平衡3分木構造の改良とその評価

Rep. Fac. Sci., Kagoshima Univ.,
No. 48, pp. 23–34 (2015)

平衡3分木構造の改良とその評価

Improvement of Balanced Trees with 3-subtrees and its Evaluation

新森修一[1)]・永福俊也[2)]
Shuichi SHINMORI[1)], Shunya EIFUKU[2)]

Abstract: In this paper, we propose to improve a balanced tree with 3-subtrees for the purpose of compressing the memory capacity of the extended AVL tree, and increasing search efficiency. The proposing structure of an improvement balanced tree with 3-subtrees is the same structure as a balanced tree with 3-subtrees. But to settle an increase problem of the memory capacity of the extended AVL tree, each node stocks just one character and only information on necessity minimum is secured dynamically. A numerical experiment using 10 million pieces with 100-digit character strings in decimal number confirmed that the memory capacity is reduced about 73% by the improvement. Moreover, that experiment confirmed that the search time of data is reduced about 10% by the improvement.

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について，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発見しやすくする方法として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。その一つとして文字列の検索に応用された AVL 木として「拡張した AVL 木」の提案と様々な評価や比較 [2–5] がなされている。また，3分木に拡張された AVL 木（平衡3分木）の構造と探索・挿入・削除などの操作を行うアルゴリズムの研究 [6] を行っている。この研究においては，各節点が3つの部分木 (left, right, 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 木よりも短時間でデータ構造を構築することができること，データ探索時間の短縮化が可能なことなどの結果が得られている。一方，同じ100桁のデータ10,000,000個を用いて木構造を構築しても，既存の AVL 木と比較して節点数が約1.4倍，領域量は約1.5倍と大きくなりすぎるという問題点があった。そこで本論文では，平衡3分木の領域量を削減するための改良を加えた「改良平衡3分木」のデータ構造，および探索・挿入・削除などの操作を行うアルゴリズムの提案する。また平衡3分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節では改良平衡3分木の提案，3節では基本操作アルゴリズムの紹介，4節では10進数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. 改良平衡3分木の提案

まず，文字列探索を行う平衡3分木は，次の定義によって定められたデータ構造である。

［定義1］平衡3分木は次の4つの条件を満たす木構造である。

1. S 桁のデータ $a_0, a_1, \ldots, a_{S-1}$ と終端文字 a_S の S+1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。

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 Engineering, Kagoshima University, Kagoshima 890-0065, Japan
2）イー・アンド・エム株式会社　〒102-0084 東京都千代田区二番町9番地

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，（AVL木の性質より）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に節点uと同一のデータを持つ節点が存在する。ラベルuが添え字sの文字a_sをキーとするとき，T_cの根はa_{s+1}をキーとする。

上の定義1の条件1., 2. より，平衡3分木は3分木の形をしているが，実際はAVL木に中央部分木を追加したものである。挿入・削除操作の後でAVL木の性質を満たしているか考察を行い，T_l, T_rで高さの差が2となった節点に対して回転操作を実行することにより，木構造の左右のバランスを常に保つ平衡化を行う。回転操作には図1, 2のような2種類があり，節点の状態によって使い分ける。また，各節点には節点の状態を判断するための情報を格納している。

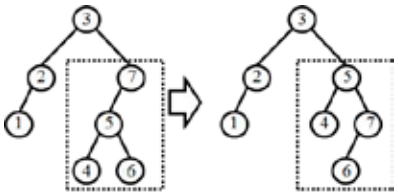

図1　一重回転の例

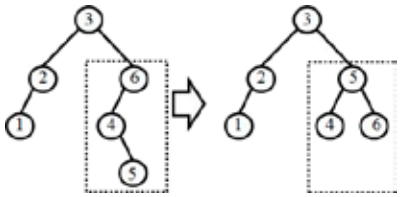

図2　二重回転の例

条件3., 4. を平衡3分木の簡単な例を用いて説明する。図3は平衡3分木の一例であり，$S = 3$桁の10進数文字列データを12個登録している。破線の丸はラベルを表し，節点は16個存在する。また，各節点の領域量は$S + 29$バイトであり，全体で$16 \times (3 + 29) = 512$バイトを使っている。

図3　平衡3分木の一例

図3の平衡3分木から文字列“171”を探索したい場合を考える。根の節点に着目して，探索文字列と格納文字列を先頭から比較して，不一致ならば大小関係に従いT_l, T_rに進行して同じ桁を比較する。一致ならば，T_cがあるなら進行して，次の桁を比較する。$S - 1$桁目までの一致で発見（探索成功）とする。各節点で比較した文字は表1の網掛けの部分である。便宜上，終端文字を‘#’で表現する。

表1　着目節点の比較した文字と進行方向

節点	種類	0桁目	1桁目	2桁目	3桁目	進行方向
探索文字列		‘1’	‘7’	‘1’	‘#’	
“352”	通常	‘3’	‘5’	‘2’	‘#’	T_l
“145”	ラベル	‘1’	‘4’	‘5’	‘#’	T_c
“145”	ラベル	‘1’	‘4’	‘5’	‘#’	T_r
“173”	ラベル	‘1’	‘7’	‘3’	‘#’	T_c
“173”	通常	‘1’	‘7’	‘3’	‘#’	T_l
“171”	通常	‘1’	‘7’	‘1’	‘#’	発見

ラベルである節点には必ず T_c が存在し，そこに属する節点は同じ前方一致の文字列を格納している。またこの例では6回の比較を行っているが，各節点（特にラベル）では1回の比較を行い，部分木へ進行している。この比較した文字がキーであり，各節点に1つずつ存在している。キーの桁は T_c へ進行することで次の桁に移るので，図4のように平衡3分木を T_c で区切ることで，同じ s 桁目をキーとする幾つかのAVL 木に分けることが可能である。この s を階層と呼ぶことにする。

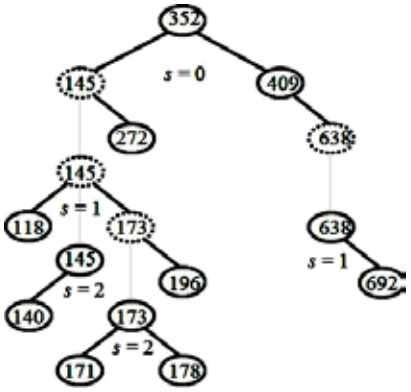

図4　平衡3分木の階層分け

平衡3分木では，全ての節点が S 桁の文字列を格納している。しかしラベルは1文字しか参照せず，通常節点も S 桁参照することは多くない。節点数が増えることで，各節点の無駄な文字も増えていくことになる。このことが領域量増大の原因であると考えられる。

そこで木構造全体の領域量を削減するために，各節点において参照する必要最低限の文字だけを格納できる平衡3分木のデータ構造を考案する。本論文では，このような改良を実装した平衡3分木を「改良平衡3分木」と呼ぶことにする。各節点にはキー1文字だけを格納できるようにして，残りの文字列を必要な分だけ領域確保した配列に格納して節点から参照できるようにする。そうすることで，データの整合性を保ったまま，領域量を削減することが可能となる。階層が s のときにキーは s 桁目であり，各節点が1文字を格納するので，確保する配列の領域量は $S-s$ 文字分（つまり $S-s$ バイト）となる。今後は，$S-s$ を mem(s) と表現する。

改良平衡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。

```
1   enum  sub {RO, LT, RT, CT};
2   enum  diff {EVEN, LEFT, RIGHT};
3   struct  NODE{
4         char  key;                    // キー1文字を格納
5         char  *data ;                 // 残りの文字列を指すポインタ
6         enum sub  sub_tree ;          // 親からみた部分木としての方向
7         enum diff  diff ;             // 左右部分木の高い方向
8         struct NODE  *left ;          // 左の子を指すポインタ
9         struct NODE  *right ;         // 右の子を指すポインタ
10        struct NODE  *center ;        // 中央の子を指すポインタ
11        struct NODE  *parent ;        // 親節点を指すポインタ
12  }
```

4行目にキー1文字を格納し，5行目のポインタで残りの文字列を格納した配列を指す。ラベルならばキー1文字しか参照しないので，ポインタが NULL ならばラベルである。6, 7行目は節点の状態を判断するための情報である。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 ならば中央部分木である。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。8から11行目は各部分木と親節点を指すポインタである。6から11行目は平衡3分木と同様である。一節点の領域量は29バイトであり，必要に応じて mem(s) バイトを確保することになる。

図5は図3と同じデータを登録した改良平衡3分木である。四角は配列であり，配列を指していない節点

がラベルである。図3と比較すると，同型の構造であることが確認できる。節点が16個であり，確保した領域が22バイトなので，木構造全体では$29 \times 16 + 22 = 486$バイトの領域を使う。次節でこの改良平衡3分木の操作方法を説明する。

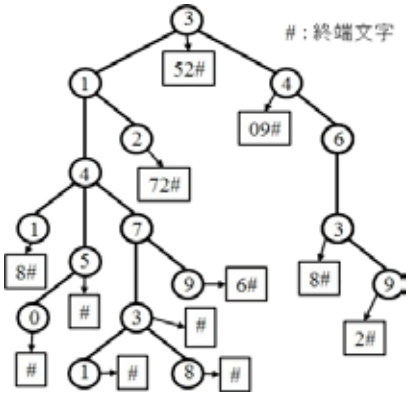

図5　改良平衡3分木Ｔの例

3. 基本操作アルゴリズム

まず，目的の節点を探す探索アルゴリズムを述べる。平衡3分木の場合と同様に，探索文字列$a_0 a_1 \ldots a_S$を用意し，全順序に従って，着目している節点pのキー key と大小比較を行い，結果に従って着目節点を移動させる。$S-1$桁目までの一致で探索成功とする。a_sと key が一致したが節点にT_cが存在しない場合，残りの探索文字列$a_{s+1} \ldots a_S$を節点の指す配列の文字列$b'_0 \ldots b'_{\mathrm{mem}(s)-1}$の先頭から比較し，一致した場合には次の桁を比較し，不一致の場合は探索失敗とする。

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 key 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 key 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T_cの根を指す。
　　　　3.3.2.1.2　sに1加算する。（3に戻る）
　　　3.3.2.2　節点pが中央部分木を持たない場合，
　　　　3.3.2.2.1　比較桁 i を0文字目に初期化する。
　　　　3.3.2.2.2　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）

ループ A は次のようになる。ループ A を実行する状況とは，a_s = key だが$s \neq S-1$であり，節点pが中央部分木のない通常節点である場合が該当する。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。

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'_i 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，i に1加算する。（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，key =‘3’ で a_0 < key 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 = key なので，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。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 = key となったので，ループ A を実行し残りの桁を配列の文字列と比較する。$a_1 = b'_0$ であるが $a_2 \neq b'_1$ なので，“359” は木構造 T に登録されていないとわかる。

次に，文字列 $a_0 a_1 \ldots a_S$ を木構造に登録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キー key と大小比較を行い，結果に従って着目節点を移動させる。a_s を比較して登録されていないことが判明したら，a_s をキーとするを持つ節点を辞書式順序を満たすように挿入する。配列と比較して食い違いがあった場合には，配列との比較回数だけ中央部分木に節点を作成する。節点作成の際には文字列 $d_s \ldots d_S$ と列挙型 sub と節点 p を参考に，表2の情報を格納した節点を作成する。*data には，ラベルの場合は NULL を，通常節点の場合は $d_{s+1} \ldots d_S$ を格納した配列のアドレスを格納する。

表2　作成する節点の情報

変数	key	*data	sub_tree	diff	*left	*right	*center	*parent
値	d_s		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 で木構造の根を指す。
4　ポインタ p を NULL とする。
5　*t が空木でない限り，a_s と key 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 key 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　カウンタ k を0で初期化する。

5.3.2.2 i = 0を mem(s)−1まで増加させながら，次のことを繰り返す。
5.3.2.2.1 k に1加算する。
5.3.2.2.2 a_{s+k} と b'_i を比較し，不一致ならば繰り返しから抜ける。
5.3.2.2.3 5.3.2.2に戻る。
5.3.2.3 繰り返しを回りきったならば，登録済み。（終了）
5.3.2.4 sub に CT を記録する。
5.3.2.5 次を k 回繰り返す。（終われば5に戻る）。
5.3.2.5.1 ポインタ p で *t の行き先の節点を指す。
5.3.2.5.2 *t の行き先の節点が指す $b'_0 \ldots b'_{\mathrm{mem}(s)-1}$ を格納する節点を，T_c に作成する。
5.3.2.5.3 *t の行き先の節点が指す配列を解放して，*data を NULL とする。
5.3.2.5.4 ポインタ *t で T_c を指す。（5.3.2.5に戻る）

6 $s \neq 0$ならば，i = 0を mem(s) まで増加させながら，次のことを繰り返し文字列 $a_s \ldots a_S$ を作成する。
6.1 a_i を a_{s+i} に書き換える。（6に戻る）

7 登録可能であり，$a_s \ldots a_S$ 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。階層0の‘4’をキーとする節点 p に着目し，a_0 は キーと一致した。節点 p は中央部分木を持たないので，残りの桁を比較して $a_2 \neq b'_1$ 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，中央部分木にキー‘0’のラベル p’と，キー‘9’の通常節点 p”が作られた。最後に大小比較を行い，$a_2 <$ key なので節点 p” に左部分木に節点を作成し“405”を登録する。図6は挿入後の木構造 T である。

図6 節点挿入後の木構造 T

次に，目的の文字列を木構造から削除するアルゴリズムを述べる。目的の文字列 $a_0 a_1 \ldots a_S$ を探す工程と実際に節点を削除する工程に分けられるが，目的の文字列を探す工程は探索アルゴリズムと同じであり，探索成功で削除対象節点 q が見つかったとする。削除する工程では平衡3分木の場合と同様に，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，削除対象節点を 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）p に移し，p が通常節点であれば節点 q の文字列を節点 p の文字列に書き換えて，節点 p を削除する。図7, 8は節点削除の一例であり，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。削除した節点と配列の領域は解放する。節点 p も部分木を持っていた場合，削除対象節点をさらに移すことになる。

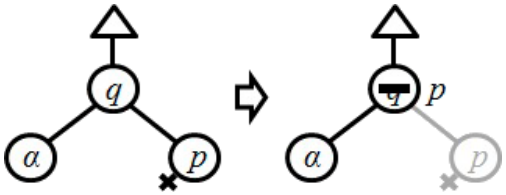

図7　節点削除の例1

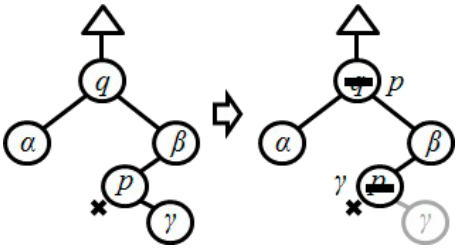

図8　節点削除の例2

しかし，削除対象節点 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9, 10は上記のような状態と削除後の様子である。以上を考慮した「節点削除アルゴリズム」を次に示す。

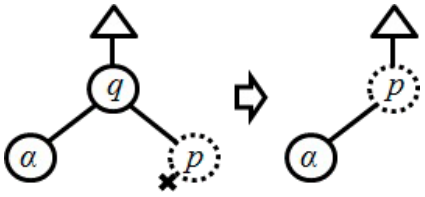

図9　節点削除の例3

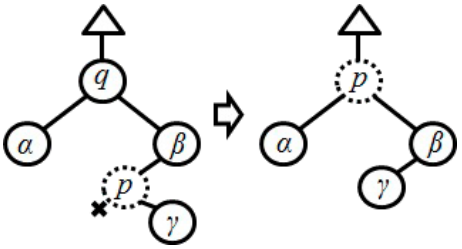

図10　節点削除の例4

節点削除アルゴリズム

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 のキーを節点 p の key に書き換える。
　2.4　節点 q の指す配列を節点 p の指す $b'_0 \ldots b'_{\mathrm{mem}(s)-1}$ に書き換える。
　2.5　$*t$ の行き先の節点に対して再帰実行する。
　2.6　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 のキーを節点 p の key に書き換える。
3.6 節点 q の指す配列を節点 p の指す $b'_0 \ldots b'_{\mathrm{mem}(s)-1}$ に書き換える。
3.7 *t の行き先の節点に対して再帰実行する。
3.8 節点 q の親節点は，そのまま節点 q を部分木とする。（終了）

改良平衡3分木でも平衡3分木と同様に，削除後に最適化操作を行う必要がある。最適化操作とは節点を削除して AVL 木の性質を考察した後に行う操作であり，節点を用意する必要のない余分な節点を削除することで木構造の節点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する操作である。ここで削除する余分な節点 p とは，親節点の T_c の根であり，節点 p の T_l, T_r, T_c が全て空木である節点のことである。節点 p の文字列を全て親節点の指す配列に格納してから節点 p を削除することで，登録文字列の整合性を保ったまま節点数を減らすことができる。この操作を行うアルゴリズムを以下に示す。引数として節点 p と階層 s を用意する。また文字列を一度保存する配列 stock[S] を，このアルゴリズムでは用いている。

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 stock[0] に節点 p の key を保存する。
3 i = 0を mem(s)−1まで増加させながら，次のことを繰り返す。
3.1 stock[i+1] に節点 p の指す配列の b'_i を保存する。（3に戻る）
4 ポインタ p で節点 p の親節点を指す。
5 s を1減少させる。
6 節点 p は，T_c を空木とする。
7 節点 p の *data に mem(s) 文字分の領域を確保する。
8 i = 0を mem(s)−1まで増加させながら，次のことを繰り返す。
8.1 stock[i] を節点 p の指す配列の b'_i に格納する。（8に戻る）
9 節点 p に対して再帰実行する。（終了）

例として図6の木構造 T から文字列 “409” を削除する。探索の結果，階層2の ‘9’ をキーとする節点 p” を最初の削除対象節点とする。節点 p” は左部分木にキー ‘5’ の節点を持つので，この節点に削除対象節点を移す。この節点は子を持たないので，節点 p” のキーを ‘5’ に書き換えて，この節点を削除する。図11はここまでの操作を反映させた木構造 T であり，この時点で木構造から文字列 “409” は削除されていることになる。

図11　“409” を削除した図6の木構造 T

図11の木構造 T の節点 p” に最適化を行う。キー ‘5’ の節点 p” は中央部分木である必要のない節点なので，キーである ‘5’ と終端文字 ‘#’ を2文字分確保した配列に格納して，‘0’ をキーとする親節点 p’ でその配列を指して節点 p” を削除する。こうすることで余分な節点 p” を削除することができた。次に節点 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 ‘0’，‘5’，‘#’ を格納した配列を親節点 p で指して節点 p’ を削除する。節点 p は中央部分木ではないので，ここで最適化を終える。図12は最適化を実行した後の木構造 T である。最適化によって，登録文字列の整合性を保ったまま節点の個数を必要最小限にできた。

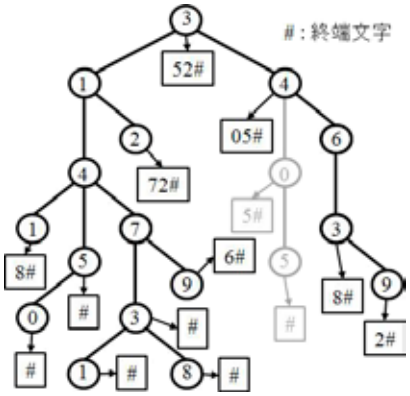

図12　最適化を行った木構造 T

4. 数値実験による評価

平衡3分木と改良平衡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する。以下の項目を比較項目とする。平衡3分木の領域量は一節点の領域量×節点数で計算するが，改良平衡3分木の場合は各節点が指す配列の総容量を加えたものを領域量とする。

- 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 と格納文字列を比較した回数。

* CPU Intel Core i5 2.5GHz, OS Windows 7 Professional, GCC 4.8.2

- 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）。

(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3のようになった。

表3　木構造構築の資源

	実行時間	比較回数	節点数	領域量
平衡3分木 (a)	21.76	200,568,387	14,276,243	1,841,635,334
改良平衡3分木 (b)	25.63	200,568,387	14,276,243	1,346,628,184
(b) / (a)	137.8%	100.0%	100.0%	73.1%

改良平衡3分木は平衡3分木と同型の木構造なので，節点数や比較回数は同数である。しかし，改良平衡3分木の方が領域の確保などの処理が多くなるので実行時間を多く必要とする。一方，各節点に必要最小限だけの領域を確保することで，領域量を約27% 抑えることができた。

(2) 木構造の撤去について

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，登録した文字列に対して格納順に全て削除操作を実行する。結果は表4のようになった。

表4　木構造撤去の資源

	実行時間	比較回数	節点数	領域量
平衡3分木 (a)	23.57	1,121,719,378	0	0
改良平衡3分木 (b)	28.19	1,121,719,378	0	0
(b) / (a)	119.6%	100.0%		

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。撤去の場合も処理の多さから実行時間を多く要している。また，10,000,000回のデータ削除操作で全ての節点を完全に削除できることも確認できた。

(3) データ探索の速さについて

木構造で一般的なデータ探索を行う速さを調べる。構築された木構造に対して，既に登録された一千万個のデータ（既登録データ）をすべて探索した場合と，別に発生させた一千万個の乱数データ（未登録データ）をすべて探索した場合を考える。既登録データを用いた場合は全て探索成功となるが，未登録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々小さいので，大抵全て探索失敗となる。数値実験の結果は表5のようになった。

表5　データ探索の速さ（実行時間（sec））

	既登録データ	未登録データ
平衡3分木 (a)	18.28	15.87
改良平衡3分木 (b)	16.87	14.34
(b) / (a)	92.3%	90.3%

結果として，どちらのデータに対しても改良平衡3分木の方が速く探索できることが分かった。平衡3分木の各節点は S+1桁の配列を持っており，比較の際にはアルゴリズム内で管理していた比較桁 s の値に

従って間接的に文字を参照していた。しかし改良平衡3分木の場合は各節点がキー1文字だけを独立して持っているため，比較の際には直接的に文字を参照することができていた。このことが改良平衡3分木の方が速く探索できた理由として挙げられる。

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」の一つである平衡3分木の改良を提案し，これに対して様々な考察を行った。

改良平衡3分木は平衡3分木と同型の構造であるが，全体の領域量をできるだけ削減することを目指した木構造である。平衡3分木の各節点のキーとなる桁は階層によって決まるという性質があり，改良平衡3分木ではキー1文字を各節点が格納することで参照しない文字を削減した。また必要に応じて残りの文字列を格納する配列を用意することで，ラベルか否かの判別を行い，節点から必要最小限の文字だけを参照できるようにした。平衡3分木と同型なので，同様に探索を行う事も可能であり，挿入・削除操作も最適化を含め同様に可能である。

数値実験により平衡3分木比較して，木構造の構築・撤去の実行時間は増えたが，構築後の領域量を約73% に，また完全撤去が可能であることを確認できた。またデータ探索において，既登録データで約92%，未登録データで約90% の実行時間で探索可能であり，一般的な探索操作では平衡3分木よりも速く探索可能であることが判明した。

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 [18,19]）との比較，自由な長さの文字列を扱える探索木の提案，実用性のある「ひらがな」など10進数以外や全角文字への拡張などがあげられる。

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] 新森修一，永福俊也，礒道隆一．平衡性を持つ3分木構造の提案とその評価．鹿児島大学理学部紀要．Vol.47, pp.1–12, 2014.

[7] 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.

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

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

[10] Ryozo NAKAMURA, Akio TADA, and Tsuyoshi ITOKAWA, An Analysis of the AVL Balanced Tree Insertion Algorithm, IEICE transactions on fundamentals of electronics, communications and computer sciences, Vol.86, pp.1067–1074, 2003.

[11] 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.
[12] Wojciech Rytter, Application of Lempel-Ziv Factorization To The Approximation of Grammar-Based Commpression, Theoretical Computer Science, Vol.302, pp.211–222, 2003.
[13] David Wood, Kowari: A Platform for Semantic Web Storage and Analysis, In Xtech 2005 Conference, pp.05–0402, 2005.
[14] 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.
[15] 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.
[16] Rudolf Bayerand E.McCreight. Organization and maintenance of large ordered indexes, pp.245–262. Springer-Verlag New York, Inc., NewYork, NY, USA, 2002.
[17] Robert Sedgewick. Algorithm in C Third Edition.Addison-Wesley, Pearson Education, Inc, 2004. 野下浩平，星守，佐藤創，田口東訳：近代科学社
[18] 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.
[19] 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．情報処理学会研究報告．情報学基礎研究会報告，Vol.2007, No.34, pp.41–46, 2007-03-27