修　士　論　文

効率良く図形を描画するための
データ構造に関する研究

指導教官 浅野 哲夫 教授

審査委員主査　浅野 哲夫 教授
審査委員　　宮地 充子 助教授
審査委員　　平石 邦彦 助教授

北陸先端科学技術大学院大学
情報科学研究科 情報処理学専攻
学生番号 910003

天野 隆

2001 年 2 月 15 日

要 旨

図形を描画するための CAD などのツールでは, 描かれた幾何データを蓄積する際に, 図形が入力または削除される毎にデータの更新を行っている. そのためデータの更新が頻繁に行われることになり, 幾何データを膨大に扱う場合には, 操作の応答が遅れるといった問題が生じる.

本稿では, 人間が図形を入力する際にできる思考時間を有効に利用することを考える. その思考時間の間にデータを一度にまとめて更新する “**グループ更新**” を行うデータ構造を提案し, 効率良く図形を描画できるようにする.

目 次

図 目 次

表 目 次

第 1 章

はじめに

図形を描画する際に, 計算機の処理待ちがあっては使用者にとっては非常に使いづらいものになってしまう. 本研究では人間が図形を描画するときにできる思考時間を利用し, その間にデータの更新を行うことを考える. 従来のように更新操作をする毎にデータの更新を行うのではなく, ある程度更新操作をためておき思考時間の間に一度にデータの更新を行うデータ構造を実装し, 評価を行う.

1.1 本研究の背景と目的

計算幾何学では, 大規模な問題を想定して理論や手法の研究が進められている. 大規模なデータを扱う需要が高まれば, それを支える技術が要求される. いくら計算機の能力が高まったからといって, 記憶装置のより効率的な利用法とより高速な算法の技術の重要さが減ることはない. そこで, データ数が膨大であっても, 効率よく探索などが行えるアルゴリズムやデータ構造が求められている. 膨大なデータを扱う場合重要になってくるのが, 効率の良さである. 文書データベースや WWW 検索エンジンなどでは, 膨大なテキストデータを扱うため効率の良いアルゴリズム [4][5] が必要になる.

幾何データを扱う問題を解く場合において計算機の果たす役割は非常に大きく, 幾何学的な図形に係わる問題を計算機で効率的に処理する必要性が近年ますます高まってきている. 幾何データを管理するデータ構造についての研究は盛んに行われており, さまざまなデータ構造が提案されている. 代表的なデータ構造として AVL 木 [1] や区分木 [2] などがある.

探索の効率化を考慮する問題を特に探索問題という. さらに, データ集合が挿入や削除で更新される探索問題を動的な探索問題といい, そうでない場合を静的な探索問題という.

CAD などのデータ構造は挿入および削除を行いデータの更新を行うことから, 動的な探索問題であるといえる.

本研究の特色としては, 人間が CAD などのツールを利用するときにできる思考時間を有効に利用し, そのデータをグループ毎に更新するところにある. この思考時間を利用し効率よく図形を描画するためのデータ構造の構築を行う. CAD などのツールは膨大な幾何データを扱うとともに, 必要なデータを瞬時に挿入, 削除および選択できなければならない. そのためには適したアルゴリズムとデータ構造が必要となる. また, 利用者がコンピュータの処理時間に不快感を感じないことが重要である. 現在, CAD などのデータ構造には AVL 木に代表される平衡二分探索木を使うのが一般的である. AVL 木では, データ構造の更新を 1 つのデータの挿入や削除ごとに行う. 幾何データを膨大に扱い, データの更新が頻繁に行われる場合には, 処理に時間がかかり応答が遅れてしまうといった問題が生じる. そのため, 多数の更新をグループにまとめて, 一度に挿入や削除を行う手法 [7] も必要である. この方法は, 緩和平衡木 [13] を用いた手法 [6] の他に, B-木 [14] を用いた手法 [3] もある. 一度にまとめてデータの更新を行う手法を “**グループ更新**” と呼ぶ.

人間が図形を描画する際には, 常にデータを入力し続けるわけではなく一般的に思考時間があるものである. 従来の考え方ではデータを挿入や削除するごとにデータの更新を行っていたが, 多数の操作をグループにまとめて挿入, 削除する方法も必要である.

この研究では, 人間が図形を描画する際にできる思考時間 (入力待ち時間) を有効に利用しデータをグループごとに更新し, 選択, 削除および挿入などの処理に素早く対応できる実用的な意味でのアルゴリズムとデータ構造を提案する.

人間が図形を描画するときには, 連続して入力をするわけではなく一般的に描こうとする図形を頭に思い描いてから入力を行う. 計算機では, その思考時間の間は図形が入力されない入力待ち状態になる. つまり人間による図形の描画は, 描こうとする図形の思考時間と実際に図形を入力する時間に分けることができる. その 2 つの処理を繰り返すことで図形を描画していく. この人間による入力の特性から生まれる計算機の待ち時間を有効に利用するのである. 待ち時間の間にデータ更新の処理を行うようにすれば, 図形の描画に関する操作がスムーズ行われ, 操作に対する計算機の応答も瞬時に行われることが期待できる. 思考時間の間に計算機で処理を行うといっても, 最低でも今現在の描画処理は行わなければならない. そうでないと, どこにどんな図形を描画しているのか分からなくなってしまうからである. そこで, 本研究では入力時に行われる最低限の描画処理はすぐに処理し, 時間がかかってしまうデータ構造の更新を後に行うことで効率良く図形を描画しようというわけである.

その手法のデータ構造として, データを蓄積する部分を 3 つ用意する. 1 つはメインと

なって全体のデータを蓄積する部分で, 残りの 2 つは一時的に更新するデータを蓄積しておく部分である. 挿入されるデータを蓄積する部分と, 削除されるデータを蓄積する部分である. 人間が図形を描画しているときに挿入されるデータ, または削除されるデータは一時的に蓄積しておく部分に保存する. 次に人間が思考時間をとっている間に, 今まで一時的に蓄積されたデータをメインのデータ蓄積部分に挿入, またはメインのデータからデータを削除する. 従来のように蓄積部分が 1 つである場合には, そこにすべてのデータが蓄積されることになり, データの更新に時間がかかってしまうことになる. 今回の手法は, 今現在の更新操作だけを一時的に蓄積するのであり, メインのデータに対しての更新操作は後で行うので, 今現在のデータの更新にかかる手間は軽減されることになる.

グループ更新は, 一時的に蓄積された更新操作を従来のように単純な手法でメインのデータ構造に反映するのではなく, 単純な手法よりも効率良く更新を行えるように工夫を凝らす. 単純な手法とは, データを 1 つずつメインに挿入, または削除することである. グループ更新を効率よく行うために必要なデータ構造として, 平衡探索木である二色木 [8] の平衡条件を緩和した緩和二色木 [9][10][11] を用いる. 従来のデータ構造と比較するために AVL 木と二色木の他に, 平衡探索木と同程度の能率をもつスキップリスト [12] も用いる.

1.2 本論文の構成

本論文の構成は以下の通りである. 第 2 章では, 従来からある平衡探索木の AVL 木と二色木, また平衡探索木と同程度の能率をもつスキップリストについて述べ, 第 3 章では, 本研究の主要素であるグループ更新について述べる. 第 4 章では, そのデータ構造を実装したツールの紹介と, 従来の手法との比較実験の結果を示し考察する. 第 5 章では, 実験結果の評価, 本研究のまとめ及び今後の研究課題について述べる.

第 2章

図形の描画に利用されるデータ構造

この章では, 図形の描画に利用される従来からある平衡探索木の AVL 木と二色木, 平衡探索木と同程度の能率をもつスキップリストについて説明する.

2.1 アルゴリズムとデータ構造の評価

アルゴリズムとデータ構造とは何か, その評価はどのように行うのかここで簡単に説明する.

計算機を用いてデータの処理を行うとき, どのようにデータを蓄えておくかによって処理の効率は大きく変化する. このデータを蓄えるための構造をデータ構造という. データ構造には配列, リスト, スタック, キュー, ヒープや木などがあり今も多くの研究者により効率のよいデータ構造が研究されている.

アルゴリズムとは, 簡単にいうと, ある値または値の集合を入力とし, ある値または値の集合を出力する明確に定義された計算手続きのことである. したがって, アルゴリズムは入力を出力に変換する計算のステップの系列でる. 基本的なアルゴリズムとして分割統治法, 動的計画法, 貪欲法や再帰法などがある.

アルゴリズムの評価基準は, ある意味では単純である. 速く解を求めることができるほど, また少ないメモリ領域で計算できるほど, 優れたアルゴリズムであるといえる. アルゴリズムの良し悪しを評価する基準として最も重要なのは, それぞれのアルゴリズムが答えを出すまでにどの程度の計算時間 (手間) を必要とするかである.

まず, 問題の大きさを表すパラメータを設定し, それの関数として計算量を表すのが一般的である. このパラメータを n で表すとする. n に関して最も速く増加する項 (主要項) だけを残して, 他の項は無視してよい. n が大きくなったときの計算量を決めるのはこの

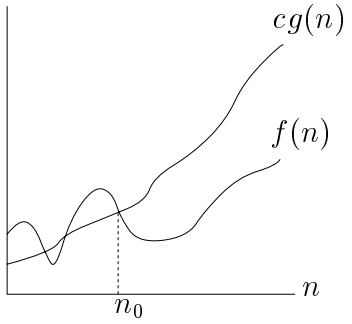


図 2.1: $f(n) = O(g(n))$

項だからである. 主要項だけを考えることにすると, すべての実行回数を数える必要がなく, 最も実行回数の多い操作操作だけを考えればよいことになる. 単純なプログラムで考えると, 定数回実行される文は無視し, ループ処理について考えればよいということである. 計算量の主要項以外を無視するということは, $n \to \infty$ のときの計算量の振る舞いを調べるということである. これを計算量の漸近的な振る舞いという.

漸近的な計算量の議論を容易にするために, O 記法を用いる. パラメータ n についての関数 $f(n)$ に対して, ある一定の値 n_0 と正の定数 c があり $n \geq n_0$ を満たすすべての n に対して $f(n) \leq cg(n)$ となるとき,

$$f(n) = O(g(n))$$

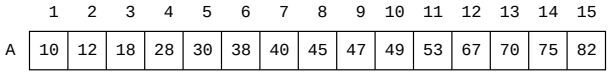
と表記する (図 2.1).

$f(n) \leq cg(n)$ でありさえすればよいのだから, 線形時間の探索は $O(2n)$ でも, $O(n^2)$ でもあるのだが, 普通このような表し方はしない. 対象とする関数にできるだけ近い, しかもなるべく簡単な形 $O(n)$ を採用する.

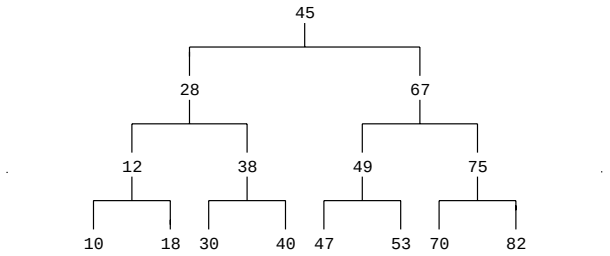
今までに述べたようにアルゴリズムの解析では, 主要項以外の項を無視し主要項の係数も無視する. O 記法による表現はちょうどこれらの要素を無視した肝心な部分だけを取り出していることにあたる. O 記法がアルゴリズムの議論に適しているのは, このためである. O 記法はアルゴリズムがこれ以上遅くならないという意味で, 計算量の上限を与える.

2.2 二分探索法

整数の異なった値をもつ要素からなる集合 X に関する探索の操作 member について考える.



(a) 配列に昇順に蓄えられたデータ



(b) 上の配列に関する二分探索の様子を表す図式

図 2.2: 二分探索の図的表現

- member(x,X)：x が X に含まれるかどうかを判定する.

固定された集合 X に関する member を高速に行う手法である二分探索法について述べる. データ構造としては集合 X の n 個の要素をその値の順にソートして, 小さい順に配列 A で $A[1]$ から $A[n]$ まで覚えておくだけである (図 2.2). このとき, member(x,X) は次のようにして行える.

1. $i := 1,\ j := n$.
2. もし $i > j$ なら $x \notin X$ と答えて停止する.
3. $k := \lfloor (i+j)/2 \rfloor$. もし $A[k] = x$ なら $x \in X$ と答えて停止する.
   もし $A[k] < x$ なら $i := k+1$ として 2 へ戻る.
   もし $A[k] > x$ なら $j := k-1$ として 2 へ戻る.

このアルゴリズムの正当性は, $i \le k \le j$ なる k に対して $A[k] < x$ ($A[k] > x$) なら $A[1] < \cdots < A[n]$ の性質より $A[i]$ から $A[k]$($A[k]$ から $A[j]$) の間には x がないことがわかることにより示される. また容易にわかるように, 二分探索法は高々 $\lceil \log(n+1) \rceil$ 回の比較しか必要としない. すなわち, 二分探索法は固定した集合に対して $O(\log n)$ の手間で member を行う.

2.3 辞書

CAD などのツールを使って図形を描画する際に必要な操作は, 次の 3 つが基本となると考えられる. 図形の入力, 図形の選択, そして図形の削除である. 基本操作では新たなデータの追加や既存のデータの削除を行うことにより, 蓄積されたデータが更新される. データが更新されるデータ構造のことを動的なデータ構造であると呼ぶ. そしてこのような動的な操作を伴うものは辞書と呼ばれ, 要求される操作は以下の 3 つである.

(1) 探索

(2) 挿入

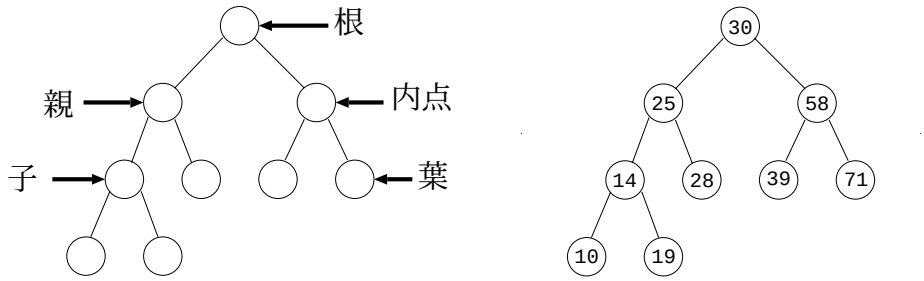
(3) 削除

辞書を実現するもっとも簡単な方法は, 一次元配列を用いて, これにデータを昇順 (または降順) に蓄えるというものである. このとき, 探索には二分探索が使えるから, 辞書に n 個のデータが蓄えられているとき, 探索には $O(\log n)$ の時間しかかからない.

単純な配列で辞書を実現するときに問題になるのは, 新たなデータの挿入とデータの削除である. このデータ構造ではデータが配列にすき間なく並べられているから, 新たな要素を挿入するときは, 挿入場所を確保するために, ソート順を崩さないように要素を移動する必要がある. このとき, 挿入すべきデータがたまたま辞書に蓄えられている最大要素よりも大きい場合には, 配列の最後尾に加えるだけでよいから, データの移動は生じないが, 逆に辞書の最小要素よりも小さいデータを挿入する場合には, すべての要素を移動しなければならない. 同様のことが削除のときにもいえる. したがって, データの挿入と削除に必要な時間は, 最悪の場合 $O(n)$ となってしまい非常に効率が悪い. そのため, 平衡二分探索木によって辞書が実現されることが多い.

2.4 二分探索木

木は閉路をもたない連結無向グラフである. 木では頂点を節点, 辺を枝と呼ぶ. 根と呼ばれる 1 個の特別視された節点をもつ木を根付き木という. 根から他の節点への単純なパスは 1 本だけ存在するが, そのパス上で節点 v の直前の節点 w を v の親といい, $w = p(v)$ と表す. v を w の子という. 同じ親をもつ節点は兄弟と呼ばれる. 親の親である節点は祖父, 子の子である節点は孫と呼ばれる. 根から節点 v へのパス上の節点を v の先祖という. 節点 v を先祖とするような節点 w は v の子孫と呼ばれる. 子をもつ節点は内点, 子をもたない節点は外点あるいは葉と呼ばれる. 根付き木での節点 v の深さ $\mathrm{depth}(v)$ は,



(a) 二分木　　(b) 二分探索木

図 2.3: 木の構造

$$\mathrm{depth}(v) = \begin{cases} 0 & (v \text{ が根のとき}) \\ \mathrm{depth}(p(v)) + 1 & (\text{それ以外のとき}) \end{cases}$$

と定義される. また, 高さ height は,

$$\mathrm{height}(v) = \begin{cases} 0 & (v \text{ が外点のとき}) \\ \max\{\mathrm{height}(w) | w \text{ は } v \text{ の子 }\} + 1 & (\text{それ以外のとき}) \end{cases}$$

と定義され, 根の高さを木の高さという. 深さ最大となる節点の深さを木の深さという. 木の深さと高さは一致する. 通常, 根を一番上にして図示される (図 2.3).

二分木は再帰的に定義される. 二分木 T とは, 次のいずれかの条件を満足する木である.

- T は節点を全くもっていない.
- T は根, 左部分木と呼ばれる二分木, 並びに右部分木と呼ばれる二分木の 3 つの節点集合 (共通要素をもたない) から構成される.

節点をもたない二分木は空木, あるいはヌル木 (NIL で表される) と呼ばれる. 左部分木が空でない場合, その根は木全体の根の左の子と呼ばれる. 右にもついても同様に右の子が定義される. 部分木がヌル木 NIL である場合, 子はいないといわれる. 二分木は単に高々2 の次数の節点からなる順序木ではない. たとえば, 二分木上である節点が 1 つの子供節点をもつ場合, その子供節点の位置, すなわち右か左のいずれかに位置するかが問題になる. 順序木ではこれは問題とならない. すべての節点で子供をまったくもたないか, または右と左の両方の子供をもつかどちらかの場合には, 全二分木と呼ばれる. 根から深さの小さ

い順に, また同じ深さのところでは左から右へ順番に頂点をできるだけつめて得られる二分木は完全二分木と呼ばれる.

二分探索木は, 各節点 v に割り当てられる要素を $key[v]$ で表すと, 任意の節点 v において, v の左の子の任意の子孫 u および v の右の子の任意の子孫 w に対して,

$$key[u] \leq key[v] < key[w]$$

が成立するように割り当てられ, 対称順に並べられているという. 探索木では子の個数が 1 つの節点があるとアルゴリズムが煩雑になるので通常は全二分木が用いられる. 葉にデータを蓄える木のことを葉優先木と呼ぶ.

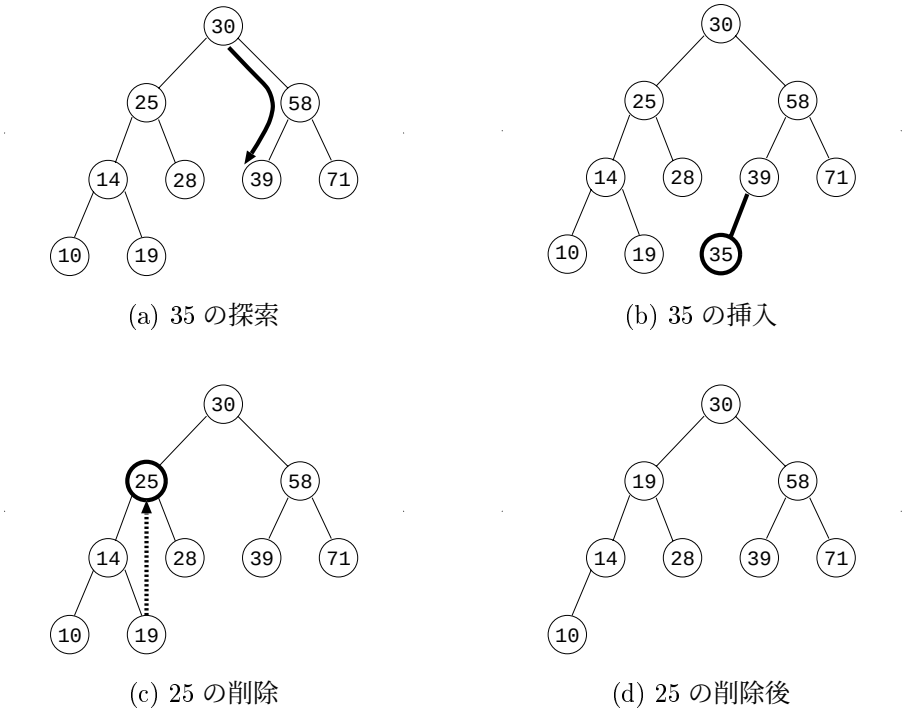
2.4.1 データの探索

二分探索木とは, 節点 v に蓄えられているデータを $key(v)$ と書くとすると, どの節点 v においてもその左部分木には $key(v)$ 以下のデータを蓄え, その右部分木には $key(v)$ より大きいデータを蓄えている. したがって, 根から始めて, 与えられたデータ x を現在訪れている節点のデータ $key(v)$ と比較し, $x \leq key(v)$ なら次の節点 v の左部分木を訪れ, $x > key(v)$ なら右部分木を訪れる. もちろん, 途中で $x = key(v)$ が成り立てば, 求めるデータが発見できたことがわかる. 葉に到達しても x に等しいデータが発見できなかったときは, x に等しいデータはこの二分探索木には含まれていないと結論づけることができる (図 2.4(a)).

二分探索木によって辞書を実現した場合, 3 つの操作はいずれも探索木の深さ (根から葉までのパス上の枝数) に比例する時間で実行することができる. もちろん, データが昇順に次々と挿入された場合, 枝分かれがないから探索木は一方向に延び, 探索木の深さが辞書のサイズに一致してしまう. このように, 最悪の場合にはどの操作にも $O(n)$ の時間がかかることになり, 単純な一次元配列を用いた場合よりも劣ることになる.

2.4.2 データの挿入

次に二分探索木にデータ x を挿入する. これは探索の場合とほぼ同様で, 根から葉に至るまで, x と節点に蓄えられたデータとの大小比較により左または右の子へとたどっていく. ただし, 同じデータが存在する場合は左の子をたどることにしておく (右の子をたどることにしてもよい). このようにたどり着いた葉が, このデータ x を蓄えるべき場所である. そこで, その節点を内点に変更し, そこにデータを蓄える (図 2.4(b)).



(a) 35 の探索　(b) 35 の挿入

(c) 25 の削除　(d) 25 の削除後

図 2.4: 二分探索木の操作

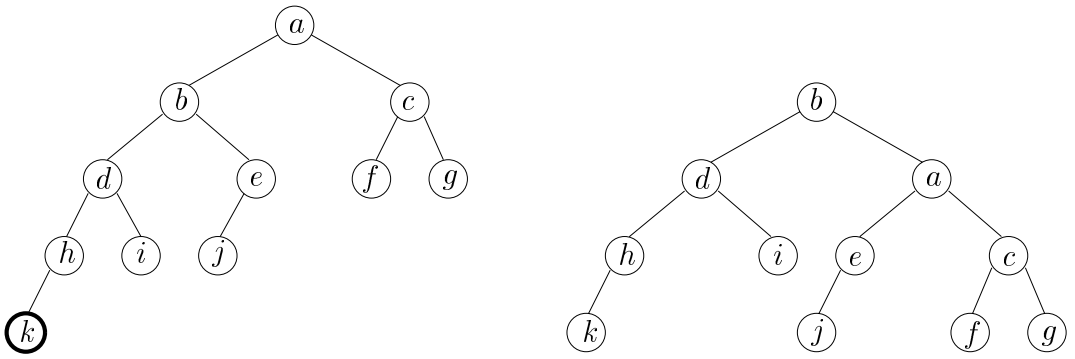
2.4.3　データの削除

データの削除は若干面倒である. 最初に探索と同様の方法で, 削除すべきデータを含む節点 v を見つける. このとき, 節点 v の少なくとも一方の子が葉ならば, 単に節点 v を他方の子で置き換えるだけでよい. 一方, 節点 v の子が両方とも内点である場合は, 節点 v の左部分木の最大データを求めて, これを節点 v まで移動する. そのために節点 v の左の子から始めて, 右の子がある限り右へ右へとたどる. このようにして節点 w を求めておいて右の子がなくなれば, 節点 w のデータが求める最大のデータであるから, これを節点 v に移動し, 節点 w の左部分木を節点 w の位置まで持ち上げる (図 2.4(c), (d)).

2.5　平衡二分探索木

先に述べたように, 単純な二分探索木の探索では最悪の場合 $O(n)$ の時間がかかってしまう. しかし, 木の各節点や葉がある程度調節されてそれほど差がないとき, すなわち $O(\log n)$ のときは, 探索の手間は $O(\log n)$ となる. このような深さが $O(\log n)$ の二分探索

木は, 平衡二分探索木と呼ばれる. 平衡二分探索木は木の深さが $O(\log n)$ に保たれるように平衡条件が存在し, その平衡条件により分類され, AVL 木や二色木などがある. いずれも, 挿入や削除によって平衡条件が満たされなくなることもあるので, そのときは平衡条件を保つようにバランスを復元しなければならない. その操作を再平衡化と呼ぶ. AVL 木や二色木は再平衡化が $O(\log n)$ で可能である. したがって, 探索, 挿入や削除の操作は $O(\log n)$ の時間で可能である. すなわち, 探索時間, 更新時間は $O(\log n)$ の時間で可能である. データ構造に必要な領域は $O(n)$ である. 次に, AVL 木と二色木について説明する.



(a) AVL 木のバランスを崩す入力　　(b) 再平衡化した後

図 2.5: AVL 木の再平衡化の例

2.5.1 AVL 木

AVL 木 [1] のデータ構造は先の二分探索木とほぼ同じであるが左右の部分木のバランスを監視することが重要であるので, 各節点において右部分木の深さから左部分木の深さを引いた値をその節点のバランス度として記憶する. 新たなデータの挿入は次のようにして行われる.

(1) 通常の二分探索木の場合と同様にして, 新たなデータを蓄えるべき節点 v を求める.

(2) 節点 v を内点として, ここに与えられたデータを蓄える.

(3) このデータの挿入によるバランス度の変化を計算する.

(4) 探索パスに沿って戻りながら, 各節点のバランス度を調べ, バランス度が崩れている場合には再平衡化を行う.

バランス度を各節点でもつ必要があるので, 節点を表す形式はその分だけ単純な二分探索木とは異なる.

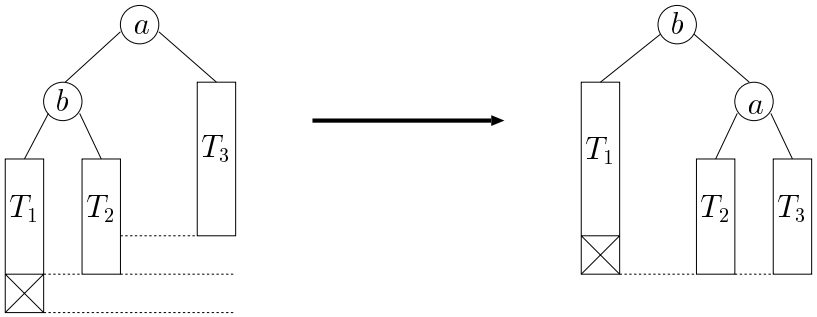
例えば, 図 2.5(a) に示す AVL 木にデータ k を挿入すると, 根において左右の部分木の差が 2 になってしまう. この場合は b を新たな根とし, b の左部分木はそのまま b の左部分木とし, b の右部分木には元の根 a の下に b の元の右部分木と a の右部分木を置く. そうすると図 2.5(b) に示すように左右のバランスがとれる. このような再平衡化の操作を単一 LL 回転というが, この他にも図 2.6 に示す単一回転と二重回転がある. この 2 種類の回転操作だけでバランスを保つことができる. ここで重要なことは, 1 個のデータの入力に関して再平衡化のための単一回転操作は高々2 回しか行われないことである. したがって, 回転操作を行っても極端に効率が悪くなることはない. 二重 LR 回転は単一 RR 回転の次に単一 LL 回転を施すことによって実現できる.

さて, 図 2.6 で 4 種類の回転操作を説明しているが, 自分の手で AVL 木の動作を確かめようとすると, いろいろとまぎらわしい場合が生じる. たとえば, 単一 LL 回転を示した図 2.6(a) では, T_1, T_2, T_3 という 3 個の部分木が現れる. 図 2.6(a) のように T_1 の方が T_2 より 1 レベルだけ深くなると確かに T_3 とのレベル差が 2 となり, バランスがくずれるが, T_2 が T_1 より 1 レベルだけ深くなっても同様の不均衡が生じる. これらの状況は互いに非常によく似ているが, 後者の場合には実際には単一 LL 回転ではなく, 二重 LR 回転を適用すべきである. 同様の状況が単一 RR 回転の場合にも生じる.

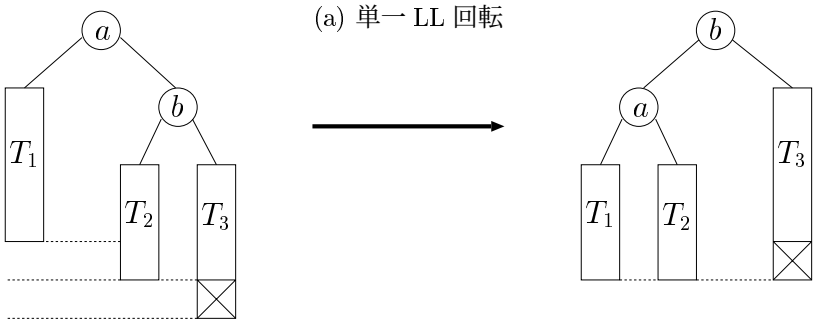
一方, 二重回転の場合はどうだろうか. たとえば二重 LR 回転を示した図 2.6(c) をみると, 4 個の部分木 T_1, T_2, T_3, T_4 が関係している. 図 2.6(c) では T_2 の方が T_3 よりも 1 レベルだけ深くなった状況を示しているが, T_3 が T_2 より 1 レベル深くなっても同様の不均衡が生じる. この場合, 回転後の状況を見ても, T_2 と T_3 は根からの深さが同じなので, どちらが 1 レベル深くなっても同じである. したがって, 二重 LR 回転は, 図 2.6(c) のパターンだけでなく, 図 2.7 に示したパターンについても適用可能である.

同様の回転操作は, データの削除によって不均衡が生じた場合にも必要になる. データの挿入の場合には最悪の場合でも単一回転操作を高々2 回だけ行えばよいが, 削除の場合には, 探索パスに沿った各節点において回転操作が必要になることがありうるが, 平均的には挿入の場合と同程度の回数ですむことが知られている.

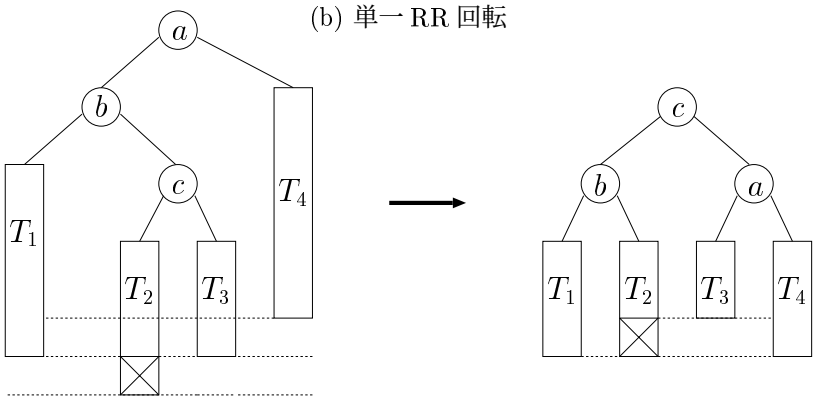
AVL 木のようなバランスのとれた二分探索木においては, 木の深さが節点数 n に対して $O(\log n)$ であることは確かであろう. 1 回の回転操作は一定時間でできるから, 探索に限らず, 挿入や削除の操作も最悪の場合でも $O(\log n)$ の時間で実行できる.



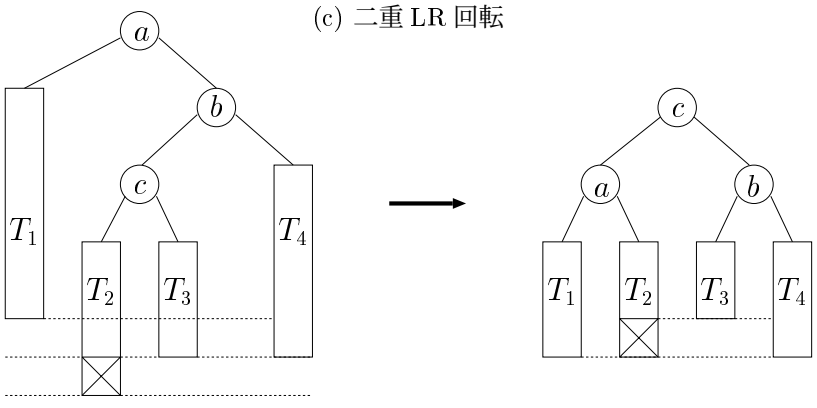
(a) 単一 LL 回転



(b) 単一 RR 回転



(c) 二重 LR 回転



(d) 二重 RL 回転

図 2.6: AVL 木の再平衡化のための回転操作

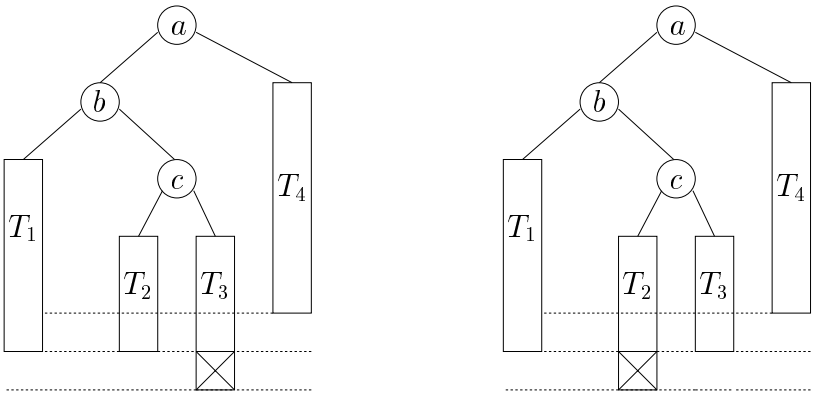


図 2.7: 二重 LR 回転が適用できるパターン

2.5.2 二色木

二色木 [8](図 2.8) とは, 次の平衡条件 (1)〜(4) を満たす全二分探索木である.

(1) すべての節点は赤か黒のいずれかである.

(2) 赤い節点は必ず黒い親をもつ.

(3) 根と葉は黒である.

(4) 根から葉へのパスはどのパスも同数個の黒い節点をもつ.

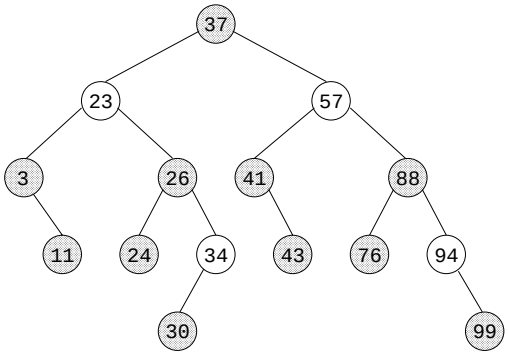


図 2.8: 二色木. 塗り潰された節点は黒, そうでない節点は赤を表す.

この平衡条件は根から葉までのパスで最短なもの (すべての節点が黒) と最長なもの (赤の節点と黒の節点が交互に現れるもの) とは長さが高々2 倍の違いしかないことを意味し

ている. したがって, 二色木深さは $O(\log n)$ となる. 実際, 二色木の葉の深さの最小値を c とすると木の深さ d は, $d \leq 2c$ となる. また c の定義より, 深さ $b < c$ の節点はいずれも内点で 2 個の子をもつので, 深さ b の内点の個数は 2^b となり, 二色木の内点の総数 n は $2^c - 1$ 以上となる. したがって, $d \leq 2c \leq 2\lfloor \log(n+1) \rfloor$ が得られる.

条件 (4) より, 節点 u から葉へのパスに含まれる黒い節点の個数はパスの選び方によらないので, $k(u)$ とおける. u が赤い節点ならば $\mathrm{rank}(u) = k(u)$ とし, u が黒い節点ならば $\mathrm{rank}(u) = k(u) - 1$ と定義すると, $\mathrm{rank}(u)$ は以下の条件 (a)〜(c) を満たす.

(a) u が親 $v = p(u)$ をもてば $\mathrm{rank}(u) \leq \mathrm{rank}\ (v) \leq \mathrm{rank}\ (u) + 1$.

(b) u が祖父 $w = p(p(u))$ をもてば, $\mathrm{rank}(u) < \mathrm{rank}\ (w)$.

(c) 葉 u は $\mathrm{rank}(u) = 0$ であり, 親 $v = p(u)$ は存在すれば $\mathrm{rank}(v) = 1$ である.

逆に rank が条件 (a)〜(c) を満たすとき, 節点 u が $\mathrm{rank}(p(u)) = \mathrm{rank}\ (u) + 1$ を満たすか根であるとき u を黒にし, それ以外なら赤にすると, 平衡条件 (1)〜(4) が成り立つことも容易にいえる. したがって, (a)〜(c) を二色木の平衡条件としてもよい.

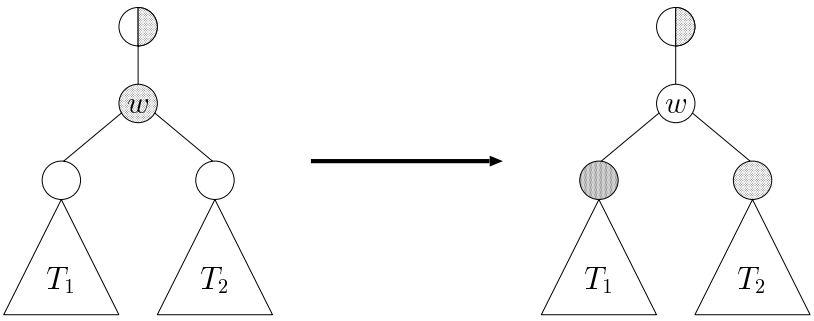


図 2.9: 挿入時の色変換.

二色木の内点のデータの並びや, 探索の操作は通常の二分探索木と同様に行われる. 挿入や削除も同様であるが, 平衡条件がくずれることもあるので再平衡化が必要である. 再平衡化で基本となる操作は, 色変換 (図 2.9) と回転である. 回転操作は単一回転と二重回転があり, AVL 木で用いたものと同じ操作を用いることができる. 回転後もデータは対称順に記憶されていることに注意されたい.

挿入後の再平衡化は, 根に向かって以下のように行われる. 挿入により新たに内点となった節点 u は, 条件 (4) により根でない限り赤にしなければならない (u を黒にすると根から u を通って葉へ至るパスが黒い節点を 1 個多く含むようになってしまう). こうすると, u

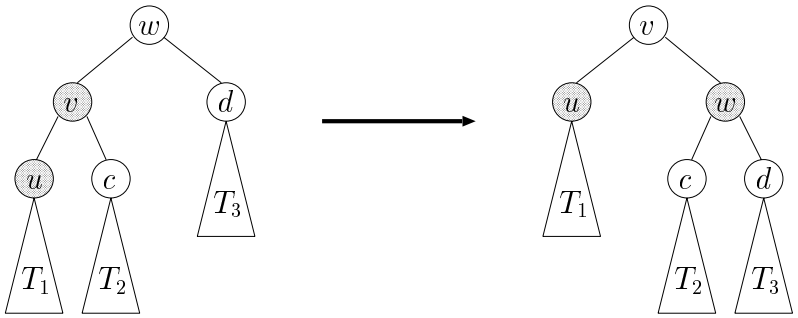


図 2.10: 挿入時の単一回転.

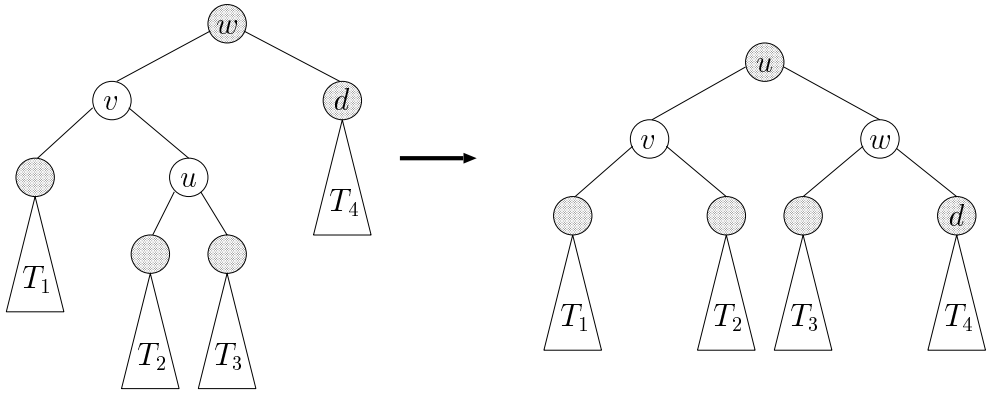


図 2.11: 挿入時の二重回転.

の親 $p(u)$ が黒ならば問題ないが, 赤だと条件 (2) が満たされなくなる. そのときには, u の祖父 $w = p(p(u))$ が 2 個の赤い子をもつかどうかを調べる. もしそうなら色変換を行う. すなわち, w の 2 個の子を黒にする (図 2.9). さらに w が根でないときには, w を赤に換え $u := w$ として新しい u について再平衡化の手続きを再帰的に行う (w が根なら終了). そうでない (w が黒い子をもつ) なら, 図 2.10, 図 2.11 のように単一回転か二重回転を行い終了する. してがって, 挿入後の再平衡化は $O(\log n)$ 回の色変換とそれに続く高々二回の単一回転 (二重回転は 2 回の単一回転で可能) で実行できる.

削除後の再平衡化は以下のように行われる. 削除の最後の段階では, 除かれるべき節点 d はその内点の子と置き換えられるか (図 2.12(c)), 内点の子がないときには単に葉で置き換えられる (図 2.12(a), (b)). d に内点の子がある場合には唯一赤い節点 (d は黒) となるので黒にすればよい (図 2.12(c)). 内点の子がないときでも, d が赤のときは単に d を黒の葉にすればよい (図 2.12(b)). 問題なのは, d に内点の子がなくて d が黒のときである (d を黒

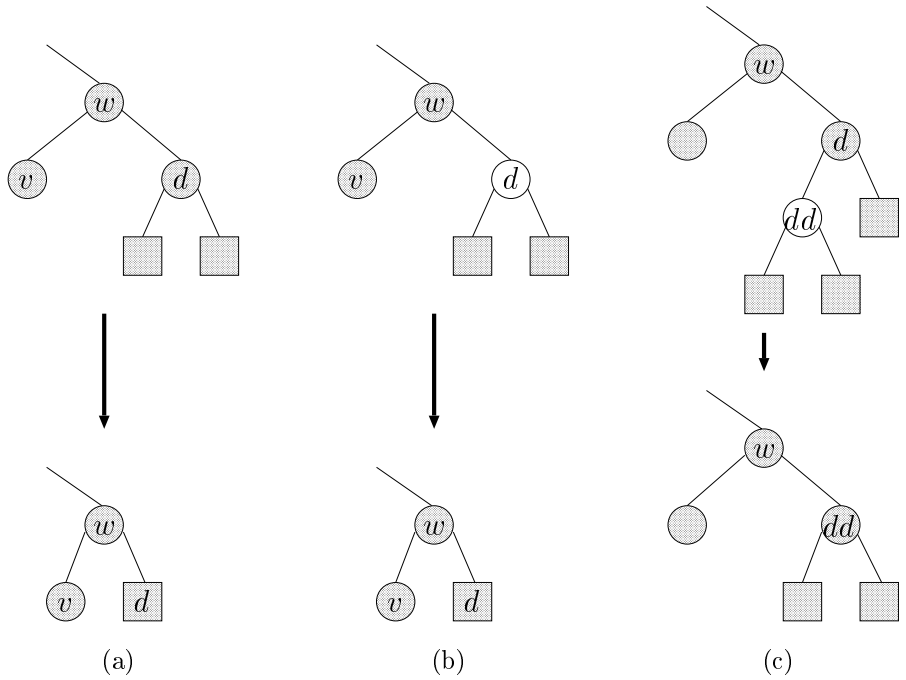


図 2.12: データの削除. 葉は四角で表示する.

の葉にすると, 根から d へのパスの黒い節点が根から他の葉へのパスの黒い節点より 1 個少なくなり, 平衡条件の (4) が満たされなくなる)(図 2.12(a)). このときは, 以下の場合に従って, d から根に向かって再平衡化を行う. ここで, w は d の親, v は d の兄弟であるが, v は必ず子をもち葉ではない (根から d 経由の葉へのパスは v 経由のパスより黒い節点が 1 個少ないから).

I. v が黒い節点の場合

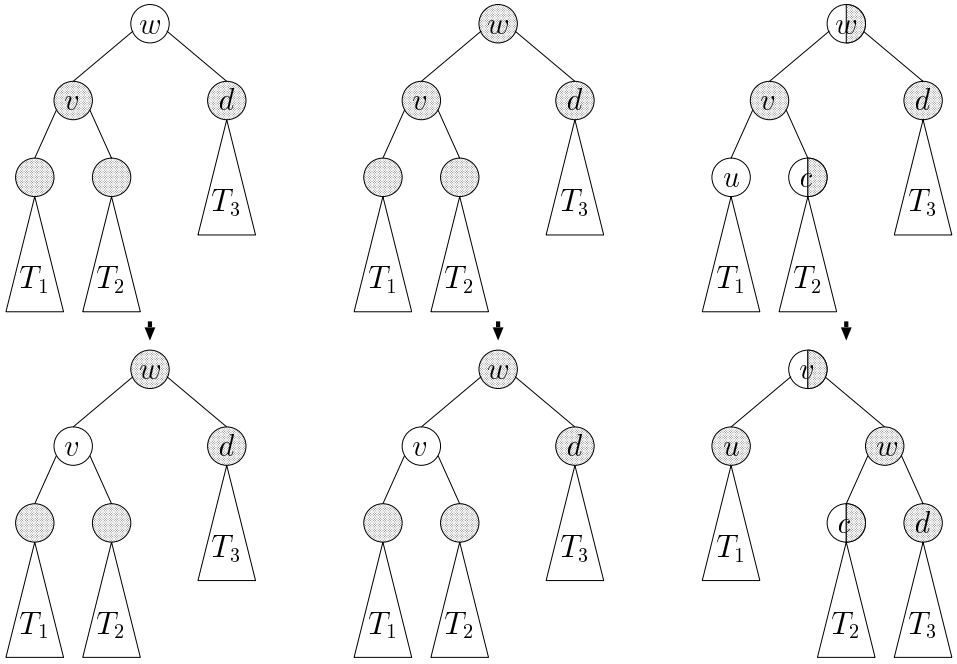
(a) v の子がともに黒い節点の場合 (図 2.13(a), (b))

v を赤にする (色変換). もし w が根であるか赤い節点ならば停止する (w が赤のときは w を黒にする). そうでなければ, $d := w$ として再帰的に I あるいは II を行う.

(b) d から遠い方の v の子 u が赤い節点の場合 (図 2.13(c))

w で単一回転を行い停止する (単一回転後 u を黒として w と v の色は交換する).

(c) d に近い方の v の子 u が赤い節点でその兄弟が黒い節点の場合 (図 2.14(a))



(a) I(a) の場合で w が赤のときは w を黒にする.

(b) I(a) の場合で w が黒のときは, $d := w$ として繰り返す.

(c) I(b) の場合

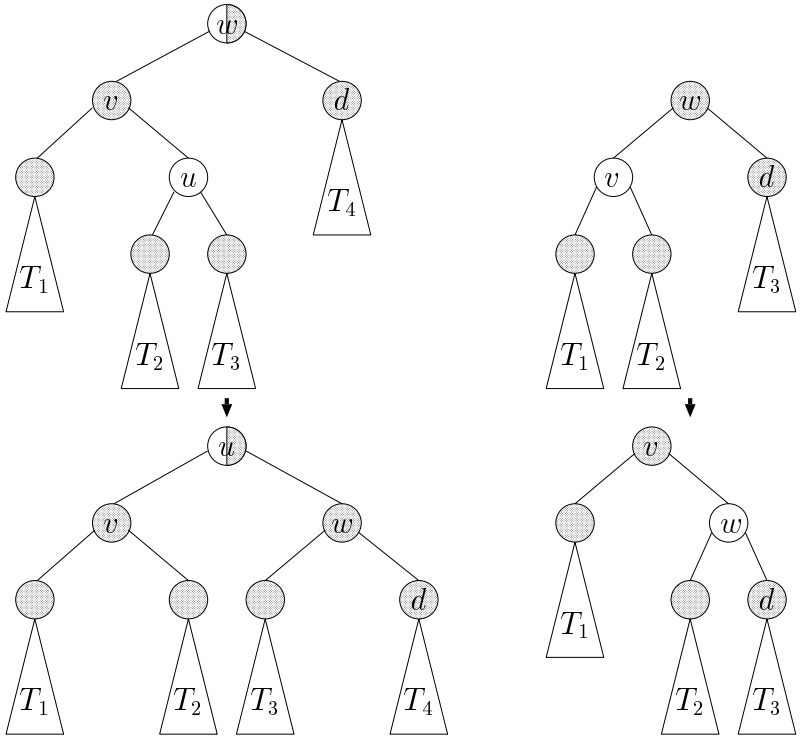
図 2.13: 削除の再平衡化 (1). 黒と赤が半分の節点はどちらの色でもかまわない.

w で二重回転を行い停止する (二重回転後 u を w の回転前の色とし w は黒とする).

II. v が赤い節点の場合 (図 2.14(b))

w で単一回転を行い w と v の色を変換し I の場合に行く (回転後 d の兄弟は黒であるので, I(a) の場合になっても w は赤でり, いずれの場合でもそれ以上進まず停止する).

このように削除後の再平衡化は $O(\log n)$ 回の色変換と高々3 回の単一回転で実行できる. これは他の代表的な平衡探索木である AVL 木などには見られない特徴である (AVL 木では挿入時の回転は定数回で抑えられるが, 削除時の回転は $O(\log n)$ 回行われることもある). 構造を変化させる回転の操作はできるだけ少ない方がよい. 特に幾何学的な問題に応用して探索木を用いるときには, 探索木の各節点にさらに副となる探索木を付随させる



(a) I(c) の場合　　(b) II の場合

図 2.14: 削除の再平衡化 (2)

ような状況も多々ある. このような状況では, 構造を変えることの少ない二色木の方が有利である. 実際, AVL 木では $O(\log^2 n)$ の時間がかかるような問題を, 二色木では $O(\log n)$ の時間ですませることもできる例が幾何学的探索問題では数多くある.

2.6 スキップリスト

ここまでに平衡二分探索木を 2 つ説明したが, そもそも平衡探索木が考えられたのは, 単純な二分木では最悪な場合に根から葉までのパスの長さが非常に不均衡になってしまうことがあるからである. データがソート順に入力されれば二分木はひとつながりのリストと等しくなってしまうが, ランダムな順序で入力されれば, 十分にバランスがとれている. この性質に注目すると, 次のような方法が考えられる. すなわち, 入力があっても直ちに二分木に挿入するのではなく, 一定個数に達すれば, スタックの内容を適当に乱数を用いて入れ換えてから二分木に挿入する. このようにすると, いつもデータの入力はランダム

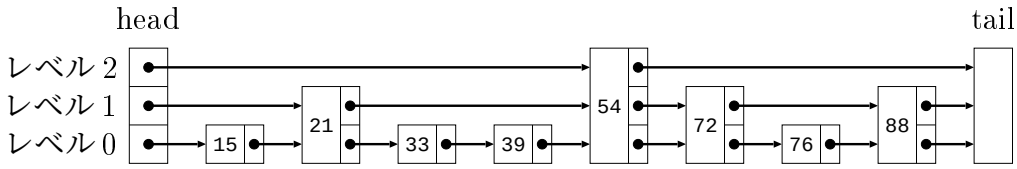


図 2.15: スキップリスト. tail のポインタは NIL.

であると考えられるから, 二分木は常にバランスがとれていると考えられる.

一方, n 個のデータが昇順に一次元配列に蓄えられているとき, ある指定されたデータを探索するのに逐次探索を行うと, もちろん最悪の場合には $n-1$ 回の比較が必要になる. しかしながら, 配列を m 個ずつのデータからなるブロックに分割し, 各ブロックの最大のデータと順に比較をしてどのブロックに探索データが存在するか調べ, その後そのブロック内を逐次比較により調べると, 比較回数は $m + n/m$ に減少する. この方法は m-ブロック法と呼ばれる方法である. この m-ブロック法において, ブロック内を逐次探索する部分に再び m-ブロック法を用いたのが二重 m-ブロック法である. この方法を用いると, 探索のための比較回数は高々$m \log_m n$ となる. したがって, m を定数と考えると, 探索のための比較回数は $O(\log n)$ であるということができる.

上に述べた 2 つ方法を組み合わせたのが次に述べるスキップリスト [12] である. この方法では, 図 2.15 に示すようなリストを用いる. 図 2.15 のようにリストは高さの異なるレコードが混在している. 高さ h のレコードは h 個のポインタを持つが, それらは $1 \leq k \leq h$ なる各 k に対して, その右にある最初の高さ k のレコードを指し示している. したがって, 最も下のレベル (レベル 1) ではすべてのレコードが連結しているが, その上のレベル 2 ではいくつかのレコードがスキップされる. 一般に, レベルが高くなるにつれ, スキップされるレコード数は倍増する.

2.6.1 データの探索

与えれたデータ x を含むレコードが存在するかどうかを調べる場合, 最初は最大のレベルにおいて探索する. このレベルのレコードのうち, x を超えない最大のデータを含むものを求める. 次にレベルを 1 つだけ下げて, そのレベルのレコードのうち x を超えない最大のレコードを含むものを求める. 同様に, それ以下のレベルにおいても x を超えない最大のデータを含むレコードを求める. 結局, このようにしてすべてのレベルにおいて探索を行ったとき, どのレベルでも x に等しいデータを含むレコードが見つからなければ, そ

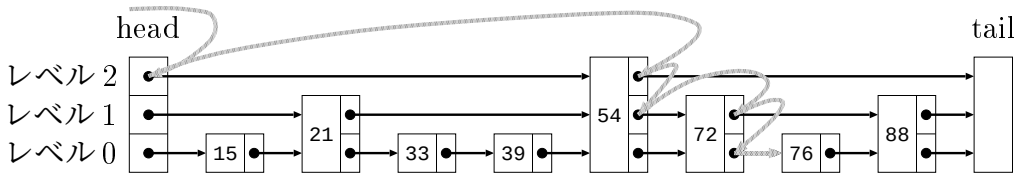


図 2.16: スキップリストの探索

のようなレコードは元々含まれていなかったことになる.

図 2.16 に示すスキップリストに値 76 が含まれるかどうかを探索する場合を考えてみる. 上に述べたように, 最大のレベルから順に 76 を超えない最大のデータを含むレコードを探すから, レベル 2 では 54 が見つかり, レベル 1 では 54 から探索を始めて 72 を見つける. 最後のレベル 0 ではポインタが 72 を含むレコードを指し示している状態で終わるが, これが 76 を超えない (正確には, 76 より小さいデータの中の) 最大値を含むレコードであるから, これの次のレコードを調べれば, 求めるレコードが存在したかどうかがわかる.

2.6.2 データの挿入

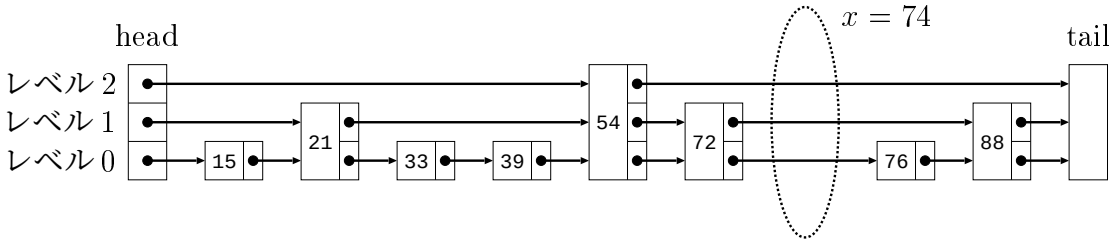


図 2.17: スキップリストにデータ $x = 74$ を挿入する位置

スキップリストに新たなデータ x を追加する場合, まず上に述べた探索と同じ方法で, 挿入すべき場所を求める. 各レベルにおいて x を超えない最大のデータを含むレコードを求めるのは探索の場合と同じであるが, 今度はそれらのレコードへのポインタを配列 $update[]$ に入れて記憶しておく (図 2.17). その後, レコードの高さは乱数によって決める. この場合, レコードが i 以上のレベルにある確率が p^i になるように決めれば平均的に $O(\log n)$ 時間の探索が可能となる. p の値としては, 例えば $p = 0.5$ とすればよい. このように決めるとレコードが i レベルにある確率は $(1-p)p^i$ となる.

挿入すべきレコードの高さをランダムに決めた後, 実際にリストに挿入する. 先にも述べたように, 挿入すべき場所が 1 つ手前のレコードのポインタが配列 $update[]$ に格納されているから, 後はポインタのつけかえだけである.

このようにして多数のデータを挿入していくとき, リストのサイズに比べて最大レベル $maxh$ が小さくなり過ぎると, 探索・挿入の時間を $O(\log n)$ に保つことができない. そこで, $\log \mathrm{size} + 1 > maxh$ が成り立てば, 最大レベルを 1 つだけ上げる. 具体的には, 新たな最大レベル以上の高さをもつレコードを順に連結する. (挿入だけを考えるとそのようなレコードは存在しないが, 削除によっては最大レベルが下がることもあるので, 必ずしも各レコードのすべてのポインタが使われているとは限らないことに注意).

先の例で $x = 74$ をリストに挿入する場合, レコードの高さが 2 と決まれば, 図 2.18 のようになる.

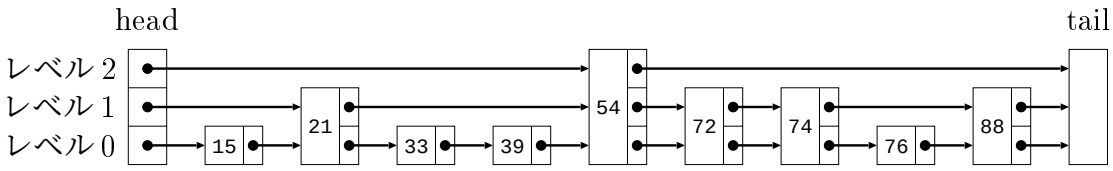


図 2.18: スキップリストにデータ $x = 74$ を挿入後

2.6.3 データの削除

データの削除はもはやあきらかであろう. 削除すべきデータ x が与えられると, 挿入の場合と同様にして, x を超えない最大のデータを含むレコードへのポインタを配列 $update[]$ に入れて記憶する. 後は, 削除すべきレコードの高さ以下のレベルにおいて前後のポインタをつなぎかえることによってレコードを削除する (図 2.19). 削除の結果, リストのサイズの log 値が最大レベル $maxh$ より小さくなってしまえば, 最大レベル $maxh$ を 1 だけ下げる. この場合は挿入のときのような厄介な操作は必要なく, 単に $maxh$ の値を 1 だけ減らせばよい.

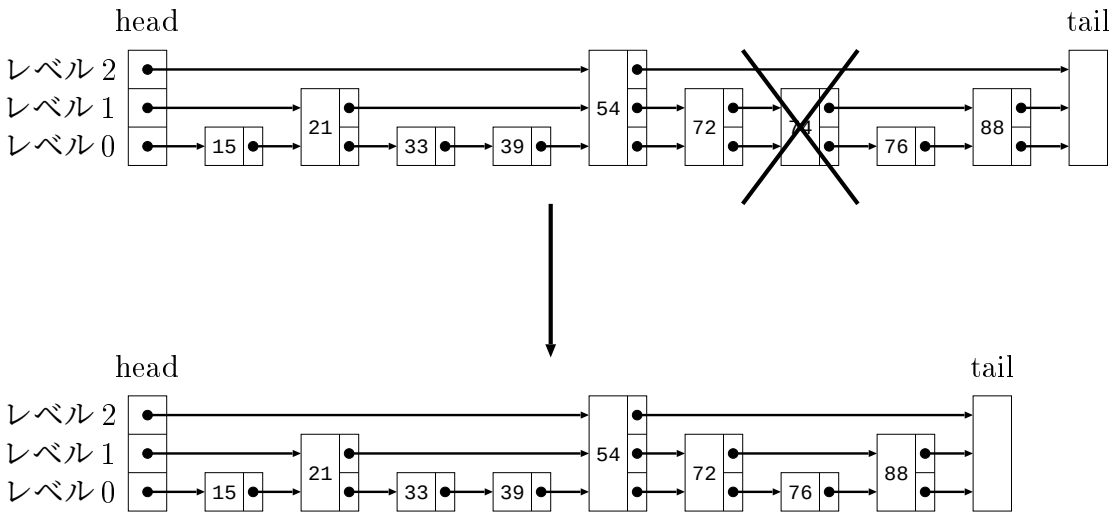


図 2.19: スキップリストの削除

第 3章

グループ更新

まず, 人間の図形を描画するときの特性を説明し, それを利用するデータ構造を提案する. そして, グループ更新のアルゴリズムと効率を説明し, 従来の更新方法との理論的比較を行う.

3.1 人間による描画の特性とそれを利用するデータ構造

人間が図形を描画するときには, 連続して入力をするわけではなく一般的に描こうとする図形を頭に思い描いてから入力を行う. 計算機では, その思考時間の間は図形が入力されない入力待ち状態になっている. つまり人間による図形の描画は, 描こうとする図形の思考時間と実際に図形を入力する時間に分けることができ, それを繰り返すことで図形を描画していく. この人間による入力の特性から生まれる計算機の待ち時間を有効に利用し, その間に計算機側の処理を行うようにすれば, 図形の描画に関する操作がスムーズ行われ, 操作に対する計算機の応答も瞬時に行われることが期待できる.

思考時間の間に計算機で処理を行うと言っても, 最低でも今現在の描画処理は行わないとどこにどんな図形を描画しているのか分からなくなってしまう.

そこで, 本研究で考えるところは入力時に行われる最低限の描画処理はすぐに処理し, 時間がかかってしまうデータ構造の更新を後にすることで効率良く図形をしようというわけである. 実際どうのようにするのかというと, データを蓄積する部分を 3 つ用意する. 1 つはメインとなって全体のデータを蓄積する部分で, 残りの 2 つは一時的に更新するデータを蓄積しておく部分である. 挿入されるデータを蓄積する部分と, 削除されるデータを蓄積する部分である.

人間が図形を描画しているときに挿入されるデータ, または削除されるデータは一時的

に蓄積しておく部分に蓄積する. 次に人間が思考時間をとっている間に, 今まで一時的に蓄積されたデータをメインのデータ蓄積部分に挿入, またはメインのデータから削除する.

蓄積部分が 1 つである場合には, そこにすべてのデータが蓄積されることになり, データの更新に時間がかかってしまうことになる. 今回の方法は, 今現在の更新操作だけを一時的に蓄積するのであり, メインのデータに対しての更新操作は後で行われるので, 今現在のデータの更新にかかる手間は軽減されることになる.

つまり, 思考時間に一度にまとめてデータの更新を行おうというわけである. これを “グループ更新” と呼ぶことにする. このグループ更新は, 一時的に蓄積された更新操作を従来のように単純なやり方でメインのデータ構造に反映するのではなく, 単純な方法よりも効率良く更新をおこなえるうように工夫を凝らす. 単純な方法とは, データを 1 つずつメインに挿入, または削除することである. グループ更新の詳しいアルゴリズムは 3.3 で説明をする.

このグループ更新を効率よく行うために必要なデータ構造として緩和二色木がある.

3.2 緩和木

緩和木を使う意味は, 挿入と削除に関係づいている再平衡化を実際の更新と分離させることと, それを随意に遅らせることができることである. そのような環境では, 探索木のために適応される操作は探索, 挿入, 削除と再平衡化である. さらに, 再平衡化の操作が必要な節点の数が少なくなる. これは適用されるすべての操作が, 一度に少ない節点を含み, 少ないステップで進められるべきなので, 並列処理では重要なことである.

緩和木は二分探索木に関する並列処理問題の効率の良い解法を与える. 緩和木の要点の 1 つとして, 探索木を使った並列処理の効率をかなり向上させることができる. その理由として, 緩和木は 2 つの独立した部分で挿入と削除を実行している. 第 1 に実際の挿入と削除を実行する. 第 2 にそれに対する再平衡化を実行する.

並列処理を増やすため挿入や削除の後すぐに再平衡化をしないことが試みられ, その詳細は共有メモリに関するデータ構造の文献で知ることができる.

3.2.1 緩和二色木

緩和二色木 [9] は共有メモリの並列処理のための葉優先の全二分探索木である. 並列度を下げることなく, 更新と再平衡化の矛盾をなくすためのロック機構が紹介された [11]. 再平衡化は独立した小さなステップでバックグラウンドで処理される. 更新操作から再平衡

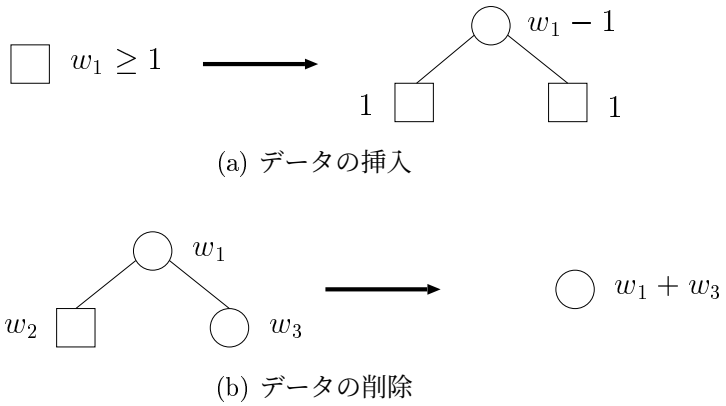


図 3.1: 緩和二色木の更新操作

化を分離したことでよいことは, すべてのまたは一部の再平衡化を更新の処理が多いときにそれが終わるまで延期できることである. 都合が悪いこととしては, 再平衡化の処理があまりにも延期されすぎると木のバランスが保てなくなってしまうことである. 再平衡化の手間として, 挿入のときは $\lfloor \log(n+i) \rfloor$ で, 削除のときは $\lfloor \log(n+i) \rfloor - 1$ となる.

緩和二色木は, 葉優先の全二分探索木であるので, データは葉に蓄えられすべての節点の子は 0 か 2 つのどちらかである. そして内点は, 木の探索を行うときに目的のデータまでたどり着くように案内をする役目をもち, ルータと呼ぶことにする. 節点 v のルータは, 左部分木のどのキーよりも大きいか等しく, 右部分木のどのキーよりも小さくなければらない. ルータは削除されるキーと一致するキーをもった節点であるので, 木で与えられるキーではなくルータとして機能する. 特に, ルータは木のキーである必要はない.

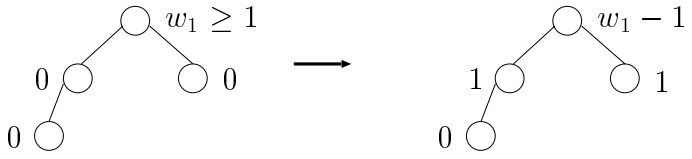
二色木の平衡条件を緩和してできたものが緩和二色木である. よって二色木と似ている. 二色木では節点に赤と黒の色を割り当てていたが, 緩和二色木では正の整数を重みとしてそれぞれの節点にもたせる. 二色木で赤に相当する重みは 0 であり, 黒に相当する重みは 1 である. 重みが 1 を超えてしまう節点をオーバウエイトと呼ぶことにする. パスの重みとは, そのパス上の節点の重みを合計した値である. 節点の重みのレベルとは, 根からその節点までのパスの重みである. 二色木の平衡条件は以下の 4 つであった.

(1) すべての節点は赤か黒のいずれかである.

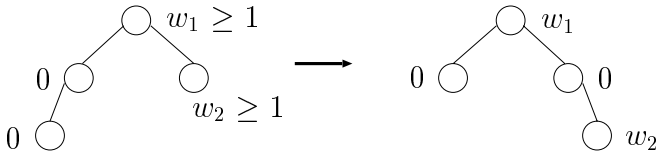
(2) 赤い節点は必ず黒い親をもつ.

(3) 根と葉は黒である.

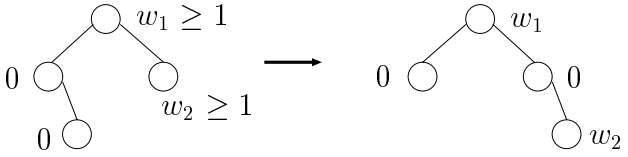
(4) 根から葉へのパスはどのパスも同数個の黒い節点をもつ.



(a) blacking



(b) rb1



(c) rb2

図 3.2: 赤の衝突を取り除く操作

緩和二色木はこれらの平衡条件を緩和して定義される. それは以下である.

(1) 葉は赤ではない.

(2) 根からすべての葉までのパスの重みは同じである.

二色木は当然この平衡条件を満たしている. 二色木と異なるところは, 赤の衝突が許されることと, 重みが 1 を超える場合があることである. このように平衡条件を緩和することで更新後に必ずすぐに再平衡化をする必要がなくなる.

挿入と削除の操作は図 3.1 に, 再平衡化の操作は図 3.2, 図 3.3 に示す. これらの再平衡化の操作は, 適用される順番は決っておらずどの順番に適用されてもバランスがぐずれた状態から平衡条件を満たした緩和二色木に戻すことができる. 緩和二色木の再平衡化は緩和二色木の平衡条件を満たすようにするとともに, その再平衡化を十分に行うことで最終的には二色木へと変形していく. これは [9] で証明されている.

もし緩和二色木が二色木の平衡条件を満たしたいない場合は, 2 つの原因が考えられる. それは, 赤の衝突とオーバーウエイトの存在である. 赤の衝突とは, 重みが 0 である節点

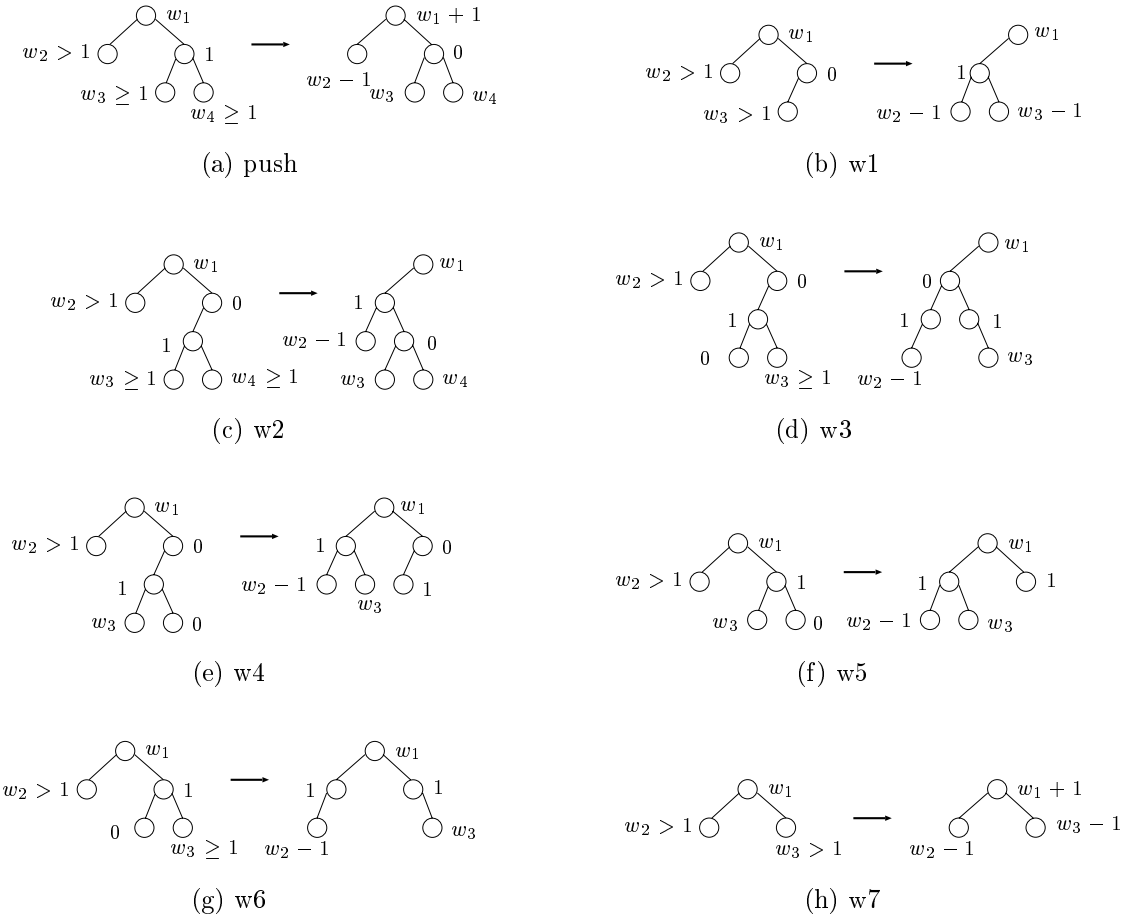


図 3.3: オーバーウエイトを取り除く操作

がパス上で連続して現れていることをいう. これは二色木の平衡条件の (2) を満たさない. また, オーバーウエイトの存在は, 二色木の平衡条件の (1) を満たしていない. この 2 つを再平衡化によって取り除くことで二色木へと変形していく.

このように平衡条件を緩和したことにより, 更新をした後ですぐに再平衡化をしなければならない場合が軽減されている.

挿入: 最初に新しく挿入するデータで探索を行う. そのデータがもしも見つかったら処理は終了する (蓄積するデータはすべて異なると仮定). そのデータが見つからず探索に失敗したら, ある葉 v で探索は終了しているのでそこの葉 v に新しいデータを挿入することになる. 新しい内点を u とすると, u を v の位置に挿入する. そして u の葉として v と新しいデータを挿入する. そのとき, 小さいデータの方が u の左の子と

なる. u のルータは左の子のデータとするのでそのデータを u のルータとする. u の重みは v の重み –1 とし, v と新しく挿入された節点の重みは 1 とする (図 3.1(a)).

削除: 最初に削除するデータを探索する. もしも見つからなければ処理は終了する. 見つかればそのデータをもつ葉を削除する. その親は削除される葉ではないもう一方の葉 u で置き換えられる. その u の重みは前の u 重みと u の親であった節点重みの合計値とする (図 3.1(b)).

挿入によって重みが 0 の節点が作られるかも知れない. もし作られた場合, 赤の衝突が発生する可能性がある. また, 削除による結果としてはオーバーウエイトを新たに作るかも知れない. そして, すでにあるオーバーウエイトの重みを増やすかもしれない.

このようにして発生した赤の衝突やオーバーウエイトは, 再平衡化を十分行うことで取り除くことができる. 再平衡化について, 赤の衝突は blacking 操作と red-blancing(rb1,2) 操作で 1 つ減らすことができる. オーバーウエイトは, push 操作と w1-w7 で重みを 1 減らすことができる.

3.3 グループ更新

この節では, 従来のように挿入や削除といった更新操作が行われるごとに再平衡化を実行していたのに対し, いくつかの更新操作を 1 つのグループにまとめてデータ構造の更新を一度に行いその後で再平衡化を行うアルゴリズムを説明する.

データを蓄積しておく部分は 3 つ用意する. 1 つはメインとなるデータ蓄積部分で緩和二色木を用いたデータ構造である. 残りの 2 つは更新操作を一時的に蓄積しておく部分で, そのデータ構造は二色木を用いる. 挿入のデータを一時的に蓄積しておく部分を $TEMP1$, 削除のデータを一時的に蓄積しておく部分を $TEMP2$ とする.

連続して入力があるときには, 各 $TEMP$ 領域に更新操作を保存しておきユーザが思考時間に入ったらこの $TEMP$ 領域に蓄積された更新操作をまとめてグループ更新を行う. グループ挿入, グループ削除のアルゴリズムをこれから説明していく. このグループ更新を用いる場合には, メインに挿入する, またはメインから削除するデータはソートされている必要がある. ソートされていない場合は, ソートする必要がある. データのソートにかかる時間は $O(n \log n)$ である. n はデータ数とする. 本研究では, 各 $TEMP$ 領域に葉優先の二色木を用いているため, 最小のデータは最も左の葉に, 最大のデータは最も右の葉に蓄積されている. だから, データを昇順に取り出す場合は, 最も左の葉から順に右にたどっていけばよいので, 特にソートする必要はない.

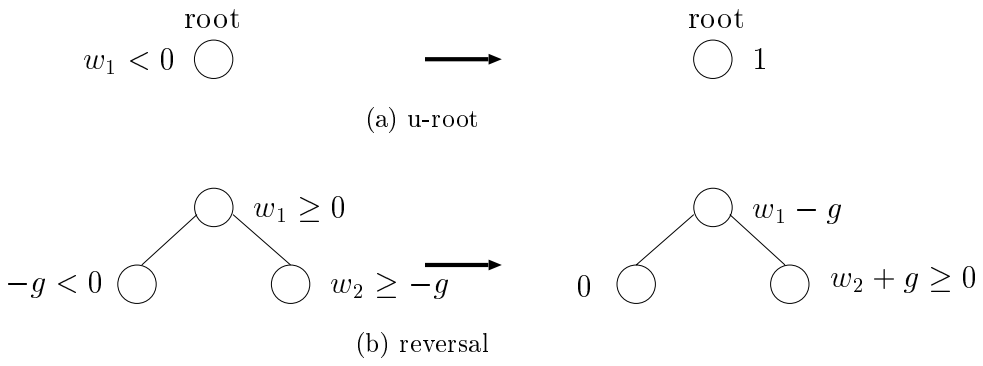


図 3.4: アンダーウエイトの除去

本来, 緩和二色木の更新操作と再平衡化は分離して行われ, 再平衡化に関してはバックグラウンドで実行されることを説明した. これはマルチプロセッサによる並列処理を前提にしたものである. 今回はシングルプロセッサを用いて実装することを前提としているので, 更新操作と再平衡化は分離されておらず, 再平衡化は更新操作が行われた直後に実行されるものとする.

3.3.1 グループ挿入

メインの緩和二色木を T とし, 挿入操作されるデータが一時的に蓄積されている二色木を $TEMP1$ とする. グループ挿入のアルゴリズムの大まかな流れは以下のようになる (図 3.5).

1. $TEMP1$ のデータを, T で探索したときに終了する葉 l_i ごとに, b ブロックに分割する.
2. $TEMP1$ のデータが挿入される場所の葉 l_i のデータを含めてブロックのデータで二色木 T_i を構成する.
3. T の葉 l_i を T_i で置き換え, l_i の重みを $1 - w$ とする. ここで w は T_i の根から葉までのパスの重み −1 とする.
4. T の再平衡化を行う.
5. 挿入ブロックが残っていたら 2 に行く.

次は再平衡化について説明をする. 緩和二色木は正の整数だけを重みにするが, 今回のグループ挿入を実行するときには負の整数も重みにできるようにする. 負の重みをもつ節

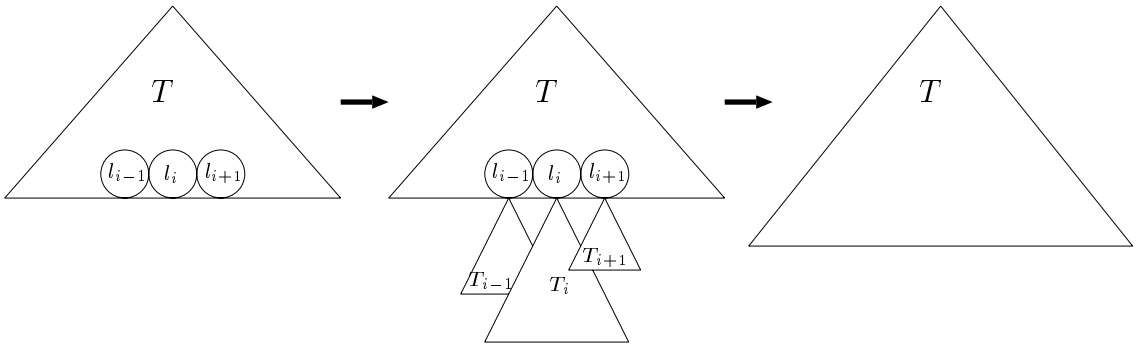


図 3.5: グループ挿入のイメージ

点をアンダーウエイトと呼ぶ.

グループ挿入の再平衡化は以下のように行われる.

1. アンダーウエイトなら重み $-g$ を $+g$ して 0 にする. もしも根がアンダーウエイトだったら, 重みを 1 にして終了する (図 3.4).
2. 1 の操作で赤の衝突が発生したら, 緩和二色木の再平衡化を適用する.
3. 他方の子の重み w_1 に $+g$ して, $w_1 := w_1 + g$ とする.
4. 他方の子の重み w_1 がオーバーウエイトなら緩和二色木再平衡化を操作を適用する.
5. 親の重み w_2 に $-g$ して, $w_2 := w_2 - g$ とする.
6. 親がアンダーウエイトならば親の節点で 1 を繰り返す. 親の重みが 0 なら 2 に行く. 重みが 1 ならグループ挿入の再平衡化は終了する.

具体的な方法については次のグループ挿入の手間で説明する.

3.3.2 グループ挿入の手間

メインの緩和二色木のデータ数を N とする. $TEMP1$ のデータ数を m とし, ブロック分割されるブロック数を b とする. 各ブロックのデータ数は m_i とする ($m = \sum_{i=1}^{b} m_i$).

二色木を実装する構造上により, $TEMP1$ から昇順にデータを取り出すことは葉を左から右に順番にたどっていけばよいのでソートをする必要がない. ブロック分割にかかる手間は, まず最小のデータから始めていき T で挿入されるべき葉 l_1 を探索する. その葉 l_1 のデータの値より小さい $TEMP1$ のデータはすべてその葉 l_1 に挿入されることになるので,

それらのデータによりブロック 1 が決定する. ブロック 1 のデータは $TEMP1$ の各データを昇順に葉 l_1 と比較することで容易に求まる. 葉 l_1 より大きい $TEMP1$ のデータが現れたら, そのデータが挿入されるべき T の葉 l_2 を探索する. ブロック 1 を決めた方法と同様の操作でブロック 2 を決定する. これをブロック b まで繰り返す. ブロック i のデータを決めるのに T を探索するデータは, ブロック i の最小データだけであるので全体では, b 回だけ T を探索することになり, その探索に必要な合計時間は $b \log N$ である.

次に各ブロックごとに構成する二色木にかかる手間はについて考える. データ M からなる二色木を構成するのにかかる手間は $M \log M$ であるので, b 個の m_i のデータを含むブロックから二色木を構成するためにかかる手間は, $\sum_{i=1}^{b} m_i \log m_i$ となる.

次は再平衡化に関しての手間について考える. グループ挿入の再平衡化はアンダーウエイトを減らしながら, オーバーウエイトと赤の衝突を減らしていく. 向かう方向は, l_i から根に向かってボトムアップで行う. グループ挿入の再平衡化でアンダーウエイトの重みを 0 にし, 他方の子の重みにその分を足し, 親の重みからは引く操作を $reversal$ と呼ぶ (図 3.4). $reversal$ では親の重みが 1 であるときアンダーウエイトの重みを 1 足すことができ, アンダーウエイトを減らすことができる. 親の重みが 0 であった場合にはアンダーウエイトは根の方に移っていくだけで, アンダーウエイトを減らすことができない. アンダーウエイトの重みが $-g$ である場合の $reversal$ の実行回数を考える. 再平衡化が十分に行われている緩和二色木は二色木の平衡条件を満たしているので, グループ挿入をする前は二色木の平衡条件を満たしている. つまり, パスが一番長いときは節点の重みが 0 と 1 で交互になっているときである. このときに $reversal$ が最大回数行われることになり, $-g$ の重みを 0 にするためには, 高々 $2g$ 回の $reversal$ をすればよい. それが b ブロックあるので全体では $2\sum_{i=1}^{b} g_i$ 回となる.

g_i を求めるのにかかる計算時間は, m_i のデータのブロックで構成される二色木の高さは $\log m_i$ であるので, $\sum_{i=1}^{b} \log m_i$ で求まる.

緩和二色木の葉 l_i にグループ挿入をしたとき, その l_i の重みがアンダーウエイト $-g_i$ であるとすると, $reversal$ により生成される赤の衝突は $O(\sum_{i=1}^{b} g_i)$ でオーバーウエイトの節点は $O(\sum_{i=1}^{b} g_i^2)$ 生成されることが知られていて, その赤の衝突とオーバーウエイトを取り除くために必要な操作は, $O(\sum_{i=1}^{b} \log^2\ m_i)$ の回転と $O(\sum_{i=1}^{b} \log^2\ m_i + L)$ の色変換が必要となる [7]. ここで L は根からグループ挿入される葉 l_i までのパス上で色変換が必要な節点の数でる.

したがって, グループ挿入に必要な全体の手間は緩和二色木のデータを N, 分割ブロック数を b, そのデータ数をそれぞれ m_i としたとき,

$$O(b \log N + \sum_{i=1}^{b} m_i \log m_i)$$

である.

グループ挿入を用いずに, 従来のように 1 つずつデータを挿入するときにかかる手間は,

$$O(m \log N)$$

である.

3.3.3 従来の挿入方法との比較

葉優先の全二分探索木である二色木を用いた場合の従来の挿入方法とグループ挿入の理論的比較を行う.

従来の挿入とは, データを 1 つ挿入するごとに再平衡化を行う方法を指す. 挿入するデータ数は m とする. b はグループ挿入の分割ブロック数とする.

- メイン木の探索回数
  - 従来の方法

    $O(m)$ 回
  - グループ挿入

    $O(b)$ 回
- 再平衡化の回数
  - 従来の方法

    $O(m)$ 回の単回転
  - グループ挿入

    $O(\sum_{i=1}^{b} \log^2 \ m_i)$ 回の単回転

3.3.4 グループ削除

メインの緩和二色木を T とし, 削除操作されるデータが一時的に蓄積されている二色木を $TEMP2$ とする. グループ削除の大まかな流は以下のようになる (図 3.6).

1. $TEMP2$ のデータの中で T の葉で連続になるものを探し, その削除される葉だけを葉にもつような部分全二分木 T_i を探索する.
2. T から T_i を削除する.
3. T_i の根の他方の子を v とすると, その親 u を v で置き換える.
4. T の再平衡化を行う.
5. $TEMP2$ に削除しなければならないデータが残っていたら 1 に行く.

グループ削除の再平衡化については, 二色木の削除の場合とほぼ同じでり, 以下のようになる (図 2.12〜図 2.14).

- u の重みが 0 である場合は, 再平衡化は行わない.
- u 重みが 1 で v 重みが 0 の場合は, 再平衡化は行わない.
- u 重みが 1 で v 重みも 1 の場合は, 緩和二色木の再平衡化を行う.

3.3.5 グループ削除の手間

$TEMP2$ のデータの中で T の葉で連続になるものを探し, その削除される葉だけを葉にもつような部分全二分木 T_i を探索する手間を考える.

メインの緩和二色木のデータ数を N とする. $TEMP2$ のデータ数を m とし, 削除する部分全二分木 T_i の数を b とする. 各部分全二分木のデータ数は m_i とする ($m = \sum_{i=1}^{b} m_i$).

部分全二分木 T_i を決定するためにかかる手間は, まず $TEMP2$ 最小のデータから始めていき T で削除されるべき葉 l_1 を探索する. そこから 2 つの処理を行いながら部分全二分木 T_i を決定する. 行う処理の 1 つは, 葉 l_i から右に連続する葉が $TEMP2$ のデータに含まれているかどうかを調べる処理である. もう 1 つは, 削除される葉が部分全二分木に含まれるかどうか調べる処理である. この 2 つの処理は交互に行われ, どちらかが満たされなくなったら終了する.

葉 l_i から右の葉に移動しながら $TEMP2$ にその葉が含まれるか確認しつつ, 一方で葉 l_i から上にあがっていき, その度に右の子をたどって行き現時点での部分全二分木 T_i の最も右の葉が $TEMP2$ に含まれているか確認をする.

このようにして部分全二分木 T_i が決定される. それにかかる手間は, 部分全二分木かどうかの調査に $\log^2 m_i$ かかり, 右に連続する葉が $TEMP2$ のデータに含まれているかどうかを調べるのに m_i の手間が必要になるので, $O(m_i)$ である. b 個の部分全二分木を決定するのに必要な手間は $O(\sum_{i=1}^{b} m_i)$ である.

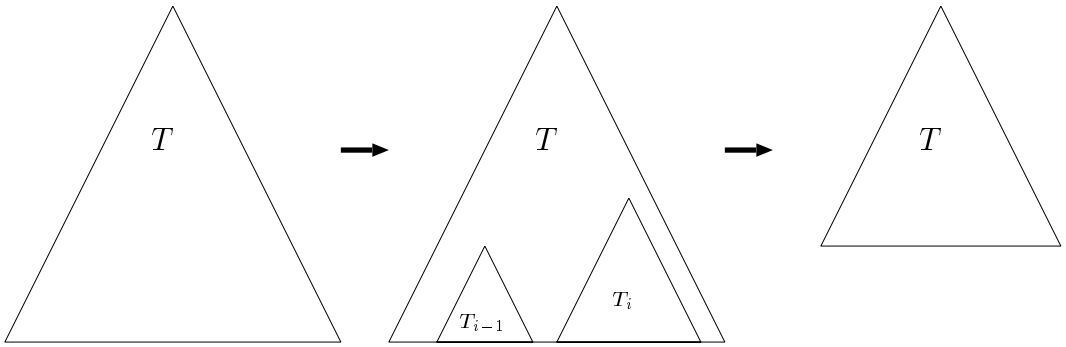


図 3.6: グループ削除のイメージ

部分全二分木 T_i を決めるのに T を根から探索するデータは, 部分全二分木 T_i の最小データだけであるので全体では, b 回だけ T を探索することになり, 必要な探索の手間は $b \log N$ である.

次にグループ削除の再平衡化にかかる手間を考える. 再平衡化は二色木の場合と同様なので, 1 つの T_i 削除後の再平衡化は $O(\log n)$ 回の色変換と高々3 回の単回転で実行できる. 部分全二分木 T_i が b 個できるとすると, それにかかる再平衡化の手間は $O(b \log n)$ 回の色変換と高々3b 回の単回転である. ここで構造を変化させる再平衡化の手間は, $O(b)$ にしかかからない.

したがって, 緩和二色木のデータを N, グループ削除のデータ数を m, 部分全二分木 T_i が b 個できるとすると, グループ削除にかかる全体の手間は以下のようになる.

$$O(b \log N + m)$$

グループ削除を用いずに, 従来のように 1 つずつデータを削除するときにかかる手間は,

$$O(m \log N)$$

である.

3.3.6 従来の削除方法との比較

葉優先の全二分探索木である二色木を用いた場合の従来の削除方法とグループ削除の理論的比較を行う.

従来の削除とは, データを 1 つ削除するごとに再平衡化を行う方法を指す. 削除するデータ数は m とする. b はグループ削除の削除する部分木の数とする.

- メイン木の探索回数
  - 従来の方法

    $O(m)$ 回
  - グループ削除

    $O(b)$ 回
- 再平衡化の回数
  - 従来の方法

    $O(m)$ 回の単回転
  - グループ削除

    $O(b)$ 回の単回転

第 4 章

実装と実験

4.1 実装

本研究で紹介したグループ更新を行うデータ構造を用いて, 実際に図形を描画するツールを実装した. プログラム言語は C++で, ドイツで開発されたアルゴリズムライブラリ兼アルゴリズム記述言語である LEDA も利用している. この LEDA の助けもあり, 比較的少ない時間と手間で実装することができた. main 関数は約 200 行で, 他に必要な部分を記述するのに約 1000 行程度を要した.

このツールを使用する上で以下の制約がある.

- 描画できる図形は線分だけである.
- すべて異なる線分データのみを扱う.
- 操作は, 挿入と削除とする.
- 使用できるデータ構造は, AVL 木, 二色木, スキップリストと緩和二色木である.
- 入力待ち時間が 1 分を超えると思考時間とみなし, グループ更新を自動的に行う.

動作画面は図 4.1 である. また, ツールを使って出力したグループ更新の様子を, 付録に添付する. 図 4.2 はグループ挿入で, 線分を 4 本描き, その後でどの線分よりも左側に 4 本線分を追加した様子である. 図 4.3 は, グループ削除として, 右から 3 本目と 4 本目の線分を削除した様子を示している.

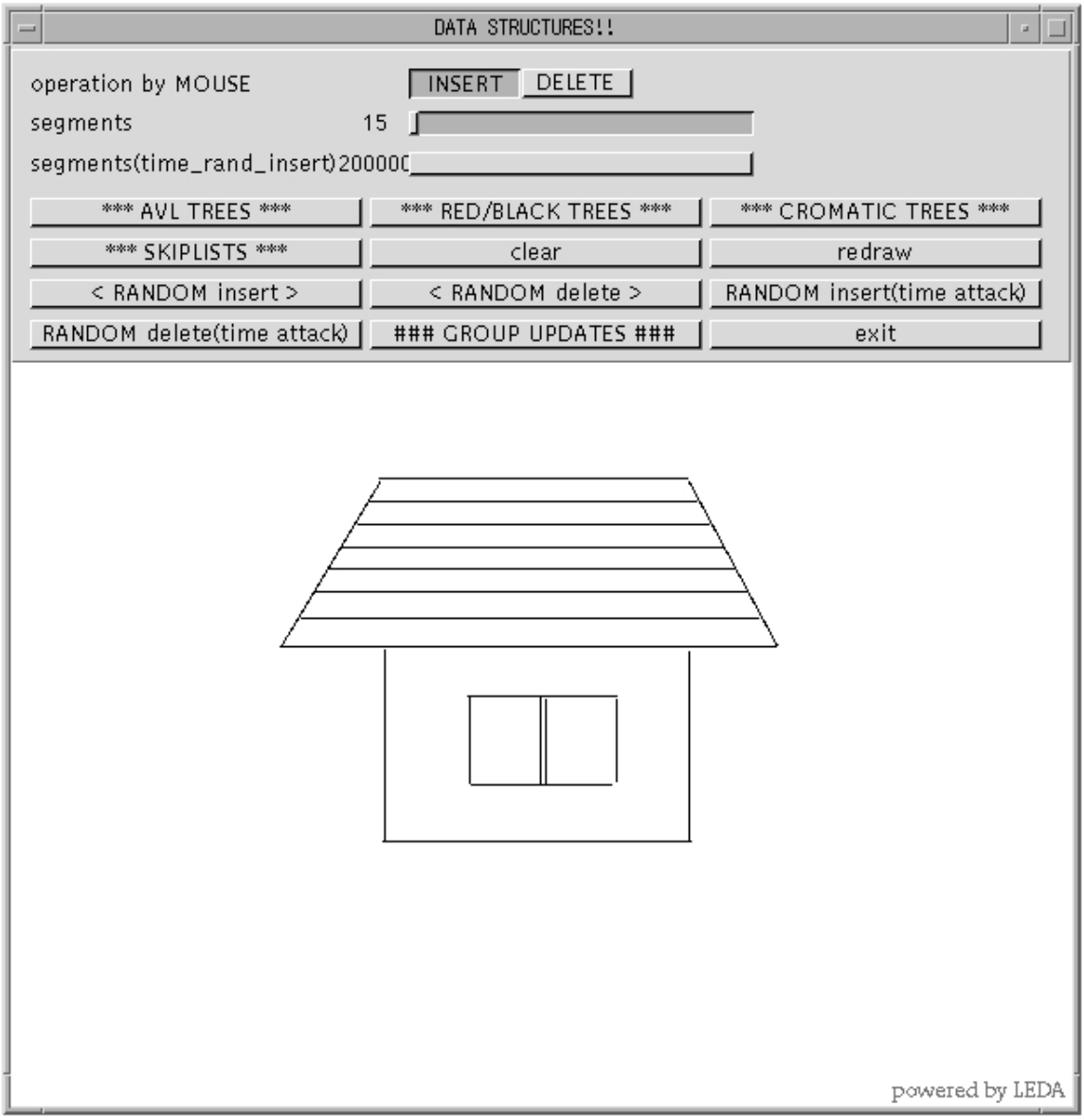
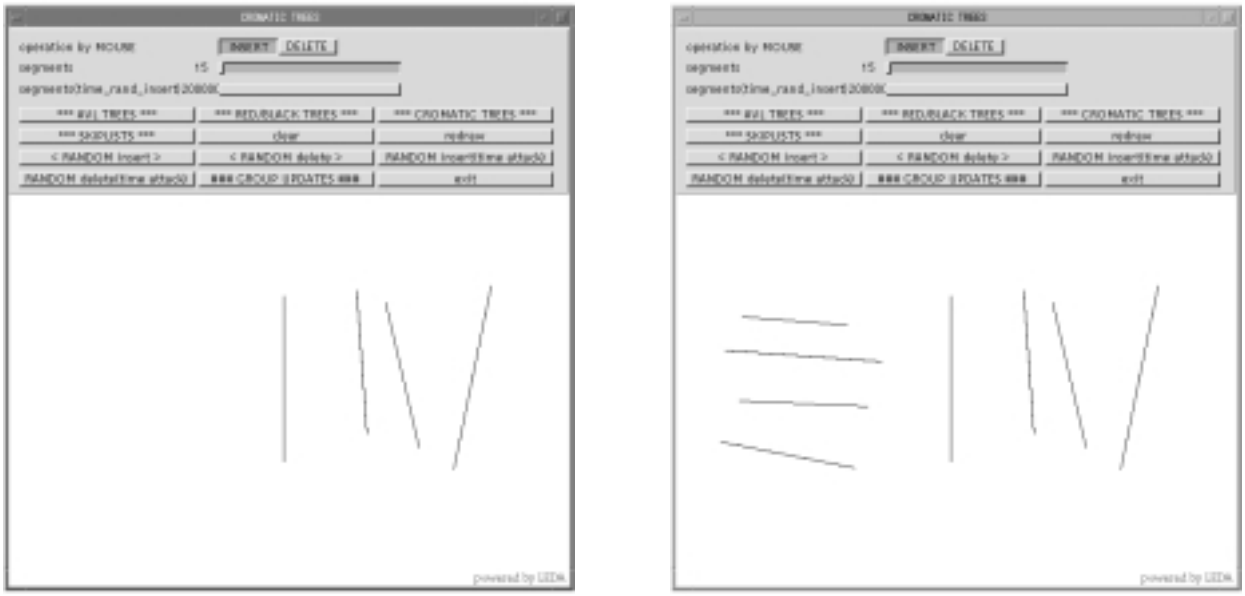

DATA STRUCTURES!!
operation by MOUSE
INSERT
DELETE
segments
15
segments(time_rand_insert)200000
*** AVL TREES ***
*** RED/BLACK TREES ***
*** CROMATIC TREES ***
*** SKIPLISTS ***
clear
redraw
< RANDOM insert >
< RANDOM delete >
RANDOM insert(time attack)
RANDOM delete(time attack)
GROUP UPDATES
exit
powered by LEDA

図 4.1: 実行画面

4.1.1 グループ挿入の様子



(a) 挿入前　　(b) 挿入後

図 4.2: グループ挿入

緩和二色木の構造の変化を以下に示す.

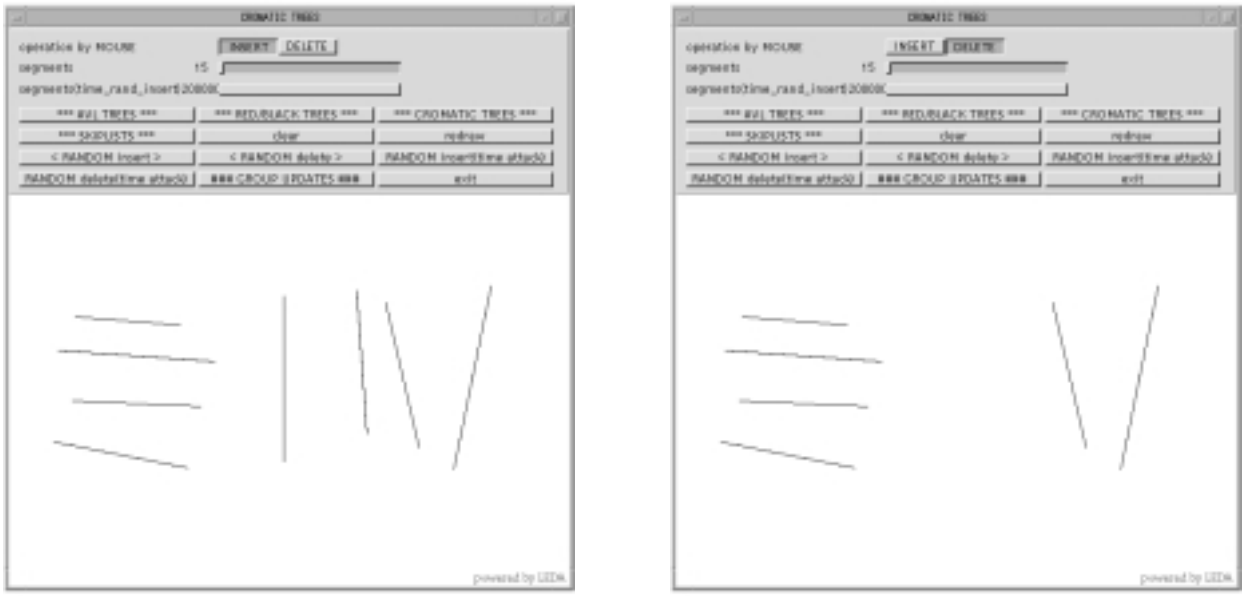
```
group updates
size = 4
                bal=1
           bal=0
                bal=1
      bal=1
                bal=1
           bal=0
                bal=1
```

```
sub tree insert
size = 8
                bal=1
           bal=0
                bal=1
      bal=1
                bal=1
           bal=0
                                bal=1
                           bal=0
                                bal=1
                     bal=1
                           bal=1
                bal=-1
                           bal=1
                     bal=1
                           bal=1
under
size = 8
                bal=1
           bal=0
                bal=1
      bal=1
                bal=2
           bal=-1
                                bal=1
                           bal=0
                                bal=1
                     bal=1
                           bal=1
                bal=0
                           bal=1
                     bal=1
                           bal=1
```

```
over
w3
size = 8
                bal=1
           bal=0
                bal=1
      bal=1
                          bal=1
                     bal=1
                          bal=1
                bal=0
                          bal=1
                     bal=1
                          bal=1
           bal=-1
                     bal=1
                bal=1
                     bal=1
under
size = 8
                bal=1
           bal=1
                bal=1
      bal=1
                          bal=1
                     bal=1
                          bal=1
                bal=0
                          bal=1
                     bal=1
                          bal=1
           bal=0
                     bal=1
                bal=1
                     bal=1
```

```
rb2
size = 8
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
     bal=1
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
result
size = 8
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
     bal=1
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
```

4.1.2 グループ削除の様子



(a) 削除前　　(b) 削除後

図 4.3: グループ削除

緩和二色木の構造の変化を次に示す.

```
group updates
size = 8
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
     bal=1
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1

sub tree delete
size = 6
               bal=1
          bal=1
               bal=1
     bal=1
                    bal=1
               bal=1
                    bal=1
          bal=0
                    bal=1
               bal=1
                    bal=1
```

4.2 実験目的

グループ更新は, 理論の上では従来の手法よりも効率が良くなっていることが確かめられたが, 実際にツールとして使ったときに従来よりも実行速度が短縮されているかどうか分からない.

この実験の目的は, グループ更新のデータ構造を実装し, 実際に理論通り動いているか確認することである.

4.3 実験環境

本研究で用いた実験環境は以下の通りである.

計算機	:	Sun Ultra 60
OS	:	Solaris 7
CPU	:	2 × 450 MHz UltraSPARC-II, 2 次キャッシュ 4MB
メモリ	:	512MB

4.4 実験方法

全体の動きや特性を見るため, ランダムな線分を入力し, その時間を測定する. また, いくつか線分を前もってデータ構造に蓄積しておき, そこに線分を挿入する, またはそこから線分を削除する場合についても測定する.

線分の大小比較は以下の優先順位で行った.

(1) 左端点の x 座標

(2) 左端点の y 座標

(3) 右端点の x 座標

(4) 右端点の y 座標

4.5 実験結果

4.5.1 データの挿入に関する CPU 時間の比較

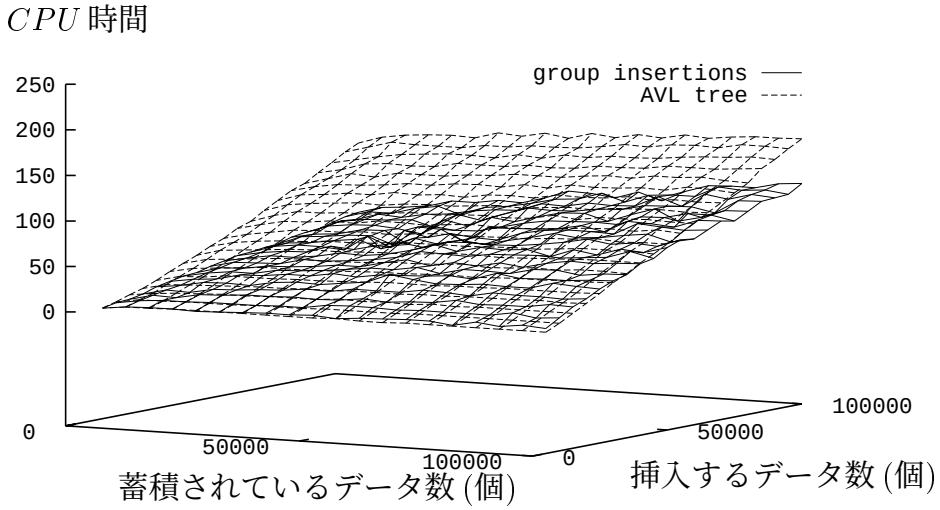


図 4.4: グループ挿入と AVL 木との CPU 時間の比較

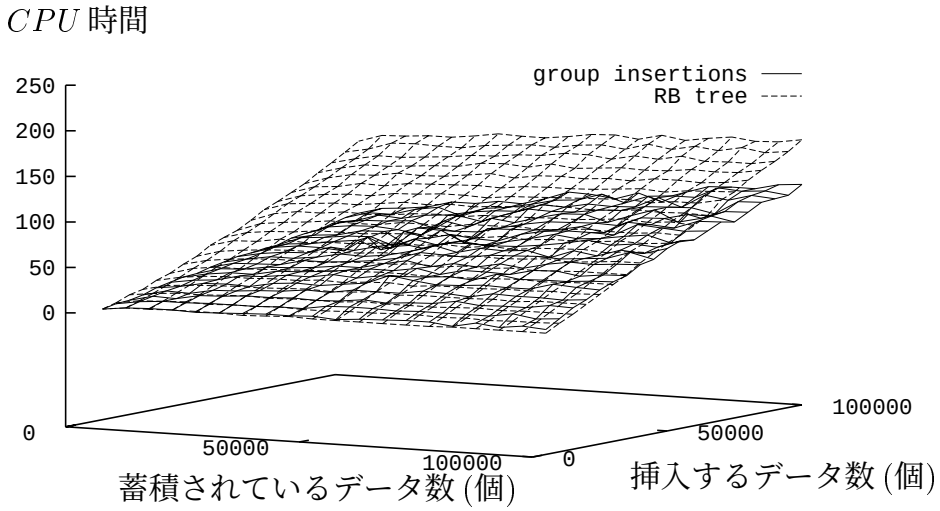


図 4.5: グループ挿入と二色木との CPU 時間の比較

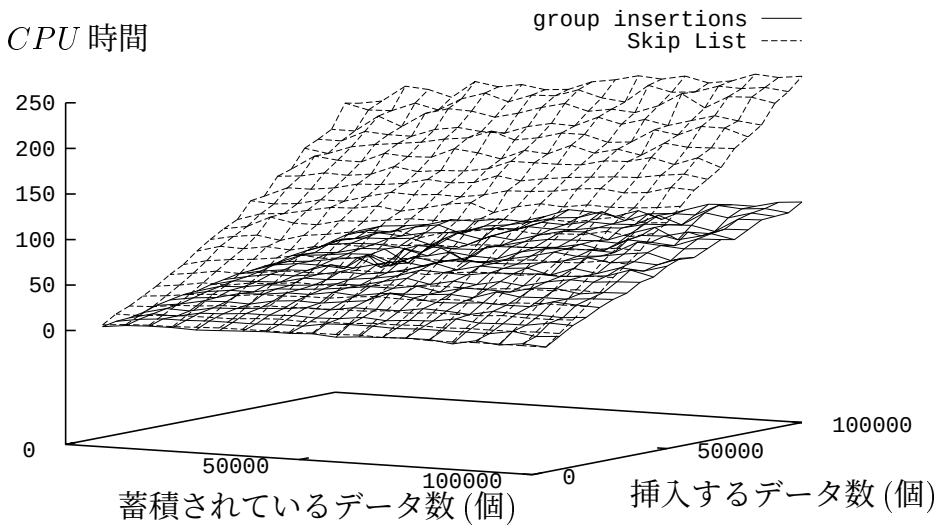


図 4.6: グループ挿入とスキップリストとの CPU 時間の比較

4.5.2 データの削除に関する CPU 時間の比較

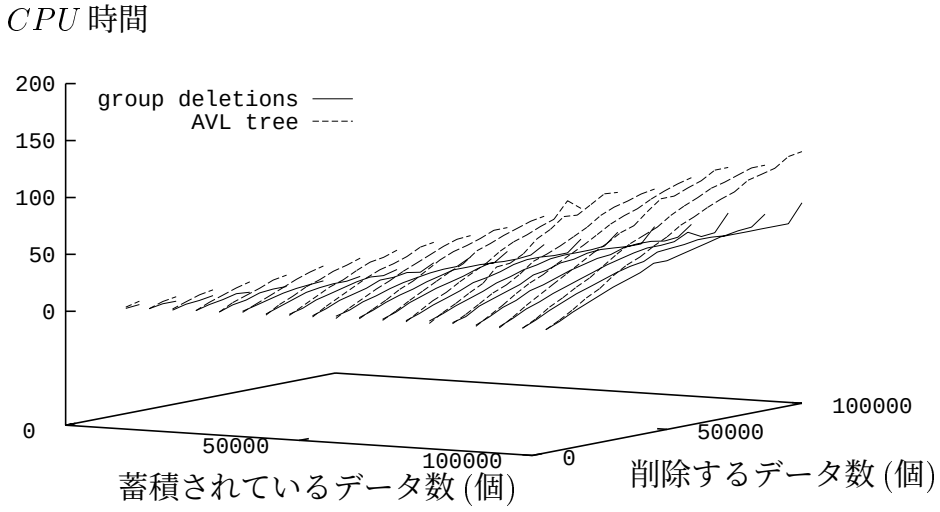


図 4.7: グループ削除と AVL 木との CPU 時間の比較

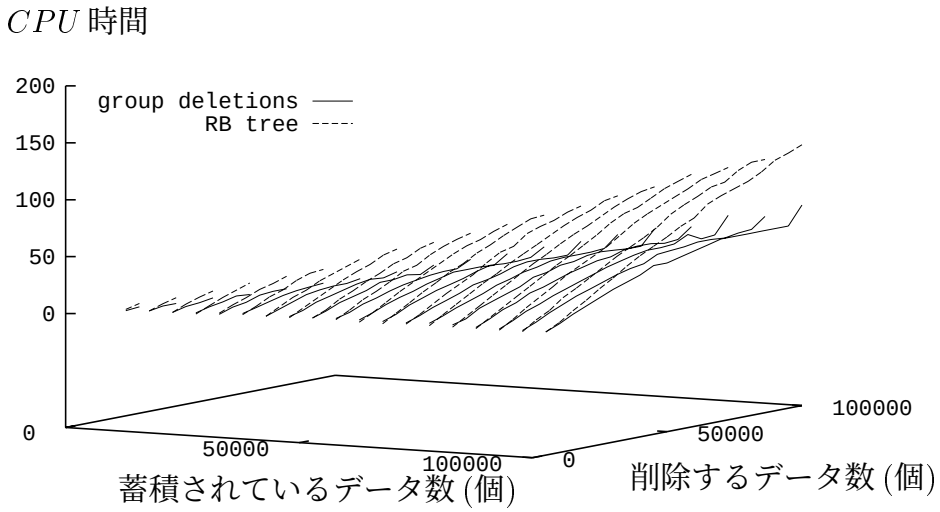


図 4.8: グループ削除と二色木との CPU 時間の比較

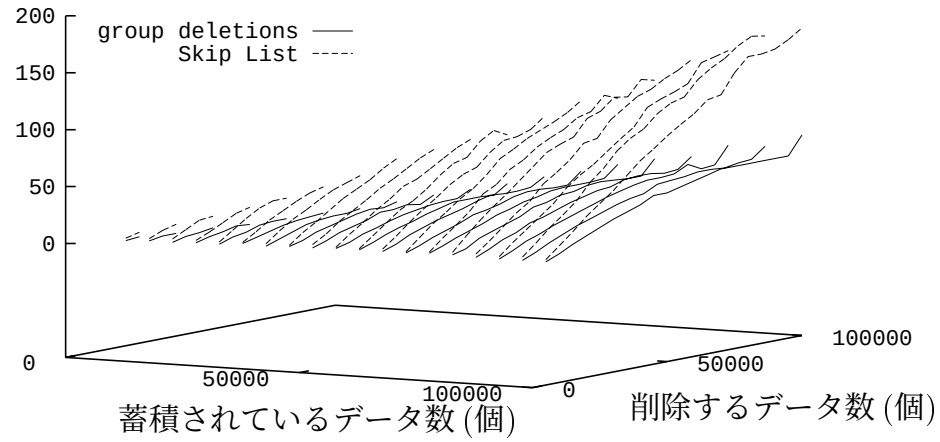


図 4.9: グループ削除とスキップリストとの CPU 時間の比較

4.5.3 二色木の更新にかかる CPU 時間

図 4.10 はデータがそれぞれ蓄積されている二色木に, データ 50000 個を挿入した CPU 時間を測定した結果である.

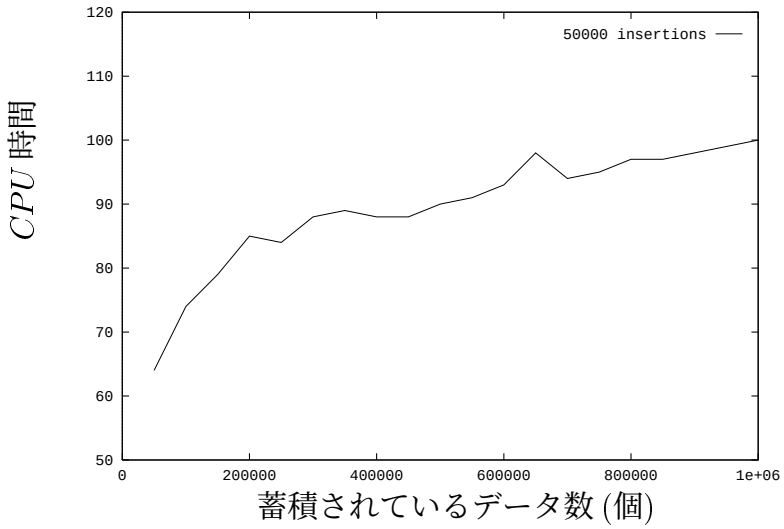


図 4.10: 一定データ数挿入の CPU 時間

図 4.11 はデータがそれぞれ蓄積されている二色木から, データ 50000 個を削除した CPU 時間を測定した結果である.

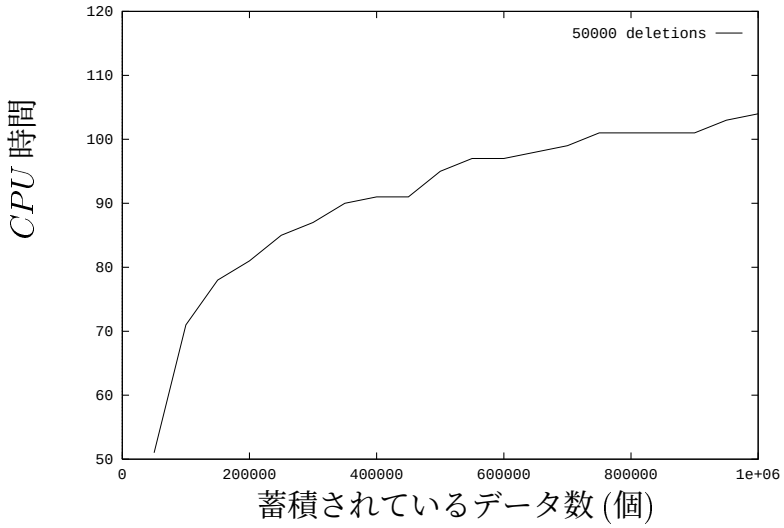


図 4.11: 一定データ数削除の CPU 時間

4.5.4 グループ挿入の分割ブロック数に関する結果

以下の図 4.12 は, メインに蓄積されているデータ数 100000 個に対し, それぞれの分割ブロック数になるように 10000 個のデータをグループ挿入したときの CPU 時間のグラフである. CPU 時間 6 の点線は, 従来の手法による二色木の蓄積データ数 100000 個にデータ 10000 個を挿入したときの時間である.

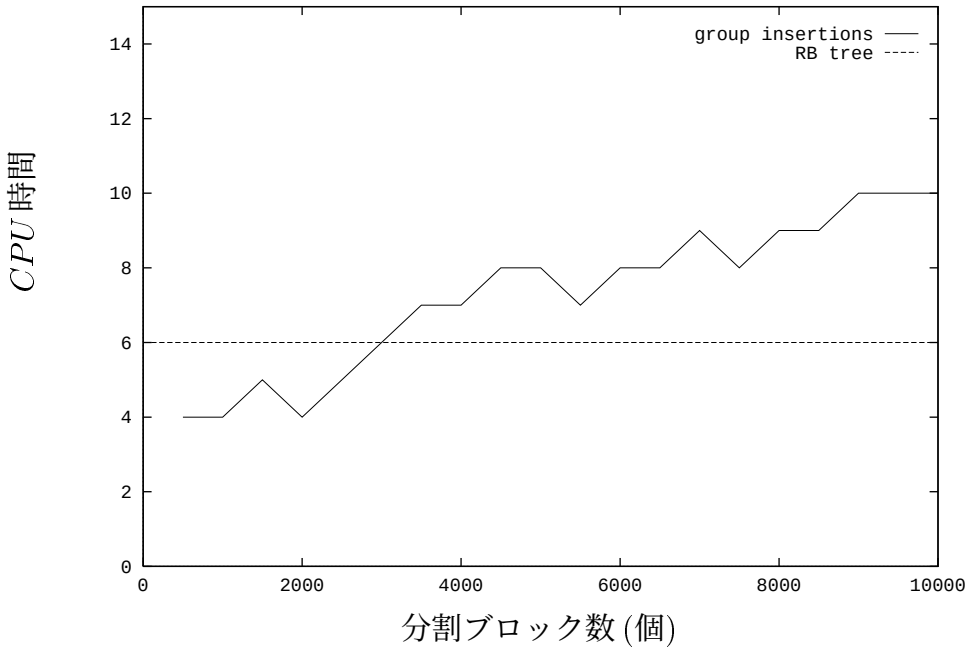


図 4.12: 分割ブロック別の CPU 時間と二色木の比較

以下の図 4.13 は, グループ挿入の各分割ブロック数を示した図である.

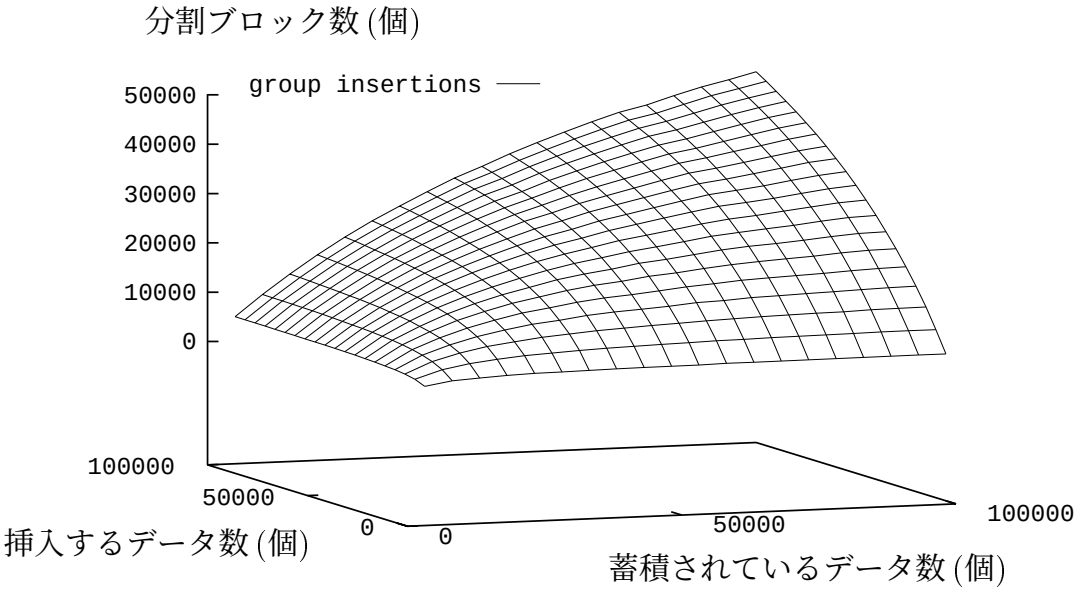


図 4.13: グループ挿入の分割ブロック数

以下の図 4.14 は, 蓄積されているデータ数 100000 に対して各データ数をグループ挿入した結果であり, 分割ブロック数を変化させ, 従来の手法とほぼ同じになる境界線を示したものである. その線より下側だとグループ挿入の方が効率が良いことを示している. 注意として, この結果は**各ブロックのデータ数を同じ**にして行っている.

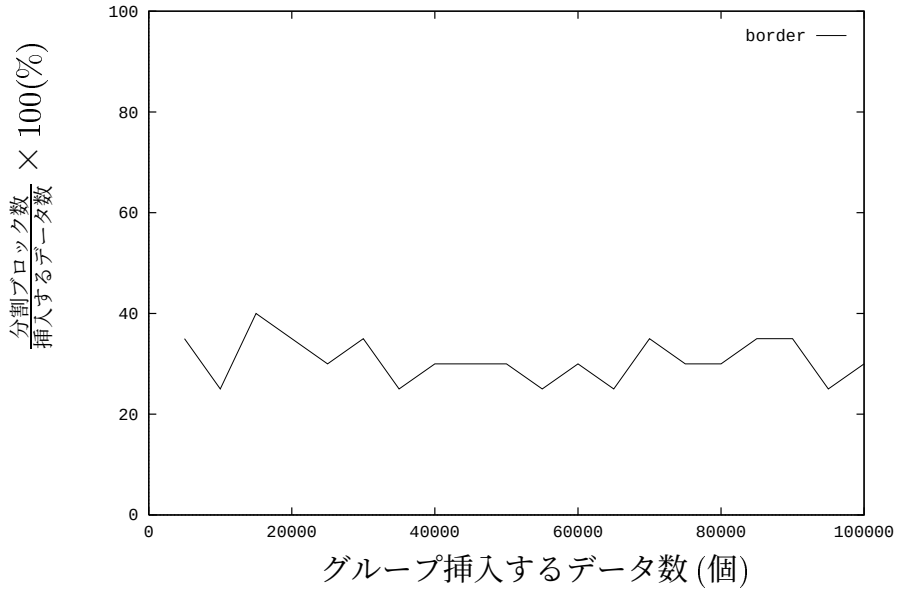


図 4.14: ブロックのデータ数一定でのグループ更新の効率の境界

4.5.5 グループ削除の部分全二分木数に関する結果

以下の図 4.15 は, グループ削除の各部分全二分木数を示した図である.

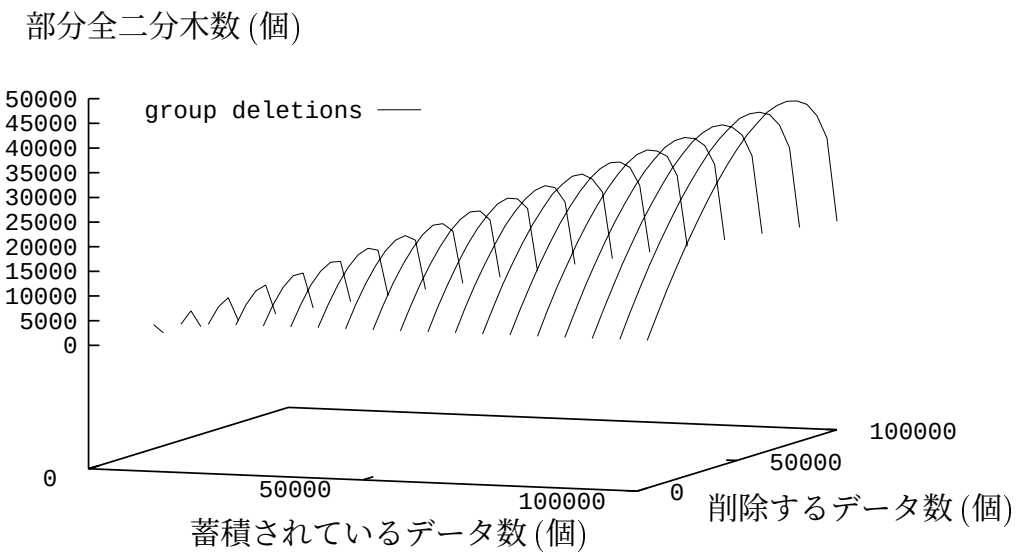


図 4.15: グループ削除の部分全二分木数

グループ削除で, 部分全二分木数と部分全二分木に含まれるデータ数を変化させたときの効率と, 従来の手法とを比較し境界を求める実験をしたが, 部分全二分木の性質上, 含まれるデータ数を変化させると部分全二分木数まで変化してしまうのでうまく実験することができなかった.

以下の表 4.1 は, 蓄積されているデータ数 100000 個にデータ 100000 個をグループ挿入したときの, 各ブロックのデータ数の内訳である.

表 4.1: 各ブロックのデータ数の内訳

ブロックのデータ数	2	3	4	5	6	7	8	9	10	～
ブロック数 (49871)	24968	12393	6118	3210	1625	748	406	208	107	88

以下の表 4.2 は, 蓄積されているデータ数 100000 個にデータ 95000 個をグループ削除したときの, 各部分全二分木のデータ数の内訳である.

表 4.2: 部分全二分木のデータ数の内訳

部分全二分木のデータ数	1	2	3	4	5	6	7	8	9	10
部分全二分木数 (34873)	14544	12004	205	**4816**	203	171	59	**1647**	139	159

11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
49	36	28	23	2	**406**	50	65	26	28	26	23	6	3	4	5	5	1	7

30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	～
3	2	**56**	11	7	5	3	7	7	7	0	2	0	1	2	3	4	1	2	10

4.6 考察

グループ挿入の結果から, すでに蓄積されているデータが少なく, これから挿入しようとするデータ数が多いといった場合に従来の手法とグループ挿入を比較すると, 他の場合よりも効率がかなり良くなっているといえる. これは入力データ数に比べて分割ブロック数が少ないためと考えられる. グループ挿入のときの分割ブロック数を表したものを図 4.13 に示す. さらに, グループ挿入の再平衡化は挿入された葉から根まで行われるが, そのパスの長さが短いということ, そのために本来必要だった再平衡化の数よりも少なくてすみ, 従来よりも処理時間が短縮されたと考えられる. 蓄積されているデータ数が多く, 挿入されるデータ数が少ない場合に, グループ挿入の効率が落ちていることが確認できた. これは, 各分割ブロック数のデータ数が少なくなってしまい効率が落ちてしまうと考えられる. その部分で, 分割ブロック数を変化させた実験を行った結果を図 4.12 に示す. この結果から, グループ挿入は挿入するデータ数に対する分割ブロック数の割合で大きく変化することが示された. どのくらいの割合ならば, 従来の手法よりも効率が良くなるのか実験した結果が図 4.14 である. この実験では**各ブロックに含まれるデータを均等**という条件で行った. その結果, 挿入するデータ数の 3 割程度の分割ブロック数になれば効率が上がることが示された. ランダムな入力線分で行った実験では, 各ブロックに含まれるデータ数は表 4.1 の様に均等ではない. この場合では, 入力データ数の半分の分割ブロック数でも従来よりも効率が上がっていることが示されている (図 4.5, 図 4.13).

グループ削除に関しては, すでに蓄積されているデータと同数個を削除する場合に, グループ削除の他の場合と比べてやや効率が悪くなっていることが分かった. 実装したツールでは, 蓄積されているデータ数と同数個のデータがグループ削除されるときには, データ数が 0 になることがわかるのでグループ削除を行わず, 緩和二色木をデータ数の初期状態に戻すと言った操作をする. 今回の実験のためにその操作を行わないようにするため, 蓄積データ数を 1 つ増やして行った. その 1 つのデータが削除されないため, 処理が遅くなったと考えられる. グループ削除のときに一度にどれくらいのデータが削除されているのかを表 4.2 に示す. この結果から, 2 の巾乗のデータ数が多く一度に削除されていることが分かる. これは, グループ削除の場合は削除されるデータを葉にもつ部分全二分木でなければならないためであるといえる. そのため, 削除されないデータがあると多くのデータを含む部分全二分木を選択できなくなり, 効率が落ちると考えられる. 同数個のデータをグループ削除する場合, 部分全二分木の数は減るが, 含まれるデータ数は 1 つか 2 つがほとんどであるため効率は落ちている. 図 4.15 から, 蓄積されている半数以上のデータをグループ削除するときに部分全二分木が減っていることが分かる. これは部分全二分木に

含まれるデータ数が増えたためといえる. そのため, 従来の削除との時間の増加率が変化していることが, 図 4.7, 図 4.8, 図 4.9 から分かる.

スキップリストは, 平衡探索木と同等の能率をもったデータ構造であるが, 平衡探索木に比べると処理に時間がかかっていることが分かった. スキップリストは乱数を用いてリストのレベルを決め, 平均的に平衡探索木と同等の能率を保っている. 平衡探索木ほど完璧な平衡は保てないということ, 挿入と削除でのリストのポインタの張り替えは, そのリストのレベルまでのポインタをすべて張り替える必要があり, リストのポインタの張り替えに時間がかかり処理が遅れたと考えられる. 実際には, これらの時間が平衡探索木の再平衡化よりも時間がかかると考えられる (図 4.6).

図 4.4, 図 4.5, 図 4.7, 図 4.8 から, AVL 木と二色木については, ほとんど差がみられなかった. AVL 木の削除の場合, 蓄積されているデータ数を n とすると, $\log n$ 回の再平衡化が行われる. しかし, 実際には毎回これだけの手間がかかるのではなく, 最悪の場合にこれだけかかるのである. この実験結果から, AVL 木で再平衡化が $\log n$ 回必要とされる場合は多くないと考えれれる.

以下にグループ更新についての考察をまとめる.

- グループ挿入で都合の良い場合

  実験の結果から, すでにメインの緩和二色木に蓄積されているデータ数が少なく, これから挿入しようとするデータ数が多い場合に従来の手法とグループ挿入を比較すると, 他の場合よりも効率がかなり良くなっているといえる. この場合, 緩和二色木の葉 1 つに対して置き換わる二色木が蓄えているデータ数は多くなる場合が出てくる. これは入力データ数に比べて分割ブロック数が少ないということを意味している. 1 つのブロックにたくさんのデータがある場合には, 葉と二色木を置き換えるときに葉であった節点の重みが小さくなる. その場合, 二色木を挿入した後の再平衡化はたくさん行われることになる. グループ挿入の再平衡化は挿入された葉から根まで行われるが, 緩和二色木のデータ数が少ない場合には, そのパスの長さは短く, そのため再平衡化の処理は根に達した時点終了し, 本来必要とされる再平衡化の数より少なくてすむと考えられる. パスが短いということは, 探索にかかる時間も少ないといえる. 以上の理由により, 他と比べて効率が大きく向上したといえる.

- グループ挿入で都合の悪い場合

  実験結果からグループ挿入の効率が悪いところは, すでに蓄積されているデータ数が多く, これから挿入するデータが少ない場合である. この場合, 分割ブロックの数が多くなり, 各ブロックで形成する二色木に蓄えるデータ数は少なくなる. 最悪

の場合, 分割ブロック数は挿入されるデータと同じ数になってしまう. この場合, グループ挿入ができるかどうか確認する処理をし, 従来よりも処理が遅くなってしまう. 仮に二色木に蓄えるデータが少し多い場合でも, 緩和二色木に蓄えられているデータ数が多いため, 二色木を挿入する葉から根までのパスの長さが長く, 再平衡化は, 途中で終了することなく最後まで行われる可能性が高く, 処理に多くの時間を費やすことになる. 以上の理由より, 他と比べて効率悪くなるといえる.

- グループ削除で都合の良い場合

  グループ削除で効率が良くなる場合は, 削除する多くのデータをすべて葉にもつように, 緩和二色木で全二分木を構成できる場合である. グループ削除では, 削除する全二分木の数だけ再平衡化を行うため, 削除する全二分木の数が少ない場合には, さらに効率がよくなる. 今回の実験では, 全ての範囲において従来の方法よりも処理時間が短くなっていることが示された. グループ削除では, 多数のデータを含んだ全二分木を削除した後でも, 従来の方法でデータ 1 つを削除するときと同じ再平衡化の手間だけしか必要にならない. そのために, すべての範囲で効率が良くなったと考えられる.

- グループ削除で都合の悪い場合

  今回用いたグループ削除のアルゴリズムでは, 削除を行うために緩和二色木から全二分木を選択し, さらにその葉すべてが削除されるデータでなければならない削除できない制約がある. そのために, 削除されるデータの連続性に大きく左右される. 今回の実験では, すでに蓄積されているデータと同数個を削除する場合に, グループ削除の他の場合と比べてやや効率が悪くなっていることが示された. 実装したツールでは, 蓄積されているデータ数と同数個のデータがグループ削除されるときには, データ数が 0 になることがわかるのでグループ削除を行わず, 緩和二色木をデータ数の初期状態 (データ数 0 個) に戻すと言った操作をする. 実験のためにその操作を行わないようにするように, 蓄積データ数を 1 つ増やして行った. その 1 つのデータが削除されないため, 処理が遅くなったと考えられる. 削除されないデータがあると, 広範囲にわたる全二分木を選択できなくなるため, 部分全二分木は小さく分割される. その数だけ, 再平衡化の時間がかかるので, 他の場合と比べて効率が落ちたと考えられる.

  また, 削除されるデータ数とが緩和二色木で構成される部分全二分木の数が同じになってしまう場合もある. その場合には, グループ削除の従来の方法よりも処理が遅れると考えられる.

以上のことからグループ挿入の効率に関しては, 緩和二色木の高さ, 分割ブロック数, ブロックに含まれるデータ数の 3 つのうち 1 つでも変化すると効率も変化するといえる.

グループ削除に関しては, 再平衡化は 1 つ部分全二分木に対して高々3 回の単回転ですむので, 緩和二色木の高さには関係ない. 関係するのは, 部分全二分木数と部分全二分木に含まれるデータ数である. この 2 つが変化することでグループ削除の効率が変化するといえる.

第 5 章

まとめ

5.1 実験結果の評価

本研究で提案したデータ構造で評価しなければならないのは次の 2 つである. 一時的な蓄積部分を用いたときの処理の違いについてと, グループ更新の処理についてである.

今回は, 一時的に蓄積する部分のデータ構造として二色木を用いた. 実験結果の図 4.10 からわかるように, 二色木の挿入に関しては蓄積されているデータの数が増えると処理の時間も多くかかっていることが分かる. 普通, メインのデータ構造に蓄積されているデータ数の方が多くなると考えられるので, 従来のように直接メインに挿入するよりは, 蓄積されているデータ数が少ない一時的に蓄積する部分に挿入した方が処理が速くなることが分かる. 削除の場合は, 挿入の場合とは少し違う. 一時的に蓄積する部分にあるデータを削除する場合には, 図 4.11 を見て分かるとおり, データ数が多いメインのデータ構造から削除する場合に比べてデータ数の少ない一時的に蓄積する部分から削除した方が処理時間が短くてすむ. 一時的に蓄積する部分に削除するデータがない場合は, メインのデータ構造を探索することになる. そこで, データがあることが確認できれば, そこで削除を行わず, 削除データを一時的に蓄積する部分に蓄積する. この場合を従来との削除と比較すると, 探索時間はあまり変わらず, メインを削除する変わりに一時的に蓄積する部分に挿入することになる. 図 4.10 と図 4.11 から, 削除と挿入にかかる再平衡化の時間はどちらも変わらないので, 処理時間はほぼ同じといえる. 以上にことから, 一時的な蓄積部分を用いると, 従来の方法よりも処理が速くなると評価できる.

次にグループ更新を従来の更新と比較して評価を行う. 各データ構造の従来の挿入と今回のグループ挿入を計算時間で比較し評価する. 図 4.12 から分割ブロック数が挿入するデータ数に近い値になると, 従来の手法よりも処理に時間がかかってしまうことが分かる.

実際の CAD で大量にデータを扱う場合, 図面の一部分に集中してデータの挿入をすることが多い. つまり, 分割ブロック数が挿入するデータの数に近い場合はあまり起らないことから, 実際にはあまり影響が少ないと考えられる. 図面の一部分に集中してデータの挿入をする場合は, 分割ブッロック数が挿入するデータ数よりも小さくなる. 分割ブッロック数が挿入するデータ数よりも小さくなる場合, 実験結果から従来の削除の手法よりも良い結果が得られたので, グループ削除を用いることで処理時間が短縮されたと評価できる.

以上ことから, 従来の更新よりもグループ更新を用いた方が処理時間が短縮され効率良く図形を描画できると評価できる.

5.2 グループ更新を用いる利点

実験から従来の単純な更新操作よりも高速に処理できることが示されたので, 従来よりは効率良くデータの更新を行うことができる. また, グループ更新ではメインのデータ構造と一時的な蓄積部分の 2 つに分けているため, あるデータが入力され, すぐに削除の操作が行われた場合, 一時的な蓄積部分内でのデータの相殺が可能となる. 更新操作の入力中は, メインのデータ構造を更新せず一時的に蓄積するため, 従来よりも処理が短縮されて入力操作がスムーズに行えるといった利点がある.

実際の CAD では, 描画する部品毎にすでに用意されいる CAD データを取り込む操作をすることも多い. そのときには本研究で実験したグループ更新を用いると素早く処理できることになる.

5.3 まとめ

本研究では, 人間による図形を描画する際の特性を利用し, 更新操作の入力中は, メインのデータ構造を更新せず一時的に更新データを蓄積しておき, 思考時間になったらグループ更新を行い効率良く図形を描画できるデータ構造を提案した. またこのデータ構造が実際にそのように動作するのか実装して実験を行った.

すでに蓄積されているデータ数を N とし, 挿入または削除を行うデータ数を m とする. グループ更新をするデータが $O(m \log m)$ の時間をかけてソートされているとすると表 5.1 のようになる. ここで, 挿入の場合の b は分割ブロック数を表し, 削除の場合の b は削除される部分全二分木数を表している.

一時的に蓄積部分を利用することで, 入力操作にかかる処理時間が短縮されていることを示した.

表 5.1: グループ更新と従来の更新との比較

	従来の更新	グループ更新
挿入	$O(m \log N)$	$O(b \log N + \sum_{i=1}^{b} m_i \log m_i)$
削除	$O(m \log N)$	$O(b \log N + m)$

グループ挿入に関しては, すでに蓄積されているデータ数が多く, グループ挿入するデータ数が少ない場合に従来の手法より効率が落ちていることが確認された. これは分割ブロック数が挿入するデータ数に近い値になってしまうことが原因である. 実験の結果, 各ブロックに均等にデータを含む場合は, データ数の 3 分の 1 程度以下のブロック数に分割されることで従来より効率が良くなることが分かった.

グループ削除に関しては, 削除する部分全二分木の数が削除するデータ数に近い値だと効率が悪くなるといえる. グループ削除をした後の再平衡化について, 従来の手法の 1 つのデータを削除したときと同じ再平衡化でよいので, 部分全二分木に多数の削除データが含まれる場合は処理時間が短縮されたことを示した.

実験の結果から, 実用的な範囲以内で従来のように 1 つずつ更新を行っていた手法よりも, グループ更新を行った方が効率よく行えることが確認できた.

特に, CAD データをファイルなどから読み込んで挿入する場合に, 本研究で提案したグループ更新を用いると有用であると考える.

5.4 今後の研究課題

- グループ削除で部分全二分木数とそれに含まれるデータ数との効率の関係.
- スキップリストを用いたグループ更新の研究と開発.
- 並列処理が可能なデータ構造の実装と比較.
- 線分に限定せず他の幾何データへの対応.

グループ削除で, 部分全二分木数と部分全二分木に含まれるデータ数を変化させたときの効率と, 従来の手法とを比較し境界を求める実験をしたが, 部分二分木の性質上, 含まれるデータ数を変化させると部分全二分木数まで変化してしまうのでうまく実験することができなかった. 実験の方法を検討し直して, 従来の手法と比較しなければならない.

今回の実験で, スキップリストは他の平衡探索木より実際の処理時間がややかかるといった結果が出た. 今後の研究課題としては, 本研究では緩和二色木を用いてグループ更新を実現したが, スキップリストを用いても可能であると考えられるので, その手法について考え, 今回提案した手法と比較をすると興味深い結果がえられると考えられる.

緩和二色木は本来, 並列処理向きのデータ構造である. 今回実装したものはシングルプロセッサを前提としたものであるので, マルチプロセッサによる並列処理を行うためのツールを実装し, 比較することは大事なことであろう. 実装はやや複雑になるが, 実行結果にどのような違いがあるのかを確かめることはとても興味深いことである.

今回のデータ構造で扱った幾何データは線分のみであった. しかし, 図形を形成する要素は線分だけでなく, 点, 円, 多角形や曲線などの基本要素がそろっていると図形も描画しやすくなると考えられる. 図形の基本要素は形が異なるためデータ構造を工夫する必要があるが, 基本的には線分で構成できるので, 線分集合として蓄積し代表値をもたせるとよいのかもしれない. そして, 木の節点にさらに木を付加するといった構造をとることも考えられる. このような基本要素を効率良く蓄積するデータ構造が今後の課題となろう.

謝辞

本研究を進めるにあたり, 研究のきっかけを与えていただき, 多くのことを教えて下さった浅野哲夫教授に心から感謝致します. 東条 敏教授, 平石 邦彦助教授と宮地 充子助教授にも貴重なご意見をいただき有り難う御座いました. また, 小保方幸次助手には特に, C/C++のライブラリ LEDA に関して多くのことをご教授いただき有り難う御座いました. 最後に, 公私にわたりお世話になりました研究室, 並びに友人の皆様有り難う御座いました.

参考文献

[1] G. M. Adel'son-Vels'kii and E. M. Landis: "An algorithm for the organisation of information", Soviet Math. Dokl., 3:1259-1262, 1962.

[2] J. L. Bentley: "Algorithms for Klee's Rectangle Problems", Carnegie-Mellon University, Pittsburgh, Penn., Department of Computer Science, unpublished notes, 1979.

[3] K. Pollari-Malmi, E. Soisalon-Soininen, T. Ylonen: "Concurrency control in B-trees with bath updates", IEEE Trans. on Knowledge and Data Engineering 8:6, pages 975-984, 1996.

[4] M. Rossi: "Concurrent full text database", Master's thesis, Department of Computer Science, Helsinki University of Technology, Finland, 1997.

[5] T. Ylonen: "An algorithm for full-text indexing", Report TKO-B75, Helsinki University of Technology, Deparment of Computer Science, 1992.

[6] L. Malmi and E. Soisalon-Soininen: "Group updates for relaxed height-balanced trees", In Proc. 18th ACM Symposium on the Principles of Database Systems, pages 358-367, 1999.

[7] Sabine Hanke and E. Soisalon-Soininen: "Group updates for Red-Black Trees", Algorithms and Complexity 4th Italian Conference, pages 253-262, 2000.

[8] L. J. Guibas and R. Sedgewick: "A dichromatic frameworkfor balanced tree", In Proc. 19th IEEE Symposium on Foundations of Computer Science, pages 8-21, 1978.

[9] O. Nurmi and E. Soisalon-Soininen: "Chromatic binary search trees: A structure for concurrent rebalancing", Acta Informatica, 33:547-557, 1996.

[10] K. Larsen: "Amortized constant relaxed rebalancing using standard rotations", Acta Informatica, 35(10):859-874, 1998.

[11] J. Boyar, R. Fagerberg and K. Larsen: "Amortization results for chromatic search trees, with an application to priority queues", Journal of Computer and System Sciences, 55(3):504-521, 1997.

[12] W. Pugh: "Skip Lists: A Probabilistic Alternative to Balanced Tree", Proc. Workshop WADS '89, 1989.

[13] K. Larsen, E. Soisalon-Soininen and P. Widmayer: "Relaxed balance through standard rotations", In Proc. 5th Workshop on Algorithms and Data Structures, volume 1272 of LNCS, pages 450-461, August 1997.

[14] R. Bayer and E. M. McCreight: "Organization and maintenance of large ordered indexes", Acta Informatica, 1(3):173-189, 1972.