

整数値 22 をもつ要素の削除後の二分探索木
(一番小さい 15 の要素の下に並びなおされる)

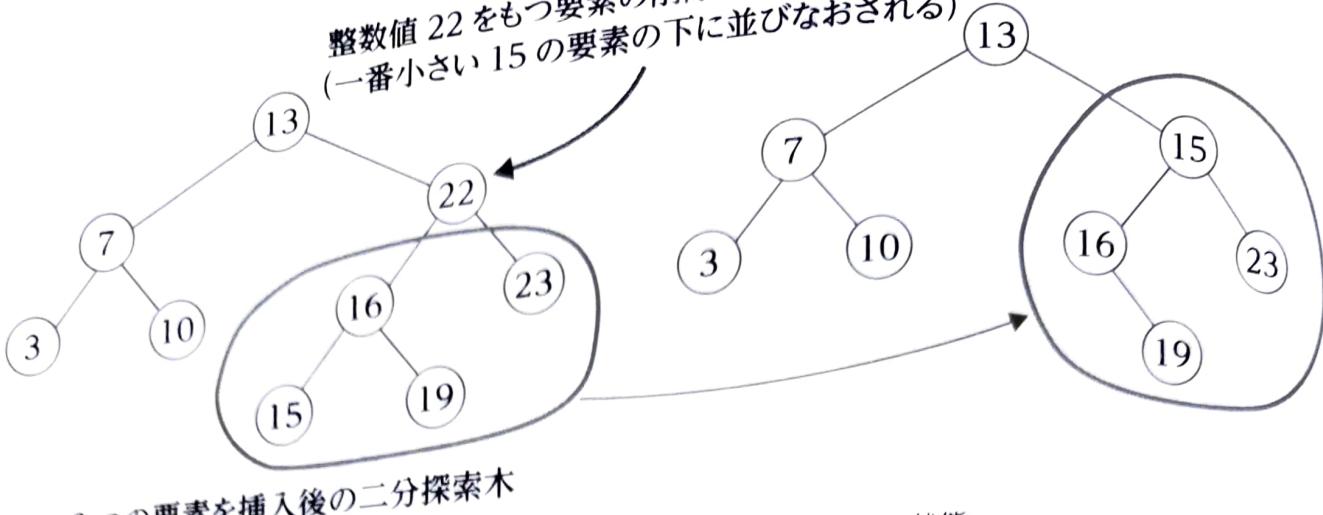


図 6.27 要素挿入後と削除後の二分探索木の状態

6.4 ヒープ (heap)

ヒープ (heap) とは二分木を用いたデータ構造で、**最大ヒープ (max heap)** と**最小ヒープ (min heap)** の 2 つに分類されます。最大ヒープは、二分木において、子がもつ値よりも親がもつ値のほうが大きいか等しい、という性質をもつデータ構造です。言い換えると、木の根となる要素がデータ構造内で一番大きな値をもちます。最小ヒープはその反対です。本節では最大ヒープを用いてヒープの仕組みを解説します。

ヒープは**優先度付きキュー (priority queue)** として使用できます。優先度付きキューの具体例は次項で説明します。最大ヒープでは、一番大きな値をもつ要素の優先度が一番高くなります。一方、最小ヒープでは、一番小さな値をもつ要素の優先度が一番高くなります。本節では、整数値を要素としてヒープに格納し、値そのものを優先度とします。

ヒープで用いる木構造は、**完全二分木 (complete binary tree)** と呼ばれる二分木を用いるため、配列で表現することができます。さらに、二分木は構造化されているので、ヒープ内の要素を昇順にソートすることができます。ヒープを用いて要素をソートするアルゴリズムを**ヒープソート (heap sort)** と呼びます。

また、他にも**フィボナッチヒープ (Fibonacci heap)** といったヒープ構造がありますが、一般的にヒープと言うと上記で説明した二分木を用いたヒープを指します。

データ構造でいうヒープは、プログラムを実行するときにメモリ領域 (静的領域やヒープ領域、ス택領域) などで出てくるヒープとは異なるので、注意してください。新しい専門用語がたくさん登場するので、1つひとつ解説していきます。

■ 6.4.1 ヒープが活躍する場面

ヒープは、優先度付きキューとしてさまざまな用途に使用できます。優先度付きキューとは、データの集合に含まれる各要素に優先度が付いているキューのことです。要素をデキューするときは、一番優先度が高い要素が取り出されます。思いつくだけでも、さまざまな用途が考えられます。

たとえば、ラスベガスのホテルにチェックインするときに、長い列に並ばないといけません。前に列に並んだ人が先にチェックイン手続きができるのでキューのと同じです。ここに VIP が来たら、ホテルの支配人が列の先頭に割り込んで VIP が次にチェックインできるように案内します。VIP は一般の顧客よりも優先度が高いからです。

また、ゲームなどのソフトウェアでも優先度付きキューは多用されます。たとえば、ドラゴンクエストなどのターン制の RPG ゲームでは、すばやさと呼ばれるスタートスをもとにバトルコマンドを実行できます。キャラと敵を優先度付きキューに入れ、すばやさの順番にキューからキャラ・敵をデキューしてバトルコマンドを実行する、といった処理になります。

さらに、次章の第 7.7 節で解説するダイクストラ法と呼ばれる最短経路を求めるアルゴリズム内でも優先度付きキューを用います。整数値の代わりに自分で定義したクラス（ユーザ定義クラスと呼ぶ）を要素としてヒープに格納し、優先度を定義します。次章でもヒープの仕組みを適応するので、本章でしっかりとポイントを抑えてください。

■ 6.4.2 完全二分木を用いたヒープの概要

ヒープは、完全二分木という類の木を用いるので、配列で表すことができます。そのため、以下のようにヒープを表す MyHeap クラスを定義します。クラスメンバとして変数 self.arr と self.size を定義します。引数として要素を含む配列を変数 arr で受け取り、self.arr を arr で初期化します。MyHeap 内でヒープの性質をもつように arr を変更するので、MyHeap のインスタンスを生成した時点で、引数 arr の要素の並びは気にしなくて構いません。また、self.size も 0 で初期化しておきます。

```
class MyHeap:
    def __init__(self, arr):
        self.arr = arr
        self.size = 0
```

ヒープは抽象型のデータ構造です。これまでと同様にデータ構造に対する操作を MyHeap クラス内で定義する必要があります。第 3.3 節で解説した、一般的なキューは先入れ先出しの性質をもち、データ構造に対する操作としてエンキュー（挿入）とデキュー（取得と削除）を定義しました。ヒープ、優先度付きキューは、挿入と取得・削除を含む 4 つの操作を定義します。

- 1) 取得 (extract_max) : 一番優先度が高い要素をヒープから取り出し、取り出した要素をヒープから削除する処理

- 2) **挿入 (insert)** : 優先度付きで、新しい要素をヒープに挿入する処理
- 3) **値の更新 (increase_val)** : ヒープ内の要素の優先度を更新する処理
- 4) **ヒープ化 (max_heapify)** : ヒープの性質を保つようする処理

取得は優先度が一番高い要素を取得するので、`extract_max` という名称を付けます。なお、最小ヒープの場合は `extract_min` と呼びます。挿入処理は、新しい要素を優先度付きでヒープに `insert` します。値の更新は、すでにヒープ内にある要素の優先度を更新することです。一般的に `increase_val` という名称を付けます。一般的なキューとの違いは、これらの操作をしたあとにヒープの性質を維持しなければならないことです。ヒープの性質を維持させることをヒープ化 (heapify) と呼びます。

さらにヒープは、データ構造内の要素をソートすることができるので、本書では5つ目の操作として、ヒープソートを定義します。

- 5) **ヒープソート (heap sort)** : ヒープ内の要素を昇順にソートする

また、ヒープ自体は二分木なので、各要素は親と左の子、右の子という関係があります。そのため、指定した要素の親と子のインデックスの取得 (`get_parent`、`get_left`、`get_right`) を定義する必要があります。

■ 配列を用いた完全二分木の表現

完全二分木 (complete binary tree) とは、木の高さを h とすると、木に含まれるすべての葉となる節点の高さが h または $h-1$ 、かつすべての葉を左寄せにした木のことです。ここで左寄せとは、一番下の階層にある葉の節点は左から順番に位置することを意味します。なお、木に含まれる要素数を n とすると、木の高さは $h = \lfloor \log_2 n \rfloor$ となります。たとえば、図 6.28(a) と (b) は完全二分木ですが、(c) は完全二分木ではありません。

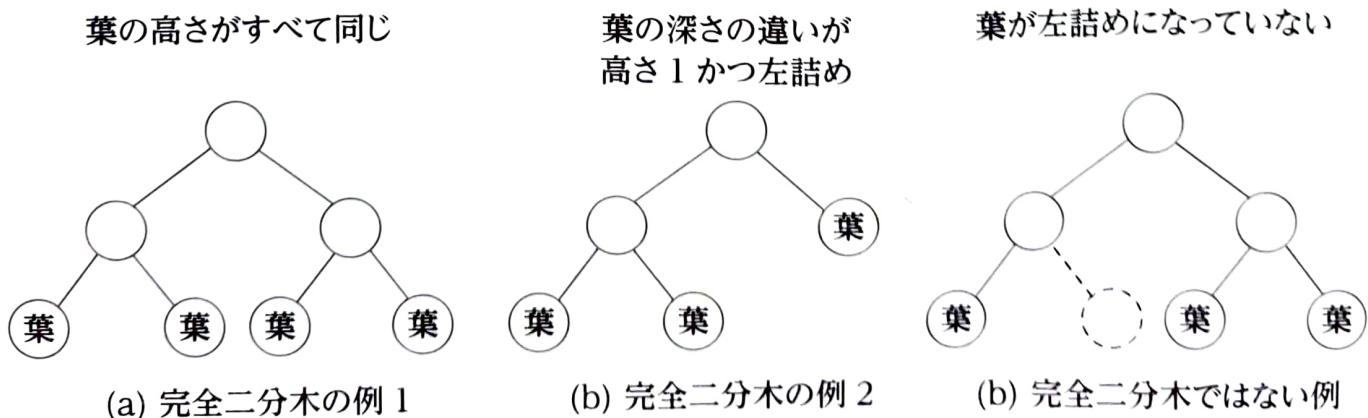


図 6.28 完全二分木とそうでない例

前節で解説した二分探索木では、場合によってはそれぞれの葉の高さが大きく異なりますが、完全二分木ではその心配がありません。

次に完全二分木と配列との関係を説明します。配列を変数 arr とし、木に含まれる要素数を変数 n とします。木に含まれる要素は、 $\text{arr}[0]$ から $\text{arr}[n-1]$ に 1 つずつ格納されます。根となる要素が $\text{arr}[0]$ 、根の左の子が $\text{arr}[1]$ 、根の右の子が $\text{arr}[2]$ に対応します。一般化すると、 $\text{arr}[\text{index}]$ に格納されている要素の左の子は $\text{arr}[2 \times (\text{index}+1)-1]$ 、右の子は $\text{arr}[2 \times (\text{index}+1)]$ に格納されます。インデックスが $n-1$ を超える場合は、 $\text{arr}[\text{index}]$ の子はいません。また、 $\text{arr}[\text{index}]$ に格納されている要素の親は $\lfloor (\text{index}+1)/2 \rfloor -1$ となります。当然、 $\text{arr}[0]$ は根なので、親はいません。

図解すると一目瞭然なので、例を図 6.29 と図 6.30 に示します。図 6.29 に示す完全二分木を配列に変換すると図 6.30 のようになります。根から階層順に、かつ左から各節点を配列に対応させると、前述の要領で木の要素が配列に格納されていることがわかります。

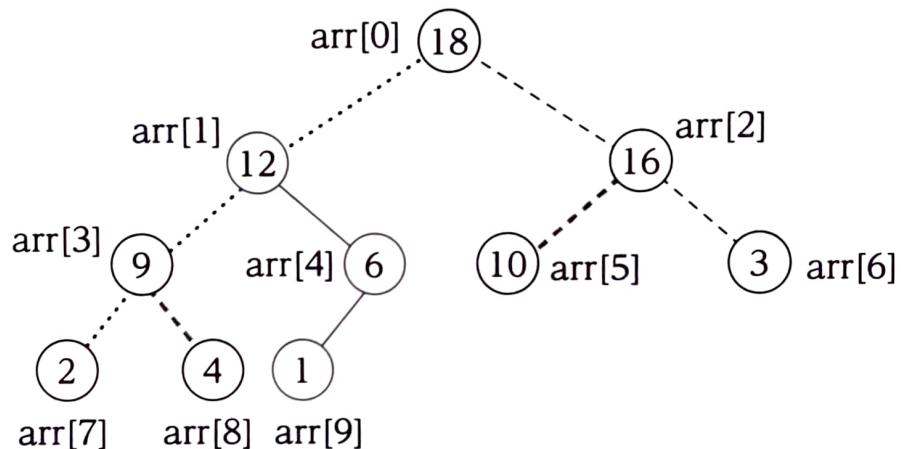


図 6.29 ヒープの例

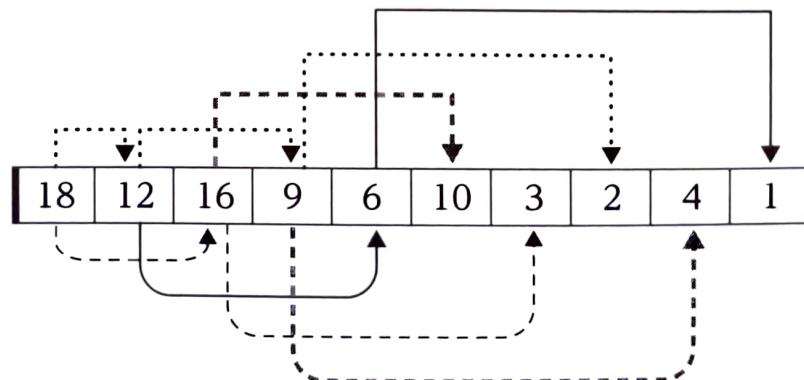


図 6.30 図 6.29 に示す二分木を配列で表した状態

たとえば、 $\text{arr}[2]$ の節点の左の子は、 $2 \times (2+1)-1=5$ なので、 $\text{arr}[5]$ に格納されている節点 10 です。左の子は $2 \times (2+1)=6$ なので、 $\text{arr}[6]$ に格納されている節点 3 です。 $\lfloor (2+1)/2 \rfloor -1=0$ なので、親は $\text{arr}[0]$ に格納されている節点 0 です。

ヒープの実装は配列ですが、仕組みは完全二分木を見ながらのほうが理解しやすいです。

■ ヒープ化 (ヒープの性質の維持)

ヒープ化 (heapify) の処理はヒープ構造で最も実行頻度の高い処理です。なぜなら、要素の取得または挿入をするたびにヒープ化を実行して、ヒープの性質（最大ヒープの場合は親がもつ値は子がもつ値より大きいか等しい）を維持する必要があるからです。

最大ヒープにするメソッドを max_heapify と呼びます。ヒープ化を行う条件として、指定した節点の左の子を根とした部分木と右の子を根とした部分木がすでにヒープになっていることです。なお、中身がランダムな配列をヒープ化する場合は、ゼロからヒープを構成する処理 (create_max_heap と名付ける) となるため、別途解説します。

図 6.31 を見てください。節点 6 を根とした完全二分木があります。対応する配列 arr は [6, 18, 16, 9, 12] です。節点 6 の左と右の子がもつ値がそれぞれ 18 と 12 なので、ヒープの性質が満たされていません。ただし、節点 6 の左の子を根とした部分木と右の子を根とした部分木はすでにヒープになっています。ここで節点 6 を指定してヒープ化を実行します。

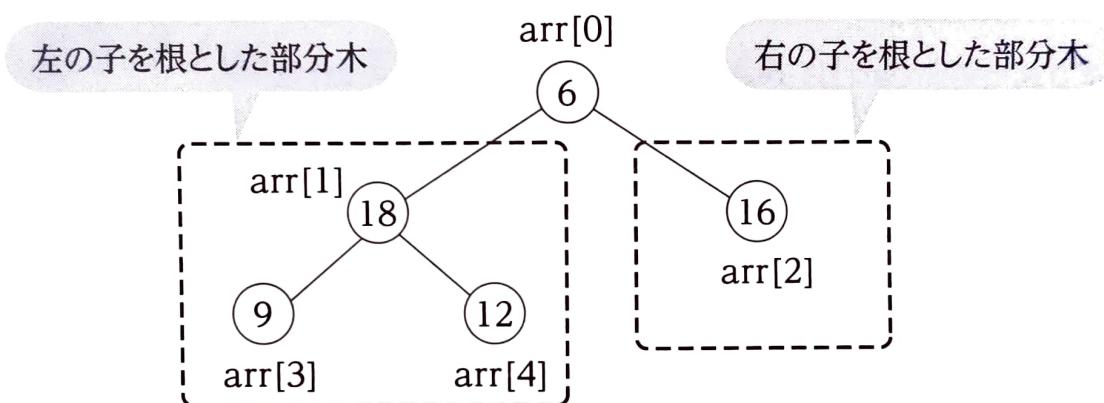


図 6.31 ヒープ化の例 1 この図はヒープの性質が満たされていない

ルールは単純です。当該節点と左の子と右の子の中で一番大きな値をもつ節点を親にするだけです。すでに当該節点が子よりも大きな値をもっていれば、入れ替えは行いません。上記の図の場合では、節点 6 と 18 と 12 の中で一番大きな値は 18 なので、節点 6 と 18 を受け入れます。そのあと、この状態を図 6.32 に示します。根となる節点 18 は左の子と右の子がもつ値より大きい状態です。

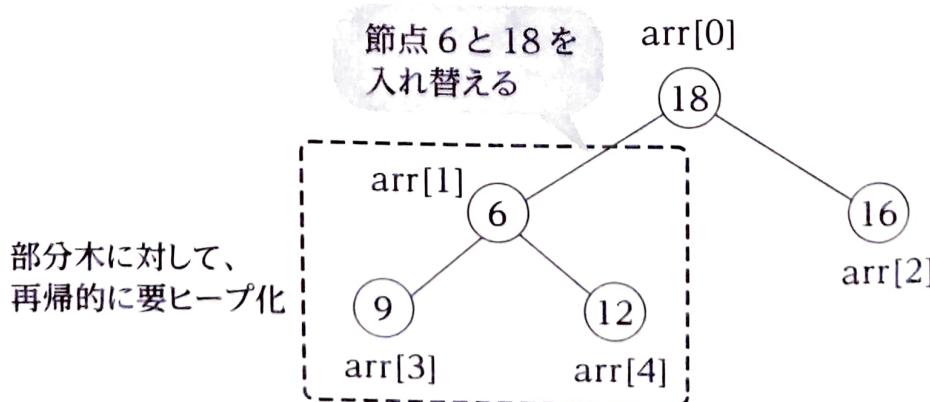


図 6.32 ヒープ化の例 2 左の部分木はヒープが満たされていない

入れ替えを行っていない右の子を根とした部分木はこれ以上の処理は必要ありません。すでに節点 16 より下の階層はヒープになっているからです。入れ替えを行った左の子を根とした部分木は、節点を移動させたのでヒープの性質が満たされていません。そこで、図 6.32 に示す節点 6 を根とした部分木に対して、ヒープ化を行います。同様の処理を木の高さ分だけ再帰的に繰り返します。再度ヒープ化を行った後の状態を図 6.33 に示します。木全体がヒープの性質を満たしていることが確認できます。

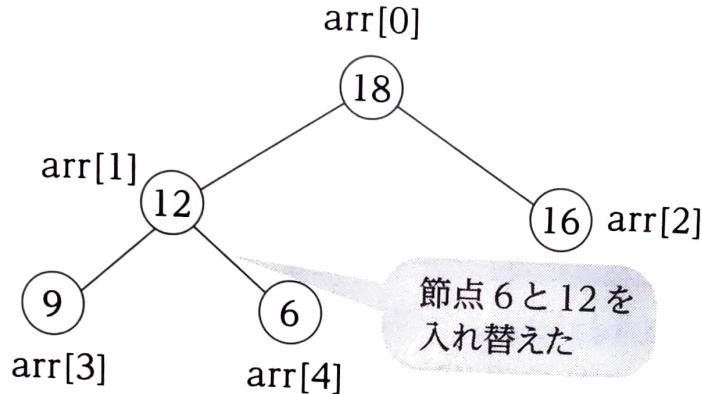


図 6.33 ヒープ化の例 3 ヒープが満たされた

また、要素数を n としたときのヒープ化にかかる時間は木の高さに依存します。すなわち、対数時間でヒープ化が可能なので、計算量は、 $O(\log n)$ です。

■ ランダムな配列からのヒープ構成

中身がランダムな配列をゼロからヒープするメソッドを `create_max_heap` と名付けます。この場合は、木の下の階層から最大ヒープ化 (`max_heapify` メソッド) を繰り返して適応します。

図 6.34 に例を示します。配列で表すと $[1, 3, 4, 6, 9, 10, 12, 16, 18]$ となり、ヒープの性質がまったく満たされていない状態です。葉となる節点は無視して、その 1 つ上の階層の節点に注目してください。節点 6 を根とした部分木と節点 4 を根とした部分木です。節点 6 の子は双方とも葉なので、左の子を根とした部分木と右の子を根とした部分木はすでにヒープ化している、と見なせます。そこで節点 6 を根とした部分木に対してヒープ化を実行します。節点 4 を根とした部分木に対しても同様のことが言えるので、ヒープ化を行います。

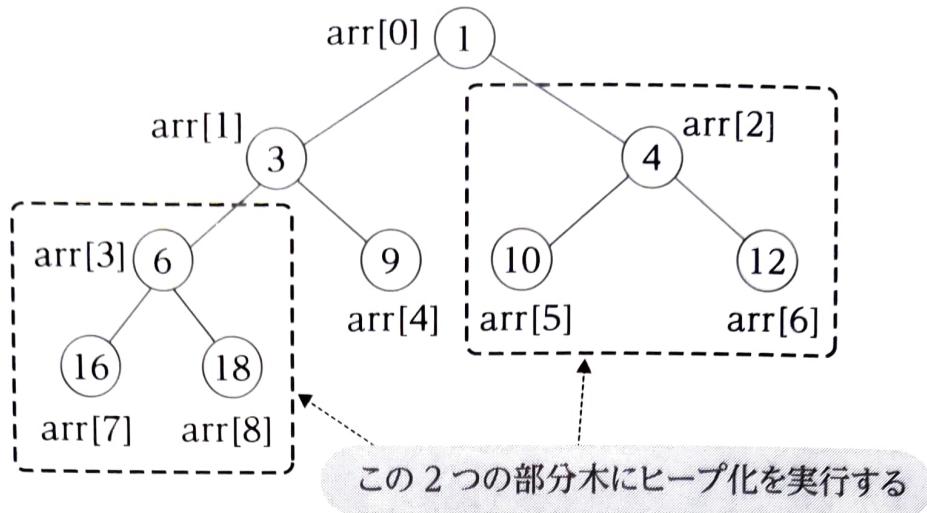


図 6.34 ランダムな配列をヒープ化する例 1

節点 6 と節点 4 を根とした部分木に対してヒープ化を行ったときの状態を図 6.35 に示します。節点 6 と 18 が入れ替わり、節点 4 と 12 が入れ替わっています。次は節点 3 を根とした部分木に注目してください。左の子である節点 18 を根とした部分木はすでにヒープになっています。右の子は葉なので、こちらもヒープになっています。そこで節点 3 を根とした部分木にヒープ化を実行します。

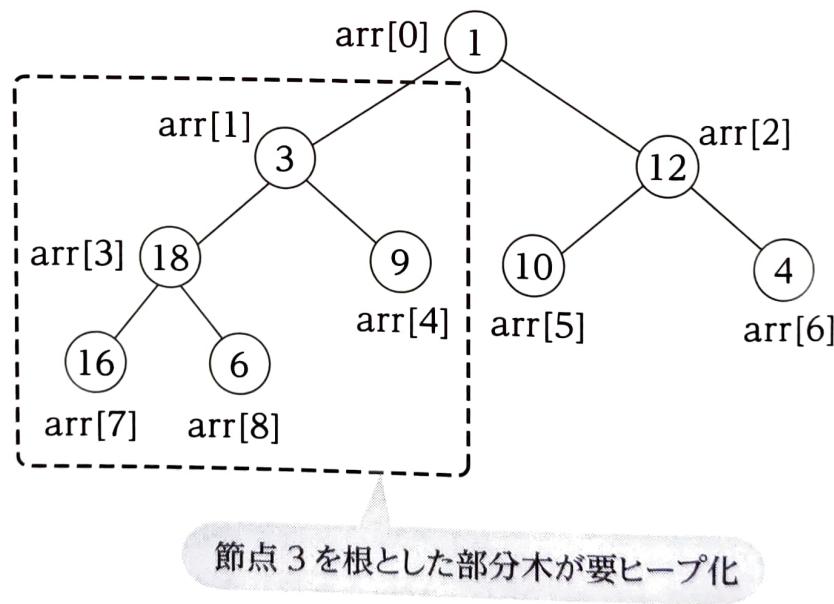
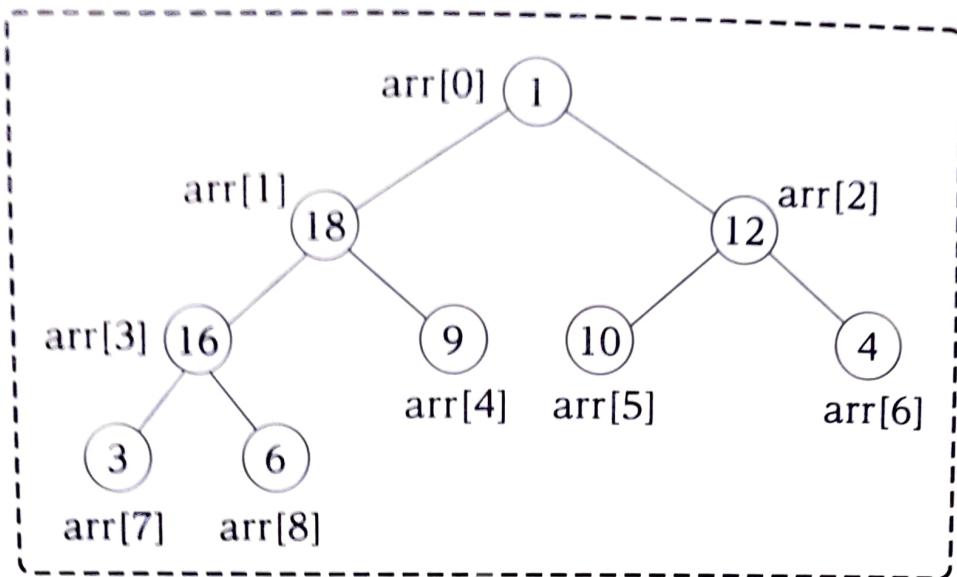


図 6.35 ランダムな配列をヒープ化する例 2

ヒープ化後の状態を図 6.36 に示します。節点 3 と 18 が入れ替わり、その処理の中で再帰的に呼び出されたヒープ化によって節点 3 と 16 が入れ替わります。見てのとおり、節点 18 を根とした部分木がヒープの性質を満たしている状態になります。



節点 1 を根とした木が要ヒープ化

図 6.36 ランダムな配列をヒープ化する例 3

最後に木の根である節点 1 に対してヒープ化を行います。図から確認できるように、左の子と右の子を根とした部分木はすでにヒープ化されています。これまでと同様にヒープ化を行うと図 6.37 に示す状態になります。木全体がヒープの性質を満たしていることが確認できます。

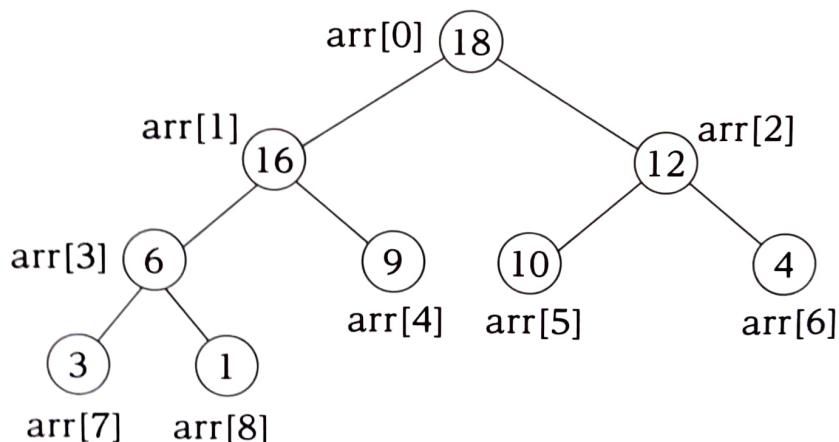


図 6.37 ランダムな配列をヒープ化する例 4 (木全体がヒープの性質を満たしている)

実際の処理は配列に対して行います。要素の約半分が葉となるので、配列 arr の前半分に対してヒープ化を繰り返すことによってランダムな配列をヒープ化できます。具体的な処理はソースコードを用いて解説します。

ランダムに並んだ配列のヒープ化の計算量は $O(n \log n)$ です。ヒープ内の約半分の要素 ($\frac{n}{2}$) に対して、ヒープ化を実行します。ヒープ化の計算量が $O(\log n)$ なので、合計で $O(n \log n)$ となります。

■ 要素の取得と削除（キューなので取得後に削除も実行する）

ヒープは優先度付きキューとして使用できます。再度、図 6.29 の木構造を見てください。この完全二分木では、各要素がもつ値はその子がもつ値よりも大きいので、ヒープの性質を満たしています。ここで優先度を各要素がもつ値と定義すると、一番大きな値をもつ要素が木の根に位置します。そして木の根は、完全二分木を配列で表したときに先頭要素である $\text{arr}[0]$ に位置します。

そのため、要素の取得 (extract_max) は、単純に $\text{arr}[0]$ にある要素にアクセスすることです。ただし、**キューの一種なので、取り出した要素はデータ構造内から削除する必要があります**。そのために、 arr の最後尾のインデックスにある要素をいったん $\text{arr}[0]$ に移動させます。この時点でヒープの性質を失うため、ここで前述のヒープ化を行います。そうすると要素が 1 つ減った状態でヒープが再構成されます。

例を図 6.38 に示します。対応する配列 arr は $[18, 12, 16, 9, 6]$ となります。木の根 ($\text{arr}[0]$) に格納されている整数値 18 が一番優先度が高い要素なので、取り出します。最後尾の要素である $\text{arr}[4]$ の整数値 6 を $\text{arr}[0]$ に移動させて、 $\text{arr}[4]$ は削除します。

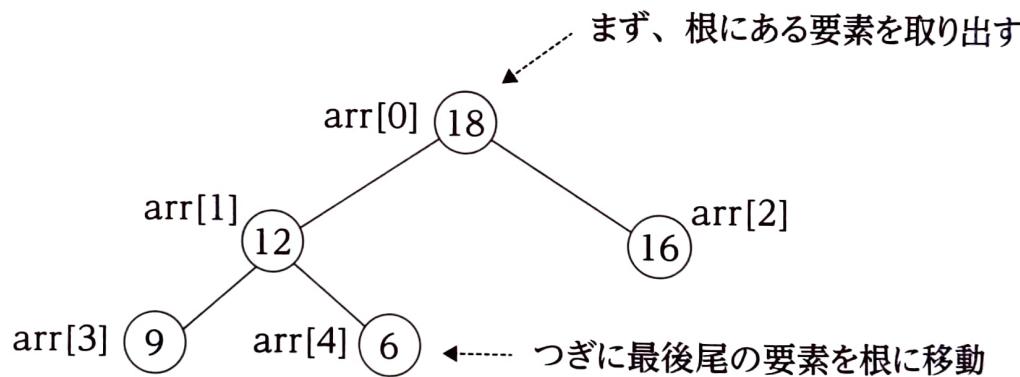


図 6.38 ヒープから要素の取得をするときの処理

要素を取り出したあとのヒープは図 6.39 に示す状態になります。配列で表すと $[6, 12, 16, 9]$ です。図に示す木のとおり、整数値の 6 が木の根に位置しており、ヒープの性質が保たれていません。ここで節点 6 を根とした木に対してヒープ化を実行します。

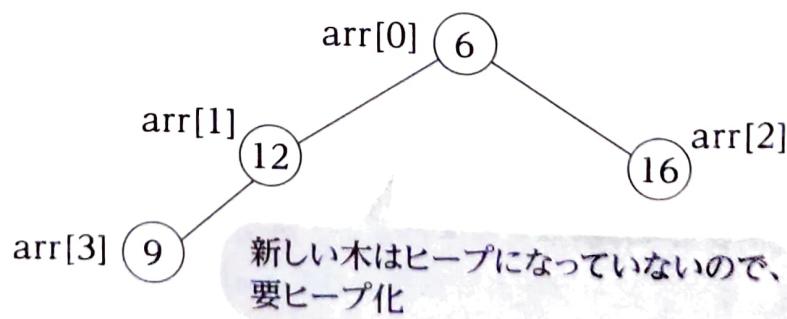


図 6.39 ヒープから節点 18 を取り出し、最後尾の節点 6 を根に移動させたときの状態

ヒープ処理は前述のとおりです。配列で表現するとヒープは、`[16, 12, 6, 9]`となります。

要素の取得に必要な計算量は $O(\log n)$ です。要素の取得自体は $O(1)$ ですが、取り出した要素を削除したあとにヒープ化を行う必要があります。そのため、 $O(\log n)$ の時間がかかります。

■ 優先度の更新

優先度の更新処理では、各要素がもつ値（優先度）を増加させて、ヒープ化を行います。値を減少させるときも似たような処理になります。本書では優先度を増加させる処理のみを解説します。

値を増加させるとヒープの性質が満たされなくなる可能性があります。そのため、優先度を増加させた後に、ヒープの性質を維持する処理をしなければなりません。子より親のほうが大きな値をもつようにするだけなので、処理内容は非常に単純です。木構造内の値を更新した節点の親の値と比較し、更新した値が親の値よりも大きければ、上の階層に移動させるだけです。これを繰り返します。

例を図 6.40 に示します。図に示す木はヒープの性質が満たされている状態です。ここで、葉である節点 1 の値を 17 に変更したいとします。値を変更するとヒープの性質が満たされなくなります。

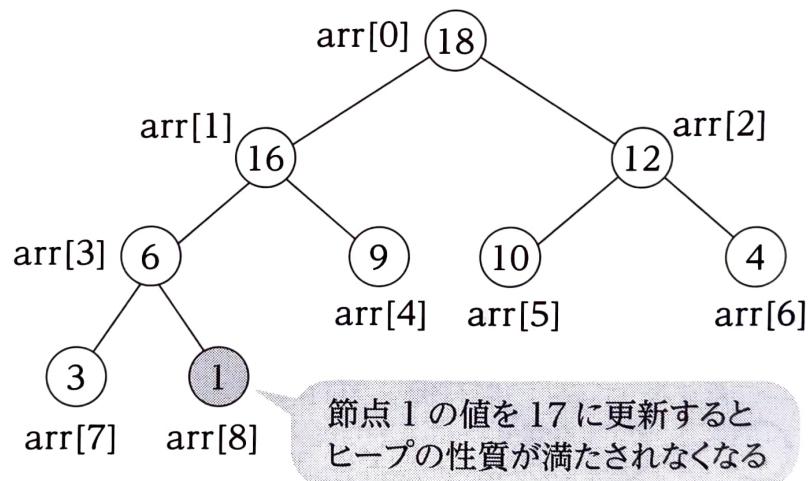


図 6.40 優先度を変更する例 1

節点 1 の値を 17 に変更するので、以降は節点 17 と呼びます。図 6.41 に示すとおり、値を増加させた節点から開始して、各節点の親がもつ値と比較していきます。 $17 > 6$ なので、節点 6 と 17 を入れ替えます。次に、その上の階層にある節点の値と比較すると、 $17 > 16$ なので、節点 16 と 17 を入れ替えます。最後に木の根である節点 18 と比較しますが、 $17 < 18$ なので入れ替えは行いません。

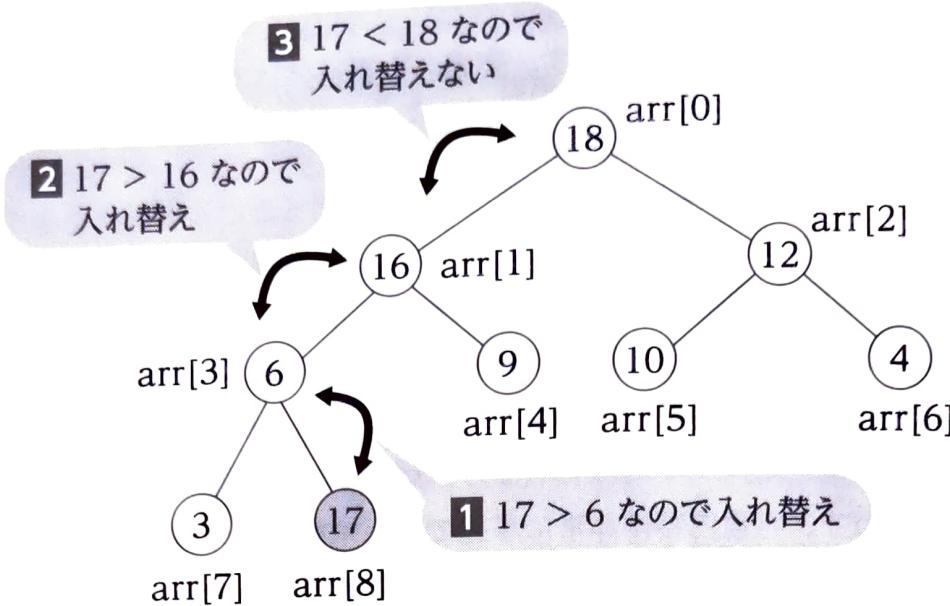


図 6.41 優先度を変更する例 2

ヒープ化したあとの木の状態を図 6.42 に示します。優先度を更新した節点 17 は、木の根である節点 18 の左の子になります。ヒープの性質が満たされていることが確認できます。

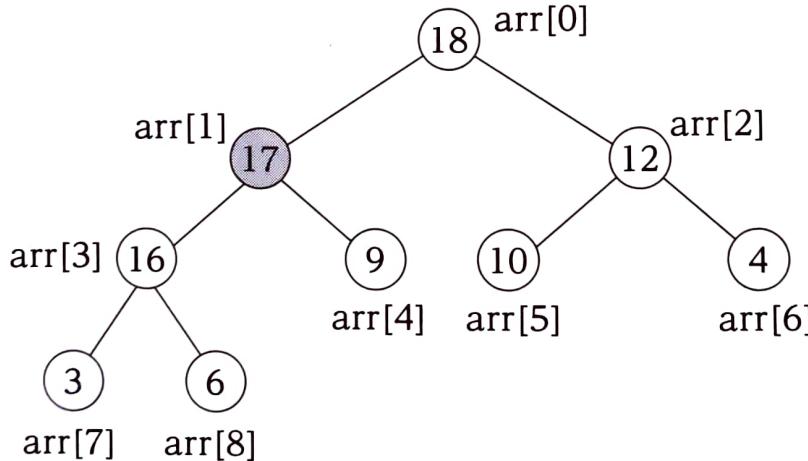


図 6.42 優先度の変更とヒープ化を行ったときの状態

優先度の更新にかかる計算量は $O(\log n)$ です。値を更新する要素に対応する節点が、どの階層に属するである節点かによって実際の計算量が異なります。最悪ケースでは、葉となる節点の値を更新することになります。この場合、計算量が木の高さに依存するため、対数時間となります。

■ ヒープへの要素の挿入

ヒープへの新しい要素の挿入は優先度の更新に少し手を加えるだけです。新しい要素がもつ値を val とします。

最大ヒープでは、各節点は子より大きな値をもちます。なので、新しい要素の値を $-\infty$ (マイナス無限大)にして、配列arrの最後尾に格納します。そして、新しく挿入した要素の値を本来の値であるvalに更新します。すなわち、arr最後尾の要素の値を $-\infty$ からvalに増加させた、という処理と同じです。

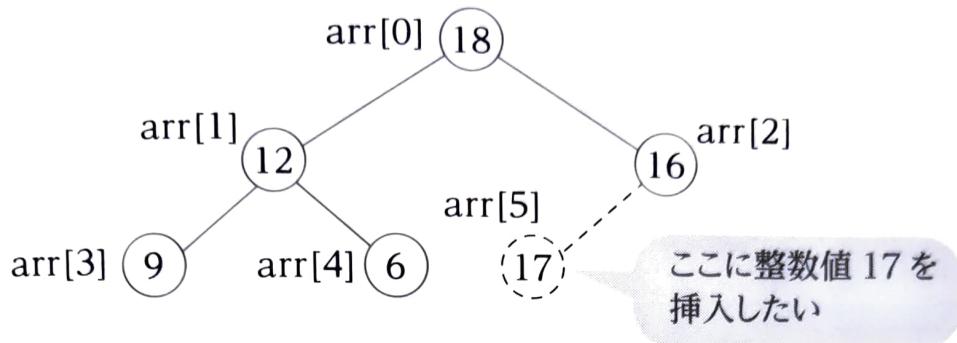


図 6.43 ヒープに新たな要素を挿入する例 1

たとえば、図 6.43 に示すヒープに整数値の 17 を新しい要素として挿入したいとします。ヒープに対応する配列 arr の中身は [18, 12, 16, 9, 6] です。図内では点線で表している節点と枝を arr[5] の箇所に追加します。

新しい要素の値を $-\infty$ と設定して、arr[5] の箇所に新たな節点を加えた木を図 6.44 に示します。新しく挿入した要素は葉となり、かつ優先度が $-\infty$ なので、どの節点よりも値が小さくなります。そのため、ヒープの性質が保たれています。

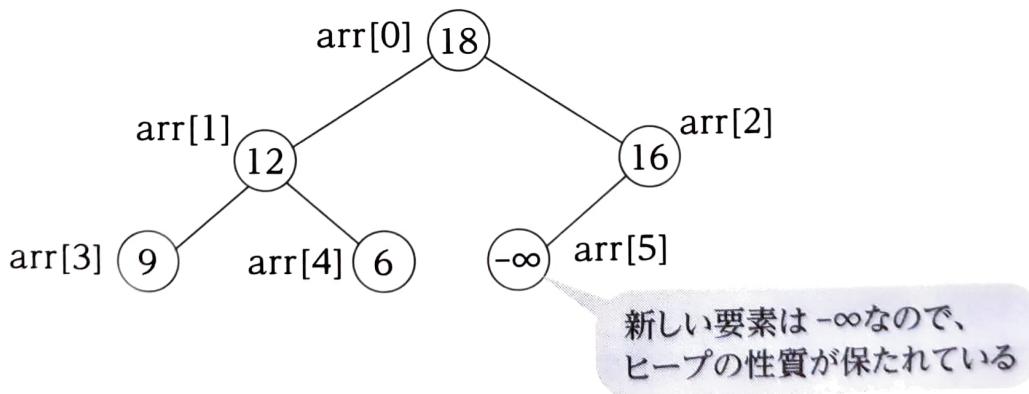


図 6.44 ヒープに新たな要素を挿入する例 2

ここで、arr[5] の値を 17 に変更する、といった処理を行います。処理内容の概要は優先度の更新で説明したとおりです。

なお、要素の挿入処理に必要な計算量は、木の高さに依存するため、 $O(\log n)$ となります。

■ ヒープソート

ヒープソート(heap sort)はヒープに格納された要素を $O(n \log n)$ の計算量で昇順にソートする

アルゴリズムです。本書では最大ヒープの要素を昇順に並べ替える方法を説明します。

図 6.45 にソート前のヒープの中身を示します。対応する配列 arr は [18, 12, 16, 9, 6] となり、ヒープの性質が満たされています。

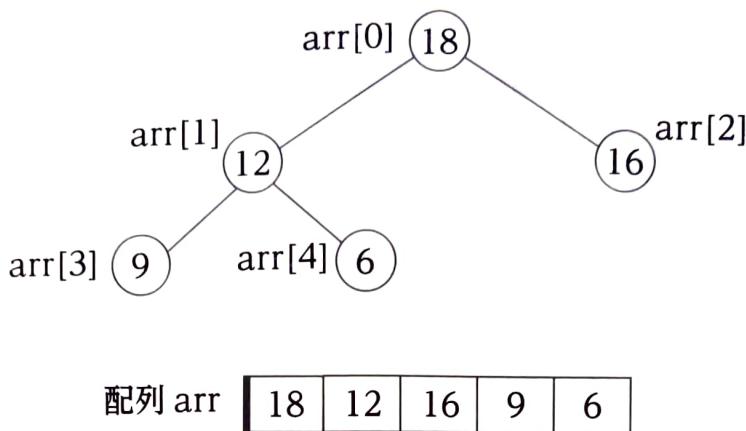


図 6.45 ソート前のヒープ

これをヒープソートで並び替えて、**図 6.46** に示す状態にします。

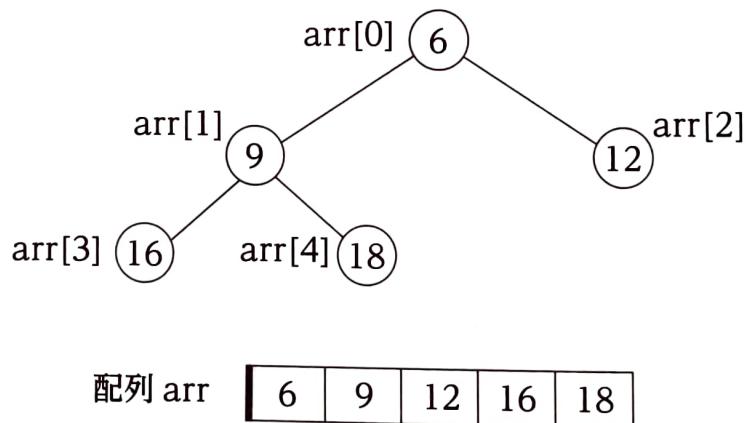


図 6.46 昇順にソート後のヒープ

ヒープ構造では、木の根（配列の先頭要素）が一番大きな値（優先度）をもちます。そのため、配列の先頭要素 arr[0] と最後尾の要素 arr[n-1] を交換し、部分配列 arr[0:n-2] に対してヒープ化を行います。同様の処理を繰り返すと配列が昇順にソートされます。以下に骨格を示します。

```
for i in range(n - 1, 0, -1):
    arr[0] と arr[n - 1] の値を交換
    n = n - 1
    arrをヒープ化
```

図 6.45 に示すように、木の根 (arr[0]) に位置する節点 18 の値は一番大きな値です。そのため、

ソート後は、必ず最後尾の $\text{arr}[4]$ の場所に移動します。移動後、部分配列 $\text{arr}[4:4]$ はソート済みになります。節点 18 ($\text{arr}[0]$) と節点 6 ($\text{arr}[4]$) の要素を交換し、配列の大きさを 1 減らしたときの状態を図 6.47 に示します。

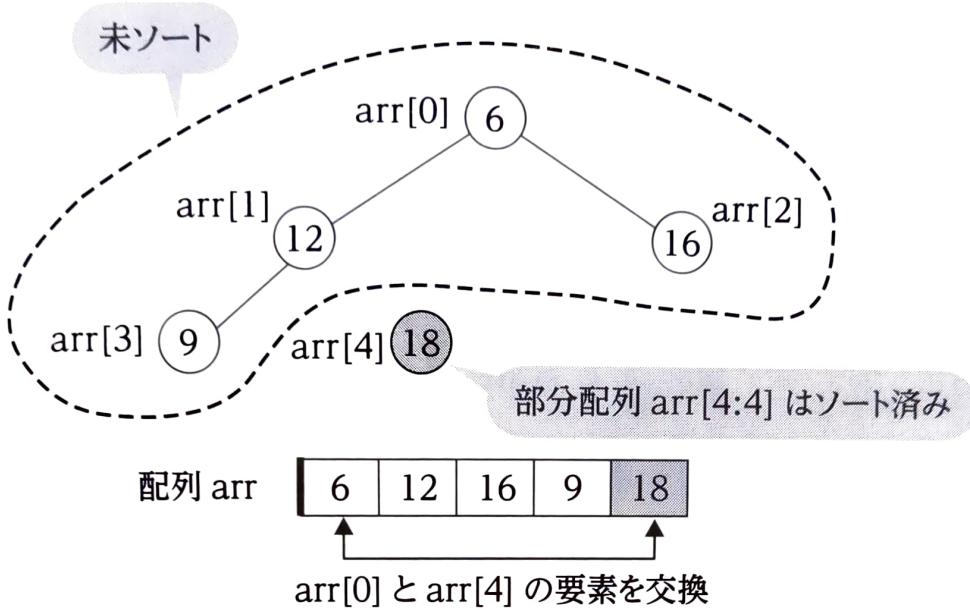


図 6.47 ヒープソートの概要 1

節点 18 はソート済みになり、その他の節点から構成される部分木は未ソートのままでです。節点を入れ替えたことで、図 6.47 内の点線で囲んだ部分木はヒープの性質を失っています。そのため、この部分木 ($\text{arr}[0:3]$) に対してヒープ化を行います。上記の骨格となる擬似コードの 4 行目の命令に相当します。

ヒープ化を行った後の木の状態を図 6.48 に示します。節点 16 と節点 6 の場所が入れ替わっていることが確認できます。また、木の根である節点 16 が一番大きな値をもっていることが確認できます。

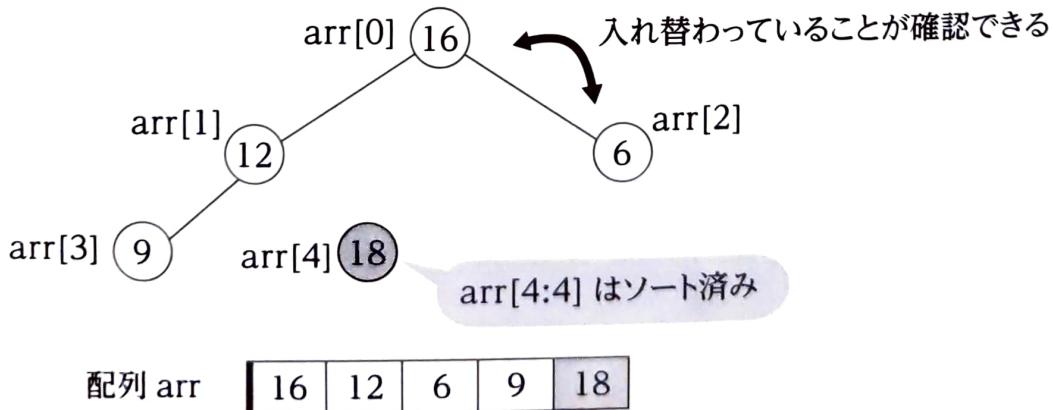


図 6.48 ヒープソートの概要 2 未ソートの部分木をヒープ化

骨格となる擬似コードの for ループの先頭に戻り、同様の処理を繰り返します。**1 木の根** (arr[0]) と最後尾の葉 (arr[3]) の要素を交換します。この時点で部分配列 $\text{arr}[3:4]$ の中身が

[16, 18]となりソートされた状態になります。このときの状態を図 6.49 に示します。

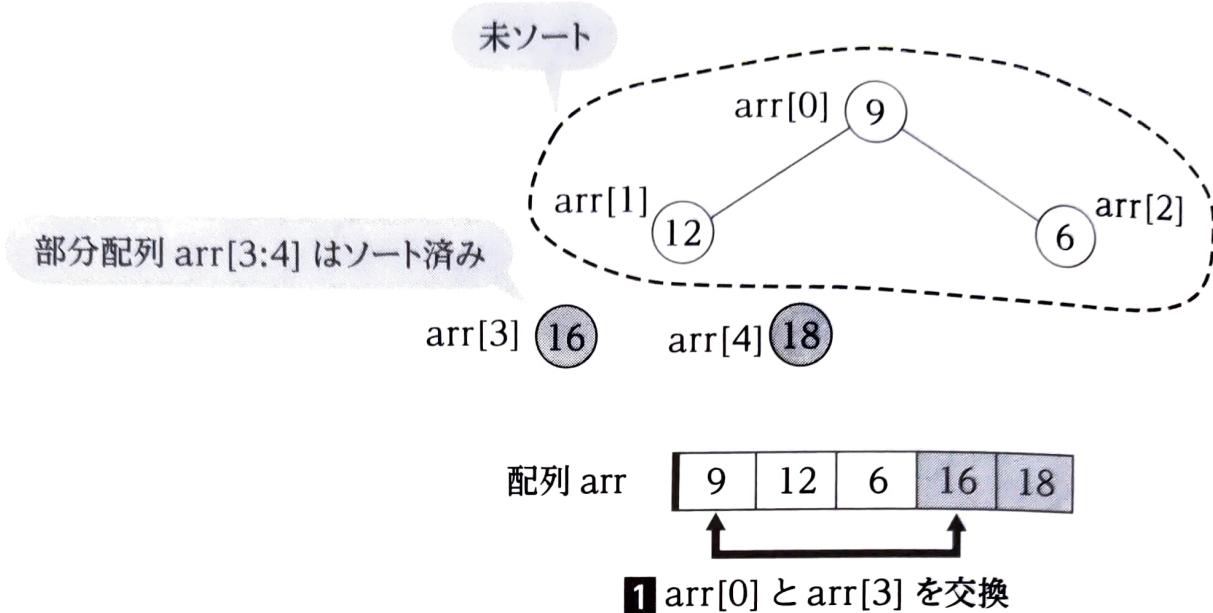


図 6.49 ヒープソートの概要 3

図 6.49 の点線で囲んだ部分配列 arr[0:2] に対応する部分木に関しては、また、ヒープの性質が失われたので、再度ヒープ化します。同様の処理を繰り返すと、最初の方で示した図 6.46 のとおり、配列全体がソートされます。

ヒープソートの計算量に関しては、骨格の擬似コードで示したとおり、for ループで $O(n)$ の時間がかかり、ループ内のヒープ化で $O(\log n)$ かかります。そのため、計算量が $O(n \log n)$ となります。

6.4.3 ヒープの実装例

ソースコード 6.3 (my_heap.py) にヒープの実装例を示します。大きな値をもつ要素が木構造の上の階層に位置する最大ヒープを実装したプログラムです。

ソースコード 6.3 ヒープの実装

~/ohm/ch6/my_heap.py

ソースコードの概要

7 行目～10 行目	配列内の 2 つの要素を交換する swap 関数の定義
12 行目～101 行目	MyHeap クラスの定義
19 行目～33 行目	最大ヒープ化を行う max_heapify メソッドの定義
36 行目～39 行目	最大ヒープを生成する create_max_heap メソッドの定義
42 行目と 43 行目	配列 arr について、arr[index] の親のインデックスを返す get_parent メソッドの定義
46 行目と 47 行目	配列 arr について、arr[index] の左の子のインデックスを返す get_left メソッドの定義
50 行目と 51 行目	配列 arr について arr[index] の右の子のインデックスを返す get_right メソッドの定義

```
120 print("ソート後のヒープ: ", my_heap.to_string())
```

ソースコード内で floor 関数を使用するため、1 行目で **math モジュール**をインポートします。4 行目は無限大の定義です。また、7 行目～10 行目の swap 関数は、arr[i] と arr[j] の値を入れ替えます。42 行目～51 行目で定義した get_parent メソッドと get_left メソッド、get_right メソッドは、前節で解説したとおりの内容なので解説は省略します。

MyHeap クラスのコンストラクタは 13 行目～16 行目で定義しています。ヒープを表す完全二分木に対応する配列は、クラスメンバ self.arr に格納されます。self.size はヒープ内の要素数を表します。多くの場合、配列の大きさである len(arr) で代用できますが、heap_sort メソッド内ではヒープ内の未ソートである部分木の大きさを参照するために必要なので、このように self.size を定義しました。また、16 行目の self.create_max_heap() という命令で、配列 arr をヒープ化します。

■ main 関数の説明

103 行目～120 行目で main 関数を定義しています。105 行目で変数 arr を宣言して、[6, 3, 18, 8, 1, 13] で初期化します。ランダムな配列になっています。106 行目で変数 my_heap を宣言し、MyHeap クラスのインスタンスを生成します。変数として配列 arr を渡します。MyHeap クラスのコンストラクタ内の 16 行目で、create_max_heap メソッドが呼び出され、自動的にヒープ化されます。107 行目で、my_heap.to_string() が出力する文字列を print 関数で表示します。ここでは my_heap がもつ配列 arr の中身が表示されます。ヒープ化されているため、配列の並びが [18, 8, 13, 3, 1, 6] となります。具体的にどのような過程でそうなったのかについては、create_max_heap メソッドの説明の箇所で説明します。

110 行目では extract_max メソッドで優先度が一番高い要素を取り出し変数 x に格納します。同時に取り出した要素はヒープから削除されます。111 行目と 112 行目で、変数 x の値を表示するとともに要素削除後のヒープの中身を表示します。整数値 18 が取り出され、ヒープの中身は [13, 8, 6, 3, 1] となります。

115 行目は insert メソッドで、整数値 10 をもつ要素をヒープに追加します。116 行目で要素追加後のヒープの中身を表示します。整数値 10 をヒープへ挿入してヒープ化した場合、配列 arr は [13, 8, 10, 3, 1, 6] となります。

119 行目で heap_sort メソッドを実行します。ヒープ内の配列 arr が昇順に並び替えられるので、120 行目の命令で [1, 3, 6, 8, 10, 13] と表示されるはずです。

■ max_heapify メソッド（最大ヒープ化）の説明

19 行目～33 行目で最大ヒープ化を行う max_heapify メソッドを定義しています。引数として、ヒープ化を実行する部分木の根に対応する配列のインデックスを変数 index で受け取ります。21 行目～28 行目は節点 index と節点 left と節点 right の中で一番大きな値をもつ節点を調べています。

そして、31行目～33行目で一番大きな値をもつ節点と節点 index の値を交換します。

まず、21行目で変数 left に self.arr [index] の左の子のインデックスを get_left メソッドで調べます。同様に右の子のインデックスは 22 行目の変数 right に格納します。23 行目の if 文の条件判定式に $left < self.size$ という式が含まれています。配列のインデックスの範囲は $0 \sim self.size - 1$ です。もし index が子をもたなければ、left の値が self.size 以上になります。左の子をもつかどうかを判定し、True であれば $self.arr [left] > self.arr [index]$ を判定します。True であれば、24 行目で変数 largest に left を代入します。False であれば、else ブロック内の 26 行目で、largest に index の値を格納します。

27 行目の if 文の条件判定式も同様です。self.arr [index] が右の子をもつかどうかを調べ、self.arr [index] と self.arr [left] の大きい方の値と self.arr [right] を比べます。True であれば、self.arr [right] が一番大きな値をもつので、28 行目で largest の値を right にします。

31 行目の if 文では largest と index の値が異なるかどうかを確認します。False であれば、節点の移動（配列の要素の交換）が生じないので、これでヒープ化の処理は終了です。True であれば、節点 index と節点 largest (節点 left と right のうち値が大きい方) の交換を行います。実際には配列に対して操作を行うので、self.arr [index] と self.arr [largest] の値を交換します。このために 32 行目の命令で swap 関数を用います。

max_heapify メソッドは、create_max_heap メソッドと extract_max メソッド内で使用されます。処理内容は前項のヒープの概要で解説したとおりなので、具体例は次の create_max_heap メソッドの具体例で解説します。

■ create_max_heap メソッド（最大ヒープの生成）の説明

36 行目～39 行目で最大ヒープを生成する create_max_heap メソッドを定義しています。前項の説明では、二分木の葉の 1 つ上の階層の節点から順にヒープ化を行う、と説明しました。self.size の値を n とした場合、配列 arr のインデックスで表すと $\lfloor n/2 \rfloor - 1$ になります。すなわち、arr [0: $\lfloor n/2 \rfloor - 1$] に含まれるすべての要素の節点に対して下の階層から max_heapify メソッドを実行します。

まず、37 行目の命令で、クラスメンバ self.size に配列の大きさである len(self.arr) の値を代入します。38 行目の for ループのループカウンタの範囲を range(math.floor(self.size/2) - 1, -1, -1) としています。self.arr [math.floor(self.size/2) - 1] から self.arr [0] までの配列のインデックスに対して、39 行目の max_heapify (i) を実行します。

main 関数内の 105 行目で生成した配列 arr である [6, 3, 18, 8, 1, 13] からヒープを生成する具体例を示します。MyHeap クラスのコンストラクタ内で、create_max_heap を生成する前の配列 arr を二分木で表すと **図 6.50** のようになります。38 行目の for ループ開始前の状態です。

create_max_heap メソッドを呼び出し、37 行目の命令で self.size の値を配列の大きさ（この例の場合は 6）に設定し、38 行目の for ループへ進みます。ループカウンタ i は 2 から 0 の整数値となります。図のとおり、arr [3:5] の節点は二分木の葉になっています。この箇所はすでにヒープに

なっているので飛ばします。部分配列 $\text{arr}[0:2]$ に対応する各節点に対して `max_heapify` メソッドを実行します。

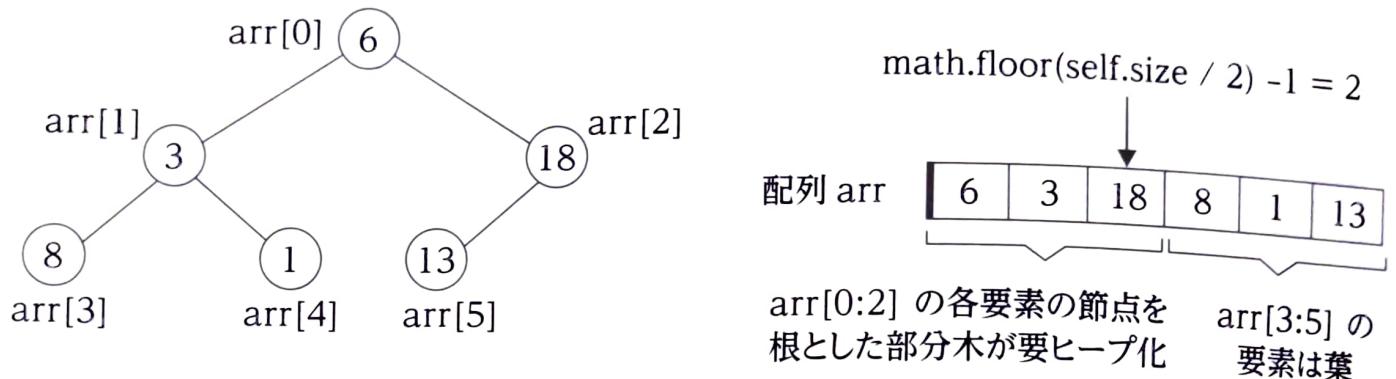


図 6.50 最大ヒープの生成の例 1 `create_max_heap()` 実行 for ループ開始前の状態

38 行目から始まる for ループの 1 回目のイテレーションが終了したときの状態を図 6.51 に示します。1 回目のループでは 39 行目の `max_heapify(2)` を実行し、19 行目のメソッドへ実行制御が移ります。`max_heapify` メソッド内の引数 `index` の値は 2 です。21 行目と 22 行目で左と右の子のインデックスを調べます。変数 `left` は 5 になり、変数 `right` は `self.size-1` より大きな値になります。右の子は存在しないので、ここでは無視します。`self.arr[index]` と `self.arr[index].left` を比較すると `self.arr[index]` のほうが大きいことがわかります。そのため、31 行目の if 文の条件判定式では、`index == largest` となります。判定結果が `False` なので、何もせずに処理を終えます。二分木において節点 18 を根とした部分木を見ても、すでにヒープの性質を満たしているので、何もしなくて良いことがわかります。図 6.50 と図 6.51 を比べても、配列 `arr` の中身に変化はありません。

1 回目のイテレーションの終了時

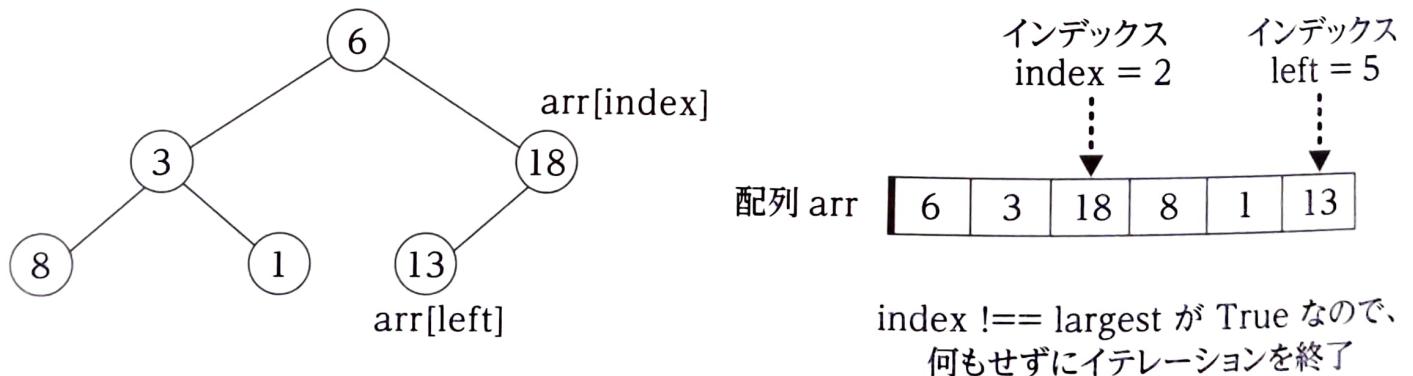


図 6.51 最大ヒープの生成の例 2

`create_max_heap` メソッドに制御が戻り、2 回目のイテレーション (`i` の値が 1) に進みます。2 回目のイテレーションが終了したときの状態を図 6.52 に示します。図 6.51 と見比べながら、どのようにヒープの中身が変わるかを説明していきます。

図の配列を見ると `index` が 1、`left` が 3、`right` が 4 です。二分木を確認すると、`self.arr[1]` と

`self.arr[3]` と `self.arr[4]` の中で一番大きな値をもつのは `self.arr[3]` に場所にある整数値 3 です。そのため、① `largest` の値が `left` の 3 になります。31 行目の `if` 文が `True` になるので、② `self.arr[index]` と `self.arr[largest]` の値を `swap` 関数で交換します。配列の要素を交換したので、33 行目の命令のとおり再帰的に `self.max_heapify(largest)` を実行します。`largest` の値が 3 なので、`self.max_heapify(3)` を実行することとなりますが、`self.arr[3]` は二分木の葉なので、すでにヒープ化されている状態です。何もせずにメソッドの処理が終了します。

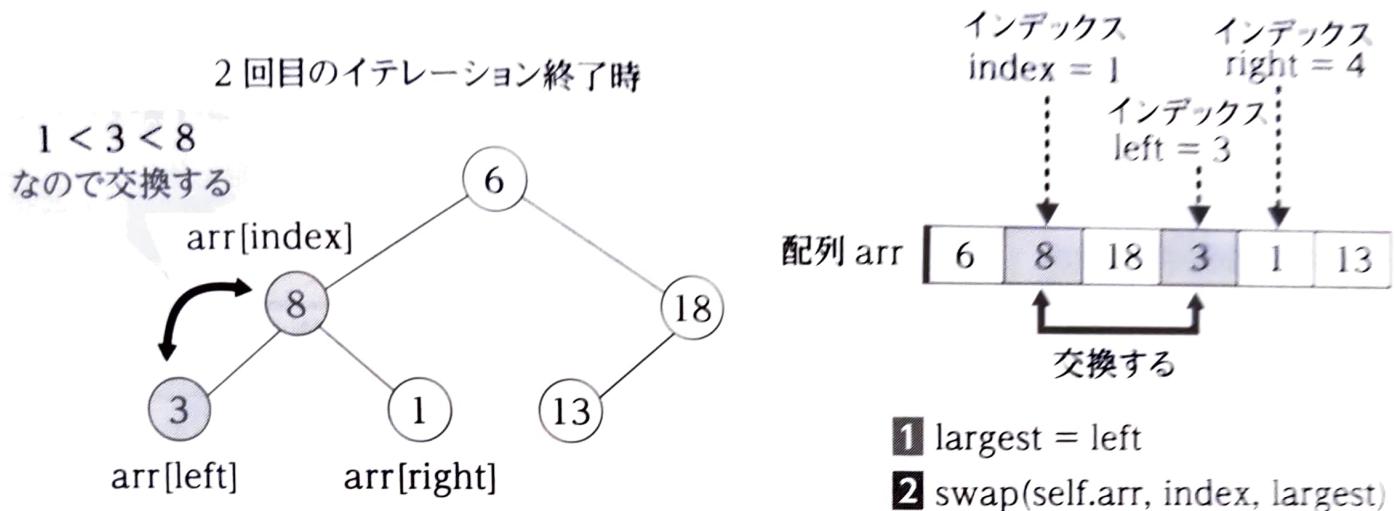


図 6.52 最大ヒープの生成の例 3

ループカウンタ `i` の値を 0 として、3 回目のイテレーションに入り、`self.max_heapify(0)` を実行します。`left` と `right` の値はそれぞれ 1 と 2 です。`self.arr[index]` と `self.arr[left]` と `self.arr[right]` の中で一番大きな値は `self.arr[right]` の 18 です。そのため、変数 `largest` に `right` の値である整数値 2 を代入して、`swap` 関数で `self.arr[index]` と `self.arr[largest]` の値を交換します。3 回目のイテレーションで 39 行目の `self.max_heapify(0)` を実行したときのヒープの状態を図 6.53 に示します。配列 `arr` の中身が更新されています。

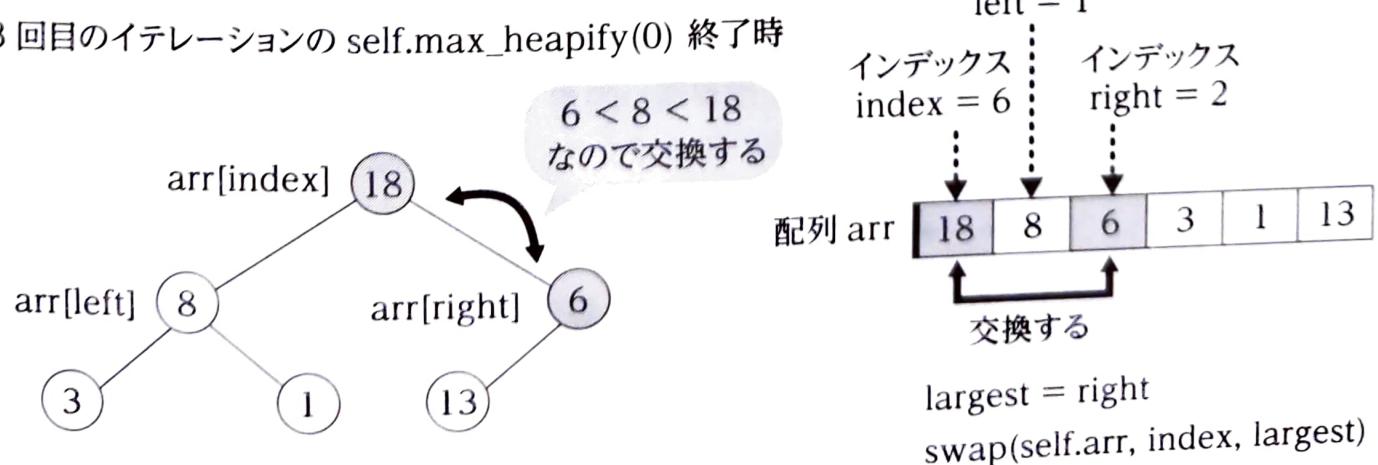


図 6.53 最大ヒープの生成の例 4

要素の交換が行われたので、再帰的に `self.max_heapify(2)` を実行します。`self.arr[2]` に対応する節点 6 を根とした部分木はヒープの性質が満たされていません。そのため、同様の処理を行います。`self.arr[index]` より `self.arr[index].left` の値のほうが大きいので、要素の交換を行います。その後のヒープの状態を図 6.54 に示します。`arr[2]` と `arr[5]` が交換されている（前記の処理で、節点 6 と 13 が入れ替わっている）ことを確認してください。

3回目のイテレーション終了時
`self.max_heapify(2)` を再帰的に呼び出す

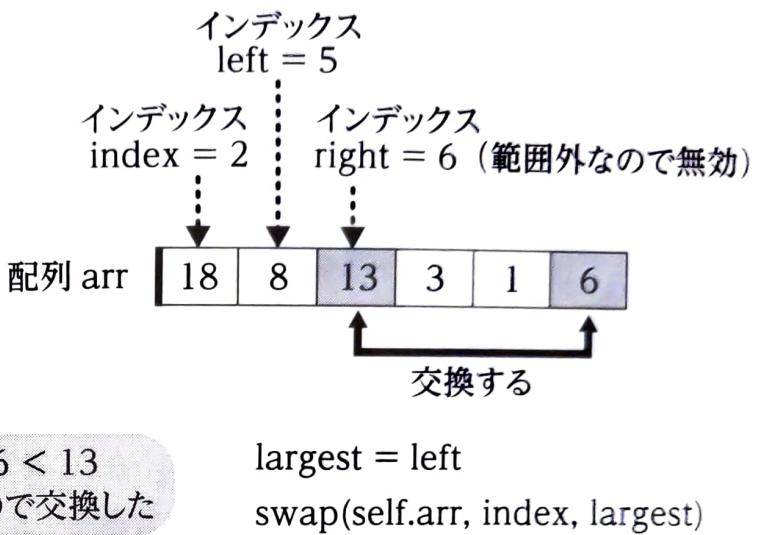
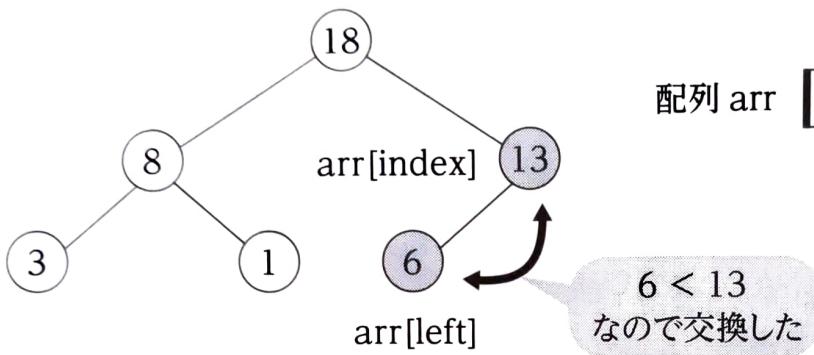


図 6.54 最大ヒープの生成の例 5

変数 `largest` の値は 5 になります。要素の交換を行ったため、再度、`self.max_heapify(5)` が呼び出されます。`self.arr[5]` は葉なので、何もせずに制御が `create_max_heap` メソッドに戻ってきます。これで 3 回目のイテレーションは終了です。

3回目のイテレーションを終えると for ループを抜けます。最終的なヒープの中身はさきほどの図 6.54 と同じです。二分木を見ると、ヒープの性質が満たされていることが確認できます。また、配列 `arr` は `[18, 8, 13, 3, 1, 6]` となります。

■ `extract_max` メソッド（一番優先度が高い要素の取得と、取得した要素を削除）の説明

54 行目～67 行目でヒープ内で一番優先度が高い要素を取得し、取得した要素を削除する `extract_max` メソッドを定義しています。56 行目の if 文で、ヒープが空かどうかを確認します。空であれば、取り出す要素がないので、if ブロックの中に入り、エラーメッセージを表示して処理を終了します。

ヒープに要素が含まれていれば、61 行目以降の処理を続行します。配列 `arr` の先頭要素が優先度が一番高い（値が一番大きい）要素なので、61 行目に示すように変数 `max` に `self.arr[0]` の値を保存します。取り出した要素はヒープ内から削除する必要があります。62 行目で配列の最後尾の要素を先頭にコピーし、63 行目で最後尾の削除します。配列として使用している Python の標準ライブラリのリストには、`pop` メソッドが提供されています。このメソッドは指定したインデックスの要