ソート (整列)

奈良女子大学理学部情報科学科 鴨浩靖

2009 年 12 月 10 日 初版
2011 年 11 月 29 日 第二版
2013 年 11 月 18 日 第三版
2020 年 12 月 21 日 第四版
2021 年 1 月 24 日 第四版改訂版

1　ソート (整列)

一列に並んだデータを大きさの順に並べかえることを、ソート (整列) という。

プログラミングでソートが必要となることは多い。たとえば、配列上で二分探索をするためにはその配列がソートされている必要がある。多くの場合は、まず、順にデータを配列に格納した上で一気にソートすると良い。データの入力は最初にまとめて行い、その後はデータの増減・変更ががない場合は、特にそうである。一方、追加・削除が繁雑にあるようなら二分探索木を使うほうが良い。適材適所をこころがけること。

ソートは古くから研究されていて、さまざまなアルゴリズムが発見されている。

1.1　代表的なソートのアルゴリズム

以下、$a[0], a[1], a[2], \ldots, a[n-1]$ を小さい順にソートする関数を例示する。int の値を交換する関数 `swap()` が以下のように定義されているものとし、サンプルコード中で使用する。

```
void
swap(int *p, int *q)
{
        int     tmp = *p;
        *p = *q;
        *q = tmp;
}
```

1.1.1　バブルソート

まず、次の繰り返しを実行する。

$a[0]$ と $a[1]$ を比較して、$a[0] > a[1]$ ならば交換する。

$a[1]$ と $a[2]$ を比較して、$a[2] > a[1]$ ならば交換する。
$a[2]$ と $a[3]$ を比較して、$a[3] > a[2]$ ならば交換する。
...
$a[n-2]$ と $a[n-1]$ を比較して、$a[n-2] > a[n-1]$ ならば交換する。

すると、$a[n-1]$ に最大の要素が入った状態になる。そこで、繰り返しの回数を一回減らして、同じことを実行する。

$a[0]$ と $a[1]$ を比較して、$a[0] > a[1]$ ならば交換する。
$a[1]$ と $a[2]$ を比較して、$a[2] > a[1]$ ならば交換する。
...
$a[n-2]$ と $a[n-1]$ を比較して、$a[n-2] > a[n-1]$ ならば交換する。

今度は、$a[n-2]$ に二番目に大きい要素が入った状態になる。さらに同じことを終端を減らしながら、同じことを実行しつづけ、最後に次を行って終る。

$a[0]$ と $a[1]$ を比較して、$a[0] > a[1]$ ならば交換する。

このアルゴリズムを**バブルソート** (泡立て法) と呼ぶ。

バブルソートの計算量は、最悪 $O(n^2)$、平均 $O(n^2)$ である。

バブルソートは、二重ループを使って C のプログラムとして書き下すことができる。以下はその例である。(`#include` 省略)

```
void
bubble_sort(int a[], size_t n)
{
        size_t  i, j;
        for (i = n - 1; i > 0; i --) {
                for (j = 0; j < i; j ++) {
                        if (a[j + 1] < a[j]) {
                                swap(&a[j], &a[j + 1]);
                        }
                }
        }
}
```

1.1.2　選択ソート

小さい要素から順に選んでいくことで、ソートすることもできる。それは、次のような繰り返しである。

$a[0]$ から $a[n-1]$ までを比較して最小のものを選び、$a[0]$ と交換する。
$a[1]$ から $a[n-1]$ までを比較して最小のものを選び、$a[1]$ と交換する。
$a[2]$ から $a[n-1]$ までを比較して最小のものを選び、$a[2]$ と交換する。
...

$a[n-2]$ から $a[n-1]$ までを比較して最小のものを選び、$a[n-2]$ と
交換する。

このアルゴリズムを**選択ソート**と呼ぶ。

選択ソートの計算量は、最悪 $O(n^2)$、平均 $O(n^2)$ である。

選択ソートも、二重ループを使って C のプログラムとして書き下すことができる。以下はその例である。(`#include` 省略)

```
void
selection_sort(int a[], size_t n)
{
        size_t  i, j, m;
        for (i = 0; i < n - 1; i ++) {
                m = i;
                for (j = i + 1; j < n; j ++) {
                        if (a[j] < a[m]) {
                                m = j;
                        }
                }
                swap(&a[i], &a[m]);
        }
}
```

1.1.3　挿入ソート

本来あるべき位置に挿入する操作を、前から順に行うことでソートすることもできる。それは、次のような繰り返しである。

$a[0]$〜$a[0]$ はソートされているので、$a[1]$ を適切な位置に挿入して、$a[0]$〜$a[1]$ がソートされている状態にする。

$a[0]$〜$a[1]$ はソートされているので、$a[2]$ を適切な位置に挿入して、$a[0]$〜$a[2]$ がソートされている状態にする。

$a[0]$〜$a[2]$ はソートされているので、$a[3]$ を適切な位置に挿入して、$a[0]$〜$a[3]$ がソートされている状態にする。

...

$a[0]$〜$a[n-2]$ はソートされているので、$a[n-1]$ を適切な位置に挿入して、$a[0]$〜$a[n-1]$ がソートされている状態にする。

このアルゴリズムを**選択ソート**と呼ぶ。

挿入ソートの計算量は、最悪 $O(n^2)$、平均 $O(n^2)$ である。

挿入ソートも、二重ループを使って C のプログラムとして書き下すことができる。以下はその例である。(#include 省略)

```
void
insertion_sort(int a[], size_t n)
{
        size_t  i, j;
        for (i = 1; i < n; i ++) {
                for (j = i; j > 0 && a[j] < a[j - 1]; j --) {
                        swap(&a[j - 1], &a[j]);
                }
        }
}
```

1.1.4　クイックソート

長さ 1 以下の列は自明にソートされている。それより長い列をソートするには、ます、小さな要素と大きな要素に分け、それぞれをソートして並べれば良い。

> $a[0], a[1], a_[2], \ldots, a[n-1]$ から、基準値 $a[p]$ を選ぶ。
>
> $a[0], a[1], a_[2], \ldots, a[n-1]$ から $a[p]$ を除いた $n-1$ 個のうち、m 個が $a[p]$ 以下であるとする。$a[p]$ を $a[m]$ に、$a[p]$ 以下である m 個を $a[0]$〜$a[m-1]$ に、その他の $n-m-1$ 個を $a[m+1]$〜$a[n-1]$ に移動させる。
>
> $a[0]$〜$a[m-1]$ と $a[m+1]$〜$a[n-1]$ を再帰的にそれぞれソートする。

このアルゴリズムを**クイックソート**と呼ぶ。

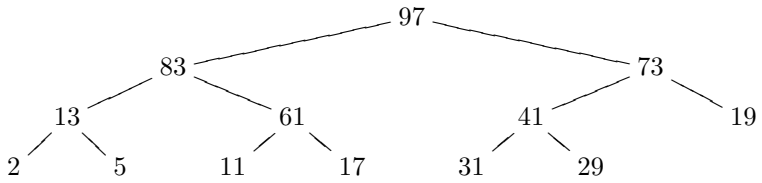
クイックソートの計算量は、最悪 $O(n^2)$、平均 $O(n \log n)$ である。最悪の場合が生じる確率は、基準値の選びかたに依存する。どのように選ぶのが良いかについては、数多くの研究があり、今も研究が進んでいる。

クイックソートを C のプログラムとして書き下すには、そのまま再帰呼び出しを実装するのが一つの方法である。他に、スタックを使って実装する方法もある。以下は、再帰呼び出しを使う例である。

```
void
quick_sort(int a[], size_t n)
{
        size_t  i, j;
        switch (n) {
        case 0:
        case 1:
                return;
        case 2:
                if (a[1] < a[0]) {
                        swap(&a[0], &a[1]);
                }
                return;
        }
        i = 1;
        j = n - 1;
        for (;;) {
                while (i <= j && a[0] >= a[i]) {
                        i ++;
                }
                if (j < i)
                        break;
                while (i <= j && a[0] < a[j]) {
                        j --;
                }
                if (j < i)
                        break;
                swap(&a[i], &a[j]);
                i ++;
                j --;
                if (j < i)
                        break;
        }
        swap(&a[0], &a[j]);
        quick_sort(a + j + 1, n - (j + 1));
        quick_sort(a, j);
}
```

1.1.5　ヒープソート

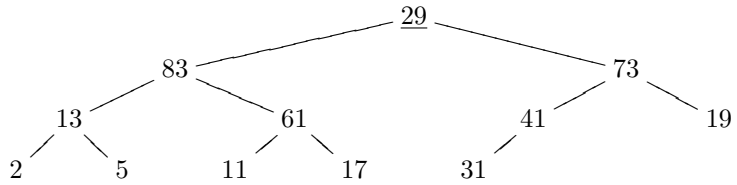
親のラベルが常に子のラベルより大きい二分木を**部分順序つき二分木**と呼ぶ。たとえば、



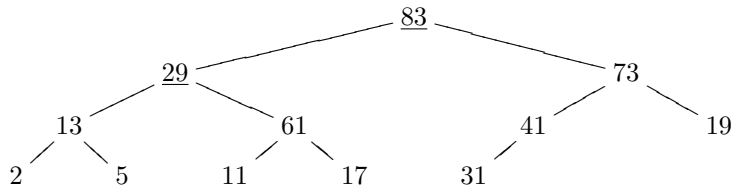
(♯)

根だけが部分順序つき二分木の条件をみたしていない二分木は、以下の手順で部分順序つき二分木にできる。

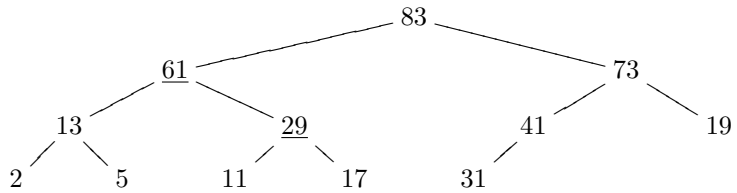
1. 根だけ、条件をみたしていない。



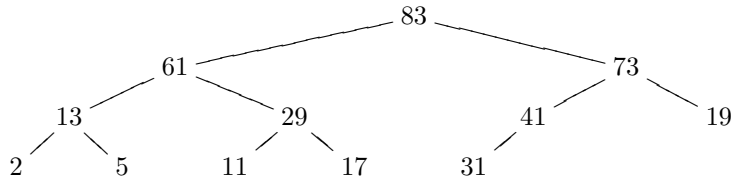
2. 親子の大小関係が正しくないので交換



3. 親子の大小関係が正しくないので交換

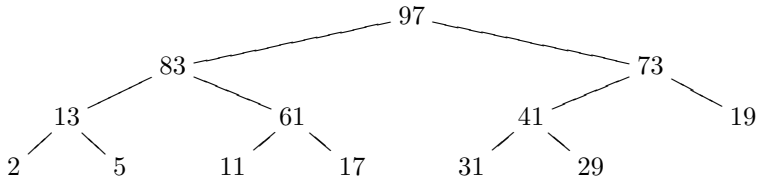


4. 全体が条件をみたした状態になった。

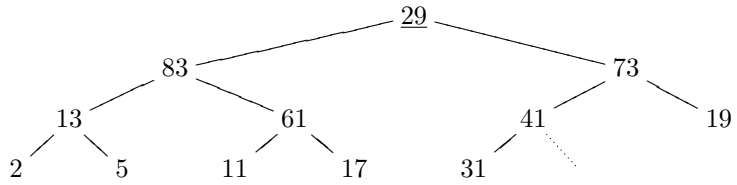


これを利用すると、部分順序つき二分木の構造を保ったまま (♯) から根の 97 を削除することが、以下の手順でできる。

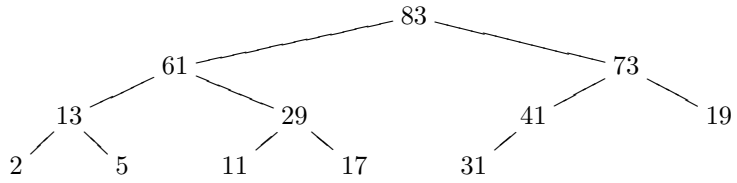
1. ヒープの条件をみたしている。



2. とりあえず、一番下の要素で穴を埋める。



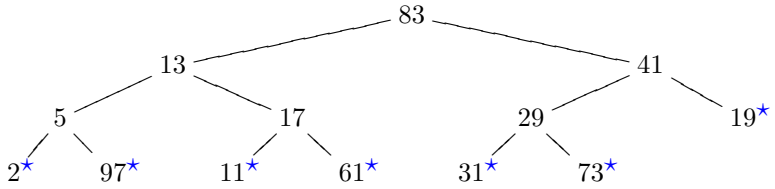
3. さきほどの手順で再構築



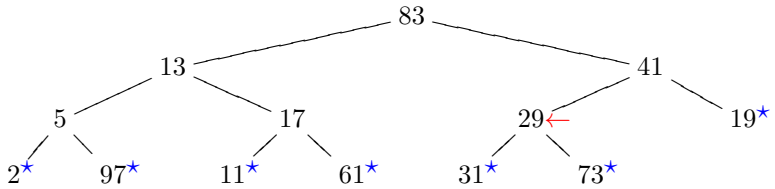
いったん木全体を部分順序つき二分木にすることができれば、根の要素の削除を繰り返すことで大きさの順に並べた列を構成できることがわかる。

そこで、木全体を部分順序つき二分木にする手順を考える。

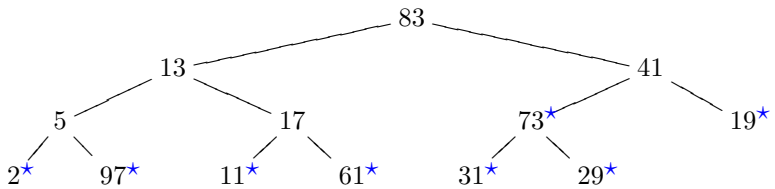
1. 葉は自明に部分順序つき二分木の条件をみたしている。



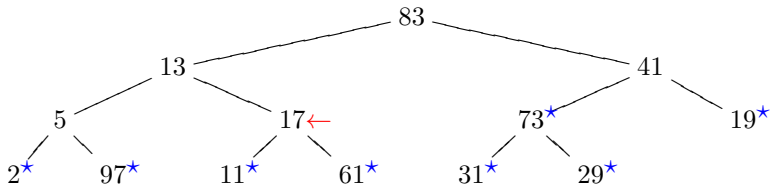
2. 部分順序つき二分木の条件をみたしているとわかっている部分の真上に着目。



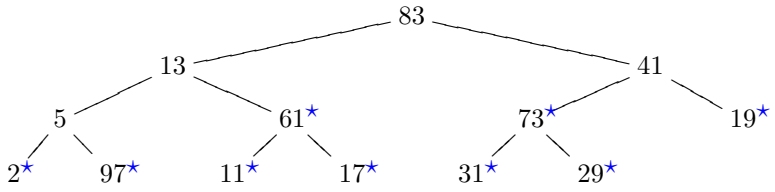
3. さきほどの手順で再構築



4. また、部分順序つき二分木の条件をみたしているとわかっている部分の真上に着目。



5. さきほどの手順で再構築



これを繰り返すと、木全体が部分順序つき二分木の条件をみたすようになる。

ところで、配列を配列のままで二分木とみなすことができる。$a[i]$ の左の子が $a[2i+1]$、右の子が $a[2i+2]$ と考えることで、配列を二分木とみなすことができる。そのようにして二分木とみなして部分順序つき二分木の構造を持つ配列を**ヒープ**と呼ぶ。

ヒープソートはヒープを利用してソートする。

1. 下から順にヒープを構築していく。
2. 以下をヒープの長さが 1 になるまで繰り返す。
   (a) ヒープの根と末尾を交換する。
   (b) ヒープの長さが 1 短かくなったとみなして、ヒープを再構築する。

ヒープソートの計算量は、最悪 $O(n \log n)$、平均 $O(n \log n)$ である。

ヒープソートを C のプログラムとして書き下すと、たとえば、次のようになる。(`#include` 省略)

```
static void
sift(int a[], size_t m, size_t n)
{
        size_t  i, j;
        for (i = m; (j = i * 2 + 1) < n; i = j) {
                if (j + 1 < n && a[j] < a[j + 1]) {
                        j ++;
                }
                if (a[i] < a[j]) {
                        swap(&a[j], &a[i]);
                } else
                        break;
        }
}

void
head_sort(int a[], size_t n)
{
        size_t  i;

        if (n <= 1)
                return;
        for (i = n / 2; i > 0;) {
                i --;
                sift(a, i, n);
        }
        for (i = n - 1; i > 0; i --) {
                swap(&a[i], &a[0]);
                sift(a, 0, i);
        }
}
```

1.1.6　マージソート

以下の二つは明らかに成り立つ。

- 長さ 1 の列はソートされている
- ソートされた二つをマージしてソートされた一つの列にすることは、長さに比例する計算時間で実行できる。

そこで、対象の列を最初は長さ 1 のソートされた列の並びと見なし、ふたつずつマージして長さ 2 のソートさ

れた列の並びとし、さらにふたつずつマージして長さ 4 のソートされた列の並びとし、これを繰り返して最後に全体をソートされた列にするのが、マージソートである。

たとえば、

5 41 11 23 47 31 17 43 3 29 53 2 37 7 13 19
5 41 11 23 31 47 17 43 3 29 2 53 7 37 13 19
5 11 23 41 17 31 43 47 2 3 29 53 7 13 19 37
5 11 17 23 31 41 43 47 2 3 7 13 19 29 37 53
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53

なお、マージソートには改良版がいくつかある。

（改良版を含めて）マージソートの時間計算量は、最悪 $O(n \log n)$、平均 $O(n \log n)$ である。ただし、配列にマージソートを適用する場合は、対象の配列の他に $O(n)$ の大きさの作業用記憶が別途必要となる。線形リストの場合は、そのような別の記憶容量は必要ではない。

1.2 まとめ

ソートのアルゴリズムはさまざまなものが発見されている。

	時間計算量		作業用記憶	
	最悪	平均	最悪	平均
バブルソート	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
選択ソート	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
挿入ソート	$O(n^2)$	$O(n^2)$	$O(1)$	$O(1)$
クイックソート	$O(n^2)$	$O(n \log n)$	$O(n)$	$O(\log n)$
ヒープソート	$O(n \log n)$	$O(n \log n)$	$O(1)$	$O(1)$
マージソート (配列)	$O(n \log n)$	$O(n \log n)$	$O(n)$	$O(n)$
マージソート (リスト)	$O(n \log n)$	$O(n \log n)$	$O(1)$	$O(1)$

長所短所を良く理解して、適切なものを選ぶこと。