解　説

ハッシング・プロセッサ*

後藤英一** 井田哲雄***

1. 序　論

ハッシュ（符号）法（Hash Coding）と呼ばれる連想的高速索表法は，Knuth[1]によれば，1950年代にIBMにおいて，アセンブラとコンパイラの変数名表の検索の高速化に使われたのがその起源とされ，またソ連のErshovによっても同じ頃独立に使われたとされている．また，ハッシュ（HASH）法という名称が定着したのは比較的新らしい[1]．ハッシュ法に関しては非常に多くの論文が発表されており，最近の英米の総合報告[2),3)]には150以上の文献のリストがあるが，残念ながら本会の論文賞を受賞した2編の論文[4),5)]はこれらのリストには漏れている．各種のハッシングアルゴリズムの解析はKnuth[1]に詳細に述べられており，またMorrisの論文[6)]はハッシュ法の研究史上大きな影響を与えたものとされている．ハードウェアに関して，MaurerとLewisはその総合報告[3)]に，これだけ広く使われているのだからハードウェアを作る意味は十分にあろうと述べるとともに，ハッシング・ハードウェアは研究（Research）のテーマというよりも，開発（development）の問題だとしているが，研究はもういらないという点は必ずしも当を得ていない．

本稿では，まず**2.**で純ソフトウェア上のハッシュ法とその問題点およびハッシュ法の応用について述べる．**3.**では筆者たちの提案したハッシング・ハードウェアについて説明する．

2. ソフトウェア向きのハッシングアルゴリズムの問題点とその応用

ハッシュ法は鍵（Key，データ名）の“衝突”（Collision，定義は後出）を解決する方法により，開アドレス法（Open Addressing）と連鎖法（Chaining）に類別される．それにMorris[6)]の提案した指数表（(Index Table) その実体は間接アドレス表）を使用するしないの区分を含めると4種類の方法がある．**図-1**はこ

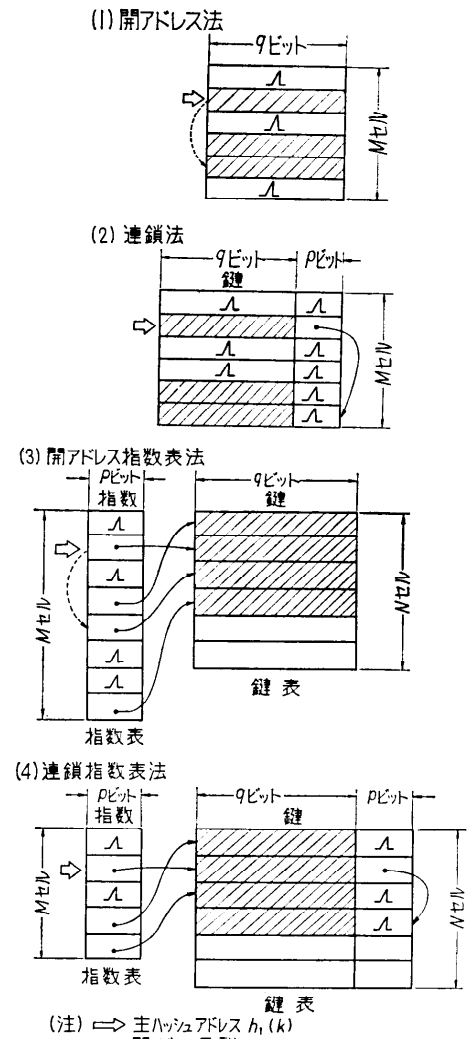

図-1　ハッシュ法の分類

* Hashing Processor by Eiichi GOTO (Department of Information Science University of Tokyo) and Tetsuo IDA (The Institute of Physical and Chemical Research).

** 東京大学理学部
*** 理化学研究所

れらを固定長鍵の場合について示したものである．いずれの方法でも鍵 k の登録と検索に鍵空間から，ハッシュアドレス空間への写像(関数) $h_1(k)$ を使う．$h_1(k)$ で指定された場所（指数表の場合には1回間接アドレスを経由）に k とは異なる鍵 k' がすでに入っている場合を衝突という．開アドレス法では衝突があると，再ハッシュ関数列 $h_2(k), h_3(k), \cdots$ を順次に衝突がなくなるまで試行する．関数列 $h_1(k), h_2(k), \cdots$ の生成法については数多くの論文があるが[2),3)]，ハッシュアドレスに重複がないもの，すなわち，$(h_1(k), h_2(k), \cdots, h_M(k))$ が整数の組 $(1, 2, \cdots, M)$ の順列になっているもので，かつ鍵 k に応じて，順列が擬似乱数的に選ばれるものが一般に良い結果を与え，Knuth[1)] はこれを一様ハッシュ法 (Uniform hashing) と名付けている．さらにハッシュ関数列は高速に生成できることが要求されるが，特にその生成をハードウェア化する場合には，乗除算は使用せず，ビットの桁混合（これはソフトウェアでは高速化し難い）とマスキングと加減算だけの組み合わせで生成するのが有利であろう．連鎖法では，衝突をポインターの連鎖を使って処理するが，この方法は開アドレス法でそれまでに1回でも使用した再ハッシュ関数列 $h_2(k), h_3(k), \cdots$ をすべて表中のポインター部に記録しておいて再計算しないようにしたものとみることができる．

図-1 の4種の方法の長短得失は，後に示すように，対象とするデータ構造の性質にも依存するが，それらの評価の基礎となる式を**表-1** に示す．ここで注意すべき点は，従来記憶装置の利用率を示す量として，表の負荷率 α=(表中の有効鍵数, n)/(表に入れ得る鍵の総数, M) がもっぱら使われてきた．しかしこれではポインターなどに使われる記憶使用量が正しく考慮されていない．そこで筆者たち[7)]は負荷率 α に加えて，記憶利用率 σ=(表中の有効情報量)/(ポインター等の補助情報を含む表の全情報量) を導入した．α から σ への換算には，鍵の相対長 κ=(鍵のビット長, q)/(ポインターのビット長, q) と指数表の相対長 τ=(指数表の項目数, M)/(鍵表の項目数, N) が関与する[7)]．

つぎに探索時間を示す量としては従来から使われてきた索表平均回数（ハッシュ表の記憶アクセス回数にほぼ等しいが，指数表とポインターの間接アドレス引用時間と鍵の書き込み時間は従来の伝統的解析結果との比較の都合上含めない）を使用する．また従来は，成功索表平均回数 S (Successful Search, Knuth[1)] の c_N) と不成功索表平均回数 U (Unsuccessful Search, Knuth[1)] の $c_{N'}$) のみが考慮されてきたが，鍵挿入索表平均回数 I (Insertion) も別個の量として表-1 に記載してある[7)]．また表-1 に古川[4)]の衝突タグビット法の場合も記載した．古川法では $U<I$ となり，U と I は一致しない．なお表-1（表-2 も同様）の式は M と N が十分に大きいとした場合の漸近関形数（連続近似)を示す[7)]．従来の文献[1)~6)]には見られない式の導出は文献[7)]にある．表-1 によって同一記憶利用率 σ での索表回数を数値的に評価した結果[7)] は従来の表の負荷率 α に基づく評価と比較して，指数表なし開アドレス法が有利な場合が多く，特に $\kappa \leq 2$ の短い鍵の場合には開アドレス古川法が索表回数の少ない点で最良という結果になっている[7)]．

つぎに可変長(任意長)鍵の処理法と各鍵にデータを付値する方法について述べる．鍵は文字列とし，1語には2文字詰めるとし，ϕ は無効文字，nil は文字列の終りを示す特殊記号 (例えば nil=$\phi\phi$) である．**図-2** (次頁参照)の方法 (Knuth[1)] p. 549, EX. 65, Gries[8)]

表-1 鍵の削除がない場合の索表平均回数の記憶利用率 σ に対する依存性

Algorithm	U	I	S	$\sigma \leq \sigma_m$	σ_m
U*	$\frac{1}{1-\alpha}$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$	α	1
UF*	$\frac{1}{1-\alpha}\frac{1}{1-\ln(1-\alpha)}$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$	α	1
CC**	$1+\frac{1}{4}(e^{2\alpha}-1-2\alpha)$	$1+\frac{1}{4}(e^{2\alpha}-1-2\alpha)+\alpha e^{\alpha}$	$1+\frac{1}{8\alpha}(e^{2\alpha}-1-2\alpha)+\frac{1}{4}\alpha$	$\frac{\kappa}{\kappa+1}\alpha$	$\frac{\kappa}{\kappa+1}$
SO†	$\frac{1}{1-\alpha}-\alpha$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\ln(1-\alpha)$	$\frac{\kappa\tau}{\tau+\kappa}\alpha$	$\frac{\kappa}{\tau+\kappa}$
SC††	$e^{-\alpha}+\alpha$	$1+\alpha$	$-\frac{1}{2}\alpha$	$\frac{\kappa\tau}{\tau+\kappa+1}\alpha$	$\frac{\kappa}{\tau+\kappa+1}$

* Uniform hashing
** Uniform hashing with Furukawa's collision flag method
*** Coalesced Chaining
† Scatter index table technique with Open addressing
†† Scatter index table technique with Chaining

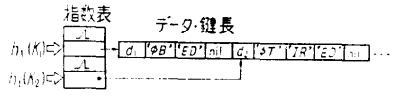

図-2 開アドレス指数表法に基づく可変長鍵の処理

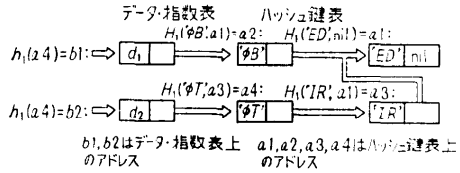

図-3 鍵ポインタ法による可変長鍵の処理

参照）は鍵 k_1="BED", k_2=TIRED" に関するそれぞれ1語長のデータ d_1 と d_2 を鍵表上に置く可変長鍵データ表を伴う開アドレス指数表法である．**図-3** の方法[9),10)]では開アドレス指数表上にデータを置き，鍵は短い文字組（図-3 では2文字組）のリスト構造に分割し，鍵表では文字列 c とリスト構造のポインター p の両者を探索の鍵とする開ハッシュ関数列 $H_1(c, p)$, $H_2(c, p)$, … を使う．図-1 の連鎖法では，ポインターは探索の鍵には使用しないのに対し，図-3 の場合には，ポインターも鍵に使用するので鍵ポインター法と呼んでおく．可変長鍵の文字列を末尾から（先頭からでも同様）2文字組のリストに分割して順次ハッシュ鍵表に登録する．従って，同一リスト構造に対して唯一の代表が登録されることになる．図-3 の例では $a1$: ("ED" nil) という構造は唯一個しか登録されないため，"BED" と "TIRED" に共用される．文字列"BED" が与えられると，簡単のため衝突はないとして，$a1$=H_1("ED", nil), $a2$=H_1('ϕB', $a1$), $h_1|a2$) と3回のハッシュ探索（次章のハードウェアを使えば，ほぼ3回の記憶アクセス時間で完了）を経てデータ d_1 が取り出される．図-3 の方法では，指数表，鍵表のいずれも2語長の固定長をもつ．次章の並列ハッシュハードウェアはこのような固定語長を前提としている．

ハッシュ演算はプログラミング言語処理系（コンパイラ等）に最も広く応用されてきたが，近年は LEAP[11)], SETL[12)], HLISP[13)] などの非数値演算用言語においてプログラム実行時の重要基本演算となっている．

McCarthy[14)] は LISP の基本演算である cons にハッシングを適用して（これを hcons と呼ぶ，図-3 の $H_1(c, p)$ はその特別な場合に当る）各2分枝木構造（リストはその特別な場合）に対して唯一の代表を作るようにすることを提案し，これによって数式の共通部分式の同定の問題などは氷解するであろうと述べている．筆者らの HLISP には，この hcons 機能が備わっているが，$X+Y+Z=Y+Z+X=Z+Y+X$ 等々，加算（乗算も同様）の可換性に基づく任意性のために hcons だけでは不十分であって，ハッシュ法に基づく唯一代表作成[9),13)]を付加する必要がある．($\{X, Y, Z\}$, $\{Y, Z, X\}$, … は同一の集合を表わすので（+, $\{X, Y, Z\}$）の形のデータ構造を使えば加算の可換性の問題は解決する．）またハッシュ法を利用して，帰納的 (Recursive) なアルゴリズム，決定表 (Decision Table または Case 文) 等の実行を高速化することもできる[9)]．

以上のようなハッシュ法の応用では鍵をハッシュ表から削除（これはガーベッジ・コレクター，Garbege Collector, GC と略す，が行うものも含む）できなくてはならない．この削除はやっかいな問題で，Knuth[1)] の Algorithm-R はこれを行うものであるが，これは遅い上に鍵の再配置が起こる．鍵の再配置は鍵がポインターを含む複雑なデータ構造の場合には高速度で実行することは非常に難かしい．これに関して，筆者らは鍵の再配置を行わない鍵削除法を考案した．問題点は衝突のある語を削除した場合には空白語 Λ に直接戻せない点にある．そこで空白語 Λ の他に削除語 Δ (Deleted) を導入することが考えられるが，挿入と削除を反復すると Δ が増加して索表能率が低下してしまう (Knuth)．筆者らは GC に Δ から Λ に戻せる語は戻すという操作を加えて，この問題を解決するとともに，McCarthy[14)] が指摘した GC に2次記憶が必要になるという問題も解決した[7)]．**表-2** には筆者らの方法を開アドレス法で負荷率を $\alpha \leq \alpha_m$ (α_m は最大許容負荷率，$\alpha > \alpha_m$ となると GC を起動）に制限した場合の索表平均回数の理論的上限値を示す[7)]．$U, I,$

表-2 鍵の削除を伴う場合の索表平均回数の理論的上限値

	U	I	S
UD*	e^{β_m}	e^{β_m}	$\frac{1}{1-\alpha_m}$
UFD**	$\frac{1}{F(\beta_m)+e^{-\beta_m}}$	$\frac{1}{F(\beta_m)+e^{-\beta_m}}+\frac{1}{1-\alpha_m}-\frac{1}{1-\alpha_m+F(\beta_m)}$	$\frac{1}{1-\alpha_m}$

* Uniform hashing with Deletion

** Uniform hashing with Furukawa's collision flag method and Deletion

Note: $\beta_m=\frac{\alpha_m}{1-\alpha_m}$, $F(x)=x\sum_{n=0}^{\infty}\frac{(-x)^n}{n!}\frac{1}{x+n+1}=x_1F_1(x+1, x+2; -x)$, where ${}_1F_1$ is the confluent hyper-geometric function.

S は負荷履歴関数 $\alpha(t)$ の汎関数となり，α の関数にはならないが，表-2 の値を越えることはない．

なお図-2 の方法ではガーベッジ回収を行うごとに記憶領域の縮約 (Compactification, 鍵を動かしてすきまを詰めること) を行う必要があり，その高速化は難かしい．

3. ハッシング・プロセッサ[15]

開アドレス・ハッシュ探索は次のような基本操作の繰り返しである．即ち，ある鍵 k が与えられたとき，ハッシュ関数列を $h_1(k), h_2(k), \cdots$ とすれば

(a) ハッシュ・アドレス $h_j(k)$ の計算，
(b) アドレス $h_j(k)$ から鍵 k_j の読み出し，
(c) 鍵 k と k_j との比較，
(d) 比較によって生成される条件に従い，次にとるべき判断の決定．

ハードウェアでは，ステップ(a),(c)および(d)は高速に実行されるため，ハッシュ探索の速度を決定する要因は(b)のハッシュ表からの鍵の読み出しにある．(a)～(d)をマイクロプログラムを含めて，ハードウェアで実行することは一つの安易なハッシングのハードウェア化ではあるが，これでは(a)～(d)の繰り返し回数，P (Probe number) は減少しない．後述するように鍵の削除を伴った場合のハッシュ探索では，記憶利用率 $\sigma=0.9$ としたとき，最悪の場合には P は 2000 にもなる．

ここで並列処理を導入すれば，P を大幅に減少させることができる．即ち，ステップ(b)で，一度に複数個 (J 個) の鍵をハッシュ表から読み出し，ステップ(c)での鍵の比較も一度に行う．

したがって，ハッシュ・プロセッサは J 個の記憶バンクモジュール (各々 M 語の記憶容量を持つ)，ハッシュ・アドレス生成器，およびハッシュ演算制御ユニットから構成される．**図-4** はハッシュ・プロセッサの構成を示したものである．ここで，ハッシュ表は2次元配列 $K[1:M, 1:J]$ で表わされ，列に対して J 個の鍵が同時に読み出される．ハッシュ・アドレス生成器で計算されたハッシュ・アドレスは，アドレスバスを介して，各記憶バンクに送られる．読み出された鍵 $k_j(1\leq j\leq J)$ と k との比較のため比較器 CP (Comparator : $k=k_j$ ならば $m_j=1$ しからざれば $m_j=0$) が各記憶バンク内に付加されている．図-4 で ED (Empty word Detector) は空白語 A の検出器 (A ならば $e_j=1$ しからざれば $e_j=0$)，T/C は鍵の衝突の

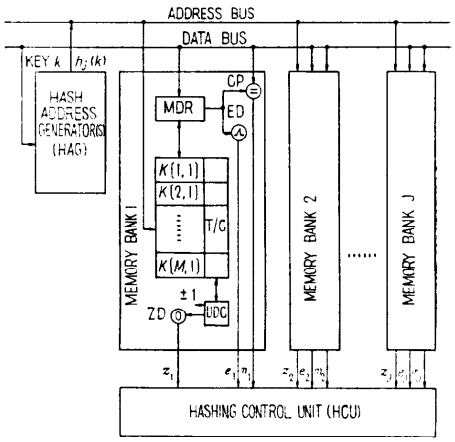

図-4 ハッシング・プロセッサ構成図

有無を記録する衝突ビット／計数器 (TあるいはCの選択については後述)，ZD (Zero Detector) は衝突検出器 (鍵が衝突状態にあれば，$Z_j=1$ しからざれば $Z_j=0$)，および UDC (Up Down Counter アップダウン計数器) が上記Cを選択した場合に必要とされる．ハッシュ演算制御ユニットはステップ(d)における判断機構を組み込んだもので，用いられるアルゴリズムに対応する構造をもつ．

3.1 並列ハッシュ・アルゴリズム

並列ハッシュ・アルゴリズムは以上に述べたハッシュ・プロセッサにインプリメントされるものである．それらはステップ(b)および(c)での並列処理に基づくものであるが，ハッシュ・アドレス生成および鍵挿入の際の手続の違いによって，以下に述べる3通りのアルゴリズムが考えられる．さらに，各アルゴリズムに対して，'S': 鍵の探索，'D': 既に表にある鍵の削除，'I': 表にない鍵の表への登録，の基礎的副アルゴリズムがある．複合副アルゴリズム，例えば，「与えられた鍵が表になければ登録」あるいは「与えられた鍵が表にあれば削除」等，についてはここでは論じない．複合副アルゴリズムは基礎的副アルゴリズムが別個生成するハッシュ関数列の一部あるいは全部を共有することができるため，副アルゴリズムを別々に実行する以上の能率を有する．

なお，以下のアルゴリズムの記述については衝突計数器Cを用いる．

3.2 アルゴリズム-1

一度の記憶参照に対して，1つのハッシュ関数列 $h_i(k), i=1,2,\cdots$ が J 個のバンクに対して共通に用いられる．したがって1回のハッシュ探索に対して，

$K[h_i(k),1]$, $K[h_i(k),2]$, …, $K[h_i(k),J]$ が同時に読み出される．アルゴリズム-1 は，並列処理であるという点を除いて，本質的にはバケット数 J の開アドレス法[1]と同一である．以下のアルゴリズム-1I, -1S および -1D では各バケット（つまり，$K[i,j]$, $j=1,2,\cdots,J$）につけられた衝突計数器 $C[1:M]$ が用いられる．

アルゴリズム-1I: 本アルゴリズムはハッシュ表にない新らしい鍵 k を登録し，変数 R に鍵 k が格納されるセルのアドレスをセットする．

(I1.1)[初期化]　$i\leftarrow 1$.

(I1.2)[ハッシュアドレス生成]　$h\leftarrow h_i(k)$.

(I1.3)[並列記憶アクセス]

　$K[h,1],\cdots,K[h,J]$ を同時に読み出す．

(I1.4)[空白語のチェック]

　$\exists j, e_j=1$ ならば $K[h,j]\leftarrow k$,

　　$R\leftarrow K[h,j]$ のアドレス，

　　アルゴリズム終了．

　しからざれば，$C[h]\leftarrow C[h]+1$

　　$i\leftarrow i+1$, (I1.2) に行く．　⊠

アルゴリズム-1S: 本アルゴリズムは与えられた鍵 k をハッシュ表から探索し，鍵 k が格納されているセルのアドレスを変数 R にセットする．もし鍵 k が表中に存在しなければ R の値は Λ である．

(S1.1)[初期化]
(S1.2)[ハッシュアドレス生成]
(S1.3)[並列記憶アクセス]　} アルゴリズム-1I と共通．

(S1.4)[一致のチェック]

　$\exists j, m_j=1$ ならば，$R\leftarrow K[h,j]$ のアドレス

　　アルゴリズム終了．

　$Z=1$（つまり $C[h]>0$）ならば，

　　$i\leftarrow i+1$, (S1.2) へ行く．

　しからざれば，$R\leftarrow\Lambda$, アルゴリズム終了．⊠

アルゴリズム-1D: 本アルゴリズムは既にハッシュ表に存在する鍵を削除する．

(D1.1)[初期化]
(D1.2)[ハッシュアドレス生成]
(D1.3)[並列記憶アクセス]　} アルゴリズム-1I と共通．

(D1.4)[一致のチェック]

　$\exists j, m_j=1$ ならば，$K[h,j]\leftarrow\Lambda$,

　　アルゴリズム終了．

　しからざれば，$C[h]\leftarrow C[h]-1$,

　　$i\leftarrow i+1$, (D1,2) へ行く．　⊠

3.3 アルゴリズム-2

互に独立な J 個のハッシュ関数列 $h_i^{(j)}$, $i=1,2,\cdots$ が各記憶バンク（$1\leq j\leq J$）について用いられる．したがって，1回のハッシュ探索に対して，$K[h_i^{(j_1)}(k),1]$, $K[h_i^{(j_2)}(k),2]$, …, $K[h_i^{(j_J)}(k),J]$ が同時に読み出され，この順序で挿入時に空セルが探される．ここで，$(j_1,j_2,\cdots,j_J)$ は鍵 k および i に依存して決定される $(1,2,\cdots,J)$ の順列である．

これから述べるアルゴリズム-2I, -2S, -2D, -3I, -3S および -3D では衝突計数器 $C[1:M,1:J]$ が用いられる．

アルゴリズム-2I: アルゴリズム-1I と同機能．

(I2.1)[初期化]　$i\leftarrow 1$.

(I2.2)[ハッシュアドレス生成]

　$h[n]\leftarrow h_i^{(n)}(k)$, $n=1,2,\cdots,J$ を同時に行う．

(I2.3)[並列記憶アクセス]

　$K[h[1],1],\cdots,K[h[J],J]$ を同時に読み出す．

(I2.4)[空白語のチェック]

　以下のことを $n=j_1,j_2,\cdots,j_n$ の順で行う；

　　$e_n=1$ ならば，$K[h[n],n]\leftarrow k$,

　　　$R\leftarrow K[h[n],n]$ のアドレス，

　　　アルゴリズム終了．

　しからざれば，$C[h[n],n]\leftarrow C[h[n]n]+1$.

　$i\leftarrow i+1$.　(I2.2) へ行く．　⊠

アルゴリズム-2S: アルゴリズム-S と同機能．

(S2.1)[初期化]
(S2.2)[ハッシュアドレス生成]
(S2.3)[並列記憶アクセス]　} アルゴリズム-2I と共通．

(S2.4)[一致のチェック]

　$\exists j, m_j=1$ ならば，$R\leftarrow K[h[j],j]$ のアドレス

　　アルゴリズム終了．

　もし $Z_{j_1}\wedge\cdots\wedge Z_{j_J}=1$ ならば，$i\leftarrow i+1$,

　(S2.2) へ移る．

　しからざれば，$R\leftarrow\Lambda$, アルゴリズム終了．⊠

アルゴリズム-2D: アルゴリズム-1D と同機能．

(D2.1)[初期化]
(D2.2)[ハッシュアドレス生成]
(D2.3)[並列記憶アクセス]　} アルゴリズム-2I と共通．

(D2.4)[一致のチェック]

　以下のことを $n=j_1,j_2,\cdots j_J$ の順で行う；

　　$m_n=1$ ならば，$K[h[n],n]\leftarrow\Lambda$.

　　アルゴリズム終了，

しからざれば，$C[h[n],n]\leftarrow C[h[n],n]-1$.

$i\leftarrow i+1$, (D2.2) へ行く． ☒

3.4 アルゴリズム-3

アルゴリズム-2 と同様に J この独立なハッシュ関数列が鍵の読み出しに用いられる．1回のハッシュ探索で，$K[h_i^{(1)}(k),1],\cdots,K[h_i^{(J)}(k),J]$ が読み出され，鍵の挿入に当っては，この順で空セルが探される．アルゴリズム-3I,-3S および -3D はしたがって，各々 (I3.4), (S3.4), (D3.4) で鍵 k および i にかかわらず $n=1,2,\cdots,J$ の順で処理が行われる以外は，アルゴリズム-2 の対応する副アルゴリズムと同じである．

3.5 ハッシュ・プロセッサの性能予測

既に述べたように，ハッシングの効率は，ハッシュ探索回数 P で決まる．特に，成功探索回数 PS および不成功探索回数 PU が基本的な効率評価のパラメーターである．PU,PS は α_m の近傍で鍵の削除，挿入を交互に繰り返した時に最大値 $\underline{PS},\underline{PU}$ (つまり最悪状態) になることが理論的に，またはシミュレーションから証明されており[7]，また具体的な数値もハッシュ関数列および鍵削除の無作為性を仮定したときに得ることができる[16]．この条件下では，$\underline{PS}=$'I' の最大探索平均回数＝'D' の最大探索平均回数である．**表-3.4** に $\underline{PS},\underline{PU}$ の値を示す．

この結果から，アルゴリズム-3 が総じて効率が良いことがわかる．アルゴリズム-2 はバンク数が大きくなるにつれ，また α_m が1に近づくにつれ，アルゴリズム-1 より効率が悪くなる．**表-5** はこの3つのアルゴリズムをハッシュ・プロセッサにインプリメントするときに通常の計算機のマルチバンクをもつ記憶システムに付加すべきハードウェアの規模をまとめたものである．GC を用いた場合，衝突数の計数は GC によってなされるので前述 T/C の T (衝突状態を示すタグビット) のみでよい．アルゴリズム-1 はアルゴリズム-3 より効率は劣るが実現が簡単であるという利点がある．

表-3 PU: ハッシュ・プロセッサによる不成功索表平均回数の上限値

		$\alpha_m=0.6$	0.7	0.8	0.9	0.95
$J=1$	PU_1	2.27	4.43	1.88×10	1.96×10^3	3.07×10^7
$J=2$	PU_1	1.98	3.66	1.42×10	1.22×10^3	1.48×10^7
	PU_2	1.46	2.50	9.67	9.81×10^2	1.53×10^7
	PU_3	1.45	2.48	9.45	9.27×10^2	1.41×10^7
$J=8$	PU_1	1.35	2.06	5.76	2.16×10^2	7.68×10^5
	PU_2	1.01	1.15	2.83	2.46×10^2	3.83×10^6
	PU_3	1.00	1.09	2.12	1.01×10^2	1.02×10^6
$J=64$	PU_1	1.00	1.03	1.37	7.43	6.35×10^3
	PU_2	1.00	1.00	1.03	3.11	4.79×10^5
	PU_3	1.00	1.00	1.00	1.00	6.57

(注) PU の添字はアルゴリズムの番号を示す．

表-4 PS: ハッシュ・プロセッサによる成功索表平均回数の上限値

		$\alpha_m=0.6$	0.7	0.8	0.9	0.95
$J=1$	PS_1	2.50	3.33	5.00	1.00×10	2.00×10
$J=2$	PS_1	1.67	2.09	2.94	5.46	1.05×10
	PS_2	1.56	1.96	2.78	5.26	1.03×10
	PS_3	1.56	1.96	2.78	5.26	1.03×10
$J=8$	PS_1	1.09	1.19	1.41	2.06	3.34
	PS_2	1.02	1.06	1.20	1.76	2.97
	PS_3	1.01	1.05	1.19	1.75	2.97
$J=64$	PS_1	1.00	1.00	1.01	1.09	1.26
	PS_2	1.00	1.00	1.00	1.03	1.04
	PS_3	1.00	1.00	1.00	1.03	1.04

(注) PS の添字はアルゴリズムの番号を示す．

表-5 通常のマルチバンクメモリーに付加すべきハードウェア

	アルゴリズム-1	アルゴリズム-2	アルゴリズム-3
ハッシュ番地生成機構	1	J	J
C/T の数 C/T のビット幅	M $\log_2 M$/1 bits	MJ $\log_2 M$/1 bits	MJ $\log_2 M$/1 bits
比 較 器	J	J	J
アドレスパス	0	$J-1$	$J-1$
ハッシュ演算制御ユニット	最も簡単	最も複雑 (順列生成機構が必要)	中 間

このように，ハッシュ・プロセッサは現行の記憶システムの構造を大きく変更することなく，比較的少ないハードウェアで実現できることがわかる．またハッシングを使わない記憶の順次検索に利用することが可能で，J 回の比較を一度に実行できるので，J 倍の速度の向上がえられる．

4. 結 論

前章に示したような記憶バンクの並列処理を行うハードウェアでは連鎖法は並列アクセスが不可能であり，かつ余分の記憶場所を消費するので，考慮に値しないであろう．前述のハードウェア ($J=64$) を使えば記憶利用率 σ を 0.8〜0.9 にとって，ハッシングはほぼ1回の記憶アクセス時間，すなわち，1間接アドレス時間に相当する時間で完了する．この事実に基づく新しいソフトウェア構成法が生れるものと期待される．本稿で論じたハッシュ法は鍵とデータが全て高速記憶にはいる場合 (小データベース) であり，プログラミング言語処理と，数式処理を始めとするいわゆる人工知能の分野の非数値演算に適していると思われ

る．大容量の二次記憶を必要とする大規模データベースへのハッシング・プロセッサの応用については今後研究すべき点が多いが，ディスクのヘッドの並列性を積極的に利用することなども考慮すべきであろう．

参考文献

1) D. E. Knuth : Sorting and Searching, The Art of Computer Programming Vol. 3. Addison-Wesley, (1973).
2) G. D. Knott : Hashing Functions, The Computor Journal, 18, pp. 265～278 (1975).
3) W. D. Maurer & T. G. Lewis : Hash Table Methods, Computing Surveys. 7, pp. 5～19 (1975).
4) 古川康一：コンフリクトフラッグをもったハッシュ記憶法，情報処理 13, pp. 533～539 (1972),
5) 西原清一，萩原宏：分割Residue Hash表とその連想的検索法，情報処理 14, pp. 546～549 (1973).
6) R. Morris : Scatter Storage Techniques CACM, Vol. 11, pp. 38～44 (1968).
7) T. Gunji and E. Goto : A Comparison of Hashing Algorithms with Key Deletion, ISTR-76-15 (Information Science Department, University of Tokyo, Technical Report).
8) D. Gries : Compiler Construction for Digital Computers, John Wiley & Sons, New York, N. Y., (1971).
9) E. Goto and Y. Kanada : Hashing Lemmas on Time Complexities with Applications to Formula Manipulation, ACM-SYMSAC 76, Yorktown Heights. N. Y. (1976).
10) E. Goto, T. Ida and T. Gunji : Parallel Hashing Algorithms (to be published in Information Processing Letters).
11) J. Feldman & P. Rovner : An Algol-Based Associative Language, CACM Vol. 12, 439～449 (1969).
12) J. Schwartz : On programming, an Interim Report of the SETL Project, Installment II : The SETL language and examples of its use, Courant Institute of Mathematical Science, New York University, New York, N. Y.
13) M. Sassa and E. Goto : A Hashing Method for Fast Set Operatioons, Information Processing Letters, Vol. 5, 31～34.
14) J. McCarthy in D. Bobrow (ed.) : Symbol Manipulation Languages and Techniques, North Holland, Amsterdam, 151～152 (1971).
15) T. Ida & E. Goto : Performance of a Parallel Hashing Hardware with Key Deletion, (To appear in the proceeding of IFIP Congress, 77 Toronto).
16) T. Ida & E. Goto : Analysis of Parallel Hashing Algorithms with Key Deletion, (Submitted for publication to the Journel of Information Processing Society of Japan).

(昭和51年12月9日受付)