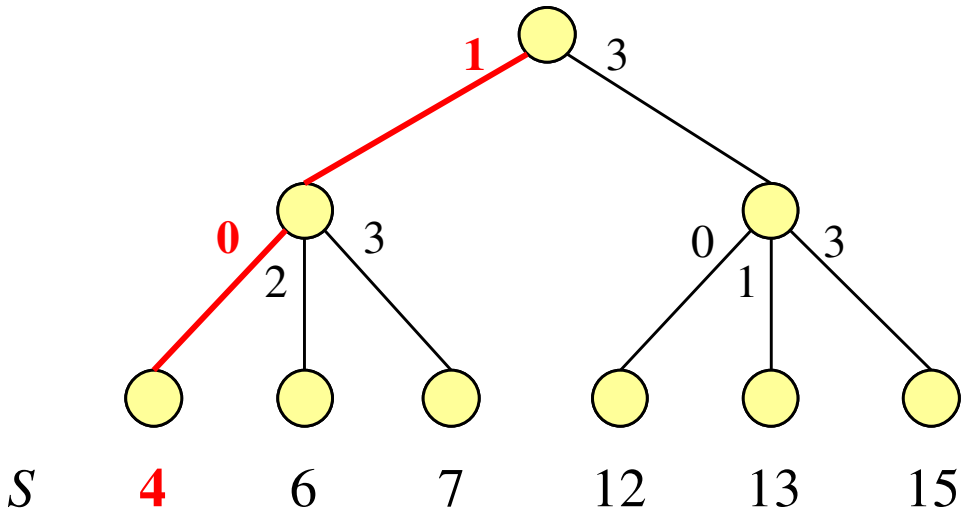
トライの葉と S の要素が1対1対応しています。

このトライでは、それぞれの子への辺に 0 から 3 までの整数を割り当てます。

例えば、左の内部頂点は、0 番目、2番目、3番目の子を持ち、1番目の子は持たない、といった具合です。

(2, 4)トライ

- $U = \{0, 1, 2, 3, \ldots, 15\}$, $S = \{4, 6, 7, 12, 13, 15\}$



$10_4 = 4_{10}$

このようなルールで子への辺をラベル付けしたのには理由があります。

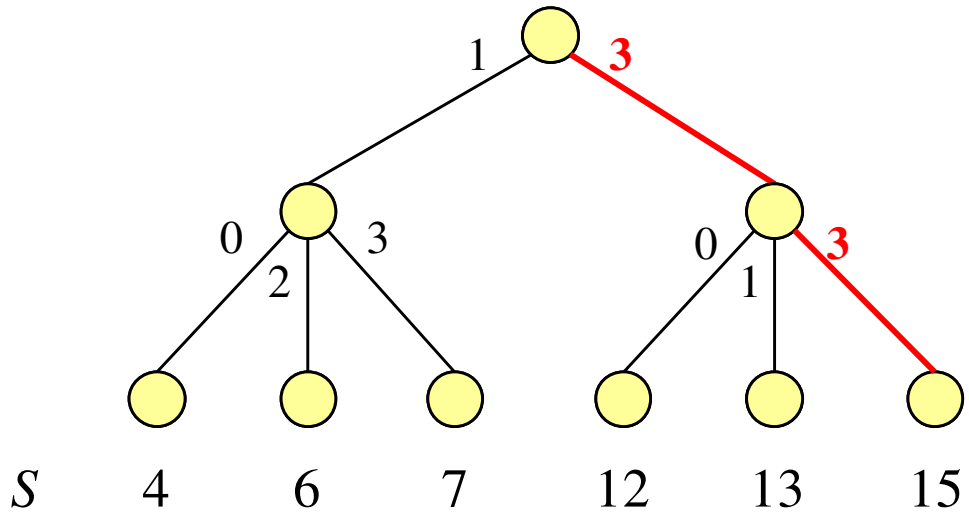
S の要素 4 に対応するこの葉に注目してみましょう。

根からこの葉へのパスラベルは 10 です。

実は、これは4進数の 10 に対応していて(これを 10_4 と書きます)、さらに 10_4 を10進数に変換すると 4(これを 4_{10} と書きます)となり、確かに葉のラベル 4 と対応していることがわかります。

(2, 4)トライ

- $U = \{0, 1, 2, 3, \ldots, 15\}$, $S = \{4, 6, 7, 12, 13, 15\}$



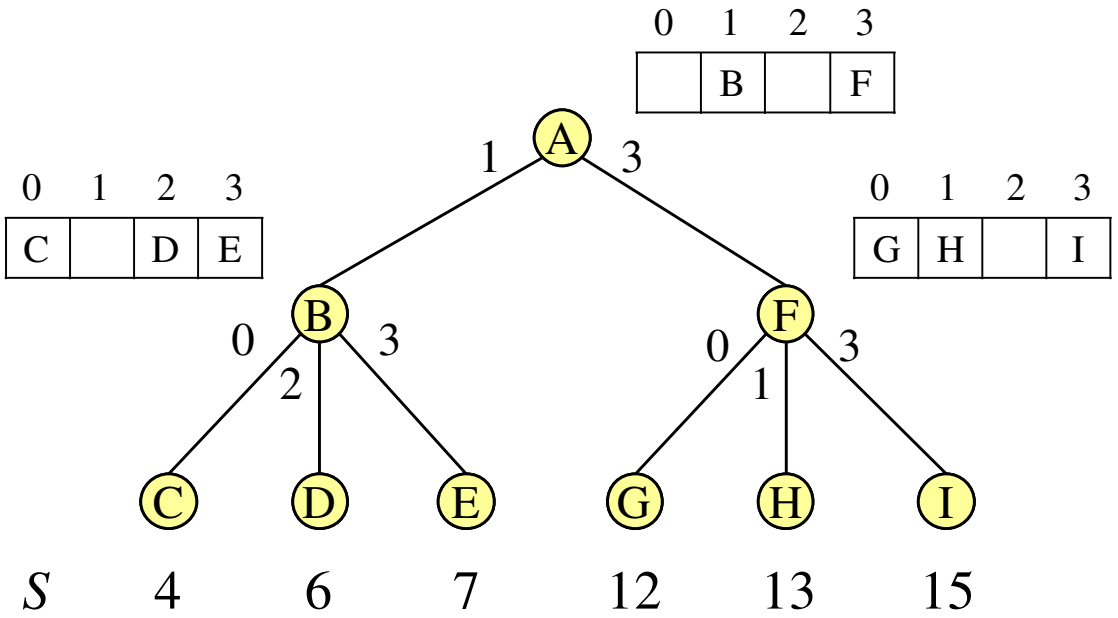
$33_4 = 15_{10}$

他の葉でも同様のことが成り立ちます。

例えば、15 を格納するこの葉について、根からのパスラベルは 33（4進数）であり、その10進数表現は 15 です。

child 配列

- $child_v[i]$:頂点 v の i 番目の子



このトライ上の操作を高速に行うために、各頂点 v に child 配列を保持します。

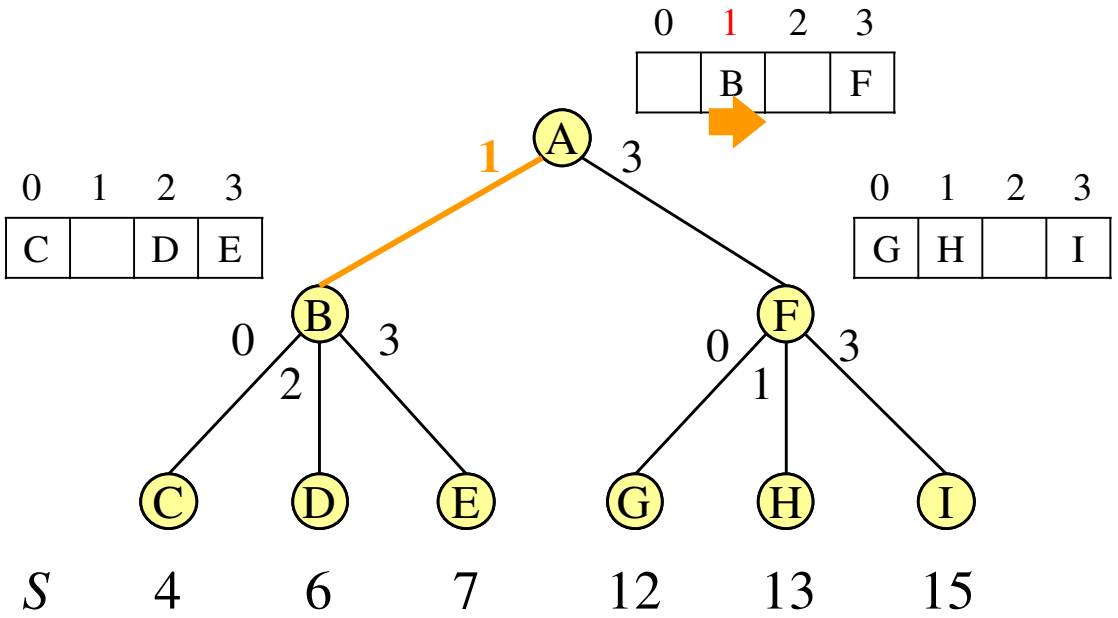
頂点 v の child 配列を $child_v$ と表します。ここで、その i 番目の要素は v の i 番目の子へのポインタを格納します。

この例で説明します。各頂点を区別するために A から I までのラベルを振っています。

頂点 B について、$child_B[0]$ = C, $child_B[2]$ = D, また1番目の子は存在しないので $child_B[1]$ = nil, といった具合です。他の頂点についても同様です。

トライ上での successor クエリ

- successor(5_{10}, S) = successor(11_4, S)



child 配列を用いた素朴な successor クエリを考えてみます。

集合 S における 5 の successor を探します。そのために、5 を4進数に変換し、11 を得ます。

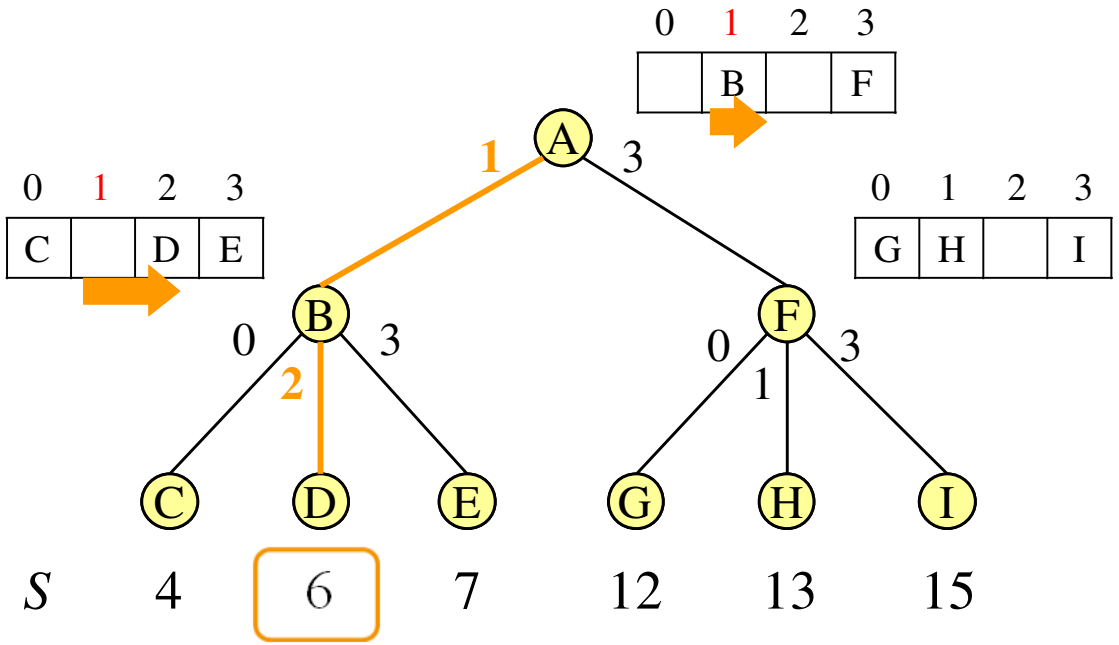
この 11 を使って、根 (A) から降りていきます。

探している 11 の最も左の桁の値が 1 なので、A の child配列 の 1 番目の要素から右に見ていき、最初に見つかった空でないセルに注目します。

ここでは、$child_A[1]$ = B なので、A の1番目の辺を辿って B に降ります。

トライ上での successor クエリ

- successor(5_{10}, S) = successor(11_4, S)



次に、いま探している 11 の左から2番目の値が 1 なので、B の child配列 の 1 番目の要素から右に見ていき、最初に見つかった空でないセルに注目します。

ここで $child_B[2]$ = D が最初に見つかるので、B の2番目の辺を辿って D に降ります。

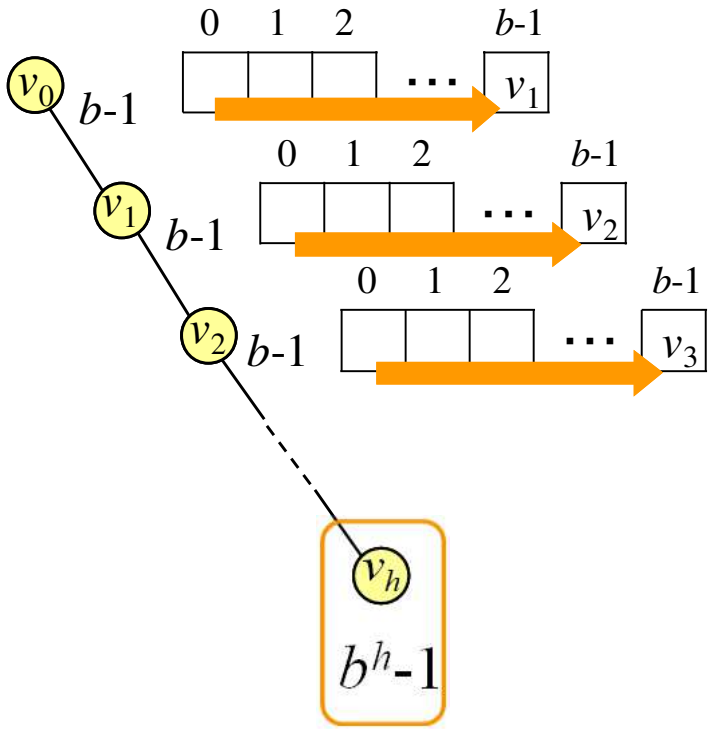
たどり着いた先が葉になったら終了です。いま、D が葉で 6 を格納しているので、5 の successor は 6 だということになります。

確かに S = {4, 6, 7, 12, 13, 15} における 5 の successor は 6 であることを確認しましょう。

この方法は、4進数表現した S の要素の中で、11_4 の直後の値を探しているので、確かに正しく 5_{10} の successor を探していることになります。

トライ上での successor クエリ

- このトライでは
  successor(x, S) に
  $O(bh)$ 時間を要する
- 極端な例）
  $S = \{b^h\text{-}1\}$, $x = 0$ のとき



p-fast trieではこれを
$O(h+\log b)$ 時間に改善

しかしながら、この方法では最悪時には successor クエリに O(bh) 時間を要してしまいます。
b は子の最大数、h はトライの高さです。

例えば、以下のような極端な例がありえます。

S を $\{b^h-1\}$ だけからなる集合とし、x = 0 の successor を探したいとしましょう。
ここで、b^h-1 は (h, b) トライが表現できる最大値です。

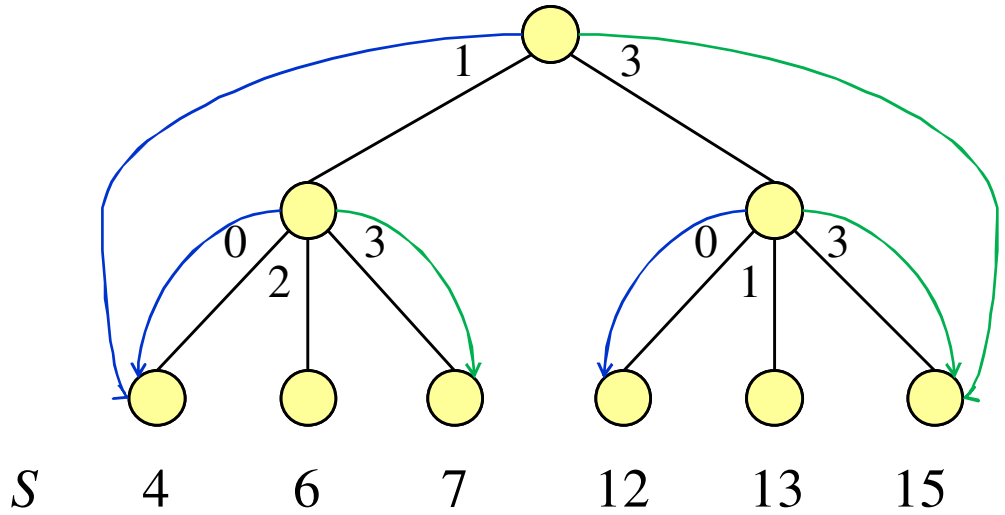
このとき、図のように、途中のどの頂点においても child 配列を左端から右端まで辿ることになってしまいます。

各 child 配列のサイズが b で、このパスの長さが h なので、O(bh) 時間かかってしまう、というわけです。

p-fast trie では、これを O(h + log b) 時間に改善します。

lowkey と highkey

- lowkey(v) : v の部分木の最小要素へのポインタ
- highkey(v) : v の部分木の最大要素へのポインタ



そのために、いくつかの補助データ構造を用います。

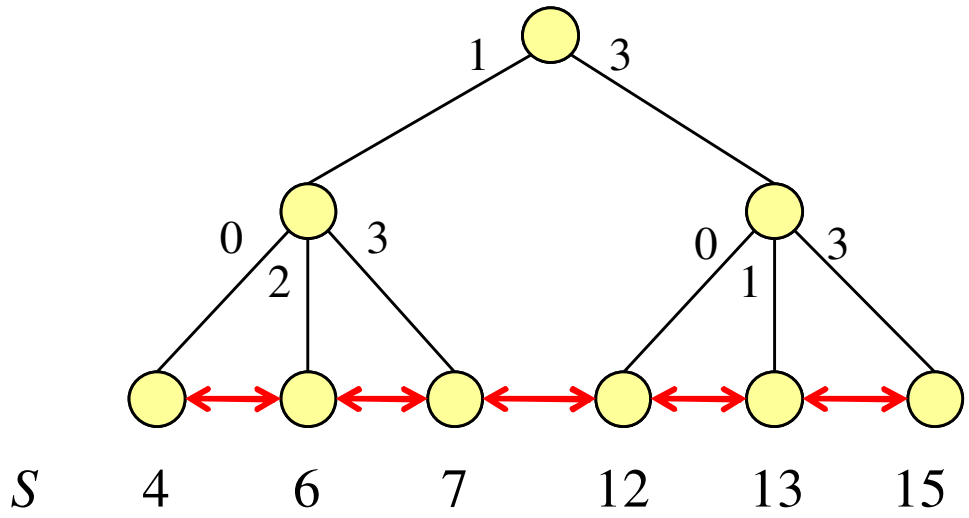
まず、葉でない各頂点から、lowkey と highkey というポインタを貼ります。

各頂点 v について、lowkey(v) と highkey(v) はそれぞれ v を根とする部分木が保持する (S の) 最小要素へのポインタです。

図の例では、青矢印が lowkey, 緑矢印が highkey を表しています。

葉の双方向連結リスト

- 隣の葉に $O(1)$ 時間でアクセス可能

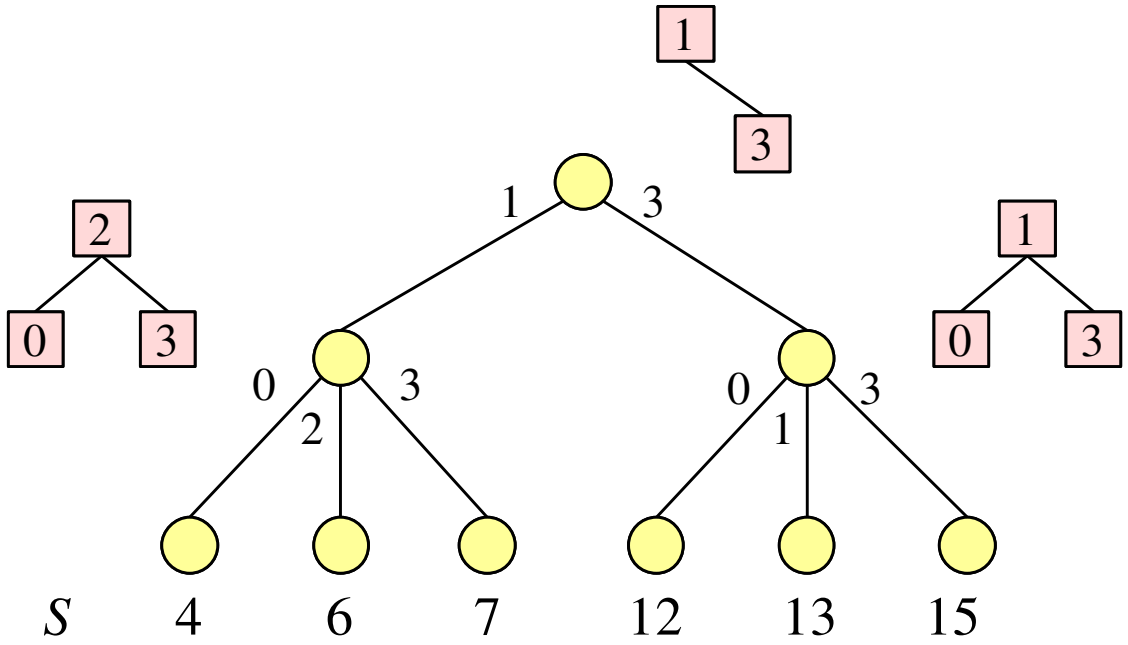


また、葉を双方向連結リストで連結します。

この双方向連結リストを用いることで、隣の葉に定数時間でアクセスできます。

innertree

- $\mathrm{innertree}(v) : \mathrm{child}_v[i] \neq \mathrm{nil}$ を満たす要素からなる平衡2分木



さらに、innertree という平衡2分探索木を各頂点に保持します。

葉でない各頂点 v について、innertree(v) は、v の child 配列の空でない要素からなる平衡2分探索木です。

この図で、例えば左の内部頂点について注目してみましょう。

この内部頂点は 0番目、2番目、3番目の子を持っているので、この頂点の innertree は 集合 {0, 2, 3} に対する平衡2分探索木です。

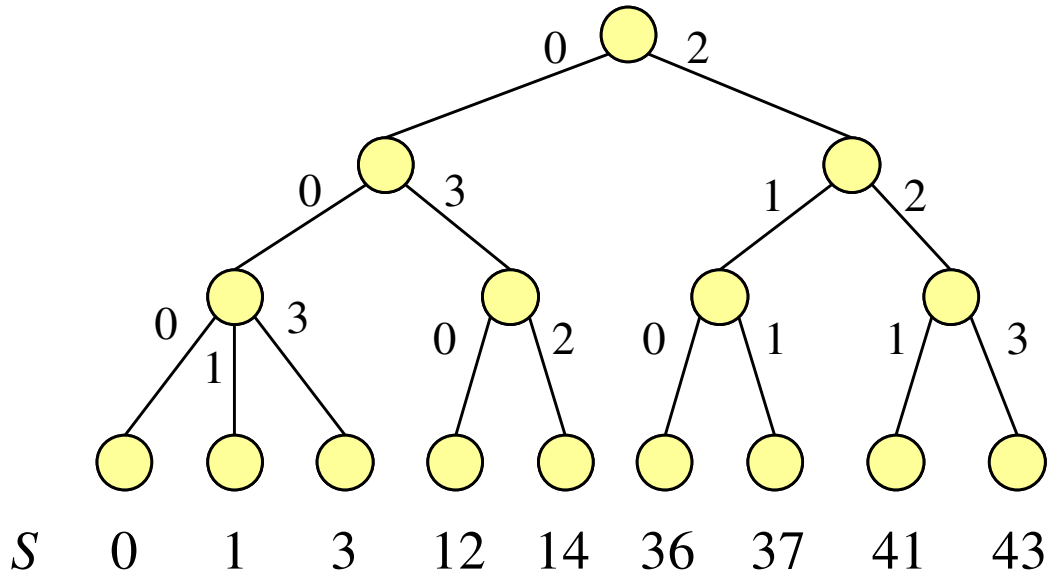
他の頂点についても同様です。

これでやっと準備が終わりました。これらの補助データ構造を持つトライを p-fast trie といいます。

以降のスライドで、p-fast trie を用いた successor クエリの処理方法を説明します。

successor with p-fast trie (場合１)

- successor(6_{10}, S) = successor(012_4, S)



p-fast trie を用いた successor クエリについて、2つの場合があります。

場合 1 を、例を使って説明します。

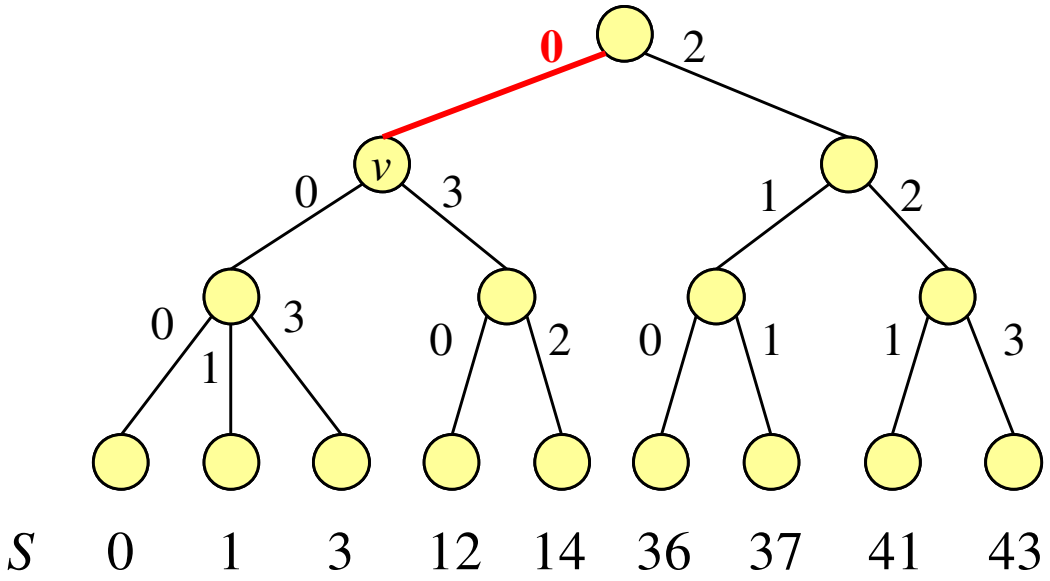
S = {0, 1, 3, 12, 14, 36, 37, 41, 43} とします。

この図の p-fast trie と使って、S における 6 の successor を探すことを考えましょう。

この p-fast trie は (3, 4) トライなので、6 を 4進数 012 に変換して根から探していきます。

successor with p-fast trie (場合 1)

- successor(6_{10}, S) = successor($\mathbf{0}12_4$, S)



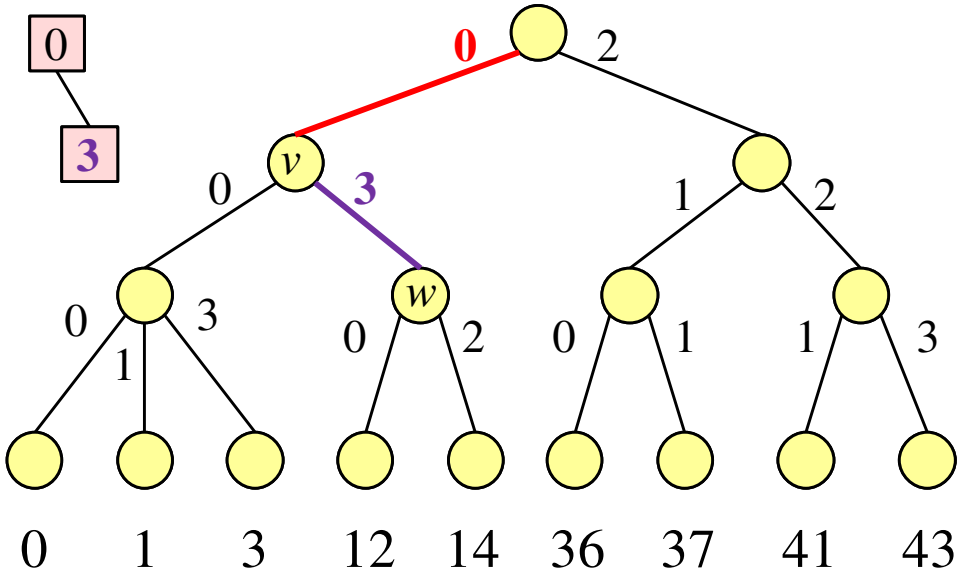
基本的には、012 で辿れるだけ根から辿っていきます。

まず根に 0 でラベル付けされた辺があるので、これを辿ります。この辺は根の child 配列を使って見つけることができます。

たどり着いた頂点を v としましょう。

successor with p-fast trie (場合 1)

- successor(6_{10}, S) = successor(012_4, S)
- successor(**1**, child$_v$) = 3



012 の左から2番目の要素 1 を v から辿ることができません。

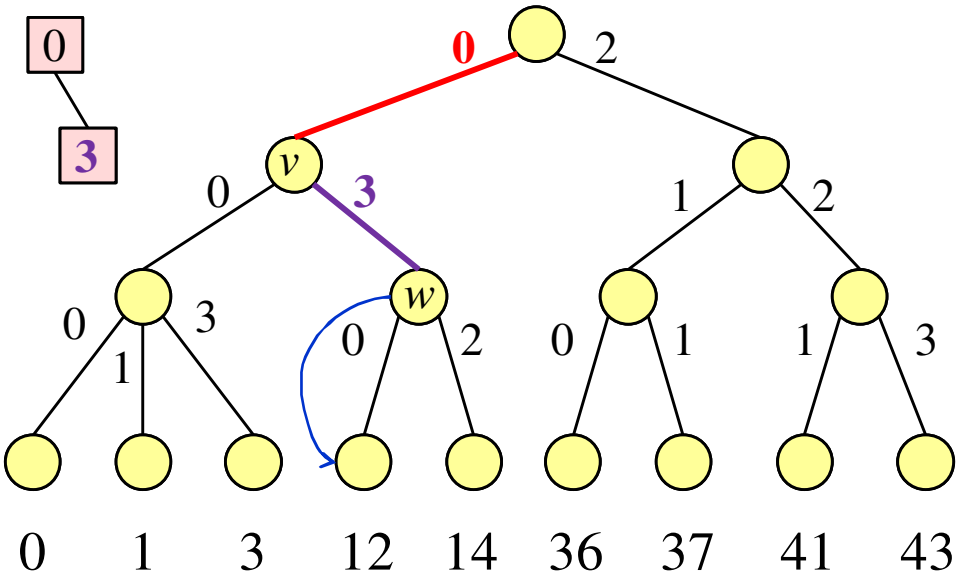
そこで、v の出力辺のラベルの集合 {0, 3} における 1 の successor を探します。答えは 3 です。

これは v の innertree（2分探索木）を使って見つけることができます（第1回講義参照）。

v から 3 でラベル付けされた辺を辿り、その行先の頂点を w とします。

successor with p-fast trie (場合 1)

- successor(6_{10}, S) = successor(012_4, S)
- successor(1, child$_v$) = 3
- lowkey(w) = 12_{10} = successor(6_{10}, S)



最後に、w の lowkey を辿って、w を根とする部分木に格納されている最小要素 12 を得ます。

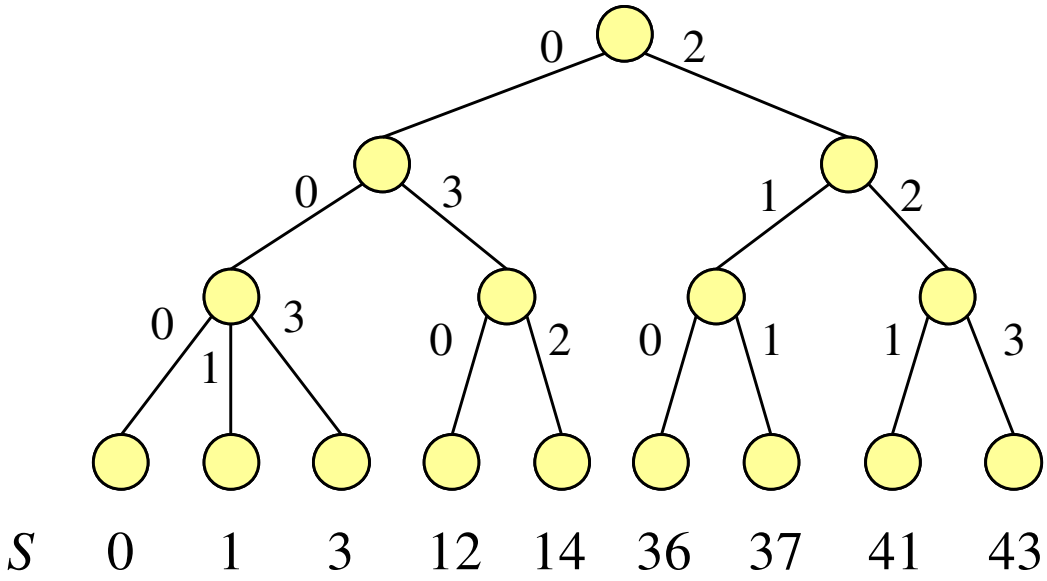
この 12 が 6 の successor です。

集合 S 中で、6 の次に大きい値は確かに 12 であることを確認しましょう。

これが場合1です。

successor with p-fast trie (場合２)

- successor(15_{10}, S) = successor(033_4, S)

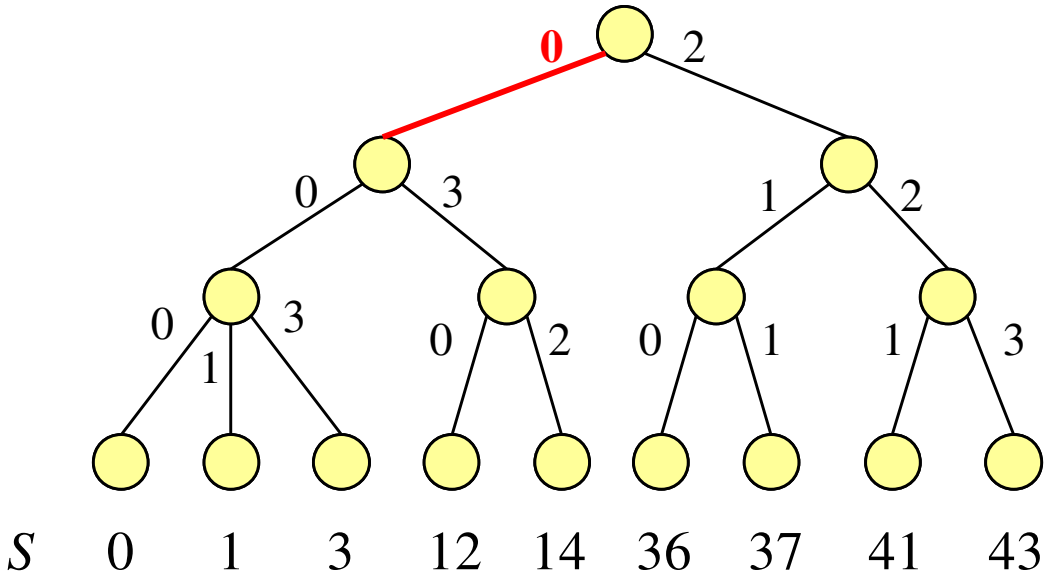


場合2について説明します。

同じ p-fast trie を使って、15 の successor を探します。4進数に変換すると 033 です。

successor with p-fast trie (場合２)

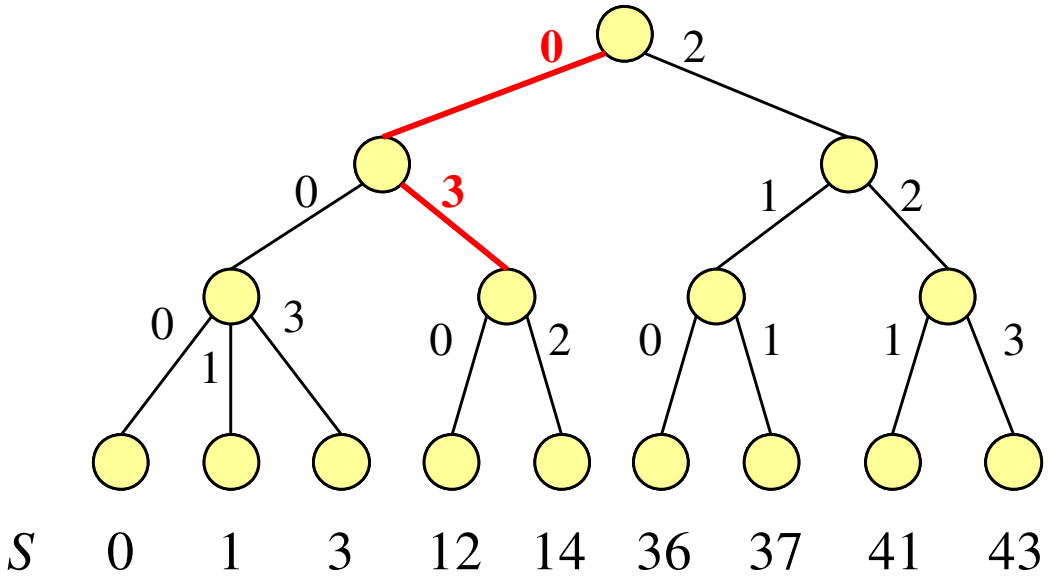
- successor(15_{10}, S) = successor($\mathbf{0}33_4$, S)



まずは、先ほどと同様に根から
033 を辿れるだけ辿ります。

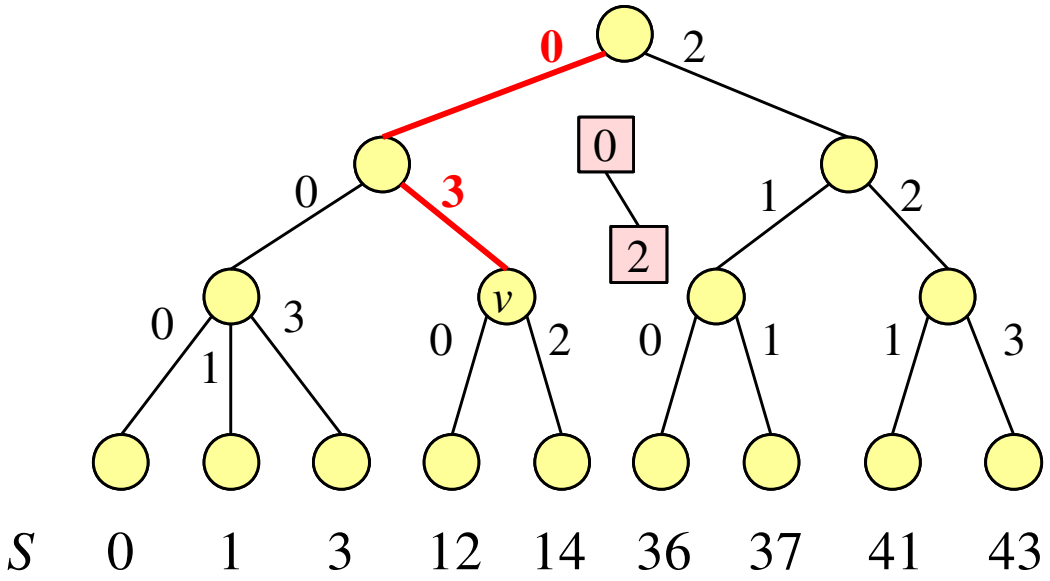
successor with p-fast trie （場合２）

- successor(15_{10}, S) = successor($\mathbf{03}3_4$, S)



successor with p-fast trie (場合２)

- successor(15_{10}, S) = successor(033_4, S)
- successor(3, v) = nil



根から 0, 3 と辿って、たどり着いた先の頂点を v とします。

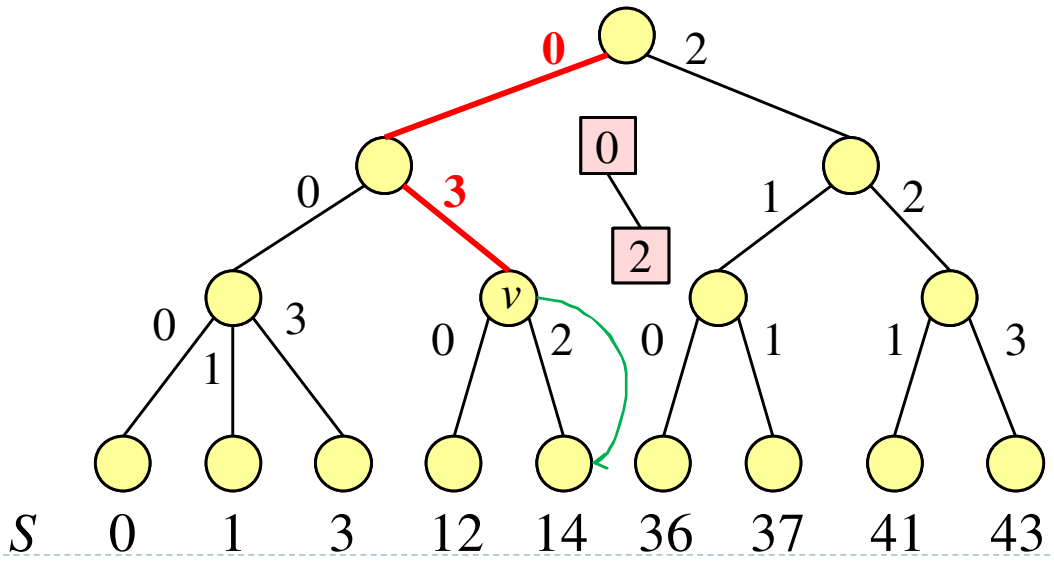
v からは次の 3（図では紫の字）では辿れません。

ここで、v の出力辺ラベルの集合 {0, 2} に対して 3 の successor を探しますが、存在しません (nil)。

なお、ここが場合1と異なるところです。

successor with p-fast trie (場合２)

- successor(15_{10}, S) = successor($\mathbf{033}_4$, S)
- successor(**3**, v) = nil
- highkey(v) = 14_{10}

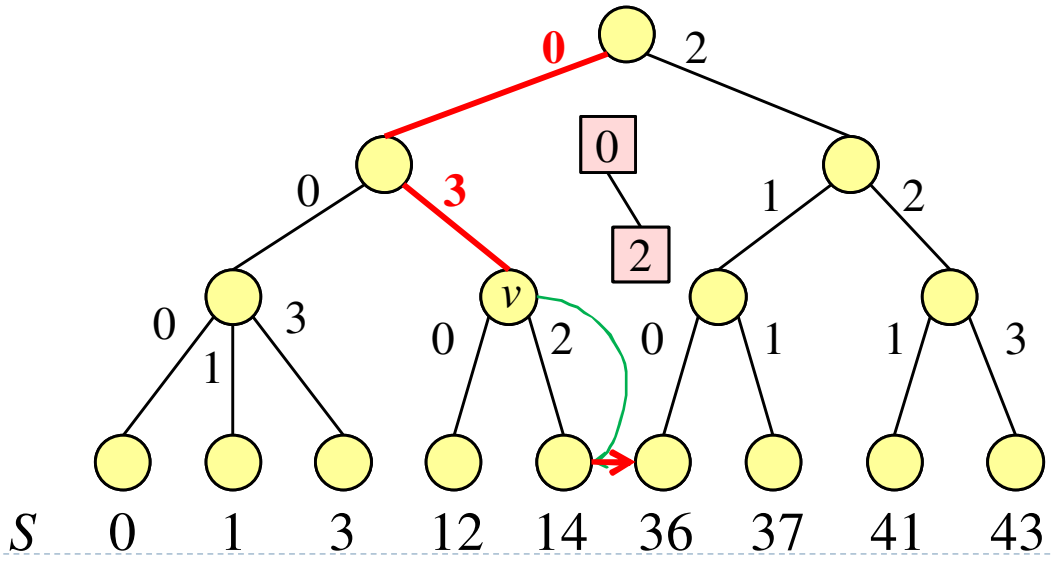


v における successor が存在しなかった場合には、v の highkey を辿ります。

辿った先の葉は 14 を格納しています。

successor with p-fast trie (場合２)

- successor(15_{10}, S) = successor($\mathbf{033}_4$, S)
- successor(**3**, v) = nil
- highkey(v) = 14_{10}
- successor(14_{10}, S) = 36_{10} = successor(15_{10}, S)



この 14 の葉から双方向連結リストを辿り、右隣りの葉 36 にアクセスします。

ここで得られた 36 が S における 15 の successor です。

場合2は以上です。

演習問題

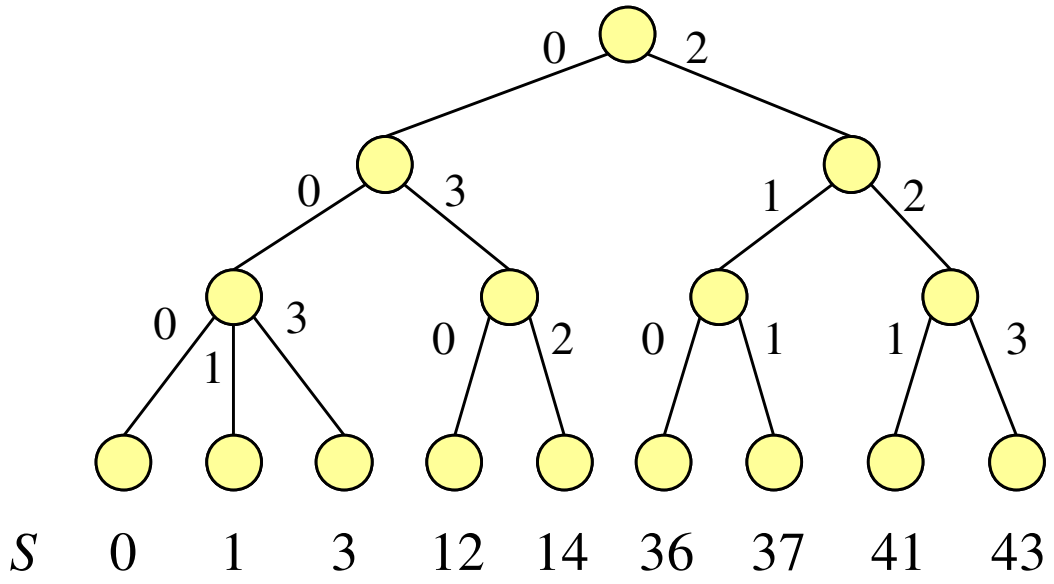
※ 演習問題 **提出〆切：6月11日（木）23:59**
pdf の作成方法について、次ページを参照のこと

以下の p-fast trie を用いて,

(1) successor(4_{10}, S)

(2) successor(39_{10}, S)

を求めよ. ただし, 途中経過を省略せずに書くこと.



S 0 1 3 12 14 36 37 41 43

それでは、本日の演習問題です。

以下の p-fast trie を用いて、

successor(4, S)
successor(3, S)

を求めてください。

ただし、前ページまでの例に倣って、計算の途中経過を省略せずに書くようにしてください。

successor with p-fast trie

補題

サイズ (h, b) の p-fast trie を用いることにより、
successor(x, S) を $O(h + \log b)$ 時間で計算できる

【証明】

- トライの探索 ➔ 各深さで child 配列を用いて合計 $O(h)$ 時間
- 頂点での successor 探索 ➔ 平衡2分木を用いて $O(\log b)$ 時間
- 対応する葉の探索 ➔ lowkey, highkey を用いて $O(1)$ 時間
- 葉の successor 探索 ➔ 連結リストを用いて $O(1)$ 時間
- 全体で $O(h) + O(\log b) + O(1) = O(h + \log b)$ 時間

これまでの議論から、次の補題を得ます。

サイズ (h, b) の p-fast trie を用いることにより、successor(x, S) を O(h + log b) 時間で計算できる。

アルゴリズムの正当性（正しく successor を計算できる保証）は、先ほどの説明から明らかです。

計算時間について見ていきましょう。

（次スライドに続く）

successor with p-fast trie

補題

サイズ (h, b) の p-fast trie を用いることにより、
successor(x, S) を $O(h + \log b)$ 時間で計算できる

【証明】

- トライの探索 ➔ 各深さで child 配列を用いて合計 $O(h)$ 時間
- 頂点での successor 探索 ➔ 平衡2分木を用いて $O(\log b)$ 時間
- 対応する葉の探索 ➔ lowkey, highkey を用いて $O(1)$ 時間
- 葉の successor 探索 ➔ 連結リストを用いて $O(1)$ 時間
- 全体で $O(h) + O(\log b) + O(1) = O(h + \log b)$ 時間

successor の探索では、根からトライを辿れるだけ辿っていました。

トライの各深さの頂点において、child 配列を参照することで、毎回 O(1) 時間で辺を辿ることができます。
トライの高さは h なので、高々 h 個の辺の探索を行うので、このステップは O(h) 時間で処理できます。

次に、トライを辿れなくなった頂点（これを v とする）において、v の出力辺のラベルの集合に対する successor を行っていました。
これには、innertree という平衡2分探索木を使いました。トライの分岐数は最大で b なので、innertree の要素数は高々 b です。
よって、innertree の高さは O(log b) となり、successor の計算は O(log b) 時間で抑えられます。

（次スライドに続く）

successor with p-fast trie

補題

サイズ (h, b) の p-fast trie を用いることにより、
successor(x, S) を $O(h + \log b)$ 時間で計算できる

【証明】

- トライの探索 ➔ 各深さで child 配列を用いて合計 $O(h)$ 時間
- 頂点での successor 探索 ➔ 平衡2分木を用いて $O(\log b)$ 時間
- 対応する葉の探索 ➔ lowkey, highkey を用いて $O(1)$ 時間
- 葉の successor 探索 ➔ 連結リストを用いて $O(1)$ 時間
- 全体で $O(h) + O(\log b) + O(1) = O(h + \log b)$ 時間

v の successor を w とします。
w の lowkey (場合1)1、highkey (場合2) を辿る時間は O(1) です。

場合2では、highkey で辿った葉の successor を連結リストで O(1) 時間で辿ります。
よって、これらのステップでの時間計算量は O(1) です。

以上をすべて合わせて、O(h + log b) 時間を得ます。

証明は以上です。

successor with p-fast trie

定理

successor(x, S) を $O(\sqrt{\log u})$ 時間で計算できる p-fast trie が存在し、その領域計算量は $O(n\sqrt{\log u}\,2^{\sqrt{\log u}})$ である

【時間計算量の証明】

- $b = 2^{\sqrt{\log u}}$, $h = \sqrt{\log u}$ とすると,
  補題より $O(h + \log b) = O(\sqrt{\log u})$ 時間

前ページの補題から、直ちにこの定理を得ることができます。

successor(x, S) を O(√log u) 時間で計算できる p-fast trie が存在し、その領域計算量は左に示す通りです。

まず、時間計算量について説明します。

前ページの補題で、successor を O(h + log b) 時間で処理できるとありました。

h と b はそれぞれ p-fast trie の高さと最大分岐数を決定するパラメータです。

そこで、b = $2^{\sqrt{\log u}}$, h = √log u と設定すると、補題より
O(h + log b) = O(√log u) 時間で successor を計算できることになります。

successor with p-fast trie

定理

successor(x, S) を$O(\sqrt{\log u})$ 時間で計算できる p-fast trie が存在し、その領域計算量は $O(n\sqrt{\log u}\,2^{\sqrt{\log u}})$ である

【領域計算量の証明】

- 各深さにおいて，高々 n 個の頂点が存在
- 各頂点の領域計算量 ➔ $O(b)$
- 全体で $O(nhb) = O(n\sqrt{\log u}\,2^{\sqrt{\log u}})$ 領域

次に、領域計算量について証明します。

p-fast trie の各深さにおいて、頂点の最大数を考えます。

最も深い位置の頂点は葉です。明らかに葉はちょうど n 個あります（n は S の要素数）。

葉から1段ずつ上がっていったとき、頂点数は単調非増加します。

よって、各深さにおいて、高々 n 個の頂点が存在しいます。

各頂点は高々 b 個の子を持ち、child 配列、innertree のサイズは O(b)、lowkey / highkey のサイズは O(1) なので、各頂点の領域計算量は O(b) です。

よって、これらをすべて掛け合わせた O(nhb) で p-fast trie のサイズを抑えることができます。
先ほど決めたパラメタ h と b の値を代入して、このような領域計算量が得られます。

次回予告など

- 次回は，クエリの時間を $O(\sqrt{\log u})$ に保ったまま領域を $O(n)$ に改善したデータ構造 (q-fast trie) を取り扱います．

本日の講義は以上です。

次回は、クエリの時間を O(√log u) に保ったまま、領域計算量を O(n) に改善したデータ構造 q-fast trie を扱います。