計量国語学第十七巻第八号〔Mathematical Linguistics, vol. 17 no. 8〕1991年

解説

大規模データ上での効率的な検索手法について
——ハッシュ，トライ

伊藤　克亘　（東京工業大学院生）
奥村　学　（東京工業大学）
田中　穂積　（東京工業大学）

ディスクリプタ：　電子化辞書　コーパス　データ検索　ハッシュ　トライ構造
ソート　表記のゆれ　LR構文解析法　動的ハッシュ法

1．はじめに

近年，大規模なテキストデータを利用した自然言語処理研究が各方面で盛んに行なわれている．この理由としては，記憶装置が大容量・高速・安価になってくるなど計算機環境が充実してきていること，比較的広い範囲の実データをカバーできるようなシステムが必要であると研究者自身が感じていることがあげられる．従来人間が引いて使うものであった，国語辞典・英和辞典などの辞書を計算機上に実現した電子化辞書[1,3]や，新聞などから用例を集めたコーパスを用いた研究はこの範疇に入る．

このように大規模なテキストデータを計算機上で扱う際には，蓄積してあるデータから必要なデータをどのように効率的に検索するかが問題になる．データを配列やリストの形式で蓄積しておくと，データの検索は線形探索となり，データの数に比例した時間を要してしまう．これに対し，検索のキーとなる文字列と格納場所を対応づけるインデックスをあらかじめ用意しておきそれを検索に利用する方法が考えられる．このインデックスを構成する方法として，ハッシュ表を用いるものと，トライを用いるものがある．

ハッシュ表を用いた検索では，検索用のキーを与えると検索するデータの格納場所を計算するような関数（ハッシュ関数）を用意しておく．ハッシュ法は高速ではあるが，機能的に不十分な点がある．すなわち，検索用のキーと全く同じものを与えないと検索が行なえない点である．たとえば，「セレナーデ」が辞書に登録されている場合に，「セレナード」をキーとして検索しようとしても，ハッシュ表には存在しないため検索できない．さらに，追加・削除などのデータの更新時における柔軟性にも問題がある．伝統的な（静的）ハッシュ法では，ハッシュ表に登録されているデータが増えるにつれて検索・登録の効率が悪くなる．このためデータが想定していた容量を超えて増加した場合には，データ全体を再構成し直す必要があり，非常に大きなコストを必要とする．

これらの問題に対して，トライは一つの解決を与える．トライを使って構成した辞書では，キーと完全に一致する見出しがない場合に，それに近い見出しを検索することがある

ITOU Katunobu, OKUMURA Manabu (Tokyo Institute of Technology), et TANAKA Hozumi (Tokyo Institute of Technology)——On Efficient Retrieval Methods from Large-Scale Databases : Hash and Trie

程度可能になる．また，トライは動的ハッシュ法の基礎となるデータ構造である．動的ハッシュ法は，トライを用いてハッシュ表を増進的・局所的に更新することで，データを追加するときの柔軟性をあたえる手法である．

本稿では，2でハッシュ法と，ハッシュ法を辞書に応用する方法をのべる．オープンハッシュ法とクローズドハッシュ法について説明し，ふたつの方法を比較する．3では，トライについて説明し，トライを用いて実現した辞書とハッシュ法を用いて実現した辞書を比較する．4では，動的ハッシュ法の基本概念を簡単に紹介する．

2．ハッシュ法

ハッシュ法は，データを効率よく登録・検索する方法としてよく知られている．この手法では，ハッシュ表とよばれる配列とハッシュ関数とよばれる関数を用いる．ハッシュ法を用いて辞書を構成した場合，登録・検索に必要な平均時間は一定だが，最悪の場合には項目数に比例した時間が必要になる．しかし，ハッシュ関数の作り方などを工夫することで，登録・検索に必要な時間が一定をこえる確率をいくらでも小さくできる．

ハッシュ法は，大量のデータをなるべくたくさんの均等な項目数の集合に分けてあつかうことで登録・検索の平均時間を小さくしている．ハッシュ関数は，データを分類するためにつかい，ハッシュ表は分類したデータを管理するためにつかう．

ハッシュ関数は，あるキーがあたえられるとハッシュ表のインデックスをひとつ決定する．ここでは，キーとして文字列をあたえる場合のハッシュ関数として，完全ではないが簡単なものを紹介する．計算機では普通，それぞれの文字に整数のコードがあたえられている．例えば，よく用いられているEUCコードを文字コードとして使うと，文字列「セレナーデ」のときは，各文字のコードの和は下の式に示したようになる．

$$42427(セ)+42476(レ)+42442(ナ)+41404(ー)+42439(デ)=211188$$

この文字コードの和を用いれば，文字列に対して一意に整数を対応させることができる．ここで，ハッシュ表の大きさをTとすると，この整数をTで割った余りは0から$T-1$の値をとるので，この値をハッシュ関数の値とする．この関数は文字列に対して一意にハッシュ表のインデックスをかえす．例えば，上の式で$T=1009$だとすると

$$211188=1009\times209+307$$

なので，この関数の値は307となる．

ハッシュ関数は，異なるキーに対して同じ値を返してしまうことがある．しかし，検索・登録の効率をよくするためには，なるべくこのような重複を起こさないように設計しなければならない．

ハッシュ法は，ハッシュ表に見出しを直接格納するかどうかでふたつの方法に大きく分けられる．ひとつは，オープンハッシュ法とよぶ方法で，見出しを直接ハッシュ表に格納しない．もうひとつは，クローズドハッシュ法とよぶ方法で，見出しを直接ハッシュ表に格納する．

2.1. オープンハッシュ法

オープンハッシュ法では，図1に示すようにハッシュ表に見出しからなるリストの先頭へのポインタを格納する．したがって，ハッシュ表の大きさは一定だが，リストの長さには制限がないので記憶容量が許す限り見出しを登録できる．

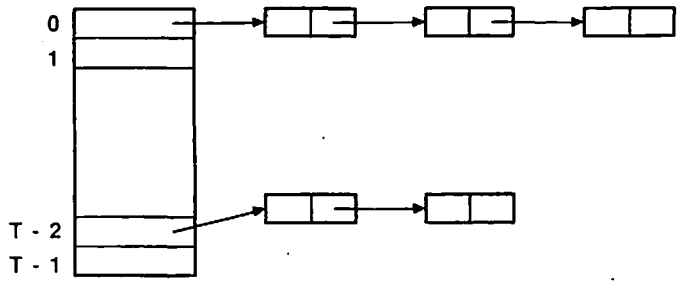

図1　オープンハッシュ法で用いるハッシュ表

登録アルゴリズム

キーに対してハッシュ関数の値を求める．その値がaだとすると，ハッシュ表のa番目を調べる．その場所にリストがつながっていない場合には，そのキーを見出しとして，その見出しひとつからなるリストをつなげる．リストがつながっている場合には，そのリストを調べてキーと同じ見出しがあれば，すでに登録されている．なければ，キーを見出しとしてリストにつなげる．

検索アルゴリズム

キーに対してハッシュ関数の値を求める．その値がaだとすると，ハッシュ表のa番目を調べる．その場所にリストがつながっていない場合と，その場所につながっているリストにキーと同じ見出しが存在しない場合は，キーが辞書に登録されていない．その場所につながっているリストにキーと同じ見出しが存在する場合には，その見出しを検索結果として返す．

このアルゴリズムから明らかなように，ハッシュ関数の重複が多ければ多いほど，リストが長くなっていく．リストが長くなると，それを線形探索するためそれだけ検索・登録の効率が悪くなる．登録する項目数をN，ハッシュ表の大きさをTとすると，リストひとつに含まれる平均の項目数はN/Tとなる．あらかじめ登録する項目数がわかる場合には，Tを項目数と同じくらいの大きさにすれば，それぞれのリストに含まれる項目数はひとつかふたつくらいになり，検索・登録に要する時は間NやTに関係なく一定時間でおさえられる．

2.2.　クローズドハッシュ法

クローズドハッシュ法では，図2に示すようにハッシュ表に直接見出しを格納する．したがって，ハッシュ表の大きさと同じ数だけの見出ししか登録できない．しかし，ハッシュ表以外のスペースはまったく使わないので，その分ハッシュ表を大きくでき，オープンハッシュ法よりあつかえる項目数が少ないわけではない．

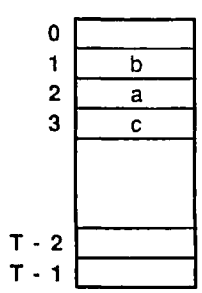

図2　クローズドハッシュ法で用いるハッシュ表

登録アルゴリズム

キーに対してハッシュ関数の値を求める．その値がaだとすると，ハッシュ表のa番目を調べる．その場所がふさがっていない場合には，その場所にキーを見出しとして格納する．ふさがっている場合（この状態を衝突とよぶ）には，違う場所を探して空いている場所を見つける必要がある．空いている場所を探すのにはいろいろな方法があるが，ここでは最も簡単な手法（最適ではない）を紹介する．

例えば，ハッシュ表のa番目がふさがっていた場合に，この方法では，$a+1$番目，$a+2$番目と順に調べて空いている場所がみつかったら，そこに登録する．ハッシュ表の最後の$T-1$番目まで調べて空いている場所がない場合には先頭に戻って，0番目，1番目と順に$a-1$番目まで調べる．そこまで調べても空いている場所がない場合は，表がいっぱいなのでその見出しは登録できない．

検索アルゴリズム

キーに対してハッシュ関数の値を求める．その値がaだとすると，ハッシュ表のa番目を調べる．その場所に登録されている見出しがキーと違うときには，$a+1$番目，$a+2$番目と順に調べて（この手順は上で紹介した空き場所の探索方法に対応するもので，空き場所の探索方法がちがう場合には，手順はかわる）空いている場所に達するか，キーと同じ見出しがあらわれるまで調べる．ハッシュ表の最後まで調べても，見つからない場合には，登録のときと同じように先頭に戻り，空いている場所に達するか$a-1$番目まで調べる．ここまで調べても同じ見出しがあらわれない場合には，キーが辞書に登録されていない．キーと同じ見出しがあらわれたら，その見出しを検索結果としてかえす．

このアルゴリズムから明らかなように，見出しを登録済みの場所が連続すると効率が悪化する．何の工夫もしない場合には，ハッシュ関数の重複以外にも項目数が増えるのにつれて，別々のかたまりだった部分があわさって大きなかたまりとなっていく．したがって，項目数が増えるとオープンハッシュ法よりも効率の悪化がひどくなる．

2.3. ふたつの手法の比較

2.2で述べたように，クローズドハッシュ法では，ハッシュ関数が同じになる見出しは連続して格納される．したがって，ある見出しを辞書から削除しても，その場所をもとの空いている状態に戻してしまうと，削除した見出しより後に連続している見出しをたどれなくなってしまう．したがって，見出しを削除しても，その場所の再利用は工夫を必要とする．しかし，オープンハッシュ法では，削除はリストからの要素の削除であるので，容易である．

このような違いはあるが，ふたつの手法は，いずれも異なるキーでハッシュ関数の値が重複すると効率が悪くなるという点で一致している．値の重複はハッシュ関数の設計を工夫することである程度は減らすこともできる．しかし，登録する項目数が多い場合や，あらかじめ登録されるキーがわかっていない場合には，重複を減らすようなハッシュ関数を設計するのは困難である．

したがって，どちらの手法もハッシュ表に登録した項目数が増えてくるにつれて，効率が悪くなる．しかし，効率が低下する度合はそれぞれの方法で異なっている．オープンハッシュ法では，ハッシュ関数の値が異なり別の集合として扱われた項目は最後まで別々に扱われる．一方，クローズドハッシュ法では，空いている場所が少なくなるにつれて，異

なるハッシュ関数の値をとる見出しの間の場所もふさがってしまって，連続した部分がどんどん大きくなり，同じ集合として扱われてしまう．（最終的には，ハッシュ表全体がひとつのかたまりとなってしまう．）

効率をよくするためには，オープンハッシュ法の場合は項目の削除でも効果はあるが，根本的にはある程度項目数が多くなって効率が悪くなってきたら，表を大きなものに作り変えるしかない．

ハッシュ表を作り変えるためには，そのときまでに登録されている項目をすべて再配置しなければならない．したがって，そのコストは非常に大きい．しかし，項目数の増加は検索・登録の効率を急速に低下させるので非常に大きなコストを必要としても作り変えた方がよい．ハッシュ表の大きさをT，登録した項目数をNとすると，表がうまっている割合はN/Tで表せる．この割合を目安として，クローズドハッシュ法の場合には，$N/T \geq 0.9$，オープンハッシュ法の場合には，$N/T \geq 2$となったときに，2倍（つまり$2T$）の大きさの新しいハッシュ表を作ればよい．[6]

2.4. ハッシュ法の問題点

ハッシュ関数は，五十音順といった本来のキーの順序と全く関係のない値をとる．例えば上で紹介したハッシュ関数で「トライ」と「トライアル」をキーとした場合の値を計算すると183と301という全く関係のない値となってしまう．したがって，キーのソートを行なうには，全項目を線形探索しなければならず，効率が悪い．また，一般的な辞書であればこのふたつの項目は，ほとんど連続した位置にくる．したがって，たとえば，「トライアル」というキーに対して「トライアル」という見出しが登録されていない場合に，「トライ」という見出しを検索結果として得るということも簡単にできるが，ハッシュ表を用いた辞書では，このような検索は不可能である．

またキーの文字列全体が得られてからハッシュ関数を計算して辞書を検索するので，入力を一文字ずつ処理しながら，形態素解析をするような応用には適していない．

3. トライ

3.1. トライ構造化辞書

トライ（trie：retrievalのまんなかの4文字をとって名付けられている）は，文字列の集合を表現する特別な構造である．まず，トライ構造を使って辞書を構成した例（トライ構造化辞書とよぶ）を図3にしめす．

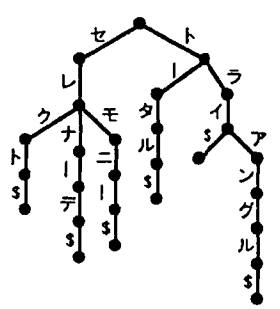

図3　トライ構造化辞書の例

図3にしめしたように，トライ構造化辞書は，n分木（有限状態オートマトン）の形に

なる．この辞書では，木の根から葉までのひとつの経路がひとつの項目に対応する．つまり，木の途中の節点は語の接頭部(prefix)を表現する．たとえば，上の辞書では，根から右にのびる枝は「ト」という接頭部を表現する．この枝につながる節点を根とする部分木を含む項目は，「トータル」「トライ」「トライアングル」とすべて「ト」という接頭部を持つ．このような形で，トライ構造化辞書は，語の集合を表現する．したがって，接頭部がたくさんの語で共有され，接頭部の種類が少ないときには，トライ構造が力を発揮する．たとよば，よみ(かな)で検索する辞書を構成するときがこのような場合にあたる．図3の$は語の終端を示す記号である．この記号を用意することで，「トライ」のように「トライアングル」という別の語の接頭部となる語も識別できる．

トライ構造化辞書では，キーの構造に基づいて集合を作るので，ハッシュ法とは違って，たとえば五十音順といったキーの順序を反映した形で項目を格納できる．したがって，キーと完全に一致する項目が存在しなくても，途中まで同じ文字列を含む見出しがあればそれを検索結果とすることができる．

また，図3から明らかなように，それぞれの枝が一文字に対応しているので，キーを一文字ずつ処理できる．したがって，辞書を引きながら形態素解析する手法に応用しやすい．

トライ構造化辞書の登録・検索アルゴリズムを次にしめす．

登録アルゴリズム

根から，キーの文字列中の文字にしたがって次々に木をたどる．もし，キーの途中で進む節点がない場合には，そのときの入力文字にあわせて新しい子の節点を作成する．$のところまでたどれる場合は，そのキーと同じ見出しがすでに登録されている．

検索アルゴリズム

根から，キーの文字列中の文字にしたがって次々に木をたどる．$までたどれれば，キーと同じ見出しが辞書に登録されているので，その見出しを検索結果としてかえす．もし，途中でたどれなくなったら，そのときの記号の前か後にくるものを検索結果としてかえす．たとえば，図3の辞書で「セレナード」がキーとしてあたえられた場合は，「セレナー」までたどれるので，「セレナーデ」を検索結果としてかえす．このようにすれば，辞書中でキーに最も近い見出しをえられる．

トライ構造化辞書では，ひとつの節点がもつ子(節点)の数は最大で，文字列を構成する記号の種類の数(に語の終端をしめす記号を加えた数)になる．しかし，実際には，(辞書の項目数にもよるが)子の数は最大の場合よりもはるかに少ない．したがって，トライ構造化辞書を実現するとき，辞書がローマ字やかなのように数が少ない記号で構成される場合には，子の節点へのポインタを格納する場所として配列を用いても無駄は少ない．しかし，漢字かなまじり表記に対応する辞書のように使われる記号の種類が多くなる場合には，リストやハッシュ表などを使って子の節点へのポインタを格納したほうがよい．

3.2. ハッシュ法を用いた辞書との比較

ハッシュ法を用いた辞書とトライ構造化辞書ではキーの扱い方が大きく異なる．ハッシュ法は，キーを構成する文字列全体をひとつのものとして見出しを検索する．一方，トライ構造化辞書は，キーを構成する文字を一文字ずつ処理して検索を行なう．このような違いはあるが，どちらもキーを構成する文字を全て処理する点では同じなので，検索効率はさほど変らない．[注]（検索効率は，トライの節点のデータ構造やハッシュ関数の設計方法

に依存するので，ここではこれ以上厳密には議論しない．）

辞書を実現するために必要な空間は，トライの場合接頭部がどれだけ共有されるかでかわってくる．しかし，トライ構造化辞書では，図3の「トライアングル」の「アングル」の部分のように，経路が一意に決定している場合でも，それぞれの文字を節点として別々にあつかわなければならない分，同じ辞書をつくる場合にトライのほうがより大きな記憶容量を必要とする．

3.1で示したアルゴリズムを用いるため，トライ構造化辞書では，新たに項目を追加する場合に，新しく分岐する節点しか変更しない．例えば，図3の辞書に「トライアル」を加えるときには，「トライア」までたどり，次に「ル」がないので，「ル」の枝を作る．このように途中の「トライ」の部分や，「セレクト」などの部分には全く影響をあたえない．したがって；登録した項目数が増えても，それほど効率は悪くならない．

また，トライ構造をソートに使うには，ひとつのキーに対して一度トライをたどればよい．したがって，効率よくソートできる．しかし，2.4で述べたようにハッシュ法ではソートの効率は配列などと同じで線形探索となり，ひとつのキーに対してつねに全項目をしらべなくてはならず，効率の差は大きい．

3.3. トライ構造化辞書の限界

キーに完全に一致する見出しが辞書中にない場合に，トライ構造化辞書では，3.1で示したように，共通している接頭部分だけトライ構造化辞書をたどってキーに近い見出しを検索結果としてかえすことができる．したがって，「セレナード」と「セレナーデ」などの語尾のゆれを考慮すれば代用できる場合には有用である．しかし，たとえば，「アーティスト」と「アーチスト」のように語中でゆれる場合には，共通部分としてたどれる部分が小さいため近い見出しとして検索することは容易ではない．また，語尾のゆれの場合でも図3の辞書を使って「トライアル」をキーとした検索結果を「トライ」としたい場合にはうまくいかない．（この場合には「トライアングル」となる）

3.4. トライ構造化辞書とLR表の関係

辞書を

見出し → セ，レ，ナ，ー，デ．

見出し → セ，レ，ク，シ，ョ，ン．

と表現すると，辞書は一文字をひとつの終端記号とした文脈自由規則の集合とみなせる．このような規則の集合から，構文解析手法としてよく知られるLR構文解析法で用いるLR表を作成すると，上に示したトライ構造化辞書と等価なものになる．

LR構文解析法は，LR表をプッシュダウンオートマトンとして用いる手法である．有限オートマトンはプッシュダウンオートマトンに含まれるので，有限オートマトンであるトライ構造化辞書は，LR表の特殊な場合であるとみなせる．したがって，文脈自由形式で表現できる構造を持つキーの場合にはLR構文解析法を使えば，トライ構造化辞書の持つ利点を持つような辞書を構成することができる．

注　ハッシュ表の規模によっては，ハッシュ関数はキーの先頭の一文字とか最後の一文字だけを使うものもあるが，ハッシュ表が大規模になると，キー全体を使うハッシュ関数でないとよいものはできない．したがって，本稿では，キー全体を使うハッシュ関数を使うことを前提に比較している．

4. 動的ハッシュ法

2.3でのべたように，ハッシュ法ではハッシュ表がうまっている割合が大きくなってくると，ハッシュ表を作りかえなければならない．しかし，通常のハッシュ法では，ハッシュ表を作りかえるときに登録されている全ての項目を再配置しなければならず，ハッシュ表の規模が大きくなればなるほど大きなコストを必要とする．

このような問題点を解消するのが，動的ハッシュ法である．動的ハッシュ法では，効率が悪くなってからハッシュ表を全体的に作り変えるのではなく，ハッシュ表の作り変えを増進的・局所的に行なう．

4.1. 基本概念

トライは3.2で述べたように登録する項目数が増えても，それほど効率が悪くならない．また，構造の変更も増進的・局所的に行なえる．動的ハッシュ法では，これらの性質に着目し，トライを用いることで，ハッシュ表の作り変えを増進的・局所的に行なえるようにしている．以下では，これまで説明したハッシュ法のことを区別のため静的ハッシュ法とよぶ．

動的ハッシュ法は簡単にいうと，見出しのハッシュ関数の値の一部でトライを構成し，その葉に静的ハッシュ法で用いるハッシュ表をつなげる．本稿では，どのようにしてトライと静的ハッシュ法を組合せるかの，基本概念だけを簡単に説明する．ここでは，説明を簡単にするために，見出しの（文字列の）一部でトライを作ることにする．この方法で図3のトライ構造化辞書とおなじ辞書を動的ハッシュ法で作る場合について説明する．

まず，見出しの先頭の一文字でトライを構成する．このときの，ハッシュ表を図4にしめす．

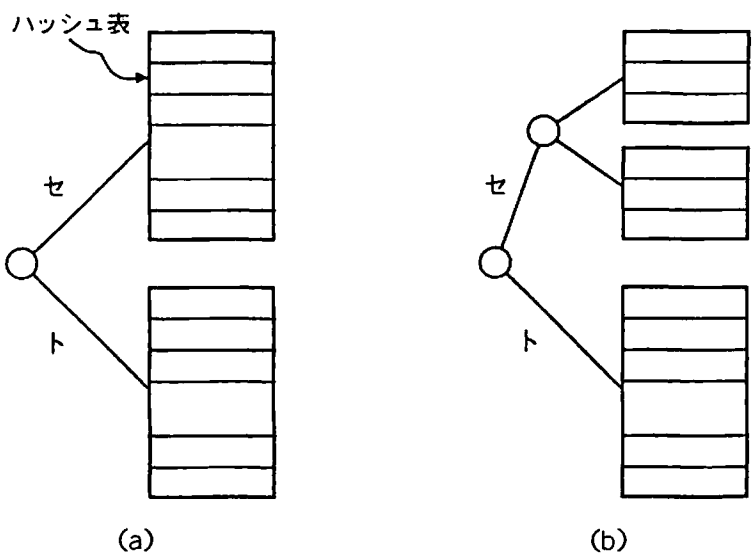

図4　動的ハッシュ法のデータ構造の概念図

このように，あらかじめトライで分割したハッシュ表を作成する．この動的ハッシュ表を使って検索・登録する場合には，まず，トライで，葉までたどり，その節点にある（静的）ハッシュ表をつかって検索する．ハッシュ関数が重複するキーの場合でも，動的ハッシュ法では，トライで別の葉にたどれれば，別の静的ハッシュ表を用いることになるので項目数が増えても効率の悪化は静的ハッシュ法よりも深刻ではない．

しかし，登録した項目数がふえてくるとトライのそれぞれの葉で，ハッシュ関数の重複が問題となってくる．このような場合はハッシュ表を作り変える必要がある．たとえば，(a)のハッシュ表で「セ」の葉でハッシュ関数の重複が問題になった（つまり「セ」の葉の部分に登録されている項目数がふえてきた）場合には(b)のように「セ」の部分だけ先頭から２文字目まで使ったトライをつくって，ハッシュ表をつくりなおす．このように，「セ」以外の部分のハッシュ表に登録されている項目には全く影響をおよぼさずにハッシュ表を更新する．

実際の動的ハッシュ法では，トライがなるべくバランスして構成されていくように，ハッシュ関数の一部を用いてトライを構成する．動的ハッシュ法の実現のためには，そのような条件をみたすハッシュ関数の設計や，ハッシュ表の更新アルゴリズムなどを理解しなければならないが，それらの点については本稿ではふれない．

5. さらに理解を深めるために

ハッシュ法・トライについては，[6]がわかりやすく，さらに詳しい説明がある．また，ハッシュ法・トライの実際のプログラムの例としては，[7]にＣ言語で書かれたものがある．動的ハッシュ法の解説としては，[8]が詳しい．

トライを自然言語処理に応用した例としては，[1]，[2]，[5]がある．[1]は，オンライン国語辞書の検索の効率化にトライを用いている．[2]は，トライを拡張して，構造を持つ句をあつかえるようにして構文解析のときに熟語処理を可能とする辞書を構成している．[5]では，トライ構造化辞書（と等価な辞書）を用いて未知語を含んだ入力を形態素・構文解析する手法について述べられている．

オートマトン全般については，[9]に詳しくのべられている．LR構文解析法を自然言語処理に用いる手法について解説した文献としては[4]があげられる．

参考文献

[1] 今津英世，田中穂積．電子国語辞典の構成と実現．計量国語学，16(5)：189-204，1988．

[2] 上脇正，田中穂積．辞書のＴＲＩＥ構造化と熟語処理．Logic Programming Conference'85，pp.329-340，ICOT，1985．

[3] 鶴丸弘昭．日常辞書の機械化とその応用．第１回「大学と科学」公開シンポジウム 日本語の特性と機械翻訳　予稿集，pp.116-129，1987．

[4] 田中穂積．自然言語解析の基礎．産業図書，1989．

[5] 堀内靖雄，伊藤克亘，田中穂積．拡張ＬＲ構文解析アルゴリズムによる未定義語を含む日本語文の構文解析．情報処理学会第40回全国大会，pp.325-326，1990．

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983. 邦訳：大野義夫訳，データ構造とアルゴリズム，培風館，1987．

[7] L Ammeraal. *Programs and Data Structure in C*. John Wiley & Sons, 1987. 邦訳：小山裕徳訳，Ｃ ―― データ構造とプログラム ――，オーム社，1990．

[8] R. J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM*

Computing Surveys, 20(2):85-113, 1988. 邦訳：遠山元道訳，動的ハッシュ法，bit別冊コンピュータ・サイエンス，共立出版，1990.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979. 邦訳：野崎昭弘他訳，オートマトン 言語理論 計算論 Ⅰ，Ⅱ，サイエンス社，1984.

(1991年2月1日受付)

受贈図書

寄贈者	図書名
中国語言研究所 (1)-(3)	方言. 1990年1号-3号.
Basil Blackwell社 (4)	*Linguistics Abstracts*. 6巻3号，1990.
ソ連科学アカデミ (5)	Вопросы Языкознания, 1990年5号.
トルヒヨ大学 (6)	*Lenguaje y Ciencias*. 29巻2号.
筑波大国語国文学会 (7)	日本語と日本文学. 13号，1990.

関係論文標題抄

李徳奉： 比喩の意味における喩辞と被喩辞の相互関係について. (7)左10-22.