プログラミング-言語・基礎・実践- 11－1
(1993. 3. 10)

Persistent tree を用いる二分探索木の対話的視覚化

別所正隆　今宮淳美
山梨大学　電子情報工学科
山梨県甲府市武田 4-3-11

あらまし　本稿では，平衡二分探索木のひとつである赤黒木を対話的に操作し，更新操作による赤黒木の構造変化を図示する対話型システムの実現について述べる．このシステムで操作する赤黒木は，Persistent tree 表現を用いて，木構造の変化の履歴を保持する木構造である．履歴を保持しているので，赤黒木に関するアルゴリズムを理解しやすく，アルゴリズムの設計・解析に利用することができる．さらに Persistent tree は履歴を参照するだけでなく，ある時点の木構造に戻って操作を実行し直すことができるので，更新操作に対するアンドゥ機能が可能となる．対話型システムの作成について述べるとともに，このシステムで重要となる履歴データの管理方法および保存方法を提案する．
和文キーワード　対話型システム，赤黒木，パーシステント木，バージョン管理，視覚化

An interactive visualization of binary search trees with the persistent tree

Masataka Bessho　Atsumi Imamiya
Department of Electrical Engineering and Computer Science, Yamanashi University
4-3-11 Takeda,Kofu,Yamanashi 400,Japan

Abstract　This paper presents the implementation of an interactive system to operate and visualize a balanced binary search tree, called a red-black tree. This system allows user to perform search tree operations on a red-black tree, and keep the operationed history(tree's versions) by maintaining the tree's structures, using the persistent tree. Since the persistent tree of red-black trees allows user to return to any version, the system provides user with multilevel undo facility.

英文 key words　interactive system, red-black tree, persistent tree, version management, visualization

1 はじめに

Tarjan は理論と実践の関係を深める必要性から実際にプログラムを書きコンピュータ上で実行するというアルゴリズムの実験的解析の重要性を指摘している [12]. 通常，データ構造に関するアルゴリズムの設計・解析，およびアルゴリズムの動作を理解するのに私たちはデータ構造の変化を図示してみる．データ構造を対話的に操作し，データ構造の変化をディスプレイ上に図示する対話型システムはアルゴリズムの設計・解析には重要であると私たちは考える．

本研究の目的は，平衡二分探索木のひとつである赤黒木をコンピュータ上で対話的に操作して木構造の変化を図示することにより，赤黒木に関するアルゴリズムを理解するための対話型システムを構築することである．この対話型システムで重要な機能は，赤黒木の変化の履歴をいかに保持するかということと，赤黒木の更新操作に対するアンドゥ機能である．これらの機能を実現するために Persistent tree を利用する．

Persistent tree[1] とは更新操作を連続して実行した場合の木構造の変化履歴を残す木である．従来，木は常にひとつの状態しか保持していないので，この木の構造から実行した更新操作を推測しなければならなかった．実行した操作や操作後の木構造の状態をすべて記憶していれば，木に対する操作や動作を理解しやすく，数学的解析にも利用できる．

また，Persistent tree は単にその履歴を参照するだけでなく，ある時点の木構造の状態に戻って処理を続けることのできる木である．通常，アンドゥ機能は更新操作を実行する前の状態に戻って処理を続けるための機能であるが，Persistent tree はそれまでに発生した状態すべてに戻ることができる．これらの Persistent tree の利点を用いて，履歴を残す赤黒木の更新操作に対するアンドゥ機能を実現する．作成する対話型システムでは，履歴を残していることによって実現できる質問操作を用意する．また，保持している履歴データの管理方法および保存方法を提案する．

2 赤黒木

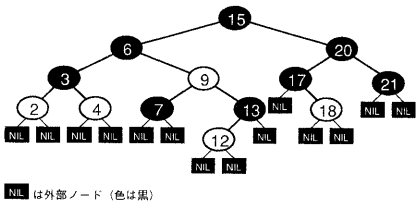


図 1: 赤黒木

二分探索木の基本操作は探索，挿入および削除である．キーの探索はルートから木の下方に向かってノードを順に調べる．ノードの挿入および削除は，まず操作するキーを探索する．挿入は，挿入するキーの探索に失敗したとき，最後に訪れたノードの子として挿入する．削除するノードが 2 つの子をもつ場合，右部分木の最小ノードのキーに置き換え，実際には最小ノードを削除する．

n 個のノードを含む二分探索木の探索は，平均 $O(\lg n)$ 時間，ノードが線形に結合された最悪の場合は $O(n)$ 時間を必要とする．回転操作によって木の高さを可能な限り低くして，バランスのとれた構造にすれば，訪れるノード数が減り，探索の実行時間は少なくなる．この平衡化はノードの挿入や削除にともない，バランスのとれた木構造に作り直す技法で，回転操作を基本アルゴリズムとする．平衡二分探索木は木の深さが $O(\lg n)$ であるので，探索，挿入，削除の基本操作が $O(\lg n)$ 時間で実行できる．

赤黒木 [4, 6, 7] はノードごとに 1 ビット余分な記憶領域をもつ二分探索木で，1 ビット情報（赤と黒）は木のバランスを保つために用いられる．赤黒木はノードの色変更や回転操作により，常に色の制約を満たすように木構造を調節してバランスを保つ．赤黒木の色の制約を次のように定義する [4].

赤黒木の色の制約：

1. すべてのノードは赤または黒である．
2. ルートとすべての外部ノードは黒である．
3. 赤ノードの子は両方とも黒である．
4. あるノードからその子孫である外部ノードまでの単純な道はすべて同じ数の黒ノードを含む．

ノードの挿入や削除により制約条件３または４に合わない場合が生じ，制約を満足するまで木構造を調節する．

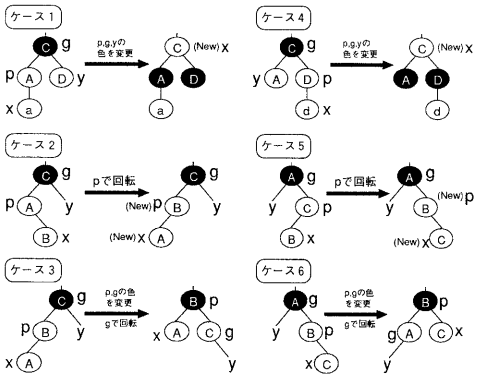


図 2: 挿入後の調節処理

ノードの挿入： 二分探索木の挿入と同じ手続きでノードを挿入する．挿入したノードは赤とする．挿入ノードの親が赤のとき，制約条件３に違反するので，ノードの色変更や回転によって木構造を調節する．挿入ノードを x，x の親を p，p の兄弟を y，p の親を g とすると木の調節処理は

ケース１； x，p，y が赤のとき
ケース２； x，p が赤，y が黒で
　　x が p の右の子のとき
ケース３； x，p が赤，y が黒で，
　　x が p の左の子のとき

の３つのケースに分けられる（図２）．調節処理はケース１を０回以上繰り返した後，ケース２と３，またはケース３を実行し，ルートを黒にして処理を終る．図 2 におけるケース４から６は，それぞれケース１から３と対称的な処理である．

ノードの削除： 二分探索木の削除と同じ手続きで，ノードを削除する．赤ノードを削除すれば色制約を破ることはないが，黒ノードを削除したとき制約条件４が満たされなくなる．したがって黒ノードを削除したとき，ノードの削除処理後に木構造を調節する．調整処理は，削除したノードからルートまで木の下から上に向かって実行される．削除したノードの子を x，x の兄弟を w，w の左右の子をそれぞれ wl,wr とすると，木の調節処理は

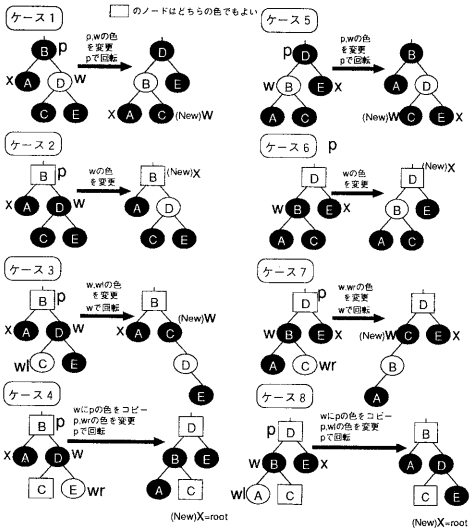


図 3: 削除後の調節処理

ケース１； w が赤のとき
ケース２； w，wl，wr が黒のとき
ケース３； w，wl が赤，wr が黒のとき
ケース４； w が黒，wr が赤のとき

の４つのケースに分けられる（図３）．調節処理ではケース２を０回以上実行した後，ケース１と２，ケース１と３と４，ケース１と４，ケース３と４，ケース４のいずれかのパターンを実行する．最後にノード x を黒にして調節処理を終る．図 3 におけるケース１から４は，それぞれケース５から８と対称的な処理である．

3 パーシステント木

一般に木構造に更新操作を施した場合，通常のデータ構造では操作前の状態は記憶せず，操作後の木構造だけを保持する．実行した更新操作の履歴や更新操作によってどのように木構造が変化してきたかという履歴は保持しない．更新操作を実行する前の木構造の状態を記憶して，更新操作回数分の木構造の状態を保持し，これまで発生したどの状態の木構造にもアクセス可能な木を**パーシステント木**とよぶ [1, 2]．パーシステント木への更新操作は，空の木から始まり，オンラインで実行される．

パーシステント木では，各更新操作や操作後の木構造の状態を明確に区別し，順序を一意に定めるために，各更新操作および木構造の状態にタイムスタンプを割り当てる．タイムスタンプは自然数で表す．初期状態である空の木をバージョン0とよぶ．空の木に対して実行する最初の操作（挿入）は時刻1の更新操作で，バージョン1を作成する．一般に時刻$i(i\geq 1)$の更新操作はバージョンiの木を作成する．

探索や走査は木の構造を変更しないので，新しいバージョンを発生しない．

パーシステント木が保持しているいくつかのバージョンの中で，最新バージョンだけに更新操作が可能な木を**部分パーシステント木**，すべてのバージョンに更新操作が可能な木を**完全パーシステント木**とよぶ[1, 2]．

部分パーシステント木では常にバージョン$i(i\geq 0)$からバージョン$i+1$を作成するので，バージョン番号は線形順序を保つ．

完全パーシステント木についてバージョンiからバージョンjを作成したとすると，iは$0\leq i<j$を満たす整数になる．完全パーシステント木のバージョン番号は半順序で並べられるので，iを親，jを子とする木で表わし，この木を**バージョン木**とよぶ．バージョン木は更新操作直前のバージョンと更新操作後のバージョンの関係を表わす（図4）．完全パーシステント木では，まったく同じ構造の木を異なったバージョンの木として保持する場合が生じる．したがって，バージョン番号の関係は木ではなくグラフとして表される．しかし，木の更新操作を実行するたびに木の構造をチェックすれば，バージョン番号は木で管理できる．

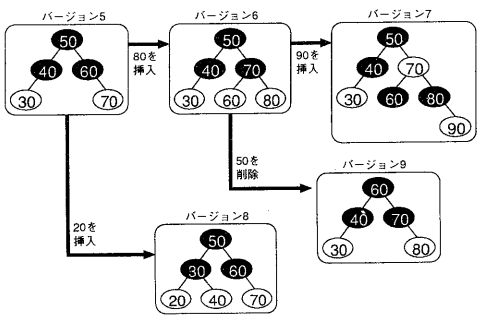


図4: 完全パーシステント木のバージョン木

完全パーシステント化することの利点： 木を完全パーシステント化することの利点のひとつは，更新操作ごとの木構造の状態を記憶しているので，木がどのように変化してきたかを理解しやすい．赤黒木の動作を理解するためには，赤黒木の色変更や回転による平衡化を逐次表示し，更新操作前後の木を比較できるデータを保持する必要がある．パーシステント木はこれを実現できるデータ構造である．

2番目の利点は，赤黒木の更新操作に対するアンドゥ機能が実現できることである．通常，アンドゥ機能は，実行した更新操作を取り消して，操作実行前の状態に戻るために使用される．完全パーシステント木では過去に作られたどのバージョンにも戻ることが可能で，更新操作に対するアンドゥ機能が実現できる．

4 パーシステント木の構築方法

各バージョンにおける赤黒木全体を記憶すると，記憶しなければならないデータ量が非常に多くなる．しかし，連続するバージョンの前後の赤黒木では一部の部分木がまったく同じ構造をしているので，変更された部分木だけを記憶すれば十分である．

変更された部分だけを記憶する方法として**パスコピー法**と**ファットノード法**がある[1, 2]．どちらの方法も，基本操作をノードの挿入とキーの探索に制限して考案されたパーシステント木であるが，本研究ではノードの削除も基本操作として考える．

パスコピー法： パスコピー法とは，挿入したノードや実際に削除したノードまでのパス上のノードだけを記憶していく方法である．実際には挿入または削除するキーを探索するときに訪れたノードをコピーする．コピーしたノードに対して挿入や削除を実行する．赤黒木の場合，更新したノードまでのパス上で回転操作やノードの色変更が実行されるので，パス上のノードだけでなくパス上のノードの子ノードもコピーしなければならない場合が生じる．

パスコピー法では各バージョンごとに異なったルートをもつ赤黒木を作成するが，一部の部分木は共有される．各バージョンのルートに

アクセスするための補助データ構造を必要とする．図 4 のパーシステント赤黒木をパスコ

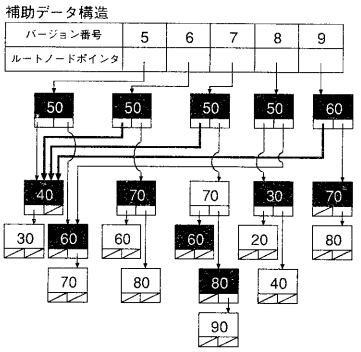


図 5: パスコピー法

ピー法で実現する場合のデータ構造を図 5 に示す．バージョン 5 のノード 40 をバージョン 6，7，9 のデータとして共有する．データが共有できるのは，その更新操作の対象となるバージョンと操作後発生するバージョンとの間であるので，バージョン 8 はバージョン 5 のデータを共有することになる．パスコピー法は各バージョンのルートを記憶するための補助データ構造を用意すれば，従来の赤黒木ノードと同じ構造体でパーシステント赤黒木ノードを表現できる．

ファットノード法： 黒ノードの子として新たにノードを挿入した場合，変更する部分は親ノードのポインタフィールドだけである．また，葉ノードである赤色のノードを削除した場合も親ノードのポインタフィールドを変更するだけでよい．これらの場合，パスコピー法のように操作するノードまでのパスをすべて変更する必要はない．各ノードのポインタフィールドを複数の値がもてるリストで表現すればノードに対する変更をノード自身の中に記憶できる．左右の子ノードを指すポインタやノードの色を記憶するフィールドを動的なリスト構造で表現し，変更するデータをノード内だけに記憶してノード自身を太らせる方法をファットノード法とよぶ．

図 4 のパーシステント赤黒木をファットノード法で実現する場合のデータ構造を図 6 に示す．ファットノード法では，各バージョンで記憶するデータ量がパスコピー法に比べかな

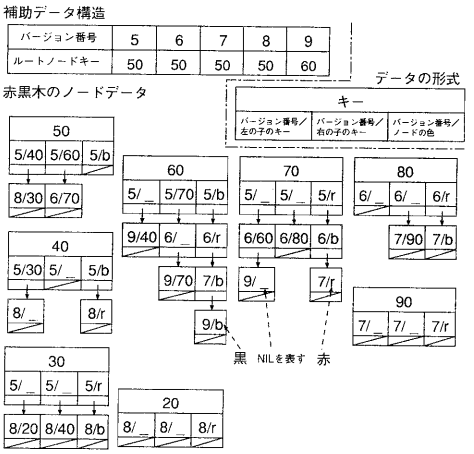


図 6: ファットノード法

り少なくなる．しかし，ノード自身が複数の値をもつので，適切な値を選択する探索が各ノードの各フィールドごとに必要である．したがって，カレントバージョンの木を構築するまでの時間が必要となり，新しい値を追加する処理も複雑になる．

5 パーシステント赤黒木のための対話システム

5.1 設計

本研究では赤黒木の平衡化の過程を逐次表示し，基本操作を対話的に実行できる対話型システムを作成する．赤黒木は更新操作ごとの木構造の状態を記憶している完全パーシステント赤黒木とし，バージョン番号は木で管理する．対話型システムは SICStus2.1 Prolog[8, 9] とグラフィクスマネージャ (GM)[11] を使って作成する．SICStus2.1 Prolog は DECsystem-10 Prolog や Quintus Prolog と互換性があり，X Window System とのインタフェースを確立している．GMは，X Window 環境でグラフィクスや対話を扱うパッケージで，SICStus2.1 Prolog から起動でき，完全に制御できる別プロセスである．GMは InterViews system [1]で書かれている．

[1]Copyright(c) 1987.1988.1989 Stanford University

パスコピー法とファットノード法でパーシステント赤黒木を構築する場合のノードデータの形式を次のように定める．

・パスコピー法

各ノードを１つの事実で表現し，事実を一意に定めるため識別子を割り当てる．同じキーをもつノードでもバージョンが違えば異なる識別子を割り当てる．データ形式を次に示す．

pnode(K,Clr,ID,[LID,RID],RtID,V).

K はノードがもつキーの値で，Clr はノードの色で赤を r 黒を b で表す．ID, LID, RID はそれぞれノード自体，左右の子ノードの識別子で，アトムで表す．ノードがあるバージョンのルートノードの場合，RtID にアトムを代入し，ルートでなければ何も代入されない．アトムはバージョンを一意に定めるための識別子である．RtID のフィールドを用意することによって，各バージョンのルートを示すための補助データ構造は不要になる．最後の引数 V はノードが作られたときのバージョン番号を表す．

・ファットノード法

ファットノード法は値の変更が必要なポインタフィールドだけを動的に保持し，そこに変更する値を記憶する方法である．ノードはただひとつのキーをもつので，ファットノード法では１つのキーに対して１つの事実でノードを表現する．ノードデータの形式を以下に示す．

fnode(K,[V/Lk,..],[V/Rk,..],[V/Clr,..]).

K はノードがもつキーで，３つの引数は複数の値がもてるようにリストで表現する．それぞれ左の子がもつキーの値，右の子がもつキーの値，ノードの色を表し，値をバージョン番号とともに記憶する．このデータ形式では各バージョンのルートを特定するために，補助データ構造を必要とする．

Prolog では，１つのノードにアクセスするにもあらゆるバージョンのすべてのノードデータの中から探索しなければならない．したがって，Pascal などのリンク構造をたどっていくよりもノードデータへのアクセスに時間がかかる．しかし，ノードデータを事実として定義すると，各バージョンで増えていくデータを明確に視覚化できる．また，ルートからリンク構造をたどって，各ノードにアクセスするだけでなく，組み込み述語 listing を使って直接特定のノードデータにアクセスすることができる．

5.2 実現

作成した対話システムの画面を図 7 に示す．赤黒木はパスコピー法によってパーシステント化し，更新操作や質問操作をコマンドとして実現した．図 7 ではカレントバージョン

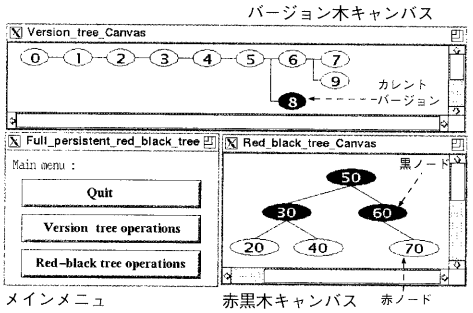


図 7: 対話システムの画面

は８で，赤黒木キャンバスはバージョン８の赤黒木である．表示されているカレントバージョンに対して赤黒木の操作を実行する．カレントバージョンはバージョン木キャンバス上のノードをマウスでクリックして変更する．赤黒木やバージョン木に対する操作はボタンコマンドとして用意され，メインメニュより下位の階層のメニュで実行される．

赤黒木の操作は，ノードの挿入と削除，およびキーの探索である．挿入や削除後の木の調節処理は逐次表示し，操作後の木がカレントバージョンになる．赤黒木への挿入や削除により，バージョン木のノードが一つ増える．探索コマンドには要求するキーの探索の他に，最大または最小キー，あるキーの直前または直後キーを探索するコマンドを用意した．探索が成功すれば探索ノードを強調表示する．

探索の他にノードに関する情報と実行した更新操作情報を表示するコマンドを用意してある．ノードに関する情報はシステムが保持している赤黒木のすべてのノードデータの中で，あるキーを含むノードのバージョン番号，色，左右の子ノードのキーで，テキスト形式で表示する．図 7 でキー 70 としてコマンドを実行する

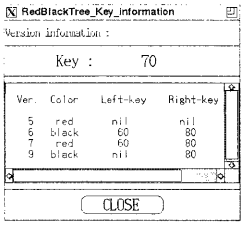


図 8: ノード情報の表示

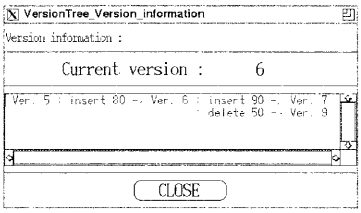
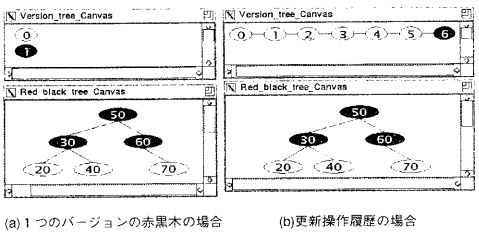


図 9: 更新操作情報の表示

と図 8 の情報を表示する．更新操作情報とはあるバージョンに実行した更新操作とキーである．図 7 でカレントバージョンを 6 に変更してこのコマンドを実行すると図 9 のような情報を表示する．



(a) 1 つのバージョンの赤黒木の場合　　(b)更新操作履歴の場合

図 10: 赤黒木データのロード

作成した対話システムでは，赤黒木のデータをファイルに書き込んだりファイルから読み込むためのコマンドを 2 種類用意してある．1 つのバージョンの赤黒木の場合と更新操作履歴の場合がある．図 7 で 2 種類のデータセーブを実行すると，それぞれ別の拡張子をつけたファイルにデータを書き込む．システムを一旦終了し再度起動してそれぞれのファイルをロードすると図 10 のようになる．図 10 (a) のロードは赤黒木の更新操作と同様にバージョン番号を割り当てる．このロードは小さなバージョン木で多くのノードを含む赤黒木を扱うことが目的で作成した．図 10 (b) のロードは空の木から行なった更新操作を順に実行してパーシステント赤黒木を復元する．

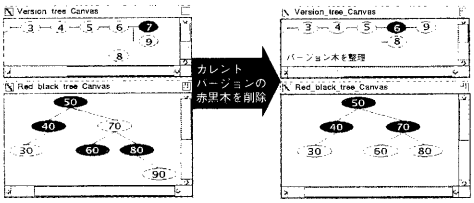


図 11: バージョン木ノードの削除

不要になった赤黒木データを削除するコマンドを用意してある．カレントバージョンがバージョン木の葉ノードであれば，カレントバージョンの赤黒木データを削除し，バージョン木のノードを削除する．図 11 にバージョン木を再構成する様子をに示す．

6 検討

作成した対話システムの問題点および拡張項目を次に示す．

- 赤黒木の表示ウィンドウを複数にする．
- ノード情報や更新操作情報をテキストでなく木で表現する．
- 赤黒木の回転操作を表示する場合，ノードを一旦削除してから再描画しているが，アニメーション的に移動して回転の様子を明示的に視覚化する．
- バージョン木でバージョン番号を管理すると，同じリンク構造をしている赤黒木でも別のバージョンとして保持する可能性がある．ノードの挿入や削除を実行して新しいバージョンの赤黒木を作成したとき，それまでに保持しているバージョンの赤黒木とまったく同じリンク構造をしているかどうかをチェックする．

キー探索： 赤黒木でキーを探索する場合，ルートからリンク構造をたどって対応するキーをもつノードを捜し出す．パスコピー法によるパーシステント赤黒木ではノードデータを事実として記憶するため，リンク構造を使用しないで Prolog への質問でキーを捜し出すことができる．

ノード情報表示コマンド： パスコピー法では組み込み述語 listing により複数のノードデータ pnode にアクセスして表示する情報を作成する．各ノードのバージョン番号を表示するために，ノードデータはバージョン番号のフィールドをもつ．ファットノード法では1つのノードデータ fnode が複数のバージョンのデータを含んでいるので，1つの fnode だけにアクセスすれば表示情報を得る．

バージョン木ノード削除コマンド： 赤黒木のノードデータが減ることによりノードデータへのアクセス時間の短縮，バージョン木の整理，赤黒木の更新操作に対するアンドゥの実現を目的に用意したコマンドである．

パスコピー法では，各ノードデータのバージョン番号フィールドを用いて組み込み述語 retract によって赤黒木データを削除することができる．ファットノード法では，ノードデータの各フィールドをリストで表現しているので，各リストで線形探索を行い，要求されたバージョン番号をもつデータを削除する．

更新操作情報表示コマンド： 表示する更新操作情報は各バージョンの赤黒木に前順走査により生成するキーのリストから求める．図9の場合に生成したリストとキーの和は次のようになる．

バージョン	キーの前順リスト	和
5	[50,40,30,60,70]	250
6	[50,40,30,70,60,80]	330
7	[50,40,30,70,60,80,90]	420
9	[60,40,30,70,80]	280

前後のバージョン間でリストの長さから挿入または削除を判断し，キーの和の差をとって操作したキーを求める．パスコピー法，ファットノード法とも同じ方法で表示する情報を作成する．

パスコピー法の場合，パーシステント化した木が2分探索木ならば，更新操作後新たに追加したデータであるパスと，その前のバージョンの同じパスを比較するだけで更新操作やキーを求めることができる．赤黒木では更新操作後に回転するので，パスを調べるだけでは更新操作やキーを特定することができない．前順リストを比較する方法では，ノードすべてにアクセスしてリストを生成しているが，パスにアクセスするだけで表示情報を得ることが可能になる．

更新操作履歴のセーブおよびロードコマンド： ファイルに書き込むデータは，上記の更新操作情報表示コマンドと同様に，前順走査によるリストから求めた更新操作と操作キーである．図 10 (b) で読み込んだデータは，

```
[60,ins,50,ins,40,ins,
      30,ins,70,ins,20,ins].
```

である．このデータをロードする場合，リストの先頭から順に操作を繰り返し実行する．

ファットノード法では，読み書きするデータの単位をノードデータとし，バージョン番号を修正してコピーすることによりロードが可能である．ファットノード法のノードデータは，1つのキーに関する事実が複数のバージョンのデータを保持しているので，ロードにかかる処理時間が大幅に短縮できる．しかし，セーブするときにバージョン番号を置き換えたり，どのキーのデータだけをコピーするかを求めるのに木の走査を必要とする．更新操作履歴をデータの単位とすれば実現しやすく，木の動作も視覚化しやすい．パスコピー法では識別子を置き換えるよりも簡単な処理で赤黒木が復元できる．

赤黒木データのセーブおよびロードコマンド： カレントバージョンの赤黒木データをセーブおよびロードする．ファイルに書き込むデータは前順走査により生成するキーのリストでリストの要素はキーの値とノードの色である．図 10 (a) で読み込んだデータは，

```
[ 50/b,30/b,20/r,40/r,60/b,70/r ].
```

である．このデータをロードする場合，リストの先頭から順に挿入する．挿入は赤黒木への挿入ではなく，色変更や回転をともなわない二分探索木の挿入である．パスコピー法では識別子を置き換えるよりも簡単な処理で赤黒木が復元できる．

ファットノード法のデータ形式では識別子がないので，セーブする場合にノードデータをそのままコピーしてもデータをロードできる．セーブする場合，赤黒木のノードすべてにアクセスしてコピーするキーの値を調べなければな

らない．ロードする場合，挿入処理より単純にデータをコピーする方が速く読み込むことができる．

ファットノード法： ノードデータの各フィールドをリストで表現しているので，挿入や削除，色の変更や回転，描画の各ルーチンがノードデータにアクセスするたびに同じ値を見つけるための線形探索を繰り返す．次に，各ルーチンがアクセスするノードデータとして，通常の赤黒木データと同じ形式のデータを用意する．リストで表したデータに対してはフィールドの値が変わるときだけアクセスする．ファットノード法は記憶領域をできる限り少なくする場合に適した実現方法である．

パスコピー法： パスコピー法でパーシステント赤黒木を構築するために，通常の赤黒木のノードデータおよび操作ルーチンの変更点を次に示す．

- バージョンごとにルートが異なるので，それぞれのルートを特定するための補助データ構造を用意する．
- 各ノードのバージョン番号を記憶するフィールドを用意する．
- 挿入または削除の操作ルーチンで，キーの探索で訪れたノードをコピーする．コピーしたノードデータに対して挿入や削除を実行する．
- 挿入や削除後の色変更や回転ルーチンで，カレントバージョン以前のノードデータを更新する場合ノードデータをコピーして変更する．

質問操作や木の描画ルーチンはまったく変更する必要がない．以上から，比較的簡単な変更でパーシステント赤黒木を構築することができる．しかし，パスコピー法では，各ノードデータに親を指すポインタフィールドがもてないこと，および1つのノードを挿入または削除しただけの変更にしてはノードデータが増えるという欠点がある．

木のリンク構造のチェック： まったく同じキーをもち，ノードのリンク構造も同じ赤黒木を違うバージョン番号のデータとして保持しているかをチェックする場合，新しいバージョンの赤黒木を作成するたびにチェックするオンラインチェックと，ボタンコマンドとして用意するオフラインチェックの2つの方法がある．

オンラインチェックでは，新しいバージョンの赤黒木を作成するとき，すなわちノードの挿入または削除を実行するたびに，システムが保持しているすべてのバージョンの赤黒木と作成するバージョンの赤黒木を比較し，同じ場合は新しいバージョンを生成しない．オンラインチェックによりバージョン番号は木で管理できる．

オフラインチェックではコマンドを実行した場合にチェックする．すでにシステムが保持しているデータなので，バージョン木の番号を置き換えたり，バージョン木の構成を更新する必要がある．しかし，システム内部の赤黒木のノードデータをどう変更するかなど問題点も多く，実現が困難である．

Prologによる二分木の表現： 一般にPrologでは，二分木を項表現とリスト表現の2つの方法で表現する．たとえば，図 7 のバージョン5の赤黒木について；

項表現：

```
t(t(t(n,30,n),40,n),50,t(n,60,t(n,70,n))).
```

リスト表現：

```
[[[n,30,n],40,n],50,[n,60,[n,70,n]]].
```

と表わす．n は外部ノードを意味する．

パスコピー法：

各バージョンで変更の必要なノードだけを記憶するので，ノードごとのデータで木を表現しなければならない．そこで各ノードを1つの事実（項）で表現した．このデータ形式では木のリンク構造をたどってノードを参照する処理がすべてのノードデータの中からの探索になる．しかし，どのノードデータもリンク構造をたどらずに直接アクセスできるので，ノード情報やバージョンノードの削除が実現しやすくなった．

ファットノード法：

各ノードを1つの事実で表現し，パスコピー法との比較を試みた．ファットノード法では，一般的な木構造表現のように，木全体を1つの項とするデータ形式でもパーシステント木を構築できる．たとえば，10,20,30,40 の順に挿入した場合，ファットノード法による木のデータは

```
[[[],[3/10],[]],[1/10,3/20],
  [[],[2/20,3/30],[[],[4/40],[]]]].
```

と表すことができる．外部ノードは空リスト [] で表現している．

以上のように，Prologではさまざまな表現方法が可能であるので，パーシステント木の実現方法に合うデータの形式を考え，システムを作成しなければならない．

7 おわりに

本稿ではパーシステント赤黒木を操作する対話システムを作成した．このシステムで次のことを実現した．

- 赤黒木の基本操作のコマンドを用意して赤黒木の対話的操作システムを実現した．
- 挿入や削除後のノードの色変更および回転の平衡化の過程を逐次表示して赤黒木の構造変化動作を視覚化した．
- 木構造の変化の履歴を保持するパーシステント赤黒木を実現した．

木のパーシステント化に基づき実現できる機能，および今後追加すべき処理を示した．

- それぞれのバージョンについてあるキーをもつノード情報の表示，および実行した更新操作情報を木構造から求めて表示する機能を実現した．
- オンラインの更新操作を保持するだけでなく，操作履歴および赤黒木のデータをセーブおよびロードするコマンドを作成した．
- 赤黒木に対する更新操作のアンドゥ機能を実現するために，バージョン木ノードの削除コマンドを作成した．

また，Prologで作成することの有効性や問題を指摘した．

参考文献

[1] J.R.Driscoll, N.Sarnak, D.D.Sleator, R.E.Tarjan: Making Data Structures Persistent, *Annual IEEE Symp. on Foundations of Computer Science*, pp. 109 – 121, (1986).

[2] N.Sarnak, R.E.Tarjan: Planar Point Location Using Persistent Search Trees, *Comm. ACM*, Vol.29, pp. 669–679, (1986).

[3] M.H.Overmars: Searching in the past II: general transforms, Technical Report RUU-CS-81-9, Department of Computer Science, University of Utrecht, Utrecht, The Netherlands,(1981).

[4] T.H.Cormen, C.E.Leiserson, R.L.Rivest: Introduction to Algorithms, MIT Press, pp.244 – 280, (1990).

[5] 石畑清: アルゴリズムとデータ構造, 岩波書店, pp.73–86, (1989).

[6] 平田富夫: アルゴリズムとデータ構造, 森北出版, pp.38–44, (1990).

[7] R. セジヴィック (野下浩平 他訳): アルゴリズム 第2巻＝探索・文字列・計算幾何, 近代科学社, pp.30-39, (1992).

[8] Bratko,I.: PROLOG PROGRAMMING FOR ARTIFICAL INTELLIGENCE 2nd ed., Addison-Wesley Publishing Company,(1990) （安部憲広訳,Prologへの入門, 近代科学社,(1990)）.

[9] 小川束: Prologによる論理プログラミング入門, 啓学出版,(1990).

[10] SICStus Prolog User's Manual, Swedish Institute of Computer Science,(1991).

[11] SICStus Prolog Library Manual, Swedish Institute of Computer Science,(1991).

[12] R.E.Tarjan: Algorithm Design, *CACM* Vol.30, No.3, pp.204–212,Mar.,(1987), （山田眞市訳, 1986年度ACMチューリング賞授賞記念講演 アルゴリズムの設計, bit, Vol.19, No.11, Oct.,(1987)）.

[13] 別所正隆, 今宮淳美: Persistent treeを用いる二分探索木の対話的視覚化, 山梨大学計算機科学修士論文, (1993).