アルゴリズム論
（第4回）

岩手県立大学
Iwate Prefectural University

佐々木研（情報システム構築学講座）
講師　山田敬三
k-yamada@iwate-pu.ac.jp

木構造

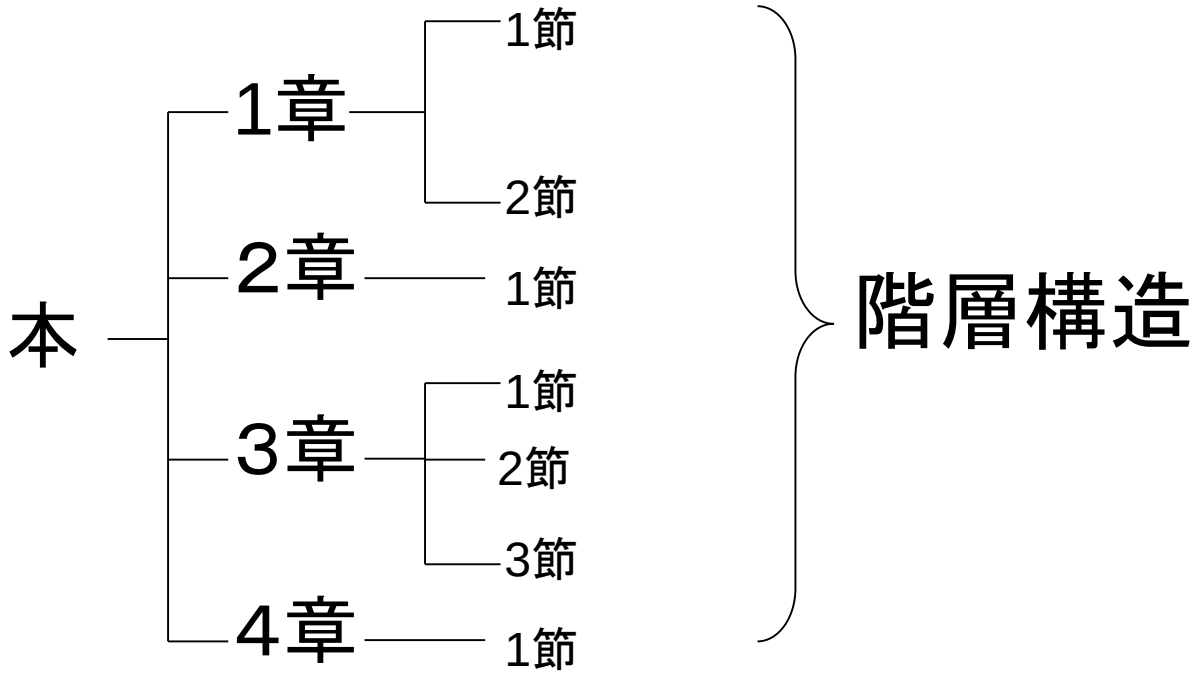
岩手県立大学
Iwate Prefectural University

木構造

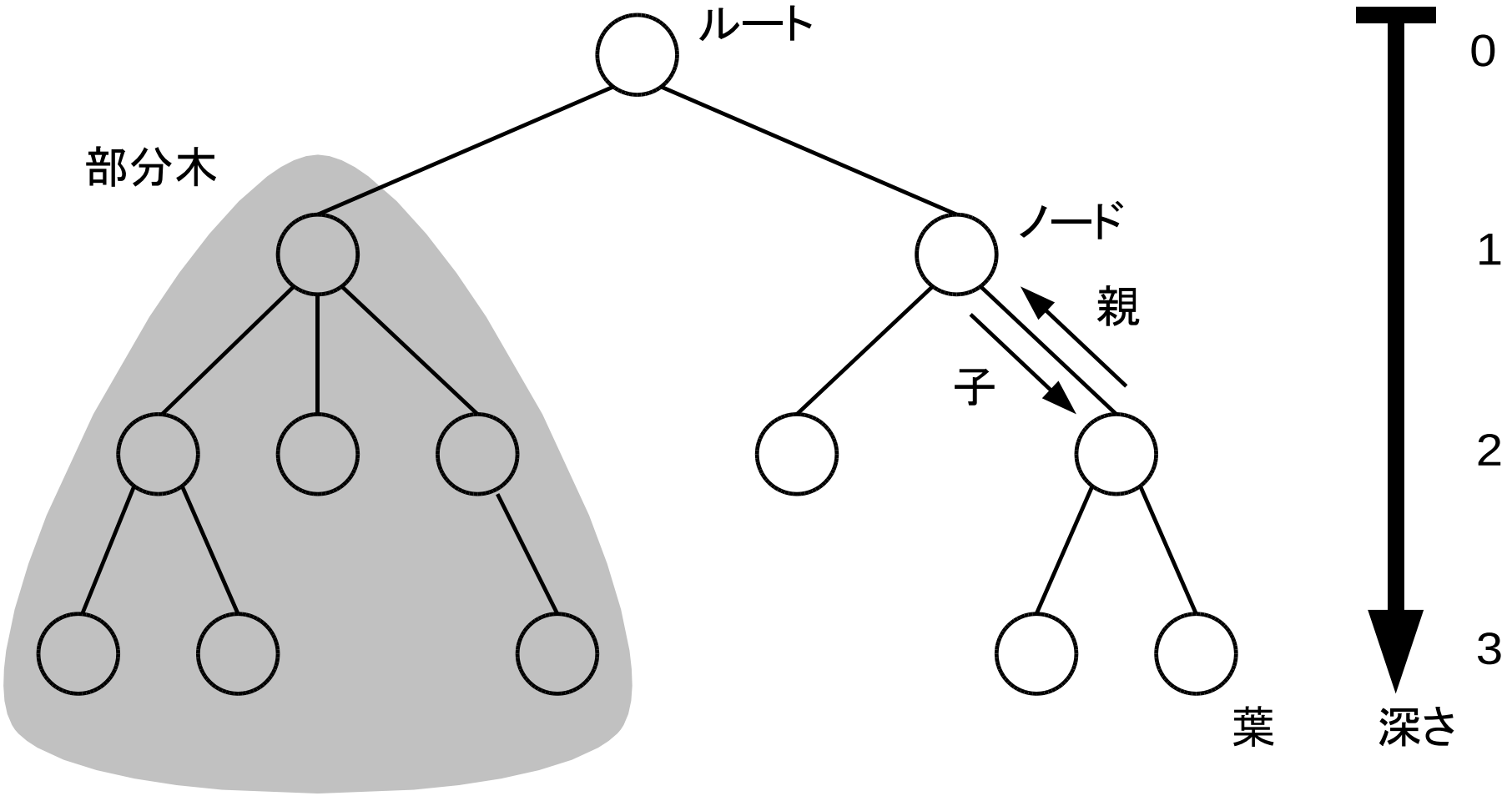
- Tree Structure
- 階層構造を持つデータを扱う.
  - 章や節などの構造をもつ本
  - ディレクトリ，など
- 複数のノードからなるデータ構造

リスト

- 要素が順番に一列に並んだデータ
- 本の構造（章，節，小節）をデータとして扱うときは，リストでは困難



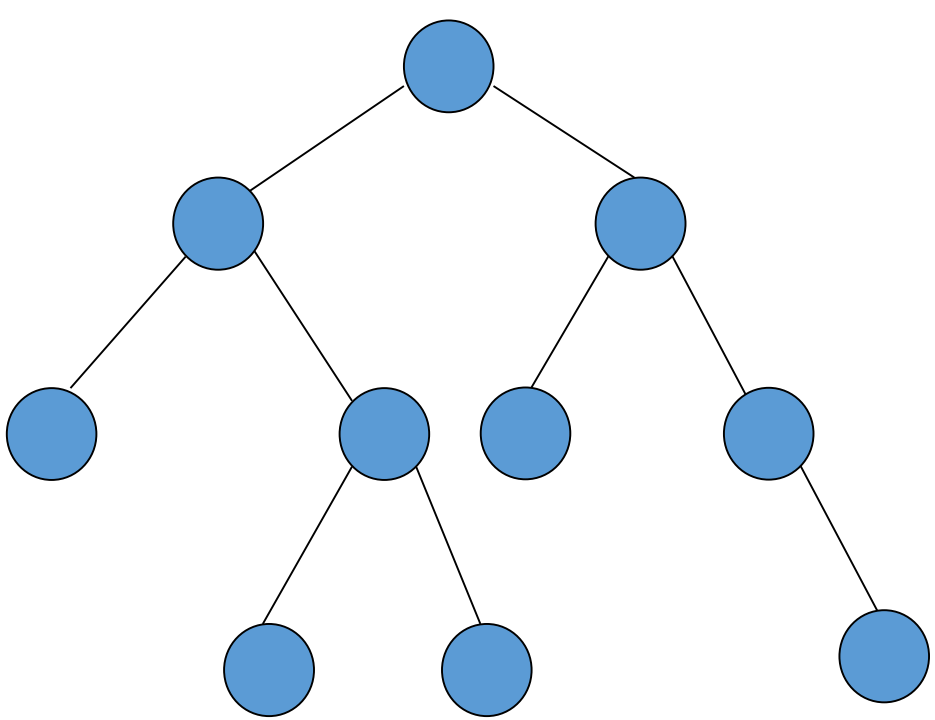
木構造



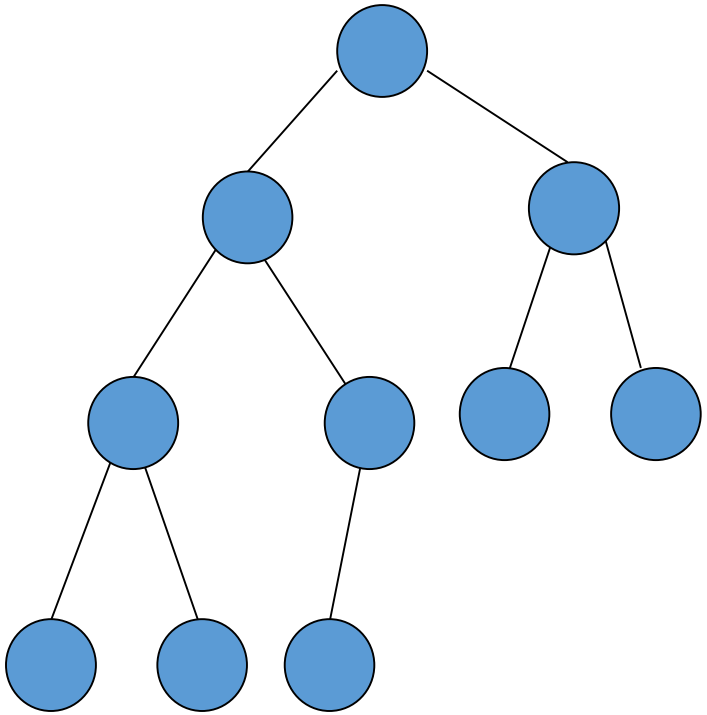
木構造

- 二分木（Binary Tree）
  - 各ノードで枝分かれ
  - 子が2つ以下の木構造
- 完全二分木（Complete Binary Tree）
  - 詰められた二分木
    - ルートから，上から下へ
    - 同レベルでは左から右へ

二分木と完全木分木

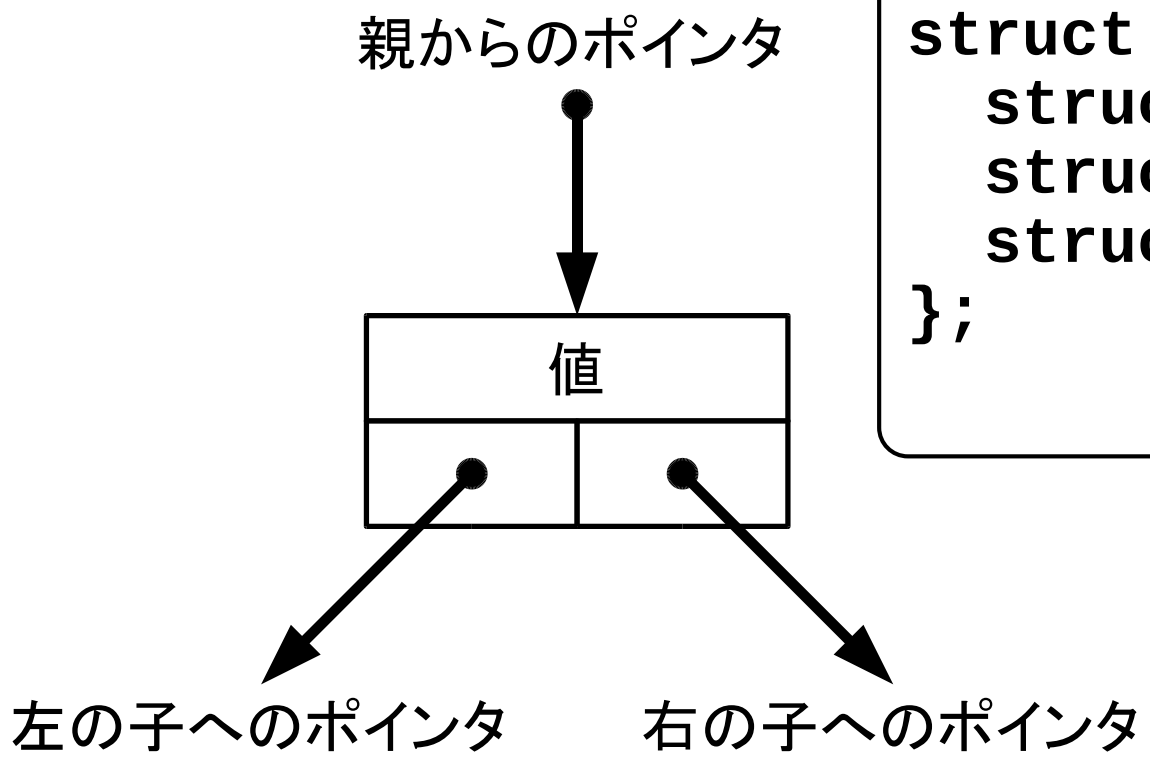


(a)二分木



(b)完全二分木

二分木の実装

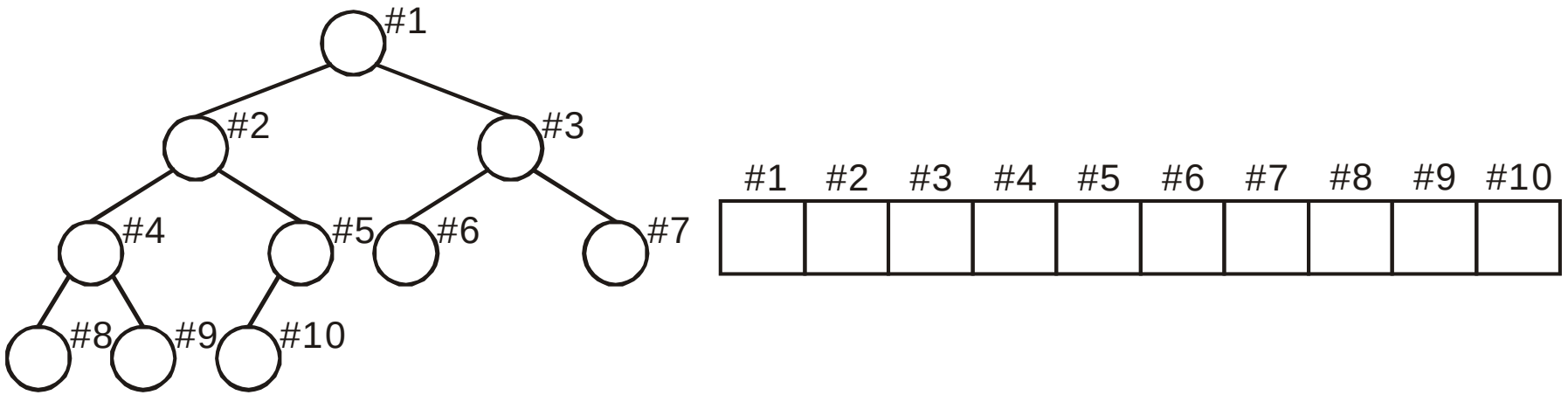



```
struct person {
  char *name;
  int year;
};

struct node {
  struct person *value;
  struct node *child_l;
  struct node *child_r;
};
```

完全二分木の場合

- 配列と簡単な計算式で実現可能
- あるノード **n** に対して
  - 親ノードは **n/2**
  - 子ノードは **2n** および **2n+1**



二分探索木

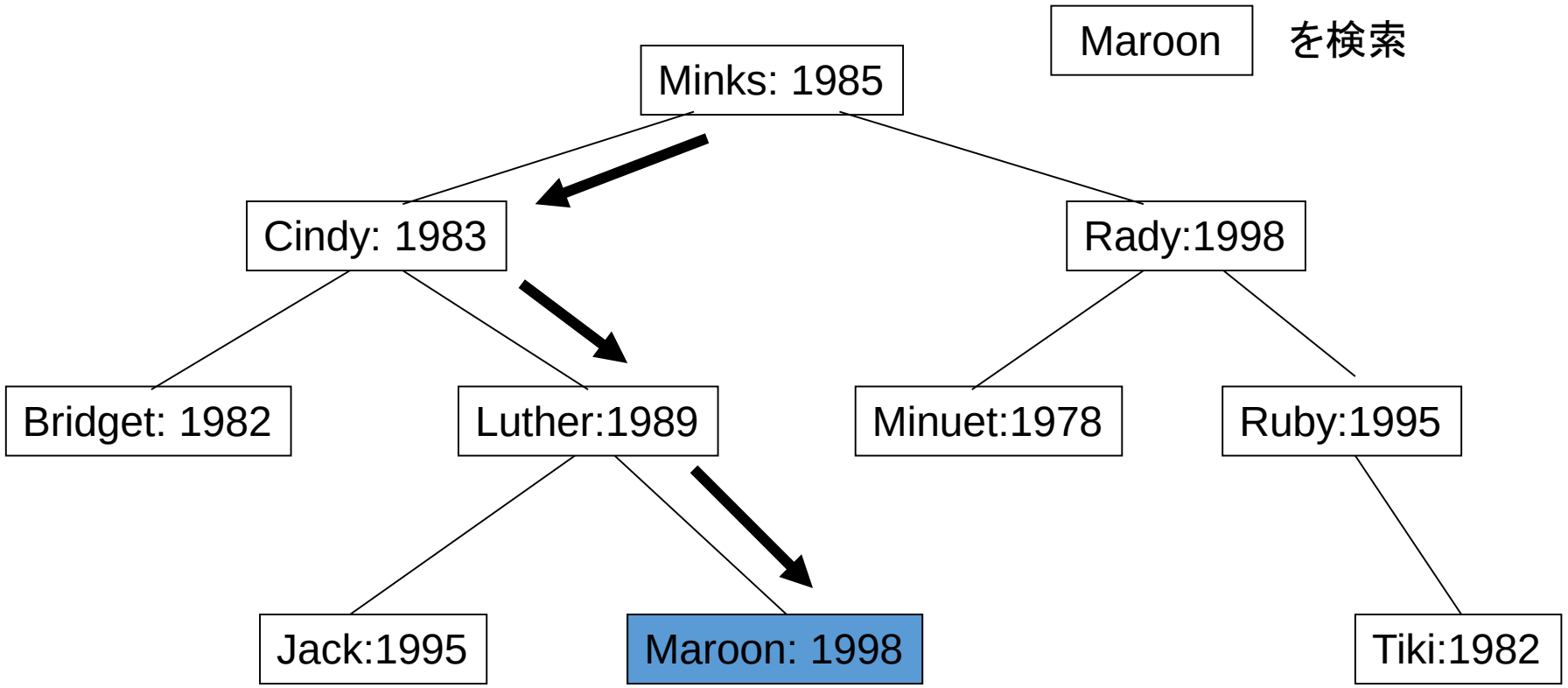
岩手県立大学
Iwate Prefectural University

二分探索木

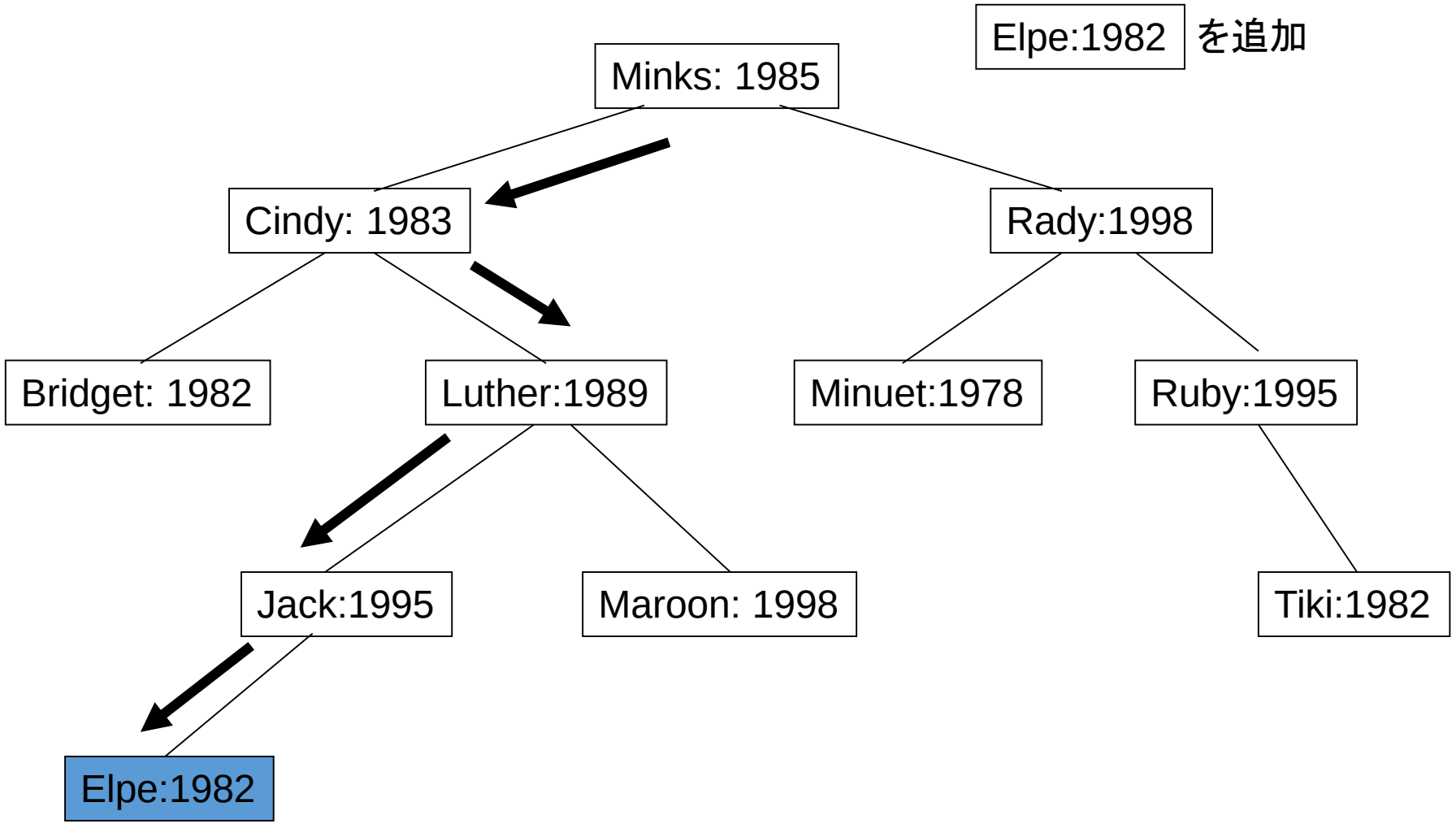
- Binary Search Tree
  - ある検索キーを与えると，それに対応する値を検索可能

- あるノード **x** に対して
  - 左部分木の各ノードの値は **x** より小さい.
  - 右部分木の各ノードの値は **x** より大きい.

ノードの探索


Maroon
を検索
Minks: 1985
Cindy: 1983
Rady:1998
Bridget: 1982
Luther:1989
Minuet:1978
Ruby:1995
Jack:1995
Maroon: 1998
Tiki:1982

ノードの追加


Elpe:1982 を追加
Minks: 1985
Cindy: 1983
Rady:1998
Bridget: 1982
Luther:1989
Minuet:1978
Ruby:1995
Jack:1995
Maroon: 1998
Tiki:1982
Elpe:1982

ノードを探索する関数

```
struct node *search_node(struct
 node *pointer, char *key);
```

- 見つかったノードへのポインタを返す
- ノードへのポインタと探索用のキーを引数

ノードを探索する関数

1. 引数として与えられたポインタが**NULL**ならば、見つからなかったとして**NULL**を返す

```
if(pointerの示す先がNULL) {
  NULLを返す;
}
```

2. 値が探索対象だったら、そのノードへのポインタを返す

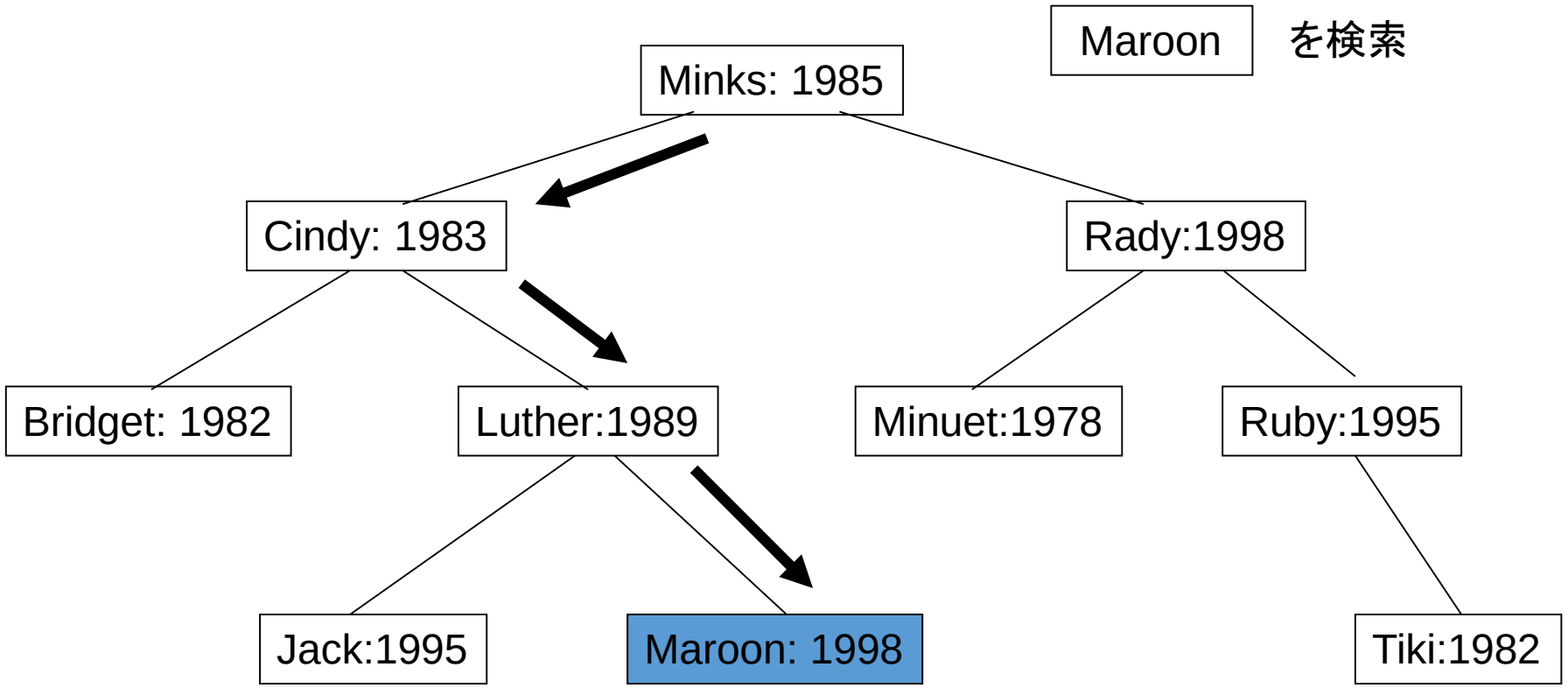
```
if(pointerの示す先の値がキー) {
  pointerの値を返す;
}
```

ノードを探索する関数

3. ポインタの示す先の値が探索対象より大きければ(探索対象のほうが小さければ)左子部分木を再帰的に探索、小さければ右子部分木を再帰的に探索

```
if(pointerの示す値が対象より大きい) {
   左の子部分木を探した結果を返す;
} else {
   右の子部分木を探した結果を返す;
}
```

ノードの探索


Maroon
を検索
Minks: 1985
Cindy: 1983
Rady:1998
Bridget: 1982
Luther:1989
Minuet:1978
Ruby:1995
Jack:1995
Maroon: 1998
Tiki:1982

ノードを追加する関数

```
void add_node(struct node
 **pointer, struct node *new_node)
```

- ノードの追加位置の候補を示すポインタへのポインタと、追加するノードへのポインタを引数

1. 追加位置の候補を示すポインタがNULLなら追加

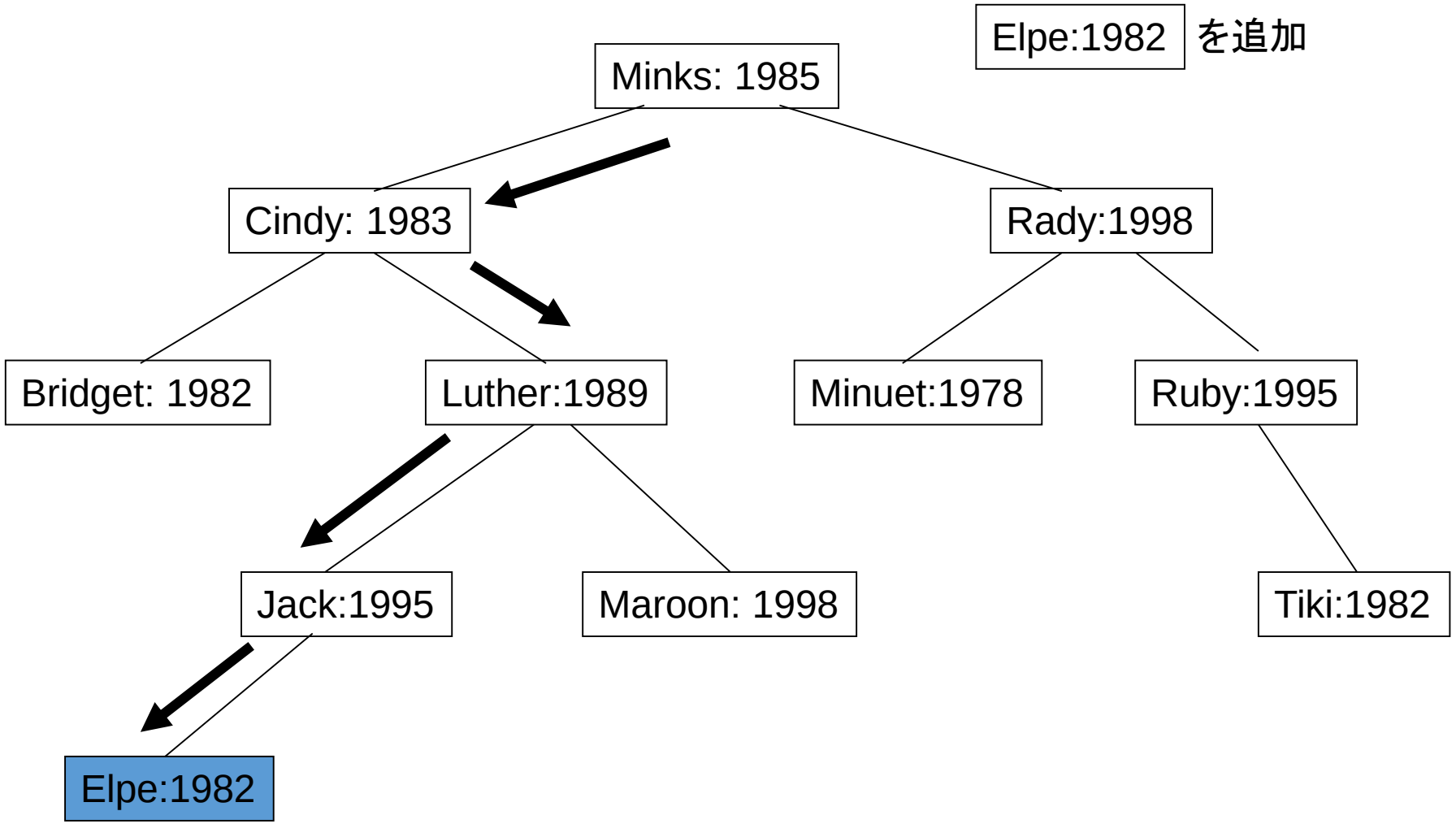
```
if(*pointerの示す先がNULL) {
  *pointerに新規ノードへのポインタを代入;
}
```

ノードを追加する関数

2. 空いてなければ、そのノードの値と新規ノードを比較し、新規ノードの値が小さければ左の子、大きければ右の子を調べる

```
if(*pointerの示す値より新規ノードが小さい) {
   左の子部分木に対し追加位置を調べる;
} else {
   右の子部分木に対し追加位置を調べる;
}
```

ノードの追加


Elpe:1982 を追加
Minks: 1985
Cindy: 1983
Rady:1998
Bridget: 1982
Luther:1989
Minuet:1978
Ruby:1995
Jack:1995
Maroon: 1998
Tiki:1982
Elpe:1982

トラバーサル

- 二分探索木のすべてのノードを訪問
  - 二分木のノードの表示に使用
  - 再帰的な処理
- 訪れたノードを表示する処理をどこに置くかで3種類

行きがけ順

- 先順 : Preorder Traversal

  1. ノードの表示
  2. 左の木を走査する再帰呼出
  3. 右の木を走査する再帰呼出

通りがけ順

- 中順：Inorder Traversal

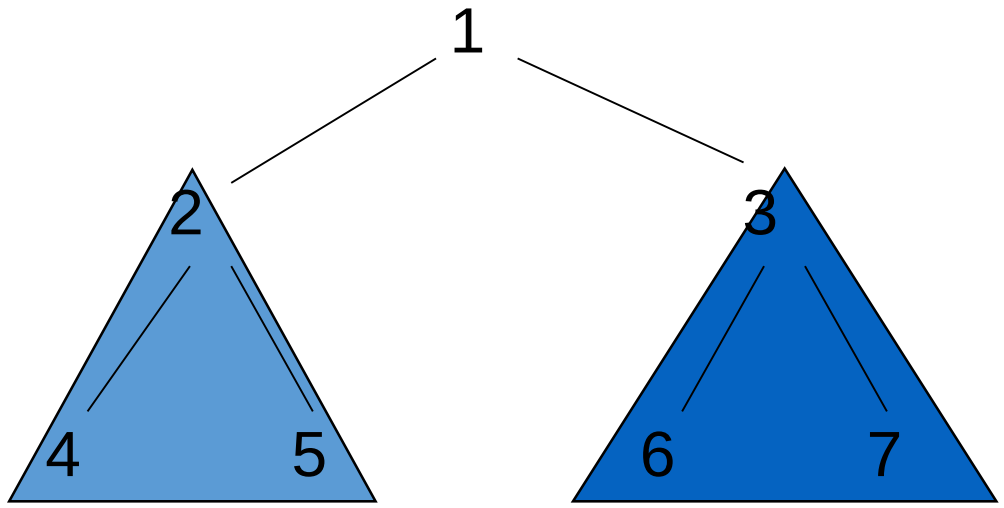
  1. 左の木を走査する再帰呼出
  2. ノードの表示
  3. 右の木を走査する再帰呼出

帰りがけ順

- 後順 : Postorder Traversal

  1. 左の木を走査する再帰呼出
  2. 右の木を走査する再帰呼出
  3. ノードの表示

トラバースの結果



Pre-order: 1 2 4 5 3 6 7

In-order: 4 2 5 1 6 3 7

Post-order: 4 5 2 6 7 3 1

ヒープ

岩手県立大学
Iwate Prefectural University

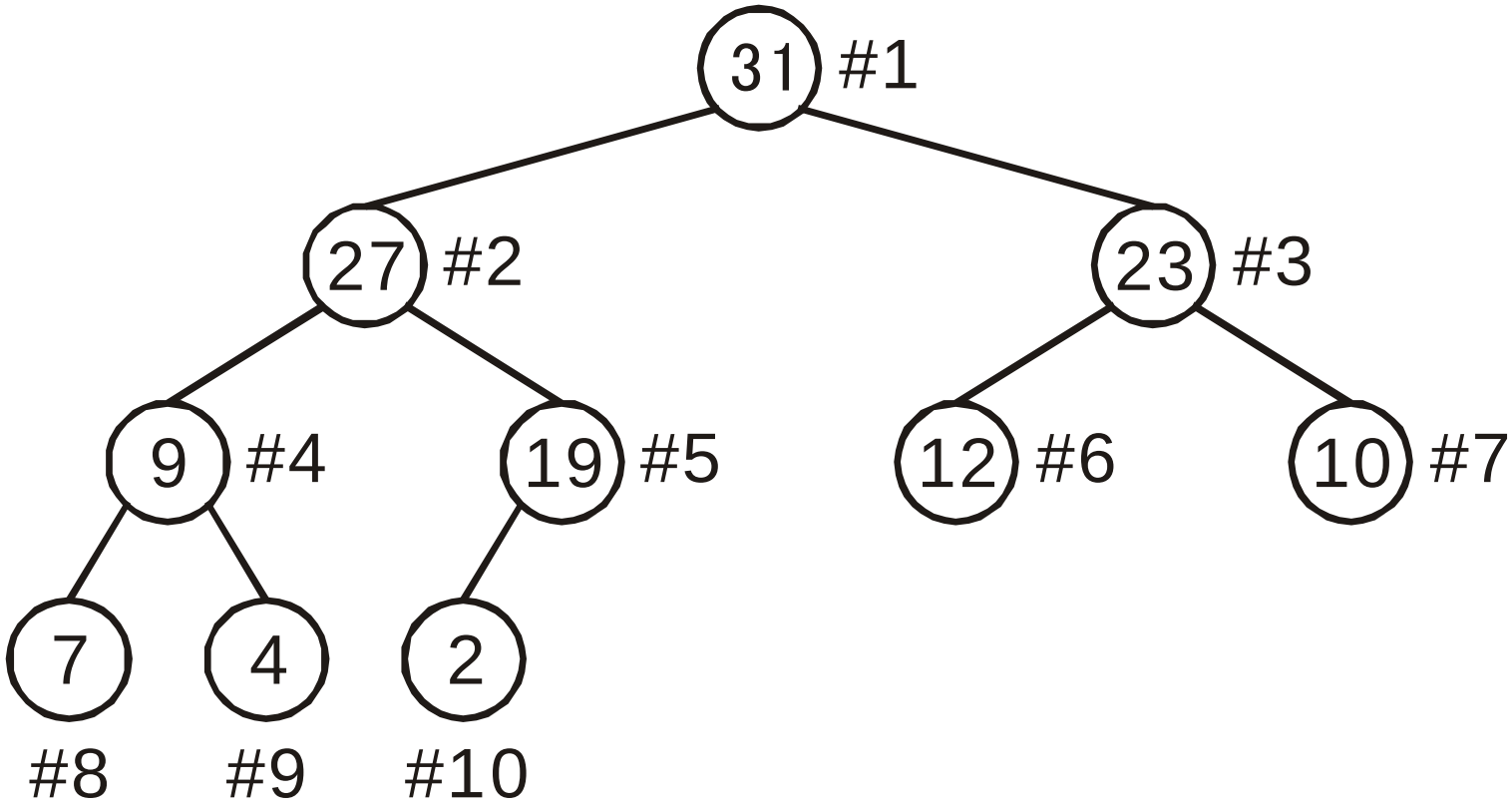
ヒープ(Heap)

- 完全二分木で，以下の条件を満たすもの

  （配列で実現可能）

- ヒープ条件
  - 任意のノードの値は，そのノードのどちらの子の値よりも大きいか等しい．

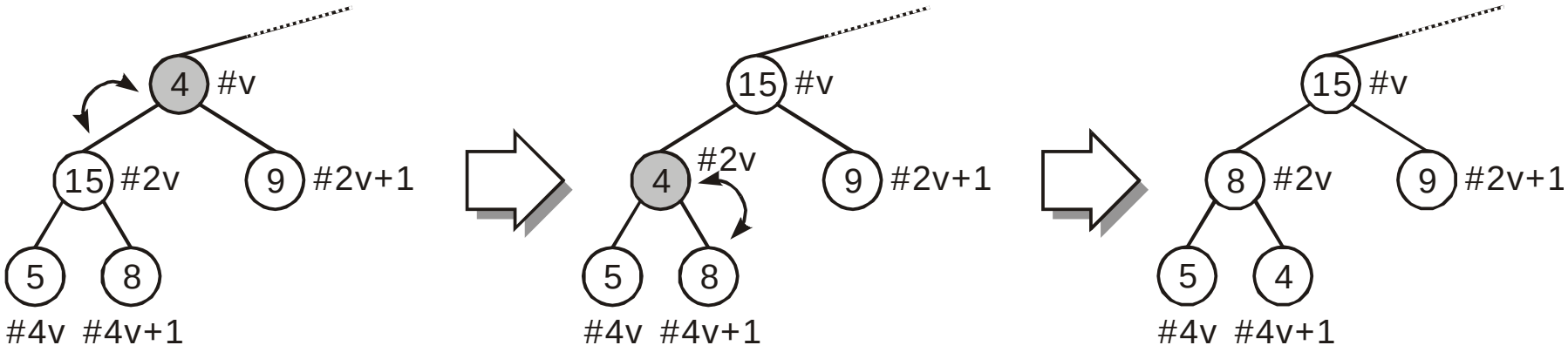
ヒープ(Heap)



下降修復 (Downheap)

- ヒープ条件を満たしていない
  完全二分木をヒープ化する.
  1. ノードvの値がそのどちらかの子の値より小さければ
  2. 値が大きい方の子wの値とvの値を入れ替える
  3. wに対して下降修復を繰り返す.

下降修復 (Downheap)

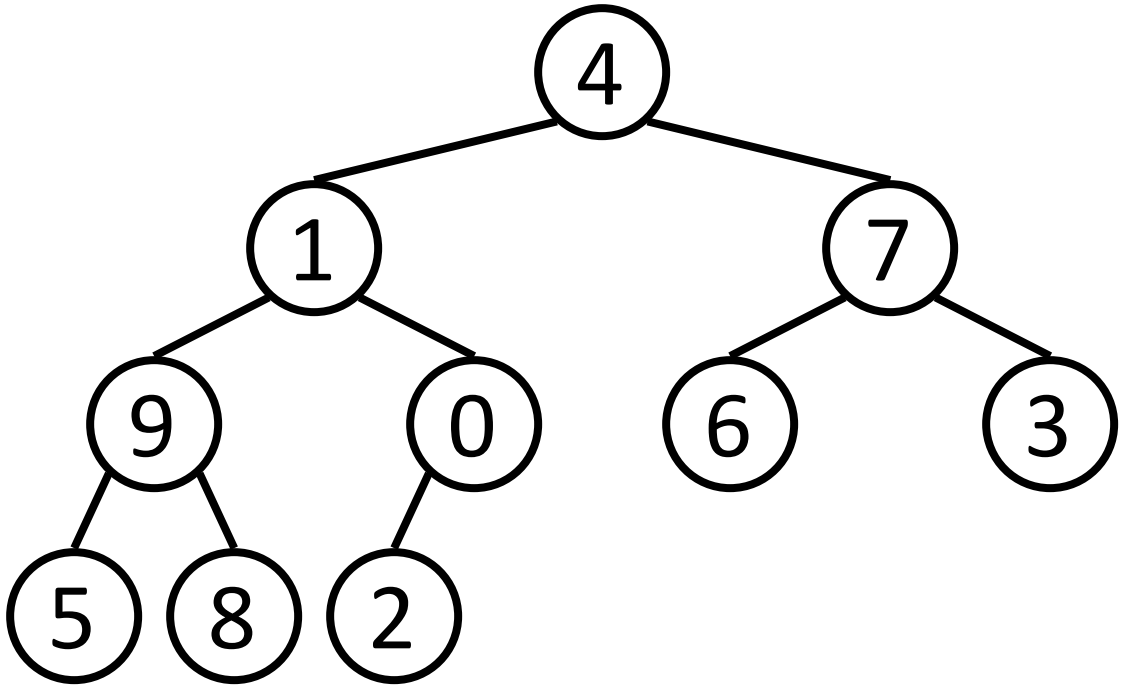


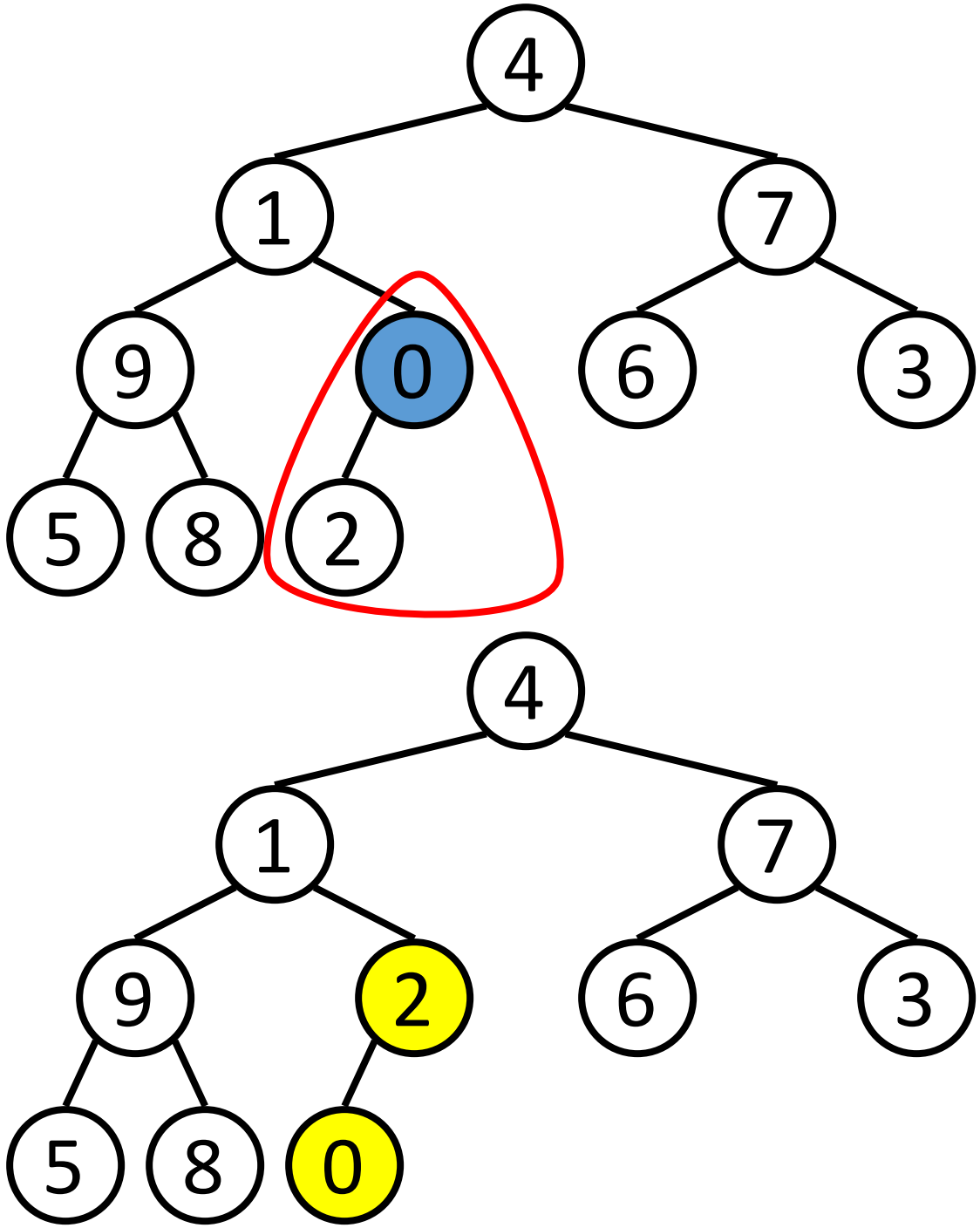
下降修復 (Downheap)

```
if(v>(N/2)) return;
if(右の子がある&&左の子よりも右の子が大きい)
   右の子をwとする;
else
   左の子をwとする;
if(vよりもwが大きい) {
   vとwを交換;
   wを頂点とする部分木に対して下降修復
}
```

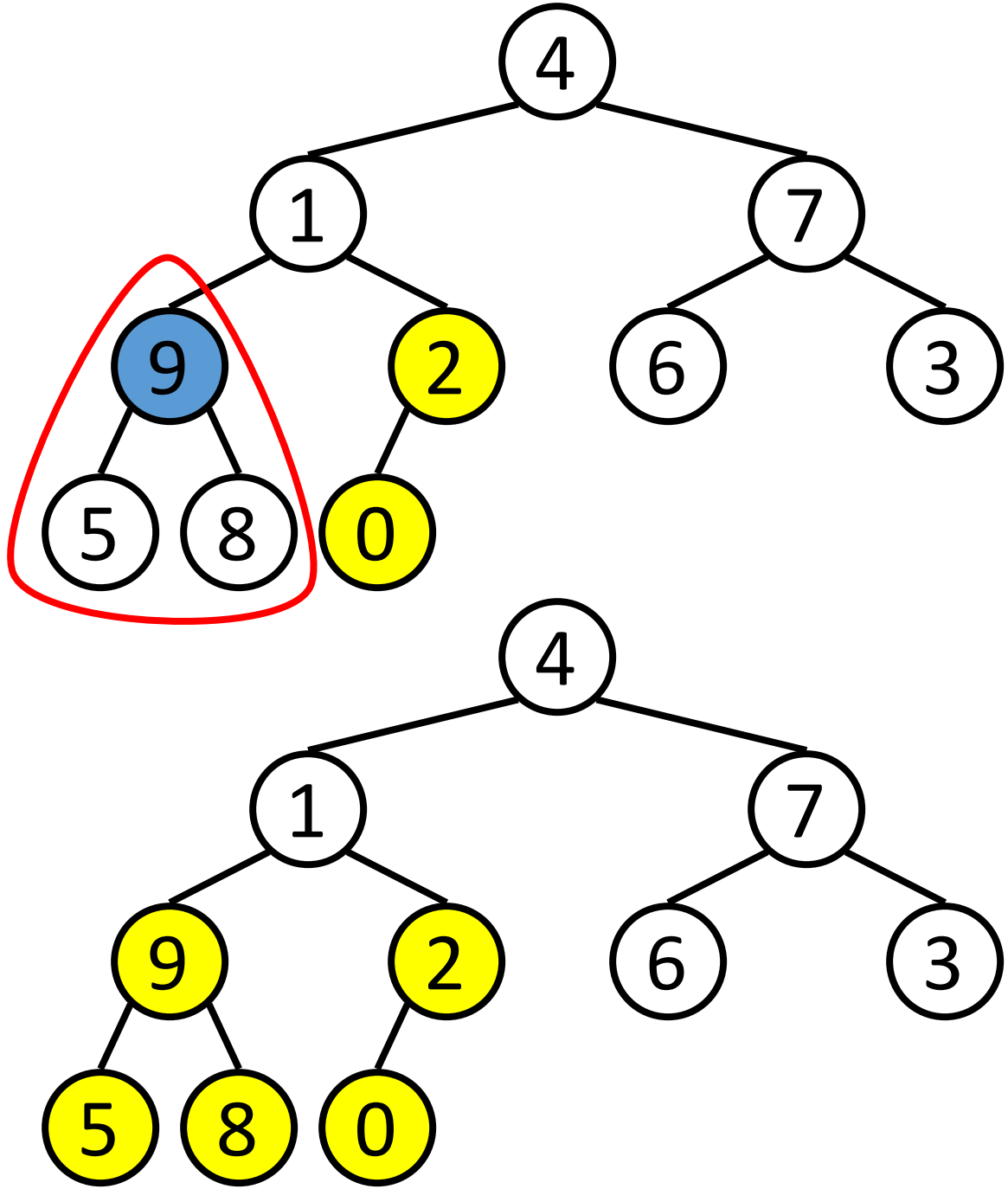
ヒープ化

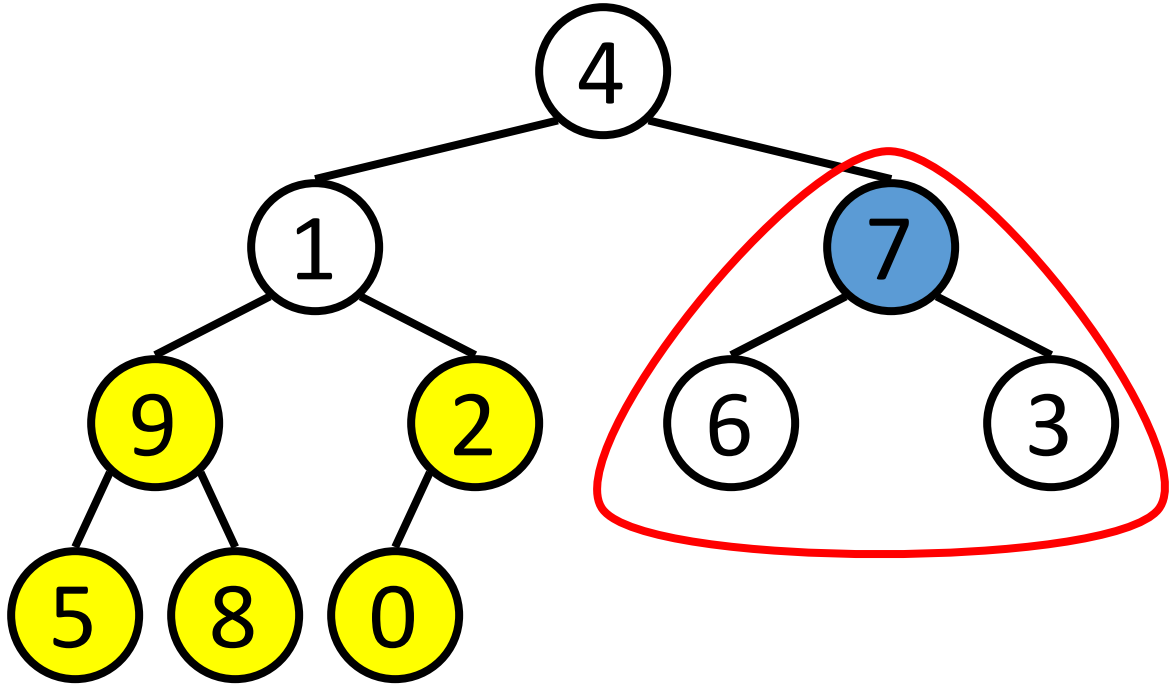
- 大きな木は，下方の部分木から繰り返す．

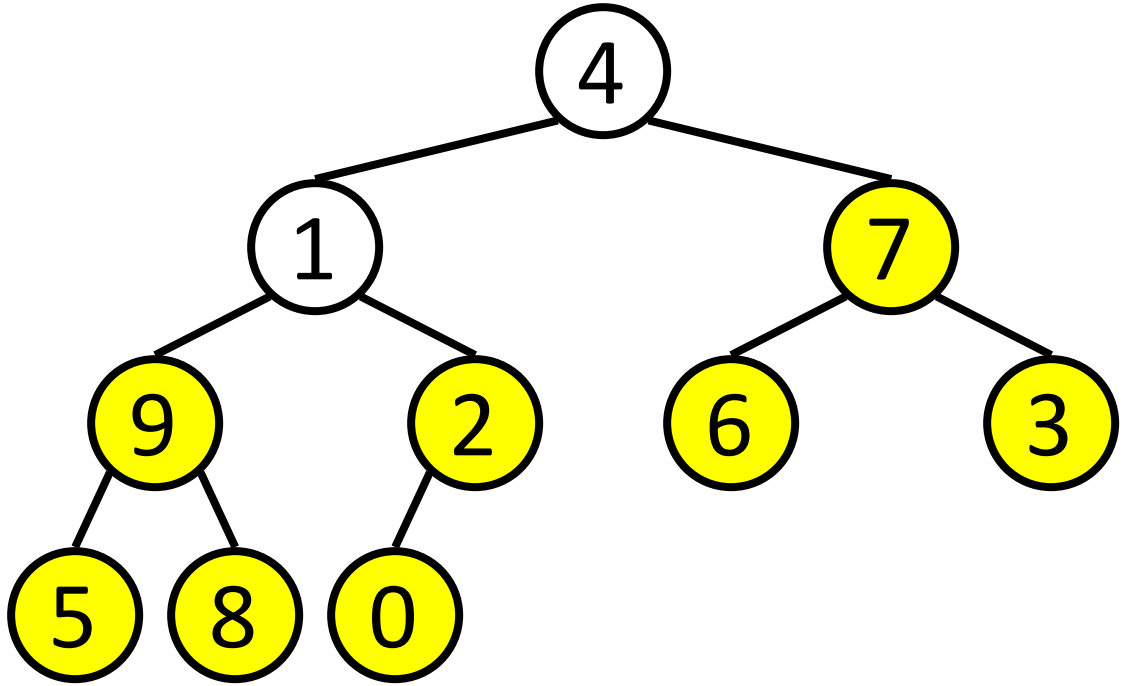


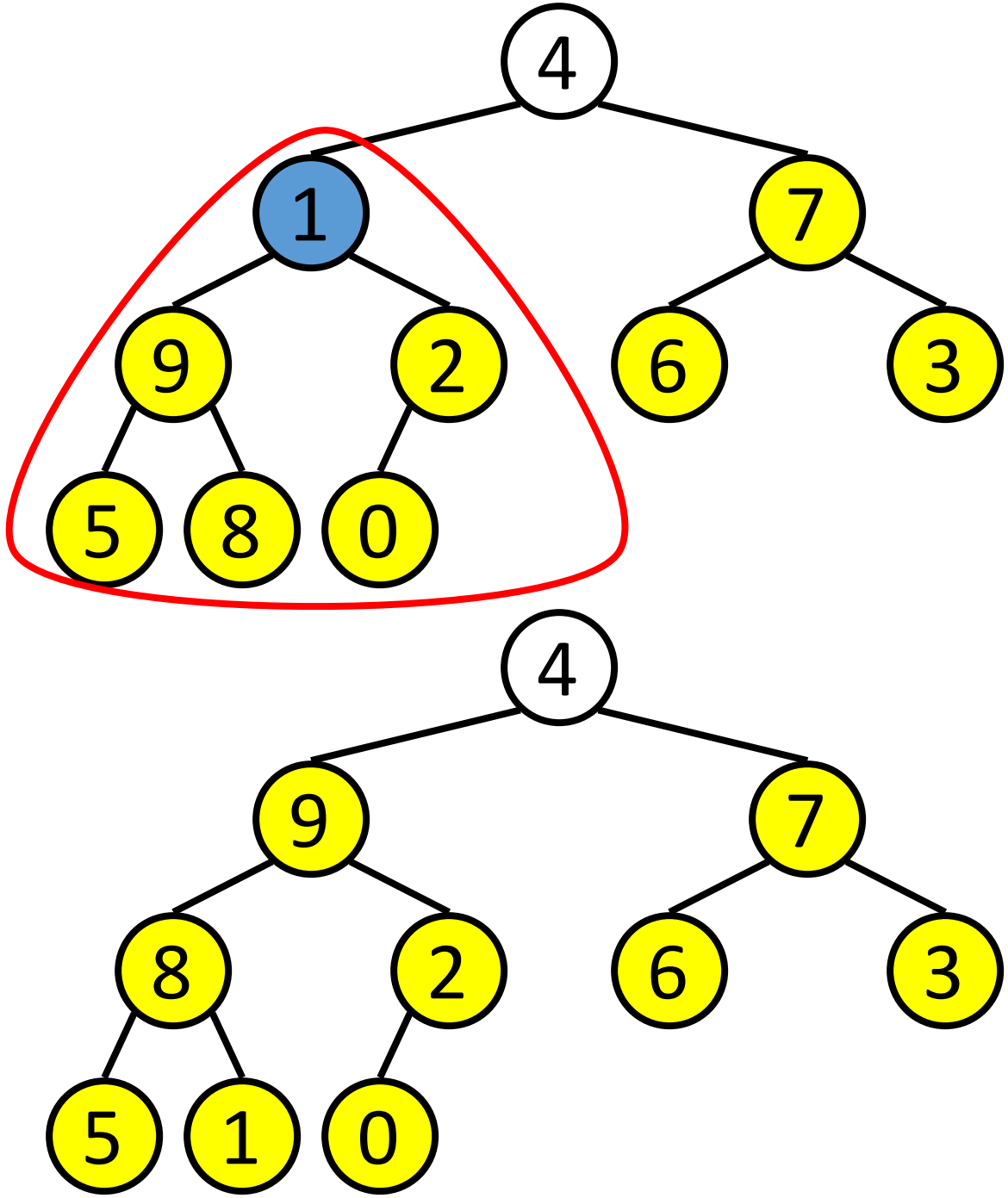


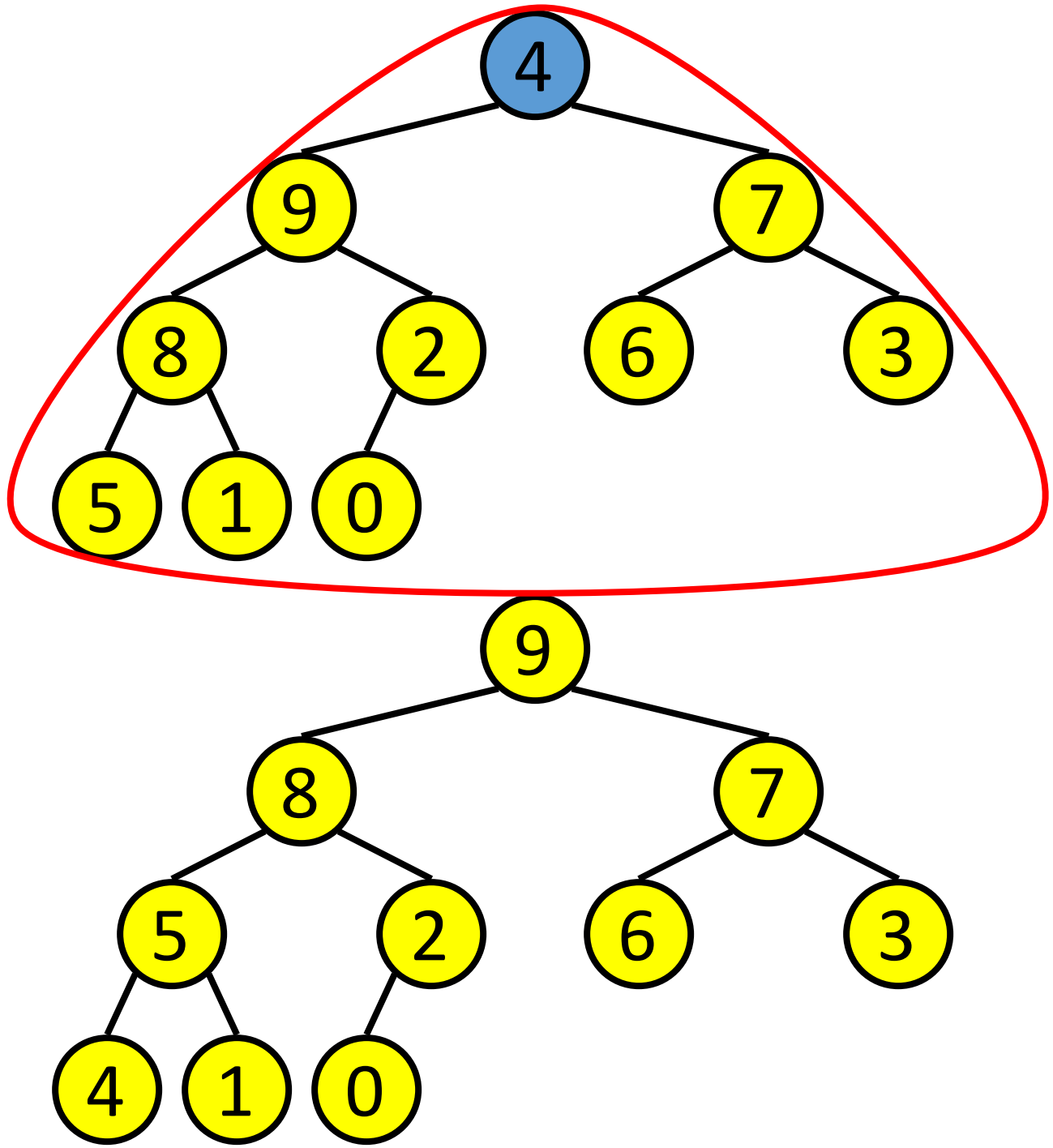
配列


4
1
7
9
2
6
3
5
8
0
4
1
7
9
2
6
3
5
8
0


4
1
7
9
2
6
3
5
8
0


4
1
7
9
2
6
3
5
8
0

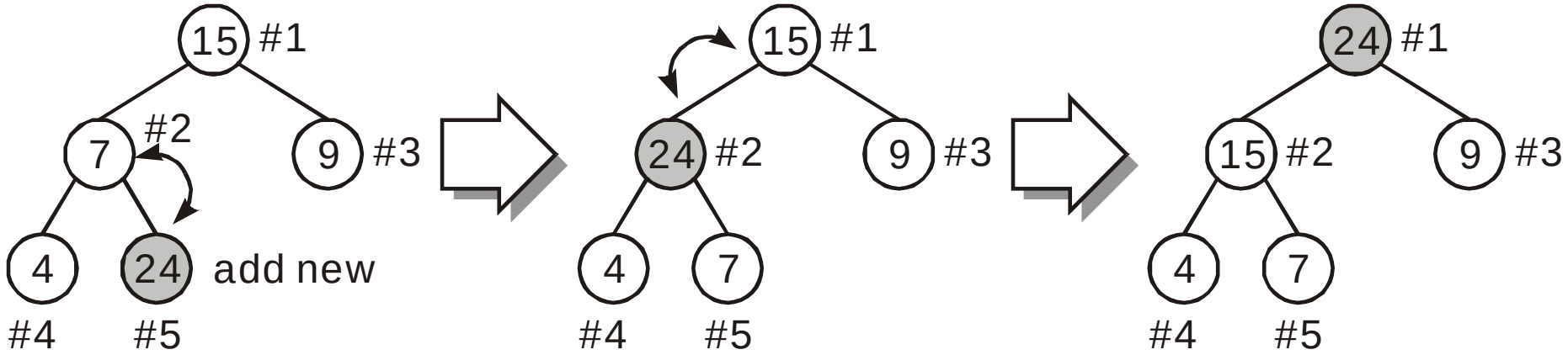

4
1
7
9
2
6
3
5
8
0
4
9
7
8
2
6
3
5
1
0


4
9
7
8
2
6
3
5
1
0
9
8
7
5
2
6
3
4
1
0

上昇修復 (Upheap)

- 新要素を追加した場合，上昇修復を行うことでヒープを復元
  1. ノード**v**の値がその親**u**の値より大きければ
  2. **u**と**v**の値を交換
  3. **u**に対して上昇修復を繰り返す.

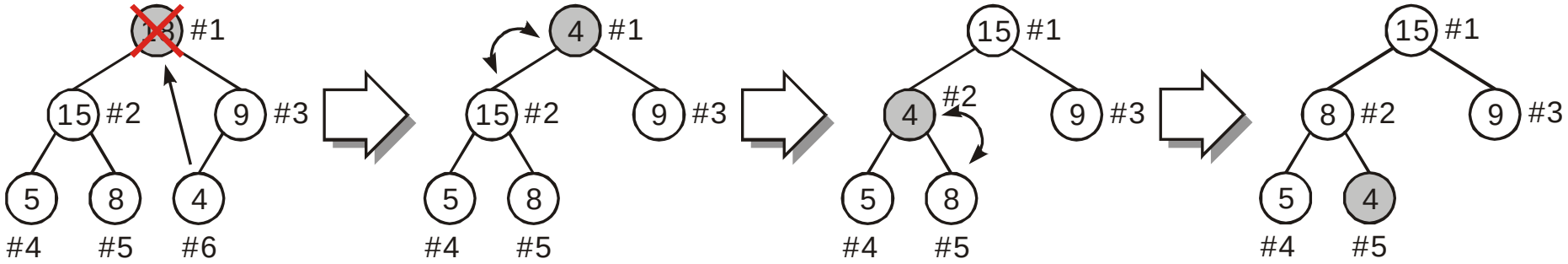
上昇修復 (Upheap)



ノードの削除

1. あるノードを削除した場合
2. **N-1**の要素を削除した部分に
   移動
   ※すなわち，配列の最後の要素
3. そこから下降修復を行うことに
   よりヒープを修復する.

ノードの削除

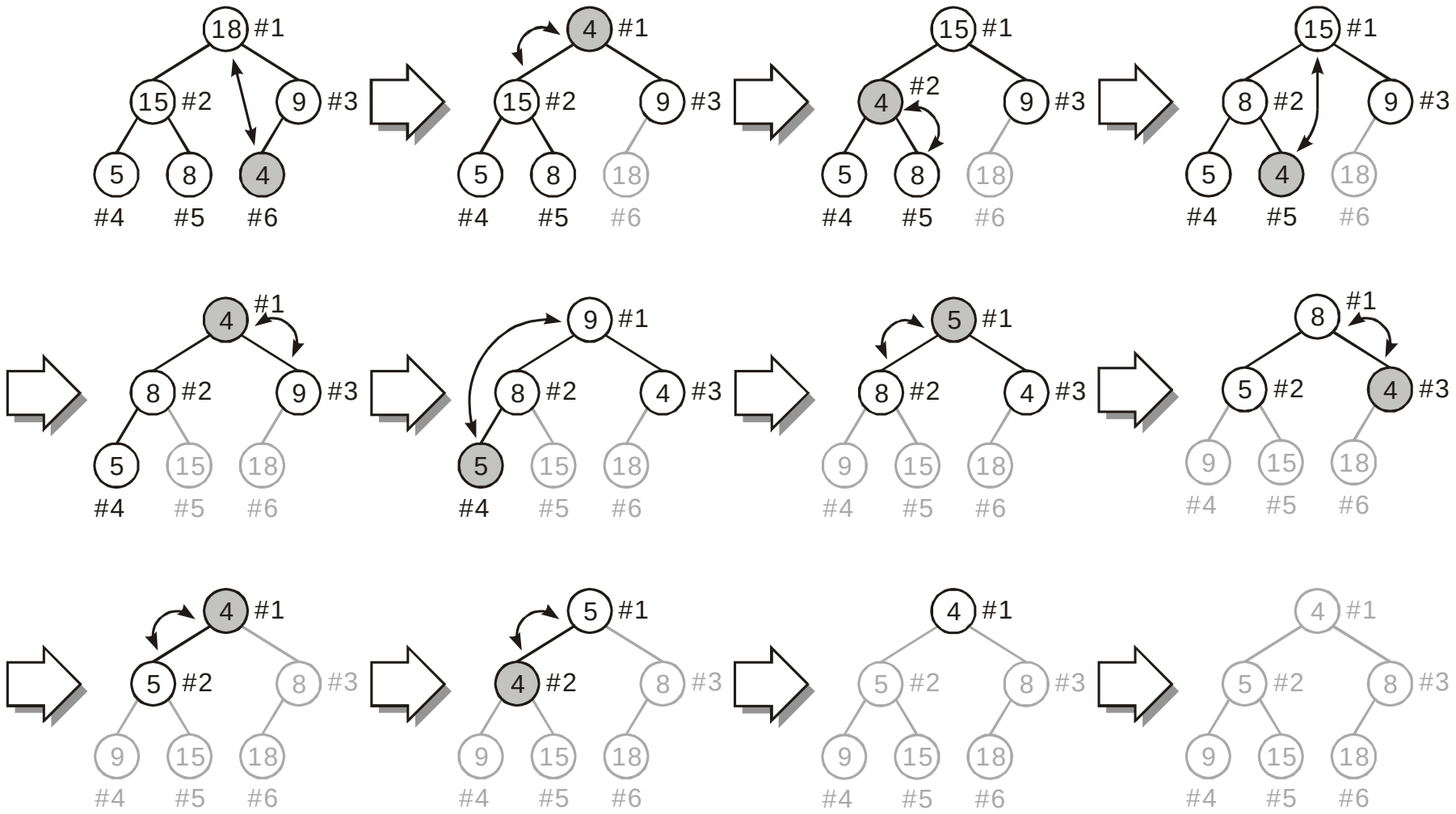


ヒープソート

1. ヒープ化
   - 配列を完全二分木とみなし
     ヒープ化する.
2. ルートの値を取り出す.
   - 最後のノードと交換
     ※ルートの値は最大
3. 残りのノードをヒープ化する.
   - ノードの削除と同様の処理

- 2., 3. を繰り返すことによりソートが可能

ヒープソート


18 #1
15 #2
9 #3
5 8 4
#4 #5 #6
4 #1
15 #2
9 #3
5 8 18
#4 #5 #6
15 #1
4 #2
9 #3
5 8 18
#4 #5 #6
15 #1
8 #2
9 #3
5 4 18
#4 #5 #6
4 #1
8 #2
9 #3
5 15 18
#4 #5 #6
9 #1
8 #2
4 #3
5 15 18
#4 #5 #6
5 #1
8 #2
4 #3
9 15 18
#4 #5 #6
8 #1
5 #2
4 #3
9 15 18
#4 #5 #6
4 #1
5 #2
8 #3
9 15 18
#4 #5 #6
5 #1
4 #2
8 #3
9 15 18
#4 #5 #6
4 #1
5 #2
8 #3
9 15 18
#4 #5 #6
4 #1
5 #2
8 #3
9 15 18
#4 #5 #6

ヒープソートの計算量

- 交換回数
  - ヒープの木，$\log_2 N$段
  - N個の要素に対して操作
  - 最悪$\log_2 N$段分の交換
  - $O(N\log_2 N)$
- 比較回数
  - 交換回数と同じ
  - 左右の比較を行うから2倍
  - $O(2N\log_2 N) \rightarrow O(N\log_2 N)$