データ構造とアルゴリズム
（第7回）

モバイルコンピューティング研究室
柴田史久

1

1

本日の講義内容

- 探索（2）
  - 二分探索木
  - 二分探索木の操作
    - 探索
    - 挿入
    - 削除
  - 二分探索木の性質
  - 平衡木とは
  - AVL木
  - B木（割愛）

2

2

教科書 第9章（pp.218～242）

二分探索木

3

3

木構造による探索アルゴリズム

- 根から葉に向かってたどる経路が探索のプロセス
- 木構造を利用した探索アルゴリズム
  - 二分探索木
  - AVL木
  - 2-3木
  - B木
  - B*木
  - トライ（trie）

4

4

二分探索木

- 二分木をもとにして探索を行うのが二分探索木

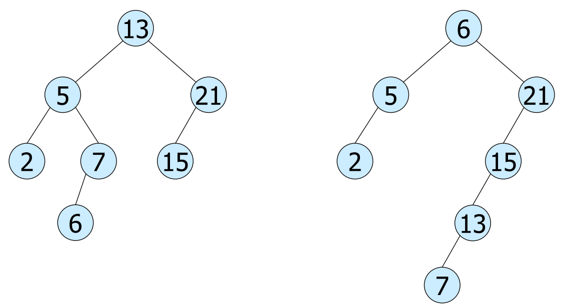
- 二分探索木
  - 二分木の各ノード（節）に要素（データ）を持たせたもの
  - 任意のノード x について
    - 左部分木に含まれる要素はノード x よりも小さい
    - 右部分木に含まれる要素はノード x よりも大きい

5

5

二分探索木の例

- 7つの要素：2, 5, 6, 7, 13, 15 ,21



6

6

二分探索木の探索

探索過程を記入してみよう

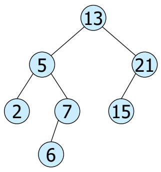
- 注目ノードの内容とキー（k）を比較し，
  - キーがノードの内容より小さければ左の子を訪問
  - キーがノードの内容より大きければ右の子を訪問
  - キーとノードの内容が一致すれば見つかった
  - 注目ノードが存在しなければ見つからなかった

k = 7 を探索

k = 7 < 13

k = 7 > 5

k = 7 = 7



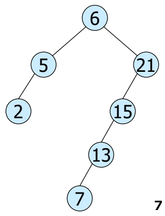
k = 7 > 6

k = 7 < 21

k = 7 < 15

k = 7 < 13

k = 7 = 7



二分探索木の実装(1)（教科書 pp.221～）

- 節を表すNodeクラス
  - データ：data
  - 左右の部分木：left, right
- コンストラクタで初期化
  - データをセットし，左右の部分木はnull

詳細は教科書 p.221 List 9.1

```
class Node {
  Integer data;   // データ
  Node    left;   // 左部分木
  Node    right;  // 右部分木

  // コンストラクタ
  Node(Integer data) {
    left      = null;
    right     = null;
    this.data = data;
  }
}
```


Integerクラスについて

- プリミティブ型 int のラッパークラス
- Comparableインタフェースを実装
  - 値の大小を判定する compareTo メソッドを持つ
- compareToメソッドは自身 this と引数 x を比較
  - public int compareTo(Integer x)
    - this < x　→ 負の整数
    - this == x　→ 0
    - this > x　→ 正の整数
- データとしてはComparableインタフェースを実装した他のクラスも利用可能

二分探索木の実装(2)

- 二分探索木を表すBinarySearchTreeクラス
  - 根を表す Node root
- コンストラクタで初期化
  - 初期状態は空の木

詳細は教科書 p.223 List 9.2

```
class BinarySearchTree {
  private Node root;   // 二分探索木の根

  // コンストラクタ
  BinarySearchTree() {
    root = null;       // 初期状態は空の木
  }

  // 以下の部分にメソッドを定義
}
```


二分探索木の実装(3)

- 探索：searchメソッド
  - 探索すべきキーを受け取る
  - 探索に成功した場合，そのデータを持つノードを返す
  - 探索に失敗した場合，nullを返す

詳細は教科書 p.224 List 9.3

```
class BinarySearchTree {
  public Node search(Integer key) {
    Node p = root;          // まず根に注目
    while (p != null) {     // 注目するノードがある限りループ
      int result = key.compareTo(p.data);   // 注目するノードとキーを比較
      if (result == 0) {    // ノードとキーが等しい場合
        return p;
      } else if (result < 0) { // キーが小さいので
        p = p.left;            // 左部分木へすすむ
      } else {                 // キーが大きいので
        p = p.right;           // 右部分木へすすむ
      }
    }
    return null;    // ここに到達した場合，見つからなかった
  }
}
```


二分探索木への挿入

挿入過程を記入してみよう

- 注目ノードと挿入データのキー（k）を比較し，
  - キーがノードの内容より小さければ左の子を訪問
  - キーがノードの内容より大きければ右の子を訪問
  - キーとノードの内容が一致すれば既に登録済み
  - 注目ノードが存在しなければ挿入場所が見つかった

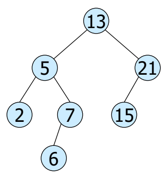
k = 8 を挿入

k = 8 < 13

k = 8 > 5

k = 8 > 7

7の右部分木がない



k = 8 > 6

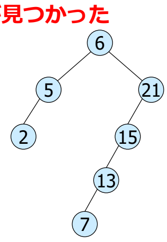
k = 8 < 21

k = 8 < 15

k = 8 < 13

k = 8 > 7

7の右部分木がない



二分探索木の実装(4)

- 挿入：insertメソッド
  - 挿入すべきキーを受け取る
  - 挿入に成功した場合，そのデータを持つノードを返す
  - 挿入に失敗した場合，nullを返す

詳細は教科書 p.227 List 9.4

```
class BinarySearchTree {
  public Node insert(Integer key) {
    Node p      = root;  // 根に注目
    Node parent = null;  // 注目するノードの親. 初期値はnull
    boolean isLeftChild = false;  // parent の左の子かどうか
    // 挿入場所を探す
    while (p != null) {  // 注目するノードがある限りループ
      // 省略. search メソッドと似た感じ
      // 部分木に進む際 parent に現在のノードを記録
      // isLeftChild に左の子かどうかを記録
    }
    // 新しいノードを適切な場所に挿入
    Node newNode = new Node(key);  // 新しいノードを準備
    // 省略. 根になるか左の子になるか右の子になるか
  }
}
```

parentに親を覚えておく
parentの左or右の子として
ノードを挿入するため

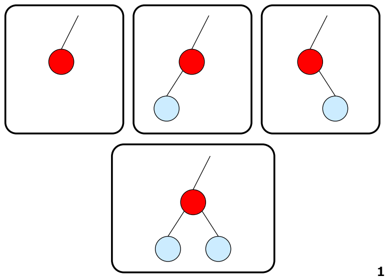
13

13

二分探索木からの削除

コードは割愛，詳細は教科書 pp.234~235 List 9.5
p.239 List 9.6

- 考え方
  - 削除後の木構造が二分探索木の条件を満たすこと
  - 削除対象のノードの状態によって場合分け
- 場合分け
  - 子がない
  - １つの子を持つ
    - 右の子のみ
    - 左の子のみ
  - ２つの子を持つ



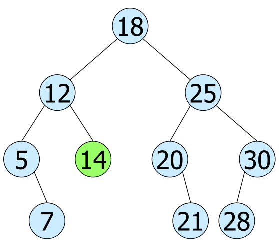
14

14

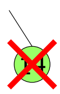
ノードの削除(1) 子ノードがない

削除対象がルートノード　→　そのまま削除

削除対象がルートノード以外



親ノードからのエッジを切り離し
削除対象のノードを削除



15

15

ノードの削除(2) 1つの子ノードを持つ(1)

削除対象がルートノード　→　子ノードを削除後の
新しいルートノードに

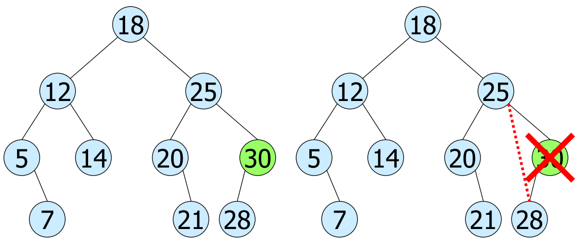
16

16

ノードの削除(3) 1つの子ノードを持つ(2)

削除対象が
ルートノード以外　→　親ノードからのエッジを
削除対象のノードの子ノードに

親ノードの左右どちらの子ノードとするかで
場合分けが必要

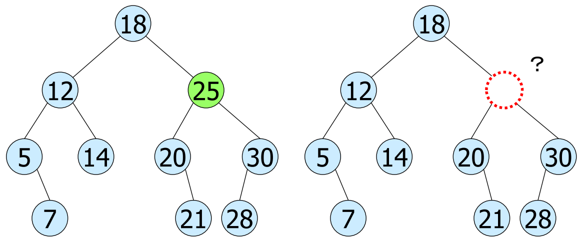


17

17

ノードの削除(4) 2つの子ノードを持つ(1)

- 子ノードが２つあるため，削除後のノードを
子ノードで置き換えることができない
- どうやればいい？

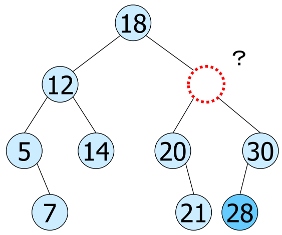


18

18

ノードの削除(5) 2つの子ノードを持つ(2)

- 削除対象ノードを置き換えるのに適切なノードは？
  - 二分探索木の条件
    - 各ノードにおいて自分より小さなデータを持つノードは左部分木に，大きなデータを持つノードは右部分木に存在する
  - 右の部分木の中で最小のデータを持つノード



19

19

ノードの削除(6) 2つの子ノードを持つ(3)

- 部分木における最小ノードの探索と削除
  - deleteMinメソッド
- 処理の流れ
  - 引数として部分木のルートノードを指定
  - 左の子ノードをたどっていき，左の子ノードがなくなれば探索を終了
    - 注意：右の子ノードは持っている可能性がある
    - 探索したノードを保持した上で，自身の右の子で置き換える
  - 最小のデータを持つノード＝右最小ノード

20

20

ノードの削除(7) 2つの子ノードを持つ(4)

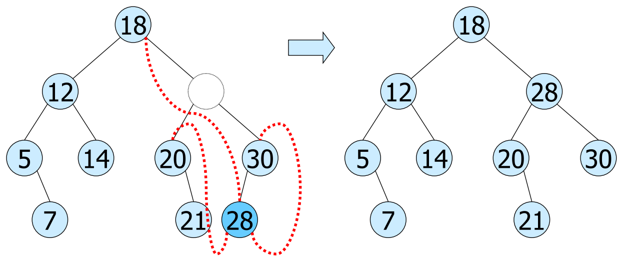
- ノードの置き換えの手順
  - 右最小ノードを右部分木から持ってくる
    - deleteMinメソッドを利用
  - 右最小ノードを親ノードの子に設定（左右どちらの場合もあるので注意）
  - 削除対象ノードの右子ノードを右最小ノードの右子ノードに設定
  - 削除対象ノードの左子ノードを右最小ノードの左子ノードに設定

21

21

ノードの削除(8) 2つの子ノードを持つ(5)

- 右最小ノードを削除対象のノードと置き換え

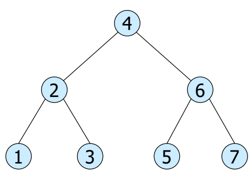


22

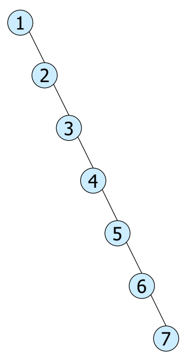
22

二分探索木の性質(1)

- 最良パターンと最悪パターン



完全二分木：
根からすべての葉までの
経路長が等しい



23

23

二分探索木の性質(2)

- 完全二分木
  - 厳密にはノード数が 2^n-1
  - 広い意味の完全二分木：根から葉までの経路差が高々1
- n個の要素を持つ完全二分木
  - 根から節への最大経路長：$\log_2 n + 1$
  - 平均経路長は $O(\log n)$
- 最悪パターンでは？
  - 根から節への最大経路長：$n - 1$
  - 平均経路長は $O(n)$

24

24

二分探索木の性質(3)

- 二分探索木の計算量は？
- データをランダムな順で挿入すると仮定すると
  - 根から節までの経路長の平均： $O(\log n)$
  - よって計算量も $O(\log n)$

25

25

二分探索木の性質(4)

- ハッシュ法（計算量$O(1)$）とは勝負にならない
- 二分探索木にメリットはある？
  - 最小（最大）の要素の探索・削除
    - どちらも $O(\log n)$ で実行可能
    - ハッシュ法は要素の大小関係が失われるため $O(n)$
  - 昇順での出力
    - 通りがけ順でなぞればOK
- ワーストケースを避けるには完全二分木に近くなるようにすればよい　→　平衡木

26

26

教科書 第10章（pp.243～282） ※注：B木は範囲外

平衡木

27

27

平衡木(balanced tree)

- 挿入・削除のたびに木の形を変形し，高さが $\log n$ 程度に収まるようにした木
- 木の形の見直しにかかる手間は $O(\log n)$ である必要

28

28

AVL木

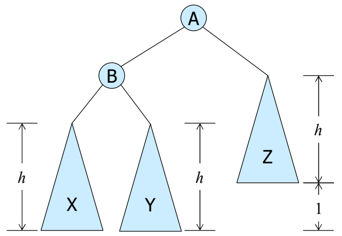
- 最初の平衡木
- 1962年にAdel’son-Vel’skiiとLandisが考案
- 最悪でも探索・挿入・削除が $O(\log n)$ で実行可能なことを初めて示した
- 現在ではより性能のよい木が存在
- AVL木の制約
  - すべての節で左部分木と右部分木の高さの差が1以内

29

29

AVL木の操作(1)

- 対象とするAVL木
  - Aの部分木：Z
  - Bの部分木：X，Y
  - 部分木X, Y, Zの高さ h は等しい
  - Aから見た高さの差は1



30

30

AVL木の操作(2)

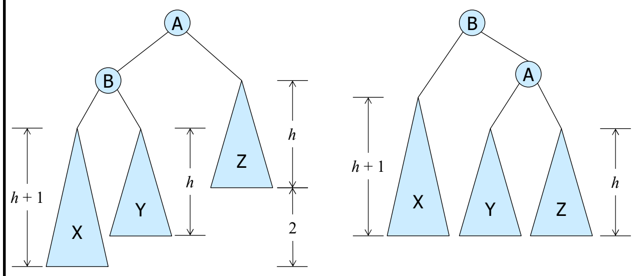
- 節Aの左部分木に要素を挿入
- 挿入結果は以下の２つのいずれか
  - パターン１
    - 節Ｂの左部分木に要素を挿入し，部分木Xの高さがh+1になった
  - パターン２
    - 節Ｂの右部分木に要素を挿入し，部分木Yの高さがh+1になった
- いずれにせよAVL木の要件を満たさなくなる

31

31

AVL木の操作(3) パターン1

- 節Aと節Bを入れ替える
  - 一重回転（single rotation）

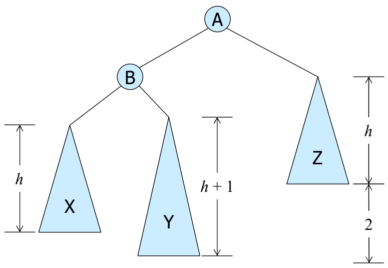


32

32

AVL木の操作(3) パターン2-1

- まずはYを分解
- 部分木Yは，根Cと左部分木Y_1，右部分木Y_2

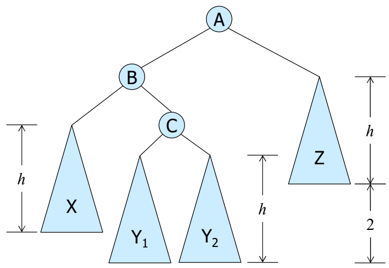


33

33

AVL木の操作(4) パターン2-2

- まずはYを分解
- 部分木Yは，根Cと左部分木Y_1，右部分木Y_2
  - 実際にはY_1とY_2の高さはどちらかが $h-1$ だが気にしない

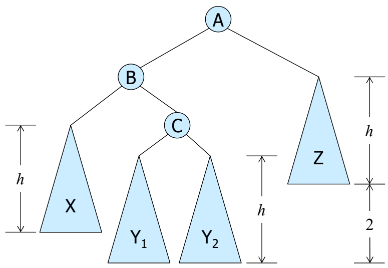


34

34

AVL木の操作(4) パターン2-2

- まずはYを分解
- 部分木Yは，根Cと左部分木Y_1，右部分木Y_2
  - 実際にはY_1とY_2の高さはどちらかが $h-1$ だが気にしない

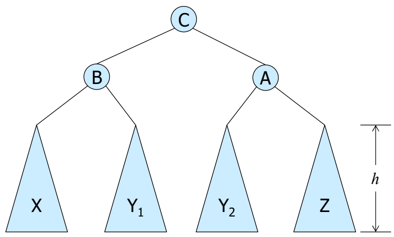


35

35

AVL木の操作(4) パターン2-3

- 節Cが親になるように節A，B，Cを入れ替える
  - 二重回転（double rotation）



36

36

AVL木の操作(5)

- 右部分木への挿入
  - パターン1，2を左右対称にすればOK
- バランスの見直し／修正は葉から親をたどって根へ
- 削除は？
  - 要素を削除してから回転してバランスを回復

37

37

AVL木の操作(6)

- 計算量は？
  - 一重回転，二重回転は1回あたり $O(1)$
  - 見直し対象の節は木の高さ程度：$O(\log n)$
  - 挿入の計算量は $O(\log n)$

38

38

B木

- B木はm分探索木
  - m分木：多分木（multi-way tree）
  - m分探索木：多分探索木（multi-way search tree）
- m階のB木の条件
  - 根は，葉であるか，あるいは $2 \sim m$ 個の子をもつ
  - 根，葉以外の節は $\lceil m/2 \rceil \sim m$ 個の子をもつ
    - $\lceil x \rceil$ はx以上の最小の整数　=　切り上げ
  - 根からすべての葉までの経路の長さが等しい
  - データを持つのは葉のみ

39

39

B木の特徴

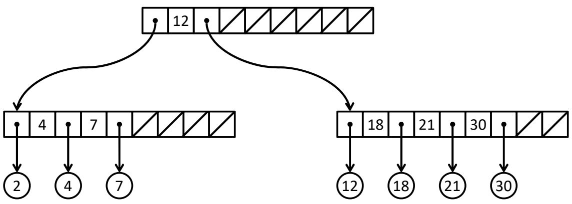
- B木は常にバランスがとれている
  - すべての葉までの経路長が等しい
- n 個の要素をもつB木の高さは $O(\log n)$
  - 最悪：$\log_{\lceil m/2 \rceil} n$
  - 最良：$\log_m n$

40

40

B木の例

- 節にはm個の子へのリンクとm-1個の境目を表す値
  - 境目の値はすぐ右の部分木の最小値
  - 部分木は左から右へ向かって小さいもの順に配置
- 5階のB木の例
  - 内部節は3～5個の子をもつ



41

41

まとめ

- 二分探索木
- 二分探索木の操作
  - 探索
  - 挿入
  - 削除
- 二分探索木の性質
- 平衡木とは
- AVL木
- B木（概要のみ）

42

42

参考文献

- 定本 Javaプログラマのための
  アルゴリズムとデータ構造（近藤嘉雪）
- 新・明解 Javaで学ぶ
  アルゴリズムとデータ構造（柴田望洋）
- 岩波講座ソフトウェア科学 3
  アルゴリズムとデータ構造（石畑清）
- Javaで学ぶアルゴリズムとデータ構造
  Robert Lafore（著）・岩谷 宏（翻訳）
- Java アルゴリズム+データ構造完全制覇
  オングス (著)・杉山 貴章・後藤 大地 (監修)

43