



二分木ヒープ (**BINARY HEAP**)

2016年度

担当: 只木進一(工学系研究科)

二分木ヒープとは

- 集合・リストから「最小な」要素を取り出す
 - 二分木ヒープは、そのための標準的データ構造
- 二分木ヒープを保存するデータ構造
- 二分木ヒープの操作のメソッド
- 対象となるデータクラス
 - 識別のためのlabelフィールド
 - 値を保持するvalueフィールド

二分木ヒープとは、どういう二分木か

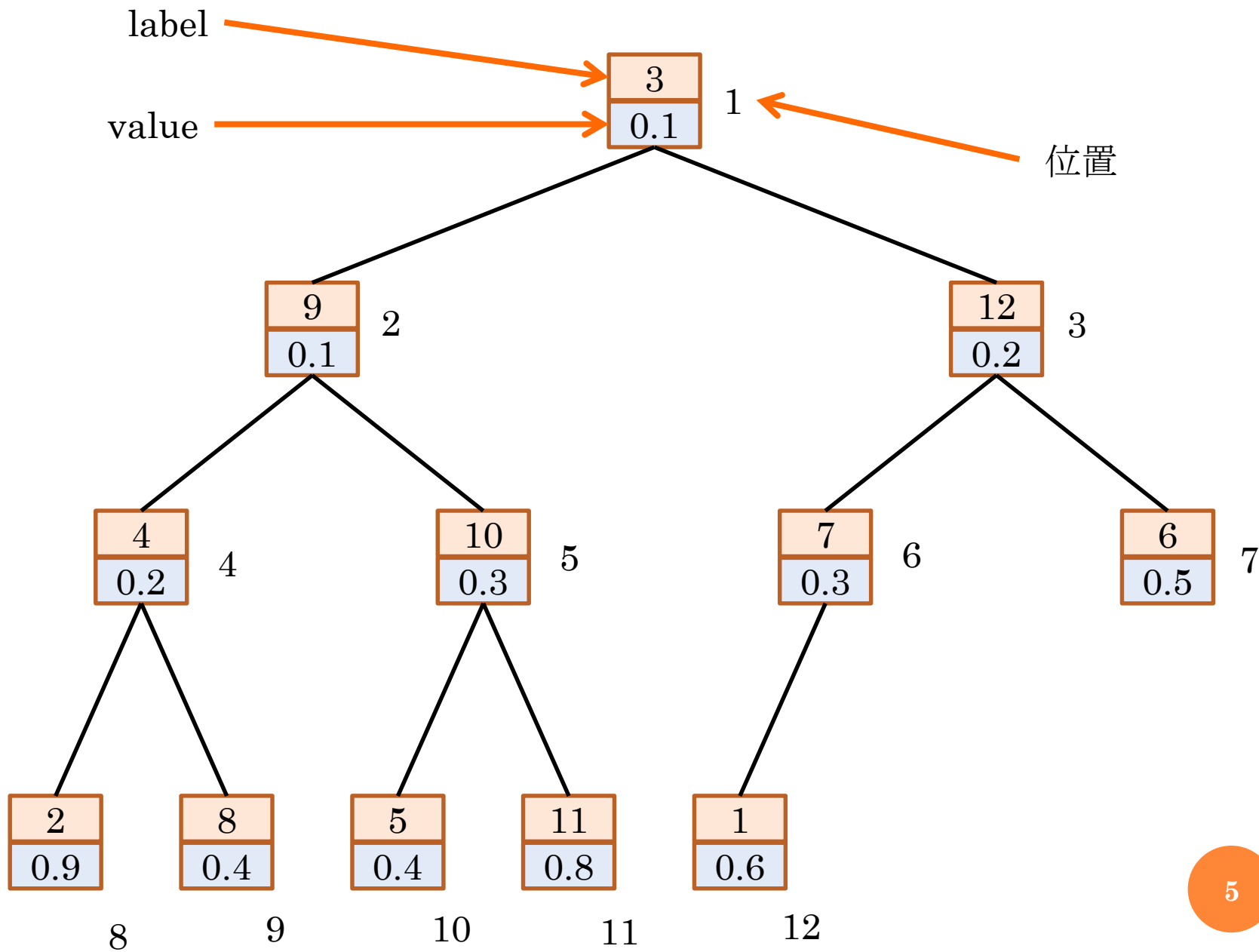
- ある頂点の要素 p のvalueは、その子 c の要素のvalueより大きくない

$$p.value \leq c.value$$

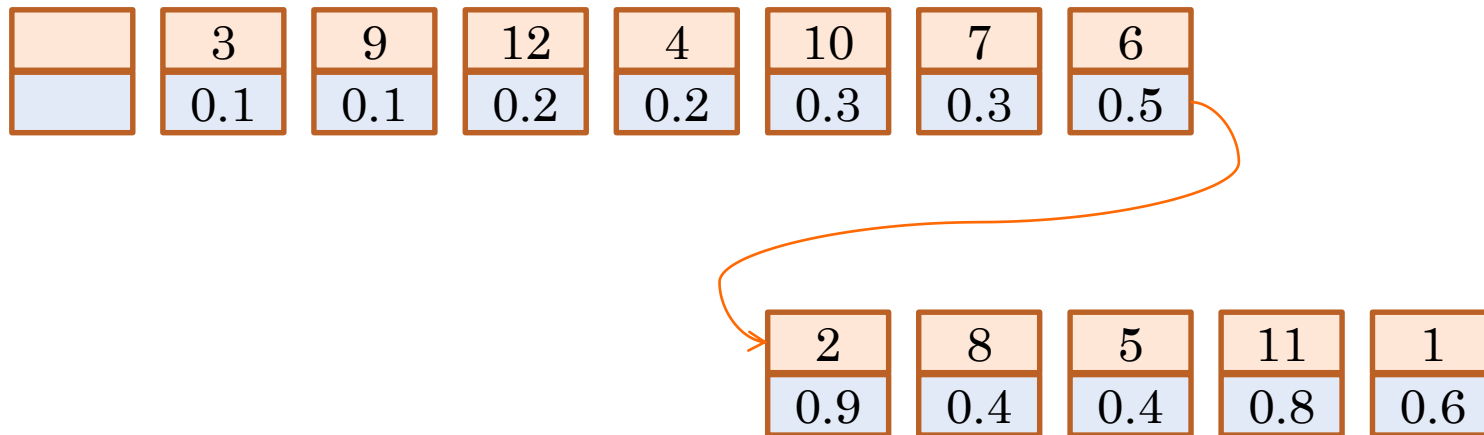
- 半順序木になっている
- 完全二分木である
 - 最下層以外の第 k 層には、 2^{k-1} 個の頂点がある。
 - 最下層は、左から詰めて頂点がある。

半順序集合(PARTIALLY ORDERED SET)または順序集合(ORDERED SET)

- 集合 P とその上の関係 \leq に以下の法則が常になりたつとき、 P を半順序集合と呼ぶ。ここで、 $a, b, c \in P$ となる任意の要素とする。
 - 反射律 (reflectivity): $a \leq a$
 - 推移律 (transitivity):
$$(a \leq b) \wedge (b \leq c) \Rightarrow a \leq c$$
 - 反対照律 (antisymmetry)
$$(a \leq b) \wedge (b \leq a) \Rightarrow a = b$$
- 任意の元の組 $a, b \in P$ に対して $a \leq b$ または $b \leq a$ の何れかが必ず成り立つとき、 P を全順序集合(totally ordered set)と呼ぶ。



データの保持形式

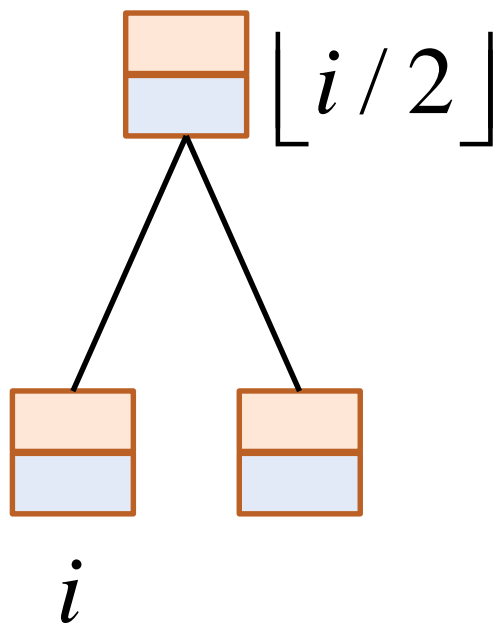


○ リストで保持

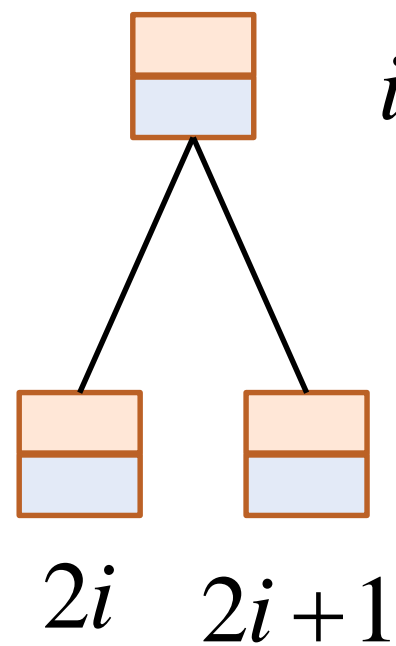
- 0番: 空 (リストのインデックスが1から始まるプログラミング言語では不要)
- 1番: 根の要素
- 2^k 番: 第 k 層の左端の要素

親子の番号の関係

○ 親の番号



○ 子の番号



要素の追加

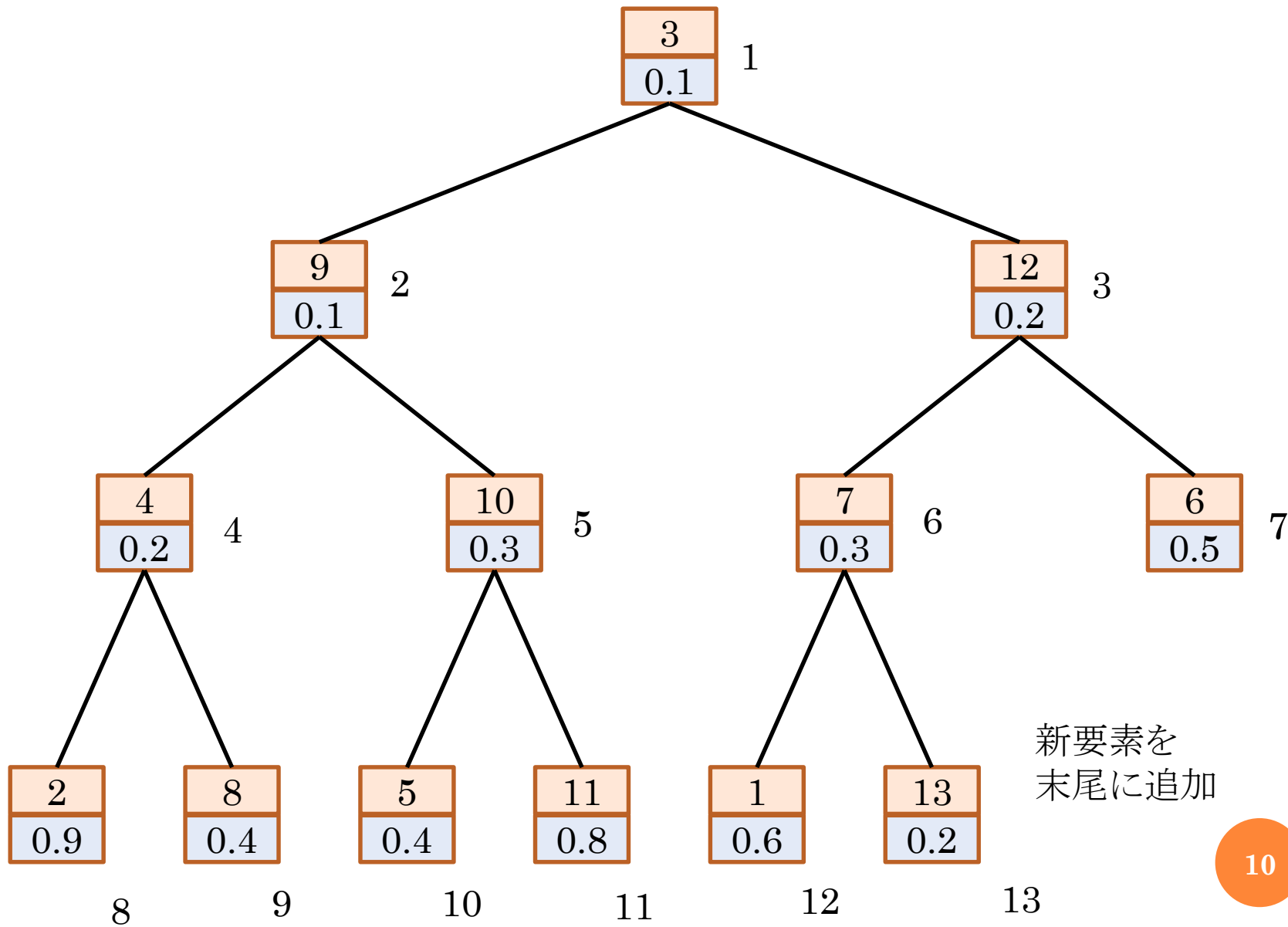
- リストの終端に要素を追加する
 - 木の最下層の一番右に追加、または新たな層を作って、その左端に追加

```
void add(O o){  
    int n = |L|;  
    L.append(o);  
    n++;  
    shiftUp(n); //要素を正しい位置へ移動  
}
```

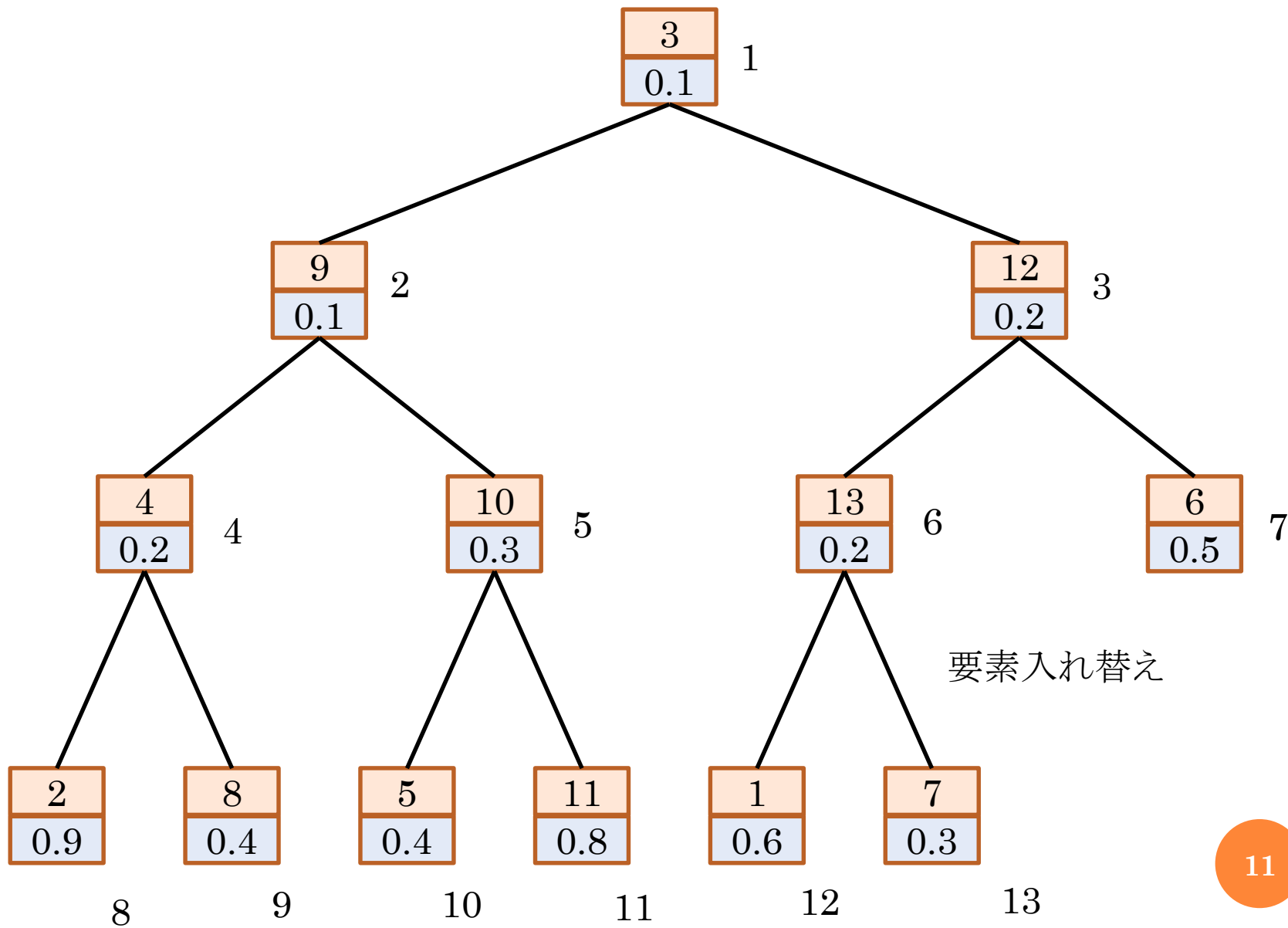

要素の追加:シフトアップ

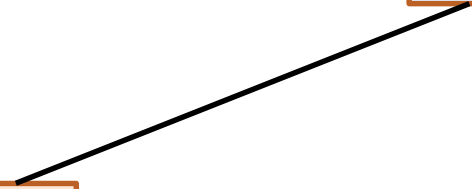
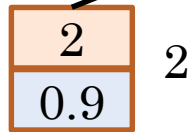
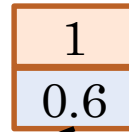
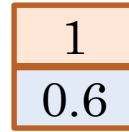
- 追加した新要素を正しい位置へ移動
- 位置 k の要素が、親の位置 $\lfloor k/2 \rfloor$ の要素よりも小さいならば、二つの要素を入れ替える
- $\text{isLess}(i, j)$
 - $o_i.value < o_j.value$ のとき真
- $\text{swap}(i, j)$: 要素入れ替え

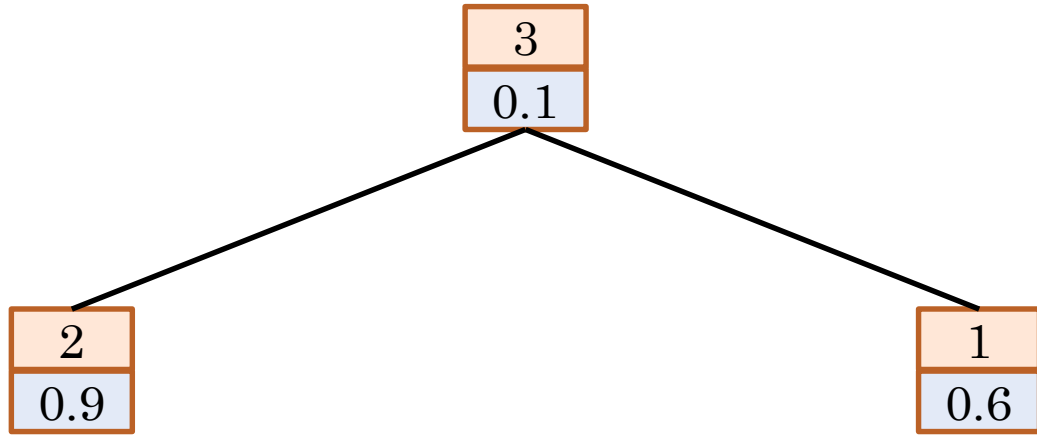
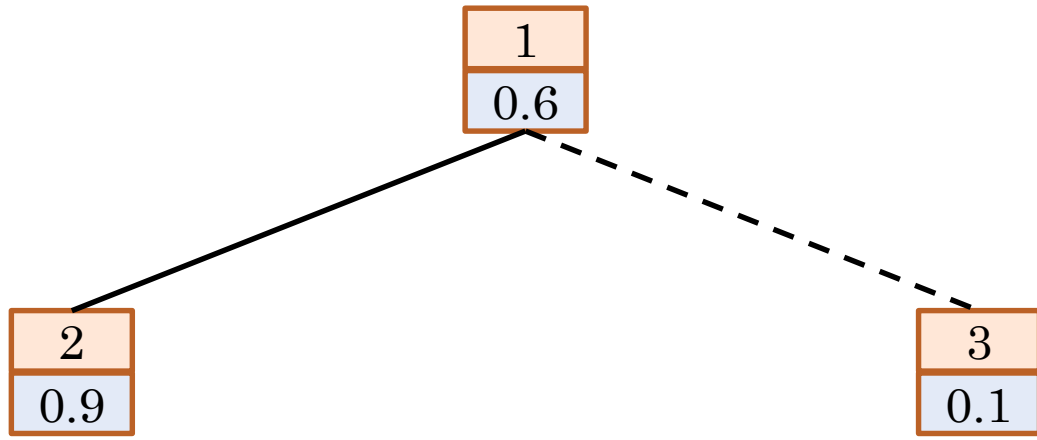
```
void shiftUp(int k){  
    if(k > 1 && isLess(k,  $\lfloor k/2 \rfloor$ )){  
        swap(k,  $\lfloor k/2 \rfloor$ );  
        k =  $\lfloor k/2 \rfloor$ ;  
        shiftUp(k);  
    }  
}
```

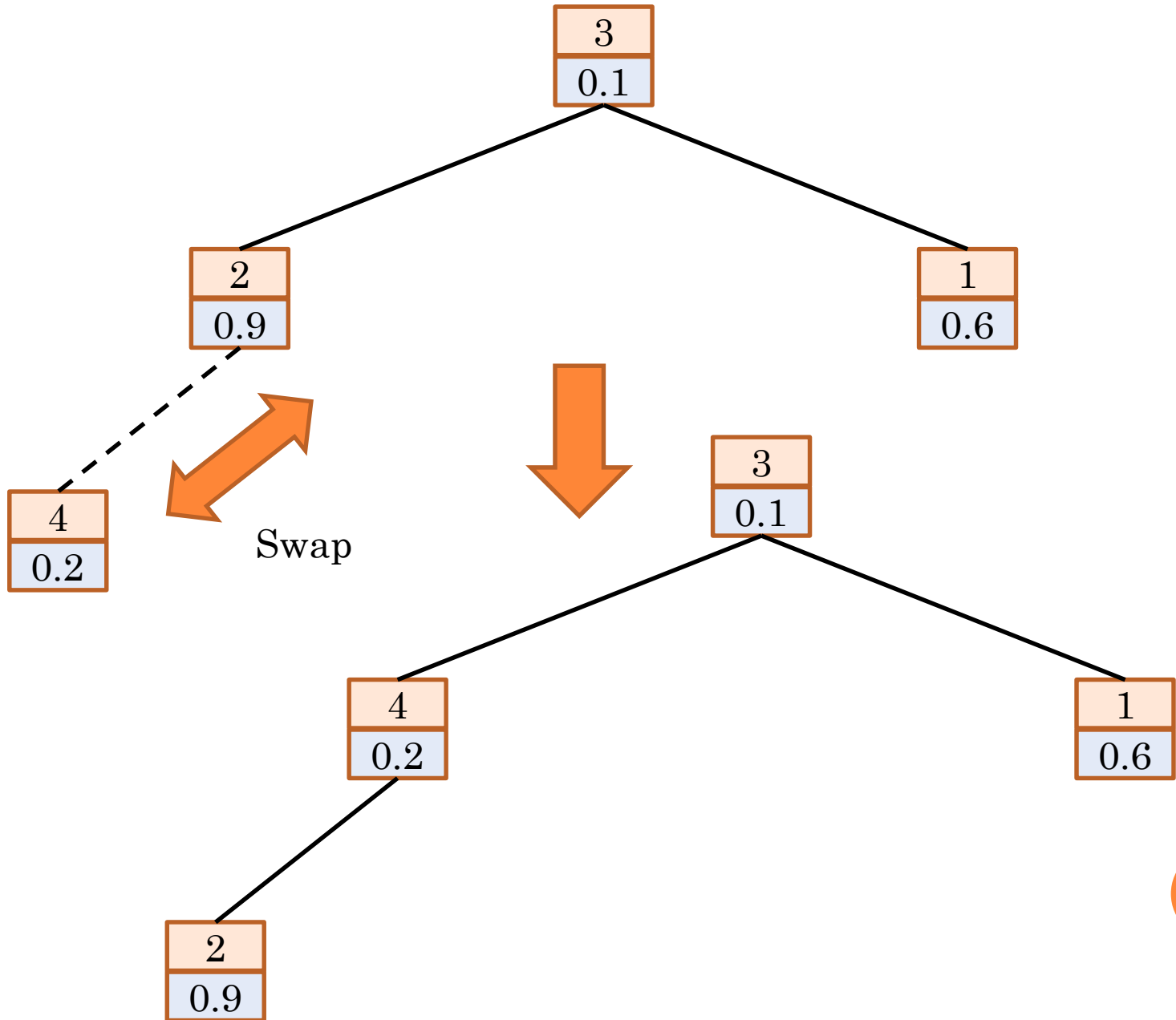


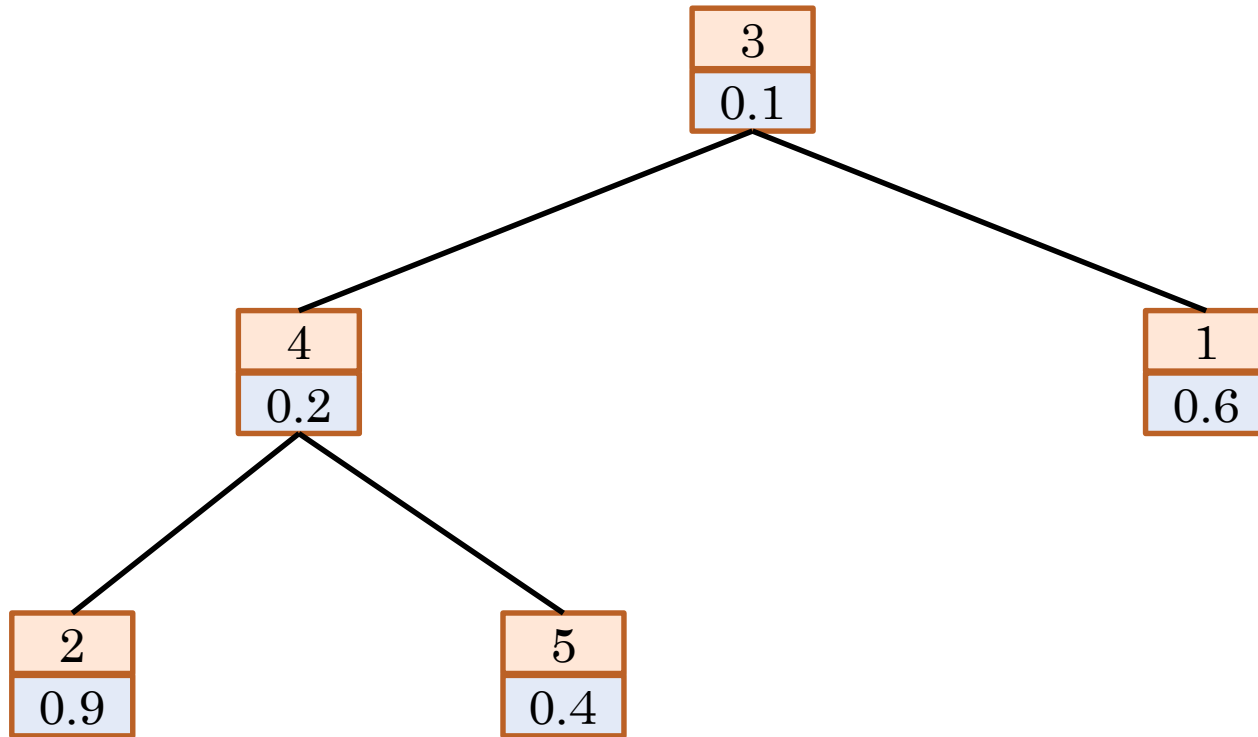
新要素を
末尾に追加

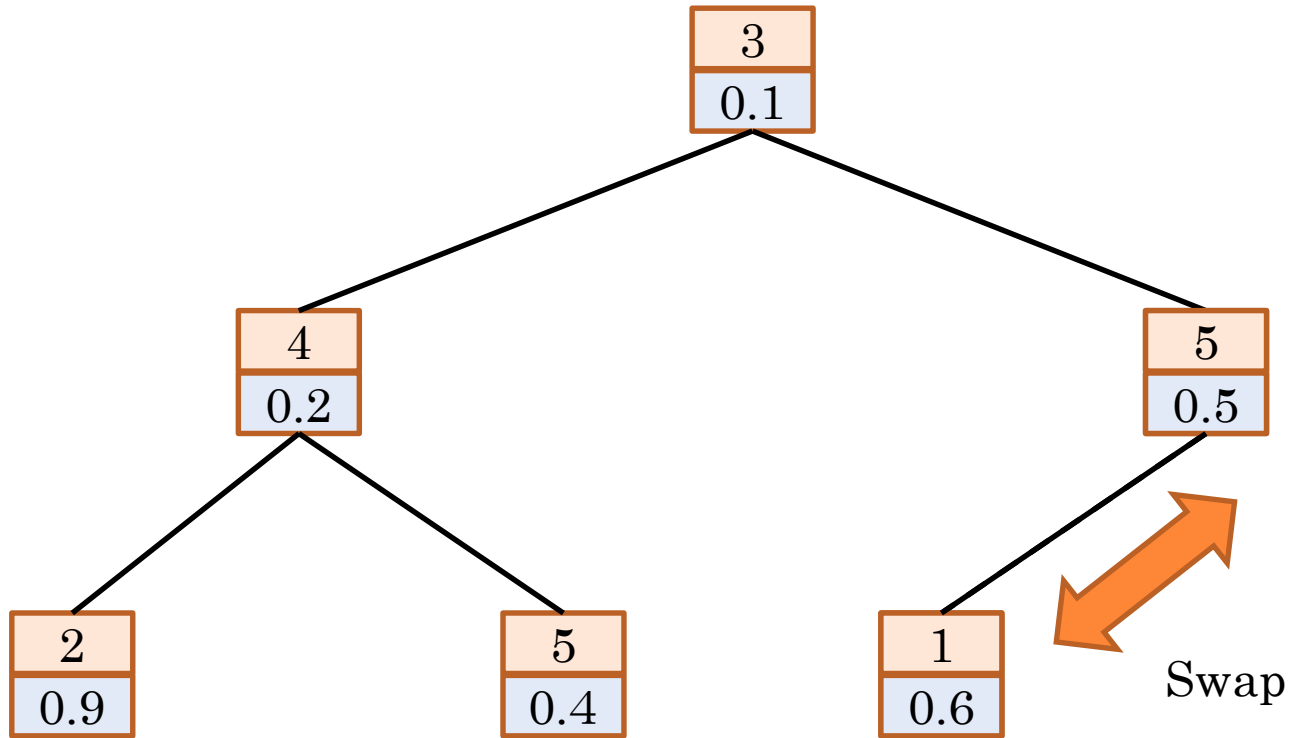


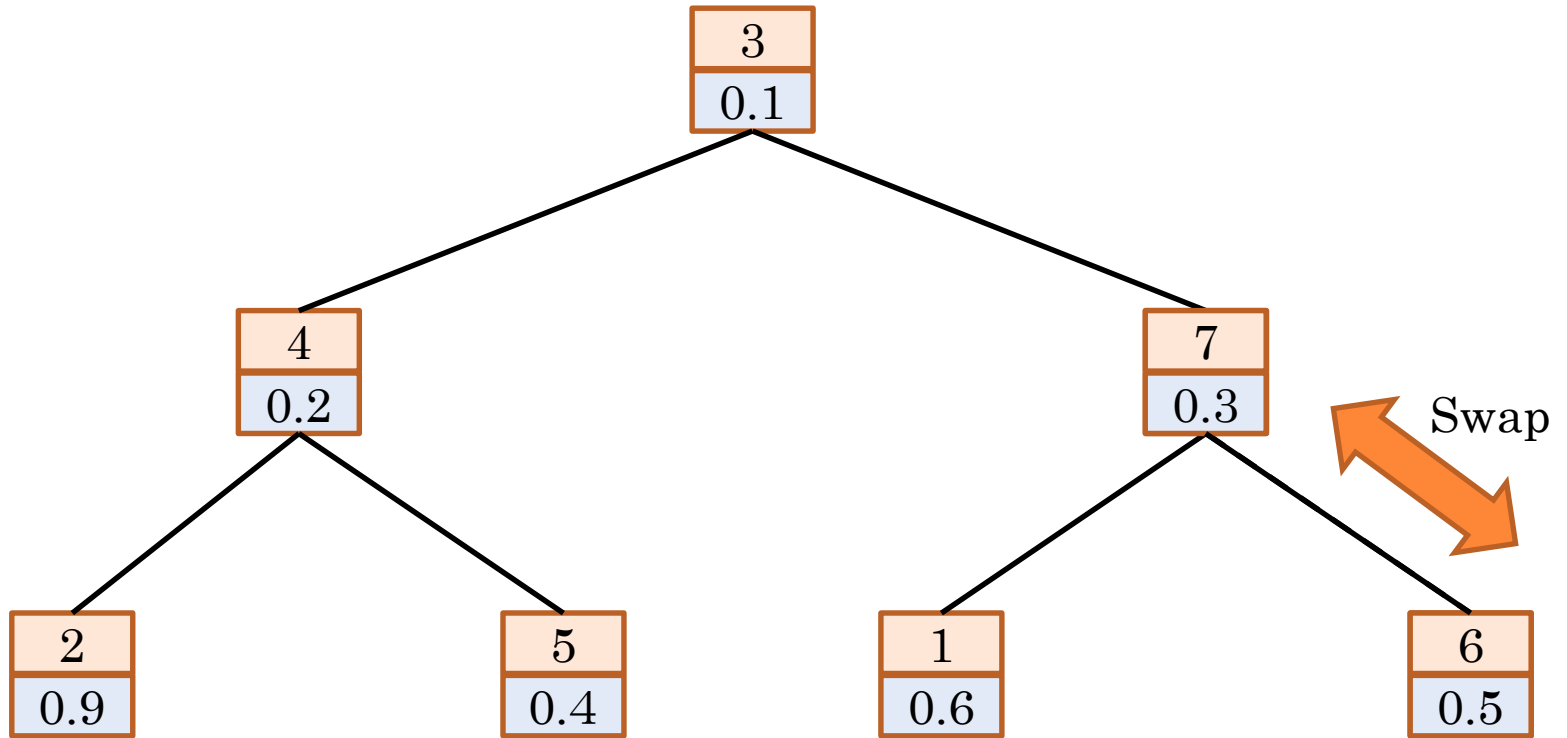


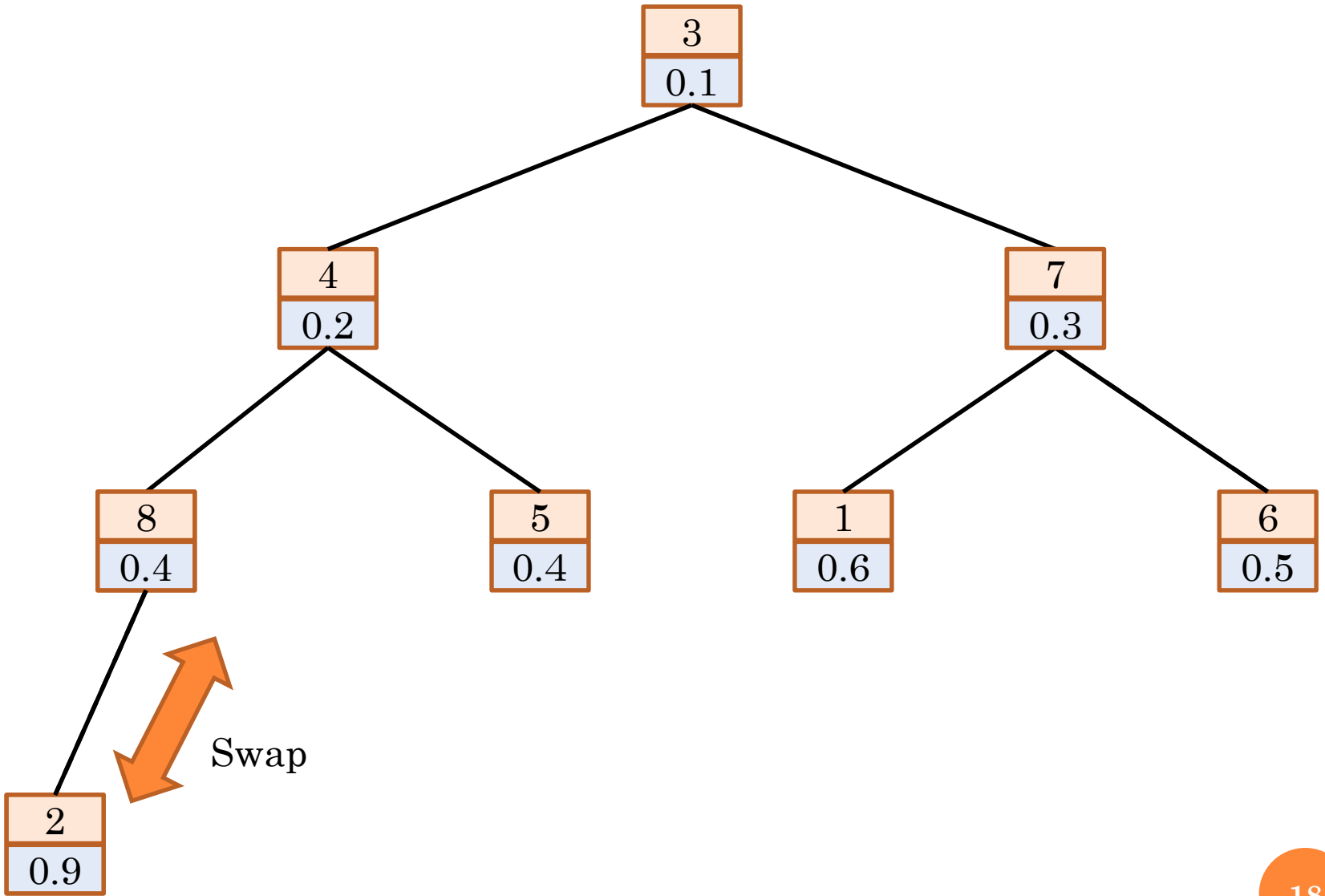












最小要素の取出し

- 最小要素は木の根として保存されている
- 最小要素を取り除き、再構築する
 - 最小要素の取り除き: リスト中の1番が空く
 - 最後尾の要素を取り除き、リストの1番に入れる
 - 適切な位置へシフトダウンする

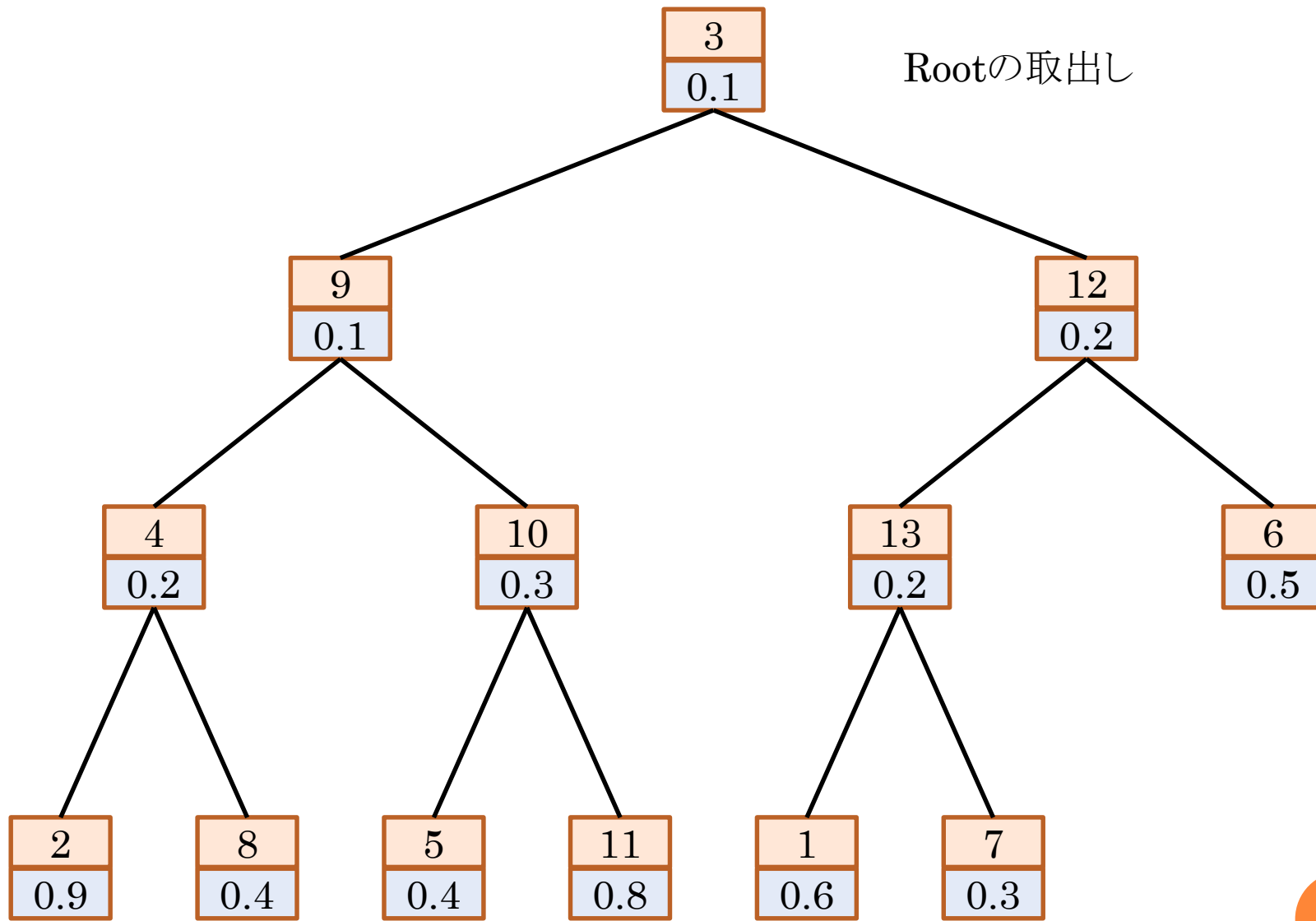
```
O poll(){  
    O t = L.get(1);  
    O x = L.removeLast();  
    L.set(1, x);  
    shiftDown(1);  
    return t;  
}
```

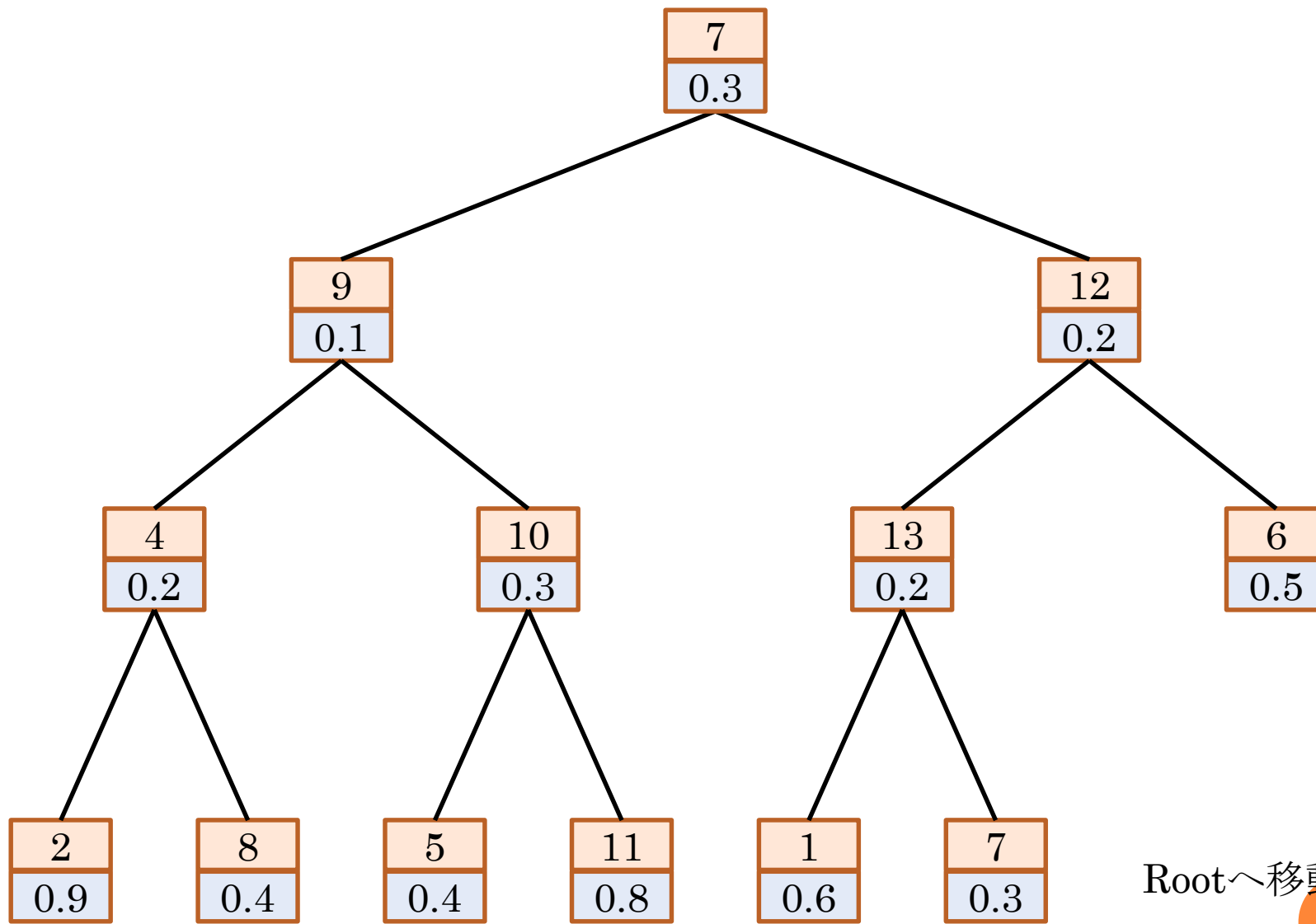
要素の取出し:シフトダウン

- 追加した新要素を正しい位置へ移動
- 位置 k の要素が、子の要素の位置 $2k$ と $2k + 1$ の小さいほうの値より大きい場合、その小さい値の子を入れ替える

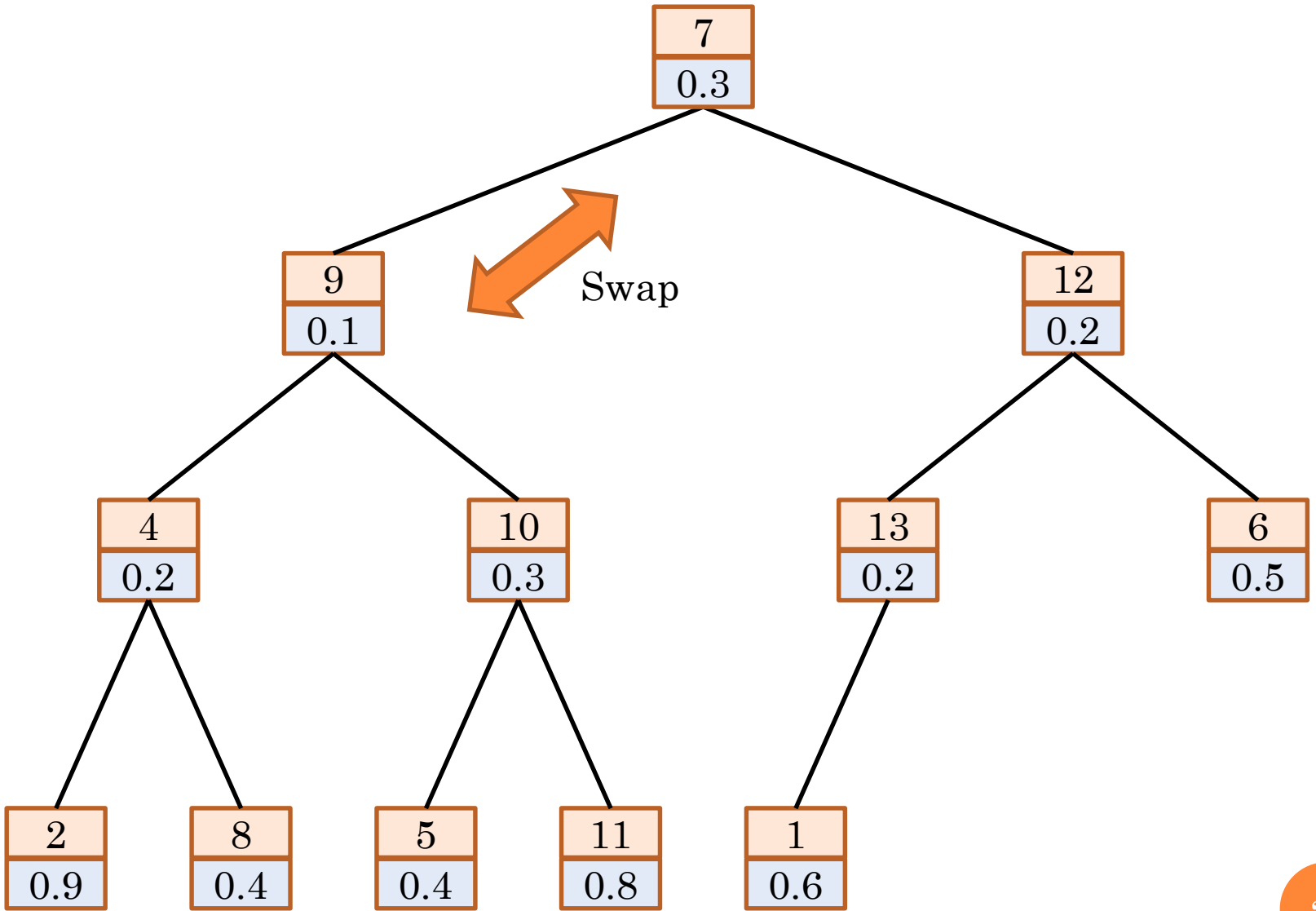
要素の取出し:シフトダウン

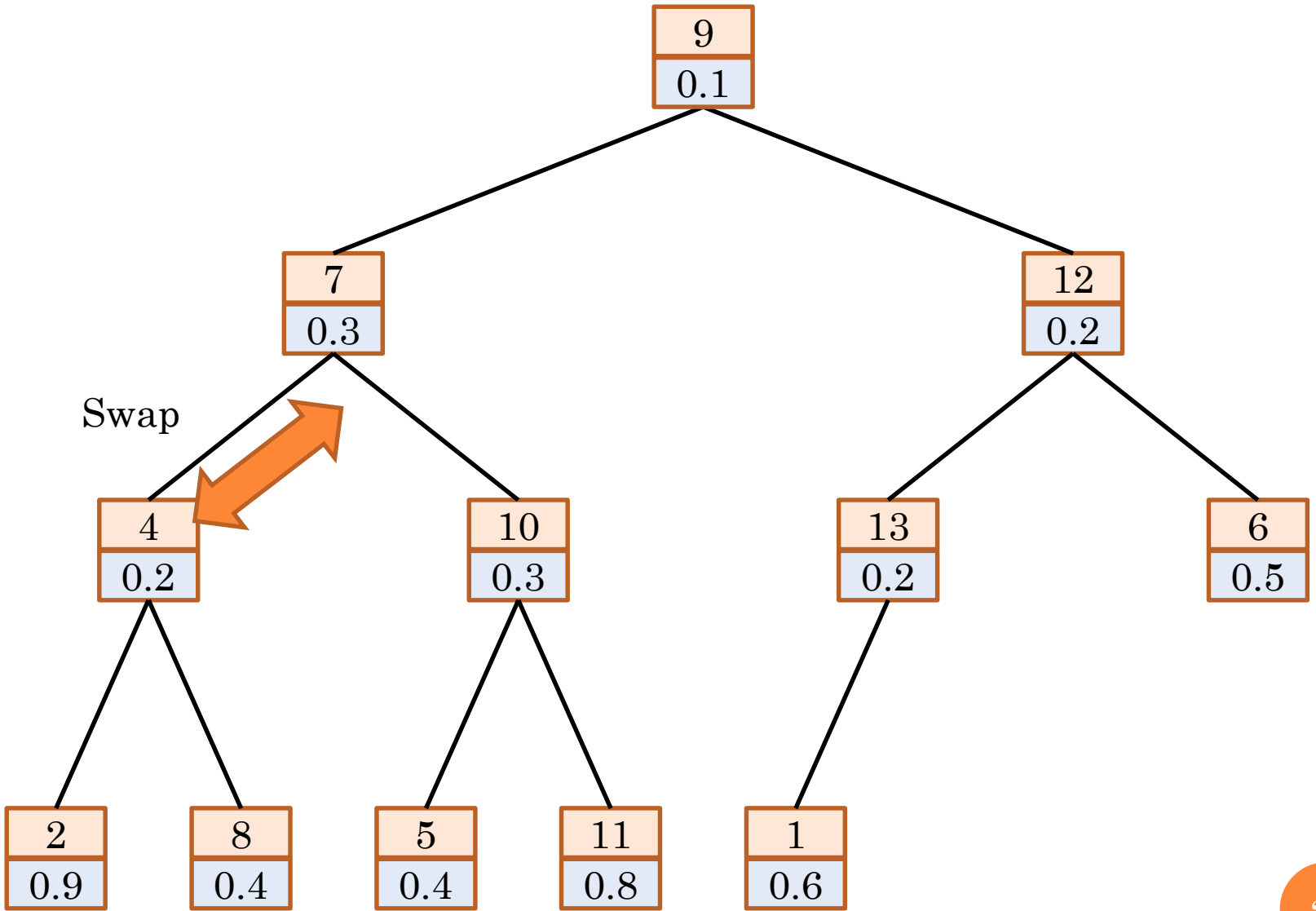
```
void shiftDown(int k){
    int n = |L|;
    if(2×k ≤ n){
        int j = 2×k;
        if(j < n && isLess(j+1, j)) j++;
        if(isLess(k, j))return;
        swap(k, j);
        shiftDown(j);
    }
}
```

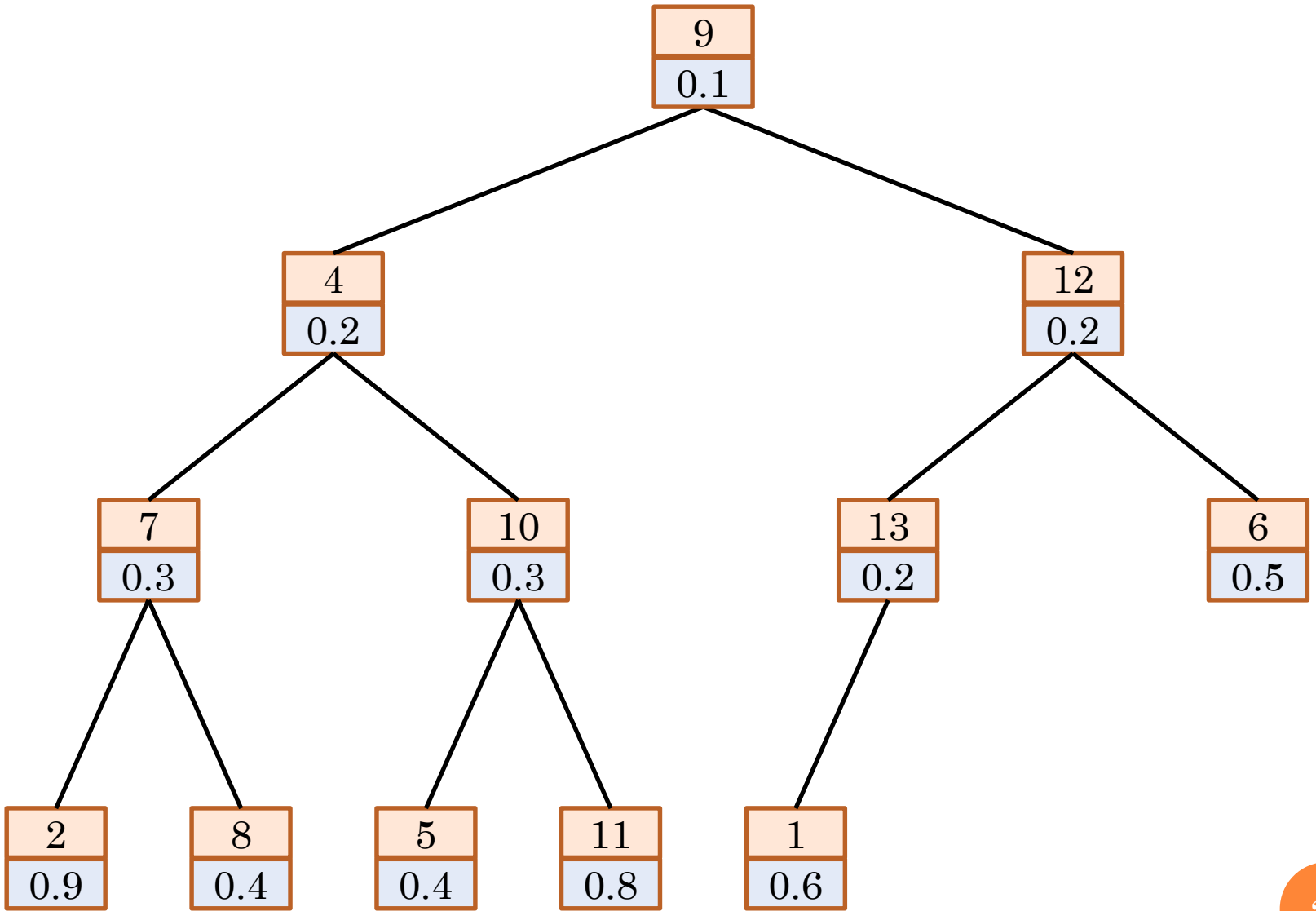




Rootへ移動







特定の要素の値を小さくする

- その要素のインデクス k
- `shiftUp`を k を起点に実施

```
void reduceValue(O o){  
     $k = o$  のリスト中のインデクス  
    shiftUp(k)  
}
```

特定の要素の値を大きくする

- その要素のインデクス k
- `shiftDown`を k を起点に実施

```
void raiseValue(O o){  
     $k = o$  のリスト中のインデクス  
    shiftDown( $k$ )  
}
```