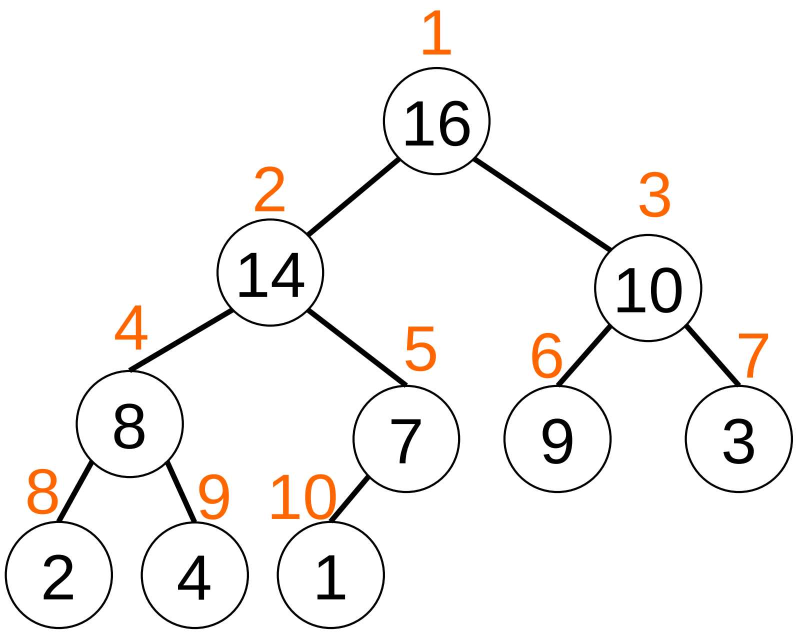
算法数理工学
第3回

定兼　邦彦

ヒープ

- ヒープ：完全2分木とみなせる配列
- 木の各節点は配列の要素に対応
- 木は最下位レベル以外の全てのレベルの点が完全に詰まっている
- 最下位のレベルは左から順に詰まっている



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

ヒープ条件 (Heap Property)

- 根以外の任意の節点 i に対して

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

- つまり, 節点の値はその親の値以下
- ヒープの最大要素は根に格納される

ヒープの操作

- HEAPIFY: ヒープ条件を保持する.
- BUILD_HEAP: 入力の配列からヒープを構成する. 線形時間.
- HEAPSORT: 配列をソートする. O(n lg n) 時間.
- EXTRACT_MAX: ヒープの最大値を取り出す. O(lg n) 時間.
- INSERT: ヒープに値を追加する. O(lg n) 時間.

ヒープ条件の保持

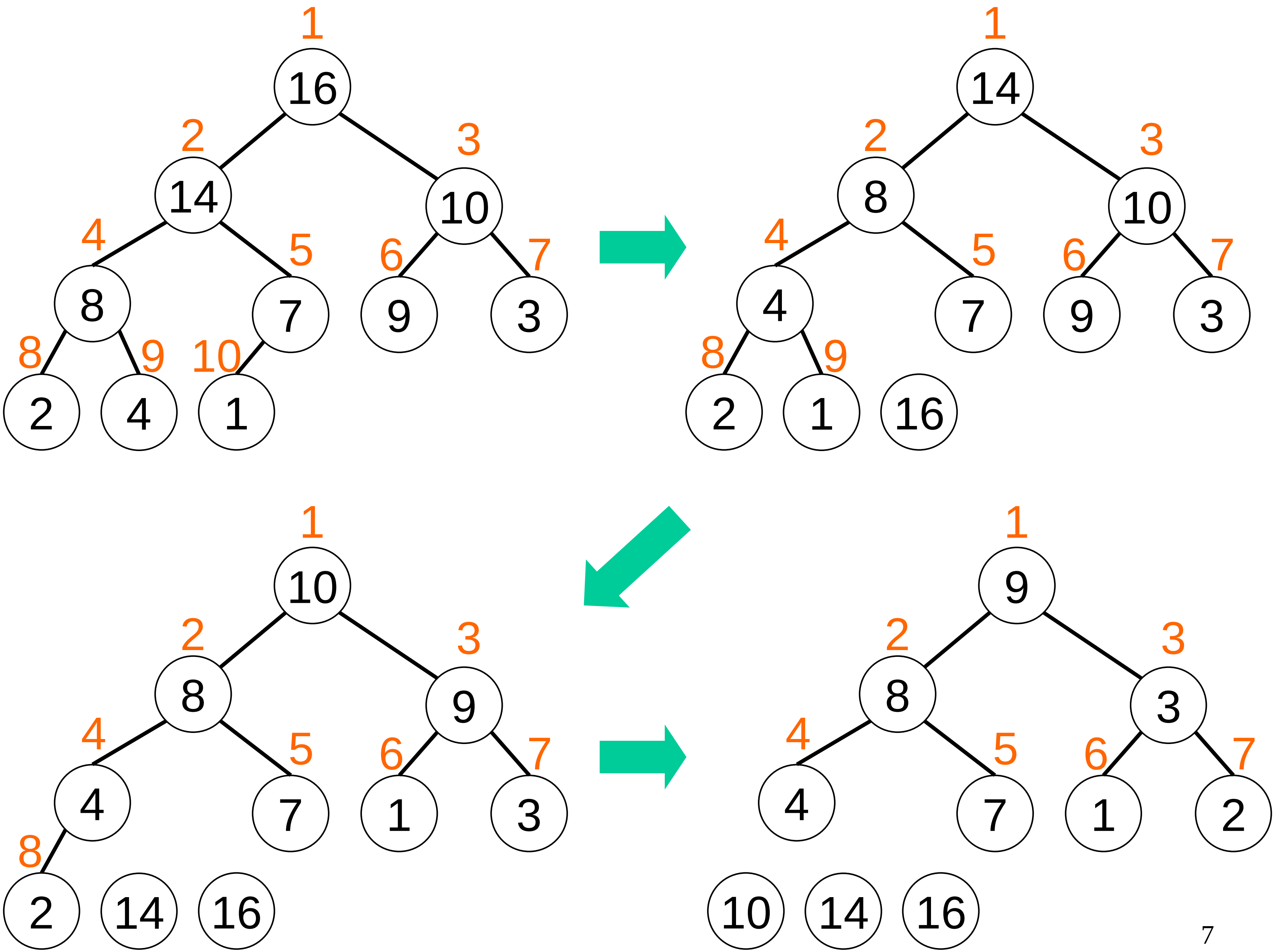
- HEAPIFY(H,i): $A[i]$ を根とする部分木がヒープになるようにする. ただし LEFT(i) と RIGHT(i) を根とする2分木はヒープと仮定.

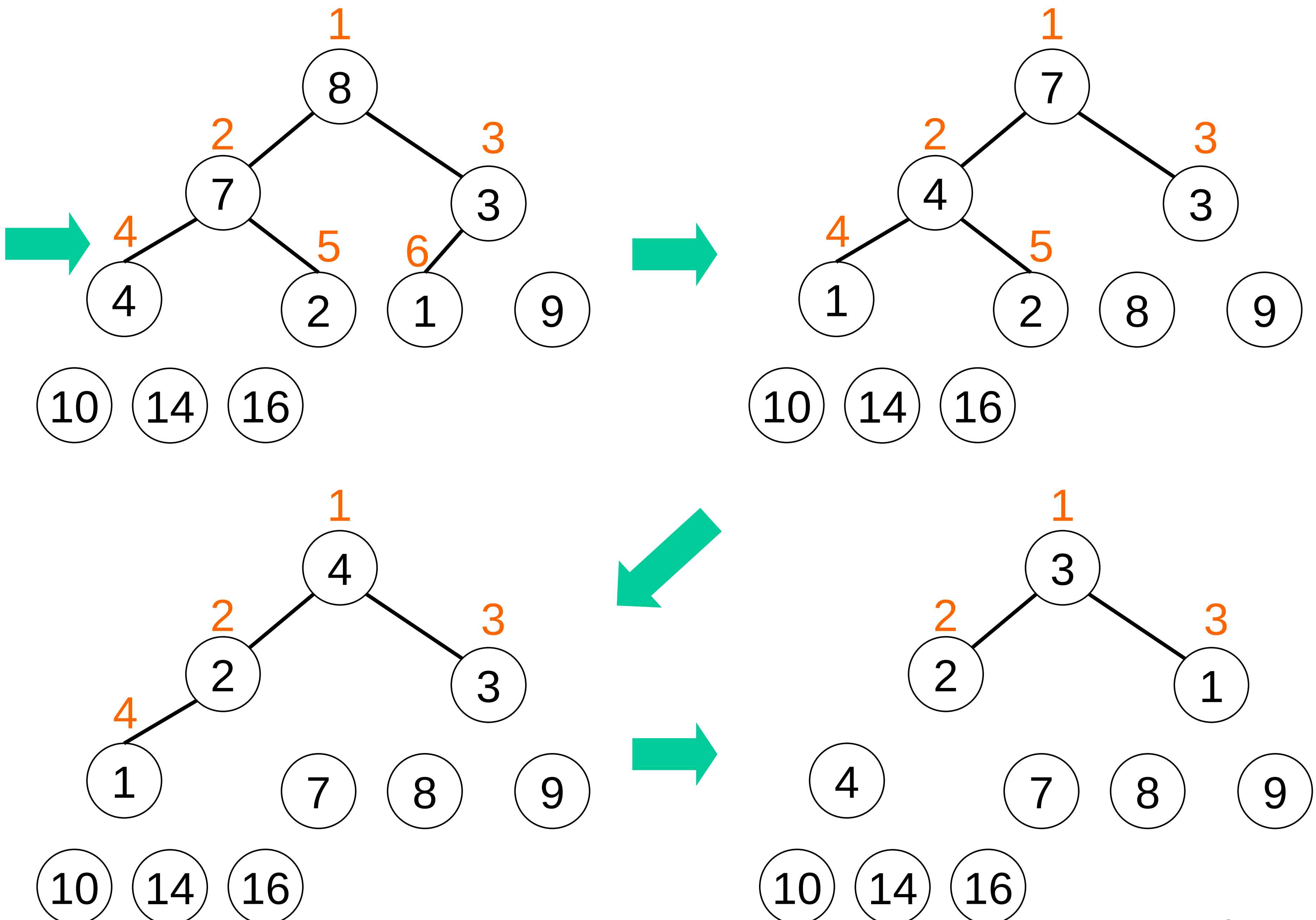
```
void HEAPIFY(HEAP *H, int i)
{
  int l, r, largest, heap_size;
  data tmp, *A;
  A = H->A;  heap_size = H->heap_size;
  l = LEFT(i);   r = RIGHT(i);
  if (l <= heap_size && A[l] > A[i]) largest = l; // A[i] と左の子で大きい
  else  largest = i;                               // 方をlargestに
  if (r <= heap_size && A[r] > A[largest])        // 右の子の方が大きい
       largest = r;
  if (largest != i) {
    tmp = A[i];  A[i] = A[largest];  A[largest] = tmp; // A[i]を子供と入れ替える
    HEAPIFY(H, largest);
  }
}
```

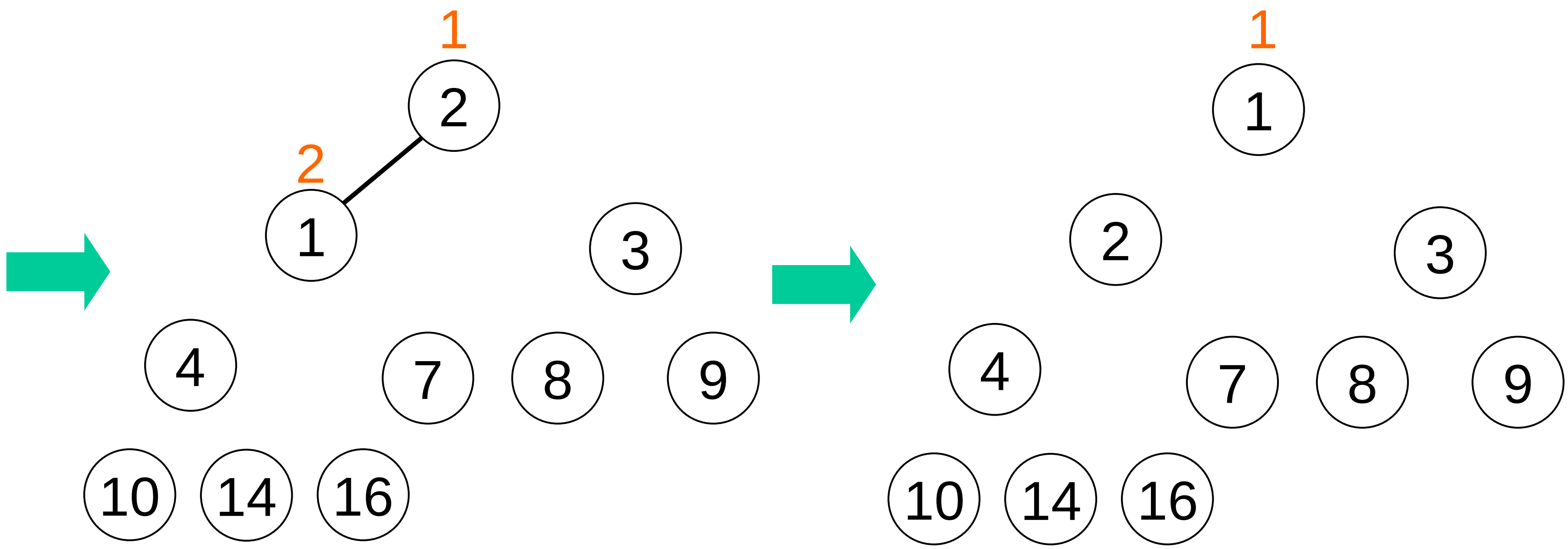
ヒープソート

- まずヒープを作る
- すると最大要素が $A[1]$ に入る
- $A[1]$と$A[n]$を交換すると，最大要素が$A[n]$に入る
- ヒープのサイズを1つ減らしてヒープを維持する

```
void HEAPSORT(int n, data *A)
{
  int i;
  data tmp;
  HEAP H;
  BUILD_HEAP(&H,n,A,n);
  for (i = n; i >= 2; i--) {
    tmp = A[1];  A[1] = A[i];  A[i] = tmp;  // 根と最後の要素を交換
    H.heap_size = H.heap_size - 1;
    HEAPIFY(&H,1);
  }
}
```


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


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



A	1	2	3	4	7	8	9	10	14	16

計算量

- BUILD_HEAP: O(*n*) 時間
- HEAPIFY: O(*n* lg *n*) 時間

全体で O(*n* lg *n*) 時間

要素の挿入

```
void INSERT(HEAP *H, data key)  // O(lg n) 時間
{
  int i;
  data *A;

  A = H->A;

  H->heap_size = H->heap_size + 1;
  if (H->heap_size > H->length) {
    printf("ERROR ヒープのオーバーフロー¥n");
    exit(1);
  }

  i = H->heap_size;
  while (i > 1 && A[PARENT(i)] < key) {
    A[i] = A[PARENT(i)];
    i = PARENT(i);
  }
  A[i] = key;
}
```

正当性の証明

- 新しい要素を配列の最後 $A[n]$ に置く
- $A[\mathrm{PARENT}(n)] \geq A[n]$ なら条件を満たす
- そうでなければ $A[n]$ と親を交換する

- つまり，根から $A[n]$ の親までのパス上の要素は大きい順に並んでいるので，$A[n]$ を挿入すべき場所を探索してそこに挿入する
- パス上の値は大きくなるだけなので，ヒープ条件は必ず満たしている

要素の削除

- 削除したい値がヒープ中のどこに格納されているか分かっているとする

```
void DELETE(HEAP *H, int i)  // O(lg n) 時間
{
  data *A;
  A = H->A;
  if (i < 1 || i > H->heap_size) {
    printf("ERROR 範囲エラー (%d,%d)¥n",i,H->heap_size);
    exit(1);
  }

  while (i > 1) {
    A[i] = A[PARENT(i)]; // A[i] の祖先を1つずつ下におろす
    i = PARENT(i);
  }
  A[1] = A[H->heap_size]; // 根が空になるので，最後の値を根に持っていく
  H->heap_size = H->heap_size - 1;
  HEAPIFY(H,1);
}
```

正当性の証明

- $A[i]$ を削除するとき，$A[i]$ から根までのパス上の値を1つずつ下ろす
- 値は大きくなるだけなのでヒープ条件は満たす
- 根の値が無くなるので，ヒープの最後の値を移動
- 根がヒープ条件を満たさなくなるのでHEAPIFYを行う

- 注意:削除したい値がヒープ中のどこにあるかは分からないときは，探索に O(n) 時間かかる

- ヒープに格納する値が 1 から n の整数で，重複は無いとする
- 整数の配列 $I[1..n]$ を用意する
  - $I[x] = j$ のとき，整数 x がヒープの $A[j]$ に格納されていることを表す
  - $I[x] = -1$ ならば x は格納されていないとする
  - 要素の移動を行うときは同時に I も更新する
  - $A[j] = x \Leftrightarrow I[x] = j$ が常に成り立つ(ように更新)

プライオリティキュー

- 要素の集合 S を保持するためのデータ構造
- 各要素はキーと呼ばれる値を持つ
- 次の操作をサポートする
  – INSERT(S,x): S に要素 x を追加する
  – MAXIMUM(S): 最大のキーを持つ S の要素を返す
  – EXTRACT_MAX(S): 最大のキーを持つ S の要素を削除し，その値を返す

クイックソートの確率的アルゴリズム

- クイックソートの平均的な場合の実行時間を解析する場合，入力の頻度を仮定する必要がある．
- 通常は，すべての順列が等確率で現れると仮定
- しかし実際にはこの仮定は必ずしも期待できない
- この仮定が成り立たなくてもうまく動作するクイックソートの確率的アルゴリズムを示す

確率的 (randomized) アルゴリズム

- 動作が入力だけでなく乱数発生器 (random-number generator)に依存するアルゴリズム
- 関数 RANDOM(a,b): a 以上 b 以下の整数を等確率で返すとする.
- プログラミング言語は擬似乱数発生器 (pseudorandom-number generator) を備える
- 擬似乱数: 統計的にはランダムに見えるが, 決定的に作られる数(の列)

確率的アルゴリズム1

- クイックソートを行う前に入力配列の要素をランダムに並び替える
- 実行時間の期待値は入力順序に依存しなくなる
- アルゴリズムがうまく動作しないのは，乱数発生器によって運の悪い順列を作る場合のみ
- 最悪の実行時間は改善されない $(\Theta(n^2))$
- 最悪の場合はほとんど起きない

確率的アルゴリズム2

- 配列を $A[p..r]$ を分割する前に，$A[p]$ と $A[p..r]$ からランダムに選んだ要素を交換
- pivotが $r-p+1$ 個の要素から等確率で選ばれることを保障する
- 分割が平均的にはバランスのとれたものになることが期待できる

```
int RANDOMIZED_PARTITION(data *A, int p, int r)
{
  int i;
  data tmp;
  i = RANDOM(p,r);
  tmp = A[i];  A[i] = A[p];   A[p] = tmp;    //   値の交換
  return PARTITION(A,p,r);
}

void RANDOMIZED_QUICKSORT(data *A, int p, int r)
{
  int q;
  if (p < r) {
      q = RANDOMIZED_PARTITION(A,p,r);
      RANDOMIZED_QUICKSORT(A,p,q);
      RANDOMIZED_QUICKSORT(A,q+1,r);
  }
}
```

最悪の場合の解析

- $T(n)$: サイズ n の入力に対するQUICKSORTの最悪実行時間

$$T(n) = \max_{1 \le q \le n-1} \bigl(T(q) + T(n-q)\bigr) + \Theta(n)$$

- $T(n) = O(n^2)$ を示す
- $m < n$ に対し $T(m) \le cm^2$ と仮定

$$
\begin{aligned}
T(n) &= \max_{1\le q\le n-1}\left(T(q)+T(n-q)\right)+\Theta(n) \\
&\le c\cdot \max_{1\le q\le n-1}\left(q^2+(n-q)^2\right)+\Theta(n) \\
&\le c\cdot\left(1^2+(n-1)^2\right)+\Theta(n) \\
&= cn^2-2c(n-1)+\Theta(n) \\
&\le cn^2
\end{aligned}
$$

c は $2c(n-1)$ が $\Theta(n)$ の項よりも大きくなるように十分大きくとる

よって $T(n) = O(n^2)$ が示された

平均的な場合の解析

- $T(n)$: サイズ n の入力に対するRANDOMIZED QUICKSORTの平均実行時間
- $T(n)$ に関する漸化式を解く
- 入力データはすべて異なる数とする

分割の解析

- RANDOMIZED_PARTITIONでPARTITIONが呼ばれるとき, $A[p]$ の要素は $A[p..r]$ のランダムな要素と置き換えられている.
- 観察:PARTITIONによって返される q の値は $A[p..r]$ の中での $x = A[p]$ のランクのみに依存
  – A の値は全て異なるので, $A[p..q]$ は x より小さく, $A[q+1..r]$ は x より大きい
- x のランク $\mathrm{rank}(x)$ = 集合中の x 以下の要素数
- 要素数 $n = r-p+1$
- $\mathrm{rank}(x) = i$ となる確率は $1/n$

- rank(x) = 1 のとき，PARTITIONでのループは $i = j = p$ で終了
- このとき $q = j = p$ つまり分割の小さい方のサイズは 1. この事象が起きる確率は $1/n$
- rank(x) ≥ 2 のとき，$x = A[p]$ より小さい値が少なくとも1つ存在
- PARTITIONでの最初のループ実行後は

  $i = p,\ j > p$
- $A[i]$ と $A[j]$ を入れ替えるため，$x = A[p]$ は右の分割に入る

- PARTITIONが停止したとき，左の分割には rank(x)–1 個の要素があり，それらは x より小さい
- rank(x) ≥ 2 のとき，左の分割のサイズが i である確率は $1/n$ (i = 1,2,...,n-1)
- rank(x) が1の場合と2以上の場合を合わせると，
- 左の分割のサイズ $r–p+1$ が
  - 1 になる確率: $2/n$
  - i になる確率: $1/n$ (i = 2,3,...,n-1)

平均時に対する漸化式

- $T(n)$: n 要素の入力配列をソートするのに必要な平均時間
- $T(1) = \Theta(1)$
- 長さ n の配列に対してソートする場合
  - 配列の分割: $\Theta(n)$ 時間
  - 統治: 長さ q と $n\!-\!q$ の部分配列を再帰的にソート

$$T(n) = \frac{1}{n}\left(T(1) + T(n-1) + \sum_{q=1}^{n-1} \left(T(q) + T(n-q)\right) \right) + \Theta(n)$$

$T(1)=\Theta(1)$, $T(n-1) = \mathrm{O}(n^2)$ より

$$\frac{1}{n}\Bigl(T(1)+T(n-1)\Bigr)=\frac{1}{n}\Bigl(\Theta(1)+\mathrm{O}(n^2)\Bigr)$$
$$=\mathrm{O}(n)$$

よって $T(n)$ は次のように書ける

$$T(n)=\frac{1}{n}\sum_{q=1}^{n-1}(T(q)+T(n-q))+\Theta(n)$$
$$=\frac{2}{n}\sum_{k=1}^{n-1}T(k)+\Theta(n)$$

$m < n$ に対し $T(m) \le am \lg m + b$ (a>0, b>0) と仮定

$$T(n) = \frac{2}{n}\sum_{k=1}^{n-1} T(k) + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=1}^{n-1} (ak \lg k + b) + \Theta(n)$$

$$= \frac{2a}{n}\sum_{k=1}^{n-1} k \lg k + \frac{2b}{n}(n-1) + \Theta(n)$$

$\displaystyle\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$ を用いる

$\dfrac{a}{4}n$ が $\Theta(n)+b$ 以上になるように a を選ぶと

$$
\begin{aligned}
T(n) &\le \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \frac{2b}{n}(n-1) + \Theta(n) \\
&\le an\lg n - \frac{a}{4}n + 2b + \Theta(n) \\
&= an\lg n + b + \left(\Theta(n) + b - \frac{a}{4}n\right) \\
&\le an\lg n + b
\end{aligned}
$$

$T(n)$ においても成り立つ
よってクイックソートの平均実行時間は $O(n \lg n)$

$\sum_{k=1}^{n-1} k \lg k \le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$ の証明

$$\sum_{k=1}^{n-1} k \lg k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg k$$

$$\le \sum_{k=1}^{\lceil n/2 \rceil - 1} k \lg \frac{n}{2} + \sum_{k=\lceil n/2 \rceil}^{n-1} k \lg n = (\lg n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \lg n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$= \lg n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\le \frac{1}{2} n(n-1) \lg n - \frac{1}{2}\left(\frac{n}{2} - 1\right)\frac{n}{2}$$

$$\le \frac{1}{2} n^2 \lg n - \frac{1}{8} n^2$$

ソーティングの下界

- ソーティングの入力: $\langle a_1, a_2, \ldots, a_n \rangle$
- 比較ソートでは要素間の比較のみを用いてソートを行う
- 2つの要素 a_i, a_j が与えられたとき，それらの相対的な順序を決定するためにテストを行う
  - $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, $a_i > a_j$ のみ
- これ以外の方法では要素のテストはできない

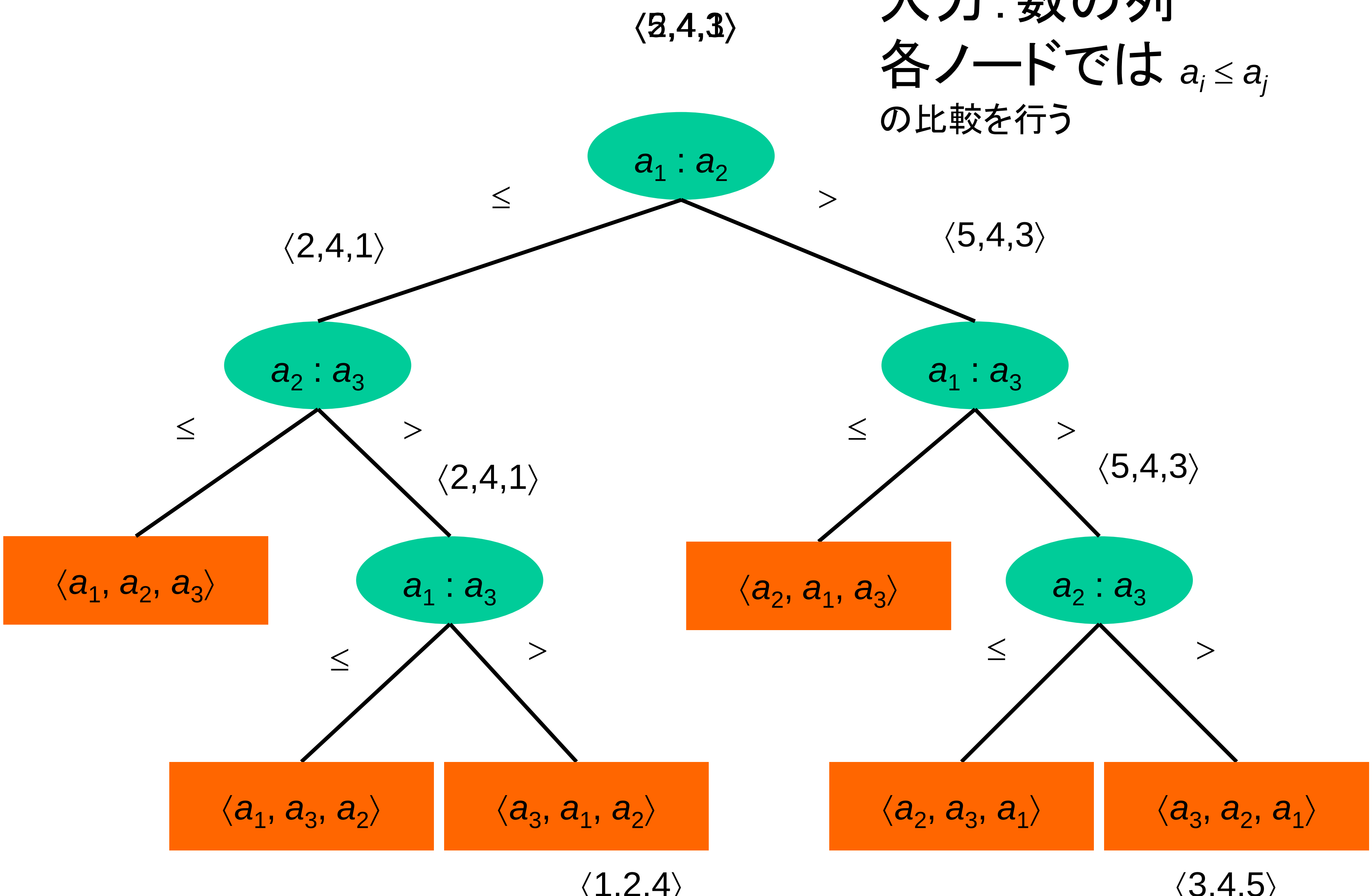
仮定

- すべての入力要素は異なると仮定する
  - $a_i = a_j$ という比較は行わないと仮定できる
- $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$ は全て等価
- 全ての比較は $a_i \leq a_j$ という形と仮定できる

決定木モデル

- 比較ソートは決定木 (decision tree) とみなせる
- 決定木はソーティングアルゴリズムで実行される比較を表現している
- アルゴリズム中における制御，データの移動などの要因は無視する

入力: 数の列
各ノードでは $a_i \le a_j$
の比較を行う



葉は入力の置換に対応

決定木の高さと比較回数

- 決定木はソートアルゴリズム A から決まる
- 入力数列を与えると決定木の対応する葉が決まる
- 根から葉までのパスの長さ
  ＝Aを実行する際の比較回数
- 根から葉までのパス長の最大値
  ＝実行されるソートアルゴリズムの最悪比較回数
- 比較ソートでの最悪の比較回数は決定木の高さに対応

最悪時の下界

- 決定木の高さの下界＝任意の比較ソートアルゴリズムの実行時間の下界

定理1　n 要素をソートするどんな決定木の高さも $\Omega(n \lg n)$

証明 n 要素をソートする高さ h の決定木を考える.

ソートを正しく行うには, n 要素の $n!$ 通りの置換全てが葉に現れなければならない.

高さ h の2分木の葉の数は高々 2^h. よって

$n! > 2^h$ ではソートできない. つまり $h \geq \lg(n!)$

$$h \geq \lg(n!)$$

Stirlingの公式より $\sqrt{2\pi n}\left(\frac{n}{e}\right)^n \leq n! \leq e\sqrt{n}\left(\frac{n}{e}\right)^n$

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \lg\left(\frac{n}{e}\right)^n$$

$$= n\lg n - n\lg e$$

$$= \Omega(n\lg n)$$

系 1　ヒープソートとマージソートは漸近的に最適な比較ソートである

証明　これらの実行時間の上界 $O(n \lg n)$ は定理 1 の最悪時の下界 $\Omega(n \lg n)$ と一致する.