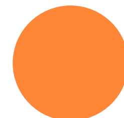
二分木ヒープ (Binary Heap)

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



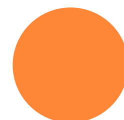
二分木ヒープとは、どういう二分木か

- 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

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

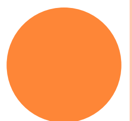
  - 半順序木になっている

- 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≤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) $a \le a$
  - 推移律 (transitivity) $(a \le b) \wedge (b \le c) \Rightarrow a \le c$
  - 反対照律 (antisymmetry) $(a \le b) \wedge (b \le a) \Rightarrow a = b$
- 任意の元の組$a,b \in P$に対して$a \le b$または$b \le a$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。



label

value

3
0.1
1

位置

9
0.1
2

12
0.2
3

4
0.2
4

10
0.3
5

7
0.3
6

6
0.5
7

2
0.9

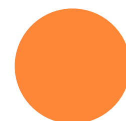
8
0.4

5
0.4

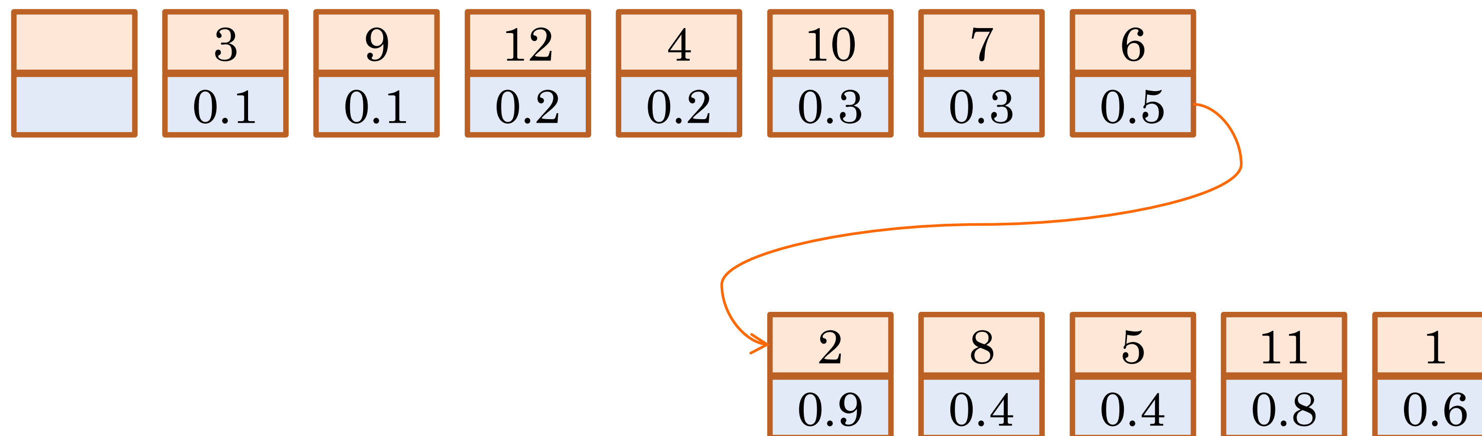
11
0.8

1
0.6

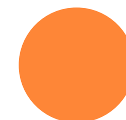
8 9 10 11 12



データの保持形式

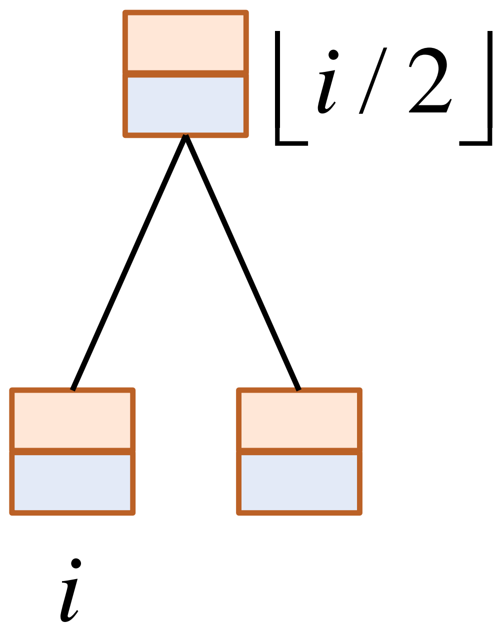


- リストで保持
  - 0番：空（リストのインデクスが1から始まるプログラミング言語では不要）
  - 1番：根の要素
  - 2^k番：第k層の左端の要素

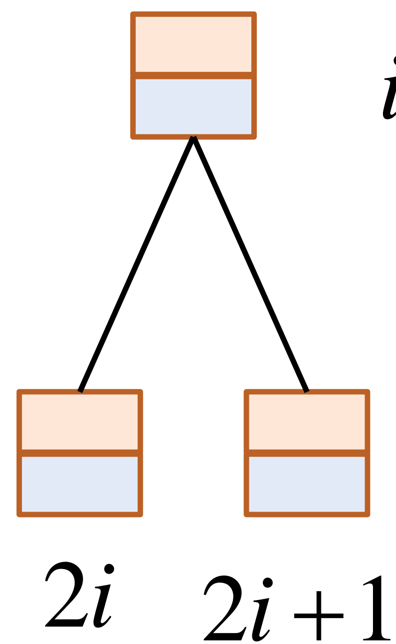


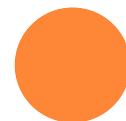
親子の番号の関係

- 親の番号



- 子の番号

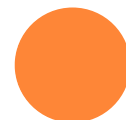




要素の追加

- 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

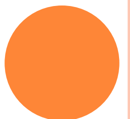
```
void add(O o){
        int  n = |L|;
        L.append(o);
        n++;
        shiftUp(n);//要素を正しい位置へ移動
}
```



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
- isLess(i,j)
  - $o_i.value < o_j.value$ のとき真
- swap(i,j):要素入れ替え

```
void shiftUp(int k){
        if(k > 1 && isLess(k, ⌊k/2⌋)){
                swap(k, ⌊k/2⌋);
                k = ⌊k/2⌋;
                shiftUp(k);
        }
}
```



3
0.1
1

9
0.1
2

12
0.2
3

4
0.2
4

10
0.3
5

7
0.3
6

6
0.5
7

2
0.9
8

8
0.4
9

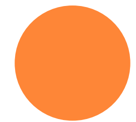
5
0.4
10

11
0.8
11

1
0.6
12

13
0.2
13

新要素を
末尾に追加



3
0.1
1

9
0.1
2

12
0.2
3

4
0.2
4

10
0.3
5

13
0.2
6

6
0.5
7

要素入れ替え

2
0.9

8
0.4

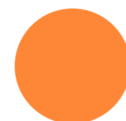
5
0.4

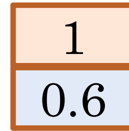
11
0.8

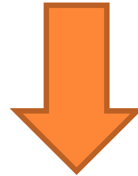
1
0.6

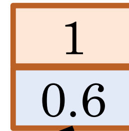
7
0.3

8 9 10 11 12 13

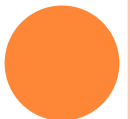


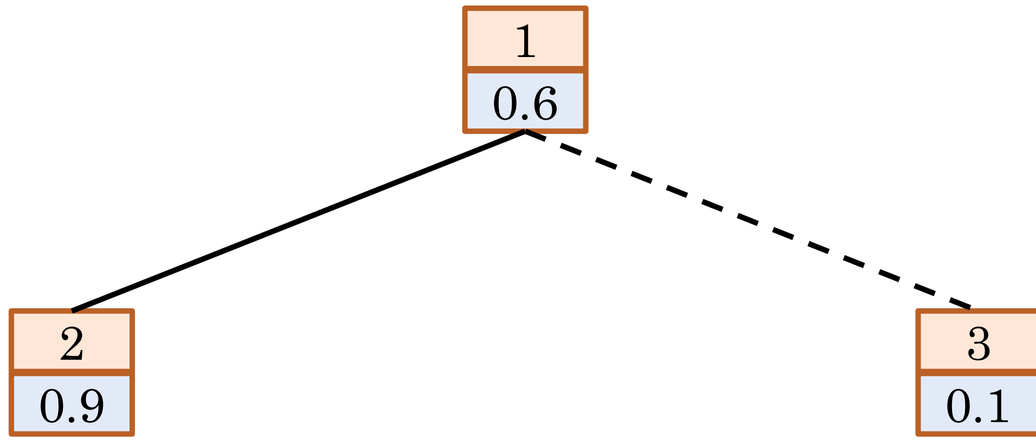

1
0.6

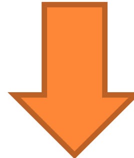


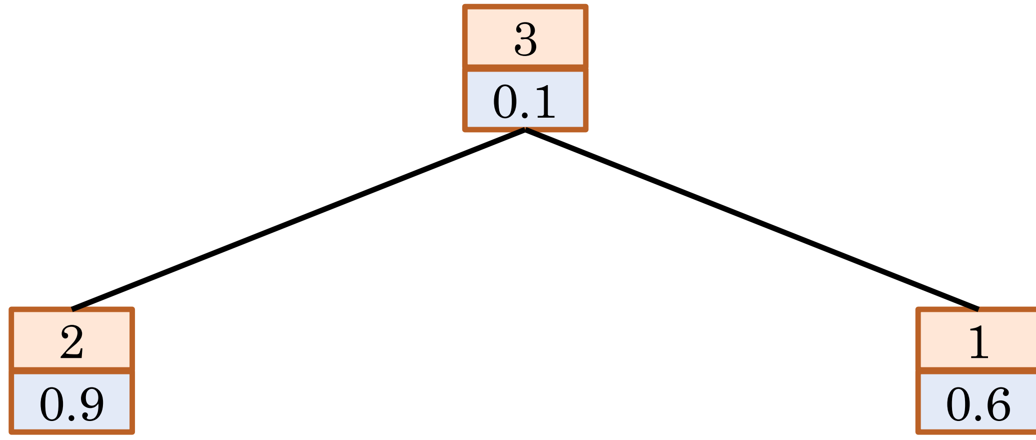

1
0.6

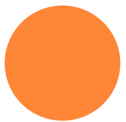
2
0.9
2

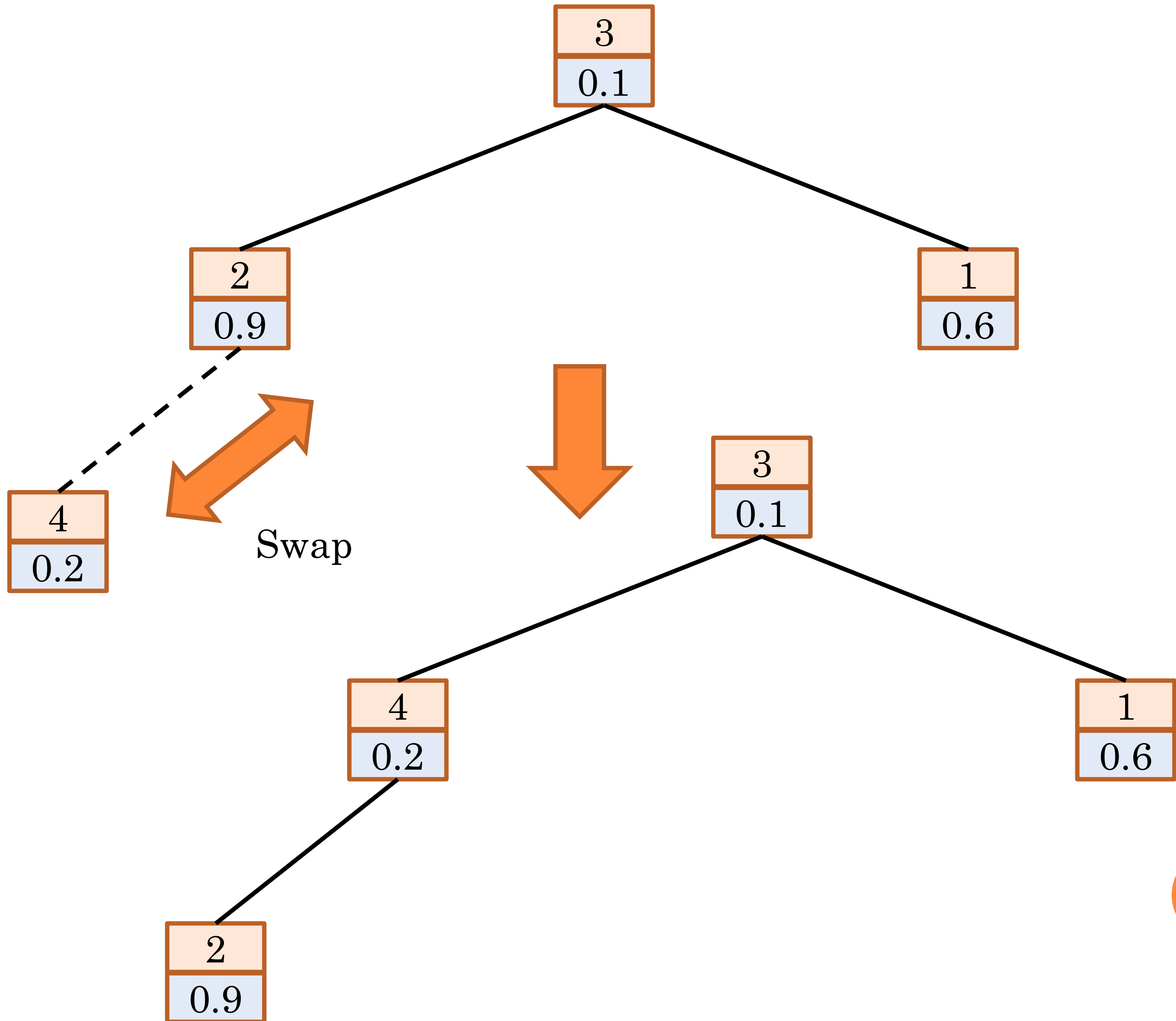


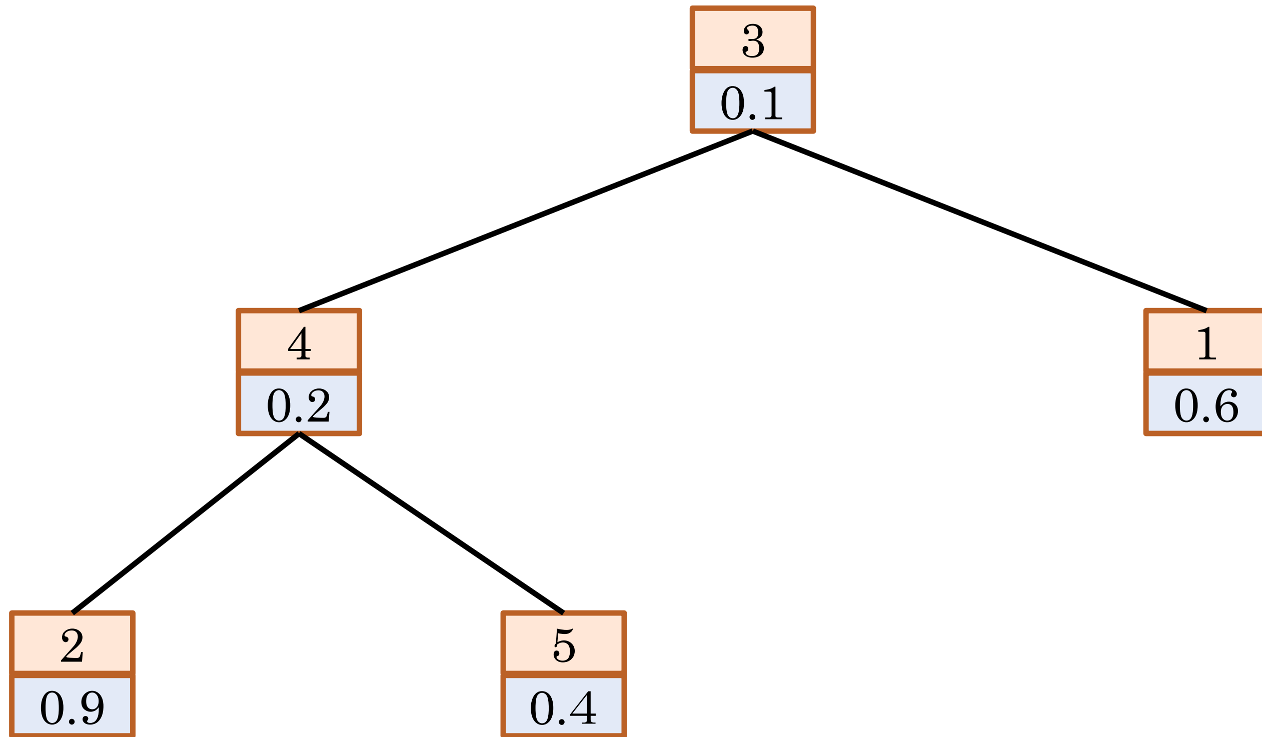
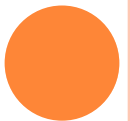

1
0.6
2
0.9
3
0.1

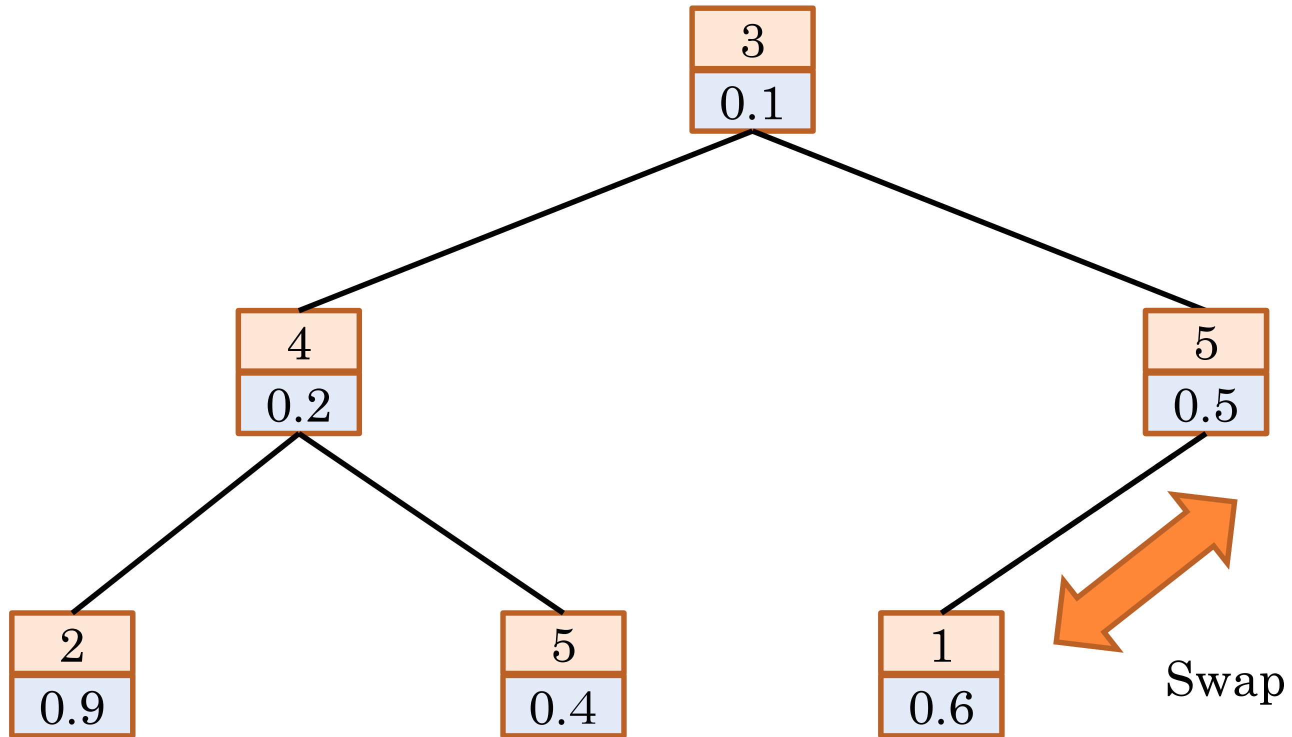


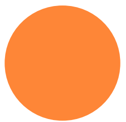

3
0.1
2
0.9
1
0.6




3
0.1
2
0.9
1
0.6
4
0.2
Swap
3
0.1
4
0.2
1
0.6
2
0.9


3
0.1
4
0.2
1
0.6
2
0.9
5
0.4



3
0.1
4
0.2
5
0.5
2
0.9
5
0.4
1
0.6
Swap



3
0.1

4
0.2

7
0.3

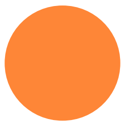
Swap

2
0.9

5
0.4

1
0.6

6
0.5



3
0.1

4
0.2

7
0.3

8
0.4

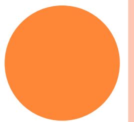
5
0.4

1
0.6

6
0.5

2
0.9

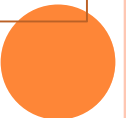
Swap



最小要素の取出し

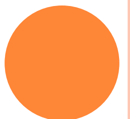
- 最小要素は木の根として保存されている
- 最小要素を取り除き、再構築する
  - 最小要素の取り除き:リスト中の1番が空く
  - 最後尾の要素を取り除き、リストの1番に入れる
  - 適切な位置へシフトダウンする

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



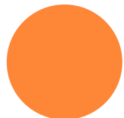
要素の取出し:シフトダウン

- 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の取出し

3
0.1

9
0.1

12
0.2

4
0.2

10
0.3

13
0.2

6
0.5

2
0.9

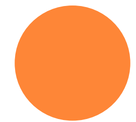
8
0.4

5
0.4

11
0.8

1
0.6

7
0.3



7
0.3

9
0.1

12
0.2

4
0.2

10
0.3

13
0.2

6
0.5

2
0.9

8
0.4

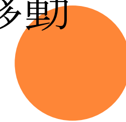
5
0.4

11
0.8

1
0.6

7
0.3

Rootへ移動



7
0.3

9
0.1

Swap

12
0.2

4
0.2

10
0.3

13
0.2

6
0.5

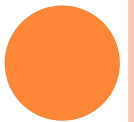
2
0.9

8
0.4

5
0.4

11
0.8

1
0.6



9
0.1

7
0.3

12
0.2

Swap

4
0.2

10
0.3

13
0.2

6
0.5

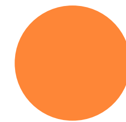
2
0.9

8
0.4

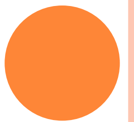
5
0.4

11
0.8

1
0.6



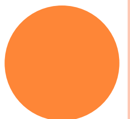
- 9 / 0.2
  - 4 / 0.2
    - 7 / 0.3
      - 2 / 0.9
      - 8 / 0.4
    - 10 / 0.3
      - 5 / 0.4
      - 11 / 0.8
  - 12 / 0.2
    - 13 / 0.2
      - 1 / 0.6
    - 6 / 0.5



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)
}
```