プログラム設計とアルゴリズム
第13回 (12/20)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習)文字列マッチング問題

- ある長い文字列テキストTと、短い文字列パターンSが与えられた時に、Tの中に出現するパターンSの位置を全て出力せよ。

- 例)

```
T: ACTGCACGTCTGTACGTCAT
S: ACGT
```

答)

ACTGC**ACGT**CTGT**ACGT**CAT

0-originとすると、T[5]及びT[13]からSが始まっている。

(前回の復習)brute-force法(力任せ法)

- Sを1文字ずつずらしながら、パターンとマッチするかを調べる。
  例)

```
T = ACATATAG
S = ATAT
```

- 1回目の比較:

```
ACATATAG
|X
ATAT
```

- 2回目の比較:

```
ACATATAG
 X
 ATAT
```

- 3回目の比較:

```
ACATATAG
  ||||
  ATAT
```

(前回の復習)力任せ法の最悪計算量

- 下記のようなケースを考えると、その最悪計算量はO(|T||S|)となることがわかる。
  例)

```
T = AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAT
S = AAT
```

- 1回の比較ごとに、|S|文字の比較が必要であり、それを|T|-|S|+1回分行わなければいけないため。

- とはいえこれは最悪ケースであり、平均的にはO(T)である。
  Tがランダム文字列であり、|A|を表れうる文字の数とすると、
  1回あたりの比較回数は、

  $1+(1/|A|)+(1/|A|)^2+\cdots+(1/|A|)^{|S-1|}$

(前回の復習) KMP法

- 1回目の比較:

```
CTACTGATCTGATCGCTAGATGC
||X
CTGATCTGC
```

- S[2]でミスマッチ→S[1](T)はマッチ
  つまりT[1]はCではないので、スキップして良い。

- つまり、1文字後を見るのではなく2文字後を見て良い。

- 2回目の比較:

```
CTACTGATCTGATCGCTAGATGC
  X
  CTGATCTGC
```

(前回の復習) KMP法

- KMP法の概略:
  1. 与えられたSから表hを作成する。
     h[i]は、S[i]まで調べた時何文字スキップするかを意味する。
     (表hはTには依存しないことに注意せよ)

  2. Tに対して、Sを前からパターンマッチしていく。
     パターンマッチに成功／失敗してSをずらす際には、1つずつずらすのではなく、i文字目まで調べたらh[i]だけずらす。

- `S:CTGATCTGC`
  の時の表h

h[0]	h[1]	h[2]	h[3]	h[4]	h[5]	h[6]	h[7]	h[8]
1	1	2	3	4	6	6	7	5

- KMP法の計算量はO(|S|+|T|)である。

(前回の復習) BM法

- 1回目の比較

```
CTACTGATCTGTTCGCTAGATGC
     X
TAATAA
```

- 不一致が起きた際に、Gと不一致が起きていることに注目する。パターンSの中にGは存在しないので、SはGと被らないようにして良い。よって大幅にスキップできる。

- 2回目の比較

```
CTACTGATCTGTTCGCTAGATGC
           X
      TAATAA
```

(前回の復習)Rabin-Karp法

- ハッシュ関数を用いた文字列検索マッチング
- まず、文字列に対するハッシュ関数を定義する
  例)ローリングハッシュ
  　文字列 $x = c_1c_2\cdots c_m$としたとき、
  　$hash(x) = (c_1a^{m-1}+c_2a^{m-2}+\cdots+c_ma^0)\ \%\ M$
- まずhash(S)を計算しておく。ここでSの長さはmとする。
  その後、hash(T[0..m-1])を計算し、値がhash(S)と同じなら同じ文字列かチェックする。違う値なら次に移動する。
- 次はhash(T[1..m])を計算する。これを繰り返し、T内の長さmの部分文字列全てに対してhash値の計算を行い、文字列の判定を行う。

(前回の復習) Rabin-Karp法の例

- ローリングハッシュを用い、a=2, M=5とする。

```
S: 10101
T: 11001010110
```

- hash(S) = (16+4+1)%5 = 1となる。

- hash(T[0..4]) = hash(‘**11001**’) = (16+8+1)%5 == 0 よって次に進む

  hash(T[1..5]) = hash(‘**10010**’) = (16+2)%5 == 3

  hash(T[2..6]) = hash(‘**00101**’) = (4+1)% 5 == 0

  hash(T[3..7]) = hash(‘**01010**’) = (8+2)% 5 == 0

  hash(T[4..8]) = hash(‘**10101**’) = (16+4+1)%5 = 1

  hash(S)と一致するので、SとT[4..8]が一致するかを調査する。

発展的なデータ構造

1. セグメント木
2. 二項ヒープ

Range minimum query(RMQ)

- 区間最小値問題:

  N個の数a_0, …, a_{N-1}と、2つの整数l, rが与えられます。ただし、
  $0 \leqq l < r \leqq N-1$ であるとします。この時、a_l, …, a_r の中で、
  最も小さい値を求めなさい。

- 例:
  a = {7, 3, 13, 15, 3, 9, 1, 15}, l =2, r = 5
  答:
  a_2 =13, a_3 =15, a_4 = 3, a_5 =9 の中から最小値を選ぶので、
  最小値は3となる。

Range minimum query(RMQ)

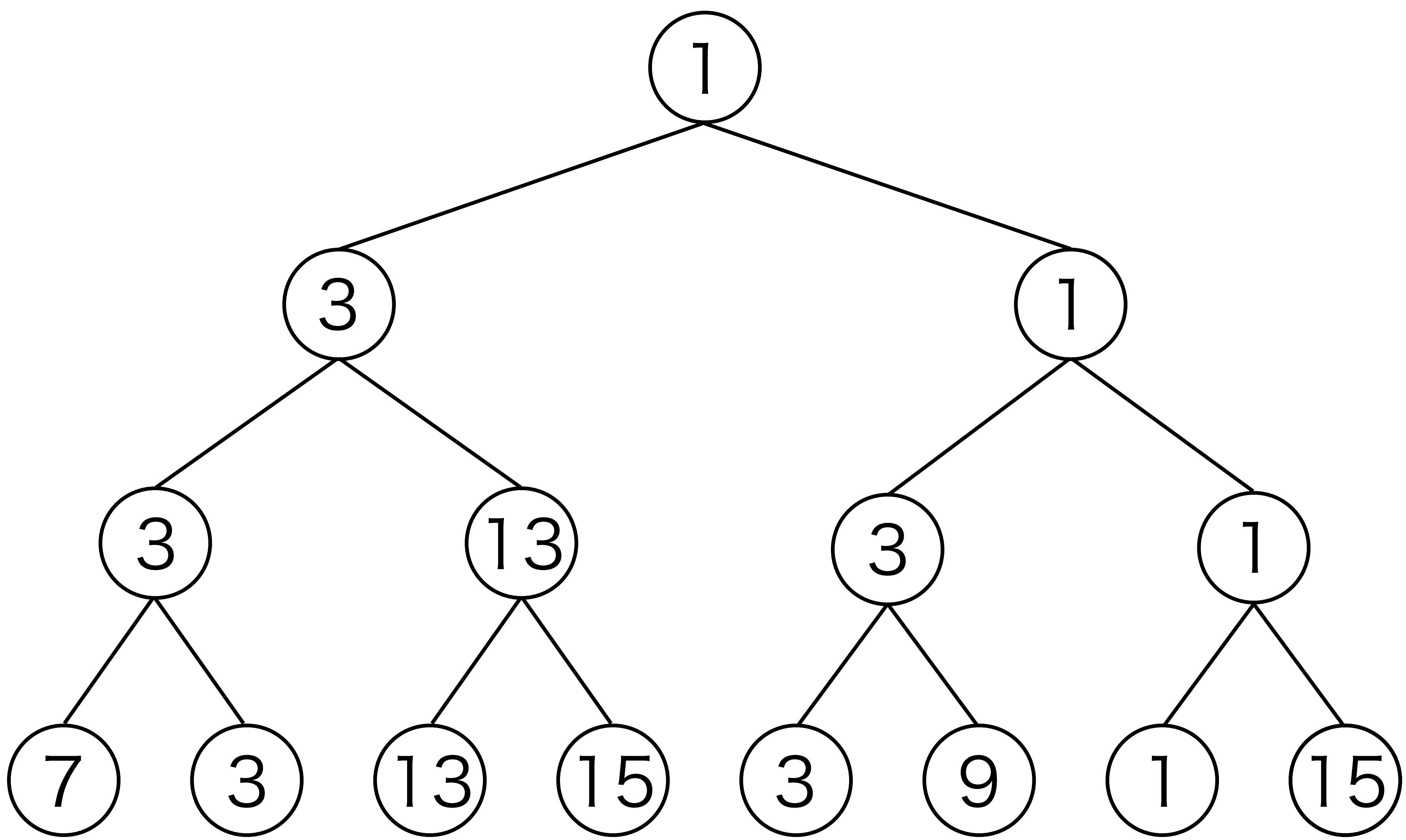
- 愚直なアルゴリズム:
  a_l, …, a_r の値を一つずつ調べれば良いので、調べる回数は$r - l + 1$回となる。よって最悪計算量は、O(N)となる。

- 配列aが与えられた時に、セグメント木というデータ構造を作成しておくと、その最悪計算量がO(log N)となる。

- ただし、セグメント木の構築にはO(N)かかるので、RMQが一度行われるだけなら、効率的ではない。同一のaに対して、lやrを変えて何度もRMQが行われる時にセグメント木を作るという前処理が有効になる。

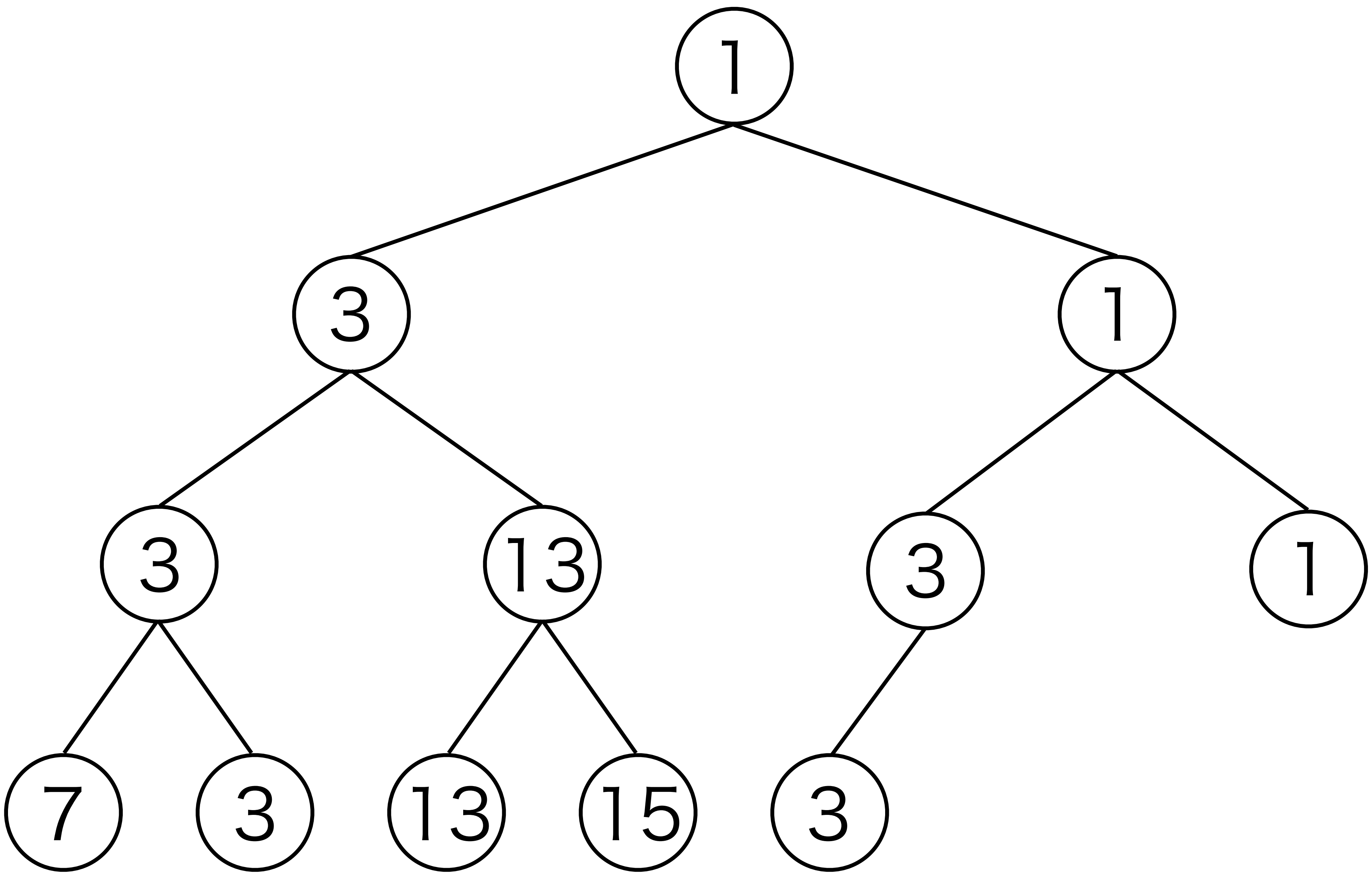
セグメント木

- セグメント木は強平衡な二分木であり、葉は配列の各要素を表す。
  各頂点xは、xの子孫の葉のうち最も小さいものを保持する。

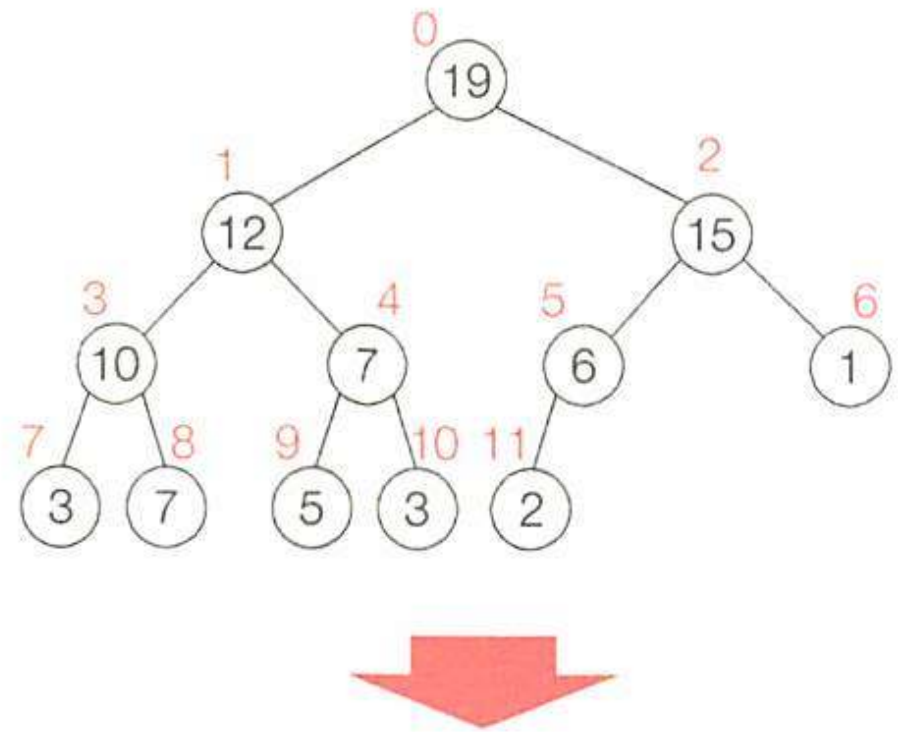


セグメント木

- 要素数が2^nでないときは、葉の要素は左詰とする(ヒープと同様)。
  このことから、セグメント木は配列を用いて表現することが可能である。



(復習)配列によるヒープの実現



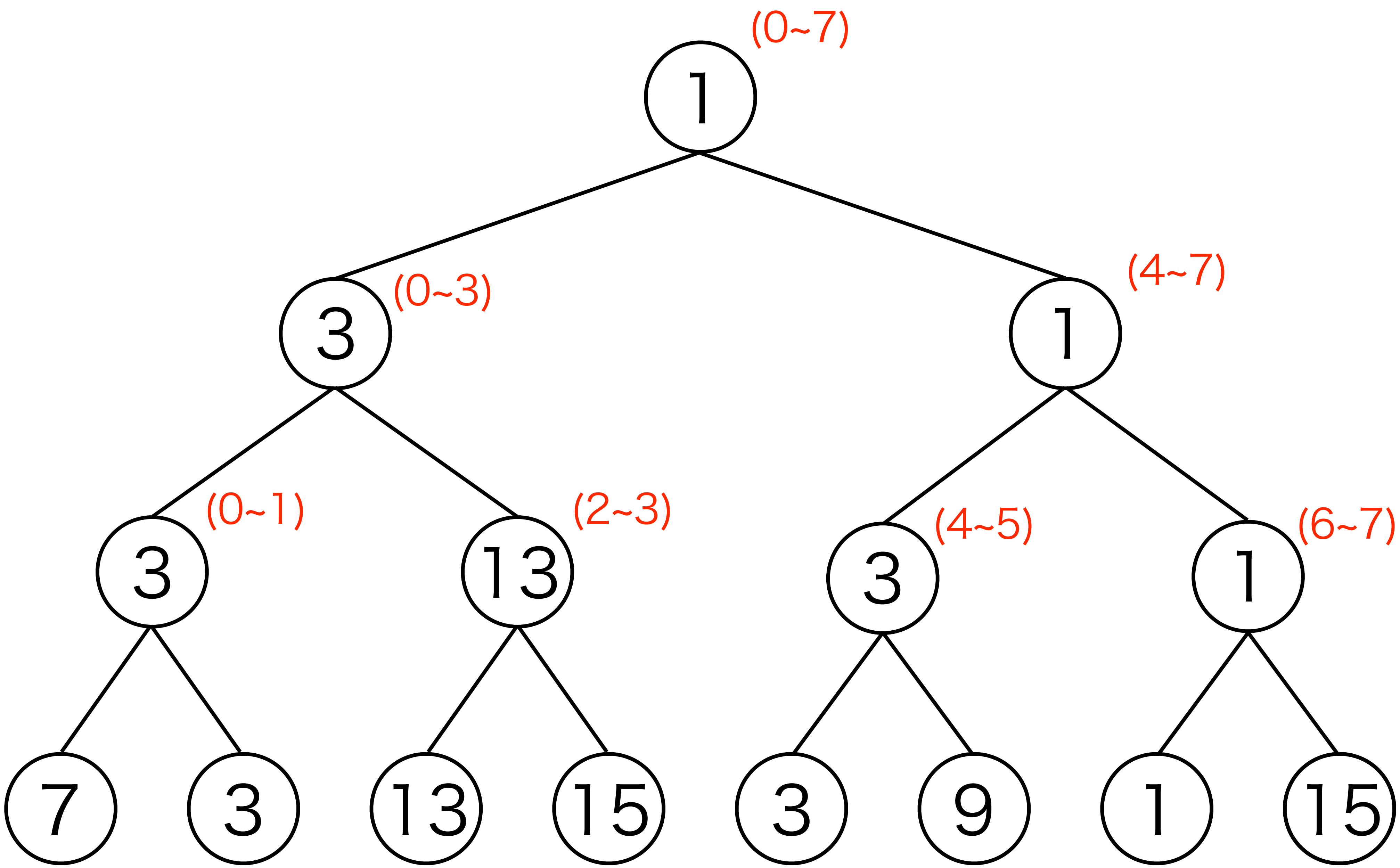
0	1	2	3	4	5	6	7	8	9	10	11
19	12	15	10	7	6	1	3	7	5	3	2

図 10.19　ヒープの配列を用いた実現方法

- 上から順番、同じ深さの場合左から順番に添字を振り、配列の添字が対応する箇所に要素を格納する。
- 頂点xにおいて、親の添字は(x-1)/2(切り捨て)、子の添字はx*2+1とx*2+2となる。

セグメント木

- 各頂点はある範囲の最小値意味することになるが、その範囲を赤字で示す(この情報は配列の中に保持しておかずとも木の高さのみで定まる)



セグメント木への問い合わせ

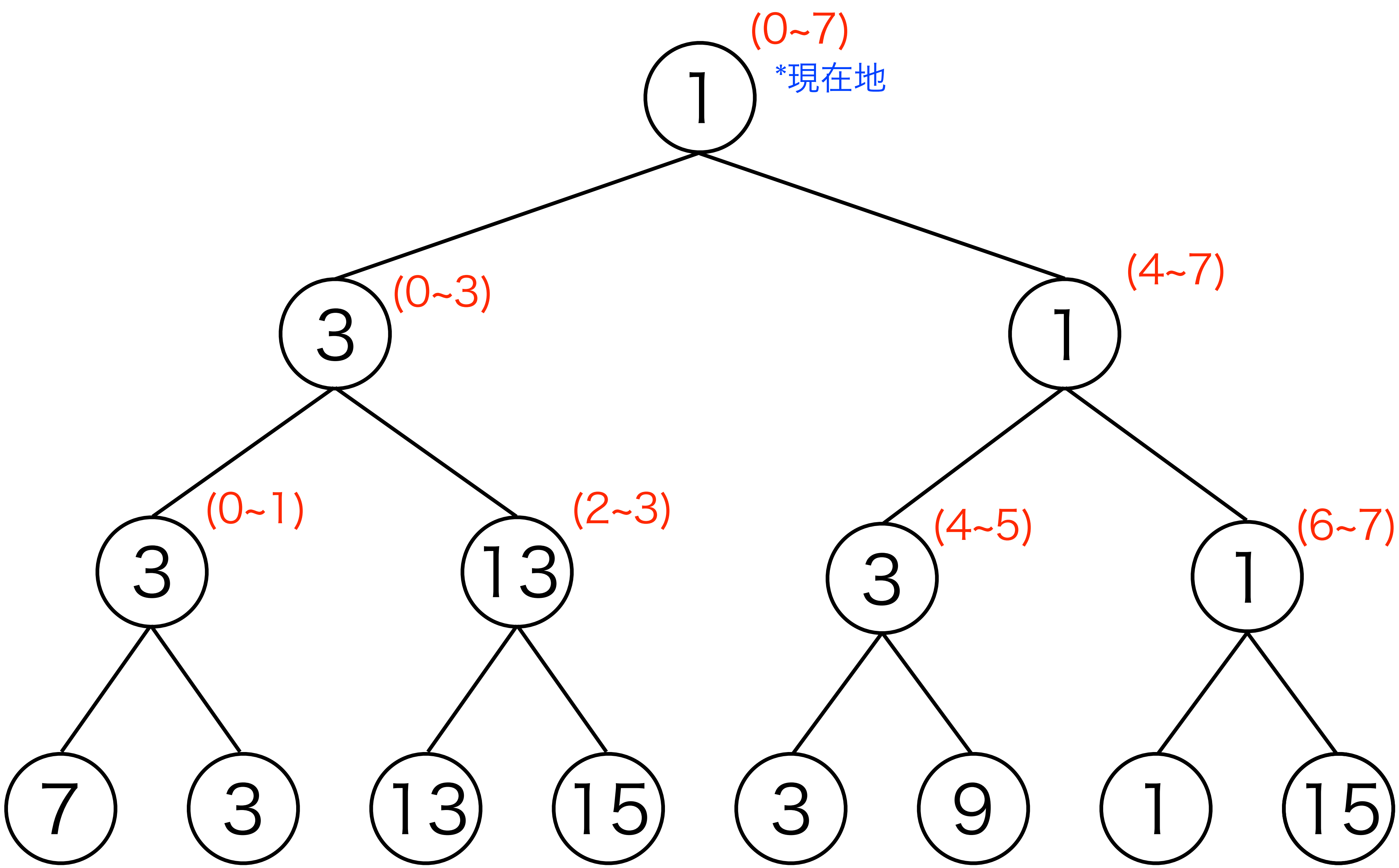
- セグメント木に対して区間最小値を求める問い合わせは、再帰を利用することで達成される。

- 現在地の頂点の範囲を(a~b)であるとする。また、問い合わせの範囲を(l~r)であるとする。この時、次の3つの場合が考えられる。

1) r < a または b < lである場合
　範囲外なので考慮しなくてよい
2) l <= a かつ b <= rである場合
　完全に範囲内なので、頂点の値を返す
3) それ以外(部分的に重なる場合)
　2つの子供に対して同じ問い合わせを行い、その返り値の最小値を返す

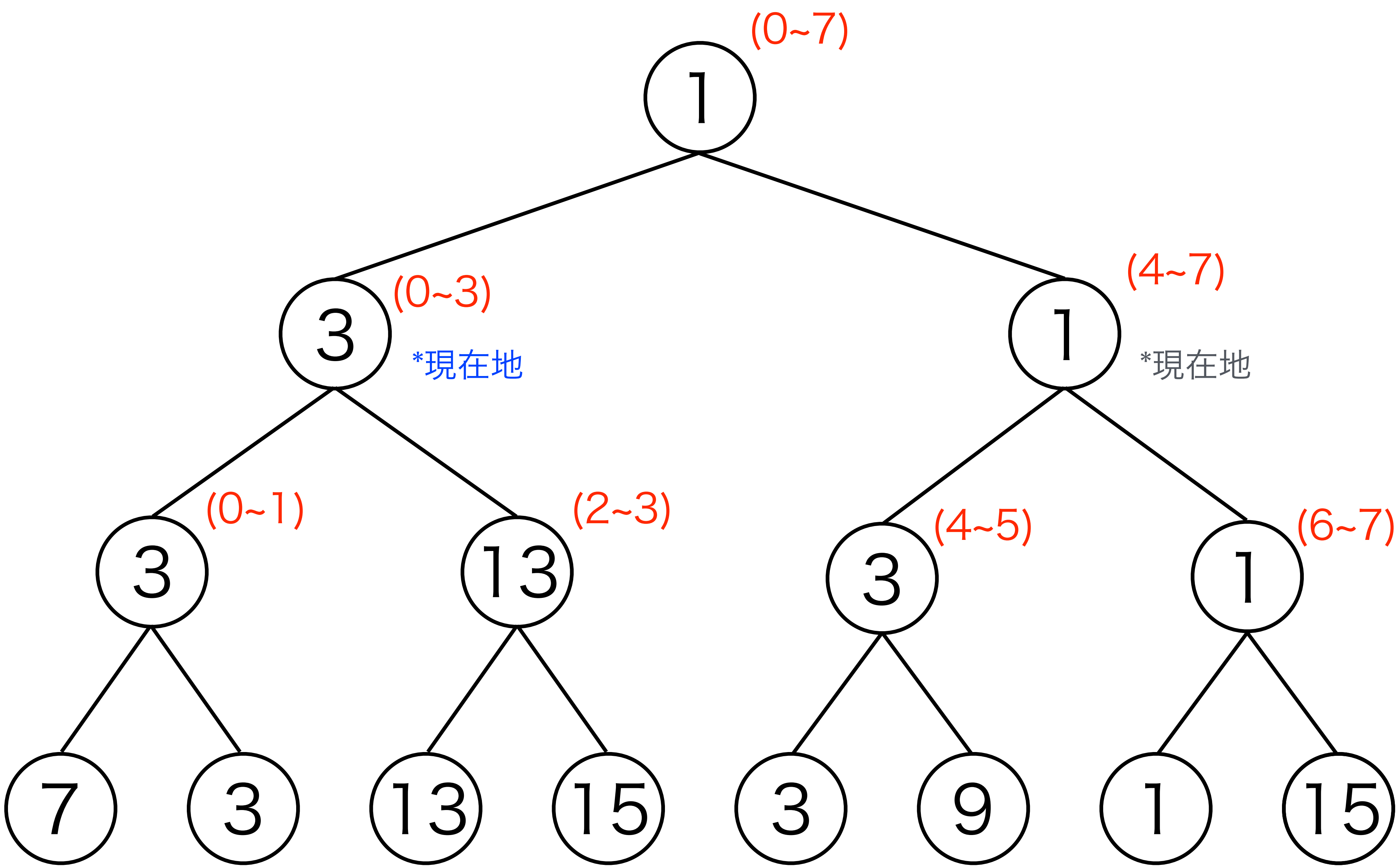
セグメント木への問い合わせ例

- $l = 2, r = 5$とする。根を見ると、$a= 0, b = 7$であり、3)に該当するので、2つの子に対して問い合わせを行う。



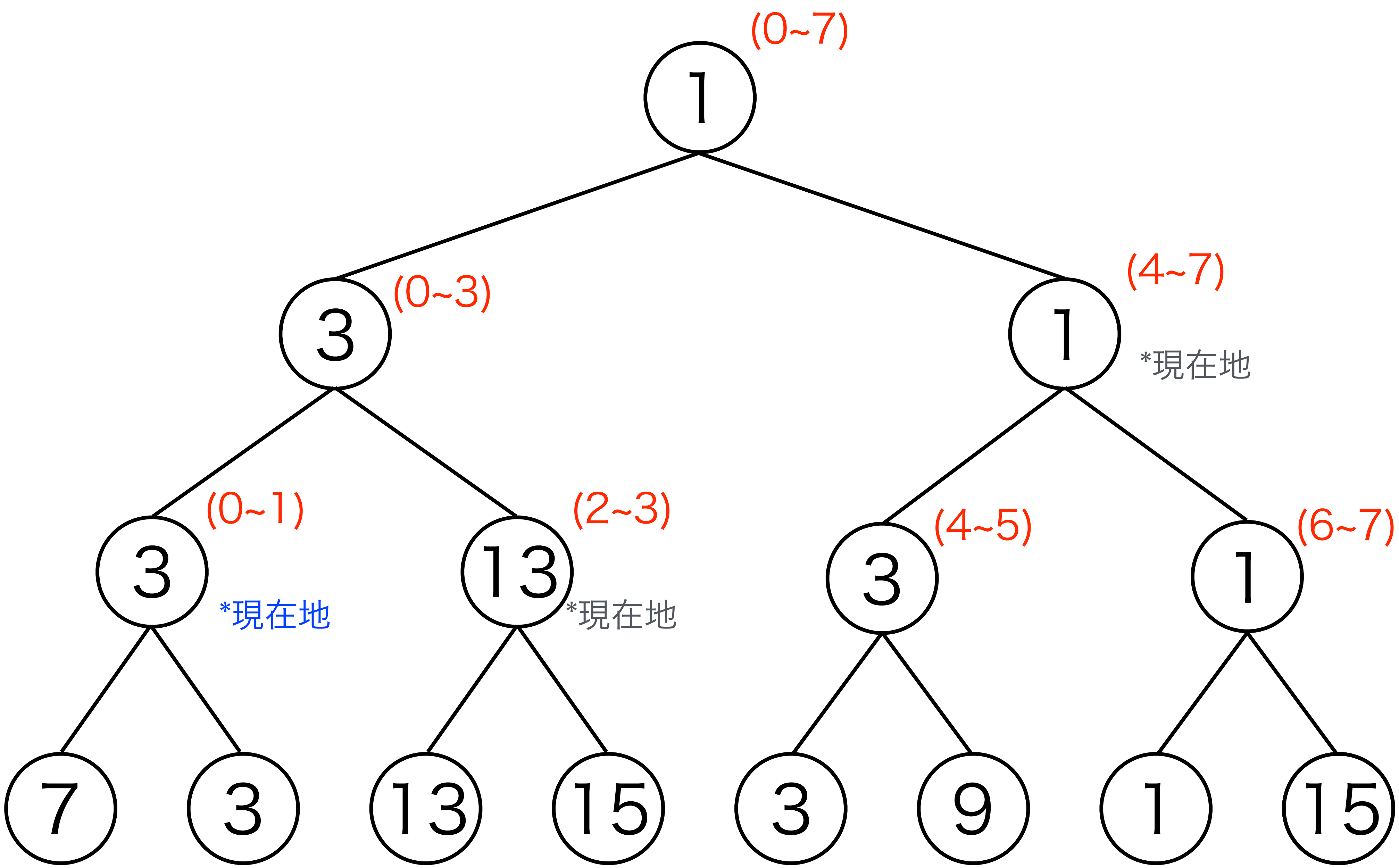
セグメント木への問い合わせ例

- 左の子を先に見るとする。l = 2, r = 5, a= 0, b = 3なので、3)に該当。2つの子を見に行く。



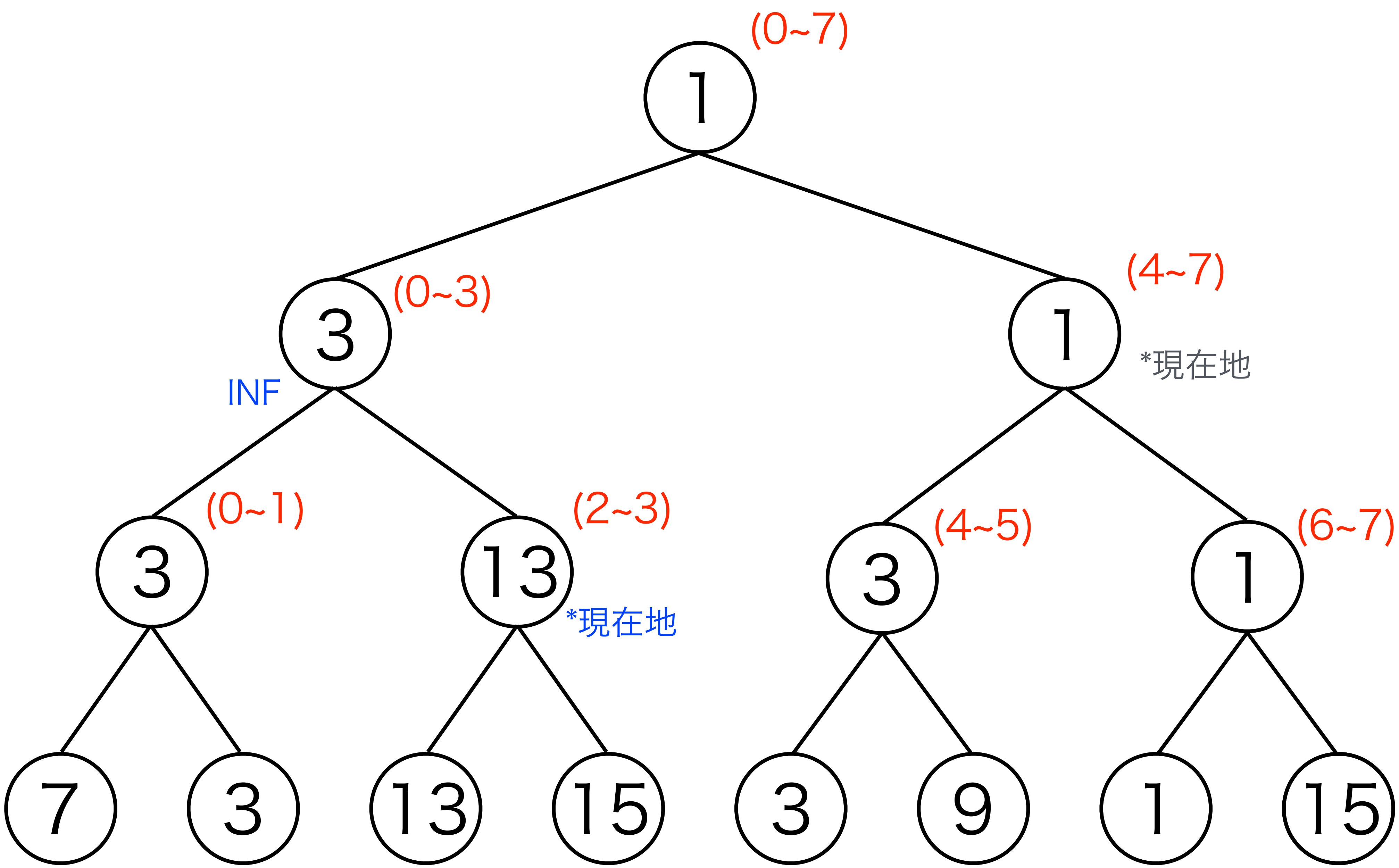
セグメント木への問い合わせ例

- 左の子を先に見る。l = 2, r = 5, a= 0, b = 1なので、1)に該当。
  無視して良い(とりあえずINFを返すものとする。)



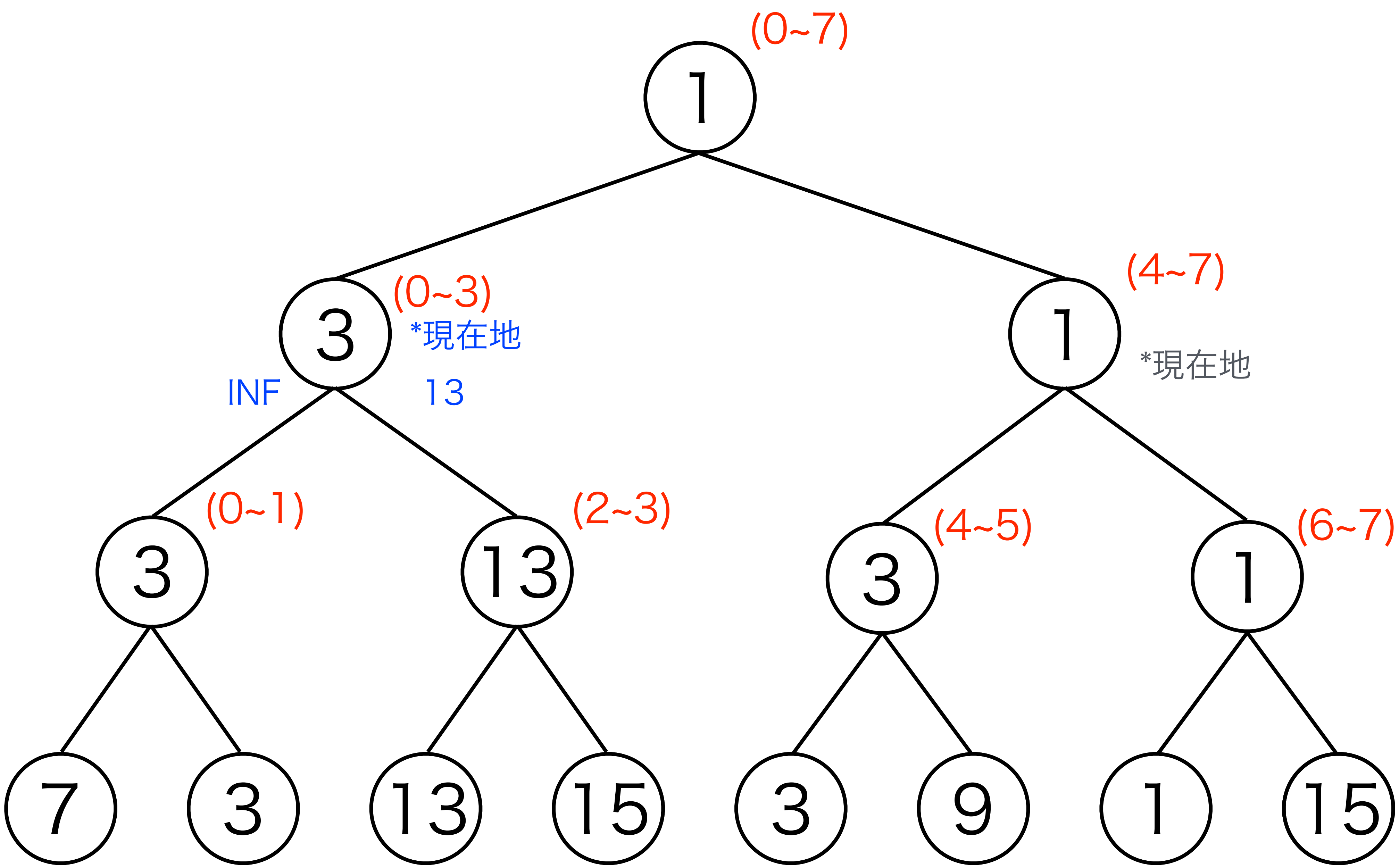
セグメント木への問い合わせ例

- 後回しにしていた右の子を見る。l = 2, r = 5, a= 2, b = 3なので、2)に該当、その値をそのまま返す



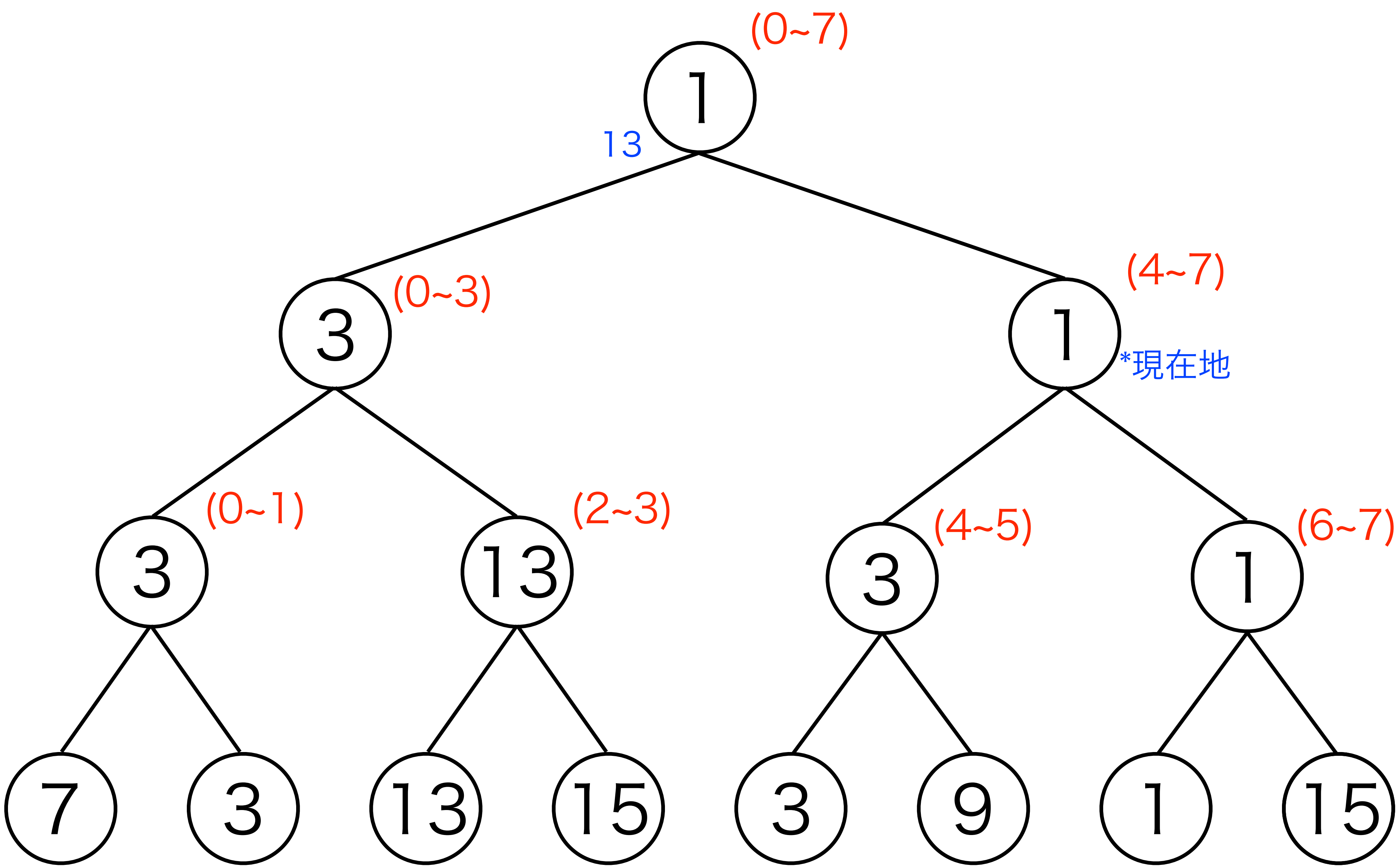
セグメント木への問い合わせ例

- 2つの子の返り値のうち最小値(13)を返す。



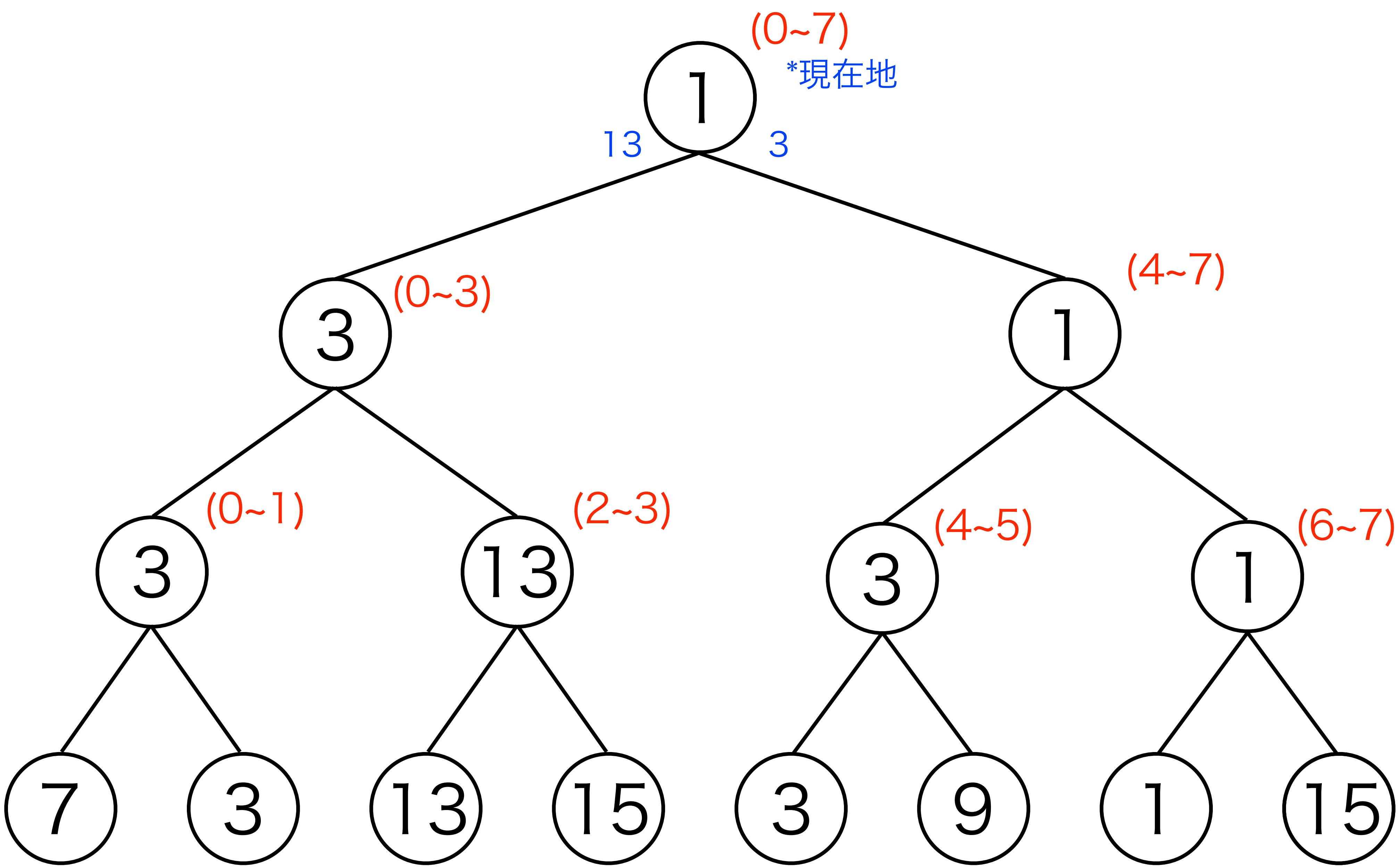
セグメント木への問い合わせ例

- 同様の操作を右の子に対しても行なっていくと、3を返すことになる。



セグメント木への問い合わせ例

- 2つの子の最小値(3)を返し、これが答えとなる。
  木の各高さにおいて、探索される頂点の数は高々4つであるため、
  その計算量はO(logN)となる。



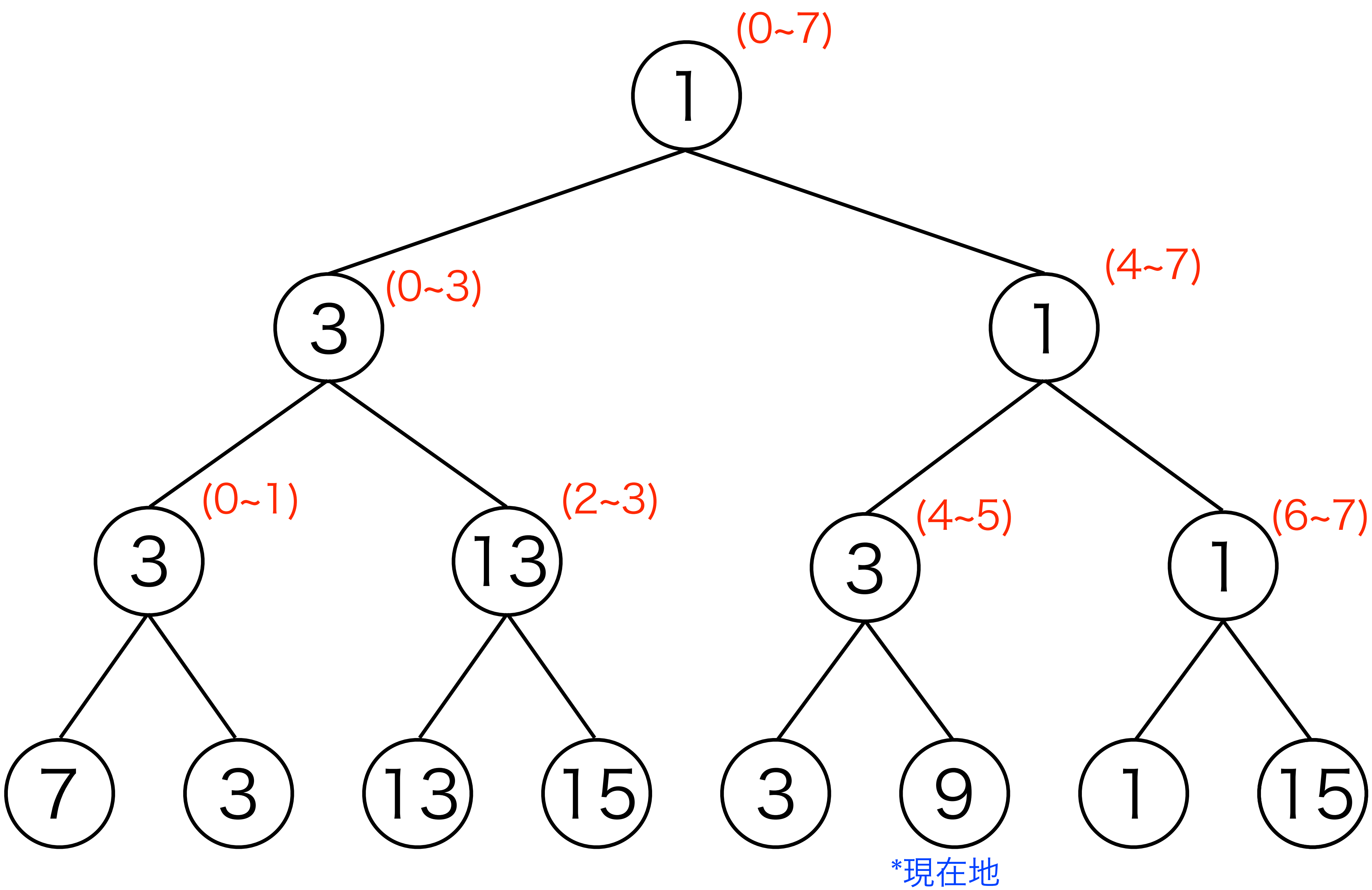
セグメント木の構築と更新

- セグメント木の構築はO(N)で行う事ができる。
  (葉を配列aの要素で埋めた後、葉から根へ向かって子の最小値の値を
  入れていけば良い)

- 配列aの要素が一つ変更された時に、セグメント木の更新を
  行いたいものとする。これも、変更された要素の葉から
  親へ向かって要素を更新していけば良いので、その計算量は
  高々O(log N)である。

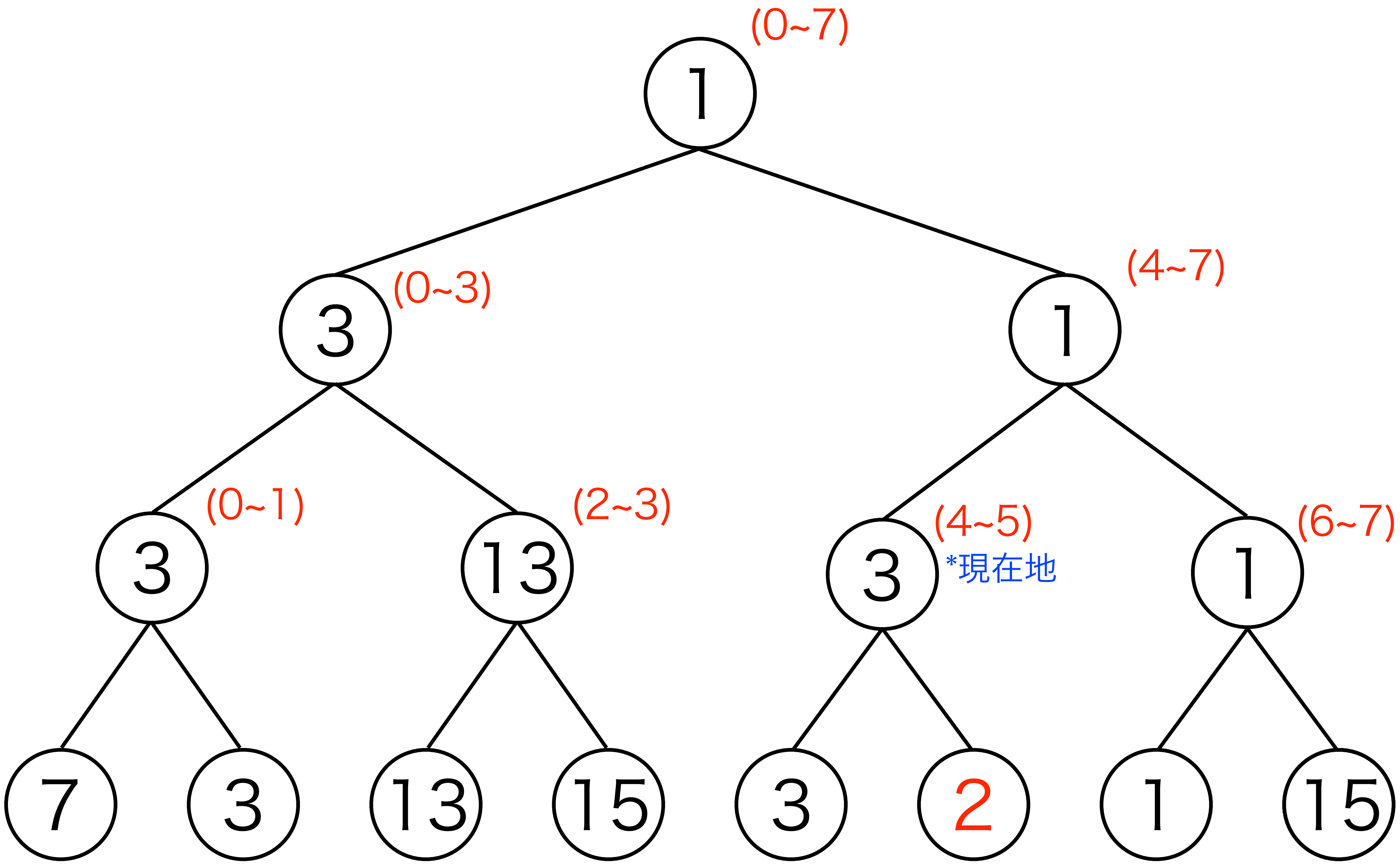
セグメント木の更新

- a_5を2に更新したとする。



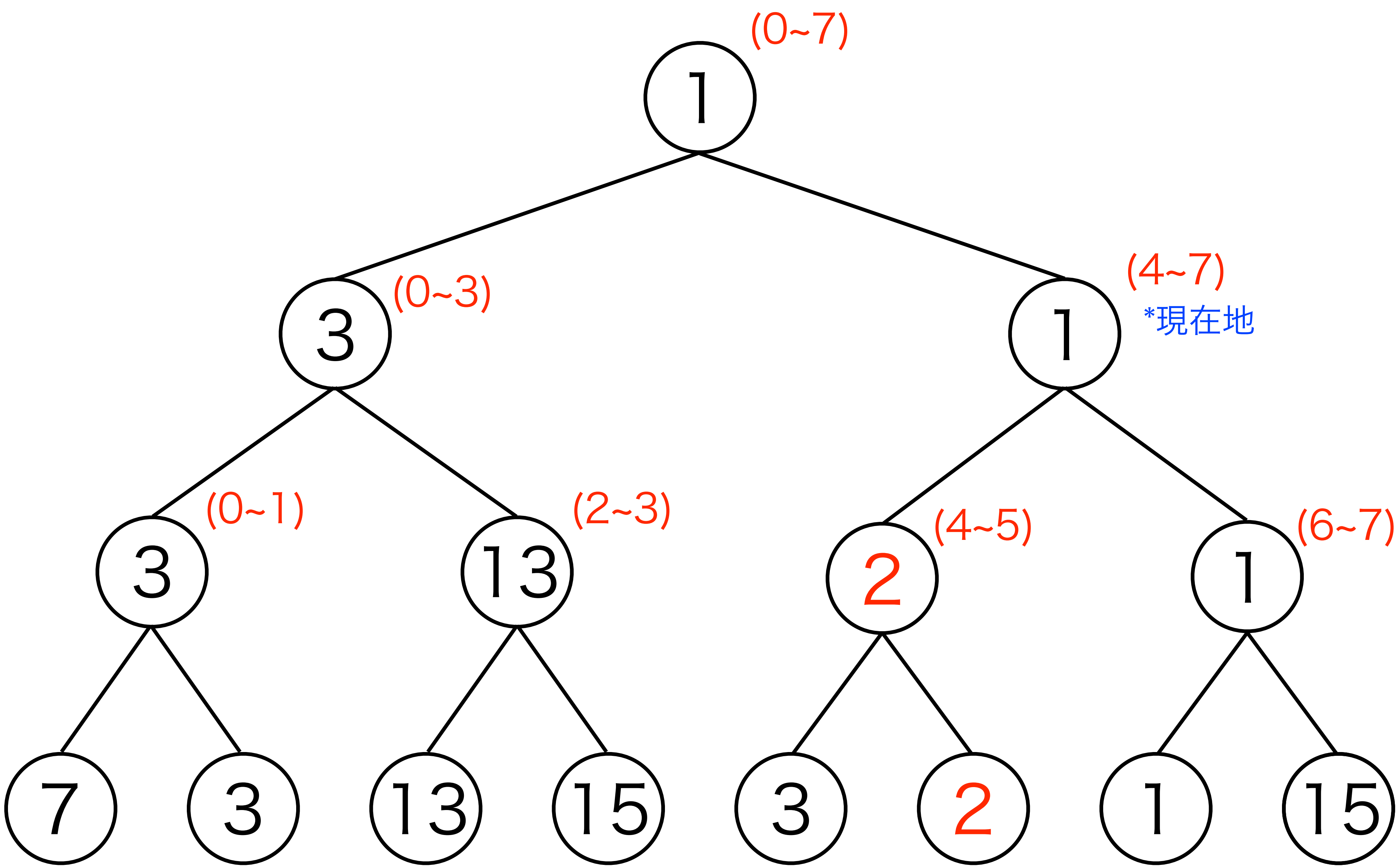
セグメント木の更新

- 親へ向かい、要素を更新する必要があるかを調べる。この場合、3 > 2であるため、要素を更新する。



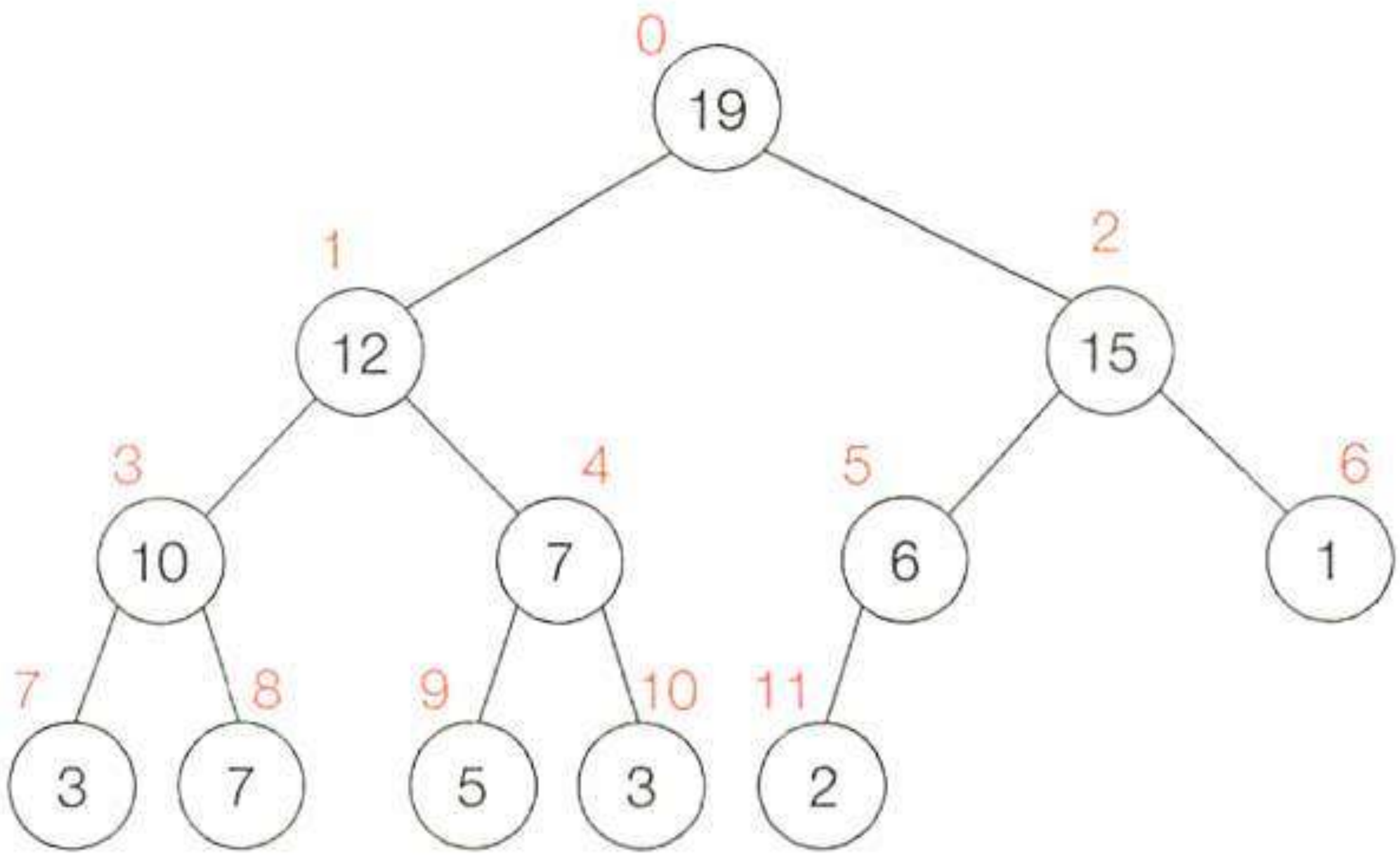
セグメント木の更新

- さらに親へ向かい、要素を更新する必要があるかを調べる。
  1 < 2 であるため、要素を更新する必要は無く、これ以上親へ行く必要もなし



(復習)二分ヒープ

- 最大値を取得するのに適したデータ構造
  親の値は必ず子の値以上となる
  強平衡二分木



二分ヒープの特徴

- 最大値を取得するのに必要な計算量はO(1)
  新しい要素を挿入するのに必要な計算量はO(logN)
  最大値削除処理はO(logN)
  新規にヒープを構築するのに必要な計算量はO(N)

- N個の要素からなる二分ヒープとM個の要素からなる二分ヒープを
  マージして、N+M個の要素からなるヒープを構築することを考える。
  その計算量は？

- 一から新規構築し直すとO(N+M)となるが、二分ヒープでは
  これより効率的にマージを行う事はできない。

  →マージを頻繁に行う場合は、二項ヒープを用いる。

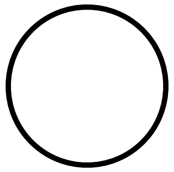
二項木

- 二項ヒープは、二項木の集合である。

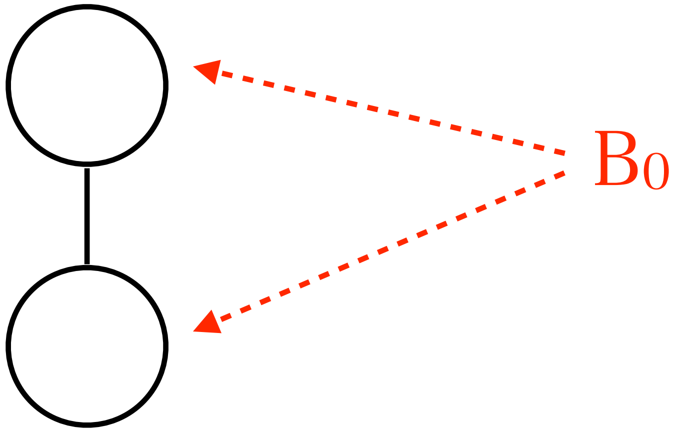
- 二項木は、次のように定義される。
  1) 0次の二項木B_0は、頂点は一つのみの木構造である。
  2) n次の二項木B_nは、B_{n-1}の頂点の最も左の子として、
  　B_{n-1}の根を結合させた木構造である。

二項木

B_0: 頂点は一つのみ

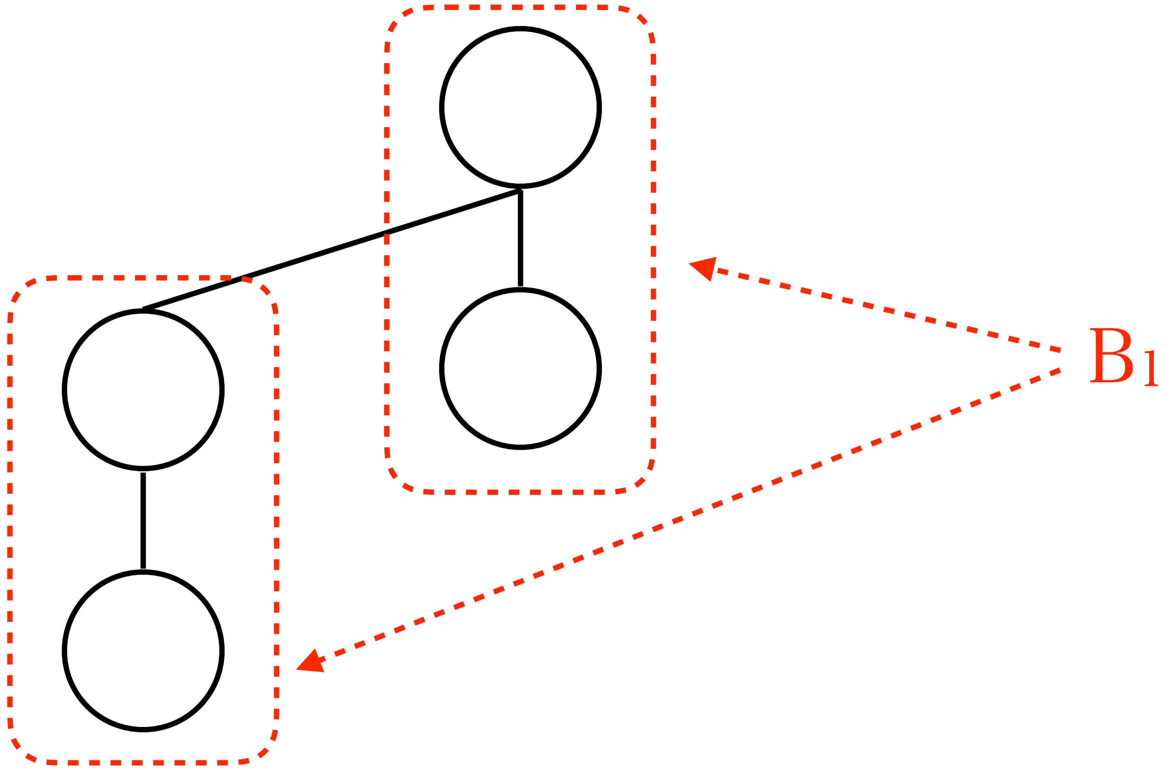


B_1: B_1は、B_0の頂点の最も左の子として、B_0の根を結合



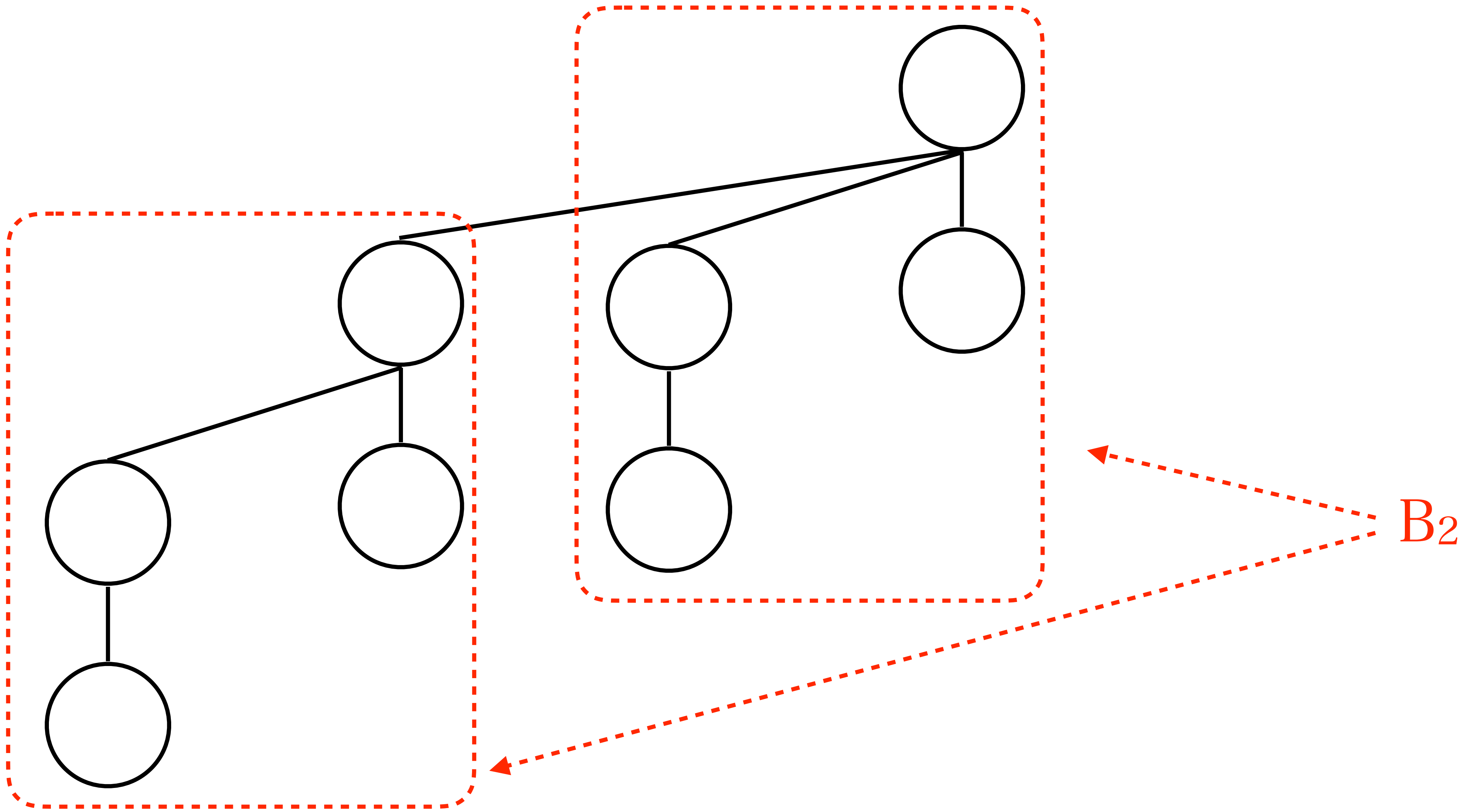
二項木

B_2: B_2は、B_1の頂点の最も左の子として、B_1の根を結合



二項木

B_3: B_3は、B_2の頂点の最も左の子として、B_2の根を結合



二項木

- 二項ヒープは、二項木の集合である。

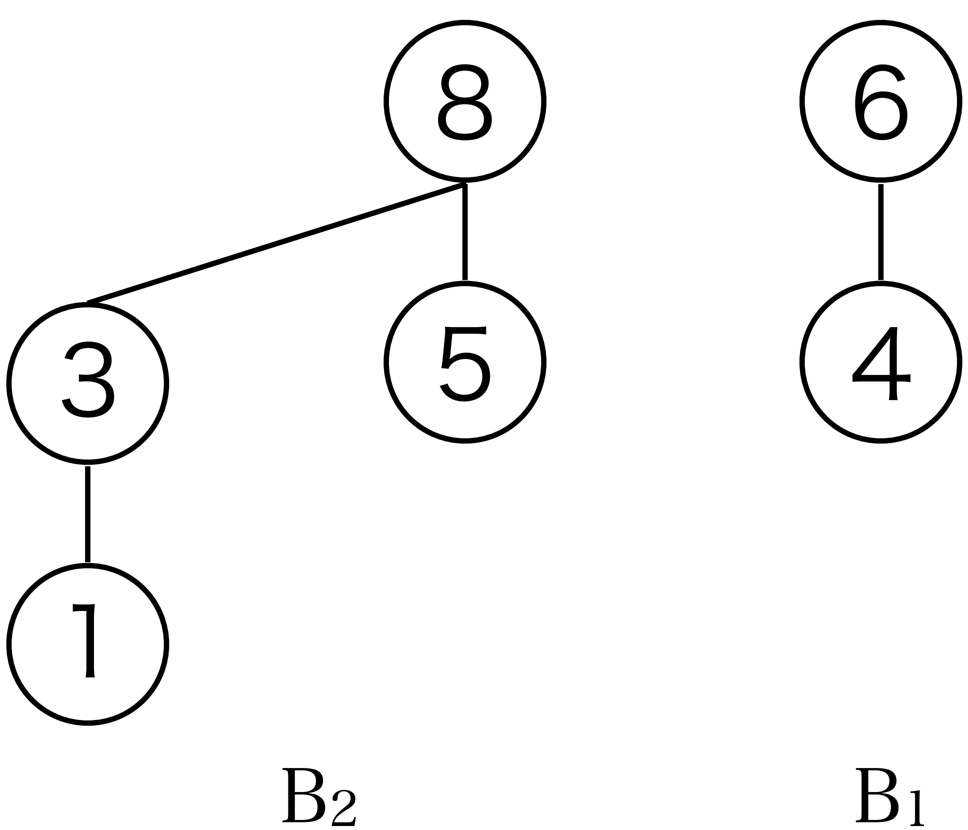
- 二項木は、次のように定義される。
  1) 0次の二項木B_0は、頂点は一つのみの木構造である。
  2) n次の二項木B_nは、B_{n-1}の頂点の最も左の子として、
  B_{n-1}の根を結合させた木構造である。

- B_nの根の子頂点の数はn個 (二分木ではない)
  B_nの木の高さはn
  B_nの節点数はB_{n-1}の2倍であり、すなわち2^n個

二項ヒープ

- 二項ヒープは、二項木の集合として表現された集合
  1) 親の値は必ず子の値以上となる
  2) 二項ヒープの中に、同じ次数の二項木は高々一つしか存在しない。

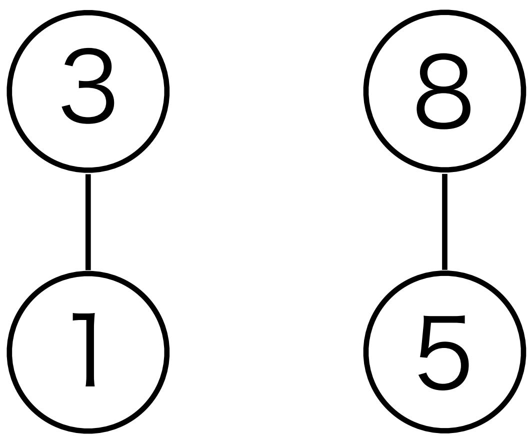
- 例) 6個の要素からなる二項ヒープ



二項ヒープ

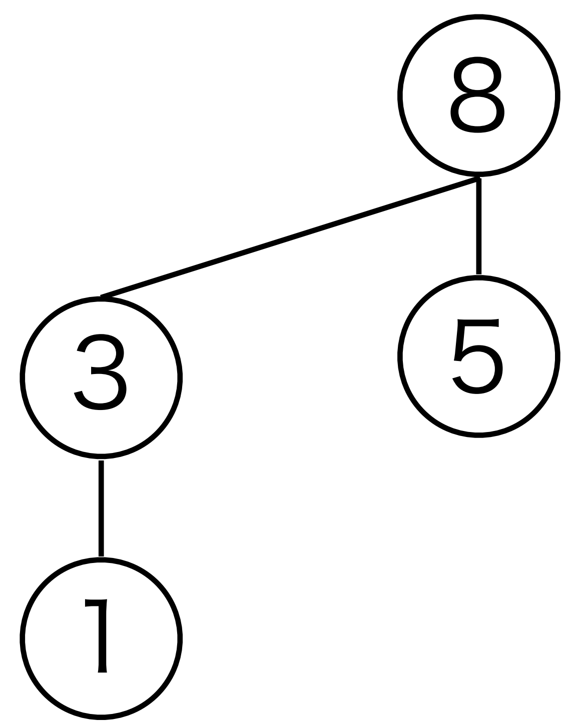
- 誤った例) 4個の要素からなる二項ヒープ

  同じ次数の二項木が2つあってはいけない



B_1 B_1

- 4個の要素からなる二項ヒープ



B_2

二項ヒープ

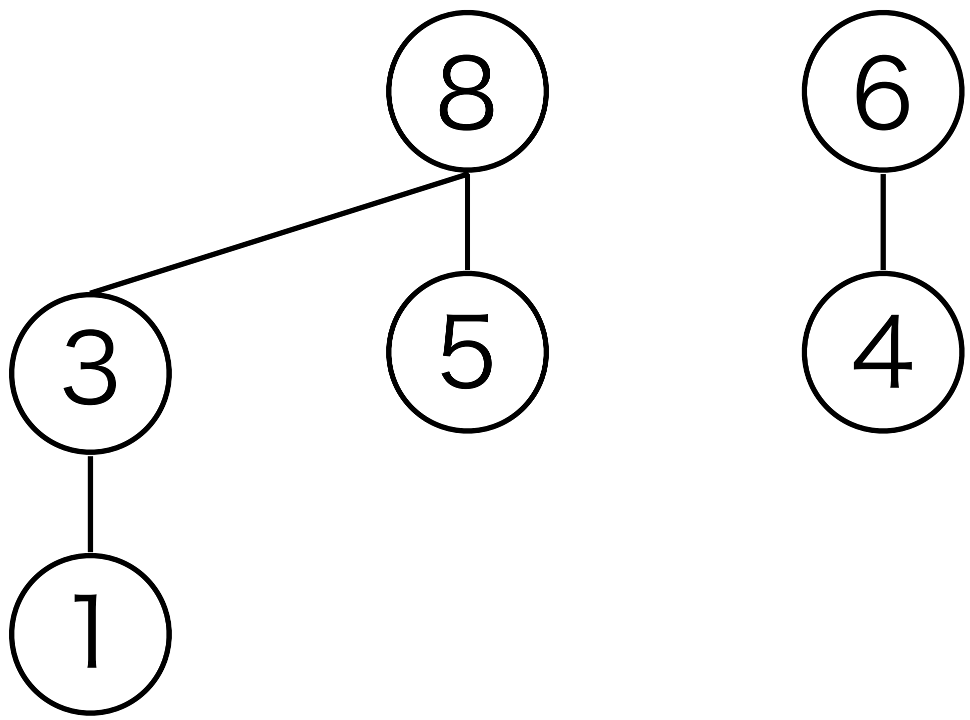
- 要素数が決まっていれば、どの二項木を用いるかは一意に定まる。
(2進数で表現した際に、1となる桁の次数の二項木を使う)

- 例)
$10 = 2^3 + 2^1$ なので、 B_3とB_1を用いる。
$17 = 2^4 + 2^0$ なので、 B_4とB_0を用いる。

- 要素数N個の二項ヒープを考えたとすると、
その二項ヒープに含まれる最大次数の二項木は $\lfloor \log N \rfloor$ なので、
二項ヒープに含まれる木の数、各二項木の高さ、各節点の
子接点の数はO(log N)となる。

最大値の取得

- 各二項木の根のいずれかが最大値になるので、根を全て調べれば良い



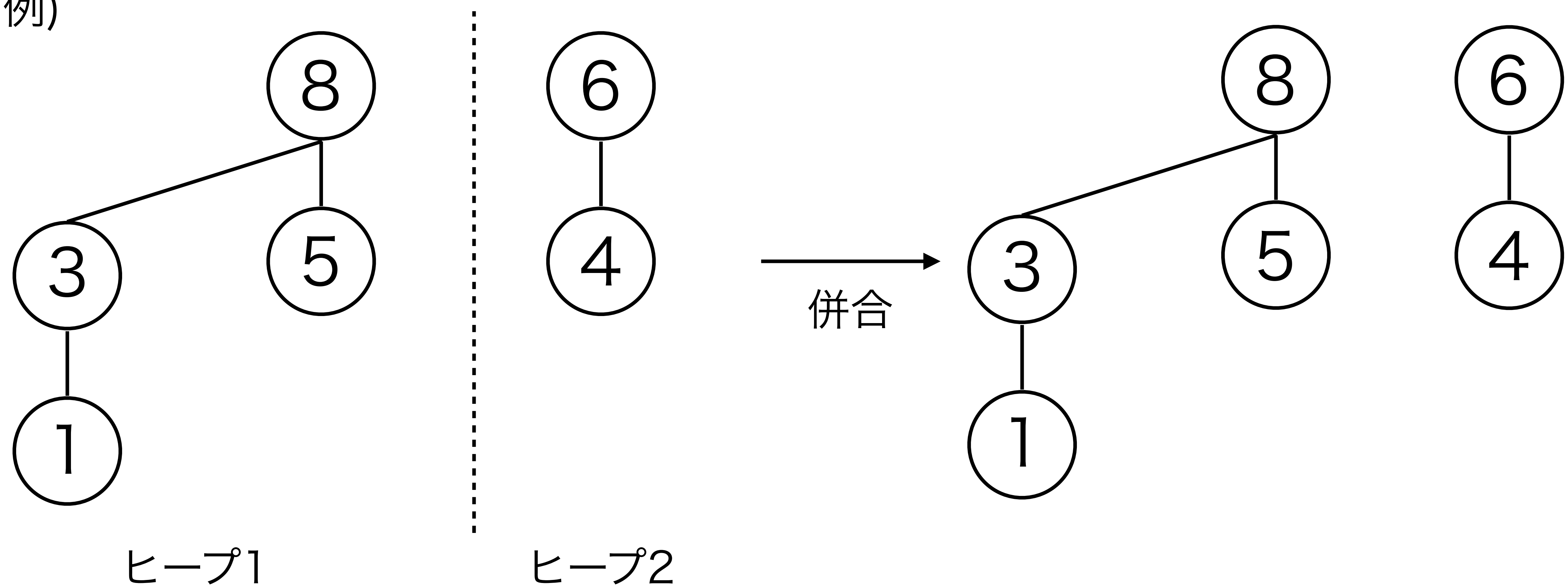
- そのため計算量はO(logN)だが、二項ヒープの形成時などに
あらかじめ最大値を探しておき、その根の位置を保持しておけば
O(1)で取得することも出来る。
(ヒープの更新時などにも、最大値を探す処理を行う必要がある。)

ヒープの併合

- 二項ヒープの特徴は、ヒープの併合が容易なことである。
  まず、二項ヒープに含まれる二項木がそれぞれ一つずつの場合を
  考える。

- さらに、2つの二項ヒープに含まれる二項木の次数が異なる場合を
  考えると、これは単に両方持つだけで良い

例)

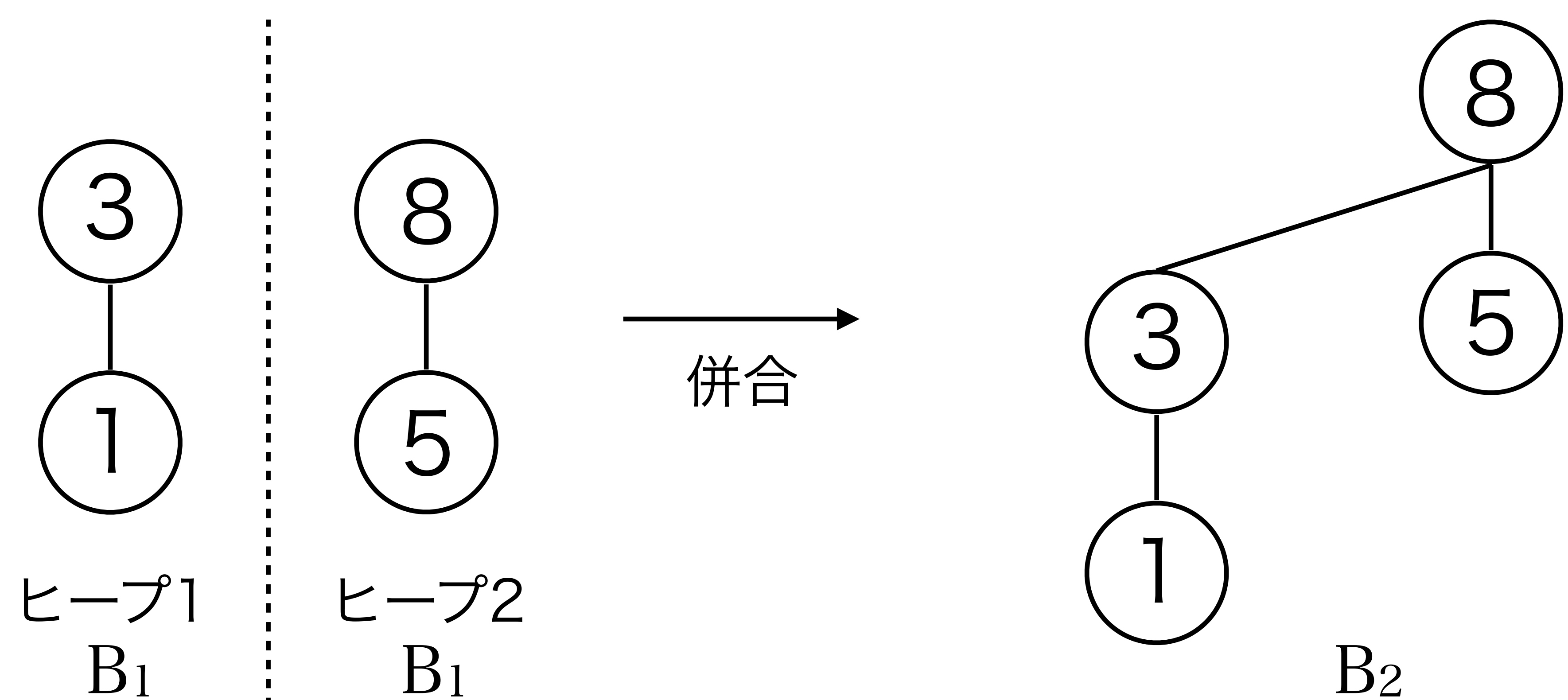


ヒープの併合

- 次に、二項ヒープに含まれる二項木がそれぞれ一つずつであり、2つの二項ヒープに含まれる二項木の次数が同じ場合を考える。

- 親子の値の大きさの関係を満たすように、片方のヒープの根をもう片方のヒープにつなげればよい
(2つのB_nの木から1つのB_{n+1}の木を作る)

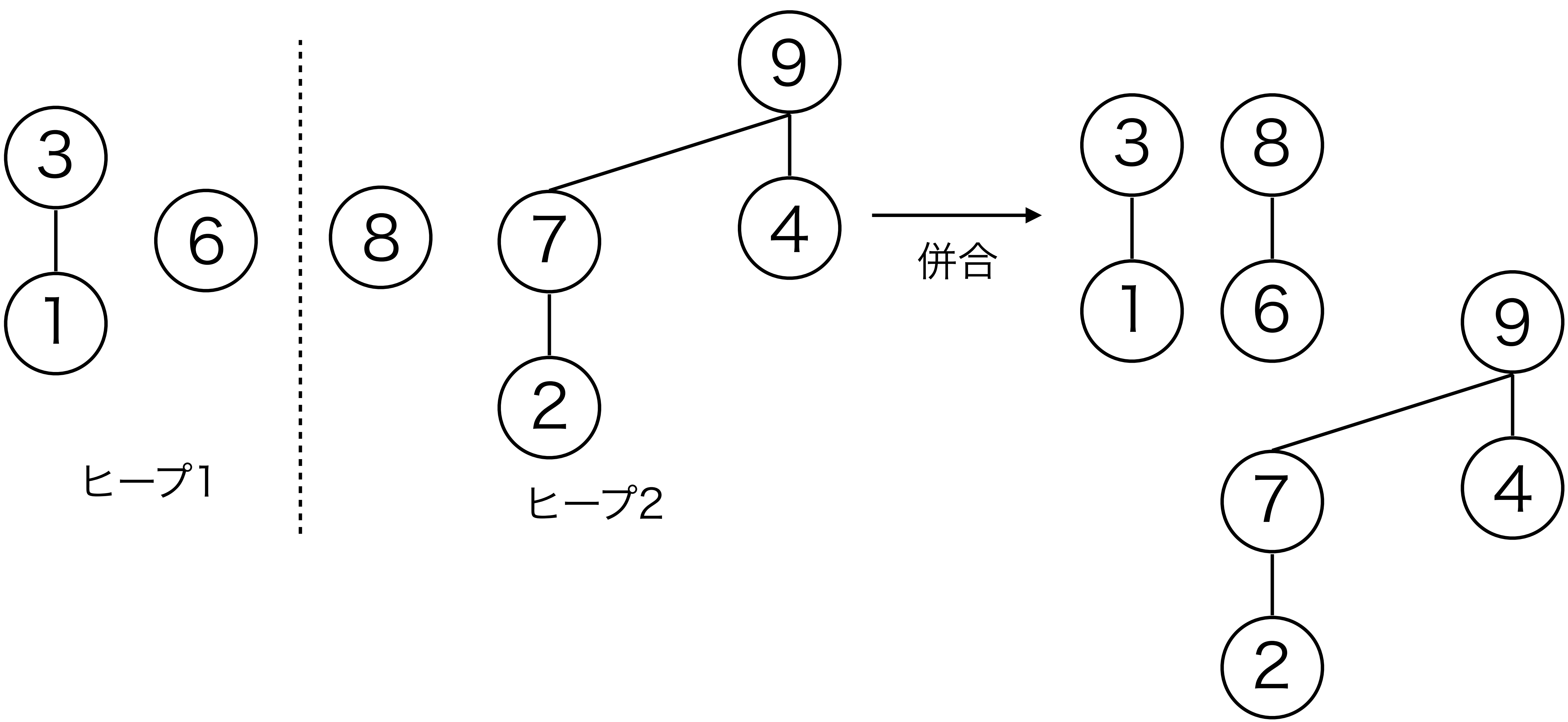
例)



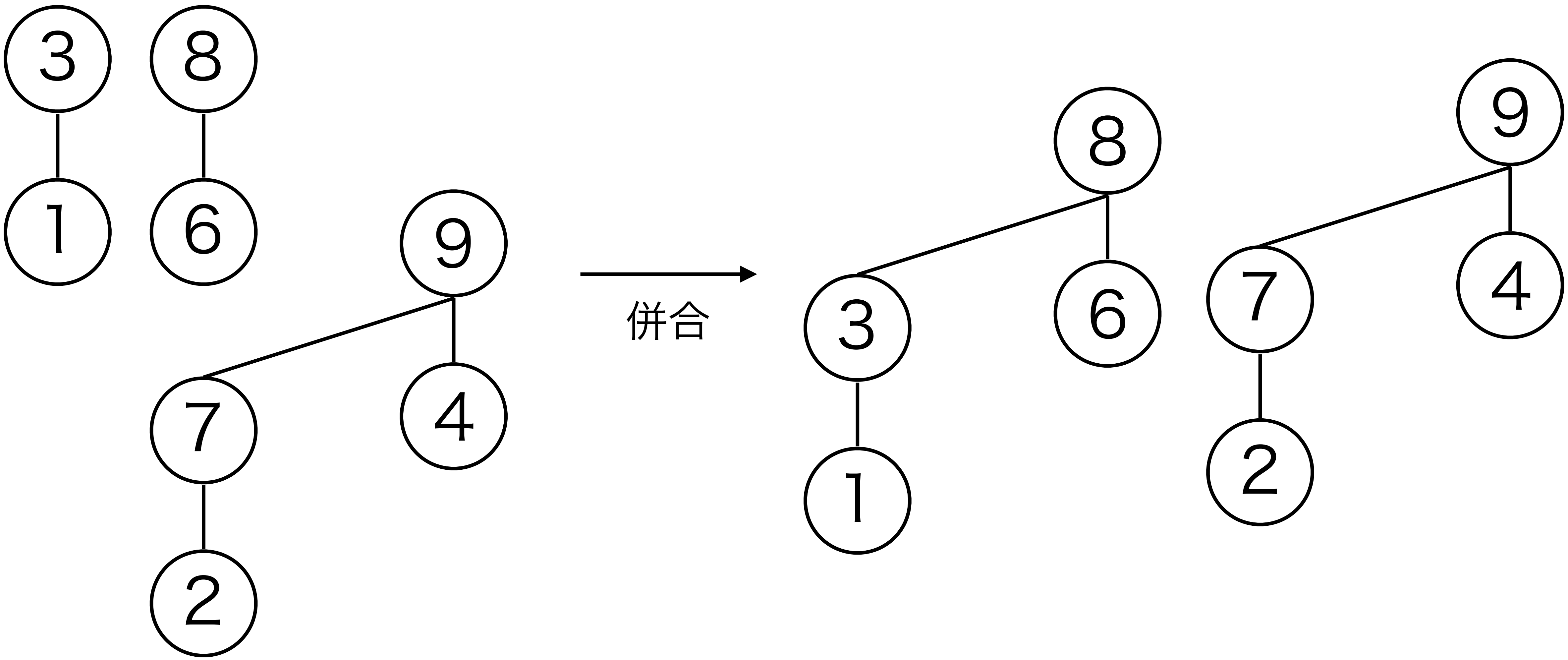
ヒープの併合

- 二項ヒープに含まれる二項木が複数の場合

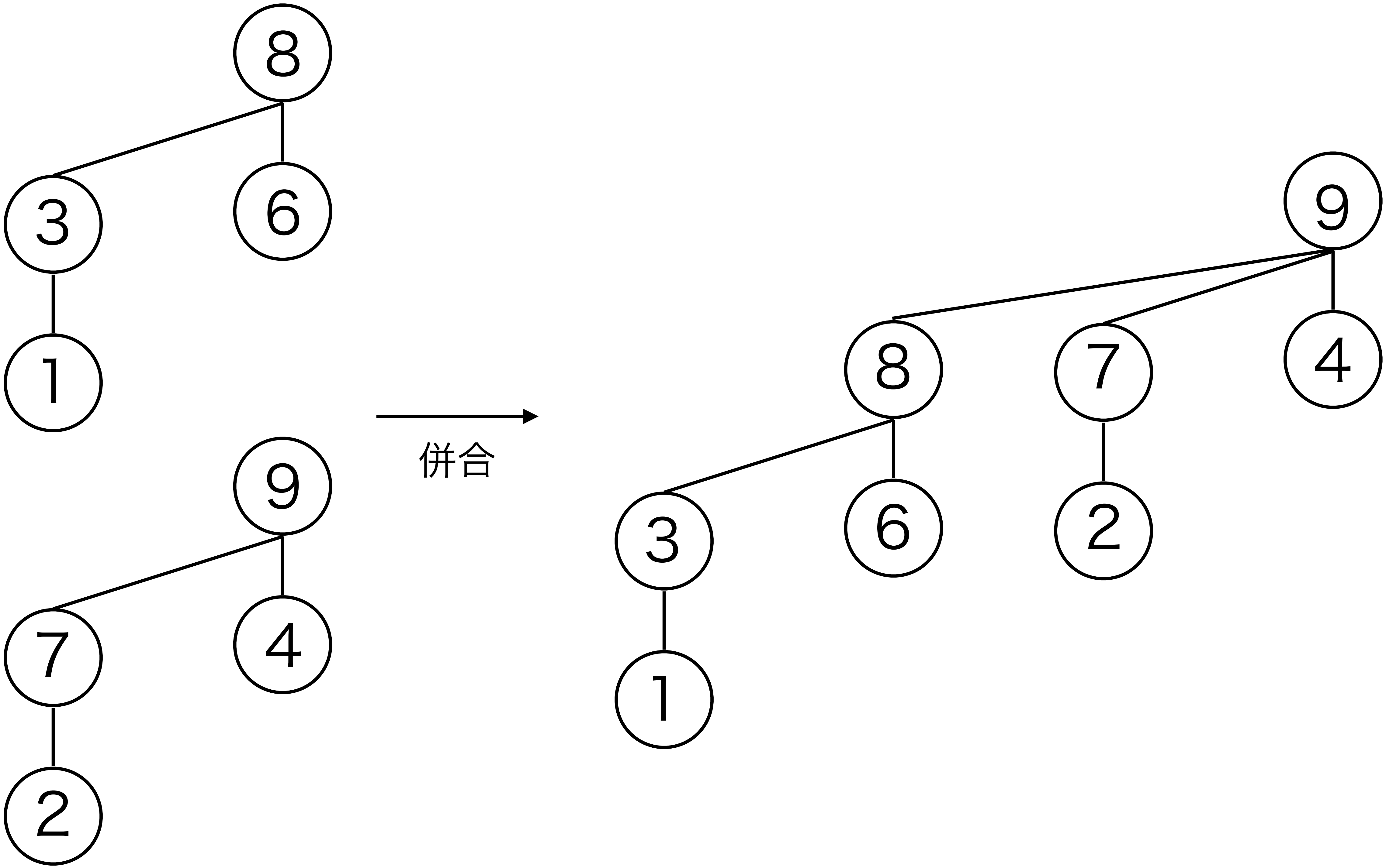
  1. 同一の次数(B_n)の木があれば、先ほどのように、
     その2つのB_nの木から1つのB_{n+1}の木を作る。
  2. 同一の次数の木がなくなるまで、1.の木の融合操作を繰り返す。



ヒープの併合



ヒープの併合



ヒープの併合

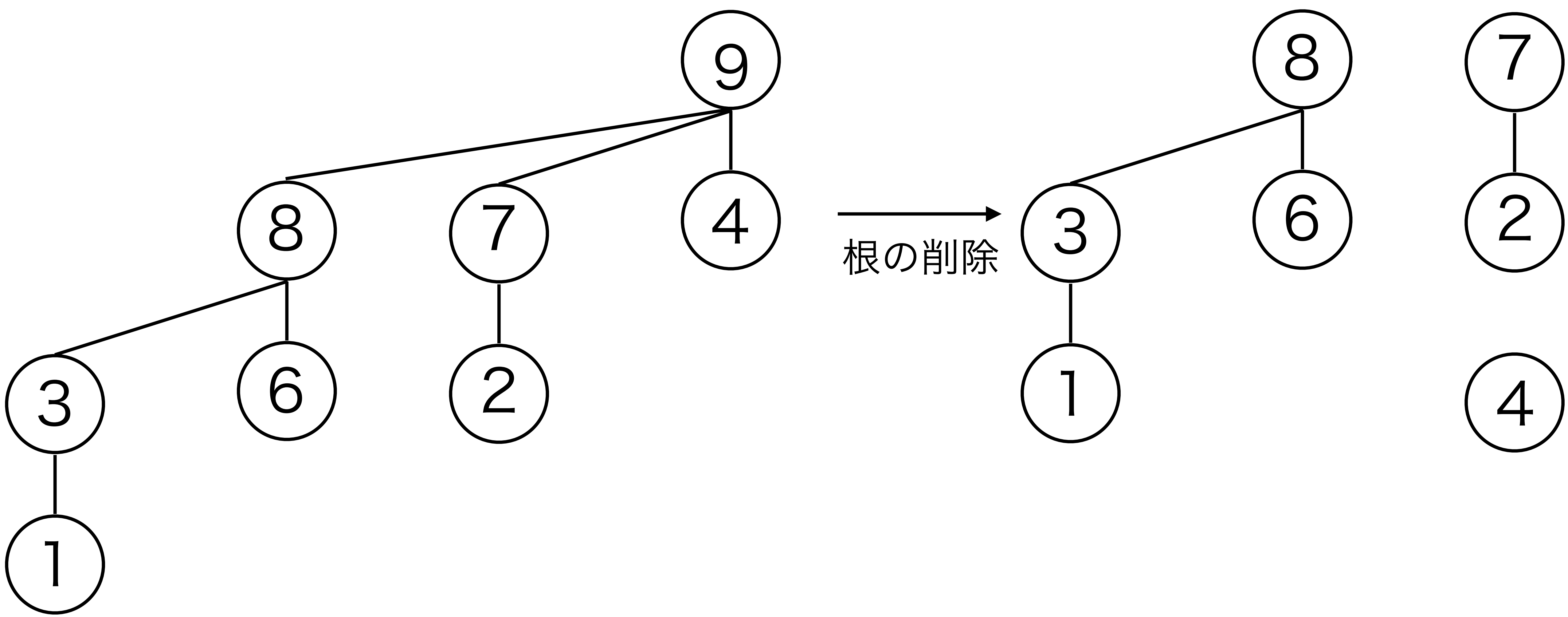
- ヒープの併合の計算量は、何回木の併合操作を行ったかと同じ
  木の併合回数は、元々のヒープにあった木の個数の総和を超えない
  よって、その計算量はO(logN + logM)となる。

- これは二分ヒープのO(N+M)に比べて効率が良い

- 二項ヒープへの要素の挿入は、その二項ヒープとB_0のみからなる
  二項ヒープの併合とみなせる。よってO(logN)

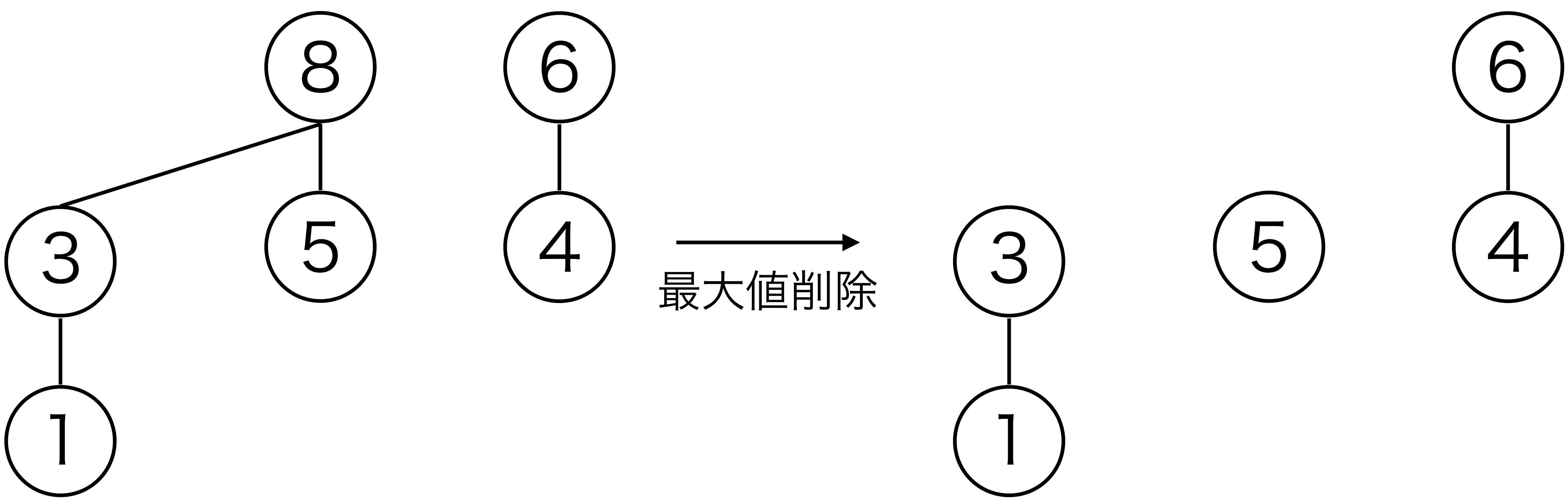
二項ヒープにおける最大値の削除

- 二項木は、根を削除すると残りが二項木の集合になるという性質を持つ



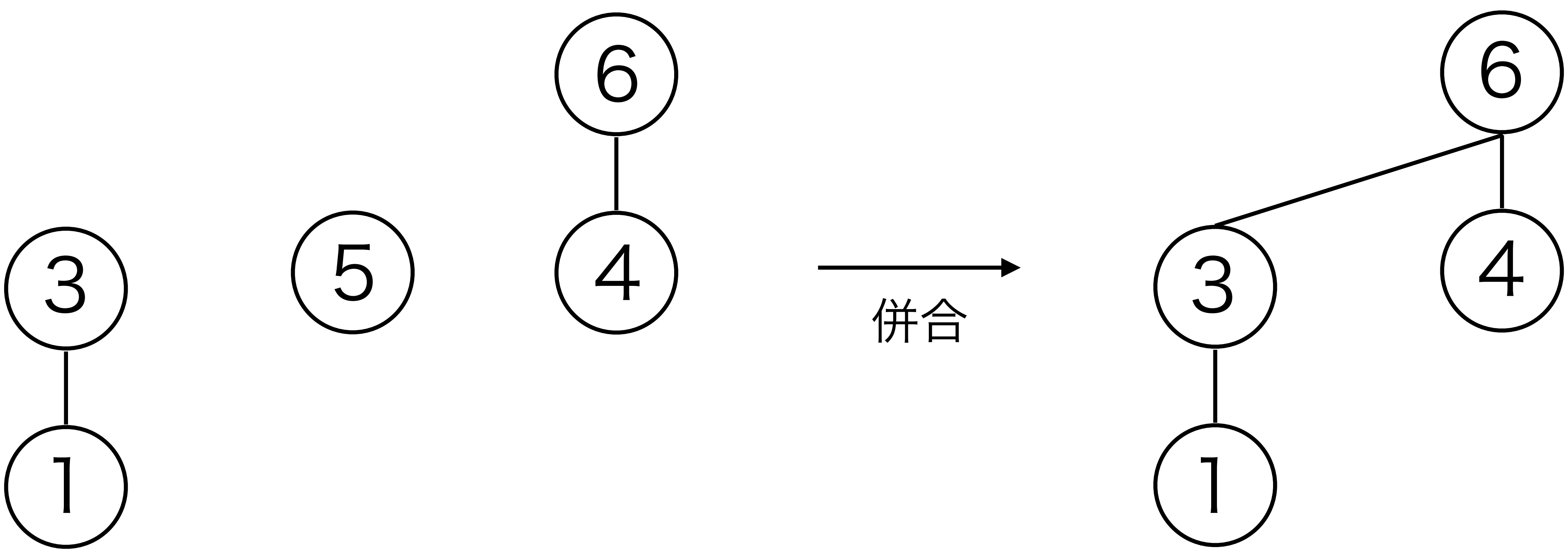
二項ヒープにおける最大値の削除

- よって、最大値を削除後、得られた二項木の集合と元々あった二項木を併合させる処理を行うことで、再び二項ヒープが得られる。



二項ヒープにおける最大値の削除

- よって、最大値を削除後、得られた二項木の集合と元々あった二項木を併合させる処理を行うことで、再び二項ヒープが得られる。



- この操作はO(logN)となる(各自考えてください)

二項ヒープの構築

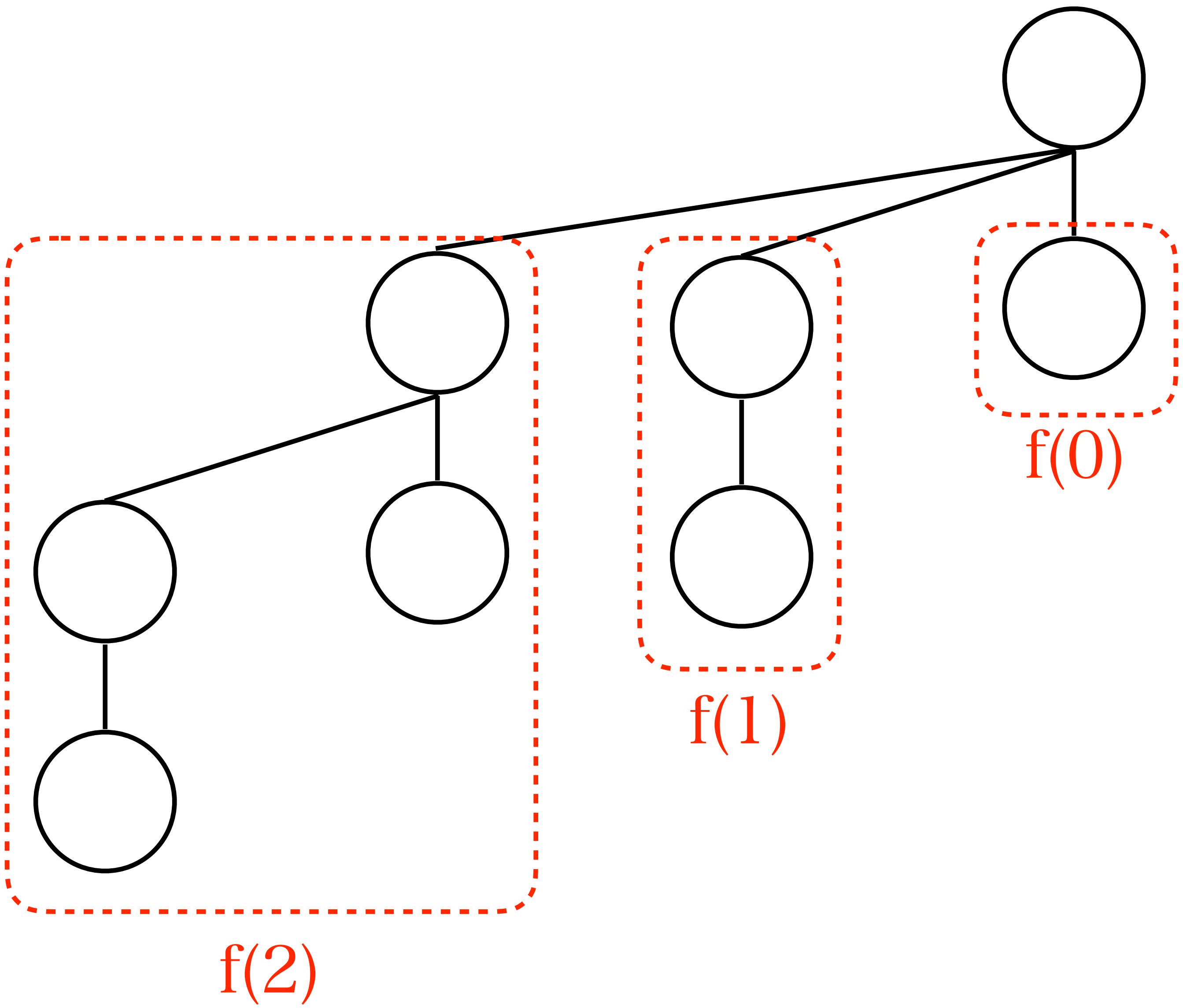
- 要素数Nの二項ヒープを作成する時、まず対応する二項木の集合を与え、要素を適当に入れて初期化する。
- その後、二分ヒープの時と同様に、葉の方から見ていって、大小関係が正しくない部分を入れ替えていく。

- その最悪計算量について、まず単一の二項木で構成されているケースを考え、再帰的に考える。

- B_nの構築にかかる最悪比較回数を$f(n)$とすると、明らかに$f(0) = 0$

二項ヒープの構築

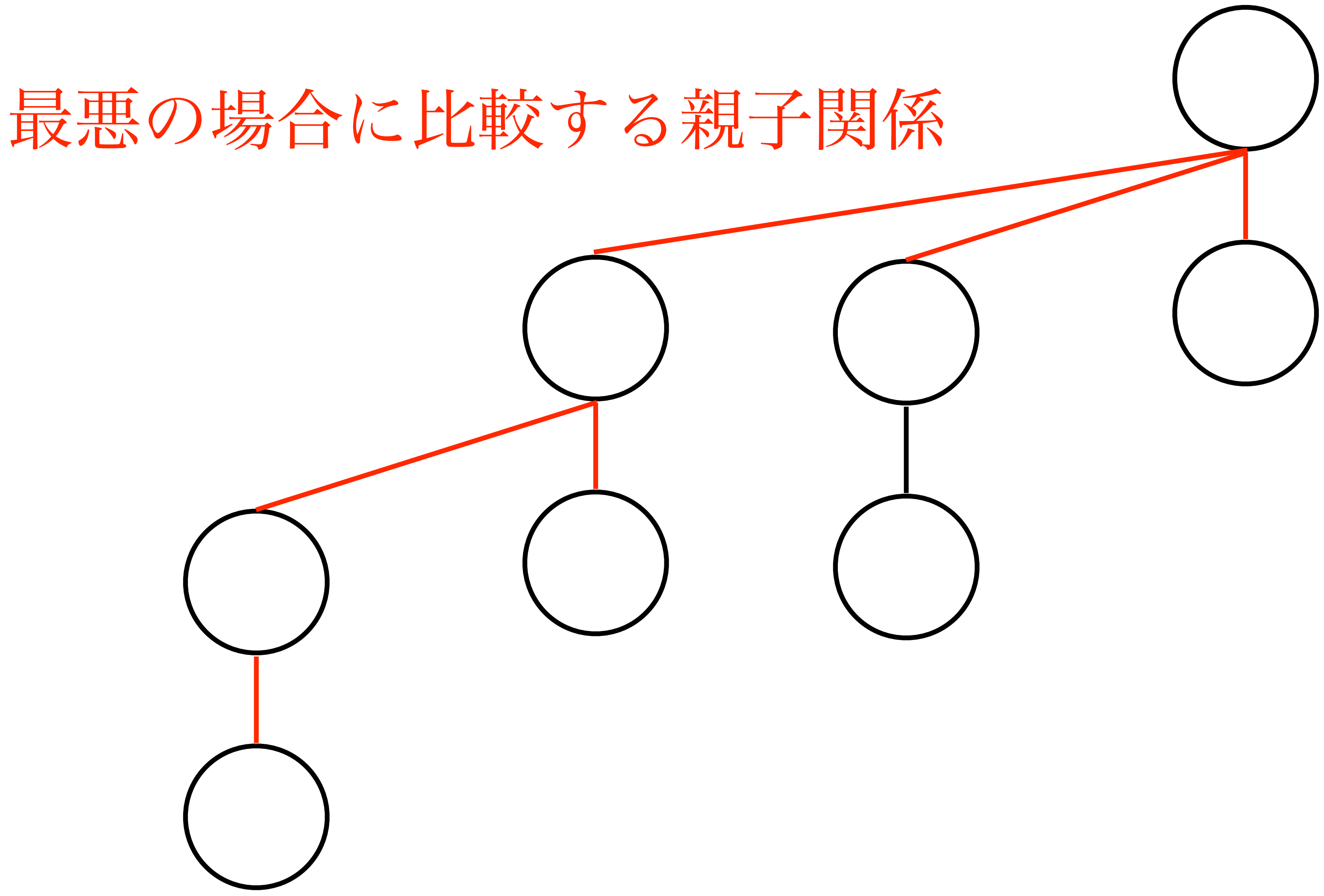
- 例としてB_3の場合を考え、まず根以外の部分を考える。



その計算量は、f(0) + f(1) + f(2)となる。

二項ヒープの構築

- 最後に根について考えると、その子供のうち最も大きいものと交換を繰り返すので、最悪 3+2+1 = 6回の回数の比較が必要となる。



二項ヒープの構築

- これを一般化すると、B_n の構築に必要な計算時間は、

$$\underbrace{\sum_{i=0}^{n-1} f(i)}_{\text{根以外}} + \underbrace{{}_{k+1}C_2}_{\text{根の部分:}}$$

根以外 根の部分:

k+ (k-1)+⋯+1

- この漸化式を解くと、$f(k) = 2^{k+1} - k - 2$となり、よってO(N)である

- 複数の二項木から構成されている場合も、各二項木は独立に
ヒープ化出来来、各二項木の計算時間はその木のサイズに線形なので、
全体で見ればO(N)で構築可能である。

試験について

- 第1回でお伝えした通り、1/24 16:30-18:00 @ 56-101を予定

- 試験範囲は第1回から第13回(本日)の講義分まで
  次回(第14回)は、「生命情報科学におけるアルゴリズム」について講義することにします(試験には出しません)。

- 講義で説明していない内容は(教科書内でも)試験には出題しません。
  またソースコードを書かせる問題も出題しません。
  発展的な問題を出題するというより、講義の内容を理解しているかということを広く問う試験を出す予定です。

- COVID-19の感染状況が悪化し大学が学生の登校を禁止した場合、
  試験は中止し、レポートに切り替えます。

試験範囲の例外

- 下記の内容は授業スライドにありますが試験範囲には含めません。
  - 第2回: 組み合わせの全探索
    (部分和問題そのものは再帰や動的計画法でも説明しているため試験範囲である。)
  - 第4回: 二分探索法の例(2):射撃王
    乱拓アルゴリズムの行列積計算において、行列積が一致する確率が高々1/2であることの証明
  - 第5回: キューの実装におけるリングバッファ
  - 第9回: ダイクストラ法の正当性の証明
  - 第10回: クラスカル法／辺連結度を求めるアルゴリズムの正当性の証明