平衡木

この回の要点

- 二分探索木の問題点
  - 木のバランスが問題
- 平衡探索木（balanced search tree）
- 平衡木（balanced tree）
  - なぜ必要なのか？
  - どういうものか？
    - 木の種類によってバランスの取り方が工夫されている
    - AVL木
    - B木

二分探索木の問題点

- 平均では$O(\log n)$の計算量
- 最悪の場合、探索が$O(n)$
  - 挿入する順番が昇順（あるいは降順）の場合
    - これは、実際のケースとしてはよくある
- 最悪でなくても、木のバランスが悪いと$O(\log n)$の計算量にならない
- 木の構造が完全二分木の場合が最良
  - 探索が根から葉に向かってたどるから、木の高さが低いほうが効率がよい
  - 完全二分木は、nが2のべき乗でないと不可能
  - 完全二分木でなくても、十分に枝分かれしていれば、性能は$O(\log n)$程度に収まるだろう
  - せめて、木の高さの差が１以内ならば・・・

平衡木とは

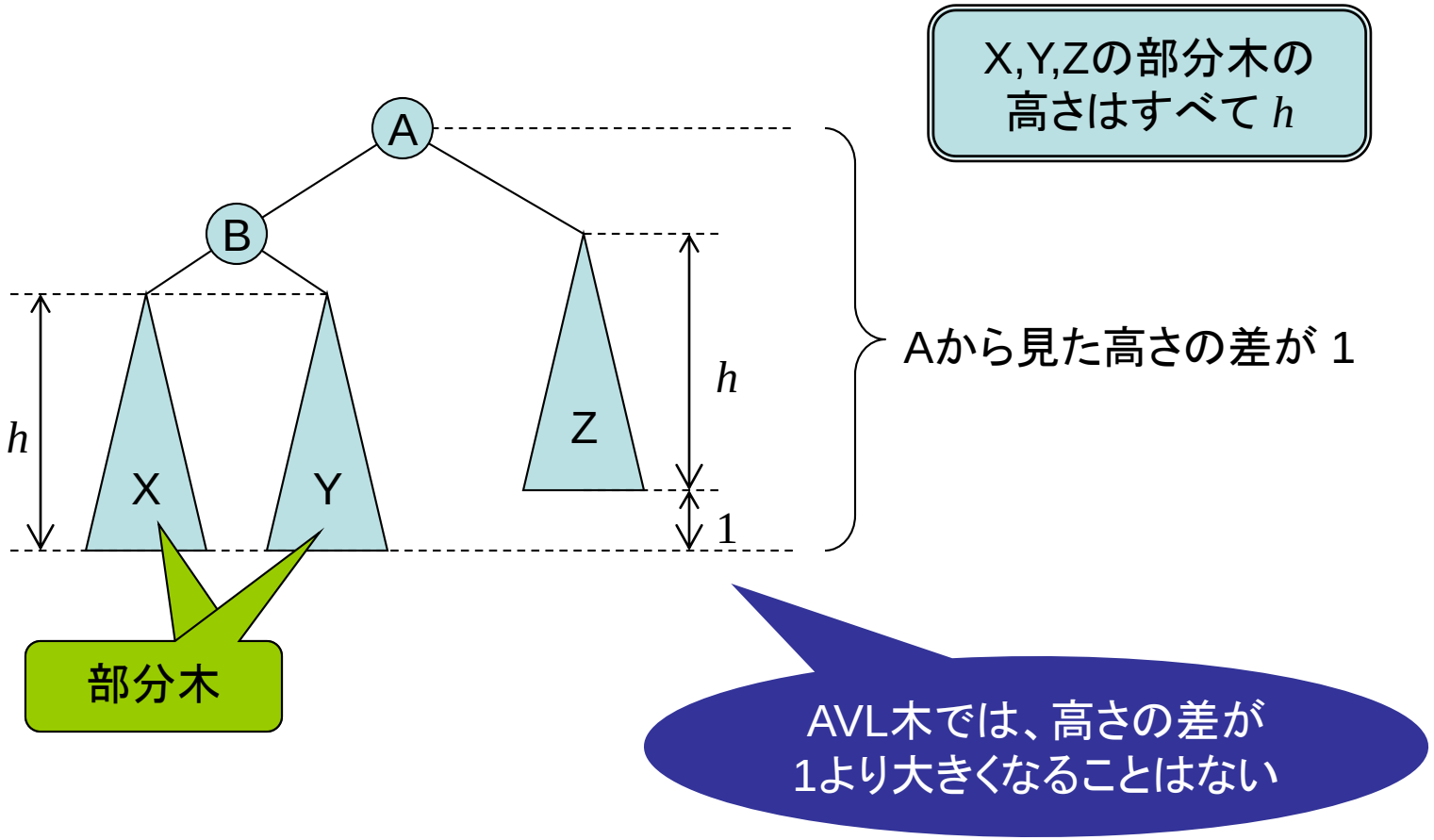
- 平衡木（balanced tree）
- 木の高さが常に$O(\log n)$程度の木
  - 木の形が変わる（＝データの挿入・削除が行われる）たびに、木の形を見直して変形する
  - 木の高さが$O(\log n)$であれば、探索に要する計算量も$O(\log n)$となる。
- どのようにして木の形を見直すか？
  - その方法により、さまざまな木が提案されている
    - AVL木、B木、B*木、2-3木、二色木
  - 見直しに要する手間が$O(\log n)$以内に収まらなければ、意味がない
    - 挿入、削除を行うたびに見直しも行うので、もし、見直しの手間が$O(n)$ならば、全体の計算量も$O(n)$となるから

AVL木（AVL-tree）

- Adel'son-Vel'skiiとLandisが考案（1962年）
- 初めて示された平衡木だが、実用上はB木の方が優秀（オーダーは同じ、定数係数が違う）
- AVL木の定義：
  - すべてのノードにおいて、左部分木と右部分木の高さの差が1以内に収まるような二分木
- 基本的な考え方
  - 二分探索木と同様に、葉として挿入
  - 左右の部分木の高さの差が2以上であれば、ノードを回転させて、高さの差が1以内になるようにする

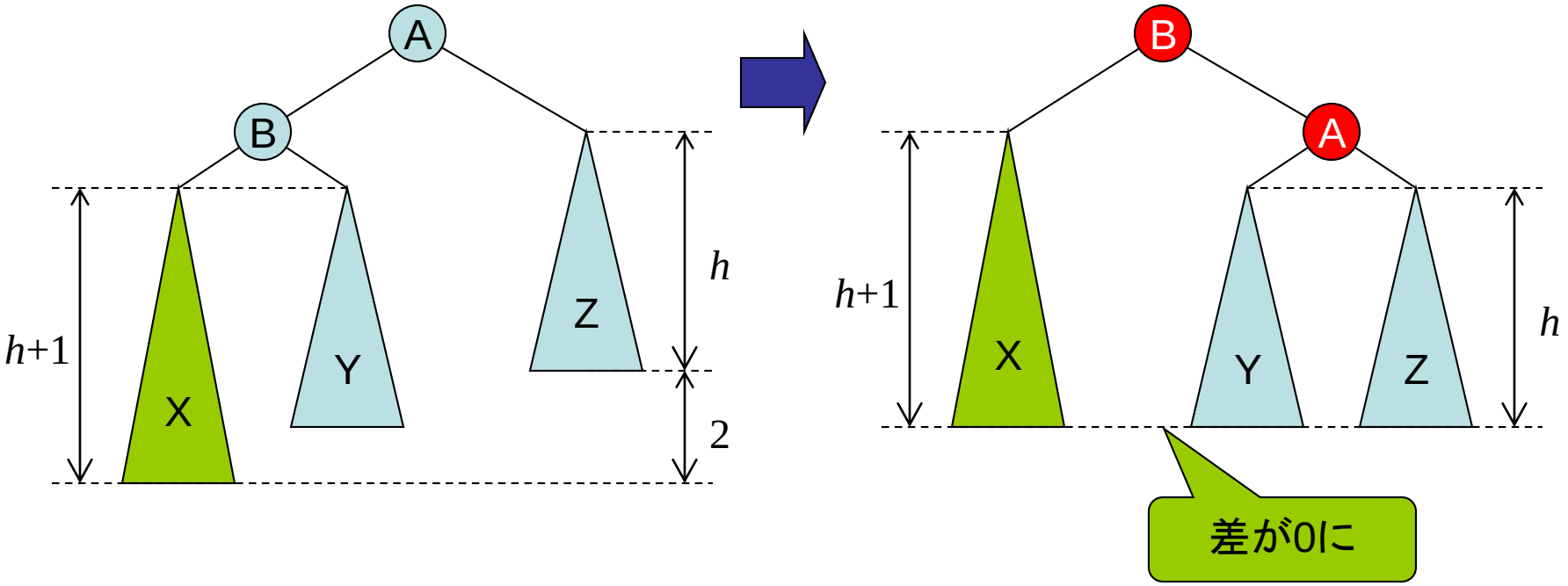
AVL木への挿入1

- 挿入前のAVL木の構造



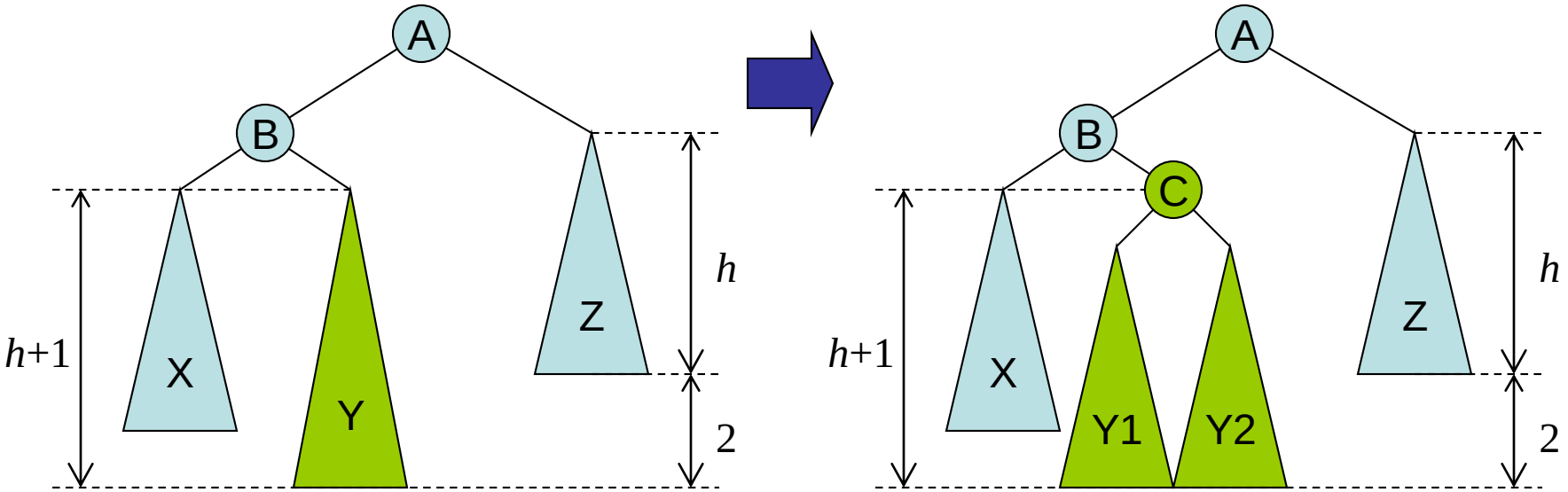
AVL木への挿入2

- 新しい要素がXに挿入され、Xの高さが$h+1$に
- AとBとを入れ替えて、BをAの親にする
  - 一重回転（single rotation）
- 二分探索木の性質は保たれる



AVL木への挿入3

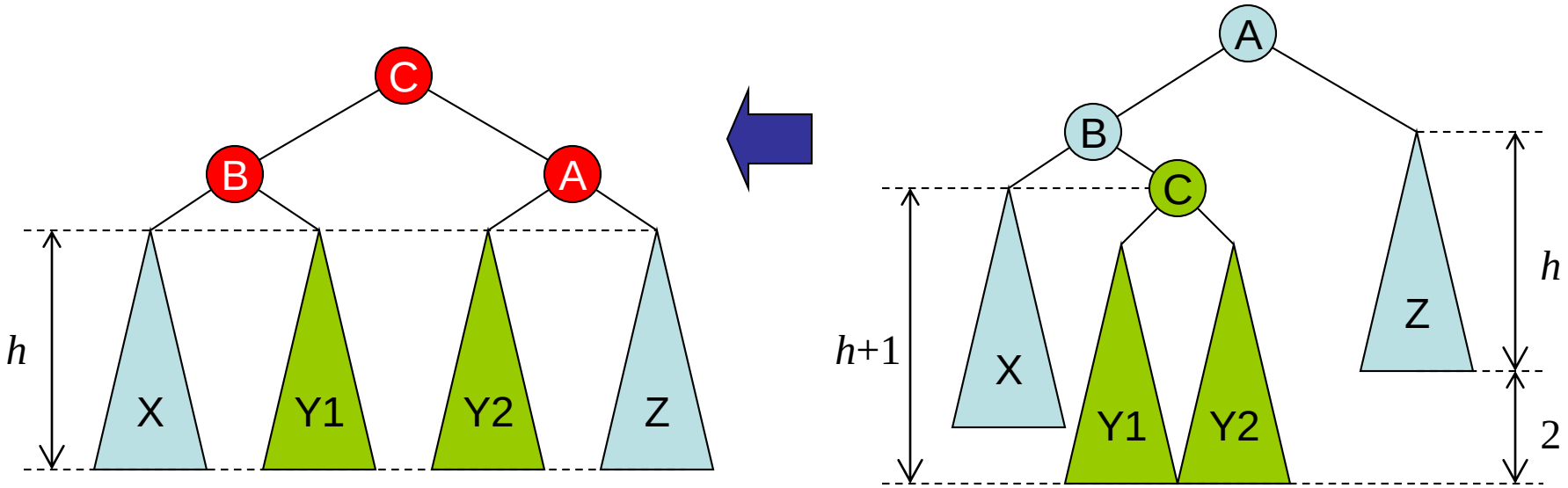
- 新しい要素がYに挿入され、Yの高さが$h+1$に
- 部分木Yを、根ノードCと部分木Y1,Y2に分ける



AVL木への挿入4

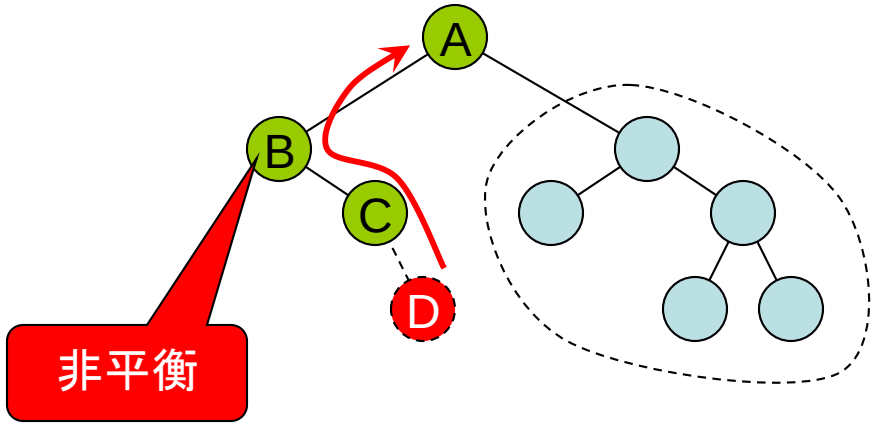
- ノードA,B,Cを入れ替える
  - Cの右の子ノードがA
  - Cの左の子ノードがB
  - 二重回転（double rotation）
- 二分探索木の性質は保たれる

左右対称な右部分木の
場合も、同様に考える



AVL木への挿入5

- 回転による木のバランス調整
  - 葉から根に向かって順次行う
  - 必要ならば、回転を行う
  - 調整の対象とならない部分木の構造は変化しない
- 必要な計算量
  - 一重回転、二重回転ともに$O(1)$
    - 必要な作業は対象ノードのリンクのつなぎかえのみなので
  - 葉から根にたどることに$O(\log n)$
  - トータルでは$O(\log n)$

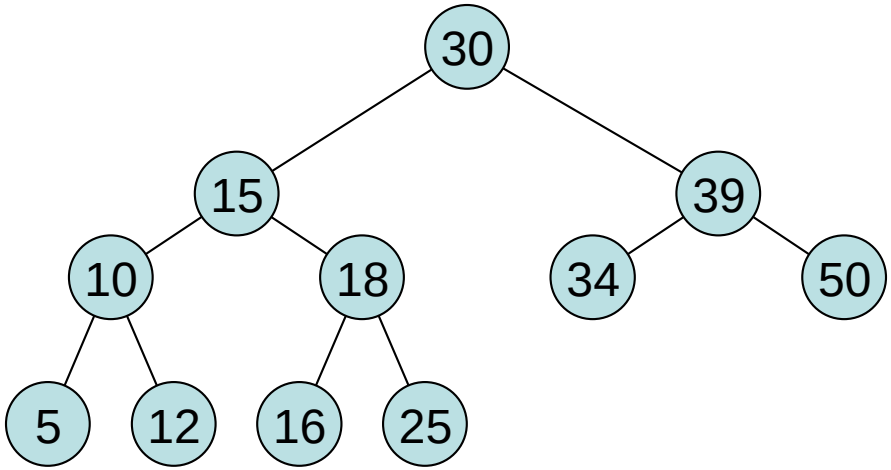


AVL木からの削除

- 通常の削除を行った後、回転してバランス調整
- 削除の計算量
  - ノードの削除$O(\log n)$
  - 回転によるバランス調整$O(\log n)$
  - トータルでは$O(\log n)$

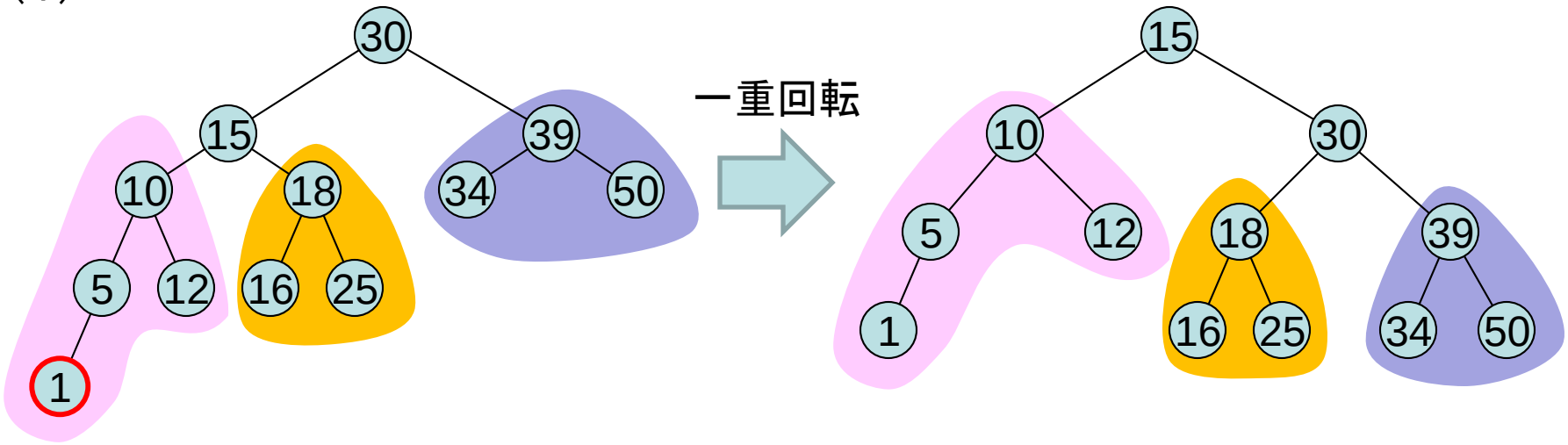
練習問題

- 下記のAVL木に以下の操作を行った結果を示せ。
  （1） 要素1を挿入する
  （2） 要素28を挿入する

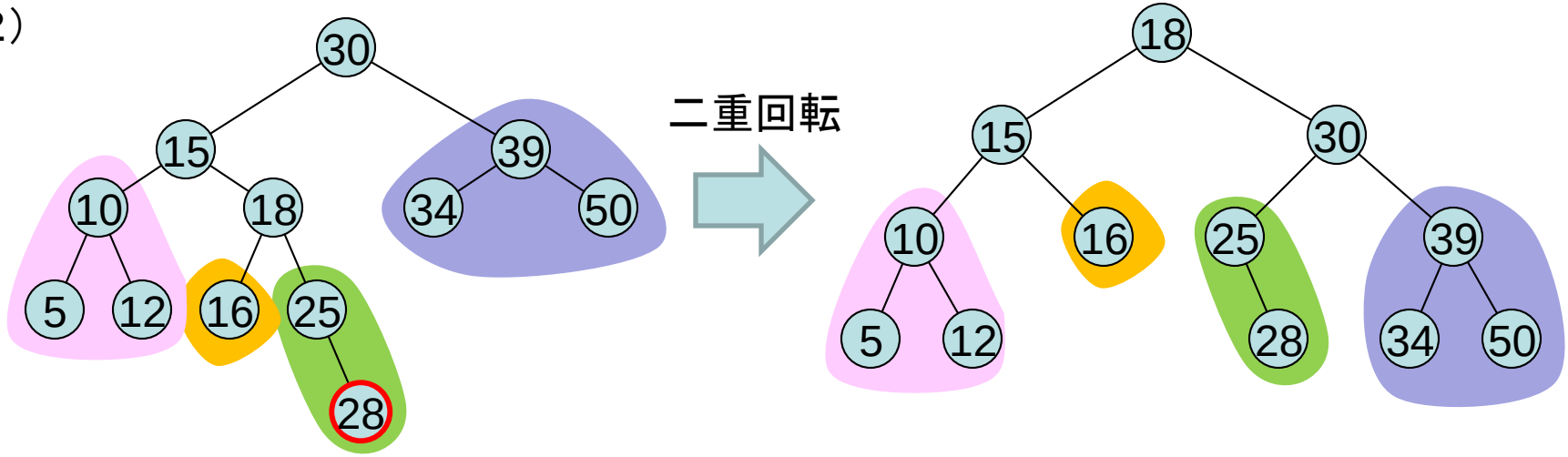


解答

（1）


30
15
39
10
18
34
50
5
12
16
25
1
一重回転
15
10
30
5
12
18
39
1
16
25
34
50

（2）


30
15
39
10
18
34
50
5
12
16
25
28
二重回転
18
15
30
10
16
25
39
5
12
28
34
50

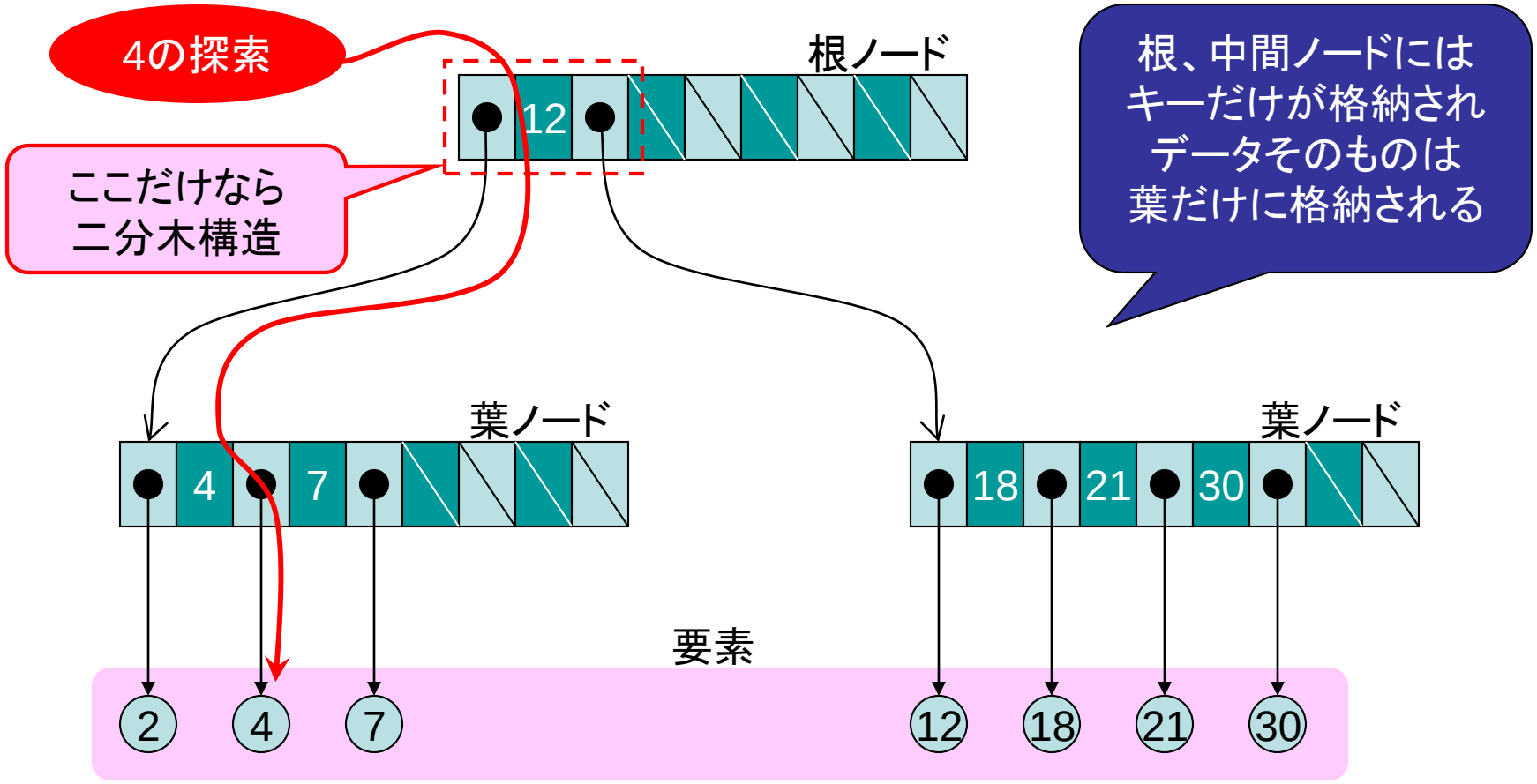
AVL木の実装

- 基本的にはBinSearchTree
- BinTreeに以下の機能を追加する。
  - insertAVL(BinTree *tree,void *key,void(*comp)(void*,void))
    - まずinsert()を呼び出し、要素を挿入する。
    - ノードを根に向かってたどりながらbalanceする。
  - removeAVL(BinTree *tree,void *key,void(*comp)(void*,void*))
    - まずremove()を呼び出し、要素を削除する。
    - ノードを根に向かってたどりながらbalanceする。
  - balance(BinTree *t,BinTreeNode *n)
    - 与えられたノードnの下のバランスを調整する。
    - 必要に応じて、ノードの回転を行う。
  - rotateRightS(BinTree *t,BinTreeNode *n) – 右一重回転
  - rotateRightD(BinTree *t,BinTreeNode *n) – 右二重回転
  - rotateLeftS(BinTree *t,BinTreeNode *n) – 左一重回転
  - rotateLeftD(BinTree *t,BinTreeNode *n) – 左二重回転

B木（B-tree）

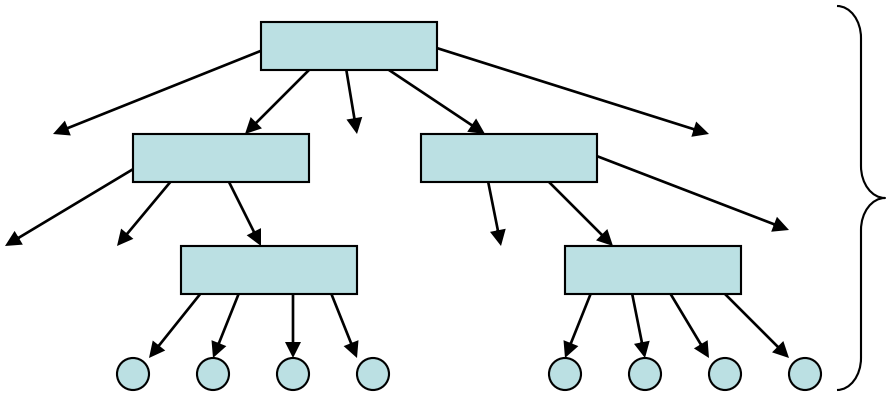
- BayerとMcCreightが考案（1972年）
- 実用上の価値が高いデータ構造
- B木の構造
  - m分木構造（$m \geqq 2$）（多分木構造）
  - m分探索木のB木を、m階のB木と呼ぶ
- m階のB木の条件
  - 根は、葉であるか、あるいは2～m個の子を持つ
  - 根、葉以外のノードは、ceil(m/2)～m個の子を持つ
  - 根からすべての葉までの経路の長さが等しい
- B木の特徴
  - データを持つノードは葉のみ
  - 内部ノードはキーのみを持つ
  - 常にバランスが取れている状態を保つ

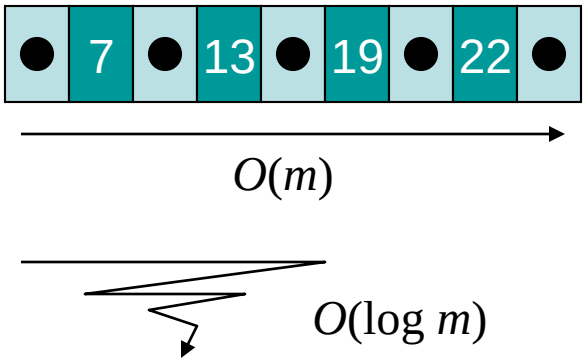
B木の探索（5階のB木）


4の探索
ここだけなら
二分木構造
根ノード
12
根、中間ノードには
キーだけが格納され
データそのものは
葉だけに格納される
葉ノード
4
7
葉ノード
18
21
30
要素
2
4
7
12
18
21
30

m階のB木の探索計算量

- m階B木における探索
  - 根から葉まで$O(\log n)$個のノードをたどる
  - 各ノードには、m-1個の比較キーがある
    - 線形探索の場合、$O(m)$
    - 二分探索の場合、$O(\log m)$
    - いずれにせよ、$m<<n$であるから、nについては$O(1)$
  - これより、B木の探索は$O(\log n)$である

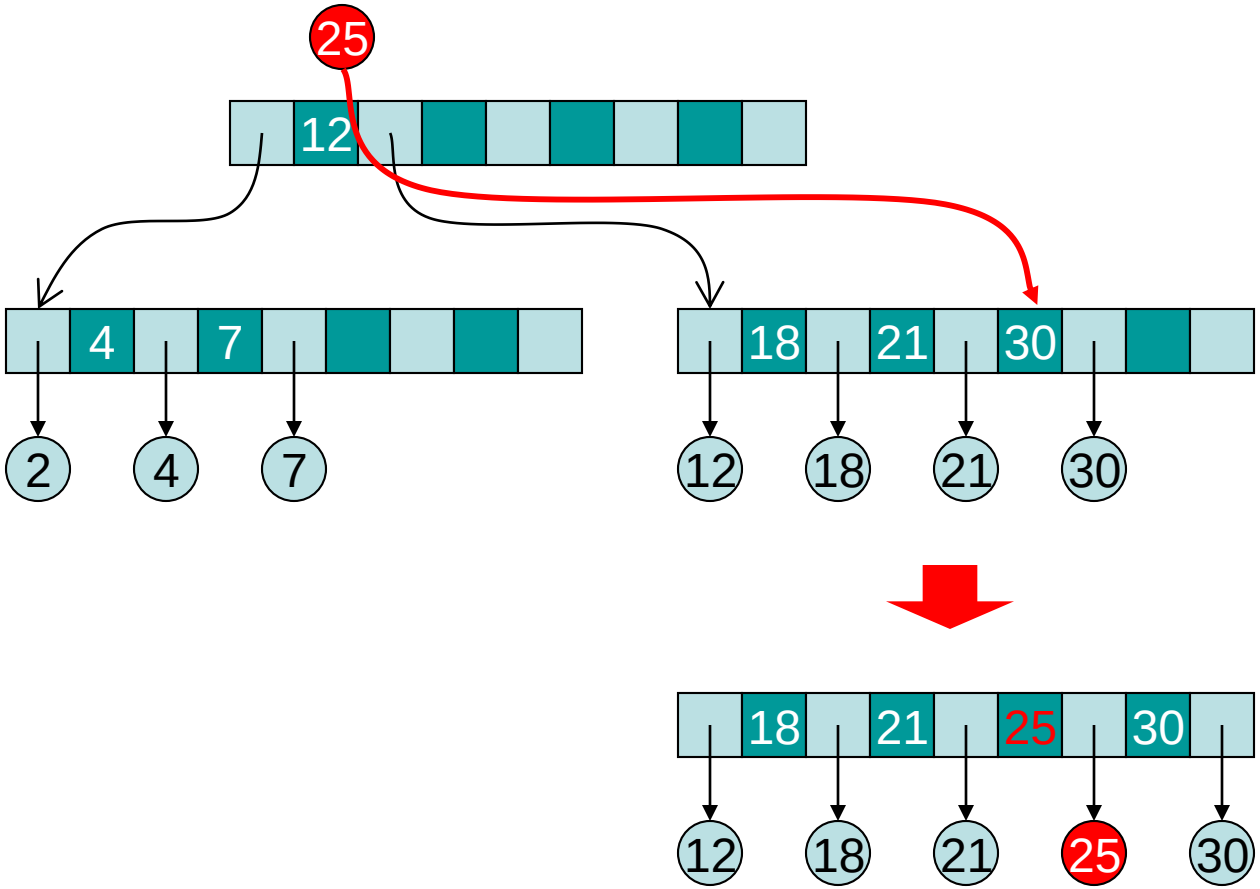




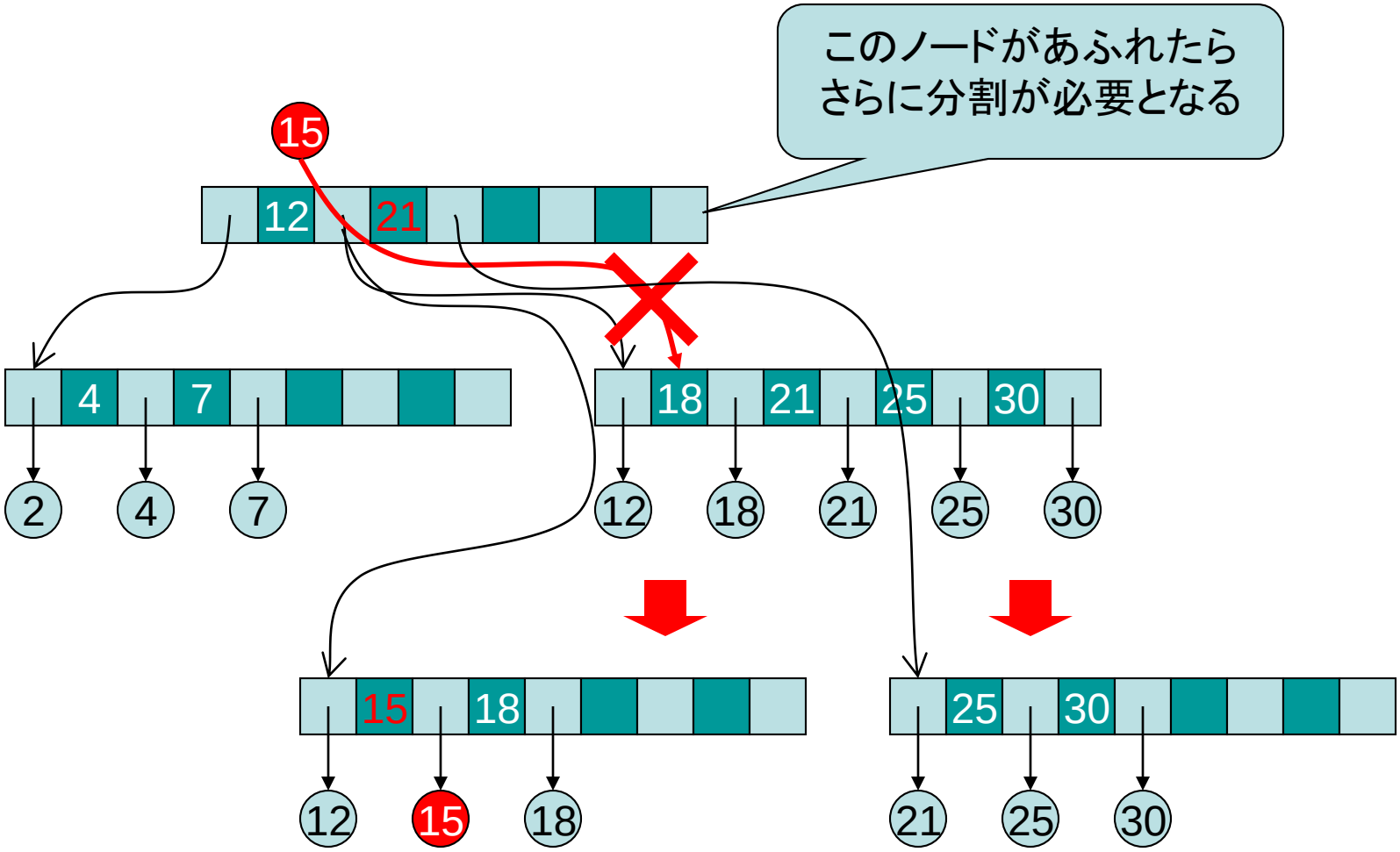
B木への挿入1

- 挿入手順
  - 挿入したいデータを探す
  - 挿入すべき葉ノードの1つ手前のノードをA
    - ノードAに新しい葉を追加する余裕があれば、追加する
    - ノードAがいっぱいの場合は、Aを2つに分割する
      - 新しいノードBを生成し、葉を半分ずつにする
  - ノードAの親ノードにとっては、子Bが1つ増えた
    - 親ノードに余裕があれば、Bを追加
    - 余裕がなければ、親ノードも分割
  - この処理を根ノードまで繰り返す
    - 根ノードにも余裕がなければ、分割して新しい根ノードを生成

B木への挿入2



B木への挿入3


このノードがあふれたら
さらに分割が必要となる
15
12
21
4
7
18
21
25
30
2
4
7
12
18
21
25
30
15
18
25
30
12
15
18
21
25
30

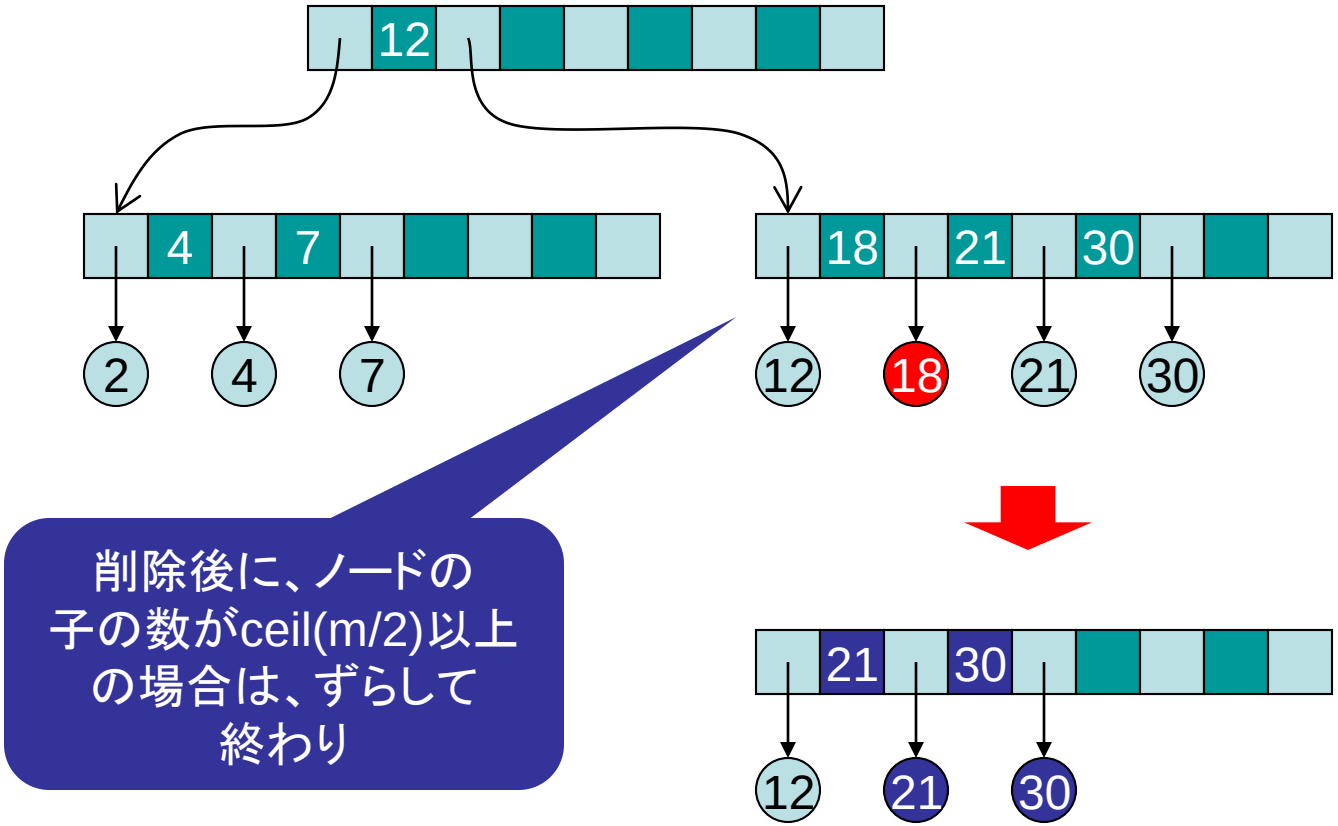
B木への挿入4

- 挿入操作は親ノードへと伝播する
  - 分割されないノードがあれば終了
  - 分割が根ノードまで伝播した場合、
    - 根ノードがいっぱいならば、根ノードを分割して、新しい根ノードを作成する
      - この場合のみ、B木の高さが増える
    - 根ノードまで伝播した場合が、挿入の最悪のケース
      - この場合のノードの分割回数は$O(\log n)$である
- ノードの分割操作
  - $m+1$個の要素を2つのノードに振り分ける
  - 計算量は$O(m)$であるが、$m<<n$であるから$O(1)$
- したがって、B木への挿入の計算量は$O(\log n)$

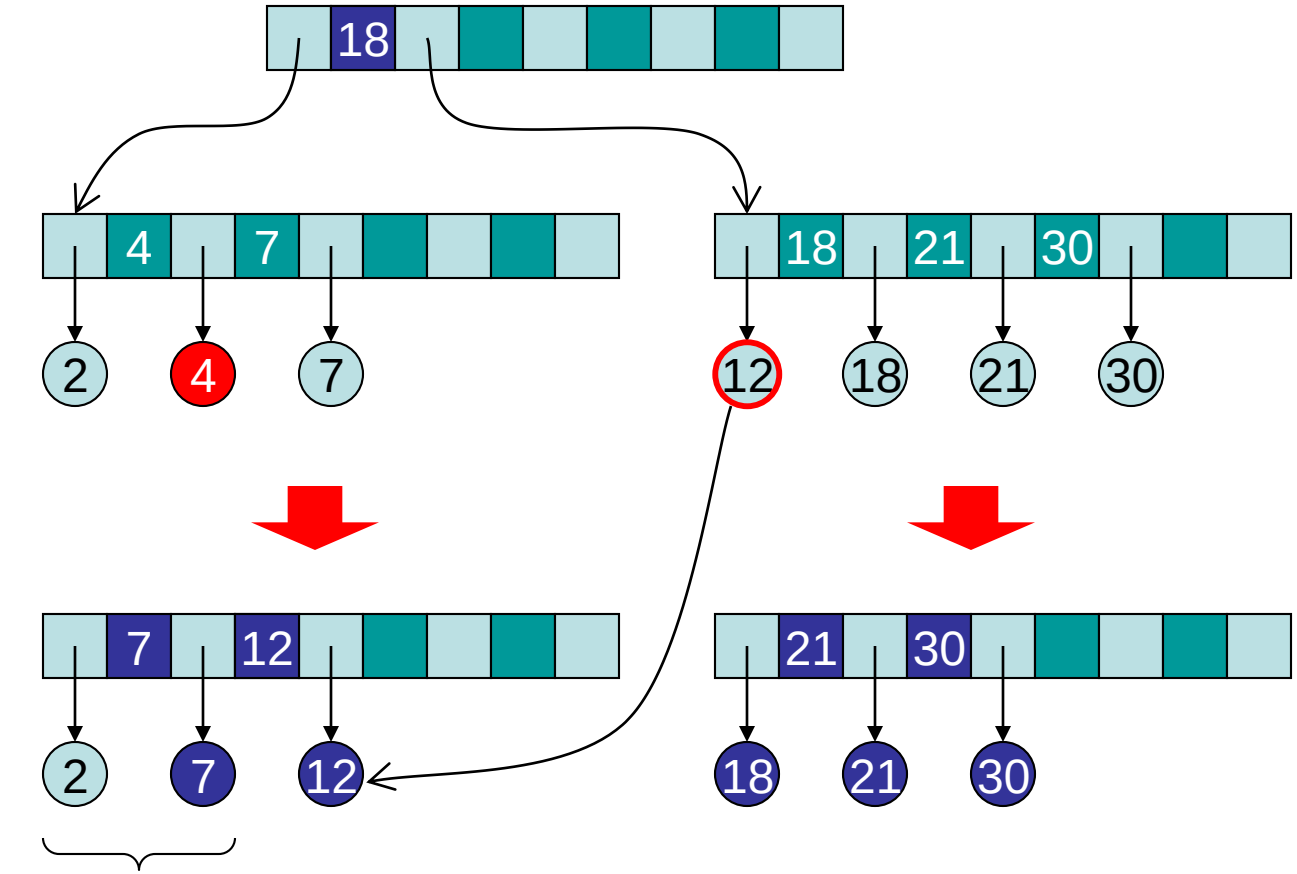
B木からの削除1

- 削除手順
  - 削除する葉Lを含むノードAを探索
  - Lを削除したとき、
    - Aにceil(*m*/2)個以上の葉が残っていれば終了
    - Aがceil(*m*/2)-1個のときは、
      - 隣のノードの葉と合わせて半分ずつにする
      - 隣のノードがちょうどceil(*m*/2)個の葉しか持たなかった場合は、2つを併合する
        - » 併合した場合、親ノードから見れば子が1つ減ったことになるので、削除処理が親に伝播する

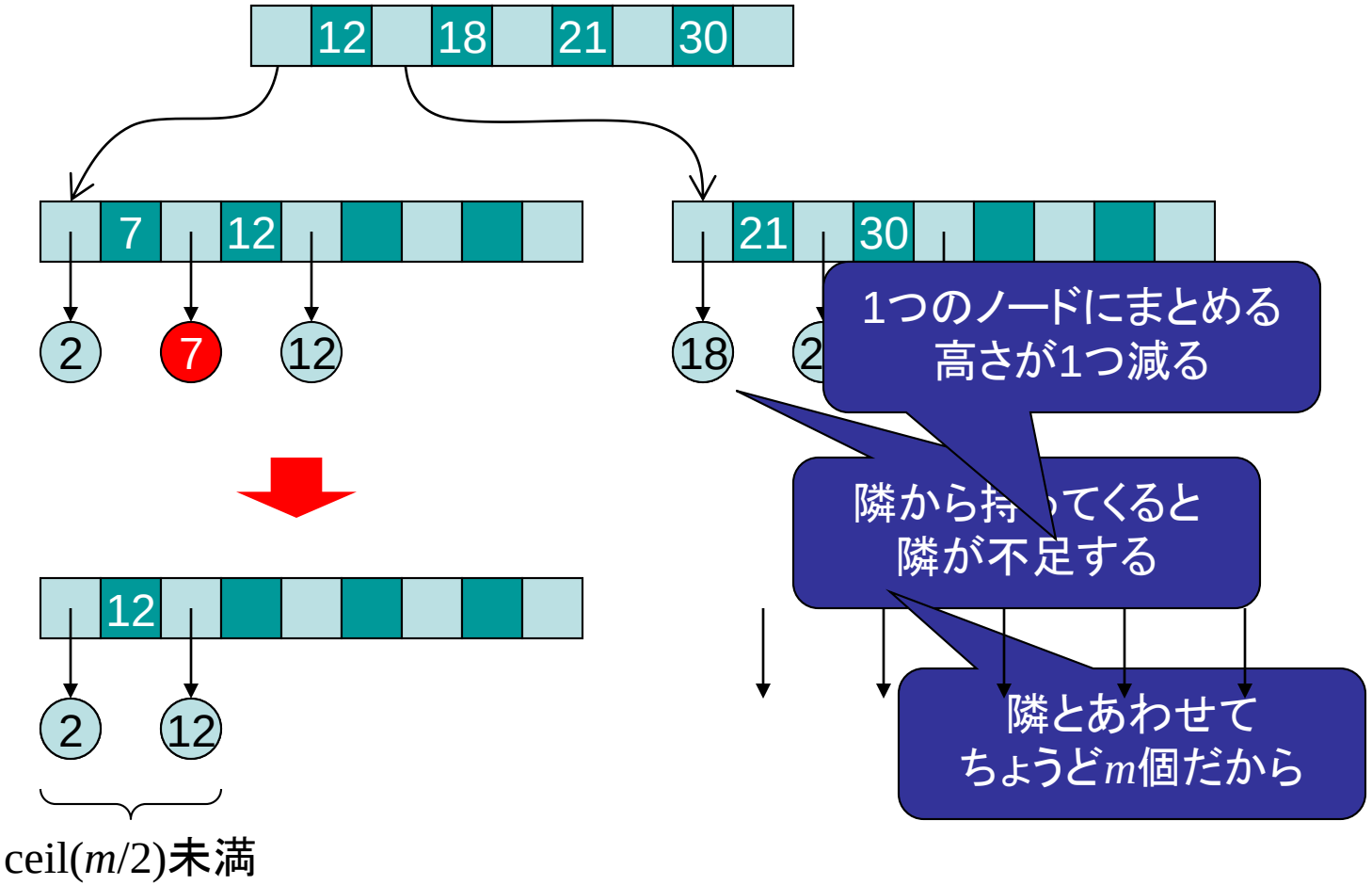
B木からの削除2


12
4
7
18
21
30
2
4
7
12
18
21
30
削除後に、ノードの
子の数がceil(m/2)以上
の場合は、ずらして
終わり
21
30
12
21
30

B木からの削除3



B木からの削除4


12
18
21
30
7
12
21
30
2
7
12
18
2
1つのノードにまとめる
高さが1つ減る
隣から持ってくると
隣が不足する
12
2
12
隣とあわせて
ちょうどm個だから
ceil(m/2)未満

B木からの削除5

- 削除操作も親ノードに伝播する
  - 併合しないノードがあれば終了
  - 併合が根ノードまで伝播した場合、
    - 根ノードの子ノードを併合した場合、根ノードが1つ減る
      - この場合のみ、B木の高さが低くなる
    - 根ノードまで伝播した場合が、削除の最悪のケース
      - この場合のノードの併合回数は$O(\log n)$である
- ノードの併合操作
  - m個の要素を1つのノードに格納する
  - 計算量は$O(m)$であるが、$m<<n$であるから$O(1)$
- したがって、B木への削除の計算量は$O(\log n)$

B木の性質

- B木の応用
  - ディスク装置上でのセクタ管理
    - セクタ長1024、キー長10バイト、オフセット4バイトの場合、1024/(10+4)=73で、73階のB木を構成可能
    - 3段で73^3=38万件、4段で73^4=2839万件の検索が可能
  - データベース検索（Berkeley DB）
- B木の欠点
  - 内部ノードの子の数が最小でceil(m/2)
    - 最悪の場合、半分のノード用メモリが無駄
  - 改良→B*木

その他の平衡木

- 2-3木
  - 各ノードの子の数を3個まで許す
  - m=3の場合のB木
- B*木
  - 内部ノードの持つ子の数を、ceil(2m/3)以上、m以下に変更したB木
  - B木では、最悪の場合、内部ノードの半分しか使われないため、半分は未使用となり無駄
  - これを、2/3に変えたもの
- 赤黒木（red-black-tree）
  - AVL木の改良
  - 回転操作が定数回で済む
  - 木の各辺に赤か黒の色を付ける
  - 赤ノードが2つ連続することはない（ようにする）
  - 根から葉までの最長経路長は、最短経路長の2倍を超えない
    - （あるていどの）平衡木の性質を持つ

平衡木の実験

- 平衡木の操作状況を視覚的に確認する
- HPからダウンロードし、ダブルクリックで実行
  - ExpA.jar：通常の二分探索木
  - ExpB.jar：AVL木
  - ExpC.jar：B木
- 機能
  - 指定した要素を木に挿入する（数値で）
  - 指定した要素を木から削除する（選択で、数値で）
  - ランダムに10個の要素を木に挿入する
  - 全体を拡大縮小する（ホイール、Shift+ホイール）
  - 全体を移動する（ドラッグ）

課題151221

- ExpA.jarを用いて、
  - 10個の要素を手動で追加し、バランスの良い木を示せ。
  - 10個の要素を手動で追加し、バランスの悪い木を示せ。
- ExpB.jarを用いて、
  - 一重回転が起こった前後の図を示せ。
  - 二重回転が起こった前後の図を示せ。
- ExpC.jarを用いて、
  - 木の高さが3になる前後の図を示せ。

- ワードで作成し、文章と図で説明すること。
- 本文の先頭に学籍番号と氏名を記載すること。
- ワードのファイル名は、”scXXXXXX-al151221.docx”とすること。
- メールに添付して渕田まで送信すること。
- メールのタイトルは、“アルゴリズム課題151221”とし、メールの本文中にも学籍番号と氏名を記載すること。
- 提出締切：2016年1月3日（日） 24:00

平衡木

終了