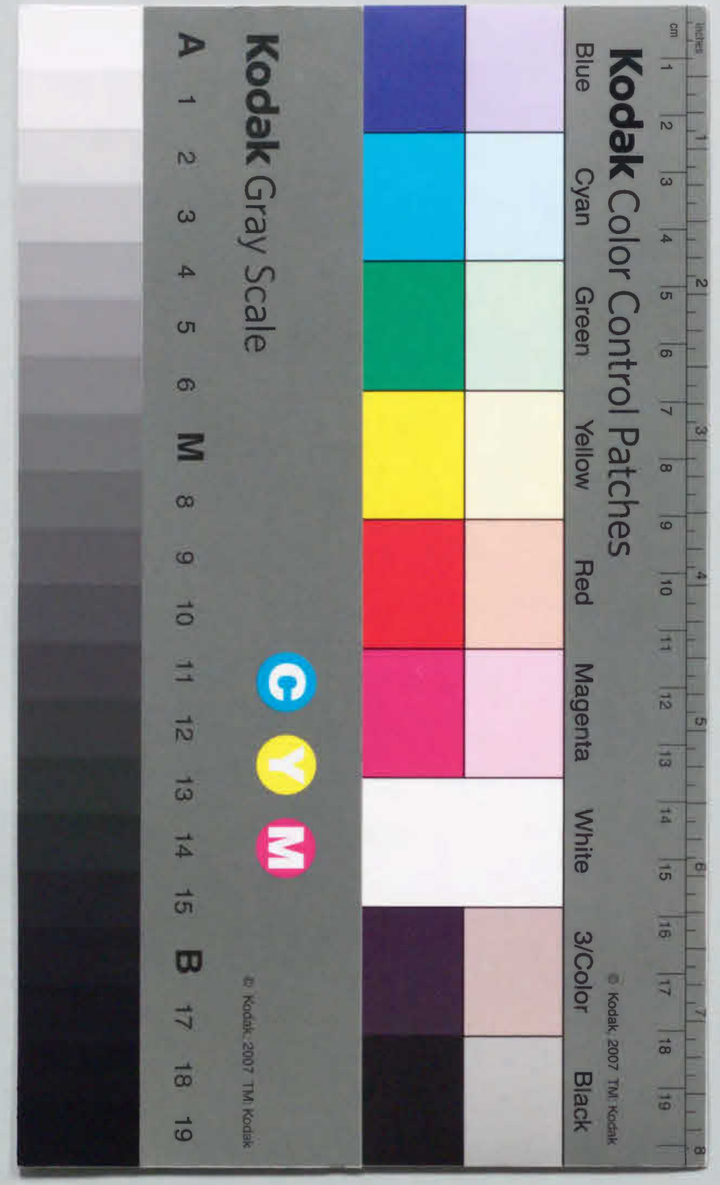


論　　文　　目　　録

報告番号	甲 工 (乙 工) 工 修 第 32 号	氏　　名	獅々堀　正幹
学位論文題目	２進ディジタル探索木を用いた辞書検索法とその応用に関する研究		

論文の目次

参考論文

主論文

1．“片仮名異表記の生成および統一手法”，獅々堀正幹，津田和彦，青江順一，電子情報通信学会論文誌，**Vol. J77-D-II**, No. 2, pp. 380-387 (1994年2月).
2．"An Automatic Selection Method of Key Search Algorithms", Masami Shishibori, Jun-ichi Aoe, Ki-Hong Park and Hisatoshi Mochizuki, *IEICE Transactions on Information and Systems*, **Vol. E78-D**, No. 4, pp. 383-393 (Apr. 1995).
3．“階層化による２進ディジタル探索（BDS）木の改善”，獅々堀正幹，清原聡，青江順一，電子情報通信学会論文誌，**Vol. J79-D-I**, No. 2, pp. 79-87 (1996年2月).
4．"Fast Allocation of Diagrams without Backtracking Processes", Masami Shishibori and Jun-ichi Aoe, *International Journal of Information Sciences*, **Vol. 92**, No. 1-4, pp. 65-85 (Apr. 1996).
5．"An Algorithm to Allocate Diagrams Automatically on Document Formatting Systems", Masami Shishibori, Takeshi Arita, Hisatoshi Mochizuki and Jun-ichi Aoe, *IEICE Transactions on Information and Systems*, **Vol. E80-D**, No. 2, pp. 259-273 (Feb. 1997).
6．"An Order Searching Algorithm of Extensible Hashing", Masami Shishibori and Jun-ichi Aoe, *International Journal of Computer Mathematics*, **Vol. 63**, Nos. 3+4, pp. 179-201 (Apr. 1997).

副論文

1．"An Efficient Algorithm for Allocating Diagram on Automatic Document Layout", Masami Shishibori, Hisatoshi Mochizuki, Takeshi Arita and Jun-ichi Aoe, *Proceedings of International Joint Conference on Information Sciences*, NC, USA, pp. 48-51 (Nov. 1994).
2．"An Efficient Algorithm for Diagram Allocation on Automatic Document Layouts", Masami Shishibori, Masao Fuketa, Takeshi Arita and Jun-ichi Aoe, *Proceedings of International Computer Symposium*, Hsinchu, Taiwan, pp. 630-637 (Dec. 1994).
3．"Extensible Hashing Using an Improved Binary Digital Search-tree", Masami Shishibori, Kazuaki Ando, Hirokazu Iriguchi and Jun-ichi Aoe, *Proceedings of the second International Joint Conference on Information Sciences*, NC, USA, pp. 417-420 (Sept. 1995).
4．"An Order-Preserving Access Method Using an Improved BDS-tree", Masami Shishibori, Kazuaki Ando, Hirokazu Iriguchi and Jun-ichi Aoe, *Proceedings of the third Natural Language Processing Pacific Rim Symposium*, Seoul, Korea, pp. 473-479 (Dec. 1995).
5．"An Efficient Order-Preserving Access Method Using Trie Hashing", Masami Shishibori, Shoji Mizobuchi, Kazuhiro Morita and Jun-ichi Aoe, *Proceedings of the International Workshop on Information Retrieval with Oriental Languages*, Taejon, Korea, pp. 102-107 (June. 1996).
6．"An Efficient Retrieval Algorithm of Binary Digital Search-trees Using Hierarchical Structures", Masami Shishibori, Yoshitaka Hayashi, Kazuhiro Morita and Jun-ichi Aoe, *Proceedings of the 19th International Conference on Research and Development in Information Retrieval*, Zurich, Switzerland, pp. 340 (Aug. 1996).
7．"An Efficient Method of Compressing Binary Trie", Masami Shishibori, Hisatoshi Mochizuki, Takeshi Arita and Jun-ichi Aoe, *Proceedings of 1996 IEEE International Conference on Systems, Man and Cybernetics*, Beijing, China, pp. 2133-2138 (Oct. 1996).
8．"A Trie Compaction Algorithm for a Large Set of Keys", Jun-ichi Aoe, Katsushi Morimoto, Masami Shishibori and Ki-Hong Park, *IEEE Transactions on Knowledge and Data Engineering*, **Vol. KDE-8**, No. 3, pp. 476-491 (Mar. 1996).

2

論　文　内　容　要　旨

報告番号	甲　工 (乙　工)　第　32　号 工　修	氏　　名	獅々堀　正幹
学位論文題目	2進ディジタル探索木を用いた辞書検索法とその応用に関する研究		

内容要旨

本論文は，キー検索技法の中で特に2進ディジタル探索木（Binary Digital Search Tree：BDS木と呼ぶ）を用いた辞書検索法に関する研究の成果をまとめたものであり，次の6章より構成される．

まず，第1章では，緒論として，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，本研究の意義および位置付けを明確にする．更に，本研究によって得られた諸成果を概説する．

次に，第2章では，キー検索技法の歴史的背景を述べると共に，各種キー検索技法の中でも，特に2分探索木法，多分探索木法，ハッシュ法，トライ法についての構成法を紹介し，各技法の特徴および問題点について解説する．

そして，第3章では，本研究の基となるBDS木を用いたトライハッシュ法の構成法を説明する．トライハッシュ法は，ハッシュ法の特徴である高速な検索能力を継承しつつ，従来のハッシュ法では困難であった順検索を可能にする手法である．しかしながら，登録キー数が多くなると，ハッシュ法の索引部を形成するBDS木のサイズが大きくなり，BDS木全体を主記憶上に格納できなくなる．この問題点を解決するため，JongeらはBDS木を先行順ビット列と呼ばれるコンパクトなビット列に圧縮する手法を提案した．そこで，本章ではJongeらの提案した圧縮手法について解説した後，先行順ビット列上でのキーの検索・追加アルゴリズムを示す．

第4章では，第3章で紹介した先行順ビット列の時間的および空間的な問題点を明確にした後，先行順ビット列により表現されたBDS木の時間効率および空間効率を改善する手法をそれぞれ提案する．まず，キー集合が大規模になると，先行順ビット列が非常に長くなり，その結果，ビット列の後方に位置するキーに対する処理の時間効率が悪化する．この時間的な問題点に対して，本論文では，BDS木の構造を階層的に分割管理し，不必要な部分木に対する処理を削減することにより，大規模なキー集合に対しても時間効率の悪化を防ぐ手法を提案する．また，先行順ビット列上でキーの検索・追加を実現するためには，BDS木のすべてのノードが2本の枝を有する必要がある．従来の手法では，1本の枝しか持たないノードに対してダミーリーフと呼ばれる擬似的な葉を持たせていた．しかしながら，大規模なキー集合に対しては，ダミーリーフ数が非常に多くなり，その結果，ビット列が非常に長くなる．この空間的な問題点に対して，本論文では，ダミーリーフを用いずに，よりコンパクトなビット列に圧縮する手法を提案する．更に，それぞれの提案手法の理論的評価，及び実験による具体的評価を与え，本手法の有効性を確かめる．

また，第5章では，第4章で提案した2進ディジタル探索木を用いた辞書検索法の文書処理への応用として，2進ディジタル探索木で構成された片仮名辞書と片仮名変換ルールを用いた効率的な片仮名異表記生成手法，および2進木構造で構築された文書構造データベースを用いた自動図表配置手法について述べる．

最後に，第6章では，本研究で得られた諸成果の統括を行い，今後の研究課題について述べる．

2進ディジタル探索木を用いた辞書検索法とその応用に関する研究

1997年4月

獅々堀 正幹

2進ディジタル探索木を用いた辞書検索法とその応用に関する研究

1997年4月

獅々堀　正幹

内容梗概

本論文は，キー検索技法の中で特に２進木ディジタル探索木（Binary Digital Search Tree：BDS 木と呼ぶ）用いた辞書検索法に関する研究の成果をまとめたものであり，次の６章より構成される．

第１章では，緒論として，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，キー検索技法の歴史的背景を述べると共に，これまでに提案されている各種キー検索技法の特徴および問題点について解説する．

第３章では，本研究の基となる BDS 木を用いたトライハッシュ法を説明する．また，ハッシュ法の索引部を成す BDS 木を先行順ビット列と呼ばれるコンパクトなビット列に圧縮する手法について解説した後，先行順ビット列を用いたキーの検索・追加アルゴリズムを示す．

第４章では，先行順ビット列の問題点を明らかにした後，時間効率および空間効率を改善する手法をそれぞれ提案する．まず，先行順ビット列はキー集合が大規模になると時間効率が低下する．本論文では，BDS 木の構造を階層的に分割管理し，不必要な部分木に対する処理を削減することにより，時間効率の低下を防ぐ手法を提案する．また，先行順ビット列上でのキーの検索・追加を実現するためには，BDS 木のすべてのノードが２本の枝をもつ必要がある．従来の手法では，１本の枝しか持たないノードに対して擬似的な葉を持たせていた．本論文では，擬似的な葉を用いずに，よりコンパクトなビット列に圧縮する手法を提案する．更に，それぞれの提案手法の理論的評価，及び実験による具体的評価を与え，本手法の有効性を確かめる．

第５章では，２進木ディジタル探索木を用いた辞書検索法の文書処理への応用として，片仮名辞書と片仮名変換ルールを用いた効率的な片仮名異表記生成手法，および文書構造を用いた図表自動配置手法について述べる．

第６章は結論であって，本研究で得られた諸成果を総括的に述べるとともに，今後の研究課題についても触れる．

関連発表論文

【主論文】

1. 獅々堀正幹，津田和彦，青江順一，“片仮名異表記の生成および統一手法”，電子情報通信学会論文誌，Vol.J77-D-II，No.2，pp.380–387 (1994年2月).
2. Masami Shishibori, Jun-ichi Aoe, Ki-Hong Park and Hisatoshi Mochizuki, “An Automatic Selection Method of Key Search Algorithms”, IEICE Transactions on Information and Systems, Vol.E78-D, No.4, pp.383–393 (Apr. 1995).
3. 獅々堀正幹，清原聡，青江順一，“階層化による2進ディジタル探索（BDS）木の改善”，電子情報通信学会論文誌，Vol.J79-D-I，No.2，pp.79–87 (1996年2月).
4. Masami Shishibori and Jun-ichi Aoe, “Fast Allocation of Diagrams without Backtracking Processes”, International Journal of Information Sciences, Vol.92, No.4, pp.65–85 (Apr. 1996).
5. Masami Shishibori, Takeshi Arita, Hisatoshi Mochizuki and Jun-ichi Aoe, “An Algorithm to Allocate Diagrams Automatically on Document Formatting Systems”, IEICE Transactions on Information and Systems, Vol.E80-D, No.2, pp.259–273 (Feb. 1997).
6. Masami Shishibori and Jun-ichi Aoe, “An Order Searching Algorithm of Extensible Hashing”, International Journal of Computer Mathematics, Vol.63, Nos.3+4, pp.179–201 (Apr. 1997).

【副論文】

1. Masami Shishibori, Hisatoshi Mochizuki, Takeshi Arita and Jun-ichi Aoe, "An Efficient Algorithm for Allocating Diagrams on Automatic Document Layouts", Proceedings of International Joint Conference on Information Sciences, North Carolina, USA, pp.48–51 (Nov. 1994).
2. Masami Shishibori, Masao Fuketa, Takeshi Arita and Jun-ichi Aoe, "An Efficient Algorithm for Diagram Allocations on Automatic Document Layouts", Proceedings of International Computer Symposium, Hsinchu, Taiwan, pp.630–637 (Dec. 1994).
3. Masami Shishibori, Kazuaki Ando, Hirokazu Iriguchi and Jun-ichi Aoe, "Extensible Hashing Using an Improved Binary Digital Search-tree", Proceedings of the second International Joint Conference on Information Sciences, North Carolina, USA, pp.417–420 (Sept. 1995).
4. Masami Shishibori, Kazuaki Ando, Hirokazu Iriguchi and Jun-ichi Aoe, "An Order-Preserving Access Method Using an Improved BDS-tree", Proceedings of the third Natural Language Processing Pacific Rim Symposium, Seoul, Korea, pp.473–479 (Dec. 1995).
5. Masami Shishibori, Shoji Mizobuchi, Kazuhiro Morita and Jun-ichi Aoe, "An Efficient Order-Preserving Access Method Using Trie Hashing", Proceedings of the International Workshop on Information Retrieval with Oriental Languages, Taejon, Korea, pp.102–107 (June. 1996).
6. Masami Shishibori, Yoshitaka Hayashi, Kazuhiro Morita and Jun-ichi Aoe, "An Efficient Retrieval Algorithm of Binary Digital Search-trees Using Hierarchical Structures", Proceedings of the 19th International Conference on Research and Development in Information Retrieval, Zurich, Switzerland, pp.340 (Aug. 1996).
7. Masami Shishibori, Hisatoshi Mochizuki, Takeshi Arita and Jun-ichi Aoe, "An Efficient Method of Compressing Binary Tries", Proceedings of 1996 IEEE International Conference on Systems, Man and Cybernetics, Beijing, China, pp.2133–2138 (Oct. 1996).

8. Jun-ichi Aoe, Katsushi Morimoto, Masami Shishibori and Ki-Hong Park, "A Trie Compaction Algorithm for a Large Set of Keys", IEEE Transactions on Knowledge and Data Engineering, Vol.KDE-8, No.3, pp.476–491 (Mar. 1996).

【研究会資料】

1. 獅々堀正幹，青江順一，"文章校正支援システムにおける校正知識獲得"，情報処理学会自然言語処理研究会，92-NL-88，pp.79–86 (1992年3月).
2. 獅々堀正幹，青江順一，"片仮名異表記の生成および統一手法"，情報処理学会自然言語処理研究会，93-NL-94，pp.33–40 (1993年3月).
3. 獅々堀正幹，青江順一，"文書構造知識を利用した図表の自動レイアウトアルゴリズム"，情報処理学会ヒューマンインタフェース研究会，95-HI-59，pp.23–30 (1995年3月).
4. 林叔隆，獅々堀正幹，伊與田敦，津田和彦，青江順一，"複合語キーワードの効率的抽出法"，情報処理学会自然言語処理研究会，94-NL-104，pp.63–70 (1994年11月).
5. 小山雅史，林叔隆，獅々堀正幹，青江順一，"トライ構造による概念階層の高速判定アルゴリズム"，情報処理学会自然言語処理研究会，96-NL-112，pp.1–6 (1996年3月).
6. 安藤一秋，辻孝子，獅々堀正幹，青江順一，"日本語定型表現の分析と効率的照合アルゴリズム"，情報処理学会自然言語処理研究会，96-NL-112，pp.69–74 (1996年3月).

【講演報告】

1. 獅々堀正幹，青江順一，"分類知識表現を用いたキー検索アルゴリズムの決定法"，情報処理学会第45回全国大会，5U-5，pp.5-363～5-364 (1992年10月).（研究奨励賞受賞）
2. 獅々堀正幹，津田和彦，青江順一，"文書レイアウトにおける自動図表配置手法"，情報処理学会第50回全国大会，4N-1，pp.3-183～3-184 (1995年3月).

3. 獅々堀正幹，青江順一，“文書構造を利用した自動図表配置手法におけるバックトラック処理の解消”，言語処理学会第 1 回年次大会，A1-5，pp.37–40 (1995 年 3 月).
4. 獅々堀正幹，望月久稔，弘田正雄，青江順一，“2 進木トライ構造の効率的な圧縮手法”，情報処理学会第 5 2 回全国大会，4U-6，pp.1-65～1-66 (1996 年 3 月).
5. 安藤一秋，藤沢貴之，獅々堀正幹，青江順一，“定型表現を利用した効率的な形態素解析の実現”，情報処理学会第 5 1 回全国大会，2H-4，pp.3-27～3-28 (1995 年 10 月).
6. 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 5 1 回全国大会，7E-2，pp.4-235～4-236 (1995 年 10 月).
7. 辻孝子，安藤一秋，獅々堀正幹，青江順一，“活用語を含む助詞的定型表現の分析”，情報処理学会第 5 2 回全国大会，7B-2，pp.3-101～3-102 (1996 年 3 月).
8. 溝渕昭二，弘田正雄，獅々堀正幹，青江順一，“類似用例文の効率的検索手法とその応用”，情報処理学会第 5 3 回全国大会，5L-1，pp.2-79～2-80 (1996 年 10 月).
9. 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 5 3 回全国大会，5L-2，pp.2-81～2-82 (1996 年 10 月).

目 次

図 目 次

表目次

第1章

緒　論

近年，計算機能力の向上，低価格化には目覚ましいものがあり，その結果，計算機は社会の幅広い分野に浸透し，様々な用途で利用されている．この計算機利用における一つの大きな利点は，高速な検索能力であり，我々は各所でその恩恵を受けている．キー検索とは，キーを見出しとしてその関連情報であるレコードを探す技法であり，データベースシステムや自然言語処理システムなど非常に多くの分野で利用され，計算機による情報処理の基礎となるものである．現在までに数多くの検索手法 [4, 17] が提案されているが，各手法それぞれに長所と短所を兼ね備えており，すべての検索要求を満たす検索手法は存在していないのが現状である．そこで，できるだけ多くの検索要求を同時に満たすことができる新しい検索手法の考案は重要な課題である．

第２章では，キー検索手法の諸問題について議論すると共に，現在までに行われてきたキー検索手法の研究について紹介し，その問題点を明確にする．キー検索法には様々な手法があるが，その中でもハッシュ法は，高速な手法として様々な分野で応用されている．ハッシュ法は，ハッシュ表の大きさが変化しない静的ハッシュ法 [6, 19, 38] とその大きさが変化する動的ハッシュ法 [7, 13] に分類できる．静的ハッシュ法は，キー集合の性質（最大個数，構成文字など）が予測できる分野に対して高速な検索が実現できるが，キー集合の性質が予知できない分野では，ハッシュ表を大きく取りすぎると記憶領域が無駄になり，逆に小さく見積もると，衝突が頻発して検索効率が低下する [6]．これに対して，動的ハッシュ法はハッシュ関数とファイル構造を局所的に再構成することにより，キー集合の性質が予測できない環境においても，高速な検索を維持することができる [7]．必然的に，こ

のような環境では，レコード件数が十分に多く，ランダムアクセス可能な2次記憶を利用する場合を仮定する．この動的ハッシュ法は，2次記憶ファイルを利用したキー検索技法として，近年活発な研究が行われている [7, 13]．2次記憶ファイルを利用する代表的キー検索手法としては，B，B^+，B^*木（以後，B木で代表する）法 [11, 28, 36] が伝統的であるが，動的ハッシュ法の索引部はB木法よりコンパクトであるので，索引部を主記憶上に置くことで，ディスクアクセス回数を少なくできる特徴を有する．しかし，順検索を実現するには，キー分布の仮定や更新頻度を考慮する必要があり，一般的ではなかった．

本研究では，動的ハッシュ法の長所を継承し，かつ，辞書検索には必要不可欠な順検索が効率的に行える新しい検索手法の考案を目的とする．まず，本研究では動的ハッシュ法の一つである拡張ハッシュ法 [14, 34, 52] に着目する．拡張ハッシュ法は，ハッシュ関数値をビット列で定義し，バケット（レコードの格納場所）の分割・併合と同時にビット長を伸縮してハッシュ表を制御する．しかし，ハッシュ表のサイズが大きくなると，各バケットの局所的深さに隔たりが生じ，ハッシュ表は冗長になってしまう．この冗長性を解消するため，ハッシュ表をビット列の2進木構造で表現する手法 [24] が代表的方法として知られている．この拡張ハッシュ法でキーの順検索を実現するには，ハッシュ関数値として，キーを構成する文字列の2進数表現を用いることになり，キー数が多くなれば非常に大きい2進木を索引部で管理しなくてはならない．従って，2進木構造をコンパクトに圧縮することが重要な問題となる．これに対して，Jongeら [24] は2進木構造を先行順走査に従って，コンパクトなビット列（先行順ビット列と呼ぶ）に圧縮する手法を提案した．そこで，第3章では，2進ディジタル探索木を用いた拡張ハッシュ法について解説を行った後，Jongeらに従って，2進ディジタル探索木の先行順ビット列への圧縮方法を示す．

Jongeらが提案したデータ構造は非常にコンパクトではあるが，大規模なキー集合に対しては先行順ビット列が非常に長くなり，ビット列の後方に位置するハッシュ値の探索効率が低下する．また，キーの挿入・削除でビット列の前の部分の変更を行うと，それ以降のビット列をシフトする必要があり，更新時間も長くなる．このように，従来の手法ではキー集合の大規模化に伴う時間効率の低下が避けられない．一方，先行順ビット列でキーの検索・追加を実現するためには，2進木の全ノードが2本の枝を持つ必要がある．そこで，Jongeらは，1本の枝しか持たないノードに対してはダミーリーフと呼ばれる擬似的な葉を持たせた．しかしながら，このダミーリーフは全外部ノードの40〜60%を占めるため，ビット列が必要以上に長くなる．このように，従来の2進木構造にJongeらの圧縮法をそのまま適用すると，時間的・空間的な問題点が顕著になる．

第4章では，先行順ビット列を分割して管理することで，大規模なキー集合に対してビット列が長くなった場合でも，上記の時間効率の低下を解消する手法を提案する．本手法は2進木構造の高さを制限することで，木構造を階層的に分割し，分割された2進木構造単位で先行順ビット列を効率的に記憶管理する．更に，2進木構造をパトリシア構造に拡張することで，ダミーリーフを採用せずに，2進木構造をよりコンパクトなビット列に圧縮する手法も提案し，空間効率の問題点を解決する．パトリシア構造は1本の枝しか持たない内部ノードがすべて削除され，すべての内部ノードが2本の枝を有する構造である．本手法はパトリシア構造に含まれる削除ノードに関する情報を2進木構造内の内部ノード数と同じ長さのビット列で表現する．

本手法の評価としては，理論的な評価のみにとどまらず，種々のキー集合に対する実験も行った．その結果，2進木構造を階層的に管理することにより，処理速度が大幅に向上（検索が18〜20倍，追加は11〜13倍，削除では4〜6倍と高速化）されること，更に，パトリシア構造を導入し，ダミーリーフを削除することにより，先行順ビット列の長さを25〜40%短縮できることが実証された．

また，文書校正支援システム [39]，キーワード検索システム [18]，及び情報検索システム [25] などの辞書検索が要求される分野に，本検索手法を用いることにより，各種処理の効率化が見込まれる．しかしながら，片仮名表記のように表記に揺れを有する単語，即ち，異表記 [9] を有する単語を検索キーとする場合，辞書に登録されていない異表記は未登録語となり，処理精度の低下を招く．また，すべての異表記を辞書に登録する手法は，辞書の保守性，応用性において好ましくない．

第5章前半部では，このような異表記を有する片仮名表記の検索要求に対処するために，片仮名表記の揺れが外来語特有の表記方法に起因することに着目し，表記の揺れが存在する部分の音（文字列）を整理した片仮名異表記変換ルールと，このルールが容易に適用可能な正規表記を定義する．そして，片仮名異表記変換ルールと正規表記を用いた異表記生成手法を提案する．本手法は正規表記のみを辞書登録すればよいため，コンパクトで，かつ，すべての表記に対応可能な辞書が作成できる．また，本手法は従来の手法に比べて辞書アクセスの回数が数倍に増えるが，本検索手法は高速なキー検索が可能であり，特に，未登録語はディスクアクセスせずに判定できるため，実用的な速度で片仮名異表記を生成できる．

本手法の有効性を確認するため，135個の片仮名異表記変換ルールを定義し，約7万語の片仮名データに対して実験評価を行った結果，辞書圧縮率は79.5%，特に表記の揺れを有する語のみに対しては42.6%に圧縮でき，異表記生成処理に関しては99.0%の生成精度が得られ，本手法の有効性が確認できた．

更に，文書整形システム[29, 21]や文書編集システムなどで代表される文書処理システムにおいて，文書構造を自動抽出するためには，形態素解析などの辞書検索を行う自然言語処理技術が必要となることは勿論であるが，抽出した文書構造を効率的に木構造に蓄積する技術も重要である[54]．そこで，第5章後半部では，コンパクトなデータ構造に圧縮した文書構造を用いて図表を自動配置する手法を提案する．本手法により，格納された文書構造を高速にアクセスできるので，効率的（高速かつ正確）な自動レイアウトが可能となる．

第6章では，本研究で得られた結果をまとめ，今後の課題について述べる．

第2章

各種キー検索技法

2.1　緒　言

コンピュータ利用における一つの大きな利点は，高速な探索能力であり，我々は各所でその恩恵を受けている．しかしながら，すべての分野にうまく適合する探索手法は存在しないため，ソフトウェア開発者は適材適所となるべき探索手法の設計に労力を払わなければならない[4, 17, 23]．

キー検索（key retrieval, key search）法は，キー（key）を“見出し”としてその内容（レコード）を探す手法であり，キーとレコードの更新が行われるか否か，2次記憶を用いるか否か，順検索（order search）を必要とするか否か等の条件に応じて非常に多くの改善と拡張，また新しい手法の提案が行われている．それらは，各々特徴があり，図2.1のように数十種の技法に分類できる[45]．

図2.1に示すように，キー検索法は，表探索法（table search）と探索木法（tree search）に大別される．

まず，表探索法は，キーを表に蓄えて検索する方法である．この手法には，キーを逐次探索する線形探索（linear search），アルファベット順に並べておいて探索領域を2分の1に狭めていく2分探索法（binary search），キーをハッシュ表に分散記憶するハッシュ法（hashing）が属する．ここで，ハッシュ法はハッシュ表のキー分散が一様な場合，探索は極めて高速であるが，固定されたハッシュ表の大きさでは，予期されないキーの増加に対応しきれない欠点がある．この欠点は，ハッシュ表を局所的に大きくできる動的ハッ

```
(Table_Search (Linear_Search
                    (Sorted_Linear_Search))
              (Binary_Search)
              (Hashing (Perfect_Hashing
                    (Minimal_Perfect)
                    (Perfect)
                    (Composite_Perfect)
                    (Near_Perfect)
                    (Ordered_Perfect))
                  (Open_Hashing
                    (Linear_Open)
                    (Quadratic_Open)
                    (Double_Open)
                    (Ordered_Open))
                  (Chaining_Hashing
                    (Separate_Chaining)
                    (Coalesced_Chaining))
                  (Dynamic_Hashing
                    (Linear_Hashing)
                    (Extensible_Hashing))))
(Tree_Search (Binary_Tree
                    (Complete_Balanced_Tree)
                    (AVL_Tree))
              (Multiway_Tree
                    (B-Tree)
                    (B+-Tree)
                    (B*-Tree)
                    (Prefix_B+-Tree))
              (Digital_Tree
                    (Binary_Trie)
                    (Trie)))
```

図 2.1　キー検索法の記憶検索構造による分類例

シュ法（dynamic hashing）を用いれば解決できる．

次に，探索木法は，キーの値の大小関係により記憶検索する方法であり，キーの順検索が容易に実現できる．一方，ハッシュ法で順検索を行う場合には，キー分布の仮定や更新の頻度を考慮する必要があり，従来のハッシュ法では一般的に順検索を行うのは不向きである．このような探索木法には，2分探索木法（binary search tree）や B 木法（B-tree）

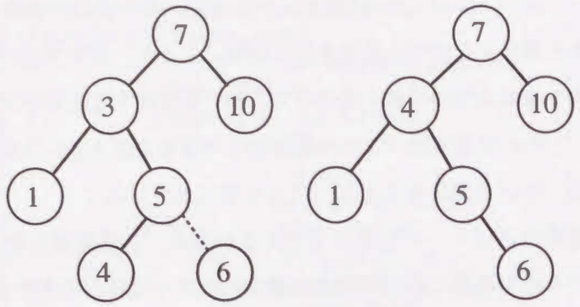
等で代表される多分木法（multiway tree）が属する．また，ディジタル探索木法（digital tree search）は，トライ（trie）法に代表されるように，キー自身の文字（または，値の桁）単位に探索木を構成するので，探索木法の範疇に入るが，検索方法はキーの値を一度に比較する一般的な探索木法の手法とは異なり，キーを構成する文字単位で比較が行われる．トライ法は，1回の走査ですべての共通接尾語が検索できるという特徴により，自然言語処理システムにおける辞書検索等 [3, 37] に応用されている．

このように，検索技法はキーの更新が行われるか否か，2次記憶を使うか否か，順検索が行われるか否か等の条件に応じて非常に多くの改善や拡張が行われている．従って，キー検索を応用する場合は，応用分野の特徴と各種キー検索手法の特徴を十分把握して最適な技法を採用することが必要不可欠である．また，すべての分野に適応可能な検索手法が存在しないため，より幅広い応用範囲に適合可能な機能を有する新しい検索技法の開発が待ち望まれている．

本章では，本研究での基礎となる各種キー検索技法の説明を行う．まず，2．2節では探索木法について述べる．探索木法に対しては，2分探索木法と多分木法を取り上げ，特に多分木法については代表的な B 木法，及び B 木の拡張である B^+木法について説明する．2．3節では表探索法として代表的なハッシュ法の概要を述べる．特に，ハッシュ表の大きさが変化しない静的（static）ハッシュ法と，ハッシュ表の大きさが変化する動的（dynamic）ハッシュ法に分けて説明する．そして，2．4節でディジタル探索法であるトライ法について説明する．

2.2　探索木法

探索木法 [28, 1, 20] はキー比較による大小関係から探索範囲を縮小しつつ，キーを探す方法であり，検索コストはキーの総数の対数オーダとなる．また，探索木法は，キーの挿入・削除が頻繁な動的キー集合に向いており，キーの値順を反映した構造になっているため，キーの順検索や接頭辞を指定しての検索に対して優れている．



(a)　6を挿入　　(b)　3を削除

図 2.2　2分探索木法の例

2.2.1　2分探索木法

2分探索木法は，図 2.2 に示すように各ノードが高々1組の左右の子を持つ順序木であり，各ノードは1個のキーの値を格納する．例えば，図 2.2-(a) に示す2分木において，キー5を探索する場合，根の値7に対する大小関係 $5 < 7$ より左の枝へ，次に $5 > 3$ より右の枝へと辿る．n個のキーが平衡（各ノードの部分木の高さが等しい状態）しているならば，1回の比較で探索領域は半分に狭められ，検索時間計算量は $O(log_2 n)$ となる [1]．このような木を完全平衡木と呼ぶ．

キーの挿入は，探索が失敗した場所に新しい葉を作成する．例えば，図 2.2-(a) に対してキー6は，キー5のノードの右の枝の葉として挿入される．削除は葉であるか1つの子しか持たないノードの場合は簡単であるが，2つの子を持つ場合は，削除するノードの右部分木の最小要素で置き換える．例えば，図 2.2-(a) から3を削除する場合は右部分木の最小要素4で置き換えられ，図 2.2-(b) が得られる．2分探索木法で2次記憶上に格納されたキーとレコードを検索する場合，木の高さ＋1が最大探索回数になるので，2次記憶から主記憶へのデータ転送回数が多くなる．これに対して，主記憶と2次記憶間のデータ転送のブロックに複数個のキーを格納し，転送回数を軽減する構造が多分木であり，2分木と比べ木の高さは低くなる．但し，検索性能を保証するためには，2分木と同様に平衡

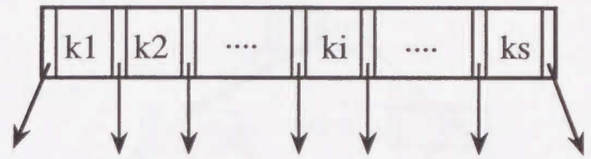


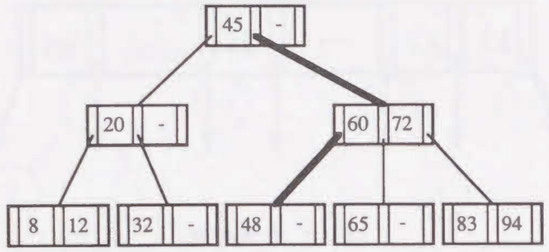
図 2.3　B 木の例

木にする必要がある．

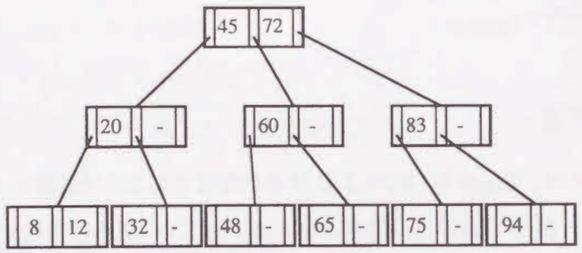
2.2.2　多分木法

多分木法としては，Bayer ら [10] による B 木が有名である．分岐数 m の B 木の各ノードは，最大 m 個，最小 $\lceil m/2 \rceil$（記号 $\lceil x \rceil$ は，x 以上の最小の整数を表す）個（ノードが根の時は1個）のキーとキー数より1つ多いポインタからなる．挿入と削除に対し，ノードの同一レベル内での分割，併合操作により，平衡木の性質を維持している．あるノードに s（$\lceil m/2 \rceil \leq s \leq m$）個のキー $k_1, k_2, \ldots, k_s$ が順に入っているとすると，$k_1 < k_2 < \ldots < k_s$ である（図 2.3）．このノードでキーxを検索する場合，k_1 から順に比較し，$x = k_i$ $(1 \leq i \leq s)$ が見つかれば成功し，そうでなければ，$x < k_i$ となる最初のキーの左にあるポインタを辿り，1段下のノードについて検索を行う．但し，$x > k_s$ のときは右端のポインタを辿る．葉レベルまで降りてもキー x が見つからなかった場合，x は未登録となる．

図 2.4 はノードに2つのキーを有する B 木の例であるが，図 2.4-(a) には 48 を検索する経路を太線で示してある．同図 (a) に 75 を挿入する場合，根から検索で 75 が入るべき場所を見つけると，83 と 94 が入っているノードに辿り着くが，満杯で入らない．そこで 75，83，94 の中央値 83 を上のレベルに移動し，残りのキーを2つのノードに分割する．しかし，上のレベルのノードも満杯で入らない．ここで，さらにこのノードを分割し，60，72，83 のうちの中央値 72 をさらに上のルートノードに挿入し，図 2.4-(b) が得られる．このルートノードも満杯ならば，もう1レベル上に新しいルートノードを作ることになる．



(a) キー４８を探索する例



(b) (a)のＢ木にキー７５を挿入する例

図 2.4　B 木の検索・挿入例

ルートノード以外のノードの分岐数は $\lceil m/2 \rceil + 1$ であるので，検索のためにノードを訪問する回数は，キー数の m を底とする対数オーダーである．Fagin ら [14] の理論的解析によると，ランダムにキーを挿入した場合，B 木の記憶利用率は 69%に収束する．

B^+木 [29, 11] はキーの直接検索だけでなく，順検索も行えるように B 木を拡張した手法であり，データベースの索引等に使用されている（図 2.5 ）．B^+木では，全てのキーを葉にのみ置いて，葉より上位レベルはランダム探索のための探索索引部と考える．そして，葉同志は順検索を可能にするためにリンクされる．従って，検索時には探索路を葉まで辿り，その葉でキーの存在を確認すればよい．但し，検索途中の節でキーが等しくなっても，最終確認は葉まで辿ってから行う必要がある．また，削除されるレコードは常に葉にあるので，B 木に比べ削除操作は簡単である．即ち，索引部にそのレコードのキーが置かれていても，以降も正しく分離値として働くので，削除せずそのまま残しておいてもよ

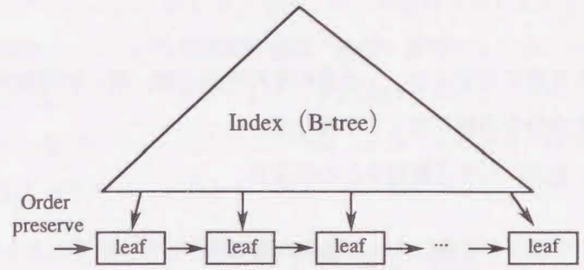


図 2.5　B^+木の構成

い．削除により葉の使用率が半分未満になれば，B 木と同様の分配または併合の処理を行う．

B 木は，あるキーの次の値のキーを探すのに $log_m n$ 回のアクセスを要するが，B^+木は次のキー探索には高々１回の２次記憶アクセスしか要しない．このように B^+木は，B 木のランダム検索効率を維持した上で効率的な順検索を可能にした技法である．

2.3　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．このハッシュ法には，ハッシュ表の大きさを固定した静的ハッシュ法と，ハッシュ関数とファイル構造を局所的に再構成する動的ハッシュ法がある．

2.3.1　静的ハッシュ法

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 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．静的ハッシュ法は，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，以下のような項目事項を考慮する必要がある．

1. キーの検索と更新に対するハッシュ表の平均探索回数，即ち時間効率．
2. ハッシュ表や余分な領域に関する空間効率．
3. 適用するキー集合に対する制約などの応用性．

そして，このような項目を考慮に入れた衝突の解消法として，以下のような手法が提案されている．

・完全（perfect）ハッシュ法：

衝突の起こらないハッシュ関数を工夫する手法．最悪の探索回数が常に1回となり，高速な検索が可能であるが，小さくてしかも追加と削除が起こらない静的キー集合に適用範囲が制約される．

・開番（open addressing）地法 [38]：

衝突した番地に代わる新たな番地を再計算する方法．追加と削除のある動的キー集合に対して適用できるが，表が一杯になると探索効率が悪くなる．

・連鎖（chaining）法 [38]：

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．開番地法と同様に動的キー集合への適用が可能で，しかも平均探索回数は開番地法よりは少なく，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．

2.3.2　動的ハッシュ法

静的ハッシュ法は，キー集合の性質（最大個数，構成文字）が予測できる分野に対して高速な検索が実現できるが，この性質が予知できない分野では，ハッシュ表を大きく取りすぎると記憶領域が無駄になり，逆に小さくしすぎると，衝突が頻発して検索効率が低下する．そこで，キーの性質が予測ができない環境においても，ハッシュ関数とファイル構造を局所的に再構成することにより，高速検索能力を維持させる動的ハッシュ法が考案された．静的ハッシュ法では，キーを発見するための平均探索回数が検索時間の評価基準となっているが，動的ハッシュ法では2次記憶を使用するので，時間評価は目的とするキー

を探索するために必要なバケットのディスクアクセス回数で決定される．即ち，ディスクアクセスコストはハッシュ法で必要な主記憶上の内部処理コストより十分に大きいからである．また，静的ハッシュ法の空間的評価は，衝突を解消するために必要な記憶領域を基準としていたが，動的ハッシュ法ではレコードが格納されたファイル全体の空間効率が重要な基準となる．動的ハッシュ法の空間効率は，レコード数を n，バケット数を w，バケットに格納できる最大レコード数を b で表すとき，動的ハッシュ法での空間効率は，

$$n\ /\ (wb)$$

で表される．よい空間効率を維持するためには，アクセス時間を犠牲にする必要がある．

動的ハッシュ法は，ハッシュ関数値をビット列で定義し，バケットの分割・併合と同時にビット長を伸縮してハッシュ表を制御する拡張ハッシュ法 [14, 34, 52] と，ハッシュ関数値を直接バケット番号に対応させ，ハッシュ表の伸縮をバケットの分割・併合と非同期に制御する線形ハッシュ法 [30, 31, 33] に分類できる．

拡張ハッシュ法では，キー k を十分に大きい m 長のビット列に写像するため，以下のようなハッシュ関数を定義する．

$$\mathrm{H(k)}=b_{m-1}\ \ldots\ b_2\ b_1\ b_0$$

（但し，b_i（$0 \leq i \leq m-1$）は，0または1のビットを表す）

また，H(k) に対して末尾の i 個のビット列を抽出したビット列を関数を

$$h_i(k)=b_{i-1}\ \ldots\ b_2\ b_1\ b_0$$

と定義する．

ここで，英語月名（3文字省略語）のキー集合 K と ASCII による内部コード値の総和 w(k) を利用し，w(k) の後尾6ビット分のビット列をハッシュ値とした場合の例を表 2.1 に示す．

いま，H(k) の末尾の1ビットに対応する h_1(k) なるハッシュ関数を選んで，キー JAN, FEB, MAR を格納したハッシュ表を図 2.6-(a) に示す．但し，バケットに格納できるキーの最大数は2とする．h_1(k)=0 に対応するバケット1には MAR が，h_1(k)=1 に対応するバケット2には JAN と FEB が格納される．ここで，ハッシュ表の全域深さ（図中 A の値）はハッシュ表全体の番地計算に必要なビット数を表し，各バケットの局所深さ（図中 B の値）はそのバケットを他のバケットと区別するために必要なビット数を表す．

表 2.1　ハッシュ関数の例

k	w(k)	H(k)
JAN	217	011001
FEB	205	001101
MAR	224	100000
APR	227	100011
MAY	231	100111
JUN	237	101101
JUL	235	101011
AUG	221	011101
SEP	232	101000
OCT	230	100110
NOV	243	110011
DEC	204	001100

次に，図 2.6-(a) に対して，キー APR(100011)，h_1(APR)=1 を追加すると，バケット 2 であふれが生じる．そこで，2 ビット分のハッシュ関数 h_2(k) を採用して，ハッシュ表の大きさを 2 倍に成長させる（図 2.6-(b) ）．図 2.6-(b) では，番地 01 と 11 は唯一のバケットに対応するが，バケット 1 には複数の番地 00 と 10 が対応する．これは，バケット 1 が将来キーの追加で領域があふれたとき，番地 00 と 01 にバケットを分割できることを意味する．更に，MAY(100111) と JUN(101101) の追加を考えるとき，MAY はバケット 3 に追加できるが，JUN はバケット 2 に追加できない．しかも，バケット 2 の局所深さと全域深さはともに 2 で等しいので，バケットの分割ができない．従って，3 ビット分のハッシュ関数 h_3(k) を採用し，再びハッシュ表を成長させて，JUN を追加する（図 2.6-(c) ）．

このように，ハッシュ表はハッシュ関数値の伸縮に伴って大きくなるが，ハッシュ表が大きくなるにつれて各バケットの局所的深さはバラバラになり，ハッシュ表も冗長になる．従って，ハッシュ表をビット列からなる 2 進木（binary tree）構造で表現する場合が多い [33, 24]．この構造はトライ（trie）ハッシュと呼ばれる．図 2.6-(c) に対して，残りの全てのキーを追加したトライによる拡張ハッシュ表を図 2.7 に示す．

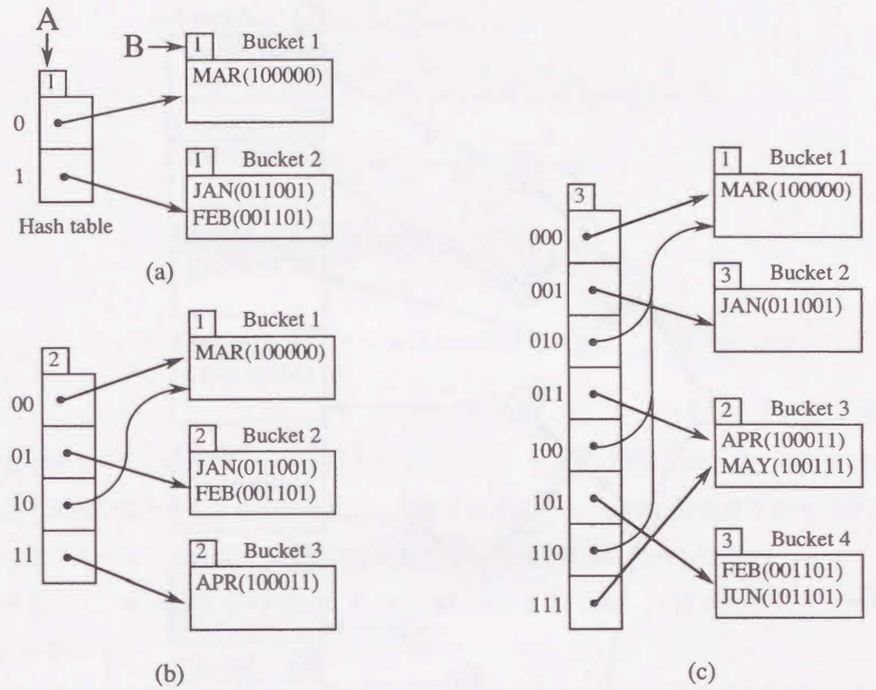


図 2.6　拡張ハッシュ表の例

図 2.7 の例では，全域深さは 5，最小の局所深さはバケット 3 の 2 であり，本来ならばバケット 3 には 25-2=23=8 個の番地が対応するはずであるが，図 2.7 ではノード 6 だけがバケット 3 に連結されている．キー OCT(100110) を検索してみると，まずルートノード 1 よりスタートし，最後尾のビット 0 の枝（ノード 1 から 2 ）を辿り，次の 2 番目のビット 1 に対する枝をノード 2 から 6 に辿ると，トライの葉を経由してバケット 3 にアクセスし，キー OCT が発見される．

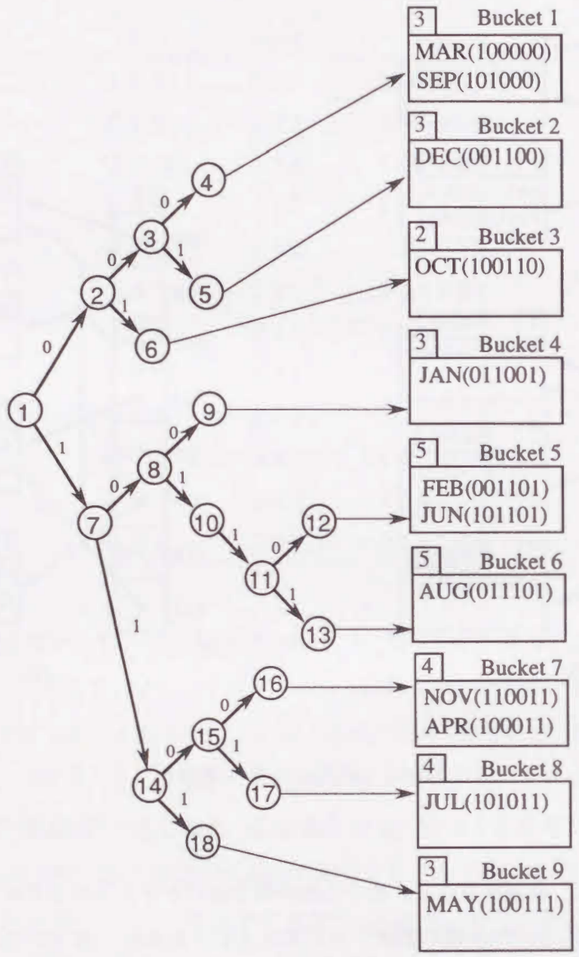


図 2.7　キー集合Ｋに対するトライハッシュ構造

2.4　トライ法

トライ（Trie）[28, 1]は，キー集合Ｋの各キーの共通接頭辞を併合して作られる木構造であり，そのアークラベルにはキーを構成する文字や数字が対応する．ハッシュ法や２分

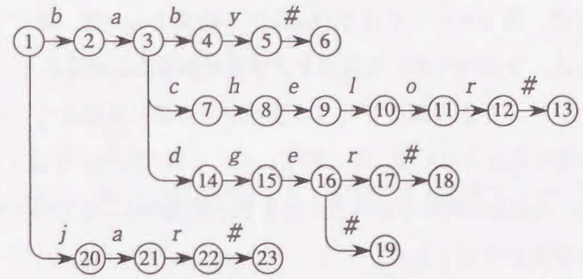


図 2.8　キー集合Ｋに対するトライ

探索木法がキー全体を値として比較するのに対して，トライの探索では，キーの１文字１文字を比較対象とする．この特徴より，トライはプログラミング言語処理系 [2] や自然言語処理システム [53] における文字列を対象とした語彙の検索に向いている．

例として，キー集合 K={baby♯, bachelor♯, badge♯, badger♯, jar♯} に対するトライを図 2.8 に示す．

キー集合Ｋにおいて，記号‘♯’は，トライの葉とキーを１対１に対応させるために，キー自信には含まれない端記号（endmarker）とする．また，ノード n からノード m に対してアークラベル‘a’が定義されていることを関数 g を用いて，g(n,a)=m で表す．

トライでの検索例として，図 2.8 のトライ上で文字 “baby♯” を検索する場合を説明する．まず，ルートノード 1 から g(1,b)=2 なるラベル‘b’のアークが存在するから第１文字‘b’がマッチしたことになる．このような１文字単位の比較をノード 3 , 4 , 5 へと繰り返して葉ノード 6 まで到達すると，“baby♯” の検索は成功し，このキーはＫに含まれることが分かる．また，“bag♯” を検索するとノード 3 以降は辿れないので，検索は失敗し，“bag♯” はＫに含まれないことが分かる．

トライには，大きく三つの特徴があり，まず，一つ目の特徴は，入力文字列の左端より始まる全ての接頭辞（最左部分列と呼ぶ）が１回の入力走査で探索できることである．例えば，図 2.8 のトライで入力文字列 “badgers♯” を探索する場合，その接頭辞 “badgers” と “badge” が１回の走査で探索できる．但し，この探索を可能にするには，ノード１６と１７で g(16, ♯)=19 と g(17, ♯)=18 なるアークを確認する必要がある．

二つ目の特徴は，探索失敗の場合でも入力文字列と部分マッチする接頭辞が検出できることである．例えば，図 2.8 のトライで “bady♯” を検索するとき，部分マッチする “ba’ まで検索が進められ，ミスマッチの位置が 3 文字目であることが分かる．

三つ目の特徴は，ノードから出るアークの探索がその数に関係なく一定時間で検索できるデータ構造が用いられるならば，キー検索に対する最大時間計算量（worst-case time complexity）はキーの総数に関係なく，キーの長さに比例することである．即ち，トライでは非常に高速な検索が可能となる．

2.5　結　言

以上のように，現在までに様々な検索手法が考案されているが，それぞれに長所と短所を兼ね備えている．まず，2 分探索木法は順検索が容易に実現できるという長所を有するが，多くのディスクアクセスを要するため検索速度が遅いという短所を持つ．また，B^+ 木法は順検索が可能で，しかも 2 分探索木法においては短所であったディスクアクセス回数も少なくなる．しかしながら，検索を実行するために大きな索引部が必要となる．一方，動的ハッシュ法の代表的手法である拡張ハッシュ法は高速な検索能力があるが，順検索には不向きである．更に，ハッシュ表が大きくなるにつれて各バケットの局所的深さが不均一になり，ハッシュ表が冗長になるという短所を持つ．

このように，キー検索手法は，キーとレコードの更新が可能か否か，順検索が可能か否か，2 次記憶を使用するか否か等の条件に応じて，非常に多くの改善と拡張，また，新しい手法の提案が行われている．しかしながら，すべての分野にうまく適合する検索手法が存在しないため，ソフトウェア開発者は適材適所となるべき探索手法の設計に労力を払わなければならない．

そこで，次章では拡張ハッシュ法の欠点をカバーするために考案されたトライハッシュ法，即ち，ハッシュ表を 2 進木トライ構造で表現する手法の詳細，及び索引部となる 2 進木トライの圧縮法を紹介する．この手法を用いれば，コンパクトな索引部で高速な順検索が可能になる．

第3章

2 進ディジタル探索(BDS)木による検索手法

3.1　緒　言

拡張ハッシュ法は動的なキー集合に対して，高速な検索を実現する手法であるが，一般的に，キーに対する順検索には不向きであった．この欠点を解消するため，Jonge ら [24] はキーの 2 進数表現から成る 2 進ディジタル探索木（Binary Digital Search Tree：BDS 木と呼ぶ）をハッシュ表に用い，更に，BDS 木を先行順ビット列（pre-order bit stream）[24] と呼ばれるコンパクトなデータ構造に圧縮する手法を提案した．この手法により，B^+木法よりコンパクトな索引部で順検索を行うことが可能になった．そこで，本章では，Jonge らに基づき BDS 木を用いた拡張ハッシュ法（2 進木トライハッシュ法）の概要を説明した後，先行順ビット列による BDS 木の圧縮法，及び先行順ビット列上での検索，更新アルゴリズムを示す．まず，3．2 節で BDS 木をハッシュ表として用いた拡張ハッシュ法の解説を行う．次に，3．3 節では索引部となる BDS 木の先行順ビット列への圧縮法を説明する．そして，3．4 節，及び 3．5 節では先行順ビット列で表現された BDS 木の検索，及び更新アルゴリズムを示す．

3.2　BDS木を用いた拡張ハッシュ法

拡張ハッシュ法は，主記憶上にキー索引部であるハッシュ表を格納することで，2次記憶上にあるバケットへのアクセス回数を1回に抑えられる高速な検索技法である．キーを格納するバケット（固定長のバケットに格納可能なキーの数を B_SIZE とする）はバケット番号によって参照され，このバケット番号はハッシュ関数から求めたハッシュ値を用いてハッシュ表より決定される．ハッシュ表の大きさを動的に伸縮させるためには値域が変化するハッシュ関数が必要であるので，キーkを十分に大きい m 長のビット列に写像する次のハッシュ関数 H を定義する．

$$H(k) = b_0\ b_1\ b_2\ \ldots\ b_{m-1}$$

ここで，b_i $(0 \leq i \leq m-1)$ は，0または1のビットを表す．ここで，順検索を実現するためには，キー文字列の2進数表現をハッシュ値として用いることとなり，その結果，ハッシュ表はキーの2進数表現から成る2進木構造，即ち，BDS木となる．BDS木は，左の枝を0，右の枝を1とラベル付けし，葉に到達するまでの経路を，キーの2進数表現の上位ビット列で構成する．また，各葉には対応するバケットへのポインター，もしくはアドレスが格納される．このようにすれば，キーの内部コードが保存されるので，キーの値順を反映した構造になり，木を先行順走査で検索すれば，順検索が可能となる．

例えば，3文字の英単語をキーとし，キーを構成する文字の内部コード（Internal codes）値（a, b, c, ... , z の内部コード値をそれぞれ1, 2, ... , 26 とする）を各々5ビットの2進数表現にした関数 $H(k)$ を用いるとき，キー集合

K={cat, ear, job, pen, sea, sun, zoo}

に対する $H(k)$ は表 3.1 で与えられる．また，B_SIZE を2とした場合，キー集合Kに対するBDS木を図 3.1 に示す．

ここで，Jonge[24] らが採用したBDS木には，各ノードから必ず2本の枝が出ることを保証するため，1本の枝しか持たないノードにはダミーリーフが付加されている．このダミーリーフの採用によって，次のような2つのメリットが生じる．まず，ダミーリーフの採用により，BDS木内のすべてのノードが2本の枝を持てば，葉（外部ノード）の数が内部ノードの数よりも1つ多くなるという2進木の特徴が利用できる．即ち，BDS木内の

表 3.1　ハッシュ関数値の例

k	Internal codes	$H(k)$
c a t	3 / 1 / 20	00011 00001 10100
e a r	5 / 1 / 18	00101 00001 10010
j o b	10 / 15 / 2	01010 01111 00010
p e n	16 / 5 / 14	10000 00101 01110
s e a	19 / 5 / 1	10011 00101 00001
s u n	19 / 21 / 14	10011 10101 01110
z o o	26 / 15 / 15	11010 01111 01111

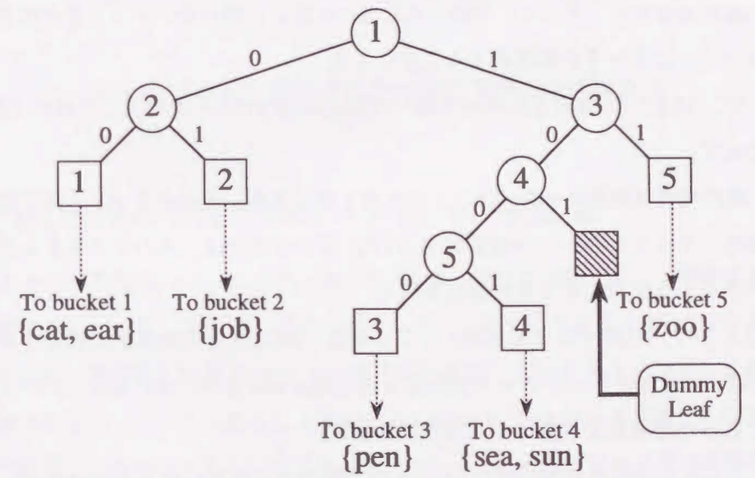


図 3.1　キー集合Kに対するBDS木

如何なる部分木においても葉の数が内部ノードの数より1つ多くなる．次に，検索キーがダミーリーフに辿り着いた時点で，そのキーが未登録キーと判定できる．その結果，ディスクアクセスが軽減でき，未登録キーの検索が高速に行える．尚，このダミーリーフは，バケットへのポインターをもたないので，ディスクの空間効率には影響しない．

BDS 木へのキー挿入操作は，

1. 挿入キーに対応する葉がダミーリーフ；
2. 挿入キーに対応するバケット容量に余裕がある；
3. 挿入キーに対応するバケット容量が満杯；

の場合に分類できる．

1．の場合は，挿入キーに対応する葉を非ダミーリーフに変更した後，その非ダミーリーフに対応した新たなバケットを生成し，そのバケットに挿入キーを格納する．

2．の場合は，挿入キーに対応するバケットに挿入キーを格納する．

3．の場合は，オーバーフローを起こしたバケットに対応する葉を内部ノードに変換した後，バケット内に格納されるべきキーの共通接頭部分を枝として生成し，その下に新たな2つの葉を追加する．そして，追加された葉に対応した新たなバケットをそれぞれ生成し，各バケットにキーを分類格納することになる．

例として，図 3.1 の BDS 木にキー “sit”（H(sit)=100110…）を挿入した後の BDS 木を図 3.2 に示す．

また，削除操作は削除キーをバケットから取り除いた後，そのバケットと同じ親ノードを持つ他のバケットとのキーの総和が B_SIZE 以下の場合は，木のサイズを小さくするため，バケット及び対応する葉の併合を行う．

以上のように，BDS 木による拡張ハッシュ法は，BDS 木を主記憶上に格納することを仮定すれば，バケットへのアクセス回数を 1 回に抑えることができ，更に，ハッシュ法では困難であった順検索を可能にする技法である．

更に，Jonge ら [24] は BDS 木のバケットの空間効率を改善した S（Separator）木も提案している．S 木での挿入操作は，1．の場合にダミーリーフを非ダミーリーフには変更せず，ダミーリーフの最も近い左隣の非ダミーリーフが指すバケットにキーを格納することによって，B_SIZE よりも極端に少ないキー数から成るバケットを抑えている．しかし，一般的に S 木では約 70%の葉がダミーリーフとなることが報告されている [24]．

このダミーリーフはバケットの空間効率には悪影響は与えないが，BDS 木自体の空間効率には悪影響を及ぼす．このように，ダミーリーフ数が増加すれば，木のサイズが大きくなり，空間効率が低下する．そこで，本論文ではダミーリーフを採用しない BDS 木をハッシュ表として用いる手法を **4．4** で提案する．

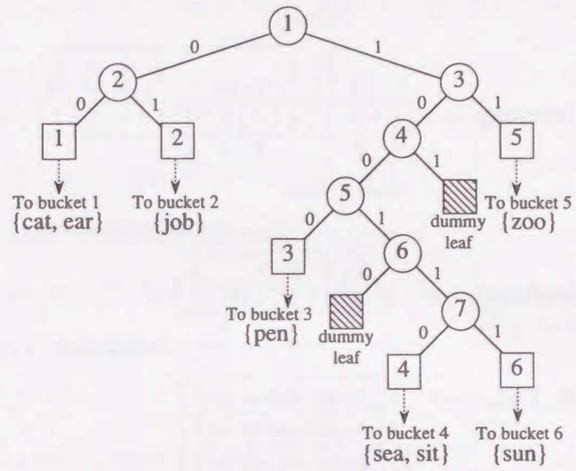


図 3.2　図 3.1 の BDS 木にキー “sit” を挿入後の BDS 木

3.3　BDS木から先行順ビット列への圧縮法

BDS 木を用いた拡張ハッシュ法は，従来のハッシュ法では困難であった順検索を可能にする手法である．しかしながら，通常のポインターを有する木構造で BDS 木を表現した場合，登録キー数が多くなると BDS 木のサイズが増加し，BDS 木を主記憶上に格納することが困難になる．ハッシュ表として用いられている BDS 木を 2 次記憶上に配置し，分割的に主記憶上に呼び出すように変更すれば，ハッシュ法の特徴である高速な検索能力が損なわれてしまう．そこで，Jonge ら [24] は BDS 木を先行順ビット列と呼ばれる非常にコンパクトなビット列に圧縮する手法を提案した．

先行順ビット列は treemap，及び leafmap と呼ばれる 2 つのビット列と B_TBL と呼ばれる 1 つのテーブルから構成される．まず，treemap は BDS 木の状態を表し，leafmap は対応する葉がダミーか非ダミーかを表す．また，B_TBL は各バケットのアドレスを格納したアドレス表である．treemap は BDS 木を先行順走査し，図 3.1 の BDS 木に示される ○印の内部ノードを通過時点でビット値 0，□印の外部ノードを通過時点でビット値 1 を出力したビット列である．leafmap も同様に先行順走査し，各葉を通過時にダミーリーフな

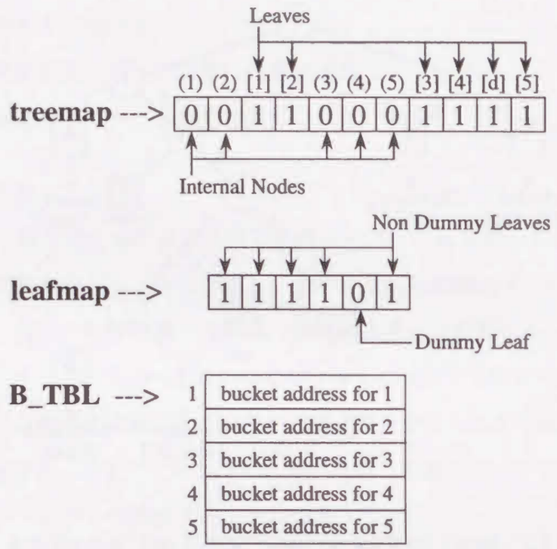


図 3.3　図 3.1 の BDS 木に対する先行順ビット列

らばビット値0を，非ダミーリーフならばビット値1を出力したビット列である．B_TBL も同様に先行順走査し，各葉に対応するバケットのアドレスを順次格納したテーブルである．図 3.1 の BDS 木に対する先行順ビット列を図 3.3 に示す．ここで，BDS 木との対応を示すため，( ) 内に内部ノード番号，[ ] 内に外部ノード番号（d はダミーリーフ）を treemap の上部に記述する．

3.4　先行順ビット列を用いた検索アルゴリズム

先行順ビット列で表現された BDS 木の検索は，先行順ビット列の先頭ビットから順番に値を調べるため，BDS 木を先行順走査しながら検索することになる．例えば，図 3.1 でキー “pen” を検索する場合，ノード 1 からスタートして，ノード 2，葉 1，2 を通過し，ノード 3，4，5 を経て葉 3 に辿り着いた後，対応するバケット 3 にアクセスすることによって，キー “pen” が検索される．また，葉を通過する度にバケットにアクセスすれば，

順検索が行える．

以下に先行順ビット列を用いた検索アルゴリズムを示す．但し，次のような変数を用いる．

key　：検索キー；

keypos　：H(key) における現在処理中のビット位置；

treepos　：treemap における現在処理中のビット位置；

leafpos　：leafmap における現在処理中のビット位置；

index　：B_TBL 内の検索すべきスロット番号；

【先行順ビット列を用いた BDS 木の検索アルゴリズム】

入力：*key*：検索すべきキー；

出力：検索成功ならば TRUE，失敗ならば FALSE；

手順 (S-1)　：{ 各変数の初期化 }

keypos ← 1，*treepos* ← 1，*leafpos* ← 1；

手順 (S-2)　：{ ハッシュ値の検証 }

H(*key*) の *keypos* 位置のビット値が 1 ならば手順 (S-3) に，

ビット値が 0 ならば手順 (S-4) に進む；

手順 (S-3)　：{ 左部分木のスキップ処理 }

treemap 上の *treepos* 位置から，1 のビット数が 0 のビット数より 1 つ多くなるまで *treepos* を進めた後，手順 (S-4) に進む；

手順 (S-4)　：{ 枝を 1 つ辿る処理 }

treepos の位置を 1 つ進めた後，treemap 上の *treepos* 位置のビット値が 0（内部ノード）ならば *keypos* の位置を 1 つ進め，手順 (S-2) に戻り，ビット値が 1（外部ノード）ならば手順 (S-5) に進む；

手順 (S-5)　：{ 葉の状態がダミーか非ダミーかの検証 }

treemap 上で先頭ビットから *treepos* 位置までに存在するビット値 1 のビット数をカウントし，その値を *leafpos* に代入する；

leafmap の *leafpos* 位置のビット値が 0（ダミーリーフ）ならば FALSE を返し，1（非ダミーリーフ）ならば手順 (S-6) に進む；

手順 (S-6)　：{バケットアドレスの獲得処理}

leafmap 上で先頭ビットから *leafpos* 位置までに存在するビット値が 1 のビット数をカウントし，その値を *index* に代入した後，$B\_TBL[index]$ を参照し対応するバケットのアドレスを獲得する；

手順 (S-7)　：{バケット内でのキーの検索}

得られたアドレスにより示されるバケット内に *key* が存在すれば TRUE，存在しなければ FALSE を返す；

上記のアルゴリズムにおいて，treemap 内では左部分木は親ノードの次に表現するので，左部分木に進む場合は手順 (S-4) のみを実行するが，右部分木に進む場合は左部分木をスキップする処理である手順 (S-3) が追加される．この処理では，如何なる部分木も葉の数がノードの数より 1 つ多いという 2 進木の特徴を利用し，左部分木の終了位置を求めている．また，手順 (S-5) では，treemap 内で値が 0 のビットを無視すれば，各値は leafmap 内の各値と 1 対 1 に対応するという先行順ビット列の特徴を活用し，葉の状態を検証している．更に，手順 (S-6) では，leafmap において，値が 0 のビットを削除すれば，各値が B_TBL 中のスロット番号と 1 対 1 に対応するという特徴を利用している．

ここで，例として図 3.3 の先行順ビット列からキー “pen” を検索する手順を図 3.4〜図 3.6 に従って，以下に示す．

手順 (S-1)　：*keypos*=1，*treepos*=1，*leafpos*=1；

手順 (S-2)　：H(pen)=1000... の第 1 ビット値が 1 なので手順（S-3）に進む；

手順 (S-3)　：図 3.1 示される BDS 木のノード番号 2 をルートとする部分木をスキップし，*treepos*=4 とする；

手順 (S-4)　：ノード番号 3 に進むため，*treepos*=5 とした後，treemap の第 5 ビット値が 0 なので，*keypos*=2 とし，手順（S-2）に戻る（図 3.4 ）；

手順 (S-2)　：H(pen)=1000... の第 2 ビット値が 0 なので手順（S-4）に進む；

手順 (S-4)　：ノード番号 4 に進むため，*treepos*=6 とした後，treemap の第 6 ビット値が 0 なので，*keypos*=3 とし，手順（S-2）に戻る；

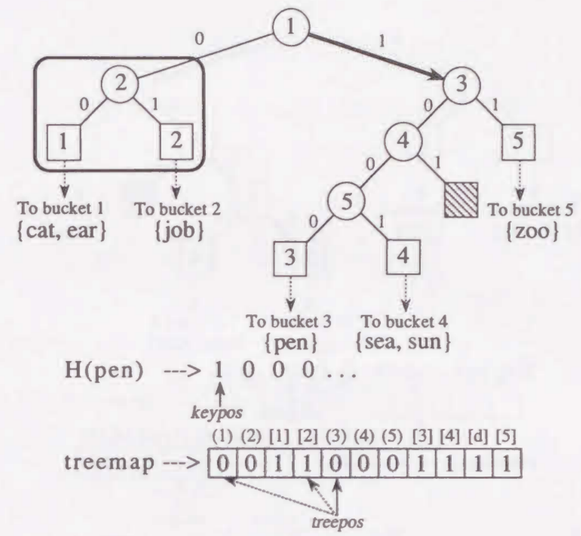


図 3.4　図 3.3 の先行順ビット列上でのキー “pen” の検索説明図 A

手順 (S-2)　：H(pen)=1000... の第 3 ビット値が 0 なので手順（S-4）に進む；

手順 (S-4)　：ノード番号 5 に進むため，*treepos*=7 とした後，treemap の第 7 ビット値が 0 なので，*keypos*=4 とし，手順（S-2）に戻る；

手順 (S-2)　：H(pen)=1000... の第 4 ビット値が 0 なので手順（S-4）に進む；

手順 (S-4)　：葉番号 3 に進むため，*treepos*=8 とした後，treemap の第 8 ビット値が 1 なので，手順（S-5）に進む（図 3.5 ）；

手順 (S-5)　：treemap の第 1 ビットから第 8 ビットまでに値が 1 のビットが 3 個存在するので，*leafpos*=3 とし，leafmap の第 3 ビット値が 1 なので手順（S-6）に進む；

手順 (S-6)　：leafmap の第 1 ビットから第 3 ビットまでに値が 1 のビットが 3 個存在するので，*index*=3 とし，$B\_TBL[3]$ からバケット 3 のアドレスを得る（図 3.6 ）；

手順 (S-7)　：バケット 3 に *key*=“pen” が含まれているので，TRUE を返す；

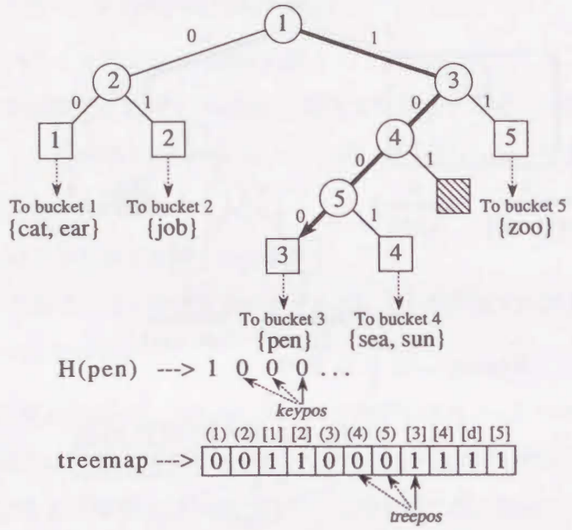


図 3.5　図 3.3 の先行順ビット列上でのキー “pen”の検索説明図 B

3.5　先行順ビット列を用いた更新アルゴリズム

先行順ビット列で表現されたBDS木の更新処理として，キー挿入後バケットがオーバーフローし，バケット分割を起こす処理について説明する．バケットがオーバーフローした場合，オーバーフローを起こしたバケット（フルバケットと呼ぶ）を指す葉を1つのノードと2つのダミーリーフから成る木（単位木と呼ぶ）に変換した後，フルバケット内のキー及び登録キーを単位木内のダミーリーフに割り振る，という操作をオーバーフローが起こらなくなるまで繰り返す．

先行順ビット列上では，単位木を表すビット列 “011” と単位木のダミーリーフを表すビット列 “00” を treemap，leafmap にそれぞれ適時挿入することになる．以下に更新アルゴリズムを示す．但し，以下のような変数，及び関数を用いる．

K_SET　：B_SIZE+1 個のキーから成るキー集合；

$full\_depth$　：フルバケットを示す葉までの木の深さ；

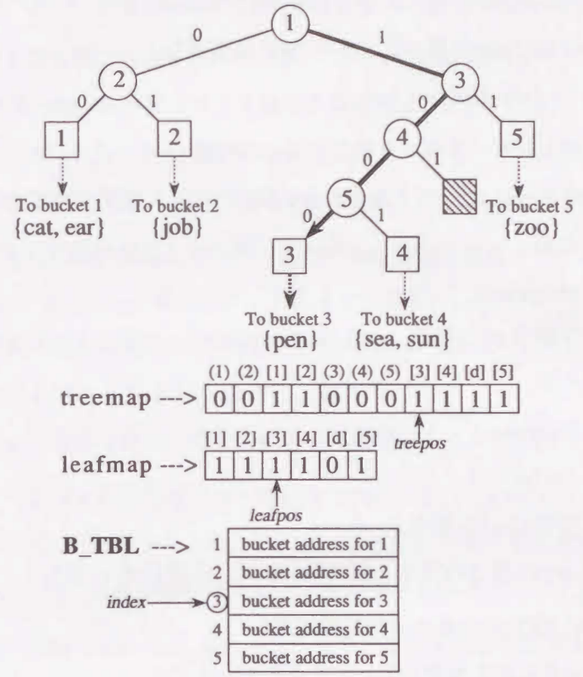


図 3.6　図 3.3 の先行順ビット列上でのキー “pen”の検索説明図 C

$StoreKey(flag, key)$　：$flag$ の値に従って，B_TBL を更新し，バケット内に key を格納する関数；

尚，更新は検索後に行われ，$treepos$, $leafpos$, $index$, $full\_depth$[1] はフルバケットに対応した位置を示しているものとする．

【先行順ビット列を用いたBDS木の更新アルゴリズム】

入力：key：検索の後，登録されるキー；

出力：なし；

手順 (I-1)　：{ フルバケット内のクリア }

フルバケット内のキーと登録キーを K_SET に代入し，フルバケット内を空にする；

[1] $full\_depth$ は手順 (S-4) でカウントすることによって設定する．

手順 (I-2)　：{ 単位木の挿入処理 }

treemap 上の *treepos* 位置のビット値を “011” に，

leafmap 上の *leafpos* 位置のビットを “00” に変更し，

$full\_depth \leftarrow full\_depth+1$；

手順 (I-3)　：{ 再オーバーフローを起こす場合の処理 }

K_SET 内のキーにおいて，*full_depth* 番目のビット値がすべて 0 ならば，左の枝を 1 つ辿るため，

$treepos \leftarrow treepos+1$

とした後，手順 (I-2) に戻り，*full_depth* 番目のビット値がすべて 1 ならば，右の枝を 1 つ辿るため，

$treepos \leftarrow treepos+2$

$leafpos \leftarrow leafpos+1$

とした後，手順 (I-2) に戻る；

また，*full_depth* 番目のビット値が異なれば，手順 (I-4) に進む；

手順 (I-4)　：{*K_SET* 内の各キーの格納処理 }

K_SET からキーを 1 個取り出し，それを *key* とする；

H(*key*) の *full_depth* 位置のビット値が 0 ならば，*StoreKey*(“left”, *key*) を行い，

ビット値が 1 ならば，*StoreKey*(“right”, *key*) を行う；

手順 (I-5)　：{ 終了判定処理 }

$K\_SET=\emptyset$ ならば処理を終了し，そうでなければ手順 (I-4) に戻る；

次に，関数 *StoreKey*(*flag*, *key*) の処理手順を示す．但し，以下のような変数を用いる．

full_bucket　：フルバケットのアドレス；

free_bucket　：バケットのために用意された未使用領域の先頭アドレス；

【関数 *StoreKey*(*flag*, *key*) のアルゴリズム】

入力：*flag*：左右どちらの葉に対応したバケットに登録するかを示すフラグ；

key：検索の後，登録されるキー；

出力：なし；

手順 (sk-1)　：{*flag* の判定処理 }

flag=“left” ならば手順 (sk-2) を，*flag*=“right” ならば手順 (sk-3) を行う；

手順 (sk-2)　：{ 左の葉に対応したバケットへの *key* の格納処理 }

leafmap 上の *leafpos* 番目のビット値を 1 に変更し，

$B\_TBL[index] \leftarrow full\_bucket$ とした後，

full_bucket で参照されるバケットに *key* を格納する；

手順 (sk-3)　：{ 右の葉に対応したバケットへの *key* の格納処理 }

leafmap 上の *leafpos*+1 番目のビット値を 1 に変更し，

B_TBL 内の *index*+1 番目以降の要素を 1 つ後方にずらし，

$B\_TBL[index+1] \leftarrow free\_bucket$ とした後，

free_bucket で参照されるバケットに *key* を格納する；

次に，例として図 3.3 の先行順ビット列にキー “sit”（H(sit)=100110...）を登録する処理手順を図 3.7 に従って説明する．尚，キー “sit” を検索した後の各変数値は，*treepos*=9，*leafpos*=4，*index*=4，*full_depth*=4 とし，フルバケットはバケット 4 とする．

手順 (I-1)　：*K_SET*={sea, sun, sit} とし，フルバケット内を空にする；

手順 (I-2)　：葉番号 4 のノードを単位木に変換するため，treemap の第 9 ビット値を “011” に，leafmap の第 4 ビット値を “00” に変更し，木の深さを *full_depth*=5 とする（図 3.7-(a)）；

手順 (I-3)　：*K_SET* 内の各キーのハッシュ値は，

H(sea)=1 0 0 1 1 0 ...

H(sun)=1 0 0 1 1 1 ...

H(sit) =1 0 0 1 1 0 ...

であり，すべてのキーの第 5 ビット値が 1 なので，手順（I-2）で追加された単位木の右の葉に対応するバケットが再オーバーフローする．そこで，単位木の右の葉に進むため，*treepos*=11，*leafpos*=5 とし，手順（I-2）に戻る（図 3.7-(b)）；

手順 (I-2)　：辿った右の葉を再度，単位木に変換するため，treemap の第 1 1 ビット値を “011” に，leafmap の第 5 ビット値を “00” に変更し，木の深さを *full_depth*=6 とする（図 3.7-(b)）；

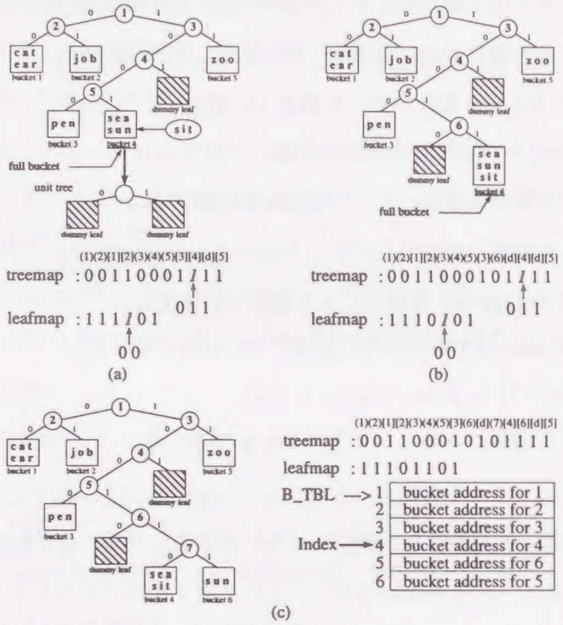


図 3.7　図 3.3 の先行順ビット列へのキー “sit” 追加説明図

手順 (I-3)　：K_SET 内の各キーのハッシュ値の第 6 ビット値が異なるので，手順（I-4）に進む；

手順 (I-4)　：key=sea（H(sea)=100110...）とし，H(sea) の第 6 ビット値が 0 なので，$StoreKey$(“left”, “sea”) が行われる；

手順 (sk-1)　：$flag$=“left” なので，手順（sk-2）を行う；

手順 (sk-2)　：leafmap の第 5 ビット値を 1 に変更し（leafmap=11101001），$B\_TBL[4]$ にバケット 4 のアドレスを代入した後，バケット 4 にキー “sea” を格納する；

手順 (I-5)　：K_SET={sun, sit}$\neq \emptyset$ なので，手順（I-4）に戻る；

手順 (I-4)　：key=sun（H(sun)=100111...）とし，H(sun) の第 6 ビット値が 1 なので，$StoreKey$(“right”, “sun”) が行われる；

手順 (sk-1)　：$flag$=“right” なので，手順（sk-3）を行う；

手順 (sk-3)　：leafmap の第 6 ビット値を 1 に変更し（leafmap=11101101），$B\_TBL[5]$ 以降のスロットの内容を 1 つ後方にスライドした後，$B\_TBL[5]$ に新たなバケット 6 のアドレスを代入し，そして，バケット 6 にキー “sun” を格納する；

手順 (I-5)　：K_SET={sit}$\neq \emptyset$ なので，手順（I-4）に戻る；

手順 (I-4)　：key=sit（H(sit)=100110...）とし，H(sit) の第 6 ビット値が 0 なので，$StoreKey$(“left”, “sit”) が行われる；

手順 (sk-1)　：$flag$=“left” なので，手順（sk-2）を行う；

手順 (sk-2)　：バケット 4 にキー “sit” を格納する；

以上，キー “sit” を挿入後の BDS 木，及び対応する先行順ビット列を図 3.7-(c) に示す．

3.6　結　言

BDS 木を用いた拡張ハッシュ法は，ハッシュ法の高速な検索能力を継承しつつ，従来のハッシュ法ではキー集合の性質を事前に調べておかなければ困難であった順検索を可能にする検索技法である．また，索引部を形成する BDS 木を先行順ビット列に圧縮すれば，B^+木法よりコンパクトな索引部となるので，索引部全体を主記憶上に格納できる．しかしながら，先行順ビット列に従来の BDS 木の構造をそのまま適用すると，大規模なキー集合に対してはビット列が非常に長くなり，その結果，時間効率及び空間効率の面で問題点が発生する．

そこで，次章では BDS 木を先行順ビット列で表現した場合の時間的問題点，及び空間的問題点を明らかにした後，それぞれの問題点を解決する手法を提案する．更に，提案手法の有効性を実証するため，理論的評価，及び具体的実験結果を示す．本手法を用いることにより，大規模なキー集合に対しても使用領域を抑えつつ，しかも高速な検索が維持できる．

第4章

2進ディジタル探索(BDS)木の改善

4.1 緒 言

Jongeら[24]の手法を用いると，BDS木を先行順ビット列と呼ばれるコンパクトなデータ構造に圧縮することができる．しかし，膨大なキー集合に対してはビット列が長くなり，検索・更新に対する時間効率の低下が重要な問題となる．また，圧縮に際し，BDS木のすべてのノードが2本の枝を有することを保証するため，1本の枝しか持たないノードにはダミーリーフを採用している．このダミーリーフはバケットの空間効率には影響を与えないが，BDS木の大きさ，即ち，先行順ビット列の長さには悪影響を及ぼす．そこで，本章では，順検索が可能で，しかも，上記の問題点を解決する新しいBDS木の構造を提案する．更に，この構造に対して先行順ビット列を用いたキーの検索・追加アルゴリズムも提案する．本手法では時間的問題点を解決するために，BDS木を階層的に分割し，分割された木構造単位で先行順ビット列を効率的に管理する．一方，空間的問題点に対しては，ダミーリーフを用いずにすべてのノードが2本の枝を有することを保証するため，1本の枝しか持たないノードをすべて削除した新たなBDS木構造に対して，木の状態を表すtreemap，削除されたノードの状態を示すnodemapから成る新しい形式の先行順ビット列に圧縮する．また，本手法の有効性を確認するため，日本語名詞5万語，および英単語5万語に対して行った実験結果を示す．

4.2　圧縮されたBDS木の問題点

先行順ビット列はBDS木を非常にコンパクトなビット列に圧縮できるデータ構造であるが，通常のBDS木をそのまま先行順ビット列に適用すれば，数々の問題点が発生する．特に，大規模なキー集合に対して，サイズが大きくなったBDS木を先行順ビット列に変換すれば，非常に長いビット列が生成され，その結果，各種処理に対する時間効率，及び空間効率の問題点が露呈する．以下，これら時間・空間効率に関する問題点を個別に明らかにする．

4.2.1　時間効率に関する問題点

先行順ビット列で表現されたBDS木は非常にコンパクトなデータ構造であるが，登録キー数が多くなると，ビット列（treemapとleafmap）が非常に長くなり，その結果，ビット列の後方に位置するキーに対する検索の時間効率が低下する．また，追加・削除処理の際には，ビット列上での追加・削除位置以降のすべてのビットをシフトする必要があり，ビット列が長くなると，追加・削除処理の時間効率も低下する．例えば，図4.1に示すBDS木，および先行順ビット列において，検索処理の最悪のケースは最も右に位置する葉を検索する場合である．この場合，先行順ビット列ではtreemapとleafmapのすべてのビットを走査しなければならない．一方，追加・削除処理における最悪のケースは最も左に位置する葉に対して追加・削除を行う場合である．この場合，キーの追加・削除処理に伴って，バケットの分割・併合が行われると仮定すると，treemapとleafmapの対応するビット位置以降のビットを後方にシフトしなければならない．

このように，通常のBDS木から生成された先行順ビット列には，登録キー集合が大規模になると，その長さが非常に長くなり，その結果，各種処理に対する時間効率が低下するという欠点がある．最悪の場合，検索では全ビットを走査し，挿入と削除では，全ビットの移動を行う必要があるので，大きなBDS木では，このビット処理時間に対する改善が必要となる．

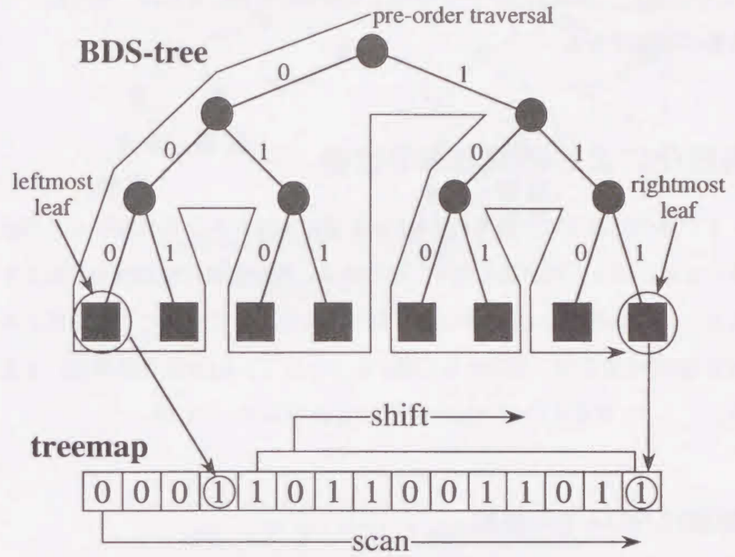


図 4.1　最悪の場合のBDS木の検索・追加処理例

4.2.2　空間効率に関する問題点

先行順ビット列上で適切なキー検索を行うためには，BDS木にダミーリーフを付加する必要がある．1本の枝しか持たない各ノードに対してダミーリーフを付加し，BDS木のすべてのノードが2本の枝を持つことを保証すれば，如何なる部分木も葉の数がノードの数より1つ多いという2進木の特徴が利用できる．この特徴が先行順ビット列を用いたBDS木の検索アルゴリズムの基礎となっている．

このダミーリーフはバケットへのポインターやアドレスを持たないため，バケットの空間効率には影響を及ぼさない．しかしながら，先行順ビット列の長さには悪影響を及ぼす．treemapはダミーリーフも含めた葉の数だけ1のビットが存在するため，ダミーリーフ数が増加すると，それだけビット列が長くなる．特に，leafmapは各葉の状態，即ち，ダミーリーフか非ダミーリーフかの区別をしめすビット列なので，ダミーリーフがなければ，leafmap自体を生成する必要もなくなる．一般的に，ダミーリーフ数は全外部ノード

数の40～60%を占めることが報告されている [47]．以上のように，ダミーリーフを付加せずにBDS木を圧縮し，圧縮したデータ構造上で検索が可能になれば，先行順ビット列のサイズは大幅に削減できる．

4.3　階層化による時間効率の改善

4．2．1で述べたように，通常のBDS木を基に生成された先行順ビット列は，登録キー数が多くなるとビット列が長くなり，その結果，各種処理の時間効率が低下するという欠点がある．この問題点はBDS木全体を処理の対象としていることに原因があり，何だかの方法でBDS木を分割・管理する必要がある．以下，BDS木の階層化による改善法を説明する．

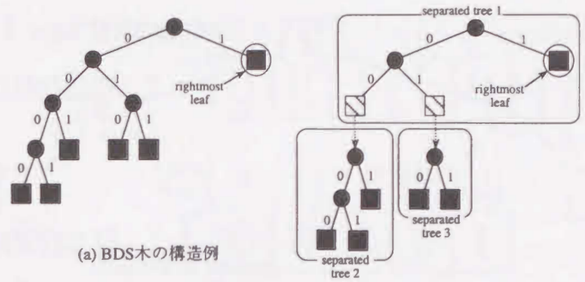
4.3.1　階層化BDS木の概要

先行順ビット列により表現されたBDS木の時間効率に関する問題点を解決するため，BDS木の階層化を行う改善手法を提案する．

BDS木の分割は，木の深さが一定値に達した時点で行い，この深さを分割深さ（Separation depth），また，分割された各木構造を分割木（Separated tree）と呼ぶ．更に，各分割木は番号付けされており，ポインタで繋がれている．以上のような方法で階層化されたBDS木を階層化BDS木（Hierarchical Binary Digital Search Tree：HBDS木）と呼ぶ．

このように，一定の深さ（分割深さ）毎にBDS木を階層化すれば，左部分木をスキップする際に走査するビット数，挿入・削除の際にシフトすべきビット数を分割深さに対応した一定の値に抑制できる．また，検索キーのハッシュ値（H(k)）の1ビット分がBDS木の1つの深さと対応しているため，BDS木と同様にハッシュ値も容易に分割・管理できる．図4.2に示す例では，図4.2-(a)の通常のBDS木で最右端の葉を検索する場合は，すべてのノードを走査しなければならないが，図4.2-(b)に示すように階層化を行うと（分割深さ=2），分割木2，3に含まれるノードへの走査を省くことができ，各種処理の高速化が計れる．

先行順ビット列を用いた階層化BDS木の検索アルゴリズムでは，分割深さをLとすると，以下のようにキーの2進数表現をLビットごとに分割する．



(a) BDS木の構造例　　(b) 階層化BDS木の構造例

図 4.2　階層化構造を用いたBDS木の改善例

$$H(\mathrm{k})=H_1(\mathrm{k})\ H_2(\mathrm{k})\ \ldots\ H_j(\mathrm{k})\ \ldots\ H_n(\mathrm{k})$$

ここで，$H_1(\mathrm{k})$ ～ $H_{n-1}(\mathrm{k})$ はLビット長，$H_n(\mathrm{k})$ はLビット長以下となる．

図4.3は，図3.1のBDS木の分割深さを2に設定した階層化BDS木である．例えば，図4.3の階層化BDS木でキー“sun”（H(sun)=1001...）を検索する場合には，キーのビット列を分割深さ以下になるように区切る（H(sun)=10 / 01 /...）．まず，1つ目の分割されたキー H_1(sun)=10 を用いて，分割木1を先行順走査する．このとき，外部ノード4に次の分割木2へのポインタがあるので，H_2(sun)=01 を用いて，分割木2を先行順走査する．ここで，葉4に到達するので，対応するバケット内に格納されているキーとの比較によって，キー“sun”が登録されていることを確認する．また，キー“zoo”（H(zoo)=11...）を検索する場合，従来のBDS木では木を全て走査する必要があったが，階層化BDS木では分割木1のみを走査するだけで葉5に到達し，キー“zoo”の検索が成功する．

この改良法により，探索に不必要な各分割木に対する先行順ビット列の走査が省けるので，検索は高速化でき，また更新・削除も関係する分割木のみを局所的に処理できるので，高速化が同様に可能となる．

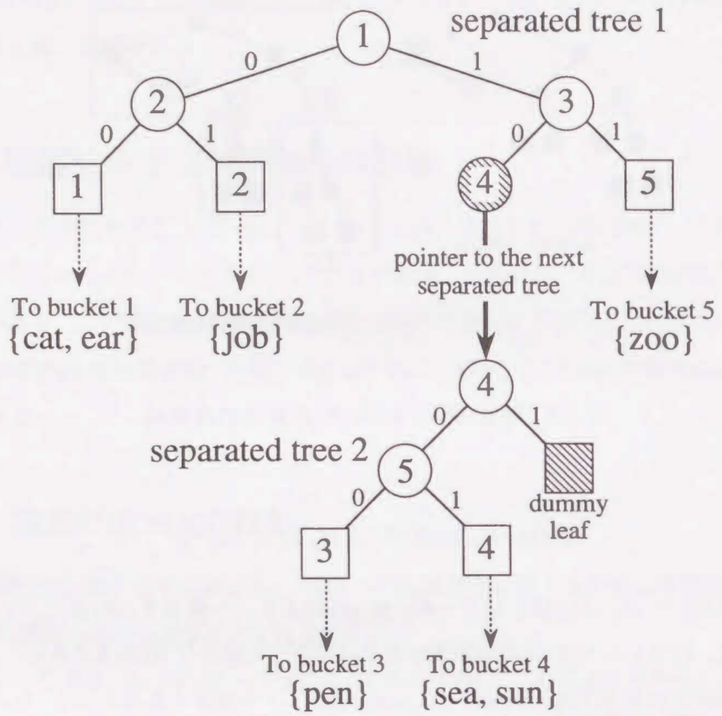


図 4.3　図 3.1 の BDS 木を基にした階層化 BDS 木

4.3.2　階層化 BDS 木の圧縮法

階層化 BDS 木の第 i 番目の分割木に対する先行順ビット列は，
分割木の状態を示す $treemap_i$；
葉の状態を示す $leafmap_i$；
バケットのアドレスを格納したバケット表 B_TBL_i から構成される．

但し，次の分割木へのポインタとなる内部ノードを葉とみなし，その葉に対応する $treemap_i$ と $leafmap_i$ のビット値は 1 とし，バケット表中には次の分割木番号にマイナスを付けた

$treemap_1$:

(1)	(2)	[1]	[2]	(3)	<4>	(5)
0	0	1	1	0	1	1

$treemap_2$:

(4)	(5)	[3]	[4]	[d]
0	0	1	1	1

$leafmap_1$:

1	1	1	1

$leafmap_2$:

1	1	0

B_TBL_1 :

1	bucket address for 1
2	bucket address for 2
3	- 2
4	bucket address for 5

B_TBL_2 :

1	bucket address for 3
2	bucket address for 4

図 4.4　図 4.3 の階層化 BDS 木に対する先行順ビット列

値を格納する．図 4.3 の階層化 BDS 木に対する先行順ビット列を図 4.4 に示す．ここで，$treemap_i$ の上部に記述される <> 内の値は，分割木へのポインタとなる内部ノード番号を表す．

4.3.3　階層化BDS木の検索アルゴリズム

以下に，先行順ビット列を用いた階層化BDS木の検索アルゴリズムを示す．但し，次のような変数を用いる．

key　：検索キー；

i　：現在探索している分割木の番号，初期状態として1に設定される；

$keypos$　：$H_i(key)$ における現在処理中のビット位置；

$treepos$　：$treemap_i$ における現在処理中のビット位置；

$leafpos$　：$leafmap_i$ における現在処理中のビット位置；

$index$　：B_TBL_i 内の検索すべきスロット番号；

$address$　：B_TBL_i より求められるバケットのアドレス；

【先行順ビット列を用いた階層化BDS木の検索アルゴリズム】

入力：key：検索すべきキー；

出力：検索成功ならばTRUE，失敗ならばFALSE；

手順 (S'-1)　：{ 各変数の初期化 }

$keypos \leftarrow 1$，$treepos \leftarrow 1$，$leafpos \leftarrow 1$；

手順 (S'-2)　：{ 分割されたハッシュ値の検証 }

$H_i(key)$ の $keypos$ 位置のビット値が1ならば手順 (S'-3) に，

ビット値が0ならば手順 (S'-4) に進む；

手順 (S'-3)　：{ 左部分木のスキップ処理 }

$treemap_i$ 上の $treepos$ 位置から，1のビット数が0のビット数より1つ多くなるまで $treepos$ を進めた後，手順 (S'-4) に進む；

手順 (S'-4)　：{ 枝を1つ辿る処理 }

$treepos$ の位置を1つ進めた後，$treemap_i$ 上の $treepos$ 位置のビット値が0（内部ノード）ならば $keypos$ の位置を1つ進め，手順 (S'-2) に戻り，ビット値が1（外部ノード）ならば手順 (S'-5) に進む；

手順 (S'-5)　：{ 葉の状態がダミーか非ダミーかの検証 }

$treemap_i$ 上で先頭ビットから $treepos$ 位置までに存在するビット値1のビット数をカウントし，その値を $leafpos$ に代入する；

$leafmap_i$ 上の $leafpos$ 位置のビット値が0（ダミーリーフ）ならばFALSEを返し，1（非ダミーリーフ）ならば手順 (S'-6) に進む；

手順 (S'-6)　：{ バケットアドレスの獲得処理 }

$leafmap_i$ 上で先頭ビットから $leafpos$ 位置までに存在するビット値が1のビット数をカウントし，その値を $index$ に代入した後，$B\_TBL_i[index]$ を参照し対応するバケットのアドレスを $address$ に獲得する；

得られた $address$ が負ならば手順 (S'-7)，正ならば手順 (S'-8) を行う；

手順 (S'-7)　：{ バケット内でのキーの検索 }

$address$ で参照されるバケット内に key が存在すればTRUE，存在しなければFALSEを返す；

先行順ビット列に表現された階層化BDS木の検索アルゴリズムは，基本的には従来のBDS木の検索アルゴリズムと同じである．しかしながら，通常は各バケットアドレスを蓄える B_TBL_i に各分割木の番号が負数として格納されているため，手順 (S'-6) で B_TBL_i からバケットアドレスを獲得する際に，負数ならば分割木番号を獲得した後，対応する分割木に検索処理を移す手順が追加される．

例として図 4.4 の先行順ビット列からキー "pen" を検索する手順を図 4.5〜図 4.8 に従って，以下に示す．但し，初期状態としてiの値は1とする．

手順 (S'-1)　：$keypos$=1，$treepos$=1，$leafpos$=1；

手順 (S'-2)　：H_1(pen)=10 の第1ビット値が1なので手順（S'-3）に進む；

手順 (S'-3)　：図 4.2 示される階層化BDS木において，分割木1のノード番号2をルートとする部分木をスキップし，$treepos$=4 とする；

手順 (S'-4)　：ノード番号3に進むため，$treepos$=5 とした後，$treemap_1$ の第5ビット値が0なので，$keypos$=2 とし，手順（S'-2）に戻る（図 4.5 ）；

手順 (S'-2)　：H_1(pen)=10 の第2ビット値が0なので手順（S'-4）に進む；

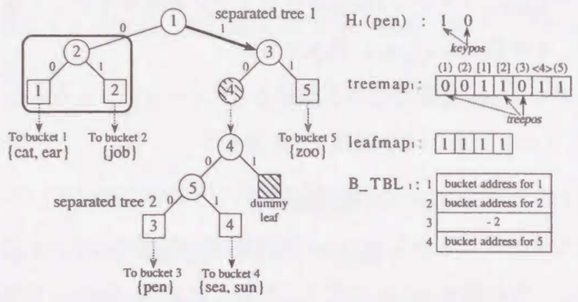


図 4.5　図 4.4 の先行順ビット列上でのキー “pen” の検索説明図 A

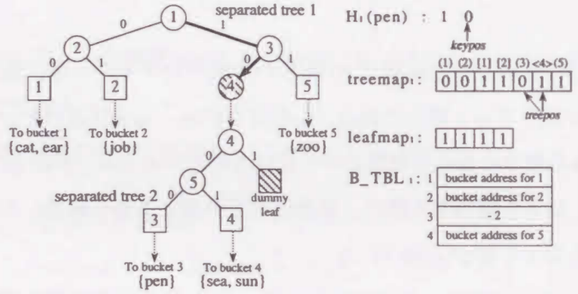


図 4.6　図 4.4 の先行順ビット列上でのキー “pen”の検索説明図 B

手順 (S'-4)　：ノード番号 4 に進むため，$treepos$=6 とした後，$treemap_1$の第 6 ビット値が 1 なので，手順（S'-5）に進む（図 4.6 ）；

手順 (S'-5)　：$treemap_1$の第 1 ビットから第 6 ビットまでに値が 1 のビットが 3 個存在し，更に，$leafmap_1$の第 3 ビット値が 1 なので手順（S'-6）に進む；

手順 (S'-6)　：$leafmap_1$の第 1 ビットから第 3 ビットまでに値が 1 のビットが 3 個存在するので，$index$=3 とし，$B\_TBL_1[3]$ を参照すると $address$=-2 を得るが，その値

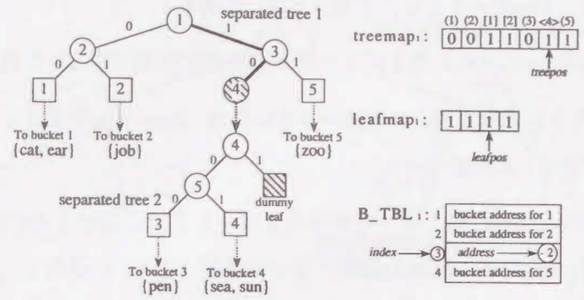


図 4.7　図 4.4 の先行順ビット列上でのキー “pen”の検索説明図 C

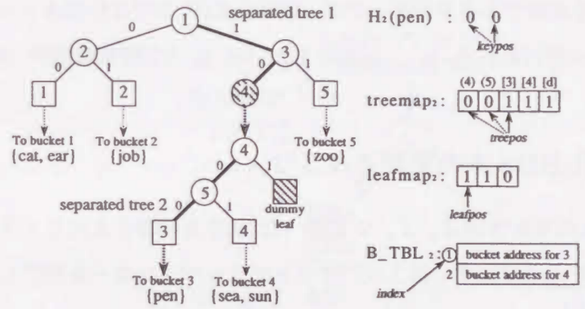


図 4.8　図 4.4 の先行順ビット列上でのキー “pen”の検索説明図 D

が負なので手順 (S'-7) を行う；

手順 (S'-7)　：i=2 に変更し，手順 (S'-1) に戻る（図 4.7 ）；
以後，分割木 2 に対して分割木 1 と同様の走査を行う；

手順 (S'-1)　：$keypos$=1，$treepos$=1，$leafpos$=1；

手順 (S'-2)　：H_2(pen)=00 の第 1 ビット値が 0 なので手順（S'-4）に進む；

手順 (S'-4)　：ノード番号5に進むため，$treepos$=2とした後，$treemap_2$の第2ビット値が0なので，$keypos$=2とし，手順（S'-2）に戻る；

手順 (S'-2)　：H_2(pen)=00の第2ビット値が0なので手順（S'-4）に進む；

手順 (S'-4)　：葉3に進むため，$treepos$=3とした後，$treemap_2$の第3ビット値が1なので，手順（S'-5）に進む；

手順 (S'-5)　：$treemap_2$の第1ビットから第3ビットまでに値が1のビットが1個あり，更に，$leafmap_2$の第1ビット値が1なので手順（S'-6）に進む；

手順 (S'-6)　：$leafmap_2$の第1ビット値が1なので$index$=1とし，$B_TBL_2[1]$からバケット3のアドレスを得る（図 4.8 ）；

手順 (S'-8)　：バケット3にkey="pen"が含まれているので，TRUEを返す；

また，特にキー "zoo"（H(zoo)=11...）の検索の場合，BDS木では先行順ビット列上のすべてのビットを走査する必要があったが，階層化BDS木では分割木1のみのビットを走査するがけで検索が行われるため，従来のBDS木に比べて検索時間が大幅に短縮される．

4.3.4　階層化BDS木の更新アルゴリズム

階層化BDS木の更新方法は，3．2で述べたBDS木の場合と同じく3種類に分類される．ここでは，その中でもキー挿入後バケットがオーバーフローを起こし，バケットを分割する場合の処理について説明し，その他の場合は，簡単のため説明を省略する．まず，図 4.3 の階層化BDS木にキー "sit"（H(sit)=10 / 01 / 10…）追加後の階層化BDS木を図 4.9 に示す．

登録キーを追加した後，バケットがオーバーフローを起こした場合，BDS木では次の操作をオーバーフローが起こらなくなるまで繰り返す．まず，オーバーフローを起こしたバケット（フルバケットと呼ぶ）に対応した外部ノードを単位木に変換した後，フルバケット内のキー及び登録キーを単位木内のダミーリーフに割り振る．これに対して，階層化BDS木においては，単位木を生成する際に，各分割木の高さが分割深さを越える毎に新しい分割木を生成する必要がある．

先行順ビット列を用いた階層化BDS木の更新処理は，単位木を表すビット列 "011" とダミーリーフを表すビット列 "00" を$treemap_i$，$leafmap_i$に適時挿入することになる．ま

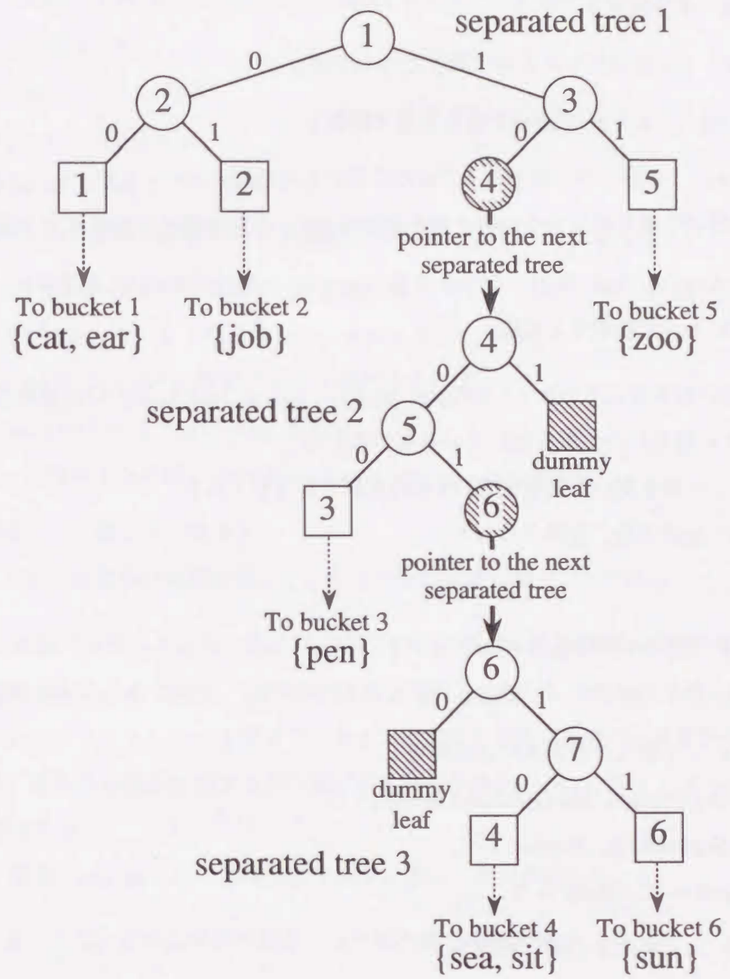


図 4.9　図 4.3 の階層化BDS木にキー "sit" を挿入後の階層化BDS木

た，木の深さが分割深さを越える毎に，新たな分割木に対する先行順ビット列を生成する必要がある．以下に階層化BDS木の更新アルゴリズムを示す．但し，以下のような変数，及び関数を用いる．

K_SET　：B_SIZE+1 個のキーから成るキー集合；

$sepa\_depth$　：分割深さ；

$full\_depth$　：フルバケットを示す葉までの木の深さ；

$MOD(n, m)$　：n を m で割った余りを返す関数；

$Separate(n)$　：n 番目の分割木の下に新たな分割木へのポインターを繋ぎ（B_TBL 内に次の分割木番号に負数を付けた値を格納する），その分割木の番号を返す関数；

$SepaStoreKey(n, flag, key)$　：$flag$ の値に従って，n 番目の B_TBL を更新し，バケット内に key を格納する関数；

尚，更新は検索後に行われ，i, $treepos$, $leafpos$, $index$, $full\_depth$[1] は検索の結果，フルバケットに対応した位置を示しているものとする..

【先行順ビット列を用いた階層化BDS木の更新アルゴリズム】

入力：key：検索の後，登録されるキー；

出力：なし；

手順 (I'-1)　：{ フルバケット内のクリア }

フルバケット内のキーと登録キーを K_SET に代入し，フルバケット内を空にする；

手順 (I'-2)　：{ 新たな分割木の生成処理 }

$MOD(full_depth, sepa_depth)$=0 ならば，

$i \leftarrow Separate(i)$，$treepos \leftarrow 1$，

$leafpos \leftarrow 1$，$index \leftarrow 1$；

手順 (I'-3)　：{ 単位木の挿入処理 }

$treemap_i$ 上の $treepos$ 位置のビット値を “011” に，

$leafmap_i$ 上の $leafpos$ 位置のビットを “00” に変更し，

$full\_depth \leftarrow full\_depth$+1 とする；

手順 (I'-4)　：{ 再オーバーフローを起こす場合の処理 }

K_SET 内のキーにおいて，$full\_depth$ 番目のビット値がすべて 0 ならば，左の枝を 1 つ辿るため，

[1] $full\_depth$ は手順 (S'-4) でカウントすることによって設定する．

$treepos \leftarrow treepos$+1

とした後，手順 (I'-2) に戻り，$full\_depth$ 番目のビット値がすべて 1 ならば，右の枝を 1 つ辿るため，

$treepos \leftarrow treepos$+2

$leafpos \leftarrow leafpos$+1

とした後，手順 (I'-2) に戻る；

また，$full\_depth$ 番目のビット値が異なれば，手順 (I'-5) に進む；

手順 (I'-5)　：{K_SET 内の各キーの格納処理 }

K_SET からキーを 1 個取り出し，それを key とする；

H(key) の $full\_depth$ 位置のビット値が 0 ならば，

$SepaStoreKey(i,$ “left”, $key)$ を行い，

ビット値が 1 ならば，$SepaStoreKey(i,$ “right”, $key)$ を行う；

手順 (I'-6)　：{ 終了判定処理 }

K_SET=∅ ならば処理を終了し，そうでなければ手順 (I'-5) に戻る；

上記の更新アルゴリズムは，BDS 木の更新アルゴリズムを各分割木の先行順ビット列に対して適用したものであり，基本的には同じである．しかしながら，手順（I'-2）において，フルバケットを指し示す葉までの深さが分割深さを越える度に木構造を分割する処理，即ち，新たな分割木に対する先行順ビット列の生成と，それに伴う B_TBL の更新処理が追加される．

次に，関数 $Separate(n)$ の処理手順を示す．但し，次の変数を用いる．

max_num　：現在までに使用されている分割木番号の最大値；

【関数 $Separate(n)$ のアルゴリズム】

入力：n：処理対象となる分割木の番号；

出力：ポインターで繋がれた新たな分割木の番号；

手順 (sp-1)　：{ 新たな分割木へのポインターを持つ内部ノードの設定 }

$leafmap_n$ 上の $leafpos$ 位置のビット値を 1 に変更する；

手順 (sp-2)　：{ 新たな分割木へのポインターを作成 }

$B\_TBL_n[index] \leftarrow$ -1 ×（max_num+1）；

手順 (sp-3)　：{ 分割木番号の最大値を更新 }

max_num の値を 1 つ増加した後，その max_num の値を返す；

また，関数 $SepaStoreKey(n, flag, key)$ の処理手順を以下に示す．但し，関数内で用いる変数 $full\_bucket$ および $free\_bucket$ は，関数 $StoreKey$ で用いたものと同じである．

【関数 $SepaStoreKey(n, flag, key)$ のアルゴリズム】

入力：$flag$：左右どちらの葉に対応したバケットに登録するかを示すフラグ；

key：検索の後，登録されるキー；

出力：なし；

手順 (ss-1)　：{$flag$ の判定処理 }

$flag$=“left”ならば手順 (ss-2) を，$flag$=“right”ならば手順 (ss-3) を行う；

手順 (ss-2)　：{ 左の葉に対応したバケットへの key の格納処理 }

$leafmap_n$ 上の $leafpos$ 番目のビット値を 1 に変更し，

$B\_TBL_n[index] \leftarrow full\_bucket$ とした後，

$full\_bucket$ で参照されるバケットに key を格納する；

手順 (ss-3)　：{ 右の葉に対応したバケットへの key の格納処理 }

$leafmap_n$ 上の $leafpos$+1 番目のビット値を 1 に変更し，

B_TBL_n 内の $index$+1 番目以降の要素を 1 つ後方にずらし，

$B\_TBL_n[index+1] \leftarrow free\_bucket$ とした後，

$free\_bucket$ で参照されるバケットに key を格納する；

次に，例として図 4.4 の先行順ビット列にキー “sit”（H(sit)=10 / 01 / 10 ... ）を登録する場合の各ビット値の変化の過程を図 4.10 ～図 4.12 に示し，その処理手順を説明する．尚，キー “sit”を検索した後の各変数値は，図 4.10 に示すように $treemap_2$，$leafmap_2$に対して，$treepos$=4，$leafpos$=2，$full\_depth$=4，$index$=2 となっている．但し，$sepa\_depth$ の値は 2 とする．

手順 (I'-1)　：K_SET={sea, sun, sit} とし，フルバケット（バケット 4 ）内を空にする；

手順 (I'-2)　：$MOD(full\_depth, sepa\_depth)=MOD(4, 2)=0$ より，$Separate(2)$ が行われる；

手順 (sp-1)　：$leafmap_2$ の $leafpos$ 位置，第 2 ビット値を 1 に変更する；

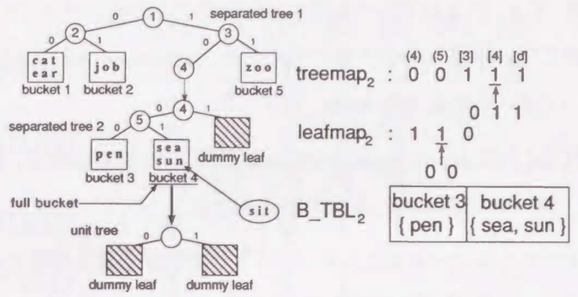


図 4.10　単位木の追加説明図

図 4.11　再オーバーフローの説明図

手順 (sp-2)　：max_num=2 とすると，B_TBL_2 の $index$ 番目のスロットを $B\_TBL_2[2]$=-1 ×（2+1）=-3 に変更する；

手順 (sp-3)　：max_num=3；

手順 (I'-2)　：i=3，$treepos$=1，$leafpos$=1，$index$=1 に変更される；

手順 (I'-3)　：分割木 3 として単位木が生成され，

$treemap_3$=011，$leafmap_3$=00，$full\_depth$=4+1=5 とする；

手順 (I'-4)　：K_SET 内の各キーのハッシュ値において，すべてのキーの第5ビット値が1なので，手順 (I'-3) で追加された単位木の右の葉に対応するバケットが再オーバーフローする．そこで，単位木の右の葉に進むため，$treepos$=1+2=3，$leafpos$=1+1=2とし，手順 (I'-2) に戻る (図 4.11)；

手順 (I'-2)　：$MOD(full\_depth, sepa\_depth)$=$MOD(5, 2) \neq 0$ なので，新たな分割木は生成されず，同一の分割木内に単位木が追加される；

手順 (I'-3)　：再オーバーフローを起こしたバケットに対応する葉を単位木に変換するため，$treemap_3$の $treepos$ 位置である第3ビット値を "011" に，また $leafmap_3$の $leafpos$ 位置である第2ビット値を "00" に変更し，更に木の深さを $full\_depth$=5+1=6 とする；

手順 (I'-4)　：K_SET 内の各キーのハッシュ値の第6ビット値が異なるので，手順 (I'-5) に進む；

手順 (I'-5)　：key=sea (H(sea)=100110...) とし，H(sea) の第6ビット値が0なので，$SepaStoreKey$(3, "left", "sea") が行われる；

手順 (ss-1)　：$flag$="left"なので，手順 (ss-2) を行う；

手順 (ss-2)　：$leafmap_3$の $leafpos$ 位置である第2ビット値を1に変更し ($leafmap_3$=010)，B_TBL_3の $index$ 番目のスロット，B_TBL_3[1] にバケット4のアドレスを代入した後，バケット4にキー "sea"を格納する；

手順 (I'-6)　：K_SET={sun, sit}$\neq \emptyset$ なので，手順 (I'-5) に戻る；

手順 (I'-5)　：key="sun" (H(sun)=100111...) とし，H(sun) の第6ビット値が1なので，$SepaStoreKey$(3, "right", "sun") が行われる；

手順 (ss-1)　：$flag$="right"なので，手順 (ss-3) を行う；

手順 (ss-3)　：$leafmap_3$の $leafpos$+1 位置である第3ビット値を1に変更し ($leafmap_3$=011)，B_TBL_3の $index$+1 番目のスロット，B_TBL_3[2] に新たなバケット6のアドレスを代入し，そして，バケット6にキー "sun"を格納する；

手順 (I'-6)　：K_SET={sit}$\neq \emptyset$ なので，手順 (I'-5) に戻る；

手順 (I'-5)　：key="sit" (H(sit)=100110...) とし，H(sit) の第6ビット値が0なので，$SepaStoreKey$(3, "left", "sit") が行われる；

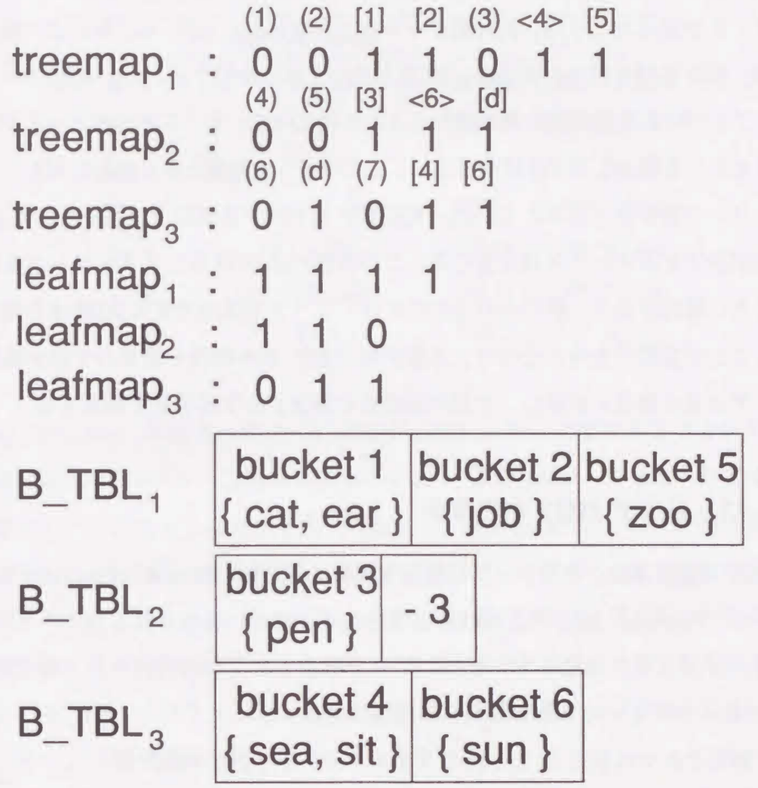


図 4.12　図 4.4 の先行順ビット列へのキー "sit" 追加説明図

手順 (ss-1)　：$flag$="left"なので，手順 (ss-2) を行う；

手順 (ss-2)　：バケット4にキー "sit"を格納する；

以上，キー "sit"を挿入後の階層化 BDS 木，及び対応する先行順ビット列を図 4.12 に示す．

4.4　パトリシアトライへの拡張による空間効率の改善

4．2．2で述べたように，先行順ビット列で表現されたBDS木において，適切な検索，追加，削除処理を行うためには，BDS木内の1本しか枝を持たない内部ノードにダミーリーフと呼ばれる擬似的な葉を持たせることによって，すべての内部ノードが2本の枝を有することを保証しなければならない．ところが，大規模なキー集合に対しては，このダミーリーフ数が多くなることから，先行順ビット列が必要以上に長くなってしまい，空間効率が低下するという欠点が生じる．この問題点はBDS木にダミーリーフを採用していることに原因があり，何だかの方法でダミーリーフを採用せずにBDS木を効率的に圧縮することが必要である．そこで，本節では，BDS木の構造を通常の2進木構造からパトリシア2進木構造に拡張し，上記の問題点を解決する改善手法を提案する．

4.4.1　パトリシアBDS木の概要

一般的に，BDS木は，そのノードの削除方法により，Full BDS木，Ordinary BDS木，パトリシア（Patricia）BDS木と呼ばれる3種類の木構造に分類される[17]—．

例えば，3文字の英単語をキーとし，キーを構成する文字の内部コード値（Internal codes）を各々5ビットの2進数表現を木構造に登録するとき（但し，a, b, c, ... , z の内部コード値をそれぞれ0, 1, 2, ... , 25とする），次のようなキー集合K'

K'={air, art, bag, bus, tea, try, zoo}

に対する，各キーの2進数表現（Binary sequences）は表4.1で与えられ，このキー集合K'から生成される上記3種類のBDS木を図4.13に示す．

まず，図4.13-(a) のBDS木はFull BDS木と呼ばれ，登録されているすべての文字（2進数値）が各ノード間の枝の値としてラベル付けされている．次に，図4.13-(b) のBDS木はOrdinary BDS木と呼ばれ，葉（外部ノード）から遡って分岐が起こらないノードをすべて削除した構造を持つ．削除されたシンボル（各ノード間の値）は，通常，各キーのレコード情報として，2次記憶上に格納され，このレコード情報へのポインターは各葉が格納している．最後に，図4.13-(c) のBDS木をパトリシアBDS木と呼ぶ．パトリシアBDS木はOrdinary BDS木から1本の枝しか持たないノードをすべて削除することにより形成されている．このパトリシアBDS木構造内で適切な検索を行うためには，

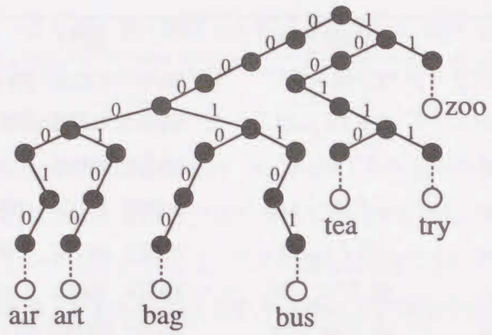
表4.1　キー集合K'の2進数表現

k	Internal codes	H(k)
air	0 / 8 / 17	00000 01000 10001
art	0 / 17 / 19	00000 10001 10011
bag	1 / 0 / 6	00001 00000 00110
bus	1 / 20 / 18	00001 10100 10010
tea	19 / 4 / 0	10011 00100 00000
try	19 / 17 / 24	10011 10001 11000
zoo	25 / 14 / 14	11001 01110 01110

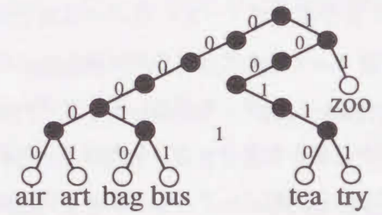
Ordinary BDS木から削除されたノード（Eliminated nodes）の数を示すカウンター，もしくは削除されたシンボル（Eliminated symbols）を蓄えたレコード情報へのポインターを各内部ノードに持たせる必要がある．削除ノード数を各内部ノードに持たせる場合には，検索のために各内部ノードが辿られる度に，削除ノード数のビット分だけ検索キーの2進数列をスキップすることになる．更に，葉に辿り着いた後で，葉が指し示す文字列と実際の検索キー文字列とのパターンマッチング処理が必要となる．一方，各内部ノードが削除シンボルへのポインターを保有している場合には，各内部ノードを辿る度に，削除シンボルとキー文字列の比較を行わなければならない．

例えば，図4.13-(b) に示されるOrdinary BDS木において，ルートノードからキー“air”に辿る経路には，5個の内部ノード（但し，ルートノードを除く）が存在する．これらの内，枝を1本しか有さないノードは，レベル1，2，3の3ノードである．これらの3ノードは，図4.13-(c) のパトリシアBDS木上では，ルートノードからキー“air”に辿る経路の第1レベルのノードに相当する．そのため，このパトリシアBDS木上でキー“air”を検索するためには，第1レベルの左ノードが削除された3個のシンボル“000”に関する情報を保持する必要がある．また，実際にパトリシアBDS木上でキー“air”を検索する場合には，キー“air”の2進数表現が“000000...”なので，第2ビット値の検証に至った時点で，第2ビットから第5ビットまでスキップし，更にスキップした3ビット値“000”と第1レベルの左ノードが保持している削除シンボル値“000”との比較を行う必要がある．

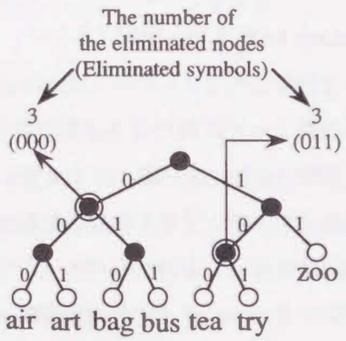
このように，通常のパトリシアBDS木では，他の2種類の木構造に比べて，木の深さ



(a) Full BDS-tree



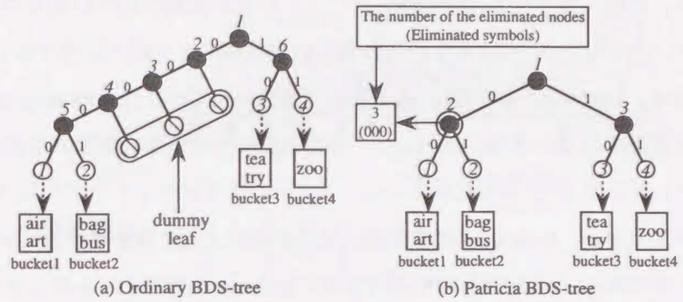
(b) Ordinary BDS-tree



(c) Patricia BDS-tree

図 4.13　BDS 木の 3 種類の構造説明図

が浅くなり，ノード数も激減するが，補助的に用いる記憶量，即ち，削除ノードに関する情報のために用意すべき記憶量や削除ノードをマッチングするための処理が追加されるので，一概に効率的な木構造とは言えない．しかしながら，パトリシア BDS 木では常にすべてのノードが 2 本の枝を持つことが保証されているので，本手法では，この特徴を利用し，パトリシア BDS 木を効率的に先行順ビット列に圧縮する手法を提案する．



(a) Ordinary BDS-tree　　(b) Patricia BDS-tree

図 4.14　バケットを用いた Ordinary BDS 木とパトリシア BDS 木

4.4.2　パトリシア BDS 木の圧縮法

図 4.13 の各 BDS 木構造は，1 つの葉ノードが唯一のキーに対応しているため，木の深さが非常に深くなるという問題点がある．これに対して，バケットを採用し，各バケットに対するポインターを各葉に持たせると，木の深さを浅くすることができる．キー集合 K' に対して，バケットを採用した Ordinary BDS 木とパトリシア BDS 木を図 4.14 に示す．但し，バケットのサイズ B_SIZE は 2 とする．

Ordinary BDS 木を先行順ビット列に圧縮する場合は，ダミーリーフを採用したが，パトリシア BDS 木の場合は，常にすべてのノードが 2 本の枝を持つことが保証されているので，ダミーリーフを採用せずに先行順ビット列に圧縮することができる．即ち，ダミーリーフを用いなくても，葉の総数が内部ノードの総数より常に 1 個多いという 2 進木構造の特徴を利用できる．ダミーリーフを用いずに先行順ビット列に圧縮することにより，treemap はダミーリーフ数の 2 倍だけビット数が減少し，更に，leafmap に関しては，ダミーリーフが必要なくなると leafmap 自体を生成する必要もなくなり，先行順ビット列のサイズが大幅に削減される．

しかしながら，パトリシア BDS 木内の各ノードは，Ordinary BDS 木から削除されたノードに関する情報を保持しており，この情報がなければ各キーを正しく検索できない．従って，パトリシア BDS 木を圧縮する場合にも，削除ノードに関する情報を先行順ビット列に加える必要がある．一般的に，パトリシア BDS 木を用いてキー検索を行うために

は，削除されたノード数，または削除されたシンボルに関する情報を各内部ノード内に保持しているが，以後，各ノードには削除されたノード数が格納されているものとする．

本手法では，パトリシア BDS 木を新しい形式の先行順ビット列に圧縮する．新たな先行順ビット列は，treemap，*B_TBL*，そして，leafmap の代わりに nodemap と呼ばれるビット列から構成される．まず，treemap と *B_TBL* については，従来の手法と同様に，treemap はパトリシア BDS 木の形状を示し，*B_TBL* は各バケットのアドレスを格納している．しかしながら，nodemap は各内部ノードが削除ノード数を格納しているか否かを表すビット列である．nodemap の生成方法は，leafmap と同様にパトリシア BDS 木を先行順に走査し，各内部ノードを辿る度に，その内部ノードが削除ノード数を保持していれば，削除ノード数と同じ回数だけビット値 1 を出力した後，1 個のビット値 0 を出力する．また，辿った内部ノードが削除ノード数を保持していなければ（または，削除されたノード数が 0 ならば），ビット値 0 を 1 回だけ出力する．このような方法で生成された nodemap の長さは，Ordinary BDS 木内に含まれる内部ノードの総数と等しくなる．例として，図 4.14 の Ordinary BDS 木とパトリシア BDS 木に対する先行順ビット列を図 4.15 ，図 4.16 に示す．

上記の例において，図 4.16 の先行順ビット列を生成するためには，図 4.14-(b) の完全木となっているパトリシア BDS 木を先行順に走査する．

まず，treemap に関しては，内部ノード 1，2，葉（外部ノード）1，2，内部ノード 3，葉 3，4 の各ノードが順番に辿られ，その結果，treemap は “0011011” に設定される．ここで，treemap の長さに注目すると，その長さは，図 4.15 に示される Ordinary BDS 木の treemap の長さより Ordinary BDS 木に含まれるダミーリーフ数の 2 倍だけ短くなっている．即ち，Ordinary BDS 木内に含まれるダミーリーフ数が多ければ多いほど，より短い treemap がパトリシア BDS 木に対して生成されることになる．

また，nodemap に関しては，パトリシア BDS 木の内部ノードのみを対象に先行順走査し，内部ノード 1，2，3 を順番に辿る．最初に辿られる内部ノード 1 は，削除ノードが含まれないルートノードであるので，nodemap の第 1 ビット値には 0 がセットされる．次に，内部ノード 2 は削除ノード数を示すポインターを持っており，また，その値が 3 であるので，nodemap の第 2 ビットから第 4 ビットまでの 3 ビットの値は 1 が設定される．その後，実際の内部ノード 2 を表すビット値として，第 5 ビットに 0 が出力される．最後に，内部ノード 3 は削除ノード数を指し示すポインターを保持していないので，nodemap

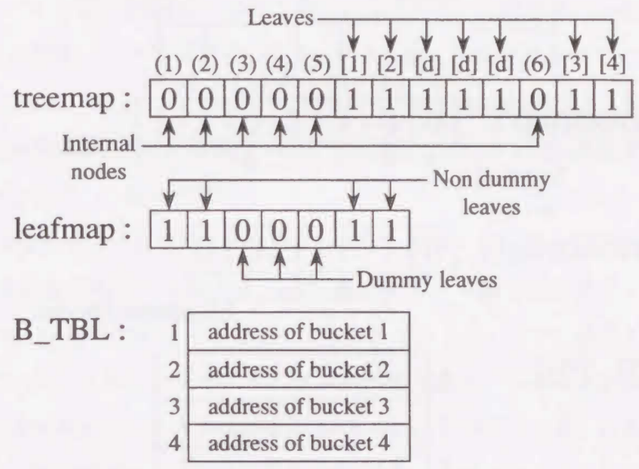


図 4.15　図 4.14-(a) の Ordinary BDS 木に対する先行順ビット列

の第 6 ビット値は 0 になる．以上の処理過程の結果，ビット列 “011100” が nodemap として生成される．尚，パトリシア BDS 木に対する nodemap の長さは，対応する Ordinary BDS 木内に含まれる内部ノード数と同じビット数となる．

4.4.3　パトリシア BDS 木の検索アルゴリズム

新しい先行順ビット列で表現されたパトリシア BDS 木を用いた検索処理も Ordinary BDS 木と同様に，木を先行順走査することによって実行される．即ち，ビット列の先頭ビットから最右端ビットに向かって 1 ビット毎にビット値を検証する作業を行う．本手法はダミーリーフを使用せず，また，先行順ビット列として leafmap も準備しない．しかしながら，4．4．1 で述べたように，パトリシア BDS 木を用いた検索では，Ordinary BDS 木から削除されたノードに関する情報を補助的に用いるため，本手法においても削除されたノードの状態（削除ノード数）が表現された nodemap を検証する必要がある．そのため，3．4 で示した検索アルゴリズム，即ち，先行順ビット列で表現された Ordinary BDS 木の検索アルゴリズムに対して，若干の変更を施す必要がある．

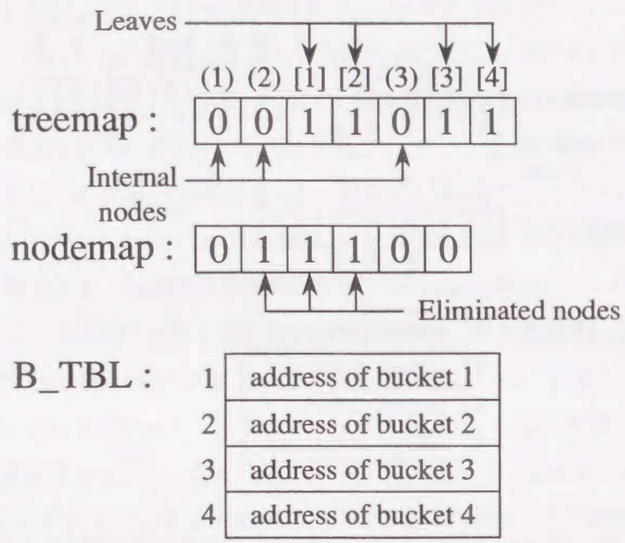


図 4.16　図 4.14-(b) のパトリシア BDS 木に対する先行順ビット列

最初の変更箇所としては，nodemap に対するスキップ処理の追加が挙げられる．まず，Ordinary BDS 木において，右の枝に検索処理が進む際には，左部分木に相当する treemap 上のビット列をスキップする処理が必要であった．パトリシア BDS 木において左部分木をスキップする際には，treemap と同様に nodemap に対しても左部分木に相当するビット列をスキップする必要がある．ここで，パトリシア BDS 木に対する先行順ビット列において，treemap 上で n 番目に出現する 0 のビットと nodemap 上で n 番目に出現する 0 のビットは双方ともパトリシア BDS 木内の同じ番号 n の内部ノードを示している．例えば，図 4.16 の先行順ビット列において，treemap の第 2 ビットと nodemap 上の第 5 ビットとは双方ともに図 4.14-(b) のパトリシア BDS 木内の内部ノード 2 を示しており，更に，双方のビット共，それぞれのビット列上では 2 番目に出現する 0 のビットである．従って，nodemap 上のスキップ処理は，treemap 上でのスキップ処理の後に行われ，nodemap 上でスキップされる 0 のビット数が treemap 上でスキップされた 0 のビット数と同数になるまで，nodemap のビット位置を進めることにより実現される．

また，処理対象の内部ノードが削除ノード数を格納しているか否かを検証する処理も追加される．パトリシア BDS 木に対する新しい先行順ビット列では，nodemap 内の 1 個のビット値 1 が削除ノード 1 個を表している．それ故，検索処理途中で，nodemap 内のビット値 1 に出会した場合，ビット値が 0 になるまで，nodemap のビット位置をスキップする処理が追加される．更に，検索キーを 2 進数表現したビット列に対しても，nodemap 上でスキップしたビット数と同数のビットをスキップする処理が行われる．

更に，B_TBL からキーに該当するスロット番号を獲得する処理にも変更が生じる．Ordinary BDS 木の先行順ビット列では，leafmap 内のビット値 1 が非ダミーリーフを表しているため，leafmap 上でのビット値 1 をカウントすることにより，該当するスロット番号を取得していた．しかしながら，パトリシア BDS 木はダミーリーフを持たず，木構造内のすべての葉が非ダミーリーフであるので，treemap 上のビット値 1 は各葉に対応している．従って，本手法では，treemap 上でのビット値 1 をカウントすることによって，該当スロットを獲得する．

新しい先行順ビット列により表現されたパトリシア BDS 木の検索アルゴリズムを以下に示す．但し，3．4 で述べた Ordinary BDS 木の検索アルゴリズムで用いた変数と同じ変数を用い，更に，次の変数を *leafpos* の代わりに追加的に用いる．

nodepos　：nodemap における現在処理中のビット位置；

【新しい先行順ビット列を用いたパトリシア BDS 木の検索アルゴリズム】

入力：*key*：検索すべきキー；

出力：検索成功ならば TRUE，失敗ならば FALSE；

手順 (S”-1)　：{ 各変数の初期化 }

keypos ← 1，*treepos* ← 1，*nodepos* ← 1；

手順 (S”-2)　：{ ハッシュ値の検証 }

H(*key*) の *keypos* 位置のビット値が 1 ならば手順 (S”-3) に，

ビット値が 0 ならば手順 (S”-5) に進む；

手順 (S”-3)　：{treemap 上での左部分木のスキップ処理 }

treemap 上の *treepos* 位置から，1 のビット数が 0 のビット数より 1 つ多くなるまで *treepos* を進めた後，手順 (S”-4) に進む；

手順 (S”-4)　：{nodemap 上での左部分木のスキップ処理 }

手順 (S”-3) にて treemap 上でスキップした 0 のビット数と同じ回数だけ nodemap 上でもビット値 0 をスキップするまで，*nodepos* を進めた後，手順 (S”-5) に進む；

手順 (S”-5)　：{ 枝を 1 つ辿る処理 }

treepos の位置を 1 つ進めた後，treemap 上の *treepos* 位置のビット値が 0（内部ノード）ならば *nodepos* と *keypos* の位置を 1 つ進め，手順 (S”-6) へ，また，ビット値が 1（葉）ならば手順 (S”-7) へ進む；

手順 (S”-6)　：{ 内部ノードが削除ノード数を格納しているか否かの検証 }

nodemap 上の *nodepos* 位置のビット値が 1 ならば，*nodepos* 位置のビット値が 0 になるまで，*nodepos* と *keypos* を進め，手順 (S”-2) に戻り，

また，ビット値が 0 ならば上記のスキップ処理は行わずに手順 (S”-2) に戻る；

手順 (S”-7)　：{ バケットアドレスの獲得処理 }

treemap 上で先頭ビットから *treepos* 位置までに存在するビット値が 1 のビット数をカウントし，その値を *index* に代入した後，$B\_TBL[index]$ を参照し対応するバケットのアドレスを獲得する；

手順 (S”-8)　：{ バケット内でのキーの検索 }

得られたアドレスにより示されるバケット内に *key* が存在すれば TRUE，存在しなければ FALSE を返す；

上記のアルゴリズムにおいて，nodemap に対する左部分木をスキップする処理は，手順 (S”-4) を追加することによって実現できる．また，処理対象の内部ノードが削除ノード数を格納しているか否かを検証する処理も，手順 (S”-6) において，nodemap を参照することにより，可能となる．更に，treemap 上のビット値 1 の総数により，B_TBL からスロット番号を獲得する処理は手順 (S”-7) で行われる．

例として図 4.16 の先行順ビット列からキー “air”（H(air)=00000...）を検索する手順を図 4.17〜図 4.19 に従って，以下に示す．

手順 (S”-1)　：*keypos*=1，*treepos*=1，*leafpos*=1；

手順 (S”-2)　：H(air)=00000... の第 1 ビット値が 0 なので手順（S”-5）に進む；

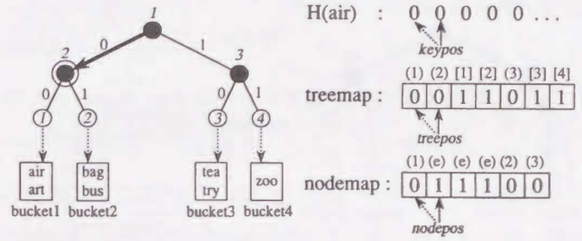


図 4.17　キー “air”検索の説明図 A

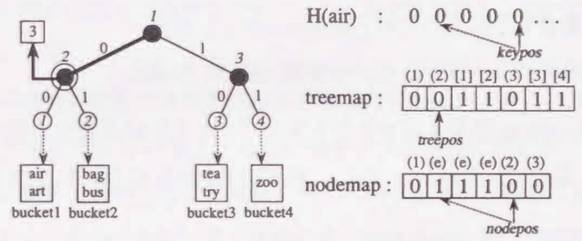


図 4.18　キー “air”検索の説明図 B

手順 (S”-5)　：ノード番号 2 に進むため，*treepos*=2 とした後，treemap の第 2 ビット値が 0 なので，*nodepos*=2，*keypos*=2 とし，手順（S”-6）へ進む（図 4.17 ）；

手順 (S”-6)　：nodemap の第 2 ビット値が 1 なので，内部ノード 2 が削除ノードを含んでいることが分かる；

そこで，nodemap の第 3，4 ビット値は共に 1 であり，第 5 ビット値が 0 であるので，*nodepos* を 3 ビット進まし，*nodepos*=5 とする；

また同時に，*keypos* も 3 ビット進まし，*keypos*=5 とした後，手順（S”-2）に戻る（図 4.18 ）；

手順 (S”-2)　：H(air)=00000... の第 5 ビット値が 0 なので手順（S”-5）に進む；

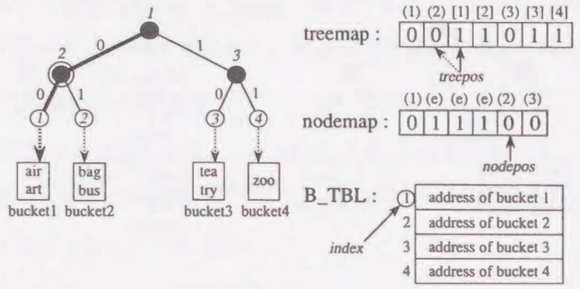


図 4.19　キー “air”検索の説明図 C

手順 (S”-5)　：葉番号 1 に進むため，*treepos*=3 とした後，
treemap の第 3 ビット値が 1 なので手順（S”-7）へ進む；

手順 (S”-7)　：treemap の第 1 ビットから第 3 ビットまでに存在する値が 1 のビット数は 1 個であるので，*index*=1 とし，*B_TBL*[1] からバケット 1 のアドレスを得る（図 4.19 ）；

手順 (S”-8)　：バケット 1 に *key*=“air” が含まれているので，TRUE を返す；

4.4.4　パトリシア BDS 木の更新アルゴリズム

パトリシア BDS 木に対するキーの登録に関しても，Ordinary BDS 木の場合と同様にキーを検索した後，未登録と判断されたキーについて登録処理を行う．但し，Ordinary BDS 木へのキーの挿入操作は，3．2で示した 3 種類に分類できたが，パトリシア BDS 木にはダミーリーフがなく，また，各内部ノードが削除ノードに関する情報を含んでいるため，パトリシア BDS 木へのキーの挿入操作は，以下の 3 種類に分類される．

（1）　：挿入キーに対応するバケット容量に余裕がある場合；

（2）　：挿入キーに対応するバケット容量が満杯の場合；

（3）　：挿入キーに対応しないバケットに辿り着いた場合；

まず，（1）は挿入キーに対応するバケットが見つかり，そのバケットに登録済みのキー数が *B_SIZE* - 1 以下の場合である．この場合は，単純に挿入キーをそのバケットに登録するだけで実現できる．

また，（2）は挿入キーに対応するバケットは見つかるのだが，そのバケットには既に *B_SIZE* 個のキーが登録されている場合である．この場合は，3．5で示した Ordinary BDS 木の更新処理と同様に，オーバーフローを起こしたバケットに対応する葉を内部ノードに変換する必要がある．但し，Ordinary BDS 木では，バケット内に格納されるべきキーの共通接頭部分を枝として生成し，その下に新たな 2 つの葉を追加していたが，パトリシア BDS 木の場合には，フルバケット以下の共通接頭部分は 1 個の内部ノードとしてのみ生成し，共通ビット値，あるいは，共通ビットの個数は削除ノードに関する情報として，補助的に格納する必要がある．そして，新たに生成された内部ノードの下に 2 つの葉を追加し，追加された葉に対応した新たなバケットをそれぞれ生成し，各バケットにキーを分類格納することになる．

最後に，（3）は検索の結果バケットには辿り着くのであるが，そのバケットが挿入キーを格納すべきバケットではない場合である．つまり，本手法で用いるパトリシア構造では，削除ノードに関する情報として，削除ノードに対するビット値そのものではなく，削除ノードの個数のみが保持されている．そのため，検索処理において削除ノードを含む内部ノードを辿る際には，キーとなるビット列を削除ノードの個数分スキップするだけで，スキップされたビット値の検証は行っていない．登録済みのキーを検索する場合には，このような処理で支障ないが，（更新処理の前処理として行われる）未登録語を検索する場合には，実際の削除ノード値と異なるビット値がマッチングされてしまう可能性がある．このようなミスマッチングを**フォルスドロップ（false drop）**と呼ぶ．もし，フォルスドロップが起こった後に辿り着いたバケットに挿入キーを登録すれば，削除ノードに関する情報の一意性が失われてしまうため，正しい検索処理ができなくなる．

そこで，（3）のような場合には，まず，フォルスドロップが発生した内部ノードを確定する．そして，その内部ノードの親ノードとして新たな内部ノードを作成し，新たな内部ノードに対応する枝と葉を 1 個ずつ生成した後，それぞれの内部ノードに対応する削除ノード数を割り振ることになる．

次に，（1）～（3）の各処理への分類方法，また，（3）の処理において如何にしてフォルスドロップが発生したノードを発見するか，更に，削除ノードの割り振り方等の詳細に

ついて説明する．

まず，挿入キーは検索の後，未登録語と判定されるのだが，その挿入キーを n 長のビット列 X として以下のように定義する；

$$X = x_0\ x_1\ x_2\ \ldots\ x_{n-1}$$

但し，x_i $(0 \leq i \leq n-1)$ は，0または1のビット値を表す．

また，検索の結果，辿り着いたバケットに既に登録されている任意のキーを m 長のビット列 B として以下のように定義する；

$$B = b_0\ b_1\ b_2\ \ldots\ b_{m-1}$$

但し，b_i $(0 \leq i \leq m-1)$ は，0または1のビット値を表す．

次に，ビット列 X と B とのビット単位の比較演算を行い，双方とも第αビット目までがすべて同一値であったと仮定する．即ち，以下のようにビット列 X と B とを以下のように再定義すると，

$$X = x_0\ x_1\ x_2\ \ldots\ x_{\alpha-1}\ x_\alpha\ \ldots\ x_{n-1}$$
$$B = b_0\ b_1\ b_2\ \ldots\ b_{\alpha-1}\ b_\alpha\ \ldots\ b_{m-1}$$

但し，$x_0 = b_0$，$x_1 = b_1$,...，$x_{\alpha-1} = b_{\alpha-1}$，$x_\alpha \neq b_\alpha$,... とする．

尚，この段階で行われるビット単位の比較演算は，ビット列 X と B との排他的論理和を求めることにより実現でき，演算結果が0のビットは双方とも同一値であり，1のビットは異なる値を持つことになる．即ち，排他的論理和の演算結果において，最初にビット値1が出現するビット位置がα+1となる．

ここで，ビット列 B が格納されているバケットに対応する葉までの木の深さを d $(0 \leq d \leq m-1)$ とする．但し，d はOrdinary BDS木での木の深さを表すものとし，d の値は，パトリシアBDS木でのバケットまでの深さに，そのバケットまでの経路に含まれる削除ノード数を加えることにより求める．

そして，d とαとの大小関係より，以下のように各種処理に分類する．但し，検索の結果辿り着いたバケットに既に登録済みのキー数を K_NUM とする．

- $d \leq \alpha$，かつ，$K\_NUM < B\_SIZE$ ならば，(1) の処理を行う；

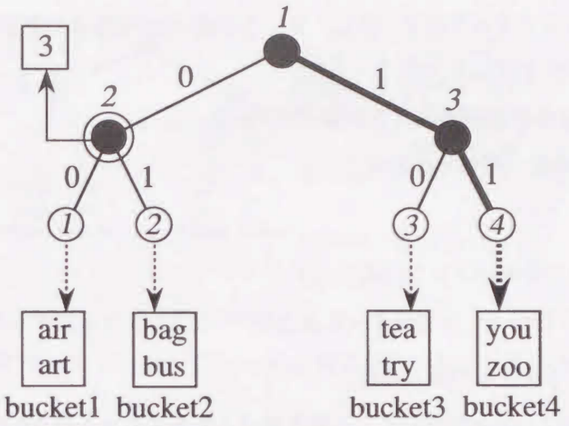


図 4.20　図 4.14-(b) にキー “you” 追加後のパトリシア BDS 木

- $d \leq \alpha$，かつ，$K\_NUM = B\_SIZE$ ならば，(2) の処理を行う；
- $d > \alpha$ ならば，(3) の処理を行う；

次に，各処理内容の詳細を説明する

(1) の場合は，単純に挿入キーを対応するバケットに格納した後，K_NUM の値を1つ増すだけでよい．例として，図4.14-(b) のパトリシアBDS木にキー “you” (H(you)=11000 ...) を挿入した後の構造を図4.20に示す．尚，先行順ビット列の内容は変化しない．

(2) の場合は，

(i)　：Ordinary BDS木と同様に，フルバケットに対応した葉を単位木に変換する；

(ii)　：Ordinary BDS木のようなバケットの再オーバーフロー処理は行わず，共通接頭部分は削除ノードとして新たに生成された内部ノード（単位木のルートノード）内に格納される；
但し，その内部ノードが保持する削除ノード数は$\alpha - d$個とする；

(iii)　：単位木内の2つの葉に各キーを割り振るため，各キービット列の$\alpha+1$番目のビット値が0ならば単位木の左の葉が示すバケットに，$\alpha+1$番目のビット値が1ならば単位木の右の葉が示すバケットに各キーを格納する；

以下に，新しい先行順ビット列により表現されたパトリシア BDS 木のパターン（2）に対する更新アルゴリズムを示す．但し，3．5で述べた BDS 木の更新アルゴリズムで用いた変数 *K_SET* を用いる．

【パトリシア BDS 木の更新アルゴリズム・パターン（2）】

入力：*key*：検索の後，登録されるキー；

出力：なし；

手順 (I"-2-1)　：{フルバケット内のクリア}

フルバケット内のキーと登録キーを *K_SET* に代入し，フルバケット内を空にする；

手順 (I"-2-2)　：{単位木の挿入処理}

treemap 上の *treepos* 位置のビット値を単位木を表すビット列 "011" に変更する；

手順 (I"-2-3)　：{削除ノード数の設定処理}

$\alpha - d$ 個のビット値1とその後に続く1個のビット値0から成るビット列を nodemap 上の *nodepos* 位置の次に挿入する；

手順 (I"-2-4)　：{*K_SET* 内の各キーの格納処理}

K_SET 内の各キーにおいて，キービット列の $\alpha+1$ 番目が0ならば，単位木の左の葉が示すバケットに，$\alpha+1$ 番目が1ならば，単位木の右の葉が示すバケットに，各キーを格納する；

例として，図 4.14-(b) のパトリシア BDS 木，および図 4.16 の先行順ビット列にキー "tax"（H(tax)=100110 ... ）を挿入する場合に行われる手順を図 4.21，図 4.22 に従って説明する．

まず，バケット3まで検索が行われた際の各変数値を図 4.21 に示す．*treepos* は第6ビット目を示し，*nodepos* は最終ビットである第6ビット目を示している．また，葉番号3までの木の深さは d は2である．更に，H(tax)，H(tea)，H(try) の各ビット列をビット単位に比較演算すると，第5ビット目までのビット値がすべてのキーに対して等しいので，αの値は5となる．その結果，$d \leq \alpha$で，かつ，バケットがオーバーフローするので，パターン（2）の更新処理が行われる．以下に手順を示す．

手順 (I"-2-1)　：挿入キーとフルバケット内の各キーとを *K_SET* に代入し，

$$K\_SET=\{\text{tax, tea, try}\}$$

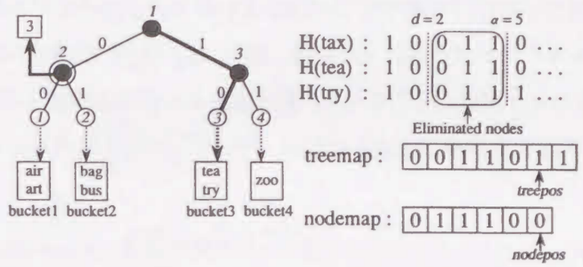


図 4.21　図 4.14-(b) でキー "tax"検索後のパトリシア BDS 木

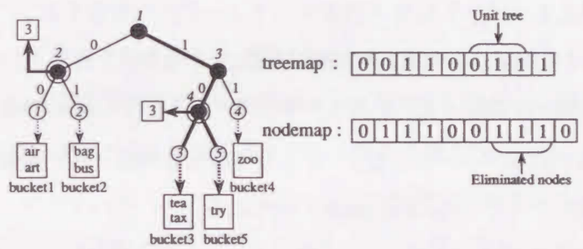


図 4.22　図 4.14-(b) にキー "tax"挿入後のパトリシア BDS 木

とする；

手順 (I"-2-2)　：treemap の第6ビット値をビット列 "011" に変更する；

手順 (I"-2-3)　：各キーの第3～第5ビットまでのビット列 "011" を削除ノードとするため，ビット列 "1110" を nodemap の最終ビットの後に追加する；

手順 (I"-2-4)　：H(tax)，H(tea) の第6ビット値が0なのでキー "tax"と "tea"はバケット3に，H(try) の第6ビット値は1なのでキー "try"はバケット5に格納する；

以上，キー "tax"を挿入後のパトリシア BDS 木，及び対応する先行順ビット列を図 4.22 に示す．

次に,（3）の場合は，木構造の特徴からビット列XとBとの間で最初に異なるビット位置に相当する枝を1本だけを増やしてやればよいので，双方のビット列間の相違部分内で最初にフォルスドロップが起こった位置，即ち，x_α，b_αの周辺にだけ注目する．尚，最初にフォルスドロップが起こったビット，b_αの隣接ビットには削除ノードが存在するので，ビット列Bを以下のように再定義する．

$$B = b_0\ b_1\ ...\ b_i\ ...\ b_\alpha\ ...\ b_j\ ...\ b_{m-1}$$

但し，$i \leq \alpha \leq j$, $i \neq j$とする．ここで，$b_i \sim b_j$のビット列が削除ノードが示すビットである．

（3）の場合に行われるパトリシアBDS木上の手順を以下に示す．

(i)　：フォルスドロップを起こしたノード（フォルスドロップノードと呼ぶ）を確定するため，挿入キービット列Xを再度ルートノードから検索する；
この際，d=1に初期化した後，dの値を設定しながら検索を行い，$d = \alpha+1$となった時点で辿り着いた内部ノードをフォルスドロップノード（false drop node）とする；

(i i)　：フォルスドロップノードの親ノードとして新たな内部ノードを挿入し，この内部ノードを挿入ノード（inserted node）と呼ぶ；

(iii)　：挿入ノードが保持する削除ノード数を$\alpha - i$個，フォルスドロップノードが保持する削除ノード数を$j - \alpha$個とする；

(iv)　：x_αの値が0ならば，挿入ノードから左の枝と葉を1個ずつ生成し，x_αの値が1ならば，挿入ノードから右の枝と葉を1個ずつ生成する；

(v)　：x_αの値が0ならば，左の葉が示すバケットに，x_αの値が1ならば，右の葉が示すバケットに挿入キーを格納する；

以下に，新しい先行順ビット列により表現されたパトリシアBDS木のパターン（3）に対する更新アルゴリズムを示す．

【パトリシアBDS木の更新アルゴリズム・パターン（3）】

入力：*key*：検索の後，登録されるキー；

出力：なし；

手順 (I"-3-1)　：{フォルスドロップノードの確定処理}

挿入キービット列Xを4．4．3に示したアルゴリズムにより，

$d^2 = \alpha+1$になるまで検索する；

尚，検索終了後に辿り着いたノードがフォルスドロップノードであり，この時点で*treepos*はフォルスドロップノード，*nodepos*はb_αに対応するビットを指す；

手順 (I"-3-2)　：{挿入ノードの挿入処理}

treemap上の*treepos*位置の次にビット値0を挿入する；これで，*treepos*は挿入ノードを指し示すことになる；

手順 (I"-3-3)　：{削除ノード数の設定処理}

*nodepos*が指し示す位置で削除ノード数をフォルスドロップノードと挿入ノードの2つの内部ノードに分割するため，nodemap上の*nodepos*位置のビット値を0に変更する；

手順 (I"-3-4)　：{挿入ノードからx_αの値に対応する枝の作成処理}

x_αの値が0ならば，挿入ノードの左側に1個の枝葉を作成するため，treemap上の*treepos*位置の次に1個の枝葉を表すビット値1を挿入し，

x_αの値が1ならば，挿入ノードの右側に1個の枝葉を作成するため，treemap上で挿入ノードの左部分木をスキップした後のビット位置，即ち，*treepos*位置から1のビット数が0のビット数より1つ多くなるまで*treepos*を進めた後，*treepos*位置の次にビット値1を挿入する；

手順 (I"-3-5)　：{挿入キーの格納処理}

新たに作成された葉に対応するバケットに挿入キーを格納する；

例として，図4.14-(b)のパトリシアBDS木，および図4.16の先行順ビット列へのキー“ear”（H(ear)=00100 ... ）の挿入処理を図4.23 ～図4.25に従って説明する．

まず，キー“ear”を検索すると，バケット1に辿り着き，バケット1内部の検索の結果，未登録語と判断される．そこで，H(ear)，H(air)，H(art)をビット単位で比較演算すると，第2ビット目までのビット値が等しいので，α=2となる．また，バケット1までの木の深さdは，内部ノード2が削除ノードを3個含んでいるので，d=5である．よって，$d > \alpha$なので，パターン（3）の更新処理が行われる．以下に手順を示す．

[2] 4．4．3に示したアルゴリズム内で用いられた*keypos*がdの値と一致する．

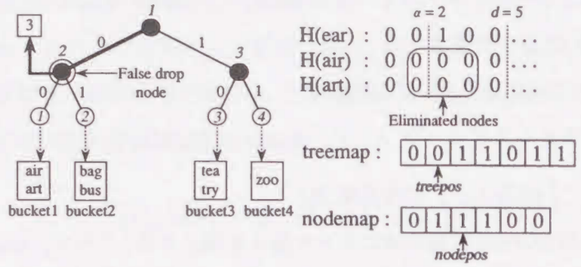


図 4.23　図 4.14-(b) でキー “ear”再検索後のパトリシア BDS 木

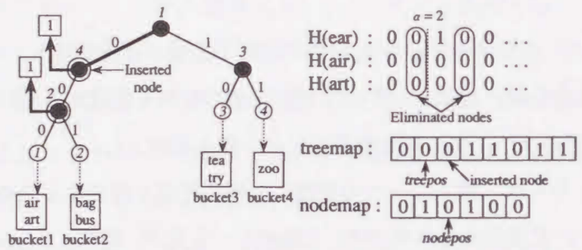


図 4.24　図 4.23 から削除ノードを分離したパトリシア BDS 木

手順 (I”-3-1)　：H(ear) に対して，d=3 になるまで再度検索する；

この検索処理は，内部ノード 2，即ち，フォルスドロップノードまで進み，*treepos* は第 2 ビット目を示し，*nodepos* は第 3 ビット目を示す；

尚，検索後の各変数値を図 4.23 に示す；

手順 (I”-3-2)　：挿入ノードを作成するため，treemap の第 2 ビットの次にビット値 0 を挿入する；

手順 (I”-3-3)　：nodemap の第 3 ビット値を 0 に変更する；

この処理により，削除ノード数は図 4.24 に示すように内部ノード 4（挿入ノード）

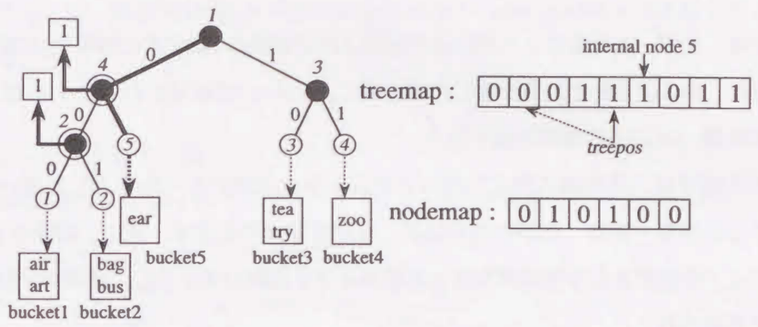


図 4.25　図 4.14-(b) にキー “ear”挿入後のパトリシア BDS 木

に 1 個，また，内部ノード 2（フォルスドロップノード）に 1 個分類される；

手順 (I”-3-4)　：H(ear) の第 3 ビット値が 1 なので，挿入ノードの右側に 1 個の枝葉を作成する；

まず，treemap の第 2 ビット値からビット値 1 の数がビット値 0 の数より 1 つ多くなるまで *treepos* を進め，*treepos*=5 とする；

そして，この 1 つ後のビット位置，即ち，第 6 ビット位置に右の枝葉を表すビット値 1 を挿入する；

手順 (I”-3-5)　：葉 5 に対応する新しいバケット 5 に挿入キー “ear”を格納し，図 4.25 に示されるパトリシア BDS 木，および，対応する先行順ビット列を得る；

4.5　各種改善手法に対する評価

本節では，まず，先行順ビット列により表現された従来のBDS木と階層化BDS木との理論的評価，および具体的評価を論じた後，更に，従来のBDS木とパトリシアBDS木との理論的評価，および具体的評価も行う．

具体的評価では，日本語名詞5万語および英単語5万語をキー集合として用いた場合の，各手法の検索・追加・削除の処理時間，使用記憶量等を示す．更に，階層化BDS木については，分割深さと各種処理時間との関係を表す実験結果を示し，最適な分割深さについての考察を行う．

4.5.1　階層化BDS木の理論的評価

先行順ビット列を用いたBDS木と階層化BDS木の時間効率と空間効率を比較する際に，木構造は完全木とし，次のようなパラメータを用いる．

n　：完全木の最大深さ；

m　：分割深さ；

α　：[n / m]（n / m以上である最小の整数）で得られる分割木の最大リンク数；

まず，時間効率に関して，各手法の検索・更新・削除の最大時間計算量（以後，計算量と略する）を以下に示す．

BDS木の検索は最悪の場合，完全木全体を走査するので，計算量は$O(2^n)$となる．これに対して，階層化BDS木では，α個の分割木を完全に走査すればよいので，$O(\alpha 2^m)$となる．

更新・削除に関しては，最も左に位置する葉（バケット）を対象とする処理が最悪の場合となる．この場合，バケットの分割・併合が行われると仮定すれば，葉に対応したビット位置から以降に存在するすべてのビット列をシフト移動しなければならないため，BDS木での計算量は$O(2^n - n)$となり，階層化BDS木では1つの分割木のみで同じ操作が行われるため$O(2^m - m)$となる．一般に，2^n，2^mに対してn，mは小さいため，更新・削除の時間計算量は，BDS木で$O(2^n)$，階層化BDS木では$O(2^m)$となる．

次に，空間効率に関しては，BDS木を先行順ビット列を用いて構築すると，treemapは完全木の全ノード（葉も含む）数から成るので$2^{n+1}-1$，leafmapは完全木の葉の数なの

で2^nのビット数が必要となる．

これに対して，階層化BDS木でのtreemapおよびleafmapのサイズは以下のように求められる．

treemapに要するビット数

= (1個の分割木内に含まれるの全ノード数) × (完全木に含まれる全分割木数)

$$= \sum_{k=0}^{m} 2^k \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = (2^{m+1}-1)\frac{2^{m\alpha}-1}{2^m-1}$$

$$= \{2(2^m-1)+1\}\frac{2^{m\alpha}-1}{2^m-1} = (2^{m\alpha+1}-2) + \frac{2^{m\alpha}-1}{2^m-1}$$

$$= (2^{n+1}-1) + \frac{2^n-1}{2^m-1} - 1$$

leafmapに要するビット数

= (1個の分割木内に含まれる葉の数)×(完全木に含まれる全分割木数)

$$= 2^m \times \sum_{k=1}^{\alpha} 2^{m(k-1)} = 2^m \frac{2^{m\alpha}-1}{2^m-1}$$

$$= (2^m - 1 + 1)\frac{2^{m\alpha}-1}{2^m-1} = (2^{m\alpha}-1) + \frac{2^{m\alpha}-1}{2^m-1}$$

$$= 2^n + \frac{2^n-1}{2^m-1} - 1$$

上記の結果より，BDS木を階層化することによって，treemapおよびleafmapともに

$$(2n-1) \,/\, (2m-1) \;-\; 1 \simeq 2^{n-m}$$

で得られるビット数だけ記憶量が増加する．尚，B_TBLについては，ダミーリーフの割合によって記憶量が変化するため，実験による値を4．5．2で示す．

表 4.2　階層化 BDS 木の実験結果

Key sets	Japanese nouns		English words	
Kinds of trees	BDS-tree	HBDS-tree	BDS-tree	HBDS-tree
Number of				
non_dummy leaves	6,002		6,159	
dummy leaves	3,649		8,411	
Internal nodes	9,650		14,569	
depth	82		70	
separated tree	2,060		2,940	
Time (Second)				
Registration	870	146	1875	164
Time (Milli-Second)				
Retrieval	8.68	0.48	11.26	0.56
Insertion	38.00	3.00	37.50	3.28
Storage (K-byte)				
treemap	2.41	2.67	3.64	4.00
leafmap	1.21	1.46	1.82	2.19
B_TBL	12.00	16.12	12.32	18.20

4.5.2　階層化BDS木の具体的評価

本手法の構成システムは約2,000行のC言語で記述されており，Sparc Station2（28MIPS）上で稼働している．

本手法の有効性を確認するため，従来のBDS木（BDS-tree）と階層化BDS木（HBDS-tree）との比較を行った．ランダムな日本語名詞5万語（平均長6バイト），および英単語5万語（平均長9バイト）に対する実験結果を表 4.2 に示す．尚，階層化BDS木の分割深さは5，*B_SIZE* を 16 とした．

検索，削除時間は，最初に全ての語を登録しておき，登録済みの全ての語について検索・削除を行った場合の1語当たりに要する時間である．また，追加時間は，すべてのキー集合を登録した後，1,000語の未登録語を追加した場合の1語当たりに要する時間である．

まず，日本語名詞5万語をキー集合として登録した場合，内部ノードの総数は9,650個，葉の総数は9,651個，その内，ダミーリーフの総数は3,649個であり，約38%の外部ノードがダミーリーフとなった．また，生成された分割木の総数は2,060個であり，木構造の最大深さは82となった．尚，日本語名詞5万語の登録には，BDS木法では870秒掛かったのに対し，階層化BDS木法では146秒で終了した．

次に，英単語5万語をキー集合として登録した場合，内部ノードの総数は14,569個，葉の総数は14,570個，その内，ダミーリーフの総数は8,411個であり，約58%の外部ノードがダミーリーフであった．また，生成された分割木の総数は2,940個であり，木構造の最大深さは70となった．尚，英単語5万語の登録には，BDS木法では1875秒掛かったのに対し，階層化BDS木法では164秒で終了した．

以上の実験結果から，従来の手法に比べて本手法は，検索では18～20倍，追加では11～13倍，削除では4～6倍といずれも非常に高速になっており，各処理時間が大幅に短縮されたことが分かる．一方，記憶量は改良前に比べ，treemapで1.10～1.11倍，leafmapで1.20～1.21倍，バケット表で1.34～1.48倍程増加しているが，先行順ビット列が元来非常にコンパクトであるので十分実用的である．

また，従来のBDS木では，キー集合の性質により，各種処理時間に大きな較差が生まれている．例えば，キー集合全体の登録に関しては，日本語名詞が870秒であるのに対して，英単語は1875秒と約2.2倍もの時間を要している．一方，階層化BDS木の場合は，日本語名詞キー集合の登録には146秒，英単語キー集合には164秒と，そほど時間差を生じていない．つまり，BDS木を階層化することによって，高速化が実現できるだけでなく，キー集合の性質から生じる各種処理時間の較差を抑えることもできる．

更に，キー1個当たりに必要な記憶容量は，BDS木の場合では2.50ビット，階層化BDS木の場合では3.24ビットとなり，本手法は B 木や B^+木に比べて非常にコンパクトな記憶検索法といえる．

分割深さと1語当たりの検索・更新時間との関係を図 4.26 に示す．まず，検索に関しては，分割深さが大きくなると走査する必要のないノード数が増加するため，検索時間は指数関数的に増加する．また，更新に関しては，分割深さが小さすぎると木の分割が頻繁に起こるため，更新時間が増加し，逆に，分割深さが大きすぎるとバケット分割の際に単位木を挿入するコストが悪影響を及ぼす．以上より，分割深さが１０前後が最も適していることが分かる．

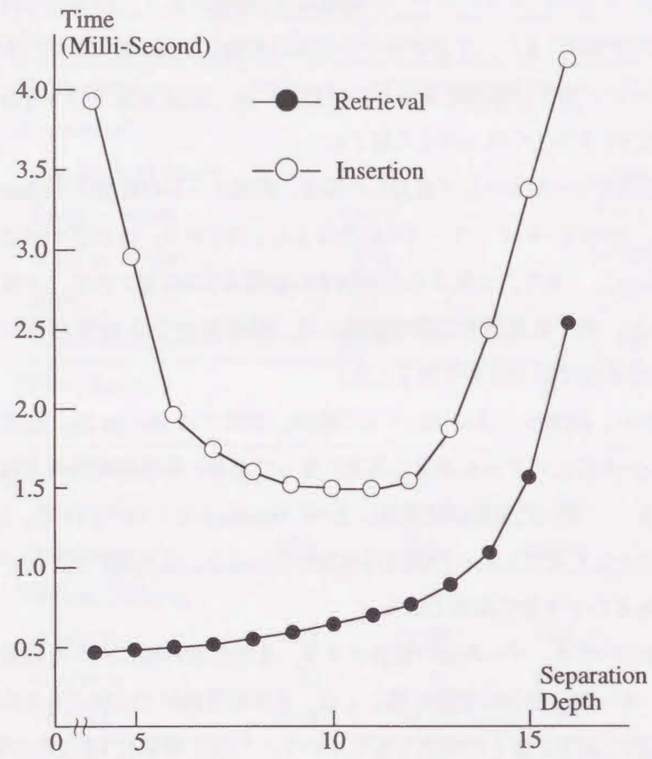


図 4.26　分割深さと検索・更新時間との関係

4.5.3　パトリシア BDS 木の理論的評価

先行順ビット列で表現された Ordinary BDS 木とパトリシア BDS 木との空間効率，および時間効率を比較するため，まず，比較対象とする Ordinary BDS 木は完全木とし，その完全木の最大深さを n とする．更に，完全 Ordinary BDS 木に含まれるダミーリーフの比率（全外部ノード数に対するダミーリーフの割合）を $D \times 100(\%)$ とし，D の値の範囲は $0 \leq D \leq 1$ とする．

まず最初に，各データ構造の空間効率を考察するため，各木構造に対する先行順ビット列，treemap，leafmap，および nodemap に必要なビット数を求める．但し，Ordinary BDS 木に対する treemap を $treemap_o$ とし，また，パトリシア BDS 木に対する treemap を $treemap_p$ とする．

Ordinary BDS 木に対する先行順ビット列，即ち $treemap_o$ と leafmap に必要なビット数を以下に示す．$treemap_o$ は完全 Ordinary BDS 木内の全ノード（葉も含む）から構成されるので，$treemap_o$ を生成するために要求されるビット数は $2^{n+1}-1$ となる．また，leafmap は完全 Ordinary BDS 木内の葉の総数に対応するため，leafmap を構成するためには 2^n ビット必要となる．

一方，パトリシア BDS 木に対する先行順ビット列，即ち $treemap_p$ と nodemap のサイズは以下のように計算される．尚，完全 Ordinary BDS 木内に含まれるダミーリーフは，葉の総数の $D \times 100\%$ であるので，ダミーリーフの総数は $D2^n$ となる．

$treemap_p$ に要するビット数
=(完全 Ordinary BDS 木内に含まれるの全ノード数) - (完全 Ordinary BDS 木内に含まれるダミーリーフ数) × 2

$$= \sum_{k=0}^{n} 2^k - (2D2^n) = 2^{n+1} - 1 - D2^{n+1} = (1-D)2^{n+1} - 1$$

nodemap に要するビット数
= 完全 Ordinary BDS 木内に含まれるの内部ノード数
$= 2^n - 1$

本手法によって，leafmap と nodemap とのビット数は殆ど同じであるが，treemap に対してはダミーリーフ数の2倍のビット数が減少することになる．ここで，この treemap の減少ビット数（ダミーリーフ数の2倍）と元の $treemap_o$ に要するビット数との比率を“treemap の減少比率”と定義する．この比率は，以下のように計算される．

treemap の減少比率

$$= (\text{treemap の減少ビット数}) \,/\, (treemap_o \text{に要するビット数})$$

$$= \frac{2D2^n}{\sum_{k=0}^{n} 2^k} = \frac{D2^{n+1}}{2^{n+1}-1} = \frac{D(2^{n+1}-1)+D}{2^{n+1}-1}$$

$$= D + \frac{D}{2^{n+1}-1} \simeq D$$

但し，nが大きくなるに従って，$D \,/\, (2^{n+1}-1) \to 0$ とする．

上記の結果から，Ordinary BDS 木内に含まれるダミーリーフの割合が多くなればなるほど，パトリシア BDS 木に変換した場合の treemap の減少率も大きくなり，よりコンパクトな treemap が生成できることが分かる．より多くのダミーリーフが生成されるキー集合，換言すると，キーの分布状況に偏りがあるキー集合に対しては，パトリシア BDS 木に対する新しい先行順ビット列の方が，Ordinary BDS 木に対する従来の先行順ビット列よりもコンパクトなデータ構造となる．また，B_TBL に関しては，Ordinary BDS 木およびパトリシア BDS 木共に非ダミーリーフ数，即ちバケットの総数は同じであるので，B_TBL のサイズは変化しない．

次に，空間効率について，各手法の検索・更新・削除処理に対する最大時間計算量（以後，計算量と略する）を以下に示す．まず，Ordinary BDS 木の検索において，最悪の場合は，完全木全体が走査される．この場合の計算量は，4．5．1で述べたように $O(2^n)$ となる．また，Ordinary BDS 木の更新・削除に関しては，最も左に位置する葉に対して処理が行われ，先行順ビット列上では葉に対応するビット以降のビット列をすべてシフトしなければならない場合が最悪となる．そして，この場合の計算量も，4．5．1で述べたように，先行順ビット列の最左端の葉に対応したビット位置より以降のすべてのビット列をシフト移動しなければならないので $O(2^n - n)$ となる．一方，パトリシア BDS 木に関しては，前述の空間効率に関する議論内でも述べたように，$treemap_p$ は $treemap_o$ よりも $D \times 100\%$ 短くなる．そのため，パトリシア BDS 木の先行順ビット列に対する各

種処理に要する時間は，Ordinary BDS 木のビット列に要する時間よりも $D \times 100\%$ 短縮される．即ち，パトリシア BDS 木の検索処理に対する計算量は $O(D2^n)$ となり，更新・削除に対しては $O(D(2^n - n))$ となる．一般的に，n は 2^n に比べて非常に小さな値なので，Ordinary BDS 木に対する各種処理の計算量は $O(2^n)$，パトリシア BDS 木に対しては $O(D2^n)$ となる．

また，本手法により生成される先行順ビット列は非常にコンパクトなデータ構造である．それ故，先行順ビット列により構成された索引を主記憶上に格納すること前提にし，一般的なサイズ（万単位）のキー集合に対して本手法を適用した場合，キー検索に要する2次記憶へのディスクアクセス数を1回に抑えることができる．更に，大規模なサイズ（億単位）のキー集合に対しては，B_TBL を2次記憶上に配置したとしても，2回のディスクアクセスのみで高速にキーを検索することが可能となる．

現在までに議論してきた検索処理は，登録済みのキーに対する検索処理であった．そこで，次に未登録キーに対する検索処理の効率について考察を行う．まず，Ordinary BDS 木への未登録語の検索に関して，未登録語の検索処理がダミーリーフに辿り着いた場合，先行順ビット列が主記憶上に格納されていると仮定すれば，検索キーを未登録語と判別するのにディスクアクセスを必要としない．一方，パトリシア BDS 木はダミーリーフを持たないので，検索キーが未登録語と判定するためには，バケットへのアクセス，即ち，2次記憶へのディスクアクセスが必要となる．

更に，キー集合が大きくなり，その結果，木のサイズ，即ち，木の最大深さ n の値が大きくなると，パトリシア BDS 木の先行順ビット列に対する時間効率も Ordinary BDS 木と同様に低下してしまう．しかしながら，4．3で提案した木構造を階層的に分割する改善手法を用いることにより，時間効率を大幅に改善することが可能である．

4.5.4　パトリシア BDS 木の具体的評価

本提案手法を実現するシステムは約3,500行のC言語で記述されており，Sparc Station2（28MIPS）上で稼働している．

本手法の有効性を確認するため，Ordinary BDS 木に対する従来の先行順ビット列と本手法によりパトリシア BDS 木から生成された新しい先行順ビット列との比較を行った．日本語名詞5万語（平均長6バイト），および英単語5万語（平均長9バイト）の各キー集

合に対して行った実験結果を表4.3に示す．但し，バケットのサイズはB_SIZE=16とした．

表4.3において，項目“ダミー比率（Dummy rate）”とは，各木構造内に含まれるダミーリーフ数と外部ノード（ダミーリーフも含む）の総数との間の比率を示し，その値は以下のように計算される．

ダミー比率
=ダミーリーフ数 / 総外部ノード数；

また，項目“treemapの減少率（Rate of decrease in treemap）”とは，本手法によってtreemapが何パーセント減少するかを示すものであり，その値は次のような計算により求められる．

treemapの減少率
=（$treemap_o$のサイズ - $treemap_p$のサイズ）/ $treemap_o$のサイズ；

更に，項目“ビット列の減少率（Rate of decrease in bit streams）”とは，B_TBLを除いた先行順ビット列が本手法によりどの程度減少したかを表すものであり，その値は以下のように求められる．

ビット列の減少率
=（$treemap_o$とleafmapとの合計サイズ - $treemap_p$とnodemapとの合計サイズ）
/ $treemap_o$とleafmapとの合計サイズ；

上記の実験結果から，本手法を用いることにより，treemapは40〜60%に減少し，leafmapやnodemapを含めたビット列では25〜40%のビットが削減されることが分かる．また，双方のキー集合共に，“treemapの減少率”の値と“ダミー比率”の値が同一である．日本語名詞の場合には37.8あり，英単語の場合には57.7となっている．これは，$treemap_p$のサイズが$treemap_o$のサイズに比べて，Ordinary BDS木に含まれるダミーリーフの比率だけ短くなることを表している．一般的に，Ordinary BDS木の40〜60%の外部ノードがダミーリーフになることも報告されており[45]，以上の結果から，本手法は如何なるキー集合に対しても，$treemap_o$より40〜60%短いtreemapを生成することが可能であると言える．

表4.3　パトリシアBDS木の実験結果

Key sets	Japanese nouns		English words	
Kinds of trees	Or_BDS-tree	Pa_BDS-tree	Or_BDS-tree	Pa_BDS-tree
Information of trees				
Number of				
nodes	19,301	12,003	29,139	12,317
Internal nodes	9,650	6,001	14,569	6,158
external nodes	9,651	6,002	14,570	6,159
non-dummy leaves	6,002	6,002	6,159	6,159
dummy leaves	3,649	0	8,411	0
Dummy rate(%)	37.8	0	57.7	0
Information of				
pre-order bit streams				
Size of (Kbyte)				
treemap	2.41	1.50	3.64	1.54
leafmap	1.21	0	1.82	0
nodemap	0	1.21	0	1.82
B_TBL	12.00	12.00	12.32	12.32
Rate of decrease in (%)				
treemap	37.8		57.7	
bit streams	25.2		38.5	

更に，従来の手法では（Ordinary BDS木を従来の先行順ビット列で表現した場合），日本語名詞キー集合からなる先行順ビット列に要する記憶量は15.6Kバイトであるのに対して，英単語キー集合では17.8Kバイトに増加しており，キー集合の特性が異なると，先行順ビット列のサイズに大きな較差（2.2Kバイト）が生じている．この較差は，キー集合内の各キーの値の均一さに従って，ダミーリーフ数が増減することに原因がある．一方，提案手法では（パトリシアBDS木を新しい先行順ビット列で表現した場合），日本

語名詞に対する記憶量は14.7Kバイト，英単語に対しては15.7Kバイト，各キー集合間の差が1.0Kバイトとなり，較差が非常に小さな値に抑えられている．このように，本手法は，キー集合の特性により生じる必要記憶量の較差を抑える効力も有する．

新しい先行順ビット列で表現されたパトリシアBDS木において，1個のキー当たりに必要な記憶量は2.4〜2.5ビットであり，本手法はB木やB^+木等よりもコンパクトな索引を作成することが可能である．

また，1個のキーを検索するために要する時間は，Ordinary BDS木の場合，4．5．2で示したように，日本語名詞に対しては0.48ミリ秒，英単語では0.56ミリ秒であった．これに対して，パトリシアBDS木の場合は，日本語名詞で0.40ミリ秒，英単語では0.42ミリ秒となり，時間効率も向上している．勿論，唯1回のみのディスクアクセスで検索は行われた．但し，索引となる先行順ビット列は主記憶上に格納されており，また，対応する各木構造は4．3で示した階層化による改善手法が施されている．

このように，本手法により表現された新しい先行順ビット列は，Jongeら[24]が提案したビット列よりもコンパクトであり，更に，1回のディスクアクセスで検索が実現できるため，PAT array[32]を用いた場合よりも高速な検索が可能となる．

4.6 結　言

本章では，先行順ビット列を用いたBDS木が順検索は可能であるが，大きな木構造では処理時間が非常に長くなることに着目し，木構造を分割することにより，時間効率を改善する手法を提案した．

また，従来の手法では，先行順ビット列上での検索処理を可能にするため，BDS木内に含まれる1本の枝しか持たないノードには，ダミーリーフと呼ばれる擬似的な葉を付加していた．しかしながら，このダミーリーフ数が増加すると，ビット列が必要以上に長くなり，空間効率が低下する．そこで，BDS木をパトリシア構造に拡張し，ダミーリーフを用いずに，よりコンパクトなビット列に圧縮する手法，即ち，空間効率を改善する手法を提案した．

更に，本手法の時間効率，および空間効率の理論的評価だけに止まらず，実際に種々のキー集合に対して実験を行い，本手法を用いると，使用する記憶量を抑えつつ，処理速度を大幅に向上できることを実証した．

本提案手法は次のような特徴を持つ．

1) 本手法は，木構造のサイズが大きくなった場合に起こりうる時間効率の低下を防ぐことができる．

2) 本手法は，BDS木内に含まれるダミーリーフ数が多ければ多いほど，よりコンパクトなビット列に圧縮できる．

3) 本手法により圧縮されたビット列のサイズは，キー集合の特性による影響を受けない．

まず最初の特徴は，従来の手法が大規模なキー集合に対しては時間効率が低下してしまうという問題点を解決し，また，動的なキー集合に対して本手法が有効であることも示している．

次に第2の特徴に関してであるが，従来の手法では，ダミーリーフなしでは先行順ビット列に圧縮できなかった．そのため，ダミーリーフ数が多くなればなるほど，ビット列が必要以上に長くなり，空間効率が低下するという問題点を持っていた．しかしながら，第2の特徴により，空間効率が改善され，ダミーリーフ数を気にすることなくキー登録が行えることになる．

更に，従来の手法では，ダミーリーフを用いているため，各キーの値が不均等なキー集合に対しては，ダミーリーフ数の増減が避けられなかった．そして，このダミーリーフ数が多くなると，先行順ビット列のサイズも大きくなるため，生成されるビット列のサイズはキー集合の特性に大きく左右されていた．そのため，生成される先行順ビット列のサイズを事前に予測することは困難であった．このような問題点を解決するのが第3の特徴であり，この特徴によって，生成されるであろうビット列のサイズをキーの登録数に従って予測することが可能となる．

最後に今後の課題としては，分割管理されたビット列の均一化が挙げられる．特にBDS木の対称性が悪い場合，本手法では先行順ビット列の長さの不均一は避けられず，時間効率が低下する．そこで，今後は対称性が悪いBDS木に対してもビット長を均等に保つ階層化の手法を考案し，より深い評価を行っていく予定である．

第5章

文書処理への応用

5.1 緒　言

文書処理の分野，特に文書校正支援システム[39]，キーワード検索システム[18]，および情報検索システム[25]において，表記の揺れを自動的に統一する機能は必要不可欠な部分である．このような表記の揺れを統一するには，各表記の揺れのスタイルを格納した辞書を用い，辞書検索を行う必要がある．また，表記の揺れを高速に統一するためには，高速な検索能力を持つキー検索技法を適用する必要がある．しかしながら，表記の揺れのパターンをすべて辞書に登録する方法は辞書のメンテナンスの面で好ましくない．そこで，何だかの方法で揺れのパターンを自動生成する必要がある．本章前半部では，表記の揺れを有する語として，特に片仮名表記に着目し，片仮名異表記を片仮名表記変換ルールを用いて自動的に生成する手法を提案する．

本手法は，片仮名表記の揺れが外来語特有の表記方法に起因することに着目し，表記の揺れが存在する部分の音（文字列）を整理した片仮名ルールと，このルールが容易に適用可能な正規表記を定義する．そして，片仮名ルールと正規表記を用いた異表記生成手法を提案する．これにより，正規表記のみを辞書登録すればよいため，コンパクトで，かつすべての表記に対応可能な辞書が作成できる．本手法は，入力表記から正規表記を生成する正規表記生成部と正規表記から異表記を生成する異表記生成部から成る．また，本手法で用いる辞書検索法は，4章で述べた先行順ビット列を検索のための索引としているため，ディスクアクセス数が少なく，高速な検索が行える．更に，本片仮名異表記生成手法の有

効性を確認するため，１２０個のルールを定義し，約７万語のカタカナデータに対して実験評価を行った．その結果，辞書圧縮率は 79.5%，特に表記のゆれを有する語のみに対しては 42.6%に圧縮でき，異表記生成処理に関しては 99.0%の生成精度が得られ，本手法の有効性を確認した．また同時に，辞書の保守性，応用性を向上させることもできた．

また，近年のワードプロセッサー，DTP（Desk Top Publishing）などの文書処理システムの普及に従って，計算機上で文書の自動レイアウトを行うことが日常的になってきた．しかしながら，図表の配置に関しては，利用者自身が図表の配置座標を計算しなければならず，効率的な文書作成作業の妨げとなっている．このように，文書処理において図表の自動配置処理は重要な課題の一つである．そこで，本章後半では，適切な位置に図表を配置するための基準を定義し，これらの基準を満足し，しかも配置済みの文章や図表をページを越えて変更せずに図表を自動的に配置するアルゴリズムを提案する．

本手法は，文書構造を用いて，図表と参照箇所との位置関係を基準にする後方処理，全体的な配置バランスを基準にする均等処理，図表と関連する文章との位置関係を基準にする関連処理等を行う．また，本手法で用いる文書構造は**４章**で述べたコンパクトなデータ構造で格納されるため，高速な自動レイアウトが可能となる．更に，本図表配置手法の有効性を確認するため，実際に出版されている論文誌５０編（５５６の図表を含む）を調査した結果，94%の図表が本論文で定義した基準に適合しており，基準の正当性を確認した．更に，本手法で配置された図表の 82%が論文誌と同じ位置に，98%の図表が同ページ内に配置され，実際の論文誌レベルの図表配置を実現できることを確認した．また，提案されたアルゴリズムは１文書当たり平均 0.5 秒で実行できることも確認された．

以下，**５．２節**では片仮名異表記の生成手法として，２種類の片仮名ルール，および，それらの変換ルールが適用可能な標準的な表記スタイルを定義し，片仮名異表記を高速かつ正確に生成するアルゴリズムを提案する．次に**５．３節**では１２０個の片仮名ルールと約７万語の片仮名表記データに対して行った実験結果を示し，本手法の評価を行う．また，**５．４節**では，文書構造を用いた図表自動配置手法として，配置基準の定義，および自動配置アルゴリズムについて説明する．**５．５節**では，実際に出版されている論文誌５０編（５５６個の図表を含む）に対して，配置基準の妥当性，および自動配置アルゴリズムの評価を行い，**５．６節**では今後の計画と課題について触れる．

5.2 片仮名異表記の生成と統一手法への応用

5.2.1 片仮名異表記の重要性

近年，外来語の使用頻度が多くなるにつれ，日本語文書中に占める片仮名語の割合も急増している．これに伴って，平成４年の国語審議会報告では，「外来語の表記」が正式な内閣告示となった．この告示では，できるだけ緩やかな規範を前提としており，[ヴァイオリン]に代表されるような特殊表記が正式に認められた．特に，[ベネチア]に関しては，[ヴェネチア，ヴェネツィア，ベネツィア]の４通りの表記形式が認められた．しかしながら，片仮名語には異表記 [9]（katakana variant notations）が複数存在し，どの表記形式が用いられるべきかは統一がとれていない．そのため，日本語文書校正支援システム [51, 40, 57, 49, 50, 39]，仮名漢字変換システム [55, 8]，日英機械翻訳システム [9, 35, 44]，日本語ＯＣＲシステム [22]，及び情報検索システム [40, 25] 等では，辞書に登録されていない異表記は未登録語となり，処理精度の低下を招く．また，全ての異表記を辞書に登録する手法は，辞書の保守性，応用性において好ましくない．例えば，機械翻訳システムの対訳辞書では，個々の異表記に意味属性などの意味情報を付与する必要があり [35]，辞書の収録語数が膨大になる．また，日本語文書校正支援システムの校正辞書では，表記の統一を計るために，全ての異表記に対象文書の校正情報を付与する必要があり，辞書の応用性が乏しくなる．このように，自然言語処理システムにおいては，片仮名語の表記の揺れの解消は重要な課題となっている．

上記の課題を解決するため，高木ら [51]，宮崎ら [35]，川口ら [25] は，表記の揺れの可能性のある片仮名表記を全て辞書に登録していた．しかし，この手法では，辞書の登録単語数が非常に多くなり，辞書保守の負担を招く．また，大深 [40] は外来語の原表記（英語表記）の発音をもとにして表記の揺れを解決する手法を提案した．しかし，この手法では，片仮名表記の形態素情報に対応する原表記を付加する必要があり，辞書作成に多大な労力を要する．また，原表記と片仮名表記との子音並びが必ずしも同一でないため誤認識を起こす割合が高い．更に，相川ら [9]，島津ら [44] は片仮名表記に関するルールを用いて，片仮名異表記を生成する手法を提案した．しかしながら，ルール数が不十分かつ不正確であることより，異表記生成精度は実用化レベルに到達していないのが現状である．

そこで，本研究では，辞書の保守性を考慮した上で，表記の揺れのある片仮名語の異表

記を効率的かつ正確に生成することによって，片仮名表記の揺れを解消することを目的とする．

本手法では，上記の目的を効率的に実現するため，表記の揺れのある箇所の音（文字列）に着目した独自の異表記変換ルールを１２０個定義する．更に，保守性に優れた片仮名辞書を作成し，それらを用いた片仮名変換アルゴリズムを提案する．異表記変換ルールについては，異表記生成精度を高めるため，各ルールの特徴により大きく２種類に分類する．また，片仮名辞書は，その保守性を考慮し，表記の揺れのある片仮名語について，その中の一語のみを正規表記として辞書に登録することにより，辞書サイズを大幅に縮小する．更に，片仮名変換アルゴリズムに関しては，対象語の異表記をより正確に生成するため，表記の揺れのある語を辞書に登録されている正規表記に整形した後，その正規表記に関する異表記を生成する．以上の手法を用いることで，片仮名表記の揺れの発見，及び片仮名異表記のより正確な生成が可能になる．

5.2.2　片仮名異表記変換ルールの作成法

本手法で用いる片仮名異表記変換ルールは，表記の揺れのある箇所の音（文字列）に着目し，以下の項目を考慮して作成する．

(1)　揺れのある箇所の文字

(2)　揺れのある箇所に隣接する文字

(3)　揺れのある箇所の文字位置

作成の手順としては，まず，項目(1)に従ってルールを作成する．例えば，[ウィスキー](ウイスキー)からは，揺れのある箇所の文字だけに着目し，{ィ}から{イ}への変換を表すルールを作成する．また，このようなルールだけでは対応不可能な表記については項目(2)を適用し，ルールを拡張する．例えば，[スウィング](スイング)については，上記のルールでは対応不可能である．そこで，揺れのある箇所の前後の文字を付加したルールを新たに定義する．即ち，上記のルールを{ウィ}から{ウイ}，{スウィ}から{スイ}への変換を表すルールに拡張する．また，項目(3)に関しては，ルールをより正確に適用するために必要な情報である．そのため，ルールの適用位置を制約する条件をコード化してルール内に付加する．以上，作成したルールと逆のルールとを対とし，次項において述べる分類方法に従って，２種類のルール群に分類する．

5.2.3　片仮名異表記変換ルールの分類法

本手法では，揺れの部分を書き表す際に，最も原音に忠実な文字列を正規文字列（regular strings），それ以外を一般文字列（general strings）とする．そのため，特殊表記[1]，促音に関しては，これらを用いる文字列を正規文字列とする．例えば，[バイオリン]については，{バ}の原音は【ｖａ】だから，{ヴァ}を正規文字列とする．また，長音に関しては，用いない文字列を正規文字列とする．例えば，[フェース]については，{フェー}の原音が【ｆｅｉｓ】であるため，{フェイ}を正規文字列とする．更に，中黒点は片仮名複合語の区切り記号であるため，用いる文字列を正規文字列とする．また，[ベネチア]を[ヴェネツィア]と表記した場合，揺れが存在する文字列{ヴェ}，{ツィ}を正規文字列とし，{ベ}，{チ}を一般文字列とする．

次に，ルールの分類方法について説明する．各ルールの分類方法は，片仮名表記の揺れが正規文字列を用いるか否かによって発生していることに着目し，一般文字列から正規文字列への変換を行うルールは正規化ルール群（regular rules）に，一方，正規文字列から一般文字列への変換を行うルールは一般化ルール群（general rules）に分類する．例えば，{ベ}から{ヴェ}，{チ}から{ツィ}，{・}を用いないものから用いるもの等への変換に関するルールは正規化ルール群に，{ヴェ}から{ベ}，{ツィ}から{チ}，{・}を用いるものから用いないもの等への変換に関するルールは一般化ルール群に格納する．片仮名異表記変換ルール分類の定義を図 5.1 に示す．

ここで，一般文字列は正規文字列を包括する関係が成立するため，各ルールは次のように特徴づけられる．即ち，正規化ルール群は，全ての表記について必ずしも適用可能ではないが，一般化ルール群については，全ての表記に適用可能である．例として，一般文字列{バ}と正規文字列{ヴァ}に関するルール分類方法の概念図を図 5.2 に示す．図 5.2 に示されるように，正規文字列を用いている[ヴァイオリン]は一般文字列を用いた[バイオリン]に書き換え可能だが，一般文字列を用いた[バイク]，及び[バナナ]等は正規文字列を用いた[ヴァイク]，[ヴァナナ]等には書き換え不可能である．即ち，{バ}は{ヴァ}を

[1]特殊表記とは，「ティ」「ディ」「ヴァ」等の「現代仮名遣い」では用いない外来語特有の表記方法[26]示す．

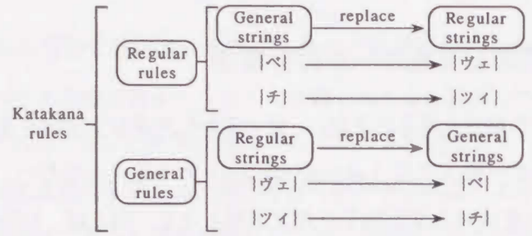


図 5.1　片仮名異表記変換ルール分類の定義

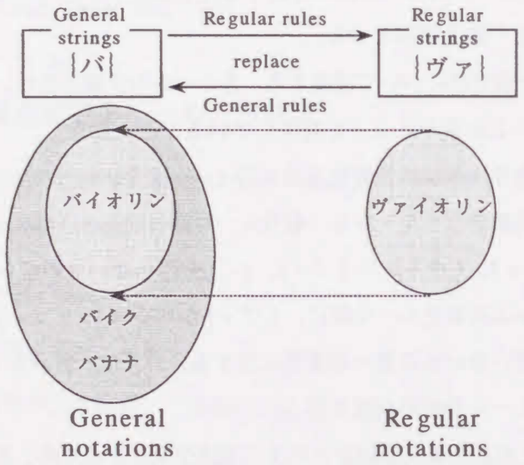
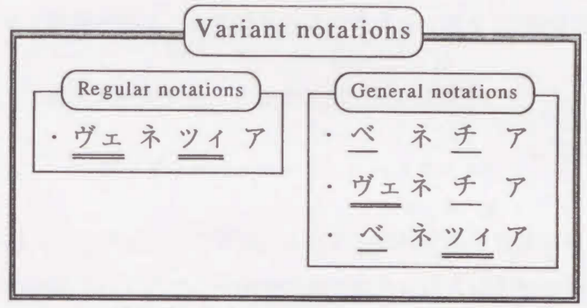


図 5.2　片仮名異表記変換ルール分類の概念図

包括する関係が成立し，{バ}から{ヴァ}への変換を行うルールは正規化ルール群に，また，{ヴァ}から{バ}への変換を行うルールは一般化ルール群に分類する．

従来の手法 [44] では，正規化ルール群と一般化ルール群を区別せずに定義し，同じレベルで用いている．そのため，誤変換が発生し，正確さにおいて問題を生じている．例えば，[バイク] に対して{バ}から{ヴァ}への変換ルールが適用されると [ヴァイク] が生



アンダーライン：General strings
ダブルアンダーライン：Regular strings

図 5.3　片仮名異表記分類の定義

成され，変換する必要がない文字列を変換してしまう．また，[ベネチア] に対して{チ}から{ツィ}または{ティ}への変換ルールが適用されると，誤変換として [ベネティア] が生成される．前者の誤り例は，表記の揺れの有無に関するマーカーを付加することにより防げるが，後者は避けられない誤りである．そこで，本手法では，上記の2つのルール群を区別して定義している．

更に，片仮名異表記については，揺れが存在するすべての箇所に正規化ルールが適用された異表記を正規表記（regular notations）とし，それ以外の異表記を一般表記（general notations）とする．図 5.3 に示す例では，[ベネチア]，[ヴェネチア]，[ベネツィア] が一般表記，[ヴェネツィア] が正規表記となる．但し，[バイク] 等の表記の揺れが存在しない語については，そのままの表記を正規表記とする．

以上のように，対となる片仮名ルールを区別して定義し，正規化ルール群は入力表記から正規表記を生成するために用い，一般化ルール群は正規表記から一般表記を生成するために用いることによって，より正確な異表記生成処理を行う．

5.2.4　片仮名異表記変換ルール構文

以下に，ＢＮＦ記法 [2] を用いた片仮名異表記変換ルールの生成構文を示す．但し，

ε　：空記号列；

x, y　：ストリング；

$A \rightarrow x \mid y$　：$A \rightarrow x$, $A \rightarrow y$；

$A \mid B \rightarrow x$　：$A \rightarrow x$, $B \rightarrow x$；

を表す．

【片仮名異表記変換ルール構文】

＜ルール＞ → ＜原形＞：＜変換情報＞♯

＜変換情報＞ → ε | ＜変換情報＞＜個別変換情報＞；

＜個別変換情報＞ → ＜位置制御情報＞＜変換文字情報＞

＜変換文字情報＞ → ε | ＜変換文字情報＞=＜変換文字列＞

＜位置制御情報＞ → ε | ＜位置制御情報＞（＜位置制御コード＞）

＜原形＞ | ＜変換文字列＞ → ストリング

＜位置制御コード＞ → 記号

♯ は各片仮名ルールの区切り記号を表す．

＜原形＞は，変換前表記における変換対象部分の文字列を表す．

＜変換文字列＞は，＜原形＞の変換候補となる文字列を表す．

＜位置制御コード＞は，表記中に存在する＜原形＞の位置を制約するコードであり，この制約条件を満足した場合にのみ変換処理を行う．

また，位置に関する制約がない場合は，＜位置制御コード＞は記述しない．

ここで，コード記号と制約条件との関係は以下のように定義する．

[位置制御コード]：[制約条件]

F：先頭文字列に存在する；

E：語尾文字列に存在する；

NF：先頭文字列に存在しない；

NE：語尾文字列に存在しない；

以後，例に従って構文について説明する．

まず，＜変換文字列＞が複数記述される構文例を以下に示す．

ウェイ：=ウェー=ウエー=ウエイ；♯

この例では＜原形＞が｛ウェイ｝であり，＜原形＞に対する＜変換情報＞には１つの＜個別変換情報＞が記述されており，＜個別変換情報＞には＜位置制御情報＞が記述されておらず，位置に関する制約条件は設定されていない．更に，＜変換文字情報＞には，｛ウェー｝，｛ウエー｝，｛ウエイ｝の３つの＜変換文字列＞の記述がある．即ち，このルールは，入力表記中のどこの位置に＜原形＞が存在しても上記の＜変換文字列＞に変換可能であることを表す．尚，このルールは一般化ルール群に格納されている．

次に，＜個別変換情報＞が複数記述される構文例を以下に示す．

クェ：=クエ；(NF)=ケ；♯

この例では＜原形＞が｛クェ｝であり，＜原形＞に対する＜変換情報＞には，２つの＜個別変換情報＞が記述されている．まず，最初の＜個別変換情報＞には，＜位置制御情報＞は記述されておらず，＜変換文字情報＞には＜変換文字列＞｛クエ｝の記述がある．また，２番目の＜個別変換情報＞には，＜位置制御情報＞として１つの＜位置制御コード＞(NF) が記述されており，＜変換文字情報＞には＜変換文字列＞｛ケ｝の指定がある．即ち，このルールは，入力表記中のどこの位置に＜原形＞が存在しても｛クエ｝には変換できるが，入力表記中の先頭位置に＜原形＞が存在しない場合に限り｛ケ｝に変換可能であることを示す．尚，このルールは一般化ルール群に格納されている．

以上の構文に従って，正規化ルール群には６３個，一般化ルール群には５７個の片仮名異表記変換ルールを定義した．

5.2.5　片仮名異表記生成手法

次に，図 5.4 に従って片仮名異表記生成手法を説明する．片仮名辞書には各種片仮名形態素が格納される．ここで，この片仮名辞書は保守性を考慮し，表記の揺れが存在する語

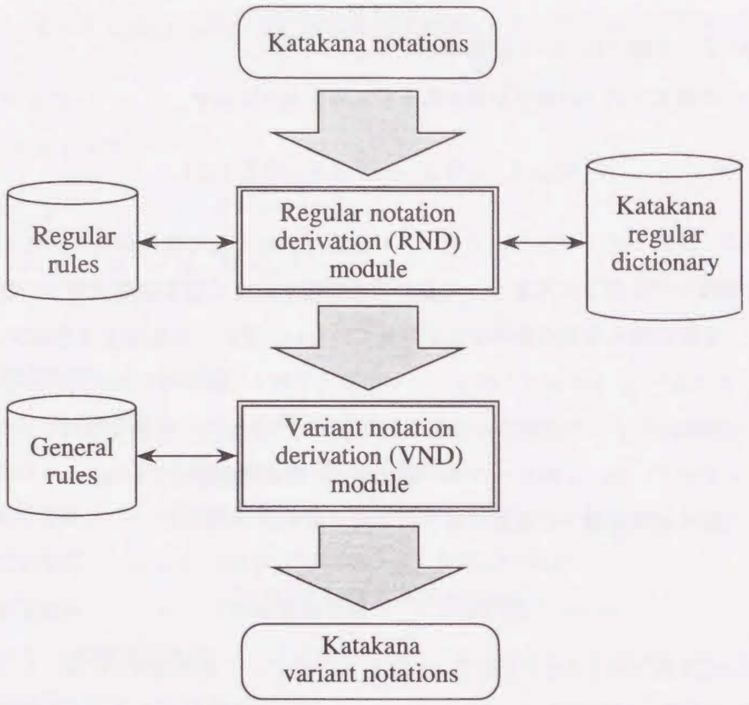


図 5.4　片仮名異表記生成処理の説明図

については，すべての異表記を登録するのではなく，正規表記のみを登録する．この正規表記のみを登録した辞書を特に正規化辞書（katakana regular dictionary）と呼ぶ．また，正規表記内において正規文字列が用いられている箇所には，揺れが存在することが示されている．そのため，本手法を用いると全ての異表記を辞書に登録する手間が省けるだけでなく，表記の揺れの有無，および表記の揺れの箇所等を示すマーカーを付加する必要もなくなる．

本システムの処理部は正規表記生成部（Regular Notation Derivation module ：RND module）と異表記生成部（Variant Notation Derivation module：VND module）から成る．まず，正規表記生成部では，正規化辞書，および正規化ルール群を用いて入力表記から正規表記への書き換えを行い，入力表記に対応する正規表記を生成する．また，ここでの処理において，最初の段階で正規化辞書検索を行うため，検索可能な表記は（正規表記の特徴より）ゆれが存在する可能性が低く，検索不能な表記はゆれが存在する確率が高いと判定できる．次に異表記生成部では，正規表記生成部において決定した正規表記を対象とし，一般化ルール群を用いた片仮名変換処理を行うことによって片仮名異表記を生成する．

以上の処理を例[ベネチア]で説明する．まず，正規表記生成部において，[ベネチア]は[ヴェネツィア]で正規化辞書に登録しているため検索失敗となるが，正規化ルール群と正規化辞書を用いた片仮名変換処理を行い，正規表記として[ヴェネツィア]を生成する．その後，処理は異表記生成部に移り，正規表記[ヴェネツィア]を対象として，一般化ルール群を用いた片仮名変換処理を行う．その結果，[ヴェネツィア]，[ベネツィア]，[ヴェネチア]，[ベネチア]の4通りの異表記が生成される．以上のように，揺れを有する片仮名語をルール適用が容易な正規表記に変換した後，異表記生成を行うことによって，より適切な異表記を生成することが可能となる．

次に，前述した片仮名異表記変換ルールを用いた片仮名異表記変換アルゴリズムを示す．図5.5と例を用いて本アルゴリズムの概要を説明する．尚，図5.5で破線で囲まれた部分は正規表記生成部でのみ追加されるルーチンである．即ち，異表記生成部においては破線部を除いた部分でカタカナ変換がなされる．

ここで，本異表記変換アルゴリズム内で用いられる用語の説明を行う．

まず，入力表記（input notation）とは，正規表記生成部の場合では正規化辞書検索の結果，未登録となった表記を示す．また，異表記生成部の場合には正規表記生成部で決定した正規表記を示す．

ルール集合（rule set）は個々の片仮名異表記変換ルールの有限集合であり，正規表記生成部で用いられた場合には正規化ルール群を示し，異表記生成部で用いられた場合には一般化ルール群を示す．

適用候補ルール集合（candidate rule set）は，入力表記に対して適用の候補となる片仮名異表記変換ルールの有限集合である．

適用ルール集合（applicable rule set）は，入力表記に対して適用可能な片仮名異表記変換ルールの有限集合である．

生成表記（generated notations）は，適用ルール集合内に含まれる個々の片仮名異表記

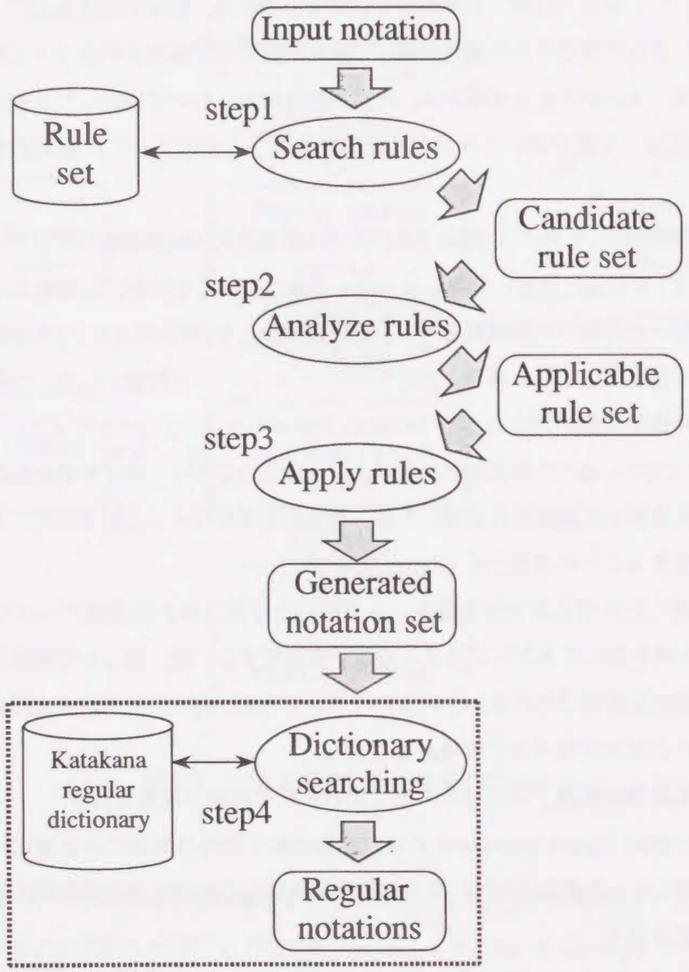


図 5.5　片仮名異表記変換アルゴリズム説明図

変換ルールを入力表記に適用した結果，新しく生成される表記であり，また，生成表記集合（generated notations set）は，生成表記の有限集合である．

以下に片仮名異表記変換ルールを用いた片仮名異表記変換アルゴリズムを示す．但し，次のような変数を用いる．

str　：入力表記；

rule　：個々の片仮名異表記変換ルール；

RULE_SET　：片仮名異表記変換ルール集合；

C_SET　：適用候補ルールの集合；

A_SET　：適用ルールの集合；

new_str　：生成表記；

NEW_SET　：生成表記の集合；

尚，*NEW_SET* の初期状態は *NEW_SET*={*str*} とする．以下に本アルゴリズムの流れを示す．

【片仮名異表記変換ルールを用いた片仮名異表記変換アルゴリズム】

入力：正規表記生成部：正規化辞書検索の結果，未登録となった入力表記；

　　異表記生成部：正規表記生成部で生成された正規表記；

出力：正規表記生成部：正規化辞書の検索成功ならば生成された正規表記，失敗ならば入力表記を未登録語として返す；

　　異表記生成部：新しく生成された生成表記集合；

手順1　：{片仮名異表記変換ルールの検索処理}

　str に対して適用可能な *rule* を *RULE_SET* 内から検索し，それを *C_SET* に代入する；

　この検索は<原形>の文字列（*key* とする）と *str* との最長一致法で行う；

手順2　：{片仮名異表記変換ルールの解析処理}

　C_SET 内の各 *rule* に対して，<位置制御コード>の真偽を調べる[2]；

　その結果，真の *rule* のみを *A_SET* に代入する[3]；

手順3　：{片仮名異表記変換ルールの適用処理}

　A_SET 内の各 *rule* を *str* に適用し，新しく生成された *new_str* を *NEW_STR* に追加する；

[2]<位置制御コード>が記述されていない場合は真と判断する．

[3]*rule* 内に複数の<個別変換情報>が含まれている場合，<位置制御コード>が偽である<個別変換情報>が削除された *rule* が新しく代入される．

尚，異表記生成部では，手順3までのNEW_STRが出力結果となるが，正規表記生成部では，手順3が終了後，更に手順4に進む；

手順4　：{正規化辞書の検索処理}

NEW_SET内の各new_strについて正規化辞書の検索を行い，検索が成功したnew_strを正規表記とする；

但し，全てのnew_strについて検索が失敗ならば，strを未登録語とする；

次に，入力表記をstr=[インタフェース]とした場合を例にし，上記のアルゴリズムの流れを説明する．

まず，正規表記生成部では，以下のような手順が行われる．

【正規表記生成部】

手順1　：$RULE_SET$として正規化ルール群を用い，key=[フェー]を発見し，対応する$rule$をC_SETに格納することにより，

C_SET={[フェー：=フェイ；♯]}

となる；

手順2　：$rule$=[フェー：=フェイ；♯]は<位置制御コード>が記述されていないため真となり，その$rule$はA_SETに格納され，

A_SET={[フェー：=フェイ；♯]}

となる；

手順3　：$rule$=[フェー：=フェイ；♯]を初期状態の

NEW_STR={インタフェース}

に適用すると，new_str=[インタフェイス]が得られ，

NEW_STR={インタフェース，インタフェイス}

となる；

手順4　：NEW_STR内の各要素に対して，中黒点を考慮した正規化辞書検索[4]をすると，[インタ・フェイス]が検索成功となり，この表記形式を正規表記とする；

次に，片仮名異表記[インタ・フェイス]を正規表記として異表記生成部に入力した場合に行われる手順を以下に示す．

[4]デジタル検索によるストリングマッチ[5]を行う際，各文字間に中黒点が存在すると仮定しながら行う．

【異表記生成部】

手順1　：$RULE_SET$として一般化ルール群を用い，

str=[インタ・フェイス]に対して，key=[タ]，[フェイ]を発見し，

対応する$rule$をC_SETに格納することにより，

C_SET={[タ：(E)=ター；♯]，[フェイ：=フェー=フエイ；♯]}

となる；

手順2　：$rule$=[タ：(E)=ター；♯]，[フェイ：=フェー=フエイ；♯]共に<位置制御コード>は真となり[5]，各$rule$はA_SETに格納され，

A_SET={[タ：(E)=ター；♯]，[フェイ：=フェー=フエイ；♯]}

となる；

手順3　：$rule$=[タ：(E)=ター；♯]を初期状態の

NEW_STR=インタ・フェイス

に適用すると，new_str=[インター・フェイス]が得られ，

NEW_STR={インタ・フェイス，インター・フェイス}となる；

また，$rule$=[フェイ：=フェー=フエイ；♯]をNEW_STRの各要素に適用すると，

new_str=[インタフェース]，[インターフェース]，[インタフエイス]，[インターフエイス]が得られ，

NEW_STR={インタ・フェイス，インター・フェイス，インタフェース，インターフェース，インタフエイス，インターフエイス}

となる；

更に，出力される異表記は上記のNEW_STRの各要素から中黒点を除いた表記形式も追加される；

[5][インタ・フェイス]のように，入力文字が中黒点を用いた複合語の場合は，接頭語，接尾語を分割して<位置制御コード>を適用する．

5.3　片仮名異表記生成手法の評価

本片仮名異表記生成システムは，片仮名辞書圧縮，片仮名辞書構築，片仮名異表記変換ルール構築，及び片仮名異表記変換処理ルーチンで構成されており，C言語でインプリメントした．片仮名辞書圧縮ルーチンは，一般的な辞書（以後，基本辞書と呼ぶ）から正規表記のみを抽出し，正規化辞書を作成する．ここで，基本辞書としては，EDR概念辞書とICOT形態素辞書を中心に抽出した約7万語（約1．5万語の表記の揺れを有する片仮名語を含む）の辞書を準備した．尚，基本辞書内の表記に対しては，揺れの有無に関するマーカーを付加している．一方，正規化辞書としては，基本辞書から正規表記のみを抽出し，全てのマーカーを排除した辞書を作成した．片仮名辞書構築ルーチンでは，先行順ビット列により表現された2進木トライ（4章で述べた改良手法が施された木構造）により正規化辞書を構築する．また，片仮名異表記変換ルール構築ルーチンも同様に各種ルールを2進木トライにより構築する．

次に，本手法の有効性を確認するため，辞書の保守性と異表記生成精度の2点に関する評価を行った．尚，今回の実験では，１２０個の片仮名異表記変換ルール（正規化ルール群として６３個，一般化ルール群として５７個）を用いた．

まず，辞書の保守性に関しては，基本辞書と正規化辞書とのサイズを比較するため，辞書圧縮率（正規化辞書のサイズ／基本辞書のサイズ）を求めた．その結果，全データに対しては79.5%，表記の揺れを有する片仮名語のみに対しては42.6%の辞書圧縮率が得られた．以上の結果から，表記の揺れを有する片仮名語が辞書中に含まれる割合が高くなればなるほど，本手法では辞書圧縮の割合が高くなり，辞書の保守性が向上すると言える．

次に，異表記生成精度に関しては，島津ら[44]提案した手法と本手法との異表記生成処理の正確性を比較するため，7,500語の片仮名データ（揺れを有する片仮名語を含む）に対して異表記生成処理を行い，以下の式で表される生成精度を求めた．

生成精度＝（生成個数－失敗個数）／生成個数；

ここで，従来の手法としては，辞書としては基本辞書を用い，正規化ルール群と一般化ルール群を区別せずに同じレベルで適用することにより生成処理を行うシステムを用いた．また，生成した異表記が正しいか否かの判定は，基本辞書に登録されているものは正しいと判断し，未登録のものについては片仮名語に関する専門辞書[26, 48]を参照した．その結果を表5.1に示す．

表5.1　片仮名異表記生成に関する実験結果

	従来の手法	本手法
(A) 未登録語数	1,125	1,097
(B) ゆれを有する表記数	738	766
(C) 生成異表記数	1,994	1,747
(D) 生成失敗数	491	18
(E) 異表記生成精度 (%)	**75.4**	**99.0**

表5.1から，本手法を用いることにより，生成精度が非常に高くなり，正確な異表記生成処理を行えることが分かる．更に，本手法を用いると“未登録語数”が以下のように２８個減少し，

[従来の手法のＡ項目] - [本手法のＡ項目] ＝ ２８個

その減少分だけ“揺れを有する表記数”が以下のように増加していることが分かる．

[従来の手法のＢ項目] ＋ ２８個 ＝ [本手法のＢ項目]

これは，従来の手法が，基本辞書に登録されていなければ全て未登録語として処理していたのに対し，本手法では，基本辞書に登録されていない語でも，その正規表記が正規化辞書に登録されてさえいれば未登録語とならないことに起因する．例えば，基本辞書に[インターフェイス]が登録されていなくても，正規化辞書に正規表記[インタ・フェイス]が登録されていれば片仮名変換処理の結果，未登録語としては処理されず，揺れを有する表記として処理されることになる．即ち，本手法は，表記の揺れを有する語が未登録語となる確率を抑制し，表記の揺れの発見率を高める効果をもっている．これは，多数の異表記を有する語について，本手法が有効であることをも示している．

尚，本手法において生成処理が失敗した片仮名語は，意味的曖昧性を有しており，現在のアルゴリズムでは対処することができないものである．これは，表記の揺れを有する片仮名語が複数の違った意味もつ場合に発生する．例えば，[バレー]という語は，その意味が舞踏を表しているならば[バレエ]と変換できるが，球技を表しているならば変換できない．即ち，このような片仮名語は，それ自体が表している意味を確定しなければ正しい変換処理は行えない．

また，異表記生成処理時間に関して，本手法では正規表記を決定する際に，2度の辞書検索を行い，更に，正確な異表記を生成するために，2種類の片仮名異表記変換ルールに対して検索処理を行わなければならない．そのため，従来の手法に比べて1.8倍の処理時間を必要とする．しかしながら，本システムでは，高速な辞書検索法を用いているため，*DELLOptiPlexGXMT*5133上で約1万語／秒の処理速度が得られており，十分に実用化に耐え得るものであると思われる．

尚，今回の実験は，中黒点について考慮していない．中黒点を考慮することで，更に生成精度は増すであろう．また，片仮名ルールに，より検討を加えることによって，辞書圧縮率，及び生成精度の値はより向上し得ると思われる．また，本手法を文書校正知識獲得手法[46]に併合することにより，片仮名語の揺れを自動訂正することも可能となる．

5.4　文書構造を用いた図表自動配置手法への応用

5.4.1　図表自動配置機能の必要性

文書内の図表は説明の内容を容易にするための手段であり，その配置位置は読み易さを向上させる重要な要素である．しかし，十分な整形能力をもつWYSIWYG（What You See Is What You Get）方式の文書整形システムが少ないため，利用者自身が自分の好みを反映した場所に図表を配置するためには，図表の参照箇所や図表の大きさ等を考慮して配置座標を計算する必要があり，その配置が適切か否かの評価も難しい．そこで，前処理としてシステムが適切と判断した位置に図表を配置した文書を作成した後，プレビュー機能を備えたマン・マシンインタフェース[16]を用いて図表配置の調整を行う方法が用いられる．ここで，前処理の内容が優れていれば調整作業も少なくなり，文書作成の能率も高まる．従って，より適切な位置に図表を自動配置する手法は，文書整形システムに対する一つの重要な課題である．

troff／nroff[27]，Scribe[41]，浄書[43]等の文書整形システムでは，挿入する図表の座標をソースファイル内で指定する必要があり，図表と参照箇所との最適な位置関係を計算する処理の自動化は実現されていない[42]．これに対して，TEX[29]，LATEX[21]は，table環境やfigure環境を用いて図表が配置される位置の優先順位をオプション引数として指定すれば，参照箇所との関係も考慮した上で図表を自動的に配置する．また，山口ら[56, 15]は文書の論理構造と参照箇所の位置等を用いて，図表を自動的に配置する手法を提案した．しかし，これら従来の手法では，幅の異なる図表間の配置位置に関する優先度が設定されてないため，同一ページ内に幅の異なる図表を配置する場合，幅の広い図表より幅の狭い図表の方がページの端に配置され，見栄えが損なわれる．また，従来の手法は文書全体ではなくローカルなページ単位の最適化を行っているため，大きな図表が何個も連続して参照された場合，それらの図表が次々に後方へ追いやられ，文書全体の配置バランスが悪くなる．更に，ページ単位の最適化に関しても，配置済みの図表を再配置しないため，同一ページにサイズの異なる複数の図表をうまく配置できない．

また，WYSIWYGの概念を有している*Microsoft WORD*等の文書編集システムでは，図表と文章との関係を定義することが可能である．これらのシステムでは，図表，およびその図表と関係のある文章を同じページや同じカラムに配置するため，各図表には，図表

と対応する文章（段落）との間の配置関係にリンクを張るアンカーを持たせている．しかしながら，文書のスタイルが2段組み以上の場合，文書を編集中にリンクが張られた関連段落が隣のカラムに移動したとしても，これらのシステムでは，他のカラムに図表を移動できないため，より適切な位置に図表を再配置することができない．すなわち，これらのシステムは，ページ内の再配置処理の機能を持たない．例えば，2段組みの文書に対して，図表と対応する段落が同じカラムに配置されるようにアンカーがセットされているとする．このような状況下で，アンカーがセットされている段落が文書編集によって隣の段落に移動した場合，対応する図表は元のカラム内に止まったままで，移動した段落に追随することができない．更に，アンカーが図表と関連段落とが常に同じページに配置されるようにセットされている場合，関連段落が文書編集中に隣のページに移動すると，アンカーがセットされている図表も同時に次のページに移動してしまう．その結果，元のページの下部に空領域が必然的に生成されてしまう．このような場合，もし，図表と関連段落が共に同じページに配置できなければ，図表が配置されたページに無理矢理に関連段落を配置するのではなく，次のページの上部に図表だけをまず配置し，関連段落は元のページに分割して配置すべきである．

本節では，まず，出版済みの論文誌を調査し，適切な位置に図表を配置するための基準を定義する．これらの基準は，参照箇所の位置，文章と図表の関連性，文書全体における図表配置の均等性，図表幅による配置位置の優先度等に関するものである．そして，文書構造データ [12]（文書の論理的な構造や参照箇所の位置等）と図表データ（図表のサイズ等）を用いて基準を満たす位置に図表を自動配置する手法を提案する．尚，本論文で対象にする文書スタイルは，1段組み，および，2段組みとし，図表サイズは，フレーム（カラム）幅に整合する片幅図表，および，ページ幅に整合する両幅図表を対象とする．

本手法は，図表を配置すべきページとその中での上下位置（これらのデータを概略位置と称す）を決定する概略位置決定部（General Position Decision Module：GPDM），及び概略位置を基にしてページの上端から図表の上端までの行数（このデータを詳細位置と称す）を決定する詳細位置決定部（Detailed Position Decision Module：DPDM）より成る．

概略位置決定部は，

(1)　参照箇所の後方近くに配置位置を定める**後方処理**；

(2)　文書内に図表を均等に配置する**前方処理**；

(3)　配置すべきでない場所を避けて図表を配置する**制約処理**；

を行う．一般に，前方，制約処理では既配置の文章や図表をページを越えて変更する（バックトラックと呼ぶ）必要があり，処理時間が長くなるが，本手法ではこのバックトラックを解消するために文章と図表の残量を考慮した高速なアルゴリズムを提案する．

また，詳細位置決定部は，幅の広い図表を優先的にページ端に配置したり，同じ幅の図表ならば図表番号の小さい図表ほど上部に配置する等の優先度を考慮し，詳細位置を決定する．

以下，5．4．2では図表をより適切な位置に配置するための基準を定義するが，本論文では各学会誌論文を対象にこの基準を決定する．5．4．3では図表配置処理の全体の概要を述べ，5．4．4では概略位置決定部の各処理について，5．4．5では5．4．4のアルゴリズムより得られた概略位置データを基にした詳細位置決定部の処理について述べる．

5.4.2　図表配置基準の定義

より適切な位置に図表を配置するための基準を，情報処理学会と電子情報通信学会の論文を対象に調査し，次のような基準を決定した．

(1) **順序基準**　：文書内では昇順に図表を参照するのが原則であるので，図表を番号順に配置する基準；

(2) **安定基準**　：図表の座りを安定させ，文章領域と図表領域との境を明確にするため，ページの上部または下部に配置する基準；

(3) **後方基準**　：文書内の話の流れは後方向であるため，読者が現在参照すべき図表を自然に目で追えるようにするために，参照箇所の後方近くに配置する基準；

(4) **均等基準**　：後方基準や関連基準のみでは，図表が集中して現れる場合に図表が後方へと追いやられ，参照箇所と図表の位置が大きく離れてしまう．そこで，全体的に

バランス良く図表を配置するために，参照箇所から一定のページ内に図表を配置する基準；

(5) 制約基準　：論文では，始章内，終章以降の領域には図表が配置されない．このように，配置すべきでない場所を避けて図表を配置する基準；

(6) 優先基準　：幅の広い図表を優先的にページ端に配置し，同じ幅の図表の場合は，図表番号の小さい図表を優先的に上部に配置する基準；

次に，本手法で用いる各種データの定義を行う．

文書は2段組（1ページ内に2フレーム[56]が存在する），図表は1フレーム幅に整合する図表（片幅図表と呼ぶ）とページ幅の図表（両幅図表と呼ぶ）を対象とし，以下のような形式のデータを定義する．

(1) p番目の段落に対するデータの形式

$P\_ATTRI(p)$　：見出し，始章・終章見出し（特殊見出しと呼ぶ），本文の何れかの論理構造で示される段落の属性；

$P\_REF(p)$　：段落内で参照されている図表番号dとその参照箇所までの行数h[6]の組(d, h)から成る集合；

図5.6の例では，

$P\_ATTRI(p)$=本文

$P\_REF(p)$={(d+1,20), (d+2,20), ($last$[7],5)}

となる．

尚，図中の$r_p\ i$はi番目の図表の参照箇所を示す．

(2) d番目の図表に対するデータの形式

$D\_LINE(d)$　：図表行数；

$D\_WIDTH(d)$　：片幅図表は$SINGLE$，両幅図表は$DOUBLE$なる値をもつ図表の幅；

$D\_POS(d)$　：フレーム番号，フレーム内の上下位置を示す値（上部はTOP，下部は$BOTTOM$），フレームの上端から図表上端までの行数の組；

[6]段落内に2個以上の参照箇所があれば，一つ前の参照箇所からの行数をhとする．

[7]$last$は段落の終端を表す記号とする．

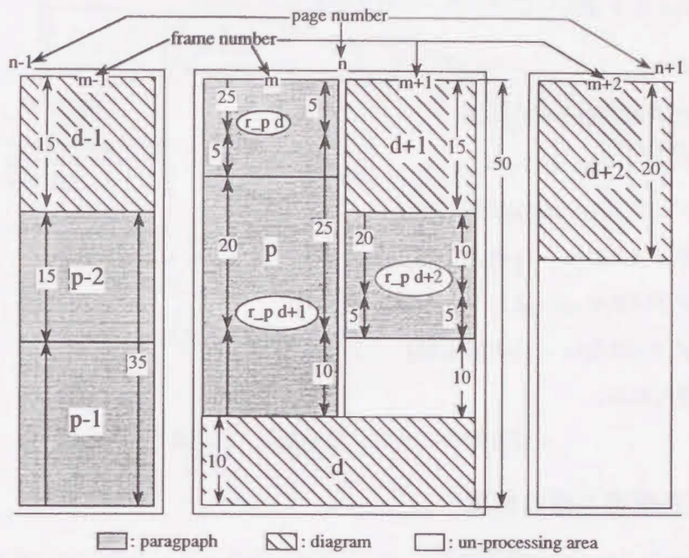


図5.6　各種データ形式の説明図

図5.6の例では，

$D\_LINE(d)$=10；

$D\_WIDTH(d)$=$DOUBLE$；

$D\_POS(d)$=(m, $BOTTOM$, 40)；

$D\_LINE(d+1)$=15；

$D\_WIDTH(d+1)$=$SINGLE$；

$D\_POS(d+1)$=(m+1, TOP, 0)；

となる．

(3) f番目のフレームに対するデータの形式

$F\_PAGE(f)$　：フレームが属するページ番号；

$F\_MAX(f)$　：フレーム内に割付けれる最大行数；

$F\_AVAIL(f)$　：フレーム内の未使用行数；

$F\_PARA(f)$　：フレーム内に割付けられる文章の行数と文章内での参照箇所の位置，尚，
F_PARAはP_REFと同じデータ形式である；

図 5.6 の例では，

$F\_PAGE(m+1) = n$；
$F\_MAX(m+1)$=50；
$F\_AVAIL(m+1)$=10；
$F\_PARA(m+1)$=(d+2,10), ($last$,5)
$F\_PARA(m)$=(d,5), (d+1,25), ($last$,10)
$F\_PARA(m-1)$=($last$,35)

のデータが得られる．

5.4.3　図表配置処理の概要

本手法で用いる片仮名異表記変換ルールは，表記の揺れのある箇所の音（文字列）に着目し，以下の項目を考慮して作成する．

(1)　揺れのある箇所の文字

(2)　揺れのある箇所に隣接する文字

(3)　揺れのある箇所の文字位置

本手法で図表の配置位置を決定する全体的な流れを図 5.7 に示し，各モジュールの機能を以下に説明する．

ユーザから入力される文書（input document）は，図表が挿入されていない文書（源文書と呼ぶ）とする．図表については，図表番号と図表幅のみを指定し，図表幅が片幅指定ならばフレーム幅に，両幅指定ならばページ幅に縦横同じ比率で縮小（拡大）した図表の縦の長さを行数に換算した値が図表行数となる．フレームに関しては，フレーム幅やフレーム行数等をユーザから受け付け，F_PARA以外のフレームデータを作成する．

その後，文書構造抽出部（Document Structure Extraction Module：DSEM）では源文書の論理構造（logical structure）と参照構造（reference structure）（参照位置や参照されている図表番号）の抽出し，その結果を先行順ビット列で表現された 2 進木構造に格納する．これにより，論理構造および参照構造の順検索が高速に行えるため，図表配置の処理

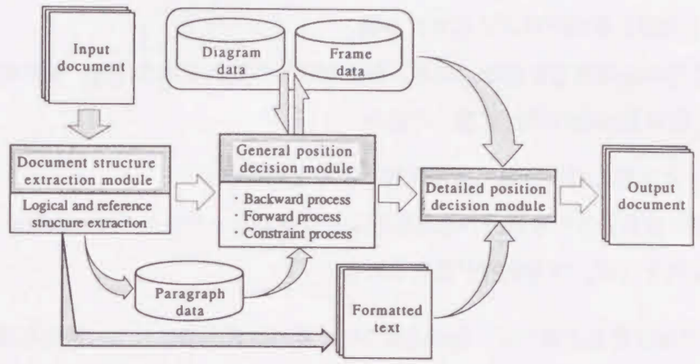


図 5.7　図表配置システムの概要図

時間の高速化がはかれる．尚，本手法では土井ら [12] が提案したルールを用いて上記の構造を自動抽出する．また，ここでは，源文書をフレーム幅に整合させたフレーム幅文書（formatted text）の作成を行いながら，すべての段落データを構築する．

次に，概略位置決定部では，まず最初に，各段落毎のフレーム幅文書を配置し，文章領域の割り付け（F_PARAの設定）を行う．そして，配置された段落内に参照箇所があれば，その参照箇所に対応する図表の概略位置を決定する．この概略位置決定部は，先に定義した配置基準を満たすため，後方処理（backward process），前方処理（forward process），制約処理（constraint process）と呼ばれる 3 つの処理から構成される．各処理は，以下のように，それぞれ対応する基準を満たす．

後方処理　：**後方基準**，順序基準，安定基準，優先基準を満たす位置に図表を配置する；

前方処理　：**均等基準**，順序基準，安定基準，優先基準を満たす位置に図表を配置する；

制約処理　：**制約基準**，順序基準，安定基準，優先基準を満たす位置に図表を配置する；

順序基準，安定基準，優先基準はすべての処理で満たされることになるが，残りの 3 基準（後方基準，前方基準，制約基準）は，処理対象となる段落の属性値によって，または，未処理の図表行数が未処理の段落行数より多いか否か等によって，各処理が選択される．この各処理の選択手順を以下に示す．

【各種処理の選択アルゴリズム】

手順1　：{処理対象の段落による選択処理}

処理段落の論理構造が始章，終章，参考文献等の制約領域の場合，**制約処理**が実行され，それ以外ならば，手順2に進む；

手順2　：{未処理の図表行数・段落行数による選択処理}

未処理の図表行数が未処理の段落行数よりも多くなった場合，**前方処理**が実行され，それ以外ならば，**後方処理**が実行される；

最後に，詳細位置決定部では，先に決定された概略位置を基にして，**優先基準**を考慮しながら詳細位置（*D_POS*の第3要素）を決定し，最終的に文章と図表をページ上に割付けるが，ここで得られる詳細位置をTEX[29]，LATEX[21]等の記述に利用することも可能である．

このように，2段階で配置座標を決定する理由は，効率的な配置処理を行うためである．例えば，図5.8のように図表がA, B, Cの順序で優先基準を満たしながら配置される場合，一度に配置座標を決定すれば，Bが処理される際にAの座標を，Cが処理される際にはB, Aの座標を再計算する必要がある．そこで，本手法では全ての図表について一旦概略位置まで求めた後，座標の再計算を必要としないアルゴリズムによって各図表の配置座標を求める．

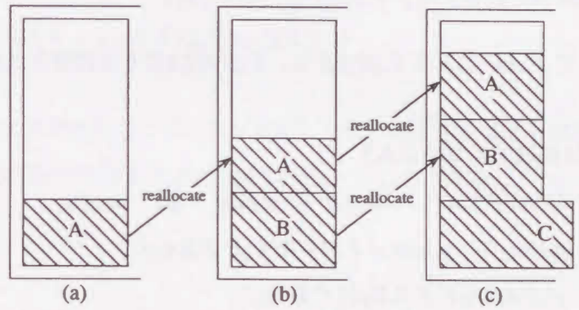


図5.8　2段階処理の説明図

5.4.4　概略位置決定処理の概要

本項では，まず，フレーム幅文書に対して行われる文章領域割り付け処理について説明し，その後，概略位置決定部で行われる各種処理（後方処理・前方処理・制約処理）の詳細について述べる．

(A) 文章領域の割付け処理

処理中の段落を p とすると，$P\_REF(p)$ 内の各組に対して以下の操作を行うことによって F_PARA を得る．尚，用いる変数を以下のように定義する．

$P\_FRA(p)$　：段落 p を配置するフレーム番号；

$(x,\ p\_line)$　：段落 p の段落データ $P\_REF(p)$ 内の各組；

但し，各フレームの F_AVAIL は F_MAX に，F_PARA は0に初期化されているものとする．

【文章領域の割付け処理アルゴリズム】

while　$F\_AVAIL(P\_FRA(p)) \leq p\_line$　**do**　**begin**
　$p\_line \leftarrow p\_line - F\_AVAIL(P\_FRA(p))$;
　$F\_PARA(P\_FRA(p))$ の最後に
　　$(last,\ F\_AVAIL(P\_FRA(p)))$ を加える;
　$F\_AVAIL(P\_FRA(p)) \leftarrow 0$;
　$P\_FRA(p) \leftarrow P\_FRA(p)+1$　**end**
$F\_PARA(P\_FRA(p))$ の最後に $(x,\ p\_line)$ を加える;
$F\_AVAIL(P\_FRA(p)) \leftarrow F\_AVAIL(P\_FRA(p)) - p\_line$;

尚，F_PARA に新たな組を追加する際には，次の併合規則に従って行う．

【併合規則】

隣接する2つの組を $(x,\ x\_line)$，$(y,\ y\_line)$ とすると，
$x = last$ ならば $(y,\ x\_line + y\_line)$ に併合できる；

図 5.7 において，$F\_PARA(m)$={$(d$,5)，$(last$,5)} の状態に段落 p を配置する例を説明

する．

まず，$P\_REF(p)$ の第1組 (d+1,20) を配置すると，

$F\_PARA(m)$={(d,5)，($last$,5)，(d+1,20)}

となるが，併合規則により，

$F\_PARA(m)$={(d,5)，(d+1,25)}

になる．

次の組 (d+2,20) に対しては，段落が分割され

$F\_PARA(m)$={(d,5)，(d+1,25)，($last$,10)}
$F\_PARA(m+1)$={(d+2,10)}

となり，最後の ($last$,5) で，

$F\_PARA(m+1)$={(d+2,10)，($last$,5)}

となる．

また，本手法は参照箇所までの文章を配置した後，対応する図表を処理するので，処理対象となる図表の参照箇所は $P\_FRA(p)$ のフレームに存在する．

(B) 後方処理

処理対象が片幅と両幅図表である場合の配置処理法を個別に提案する．

(B-1) 片幅図表の後方処理

後方処理では**順序基準**を満たすために，図表 d を図表 d-1が配置されているフレーム以降に配置する．また，**後方基準**を満たすために，参照箇所と図表が同フレームならば下部に，異フレームならば図表が配置されるフレームの上部に配置する．以下に配置手順を示す．尚，処理中の図表 d を配置するフレーム番号を $D\_FRA(d)$ とする．

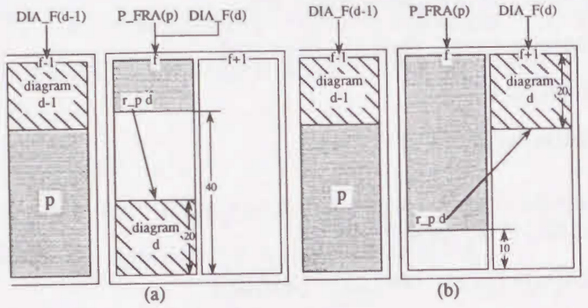


図 5.9　片幅図表の配置例

【片幅図表の後方処理アルゴリズム】

手順 (1-1)　：{ 図表行数以上の空き領域があるフレームの検索 }

$D\_FRA(d) \leftarrow D\_FRA(d-1)$ とした後，

$D\_LINE(d) \leq F\_AVAIL(D\_FRA(d))$ となる間，

$D\_FRA(d) \leftarrow D\_FRA(d)+1$ を行う；

手順 (1-2)　：{ 参照位置に従った図表の配置 }

$D\_FRA(d)$ と $P\_FRA(p)$ が同フレームならば，

$D\_POS(d) \leftarrow (D\_FRA(d), BOTTOM, *$[8]$)$ とし，

異フレームならば，$D\_POS(d) \leftarrow (D\_FRA(d), TOP, *)$ とする；

手順 (1-3)　：{ フレーム内の空き行数の変更 }

$F\_AVAIL(D\_FRA(d)) \leftarrow F\_AVAIL(D\_FRA(d)) - D\_LINE(d)$ を行う；

次に，上記のアルゴリズムの流れを例（図 5.9 ）に従って説明する．図 5.9-(a) の例は，図表 d の参照箇所を含む段落 p が f-1 番目のフレームから配置され，また，図表 d が f 番目のフレーム内で参照されている状況を示す．このような状況下で，図表 d は以下のような手順に従って配置される．尚，$P\_FRA(p)$=f，$D\_LINE(d)$=20 とする．

手順 (1-1)　：まず，$D\_FRA(d) \leftarrow f-1$ とした後，

[8] この段階で第3要素の値は決定しないため，'*' で記述する．

$F\_AVAIL(f-1)$=0 であるが，$F\_AVAIL(f)$=40 であり，

$D\_LINE(d)$=20 より多いので，$D\_FRA(d) \leftarrow f$ とする；

手順 (1-2)　：$D\_FRA(d)$ と $P\_FRA(p)$ とが同じフレーム番号 f を示しているので，

$D\_POS(d) \leftarrow (f, BOTTOM, *)$ とする；

手順 (1-3)　：$F\_AVAIL(f)$=40 から $D\_LINE(d)$=20 を差し引き，

$F\_AVAIL(f) \leftarrow 20$ とする；

一方，段落 p が図 5.9-(b) のように配置され，$F\_AVAIL(f) = 10$ となる場合，図表 d は以下のような手順に従って配置される．

手順 (1-1)　：まず，$D\_FRA(d) \leftarrow f-1$ とした後，

$F\_AVAIL(f) = 10 < D\_LINE(d) = 20$ であり，

$F\_AVAIL(f+1) = 40 > D\_LINE(d) = 20$ なので，

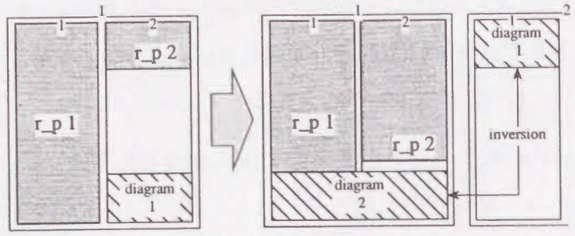
$D\_FRA(d) \leftarrow f+1$ とする；

手順 (1-2)　：$D\_FRA(d)$ と $P\_FRA(p)$ とのフレーム番号が異なるので，

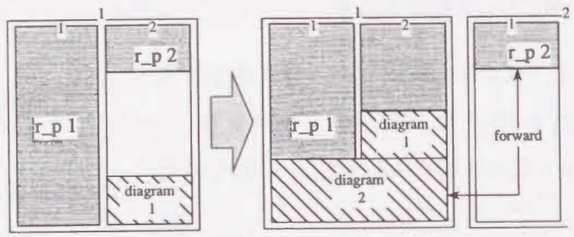
$D\_POS(d) \leftarrow (f+1, TOP, *)$ とする；

手順 (1-3)　：$F\_AVAIL(f+1)$=50 から $D\_LINE(d)$=20 を差し引き，

$F\_AVAIL(f+1) \leftarrow 30$ とする；



(a) ページを越えた図表の再配置例



(b) ページを越えた文章領域の再配置例

図 5.10　ページを越えた再配置処理の説明図

(B-2) 両幅図表の後方処理

次に，両幅図表を後方処理により配置位置を決定するアルゴリズムを示すが，まず最初に，両幅図表を配置する際に生じる制約項目を述べる．

両幅図表処理では，ページ内に両幅図表分（$D\_LINE(d) \times 2$）の未使用行数が残っていたとしても，ページ内の両フレームにそれぞれ $D\_LINE(d)$ の未使用行数がなければ配置できない．従って，ページ内に既配置の図表や文章の再配置処理が必要となるが，2つの矛盾が生じる．

まず，図 5.10-(a) のように既配置の図表を次ページに移動すれば，配置対象の図表が既配置の図表より前に配置されることになり，**順序基準**に反する．また，図 5.10-(b) に示すように文章を次ページに移動すれば，配置対象の図表は対応する参照箇所より以前に配置され，**後方基準**に反する．即ち，後方への再配置はページを越えてはいけないという一つの制約が生じる．

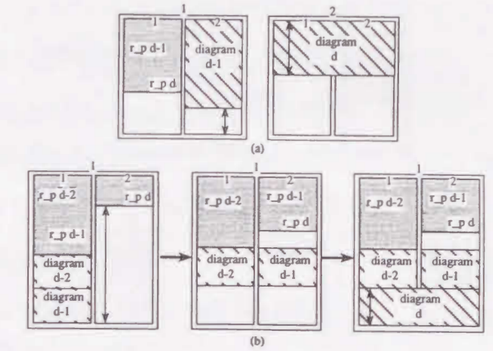


図 5.11　配置パターン A の説明図

次に，本手法は基準を満たしながら逐次的に配置決定するので，既配置の図表や文章を前方に再配置すれば，今まで満たしていた基準が破棄される．即ち，前方への再配置はできないという制約も生じる．

これらの制約から，ページの最終フレームに含まれる図表や文章は再配置できないことになり，段組み文書では，再配置可能な図表や文章はページの先頭フレームのものに限られる．

以上より，両幅図表の配置処理はページの最終フレームに図表や文章があるか否かによって二つに分類できる．

まず，図 5.11 のようにページの最終フレームに既配置の図表・文章がある場合（パターン A とする）は，まず最初に最終フレームの未使用行数を調べる．そして，その行数が図表行数より少なければ，最終フレームの図表・文章は再配置できないため，図表 d を次のページに配置する（図 5.11-(a)）．逆に多ければ，既配置の図表・文章を順序・後方基準を守りながら最終フレームに移動（この処理を再配置処理と呼び，処理内容の詳細は後で示す）して図表の領域を確保する（図 5.11-(b)）．

次に，図 5.12 のように，ページの最終フレームが空フレームの場合（パターン B とする）は，まず先頭フレームの未使用行数を調べ，その行数が図表行数以上ならば，最終フレームが空フレームであるため，図表 d をこのページに配置し（図 5.12-(a)），少なけれ

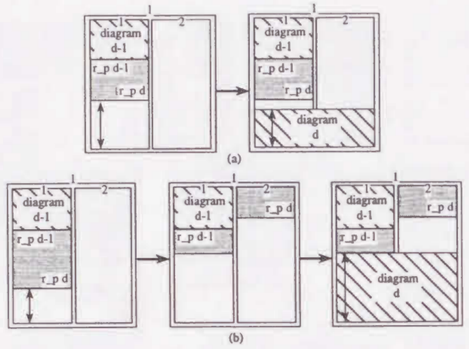


図 5.12　配置パターン B の説明図

ば再配置処理を行う（図 5.12-(b) ）.

以下に，両幅図表の処理手順を示す．尚，以下のような変数を用いる．

$P\_PAGE(p)$　：処理中の段落 p を配置するページ番号；

$F\_PAGE(P\_FRA(p))$ で与えられる；

$D\_PAGE(d)$　：処理中の両幅図表 d を配置するページ番号；

$F\_PAGE(D\_FRA(d))$ で与えられる；

【両幅図表の後方処理アルゴリズム】

手順 (2-1)　：{$D\_PAGE(d)$ の初期設定}

$D\_PAGE(d) \leftarrow D\_PAGE(d-1)$ と設定する；

手順 (2-2)　：{配置パターンの分類}

$D\_PAGE(d)$ の最終フレームに既配置の図表・文章があれば手順 (2-3) に，

既配置の図表・文章がなければ手順 (2-4) に進む；

手順 (2-3)　：{パターン A の配置処理}

$D\_PAGE(d)$ の最終フレームの未使用行数が $D\_LINE(d)$ 以上ならば手順 (2-5) に，

$D\_LINE(d)$ より少なければ手順 (2-7) に進む；

手順 (2-4)　：{パターン B の配置処理}

$D\_PAGE(d)$ の先頭フレームの未使用行数が $D\_LINE(d)$ 以上ならば手順 (2-8) に，

$D\_LINE(d)$ より少なければ手順 (2-5) に進む；

手順 (2-5)　：{$D\_PAGE(d)$ 内の未使用行数の検証}

$D\_PAGE(d)$ の未使用行数が $D\_LINE(d) \times 2$ 以上ならば手順 (2-6) に，

$D\_LINE(d) \times 2$ より少なければ手順 (2-7) に進む；

手順 (2-6)　：{再配置処理の実行}

再配置処理を行い，成功すれば手順 (2-8) に，

失敗すれば手順 (2-7) に進む；

手順 (2-7)　：{次のページを $D\_PAGE(d)$ とする}

$D\_PAGE(d) \leftarrow D\_PAGE(d)+1$ とし，手順 (2-2) に戻る；

手順 (2-8)　：{$D\_POS(d)$ の値を設定}

$P\_PAGE(p)$ と $D\_PAGE(d)$ が同ページならば，

$D\_POS(d) \leftarrow (D\_PAGE(d) \times 2, BOTTOM, *)$ とし，

異ページならば，

$D\_POS(d) \leftarrow (D\_PAGE(d) \times 2, TOP, *)$ とする；

両幅図表を配置するための空き領域を確保するため，再配置処理は順序基準・後方基準を満たしながら，配置済みの図表・文章領域をページ内で再配置する処理である．

まず，**順序基準**を守るために，移動すべき図表や文章領域（移動領域と呼ぶ）を前方向に拡張する．

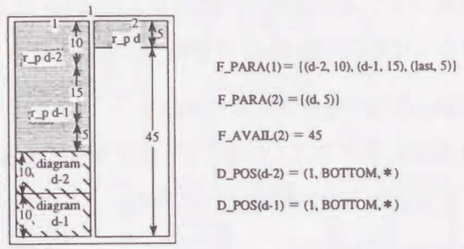
また，**後方基準**を守るために，

(1)　参照箇所手前までの文章；

(2)　参照箇所に対応する図表；

の順番で移動領域を拡張する．

例えば，フレーム f に対する再配置処理では，$F\_PARA(f)$ の最後の組 (x, x_line) を，併合規則を適用しながら $F\_PARA(f+1)$ の先頭に移動し，次式を行うことで (1) は実現できる．



(a)

図 5.13　両幅図表配置例 A

$$F\_AVAIL(f) \leftarrow F\_AVAIL(f) + x\_line\,;$$
$$F\_AVAIL(f+1) \leftarrow F\_AVAIL(f+1) - x\_line\,;$$

また，$D\_POS(x)$ の第 1 要素を f から $f+1$ に変更し，次式を行えば (2) が実現する．

$$F\_AVAIL(f) \leftarrow F\_AVAIL(f) + D\_LINE(x)\,;$$
$$F\_AVAIL(f+1) \leftarrow F\_AVAIL(f+1) - D\_LINE(x)\,;$$

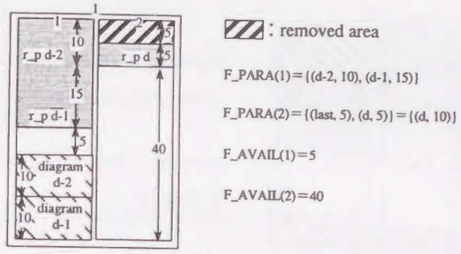
$F\_AVAIL(f) \geq D\_LINE(d)$ になるまで移動領域を拡張した後，$F\_AVAIL(f) - D\_LINE(d)$ で得られる行数分の文章を $f+1$ から f に戻す．尚，この行数分の文章が移動領域内にあれば成功とし，なければ失敗とする．

次に，上記のアルゴリズムの流れを例（図 5.13〜図 5.17 ）に従って説明する．図 5.13 の状態に $D\_LINE(d)$=20 の両幅図表 d を配置する場合，以下のような手順が行われる．

手順 (2-1)　：まず，$D\_PAGE(d) \leftarrow D\_PAGE(d-1) = 1$ と設定する；

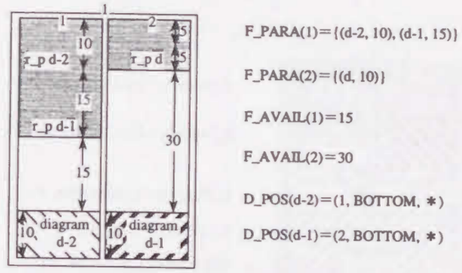
手順 (2-2)　：$D\_PAGE(d) = 1$ の最終フレーム 2 に既配置の文章があるので，パターン A の配置処理を行うため，手順 (2-3) に進む；

手順 (2-3)　：$F\_AVAIL(2) = 45$ が $D\_LINE(d) = 20$ より大きいので，ページ 1 内の未使用行数を調べるため，手順 (2-5) に進む；



(b)

図 5.14　両幅図表配置例 B



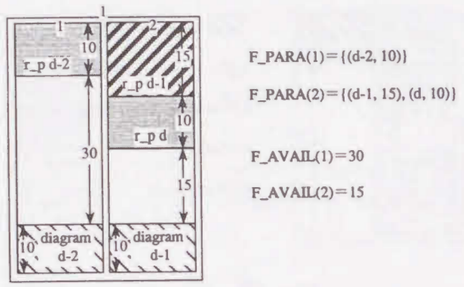
(c)

図 5.15　両幅図表配置例 C

手順 (2-5)　：$F\_AVAIL(1) + F\_AVAIL(2) = 45$ が $D\_LINE(d) \times 2 = 40$ より大きいので，ページ 1 内の既配置図表・文章の再配置処理を行うため，手順 (2-6) に進む；

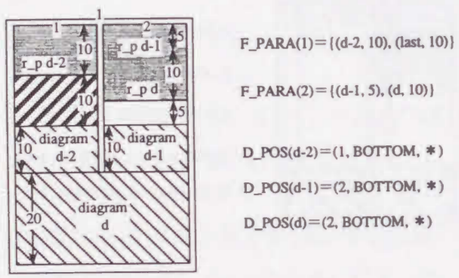
手順 (2-6)　：再配置処理を行う；

【再配置処理】



(d)

図 5.16　両幅図表配置例 D



(e)

図 5.17　両幅図表配置例 E

手順 (2-6-1)　：まず，図 5.14 に示すように，$F\_PARA(1)$ の最後の組 ($last$,5) を $F\_PARA(2)$ に移動する；

手順 (2-6-2)　：次に，図 5.15 のように，(d-1,15) が $F\_PARA(1)$ の新たな最後の組なので，図 d-1 をフレーム 2 に移動する；

手順 (2-6-3)　：その後，最後の組 (d-1,15) を $F\_PARA(2)$ に移動すると，図 5.16 のように，フレーム 1 に $D\_LINE(d) = 20$ 以上の空き領域が確保できる；

手順 (2-6-4)　：最後に，$F\_AVAIL(1) = 30$ から $D\_LINE(d) = 20$ を引いた 10 行分の文章領域をフレーム 1 に戻す（図 5.17 ）；

（再配置処理終了）

手順 (2-8)　：$P\_PAGE(p)$ と $D\_PAGE(d)$ が同じページ 1 を示しているので，図 5.17 のように，$D\_POS(d) \leftarrow$ (2,$BOTTOM$, *) とする；

(C) 前方処理

多くの図表が集中した箇所で参照された場合，各図表が小さければ，それらの図表を適切な位置に後方処理によって配置できる．しかしながら，多くの大規模な図表が集中した箇所で参照された場合，それらを後方処理のみで配置すると，それらの図表の多くが文書の後方に追いやられ，その結果，図表と参照箇所との位置が大きく離れる．この状況を防ぐために，未処理の文章行数と未処理の図表行数の割合が閾値を越えた場合，配置処理を後方処理から前方処理へ切り替える．例えば，この閾値を1に設定すると，未処理の図表行数が未処理の文章行数より多くなった場合に，後方処理の代わりに前方処理が呼び出されることになる．

前方処理では，参照箇所が出現した際に（この時点では図表の残量≥文章の残量），その参照箇所以前の文章領域を文章の残量に加えながら，図表の残量≤文章の残量となる部分まで遡り，遡った文章領域も空きスペースとして利用する．そして，その空きスペースに図表を配置した後に，先程遡った文章領域を再配置する．このようにすることで参照箇所の前方に図表を配置し，文書の後半に図表が追いやられることを防ぐことができる．

また，ページ内の再配置に関しては，片カラム図表を配置する際には不要だが，両カラム図表を配置する際には片カラム図表の移動に伴うページ内の再配置処理が必要になる．この再配置処理は先に述べた再配置処理と同じ処理が呼び出され，ページ内の片幅図表や文章領域を再配置する．

ここで注目すべき点は，文章領域を遡ることによって後方処理を行った参照箇所までもが空き領域に利用されてしまい，その結果，現在までに満たされていた**後方基準**が破棄されてしまうのではないかという疑問が起きるかもしれない．しかしながら，後方処理が行われた参照箇所では図表の残量≤文章の残量が成立していたから後方処理が行われたのであり，この参照箇所に至るまでに図表の残量≤文章の残量という条件式は満たされる．即ち，後方処理が行われた参照箇所まで文章領域が遡られることはあり得ない．

以上の処理を図 5.18 に従って説明する．図 5.18 内の図 d+2〜d+5 のように大規模な図表が集中して参照された場合，後方処理だけでは図 5.18-(a) のように図表が後方に追いやられてしまう．特に，図 d+4，d+5 等は2ページも後方に配置され，参照箇所との位置が大きく懸け離れてしまう．更に，もし，図 d+5 の参照箇所（$r\_p\ d$+5）が文章領域の最後だとすれば，図 d+3〜d+5 は文書領域外に配置されることになる．

一方，前方処理を採用した場合，図 d+2 の参照箇所（$r\_p\ d$+2）の位置で図表の残量≥

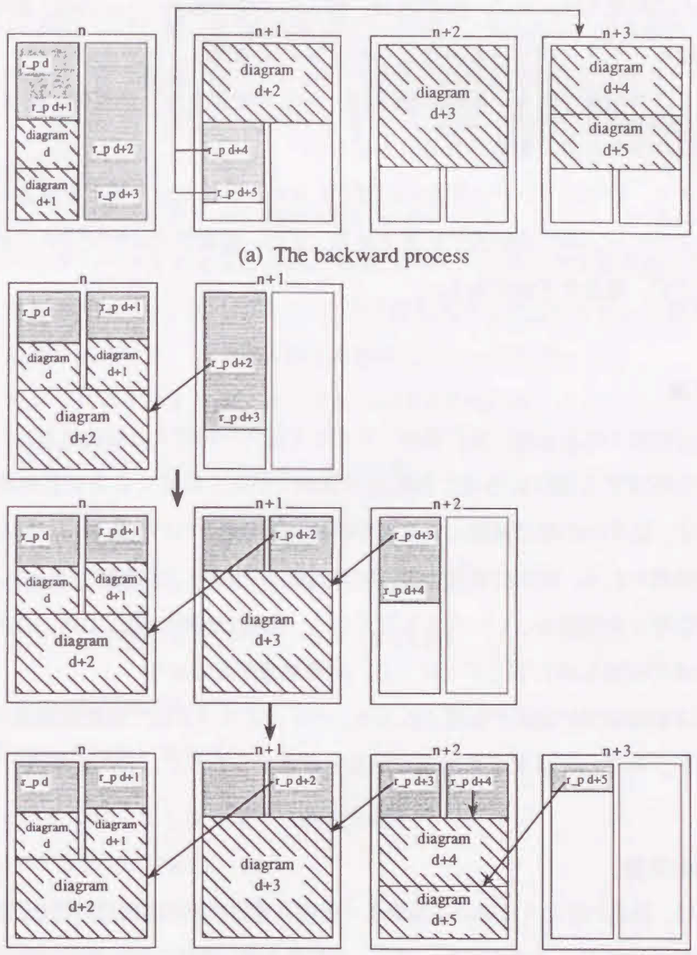


図 5.18　前方処理の配置例

文章の残量だと仮定すると，ここで前方処理が呼び出され，図 d+1 の参照箇所（$r\_p\ d$+1）から図 d+2 の参照箇所（$r\_p\ d$+2）までの領域も空きスペースとみなし図 d+2 をページ n に配置する．尚，ここで片幅図 d+2 のフレーム間移動が再配置処理により行われている．

その後，図 d+2 の参照箇所（r_p d+2）までの文章領域を配置すると r_p d+2 はページ n+1 の第1カラムに配置され，次に，図 d+3 は，図 d+3 の参照箇所（r_p d+3）から r_p d+2 までの文章領域も空きスペースとみなしページ n+1 に配置し，r_p d+3 はページ n+2 の第1フレームに再配置される．同様に図 d+4，d+5 も配置され，最終的に図 5.18-(b) のように図表はバランス良く配置されることになる．

本手法により，参照箇所以前に図表が配置されるが，各図表は均等に配置され，参照箇所の前後一定のページ内に各図表が配置される．また，提案手法は配置済みの図表をバックトラックせず，高速な手法である．

(D) 制約処理

図表配置が制約される領域（制約領域）を避けて図表を配置する制約処理は，制約領域以降に図表を配置する前制約処理と制約処理以前に図表を配置する後制約処理に分類される．例えば，始章内が制約領域になる場合は前制約処理を行い，終章内が制約領域にならば後制約処理を行う．前制約処理では，制約領域内で出現した図表に対して，制約領域が終了した位置を参照箇所として後方処理を行う．後制約処理に関しては，全段落の行数から制約領域の行数を差し引くことにより，均等処理が行われる．

本来ならば制約領域に図表が配置されると，バックトラックして既配置図表の再配置が必要であるが，本手法では文章や図表の残量を考慮することで，これを解決している．

(E) その他の処理

関連処理は，話題の区切りに当たる見出し（区切り見出し）が出現した場合，$P\_FRA(p)$ の下部に既配置の図表を上部に配置し直す．章節から章，項から章・節へ変化すると話題が区切られるので，章見出し，及び項見出しの後に出現した節見出しを区切り見出とする．

また，以上までの処理では，安定基準は満たしているが，ページの上下双方に図表が配置されるため，文章は図表間に割付けられる．そこで，図表と文章領域の境を一つにするため，最初に配置された図表の上下位置を優先し，上下どちらか一方に図表をまとめて配置する固め処理を行う．

更に，図表フレーム下部に脚注があれば，図表を下部から上部に配置し直す脚注処理や最終行の見出しを次フレーム先頭に配置する見出し処理も取り入れる．

5.4.5　詳細位置決定処理の概要

各図表の D_POS には，第1要素には配置すべきフレームの番号，第2要素にはフレーム内での上下位置，第3要素にはフレームの上端から図表までの行数が示される．詳細位置決定部では，概略位置（第1，2要素）を用いて，**順序**，**安定**，及び**優先基準**を考慮しながら詳細位置（第3要素）を求める．

本処理はページ上部に配置すべき図表（第2要素が TOP の図表で上配置図表と呼ぶ）の後，下部に配置すべき図表（第2要素が $BOTTOM$ の図表で下配置図表と呼ぶ）に対して行う．両幅図表をページ端に配置するという優先基準を満たすため，上配置図表では両幅図表を最初に処理する．この配置順序の制御にはキューを用いる．

以下にアルゴリズムを示す．尚，以下のような変数を用いる．

D_SET　：対象ページに配置すべき図表番号の集合；

Q_SET　：キューに格納すべき図表番号の集合；

L_Pos　：各図表を配置するための左フレームの基準点；

R_Pos　：各図表を配置するための右フレームの基準点；

但し，各基準点の初期値は0とする．

【上配置図表のレイアウトアルゴリズム】

手順1　：{配置対象とする図表番号の決定処理}

$D\_SET = \emptyset$ ならば手順4に進み，

そうでなければ D_SET から図表番号を1個取り出し，

その図表の番号を d とする；

手順2　：{図表幅による分類処理}

$D\_WIDTH(d) = SINGLE$ ならば d を Q_SET に入力した後，

手順1に戻り，

$D\_WIDTH(d) = DOUBLE$ ならば手順3に進む；

手順3　：{両幅図表の詳細位置決定処理}

$D\_POS(d)$ の第3要素に L_Pos の示す値を代入し，

各基準点を以下のように変更した後，手順1に戻る；

表 5.2　図表データの例

d	$D\_LINE(d)$	$D\_WIDTH(d)$	$D\_POS(d)$
$d1$	5	$SINGLE$	$(1, TOP, *)$
$d2$	10	$DOUBLE$	$(1, TOP, *)$
$d3$	5	$SINGLE$	$(1, TOP, *)$
$d4$	10	$DOUBLE$	$(2, BOTTOM, *)$
$d5$	5	$DOUBLE$	$(2, BOTTOM, *)$
$d6$	5	$SINGLE$	$(2, BOTTOM, *)$

$L\_Pos \leftarrow L\_Pos + D\_LINE(d)$；

$R\_Pos \leftarrow R\_Pos + D\_LINE(d)$；

手順4　：{配置対象とする片幅図表番号の決定処理}

Q_SET から図表番号を1個取り出し，その図表の番号を d とする；

手順5　：{片幅図表の詳細位置決定処理}

$D\_POS(d)$ の第1要素がページ内の先頭フレーム番号ならば，

$D\_POS(d)$ の第3要素に L_Pos の示す値を代入し，次式を行う；

$L\_Pos \leftarrow L\_Pos + D\_LINE(d)$；

また，$D\_POS(d)$ の第1要素がページ内の最終フレーム番号ならば，

$D\_POS(d)$ の第3要素に R_Pos の示す値を代入し，次式を行う；

$R\_Pos \leftarrow R\_Pos + D\_LINE(d)$；

手順6　：{終了判定処理}

$Q\_SET \neq \emptyset$ ならば手順4に戻り，

$Q\_SET = \emptyset$ ならば処理を終了する；

次に，F_PARA で示される行数分だけ基準点を下げる．下配置図表に対しても，優先基準を満たすため，両幅図表をページ端に配置する．よって，上配置図表の際は片幅図表をキューに蓄積したが，下配置図表の場合は両幅図表をキューに入力する．

例として，表5.2に示す図表データを用いた処理の流れを図5.19～図5.23に従って説明する．

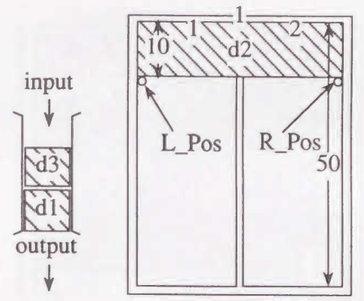


図 5.19　詳細位置決定処理の例 A

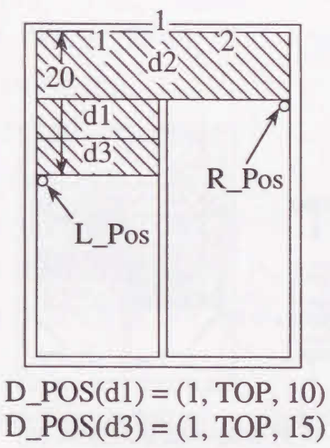


図 5.20　詳細位置決定処理の例 B

手順1　：まず，図表 $d1$,$d2$,$d3$ の D_POS の第2要素が TOP なので，

D_SET=$\{d1, d2, d3\}$ とする；

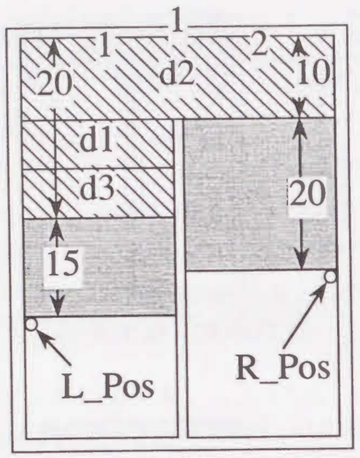
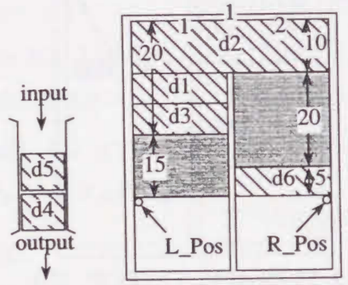


図 5.21　詳細位置決定処理の例 C

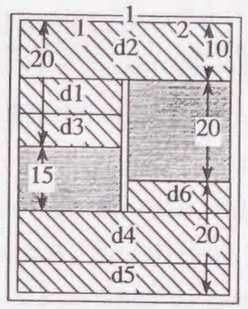


D_POS(d6) = (2, BOTTOM, 30)

図 5.22　詳細位置決定処理の例 D

手順 2　：$D\_WIDTH(d1)$=$D\_WIDTH(d3)$ = $SINGLE$ なので，

Q_SET=$\{d1, d3\}$ とする；



D_POS(d4) = (2, BOTTOM, 35)

D_POS(d5) = (2, BOTTOM, 45)

図 5.23　詳細位置決定処理の例 E

手順 3　：$D\_WIDTH(d2)$=$DOUBLE$ なので，

両幅図表 $d2$ を L_Pos=0 に配置し，

左右の基準点を図 5.19 のように移動する；

$D\_POS(d2) \leftarrow (1, TOP, L\_Pos)$ より，

$D\_POS(d2) = (1, TOP, 0)$；

$L\_Pos \leftarrow 0 + D\_LINE(d2)$ より，L_Pos=10；

$R\_Pos \leftarrow 0 + D\_LINE(d2)$ より，R_Pos=10；

手順 4　：Q_SET から図表番号 $d1$, $d3$ を順番に取り出す；

手順 5，6　：$D\_POS(d1)$，$D\_POS(d3)$ の第 1 要素がフレーム番号 1，つまり，先頭フレームなので，図 5.20 のように，それぞれの図表を L_Pos=10 から配置し，L_Pos を図 5.20 のように移動する；

$D\_POS(d1) \leftarrow (1, TOP, L\_Pos)$ より，

$D\_POS(d1) = (1, TOP, 10)$；

$L\_Pos \leftarrow 10 + D\_LINE(d1)$ より，L_Pos=15；

$D\_POS(d3) \leftarrow (1, TOP, L\_Pos)$ より，

$D\_POS(d3) = (1, TOP, 15)$；

$L\_Pos \leftarrow 15 + D\_LINE(d3)$ より，$L\_Pos = 20$；

次に，フレーム1に15行の文章，フレーム2に20行の文章を配置すると，L_Pos=15+15=35，R_Pos=10+20=30となる（図 5.21 ）．

更に，D_POSの第2要素が$BOTTOM$の下配置図表(d4, d5, d6)に関しては，$D\_WIDTH(d6) = SINGLE$，$D_POS(d6)$の第1要素がフレーム2（最終フレーム）なので，図 5.22 のように図表$d6$をR_Posから配置し，図表$d4$，図表$d5$はキューに入力される．その後，キューの出力順に各図表を配置し，最終的に図 5.23 の完成文書を作成する．

5.5　図表自動配置手法の評価

本節では，提案手法の有効性を確認するため，まず，５．４．２で定義した配置基準の正当性を示した後，本手法で配置された図表がどの程度その配置基準を満たしているかを示す．更に，本手法による配置結果と実際に出版されている文書内の配置結果とを比較検討する．

5.5.1　基準の評価

提案基準が実際の文書にどの程度適合するかを評価する（評価Ａ）．対象文書は，論文５０編（約４４０ページ，図表５５６個，片幅図表４４６個，両幅図表１１０個）であり，制約領域は始章内及び終章以降の領域とした．調査結果を表 5.3 に示す．

後方基準に不適合な図表は，大半が均等及び制約基準を満たすために参照箇所の前方に配置されたものである．即ち，後方基準と均等基準は相反するものであり，調査対象の論文誌内では，図表と参照箇所との距離を均等に保つことが優先されているため，後方基準の適合率（adaptation rate）が低下している．また，均等基準に関しては，サイズがほぼ1ページの図表が連続して参照されたため，６個の図表が均等基準を満たさなかった．制約基準の対象となる図表（１４個）は，すべて制約領域外に配置されていた．表 5.3 から全基準の平均的な適合率は94.7%となり，導入した基準は妥当であると言える．

表 5.3　評価Ａによる結果

基準	適合率 (%)	不適合な図表数
順序基準	99.3	2
安定基準	100	0
後方基準	70.1	81
均等基準	98.9	6
制約基準	100	0
優先基準	100	0

表 5.4　評価Bによる結果

基準	適合率 (%)	不適合な図表数
順序基準	100	0
安定基準	100	0
後方基準	100	0
均等基準	98.9	3
制約基準	100	0
優先基準	100	0

5.5.2　アルゴリズムの評価

本システムの結果と各基準の適合度を示すことによって，本アルゴリズムを評価する（評価B）．尚，評価Aと同じ文書データから図表を抜いた源文書を評価データとした．また，後方基準と均等基準は相反するため，均等処理等を行った図表は後方基準の対象外とした．実験結果を表 5.4 に示す．

後方基準に対して均等処理等を対象外とすると，全ての図表が適合した．均等処理が行われた図表は１４６個あり，もし均等処理をしなければ均等基準に反する図表は６４個であった．この結果，均等処理をしなければ適合率は88%に低下し，均等処理の有効性がわかる．表 5.4 より全基準の平均適合率は 99.8%となり，本アルゴリズムの有効性が確認できた．

5.5.3　アルゴリズムの評価

本手法の有効性を確認するため，本手法により配置された図表位置と従来の手法により配置された図表位置とを比較する．そして，双方の手法による配置結果が出版済みの論文誌上のレイアウトとどの程度の適合しているか調査した（評価C）．

ここで，従来の文書整形／編集システムでは，単純に空き領域内に図表を自動配置することはできるが，複数の図表が既に配置されているページにおいて配置済みの図表を再配置しながら空き領域を確保し，図表を配置するようなより完全なレイアウトを行うことはできない．また，文書全体のバランスを考慮した図表配置を行うシステムも存在しない．

表 5.5　評価Cによる結果

ランク	従来の手法	本手法
E	377	454
(Excellent)	(67.8 %)	(81.7 %)
G	57	68
(Good)	(10.2 %)	(12.2 %)
F	50	24
(Fair)	(9.0 %)	(4.3 %)
P	72	10
(Poor)	(13.0 %)	(1.8 %)

一方，本手法は再配置処理を採用することにより，基準を満たした適切な位置に片幅図表を再移動した後，両幅図表を配置することを可能にした．また，前方処理により，文書全体の配置バランスを考慮した図表配置も可能にした．これらは，本手法の特徴であるため，これらの特徴を除いたものを従来の手法（common method）として用いる．これらの特徴を除いた手法，即ち，５．４．４の(B-1)で述べた片幅図表配置，及び図表のフレーム間移動を考慮しない両幅図表配置は従来のシステムでも実現されている機能である．

評価に用いるデータは評価A，Bと同じデータを用い，それらのデータから作成した段落，図表データを本手法および従来の手法の入力データとした．また，双方の出力結果を編集のプロ達が人手で行った論文誌における配置位置を基準にして，以下のランクに分類した．分類結果を表 5.5 に示す．

E (Excellent)　：論文誌で配置された位置と同じ位置に配置された図表；

G (Good)　：論文誌で配置された位置と同じフレーム内に配置された図表，但し，Eランクの図表を除く；

F (Fair)　：論文誌で配置された位置と同じページ内に配置された図表，但し，E，Gランクの図表を除く；

P (Poor)　：論文誌で配置された位置と異なるページに配置された図表；

分類結果の表 5.5 から，本手法を用いることにより，Eランクの図表が増加し，Pランクの図表が減少している．これは，従来の手法が各ページ単位の最適化のみを行い，グ

ローバルな最適化は行わないのに対し，本手法では均等処理を行っているためである．また，ページ内の最適化に関しても，本手法では参照位置を考慮した再配置処理を行うため，F・Pランクの図表数を抑えることもできる．これは，図表幅の異なる複数の図表を同一ページに配置する場合に，本手法が有効であることを示している．このように，本手法により配置された図表の82%が論文誌と同じ位置に，94%の図表が同フレーム内に，98%の図表が同ページ内に配置され，本手法が精度の高い図表配置を行えることが分かる．

更に，処理時間に関しては，文書構造抽出の計算量は文書の段落数pに比例し$O(p)$となる．また，図表配置の計算量は文書の段落数と図表数dに依存し，バックトラックは起こらないため，$O(p+d)$となる．また，制約領域や図表が文書後半に固まった場合，前方処理を採用せずにバックトラックを行ない，ページを越えた図の再配置を行う際の最悪のケースは，図表を配置する毎にそれ以前にある配置済み図表を全て再配置する場合になる．この場合の最悪時間計算量は，$O(p+\sum d)$，即ち$O(p+d^2)$になり，バックトラックしない場合の$O(p+d)$に比べ，時間効率が低下する．

また，実験システムは，計算機NEC $PC-98AP2$上に言語Cで開発され，評価データ中で段落と図表数が最も多い文書A（段落数=１１２個，図表数=２３個の文書データ）に対する実験的評価を行った結果，本手法（前方処理を採用し，バックトラックを行わない手法）では0.7秒で図表配置位置が決定した．勿論，制約領域などは避けて配置された．しかし，前方処理を用いず後方処理のみの場合には，実験データ内の図表数が多かったために８個の図表が文書後半に追いやられ本文中に配置できなかった．そこで，文書後半に固まっている５個の図表を未配置の図表８個を配置する度にバックトラックし，その後文書中盤の配置済み図表も含めてバックトラックした結果，結局７２個の図表が再配置を起こし，全体の図表配置に3.2秒の時間を要した．また，各種データの記憶量も$O(p+d)$であり，文書Aで約３キロバイトのコンパクトな容量で稼働できた．

本論文内では，文書スタイルが２段組みの場合における図表配置処理を提案しているが，文書スタイルが３段組み以上の場合の本手法の拡張性を考察する．まず，１段幅（片幅）図表，２段幅（両幅）図表に関しては，ページ内の２フレーム毎を配置対象領域とすることにより本手法が適用化のである．しかしながら，３段幅以上の図表に関しては，ページの最終フレームに配置済みの図表・文章領域を再配置できないという制約については同じであるが，各フレーム内にD_LINE以上の空き領域を確保するための再配置処理アルゴリズムに改良が必要である．

また，本手法は以下のような仮定の下にバックトラックを行わないアルゴリズムである．

(1)　対象とする図表のサイズは片幅・両幅の図表；

(2)　文書内の行間は一定値に固定；

それ故に，中途半端な幅の図表や，非常に複雑なレイアウトを対象にした場合はある程度のバックトラックは起こり得ると思われる．

5.6 結　言

本章前半部では，片仮名語の表記の揺れが規則的なことに着目し，片仮名表記に関する変換ルールを用いたより正確な異表記生成手法，及びこれらの規則適用が容易な正規表記を定義し，この規則と正規表記を用いた片仮名辞書の圧縮手法を提案した．

また，7,500語の片仮名データに対する実験結果から，本手法による片仮名辞書の圧縮率と異表記生成精度の有効性を実証した．

本手法は，片仮名異表記を生成する際に，数回の辞書アクセスを必要とするが，辞書アクセスのための索引をコンパクトな先行順ビット列により構築しているため，索引表全体を主記憶上に格納できる．また，本辞書検索手法は，未登録語を検索する際に２次記憶へのアクセスを必要としない場合も多々発生するため，高速な片仮名異表記生成処理が可能である．

しかしながら，本システムにおいては，片仮名語の意味も考慮した異表記生成手法については実現に至っていないので，今後の課題として研究を進める計画である．

また，本章後半部では，図表配置に対する参照箇所の位置，配置の均等性等に対する基準を考慮し，対象文書から抽出した文書構造をコンパクトなデータ構造を持つ２進木構造に格納し，バックトラックなしに高速に図表配置を決定する手法を提案した．また，５０編の論文データに対する実験結果から，本手法による図表配置処理の有効性を実証した．

本図表配置手法は以下のような特徴をもつ．

(1)　ページ内の再配置処理を行うため，図表が参照箇所の位置も考慮してフレーム間を移動することが可能．これにより，２段組みの文書内に幅の異なる複数の図表が配置されるような場合にでも，より完全なレイアウトが実現できる．

(2)　文章領域に比べ図表領域の残量が多い場合に，図表を関連する段落より先に配置（参照箇所の前方に図表を配置）する前方処理を行うため，文書全体を通してバランスの良いレイアウトが可能．この機能により，集中した箇所で複数の大きな図表が参照されたり，文書の後方で多くの図表が参照された場合にでも，図表が後方に追いやられて文書の後方に図表が固まるといった偏ったレイアウトが避けられる．

(3)　文書構造を抽出した後にレイアウトを行うため，図表を配置すべきでない領域を避けて配置する制約処理が可能である．

但し，上記の2，3番目の項目は，文書整形システムのようにバッチ的に文書のレイアウトを行う場合に有効な特徴である．

本研究の今後の課題としては，片幅と両幅に属さない図表や他の段組みに対する処理の考案が計画されている．また，他種の文書についても本論文で提案した基準とその基準に基づく自動配置手法の有効性を確かめることも今後の課題である．

第6章

結　論

以上，本論文では2進ディジタル探索木を用いた辞書検索法とその応用に関する研究として，現在までに考案されている各種キー検索技法を紹介すると共に，各技法の問題点について論議した．そして，特に，従来の動的ハッシュ法では困難であった順検索を可能にする手法として，2進ディジタル探索木をハッシュ表に用いた拡張ハッシュ法（トライハッシュ法）を紹介した．

そして，3章では，ハッシュ表となる2進ディジタル探索木を主記憶上に格納するために，先行順ビット列と呼ばれるコンパクトなデータ構造に圧縮する手法をJongeらの提案に従って紹介した．

その後，Jongeらの手法が大規模なキー集合に対しては，時間的および空間的な問題点が発生することを明言し，それらの問題点に対する解決手法を4章において提案した．大規模なキー集合に対しては，先行順ビット列が非常に長くなり，各種処理時間効率が低下するという問題点を解決するため，階層的な木構造を持つ階層2進ディジタル探索木を提案し，この構造に対する先行順ビット列への圧縮方法，および検索，追加アルゴリズムを示した．また，大規模なキー集合に対しては，ダミーリーフ数が多くなることから，必要以上にビット列が長くなるという問題点に対して，2進ディジタル探索木をパトリシア構造に拡張し，ダミーリーフを用いずに新しい先行順ビット列に圧縮する手法を提案した．そして，各種の大規模なキー集合に対して実験を行い，本手法の有効性を実証した．

更に，5章では，非常にコンパクトな索引部で高速な順検索が行えるという本手法の特徴を応用するため，本手法により構築された片仮名辞書，および独自の片仮名異表記変換

ルールを用いた片仮名異表記生成手法を提案し，高速に片仮名異表記が生成できることを実験により実証した．また，文書内から抽出した文書構造を本手法により構築し，高速に文書内に図表を自動配置する手法を紹介した．

本研究テーマに関する今後の課題は，バランスの悪い２進木構造に対する効率的なビット列への圧縮方法の提案が挙げられる．木構造の対称性が悪い場合，本手法では先行順ビット列の長さの不均一は避けられず，時間効率が低下する．そこで，今後は対称性が悪い木構造に対してもビット長を均等に保つ階層化の手法を考案する必要がある．また，本手法は２進木構造に対してのみ有効な手法であるので，今後は，自然言語辞書に頻繁に用いられるトライ構造に対して適用可能な圧縮手法に関して研究を行う予定である．

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 大恵俊一郎教授，矢野米雄教授に深く感謝する．

本研究にあたって，情報工学一般及び各専門分野に関して御指導，御教示を賜った徳島大学工学部知能情報工学教室 高橋義造教授，小野典彦助教授に感謝する．

住友金属工業株式会社の津田和彦博士，群山大学工学部計算機科学科の朴棋鴻副教授，株式会社リコーの森本勝士博士，ＡＴＲ音声翻訳通信研究所の美馬秀樹博士，住友金属工業株式会社の入口浩一博士，徳島大学工学部知能情報工学教室の富士正人技官，井上富夫技官，同教室の青江研究室の大学院生 有田健氏，林淑隆，弘田正雄氏，望月久稔氏，Kirschning Ingrid 氏，安藤一秋氏，小山雅史氏，溝渕昭二氏，森田和宏氏はじめ，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] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley, Reading Mass., Chs.4, 5 and 11 (1983). 大野義夫訳，倍風館 (1987).

[2] A. V. Aho, R. Sethi and J. D. Ullman, "Compilers-Principles, Techniques and Tools," Addison-Wesley, Reading Mass., Ch.2-3 (1986).

[3] J. Aoe, "An Efficient Implementation of Static String Pattern Matching Machines," IEEE Trans. Softw. Eng., **Vol.SE-15**, No.8, pp.1010–1016 (1989).

[4] J. Aoe, "Computer Algorithms–Key Search Strategies," IEEE Computer Society Press. (1991).

[5] 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.9，pp.1592–1600 (1987).

[6] 青江順一，"静的ハッシュ法とその応用"，情報処理学会誌，**Vol.33**，No.11，pp.1359–1366 (1992).

[7] 青江順一，"動的ハッシュ法とその応用"，情報処理学会誌，**Vol.33**，No. 12，pp.1465–1472 (1992).

[8] 青江順一，"仮名漢字変換にはどのようなアルゴリズムが使われているか"，情報処理学会誌，**Vol.35**，No.2，pp.157–158 (1994).

[9] 相川勇之，宮原浩二，高山泰博，鈴木克志，丸山冬樹，"片仮名異表記を考慮したユーザ辞書システムの拡張"，情処第 44 回全国大会，**1P-6**，pp.3-119–3-120 (1992).

[10] R. Bayer and E. McCreight, "Organization and maintenance of large order indexes," Acta Inf. **Vol.1**, pp.173–189 (1972).

[11] D. Comer, "The ubiquitous B-Tree," ACM Computering Surveys, **Vol.11**, No.2, pp.121–137 (1979). 上田訳：Bit 別冊，共立出版，pp.21–39 (1980).

[12] 土井美和子，福井美佳，山口浩司，竹林洋一，岩井勇，“文書構造抽出技法の開発”，電子情報通信学会論文誌，**Vol.J76-D-II**，No.9，pp.2042–2052 (1993).

[13] R. J. Enbody and H. C. Du, “Dynamic Hashing Schemes,” ACM Computing Surveys, **Vol.20,** No.2, pp.85–113 (1988). 遠山道夫訳：Bit 別冊，共立出版，pp.43–68 (1990).

[14] R. Fagin, J. Nievergelt, N. Pippenger and H. R. Storing, “Extendible hashing –A fast access method for dynamic files,” ACM Trans. Database Syst., **Vol.4**, No.3, pp.315–344 (1979).

[15] 福井美佳，土井美和子，竹林洋一，山口浩司，岩井勇，大黒和夫，“文書構造を用いた自動レイアウトシステム”，情処文書処理とヒューマンインタフェース研究会資料，**Vol.20**，No.3，pp.1–10 (1988).

[16] 福井美佳，山口浩司，土井美和子，岩井勇，“文書自動レイアウトシステムにおける図表配置候補生成方式”，情報処理学会論文誌，**Vol.34**，No.7，pp.1499–1506 (1993).

[17] G. H. Gonnet, “Handbook of Algorithms and Data Structures,” Addison-Wesley, Reading Mass., Ch. 3 (Searching Algorithms) pp.25–147 (1984).

[18] 林淑隆，獅々堀正幹，伊与田敦，津田和彦，青江順一，“複合語キーワードの効率的抽出法”，情処自然言語処理研究会資料，**Vol.104**，No.9，pp.63–70 (1994).

[19] 井田哲雄，“ハッシュ法”，情報処理学会誌，**Vol.24**，No.4，pp.391–395 (1983).

[20] 石畑清，“データ構造とアルゴリズム”，岩波書店，pp.73–123 (1989).

[21] 伊藤和人，“LaTeX トータルガイド”，秀和システム (1991).

[22] 伊藤伸泰，丸山宏，“OCR 入力された日本語文の誤り検出と自動訂正”，情報処理学会論文誌，**Vol.33**，No.5，pp.664–670 (1992).

[23] R. L. James, “Automatic Data Structure Selection: An Example and Overview,” Comm. ACM, **Vol.21**, No.5, pp.376–385 (1978).

[24] W. D. Jonge, A. S. Tanenbaum and R. P. Riet, “Two access methods using compact binary trees,” IEEE Trans. Softw. Eng., **Vol.SE-13**, No.7, pp.799–810 (1987).

[25] 川口久光，加藤寛次，藤沢浩道，畠山　敦，藤縄雅章，“自由語検索のための高速文字列検索方式”，情処第 39 回全国大会，**2N-8**，pp.1078–1079 (1989).

[26] 河上恭雄編集，“官報，白書，教科書の外来語，主な仮名表記のゆれ一覧”，文化庁・外来語実態調査資料 3 (1990).

[27] B. W. Kernighan, “UNIX Time-Sharing System : Document Preparation,” Bell Syst. Tech. J., **Vol.57**, No.6, pp.2115–2135 (1978).

[28] D. E. Knuth, “The Art of Computer Programming,” Addison-Wesley, Reading Mass., Ch.3 (Sorting and Seraching) pp.422–480 (1973).

[29] D. E. Knuth, “The TeXbook,” Addison-Wesley (1983).

[30] P. A. Larson, “Performance analysis of linear hashing with partial expansions,” ACM Trans. Database Syst., **Vol.7**, No.4, pp.566–587 (1982).

[31] P. A. Larson, “Linear hashing with separaters - a dynamic hashing scheme achieving one - access retrieval,” ACM Trans. Database Syst., **Vol.13**, No.13, pp.366–388 (1988).

[32] U. Manber and R. Baeza-Yates, “An algorithm for string matching with a sequence of don’t cares,” Information Processing Letters, **Vol.37**, pp.133–136 (1991).

[33] K. Mehlhorn, “Dynamic binary search tree,” SIAM J. Comput., **Vol.8**, No.2, pp.175–198 (1979).

[34] H. Mendelson, “Analysis of extensible hashing,” IEEE Trans. Softw. Eng., **Vol.SE-8**, No.6, pp.611–619 (1982).

[35] 宮崎正弘，池原悟，横尾昭男，“単語結合型辞書引きを用いた日英機械翻訳用辞書の構成”，情処自然言語処理研究会資料，**Vol.84**，No.12，pp.87–94 (1991).

[36] 溝口徹夫，“B-tree によるデータ管理”，情報処理学会誌，**Vol.21**，No.7，pp.769–776 (1980).

[37] 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，**Vol.J76-DII**，No.11，pp.2374–2384 (1993).

[38] 西原清一，“ハッシングの技法と応用”，情報処理学会誌，**Vol.21**，No.9，pp.980–991 (1980).

[39] 奥村薫，建石由佳，脇田早紀子，金子宏，“日本語校正支援システム「FLeCS」”，情処自然言語処理研究会資料，**Vol.87**，No.11，pp.83-90 (1992).

[40] 大深悦子，“原表記とカナ表記の対応判定アルゴリズム”，情処第37回全国大会，**6B-3**，pp.1010–1011 (1988).

[41] B. K. Reid, “A High-Level Approach to Computer Document Formatting,” Proc. of 7th ACM Symposium of Principles of Programming Languages, pp.24–31 (1980).

[42] 斉藤康己，“文書整形システムの現状と将来”，情報処理，**Vol.31**，No.11，pp.1543–1562 (1990).

[43] 里山元章，中川正樹，高橋延匡，“文書の論理構造を備えた日本語清書システム「浄書」の設計と実現”，情報処理学会論文誌，**Vol.30**，No.9，pp.1126–1134 (1989).

[44] 島津美和子，吉村裕美子，平川秀樹，天野真家，“カタカナ異形表記・誤記修正機能の開発・評価”，情処第44回全国大会，**3Q-4**，pp.3-249–3-250 (1992).

[45] M. Shishibori, J. Aoe, K. H. Park and H. Mochizuki, “An Automatic Selection Method of Key Search Algorithms,” IEICE Trans. on Information and Systems, **Vol.E78-D**, No.4, pp.383–393 (1995).

[46] 獅々堀正幹，青江順一，“文書校正支援システムにおける校正知識の構築法”，情処自然言語処理研究会資料，**Vol.88**，No.11，pp.79–86 (1992).

[47] 獅々堀正幹，清原聡，青江順一，“階層化による2進ディジタル探索（BDS）木の改善”，電子情報通信学会論文誌，Vol.J79-D-I，No.2，pp.79–87 (1996).

[48] 相賀徹夫，“例文で読むカタカナ語の辞典”，小学館 (1990).

[49] 墨康成，柴田昌宏，“文書校正ルールの適用率向上に関する考察”，情処第44回全国大会，**3Q-2**，pp.3-245–3-246 (1992).

[50] 鈴木恵美子，武田浩一，藤崎哲之助，“日本語文書校正支援システム CRITAC”，情処日本語文書処理研究会資料，**Vol.8**，No.5，pp.1–10 (1986).

[51] 高木伸一郎，安田恒雄，島崎勝美，松岡浩司，“日本文訂正支援システムにおける未知語訂正候補抽出方式”，情処第37回全国大会，**6B-4**，pp.1012–1013 (1988).

[52] M. Tamminen, “Order preserving extendible hashing and bucket tries,” BIT, **Vol.21**, No.4, pp.419-435 (1981).

[53] 田中穂積，“自然言語解析の基礎”，産業図書，3章，pp.143–151，5章，pp.227–231 (1989).

[54] Y. Y. Tang, C. D. Yan and C. Y. Suen, “Document processing for automatic knowledge acquisition,” IEEE Trans. Knowl. Data. Eng., **Vol.KDE-6**, No.1, pp.3–21 (1994).

[55] 内田裕士，杉山健司，“自由入力形式のカナ漢字変換”，情処自然言語処理研究会資料，**Vol.27**，No.3，pp.1–8 (1981).

[56] 山口浩司，福井美佳，岩井勇，“知的文書処理システムにおける図表の割付け方式”，情処第36回全国大会，**6U-4**，pp.1303–1304 (1988).

[57] 山中紀子，田野崎康雄，斉藤裕美，小林賢一郎，“日本語テキストリーダにおける日本語校正支援機能”，情処第44回全国大会，**3Q-1**，pp.3-243–3-245 (1992).

付 録A

片仮名異表記変換ルール一覧

【正規化ルールに属する規則】

（1）イア：=イヤ；#
（2）イアー：=イヤー；#
（3）イッチ：=ウィッチ；#
（4）ウイ：=ウィ；#
（5）ウエ：=ウェ；#
（6）ウエー：=ウェー；#
（7）ウェイ：=ウェー；#
（8）ウエイ：=ウェー；#
（9）ウオ：=ウォ；#
（10）エー：=エイ；#
（11）ガ：F=グァ；#
（12）クエ：=クウェ；=クェ；#
（13）クオ：=クォ；=クウォ；#
（14）クス：=ックス；#
（15）グア：=グァ；#
（16）ケ：NF=クェ；#
（17）サンス：=ッサンス；#
（18）ザー：E=ザ；#

（１９）シア：E=シャ；#
（２０）シアル：=シャル；#
（２１）シン：=シム；#
（２２）ジュール：=デュール；#
（２３）スイ：=スウィ；#
（２４）スエ：=スウェ；#
（２５）スムーズ：=スムース；#
（２６）セサ：E=セッサ；#
（２７）セサー：E=セッサ；#
（２８）セッサー：E=セッサ；#
（２９）ター：E=タ；#
（３０）チ：=ツィ；#
（３１）チスト：E=ティスト；#
（３２）チズム：E=ティズム；#
（３３）チック：E=ティック；#
（３４）チャ：E=チュア；#
（３５）チャネ：F=チャンネ；#
（３６）チャー：E=チュア；#
（３７）チュ：=テュ；#
（３８）チン：E=ティン；#
（３９）ティー：E=ティ；#
（４０）デイジ：F=ディジ；#
（４１）ディー：E=ディ；#
（４２）ト：=トゥ；#
（４３）バ：=ヴァ；#
（４４）バッチ：=バッジ；#
（４５）バレー：=バレエ；#
（４６）ビ：=ヴィ；#
（４７）ビュ：=ヴュ；#
（４８）ファー：=ファウ；#

（４９）フェース：=フェイス；#
（５０）ベ：=ヴェ；#
（５１）ペチ：=ペティ；#
（５２）ホン：E=フォン；#
（５３）ホーム：=フォーム；#
（５４）ボ：=ヴォ；#
（５５）ボー：=ボウ；#
（５６）マー：E=マ；#
（５７）レクレー：=レクリエー；#
（５８）レー：=レイ；#
（５９）ローヤル：=ロイヤル；#
（６０）ワイヤ：=ワイヤー；#
（６１）ユーム：E=ウム；#
（６２）ーサー：E=ーサ；#
（６３）ーター：E=ータ；#

【一般化ルールに属する規則】

（１）イヤ：=イア；#
（２）イヤー：=イアー；#
（３）ウム：E=ユーム；#
（４）ウィ：=ウイ；#
（５）ウィッチ：=イッチ；#
（６）ウェ：=ウエ；#
（７）ウェー：=ウエー；=ウェイ；=ウエイ；#
（８）ウォ：=ウオ；#
（９）エイ：=エー；#
（１０）クウェ：=クエ；#
（１１）クウォ：=クォ：=クオ；#
（１２）クェ：NF=ケ；=クエ；#
（１３）グァ：=グア；F=ガ；#
（１４）ザ：E=ザー；#
（１５）シム：=シン；#
（１６）シャ：E=シア；#
（１７）シャル：=シアル；#
（１８）スウィ：=スイ；#
（１９）スウェ：=スエ；#
（２０）スムース：=スムーズ；#
（２１）セッサ：E=セッサー；E=セサ；E=セサー；#
（２２）タ：E=ター；#
（２３）チュア：E=チャー；E=チャ；#
（２４）チャンネ：F=チャネ；#
（２５）ツィ：=チ；#
（２６）ティ：E=ティー；#
（２７）ティン：E=チン；#
（２８）ティック：E=チック；#

（２９）ティズム：E=チズム；#
（３０）ティスト：E=チスト；#
（３１）テュ：=チュ；#
（３２）ディジ：F=テイジ；#
（３３）ディ：E=ディー；#
（３４）デュール：E=ジュール；#
（３５）トゥ：=ト；#
（３６）バレエ：=バレー；#
（３７）バッジ：=バッチ；#
（３８）ファウ：=ファー；#
（３９）フェイス：=フェース；#
（４０）フォン：E=ホン；#
（４１）フォーム：=ホーム；#
（４２）ペティ：=ペチ；#
（４３）ボウ：=ボー；#
（４４）マ：E=マー；#
（４５）レイ：=レー；#
（４６）レクリエー：=レクレー；#
（４７）ロイヤル：=ローヤル；#
（４８）ワイヤー：=ワイヤ；#
（４９）ーサ：E=ーサー；#
（５０）ックス：=クス；#
（５１）ッサンス：=サンス；#
（５２）ヴァ：=バ；#
（５３）ヴィ：=ビ；#
（５４）ヴェ：=ベ；#
（５５）ヴォ：=ボ；#
（５６）ヴュ：=ビュ；#

4

様式9

論 文 審 査 の 結 果 の 要 旨

報告番号	甲 工 乙 工 第 32 号 工 修	氏 名	獅々堀 正幹
審査委員	主 査 青 江 順 一 副 査 大 恵 俊一郎 副 査 矢 野 米 雄		

学位論文題目
2進ディジタル探索木を用いた辞書検索法とその応用に関する研究

審査結果の要旨

本論文は，キー検索技法の中で特に2進ディジタル探索木（Binary Digital Search Tree：BDS木と呼ぶ）をハッシュ表に用いた拡張（トライ）ハッシュ法の効率的な検索法について研究したものである。主な内容は，BDS木を先行順ビット列と呼ばれるビット列に圧縮した場合に起こりうる時間的・空間的な問題点に対する改善手法の提案である。また，改善手法を用いた辞書検索法を文書処理の分野へ応用した成果についても報告している。

第1章では本研究の目的を述べると共に本研究の工学上の意義付けを行い，第2章ではキー検索技法の歴史的背景を述べ，各種キー検索技法の特徴および問題点を解説している。第3章では各種キー検索技法の中でも特に高速な検索能力を持つハッシュ法に着目し，BDS木をハッシュ表として用いるトライハッシュ法の構成について説明している。また，登録キー数が多くなった場合にもハッシュ表を主記憶上に格納するために考案された圧縮法，即ちBDS木を先行順ビット列と呼ばれるコンパクトなビット列に圧縮する手法を紹介している。第4章では先行順ビット列で表現されたBDS木の時間的・空間的な問題点を明確にし，それぞれの問題点に対する改善手法を提案している。まず，キー集合が大規模になると，先行順ビット列が非常に長くなり，その結果，ビット列の後方に位置するキーに対する処理の時間効率が低下する。この問題点に対して，BDS木を階層的に分割管理し，不必要な部分木に対する処理を削減することにより，大規模なキー集合に対する時間効率の低下を防いでいる。また，BDS木をパトリシア構造に拡張することにより，従来の手法で用いられていたダミーリーフと呼ばれる擬似的な葉を用いずに，よりコンパクトなビット列に圧縮することに成功している。更に，それぞれの提案手法の理論的評価，及び実験による具体的評価を与え，本手法の有効性を実証している。更に，第5章では文書処理分野への応用として，改善手法が施されたBDS木（改良BDS木）で構成された片仮名辞書と片仮名変換ルールを用いた高速かつ正確な片仮名異表記生成手法，および改良BDS木で構成された文書構造データベースと図表配置基準に従って考案した図表配置アルゴリズムを用いた自動図表配置手法について述べている。

以上本研究は，先行順ビット列で表現された2進ディジタル探索木の問題点に対する改善手法と実際の文書処理分野への応用について議論，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。