平衡性を持つ3分木構造の提案とその評価

著者	新森 修一, 永福 俊也, 礒道 隆一
雑誌名	鹿児島大学理学部紀要=Reports of the Faculty of Science, Kagoshima University
巻	47
ページ	1-12
URL	http://hdl.handle.net/10232/00005878

Rep. Fac. Sci., Kagoshima Univ.,
No. 47, pp. 1–12 (2014)

平衡性を持つ3分木構造の提案とその評価

A Proposal of Balanced Trees with 3-subtrees and its Evaluation

新森修一[1)]・永福俊也[1)]・礒道隆一[2)]
Shuichi SHINMORI[1)], Shunya EIFUKU[1)] and Ryuichi ISOMICHI[2)]

Abstract: In this paper, we propose to develop a balanced tree with 3-subtrees, for the purpose of increasing search efficiency, and examine various evaluations for the extended AVL tree. The data structure of this balanced tree contains right-and-left balanced subtrees and central subtrees that match prefixes character by character by implementing the radix search method. By introducing central partially subtrees, it becomes easy to look for the target data. A numerical experiment using 10 million pieces with 100-digit character strings in decimal number confirmed that the construction time was about 81% of the usual AVL tree. Moreover, it is possible to delete data completely and that experiment confirmed that the removal time was about 78% of the usual AVL trees.

Keyword: data structure, algorithm, AVL trees, computational complexity.

1. はじめに

多数のデータの中から必要なデータを探しだす探索方法として，木構造を利用したアルゴリズムが頻繁に利用されている。最も基本的な木構造に2分探索木があるが，木構造の偏りを解消し必要なデータを素早く発見することを目的に平衡木の一種である AVL 木 [1] 提案され，幾つかの考察や他分野への応用研究 [6-14] が行われている。その中の一研究において，文字列の検索に応用された AVL 木として「拡張した AVL 木」の提案 [2,3] があり，5分木に拡張された AVL 木の構造と探索・挿入・削除などの基本操作を行うアルゴリズムの研究がなされている。この研究においては，各節点が5つの部分木（left, right, front, back, center）を持ち，1文字（桁）ずつ比較して前方が一致する文字列を部分木とするもので，既存の AVL 木や B 木（*k* 分木）[15,16] よりも短時間でデータ構造を構築することができるなどの結果 [4,5] が得られている。しかし，5分木中2つの部分木（front, back）に対しては構造的に平衡性を維持できないこと，条件によってはデータを完全に削除することができないこと，データの挿入削除が繰り返されるうちに不要となったラベルが残ってしまうことなどの問題点があった。そこで本論文では，平衡性を持つために各節点が3つの部分木（left, right, center）を持つ「拡張した AVL 木」のデータ構造，およびに探索・挿入・削除などの操作を行うアルゴリズムを提案する。また既存の AVL 木との比較を行うことで，文字列探索における効率性を考察する。以下，2節では平衡性を持つ3分木構造の提案，3節では基本操作アルゴリズムの詳述，4節では10進数の文字列データを用いた数値実験による評価，5節ではまとめと今後の課題について触れる。

2. 平衡性を持つ3分木構造の提案

まず，文字列探索に対応した既存の AVL 木には以下のような特徴がある。

特徴1：2分木を基にしたリスト構造による動的な処理が可能である。

特徴2：効率的な探索のために，各節点が平衡性を持つ。

特徴3：各節点に文字列を格納し，木構造全体で辞書式順序を保持する。

特徴1について，既存の AVL 木では各節点の持つ左右の子である節点と親節点を指すポインタを持つこ

1）鹿児島大学大学院理工学研究科数理情報科専攻　〒890-0065 鹿児島市郡元1丁目21-35
Graduate School of Science and Engineering, Kagoshima University, Kagoshima 890-0065, Japan
2）株式会社富士通鹿児島インフォネット　〒890-0064　鹿児島市鴨池新町5-1

とで，双方向リストを実現している。また必要に応じて節点同士の連結や分離ができるので，挿入操作や削除操作など動的な処理を行うことが可能である。

特徴2について，AVL 木とは，どの節点においても左右部分木の高さの差が最大1の木構造のことである。そして左右部分木の高さの差が2となった節点に対して回転操作を実行することで，木構造の平衡化を行う。回転操作とは，左右部分木の高さの差が1以内となるように節点と部分木の関係を組み替えることで，一重回転と二重回転の2種類がある。図1と図2はそれぞれの一例であり，四角の破線で囲まれた部分が回転操作によって変化した部分である。2種類の回転操作は節点の状態によって使い分けられる。つまり回転操作の必要性やその使い分けは，節点の状態から判断する必要がある。よって各節点には，節点自身の状態を格納させておく必要がある。

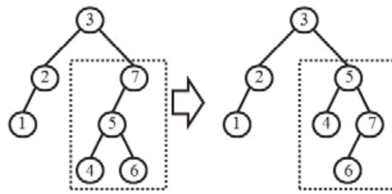

図1　一重回転の例

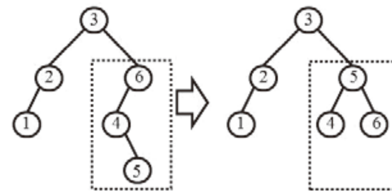

図2　二重回転の例

特徴3について，既存の AVL 木では各節点が大きさ S（桁）の文字列と終端文字を格納している。特に本研究では10進数の整数データを対象にしているので，桁を揃えるために不足した前方の桁を ‘0’ で埋めている。例として $S = 3$の場合，整数24を文字列 “024” として扱う。また文字列探索のアルゴリズムとして，基数探索法を実装している。これは一文字ごとにデータ同士の比較を行い，これを繰り返すことで整列する方法である。文字は全順序の性質があり，文字で構成された文字列は辞書式順序の性質を持つ。よって木構造内の節点は，格納した文字列の辞書式順序に従った配置となる。

しかし基数探索法の考え方をそのまま実装した既存の AVL 木では，ある程度の桁の比較を進めて左右どちらかの部分木へ進行するが，進行後，また最初の桁から比較を進める必要がある。これは先頭の数文字が同一であるデータ群から目的のデータを探す場合に，同一である数文字を何度も比較しなおすことになる。例として図3のような構造の木があるとする。この例では $S = 3$であり，終端文字（記号 # で表す）を含めて4文字格納している。文字列 “035” を探したい場合に，根αから探索を始めて ‘2’ で対応する桁の探索値 ‘3’ の方が大きいことが分かる。右部分木に進行することで目的の節点βが発見できるが，発見するまでに比較した文字と比較回数は ‘0’，‘2’，‘0’，‘3’，‘5’ の5回であり，2節点で一致した先頭文字 ‘0’ の比較が重複している。同じ先頭文字を持つ節点が増えたり，または先頭文字がより多く一致すれば，何度も文字列の先頭から同じ文字を比較することになり，それだけ探索に時間がかかることになる。

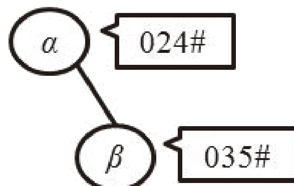

図3　既存の AVL 木の一例

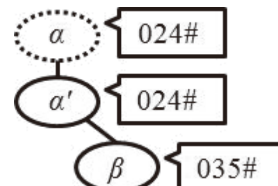

図4　3分木構造の一例

そこで，節点の移動ごとに先頭から比較する無駄を除き，比較桁を戻さずに探索を行う「拡張した AVL 木」が考えられている。既存の AVL 木を多分木に拡張することで，比較桁を戻さないデータ構造とし，探索を高速に行えるようにする試みである。本論文では，平衡性を持つ左右部分木（left, right）の他に，中央部分木（center）を持った「拡張した AVL 木」を提案し，「平衡3分木」と呼ぶことにする。平衡3分木の構造は次のように定義できる。

［定義1］　平衡3分木の構造

1. S 桁のデータ $a_0, a_1, \ldots, a_{s-1}$ と終端文字を持つ節点 u と u の3つの部分木である T_l, T_r, T_c からなる。3つの部分木は，左部分木 T_l，右部分木 T_r，中央部分木 T_c とする。ただし，T_l, T_r, T_c は全く節点を持たない空木となることもある。

2. 部分木の高さとはT_l, T_rのみであり，T_cの高さは含めない。このとき各部分木の根に対して，T_lとT_rの高さの差は高々1である。
3. sを比較する桁の進行数として，各節点の添え字sの文字をキーとする。任意の節点uのキーa_sとそれぞれ比較して，T_lの添え字sの文字はすべてa_sより小さく，T_cの添え字sの文字はすべてa_sと等しく，T_rの添え字sの文字はすべてa_sより大きい。
4. T_cが空木とならない場合，T_cの親uはラベルとし，T_cの根に節点uと同一のデータを保持させる。ラベルuが添え字sの文字をキーとするとき，T_cの根はa_{s+1}をキーとする。

図4は図3を平衡3分木に変換したものである。節点αの中央部分木の根には，αと同じデータを持った節点α′が存在する。節点αはラベルであり，破線の丸で表現している。ラベルとは構造上データを持つ通常の節点と同じであるが，キーの値で節点同士の大小関係を識別する機能のみを持つ節点でありデータを持たない。文字列探索の際には，探索値とラベルのキーが一致した場合に中央部分木へ進行する。“035” を発見するまでに比較した文字と比較回数は ‘0’，‘2’，‘3’，‘5’ の4回で，既存の AVL 木よりも少ない比較回数で発見できる。

平衡3分木をプログラミング言語 C を用いて実現した各節点のデータ構造を次に載せる。一節点でS + 29バイト（Sは文字列の桁数）の領域を使うことになる。

```
 1  enum  flag {LA, NU};
 2  enum  sub {RO, LT, RT, CT};
 3  enum  diff {EVEN, LEFT, RIGHT};
 4  struct  ext_avl{
 5        char   element[S+1] ;              // S 文字と終端文字を格納
 6        enum flag  flag ;                  // ラベルかどうかの判別
 7        enum sub  sub_tree ;               // 親からみた部分木としての方向
 8        enum diff  diff ;                  // 左右部分木の高い方向
 9        struct ext_avl  *left ;            // 左の子を指すポインタ
10        struct ext_avl  *right ;           // 右の子を指すポインタ
11        struct ext_avl  *center ;          // 中央の子を指すポインタ
12        struct ext_avl  *parent ;          // 親節点を指すポインタ
13  }
```

5行目は既存の AVL 木の特徴3を満たす。6行目は節点がラベルかどうかを判別する。LA ならばラベル，NU ならば通常の文字列データを持つ節点である。7, 8行目は既存の AVL 木の特徴2を満たす。sub_tree が RO ならば根の節点，LT ならば左部分木，RT ならば右部分木，CT ならば中央部分木を意味する。diff が EVEN ならば左右部分木の高さの差は等しく，LEFT ならば左部分木が高く，RIGHT ならば右部分木が高いことを示す。9から12行目は既存の AVL 木の特徴1を満たす。中央部分木も左右部分木と同様に双方向リストとなっている。

図5は平衡3分木Tの例である。S = 3であり，12個のデータを格納している。同じ先頭文字を持つデータもあるのでラベルが4個あり，木構造は16個の節点で構成されている。一節点に対しS+29=3+29=32バイトが必要であるため，図の木構造全体では32 × 16 = 512バイトの領域を使う。次節でこの平衡3分木の基本操作アルゴリズムを詳述する。

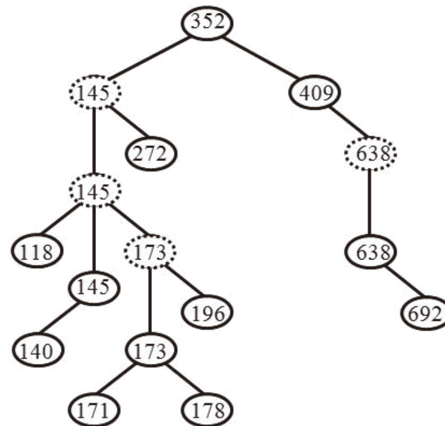

図5　平衡3分木 T の例

3. 基本操作アルゴリズム

まず，目的の節点（文字列データ）を探すアルゴリズムを述べる。探索したい文字列$a_0a_1\ldots a_S$を用意し，全順序に従って，着目している節点pの持つデータ$b_0b_1\ldots b_S$とを1文字ずつ大小比較を行い，結果に従って着目節点を移動させる。S桁目は終端文字なので$S-1$桁目までの一致で目的の節点pを探索できたこと（探索成功）になる。なお，ラベルは中央部分木へ進行するかを判定する識別用の節点でありデータを持たないので，目的の節点ではない。

探索アルゴリズム

1　比較桁sを0文字目に初期化する。

2　ポインタpで木構造の根を指す。

3　節点pが存在している限り，a_sとb_sの大小比較を繰り返す。

　3.1　a_sが小さい場合，ポインタpでT_lの根を指す。（3に戻る）

　3.2　a_sが大きい場合，ポインタpでT_rの根を指す。（3に戻る）

　3.3　a_sがb_sと等しい場合，

　　3.3.1　$s=S-1$ならば，探索成功。（終了）

　　3.3.2　$s\neq S-1$ならば，

　　　3.3.2.1　節点pが中央部分木を持つ場合，

　　　　3.3.2.1.1　ポインタpでの根を指す。

　　　　3.3.2.1.2　sを1増加させる。（3に戻る）

　　　3.3.2.2　節点pが中央部分木を持たない場合，ループAを繰り返す。

4　目的の節点が見つからず探索失敗。（終了）

上のアルゴリズム中，ループAがあるが，これは次のようになる。ループAを実行する状況とは，$a_s=b_s$，$s\neq S-1$の条件を満たし，節点pが中央部分木のない通常節点である場合である。ループAを実行するとpが他の節点に移ることはなく，残りの文字を比較することになる。

探索アルゴリズムのループA

1　比較桁sを1増加させる。

2　a_sとb_sとを比較する。

　2.1　比較して異なる場合，目的のデータではないので探索失敗。（終了）

　2.2　比較して同じである場合，

2.2.1　$s = S - 1$ ならば，探索成功。（終了）
2.2.2　そうでないならば，1に戻る。

例として図5の木構造 T から文字列 “145” を探してみる。T の根の節点は存在し，b_0 = ‘3’ で $a_0 < b_0$ なので，着目節点が左部分木の根に移る。次は b_0 = ‘1’ で $a_0 = b_0$ であり，中央部分木に進行して次の桁を比較する。次も同様に中央部分木に進行して，$a_2 = b_2$ となりデータ “145” を持つ通常の節点を発見できた。このとき，比較回数は ‘3’，‘1’，‘4’，‘5’ の4回である。今度は木構造 T から文字列 “359” を探してみる。根の節点で $a_0 = b_0$ となったので，ループ A を実行し残りの桁を比較する。$s = 1$ のときは同じだったが，$s = 2$ のとき比較しても異なるので，“359” は木構造に格納されていないとわかる。

次に，文字列 $a_0a_1 \ldots a_s$ を木構造に格納するアルゴリズムを述べる。着目ポインタ $*t$ の行き先の節点が持つデータと $b_0b_1 \ldots b_s$ を1文字ずつ大小比較を行い，結果に従って着目節点を移動させる。目的のデータが格納されていなければ，データ $a_0a_1 \ldots a_s$ を持つ節点を辞書式順序を満たすように挿入する。挿入の際には文字列 $d_0d_1 \ldots d_s$ と列挙型 sub と節点 p を参考に，表1の情報を格納した節点を作成する。

表1　作成する節点の情報

変数	element[S+1]	flag	sub_tree	diff	*left	*right	*center	*parent
値	$d_0d_1 \ldots d_s$	NU	sub	EVEN	NULL	NULL	NULL	p

挿入アルゴリズム

1　比較桁 s を0文字目に初期化する。
2　sub を RO で初期化する。
3　ポインタ $*t$ で木構造を指す。
4　ポインタ p で $*t$ の行き先の節点を指す。
5　$*t$ が空木でない限り，a_s と b_s の大小比較を繰り返す。
5.1　a_s が小さい場合，
5.1.1　sub に LT を記録する。
5.1.2　ポインタ p で $*t$ の行き先の節点を指す。
5.1.3　ポインタ $*t$ で T_l を指す。（5に戻る）
5.2　a_s が大きい場合，
5.2.1　sub に RT を記録する。
5.2.2　ポインタ p で $*t$ の行き先の節点を指す。
5.2.3　ポインタ $*t$ で T_r を指す。（5に戻る）
5.3　a_s が b_s と等しい場合，
5.3.1　$*t$ の行き先の節点が中央部分木を持つ場合，
5.3.1.1　ポインタ $*t$ で T_c を指す。
5.3.1.2　s を1増加させる。（5に戻る）
5.3.2　$*t$ の行き先の節点が中央部分木を持たない場合，
5.3.2.1　整数 k を0で初期化する。
5.3.2.2　次を実行し，a_s と b_s とが食い違う桁を調べる。
5.3.2.2.1　s と k をそれぞれ1増加させる。
5.3.2.2.2　$s = S - 1$ ならば，格納済みで挿入不能。（終了）
5.3.2.2.3　a_s と b_s を比較し，一致ならば5.3.2.2に戻る。
5.3.2.3　sub に CT を記録する。
5.3.2.4　次を k 回繰り返す。（終われば5に戻る）。
5.3.2.4.1　ポインタ p で $*t$ の行き先の節点を指す。

5.3.2.4.2　節点 p をラベルにする。

5.3.2.4.3　$b_0 b_1 \ldots b_s$ を格納する節点を，$*t$ の行き先の節点の T_c に作成する。

5.3.2.4.4　ポインタ $*t$ で T_c を指す。（5.3.2.4に戻る）

6　挿入可能であり，$a_0 a_1 \ldots a_s$ を格納する節点を，$*t$ の行き先に作成する。（終了）

例として図5の木構造 T に文字列 “405” を格納する。T の根から右に移動し，“409” を格納する節点 p と大小比較し，$a_0 = b_0$ となった。p は中央部分木を持たないので，残りの桁を比較して $a_2 \neq b_2$ となることが分かった。よって，節点 p をラベル節点にして，中央部分木にラベル節点 p' と，その中央部分木に通常の節点 p'' が作られた。最後に大小比較を行い $a_2 < b_2$ なので節点 p'' に左部分木に節点を作成し “405” を格納する。図6は挿入後の木構造 T である。

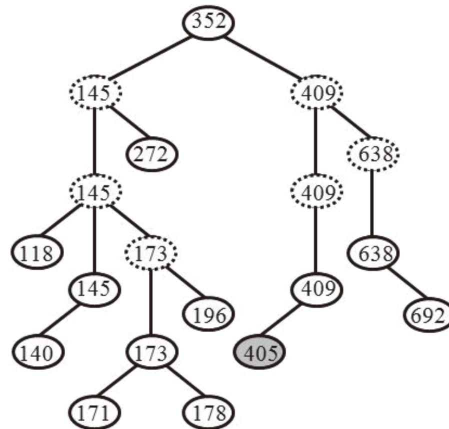

図6　図5の T に “405” を格納した場合

次に，目的のデータを木構造から削除するアルゴリズムを述べる。目的のデータ $a_0 a_1 \ldots a_s$ を探す過程と実際に削除する過程に分けられるが，目的のデータを探す過程は探索アルゴリズムと同じであり，探索成功で目的の節点 q が見つかったとする。削除する過程では，既存の AVL 木と同様に節点 q の部分木の有無で場合分けを行う。部分木を持つならば，右部分木 T_r の最小値が格納されている節点（なければ左部分木の最大値が格納されている節点）を p としたとき，節点 q のデータを節点 p のデータ $b_0 b_1 \ldots b_s$ に書き換えて，節点 p を削除する。

簡単な例であるが，図7は図4から “024” を削除した木構造である。α' のデータを “035” に書き換えβを分離することで，“024” を格納した節点を木構造から削除したことになる。節点βに確保した領域は解放しておく。

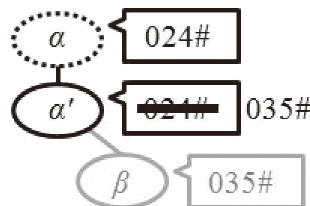

図7　データ削除の例1

しかし，節点 q の代わりに削除する節点 p がラベルだった場合は，節点 q を削除してその位置に節点 p を持ってくることになる。図8, 9は上記のような状態と削除後の様子である。三角は親の省略，×印はその方向に部分木を持たないことを表す。削除した節点の領域は解放しておく。以上を考慮した「データ削除アルゴリズム」を次に示す。

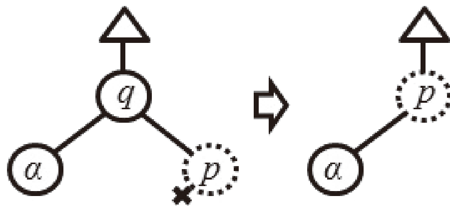

図8　データ削除の例2

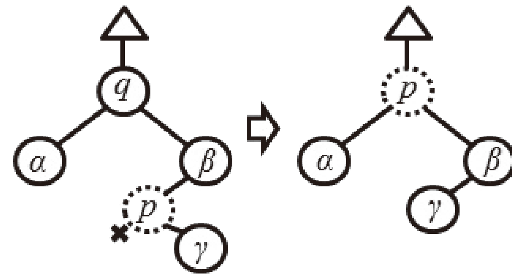

図9　データ削除の例3

データ削除アルゴリズム

1　節点 q の T_l, T_r が空木である場合，
  1.1　節点 q の親節点は，節点 q を切り離す。（終了）
2　節点 q の T_l が空木である場合（T_r に関しても対称的に同様），
  2.1　ポインタ *t で節点 q の T_r を指す。
  2.2　*t の行き先の節点がラベルならば，
    2.2.1　ポインタ p で *t の行き先の節点を指す。
    2.2.2　節点 p に節点 q の sub_tree を格納する。
    2.2.3　節点 p の親として，節点 q の親節点を連結させる。（終了）
  2.3　節点 q に *t の行き先 $b_0b_1 \ldots b_s$ の文字列を格納する。
  2.4　*t の行き先の節点に対して再帰実行する。
  2.5　節点 q の親節点は，そのまま節点 q を部分木とする。（終了）
3　節点 q の T_l, T_r が空木でない場合，
  3.1　ポインタ *t で節点 q の T_r を指す。
  3.2　*t の行き先の節点がラベルであり，かつ T_l を空木とするならば，
    3.2.1　ポインタ p で *t の行き先を指す。
    3.2.2　節点 p に節点 q の sub_tree と diff を格納する。
    3.2.3　節点 p の左部分木として，節点 q の T_l の根を連結させる。
    3.2.4　節点 p の親として，節点 q の親節点を連結させる。（終了）
  3.3　*t の行き先の節点が T_l を持つ限り，次を実行する。
    3.3.1　ポインタ *t で T_l を指す。（3.3に戻る）
  3.4　*t の行き先の節点がラベルならば，
    3.4.1　ポインタ p で *t の行き先を指す。
    3.4.2　節点 p の T_r が空木でないならば，T_r の根に節点 q の sub_tree を格納する。
    3.4.3　節点 p の親節点の左部分木として，節点 p の T_r の根を連結させる。
    3.4.4　節点 p に節点 q の sub_tree と diff を格納する。
    3.4.5　節点 p の左部分木として，節点 q の T_l の根を連結させる。
    3.4.6　節点 p の右部分木として，節点 q の T_r の根を連結させる。
    3.4.7　節点 p の親として，節点 q の親節点を連結させる。（終了）
  3.5　節点 q に *t の行き先の文字列 $b_0b_1 \ldots b_s$ を格納する。
  3.6　*t の行き先の節点に対して再帰実行する。
  3.7　節点 q の親節点は，そのまま節点 q を部分木とする。（終了）

例として図6の木構造 T から文字列 “409” を削除する。“409” を格納した通常の節点 q は左部分木の節点 p のみを持ち，さらに節点 p はラベルではないので，“405” を節点 r に格納して節点 p について再帰実行する。節点 p は部分木を持たないので，木構造全体から節点 p を削除すればよい。図10は削除後の T である。

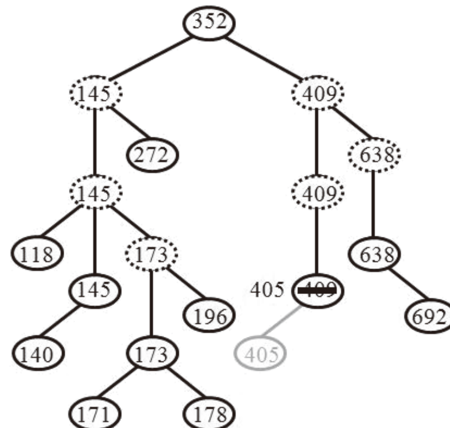

図10 “409” を削除した図6の T

ここまでに述べたアルゴリズムを用いても，全ての節点を削除できないことがある。例として図11は図7の木構造から “035” を削除したときである。ラベル節点を削除する手段がないので，格納したデータを全て削除しても元々データを持たないラベル節点が残り続ける。その事態を回避するためにデータ削除後に余分な節点を削除すること，すなわち，具体例として図7から図12の状態を目指すことで，節点の個数を必要最小限に保ち，全ての節点を削除できるようになる。その操作を最適化と呼ぶことにする。

図11 ラベルが残った平衡3分木

図12 最適化を行った図7の平衡3分木

削除すべき余分な節点について考える。中央部分木の根である節点は，作成時点で必ず左右どちらかの部分木に新たな文字列を格納する節点を持つ。よって T_l または T_r が空木でなければ，節点 p は中央部分木である必要のある節点である。つまり最適化を行う節点 p の条件は，親節点の T_c の根であり，かつ節点 p の T_l, T_r が両方空木である場合である。また節点 p の T_c が空木でないならば，節点 p はラベルであり識別用の節点として必要である。以下では不要となった節点を削除する最適化アルゴリズムを述べる。着目する節点 p を参照して処理を行う。

最適化アルゴリズム

1 次のような場合は，アルゴリズムを終了する。
 1.1 節点 p が存在しない場合や，親から見た中央部分木でない場合。
 1.2 節点 p の T_l, T_r が空木でない場合。
 1.3 節点 p の T_c が空木でない場合。
2 ポインタ p で節点 p の親節点を指す。
3 節点 p に T_c の根の文字列を格納する
4 節点 p は，T_c を空木とする。
5 節点 p を通常の節点とする。
6 節点 p に対して再帰実行する。（終了）

例として図10の木構造 T の “405” を格納した節点 p に最適化を行う。節点 p は中央部分木である必要のない節点なので，親節点 p' を通常の節点にして節点 p を削除する。その際にデータ “405” を木構造に保持させるために，節点 p が格納する文字列 “405” を親節点 p' にも格納する。こうすることで余分な節点 p を削除することができた。そして，節点 p' に対しても最適化が必要かを考える。節点 p' もまた中央部分木である必要のない節点なので，同様に節点 p' の親節点 p'' を通常の節点にして節点 p' を削除する。節点 p'' は中央部分木ではないので最適化を終える。図13は最適化を実行した後の図10の木構造 T である。最適化によって，節点の個数を必要最小限に保つことができた。

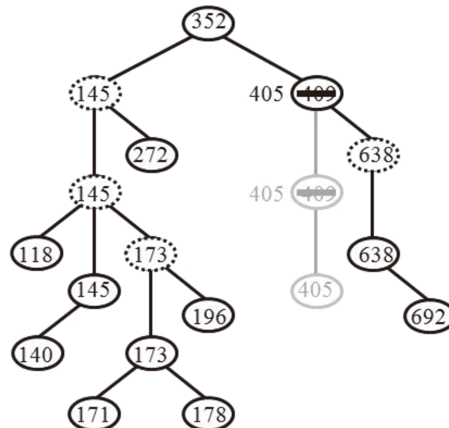

図13　最適化を行った図10の木 T

4. 数値実験による評価

既存の AVL 木と本研究で提案した平衡3分木に対して，時間や領域などの資源について数値実験により比較評価を行う。実験に先立って，乱数によって生成された S = 100桁の10進数の文字列データ10,000,000個を1組として10組を用意する。次の数値実験では木構造に対する操作の後に比較項目を記録し，各組の記録の平均値を結果としている。以下の項目を比較項目とする。

- 実行時間
  各操作の開始から終了までに要した時間（秒）。
- 比較回数
  木構造の探索のために，探索値 a_s と格納データ b_s を比較した回数。
- 節点数
  木構造内の通常の節点とラベル節点の数。
- 領域量
  木構造全体の領域量（バイト）。一節点の領域量×節点数で計算する。

（1）木構造の構築について

まず木構造の構築に必要な資源を考える。データ10,000,000個を読みとり，空木から木構造を構築する。数値実験の結果は表2のようになった。

表2　木構造構築の場合

	実行時間	比較回数	節点数	領域量
既存の AVL 木 (a)	26.92	879,148,021	10,000,000	1,210,000,000
平衡３分木 (b)	21.71	200,568,387	14,276,243	1,841,635,334
(b) / (a)	80.6%	22.8%	142.8%	152.2%

2分木構造から3分木構造に変えたことで，節点移動の際に比較桁を0桁目に戻す必要がなくなったので，比較回数は約23% と大きく減少した。そのことが，実行時間が約81% に減少したことに繋がっていると考えられる。実行時間が比較回数に比例しなかった理由として，一節点のメンバが多いので節点作成に必要な時間が増えたことや，探索における処理が増えたことがあげられる。また，データを持たないラベル節点を作成する必要があるために，節点数が約1.43倍，領域量が約1.52倍に増えることがわかった。ただ，領域量の増加率については，理論的には10進数の文字列データの場合，データ数が増加すれば最小で約1.11倍に近づいて行くこと，また，一般の文字列では，1.1から1.2倍程度のかなり小さな値になることが予想される。

（2）木構造の完全撤去について

次に木構造の撤去に必要な資源を考える。先ほど構築した木構造に対して，同じデータを使うことで，格納したデータを格納順に全て削除することができる。数値実験の結果は表3のようになった。

表3　木構造撤去の場合

	実行時間	比較回数	節点数	領域量
既存の AVL 木 (a)	30.47	1,718,808,620	0	0
平衡３分木 (b)	23.77	1,121,719,378	0	0
(b) / (a)	78.0%	65.3%		

削除の際の節点探索では，目的の文字列と格納したデータが完全一致するまで比較を行うので，木構造の撤去には構築よりも多くの比較を行う。それでも比較桁を0桁目に戻さないので，比較回数を約65％まで減らすことができ，実行時間も約78％に短縮できた。また，構築と撤去で実行時間の差が小さい理由として，削除の際にはメンバの多さが実行時間に影響しなかったことが考えられる。表からもわかるように，10,000,000回のデータ削除操作で全ての節点を完全に削除可能であることが確認できた。

（3）データ探索について

木構造でデータ探索を行う際に必要な資源を考える。構築された木構造に対して，同じデータを用いて探索した場合と，別のデータを用いて探索した場合を考える。同じデータを用いた場合は全て探索成功となるが，別のデータを用いた場合は格納されたデータと一致する可能性は極めて小さいので，その場合は大抵全て探索失敗となる。数値実験の結果はそれぞれ表4と表5のようになった。

表4　同じデータを用いた探索の場合

	実行時間	比較回数	節点	領域量
既存の AVL 木 (a)	21.79	1,846,635,046	10,000,000	1,210,000,000
平衡３分木 (b)	18.34	1,131,822,906	14,276,243	1,841,635,334
(b) / (a)	84.2%	61.3%	142.8%	152.2%

表5　別のデータを用いた探索の場合

	実行時間	比較回数	節点	領域量
既存の AVL 木 (a)	21.87	985,070,464	10,000,000	1,210,000,000
平衡３分木 (b)	15.75	209,322,777	14,276,243	1,841,635,334
(b) / (a)	72.0%	21.2%	142.8%	152.2%

表4の結果は，10,000,000個のデータを探索した際における実行時間，比較回数の各比較項目の最大値といえる。このことからデータ探索の実行時間の最大値を約84% に短縮できることがわかった。また表5の結果は，乱数によって生成したデータを用いているので，10,000,000個のデータを探索した際における実行時間の推定値といえる。このことから乱数データの探索における実行時間を約72% に抑えることが可能な構造となっていることがわかった。

表4と表5の結果を比較する。ここでは実行時間に関して「（表5の結果）÷（表4の結果）」の比率を考える。既存の AVL 木では，実行時間の比率は約100% とほとんど同じといえる結果となった。その理由として，$a_s \neq b_s$となる確率が10進数データなので90% と大きく，さらに$a_s \neq b_s$の場合には$a_s = b_s$の場合よりも処理が多いことが挙げられる。しかし平衡3分木では，実行時間の比率が約86% となった。これは既存の AVL 木とは逆に，$a_s \neq b_s$の場合の処理が1つだけと$a_s = b_s$の場合よりも処理が少ないので，実行時間の最大値よりも推定値が小さくなったと考えられる。

5. まとめ

本論文では，木構造の文字列探索における効率性の向上を目的とした「拡張した AVL 木」の一つとして，平衡3分木のデータ構造とその基本アルゴリズムを提案し，これに対して様々な数値実験や考察を行った。

提案した平衡3分木は文字列探索に対応した3分木構造で，特に中央部分木には前方一致となる文字列を格納した節点が配置された木構造を構成している。文字列探索の際に文字列を先頭から見直す必要がない「拡張した AVL 木」の長所を引き継いでおり，比較回数を既存の AVL 木よりも格段に少なくできる構造である。また３分木構造にしたことで，既存の AVL 木と同様に平衡性の維持が容易になり，さらに最適化と呼ばれる操作を導入したことで，ラベルも含めた全ての節点を完全に削除することが可能になった。

数値実験により既存の AVL 木と比較して，木構造の構築時間を約81% に，完全撤去時間を約78% に抑えることが確認できた。またデータ探索において，推定で約72%，最悪でも約84% の実行時間で探索可能であること，探索データ次第でほとんど最大値であった実行時間を短くすることが可能であることが判明した。以上のことより，平衡3分木によって効率的なデータ探索が可能であるといえる結果を得ることができた。

今後の課題として，他の「拡張した AVL 木」への応用，実用性の高い他のデータ構造（例えば [17,18]）との比較，領域量の理論的な解析や削減方法の提案，実用性のある「ひらがな」など10進数以外の文字列データや全角文字への適用などが挙げられる。

参考文献

[1] Adel'son-Vel'skii G.M. and Landis Y.M.: An algorithms for the organization of information, Soviet Math., Vol.3, pp.1259–1263, 1962.

[2] 竹之下朗，新森修一：AVL 木の拡張とそのアルゴリズム，日本応用数理学会2007年度年会講演予稿集，(2007), 218–219.

[3] 竹之下朗：文字列探索に適した AVL 木の拡張とその評価に関する研究，PhD thesis, 鹿児島大学大学院理工学研究科，2011.

[4] 竹之下朗，新森修一：5分木に拡張した AVL 木の拡張とその評価，日本応用数理学会論文誌，Vol.20, No.3, pp.203–218, 2010.

[5] 竹之下朗，新森修一：AVL 木の拡張と B 木との比較評価，鹿児島大学理学部紀要，Vol.44, pp.15–26, 2011.

[6] Robert W. Irving and Lorna Love: The suffix binary search tree and suffix AVL tree, Journal of Discrete Algorithms., Vol.1, pp.387–408, 2003.

[7] Kim S. Larson: AVL Tree with Relaxed Balance, Journal of Computer and System Science, Vol.61, pp.508–522, 2000.

[8] Kim S. Larsen, Eljas Soisalon-Soininen and Peter Widmaner: Relaxed Balance Using Standard Rotation, Algorismica, Vol.31. pp.501–512, 2001.

[9] R. Nakamura, A. Tada and T. Itokawa: An Analysis of the AVL Balanced Tree Insertion Algorithm, IEICE Transactions, Vol.86, pp.1067–1074, 2003.

[10] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev: Using AVL Trees for Fault Tolerant Group Key Management, Informational Journal on Information Security, Vol.1, pp.84–99, 2000.

[11] Wojciech Rytter: Application of Lempel-Ziv Factorization To The Approximation of Grammar-Based Commpression, Theoretical Computer Science, Vol.302, pp.211–222, 2003.

[12] David Wood, Kowari: A Platform for Semantic Web Storage and Analysis, In Xtech 2005 Conference, pp.05–0402, 2005.

[13] 横尾英俊，杉浦由紀江：AVL 木を利用した適応的数値データ圧縮法とその改良，情報処理学会論文誌，Vol.34, pp.1–9, 1993.

[14] Yi-Ying Zhang, Wen-Cheng Yang, Kee-Bum Kim and Myong-Soon Park: An AVL Tree-Based Dynamic Key Management in Hierarchical Wireless Sensor Network, International Conference on Intelligent Information Hiding and Multimadia Signal Processing, pp.298–303, 2008.

[15] Rudolf Bayer and E.McCreight: Organization and maintenance of large ordered indexes, pp.245–262. Springer-Verlag New York,Inc., NewYork, NY, USA, 2002.

[16] Robert Sedgewick: Algorithm in C Third Edition.Addison-Wesley, Pearson Education, Inc, 2004. 野下，星，佐藤，田口共訳，近代科学社，2004.

[17] 望月久稔，中村康正，尾崎拓郎：ダブル配列によるパトリシアを拡張した基数探索法，日本データベース学会 Letters, Vol.6, pp.9–12, 2007.

[18] 中村康正，望月久稔：ダブル配列上の遷移数を抑制した基数探索法の提案．情報処理学会情報学基礎研究会報告，Vol.2007, No.34, pp.41–46, 2007-03-27.