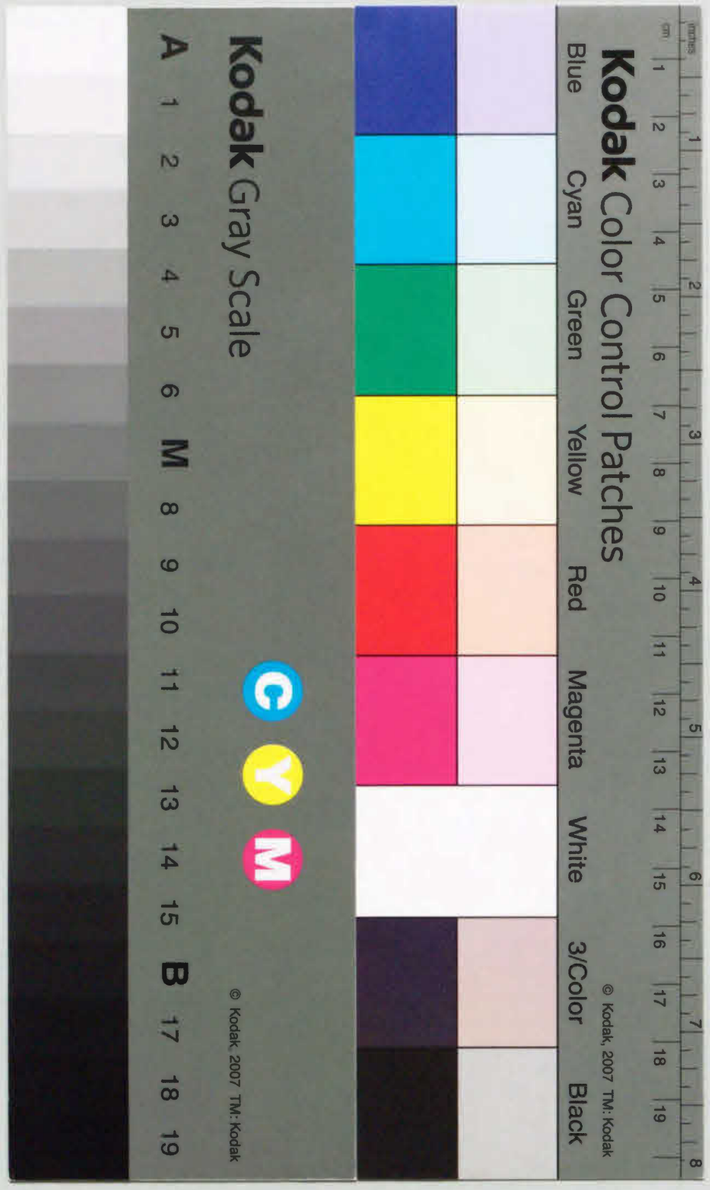


様式6

論　文　目　録

報告番号	甲　工 乙　工　第　165　号 工　修	氏名	森田和宏
学位論文題目	トライ構造を用いた共起情報の効率的記憶検索法に関する研究		

論文の目次

参考論文

主論文

1. “トライ構造を用いた共起情報の効率的検索アルゴリズム”, 森田和宏, 望月久稔, 山川善弘, 青江順一, 情報処理学会論文誌, Vol.39, No.9, pp.2563-2571 (1998).
2. “A Link Trie Structure of Storing Multiple Attribute Relationships for Natural Language Dictionaries”, Kazuhiro Morita, Masafumi Koyama, Masao Fuketa and Jun-ichi Aoe, International Journal of Computer Mathematics, Vol.72, No.4, pp.463-476 (1999).
3. “Fast Insertion Methods of a Double-Array Structure”, Kazuhiro Morita, Toru Sumitomo, Masao Fuketa and Jun-ichi Aoe, Software Practice & Experience, (条件付採録).

副論文

1. “拡張ハッシュ法における部分文字列検索の設計と実現”, 望月久稔, 森田和宏, 獅々堀正幹, 青江順一, 情報処理学会論文誌, Vol.38, No.2, pp.310-320 (1997).
2. “特徴ベクトルによる全文検索の一改善法”, 有田健, 森田和宏, 溝渕昭二, 青江順一, 情報処理学会論文誌, Vol.39, No.3, pp.826-829 (1998).
3. “A Fast and Compact Data Structure of Storing Multi-Attribute Relations among Words”, Kazuhiro Morita, Toru Sumitomo, Masao Fuketa and Jun-ichi Aoe, 1998 IEEE International Conference on Systems, Man, and Cybernetics, pp.2791-2796, San Diego, California, USA (Oct. 1998).
4. “A Fast Retrieving Algorithm of Hierarchical Relationships Using Trie Structures”, Masafumi Koyama, Kazuhiro Morita, Masao Fuketa and Jun-ichi Aoe, Information Processing & Management, Vol.34, No.6, pp.761-773 (1998).
5. “A Method of Storing Multi-Attribute Relations in Natural Language Dictionaries”, Kazuhiro Morita, Makoto Okada, Masao Fuketa and Jun-ichi Aoe, Proc. of the 18th Int’l Conf. on Computer Processing of Oriental Languages, pp.437-442, Tokushima, Japan (Mar. 1999).
6. “Efficient Controlling of Parsing-Stack Operations for LR Parsers”, Masao Fuketa, Kazuhiro Morita, Sangkon Lee and Jun-ichi Aoe, International Journal of Information Sciences, Vol.118, pp.145-157 (1999).
7. “Similarity measures using negative weight and its application to word similarity”, EL Sayed Atlam, Kazuhiro Morita, Masao Fuketa and Jun-ichi Aoe, Information Processing & Management (印刷中).
8. “複合語の分野連想語の効率的決定法”, 辻孝子, 弘田正雄, 森田和宏, 青江順一, 自然言語処理 (印刷中).
9. “キーワードの遅延抽出を考慮した文書検索構造の効率的構成法”, 岡田真, 安藤一秋, 森田和宏, 青江順一, 情報処理学会論文誌 (条件付採録).

備考

1　論文目録は，用語が英語以外の外国語のときは日本語訳をつけて，外国語，日本語の順に列記すること．
2　参考論文は，論文題目，著者名，公刊の方法及び時期を順に明記すること．
3　参考論文は，博士論文の場合に記載すること．

様式7

論 文 内 容 要 旨

報告番号	(甲　工) 乙　工　第　165 号 工　修	氏名	森 田 和 宏
学位論文題目	トライ構造を用いた共起情報の効率的記憶検索法に関する研究		

本論文は，自然言語辞書に構築される基本語彙の関係を定義することで無限に作り出される共起情報の効率的な記憶検索技法に関する研究の成果をまとめたものであり，以下の7章により構成される．

第1章では，緒論として，自然言語処理システムにおける共起情報の利用の歴史的背景を述べると共に，本研究の目的ならびにその工学上の意義を述べることで，本研究の意義及び位置付けを明確にする．

第2章では，共起情報の概要と意義について説明する．また，自然言語処理システムにおける辞書の意義についても述べ，辞書を構成する基本的なアルゴリズムについて説明する．

第3章では，辞書の構成法として最も適しているトライ構造について概要を述べ，その検索，更新アルゴリズムについて説明する．

第4章では，トライ構造を実装するためのデータ構造について説明し，最も効率的な手法であるダブル配列法について，検索，更新アルゴリズムとともに詳細に述べる．また，ダブル配列法の問題点であるキー追加の速度を改善する手法として，ダブル配列で構築された辞書を変更することなく高速化する手法と，ダブル配列に格納した情報を利用して高速化を実現する手法を提案する．

第5章では，共起情報を効率的に記憶検索する手法として，関連研究とともに提案手法であるリンクトライについて述べ，リンクトライにおいて定義されるリンク情報により，冗長性を排除した効率的な記憶が可能となることを示し，同時に提案した構造における共起情報の検索，更新アルゴリズムを説明する．また，ダブル配列を用いたリンクトライのデータ構造についても説明し，リンクトライのデータ構造を提案する上で考案された，複数の辞書を1つのダブル配列で管理する手法も述べる．

第6章では，提案手法に対して検索速度，記憶効率の理論的評価，及び実験による評価を与え，本手法の有効性を確かめ，考察を加える．

第7章では，本研究で得られた諸成果の総括を行い，今後の研究課題について述べる．

トライ構造を用いた共起情報の効率的記憶検索法に関する研究

2000年3月

森　田　和　宏

③

トライ構造を用いた共起情報の効率的記憶検索法に関する研究

２０００年3月

森田和宏

内容梗概

本論文は，自然言語辞書に構築される基本語彙の関係を定義することで無限に作り出される共起情報の効率的な記憶検索技法に関する研究の成果をまとめたものであり，以下の 7 章により構成される．

第 1 章では，緒論として，自然言語処理システムにおける共起情報の利用の歴史的背景を述べると共に，本研究の目的ならびにその工学上の意義を述べることで，本研究の意義及び位置付けを明確にする．

第 2 章では，共起情報の概要と意義について説明する．また，自然言語処理システムにおける辞書の意義についても述べ，辞書を構成する基本的なアルゴリズムについて説明する．

第 3 章では，辞書の構成法として最も適しているトライ構造について概要を述べ，その検索，更新アルゴリズムについて説明する．

第 4 章では，トライ構造を実装するためのデータ構造について説明し，最も効率的な手法であるダブル配列法について，検索，更新アルゴリズムとともに詳細に述べる．また，ダブル配列法の問題点であるキー追加の速度を改善する手法として，ダブル配列で構築された辞書を変更することなく高速化する手法と，ダブル配列に格納した情報を利用して高速化を実現する手法を提案する．

第 5 章では，共起情報を効率的に記憶検索する手法として，関連研究とともに提案手法であるリンクトライについて述べ，リンクトライにおいて定義されるリンク情報により，冗長性を排除した効率的な記憶が可能となることを示し，同時に提案した構造における共起情報の検索，更新アルゴリズムを説明する．また，ダブル配列を用いたリンクトライのデータ構造についても説明し，リンクトライのデータ構造を提案する上で考案された，複数の辞書を 1 つのダブル配列で管理する手法も述べる．

第 6 章では，提案手法に対して検索速度，記憶効率の理論的評価，及び実験による評価を与え，本手法の有効性を確かめ，考察を加える．

第 7 章では，本研究で得られた諸成果の総括を行い，今後の研究課題について述べる．

関連発表論文

【主論文】

1. 森田和宏, 望月久稔, 山川善弘, 青江順一 : “トライ構造を用いた共起情報の効率的検索アルゴリズム”, 情報処理学会論文誌, Vol.39, No.9, pp.2563–2571 (1998).
2. Kazuhiro Morita, Masafumi Koyama, Masao Fuketa and Jun-ichi Aoe : “A Link Trie Structure of Storing Multiple Attribute Relationships for Natural Language Dictionaries”, International Journal of Computer Mathematics, Vol.72, No.4, pp.463–476 (1999).
3. Kazuhiro Morita, Toru Sumitomo, Masao Fuketa and Jun-ichi Aoe : “Fast Insertion Methods of a Double-Array Structure”, Software Practice & Experience, (条件付採録).

【副論文】

1. 望月久稔, 森田和宏, 獅々堀正幹, 青江順一 : “拡張ハッシュ法における部分文字列検索の設計と実現”, 情報処理学会論文誌, Vol.38, No.2, pp.310–320 (1997).
2. 有田健, 森田和宏, 溝渕昭二, 青江順一 : “特徴ベクトルによる全文検索の一改善法”, 情報処理学会論文誌, Vol.39, No.3, pp.826–829 (1998).
3. Kazuhiro Morita, Toru Sumitomo, Masao Fuketa and Jun-ichi Aoe : “A Fast and Compact Data Structure of Storing Multi-Attribute Relations among Words”, 1998 IEEE International Conference on Systems, Man, and Cybernetics, pp.2791–2796, San Diego, California, USA (Oct. 1998).
4. Masafumi Koyama, Kazuhiro Morita, Masao Fuketa and Jun-ichi Aoe : “A Fast Retrieving Algorithm of Hierarchical Relationships Using Trie Structures”, Information Processing & Management, Vol.34, No.6, pp.761–773 (1998).

5. Kazuhiro Morita, Makoto Okada, Masao Fuketa and Jun-ichi Aoe : "A Method of Storing Multi-Attribute Relations in Natural Language Dictionaries", Proc. of the 18th Int'l Conf. on Computer Processing of Oriental Languages, pp.437–442, Tokushima, Japan (Mar. 1999).
6. Masao Fuketa, Kazuhiro Morita, Sangkon Lee and Jun-ichi Aoe : "Efficient Controlling of Parsing-Stack Operation for LR Parsers", International Journal of Information Sciences, Vol.118, pp.145–157 (1999).
7. EL Sayed Atlam, Masao Fuketa, Kazuhiro Morita and Jun-ichi Aoe : "Similarity Measurement Using Term Negative Weight and Its Application to Word Similarity", Information Processing & Management, Vol.36, No.3 (2000) (印刷中).
8. 辻孝子, 泓田正雄, 森田和宏, 青江順一 : "複合語の分野連想語の効率的決定法", 自然言語処理, Vol.7, No.2 (2000) (印刷中).
9. 岡田真, 安藤一秋, 森田和宏, 青江順一 : "キーワードの遅延抽出を考慮した文書検索構造の効率的構成法", 情報処理学会論文誌 (条件付採録).

【講演報告】

1. 森田和宏, 望月久稔, 獅々堀正幹, 青江順一 : "トライ構造を用いた共起情報の効率的検索アルゴリズム", 情報処理学会第 53 回全国大会, 5L–2, pp.2–81～2–82 (1996).

目 次

図 目 次

表目次

第1章

緒　論

自然言語処理システムにおける単語の共起情報は有用であり，自然言語解析における曖昧性の解決手段 [1, 2]，機械翻訳での訳語の選択 [3, 4]，かな漢字変換の同音異義語の決定 [5, 6, 7]，音声認識における候補文字の制約 [8] などに利用されており，それぞれの応用辞書だけでも膨大な共起情報が必要となる．特に，形態素解析の精度向上でも必要となる基本単語[1]から造語される長単位語（複合語と呼ぶ）は，文書検索システムの索引語における重要語，機械翻訳の訳語解釈などのように，正確な意味解釈をする上でも必要不可欠なものであり，その数は膨大なものとなる．この観点より，共起情報を格納する辞書の効率的記憶検索は重要な課題であるといえる．この課題に関連する研究として，複合語の接頭辞と接尾辞を圧縮して格納する手法 [9] が提案されているが，この手法は一般的な共起情報の格納方式を提案したものではない．

以上のように，自然言語処理システムには様々な共起情報による知識辞書が必要であるが，これらの知識辞書の検索の切り口（キー，見出し語）は，やはり形態素表記である．即ち，抽象的な概念化が行われたとしても，人間が認知する上で概念名も形態素表記に合致する場合が多い．例えば，“タクシー” や “バス” は，“乗り物” と概念化されるが，この概念名も形態素表記となる．従って，共起情報を格納する知識辞書は，自然言語処理システムの基本辞書である形態素辞書の見出し語と融合して構築するのが効率的である．更に，異なる属性の共起情報による全ての知識辞書を形態素辞書の中に統合化して格納できれば，その利便性は非常に高いものとなる．

[1]文章を構成する最小単位の単語，即ち形態素表記を示す．

本論文では，上記のような知識辞書と形態素辞書の統合化などに必要な様々な付加情報を格納できるよう拡張性を持たせた，効率的な共起情報の記憶検索手法を提案する．本手法では，形態素表記が格納されたトライ上の葉ノード間のリンク関数により共起情報を定義するので，登録による記憶の増加はこのリンク情報のみとなる．また，このリンク情報に複数の関係情報を格納する方式を提案することで，種々の自然言語処理システムへの応用性を高める．

以下，2章で共起情報の概要と意義について説明する．また，自然言語処理システムにおける辞書の意義についても述べ，辞書を構成する基本的なアルゴリズムについて説明する．3章では，辞書の構成法として最も適しているトライ構造について概要を述べ，その検索，更新アルゴリズムについて説明する．4章ではトライ構造を実装するためのデータ構造について説明し，最も効率的な手法であるダブル配列法について，検索，更新アルゴリズムとともに詳細に述べる．また，ダブル配列法の問題点であるキー追加の速度を改善する手法を提案する．5章では，共起情報を効率的に記憶検索する手法として，関連研究とともに提案手法であるリンクトライについて述べ，ダブル配列を用いたリンクトライのデータ構造についても説明する．また，リンクトライのデータ構造を提案する上で考案された，複数の辞書を1つのダブル配列で管理する手法も述べる．6章では，提案手法の理論的評価と実験による評価を与え，考察を加える．7章では，本論文のまとめと今後の課題について触れる．

第2章

共起情報と辞書構成法

2.1　緒言

近年の計算機の発展により，複雑で高度な処理を高速に計算させることが可能になってきた．それに伴い，自然言語処理システムにおいても多くの言語知識を利用して，システムの精度を向上させることが可能となってきている．その意味で，自然言語処理システムにおける共起情報の必要性はいっそう増してきており，共起情報を利用するさまざまな研究がなされている．

また，自然言語処理システムにおいて，辞書は必要不可欠なものであり，辞書の量と質によってシステム全体の能力が左右されるため，システム構築の上で，最も労力を割かれるものでもある．

本章では，自然言語処理の基礎となるこれらの共起情報と辞書について，概要を述べることでこれらの意義を確認する．

2.2　共起情報

自然言語処理システムにおける共起情報の有用性はきわめて高く，自然言語における曖昧性や多義性を解消し，システム全体の精度を向上させる上で非常に重要な情報であり，共起情報を利用するさまざまな研究 [10, 2, 11] がなされている．

また，元来人手により収集されてきた共起情報は，収集者の視点によりばらつきが見

られたり，時間的制約による収集量の限界があったが，共起情報自体の精度向上を目指して，大量のコーパスから客観的手法により抽出し，その結果を利用する研究[12, 13]も多くなされている．

これらの共起情報は，通常2単語間に存在する意味的関係を定義し，本論文では基本単語 X，Y 間に関係情報 α が定義されるものとし，(X, Y, α) で表す．この関係情報 α は以下の様々な形態での定義と検索要求がある．

(A) 概念階層の上位と下位の関係（階層関係）

シソーラスで代表される分類体系（概念階層と呼ぶ）は，非常にシンプルな知識表現であり，応用範囲は非常に広い．この表現の基本的推論は，概念“衣類”と“カッターシャツ”などの上位と下位の関係であり，(“カッターシャツ”，“衣類”，上位語) なる共起情報で定義される．また，(“アメリカ”，“国名”，上位語)，(“カナダ”，“国名”，上位語) も定義できる．

(B) 格構造における動詞と名詞句の関係（格関係）

格構造は，動詞に対する名詞句の意味的制約を格納するものであり，例えば動詞“合う”の主格名詞句には，概念“衣類”や表記“気候”の制約があるので，格関係において (“合う”，“衣類”，主格)，(“合う”，“気候”，主格) が得られる．なお，格関係 (“合う”，“カッターシャツ”，主格) は，(“カッターシャツ”，“衣類”，上位語) と (“合う”，“衣類”，主格) の共起情報から推論できる．

(C) 慣用句関係や選択宣言関係

“油を売る”からの (“油”，“売る”，慣用句) や，“馬がいななく”からの (“馬”，“いななく”，選択宣言)．

(D) 同音語判定関係

上記の (B) の例も格関係による同音語判定関係の共起情報を意味する．この他に，名詞句の共起による同音語判定関係として，“シャツの生地”で同音語“記事”を区別する (“衣類”，“生地”，同音語判定) や“アメリカの気候”を“帰港”と区別する (“国名”，“気候”，同音語判定) が定義できる．

(E) 複合語関係

“カナダ国籍”を造語する (“国名”，“国籍”，複合語) や (“アメリカ”，“合衆国”，複合語)

(F) 日英対訳関係（同義語）

(“パソコン”，“personal computer”，対訳)

(G) 同義語関係

(“アメリカ”，“合衆国”，同義語)，(“カッター”，“カッターシャツ”，同義語の短縮語)

広義に解釈すると共起情報は，以上の例にとどまらず（“前略”，“早々”，頭語結語），(“サッカー”，“スポーツ”，分野分類）など非常に多く存在する．

以後，簡単のため関係情報 α は記号 $\alpha_1, \alpha_2, \ldots$ 等で表す．

2.3　辞書構成法

共起情報に限らず，自然言語処理システムにおいて言語知識は大変重要である．計算機が言語を解析したり生成したりすることができるようにするには，言語知識を蓄え，それを使えるようにしなければならない．自然言語処理では，言語に関する知識は，文法規則集や辞書としてまとめられ，文の解析や生成をする際に参照される．

このため，自然言語処理システムでは，辞書にない言葉が使われたり，辞書の情報が誤っていたり矛盾していたりすると，平気で誤った解釈を行ったり，判断を停止してしまったりする．

従って，言語知識を辞書システムから取り出してくることは，自然言語処理のあらゆる段階で必要となる最も基本的な操作となり，辞書構成の量と質によって自然言語処理の能力は大きく左右される．

また，言語知識は，多種多様な分類の仕方があるので，分類ごとに辞書が作成され，自然言語処理のシステムによって必要な辞書を複数選定し，使用している．共起情報をまとめた共起辞書は，その重要性から現在では概ねどのシステムにおいても使用されている．

以上のことから，辞書構成法や辞書システムに関する多くの研究[14, 15]がなされているが，本節では，辞書システムを構成する基本的なアルゴリズムについて述べる．

キー1	レコード1
キー2	レコード2
⋮	⋮
キー$n/4$	レコード$n/4$
⋮	⋮
キー$n/2$	レコード$n/2$
⋮	⋮
キーn	レコードn

図 2.1　表形式の辞書モデル

2.3.1　2分探索法

辞書システムにおける最も単純なモデルは，図 2.1 に示すような表形式のモデルである．すなわち，辞書の1項目が見出し語に対応するキーと見出し語に関する情報に対応する内容（レコード）からなる．共起辞書は，2単語間の関係を調査する目的で使用されるため，検索キーとして X,Y が与えられるが，これらを一意に決定するために，通常連鎖語彙 XY を構成し，この XY をキーとして辞書が構築される．このため，キー数が著しく増加し，記憶量との兼ね合いから，頻繁に使用される共起情報のみを登録している場合が多い．

このモデルにおいて，辞書検索はキーを指定したときにその内容を選び出す探索の問題となる．

最も単純な探索法は線形探索法であるが，この手法は表の先頭から順に入力キーと各要素のキーが一致するかを比較していくため，キー数が n のとき計算量が $O(n)$ となり，実

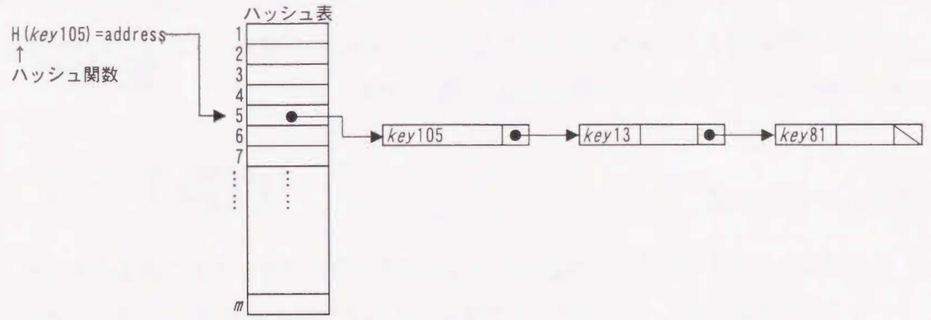


図 2.2　ハッシュ法

用規模の辞書のキー数数千～数十万に対しては現実的な手法ではない．

これに対して，表中のキーが順序関係に従って並べられている場合は2分探索法が利用できる．2分探索法では，まず始めに，表の中央 $n/2$ の要素のキーと入力キーを比較し，もし入力キーが $n/2$ のキーより小さければ求めるキーは $n/2$ より前の位置にあるので，次に $n/4$ のキーと比較する．このように，1回の比較を行うたびに探索範囲が半分になるので，計算量が $O(\log n)$ となる．

2.3.2　ハッシュ法

ハッシュ法は代表的な表探索法であり，キーをハッシュ表に分散記憶し，ハッシュ関数による番地計算によりキーを探索する手法である（図 2.2 ）．このため，計算量は事実上 $O(1)$ となる．

ハッシュ法で用いられるハッシュ表の各要素はバケット（bucket）と呼ばれ，番地が付けられている．バケットの番地は，キー k に対するハッシュ関数 H により $\mathrm{H}(k)$ として計算される．即ち，ハッシュ表の大きさを M とするとき，ハッシュ関数はキー集合を0から $M-1$ の番地の集合に写像するキー番地変換関数である．また，通常キー数に対して M の値は小さいので，異なるキーがハッシュ関数によって同じ番地に写像されるという，衝突が起こる．

ハッシュ法は，如何に衝突を効率的に解消するかが重要な問題であるが，最も簡単な方法としては，チェイン法がある（図 2.2 はチェイン法の例）．チェイン法は，衝突したキーをポインタで鎖のように連結して格納する方法で，動的キー集合への適用が可能であり，表が一杯になっても，あふれ領域を別にとることができる．

2.3.3　トライ法

日本語文の形態素解析を行う場合には，入力文中のどの部分が単語であるかということがわかっていないため，入力文のある位置から末尾までの文字列の中で，その位置から始まる全ての単語を取り出す必要がある．このことを，表形式の辞書モデルで実現するには，まず最初の 1 文字が単語であるか辞書引きし，次に 2 文字の文字列を辞書引きし，これを 3 文字，4 文字と繰り返し，最後に末尾までの文字列で辞書引きする必要があり，この処理を入力文中の各位置で行うので，入力文の文字数を l とすれば辞書引きだけで $O(l)$ の計算量が必要になり，非常に効率が悪い．

このように単語が確定していない場合の辞書引きに適した探索アルゴリズムとしてトライ法がある．トライについては次章以降で詳細に述べる．

2.4　結言

本章では，共起情報の概要と意義について説明した．そして，自然言語処理システムにおける辞書の意義についても説明し，辞書を構成する基本的なアルゴリズムについて，2 分探索法，ハッシュ法について述べ，共起辞書の構成法とその問題点についても述べた．また，トライ法について簡単に触れた．次章ではこのトライ構造について詳しく説明する．

第3章

トライ構造

3.1　緒言

与えられた文字列がキーワードとして登録されているか否かを調べることは計算機処理の基本であり，キー検索と呼ばれる．キー検索技法に対する要求は利用分野により異なるが，近年の計算機の利用分野の拡大により様々なものが研究・実用化されている．

なかでも，キーの表記文字単位に構成された木構造であるトライ構造は，登録キーの総数に依存せず高速な検索ができること，検索失敗位置の特定が容易であること，検索文字列中の接頭辞の検出が容易であることなどの理由により，形態素辞書，かな漢字変換辞書などの自然言語辞書を中心として広く用いられている．

本章では，まずトライ構造の概要について述べ，大規模キー集合におけるトライの記憶量軽減のための圧縮法について説明し，このトライに対するキーの検索，更新アルゴリズムについて述べる．

3.2　トライ構造の概要

トライ (trie) 構造は “retrieval” の真中のスペル “trie” を語源とし，Fredkin[20] により命名された．発音上，木（tree）と区別するためにトライと呼ばれる．2 分探索木や B 木などの木を利用する探索アルゴリズムが，分岐する枝を選ぶのにキー同士の比較を使うのに対し，トライは，キーの値自身による添字付けによって分岐する枝を決定する．このた

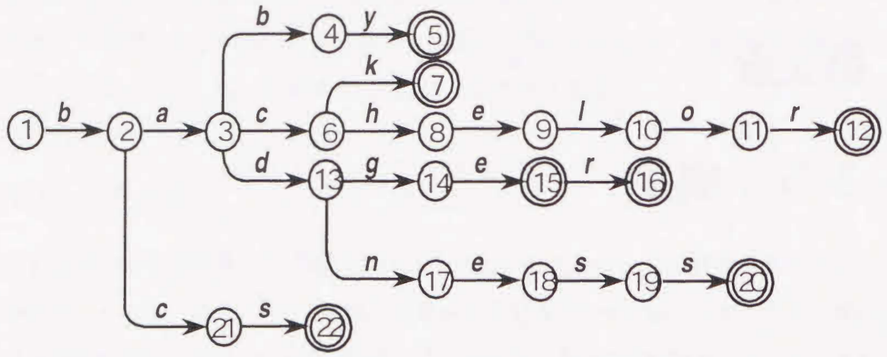


図 3.1　キー集合 $K1$ に対するトライ

め，トライはキーを構成する文字を単位として木構造を構成し，探索も一文字ごとに行われる．

図 3.1 にキー集合

$$K1 = \{\text{“baby”}, \text{“bachelor”}, \text{“back”}, \text{“badge”}, \text{“badger”}, \text{“badness”}, \text{“bcs”}\}$$

に対するトライを示す．図中の2重丸で示されるノードは出力ノードである．図 3.1 に示すように，トライは，キーの共通接頭辞が併合圧縮される．また，トライ上で深さ h の位置にあるノードはキーの h 番目の文字に依存し，その遷移先はキーの $h+1$ 番目の文字により決定される．従って，検索時間は検索キーの長さに依存し，登録されているキーの総数には依存しないので，大規模なキー集合を取り扱った場合でも，検索の高速性が損なわれることはない．

以後の議論のためにここでいくつかの定義を示す．

[定義 3.1]

トライ上で，ノード s からノード t への遷移が記号 a に対して定義されていれば，関数 g を用いて，

$$g(s,a) = t$$

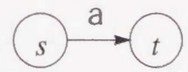


図 3.2　遷移 $g(s,a) = t$

と書き（図 3.2 ），そうでなければ，

$$g(s,a) = fail$$

と書く．この関数 g は goto 関数と呼ばれる．また，連続する遷移の並びをパス (path) と呼び，goto 関数はパスに対しても利用され，パス上の記号による文字列 X に対して，

$$g(s,X) = t$$

と記述する．

（定義終）

[定義 3.2]

goto 関数 g に対する逆関数を g^{-1} と定義し，

$$g(s,a) = t$$

に対して，

$$g^{-1}(t,a) = s$$

と書く．

（定義終）

[定義 3.3]

ノード s に対して，s に入る遷移の数を，

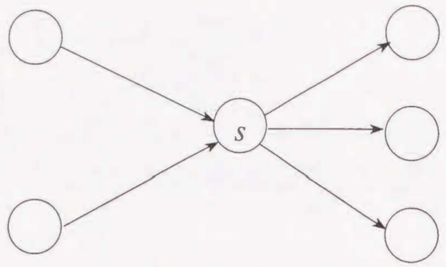


図 3.3　$indegree(s) = 2$，$outdegree(s) = 3$ の例

$indegree(s)$

で表し，s から出る遷移の数を，

$outdegree(s)$

で表す（図 3.3）．トライにおいては，$indegree(s)$ は常に 1 となる．また，$outdegree(s) = 0$ なるノード s を最終ノードと呼ぶが，トライは木構造であるので，葉ノードと呼ぶ場合もある．

（定義終）

[例 3.1]

図 3.1 のトライ上でキー “badger” を検索する場合を考える．まず最初にノード 1 からスタートし，$g(1,\text{'b'}) = 2$ により，ノード 1 から文字 ‘b’ を辿りノード 2 へ到達できるので，文字 ‘b’ がマッチしたことになる．以後同様に，$g(2,\text{'a'}) = 3$ により，ノード 2 から ‘a’ を辿りノード 3 へ，$g(3,\text{'d'}) = 13$，$g(13,\text{'g'}) = 14$，$g(14,\text{'e'}) = 15$，$g(15,\text{'r'}) = 16$ により，ノード 16 に到達する．このノード 16 は，出力ノードであるので，キー “badger” の検索に成功したことになる．

（例終）

トライへのキーの追加，削除はキー X に対して $g(1,X) = t$（t は出力ノード）を定義するか，削除することで簡単に行うことができる．

例 3.1 において，$g(14,\text{'e'}) = 15$ により，ノード 14 から 15 への遷移が起こっているが，このノード 15 は出力ノードである．つまり，キー “badger” を検索する過程において，キー “badge” を発見することが可能であり，このことは，任意の入力文字列に対して，キーの最長一致検索や接頭辞のみが一致する検索を容易にさせる．

このためトライは，語と語の間に空白を持たない自然言語（日本語，中国語など）での形態素解析において，最適な辞書引きを行うアルゴリズムとされている [21]．

3.3　トライの圧縮

前節でも述べたように，トライは共通接頭辞を併合する特徴をもつが，一度パスが分岐すると以降は共通な接尾辞をもつキー同士の間でも接尾辞の併合は行われず，各キーは他のキーとは独立して接尾辞に対して遷移を作るため，接尾辞の記憶に関して記憶量の節約は行われない．

このため，大規模なキー集合を取り扱うと，トライのノード数は膨大なものとなり，より多くの記憶領域を必要とするという問題が生じる．

そこで，トライのノード数を縮小するために，シングルストリングを導入する．導入の準備として，キーに現われない端記号（endmarker）[22]‘#’ をキーの最後尾に付加する．キー集合 $K1$ に端記号を付加したキー集合 $K2$ に対するトライを図 3.4 に示す．図に示されるように，端記号を用いたトライでは最終ノードと出力ノードが常に一致する．

以下に，セパレートノード，シングルストリングの定義を示す．

[定義 3.4]

goto 関数に対して，次のノードを定義する．

$g(r,a) = t$，$outdegree(r) \geq 2$ なるノード t から到達可能な最終ノードまでの遷移列上に，$outdegree(k) \geq 2$ なるノード k が存在しないならば，ノード t をセパレートノード (separate node) と呼ぶ．セパレートノードは，検索中のキーを他のキーと一意に区別する最初のノードであり，セパレートノード以降は遷移の分岐は存在しない．

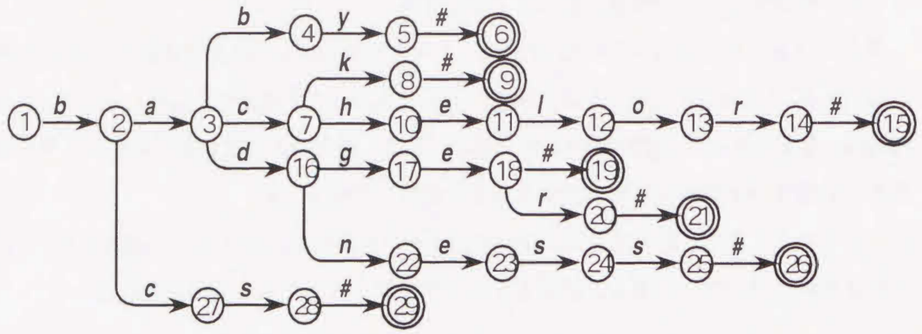


図 3.4　キー集合 $K2$ に対するトライ

（定義終）

[定義 3.5]

$g(r, Y) = t$ において，r がセパレートノード，t が出力ノードとなる文字列 Y をセパレートノード r に対するシングルストリング (single string) と呼び，STR[r] で表す．シングルストリングに対するパスは分岐がなく，また，他のシングルストリングに対するパスと共有される遷移を持たない．

（定義終）

[例 3.2]

図 3.4 において，セパレートノードは，4，8，10，19，20，22，27 であって，セパレートノードと対応するシングルストリングは，

STR[4]=“y#”,

STR[8]=“#”,

STR[10]=“elor#”,

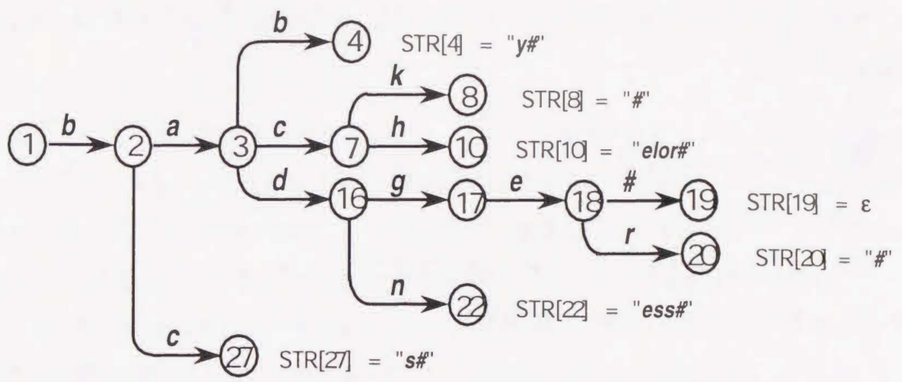


図 3.5　シングルストリングを用いたキー集合 $K2$ に対するトライ

STR[19]=ϵ,

STR[20]=“#”,

STR[22]=“ess#”,

STR[27]=“s#”

となる．シングルストリングが存在しない時は ϵ で表す．

（例終）

図 3.4 のトライをシングルストリングを用いて表したトライを図 3.5 に示す．シングルストリングを用いることで，ノード数は大幅に軽減される．

以後の議論においてトライとは，全てシングルストリングを用いたトライをさすものとする．また，断りのない限り，文字（記号）は $a, b, c, \ldots$ を使用し，文字列（記号列）は X, Y, Z を使用する．また，端記号付きの文字列を $X\#, Y\#, Z\#$ と表す．

3.3.1　トライの検索アルゴリズム

本項では，トライの検索アルゴリズムを示す．但し，今後の議論のために関数Trie_ Searchとして与える．

［関数 **Trie_Search(T,X)**］

入力：トライT，キーX．

出力：$g(1, X\#) = s$なるノードs．検索に失敗したときは，$fail$．

$X\# = a_1 a_2 \ldots a_n a_{n+1}$，$a_{n+1}$ =‘#’と表す．

手順（T-1）：｛トライ検索の初期化｝

トライTのノード番号を表す変数$node$を初期ノード1に，文字位置を表す変数posを1にセットする．

手順（T-2）：｛トライ検索｝

$next = g(node, a_{pos})$なる$next$が$fail$ならば，Xはトライに格納されていないので，$fail$を返す．$fail$でなければ，$node = next$とし，posをインクリメントする．

手順（T-3）：｛探索の終了と出力の判定｝

$node$がセパレートノードでなければ，手順（T-2）へ戻り，セパレートノードならば，キーの残りの文字列$X_{pos} = a_{pos} a_{pos+1} \ldots a_{n+1}$と，$node$に対するシングルストリングSTR[$node$]とを比較する．比較した結果が等しければキー$X$が見つかったので，$node$を出力し，等しくなければ検索に失敗したので$fail$を返す．

（関数終）

［例 **3.3**］

図3.5のトライ上でキー“badness#”を検索する場合を考える．まず，手順（T-1）で，$node$，posを1にセットし，手順（T-2）で，$next = g(node, a_{pos})$= g(1,‘b’) = 2を得る．$next$は$fail$ではないので，$node = 2$，$pos = 2$となる．手順（T-3）で，posはセパレートノードではないので，手順（T-2）に戻る．以後同様にg(2,‘a’) = 3，g(3,‘d’) = 16，g(16,‘n’) = 22，となり，$pos = 5$のとき，$node = 22$はセパレートノードなので，手順

（T-3）で残りの文字列“ess#”と，シングルストリングSTR[22]=“ess#”を比較し，等しいので，$node = 22$を得る．

（例終）

図3.5のトライ上で検索を失敗するのは，トライを検索中に文字を辿れなくなる場合と，シングルストリングと検索文字列の残りの部分とがマッチしなかった場合の2通りがある．例えば，“bag#”の検索は，ノード3までは文字‘b’と‘a’を辿れるが，ノード3から文字‘g’を辿ることができないので，検索に失敗する．また“babel#”の検索は，ノード4まで検索に成功する．ノード4のシングルストリング“y#”と検索文字列の残りの文字列“el#”を比較するとマッチしないので，“babel#”の検索は失敗する．

3.3.2　トライの追加アルゴリズム

トライへのキーの追加は，トライの検索を失敗したノードから，新たに残りの文字列に対するノードを作成することで追加される．このため，前項で述べた，トライで検索を失敗する2通りの場合について，追加処理を施す必要がある．

以下にトライの追加アルゴリズムを関数Trie_Insertとして示す．但し，ここでは関数Trie_Searchにおいて$fail$を返す部分の変更処理についてのみ示す．その他の処理は，関数Trie_Searchと同様である．

［関数 **Trie_Insert(T,X)**］

入力：トライT，キーX．

出力：$g(1, X\#) = s$なるノードs．

手順（T-2）で$fail$を返すときの変更

手順（T-2a）：｛残りの文字列の登録｝

$next = g(node, a_{pos})$を定義し，$node = next$とし，posをインクリメントする．ここで，$node$はセパレートノードとなるので，キーの残りの文字列$X_{pos} = a_{pos} a_{pos+1} \ldots a_{n+1}$をシングルストリングSTR[$node$]に格納し，$node$を返す．

手順（T-3）で $fail$ を返すときの変更

手順（T-3a）：|共通接頭辞の登録|

キーの残りの文字列 X_{pos} と，シングルストリング STR[$node$]$= b_1 b_2 \ldots b_{m+1}$ の，共通接頭辞を $X_p = a_{pos} a_{pos+1} \ldots a_{pos+p}$ とし，STR[$node$] から共通接頭辞を除いた文字列を $Y = b_{p+1} b_{p+2} \ldots b_{m+1}$ とする．X_p に対して，$next = g(node, X_p)$ を定義し，$node = next$，$pos = pos + p + 1$ とする．

手順（T-3b）：|残りの文字列の登録|

シングルストリングの残りの文字列 Y に対して，$next = g(node, b_{p+1})$ を定義する．これがセパレートノードのなるので，新しいシングルストリング STR[$next$] に文字列 $b_{p+2} \ldots b_{m+1}$ を格納する．次に，キーの残りの文字列 X_{pos} に対して，手順（T-2a）と同じ処理をし，$node$ を返す．

（関数終）

[例 3.4]

図 3.5 のトライ上で，キー “babylon#” の追加を考える．まずノード 1 から検索を開始し，$g(1,\text{‘b’}) = 2$，$g(2,\text{‘a’}) = 3$，$g(3,\text{‘b’}) = 4$ により，ノード 4 まで遷移する．ここで，ノード 4 はセパレートノードであるので，キーの残りの文字列 “ylon#” と，シングルストリング STR[4]=“y#” を比較する．比較が失敗するので手順（T-3a）において，共通接頭辞 “y” に対して，$g(4,\text{‘y’}) = 30$ を定義する．

次に手順（T-3b）において，STR[4] の残りの文字列 “#” に対して，$g(30,\text{‘\#’}) = 31$ を定義し，新しいシングルストリング STR[31] に，STR[31]$= \epsilon$ を設定する．

最後に手順（T-2a）において，キーの残りの文字列 “lon#” に対して，$g(30,\text{‘l’}) = 32$ を定義し，STR[32] に文字列 “on#” を設定し，ノード 32 を返す．

図 3.5 のトライに，キー “babylon#” を追加した例を図 3.6 に示す．

（例終）

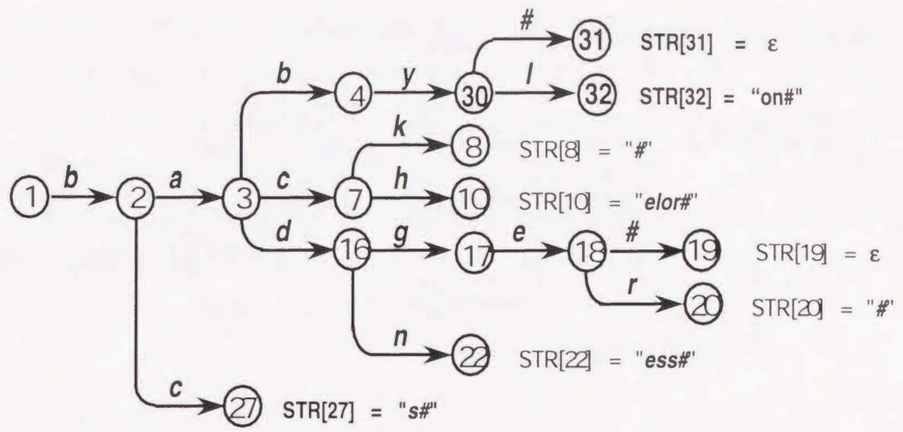


図 3.6　図 3.5 のトライに，キー “babylon#” を追加した例

3.3.3　トライの削除アルゴリズム

キーの削除については，関数 Trie_Search において検索が成功した場合にそのキーに対するセパレートノードを削除することで簡単に行うことができる．

以下にトライの削除アルゴリズムを関数 Trie_delete として示す．但し，ここでは関数 Trie_Search において検索が成功する部分の変更処理についてのみ示す．その他の処理は，関数 Trie_Search と同様である．

[関数 Trie_Delete(T,X)]

入力：トライ T，キー X．

出力：キーを削除したときは $g(1, X\#) = s$ なるノード s．削除キーがなかった（検索に失敗した）ときは，$fail$．

手順（T-3）で検索成功するときの削除処理

手順（T-3c）：|セパレートノードの削除|

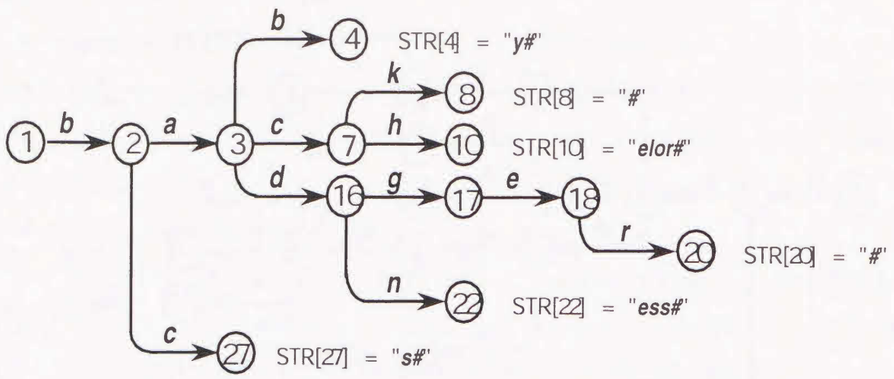


図 3.7　図 3.5 のトライから，キー “badge#” を削除した例

シングルストリング STR[*node*] とトライ上のノード *node* を削除し，*node* を返す．

（関数終）

［例 3.5］

図 3.5 のトライ上で，キー “badge#” の削除を考える．ノード 1 から検索を開始し，$g(1,\text{‘b’}) = 2$，$g(2,\text{‘a’}) = 3$，$g(3,\text{‘d’}) = 16$，$g(16,\text{‘g’}) = 17$，$g(17,\text{‘e’}) = 18$，$g(18,\text{‘\#’}) = 19$ により，ノード 19 まで遷移する．ノード 19 はセパレートノードであるが，STR[19]$= \epsilon$ であるので，キー “badge#” を検索成功する．ここで，手順（T-3c）において，ノード 19 を削除することにより，$g(18,\text{‘\#’}) = 19$ が削除され，キー “badge#” が削除される．

図 3.5 のトライから，キー “badge#” を削除した例を図 3.7 に示す．

（例終）

図 3.7 で示すように，キー “badge#” が削除されると，キー “badger#” のセパレートノードがノード 20 から 17 に変わるが，このセパレートノードの変更に伴うトライの再構成処理を行うことは，削除時間の無駄となるだけで，検索時間には影響を与えないので，削除アルゴリズムにこの処理は含まれない．

3.4　結言

本章では，形態素辞書，かな漢字変換辞書などの自然言語辞書を中心として広く用いられているキー検索技法であるトライ構造について，概要を述べるとともに，大規模キー集合におけるトライの記憶量軽減のための圧縮法について説明し，トライに対するキーの検索，更新アルゴリズムについて述べた．

トライを実現するにあたって，記憶領域の非効率さを改善する手法が提案されているが，次章ではそのひとつであるダブル配列法について説明する．

第4章

ダブル配列法

4.1 緒言

本章では，トライを実現するデータ構造として，配列構造，Johnsonの方法を簡単に述べ，青江[24]の提案したダブル配列について説明する．ダブル配列はトライを2つの配列で表現し，検索の高速性とコンパクト性の両方の特徴を持つ優れたデータ構造である．

ダブル配列法については，検索，更新アルゴリズムを詳細に述べるとともに，ダブル配列法の問題点であるキー追加の速度を改善する手法として，ダブル配列で構築された辞書を変更することなく高速化する手法と，ダブル配列に格納した情報を利用して高速化を実現する手法を提案する．

4.2 トライのデータ構造

トライのデータ構造を考える場合，トライの特徴である検索の高速性を考慮する必要がある．前章でも述べたが，トライの検索時間はキーの長さkに依存し，登録されているキーの総数には依存しない．つまり，goto関数$g(s,a)=t$の検索時間が$O(1)$を実現できれば，キーの検索時間を$O(k)$とすることが可能となる．

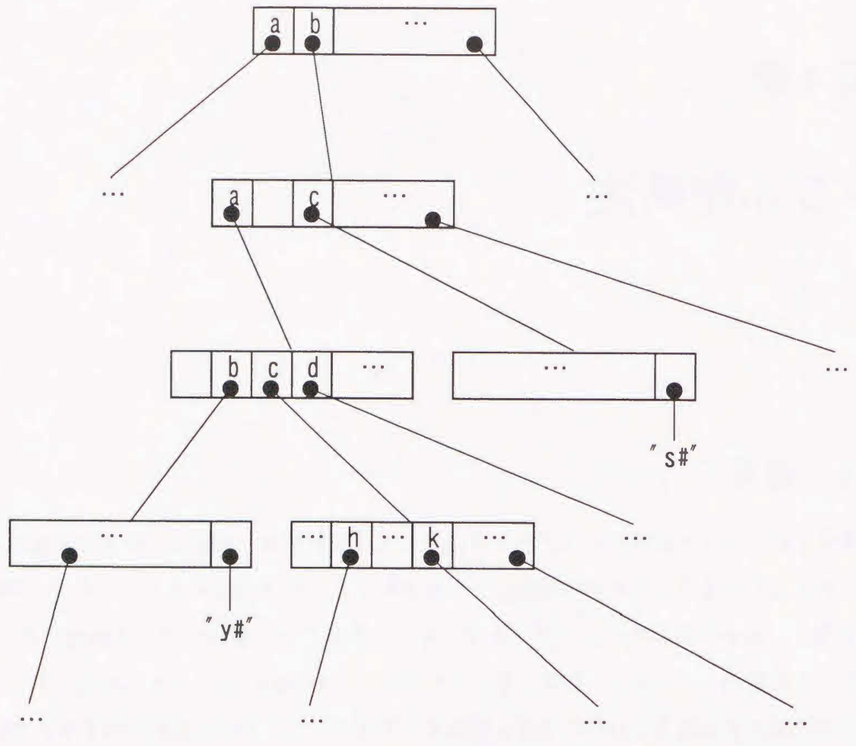


図 4.1　配列を用いたトライの例

4.2.1　配列構造

goto 関数の検索時間を $O(1)$ とする最も簡単なデータ構造は，入力記号 (文字の種類) の総数を e としたとき，トライの各ノードに対して，長さ e の配列を用意することである [25, 21].

配列を用いたトライの例を図 4.1 に示す．図のように，各ノードから遷移される子ノードが存在する場合は子ノードへのポインタを定義することで，トライを実現できる．また，シングルストリングについては，各ノードの配列にシングルストリングへのポインタを用意することで実現できる．

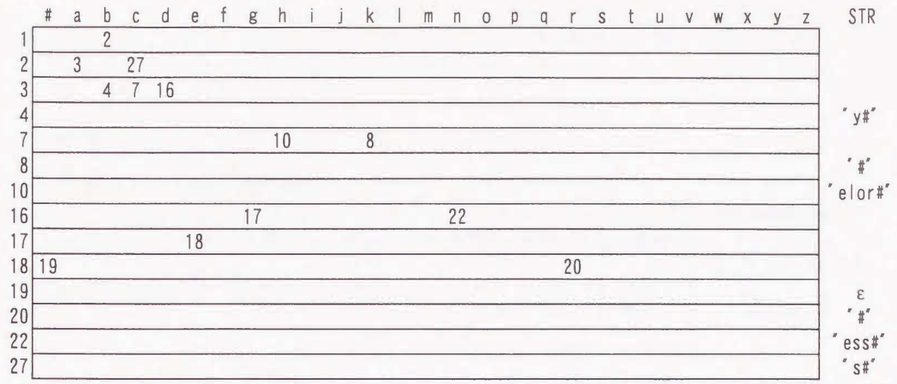


図 4.2　配列を用いたキー集合 $K2$ に対するトライ

[例 4.1]

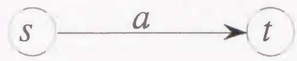
図 3.5 のトライを配列構造を用いて構築したものを図 4.2 に示す．図 4.2 のように，子ノードへのポインタは，各ノードに対してノード番号が与えられている場合，ノード番号を設定しておくことができる．

(例終)

図 4.2 を見れば自明であるが，配列によるトライ構造は，トライの総ノード数を n とすると，$n \cdot e$ の容量が必要となるため記憶量にかなりの無駄が生じる．これは大規模でスパースなデータに対しては更に顕著になる．

4.2.2　Johnson の方法

配列構造の欠点であるスパースな場合における記憶容量の無駄をなくすため，S.C. Johnson により提案された手法である（Aho ら [26] により紹介されている）．また，より記憶容量を減らすために，ノードから出るアークの類似性を利用している．この方法では，次の 4 つの配列を使う．



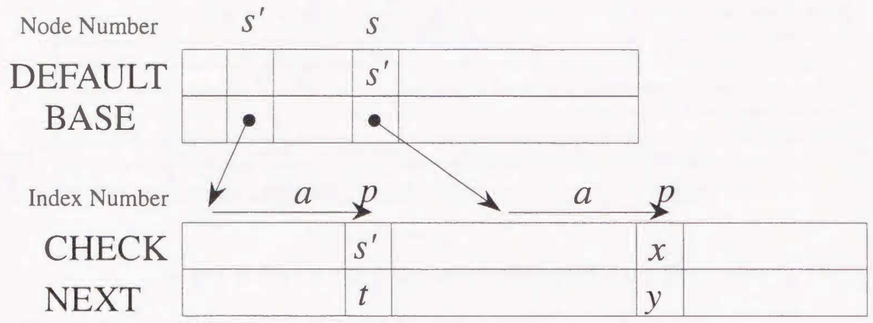


図 4.3　Johnson の方法

DEFAULT ： BASE が正しくなかったときに代わりとして使うノード番号を格納

BASE ： NEXT, CHECK に記憶した情報に対するオフセットを格納

CHECK ： どのノードからのアークかを確認するためにノード番号を格納

NEXT ： 次のノード番号を格納

Johnson の方法では遷移 $g(s,a)=t$ に対して，図 4.3 にあるように，BASE[s] を CHECK へのベースポインタとして，その値にラベル a の内部表現値 (numerical value) をオフセット値として加えた番地[1] $p(p=\mathrm{BASE}[s]+\mathrm{N}[a])$（N[$a$] はラベル a の内部表現値）を求める．そして，現在のノード番号 s を CHECK[p] に，次に進むべきノード番号 t を NEXT[p] に格納する．また，ノードから出るアークの類似性を利用して，情報を畳み込むために，検索が失敗したときに移行するノード番号を DEFAULT[s] に格納する．

[1] 配列 CHECK, NEXT の番地は，Index Number とする．

このデータ構造に対して $g(s,a)$ を検索する関数 Goto(s,a) を次に示す．

[関数 **Goto**(s,a)]

```
begin
    if s = 0 then return(0)
    if CHECK[BASE[s] + N[a]] = s then return(NEXT[BASE[s] + N[a]])
    else Goto(DEFAULT[s],a)
end
```

（関数終）

Johnson の方法ではトライの構成法が提案されていないが，配列構造に比べてコンパクトな記憶量を実現することができる．しかし，goto 関数の検索時間が，配列 DEFAULT をたどる回数に依存するため，$O(1)$ を実現できない．

4.2.3　ダブル配列法

青江 [24, 27, 23] の提案したこの方法は，Johnson の方法を改良したもので，現在のノード番号とラベルの内部表現値との関係を利用して，次に進むノード番号をダブル配列のインデックスとうまく対応させているので，Johnson の方法における配列 NEXT は不要となる．これは，ノード番号を変更することによって可能となる．この手法では，次に示す 2 つの配列（ダブル配列）を使用する．

BASE ： CHECK に記憶した情報に対するオフセットを格納

CHECK ： どのノードからのアークかを確認するためにノード番号を格納

ノード s からノード t に向かうラベル a をもつ遷移 $g(s,a)=t$ を図 4.4 のように格納する．

この図からわかるように，BASE[s] を CHECK へのベースポインタとして，その値にラベル a の内部表現値 (numerical value) をオフセット値として加えた番地 $t(t=\mathrm{BASE}[s]+\mathrm{N}[a])$

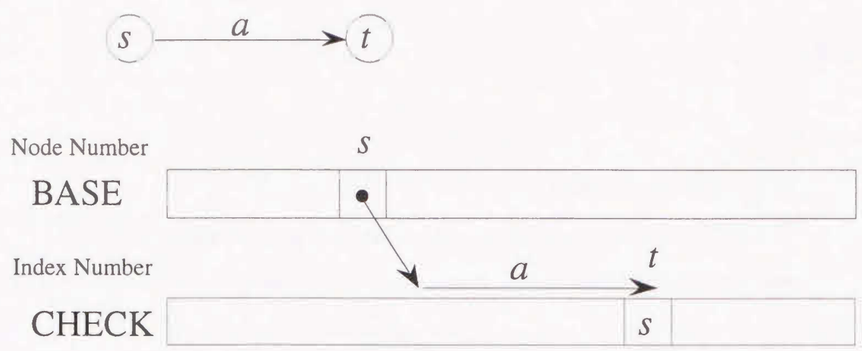


図 4.4　ダブル配列構造

を求める．そしてそこで得たtが次のノードとなる．従って，$g(s,a)$を確認するための関数 Forward(s,a) は，ダブル配列で使用されている最大のインデックス番号をDA_SIZEとすると，次で示される．

[関数 **Forward**(s,a)]

begin
　$t = \mathrm{BASE}[s] + \mathrm{N}[a]$;
　if $(0 < t < DA\_SIZE + 1)$ **and** $(\mathrm{CHECK}[t] = s)$ **then** return(t)
　else return(0);
end;

（関数終）

ダブル配列上のインデックス番号はトライのノード番号と一対一に対応する．以後簡単のため，両者の値を一致させ，両者を同等に扱って説明する．また，ノード番号は1以上の値をとるので，ダブル配列の未使用要素は，そのインデックスqに対し，$\mathrm{BASE}[q] = \mathrm{CHECK}[q] = 0$とする．

ダブル配列において，シングルストリングを扱うためには，セパレートノードからシングルストリングへの連結を考える必要がある．そこで，以下の定義を与える．

[定義 4.1]

セパレートノードs_rにおいて，次が成立する．

$\mathrm{BASE}[s_r] < 0$

（定義終）

[定義 4.2]

セパレートノードs_rと，そのシングルストリング$\mathrm{STR}[s_r] = b_1 b_2 \ldots b_m$に対して，次が成立する．

$p = -\mathrm{BASE}[s_r]$;
　$\mathrm{TAIL}[p] = b_1$，$\mathrm{TAIL}[p+1] = b_2$，…，$\mathrm{TAIL}[p+m-1] = b_m$

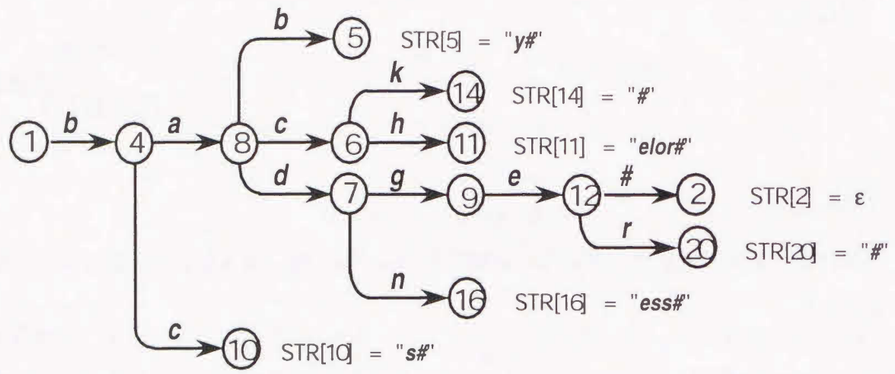
（定義終）

定義 4.1 は，$\mathrm{BASE}[s]$が負ならば，sはセパレートノードであることを示し，定義 4.2 は，対応するシングルストリングが，配列 TAIL の位置 $-\mathrm{BASE}[s]$からアクセスできることを意味する．また配列 TAIL によって，各セパレートノードに対応したシングルストリングを1つの配列で管理できるようになる．

[例 4.2]

図 3.5 のトライをダブル配列法を用いて構築した例を図 4.5 に示す．配列 TAIL 上の記号$はレコード情報へのポインタを表し，記号?はごみを表す．また内部表現値は，端記号'#'を1，文字a～zを2～27とする．ダブル配列は，配列構造において未使用要素を埋めるように各配列をスライドさせ，併合したような構造をとっているため，図 4.5(b) を分解すると図 4.6 のようになり，図 4.2 の各ノードをスライドさせていることがわかる．

（例終）



(a) $K2$ に対するトライ

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-13	0	6	-17	2	1	2	6	-10	-1	1	0	-20	0	-24	0	0	0	-22
CHECK	1	12	0	1	8	8	8	4	7	4	6	9	0	6	0	7	0	0	0	12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	e	l	o	r	#	$	?	?	?	s	#	$	$	?	?	?	y	#	$	#	$

	22	23	24	25	26	27	28
TAIL	#	$	e	s	s	#	$

(b) $K2$ に対するダブル配列

図 4.5　$K2$ に対するトライとダブル配列

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-13	0	6	-17	2	1	2	6	-10	-1	1	0	-20	0	-24	0	0	0	-22
CHECK	1	12	0	1	8	8	8	4	7	4	6	9	0	6	0	7	0	0	0	12

ノード1

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t
			6																	
			1																	

ノード4

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n
	2		-10											
	4		4											

ノード8

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r
		-17	2	1														
		8	8	8														

ノード6

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
									-1		-20								
									6		6								

ノード7

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
							6							-24					
							7							7					

ノード9

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n
					1									
					9									

ノード12

#	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s
-13																		-22	
12																		12	

図 4.6　ダブル配列をノードで分解した例

ダブル配列による状態の遷移は2つの式を確認するだけであるため，常に$O(1)$で行われ，極めて高速である．また，記憶量は，トライのノード数と未使用インデックス数に依存するが，未使用インデックス数が少なければ，非常にコンパクトになる．

次節以降では，このダブル配列法について詳しく述べる．

4.3　ダブル配列の検索アルゴリズム

以下にダブル配列によるキーの検索アルゴリズムを関数Trie_Searchとして示す．関数Trie_Searchでは以下の全域変数を利用する．

index: 現在の処理中のノード番号．

t: 遷移先のノード番号．

pos: 入力文字列の現在の処理位置を表す．

S_TEMP: TAIL中のストリングを格納．

[関数 Trie_Search(D(K),X)]

入力：キー集合Kに対するダブル配列D(K)，キーX．

出力：$g(1, X\#) = s$なるノードs．検索に失敗したときは，0．

$X\# = a_1 a_2 \ldots a_n a_{n+1}$，$a_{n+1}$ =‘#’と表す．

手順（D-1）：|変数の初期化|

index ← 1;

pos ← 0;

手順（D-2）：|トライの検索|

repeat

pos ← *pos* + 1;

t ←Forward(*index*, a_{pos});

if $t = 0$ **then** return(0);

index ← *t*;

until BASE[*index*] < 0;

手順（D-3）：|残り文字列の比較|

if a_{pos} = ‘#’ **then** return(*index*);

else

begin

TAIL上の −BASE[*index*] の位置より‘#’までの文字列を取り出し，

S_TEMP にセットする;

if (キーの残りの文字列と *S_TEMP* が一致する) **then**

return(*index*);

else

return(0);

end

（関数終）

手順（D-1）は各種変数の初期化を行っている．まず，*index*を初期状態である1にし，キー中の現在の取り扱い文字の位置を表わす*pos*を0にしてダブル配列による検索準備をしている．手順（D-2）はダブル配列により構築されたトライ上での探索を行う．ここではノード*index*が最終状態になるまで一文字ずつ探索を進める．ノード*index*がセパレートノードになる前に遷移先がなくなれば検索は失敗する．BASE[*index*] < 0となるとノード*index*はセパレートノードである．a_{pos}が‘#’であれば検索は成功するが，‘#’でなければTAIL上の −BASE[*index*] の位置より‘#’までのシングルストリングを取り出す．このシングルストリングがキーの残りの文字列と一致すれば検索は成功し，一致しなければ失敗する．

[例 4.3]

図4.5のダブル配列でのキー “baby#” の検索を考える．まず最初に，手順（D-1）で検索に使用される変数*index*と*pos*にそれぞれ1，0を代入する．次に手順（D-2）で，*pos*に*pos*+1 = 0+1 = 1を代入し，BASE[1]+3 = 4，CHECK[4] = 1よりForward(1,‘b’) = 4となり，

これをtに代入する．tは0ではないので，$index = t$となる．同様に，Forward(4,'a') = 8，Forward(8,'b') = 5となり，$pos = 3$のとき，$index = 5$，BASE[5] = $-17 < 0$であるのでノード5はセパレートノードとなる．最後に手順（D-3）でTAIL中の$-$BASE[5] = 17の位置から，シングルストリング"y#"をとりだし，S_TEMPに格納する．キーの残りの文字列とS_TEMPは一致するので検索は成功となり，$index = 5$を返す．

（例終）

4.4　ダブル配列の更新アルゴリズム

4.4.1　追加アルゴリズム

キーの追加は，追加キーがキー集合中に存在しないときに行われるため，まず追加キーの検索を行い，検索が失敗した時に追加される．

以下にダブル配列の追加アルゴリズムを関数Trie_Insertとして示す．但し，ここでは関数Trie_Searchにおいてreturn(0)の実行の変更処理についてのみ示す．その他の処理は，関数Trie_Searchと同様である．

[関数 **Trie_Insert**(D(K),X)]

入力：キー集合Kに対するダブル配列D(K)，キーX．

出力：$g(1, X\#) = s$なるノードs．

キーの残りの文字列を，$X_{pos} = a_{pos}a_{pos+1}\cdots a_{n+1}$とする．

手順（D-2a）｜手順（D-2）のreturn(0)の変更｜

begin

　　$t \leftarrow$A_INSERT($index, X_{pos}$);

　　return(t);

end

手順（D-3a）｜手順（D-3）のreturn(0)の変更｜

begin

　　$t \leftarrow$B_INSERT($index, X_{pos}, S\_TEMP$);

　　return(t);

end

（関数終）

追加アルゴリズムでは以下の変数や関数を利用する．

$LISTR, LISTH$: 終端記号'#'を含む入力記号の部分集合．

NUM($LIST$): $LIST$の要素数を返す関数．

$TAIL\_POS$: TAILの長さに1を加えた値を保持するグローバル変数であって，初期値は1である．

W_BASE(idx, val): BASE[idx] $\leftarrow$ valを行う．また，$idx > DA_SIZE$のときのDA_SIZEの更新も行う．今後の拡張のために用意．

W_CHECK(idx, val): CHECK[idx] $\leftarrow$ valを行う．また，$idx > DA_SIZE$のときのDA_SIZEの更新も行う．今後の拡張のために用意．

SET_STR(p, Y): 文字列Yを配列TAILのインデックスpの位置から格納し，pがTAIL_POSと等しければTAIL_POSにYの長さを加えた値を，そうでなければTAIL_POSの値を返す．

[関数 **A_INSERT**($index, X_{pos}$)]

入力：現在のノード番号$index$，キーの残りの文字列X_{pos}．

出力：$g(1, X\#) = s$なるノードs．

　　begin

(a-1)　　$t \leftarrow$ BASE[$index$] + N[a_{pos}];

(a-2)　　**if** CHECK[t] > 0 **then**

　　begin

(a-3)　　$LISTR \leftarrow$ GET_LIST($index$);

```
(a-4)          LISTH ← GET_LIST(CHECK[t]);
(a-5)          if NUM(LISTH) > NUM(LISTR) + 1
               then
(a-6)              index ←MODIFY(index, index, a_pos, LISTR);
               else
(a-7)              index ←MODIFY(index,CHECK[t],φ, LISTH);
          end;
(a-8)     return INS_STR(index, X_pos, TAIL_POS);
     end;
```

（関数終）

関数A_INSERTは手順（D-2a）から呼ばれ，追加キーがダブル配列上を検索中に失敗したときに，新たな状態を追加する場所を探して追加を行う．

[関数 **GET_LIST**(*parent*)]

入力：親ノード番号 *parent*.

出力：*parent* から遷移する全ての入力記号の集合.

入力記号全体の集合を I とする.

```
     begin
(g-1)     LIST ← φ;
(g-2)     r ← BASE[parent];
(g-3)     for each c in I do
          begin
(g-4)         if CHECK[r+N[c]] = parent then LIST = LIST ∪ c;
          end;
(g-5)     return (LIST);
     end;
```

（関数終）

関数GET_LISTは関数A_INSERTから呼ばれ，関数MODIFYでの変更対象となるノードから遷移する入力記号の集合を返す.

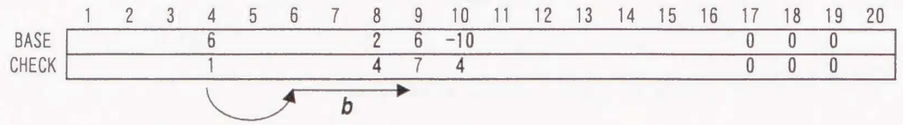
[関数 **MODIFY**($cur, idx, a, LIST$)]

入力：現在のノード番号 cur，変更対象のノード番号 idx，$LIST$ に付加する記号 a，idx から遷移する記号の集合 $LIST$.

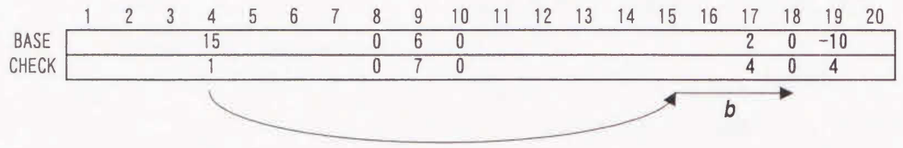
出力：変更された現在のノード番号 cur.

```
     begin
(m-1)     oldbase ← BASE[idx];
(m-2)     W_BASE(idx,X_CHECK(LIST ∪ {a}));
(m-3)     for each c in LIST do
          begin
(m-4)         t ← oldbase + N[c];
(m-5)         t' ← BASE[idx] + N[c];
(m-6)         W_CHECK(t', idx);
(m-7)         W_BASE(t',BASE[t]);
(m-8)         if BASE[t] > 0 then
              begin
(m-9)             for each q such that CHECK[q] = t do
(m-10)                W_CHECK(q, t');
(m-11)                if t = cur then cur ← t';
              end;
(m-12)        W_BASE(t, 0);
(m-13)        W_CHECK(t, 0);
          end;
(m-14)    return(cur);
     end;
```

（関数終）



(a) ノード追加時の衝突例



(b) ノード移動例

図 4.7　関数 MODIFY の動作例

関数 MODIFY は関数 A_INSERT から呼ばれ，新たなノードが追加されたときのダブル配列上でのノードの衝突を回避するために，ノードの移動（ノード番号の変更）を行う．たとえば図 4.5 において，ノード 4 から文字 'b' の遷移を追加するとき，図 4.7(a) のようにノード 9 は既に使用されている．そこで，図 4.7(b) のように，ノード 4 の子ノード全てを移動することによって，文字 'b' の遷移先を確保する．

[関数 **X_CHECK**($LIST$)]

入力：文字集合 $LIST$.

出力：$LIST$ に含まれる文字 c 全てが CHECK$[q + \mathrm{N}[c]] = 0$ を満足する最小のインデックス q．但し，$q > 1$.

```
        begin
(xx-1)      q = 1;
(xx-2)      while q ≤ DA_SIZE do
            begin
```



```
(xx-3)      flg ← true;
(xx-4)      for each c in LIST do
            begin
(xx-5)          if CHECK[q + N[c]] <> 0 then flg ← false;
            end;
(xx-6)      if flg = true then return (q);
(xx-7)      q = q + 1;
        end;
(xx-8)  return (q);
    end;
```

（関数終）

関数 X_CHECK は，ダブル配列上で遷移を追加するとき，遷移先のノードを確保する（遷移先が未使用ノードである）ような適切な BASE 値を探す．

[関数 **INS_STR**($index, X_{pos}, tailpos$)]

入力：現在のノード番号 $index$，キーの残りの文字列 X_{pos}，配列 TAIL へのシングルストリングの設定位置 $tailpos$

出力：$g(1, X\#) = s$ なるノード s.

```
        begin
(s-1)       t ← BASE[index] + N[a_pos];
(s-2)       W_CHECK(t, index);
(s-3)       W_BASE(t, −tailpos);
(s-4)       TAIL_POS ← SET_STR(tailpos, X_pos+1);
(s-5)       return (t);
        end;
```

（関数終）

関数INS_STRは，セパレートノードへの遷移の定義と，シングルストリングを配列TAILに設定する．

[関数 **B_INSERT**($index, X_{pos}, Y$)]

入力:現在のノード番号$index$，キーの残りの文字列X_{pos}，比較対照の文字列$Y = b_1 b_2 \ldots$

出力：$g(1, X\#) = s$なるノードs.

X_{pos}とYの共通接頭辞を$Z = c_1 c_2 \ldots c_m$，Zを除いた文字列をそれぞれX_{m+1}，Y_{m+1}とする．

```
      begin
(b-1)     old_pos ← −BASE[index];
(b-2)     for i ← 1 to m do
          begin
(b-3)         W_BASE(index,X_CHECK({c_i}));
(b-4)         W_CHECK(BASE[index] + N[c_i],r);
(b-5)         index ← BASE[index] + N[c_i];
          end;
(b-6)     W_BASE(index,X_CHECK({a_{m+1}, b_{m+1}}));
(b-7)     INS_STR(index, Y_{m+1}, old_pos);
(b-8)     return INS_STR(index, X_{m+1}, TAIL_POS);
      end;
```

（関数終）

手続きB_INSERTは，既にTAILに追加されているシングルストリングと，追加キーのダブル配列上に存在しない部分とがマッチしないときに，シングルストリングとの共通部分のノードを作成し，共通でない部分を新たなシングルストリングとして，TAIL上に格納している．

[例 4.4]

図4.5のダブル配列でのキー“bbq#”の追加を考える．まず最初に，手順（D-1），手順（D-2）で，$pos = 1$，$index = 4$となり，次のBASE[4] + 'b' = 9, CHECK[9] $\neq 4$よりForward(4,'b') = 0となり，手順（D-2a）より関数A_INSERTが呼び出される．

関数A_INSERTでは，(a-3)行でノード4から遷移する記号の集合{'a','c'}が，(a-4)行でノードCHECK[9] = 7から遷移する記号の集合{'g','n'}が取り出され，(a-5)行の条件式により，(a-6)行において関数MODIFYが呼び出される．

関数MODIFYでは，集合{'a','c'}に入力記号を付加した集合{'a','b','c'}について，(m-2)行で関数X_CHECKが呼び出され，BASE値15が返され，これがノード4の新たなBASE値になる[2]．次に入力記号を含めない集合{'a','c'}に対する遷移をノード17，19に移動するため，ノード8，10の内容をノード17，19にコピーし（(m-4)〜(m-8)行），古いノード8，10の内容を削除する（(m-12)〜(m-13)行）．また，(m-9)〜(m-11)行では，ノード17，19に移動したことによる子ノード5，6，7のCHECK値を修正している．

関数MODIFYでの処理が終わると，残りの入力文字列“bq#”を挿入する準備が整うので，関数INS_STRで挿入処理を行う．関数INS_STRでは，ノード4から18への遷移を定義し，残りの“q#”を配列TAILに格納している．

図4.5のダブル配列にキー“bbq#”を追加した例を図4.8に示す．

（例終）

4.4.2 削除アルゴリズム

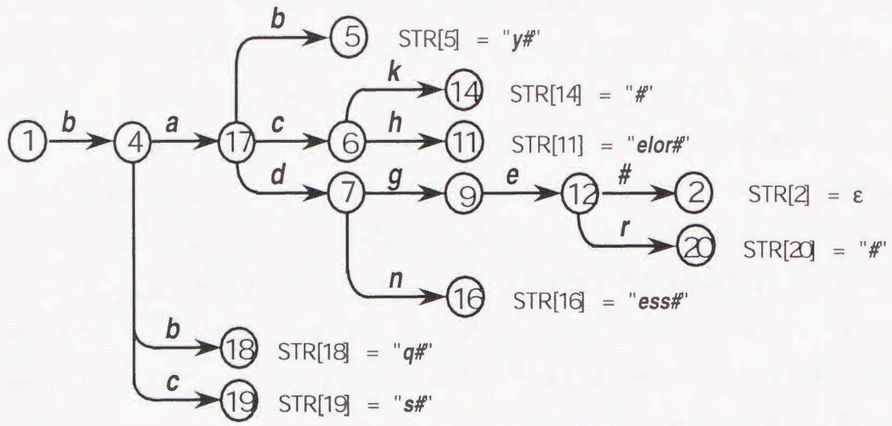
キーの削除は，ダブル配列上に削除キーが存在したときに，そのキーの定義を削除すればよい．

以下にダブル配列の削除アルゴリズムを関数Trie_Deleteとして示す．但し，ここでは関数Trie_Searchにおいてreturn($index$)の実行の変更処理についてのみ示す．その他の処理は，関数Trie_Searchと同様である．

[関数 **Trie_Delete**(**D**(K),X)]

入力：キー集合Kに対するダブル配列D(K)，キーX.

[2]関数X_CHECKでは，集合{'a','b','c'}が空ノード17，18，19に遷移できることを発見し，そのBASE値15を返す



(a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-13	0	6	-17	2	1	0	6	0	-1	1	0	-20	0	-24	2	-29	-10	-22
CHECK	1	12	0	1	17	17	17	0	7	0	6	9	0	6	0	7	4	4	4	12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	e	l	o	r	#	$	?	?	?	s	#	$	$	?	?	?	y	#	$	#	$

	22	23	24	25	26	27	28	29	30	31
TAIL	#	$	e	s	s	#	$	q	#	$

(b)

図 4.8　図 4.5 のダブル配列にキー “bbq#” を追加した例

出力：キーを削除したときは $g(1, X\#) = s$ なるノード s．削除キーがなかった（検索に失敗した）ときは，0．

手順（D-3b）｛手順（D-3）の return($index$) の変更｝

```
begin
    W_BASE(index, 0);
    W_CHECK(index, 0);
    return(index);
end
```

（関数終）

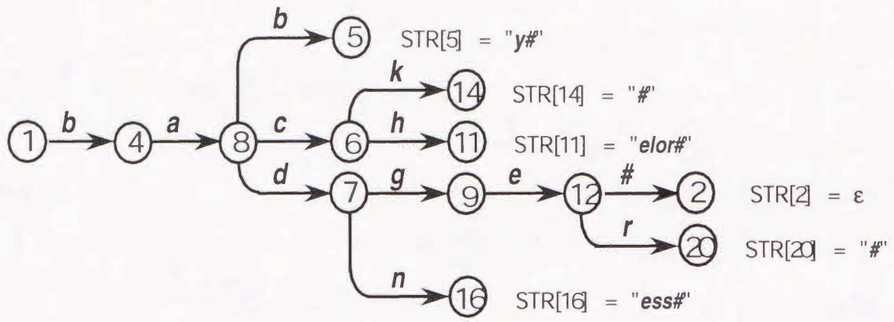
［例 4.5］

図 4.5 のダブル配列でのキー “bcs#” の削除を考える．まず最初に，手順（D-1）で検索に使用される変数 $index$ と pos にそれぞれ1，0を代入する．次に手順（D-2）で，pos に $pos+1 = 0+1 = 1$ を代入し，BASE[1]+3 = 4，CHECK[4] = 1 よりに Forward(1,‘b’) = 4 となり，これを t に代入する．t は0ではないので，$index = t$ となる．同様に，Forward(4,‘c’) = 10 となり，$pos = 2$ のとき，$index = 10$，BASE[10] = −10 < 0 であるのでノード10はセパレートノードとなる．手順（D-3）で TAIL 中の −BASE[10] = 10 の位置から，シングルストリング “s#” をとりだし，S_TEMP に格納する．キーの残りの文字列と S_TEMP は一致するので検索は成功となり，手順（D-3b）で，W_BASE(10,0)，W_CHECK(10,0) より，BASE[10] と CHECK[10] に0を代入し，$index = 5$ を返す．

図 4.5 のダブル配列からキー “bcs#” を削除した例を図 4.9 に示す．

（例終）

関数 Trie_Delete の動作は，トライ上のセパレートノードを削除しているだけである．このため，配列 TAIL 上の，削除したセパレートノードに対するシングルストリングを格納していた部分がごみとなってしまうが，ごみを排除するための再構成処理は削除時間に多くの影響を与えるので，削除アルゴリズムには含まれない．



(a)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-13	0	6	-17	2	1	2	6	0	-1	1	0	-20	0	-24	0	0	0	-22
CHECK	1	12	0	1	8	8	8	4	7	0	6	9	0	6	0	7	0	0	0	12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	e	l	o	r	#	$	?	?	?	?	?	?	$	?	?	?	y	#	$	#	$

	22	23	24	25	26	27	28
TAIL	#	$	e	s	s	#	$

(b)

図 4.9　図 4.5 のダブル配列からキー “bcs#” を削除した例

4.5　追加の高速化

ダブル配列は，トライを実現するデータ構造として，検索の高速性とコンパクト性を両立する優れた構造であるが，問題点もある．4.4.1 項で述べたように，追加アルゴリズムが複雑となるため追加時間に影響を及ぼすことである．

特に問題となるのが，関数 X_CHECK の動作である．関数 X_CHECK は最適なインデックスを探すためにダブル配列上のインデックスをシーケンシャルに調べている．このため，最悪の場合 $O(DA\_SIZE)$ の動作時間を必要とし，ダブル配列のサイズが大きくなると，この動作時間の問題は無視できないものとなる．

そこで，本節では関数 X_CHECK の動作を高速化することを中心に，追加時間の高速化を実現する手法を提案する．

4.5.1　範囲限定法

通常の関数 X_CHECK の動作では，空ノード（未使用ノード）を発見するためにダブル配列上のインデックスを全て調べるが，ダブル配列のインデックス全体における空ノードの割合は，非常に小さい（6 章で示す）．

また，関数 X_CHECK は最小のインデックスを返し，かつ確保された空ノードはその後すぐに利用されるので，空ノードはダブル配列全体の後方（インデックス番号の大きい方）に集中すると仮定できる．

そこで，関数 X_CHECK でのインデックス調査開始位置を後方にずらすことで，X_CHECK の動作時間の短縮を図るのが範囲限定法である．範囲限定法はダブル配列の要素に変更を加えることはないので，既にダブル配列によって構築された辞書に対しても適用できる利点を持つ．

範囲限定法では，インデックス調査開始位置をインデックス全体の割合を示す変数 *skip_rate*（< 1.0）を用いて表現し，関数 X_CHECK の動作は，(xx-1) 行を以下のように変更することで簡単に行える．

(xx-1)　　$q = DA\_SIZE * skip\_rate;$

ここで，変数 *skip_rate* の値を確定するために，簡単な実験を行った．図 4.10 は，変数 *skip_rate* の値を変えていったときの追加時間と，空ノードのインデックス全体に対する

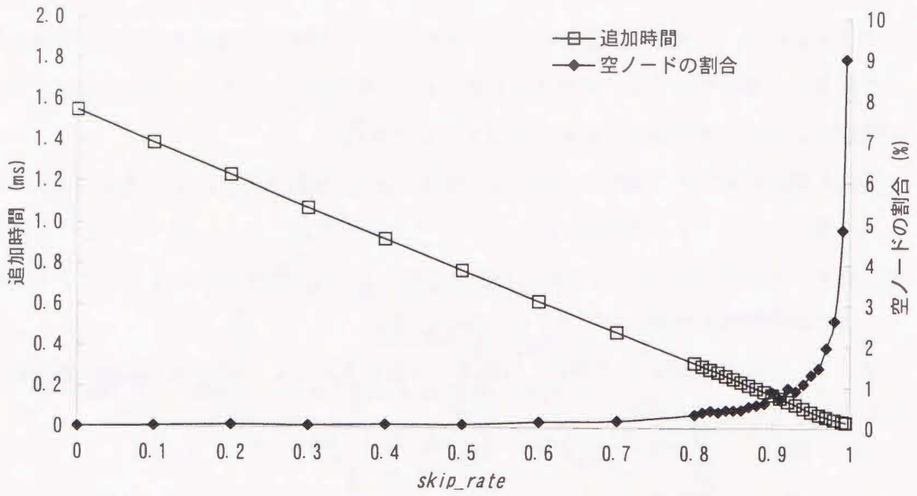


図 4.10　*skip_rate* の値による追加時間と空ノードの割合

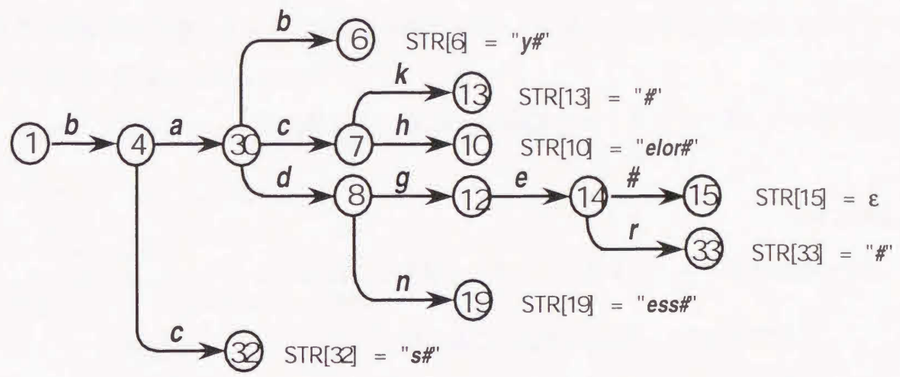
割合を示している．図を見てもわかるように，*skip_rate* の値が小さいと通常のX_CHEKの動作と同様になるので追加時間が大きくなり，1に近づくと空ノードの割合が大きくなるため，記憶容量に影響を与える．従って，図 4.10 の結果から，変数 *skip_rate* の値は *skip_rate* = 0.9 が最も妥当であるといえる．

［例 4.6］

範囲限定法を用いて，キー集合 $K2$ のキーを登録した場合のダブル配列の例をトライの例とともに図 4.11 に示す．但し，*skip_rate* = 0.9 を使用する．

（例終）

範囲限定法は，従来の追加アルゴリズムに非常に簡単な変更で，追加時間に十分な高速化をもたらす手法であるといえる．



(a) 範囲限定法を用いたトライ

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	0	0	28	0	-17	1	4	0	-1	0	8	-20	14	-13	0	0	0	-24	0
CHECK	1	0	0	1	0	30	30	30	0	7	0	8	7	12	14	0	0	0	8	0

	21	22	23	24	25	26	27	28	29	30	31	32	33
BASE	0	0	0	0	0	0	0	0	0	3	0	-10	-22
CHECK	0	0	0	0	0	0	0	0	0	4	0	4	14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	e	l	o	r	#	$	?	?	?	s	#	$	$	?	?	?	y	#	$	#	$

	22	23	24	25	26	27	28
TAIL	#	$	e	s	s	#	$

(b) 範囲限定法を用いたダブル配列

図 4.11　範囲限定法を用いたトライとダブル配列

4.5.2　空ノードリンク法

前項でも述べたように，ダブル配列のインデックス全体における空ノードの割合は非常に小さい．このため，関数X_CHECKにおいて調査対象が空ノードだけであれば，非常に高速に処理を行うことができる．

そこで，空ノードリンク法では空ノード同士をリンクで結び，X_CHECKにおいて空ノードをリンクでたどることによって，調査対象を最小限に抑えることで追加時間の高速化を図る．

まず始めに，空ノードをリンクで結ぶために以下を定義する．

[定義 4.3]

ダブル配列の空ノード（未使用ノード）の番号を，昇順に$r_1, r_2, \ldots, r_m$とするとき，

$$\mathrm{CHECK}[r_i] = -r_{(i+1)} \quad (1 \leq i \leq m-1)$$

$$\mathrm{CHECK}[r_m] = -(DA\_SIZE + 1)$$

なるリンクを作成する．但し，r_1とr_mはそれぞれ$EMPTY_HEAD$と$EMPTY_TAIL$なる変数に格納される．これらを空ノードリンクと呼ぶ．空ノードのCHECK値を負数にするのは，通常のノードと区別するためである．

（定義終）

空ノードリンクの導入に伴い，ダブル配列の追加アルゴリズムにおいて，関数W_BASE，W_CHECK，X_CHECKが変更される．また，新たに関数SET_EMPTY_LINKが導入される．以下，それぞれの関数の定義を示す．

[関数 W_BASE(idx, val)]

入力：ノード番号idx，BASE[idx]に設定する値val.

出力：なし.

```
      begin
(wb-1)    BASE[idx] ← val;
(wb-2)    if idx > DA_SIZE then SET_EMPTY_LINK(idx);
```



```
      end;
```

（関数終）

関数W_BASEは，BASEの値を設定し，ダブル配列のサイズが大きくなったときの空ノードリンクの設定をする．

[関数 W_CHECK(idx, val)]

入力：ノード番号idx，CHECK[idx]に設定する値val.

出力：なし.

```
      begin
(wc-1)    if CHECK[idx] < 0 then
          begin
              prev_idx ← EMPTY_HEAD;
              while prev_idx < idx do
              begin
                  if −CHECK[prev_idx] < idx then
                      prev_idx ← −CHECK[prev_idx];
              end
              if idx = EMPTY_HEAD then
              begin
                  if EMPTY_HEAD = EMPTY_TAIL then
                      EMPTY_TAIL ← −CHECK[idx];
                  EMPTY_HEAD = −CHECK[idx];
              end
              else begin
                  CHECK[prev_idx] ← −CHECK[idx];
                  if EMPTY_TAIL = idx then EMPTY_TAIL ← prev_idx;
              end
      end
```

```
(wc-2)      if val = 0 then
            begin
                prev_idx ← EMPTY_HEAD;
                while prev_idx < idx do
                begin
                    if −CHECK[prev_idx] < idx then
                        prev_idx ← −CHECK[prev_idx];
                end
                if idx < EMPTY_HEAD then
                begin
                    CHECK[idx] ← −EMPTY_HEAD;
                    if EMPTY_HEAD = EMPTY_TAIL and
                    EMPTY_TAIL > DA_SIZE then
                        EMPTY_TAIL ← idx;
                    EMPTY_HEAD ← idx;
                end
                else begin
                    CHECK[idx] ←CHECK[prev_idx];
                    CHECK[prev_idx] ← −idx;
                    if EMPTY_TAIL = prev_idx then EMPTY_TAIL = idx;
                end
            end
(wc-3)      else CHECK[idx] ← val;
(wc-4)      if idx > DA_SIZE then SET_EMPTY_LINK(idx);
        end;
```

（関数終）

関数W_CHECKは，空ノードリンクの設定も兼ねるので処理がやや複雑であるが，大まかに分けると2つの動作をする．(wc-1)行のif文からは，CHECK[idx]が空ノードだった場合に，値valを設定する前にCHECK[idx]を空ノードリンクから削除する処理を行っ

ている．(wc-2)行のif文からは，CHECK[idx]が空ノードになる場合，CHECK[idx]を空ノードリンクに挿入する処理を行っている．また，(wc-4)行は，関数W_BASEと同様に，ダブル配列のサイズが大きくなったときの処理を行っている．

[関数 SET_EMPTY_LINK(idx)]

入力：ノード番号idx.

出力：なし.

```
            begin
                start_idx ← −CHECK[EMPTY_TAIL];
                DA_SIZE ← idx;
                end_idx ← idx − 1;
(l-1)           if start_idx > end_idx then
                begin
                    if start_idx = EMPTY_TAIL then
                    begin
                        EMPTY_HEAD ← start_idx + 1;
                        EMPTY_TAIL ← start_idx + 1;
                    end
                    else CHECK[EMPTY_TAIL] ← −(start_idx + 1);
                    return;
                end
                EMPTY_TAIL ← end_idx;
(l-2)           while start_idx < end_idx do
                begin
                    BASE[start_idx] ← 0;
                    CHECK[start_idx] ← −(start_idx + 1);
                    start_idx ← start_idx + 1;
                end
                BASE[start_idx] ← 0;
```

					9	10
BASE	…	0	…		0	3
CHECK	…	−9	…		−11	7

DA_SIZE=10
EMPTY_TAIL=9

(a)

					9	10	11	12	13	14
BASE	…	0	…		0	3	0	0	0	−22
CHECK	…	−9	…		−11	7	−12	−13	−15	10

DA_SIZE=14
EMPTY_TAIL=13

(b)

図 4.12　関数SET_EMPTY_LINK の動作

$CHECK[start\_idx] \leftarrow -(DA\_SIZE+1)$;

end;

（関数終）

関数SET_EMPTY_LINK は，W_BASE，W_CHECK によって，ダブル配列のサイズが大きくなったとき，新しくできた空ノードを空ノードリンクに登録する処理をする．たとえば，図 4.12(a) の状態から，W_CHECK(14,10) などが呼ばれたとき，新たにできる空ノード 11，12，13 を空ノードリンクに登録し，図 4.12(b) のような状態にする．

［関数 **X_CHECK**(*LIST*)］

入力：文字集合 *LIST*．

出力：*LIST* に含まれる文字 c 全てが CHECK$[q+\mathrm{N}[c]]=0$ を満足する最小のインデックス q．但し，$q>1$．

LIST に含まれる文字 c のうち，最小のものを c_1 とする．

```
      begin
(x-1)     q ← EMPTY_HEAD;
(x-2)     while q ≤ DA_SIZE do
          begin
(x-3)         qq ← q − c1;
(x-4)         flg ← true;
(x-5)         for each c in LIST do
              begin
(x-6)             if CHECK[qq + N[c]] > 0 then flg ← false;
              end;
(x-7)         if flg = true then return (qq);
(x-8)         q ← −CHECK[q];
          end;
(x-9)     return (q − c1);
      end;
```

（関数終）

変数 q は，(x-1) 行で *EMPTY_HEAD* に設定され，(x-8) 行で次の空ノードをたどっていることから，常に空ノードをさすことになるので，CHECK$[q]=$CHECK$[qq+c_1]$ なる変数 qq が戻り値の候補となる．そこで，(x-5) 行での for 文に入る前に，(x-3) 行で変数 qq の設定をしている．

［例 **4.7**］

空ノードリンク法を用いて，キー集合 $K2$ のキーを登録した場合のダブル配列の例を図 4.13 に示す．図 4.5(b) との違いは，空ノードの CHECK 値が負数になっているところのみである．トライの例は図 4.5(a) と同様である．

（例終）

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-13	0	6	-17	2	1	2	6	-10	-1	1	0	-20	0	-24	0	0	0	-22
CHECK	1	12	-13	1	8	8	8	4	7	4	6	9	-15	6	-17	7	-18	-19	-21	12

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
TAIL	e	l	o	r	#	$	?	?	?	s	#	$	$	?	?	?	y	#	$	#	$

	22	23	24	25	26	27	28
TAIL	#	$	e	s	s	#	$

図 4.13　空ノードリンク法を用いたダブル配列

空ノードリンク法は，追加アルゴリズムを更に複雑なものにするが，追加時間に劇的な高速化をもたらす手法であるといえる．

4.6　結言

本章では，トライを実現するデータ構造として，青江 [24] の提案したダブル配列について説明した．また，検索，更新アルゴリズムを詳細に述べるとともに，ダブル配列法の問題点であるキー追加の速度を改善する手法として，ダブル配列で構築された辞書を変更することなく高速化する範囲限定法法と，空ノードに格納したリンク情報によって高速化を実現する空ノードリンク法を提案した．これらの手法に対する評価は，6 章で行う．

次章では，トライとそのデータ構造であるダブル配列を用いて共起情報を効率的に記憶検索する手法について説明する．

第5章

リンクトライ

5.1　緒言

本章では，共起情報を効率的に記憶検索する手法として，関連研究であるダブルトライについて簡単に述べ，提案手法であるリンクトライについて概要と共起情報の検索，更新アルゴリズムを説明する．また，ダブル配列を用いたリンクトライのデータ構造についても説明する．

リンクトライのデータ構造ではその構成要素を全てダブル配列で表現し，ダブル配列同士を連結する手法を述べる．また，リンクトライでは複数のダブル配列を利用することになるので，管理の効率化のために複数のダブル配列を 1 つに統合する手法も述べる．

5.2　共起情報の格納構造

5.2.1　ダブルトライ

2 章でも述べたが，一般に，共起情報の格納方式は基本単語 X と Y に対する連鎖語彙 XY を構成し，この連鎖語彙をキーとして，関係情報をレコード情報として格納する．しかし，この格納方式をトライに適用すると，連鎖語彙の作成に伴って平均語彙長が増加するため，共通接頭辞を圧縮できても効率的に記憶されるとはいえない．この問題に関する研究として，森本ら [9] によりダブルトライが提案されている．

ダブルトライは，二つのトライを使用して前半と後半のキーの部分的な共有化を実現す

る．この方法では登録キーを他のキーと区別できる接頭辞で区切り，これを最初のトライ（左トライと呼ぶ）に格納する．残りの接尾辞は逆向きに，即ち，キーの最後の文字がトライの根となるようにもう一つのトライ（右トライと呼ぶ）へ格納する．そして，接尾辞と接頭辞を関係づけるアーク（リンクアークと呼ぶ）を構成する．また，レコード情報は左トライの葉から格納する．

［例 **5.1**］

図 5.1 にキー集合

$K3$ = {“アメリカ#”,“カナダ#”,“カッター#”,“カッターシャツ#”,“衣類#”,“生地#”,“気候#”,“合衆国#”,“合う#”,“国名#”,“国籍#”}

に対するダブルトライの例を示す．左トライのノード4から右トライのノード6への破線アークが，“カナダ#” を左右のトライに対して定義し，左トライのノード4から関係情報のレコードが格納される [9]．このキー “カナダ#” の検索は，$g(1,\text{‘カ’})=3$，$g(3,\text{‘ナ’})=4$ と左トライを辿り，破線アークを辿って右トライを $g^{-1}(6,\text{‘ダ’})=2$，$g^{-1}(2,\text{‘\#’})=1$ と辿ることで検索できる．

（例終）

例 5.1 で示したように，ダブルトライはレコード情報を一意に検索できるトライの特性を生かし，かつ接尾辞も可能な限り併合圧縮する手法であり，一般的なキーを対象としているため，膨大な共起情報の格納には通常のトライ同様，効率的ではない．また，ダブルトライの更新は，例えば例 5.1 において “アフリカ#” を追加すると，右トライのノード4から5へのアークを左トライに移動しなければならないなど，左右トライのアークの部分的な移動，追加，削除が生じるので，取り扱いは複雑になる．但し，キーの判定とそのレコード情報の検索は，トライのアークの探索時間計算量を $O(1)$ で行えるならば，トライに格納される語彙数に関係なく，キーの長さに比例した計算量で実行できる特徴を有する．

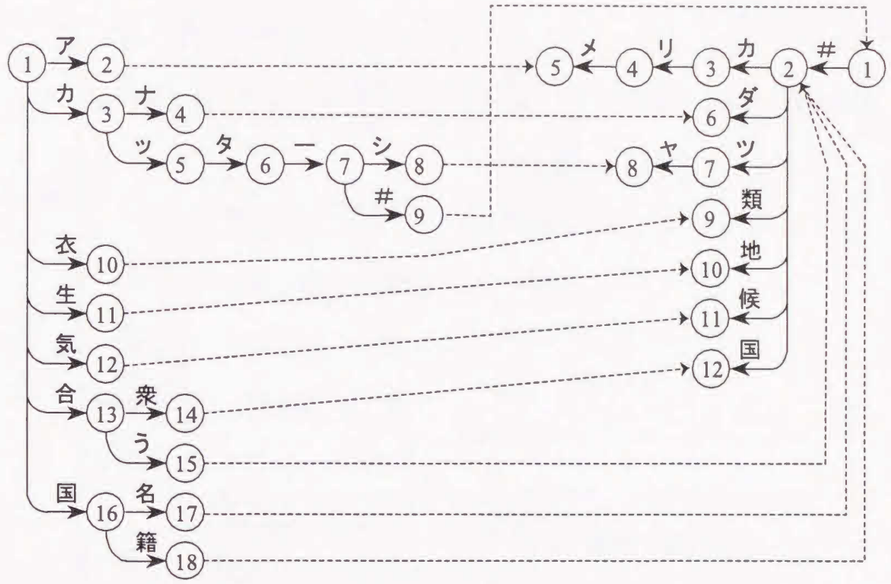


図 5.1　$K3$ に対するダブルトライ

5.2.2　リンクトライ

リンクトライは，ダブルトライのように共起情報を XY の文字列として捉えるのではなく，X，Y の独立した単語間の関係を，X，Y を完全に共有化した構造で検索できる手法である．

リンクトライでは，(X,Y,α) に対して，$X\#$，$Y\#$ を一つのトライへ格納し，2項関係を定義するために，トライの葉ノード間にリンクを作成する．葉ノード s から出るリンク情報は関数 $f(s)$ で定義され，$f(s)$ が要素 t を含むとき，葉ノード s から葉ノード t へのリンクが存在し，2項関係が定義されていることを意味する．

トライにアーク $g(1,X\#)=s$ なる s が存在するとき，葉ノード s は X と1対1に対応するので，キー X に関するレコード情報は葉ノード番号 s に対応したレコードに格納できる．そこでリンクトライでは，$f(s) \ni t$ となり，ノード s から $g(1,Y\#)=t$ なる葉ノード t へのリンクが存在するとき，X，Y の関係情報 α をレコード情報の集合 $REC(s,t)$ の

要素として格納するものとする．

［例 5.2］

共起情報の集合

$$
\begin{aligned}
C1 = \{\ & (\text{“アメリカ”},\text{“合衆国”},\alpha_4),\\
& (\text{“アメリカ”},\text{“合衆国”},\alpha_5),\\
& (\text{“アメリカ”},\text{“国名”},\alpha_1),\\
& (\text{“カナダ”},\text{“国名”},\alpha_1),\\
& (\text{“カッターシャツ”},\text{“衣類”},\alpha_1),\\
& (\text{“カッター”},\text{“カッターシャツ”},\alpha_6),\\
& (\text{“衣類”},\text{“生地”},\alpha_3),\\
& (\text{“合う”},\text{“衣類”},\alpha_2),\\
& (\text{“合う”},\text{“気候”},\alpha_2),\\
& (\text{“国名”},\text{“気候”},\alpha_3),\\
& (\text{“国名”},\text{“国籍”},\alpha_4)\ \}
\end{aligned}
$$

に対するリンクトライの例を，トライ部を図 5.2 に，リンク関数を表 5.1 に示す．

（例終）

関数 $f(s)$ によって定義されるリンク情報は，表 5.1 で示したように X を基準にその全てを取得可能である．言い換えれば，(X,Y,α) の検索はもとより，X，Y に定義される全ての α の検索や，X との関係が定義される全ての Y の検索が可能となる．これは，基本単語と共起情報を共有化した構造を持つリンクトライの特徴であり，共起情報をより効率的に記憶する手段でもある．

次節以降では，このリンクトライについて詳しく述べる．

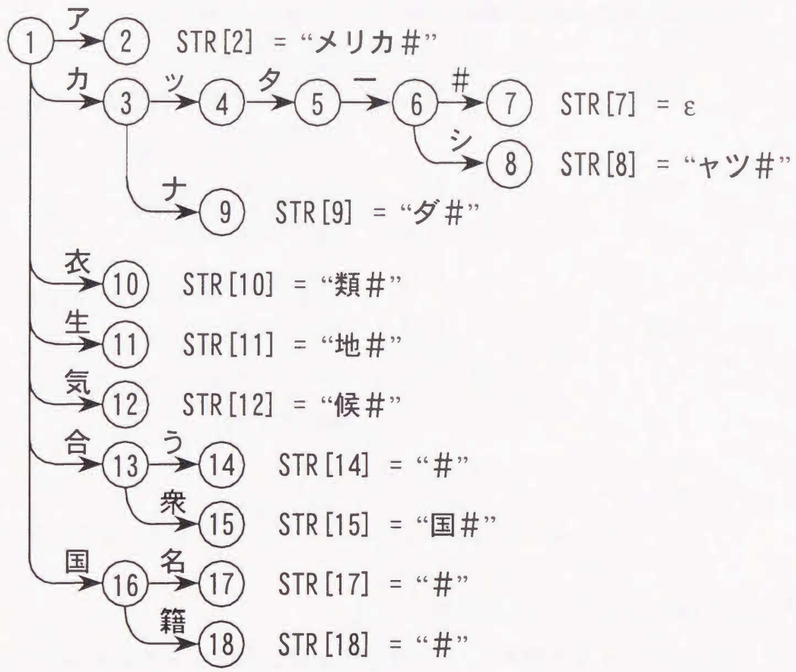


図 5.2　$C1$ に対するリンクトライのトライ部

5.3　リンクトライの検索アルゴリズム

リンクトライを用いて共起情報 (X,Y,α) を検索するアルゴリズムを示す．

［アルゴリズム LT_Search］

入力：X，Y，α．

出力：X，Y に α が定義されていれば，その α を含むレコード $REC(p,q)$ に対応するノード番号 p，q．即ち，$g(1,X\#) = p$，$g(1,Y\#) = q$．定義されていなければ，$p = 0$，$q = 0$ が返される．

表 5.1　$C1$ に対するリンクトライのリンク関数

X	s	$f(s)$	REC(s,t)
アメリカ	2	{15,17}	$REC(2,15)=\{\alpha_4,\alpha_5\}$,
			$REC(2,17)=\{\alpha_1\}$
カッター	7	{8}	$REC(7,8)=\{\alpha_6\}$
カッターシャツ	8	{10}	$REC(8,10)=\{\alpha_1\}$
カナダ	9	{17}	$REC(9,17)=\{\alpha_1\}$
衣類	10	{11}	$REC(10,11)=\{\alpha_3\}$
合う	14	{10,12}	$REC(14,10)=\{\alpha_2\}$,
			$REC(14,12)=\{\alpha_2\}$
国名	17	{12,18}	$REC(17,12)=\{\alpha_3\}$,
			$REC(17,18)=\{\alpha_4\}$

手順（S1-1）：|X，Y のトライ上の検索|

X，Y に対してトライ T を検索し，$s=$ Trie_Search(T,X) と $t=$ Trie_Search(T,Y) を得る．s，t のいずれかが $fail$ ならば，X，Y のいずれかがトライに格納されていないので，$p=0$，$q=0$ を出力し，アルゴリズムを終了する．そうでなければ，次へ進む．

手順（S1-2）：|関係定義の探索と出力処理|

$f(s)\ni t$ かつ $REC(s,t)\ni\alpha$ ならば $p=s$，$q=t$ を出力し，そうでなければ $p=0$，$q=0$ を出力し，アルゴリズムを終了する．

（アルゴリズム終）

[例 5.3]

図 5.2 と表 5.1 に示すリンクトライにおいて，共起情報（“アメリカ”,“合衆国”,α_5）を検索する例を示す．まず，手順（S1-1）で，Trie_Search(T,“アメリカ”) = 2，Trie_Search(T,“合衆国”) = 15 を得る．手順（S1-2）で，$f(2)\ni 15$ かつ $REC(2,15)\ni\alpha_5$ であるので，$p=2$，$q=15$ を出力する．

（例終）

次に，X に関連する全ての共起情報 (X,Y,α) を取得するアルゴリズムを示す．まず，トライの葉ノードから登録されているキーを取得する関数 Reverse_Out を与える．

[関数 Reverse_Out(*node*)]

入力：トライの葉ノード *node*.

出力：トライに登録されているキー Y.

手順（R1-1）：|トライ中の文字の取得|

現在のノードを示す変数 s を $s=node$ で初期化し，$s>1$ である間，$t=g^{-1}(s,a)$ なる文字 a を順次 Y に追加し，$s=t$ とする．

手順（R1-2）：|シングルストリングの取得|

Y 中の文字を逆順に並べ替え，シングルストリング STR[*node*] を Y に追加し，Y から端記号#を削除して Y を返す．

（関数終）

[例 5.4]

図 5.2 のトライに対して，葉ノード 8 からのキー取得を考える．まず手順（R1-1）で，$g^{-1}(8,\text{‘シ’})=6$ より文字‘シ’を得る．同様に，$g^{-1}(6,\text{‘ー’})=5$，$g^{-1}(5,\text{‘タ’})=4$，$g^{-1}(4,\text{‘ッ’})=3$，$g^{-1}(3,\text{‘カ’})=1$ とたどって文字を取得し，$Y=$ “シータッカ” となる．次に，手順（R1-2）で，Y の文字を逆順に並べ替えて $Y=$ “カッターシ” とし，シングルストリング STR[8] = “ャツ#” を追加して，$Y=$ “カッターシャツ#” とする．最後に端記号#を削除し，文字列 “カッターシャツ” を出力する．

（例終）

[アルゴリズム LT_GetAll]

入力：X.

出力：X との関係が定義されている全ての共起情報の集合 C．X が存在しない場合は，ϕ.

手順（G1-1）：|Xのトライ上の検索|

Xに対してトライTを検索し，s = Trie_Search(T,X)を得る．sが$fail$ならば，Xがトライに格納されていないので，$C = \phi$を出力し，アルゴリズムを終了する．そうでなければ，次へ進む．

手順（G1-2）：|関係定義の探索と出力処理|

$f(s)$にある全てのtに対して，Y = Reverse_Out(t)を得，$REC(s,t)$から全てのαを取り出し，$C \cup (X,Y,\alpha)$とする．最後にCを出力し，アルゴリズムを終了する．

（アルゴリズム終）

［例 5.5］

図5.2と表5.1に示すリンクトライにおいて，キー“合う”に関連する共起情報を取得する例を示す．まず，手順（G1-1）で，Trie_Search(T,“合う”) = 14を得る．手順（G1-2）で，$f(14) = \{10,12\}$であるので，ノード10に対して，Reverse_Out(10) = “衣類”，$REC(14,10) = \alpha_2$を得，ノード12に対して，Reverse_Out(12) = “気候”，$REC(14,12) = \alpha_2$を得る．従って，共起情報の集合{(“合う”,“衣類”,α_2),(“合う”,“気候”α_2)}を出力する．

（例終）

5.4　リンクトライの更新アルゴリズム

5.4.1　追加アルゴリズム

リンクトライに対するキーの追加アルゴリズムを示す．

［アルゴリズム LT_Insert］

入力：リンクトライに登録されていない(X,Y,α)．

出力：なし．

手順（A-1）：|キーX,Yの登録|

s = Trie_Insert(T,X)，t = Trie_Insert(T,Y)より，キーX,Yをトライに登録し，葉ノードs,tを得る．

手順（A-2）：|関係の定義|

$f(s) \ni t$ならば，$REC(s,t)$にαを追加し，$f(s) \not\ni t$ならば，関係は未定義なので，$f(s)$にtを加えてから，$REC(s,t)$にαを追加する．

（アルゴリズム終）

［例 5.6］

図5.2と表5.1に示すリンクトライにおいて，共起情報(“カッター”,“衣類”,α_1)の追加を考える．まず，手順（A-1）でTrie_Insert(T,“カッター”) = 7，Trie_Insert(T,“衣類”) = 10となり，手順（A-2）で$f(7)$には10が含まれていないので，$f(7)$に10を加え，$REC(7,10)$にα_1を加える．

（例終）

5.4.2　削除アルゴリズム

リンクトライに対するキーの削除アルゴリズムを示す．

［アルゴリズム LT_Delete］

入力：リンクトライに登録されている(X,Y,α)．

出力：なし．

手順（D-1）：|キーX,Yの検索|

X，Yに対してトライTを検索し，s = Trie_Search(T,X)とt = Trie_Search(T,Y)を得る．

手順（D-2）：|関係の削除|

sとtのどちらかが$fail$ならば，共起情報は未定義なので終了する．どちらも$fail$でなければ$REC(s,t)$からαを削除する．$REC(s,t) = \phi$となれば，$f(s)$からtを削除する．

手順（D-3）：|キーの削除|

$f(s)=\phi$ となれば，全ての関係がなくなるので，Trie_Delete(T,X) により，キー X を削除する．

（アルゴリズム終）

［例 5.7］

図 5.2 と表 5.1 に示すリンクトライにおいて，共起情報（“アメリカ”,“国名”,α_1）の削除を考える．まず手順（D-1）で $s=$ Trie_Search(T,“アメリカ”) $=2$，$t=$ Trie_Search(T,“国名”) $=17$ を得る．手順（D-2）で，s，t ともに $fail$ ではないので，$REC(2,17)$ から α_1 を削除する．$REC(2,17)=\phi$ となるので，$f(2)$ から 17 を削除し，$f(2)=\{15\}$ となる．手順（D-3）で，$f(2)\neq\phi$ なので何もしない．

（例終）

5.5　リンクトライのデータ構造

リンクトライは，キー集合 K を格納するトライ部，葉ノード s に対するリンク情報を格納する関数 $f(s)$，及びキー情報と共起情報を格納するレコード情報 $REC(s,t)$ から構成されるので，これらの個々のデータ構造の効率化を満足した上で，リンクトライの全体効率が低下しない相互構造の連結手法が必要である．

5.5.1　トライ部のデータ構造

トライ部のデータ構造は，4 章で述べた検索の高速性とコンパクト性を満足するダブル配列を使用する．

［例 5.8］

図 5.2 のトライをダブル配列で表したものを，入力文字とその内部表現値とともに図 5.3 に示す．但し，以後の拡張のために入力文字に含まれない文字も含む．

（例終）

文字	#	0	1	2	3	4	5	6	7	8	9	ー	う	ア	カ	シ	タ	ダ	ツ	ッ
内部表現値	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

文字	ナ	メ	ャ	リ	衣	気	候	合	国	衆	生	籍	地	名	類
内部表現値	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	-33	0	0	0	0	0	0	0	0	0	0	1	-39	-1	1	-21	1	0	0
CHECK	1	13	0	0	0	0	0	0	0	0	0	0	18	29	1	1	13	21	0	0

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
BASE	1	-16	0	0	0	-29	-42	0	1	1	-7	-35	-46	0	-12
CHECK	16	16	0	0	0	1	1	0	1	1	29	1	30	0	30

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TAIL	メ	リ	カ	#	$	?	国	#	$	?	?	#	$	?	?	ダ	#	$	?	?

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41
TAIL	ャ	ツ	#	$	?	?	?	?	類	#	$	?	$	?	地	#	$	?	#	$	?

	42	43	44	45	46	47
TAIL	候	#	$	?	#	$

図 5.3　図 5.2 に対するダブル配列

5.5.2　リンク関数 $f(s)$ のデータ構造

$f(s)$ の検索は，ノード番号集合から指定されたキー番号を探索する問題であるので，種々のキー検索技法が適用できる．一つの方法として，番号集合をソートして配列に格納し，2 分探索を適用するならば，配列の長さ n に対して検索時間計算量は $O(\log_2 n)$ となる．しかし，本手法では高速検索の実現を目標にしているので，ここにも $f(s)$ に対する個別のダブル配列 D($f(s)$) を次のように導入する．

$f(s)$ の要素ノード番号 t の内部コード列 w と $REC(s,t)\ni\alpha$ なる α の内部コード列 v に対し，vw なるコード列をトライのアークとするダブル配列 D($f(s)$) を構成する．但し，vw から v，w を一意に決定するために，v，w のそれぞれの列長は固定する．

［例 5.9］

表 5.1 において，キーが “国名” の場合の $f(17)$ に対するダブル配列 D($f(17)$) の例を図 5.4 に示す．ここでは簡単のために，$f(17)$ の要素を数値文字列として表現し，各関係情

	1	2	3	4	5	6	7
F_BASE	1	0	0	0	0	-1	-5
F_CHECK	1	0	0	0	0	1	1

	1	2	3	4	5	6	7	8
F_TAIL	1	2	#	$	1	8	#	$

図 5.4　ダブル配列 D($f(17)$) の例

報 α_i に対しては i を使用する．コード列長はそれぞれ2，1とし，内部表現値は図 5.3 のものを使用する．また，D(K) と区別するためにF_BASE，F_CHECK，F_TAILとしている．たとえば $f(17)$ の要素18は，$REC(17,18) \ni \alpha_4$ より，v =“4”，w =“18”，vw =“418” としてD($f(17)$) に格納される．このダブル配列の検索は，Trie_Search(D($f(17)$),“418”) = 7 のように，関数 Trie_Search で検索できる．

（例終）

5.5.3　トライ部とリンク関数 $f(s)$，$REC(s,t)$ の連結と更新処理

静的な D(K) では，D(K) のインデックス番号 s を手掛かりに D($f(s)$) がアクセスできるが，動的な D(K) では s は更新される可能性があるので，インデックス s は使用できない．解決法として，D($f(s)$) を格納する領域へのポインタ p を利用して，s からポインタ p を得るためのポインタ表 PTR[s] = p を構成すれば，トライ部の D(K) とリンク関数の D($f(s)$) の連結は成功する．しかし，トライの葉ノード s が変更される場合，s の値は D(K) の最大インデックス番号までの任意の値を取るので，最大インデックス番号に対するポインタ表が必要となり，余分な記憶領域や管理処理が必要となる．

更に，D(K) のインデックス番号の変更は $f(s)$ の要素 t にも関係するので，s，t に関係するポインタ表の更新だけでなく，t を格納する全ての D($f(s)$) の更新を余儀なくされ，リンクトライ全体の更新効率を低下させる深刻な問題となる．

しかし，$f(s)$ を定義するノード s とその要素 t は，全てトライの葉ノードであり，しか

も各葉ノードはキーと1対1に対応するので，葉ノード番号の代りにキーに対するユニークな識別番号を $f(s)$ の構成に利用すれば，上記の問題は解決する．不変な識別番号として，最も適切なものはキー X のレコード情報の実体へのポインタ値 r であり，これは必然的にトライの葉ノード s から $f(s)$ の実体へのポインタ値 r と考えても同じ意味である．しかし，実体のデータ領域をアクセスするポインタ値 r も更新されるので，キー X の識別番号 p とこのポインタ値 r を格納する表 KEYID[p] = r を構成し，表 KEYID のインデックス番号 p をキーの識別子として定義する．表 KEYID は，上記の表 PTR のように最大のインデックス番号に対応した大きさにならず，キーの数に比例した大きさで管理できる．

表 KEYID の導入により，キー X を識別番号 p で表現することで，リンクトライの更新効率を確保できるようになったが，現在のままだと問題が残る．アルゴリズム LT_Search はキー X,Y を検索後，α の確認を行うだけなので，キー X,Y の識別番号 p,q の使用による問題はないが，アルゴリズム LT_GetAll では，関数 Reverse_Out によってキー Y を取得するので，識別番号 q がキー Y の位置を保存しておかなければキー Y を取得することができない．そこで，KEYID[p] には，関数 $f(s)$ へのポインタ値 r のほかに，キー X のセパレートノード番号 s も格納する．

以上の議論の確認として，以下を定義する．

[定義 5.1]

キー X の識別番号 p に対し，トライ部とリンク関数を連結する表 KEYID を定義する．KEYID[p] は，キー X のセパレートノード番号 s と，関数 $f(s)$ へのポインタ値 r の2要素を格納するものとし，

$$\text{KEYID}[p] = \{s, r\}$$

と記述する．

（定義終）

ダブル配列にこの識別番号 p を格納する場合，配列 TAIL のシングルストリングの直後に格納すれば，キーが発見されると同時に表 KEYID を経由して $f(s)$ のポインタも知ることができる．ここで，ダブル配列のセパレートノードと配列 TAIL の連結に注目すると，キー削除などで発生した TAIL 上のごみの再構成処理を行わない限り，1度登録さ

れたキーに対するTAILの位置は変更されることはない．つまり，このTAIL位置（セパレートノードsに対する$-$BASE[s]）はキーに対するユニークな識別番号となり，シングルストリングの直後に要素$\{s,r\}$を格納すれば，表KEYIDを配列TAILに吸収させることができ，余分な経由も排除できる．

説明を簡単にするために，$-$BASE[s] $= p$なる識別番号pでのリンク関数$f(s)$のダブル配列をKEYID[p] $= \{s,r\}$なるポインタ値rに対して，DF[r]で参照できるものとし，DF[r]での$-$F_BASE[t] $= u$なるポインタ値uに対するレコード情報を$REC(r,u)$と定義する．

［例 5.10］

図5.5に図5.2と表5.1に対するKEYIDとDF[r]の例を示す．図5.5では簡単のために，図5.4と同様のコード列を用いている．また，KEYIDのインデックスpは簡単のため，セパレートノードに対応している．

（例終）

5.5.4　複数ダブル配列の統合

前項までで述べたように，リンクトライはトライ部とリンク関数で構成されており，これらは全てダブル配列で実現されている．しかし，リンク関数は登録されている共起情報の数に依存するため，登録する共起情報が増えれば，管理しなければならないダブル配列の数も増加する．

また，ひとつのリンク関数に登録されるキー数は，トライ部のそれに比べてずいぶん少ない．このため，ひとつのリンク関数を構成するダブル配列DF[r]の空ノードの割合が多くなり（6章で示す），リンクトライの容量全体にも影響を及ぼす．

ここで，トライ部とリンク関数の連結について再度考える．トライ部とリンク関数がともにダブル配列で構成されるということは，図5.6に示すように，表KEYIDを経由するものの，トライ部のセパレートノードから，その子ノードとしてリンク関数のトライが接続されているということになる．これは，ひとつのダブル配列でリンクトライを構成することが可能であることを示しており，実現すれば管理，容量の問題を一気に解決できる．

KEYID	2	17	9	8	10	7	11	14	12	18
ノード	15	35	22	17	26	2	32	14	27	33
DF[r]	1	2	3	4	5	6	0	7	0	0

DF[1]

	1	2	3	4	5	6	7	8
F_BASE	1	0	0	−9	0	0	−1	−5
F_CHECK	1	0	0	1	0	0	1	1

	1	2	3	4	5	6	7	8	9	10	11	12
F_TAIL	1	5	#	$	1	5	#	$	1	7	#	$

DF[2]

	1	2	3	4	5	6	7
F_BASE	1	0	0	0	0	−1	−5
F_CHECK	1	0	0	0	0	1	1

	1	2	3	4	5	6	7	8
F_TAIL	1	2	#	$	1	8	#	$

DF[3]

	1	2	3	4
F_BASE	1	0	0	−1
F_CHECK	1	0	0	1

	1	2	3	4
F_TAIL	1	7	#	$

DF[4]

	1	2	3	4
F_BASE	1	0	0	−1
F_CHECK	1	0	0	1

	1	2	3	4
F_TAIL	1	0	#	$

DF[5]

	1	2	3	4	5	6
F_BASE	1	0	0	0	0	−1
F_CHECK	1	0	0	0	0	1

	1	2	3	4
F_TAIL	1	1	#	$

DF[6]

	1	2	3	4	5	6	7	8	9
F_BASE	1	0	0	0	0	0	0	0	−1
F_CHECK	1	0	0	0	0	0	0	0	1

	1	2	3	4
F_TAIL	0	8	#	$

DF[7]

	1	2	3	4	5	6	7	8
F_BASE	1	0	0	4	1	−1	0	−3
F_CHECK	1	0	0	5	1	4	0	4

	1	2	3	4
F_TAIL	#	$	#	$

図 5.5　ダブル配列DF[r]と表KEYIDの例

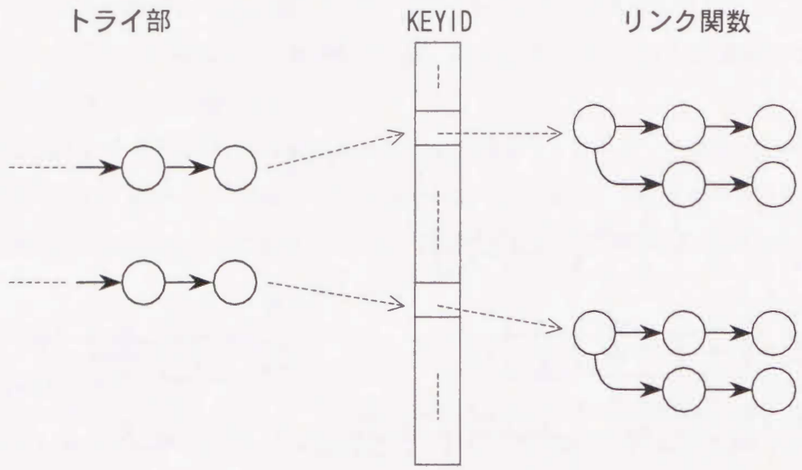


図 5.6　トライ部とリンク関数の連結

このリンク関数の統合は，KEYID[p] = $\{s, r\}$ において，DF[r] へのポインタ値 r を，ルートノードのノード番号とすることで，簡単に実現できるが，汎用的な拡張を目指して以下を定義する．

[定義 5.2]

ダブル配列のルートノード n_r に対し，識別番号 n_{ID} を定義し，これを

$$\mathrm{ROOTID}[n_{ID}] = n_r$$

として表 ROOTID で管理する．また，ルートノード n_r の CHECK 値には，

$$\mathrm{CHECK}[n_r] = -n_{ID}$$

のように識別番号 n_{ID} を格納する．

（定義終）

ルートノードの CHECK 値が負数となるのは他のノードと区別するためであるが，4.5.2 項で述べた空ノードリンク法も CHECK 値に負数を利用するため，更新処理において空ノードを判定するために BASE 値も利用する．

この表 ROOTID の導入によって，通常のダブル配列においても，複数のトライを 1 つのダブル配列上に構築することが可能となり，複数の辞書を多用する自然言語処理システムにおける辞書システムの管理効率を改善できる．

表 ROOTID の導入によるダブル配列のアルゴリズムの変更箇所を以下に示す．

[関数 Trie_Search の変更]

入力：ルートノード ID n_{ID} から開始されるダブル配列 D(n_{ID})，キー X．

出力：キー X（セパレートノード s）に対する KEYID $p = -$BASE[s]．検索に失敗したときは，0．

手順（D-1）：|変数の初期化|

　$index \leftarrow$ ROOTID[n_{ID}];

　$pos \leftarrow 0$;

（変更終）

この変更は，ルートノードが複数存在することにより，初期化時にルートノードを決定する必要があることによる．出力については関数 Trie_Insert，Trie_Delete についても同様に KEYID p となる．

[関数 A_INSERT の変更]

(a-8) 行の前に以下を挿入する．

(aa-1)　**else if** BASE[t] > 0 **then**
　　begin
(aa-2)　　$t' \leftarrow EMPTY\_HEAD$;
(aa-3)　　W_CHECK(t',CHECK[t]);
(aa-4)　　W_BASE(t',BASE[t]);

(aa-5)　　**for each** q **such that** CHECK[q] $= t$ **do**

　　W_CHECK(q, t');

(aa-6)　　**if** $t = index$ **then** $index \leftarrow t'$;

(aa-7)　　W_CHECK(t, 0);

(aa-8)　　W_BASE(t, 0);

(aa-9)　　ROOTID[n_{ID}] $\leftarrow t'$;

end;

(変更終)

この変更は，ルートノードを移動する必要があるときの更新処理である．(aa-5) 行から (aa-8) 行は，関数 MODIFY での (m-8) 行から (m-13) 行に相当するので，関数化することも可能である．

[関数 X_CHECK の変更]

(x-6) 行の変更

(x-6)　　**if** BASE[qq + N[c]] <> 0 **and** CHECK[qq + N[c]] > 0 **then** $flg \leftarrow false$;

(変更終)

[関数 W_CHECK の変更]

(wc-1) 行の変更

(wc-1)　　**if** BASE[idx] = 0 **and** CHECK[idx] < 0 **then**

(変更終)

これらの変更は空ノードの判定方法の変更による．

以上のデータ構造によるリンクトライのアルゴリズムを以下に示す．

[アルゴリズム LT_Search]

入力：X，Y，α．

出力：X，Y に α が定義されていれば，その α を含むレコード $REC(r, u)$ に対応するポインタ値 r，u．定義されていなければ，$r = 0$，$u = 0$ を返す．

トライ部のルートノード番号を ROOTID[1] = 1 とする．

手順（S2-1）：¦トライ部のダブル配列 D(1) における X，Y の検索¦

X，Y に対してダブル配列 D(1) を検索し，p = Trie_Search(D(1), X) と，q = Trie_Search (D(1), Y) を得る．p，q のいずれかが 0 ならば，X，Y のいずれかがトライに格納されていないので，$r = 0$，$u = 0$ を出力し，アルゴリズムを終了する．そうでなければ，次へ進む．

手順（S2-2）：¦リンク関数とレコード情報の検索¦

X の識別番号 p で指定されるリンク関数の KEYID[p] = $\{s, r\}$ なる r，キー Y における識別番号 q のバイトコード列 w，及び α のバイトコード列 v に対して u = Trie_Search (D(r),vw) を検索し，r，u を出力する．

(アルゴリズム終)

アルゴリズム LT_GetAll を定義するにあたって，次の関数を使用する．

Trie_GetAll(D(n_{ID})): ダブル配列 D(n_{ID}) に登録されているキー全ての集合を取得する．

　ダブル配列（トライ）は，ルートノードからの深さ優先探索によって全てのキーを取得することが可能である．

[アルゴリズム LT_GetAll]

入力：X．

出力：X との関係が定義されている全ての共起情報の集合 C．X が存在しない場合は，ϕ．

手順（G1-1）：¦X のトライ上の検索¦

X に対してダブル配列 D(1) を検索し，p = Trie_Search(D(1),X) を得る．p が 0 ならば，X がトライに格納されていないので，$C = \phi$ を出力し，アルゴリズムを終了する．そうでなければ，次へ進む．

手順（G1-2）：|関係定義の探索と出力処理|

X の識別番号 p で指定されるリンク関数の KEYID[p] = $\{s, r\}$ なる r から，C_{vw} = Trie_GetAll(D(r)) なるリンク関数集合 C_{vw} を取得し，C_{vw} 中の全ての要素 vw に対し，KEYID[w] = $\{t, rr\}$ なる t，Y = Reverse_Out(t) を得，$C \cup (X, Y, \alpha_v)$ とする．最後に C を出力し，アルゴリズムを終了する．

（アルゴリズム終）

［アルゴリズム LT_Insert］

入力：リンクトライに登録されていない (X, Y, α).

出力：なし．

手順（A-1）：|キー X, Y の登録|

p = Trie_Insert(D(1),X)，q = Trie_Insert(D(1),Y) より，キー X, Y をダブル配列に登録し，KEYID p, q を得る．

手順（A-2）：|関係の定義|

α のバイトコード列 v，q のバイトコード列 w，KEYID[p] = $\{s, r\}$ なる r に対し，Trie_Insert(D(r), vw) により vw を登録する．

（アルゴリズム終）

［アルゴリズム LT_Delete］

入力：リンクトライに登録されている (X, Y, α).

出力：なし．

手順（D-1）：|キー X, Y の検索|

X，Y に対してダブル配列 D(1) を検索し，s = Trie_Search(D(1),X) と t = Trie_Search(D(1),Y) を得る．

手順（D-2）：|関係の削除|

s と t のどちらかが 0 ならば，共起情報は未定義なので終了する．どちらも 0 でなければ α のバイトコード列 v，q のバイトコード列 w，KEYID[p] = $\{s, r\}$ なる r に対し，Trie_Delete(D(r), vw) により vw を削除する．

手順（D-3）：|キーの削除|

Trie_GetAll(D(r)) = ϕ となれば，全ての関係がなくなるので，Trie_Delete(D(1),X) により，キー X を削除する．

（アルゴリズム終）

［例 5.11］

共起情報集合 $C1$ をリンクトライに登録した例を図 5.7 に，図 5.7 をトライで表現した例を図 5.8 に示す．図 5.7 でのダブル配列は，図 5.3 と同じ内部表現値を使用し，空ノードリンク法，複数のダブル配列統合法が利用されている．また，配列 TAIL は通常 1 次元の配列であるが，簡単のため 2 次元の配列で表記し，KEYID，ROOTID は配列 TAIL に吸収されている．理解を深めるために，ダブル配列中のリンク関数に相当する要素は斜体で示している．また，図 5.8 でのノード番号は，図 5.7 のインデックス番号と一致させている．

（例終）

5.6　結言

本章では，共起情報を効率的に記憶検索する手法として，リンクトライを提案し，リンクトライを用いた共起情報の検索，更新アルゴリズムを説明した．また，ダブル配列を用いたリンクトライのデータ構造についても説明し，リンクトライでのダブル配列の管理を効率化するために複数のダブル配列を 1 つに統合する手法も述べ，これらについての検索，更新アルゴリズムを説明した．

次章では，リンクトライの有効性を確認するために理論的評価と実験による評価を行う．

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
BASE	1	*1*	*2*	*-6*	*-8*	*6*	*-3*	*-4*	*-11*	-12	*4*	*-13*	9	*12*	-2	1	*-15*	1	-16	*19*
CHECK	1	*-2*	*-7*	*2*	*3*	*-10*	*2*	*2*	*6*	13	*-12*	*11*	18	*-9*	1	1	*14*	21	29	*-16*

	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
BASE	1	*-7*	*31*	*33*	-10	-9	-18	0	6	3	0	-14	*-17*	*-19*	-21	-1	-5	*-20*	*-22*
CHECK	16	16	*20*	*-5*	13	1	1	-31	1	1	-40	1	*23*	*23*	30	29	30	*24*	*24*

TAIL	1	2	3	4	5	6
1	国	#	36	0		
2	メ	リ	カ	#	15	2
3	*0*	*1*	*#*			
4	*0*	*1*	*#*			
5	#	37	24			
6	*0*	*5*	*#*			
7	ダ	#	22	3		
8	*0*	*5*	*#*			
9	類	#	26	14		
10	ヤ	ツ	#	25	6	
11	*0*	*9*	*#*			
12	10	11				
13	*1*	*0*	*#*			
14	地	#	32	0		
15	*1*	*4*	*#*			
16	#	19	20			
17	*9*	*#*				
18	候	#	27	0		
19	*8*	*#*				
20	*1*	*8*	*#*			
21	#	35	0			
22	*2*	*1*	*#*			

図 5.7　*C*1 に対するリンクトライのダブル配列表現

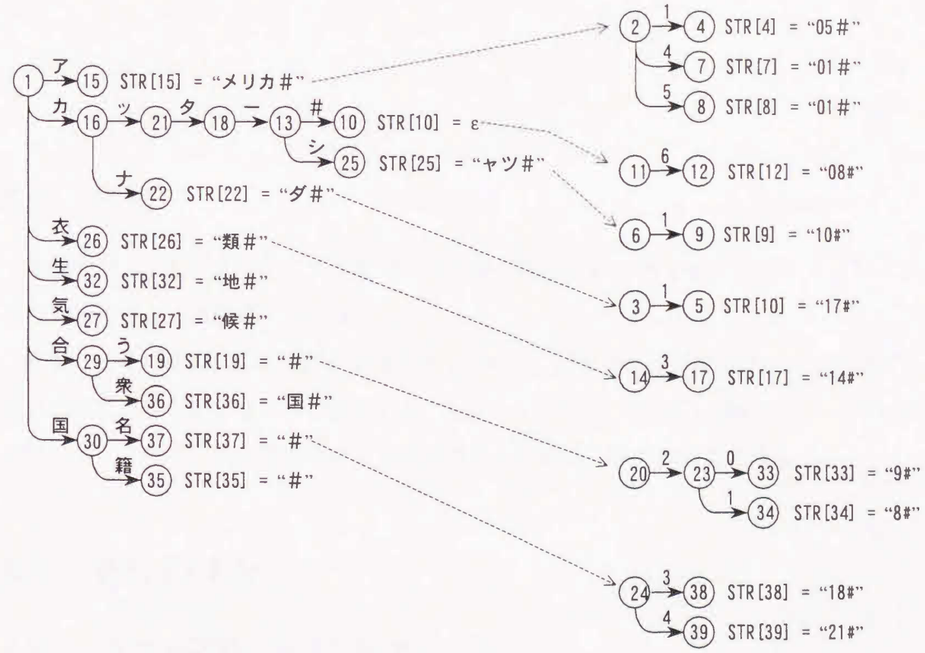


図 5.8　*C*1 に対するリンクトライのトライ表現

第6章

評価と考察

6.1 緒言

本章では，4 章で述べたダブル配列の追加の高速化法と 5 章で述べたリンクトライ法の理論的評価と，実験による評価を行う．

実験による評価では，本手法と比較を行うために，通常のダブル配列と，ダブル配列を用いた共起辞書を従来法として使用する．それぞれの手法において，検索に用いるデータのサイズと，様々な検索要求に対する検索速度を測定し，本手法の有効性を示す．

6.2 理論的評価

6.2.1 ダブル配列に対する評価

ダブル配列によるトライの遷移検索の最悪時間計算量は $O(1)$ であるので，検索キーの長さを k とするとダブル配列によるトライ検索の最悪時間計算量は $O(k)$ となる．

ダブル配列への追加の最悪時間計算量は，関数 MODIFY と X_CHECK に依存する．更に，空ノードリンク法を使用する場合は関数 W_CHECK にも依存する．評価するにあたって，トライの入力記号の最大数を e，ダブル配列の使用，未使用インデックス番号の数をそれぞれ n, m とする．

関数 W_CHECK については，$O(1)$ であるが，空ノードリンク法の場合は，空ノード

の位置を決定する必要があるため，最悪の場合全ての空ノードをたどる必要がある．このため $O(m)$ となる．

関数 MODIFY については，(m-3) 行と (m-9) 行で2重のループ構造となっており，共に最悪の場合 e 回繰り返す．また，共に関数 W_CHECK を持っているが，(m-9) 行のループ内での W_CHECK は既存の値を変更するだけであるので常に $O(1)$ である．従って，関数 MODIFY の計算量は $O(e^2)$，空ノードリンク法の場合は $O(m \cdot e^2)$ となる．

関数 X_CHECK についても，(xx-2),(x-2) 行と (xx-4),(x-5) 行で，2重ループ構造となっており，(xx-4),(x-5) 行では e 回繰り返す．(xx-2) 行では，ダブル配列のインデックス全てをたどるため $n+m$ 回繰り返すことになる．但し範囲限定法では全体の $(1-skip\_rate)$ の割合だけたどるので，$(1-skip\_rate)\cdot(n+m)$ 回繰り返す．空ノードリンク法の (x-2) 行では空ノードのみをたどるので，m 回繰り返す．よって関数 X_CHECK の計算量はそれぞれ，$O((n+m)\cdot e)$，$O((1-skip\_rate)\cdot(n+m)\cdot e)$，$O(m\cdot e)$ となる．

以上より，追加の最悪時間計算量は以下のようになる．

従来のダブル配列　：　$O((n+m)\cdot e+e^2)$

範囲限定法　：　$O((1-skip\_rate)\cdot(n+m)\cdot e+e^2)$

空ノードリンク法　：　$O((m\cdot e)\cdot(1+e))$

ここで，$skip\_rate=0.9$ とし，m の値が非常に小さい（次節で述べる）ことを考慮すると以下のように近似できる．

従来のダブル配列　：　$O(e\cdot(n+e))$

範囲限定法　：　$O(e\cdot(0.1\cdot n+e))$

空ノードリンク法　：　$O(e\cdot(1+e))$

これらより，範囲限定法，空ノードリンク法は大量のキー集合に対しても十分実用的な速度を実現できる．特に空ノードリンク法はダブル配列のサイズに依存しなくなるので非常に高速となる．

削除の最悪時間計算量は，セパレートノードを削除するだけであるので，関数W_CHECKにのみ依存する．従って，従来のダブル配列と範囲限定法は $O(1)$，空ノードリンク法は $O(m)$ となる．空ノードリンク法は空ノードの数に依存するので，大量のキー削除によって空ノードが膨大な数となったときは問題となるが，ダブル配列の応用において通常そのような削除はありえないので実用的な速度は確保できる．

6.2.2　リンクトライに対する評価

リンクトライの検索は，キー X,Y とリンク関数の検索時間の総和である．但し，トライ部とリンク関数の連結に表KEYIDを使用しているので，KEYIDの遷移時間 $O(1)$ も付加される．リンク関数には，α とキー Y の位置が格納されるので，これらのキーの長さをそれぞれ k_X,k_Y,k_{vw} とすると，検索の最悪時間計算量は $O(k_X+k_Y+k_{vw}+1)$ となる．

ここで，共起情報を従来のトライに登録することを考える．リンクトライでの検索アルゴリズム LT_Search と同等の検索を可能とするためには，X と Y の連鎖語彙 XY をキーとし，α をレコード情報としてトライに登録する必要がある．キー XY に対する α は複数存在する場合があるので，その数を r_n とするならば，レコード情報の取得時間は $O(r_n)$ となり，検索時間計算量は $O(k_X+k_Y+r_n)$ となる．

従って，リンクトライと従来のトライの検索時間の差は，$O(k_{vw}+1)$ と $O(r_n)$ の差ということになるが，これらはともにキー検索時間 $O(k_X+k_Y)$ より小さいので，実用上問題となることはない．

リンクトライの更新は，ダブル配列の更新とほぼ同様である．表KEYIDの更新処理が必要であるが，$O(1)$ で処理できるので無視できる．また，キー X,Y とリンク関数のそれぞれを処理することになるので，従来のトライの1回に対して処理は遅くなる．しかし，キーの平均語長が短いことにより1回のキー検索時間が短くなり，異なり語数が少ないので実際の更新回数が減少する．よって，実用上問題となることはない．

リンクトライの記憶量は従来のトライと同様，ダブル配列の記憶量に依存する．リンクトライはリンク関数もダブル配列に登録しなければならないが，従来のトライでは連鎖語彙 XY を登録するため，キーの平均語長が長くなり，ノード数が爆発的に増加するので，リンク関数のノード数を大幅に上回る．また，連鎖語彙 XY を登録する従来のトライでは，アルゴリズム LT_GetAll と同等の検索を実現できない．これは，キー X が登録されないことによるので，従来のトライでアルゴリズム LT_GetAll と同等の検索を実現するためには，X をキーに，Y,α をレコード情報として登録するトライを別に用意する必要がある．従って，従来は2つのトライが必要であったのに対し，リンクトライは1つです

むので記憶量は大幅に減少する．

自然言語処理システムでの辞書形態は，多くの項目が登録された基本辞書が利用されるので，更新速度よりは検索速度とコンパクト性が重要であり，この観点より，提案手法は有効であるといえる．

6.3　実験による評価

本手法の構成システムは約1,000行のC++言語で記述されており，DELL Precision 410（OS:WindowsNT4.0，CPU:pentiumII[400MHz]）上で稼動している．入力記号の総数eは，1バイトで表現できる数256に端記号を加えた257である．このため，日本語など2バイトで表現される文字からは連続する2つの遷移が構成される．また，リンクトライでのリンク関数は，αの内部表現値を2バイト，キーYへのKEYID番号を4バイト，計6バイトで構成されている．

6.3.1　ダブル配列に対する評価

ダブル配列に対しては，以下の評価を行った．

- 空ノードの割合の評価
- 範囲限定法，空ノードリンク法の評価

また，実験に用いたデータは，EDR電子化辞書[30]の英語単語辞書である．

空ノードの割合の評価

ダブル配列のインデックス数に対する空ノードの割合が非常に小さいことを確認するために次の実験を行った．

図6.1に，登録するキー数を変化させた場合の総インデックス数に対する空ノードの割合をインデックスの範囲ごとに調査した結果を示す．結果から，登録キー数が多くなるにつれて空ノードの割合が急激に減っていることがわかる．このことから，実用規模での辞書のキー数に対する空ノード数は無視できる程度になる．

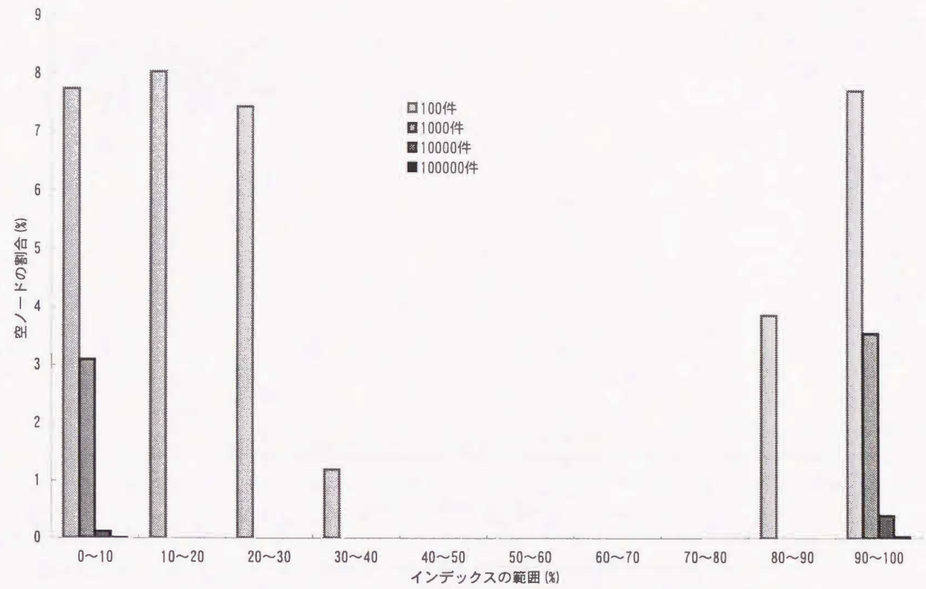


図6.1　キー追加時の空ノードの割合の結果

また，登録キー数が多くなるにつれて，空ノードがインデックスの前後に集中することがわかり，インデックスの後方の方がより集中している．このことから，4.5.1項で述べた範囲限定法において，空ノードはダブル配列全体の後方（インデックス番号の大きいほう）に集中するという仮定を確認する結果となった．

範囲限定法，空ノードリンク法の評価

範囲限定法，空ノードリンク法の有効性を確認するために，従来のダブル配列と比較して評価を行った．

範囲限定法は，関数X_CHECKでの調査範囲が限定されるため，調査範囲外に空ノードが存在する場合，ごみとして残ってしまい，記憶容量に影響を及ぼす．そこで，記憶容量の評価として登録キー数を変化させた場合の空ノードの割合を調査した結果を図6.2に示す．図6.1の結果と同様に，キー登録数が増えるに従って空ノードの割合が小さくなっ

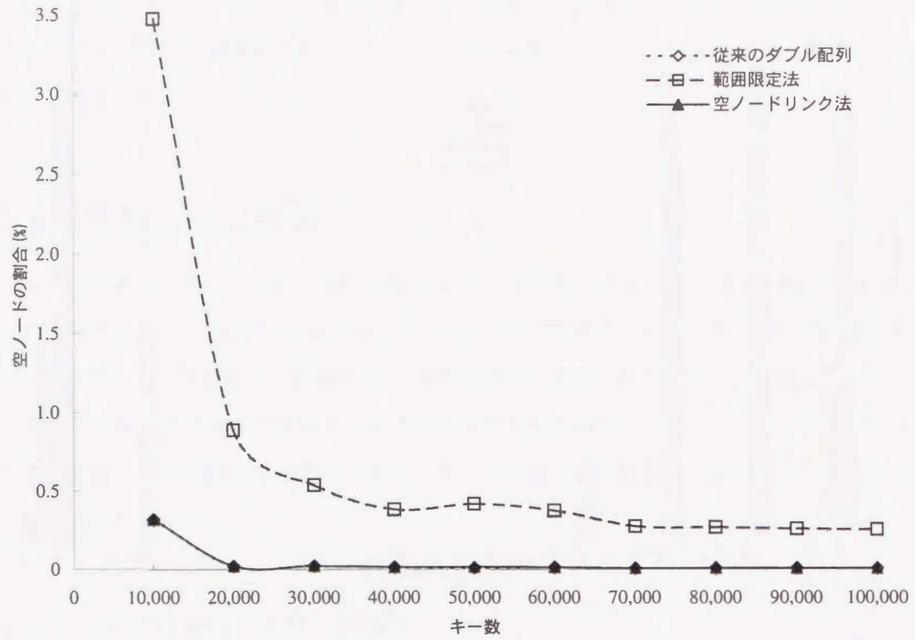


図 6.2　提案手法での空ノードの割合の結果

ている．範囲限定法は他の2つの手法に比べて空ノードの割合が多いが，それでも10万件登録時には0.2%ほどであるので，十分小さい値となることがわかる．

範囲限定法，空ノードリンク法はともにキー追加時間の高速化手法であり，これらの手法を導入することによって検索アルゴリズムに変更が加えられることはないので，ここでは検索時間の評価は行う必要はない．

図 6.3 に登録キー数を変化させた場合の追加時間の実験結果を示す．図中の追加時間は全キーの登録時間をキー数で割った，1件あたりの時間を示している．図 6.3 の結果から，範囲限定法は6〜16倍，空ノードリンク法は9〜320倍高速に追加できることがわかる．また，追加時間がダブル配列のインデックス数に依存する従来の手法と範囲限定法は，追加キー数が増加するにつれて追加時間も増加しているのに対し，ダブル配列のインデックス数に依存しない空ノードリンク法はほぼ一定の値を示していることもわかる．

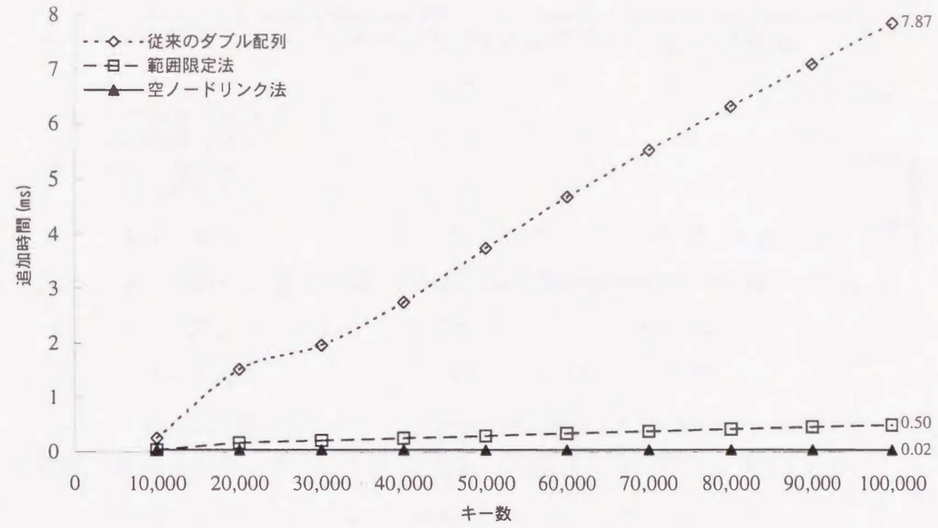


図 6.3　ダブル配列のキー追加時間の結果

図 6.4 に登録キー数を変化させた場合のキー登録後，100件のキーを削除した場合の削除時間の実験結果を示す．図中の削除時間は1件あたりの時間である．図 6.4 の結果から，登録キー数に関係なく一定の値を示していることがわかる．また，空ノードリンク法は削除時間が空ノードの数に依存するので他の手法に比べ削除時間が遅くなっているが，それでも1件あたり0.012msであり非常に高速であることがわかる．

6.3.2　リンクトライに対する評価

リンクトライの有効性を確認するため，対象手法として従来のトライに対して，連鎖語彙 XY をキーとしたものとの比較を行った．以後，簡単のため以下の表記を使用する．

LT ： リンクトライ

TXY ： 従来のトライにキーを連鎖語彙 XY，

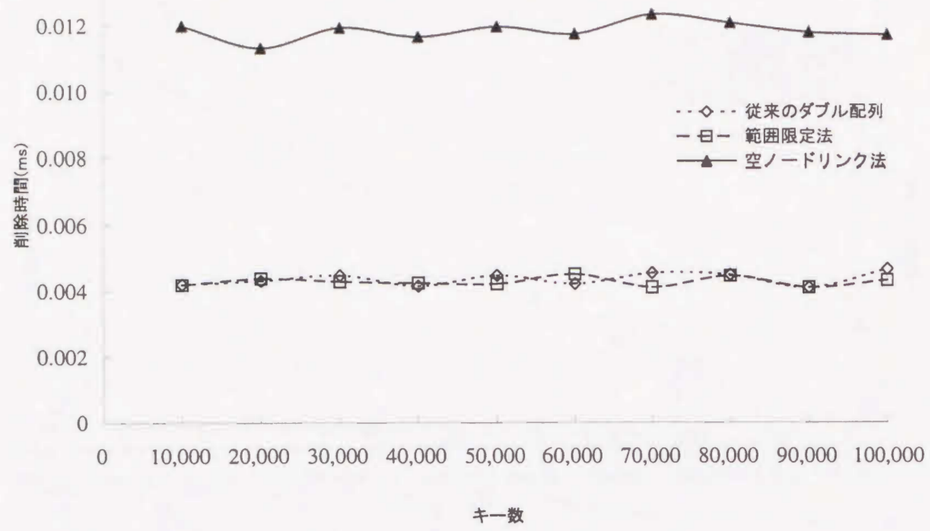


図 6.4　ダブル配列のキー削除時間の結果

レコード情報をαとして登録したもの

また，実験に使用したデータは，EDR 電子化辞書 [30] の日本語共起辞書，英語共起辞書であり，キー総数はそれぞれ，959,606 件と 475,149 件である．関係情報 α については，EDR での共起関係子にユニークな番号を割り当て，これを内部表現値とした．

表 6.1 に実験結果をを示す．ここで，平均長，最大長，異なり語数は，LT についてはキー X,Y，TXY については連鎖語彙 XY に対するものである．配列 TAIL については KEYID，レコード情報も含む．また，検索，追加時間は全件に対して検索，追加を，削除時間は 1,000 件の削除を行った場合の 1 件あたりに要する時間である．

リンクトライ LT のノード数は，従来のトライ TXY と比べて，トライ部に対するノード数が効率的に圧縮されていることがわかる．また，リンク関数に対するノード数を加えても，リンクトライのノード数は少なくなっており．本手法の有効性を確認できる．

検索時間については，従来手法とほぼ対等であるといえ，$O(k_{vw}+1)$ と $O(r_n)$ の誤差

表 6.1　リンクトライの実験結果

	日本語共起辞書		英語共起辞書	
	LT	TXY	LT	TXY
平均長 (byte)	8.2	9.7	10.4	12.8
最大長 (byte)	64	92	86	92
異なり語数	114,014	840,677	55,191	445,618
ノード数	187,605	4,240,913	167,534	3,013,437
ノード数 (リンク関数)	1,376,662		598,182	
空ノード数	70	86	1071	46
空ノードの割合 (%)	0.004	0.002	0.14	0.002
配列 TAIL	2,558,036	5,338,201	1,202,677	2,337,822
平均リンク数	9.5		11.7	
最大リンク数	3,654		2,924	
検索時間 (ms)	0.012	0.009	0.014	0.010
追加時間 (ms)	0.140	0.058	0.213	0.050
削除時間 (ms)	0.110	0.063	0.078	0.062

程度に収まっているといえる．更新時間については，リンクトライに 1 つの共起情報を登録する場合に 3 回ダブル配列の操作を行う必要があることから，他の手法に比べて遅くはなっているが，それでも追加時間で 1 件あたり 0.2ms 程度であり，非常に高速である．

記憶量に対する効果を確認するために，登録キー数を変化させた場合の記憶量の結果を図 6.5 に示す．図 6.5 から，リンクトライの記憶量は従来のトライに比べて 1/3 以上コンパクトになり，本手法の有効性がわかる．

6.4　リンクトライの応用

リンクトライは，共起情報の全てのデータをダブル配列（トライ）上に記憶するため，レコード情報には何も記憶されていない．更に，キー X,Y とリンク情報の全てがレコード情報を保持することが可能である．

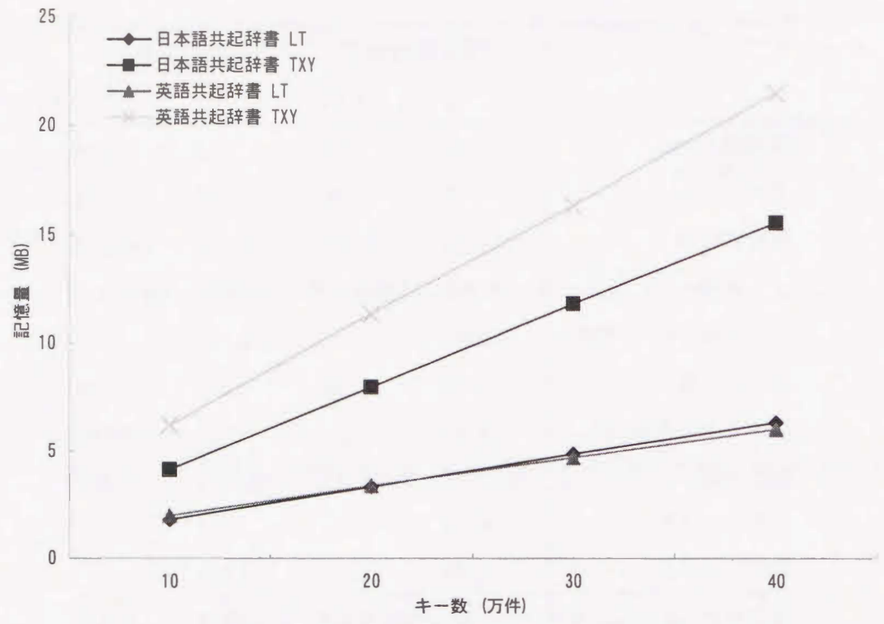


図 6.5　記憶量に対する結果

これを形態素解析などに応用すれば，共起辞書の他に形態素辞書もリンクトライに登録することができ，更に共起情報のキー X,Y は形態素辞書の形態素に相当するため，形態素辞書を登録しても必要な記憶量はレコード情報に対するものだけとなる．その他，必要な辞書を全て1つのリンクトライに登録することができ，記憶容量や辞書管理の面においても非常に効率的になるといえる．

また，リンクトライはその構造から，共起情報だけでなく，ある2つのデータ間に何らかの関係が定義されている情報であればどんなものでも格納できるので，その応用性は高い．

たとえば多言語情報処理の研究 [31, 32] では対訳辞書を多用するが，対訳辞書はレコード情報として訳語を格納するので，対応言語数が増えるとそれだけ対訳辞書も必要となり，検索要求に応じて適当な辞書を使用しなければならないので，処理が複雑になるが，

リンクトライに対訳データと言語情報を格納すれば複数の対訳辞書を統合することができるようになる．また，多言語情報検索の研究 [33] などでは，検索クエリを翻訳する際の訳語の選択に共起情報を利用するのでこれらもリンクトライに統合できる．

6.5　結言

本章では，4章で述べたダブル配列の追加の高速化法と5章で述べたリンクトライ法の理論的評価と，実験による評価を行い，その有効性を示した．

ダブル配列の追加時間の高速化手法では，空ノードリンク法が最大約320倍高速化され，劇的な改善が見られることがわかった．

リンクトライの評価では，検索の高速性は失われずに，記憶量の効率化が図れていることがわかった．

リンクトライは自然言語処理システムで使用される複数の辞書を統合化することが可能であり，統合されれば更に記憶量の効率化が図れると考えられる．

第7章

結　論

本論文では，自然言語処理システムにおける共起情報の重要性について述べ，従来の辞書システムにおいての共起情報の格納方式に対する問題点をあげた．そして，共起情報を効率的に記憶検索する手法を提案し，ダブル配列を用いたデータ構造と検索，更新アルゴリズムを提案した．また，提案したデータ構造の基礎構造であるダブル配列法について，その問題点であったキー追加の速度を高速化する手法を提案した．更に，複数の辞書を使用する辞書システムの管理効率を向上する手法を提案した．そして，提案手法の理論的評価と実験による評価で有効性を示した．

本手法は，共起情報の効率的な記憶検索を実現したものであるのと同時に，辞書システムにおいて関連する複数の辞書の統合化を実現できるものである．また，共起情報に限らず，2つのデータ間にある関係が定義できる情報であればなんでも登録できるので，その応用範囲も広く，実用性も高いといえる．

本論文では，提案手法のデータ構造とアルゴリズムは主記憶に記憶することを前提に議論してきたが，通常辞書は2次記憶に蓄えられるため，2次記憶に対するデータ構造が必要である．これはダブル配列を2次記憶に記憶させる問題となり，文献[24]によると，キー集合を分割することでダブル配列を複数のブロックに分割し（複数のトライを構成することになる），ブロックの選定にハッシュ法などを利用する．ブロック分割によってダブル配列の1要素サイズを小さくできる利点もある．しかし，この方法では複数のトライを1つのダブル配列で管理する提案手法の利点が活かされなくなるので，提案手法の利点を活かしたまま2次記憶に効率的に記憶する手法の考案が今後の課題となる．

謝辞

本研究の全課程を通じ，直接懇切なる御指導，御鞭撻を賜った徳島大学工学部知能情報工学教室 青江順一教授に心よりの感謝の意を表する．

本研究にあたって，御指導，御教示を賜った徳島大学工学部知能情報工学教室 矢野米雄教授，小野典彦教授に深く感謝する．

筑波大学大学院経営システム科学の 津田和彦助教授，徳島大学工学部知能情報工学教室の 獅々堀正幹講師，弘田正雄助手，技術研究組合新情報処理開発機構の 入口浩一博士，株式会社ナムコの 有田健博士，住友金属工業株式会社の 林淑隆博士，大阪府立工業高等専門学校電子情報工学科の 望月久稔講師，日本学術振興会特別研究員の 安藤一秋博士，奈良工業高等専門学校情報工学科の小山雅史助手，徳島大学工学部知能情報工学教室の 富士正人技官，同教室の青江研究室の大学院生 辻孝子氏，溝渕昭二氏， 岡田真氏，徳永秀和氏，山川善弘氏，李相坤氏，李泰憲氏，EL Sayed Atlam 氏， 住友徹氏，鄭洙氏はじめ，研究室の諸氏には熱心なご健闘をいただいた．

ここに記して，以上の方々に深く感謝の意を表する．

参考文献

[1] 奥村学, 田中穂積. 自然言語解析における意味的曖昧性を増進的に解消する計算モデル. 人工知能学会誌, Vol. 4, No. 6, pp. 687–694, 1989.

[2] 高橋直人, 板橋秀一. 単語共起頻度を利用した形態素解析. 情報処理学会自然言語処理研究会, 第 88–NL–69 巻, pp. 1–8, 1988.

[3] 高松忍, 西田富士夫. 動詞パターンと格構造に基づく英日機械翻訳. 電子情報通信学会論文誌, Vol. J64–D, No. 9, pp. 815–822, 1981.

[4] 中岩浩巳, 池原悟. 日英翻訳システムにおける用言意味属性を用いたゼロ代名詞照応解析. 情報処理学会論文誌, Vol. 34, No. 8, pp. 1705–1715, 1993.

[5] 大島義光, 阿部正博, 湯浦克彦, 武市宣之. 格文法による仮名漢字変換の多義解消. 情報処理学会論文誌, Vol. 27, No. 7, pp. 679–687, 1986.

[6] 牧野寛, 木澤誠. べた書き文のかな漢字変換システムとその同音語処理. 情報処理学会論文誌, Vol. 22, No. 1, pp. 59–67, 1981.

[7] 山本喜大, 久保田淳市. 共起グループを用いたかな漢字変換. 情報処理学会第 44 回全国大会, No. 3, pp. 189–190, 1992.

[8] 澗潟謙一, 荒木健治, 宮永喜一, 栃内香次. 表層より得られる単語共起情報とその評価. 情報処理学会自然言語処理研究会, 第 90–NL–80–2 巻, 1990.

[9] 森本勝士, 入口浩一, 青江順一. 二つのトライを用いた辞書検索アルゴリズム. 電子情報通信学会論文誌 D–II, Vol. J76–D–II, No. 11, pp. 2374–2383, 1993.

[10] 小林義行, 徳永健伸, 田中穂積. 名詞間の意味的共起情報を用いた複合名詞の解析. 自然言語処理, Vol. 3, No. 1, pp. 29–43, 1996.

[11] 松川智義, 中村順一, 長尾真. 共起関係に注目したDM分解と確率的推定による単語のクラスタリング. 情報処理学会自然言語処理研究会, 第 89–NL–72 巻, pp. 1–8, 1989.

[12] 松本一則. 解析木データベースを用いた単語間の共起関係の抽出およびその構文解析への利用. 電子情報通信学会論文誌 D-II, Vol. J75-D-II, No. 3, pp. 589-600, 1992.

[13] 横山弘子, 大町真一郎, 阿曽弘具. コーパスからの共起情報を用いた日本語動詞の意味分類. 電子情報通信学会言語理解とコミュニケーション研究会, 第 NCL97-55 巻, 1998.

[14] 宮崎正弘, 池原悟, 横尾昭男. 複合語の構造化に基づく対訳辞書の単語結合型辞書引き. 情報処理学会論文誌, Vol. 34, No. 4, pp. 743–754, 1993.

[15] 小山雅史, 獅々堀正幹, 青江順一. 階層化概念辞書の高速検索アルゴリズム. 情報処理学会第 51 回全国大会, 第 7E-2 巻, pp. 4–235–4–236, 1995.

[16] 桑畑和佳子, 橋本三奈子, 青山文啓. Ipal 名詞辞書による多義性解消のためのコロケーションの分析. 情報処理学会論文誌, Vol. 39, No. 6, pp. 1925–1934, 1998.

[17] 藤井洋一, 鈴木克志, 今村誠, 高山泰博. 共起情報を利用した文書の自動分類. 情報処理学会自然言語処理研究会, 第 97-NL-118 巻, pp. 97–104, 1997.

[18] 田中穂積. 自然言語処理 —基礎と応用—, 第 5 章. 電子情報通信学会, 1999.

[19] 小山雅史. 格構造解析における概念の効率的階層判定法の研究. PhD thesis, 徳島大学, 3 1999.

[20] Fredkin E. Trie memory. *Commun.ACM.*, Vol. 3, No. 9, pp. 490–500, 1960.

[21] 長尾真（編）. 自然言語処理, 第 6 章. 岩波書店, 1996.

[22] Knuth D.E. *The Art of Computer Programming*, chapter 6. 1973.

[23] 青江順一. キー検索技法－トライ法とその応用. 情報処理, Vol. 34, pp. 1244–251, 1993.

[24] 青江順一. ダブル配列による高速ディジタル検索アルゴリズム. 電子情報通信学会論文誌, Vol. J71-D, No. 4, pp. 1592–1600, 1988.

[25] 石畑清. アルゴリズムとデータ構造, 第 2 章. 岩波書店, 1989.

[26] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers —Principles, Techniques, and Tools—*, chapter 3, 4. Addison-Wesley, Reading Mass., 1986.

[27] Aoe J. An efficient digital search algorithm by using a double-array structure. *IEEE Transactions on Software Engineering*, Vol. SE-15, No. 9, pp. 1066–1077, 1989.

[28] 入口浩一. グラフ構造に対する効率的記憶検索法. PhD thesis, 徳島大学, 3 1997.

[29] 森本勝士. トライ構造による辞書検索法に関する研究. PhD thesis, 徳島大学, 3 1995.

[30] 日本電子化辞書研究所. EDR 電子化辞書, 1996.

[31] 北研二, 山口直宏. World wide web からの対訳データの自動収集. 情報処理学会自然言語処理研究会, 第 98-NL-128 巻, pp. 127–134, 1998.

[32] 藤井敦, 石川徹也. 日本語複合語の自動分割と日英語基対訳辞書の作成. 情報処理学会自然言語処理研究会, 第 98-NL-128 巻, pp. 67–72, 1998.

[33] 奥村明俊, 石川開, 佐藤研治. コンパラブルコーパスと対訳辞書による日英クロス言語検索. 自然言語処理, Vol. 5, No. 4, pp. 77–93, 1998.

付 録A

ダブル配列の実験に使用したキー集合

Weltanschauung
Weltanschauungen
Weltansicht
Weltansichten
Weltpolitik
Weltpolitiken
Welwitschia
Wend
Wendic
Wendish
Wertherian
Wertherism
Wesleyan
Wesleyanism
Westerner
Westernization
Westm.
Westminster
Westphalian
Westralian
Whig
Whiggery
Whiggish
Whiggism
Whistlerian
Whitehall
Whitmanesque
Whitmonday
Whitsun
Whitsunday
Wicca
Wichita
Wiegenlied
Wiegenlieder
Wigwam
Wiikite
Wilsonian
Wilsonianism
Wilton
Wiltshire
Winchester
Windsurfer
Winesap
Winnebago
Winnie
Winnipegger
Wintu
Wirephoto
Wis.
Wisconsin
Wisdom
Wm.
Wobbly
Woden
Wolf
Wolffian
Wolof

Wolverine
Woodland
Woodstock
Wordsworthian
Work
Worship
Worshipful
Wotan
Wraf
Wren
Writings
Wu
Wy.
Wyandotte
Wycliffite
Wykehamical
Wykehamist
Wyo.
Wyomingite
X
X-disease
X-irradiation
X-radiation
X-ray
X-rts.
X.D.
X.L.
X.i.
XI
XP
XV
Xanadu
Xanthippe
Xerography
Xerox
Xhosa
Xn.
Xnty.
Xray
Xt.
Xtian.
Xty.
Xylocain
Y
Y-ogen
Y-track
Y.B.
Y.C.
Y.M.
Y.M.C.A.
Y.M.Cath.A.
Y.P.S.C.E.
Y.T.
Y.W.C.A.
YAG
YAVIS
YHWH
YIG
YMCA
YMHA
YSO
YWCA
YWHA
Yahoo
Yahrzeit
Yahvistic
Yahweh
Yahwism
Yahwist
Yahwistic
Yakima
Yakut
Yamasee
Yank
Yankee
Yankee-Doodle
Yankeedom
Yankeefied
Yankeeism
Yankeeland
Yanomama
Yanomamo
Yao
Yaqui

－－　以下省略　－－

付 録B

リンクトライの実験に使用したキー集合

B.1　関係情報の内部表現値

［内部表現値］	［日本語共起辞書］	［英語共起辞書］
1	@rentai	@co-modifier
2	@renyou	@composite
3	@unit	@d-object
4	あたりの	@d-object(bare)
5	あるいは	@d-object(ing)
6	いう	@d-object(to)
7	いえば	@d-object(to_be_done)
8	いた	@i-object
9	いった	@o-complement
10	いる	@o-complement(bare)
11	および	@o-complement(ing)
12	か	@o-complement(pp)
13	かが	@o-complement(to)
14	かぎりの	@o-complement(to_be_doing)
15	かつ	@o-complement(to_be_done)

16	かで	@passive-by
17	かと	@passive-complement
18	かという	@passive-complement(bare)
19	かということでは	@passive-complement(ing)
20	かということは	@passive-complement(pp)
21	かというと	@passive-complement(to)
22	かとも	@passive-complement(to_be_doing)
23	かどうか	@passive-complement(to_be_done)
24	かに	@passive-object
25	かについて	@passive-subj
26	かにも	@post-modifier
27	かによって	@pre-modifier
28	かの	@pred-subj
29	かは	@s-complement
30	かへ	@s-complement(ing)
31	かも	@s-complement(pp)
32	かもしれないと	@s-complement(to)
33	から	@s-complement(to_be_doing)
34	からすると	@s-complement(to_be_done)
35	からだけ	@subject
36	からだけでは	@unit
37	からだと	For
38	からで	Of
39	からでは	Wi’
40	からでも	a

－－　以下省略　－－

B.2　日本語共起辞書

”´70S”	ファッション	977
”＜知らされなかった議会＞計画”	予定	255
”#II#族”	原子	977
”#III#‐1”	掲げ	474
”#III#族”	”#V#族”	977
”#IV#族”	原子	977
”#VI#族”	原子	977
いくつ	調査報告	28
007	商社	12
05mk2	機能	625
OA 化計画	始ま	849
1	示	849
—	い	280
—	大学	625
—	断面	977
—	有望	977
”1.0.”	開始	49
”1/4 波長変成器”	ラットレース回路	977
10	3 月	49
10	開始	849
10	出荷	944
100	な	554
100	運動	977
100	シリーズ	977
10000	採用	554
1000L	よ	280
100R	後継	625
1041GT	発売	944

1045SP	1047SP	280
1047SP	機種	625
104X	シリーズ	977
1050V	1080V	977
1070V	1080V	977
”10-BASE3”	用意	554
10E	用意	554
10 円玉	あ	49
10 円玉	な	49
10 円玉	公衆電話	625
10 円玉	置	944
10 円玉	党	977
10 月債	引き上げ	33
10 進演算	結果	625
10 進計数回路	よ	280
10 進計数回路	あ	558
10 進計数回路	回路	849
10 進固定小数点方式	採用	944
10 進数	表わ	165
10 進数	16 進数	280
10 進数	加え	474
10 進数	変換	474
10 進数	表わ	944
10 進数	計数	944
10 進数	含	944
10 進数	データ	977
10 進数	16 進数	977
10 進数	浮動小数点数	977
10 進数字	整数	571

10 進表現	採用	944
10 進法	小数	255
10 進法	採用	944
10 進法	16 進法	977
10 人委員会	決め	165
11	項目	977
110	継続	912
1100	シリーズ	977
1100	”1100/82”	977
”1100/60”	”1100/90”	33
”1100/60B”	リプレース	474
”1100/81”	リプレース	944
”1100/90”	サポート	165
”1100/90”	強化	977
”1100/90”	シリーズ	977
”1100/90Model2.”	シリーズ	977
”1100/94”	”1100/72”	280
1121	サポート	165
116 号事件	捜査	977
1171	プロセッサー	977
119 番通報	直後	977
12	もたら	625
12	も	944
120E	同等	280
120S	同等	849
12 イマーム派	信じ	849

－－　以下省略　－－

B.3　英語共起辞書

"%"	best	65
"%"	reform	121
"%"	sale	121
"'30s"	classic	2
"'88"	Olympian	2
"'Pop Art was'"	tall	35
"'s"	quality	178
"'s"	have	35
Okada	port	2
one	member	27
one	shoulder	2
one	van	121
one	capable	121
one	important	121
"a quarter"	counterpart	121
"1-a"	classification	2
"1.45 million"	mark	2
"1.76 million"	yen	2
"10-year-old"	boy	27
billion	dollar	2
100th	Congress	27
100th	anniversar	27
100th	birthday	2
101st	Congress	27
101st	Congress	2
107th	birthday	27
10th	victor	27
10th	anniversar	2

10th	boundary	2
10th	place	2
10th	work	101
10th	month	121
115th	Derby	2
11th	floor	27
11th	year	27
11th	round	2
11th	seed	2
11th	year	2
"12-passenger"	wagon	27
"12-to-one"	engine	27
12th	anniversar	27
12th	place	27
"13.5 million"	dollar	2
13th	floor	2
13th	series	105
"14-power"	conference	27
14th	automobile	27
14th	centur	27
14th	floor	2
14th	victor	2
14th	May	121
"15-to-one"	engine	27
15th	anniversar	2
15th	centur	2
15th	November	121
"16-hour"	training	27
16th	centur	27

16th	centur	2
16th	green	2
17th	"Area Support Group"	27
17th	Century	27
17th	year	27
17th	centur	2
17th	minute	2
17th	place	2
17th	tee	2
18th	anniversar	27
"1938-39"	rule	27
1980s	cinema	2
1980s	slump	2
"1988 Summer Olympic Games"	Seoul	105
"1990 World Cup"	squad	2
19th	straight	1
19th	career	27
19th	centur	27
19th	month	27
19th	centur	2
1st	game	27
1st	inning	27
1st	victor	27
1st	"Royal Tank Regiment"	2
1st	run	2
1st	start	2

－－　以下省略　－－

4

様式9

論文審査の結果の要旨

報告番号	甲　工 乙　工　第　165　号 工　修	氏名	森田和宏
審査委員	主　査　青江順一 副　査　矢野米雄 副　査　小野典彦		
学位論文題目	トライ構造を用いた共起情報の効率的記憶検索法に関する研究		

審査結果の要旨

本論文は，自然言語辞書に構築される基本語彙の関係を定義することで無限に作り出される共起情報の効率的な記憶検索技法に関する研究の成果をまとめたものである．

第1章では，緒論として，自然言語処理システムにおける共起情報の利用の歴史的背景を述べると共に，本研究の目的ならびにその工学上の意義を述べることで，本研究の意義及び位置付けを明確にしている．第2章では，共起情報の概要と意義について説明している．また，自然言語処理システムにおける辞書の意義についても述べると共に，辞書を構成する基本的なアルゴリズムについて説明している．第3章では，辞書の構成法として最も適しているトライ構造について概要を述べると共に，その検索，更新アルゴリズムについて説明している．第4章では，トライ構造を実装するためのデータ構造について説明すると共に，最も効率的な手法であるダブル配列法について，検索，更新アルゴリズムとともに詳細に述べている．また，ダブル配列法の問題点であるキー追加の速度を改善する手法として，ダブル配列で構築された辞書を変更することなく高速化する手法と，ダブル配列に格納した情報を利用して高速化を実現する手法を提案している．第5章では，共起情報を効率的に記憶検索する手法として，関連研究とともに提案手法であるリンクトライについて述べ，リンクトライにおいて定義されるリンク情報により，冗長性を排除した効率的な記憶が可能となることを示すと共に，提案した構造における共起情報の検索，更新アルゴリズムを説明している．また，ダブル配列を用いたリンクトライのデータ構造についても説明し，リンクトライのデータ構造を提案する上で考案された，複数の辞書を1つのダブル配列で管理する手法も述べている．第6章では，提案手法に対して検索速度，記憶効率の理論的評価，及び実験による評価を与え，本手法の有効性を確かめると共に，考察を加えている．最後に，第7章で本研究で得られた諸成果の総括を行い，今後の研究課題について述べている．

以上本研究は，トライ構造を用いた共起情報の効率的な記憶検索技法を提案し，その有効性を考察したものであり，本論文は博士（工学）の学位授与に値するものと判定する．