データ構造のたのしみ

Delight in Data Structures

有村 博紀

Hiroki Arimura

2023 年 5 月 2 日

目　次

第I部

先端的な辞書データ構造

第 1 章

基本的な記法

1.1 数の記法

$\mathbb{Z}$ で**整数** (*integers*) の全体を表し，$\mathbb{R}$ で**実数** (*real numbers*) の全体を，$\mathbb{Q} \subseteq \mathbb{R}$ で**有理数** (*rational numbers*) の全体を，$\mathbb{N} = \{0, 1, 2, \ldots\}$ で**非負整数** (*non-negative integers*) の全体を表わす．$\mathbb{R}$ または $\mathbb{Z}$ 上において，通常の四則演算（加算・減算・乗算・除算）を，$+, -, \times, /$ で表す．数 x を数 p $(p \neq 0)$ で割った**余り** residue または**剰余** (*residue*) を $x \bmod p$ と表す．数 $x \in \mathbb{R} \cup \{\infty\}$ に対して，x の値が有限であることを，$x < \infty$ と表す．

任意の実数 $x \in \mathbb{R}$ に対して，$\lceil x \rceil \in \mathbb{Z}$ で，x の**切り上げ** (*round up*)，すなわち，x 以上の最小の整数を表し，$\lfloor x \rfloor \in \mathbb{Z}$ で，x の**切り捨て** (*round down*)，すなわち，x 以下の最大の整数を表す．整数 x に対しては，$\lceil x \rceil = \lfloor x \rfloor = x$ である．[†1] 例として，正数 $x = 2.4$ に対して $\lceil 2.4 \rceil = 3$ と $\lfloor 2.4 \rfloor = 2$ となる．負数 $x = -2.4$ に対しては，$\lceil -2.4 \rceil = 2$ と $\lfloor -2.4 \rfloor = -3$ となる．

真 (*true*) または**偽** (*false*) を返す文 ψ を**述語** (*predicate*) という．述語 ψ の**指示関数** (*indicator function*) を，$\mathbb{I}[\psi] \in \{0, 1\}$ で表し，ψ が真ならば 1 を返し，偽ならば 0 を返すようなブール関数と定める．

[†1] $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$ かつ $\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\}$ である．

1.2 区間の記法

任意の実数 $a, b \in \mathbb{R}$ $(a \leq b)$ に対して，a と b を端点にもつ**実数閉区間** (*real closed interval*) を $[a, b] := \{ x \in \mathbb{R} \mid a \leq x \leq b \}$ と定める．同様に，開区間 $(a, b) := \{x \in \mathbb{R} \mid a < x < b\}$，および右半開区間 $[a, b)$，左半開区間 $(a, b]$ を定める．

同様に，整数の範囲の記法を導入する．任意の整数 $i \leq j$ に対して，$[i..j] = i..j := \{i, i+1, \ldots, j\}$ と $[i..j) := \{i, i+1, \ldots, j-1\}$ と定義する．後者 $[i..j)$ は，右端 j を含まないことに注意されたい．[†2]

1.3 解析

数の解析について復習する．級数の収束や，実数上の四則演算や，平方根，三角関数，対数関数等の初等的な関数については，読者が基本的な知識をもつと仮定する．[†3]

自然対数の底 (*he base of the natural logarithm*) と呼ばれる定数を $e := \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n = 2.71828...$ とする．[†4] 任意の実数 $x > 0$ に対して，**指数関数** (*exponential function*) $\exp x = e^x$ は，等式 $\exp x = \lim_{n\to\infty} \left(1 + \frac{x}{n}\right)^n$ で与えられる実数関数である．[†5] 実数 x を**指数** (*exponent*) と呼び，e を**底** (*base*) と呼ぶ．特別な場合として指数が非負整数 n の場合に，指数関数 e^n は，底 e を n 回繰り返して乗算したもの，すなわち底 e の n 乗 $e^n = \overbrace{e \ldots e}^{n}$ に等しい．任意の実数 x, y に対して，等式 $e^{x+y} = e^x \cdot e^y$，と $e^{x \cdot y} = (e^x)^y$ が成立する．2 番目の式から，任意の正数 $a > 0$ を底とする指数関数 a^b $(b \in \mathbb{R})$ を定めることができる．[†6]

[†2] これらの $[i..j]$ や $[i..j)$ のような整数範囲の記法は，配列のスライス記法と呼ばれて，最近のプログラミング言語に取り入れられており，アルゴリズムの簡潔な記述に便利である．

[†3] この節の内容は，いわゆる解析学（mathematical analysis）の初歩である．情報科学分野では，Cormen et al. [2009] の付録が解説を含んでおり，離散数学の教科書 Graham et al. [1989] は，アルゴリズム解析で現れる各種の級数の解法等，高度な内容を扱っている．

[†4] 他にも各種の等価な定義が存在する．**ネイピア数** (*Napier's number*) ともいう．

[†5] 他に，級数 $\exp x = \lim_{n\to\infty} \frac{x^n}{n!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \ldots$ としても与えられる．

[†6] 実際，自然対数関数を用いると，$a^b := e^{b \ln a}$ と書ける．

任意の正定数 $a > 0$ に対して，a を底とする数 $x > 0$ の**対数関数** (*logarithmic function*) $\log_a x$ は，任意の正数 $x > 0$ に対して，等式 $e^p = x$ を満たす一意な実数 p として定義される．[†7] 任意の実数 $x, y, z,\ (x, y > 0)$ に対して，等式 $\log_a(x+y) = \log_a x + \log_a y$ と $\log_a x^z = z \log_a x$, が成立する．正整数 2 を底とする対数を $\lg x = \log_2 x$ で表す．ネイピア数 e を底とする対数 $\ln x = \log_x e$ を，x の**自然対数** (*natural logarithm*) と呼ぶ．底の変換によって，対数関数は定数倍しか変化しない．すなわち，任意の $a > 0$ に対して，等式 $\log_a x = \frac{1}{\ln a} \ln x$ が成立する．[†8]

補題 1.1　指数関数に関して，次が成立する．

$$e^x \geq 1 + x, \qquad e^{-x} \geq 1 - x.$$

ここに，$x = 0$ のとき，等号が成立する．

補題 1.1 はアルゴリズムの確率的解析でよく用いられる．表が出る確率が $0 < p < 1$ であるコインを考えて，何度も繰り返し独立なコイン投げの試行をすれば，高い確率で一度は表の事象が生じることを示したいとする．このとき，1 回のコイン投げで表が出ない確率は $(1-p)$ であるので，任意の非負整数 $n \geq 0$ に関して，n 回のコイン投げの連続試行のうち，表が一度も出ない確率 q_{bad} を見積もると，

$$q_{\mathrm{bad}} = (1-p)^n \leq e^{-pn}$$

となり，n の指数関数で上から抑えられることが言える．

補題 1.2　対数関数に関して，次が成立する．

$$\frac{d}{dx} \ln x = \frac{1}{x}, \qquad \int \frac{1}{x}\, dx = \ln x, \qquad\qquad x > 0$$

一般の底 $a > 0$ の場合も，補題 1.2 から求められる．[†9]

[†7] この定義より，底 a が同じ指数関数 $y = a^x$ と対数関数 $x = \log_a y$ は互いに逆関数になっていることがわかる．

[†8] このことより，オーダー表記のように定数倍を問題としない場合には，底 a を指定せず $O(\log x)$ のように書いても正確さを失わない．

[†9] 一般の底 $a > 0$ に対して，指数関数 $y = a^x$ は $y = e^z, z = x \ln a$ と合成関数の形で書ける

数列の和

アルゴリズムの計算量解析において，良く使われる基本的な結果について復習する．

補題 1.3（無限等比級数） 任意の正数 $0 < \alpha < 1$ に対して，次の等式が成立する．

$$\sum_{i=0}^{\infty} \alpha^i = \alpha^0 + \alpha^1 + \cdots + \alpha^n + \cdots = \frac{1}{1-\alpha}$$

証明: 証明のスケッチを示す．条件 $\alpha < 1$ より，この級数は有限な値に収束するので，その値を $S < \infty$ とおく．次に S から αS を差し引くと，等式

$$\begin{aligned} S - \alpha S &= \sum_{i=0}^{\infty} \alpha^i - \sum_{i=0}^{\infty} \alpha\alpha^i = \sum_{i=0}^{\infty} \alpha^i - \left(-\alpha^0 + \sum_{i=1}^{\infty} \alpha^i \right) \\ &= S - (-1 + S) = 1, \end{aligned}$$

を得る．この等式 $S - \alpha S = (1-\alpha)S = 1$ を S について解くと，解として $S = 1/(1-\alpha)$ を得る．[†10] □

補題 1.3 より直ちに，初項が β で等比が α の無限等比級数の値は $\frac{\beta}{1-\alpha}$ となる．

系 1.4 任意の正数 $0 < \alpha < 1$ に対して，次の等式が成立する．

$$\sum_{i=1}^{\infty} \alpha^i = \alpha^1 + \alpha^2 + \cdots + \alpha^n + \cdots = \frac{\alpha}{1-\alpha}$$

証明: 求める級数は $\sum_{i=1}^{\infty} \alpha^i = \alpha \sum_{i=0}^{\infty} \alpha^i$ と書けるので，補題 1.3 から直ちに導かれる． □

以上の結果をもとに，より複雑な級数の値も求めることができる．

ので，合成関数の微分を用いると，$\frac{d}{dx}a^x = \frac{dy}{dx} = \frac{dy}{dz}\cdot\frac{dz}{dx} = e^z \cdot \ln a = \ln a (e^{\ln a})^x = (\ln a)a^x$ を得る．不定積分は，$\int a^x dx = \frac{1}{\ln a}a^x$ である．対数関数 $y = \log_a x = \frac{1}{\ln a}\ln x$ は，自然対数 $\ln x$ の定数倍なので，$\frac{d}{dx}\log_a x = \frac{1}{\ln a}\frac{d}{dx}\ln x = \frac{1}{x \ln a}$ を得る．

[†10] この形式の証明は，**望遠鏡論法** (*telescope argument*) と呼ばれる．この名前は，項が単調に減少する級数を操作するようすが，伸縮する折り畳み式の望遠鏡に似ていることに由来しており，漸化式の解法によく現れる [Graham et al., 1989].

補題 1.5　任意の正数 $0 < \alpha < 1$ に対して，次の等式が成立する．

$$\sum_{i=1}^{\infty} i \cdot \alpha^i = 1 \cdot \alpha^1 + \cdots + n \cdot \alpha^n + \cdots = \frac{\alpha}{(1-\alpha)^2}$$

証明:　証明のスケッチを示す．求める級数は，$\alpha < 1$ のとき収束するので，これを T とおく．ここで，差分 $T - \alpha T = (1-\alpha)T$ を求めると，これは補題 1.4 の級数 S に一致する．そこで，等式 $(1-\alpha)T = S$ を T について解いて，補題を得る．　□

級数 $S_\infty = \sum_{k=1}^{\infty} \frac{1}{k}$ は**調和級数** (*harmonic series*) と呼ばれる．調和級数は発散するが，第 n 項までの部分和について次を得る．

補題 1.6（調和級数の部分和）　任意の正整数 $n \geq 1$ に対して，次式が成立する．

$$\ln(n+1) < \sum_{k=1}^{n} \frac{1}{k} = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} \leq 1 + \ln n$$

証明:　求める部分和 $H_n = \sum_{k=1}^{n} \frac{1}{k}$ は，区間 $I = [1, n+1)$ 上の階段関数 $g(x) = \lfloor x \rfloor$ の下の面積に一致する．H_n の下限は，定積分 $\int_1^{n+1} \frac{1}{x} dx = [\ln x]_1^{n+1} = \ln(n+1)$ で与えられる．第 1 項以外の $H_n - 1$ の上限は，定積分 $\int_1^n \frac{1}{x} dx = [\ln x]_1^n = \ln n$ で与えられるので，H_n の上限 $1 + \ln n$ を得る．[†11]　□

再帰アルゴリズムの時間計算量

再帰アルゴリズムの入力サイズ n に対する計算時間 $f(n)$ の解析では，f に関する漸化式を立てて，それを解く．下記は，良く用いられる漸化式の例とその解である．

補題 1.7（漸化式の解）　任意の定数を $a, b > 0$ とおく．

(1) 漸化式 $f(1) \leq a$; $f(n) \leq 2f(\frac{n}{2}) + bn$ $(\forall n > 1)$ を満たす任意の f に対して，$f(n) = O(n \log n)$ が成立する．
(2) 任意の定数 $0 < \alpha < 1$ に対して，漸化式 $f(1) \leq a$; $f(n) \leq f(\alpha n) + bn$ $(\forall n > 1)$ を満たす任意の f に対して，$f(n) = O(n)$ が成立する．

[†11] 図では，階段関数の 1 項目を除き，関数のグラフ一つ左にずらすことに相当する．

証明: 証明のスケッチを与える．

(1) 望遠鏡論法を用いる．再帰の深さを $k = \lceil \lg n \rceil$ とおき，任意の $i = 1, 2, 3, \ldots, k$ に対して，漸化式に $n \leftarrow n(\frac{1}{2})^i$ を代入し，両辺に 2^i を掛けたものを作り，両辺をそれぞれ足し合わせると，f に関する途中の項が互いにキャンセルして，$f(n) \leq 2^k f(0) + \sum_{i=1}^{k} bn = 2^{\lceil \lg n \rceil} a + kbn = O(n \log n)$ を得る．

(2) 無限等比数列の和を用いて上限を抑える．再帰を k 回繰り返すと仮定し，条件 $\alpha^k n \leq 1$ を解いて，これを満たす正整数 $k = O(\log n)$ を求める．ここで，$f(n)$ を k 回展開すると，不等式 $f(n) \leq a + \sum_{i=1}^{k} b\alpha^i \leq a + bn \sum_{i=1}^{\infty} \alpha^i = a + bn\frac{1}{1-\alpha} = O(n)$ を得る．ここに，3 番目の式から 4 番目の式を導くには，無限等比級数の値に関する補題 1.3 を用いた． □

計算量解析への良く知られた応用例として，補題 1.7 は次のような問題の解析に適用されている: (1) の形：入力を等長に 2 分割しながら，両側で再帰を行うクイックソートやマージソート等の整列．(2) の形：定数割合で 2 分割して片側のみ再帰を行う数値列の中央値の探索や，接尾辞配列の線形時間構築．上記以外の形の漸化式の一般的な解法については，教科書 [Cormen et al., 2009, Graham et al., 1989] 等を参照されたい．

1.4 組合せ

階乗と関連した演算を導入する．正整数 $n \geq 1$ の**階乗**（かいじょう） (*factorial*) は，次のように，n から 1 までの数の乗算で得られる数

$$n! := \prod_{i=n}^{1} i = n(n-1)\cdots 1$$

である．ここに，$0! = 1$ と定義する．

[この項目は要検討] **下降階乗巾** (*falling factorial*) を，

$$n^{\underline{k}} = (n)_k := \prod_{i=0}^{k-1} (n-i) = \overbrace{n(n-1)\cdots(n-k+1)}^{k}$$

と定め，**上昇階乗巾** (*riksikg factorial*) を，

$$n^{\overline{k}} = (n)^{(k)} := \prod_{i=0}^{k-1}(n+i) = \overbrace{n(n+1)\cdots(n+k-1)}^{k}$$

と定める．階乗は $n! = n^{\underline{n}} = 1^{\overline{n}}$ とも書ける．

任意の整数 $n \geq 0$ と $k \in [0..n]$ に対して，n 個の相異なる要素から k 個を選ぶ**順列**（じゅんれつ）(*permutation*) の数を，

$$\left[\begin{array}{c} n \\ k \end{array} \right] := \overbrace{n(n-1)\ldots(n-k+1)}^{k} = \frac{n!}{(n-k)!}$$

で表す．

階乗 $n!$ の上下限として，$2^{n-1} \leq n! \leq n^n$ は明らかである．より正確な上下限として，次に示す**スターリングの近似式** (*Stirling's approximation*) が知られている．

補題 1.8（スターリングの近似式） 任意の正整数 $n \geq 1$ に対して，次の不等式が成立する．

$$\sqrt{2\pi}n^{n+\frac{1}{2}}e^{-n} \leq n! \leq en^{n+\frac{1}{2}}e^{-n}$$

任意の整数 $n \geq 0$ と $k \in [0..n]$ に対して，n 個の相異なる要素から k 個を選ぶ**組合せ**（くみあわせ）(*combination*) の数

$$\binom{n}{k} := \frac{\overbrace{n(n-1)\ldots(n-k+1)}^{k}}{\underbrace{k(k-1)\ldots 1}_{k}} = \frac{n^{\underline{k}}}{k!} = \frac{n!}{(n-k)!k!}$$

を，**二項係数** (*binomial coefficient*) と呼ぶ．

補題 1.9（二項定理）

$$(x+y)^n = \sum_{i=0}^{n} \binom{n}{i} x^{n-i}y^i$$

二項定理で $x = y = 1$ とおくと，次の等式が得られる．

$$\sum_{i=0}^{n} \binom{n}{i} = 2^n$$

1.5 確率

本書で必要な確率論の初歩を導入する．コルモゴロフの確率の公理については後で書く．以下では，値の領域を A で表す.（確率変数 X と実現値 a を導入すること .）

多変数確率分布

多変数確率分布について，以降の章で用いるいくつかの記法と補題を与える．実現値の列 $\mathbf{a} = \{a_i\}_{i=1}^m \in A^m$ に対して，m 個の変数の列 $X = \{X_i\}_{i=1}^m$ の**同時確率** (*joint distribution*) を

$$P(X_1 = a_1, \ldots, X_m = a_m)$$

で表す．文脈から $\mathbf{X}$ や $\mathbf{a}$ が明らかな場合は，$P(\mathbf{X})$, $P(\mathbf{a})$，$P_{\mathbf{X}}(\mathbf{a})$ などと書く．

いくつかの有用な等式を与える．以下では， 長さ m と n の任意の変数列を $\mathbf{X} = \{X_i\}_{i=1}^m, \mathbf{Y} = \{Y_i\}_{i=1}^n$ とおき，X と Y それぞれの任意の実現値の列を $\mathbf{a} = \{a_i\}_{i=1}^m \in A^m, \mathbf{b} = \{b_i\}_{i=1}^n \in A^n$ とおく．

補題 1.10（周辺化） 同時確率分布 $P(XY)$ に対して，X に関する P の**周辺化**(しゅうへんか) (*marginalization*) とは，確率分布

$$P(\mathbf{X} = \mathbf{a}) = \sum_{\mathbf{b} \in A^n} P(\mathbf{X} = \mathbf{a}, \mathbf{Y} = \mathbf{b})$$

をいう．明らかな場合は，これを単に，$P(\mathbf{X}) = \sum_{\mathbf{Y}} P(\mathbf{X}, \mathbf{Y})$ と書く．

次の補題は，確率の**乗算規則** (*product rule*)，または，確率の**連鎖規則** (*chain rule*) と呼ばれ，複雑な確率を分解するときに有用である．

補題 1.11（乗法規則） 変数列 $X = \{X_i\}_{i=1}^m, Y = \{Y_i\}_{i=1}^n$ と任意の実現値の列 $\mathbf{a} = \{a_i\}_{i=1}^m \in A^m, \mathbf{a} = \{a_i\}_{i=1}^m \in A^m$ に対して，次の等式が成立する．

$$P(\mathbf{X} = \mathbf{a}, \mathbf{Y} = \mathbf{b}) = P(\mathbf{X} = \mathbf{a} \mid \mathbf{Y} = \mathbf{b}) P(\mathbf{Y} = \mathbf{b})$$

明らかな場合は，これを単に，$P(\mathbf{X}, \mathbf{Y}) = P(\mathbf{X} \mid \mathbf{Y}) P(\mathbf{Y})$ と書く．

確率論では，複数の変数（列）の独立性(どくりつせい)を次のように定義する．

定義 1.1（独立性） 任意の正整数 $n \geq 1$ に対して，n 個の変数 $X_1, \ldots, X_n$ が**独立** (*independent*) であるとは，それらの任意の実現値 $a_1, \ldots, a_n \in A$ に対して，次の等式が成立することをいう．

$$\begin{aligned} P(X_1 = a_1, \ldots, X_n = a_n) &= P(X_1 = a_1) \cdots P(X_n = a_n) \\ &= \prod_{i=1}^{n} P(X_i = a_i) \end{aligned}$$

変数の独立性と，その亜種については，**??**節（対独立なランダム変数）で詳しく学ぶ．

包除原理

任意の確率分布 P に対して，二つの事象 X, Y が互いに背反な場合，すなわち，X と Y が同時に成立しない場合は，和事象 $X \cup Y$ の確率は，確率の公理から，$\mathrm{P}(X \cup Y) = \mathrm{P}(X) + \mathrm{P}(Y)$ で与えられる．それでは，X, Y が互いに背反でない場合に，和事象の確率 $\mathrm{P}(X \cup Y)$ はどのようにして求められるだろうか？ この問に対しては，次の等式が成立することが知られている．

$$\mathrm{P}(X \cup Y) = \mathrm{P}(X) + \mathrm{P}(Y) - \mathrm{P}(XY) \tag{1.1}$$

式(1.1) を一般化したものが，次に示す**包除原理** (*inclusion-exclusion principle*) である．

補題 1.12（包除原理） 任意の n 個の事象を $X_1, \ldots, X_n$ とおく．このとき，和事象 $X := \bigcup_{i=1}^{n} X_i$ について次の等式が成立する．

$$\mathrm{P}\left(\bigcup_{i=1}^{n} X_i\right) = \sum_{J \subseteq [1..n], J \neq \emptyset} (-1)^{|J|-1} \times \mathrm{P}\left(\bigcap_{j \in J} X_j\right)$$

1.6 計算モデル

本稿では、とくに断らない限り、計算モデルとして**単位コストのランダムアクセス機械** (*unit-cost random access machine*) を用いる。以降、これをRAM（ラム）と略す。RAM は、**語長** (*word length*) が w (ビット) の整数を格納できるレジ

スタとメモリをもつ通常の計算機のモデルである．ここに，レジスタや，メモリセル，整数型の値など，計算機における基本的なデータ単位を**語** (*word*) と呼ぶ．[†12]

RAM では，整数に対する次の演算が，いずれも語長 w によらず定数時間で実行できると仮定する：

- 番地指定による**メモリの読み書き** (*random access*).
- 値の等号比較(とうごうひかく)$=$ と大小比較 $\leq$.
- ビット毎の論理積 $\&$, 論理和 $|$, 否定 $\sim$, および左シフト $\ll$, 右シフト $\gg$ などの**ビット毎の論理演算** (*bit-wise operation*).
- 加算(かさん)$+$, 減算(げんざん)$-$, 乗算(じょうざん) $\times$, 除算(じょざん)**div**（または /）, 剰余(じょうよ)mod などの**数値演算**(すうち) (*arithmetic operation*).

語のビット配置は，**最上位ビット** (*most significant bit, MSB*) が左に，**最下位ビット** (*least significant bit, LSB*) が右に来るようにビット配置されていると仮定する．[†13].

RAM が，メモリの読み書きと，値の等号比較，大小比較の基本演算だけをもつとき，**比較に基づく RAM** (*comparison-based RAM*) と呼ぶ．さらに基本演算に加えて，数値演算とビット演算も許すとき，**ワード RAM** (*Word RAM, WRAM*) と呼ぶ．ワード RAM では，入力に出現する最大の数 n に対して，$w = \Omega(n)$bits の語長 w をもつと仮定する．[†14]ワード RAM において，WRAM($\emptyset$) と書いた時，基本 RAM にビット演算を追加したものを表し，さらに $+, \times, \ldots$ 等の数値演算を追加したものを WRAM($+, \times, \ldots$) 等と表す．また，実数の計算など，必要なときはその都度，拡張された演算を追加するものとする．

---

†12 多くの計算機で，基本的な整数型のビット数と，CPU のレジスタ，バス幅は同じことが多い．例えば，多くの 64 ビット計算機では，int 型の値とレジスタは 64 ビット幅をもつ．

†13 これに対して，$O(\log n)$ 演算時間を仮定したモデルを**対数コスト RAM** (*log-cost RAM*) と呼ぶ．大きな数を扱う数値算法などは，対数コストモデルによる計算量評価が妥当である．

†14 探索や整列等の基本的な問題でも，比較 RAM とワード RAM で計算時間の下限が異なる場合があることが知られている．また，ワード RAM は，表引きによる高速化や決定性ハッシュ法を多用する簡潔データ構造やビット並列手法などの計算量評価のモデルとしてよく使われる．

第2章

表構造

本章では，集合の検索を実現するための**静的辞書** (*static dictionary*) （または**表** (*table*)）と呼ばれる抽象データ型を説明する．単に**表** (*table*) と呼ばれることも多い．これらの構造は，Cormen et al. [2009] 等のアルゴリズムの教科書で**完全ハッシュスキーム** (*perfect hash scheme*) と呼ばれる技法を用いて実装される．

本節では，読者は，学部レベルのデータ構造とアルゴリズムの教科書（例えば，Cormen et al. [2009] 等）の内容は学習済みであると仮定する．以下では，大きさが高々$M \geq 1$ の値からなる**全体集合** (*universe*) を $U = \{0, 1, \ldots, M\}$ とおく．また，計算モデルとしてワード RAM を仮定する．

2.1 辞書

値の全体集合を U とおく．U 上の**辞書** (*dictionary*) は，U の n 個の要素からなる集合 $S = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq U$ を管理するデータ構造である．辞書は， キーの集合 S を管理し，次の演算をサポートする：

- $S =$ `create`() は，空の辞書を生成する．
- `makeset`(S) は，集合 S を格納した辞書を生成する．
- `find`(x) は，与えられたキー x が S に含まれるかに答える．もしキーが含まれるならば関連するデータへのポインタを返し，そうでなければ値 `null` を返す．
- `insert`(x) は，与えられたキー x が S に含まれないならば，x と関連す

るデータへのポインタを S に追加する．そうでなければ，何もしない．

- `delete`(x) は，与えられたキー x が S に含まれるならば，x と関連するデータへのポインタを S から除去する．そうでなければ，何もしない．

2.2 静的辞書（表）

辞書の特別な場合として，集合 S の変更演算である `insert` と `delete` の二つの演算をもたない辞書を，**静的辞書** (*static dictionary*)，または，**表** (*table*) と呼ぶ．静的辞書は，前処理と実行時の二段階で実現する．前処理では，あらかじめ与えられた集合 S から，`makeset` 演算を用いて構築される．実行時は，任意の時点における利用者からの**問合せ** (*query*) に応じて，与えられたキー x に対する `find` 演算を実行する．[†1] これに対して，通常の `insert` と `delete` の二つの変更演算をもつ辞書を，**動的辞書** (*dynamic dictionary*) と呼ぶことがある．

表の実現方式を概観しよう．表を実現するためのもっとも一般的な方法は，`find` と `insert` 演算のみを提供するハッシュ表を用いることである．`insert` 演算は，前処理で `makeset` を実現するのに用いる．これは，確率的な演算を用いて，全ての演算を $O(1)$ 時間で提供する動的辞書を実現する．

静的辞書が決定的な演算しか用いないとき，**決定的** (*deterministic*) と呼ばれる．決定的な静的辞書は，完全ハッシュスキーム技法を用いて構築される (Sec. 11.5, Cormen et al. [2009])．Fredman et al. [1984] らは，定数時間の探索を可能にする静的辞書に対する完全ハッシュスキームを与えている．これは，著者らの名前の頭文字をとって，FKS スキームと呼ばれる，この FSK スキームについては，2.5 節で学ぶ．

定理 2.1（Fredman et al. [1984]) n 個の要素からなる集合 $S \subseteq U$ において，FKS の完全ハッシュスキームは辞書の `Find`(x) 演算を $O(1)$ 最悪時間で実行するように実現する．

現代的な多くのデータ構造では，設計の際の部品として定数時間探索を必要

[†1] 静的辞書は，集合を変更するための演算である `insert` と `delete` をもたないので，代わりに前処理として，与えられた集合 S から表を構築するための `makeset` 演算をもつ．

とする．その場合に，FKS スキームによる決定性辞書は，ハッシュ表の代わりに使われることが多い．ただし，ハッシュ表と異なり，決定性辞書は静的データ構造なので，探索しか行えず，更新演算が必要な場合は別のデータ構造を用いる必要がある．

最近，カッコウハッシュ表と呼ばれる方式の表が提案されている．FKS スキームは，決定性な静的辞書を構築するためのきわめて一般的な枠組みであるが，前処理時間がハッシュ関数族の探索時間に依存しており，用途に合わせた前処理の方法が必要となる．そのような実際的な決定性辞書として，**カッコウハッシュ表** (*cuckoo hashing*) が，Pagh and Rodler [2004] によって提案されている．

2.3 $O(1)$ 期待探索時間をもつ動的辞書

ハッシュ表 (*hash table*) は，探索と更新演算を定数期待時間で実現する非順序辞書である．これは，バケットサイズを $s \geq n$ とするとき，入力元 x をランダムな添字 $h(x)$ に変換するハッシュ関数 $h : U \to [0..s)$ を用いて，バケット配列 $B[0..s)$ において入力元を格納する位置（バケットと呼ぶ）を見つけるデータ構造である．

ハッシュ表において，異なる二つの値 $x, x' \in S$ が同じバケットに変換されることを**衝突** (*collision*) という．ハッシュ表で衝突が起きた際の対応の仕方に，チェイニングと線形アドレッシングの二つの方法がある．前者の**チェイニング** (*chaining*) では，値の線形リストを用いることで，一つのバケットに複数の元を格納する．元 x の探索の際は，x をハッシュして得た番地 $i = h(x)$ のバケットが保持するリストを先頭から線形探索して目的の元 x を見つける．

後者の**線形アドレッシング** (*linear addressing*) では，元 x の挿入の際に，x をハッシュして得た番地 $i = h(x)$ が，既に他の元 y に使用されていた場合には，その地点 i から空き番地が見つかるまで $i, i+1, i+2, \ldots$ と連続した番地を昇順に見ていき，見つかった最初の空きに元を格納する．元 x の探索では同じように，目的の x と同じ元が見つかるまで，ハッシュされた番地 $i = h(x)$ から開始して，バケット配列を昇順で探索する．もし途中で空き番地に出会った

ら，探索を中止し，探索失敗として null を返す．

定理 2.2（期待定数時間演算をもつ動的辞書 Cormen et al. [2009]） 値が入力 n の多項式でおさえられる n 個の非負整数からなる集合 S を $s = O(n)$ 語の領域で格納し，Find(x) と，Insert(x, y)，Delete(x) 演算を $O(1)$ 期待時間で実行するように実現できる．

ハッシュ表は，実際的な性能も良く，実用的なアルゴリズムの実装に広く使われている．一方で，理論的なアルゴリズム設計の部品として用いる場合には，ハッシュ表は元同士の順序が扱えないことと，演算時間はあくまでも期待値であること，最悪時に $O(n)$ 時間を要する確率が 0 でないことに注意する必要がある．

2.4 $O(1)$ 最悪探索時間をもつ動的辞書

Dietzfelbinger et al. [1994] らは，探索を定数最悪時間に保ったまま，Insert(x) と Delete(x) にランダム性を導入し，ならし定数期待時間で実現することで，FSK の完全ハッシュスキームを拡張して動的辞書が構築できることを示している．

定理 2.3（Dietzfelbinger et al. [1994]） n 個の要素からなる集合 $S = \{0, 1, \ldots, n-1\}$ において，Find(x) 演算を $O(1)$ 最悪時間で実行し，Insert(x) と Delete(x) 演算をならし定数期待時間で実行するように動的辞書を実現可能である．

同時に，Dietzfelbinger et al. [1994] らは，ある計算モデルの下で，Find(x) と，Insert(x)，Delete(x) の全ての演算を $O(1)$ 最悪時間で実行可能なデータ構造は存在しないことを証明している．

2.5 $O(1)$ 最悪探索時間の静的辞書

本節では，Fredman et al. [1984] にしたがって，FKS スキームにもとづく静的辞書を説明する．これは，FKS の完全ハッシュスキームと呼ばれ，大きさが高々$M \geq 1$ の値の全体集合 $U = \{0, 1, \ldots, M-1\}$ を仮定した場合に，U の

n 個の要素からなる集合 $S = \{x_0, \ldots, x_{n-1}\} \subseteq U$ を，$O(n)$ 語の領域で格納し，定数最悪時間の探索を実現するための代表的な静的辞書である．**アルゴリズム 2.1** に表の構築と，挿入，探索のアルゴリズムの疑似コードを示す．

FKS スキームは 2 段のハッシュ表から構成される．一番目の特徴として，FKS スキームでは，与えられた元 x が 1 段目のハッシュ関数 h で変換され，同じハッシュ値 $j = h(x)$ をもつ元どうしは，さらに 2 段目のハッシュ関数 $k = h_j(x)$ でバケットの相対番地に変換される．二番目の特徴として，通常のハッシュ表では，データが与えられる前にハッシュ関数を決めておくが，FKS スキームでは，与えられた集合 S に対して，あるハッシュ関数の族を網羅的に探索して，全ての元 $x \in S$ に対して衝突を起こさないような一組のハッシュ関数を選択し，これを用いて S の元を表に格納する．

理論的な解析により，1 段目と 2 段目のバケットサイズを注意深く選ぶことで，S の要素に対する 1 段目での衝突をたかだか $O(n)$ 個におさえ，2 段目での衝突を起こさないようなハッシュ関数を選べることが示される．以下では FKS スキームでこれを可能にする一組のバケットサイズとハッシュ関数の具体的な構成法をみていく．

今，値の全体集合を $U = \{0, 1, \ldots, M-1\}$ とおく．バケットサイズを $s \geq 1$ として，万能ハッシュ関数の族

$$\mathcal{H}_s = \{h : U \to [0..s) \mid h(x) = (kx \bmod p) \bmod s, k \in [1..p)\}$$

を考える．剰余群を用いた $\mathcal{H}_s$ の構成方法に関する詳細は，**??**節を参照されたい．ここに，$p \in \mathbb{Z}$ は素数であり，$\mathbb{Z}_p^* = [1..p)$ は素数 p を法とする剰余類である．整数パラメータ p, s は，$1 \leq s \leq M \leq p$ を満たすと仮定し，s の値は後で適切に定めるとする．[†2] 第 1 段目のハッシュ関数 $h \in \mathcal{H}_s$ は，入力元 $x \in S$ を s 個の異なるハッシュ値 $j = h(x)$ に割り当てる．したがって，h によって，集合 S は W_1 から W_s までの s 個の「ブロック」に分割される．ここに各ブロックは

$$W_j = W_j^{(h)} = \{x \in S \mid h(x) = j\}, \quad \forall j \in [1..s]$$

[†2] 作業メモ: $s \mapsto m$ とすること．素数 p を法とする有限体 $\mathbb{F}_p$ と，万能ハッシュ関数 $g_a(x) = ax \pmod p$ を導入しておくこと，

である．ブロック $W_j^{(h)}$ は h の選択に依存する．次に，第 2 段目のブロックに対して，s 個のバケットサイズ $s_1, \ldots, s_s \geq 1$ と s 個のハッシュ関数 $h_1, \ldots, h_s$ を選ぶ．各 j に対して，第 2 段目のハッシュ関数 $h_j \in \mathcal{H}_{s_j}$ は，第 j 番目のブロックに分類された元 $x \in W_j$ を，さらに s_j 個の異なるハッシュ値 $k = h_j(x)$ に割り当て，この値にしたがって x を s_j 個のバケットのどれか（$W_{j,k}$ で表す）に割り当てる．

このとき，Fredman et al. [1984] は，もしハッシュ関数 h が族 $\mathcal{H}_s$ から一様ランダムに選ばれたとすると，

$$\mathop{E}_{h}\left(\sum_{0\leq j<s}\binom{|W_j|}{2}\right) \leq \frac{n(n-1)}{s}$$

が成立することを示した．これより，Markov の不等式を用いて

$$\mathrm{P}_h\left(\sum_{0\leq j<s}\binom{|W_j|}{2} < \frac{2n(n-1)}{s}\right) \geq \frac{1}{2} \tag{2.1}$$

が導かれる．[†3] ここで $s = 2(n-1)$ と選ぶと，式(2.1) から，関数 h を $\mathcal{H}_s$ から一様ランダムに選択しているので，$h \in \mathcal{H}_s$ のうち少なくとも半分は事象

$$\sum_{0\leq j<s}\binom{|W_j|}{2} < \frac{2n(n-1)}{s} = n \tag{2.2}$$

を満たすことがわかる．そこで，与えられた S に対して，そのような関数のうち一つを $\mathcal{H}_s$ から選んで，第 1 段目で S を s 個のブロック $W_1, \ldots, W_s$ に分割するのに用いる．[†4]

次にこれらの s 個のブロックに対して，それぞれ，第 2 段目の s 個のハッシュ関数 $h_1, \ldots, h_s$ を選ぶ．ここで，上の技法を適用しよう．再び式(2.1) に関する

[†3] 式(2.1) の導出は Markov の不等式 $Pr\{X \geq t\} \leq E\{X\}/t$ による．ここに，$X \geq 0$ は確率変数であり，$t > 0$ は任意の正数である．実際，$X = \sum_{0\leq j<s}\binom{|W_j|}{2}$ と $t = \frac{2n(n-1)}{s}$ とおくと，$P\{X \geq t\} \leq E\{X\}/t$ から，$P\{X < t\} = 1 - P\{X \geq t\} \geq E\{X\}/t = \frac{n(n-1)}{s}\frac{1}{t} = \frac{n(n-1)}{s}\frac{s}{2n(n-1)} = 1/2$ が導かれる．

[†4] このような方法を確率的議論と呼ぶ．確率的な議論を用いて，ある事象が必ず存在することを（あるいは全く存在しないことを）示すのは，慣れないと奇妙な論法だが，組合せ論における強力な証明手法の一つである．興味がある読者は，Alon and Spencer [2016] や Li et al. [2008] などの教科書を参照されたい．

アルゴリズム 2.1 FKS スキームの完全ハッシュ表

手続き MakeSet(S)

入力: 値の全体集合 $U = [0..M)$ からとられた n 個の値の集合 S

出力: 正整数列 $s, (s_j)_{j=1}^{s}$, ハッシュ関数 $h, (h_j)_{j=1}^{s}$, ヘッダー配列 $C[1..s]$, バケット配列 $B[0..b]$, バケット長 $b = \sum_j s_j = O(n)$

作業変数: バケットサイズ $(s, (s_j)_{j=1}^{s})$, バケット $(W_j, (W_{j,k})_{k=1}^{s_j-1})_{j=1}^{s-1}$

1: $s = 2(n-1)$
2: **for** $h \in \mathcal{H}_s$ **do** ▷ 第 1 段目のハッシュ関数を探す
3: $W_j = \{x \in S \mid h(x) = j\}, j \in [1..s)$ を S から求める.
4: **if** $\sum_{0 \le j < s} \binom{|W_j|}{2} < n$ **then**
5: **break** for-loop ▷ ハッシュ関数 h が見つかった
6: **for** $j \in [1..s)$ **do**
7: $s_j = \max\{1, 2|W_j|(|W_j|-1)\}$
8: **for** $h_j \in \mathcal{H}_{s_j}$ **do** ▷ 第 2 段の第 j ブロックのハッシュ関数を探す
9: $W_{j,k} = \{x \in S \mid h(x) = k\}, k \in [1..s_j)$ を S から求める.
10: **if** $\sum_{0 \le k < s_j} \binom{|W_{j,k}|}{2} < 1$ **then**
11: **break** for-loop ▷ ハッシュ関数 h_j が見つかった
12: **for** $j \in [1..s)$ **do** $C[j] = \sum_{i=1}^{j-1} s_i$ ▷ ヘッダー配列 $C[1..s]$
13: $b = \sum_{i=1}^{s} s_i$ ▷ バケット長
14: **for** $i \in [0..b)$ **do** $B[i] =$ `null` ▷ バケット配列 $B[0..b)$]
15: **for** $x \in S$ **do** Install(x) ▷ データを挿入する
16: **return** $(h, (h_j)_{j=1}^{s}), C[1..s], B[0..b], b$;

手続き Install(x)

17: $B[C[j] + h_j(x)] = 1$ **with** $j = h(x)$

手続き Find(x)

18: $y = B[C[h(x)] + h_{h(x)}(x)]$
19: **if** $x = y$ **then return** `true`
20: **else return** `false`

観察から，今度は各 $j \in [1..s]$ に対して，第 j 番目のハッシュ関数のバケットの大きさを $s_j = \max\{1, 2|W_j|(|W_j|-1)\}$ と設定する．ここで，関数 h_j で値 k へハッシュされる W_j 中の値の集合を $W_{j,k} = \{x \in S \mid h(x) = k\}, k \in [1..s_j)$ とおく．すると，第 1 段目と同様に，第 2 段目のブロック $W_{j,k}$ においても $\mathcal{H}_{s_j}$ のうちで少なくとも半分のハッシュ関数が

$$\sum_{0 \le k < s_j} \binom{|W_{j,k}|}{2} < \frac{2n(n-1)}{s_j} = 1$$

を満たし，衝突を生じない，すなわちブロック W_j が含む値に対して，関数 h_j は 1 対 1 関数であることがわかる．そこで，各 W_j に対して，S を用いてテストしながら，そのようなハッシュ関数 $h_j \in \mathcal{H}_{s_j}$ を一つ選ぶことで，第 2 段目が完成する．

上記の FKS スキームによって，一連のバケットサイズ $s, (s_j)_{j=1}^s$ と一連のハッシュ関数 $h, (h_j)_{j=1}^s$ が選ばれると，次のようにして連続した番地の並びであるバケット配列 $B[0..b]$ に，S の元を衝突なしに一意に格納できる．まずバケット配列の長さ b を，$b = \sum_{j=1}^s s_j$ ととる．次に B を，幅が $s_1, \ldots, s_s$ の s 個の区間 $B_1, \ldots, B_s$ に分割する．このとき，ブロック W_j の元は，区間 $B_j = B[c_j..c_{j+1}-1]$ に格納される．ここに，c_j は区間 B_j の開始番地であり，$c_j = \sum_{i=1}^{j-1} s_i$ と書ける．これを補助配列 $C[0..s)$ に $C[j] = c_j$ と記録する．ここで，配列の長さ b を見積もると

$b = \sum_{0 \le j < s} s_j \le \sum_{0 \le j < s} 2|W_j|(|W_j-1|)$ ここで $\binom{|W_j|}{2} = \frac{1}{2}|W_j|(|W_j-1|)$ を用いると，

$\le 4 \sum_{0 \le j < s} \binom{|W_j|}{2}$ となる．さらに，式(2.2) から

$< 4n$

となって，最終的に使用領域 $O(b) = O(n)$ が示される．

一組のハッシュ関数 $h, (h_j)_{j=1}^s$ が与えられると，1 段目と 2 段目のハッシュ関数を用いて，任意の元 $x \in S$ を一意かつ衝突することなくバケット配列 B 中の番地 $i_x = C[h(x)] + h_{h(x)}(x)$ に割り当てることができる．これにより，**アルゴリズム 2.1** の 17 行目のように，元 x に対応するバケット $B[i_x]$ にアクセ

スする．これにより数値演算 $+, \times, \mathrm{mod}$ をもつワード RAM 上において，格納する要素数 n の線形領域で集合 S を格納し，探索 $\texttt{Find}(x)$, 挿入 $\texttt{Insert}(x)$, 削除 $\texttt{Insert}(x)$ を $O(1)$ 最悪時間で実行可能である．

前処理では，第 1 段目と第 2 段目のハッシュ関数 $h, (h_j)_{0 \le j < s}$ を選択するのに族 $\mathcal{H}_s$ を走査する必要がある．値の全体集合は $U = \{0, 1, \ldots, M\}$ だったことを想いだそう．族の大きさ $|\mathcal{H}_s|$ は $O(p) = O(M)$ なので [†5]，一連のハッシュ関数の探索は $O(n^2 M)$ 最悪時間で実行可能である．したがって，M が入力サイズ n の多項式のとき，前処理において，表を n の多項式時間で構築可能である．例えば，ある定数 $c > 0$ と格納する値の総数 n に対して，U が高々 $w = c \log n$ ビットで表せる非負整数全体ならば，$M \le 2^{c \log n} = n^c = O(\mathrm{poly}(n))$ となり，仮定は満たされる．表の構築時間については，その後も改良が続けられている．詳しくは 2.6 節の文献ノートを参照されたい．

2.6 文献ノート

決定性の完全ハッシュ表の研究は，Fredman et al. [1984] らを嚆矢とする．彼らは，現在 FKS スキームとして知られる方式を提案し，多項式サイズの n 個の元を $O(n)$ 語の領域で格納し，定数時間の探索を実現するデータ構造を n の多項式時間で構築できることを示した．Alon and Naor [1996] らは，n 個の元を $O(n)$ 語の領域で格納し，$O(1)$ 時間で探索を実現する完全ハッシュ表を $O(n \lg^5 n)$ 決定性時間で構築できることを示した．その後も，決定性の完全ハッシュ表に関して構築時間の改良の努力が続けられている．Hagerup et al. [2001] らは構築時間を $O(n \lg n)$ 決定性時間に改善している．

データ構造の利用について，ここで学んだ辞書は，計算機の登場以来，もっとも古くから用いられて来たデータ構造の一つである．

一方，最近のネットワーク上の情報システムにおいて広く用いられている，いわゆる**キー-値ストア** (*key-value store, KVS*) は，辞書の一種である．KVS は，NoSQL の代表とされ，アカウントやパスワード等の認証情報や，システムパラメータの格納，運用情報管理など，各種の情報システムにおける情報管理

[†5] 素数 $p > M$ は $p = O(M)$ を満たすようにとれる．

の基盤として広く用いられている．代表的な KVS である `Berkeley DB` は，2020 年度の ACM ソフトウェアシステム賞を受賞している．[†6] 2000 年代後半以降は，`memcached` や `Google BigTable` などに代表される大規模な分散型 KVS の利用と研究開発が盛んになっている．

[†6] ACM ソフトウェアシステム賞は，広く影響を与えた優れたソフトウェアシステムに与えられる表彰の一つ毎年一つのグループに授賞される．第 1 回は UNIX オペレーティングシステムの開発に関する貢献で Dennis M. Ritchie と Kenneth Lane Thompson が受賞している．ACM Software System Award: https://awards.acm.org/software-system/.

第3章

順序辞書構造

本章では，各種のデータ構造構築の部品として用いられる順序辞書構造を説明する．**順序辞書** (*predecessor dictionary*) は，二分探索木の演算を抽象化したデータ構造である．任意の全順序集合を $(\mathcal{K}, \leq)$ とする．ここに，$\mathcal{K}$ は値の集合であり，$\leq \subseteq \mathcal{K}^2$ は $\mathcal{K}$ 上の全順序である．すなわち，$\mathcal{K}$ の任意の二つの元 x, y が比較可能である，すなわち，条件 $(x \leq y) \vee (y \leq x)$ が成立するものとする．本節では，ワード RAM を仮定する．

3.1 基本的な順序辞書

静的 (*static*) な順序辞書は，それぞれが高々M の大きさをもつ n 個の非負整数からなる集合 $S = \{x_1, \ldots, x_n\}$ を管理するデータ構造であり，次の演算をサポートする．

- `Find`(x) は，与えられたキー $x \in \mathcal{K}$ をもつデータが S に含まれるかに答える．もしキーが含まれたらデータへのポインタを返し，そうでなければ値 `null` を返す．
- `Predecessor(x)` は，キー値が x より小さいデータで，最大のキー値をもつものを返す．
- `Successsor(x)` は，キー値が x より大きいデータで，最小のキー値をもつものを返す．

図 3.1 に，`Predecessor(x)` 演算と `Successsor(x)` の説明を示す．も

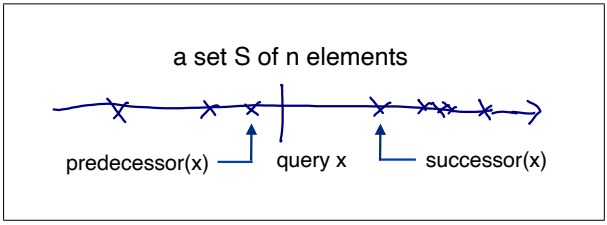


図 3.1: 順序辞書における `Predecessor(x)` と `Successor(x)` 演算

し演算として `Find`(x) しか持たなければ，これはハッシュ表のような順序のない静的な**辞書構造** (*dictionary*) となる．上の演算のうち，`Predecessor(x)` と `Successsor(x)` の二つの演算は，**最長接頭辞演算** (*longest common prefix operation*) と呼ばれることもある．これらは，キー値 x が S に含まれる場合は，`Find`(x) と同じデータを返す．これらの二つの演算は対称性をもつので，どちらか一つの実装だけを与えることが多い．本節では `Predecessor(x)` の実装だけを考え，`Successsor(x)` は省略する．

最も基本的な順序辞書は，整列された配列を用いるものであり，これは静的な辞書である．この実装法では，静的に割り当てた配列上に，n 個の元からなる集合を $O(n \lg n)$ 時間で整列して $O(n)$ 語で格納し，`Find`(x) と `Predecessor(x)` を二分探索を用いて $O(\lg n)$ 最悪時間で実現する．

動的 (*dynamic*) な順序辞書は，静的な順序辞書に次の演算を追加したものである．

- `Insert`(x, y) は，与えられたキー $x \in \mathcal{K}$ をもつデータ y が S に含まれなければ，S に挿入する．もしキーが含まれたら何もしない．
- `Delete`(x) は，与えられたキー $x \in \mathcal{K}$ をもつデータ y が S に含まれるならば，S からそのデータを削除する．もしキーが含まれなければ何もしない．

最も基本的な動的な順序辞書は，234 木や赤黒木などの**平衡二分探索木** (*balanced binary search tree*) である．これらは，各ノードに元 x をもち，その左の部分木（右の部分木）に含まれる任意の元は x より小さい（x より大きい）と

いう**二分探索木性** (*binary search tree property*) と呼ばれる性質を満たす根付き木である．さらに，木の高さを $O(\lg n)$ に保つため，挿入と削除を通して全ての葉の深さが高々1つだけ異なるように管理される [†1]．平衡二分探索木は，n 個の元からなる動的な集合を $O(n)$ 語で格納し，`Find`(x) と `Predecessor(x)` を `Insert`(x, y) と `Delete`(x) を $O(\lg n)$ 最悪時間で実現する．これらの平衡二分探索木を用いて，次の結果が示される．

定理 3.1（Cormen et al. [2009]） 全順序集合の n 個の元からなる集合 S を $s = O(n)$ 語の領域で格納し，動的順序辞書の任意の演算を $O(\lg n)$ 時間で実行するように実現できる．

平衡二分探索木を用いた実装において，`Predecessor(x)` 演算（または `Successsor(x)` 演算）は，キーが集合 S に含まれない場合は，そのキーで到達可能な最も深いノード v を発見して，その左の子として直前のノードを（または右の子として直後のノードを）返す．このとき，キーのパスと S の元のパスの最長接頭辞にノード v が対応することから，`Predecessor(x)` と `Successsor(x)` の二つの演算をまとめて，**最長接頭辞** (*longest common prefix*) 演算と呼ぶことがある．

3.2 $O(\lg \lg n)$ 演算時間: van Emde Boas 木

キー値の大きさが限定されている場合には，高速な静的順序辞書を実現できる．**van Emde Boas 木 (vEB 木)** (*van Emde Boas tree*) は，値が高々 $u > 0$ の n 個の非負整数からなる集合 $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$ を前処理し，`Predecessor(x)` を $t = O(\lg \lg u)$ 最悪時間で実現する．ただし領域効率はあまり良くなく，実際のデータの個数 n と無関係に，格納には $O(u)$ 語の領域が必要である．

思考実験として，`Find` 演算のみを実現すれば良く，`Predecessor` 演算は不要な場合を考えよう．[†2] このときに，$s = O(u)$ 語領域を用いて良いならば，値の

[†1] 平衡二分探索木における挿入演算と削除演算の実装の詳細については Cormen et al. [2009] などの教科書を参照されたい．

[†2] これは，静的辞書そのものである．

領域 $[0..u)$ を長さ u の固定長のビット配列 $B[0..u) \in \{0,1\}^u$ で表せば良いことがすぐにわかる．クエリ q に対する $\texttt{Find}(q)$ 演算は，もし $B[q] = 1$ なら q を返し，それ以外は `null` を返せば良い．しかし，このやり方では `Predecessor` 演算は上手く行かない．もし $\texttt{Find}(q)$ 演算は，もし $B[q] = 0$ だったとしよう，すると q の先行者を求めるにはには，添字 q から左側を逐次的にみていって，$B[j]$ となる添字 $j < q$ を見つける必要があるが，これは $O(u)$ 時間を要する．

この問題に対して，vEB 木では，次のようなアイディアに基づいて探索を行う．まず配列 B に各要素データを表すビットを直接入れるのでなく，データを塊にしてそれを配列の要素に格納する．具体的には，ブロックサイズとブロックの個数のバランスをとるために，$u = u' \times u'$ となる数，すなわち $u' = \sqrt{u}$ をとって，上記の配列 B を高々u' 個のサイズ $O(u')$ のブロックに分割する．さらに，各ブロックが一つ以上の要素を含むかどうかが，瞬時にわかるようにする．

これにより，もしクエリ q を用いた添字演算で見つけたブロックが，一つ以上の要素を含めば，再帰的にその中を探せば良い．要素を一つも含まないならば，空でないブロックを検索するための上位構造を再帰的に用いて，そのようなブロックを見つければ良い．これを再帰的に繰り返すことで，`Predecessor` 演算を実現できそうである．

3.2.1 構築

vEB 木は次のように再帰的に構築される．現在のレベルにおいて，領域サイズ u のデータ $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$ が与えられているとする．現在の領域サイズ u に対して，一つ下のレベルの領域サイズを $u' = \sqrt{u}$ とおく．このとき，S の vEB 木 T は次の情報をもつ．

- 準備として，はじめに S の要素を昇順で整列し，高々u' 個のサイズが高々 u' の部分集合列（ブロックと呼ぶ） $S_1, \ldots, S_{u'}$ に値が小さな方から大きな方へ分割しておく．ここに，$1 \leq i \leq u'$ に対して $|S_i| = O(u')$ である．
- もし S が空でなければ，S の値の最小値 $T.\texttt{min} = \min S$ と最大値 $T.\texttt{max} = \max S$ を保持する．もし S が空ならば，$\texttt{min} = +\infty$ および $\texttt{max} = -\infty$ を保持する．

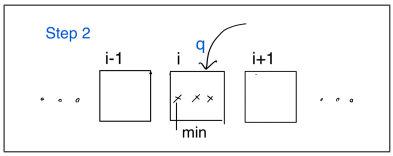
- 上位レベルの構造として，u' 個のブロック中で，空でないブロックの添字からなる集合 $S_0 = \{i \in 1..u' \mid S_i \neq \emptyset\} \subseteq [0..u')$ を考えて，これを再帰的に格納する vEB 木 $T.Galaxty$ を保持する．そのサイズは $O(u')$ である．
- 下位レベルの構造として，u' 個のブロック $S_1, \ldots, S_{u'}$ をそれぞれ格納する u' 個の vEB 木 $T_1, \ldots, T_{u'}$ を要素にもつ長さ u' の配列 $T.Cluster[1..u']$ を保持する．

再帰的分割により，ある定数 $E = O(1)$ に対して領域サイズ u が $u < E$ となったら，分割を停止する．

3.2.2 探索

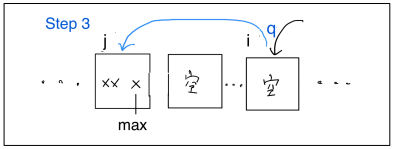
vEB tree の探索は次のように行う．現在のレベルの領域サイズ u に対して，一つ下のレベルの領域サイズを $u' = \sqrt{u}$ とする．

- **Step 1:** キー $q \in [0..u)$ を受け取ると，キーをビット列と考えて，上位ビット $i = q/u' \in [0..u')$ と下位ビット $k = q \bmod u' \in [0..u')$ にわける．クラスター配列の直接アクセスで、キー q を範囲に含む第 i 番目のクラスタ $T_i = T.Cluster[i]$ へ行く。この T_i をキー箱と呼ぶ。ここで、必ずしもキー箱 T_i はデータとしてキー q を含むとは限らないことに注意しよう。キー箱が. キーを含むならば Step 2 へ行き、含まないならば Step 3 へ行く。
- **Step 2:** もしこのキー箱 i が含む最小値 $T.\texttt{min}$ よりキー q が大きければ、キーより小さい値が少なくとも一つはあるので、下の図のようにキーの先行者が必ずこの箱 i の中に含まれることが分かる。よって、キー箱が先行



者箱である。よって後はキー箱の vEB 木 T_i を用いて，その中に含まれるキー k の先行者のアドレス $i_* = T_i.\texttt{Predecessor}(k)$ を見つける．最後に先行者の全体のアドレスとして $u'(i-1) + i_*$ を返せば良い。

- **Step 3:** それ以外のときは $q < T.\texttt{min}$ なので，先行者がこの箱の中にはないことがわかる。するとキー q の先行者は，このキー箱 i より左側にある箱の並びの中で，最初に見つかる空でない箱 $j < i$ の中にあるはずである。そこで、下の図のように，キー箱の添字 i をキーとして $T.Galaxy$ を探索し、$j < i$ かつ空でない箱の添字 $j = T.Galaxy.\texttt{Predecessor}(i)$ を見つけて，対応する vEB 木 $T_j = T.Cluster[j]$ を取り出す．すると，$j < i$



よりキー q は先行者箱 j の中のすべての値よりも大きいはずなので、キー q の先行者は先行者箱 j の中の値の最大値 $T_j.\texttt{max}$ になる。

以上の議論から，上記の手続きが van Emde Boas 木において正しく `Predecessor` 演算を実現することがわかる．

3.2.3 解析

次に，van Emde Boas 木における探索時間を解析する．ここで一般性を失うことなく，領域サイズの下限を $E = 2$ とする．領域サイズ u に対して，クエリ時間 $T(u)$ は次の再帰式で与えられる:

$$T(u) \leq T(u^{\frac{1}{2}}) + C,\ u \geq E = 2. \tag{3.1}$$

ここに，$C, D > 0$ を定数とし，$u \leq 2$ に対して $T(u) = D$ と仮定する．ここで，式(3.2) を $k \geq 0$ 回展開すると，

$$T(u) \leq T(u^{\left(\frac{1}{2}\right)^k}) + k \cdot C \tag{3.2}$$

の形になる．この展開は，次の式が成立している限りは実行できる：

$$u^{\left(\frac{1}{2}\right)^k} \geq E = 2. \tag{3.3}$$

式(3.3) を k について解くために，両辺を続けて 2 回 lg をとる．まず右辺については，$\lg\lg 2 = \lg 1 = 0$ となり，左辺については，

$$\lg\lg u^{\left(\frac{1}{2}\right)^k} = \lg\left((1/2)^k \lg u\right) = k\lg(1/2) + \lg\lg u = -k + \lg\lg u$$

となる．よって，左辺と右辺を合わせて，$k \leq \lg\lg u$ を得る．これを式(3.2) に代入して

$$T(u) \leq kC \leq C\lg\lg u$$

を得る．以上の議論から，探索時間は $O(\lg\lg u)$ 決定性時間であることを示した．

3.2.4 議論

vEB tree では、空の可能性も許した上で，キーを含み得る箱を見つける作業を，配列の直接アクセスで実現するのが効率化の鍵である．Step 1 の配列アクセスで空な箱が見つかったときは、Step 3 では先行する空でない箱 T_j を見つけるのに $T.Galaxy$ を用いた再帰的な探索が 1 回必要であるが，箱が含む値の最大値 $T.\texttt{max}$ を記録しておくことで、箱の中の再帰的な先行者探索を避けられる。

これにより，Step 2 と Step 3 のどちらを実行する場合も，再帰の繰り返しで 1 回しか子を呼び出さないため，クエリ時間の再帰式が $T(u) = T(u^{\frac{1}{2}}) + C$ となる．反対に，Step 1 で空でない先行者箱を、索引を再帰的に用いて見つけるやり方では，再帰式が $T(u) = 2T(u^{\frac{1}{2}}) + C$ となってしまうので、これを解いてもクエリ時間が $t = O(\lg\lg n)$ とならない．[†3]

vEB 木の弱点は，格納に $O(u)$ 語領域を必要とする点である．これを実際に格納される元の数 n の線形領域で済ませるのが，当面の目標である．

3.3 $O(\lg\lg n)$ 演算時間と線形領域: y-fast トライ

Y-fast トライ (*Y-fast trie*) は，vEB 木のメモリ効率の悪さを，データ数 n の線形領域に改善したものである．Y-fast トライは，深さ方向の二分探索を行う拡張されたトライ構造と，領域を削減するための平衡二分探索木を組み合わせて構築される．

[†3] 実際，$t = O(\lg n)$ となってしまう。

定理 3.2（Willard [1983]） 高々u の大きさをもつ n 個の非負整数からなる集合 $S = \{x_1, \ldots, x_n\} \subseteq [0..u)$ を $O(n)$ 語の領域で格納し，静的順序辞書の任意の演算を $t = O(\lg \lg u)$ 決定性時間で実現する．

定理 3.2 の系として，キーの値の上限 u が入力サイズ n の多項式のとき，y-fast トライは $t = O(\lg \lg n)$ 最悪クエリ時間を達成する．このことは，仮定よりある定数 $c > 0$ に対して $u \leq cn^c$ なので，最悪クエリ時間は，$t = O(\lg \lg(cn^c)) = O(\lg \lg n + \lg c + \lg \lg c) = O(\lg \lg n)$ となることから直ちにわかる．

本小節の残りでは，Y-fast トライを説明し，上記の定理の証明を与える．

3.3.1 トライを用いた順序辞書の実現法

X-fast トライと Y-fast トライを説明するために，それらの素朴な原型として，辞書が `Find`(x) 演算しかもたない場合を考えてみよう．先に述べたように，これは順序のない静的な辞書そのものであるから，すべての元を整数キーとして登録すれば，ハッシュ表を用いることで，$O(n)$ 領域と $O(1)$ 期待クエリ時間で実現できる．

次に順序辞書に拡張するために，`precessor` と `successor` の最長接頭辞クエリの実現を考える．これらのクエリでは，クエリのキー x が集合 S に含まれていない場合でも，S に含まれる「最も近い」キーを見つける部分的な照合が必要である．トライを用いた順序辞書の実現法は次の通りである．

- 空の二進トライ $\mathcal{T}$ を生成する．
- S が含む整数キー $x \in [0..u)$ それぞれに対して，次を行う：
  - キー x を表現する長さ $d = O(\lg u)$ ビットの二進数列 $z = bin(x) \in \{0,1\}^d$ を求める．
  - 二進列 z の桁数が d 未満ならば，高位ビット（左側）に 0 を詰め物して，長さ d をもつようにする．
  - キー z をパスとして用いて，データをトライ $\mathcal{T}$ に挿入する．

このトライは，深さが d であり，各辺のラベルは $\Sigma = \{0,1\}$ をもち，内部ノードは高々二つの子をもつ．トライにおいて，整数キー $x \in [0..u)$ が与えら

れたとき，Find(x) と Predecessor(x) を，次の探索手続きを用いて，木の深さ d に比例した時間で実現できる．

- キーの二進文字列 $bin(x)$ を先頭から 1 ビットずつ読みながら，そのビットをもつ辺を選択しつつ，根から葉の方向へ降りていく．
- あるノード v に到達してそれ以上進めなくなったら停止する．もしクエリのすべてのビットを読み終わっていれば v が答えの葉であるので，それが格納するデータへのポインタを返す．
- 途中で終わっていれば，クエリのキーは S に含まれていないので，木の**通りがけ順** (*in-order*) で，v より左側の最近の葉を見つけて返す．

Insert(x, y) と Delete(x) 演算も探索と同様に，木の深さ d に比例した時間で実現できる．このトライの探索は，木の深さ，すなわち，キーのビット長に比例した時間である $t = d = O(\lg u)$ 最悪時間で実行できる．ただし，平衡二分探索木と比較した場合，値の上限 u が入力の線形サイズ $O(n)$ 程度に小さくても，$t = O(\lg n)$ 時間と平衡二分探索木同じ時間を要する．もしキー長が n の多項式より大きい場合は，平衡二分探索木よりも遅くなる可能性がある．

3.3.2 X-fast トライ: ビット方向の二分探索による高速化

前項のトライの探索時間を対数的に高速化するために，木のパス方向に二分探索を行うことを考える．

- まず，トライを高さ $k \in [0..d)$ 毎に水平方向に輪切りにして，各高さのノードの集合を第 k 番目のレベル L_k と呼ぶ．
- 次に，レベル $L_0, \ldots, L_d$ に対応する d 個の空のハッシュ表 $(H_0, \ldots, H_d)$ を用意する．これらのハッシュ表 H_k は範囲 $[0..u)$ の整数キーをもつ．
- 各レベル k に含まれるノード v に対して，根から v へのパスが表すビット列 $\beta = bit(v) \in \{0,1\}^k$ に対して，ノード v を値とし，キー $key(\beta, k) = \beta \cdot 0^{d-k} \in \{0,1\}^d$ を非負整数としてハッシュ表 H_k に挿入することを繰り返す．

ここに，ビット列の下位ビットには 0 を詰め物して，長さをちょうど d にして

いる．各 $k \in [0..d)$ のハッシュ表 H_k には，値のビット列を先頭 k ビットで打ち切って，後半を 0 で詰め物したキーが挿入されたわけである．

X-fast トライにおいて，キー $x \in [0..u)$ が与えられたとき，d 個のハッシュ表 $(H_0, \ldots, H_d)$ において，探索を次のように行う．

- $m = \lceil d/2 \rceil$ から開始して，$k \in [0..d)$ に関して二分探索を行い，ハッシュ表 H_k がキー $key_k = key(bin(x), k)$ を含む最大の $\hat{k} = k$ を見つける．
- このとき，$\hat{k}$ に対する値 $v = H_k[key_{\hat{k}}]$ が求める最長接頭辞のノードである．
- 見つけた v から `Predecessor`(x) となる葉 $w \in S$ を $O(1)$ 時間で見つけることができる（詳細は省く）．

ここで X-fast トライの時間と領域量について解析する．トライの深さが $d = O(\lg u)$ であるので，二分探索により $t = O(\lg d) = O(\lg \lg u)$ 時間で答えとなる w を見つけられる．領域計算量については，トライは d レベルをもち，分岐しないノードもあるため，最悪時に各レベルは $O(n)$ 個のノードを含むため，X-fast トライは，全体で $s = O(n \lg u)$ 語の領域を要してしまう．

3.3.3 Y-fast トライ：ブロック分割による省メモリ化

ブロック分割を用いて並行二分探索木を組み合わせることで，X-fast トライの領域量を改善したものが，**Y-fast トライ** (*Y-fast trie*) である．以下では，Y-fast トライの構成を与える．

今，ブロックサイズを $b = O(\lg u)$ とおく．

- 最初に，キーの集合 S を昇順に整列する．S を先頭から見ていき，サイズが高々b 個の元を含むブロックに区切る．結果として S を，$m = \lceil n/b \rceil = O(n/\lg u)$ 個のブロック $B_0, \ldots, B_{m-1}$ に分割する．
- 次に「大きな木」(macro tree) として，ブロック B_i の最小値をキーとし，ブロック自体をデータとする X-fast トライ $\mathcal{T}$ を構築する．
- 最後に「小さな木」(micro tree) として，m 個のブロック $B_0, \ldots, B_{m-1}$ にそれぞれ対応する m 個の平衡二分探索木 $\mathcal{R}_0, \ldots, \mathcal{R}_{m-1}$ を構築する．

`Predecessor`(x) の探索は次のように行う．`Find`(x) の実現は，解が葉で

ある場合い制限された特殊な場合として実現できるので，省略する．

- キー x が与えられると，はじめに大きな木を用いて $t_0 = O(\lg\lg u)$ 時間で，そのキーを範囲に含むブロック B_i $(i \in [0..m))$ をみつける．
- 次に $b = O(\lg u)$ 個の元を含む小さな木で x を再度探索し，$t_1 = O(\lg b) = O(\lg\lg u)$ 時間で解を見つける．

定理 3.2 の証明を示す．

証明: 探索の構成から，Y-fast トライにおいて，`Predecessor`(x) 演算を $t = t_0 + t_1 = O(\lg\lg u) + O(\lg\lg u) = O(\lg\lg u)$ 時間で実行可能である． 最後に，領域量をみつもる．大きな木 $\mathcal{T}$ は間引いて選んだ $m = O(n/\lg u)$ 個のキーを含むので，因数 $\lg n$ が打ち消し合い，領域量は $s_0 = O(m\lg u) = O((n/\lg u)/\lg u) = O(n)$ 語となる．小さな木それぞれは b 個のキーを含むので $s_1 = O(b) = O(\lg u)$ 語領域であり，全体で m 個の小さな木全体の領域量の総和は，$s_{1*} = O(s_1 m) = O(\lg u \cdot (n/\lg u)) = O(n)$ 語領域となる．合わせると領域量は $s = s_0 + s_{1*} = O(n)$ 語領域となるので，定理 3.2 が示された． □

参考文献

N. Alon and M. Naor. Derandomization, witnesses for boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16 (4-5):434–449, 1996.

N. Alon and J. H. Spencer. *The probabilistic method*. John Wiley & Sons, 2016.

T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. MIT press, 3rd edition, 2009.

M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer Auf Der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1989.

T. Hagerup, P. B. Miltersen, and R. Pagh. Deterministic dictionaries. *Journal of Algorithms*, 41(1):69–85, 2001.

M. Li, P. Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.

R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2): 122–144, 2004.

D. E. Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2):81–84, 1983.

索　引