算法数理工学
第2回

定兼　邦彦

マージソート

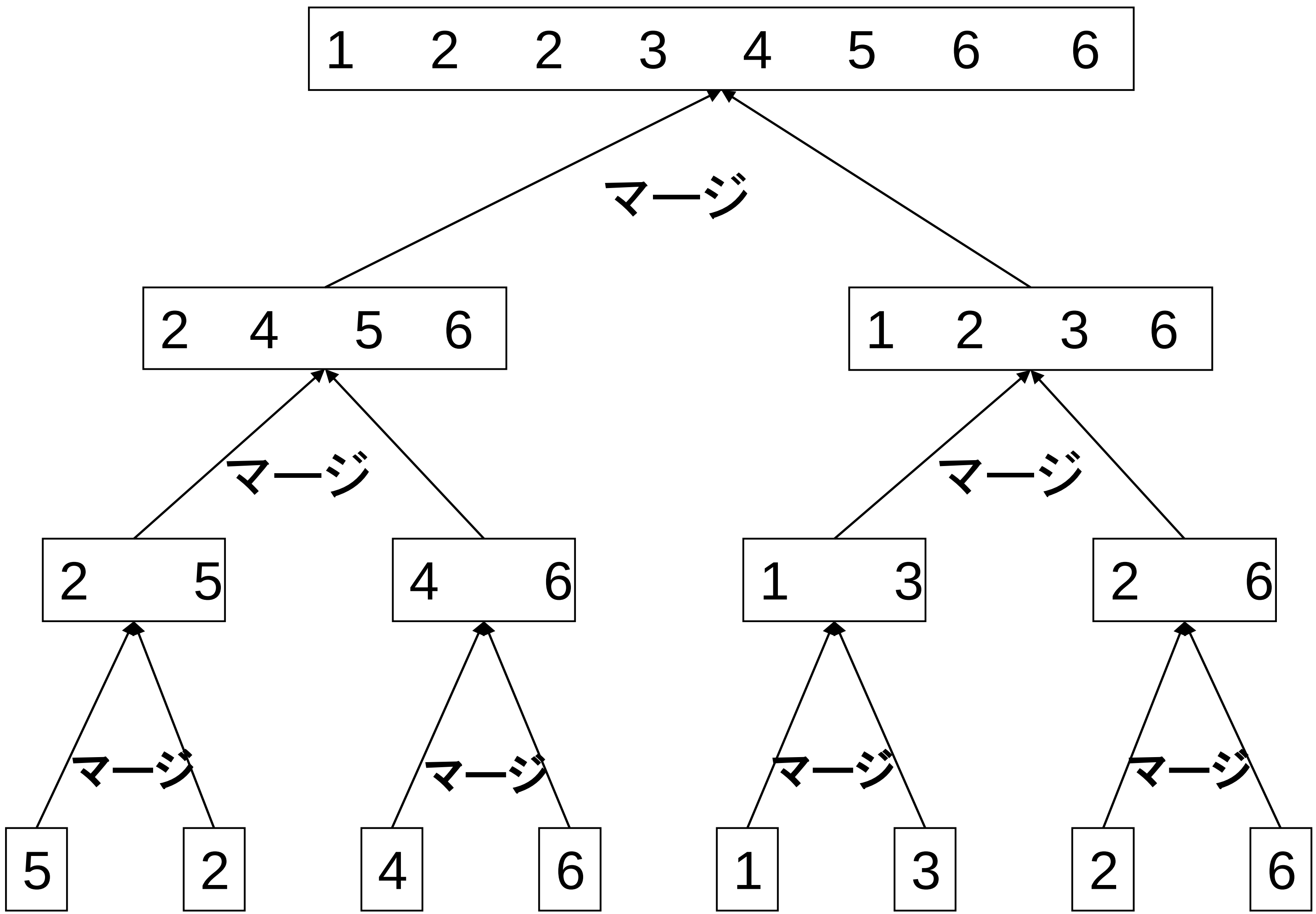
```
void MERGE_SORT(data *A, int p, int r, data *B)     // A[p..r] をソート
{
  int q;

  if (p < r) {                                // p==r ならソートする必要なし
    q = (p+r)/2;
    MERGE_SORT(A, p, q, B);   //  A[p..q] を再帰的にソート
    MERGE_SORT(A, q+1, r, B); //  A[q+1..r] を再帰的にソート
    MERGE(A, p, q, r, B);  // ソートされた部分列 A[p..q] と A[q+1..r] を統合
  }
}
```

マージ

- 一時的な配列 $B[0,n\text{-}1]$ を用いる

```
void MERGE(data *A, int p, int q, int r, data *B)
{     // ソートされた部分列 A[p..q] と A[q+1..r] を統合
  int i,j,k;
  data t;

  k = p;   i = p;   j = q+1;
  while (k <= r) {
    if (j > r) t = A[i++];              // 前半のみにデータがある
    else if (i > q)  t = A[j++];        // 後半のみにデータがある
    else if (A[i] <= A[j]) t = A[i++];  // 前半のほうが小さい
    else t = A[j++];                    //  後半のほうが小さい
    B[k++] = t;                         // 一時的な配列に保存
  }
  for (i=p; i<=r; i++) A[i] = B[i];  // 元の配列に書き戻す
}
```

```
void MERGE_SORT(data *A, int p, int r, data *B)     // A[p..r] をソート
{
  int q;
  if (p < r) {              // p==r ならソートする必要なし
    q = (p+r)/2;
    MERGE_SORT(A, p, q, B);   //  A[p..q] を再帰的にソート
    MERGE_SORT(A, q+1, r, B); //  A[q+1..r] を再帰的にソート
    MERGE(A, p, q, r, B);  // ソートされた部分列 A[p..q] と A[q+1..r] を統合
  }
}

int main(int argc, char *argv[])
{
  data A[14] = {27,17,3,16,13,10,1,5,7,12,4,8,9,0};
  data B[14];
  int i,n;

  n = 14;
  MERGE_SORT(A,0,n-1,B);
  for (i=0;i<n;i++) printf("%d ",A[i]);
  printf("¥n");

}
```


1 2 2 3 4 5 6 6
マージ
2 4 5 6
1 2 3 6
マージ
マージ
2 5
4 6
1 3
2 6
マージ
マージ
マージ
マージ
5
2
4
6
1
3
2
6

問題の計算量

- アルゴリズムの計算量ではなく, 問題の計算量（計算複雑度）という概念がある
- ある問題の計算量が O(f(n)) とは
  - その問題を解くあるアルゴリズム A が存在し, 任意の入力に対する計算時間が O(f(n)) である.
  - 例：ソーティングの計算量は O(n log n)
    (マージソートはどんな入力でも O(n log n) 時間だから)

- ある問題の計算量が $\Omega(f(n))$ とは
  - その問題に対するどんなアルゴリズムにも，計算時間が $\Omega(f(n))$ となる入力が存在する
  - 例：比較ソートの計算量は $\Omega(n \log n)$
    証明は後日行う
- ある問題の計算量が $\Theta(f(n))$ とは
  - $O(f(n))$ かつ $\Omega(f(n))$
  - 例：比較ソートの計算量は $\Theta(n \log n)$
  - つまりマージソートよりも漸近的に良い（オーダの低い）ソートアルゴリズムは存在しない．
- 計算量の議論をする場合は，計算モデルを明確にする必要がある

クイックソート

- n 個の数に対して最悪実行時間$\Theta(n^2)$のソーティングアルゴリズム
- 平均実行時間は$\Theta(n \log n)$
- Θ記法に隠された定数も小さい
- in-place (一時的な配列が必要ない)

クイックソートの記述

- 分割統治法に基づく
- 部分配列 $A[p..r]$ のソーティング

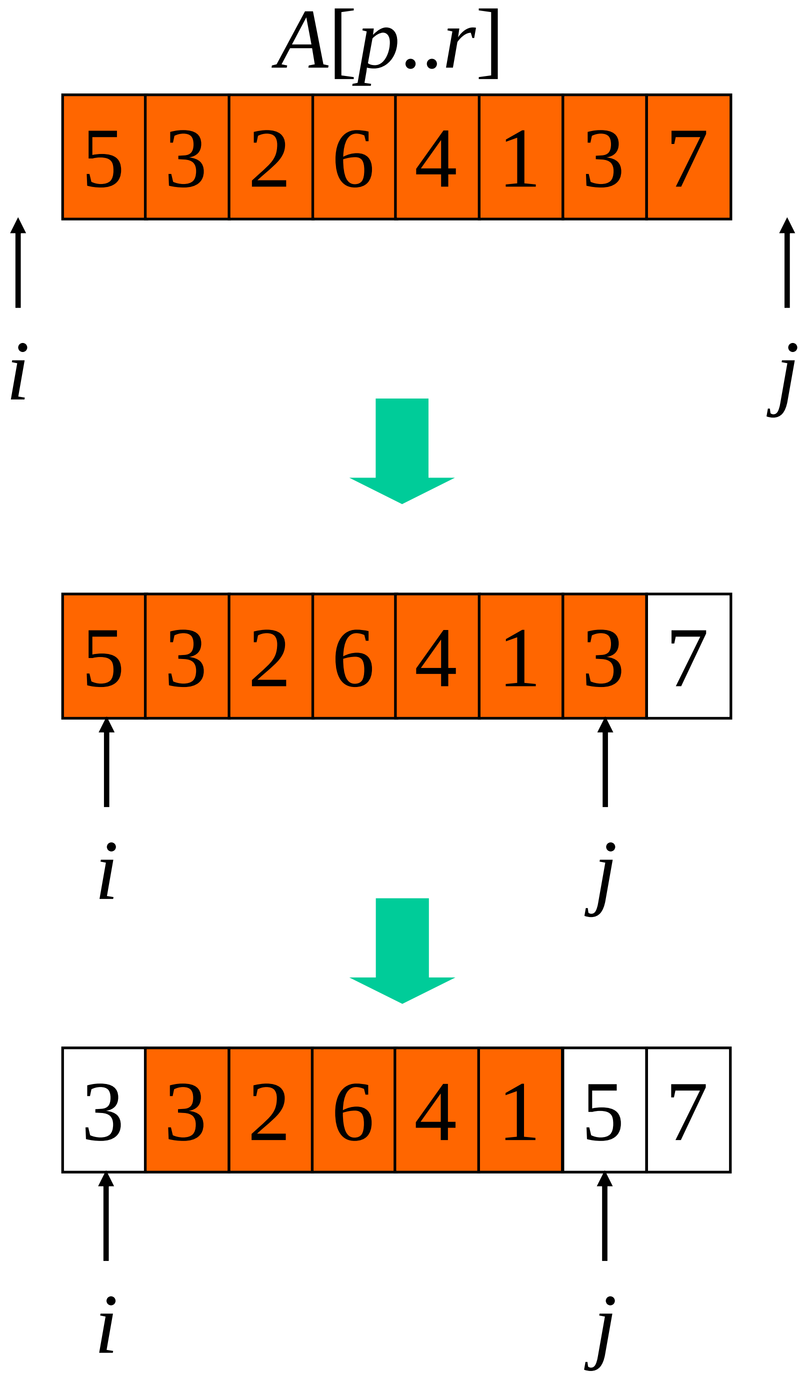
1. 部分問題への分割:
   - 配列 $A[p..r]$ を2つの部分配列 $A[p..q]$ と $A[q+1..r]$ に再配置する.
   - $A[p..q]$ のどの要素も $A[q+1..r]$ の要素以下にする.
   - 添字 q はこの分割手続きの中で計算する.
2. 部分問題の解決 (統治):
   - 部分配列 $A[p..q]$ と $A[q+1..r]$ を再帰的にソート
3. 部分問題の統合
   - $A[p..r]$ はすでにソートされているので何もしない

クイックソートのコード

```
void QUICKSORT(data *A, int p, int r)
{
   int q;

   if (p < r) {
        q = PARTITION(A,p,r);
        QUICKSORT(A,p,q);
        QUICKSORT(A,q+1,r);
   }
}

main()
{
     QUICKSORT(A,0,n-1);
}
```

配列の分割

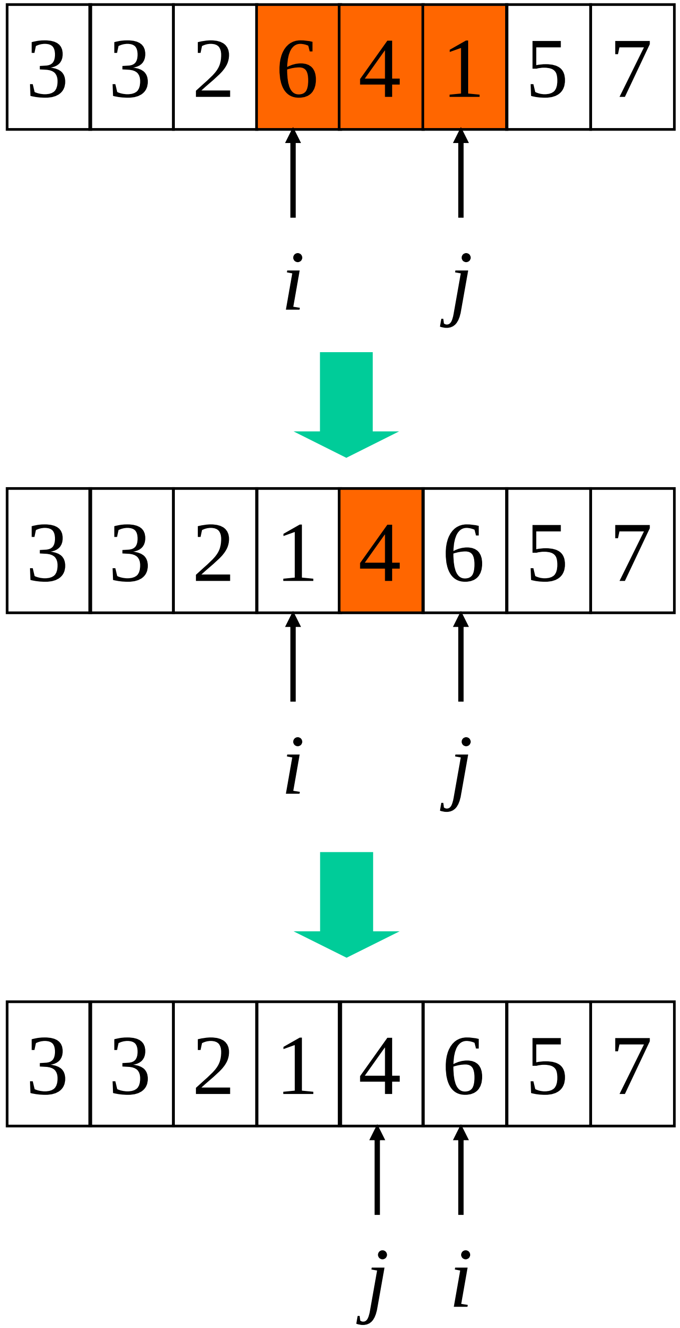
```
int PARTITION(data *A, int p, int r)    //  O(n) 時間
{
   int q, i, j;
   data x, tmp;
   x = A[p];
   i = p-1;   j = r+1;
   while (1) {
       do {j = j-1;} while (A[j] > x);
       do {i = i+1;} while (A[i] < x);
       if (i < j) {
           tmp = A[i];  A[i] = A[j];   A[j] = tmp;
       } else {
           return j;
       }
   }
}
```



初期状態:
i と j は配列の範囲外
$x = A[p] = 5$ によって分割
x: 枢軸 (pivot) と呼ぶ

$A[i] \geq x$
$A[j] \leq x$ となる最初の i, j
7 は正しい分割位置にある

$A[i]$ と $A[j]$ を交換
正しい分割位置になる



$A[i] \geq x$

$A[j] \leq x$ となる最初の i, j

$A[i]$ と $A[j]$ を交換

$i \geq j$ となったので

$q = j$ を返す

$A[p..q]$ は x 以下

$A[q+1..r]$ は x 以上

PARTITIONの正当性

- PARTITIONの返り値を q とする
- $A[p..r]$ の分割後に満たされるべき条件
  - $A[p..q]$ はある pivot x 以下
  - $A[q+1..r]$ は x 以上
  - $p \leq q < r$
- $q = r$ となると，QUICKSORTは停止しないため，どんな A に対しても $q < r$ となることを保障する必要がある

- 初期状態は $i < j$
- do {j = j-1;} while (A[j] > x);

  do {i = i+1;} while (A[i] < x); の終了後
  - $p \le i, j \le r$
  - A[j+1..r] は x 以上
  - A[p..i-1] は x 以下
  - $A[i] \ge x \ge A[j]$
- $i < j$ のとき，A[i] と A[j] を交換すると
  - A[j..r] は x 以上
  - A[p..i] は x 以下
- $i \ge j$ のとき $q = j$

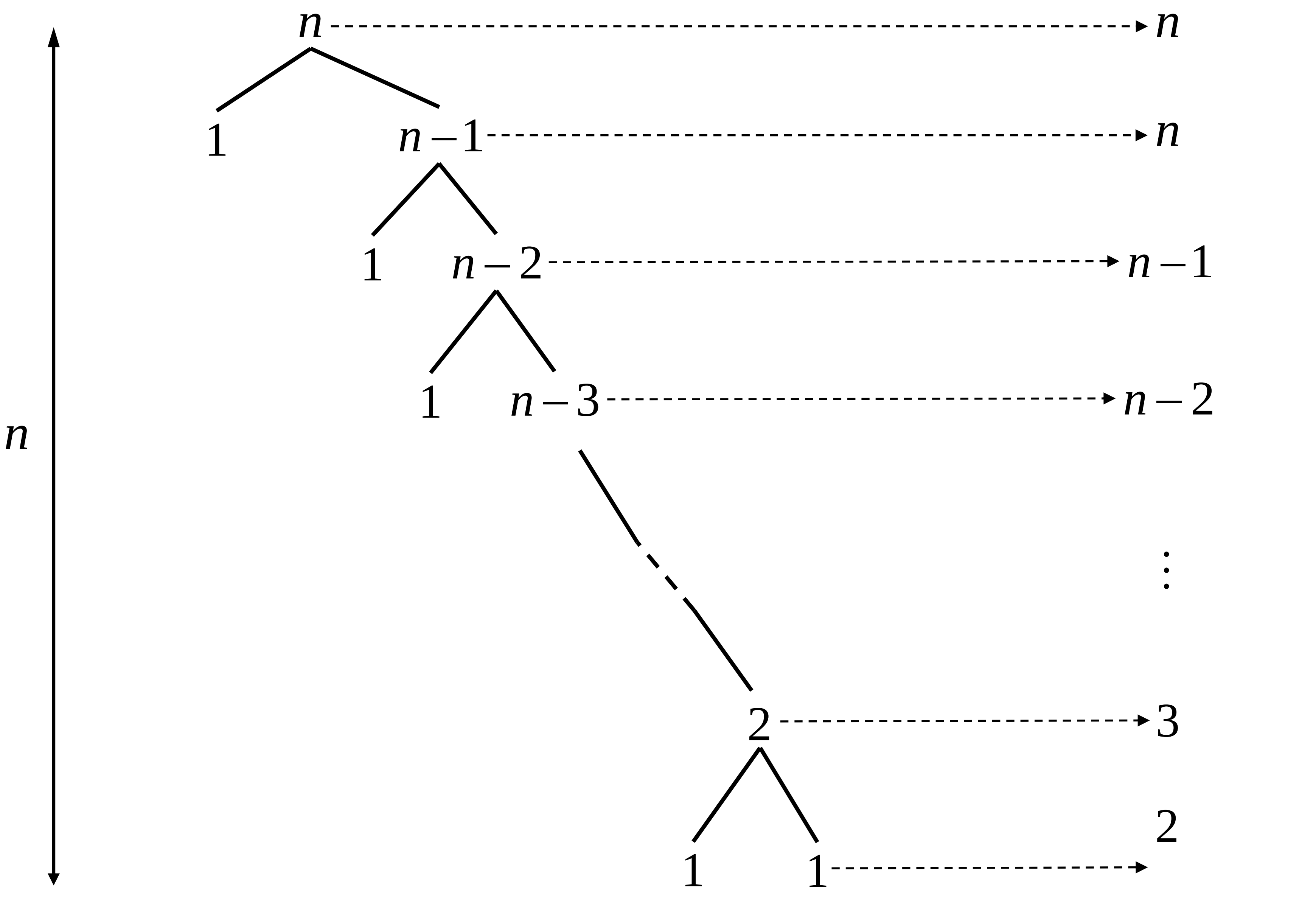
- PARTITIONの終了時に $q = j = r$ とする
  - while (1) のループを実行する毎に j は1以上減る
  - 終了時に $j = r$ ならばこのループは1度しか実行されていない
  - しかし1回目のループでは $x = A[p]$ なので $i = p$
- PARTITIONは $p < r$ の場合のみ呼ばれるので，1回目のループでは $p = i < j = r$
- つまり1回目のループでは終了しない
- よってPARTITION終了時に $q = r$ とはならない．つまり $q < r$

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平均化されているかどうかに依存
- これはpivotの選択に依存
- 分割が平均化されていれば実行時間は漸近的にマージソートと同じ ($\Theta(n \log n)$)
- 最悪時は $\Theta(n^2)$

最悪の分割

- 最悪の場合は，PARTITIONによって領域が 1 要素と n-1 要素に分けられる場合
- 分割には $\Theta(n)$ 時間かかる
- 実行時間に対する漸化式は
  - $T(n) = T(n-1) + \Theta(n)$, $T(1) = \Theta(1)$
- $T(n) = \Theta(n^2)$
- 最悪実行時間は挿入ソートと同じ
- 入力が完全にソートされているときに起こる (挿入ソートなら $O(n)$ 時間)

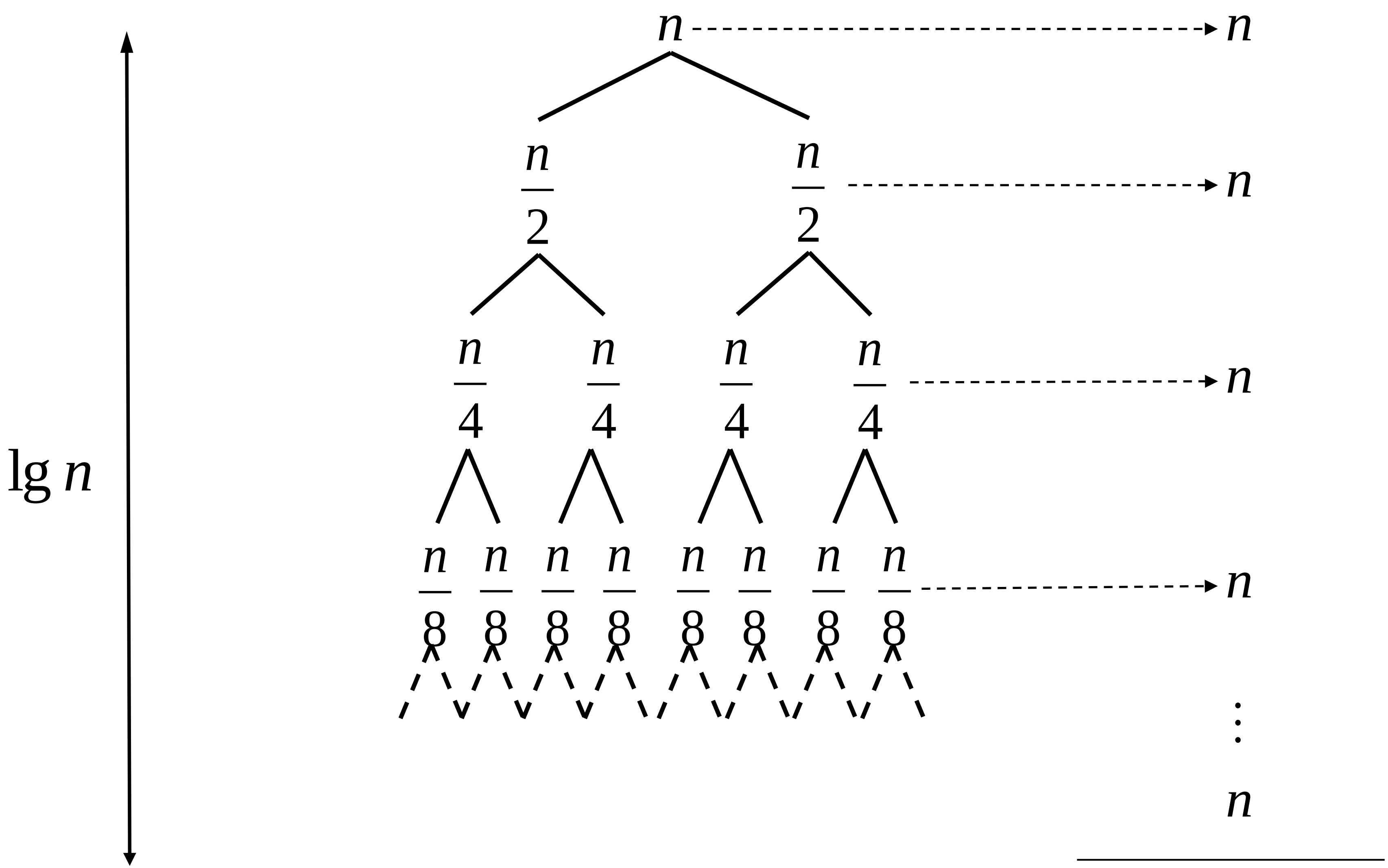


再帰木

$$\mathrm{Total}:\Theta\left(n^2\right)$$

最良の分割

- クイックソートが最も速く動作するのは，PARTITIONによってサイズ $n/2$ の2つの領域に分割されるとき.
- $T(n) = 2T(n/2) + \Theta(n)$
- $T(n) = \Theta(n \lg n)$
- ちょうど半分に分割される場合が最速



$$\text{Total}: \Theta(n \lg n)$$

平衡分割

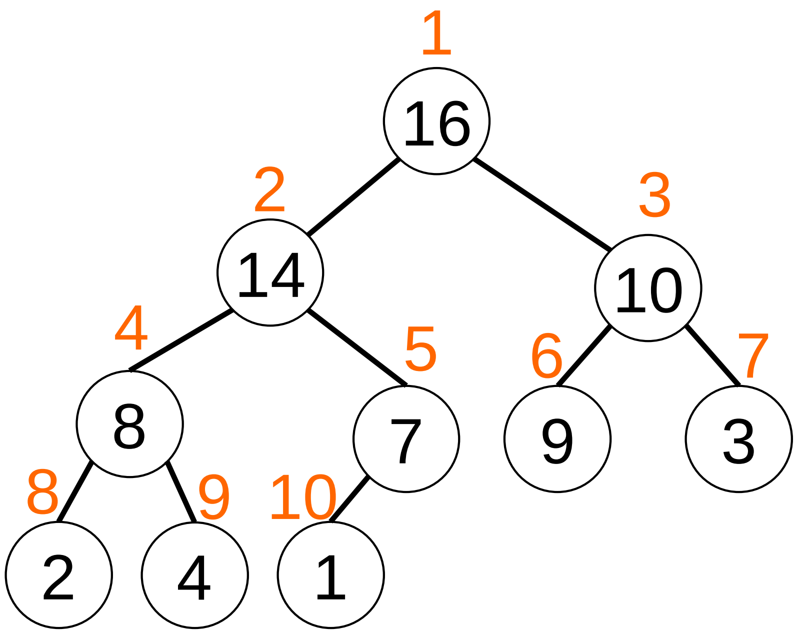
- PARTITIONで常に 9 対 1 の比で分割されると仮定する
- $T(n) = T(9n/10) + T(n/10) + \Theta(n)$
- 再帰木の各レベルのコストは $O(n)$
- 再帰の深さは $\log_{\frac{10}{9}} n = \Theta(\lg n)$
- 漸近的には中央で分割するのと同じ
- 分割の比が 99 対 1 でも同じ (定数比なら良い)

ヒープソート

- O(n lg n) 時間アルゴリズム, in-place
- ヒープ (heap) と呼ばれるデータ構造を用いる
- ヒープはプライオリティキュー (priority queue) を効率よく実現する

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



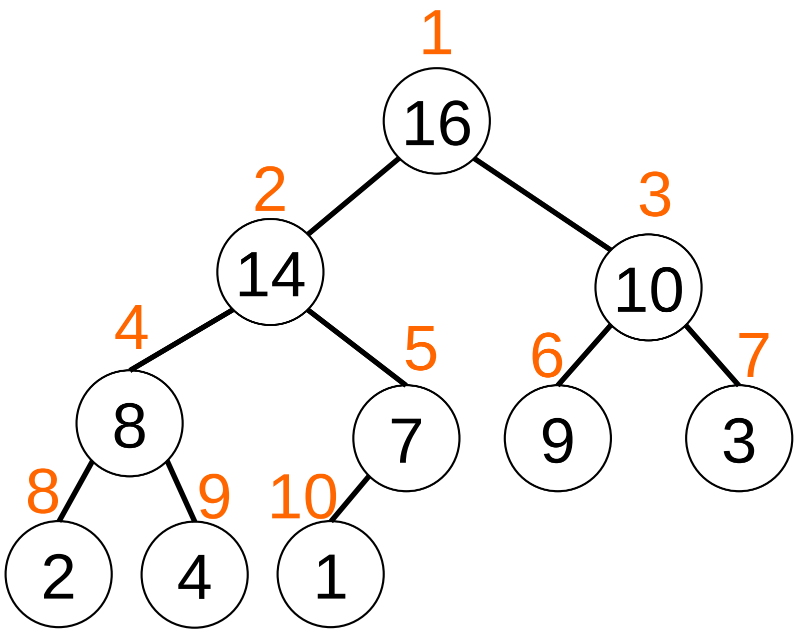
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

ヒープを表す構造体

```
typedef struct {
    int length;             // 配列 A に格納できる最大要素数
    int heap_size;          // ヒープに格納されている要素の数
    data *A;                // 要素を格納する配列
} HEAP;
```

- ヒープに後から要素を追加する場合があるとき，配列 A は大きいものを確保しておく

- length(H): 配列 A に格納できる最大要素数
- heap_size(H): H に格納されているヒープの要素数
- heap_size(H) $\leq$ length(H)
- 木の根: A[1]
- 節点の添え字が i のとき
  - 親 PARENT(i) = $\lfloor i / 2 \rfloor$
  - 左の子 LEFT(i) = $2\,i$
  - 右の子 RIGHT(i) = $2\,i + 1$
- 木の高さは $\Theta(\lg n)$



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

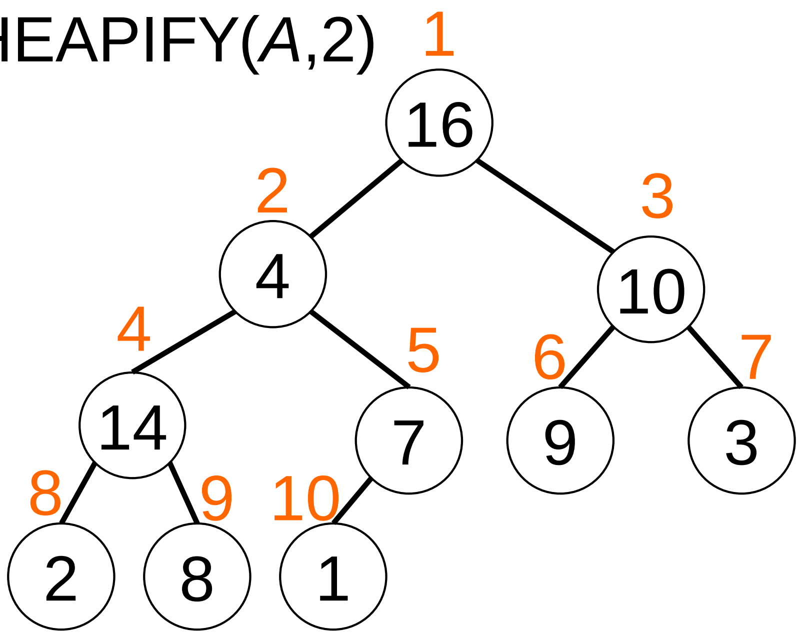
- HEAPIFY: ヒープ条件を保持する.
- BUILD_HEAP: 入力の配列からヒープを構成する. 線形時間.
- HEAPSORT: 配列をソートする. O(n lg n) 時間.
- EXTRACT_MAX: ヒープの最大値を取り出す. O(lg n) 時間.
- INSERT: ヒープに値を追加する. O(lg n) 時間.

ヒープ条件の保持

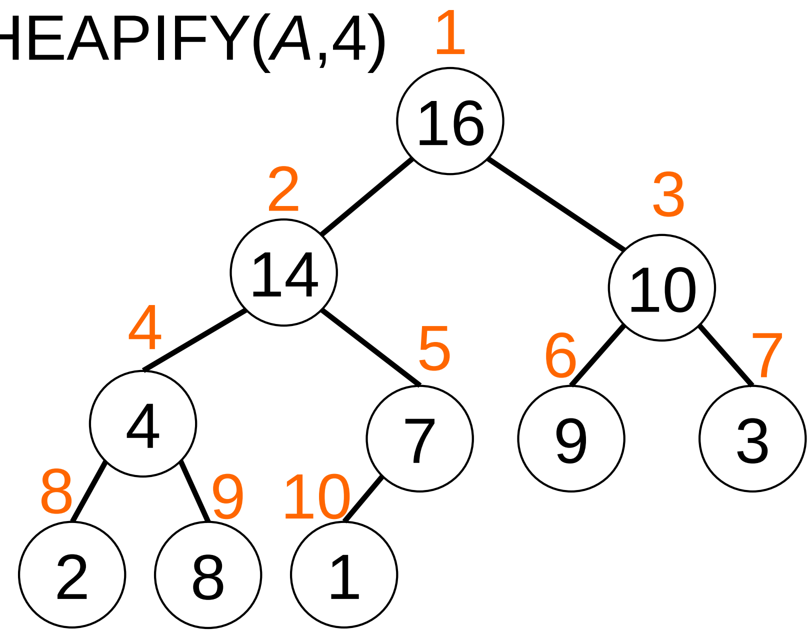
- HEAPIFY(H,i): $A[i]$ を根とする部分木がヒープになるようにする. ただし LEFT(i) と RIGHT(i) を根とする2分木はヒープと仮定.

```
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);
  }
}
```

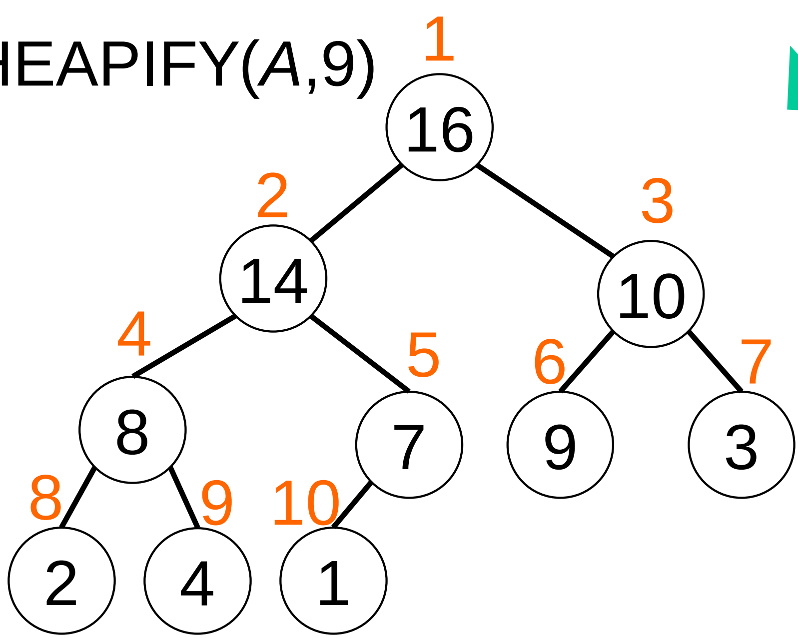
HEAPIFY(A,2)



HEAPIFY(A,4)



HEAPIFY(A,9)



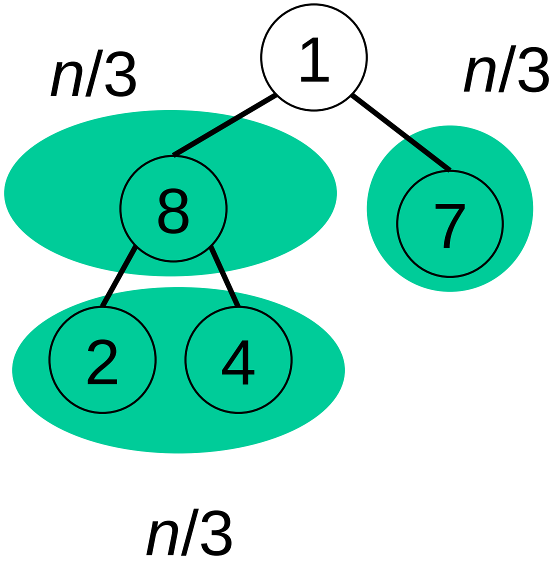
正当性の証明

- HEAPIFY(H,i) を行うと
- $A[i]$ を根とする部分木がヒープなら何もしない
- ヒープでなければ
  - $A[i]$, 左の子, 右の子の中で左の子が最大とする
  - 左の子と $A[i]$ を入れ替える
  - 右の子の親は値が大きくなっているので, 右の子ではヒープ条件を満たす
  - $A[i]$ は部分木中の最大値を格納
  - 左の子はヒープ条件を満たしていない可能性があるので, 1つ下に降りて繰り返す
  - $\log n$ 回で終了

HEAPIFYの実行時間

- 節点 i を根とする，サイズ n の部分木に対するHEAPIFYの実行時間 $T(n)$
  - 部分木のサイズは $2n/3$ 以下
  - $T(n) \leq T(2n/3) + \Theta(1)$
  - $T(n) = O(\lg n)$
- 高さ h の節点での実行時間は

  $O(h)$

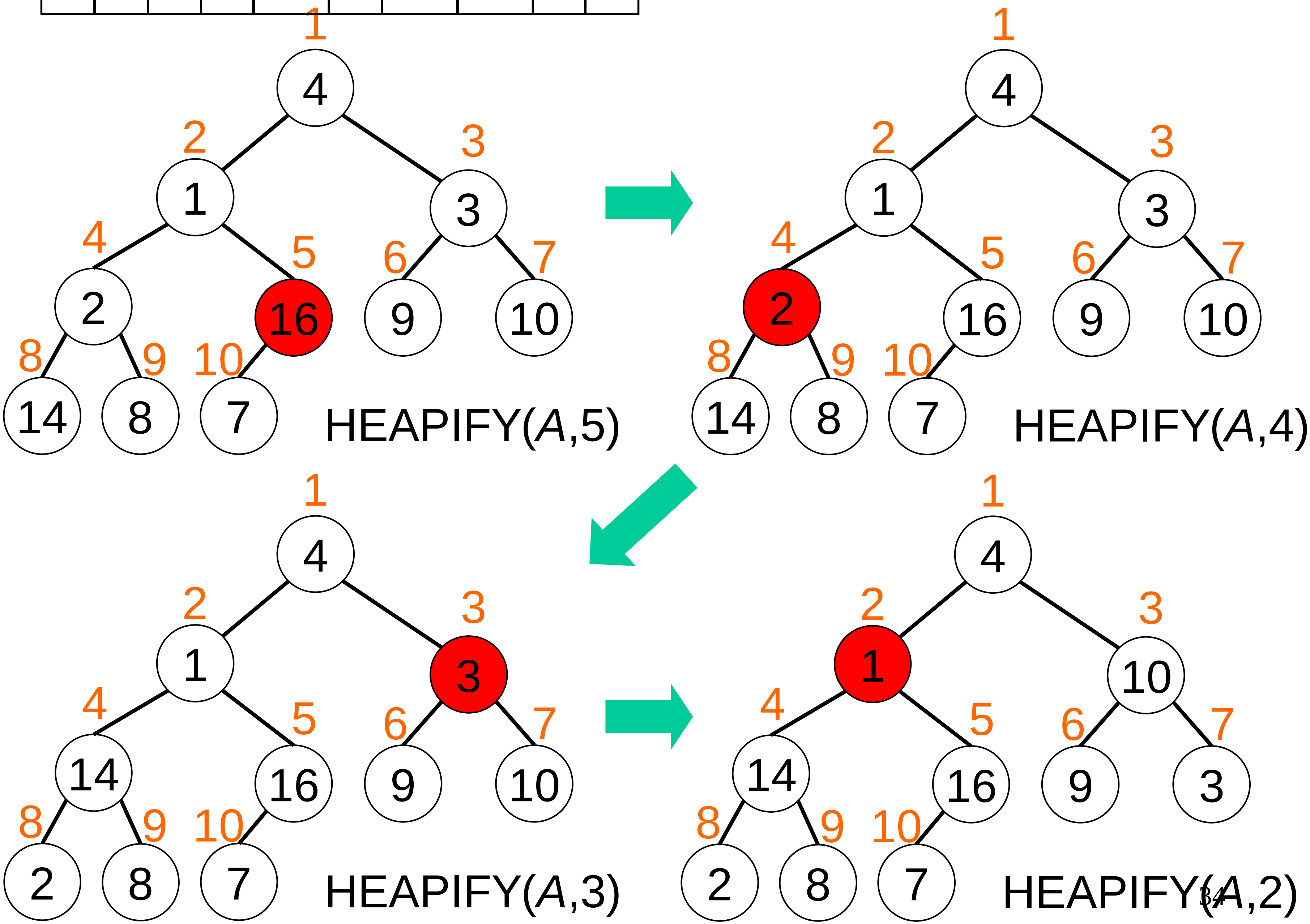


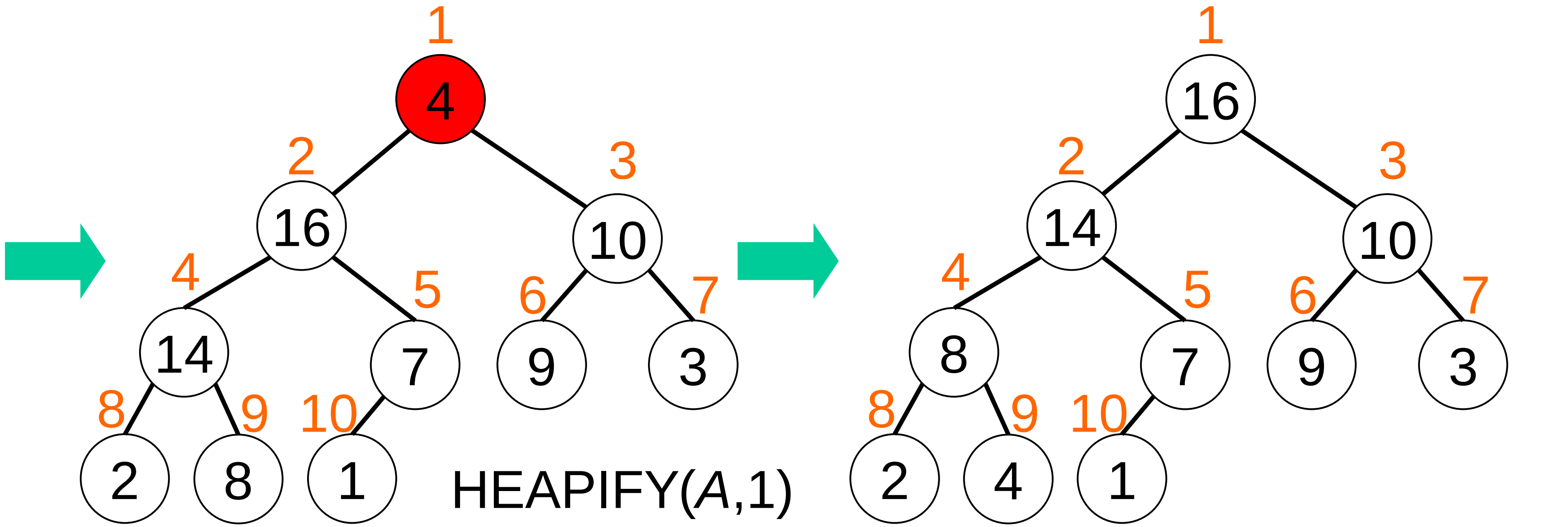
ヒープの構成

- HEAPIFYでは左右の部分木がヒープである必要がある
- 全体をヒープにするには，2分木の葉の方からヒープにしていけばいい

```
void BUILD_HEAP(HEAP *H, int n, data *A, int length)
{
  int i;
  H->length = length;
  H->heap_size = n;
  H->A = A;

  for (i = n/2; i >= 1; i--) {
    HEAPIFY(H,i);
  }
}
```

A	4	1	3	2	16	9	10	14	8	7





HEAPIFY(A,1)

計算量の解析

- O(lg n) 時間のHEAPIFYが O(n) 回
  ⇒O(n lg n)時間　(注: これはタイトではない)
- 高さ h の節点でのHEAPIFYは O(h) 時間
- n 要素のヒープ内に高さ h の節点は高々$\lceil n/2^{h+1} \rceil$個

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot \mathrm{O}(h) = \mathrm{O}\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = \mathrm{O}(n)$$

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$ を用いる

最大値の削除

- 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

```
data EXTRACT_MAX(HEAP *H)  // O(lg n) 時間
{
  data MAX, *A;

  A = H->A;

  if (H->heap_size < 1) {
    printf("ERROR ヒープのアンダーフロー¥n");
    exit(1);
  }
  MAX = A[1];
  A[1] = A[H->heap_size]; // ヒープの最後の値を根に移動する
  H->heap_size = H->heap_size - 1;
  HEAPIFY(H,1); // ヒープを作り直す
  return MAX;
}
```

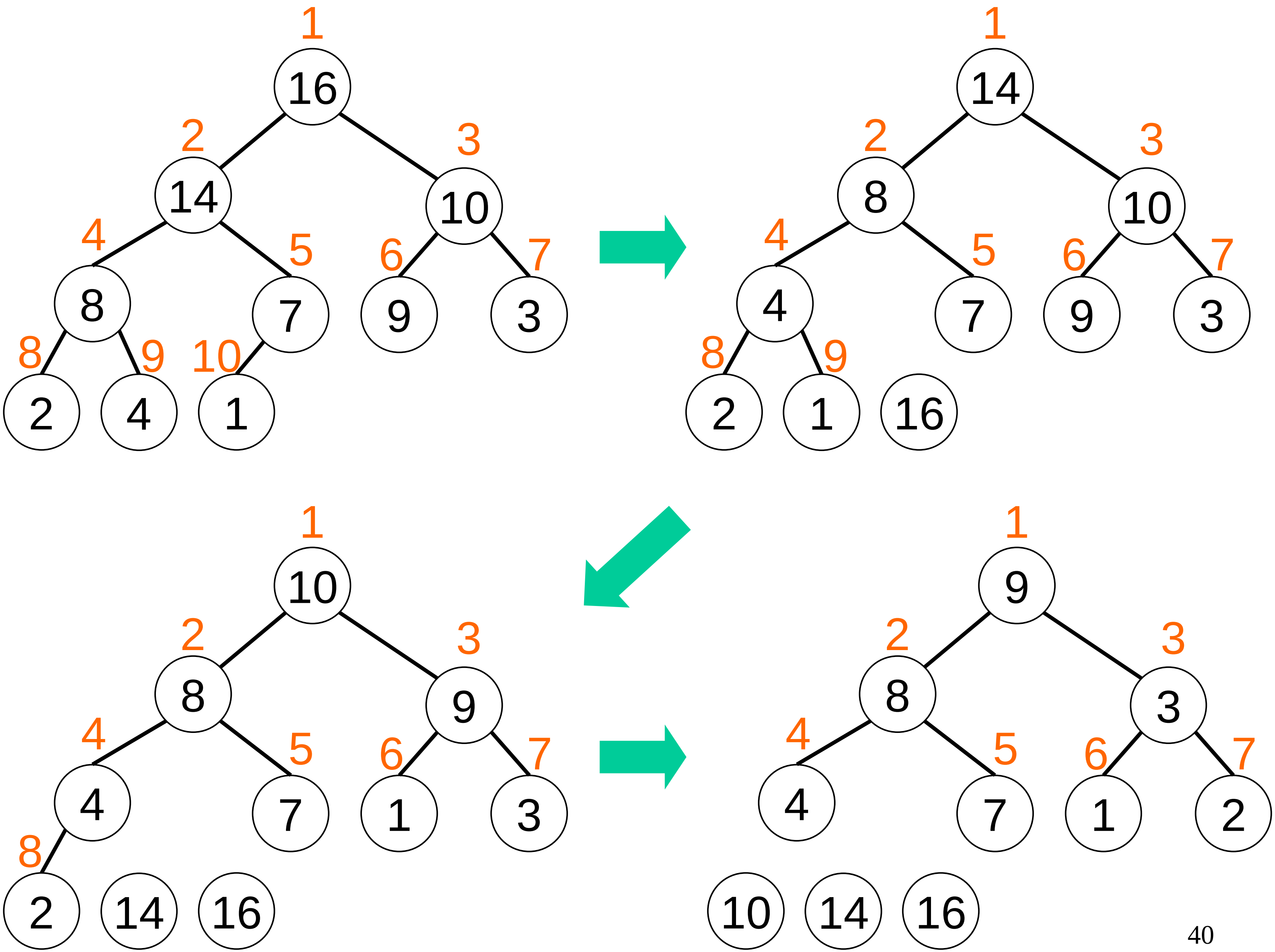
正当性の証明

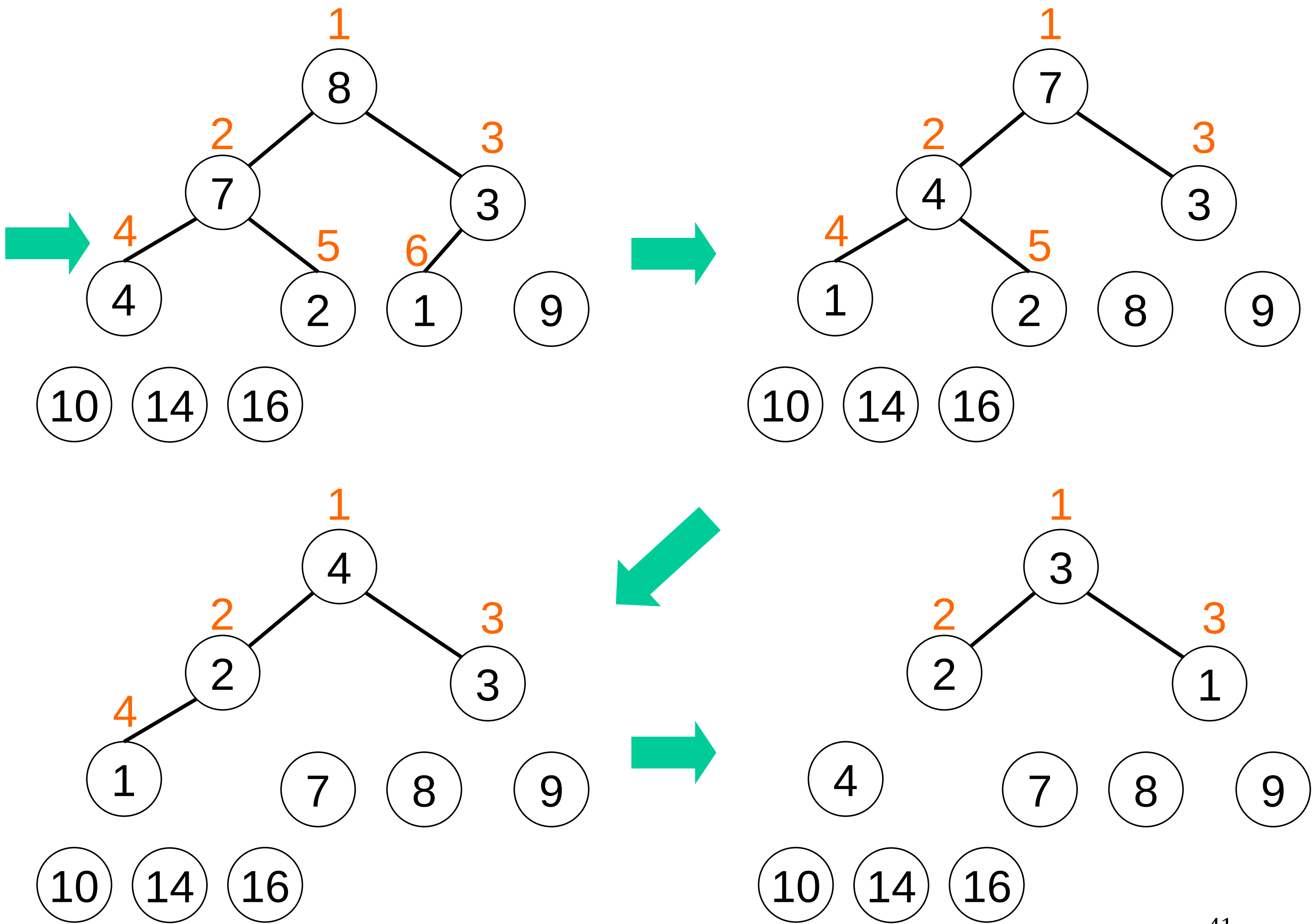
- 削除前
  - 根には最大値が入っている
  - 根も，左右の子もヒープになっている
- 削除後
  - 最後の要素が根に入っている
  - 根はヒープ条件を満たしていない可能性がある
  - 根の左右の子はヒープになっている
  - 根でHEAPIFYを行えば全体がヒープになる

ヒープソート

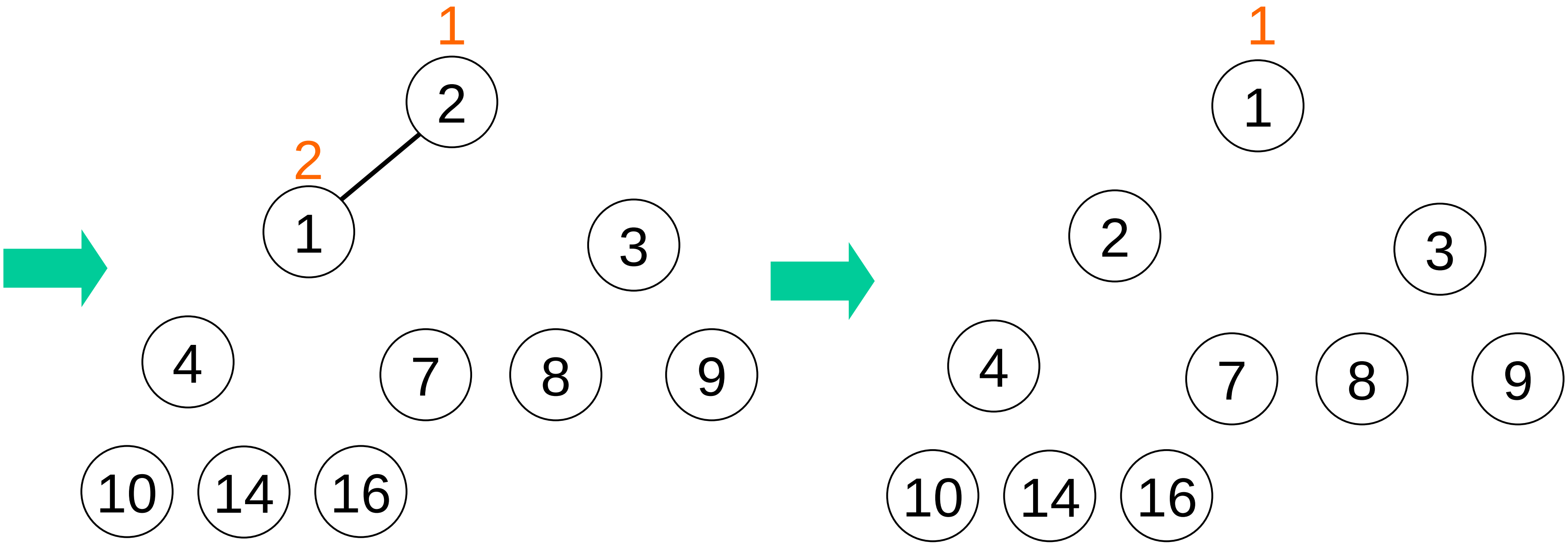
- まずヒープを作る
- すると最大要素が $A[1]$ に入る
- $A[1]$と$A[n]$を交換すると，最大要素が$A[n]$に入る
- ヒープのサイズを1つ減らしてヒープを維持する

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