素を取り出して削除する処理を実行します。ここでは最後尾の要素を削除したいだけなので、self.arr.pop(len(self.size) -1) と記述しています。ヒープから要素を 1 つ取り出したので、64 行目で self.size の値を 1 減らします。

最後尾の要素を先頭に移動させたので、この時点でヒープの性質が失われています。そこで 65 行目の命令で max_heapify メソッドを呼び出し、ヒープ構造に全体に対してヒープ化を行います。最後の 67 行目で、変数 max に保存しておいた要素を戻り値として返します。

具体例を**図 6.55** に示します。**1** create_max_heap メソッドで構成したヒープから一番優先度の高い要素は self.arr [0] に格納されているので、それを取り出します。**2** 取り出した要素は削除するので、最後尾にある self.arr [self.size-1] を self.arr [0] にコピーして、**3** 最後尾の要素を削除します。

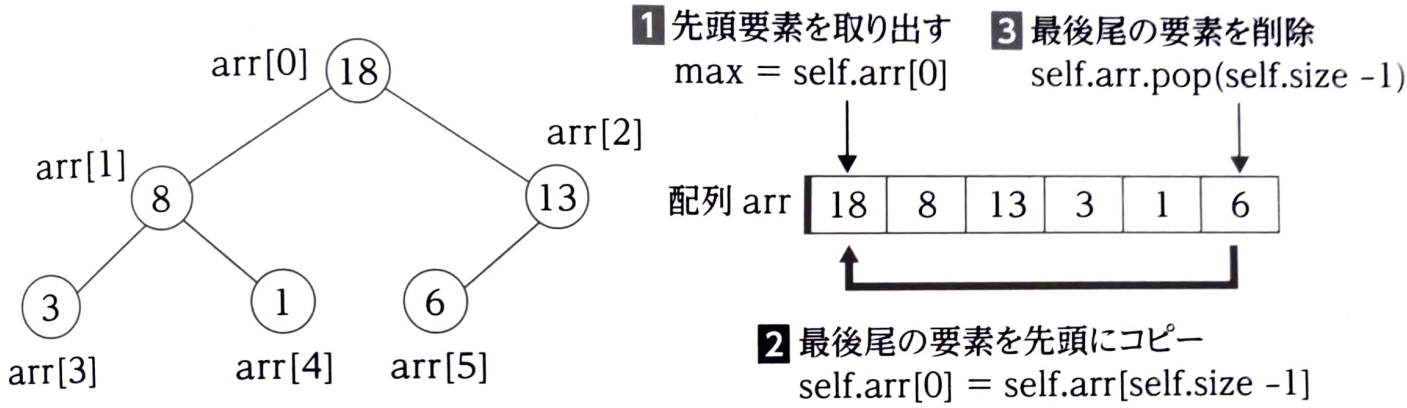

図 6.55 extract_max メソッドの例 1

要素の取り出しと、削除が終わった後のヒープの状態を**図 6.56** に示します。最後尾にあった整数値 6 が self.arr [0] に移動し、配列の大きさが 1 つ小さくなっています。二分木のほうも同様に変更されています。

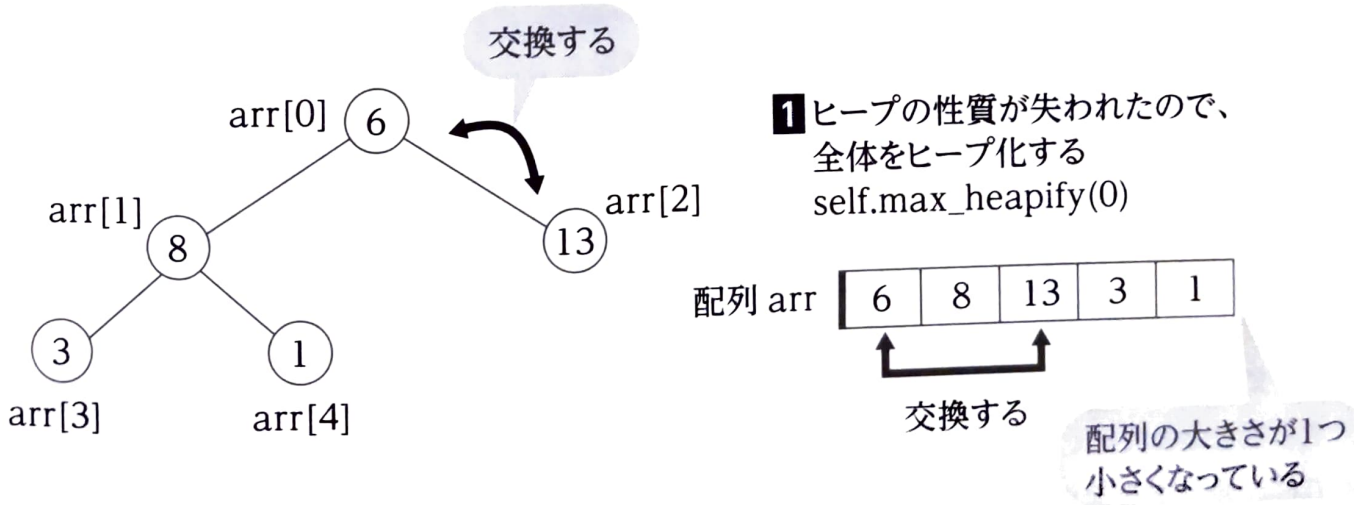

図 6.56 extract_max メソッドの例 2

この時点でヒープの性質を失っているので、二分木全体に対して **1** self.max_heapify(0) を実行

し、ヒープ化します。そのあとの状態を**図 6.57** に示します。取り出した要素が削除され、二分木と配列が再構成されていることが確認できます。

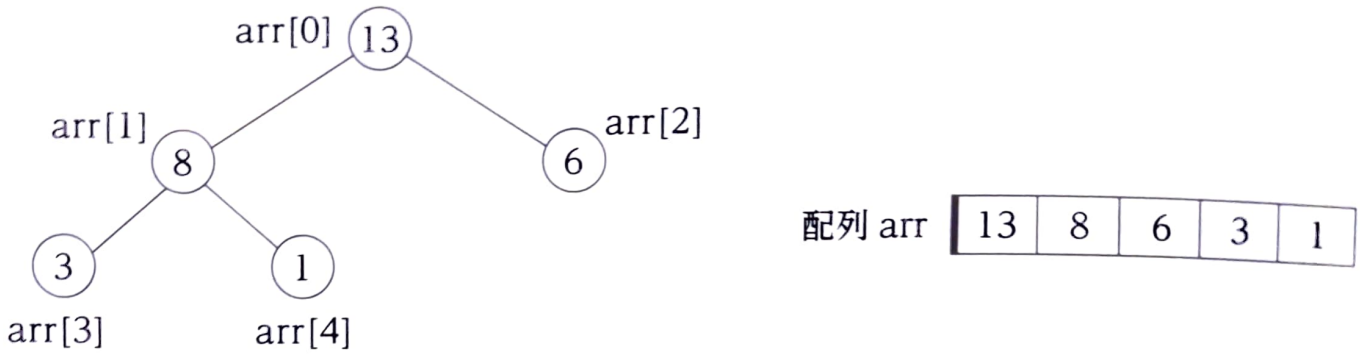

図 6.57 extract_max メソッドの例 3　取り出した要素が削除され、二分木と配列が再構成された

insert メソッド（新たな要素の挿入）の説明

70 行目～ 73 行目で新たな要素を挿入する insert メソッドを定義しています。引数として、新しい要素である整数値を変数 val で受け取ります。要素数が 1 つ増えるため、71 行目で self.size の値を 1 増加させます。72 行目では、いったん -∞ を値としてもつ要素を配列 arr の最後尾に挿入します。そして、73 行目の increase_val メソッドで新たに追加した要素の値を本来の値である val に変更します。以後の処理は、次に説明する increase_val メソッドと同じです。

increase_val メソッド（要素がもつ値の更新）の説明

76 行目～ 86 行目で要素がもつ値を更新する increase_val メソッドを定義しています。引数として、値を変更したい配列のインデックスと変更後の値をそれぞれ変数 index と val で受け取ります。78 行目～ 80 行目では、self.arr [index] の値と val の値を比較して、val のほうが値が小さければ処理を中断します。値を減少させる場合は処理内容が異なるため、increase_val メソッドでは対応できないからです。変数 val の値のほうが大きい場合は、処理を続行します。

83 行目で self.arr [index] の値を val に更新します。ここでヒープの性質が失われる可能性があるので、前項の説明のとおり、二分木の根に向かって節点を交換していきます。84 行目の while 文を用いて繰り返し処理を行います。ソースコード内では配列に対して処理を行うので、self.arr [index] とその親である self.arr [self.get_parent(index)] の値を比較して交換処理を行います。

while 文の条件式に index ＞ 0 が含まれています。index の値が 0 であれば、self.arr [index] は二分木の根となるため、これ以上ループを続ける必要はありません。そのため、index ＞ 0 が False であれば、ループを抜けます。もう片方の条件式は、self.arr [index] の値がその親の値よりも大きいかどうかです。False であれば、すでにヒープの性質を満たしていることを意味するので while ループを抜けます。False の場合は、if ブロックの中に入り、85 行目で self.arr [index] とその親の値を swap 関数で交換します。86 行目で index の値をその親のインデックスに更新し、ループを繰

り返します。

図 6.58 に示すヒープに整数値 10 をもつ新たな要素を挿入する例を示します。たとえば、main 関数内で my_heap.insert(10) を実行したとします。70 行目の insert メソッドから処理が始まります。self.arr の中身を [13, 8, 6, 3, 1] とします。まず、71 行目と 72 行目で、配列 arr の大きさを 1 つ増やして、負の無限大の値 (-∞) をもつ要素を挿入します。self.arr が [13, 8, 6, 3, 1, -∞] となるため、配列の大きさである self.size の値は 6 になります。その後の 73 行目で self.increase_val(5, 10) を実行します。すなわち、self.arr [5] に格納されいてる -∞の値を整数値 5 に更新します。

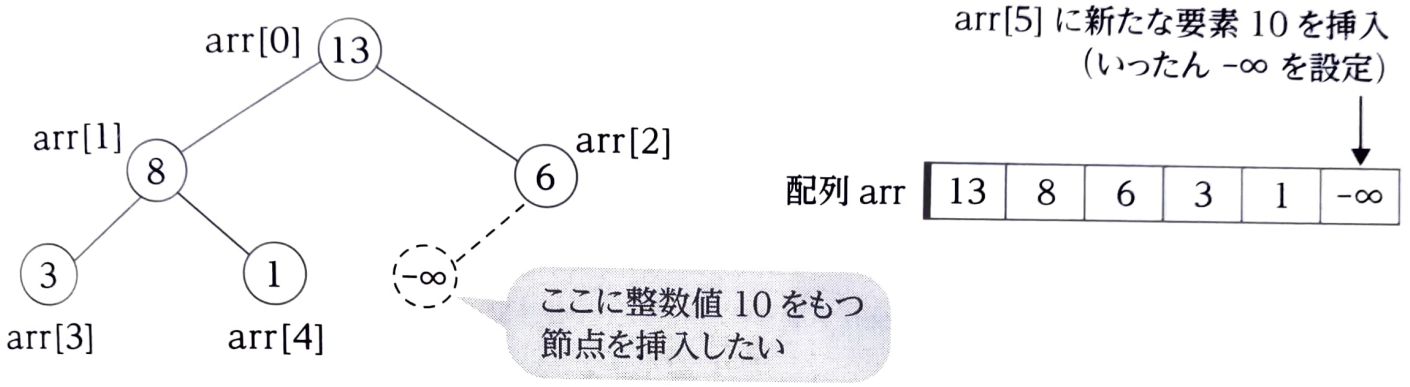

図 6.58 insert メソッドと inrease_val メソッドの例 1

プログラムの制御が 76 行目の increase_val メソッドに移ります。**1** 引数の index と val の値はそれぞれ 5 と 10 です。-∞を整数値の 10 に変更するため、78 行目の if 文の条件判定は False となり、78 行目～ 80 行目の処理は飛ばします。**2** 83 行目の self.arr [index] = val という命令で、self.arr [5] の値を 10 にします。84 行目の while ループを開始する前のヒープの状態を**図 6.59** に示します。

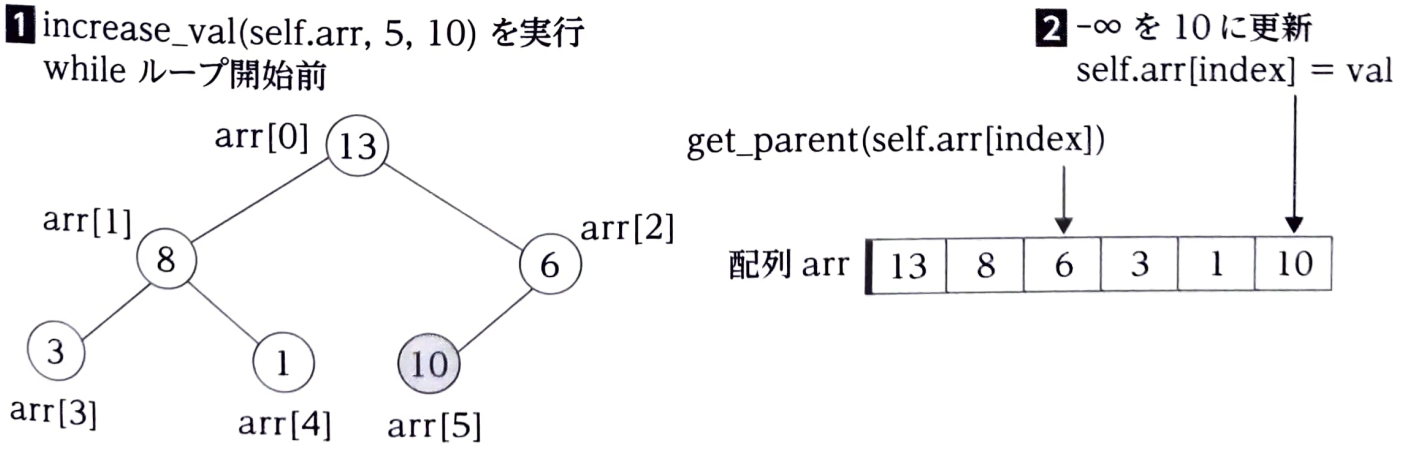

図 6.59 insert メソッドと inrease_val メソッドの例 2

while ループで self.arr [5] の親であるインデックスを計算します。親のインデックスは 2 です。self.arr [5] と self.arr [2] を比べると self.arr [5] のほうが大きいことがわかります。while ループ

の条件判定が True なので、ループ内に入ります。85 行目の命令で、self.arr [5] と self.arr [2] を交換します。86 行目で index の値をその親のインデックスに更新し、ヒープを表す二分木の上の階層へ進みます。1 回目のイテレーション終了時の状態を**図 6.60** に示します。配列の中身が変わっていることを確認してください。

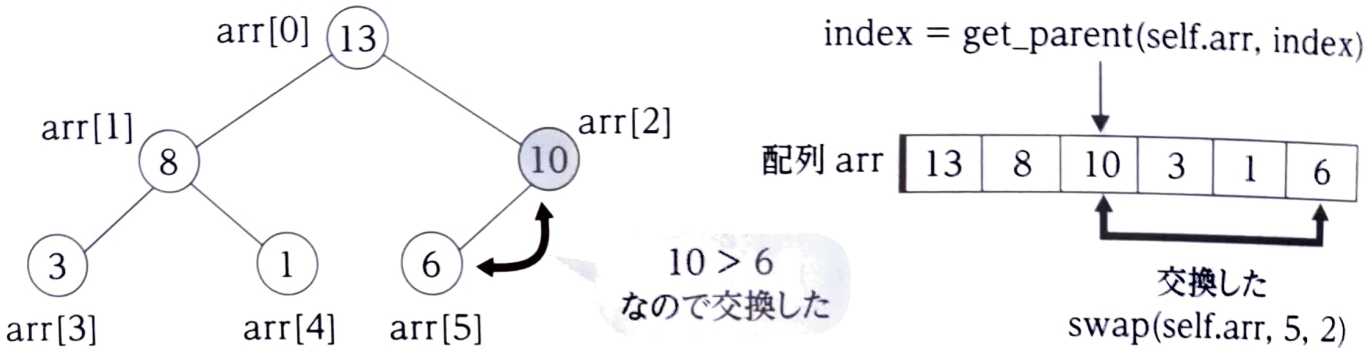

図 6.60 insert メソッドと inrease_val メソッドの例 3

2 回目のイテレーションでは、index の値が 2、その親のインデックスは 0 です。**1** self.arr [0] の値が self.arr [2] の値よりも大きいので、while ループの条件判定が False になります。そのため、ループを抜けます。**図 6.61** に示す二分木のとおり、すでにヒープの性質を満たしています。

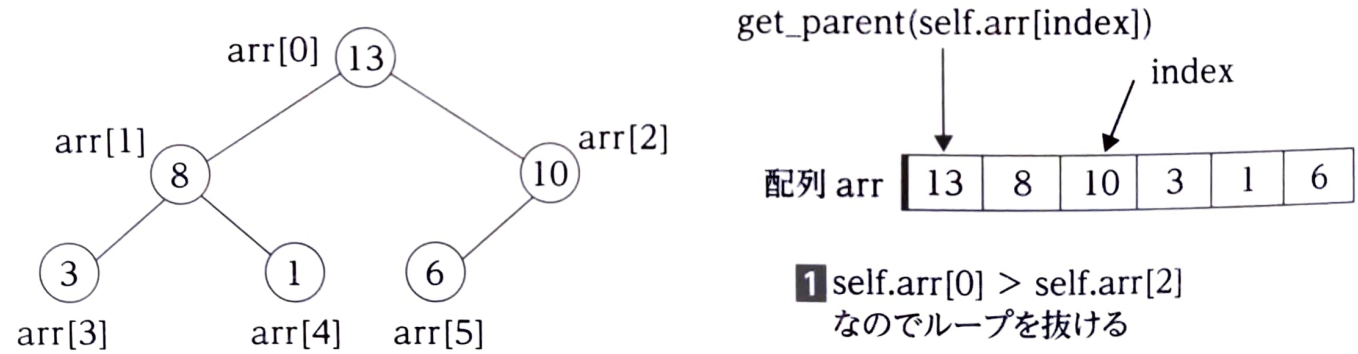

図 6.61 insert メソッドと inrease_val メソッドの例 4

以上で、要素の挿入処理と優先度の更新処理は終了です。配列 arr の中身は [13, 8, 10, 3, 1, 6] となります。

heap sort メソッド（ヒープソートの実行）の説明

89 行目～97 行目でヒープソートを実行する heap sort メソッドを定義しています。self.arr はヒープの性質を満たすように並んでいますが、ヒープソートは self.arr を昇順に並べ替えます。前項の

概要で解説したとおり、self.arr の最後尾の要素から順番を確定していきます。

91 行目の for ループで self.size-1 から 1 になるまで -1 ずつイテレーションします。ループカウンタ i は、未ソートの部分配列の最後尾のインデックスを指します。ヒープ内で一番大きな値は常に self.arr [0] に格納されています。そのため、92 行目で self.arr の先頭要素と未ソートの部分配列の最後尾の要素を交換し、93 行目で self.size を 1 減らします。self.size の値を減らすだけで、self.arr から要素を削除するわけではありませんので注意してください。

要素の交換を行うとヒープの性質が失われるので、94 行目で max_heapify メソッドでヒープ化します。このときにインデックスの 0 を指定しているため、二分木の根からヒープ化を行いますが、self.size の値が self.arr の大きさより小さくなっているはずなので、実際にはソート済みの節点を除いた部分木 (self.arr [0:self.size-1]) に対してヒープ化の処理をすることとなります。最後に 97 行目で self.size の値を len(self.arr) の値に戻します。

配列 arr が [13, 8, 10, 3, 1, 6] のときに、heap_sort メソッドを実行する具体例を示します。**図 6.62** に 91 行目の for ループ開始前のヒープの状態を示します。

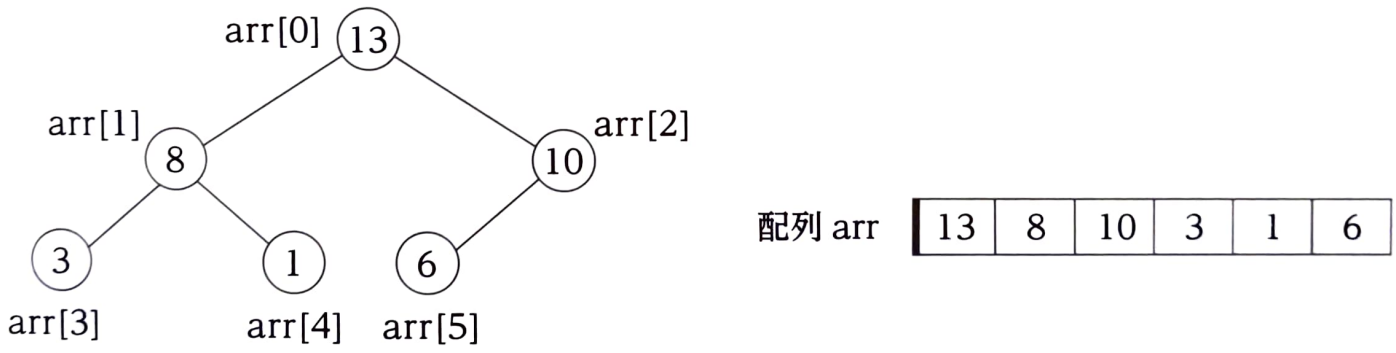

図 6.62 heap_sort メソッドの例 (1/6)

1 回目のイテレーションでは、ループカウンタ i の値は 5 です。ループに入り、92 行目～ 93 行目の処理が終わったときの配列 arr の状態を**図 6.63** の右側の上に示します。**1** self.arr [0] にあった整数値 13 と self.arr [5] にあった整数値 6 の場所が入れ替わります。self.size の値を 1 つ減らし、未ソートの部分配列として arr [0:4] を参照するようにします。そして 94 行目の **2** self.max_heapify(0) を実行します。max_heapify メソッド内で self.size を参照しますが、値が 4 となっているので、部分配列 arr [0:4] に対してヒープ化が行われます。1 回目のイテレーションが終了したときの二分木の状態と配列の中身をそれぞれ**図 6.63** の左側と右側の下に示します。なお、図内の灰色で塗りつぶした二分木の節点と配列の要素はソート済みであることを意味します。以降、同じ要領でイテレーション毎のヒープの状態を示します。

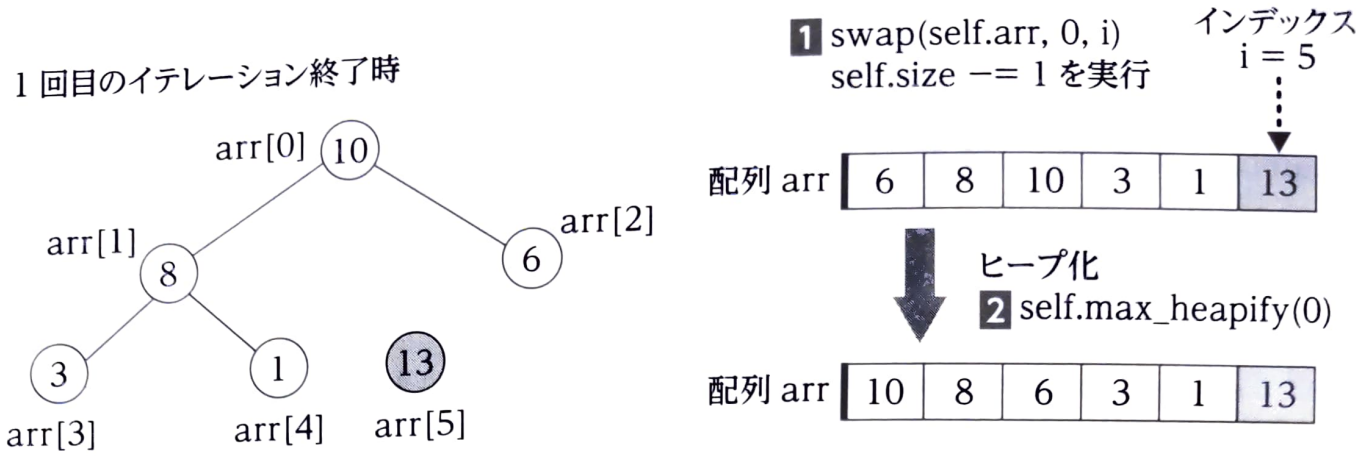

図 6.63　heap_sort メソッドの例（2/6）

2 回目のイテレーションに入り、**1** self.arr[0] と値を self.arr[4] の値を交換し、**2** ヒープ化を行います。2 回目のイテレーション終了後の状態を**図 6.64** に示します。

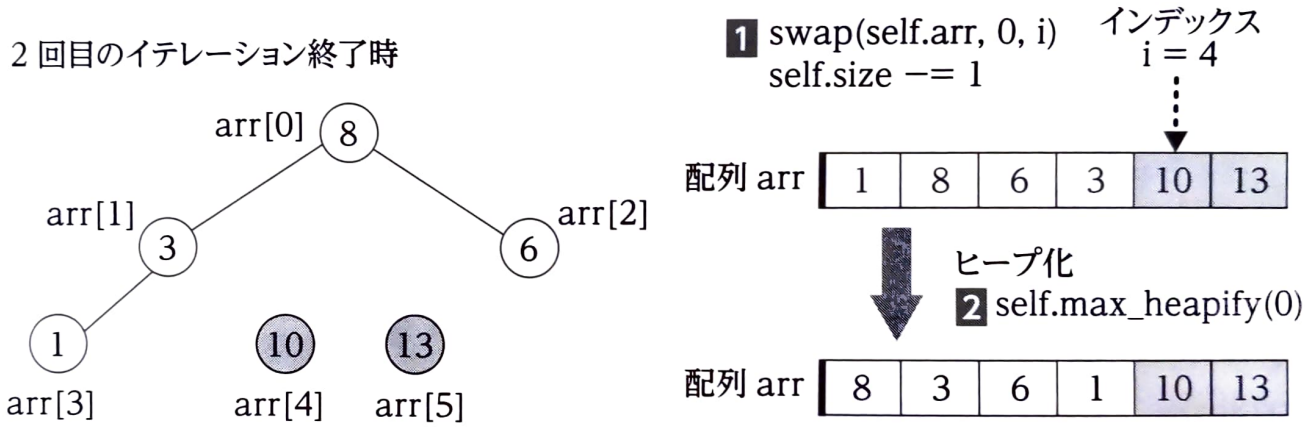

図 6.64　heap_sort メソッドの例（3/6）

以降も同様の処理を繰り返します。3 回目と 4 回目のイテレーション終了後の状態を**図 6.65** と**図 6.66** に示します。イテレーション毎に、配列の中身がどのように変化するかを確認してください。

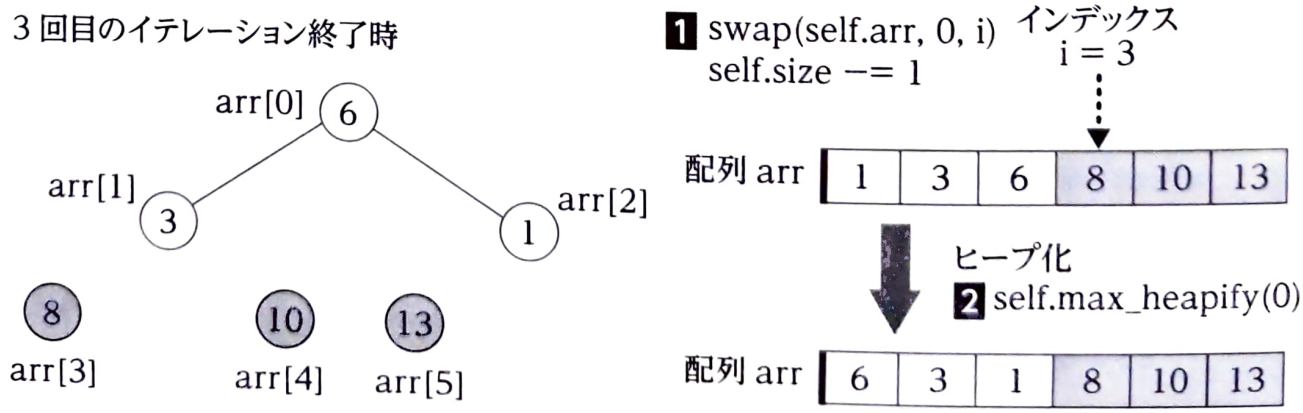

図 6.65　heap_sort メソッドの例（4/6）

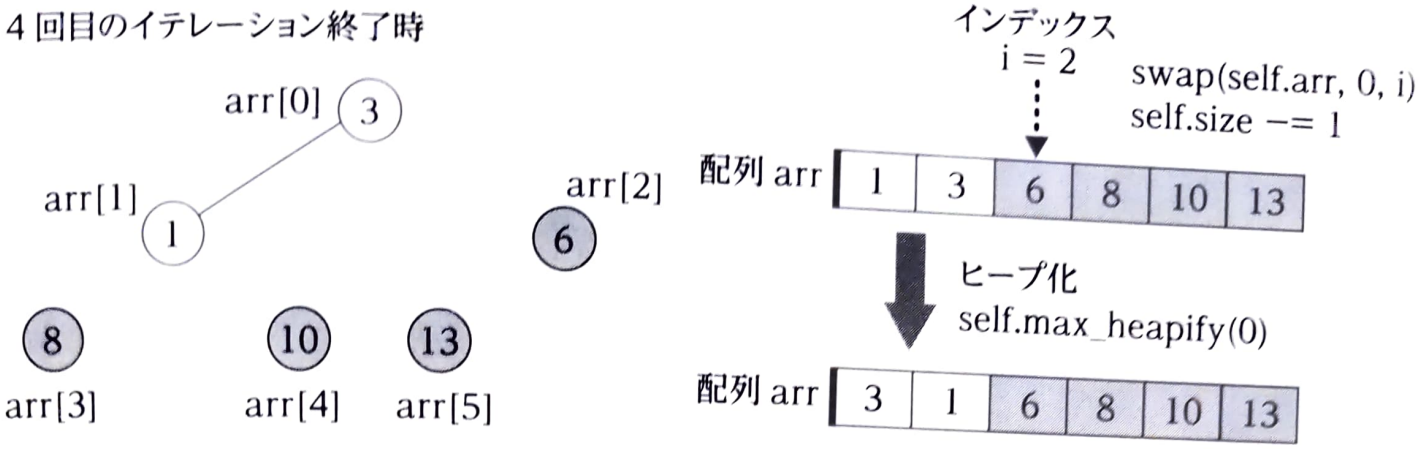

図 6.66 heap_sort メソッドの例 (5/6)

5 回目のイテレーション終了後の状態を**図 6.67** に示します。for ループはループカウンタ i の値が 1 のときが最後のイテレーションになります。部分配列 arr[1:5] をソートした時点で、self.arr[0] の値は self.arr[1] ~ self.arr[5] のどの値よりも小さくなります。そのため、ループカウンタ i が 0 の場合のイテレーションを繰り返す必要はありません。

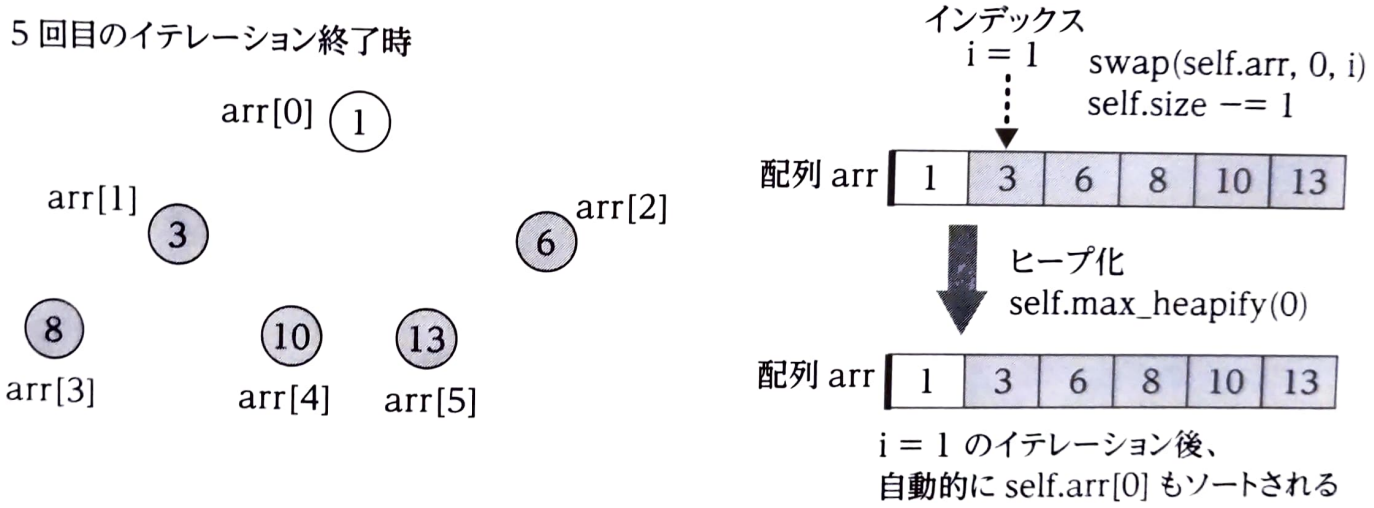

図 6.67 heap_sort メソッドの例 (6/6)

図 6.67 で示す二分木と配列から確認できるように、各要素が昇順にソートされています。

■ ヒープの実装プログラムの実行

ソースコード 6.3 (my_heap.py) をコンパイルして実行した結果を**ログ 6.4** に示します。2 行目で MyHeap のインスタンスを生成したあとのヒープの中身が表示されています。すでにヒープ化された状態になっています。create_max_heap メソッドの説明で示した**図 6.54** で示した配列と同じです。

3 行目は extract_max メソッドで取り出した要素の値を表示しています。2 行目で示した配列の

先頭要素（優先度が一番高い要素）が 18 なので、3 行目でも整数値の 18 が表示されます。取り出した整数値 18 をもつ要素はヒープから削除されます。そのときのヒープの中身が 4 行目に表示されています。少しページを戻って**図 6.57** を参照していただくと、説明どおりの結果が表示されていることが確認できます。

ログ 6.4　my_heap.py プログラムの実行

```
01 $ python3 my_heap.py
02 最大ヒープ化後のヒープ:  [18, 8, 13, 3, 1, 6]
03 取り出した要素:  18
04 要素の削除後のヒープ:  [13, 8, 6, 3, 1]
05 要素の挿入後のヒープ:  [13, 8, 10, 3, 1, 6]
06 ソート後のヒープ:  [1, 3, 6, 8, 10, 13]
```

5 行目はヒープに整数値 10 をもつ新たな要素を追加した後のヒープの中身です。**図 6.61** で示したとおりです。最後の 6 行目は、ヒープソートを適応したあとの状態を示しています。5 行目で表示された配列が、6 行目で昇順にソートされていることが確認できます。