AVL trees

2020年度前期・高度データ構造

本日の内容

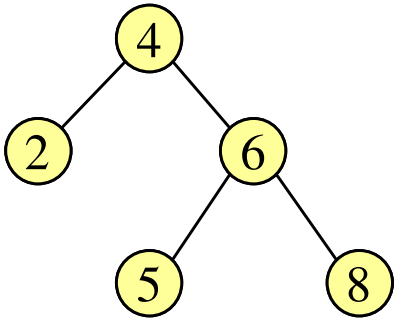
- 平衡探索木：要素の追加／削除を行っても
  高さを低く（要素数の対数程度に）保つことができる
  探索木
  - AVL tree

前回は、整数集合上の様々なクエリや操作を、2分探索木の高さ h に比例した時間で行えることを学びました。

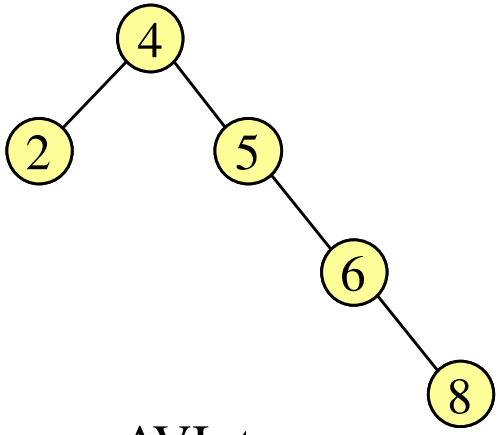
本日の講義では、要素の追加や削除を行っても、高さを低く保つことができる2分探索木
・AVL tree
について学びます。

AVL tree [Adelson-Velskii & Landis, 1962]

- 2分探索木 T のすべての頂点 v について，
  $\mathrm{bf}(v) \in \{-1, 0, 1\}$ を満たすとき，T を AVL tree とよぶ
  - $\mathrm{bf}(v) = v$ の左の子の高さ − v の右の子の高さ



AVL tree



non AVL tree

AVL tree は、Adelson-Velskii と Landis によって提案されたデータ構造です。
もしかすると、すでに知っている人もいるかもしれませんが、そういう人はおさらいだと思って読み進めていってください。

任意の頂点 v について、関数 bf を bf(v) = v の左の子の高さ − v の右の子の高さと定義します。
2分探索木 T のすべての頂点 v について bf(v) ∈ {−1, 0, 1} を満たすとき，T を AVL tree と呼びます。

左の図は AVL tree の一例です。どの頂点においても、左の子と右の子の高さの差が1以内に収まっています。

一方、右の図は AVL tree ではありません。例えば、根 r について bf(r) = -2 であるからです。

AVL tree の概要

- n 個の節点を持つ AVL tree の高さは $O(\log n)$
  ➔ member, min/max, predecessor/successor を $O(\log n)$ 時間で実現可能.
- AVL tree の性質を保ったまま,
  insert/delete を $O(\log n)$ 時間で実現可能.
  - 頂点のローテーションを行う.
- 以降, $\mathrm{bf}(v) \in \{-1, 0, 1\}$ を満たすとき,
  頂点 v はバランスされているといい,
  そうでないとき, v はバランスが崩れているという.

AVL tree はこのような便利な性質を持っています。

特徴的なのは insert と delete の方法です。前回の講義でやったように、素朴に要素を追加していくと、木の高さがどんどん高くなっていってしまします。(例えば、1, 2, 3, ..., n のように昇順に追加すると、高さが O(n) の木ができてしまう)

この問題を回避するために、AVL tree では、頂点のローテションと呼ばれるテクニックを使います

insertion on AVL tree

insert(3, S)

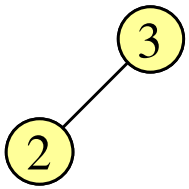
具体例で説明していきます。

まず空の木から始めて、3 を追加しました。このとき、根だけがあります。

insertion on AVL tree

insert(3, S)

insert(2, S)

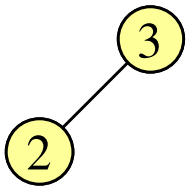


次に 2 を追加します。2 は 3 より小さいので、3 の左の子に 2 が来ます。ここまでは前回と同じです。どの頂点もバランスされているので、これ以上何もする必要はありません。

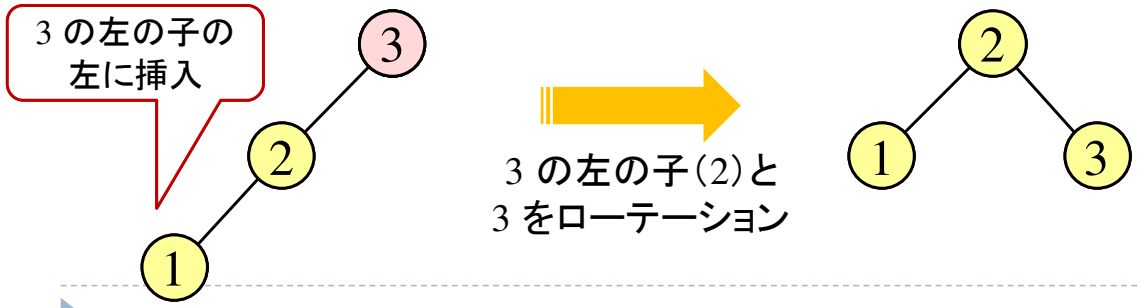
insertion on AVL tree

insert(3, *S*)

insert(2, *S*)



insert(1, *S*)

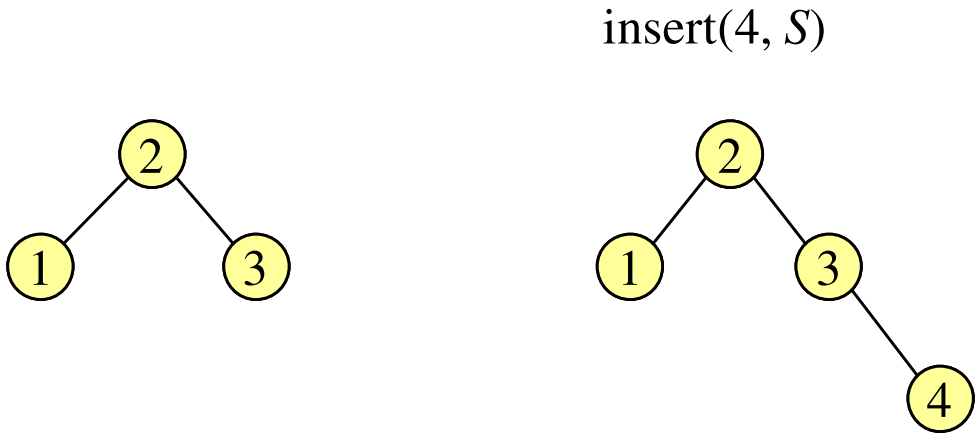


続いて、1 を追加します。すると、このピンクの頂点のバランスが崩れました。

このピンクの頂点から見て、新しい頂点 1 は「左の子の左」に挿入されています。
このような場合には、「3 の左の子(2)と 3 をローテーション」します。

すると、このようにバランスされた木が得られます。各頂点の値について、2 分探索木の性質もちゃんと満たされています。

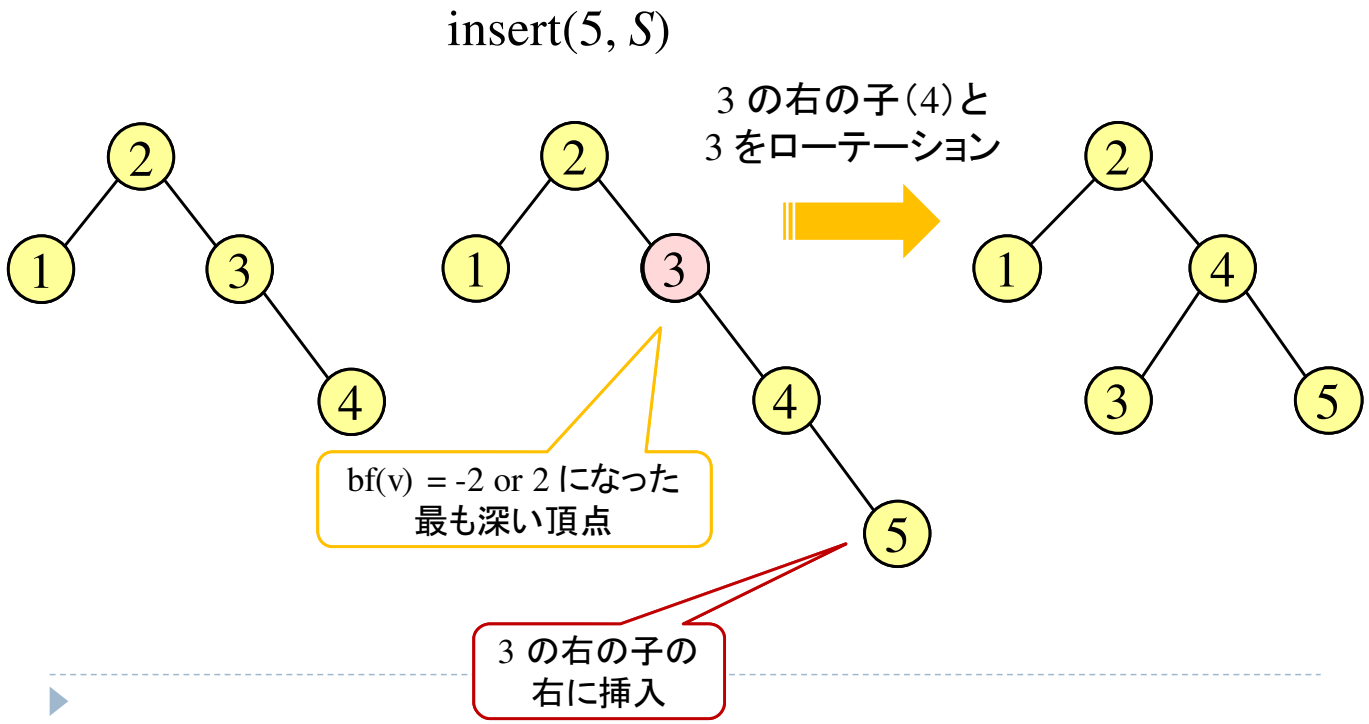
insertion on AVL tree



insert(4, *S*)

続いて4を追加します。すべての頂点はバランスされているので、このステップはこれで終わりです。

insertion on AVL tree



続いて 5 を追加します。

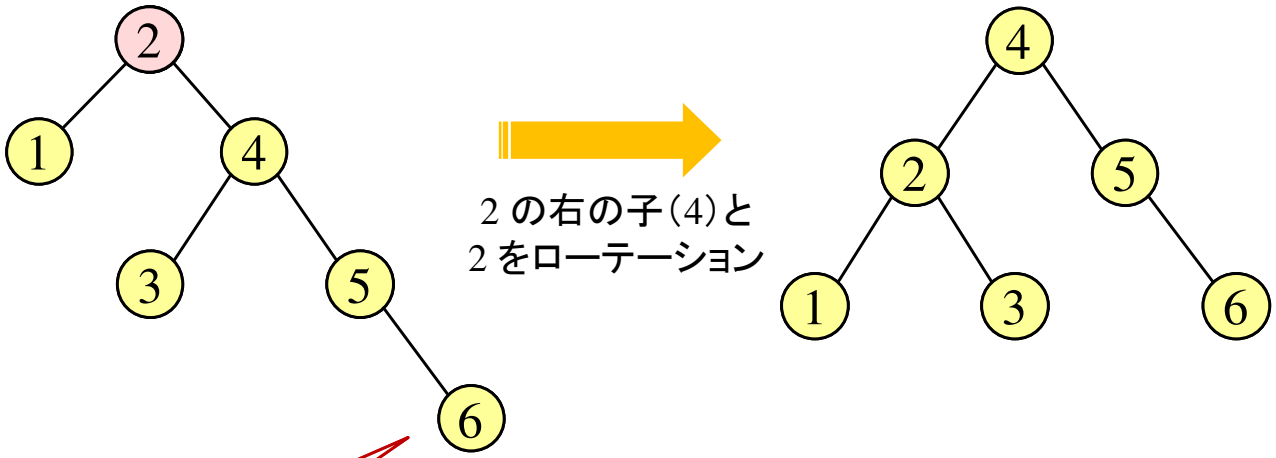
このとき、バランスが崩れました。ピンクの頂点は、バランスが崩れた最も深い頂点を示しています。

このピンクの頂点から見て、新しい頂点 5 は「右の子の右」に挿入されています。
このような場合には、「3 の右の子(4)と 3 をローテーション」します。

すると、このようにバランスされた木が得られます。各頂点の値について、2 分探索木の性質もちゃんと満たされています。
(例えば、2 の左には 2 未満の値、右には 2 を超える値が格納されている。)

insertion on AVL tree

insert(6, S)



2 の右の子(4)と
2 をローテーション

2 の右の子の
右に挿入

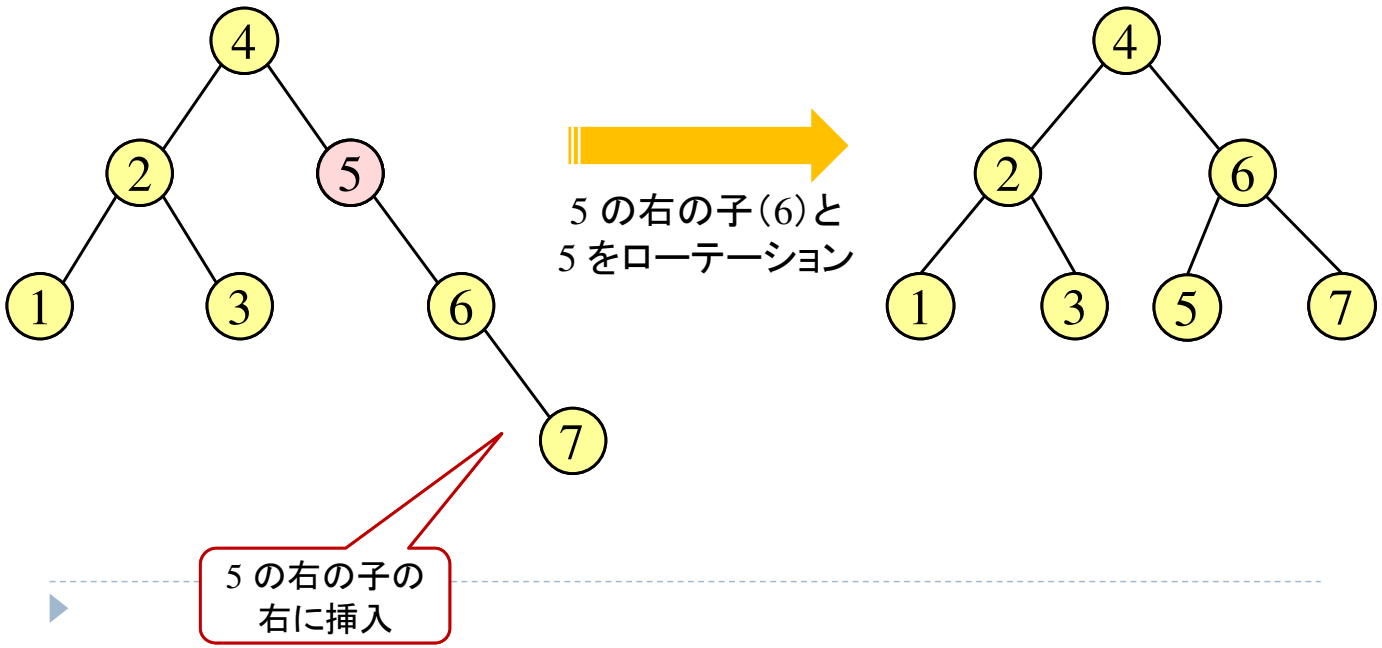
次に 6 を追加します。

これも、ピンクの頂点(バランスが崩れた最も深い頂点)から見て、右の子の右に挿入されたパターンなので、2 の右の子(4)と 2 をローテーションします。

これで、バランスされた木が得られました。

insertion on AVL tree

insert(7, S)



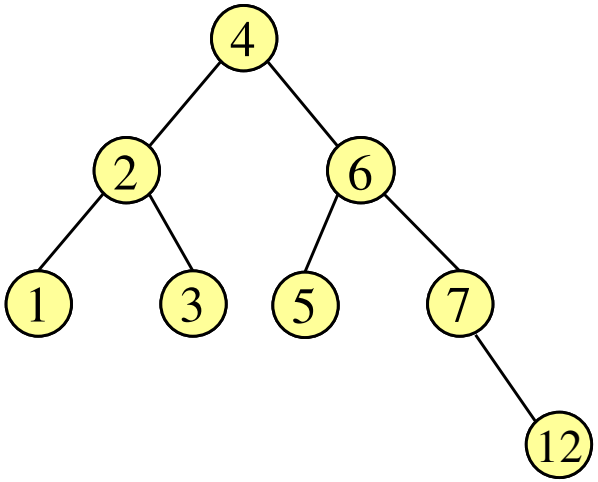
次に 7 を追加します。

これも、ピンクの頂点（バランスが崩れた最も深い頂点）から見て、右の子の右に挿入されたパターンなので、5 の右の子(6)と 5 をローテーションします。

これで、バランスされた木が得られました。

insertion on AVL tree

insert(12, *S*)

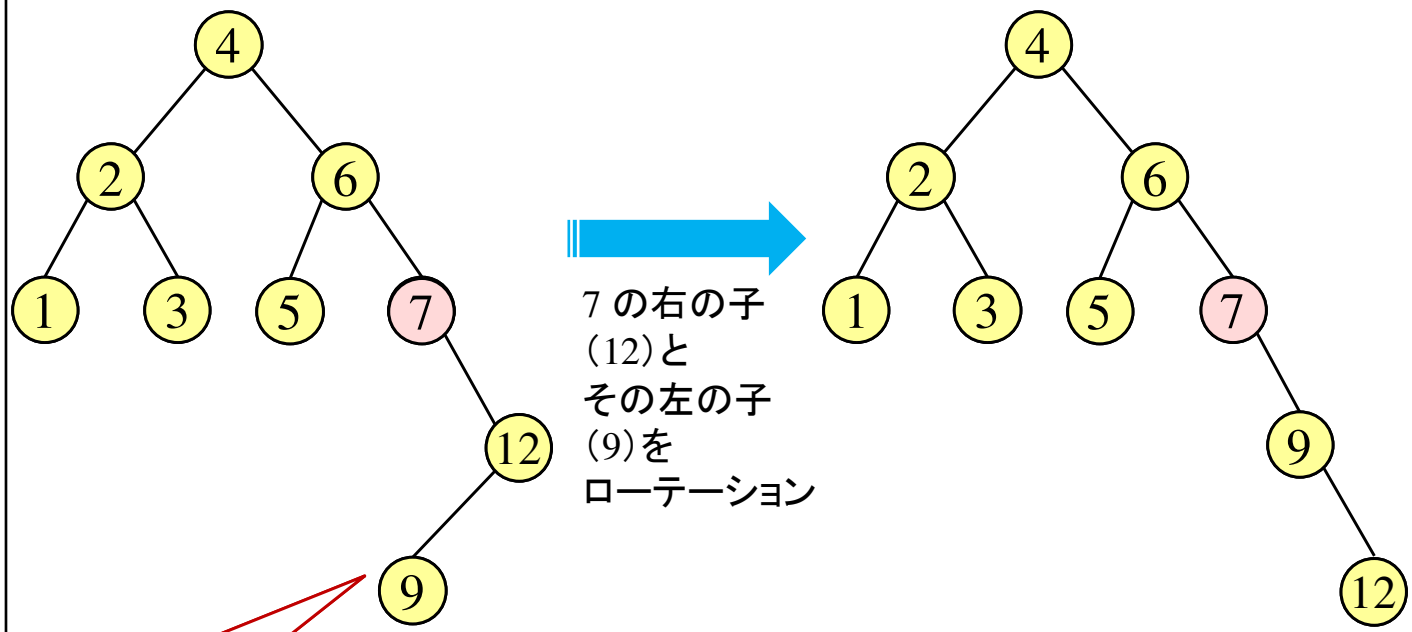


次に 12 を追加します。

バランスは保たれたままなので、このステップはこれで終わりです。

insertion on AVL tree

insert(9, *S*)



7 の右の子
(12)と
その左の子
(9)を
ローテーション

7 の右の子(12)の
左に挿入

次に 9 を追加します。

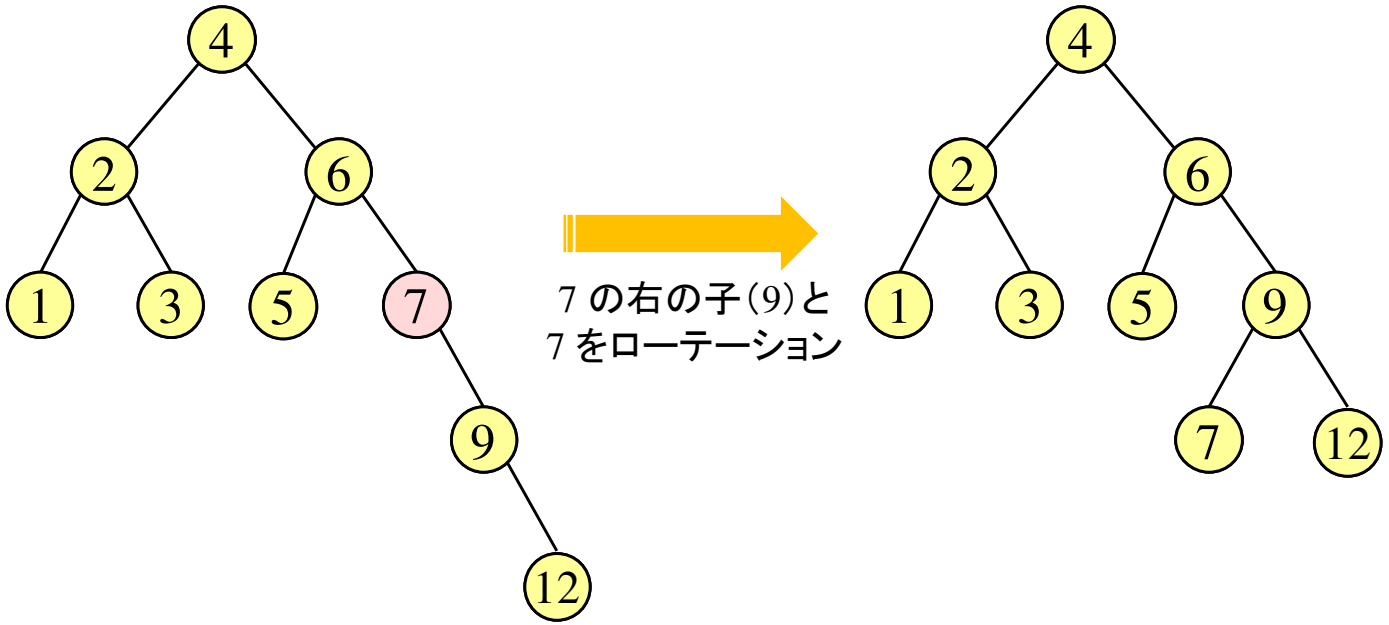
ピンクの頂点(バランスが崩れた最も深い頂点)から見ると、「右の子の左」に新しい頂点が挿入されました。これは今までになかったパターンです。この場合は、頂点のローテーションを2回行います。

まず、「7 の右の子(12)とその左の子(9)を ローテーション」します。すると、右図のような木が得られます。この木はまだバランスされていませんが、形だけ見てみると、先ほど出てきた「右の子の右」パターンと似た形をしていることがわかると思います。

(次ページに続く)

insertion on AVL tree

insert(9, S) の続き

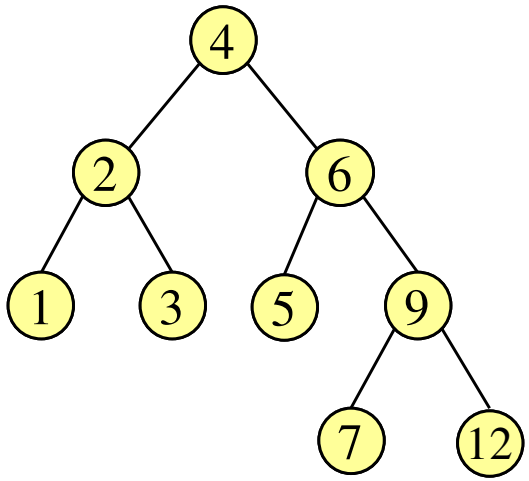


そこで、「7 の右の子(9)と 7 をローテーション」します。すると、このようにバランスされた木が得られました。各頂点の値についても、2分探索木の性質を満たしています。

以上が、実例を使った AVL tree 上での要素の追加 (insert) のアルゴリズムの解説です。

演習問題

- 以下のAVL tree に insert(8, *S*) を行った後の AVL tree を示せ



※ 演習問題**提出〆切：５月２８日（木）２３：５９**
pdf の作成方法は次ページを参照してください

では、本日の演習問題です。

図の AVL tree に新たな要素 8 を追加したのちの AVL tree を図示してください。

結果の AVL tree だけを描くのではなく、ローテーションの様子がわかるように、途中の状態も図示してください。

図はパワーポイント等で作成しても構いませんし、手書きで描いたものをスキャンしたり撮影して図にしても構いません。撮影する場合は、図がなるべく見えやすくなるようにしてくれると助かります。

(5/25 追記) なお、pdf の作成方法について、次ページを参照してください。

演習問題の pdf について (5/25追記)

5/25 以降に演習課題を提出する人は、以下の要領に
従って pdf を作成してください。

- ファイル名は 学籍番号_lastname.pdf とする。
  例）2IE00099Z_inenaga.pdf
- pdf の1ページ目の最初に
  「第XX回 演習課題」「学籍番号」「氏名」を明記する。
- 留学生は英語で作成しても構いません。
  Foreign students may write their answers in English
  if they feel it easier and more comfortable.

なお、すでに提出が完了している人は、再提出する必要は
ありません (成績に不利益を被ることはありません)

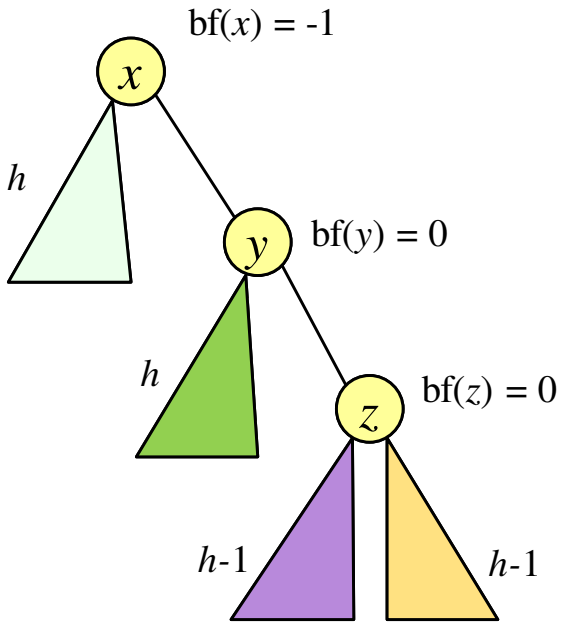
(5/25 追記)

演習問題の pdf を作成する際には、この要領に従ってください。

また、次回以降も、同様の要領で提出してください。

ローテーションのパターン１

- “右－右”パターン （“左－左”パターンも同様 ）



AVL tree

ここから先は、なぜこのようなローテーションで上手く AVL tree をメンテナンスしていけるのかを、一般的に解説していきます。

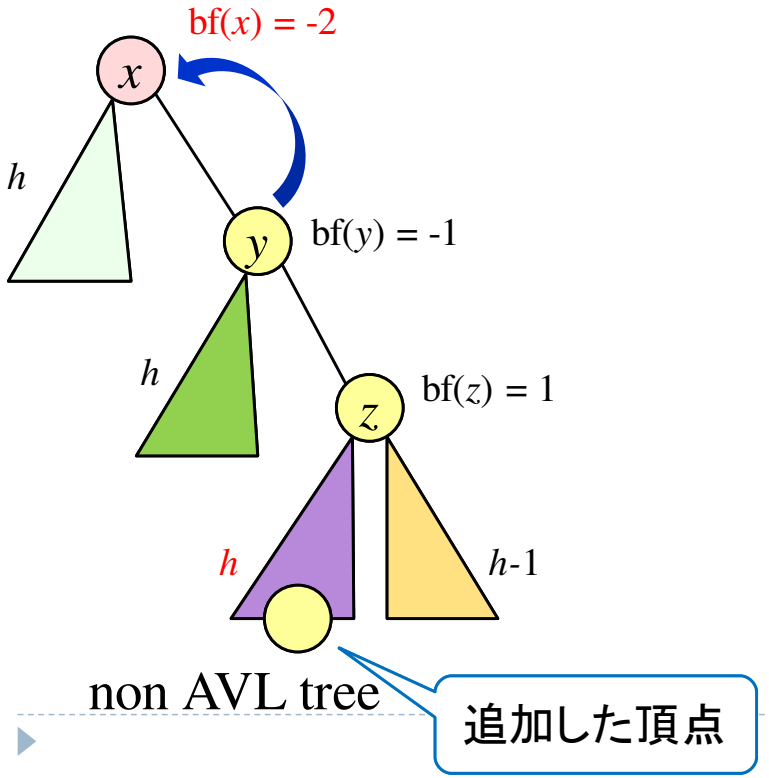
バランスが崩れた頂点の右の子の右側に新しい頂点が挿入されるパターンを”右ー右”パターンと呼ぶことにします。

この図では、h や h-1 は、それぞれの部分木の高さを示しています。いま、この概念図では、それぞれの頂点 x, y, z はバランスされています。

(次のスライドへ)

ローテーションのパターン１

- “右－右”パターン （“左－左”パターンも同様 ）

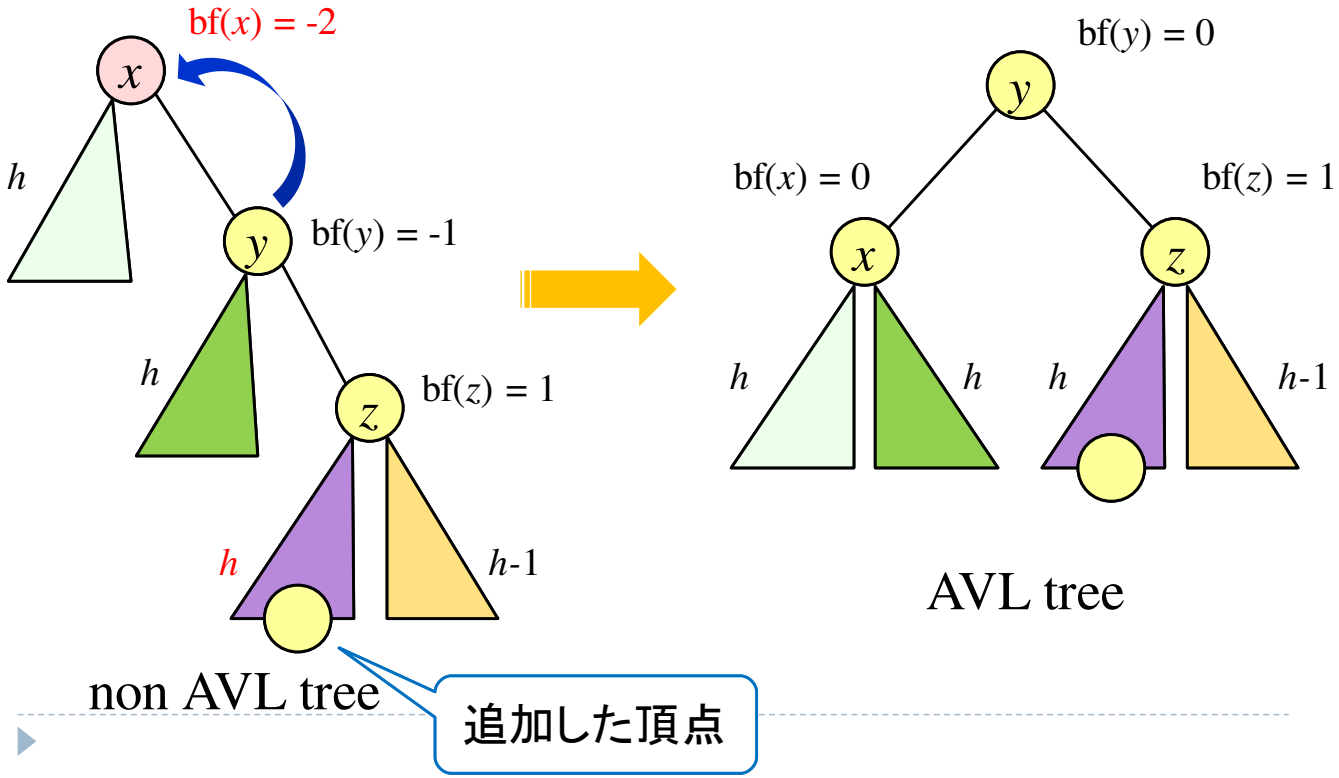


ここで、z の下に頂点が新たに追加された場合を考えます。

z の左の子の高さが h-1 から h になったことで、x のバランスが崩れました。
そこで、x と y をローテーションします。

ローテーションのパターン１

- “右－右”パターン （“左－左”パターンも同様 ）



ローテーション後の木は右図のようになります。x, y, z すべての頂点がバランスされています。

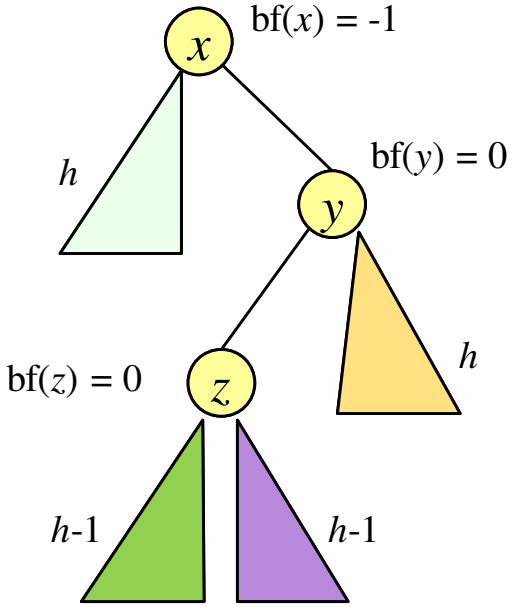
x と y が入れ替わり、y の左の部分木（緑）が x の右の部分木に付け替わりました。

2分探索木の性質から、緑の部分木の中に格納されている値は、すべて x より大きいことが保証されていますから、x の右の子に緑の部分木が来ても、大小関係は保たれています。

同様に、y から見ても、緑は元々左側にありました。ローテーション後も y の左側にあるので、大小関係は保たれています。

ローテーションのパターン２

- “右－左”パターン （“左－右”パターンも同様 ）



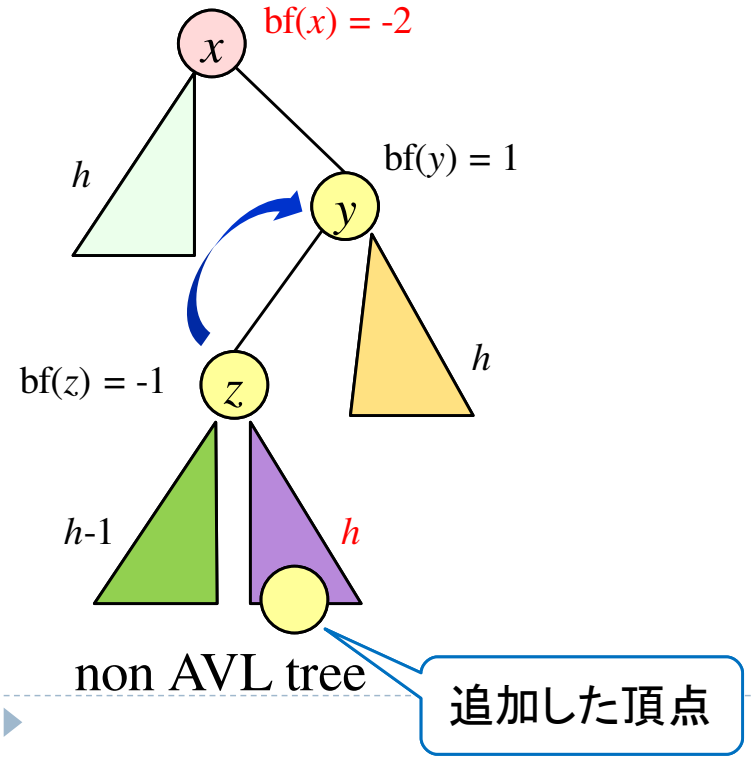
AVL tree

では次に、右ー左パターンを見ていきましょう。ローテーションが2回必要なパターンです。

(次のスライドに続く)

ローテーションのパターン２

- “右－左”パターン （“左－右”パターンも同様 ）

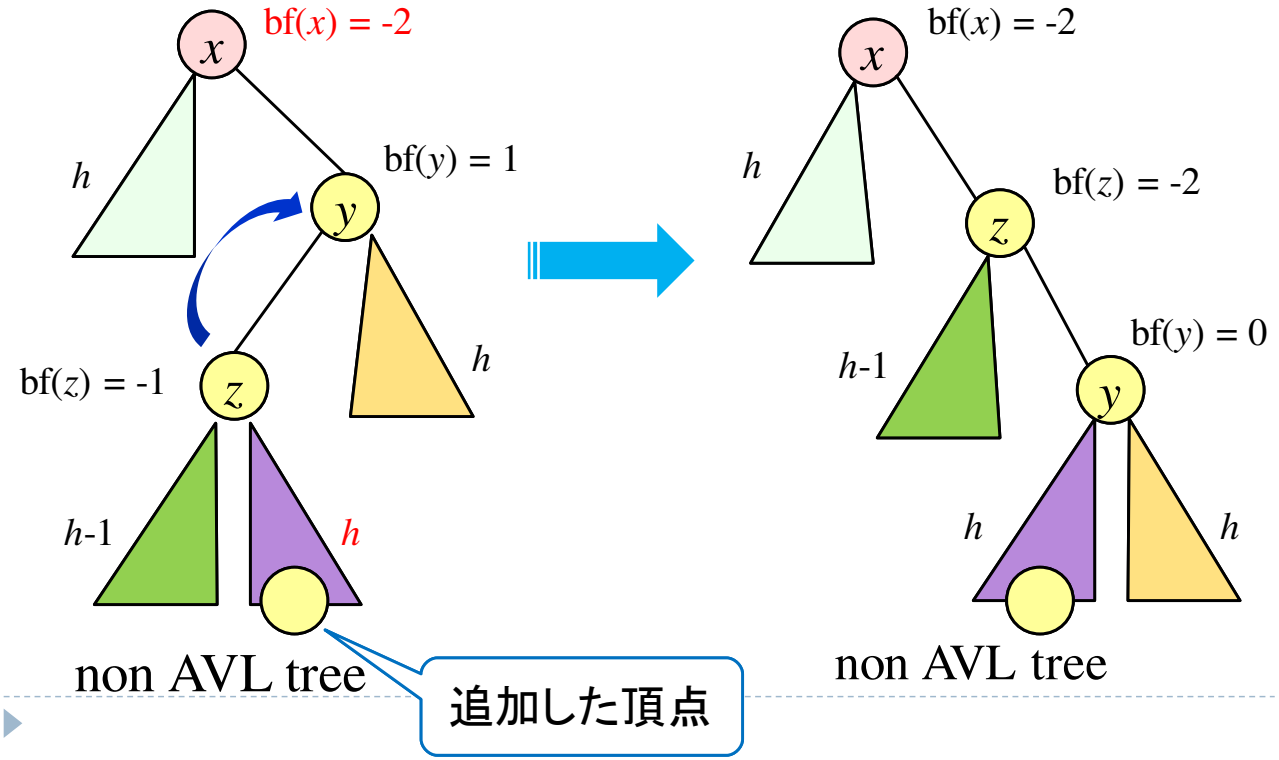


ここで、z の下に頂点が新たに追加された場合を考えます。

z の右の子の高さが h-1 から h になったことで、x のバランスが崩れました。そこで、まず z と y をローテーションします（1回目のローテーション）。

ローテーションのパターン２

- “右－左”パターン （“左－右”パターンも同様 ）

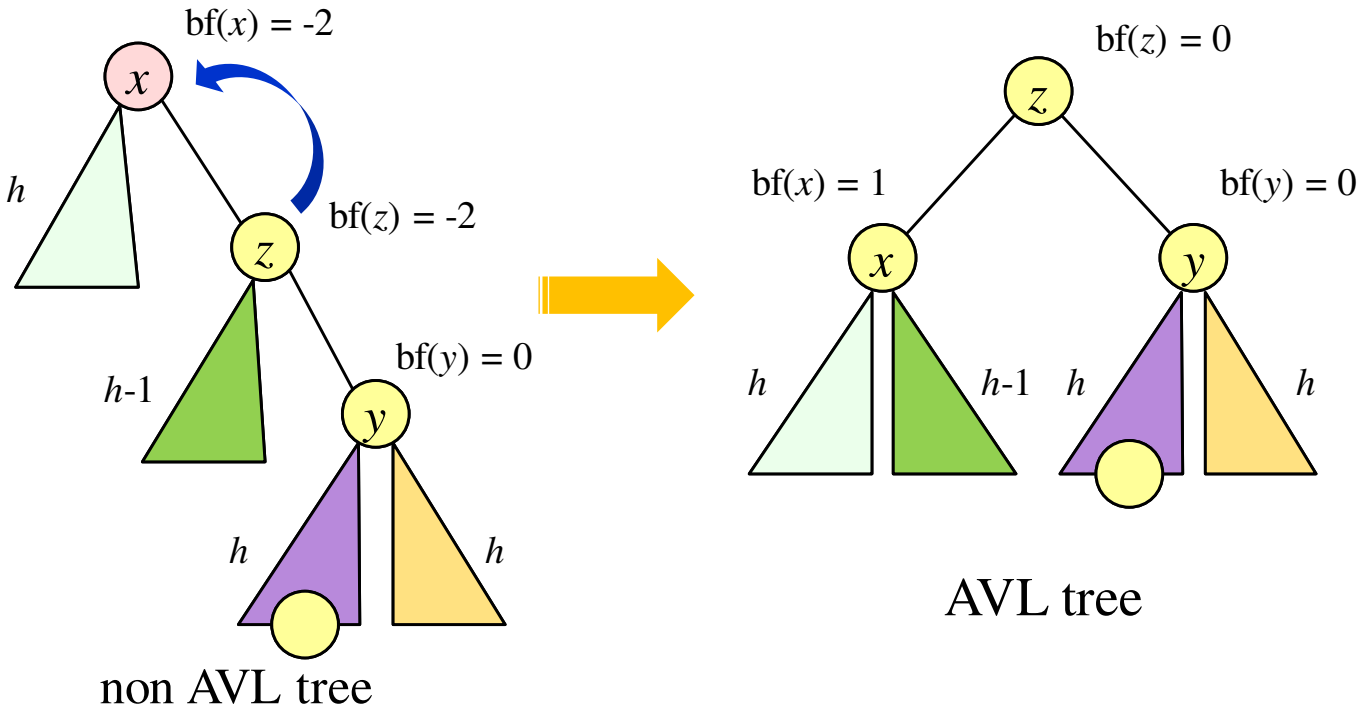


結果の木は右図のようになりました。

まだバランスは崩れたままですが、先ほどの右ー右と同じような形をしていることに注目します。

ローテーションのパターン２（つづき）

- “右－左”パターン （“左－右”パターンも同様 ）



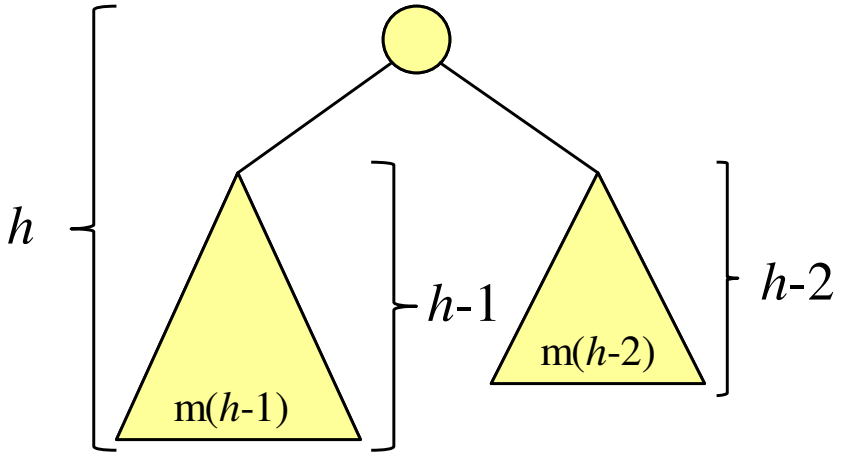
そこで、z と x をローテーションします（2回目のローテーション）

得られた木は右図のようになります。x, y, z すべての頂点がバランスされました。

このように、いずれの場合においても、高々2回の頂点のローテーションで、すべての頂点の bf の値を {-1, 0, 1} の範囲に収めることができることがわかりました。

AVL tree の高さ [1/3]

- m(h) を高さ h の AVL tree の最小の節点数とする.
- このとき, m(1) = 1, m(2) = 2,
  任意の $h > 2$ に対して m(h) = m(h – 1) + m(h – 2) + 1.



前ページまでの解説で、すべての頂点の bf の値を {−1, 0, 1} の範囲に収めることができることがわかりました。

これによって、木の高さが低くなりそうなことは直感的にわかるかと思います。では、果たして数学的にはどの程度の高さになるのでしょうか？ 以降、AVL tree の高さの上界について解説していきます。

m(h) をこのように定義します。明らかに m(1) = 1、m(2) = 2 が成り立ちます。

m という関数はなるべく頂点数を少なくしたいので、任意の h > 2 について、m(h) = m(h − 1) + m(h − 2) + 1 となります。ここで、+1 は図のように2つの部分木を繋げる根の分です。

AVL tree の高さ [2/3]

フィボナッチ数列

- F_h を h 番目のフィボナッチ数とする.
  F_h は以下のように再帰的に定義される.
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_{h+2} = F_{h+1} + F_h \quad (h \geq 0).$
- フィボナッチ数列
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, …

AVL tree の高さを、フィボナッチ数列を用いて解析します。

フィボナッチ数の定義はスライドの通りです。小さい順に F_0 から並べてみると、このような数列になります。

AVL tree の高さ [3/3]

- F_h を h 番目のフィボナッチ数とすると,

$$\mathrm{m}(1) = 1 = F_3 - 1,\ \mathrm{m}(2) = 2 = F_4 - 1.$$

任意の $h > 2$ について

$$\begin{aligned}\mathrm{m}(h) &= \mathrm{m}(h-1) + \mathrm{m}(h-2) + 1 \\ &= (F_{h+1} - 1) + (F_h - 1) + 1 = F_{h+2} - 1.\end{aligned}$$

- $F_h = (((1+\sqrt{5})/2)^h - ((1-\sqrt{5})/2)^h))/\sqrt{5}$ より,
頂点数 n の高さ h のAVL tree について

$$n \geq \mathrm{m}(h) = F_{h+2} - 1 = (((1+\sqrt{5})/2)^{h+2} - ((1-\sqrt{5})/2)^{h+2}))/\sqrt{5} - 1$$

➔ $h < 1.44 \log_2(n+1)$

フィボナッチ数列の定義が、先ほどの m(h) とよく似ていることが分かると思います。

実際、F_h を h 番目のフィボナッチ数とすると、スライドのような等式が成り立ちます。ここで、m と F で添え字 h の値が 2 ズレていることに注意しましょう。

ここで、フィボナッチ数の性質を使います。F_h はこのような式で書けることが知られています。ちなみに、この (1+√5)/2 は黄金比です。

頂点数 n で高さ h の任意の AVL tree を考えます。m(h) の定義から、n ≧ m(h) が成り立ちます。ここで m(h) = F_{h+2} なので、先ほどの式を使って両辺の対数をとると、h < 1.44 log_2(n+1) が得られます。

AVL tree の高さ [3/3]

定理 1

頂点数 n の AVL tree の高さは $1.44 \log_2(n+1)$ で抑えられる.

- 完全平衡2分木の 1.44 倍程度の高さしかない！

よって、この定理が得られました。

AVL tree のまとめ

定理 2

AVL tree を用いて，member, min/max, predecessor/successor, insert/delete を $O(\log n)$ 時間で計算できる.

【証明】

- member, min/max, predecessor/successor
  ➔ 定理 1 より明らかに $O(h) = O(\log n)$ 時間
- insert ➔ $O(\log n)$ 時間 + $O(1)$ 回のローテーション = $O(\log n)$
- delete ➔ $O(\log n)$ 時間 + $O(\log n)$ 回のローテーション = $O(\log n)$

さらに、定理 1 からこの定理 2 が得られます。

member, min, max, predecessor, successor を2分探索木の高さに比例した時間で行うことができることは、前回の講義で学びました。
したがって、AVL tree では、これらのクエリはすべて O(log n) 時間で行うことができます。

insert については、本日の講義でやったとおり、まず O(log n) 時間を使って member を行ったあと、高々2回の頂点のローテーションを行います。各ローテーションは O(1) 時間で行えるので、合計で O(log n) 時間となります。

AVL tree のまとめ

定理 2

AVL tree を用いて，member, min/max, predecessor/successor, insert/delete を $O(\log n)$ 時間で計算できる.

【証明】

- member, min/max, predecessor/successor
  ➔ 定理 1 より明らかに $O(h) = O(\log n)$ 時間
- insert ➔ $O(\log n)$ 時間 + $O(1)$ 回のローテーション = $O(\log n)$
- delete ➔ $O(\log n)$ 時間 + $O(\log n)$ 回のローテーション = $O(\log n)$

delete については本日は扱いませんでしたが、O(log n) 回のローテーションで実現することができます。興味のある人は自分で考えてみる、あるいは調べてみるとよいと思います。

以上、AVL tree について学びました。
本日の講義は以上です。