AVL木の拡張とB木との比較評価

Rep. Fac. Sci., Kagoshima Univ.,
No. 44, pp. 15–26 (2011)

AVL 木の拡張と B 木との比較評価
Evaluation for Comparison of Extended AVL Trees and B-Trees

竹之下　朗・新森修一 *
Akira TAKENOSHITA, Shuichi SHINMORI

Abstract. In this paper, we propose to develop an extended AVL tree with 5-subtrees, for the purpose of increasing search efficiency, and examine various evaluations for the extended AVL tree. The data structure of this extended AVL tree contains 5 partially balanced subtrees that match prefixes character by character by implementing the radix search method. A numerical experiment confirmed that the construction time was about 50% of B-tree. When the height of B-trees is smallest, the amount of memory becomes smaller than that of B-trees, using 10^{14} pieces of data or more. The construction time of the extended AVL trees was about 47% of that of B-trees in a numerical experiment using 10 million random pieces with 100-digit character strings in the decimal number, and the comparison frequency of the extended AVL trees obtained an excellent result of about 11% of that of B-trees. In this case, the amount of memory became about 36% for that of B-trees.

Keyword. data structure, 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効率的な探索については，ダブル配列 [8] を始めとするアルゴリズムの開発と改良 [11] が行われている．本研究では 2 分探索木である AVL 木 [1] を文字列探索に適応した 5 分木構造のデータ構造と関連するアルゴリズムなどを提案した [3, 4]．2 分探索木である AVL 木は，データ数が n のときに平衡化することで最悪の場合でも探索 · 挿入 · 削除などの各操作が $O(\log n)$ となる木である．AVL 木の性質を他の分野に応用 [13] したものは多いが，文字列検索に応用した例は殆ど無く，この文字列探索に適した AVL 木を「拡張した AVL 木」と名づけ，次を満足する木とする．

1. 語，句または単語を探索できるような文字列の探索を可能とする．
2. 効率的な文字列の探索をするために，木構造において平衡化を行う．
3. 探索を高速に行うために，既存の AVL 木を多分木に拡張したデータ構造を構築する．
4. 構築する木構造は，リスト構造を基本とする．
5. 動的な処理に対応できるデータ構造を構築する．

B 木（k 分木）[10] は，AVL 木と同様に平衡木であり各節点は高々 k 個のデータを挿入できる木である．B 木は，ベイヤー（R. Bayer）とマクレイト（E. McCreight）によって名付けられ，節点は最大 $k(\geq 2)$

* 鹿児島大学大学院理工学研究科
Graduate School of Science and Engineering, Kagoshima University

個の子をもつことができることから，マルチウェイ平衡木（multi-way balanced tree）と呼ばれる．B 木はディスク装置上のファイルを探索するなどの外部探索に適したデータ構造である [5].

以下，2 節では拡張した AVL 木の定義と基本的な性質を述べる．提案する木構造は，動的な辞書を作成できるように 5 分木に拡張した AVL 木であり，リスト構造で平衡性を一部満たす木である．3 節では B 木の定義と基本的な性質を述べる．B 木は，計算機の記憶領域で使われるなど，実用的に使われている木である．4 節では，拡張した AVL 木と B 木に対する数値結果とその考察について述べる．100 桁のランダムな数値 10,000,000 個を乱数を用いて発生させ，構築時間，メモリ使用量と比較回数を比較している．5 節で本稿の結果をまとめ，今後の課題について述べる．

2　5 分木に拡張した AVL 木

文字列の探索には，トライ法 [6] を出発点とする様々な探索法があり，データ構造の代表的な実現方法には配列構造とリスト構造がある．一般的に，配列構造での問題点は要素の疎状態の回避と動的な処理は不得意と考えられる．一方，リスト構造での問題点は，配列構造の問題点は解消されるが，枝の数が増加することで検索効率が低下すると考えられる．そこで，文字列対応かつ挿入や削除などの動的な処理が可能となる 2 分木の AVL 木を考えてみる．この木構造では，ある文字列を探索中にある節点に到達した場合，比較のためにある程度進めた桁（添字）を右または左部分木へ進む度に，再度初めの桁から比較を行わなければならない．図 1 において，節点 A と B の文字の大小比較ができる添字を覚えて記憶する変数を準備しておけばよいが，動的な処理を繰り返し行う都度この変数を更新しなければならず現実的ではない．例えば，この図では節点 A と B の大小比較ができる節点は，添字 3 であるから節点 B においては添字 3 を記憶しておく変数を用意しておく．このようにすれば，節点 B において添字 2 までは，調べなくてよく効率的な探索ができる．つまり，節点 B では親節点 A に対してのみ，探索しなくてよい添字を確定できるが，削除などにより節点を移動すれば親節点の関係がくずれ，変数を更新しなければならない．図 1 は基数探索法の考えをそのまま実装した木 T を示しているが，文字列 *str* を探索してみる．図に示すように節点 A, B にはそれぞれデータが既に格納されている．文字列 *str* と節点 A では添字 2 まで一致するが，*str* の添字 3 である ‘*p*’ が節点 A の添字 3 である ‘*o*’ より大きいので，ここでは右部分木へと進行する．次の節点 B においては添字 0 から再走査することになり，添字 3 まで比較したものを添字 0 に戻すことで探索時間に無駄が生じることになる．なお，図 1 の ‘$’ は，終端記号である．

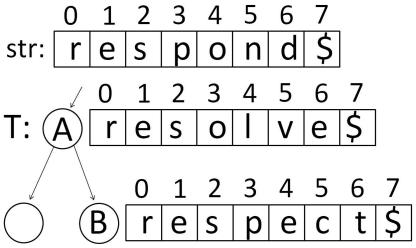

図 1　データを文字列にした AVL 木

既存の AVL 木を拡張して，節点に進行する度にデータの先頭から比較する無駄を除く，すなわち，添字を戻さない新たな平衡木を構築する方法を提案する．1 節点当たり最大 2 桁まで比較することにして，

データの比較において，添字を戻さないデータ構造とする．添字を戻さないために，左部分木，右部分木，前部分木，中央部分木，後部分木への 5 つのポインタを準備する．この提案する AVL 木を「5 分木に拡張した AVL 木」と名付け，節点の構造を図 2 に示す．この構造は，ある変数に添字を格納させるのではなく，5 分木構造とすることで添字を戻さない効率の良さを保持している．この構造ならば，部分木や節点を木構造内の矛盾した位置に移動しない限り，構造は保たれることになる．

図 2 を用いて，探索のアルゴリズムを説明する．木 T の各節点に格納されているデータを「節点データ」，探索するデータを「探索データ」と呼ぶことにする．木 T の根から探索をはじめ，根節点 A の 1 桁目で大小の比較ができた場合は左部分木または右部分木へ進むことは既存の AVL 木と同様である．すなわち，探索データの 1 桁目の値が節点データのそれよりも小さいならば左部分木へ，大きい値ならば右部分木へ進み，同じ桁から探索を行う．1 桁目の値が同じならば，探索データの 2 桁目の値と節点 A のそれとの大小を比較する．探索データの桁の値が小さいならば前部分木に進み，大きい値ならば後部分木に進み，同じ桁から探索を行う．もし，この桁の値が等しいならば，中央部分木へ進み次の桁から探索を行い，中央部分木がない場合は，節点を移動せずに次の桁から探索を行う．つまり，節点 A において，左または右部分木に進行した場合は比較を行う桁が移動しない（0 桁移動），前または後部分木に進行した場合は 1 桁移動，中央部分木に進行した場合は 2 桁移動することになる．この移動する桁数を木 T の節点に示す．以上を繰り返して，進行すべき子節点がなくなれば，探索は失敗となる．この構造では，比較する桁値を戻すことなく，1 節点当たり最大 2 桁まで比較している．これは，基数探索法から部分木に「先頭から同文字列のデータ」が集まることを意味しており，それらを探索しやすくなる．すなわち，先頭が検索語に一致する前方一致 (prefix search) に適した構造といえる．

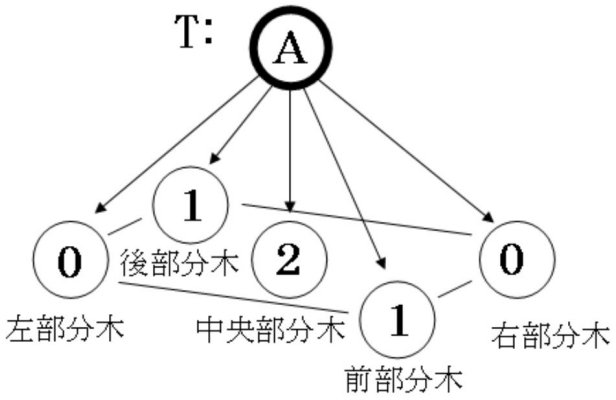

図 2　5 分木に拡張した AVL 木の節点の構造

あるデータを挿入するとき，まず挿入するデータの探索を行い，探索が失敗であれば挿入を行う．挿入の手続きにおいて，中央部分木にデータを追加するときには，木を利用したソート[*1]を実現するためにラベル付けの方法を導入している．A の中央部分木に節点を追加する場合は，A にラベルを付加したものを A' とし，A' にラベルを付加していない A を結ぶ．これによって，A' はデータではなくなるが，A は A' の中央部分木となることで，この部分木においてソート可能となる．

[*1] 集合の全ての要素を全順序にしたがって並べること．整列あるいはソーティングともいう．

図 3 の簡単な例を用いて，挿入の手続きを述べる．T の根に ‘*NEW*’ のみが挿入されていると仮定して，‘*BIG*’，‘*OLD*’，‘*NAS*’，‘*NOW*’，‘*NEE*’，‘*NEX*’，‘*NEW*’ の 7 単語を挿入する．7 単語はすべて大文字のアルファベットであり，‘A’ を最も小さい値，‘Z’ を最も大きな値とする．*BIG* を挿入する場合は，*NEW* の 1 桁目を比較して *B* が *N* より小さいので，左部分木に進行する．左部分木がないので，そこへ *BIG* を挿入する．*OLD* の場合は，*NEW* の右部分木に進行して同様の操作を行う．*NAS* を挿入する場合は，*N* が同じ文字なので 2 桁目を比較して *A* が *E* より小さいので，前部分木に進行する．前部分木がないので *NAS* を挿入する．*NOW* の場合は，*NEW* の後部分木に進行して同様の操作を行う．*NEE* を挿入する場合は，1・2 桁目とも同じ文字なので根節点をラベル付き節点とする．図 3 では，ラベル付き節点からのポインタを点線で表し，根節点をラベル付き節点であることを示すため太い円で囲っている．根節点の中央部分木にデータの *NEW* を挿入して，3 桁目を比較して *E* が *W* より小さいので，左部分木に進行する．左部分木がないので *NEE* を挿入する．*NEX* を挿入する場合は，1・2 桁目とも同じ文字より根節点の中央部分木に進行する．3 桁目を比較して *X* が *W* より大きいので，右部分木に進行する．右部分木がないので *NEX* を挿入する．最後に *NEW* を挿入する．1・2 桁目とも同じ文字より根節点の中央部分木に進行する．3 桁目を比較するが *W* が同じ値で前または後部分木が対象となるが，文字列の終点より同一の文字列ということが判明し終了する．なお，図 3 では，根節点から比較桁の差を ‘-digit’ で示している．

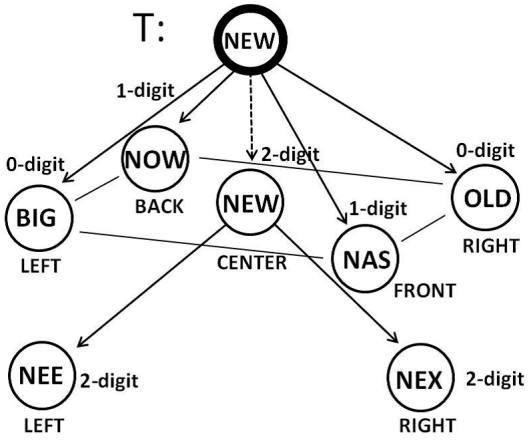

図 3　拡張した AVL 木におけるデータの挿入例

データを挿入後，拡張した AVL 木の条件を満たしているかを考察する．新しい葉から根に向かって既存の AVL 木と同様に「左部分木と右部分木の高さの差が高々 1 しか違わない」かを考察する．拡張した AVL 木の条件を満たしていなければ，回転を行うことで拡張した AVL 木の条件を満たすことになる．回転の種類には，一重回転と二重回転があり，それぞれを図 4 と図 5 に示す．図 4(a) では，新しい葉が部分木の *A* にできて，*v* の左部分木の高さが 1 高くなっている．図 5(a) では，新しい葉が部分木の *B* にできて，*w* の左部分木の高さが 1 高くなっている．どちらの図でも，(a) 回転前では，*u* の左部分木の方が右部分木よりも高さが 2 高いが，(b) 回転後では，部分木の根では左部分木と右部分木の高さの差はない．同様に，右部分木の方が左部分木よりも高い場合も対称的に扱える．

この 5 分木に拡張した AVL 木の定義は次のようになる．

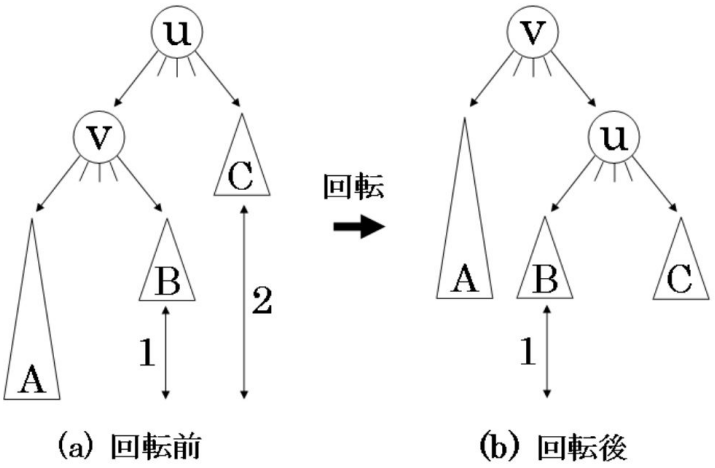

図 4　拡張した AVL 木の一重回転

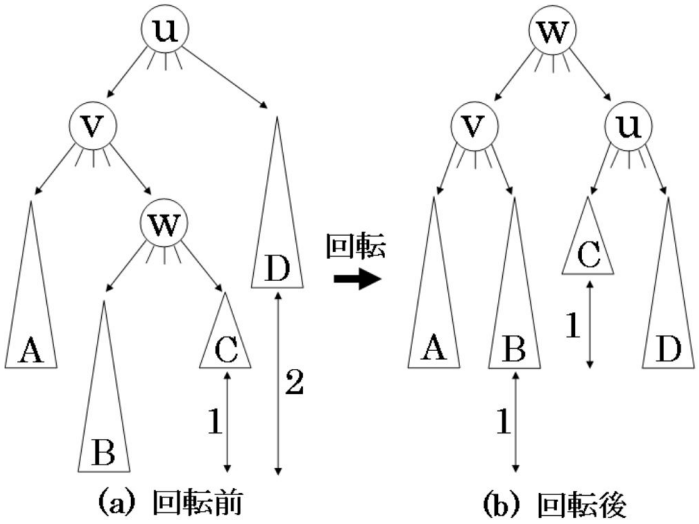

図 5　拡張した AVL 木の二重回転

定義 2.1. 拡張した AVL 木とは，次の条件を満たす 5 分木である．

(1)　n 桁のデータ $a_0, a_1, \cdots, a_{n-1}$ をもつ節点 u と u の 5 つの部分木である T_1 ～ T_5 からなる．5 つの部分木は，左部分木 (T_1)，前部分木 (T_2)，中央部分木 (T_3)，前部分木 (T_4)，後部分木 (T_5) とする．ただし，T_1～T_5 は，まったく節点を持たない空木となることもある．

(2)　節点 u のデータの添字 $i\,(i \geq 0)$ をキー a_i としたとき，T_1 の添字 i のキーはすべて a_i より小さく，T_2～T_4 の添字 i のキーはすべて a_i と等しく，T_5 の添字 i のキーはすべて a_i より大きい．添字 i のキーが同じ値ならば，$i+1$ 桁目の値で大小の比較を行い，T_2 の添字 $i+1$ のキーはすべて a_{i+1} より小さく，T_4 の添字 $i+1$ のキーはすべて a_{i+1} より大きい．T_3 の添字 $i+1$ のキーはすべて a_{i+1} と等しい．

(3) T_3 が空木とならない場合，T_3 の親 u はラベルとし，T_3 の根に節点 u のデータを保持させる．ラベルは添字 i と $i+1$ のキー a_i と a_{i+1} をもつ．

(4) 部分木の深さとは，T_1 と T_5 のみであり，T_2, T_3, T_4 の深さを含めない．このとき，各部分木の根に対して，T_1 と T_5 の深さの差は高々 1 である．□

上の定義（3）において，ラベルとは構造上はデータをもつ節点と同じであるが，データ節点ではない識別用の節点である．節点に中央部分木ができた場合のみ，節点がデータではないラベルとなる．これによって，中央部分木でのソートが可能となる．（4）において，「部分木の深さとは，左と右部分木のみであり，前，中央，後部分木の深さを含めない.」としているのは，前，中央，後部分木に進行した時点で比較する桁が移動してしまい，ここで平衡化を行ってしまうと構造が崩れてしまうからである．この例を図 6 に示す．図では，節点 A の前部分木（節点 B）と後部分木（節点ない）の高さの差が 2 となり，

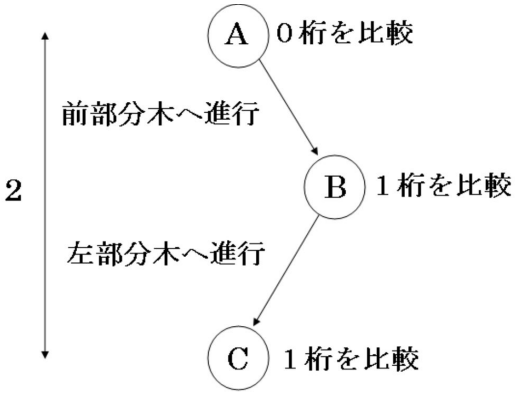

図 6　前，中央，後部分木の深さを含めない一例

仮に平衡を行なうと節点 B と C では 1 桁の値が木に反映してしまい構造が崩れる．この T では，前部分木と後部分木だけに着目すると，平衡化を行っていないために 2 分探索木と同じ状態となる．

拡張した AVL 木のデータ構造は，プログラミング言語 C により実現しており，拡張した AVL 木のデータ構造を次に示す．

5 分木に拡張した AVL 木のデータ構造

```
1 struct ext_avl {
2   char element[S+4];
3   struct ext_avl *left;
4   struct ext_avl *front;
5   struct ext_avl *center;
6   struct ext_avl *back;
7   struct ext_avl *right;
8   struct ext_avl *parent;
9 };
```

2 行目で S 桁の文字列（S バイト）と終端記号（1 バイト）を格納し，残る 3 バイトは平衡条件の情報を格納する．3 バイトの内訳は次の通りである．

- 節点の種類（ラベルまたはデータ）
- 部分木の種類（根，左，右，前、後，中央部分木）
- 左右の部分木の状態（平衡状態，左，右部分木が重い）

3 行目から 7 行目は各節点へのポインタであり，8 行目は親節点へのポインタである．ポインタの領域を 4 バイトとすると，1 節点あたりの領域量は $S + 28$ バイトである．

3 B 木

比較に使用した B 木のデータ構造を次に示す．このデータ構造は，参考文献 [9] を参考にしており，文字列に対応した 5 分木の B 木となっている．B 木の節点は 11 行目以下で構成されており，k 個の配列とキーの個数を保持する変数 m で構成される．k 個の配列の数が定義 3.1 の k 次とリンク数の k にあたる．m はキーの個数を記憶し，k より大きくなったら，節点の分割を行う．2 行目以下は，この配列の構造であり，内部節点の場合は 4 行目のキーと 6 行目の節点へのリンクで構成し，外部節点の場合は 4 行目のキーと 7 行目の文字列データで構成する．

```
 1 typedef struct STnode* link;
 2 // 配列の構造
 3 typedef struct {
 4   char key[S+1];
 5   union {
 6     link next;
 7     char element[S+1];
 8   }ref;
 9 }entry;
10
11 // 節点の構造
12 struct STnode {
13   entry b[k+1];
14   int m;
15 };
```

一般に k は奇数とし，B 木の定義は次のようになる．

定義 3.1. k 次の B 木は，空か k– 節点からなる木である．k– 節点は k 個のキーをもち，それらのキーで区切られる k 個の区間のそれぞれを表す部分木への k 個のリンクをもつ．B 木とは次の構造的条件を満たす k 部分木である．

(1) 根では，要素の数は 1 以上 k 以下でなければならない．
(2) 他の節点では，要素の数は $(k+1)/2$ 以上 k 以下でなければならない．
(3) 空な木へのリンクはすべて根から同じ距離でなければならない． □

定義 3.1 の k は奇数であるが，13 行目の $k+1$ 個（偶数個）の配列 b を用意する．配列 b の $k+1$ 番目のレコードは分割に使う操作用レコードである．図 7 は，このデータ構造を視覚化したものである（分割に使用する操作用レコードは省いている）．図は 5 分木の B 木であり，1 節点を 5 つのレコード（配列）で表し，内部節点の場合はキーとリンクの配列，外部節点の場合はキーと文字列のデータの配列である．5 次の B 木は，内部節点では $6(S+5)+4$ バイトであり，外部節点では $12(S+1)+4$ バイトである．定義 3.1 より 5 分木の場合は，根では 1〜5 個のキー数であり，他の節点では 3〜5 個のキー数となる．探索は，木の根から行い内部節点ではリンクを辿り，外部節点でキーを挿入するプログラムとなる．外部節点に要素を挿入したときに，要素の数が 6 個以上になれば，節点の分割を行う．節点の分割が行われれば，新しい節点へのリンクが親節点に必要になるが，このリンク数が 6 個以上になれば更に分割

が必要となる．この操作を下階層から上階層へと分割を行っていき，根で分割が行われたときに高さが 1 高くなる．

図 7 は，根節点に文字列のキーとデータ「0, 10, 100, 1000, 10000」が挿入された図である．6 番目のレコードは，分割に使う操作用レコードである．図 7 に文字列データ ‘150’ を探索失敗後に挿入すると，外部節点が分割して図 8 となる．根で分割したときに高さが 1 高くなる．

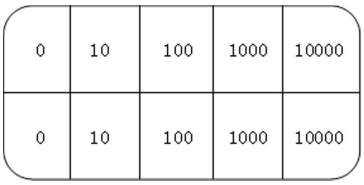

図 7　根節点におけるキーの満杯の様子

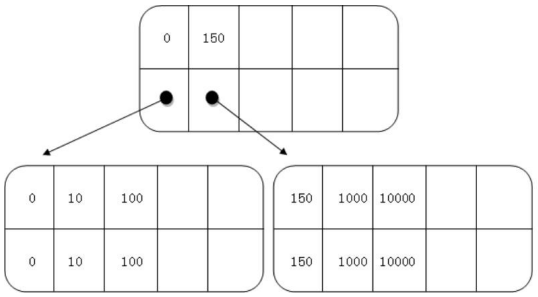

図 8　高さの増加（根節点の分割）

定義 3.1 の「空な木へのリンクはすべて根から同じ距離でなければならない」より，B 木はつねに平衡を保っていることになる．むしろ，根から葉節点までの距離がすべて同じであるから，高さの観点からいえば AVL 木よりバランスがよいといえる．

4　拡張した AVL 木と B 木との比較

4.1 で「構築時間」と「探索時間」を，4.2 で「領域量」の比較を行う．

4.1　時間計算量の比較

拡張した AVL 木と B 木に対して，幾つかの項目を設定して比較を行う．数値実験で使用するデータの条件等を箇条書きで以下に示す[*2]．

- データは，0〜9 までの 10 通りの文字とする（$m = 10$）．
- データの長さ（桁数）は，100 桁とする（$S = 100$）．
- データの数は，10,000,000 個とする．（$n = 10,000,000$）
- データは，現在の時刻で擬似乱数を発生させ，データはファイルに書き込んだものを用いる．なお，精度は 2^{32} である．
- 上述のデータを乱数で 10 回ずつ生成し，実験結果の平均を表 1 に示す．
- 各木においてデータは同一ものを使用している．
- 表 1 では，木の構築時間（秒），領域量（MB）と比較回数（回）を示している．
- 領域量は 1 節点当たりの理論値を用いている．

[*2] CPU Intel Core 2 Duo 2.4GHz, Memory 2GB, OS Mac OS X 10.5.2, GCC 4.0.1

表 1 と表 2 に，拡張した AVL 木と B 木の比較結果を載せる．表 1 と表 2 での比較項目（1）～（5）は以下の通りである．

（1）データを木に挿入するのに要した時間（秒）

与えられた全てのデータを各木構造（B 木，拡張した AVL 木）に挿入し，データ構造を構築するのに要した時間である．表 1 での構築時間に該当する．time コマンドの user 時間を記している．

（2）メモリ使用量（MB）

与えられた全てのデータを各木構造に挿入したとき，必要となるデータと構造の総メモリ容量である．表 1 での領域量に該当する．各木において，節点は構造体で表されているが，節点における構造体メンバの総メモリ容量を記している．

（3）木の構築における比較回数（回）

与えられた全てのデータを各木構造に挿入するときに要した比較の総回数である．各木においては，ある節点から次の節点に移動するときに，ある回数の比較が必要であり，この総和を記している．表 1 での比較回数に該当する．

（4）探索時間（秒）

表 1 で構築したそれぞれの木構造に対して，木に含まれないデータ 1,000,000 個を探索したときの時間を記している．表 2 での探索時間に該当する．

（5）探索における比較回数（回）

表 1 で構築したそれぞれの木構造に対して，木に含まれないデータ 1,000,000 個を探索したときの比較回数を記している．表 2 での比較回数に該当する．

表 1　拡張した AVL 木と B 木の構築時間

項目	拡張した AVL 木 (a)	B 木 (b)	比率 (a) / (b) (%)
構築時間	30.69	65.57	46.80
領域量	1334.96	3733.05	35.76
比較回数	208,085,583	1,901,987,367	10.94

表 1 より，拡張した AVL 木では先頭から後戻りすることなく文字列を 1 桁ずつ比較する構造のため，比較回数は B 木の約 11% になっている．比較回数は，構築時間に最も影響を与えるが，拡張した AVL 木の構築時間は B 木の約 47% であった．領域の効率性では、構築した B 木は高さ 12 から $n = 1,220,703,125$ 個の要素を格納できる木であり、かなり無駄な領域が生じている．これは，100 桁の文字列（10^{100} 個）から要素 10,000,000 個を選び出しているためである．拡張した AVL 木では，文字列の大小で木を構築 · 平衡するので，領域は要素数 n にほぼ比例する．これより，拡張した AVL 木の領域量は B 木の約 36% になっている．

次に，上記で構築した木に含まれないデータを $n = 1,000,000$ 個探索するのに要した時間（秒）と比較回数（回）を表 2 に示す．

表 2 の 5 分木に拡張した AVL 木と 5 分木の B 木について結果を考察する．数値結果は，拡張した AVL 木の探索時間は B 木の約 55% であり，また比較回数は約 1% と良好な結果が得られた．一方，B

表 2　各木の探索時間

項目	拡張した AVL 木 (*a*)	B 木 (*b*)	比率 (*a*) / (*b*) (%)
探索時間	2.91	5.29	55.01
比較回数	22,056,146	1,994,227,702	1.11

木に対する拡張した AVL 木の探索時間と構築時間の割合が 47% から 55% へと悪化しているように見受けられるが，これについて 2 つの木の各項目を検討する．B 木の比較回数は，データ数が 10,000,000 個から 1,000,000 個になっているにも関わらず，比較回数が増えている．単純な平均の探索比較回数は，1 個の探索データ当たり約 1994 回と非常に多い．これは，高さが 12 の木構造に対して，探索を行っているが，次の数値との比較はデータの先頭から再度始めることが一因である．さらに，今回の B 木のデータ構造は，データが木に含まれているかどうかは，葉まで到達しないと分からない構造であることにも起因している．一方，拡張した AVL 木の探索の比較回数は，挿入時の比較回数の約 11% と減少している．単純な平均の探索比較回数では，1 個の探索データ当たり約 22 回と非常に少ない．これより，拡張した AVL 木には多くの部分木があり，対象の部分木に早く到達する効率のよい探索になっていることが分かる．また，約 11% の比較回数を単純に 10 倍すると，探索時間は約 29 秒となり構築時間の 30.69 秒に近くなる．このように考えると，拡張した AVL 木の探索の比較回数と探索時間に対する構造的な説明が可能である．B 木は探索の比較回数が増大しているのにも関わらず，探索の時間は 9% となっている．これより，B 木は挿入時の木構造の変形にかなりの時間を費やすと考えられる．探索の比較回数の結果からも，拡張した AVL 木には多くの部分木があり，対象の部分木に早く到達する効率の良い探索になっていると考えられる．

4.2　領域計算量の比較

データ数を n，データの文字列長を m 進 S 桁とする．データを数値の $m = 10$ とするとデータ数 n は，$n = 10^S$ である．このデータ数 n で，拡張した AVL 木と B 木の領域量の比較を行う．

1 節点当たり $S + 28$ バイトとラベル数の割合は $m = 10$ の場合で 1.010%[3] であるので，5 分木に拡張した AVL 木の領域量は，

$$n(S + 28) + \frac{101}{10000} n(S + 28) \approx n \log n + 28n \tag{1}$$

となる (log は常用対数).

一方の B 木では，最も高さが小さくなる場合でバイト数を試算する．すなわち，各節点には 5 個のキーが挿入されると仮定する．このとき，深さ h では 5^h の節点数が存在して，かつ各節点には 5 個のキーが格納されていると仮定する．根から深さ $h-1$ までの葉以外のバイト数は，1 節点当たり $6(S + 5) + 4$ バイトであるので，

$$\frac{\{6(S + 5) + 4\}\left(5^h - 1\right)}{4} = \frac{n - 5}{10}(3 \log n + 17) \tag{2}$$

となる．葉の高さ h には，5^h 個の節点があり葉のバイト数は，1 節点当たり $12(S + 1) + 4$ バイトである

ので,

$$5^h\{12\,(S+1)+4\} = \frac{n}{5}\,(12\log n + 16) \tag{3}$$

となる．式 (2) と式 (3) より，B 木における節点のバイト数は,

$$(2) + (3) \approx 2.7n\log n + 4.9n \tag{4}$$

である．式 (4) と式 (1) の差は,

$$f\,(n) = 1.7n\log n - 23.1n \tag{5}$$

となる．式 (5) は増加関数であり，データ数 $n = 10^{14}$ 以上であれば 5 分木に拡張した AVL 木が B 木よりも領域は小さくなることがわかる．なお，試算したものは B 木の領域が一番小さくなる場合である．

5 まとめ

本論文では，木構造の探索操作における効率性の向上を目的として 5 分木に拡張した AVL 木を提案し，これに対して様々な考察や評価を行った．木構造を利用した探索は，ハッシュ法とともに多くの分野で利用されているが，キー値の順番を反映した構造であり，値の前後や共通接頭辞を指定した探索はハッシュ法より優れているといえる．5 分木に拡張した AVL 木は，2 分木の AVL 木を拡張したもので，2 分木の左と右部分木に加え，前・後・中央部分木を加えた 5 分木構造で前方一致となる文字列を部分木とする木構造となっている．データ構造は，文字列を配列で格納し，動的に挿入・削除できるようにリスト構造を採用し，平衡性を一部満たす木構造である．アルゴリズムには，トライやパトリシアがもつ基数探索法を利用して文字列の探索を可能としたものであり，文字列を 1 節点につき最大 2 文字までを比較することが可能である．拡張した AVL 木の平衡化は，左と右部分木の深さの差が 2 以上で行い，これによって比較回数を抑える．比較実験では，B 木の高さが最も小さくなる場合を仮定するとデータ数が 10^{14} 個以上で B 木よりも領域量は理論的に小さくなることが分かった．文字が 10 進数で文字列の長さが 100 桁のデータ数 10^7 個のランダムなデータに対する数値実験では，拡張した AVL 木は，先頭から後戻りすることなく文字列を 1 桁ずつ比較するため，データ探索において比較回数がかなり少なくなり，構築時間は B 木の約 47%，比較回数は約 11% でいう良好な結果が得られた．この場合の 5 分木に拡張した AVL 木の領域量は B 木の約 36% であった．今後の課題は，拡張した AVL 木の一般化と理論的性質の考察，削除の検証と改良，領域の削減とそのプログラム化，実際の環境下に合わせたひらがなや漢字のような文字種の数値実験，実用性の高い他のデータ構造（例えば，[7, 12]）との比較などである．

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. 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.

[3] 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.

[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] Douglas Cormer. The ubiquitous b-tree. *ACM Computing Surveys*, Vol. 11, pp. 121–137, 1979.

[6] Edward Fredkin. Trie memory. *Commun*. ACM, Vol. 3, pp. 490–499, September 1960.

[7] 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.

[8] 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. D , 情報・システム, Vol. 71, No. 9, pp. 1592–1600, 1988-09.

[9] 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

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

[11] 矢田晋, 大野将樹, 森田和宏, 泓田正雄, 吉成友子, 青江順一. 接頭辞ダブル配列における空間効率を低下させないキー削除法. 情報処理学会論文誌, Vol. 47, No. 6, pp. 1894–1902, 2006-06-15.

[12] 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

[13] 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. *Intelligent Information Hiding and Multimedia Signal Processing, International Conference on*, Vol. 0, pp. 298–303, 2008.