

# データ構造とアルゴリズム

## 第5週

掛下 哲郎

kake@is.saga-u.ac.jp

代講: 大月 美佳

mika@is.saga-u.ac.jp

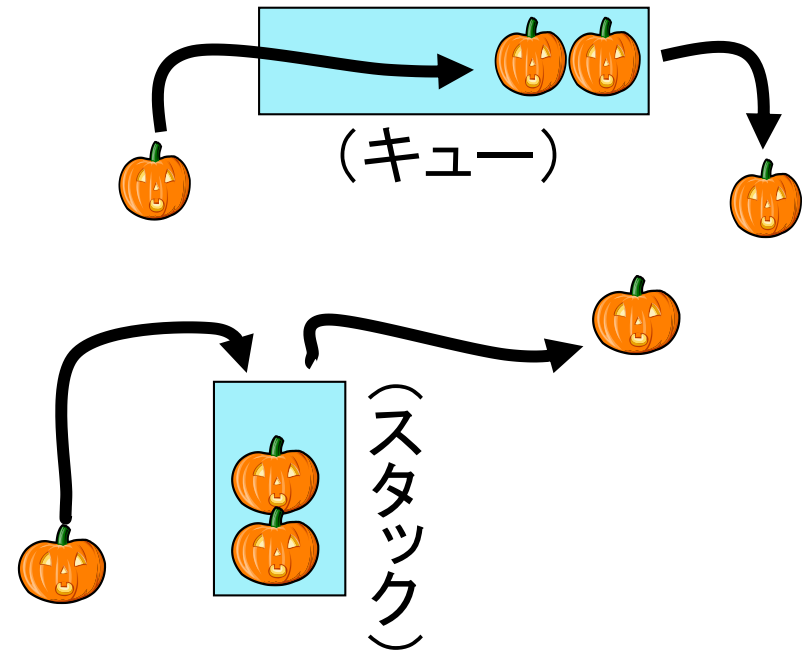
# 前回のまとめ: その1

## データ構造の実装法

- **配列** (Array)
  - 長所: データアクセスは  $O(1)$  で高速
  - 短所: データが大小順に並ぶ場合は, 挿入・削除とも最悪  $O(n)$  で遅い.
  - 最初に定義した大きさ以上にデータを格納できない
- **連結リスト** (Linked List)
  - 長所: 場所が分かっているならば, 挿入・削除が  $O(1)$  で高速
  - 長所: データ容量に関する制限が事実上ない
  - 短所:  $k$ 番目のデータにアクセスするには, 先頭から  $k$ 個順番に辿る → 最悪の場合  $O(n)$

# 前回のまとめ: その2

- データの挿入・取り出しに特徴のあるデータ構造
  - キュー (queue)
    - FIFO (先入れ先出し)
    - 挿入・取り出しともに  $O(1)$
  - スタック (stack)
    - LIFO (後入れ先出し)
    - 挿入・取り出しともに  $O(1)$



参考: 教材ページの操作手順

# 講義スケジュール

週	講義計画
1-2	導入
3	探索問題
4-5	基本的なデータ構造
6	動的探索問題とデータ構造
7	アルゴリズム演習(第1回)
8-9	データの整列
10-11	グラフアルゴリズム
12	文字列照合のアルゴリズム
13	アルゴリズム演習(第2回)
14	アルゴリズムの設計手法
15	計算困難な問題への対応



今日学ぶこと

## 基本的なデータ構造

- 配列
- 連結リスト
- キュー
- スタック
- ヒープ
- 二分探索木

# ヒープ (heap)

- スタックやキューとの違い:
  - 現在蓄えられているデータの中で**最大のものを取り出す**
- 配列で実現しようとするすると $O(n)$ かかる
  - 2分探索を利用すれば最大値の取り出しは $O(\log n)$ で済むが、挿入に $O(n)$ かかる(順番に並べる必要があるため)。
- 工夫すれば、挿入・取り出しともに $O(\log n)$ 
  - **木**を利用

# 木 (tree)

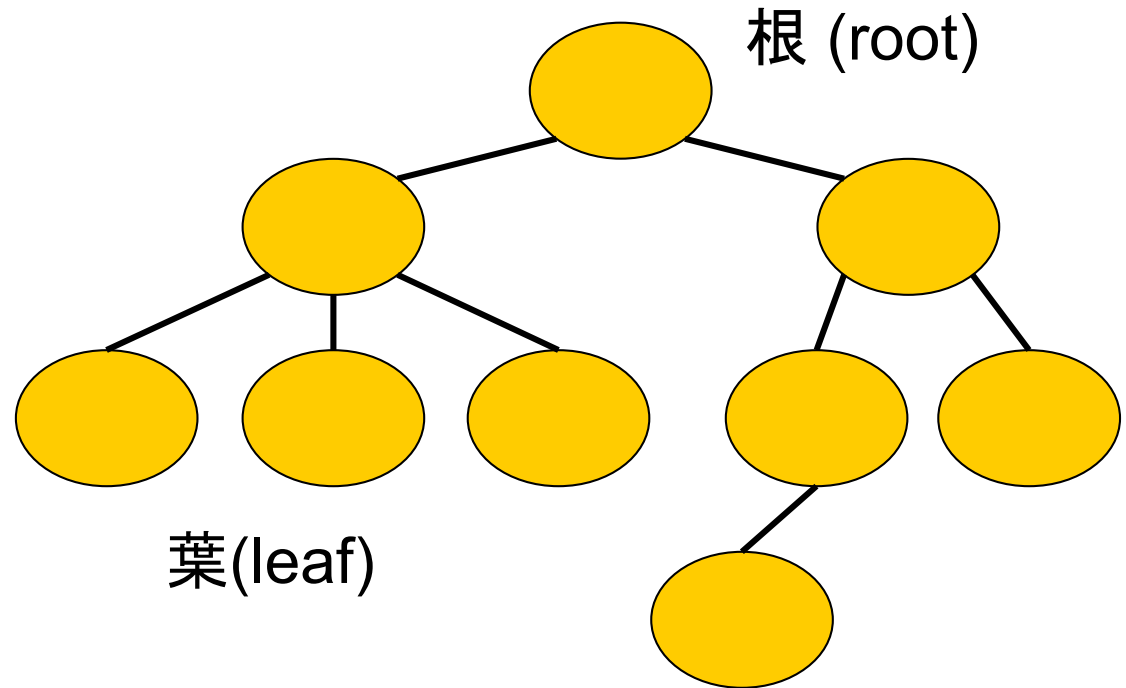
(用語)

● 節点 (node)

| 枝 (edge)

● 親 (parent)

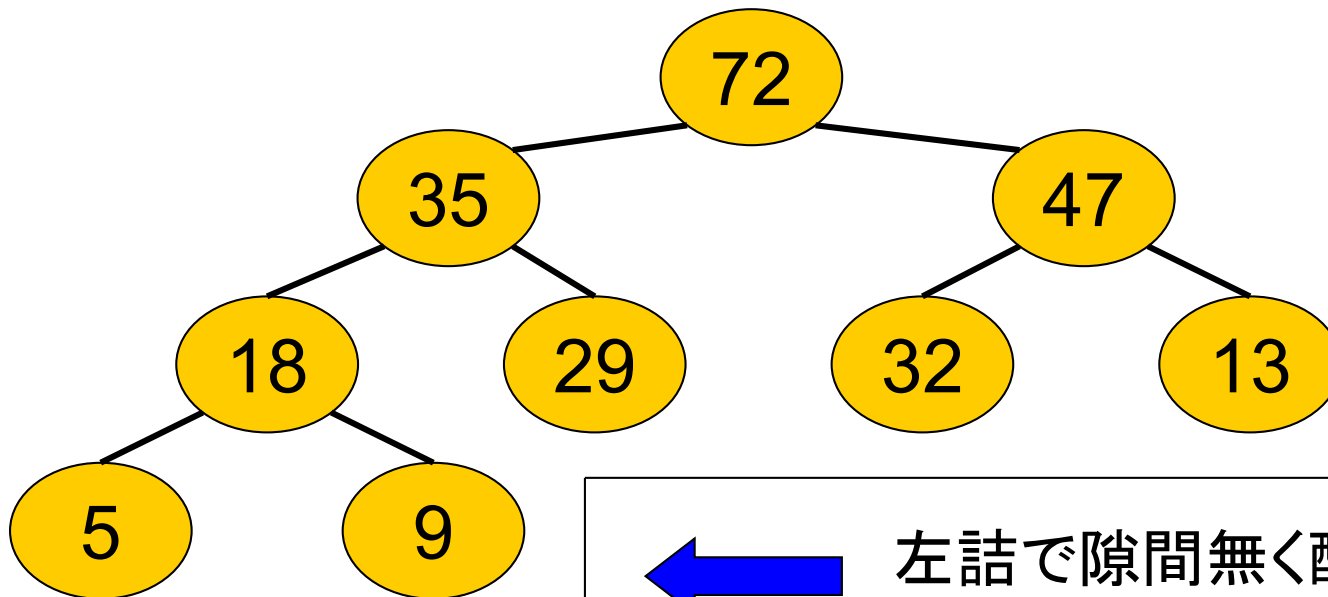
● 子 (child)



# ヒープの定義

1. 2分木: 各節点には高々2個の子
2. 親のデータ > 子のデータ
3. 左詰めで隙間なくデータを配置

木の高さは  $\log n$   
で抑えられる

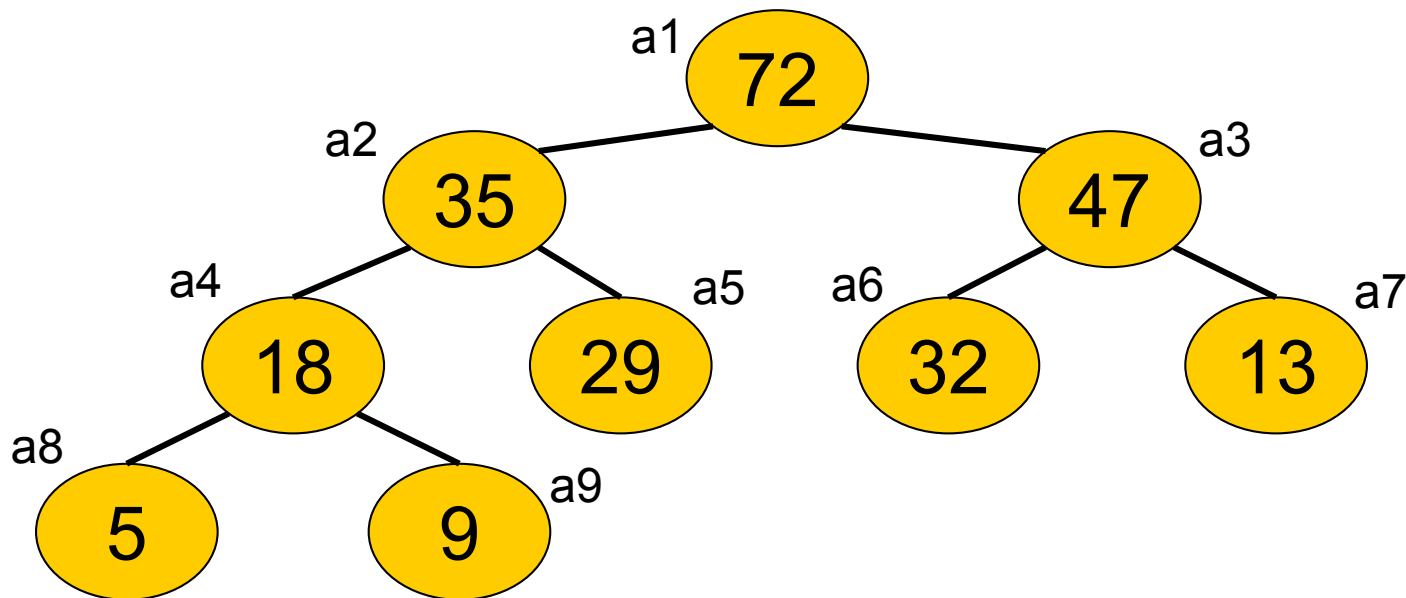


左詰で隙間無く配置  
(完全2分木)

# 配列を用いたヒープの表現

$a_k$ の子は  $a_{2k}$  と  $a_{2k+1}$   $\rightarrow$  配列に隙間なく配置できる

a1	a2	a3	a4	a5	a6	a7	a8	a9	-	-	-
----	----	----	----	----	----	----	----	----	---	---	---





# ヒープにおける操作

## データの取り出し

- 根のデータが最大値  
→ 根のデータを取り出せば良い  
(が、取り出したデータの削除はどうする?)

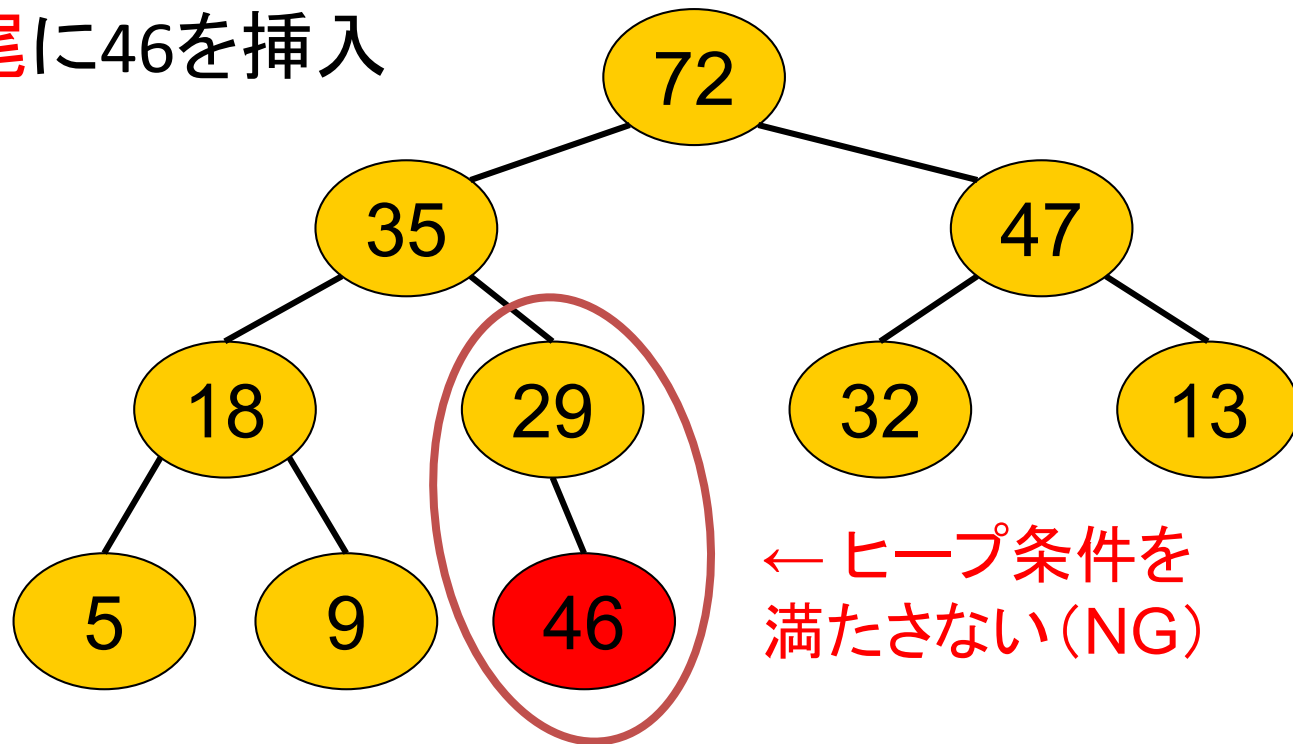
## データの挿入

- どこに入れる?

どうやって  
実現する?

# ヒープへのデータ挿入例(その1)

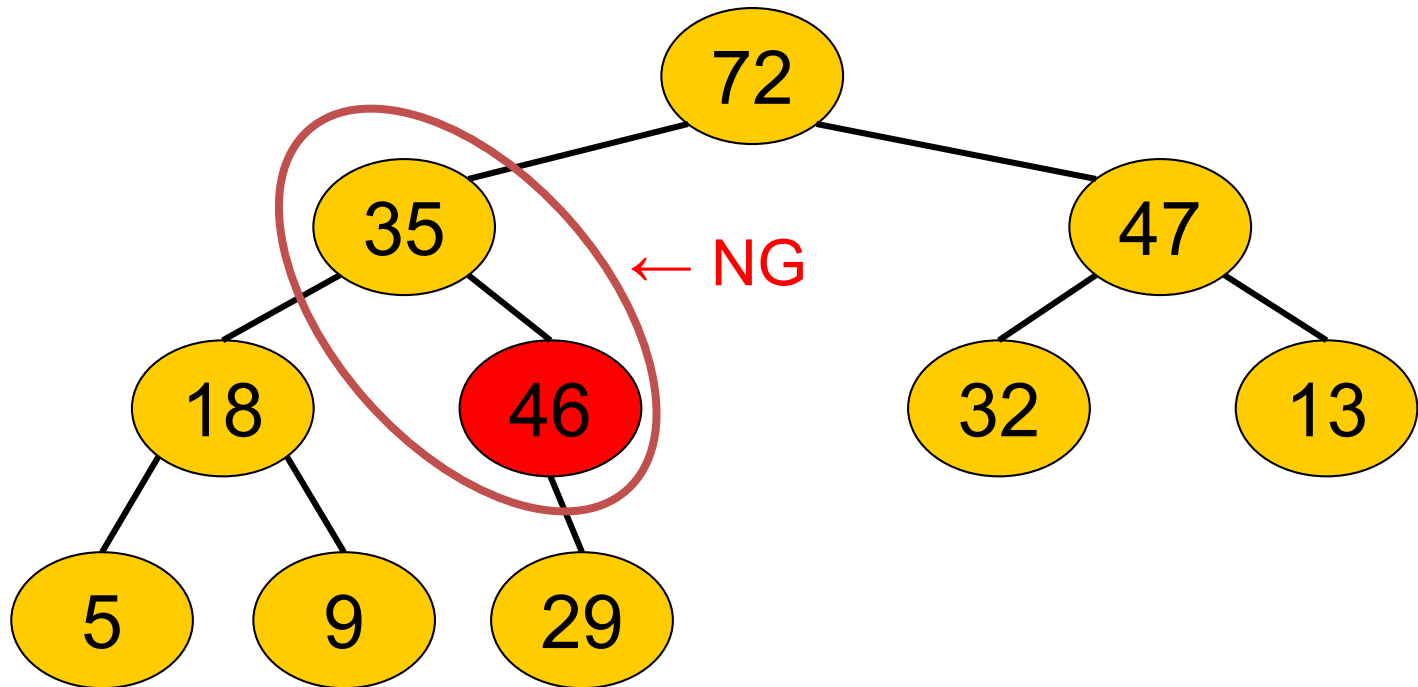
- 例: 46を挿入したい
- とりあえず,  
**最後尾**に46を挿入



ヒープ条件: 親のデータ > 子のデータ

# ヒープへのデータ挿入例(その2)

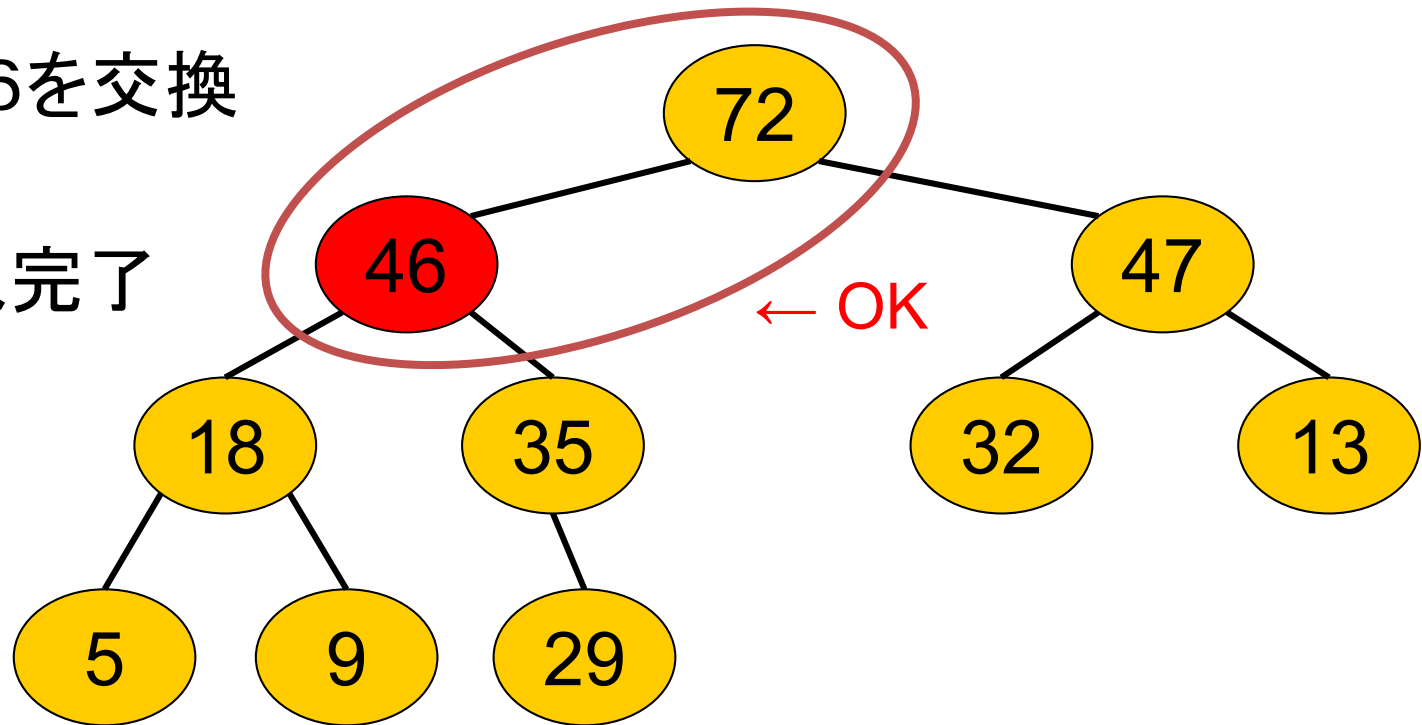
- 29と46を交換



# ヒープへのデータ挿入例(その3)

- 35と46を交換

→ 挿入完了



一般  
ルール

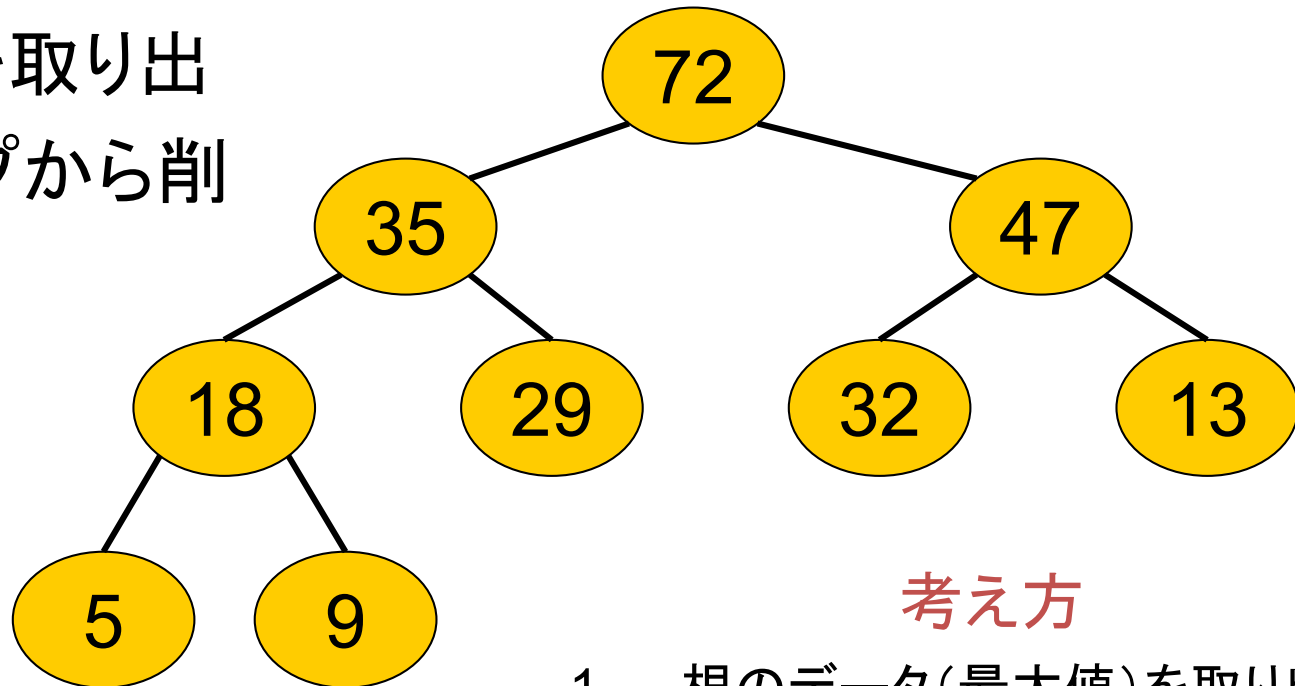
1. 挿入したいデータを最後尾に追加
2. ヒープ条件(親 $>$ 子)を満たすように親子を交換

# データ挿入の効率

- 最悪の場合, 木の一番下から根まで, 順次交換していく
  - 1回の操作は, 比較とデータ交換  $\rightarrow O(1)$
- 繰り返し回数
  - データ数が $n$ の完全2分木の高さ  
 $\rightarrow \log n$  で抑えられる
- よって,  $O(\log n)$

# ヒープからのデータの取り出し

最大値を取り出し、  
ヒープから削除したい

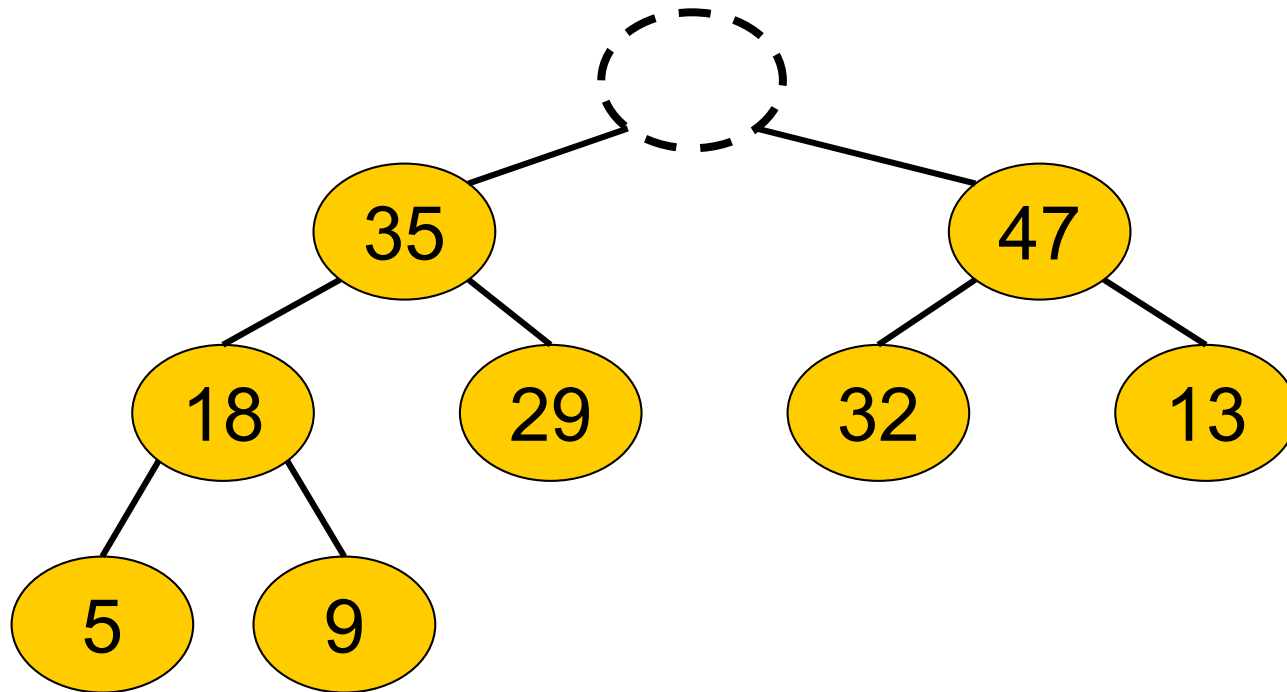


## 考え方

1. 根のデータ(最大値)を取り出す
2. 最後尾のデータを(暫定的に)根に据える
3. 条件(親子間のデータの大小関係)を満たすように親子を交換

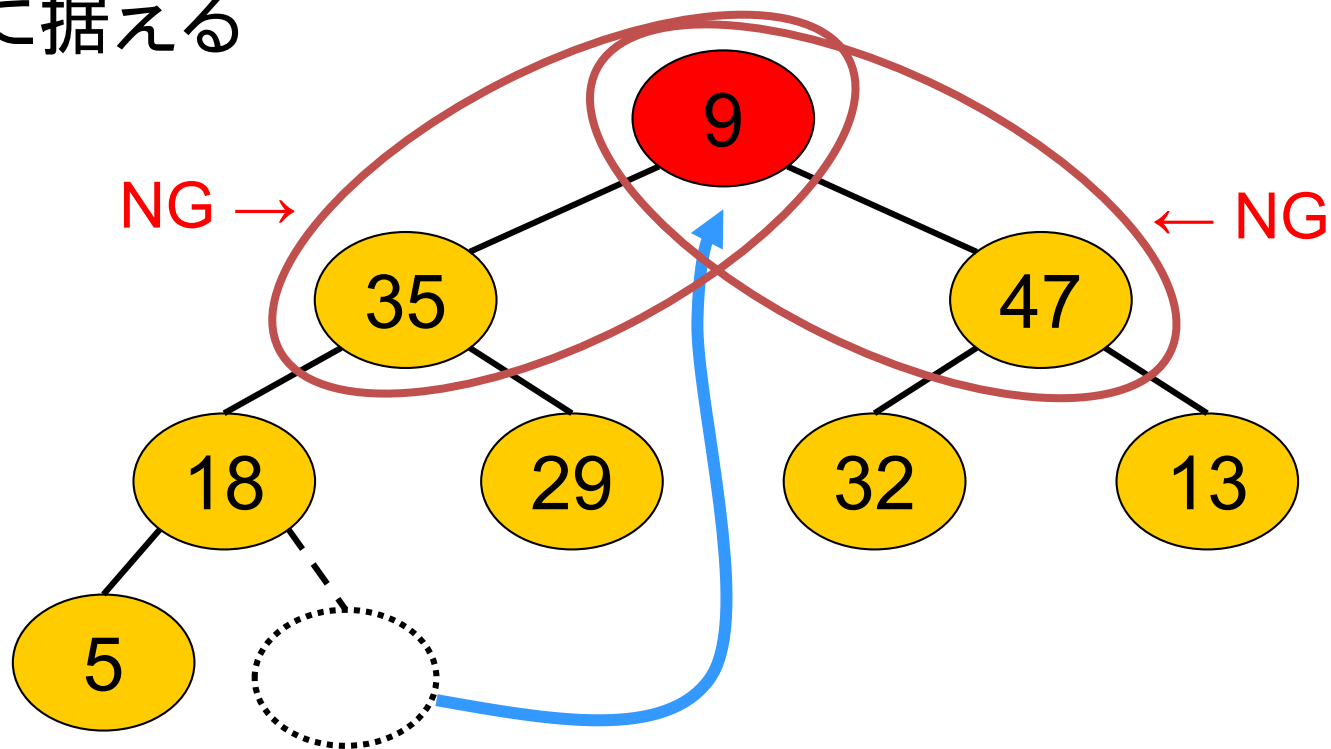
# データ取り出しの例(その1)

- 72(最大値)を取り出す



## データ取り出しの例(その2)

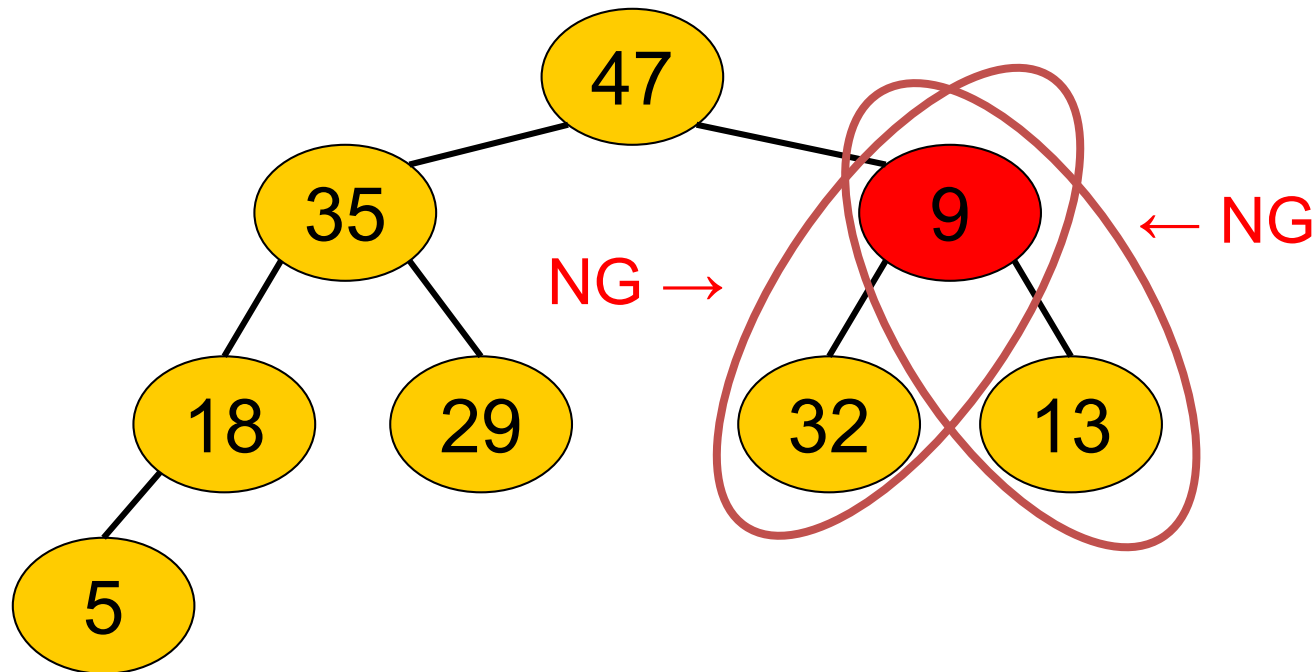
- 9(最後尾)を, 暫定的に根に据える





# データ取り出しの例(その3)

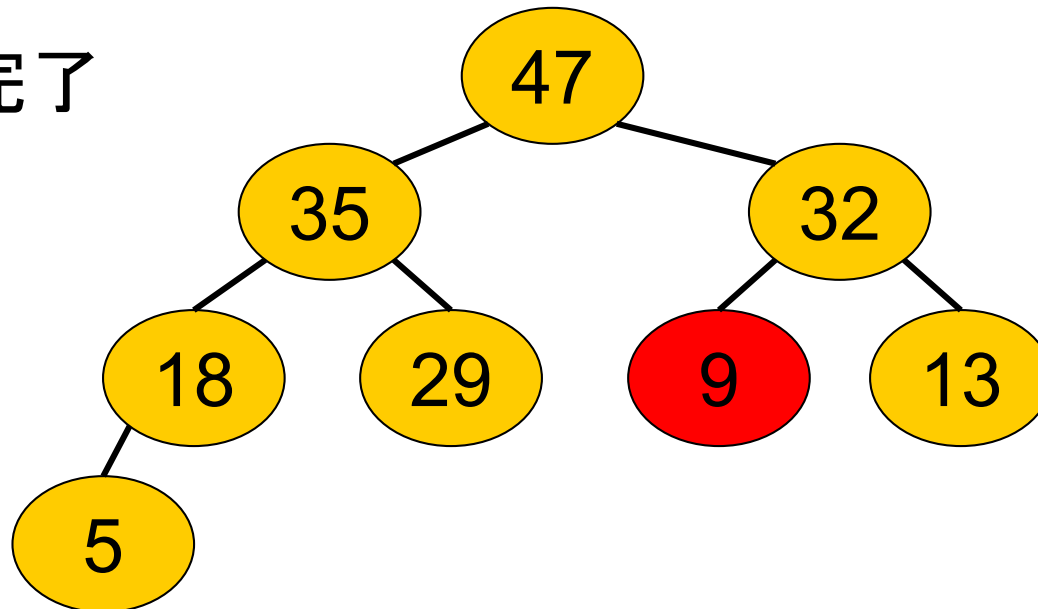
- 9と47を交換
  - 9と35を交換してはいけない
  - 子ノードのうち、**大きい方**と交換



# データ取り出しの例(その4)

- 9と32を交換

→ 調整完了



# データ取り出しの効率

- 最悪の場合、木の根から一番下まで、順次交換していく
  - 1回の交換は、2回の比較とデータの入れ替え →  $O(1)$
- データ数が $n$ の完全2分木の高さ  
→  $\log n$  で抑えられる
- よって、 $O(\log n)$

# ヒープソート(Heap Sort)

## ヒープを活用した並べ替えアルゴリズム

### アルゴリズム

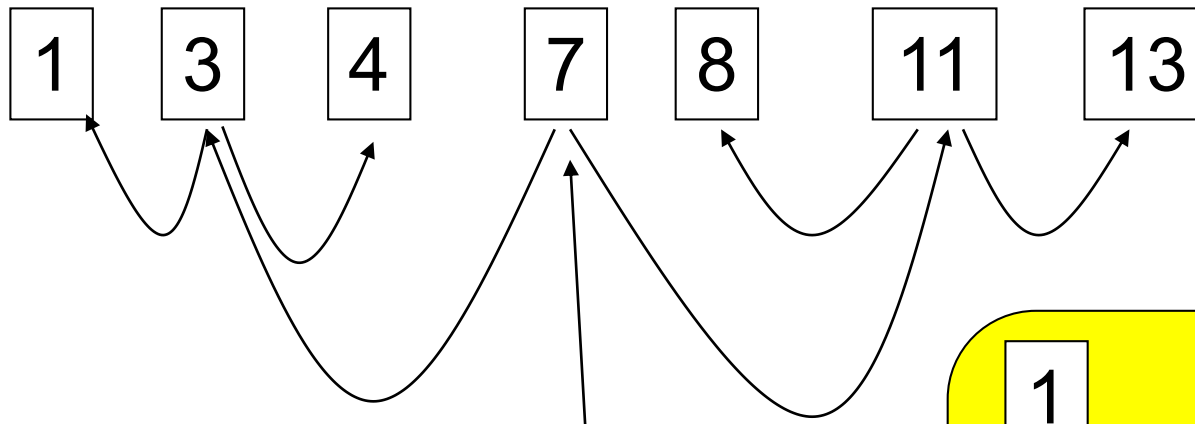
1. 空のヒープを用意する.  $O(1)$
2. 入力された各データをヒープに挿入する. 反復回数  $n \times$  ヒープへの挿入  $O(\log n)$
3. ヒープが空でない限り, 以下の処理を繰り返す. 反復回数  $n$ 
  - 3-1. ヒープから最大値を取り出し, それを表示する. ヒープからの取り出し  $O(\log n)$

### アルゴリズムの効率

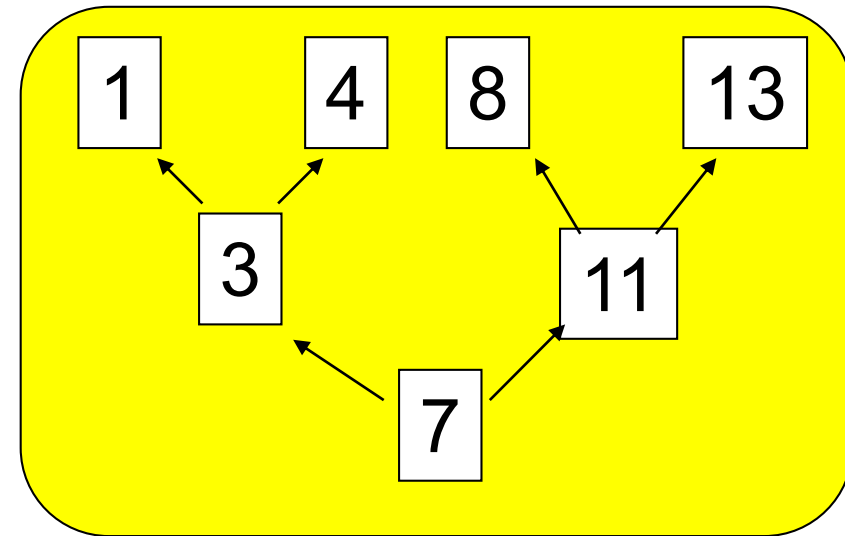
$O(n \log n)$   $n$ : データ数

# 二分探索木 (Binary Search Tree)

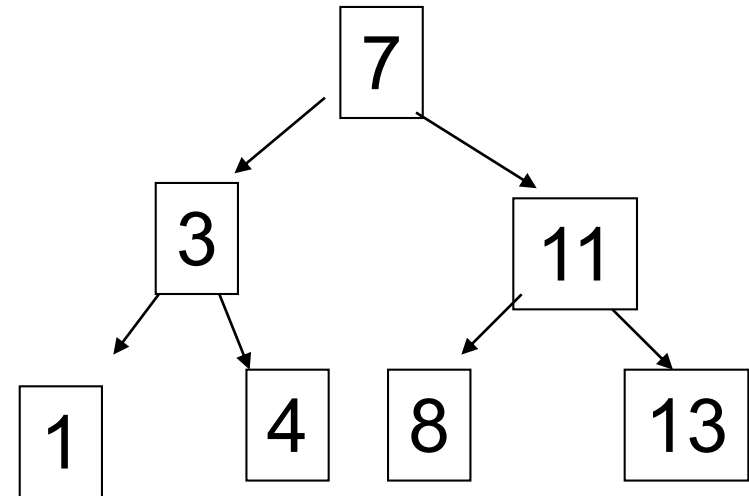
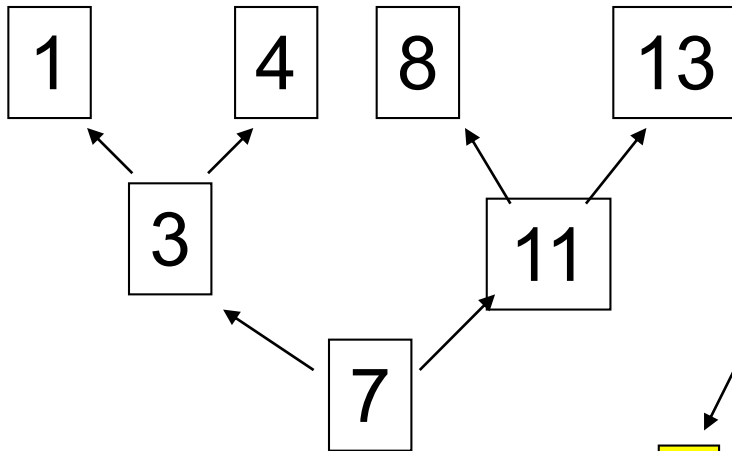
- 二分探索に適したデータ構造



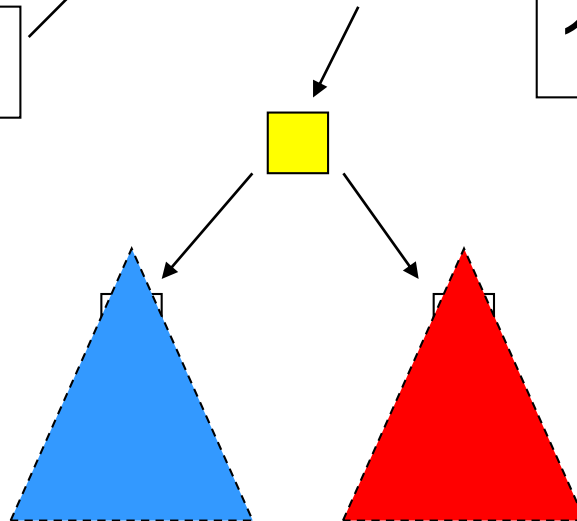
高さ $k$ の二分探索木は $2^{k-1}-1$ 個  
のデータを保持  
→ 二分探索木の高さは  
データ数 $n$ に対して $O(\log n)$



# 二分探索木

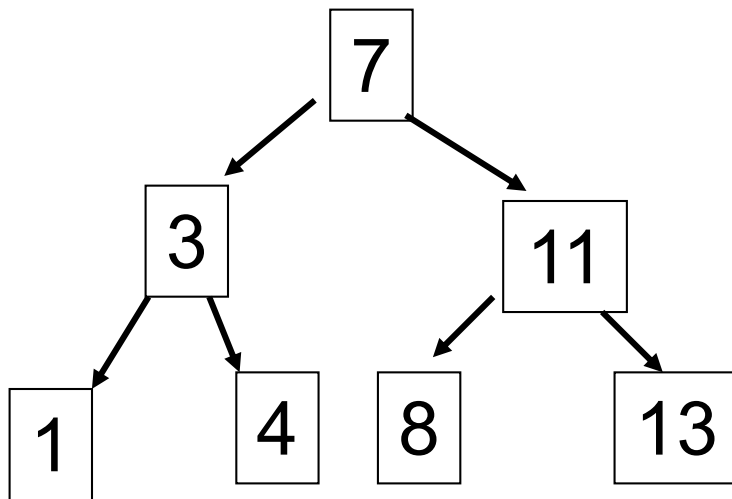


(通常はこのように  
根を上を書く)



各節点での大小関係

# 二分探索木上でのデータ探索

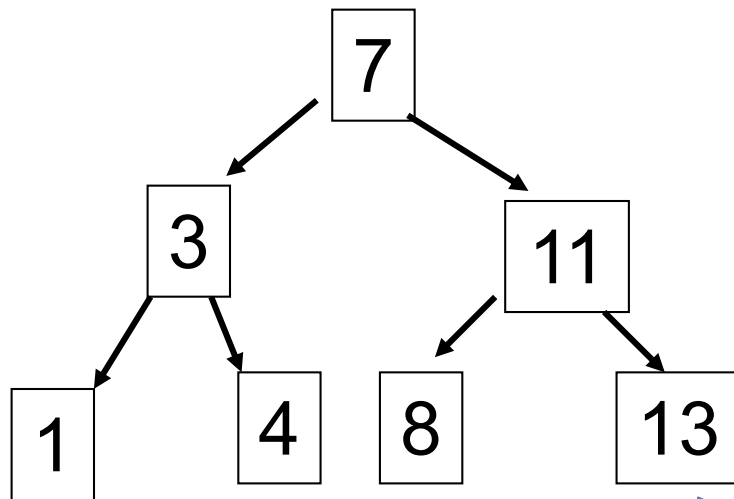


二分探索木の高さはデータ数 $n$ に対して $O(\log n)$   
→ データ探索アルゴリズムの効率も $O(\log n)$

## アルゴリズム

1. 根節点からスタート
2. 葉節点に到達するまで以下の処理を繰り返す.
  - 2-1. 現在の節点がデータを保持しているならば検索成功として終了.
  - 2-2. 検索したい値が現在の節点よりも小さいならば, 左の子に移る
  - 2-3. そうでなければ, 右の子に移る.
3. 検索失敗とする.

# 二分探索木の最大値と最小値



最小値

最大値

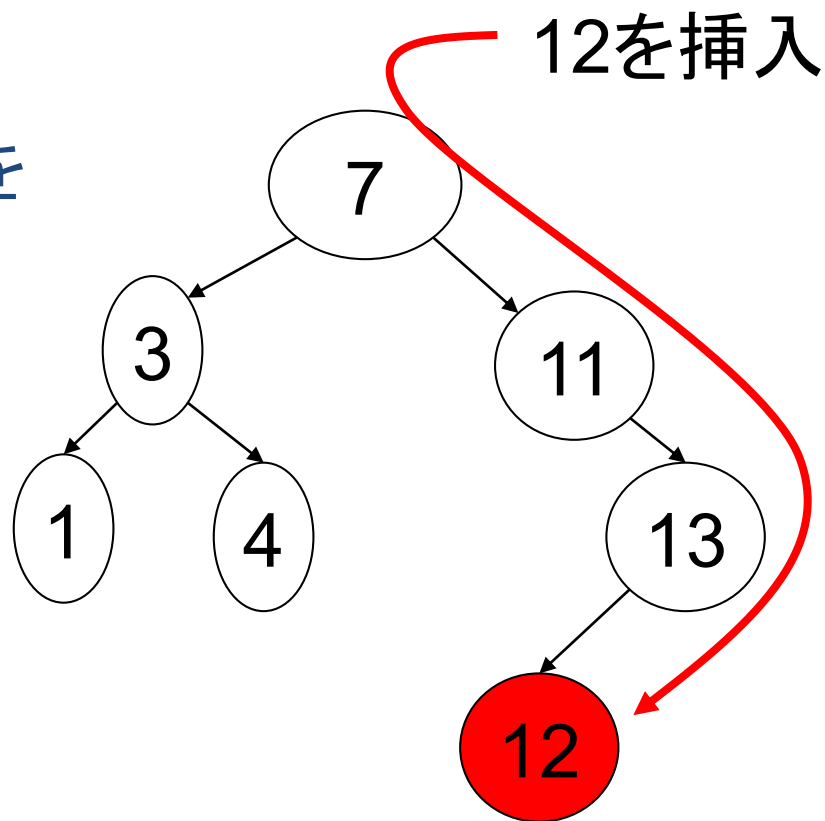
二分探索木の高さはデータ数 $n$ に対して $O(\log n)$   
→ 最大値/最小値を求めるアルゴリズムの効率も $O(\log n)$

- 根から始めて, **右の子**を辿れるだけ辿る ⇒ **最大値**
- 根から始めて, **左の子**を辿れるだけ辿る ⇒ **最小値**



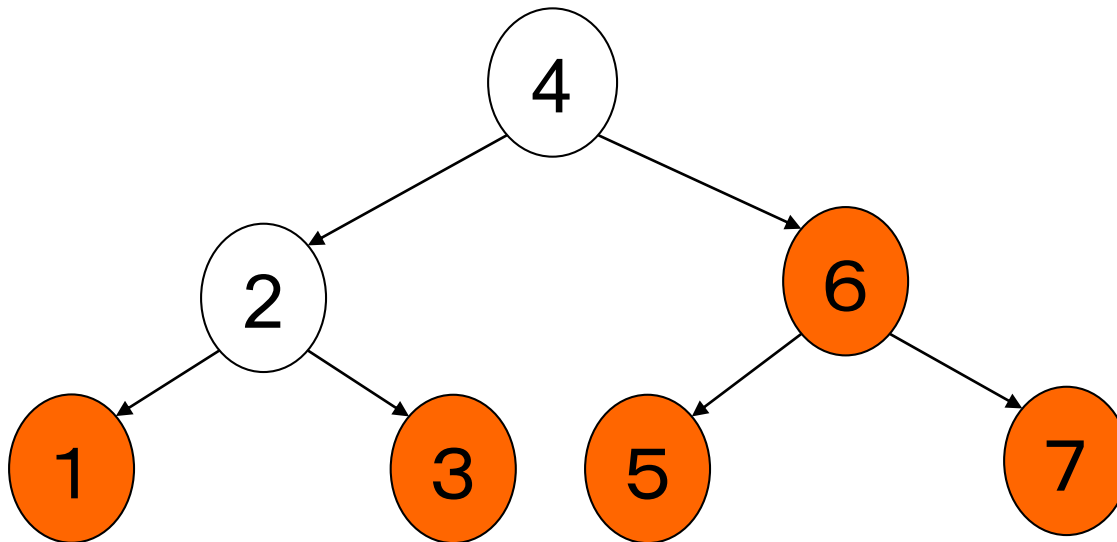
# 二分探索木への挿入

- 挿入の考え方
  - 値を挿入すべき場所を「探索」
  - 探索した場所に挿入



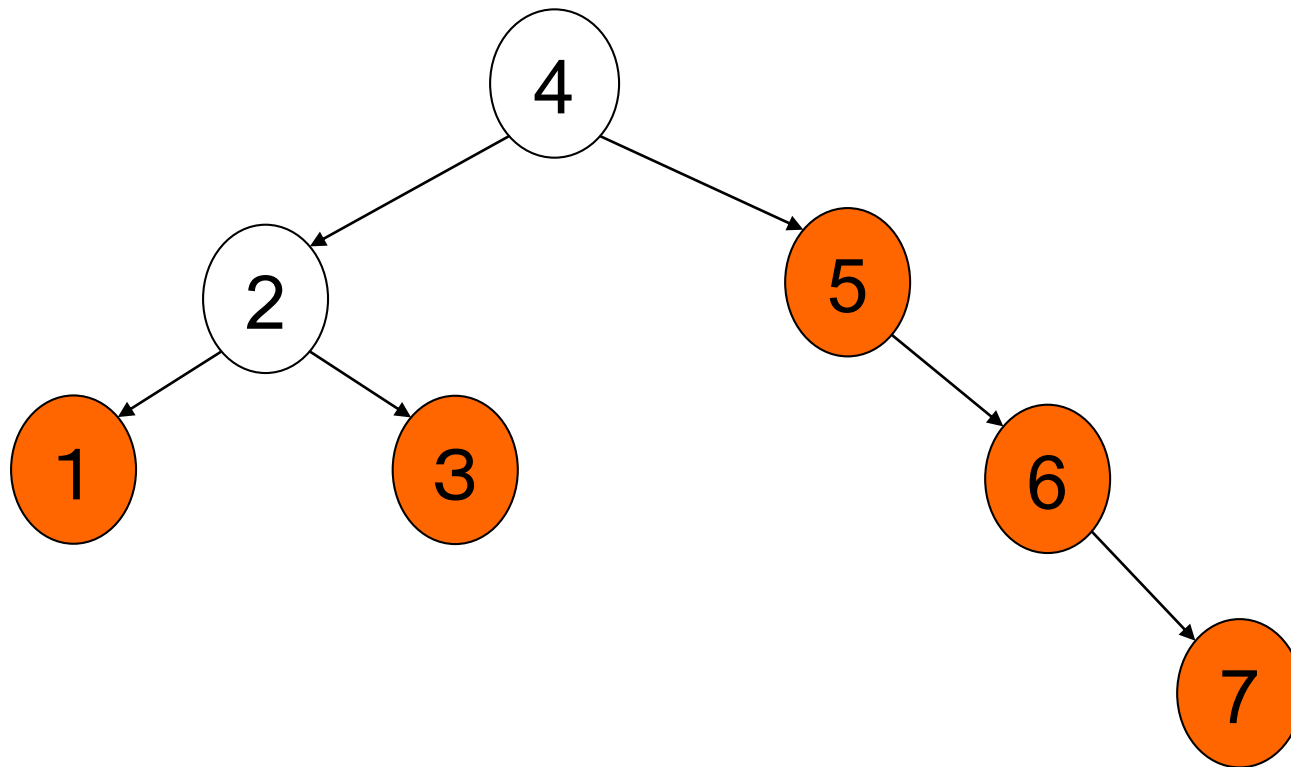
# Q1

- 以下の二分探索木に、1,3,6,7,5の順にデータを挿入



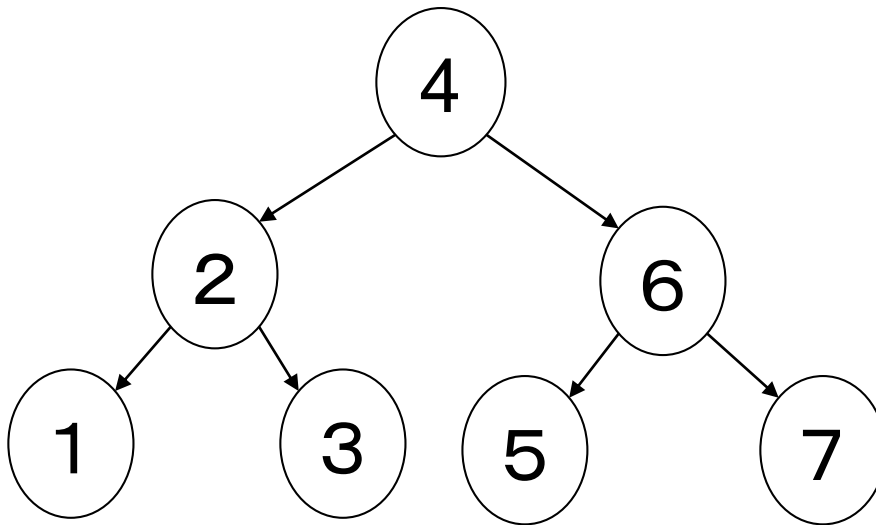
## Q2

- 以下の二分探索木に、1,3,5,6,7の順にデータを挿入

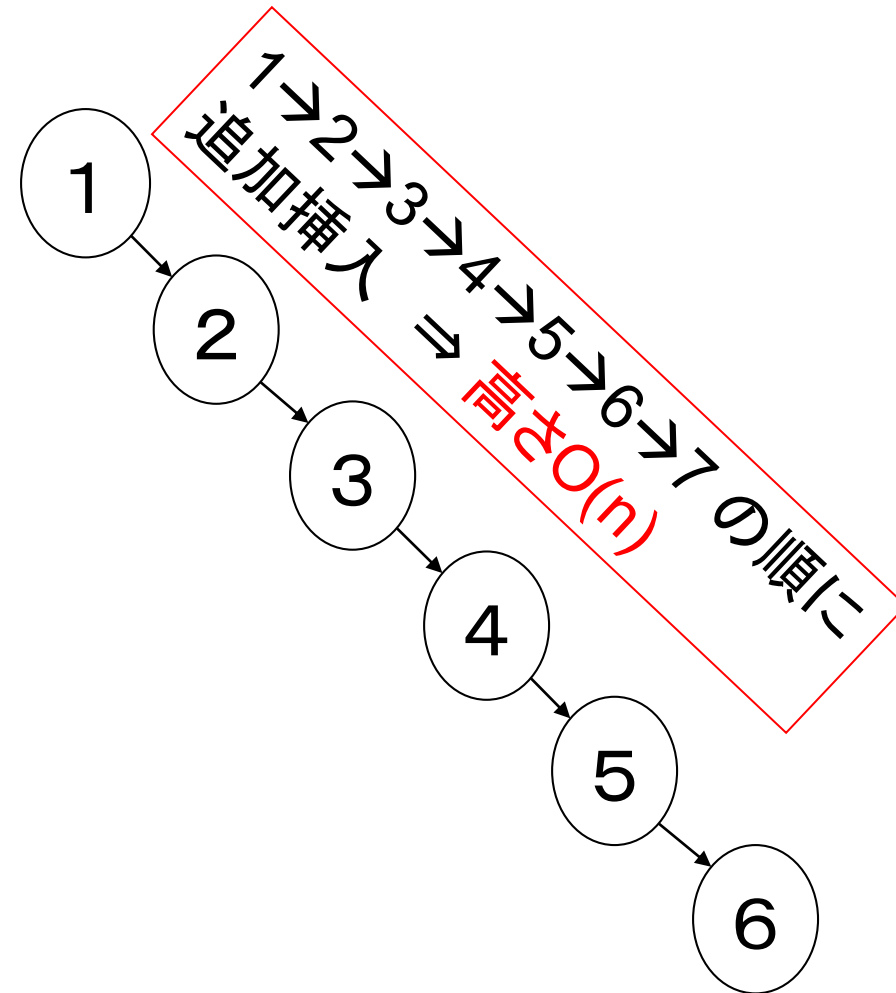


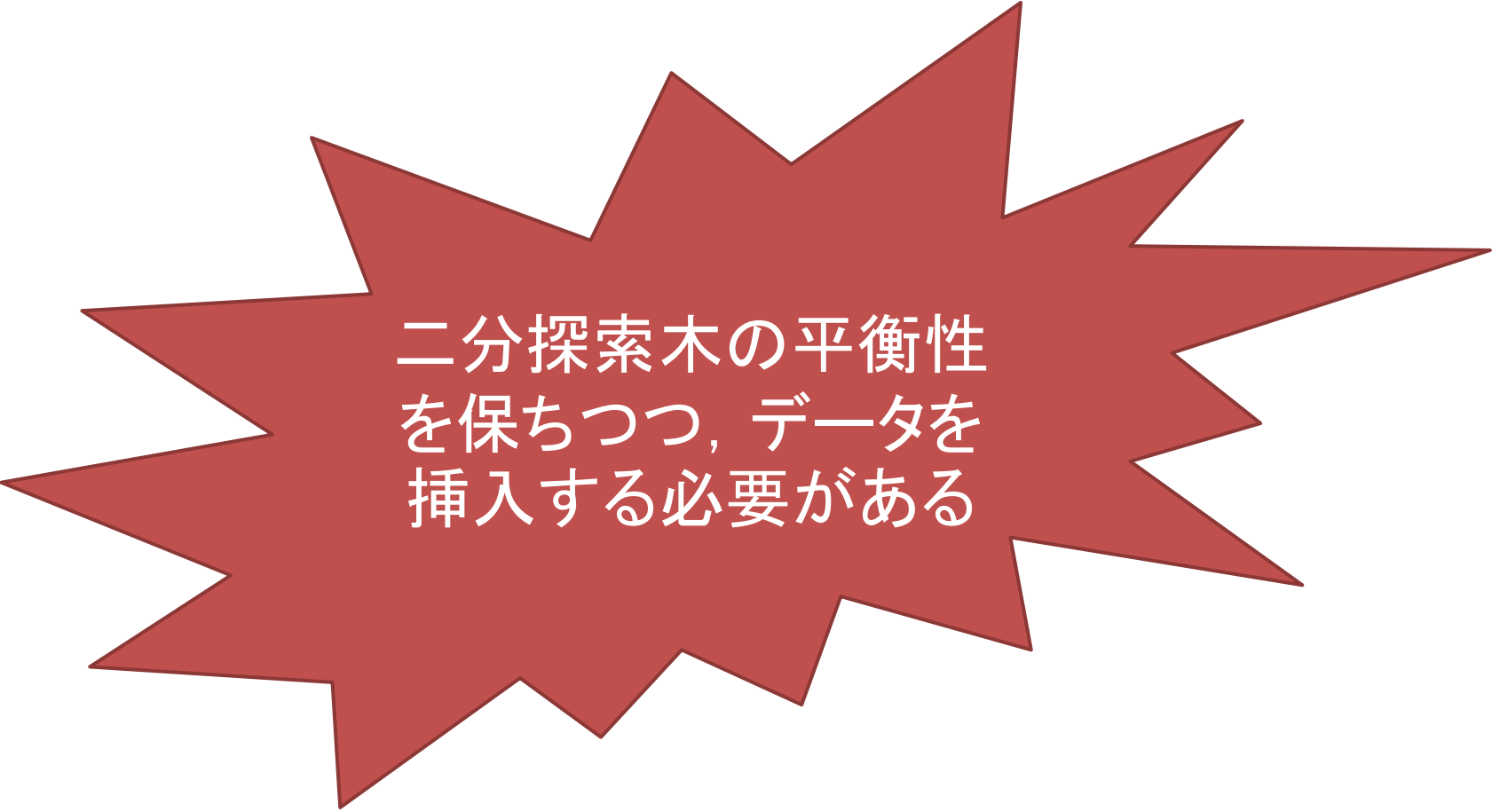
# 二分探索木のバランス

- 何も考えずにデータ挿入すると、バランスがメチャクチャ



4→2→6→1→3→5→7 の順に  
追加挿入 ⇒ 高さ $O(\log n)$





二分探索木の平衡性  
を保ちつつ、データを  
挿入する必要がある

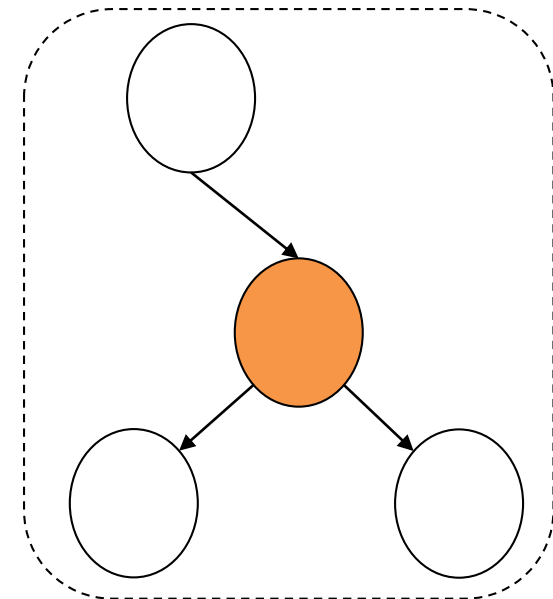
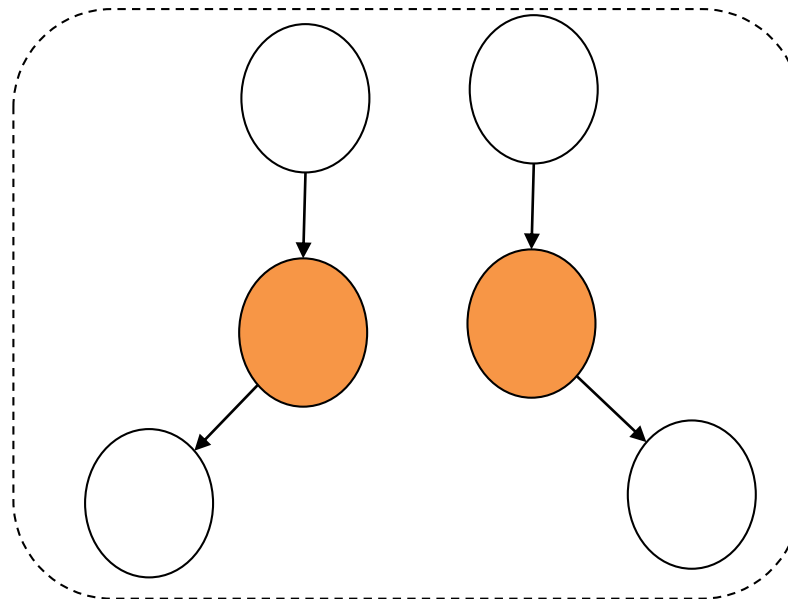
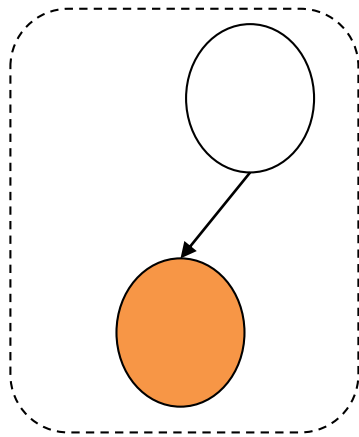
# 二分探索木からの削除

- 削除したい節点の状態で場合分け

ケース1: 子が無い (葉)

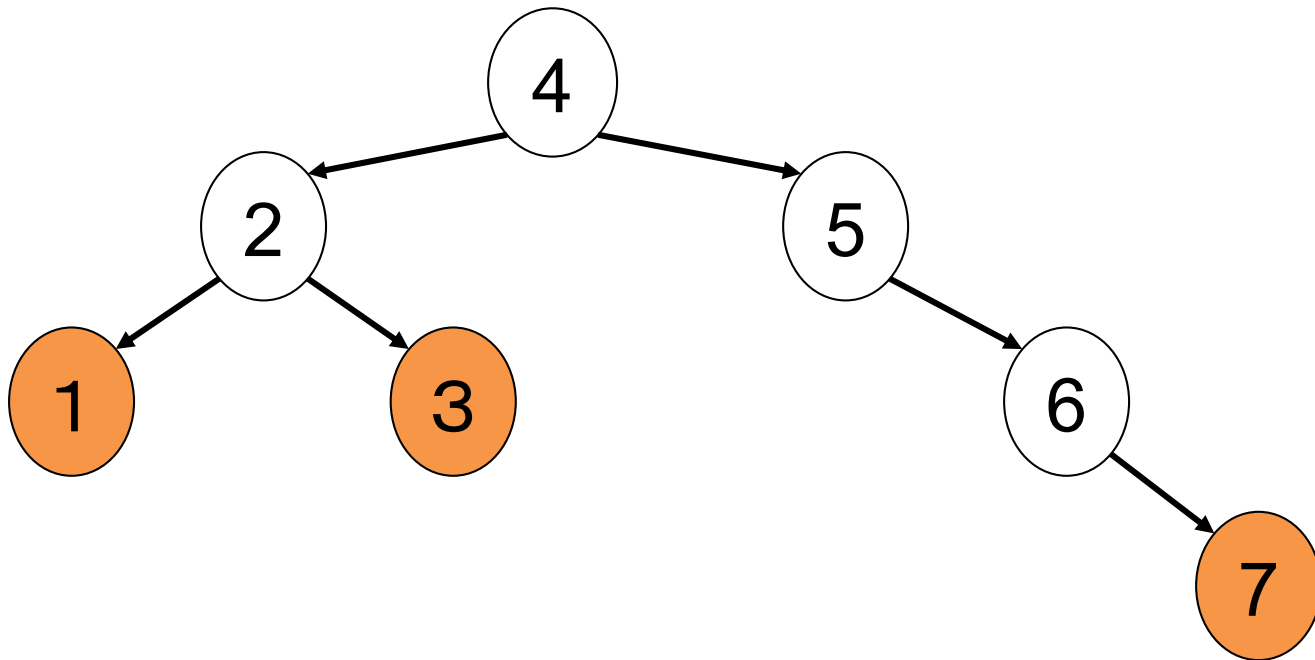
ケース2: 子が1つ

ケース3: 子が2つ



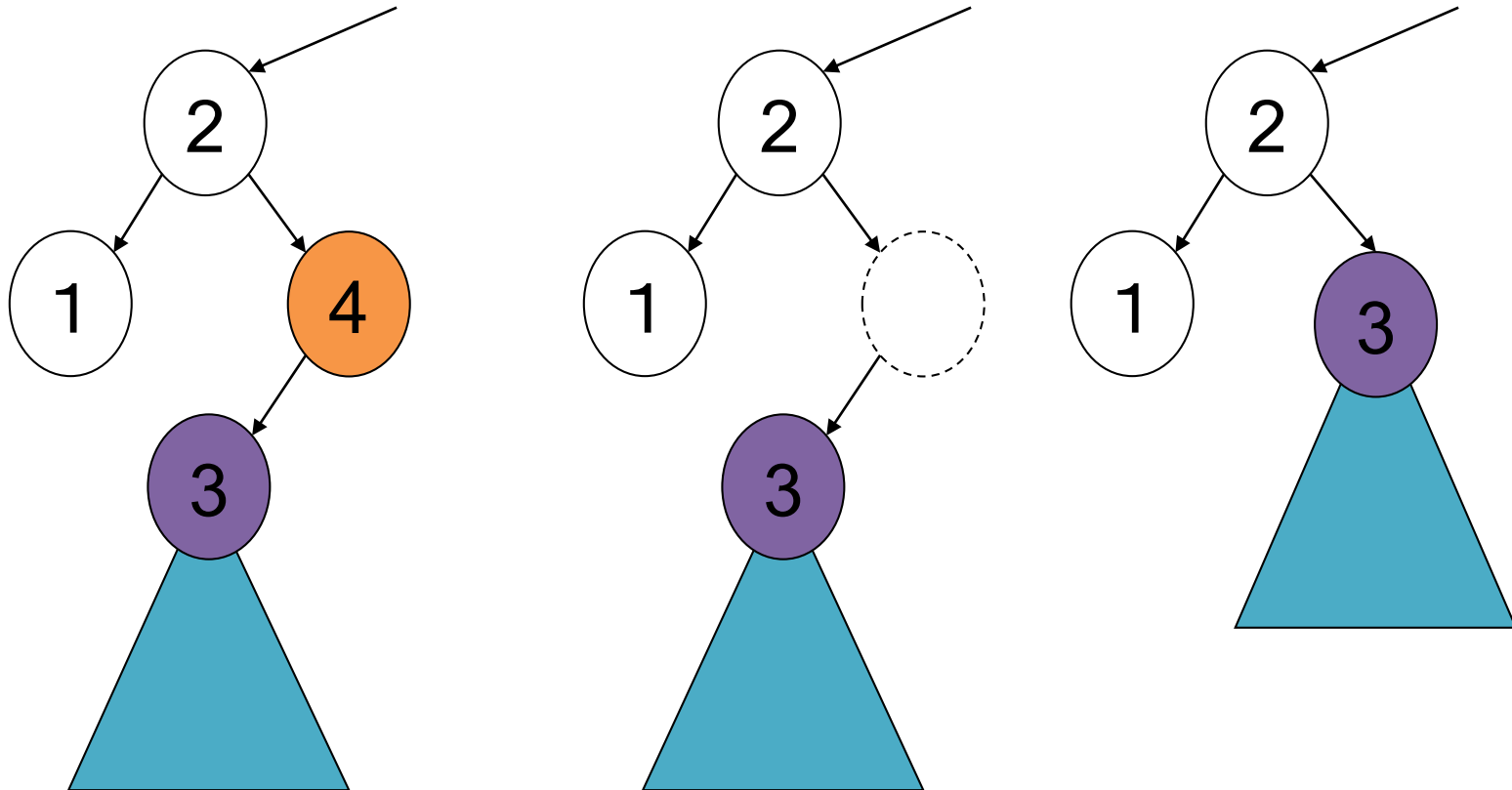
# ケース1:「子を持たない節点」の削除

- その節点を削除するだけでOK. 簡単



## ケース2:「子を1つ持つ節点」の削除

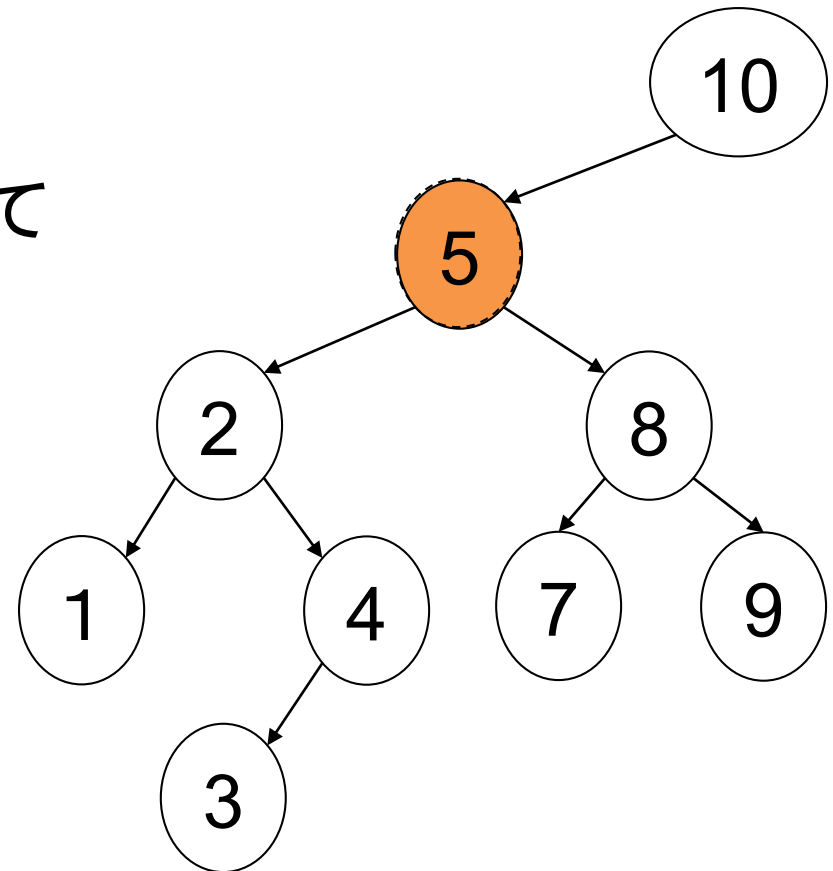
- その節点を削除し、親と子をつなげるだけ





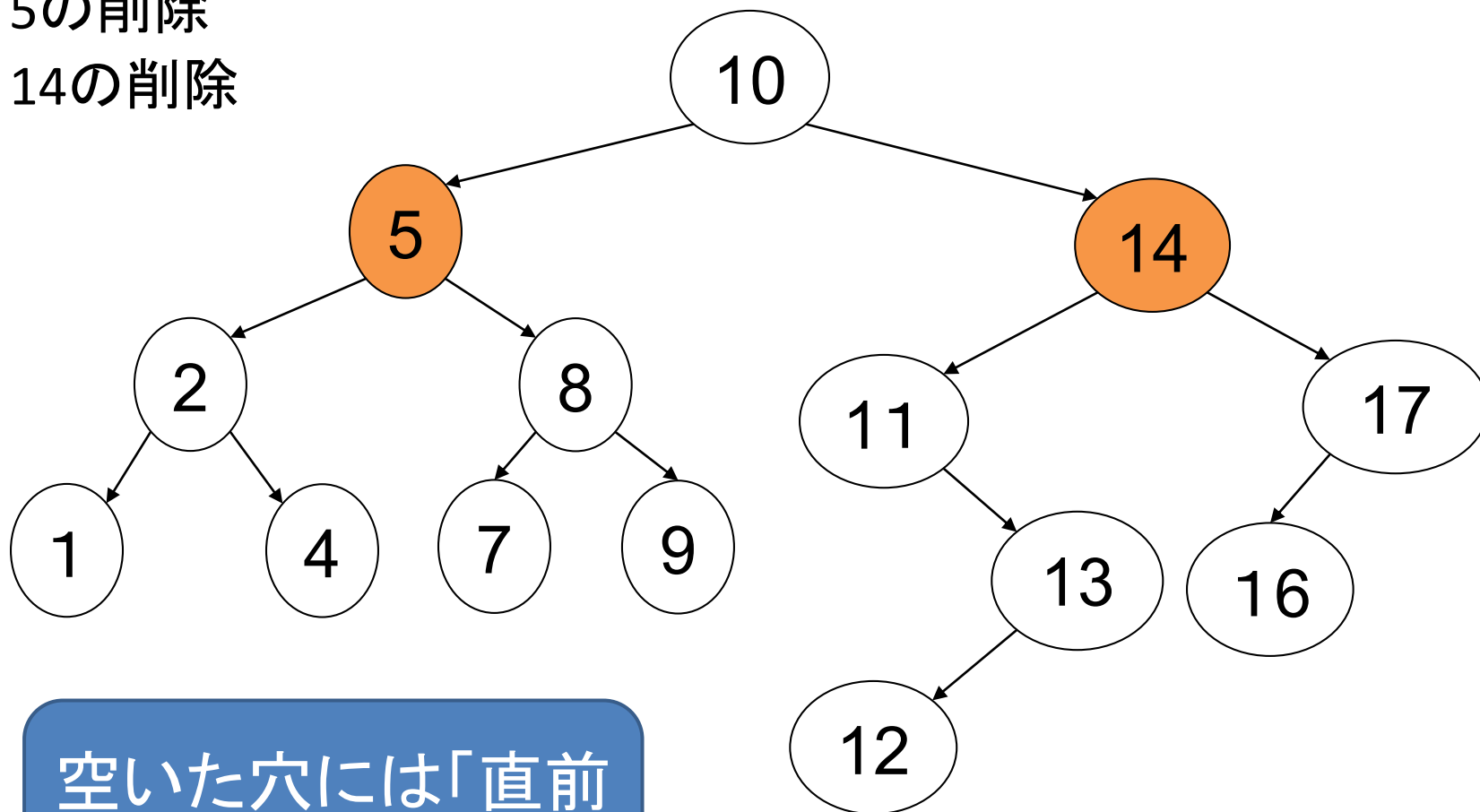
## ケース3:「子を2つ持つ節点」の削除

- その節点を削除する。
- 空いた「穴」に、「適切なもの」を持ってきてはめる
  - ▣ 「適切なもの」って？



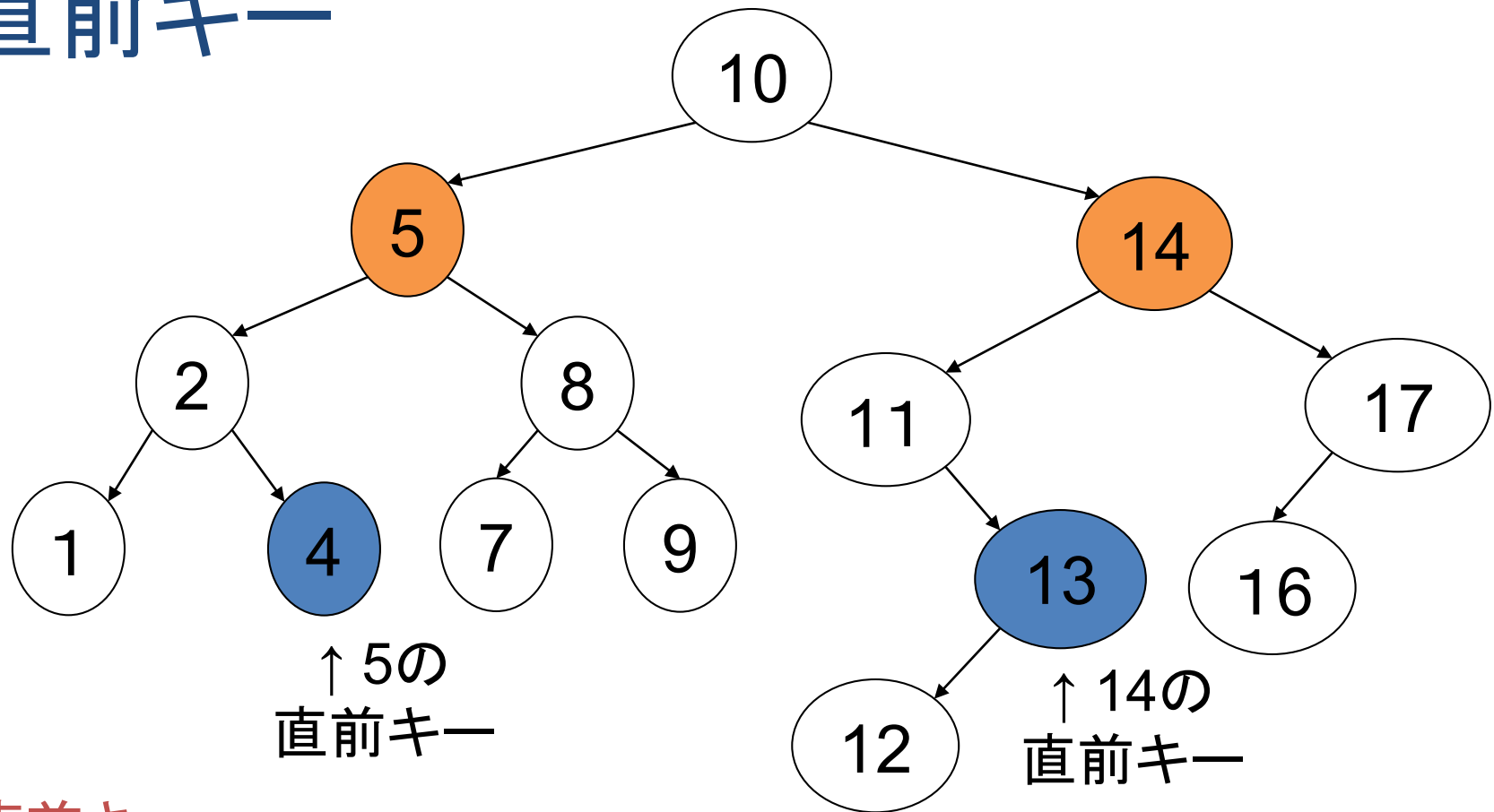
## 空いた穴には何を入れるべき？

- 5の削除
- 14の削除



空いた穴には「直前  
キー」を入れる

# 直前キー

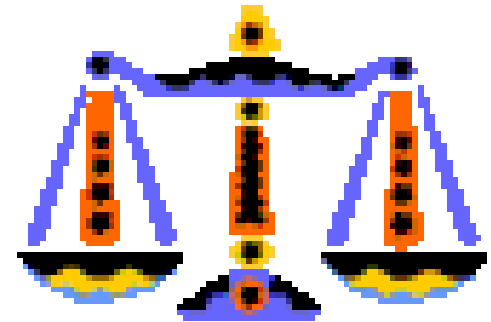


## 直前キー

- 削除した節点の左部分木のうち最大の要素
- 削除した節点から左下に1つ辿り、その後右へ右へと辿って行き着いた節点が「直前キー」になる

# まとめ

- ヒープ (Heap)
  - 最大値を取り出す
  - 挿入・取り出しともに  $O(\log n)$
  - 応用: ヒープソート
- 二分探索木 (Binary Search Tree)
  - 木の**バランスが良ければ**、  
探索、挿入、削除 ともに  $O(\log n)$ 
    - ※ ノード数  $n$  の木の高さは、バランスがよければ  $O(\log n)$
  - 木のバランスを保つための工夫
    - ・ 挿入・削除の操作時、**バランスが崩れる可能性がある**  
→ **バランスが崩れそうになったら補正**
      - ・ ただし、補正に手間をかけすぎると本末転倒



最悪時  $O(n)$

# 確認テスト(第5回)

- ヒープのアルゴリズムの動作
  - データ挿入
  - 最大データの取り出し
- 二分探索木のアルゴリズムの作成
  - データ挿入
  - データ削除