プログラム設計とアルゴリズム
第6回 (11/1)

早稲田大学高等研究所　講師
福永津嵩

(前回の復習) 配列

- 要素を順番に並べたデータ構造。

- C++でいうvectorであり、プログラミング入門で習った配列そのもの

- a= (4, 3, 12, 7, 11, 1, 9, 8, 14, 6)とすると、
  a[0] = 4, a[1] = 3, a[2] = 12として要素にアクセス可能である。

- C/C++/Pythonの場合は、要素がメモリ上で連続に並んでいる。

図8.1

(前回の復習)連結リスト

- 連結リストは配列とは違い、要素の挿入・削除に強いデータ構造である。
- リストでは、要素間の前後関係はポインタという矢印で繋がれている。
- ここで、「要素」と「次の要素を指し示すポインタ」の組をノードと呼ぶ。

図8.4

(前回の復習)ハッシュテーブル

- 要素の検索もO(1)で行うためのデータ構造

- まず、格納する要素がM未満の整数に限られる場合を考える。

$\boxed{\text{ハッシュテーブルのアイデアを示す配列}}$

- この方法では、まず全ての値をfalseで初期化した後、
  挿入: T[x]にtrueを代入
  削除: T[x]にfalseを代入
  検索: T[x]の値を調べる
  ことで、O(1)で挿入・削除・検索を行うことが可能となる。
  (ただし、要素を順番に調べると言ったようなことは苦手である)

(前回の復習)ハッシュテーブル

- 先ほどの方法を、要素がどのようなデータであっても対応できるよう拡張したものがハッシュテーブルである。
- 格納したい要素xに対して、何らかの関数h(x)を定義する。ただし、0 <= h(x) < Mを満たすものとする。
- このh(x)をハッシュ関数と呼び、xをキー、得られたh(x)の値をハッシュ値と呼ぶ。
- 異なるキーxに対して、ハッシュ値h(x)が必ず異なる値になるハッシュ関数を完全ハッシュ関数と呼ぶ。

(前回の復習)ハッシュの衝突対策

- 衝突した場合データを捨てるのは困るので、何らかの対策が必要になる
- 連鎖法は、ハッシュとリストを兼ね備えたデータ構造
  (true/falseだけでなく要素も格納する)

図8.11

(前回の復習) 基本的なデータ構造

表8.1

- データ構造によって、得意(可能)な処理が異なる。
- よってプログラム設計においては、どのような処理を行うかで用いるデータ構造を使い分ける必要がある。

(前回の復習)スタックの挙動

図9.3

- スタックでは要素の追加／取り出しはpush／popと呼ばれる。

(前回の復習)キューの挙動

$\boxed{\text{図9.4}}$

- キューでは要素の追加／取り出しはenqueue /dequeueと呼ばれる。

第十章
データ構造(3):
(グラフ)・木

木とは何か

- 頂点と、その頂点を結ぶ辺(枝)からなるデータ構造であり、全ての頂点が連結しており、循環路(サイクル)を持たないものを木と呼ぶ。

図10.13

木の用語定義

- 木の頂点のうち、特別な一つの頂点を根とすることがあり、
  慣例上根は最上部に描画される。

- 根を持つ木は根付き木(有根木)、根を持たない木を根なし木(無根木)
  と呼ぶ。

- 辺によって結ばれている頂点のうち、根に近い側を親と呼び、
  根から遠い側を子と呼ぶ。また同じ親を持つ頂点を兄弟姉妹と呼ぶ。

- 一本しか辺が接続していない頂点を、葉と呼ぶ。

木の用語定義

図10.14

木の用語定義

- 木の各頂点について、ある頂点xから子供の方だけをみると、
xを根とした木とみなすことができる。これをxを根とする部分木と呼ぶ。

- xの部分木に含まれる頂点を、xの子孫と呼ぶ。

- 根と各頂点を結ぶ辺の数を、その頂点の深さと呼ぶ。

- 各頂点の深さの最大値を、その木の高さと呼ぶ。

木の用語定義

図10.15

順序木

- 根つき木において、兄弟姉妹間において順序関係が存在するとき、それを順序木と呼ぶ。順序木の表現はポインタを利用することが多い。

図10.16

- 親に子へのポインタを全て持たせる実装も多くみられる

k分木

- 全ての頂点について、高々k個の頂点しか持たないものをk分木と呼ぶ。
- つまり、(このような呼び方はしないが)連結リストは一分木である。
- この中でも二分木は、データ構造において広く利用される木構造である。
- 二分木において、根から見て右の部分木を右部分木と呼び、
  根から見て左の部分木を左部分木と呼ぶ。

平衡木

- 木のデータ構造では、クエリの計算量はO(h)となることがほとんど(hは木の高さ)

- 要素数が同じでも、hは大きく異なりうる。

図10.17

平衡木

- 左右の頂点数のバランスが良い木は、木の高さが低くなりやすい。
- 二分木であり、全ての葉の深さが高々一しか異ならないものを、
強平衡二分木と呼ぶ。
- また、全ての葉の深さが全く同一であるものを完全二分木と呼ぶ。
(教科書によって定義が異なることがある)
- 完全二分木の高さをhとすると、
$N = 2^{h+1} - 1$であることから、$h = O(\log N)$ である。
(強平衡二分木もほぼ同様)

二分木を用いるデータ構造(1): 二分ヒープ

- データから最大値(最小値)を取得するのに適したデータ構造

- 各頂点xが値key[x]を持つ二分木であり、次の条件を満たす

  1. xの親頂点をpとしたとき、key[p] >= key[x]が成立する。
     (不等号を逆にすると最小値の取得になる)
  2. 木の高さをhとすると、h-1以下の部分は完全二分木である。
  3. 高さhの部分は、頂点が左詰されている。

- すなわち、二分ヒープは強平衡二分木である。

- 兄弟姉妹の順序などはkey[x]に依存せずどうでも良いことに注意

二分ヒープの例

図10.18

- 定義から明らかに、根が最大値になるので、O(1)で最大値を取得することが可能である。
- 要素の検索のような操作には向いていない。

配列によるヒープの実現

図10.19

- 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。

ヒープの挿入処理

- ヒープに要素を挿入するとき、親子の
  順序が保たれなければいけない。

- まず、最後尾に要素を追加する。
  その後、親と順序が異なるようなら
  親と子で要素を入れ替える。

図10.20左

- 親子の順序が満たされるまで、
  この親と子の入れ替えを行う。

- 計算量はO(log N)となる。

ヒープの最大値削除処理

- 根の要素を削除し、最後尾の要素を根に持ってくる。

- 順序関係が保たれていないなら、2つの子のうち大きい方と順序を入れ替える。

$\boxed{\text{図10.20右}}$

- 親子の順序が満たされるまで、この親と子の入れ替えを行う。計算量はO(log N)となる。

O(N)時間でヒープの構築

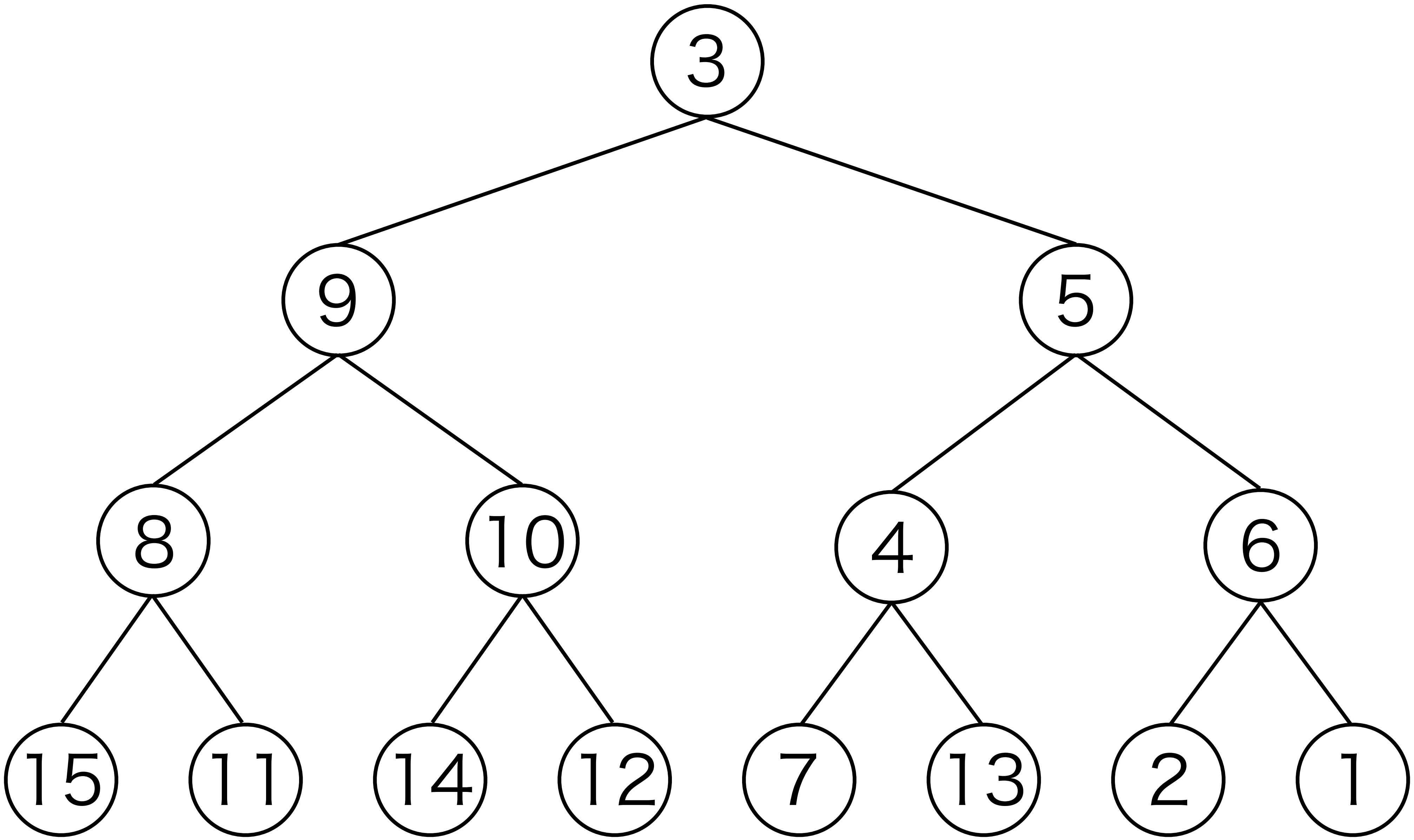
- N個の要素に対して要素の挿入を行ってヒープを構築すると O(NlogN)の時間がかかる。
- 一方で、N個の要素が格納された配列を、ヒープを満たすように並べ替える処理は、以下のようにO(N)で行うことができる。
- ある頂点xの部分木を考えたとき、xの左右の部分木についてはヒープ条件が満たされており、xとxの子の間にのみヒープ条件が満たされていない可能性があるとする。
- この部分木をヒープにする操作をheapifyと呼ぶとすると、その操作は最大値削除処理の操作と同一である。

O(N)時間でヒープの構築

- 配列の後ろの要素から順番にheapifyの操作を行うとする。
  このとき、要素が葉であれば何も要素を行わない。

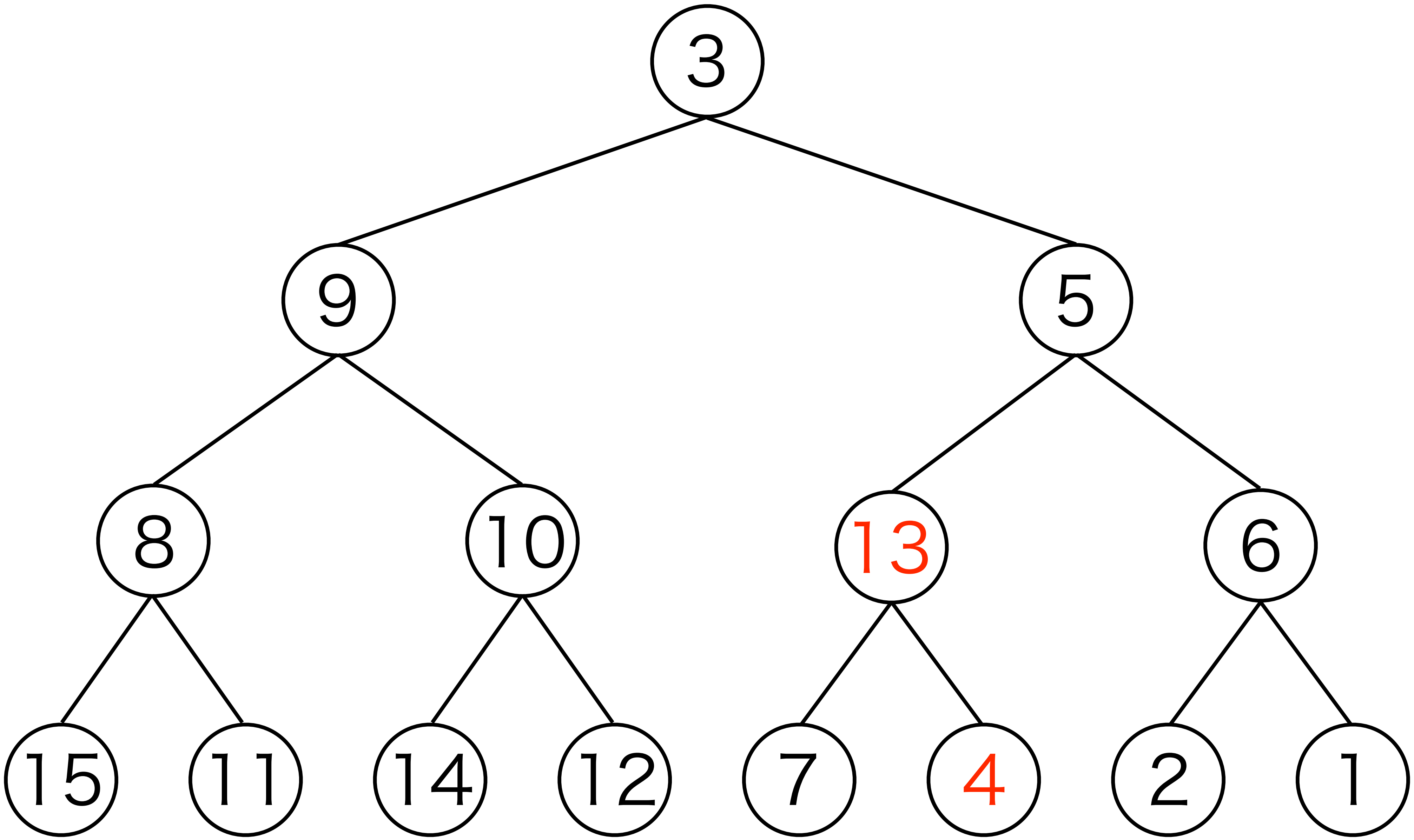
- 後ろから順番にheapifyを行うので、左右の部分木がヒープ条件を満たすという条件は成立している。全ての要素に対してheapifyを行うと、その配列はヒープ要件を満たす。

O(N)時間でヒープの構築:具体例



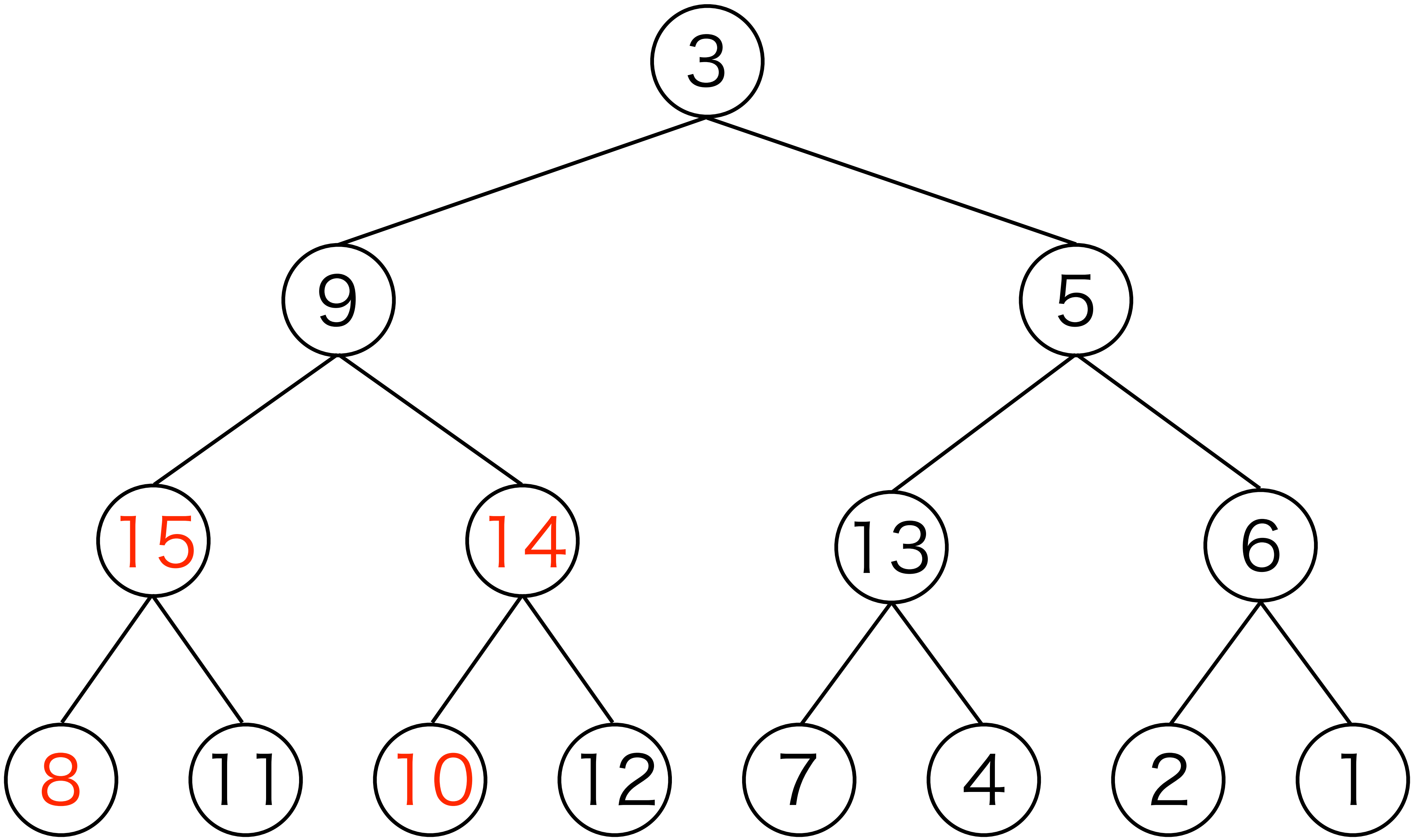
- 上の二分木をheapifyを利用してヒープに入れ替えることを考える。
- 後ろから順番に見ていくが、葉の操作はしないので、下から2段目から見ていく。

O(N)時間でヒープの構築:具体例



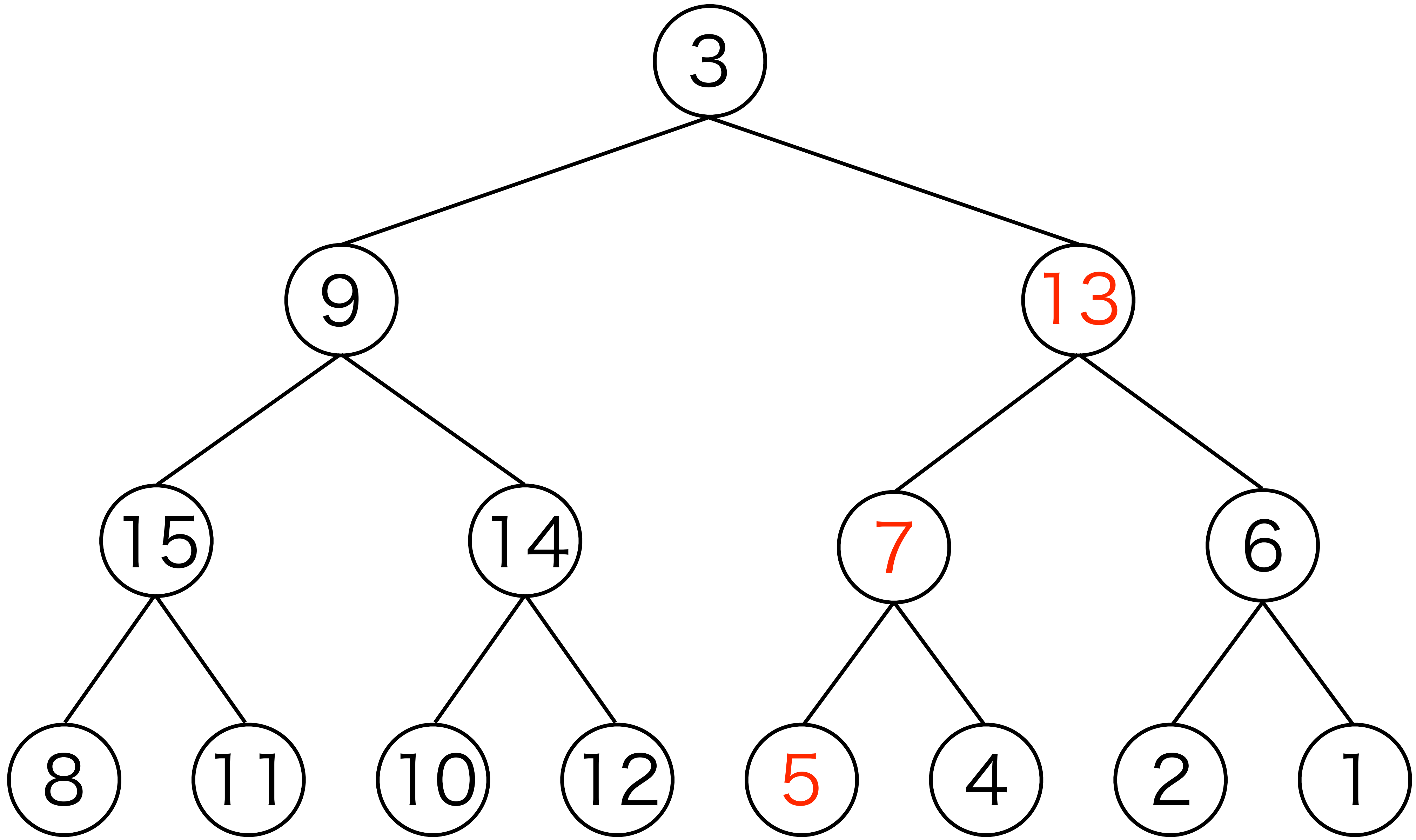
- 6は順序を満たすので入れ替えない。4は満たさないので、子のうち大きい方の13と入れ替える。

O(N)時間でヒープの構築:具体例



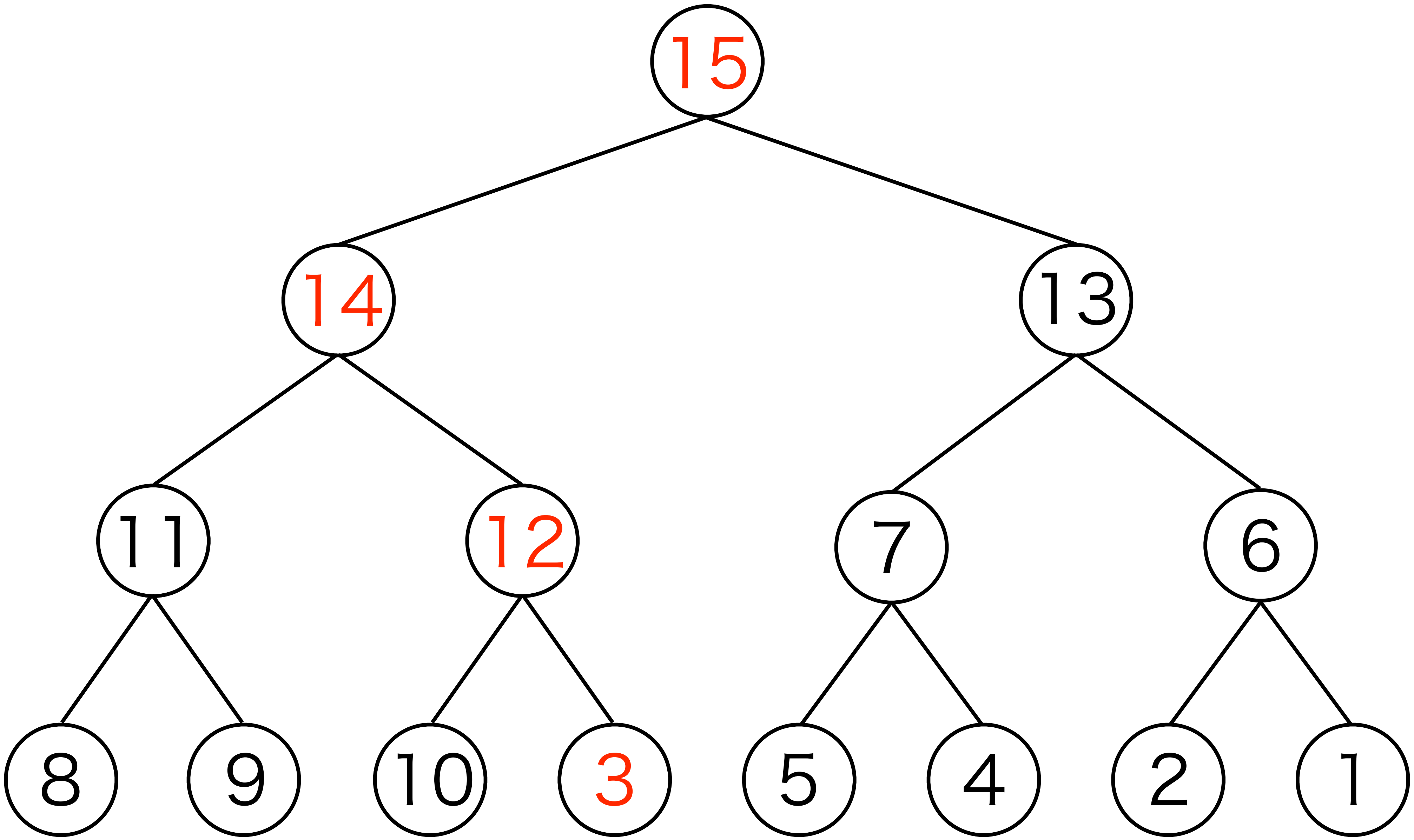
- 10と8も同様に入れ替える。これで下から２番目の操作は済みとなる。

O(N)時間でヒープの構築:具体例



- 上から2段目の行に写る。5は要件を満たさないので13と入れ替え、入れ替えた先でも要件を満たさないので7と入れ替える。

O(N)時間でヒープの構築:具体例



- 最後に、根の部分を入れ替えてヒープが完成する。

O(N)時間でヒープの構築

- heapifyを利用したヒープの構築の計算量は次の通りとなる。

- 二分木の高さをhとすると、深さdの頂点でheapifyを行ったとき、交換が起きる回数は高々h−d回である。

- 深さdの節点数は高々2^d個であるため、交換の起きる回数は高々

  $2^{h-1}*1 + 2^{h-2}*2 + 2^{h-3}*3 + \cdots$

  よって全体の計算量は$O(2^h) = O(N)$となる。

ヒープソート

- ヒープを利用した次のソートを、ヒープソートと呼ぶ。

  1. 与えられた配列からヒープを構築する。(O(N)の計算量)
  2. 最大値を順に取り出して配列の後ろから詰めていく
     (O(NlogN)の計算量)

- 全体の計算量はO(NlogN)となる。

- O(NlogN)のソートは他にもあり、ヒープソート自体は平均的に遅いため全体をソートしたい時にはあまり使われない。

- ただし、大きい方から上位K個を取り出してソートしたいという時には、O(KlogN)でソートが可能という特徴を持つ。

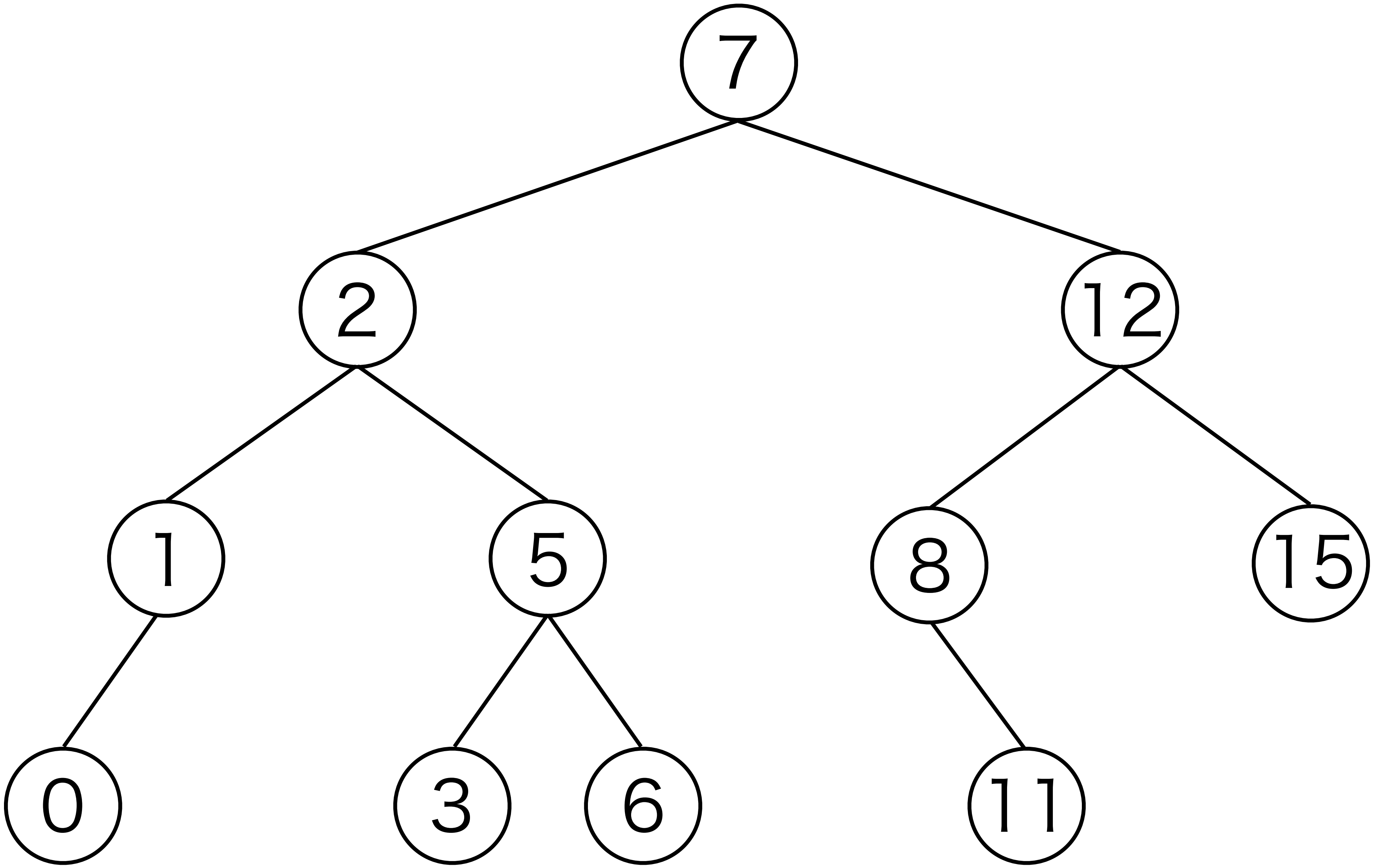
二分探索木

- ハッシュテーブルや連結リストと同様に、要素の挿入・削除・検索をサポートするデータ構造

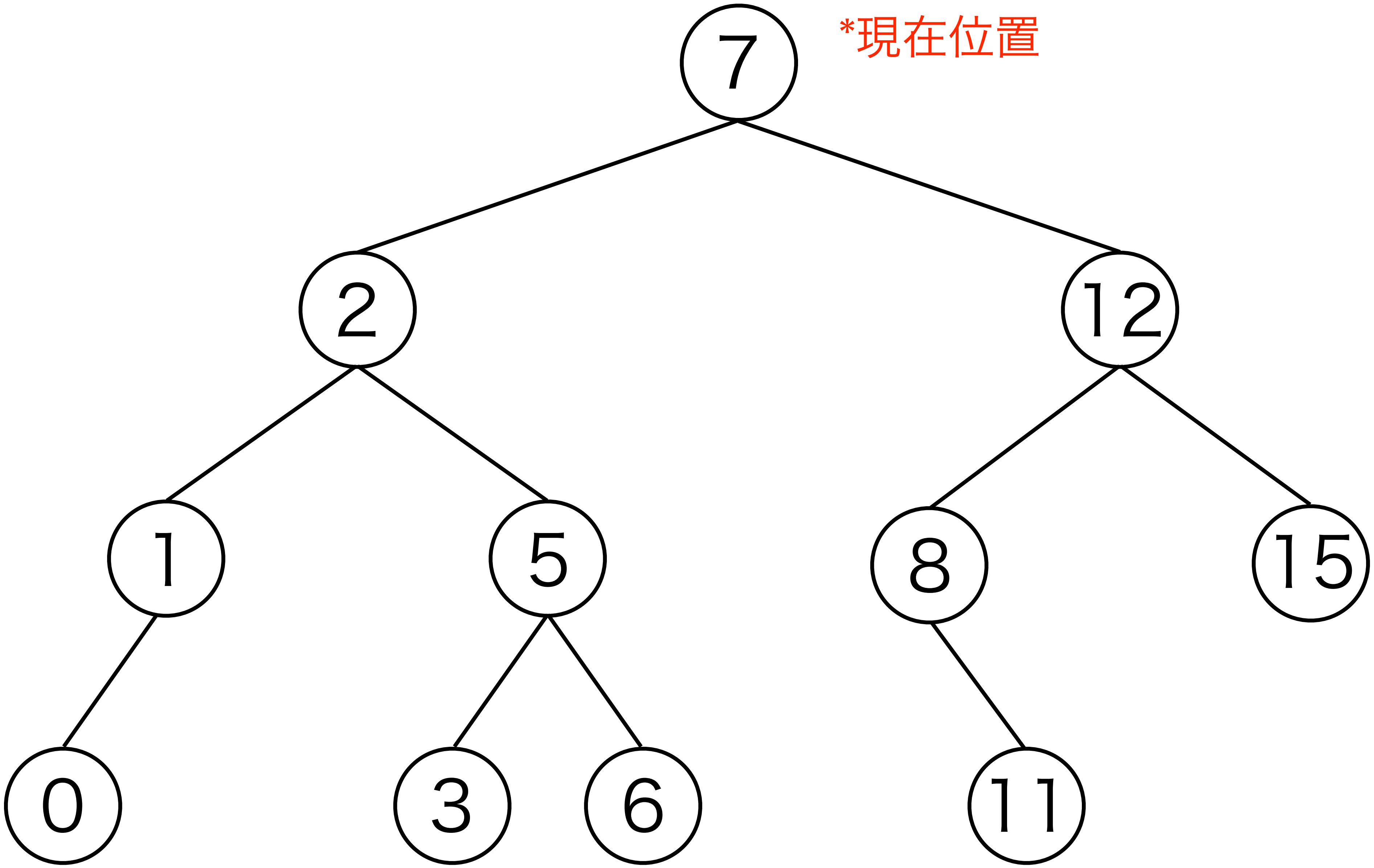
- 二分探索木は、各頂点vが値key[v]を持つ二分木であり、次の条件を満たすもののことを言う

二分探索木の条件

二分探索木の例

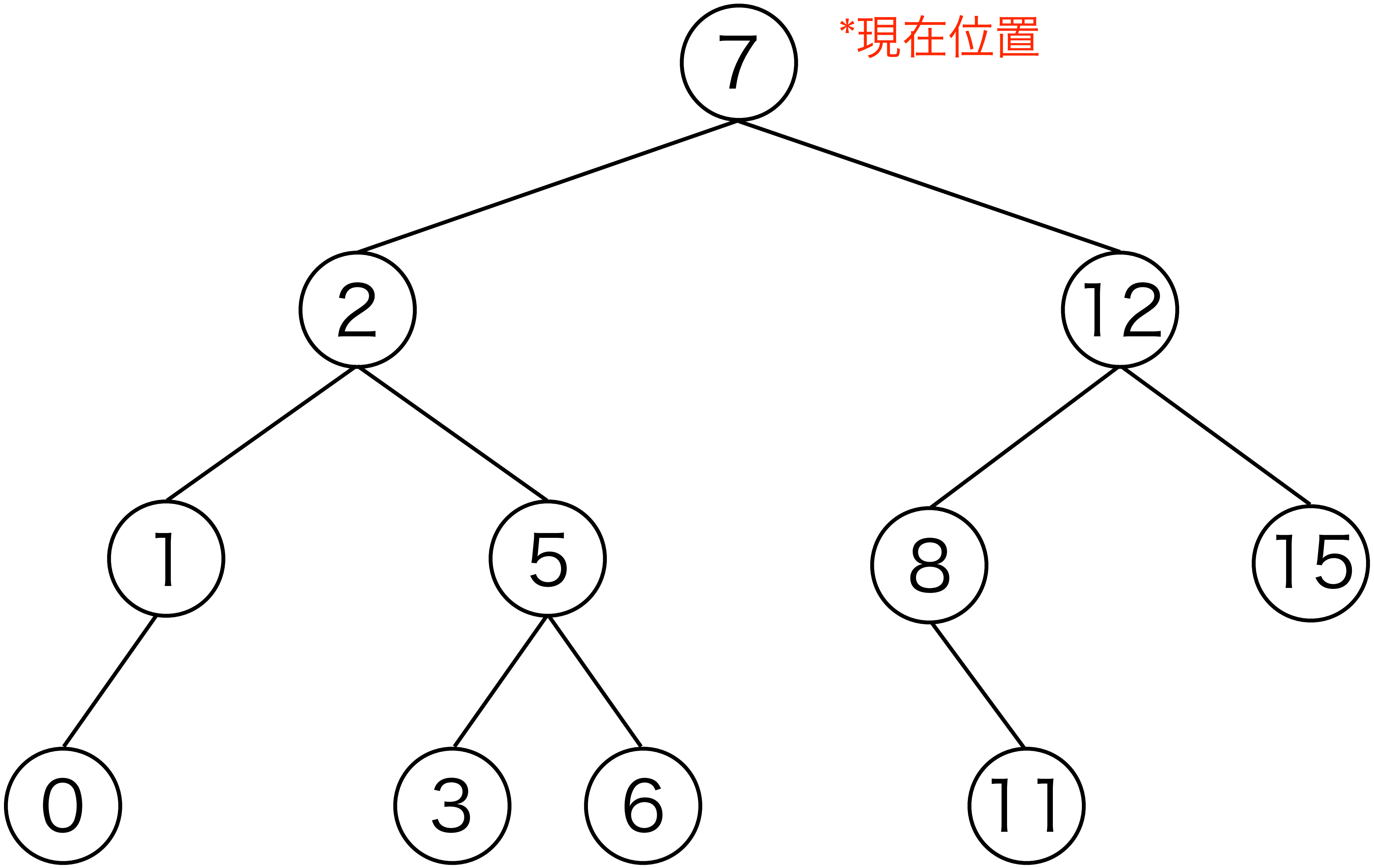


二分探索木における要素の検索



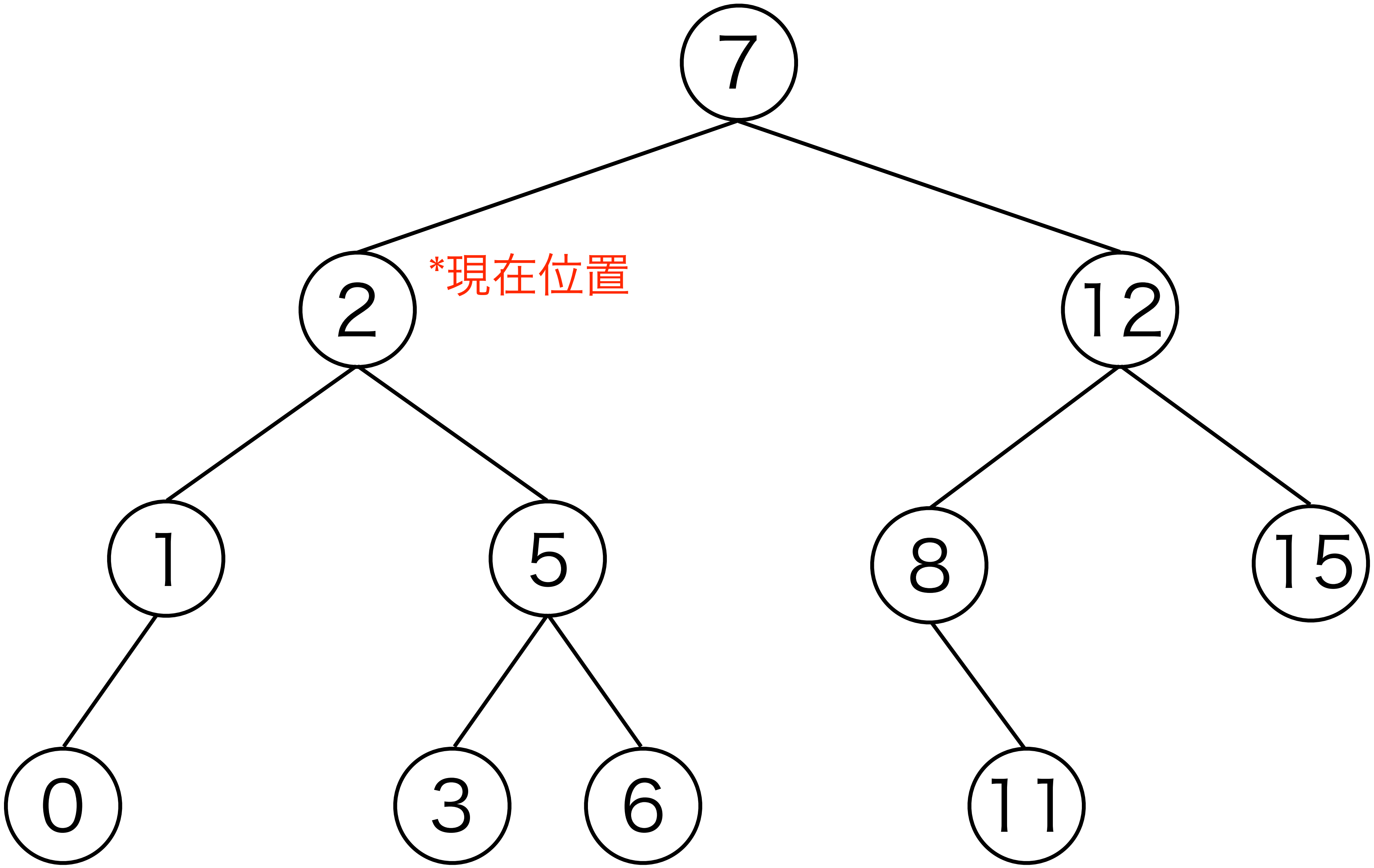
- 二分探索木の中に3が含まれているかどうかを判定したいとする。
  まず、根からスタートする。

二分探索木における要素の検索



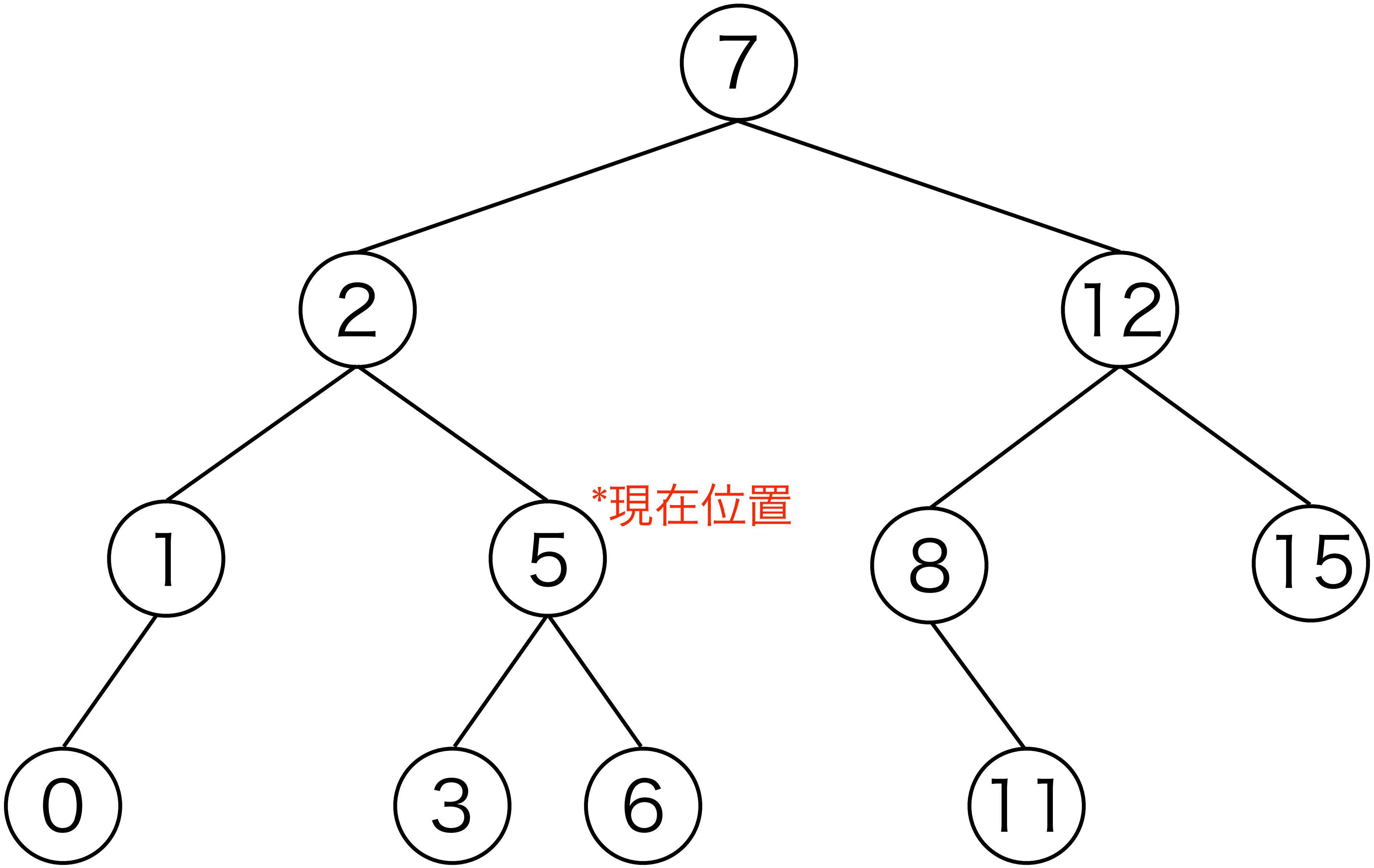
- 現在位置の7と3を比較すると、3の方が小さい。よって、3が含まれているなら左部分木に存在する。そのため、現在位置を左部分木へ移動する。

二分探索木における要素の検索



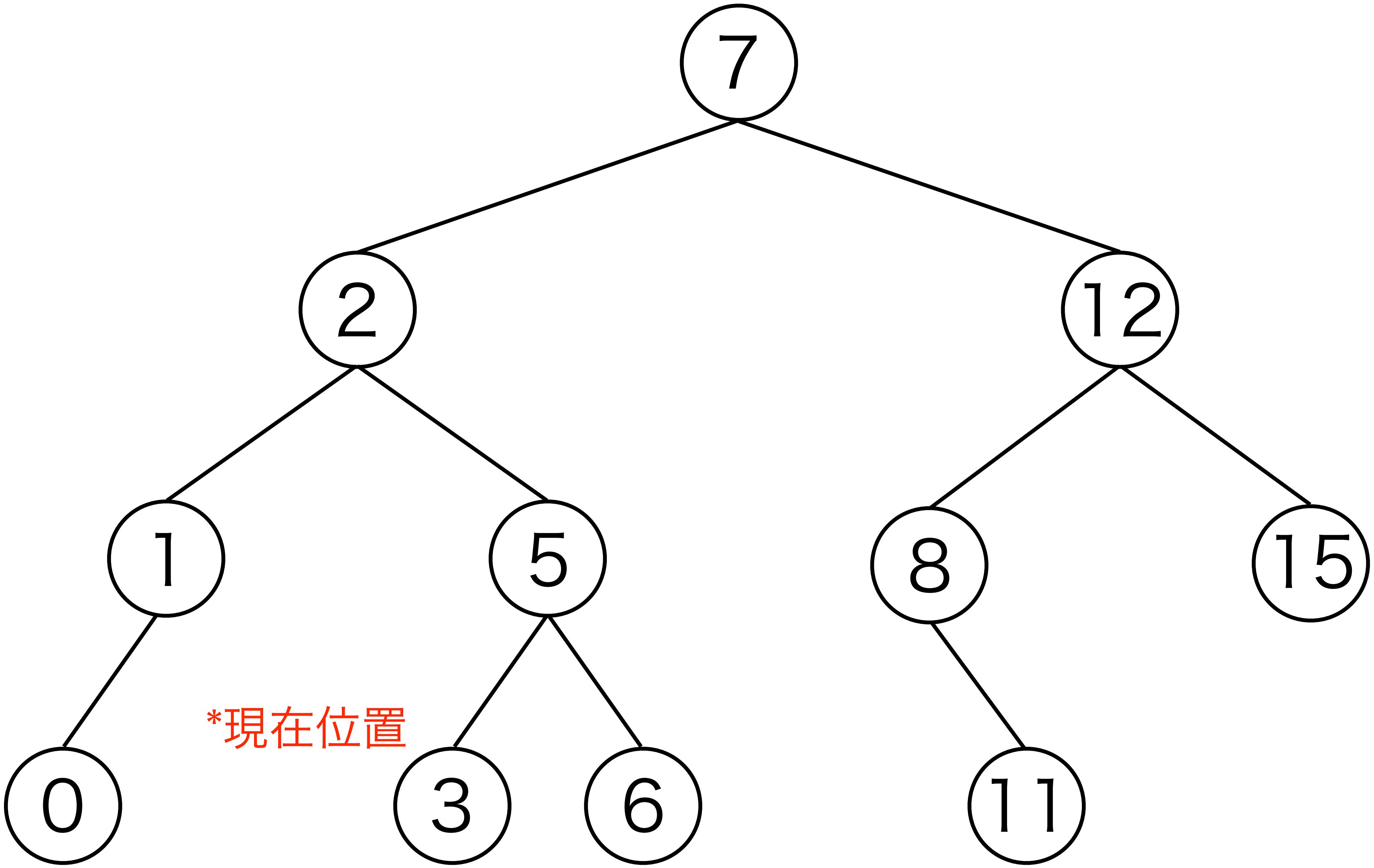
- 現在位置の2と3を比較すると、3の方が大きい。よって、3が含まれているなら右部分木に存在する。そのため、現在位置を右部分木へ移動する。

二分探索木における要素の検索



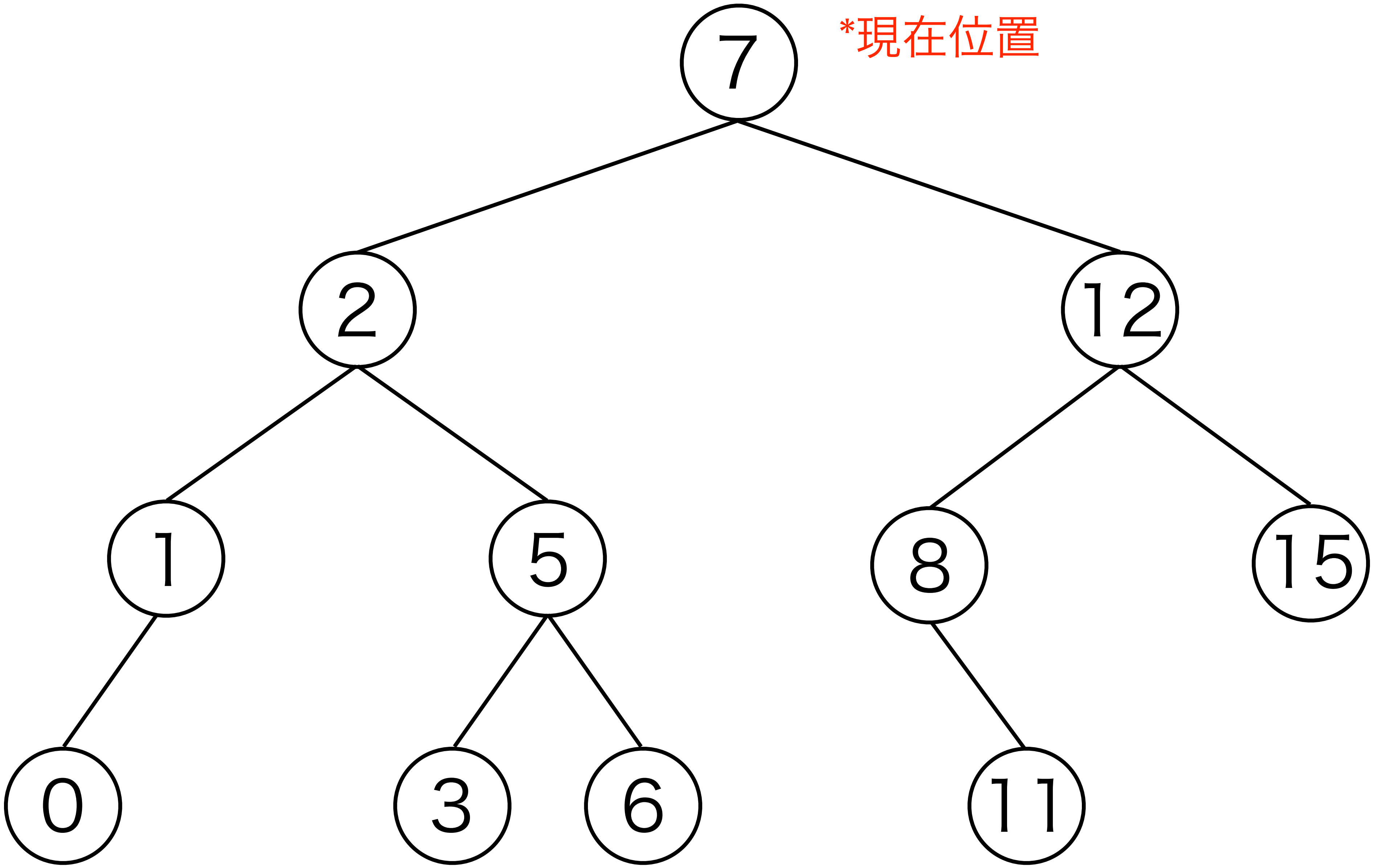
- 現在位置の5と3を比較すると、3の方が大きい。よって、3が含まれているなら左部分木に存在する。そのため、現在位置を左部分木へ移動する。

二分探索木における要素の検索



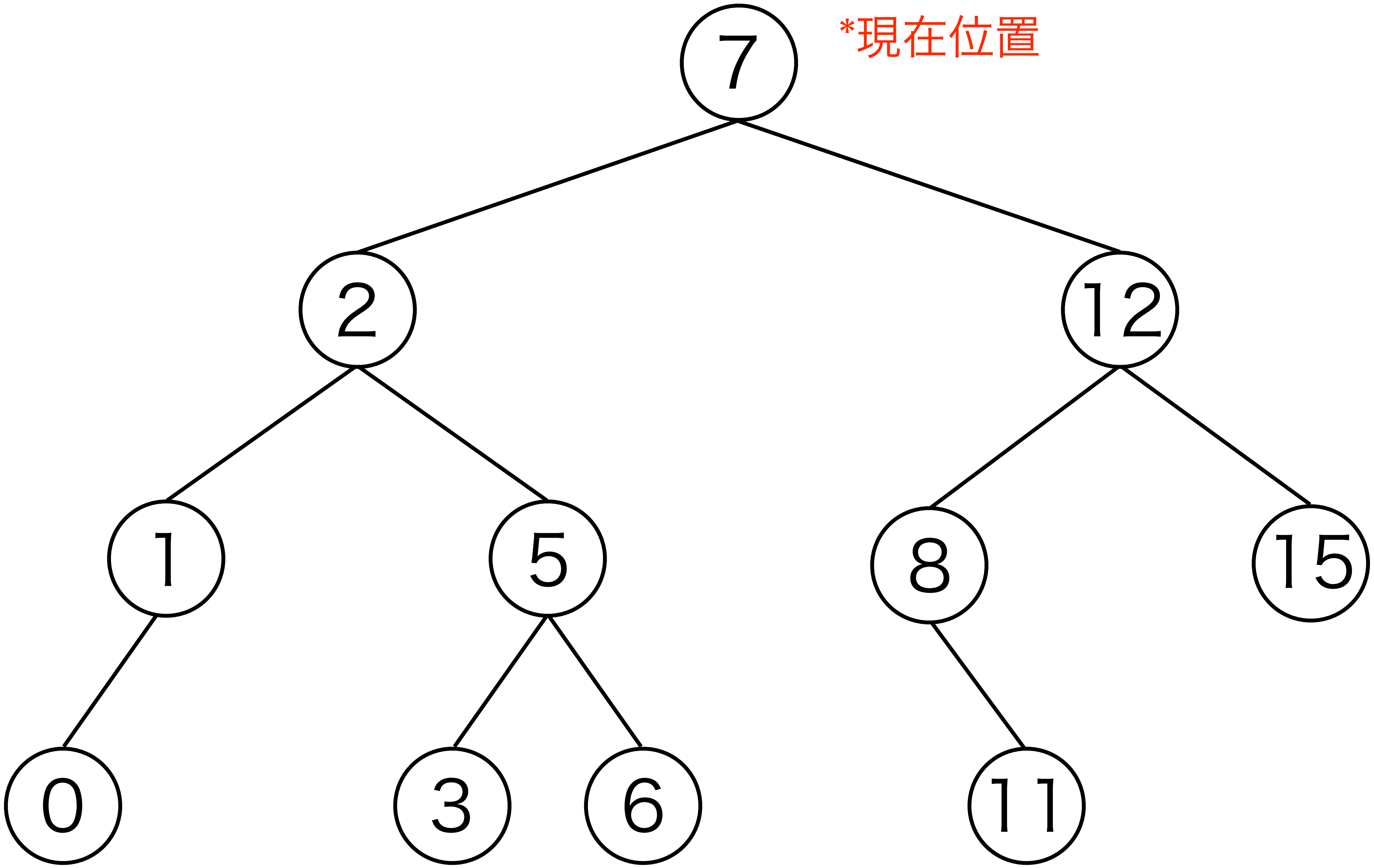
- 3を発見した。(もし葉にたどり着いたにも関わらず発見できなかった場合は、データ構造内にその要素は存在しない。)

二分探索木における要素の挿入



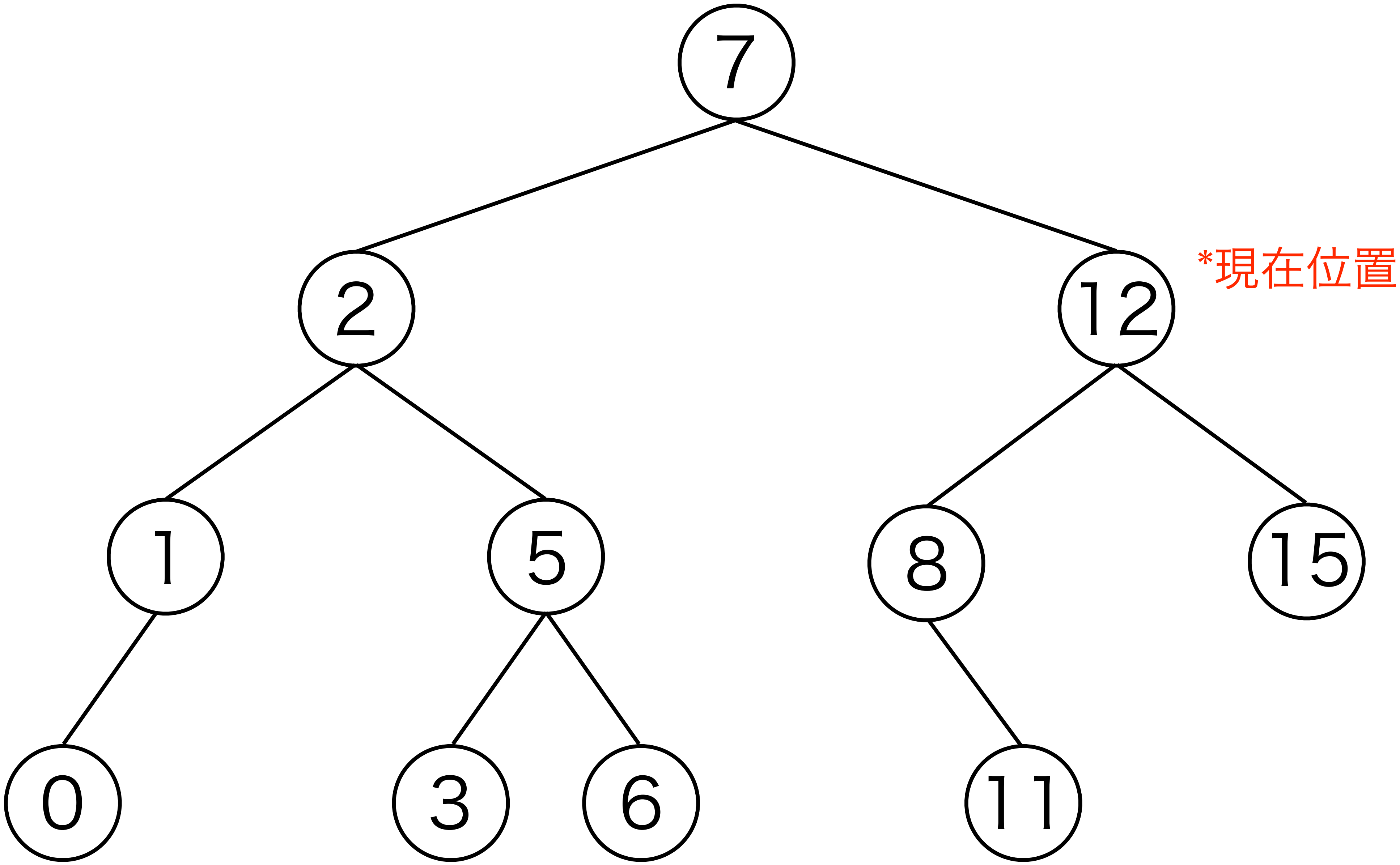
- 二分探索木に13を挿入したいとする。
  まず、根からスタートする。

二分探索木における要素の挿入



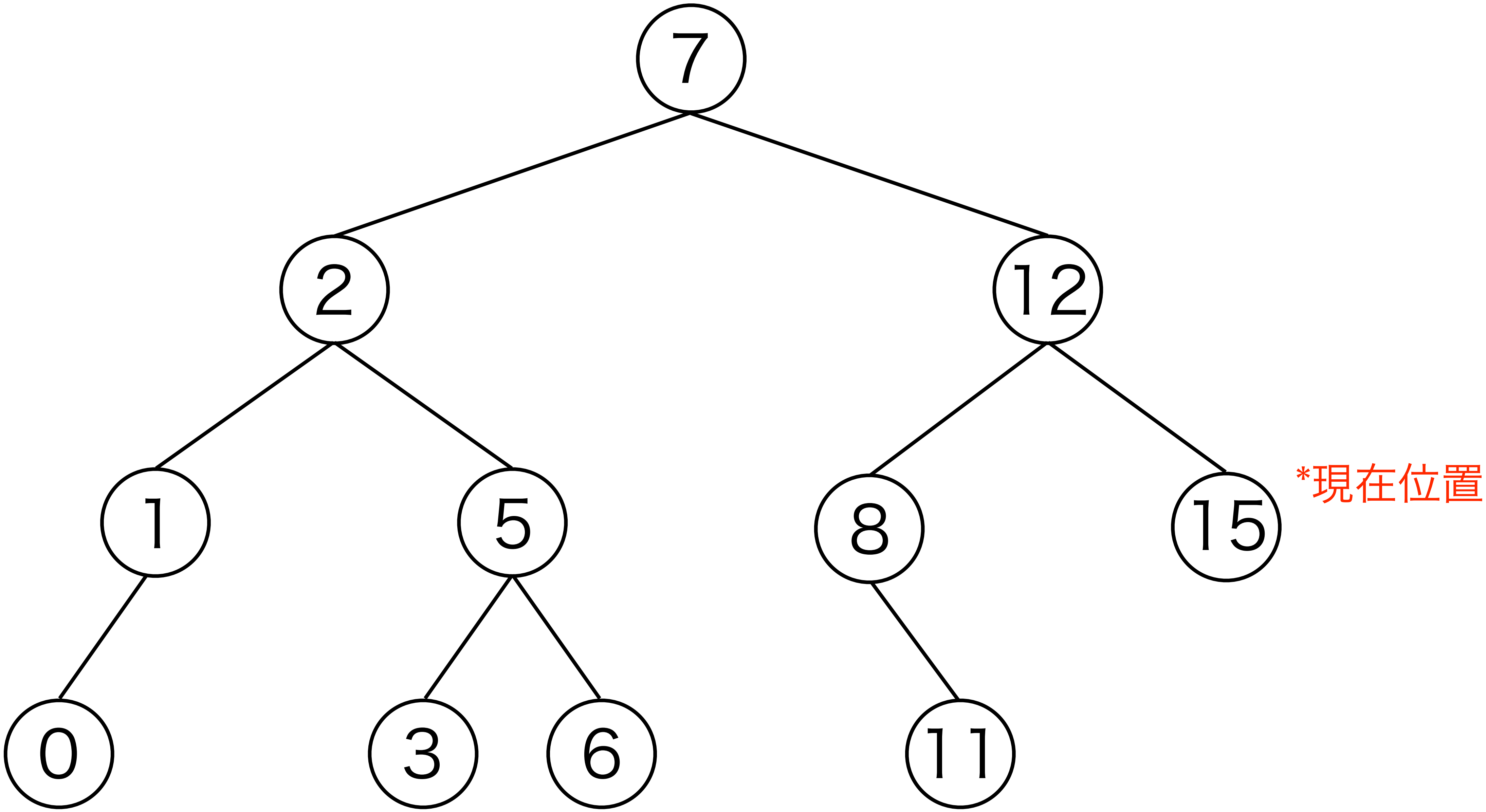
- 現在位置の7と13を比較すると、13の方が大きい。よって、13はその右部分木に挿入しなければならない。よって右部分木へ移動する。

二分探索木における要素の挿入



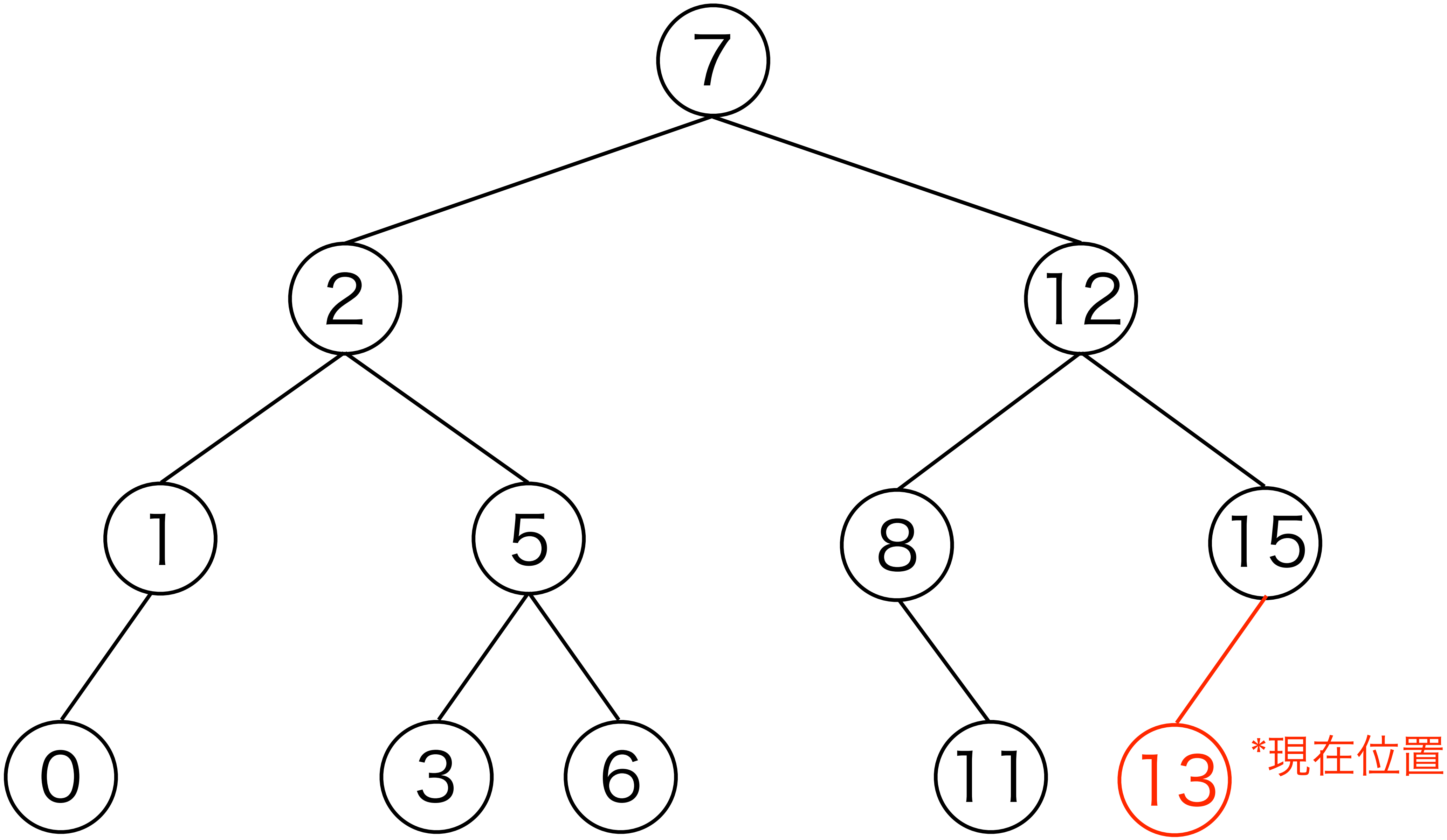
- 現在位置の12と13を比較すると、13の方が大きい。よって、13はその右部分木に挿入しなければならない。よって右部分木へ移動する。

二分探索木における要素の挿入



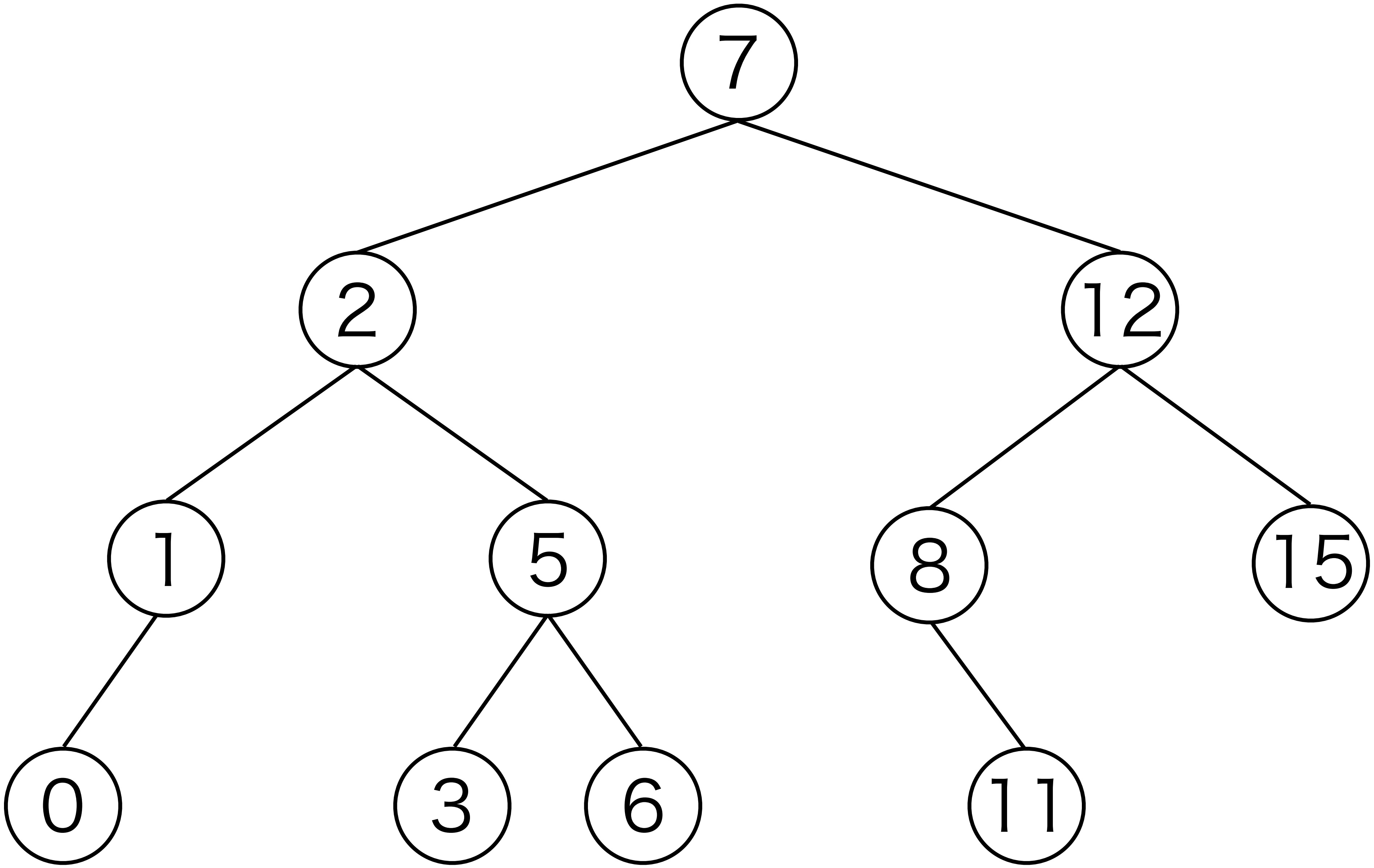
- 現在位置の15と13を比較すると、13の方が大きい。よって、13はその左部分木に挿入しなければならない。よって左部分木へ移動する。

二分探索木における要素の挿入



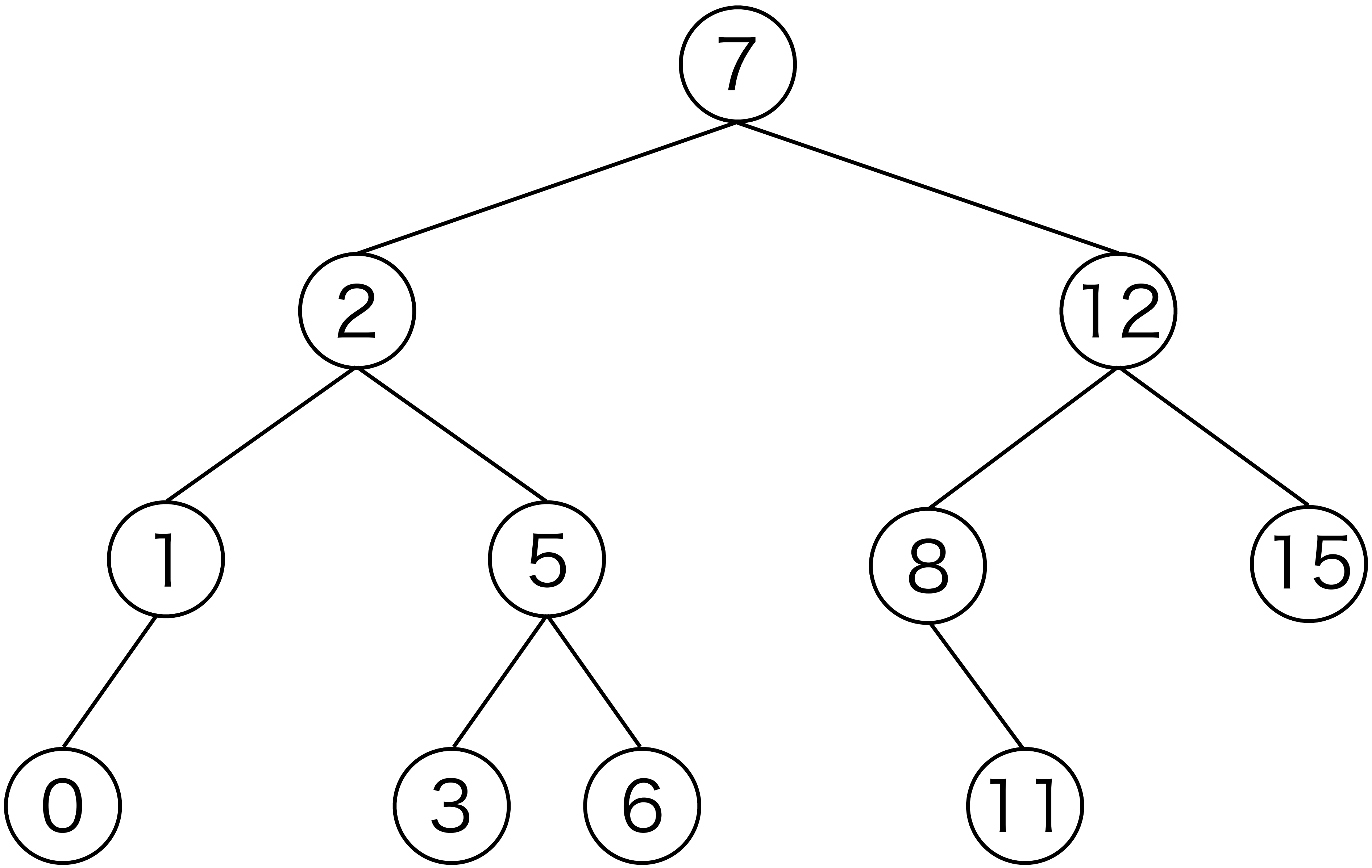
- 15は左に子供がいないので、そこに13を挿入する。

二分探索木における要素の削除



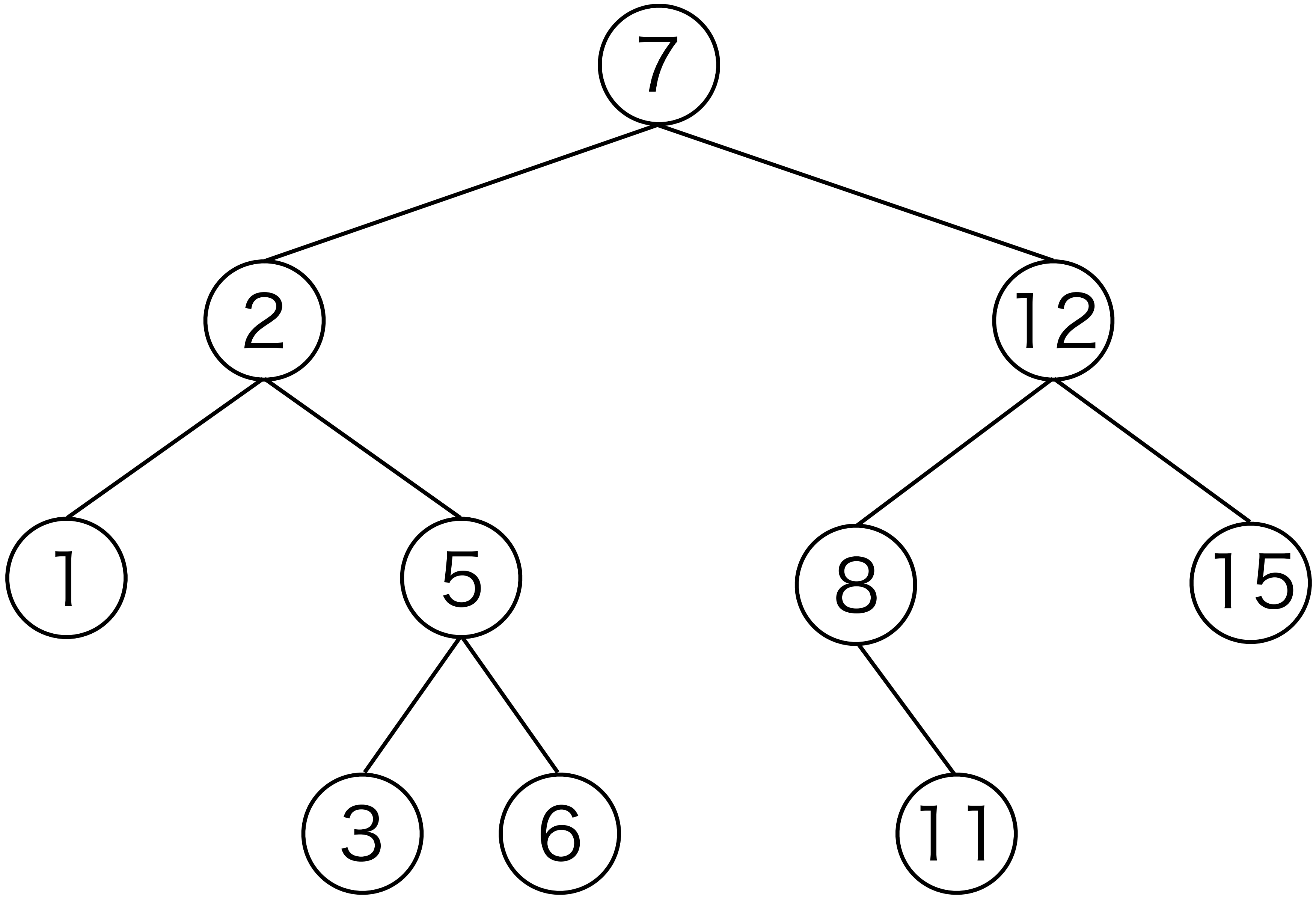
- 二分探索木における要素の削除は次の3つのパターンを考える必要がある。
  1) 削除する要素に子がいない場合
  2) 削除する要素が子を1つ持つ場合
  3) 削除する要素が子を2つ持つ場合

削除する要素に子がいない場合

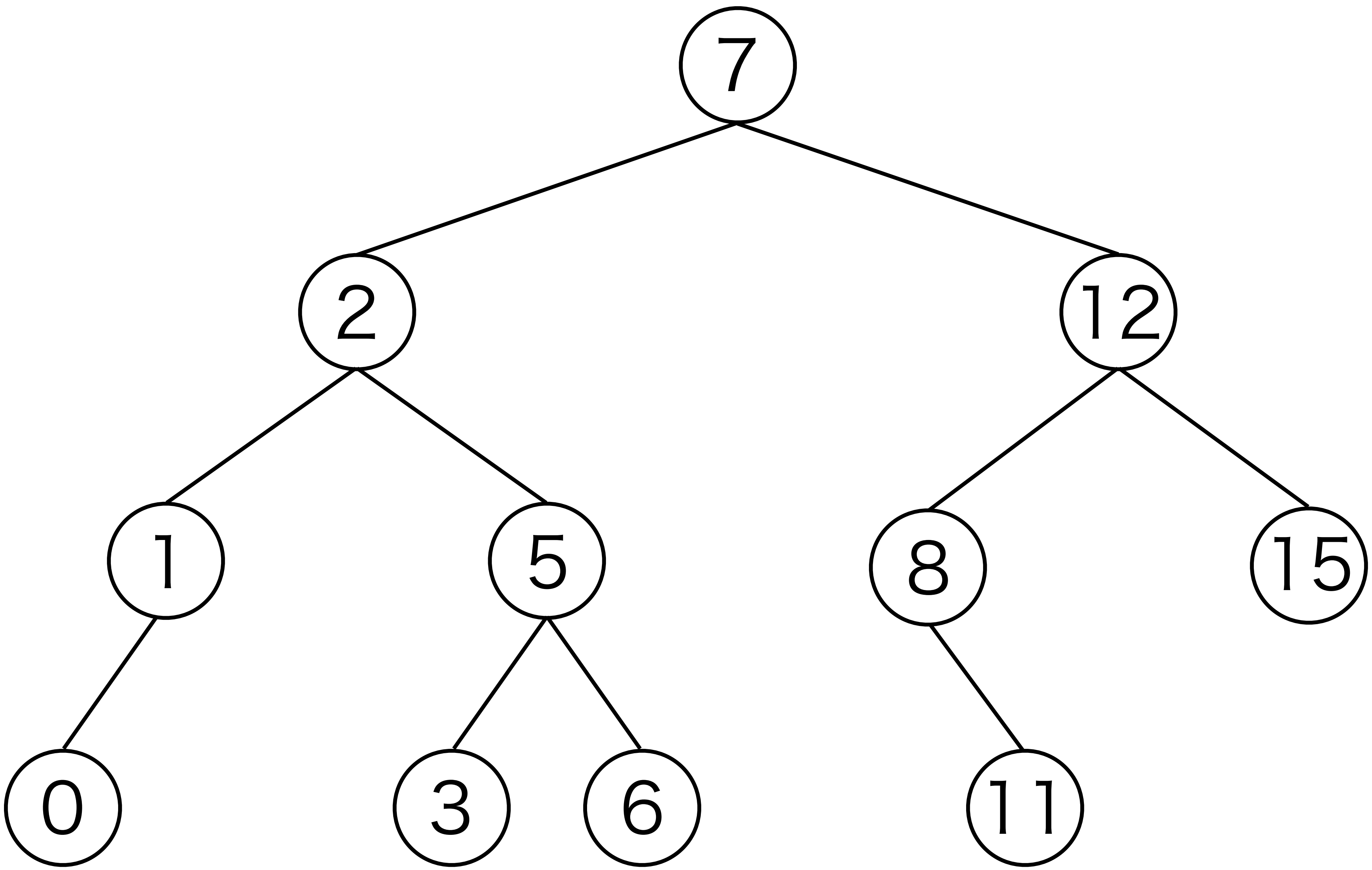


- たとえば0を削除することを考えると、その頂点を取り除く。また親からその頂点への辺を取り除く。

削除する要素に子がいない場合

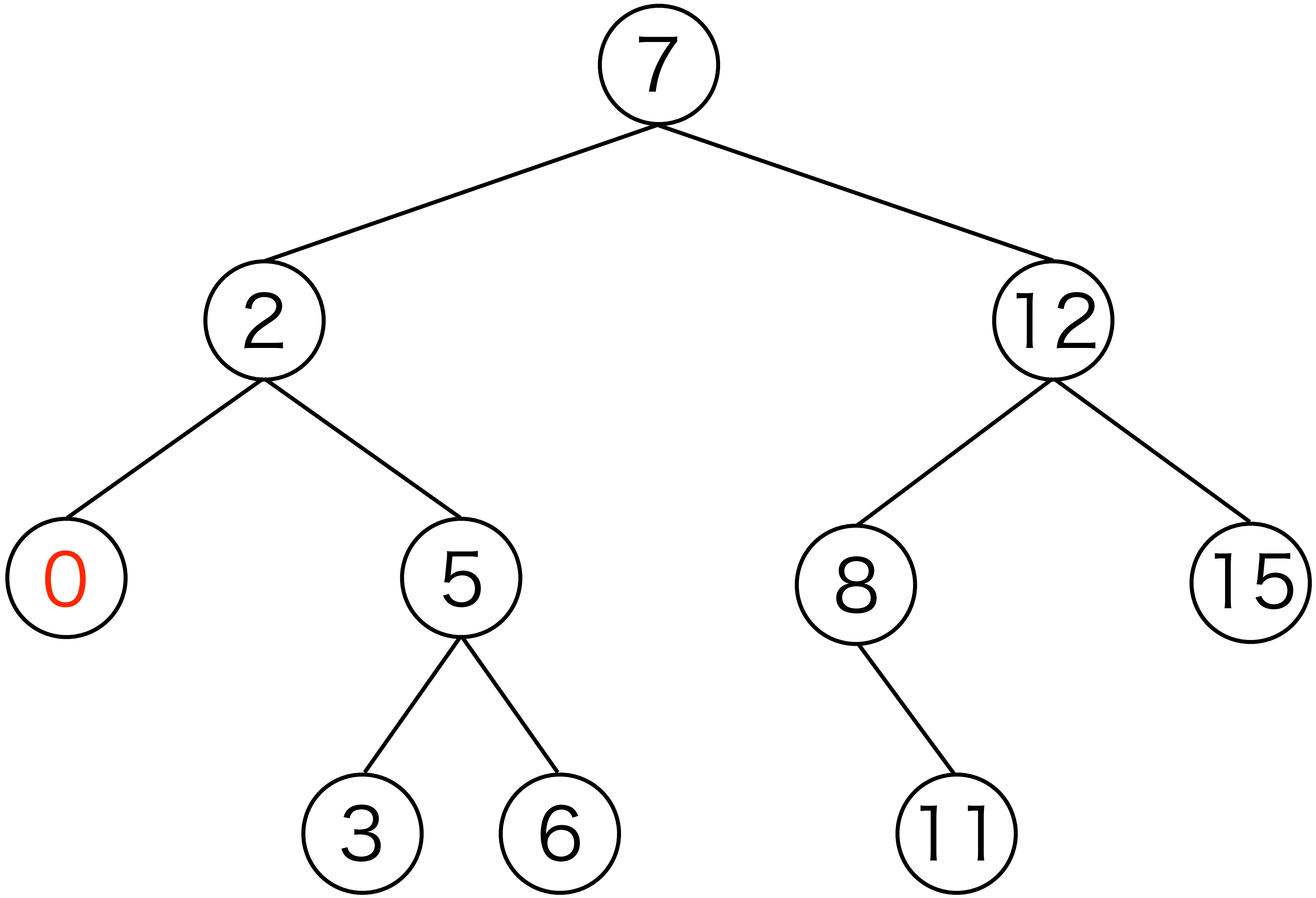


削除する要素が子を1つ持つ場合

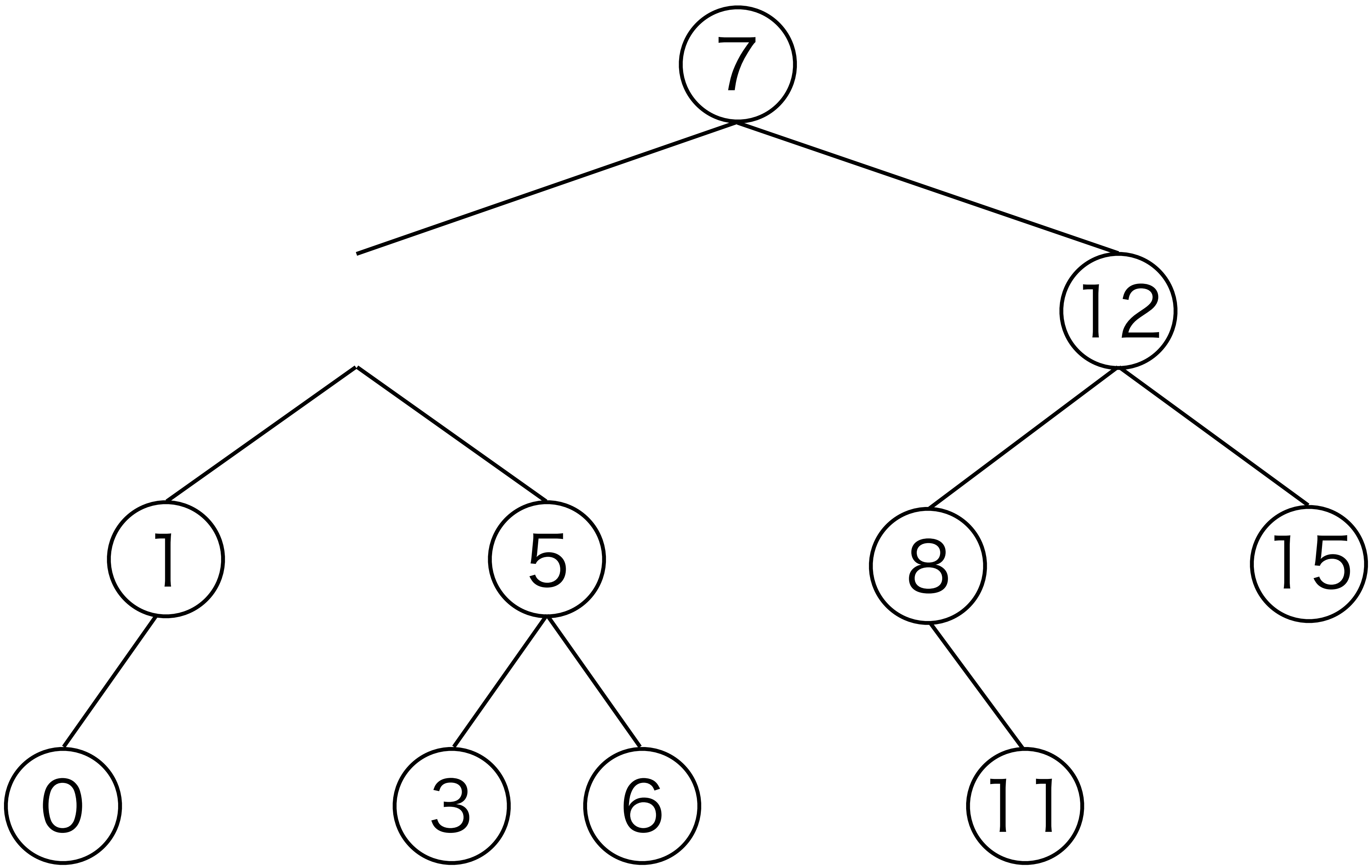


- たとえば1を削除するとする。まず、その頂点を取り除く。
  その後、削除する頂点の子(0)が、削除する頂点の親(2)の子になるように辺を結ぶ。

削除する要素が子を1つ持つ場合

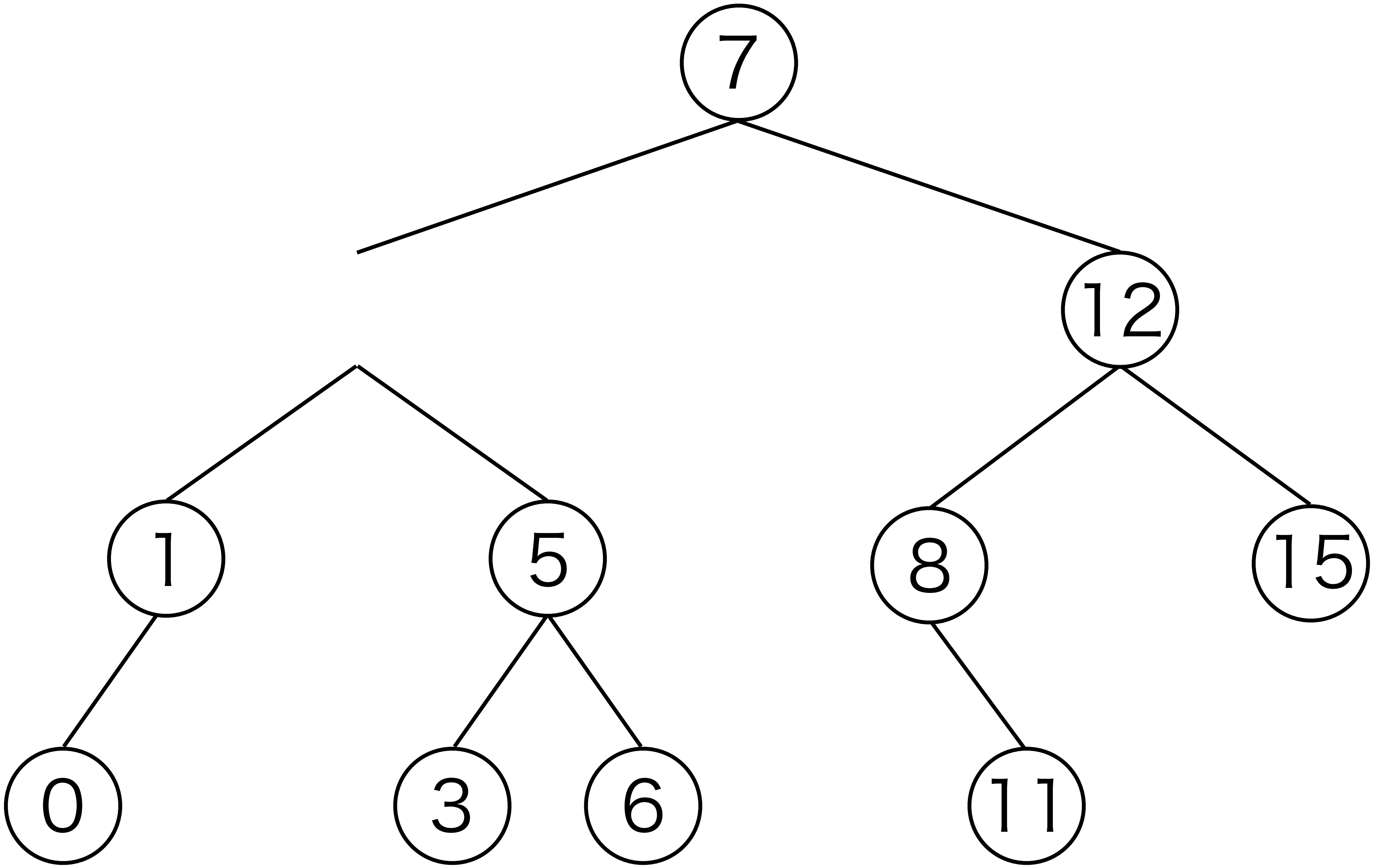


削除する要素が子を2つ持つ場合



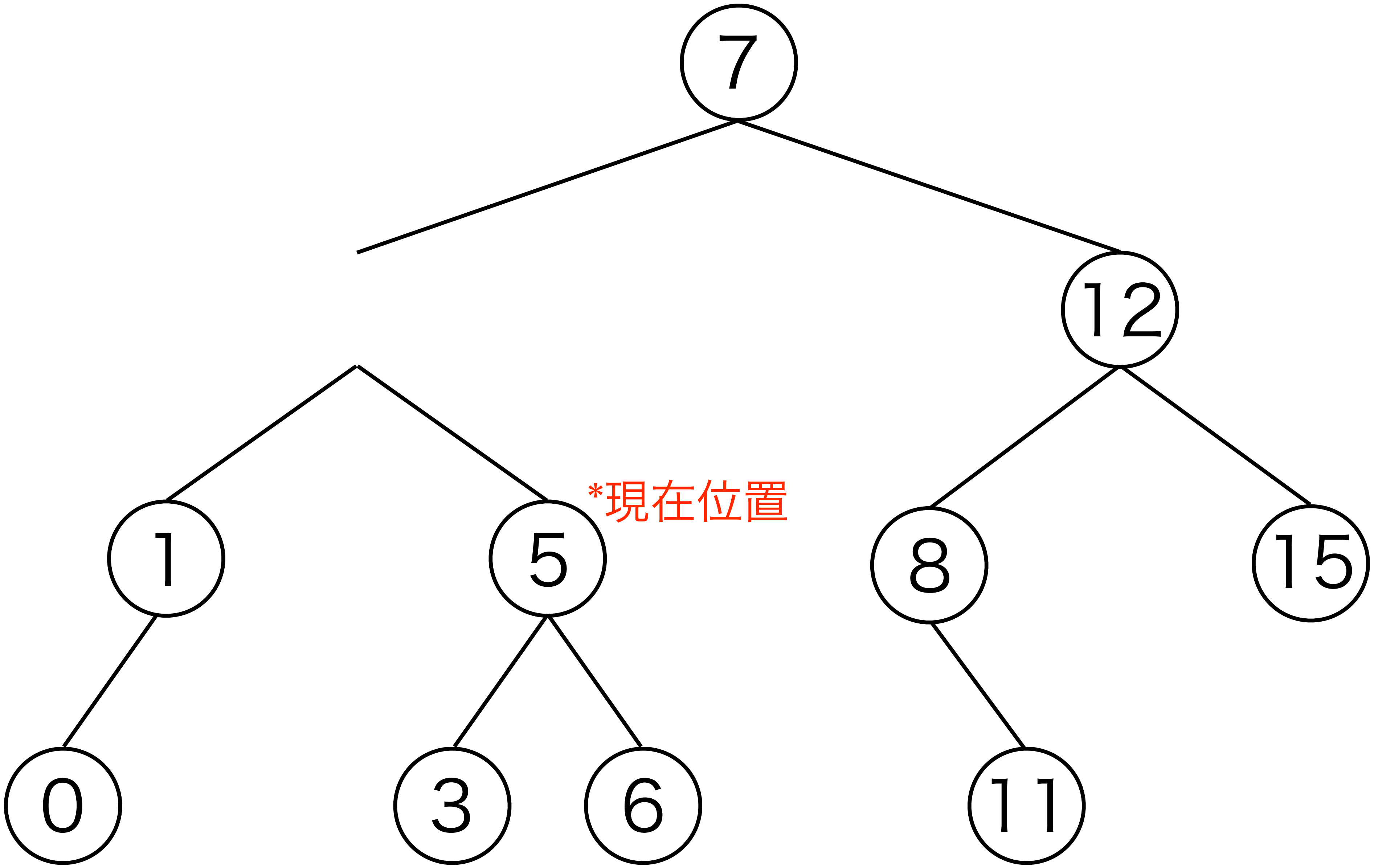
- たとえば2を削除するとする。まず、その頂点を取り除く。
  この時、2が入っていた場所に何を入れれば良いか？

削除する要素が子を2つ持つ場合



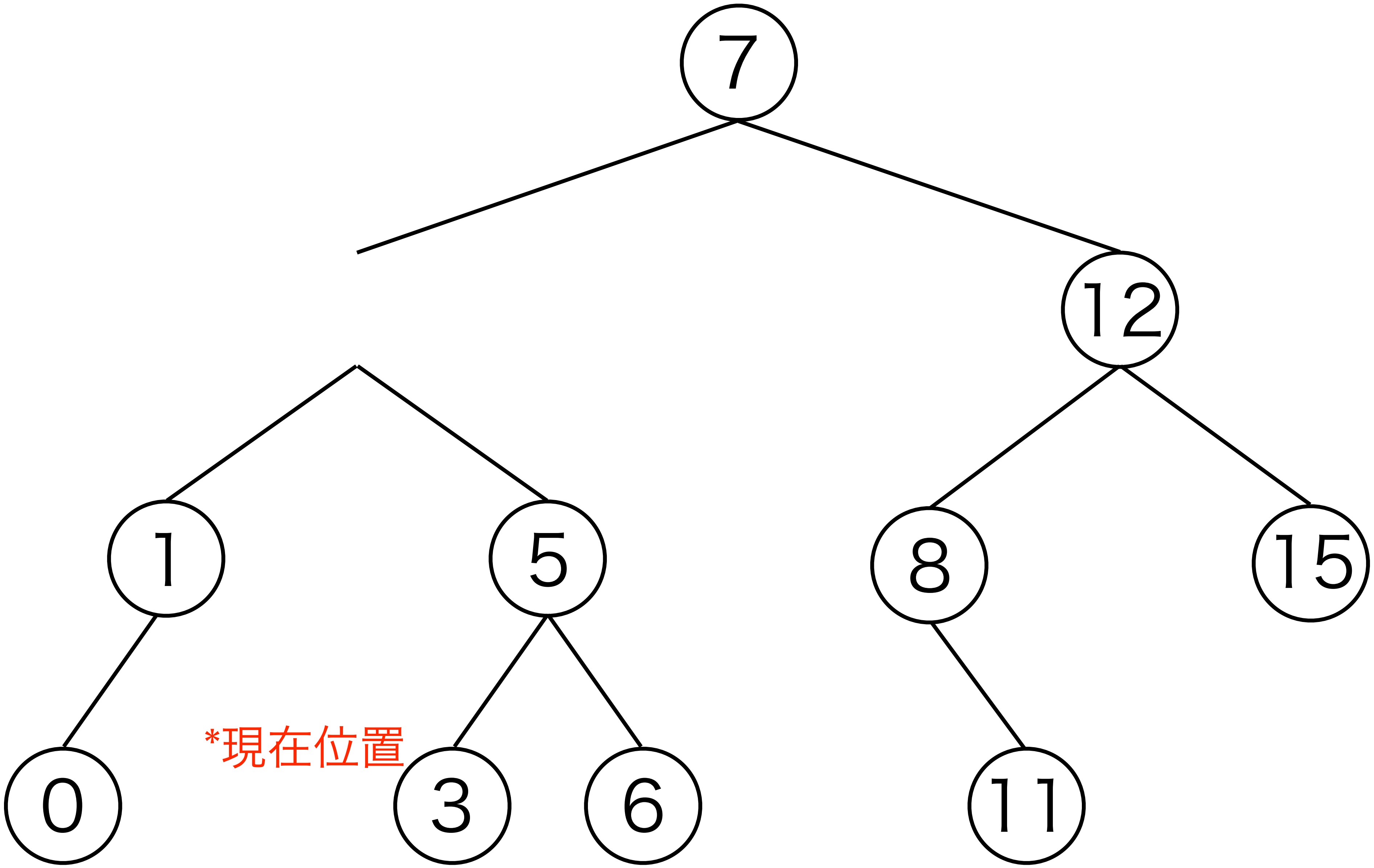
- そこに入る頂点は、全ての左部分木の要素以上であり、また全ての右部分木の要素以下でなければいけない。よって、左部分木の最大値、または右部分木の最小値を持って来れば良い。今回は後者を探す。

削除する要素が子を2つ持つ場合



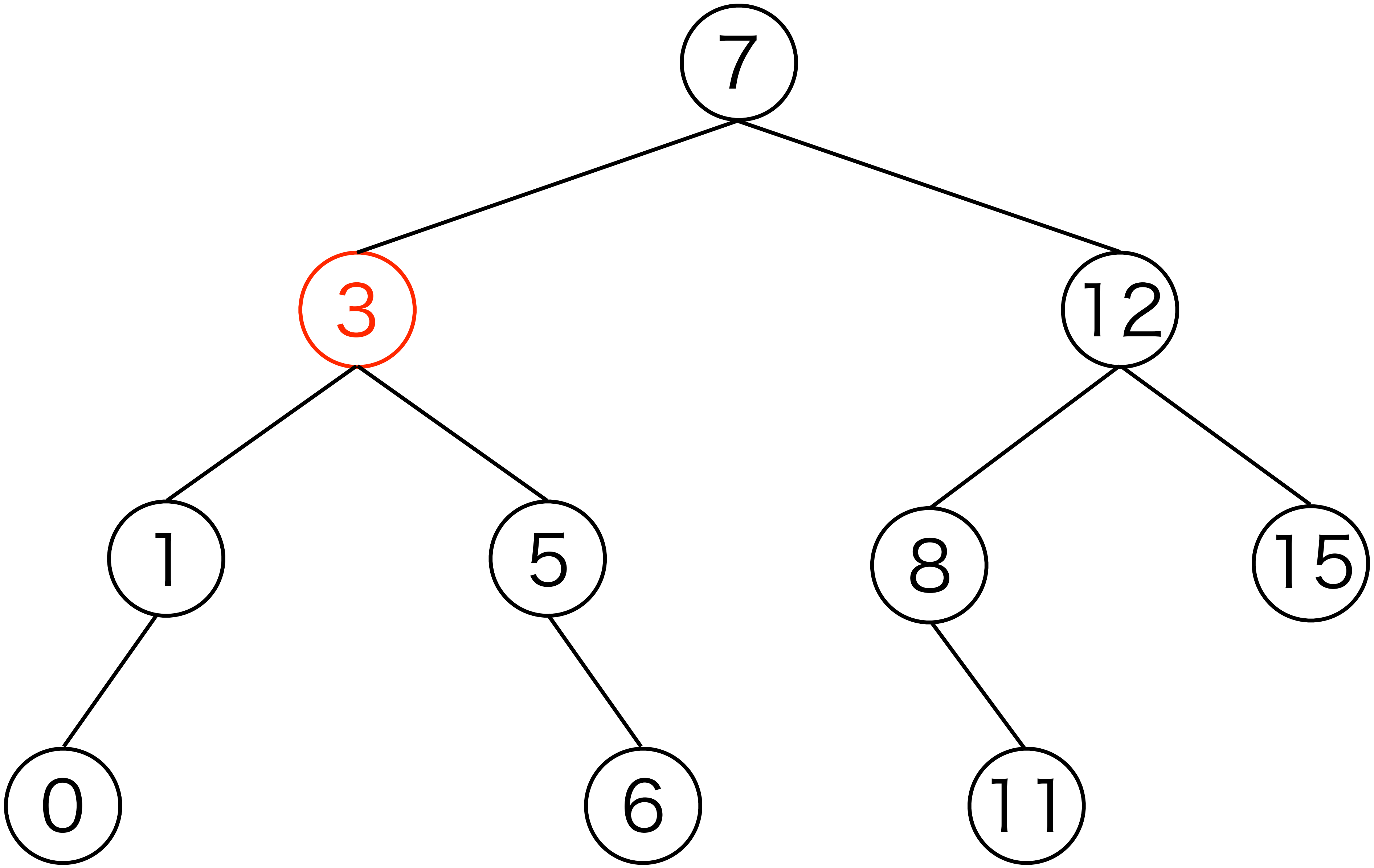
- 右部分木の最小値を探すので、まず一度右に行く。

削除する要素が子を2つ持つ場合



- その後、この部分木の最小値を探すので、左に子がいなくなるまで左に降り続ける。今回は３で止まる。

削除する要素が子を2つ持つ場合



- 発見した最小値を、削除した要素の位置に持っていく。

二分探索木の各操作の計算量

- 二分探索木は、最悪、連結リストのような一本鎖の構造になる。
よって、各種操作の最悪計算量はO(N)となる。

- 一方で二分探索木が平衡である場合は、平衡二分探索木と呼ばれ、
各種操作はO(logN)となる。

- 平衡二分探索木の代表例としては、AVL木、B木、二色木などがあげられる。(一部は講義の第12回目で取り扱う予定)

第十一章
データ構造(4):
Union-Find

Union-Findとは

- グループ分けを管理するデータ構造であり、次の処理を行う事が出来る。

  ・issame(x, y): x, yが同じグループに属するかどうかを調べる
  ・unite(x, y): xが属するグループと、yが属するグループを併合する。

- 右の例に対しては、

  issame(0, 4) = true
  issame(3, 5) = true
  issame(2, 6) = false

図11.1左

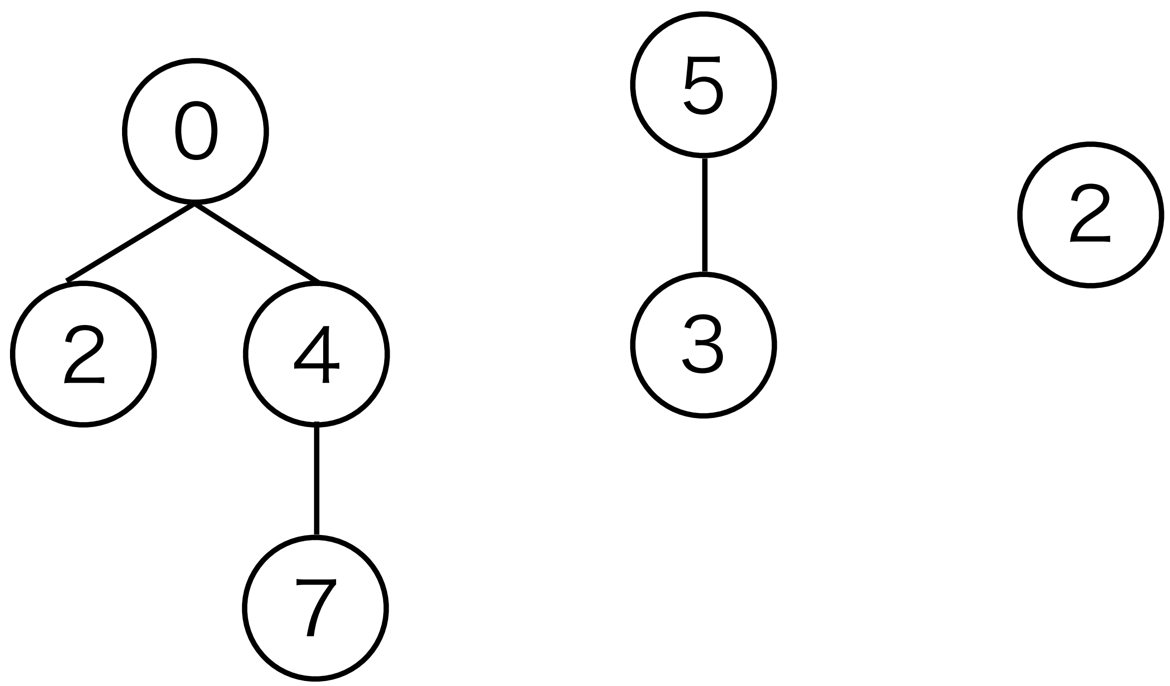
Union-Findとは

図11.1

- Union-Findでは、グループを分割するような操作を行うことはできない。

Union-Find

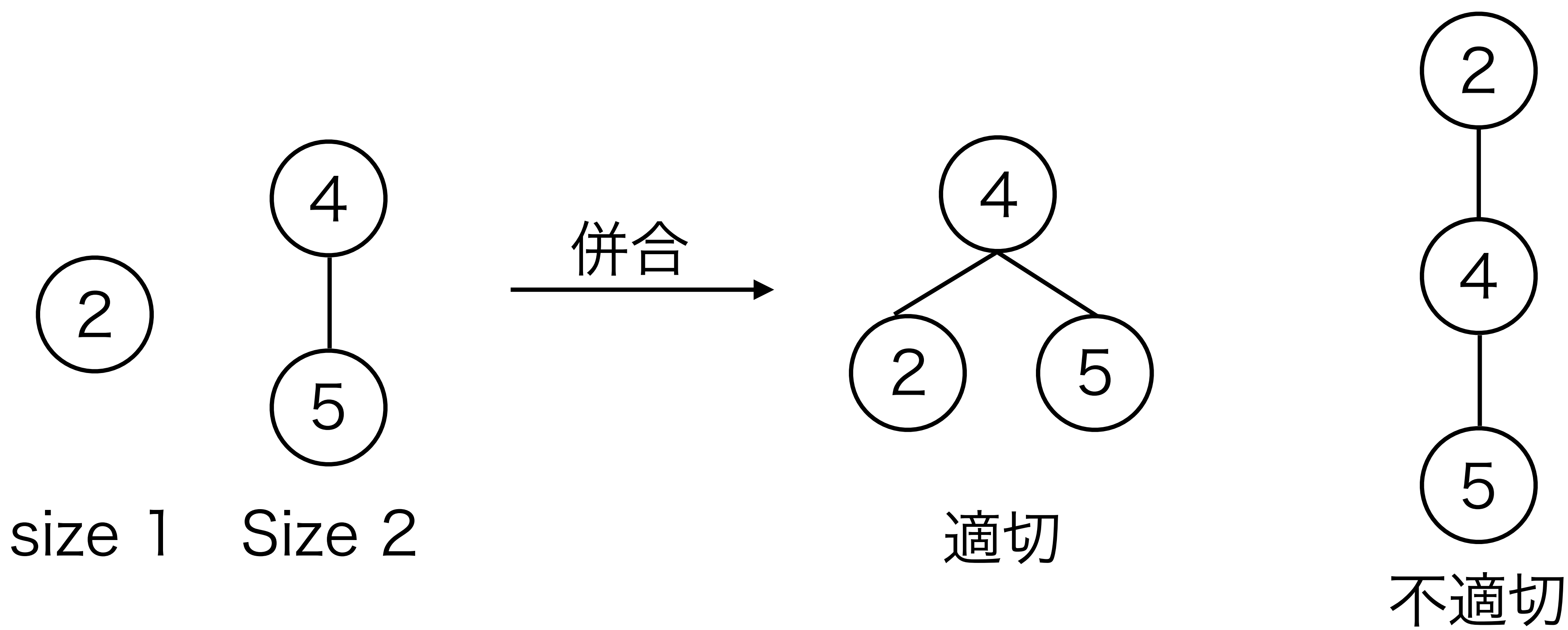
- Union-Findでは、1つのグループが1つの木で表現される。
  なので、グループの集合は森となる。



- 同一グループに属する様子が1つの木としてまとまっていれば、
  木の形や親子の関係は何でも良い。

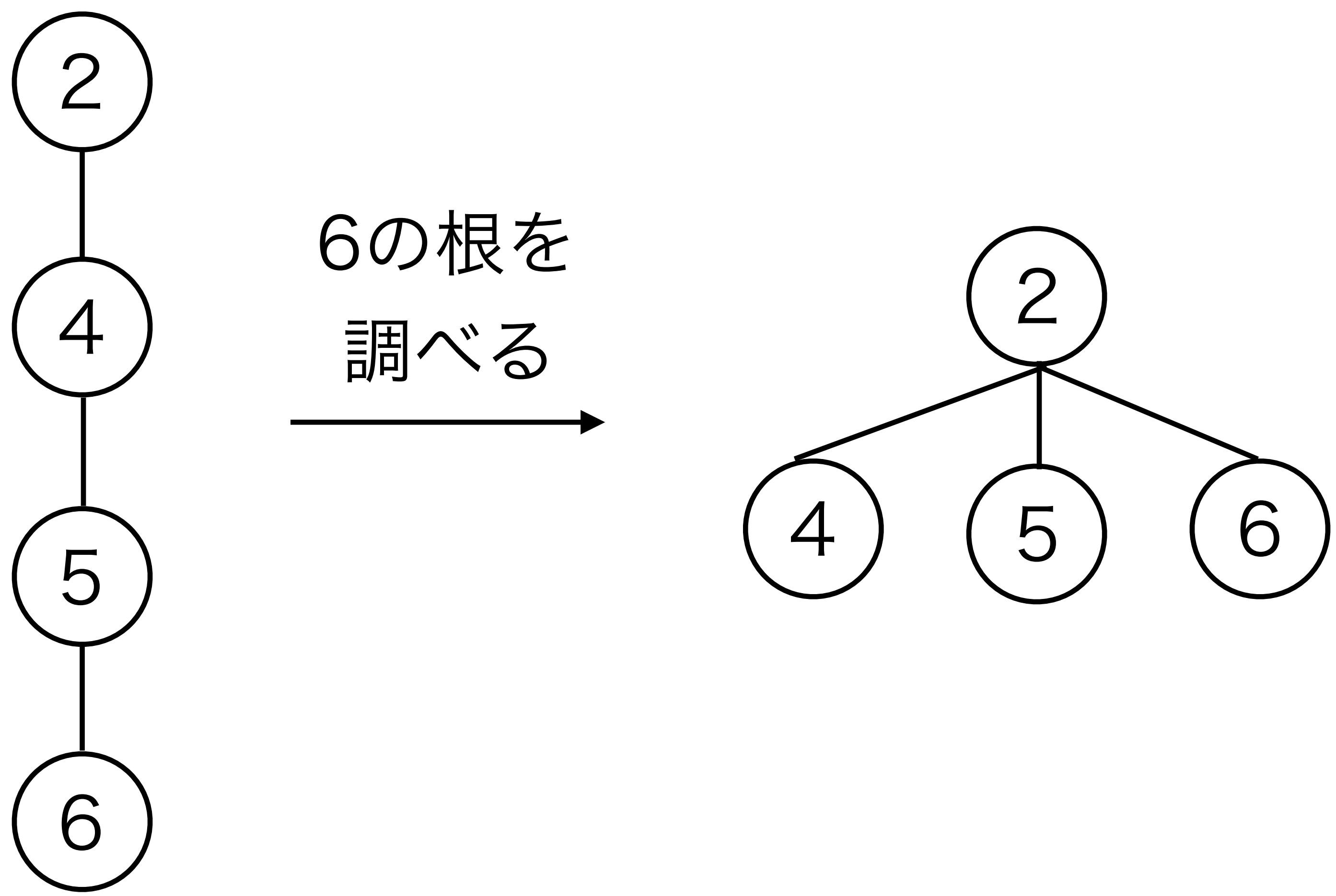
Union-Findの工夫その1 (union by size)

- 各木について、木の持つ頂点数(サイズ)を記憶しておき、
  併合する時にはサイズの小さい方の根が子頂点になるようにする。



Union-Findの工夫その2 (経路圧縮)

- ある頂点の根を調べる時、根までに辿ったNodeの親を全て根に起き変える。



Union-Findの操作の平均計算量

- 2つの工夫を施したUnion-Findの操作の平均計算量(ならし計算量)は、$O(\alpha(n))$である事が知られており、非常に高速である。($O(\log n)$よりはるかに早い。) ここで$\alpha(n)$は、Ack(n, n)の逆関数。

アッカーマン関数

$$Ack(m,n) =$$

$$\begin{cases} n+1 & \text{if } m=0 \\ Ack(m-1,1) & \text{if } n=0 \\ Ack(m-1, Ack(m,n-1)) & \text{otherwise} \end{cases}$$

アッカーマン関数の値の表

m\n	0	1	2	3	4	5
0	1	2	3	4	5	6
1	2	3	4	5	6	7
2	3	5	7	9	11	13
3	5	13	29	61	125	253
4	13	65533	$2^{65533}-3$	$2^{(2^{65533})}-3$	・・・	・・・
5	65533	・・・	・・・	・・・	・・・	・・・

Union-Findの実装

```
// Union-Find
struct UnionFind {
    vector<int> par, siz;

    // 初期化
    UnionFind(int n) : par(n, -1) , siz(n, 1) { }

    // 根を求める
    int root(int x) {
        if (par[x] == -1) return x; // x が根の場合は x を返す
        else return par[x] = root(par[x]);
    }

    // x と y が同じグループに属するかどうか (根が一致するかどうか)
    bool issame(int x, int y) {
        return root(x) == root(y);
    }
```

Union-Findの実装

```
bool unite(int x, int y) {
    // x, y をそれぞれ根まで移動する
    x = root(x);
    y = root(y);

    // すでに同じグループのときは何もしない
    if (x == y) return false;

    // union by size (y 側のサイズが小さくなるようにする)
    if (siz[x] < siz[y]) swap(x, y);

    // y を x の子とする
    par[y] = x;
    siz[x] += siz[y];
    return true;
}

// x を含むグループのサイズ
int size(int x) {
    return siz[root(x)];
}
```

まとめ

- 木のデータ構造として、ヒープ／二分探索木／Union-Findを紹介した。

- ヒープは最大値(最小値)を求めるのに適したデータ構造であり、
またヒープを活用してソートをO(NlogN)で行う事が出来る。

- 二分探索木は要素の挿入・削除・検索ができるデータ構造であり、
木が平衡である場合はいずれもO(logN)で操作が可能である。

- Union-Findはグループ分けを管理するデータ構造であり、2つの要素が
同一のグループであるか、及び2つのグループの併合の操作を、
極めて高速に行う事が出来る。