プログラム設計とアルゴリズム

3. ヒープ（ソートの基礎）

水谷　健太郎

非常勤講師

東京大学　大学院新領域創成科学研究科

特任助教

mizutani-kentaro@aoni.waseda.jp

2019年10月21日

前回の内容

- 「リスト構造」と呼ばれるデータ構造について学習した
  - データ＋ポインタ

- 「配列」と「リスト構造」との違いを学習した
  - 配列：添え字を使ってデータにランダムにアクセスできる
    リスト：先頭（または末尾）から順に辿っていくことしかできない→シーケンシャルアクセス
  - 配列：データの追加、削除の処理負荷が大きい
    リスト：データの追加、削除が容易

本日の内容

- ソートについて学びます
- 木構造の観点からアルゴリズムを見ていきます
  - 2進木
  - ヒープ

- クイックソート
- ヒープソート

- その他のソートアルゴリズム（一部は来週説明します）

ソート(sort)

- n 個の実数$(x_1, x_2, \ldots, x_n)$が与えられたとき、
  それらを小さい順に並べた$(y_1, y_2, \ldots, y_n)$を作ること

- 例:
  - (5, 4, 9, 10, 2)をソートすると
  - (2, 4, 5, 9, 10)となる

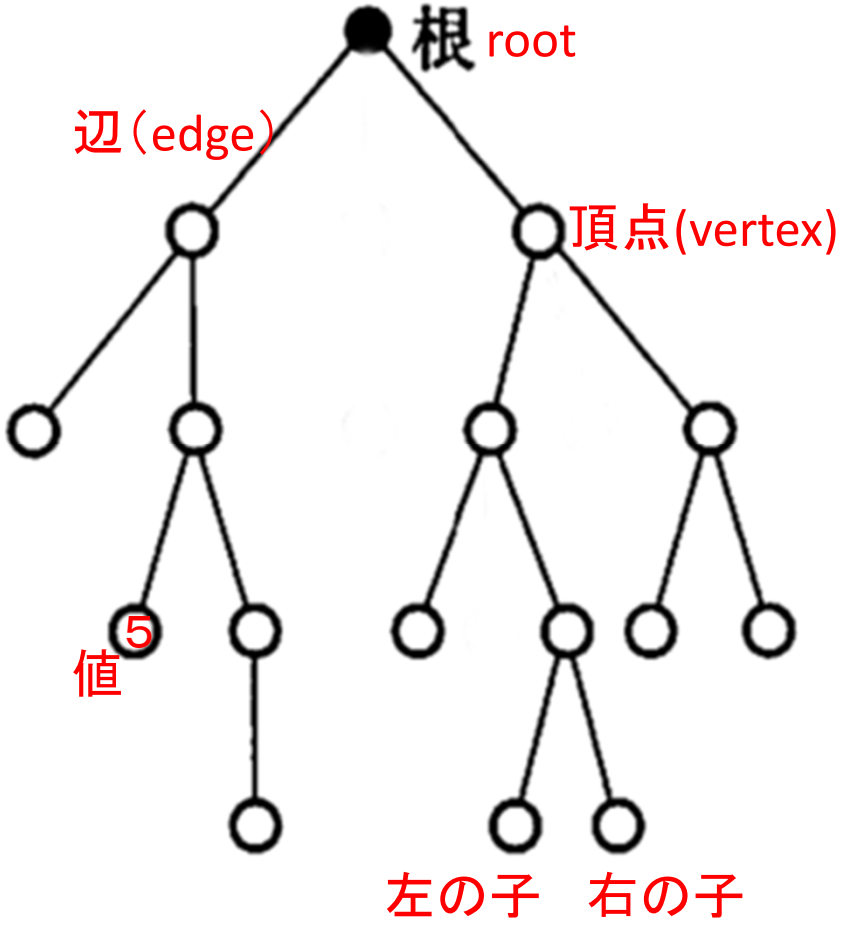
- いろいろな局面で出現する基本的な問題

- 本日は木構造とソートについて勉強する

2進木/2分木（binary tree）

- 本日のスライドでは「根付木」を単に「木」と呼ぶ
- 2進木（2分木）とは、木の中でそれぞれの頂点が持つ子の数が2以下のもの
- 1つの頂点に2つの子供を持つとき
  - 左の子
  - 右の子

  という名前を付けて区別する
    - 「順序」2進木
- 各頂点に格納した実数を、その頂点の値(value)と呼ぶ



それより下に枝が伸びていない頂点⇒葉

ソートのための2進木の構築

$x_1, x_2, \ldots, x_n$ が与えられている

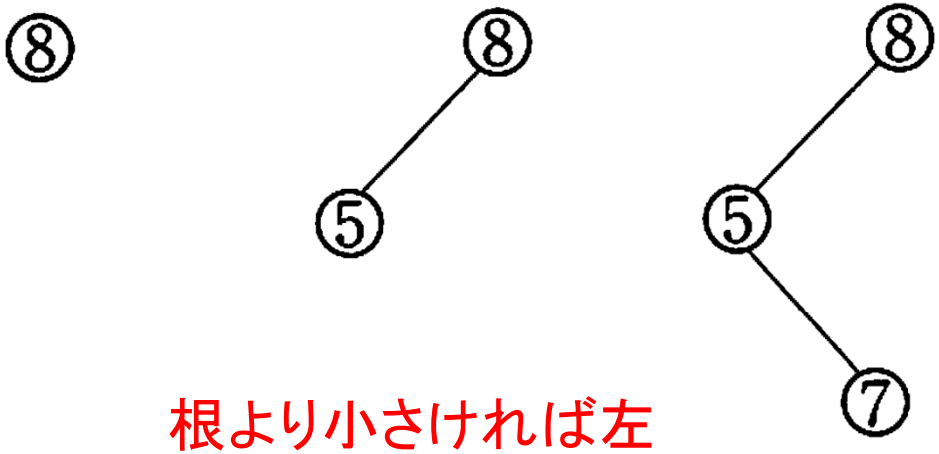
(1) まず x_1 を根の頂点に置く．

(2) $i = 2, 3, \cdots, n$ の順に x_i を読み込んだら，それを根の値と比べ x_i のほうが小さかったら左の子へ進み，そうでなければ右の子へ進む．以下，同じように，進んだ先の頂点の値と比べて，x_i のほうが小さかったら左の子へ進み，そうでなければ右の子へ進む．これをくり返すと，いずれ進んだ先に子の頂点が存在しない場所へたどり着く．そうしたらそこに頂点を設けて x_i を格納する．

- 理解できましたか？
- 例を見てみましょう

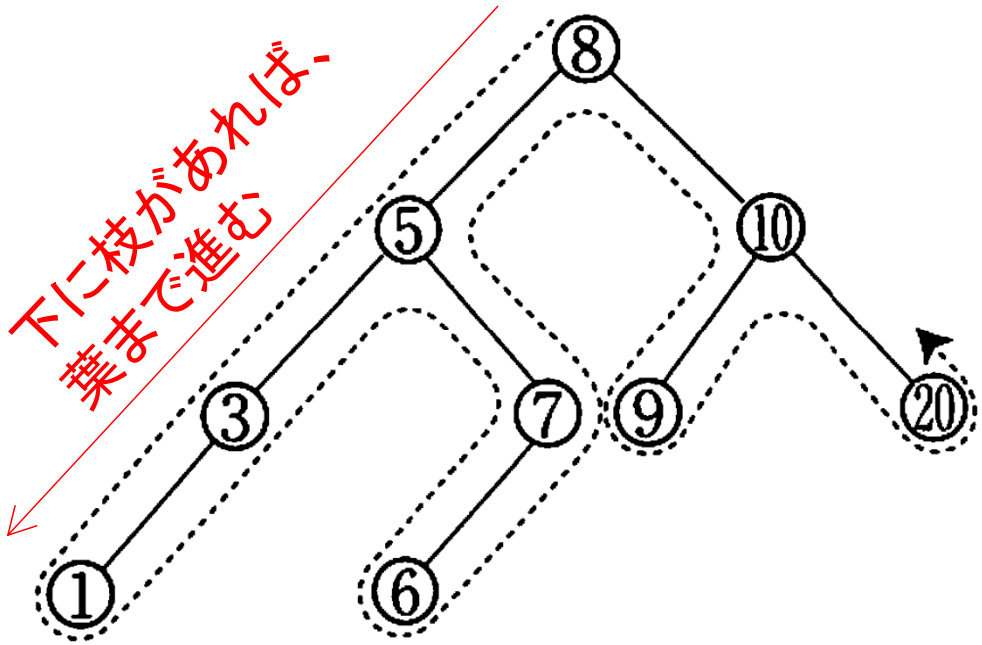
例：2進木の構築とソート

$$\{8, 5, 7, 3, 10, 9, 6, 1, 20\}$$



根より小さければ左
根より大きければ右
に置く

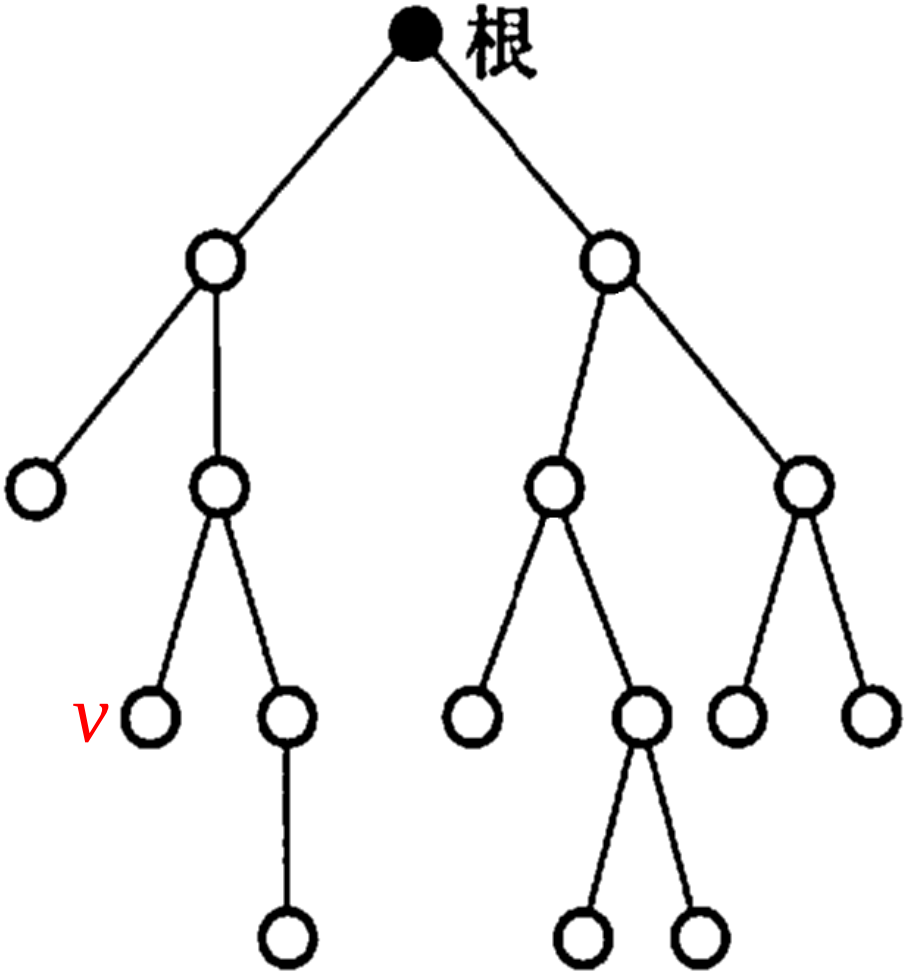
左の子ー自分ー右の子
の順に読み出す



この方法はソートを行うのに効率的な方法か？
⇒木のバランスがよければ（左右均等に広がる）効率的

深さと高さ

- 深さ（頂点に対して定義）
  - 木の一つの頂点 v に対して、根からvまでたどるとき通過する枝の数
  - 右図の頂点vの深さは 3
- 高さ（木に対して定義）
  - 全ての頂点の深さの最大値
  - 右図の木の高さは4
- 二進木では、
  - 子の数が2：完全頂点
  - 子の数が1 or 0：不完全頂点



強平衡2進木

- すべての不完全頂点の深さが高々1しか違わない2進木を強平衡2進木（strongly balanced binary tree）という

- 強平衡2進木の高さには次の定理が成り立つ

  定理：n 個の頂点をもつ強平衡2進木の高さは
  $O(\log n)$ である

定理の証明

T を高さ k の強平衡 2 進木とする．$i = 0, 1, 2, \cdots, k-1$ に対して，T は深さ i の頂点を 2^i 個もつ．そして深さ k の頂点は 1 個以上，2^k 個以下である．したがって，T の頂点数 n は

$$1 + 2 + 2^2 + \cdots + 2^{k-1} + 1 \leq n \leq 1 + 2 + 2^2 + \cdots + 2^k \qquad (3.1)$$

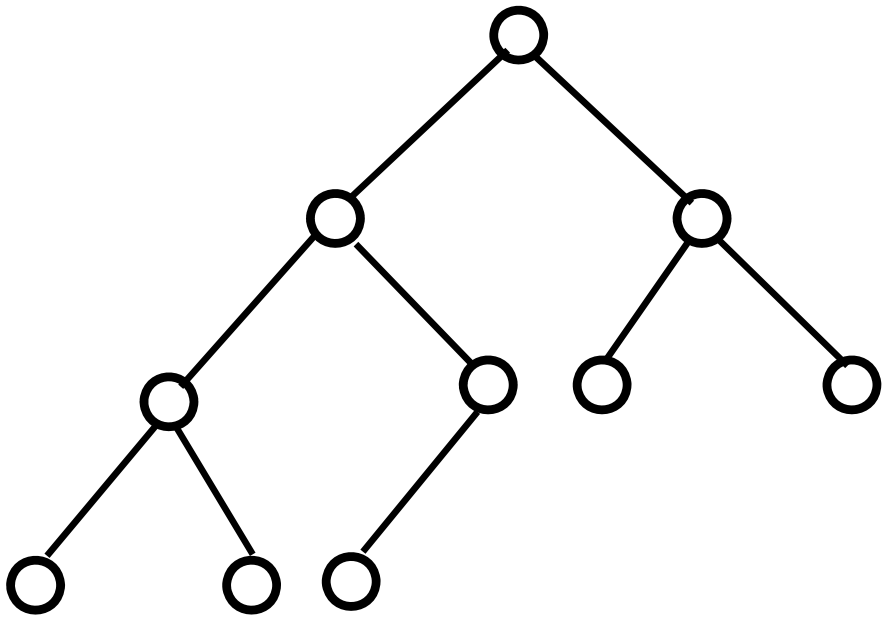
を満たす．一方

$$1 + 2 + 2^2 + \cdots + 2^k = 2^{k+1} - 1 \qquad (3.2)$$

である（演習問題 3.1）から，式 (3.1) は $2^k \leq n \leq 2^{k+1} - 1$ と書くことができる．左の不等号から $k \leq \log_2 n$ が得られ，右の不等号から $k \geq \log_2(n+1) - 1$ が得られるから $k = O(\log_2 n)$ である．

オーダの対数の底は省略可能（cf 底の変換公式とオーダの定義）

強平衡2進木の頂点数



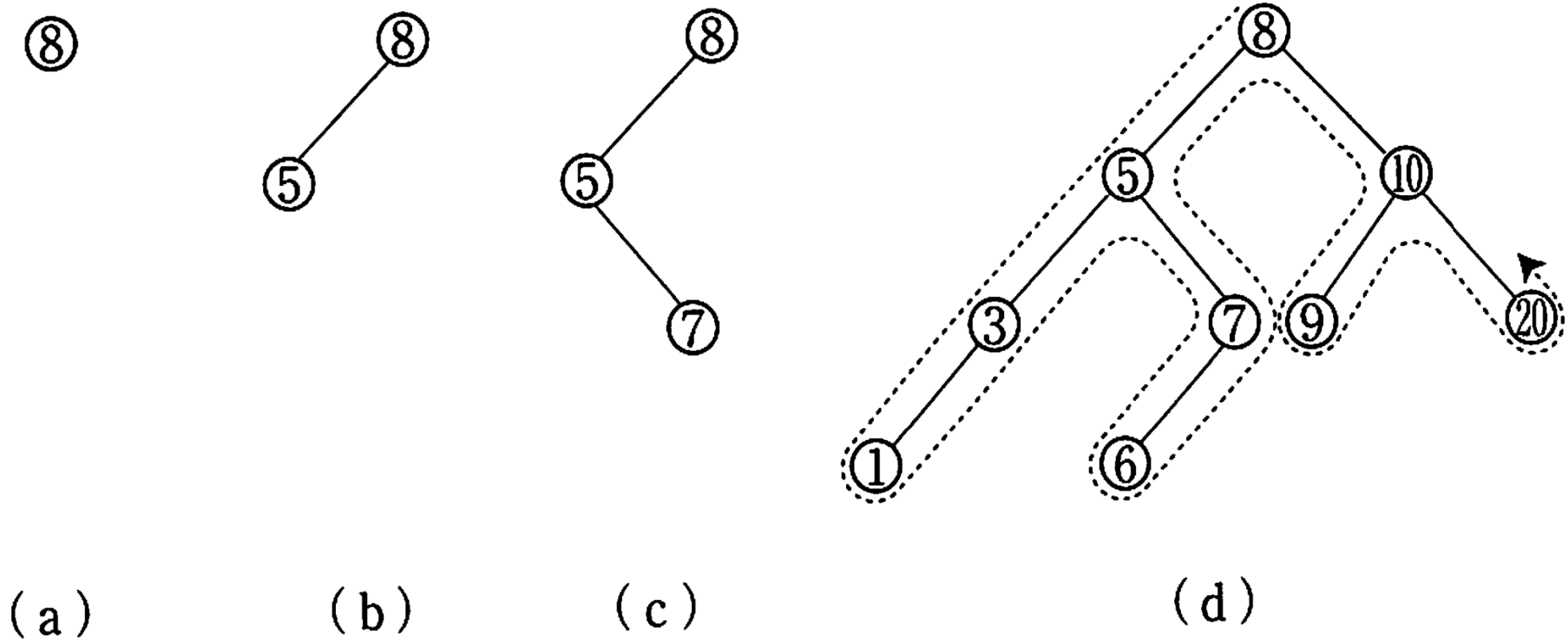
高さ k の強平衡2進木では、$i = 0, 1, 2, \ldots, k\text{-}1$に対して、

★深さ i $(< k)$の頂点数は　2^i

★深さ k の頂点数は　1以上2^k以下

2進木の構築(例)—再掲

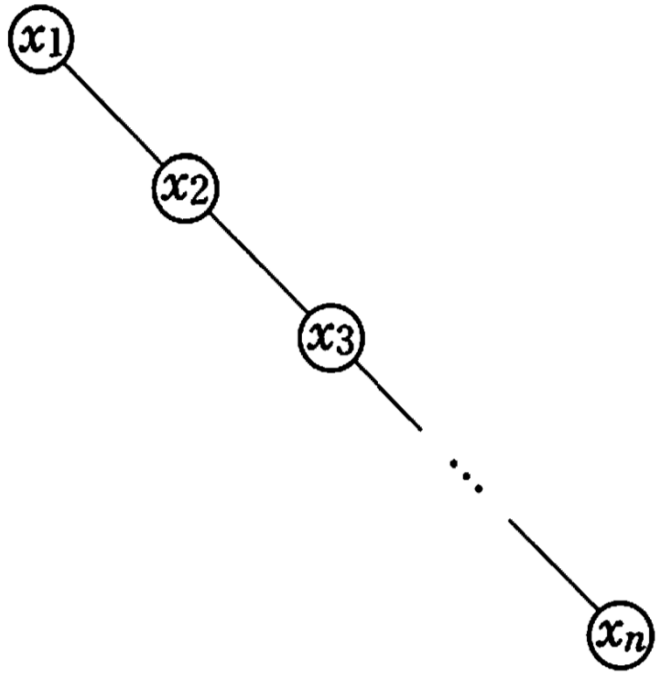
$$\{8, 5, 7, 3, 10, 9, 6, 1, 20\}$$



もし2進木の深さが$O(\log n)$であるならば、2進木の構築、値の読み出しの計算オーダは$O(n \log n)$となる

最悪の場合

- 値が最初から小さい順に並んでいた場合



頂点の格納時間

$$1+2+3+\cdots+n-1=\mathrm{O}(n^2)$$

クイックソート

- 入力をシャッフルした後で2進木を作れば、非常に高い確率で $O(n \log n)$ の計算時間を達成できる
- 値の読み出しも $O(n \log n)$

- 「クイックソート」（quick sort）とよばれる

- ただし、最悪の計算オーダは $O(n^2)$ となる

  ⇒最悪の計算オーダも $O(n \log n)$ とできないか？

  ⇒ヒープの登場
  - 特別な2進木を構築する

ヒープの定義

定義 3.2（ヒープ）　頂点 v に値 $f(v)$ が格納された高さ k の 2 進木 T が次の (i), (ii) を満たすとき，T をヒープ (heap) という．

(i) T では，深さ $k-1$ 以下の可能な頂点はすべて使われ，深さ k の頂点は左から順に使われている．　木の形に対する条件

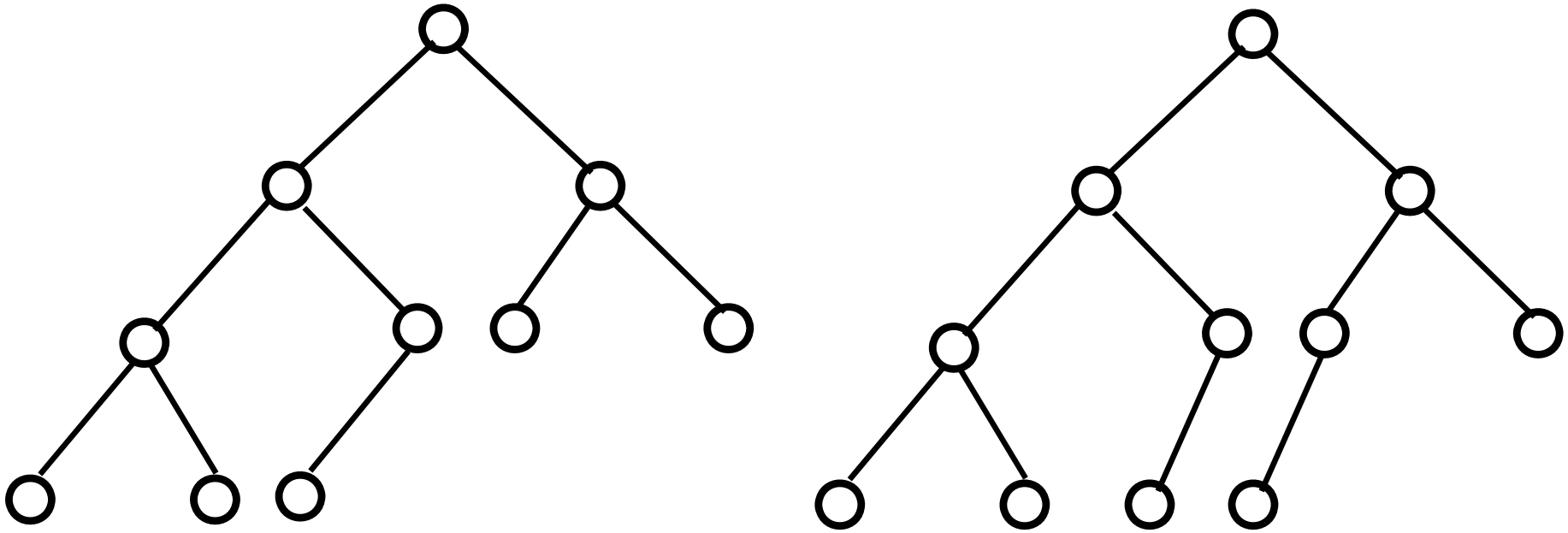
(ii) 頂点 u が頂点 v の親ならば $f(u) \geq f(v)$ を満たす．値に対する条件

注意）

メモリ管理におけるヒープ領域(heap memory)とは別物である（関係ない！）

練習：ヒープ or Notヒープ

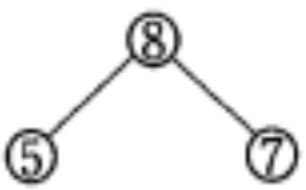
- 以下の木はヒープの条件を満たすか？



（ポイント）頂点数が決まると、ヒープの形は一意に決定する

ヒープの構築の仕方

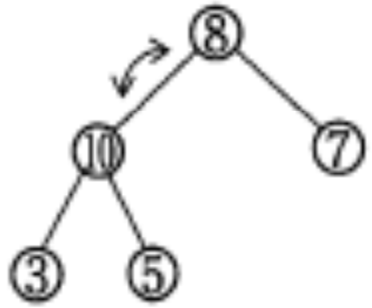
$$\{8, 5, 7, 3, 10, 9, 6, 1, 20\}$$



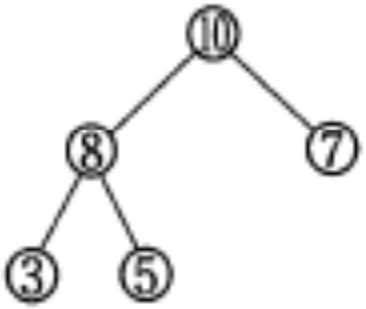
(a)

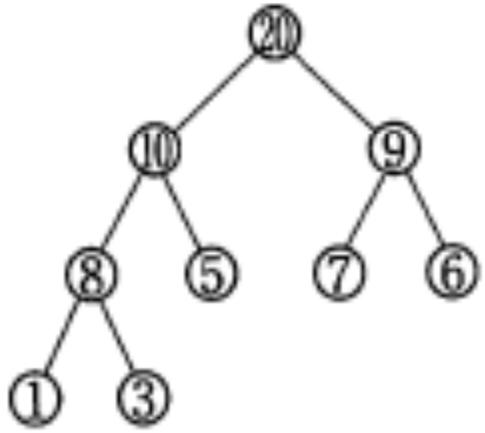
(b)

(c)



(d)



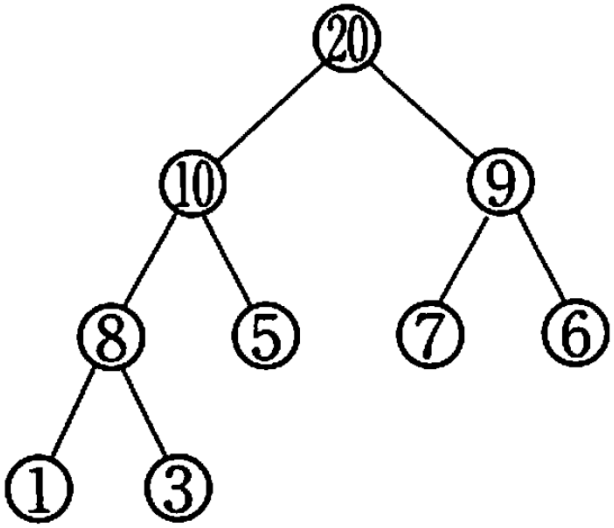


ヒープ構築の計算量

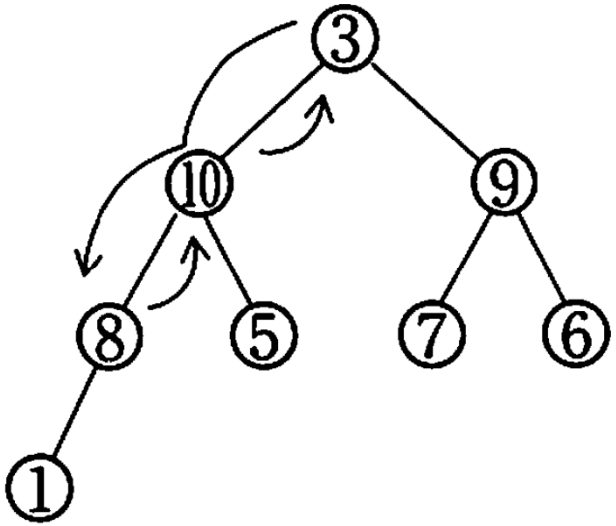
- $O(n \log n)$

- なぜなら、ヒープは強平衡2進木であるから高さは $O(\log n)$
- 各ノードの修正は高々$O(\log n)$回
- すなわち、ヒープを構築する計算量は$O(n \log n)$

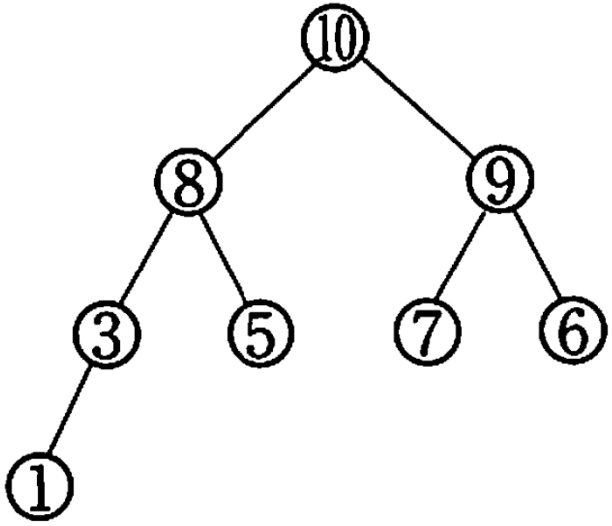
ヒープからソート結果を得る方法



まず根の「20」を読み出す
＝最大値
＝ソート結果の最後の値



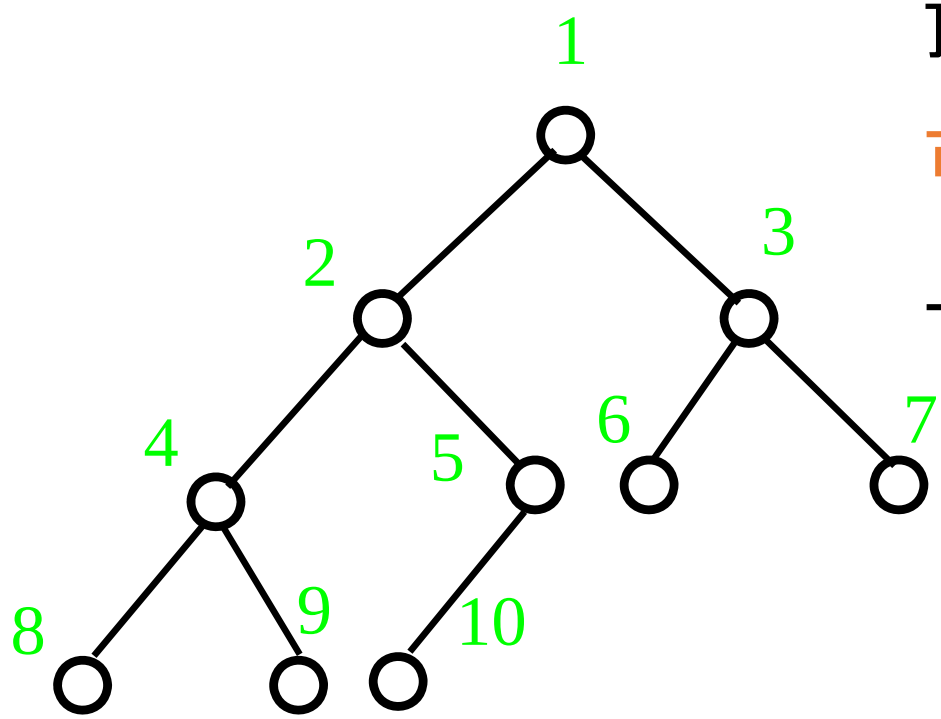
ヒープの条件を満たすように「3」を根に移動
⇒
ヒープの条件を満たすようにさらに修正
ポイント：子の値の大きい頂点と交換する



根の「10」を読み出す
＝2番目に大きい数

読み出しの計算量は$O(n \log n)$
この方法に従ってソートをする方法を「ヒープソート」と呼ぶ

ヒープソート



頂点の番号のつけ方：

可能な頂点すべてに、

上から下、左から右

iの親の番号は、

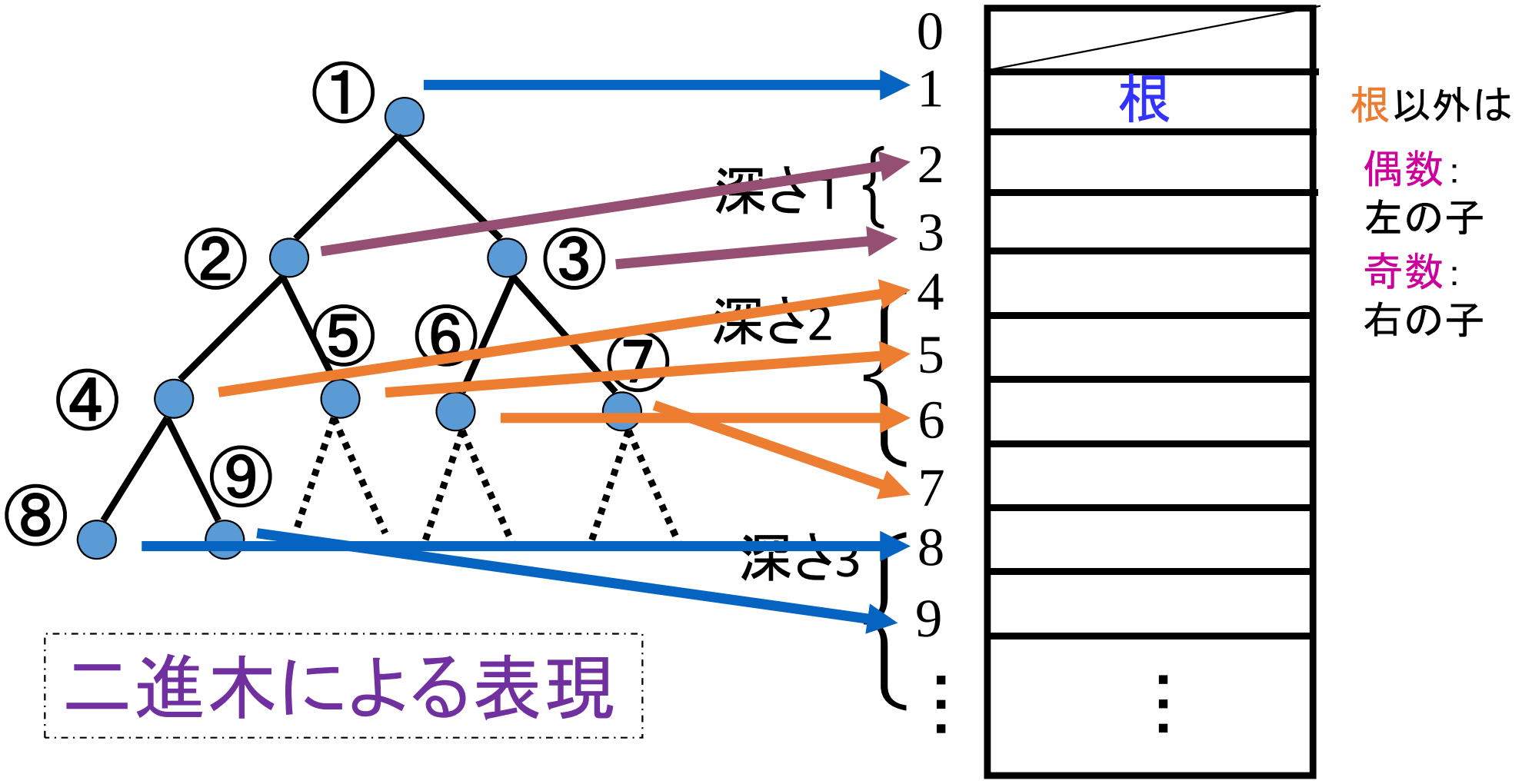
$\lfloor i/2 \rfloor$

この番号の付け方から、高さ k $(> j)$のヒープでは、

★深さ j の最も左の頂点の番号は　2^j

★深さ j の最も右の頂点の番号は　$2^{(j+1)} - 1$

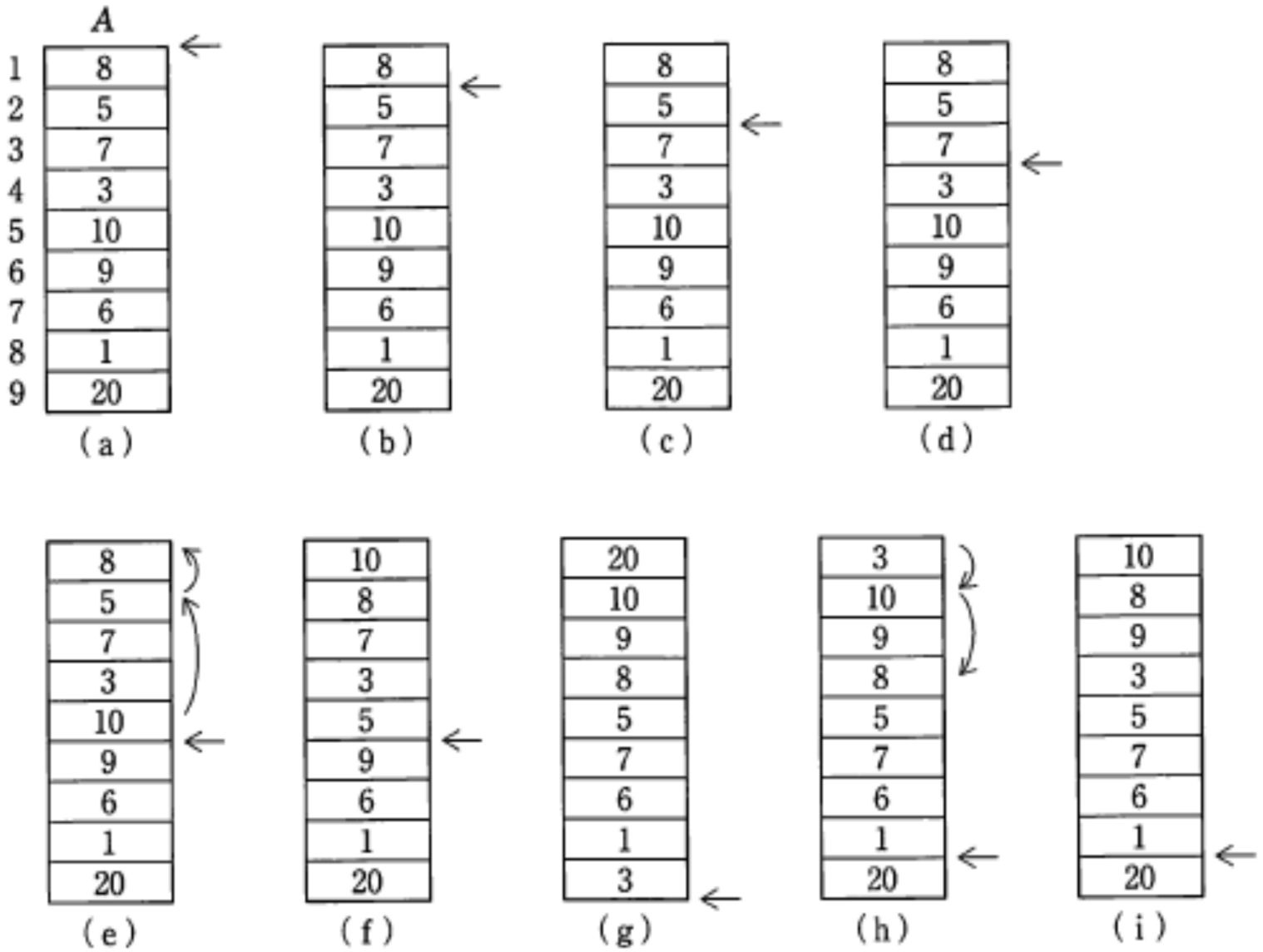
ヒープは「配列」を使って簡単に表せる



二進木による表現

配列による表現

配列を用いたヒープソート



空間計算量：$O(n)$

[参考]その他のソートアルゴリズム

- バブルソート：$O(n^2)$
- 双方向バブルソート（シェーカーソート）：$O(n^2)$
- 挿入ソート: $O(n^2)$
- 選択ソート：$O(n^2)$
- マージソート: $O(n \log n)$
  - 次々回
- 基数ソート（バケットソート）: $O(n)$
  - 次回

ただし、特別な場合にのみ利用可能

一般的なソートの時間複雑度の最速オーダは？

（参考）Wikipedia　ソート

https://ja.wikipedia.org/wiki/%E3%82%BD%E3%83%BC%E3%83%88

バブルソート

初期データ: 8 4 3 7 6 5 2 1

左から順に見ていき、大小が逆だったら数字を入れ替える。

結果が確定した部を太字で示す。

4 3 7 6 5 2 1 **8** （1回目の外側ループ終了時 交換回数:7）

3 4 6 5 2 1 **7 8** （2回目の外側ループ終了時 交換回数:5）

3 4 5 2 1 **6 7 8** （3回目の外側ループ終了時 交換回数:3）

3 4 2 1 **5 6 7 8** （4回目の外側ループ終了時 交換回数:2）

3 2 1 **4 5 6 7 8** （5回目の外側ループ終了時 交換回数:2）

2 1 **3 4 5 6 7 8** （6回目の外側ループ終了時 交換回数:2）

1 2 3 4 5 6 7 8 （7回目の外側ループ終了時 交換回数:1）

交換回数の合計: 7+5+3+2+2+2+1=22

- $O(n^2)$ のアルゴリズム
- 実装が簡単
- https://ja.wikipedia.org/wiki/%E3%83%90%E3%83%96%E3%83%AB%E3%82%BD%E3%83%BC%E3%83%88

演習課題・レポート課題

- ソートのアルゴリズムを2つ以上を実装する
- Course N@viからファイルをダウンロードする
  - Report3.java or report3.c
  - ランダムに整数列を生成するコード、ソートをチェックするコード、実行時間を計測するコードも入っています
- 実装ができたら計算量の理論値と実測値の関係について考察する
  - $O(n^2)$のアルゴリズムと$O(n \log n)$のアルゴリズムを実装し、速度の違いを調べる
  - 最悪のケースを再現してみる　・・・等
- ∝切：10月27日（日）23:59
  - ソースコードとレポートを提出すること
  - 動作確認は自分でもちゃんと行う