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

2011 年 3 月

竹之下 朗

目次

第 1 章

研究の背景とアルゴリズムの評価法

1.1 研究の背景と目的

現代社会において，コンピューターは私たちの生活に欠かせない存在である．日常生活で分からない語や句をインターネットで調べることは，コンピューターを使う身近な一例である．インターネットの環境下だけでなく，ファイルなど多数のデータから必要なデータを探し出すという操作は，最も頻繁に実行される操作の一つといえる．この例にように多数のデータの集合の中から，あるデータがこの集合に含まれているかどうかを判定する操作を**探索**（searching）という．本論文では，この探索方法の基礎を説明し効率の良い文字列探索法を提案する．集合に対する操作には，探索・挿入・削除等の操作が考えられるが，すべての操作の前に必要な基本操作が探索であり，効率の良い探索アルゴリズムが必要となる．探索は大きく分けて，**整列**と**探索**に大別できると考えられる．整列のアルゴリズムは，辞書式順序などのある規則に従ってデータを並び替える方法である．整列の代表的なアルゴリズムにはクイックソート [17] などがあり，これらは表を使ったアルゴリズムであり，並び替えるときに探索するものである．文献 [9] では整列と探索のアルゴリズムの関係が論じられている．しかし，整列は大規模なデータに対して頻繁に挿入・削除を行うには不向きである．コンピューター上で挿入や削除を頻繁に行うような動的な集合を構築する場合には，木構造が頻繁に利用される．本論文では，自由に挿入や削除ができるような動的な辞書に対応できるように，データの集合全体を木構造を用いて構築する．

最も基本的な木構造は 2 **分探索木**（binary search tree）であるが，データの挿入順序

によっては木の高さが大きくなる問題がある．これを解決する一方法として様々な**平衡木**（balanced tree）またはバランス木が提案されている．よく知られている平衡木には，AVL **木**[1]，B **木**[8]，**赤黒木**[7]，**スプレー木**[45] などがある．平衡木の中でも AVL 木は，アルゴリズムの入門書に必ず登場する木でありアルゴリズムも理解しやすいのが特徴である．しかし，AVL 木のアルゴリズム自体は理解しやすいが，詳細な振る舞いなど分かっていないことも多く様々な研究が現在も行われている．文献 [41, 44] では，AVL 木の構造に関して効率評価の研究をしている．元々の AVL 木は，削除や挿入の操作の直後に木構造を変化させなければならないようなリアルタイムな木構造である．このような頻繁な構造変化を嫌う場合でも，積極的に AVL 木のもつ平衡性を利用しようとする研究，データベースにおいてロック機能を必要とする処理でも，遅延で平衡化を可能とする研究 [23, 25, 27, 26, 36, 10] などがある．AVL 木の探索効率だけに注目する以外にも，他分野に応用される例もある．例えば，文字列処理に関する AVL 木の利用分野では，文脈自由文法の研究 [42] やパターン照合処理に有効なサフィックス木を AVL 木に適応させる研究 [22] がある．また，コンピューターに関する AVL 木の利用分野では，AVL 木のデータ構造を利用して圧縮を論じる研究 [15]，インターネット上における鍵管理 [39]，ワイヤレスネットワークにおける管理 [56]，Web ページの構造の応用 [54] などがある．以上の研究からも，AVL 木に関する研究やそれらの実用化などの研究は，データ構造を研究する上で今後とも重要であると考えられる．本論文で提案する木構造は，AVL 木のもつ平衡性を利用して効率的な探索法を提案するものである．

一方，木構造での文字列の探索には，上述したような 2 分探索木や AVL 木の他に，文字（桁）単位で探索する木構造が存在する．文字単位で探索する木構造を**基数探索法**（radix-search method）または文字列探索法と呼ぶ．2 分探索木や AVL 木は，探索に使用するキーワードを全体で比較対象にする必要があった．対する基数探索法では文字単位で比較をする．キーワードを文字単位で探索することは，文字毎の遷移の木構造が表現されることを意味し共通接頭辞の探索が容易に実現でき，自然言語処理に利用されている [24, 13]．コンピュータ上で探索を行う場合は，実質的に文字列を探索する場合が圧倒的に多く，実用性の高い文字列探索法としては，**トライ** [12, 14] がや**パトリシア**[34] が代表的であり，共通接頭辞の探索が可能という他のデータ構造にはない特徴があるので，スペルチェック [37]，索引検索 [18] や形態素解析 [33] などの幅広い分野で利用されている．

パトリシアは，トライで発生する ‘一方向分岐’ を少なくすることにより探索時間を短縮するものである．トライの代表的なデータ構造としては，配列構造とリスト構造がある．配列構造は，遷移を $O(1)$ で実現できるため，集合 S のデータの総数に依存しない高速な探索が可能である．しかし，配列構造に未使用の領域があれば，空間効率は悪くなる．リスト構造は，無駄な領域が作られず空間効率は良いが，時間計算量が節点から出る遷移の数に依存するため，探索時間はデータの総数に影響を受けてしまう．これらの問題を解決する効率の良い他の応用アルゴリズムにダブル配列がある．ダブル配列は 2 つの配列を使用して高速性とコンパクト性を実現しており，多くの応用研究 [16, 35, 55] と欠点を改善する研究 [47] が活発である．

本論文は，上述を踏まえたうえで，高速な探索を実現するために，AVL 木を出発点にして次の構造を構築する．

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．

1. は，文字列の探索のために基数探索法を利用する．
2. は，AVL 木の平衡化の操作を利用する．
3. は，既存の AVL 木は 2 分木であるが，多分木にすることで分岐数を増やし効率的な探索を実現する．
4. は，3. の多分木によって，節点から出る遷移の数を増やすことで効率的な探索を実現する．
5. は，本質的ではないが，挿入や削除を多用するような辞書を構築する．

この木を**拡張した** AVL **木**（Extended AVL tree）と名付け，多分木の平衡した AVL 木を構築することで，木の高さを小さくして効率の良い探索木を構築する．

1.2　研究の概要

第 1 章では，議論をする上での基礎となるアルゴリズムの概念を説明する．アルゴリズムの意味，計算量の評価に使う記号やアルゴリズムの評価法について説明する．

第 2 章では，第 1 章で述べたアルゴリズムがデータ構造とどう関係するかを説明する．データ構造を構成するために必要な応用範囲の広い基本的な構成法を C 言語を使用して説明する．さらに，現実の問題を定式化して解くモデルとして有効なグラフの表現法を解説する．この論文では，直接にグラフのアルゴリズムは扱わないが，木構造において重要な用語を説明する．アルゴリズムとデータ構造をひと通り説明したあと，具体的な木構造を紹介する．紹介する木構造は，最も基本的な木構造である 2 分探索木，この木の高さを小さくした平衡木の AVL 木，多分木で文字列探索に適した木構造であるトライやパトリシア木，計算機で実用的に使われている B 木である．

第 3 章では，第 1 と 2 章での結果をもとに，効率的な文字列探索に適応した木構造を提案する．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 木である．木の高さを小さくするために，部分木単位で平衡木となる．この提案する木構造の様々なアルゴリズムと計算量を述べ，既存の研究されている木構造との比較を行う．

第 4 章では，提案する木構造と既存の木構造である 2 分探索木，AVL 木と B 木との数値実験を行っている．10 桁，20 桁，50 桁，100 桁のランダムな数値を 10,000,000 個を乱数で発生して，2 分探索木と AVL 木の構築時間，メモリ使用量，比較回数などを比較した．提案する木構造は，構築時間や比較回数など 2 分探索木と AVL 木に比べて良い結果となったが，短所としてメモリ使用量が 100 桁の場合で AVL 木の 1.2 倍になっている．この短所については，第 5 章で考察することにする．B 木については，100 桁のランダムな数値を 10,000,000 個を乱数で発生して，構築時間，メモリ使用量と比較回数を比較した．すべての比較項目で，提案する木構造での良い結果がでている．

第 5 章では，第 4 章の数値実験で明らかになったメモリ使用量の削減を検討している．データ構造の改良点は，ポインタ数の削減とデータの格納場所となる配列数の削減である．これらの改良については，データが 10 進数，アルファベット，五十音の場合に対する理論的な評価を行っている．また，拡張した AVL 木の一般化と探索・挿入・削除の基

本操作の概要について述べている．第 6 章では，前章までのまとめと今後の課題について述べている．

付録には，数値実験のために作成したプログラムを載せている．

付録 A 2 分探索木のソースコード

付録 B AVL 木のソースコード

付録 C B 木のソースコード

付録 D 拡張した AVL 木のソースコード

1.3 アルゴリズム

計算機は「命令通り正確に動作する機械」であるから，有限ステップで**停止**(halt) するものが理想となる．必ずしも有限ステップで停止が保証されていない場合は，単に**手続き**(procedure) と呼ぶ．まず，**アルゴリズム**(algorithm) の定義を簡単に述べる [19].

定義 1.1. アルゴリズムとは，与えられた問題を解くための機械的操作からなる有限の手続きである．□

上の定義をより形式的にしたものが, A.M.Turing が提案した**チューリング機械**(Turing Machine; 以下 TM) である．TM は計算機の機能を極限まで単純化した仮想マシンであり，今日のアルゴリズムや計算に関する理論の基礎となっている．Turing は日常の計算行為を鋭く分析して，**計算可能**(computable) を数学的に定義し，この概念を定義する手法として TM が提案された．TM の基本的モデルとは，1 次元で無限に伸びる**入力テープ**(input tape) を持ち，入力テープは**ます目**(cell) に分けられ，左右に移動可能な読み書き用の**テープヘッド**(tape head) と**有限制御部**(finite control) と呼ばれる部分から構成されている．有限制御部はテープヘッドを通じて一つのます目の文字を読み込み，**次動作関数**(next move function) の引数である読み込んだ文字と有限制御部の状態からます目の文字を書き換えるとともに，テープヘッドを左右どちらかに 1 ます動かす．次動作関数が定義されていなければ，有限制御部の状態と受理状態の集合を比較することで，計算可能の可否を決定する．基本モデルだけを述べたが，この概念でアルゴリズムを数学的に説

明した．

さて，ある問題に対して，各人がアルゴリズムを作成したと仮定する．しかし同じ結果が得られても，アルゴリズムは多々存在する．同じ結果を満たすとき，どのアルゴリズムが優れているかを測る基準に時間効率をあげることができる．つまり，仕事量が増えても増加時間がもっとも短いアルゴリズムが優秀であると考えられる．このように増加時間でアルゴリズムの優劣をつける基準に**時間計算量**(time complexity) がある．さて時間効率を述べるにしても基準となる単位時間を明らかにしなければならないが，ここでは四則演算や記憶装置へのアクセス等の基本操作を単位時間とする．つまり仕事量となる入力サイズにおいて，プログラム中の基本操作が何回ほど実行されるかで評価しようとするものである．「何回ほど」と述べたのは，入力サイズに対して，漸近的な時間効率を測るためである．以下に様々な評価方法を述べるが，これより大体の時間効率を得ることができる．

1.4　アルゴリズムの評価法

入力サイズ n の問題が与えられたとき，これを解くアルゴリズムの実行時間を $f(n)$ とする．n が増加したときの $f(n)$ の極限の振舞いが分かれば，このアルゴリズムの評価ができる．関数 $g(n)$ と関数 $f(n)$ を以下のように定義する [49]．ここでは，基本的な評価法である O 記法，o 記法，Ω 記法，ω 記法，Θ 記法，θ 記法を述べる．

(1) O 記法 (ビッグ・オー)

定義 1.2. 関数 $g(n)$ に対し関数の集合を次で定める．

$$O(g(n)) = \{f(n) \mid \text{ある正の定数 } c \text{ と正の整数 } N \text{ が存在して} \\ \text{すべての } n \geq N \text{ に対して } 0 \leq f(n) \leq cg(n)\} \quad \square$$

このとき $f(n) \in O(g(n))$ となるが，これを $f(n) = O(g(n))$ と記す．言いかえると十分大きな入力サイズ n に対して，あるアルゴリズムの実行時間 $f(n)$ は最悪の場合でも $g(n)$ の c 倍以下である．すなわち $f(n)$ は，$g(n)$ の定数倍 c を上界として含む．これを図 1.1 に示す．

例えば，あるアルゴリズムの実行時間 $f(n)$ が最悪の場合で，

$$f(n) = 3n \log n + 2n^2 \tag{1.1}$$

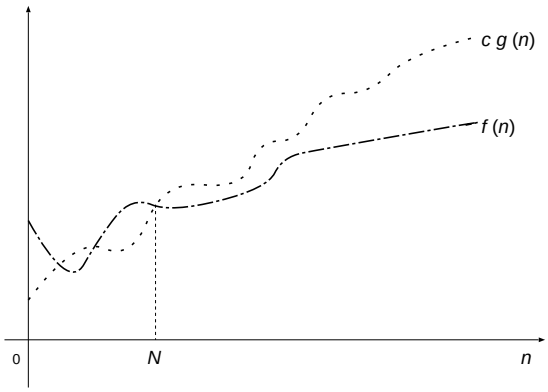


図 1.1　$f(n) = O(g(n))$

とすると，$f(n) = O(n^2)$ が成立する．$f(n) = O(n^3)$ も成立するが，$f(n) = O(n^2)$ の方が後者より精度が高くなる．

上の定義を用いて，加算を行う．2 つのアルゴリズム実行時間を $f_1(n)$，$f_2(n)$，正の定数を c_1, c_2，正の整数を N_1, N_2，関数を $g_1(n)$, $g_2(n)$ とおき，次のように定義する．

$$\begin{aligned} &\forall n \geq N_1 \quad \text{に対して} \quad f_1(n) \leq c_1 g_1(n) \\ &\forall n \geq N_2 \quad \text{に対して} \quad f_2(n) \leq c_2 g_2(n) \end{aligned} \tag{1.2}$$

このとき $N_0 = \max(N_1, N_2)$ とする．$\forall n \geq N_0$ に対して，

$$\begin{aligned} f_1(n) + f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_1 \max(g_1(n), g_2(n)) + c_2 \max(g_1(n), g_2(n)) \\ &= (c_1 + c_2) \max(g_1(n), g_2(n)) \\ &= O(\max(g_1(n), g_2(n))) \end{aligned}$$

となる．例えば，

$$\begin{aligned} &\forall n \geq N_1 \quad \text{に対して} \quad f_1(n) \leq 5n^2 \\ &\forall n \geq N_2 \quad \text{に対して} \quad f_2(n) \leq 4n^{1.83} \log n \end{aligned}$$

このとき $N_0 = \max(N_1, N_2)$ とする．$\forall n \geq N_0$ に対して，

$$\begin{aligned} f_1(n) + f_2(n) &= O(\max(n^2, n^{1.83} \log n)) \\ &= O(n^2) \end{aligned}$$

である．

同様に乗算については，式 (1.2) を使って

$$\begin{aligned} f_1(n) \cdot f_2(n) &\leq c_1 g_1(n) \cdot c_2 g_2(n) \\ &= (c_1 \cdot c_2)(g_1(n) \cdot g_2(n)) \\ &= O(g_1(n) \cdot g_2(n)) \end{aligned}$$

となる．例えば

$$\begin{aligned} &\forall n \geq N_1 \quad \text{に対して} \quad f_1(n) \leq 4n^3 \sin^2 n \\ &\forall n \geq N_2 \quad \text{に対して} \quad f_2(n) \leq \frac{2^n}{\log n} \end{aligned}$$

このとき $N_0 = \max(N_1, N_2)$ とする．$\forall n \geq N_0$ に対して

$$\begin{aligned} f_1(n) \cdot f_2(n) &= O\left(\frac{2^n n^3 \sin^2 n}{\log n}\right) \\ &= O\left(\frac{2^n n^3}{\log n}\right) \qquad \left(\because \sin^2 n \leq 1\right) \end{aligned}$$

である．

(2) o 記法 (リトル・オー)

上述の $O(g(n))$ は，$f(n)$ に関するある上界を示していたが，$f(n)$ に対して「どの程度の近さ」かを示すものではなかった．式 (1.1) では，$f(n) = O(n^2)$ または $f(n) = O(n^3)$ でも成立したが，$f(n) = O(n^3)$ よりも $f(n) = O(n^2)$ の方が，精度が高い．すなわち，$f(n) = O(n^3)$ では $f(n)$ の上界に対して改良の余地があることを次の定義で示す．

定義 1.3. 関数 $g(n)$ に対し関数の集合を次で定める．

$$\begin{aligned} o(g(n)) = &\{f(n) \mid \text{任意の正の定数 } c \text{ と正の整数 } N \text{ が存在して} \\ &\text{すべての } n \geq N \text{ に対して } 0 \leq f(n) \leq cg(n)\} \quad \square \end{aligned}$$

式 (1.1) では，$f(n) = o(n^3)$ が成立して，$f(n) = o(n^2)$ は成立しない．これで，$f(n) = o(n^2)$ が，$f(n)$ により近い関数であると分かった．

(3) Ω 記法 (ビッグ・オメガ)

定義 1.4. 関数 $g(n)$ に対し関数の集合を次で定める．

$$\Omega(g(n)) = \{f(n) \mid \text{ある正の定数 } c \text{ と正の整数 } N \text{ が存在して} \\ \text{すべての } n \geq N \text{ に対して } 0 \leq cg(n) \leq f(n)\} \quad \square$$

このとき $f(n) \in \Omega(g(n))$ となるが，これを $f(n) = \Omega(g(n))$ と記す．言いかえると十分大きな入力サイズ n に対して，あるアルゴリズムの実行時間 $f(n)$ は最良の場合でも $g(n)$ の c 倍以上である．すなわち $f(n)$ は，$g(n)$ の定数倍 c を**下界**(lower bound) として含む．これを図 1.2 に示す．

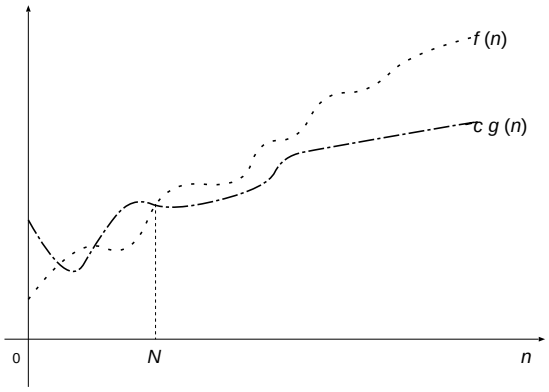


図 1.2　$f(n) = \Omega(g(n))$

例えばあるアルゴリズムの実行時間 $f(n)$ が最良の場合で，

$$f(n) = 5n^{\log n} + 4n^{100} + 3n^3 \log n \tag{1.3}$$

とすると，$f(n) = \Omega(n^{\log n})$ で成立する．$f(n) = \Omega(n^{\log n - 1})$ でも成立するが，$f(n) = \Omega(n^{\log n})$ の方が後者より精度が高くなる．

また $\Omega(g(n))$ を以下のように定義する場合もある．

定義 1.5. 関数 $g(n)$ に対し関数の集合を次で定める．

$$\Omega(g(n)) = \{f(n) \mid \text{ある正の定数 } c \text{ が存在して} \\ \text{無限個の } n \geq N \text{ に対して } 0 \leq cg(n) \leq f(n)\} \quad \square$$

この意図は,

$$f(n) = \begin{cases} n^2 & n：奇数 \\ n^3 & n：偶数 \end{cases}$$

のような場合に，$f(n) = \Omega(n^3)$ と主張したいからである.

(4) ω 記法 (リトル・オメガ)

上述の $\Omega(g(n))$ は，$f(n)$ に関するある下界を示していたが，$f(n)$ に対して「どの程度の近さ」かを示すものではなかった．式 (1.3) では，$f(n) = \Omega(n^{\log n})$ または $f(n) = \Omega(n^{\log n - 1})$ でも成立したが，$f(n) = \Omega(n^{\log n - 1})$ よりも $f(n) = \Omega(n^{\log n})$ の方が，精度が高い．すなわち，$f(n) = \Omega(n^{\log n - 1})$ では $f(n)$ の下界に対して改良の余地があることを次の定義で示す.

定義 1.6. 関数 $g(n)$ に対し関数の集合を次で定める.

$$\begin{aligned} \omega(g(n)) = \{ & f(n) \mid 任意の正の定数\ c\ と正の整数\ N\ が存在して \\ & すべての\ n \geq N\ に対して\ 0 \leq cg(n) \leq f(n) \} \quad \square \end{aligned}$$

式 (1.3) では，$f(n) = \omega(n^{\log n - 1})$ が成立して，$f(n) = \omega(n^{\log n})$ は成立しない．これで，$f(n) = \omega(n^{\log n})$ が，$f(n)$ により近い関数であると分かった.

(5) Θ 記法 (ラージ・シータ)

定義 1.7. 関数 $g(n)$ に対し関数の集合を次で定める.

$$\begin{aligned} \Theta(g(n)) = \{ & f(n) \mid ある正の定数\ c_1\ と\ c_2\ と正の整数\ N\ が存在して \\ & すべての\ n \geq N\ に対して\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \} \quad \square \end{aligned}$$

このとき $f(n) \in \Theta(g(n))$ となるが，これを $f(n) = \Theta(g(n))$ と記す．言いかえると十分大きな入力サイズ n に対して，あるアルゴリズムの実行時間 $f(n)$ は最良の場合で $g(n)$ の c_1 倍以上であり，最悪の場合でも $g(n)$ の c_2 倍以下である．すなわち，$f(n)$ は，$g(n)$ の定数倍 c_1 を下界として含み，かつ $g(n)$ の定数倍 c_2 を上界として含む．これを図 1.3 に示す.

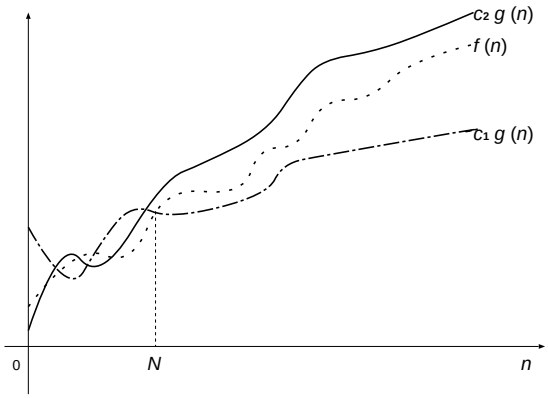


図 1.3　$f(n) = \Theta(g(n))$

例えば，あるアルゴリズムの実行時間 $f(n)$ を

$$f(n) = \begin{cases} 2n^3 + 2n^2 + 3 & \text{(最良)} \\ 4n^3 + 5n^2 & \text{(最悪)} \end{cases}$$

とすると，

$$f(n) = \Theta(n^3)$$

である．

あるアルゴリズムの実行時間 $f(n)$ を

$$f(n) = \begin{cases} 2n^3 + 2n^2 + 3 & \text{(最良)} \\ 4n^4 + 5n^2 & \text{(最悪)} \end{cases}$$

とすると，$g(n)$ を定めることができない．

アルゴリズムの実行時間

入力サイズを n，そのときのあるアルゴリズムの実行時間を $f(n)$ とする．入力サイズを変化させたときの概算の実行時間 $f(n)$ を求めてみる．ここでは，1 秒当たり 10^9 回の処理を行なうとし，実行時間の概算を表 1.1 に示す．

表 1.1 から分かるように，多項式時間となるアルゴリズム (n，$n\log_2 n$，n^2) では，入力サイズを増加させても早く処理が終わることが分かるが，指数関数時間となるアルゴリズム (2^n，$n!$[40]) では，入力サイズを増加させるにつれて，実行時間も急激に増大するの

表 1.1　各アルゴリズムでの実行時間

$f(n)$	n	$n\log_2 n$	n^2	2^n	$n!$
10	10^{-8} 秒	3×10^{-8} 秒	10^{-7} 秒	10^{-6} 秒	4×10^{-3} 秒
20	2×10^{-8} 秒	9×10^{-8} 秒	4×10^{-7} 秒	10^{-3} 秒	77 年
30	3×10^{-8} 秒	15×10^{-8} 秒	9×10^{-7} 秒	1 秒	8×10^{15} 年
40	4×10^{-8} 秒	21×10^{-8} 秒	16×10^{-7} 秒	18 分	3×10^{31} 年
50	5×10^{-8} 秒	28×10^{-8} 秒	25×10^{-7} 秒	13 日	10^{48} 年
100	10^{-7} 秒	66×10^{-8} 秒	10^{-5} 秒	4×10^{13} 年	3×10^{141} 年
1000	10^{-6} 秒	997×10^{-8} 秒	10^{-3} 秒	3×10^{284} 年	10^{2551} 年

が分かる.

第 2 章

基本的なデータ構造と木構造

2.1 基本的なデータ構造

前 1 章ではアルゴリズムについて説明した．プログラムを実現するとき，必要となるデータが分かれば，そのデータに対する操作 (アルゴリズム) が決まる．逆にデータに対する操作が明らかになると効率よく操作できるデータの表現法 (データ構造) が決まることになる．つまり，アルゴリズムとデータ構造の両者を同時に考えなければならない [53].

以下で，まずデータ型，抽象データ型，データ構造について述べる．

2.1.1 データ型と抽象データ型

計算機はデータを格納し処理を行う．**データ型**(data type) とは，これらデータに関する値のとり得る集合 (変域) を示す．例えば，C 言語は組み込まれた基本的なデータ型に整数型 (int) があるが，**変域**(domain) は $-\text{maxint} \sim \text{maxint}$ である．また，各データ型には操作 (演算) の集合が定義されている．例えば，整数型ならば演算子の集合は単項演算子・2 項演算子等が含まれる．以上よりデータ型を以下のように定義する [46].

定義 2.1. データ型を次で定める．

1. とり得る値の集合である変域 (domain) である．
2. その値に対する操作 (演算) の集合である． □

C 言語には基本データ型に整数型・実数型 (浮動・倍精度)・文字型がある．基本データ

型は値をこれ以上分解できない**成分要素**(component element) であり，**原子的**(atomic) であるという．データ型の重要な概念に**列挙型**(enumeration type) がある．すべての値を並べることで定義されるデータ型である．C 言語ではデータ型に予約語 enum，変数に signal を使って

```
enum signal {BLUE, YELLOW, RED};
```

で表すことができる．上では，signal の変域は，BLUE, YELLOW, RED である．

抽象データ型(abstruct data type) は，データ型の一般化になる．上述でも定義した通り，抽象データ型とはデータの型とその型に対する一連の操作 (演算) を組みにしたものである．例えば，データをグラフ (2.1.6 節を参照) で表すならば，データを節点で表現し，データ間の関係を枝で表現する．節点と枝を付け加えるとか除去する操作を定義すると抽象データ型が得られる．このようなデータとそのデータに対する操作手続きを組みとすることを一般化あるいは**カプセル化**(encapsulation) という．このようにすることで，用意された操作だけが利用されプログラムの信頼性を向上させる．さらにいくつかのカプセル化された抽象データ型をカプセル化することも可能となる．

2.1.2　データ構造

このデータ型を拡張して，**データ構造**(data structure) を定義する．例えば，文字の並びを C 言語で表現すると「文字型が規則的に並んだもの」である．つまり，文字型が一列にある数連なったものと考えることができる．これを便宜上データ型の一種である文字列型と仮りに名付けると，文字列型もデータ型なのである値をもつ．また，文字列型の 1 つの値においてはある数の文字型の成分要素に分けることができて，それぞれの文字は文字型の変域からの 1 つの値である．これら成分要素の値は，関連の上で文字列型のある値を決定している．以上よりデータ構造を以下のように定義する [46].

定義 2.2. 次を満たすデータ型をデータ構造と定める．

1. 値は成分要素の集合に分解することでできる．その要素は原子的であってもよいし，別のデータ構造であってもよい．
2. 値は成分要素を関連づける結合，または関係の集合 (構造) を含む．　□

2.1.3 基本的なデータ構造

次のようにデータを 0 個以上，一列にならべたものを**リスト**(list) と呼ぶ.

$$a_1, a_2, \cdots, a_n$$

特に $n = 0$ のときを**空リスト**(null list) と呼ぶ．データ a_i は i 番目の位置にあり，データ a_{i-1} は a_i の直前，データ a_{i+1} は a_i の直後にある．つまり，リストに含まれるデータは順序づけられており，この意味で**線形リスト**(linear list) と呼ばれることもある．このリストを実現するいくつかの方法を次に示す.

(1) 配列による実現

最も簡単な表現方法は**配列**(array) である．配列とは記憶領域の一単位を**セル**(cell) としたとき，連続した同じデータ型のセルの並びである．C 言語では配列の**添字**(index) は 0 から始まるので，リストの a_i をセル $a[i-1]$ に格納することによって，配列による表現が可能となる．配列の最も優れている点は，そのデータが何番目に登録されているかで，すぐに参照できることである．すなわち，ランダムアクセスが可能である．一方，配列の途中にデータを挿入・削除することは，途中からのデータをすべて繰り下げるか上げるかの操作が必要となるため多くの時間を必要とする．配列では，リストのデータ数を n 個とすると挿入・削除の時間計算量は $O(n)$ になる．また，あらかじめセルの数を決定しておかなければならなく，加えられるデータの数が限られてくる欠点もある．逆にセルの数を多めに設定すると，利用されないセルは無駄となる．空きセルがあるときのみに限り，データを加えることができる．配列の添字で位置が一意に確定されるので，参照操作は容易であるが，挿入および削除は苦手である.

(2) ポインタによる実現

リストを配列で表現しただけでは，データを挿入・削除する場合に操作は簡単ではなかった．これを解決するために別な表現方法を導入する．配列は同じデータ型を集めたデータ構造であったが, 異なるデータ型を一つのセルとする C 言語での**構造体**(structure) がある．C 言語では構造体の各要素を**メンバ**(member)，セルの位置を**ポインタ**(pointer)

と呼ぶ.

1 セルを構造体 CELL で表し，メンバを data と next とする．メンバ data はリストの整数型データであり，メンバ next は次のセルの位置を指すポインタとする．以上の 2 メンバをもつ構造体を以下のように定義する.

```
struct CELL {
        int data;
        struct CELL *next;
};
```

これにより，各セルをデータ部とポインタ部にして物理的に非連続なセルの並びを保持できる．各セル間をポインタによって繋げると，リストが作成できる．これを**連結リスト**(linked list) と呼ぶ．最初のセルは**ヘッダー**(header) と呼ばれ，1 番目のセルのアドレスが格納されている．以下，$i = 1, 2, \cdots$ に対して，ヘッダーを除く i 番目のセルにはデータと $(i+1)$ 番目のセルのアドレスが対となりセルに保存されている．最後のセルである a_n のポインタ部のアドレスには 'NULL' が保存されており，次のセルがないことを意味している．この状態を図 2.1 に示す.

図 2.1　連結リスト

連結リストの最大の利点はデータの挿入・削除が容易に操作できることである．しかし，ポインタを順番にたどるシーケンシャルアクセスしかできないため，連結リストの i 番目のデータ部を読むとか連結リストの最後尾に新たなセルを設ける操作等は苦手となる．また，前のデータへのアドレスを保存していないので，ある操作の途中において前のデータを参照する必要が生じた場合に容易でない．このようにデータのアクセスについては，連結リストは配列に比べて不利であると言える.

連結リストの最大の利点である挿入・削除について説明する．連結リストの挿入・削除ではセル自体を移動させる必要がなく，ポインタを書き換えるだけで済むので時間計算量は $O(1)$ となる．ポインタ p で指されているセルの次に，ポインタ q で指されるセルを挿

入するには

```
1    q -> next = p -> next;
2    p -> next = q;
```

となる．これを図 2.2 で示す．(a) を挿入前，(b) を挿入後としている．

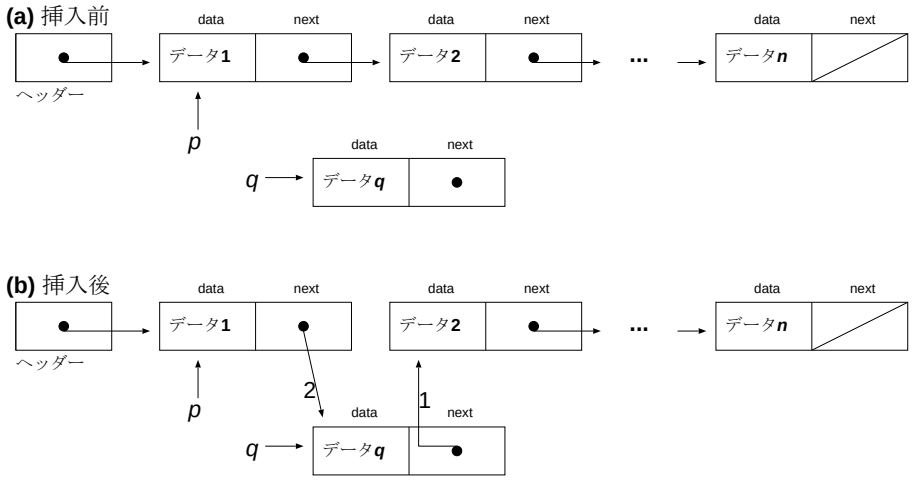


図 2.2 連結リストでの挿入

ポインタ p で指されているセルの次のセルを削除する．削除されるセルのポインタを q とすると，

```
1    q = p -> next;
2    p -> next = q -> next;
```

となり，図 2.3 で示す．(a) を削除前，(b) を削除後としている．

また，配列のようにあらかじめ領域確保をする必要がないので，領域を有効に利用できると考えられる．

(3) 配列によるポインタの実現

ポインタ機能がサポートされていない言語でも，配列を用いて連結リストを実現できる．整数データ型 init と 2 つの配列 element と next を準備して，element にはリスト a_i のデータを next には次の位置である添字を保存する．連結リストにおけるヘッダーを変数 init の代わりにして，この init に最初のデータである element の添字を保存させる．

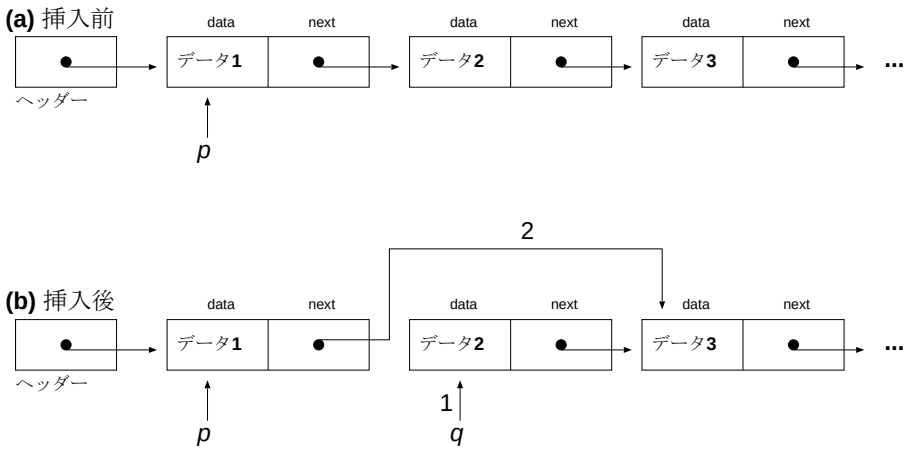


図 2.3　連結リストでの削除

element の添字には a_1 のデータが保存されている．このとき，next の添字には a_2 への添字が保存されている．つまり，element$[k]$ の内容が a_i であるとすると，next$[k]$ には a_{i+1} へのポインタが保存されている．すなわち，element[next$[k]$] の内容が a_{i+1} である．また最後のリスト a_n には，NULL の代わりとなる -1 を使う．これにより，データの挿入や削除は (2) ポインタによる実現のように簡単に行なえる．

配列 element の添字 k に a_i のデータが保存されているとき，a_i の次に配列 element の添字 j にあるデータを a_{i+1} として挿入したい場合は

```
1    next[j] = next[k];
2    next[k] = j;
```

である．

リストを

$$A, C, N, X, W, Z$$

とする．このリストの N と X の間に P を挿入する．これを図 2.4 で示す．図の (a) は挿入前を (b) は挿入後である．上のソースコードでは，$k = 1, j = 6$ であるから，

```
1    next[6] = next[1];
2    next[1] = 6;
```

とすれば，図 2.4 の（b）となる．

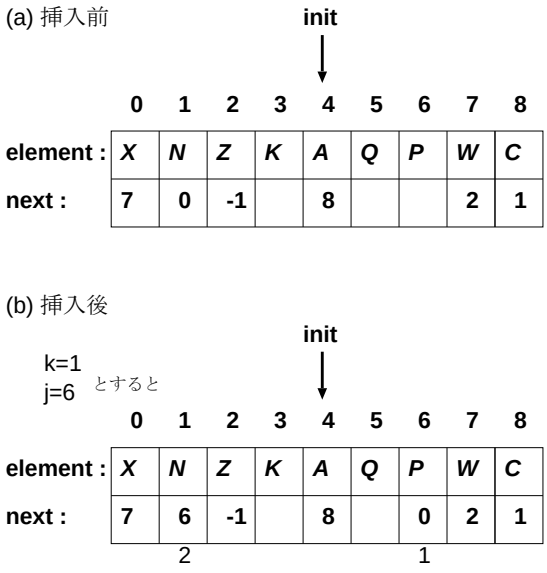


図 2.4　配列によるポインタの実現

(4) 双方向リスト

双方向リスト(doubly-linked list) とは，各セルにおいてデータ部と 2 つのポインタ部からなる．2 つのポインタ部には直前と直後のアドレスが保存されている．最大の利点は，連結リストと同様にデータの挿入・削除が容易に操作できることである．加えて，連結リストでは直前のアドレスに遡ることはできなかったが，双方向となることでできるようになる．1 セルを構造体 CELL で表し，メンバを prev と data と next とする．メンバ prev は前のセルを指すポインタ，メンバ next は次のセルの位置を指すポインタ，メンバ data はリストの整数型データとする．以上の 3 メンバをもつ構造体を以下のように定義する．

```
struct CELL {
      struct CELL *prev;
      int data;
      struct CELL *next;
};
```

このとき，次のように変数 head をリストの先頭とする．それとこの変数 head を初期化する．初期化では head の前後ポインタとも自分自身である head を指すようにする．

```
1    struct CELL head;
2    head.prev = &head;
3    head.next = &head;
```

これを図 2.5 で示す．図の数字は，上述のソースコードの番号である．初期化として変数 *head* を準備するが，これはデータ部には何も追加されない．

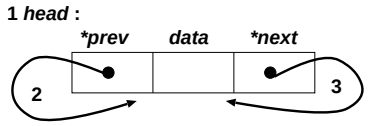


図 2.5　双方向リストの初期化

挿入のアルゴリズムを考えてみる．ポインタ p が指す直後のセルにポインタ q が指すセルを挿入する．

```
1    q -> prev = p;
2    q -> next = p -> next;
3    p -> next -> prev = q;
4    p -> next = q;
```

これを図 2.6 で示す．図の数字は，上述のソースコードの番号である．図 (a) は挿入前を (b) は挿入後を示す．

次に削除のアルゴリズムを考えてみる．削除したいセルのポインタを p とする．

```
1    p -> prev -> next = p -> next;
2    p -> next -> prev = p -> prev;
```

これを図 2.7 で示す．図の数字は，上述のソースコードの番号である．図 (a) は削除前を (b) は削除後を示す．

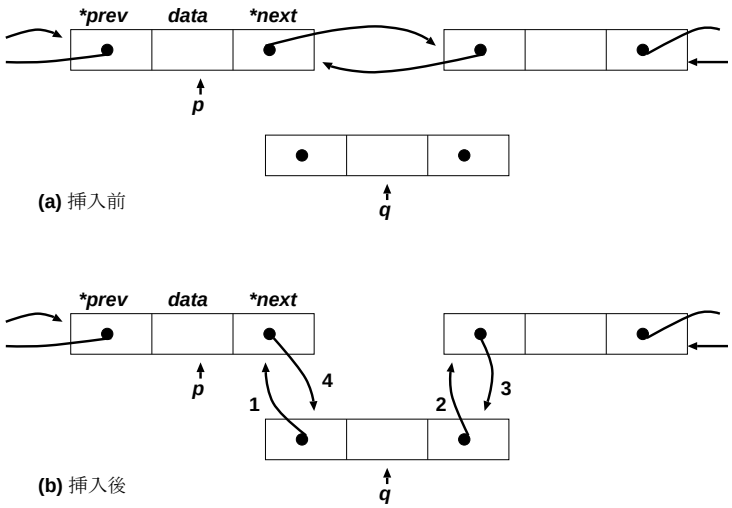


図 2.6　双方向リストの挿入

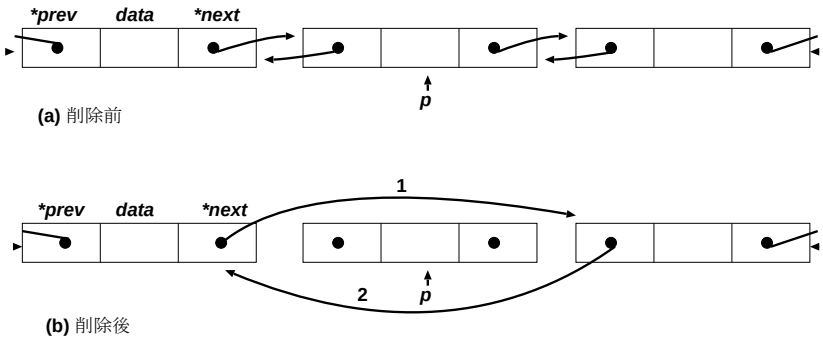


図 2.7　双方向リストの削除

リストのデータを前後に移動できるようにはなったが，データのアクセスは相変わらずシーケンシャルアクセスより，参照は不利となる．

2.1.4　スタック

これまでと異なるデータ構造を構築する．今までは，リストにおける順番を計算機で「どの様に表現するか」ということに注目してきた．つまり，リストの並びに注目して操作を行ってきたが，次に説明する**スタック**(stack) では，挿入と削除をリストの先頭セルだけに行うデータ構造である．ここでのリストの先頭セルとは，一番最後に加えられたデータであり，一番最初に加えられたデータは，最後尾セルに保存される．操作対象が先頭セ

ルのみであることから，別名 *LIFO*(last-in-first-out, 先入れ先出しリスト) と呼ばれており，計算機処理では必須のデータ構造である．例えば，C 言語での各関数は一番最後に呼び出された関数を実行し，その呼び出した関数の続きを再開するが，これを繰り返して最後に main 関数に戻ることでプログラムを実現している．また，**再帰呼出し**(recursive call) を可能とする C 言語では再帰プログラムをスタックで実現している．

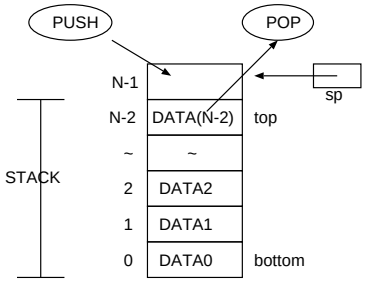


図 2.8　配列によるスタックの実現

スタックは，セルを加えられた順番で保存するデータ構造であり，データは 1 次元 (線形) であれば良い．すなわち，スタックの実現方法に配列を使うことで容易となる．スタックにおいて，先頭セルの位置を「top」，最後尾セルの位置を「bottom」で表す．また，「top」の次のセルを「ポインタ *sp*」とする．データを加える操作を「PUSH」というが，データを加える位置を「ポインタ *sp*」とする．データを削除する操作を「POP」というが，データを削除する位置を「top」とする．スタックに DATA0 〜 DATA($N-2$) を追加した図が図 2.8 である．この図において，ポインタ *sp* の初期状態を 0 として，スタックにデータがないことを意味する．この状態で PUSH 操作をすれば *sp* の位置を 1 上げ (*sp* の値を 1 加算)，POP 操作をすれば *sp* の位置を 1 下げればよい (*sp* の値を -1 加算)．気を付けなければならないことは，配列で実現しているので最大データ数が決まることである．スタックにおける主な操作には次のようなものがある．

- PUSH(x, *sp*) ： ポインタ *sp* 位置にデータ x を挿入する．
- POP($*x$, *sp*) ： ポインタ *sp* 位置の前データを削除する．削除したデータを $*x$ に代入する．

PUSH 関数のプログラム例をソースコード 2.1 に示す．配列数 (SIZE) を 10 で，スタックを構成する配列名を stack，x は整数型のデータ，*sp* はデータを加える位置である．

4 行目でスタックの空きがあるかの確認をする．スタックに空きがない場合は，10 行目で -1 を返す．スタックに空きがある場合は，5 〜 7 行目で sp の位置にデータ x を代入して，sp の値を更新後にこの値を返す．

ソースコード 2.1　PUSH 関数の例

```
1  #define SIZE 10
2  int PUSH(int x, int sp)
3  {
4    if (sp < SIZE) {
5      stack[sp] = x;
6      sp = sp + 1;
7      return sp;
8    } else {
9      puts("stack is FULL!");
10     return -1;
11   }
12 }
```

同様に POP 関数のプログラム例をソースコード 2.2 に示す．スタックを構成する配列名を stack，$*x$ は整数型を指すポインタ，sp はデータを削除する次のセル位置である．3 行目でスタックにデータの有無を確認をする．スタックにデータがない場合は，9 行目で -1 を返す．スタックにデータがある場合は，4 〜 6 行目で $(sp-1)$ 位置のデータを $*x$ に代入して，sp の値を更新後にこの値を返す．

ソースコード 2.2　POP 関数の例

```
1  int POP(int *x, int sp)
2  {
3    if (sp > 0) {
4      *x = stack[sp - 1];
5      sp = sp - 1;
6      return sp;
7    } else {
8      puts("stack is EMPTY!");
9      return -1;
10   }
11 }
```

2.1.5 待ち行列

後入れ先出しリストに対して，先入れ先出しリストを *FIFO*(first-in-first-out) と呼ぶ．これを計算機で実現する方法に**待ち行列**(queue) がある．リストにおいて，挿入は最後尾セルの後に加え，削除は先頭セルに対して行うデータ構造である．つまり，実生活でよく見かけるように，公平に順番通りに用を済ませる操作である．このデータ構造も計算機では広く利用されている．例えば，実行待ちのジョブ列や周辺機器の制御などの優先度

が同レベルにおいては，待ち行列で管理している．削除を行うリストの先頭セルの位置を $head$ ポインタで，挿入を行うリストの最後尾セルの位置を $tail$ ポインタで管理して，スタック同様に 1 次元配列を実現すれば良い．しかし，データの削除・挿入をするたび，$head \cdot tail$ ポインタが 1 ずつ加算されれば，いずれ配列の最後位置に達してしまう．この方法では，削除された位置のセルを使わなくなり無駄な領域を作ってしまう．この無駄な領域をつくらない方法に待ち行列をリング状の配列 (循環配列) とする方法がある．計算機では $tail$ ポインタの値を配列の最大長で割って余りを求めることで，リング状の配列と見なすことができる．初期状態として，$head$ ポインタと $tail$ ポインタを同じ位置にしておき，同じ位置のときはリストに何も加えられていない空状態を意味させる．リング状の配列と見なしているので，データが追加されるたびに $tail$ ポインタが時計回りに 1 ずつ回転することになる．また同様に，データが削除されるたびに $head$ ポインタが時計回りに 1 ずつ回転することになる．リング状の配列であるから，追加操作が多くなると $tail$ ポインタが一回りして，$head$ ポインタに追いついてしまう可能性もある．逆に削除操作が多くなると $head$ ポインタが $tail$ ポインタに追いついてしまう可能性もある．これらの状態では，最大データ数より多い場合か空の状態を意味するので，これを防ぐ手だてを行わなければならない．$head$ ポインタと $tail$ ポインタが同位置のときを空の状態と定義したので，$tail$ ポインタが一回りして $head$ ポインタの直前にあるときを待ち行列のデータが満杯の状態とする．つまり，待ち行列の最大データ数は待ち行列の配列数より 1 つ少ないデータ数までとなる．逆に $head$ ポインタが $tail$ ポインタと同じ位置となったときは，上述の通り空の状態である．この状態を図 2.9 に示す．図では待ち行列の配列数が 8 個なので，この待ち行列に加えることができるデータ数は 7 個である．

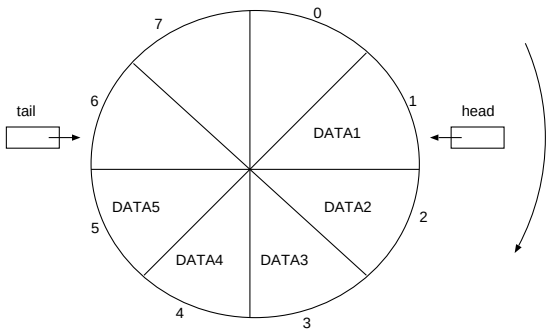


図 2.9　配列による待ち行列の実現

待ち行列における主な操作には次のようなものがある．

- ENQUEUE(x, tp, hp) ： ポインタ tp とポインタ hp が同じ位置あるいは直前の位置でなければ，ポインタ tp 位置にデータ x を挿入する．
- DEQUEUE($*x$, hp, tp) ： ポインタ hp とポインタ tp が同じ位置でなければ，ポインタ hp 位置のデータを削除する．削除したデータを $*x$ に代入する．

ENQUEUE 関数のプログラム例をソースコード 2.3 に示す．配列数 (SIZE) を 10 で，待ち行列を構成する配列名を queue，x は整数型のデータ，tp はデータを加える位置，hp はデータを削除する位置である．4 行目で待ち行列の空きを確認する．待ち行列に空きがない場合は，11 行目で -1 を返す．待ち行列に空きがある場合は，5 〜 8 行目でデータ x を tp の位置に代入して，tp の値を更新後にこの値を返す．

ソースコード 2.3 ENQUEUE 関数の例

```
1  #define SIZE 10
2  int ENQUEUE(int x, int tp, int hp)
3  {
4    if ((tp + 1) % SIZE != hp) {
5      queue[tp] = x;
6      tp = tp + 1;
7      tp = tp % SIZE;
8      return tp;
9    } else {
10     puts("queue is FULL!");
11     return -1;
12   }
13 }
```

同様に DEQUEUE 関数のプログラム例をソースコード 2.4 に示す．待ち行列を構成する配列名を queue，$*x$ は整数型を指すポインタ，tp はデータを加える位置，hp はデータを削除する位置である．3 行目で待ち行列のデータの有無を確認する．待ち行列にデータがない場合は，10 行目で -1 を返す．待ち行列にデータがある場合は，4 〜 7 行目で hp 位置のデータを $*x$ に代入して，hp の値を更新後にこの値を返す．

ソースコード 2.4 DEQUEUE 関数の例

```
1  int DEQUEUE(int *x, int hp, int tp)
2  {
3    if (hp != tp) {
4      *x = queue[hp];
5      hp = hp + 1;
6      hp = hp % SIZE;
7      return hp;
8    } else {
9      puts("queue is EMPTY!");
10     return -1;
```

```
11      }
12  }
```

2.1.6 グラフと木

グラフとは，「節点」と「辺」で描画された図である．グラフ上では「どの位置に節点がある」は関係なく「どの節点間に辺があるか」が問題となる．この意味で有限個の**節点**(node) または**頂点**(vertex) からなる有限集合 V と節点対の有限集合 $E \subseteq V \times V$ において，節点の有限集合と節点対の有限集合の組 $G = (V, E)$ を**グラフ**(graph) という．

節点を u, v とすれば，E のある要素 e が $e = (u, v)$ であるとき，この e を**辺**(edge) あるいは**枝**(branch) とよび，グラフ上で辺 e は u と v を結ぶ線分で示される．u と v は辺 e の**端点**(end nodes) である．辺に方向を考えないときを**無向グラフ**(undirected graph) とよび，辺に方向を考えるとき，つまり (u, v) と (v, u) を区別するときは**有向グラフ**(directed graph) と呼ぶ．ここでは，() 内の左側の節点から右側の節点に向かうものとする．つまり (u, v) において，u を始点，v を終点と呼ぶ．ここで，辺 (u, v) で u から v へ矢印を付けたものを**弧**(arc) と呼ぶ．無向グラフにおいて $e = (u, v)$ のとき，u と v は**隣接する**(adjacent) という．無向・有向グラフのいずれの場合でも e は，u および v に**接続する**(incident) という．

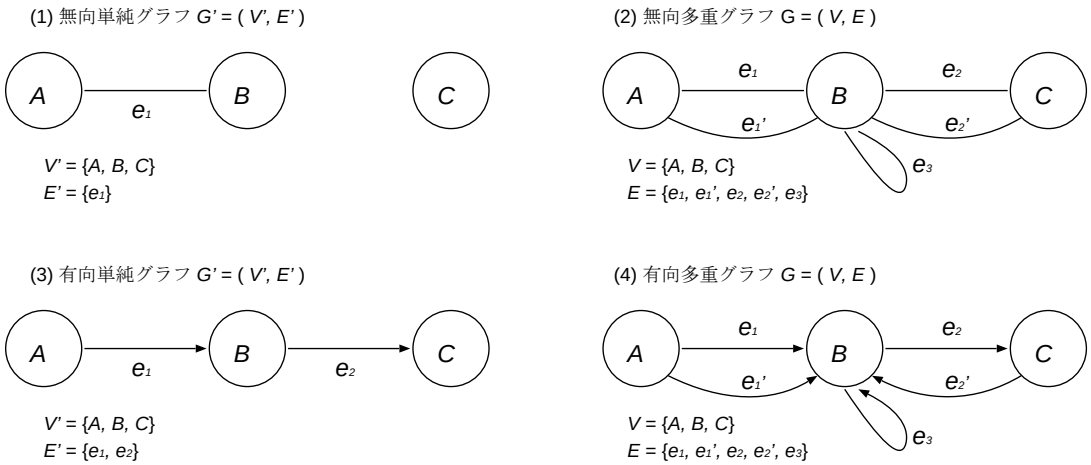


図 2.10　グラフの例

グラフの例を図 2.10 に示す．無向グラフにおける節点 u において，u に接続している

辺の数を**点次数**(degree) といい，$\deg(u)$ で表す．図の (2),(4) における辺 e_3 のように，同一の節点を結ぶ辺のことを**自己ループ**(self-loop) と呼ぶが，無向辺の場合は自己ループを除外して考える場合が多い．この意味で図の (2) では，$\deg(B) = 4$ である．有向グラフでは，節点 u を終点としてもつ辺の数を u の**入次数**(indegree) と呼び，u を始点としてもつ辺の数を u の**出次数**(outdegree) と呼ぶ．例えば図の (4) における節点 B では，入次数は 4 で出次数は 2 である．

図 2.10 の (2) における無向辺 e_1, e_1' と e_2, e_2' および図の (4) における有向辺 e_1, e_1' は 2 頂点間に 2 本以上の辺が存在し，この辺を**多重辺**(multiple edge) と呼び，特にこのグラフを**多重グラフ**(multigraph) と呼ぶ．しかし，図の (4) における有向辺 e_2, e_2' は多重辺ではない．多重辺も自己ループも存在しないことを**単純グラフ**(simple graph) と呼ぶ．

グラフ $G = (V, E)$ において，$V' \subseteq V$ と $E' \subseteq E$ の作る $G' = (V', E')$ がグラフであるとき，G' を G の**部分グラフ**(subgraph) という．つまり，二つのグラフ G と G' において，G' の節点集合と辺集合が共に G の節点集合と辺集合の部分集合であるとき，G' は G の部分グラフである．今述べた定義から

$$V' \supseteq \{u, v \in V \mid (u, v) \in E'\}$$

である．図 2.10 における (1) のグラフ G' は (2) のグラフ G の部分グラフである．このとき，

$$V' = \{u, v \in V \mid (u, v) \in E'\}$$

が成立している場合，G' は E' によって**生成された**(generated) **辺誘導部分グラフ**(induced subgraph by E') という．つまり，G' の節点集合 V' の要素が，部分集合 G' の辺集合 E' の両端点を全て含むときである．図 2.10 における (1) のグラフ G' は (2) のグラフ G の辺誘導部分グラフではない．

一方，

$$E' = \{e \in E \mid u, v \in V'\}$$

であるとき，G' は V' によって生成された**点誘導部分グラフ**(induced subgraph by V') という．つまり，G' の辺集合 E' の要素が，部分集合 G' の節点集合 V' の要素となる．図 2.10 における (1) のグラフ G' は (2) のグラフ G の点誘導部分グラフとなる．

グラフ $G = (V, E)$ の節点列

$$P : v_1, v_2, \cdots, v_k$$

が，$(v_i, v_{i+1}) \in E, \quad i = 1, 2, \cdots, k-1$ をみたすとき，v_1 から v_k への**経路**(path) あるいは**道**という．特に $v_1, v_2, \cdots, v_k$ が異なれば**単純経路**(simple path) であり，経路 P の**長さ**(length) を辺の本数 $(k-1)$ で定める．経路 P の始点 u と終点 v が等しいとき，P を**閉路**(cycle) と呼び，特に両端のみが等しい経路を**単純閉路**と言う．図 2.11 に例を示す．

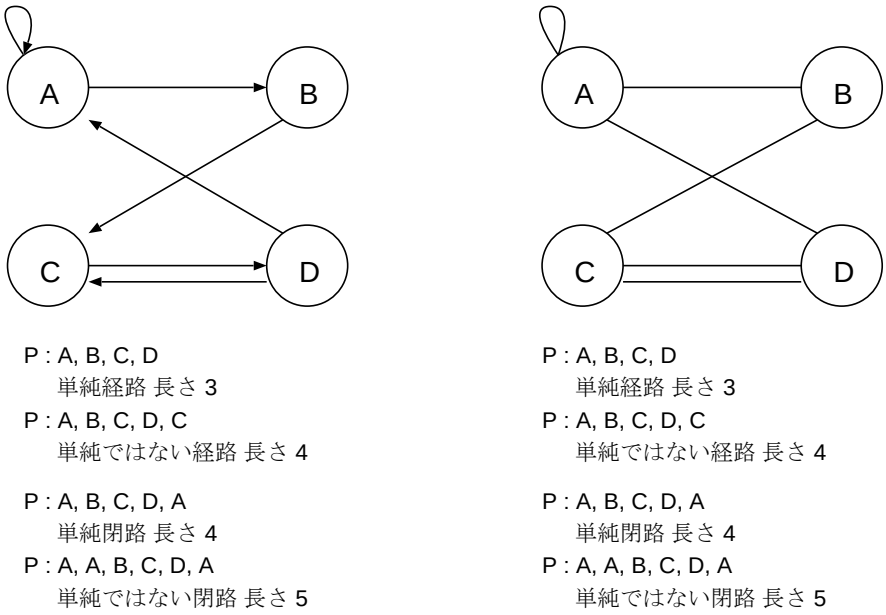
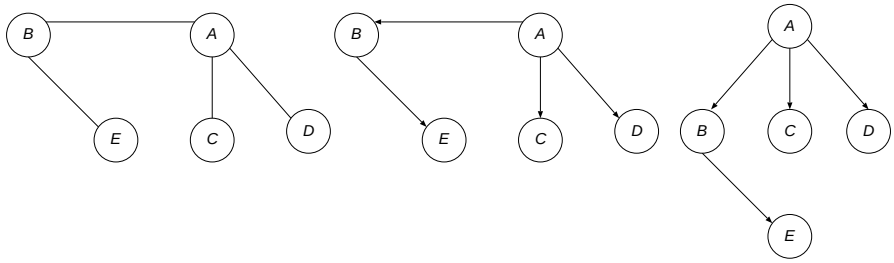


図 2.11　節点列の例

無向グラフ G において，任意の 2 点間に経路が存在するとき G は**連結**(connected) グラフと呼ばれる．無向な連結グラフにおいて閉路をもたないもの (acyclic) を**無向木**(undirected tree) という (図 2.12 の (a))．有向な連結グラフにおいて閉路をもたないのは**有向木**(directed tree) と呼ばれる (図 2.12 の (b)) が，特に**根**(root) と呼ばれる特殊な節点から任意の節点への経路が存在するときを**根付き木**(rooted tree) と呼ばれる (図 2.12 の (c))．ここでは，節点 A を根としている．この根と呼ばれる節点は入次数が 0 である．根付き木は有向グラフであるので，辺 (u, v) では節点 u を**親**(parent) といい，節点 v を**子**(child) という．子をもつ節点を**内部節点**(internal node) といい，子を持たない

節点を**外部節点**(external node) と呼ぶ．また，共通の親をもつ節点通しを**兄弟**(siblings) と呼ぶ．このときに左から右へ順番をつけるならば，最も左の子を**長男**(eldest brother) という．これからは，根付き木のみを説明するので，根付き木のことを単に**木**(tree) と呼ぶことにする．木 T において u と v は根からある節点への経路上にあるとする．節点 u が v より根に近いとき，u は v の**先祖**(ancestor) と言い，v は u の**子孫**(descendant) という．各節点は自分自身の先祖でも子孫でもあるとする．木 T の節点 u を根として，u の全ての子孫からなる木を T の**部分木**(subtree) という．木において木の辺数 $|E|$ と節点数 $|V|$ の間には，$|E| = |V| - 1$ の関係があり，点次数が 1 の節点を**葉**(leaf) という．木 T のある節点 u から葉までの最長路を u の**高さ**(height) と呼ぶ．特に**木の根の高さ**(tree height) をその木の高さという．また，根から u に至る経路の長さのことを u の**深さ**(depth) という．図 2.12 の (c) において節点 B の高さは 1，深さは 1，木 T の高さは 2 である．



(a) 無向木T (閉路なし)　(b) 有向木T (閉路なし)　(c) 根付き木T (木T) (閉路なし)

図 2.12　木の例

定義 2.3. 木は次のように再帰的に定義することができる [20].

1. 単一の節点はそれ自身を根とする木である．
2. n_1，n_2，$\cdots$，n_k を根とする木 T_1，T_2，$\cdots$，T_k があるとき，新しい節点 n を n_1，n_2，$\cdots$，n_k の親とすると，n を根とする木が得られる (図 2.13)．　□

ある一定の手順で木の全ての節点を訪れることを**木のなぞり**(treverse) という．木のなぞりでは，「訪れた節点をいつ表示するか」で出力順番が変わる．図 2.14 では，左部分木，中部分木，右部分木へと再帰的になぞっている．左部分木の方から右部分木へと訪れるの

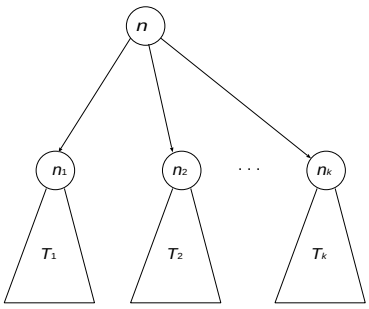


図 2.13　T_1,T_2,$\cdots$,T_k からの新しい木 T の構成

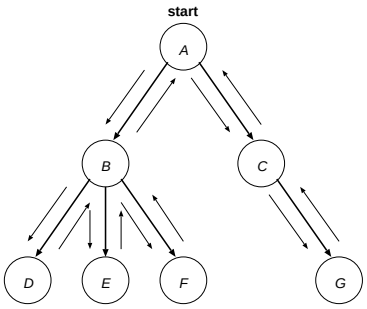


図 2.14　木のなぞり方

で，表示の仕方には次の 3 通りが考えられる．

- 前順 (preorder)
  1. 節点データの表示
  2. 左部分木をなぞる再帰呼出し
  3. 中部分木をなぞる再帰呼出し
  4. 右部分木をなぞる再帰呼出し

  リストの表示は，A，B，D，E，F，C，G となる．
- 中順 (inorder)
  1. 左部分木をなぞる再帰呼出し
  2. 節点データの表示
  3. 中部分木をなぞる再帰呼出し
  4. 右部分木をなぞる再帰呼出し

  リストの表示は，D，B，E，F，A，C，G となる．

1. 左部分木をなぞる再帰呼出し
2. 中部分木をなぞる再帰呼出し
3. 節点データの表示
4. 右部分木をなぞる再帰呼出し

リストの表示は，D，E，B，F，A，C，G となる．

- 後順 (postorder)
  1. 左部分木をなぞる再帰呼出し
  2. 中部分木をなぞる再帰呼出し
  3. 右部分木をなぞる再帰呼出し
  4. 節点データの表示

  リストは，D，E，F，B，G，C，A となる．

2.2 基本的な木構造

事前に格納されている多くの情報の中から望みの情報を引き出す操作を**探索**（searching）という．情報はいくつかの**レコード**（record）から構成され，各レコードは探索に使われる**キー**（key）を持っている．探索の目的は，与えられた**探索キー**（search key）と一致するキーを探し出すことである．[31]

2.3 2 分探索木

実生活では，階層構造をとる形態が多く存在する．このような階層構造は，木で表現することができる．その中でも，最も計算機で使われる木構造が 2 **分木**（binary tree）である．2 分木の定義を述べる [32].

定義 2.4. 2 分木 T とは次のいずれかの条件を満たす木である．

1. T はまったく節点を持たない空木である．
2. 根とよばれるただ 1 つの要素と，共通節点を持たない 2 つの 2 分木である左部分木と右部分木からなる．□

図 2.15 に 2 分木の例を示す．図 (a) と (b) の 2 分木をそれぞれ T_1 と T_2 とする．T_1 と T_2 の節点集合は，

$$T_1 = \{A, B, C, D, E, F, G\}$$
$$T_2 = \{A, B, C, D, E, F, G\}$$

である．集合 P が次の条件 1 〜 3 を満たすとき，P は**半順序集合**（partially ordered set）と呼ばれ，その要素は半順序関係を持っているという [48].

1. 任意の $x \in P$ に対して，$x \geq x$ である．
2. $x, y \in P$ で，$x \geq y$ かつ $y \geq x$ なら $x = y$ である．
3. $x, y, z \in P$ で，$x \geq y$ かつ $y \geq z$ なら $x \geq z$ である．
4. 任意の $x, y \in P$ に対して，$x \geq y$ または $y \geq x$ のどちらかが成り立つ．

半順序集合に属する 2 個の要素間では必ずしも順序づけられないが，任意の 2 個の要素をとってきても順序づけられるときを**全順序集合**（totally ordered set）と呼ぶ（条件 4 を加えた）．図 2.15 の (a) と (b) における 2 分木は，この全順序集合を満たすものである．ある節点の要素より小さいならば左部分木に，それよりも大きいならば右部分木に属するとする．そうすると，T_1 と T_2 では次の大小関係が成り立つ．

$$T_1 : B < F < A < C < G$$
$$T_2 : F < B < A < C < G$$

これから，図 (a) と (b) は異なる 2 分木である．全順序集合が満たされる図 (a) と (b) で

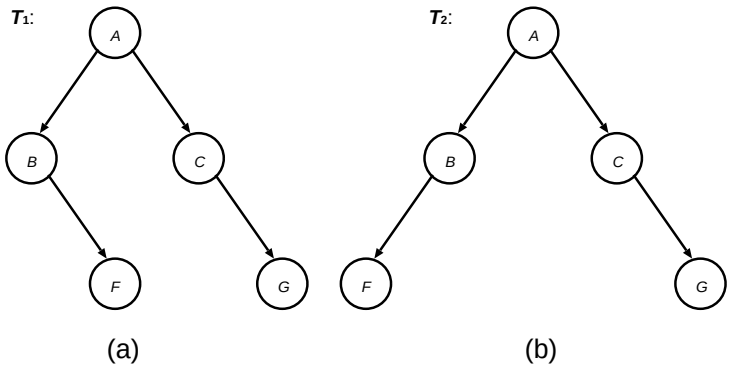


図 2.15　2 分木の例

は，根から葉に向かって，探索するデータと節点に置かれたデータと比較することで**探索**（searching）が可能となる．この意味で 2 **分探索木**（binary search tree）という．

通常の 2 分探索木では，節点に重みつき数字列を格納するが，文字列に対応した 2 分探索木のアルゴリズムとプログラムを作成する．文字列探索の代表的なアルゴリズムに基数探索法がある．基数探索法とは，一文字ごとにデータ同士の比較を行い，これを繰り返して整列する方法である．つまり，データ全体を一度に比較するのではなく，文字列の先頭から一桁ずつ比較する方法である．文字列である単語や句を文字列に分解すると，文字には整数や実数と同じように，全順序の性質が存在するので，同様に単語も全順序の性質（辞書式順序）を持たせることが可能である．すなわち，単語 $x = a_1a_2 \cdots a_k$ と $y = b_1b_2 \cdots b_l$ に対して，$a_1 < b_1$ のとき $x < y$ と定義する．さらに，ある i $(1 < i)$ に対して，$a_j = b_j (j = 1, 2, \cdots, i-1)$ かつ $a_i < b_i$ のとき，$x < y$ と定義する．ただし，空文字はどの文字よりも小さいとする [19]．すなわち，ある一桁が全順序集合であれば，この 2 分探索木を使って探索が可能となる．例えば，英字等である (計算機では一文字は，一意の番号となる)．数値の整列では，最大桁が分かれば右寄せでデータを配列に格納すれば利用可能となる．すなわち，文字列と数字列において利用可能である．

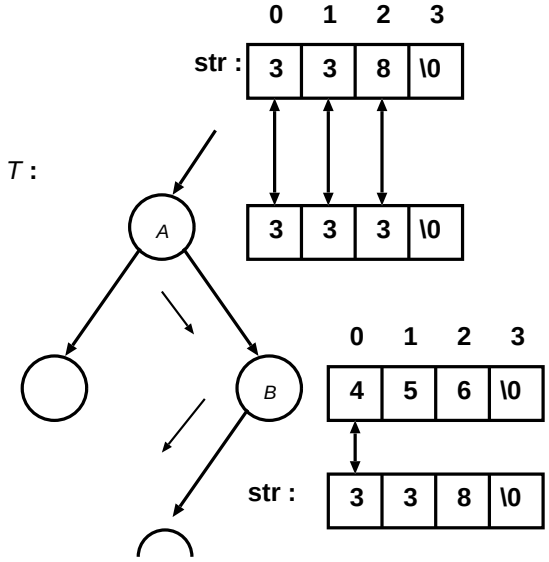


図 2.16　文字列に対応した 2 分探索木

図 2.16 において，探索したいデータを 338 とすると，節点 A の 333 とは 0 桁目，1 桁目と同じとなるが，2 桁目で探索データが大きいために節点 A の右部分木へと進行する．

節点 B に進行することになるが，再度 0 桁目から比較することになる．この結果，探索データの 0 桁目が小さいことから，節点 B の左部分木へ進行することになる．

2.3.1　2 分探索木のデータ構造

文字列に対応する 2 分探索木を構築する．2 分探索木のアルゴリズムを作成するために，データ構造を構築する．1 つの節点から 2 つの有向辺があるので，2 分探索木のデータ構造を構造体の形にすると次のように定義できる．

```
1  struct binary {
2    char element[S+1];
3    struct bin_node *left;
4    struct bin_node *right;
5  };
```

2 行目の配列 element に文字列を格納するが，この文字列を x とし，各文字は m 進数で S 桁とする．x の各文字を $a_i \in \mathbb{Z}$　$(0 \leq a_i < m,\ \ 0 \leq i \leq S-1)$ とすると x は，

$$x = a_0\, a_1 \cdots a_{S-1}$$

と表される．この x を 2 行目にある char 型の配列で保持する．$3 \cdot 4$ 行目は，左部分木を指すポインタと右部分木を指すポインタである．1 節点当たり必要となるメモリ量は，char 型は 1 バイトより，文字数 S と ‘\0’ を加えて，$S+1(=C)$ バイト，ポインタは 4 バイトであるから 8 バイト，合計で 1 節点当たり $(C+8)$ バイトとなる．

2.3.2　2 分探索木の操作

節点に格納されているデータを「節点データ」で，探索するデータを「探索データ」と呼ぶことにする．次の操作を行なうものとする．

- 探索
- 挿入
- 削除

- リストをある順番で印字

T における各操作を説明するが，数字はアルゴリズムを実行する順番，英字は場合分けを示している．いずれも入れ子の構造をとる．また，アルゴリズムの中で次の数学記号を用いる．

- '>=' は $\geq$ を表す.
- '<=' は $\leq$ を表す.
- '!=' は $\neq$ を表す.
- '&&' は 『かつ』を表す.
- '||' は 『または』を表す.
- '->' は 『ならば』を表し，左辺が成立していれば右辺の状態に移動する．

探索

T の根節点から探索を始めて，探索データの，ある 1 桁の値が節点データのそれよりも小さい値ならば左部分木に進行して，大きい値ならば右部分木へ進行する．この進行を繰り返すことで，探索データと等しい節点データが見つかれば，探索データが T に存在したことになる．逆に節点が無くなれば，探索データは T に存在しないことになる．これを探索のアルゴリズムとする．

以下に探索のアルゴリズムを示す.

添字を 0 として，T の根から始める．

1　a　**T に探索する部分木があるならば，2 を実行する．**

　　b　**T に探索する部分木がないならば，4 を実行する．**

2　a　**探索データの添字の桁の値 < 節点データの添字の桁の値**

　　-> 左部分木へ進行し，添字を 0 として，1 から実行する．

　　b　**探索データの添字の桁の値 > 節点データの添字の桁の値**

　　-> 右部分木へ進行し，添字を 0 として，1 から実行する．

　　c　**探索データの添字の桁の値 = 節点データの添字の桁の値**

-> 3 を実行する．

3　a　探索データの添字が文字列の終端である．

-> この節点が探索データであり，探索終了

b　探索データの添字が文字列の終端でない．

-> 添字を 1 加算して，1 から実行する．

4　T に探索データは存在しない． -> 探索終了

挿入

T を探索して該当の文字列が存在しなければ，新しい葉を作成する．すなわち，探索のアルゴリズムを利用して，T に節点がなくなれば，葉を作成する．逆に T に探索データが存在すれば，葉を作成しない．葉を作成する様子を図 2.17 に示す．

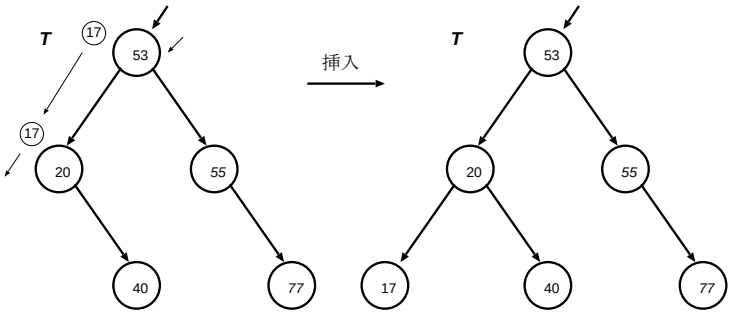


図 2.17　葉の作成

削除

削除データが T に「存在する」又は「存在しない」かは，探索のアルゴリズムを利用する．T に削除したい節点データが存在しても，そのまま削除することはできない．この節点には子節点が存在する場合があるからである．この状態を図 2.18 に示す．削除する節点を灰色，子である部分木を三角形で示している．

図 2.18(a) の子が存在しない場合は，そのまま節点 (葉) を削除すれば良い．この様子を図 2.19 に示す．

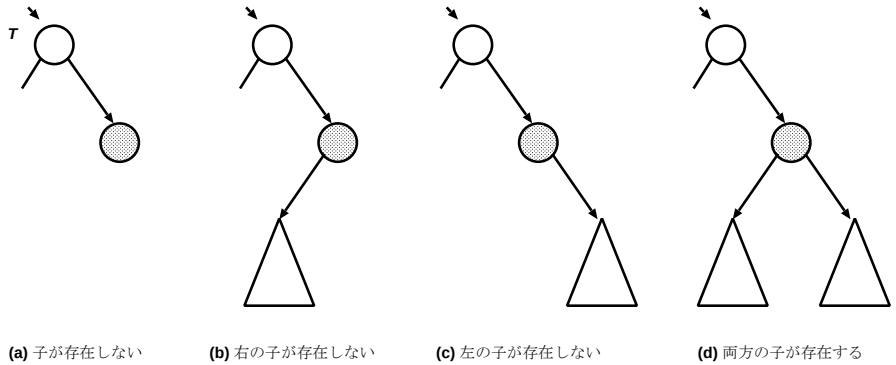


図 2.18　削除のパターン

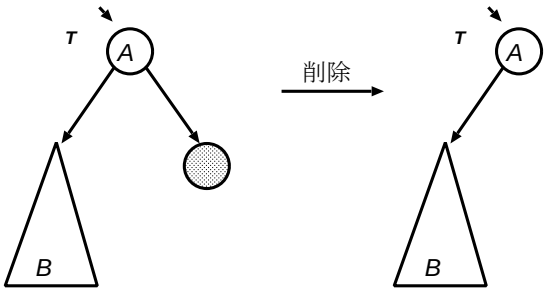


図 2.19　葉の削除

図 2.18(b) と (c) の子が一方側だけに存在する場合は，削除する節点の位置へ子を移動させれば良い．この様子を図 2.20 に示す．

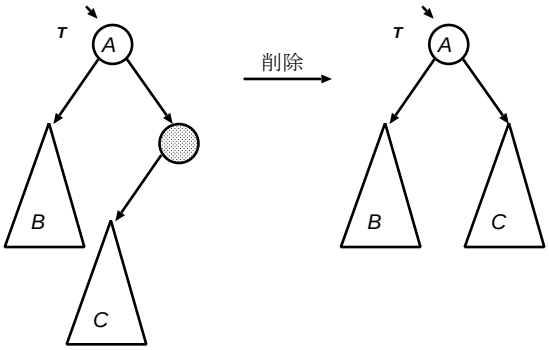


図 2.20　子が一方だけ存在するときの削除

図では，削除する節点の左部分木が存在する場合を示しているが，右部分木のみが存在する場合も同様に扱える．

図 2.18(d) の子が左 · 右部分木ともに存在する場合は，削除したい節点位置に右部分木

で最小のデータをもつ節点 (左部分木で最大のデータをもつ節点でも良い) を挿入すれば良い．この様子を図 2.21 に示す．

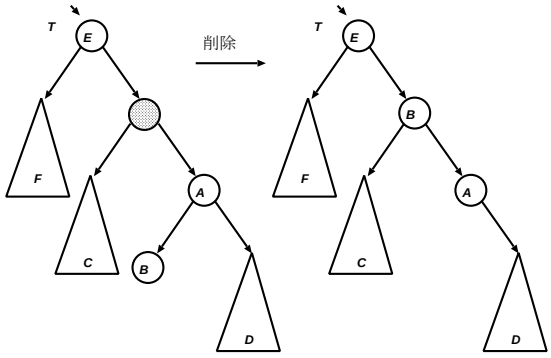


図 2.21　子が両方とも存在するときの削除

削除の 4 パターンを確認したが，このパターンを組み合わせてみる．どのパターンになるかを調べるアルゴリズムを場合分けのアルゴリズムと名付け，この様子を図 2.22 で示す．図の (a) の灰色の節点は，削除する節点であるが，この節点には 2 つの子が存在している．よって，右部分木の最小データをもつ節点である A を削除する位置へ移動する．次の (b) では，元の A の節点が削除されたとする．この元の A には 1 つの子が存在していたので，B を削除する位置へ移動する．次の (c) では，元の B の節点が削除されたとする．この元の B は，葉なので削除するだけで良い．これを削除したものが，(d) である．つまり，このアルゴリズムは必ず葉の削除で終了となる．

場合分けのアルゴリズムを示す．

1　**削除する節点の子節点の数により，削除方法を変える．**

- a　**節点の子が存在しない． -> この節点を削除して，終了**
- b　**節点の左の子が存在しない． -> 子節点を節点の位置に移動する．移動した節点を削除するとして，1 から実行する．**
- c　**節点の右の子が存在しない． -> 子節点を節点の位置に移動する．移動した節点を削除するとして，1 から実行する．**
- d　**節点の子は共に存在する． -> 節点の右部分木から最小となる節点を移動する．移動した節点を削除するとして，1 から実行する．**

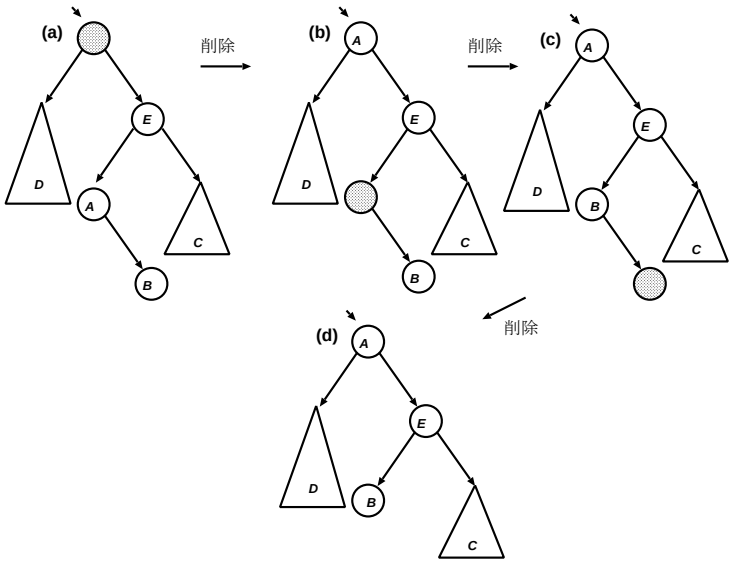


図 2.22　削除の方法

2.3.3　リストをある順番で印字

ある節点の左部分木には節点より小さい値が集まり，右部分木には節点より大きい値が集まる．よって，第 1 章の図 2.14 で説明した中順を適用すれば，リストを昇順に出力することができる．図 2.14 では 3 分木を用いて説明したが，2 分探索木では次のアルゴリズムで昇順で出力される．

1. 左部分木をなぞる再帰呼出し
2. 節点データの表示
3. 右部分木をなぞる再帰呼出し

これをなぞりのアルゴリズムとする．

なぞりのアルゴリズムを以下に示す．

T の根を u とする．

1　a　**節点 u が存在する．** -> 2 **を実行する．**

　　b　**節点 u が存在しない．** -> 5 **を実行する．**

2　u の左部分木を u として，スタックに，戻りアドレスを‘PUSH’ する．
　　-> 1 から実行する．

3　u の節点データを印字する．

4　u の右部分木を u として，スタックに，戻りアドレスを‘PUSH’ する．
　　-> 1 から実行する．

5　スタックから，戻りアドレスを‘POP’ する．
　　もし，スタックが空である． -> なぞり終了

2.3.4　2 分探索木の時間計算量

定理 2.1. データ数 n である 2 分探索木の探索，挿入，削除の時間計算量は，$O(\log n)$ となる．

証明. 2 分探索木では，探索，挿入，削除のどの操作についても，「探索」の操作を行う．各操作において，この後に行われるアルゴリズムは定数オーダであるから，「探索」が時間計算量となる．「探索」は比較の手続きであり，最大の比較回数は ‘木の高さ $+1$’ である．つまり，木の高さを求めればよい．O 記法では，定義より実行時間が最悪の場合を求めれば良かったので，総データ数が n 個，木の形状が線形となる場合では，データ数 1 個当たり $O(n)$ となる．ここでは，木に挿入されるデータ数を n 個として，平均の比較回数 (=木の高さ) を求める．すなわち，n!個の順列が等確率で起こると仮定して，平均比較回数 $C(n)$ を求める．リストの先頭データが根に挿入されるが，リストの全てのデータが挿入されるまでに，根では $(n-1)$ 回の比較が行われる．また，根のデータは仮定より，リストから等確率で選ばれることになるが，先頭のデータがリスト中で i 番目の大きさならば，この根の左部分木には $(i-1)$ 個の節点があり，右部分木には $(n-i)$ 個の節点が含まれることになる．すなわち，n による $C(n)$ は，

$$C(n) = \frac{1}{n}\sum_{i=1}^{n}\left(C(i-1) + C(n-i) + n - 1\right) \tag{2.1}$$

である．式 (2.1) において，

$$\sum_{i=1}^{n} C(i-1) + \sum_{i=1}^{n} C(n-i) = 2\sum_{i=0}^{n-1} C(i)$$

であるから，式 (2.1) は，

$$n\,C(n) = 2\sum_{i=0}^{n-1} C(i) + n(n-1) \tag{2.2}$$

式 (2.2) において，n を $n-1$ とおいて，

$$(n-1)\,C(n-1) = 2\sum_{i=0}^{n-2} C(i) + (n-1)(n-2) \tag{2.3}$$

式 (2.2) − 式 (2.3) をすると，

$$n\,C(n) - (n+1)\,C(n-1) = 2(n-1)$$

すなわち，

$$n\,C(n) - (n+1)\,C(n-1) = 4n - 2(n+1) \tag{2.4}$$

式 (2.4) を $n(n+1)$ で割ると，

$$\frac{C(n)}{n+1} = \frac{4}{n+1} - \frac{2}{n} + \frac{C(n-1)}{n} \tag{2.5}$$

式 (2.5) において，$n = 1, 2, \cdots, n$ について加え合わせていくと，

$$\begin{aligned}
\frac{C(n)}{n+1} &= 4\sum_{i=2}^{n} \frac{1}{i} - 2\sum_{i=1}^{n} \frac{1}{i} + \frac{4}{n+1} \\
&= 4\sum_{i=1}^{n} \frac{1}{i} - 2\sum_{i=1}^{n} \frac{1}{i} - 4 + \frac{4}{n+1} \\
&= 2\sum_{i=1}^{n} \frac{1}{i} - 4 + \frac{4}{n+1}
\end{aligned}$$

となるから，

$$C(n) = 2(n+1)\sum_{i=1}^{n} \frac{1}{i} - 4(n+1) + 4 \tag{2.6}$$

となる．ここで，m $(0 < m \in \mathbb{R})$ は $m < i < m+1$ において，

$0 < 1/(m+1) < 1/i < 1/m$ となるから，各々を区間 $[m, m+1]$ で i について定積分を行うと，

$$\begin{aligned} \int_m^{m+1} \frac{1}{m+1}\, di &= \frac{1}{m+1} \\ \int_m^{m+1} \frac{1}{i}\, di &= \log(m+1) - \log m \\ \int_m^{m+1} \frac{1}{m}\, di &= \frac{1}{m} \end{aligned} \tag{2.7}$$

式 (2.7) で $m = 1, 2, \cdots, n-1$ について加え合わせてみると，

$$\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \log n < 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1}$$

を得るので，

$$\sum_{i=1}^{n} \frac{1}{i} \simeq \log n \tag{2.8}$$

式 (2.6) と式 (2.8) より，

$$\begin{aligned} C(n) &= 2(n+1) \sum_{i=1}^{n} \frac{1}{i} - 4(n+1) + 4 \\ &= O(n \log n) \end{aligned}$$

データ数 n 個当たりのオーダを求めたので，データ数 1 個当たりでは，探索，挿入，削除の時間計算量は，$O(\log n)$ となる．　□

2.4　AVL 木

2.3.4 節で，データ数 n の 2 分探索木の高さの期待値は $O(\log n)$ であるが，最悪の場合では $O(n)$ となる．最悪の比較回数は葉に辿り着く場合から，木 T の高さを低くすれば良いことになる．T の高さを低くするためには，T の高さを $h\,(h \geq 1)$ としたときに，深さ $(h-1)$ 以下の全ての節点が 2 個の子をもつことが望ましい (第 4 章の定義 4.1)．少し条件を緩めて $h \geq 2$ のとき，深さが $(h-2)$ 以下の全ての節点は必ず 2 個の子を持

ち，根から全ての葉に至る深さが，高々 1 だけ異なる T を構築する．この T は**完全平衡木**(completely balanced tree) と呼ばれる．しかし，ある操作が行なわれるたびに完全平衡木に再構築すると逆に時間が掛かるおそれがある．そこで，更に条件を緩めて平衡条件を定義する [32].

定義 2.5. 平衡状態にある 2 分木とは，次の条件を満たす木である．

- 高さ平衡とは，どの節点においても左右の部分木の高さの差が一定以内のことである．
- 重さ平衡とは，どの節点においても左右の部分木の節点数の差が一定以内のことである．□

ここで紹介する AVL 木は，Adelson-Velskii と Landis が提案した高さ平衡木である．AVL 木の平衡条件を定義する [48].

定義 2.6. AVL 木とは，次の条件を満たす 2 分木である．

木のどの節点についても，その左部分木と右部分木の高さの差は高々 1 しか違わない．□

AVL 木は 2 分木の定義により木を構築して，その後，ある部分木でその左部分木と右部分木の高さの差が 1 より大きくなったら，回転操作を行う．回転の詳細は後で説明するが，図 2.23 に AVL 木の例を示す．

この図は，2.3 節の 2 分探索木と同様に，節点のデータよりも小さい値であれば左部分木へ，大きい値ならば右部分木へ進む．(a) では，データ 21 が挿入されたことで，その親のデータ 20 の節点で条件に従っているかを考察する．20 の節点では，右部分木が左部分木よりも高さが 1 高いが，条件に従っている．20 を根とする部分木では，高さが 1 増加したので，データ 22 の節点で条件に従っているかを考察する．このように，葉から根に向かうことを伝播という．22 の節点では，左部分木の高さが右部分木のそれよりも 2 高いので，回転を行う．回転後が (b) であるが，2 分探索木の条件を維持し，かつ部分木の高さが 1 低くなった．

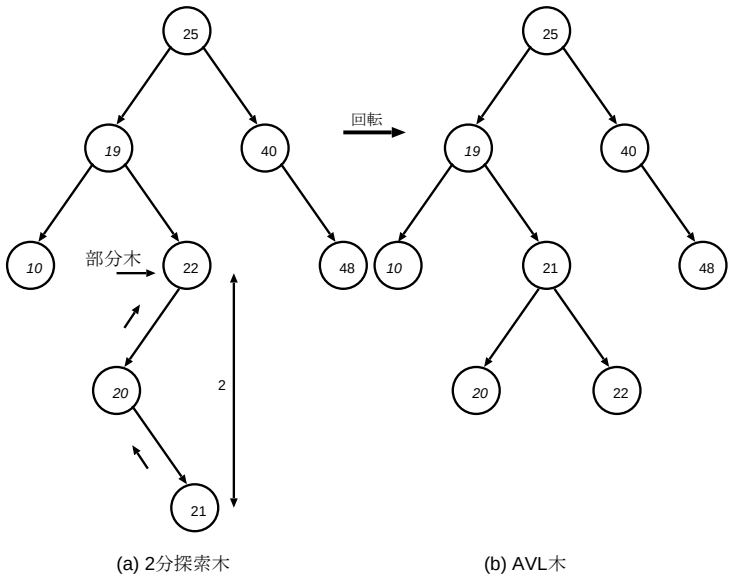


図 2.23　AVL 木の例

2.4.1　AVL 木のデータ構造

2.3 節の 2 分探索木と同様に文字列に対応する AVL 木を構築する．AVL 木のアルゴリズムを作成するために，データ構造を構築する．節点 1 つ当たりのデータ構造は 5 行目からとなるが，7, 8 行目の enum sub 型と enum diff 型を先に説明する．1 行目の enum sub 型は，節点が親の節点に対して，どの部分木であるかを保持するものである．節点が木の根であれば ‘RO’ であり，節点が親の節点に対して左部分木であれば ‘LT’ であり，節点が親の節点に対して右部分木であれば ‘RT’ を保持する．この集合を enum sub 型として定義している．2 行目の enum diff 型は，ある節点から見たときの高さの高い部分木の方を保持するための集合である．変数に ‘EVEN’ が格納されたときは，左右部分木の高さが同じであり，‘LEFT’ で左部分木が右部分木よりも高く，‘RIGHT’ では右部分木が左部分木よりも高いことを表す．この集合を enum diff 型として定義している．AVL 木のメンバを 6〜11 行目に示す．6 行目でデータを S 桁の文字列で保持する．これは，2 分探索木と同様である．9, 10 行目も 2 分探索木のデータ構造と同じとなるが，7, 8, 11 行目が新たに追加される．7, 8 行目は先に述べた通りであり，11 行目では回転のために必要となる節点から親の節点へのポインタを保持する．1 節点当たり必要となるメモリ量は，char 型

は 1 バイトより $S+1(=C)$ バイト，ポインタは 4 バイトより 12 バイト，列挙型が 4 バイトより 8 バイト，合計で 1 節点当たり $C+20$ バイトとなる．

```
 1  enum sub {RO, LT, RT};
 2  enum diff {EVEN, LEFT, RIGHT};
 3
 4  // AVL_tree のデータ構造
 5  struct avl {
 6    char element[S+1];
 7    enum sub sub_tree;
 8    enum diff diff;
 9    struct avl *left;
10    struct avl *right;
11    struct avl *parent;
12  };
```

2.4.2 木の高さによる計算量

第 4 章の式 （4.1）から，木の高さによる計算量を求める．高さ h の完全 2 分木では節点数を n とすると，

$$n = 2^{h+1} - 1 \tag{2.9}$$

となる．式 (2.9) で，両辺の 2 を底とする対数をとると，

$$\log_2(n+1) = h+1$$

すなわち，

$$h \simeq \log_2 n$$

である．これより，木の高さによる計算量が $O(\log n)$ になることは，優位性を示している．

定理 2.2. データ数 n である AVL 木の高さは，$O(\log n)$ である．

証明. AVL 木を満たす高さ h のときの最小のデータ数を $f(h)$ とおく．定義より，左部分木は右部分木より高さが 1 だけ高いとすると，漸化式 $f(h)$ は，

$$\begin{aligned} &f(h) = f(h-1) + f(h-2) + 1 \quad (h \geq 2) \\ &f(0) = 1,\ f(1) = 2 \end{aligned} \tag{2.10}$$

という式が得られる．式 (2.10) で $f(h) = F(h) - 1$ とおくと，

$$\begin{aligned} &F(h) = F(h-1) + F(h-2) \quad (h \geq 2) \\ &F(0) = 2,\ F(1) = 3 \end{aligned} \tag{2.11}$$

という式が得られる．一方，$x^2 - x - 1 = 0$ の解を ϕ_1, ϕ_2 とすると，解と係数の関係より，

$$\begin{aligned} &\phi_1 + \phi_2 = 1 \\ &\phi_1 \cdot \phi_2 = -1 \end{aligned} \tag{2.12}$$

となる．式 (2.12) を式 (2.11) に代入すると，

$$F(h) - (\phi_1 + \phi_2) \cdot F(h-1) + \phi_1\phi_2 \cdot F(h-2) = 0 \tag{2.13}$$

となる．式 (2.13) を移行すると，

$$F(h) - \phi_1 F(h-1) = \phi_2 \cdot \{F(h-1) - \phi_1 \cdot F(h-2)\} \tag{2.14}$$

$$F(h) - \phi_2 F(h-1) = \phi_1 \cdot \{F(h-1) - \phi_2 \cdot F(h-2)\} \tag{2.15}$$

の 2 つの式が得られる．式 (2.14) は，

$$F(h) - \phi_1 \cdot F(h-1) = (3 - 2\phi_1)\phi_2^{h-1} \tag{2.16}$$

であり，式 (2.15) は，

$$F(h) - \phi_2 \cdot F(h-1) = (3 - 2\phi_2)\phi_1^{h-1} \tag{2.17}$$

となる．式 (2.16) $\times\phi_2$ − 式 (2.17) $\times\phi_1$ とすると，1 桁当たり節点数が m 個あれば良い．

$$F(h) \cdot (\phi_2 - \phi_1) = (3 - 2\phi_1) \cdot \phi_2^h - (3 - 2\phi_2) \cdot \phi_1^h \tag{2.18}$$

である．ここで，

$$\phi_1 = \frac{1+\sqrt{5}}{2}$$
$$\phi_2 = \frac{1-\sqrt{5}}{2}$$

とすると，

$$3 - 2\phi_1 = 2 - \sqrt{5} = \left(\frac{1-\sqrt{5}}{2}\right)^3 = \phi_2^3$$
$$3 - 2\phi_2 = 2 + \sqrt{5} = \left(\frac{1+\sqrt{5}}{2}\right)^3 = \phi_1^3$$
$$\phi_2 - \phi_1 = -\sqrt{5}$$

であり，これらを式 (2.18) に代入すると，

$$f(h) = \frac{1}{\sqrt{5}}(\phi_1^{h+3} - \phi_2^{h+3}) - 1 \tag{2.19}$$

が，得られる．高さ h のときの AVL 木に挿入できるデータ数 n とおくと，

$$\frac{1}{\sqrt{5}}(\phi_1^{h+3} - \phi_2^{h+3}) - 1 \leq n \tag{2.20}$$

となる関係式が得られ，

$$\phi_1^{h+3} - \phi_2^{h+3} \leq \sqrt{5}(n+1) \tag{2.21}$$

となる．式 (2.21) の左辺の第 2 項の絶対値は 1 より小さいので，十分大きな h に対しては，

$$\phi_1^{h+3} \leq \sqrt{5}(n+1) \tag{2.22}$$

が成り立つ．式 (2.22) において，底 > 1 の対数をとると，

$$(h+3)\log\phi_1 \leq \log\sqrt{5} + \log(n+1)$$

より，

$$\begin{aligned}\therefore\ h &\leq \frac{\log\sqrt{5}}{\log\phi_1} + \frac{\log(n+1)}{\log\phi_1} - 3 \\ &= O(\log n)\end{aligned}$$

これより，データ数 n である AVL 木の高さは，$O(\log n)$ である． □

AVL 木は条件が緩くなった完全平衡木であるが，それと比べてどのくらいの高さの差が生じるか求める．

性質 2.1. AVL 木の高さは完全平衡木の高さの約 1.44 倍よりも小さい．

証明. 高さ h のときで最も 2 分木の高さが高くなるときのデータ数は，$(2^{h+1}-1)$ 個であるから式 (2.20) より，十分大きな h においては，

$$\frac{1}{\sqrt{5}}\phi_1^{h+3} - 1 < n \leq 2^{h+1} - 1$$

の関係式が成り立つ．すなわち，

$$\frac{1}{\sqrt{5}}\phi_1^{h+3} - 1 < n \tag{2.23}$$

$$n \leq 2^{h+1} - 1 \tag{2.24}$$

を h について解くことで，AVL 木の大体の高さが分かる．式 (2.23)，(2.24) は，

$$\begin{aligned}h &< \frac{\log_{10} 2 \cdot \log_2 \sqrt{5}(n+1)}{\log_{10}\phi_1} - 3 \\ h &\geq \log_2(n+1) - 1\end{aligned}$$

と，それぞれなるから，

$$\log_2(n+1) \leq h + 1 < \frac{\log_{10} 2 \cdot \log_2 \sqrt{5}(n+1)}{\log_{10}\phi_1} - 2$$

である．ここで，

$$\frac{\log_{10} 2 \cdot \log_2 \sqrt{5}(n+1)}{\log_{10}\phi_1} - 2 \ \simeq\ 1.44\log_2(n+1) - 0.328$$

となるから,

$$\therefore \log_2(n+1) \leq h+1 < 1.44\log_2(n+1) - 0.328$$

これより，AVL 木の高さは完全平衡木の高さの約 1.44 倍よりも小さく抑えられる． □

2.4.3 AVL 木の操作

用語と各操作は，2.3.2 節の 2 分探索木と同じである．

探索

探索の方法は，2.3.2 節と同じである．

挿入

挿入の方法は，2.3.2 節と同じである．

回転

挿入のアルゴリズムで葉を作成した場合は，回転の手続きを行う．回転の手続きとは，以下の手順である．

1 **葉を作成したら，AVL 木の条件の考察を行う．**

- a **条件を満たしていなければ，2 を実行する．**
- b **条件を満たしている． -> 回転終了**

2 **再構築方法の選別を行う．**

- a **一重回転 -> 回転終了**
- b **二重回転 -> 回転終了**

AVL 木の条件の考察

挿入後，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,「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．AVL 木の条件を満たしていなければ，回転を行うことで AVL 木の条件を満たすことになる．

T の節点 y において，その左部分木 T_L と右部分木 T_R に関する状態を

$$s(y) = \begin{cases} \text{EVEN}, & (T_L\text{の高さ}) = (T_R\text{の高さ}) \\ \text{LEFT}, & (T_L\text{の高さ}) > (T_R\text{の高さ}) \\ \text{RIGHT}, & (T_L\text{の高さ}) < (T_R\text{の高さ}) \end{cases} \tag{2.25}$$

で定める．この状態をデータ構造の第 3 メンバで保存する．

今，部分木において，左の子 v からその親 u へ遡ってきた場合を考える．すなわち，v を根とする部分木の高さが 1 増えた状態である．v の高さが増えないときには，これ以上の考察は必要なくなる．u の第 3 メンバの値によって，次の 3 つの場合に分けられる．

- $s(u) = \text{RIGHT}$ の場合: 左部分木の高さが 1 増えたため，$s(u) = \text{EVEN}$ に変わる．u を根とする部分木の高さは変化しないので，考察はこれで終了とする．
- $s(u) = \text{EVEN}$ の場合: 左部分木の高さが 1 増えたため，$s(u) = \text{LEFT}$ に変わる．u を根とする部分木の高さが 1 増えるので，u の親に遡ることになる．u が根ならば，考察はこれで終了とする．
- $s(u) = \text{LEFT}$ の場合: u は AVL 木の条件を満たさなくなり，再構築方法の選別を呼び出す．

同様に，右の子からの場合も対称的に扱える．これを考察のアルゴリズム (挿入) とする．

以下に考察のアルゴリズム (挿入) を示す．節点 u は節点 v の親とする．

```
1   a  v が T の根である． -> 考察終了
    b  v が T の根でないならば，2 を実行する．

2   a  v が左部分木である． -> 3 を実行する．
    b  v が右部分木である． -> 4 を実行する．

3   a  s(u) = RIGHT -> s(u) = EVEN にして，考察終了
    b  s(u) = LEFT -> 再構築方法の選別を行なう．
    c  s(u) = EVEN -> s(u) = LEFT として，u を v，u の親を u として，1 から実
```

行する．

4 a `s(u) = LEFT ->` `s(u) = EVEN` **にして，考察終了**

b `s(u) = RIGHT ->` **再構築方法の選別を行なう．**

c `s(u) = EVEN ->` `s(u) = RIGHT` **として，**`u` **を** `v`**，**`u` **の親を** `u` **として，**`1` **から実行する．**

再構築方法の選別

考察のアルゴリズム (挿入) より，回転操作が必要な場合は「再構築方法の選別」を行なう．これを再構築のアルゴリズムとする．このアルゴリズムでは回転の種類を選別している．

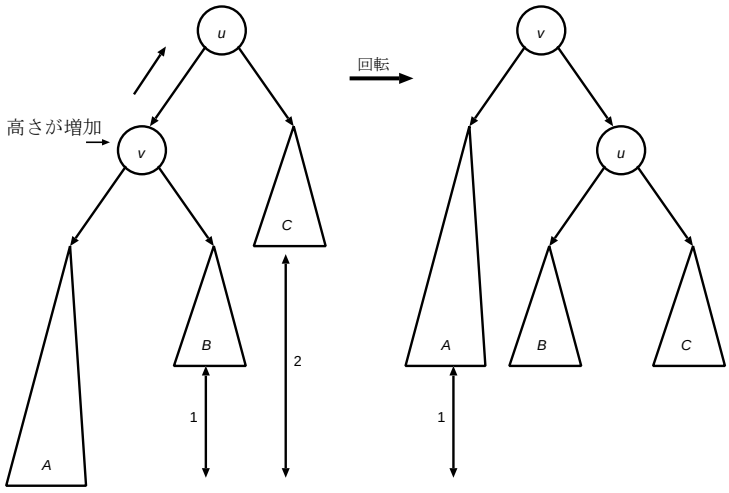


図 2.24 挿入における一重回転

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 2.24 と図 2.25 に示す．図 2.24 の (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 高くなった場合である．図 2.25 の (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 高くなった場合である．どちらの図でも，(a) の回転前では，u の左部分木の方が右部分木よりも高さが 2 高いが，(b) の回転後では，部分木の根では左部分木と右部分木の差はない．この操作で，(b) の部分木の高さは (a) の部分木のそれよりも 1 低くなり，最

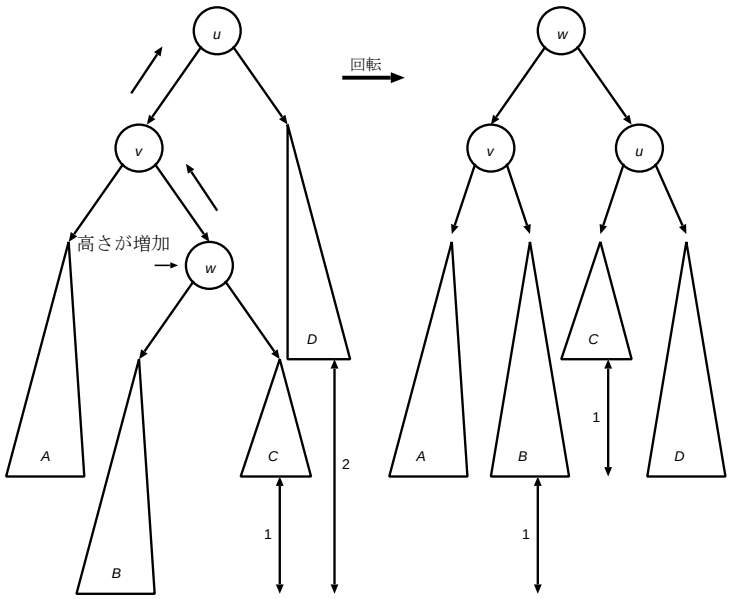


図 2.25　挿入における二重回転

悪の比較回数が 1 少なくなる．また，(b) の部分木の高さは挿入前と変わらないので，1 回の再構築だけで良いことになる．

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．

以下に再構築のアルゴリズムを示す．節点 u は節点 v の親とする．

1　a　`s(u) = LEFT ->` 2 **を実行する．**

　　b　`s(u) = RIGHT ->` 3 **を実行する．**

2　a　`s(v) = LEFT ->` **一重回転を行なう．**

　　b　`s(v) = RIGHT ->` **二重回転を行なう．**

3　a　`s(v) = RIGHT ->` **一重回転を行なう．**

　　b　`s(v) = LEFT ->` **二重回転を行なう．**

一重回転

一重回転では，図 2.24 のように節点 u と節点 v のポインタを付けかえる．これを一重回転のアルゴリズムとする．

一重回転のアルゴリズムを以下に示す．

```
1   a  u では左部分木の方が高い． -> 2 を実行する．
    b  u では右部分木の方が高い． -> 3 を実行する．

2   次の 1 ～ 3 により，子を指すポインタを付けかえる．
    1  v の右部分木である部分木 B を u の左部分木とする．
    2  u を v の右部分木とする．
    3  v を u の親の子とする．
    次の 4 ～ 6 により，親を指すポインタを付けかえる．
    4  u の左部分木である部分木 B が存在するならば，u を B の親とする．
    5  u の親を v の親とする．
    6  v を u の親とする．
    次の条件により，どちらの部分木が高いかを設定する．
    a  s(v) = LEFT
       1  s(v) = EVEN
       2  s(u) = EVEN
    b  s(v) = EVEN
       1  s(v) = RIGHT
       2  s(u) = LEFT
    次の 7 ～ 9 により，親に対して，どの部分木であるかを設定する．
    7  u がどの部分木であるかを v に代入する．
    8  u を右部分木とする．
    9  u の左部分木である部分木 B が存在するならば，B を左部分木とする．
   10  回転終了
```

3　**次の** 1 **〜** 3 **のより，子を指すポインタを付けかえる．**

1　v **の左部分木である部分木** B **を** u **の右部分木とする．**

2　u **を** v **の左部分木とする．**

3　v **を** u **の親の子とする．**

次の 4 **〜** 6 **により，親を指すポインタを付けかえる．**

4　u **の右部分木である部分木** B **が存在するならば，**u **を** B **の親とする．**

5　u **の親を** v **の親とする．**

6　v **を** u **の親とする．**

次の条件により，どちらの部分木が高いかを設定する．

a　s(v) = RIGHT

1　s(v) = EVEN

2　s(u) = EVEN

b　s(v) = EVEN

1　s(v) = LEFT

2　s(u) = RIGHT

次の 7 **〜** 9 **により，親に対して，どの部分木であるかを設定する．**

7　u **がどの部分木であるかを** v **に代入する．**

8　u **を左部分木とする．**

9　u **の右部分木である部分木** B **が存在するならば，**B **を右部分木とする．**

10　**回転終了**

二重回転

二重回転では，図 2.25 のように節点 u，節点 v 及び節点 w のポインタの付けかえる．これを二重回転のアルゴリズムとする．

以下に二重回転のアルゴリズムを示す．

1　a　u **では左部分木の方が高い．** -> 2 **を実行する．**

b　u **では右部分木の方が高い．** -> 3 **を実行する．**

2 **次の** 1 **〜** 5 **により，子を指すポインタを付けかえる．**

1 w **の左部分木である部分木** B **を** v **の右部分木とする．**

2 w **の右部分木である部分木** C **を** u **の左部分木とする．**

3 v **を** w **の左部分木とする．**

4 u **を** w **の右部分木とする．**

5 w **を** u **の親の子とする．**

次の 6 **〜** 10 **により，親を指すポインタを付けかえる．**

6 u **の親を** w **の親とする．**

7 w **を** v **の親とする．**

8 w **を** u **の親とする．**

9 v **の右部分木である部分木** B **が存在するならば，**v **を** B **の親とする．**

10 u **の左部分木である部分木** C **が存在するならば，**u **を** C **の親とする．**

次の条件と 11 **により，どちらの部分木が高いかを設定する．**

a s(w) = LEFT

1 s(v) = EVEN

2 s(u) = RIGHT

b s(w) = RIGHT

1 s(v) = LEFT

2 s(u) = EVEN

c s(w) = EVEN

1 s(v) = EVEN

2 s(u) = EVEN

11 s(w) = EVEN

次の 12 **〜** 15 **により，親に対して，どの部分木であるかを設定する．**

12 u **がどの部分木であるかを** w **に代入する．**

13 u **を右部分木とする．**

14 v **の右部分木である部分木** B **が存在するならば，**B **を右部分木とする．**

15 u **の左部分木である部分木** C **が存在するならば，**C **を左部分木とする．**

16 **回転終了**

3　次の 1 〜 5 により，子を指すポインタを付けかえる．

1　w の左部分木である部分木 B を u の右部分木とする．

2　w の右部分木である部分木 C を v の左部分木とする．

3　u を w の左部分木とする．

4　v を w の右部分木とする．

5　w を u の親の子とする．

次の 6 〜 10 により，親を指すポインタを付けかえる．

6　u の親を w の親とする．

7　w を v の親とする．

8　w を u の親とする．

9　u の右部分木である部分木 B が存在するならば，u を B の親とする．

10　v の左部分木である部分木 C が存在するならば，v を C の親とする．

次の条件と 11 により，どちらの部分木が高いかを設定する．

a　s(w) = LEFT

1　s(u) = EVEN

2　s(v) = RIGHT

b　s(w) = RIGHT

1　s(v) = EVEN

2　s(u) = LEFT

c　s(w) = EVEN

1　s(u) = EVEN

2　s(v) = EVEN

11　s(w) = EVEN

次の 12 〜 15 により，親に対して，どの部分木であるかを設定する．

12　u がどの部分木であるかを w に代入する．

13　u を左部分木とする．

14　u の右部分木である部分木 B が存在するならば，B を右部分木とする．

15　v の左部分木である部分木 C が存在するならば，C を左部分木とする．

16 **回転終了**

削除

節点データの削除は，2 分探索木の 2.3.2 節と同様に行う．その後で，AVL 木の条件を満たしているかを確認する．満たしていなければ，回転操作を行うが，挿入と違い 1 回の回転操作で終わりとは限らない．すなわち，再構築を行うと，更に部分木の高さが低くなる場合があり，このような場合は T の根に向かって再構築を繰り返す．これを削除の手続きとする．

削除の手続きを以下に示す．

1 **節点データを調べる（探索のアルゴリズム）．**

a **探索データが存在すれば，2 を実行する．**

b **探索データが存在しない． -> 削除終了**

2 **2 分探索木の条件を満たすように，節点データを削除する（場合分けのアルゴリズム）．**

3 **AVL 木の条件の考察を行う．**

a **条件を満たしていなければ，4 を実行する．**

b **条件を満たしている． -> 削除終了**

4 **再構築方法の選別を行う．**

a **一重回転 -> 3 を実行する．**

b **二重回転 -> 3 を実行する．**

上述の 1 と 2 の削除のアルゴリズムと場合分けのアルゴリズムは，2.3.2 節に準ずる．

AVL 木の条件の考察

場合分けのアルゴリズムで最後に削除した節点で，AVL 木の条件の考察を行う．削除した節点から根に向かって，AVL 木の条件である「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．

T の節点 y において，その左部分木 T_L と右部分木 T_R に関する状態を式 (2.25) で定める．この状態をデータ構造の第 3 メンバで保存する．

今，部分木において，左の子 v からその親 u へ遡ってきた場合を考える．すなわち，v を根とする部分木の高さが 1 減った場合である．v の高さが減らないときには，これ以上の考察は必要なくなる．u の第 3 メンバの値によって，次の 3 つの場合に分けられる．

- $s(u) =$ LEFT の場合: 左部分木の高さが 1 減ったため，$s(u) =$ EVEN に変わる．u を根とする部分木の高さが 1 減るので，u の親に遡ることになる．u が根ならば，考察はこれで終了とする．
- $s(u) =$ EVEN の場合: 左部分木の高さが 1 減ったため，$s(u) =$ RIGHT に変わる．u を根とする部分木の高さは変化しないので，考察はこれで終了となる．
- $s(u) =$ RIGHT の場合: u は AVL 木の条件を満たさなくなり，再構築方法の選別を呼び出す．

同様に，右の子からの場合も対称的に扱える．これを考察のアルゴリズム (削除) とする．

考察のアルゴリズム (削除) を以下に示す．節点 u は節点 v の親とする．

1　a　v **が** T **の根である．** -> **考察終了**

　　b　v **が** T **の根でないならば，**2 **を実行する．**

2　a　v **が左部分木である．** -> 3 **を実行する．**

　　b　v **が右部分木である．** -> 4 **を実行する．**

3　a　s(u) = EVEN -> s(u) = RIGHT **にして，考察終了**

　　b　s(u) = RIGHT -> **再構築方法の選別を呼び出す．再構築の後，この部分木の高さが** 1 **減っていたら，部分木の根を** v**，その親を** u **として，**1 **から実行する．**

　　c　s(u) = LEFT -> s(u) = EVEN **にして，**u **を** v**，**u **の親を** u **として，**1 **から実**

行する．

4 a `s(u) = EVEN -> s(u) = LEFT` **にして，考察終了**

b `s(u) = LEFT ->` **再構築方法の選別を呼び出す．再構築の後，この部分木の高さが** `1` **減っていたら，部分木の根を** `v`**，その親を** `u` **として，**`1` **から実行する．**

c `s(u) = RIGHT -> s(u) = EVEN` **にして，**`u` **を** `v`**，**`u` **の親を** `u` **として，**`1` **から実行する．**

再構築方法の選別

考察のアルゴリズム (削除) より，回転操作が必要な場合は「再構築方法の選別」を行なう．これを再構築のアルゴリズムとする．このアルゴリズムでは回転の種類を選別している．

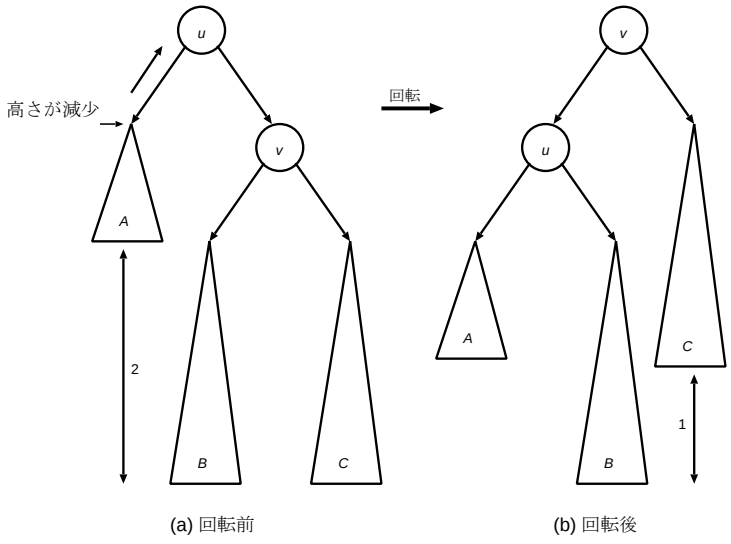


図 2.26 削除における一重回転 (高さが変わらない場合)

削除における回転の種類には，一重回転と二重回転がある．一重回転には 2 種類あり，それぞれを図 2.26 と図 2.27 に示す．二重回転を図 2.28 に示す．図 2.26，図 2.27 と図 2.28 の (a) 回転前では，部分木 A の高さが 1 減少して，u の左部分木が右部分木より高さが 2 低くなった場合である．この操作で，(b) の部分木の高さは (a) の部分木のそれより

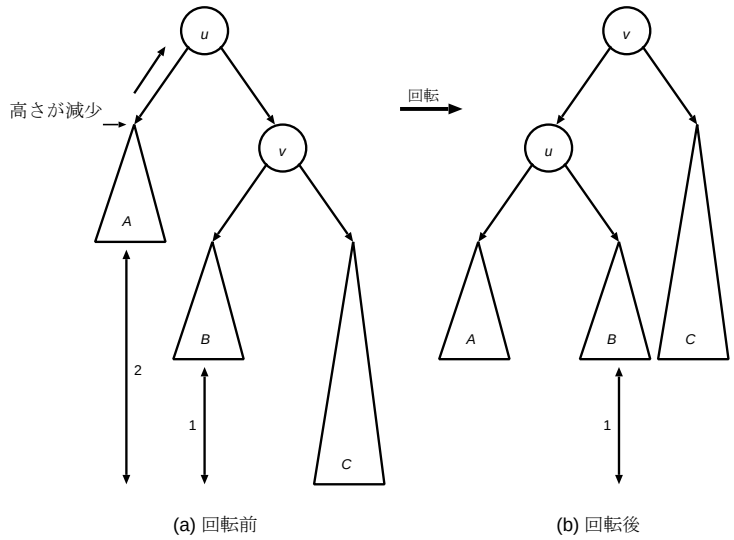


図 2.27　削除における一重回転 (高さが 1 減少する場合)

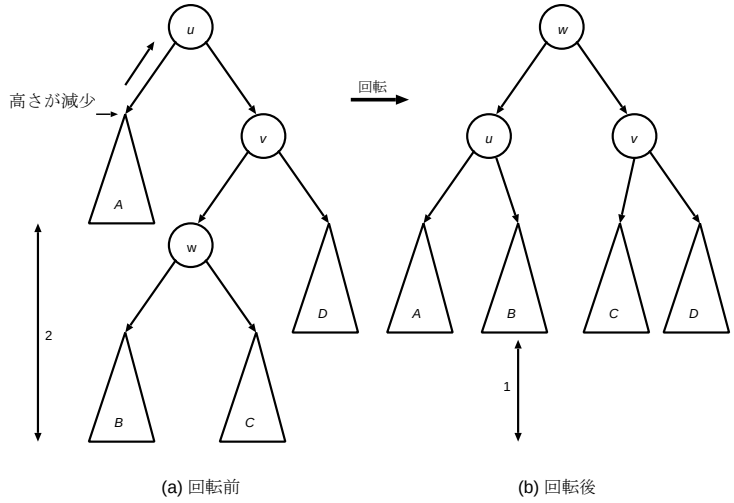


図 2.28　削除における二重回転 (高さが 1 減少する)

も同じか，又は 1 低くなる．(b) の部分木の高さが削除前のそれと変わらない場合は 1 回の再構築だけで良いが，高さが 1 低くなる場合は部分木の根の親にて考察のアルゴリズム (削除) を再実行しなければならない．すなわち，このアルゴリズムを再実行しなければならない場合は，回転後に部分木の根節点の左部分木と右部分木の高さが同じ場合である．

同様に，右部分木の方が左部分木よりも低い場合も対称的に扱える．これを再構築のアルゴリズムとする．

再構築のアルゴリズムを以下に示す．節点 u は節点 v の親とする．

1 a s(u) = LEFT -> 2 **を実行する．**

b s(u) = RIGHT -> 3 **を実行する．**

2 a s(v) = EVEN || s(v) = LEFT -> **一重回転**

b s(v) = RIGHT -> **二重回転**

3 a s(v) = EVEN || s(v) = RIGHT -> **一重回転**

b s(v) = LEFT -> **二重回転**

一重回転

一重回転では，図 2.26 と図 2.27 のように節点 u と節点 v のポインタを付けかえる．

2.4.3 節の一重回転と同じである．

二重回転

二重回転では，図 2.28 のように節点 u，節点 v 及び節点 w のポインタの付けかえる．

2.4.3 節の二重回転と同じである．

2.4.4 リストをある順番で印字

AVL 木は T の高さが低くなった 2 分探索木であるので，2.3.3 節で説明した 2 分探索木のなぞりを利用すれば，リストが昇順で印字される．

2.4.5 AVL 木の計算量

データ数を n とすると，AVL 木の高さによる計算量は，2.4.2 節より $O(\log n)$ である．対象データが見つかった後のそれぞれの手続きの計算量は，$O(1)$ であるから，AVL 木の計算量は，$O(\log n)$ である．

2.5　離散探索木

これまでの木構造は，キー自体の値を探索していたが，キーを構成している文字を先頭から桁単位で値の比較を行い，これを木構造で実現しているデータ構造を**離散探索木**（digital search tree）と呼ぶ．文字単位で探索することを**基数探索法**（radix-search method）と呼び，実生活で辞書を用い，単語を調べるのに，単語の先頭文字から順に探していく方法がこれにあたる．つまり，キーが**文字列**（string）の場合でも，1 文字の**語**（word）に分解して m 進の数値に変換する．この m を**基数**（radix）とよび，語単位で順序木を構成したものを離散探索木と呼ぶ．この離散探索木の基本的なアルゴリズムに**トライ**（Trie）と**パトリシア**（Patricia Tree）がある．トライやパトリシアは文字列処理，パターン照合処理に適したデータ構造である．

トライ，パトリシア

トライ [14] は，Fredkin により名付けられ，検索の re*trie*val から名づけた名前である．トライ構造は，キーの文字単位に構成された順序付き木構造で構築され，各キーの共通接頭辞を併合して作られる木構造であり，自然言語処理システム系 [38] など文字列を対象とした語彙の検索に向いている．トライの構造を図 2.29 に示す．図 2.29 では，節点に語を挿入しておらず，枝に語をラベル付けしている．ここでは，文字列の集合を $K = \{OLD, NAS, NEX, NEW, NEE, NEED, NO, NOW, BIG\}$ の 9 文字としている．節点の黒丸は，文字列の終点を表している．日本語の文字列でもトライ構造を図 2.29 と同様に構築できるが，各節点から出る枝の数を語の文字の種類だけ準備する必要がでてくる．節点から出る枝の数は，データ構造として前もって保持しなければならず，枝が使用されなければ領域量的には非効率となる．

トライの長所 [2, 5, 28] には，次がある．

1. 入力文字の左端より始まる共通の接頭辞が 1 回の走査で探索できる．
2. 探索失敗でも探索キーと部分マッチする文字列を検出できる．
3. 節点から出る枝の数に関係なく一定時間で探索できるならば，探索における**最大時間計算量**（worst-case time complexity）はキーの数に関係なく，キーの長さに比

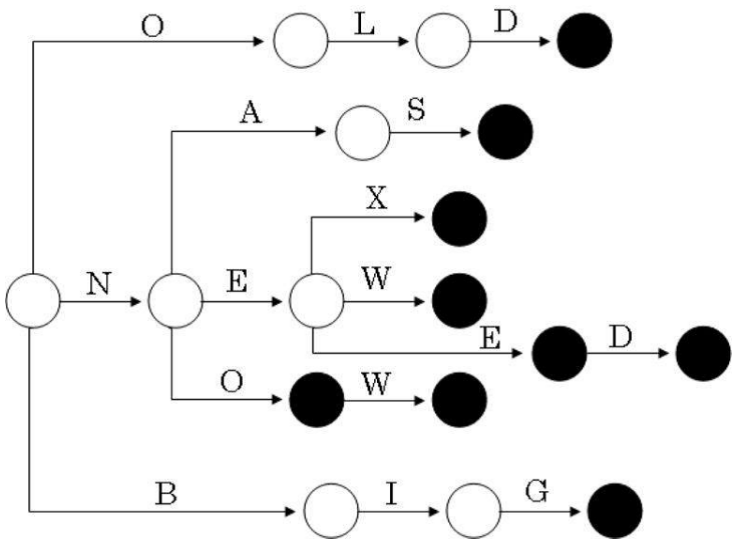


図 2.29　トライの木構造

例することである．

上述の 1. では，図 2.29 のトライで入力文字を「NOWHERE」とするならば，共通の接頭辞である「NO」と「NOW」も同時に探索でき，入力文字の左端より始まる共通の接頭辞が 1 回の走査で探索できる．2. では，図 2.29 のトライで「NOVA」を探索する場合，文字「V」で探索失敗となるが部分マッチする「NO」を検出できる．3. では，トライ構造は節点から出る枝の数が多くなると予想されるが，枝数に関係なく節点に遷移できるならば探索時間はキーの長さに比例することになり，高速な探索が可能となる．

トライの短所には，次がある．

1. 基数探索法の場合，文字の種類により節点から出る枝の数が多くなるが，木の下部にいくほど，探索文字が少なくなるので少数の枝のみで十分になるが，データ構造上は一定数の枝を持たせる必要がある．
2. 日本語のように文字の種類が多い場合は，各節点に多くの枝を持たせる必要があるが，その実装は現実的ではない．

トライの短所には，上述のようなメモリ効率の問題があり次の提案が考えられる．

1. 多バイト長の日本語などの場合は，1 バイト毎に節点を設けることで枝の数を少な

くする（$2^8 = 128$ 本）ことでデータ領域量の減少を図る．

2. 2 分木による再構成を行う．

上述の 1. では，1 バイト毎に節点を設けても，多くの節点は 128 本より少ない枝しか必要せず，無駄な領域が生成されると考えられる．2. では，各節点から出る枝の数を 2 本に限定したものである．つまり，各節点に文字の種類の枝を用意するのではなく，コンピューターが文字を認識する 2 進数まで文字を分解する．例えば，$S = \{O, L, N\}$ として，この集合 S を表 2.1 では，アルファベットを図 2.32 の内部表現値に従って 5 桁の 2 進数で示している．図 2.30 では，表 2.1 のアルファベットをトライの木構造で表現している．各

表 2.1　キー集合 S に対する 2 進数表現

アルファベット	内部表現値	2 進数表記
O	16	10000
L	13	01101
N	15	01111

節点に何ビット目を比較しているかの桁数，節点から左部分木への枝を ‘0’，右部分木への枝を ‘1’ とすれば，集合 S の各要素は図 2.30 のように表すことができる．このように，

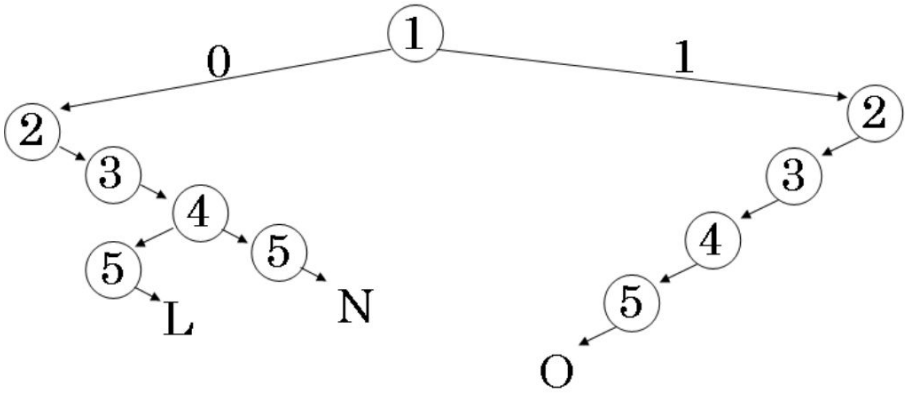


図 2.30　2 進数でのトライ

各節点から出る枝の数を 2 本に限定すれば枝の領域量は減らすことができるが，文字を表するための節点数が多くなる．図 2.30 で，根節点において右部分木に進行するのは要素「O」のみだが，すべての節点と枝を表示している．また，要素「L，N」では節点 2 と 3

のように文字のビット列が共通する場合に生じる ‘一方向分岐’ がある．これらの領域量を減少させる木構造がパトリシアである．

D.R.Morrison は，2 分木のトライの枝別れのないパスを縮退することで，領域量の減少と効率の良い探索を行う木構造を**パトリシア**（Patricia, ‘Practical Algorithm To Retrieve Information Code In Alphanumeric’）と名付けた．パトリシアを図 2.31 に示す．図 2.31 のパトリシアは，図 2.30 で枝が分岐するときに限り節点をつくったもので

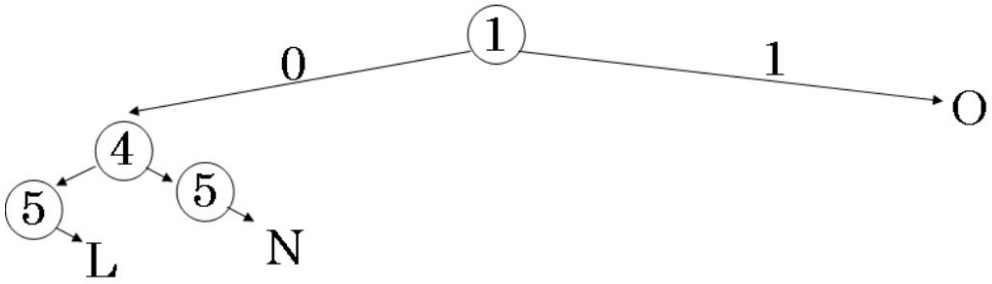


図 2.31　パトリシア

ある．この木構造で探索するときは，節点の桁数目で比較を行い，葉に辿りついたら探索キーと節点のキーを照合する．このように，一方向分岐から領域量を減らす木構造がパトリシアである．

この他に，トライを出発点として様々な提案がなされている．トライでの配列構造の圧縮法 [3, 30, 50] とリスト構造での圧縮方法 [29] もある．トライのデータ構造には，基本的に配列構造とリスト構造が考えられるが，効率的な探索にトライ構造にダブル配列 [6] を用いる手法が提案されている．ダブル配列は，2 つの配列を使用して高速性とコンパクト性を実現している．2 つの 1 次元配列を「BASE」と「CHECK」とおき，$g(r, a) = t$ なる遷移を

$$\begin{cases} t = \mathrm{BASE}[r] + a \\ \mathrm{CHECK}[t] = r \end{cases}$$

と定義する．節点 r から出るラベル $a\,(\geq 1)$ の枝とする場合，t の値は BASE$[r]$ に記号 a の内部表現値を加えた値である．CHECK$[t]$ は, この枝が節点 r から出てきたことを示す．例えば, 文字列の集合を $K = \{OLD, NAS, NEX, NEW, NEE, NEED, NO, NOW, BIG\}$ の 9 文字として，文字列の終端記号を「#」とする．アルファベットの内部表現値を

図 2.32 とする．図 2.32 では，終端記号「#」の内部表現値は 1 であり，「S」の内部表現値は 20 である．図 2.33 は，集合 K に対応するダブル配列である．ダブル配列の要素 0

#	A	B	C	D	E	F	G	H	I
1	2	3	4	5	6	7	8	9	10
J	K	L	M	N	O	P	Q	R	S
11	12	13	14	15	16	17	18	19	20
T	U	V	W	X	Y	Z			
21	22	23	24	25	26	27			

図 2.32　内部表現値

は，未使用要素である．図 2.34 は，ダブル配列の t の値を節点の要素としている．端記

r	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
BASE	1	−1	−1	10	1	1	10	−1	1	−1	−1	−1	−1	1	11	3	1	−1
CHECK	0	6	21	1	16	14	9	26	16	25	7	15	19	17	7	1	1	36

r	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
BASE	12	14	2	22	−1	0	9	7	0	0	0	0	0	0	0	0	0	17
CHECK	16	4	5	20	22	0	9	9	0	0	0	0	0	0	0	0	0	19

図 2.33　ダブル配列

号「#」を経て遷移した節点は次の遷移先を定義する必要がないので，その節点に対応する図 2.33 の BASE には「−1」を格納している．ダブル配列では，節点から出る枝数に関係なく探索する最大時間計算量は $O(1)$ である．ダブル配列によるデータ領域の圧縮（最小接頭辞トライ）もある．最小接頭辞トライとは，英単語約 3 万語に対してトライの節数は約 10 万個である [5] が，この節数を減少する方法である [4, 21]．ダブル配列の欠点は，枝の頻繁な削除があると未使用要素を容易に解放できない点にある．

2.6　B 木

2 分探索木と AVL 木は内・外部節点ともに 2 分木であったが，ここで説明する B 木の場合は 1 つの節点に高々 k 個のデータを挿入できる k 分木である．B 木は，ベイヤー（R.

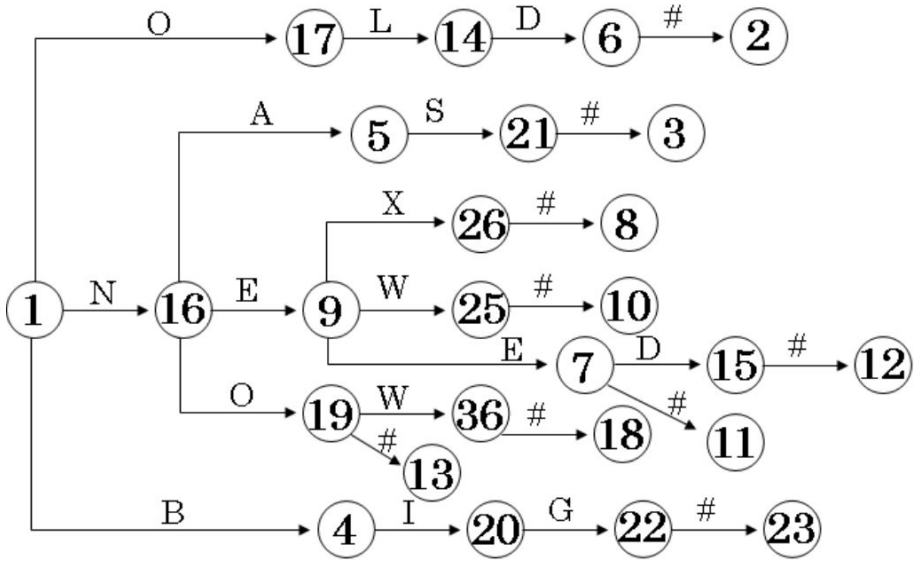


図 2.34　ダブル配列に対応するトライ

Bayer）とマクレイト（E. McCreight） [8] によって名付けられ，節点は最大 $k(\geq 2)$ 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．計算機の記憶領域は，データの連続するブロックであるページ単位に分割されており，各ページは多くのレコードを保持する．このことから，B 木は，もともと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．

k を奇数とし，B 木の定義は次のようにする．

定義 2.7. 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 木は次の構造的性質をもつ．根では，要素の数は 1 と k の間でなければならない．他の節点では $(k+1)/2$ と k の間でなければならない．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 2.7 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．節点の中にキーを k 個含めるならば，リンクの数は k 個である．挿入する場合には，木の根から探索を始め探索キーが含まれる区間のリンクを辿り次の節点に移動する．これを内部節点で

続けて外部節点に到達したら，適切な位置にレコードを挿入する．外部節点のデータの数が k より大きくなったら，外部節点の分割が必要になり，新しく生成された外部節点へのキーとリンクを親節点に挿入する．挿入された内部節点ではキーとリンクの数が 1 個増えるが，挿入の操作を下層から上層へと遡り，最終的に根で分割したときに高さが増加する．図 2.35 は，根における節点の分割と高さが高くなる図である．図では，5 次の B 木であり，○がキーもしくはデータであり，×はキーまたはデータが挿入されていない．図の（a）では 5 つのデータが挿入されており，（b）ではデータを追加したものである．図（b）の根の○はキーであり，葉の○はデータとなる．根の 2 つのキーで区間を示しており，該当する子節点へ進行し，葉でデータを探索する．図 2.36 では，高さ 2 の B 木である．

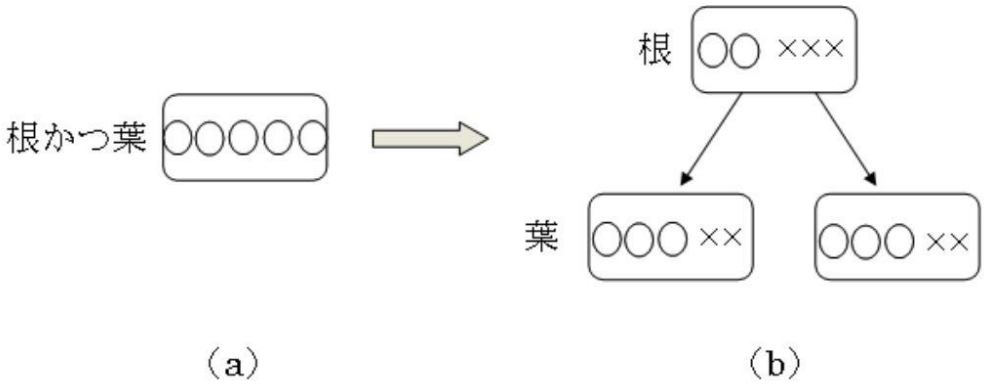


図 2.35　B 木の節点の分割について

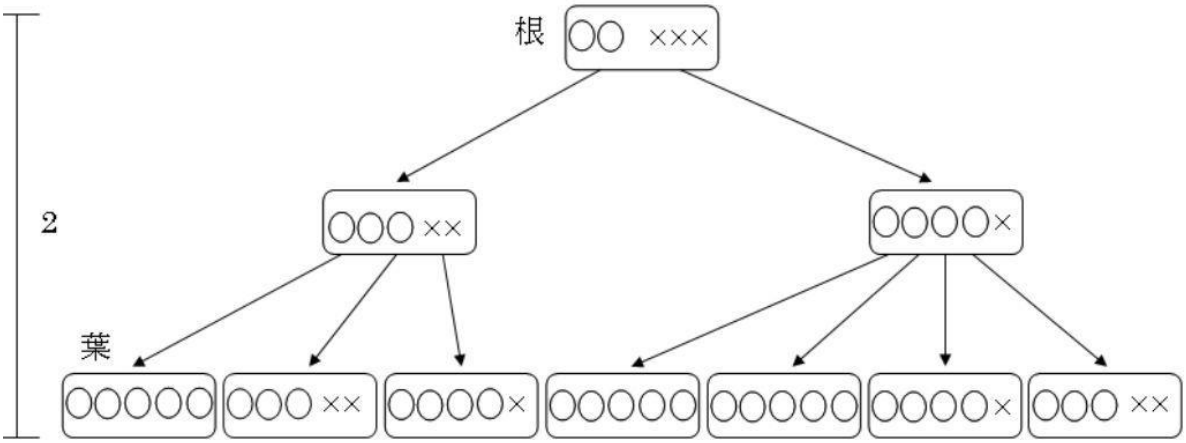


図 2.36　高さ 2 の B 木

図 2.36 では，節点の分割時に根のキー数は 2 つであり，その他の節点では $(k+1)/2$

であることが確かめられる．また，定義より根以外の節点では半分以上のキーが挿入されていることになる．B 木の構造の構成には次がある．

1. 内・外部節点ともにデータを挿入する．
2. 内部節点にデータを挿入せずに外部節点のみにデータを挿入する．

1. では，内・外部節点ともにデータ構造を同じにできるが，葉のポインタが空になり領域量が無駄になる．2. では，内部節点と外部節点のデータ構造を変える必要がある．内部節点にはキー値のみを挿入して，データはすべて葉に格納する．このようにすると，葉のポインタ位置にデータを挿入することになるが，内部節点は索引部になり必ず葉まで辿らなければならなくなる．この構造を活用した例が次の節の B+ 木である．

2.6.1　B+ 木

B+ 木 [11] ではデータを葉のみに格納することで，根から探索するランダム探索だけでなく葉における順探索も可能としたものである．図 2.37 は，B+ 木の構造を示したものである．葉における順探索も可能にしたことで，データベースのインデックスなどにも広く利用されている．その代わりに，ランダム探索をする場合は，データの有無を確かめるために葉まで辿る必要がある．削除する場合は，内部節点のキーを削除せず外部節点のレコードのみを削除する．内部節点のキーを削除しなくとも判定値として正しく作用するので，内部節点のキーは削除しない．このようにすることで効率性を維持し，削除によって外部節点の利用率が小さくなった場合に木の再構築を行うものである．B+ 木は，B 木のランダム探索の効率に加えて効率的な順探索を可能とした技法である．

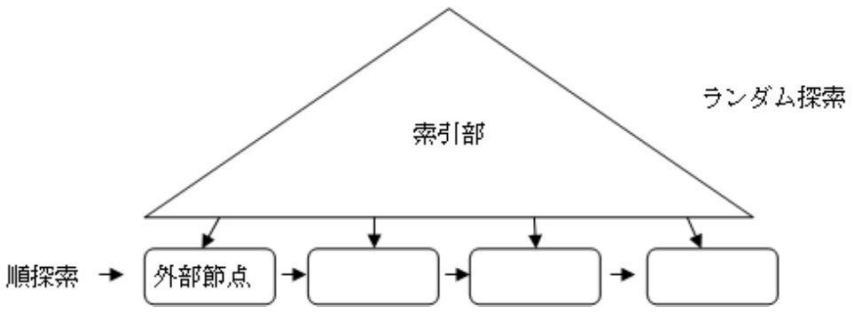


図 2.37　B+ 木の構造

第 3 章

5 分木に拡張した AVL 木の提案

3.1 はじめに

第 2 章までの準備を背景に，文字列が探索できる木構造を本章で提案する．提案する木構造のアルゴリズムは，文字列を効率的に探索できるように基数探索法を利用している．さらに，部分木単位で平衡化することで，探索の効率化を目指している．データ構造は，動的な辞書を作成できるように節点同士をリスト構造で結び，既存の AVL 木を 5 分木に拡張し，探索効率の向上を計っている．この章では，提案する文字列探索木のデータ構造と種々のアルゴリズムについての説明と計算量の考察を行う．

3.2 5 分木に拡張した AVL 木の提案

第 2 章で述べた文字列に対応させ，かつ挿入や削除などの動的な処理を必要するような 2 分探索木や AVL 木を考える．これらの木構造において，ある文字列を探索中にある節点に到達した場合，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 3.1 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ったら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，削除後の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．第 2 章の 2 分探索木や AVL 木のプログラム化は，基数探索法の考えをそのまま実装している．図 3.1 も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 3.1 の ‘$’ は，終端記号である．

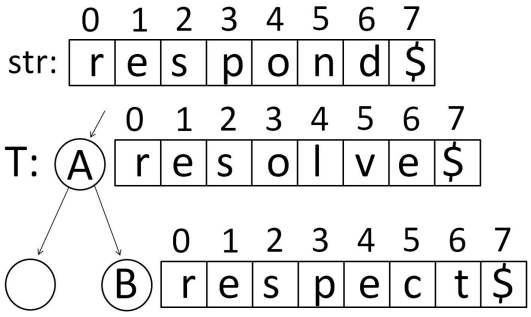


図 3.1　データを文字列にした AVL 木

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 3.2 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．

5 分木に拡張した AVL 木の定義と詳細なデータ構造は後述するが，まず，図 3.2 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

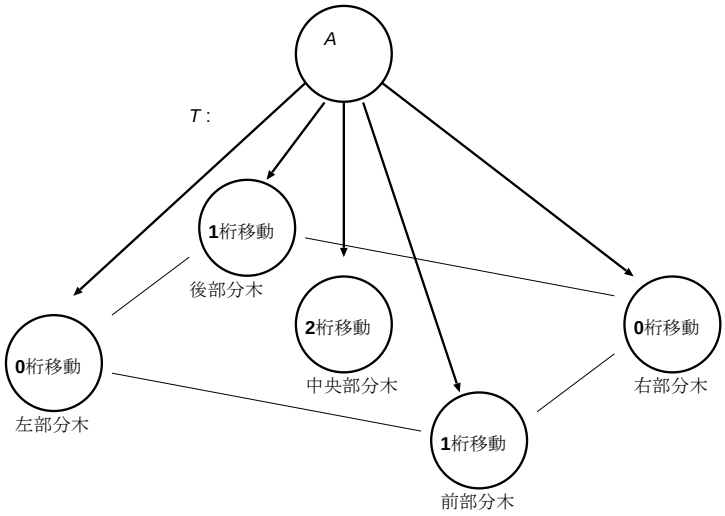


図 3.2　5 分木に拡張した AVL 木の節点の構造

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図 3.2 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．

これまで通り，あるデータを挿入するとき，まず，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

[*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．

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．

図 3.3 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 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 3.3 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文字列ということが判明される．なお，図 3.3 では，根節点からの比較する桁を ‘digit’ で示している．

拡張した AVL 木の定義は次のようになる．

定義 3.1. 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 $T_1 \sim T_5$ 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) 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

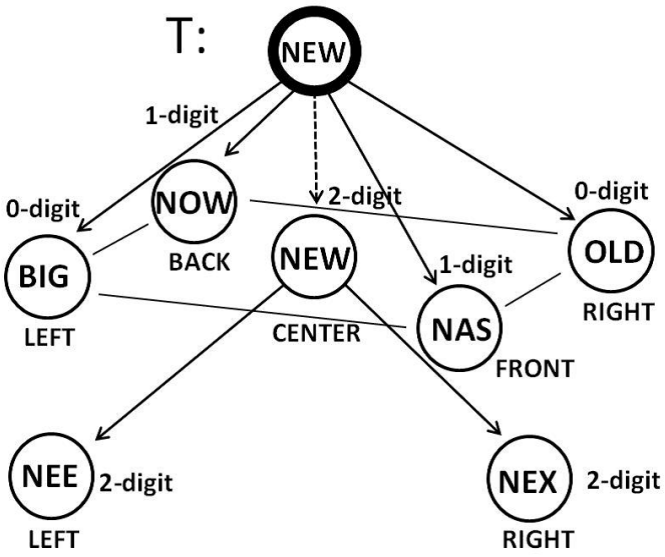


図 3.3　拡張した AVL 木におけるデータの挿入例

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．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．

3.2.1　5 分木に拡張した AVL 木のデータ構造

2 分探索木と AVL 木同様に，文字列探索に対応した 5 分木に拡張した AVL 木を構築する．拡張した AVL 木のアルゴリズムを作成するためにデータ構造を構築する．節点 1 つ当たりのデータ構造を以下のように定義する．

ソースコード 3.1　拡張した AVL 木のデータ構造（C 言語）

```
1  enum flag {LA, NU};                          // 節点の種類
2  enum sub {RO, LT, RT, FT, BT, CT};           // 部分木の種類
3  enum diff {EVEN, LEFT, RIGHT};               // バランス
4
5  // 拡張したAVL木のデータ構造
6  struct ext_avl {
7      char element[S+1];                       // 文字列
8      enum flag flag;
9      enum sub sub_tree;
10     enum diff diff;
11     struct ext_avl *left;
12     struct ext_avl *right;
13     struct ext_avl *front;
14     struct ext_avl *back;
15     struct ext_avl *center;
16     struct ext_avl *parent;
17 };
```

節点 1 つ当たりのデータ構造は 6 行目からとなるが，8, 9, 10 行目の enum flag 型，enum sub 型と enum diff 型を先に説明する．1 行目の enum flag 型は，ラベルまたはデータを表す．変数に ‘LA’ が格納されたときは，その節点はラベルということを意味して，‘NU’ では，その節点はデータであることを表す．この集合を enum flag 型として定義している．2 行目の enum sub 型は，節点が親の節点に対して，どの部分木であるかを表す．節点が木の根であれば ‘RO’ であり，節点が親の節点に対して左部分木であれば ‘LT’ であり，節点が親の節点に対して右部分木であれば ‘RT’ を保持する．ここまでは，AVL 木のデータ構造と同じであるが，これに加えて，節点が親の節点に対して前部分木であれば ‘FT’ であり，節点が親の節点に対して後部分木であれば ‘BT’ であり，節点が親の節点に対して中央部分木であれば ‘CT’ を保持する．この集合を enum sub 型として定義している．3 行目の enum diff 型は，ある節点から見たときの高さの高い部分木の方を保持する．変数に ‘EVEN’ が格納されたときは，左右部分木の高さが同じであり，‘LEFT’ で左部分木が右部分木よりも高く，‘RIGHT’ では右部分木が左部分木よりも高いことを表す．この集合を enum diff 型として定義している．拡張した AVL 木のメンバを 7〜16 行目に

示す．7 行目でデータまたはラベルを文字列で保持する．9, 10, 11, 12, 16 行目は AVL 木のデータ構造と同じとなるが，8, 13, 14, 15 行目が新たに追加される．8 行目は先に述べた通りである．13, 14, 15 行目は，それぞれ前部分木を指すポインタ，後部分木を指すポインタと中央部分木を指すポインタである．1 節点当たり必要となるメモリ量は，char 型は 1 バイトより文字数 S と '\0' を加えて，$S+1$ バイト，ポインタは 4 バイトより 24 バイト，列挙型が 4 バイトより 12 バイト，合計で 1 節点当たり $S+37$ バイトとなる．

3.3 拡張した AVL 木の時間計算量

拡張した AVL 木に対する時間計算量について述べる．一般に，木構造において最も基本的な操作はデータ探索であり，これについては次の定理 3.1 が成立する．

定理 3.1. データ数を n，各データを m 進 S 桁とする．このとき，拡張した AVL 木において，データ探索の時間計算量は $O(\log n)$ である．

証明. データ数を n，各データを m 進 S 桁とすると $n \leq m^S$ である．任意の節点において，ラベル数を含めた拡張した AVL 木の高さを求める．あるデータを探索する場合，拡張した AVL 木では桁数 S に依存して，探索するデータを各節点データと比較する．そのため，データ探索の時間計算量を $f(S)$ とおく．

$$f(S) = (\text{(n 節点 + ラベル数) をもつ拡張した AVL 木の高さ})$$

データ探索の時間計算量は比較回数を考慮すれば良いので，最大の比較回数が問題となる．最大の比較回数とは 1 桁ずつ一致することであり，このとき時間計算量は木の高さと一致する．

$S=1$ の場合は，節点数は $n=m$ 個であり，木の高さは完全 2 分木と AVL 木の中間となる．この木の高さを a_1 とする．このとき，左部分木と右部分木の高さの差が高々 1 である．$S=2$ の場合は，前・後部分木が作成されるが中央部分木は作成されない．2 桁目だけの木の高さを a_2 とする．1 桁目と 2 桁目を繋ぐ経路を加えるために 1 を足す必要がある．これを図 3.4 に示す．$S \geq 3$ の場合で中央部分木が生成されるが，中央部分木による高さは無視できる．なぜならば，中央部分木に進行するときには，むしろ部分木の高

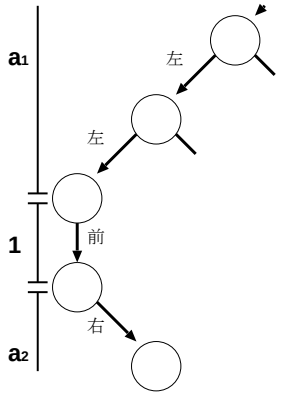


図 3.4　拡張した AVL 木の高さ ($S = 2$ の場合)

さは低くなるからである．これより，木の高さはラベル数の影響を受けないことになる．$S = 3$ で 2 桁目と 3 桁目を繋ぐ経路を加えるために 1 を足す必要がある．すなわち,

$$\begin{aligned} f(1) &= a_1 \\ f(2) &= a_1 + a_2 + 1 \\ f(3) &= a_1 + a_2 + a_3 + 2 \end{aligned}$$

であり，S 桁で,

$$f(S) = \sum_{i=1}^{S} a_i + (S - 1) \tag{3.1}$$

となる．$f(S)$ は最大の比較回数となる．$a_1, a_2, \cdots, a_S$ とも深さは変わらないので，1 桁の木の深さを a とし，$n = m^S$ のとき S を m, n で表すと式 (3.2) となる（対数の底は 2)．

$$\begin{aligned} f(S) &= S(a + 1) - 1 \\ &= \frac{a + 1}{\log m} \log n - 1 \end{aligned} \tag{3.2}$$

以上より，m 進 S 桁のデータ数 n における拡張した AVL 木のデータ探索の時間計算量は最悪の探索回数でも,

$$f(S) = O(\log n)$$

である．　□

次に式 (3.2) の 1 桁の木の深さ a を求めるために，AVL 木を満たす最小の節点数の公式（例えば，[19]）を利用する．AVL 木を満たす深さ h における最小のデータ数 $g(h)$ は，

$$\begin{cases} g(h) = g(h-1) + g(h-2) + 1 \quad (h \geq 2) \\ g(0) = 1,\ g(1) = 2 \end{cases} \tag{3.3}$$

であり，式 (3.3) より表 3.1 が得られる．

表 3.1　拡張した AVL 木の高さ h におけるデータの数

$\boldsymbol{h}$	0	1	2	3	4	5
$\boldsymbol{g(h)}$	1	2	4	7	12	20
$\boldsymbol{data}$	1	2-3	4-6	7-11	12-19	20-32
$\boldsymbol{h}$	6	7	8	9	10	11
$\boldsymbol{g(h)}$	33	54	88	143	232	376
$\boldsymbol{data}$	33-53	54-87	88-142	143-231	232-375	376-608

表 3.1 の第 1 行の h は木の高さ，第 2 行の $g(h)$ は最小のデータ数である．これより，第 3 行の $data$ は高さ h におけるデータ数の幅を示している．h を 1 桁の木の高さ a で置き換えることで，1 桁におけるデータ数（進数）を確定できる．例えば，計算機の内部コードは 8 ビット（$2^8 = 256$）であるので，拡張した AVL 木であれば，最悪でも高さ $a = 10$ で内部コードを表現可能となる．このとき，拡張した AVL 木では 10 桁の文字列の最大比較回数（時間計算量）$f(10)$ は，式 (3.2) より，

$$f(10) = \frac{11}{8} \log 256^{10} - 1 = 109$$

となる．AVL 木のデータ探索の時間計算量は完全 2 分木のそれの高々 1.44 倍（性質 2.1）であったが，拡張した AVL 木の計算量は完全 2 分木（式（4.1） $2^{h+1} - 1 = (2^8)^{10}$ より $h = 79$）のそれの高々約 1.38 倍となる．また，Multidimensional Height-Balanced Trees [52] における 256 進数で 10 文字の文字列では，木の高さ h は高々 188 程度である．

3.4　拡張した AVL 木の操作

各操作の前に用語を明記する．

- 節点に格納されているデータを「節点データ」で，探索するデータを「探索データ」と呼ぶ．
- 外部節点を「データ」で，内部節点を「グループ」と呼ぶ．
- 1 節点で最大 2 桁まで走査するが，1 桁目を 0 桁で，2 桁目を 1 桁で示す．

各操作は，2.3.2 節と同じである．

3.4.1　「データ」及び「グループ」を探索

拡張した AVL 木の探索では，T の根節点から始めて部分木にグループが存在することを利用して，「データ」及び「グループ」を探索する．これを探索のアルゴリズムと呼ぶことにする．比較する桁が 0 桁で一致すれば，節点と前・後・中央部分木が対象になり，0 桁と 1 桁が一致すれば節点と中央部分木が対象となる．

以下に探索のアルゴリズムを示す．

1　**T に探索する部分木があるかを調べる．**

　a　**T に探索する部分木があるならば，2 を実行する．**

　b　**T に探索する部分木がないならば，7 を実行する．**

2　**0 桁を比較する．**

　a　**探索データの 0 桁の値 < 節点データの 0 桁の値**

　　-> 左部分木へ進行し，この桁を 0 桁として，1 から実行する．

　b　**探索データの 0 桁の値 > 節点データの 0 桁の値**

　　-> 右部分木へ進行し，この桁を 0 桁として，1 から実行する．

　c　**探索データの 0 桁の値 = 節点データの 0 桁の値**

　　-> 3 を実行する．

3 **探索データの終端であるかを調べる．**

a **探索データの 0 桁が文字列の終端ならば，節点と前・後・中央部分木が対象である． -> 探索終了**

b **探索データの 0 桁が文字列の終端でないならば，4 を実行する．**

4 **1 桁を比較する．**

a **探索データの 1 桁の値 < 節点データの 1 桁の値**

-> 前部分木へ進行し，この桁を 0 桁として，1 から実行する．

b **探索データの 1 桁の値 > 節点データの 1 桁の値**

-> 後部分木へ進行し，この桁を 0 桁として，1 から実行する．

c **探索データの 0 桁の値 = 節点データの 0 桁の値**

-> 5 を実行する．

5 **探索データの終端であるかを調べる．**

a **探索データの 1 桁が文字列の終端ならば，節点と中央部分木が対象である． -> 探索終了**

b **探索データの 1 桁が文字列の終端でないならば，6 を実行する．**

6 **中央部分木に進行する．**

a **節点に中央部分木が存在するならば，中央部分木に進行する．**

1 **次の桁を 0 桁として，1 から実行する．**

7 **T に探索の文字列は存在しない． -> 探索終了**

3.4.2 挿入

探索後に T に該当の文字列がなければ，新しい葉を作成する．その後に拡張した AVL 木の条件を満たしているかを考察する．定義 3.1 の中で「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，平衡を行ってしまうと構造が崩れてしまうからである．この状態を図 3.5 に示す．図では，節点 A の前部分木と後部分木の

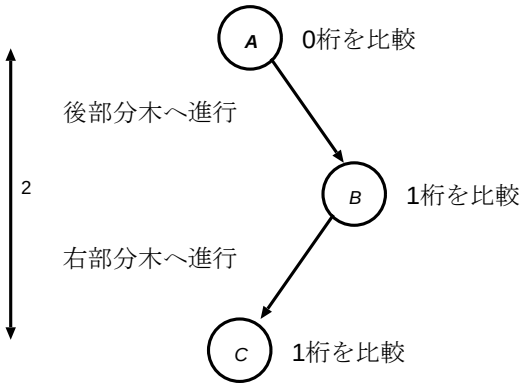


図 3.5　前，中央，後部分木の深さを含めない理由

高さの差が 2 となり，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．

T の高さを少しでも低くするために，挿入の時点で前部分木と後部分木との平衡を行うような工夫をしている．ある節点の前または後部分木に進行する時点で，節点データと探索データの桁値が進数の中央値に近い方を新しい節点に置き換える．このようにすることで，節点には中央値に近いデータが格納されて，前部分木と後部分木の平衡化が期待できる．しかし，節点に中央部分木が存在した場合には，比較する桁が移動している節点があるので，置き換えられない．よって，節点に中央部分木が存在しない場合のみに，この手続きを行う．中央値は，m 進数の m が偶数であれば，$a, b \in \mathbb{R}$ で，

$$a = \left\lfloor \frac{m-1}{2} \right\rfloor$$
$$b = \left\lfloor \frac{m}{2} \right\rfloor$$

で，2 個の中央値が得られ，m が奇数であれば，

$$a = b = \left\lfloor \frac{m-1}{2} \right\rfloor$$

で，2 個とも同じ値の中央値が得られる ($\lfloor \rfloor$ はガウス記号の $floor$)．挿入のアルゴリズム

にこのアイディアを取り入れている.

挿入のアルゴリズムを以下に示す.

1 T に探索する部分木があるかを調べる.

a T に探索する部分木があるならば，2 を実行する.

b T に探索する部分木がないならば，17 を実行する.

2 0 桁を比較する.

a 探索データの 0 桁の値 < 節点データの 0 桁の値

-> 左部分木へ進行し，この桁を 0 桁として，1 から実行する.

b 探索データの 0 桁の値 > 節点データの 0 桁の値

-> 右部分木へ進行し，この桁を 0 桁として，1 から実行する.

c 探索データの 0 桁の値 = 節点データの 0 桁の値

-> 3 を実行する.

3 1 桁を調べる.

a 探索データの 1 桁の値が，文字列の終端である.

-> 4 を実行する.

b 探索データの 1 桁の値が，文字列の終端でない.

-> 5 を実行する.

4 節点がデータかラベルかを調べる.

a 節点データがラベルである. -> データにして挿入終了

b 節点データがデータならば，同じ文字列が T に存在している.

->葉を作成しないで挿入終了

5 1 桁が同じ値かを調べる.

a 探索データの 1 桁の値 = 節点データの 1 桁の値

-> 6 を実行する.

b　探索データの 1 桁の値 != 節点データの 1 桁の値

-> 9 を実行する．

6　1 桁の次の桁の値を調べる．

a　探索データの 1 桁の次の値が，文字列の終端である．

-> 7 を実行する．

b　探索データの 1 桁の次の値が，文字列の終端でない．

-> 8 を実行する．

7　節点がデータかラベルかを調べる．

a　節点データがラベルである． -> データにして挿入終了

b　節点データがデータならば，同じ文字列が T に存在している．

->葉を作成しないで挿入終了

8　中央部分木に進行する．

a　節点に中央部分木が存在するならば，中央部分木に進行する．

-> この桁を 0 桁として，1 から実行する．

b　節点に中央部分木が存在しないならば，葉を作成して，この節点データを葉に複製する．葉に移動する． -> この桁を 0 桁として，1 から実行する．

9　前または後部分木に進行することになる．

節点データの 1 桁の値を調べる．

a　節点データの 1 桁の値が，中央値である又は中央部分木が存在する．

-> 10 を実行する．

b　節点データの 1 桁の値が中央値でない，かつ中央部分木が存在しない．

-> 11 を実行する．

10　1 桁を比較する．

a 探索データの 1 桁の値 < 節点データの 1 桁の値

-> 前部分木へ進行し，この桁を 0 桁として，1 から実行する．

b 探索データの 1 桁の値 > 節点データの 1 桁の値

-> 後部分木へ進行し，この桁を 0 桁として，1 から実行する．

11 探索データの 1 桁の値を調べる．

a 探索データの 1 桁の値が，中央値である．

-> 12 を実行する．

b 探索データの 1 桁の値が，中央値ではない．

-> 13 を実行する．

12 探索データと節点データを交換する．

a 探索データの 1 桁の値 < 節点データの 1 桁の値

-> 前部分木へ進行し，この桁を 0 桁として，1 から実行する．

b 探索データの 1 桁の値 > 節点データの 1 桁の値

-> 後部分木へ進行し，この桁を 0 桁として，1 から実行する．

13 探索データも節点データの 1 桁とも中央値ではない．

a |中央値 - 探索データ| >= |中央値 - 節点データ|

-> 15 を実行する．

b |中央値 - 探索データ| < |中央値 - 節点データ|

-> 16 を実行する．

15 節点データを平衡値とする．

a 探索データの 1 桁の値 < 節点データの 1 桁の値

-> 前部分木へ進行し，この桁を 0 桁として，1 から実行する．

b 探索データの 1 桁の値 > 節点データの 1 桁の値

-> 後部分木へ進行し，この桁を 0 桁として，1 から実行する．

16　**探索データを平衡値とするために，探索データと節点データを交換する．**

　a　**探索データの** 1 **桁の値 < 節点データの** 1 **桁の値**

　　-> **前部分木へ進行し，この桁を** 0 **桁として，**1 **から実行する．**

　b　**探索データの** 1 **桁の値 > 節点データの** 1 **桁の値**

　　-> **後部分木へ進行し，この桁を** 0 **桁として，**1 **から実行する．**

17　**葉を作成する．**

3.4.3　回転

挿入のアルゴリズムで葉を作成した場合は，回転の手続きを行う．回転の手続きは，2.4.3 節に準ずる．すなわち，回転の手続きとは，以下の手順である．

1　**葉を作成したら，拡張した** AVL **木の条件の考察を行う．**

　a　**条件を満たしていなければ，**2 **を実行する．**

　b　**条件を満たしている．　-> 回転終了**

2　**再構築方法の選別を行う．**

　a　**一重回転　-> 回転終了**

　b　**二重回転　-> 回転終了**

拡張した AVL **木の条件の考察**

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，「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．ただし，部分木が，前・後・中央部分木である場合は，拡張した AVL 木の条件の考察を止める．

拡張した AVL 木の条件の考察は，2.4.3 節の AVL 木の条件の考察と基本的な考えは同じであるので，アルゴリズムだけを載せる．節点 u は節点 v の親とする．

1 **節点** v **を調べる．**

a v **が前・後・中央部分木である．** -> **考察終了**

b v **が左または右部分木である．** -> 2 **を実行する．**

2 **節点** v **を調べる．**

a v **が** T **の根である．** -> **考察終了**

b v **が** T **の根でないならば，**3 **を実行する．**

3 **節点** v **がどの部分木かを調べる．**

a v **が左部分木である．** -> 4 **を実行する．**

b v **が右部分木である．** -> 5 **を実行する．**

4 **節点** u **の状態を調べる．**

a s(u) = RIGHT -> s(u) = EVEN **にして，考察終了**

b s(u) = LEFT -> **再構築方法の選別を行なう．**

c s(u) = EVEN -> s(u) = LEFT **として，**u **を** v**，**u **の親を** u **として，**1 **から実行する．**

5 **節点** u **の状態を調べる．**

a s(u) = LEFT -> s(u) = EVEN **にして，考察終了**

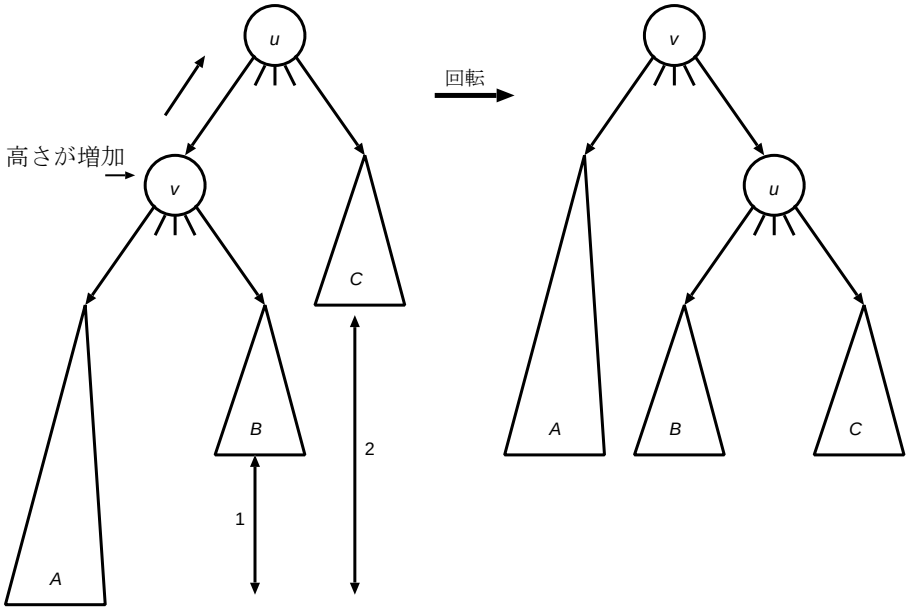
b s(u) = RIGHT -> **再構築方法の選別を行なう．**

c s(u) = EVEN -> s(u) = RIGHT **として，**u **を** v**，**u **の親を** u **として，**1 **から実行する．**

再構築方法の選別

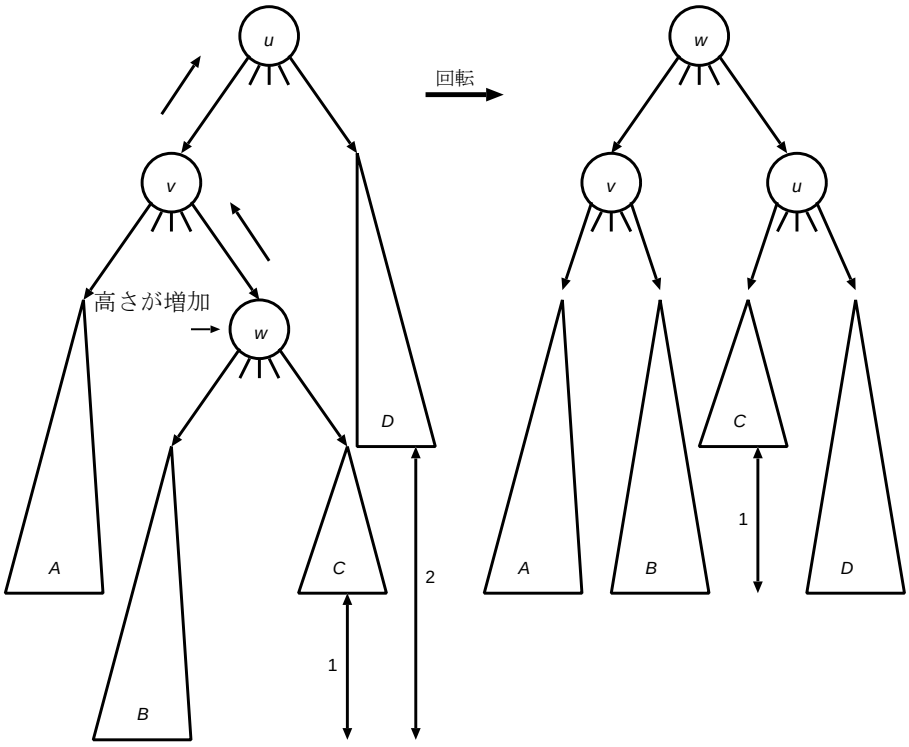
上述のアルゴリズムより，回転操作が必要な場合は「再構築方法の選別」を行なう．これを再構築のアルゴリズムとする．このアルゴリズムでは回転の種類を選別している．

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 3.6 と図 3.7 に示す．図 3.6 の (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．



(a) 回転前　　(b) 回転後

図 3.6　拡張した AVL 木の一重回転



(a) 回転前　　(b) 回転後

図 3.7　拡張した AVL 木の二重回転

図 3.7 の (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．

再構築のアルゴリズムは，2.4.3 節に準ずる．

一重回転

一重回転では，図 3.6 のように節点 u と節点 v のポインタの付けかえる．

一重回転のアルゴリズムは，2.4.3 節に準ずる．

二重回転

二重回転では，図 3.7 のように節点 u，節点 v 及び節点 w のポインタの付けかえる．

二重回転のアルゴリズムは，2.4.3 節に準ずる．

3.4.4 削除

図 3.8 の木 T では，節点の中に比較する添字番号を記入している．データの先頭添字を 0 から始めており，T の根節点 A では添字 0 から比較している．T において，A を削除する場合に，C かつ D が共に存在しない場合は通常の AVL 木の削除を適用する．それ以外の場合は，前部分木の 0 桁移動のうちで最も右にある右部分木の前部分木 J（または，後部分木の 0 桁移動のうちで最も左にある左部分木の後部分木）を A の位置へ移動する．図 3.8 では，A の位置へ J（ラベル）と同様な J'（ラベル）を用意する．J は A の比較の桁から 2 桁移動していたので，J を J' の中央部分木に移動することで木 T の構造が保持される（図 3.9 を参照）．後は部分木 J に F, G, H, I を移動する．F を O の右部分木に移動し，G を F の位置へ移動する．しかし，前部分木の 0 桁移動のうちで最も右にある右部分木の前部分木の節点が存在しない場合では，上述の削除が適用できない．例えば，図 3.8 の J が存在しない場合などは，次の方法で対処する．J が存在しないということは，葉に近い節点を削除することであり，部分木 C にはあまりデータが挿入されていないとみなして，部分木 C のデータを削除した A から再挿入して再構築することで対処可能となる．

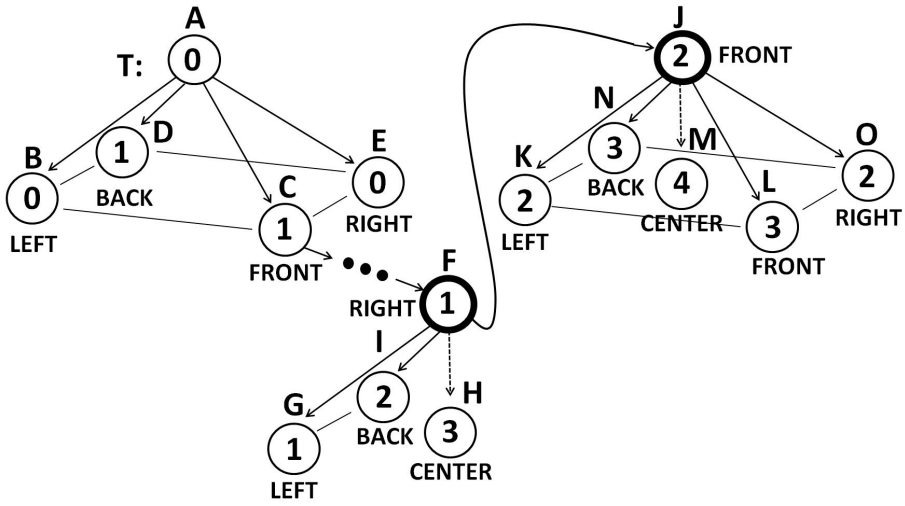


図 3.8　拡張した AVL 木のデータ削除（前）

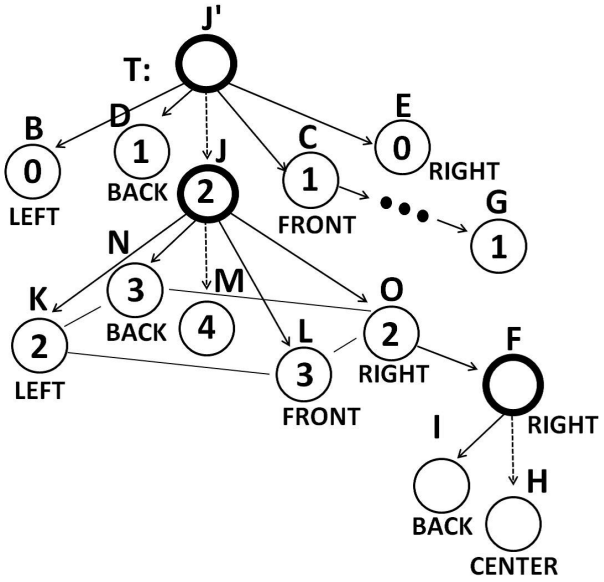


図 3.9　拡張した AVL 木のデータ削除（後）

3.4.5　リストをある順番で印字

2 分探索木と AVL 木では，節点 A に左部分木と右部分木が存在して，

$$\text{左部分木} < A < \text{右部分木}$$

の関係が成立した．これを再帰的に実行することで，ソートが可能となる．

今回の 5 分木では，節点 A に左部分木，右部分木，前部分木，後部分木，中央部分木が存在して，

$$\text{左部分木} < \text{前部分木} < A < \text{後部分木} < \text{右部分木} \tag{3.4}$$

の関係があることから，これを再帰的に実行することで，同様にソートが可能となる．もし，A に中央部分木が存在すれば中央部分木に移動して，これを A として，式 (3.4) を実行する．A に中央部分木が存在するということは，A はラベルということであるが，ラベルを印字しないことで走査を利用したソートが可能となる．

以下になぞりのアルゴリズムを示す．

T の根を u とする．

1 a **節点 u が存在する．** -> 2 **を実行する．**

b **節点 u が存在しない．** -> 8 **を実行する．**

2 u **の左部分木を** u **として，スタックに，戻りアドレスを**‘PUSH’ **する．**

-> 1 **から実行する．**

3 u **の前部分木を** u **として，スタックに，戻りアドレスを**‘PUSH’ **する．**

-> 1 **から実行する．**

4 u **がデータである．** -> u **の節点データを印字する．**

5 u **の中央部分木を** u **として，スタックに，戻りアドレスを**‘PUSH’ **する．**

-> 1 **から実行する．**

6 u **の後部分木を** u **として，スタックに，戻りアドレスを**‘PUSH’ **する．**

-> 1 **から実行する．**

7 u **の右部分木を** u **として，スタックに，戻りアドレスを**‘PUSH’ **する．**

-> 1 **から実行する．**

8　**スタックから，戻りアドレスを‘POP’する．**
もし，スタックが空である． -> なぞり終了

3.4.6　ラベルの総数

提案するデータ構造を実現するのに必要なラベルの総数について考察する．ラベルが増加するとデータを挿入や探索する場合に比較回数や領域量の増大を招くことになる．各データは m 進 S 桁，データ数を n，$n \leq m^S$ であるとする．このとき，T のラベルの総数は桁数 S だけに依存するため，これを $q\,(S)$ と記すことにする．文字列が 1,2 桁目では，中央部分木は生成されないので，

$$q(1) = 0, \quad q(2) = 0$$

となる．文字列が 3 桁以上で，中央部分木が生成されるようになる．

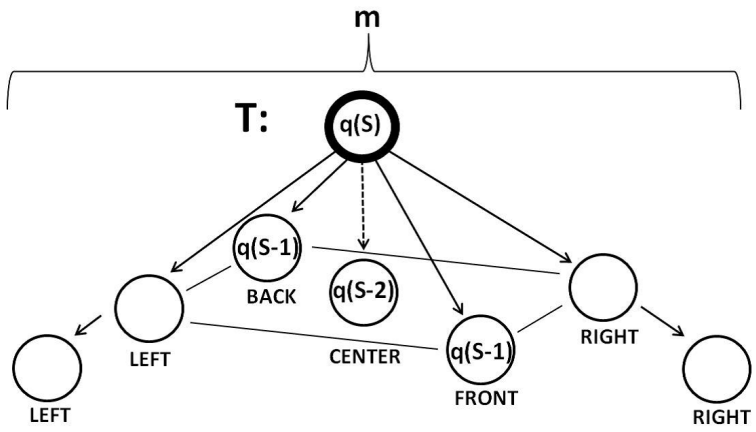


図 3.10　木 T に含まれるラベルの総数

図 3.10 において，根の前・後・中央部分木だけのラベルの総数を考察すると，T の根の前部分木および後部分木のラベルの個数の最大値は，

$$\frac{m-1}{m} \times q(S-1) \tag{3.5}$$

で求められる．式（3.5）の分子の $m-1$ は，図 3.10 において，根節点分だけ少ないから

である．根の中央部分木のラベルの総数は，$q\,(S-2)$ で表わせる．根はラベルなので，1 を加える．図 3.10 に示すように，根には左部分木および右部分木が存在するので，m 倍すると，

$$q(S) = m\left(\frac{m-1}{m}\cdot q(S-1) + q(S-2) + 1\right) \tag{3.6}$$

となる．式 (3.6) が，文字列 3 桁以上でのデータ数 $n\left(\leq m^S\right)$ に対する必要なラベル数となる．まとめると，次のようになる．

$$\begin{cases} g(1) = 0,\ g(2) = 0 \\ q(S) = m\left(\frac{m-1}{m}\cdot q(S-1) + q(S-2) + 1\right) \quad (S \geq 3) \end{cases} \tag{3.7}$$

式 (3.7) が，必要なラベル数となる．

次に，データ数 n に対するラベル数の割合を求めてみる．式 (3.7) において，$n = m^S$ とすると，

$$\begin{aligned} q(S+1) &= (m-1)\cdot q(S) + m\cdot q(S-1) + m \\ &= (m-1)\cdot q(S) + q(S) + q(S-1) - m\cdot q(S-2) - m + m \\ &= m\cdot q(S) + q(S-1) - m\cdot q(S-2) \quad (S \geq 3) \end{aligned}$$

より，データ数 $n\left(\leq m^S\right)$ に対するラベル数の割合は

$$\frac{q(S+1)}{m^{S+1}} = \frac{m\cdot q(S)}{m^{S+1}} + \frac{q(S-1) - m\cdot q(S-2)}{m^{S+1}} \quad (S \geq 3) \tag{3.8}$$

となる．式 (3.8) の右辺第二項は 0 に近い値と考えられる．これより，右辺第一項の割合のみに依存すると考えられるから，

$$\frac{q(S+1)}{m^{S+1}} \approx \frac{q(S)}{m^S} \quad (S \geq 3) \tag{3.9}$$

である．式 (3.9) は，データ数に対するラベルの割合が，ほぼ一定であることを示している．実際に式 (3.8) で求めたラベルの割合を表 3.2 に示す．第 1 列は桁数であり，第 2〜4 列はそれぞれ「5,10,15 進数」の場合である．表 3.2 からも，m を固定すれば，S の増減に関わらずラベルの割合は一定であることが分かる．さらに，m が大きくなるほどラベルの割合は減少する．これより，データが 10 進数の数値であってもラベルの数は，デー

表 3.2　木 T に含まれるラベルの割合

S 桁	5 進数	10 進数	15 進数
10	4.167%	1.010%	0.446%
20	4.167%	1.010%	0.446%
30	4.167%	1.010%	0.446%
40	4.167%	1.010%	0.446%
50	4.167%	1.010%	0.446%

タの節点に対して約 1% と大きな数値にはならないと判断できる．さらに，ひらがなの五十音では，データに対するラベル数の割合はより小さくなるので，ラベル数の増加で探索の効率が悪くなるとは考えにくい．

3.4.7　拡張した AVL 木の計算量

定理 3.3 より，データ数を n とすると 5 分木に拡張した AVL 木の高さによる計算量は，$O(\log n)$ である．対象データが見つかった後のそれぞれの手続きの計算量は，$O(1)$ であるから，5 分木に拡張した AVL 木の高さによる計算量は，$O(\log n)$ である．

3.5　まとめ

この章では，第 2 章までの様々なアルゴリズムとデータ構造を背景に，文字列探索に適した木構造を提案した．提案した木構造は，既存の AVL 木を拡張して文字列探索を可能としたものである．提案した木構造の特徴は，平衡と多分木による効率的な探索を可能としたことである．拡張した AVL 木の平衡は，挿入した節点から根に向かって部分木単位で平衡を行うことは既存の AVL 木のアルゴリズムと変わらない．拡張した AVL 木の平衡も既存の AVL 木と同様に，左・右部分木のみの高さの差で平衡をすることを踏襲している．これに加え，データを挿入した時点でなるべく部分木の根に進数の中央値となるようにしている．これによって，前と後部分木についても，回転操作はないが平衡となるようにしている．また，アルゴリズムに基数探索法を利用しているため，共通接頭辞の探索や削除に威力を発揮することがプログラムからも確認できる．提案した木構造でのデータ

の削除は，慎重に操作しないと構造が壊れてしまうが，5 分木の場合は最大 2 桁の比較を行っていることに注目して，データの削除方法を示した．ラベル付き節点の増加で探索の効率の悪化が推測されたが，実質的にデータ量に対して 1% 未満であることが証明され，ラベル付き節点の導入による効率悪化はないと考えられる．探索の計算量は，データ数を n 個としたとき最悪でも $O(\log n)$ となることが証明された．

第 4 章

拡張した AVL 木の評価

4.1 はじめに

本章では，提案する木構造の探索効率を数値実験で確かめる．比較実験は，実験対象を 2 つに分けて行っている．最初の実験は 2 分探索木と AVL 木との数値実験であり，次に B 木との比較実験である．まず，2 分探索木，AVL 木に対して，幾つかの項目を設定して比較を行った．数値実験で扱ったデータは，0〜9 の文字で長さ 10 桁，20 桁，50 桁，100 桁をそれぞれ 10,000,000 個生成している．この数値結果を表とグラフで示し，結果に対する考察を述べている．次に，B 木との比較実験を行った．データは，0〜9 の文字で長さ 100 桁を 10,000,000 個生成し，この数値結果を表に示し，詳細な考察を加えた．

4.2 完全木の考え方

2 分探索木と AVL 木および拡張した AVL 木の比較を行う．まず，構築した各木における効率性の指標となる完全 d 分木を次で定める [51].

定義 4.1. 次を満たす $d\,(d \geq 2)$ 分木を完全 d 分木と定義する．

高さ $h\,(h \geq 1)$ である d 分木において，深さが $(h-1)$ 以下の任意の節点は必ず d 個の子をもつ． □

高さ h の完全 2 分木における節点数 n_h を求める．節点数を n_i $(0 \leq i \in \mathbb{Z})$ とすると

$$
\begin{aligned}
&\text{深さ } 0 \text{ のとき, 節点数は, } n_0 = 2^0 \\
&\text{深さ } 1 \text{ のとき, 節点数は, } n_1 = 2^0 + 2^1 \\
&\text{深さ } 2 \text{ のとき, 節点数は, } n_2 = 2^0 + 2^1 + 2^2 \\
&\qquad \cdots \\
&\text{深さ } h \text{ のとき, 節点数は, } n_h = 2^{h+1} - 1
\end{aligned} \tag{4.1}
$$

であり，高さ h である完全 2 分木の節点数 n_h が求まる．完全 2 分木となった場合に，最も効率のよい探索が行われる．

同様に，高さ h の完全 5 分木における節点数 n_h を求める．節点数を n_i $(0 \leq i \in \mathbb{Z})$ とすると

$$
\begin{aligned}
&\text{深さ } 0 \text{ のとき, 節点数は, } n_0 = 5^0 \\
&\text{深さ } 1 \text{ のとき, 節点数は, } n_1 = 5^0 + 5^1 \\
&\text{深さ } 2 \text{ のとき, 節点数は, } n_2 = 5^0 + 5^1 + 5^2 \\
&\qquad \cdots \\
&\text{深さ } h \text{ のとき, 節点数は, } n_h = \frac{5^{h+1} - 1}{4}
\end{aligned} \tag{4.2}
$$

であり，高さ h である完全 5 分木の節点数 n_h が求まる．

4.3　2 分探索木，AVL 木との比較

2 分探索木，AVL 木，拡張した AVL 木に対して，幾つかの項目を設定して比較を行う．実験で使用するデータの条件を箇条書きで以下に示す．なお，数値実験に使用したプログラムは，2 分探索木は付録 A であり，AVL 木は付録 B である．

- 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，10 桁，20 桁，50 桁，100 桁とする（$S = 10, 20, 50, 100$）.
- データの数は，各桁とも 10,000,000 個である．（$n = 10,000,000$）
- 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でいる．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 4.1[*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．

疑似乱数をファイルに書き込むプログラムをソースコード 4.1 に示す．現在の時刻を種に M 進数の乱数を発生させ，23 行目の putc 関数でデータ長 S，データ数 N でファイルに文字列として書き込むプログラムである．

ソースコード 4.1　M 進数の乱数ファイル

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define M xxx                                   // M進数
5  #define N xxx                                   // データ数
6  #define S xxx                                   // データの長さ
7
8  int main(void)
9  {
10   FILE *file;                                   // ファイルポインタ
11   int n, s;                                     // カウント
12   char c;                                       // 文字
13
14   if((file = fopen("xxx", "w")) == NULL) {
15     printf("can't open file\n");
16     exit(1);
17   }
18   srand(time(NULL));
19
20   for (n = 0; n < N; n++) {
21     for (s = 0; s < S; s++) {
22       c = rand() % M + '0';
23       putc(c, file);
24     }
25   }
26   fclose(file);
27
28   return ;
29 }
```

表 4.1 に，$S = 10, 20, 50, 100$ の場合の 2 分探索木，AVL 木と拡張した AVL 木の比較結果を載せる．図 4.1 に（1）と（2）をグラフ化したものを示す．図 4.1 では，各々の桁数 S において（1）データを木に挿入するのに要した時間を折れ線グラフで，（2）その時のメモリ使用量を棒グラフで表している．実験結果の表 4.1 での比較項目（1）〜（6）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（2 分探索木，AVL 木，拡張した AVL 木）に

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

挿入し，データ構造を構築するのに要した時間である．表 4.1 での項目 time 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 4.1 での項目 memory 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）深さの平均

与えられた全てのデータを各木構造に挿入したときの，根から各節点までの深さの平均である．表 4.1 での項目 depth に該当する．

（4）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 4.1 での項目 compare に該当する．

（5）回転数（回）

与えられた全てのデータを各木構造に挿入したとき，AVL 木と拡張した AVL 木において必要となる一重回転と二重回転の回数である．表 4.1 での項目 single, double にそれぞれ該当する．

（6）ラベルの個数（個）

拡張した AVL 木のみ，ラベルの個数を記している．表 4.1 での項目 label に該当する．

表 4.1 に，$S = 10, 20, 50, 100$ の場合の 2 分探索木，AVL 木と拡張した AVL 木の比較結果を載せる．図 4.1 に（1）と（2）をグラフ化したものを示す．図 4.1 では，各々の桁数 S において（1）データを木に挿入するのに要した時間を折れ線グラフで，（2）その時のメモリ使用量を棒グラフで表している．

表 4.1 と図 4.1 から，各木の比較と拡張した AVL 木の特徴を述べる．なお，乱数により生成したデータのうち，$S = 10$ におけるデータ数 $7,475$ 個（平均値）は，既に木に挿入しているデータと一致した．そのため，$S = 10$ においては節点数は $9,992,525$ 個となる．他の S では重複するデータは存在しなかった．

表 4.1　2 分探索木および従来の AVL 木との比較 ($n = 10,0000,000$)

	項目 $\backslash S$	10	20	50	100
binary tree	time	25.00	26.14	32.34	43.08
	memory	181.20	276.57	562.67	1039.51
	depth	29.30	29.87	29.47	29.13
	compare	1,103,373,285	1,109,017,665	1,105,470,641	1,101,540,112
AVL tree	time	23.39	26.07	31.25	38.07
	memory	295.64	391.01	677.11	1153.95
	depth	21.69	21.71	21.70	21.69
	compare	878,915,647	878,986,886	879,202,647	879,453,238
	single	2,331,313	2,333,705	2,333,606	2,333,100
	double	2,322,557	2,324,774	2,324,524	2,324,749
ext AVL	time	12.91	14.34	19.28	26.60
	memory	489.76	594.21	906.98	1428.19
	depth	15.15	15.16	15.21	15.18
	compare	197,289,980	197,331,781	197,797,854	197,561,008
	single	663,034	662,918	662,962	663,148
	double	569,203	569,946	569,442	569,672
	label	926,702	931,211	931,489	930,105

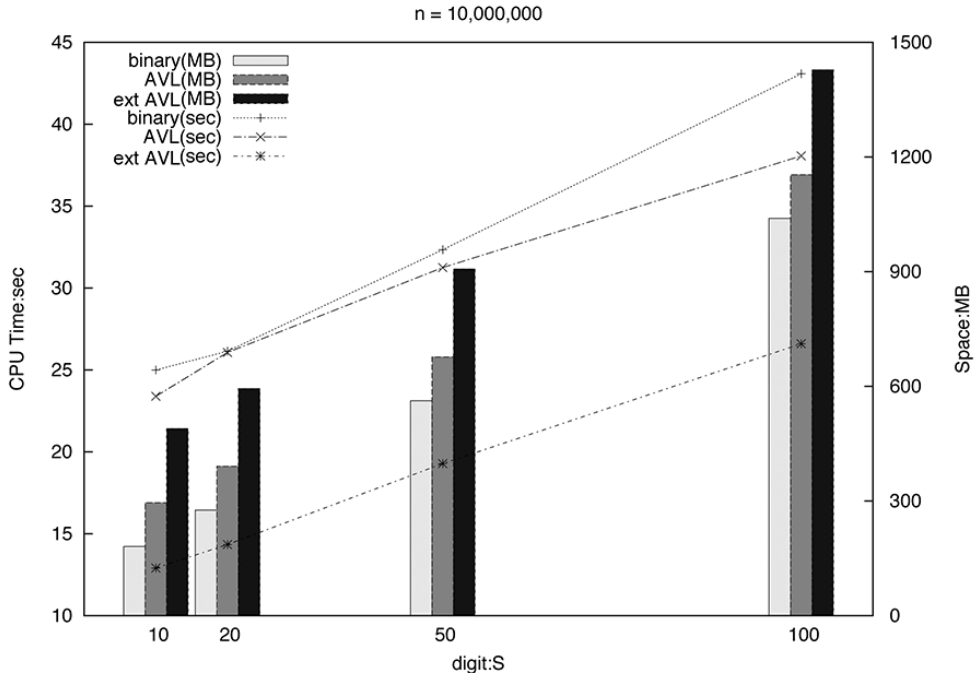


図 4.1　S 桁における各木の構築時間と領域量の比較

まず，2 分探索木と AVL 木における比較項目の結果を考察する．深さの平均（3）であるが，AVL 木の深さの平均は 2 分探索木の 73% もしくは 74% であり，比較回数（4）の割合は約 80% とほぼ一定であるが，$S = 100$ のときになって初めて，データを木に挿入するのに要した時間（1）の差が顕著に表れる．それぞれのプログラムを関数当りで解析した結果，ほとんどの時間が挿入における探索の時間と予めランダムな数値を記述したファイルから読み込む時間であり，AVL 木を平衡するための時間は比較的短いことが分かった．平衡化するための時間をあまり必要とせずに比較回数が小さくなることから，AVL 木の回転による平衡化は構築時間に対して，ほとんど無視できる時間であると言える．

拡張した AVL 木と他の木の各々の比較項目の結果を考察する．まず，深さの平均（3）であるが，拡張した AVL 木の深さ（根から各「通常の節点」と各「ラベル付き節点」までの深さの総和をデータ数で割った値）の平均は，2 分探索木の 51% もしくは 52% であり，AVL 木の約 70% である．これは他の木の 2 分木から 5 分木へとすることで，節点を分散させることで深さが小さくなっていることを示している．深さが小さいことは，平衡化による影響が親節点へ伝播することを抑えることから，構築時間が小さくなっている．次に比較回数（4）であるが，2 分探索木の約 18% であり，AVL 木の約 22〜23% である．比較回数は挿入時における節点データとの探索回数の総和であり，この絶対数が小さいことは構築時間が小さいことを意味する．また平衡のための回転数（5）であるが，一重・二重回転の実施回数は AVL 木よりも，それぞれ約 28%，25% となっている．これは，拡張した AVL 木においては，前部分木・後部分木・中央部分木に進んだ時点で平衡を行なわず，部分木で回転操作を行っていることによる．これに加えて深さの平均（3）が小さいことから，拡張した AVL 木では回転数が小さくなっている．ラベルの個数（6）は，拡張した AVL 木におけるデータ数に対するラベル数の割合であり，約 9% となっている．表 3.2 では，S の値に依らず 10 進数でデータ数に対してラベル数の割合は約 1% である．今回の数値実験では，データ数 $10,000,000$（$S = 7$）だが，S の値を 7 より大きくしたためにラベルの数が増えてラベル数の割合が大きくなっている．ラベル数の割合が増えたことは，深さが大きくなり比較回数の増大になるが，上の比較項目（3）と（4）で述べたように，拡張した AVL 木は深さの平均，比較回数とも減少した構造であると言える．

比較項目（3）〜（6）を集約したものが比較項目（1）である．データを木に挿入するのに要した時間（1）に対して，拡張した AVL 木の構築時間は 2 分探索木の 52%〜62% で

あり，AVL 木の 55%〜70% である．S が大きくなるほどに，他の木の構築時間に対する拡張した AVL 木の割合が大きくなってくる傾向があるが，現実的な S の値では優位性があると言える．短所としては（2）メモリ使用量であり，100 桁で 2 分探索木の約 1.4 倍，AVL 木の約 1.2 倍であるが，図 3.1 の 8〜10 行目は，文字型の 1 バイトずつでそれぞれ表現できるので，7 行目に一緒に格納すれば，文字列が 100 桁の場合で約 7.6% の総容量の減少になる．

4.4　B 木との比較

4.3 節では分岐数の少ない 2 分木である以上，探索の速さの評価は順当であると考えられる．今回の実験では，多分木である B 木との領域計算量，構築時間と探索時間を比較する．5 分木に拡張した AVL 木と分岐数を同じとする 5 分木の B 木を構築して比較実験を行う．[*2] 数値実験に使用したプログラムは，付録 C として記載してある．

4.4.1　拡張した AVL 木のデータ構造の改良

4.3 節の数値実験で，拡張した AVL 木の短所はメモリ使用量であった．今回の実験では，メモリ使用量を以前に比べて減少させている．改良点は，ソースコード 3.1 に示した拡張した AVL 木のデータ構造の平衡条件を維持する 3 つの情報は，各 1 バイトずつで十分なので文字列データの同じ配列に格納することである．具体的には，ソースコード 3.1 に示した拡張した AVL 木のデータ構造の 8〜10 行目をソースコード 4.2 の 3 行目に格納することで領域量の削減を行う．つまり，3 行目で S 桁の文字列，終端記号と 3 バイトの平衡条件の情報を格納する．4 行目から 8 行目は各節点へのポインタであり，9 行目は親節点へのポインタである．1 節点当たりの領域の理論値は，$S+37$ バイトから $S+28$ バイトになり，9 バイトの削減を行っている．

ソースコード 4.2　データ構造の改良

```
1 // データ構造の改良
2 struct ext_avl {
3         char element[S+4];          // 文字列と平衡条件
```

[*2] CPU Intel Xeon 2.33GHz, Memory 8GB, OS Red Hat Enterprise Linux 5.5, GCC 4.1.2

```
4           struct ext_avl *left;
5           struct ext_avl *right;
6           struct ext_avl *front;
7           struct ext_avl *back;
8           struct ext_avl *center;
9           struct ext_avl *parent;
10  };
```

4.4.2　B 木のデータ構造

B 木のデータ構造は，ソースコード 4.3 である．このデータ構造は，参考文献 [43] 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 2.7 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，外部節点の場合はキーと 7 行目の文字列データで構成する．

ソースコード 4.3　B 木のデータ構造

```
1   typedef struct STnode* link;
2   // 配列の構造
3   typedef struct {
4     char key[S+1];              // キー
5     union {
6       link next;                // 節点へのリンク
7       char element[S+1];        // データ
8     }ref;
9   }entry;
10
11  // 節点の構造
12  struct STnode {
13    entry b[k+1];               // k次のB木
14    int m;                      // キーの個数
15  };
```

定義 2.7 の k は奇数であるが，13 行目の $k+1$ 個の通り，配列 b の偶数個を用意する．配列 b の 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 4.2 は，このデータ構造を視覚化したものである．図は 5 分木の B 木であり，1 節点を 6 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で，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 2.7 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である．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．

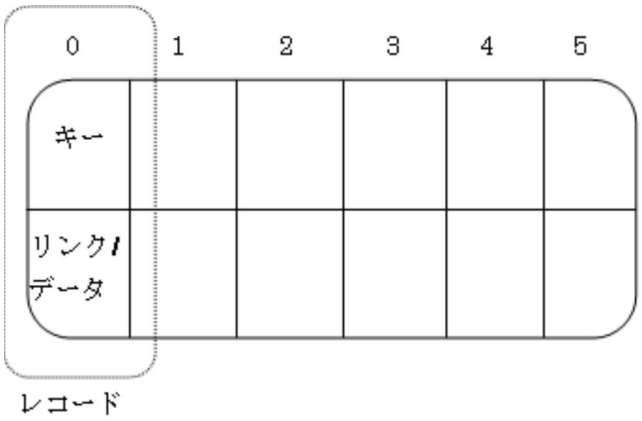


図 4.2　B 木の節点の構造

図 4.3 は，根節点に文字列のキーとデータ「0, 10, 100, 1000, 10000」を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 4.3 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 4.4 となる．根で分割したときに高さが 1 高くなる．

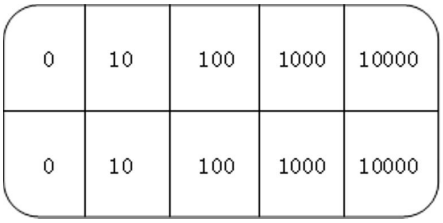


図 4.3　根節点におけるキーの満杯の様子

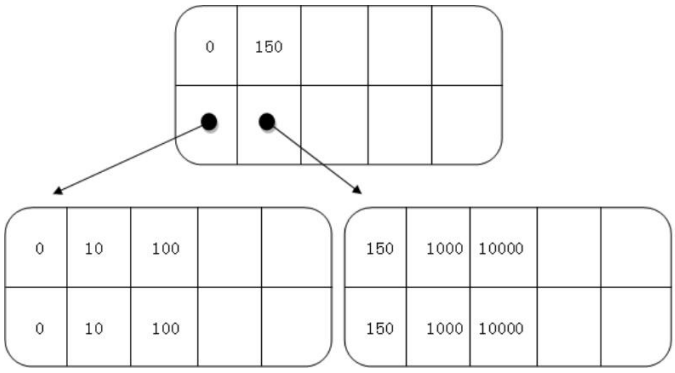


図 4.4　高さの増加（根節点の分割）

4.4.3　B 木の構築方法

B 木のプログラムの付録 C を参照して，アルゴリズムのプログラム化を説明する．これまで通り，数字はアルゴリズムを実行する順番，英字は場合分けを示している．いずれも入れ子の構造をとる．今回のプログラムでは，キーの値と文字列のデータ値を同じとして，プログラム化している．節点に格納されているキーおよび文字列データを「節点データ」で，探索するキーを「探索データ」と呼ぶことにする．アルゴリズムの説明は，次の操作のみを説明する．（その他のアルゴリズムは, 付録 C を参照）

- 探索
- 挿入

T における各操作を説明するが，数字はアルゴリズムを実行する順番であり，英字は場合分けを表している．いずれも入れ子の構造をとる．

根における処理

探索および挿入の処理は，126 行目における「STinsert 関数」からはじまる．STinsert 関数は，文字列データを引数とし根節点から探索を始め，すべての処理が終わったら，この関数に戻ってくるが，根節点が分割したかを判定する．

1　**「insertR 関数」を実行する．**

2 a **根節点が分割していない場合は，終了する．**

b **根節点が分割した場合**

1 **新しい節点を作成する．**

2 **新しい節点のデータ数を 2 とする．**

3 **分割した 2 節点へのリンクとキーを新しい節点へ，それぞれ代入する．**

4 **根を新しい節点とし，高さを 1 高くする．**

insertR 関数（探索および挿入）

節点における実質的な探索および挿入の関数である．内・外部節点における処理を 1 つの関数に再帰的処理としてまとめている．154 行目からの insertR 関数からはじまる．

1 a **外部節点の場合**

a **節点データが大きい場合は，c の「探索データを挿入する」へ移動する．**

b **探索データが大きい場合は，次のデータと比較する．**

c **同文字列の場合は，何も処理を行わない（分割しなかった印を返す）．**

b **内部節点の場合**

a **同文字列の場合は，何も処理を行わない（分割しなかった印を返す）．**

b **節点データを全て比較した場合または節点データが大きい場合**

-> **子節点へ辿り「insertR 関数」の再帰を行う．**

1 **節点が分割していなければ，分割しなかった印を返す．**

2 **節点が分割していれば，後半の節点へのリンクを親節点へ代入する．**

3 **「探索データを挿入する」へ移動する．**

c **探索データが大きい場合 -> 次のデータと比較する．**

2 **探索データを挿入する．**

1 **適切な配列に探索データを挿入する．**

a **k 以下であれば，分割しなかった印を返す．**

b **k より大きいなら「split 関数」を実行し新節点へのポインタを返す．**

split 関数（節点の分割）

節点の分割である．節点におけるキーの数を偶数個 $k+1$ にしているので，常に半分ずつを分割すればよい．239 行目からの「split 関数」からはじまる．

1 **新しい節点を生成する．**
2 **配列の後半を新しい節点に移行する．**
3 **分割された 2 個の節点のデータ数を更新する（データ数は k/2 個ずつ配分）．**
4 **新しく生成された節点へのポインタを返す．**

4.4.4 領域計算量の比較

4.4.2 節で説明した 5 次の B 木を構築すると，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．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．

5 分木に拡張した AVL 木の領域量は，式（4.3）である．1 節点当たり $S+28$ バイト（ソースコード 3.1）とラベル数の割合は $m=10$ の場合で表 3.2 より 1.010% であるので，

$$n(S+28)+\frac{101}{10000}n(S+28)\approx n\log n+28n \tag{4.3}$$

となる (log は常用対数).

一方の B 木では，もっとも高さが最も小さくなる場合でバイト数を試算してみる．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（式 (4.2)）．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，

$$\frac{\{6\,(S+5)+4\}\left(5^h-1\right)}{4}=\frac{n-5}{10}\,(3\log n+17) \tag{4.4}$$

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，

$$5^h\{12\,(S+1)+4\}=\frac{n}{5}\,(12\log n+16) \tag{4.5}$$

となる．式 (4.4) と式 (4.5) より，B 木における節点のバイト数は，

$$(4.4) + (4.5) \approx 2.7n \log n + 4.9n \tag{4.6}$$

である．式 (4.6) と式 (4.3) の差は，

$$f(n) = 1.7n \log n - 23.1n \tag{4.7}$$

となる．式 (4.7) 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．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．

数値実験による計算量の比較

付録 C（B 木）と D（拡張した AVL 木）を用いて，数値実験を行った．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．実験で使用するデータの条件を箇条書きで以下に示す．

- 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，各桁とも 10,000,000 個である．（$n = 10,000,000$）
- 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でいる．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 4.2 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 4.2 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．

表 4.2 と表 4.3 に，拡張した AVL 木と B 木の比較結果を載せる．表 4.2 と表 4.3 での比較項目（1）〜（6）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 4.2 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 4.2 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 4.2 での比較回数に該当する．

(4) 探索時間（秒）

表 4.2 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 4.3 での探索時間に該当する．

(5) 探索における比較回数（回）

表 4.2 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 4.3 での比較回数に該当する．

表 4.2　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 4.2 の 5 分木に拡張した AVL 木と 5 分木の B 木について結果を説明する．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．ただし，B 木でも文字列を先頭に後戻りしないアルゴリズムが存在すれば，比較回数と構築時間は改善すると考える．一方，領域量では，構築した B 木は高さ 12 であり，$n = 1,220,703,125$ 個のデータを格納できる木ということが分かる．B 木では，かなり無駄な領域が生じており，100 桁の文字列

(10^{100} 個）からデータ数 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・平衡化するので，領域はデータ数 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 $n = 1,000,000$ 個探索するのに掛かった時間（秒）と比較回数（回）を表 4.3 に示す．

表 4.3　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

表 4.3 の 5 分木に拡張した AVL 木と 5 分木の B 木について結果を説明する．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% である．B 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．悪化かどうかを 2 つの木の各項目を検討してみる．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よい探索になっていると考えられる．

4.5　まとめ

この章では，提案する AVL 木の効率性の数値実験を行った．数値実験は 2 つに分けて行っており，1 つ目は 2 分探索木と既存の AVL 木であり，2 つ目は 5 分木の B 木である．木の構築時間を測定したが，探索後に挿入が行われるのでデータの探索効率を判断する数値実験である．提案する AVL 木は，アルゴリズムに基数探索法，データ構造は多分木構造となっているが，数値実験から非常に探索効率の良いことが分かった．提案する AVL 木への構築時間は，既存の AVL 木の約 70%，B 木の約 50% となっている．構築時間には探索時間が含まれるが，提案する AVL 木への比較回数は，既存の AVL 木の約 23%，B 木の約 11% となっている．これは，提案する木構造が多分木であることと基数探索法を利用しているからである．多分木であることは，木構造の高さが小さくなり目的の文字列（単語）に早く到達できること，部分木の深さが小さいので平衡化の回転があまり行われることがなく構築時間の短縮に繋がっていると考えられる．一方，構築した木構造に含まれないデータを探索させたときに，提案する AVL 木の比較回数の比は B 木に対して 1% だけであった．同じ多分木である B 木に対して良い結果となった要因は，提案する AVL 木はアルゴリズムに基数探索法の採用が可能であったからと考えられる．提案する AVL 木の短所は，領域量の増加である．数値データの文字列の長さが 100 桁の場合で，提案する AVL 木の領域量は既存の AVL 木の 1.2 倍である．この領域量の削減については，次の章で考察する．

第 5 章

拡張した AVL 木の領域量の削減と一般化

5.1 はじめに

拡張した AVL 木は，2 分探索木や既存の AVL 木に比べて，効率の良い探索が可能であったが，データ構造のメモリ使用量では 100 桁の場合で AVL 木の約 1.2 倍（4.3 節）であった．拡張した AVL 木は，分岐数を増やすことで探索効率が向上したが，逆に領域量は増えている．一般的に，時間計算量と空間計算量の間には，トレードオフの関係が存在する．4.4 節では，平衡条件の情報を文字列データの方に含めることで，領域量の削減を行ったが，この章では，領域量を削減する方法を提案する．さらに，第 3 章で提案した「拡張した AVL 木」は，ある節点において，最大 k 桁（$k \geq 1$）の比較を行うと，$2k+1$ 分木として一般化できる．この章では，既存の AVL 木を奇数木に一般化する方法について考察する．

5.2 拡張した AVL 木の領域量の削減

拡張した AVL 木の領域量を減らすには，次の 2 つの方法が考えられる．

1. 共通接頭辞となる文字列の削減
2. 節点におけるポインタの削減

「共通接頭辞となる文字列の削減」を 5.2.1 節で，「節点におけるポインタの削減」を 5.2.2 節で提案する．

5.2.1　共通接頭辞となる文字列の削減

拡張した AVL 木のアルゴリズムは，基数探索法を利用して上位の桁から比較を行い，桁値が不一致ならば左または右部分木へ進行しそのままの桁を比較，桁値が 1 桁一致ならば前または後部分木に進行，桁値が 2 桁一致すれば中央部分木に進行するものであった．基数探索法のアルゴリズムでは，ある節点で比較している桁の上位の文字列は，親節点にも同一の文字が含まれることになり，各節点の配列には先頭からの同一の文字を除いた文字列のみを格納すればよいことになる．従来の構造は全ての文字列データを節点に格納していたが，今回の改良する構造では共通接頭辞となる文字列を削減する．

どの程度の文字列を削減できるかを算出する．図 5.1 の表は，10 進数の文字列で 5 桁の数字を配列に格納している．各行につき，00000〜99999 の 5 桁の数字を格納している．升目の数は 5×10^5 個であり，C 言語では 1 桁が 1 バイトであるから，この文字列の領域量は 5×10^5 バイトである．このデータ構造において，木に文字列データ「00000」が存在して「00001」を挿入するならば，上位桁の「0000」は同一の文字であり実質不要である．つまり，節点は「00000」と「1」の文字をを保存すればよく，4 バイトを削減できる．

	5桁	4桁	3桁	2桁	1桁
10^5	0	0	0	0	0
	0	0	0	0	1
		·	·	·	
		·	·	·	
	9	9	9	9	9

図 5.1　5 桁の数値の全配列構造

基数探索法において必要な文字（桁）は，5 桁目で数字の種類は 10 文字である．4 桁目は 5 桁目の文字と合わせて 10^2 文字である．3 桁目以降も同様に考えて，10^3 文字，4 桁目で 10^4 文字，5 桁目で 10^5 文字である．これらの文字を合わせると 111,110 文字であり，基数探索法では 111,110 文字で 5 桁の全数字を表現可能である．これまでのデータ構造では 5×10^5 文字が必要であったが，今回の文字数は，これまで文字数の 22.2% です

み，重複する文字数は，$388,890\left(=5\times10^5-111,110\right)$ 文字である．

10 進数の文字列で 10 桁の場合，従来のデータ構造は $10^{11}\left(=10\times10^{10}\right)$ 文字が必要である．基数探索法において必要な文字数は，$11,111,111,110\left(=\sum_{k=1}^{10}10^k\right)$ 文字であり，従来の文字数の 11.1% である．重複する桁は，88,888,888,890 桁である．

10 進数の文字列で 50 桁の場合，従来のデータ構造は文字 50×10^{50} 文字が必要である．基数探索法において必要な文字数は，$111,111,111,111,111\times10^{36}\left(=\sum_{k=1}^{50}10^k\right)$ 文字であり，従来の文字数の 2.22% である．重複する桁は，$4,888,888,888,888,890\times10^{36}$ 桁である．

以上より，10 進数で文字列の長さを $10^5,10^{10},10^{50}$ とするときの従来の領域量に対する今回の領域量の割合を表 5.1 にまとめる．

- 従来（文字）は，従来のデータ構造で必要な文字数である．
- 今回（文字）は，今回の改善するデータ構造で必要な文字数である．
- 重複（文字）は，従来と今回の文字数の差である．
- 割合（%）は，従来に対する今回の文字数の比である．

表 5.1　従来の領域量に対する今回の領域量の割合（10 進数）

データ数	10^5	10^{10}	10^{50}
従来（文字）	5×10^5	10^{11}	50×10^{50}
今回（文字）	111,110	$111,111\times10^5$	$111,111\times10^{45}$
重複（文字）	388,890	$888,889\times10^5$	$488,889\times10^{46}$
割合（%）	22.2	11.1	2.22

表 5.1 より，文字列が長くなるほど，従来の文字数に対する今回の文字数の割合が小さくなることが分かる．

次に，アルファベットの 26 文字で 5 桁の文字列を考察してみる．アルファベットの文字列で 5 桁の場合，従来のデータ構造は $59,406,880\left(=5\times26^5\right)$ 文字が必要である．基数探索法において必要な文字数は $12,356,630\left(=\sum_{k=1}^{5}26^k\right)$ 文字であり，従来の文字数の 20.8% である．重複する文字数は，47,050,250 文字である．

アルファベットの 26 文字で 10 桁の文字列を考察してみる．従来のデータ構造は

$1,411,670,956,533,760\ (=10\times26^{10})$ 文字が必要である．基数探索法において必要な文字数は，$146,813,779,479,510\ \left(=\sum_{k=1}^{10}26^k\right)$ 文字であり，従来の文字数の 10.4% である．重複する文字数は，1,264,857,177,054,250 文字である．

アルファベットの 26 文字で 50 桁の文字列を考察してみる．従来のデータ構造は $2,803,092,328,832,100\times10^{57}\ (=50\times26^{50})$ 文字が必要であった．基数探索法において必要な文字数は，$58,304,320,439,707,600\times10^{54}\ \left(=\sum_{k=1}^{50}26^k\right)$ 文字であり，従来の文字数の 2.08% である．重複する文字数は，$2,744,788,008,392,390\times10^{57}$ 文字である．

以上より，26 進数で文字列の長さを $26^5, 26^{10}, 26^{50}$ とするときの従来の領域量に対する今回の領域量の割合を表 5.2 にまとめる．

表 5.2　従来の領域量に対する今回の領域量の割合（26 進数）

データ数	26^5	26^{10}	26^{50}
従来（文字）	59,406,880	$14,116,710\times10^8$	$28,030,923\times10^{65}$
今回（文字）	12,356,630	$14,681,378\times10^7$	$58,304,320\times10^{63}$
重複（文字）	47,050,250	$12,648,572\times10^8$	$27,447,880\times10^{65}$
割合（%）	20.8	10.4	2.08

表 5.2 より，文字列が長くなるほど，従来の文字数に対する今回の文字数の割合が小さくなることが分かる．

最後に，五十音の 50 文字で 5 桁の文字列を考察してみる．五十音の文字列で 5 桁の場合，従来のデータ構造は $1,562,500,000\ (=5\times50^5)$ 文字が必要である．基数探索法において必要な文字数は $318,877,550\ \left(=\sum_{k=1}^{5}50^k\right)$ 文字であり，従来の文字数の 20.4% である．重複していた文字数は，1,243,622,450 文字である．

五十音の 50 文字で 10 桁の文字列を考察してみる．従来のデータ構造は $976,562,500\times10^9\ (=10\times50^{10})$ 文字が必要である．基数探索法において必要な文字数は，$99,649,234,693,877,600\ \left(=\sum_{k=1}^{10}50^k\right)$ 文字であり，従来の文字数の 10.2% である．重複していた文字数は，$876,913,265,306,122,000$ 文字である．

五十音の 50 文字で 50 桁の文字列を考察してみる．従来のデータ構造は $50\times50^{50}\ (=444,089,209,850,063\times10^{72})$ 文字が必要である．基数探索法において必要な文字数は，$\sum_{k=1}^{50}50^k\ (=9,063,045,098,980,870\times10^{69})$ 文字であり，従来の文字数の

2.04% である．重複していた文字数は，$435,026,164,751,082 \times 10^{72}$ 文字である．

以上より，26 進数で文字列の長さを $26^5, 26^{10}, 26^{50}$ とするときの従来の領域量に対する今回の領域量の割合を表 5.3 にまとめる．

表 5.3　従来の領域量に対する今回の領域量の割合（50 進数）

データ数	50^5	50^{10}	50^{50}
従来（文字）	$15,625,000 \times 10^2$	$97,656,250 \times 10^{10}$	$44,408,921 \times 10^{79}$
今回（文字）	$31,887,755 \times 10$	$99,649,235 \times 10^9$	$90,630,451 \times 10^{76}$
重複（文字）	$12,436,225 \times 10^2$	$87,691,327 \times 10^{10}$	$43,502,616 \times 10^{79}$
割合（%）	20.4	10.2	2.04

表 5.3 より，文字列が長くなるほど，従来の文字数に対する今回の文字数の割合が小さくなることが分かる．

表 5.1，5.2，5.3 より，桁 · 進数が大きくなるほど，従来の文字数に対する共通接頭辞を除いた文字数の割合は小さくなることから，実際の環境下のひらがなや漢字を使う場合では，この削減方法が有効になると考える．

5.2.2 節点におけるポインタの削減

従来のデータ構造（ソースコード 3.1）を改良した構造は，ソースコード 4.2 であった．本章のデータ構造は，ソースコード 4.2 をさらに改良するものである．節点の種類は，「データが格納された節点」と「ラベル付き節点」の 2 種類がある．

「データが格納された節点」は中央部分木が存在しないので，中央部分木へのポインタを削除した構造へと改良する．この改良した構造をソースコード 5.1 に示す．

ソースコード 5.1　「データが格納された節点」でのデータ構造

```
1 // データが格納された節点
2 struct ext_avl {
3         char element[S+4];          // 文字列と平衡条件
4         struct ext_avl *left;
5         struct ext_avl *right;
6         struct ext_avl *front;
7         struct ext_avl *back;
```

```
8           struct ext_avl *parent;
9   };
10  // ラベル付き節点
11      struct ext_avl *center;
```

データが格納された節点に中央部分木ができた場合に，データが格納された節点は「ラベル付き節点」に変わる．このとき，ラベル付き節点に中央部分木へのポインタが生成される．ラベル付き節点はデータではなく単なる識別子である．この識別の文字列は 2 文字のみで十分であるので，この改良した構造をソースコード 5.2 に示す．

ソースコード 5.2　ラベル付き節点

```
1   // ラベル付き節点
2   struct ext_avl {
3           char element[6];            // 識別子の文字列
4           struct ext_avl *left;
5           struct ext_avl *right;
6           struct ext_avl *front;
7           struct ext_avl *back;
8           struct ext_avl *center;
9           struct ext_avl *parent;
10  };
```

これらの構造にすると，1 節点当たりの理論値は「データが格納された節点」の場合は $S + 24$ バイトであり，「ラベル付き節点」の場合は 30 バイトである．

5.2.3　領域量の比較

5.2.1 節と 5.2.2 節で，「データの削減」と「ポインタ数の削減」を行った．この節では，これらの節を考慮して，従来（ソースコード 4.2）と今回（ソースコード 5.1，5.2）の領域量の比較を行ってみる．比較を行うときのデータ数 n を m 進 S 桁とおく．

10 進数での領域量の比較

文字を数値（$m = 10$ 進数）としたときの，従来の領域量と今回の領域量の比較を行う．このとき，最も多く加えられるデータ数 n は，$n = 10^S$ である．

従来の拡張した AVL 木の総バイト数は,

$$n(S+28)+\frac{101}{10000}n(S+28)=1.0101n(\log n+28) \tag{5.1}$$

となる．log は常用対数である．

今回の拡張した AVL 木の総バイト数は，共通接頭辞である重複桁を d とすると,

$$n(S+24)-d+\frac{101}{10000}n(2+28)=n(\log n+24.303)-d \tag{5.2}$$

となる．

10 進数 5 桁での領域量の比較

加えるデータ文字列を 10 進数 5 桁とする．このとき，最も多く加えられるデータ数 n は，$n=10^5$ である．拡張した AVL 木では，10 進数におけるラベル数の割合は 1.010%（表 3.2）である．

式 (5.1) と式 (5.2) に $n=10^5$, $d=5\times10^5-111,110=388,890$ を代入すると，式 (5.1) は,

$$1.0101\times10^5(\log 10^5+28)=3.17891\,(\mathrm{MB}) \tag{5.3}$$

となる．式 (5.2) は,

$$10^5(\log 10^5+24.303)-388,890=2.42368\,(\mathrm{MB}) \tag{5.4}$$

となる．式（5.3）と式（5.4）の差は，0.755234（MB）であり，式 (5.3) に対して，この値の割合は 23.7576% になる．

10 進数 10 桁での領域量の比較

加えるデータ文字列を 10 進数 10 桁とする．このとき，最も多く加えられるデータ数 n は，$n=10^{10}$ である．拡張した AVL 木では，10 進数におけるラベル数の割合は 1.010%（表 3.2）である．

式 (5.1) と式 (5.2) に $n=10^{10}$, $d=88,888,888,890$ を代入すると，式 (5.1) は,

$$1.0101\times10^{10}(\log 10^{10}+28)=357.477\,(\mathrm{GB}) \tag{5.5}$$

であり，式 (5.2) は，

$$10^{10}(\log 10^{10} + 24.303) - 88,888,888,890 = 236.687\,(\mathrm{GB}) \tag{5.6}$$

となる．式（5.5）と式（5.6）の差は，120.79（GB）であり，式 (5.5) に対して，この値の割合は 33.7895% になる．

10 進数 50 桁での領域量の比較

加えるデータ文字列を 10 進数 50 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 10^{50}$ である．拡張した AVL 木では，10 進数におけるラベル数の割合は 1.010%（表 3.2）である．

式 (5.1) と式 (5.2) に $n = 10^{50}$, $d = 4,888,888,888,888,890 \times 10^{36}$ を代入すると，式 (5.1) は，

$$1.0101 \times 10^{50}(\log 10^{50} + 28) = 7.33769\,(\mathrm{GB}) \tag{5.7}$$

であり，式 (5.2) は，

$$10^{50}(\log 10^{50} + 24.303) - 4,888,888,888,888,890 \times 10^{36} = 2.36687\,(\mathrm{GB}) \tag{5.8}$$

となる．式（5.7）と式（5.8）の差は，4.97081×10^{42}（GB）であり，式 (5.7) に対して，この値の割合は 67.7436% になる．

以上より，表 5.4 にデータ数が $n = 10^5, 10^{10}, 10^{50}$ の場合での従来と今回の拡張した AVL 木の領域量の比較を示す．

- 従来（GB）は，従来の 1 節点当たりのデータ構造の容量である．
- 今回（GB）は，改善する 1 節点当たりのデータ構造の容量である．
- 重複（GB）は，従来と今回の容量の差である．
- 削減率（%）は，従来に対する重複する容量の比である．

表 5.4 より，桁が大きくなるほど削減率は大きくなる傾向にある．

表 5.4　10 進数における領域量比較

データ数	10^5	10^{10}	10^{50}
従来（GB）	3.17891（MB）	357.477	7.33769×10^{42}
今回（GB）	2.42368（MB）	236.687	2.36687×10^{42}
重複（GB）	0.755234（MB）	120.79	4.97081×10^{42}
削減率（%）	23.7576	33.7895	67.7436

データがアルファベットの場合での領域量の比較

アルファベットでの領域量の従来と今回の場合の領域量の比較を求めてみる．アルファベット（26 進数）のときの，データ全体に対するラベルの割合は，次の表 5.5 となる．従来のラベルの割合に対する考察の通り，m 進数を固定すれば S の増減に関わらずラベル数の割合は一定である．

表 5.5　26 進数におけるデータ全体に対するラベルの割合

S 桁	10 桁	20 桁	30 桁	40 桁	50 桁
26 進数	0.148%	0.148%	0.148%	0.148%	0.148%

ラベル数の割合は $m = 26$ の場合で 0.148% であるから，従来の拡張した AVL 木の総バイト数は，

$$n(S + 28) + \frac{148}{100000} n(S + 28) = 1.00148n(\log_{26} n + 28) \tag{5.9}$$

となる．

今回の拡張した AVL 木の総バイト数は，共通接頭辞である重複桁を d とすると，

$$n(S + 24) - d + \frac{148}{100000} n(2 + 28) = n(\log_{26} n + 24.0444) - d \tag{5.10}$$

となる．

26 進数 5 桁での領域量の比較

加えるデータ文字列を 26 進数 5 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 26^5$ である．拡張した AVL 木では，26 進数におけるラベルの割合は 0.148% である．

式 (5.9) と式 (5.10) に $n = 26^5$, $d = 47,050,250$ を代入すると，式 (5.9) は，

$$1.00148 \times 26^5(\log_{26} 26^5 + 28) = 374.475\,(\mathrm{MB}) \tag{5.11}$$

であり，式 (5.10) は，

$$26^5(\log_{26} 26^5 + 24.0444) - 47,050,250 = 284.23\,(\mathrm{MB}) \tag{5.12}$$

となる．式（5.11）と式（5.12）の差は，90.2448（MB）であり，式 (5.11) に対して，この値の割合は 24.099% になる．

26 進数 10 桁での領域量の比較

加えるデータ文字列を 26 進数 10 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 26^{10}$ である．拡張した AVL 木では，26 進数におけるラベルの割合は 0.148% である．

式 (5.9) と式 (5.10) に $n = 26^{10}$, $d = 1,264,857,177,054,250$ を代入すると，式 (5.9) は，

$$1.00148 \times 26^{10}(\log_{26} 26^{10} + 28) = 5.00333 \times 10^6\,(\mathrm{GB}) \tag{5.13}$$

であり，式 (5.10) は，

$$26^{10}(\log_{26} 26^{10} + 24.0444) - 1,264,857,177,054,250 = 3.2979 \times 10^6\,(\mathrm{GB}) \tag{5.14}$$

となる．式（5.13）と式（5.14）の差は，1.70544×10^6（GB）であり，式 (5.13) に対して，この値の割合は 34.086% になる．

26 進数 50 桁での領域量の比較

加えるデータ文字列を 26 進数 50 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 26^{50}$ である．拡張した AVL 木では，26 進数におけるラベルの割合は 0.148% である．

式 (5.9) と式 (5.10) に $n = 26^{50}$, $d = 2,744,788,008,392,390 \times 10^{57}$ を代入すると，式 (5.9) は，

$$1.00148 \times 26^{50}(\log_{26} 26^{50} + 28) = 4.07854 \times 10^{63}\,(\mathrm{GB}) \tag{5.15}$$

であり，式 (5.10) は，

$$26^{50}(\log_{26} 26^{50} + 24.0444) - 2,744,788,008,392,390 \times 10^{57} = 1.3097 \times 10^{63}\,(\mathrm{GB}) \tag{5.16}$$

となる．式（5.15）と式（5.16）の差は，2.76884×10^{63}（GB）であり，式 (5.15) に対して，この値の割合は 67.888% になる．

以上より，表 5.6 にデータ数が $n = 26^5, 26^{10}, 26^{50}$ の場合での従来と今回の拡張した AVL 木の領域量の比較を示す．

表 5.6　26 進数における領域量比較

データ数	26^5	26^{10}	26^{50}
従来（GB）	374.475（MB）	5.00333×10^6	4.07854×10^{63}
今回（GB）	284.23（MB）	3.2979×10^6	1.3097×10^{63}
重複（GB）	90.2448（MB）	1.70544×10^6	2.76884×10^{63}
削減率（%）	24.099	34.086	67.888

表 5.6 より，桁が大きくなるほど削減率は大きくなる傾向にある．

データが五十音の場合での領域量の比較

最後に五十音での領域量の従来と今回の場合の領域量の比較を求めてみる．五十音（50 進数）のときの，データ全体に対するラベルの割合は，次の表 5.7 となる．従来のラベルの割合に対する考察の通り，m 進数を固定すれば S の増減に関わらずラベル数の割合は

一定である．

表 5.7　50 進数におけるデータ全体に対するラベルの割合

S 桁	10 桁	20 桁	30 桁	40 桁	50 桁
26 進数	0.040%	0.040%	0.040%	0.040%	0.040%

ラベル数の割合は $m = 50$ の場合で 0.040% であるから，従来の拡張した AVL 木の総バイト数は，

$$n(S + 28) + \frac{4}{10000} n(S + 28) = 1.0004n(\log_{26} n + 28) \tag{5.17}$$

となる．

今回の拡張した AVL 木の総バイト数は，共通接頭辞である重複桁を d とすると，

$$n(S + 24) - d + \frac{4}{10000} n(2 + 28) = n(\log_{50} n + 24.012) - d \tag{5.18}$$

となる．

50 進数 5 桁での領域量の比較

加えるデータ文字列を 26 進数 5 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 50^5$ である．拡張した AVL 木では，26 進数におけるラベルの割合は 0.040% である．

式 (5.17) と式 (5.18) に $n = 50^5$, $d = 1,243,622,450$ を代入すると，式 (5.17) は，

$$1.0004 \times 50^5(\log_{50} 50^5 + 28) = 9.60811\,(\mathrm{GB}) \tag{5.19}$$

であり，式 (5.18) は，

$$50^5(\log_{50} 50^5 + 24.012) - 1,243,622,450 = 7.28539\,(\mathrm{GB}) \tag{5.20}$$

となる．式（5.19）と式（5.20）の差は，2.32272（GB）であり，式 (5.19) に対して，この値の割合は 24.1745% になる．

50 進数 10 桁での領域量の比較

加えるデータ文字列を 50 進数 10 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 50^{10}$ である．拡張した AVL 木では，50 進数におけるラベルの割合は 0.040% である．

式 (5.17) と式 (5.18) に $n = 50^{10}$, $d = 876,913,265,306,122,000$ を代入すると，式 (5.17) は，

$$1.004 \times 50^{10}(\log_{50} 50^{10} + 28) = 3.45746 \times 10^{9}\,(\mathrm{GB}) \tag{5.21}$$

であり，式 (5.18) は，

$$50^{10}(\log_{50} 50^{10} + 24.012) - 876,913,265,306,122,000 = 2.27668 \times 10^{9}\,(\mathrm{GB}) \tag{5.22}$$

となる．式（5.21）と式（5.22）の差は，1.18078×10^{9}（GB）であり，式 (5.21) に対して，この値の割合は 34.1516% になる．

50 進数 50 桁での領域量の比較

加えるデータ文字列を 50 進数 50 桁とする．このとき，最も多く加えられるデータ数 n は，$n = 50^{50}$ である．拡張した AVL 木では，50 進数におけるラベルの割合は 0.040% である．

式 (5.17) と式 (5.18) に $n = 50^{50}$, $d = 435,026,164,751,082 \times 10^{72}$ を代入すると，式 (5.17) は，

$$1.004 \times 50^{50}(\log_{50} 50^{50} + 28) = 6.47782 \times 10^{77} \tag{5.23}$$

であり，式 (5.18) は，

$$50^{50}(\log_{50} 50^{50} + 24.012) - 435,026,164,751,082 \times 10^{72} = 2.07063 \times 10^{77} \tag{5.24}$$

となる．式（5.23）と式（5.24）の差は，4.40718×10^{77}（GB） であり，式 (5.23) に対して，この値の割合は 68.035% になる．

以上より，表 5.8 にデータ数が $n = 50^{5}, 50^{10}, 50^{50}$ の場合での従来と今回の拡張した AVL 木の領域量の比較を示す．

表 5.8　50 進数における領域量比較

データ数	50^5	50^{10}	50^{50}
従来（GB）	9.60811	3.45746×10^9	6.47782×10^{77}
今回（GB）	7.28539	2.27668×10^9	2.07063×10^{77}
重複（GB）	2.32272	1.18078×10^9	4.40718×10^{77}
削減率（%）	24.1745	34.1516	68.035

表 5.8 より，桁が大きくなるほど削減率は大きくなる傾向にある．

表 5.4，5.6，5.8 より，桁が大きくなるほど，削減率は大きくなる傾向にある．これより，実際の環境下で長い単語などを扱った場合に，この削減方法の効果が出ると思われる．

第 4 章では，10 進 50 桁の場合において既存の AVL 木の領域量に対して拡張した AVL 木の領域量は，1.2 倍であった．この値がどの程度まで改善できたかを調べる．なお，第 4 章での数値実験では，データ数を 10,000,000 個を選び出したが，今回の算定ではデータ数を 10^{50} 個選んでおり，そのまま比較できない．

既存の AVL 木は，AVL 木の領域量を AVL とすると 1 節点当たり $S + 21$ バイトより，

$$\mathrm{AVL} = \frac{10^{50} \times 71}{1024^3}$$

で AVL 木の領域量がメガバイト単位で求められる．

一方の拡張した AVL 木の領域量を extAVL とおくと，式（5.2）より，

$$\mathrm{extAVL} = \frac{10^{50} \times 124.303 - \sum_{k=1}^{50} 10^k}{1024^3}$$

で拡張した AVL 木の領域量がメガバイト単位で求められる．

既存の AVL 木の領域量に対する拡張した AVL 木の領域量を求めると，

$$\frac{\mathrm{extAVL}}{\mathrm{AVL}} \approx 1.04$$

となり，1.2 倍から改善した結果が出ている．

5.3　$2k+1$ 分木の一般化

拡張した AVL 木では，ある節点において最大 k 桁（$1 \leq k$）の比較を行うとすると，$2k+1$ 分木として一般化できる．

図 5.2 は，3 分木に拡張した AVL 木である．3 分木ではある節点は，左，右，中央部分木への節点があり，構造は第 3 章の定義 3.1 に準ずる．図 5.2 では，節点に移動する桁数を記入しており，中央部分木をもつ節点は，ラベル付き節点であり太い節点で示している．左または右部分木に進行した場合は比較する文字（桁）は移動しない 0 桁移動であり，中央部分木に進行した場合は比較する桁が 1 桁移動している．図の（a）は右部分木の高さが左部分木の高さより 2 高いので，二重回転の操作を行ったのが（b）である．3 分

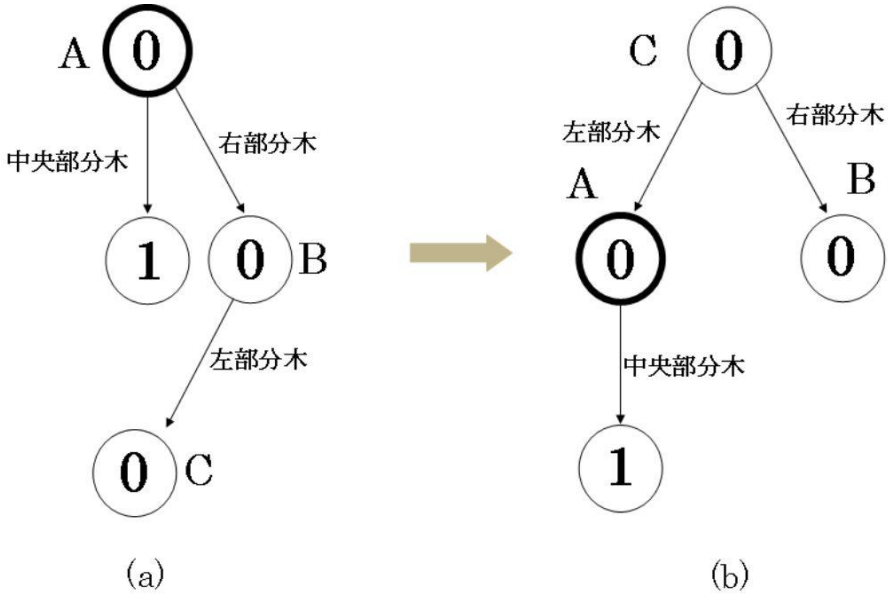


図 5.2　3 分木に拡張した AVL 木

木の例を示したが，他の k でも移動する桁が同値の節点同士で回転の操作を行えば，構造の矛盾は起こらない．

$2k+1$ 分木に対する操作を以下に限定して，主に時間計算量を考察する．

- 探索

- 挿入
- 削除

5.3.1 探索

$2k+1$ 分木において，探索の時間計算量は k の値が大きいほど木の高さが小さくなるので，時間計算量が小さくなると考えがちであるが，基数探索法での探索では木の高さが直接的には時間計算量とならない．節点のキー全体で比較するような通常の探索であれば，木の高さが時間計算量となる．しかし，基数探索法では桁単位で比較しており桁の比較回数が時間計算量となる．つまり，木の高さが違っていても，同じ比較回数で目的のデータに辿り着くならば，同じ計算量となる．実際の環境では，k の値が大きいほど分岐数が多く目的のデータに早く辿り着くので，k の値が大きい方が時間計算量が小さくなる可能性が高い．よって，探索の時間計算量は k の値が大きいほど，$2k+1$ 分木の時間計算量は小さい．

5.3.2 挿入

挿入操作は探索操作の失敗後に行われる．挿入操作での時間計算量は $O(1)$ であるから，探索操作での時間計算量が問題となるが探索の時間計算量については上述の通りである．拡張した AVL 木では，挿入時点における平衡による回転操作がある．この回転操作の時間計算量は，高々定数個と考えられるので $O(1)$ である．よって，挿入による時間計算量は探索の時間計算量となる．拡張した AVL 木の回転操作は，移動する桁値の同値同士の枝で回転操作を行えば構造的に矛盾が起こることはない．

5.3.3 削除

削除操作は探索操作の成功後に行われる．探索についての時間計算量は上述の通りである．5 分木の削除方法を基本として考えると，リンク先の節点の付け替えは $O(1)$ である．または，葉に近い節点を削除するならば，高々定数個と考えられるので $O(1)$ である．よって，削除による時間計算量も探索の時間計算量となる．削除の場合に問題となるの

は，どのリンク先の節点を移動すればよいかであるが，基本的に k が小さいほど削除操作が単純である．図 5.3 は，3 分木に拡張した AVL 木の削除の図であり，各節点には比較する桁からの移動した桁数を記入している．図の節点 A は，データなので中央部分木が存在しない．もし，中央部分木が存在する節点 A を削除する場合は，グループの削除であり，この場合には節点 A 以下の部分木を削除すれば，節点 A を満たす文字列を削除できる．図の（a）のデータの節点 A を削除する場合は，右部分木の最も値が小さい節点 B を節点 A の位置にもってくる（または，節点 A の左部分木の最も大きい節点を節点 A の位置にもってくる）．さらに節点 C を節点 B の位置にもってくる．（ 2.3.2 節の 2 分探索木の削除を参照）節点 B を節点 A の位置にもってきても，移動する桁が同じであり，構造に矛盾は起きない．また，節点 B の中央部分木である節点 D をそのまま節点 B の中央部分木とすれば，構造に矛盾は起きない．もし，上述に当てはまる節点が存在しないならば葉に近い節点を削除することあり，節点 A 以下のデータを節点 A から再投入して部分木を再構築する．図（a）の節点 A を削除したとき，図（b）となる．

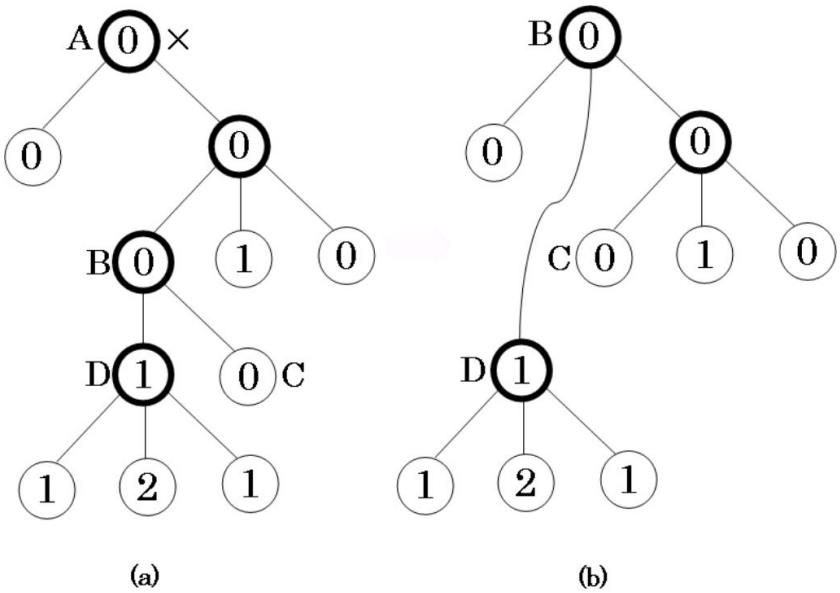


図 5.3　拡張した 3 分木の AVL 木の削除

図 5.4 は，7 分木に拡張した AVL 木の削除の例を示している．図 5.4 の各節点には比較する桁からの移動した桁数を記入している．L, R, LL, RR, C は，それぞれ左，右，前，後，中央部分木を表しており，LLL, RRR はもう一つ桁が移動した前部分木と後部分木

である．節点 A を削除する場合に，木の再構築に関して次の 4 通りが考えられる．

- 節点 A の部分木 LLL の最も右にある部分木 R 節点の部分木 LL 節点を節点 A の位置にもってくる．
- 節点 A の部分木 LLL の最も右にある部分木 R 節点の部分木 RR 節点を節点 A の位置にもってくる．
- 節点 A の部分木 RRR の最も左にある部分木 L 節点の部分木 RR 節点を節点 A の位置にもってくる．
- 節点 A の部分木 RRR の最も左にある部分木 L 節点の部分木 LL 節点を節点 A の位置にもってくる．

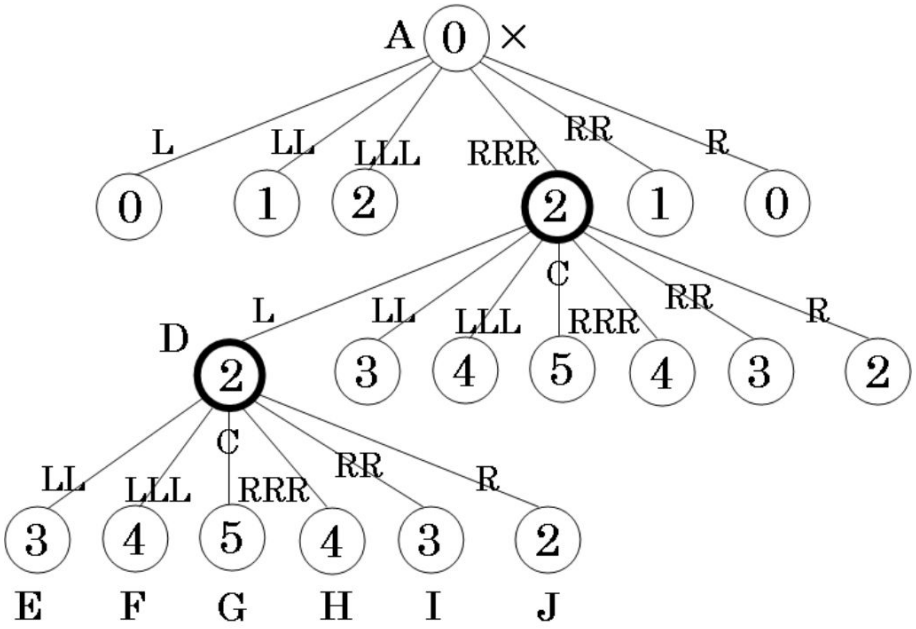


図 5.4　拡張した 7 分木の AVL 木の削除（前）

図 5.5 は，削除後の図である．今回は，節点 A の部分木 RRR の一番小さな部分木 L 節点 D の部分木 RR 節点 I を節点 A の位置にもってきている．節点 A の位置には節点 I のラベル付き節点 I' を作成して，節点 D は節点 I の最も左（L）に繋げる．節点 D, E, F, G, H には，移動する桁数を記入していないが，節点 I と D の関係より，構造の矛盾は起こらない．もし，上述に当てはまる節点が存在しないならば葉に近い節点を削除することあり，節点 A 以下のデータを節点 A から再投入して部分木を再構築する．

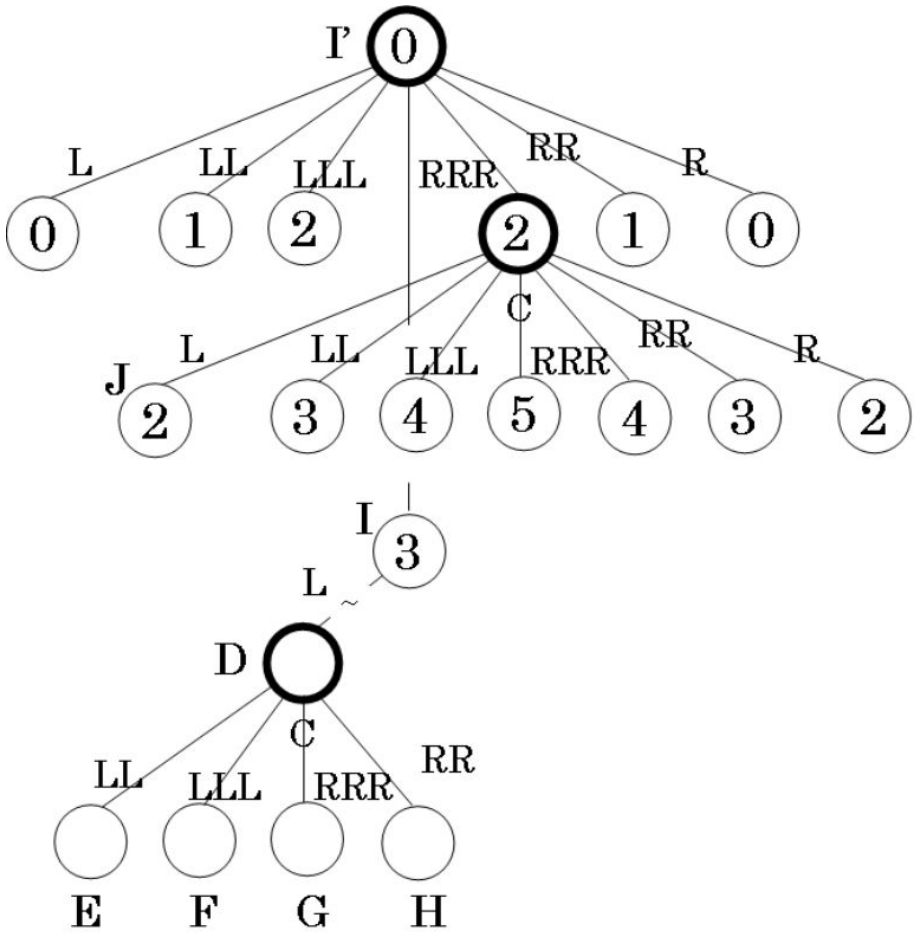


図 5.5　拡張した 7 分木の AVL 木の削除（後）

以上より探索，挿入，削除だけの時間計算量を中心に述べたが，どの値の k が一番効率が良いかなどの $2k+1$ 分木の一般化についての考察は，これからの研究課題である．

5.4　まとめ

この章では，第 4 章の数値実験の結果をもとにデータ構造の改善案を提案した．時間計算量の効率性は第 3・4 章から実証できたが，領域量の大きさが問題となった．これを改善するために，データ構造において「共通接頭辞となる文字列の削減」と「節点におけるポインタの削減」を行い，領域量を小さくすることを提案した．なお，提案した「共通接頭辞となる文字列の削減」と「節点におけるポインタの削減」のプログラム化は検討中である．

拡張した AVL 木の基数探索法では，特別な変数に比較している桁の添え字を記憶させなくとも，部分木でどの添え字を比較しているかを判断できている．これより，変数を用意するといった領域量の増加なしに共通接頭辞の削減が可能である．データ数を n，文字の種類を m，文字の長さを S として，$n = m^S$ でデータ数の最大値を設定して，文字の種類で分ける組み合わせ数について調べた．すべてのデータによる組み合わせに対するこの文字の種類で分ける組み合わせ数の割合は，文字列の長さと文字の種類である進数が大きくなるほど小さくなった．これより，実際の環境下では，特にひらがなや漢字のような文字の種類が多い場合や単語的に長い文章の探索には，この削減方法が有効になると考える．

節点におけるポインタの削減は，中央部分木へのポインタに注目した．もともと，この部分木へのポインタは，この節点がラベル付き節点に変化する場合に必要であるから，ラベル付き節点に変わるときの操作と同時に行えば，特別な時間計算量も掛らないことになる．つまり，2 つのデータ構造を準備して使い分けている．

これら，2 つの削減方法を使って，従来の方法と今回の改善の方法を比較した．「共通接頭辞となる文字列の削減」と同様に，すべてのデータによる組み合わせに対するこの文字の種類で分ける組み合わせ数の割合は，文字列の長さと文字の種類である進数が大きくなるほど小さくなった．これから，実際の環境下では，特にひらがなや漢字のような文字の種類が多い場合や単語的に長い文章の探索には，この削減方法が有効になると考える．既存の AVL 木の 10 進 50 桁の領域量に対する拡張した AVL 木の領域量は，1.04 倍と改善している．今回の算定では，データ数を 10^{50} 個で求めている．

この章では，$2k + 1$ 分木の一般化についても考察した．これは奇数分木を構築すれば，5 分木でなくとも拡張した AVL 木を構築できるものである．しかし，どの値の k が効率的であるか，これからの研究課題である．

第 6 章

まとめと今後の課題

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．拡張した AVL 木のデータ構造は，平衡性を部分的に満たす 5 分木構造であり，データ探索において語の文字列を 1 文字ずつ比較し，前方が一致する文字列を部分木とする特徴を持っている．数値実験により，拡張した AVL 木の構築時間は既存の AVL 木の約 70%，B 木の約 50% であり，探索の効率性が向上されることが確認できた．更にデータ構造の領域量の削減と奇数分木への一般化について考察した．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，領域の削減のプログラム化と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で実験することである．さらに，実用性の高い他のデータ構造（例えば，[55, 16]）との比較などである

第 1 章「研究の背景とアルゴリズムの評価法」では，文字列探索に関する研究の背景について述べ，アルゴリズムと計算量についての概念，アルゴリズムの様々な評価法を説明した．

第 2 章「基本的なデータ構造と木構造」では，計算機上でのデータ構造と木構造について説明し，それらの詳細な構造や関連するアルゴリズムは主に C 言語を使用して記述した．基本的なデータ構造として，リスト・配列・スタック・キューを取り上げ，各データ構造の特徴と長所・短所について述べた．特に動的または静的な辞書を作成するには，リ

ストと配列構造は重要な構造となる．さらに木構造の性質や定義について説明し，これを応用した 2 分探索木，AVL 木，B 木，トライ，パトリシアを C 言語を利用して詳述した．2 分探索木は，木構造で最も基本的な構造であり，AVL 木は平衡木の一種であり 2 分探索木に改良を加えて木の高さを最悪でも $O(\log n)$ としたものである．この木は，左右の部分木の高さがほぼバランスが取れており平衡木とよばれる．AVL 木のアルゴリズムは，この分野の基本であり，他分野にも応用されている．トライとパトリシアは，文字列の探索に適した実用性の高い木構造であり，基数探索法を利用しており，他分野にも応用されている実績がある．トライとパトリシアは平衡木ではないが，複雑な処理な後処理をしなくとも性能が期待されると言われ，これを応用した木構造を紹介した．B 木は，根と葉以外の各節点に複数のデータを格納できる多分木であり，その性質から外部記憶装置によく使われる．B 木は，根から全ての葉までの高さが同じことから平衡木とよばれている．

第 3 章「5 分木に拡張した AVL 木の提案」では，第 2 章までのデータ構造とアルゴリズムの長所と短所を踏まえ，既存の AVL 木を文字列探索に適応させたデータ構造を提案した．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 文字を比較する．拡張した AVL 木の平衡は，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．更に前と後部分木についても，なるべく平衡となるように，文字が m 進数であれば m の半分とする数値の文字を部分木の根に挿入し平衡を促進している．この拡張した AVL 木に対し，定義，各操作，時間計算量などについて考察した．基数探索法の利点として，前方が一致する文字列を探索または削除するときは該当する部分木を操作すればよいという点が挙げられる．前方一致文字列が同じ部分木となることで，前方一致に関する文字列の探索は高速に行われ，前方一致文字列の削除は一瞬で操作できる．前方一致文字列の削除を行ったとしても，前・後・中央部分木を削除することになり，平衡する必要はない．データ数 n において，データ探索のための時間計算量は $O(\log n)$ であり，更に 2 分木から多分木に拡張したことで目的のデータに早く辿り着くことが可能である．前方一致文字列探索の比較回数は，根か

ら部分木までの節点数の高々 2 倍であり，拡張した AVL 木の長所の一つと言える．木構造を維持するためにラベル付きの節点を導入した．ラベル付き節点は，中央部分木に進んだときに中央部分木の親節点のみがデータ節点からラベル付き節点へと変化する．ラベル付き節点は，データではなく増加すると探索時間や領域量が増加すると考えられるが，ラベルの割合は $m(\geq 10)$ 進数において，1% 以下のほぼ一定となることがわかった．数値実験では 10 進数で比較したが，実生活では 10 進数以上の文字を使用しており，この場合には 1% を大きく下回ることを確認しており，実際の環境では，ほとんど影響がないことが分かった．

第 4 章「拡張した AVL 木の評価」では，拡張した AVL 木，既存の各木を C 言語で作成し，重要な項目について数値比較を行った．比較には，2 分探索木，AVL 木，5 分木の B 木を対象とした．2 分探索木，AVL 木との比較項目は，(1) 構築時間，(2) 領域量，(3) 深さ，(4) 比較回数，(5) 平衡回数，(6) ラベル数である．拡張した AVL 木では既存の AVL 木より比較回数を約 23% に抑えられることから効率的にデータ探索が可能なこと，AVL 木の 55%〜70% の時間でデータ構造を構築できる等の結果が得られた．5 分木に拡張した AVL 木は，2 分木から多分木へと拡張したことにより木の深さ (3) が浅くなることから，(1)(4)(5) の値が小さくなり，その結果として探索時間が短くなる長所をもつことが確認できた．5 分木に拡張した AVL 木の短所は (2) 領域量であり，文字が 10 進数で文字列の長さが 100 桁の場合では，2 分木の AVL 木の約 1.2 倍である．5 分木の B 木に対しては，時間量・領域量の面から両者の計算量の比較を行った．5 分木の B 木との比較では，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} 個以上であり，5 分木に拡張した AVL 木は 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% である．

第 5 章「5 分木に拡張した AVL 木の領域量の削減と一般化」では，第 4 章の数値実験で明らかになった領域量の問題と奇数分木への AVL 木への拡張について考察している．本論文で提案した AVL 木の短所としては領域量の増加であったが，これを削減する方法

を提案している．領域量の削減には，ポインタの数を減らすことと節点のデータになる文字列の削減で対応した．ポインタ数の削減については，節点がラベルになる場合，データ用の配列は構造を維持する 2 文字だけ必要であり，データが格納されている節点では中央部分木へのポインタが不要である．一方，文字列の削減については節点に文字列全体ではなく，一致しない後方の文字列のみを保持させることで削減を行った．これにより，従来の 5 分木構造の AVL 木より 10 進数 50 桁の場合で，67.74% の削減が可能である．これによって，既存の AVL 木の 10 進 50 桁の領域量に対する拡張した AVL 木の領域量は，1.04 倍と改善している．なお，4.3 節の数値実験ではデータ数は 10,000,000 個であったが，今回の試算ではデータ数を 10^{50} 個で求めている．また，ある節点において最大 k 桁の比較を行うと，$2k+1$ 分木として一般化できることを示した．

謝辞

本論文は鹿児島大学大学院理工学研究科システム情報工学専攻において筆者が行った木構造の探索効率に関する研究の成果をまとめ上げたものである．著者が本研究分野に興味を抱いた時から今日に至るまで，終始懇切なるご指導，御助言を賜りました本学大学院理工学研究科数理情報科学専攻 教授 新森修一先生に深く感謝の意を表すと共に，厚く御礼申し上げます．

本学大学院理工学研究科数理情報科学専攻 准教授 青木敏先生，准教授 古澤仁先生には，本論文作成にあたり細部にわたりご指導頂き，貴重な御助言を頂きましたこと，心より御礼申し上げます．

最後に学友であります本学大学院理工学研究科 津曲紀宏氏，本学大学院理工学研究科卒 Sarpono Dimulyo 氏には，とても楽しく研究ができる環境を提供していただいたことに感謝致します．

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] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers-Principles, Techniques, and Tools*. Addison-Wesley, Reading Mass., 1986.

[3] M. Al-Suwaiyel and E Horowitz. Algorithms for trie compaction. *ACM Trans. Database Syst.*, Vol. 9, pp. 243–263, June 1984.

[4] Jun-ichi Aoe. An efficient digital search algorithm by using a double-array structure. *IEEE Trans. Softw. Eng.*, Vol. 15, pp. 1066–1077, September 1989.

[5] 青江順一. トライとその応用 (連載講座 キー検索技法 4). 情報処理, Vol. 34, No. 2, pp. 244–251, 1993-02-15.

[6] 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. p1592–1600, 1988-09.

[7] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, Vol. 1, pp. 290–306, 1972.

[8] 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.

[9] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings, 1997.

[10] Luc Bouge, Joaquim Gabarro, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed avl rebalancing: A unified, fine-grained approach to concurrent dictionaries, 1998.

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

[12] Rene De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, IRE-AIEE-ACM 1959 (Western), pp. 295–298, New York, NY, USA, 1959. ACM.

[13] Willian B. Frakes and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

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

[15] 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, No. 1, pp. 1–9, 1993-01-15.

[16] 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.

[17] C. A. R. Hoare. Algorithm 64: Quicksort. *COMMUNICATIONS OF THE ACM*, July 1, 1961.

[18] 田中穂積. 自然言語解析の基礎. 産業図書, 1989.

[19] 茨城俊秀. アルゴリズムとデータ構造. 昭晃堂, 1989.

[20] 茨城俊秀. C によるアルゴリズムとデータ構造. 昭晃堂, 1999.

[21] Jun ichi Aoe, Katsushi Morimoto, and Takashi Sato. An efficient implementation of trie structures, 1992.

[22] Robert W. Irving and Lorna Love. The suffix binary search tree and suffix AVL tree. *J. of Discrete Algorithms*, Vol. 1, pp. 387–408, October 2003.

[23] Lars Jacobsen and Kim S. Larsen. Complexity of layered binary search trees with relaxed balance. In *Proceedings of the 7th Italian Conference on Theoretical Computer Science*, ICTCS '01, pp. 269–284, London, UK, 2001. Springer-Verlag.

[24] 青江順一. 自然言語辞書の検索-ダブル配列による高速検索アルゴリズム. bit（共立出版）, 1990.

[25] Kim S. Larsen. AVL Trees with Relaxed Balance. *Jornal of Computer and System Sciences*, Vol. 61, No. 3, pp. 508–522, 2000.

[26] Kim S. Larsen, Thomas Ottmann, and Eljas Soisalon-Soininen. Relaxed Balance for Search Trees with Local Rebalancing. In *Proceedings of the 5th Annual European Symposium on Algorithms*, pp. 350–363, London, UK, 1997. Springer-Verlag.

[27] Kim S. Larsen, Eljas Soisalon-Soininen, and Peter Widmayer. *AVL Trees with Relaxed Balance*, pp. 501–512. Algorithmica, 2001.

[28] M. E. Lesk and E. Schmidt. *Lex a lexical analyzer generator*, pp. 375–387. W. B. Saunders Company, Philadelphia, PA, USA, 1990.

[29] W.A. Litwin, N. Roussopoulos, G. Levy, and W. Hong. Trie hashing with controlled load. *IEEE Transactions on Software Engineering*, Vol. 17, pp. 678–691, 1991.

[30] Kurt Maly. Compressed tries. *Commun. ACM*, Vol. 19, pp. 409–415, July 1976.

[31] 星守, 弓場敏嗣. 2 分探索木, B-木, k-d 木による見出し探索. 情報処理, Vol. 24, No. 4, pp. 396–400, 1983-04-15.

[32] 星守. データ構造. 昭晃堂, 2002.

[33] 竹原幹人, 大島裕明, 田中克己. Blog における書き手の興味を考慮した意見情報の提示手法 (blog, 夏のデータベースワークショップ dbws2005). 情報処理学会研究報告. データベース・システム研究会報告, Vol. 2005, No. 67, pp. 39–45, 2005-07-13.

[34] Donald R. Morrison. Patricia practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, Vol. 15, pp. 514–534, October 1968.

[35] 大沢貴之, 神谷陽一, 宮口庄司. DNS 向け 10 進 N 桁 Patricia Tree 構築法の検討. 電子情報通信学会技術研究報告. NS, ネットワークシステム, Vol. 101, No. 715, pp. 327–334, 2002-03-08.

[36] Th. Ottmann and E. Soisalon-Soininen. Relaxed balancing made simple, 1995.

[37] James L. Peterson. Computer programs for detecting and correcting spelling errors. *Commun. ACM*, Vol. 23, pp. 676–687, December 1980.

[38] James Lyle Peterson. *Computer Programs for Spelling Correction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1980.

[39] Ohad Rodeh, Kenneth P. Birman, and Danny Dolev. Using AVL trees for fault

tolerant group key management. *International Journal on Information Security*, Vol. 1, pp. 84–99, 2000.

[40] Donald E. Knuth Ronald L. Graham and Oren Patashnik. *CONCRETE MATHEMATICS*. 共立出版, 1993. 有澤誠, 安村通晃, 萩野達也, 石畑清訳.

[41] NAKAMURA Ryozo, TADA Akio, and ITOKAWA Tsuyoshi. An analysis of the avl balanced tree insertion algorithm. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, Vol. 86, No. 5, pp. 1067–1074, 2003-05-01.

[42] Wojciech Rytter and Instytut Informatyki Uniwersytet Warszawski. Application of lempel-ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, Vol. 302, p. 2003, 2002.

[43] 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.

[44] 清水道夫, 中村義作. ある種の avl 木の総数に対する多項式表現. 電子通信学会論文誌 A, Vol. 68, No. 10, pp. p1128–1129, 1985-10.

[45] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *J. ACM*, Vol. 32, pp. 652–686, July 1985.

[46] Daniel F. Stubbs and Neil W. Webre. *Data Structures with Abstract Data Types and Pascal*. オーム社, 1994. 小山裕徳訳.

[47] 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.

[48] 小澤孝夫. コンピュータ・アルゴリズム. 昭晃堂, 1990.

[49] 横森貴. アルゴリズム データ構造 計算論. サイエンス社, 2005.

[50] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Commun. ACM*, Vol. 22, pp. 606–611, November 1979.

[51] 渡邉敏正. データ構造と基本アルゴリズム. 共立出版, 2000.

[52] V.K. Vaishnavi. On the height of multidimensional height-balanced trees. *IEEE Transactions on Computers*, Vol. 35, pp. 773–780, 1986.

[53] Niklaus Wirth. *Algorithms and Data Structures.* 近代科学社, 1990. 浦昭二, 國府方久史訳.

[54] David Wood. Kowari: A platform for semantic web storage and analysis. In *In XTech 2005 Conference*, pp. 05–0402, 2005.

[55] 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.

[56] 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.

付録 A

2 分探索木のソースコード

84 行目の random_array 関数によって，乱数を使ってデータ数を N 個，文字列を S 桁生成して，2 分探索木を構築する．その後に，main 関数のメニューによって「挿入」,「探索」,「削除」,「印字」を行うプログラムである．
「挿入」は，S 桁の文字列を入力後，insert 関数で挿入位置を決定して make_leaf 関数で葉を作る．
「探索」は，S 桁の文字列を入力後，search 関数で探索結果を返す．
「削除」は，S 桁の文字列を入力後，delete 関数で文字列を削除して delete_case 関数で削除方法を決定する．
「印字」は，print_binary 関数で中順で印字する．

```
1  /* ****************************************
2   *                                        *
3   *              binary_search_tree        *
4   *           procedure Create, Search,    *
5   *               Delete & Print           *
6   *                                        *
7   ****************************************/
8
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <stdlib.h>
13 #define N xxx                    // データ数
14 #define S xxx                    // 文字列の長さ
15 #define D xxx                    // garbage用の文字列
16
17 enum answer {YES, NO};           // 答え
18
19 // binay_treeのデータ構造
20 struct binary {
21   char element[S+1];             // データをS桁格納
22   struct binary *left;           // 左部分木を指すポインタ
23   struct binary *right;          // 右部分木を指すポインタ
24 };
25
26 struct binary *header = NULL;    // 根を指すポインタ
27
28 void random_array(void);
29 int menu(void);
30 void Insert(void);
```

```
31  void Search(void);
32  void Delete(void);
33  void Print(void);
34  enum answer insert(char *);
35  struct binary *make_leaf(char *);
36  enum answer search(char *);
37  enum answer delete(char *);
38  struct binary *delete_case(struct binary *);
39  void print_binary(struct binary *);
40
41  /* ************************************************** *
42   *                    main関数                        *
43   * 木 Tを構築して、その後にメニューに従って、各操作を行う。 *
44   * ************************************************** */
45  int main(void)
46  {
47    int bt;                                    // メニュー
48
49    random_array();
50
51    while (1) {
52      bt = menu();
53      switch (bt) {
54      case 'I':                                // 挿入
55        Insert();
56        putchar('\n');
57        break;
58      case 'F':                                // 探索
59        Search();
60        putchar('\n');
61        break;
62      case 'D':                                // 削除
63        Delete();
64        putchar('\n');
65        break;
66      case 'P':                                // 印字
67        Print();
68        putchar('\n');
69        break;
70      case 'E':                                // 終了
71        exit(0);
72        break;
73      default:
74        puts("input, again!\n");
75      }
76    }
77
78    return 0;
79  }
80
81  /* ************************************************ *
82   * データを N個発生させて、Tを 2分木とする。          *
83   * ************************************************ */
84  void random_array(void)
85  {
86    int n, s;                                  // カウント
87    char str[S+1];                             // 文字列 S桁
88
89    srand(time(NULL));
```

```
90
91    for (n = 0; n < N; n++) {
92      for (s = 0; s < S; s++)
93        str[s] = (rand() % 10) + '0';
94      str[s] = '\0';
95
96      insert(str);
97    }
98
99    return;
100 }
101
102 /* ************************************************ *
103  *                  メニューを表示する。                *
104  * 戻り値は各操作のキーワードである。                   *
105  * ************************************************ */
106 int menu(void)
107 {
108   int ch;                          // 各操作のキーワード
109   char gar[D];                     // garbage
110
111   puts("MENU\n\tI...insert\n\tF...search\n\tD...delete\n\tP...print\n\tE
          ...end");
112   ch = getchar();
113   gets(gar);
114
115   return ch;
116 }
117
118 /* ************************************************ *
119  *    挿入したい文字列を入力(葉が出来たら考察をする)    *
120  * ************************************************ */
121 void Insert(void)
122 {
123   int c;                           // 一文字
124   int s;                           // 添字
125   char str[D];                     // 文字列D桁
126   enum answer a;                   // 挿入 or not
127   int len;                         // 文字列長
128
129   do {
130     printf("input␣number(%dwords).\n", S);
131     gets(str);
132     len = strlen(str);
133   } while (len != S);
134
135   a = insert(str);
136
137   if (a == NO)
138     puts("ALREADY␣EXIST.");
139
140   return ;
141 }
142
143 /* ************************************************ *
144  *                  木Tを探索する。                   *
145  * ************************************************ */
146 void Search(void)
147 {
148   int len;                              // 文字列長
```

```
149     char str[D];                            // 文字列
150     enum answer a;                          // 探索 or not
151
152     do {
153       printf("input number(%dwords).\n", S);
154       gets(str);
155       len = strlen(str);
156     } while (len != S);
157
158     a = search(str);
159
160     if (a == YES)
161       puts("FIND.");
162     else
163       puts("NOT EXIST.");
164
165     return;
166   }
167
168   /* ************************************************ *
169    *              入力する文字列を削除する。          *
170    * ************************************************ */
171   void Delete(void)
172   {
173     char str[D];                            // 文字列D桁
174     int len;                                // 文字列長
175     enum answer a;                          // 削除 or not
176
177     do {
178       printf("input number(%dwords).\n", S);
179       gets(str);
180       len = strlen(str);
181     } while (len != S);
182
183     a = delete(str);
184
185     if (a == NO)
186       puts("NOT EXIST.");
187
188     return;
189   }
190
191   /* ************************************************ *
192    *            木 Tをなぞりを利用して印字する。      *
193    * ************************************************ */
194   void Print(void)
195   {
196     extern struct binary *header;
197
198     print_binary(header);
199
200     return;
201   }
202
203   /* ************************************************ *
204    *      データ strを木 Tに挿入する。(挿入のアルゴリズム)  *
205    * 引数は、文字列 strである。                       *
206    * 戻り値は、挿入できたら'YES'を、挿入できなかったら'NO'  *
207    * を返す。                                         *
208    * ************************************************ */
```

```
209 enum answer insert(char *str)
210 {
211   int s = 0;                              // 添字
212   struct binary **q;                      // 走査ポインタ
213   extern struct binary *header;
214
215   q = &(header);
216
217   while (*q != NULL) {
218
219     if (str[s] < (*q) -> element[s]) {
220       q = &((*q) -> left);
221       s = 0;
222     }
223     else if (str[s] > (*q) -> element[s]) {
224       q = &((*q) -> right);
225       s = 0;
226     }
227     else {
228       if (s == S - 1)
229         return NO;
230       else
231         s++;
232     }
233   }
234
235   *q = make_leaf(str);
236
237   return YES;
238 }
239
240 /* ************************************************** *
241  *                    葉を生成する。                  *
242  * 引数はデータ str である。                          *
243  * 戻り値は、葉を指すポインタ leaf である。           *
244  * ************************************************** */
245 struct binary *make_leaf(char *str)
246 {
247   struct binary *leaf;                // 葉を指すポインタ
248
249   if ((leaf = malloc(sizeof(struct binary))) == NULL) {
250     printf("out of memory!!\n");
251     exit(1);
252   }
253
254   strcpy(leaf -> element, str);
255   leaf -> left = NULL;
256   leaf -> right = NULL;
257
258   return leaf;
259 }
260
261 /* ************************************************ *
262  * 「データ」を探索する。(探索のアルゴリズム)       *
263  * 引数は、文字列 str である。                      *
264  * 戻り値は、データが存在すれば 'YES' を存在しない場合 *
265  * は 'NO' を返す。                                 *
266  * ************************************************ */
267 enum answer search(char *str)
268 {
```

```
269     int s = 0;                              // 添字
270     struct binary *p;                       // 走査ポインタ
271     extern struct binary *header;
272
273     p = header;
274
275     while (p != NULL) {
276       if (str[s] < p -> element[s]) {
277         p = p -> left;
278         s = 0;
279       }
280       else if (str[s] > p -> element[s]) {
281         p = p -> right;
282         s = 0;
283       }
284       else {
285         if (s == S - 1) {
286           return YES;
287         }
288         else
289           s++;
290       }
291     }
292
293     return NO;
294   }
295
296   /* ************************************************** *
297    * 「データ」を削除する。(削除のアルゴリズム)          *
298    * 引数は探索データである str である。                 *
299    * 戻り値は、削除出来れば 'YES' を削除出来なかったら 'NO' を  *
300    * 返す。                                             *
301    * ************************************************** */
302   enum answer delete(char *str)
303   {
304     int s = 0;                              // 配列の添字
305     struct binary *p;                       // 削除節点を指す
306     struct binary **q;                      // 走査ポインタ
307     extern struct binary *header;
308
309     q = &(header);
310     p = *q;
311
312     while (*q != NULL) {
313       p = *q;
314       if (str[s] < (*q) -> element[s]) {
315         q = &((*q) -> left);
316         s = 0;
317       }
318       else if (str[s] > (*q) -> element[s]) {
319         q = &((*q) -> right);
320         s = 0;
321       }
322       else {
323         if (s == S - 1) {
324           *q = delete_case(p);
325           return YES;
326         }
327         else
328           s++;
```

```
329       }
330     }
331
332     return NO;
333   }
334
335   /* ************************************************** *
336    * 削除する節点の子で削除の仕方を変える。(場合分けのアル  *
337    * ゴリズム)                                          *
338    * 引数は削除する節点を指すポインタrである。          *
339    * 戻り値は削除する節点を指すポインタrを返す。削除する節  *
340    * 点の子が共になかったらNULLを返す。                 *
341    * ************************************************** */
342   struct binary *delete_case(struct binary *r)
343   {
344     struct binary *p;          // 削除節点を指す
345     struct binary **q;         // 走査節点
346
347     // 削除する節点の子が共にNULLである。
348     if ((r -> left == NULL) && (r -> right == NULL)) {
349       free(r);
350       return NULL;
351     }
352
353     // 削除する節点の左の子がNULLである。
354     else if (r -> left == NULL) {
355       q = &(r -> right);
356       p = r -> right;
357       strcpy(r -> element, p -> element);
358       *q = delete_case(p);
359       return r;
360     }
361
362     // 削除する節点の右の子がNULLである。
363     else if (r -> right == NULL) {
364       q = &(r -> left);
365       p = r -> left;
366       strcpy(r -> element, p -> element);
367       *q = delete_case(p);
368       return r;
369     }
370
371     // 削除する節点の子が共に存在する。
372     else {
373       q = &(r -> right);
374       p = r -> right;
375       while (p -> left != NULL) {
376         q = &(p -> left);
377         p = p -> left;
378       }
379       strcpy(r -> element, p -> element);
380       *q = delete_case(p);
381       return r;
382     }
383   }
384
385   /* ************************************************ *
386    *      木を左部分木、節点、右部分木の順に印字する。    *
387    * 引数は、部分木の根を指すポインタpである。          *
388    * ************************************************ */
```

```
389  void print_binary(struct binary *p)
390  {
391    if (p != NULL) {
392      print_binary(p -> left);
393      printf("%s\n", p -> element);
394      print_binary(p -> right);
395    }
396
397    return;
398  }
```

付録 B

AVL 木のソースコード

96 行目の random_array 関数によって，乱数を使ってデータ数を N 個，文字列を S 桁生成して，AVL 木を構築する．その後に，main 関数のメニューによって「挿入」,「探索」,「削除」,「印字」を行うプログラムである．
「挿入」は，S 桁の文字列を入力後，insert 関数で挿入位置を決定して make_leaf 関数で葉を作る．その後，think_insert 関数で回転操作の有無を考察する．
「探索」は，S 桁の文字列を入力後，search 関数で探索結果を返す．
「削除」は，S 桁の文字列を入力後，delete 関数で文字列を削除して think_delete 関数で回転操作の有無を考察する．
「印字」は，print_avl 関数で中順で印字する．

```
1  /* ****************************************
2   *                                        *
3   *                  AVL_tree              *
4   *          procedure Create, Search,     *
5   *              Delete & Print            *
6   *                                        *
7   ****************************************/
8
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <stdlib.h>
13 #define N xxx                          // データ数
14 #define S xxx                          // 文字列の長さ
15 #define D xxx                          // garbage用の文字列
16
17 enum sub {RO, LT, RT};                 // どこの部分木か
18 enum diff {EVEN, LEFT, RIGHT};         // どちらの部分木が高いか
19 enum answer {YES, NO};                 // 答え
20
21 // AVL_treeのデータ構造
22 struct avl {
23   char element[S+1];                   // データをS桁格納
24   enum sub sub_tree;                   // enum sub 参照
25   enum diff diff;                      // enum diff 参照
26   struct avl *left;                    // 左部分木を指すポインタ
27   struct avl *right;                   // 右部分木を指すポインタ
28   struct avl *parent;                  // 親を指すポインタ
29 };
30
```

```
31  struct avl *header = NULL;          // 根を指すポインタ
32  struct avl *parent_node;            // 親を指すポインタ
33  enum sub current_node;              // その節はどの部分木か
34
35  void random_array(void);
36  int menu(void);
37  void Insert(void);
38  void Search(void);
39  void Delete(void);
40  void Print(void);
41  enum answer insert(char *);
42  struct avl *make_leaf(char *);
43  void think_insert(enum sub, struct avl *);
44  void think_delete(enum sub, struct avl *);
45  struct avl *rebuilt(struct avl *, struct avl *);
46  struct avl *single_rotation(struct avl *, struct avl *);
47  struct avl *double_rotation(struct avl *, struct avl *, struct avl *);
48  enum answer search(char *);
49  enum answer delete(char *);
50  struct avl *delete_case(struct avl *);
51  void print_avl(struct avl *);
52
53  /* ************************************************** *
54   *                    main関数                        *
55   * 木 Tを構築して、その後にメニューに従って、各操作を行う。 *
56   * ************************************************** */
57  int main(void)
58  {
59    int bt;                                   // メニュー
60
61    random_array();
62
63    while (1) {
64      bt = menu();
65      switch (bt) {
66      case 'I':                               // 挿入
67        Insert();
68        putchar('\n');
69        break;
70      case 'F':                               // 探索
71        Search();
72        putchar('\n');
73        break;
74      case 'D':                               // 削除
75        Delete();
76        putchar('\n');
77        break;
78      case 'P':                               // 印字
79        Print();
80        putchar('\n');
81        break;
82      case 'E':                               // 終了
83        exit(0);
84        break;
85      default:
86        puts("input, again!\n");
87      }
88    }
89
```

```
90    return 0;
91  }
92
93  /* ************************************************ *
94   * データを N個発生させて、木 Tを AVL木で構築する。          *
95   * ************************************************ */
96  void random_array(void)
97  {
98    int n, s;                                    // カウント
99    char str[S+1];                               // 文字列 S桁
100   enum answer a;                               // 挿入 or not
101
102   srand(time(NULL));
103
104   for (n = 0; n < N; n++) {
105     for (s = 0; s < S; s++)
106       str[s] = (rand() % 10) + '0';
107     str[s] = '\0';
108
109     a = insert(str);
110
111     if (a == YES)
112       think_insert(current_node, parent_node);
113   }
114
115   return;
116 }
117
118 /* ************************************************ *
119  *                  メニューを表示する。                *
120  * 戻り値は各操作のキーワードである。                    *
121  * ************************************************ */
122 int menu(void)
123 {
124   int ch;                              // 各操作のキーワード
125   char gar[D];                         // garbage
126
127   puts("MENU\n\tI...insert\n\tF...search\n\tD...delete\n\tP...print\n\tE
        ...end");
128   ch = getchar();
129   gets(gar);
130
131   return ch;
132 }
133
134 /* ************************************************ *
135  *    挿入したい文字列を入力(葉が出来たら考察をする)     *
136  * ************************************************ */
137 void Insert(void)
138 {
139   int c;                               // 一文字
140   int s;                               // 添字
141   char str[D];                         // 文字列 D桁
142   enum answer a;                       // 挿入 or not
143   int len;                             // 文字列長
144
145   do {
146     printf("input number(%dwords).\n", S);
147     gets(str);
```

```
148       len = strlen(str);
149     } while (len != S);
150
151     a = insert(str);
152
153     if (a == YES)
154       think_insert(current_node, parent_node);
155     else
156       puts("ALREADY EXIST.");
157
158     return ;
159   }
160
161   /* ************************************************ *
162    *                  木 Tを探索する。                *
163    * ************************************************ */
164   void Search(void)
165   {
166     int len;                              // 文字列長
167     char str[D];                          // 文字列
168     enum answer a;                        // 探索 or not
169
170     do {
171       printf("input number(%dwords).\n", S);
172       gets(str);
173       len = strlen(str);
174     } while (len != S);
175
176     a = search(str);
177
178     if (a == YES)
179       puts("FIND.");
180     else
181       puts("NOT EXIST.");
182
183     return;
184   }
185
186   /* ************************************************ *
187    *              入力する文字列を削除する。          *
188    * ************************************************ */
189   void Delete(void)
190   {
191     char str[D];                          // 文字列D桁
192     int len;                              // 文字列長
193     enum answer a;                        // 削除 or not
194
195     do {
196       printf("input number(%dwords).\n", S);
197       gets(str);
198       len = strlen(str);
199     } while (len != S);
200
201     a = delete(str);
202
203     if (a == YES)
204       think_delete(current_node, parent_node);
205     else
206       puts("NOT EXIST.");
207
```

```
208     return;
209   }
210
211   /* ************************************************** *
212    *              木 Tをなぞりを利用して印字する。              *
213    * ************************************************** */
214   void Print(void)
215   {
216     extern struct avl *header;
217
218     print_avl(header);
219
220     return;
221   }
222
223   /* ************************************************** *
224    *       データ strを木 Tに挿入する。(挿入のアルゴリズム)     *
225    * 引数は、文字列 strである。                              *
226    * 戻り値は、挿入できたら'YES'を、挿入できなかったら'NO'  *
227    * を返す。                                              *
228    * ************************************************** */
229   enum answer insert(char *str)
230   {
231     int s = 0;                          // 添字
232     struct avl **q;                     // 走査ポインタ
233     extern enum sub current_node;
234     extern struct avl *header;
235     extern struct avl *parent_node;
236
237     current_node = RO;
238     q = &(header);
239     parent_node = *q;
240
241     while (*q != NULL) {
242       parent_node = *q;
243
244       if (str[s] < (*q) -> element[s]) {
245         current_node = LT;
246         q = &((*q) -> left);
247         s = 0;
248       }
249       else if (str[s] > (*q) -> element[s]) {
250         current_node = RT;
251         q = &((*q) -> right);
252         s = 0;
253       }
254       else {
255         if (s == S - 1)
256           return NO;
257         else
258           s++;
259       }
260     }
261
262     *q = make_leaf(str);
263
264     return YES;
265   }
266
267   /* ************************************************** *
```

```
268   *                    葉を生成する。                    *
269   * 引数はデータ str である。                            *
270   * 戻り値は、葉を指すポインタ leaf である。             *
271   * ************************************************** */
272  struct avl *make_leaf(char *str)
273  {
274    struct avl *leaf;              // 葉を指すポインタ
275    extern enum sub current_node;
276    extern struct avl *parent_node;
277
278
279    if ((leaf = malloc(sizeof(struct avl))) == NULL) {
280      printf("out of memory!!\n");
281      exit(1);
282    }
283
284    strcpy(leaf -> element, str);
285    leaf -> sub_tree = current_node;
286    leaf -> diff = EVEN;
287    leaf -> left = NULL;
288    leaf -> right = NULL;
289    leaf -> parent = parent_node;
290
291    return leaf;
292  }
293
294  /* ************************************************** *
295   * 挿入に関する高さの差を考察する。(考察のアルゴリズム  *
296   * (挿入))                                            *
297   * 引数は節点がどの部分木であるか?を示す sub と節点の親  *
298   * を指すポインタ u である。                          *
299   * ************************************************** */
300  void think_insert(enum sub sub, struct avl *u)
301  {
302    // 根まで到達したら終了とする
303    if (u == NULL)
304      return;
305    else
306      switch (sub) {
307        // 左部分木の高さが増えた
308      case LT:
309        if (u -> diff == RIGHT) {
310          u -> diff = EVEN;
311          return;
312        }
313        else if (u -> diff == LEFT)
314          rebuilt(u, u -> left);
315        else {
316          u -> diff = LEFT;
317          think_insert(u -> sub_tree, u -> parent);
318        }
319        break;
320
321        // 右部分木の高さが増えた
322      case RT:
323        if (u -> diff == LEFT) {
324          u -> diff = EVEN;
325          return;
326        }
327        else if (u -> diff == RIGHT)
```

```
328           rebuilt(u, u -> right);
329         else {
330           u -> diff = RIGHT;
331           think_insert(u -> sub_tree, u -> parent);
332         }
333         break;
334       }
335
336     return;
337   }
338
339   /* ********************************************** *
340    * 削除に関する高さの差を考察する。(考察のアルゴリズム  *
341    * (削))                                          *
342    * 引数は節点がどの部分木であるか?を示すsubと節点の親  *
343    * を指すポインタuである。                          *
344    * ********************************************** */
345   void think_delete(enum sub sub, struct avl *u)
346   {
347     struct avl *p;                    // 部分木の根を指す
348
349     // 根まで到達したら終了とする
350     if (u == NULL)
351       return;
352     else
353       switch (sub) {
354         // 左部分木の高さが減った
355       case LT:
356         if (u -> diff == EVEN) {
357           u -> diff = RIGHT;
358           return;
359         }
360         else if (u -> diff == RIGHT) {
361           p = rebuilt(u, u -> right);
362           if (p -> diff == EVEN)
363             think_delete(p -> sub_tree, p -> parent);
364         }
365         else {
366           u -> diff = EVEN;
367           think_delete(u -> sub_tree, u -> parent);
368         }
369         break;
370
371         // 右部分木の高さが減った
372       case RT:
373         if (u -> diff == EVEN) {
374           u -> diff = LEFT;
375           return;
376         }
377         else if (u -> diff == LEFT) {
378           p = rebuilt(u, u -> left);
379           if (p -> diff == EVEN)
380             think_delete(p -> sub_tree, p -> parent);
381         }
382         else {
383           u -> diff = EVEN;
384           think_delete(u -> sub_tree, u -> parent);
385         }
386         break;
387       }
```

```
388
389     return;
390   }
391
392   /* ************************************************ *
393    *  どの回転を行うかを選別する。(再構築のアルゴリズム)  *
394    * 引数は高さが2高い節点を指すuとその子を指すvである。  *
395    * 戻り値は、部分木の根を指すポインタpである。          *
396    * ************************************************ */
397   struct avl *rebuilt(struct avl *u, struct avl *v)
398   {
399     struct avl *p;                          // 部分木の根を指す
400
401     // 左部分木が2高くなった場合
402     if (u -> diff == LEFT) {
403       // 一重回転の場合
404       if ((v -> diff == EVEN) || (v -> diff == LEFT))
405         p = single_rotation(u, v);
406       // 二重回転の場合
407       else
408         p = double_rotation(u, v, v  -> right);
409     }
410
411     else {
412       // 右部分木が2高くなった場合
413       // 一重回転の場合
414       if ((v -> diff == EVEN) || (v -> diff == RIGHT))
415         p = single_rotation(u, v);
416       // 二重回転の場合
417       else
418         p = double_rotation(u, v, v -> left);
419     }
420
421     return p;
422   }
423
424   /* ************************************************** *
425    *                      一重回転                      *
426    * 引数は高さが2高い節点を指すuとその子を指すvである。  *
427    * 戻り値は、部分木の根を指すポインタpである。          *
428    * ************************************************** */
429   struct avl *single_rotation(struct avl *u, struct avl *v)
430   {
431     if (u -> diff == LEFT) {
432       // 子を指すポインタを設定する
433       u -> left = v -> right;
434       v -> right = u;
435       switch (u -> sub_tree) {
436       case LT:
437         u -> parent -> left = v;
438         break;
439       case RT:
440         u -> parent -> right = v;
441         break;
442       default:
443         header = v;
444       }
445       // 親を指すポインタを設定する
446       if (u -> left != NULL)
447         u -> left -> parent = u;
```

```
448       v -> parent = u -> parent;
449       u -> parent = v;
450       // どちらの部分木が高いかを設定する
451       if (v -> diff == LEFT) {
452         v -> diff = EVEN;
453         u -> diff = EVEN;
454       }
455       else {
456         v -> diff = RIGHT;
457         u -> diff = LEFT;
458       }
459       // 親に対して、どちらの部分木かを設定する
460       v -> sub_tree = u -> sub_tree;
461       u -> sub_tree = RT;
462       if (u -> left != NULL)
463         u -> left ->sub_tree = LT;
464     }
465
466     else {
467       // 子を指すポインタを設定する
468       u -> right = v -> left;
469       v -> left = u;
470       switch (u -> sub_tree) {
471       case LT:
472         u -> parent -> left = v;
473         break;
474       case RT:
475         u -> parent -> right = v;
476         break;
477       default:
478         header = v;
479       }
480       // 親を指すポインタを設定する
481       if (u -> right != NULL)
482         u -> right -> parent = u;
483       v -> parent = u -> parent;
484       u -> parent = v;
485       // どちらの部分木が高いかを設定する
486       if (v -> diff == RIGHT) {
487         v -> diff = EVEN;
488         u -> diff = EVEN;
489       }
490       else {
491         v -> diff = LEFT;
492         u -> diff = RIGHT;
493       }
494       // 親に対して、どちらの部分木かを設定する
495       v -> sub_tree = u -> sub_tree;
496       u -> sub_tree = LT;
497       if (u -> right != NULL)
498         u -> right -> sub_tree = RT;
499     }
500
501     return v;
502   }
503
504   /* ************************************************** *
505    *                    二重回転                        *
506    * 引数は高さが2高い節点を指すuとその子を指すvとその子  *
507    * を指すwである。                                    *
```

```
508   * 戻り値は、部分木の根を指すポインタ p である。                *
509   * ********************************************** */
510  struct avl *double_rotation(struct avl *u, struct avl *v, struct avl *w)
511  {
512    if (u -> diff == LEFT) {
513      // 子を指すポインタを設定する
514      v -> right = w -> left;
515      u -> left = w -> right;
516      w -> left = v;
517      w -> right = u;
518      switch (u -> sub_tree) {
519      case LT:
520        u -> parent -> left = w;
521        break;
522      case RT:
523        u -> parent -> right = w;
524        break;
525      default:
526        header = w;
527      }
528      // 親を指すポインタを設定する
529      w -> parent = u -> parent;
530      v -> parent = w;
531      u -> parent = w;
532      if (v -> right != NULL)
533        v -> right -> parent = v;
534      if (u -> left != NULL)
535        u -> left -> parent = u;
536      // どちらの部分木が高いかを設定する
537      if (w -> diff == LEFT) {
538        v -> diff = EVEN;
539        u -> diff = RIGHT;
540      }
541      else if (w -> diff == RIGHT) {
542        v -> diff = LEFT;
543        u -> diff = EVEN;
544      }
545      else {
546        v -> diff = EVEN;
547        u -> diff = EVEN;
548      }
549      w -> diff = EVEN;
550      // 親に対して、どちらの部分木かを設定する
551      w -> sub_tree = u -> sub_tree;
552      u -> sub_tree = RT;
553      if (v -> right != NULL)
554        v -> right -> sub_tree = RT;
555      if (u -> left != NULL)
556        u -> left -> sub_tree = LT;
557    }
558
559    else {
560      // 子を指すポインタを設定する
561      u -> right = w -> left;
562      v -> left = w -> right;
563      w -> left = u;
564      w -> right = v;
565      switch (u -> sub_tree) {
566      case LT:
567        u -> parent -> left = w;
568        break;
```

```
569       case RT:
570         u -> parent -> right = w;
571         break;
572       default:
573         header = w;
574       }
575       // 親を指すポインタを設定する
576       w -> parent = u -> parent;
577       v -> parent = w;
578       u -> parent = w;
579       if (u -> right != NULL)
580         u -> right -> parent = u;
581       if (v -> left != NULL)
582         v -> left -> parent = v;
583       // どちらの部分木が高いかを設定する
584       if (w -> diff == LEFT) {
585         u -> diff = EVEN;
586         v -> diff = RIGHT;
587       }
588       else if (w -> diff == RIGHT) {
589         v -> diff = EVEN;
590         u -> diff = LEFT;
591       }
592       else {
593         u -> diff = EVEN;
594         v -> diff = EVEN;
595       }
596       w -> diff = EVEN;
597       // 親に対して、どちらの部分木かを設定する
598       w -> sub_tree = u -> sub_tree;
599       u -> sub_tree = LT;
600       if (u -> right != NULL)
601         u -> right -> sub_tree = RT;
602       if (v -> left != NULL)
603         v -> left -> sub_tree = LT;
604     }
605
606     return w;
607   }
608
609   /* ********************************************** *
610    * 「データ」を探索する。(探索のアルゴリズム)          *
611    * 引数は、文字列strである。                        *
612    * 戻り値は、データが存在すれば'YES'を存在しない場合  *
613    * は'NO'を返す。                                  *
614    * ********************************************** */
615   enum answer search(char *str)
616   {
617     int s = 0;                          // 添字
618     struct avl *p;                      // 走査ポインタ
619     extern struct avl *header;
620
621     p = header;
622
623     while (p != NULL) {
624       if (str[s] < p -> element[s]) {
625         p = p -> left;
626         s = 0;
627       }
628       else if (str[s] > p -> element[s]) {
```

```
629         p = p -> right;
630         s = 0;
631       }
632       else {
633         if (s == S - 1) {
634           return YES;
635         }
636         else
637           s++;
638       }
639     }
640
641     return NO;
642   }
643
644   /* ************************************************** *
645    * 「データ」を削除する。(削除のアルゴリズム)          *
646    * 引数は探索データである str である。                 *
647    * 戻り値は、削除出来れば 'YES' を削除出来なかったら 'NO' を *
648    * 返す。                                             *
649    * ************************************************** */
650   enum answer delete(char *str)
651   {
652     int s = 0;                          // 配列の添字
653     struct avl *p;                      // 削除節点を指す
654     struct avl **q;                     // 走査ポインタ
655     extern struct avl *header;
656
657     q = &(header);
658     p = *q;
659
660     while (*q != NULL) {
661       p = *q;
662       if (str[s] < (*q) -> element[s]) {
663         q = &((*q) -> left);
664         s = 0;
665       }
666       else if (str[s] > (*q) -> element[s]) {
667         q = &((*q) -> right);
668         s = 0;
669       }
670       else {
671         if (s == S - 1) {
672           *q = delete_case(p);
673           return YES;
674         }
675         else
676           s++;
677       }
678     }
679
680     return NO;
681   }
682
683   /* ************************************************** *
684    * 削除する節点の子で削除の仕方を変える。(場合分けのアル *
685    * ゴリズム)                                          *
686    * 引数は削除する節点を指すポインタ r である。        *
687    * 戻り値は削除する節点を指すポインタ r を返す。削除する節 *
688    * 点の子が共になかったら NULL を返す。               *
```

```
689   * ************************************************** */
690  struct avl *delete_case(struct avl *r)
691  {
692    struct avl *p;          // 削除節点を指す
693    struct avl **q;         // 走査節点
694    extern struct avl *parent_node;
695    extern enum sub current_node;
696
697    // 削除する節点の子が共にNULLである。
698    if ((r -> left == NULL) && (r -> right == NULL)) {
699      current_node = r -> sub_tree;
700      parent_node = r -> parent;
701      free(r);
702      return NULL;
703    }
704
705    // 削除する節点の左の子がNULLである。
706    else if (r -> left == NULL) {
707      q = &(r -> right);
708      p = r -> right;
709      strcpy(r -> element, p -> element);
710      *q = delete_case(p);
711      return r;
712    }
713
714    // 削除する節点の右の子がNULLである。
715    else if (r -> right == NULL) {
716      q = &(r -> left);
717      p = r -> left;
718      strcpy(r -> element, p -> element);
719      *q = delete_case(p);
720      return r;
721    }
722
723    // 削除する節点の子が共に存在する。
724    else {
725      q = &(r -> right);
726      p = r -> right;
727      while (p -> left != NULL) {
728        q = &(p -> left);
729        p = p -> left;
730      }
731      strcpy(r -> element, p -> element);
732      *q = delete_case(p);
733      return r;
734    }
735  }
736
737  /* ************************************************ *
738   *      木を左部分木、節点、右部分木の順に印字する。    *
739   * 引数は部分木の根を指すポインタpである。            *
740   * ************************************************ */
741  void print_avl(struct avl *p)
742  {
743    if (p != NULL) {
744      print_avl(p -> left);
745      printf("%s\n", p -> element);
746      print_avl(p -> right);
747    }
748
```

```
749         return;
750     }
```

付録 C

B 木のソースコード

105 行目の read_temp 関数によって，乱数で生成した文字列ファイルを読んで，B 木を構築する．b_tree 関数で文字列を印字する．
構築は，insertR 関数で文字列を挿入する．節点のキーが多くなった場合は split 関数で節点を分割する．

```
1  /* ********************************************
2   *                    B_tree                    *
3   *             procedure Create & Print         *
4   * ******************************************** */
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <string.h>
9  #include <time.h>
10 #define N xxx                                  // 要素数
11 #define M xxx                                  // 進数
12 #define S xxx                                  // 桁数
13 #define I xxx                                  // 配列数
14
15 /* ********************************************* */
16 typedef struct STnode* link;
17 // 配列の構造
18 typedef struct {
19   char key[S+1];                                // キー
20   union {
21     link next;                                  // 節点へのポインタ
22     char element[S+1];                          // 要素
23   }ref;
24 }entry;
25
26 // 節点の構造
27 struct STnode {
28   entry b[I];                                   // I次のB木
29   int m;                                        // キーの個数
30 };
31 /* ********************************************* */
32
33 static link head;                              // 根を指すポインタ
34 static int H;                                  // 高さ
35 unsigned long int SAME = 0;                    // 同文字列
```

```
36  unsigned long int COMPARE = 0;          // 比較回数
37  unsigned long int ELEMENTS = 0;         // 印字された要素数
38
39  void STinit(void);
40  link NEW(void);
41  void read_temp(void);
42  void STinit(void);
43  void STinsert(char *);
44  link insertR(link, char *, int);
45  int less(char *, char *);
46  link split(link);
47  void b_tree(link, int);
48
49  /* *************** main関数 *************** */
50  int main(void)
51  {
52    extern unsigned long int SAME;
53    extern unsigned long int COMPARE;
54    extern unsigned long int ELEMENTS;
55    extern link head;
56    extern int H;
57    clock_t before;
58    double elapsed;
59
60    before = clock();
61    STinit();
62    read_temp();
63    elapsed = clock() - before;
64    printf("%.3f seconds\n", elapsed/CLOCKS_PER_SEC);
65    b_tree(head, H);
66    printf("element are %ld.\n", ELEMENTS);
67    printf("same_characters are %ld.\n", SAME);
68    printf("compare_times are %ld.\n", COMPARE);
69
70    return 0;
71  }
72
73  /* ********************************************* *
74   *                 根の高さを0にする             *
75   * ********************************************* */
76  void STinit(void)
77  {
78    extern link head;
79
80    head = NEW();
81    H = 0;
82
83    return;
84  }
85
86  /* ********************************************* *
87   *               節点を作成する                  *
88   * ********************************************* */
89  link NEW(void)
90  {
91    int i;                                  // カウント
92    link x;                                 // 新節点を指すポインタ
93
94    x = malloc(sizeof *x);
95    for (i = 0; i < I; i++)
```

```
96        memset(x -> b[i].key, '0', S+1);
97      x -> m = 0;
98
99      return x;
100   }
101
102   /* ********************************************* *
103    *           tempファイルから文字列を読む          *
104    * ********************************************* */
105   void read_temp(void)
106   {
107     FILE *file;
108     int n, s;                                  // カウント
109     char str[S+1];                             // 文字列S桁
110
111     if ((file = fopen("xxx", "r")) == NULL) {
112       puts("can't open file");
113       exit(1);
114     }
115
116     for (n = 0; n < N; n++) {
117       for (s = 0; s < S; s++)
118         str[s] = getc(file);
119       str[s] = '\0';
120       STinsert(str);
121     }
122
123     return;
124   }
125
126   /* ********************************************* *
127    *                根における処理                 *
128    * ********************************************* */
129   void STinsert(char *str)
130   {
131     link t;                                    // 新しい根節点
132     link u;                                    // 判定用ポインタ
133     extern link head;
134
135     u = insertR(head, str, H);
136
137     // 分割していない
138     if (u == NULL)
139       return;
140
141     // 分割した
142     t = NEW();
143     t -> m = 2;
144     memcpy(t -> b[0].key, head -> b[0].key, S+1);
145     t -> b[0].ref.next = head;
146     memcpy(t -> b[1].key, u -> b[0].key, S+1);
147     t -> b[1].ref.next = u;
148     head = t;
149     H++;
150
151     return;
152   }
153
154   /* ********************************************************** *
155    *  戻値は、分割しなかったらNULLで、分割したら新節点へのポインタ  *
```

```
156   * ********************************************************** */
157  link insertR(link h, char *str, int H)
158  {
159    extern unsigned long int SAME;
160    int i, j;                                    // カウント
161    int s;                                       // 関数lessの戻り値
162    entry x;                                     // 配列
163    link t;                                      // 子節点
164    link u;                                      // 新しい節点へのポインタ
165
166    // 配列xのキーと要素の値に代入
167    memcpy(x.key, str, S+1);
168    memcpy(x.ref.element, str, S+1);
169
170    // 外部節点
171    if (H == 0)
172      for (j = 0; j < h -> m; j++) {
173        s = less(x.key, h -> b[j].key);
174        if (s == 1)                              // x.key < h->b[j].key
175          break;
176        else if (s == 0)                         // x.key > h->b[j].key
177          ;
178        else {                                   // 同文字列が挿入
179          SAME++;
180          return NULL;
181        }
182      }
183
184    // 内部節点
185    else {
186      for (j = 0; j < h -> m; j++) {
187        // 同文字列が挿入
188        if (less(x.key, h -> b[j].key) == 2) {
189          SAME++;
190          return NULL;
191        }
192        // 子節点へ辿る
193        else if ((j + 1 == h -> m) || less(x.key, h -> b[j+1].key) == 1) {
194          t = h -> b[j++].ref.next;
195          u = insertR(t, str, H-1);
196          if (u == NULL)
197            return NULL;
198          memcpy(x.key, u -> b[0].key, S+1);
199          x.ref.next = u;
200          break;
201        }
202        // 次の配列と比較
203        else
204          ;
205      }
206    }
207
208    // 文字列を挿入
209    for (i = (h->m)++; i > j; i--)
210      h -> b[i] = h -> b[i-1];
211    h -> b[j] = x;
212
213    if (h->m < I)
214      return NULL;
215    else
```

```
216     return split(h);
217 }
218
219 /* ********************************************* *
220  *               region1 < region2               *
221  * ********************************************* */
222 int less(char *region1, char *region2)
223 {
224   extern unsigned long int COMPARE;
225   int i;                                          // カウント
226
227   for (i = 0; i < S; i++) {
228     COMPARE++;
229     if (region1[i] < region2[i])
230       return 1;
231     else if (region1[i] > region2[i])
232       return 0;
233   }
234
235   // 同文字列
236   return 2;
237 }
238
239 /* ********************************************* *
240  *                  節点の分割                   *
241  * ********************************************* */
242 link split(link h)
243 {
244   int j;                                          // カウント
245   link t;                                         // 新節点へのポインタ
246
247   t = NEW();
248   for (j = 0; j < I/2; j++)
249     t -> b[j] = h -> b[I/2+j];
250   h -> m = I/2;
251   t -> m = I/2;
252
253   return t;
254 }
255
256 /* ********************************************* *
257  *                  要素の印字                   *
258  * ********************************************* */
259 void b_tree(link p, int H)
260 {
261   extern unsigned long int ELEMENTS;
262   int i;                                           // カウント
263
264   if (H != 0)
265     for (i = 0; i < p->m; i++)
266       b_tree(p -> b[i].ref.next, H-1);
267
268   else
269     for (i = 0; i < p->m; i++) {
270       ELEMENTS++;
271       // printf("%s\n", p -> b[i].ref.element);
272     }
273
274   return;
275 }
```

付録 D

拡張した AVL 木のソースコード

123 行目の random_array 関数によって，乱数を使ってデータ数を N 個，文字列を S 桁生成して，拡張した AVL 木を構築する．その後に，main 関数のメニューによって「挿入」,「探索」,「削除」,「印字」を行うプログラムである．96 行目の random_array 関数によって，乱数を使ってデータ数を N 個，文字列を S 桁生成して，拡張した AVL 木を構築する．その後に，main 関数のメニューによって「挿入」,「探索」,「削除」,「印字」を行うプログラムである．
「挿入」は，S 桁の文字列を入力後，insert 関数で挿入位置を決定して make_leaf 関数で葉を作る．その後，think 関数で回転操作の有無を考察する．
「探索」は，S 桁の文字列を入力後，search 関数で探索結果を返す．
「削除」は，S 桁以下の文字列を入力後，delete/delete_group 関数で文字列を削除する．
「印字」は，ext_avl 関数で中順で印字する．

```
1  /* ****************************************
2   *                                        *
3   *                ext_AVL_tree            *
4   *          procedure Create, Search,     *
5   *              Delete & Print            *
6   *                                        *
7   ****************************************/
8
9  #include <stdio.h>
10 #include <time.h>
11 #include <string.h>
12 #include <stdlib.h>
13 #define N xxx                          // データ数
14 #define M xxx                          // 進数
15 #define S xxx                          // 文字列の長さ
16 #define D xxx                          // 文字列
17
18 enum diff {EVEN, LEFT, RIGHT};         // どちらの部分木が高いか
19 enum flag {LA, NU};                    // ラベルならLA,データならばNU
20 enum sub {RO, LT, RT, FT, BT, CT};     // どこの部分木か
21
22 // ext_AVL_treeのデータ構造
23 struct ext_avl {
24   char element[S+1];                   // データまたはラベルをS桁格納
25   enum flag flag;                      // enum flag 参照
26   enum sub sub_tree;                   // enum sub 参照
27   enum diff diff;                      // enum diff 参照
28   struct ext_avl *left;                // 左部分木を指すポインタ
```

```
29    struct ext_avl *right;              // 右部分木を指すポインタ
30    struct ext_avl *front;              // 前部分木を指すポインタ
31    struct ext_avl *back;               // 後部分木を指すポインタ
32    struct ext_avl *center;             // 中央部分木を指すポインタ
33    struct ext_avl *parent;             // 親を指すポインタ
34  };
35
36  struct ext_avl *header = NULL;        // 根を指すポインタ
37
38  int even_odd(void);
39  void random_array(char, char);
40  int menu(void);
41  void Insert(char, char);
42  void Search(void);
43  void Delete(void);
44  void Print(void);
45  struct ext_avl *insert(char *, char ,char);
46  struct ext_avl *make_leaf(char *, enum sub, struct ext_avl *);
47  void think(enum sub, struct ext_avl *);
48  void rebuilt(struct ext_avl *, struct ext_avl *);
49  void single_rotation(struct ext_avl *, struct ext_avl *);
50  void double_rotation(struct ext_avl *, struct ext_avl *, struct ext_avl
        *);
51  void search(char *, int);
52  struct ext_avl *delete(char *, int);
53  struct ext_avl *delete_group(struct ext_avl *);
54  void ext_avl(struct ext_avl *);
55
56  /* ******************************************************* *
57   *                          main関数                       *
58   * 中央値a, bを求め、木Tを構築する。その後にメニューに     *
59   * 従って、各操作を行う。                                  *
60   * ******************************************************* */
61  int main(void)
62  {
63    char a, b;                        // M進数の中央値
64    int m;                            // M進数は偶数または奇数
65    int bt;                           // メニュー
66
67    m = even_odd();
68    if (m == 0) {
69      a = ((M - 1) / 2) + '0';
70      b = (M / 2) + '0';
71    } else {
72      a = ((M - 1) / 2) + '0';
73      b = ((M - 1) / 2) + '0';
74    }
75
76    random_array(a, b);
77
78    while (1) {
79
80      bt = menu();
81
82      switch (bt) {
83      case 'I':                                   // 挿入
84        Insert(a, b);
85        putchar('\n');
86        break;
```

```
87      case 'F':                                       // 探索
88        Search();
89        putchar('\n');
90        break;
91      case 'D':                                       // 削除
92        Delete();
93        putchar('\n');
94        break;
95      case 'P':                                       // 印字
96        Print();
97        putchar('\n');
98        break;
99      case 'E':                                       // 終了
100       exit(0);
101       break;
102     default:
103       puts("input, again!\n");
104     }
105   }
106
107   return 0;
108 }
109
110 /* ************************************************ *
111  *     平衡化に必要なM進数が偶数か奇数かを判定する      *
112  * 戻り値は、偶数ならば0、奇数ならば1を返す。          *
113  * ************************************************ */
114 int even_odd(void)
115 {
116   return (M % 2);
117 }
118
119 /* ************************************************ *
120  * データを最大N個発生させて、木TをAVL+木で構築する。   *
121  * 引数は中央値a, bである。                           *
122  * ************************************************ */
123 void random_array(char a, char b)
124 {
125   int n, s;                                       // カウント
126   char str[S+1];                                  // 文字列S桁
127   struct ext_avl *leaf;                           // 葉
128
129   srand(time(NULL));
130
131   for (n = 0; n < N; n++) {
132     for (s = 0; s < S; s++)
133       str[s] = (rand() % 10) + '0';
134     str[s] = '\0';
135
136     leaf = insert(str, a, b);
137
138     if (leaf != NULL)
139       think(leaf -> sub_tree, leaf -> parent);
140   }
141
142   return ;
143 }
144
145 /* ************************************************ *
146  *                  メニューを表示する。             *
```

```
147   * 戻り値は各操作のキーワードである。                          *
148   * ************************************************ */
149  int menu(void)
150  {
151    int ch;                              // 各操作のキーワード
152    char gar[D];                         // garbage
153
154    puts("MENU\n\tI...insert\n\tF...search\n\tD...delete\n\tP...print\n\tE
         ...end");
155    ch = getchar();
156    gets(gar);
157
158    return ch;
159  }
160
161  /* ************************************************ *
162   *    挿入したい文字列を入力(葉が出来たら考察をする)    *
163   * 引数は中央値a, bである。                             *
164   * ************************************************ */
165  void Insert(char a, char b)
166  {
167    int c;                               // 一文字
168    int s;                               // 添字
169    char str[D];                         // 文字列S桁
170    int len;                             // 文字列長
171    struct ext_avl *leaf;
172
173    do {
174      printf("input number(%dwords).\n", S);
175      gets(str);
176      len = strlen(str);
177    } while (len != S);
178
179    leaf = insert(str, a, b);
180
181    if (leaf != NULL)
182      think(leaf -> sub_tree, leaf -> parent);
183    else
184      puts("ALREADY EXIST.");
185
186    return ;
187  }
188
189  /* ************************************************ *
190   *       木Tを探索する。(データ又はグループを入力)      *
191   * ************************************************ */
192  void Search(void)
193  {
194    int len;                             // 文字列長
195    char str[D];                         // 文字列
196
197    printf("input group or number.\n");
198    gets(str);
199    len = strlen(str);
200    if (len <= S)
201      search(str, len);
202    else
203      puts("wrong size!");
204
205    return;
```

```
206  }
207
208  /* ************************************************** *
209   * 入力する文字列を削除する。(データ又はグループを削除)  *
210   * ************************************************** */
211  void Delete(void)
212  {
213    char str[D];                          // 文字列
214    int len;                              // 文字列長
215    struct ext_avl *p;
216
217    puts("input delete number.");
218    gets(str);
219    len = strlen(str);
220    if (len > S)
221      puts("wrong size!");
222
223    p = delete(str, len);
224
225    if (p != NULL)
226        p -> flag = LA;
227    else
228      puts("NOT EXIST(group or number)!");
229
230    return;
231  }
232
233  /* ************************************************** *
234   *              木 Tをなぞりを利用して印字する。          *
235   * ************************************************** */
236  void Print(void)
237  {
238    ext_avl(header);
239
240    return;
241  }
242
243  /* ************************************************** *
244   *       データ strを木 Tに挿入する。(挿入のアルゴリズム)   *
245   * 引数は、文字列 strと中央値 a, bである。               *
246   * 戻り値は、挿入できたら葉を指すポインタで、挿入できな   *
247   * かったら NULLを返す。                               *
248   * ************************************************** */
249  struct ext_avl *insert(char *str, char a, char b)
250  {
251    int s = 0;                            // 添字
252    enum sub sub;                         // どの方向に進行したか?
253    struct ext_avl *p;                    // 親節点を指すポインタ
254    struct ext_avl **q;                   // 走査ポインタ
255    char swap[S+1];                       // 文字列を保存
256
257    sub = RO;
258    q = &(header);
259    p = *q;
260
261    while (*q != NULL) {
262      p = *q;
263
264      /* ***** 0桁比較 ***** */
265      if (str[s] < (*q) -> element[s]) {
```

```
266         sub = LT;
267         q = &((*q) -> left);
268       }
269
270       else if (str[s] > (*q) -> element[s]) {
271         sub = RT;
272         q = &((*q) -> right);
273       }
274
275       /* ***** 1桁比較 ***** */
276       else {
277         s++;
278         if (str[s] == '\0') {
279           // 削除していた場合
280           if ((*q) -> flag == LA) {
281             (*q) -> flag = NU;
282             return NULL;
283           }
284           // 既に要素が存在していた場合
285           else
286             return NULL;
287         }
288
289         else {
290           /* ***** 0桁・1桁同じ ***** */
291           if (str[s] == (*q) -> element[s]) {
292             s++;
293             if (str[s] == '\0') {
294               // 削除していた場合
295               if ((*q) -> flag == LA) {
296                 (*q) -> flag = NU;
297                 return NULL;
298               }
299               // 既に要素が存在していた場合
300               else
301                 return NULL;
302             }
303
304             else {
305               // center_nodeあり
306               if ((*q) -> center != NULL)
307                 q = &((*q) -> center);
308
309               // center_nodeなし
310               else {
311                 (*q) -> flag = LA;
312                 p = *q;
313                 sub = CT;
314                 (*q) -> center = make_leaf((*q) -> element, sub, p);
315                 q = &((*q) -> center);
316               }
317             }
318           }
319           else {
320             // 節点が平衡状態となる場合
321             if ((((*q) -> element[s] == a) || ((*q) -> element[s] == b))
                      || (*q) -> center != NULL) {
322               if (str[s] < (*q) -> element[s]) {
323                 sub = FT;
324                 q = &((*q) -> front);
```

```
325             }
326             else {
327               sub = BT;
328               q = &((*q) -> back);
329             }
330           }
331           else {
332             // データが平衡状態になる場合
333             if ((str[s] == a) || (str[s] == b)) {
334               strcpy(swap, (*q) -> element);
335               strcpy((*q) -> element, str);
336               strcpy(str, swap);
337               if (str[s] < (*q) -> element[s]) {
338                 sub = FT;
339                 q = &((*q) -> front);
340               }
341               else {
342                 sub = BT;
343                 q = &((*q) -> back);
344               }
345             }
346             // 平衡状態の値に近い方を選択
347             else {
348               if (((*q) -> element[s]) > b) {
349                 if (abs(b - str[s]) >= abs(b - ((*q) -> element[s]))) {
350                   if (str[s] < (*q) -> element[s]) {
351                     sub = FT;
352                     q = &((*q) -> front);
353                   }
354                   else {
355                     sub = BT;
356                     q = &((*q) -> back);
357                   }
358                 }
359                 else {
360                   strcpy(swap, (*q) -> element);
361                   strcpy((*q) -> element, str);
362                   strcpy(str, swap);
363                   if (str[s] < (*q) -> element[s]) {
364                     sub = FT;
365                     q = &((*q) -> front);
366                   }
367                   else {
368                     sub = BT;
369                     q = &((*q) -> back);
370                   }
371                 }
372               }
373               else {
374                 if (abs(a - str[s]) >= abs(a - ((*q) -> element[s]))) {
375                   if (str[s] < (*q) -> element[s]) {
376                     sub = FT;
377                     q = &((*q) -> front);
378                   }
379                   else {
380                     sub = BT;
381                     q = &((*q) -> back);
382                   }
383                 }
```

```
384                   else {
385                     strcpy(swap, (*q) -> element);
386                     strcpy((*q) -> element, str);
387                     strcpy(str, swap);
388                     if (str[s] < (*q) -> element[s]) {
389                       sub = FT;
390                       q = &((*q) -> front);
391                     }
392                     else {
393                       sub = BT;
394                       q = &((*q) -> back);
395                     }
396                   }
397                 }
398               }
399             }
400           }
401         }
402       }
403     }
404     *q = make_leaf(str, sub, p);
405
406     return *q;
407   }
408
409   /* ************************************************ *
410    *                    葉を生成する。                *
411    * 引数はデータ str、葉がどの部分木となるかを ?を示す sub と *
412    * 葉の親を指すポインタ r である。                  *
413    * 戻り値は、葉を指すポインタである。               *
414    * ************************************************ */
415   struct ext_avl *make_leaf(char *str, enum sub sub, struct ext_avl *r)
416   {
417     struct ext_avl *leaf;              // 葉を指すポインタ
418
419     if ((leaf = malloc(sizeof(struct ext_avl))) == NULL) {
420       printf("out of memory!!\n");
421       exit(1);
422     }
423
424     strcpy(leaf -> element, str);
425     leaf -> flag = NU;
426     leaf -> sub_tree = sub;
427     leaf -> diff = EVEN;
428     leaf -> left = NULL;
429     leaf -> right = NULL;
430     leaf -> front = NULL;
431     leaf -> back = NULL;
432     leaf -> center = NULL;
433     leaf -> parent = r;
434
435     return leaf;
436   }
437
438   /* ************************************************ *
439    *        高さの差を考察する。(考察のアルゴリズム)  *
440    * 引数は節点がどの部分木であるか?を示す sub と節点の親 *
441    * へのポインタ u である。                          *
442    * ************************************************ */
443   void think(enum sub sub, struct ext_avl *u)
```

```
444 {
445   // subが前・後・中央部分木であれば、直ちに考察終了
446   if (sub == FT || sub == BT || sub == CT)
447     return;
448
449   // 根まで到達したら終了とする
450   if (u == NULL)
451     return;
452   else
453     switch (sub) {
454       // 左部分木の高さが増えた
455     case LT:
456       if (u -> diff == RIGHT) {
457         u -> diff = EVEN;
458         return;
459       }
460       else if (u -> diff == LEFT)
461         rebuilt(u, u -> left);
462       else {
463         u -> diff = LEFT;
464         think(u -> sub_tree, u -> parent);
465       }
466       break;
467
468       // 右部分木の高さが増えた
469     case RT:
470       if (u -> diff == LEFT) {
471         u -> diff = EVEN;
472         return;
473       }
474       else if (u -> diff == RIGHT)
475         rebuilt(u, u -> right);
476       else {
477         u -> diff = RIGHT;
478         think(u -> sub_tree, u -> parent);
479       }
480       break;
481     }
482
483   return;
484 }
485
486 /* ************************************************ *
487  *  どの回転を行うかを選別する。(再構築のアルゴリズム)  *
488  * 引数は高さが2高い節点を指すuとその子を指すvである。  *
489  * ************************************************ */
490 void rebuilt(struct ext_avl *u, struct ext_avl *v)
491 {
492   // 左部分木が2高くなった場合
493   if (u -> diff == LEFT) {
494     // 一重回転の場合
495     if (v -> diff == LEFT)
496       single_rotation(u, v);
497     // 二重回転の場合
498     else
499       double_rotation(u, v, v -> right);
500   }
501
502   else {
503     // 右部分木が2高くなった場合
```

```
504       // 一重回転の場合
505       if (v -> diff == RIGHT)
506         single_rotation(u, v);
507       // 二重回転の場合
508       else
509         double_rotation(u, v, v -> left);
510     }
511
512     return;
513   }
514
515   /* ************************************************** *
516    *                      一重回転                      *
517    * 引数は高さが2高い節点を指すuとその子を指すvである。 *
518    * ************************************************** */
519   void single_rotation(struct ext_avl *u, struct ext_avl *v)
520   {
521     if (u -> diff == LEFT) {
522       // 子を指すポインタを設定する
523       u -> left = v -> right;
524       v -> right = u;
525       switch (u -> sub_tree) {
526       case LT:
527         u -> parent -> left = v;
528         break;
529       case RT:
530         u -> parent -> right = v;
531         break;
532       case FT:
533         u -> parent -> front = v;
534         break;
535       case BT:
536         u -> parent -> back = v;
537         break;
538       case CT:
539         u -> parent -> center = v;
540         break;
541       default:
542         header = v;
543       }
544       // 親を指すポインタを設定する
545       if (u -> left != NULL)
546         u -> left -> parent = u;
547       v -> parent = u -> parent;
548       u -> parent = v;
549       // どちらの部分木が高いかを設定する
550       v -> diff = EVEN;
551       u -> diff = EVEN;
552       // 親に対して、どちらの部分木かを設定する
553       v -> sub_tree = u -> sub_tree;
554       u -> sub_tree = RT;
555       if (u -> left != NULL)
556         u -> left ->sub_tree = LT;
557     }
558
559     else {
560       // 子を指すポインタを設定する
561       u -> right = v -> left;
562       v -> left = u;
563       switch (u -> sub_tree) {
564       case LT:
```

```
565         u -> parent -> left = v;
566         break;
567       case RT:
568         u -> parent -> right = v;
569         break;
570       case FT:
571         u -> parent -> front = v;
572         break;
573       case BT:
574         u -> parent -> back = v;
575         break;
576       case CT:
577         u -> parent -> center = v;
578         break;
579       default:
580         header = v;
581       }
582       // 親を指すポインタを設定する
583       if (u -> right != NULL)
584         u -> right -> parent = u;
585       v -> parent = u -> parent;
586       u -> parent = v;
587       // どちらの部分木が高いかを設定する
588       v -> diff = EVEN;
589       u -> diff = EVEN;
590       // 親に対して、どちらの部分木かを設定する
591       v -> sub_tree = u -> sub_tree;
592       u -> sub_tree = LT;
593       if (u -> right != NULL)
594         u -> right -> sub_tree = RT;
595     }
596
597     return;
598   }
599
600   /* ************************************************ *
601    *                    二重回転                      *
602    * 引数は高さが2高い節点を指すuとその子を指すvとその子  *
603    * を指すwである。                                  *
604    * ************************************************ */
605   void double_rotation(struct ext_avl *u, struct ext_avl *v, struct
        ext_avl *w)
606   {
607     if (u -> diff == LEFT) {
608       // 子を指すポインタを設定する
609       v -> right = w -> left;
610       u -> left = w -> right;
611       w -> left = v;
612       w -> right = u;
613       switch (u -> sub_tree) {
614       case LT:
615         u -> parent -> left = w;
616         break;
617       case RT:
618         u -> parent -> right = w;
619         break;
620       case FT:
621         u -> parent -> front = w;
622         break;
623       case BT:
624         u -> parent -> back = w;
```

```
625         break;
626       case CT:
627         u -> parent -> center = w;
628         break;
629       default:
630         header = w;
631       }
632       // 親を指すポインタを設定する
633       w -> parent = u -> parent;
634       v -> parent = w;
635       u -> parent = w;
636       if (v -> right != NULL)
637         v -> right -> parent = v;
638       if (u -> left != NULL)
639         u -> left -> parent = u;
640       // どちらの部分木が高いかを設定する
641       if (w -> diff == LEFT) {
642         v -> diff = EVEN;
643         u -> diff = RIGHT;
644       }
645       else if (w -> diff == RIGHT) {
646         v -> diff = LEFT;
647         u -> diff = EVEN;
648       }
649       else {
650         v -> diff = EVEN;
651         u -> diff = EVEN;
652       }
653       w -> diff = EVEN;
654       // 親に対して、どちらの部分木かを設定する
655       w -> sub_tree = u -> sub_tree;
656       u -> sub_tree = RT;
657       if (v -> right != NULL)
658         v -> right -> sub_tree = RT;
659       if (u -> left != NULL)
660         u -> left -> sub_tree = LT;
661     }
662
663     else {
664       // 子を指すポインタを設定する
665       u -> right = w -> left;
666       v -> left = w -> right;
667       w -> left = u;
668       w -> right = v;
669       switch (u -> sub_tree) {
670       case LT:
671         u -> parent -> left = w;
672         break;
673       case RT:
674         u -> parent -> right = w;
675         break;
676       case FT:
677         u -> parent -> front = w;
678         break;
679       case BT:
680         u -> parent -> back = w;
681         break;
682       case CT:
683         u -> parent -> center = w;
684         break;
685       default:
```

```
686         header = w;
687       }
688       // 親を指すポインタを設定する
689       w -> parent = u -> parent;
690       v -> parent = w;
691       u -> parent = w;
692       if (u -> right != NULL)
693         u -> right -> parent = u;
694       if (v -> left != NULL)
695         v -> left -> parent = v;
696       // どちらの部分木が高いかを設定する
697       if (w -> diff == LEFT) {
698         u -> diff = EVEN;
699         v -> diff = RIGHT;
700       }
701       else if (w -> diff == RIGHT) {
702         v -> diff = EVEN;
703         u -> diff = LEFT;
704       }
705       else {
706         u -> diff = EVEN;
707         v -> diff = EVEN;
708       }
709       w -> diff = EVEN;
710       // 親に対して、どちらの部分木かを設定する
711       w -> sub_tree = u -> sub_tree;
712       u -> sub_tree = LT;
713       if (u -> right != NULL)
714         u -> right -> sub_tree = RT;
715       if (v -> left != NULL)
716         v -> left -> sub_tree = LT;
717     }
718
719     return;
720   }
721
722   /* ************************************************ *
723    * 「データ」及び「グループ」を探索する。(探索のアル  *
724    * ゴリズム)                                        *
725    * 引数は、「データ」または「グループ」の文字列strと  *
726    * 文字列の長さlenである。                          *
727    * ************************************************ */
728   void search(char *str, int len)
729   {
730     int s = 0;                          // 添字
731     struct ext_avl *p;                  // 走査ポインタ
732     extern struct ext_avl *header;
733
734     p = header;
735
736     while (p != NULL) {
737
738       /* ***** 0桁の大小を比較 ***** */
739       // 探索データが節点データより小さい場合
740       if (str[s] < p -> element[s])
741         p = p -> left;
742
743       // 探索データが節点データより大きい場合
744       else if (str[s] > p -> element[s])
745         p = p -> right;
```

```
746
747       // 探索データと節点データが同じ場合
748       else {
749         if (s == len - 1) {
750           ext_avl(p -> front);
751           if (p -> flag == NU)
752             printf("%s\n", p -> element);
753           ext_avl(p -> center);
754           ext_avl(p -> back);
755           return;
756         }
757         else {
758           /* ***** 1桁を比較 ***** */
759           s++;
760           // 探索データが節点データより小さい場合
761           if (str[s] < p -> element[s])
762             p = p -> front;
763
764           // 探索データが節点データより大きい場合
765           else if (str[s] > p -> element[s])
766             p = p -> back;
767
768           // 探索データと節点データが同じ場合
769           else {
770             if (s == len - 1) {
771               if (p -> flag == NU)
772                 printf("%s\n", p -> element);
773               ext_avl(p -> center);
774               return;
775             }
776             else {
777               if (p -> center != NULL)
778                 p = p -> center;
779               s++;
780             }
781           }
782         }
783       }
784     }
785     puts("NOT_EXIST!");
786
787     return;
788   }
789
790   /* ************************************************** *
791    * 「データ」または「グループ」を削除する。(削除のアルゴ  *
792    * リズム)                                            *
793    * 引数は探索データである str と文字列長である len である。  *
794    * 戻り値は節点データを指すポインタ *q、削除するデータが   *
795    * なければ NULL を返す。                              *
796    * ************************************************** */
797   struct ext_avl *delete(char *str, int len)
798   {
799     int s = 0;                          // 配列の添字
800     struct ext_avl **q;                 // 走査ポインタ
801     extern struct ext_avl *header;
802
803     q = &header;
804
805     while (*q != NULL) {
```

```
806
807       /* *****  0桁を比較 ***** */
808       if (str[s] < (*q) -> element[s])
809         q = &((*q) -> left);
810
811       else if (str[s] > (*q) -> element[s])
812         q = &((*q) -> right);
813
814       else {
815         if (s == len - 1) {
816           // グループのとき
817           if (len < S) {
818             (*q) -> front = delete_group((*q) -> front);
819             (*q) -> center = delete_group((*q) -> center);
820             (*q) -> back = delete_group((*q) -> back);
821             return *q;
822           }
823           // データのとき
824           else
825             return *q;
826         }
827         else {
828           /* ***** 1桁を比較 ***** */
829           s++;
830           if (str[s] < (*q) -> element[s])
831             q = &((*q) -> front);
832
833           else if (str[s] > (*q) -> element[s])
834             q = &((*q) -> back);
835
836           else {
837             if (s == len - 1) {
838               // グループのとき
839               if (len < S) {
840                 (*q) -> center = delete_group((*q) -> center);
841                 return *q;
842               }
843               // データのとき
844               else
845                 return *q;
846             }
847             else {
848               if ((*q) -> center != NULL)
849                 q = &((*q) -> center);
850               s++;
851             }
852           }
853         }
854       }
855     }
856
857     return NULL;
858   }
859
860   /* ********************************************* *
861    *              「グループ」を削除する。              *
862    * 引数は節点データを指すポインタrである。              *
863    * 戻り値はNULLである。                               *
864    * ********************************************* */
865   struct ext_avl *delete_group(struct ext_avl *r)
```

```
866  {
867    struct ext_avl **q;                  // 走査ポインタ
868
869    q = &r;
870
871    if (*q != NULL) {
872      (*q) -> left = delete_group((*q) -> left);
873      (*q) -> front = delete_group((*q) -> front);
874      (*q) -> center = delete_group((*q) -> center);
875      (*q) -> back = delete_group((*q) -> back);
876      (*q) -> right = delete_group((*q) -> right);
877      free(r);
878    }
879
880    return NULL;
881  }
882
883  /* ********************************************** *
884   *      木を左部分木、前部分木、節点、中央部分木      *
885   *           後部分木、右部分木の順に印字する。      *
886   * 引数は、部分木の根を指すポインタ pである。       *
887   * ********************************************** */
888  void ext_avl(struct ext_avl *p)
889  {
890    if (p != NULL) {
891      ext_avl(p -> left);
892      ext_avl(p -> front);
893      if (p -> flag == NU)
894        printf("%s\n", p -> element);
895      ext_avl(p -> center);
896      ext_avl(p -> back);
897      ext_avl(p -> right);
898    }
899
900      return;
901  }
```