本読みゼミ第 8 回

ソートと順序統計量

高橋 優輝

森田研究室 B4

2022.6.2

目次

配列の添字について，本スライドでは『アルゴリズムイントロダクション』と同じく，1-index です．

目次

ソートとは？

ソーティング問題 (sorting problem) を以下のように定義する.

ソーティング問題

入力: n 個の数の列 $\langle a_1, a_2, \cdots, a_n \rangle$
出力: 入力を並び替えた $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ を満たす列 $\langle a'_1, a'_2, \cdots, a'_n \rangle$

ソーティングアルゴリズム

本スライドでは以下のソーティングアルゴリズムを紹介する．これらのアルゴリズムについて，以下の表でまとめられた点を中心に議論していく．

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

表の列の見出しの用語について，次のスライドで説明する．

最悪・平均・期待実行時間

最悪実行時間

サイズ n の任意の入力に対する最長の実行時間.

平均実行時間

全ての可能な入力に対する実行時間の平均.

期待実行時間

乱数生成器が返す値の分布の上で実行時間の期待値を取ったもの.

※平均実行時間は平均を取る対象が入力全体であり，期待実行時間は平均を取る対象が乱数全体である.

内部・安定・比較ソート

その場での (in place) ソート

入力配列以外の場所に格納しなければならない要素数がある定数で抑えられるとき，そのソーティングアルゴリズムを**その場での** (in-place) ソートと呼ぶ．

※表ではその場でのソートを内部ソートと言い換えている．

安定性

同じ値の要素は入力に出現する順序で出力に出現するという性質．

※安定性を持つソートを安定ソートという．

比較ソート

「ソート順は入力要素の比較にのみ基づいて決定される」という性質を持つソート．

ソーティングアルゴリズム

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

目次

挿入ソート

以下は，挿入ソートの疑似コードである．INSERTION-SORT は長さ $n(= A.length)$ の数列 A を引数として取る．

Algorithm INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

$\longrightarrow$ 最悪実行時間 $\Theta(n^2)$, 内部ソートかつ安定ソートかつ比較ソート．
平均実行時間は？

挿入ソートの平均実行時間

前回のゼミより，挿入ソートの実行時間 $T(n)$ は，擬似コードの各行の時間コストを c_i とすると，

$$T(n) = \Theta(n) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1)$$

と表される．
平均的な入力について考えると，$A[1..j-1]$ の半分の要素が $A[j]$ 以上で，半分がそれ以下となる．したがって，平均的には部分列 $A[1..j-1]$ の半分をチェックすることになるから，$t_j = j/2$ である．この平均的な場合の実行時間 $T_{ave}(n)$ は，

$$\begin{aligned} T_{ave}(n) &= \Theta(n) + c_5 \sum_{j=2}^{n} \frac{j}{2} + c_6 \sum_{j=2}^{n} (\frac{j}{2} - 1) + c_7 \sum_{j=2}^{n} (\frac{j}{2} - 1) + c_8(n-1) \\ &= \Theta(n^2) \end{aligned}$$

となる．よって，挿入ソートの平均実行時間は $\Theta(n^2)$ である．

挿入ソートまとめ

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

マージソート

次ページはマージソートの補助手続き MERGE の擬似コードである．引数は，配列 $A, p \leq q < r$ を満たす配列の要素を指す添字の p, q, r である．

Algorithm MERGE(A, p, q, r)　　(A:配列,p, q, r:配列の要素 $(p \leq q < r)$)

```
1  n_1 = q − p + 1
2  n_2 = r − q
3  create arrays L[1 .. n_1 + 1] and R[1 .. n_2 + 1].
4  for i = 1 to n_1
5      L[i] = A[p + i − 1]
6  for j = 1 to n_2
7      R[j] = A[q + j]
8  L[n_1 + 1] = ∞
9  R[n_2 + 1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

マージソート

次ページは MERGE を用いた MERGE-SORT の擬似コードである．
配列全体 $A = \langle A[1], A[2], \cdots, A[n] \rangle$ をソートするための初期呼び出しは，MERGE-SORT$(A, 1, A.length)$ である．

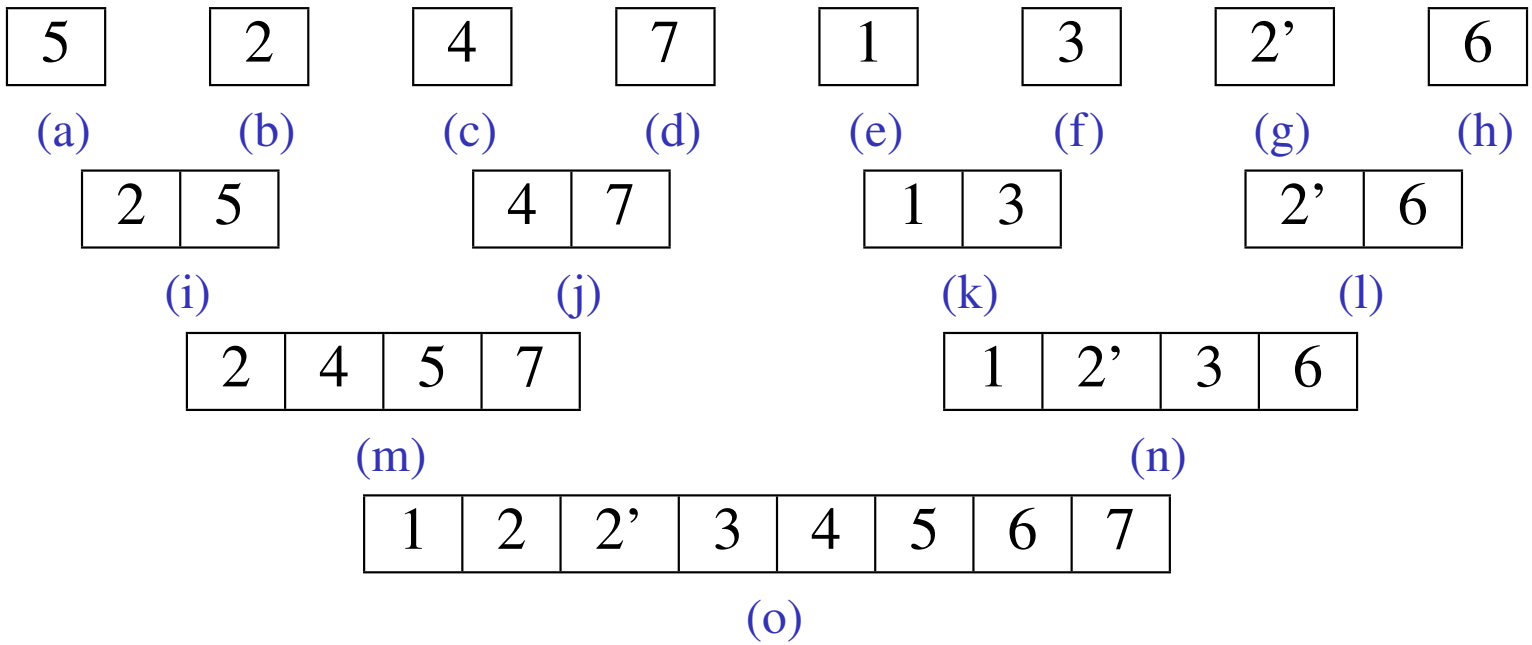
マージソートの擬似コード (MERGE-SORT)

Algorithm MERGE-SORT(A, p, r)　(A:配列,p, q, r:配列の要素$(p \leq q < r)$)

```
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

$\longrightarrow$ 最悪実行時間 $\Theta(n \lg n)$, 内部ソートでなく，安定ソートであり，比較ソートである．
平均実行時間は？

マージソートの動作例



配列 $\langle 5, 2, 4, 7, 1, 3, 2', 6\rangle$ 上での MERGE-SORT の動作．1 行目 ((a)〜(h)) が初期配列であり,2 行目 ((i)〜(l)) は 1 行目の要素を 2 つずつマージしたものである．同様に，3 行目 ((m),(n)) は 2 行目の要素をマージしたものであり，それらをマージしたものが (o) であり，ソート済み配列である．

マージソートの平均実行時間

マージソートでは，どのような入力に対しても実行時間が $\Theta(n \lg n)$ である．
すなわち，平均実行時間は $\Theta(n \lg n)$ である．

マージソートまとめ

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

目次

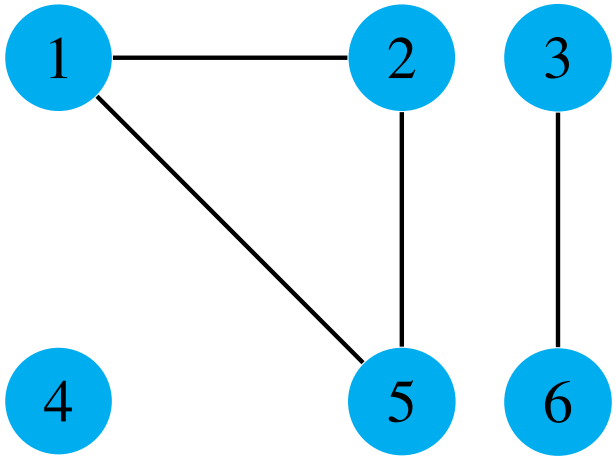
定義の確認

まず，第 6 章で用いる以下の用語について，定義を確認する．

無向グラフ，道，到達可能, 非巡回, 連結，森，(自由) 木，根付き木，
部分木，親・子，節点の次数・深さ，2 分木, 完全 k 分木,
おおよそ完全 2 分木

無向グラフ　$G = (V, E)$

無向グラフ $G = (V, E)$ は**頂点集合** V と**辺 (枝) 集合** E からなるグラフである．ここで，辺集合 E は頂点の**非順序対**の集合である．すなわち，辺は $u, v \in V$ かつ $u \neq v$ を満たす集合 $\{u, v\}$ である．
慣例に従って，辺を集合表記 $\{u, v\}$ の代わりに (u, v) を使用し，(u, v) と (v, u) は同じ辺を表すものと考える．
無向グラフでは，ある頂点からそれ自身に向かう辺である**自己ループ**は許さず，すべての辺は 2 つの異なった頂点から構成される．



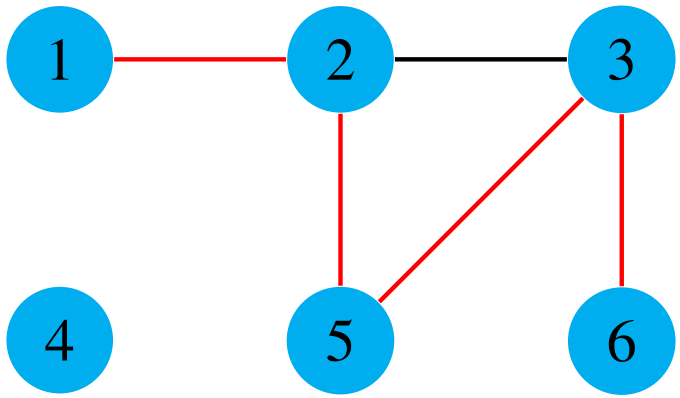
無向グラフ $G = (V, E)$.
ここで，$V = \{1, 2, 3, 4, 5, 6\}, E = \{(1, 2), (1, 5), (2, 5), (3, 6)\}$ である．頂点を円，辺を線分で表している．

道

グラフ $G = (V, E)$ の頂点 u から u' への**長さ** k の**道**は頂点の列 $\langle v_0, v_1, \cdots, v_k \rangle$ で，$u = v_0, u' = v_k$ および $i = 1, 2, \cdots, k$ に対して $(v_{i-1}, v_i) \in E$ が成り立つものである．道の長さは道の中の辺の数である．この道は頂点 $v_0, v_1, \cdots, v_k$ と辺 $(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)$ を含む．道の中の全ての頂点が異なるとき，この道を**単純道**という．

到達可能

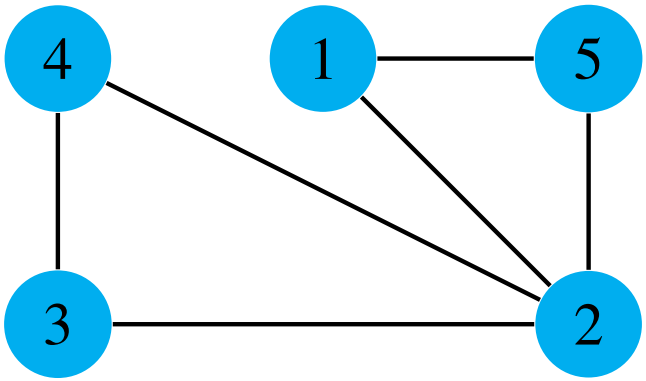
u から u' への道 p が存在するとき，u' は u から p を辿って**到達可能**であるという．



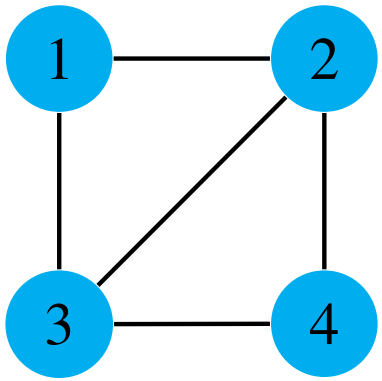
頂点の列 $p = \langle 1, 2, 5, 3, 6 \rangle$ は長さ 4 の道である (単純道)．頂点 4 は頂点 1 から p を辿って到達可能でない．

非巡回

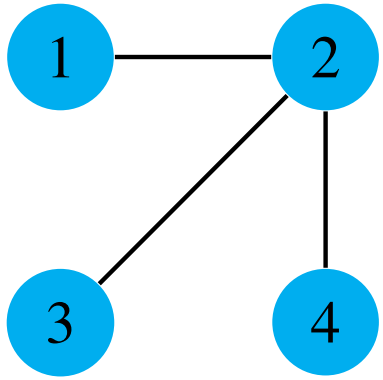
無向グラフの道 $\langle v_0, v_1, \cdots, v_k \rangle$ は，$k > 0$ かつ $v_0 = v_k$ かつ道の上のすべての辺が異なるとき**閉路**を構成する．さらに，頂点 $v_1, v_2, \cdots, v_k$ がすべて異なるとき，閉路は**単純**であるという．単純閉路を持たないグラフは**無閉路**あるいは**非巡回**であるという．



(a) 道 $\langle 1, 2, 3, 4, 2, 5, 1 \rangle$ は単純でない閉路



(b) 道 $\langle 1, 2, 3, 1 \rangle$ は単純閉路



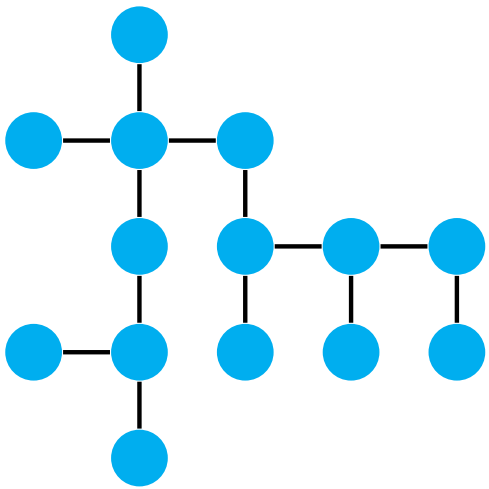
(c) 非巡回無向グラフ

連結

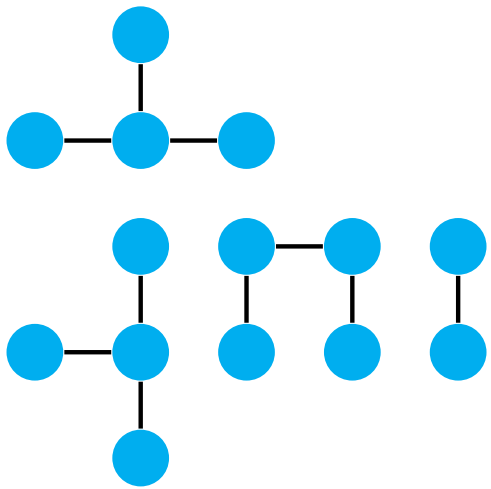
すべての頂点が別のすべての頂点から到達可能であるとき，無向グラフは**連結**であるという．

森，木

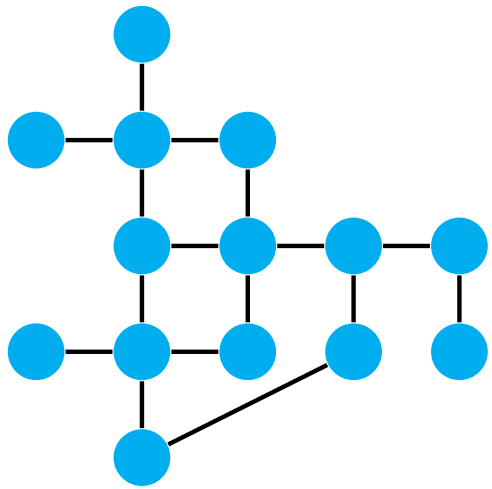
連結非巡回無向グラフ，非巡回無向グラフをそれぞれ特別に，
(自由) 木，**森**という．



(a) 自由木



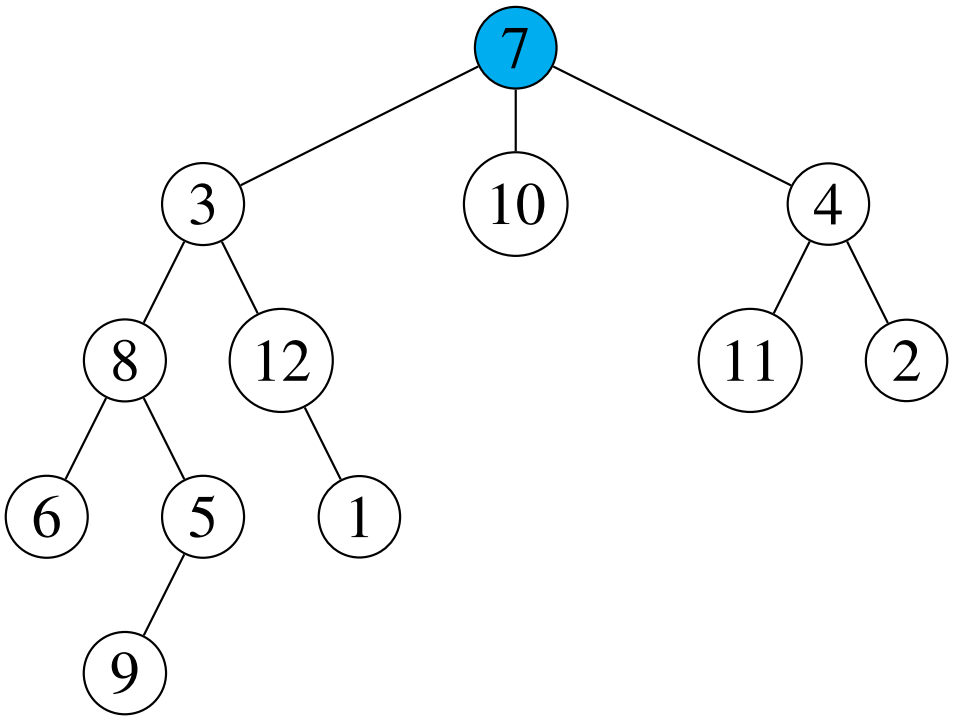
(b) 森



(c) 木でも森でもないグラフ (閉路あり)

根付き木

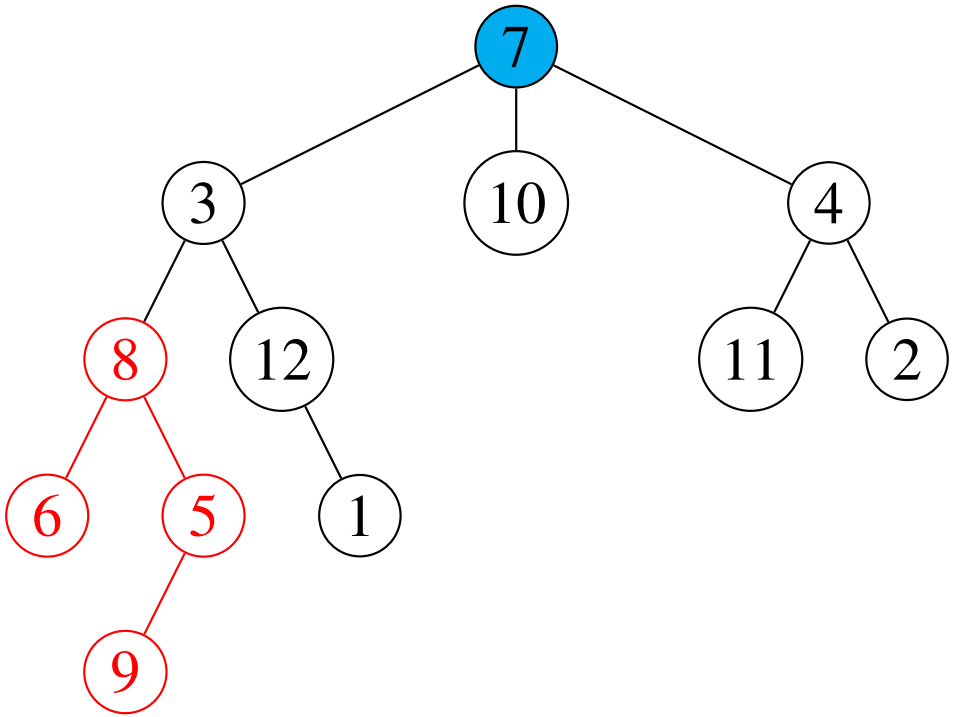
他の頂点と区別された 1 つの頂点を持つ木を**根付き木**といい，特別な頂点を**根**という．根付き木の頂点を**節点**と呼ぶ．



節点 7 を根とする根付き木

部分木

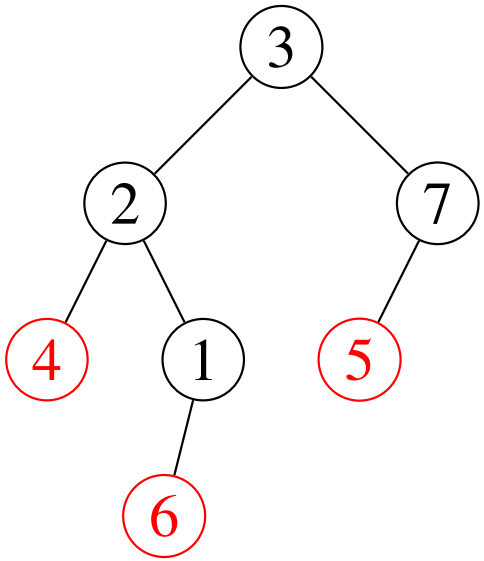
r を根とする根付き木 T の節点 x を考える．r から x へ至る唯一の単純道上の任意の節点 y を x の**祖先**と呼ぶ．y が x の祖先のとき，x を y の子孫と呼ぶ．(どの節点も自分自身の子孫であり祖先である．) y が x の祖先で $x \neq y$ のとき，y は x の**真の祖先**であり，x は y の**真の子孫**であるという．x の子孫によって構成される x を根とする部分木を x を**根とする部分木**と呼ぶ．



節点 7 を根とする根付き木の，節点 8 を根とする部分木 (赤)

親・子

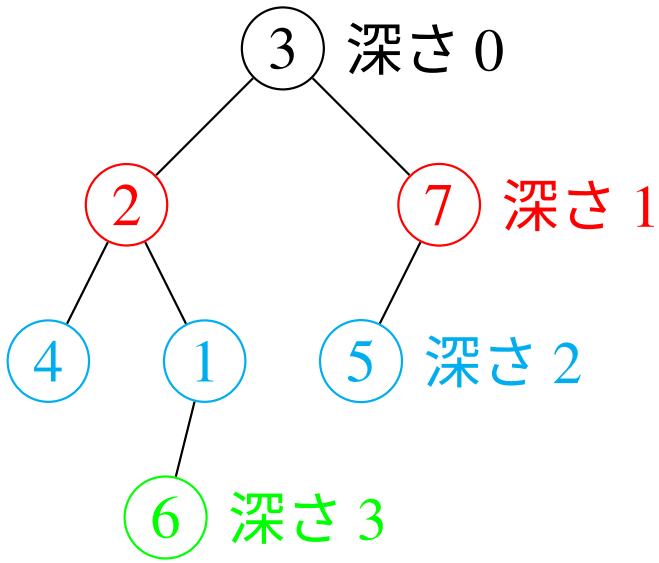
根付き木 T の根 r から節点 x に至る単純道上の最後の辺が (y, x) のとき，y を x の**親**，x を y の**子**と呼ぶ．根は T の中で親を持たない唯一の節点である．2 つの節点が同一の親を持つならば，これらは**兄弟**であるという．子を持たない節点を**葉**あるいは**外部節点**と呼ぶ．葉以外の節点は**内部節点**である．



節点 3 を根とする根付き木．赤い節点が葉である．節点 2 の子は節点 1,4 であり，これらは兄弟である．節点 1 の親は節点 2, 子は節点 6 である．

節点の次数・深さ

根付き木 T の節点 x が持つ子の数が x の**次数**である．自由木の頂点の次数は隣接する頂点数であり，根付き木と異なることに注意する．根 r から節点 x に至る単純道の長さを T における x の**深さ**と呼ぶ．木の**レベル**は同じ深さを持つすべての節点から構成される．木におけるある節点の**高さ**はこの節点からより深い方向にある任意の葉に至る最長の単純道が含む辺の数であり，木の高さはその木の根の高さである．木の高さはその木の任意の節点の深さの最大値でもある．



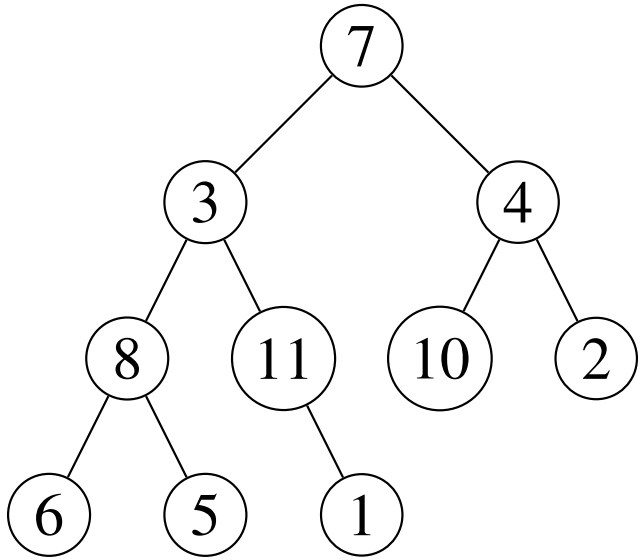
節点 3 を根とする高さ 3 の根付き木．節点 2 の次数は 2，深さは 1，高さは 2．

2 分木

2 分木を再帰的に定義する．節点の有限集合上の **2 分木** T は以下の条件のどちらかを満たす構造である．

- T は節点を含まない．
- T は共通要素を持たない 3 つの節点集合，**根**，**左部分木**と呼ぶ 2 分木，**右部分木**と呼ぶ 2 分木から構成されている．

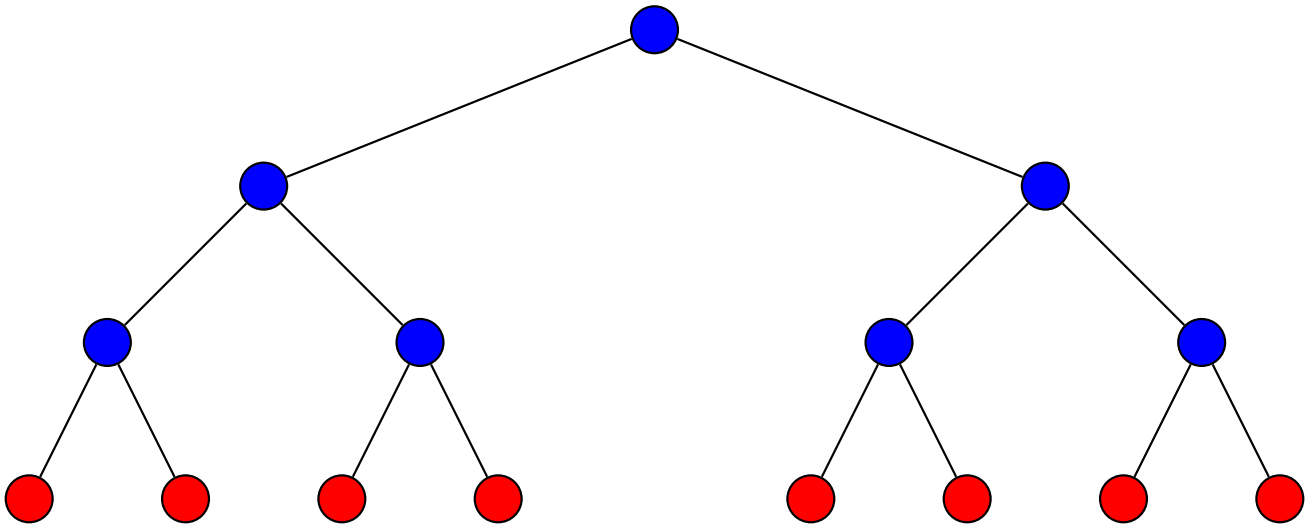
左部分木が空でないとき，その根を**左の子**，同様に，右部分木が空でないとき，その根を**右の子**と呼ぶ．



節点 7 を根とする 2 分木．節点 4 の左の子は節点 10, 右の子は節点 2 である．

完全 k 分木

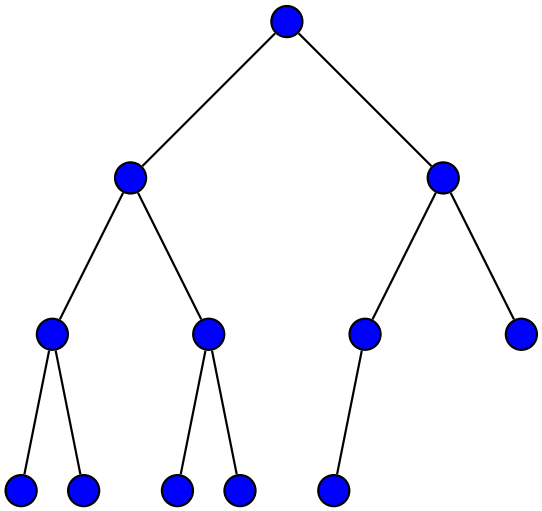
2 分木の定義を節点が子を $k(\geq 2)$ 子持つ木に拡張したものを k 分木とする．すべての葉が同じ深さを持ち，すべての内部節点の次数が k である k 分木を**完全 k 分木**と呼ぶ．



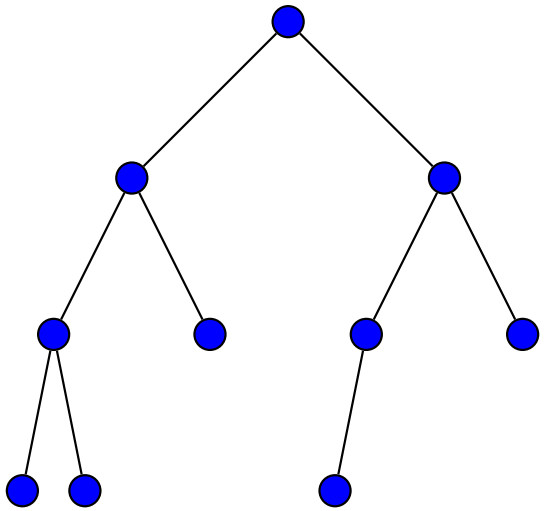
高さ 3 の完全 2 分木．8 個の葉 (赤) と 7 個の内部節点 (青) を持つ．

おおよそ完全 2 分木

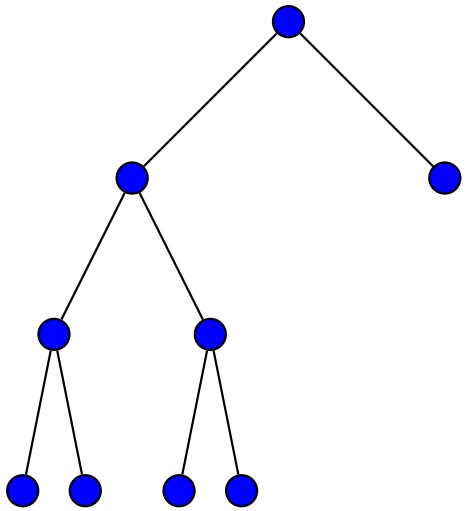
~~最下層の葉が左から順に埋まっており，任意の葉同士の深さの差が 1 以内である 2 分木をおおよそ完全 2 分木と呼ぶ．~~
木の最下位レベル以外の全てのレベルは完全に埋まっており，最下位レベルは左から順にある所まで埋まっている 2 分木をおおよそ完全 2 分木と呼ぶ．



(a) おおよそ完全 2 分木



(b) おおよそ完全 2 分木でない 2 分木 (左から埋まっていない)



(c) おおよそ完全 2 分木でない 2 分木 (葉の深さの差の最大値が 2)

目次

ヒープソートとは？

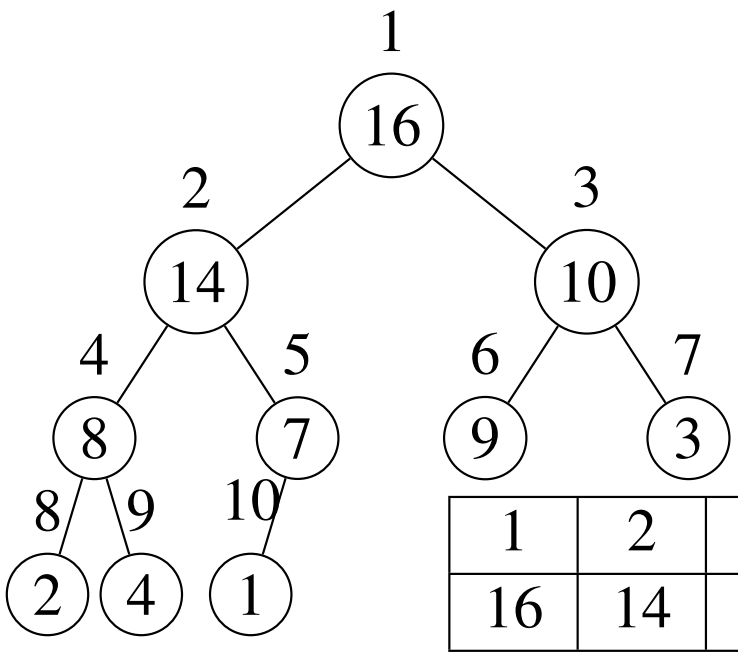
ヒープソートは (2 分木) ヒープというデータ構造を使用するソートである．
ヒープソートはマージソートと同じ最悪実行時間 $\Theta(n \lg n)$ を持ち，挿入ソートの最悪実行時間 $\Theta(n^2)$ より高速である．
また，挿入ソートと同様 (そしてマージソートと異なり)，その場でのソートである．
従って，ヒープソートはこれまでに示した 2 つのアルゴリズムの長所を兼ね備えている．
第 6 章ではこれらを確認する．

ヒープソート

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

ヒープ

(**2 分木**) **ヒープ**データ構造は，次の図のように，おおよそ完全 2 分木とみなすことができる配列オブジェクトである．木の各節点は配列のある要素に対応している．木の最下位レベル以外のすべてのレベルは完全に埋まっており，最下位レベルは左から順にある所まで埋まっている．



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

(a) ある max ヒープの 2 分木による表現

(b) (a) を配列で管理する．木の各節点の中の数はその頂点に格納されている値である．節点の上の数は対応する配列の添字である．親は常にその子の左側に位置する．

ヒープ

ヒープを表現する配列は 2 つの属性を持つオブジェクトである．属性 $A.length$ は (通常通り) 配列が含む要素数を表し，属性 $A.heap\text{-}size$ は配列 A に格納されているヒープの要素数を表す．すなわち，$A[1..A.length]$ は数を含むかもしれないが，$A[1..A.heap\text{-}size]$ の要素だけが正しいヒープの要素である．ここで，$0 \leq A.heap\text{-}size \leq A.length$ である．
$A.length$ は配列の初期化の際に決定する定数である (e.g. C 言語の $int\ \ int\_array[10000]$ の 10000) が，$A.heap\text{-}size$ は配列 A のうち正しくヒープデータ構造の要素である要素の数であり，動作の中で変動する変数である．

ヒープ

木の根は $A[1]$ であり，節点の添字 i から，以下のように，親，左の子，右の子を示す添字を簡単に計算できる．

Algorithm PARENT(i)

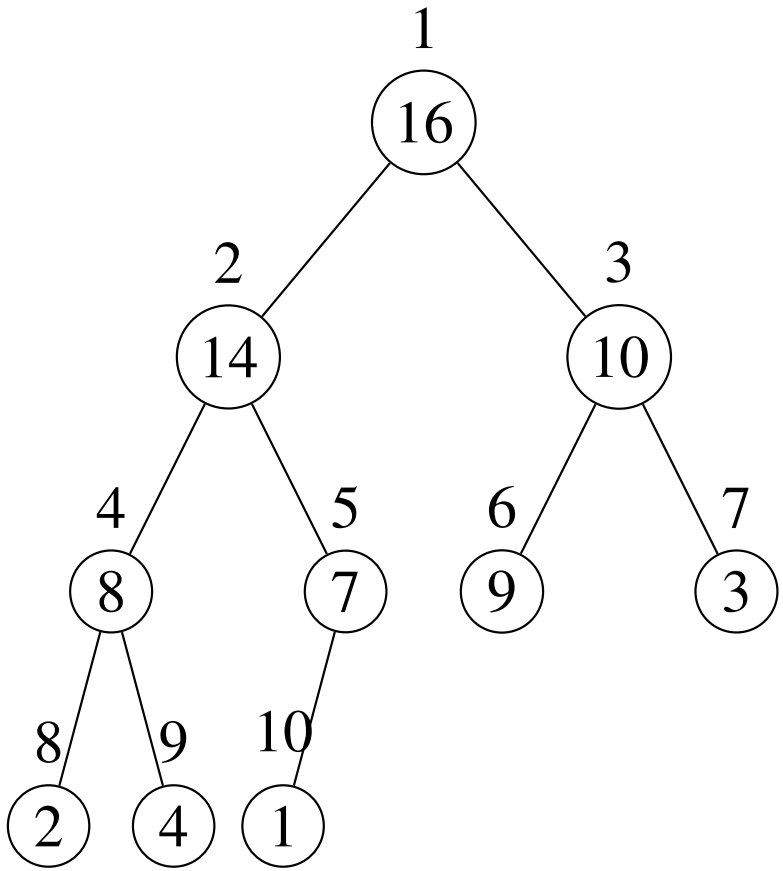
```
1 return ⌊i/2⌋
```

Algorithm LEFT(i)

```
1 return 2i
```

Algorithm RIGHT(i)

```
1 return 2i + 1
```



ヒープ

たいていの計算機では，LEFT 手続きは，i の 2 進表現に対して 1 ビット左シフト命令を実行することで $2i$ を計算する．

$$0101_{(2)}(=5) \xrightarrow[<<]{\text{1 ビット左シフト}} 01010_{(2)}(=10)$$

同様に，RIGHT 手続きは，i の 2 進表現を 1 ビット左にシフトした後，最下位ビットに 1 を立てることで $2i+1$ を高速に計算する．

$$0101_{(2)}(=5) \xrightarrow[<<]{\text{1 ビット左シフト}} 01010_{(2)}(=10) \xrightarrow[|\,1]{\text{最下位ビットを 1 に}} 01011_{(2)}(=11)$$

PARENT 手続きはを i の 2 進表示を 1 ビット右シフトすることで $\lfloor i/2 \rfloor$ 計算する．

$$0101_{(2)}(=5) \xrightarrow[>>]{\text{1 ビット右シフト}} 010_{(2)}(=2)$$

max ヒープと min ヒープ

2 分木ヒープには max ヒープと min ヒープの 2 種類ある．どちらも節点の値が**ヒープ条件**を満たすが，条件はヒープによって異なる．

- max ヒープは次の max ヒープ条件を満たす．

max ヒープ条件

根以外の任意の節点 i について $A[\mathrm{PARENT}(i)] \geq A[i]$．

したがって，max ヒープは最大の要素を根に格納し，ある節点を根とする部分木が含む値はその節点自身の値を超えない．

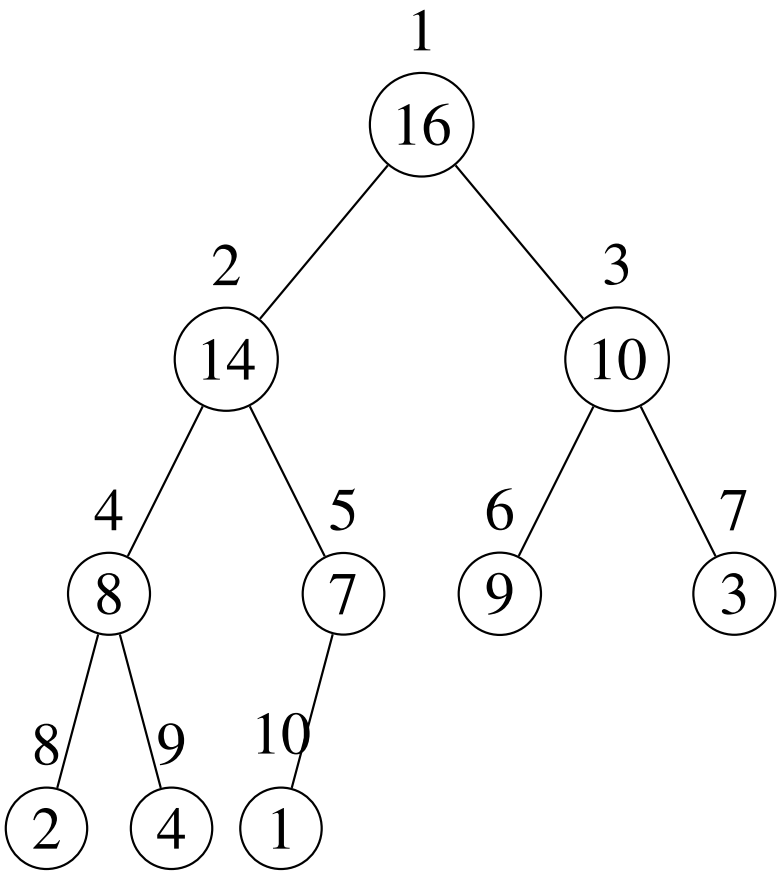
- min ヒープは次の min ヒープ条件を満たす．

min ヒープ条件

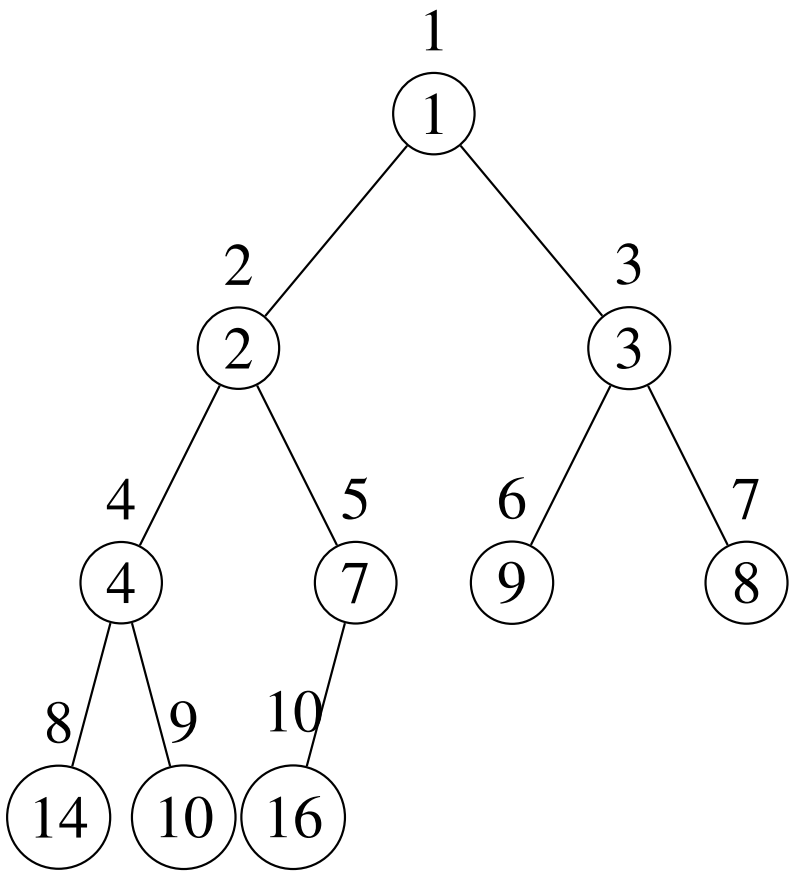
根以外の任意の節点 i について $A[\mathrm{PARENT}(i)] \leq A[i]$．

したがって，min ヒープは最小の要素を根に格納し，ある節点を根とする部分木が含む値はその節点以上の値となる．

max ヒープと min ヒープ



(a) max ヒープ
根以外の任意の節点 i について
$A[\mathrm{PARENT}(i)] \geq A[i]$ を満たす.



(b) min ヒープ
根以外の任意の節点 i について
$A[\mathrm{PARENT}(i)] \leq A[i]$ を満たす.

max ヒープと min ヒープ

ヒープソートアルゴリズムでは max ヒープを用いる．
優先度付きキューは普通 min ヒープを用いて実現する．これは，第 6.5 節で議論する．
今後，個々の応用に対して max ヒープと min ヒープのどちらが必要か明示する．どちらも適用できる場合には単に「ヒープ」と書く．

ヒープの高さ

ヒープを根付き木とみなしたとき，ヒープにおける節点の**高さ**をその節点からある葉に下る最長の単純道に含まれる辺数と定義し，ヒープの高さをその根の高さと定義する．
n 個の要素を含むヒープの高さ h について $h = \lfloor \lg n \rfloor$ が成立する．

Proof.

ヒープはおおよそ完全 2 分木で，任意の葉同士の深さが 1 以内であるので，高さ h のヒープは，最下位レベル以外の全てのレベルは完全に埋まっている．ここで，深さ $k(0 \leq k \leq h-1)$ には 2^k 個の節点があるので，最下位レベル以外には $\sum_{k=0}^{h-1} 2^k = 2^h - 1$ 個の節点が存在する．
また，最下位レベルの葉の数 m は $1 \leq m < 2^h + 1$ を満たすので，高さ h のヒープに含まれる要素数 n $(= 2^h - 1 + m)$ は，

$$
\begin{aligned}
2^h - 1 + 1 \leq n < 2^h - 1 + 2^h + 1 &\iff 2^h \leq n < 2^{h+1} \\
&\iff h \leq \lg n < h + 1 \\
&\iff \lg n - 1 < h \leq \lg n \iff h = \lfloor \lg n \rfloor
\end{aligned}
$$

□

ヒープの基本演算

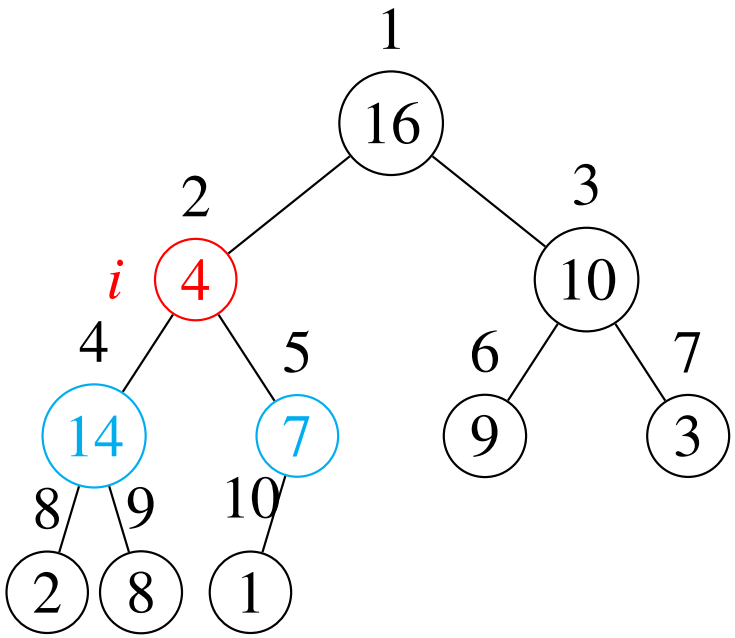
次ページから，ヒープの基本演算の手続きについて説明する．

6.2 ヒープ条件の維持

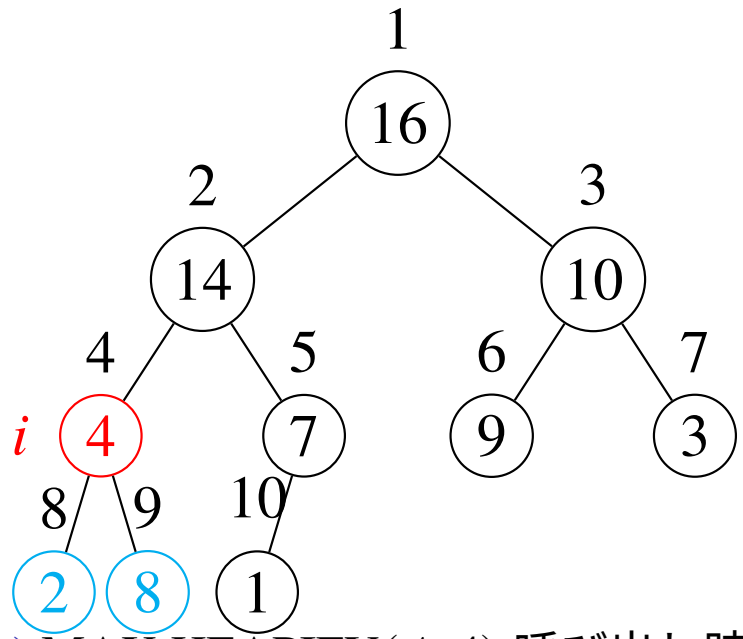
max ヒープ条件を維持するための手続きが MAX-HEAPIFY である．
入力は配列 A と配列の添字 i である．
MAX-HEAPIFY が呼び出されるとき，LEFT(i) と RIGHT(i) を根とする 2 分木は共に max ヒープ条件を満たしていると仮定する．しかし，そのとき，$A[i]$ はその子より小さく，max ヒープ条件に違反している可能性がある．MAX-HEAPIFY は $A[i]$ の値を max ヒープの中に"滑り落とす"ようにして，添字 i を根とする部分木が max ヒープ条件を満たすようにする．

max ヒープ条件 (再掲)

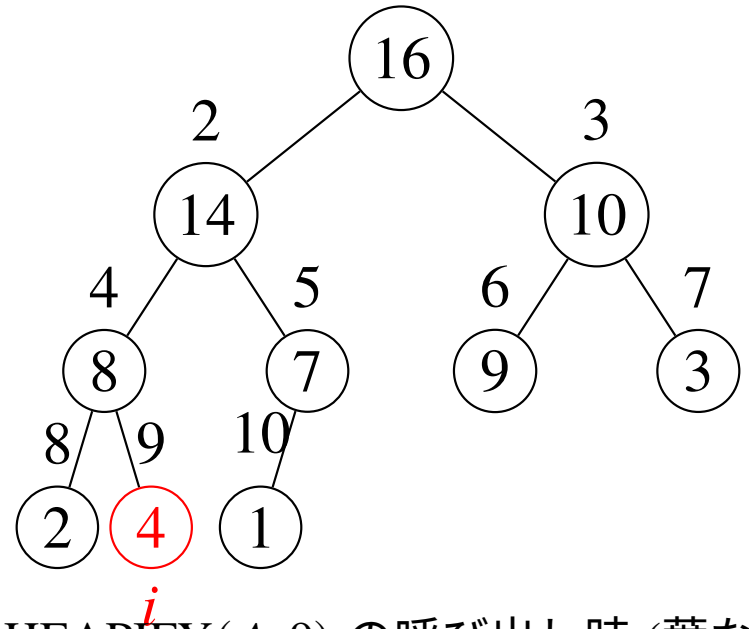
根以外の任意の節点 i について $A[\mathrm{PARENT}(i)] \geq A[i]$．



(a) 呼び出し時 (初期状態)



1 (b) MAX-HEAPIFY$(A, 4)$ 呼び出し時



(c) MAX-HEAPIFY$(A, 9)$ の呼び出し時 (葉なので終了)

MAX-HEAPIFY の擬似コード

Algorithm MAX-HEAPIFY(A, i)

```
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  else largest = i
9  if largest ≠ i
10     swap(A[i], A[largest])
11     MAX-HEAPIFY(A, largest)
```

MAX-HEAPIFY の動作

1~8 行目:$largest$ を次のように決定する．

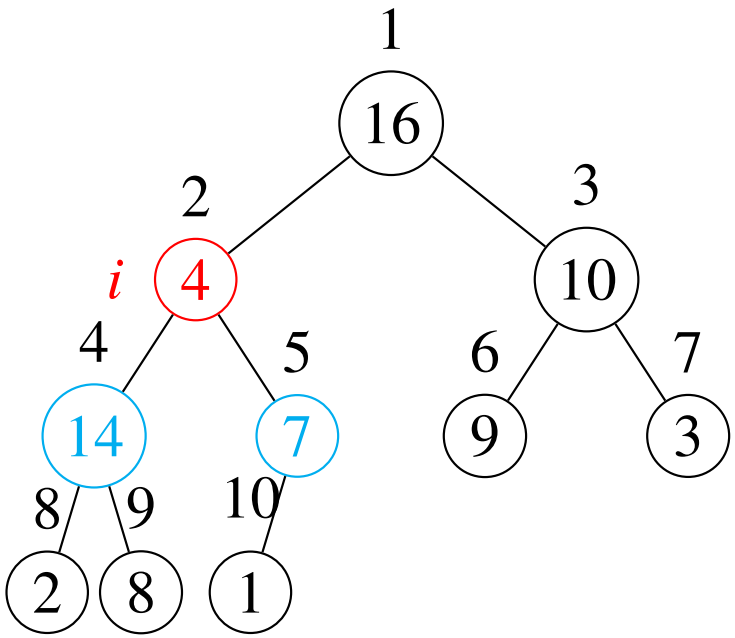
$$largest = \underset{j \in \{i, \mathrm{LEFT}[i], \mathrm{RIGHT}[i]\}}{\arg\max} A[j]$$

9 行目:今見ている $A[i]$ が最大ならヒープ条件を満たしているので，なにもしない．

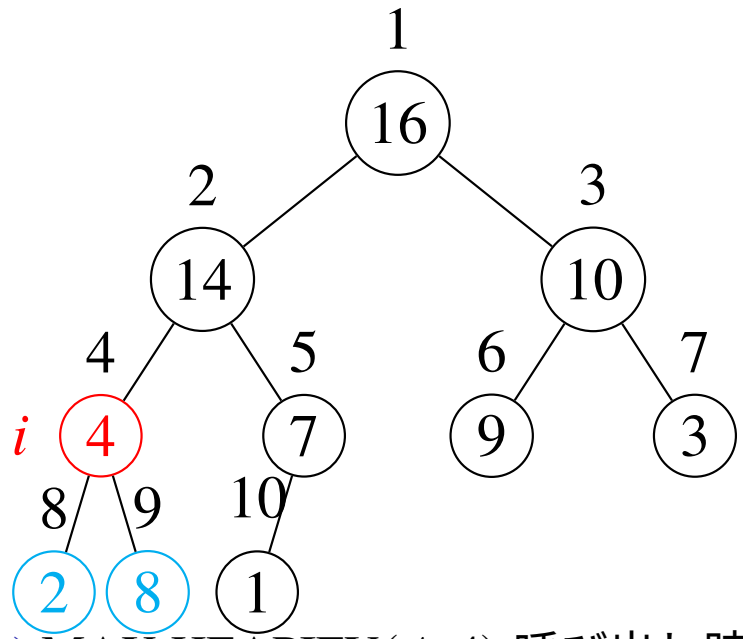
10 行目:$A[i]$ と $A[largest]$ を入れ替えることで，節点 i の max ヒープ条件を満たす．

11 行目:$A[largest]$ を根とする部分木は max ヒープ条件に違反する可能性があるので，MAX-HEAPIFY を再帰的に呼び出す．

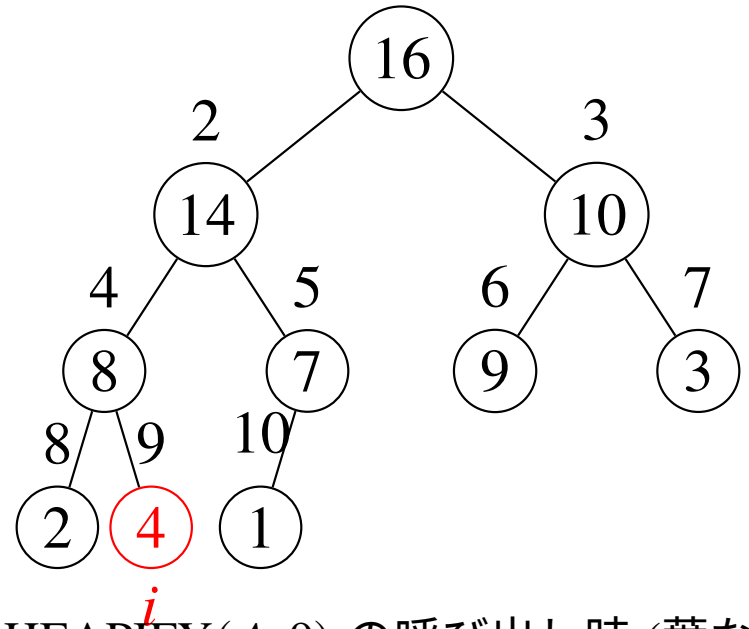
3,5 行目の **and** の前:葉の子はいない．
次のスライドは MAX-HEAPIFY(A,2) の動作である．ここで，
$A.heap\text{-}size = 10$ である．



(a) 呼び出し時 (初期状態)



1 (b) MAX-HEAPIFY$(A, 4)$ 呼び出し時



(c) MAX-HEAPIFY$(A, 9)$ の呼び出し時 (葉なので終了)

MAX-HEAPIFY の実行時間

与えられた節点 i を根とするサイズ n の部分木上の MAX-HEAPIFY の実行時間は，要素 $A[i], A[\mathrm{LEFT}(i)], A[\mathrm{RIGHT}(i)]$ の節点を入れ替え，max ヒープ条件を満たすために必要な $\Theta(1)$ 時間と，(再帰が起こると仮定したとき) 節点 i の左右どちらかの子を根とする部分木上での MAX-HEAPIFY の実行時間の和である．どちらの子の部分木のサイズも $2n/3$ 以下 (後に示す) だから，MAX-HEAPIFY の実行時間は漸化式

$$T(n) \leq T(2n/3) + \Theta(1)$$

で表現できる．マスター定理から，この漸化式の解は $T(n) = O(\lg n)$ である．

MAX-HEAPIFY の実行時間

MAX-HEAPIFY を再帰的に呼び出すとき，呼び出し時の部分木のサイズを n とすると，左右どちらの子を根とする部分木のサイズも $2n/3$ 以下であることを示す．

Proof.

節点 i を根とする部分木について，最下位のレベルがちょうど半分だけ埋まっているときが，明らかに左右どちらかの子を根とする部分木のサイズが最大になるときである．ここで，次々ページの図のように左の子を根とする部分木のみ最下位のレベルが全て埋まっているとする．
このとき，右の子を根とする部分木の深さ k には 2^k 個の節点があるので，右の子を根とする部分木の高さを $h-1$ とすると，右の子を根とする部分木は 2^h-1 個の節点を含む．ここで，$m=2^h-1$ とする．
左の子を根とする部分木は右のそれより $2^h(=m+1)$ 個節点が多いので，左の子を根とする部分木は $2m+1$ 個の節点を含む．
よって，節点 i を根とする部分木のサイズ n は節点とその子を根とする部分木の節点の数の和から，$n=1+m+(2m+1)=3m+2$ となる．

MAX-HEAPIFY の実行時間

proof (続き).

すなわち，左の子を根とする部分木のサイズと節点 i を根とする部分木のサイズの比は，

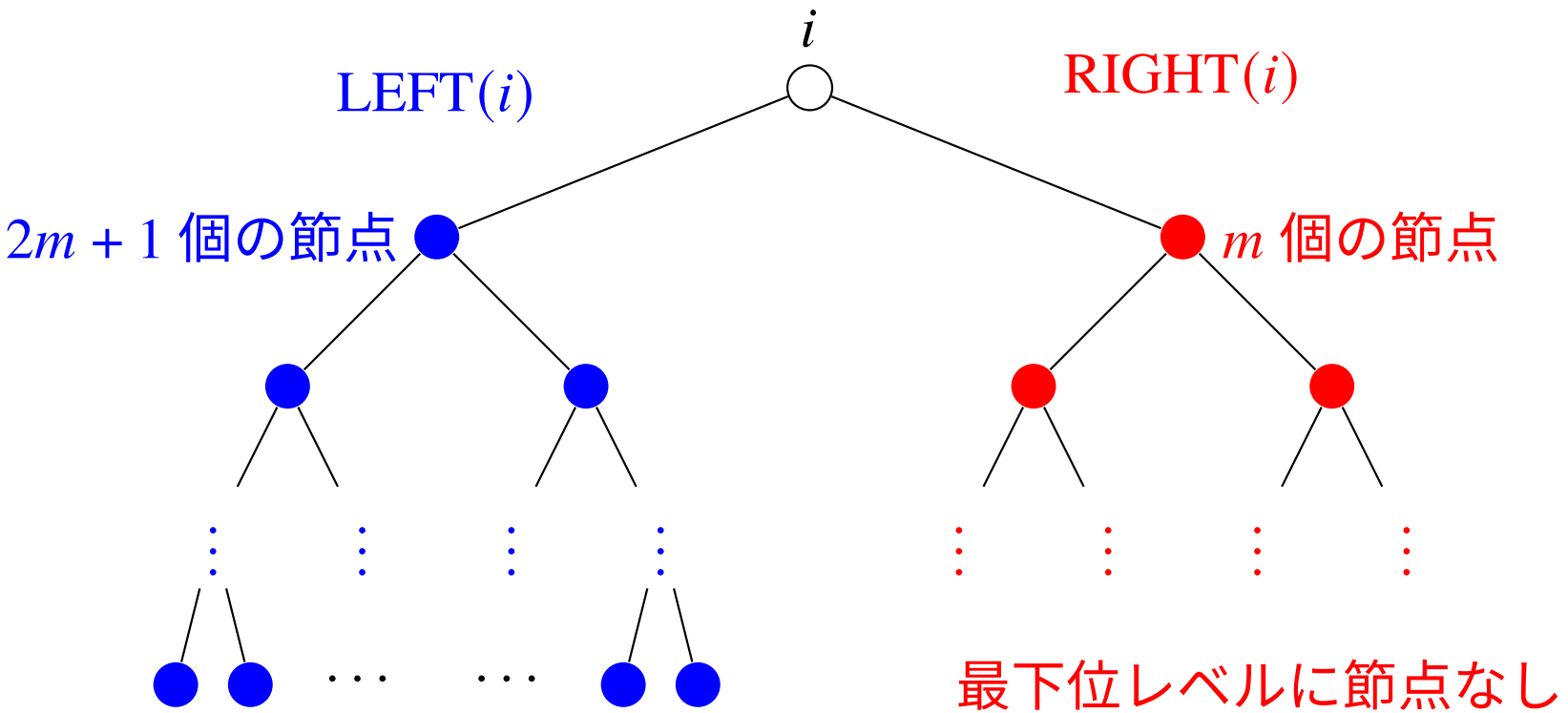
$$\frac{2m+1}{n} = \frac{2m+1}{3m+2} \leq \frac{2}{3}$$

となる．

□

MAX-HEAPIFY の実行時間 (最悪ケース)

左の子を根とする部分木　　右の子を根とする部分木



MAX-HEAPIFY の実行時間について最悪の場合

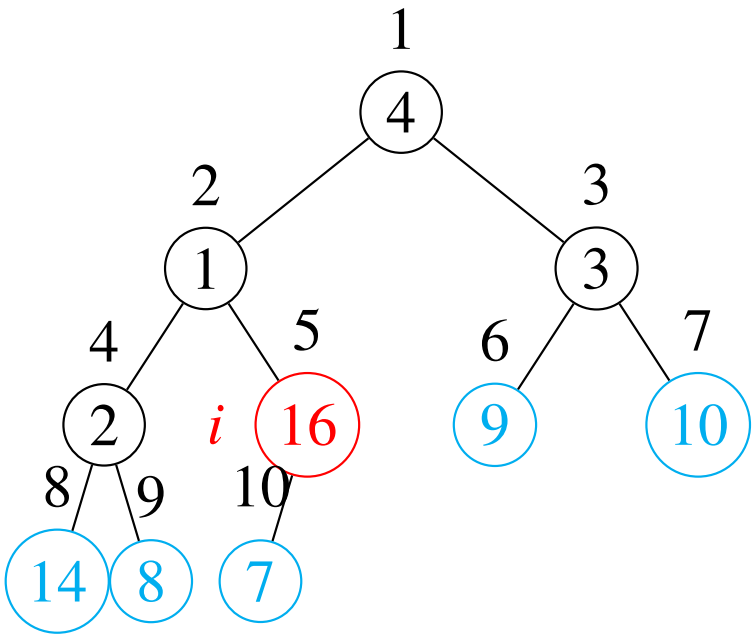
6.3 ヒープの構築

$n = A.length$ とする．ボトムアップ的に手続き MAX-HEAPIFY を呼び出すことで配列 $A[1..n]$ を max ヒープに変換できる．
ヒープを構築するための BUILD-MAX-HEAP の擬似コードを示す．

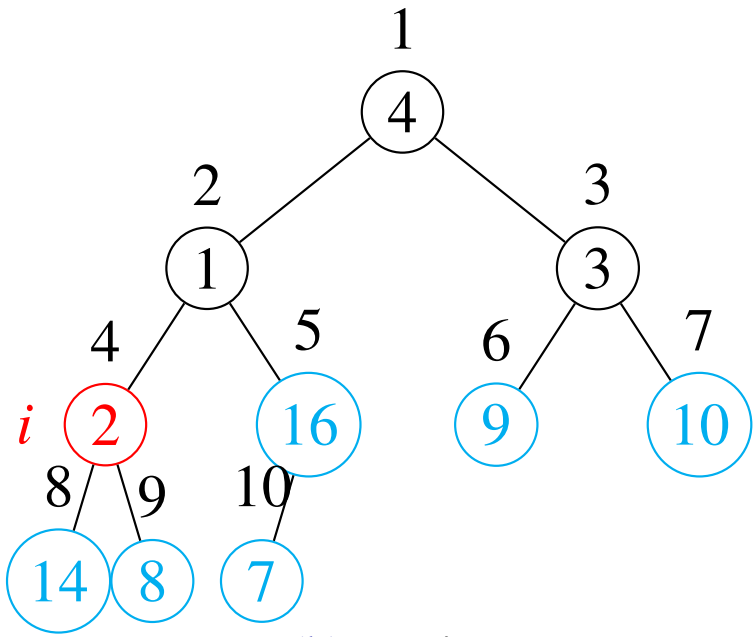
Algorithm BUILD-MAX-HEAP(A)

```
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

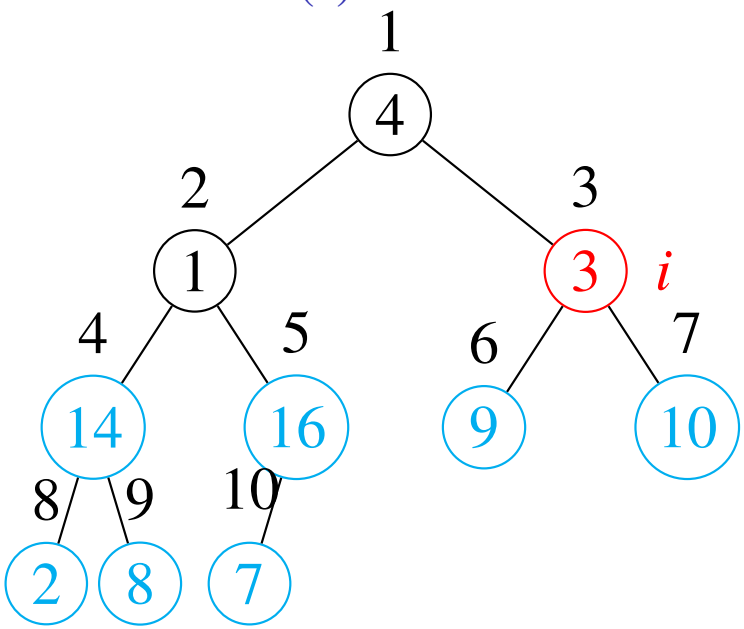
次ページに BUILD-MAX-HEAP の動作例を示す．
赤い節点が今注目している節点であり，水色の節点が max ヒープ条件を満たしている節点である．



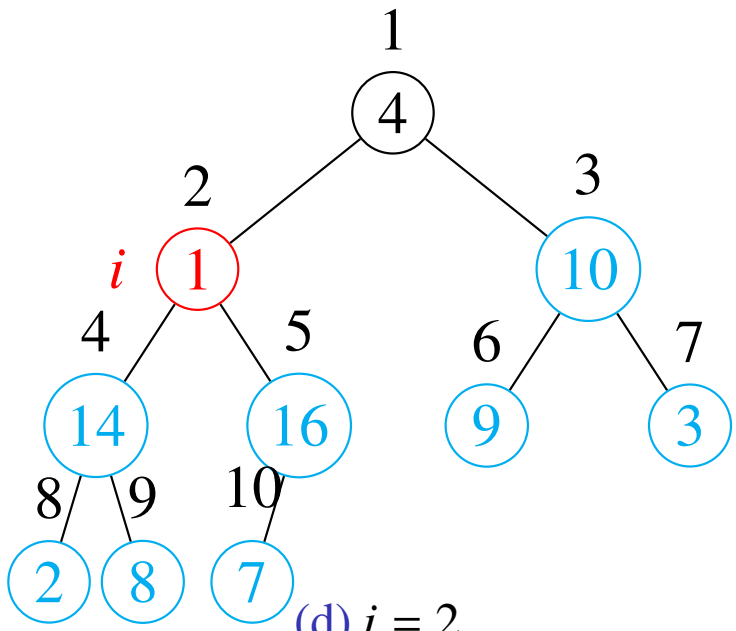
(a) $i = 5$



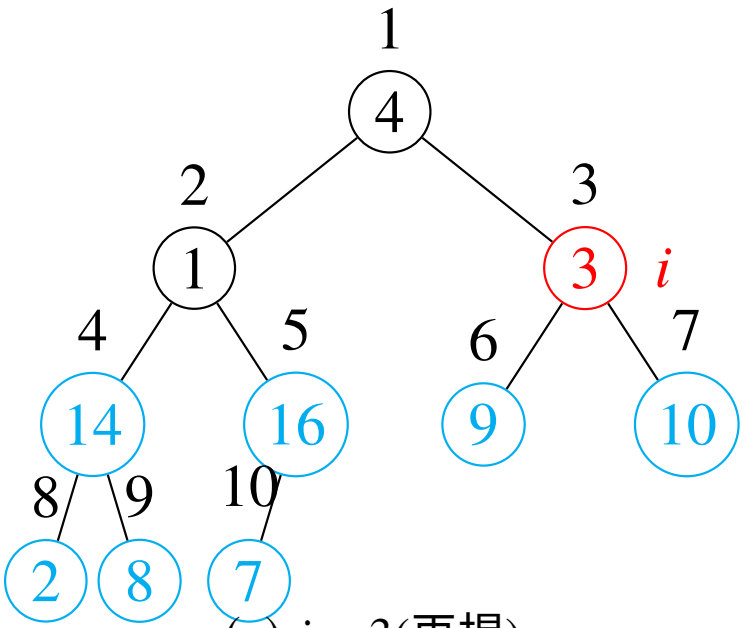
(b) $i = 4$



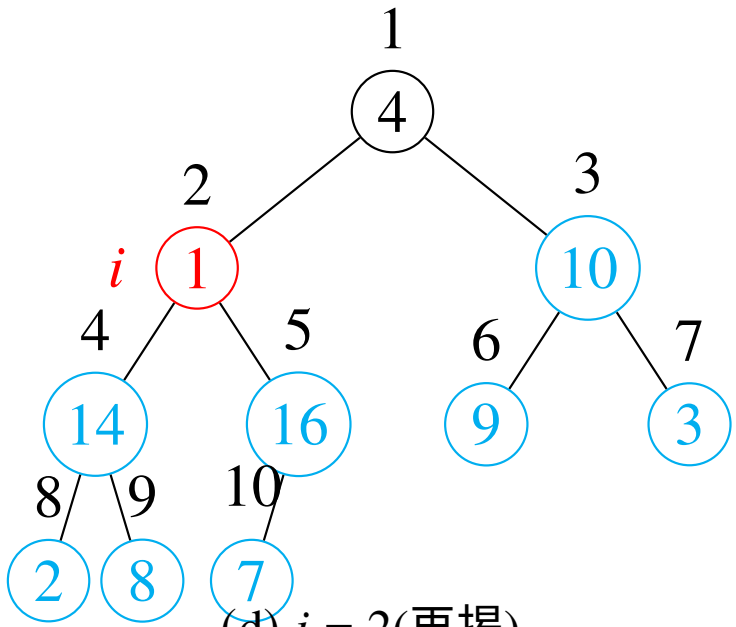
(c) $i = 3$



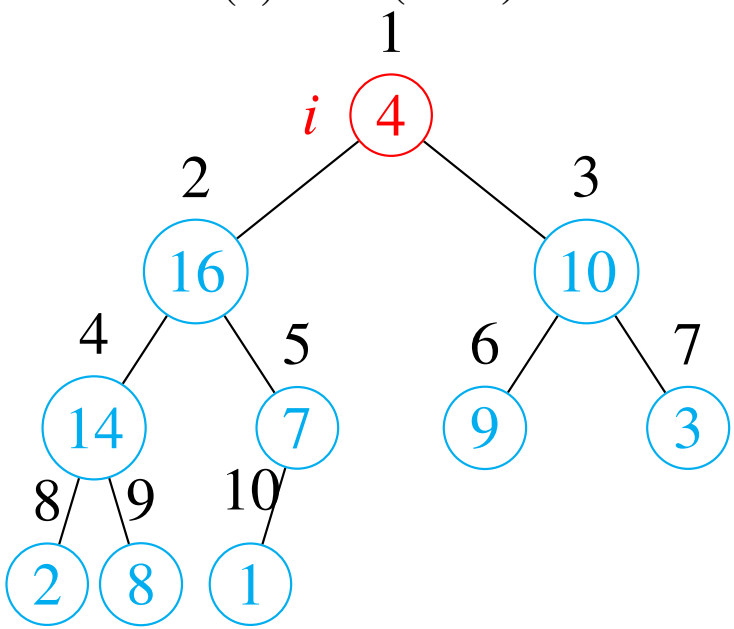
(d) $i = 2$



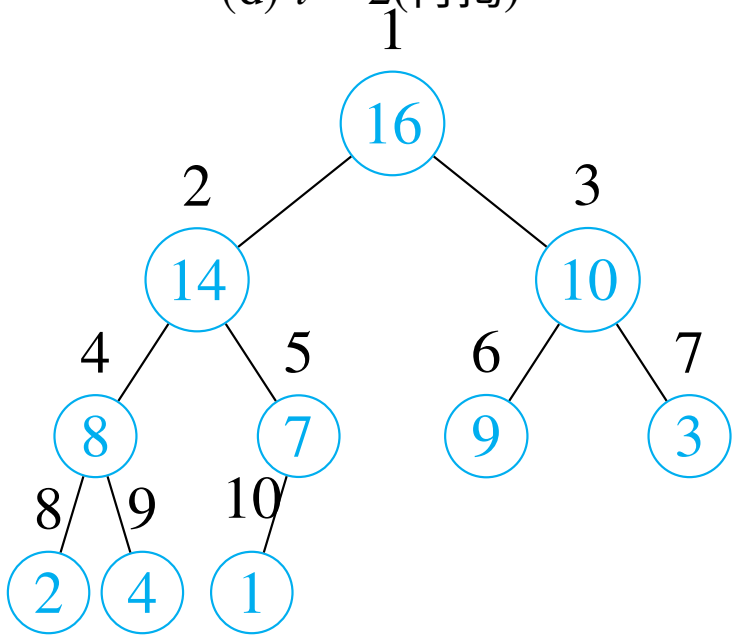
(c) $i = 3$(再掲)



(d) $i = 2$(再掲)



(e) $i = 1$



(f) 終了時点の max ヒープ

BUILD-MAX-HEAP の計算時間

BUILD-MAX-HEAP の実行時間の上界を次のように計算できる．
MAX-HEAPIFY の各呼び出しに $O(\lg n)$ 時間かかり，呼び出しは $O(n)$ 回起こる．したがって，BUILD-MAX-HEAP の動作例を示す実行時間は高々 $O(n \lg n)$ である．
しかし，この上界は誤りではないが，漸近的にタイトではない．

BUILD-MAX-HEAP の計算時間

ある節点における MAX-HEAPIFY の実行時間はその節点の高さに依存し，ほとんどの節点の高さが低いことに注意すると，よりタイトな上界が導出できる．

n 個の要素を持つヒープの高さは $\lfloor \lg n \rfloor$ であり，高さ h の節点は高々 $\lceil n/2^{h+1} \rceil$ 個しかないこと (後に示す) を用いる．

MAX-HEAPIFY が高さ h の節点で呼ばれたときに要する時間は $O(h)$ だから，BUILD-MAX-HEAP の総コストは上から，

$$\begin{aligned}\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) &= O\left(n \sum_{h=0}^{\lceil \lg n \rceil} \frac{h}{2^h} \right) \left(\because \left\lceil \frac{n}{2^{h+1}} \right\rceil < \frac{n}{2^h} \right) \\ &= O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) \\ &= O(2n) = O(n)\end{aligned}$$

で抑えられる．従って，未ソート配列から max ヒープを線形時間で構築できる．

BUILD-MAX-HEAP の計算時間

節点数 n の任意のヒープには，高さ h の節点は高々 $\lceil n/2^{h+1} \rceil$ 個しかないことを示す．

Proof.

まず，$h = 0$ の節点 (すなわち葉) の数について考える．節点 i が葉であるということは，子がいないということと同値であり，それは $\mathrm{LEFT}(i)$ が n より大であることと同値であるので，$2i > n$ を満たす節点が葉である．
すなわち，添字 $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \cdots, n$ の $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$ 個の節点が葉 (高さ $h = 0$ の節点) である．
それらの葉を除いた $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ の節点からなるヒープについて考える．$h = 0$ のときと同じように考えると，$\lfloor n/2 \rfloor - \lfloor \lfloor n/2 \rfloor /2 \rfloor = \lceil n/2^2 \rceil$ 個の節点が新たなヒープの葉 (高さ $h = 1$ の節点) である．
あとは，帰納的に高さ h の節点は高々 $\lceil n/2^{h+1} \rceil$ 個であることが示される．

□

HEAPSORT の擬似コード

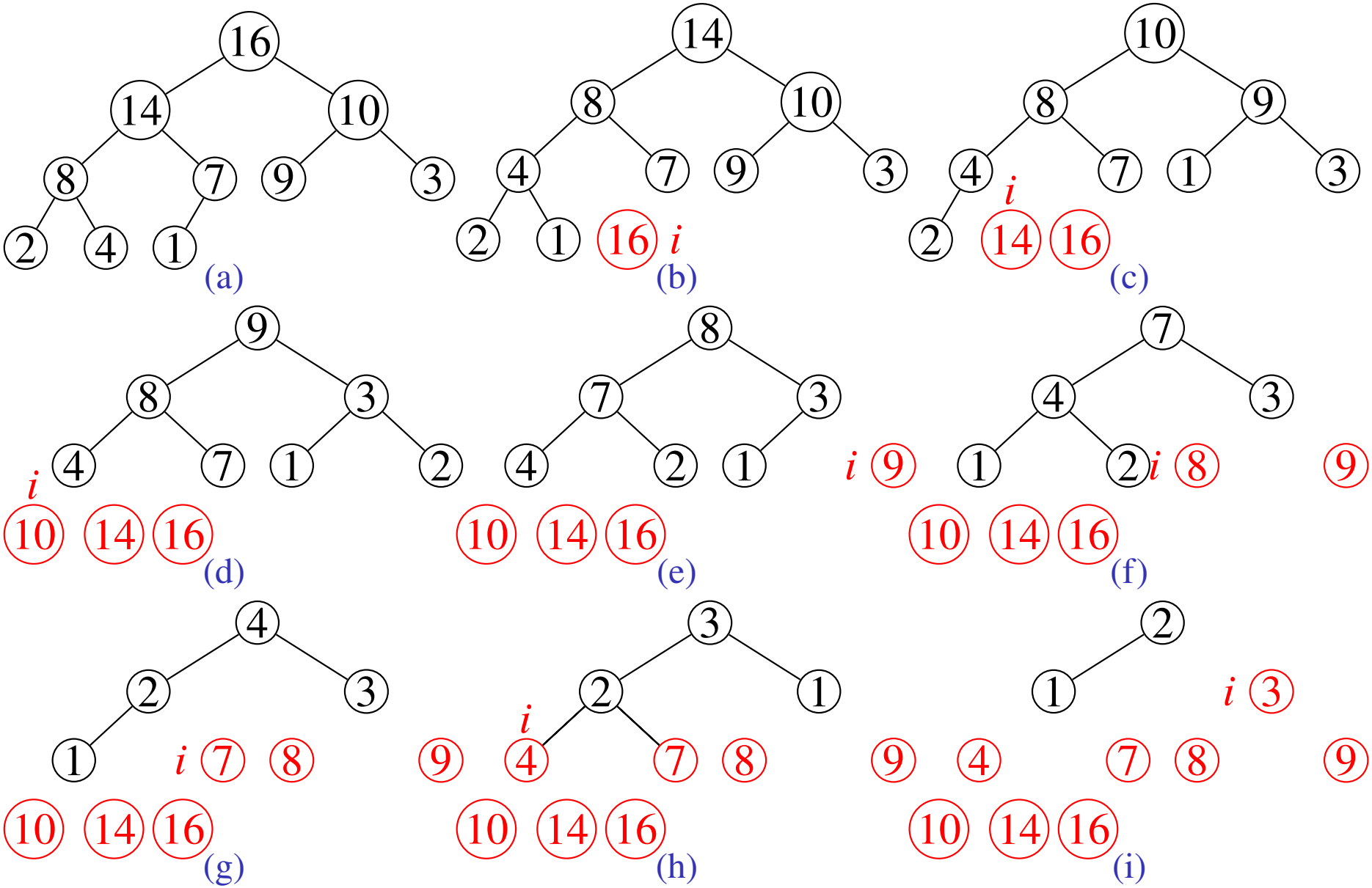
$n = A.length$ とする．ヒープソートアルゴリズムの擬似コードは以下のようになる．

Algorithm HEAPSORT(A)

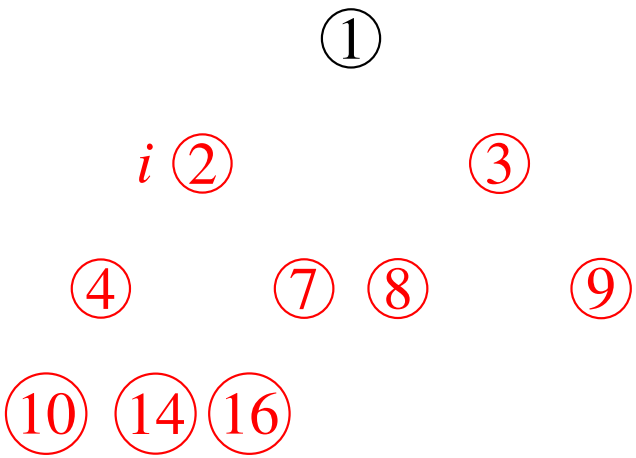
```
1 BUILD-MAX-HEAP(A)
2 for i = A.length downto 2
3     swap(A[1], A[i])
4     A.heap-size = A.heap-size − 1
5     MAX-HEAPIFY(A, 1)
```

1 行目で max ヒープを構築する．2 行目以降で，$A[1]$ に格納されている現在の max ヒープの最大値と，$A[i]$ の値を交換する．この際，$heap\text{-}size$ を一つ小さくする．この際，新しく $A[1]$ に格納された値は max ヒープ条件に違反する可能性があるので，MAX-HEAPIFY を呼び出し，max ヒープを構築する．

HEAPSORT の動作



HEAPSORT の動作



(j)

1	2	3	4	7	8	9	10	14	16

(k) ソート済み配列 A

HEAPSORT の計算時間

BUILD-MAX-HEAP の呼び出しに $O(n)$ かかり，1 回の呼び出しに $O(\lg n)$ 時間かかる MAX-HEAPIFY が $n-1$ 回呼び出されるので，HEAPSORT の最悪実行時間は $O(n \lg n)$ である．
また，外部メモリは高々定数個しか必要としないので，内部ソートである．
ここで，ヒープソートは安定ソートではない．
反例：$\langle 1, 2, 2', 3 \rangle \xrightarrow{HEAPSORT} \langle 1, 2', 2, 3 \rangle$

ヒープソートまとめ

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

優先度付きキュー

優先度付きキューはヒープデータ構造を用いた**キー**と呼ぶ値を持つ要素の集合 S を管理するためのデータ構造である．
max 優先度付きキューでは次の操作ができる．

- INSERT(S, x) は集合 S に要素 x を挿入する．この操作は演算 $S = S \cup \{x\}$ と等価である．
- MAXIMUM(S) は最大のキーを持つ S の要素を返す．
- EXTRA-MAX(S) は S から最大のキーを持つ要素を削除し，その要素を返す．
- INCREASE-KEY(S, x, k) は要素 x のキーの値を新しいキー値 k に変更する．ただし，k は x の現在のキーの値以上であると仮定する．

同様に **min 優先度付きキュー**も定義されるが，本スライドでは扱わない．

次のスライドから，サイズ n のヒープの集合上の max 優先度付きキューのこれらの操作の計算時間について考える．

max 優先度付きキューの操作

HEAP-MAXIMUM を用いることで，MAXIMUM を $O(1)$ 時間で実現できる．

Algorithm HEAP-MAXIMUM(A)

```
1 return A[i]
```

HEAP-EXTRACT-MAX を用いることで，EXTRACT-MAX を $O(\lg n)$ 時間で実現できる．

Algorithm HEAP-EXTRACT-MAX(A)

```
1 if A.heap-size < 1
2       error "heap underflow"
3 max = A[1]
4 A[1] = A[A.heap-size]
5 A.heap-size = A.heap-size − 1
6 MAX-HEAPIFY(A,1)
7 return max
```

max 優先度付きキューの操作

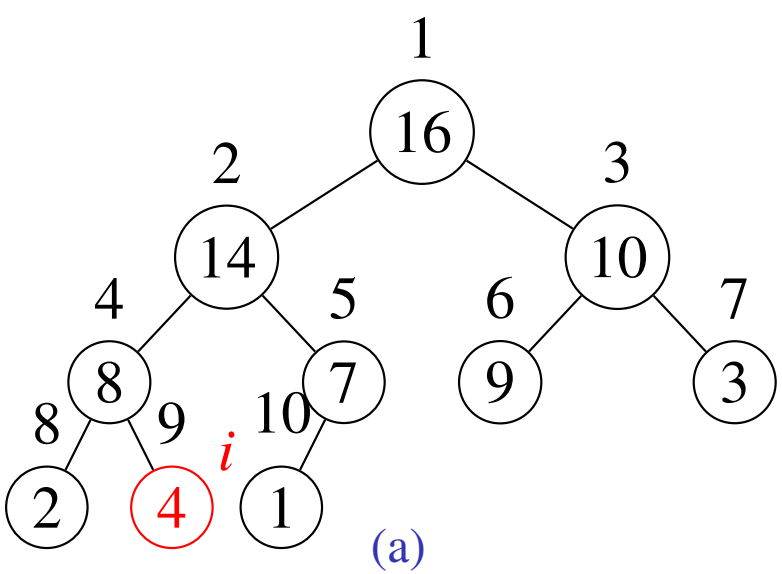
HEAP-INCREASE-KEY を用いることで，INCREASE-KEY を $O(\lg n)$ 時間で実現する．

Algorithm HEAP-INCREASE-KEY(A, i, key)

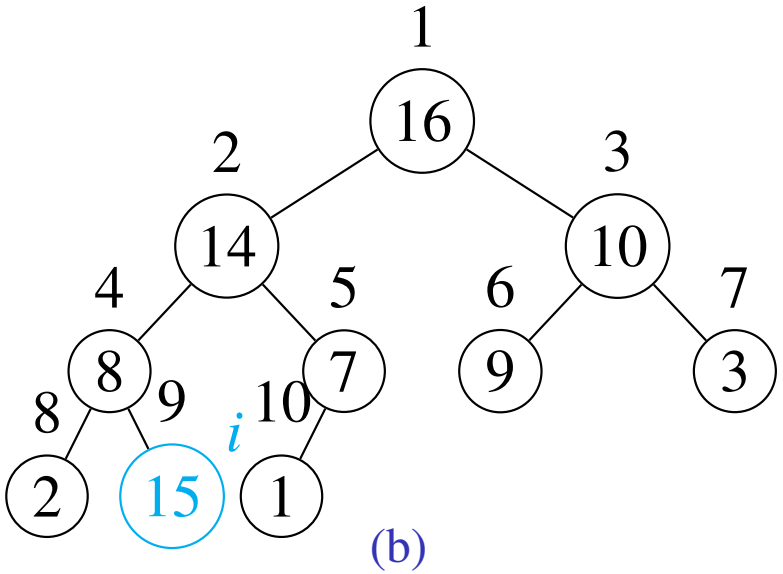
```
1 if key < A[i]
2       error "new key is smaller than current key"
3 A[i] = key
4 while i > 1 and A[PARENT(i)] < A[i]
5       swap(A[i], A[PARENT(i)])
6       i =PARENT(i)
```

キー値 $A[i]$ が増加すると，max ヒープ条件に違反する可能性が生じるので，更新されたキーを持つ節点から根に至る道を辿り，更新されたキーの正しい位置を見つける．
次のスライドに $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ に対して，
HEAP-INCREASE-KEY$(A, 9, 15)$ の動作を示す．

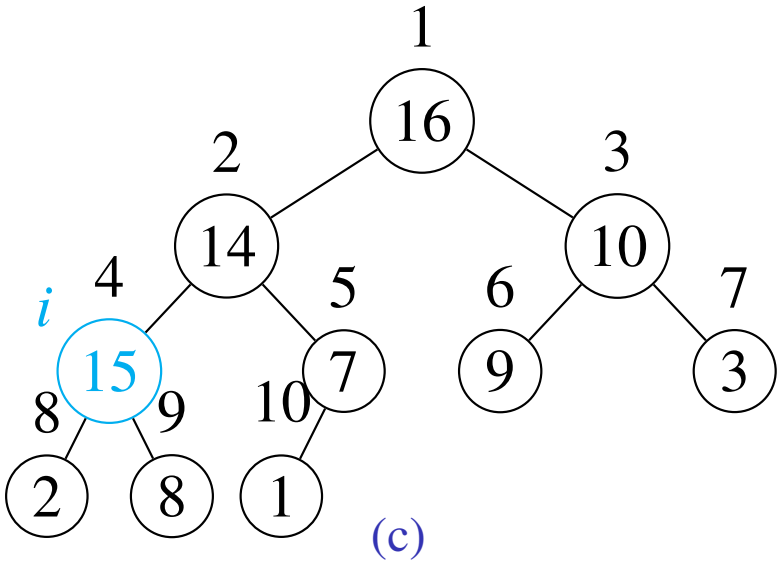
HEAP-INCREASE-KEY の動作



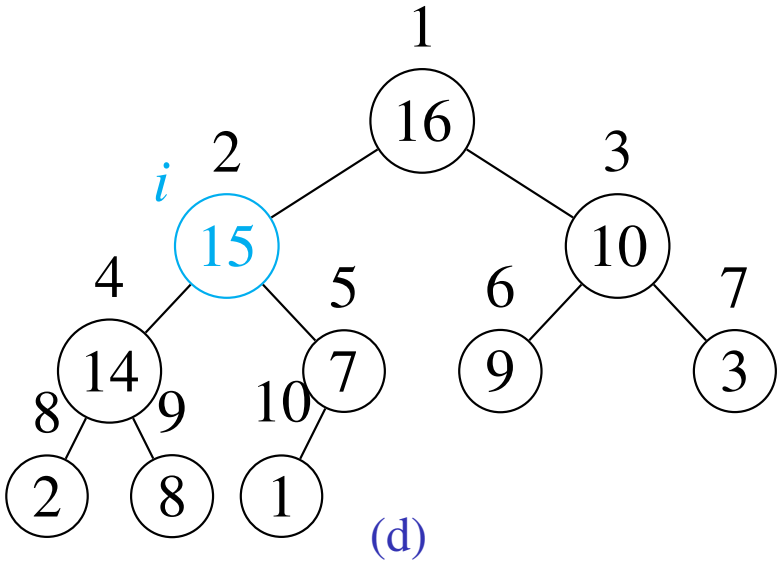
(a)



(b)



(c)



(d)

max 優先度付きキューの操作

手続き MAX-HEAP-INSERT は INSERT 操作を $O(\lg n)$ 時間で実現する．

Algorithm MAX-HEAP-INSERT(A, key)

```
1 A.heap-size = A.heap-size + 1
2 A[A.heap-size] = −∞
3 HEAP-INCREASE-KEY(A, A.heap-size, key)
```

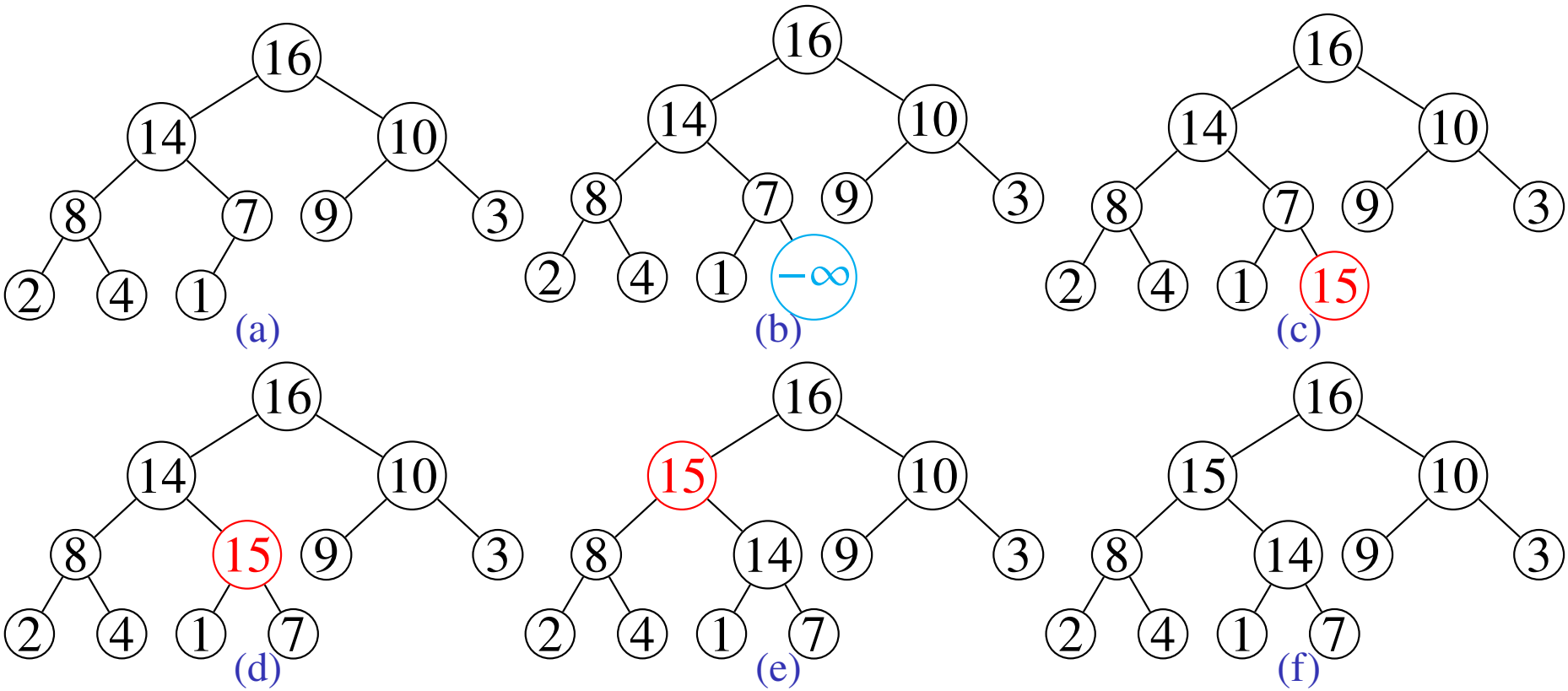
次のスライドに $A = [16, 14, 10, 8, 7, 9, 3, 2, 4, 1]$ に対して，
MAX-HEAP-INSERT$(A, 15)$ の動作を示す．

より小さい値の代入

今，キー値を増加するときのみを考えたが，キー値を減少させたい場合は，その値を代入した後に，その節点で MAX-HEAPIFY を呼び出せばよい．
MAX-HEAPIFY の計算時間も $O(\lg n)$ であるので，キー値を増加させるときも，減少させるときもどちらも $O(\lg n)$ 時間で実現できることが分かる．

(2022.06.02 追記)

MAX-HEAP-INSERT の動作



目次

クイックソート

要素数 n の入力配列上でのクイックソートアルゴリズムの期待実行時間は $\Theta(n \lg n)$ であり，$\Theta(n \lg n)$ に隠されている定数部分が非常に小さい．また，その場でのソートであり，メモリー量を抑えることができる．
クイックソートアルゴリズムは優れた期待実行時間を持っていて，全ての要素が異なるときには，どの特定の入力も最悪の振る舞いを引き起こすことはない．
この章では，クイックソートアルゴリズムの最悪実行時間が $\Theta(n^2)$ であること，そして，全ての要素が異なるという仮定の下で，期待実行時間が $\Theta(n \lg n)$ であることを示す．

7.1 クイックソートの記述

マージソートと同様に，クイックソートは分割統治法に基づいている．部分配列 $A[p..r]$ をソートするための分割統治法の 3 つの段階を以下に示す．

分割： 配列 $A[p..r]$ を 2 つの (空の可能性もある) 部分配列 $A[p..q-1]$ と $A[q..r]$ に，$A[p..q-1]$ のどの要素も $A[q]$ 以下となり，$A[q+1]..r$ のどの要素も $A[q]$ 以上になるように分割する．添字 q はこの分割手続きの中で計算する．

統治： 2 つの部分配列 $A[p..q-1]$ と $A[q..r]$ について，クイックソートを再帰的に呼び出すことでソートする．

結合： 2 つの部分配列はソート済みだから，これらを結合するのに特にすることはない．既に配列 $A[p..r]$ はソートされている．

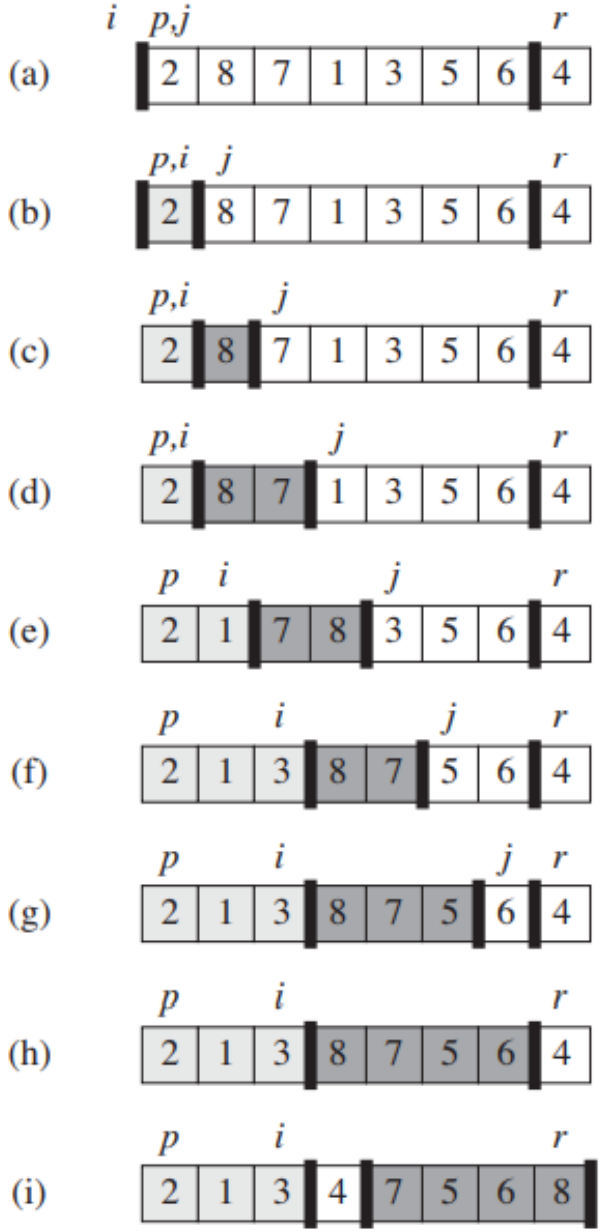
配列の分割 PARTITION の擬似コード

部分配列 $A[p..r]$ をその場で分割する手続きが PARTITION である．
$A[r]$ を常に**ピボット**として選択する．
($n = p - r + 1$ とすると，実行時間 $\Theta(n)$ である．)

Algorithm PARTITION(A, p, r)

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          swap(A[i], A[j])
7  swap(A[i + 1], A[r])
8  return i + 1
```

右の図は，PARTITION([2,8,7,1,3,5,6,4],1,8) の動作を表している．



クイックソートの擬似コード

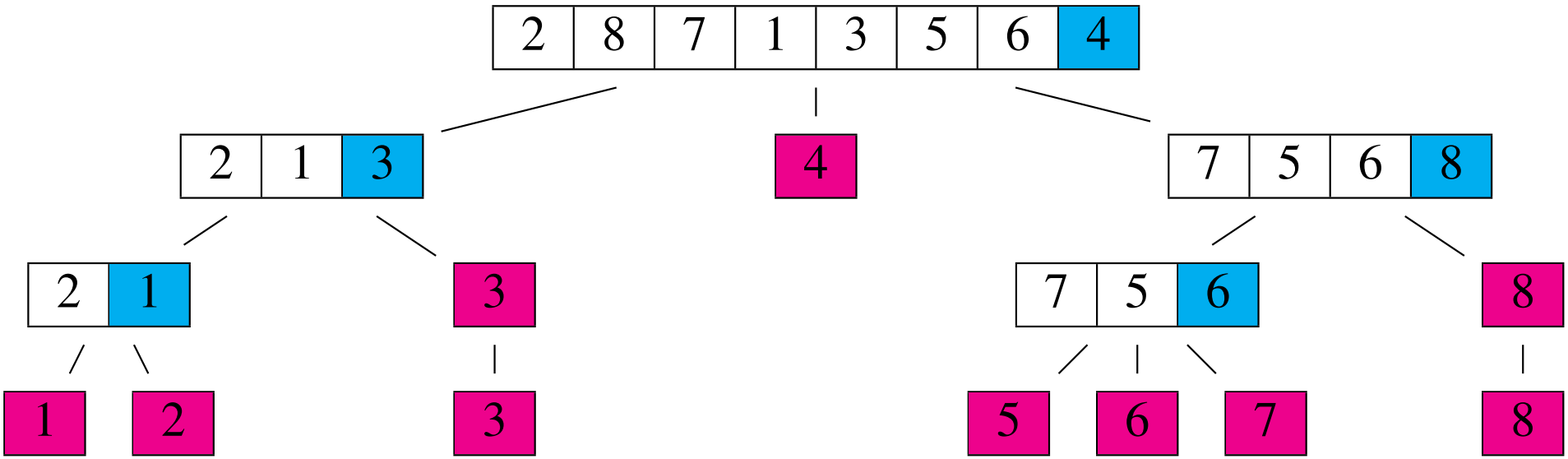
クイックソートは次の手続きによって実現される．

Algorithm QUICKSORT(A, p, r)

```
1  if p < r
2      q =PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

配列 A 全体をソートするための初期呼び出しは
QUICKSORT$(A, 1, A.length)$ である．

クイックソートの動作



クイックソートの性能

クイックソートの実行時間は分割が均等か否かに依存する．分割が均等ならば，クイックソートは漸近的にマージソートと同程度に高速だが，分割が均等でなければ，漸近的に挿入ソートと同程度に遅くなることがある．
クイックソートの性能を分割が均等である場合と不均等である場合について直観的に議論する．
ここで，要素数 n の配列に対してクイックソートを適用することを考える．

最悪の分割

クイックソートが最悪の振る舞いをするのは，直観的に分割時に元の問題を要素数 $n-1$ の部分問題と要素数 0 の部分問題に分割したときである．最悪のケースとして，アルゴリズムの各再帰呼び出しで片寄った分割が起こると仮定する．分割 (PARTITION) には $\Theta(n)$ 時間かかるので，実行時間は，漸化式

$$\begin{aligned} T(n) &= \Theta(n) + T(n-1) + T(0) \\ &= \Theta(n) + T(n-1) \end{aligned}$$

で記述される．直観的には再帰の各レベルでのコストを足し合わせると算術級数を得ることができ，$T(n) = \Theta(n^2)$ と評価できる．(厳密な証明には置き換え法を用いる．)
したがって，アルゴリズムの各再帰レベルにおいて分割が最大限に片寄ったならば，実行時間は $\Theta(n^2)$ となる．
このケースは，入力配列が既に昇順または降順にソートされている場合に起こる．

最良の分割

クイックソートが最良の振る舞いをするのは，分割時に元の問題を要素数 $\lfloor n/2 \rfloor$ の部分問題と要素数 $\lceil n/2 \rceil - 1$ の部分問題に分割したときである．このとき，クイックソートの実行時間は，漸化式

$$T(n) = \Theta(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil - 1) \fallingdotseq \Theta(n) + 2T(n/2)$$

で記述される．マスター定理からこの漸化式の解は $T(n) = \Theta(n \lg n)$ である．

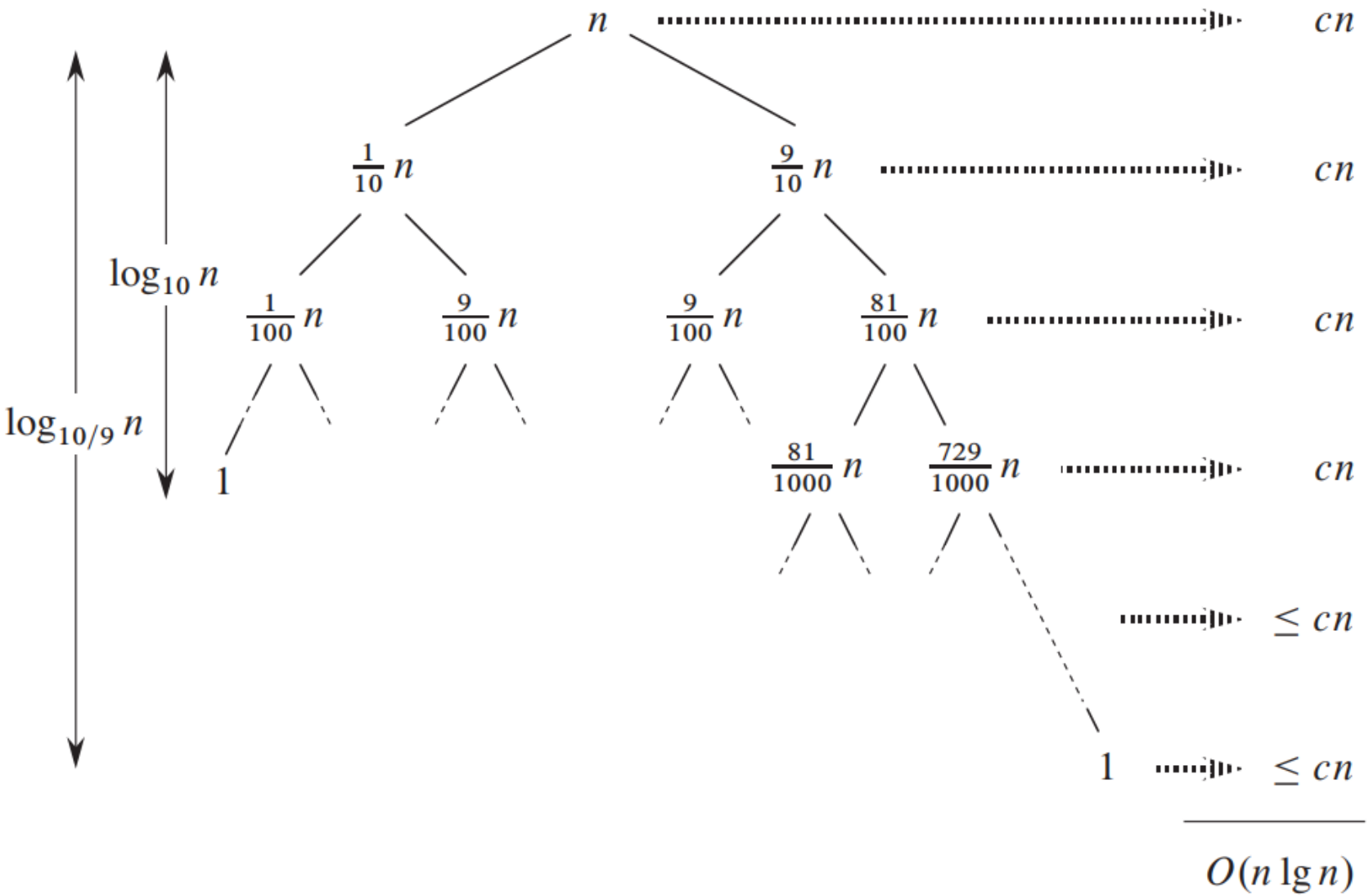
均等分割

分割アルゴリズムが常に 0.9:0.1 の比で元問題を分割すると仮定する．このとき，クイックソートの実行時間を表す漸化式は，c を $\Theta(n)$ に隠された定数であるとすると，

$$T(n) \fallingdotseq T(0.9n) + T(0.1n) + cn$$

である．この漸化式に対する再帰木を次ページに示す．
再帰木の各レベルの分割コストは cn であり，深さ $\log_{10/9} n = \Theta(\lg n)$ で再帰が終了するから，クイックソートの実行時間は $T(n) = \Theta(n \lg n)$ である．
したがって，再帰の各レベルを 0.9:0.1 の比で分割すると，直観的にかなり片寄った分割のように見えるが，漸近的に実行時間は真中で分割するのと同じである．
次々ページのスライドで一般的な議論をする．

再帰木


n
cn
1/10 n
9/10 n
cn
log10 n
1/100 n
9/100 n
9/100 n
81/100 n
cn
log10/9 n
1
81/1000 n
729/1000 n
cn
≤ cn
1
≤ cn
O(n lg n)

均等分割

任意の定数 $\alpha(0 < \alpha \leq 1/2)$ に対して，クイックソートの各レベルで $1-\alpha : \alpha$ の比で分割が行なわれると仮定する．整数の丸めを無視したとき，再帰木の最小の深さは $-\lg n/\lg \alpha$, 最大の深さは $-\lg n/\lg(1-\alpha)$ で近似できる．

Proof.

再帰木の最小の深さ d_{min} は α 側に分割され続けたときで，その深さは，

$$n \cdot \alpha^{d_{min}} = 1 \iff d_{min} = \log_{\alpha} \frac{1}{n} = -\frac{\lg n}{\lg \alpha}$$

となる．
同様に，再帰木の最大の深さ d_{max} は $1-\alpha$ 側に分割され続けたときで，その深さは，

$$n \cdot (1-\alpha)^{d_{max}} = 1 \iff d_{max} = \log_{(1-\alpha)} \frac{1}{n} = -\frac{\lg n}{\lg(1-\alpha)}$$

均等分割

したがって，分割比が**定数**である限り，再帰木の深さは $\Theta(\lg n)$, 各レベルの分割コストは $\Theta(n)$ から，実行時間は $T(n) = \Theta(n \lg n)$ である．

乱択版クイックソート

入力サイズが十分大きいとき，乱択版クイックソートは有力なソーティングアルゴリズムとして広く認知されている．
具体的には，$A[r]$ を常にピボットとする代わりに，部分配列 $A[p..r]$ から要素を無作為に抽出し，それをピボットとして用いる．ピボット要素を無作為に抽出するので，平均的には入力配列はそれなりにうまく 2 分割されると期待できる．
乱択版でないクイックソートの擬似コードに対して，分割の前にピボット要素の抽出を行うことで実装することができる．

乱択版クイックソートの手続きの擬似コード

Algorithm RANDOMIZED-PARTITION(A, p, r)

```
1 i =RANDOM(p, r)
2 swap(A[r], A[i])
3 return PARTITION(A, p, r)
```

Algorithm RANDOMIZED-QUICKSORT(A, p, r)

```
1 if p < r
2     q =RANDOMIZED-PARTITION(A, p, r)
3     RANDOMIZED-QUICKSORT(A, p, q − 1)
4     RANDOMIZED-QUICKSORT(A, q + 1, r)
```

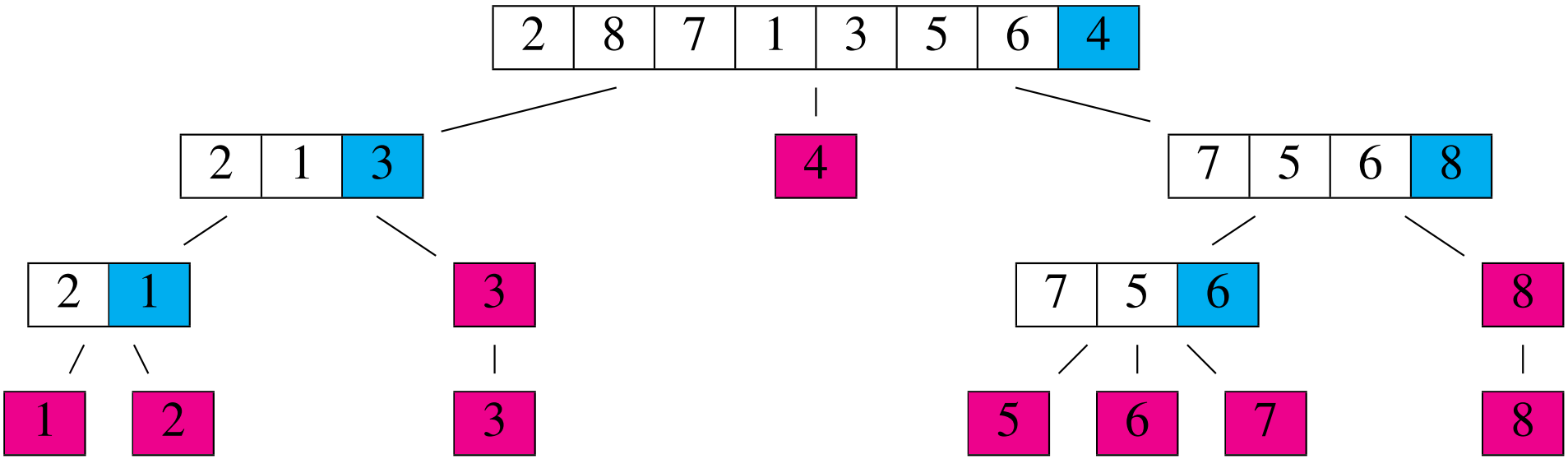
クイックソートの実行時間の解析

乱択版クイックソート (RANDOMIZED-QUICKSORT) の期待実行時間を解析する．
ここで，簡単のためソートする要素は相異なると仮定する．

実行時間と比較

PARTITION は呼び出される度にピボット要素を選択し，以降の PARTITION の呼び出し時に既にピボット要素として選択された要素は含まれないので，クイックソートの実行全体で PARTITION は高々 n 回しか呼び出されない．
また，1 回の PARTITION の呼び出しには $O(1)$ 時間と 3〜6 行目の **for** 文の繰り返し数に比例した時間を要する．**for** 文の各繰り返しで第 4 行でピボット要素を配列 A の他の要素と比較する．したがって，第 4 行の比較の総実行回数を数えれば，QUICKSORT の実行全体における **for** 文の総実行時間が評価できる．
すなわち，QUICKSORT の実行全体において，PARTITION の第 4 行の比較総数を X としたとき，QUICKSORT の実行時間は $O(n + X)$ となる．
PARTITION と RANDOMIZED-PARTITION の呼び出し回数・計算時間のオーダーは等しいので，"PARTITION"
を"RANDOMIZED-PARTITION"に読み替えても問題ない．

クイックソートの動作 (再掲)



PARTITION の擬似コード (再掲)

Algorithm RANDOMIZED-PARTITION(A, p, r)

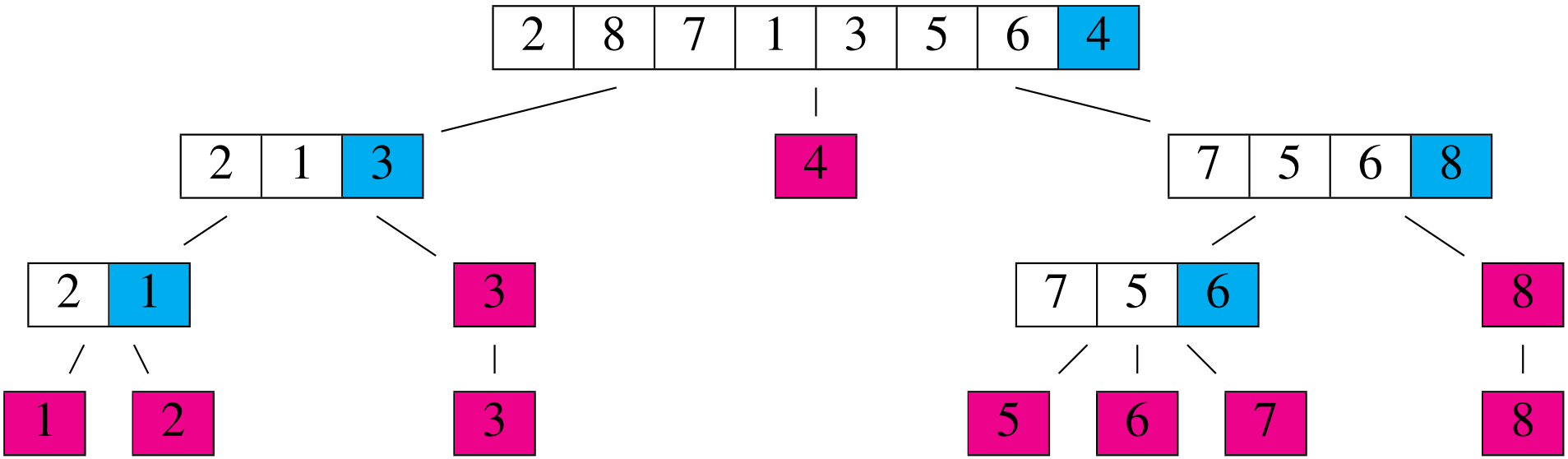
```
1 i =RANDOM(p, r)
2 swap(A[r], A[i])
3 return PARTITION(A, p, r)
```

Algorithm PARTITION(A, p, r)

```
1 x = A[r]
2 i = p − 1
3 for j = p to r − 1
4       if A[j] ≤ x
5             i = i + 1
6             swap(A[i], A[j])
7 swap(A[i + 1], A[r])
8 return i + 1
```

ここで，配列 A の要素について，i 番目に小さい値を z_i と定義し，また，$Z_{ij} = \{z_i, z_{i+1}, \cdots, z_j\}$ と定義する．すなわち，Z_{ij} は z_i 以上 z_j 以下の値を持つ A の要素の集合である．
今，各要素はピボット要素とだけ比較され，PARTITION の呼び出しの終了後，その呼び出し時に用いられたピボット要素は他の要素と二度と比較されることがないので，アルゴリズムは要素の対 (z_i, z_j) を高々 1 回しか比較しない．

クイックソートの動作（再掲）



ここで，指標確率変数 X_{ij} を，

$$X_{ij} = \begin{cases} 1 & (z_i と\ z_j が比較される) \\ 0 & (otherwise) \end{cases}$$

と定義する．このとき，アルゴリズムが実行する比較回数 X は，

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

となる．この期待値は，

$$\begin{aligned} E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i と\ z_j が比較される\} \end{aligned}$$

ここで，RANDOMIZED-PARTITION 手続きが各ピボットを一様かつ独立に選択すると仮定し，$\Pr\{z_i \text{と} z_j \text{が比較される}\}$ を求める．
ここで，ある 2 つの要素が比較されない理由について考える．例として，1〜10 までの数が任意の順でクイックソートに入力され，最初のピボット要素が 7 であると仮定する．このとき，最初の PARTITION の呼び出しは入力集合を $\{1, 2, 3, 4, 5, 6\}$ と $\{8, 9, 10\}$ に分割する．分割のためにピボット要素 7 は他の全ての要素と比較されるが，$\{1, 2, 3, 4, 5, 6\}$ の任意の要素と $\{8, 9, 10\}$ の任意の要素は，今も，また今後も絶対に比較されることはない．

ここで，入力値は全て異なると仮定しているので，一般に，$z_i < x < z_j$ を満たすピボット x が選択されると，それ以降に z_i と z_j が比較されることはない．
したがって，要素対 (z_i, z_j) は z_i または z_j が Z_{ij} から選ばれる最初のピボットであるときに限り，比較される．
Z_{ij} には $j - i + 1$ 個の要素があり，ピボットは一様かつ独立に選択されると仮定しているので，Z_{ij} の各要素がピボットとして最初に選択される確率はそれぞれ等しく $1/(j - i + 1)$ である．

したがって，

$$\begin{aligned}\Pr\{z_i \text{と} z_j \text{が比較される}\} &= \Pr\{z_i \text{または} z_j \text{が} Z_{ij} \text{から選ばれる最初のピボット}\} \\ &= \Pr\{z_i \text{が} Z_{ij} \text{から選ばれる最初のピボット}\} \\ &\quad + \Pr\{z_j \text{が} Z_{ij} \text{から選ばれる最初のピボット}\} \\ &= \frac{1}{j-i+1} + \frac{1}{j-i+1} \\ &= \frac{2}{j-i+1}\end{aligned}$$

よって，

$$\begin{aligned}E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr\{z_i \text{と} z_j \text{が比較される}\} \\ &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}\end{aligned}$$

となる．

変数を $k = j - i$ によって変換し，調和級数の和の公式 (A.7)$\left(\sum_{k=1}^{n} \frac{1}{k} = \lg n + O(1)\right)$ を使うと，次のように評価される．

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= 2 \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{1}{k} \\
&= 2 \sum_{i=1}^{n-1} (\lg n + O(1)) \\
&= O(n \lg n)
\end{aligned}
$$

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\geq \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \\
&= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{k=1}^{n-i} \frac{1}{k} + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k} \\
&> \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{k=1}^{n-i} \frac{1}{k} \\
&\geq \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \sum_{k=1}^{\lceil \frac{n}{2} \rceil} \frac{1}{k} \\
&= \sum_{i=1}^{\lfloor \frac{n}{2} \rfloor} \left(\lg \left\lceil \frac{n}{2} \right\rceil + O(1)\right) \\
&= \Omega(n \lg n)
\end{aligned}
$$

よって，アルゴリズムが実行する比較回数 X の期待値 $E[X]$ について，$E[X] = O(n \lg n)$ かつ $E[X] = \Omega(n \lg n)$ より，$E[X] = \Theta(n \lg n)$ が成立する．
よって，RANDOMIZED-PARTITION を用いるとクイックソートの期待実行時間は，n 回 RANDOMIZED-PARTITION を呼び出すための $O(n)$ 時間と，アルゴリズムが実行する比較回数の和から，
$O(n) + \Theta(n \lg n) = \Theta(n \lg n)$ と求められる．

クイックソートまとめ

※本スライドでは扱わないが，全ての要素の値が異なるという仮定を置いていたが，その仮定を無くしても，クイックソートの性質は変わらない．
クイックソートは外部メモリを高々定数個しか必要としないので，内部ソートである．
(乱択版) クイックソートではピボット要素の選択について，確率に支配されており，ピボット要素の選び方が悪いと，タイ値の順序を保持できず，安定性を持たない．

クイックソートまとめ

この表でのクイックソートとは，乱択版クイックソートのことである．

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

平均・期待実行時間 (再掲)

平均実行時間

全ての可能な入力に対する実行時間の平均.

期待実行時間

乱数生成器が返す値の分布の上で実行時間の期待値を取ったもの.

※平均実行時間は平均を取る対象が入力全体であり，期待実行時間は平均を取る対象が乱数全体である.

同じ値の要素がある場合の分割の手続き (PARTITION')

Algorithm PARTITION'(A, p, r)

```
1  x = A[r]
2  swap(A[r], A[p])
3  i = p − 1
4  k = p
5  for j = p + 1 to r-1
6      if A[j] < x
7          i = i + 1
8          k = i + 2
9          swap(A[i], A[j])
10         swap(A[k], A[j])
11     if A[j] == x
12         k = k + 1
13         swap(A[k], A[j])
14 swap(A[i + 1], A[r])
15 return  i + 1, k + 1
```

同じ値があるときの乱択版の分割の手続き (RANDOMIZED-PARTITION')

Algorithm RANDOMIZED-PARTITION'(A, p, r)

```
1 i =RANDOM(p, r)
2 swap(A[r], A[i])
3 return PARTITION'(A, p, r)
```

同じ値があるときの乱択版クイックソート (RANDOMIZED-QUICKSORT')

Algorithm RANDOMIZED-QUICKSORT'(A, p, r)

```
1 if p < r
2     q, t =RANDOMIZED-PARTITION'(A, p, r)
3     QUICKSORT'(A, p, q − 1)
4     QUICKSORT'(A, t + 1, r)
```

目次

比較ソートとそうでないソート

これまでに n 個の数列を $O(n \lg n)$ 時間でソートするアルゴリズムをいくつか紹介した．
これまでに紹介したソートアルゴリズムは全て比較ソートであった．
この章では，まず，任意の比較ソートは n 個の要素をソートするために最悪 $\Omega(n \lg n)$ 回の比較が必要であり，比較ソートでは $\Omega(n \lg n)$ 時間より早いアルゴリズムは存在しないことを証明する．
次に，線形時間で走る計数ソート，基数ソート，バケツソートを紹介する．これらのアルゴリズムはソート順を決定するために比較以外の演算を用いているので，計算時間の下界 $\Omega(n \lg n)$ は適用されない．

8.1 ソートの下界

比較ソートでは要素間の比較だけを用いて入力列 $\langle a_1, a_2, \cdots, a_n \rangle$ に関する順序情報を得る．
ここで，全ての比較は形 $a_i \leq a_j$ を持つと仮定する．

決定木モデル

比較ソートは決定木の概念を用いて抽象的に表現できる．

決定木

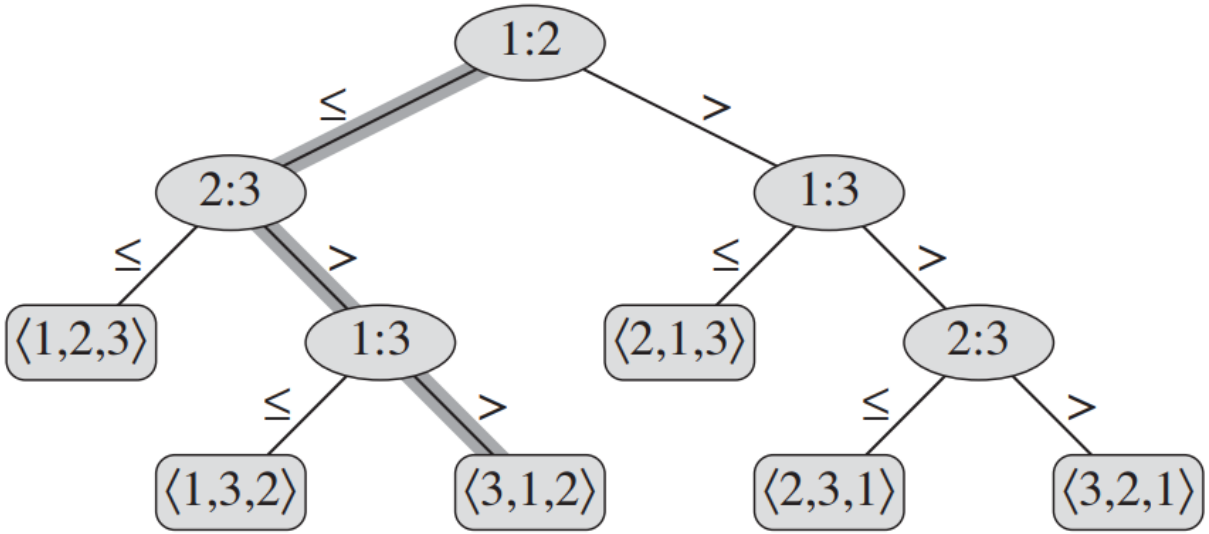
決定木は，与えられたサイズの入力上で動作する特定のソーティングアルゴリズムが実行する要素間の比較を表現する全 2 分木である．

全 2 分木

各節点は葉であるか，あるいは次数が 2 である 2 分木．

決定木モデル

3 個の要素を持つ入力数列 $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ 上で動作する決定木を示す．



入力数列 $\langle a_1 = 6, a_2 = 8, a_3 = 5 \rangle$ をソートするときに下した決定列に対応する道を網掛けして示す．この道の終端の葉にラベル付けられた順列 $\langle 3, 1, 2 \rangle$ は入力数列のソートの結果が $a_3 = 5 \leq a_1 = 6 \leq a_2 = 8$ であることを示す．入力要素集合には $3! = 6$ 個の順列があるので，決定木は 6 個の葉を持つ．

決定木モデル

n を入力数列の要素数とする．決定木の各内部節点はある 2 項組 $i : j (1 \leq i, j \leq n)$ をラベルとして持ち，各葉はある順列 $\langle \pi(1), \pi(2), \cdots, \pi(n) \rangle$ をラベルとして持つ．
ソーティングアルゴリズムの実行は決定木の根からある葉に至る単純道を辿ることに対応する．
ラベル $i : j$ を持つ内部節点では比較 $a_i \leq a_j$ を実行する．
この内部節点の左部分木は $a_i \leq a_j$ である場合の以降の比較手続きを記述し，右部分木は $a_i > a_j$ である場合の以降の比較手続きを記述する．

決定木モデル

ソーティングアルゴリズムの実行が葉に到達すると順序 $a_{\pi(1)} \leq a_{\pi(2)} \leq \cdots \leq a_{\pi(n)}$ が確定する．
任意の正当なソーティングアルゴリズムは入力の全ての順列を生成できることが必要なので，比較ソートが正当であるためには，n 個の要素から構成される $n!$ 個の順列のそれぞれが決定木の葉の 1 つとして出現する必要がある．
しかも，これらの葉はそれぞれ根から木を下る道 (このような道は比較ソートの実際の実行に対応する) を辿ることで到達可能でなければならない．
このような葉を「到達可能」な葉と呼び，全ての順列が到達可能な葉として出現する決定木のみを以下では考える．

比較ソートの最悪時の下界

決定木の高さは与えられた比較ソートの最大比較回数に等しいので，任意の比較ソートアルゴリズムは最悪時に $\Omega(n \lg n)$ 回の比較が必要である．

Proof.

高さ h の全 2 分木の葉の数は 2^h 以下であるので，各々の順列をラベルに持つ $n!$ 個の葉を持つための高さ h は，不等式

$$n! \leq 2^h$$

を満たす．両辺の対数を取ると，

$$h \geq \lg n! = \Omega(n \lg n) (\because \text{式 (3.19)})$$

ここで，決定木の高さは与えられた比較ソートの最大比較回数に等しいので，任意の比較ソートアルゴリズムは最悪時に $\Omega(n \lg n)$ 回の比較が必要である． □

最悪時の下界

ヒープソートとマージソートの実行時間の上界 $O(n \lg n)$ は比較ソートの最悪時の下界 $\Omega(n \lg n)$ と一致するので，ヒープソートとマージソートは共に漸近的に最適な比較ソートであるといえる．

計数ソート

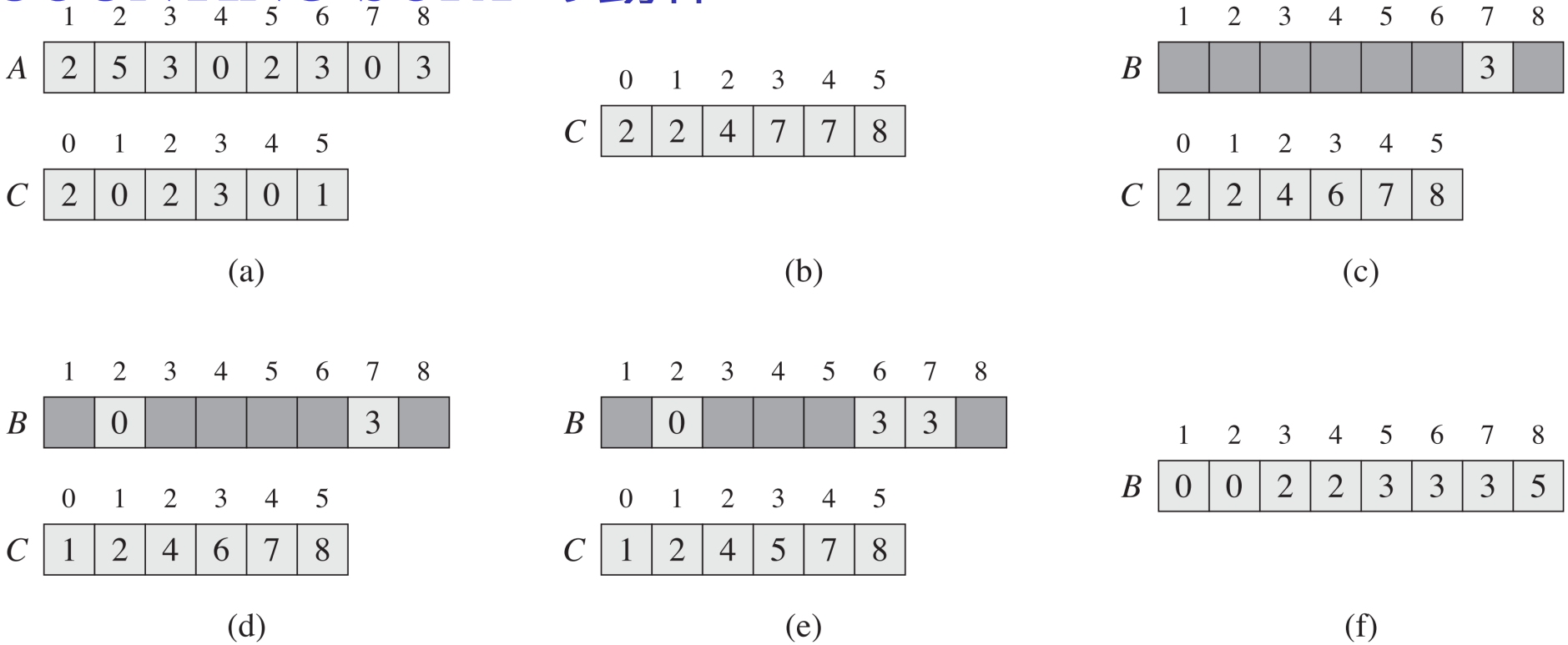
計数ソートでは，n 個の入力要素はある整数 k に対して 0 から k の範囲の整数から選ばれると仮定する．$k = O(n)$ ならば計数ソートは $O(n)$ 時間で走るアルゴリズムである．
計数ソート (COUNTING-SORT(A, B, k)) の擬似コードを次のスライドに示す．
ここで，A は入力配列 $A[1..n]$,B はソート結果を格納する配列 $B[1..n]$，C は一時的な補助領域として利用される配列 $C[0..k]$ である．

COUNTING-SORT の擬似コード

Algorithm COUNTING-SORT(A, B, k)

```
1  let C[0..k] be a new array
2  for i = 0 to k
3      C[i] = 0
4  for j = 1 to A.length
5      C[A[j]] = C[A[j]] + 1
6  // C[i] now contains the number of elements equal to i.
7  for i = 1 to k
8      C[i] = C[i] + C[i − 1]
9  // C[i] now contains the number of elements less than or equal to i.
10 for j = A.length downto 1
11     B[C[A[j]]] = A[j]
12     C[A[j]] = C[A[j]] − 1
```

COUNTING-SORT の動作



上の図は，入力配列 $A[1..8] = \langle 2, 5, 3, 0, 2, 3, 0, 3 \rangle$, $k = 5$ のときの，COUNTING-SORT の動作を表している．ここで，A の各要素は $k = 5$ 以下の非負整数である．

- (a) 第 4〜5 行の **for** 文を終了した後の配列 A と，補助配列 C．
- (b) 第 7〜8 行の **for** 文を終了した後の補助配列 C．
- (c)〜(e) 第 10〜12 行の **for** 文の 1,2,3 回目の繰り返し直後の出力配列 B と補助配列 C．
- (f) ソートされた最終出力配列 B.

COUNTING-SORT の実行時間

計数ソートの実行時間について考える．第 2〜3 行の **for** 文に $\Theta(k)$ 時間，第 4〜5 行の **for** 文に $\Theta(n)$ 時間，第 7〜8 行の **for** 文に $\Theta(k)$ 時間，第 10〜12 行の **for** 文に $\Theta(n)$ 時間かかる．したがって，全体の実行時間は $\Theta(k+n)$ である．
$k = O(n)$ のときに計数ソートを用いると，実行時間は $\Theta(n)$ となる．

安定ソート

計数ソートは**安定性**を満たすソートである．

安定性

同じ値の要素は入力に出現する順序で出力に出現するという性質．

計数ソートの安定性は次に述べる基数ソートが正当に働くために必要である．

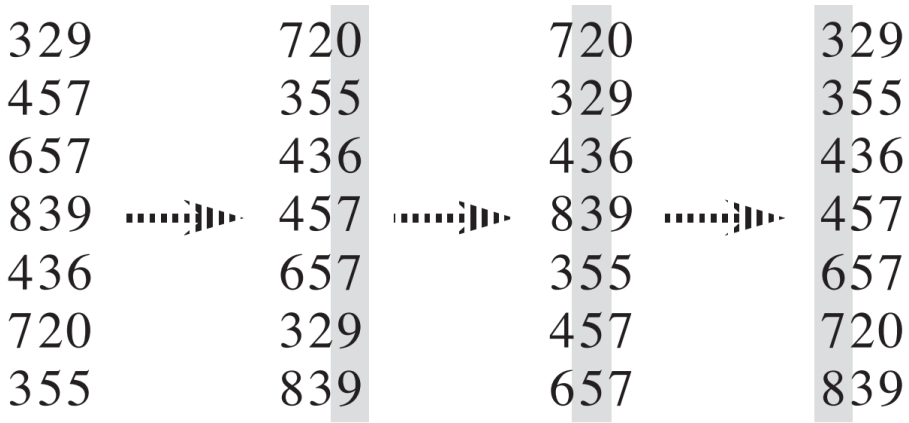
計数ソートまとめ

計数ソートは配列 B, C を必要とする (余分に $n+k$ 個の領域が必要である) ので，内部ソートではない．また，安定ソートである．

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

基数ソート

基数ソートでは，最下位の桁で数列をソートして，次に最下位から 2 桁目に関してソートし,,, という風にこの過程を繰り返して d 桁全てに関して数をソートする．



基数ソートが正しく働くには各桁のソートは安定でなければならない．

- 最上位の桁からソートすると？ (例).$\langle 432, 324 \rangle$)
- 各桁のソートが安定でないと？ (例).$\langle 124, 123 \rangle$)

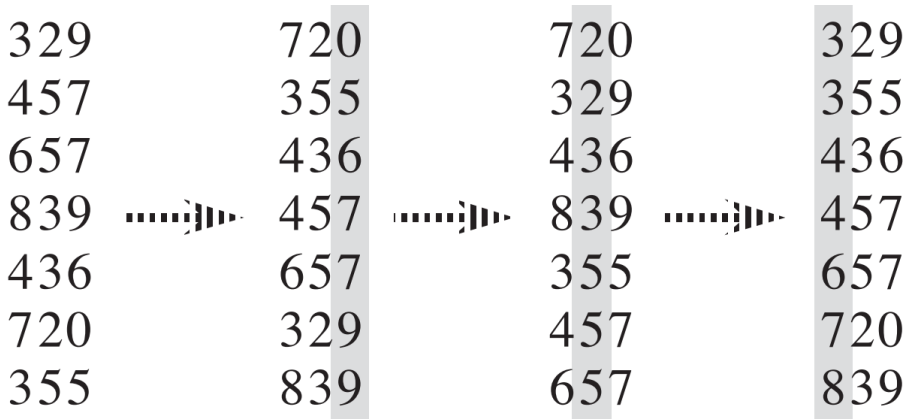
RADIX-SORT の擬似コード

基数ソートの擬似コードを示す．A は n 個の要素を持つ配列であり，その要素は d 桁の数であり，第 1 桁が最下位桁，第 d 桁が最上位桁であるとする．

Algorithm RADIX-SORT(A, d)

```
1 for i = 1 to d
2     use a stable sort to sort array A on digit i
```

次に，入力配列 $A = \langle 329, 457, 657, 839, 436, 720, 355 \rangle$, $d = 3$ のときの基数ソートの動作を示す．



RADIX-SORT の実行時間

基数ソート (RADIX-SORT) の実行時間は **for** 文のループ回数 d と安定ソートの実行時間の積となる．
よって，その安定ソートの実行時間が $\Theta(n+k)$ ならば，RADIX-SORT の実行時間は $\Theta\left(d(n+k)\right)$ となる．
すなわち，d が定数で，$k = O(n)$ ならば，基数ソートは線形時間で走る．

基数ソートまとめ

この表では，安定ソートとして，計数ソートを用いたときの性質を書いている．
計数ソートが内部ソートでないため，計数ソートを用いたとき基数ソートは内部ソートでない．

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×

バケツソート

バケツソートは入力が一様分布から抽出されると仮定するとき，平均実行時間 $O(n)$ を達成する．バケツソートが高速なのは，計数ソートと同様に，入力にある仮定を置いているからである．
バケツソートは，区間 $[0..1)$ を n 個の等しい大きさの**バケツ**と呼ぶ部分区間に分割し，n 個の入力をバケツに分割する．入力は $[0..1)$ 上に一様独立に分布しているので，多くの数が 1 つのバケツに集中しないと期待できる．そこで，各バケツごとにその中の数をソートし，次に 1 番小さい区間に対応するバケツから順番にその中のソート済みの要素をその順序で並べれば，ソート済みの数列を生成できる．

BUCKET-SORT の擬似コード

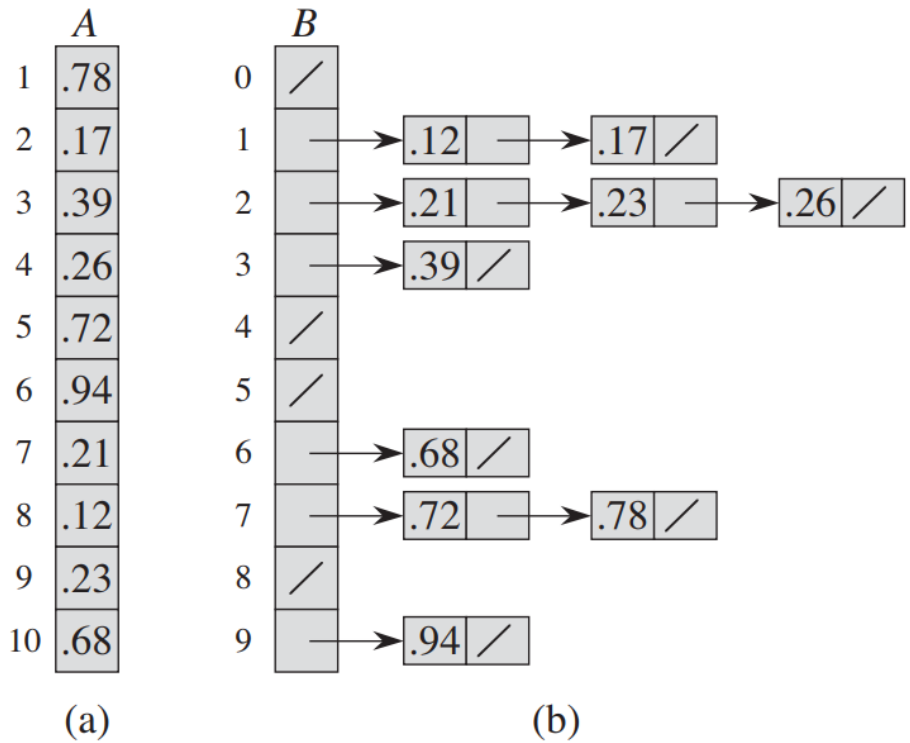
バケツソートの擬似コードを示す．A は n 個の要素を持つ配列であり，各要素は $0 \leq A[i] < 1$ を満たすと仮定する．B はリストを要素とする配列である．

Algorithm BUCKET-SORT(A)

```
1 n = A.length
2 let B[0..n − 1] be a new array
3 for i = 0 to n − 1
4       make B[i] an empty list
5 for i = 1 to n − 1
6       insert A[i] into list B[⌊nA[i]⌋]
7 for i = 0 to n − 1
8       sort list B[i] with insertion sort
9 concatenate the lists B[0], B[1], ··· , B[n − 1] together in order
```

BUCKET-SORT の動作

入力配列 $\langle 0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.21, 0.12, 0.23, 0.68 \rangle$ 上のバケツソートの動作を示す．



(a) 入力配列 $A[1..10]$

(b) アルゴリズムの第 8 行目を実行後のソートされたリスト (バケツ) の配列 $B[0..9]$. バケツ i は半開区間 $[i/10, (i+1)/10)$ に属する値を保持している．ソート済みの出力はリスト $B[0], B[1], \cdots, B[9]$ をこの順序で連結したものである．

BUCKET-SORT の実行時間

第 7,8 行目以外の実行時間は $\Theta(n)$ である．
ここで，第 8 行の挿入ソートの呼び出しコストを解析する．n_i をバケツ $B[i]$ に入る要素数を表す確率変数とする．挿入ソートは 2 次の多項式時間で走るので，バケツソートの実行時間 $T(n)$ は，

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

である．

BUCKET-SORT の実行時間

ここで，実行時間の期待値を計算することで，バケツソートの平均実行時間を解析する．期待値の線形性に注意して，

$$
\begin{aligned}
E[T(n)] &= E\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} E\left[O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(E[n_i^2]\right)
\end{aligned}
$$

を得る．

BUCKET-SORT の実行時間

ここで，指標確率変数 X_{ij} を次のように定義する．

$$X_{ij} = \begin{cases} 1 & (A[j] \text{ がバケツ } i \text{ に入る }) \\ 0 & (otherwise) \end{cases}$$

このとき，

$$n_i = \sum_{j=1}^{n} X_{ij}$$

となる．

BUCKET-SORT の実行時間

よって，

$$
\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= E\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= E\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\neq j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n} E\left[X_{ij}^2\right] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n\\ k\neq j}} E\left[X_{ij}X_{ik}\right]
\end{aligned}
$$

となる．

BUCKET-SORT の実行時間

右辺第一項について，指標確率変数 X_{ij} は確率 $1/n$ で値 1 を取り，それ以外では 0 となるので，

$$E\left[X_{ij}^2\right] = 1^2 \cdot \frac{1}{n} + 0^2 \cdot \left(1 - \frac{1}{n}\right) = \frac{1}{n}$$

である．また，右辺第二項について，$k \neq j$ ならば，確率変数 X_{ij} と X_{ik} は独立であるので，

$$E\left[X_{ij}X_{jk}\right] = E[X_{ij}]E[X_{ik}] = \frac{1}{n} \cdot \frac{1}{n} = \frac{1}{n^2}$$

よって，元々の式に代入して，

$$E[n_i^2] = \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2}$$

$$= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

となる．

BUCKET-SORT の実行時間

よって，

$$
\begin{aligned}
E\left[T(n)\right] &= \Theta(n) + \sum_{i=0}^{n-1} O\left(E[n_i^2]\right) \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(2 - \frac{1}{n}\right) \\
&= \Theta(n) + n \cdot O(2 - \frac{1}{n}) \\
&= \Theta(n)
\end{aligned}
$$

となり，バケツソートの平均実行時間は $\Theta(n)$ であることが分かった．

バケツソートまとめ

要素数 n の配列 B のためのメモリが必要なので，バケツソートは内部ソートでない．

アルゴリズム	最悪実行時間	平均/期待実行時間	内部	安定	比較
挿入ソート	$\Theta(n^2)$	$\Theta(n^2)$	○	○	○
マージソート	$\Theta(n \lg n)$	$\Theta(n \lg n)$	×	○	○
ヒープソート	$O(n \lg n)$	–	○	×	○
クイックソート	$\Theta(n^2)$	$\Theta(n \lg n)$(期待時間)	○	×	○
計数ソート	$\Theta(n+k)$	$\Theta(n+k)$	×	○	×
基数ソート	$\Theta(d(n+k))$	$\Theta(d(n+k))$	×	○	×
バケツソート	$\Theta(n^2)$	$\Theta(n)$	×	○	×