C++プログラミングIII

第13回: ヒープソート

渡邉和宏

本日の講義内容

1. 二分木とヒープ
2. ヒープソートのアルゴリズム
3. ヒープソート実装の準備
4. 課題演習

ソートとは(再掲) (1/2)

- キーとなる項目(数値など)の値の大小関係に基づき, データの集合を一定の順序で並べ替える作業
  - 昇順(Ascending Order): 小さいデータから大きいデータの順に並べる
  - 降順(Descending Order): 昇順の逆(大→小)
- ソートアルゴリズム
  - 選択ソート, バブルソート, 挿入ソート, クイックソート, マージソート, ヒープソート等
- 計算効率, 並列化の容易さ, 使用メモリ, 適応制限等はアルゴリズムに依存

ソートとは (再掲) (2/2)

種類	平均計算量	安定性 (※)	講義回
選択ソート	$O(n^2)$	安定ではない	第11回
バブルソート	$O(n^2)$	安定	第11回
クイックソート	$O(n \log(n))$	安定ではない	第11回
マージソート	$O(n \log(n))$	安定	第12回
ヒープソート	$O(n \log(n))$	安定ではない	第13回

高速なソートアルゴリズム

(※)キーが同じである要素が2つ以上存在するデータをソートした場合,ソート前とソート後でそれらの要素の順番が変わらないようなソートを安定と呼ぶ.

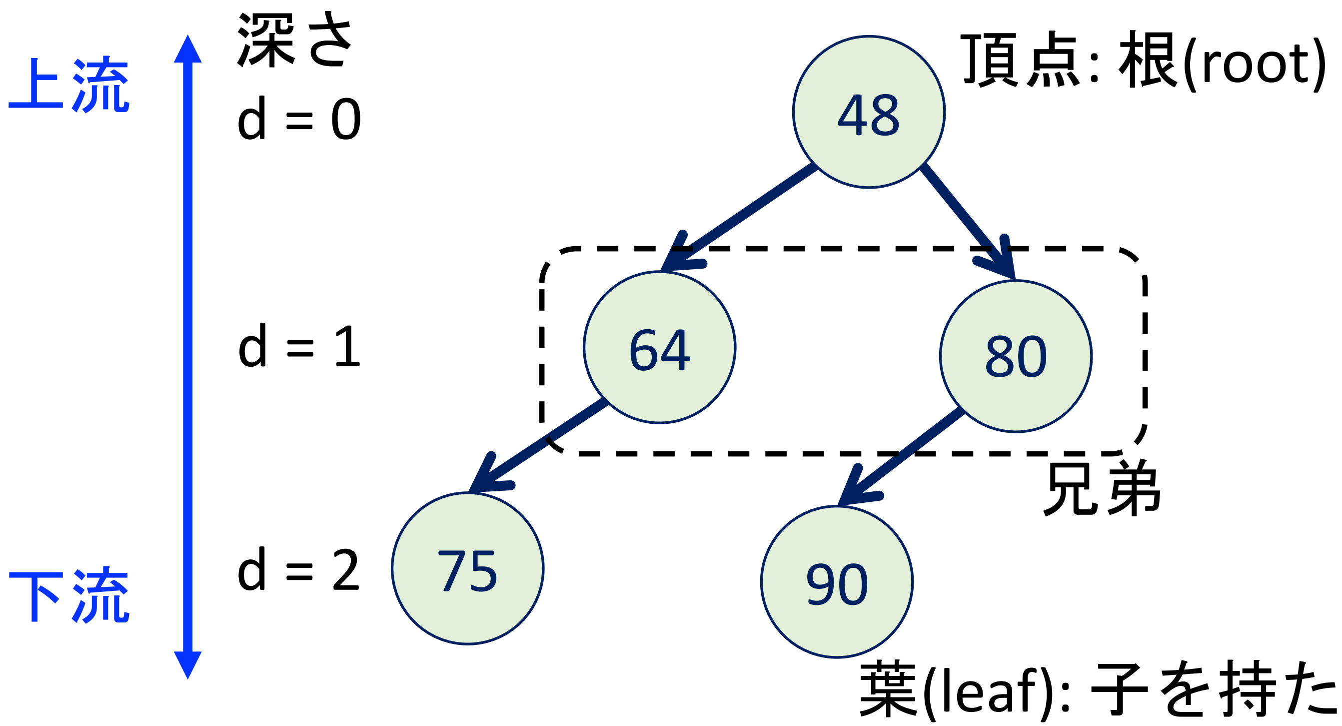
高速なソートアルゴリズムの主な違い

- クイックソート
  - ✓ 一番高速. 作業用メモリは不要.
  - ✕ 安定ではない. 大規模配列には非効率.
- マージソート
  - ✓ 並列化しやすい. 安定なソート. 大規模データでも動作する.
  - ✕ 作業用メモリ消費量が大きい. クイックソートと比べ遅い.
- ヒープソート
  - ✓ クイックソートと比べ, データの内容で性能が上下しない. 作業用メモリは不要.
  - ✕ 安定ではない. マージソートと比べ遅い.

1. 二分木とヒープ

二分木再訪 (第8回講義参照)

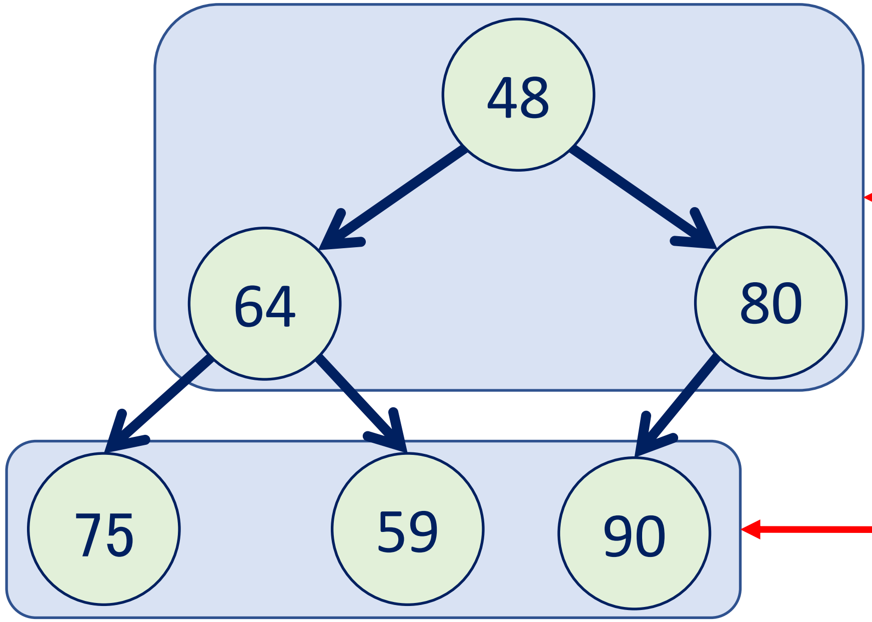
- 各ノードが左右の子のノードへの分岐を持つ木構造を二分木(binary tree)と呼ぶ.
- 片方, もしくは両方の子を持たないノードもある (葉).



各ノードは左右の部分木へのポインタを持つ.

完全二分木

- 根から下流の階層へのノードが空くことなく詰まっており, かつ同一階層内では左から右へのノードが空くことなく詰まっている二分木を完全二分木と呼ぶ.



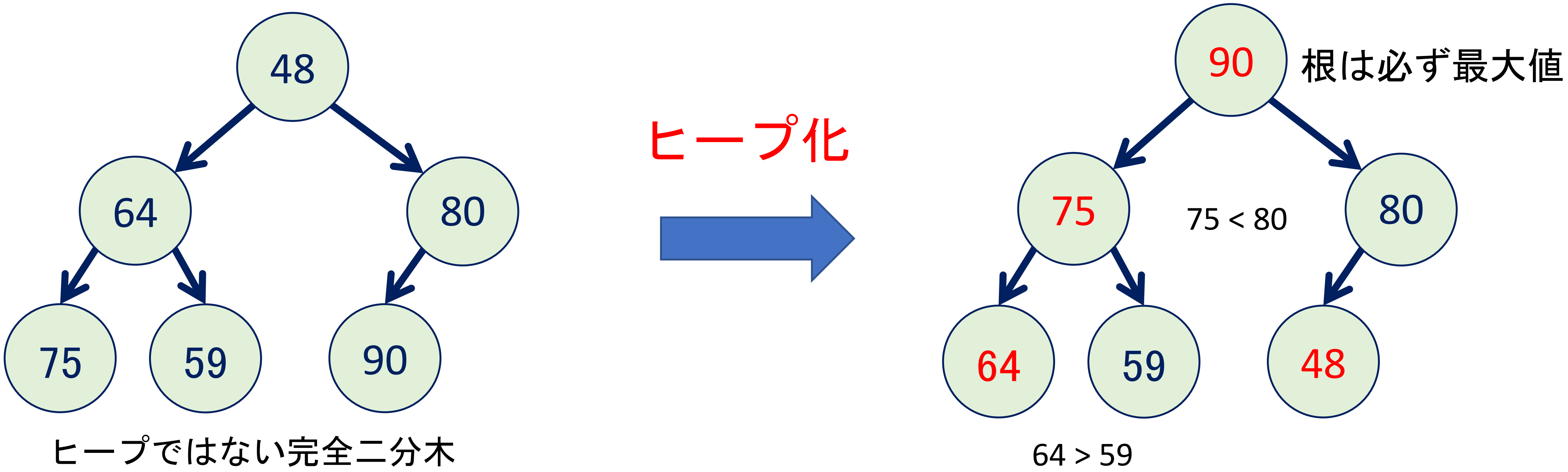
完全二分木

最下流以外の階層では,
ノードがすべて詰まっている.

最下流の階層に限っては左側からノードが詰まっていればよく, 途中までしかノードがなくても良い.
右側は空いていてもよい.

ヒープ (heap: **堆積**)

- ノードには値が与えられており, 親のノードの値が子のノードの値以上である条件を満たす完全二分木を**ヒープ**と呼ぶ.
- ただし, 逆順にソートするならば, 親の値が子の値以下となる.
- ヒープでは, 兄弟の大小における位置関係は任意 (半順序木).

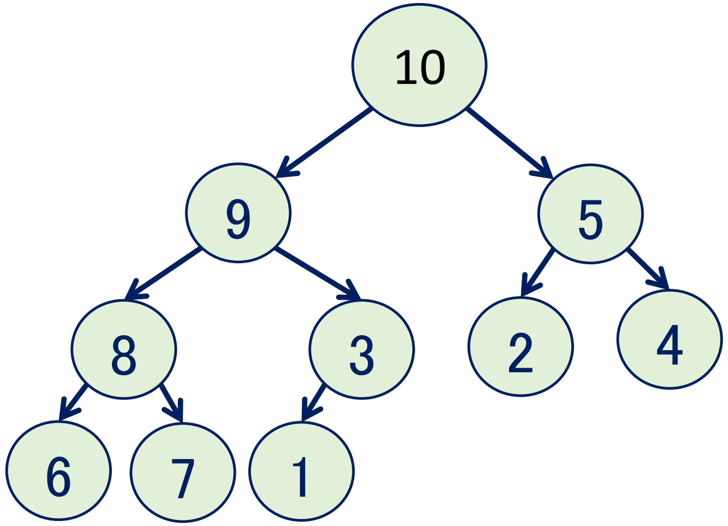


2. ヒープソートのアルゴリズム

ヒープソート概要 (1/2)

ヒープ構造を用いて, 最大値が根に位置していることを利用してソートを行うアルゴリズム.

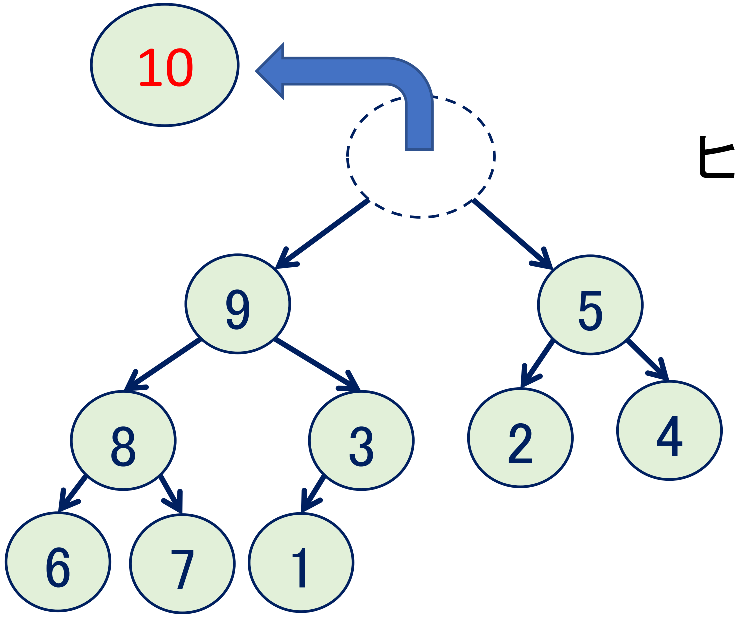
1. ヒープから最大値である根を取り出す.
2. 根以外の残った要素からヒープを再構築(再ヒープ化)する.
3. この処理を繰り返して, ソート処理を行う.



ヒープ化してい状態から
ソート処理を始める

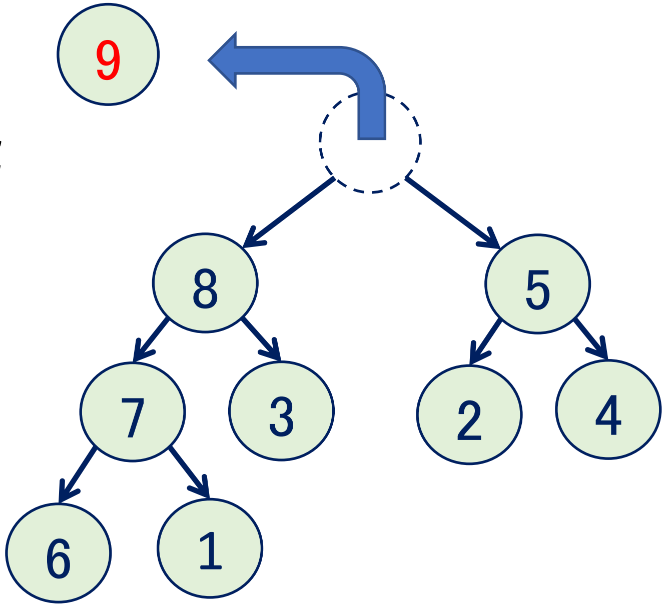
ヒープソート概要 (2/2)

最大値である根を取り出す



ヒープの再構築

残った中で最大値である
新しい根を取り出す



ヒープの再構築

・・・

この処理で取り出した根の値を並べればソートが完了;
選択ソートの応用的アルゴリズム.

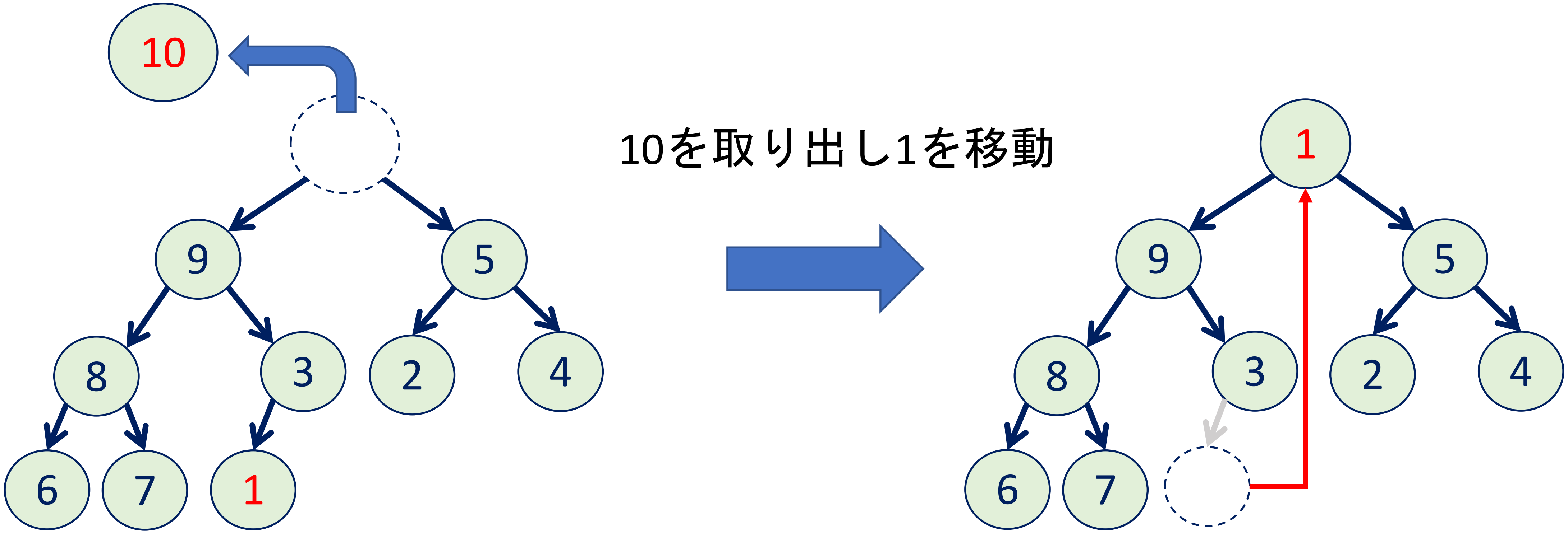
ヒープの再構築 (1/5)

<u>ヒープ再構築 (再ヒープ化)の手順</u>

1. ヒープから最大値である根を取り出す.
2. 最後の要素(最下流の最も右側に位置する要素)を根に移動.
3. 自分自身と, ２つの子のノードのうち値が大きい方の子と値を交換して１つ下流に下る.
4. 以上の処理を, 根から始めて以下の条件のいずれか一方が成立するまで繰り返す.
    - 子の値が親の値以下になる.
    - 葉 (子を持たないノード) に達する.

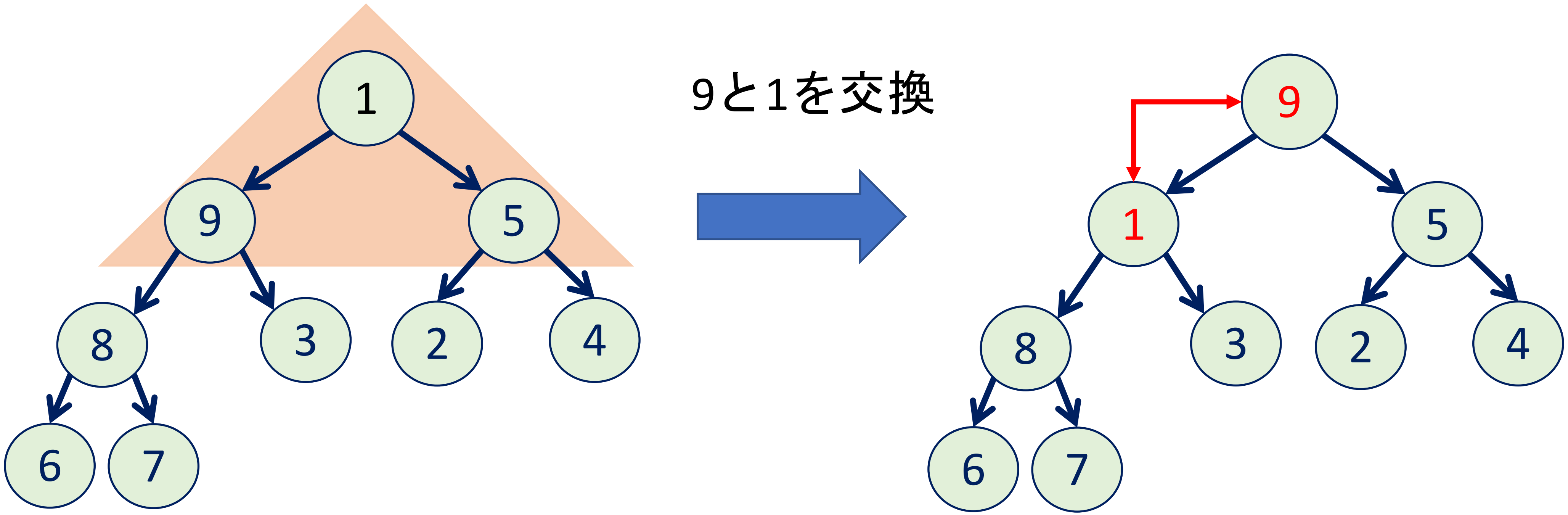
ヒープの再構築 (2/5)

ヒープから根(最大値である10)を取り出す. 空いた根の位置に, ヒープの最後の要素 (最下流の右側の要素)である1を移動.



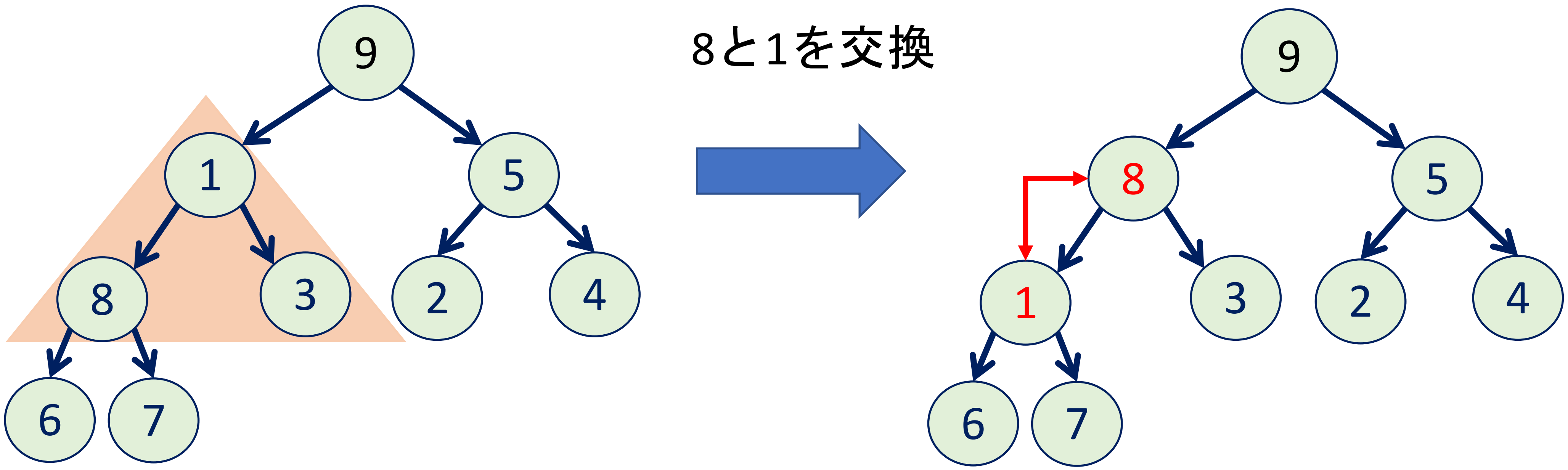
ヒープの再構築 (3/5)

移動した要素1とその子(9,5)に着目し, ヒープの条件を満たすために親と子を交換する. 親1と2つの子(9,5)を比較して, 大きい方の値を持つ子9と親1を交換.



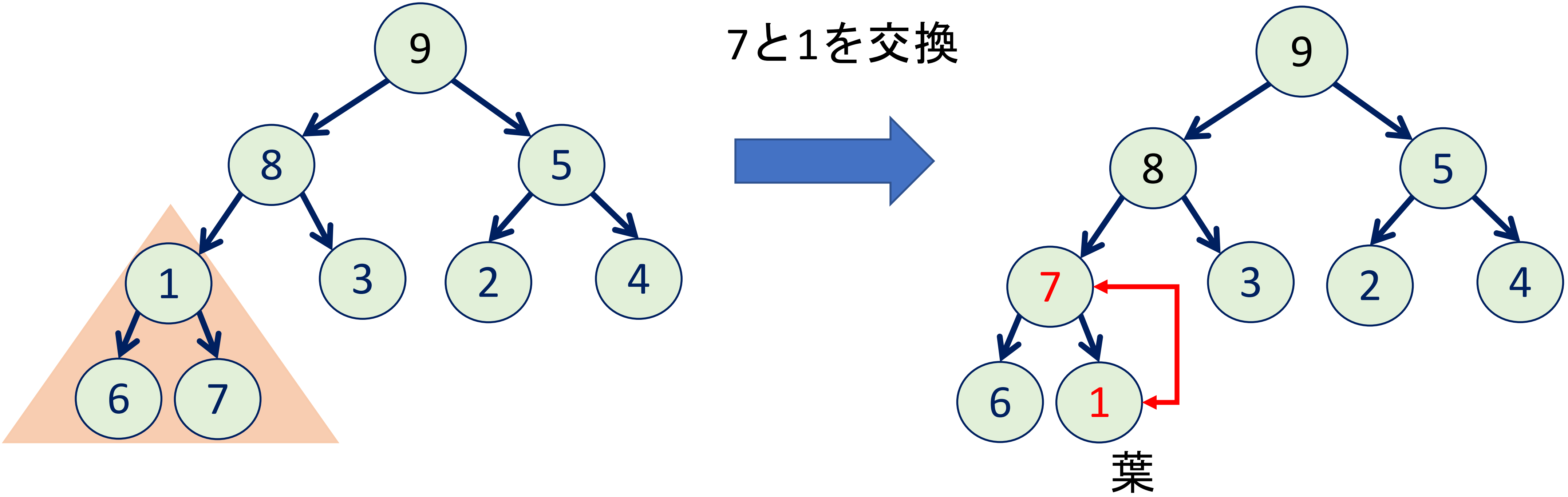
ヒープの再構築 (4/5)

交換した要素1とその子(8,3)に着目し, ヒープの条件を満たすために親と子を交換する. 親1と2つの子(8,3)を比較して, 大きい方の値を持つ子8と親1を交換.

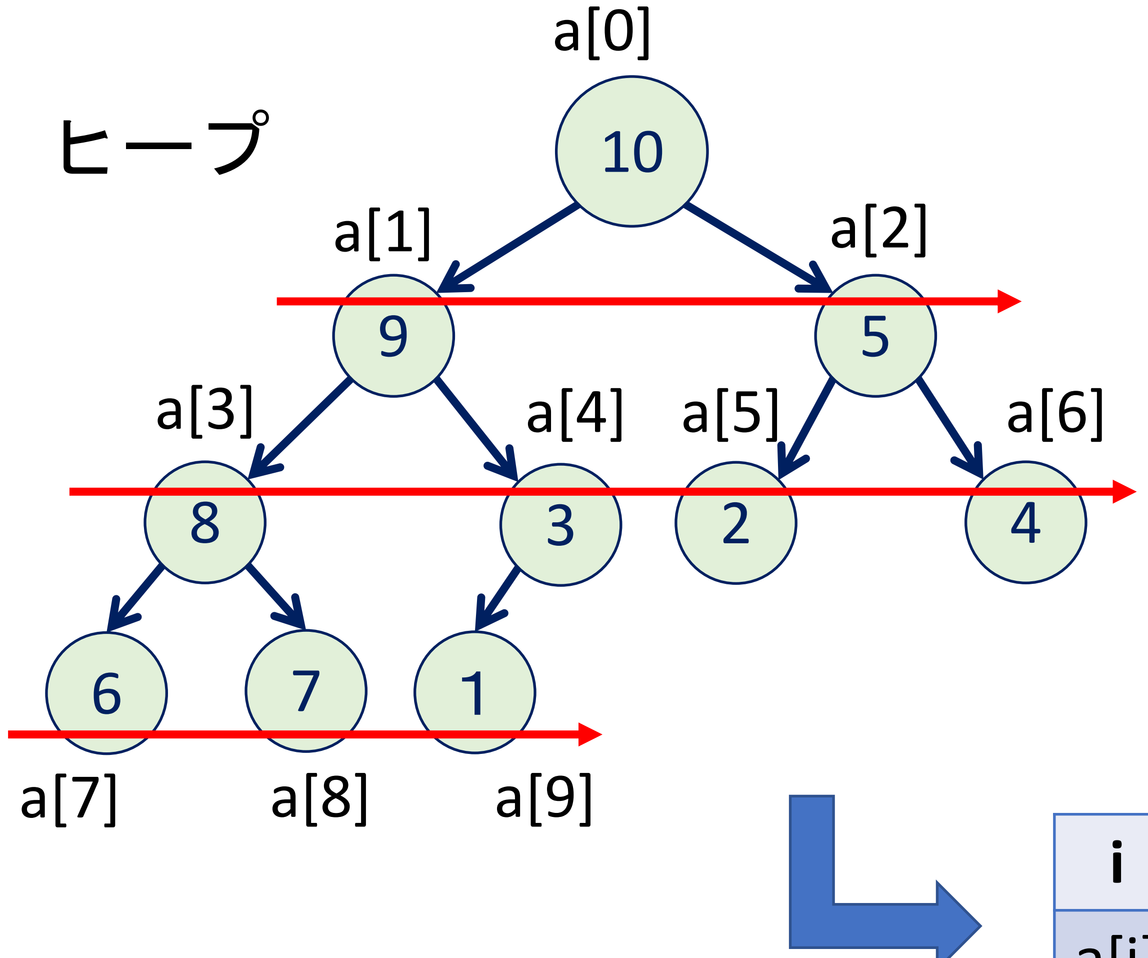


ヒープの再構築 (5/5)

交換した要素1とその子(6,7)に着目し, ヒープの条件を満たすために親と子を交換する. 親1と2つの子(6,7)を比較して, 大きい方の値を持つ子7と親1を交換. 葉に達したのでヒープの再構築が完了.



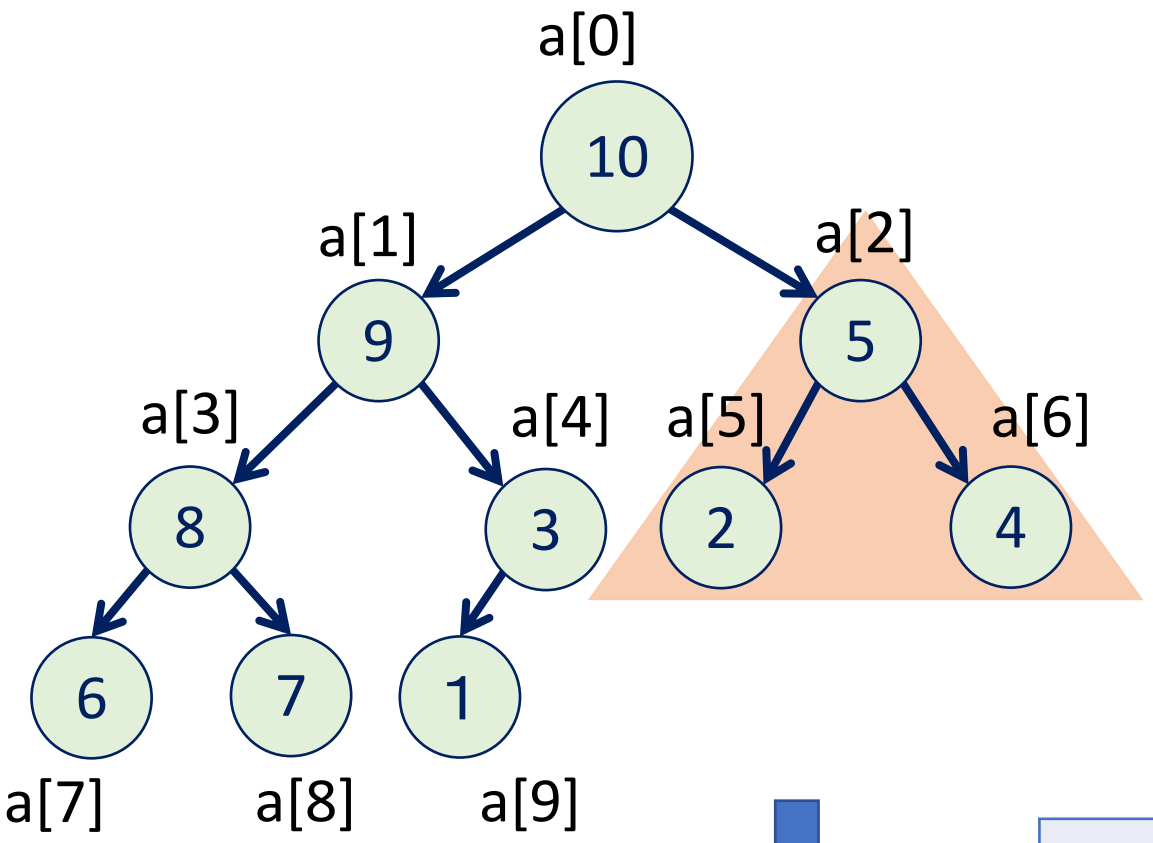
ヒープ再構築を用いたソート (1/6)



- 二分木をリスト構造ではなく配列を用いて実装する.
- 最も上流に位置する根を配列a[0]に格納する.
- １つ下流に降って, 要素を左から右へなぞり, 添字を１つずつ増やしながら, 配列に格納する.
- ヒープの配列への格納完了.

i	0	1	2	3	4	5	6	7	8	9
a[i]	10	9	5	8	3	2	4	6	7	1

ヒープ再構築を用いたソート (2/6)



任意の要素 a[i] に対して

- 親は a[(i-1)/2]
- 左の子は a[i * 2 + 1]
- 右の子は a[i * 2 + 2]

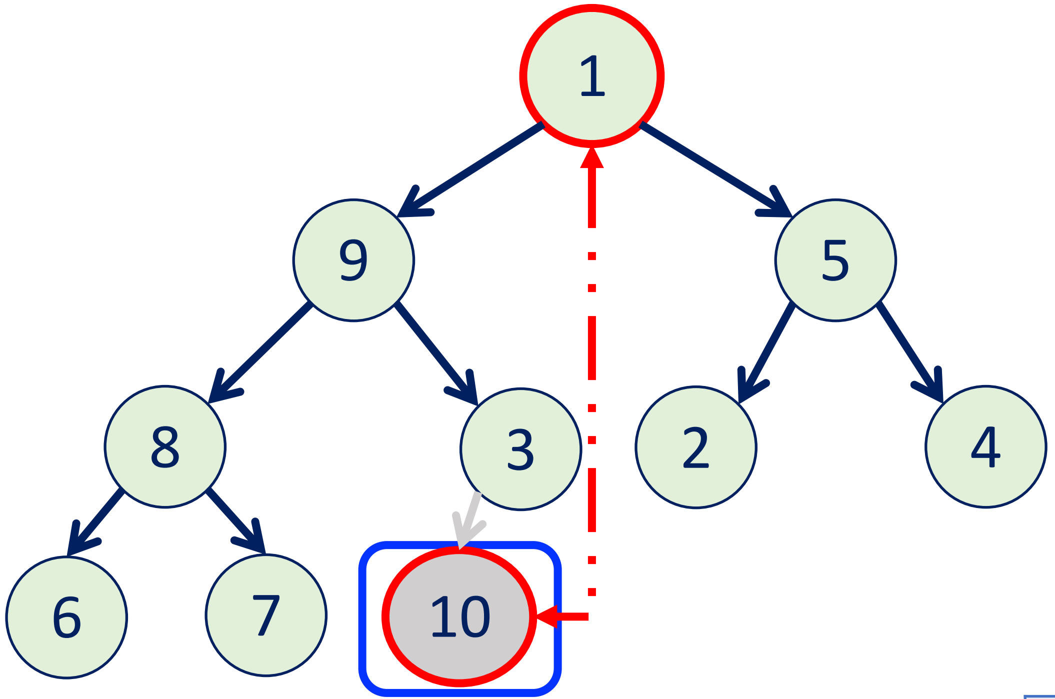
剰余は切捨てる. (**Appendix参照**)

例1: a[i=6] → 親 a[2]

例2: 親 a[i=2], 左子 a[5], 右子 a[6]

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
a[i]	10	9	5	8	3	2	4	6	7	1

ヒープ再構築を用いたソート (3/6)



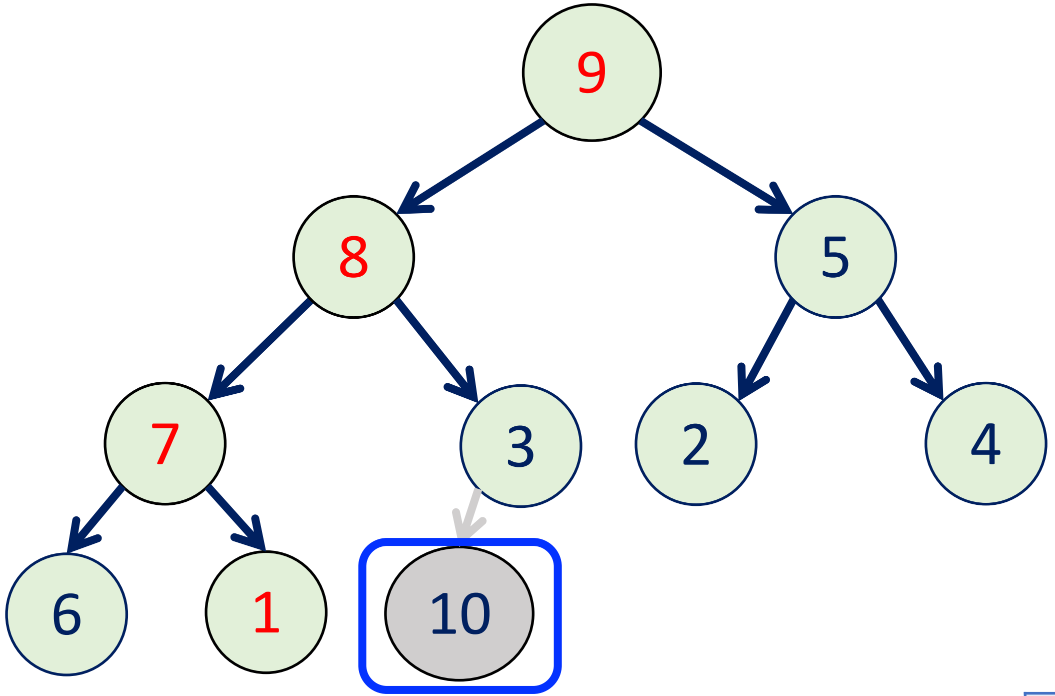
この要素は交換後触らない. ソート済みなので, 単に保管場所として使う.

- ヒープの根a[0] (最大値)と, **ヒープの最後の要素 (最下流の右側の要素＝配列の末尾要素)** であるa[9]を交換.
- ヒープ上では, 10を取り出したことを意味する.

ソート済領域

i	0	1	2	3	4	5	6	7	8	9
a[i]	1	9	5	8	3	2	4	6	7	10

ヒープ再構築を用いたソート (4/6)



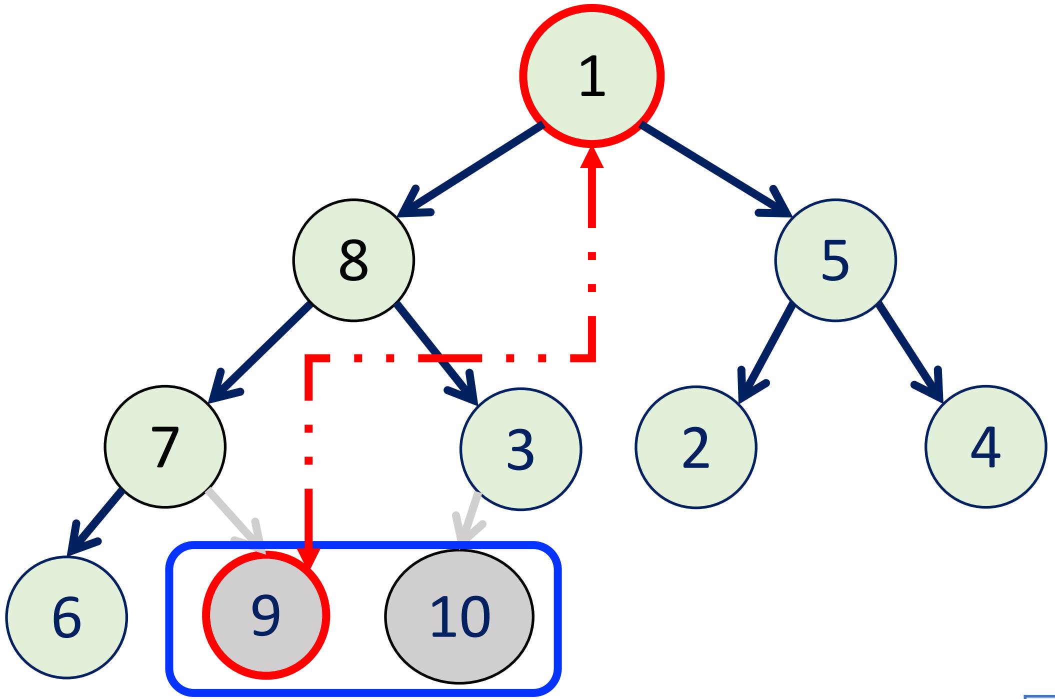
この要素は交換後触らない.
ソート済みなので, 単に保
管場所として使う.

- a[0]--a[8]の要素でヒープを再構築
(後述するdownheap処理を実行)

ソート済領域

i	0	1	2	3	4	5	6	7	8	9
a[i]	**9**	**8**	5	**7**	3	2	4	6	**1**	10

ヒープ再構築を用いたソート (5/6)



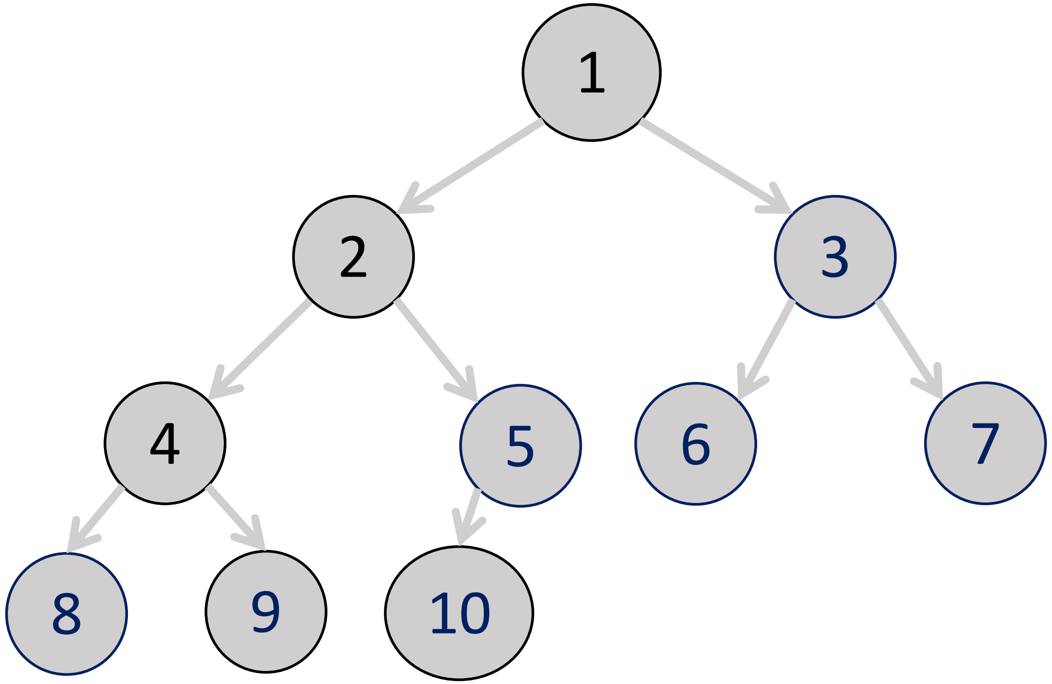
この２つの要素は交換後触らない. ソート済みなので, 単に保管場所として使う.

- ヒープの根a[0] (最大値)と, ヒープを構築した最後の要素であるa[8]を交換.

ソート済領域

i	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
a[i]	**1**	8	5	7	3	2	4	6	**9**	10

ヒープ再構築を用いたソート (6/6)



- 以下同様にa[0]--a[7]でヒープを再構築し, 要素交換.
- 以上の処理を繰り返していくと, 配列の末尾側に大きいほうから順に１つずつ値が格納される.
- 最終的に親を持たない根が残り, ソート化完了.

ソート済

i	0	1	2	3	4	5	6	7	8	9
a[i]	1	2	3	4	5	6	7	8	9	10

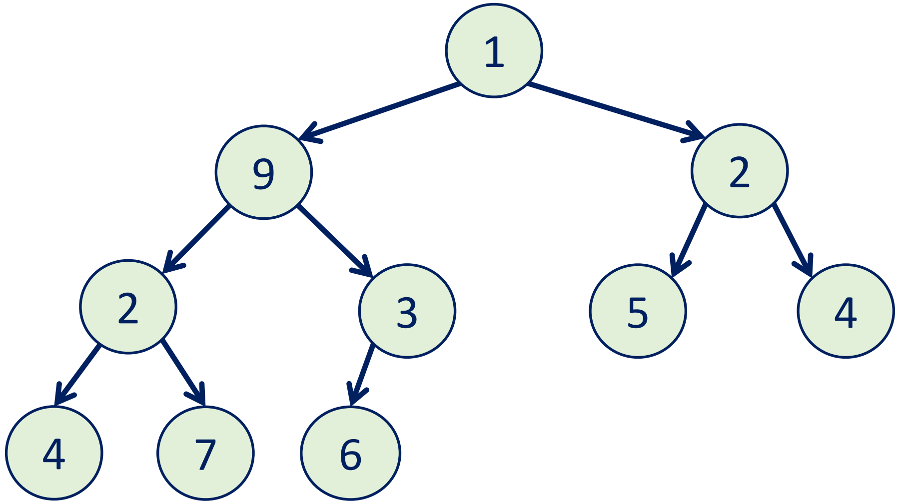
3. ヒープソートの実装の準備

ヒープソート実装手順について

- ヒープの再構築は配列の初期状態がヒープとなっていることを前提としている.
- ヒープソートを実施する上では, (i) 配列のヒープ化を行った後に, (ii) ヒープ再構築を用いたソートを行うといった, 2段階の処理が必要となる.
- 配列のヒープ化を行う明白な理由: 配列の初期状態がヒープの条件を満たしている保証はないため. 従ってソート処理を行う前の配列のヒープ化は重要. → 次スライド

配列のヒープ化の概要 (1/4)

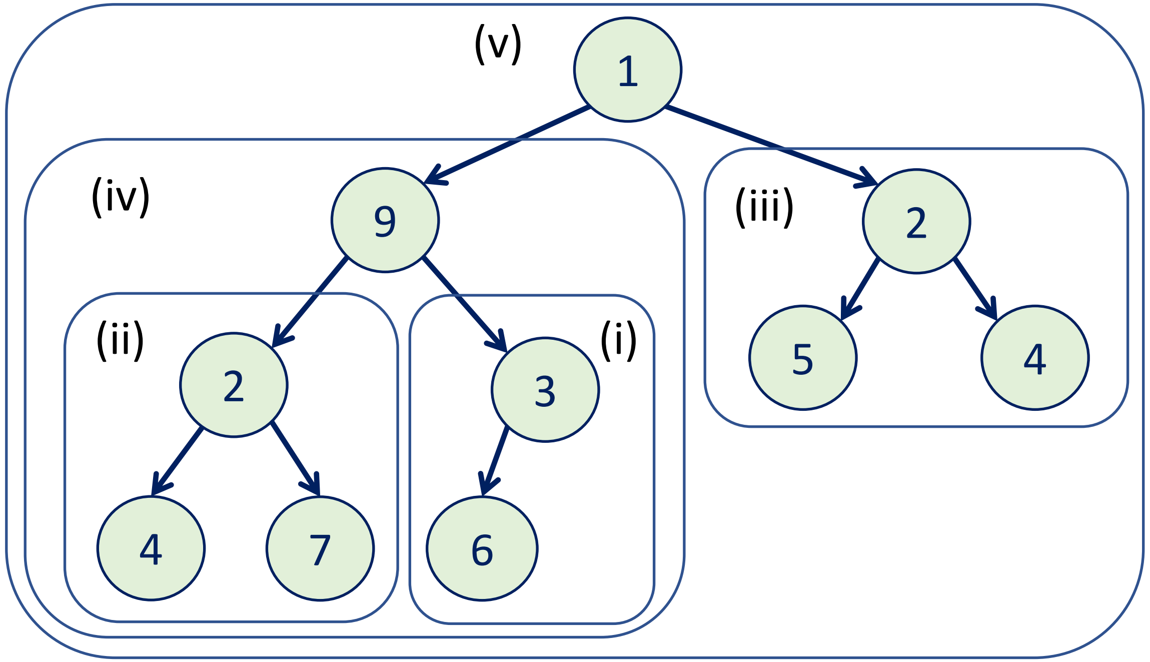
- ヒープ化とは: (i) 根を除去した後に, (ii) 最後の要素を根に移動させ, (iii) 根の要素の値を最下流まで適切に下す処理.
- 配列のヒープ化の概要: 下層の部分木からヒープ化. 次に着目する部分木の範囲を広げヒープ化. その処理を繰り返すことで, 最後には木全体をヒープ化する (ボトムアップ的なヒープ化).



配列のヒープ化の概要 (2/4)

- ボトムアップ的に部分木から順番にヒープ構造を構築する.
  - 下図では(i)--(v)の順でヒープを構築していくイメージ.

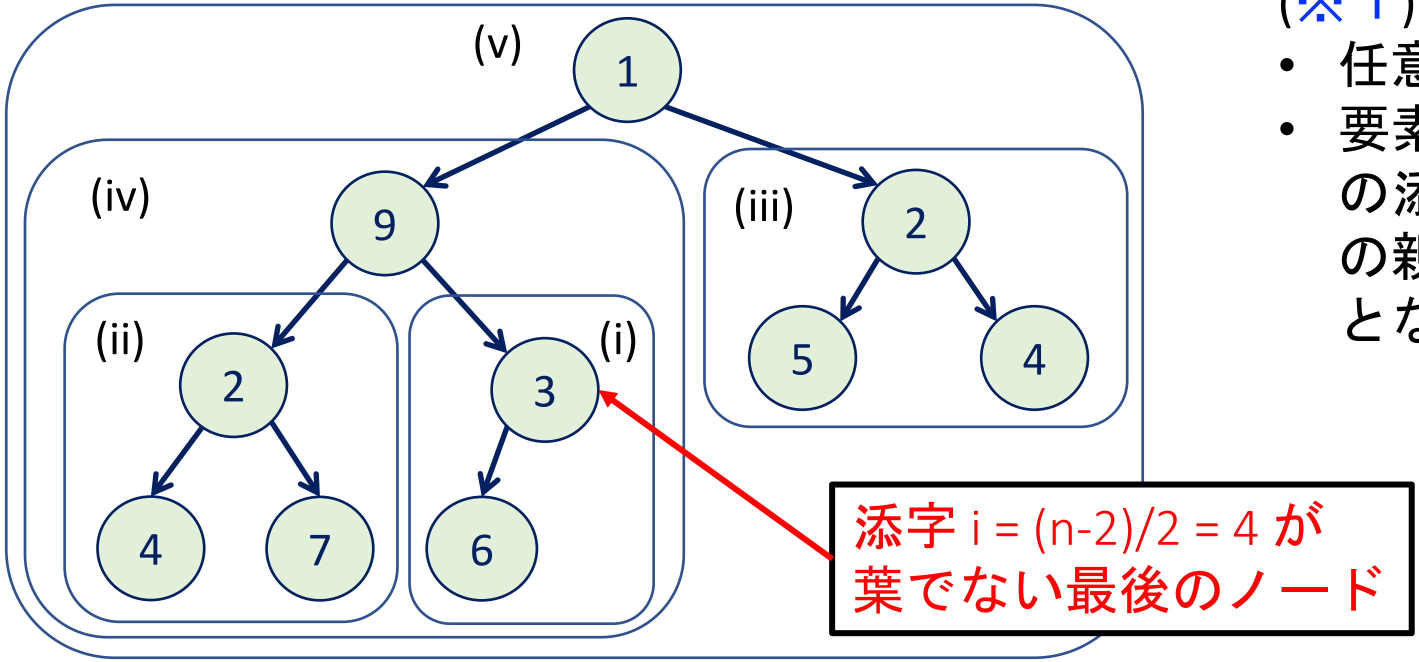
ヒープになっていない完全二分木 (ヒープ化前)



(i),(ii)が最下層の部分木, (iii),(iv)は異なる深さを持つ部分木.

配列のヒープ化の概要 (3/4)

- 配列の要素数(ノード数)をnとすると, 葉ではない最後のノードが部分木(i)の根である. その根は末尾ノードの親であるため, その親の添え字は(n-2)/2 (※1)としてアクセスできる.

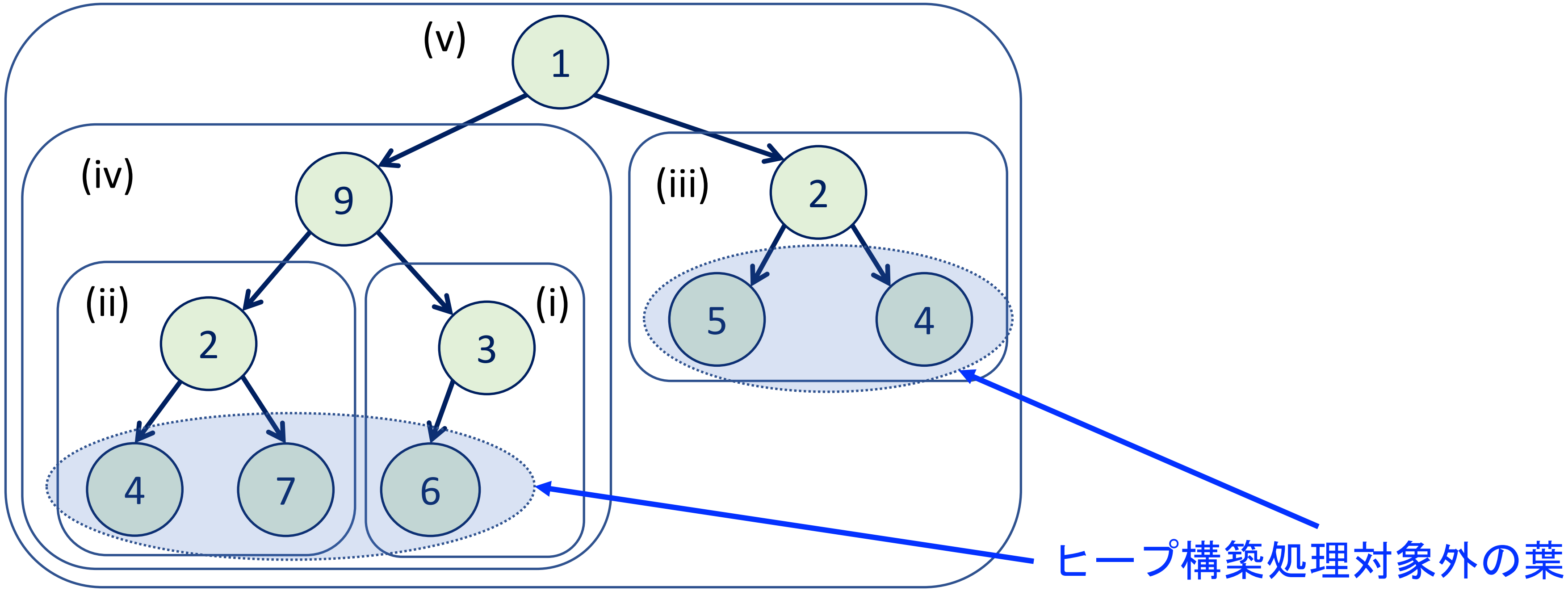


(※ 1)

- 任意のノード i の親は a[(i-1)/2]
- 要素数がnのとき, 末尾のノードの添え字はn-1であり, 末尾ノードの親はa[(n-1-1) /2] = a[(n-2) /2] となる

配列のヒープ化の概要 (4/4)

- これ以降の下流のノードは葉であるため, ヒープ構築処理の対象としない. その親のヒープ構築を行えば必然的にヒープ関係が構築される.
  - 下図はn=10なので, i=(10-2)/2=4, つまりa[4]=3が最後の親であり, a[5]=5以降はヒープ構築処理の対象としない



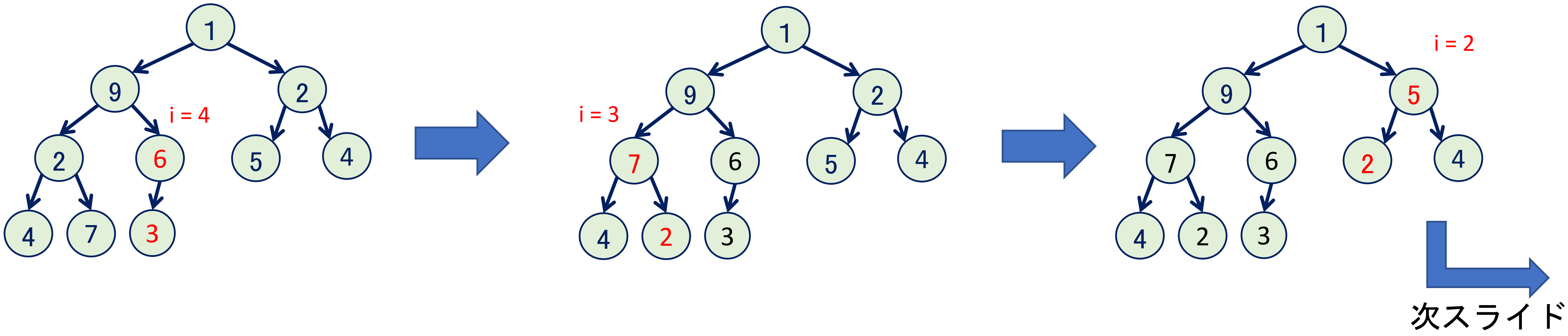
配列のヒープ化の手順 (1/2)

1. 変数iを最後の親ノードの添え字(n-2)/2で初期化.
2. a[i]の値を適切な場所まで下ろす (downheap処理).
3. i-- とデクリメントして i が 0 未満になれば終了, それ以外は手順2へ戻る.

1. i=(n-2)/2=4
2. a[4]の値を適切な場所に下ろす
3. i-- (i=3)

2. a[3]の値を適切な場所に下ろす
3. i-- (i=2)

2. a[2]の値を適切な場所に下ろす
3. i-- (i=1)

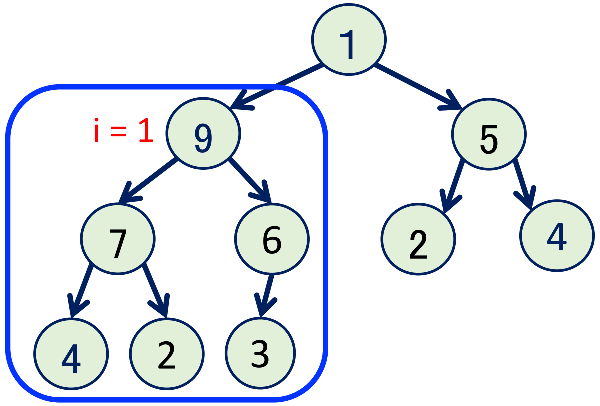


配列のヒープ化の手順 (2/2)

1. 変数iを最後の親ノードの添え字(n-2)/2で初期化.
2. a[i]の値を適切な場所まで下ろす (downheap処理).
3. i-- とデクリメントして i が 0 未満になれば終了, それ以外は手順2へ戻る.

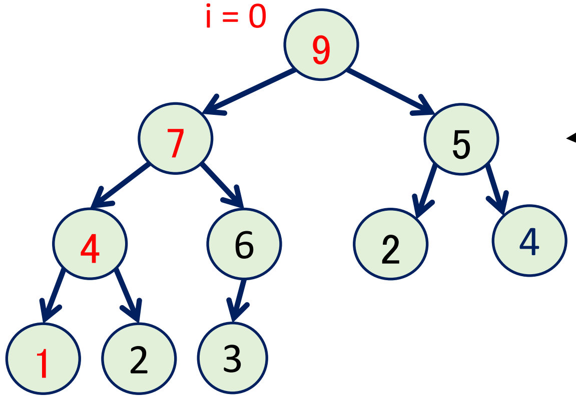
2. a[1]の値を適切な場所に下ろす
3. i-- (i=0)

2. a[0]の値を適切な場所に下ろす
3. i-- (i=-1なので終了)

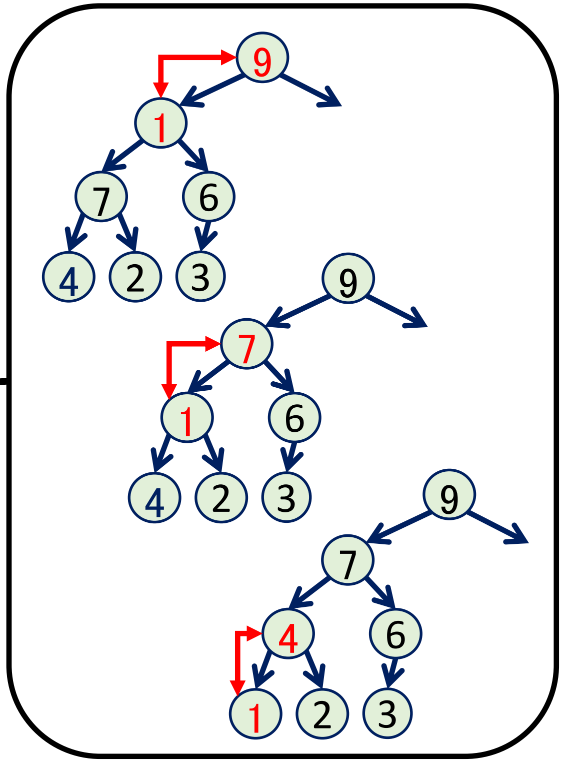


前スライドから

既にヒープ化している部分木
従ってノード位置の変更なし



ヒープ化した木



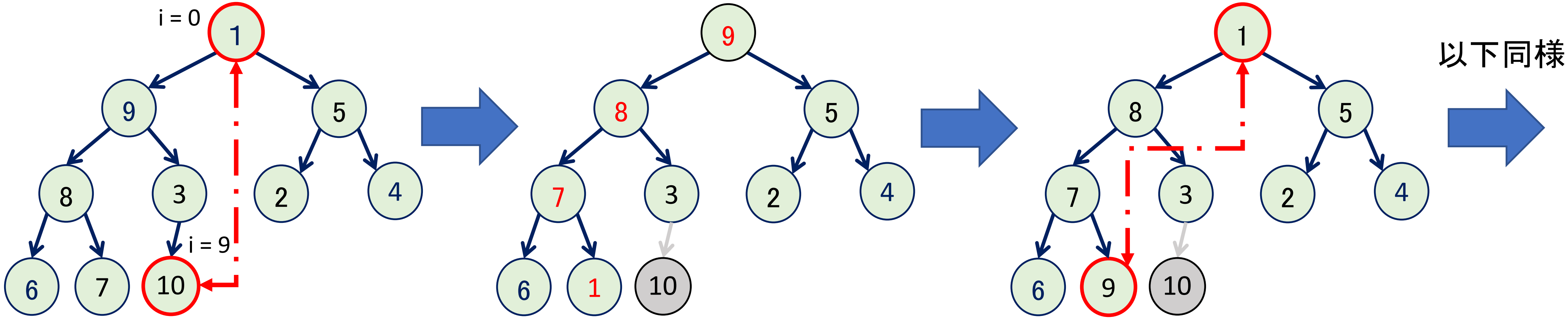
ヒープ再構築を用いたソート

1. 変数 i の値を末尾ノードの添え字 n-1 で初期化
2. a[0] と a[i] を交換する.
3. a[0]の値を適切な場所まで下ろす (downheap処理)
4. i-- とデクリメントしてi = 0 になれば終了, それ以外は手順2へ戻る.

1. i = n-1 = 9
2. a[0]とa[9]を交換

3. a[0]の値を適切な場所まで下ろす

4. i-- (i=8)
2. a[0]とa[8]を交換



4. 課題演習

課題内容 (課題ファイル参照)

- CoursePowerで配布されるスケルトンファイルを完成させ, ヒープソートを行うプログラムを作成する.
- 右に示したファイルを読み込み, 整数型の配列としてデータを格納する. csvファイルはCoursePowerで配布.
- 格納されたデータに対して, “数値”を昇順にソートして表示する. 右のような実行例が得られることを確認すること.

<u>data13.csv</u>
23,1,12,29,10,16,20,21,8,26,4,11,30,18,24

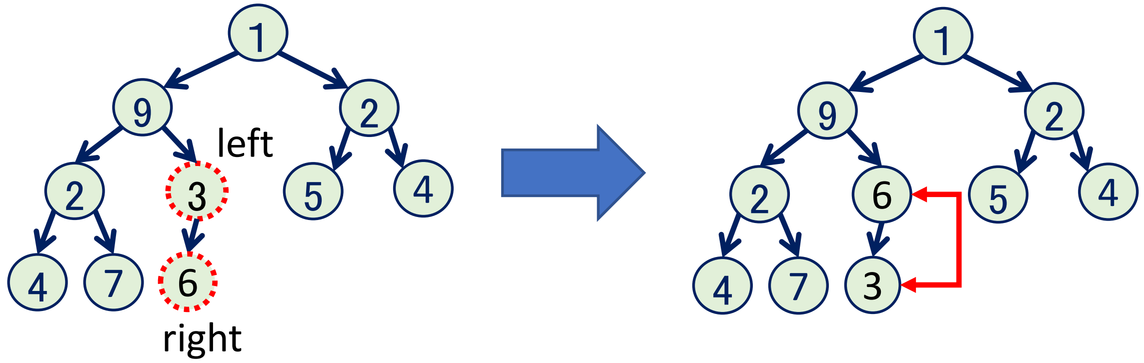
❑ コマンドラインでの実行例
$ ./a.out (もしくは./a.exe)
array before sorting
23,1,12,29,10,16,20,21,8,26,4,11,30,18,24
array after sorting
1,4,8,10,11,12,16,18,20,21,23,24,26,29,30

プログラムでのdownheap処理

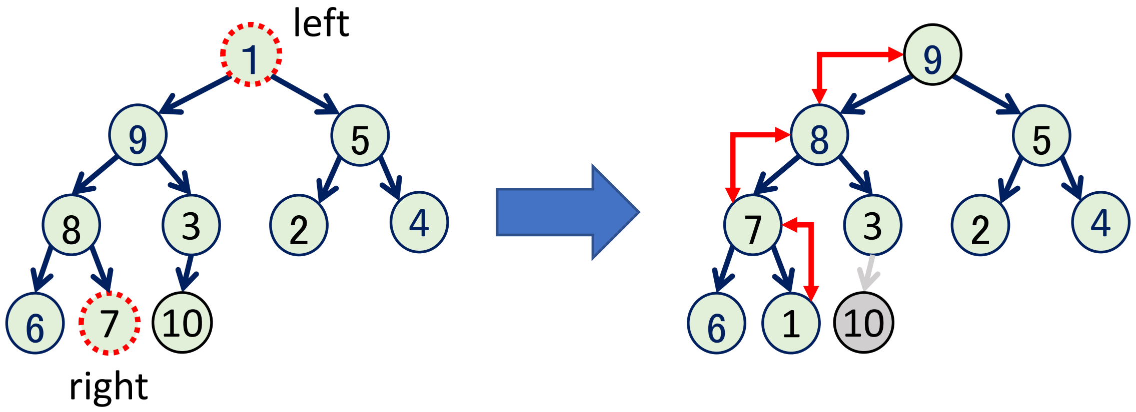
- 着目ノード(初期値はa[left](※1))の値を適切な場所まで下ろす. つまり着目ノードの2つの子のうち値が大きい方の子と値を交換して1つ下流に下る作業を, 以下の条件のいずれか一方が成立するまで繰り返す.
  - 子が親 (着目ノード) 以下の値である
  - 葉に達した

(※1) プログラムに合わせた表現

(i) 配列のヒープ化におけるdownheap()の処理



(ii) ヒープ再構築を用いたソートにおけるdownheap()の処理



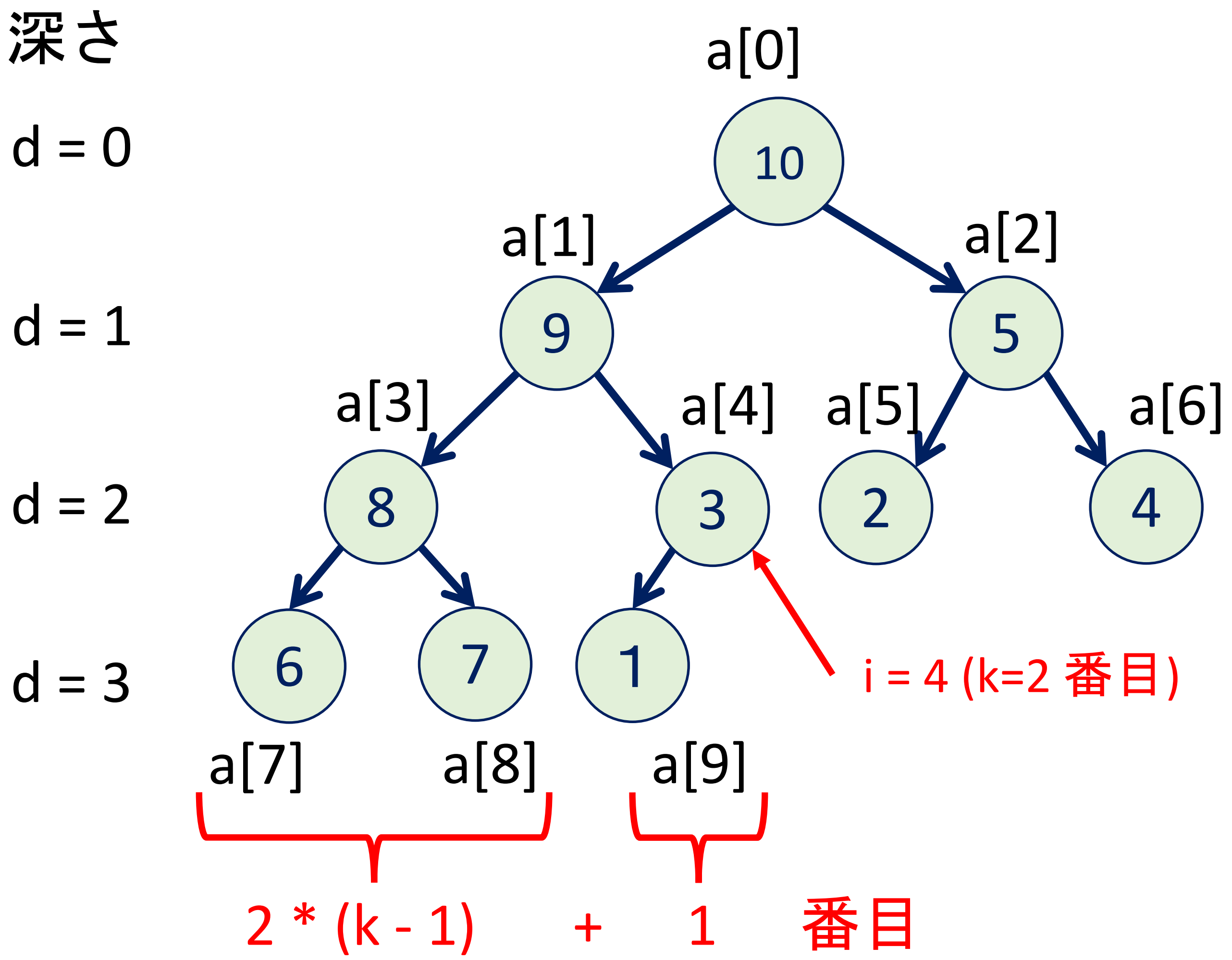
講義内容のまとめ

- ヒープ: 親のノードの値が子のノードの値以上である条件を満たす完全二分木.
- ヒープソート: ヒープ化された木の根の取り出しと, 取り出し後の木構造の再ヒープ化を繰り返すことで, ノードを並べ替えるアルゴリズム.
- プログラムでは, 初期状態の配列のヒープ化と, ヒープ再構築という２段階処理が重要.

質問・コメントを歓迎します.
対面での対応は, 12号館1階2104室.
メールもしくはteamsのチャットでも対応します.

Appendix

添字の関係式に関する補足



- 添え字iのノードが深さd, 左からk番目に存在するとするとi = $(2^d-1) + k - 1 = 2^d + k - 2$.
- 添え字iのノードの左子xは深さがd+1, 左から2(k – 1) + 1番目に位置するため, その添字は $i_x = (2^{d+1} - 1) + 2(k - 1) + 1 - 1 = 2^{d+1} + 2k - 3 = 2(2^d + k - 2) + 1 = 2i + 1$.
- 添え字iのノードの右子yは, 左子xの1つ右のため, その添字は $i_y = 2i + 1 + 1 = 2i + 2$.
- 左子ノード i の親のノードを j とすると, i = 2j + 1の関係から j = (i – 1)/2. ここで右子ノードの添え字は i + 1 かつ偶数であるため, 親ノードの添字 j はガウス記号(床関数)を用いて次式で表現される: $j = \lfloor (i - 1)/2 \rfloor$
- つまり, jは(i – 1)/2を超えない最大整数であり, 剰余は切り捨てる.