9.6 ヒープソート

ヒープソートは、図 9.3 のような**2 分木** (binary tree) というデータ構造を用い、**ヒープ化** (**ヒープ** (heap) については下記参照) と**ダウンヒープ** (down heap) という 2 つの段階を踏んでソーティングを行ないます。なお、2 分木は「○」で表された**節点** (node) と「—」で表された**枝** (branch) で構成されます。特に、節点の最上位を**根** (root)、子を持たない節点を**葉** (leaf) と呼び、根からある段までの節点を**深さ**と呼びます。また、図 9.4 のように分岐元となる節点を**親** (parent)、分岐先の節点を**子** (child) と呼び、親の要素を a_i とすると 2 つの子の要素は a_{2i} と a_{2i+1} となる親子関係を持っています。

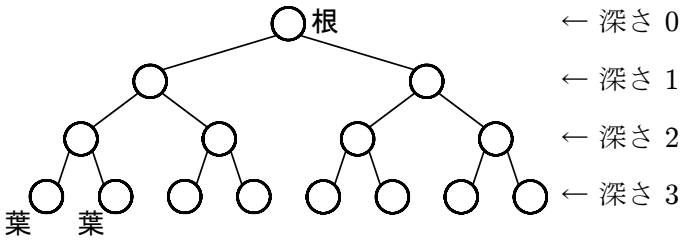

図 9.3: 2 分木

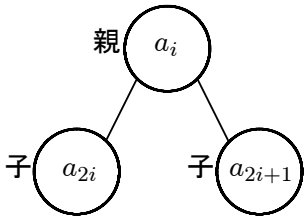

図 9.4: 2 分木の構造

従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} を 2 分木に対応させると図 9.5 のようになります。

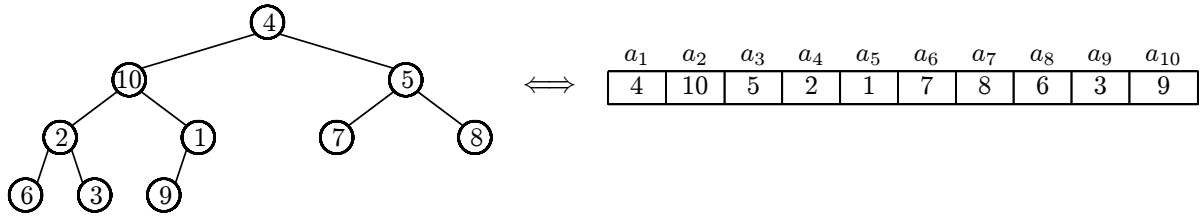

図 9.5: 2 分木と配列の対応

ヒープとダウンヒープは、基本的には同じ作業ですが、ヒープを適用する範囲が異なります。

ヒープ　「ある節点 a_i (親) をヒープする」とは、親 a_i と子 a_{2i}, a_{2i+1} の関係が「$a_i > a_{2i}$」かつ「$a_i > a_{2i+1}$」を満たすように親と子を入れ替え、この操作を下位に向かって (子, 孫, ひ孫, ··· の) 順に繰り返し行なうことです。なお、全ての親の節点で前記の親子関係が成り立つようにヒープを繰り返し適用することを**ヒープ化**と呼びます。

* 親を決定するために 2 回の比較が必要となります。

ダウンヒープ　a_1 と a_i を入れ替え ($a_1 \leftarrow a_i$ かつ $a_i \leftarrow a_1$)、節点 a_1 から a_{i-1} を対象に節点 a_1 をヒープします (節点 a_i から a_n まではソーティング済みとなるため、節点 a_1 から a_{i-1} からなる 2 分木としてヒープを行ないます)。これにより、節点 a_i から a_n までのソーティングが完了します。

* 次の Step では節点 a_1 から a_{i-1} を対象に、ダウンヒープを行ないます。

以上より、ヒープソートのアルゴリズムは、

ヒープ化：

Step 1　節点 $a_{[n/2]}$ をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1) (本当は 2×1 回)。

Step 2　節点 $a_{[n/2]-1}$ をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

⋮　⋮

Step $[n/2]$　節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

ダウンヒープ：

Step 1　節点 a_n と a_1 を交換し、節点 a_1 と a_{n-1} を対象に節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

Step 2　節点 a_{n-1} と a_1 を交換し、節点 a_1 と a_{n-2} を対象に節点 a_1 をヒープする。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

⋮　⋮

Step $n-1$　節点 a_2 と a_1 を交換する。最後は節点 a_1 をヒープする必要がない。
なぜなら、ヒープの対象となる子が存在しないからである。
* 最大で $2 \cdot m$ 回の比較が必要となる (*1)。

(*1)　計算が複雑になるので大きめに見積もります (計算量のオーダ表記の性質より)。

注意：節点 $a_{[n/2]}$ は子を持つ最後の親です ($[n/2]$ 以降の節点は子を持たない)。
注意：2 分木の深さを m (正整数) とします ($m = [(\log 2)^{-1} \cdot \log n - 1] + 1 = [(\log 2)^{-1} \cdot \log n]$)。

$$\because \quad n \fallingdotseq 1 + 2 + \cdots + 2^m = \frac{1 - 2^{m+1}}{1 - 2} = 2^{m+1} - 1 \fallingdotseq 2^{m+1} \quad \Longrightarrow \quad m \fallingdotseq (\log 2)^{-1} \cdot \log n - 1$$

* $[x]$ は x を超えない最大整数を表す (ガウス記号)。

となります。従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} をヒープソートすると図 9.6 の初期状態からヒープ化 (図 9.7〜図 9.11)、及び、ダウンヒープ (図 9.12〜図 9.17) の過程を経てソーティングが完了します (下記の「ヒープソートによるソーティングプログラム」でソーティングした場合)。

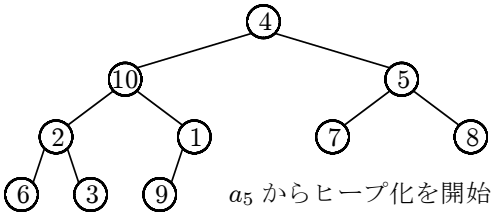

図 9.6: 初期状態

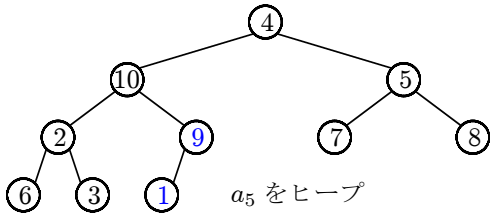

図 9.7: ヒープ化 (Step 1)

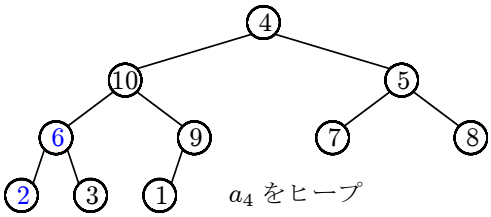

図 9.8: ヒープ化 (Step 2)

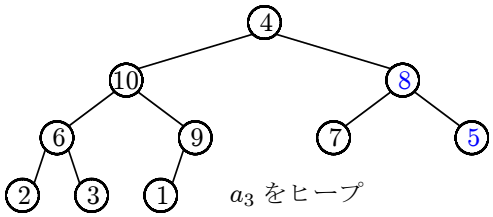

図 9.9: ヒープ化 (Step 3)

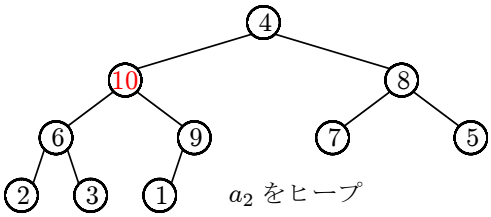

図 9.10: ヒープ化 (Step 4)

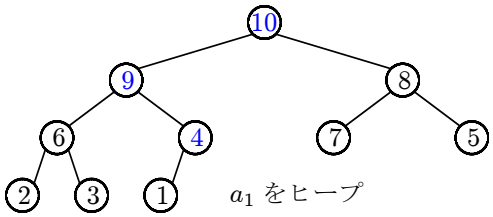

図 9.11: ヒープ化 (Step 5)

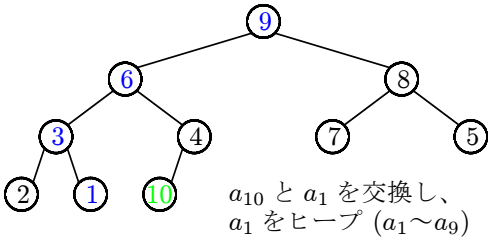

図 9.12: ダウンヒープ (Step 1)

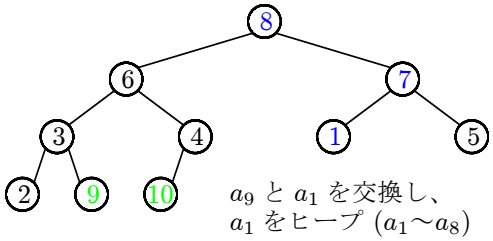

図 9.13: ダウンヒープ (Step 2)

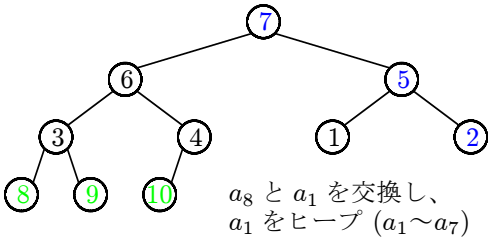

図 9.14: ダウンヒープ (Step 3)

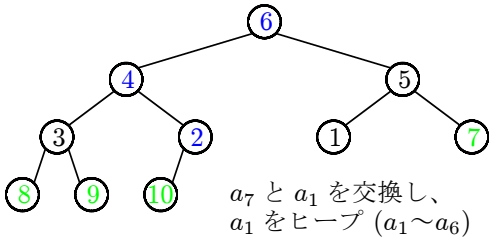

図 9.15: ダウンヒープ (Step 4)

··· 中略 ···

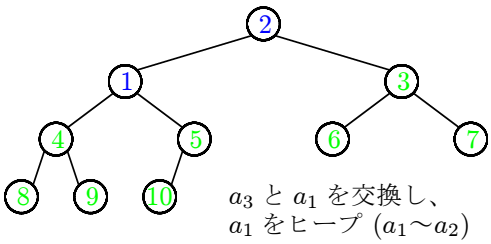

図 9.16: ダウンヒープ (Step 8)

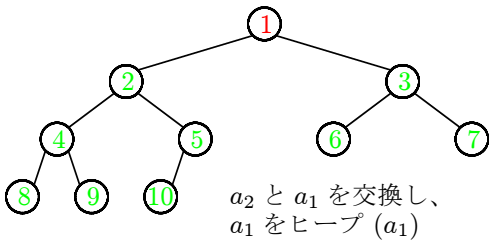

図 9.17: ダウンヒープ (Step 9)

● ヒープソートによるソーティングプログラム heapsort.c

```
 1: #include <stdio.h>
 2:
 3: #define N 10
 4:
 5: void heapsort(int a[]);
 6: void printarray(int a[]);
 7:
 8: int main(void)
 9: {
10:     int a[N+1] = {0, 4, 10, 5, 2, 1, 7, 8, 6, 3, 9};
11:
12:     heapsort(a);
13:
14:     return 0;
15: }
16:
17: void heapsort(int a[])
18: {
19:     int i, j, k, n = N, x;
20:
21:     for (k = n/2; k >= 1; k--) {
22:         i = k;
23:         x = a[i];
24:         j = 2 * i;
25:         while (j <= n) {
26:             if (j < n && a[j] < a[j+1]) j++;
27:             if (x >= a[j]) break;
28:             a[i] = a[j];
29:             i = j;
30:             j= 2 * i;
31:         }
32:         a[i] = x;
33:         printarray(a);
34:     }
35:     printf("¥n");
36:
```

```
37:     while (n > 1) {
38:         x = a[n];
39:         a[n] = a[1];
40:         n--;
41:         i = 1;
42:         j = 2 * i;
43:         while (j <= n) {
44:             if (j < n && a[j] < a[j+1]) j++;
45:             if (x >= a[j]) break;
46:             a[i] = a[j];
47:             i = j;
48:             j = 2 * i;
49:         }
50:         a[i] = x;
51:         printarray(a);
52:     }
53: }
54:
55: void printarray(int a[])
56: {
57:     int i;
58:
59:     for (i = 1; i <= N; i++) printf("%3d ", a[i]);
60:     printf("¥n");
61: }
```

なお、ヒープソートでは、ヒープ化の各 Step で最大 $2 \cdot m$ $(= 2(\log 2)^{-1} \log n)$ 回、ダウンヒープの各 Step で最大 $2 \cdot m$ $(= 2(\log 2)^{-1} \log n)$ 回の比較が必要となるので、計算量は

$$\left(\frac{n}{2}\right) \cdot (2(\log 2)^{-1} \log n) + (n-1) \cdot (2(\log 2)^{-1} \log n) = 3(\log 2)^{-1} n \log n - 2(\log 2)^{-1} \log n$$

となります。更に、オーダ記法で表すと $O(n \log n)$ となります。

ヒープソートの計算量のオーダはクイックソートと同じですが、平均的な計算速度はクイックソートに劣ります (**ヒープ化に際して要素の入れ替えが頻繁に起こるため**)。逆に、クイックソートに比べデータ列の良し悪しに左右されず安定したソーティングを行えるという長所を持ちます。

問題 1 「ヒープソートによるソーティングプログラム (heapsort.c)」を検証しなさい。

9.7　マージソート

マージソートは、2 つの整列済みデータ列を 1 つのデータ列に結合することでソーティングを行います。冒頭にも述べたとおりマージソートは外部整列なのですが、一度データ列の要素が 1 個になるまで分割し、それら要素の結合を繰り返すことで内部整列としてソーディングすることができます。アルゴリズムは、

Step 0　$\{a_1,\ a_2,\ \cdots,\ a_n\}$ を要素が 1 個になるまで 2 分割する ($\{a_1\},\ \{a_2\},\ \cdots,\ \{a_n\}$)。
Step 1　2 つの整列済みデータ列 $\{a_i\},\ \{a_j\}$ の要素を比較し、1 つの整列済みデータ列 $\{a'_i,\ a'_j\}$ に結合する ($n/2$ 個の整列済みデータ列ができる)。
　　* (最大で)$1 \cdot n/2$ 回の比較が必要となる。
Step 2　2 つの整列済みデータ列 $\{a'_i,\ a'_j\},\ \{a'_k,\ a'_l\}$ の要素を比較し、1 つの整列済みデータ列 $\{a''_i,\ a''_j,\ a''_k,\ a''_l\}$ に結合する ($n/4$ 個の整列済みデータ列ができる)。
　　* 最大で $2^2 \cdot n/4$ 回の比較が必要となる。
⋮　　⋮
Step m　2 つの整列済みデータ列の各要素を比較し、1 つの整列済みデータ列に結合する。
　　* 最大で $2^{2m} \cdot 1$ 回の比較が必要となる。

注意：Step 数 m は、クイックソートと同様に、再帰的方法による深さを表す (図 9.2 参照)。

となります。従って、データ列 {4, 10, 5, 2, 1, 7, 8, 6, 3, 9} をマージソートによってソーティングすると表 9.7 のようになります。

START	{4, 10, 5, 2, 1, 7, 8, 6, 3, 9}
Step 0	{{4, 10, 5, 2, 1}, {7, 8, 6, 3, 9}}　2 分割を繰り返す。
	{{{4, 10, 5}, {2, 1}}, {{7, 8, 6}, {3, 9}}}
	{{{{4, 10}, {5}}, {{2}, {1}}}, {{{7, 8}, {6}}, {{3}, {9}}}}
	{{{{{4}, {10}}, {5}}, {{2}, {1}}}, {{{{7}, {8}}, {6}}, {{3}, {9}}}}
Step 1	{{{{4, 10}, {5}}, {{2}, {1}}}, {{{7, 8}, {6}}, {{3}, {9}}}}
Step 2	{{{4, 5, 10}, {1, 2}}, {{6, 7, 8}, {3, 9}}}
Step 3	{{1, 2, 4, 5, 10}, {3, 6, 7, 8, 9}}
END	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

表 9.7: マージソートによるソーティングの例

なお、マージソートでは、Step 1 で 1 回の比較が $n/2$ 回、Step 2 で最大 4 回の比較が $n/4$ 回、$\cdots$、Step m で最大 2^{2m} 回の比較が $n/2^{2m}$ 回が必要となります。ただし、ステップ数 m は、表 9.7 のように再帰的に 2 分割する (2 分木を構成する) ことにすれば、$n = 2^m$ ($\Longleftrightarrow m = \log n \cdot (\log 2)^{-1}$)

が成り立ちます。従って、計算量は

$$\begin{aligned}\left(1\cdot\frac{n}{2}\right)+\left(4\cdot\frac{n}{4}\right)+\cdots+\left(2^{2m}\cdot\frac{n}{2^{2m}}\right) &= \frac{n}{2}+n+\cdots+n\\ &= \left(\frac{n}{2}-\frac{n}{2}\right)+\frac{n}{2}+n+\cdots+n\\ &= -\frac{n}{2}+n\cdot m\\ &= -\frac{n}{2}+(\log 2)^{-1}n\log n\end{aligned}$$

となります。更に、オーダ記法で表すと $O(n\log n)$ となります。

マージソートの計算量のオーダはクイックソートと同じですが、平均的な計算速度はクイックソートに劣ります (**2 つの整列済みデータ列を 1 つのデータ列にコピーする必要があるため**[6])。逆に、ヒープソートと同様、クイックソートに比べデータ列の良し悪しに左右されず安定したソーティングを行えるという長所を持ちます。

問題 1 「マージソートによるソーティングプログラム」を作成しなさい。

最後に、これまで紹介したソーティングアルゴリズムの計算量のオーダ表記について、表 9.8 にまとめておきます (**重要なのは平均的な場合**)。

	最良の場合	最悪の場合	平均的な場合
選択ソート	$O(n^2)$	$O(n^2)$	$O(n^2)$
バブルソート	$O(n)$	$O(n^2)$	$O(n^2)$
挿入ソート	$O(n)$	$O(n^2)$	$O(n^2)$
クイックソート	$O(n\log n)$	$O(n^2)$	$O(n\log n)$
ヒープソート	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$
マージソート	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$

表 9.8: ソーティングアルゴリズムの計算量の比較

[6]最低でも元の配列と同じ記憶容量 (メモリ) が必要となります。言い換えれば、他のソーティングプログラムに比べ、マージソートでは多くの記憶容量 (メモリ) が必要となることが欠点と言えます。逆に、記憶容量が十分にあれば安定したソーティングを行えるため、外部整列に向いたアルゴリズムと言えます。

(メモ用紙)