KYOTO UNIVERSITY

アルゴリズムとデータ構造③
～ リスト・ソート・ヒープソート ～

鹿島久嗣
（計算機科学コース）

DEPARTMENT OF INTELLIGENCE SCIENCE AND TECHNOLOGY

リスト

集合を管理するデータ構造：データを保持するための基本データ構造

- 集合を管理するデータ構造
  - データをコンピュータのメモリにどのように保持するか
- サポートすべき機能：
  - 集合の追加
  - 集合の削除
  - 集合の検索
- たとえば、配列ならば…：
  - メモリを必要分確保しておき、順次保管する
  - 所望の位置にアクセス可能だが、削除が面倒

リスト：
集合を管理する基本データ構造

- リスト：データをポインタで一列につなげたもの
  - ポインタ：次のデータの場所（番地）を示す

リストの利点：
データを動的に追加・削除可能

- 何が得するか：
  - 必要に応じてメモリを確保できる
  - 追加・削除が容易
    - 配列でもつと削除が大変
  - 検索は得しない（それは別のしくみ）
- 削除：ポインタの付け替えで対応
  - ポインタのつけかえには、誰が自分にポインタを指しているかを知る必要がある（単純には$O(n)$）
    - 二重線形リスト： $O(1)$ で発見可能
    - 実は二重にしなくても可能：たどる→コピー→付け替え

根付き木：
枝分かれするリスト

- 根付き木：枝分かれするリスト
  - 頂点集合とそれら結ぶ辺からなる
    - 辺に接続する頂点の片方が親でもう一方を子とする
    - 各頂点は0～複数個の子をもつ
    - 根以外の頂点は、必ずただひとつの親頂点をもつ
    - 葉：子をもたない頂点
  - 各頂点は親へのポインタ、次の兄弟へのポインタ、最初の子へのポインタをもつ
    - 全ての子へのポインタをもつかわりに最初の子だけを指す
      - 各頂点は最大3個のポインタを保持
  - 部分木：ある頂点以下の部分

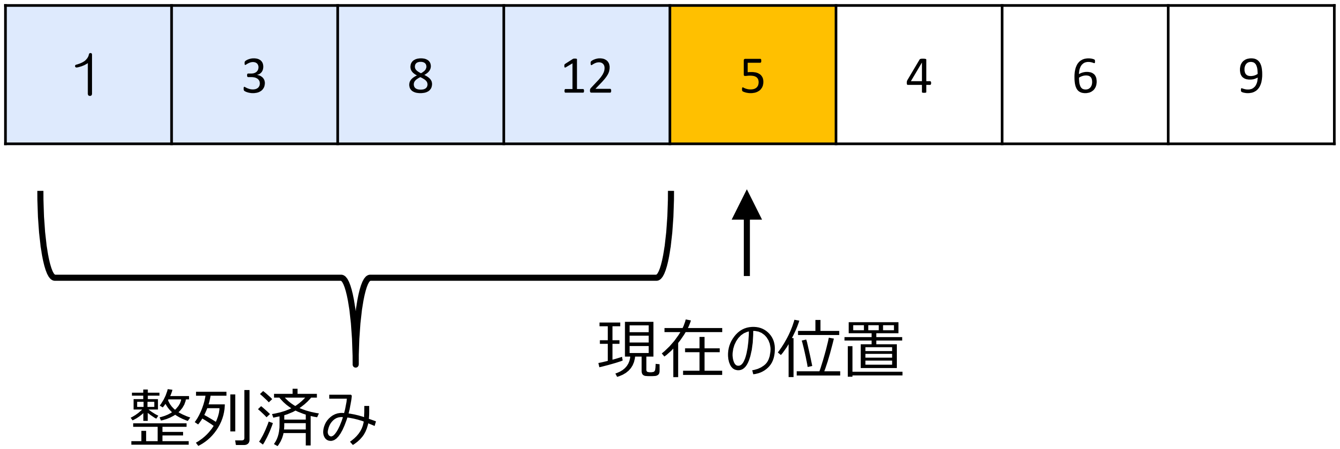
整列（ソート）のアルゴリズム

整列問題（ソート）：要素を小さい順に並び替える問題

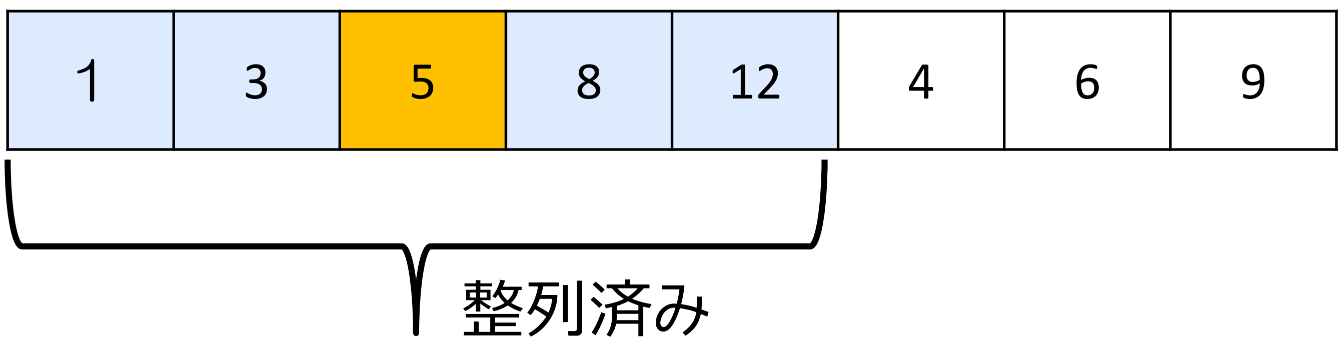
- 整列問題
  - 入力：n個の数 $a_1, a_2, \ldots, a_n$ が入った配列
  - 出力： $a_{1'} \leqq a_{2'} \leqq \ldots \leqq a_{n'}$ を満たす、入力列の置換
- 例：入力 $(4, 5, 2, 1) \to$ 出力 $(1, 2, 4, 5)$

単純なソートアルゴリズム：
ソート済み領域を左から順に拡大していく

- 現在の位置よりも左はすでに整列済みとする



- 現在の位置から左に見ていき、順序が保たれるところまで移動する



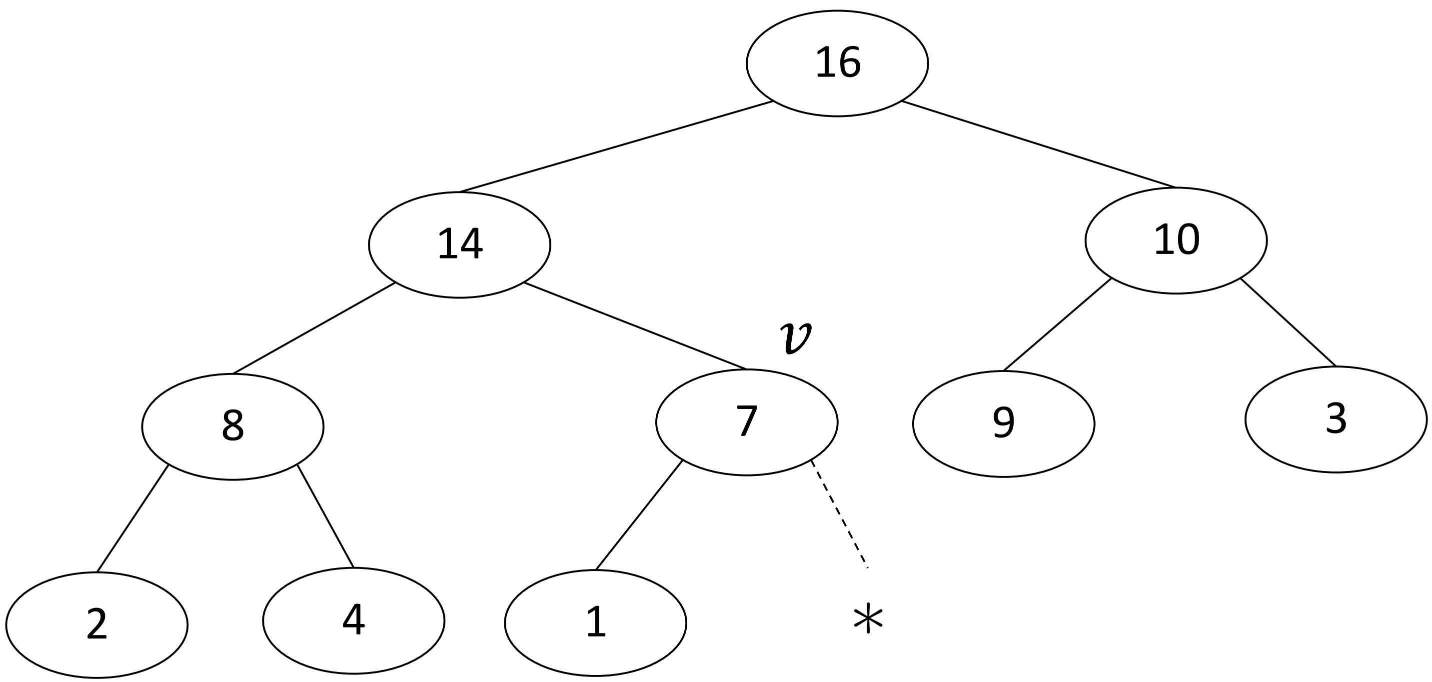
単純なソートアルゴリズムの計算量：計算効率はそれほど良くないが省スペースで実行可能

- 「現在の位置から左に見ていき、順序が保たれるところまで移動する」アルゴリズム
- 「」の操作には、現在の位置をjとすると$O(j)$回の交換が必要
- これを $j = 1,2,...,n$ まで行うと
  $\Sigma_{j=1,...,n} O(j) = O(n^2)$ なので
  あまり効率はよくない （良いアルゴリズムは$O(n \log n)$）
- ただし、「その場でのソート」が可能なので省スペース
  - 入力配列以外に定数個の領域しか使用しない

ヒープソート

ヒープソート：
データ構造「ヒープ」を使った$O(n \log n)$のソート法

- 「ヒープ」とよばれるデータ構造の一種を用いたソート法
- $O(n \log n)$ で動く「その場での」ソート法
  - $O(n \log n)$は最悪計算量としてはベスト



ヒープ：
ヒープ条件をみたす完全2分木

- ヒープ
  - （ほぼ）完全2分木
    - 2分木：全頂点の子数が最大2個の根付き木
    - 完全2分木：葉以外の頂点の子がちょうど2個で、すべての葉の高さが等しい2分木
  - 各頂点はデータをひとつずつもち、必ず「ヒープ条件」を満たしている
    - ヒープ条件：ある頂点のデータの値は、その親のもつデータの値以下である

$$A[\mathrm{parent}(i)] \geq A[i]$$

  - n頂点をもつヒープの高さは $\Theta(\log n)$

ヒープの表現：
ヒープは配列で一意に表現できる

- ヒープ = 配列

A	16	14	10	8	7	9	3	2	4	1
	1	2	3	4	5	6	7	8	9	10

  – $A[1] = $ 根

- 配列表現の性質：
  – 頂点iの左の子は$2i$番目、右の子は$2i + 1$番目
  – 頂点iの親は$\lfloor i/2 \rfloor$ 番目に入っている

ヒープソート：おおまかな流れ

- ヒープの根には最大の値が入っている
- 大まかには以下の方法で小さい順に並べることができる：

1. ヒープを構成する（ $O(n)$ ）
2. 根と、最も深く、最も右にある頂点（＝配列表現の場合は一番最後の要素）と交換する
3. 木（＝配列）のサイズをひとつ小さくする
4. 根が入れ替わったことでヒープ条件が満たされなくなっているので、ヒープを更新（ $O(\log n)$ ）する
5. 以上を頂点がなくなるまで繰り返す（→ステップ2）

ヒープの構成：
木の下方から上方に向かって構成する

- 手続き：木の下から上に向かって（ヒープになっていない）木（＝配列）をヒープにする
- BUILD_HEAP(A)

子のある頂点を添え字の大きいほうから順に

1. for $i \leftarrow \lfloor \mathrm{length}(A)/2 \rfloor$ down to 1
2. do HEAPIFY(A, i)
3. end for

i番目の頂点を根とする部分木がヒープ条件を満たすように更新する

- HEAPIFYが$\mathrm{O}(\log n)$でできるとすると、全体としては$\mathrm{O}(n)$回の呼び出しで$\mathrm{O}(n \log n)$
  – 実は$\mathrm{O}(n)$で構成可能（※ あとで示す）

HEAPIFY関数の中身：
ある頂点以下のヒープ条件を$O(\log n)$で回復する

- HEAPIFY(A, i)は配列A（を木としてみたときの）頂点i以下の頂点をヒープ条件を満たすように更新する
- 仮定：頂点iの２つの子を根とする部分木はすでにヒープ条件を満たしているとする
- アルゴリズムは木の上から下へ向かって動く
- $O(\log n)$で実行可能
- ヒープソートにおけるヒープ更新：根でHEAPIFYを実行する

HEAPIFY関数の中身：ある頂点から下へ向かって$O(\log n)$で実行可能

HEAPIFY(A, i)

1. i からスタート
2. i とその左右の子を比較
   - if i が最大 then 終了
   - else
     - i を大きい方と入れ替える
     - $i \leftarrow$入れ替えられた先の位置
     - HEAPIFY(A, i)

i を２つ子の間のヒープ条件は満たされる

新しいi とその子の間のヒープ条件は不明

- 計算量はiの高さをhとして$O(h) \leq O(\log n)$

ヒープへの挿入：
$O(\log n)$で実行可能

- ヒープに新たなデータxを挿入する

HEAP_INSERT(A, x)

1. 配列A の最後にxを付け加える
2. xとparent(x)を比較する
   - if $x \leq \mathrm{parent}(x)$ then 終了
   - else xとparent(x) を入れ替える
3. $x \leftarrow \mathrm{parent}(x)$
4. goto 2

ヒープ条件の確保

繰り返し回数は $O(\log n)$

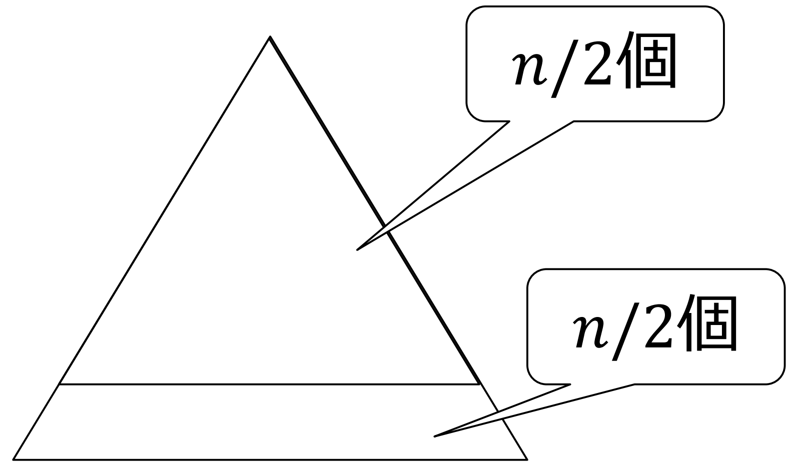
- これを繰り返すことでヒープ構成も可能$O(n \log n)$

ヒープ構成の計算量：挿入の繰り返しでも構成可能だが遅くなる

- HEAPIFYとHEAP_INSERTのどちらでもヒープを構成可能：
  - HEAPIFYは上から下に向かってヒープ条件を回復
  - HEAP_INSERTは下から上に向かってヒープ条件を回復
- 計算量は異なる：
  - HEAPIFYを使った構成は$O(n)$
  - HEAP_INSERTは$O(n \log n)$
  - 計算量の差はどこからくるか：
    - 2分木は下のほうの頂点数が多い
    - ほとんどの頂点にとって 根からの距離 > 葉への距離

ヒープ構成の計算量：
HEAPIFYなら線形時間で構成可能

- 高さhの位置に$n/2^h$個の頂点がある
  - 一番下の段にほぼ半分が
  - 次の段には、残りのうちほぼ半分が
  - ・・・
- $\sum_{h=1}^{\log n} h \cdot \frac{n}{2^h}$を評価すると$O(n)$



ヒープの応用：
プライオリティ・キュー

- 優先度順にオブジェクトを取り出す仕組み
- 計算機のジョブ割り当て：
  - ジョブが終了 or 割り込み → 最大優先度のものを取り出す
  - 新しいジョブはINSERT
- シミュレーション：
  - 優先度＝時間として、時刻順にイベントを取り出す