Library and Information Science No. 7 1969

電子計算機による文字列の処理
——基本的技法と応用——
Computer Techniques for Processing Strings of Characters

安西郁夫
Ikuo Anzai

Résumé

In this article the writer describes fundamental techniques for processing strings of characters on a computer taking the case of TOSBAC-3400 System.

As applications the writer discusses the organization of and operations on a dictionary. The focus is on the automatic compilation of and search operations on a stop-word dictionary with a list structure. In comparison with serial and binary searches the writer considers the economy and efficiency of a list-structure dictionary, discussing potential applications of list processing to the field of information storage and retrieval.

(School of Library and Information Science)

I. 序

電子計算機による情報検索の基礎ともいうべきものに文字列 (string) の処理がある。本稿においては，文字列処理の問題点と基本的技法ならびに応用例を具体的に解説する。

Digital computer (計数型計算機) には固定語長方式のものと可変語長方式のものとがある。後者に属する計算機では，1桁単位で処理が行なわれるので，一般に character machine または byte machine と呼ばれる。一方前者に属する計算機では，一定の桁数をまとめたもの (これを word [語] と呼ぶ) が処理の単位となるので，word machine と呼ばれているが，筆者が利用している慶応義塾大学工学部中央試験所計算センターの計算機

(TOSBAC-3400) は word machine であるので，本稿では固定語長方式の計算機による処理のみに対象を限定する。

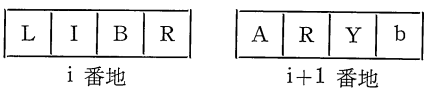
同じ固定語長方式であっても，1ワードに収容される文字数や，1文字を構成するビット数は機種によって異るので，本稿では，便宜上，1ワード4文字，1文字6ビットの方式をとる2進演算の TOSBAC-3400 に即して論を進めることにする。

なお，計算機内部の処理単位である word と，我々が計算機を用いて処理しようとする言語情報の単位である word を区別するために，本稿では前者を‘ワード’と呼び，後者を‘語’と呼ぶことにする。

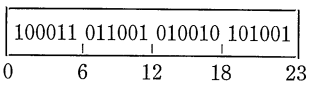
II. 文字列の大小

カードや紙テープなどの入力媒体にそれぞれのコード(外部コード)でパンチされたデータが読取装置から読み取られると，各計算機固有の内部コードに変換され，プログラムで指定された領域に転送される。データが数値である場合には，演算に備えて文字形式から2進数に変換しなければならないが，文字そのものとして処理されるデータの場合には，このような変換が不必要であることはいうまでもない。

今仮に文字列‘LIBRARY’が読みこまれたとすると，それは次のような形で記憶装置内の2ワードに貯えられる。(b はブランクを示す)



i 番地の内容をビットで表示すると次のようになる。



1ワードを構成するビットには左端から序数(0～23)が付けられており，0ビット，1ビット，2ビット，……と呼ばれているが，ビット数と紛らわしいので，本稿ではこれらを B_0～B_{23} で表現することにする。

数値が固定形で記憶される場合には，B_0 が符号ビットとなり，B_0 が 0 であればプラス，B_0 が1であればマイナスの値となる。その中味が文字であろうと，数値であろうと，24個のビットの各々が 0 または 1 であることに変りはなく，見かけ上は区別がつかない。前例の LIBR という文字列は，奇妙なことながら，同時に -7498583 という数値でもあるのである。従って，計算機内部では，文字列同志で四則演算を行なうことが可能となる。

次に例として若干の文字列とその内部コード（8進表示）を掲げてみよう。

AIDA	21312421
BABA	22212221
HARA	30215121
KANO	42214546
NOMA	45464421
SONO	62464546
WADA	66212421

AIDA (21312421) よりは BABA (22212221) の方が数値として大きいから，AIDA を被減数，BABA を減数として減算を行なうと，差はマイナスの値となる。アルファベットに対しては，8進で 21(A) から 71(Z) まで，アルファベットの順を追うに従って大きなコードが配当されているから，HARA は BABA よりも，KANO は HARA よりも，NOMA は KANO よりも値が大きく，従って，減算を利用して二つの文字列の大小を判別することを繰り返せば，我々は簡単にこれらの文字列をアルファベット順に配列することができるように思われる。

ところが，問題はそれほど簡単ではない。というのは，頭文字が A(21) から I(31) までの場合は，2進表示で B_0 が 0 となり，プラスの値となるが，J(41) 以降の場合は B_0 が 1 となり，マイナスの値となるからである。従って，BABA は AIDA より大きいが，KANO は AIDA よりも小さくなる。KANO と NOMA はともにマイナスの値であるが，NOMA の方が絶対値が大きいから，NOMA は KANO よりも小さいように思われるが，負の数を2進で表示するときは，正の数の補数を用いるから，KANO (42214546) の絶対値は 35563232，NOMA (45464421) の絶対値は 32313357 となり，従って NOMA は KANO よりも大きな値をもつことになる。

前掲の文字列を単純に正順（小→大順）で sort すると，KANO, NOMA, SONO, WADA, AIDA, BABA, HARA の順に配列される。この配列は誤まってはいるものの，誤りが乱雑ではなく，そこには規則性が見られる。すなわち，これらの文字列を頭文字によって A～I のグループと J～Z のグループに分けると，それぞれの

グループ内では正しく配列されている。従って，プラス・グループとマイナス・グループを別々に sort してから，プラス・グループの後にマイナス・グループを置けば，正しいアルファベット順が得られる筈である。

ところが，文字にはアルファベット以外にブランク，コンマ，ピリオド，コロンなどのいわゆるデリミター(delimiter) がある。例として，AN と AND という文字列を比較してみよう。(ブランクは b で表示する)

A	N	b	b

A	N	D	b

両者はメモリー内で前図のように記憶されるが，ブランクのコードは 60 であるため，AN の内部コードは 21456060, AND のそれは 21452460 となり，AND の方が AN よりも小さな値となってしまう。もしも，ブランクなどのデリミターに英数字よりも小さな値のコードが与えられていれば問題はないが，そうでない限り，デリミターの存在は文字列の処理を複雑にする。

この問題を解決する方策として，一般に固定長語計算機では，文字列を処理する際に，4字／ワードの記憶形式を1字／ワードの形式に拡散する。(図1参照)

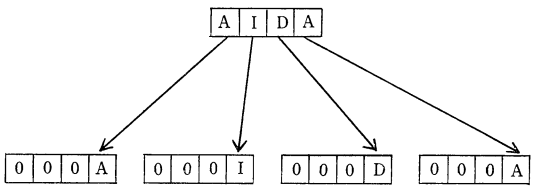


図1．文字の拡散

各文字は右詰めの形をとり，左側の18ビットには0が入る。この形をとることによって，可変長語計算機と同じように文字単位の処理が可能となるが，メモリーの使い方に無駄があることや，拡散したり，処理が済んでから元の形に集約する操作が余分に必要となる点で，固定長語計算機は不便であるといえる。

拡散する場合には，通常 U.A. (Upper A Register) と U.Q. (Upper Q Register) を結合し，処理すべきワードの中味を U. Q. に load し，U. A. を clear してから左に6桁ずらし，U.A. の中味を store するという操作を4回繰り返す。(図2参照)

処理すべきワードが偶数個であれば，2ワード分を L.A. (Lower A Register) と U.Q. を連結した48ビットに入れ，左に6桁ずつシフトする方法もある。

集約の場合は拡散のプロセスを逆にやればよい。ただし，先頭のワード(i 番地)からではなく，後尾のワード(i+3 番地) から逆順に処理する。(図3参照)

U.A.				U.Q.				
?	?	?	?	A	B	C	D	U.Q. に load
0	0	0	0	A	B	C	D	U.A. を clear
0	0	0	A	B	C	D	0	左に6桁シフトしてi番地に格納
0	0	0	0	B	C	D	0	U.A. を clear
0	0	0	B	C	D	0	0	左に6桁シフトして i+1 番地に格納
0	0	0	0	C	D	0	0	U.A. を clear
0	0	0	C	D	0	0	0	左に6桁シフトして i+2 番地に格納
0	0	0	0	D	0	0	0	U.A. を clear
0	0	0	D	0	0	0	0	左に6桁シフトして i+3 番地に格納

図2．拡散のプロセス

U.A.				U.Q.				
0	0	0	D	?	?	?	?	i+3 番地を U.A. に load
0	0	0	0	D	?	?	?	右に6桁シフト
0	0	0	C	D	?	?	?	i+2 番地を U.A. に load
0	0	0	0	C	D	?	?	右に6桁シフト
0	0	0	B	C	D	?	?	i+1 番地を U.A. に load
0	0	0	0	B	C	D	?	右に6桁シフト
0	0	0	A	B	C	D	?	i 番地を U.A. に load
0	0	0	0	A	B	C	D	右に6桁シフトしてU.Q.の内容を格納

図3．集約のプロセス

III. 辞書引き

A. 逐次探索と2分探索

入力データを辞書の最初の見出しから順を追って照合する方法を逐次探索というが，key である見出しの配列がシーケンシャルであろうと，ランダムであろうと，簡単な操作で行なえる。この場合，辞書の見出しも入力データも，1字／ワードのモードに拡散する必要はなく，1語を格納するのに用いるワード数を最長語に合わせて一定にしておけばよい。仮にそれを4ワードとすれば，1見出しとの照合に要する減算は4回である。求めるものが辞書の先頭にあるという最も運の良い場合には，照合は1回ですむが，求めるものが末尾にあるか，それとも全く無い場合には，見出しの数を N とすれば，N 回の照合を必要とする。従って，照合回数の平均は $(N+1)/2$ となる。N の値が小さい場合には，この方法も有効では

あるが，N の値が増大するにつれて効率は低減する。

2分探索 (binary search) では，入力データはまず辞書の中央位置にある見出しと照合され，入力データがその見出しより小であれば，前半の中央位置の見出しと照合され，大であれば後半の中央位置の見出しと照合される。この操作を繰り返すことによって探索の範囲を絞る。

例えば，次のような見出しによって構成されている辞書があるとする。

1. SINCE
2. SMALL
3. SMALLER
4. SMALLEST
5. SOME
6. THAN
7. THAT
8. THE
9. THEIR
10. THEM
11. THESE
12. THEY
13. THIS
14. THOSE
15. TO

入力データを SMALLER とすれば，照合は次の手順によって行なわれる。

①中央位置 8 と比較
　見出し＞入力データ
②前半の中央位置 4 と比較
　見出し＞入力データ
③第一4半分の中央位置 2 と比較　見出し＜入力データ
④3と比較
　見出し＝入力データ

見出しの数を N とすれば，最大照合回数は $1+\log_2 N$ であり，N の値が増大しても，照合回数はそれほど増加しないから，大きな辞書を引くときに有効である。

しかしながら，単に一致するかしないかを判定する逐次探索とは異なり，大小関係の判定も必要であるために，すでに述べたように，入力データと見出しの先頭ワードの符号（正，負）を比較した上で処理しなければならないし，さらにブランクの問題を解決しなければならない。

例えば，第3ステップで SMALL と比較されると，ブランクがあるために，SMALL の方が入力データより大きな値となるため，第4ステップでは SMALLER と照合されず，SINCE と照合される結果になる。このような誤りを避けるためには，見出し語に続くブランクはすべて 0 と置きかえておかねばならない。入力データについても同様のことが要求される。これには，それなりの操作が伴なうので，ブランクの内部値がアルファベットのいずれよりも小でない限りは，2分探索もあまり効率はよくない。

B. リスト処理

1. 木表現とリスト構造

2分探索で例にあげたストップ・ワード辞書を木構造で表現すると，図4のようになる。○は節 (node) と呼ばれ，節は互に枝 (branch) で結ばれている。枝で結ばれた左側の節は親，右側の節は子と呼ばれ，共通の親節から分岐した節は互に兄弟関係を持つ。(図中の点線が

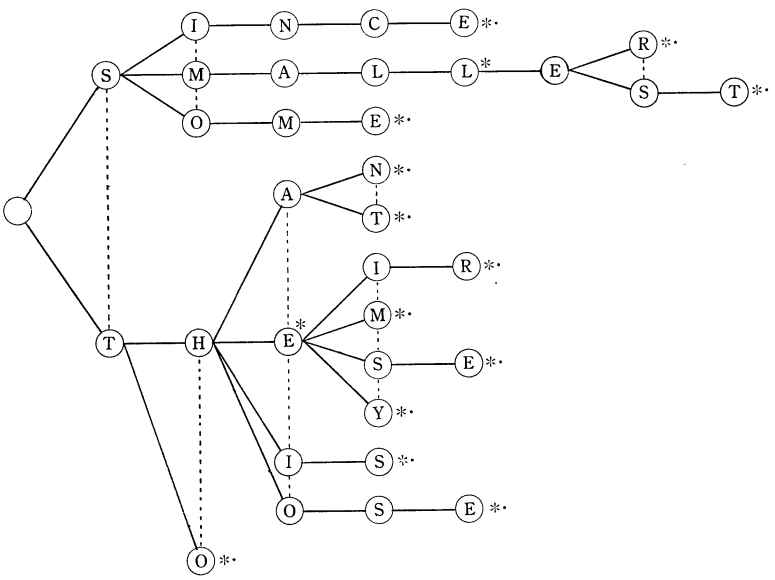
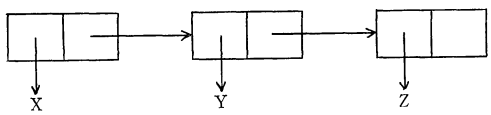


図4．ストップ・ワード辞書の木表現

兄弟関係を示す）子のみがあって親のない節は根 (root) であり，親のみがあって子のない節は葉 (leaf) と呼ばれる。節に付加された ＊印は語尾符号であり，・印は葉符号である。

このような木構造は，計算機内部ではリスト構造 (list structure) をとる。

典型的なリストの各要素は2個のポインターから成り立ち，最初のポインターは情報の所在を，2番目のポインターはリストの次の要素の所在を示す。



リストの各要素は木構造の節に当る。図4のような木構造辞書をリスト構造で編成する場合，リストの各要素は，通常 ① 節の値である文字そのもの，② 子の番地，③ 次弟の番地によって構成される。これらを格納するためには2ワードが必要である。筆者の自動索引作成システムでは，メモリーを節約するため，1要素に1ワードのみを割り当て，その中に，文字，次弟の番地，語尾符号，ならびに葉符号を格納し，子は常に1番地隣に所在しているので，子の番地は省略している。

この辞書が記憶装置内に配置された状態を示すと，図5のようになる。リストの先頭は仮に5000番地とし，番地は便宜上10進で表示してある。

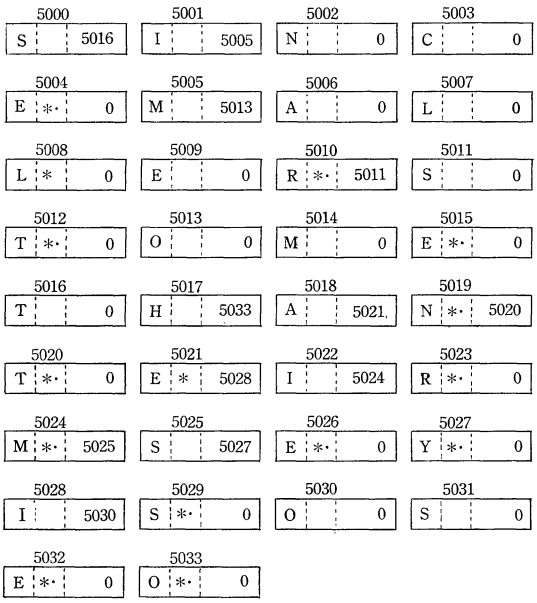


図5．辞書のリスト構造

所謂リスト処理とは，データの集合を，その物理的関係によってではなく，論理的関係によって処理する手法

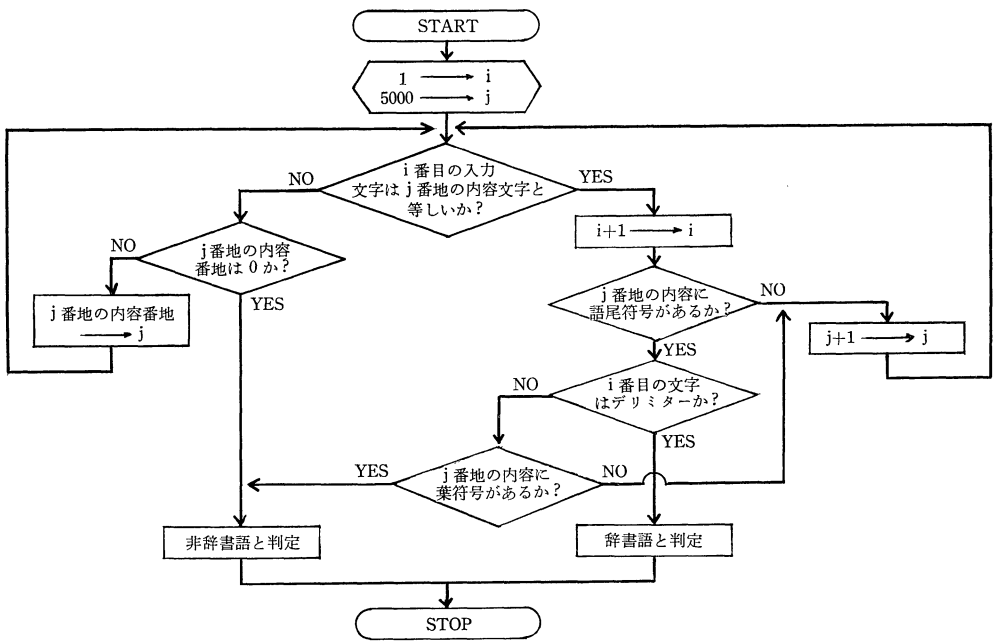


図6．リスト構造辞書引きルーチン

であり，その論理的関係を示すものが，リストの各要素に内包される子や弟の番地，すなわちポインターである。ポインターの示す番地には，絶対番地，相対番地，修飾番地の3方式がある。

メモリーを構成するワードにはそれぞれ固有の番地が与えられているが，この固有の番地をポインターとして用いるのが絶対番地方式であり，図5の例もその方式に従っている。相対番地とは，自己を0番地と見做した場合の相手の番地であり，5003番地から見た5005番地は2番地となる。換言すれば，相対番地とは，自己と相手の番地の差である。修飾番地とは，リストの先頭を0番地と見做した場合における各要素の番地である。

図5のようなリスト構造を持つ辞書の引き方をフロー・チャートで示すと，図6のようになる。仮にこれがストップ・ワードの辞書であれば，(辞書語と判定)という部分は(ストップ・ワードと判定)に，(非辞書語と判定)は（キーワードと判定）に書き換えられる。

このルーチンにはデリミターの判別が含まれているが，その判別に，文字リテラルによる減算の繰り返しを用いず，特殊な判別用ベクトルを用意すれば，この辞書引きは，アセンブラ言語で約30ステップの比較的簡単なルーチンである。

問題は辞書を記憶装置内に自動的にリスト構造で割り付ける方法であるが，これには，あらかじめ人間が図4のような木構造図を描き，ついで図5のような割付表を作成し，カードにパンチしたものを，そのままデータとして記憶装置内に読み込む方法もある。

その場合，1要素には1ワードしか割り当てないとすれば，ポインターには図5のような絶対番地は使えない。というのは，パンチカードの4欄が1ワードに相当するため，仮に4桁の絶対番地をパンチすれば，それだけで1ワードを占有してしまい，ノードの値である文字や語尾符号や葉符号を収容することができなくなるからである。仮に文字，語尾符号，葉符号にそれぞれ1欄を割り当てると，ポインターには1欄しか残されず，1桁の数字しかパンチできない。たとえ相対番地を用いるとしても，1桁の数字では実用にほど遠いから，ポインターには最低2欄を割り当てねばならず，その結果，他の3者のうち比較的生起頻度の低い葉符号をはみ出させ，別に1ワードをそれに割り当てざるをえなくなる。

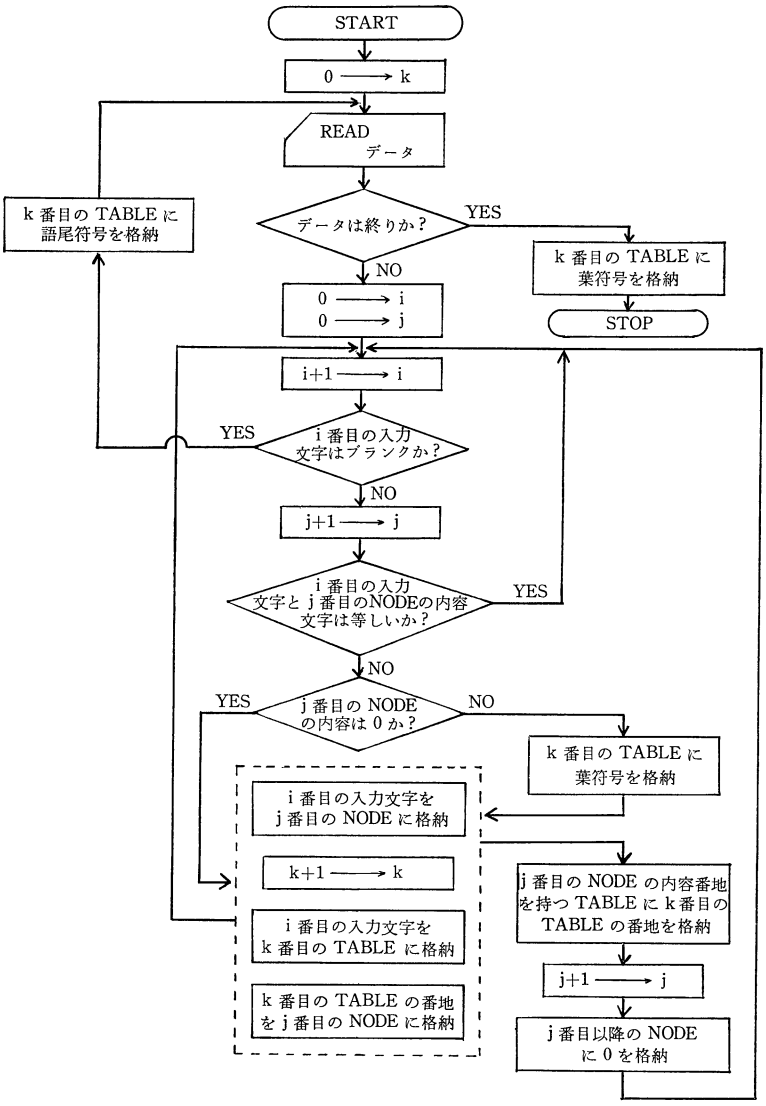


図7．リスト構造割り付けルーチン

葉符号は一見不要なもののように思われるが，これがないと，次のような誤りを犯すことになる。

仮に AND と ANY がストップ・ワードとして辞書に登録されているとしよう。今 ANDY という入力文字列があるとすれば，この文字列は登録されていないにも関わらず，ストップ・ワードとして認定されてしまうことになる。

いずれにせよ，小規模の辞書であれば，人間が割り付けることも可能ではあるが，その作業はかなり面倒であり，しかも間違いが起きやすく，規模が大きくなれば，実際には不可能となるから，プログラムによって自動的に構成することが必要となる。

図7は自動割り付けルーチンのフローを示したものである。

筆者の方式では，1ワードのビットの配分は次のようになっている。

①文字……B_0～B_5	6 ビット
②語尾符号……B_6	1 ビット
③葉符号……B_8	1 ビット
④ポインター……B_{10}～B_{23}	14 ビット

② と ③ を文字形式で格納すると各6ビットを必要とし，ポインターにはわずか6ビットしか残されないが，ビット処理により，②があるときは B_6 を ON にし，③があるときは B_8 を ON にすることにすれば，ポインターには，すべての絶対番地を表現するのに必要な14ビットを確保でき，加えて2ビットの余裕さえ生じる。

2. リストの変更

辞書を構成する語はデータとして読み込まれた上でリストに構成されるが，辞書の規模が大きくなり，ジョブを処理するたびに読み込むことが非能率となる場合には，外部記憶装置内に辞書を常駐させることになる。その場合に問題となるのは，辞書を構成する語の追加や削除などの変更作業であろう。

前掲のストップ・ワード辞書を例にとって変更の手続を説明してみよう。

たとえば，THROUGH という語を追加すると仮定する。この語を入力文字列として辞書を引くと，3番目の文字 R の照合時に 5030 番地で非辞書語と判定される。その R 以下の文字を，空番地（5034番地）以降に順次格納し，5030 番地のポインター部に 5034 という番地を格納すればよい。

前例のような場合には問題はあまりないが，たとえば THEIRS という語を追加する場合には，すでに THEIR が登録されているので，S を R の子として追加しなければならないが，筆者の方式では，子は常に1番地隣にあるものとし，子の番地をポインターとすることは省略しているので，一般にこのような方式では子の追加は不可能であるとされているが，leaf での追加が必要となる前例の場合には，次のような変更手続をとればよい。

①5023 番地の葉符号を消去し，文字 R と語尾符号を空番地である 5034 番地に移す。

②5023番地の文字部に特殊記号ⓐ（他の特殊記号でもよい）を入れ，ポインター部に 5034 という番地を入れる。

③文字 S，語尾符号ならびに葉符号を 5035 番地に格納する。

以上で変更作業を終るが，このプロセスでは，ⓐという特殊記号を格納した疑似ノードを作り出すことによって，子の追加という問題を弟を追加する問題にすりかえてしまうのである。

次に削除の例を考えよう。仮に SMALL を削除するとして，辞書を引くと，5008 番地の L で照合が終る。5008番地には葉符号がないから，語尾符号を消去するだけでよい。SMALLER を削除する場合には，照合が終る 5010 番地に葉符号があるから，この場合には文字 R を記号ⓐで置き換えて疑似ノードを作ればよい。このように，削除は追加に比較してその手続が簡単である。

3. リスト構造の経済性

リスト構造で作らた辞書の照合回数について，高橋[1]は次のような計算を示している。

s； 一つの節から出ている枝の平均数
h； 葉の高さの平均
N； 葉の数
t； 全照合回数

$$t=\frac{1}{2}(s+1)\cdot h=\frac{1}{2}(s+1)\log_s N$$

前掲式から高橋は，s を最適にすれば，照合回数は2分探索法より 24% 増加するだけであると結論している。

辞書に登録する語は，アルファベット順に配列して読み込ませるのが普通であるが，配列は必ずしもアルファベット順である必要はなく，共通の親を持つ子がグループとなっていればよい。筆者のシステムでは，頭文字はアルファベット順ではなく，T, A, O, W, I, F, B, S,

H, U, N, C, L, D, M, E, Y, V の順になっている。これは，英語の頭文字の生起頻度の高い順に配列すれば，照合回数は全体として減少するという考えに基いている。この配列順は Luhn[2] の発表したデータを参考にして定められたものである。このような配列を工夫すれば，リスト構造はきわめて効率のよいものとなる筈である。

経済性を考慮する際に重要な他の因子は，メモリー内で占有するスペースである。単一のポインターのみを内蔵する変形リストが2個のポインターを貯える通常のリストよりも経済的であることはいうまでもないが，固定長のテーブル形式に比較してもはるかに小さなスペースで事が足りる。筆者が実験的に使用している117語の辞書に必要なスペースは296ワードであり，1語に必要なスペースの平均は約2.5ワードにすぎない。固定長のテーブル方式では，1語に割り当てるワード数は，最長語に合わせるため，通常5ワードであるが，仮に4ワードに押えても，はるかに大きなスペースを必要とするといえよう。

リスト処理の手法は，各種ファイルの構成や検索式の処理など，IR の分野においても多くの応用が可能と思われる。三輪[3] はその修士論文においていくつかの試みを論じているが，今後もさまざまな応用が考究され，実用に供されることが望まれる。

（図書館・情報学科）

1) 高橋達郎．情報検索．東洋経済新報社，1968. p. 75–6.
2) Luhn, H. P. Potentialities of auto-encoding of scientific literature. ＜Schultz, C. K., ed. *H. P. Luhn: Pioneer of information science selected works*. New York, Spartan Books, 1968＞ p. 224.
3) 三輪行雄．情報検索におけるリスト処理．慶応義塾大学修士論文，1968. 74 l.

参考文献

Foster, J. M. *List processing*. New York, Elsevier Pub. Co., 1969. 54 p.

原田賢一．KOS マクロアセンブラ (KMAP) 説明書．慶応義塾大学工学部中央試験所計算センター，1968. 88 p.

橋本昌幸，笹森勝之助．“ドキュメンテーションにおける辞書の問題 ——電子計算機処理を中心として——，” **情報管理**，vol. 11, 1968. 5, p. 66–71；vol. 11, 1968. 6, p. 120–7.

Hopgood, F.R.A. *Compiling techniques*. New York, Elsevier Pub. Co., 1969. 126 p.

慶応義塾大学工学部中央試験所計算センター．アセンブラ言語（基礎編）．同センター，1969. 83 p.

近藤頌子．TOSBAC-3400 電子計算組織命令説明書．慶応義塾大学工学部中央試験所計算センター，1968. 76 p.

西村恕彦．木表現とリスト処理の算法．＜第7回プログラミング・シンポジウム報告集，1966＞ 28 p.