アルゴリズム・データ構造 I 第12回
単純な整列アルゴリズム

名城大学理工学部情報工学科
山本修身

ソーティングの定義

ソーティングとは大小比較ができる与えられたn個の要素

$$x_1, x_2, \ldots, x_n$$

を，以下のように小さい順（大きい順）に並び替える計算のことを指す

$$x_{s_1} \leq x_{s_2} \leq \cdots \leq x_{s_n}$$

たとえば，7, 3, 5, 4, 1, 2, 4 という列をソートすれば，

1, 2, 3, 4, 4, 5, 7

となる．

仮定：すべての要素は比較可能で，比較の関係は三角不等式が成り立つ

いくつかのソートアルゴリズム

本日説明するソートアルゴリズム

セレクションソート： 説明する意味はあまりないが，その場で簡易的にコーディングする場合に利用する可能性がないとは言えない．ソート対象の集合の要素数の少ない場合には利用することができる．

クイックソート： すでに説明済み．深さ優先探索によってソートする．C.A.R Hoare によって発見された．

マージソート： やはり木探索によるソートである．クイックソートの場合，木がバランスしないと計算量が爆発する可能性があるが，マージソートは強制的にバランスさせる．そのためにクイックソートよりも効率が落ちるが動作は安定している．また与えられたデータと同じくらいの追加の場所が必要となる．

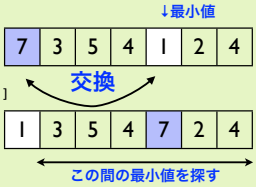
ヒープソート： 優先順位付きキュー（プライオリティキュー）（二分ヒープ）を用いてソートする．マージソートのように余分な領域を必要とせず，かつ，計算量は安定している．

シンプルだが遅いアルゴリズム(1)

```
function selection_sort(lst){
   var n = lst.length
   for (var i = 0; i< n; i++){
     var m = lst[i]
     var mj = i
     for (var j = i + 1; j < n; j++){
        if (m > lst[j]){
           m = lst[j]
           mj = j
        }
     }
     [lst[mj], lst[i]] = [lst[i], lst[mj]]
   }
}

lst = [7, 3, 5, 4, 1, 2, 4]
puts(lst)
selection_sort(lst)
puts(lst)
```

シンプルだが，一般的には遅いアルゴリズムがいくつか知られている．その一つが**セレクションソート**である．



7,3,5,4,1,2,4
1,2,3,4,4,5,7

シンプルだが遅いアルゴリズム(2)

実際にセレクションソートの速度を測ってみる．

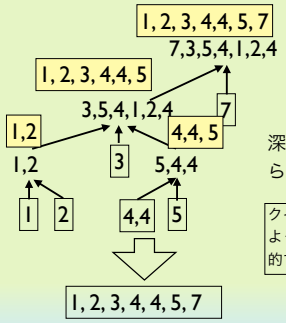
```
lst = []
for (var i = 0; i < 10000; i++)
   lst.push(Math.random())
t1 = new Date()
selection_sort(lst)
t2 = new Date()
puts((t2 - t1) + "ms")
puts(lst[0] + ":" + lst[5000] + ":" + lst[9999])
```

これを実行すれば，以下のようになる．

```
6929ms
0.00000124986700988927l7:0.494088572414469:0.9999612972741673
```


クイックソートアルゴリズム (1)

- 以前示したように，与えられた配列を分割して木構造を作りながらソートするアルゴリズムがクイックソートである．



深さ優先探索で葉を探索して得られる順に列挙すれば良い．

クイックソートではキーの選択方法によっていくつかの種類が存在するが本質的ではない

クイックソートアルゴリズム (2)

```
function qsort(lst){
   if (lst.length <= 1) return lst
   var key = lst[0]
   var lt = []
   var eq = []
   var gt = []
   for (var i = 0; i < lst.length; i++){
     var ele = lst[i]
     if (ele < key) lt.push(ele)
     else if (ele > key) gt.push(ele)
     else eq.push(ele)
   }
   return qsort(lt).concat(eq).concat(qsort(gt))
}

m = [7,3,5,4,1,2,4]
puts(m)
puts(qsort(m))
```

プログラムは再帰を用いて左のようになる．

プログラムを実行すると以下のようになる．

7,3,5,4,1,2,4
1,2,3,4,4,5,7

クイックソートアルゴリズム (3)

このプログラムの実行時間を測ってみる．

```
lst = []
for (var i = 0; i < 10000; i++)
   lst.push(Math.random())
t1 = new Date()
res = qsort(lst)
t2 = new Date()
puts((t2 - t1) + "ms")
puts(res[0] + ":" + res[5000] + ":" + res[9999])
```

プログラムを実行すると以下のようになる．

```
14ms
0.00007199321912798595:0.5064526987194872:0.9998958819354733
```

セレクションソートの約500倍速くソートできた．なぜ，こんなに速いのかは，次回の講義で述べる

クイックソートの問題点

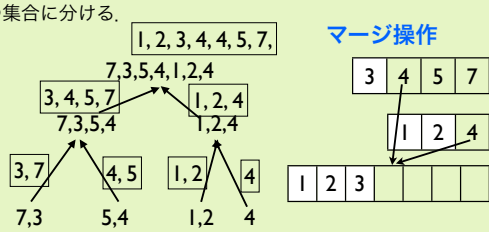
クイックソートはキーの値によって子供のノードに割り当てられるデータを決定するので，キーの値によっては，木が非常に歪な形になってしまう場合がある．実際，一度ソートされたリストを再度ソートしようとそれば，以下のように非常に効率が悪くなる．

```
lst = []
for (var i = 0; i < 1000; i++)
   lst.push(Math.random())
t1 = new Date()
res = qsort(lst)
t2 = new Date()
puts((t2 - t1) + "ms")
puts(res[0] + ":" + res[500] + ":" + res[999])
t1 = new Date()
lst = res
res = qsort(lst)
t2 = new Date()
puts((t2 - t1) + "ms")
puts(res[0] + ":" + res[500] + ":" + res[999])
```

```
3ms
0.00007232457237471568:0.4797530472329514:0.999864516975547
45ms
0.00007232457237471568:0.4797530472329514:0.999864516975547
```


マージソート (1)

もう一つの深さ優先探索を用いたソートアルゴリズムとしてマージソートがある．クイックソートはキーよりも大きな集合，キーに等しい集合，キーよりも小さな集合に分けたが，マージソートはほぼ大きさの等しい2つ集合に分ける．



半分ずつに分解した後，木を遡るときに「マージ」を行う．マージは２つの小さなソート列から１つの大きなソート列を作る作業である．

マージソート (2)

```
function merge(lst1, lst2){
  var lst = []
  while (true){
    if (lst1.length == 0) return lst.concat(lst2)
    else if (lst2.length == 0) return lst.concat(lst1)
    if (lst1[0] < lst2[0]) lst.push(lst1.shift())
    else lst.push(lst2.shift())
  }
}

function msort(lst){
  var n = lst.length
  if (n < 2) return lst
  else {
    var n2 = Math.floor(n / 2)
    return merge(
        msort(lst.slice(0, n2)), msort(lst.slice(n2, n)))
  }
}
```

マージソートのプログラムは以下のとおり．

7,3,5,4,1,2,4
1,2,3,4,4,5,7

```
m = [7,3,5,4,1,2,4]
puts(m)
puts(msort(m))
```


マージソート (3)

- マージソートの実行時間を測ってみる．

```
lst = []
for (var i = 0; i < 10000; i++)
   lst.push(Math.random())
t1 = new Date()
res = msort(lst)
t2 = new Date()
puts((t2 - t1) + "ms")
puts(res[0] + ":" + res[5000] + ":" + res[9999])
```

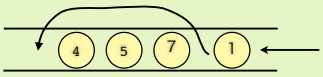
- 上のプログラムを実行すると以下のようになる．

```
59ms
0.00000194169570333579:0.49702765282789085:0.9999521926597846
```

クイックソートに比べると遅いが，セレクションソートの約120倍速くソートできた．

プライオリティーキュー (1)

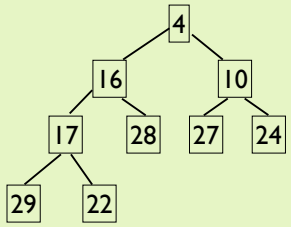
ここでソーティングから少々離れて，特殊なキューの構造について考えてみる．プライオリティーキュー（優先順位付きキュー）はデータを取り出すとき，最初に入れられたものではなく，一番小さな値を持つものが常に取り出される．

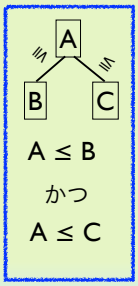


このようなキューを実現するデータ構造はフィボナッチヒープなどいくつか知られている．ここでは，２分ヒープによる実現を示す．

プライオリティーキュー (2)

- ２分ヒープは，２分木構造をしている．この２分木では，親ノードの値が必ず子供のノードよりも小さい．

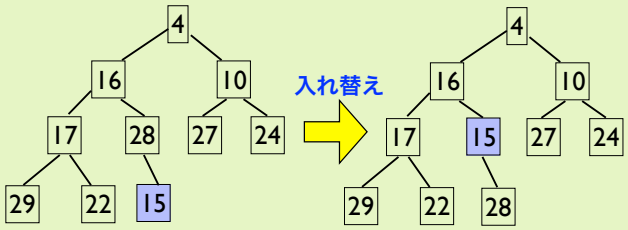




木のノードはいつも上から詰まって，一番末端は左から詰まっていくとする．

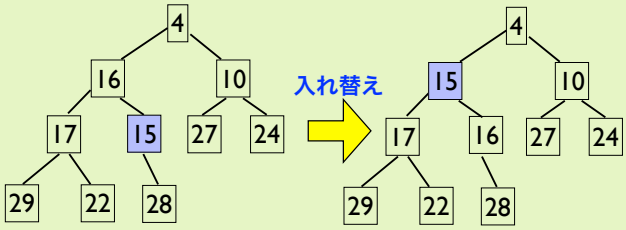
プライオリティーキュー (3)

データの挿入：この性質を保ちつつデータを付加するには，一番したの枝に要素を入れてから入れ替えによって，この性質が成り立つよう変形する．



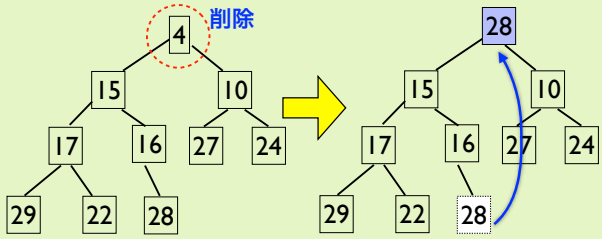
プライオリティーキュー (4)

データの挿入： 16と15の大小関係に問題があるので，さらに入れ替える．最大で根 (root) まで入れ替え作業を行えば，矛盾は解消する



プライオリティーキュー (5)

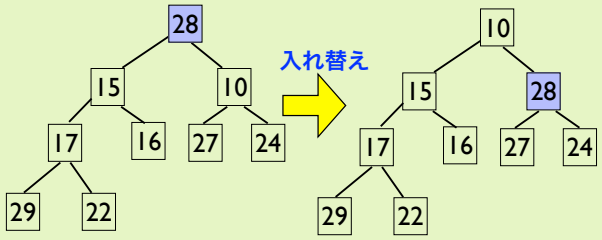
最小データの削除：一番値の小さなデータを削除するには，根の要素を削除すればよい（それが最小であることは明らか）．



削除しただけでは穴があいてしまうので最後の要素を根に持って行く

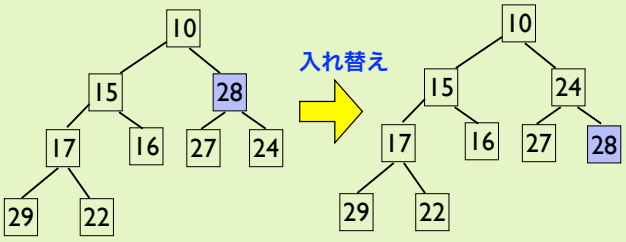
プライオリティーキュー (6)

- キューの条件を満たすために入れ替えを行う



プライオリティーキュー (7)

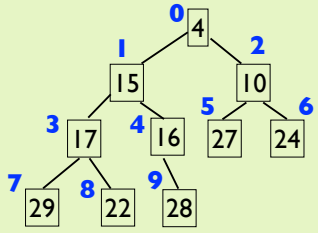
さらに入れ替えを行って，全体的に条件が満たされるようにする．



プライオリティーキュー (6)

- ２分ヒープは配列によって表現することができる

ノード i の親ノードは，Math.floor((i - 1) / 2)で計算することができる．子供のノードは，2i + 1と2i + 2になっている．



配列による表現

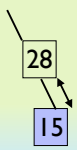
0	1	2	3	4	5	6	7	8	9
4	15	10	17	16	27	24	29	22	28

プライオリティーキューのプログラム (1)

```
heap = []
function parent(i){return Math.floor((i - 1) / 2)}
function child1(i){return 2 * i + 1}
function child2(i){return 2 * i + 2}
function exchange(i, j){
   var temp = heap[i]
   heap[i] = heap[j]
   heap[j] = temp
}

function add_element(ele){
  heap.push(ele)
  n = heap.length - 1
  while (n > 0) {
    if (heap[parent(n)] > heap[n]){
      exchange(parent(n), n)
      n = parent(n)
    } else {
      break
    }
  }
}
```

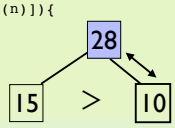
まず，親と子供のインデックスを計算する関数，ヒープの要素を交換する関数を定義する．ヒープに要素を付け加える関数add_element(ele)は最後にデータを入れて，大小関係を調べながら根に向かって交換していく．



プライオリティーキューのプログラム (2)

```
function delete_element(){
  var ret = heap[0]
  var n = heap.length - 1
  exchange(0, n)
  heap.length = n
  n = 0
  while (child1(n) < heap.length){
    if (child2(n) >= heap.length){
       if (heap[child1(n)] < heap[n])
          exchange(n, child1(n))
       break
    } else if (heap[child1(n)] > heap[n] &&
        heap[child2(n)] > heap[n]) break
    if (heap[child1(n)] < heap[child2(n)]){
       exchange(n, child1(n))
       n = child1(n)
    } else {
       exchange(n, child2(n))
       n = child2(n)
    }
  }
  return ret
}
```

要素を削除する場合には，少々複雑である．根のデータを取り出したあと，最後尾のデータを交換して，根から下に向けて交換を繰り返す．子供が1つしかないあるので，その場合は別途処理する．



どちらか小さい方を調べてそれと交換

プライオリティーキューのプログラム (3)

- 実際に動かしてみる．

```
add_element(23)
add_element(55)
add_element(1)
add_element(2)
add_element(30)
puts(heap)
puts(delete_element())
puts(heap)
puts(delete_element())
puts(heap)
```

```
1,2,23,55,30
1
2,30,23,55
2
23,30,55
```


プライオリティーキューのプログラム (4)

２分ヒープによるプライオリティーキューを使うとソートができる．これをヒープソート (heap sort) と呼ぶ．かなり速くソートできる．

```
function heap_sort(lst){
  heap = []
  for (var i = 0; i < lst.length; i++)
   add_element(lst[i])
  var ans = []
  while (heap.length > 0){
    ans.push(delete_element())
  }
  return ans
}
```

```
0.0000421288169178968 3:0.9999583648667706
time = 26 ms
```

```
lst = []
for (var i = 0; i < 10000; i++)
  lst.push(Math.random())
t1 = new Date()
ans = heap_sort(lst)
t2 = new Date()
puts(ans[0] + ":" + ans[lst.length - 1])
puts("time = " + (t2 -t1) + " ms")
```