

精読『アルゴリズムイントロダクション 第4版』

第3回

Panot

最終更新：May 13, 2023

復習

- ▶ 漸化式の解く方法色々：置き換え法、再帰木法、マスター法、Akra-Bazzi 法
- ▶

今日の内容

- ▶ ヒープソート (heapsort)
- ▶ クイックソート (quicksort)
- ▶ 線形時間ソート
- ▶ 中央値と順序統計量

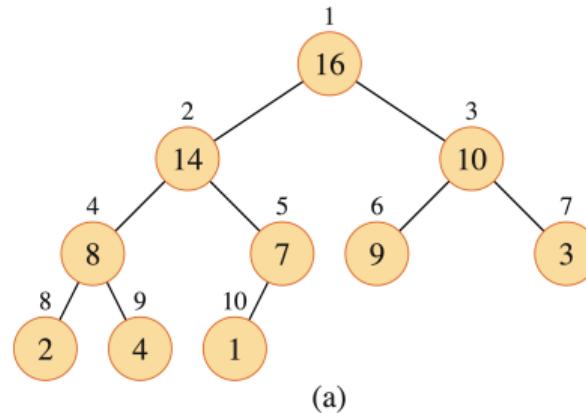
Section 1

ヒープソート

ヒープ (heap) とは

- ▶ weblio・研究社 新英和中辞典より「積み重ね、かたまり、山」。
- ▶ 二分ヒープ (**binary heap**) はおおよそ完全二分木。
- ▶ 二種類：**max ヒープ**と **min ヒープ**。
- ▶ max ヒープのヒープ条件 (**heap property**) は、節点がその親と同等またはそれより小さい。

$$A[\text{PARENT}(i)] \geq A[i]$$



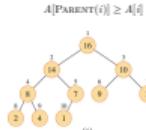
精読『アルゴリズム入門トロダクション 第4版』

└ ヒープソート

└ ヒープ (heap) とは

ヒープ (heap) とは

- ▶ weblio・研究社 新英和中辞典より「積み重ね、かたまり、山」。
- ▶ 二分ヒープ (binary heap) はおおよそ完全二分木。
- ▶ 二種類：max ヒープと min ヒープ。
- ▶ max ヒープのヒープ条件 (heap property) は、節点がその親と同等またはそれより小さい。 $A[\text{PARENT}(i)] \geq A[i]$



- 動的メモリ領域のことではない。
- 二分ではないヒープもありますが、ここでは扱いません。
- おおよそ完全2分木は、最下位階層以外すべての階層は埋まっていて、最下位階層では左から埋まっている。
- 必ず最大の値は木の根っこにある。

ヒープの配列による実装

- ▶ 配列での実装は簡単。
- ▶ 親または子の添字は簡単に計算できる。

PARENT(i)

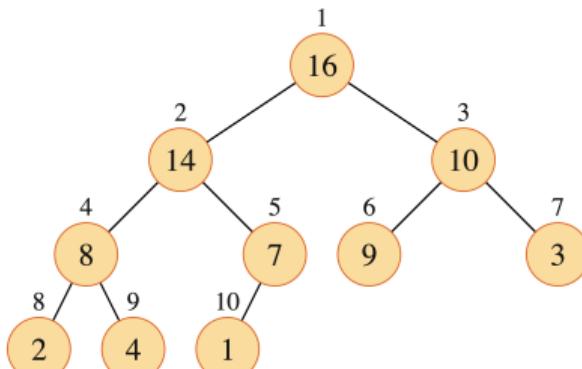
```
1 return  $\lfloor i/2 \rfloor$ 
```

LEFT(i)

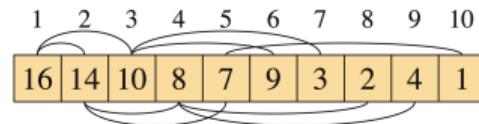
```
1 return  $2i$ 
```

RIGHT(i)

```
1 return  $2i + 1$ 
```



(a)



(b)

精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ ヒープの配列による実装

ヒープの配列による実装

- ▶ 配列での実装は簡単。
- ▶ 親または子の添字は簡単に計算できる。

PARENT(i)

 | return $\lfloor i/2 \rfloor$

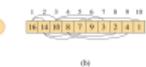
LEFT(i)

 | return $2i$

RIGHT(i)

 | return $2i + 1$

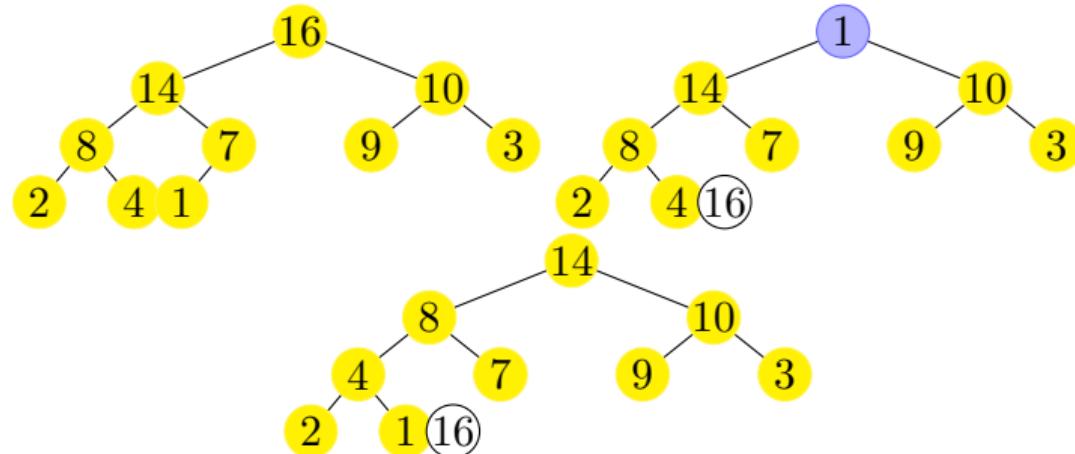
親または子の添字は簡単に計算できる。



- 普段の木の実装で `struct` を作ってポインタで参照することなど必要がない。
- 親または子の添字は簡単に計算できるし、配列だからメモリも線形で確保できるし、便利な実装です。

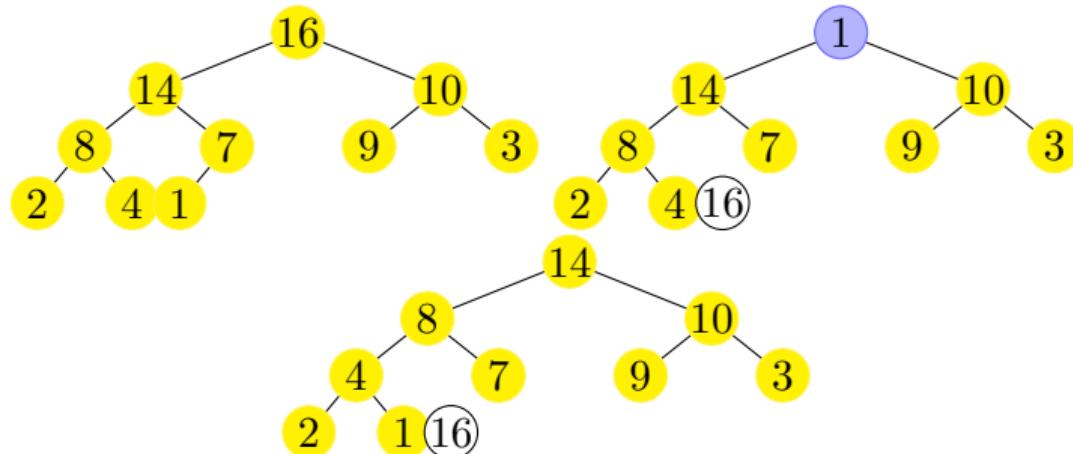
ヒープソートの戦略

1. 未整列の配列をヒープにする。最大要素が木の根っこにある
2. 最大要素を配列最後の要素と交換させて、ヒープから除外する。
3. ヒープの大きさが 0 になったら停止する。
4. 全体はヒープ条件を満たさなくなったが、根っここの両方の子節点以下の部分木はヒープ条件を満たしたまま。
5. 全体をヒープにする。
6. 2 に戻る。



ヒープソートの戦略

1. 未整列の配列をヒープにする。最大要素が木の根っこにある $O(n)$
2. 最大要素を配列最後の要素と交換させて、ヒープから除外する。 $O(1)$
3. ヒープの大きさが 0 になったら停止する。
4. 全体はヒープ条件を満たさなくなつたが、根っここの両方の子節点以下の部分木はヒープ条件を満たしたまま。
5. 全体をヒープにする。 $O(\lg k)$
6. 2 に戻る。



精読『アルゴリズム入門ダクション 第4版』

└ ヒープソート

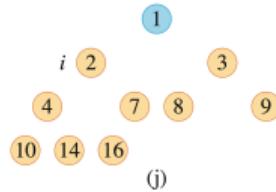
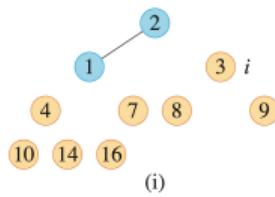
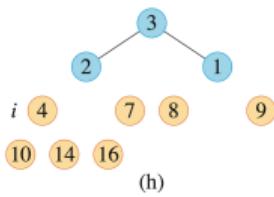
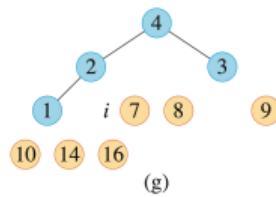
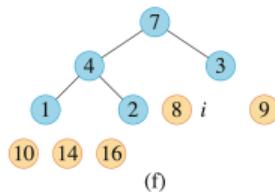
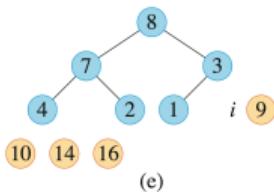
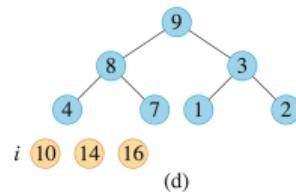
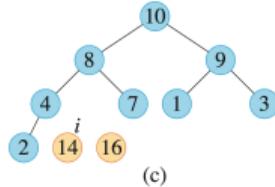
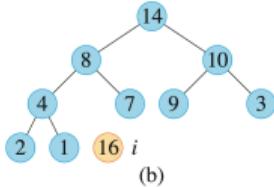
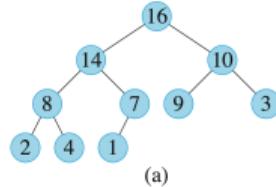
└ ヒープソートの戦略

- ヒープソートの戦略**
- 未整列の配列をヒープにする。最大要素が木の根っこにある $O(n)$
 - 最大要素を配列最後の要素と交換させて、ヒープから除外する。 $O(1)$
 - ヒープの大きさが 0 になら停止する。
 - 全体はヒープ条件を満たさなくなつたが、根っここの両方の子節点以下部分木はヒープ条件を満たします。
 - 全体をヒープにする。 $O(\lg n)$
 - 2 に戻る。



- おおまかに見れば、最大要素を 1 個ずつ探していくように見える。
- 単純にそれだけだと、**選択ソート**と同じで、実行時間は $O(n^2)$ になる。
- 無秩序に残っている要素の中で最大要素を探し出すのは、残っている要素の数 n に対して実行時間 $O(n)$ であるため、全体の実行時間は $O(n^2)$ になる。これはためだ。
- ヒープソートのキモはヒープの性質。4 に書いてあるように、最大要素を別の要素と交換しても、それ以下両方の部分木はヒープ条件を満たしているままということ。
- その性質のもとで、全体がヒープ条件を満たすようにするのは $O(\lg n)$ の実行時間しか要しないため、選択ソートより速い。

ヒープソートの実行例



A	[1 2 3 4 7 8 9 10 14 16]
---	--

(k)

必要な手続き

- ▶ 両方の部分木がヒープ条件を満たしたとき、全体をヒープにする。
- ▶ 未整列の配列をヒープにする。
- ▶ ヒープソート。

必要な手続き

- ▶ 両方の部分木がヒープ条件を満たしたとき、全体をヒープにする。
MAX-HEAPIFY $O(\lg n)$
- ▶ 未整列の配列をヒープにする。 **BUILD-MAX-HEAP** $O(n)$
- ▶ ヒープソート。 **HEAPSORT** $O(n \lg n)$

精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ 必要な手続き

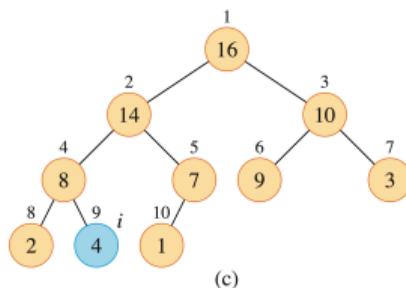
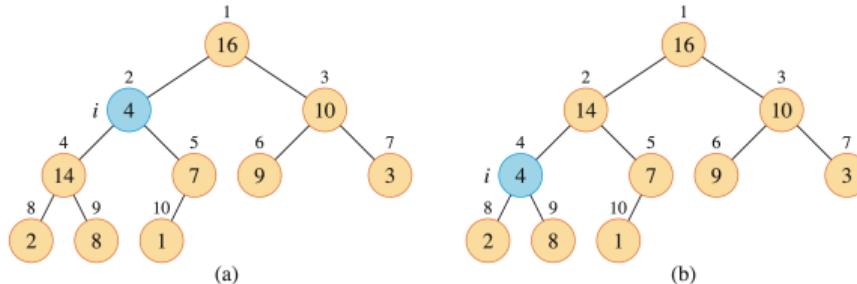
必要な手続き

- ▶ 両方の部分木がヒープ条件を満たしたとき、全体をヒープにする。
MAX-HEAPIFY $O(\lg n)$
- ▶ 未整列の配列をヒープにする。BUILD-MAX-HEAP $O(n)$
- ▶ ヒープソート。HEAPSORT $O(n \lg n)$

- MAX-HEAPIFY を先に紹介するのは BUILD-MAX-HEAP がそれを利用するためです。

MAX-HEAPIFY

- 両方の部分木がヒープ条件を満たしている。
- 根っこがその子より小さかったら 3 つの中の最大の要素を根っこと交換する。
- 交換した部分木を MAX-HEAPIFY を適用する。
- $O(\lg n)$



MAX-HEAPIFY

- ▶ 両方の部分木がヒープ条件を満たしている。
- ▶ 根っこがその子より小さかったら 3 つの中の最大の要素を根っこと交換する。
- ▶ 交換した部分木を MAX-HEAPIFY を適用する。
- ▶ $O(\lg n)$

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $largest = l$ 
5  else  $largest = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[largest]$ 
7       $largest = r$ 
8  if  $largest \neq i$ 
9      exchange  $A[i]$  with  $A[largest]$ 
10     MAX-HEAPIFY( $A, largest$ )
```

精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ MAX-HEAPIFY

MAX-HEAPIFY

- 両方の部分木がヒープ条件を満たしている。
- 根っこがその子より小さかったら、3つの中の最大の要素を根っこと交換する。
- 交換した部分木を MAX-HEAPIFY を適用する。
- $O(\lg n)$

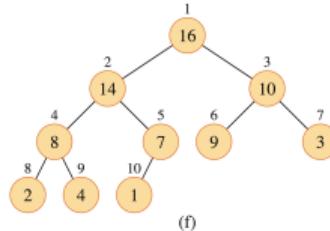
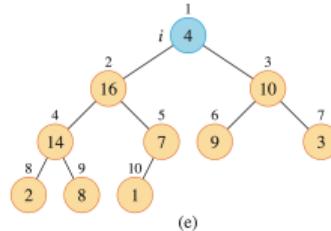
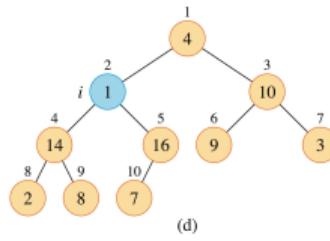
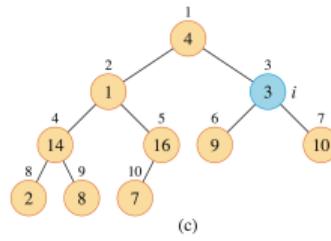
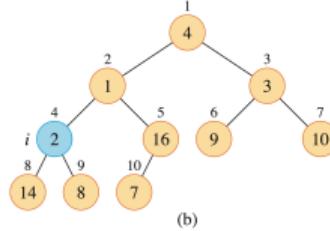
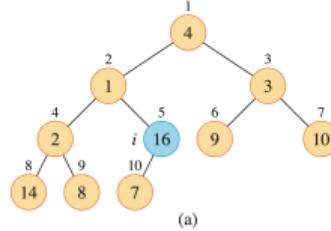
```
MAX-HEAPIFY( $A, i$ )  
1    $l = \text{LEFT}(i)$   
2    $r = \text{RIGHT}(i)$   
3   if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$   
4        $\text{largest} = l$   
5   else  $\text{largest} = i$   
6   if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$   
7        $\text{largest} = r$   
8   if  $\text{largest} \neq i$   
9       exchange  $A[i]$  with  $A[\text{largest}]$   
10      MAX-HEAPIFY( $A, \text{largest}$ )
```

- 根っこを適当なところまで沈める。

BUILD-MAX-HEAP

下からヒープを作る。

A [4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7]



BUILD-MAX-HEAP

下からヒープを作る。 $O(n \lg n)$ だが、 $O(\lg n)$ でもある！

BUILD-MAX-HEAP(A, n)

1 $A.heap-size = n$

2 **for** $i = \lfloor n/2 \rfloor$ **downto** 1

3 MAX-HEAPIFY(A, i)

精読『アルゴリズム入門トロダクション 第4版』

└ ヒープソート

└ BUILD-MAX-HEAP

BUILD-MAX-HEAP

下からヒープを作る。 $O(n \lg n)$ だが、 $O(n \lg n)$ もある！BUILD-MAX-HEAP(A, n)

```
1   $A.\text{heap-size} = n$ 
2  for  $i = \lfloor n/2 \rfloor$  downto 1
3      MAX-HEAPIFY( $A, i$ )
```

- 未整列の配列を二分木とみなす。
- 葉っぱは既にヒープである。
- MAX-HEAPIFY を使って、下から全体をヒープにしていく。

ヒープソート

1. 未整列の配列をヒープにする。最大要素が木の根っこにある $O(n)$
2. 最大要素を配列最後の要素と交換させて、ヒープから除外する。 $O(1)$
3. ヒープの大きさが 0 になったら停止する。
4. 全体はヒープ条件を満たさなくなったが、根っここの両方の子節点以下の部分木はヒープ条件を満たしたまま。
5. 全体をヒープにする。 $O(\lg k)$
6. 2 に戻る。

ヒープソート

1. 未整列の配列をヒープにする。最大要素が木の根っこにある $O(n)$
2. 最大要素を配列最後の要素と交換させて、ヒープから除外する。 $O(1)$
3. ヒープの大きさが 0 になったら停止する。
4. 全体はヒープ条件を満たさなくなったが、根っここの両方の子節点以下の部分木はヒープ条件を満たしたまま。
5. 全体をヒープにする。 $O(\lg k)$
6. 2 に戻る。

HEAPSORT(A, n)

- 1 BUILD-MAX-HEAP(A, n)
- 2 **for** $i = n$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 5 MAX-HEAPIFY($A, 1$)

ヒープソート

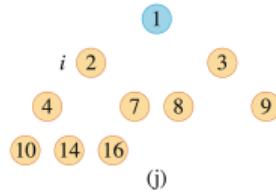
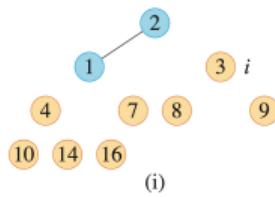
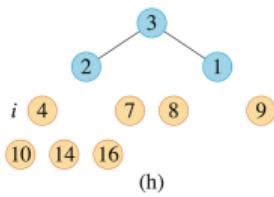
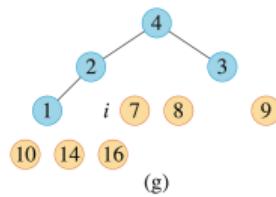
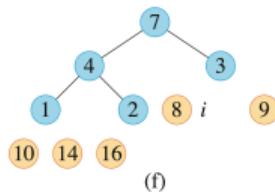
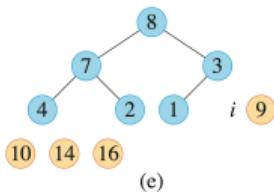
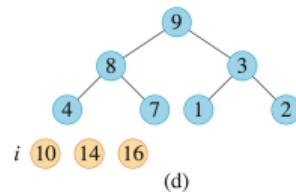
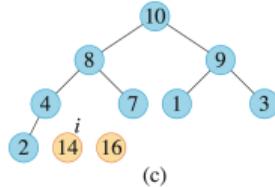
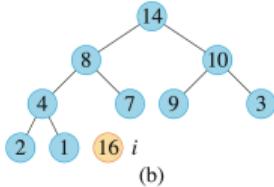
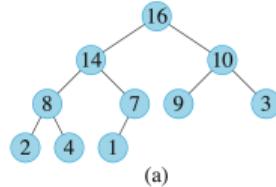
1. 未整列の配列をヒープにする。最大要素が木の根っこにある $O(n)$
2. 最大要素を配列最後の要素と交換させて、ヒープから除外する。 $O(1)$
3. ヒープの大きさが 0 になったら停止する。
4. 全体はヒープ条件を満たさなくなったが、根っここの両方の子節点以下の部分木はヒープ条件を満たしたまま。
5. 全体をヒープにする。 $O(\lg k)$
6. 2 に戻る。

HEAPSORT(A, n)

- 1 BUILD-MAX-HEAP(A, n)
- 2 **for** $i = n$ **downto** 2
- 3 exchange $A[1]$ with $A[i]$
- 4 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 5 MAX-HEAPIFY($A, 1$)

ヒープソートは $O(n \lg n)$

ヒープソートの実行例



A	[1 2 3 4 7 8 9 10 14 16]
---	--

(k)

優先度付きキュー (priority queue)

- ▶ 優先度付きキュー (priority queue) は、各要素に優先度がついている要素の集合。
- ▶ **max** 優先度付きキューは、この集合から常に優先度が最大のものを取り出す操作が用意されているもの。
- ▶ 次の操作が用意されている：
 - ▶ INSERT 挿入
 - ▶ MAXIMUM 優先度が最大の要素を返す
 - ▶ EXTRACT-MAXIMUM 優先度が最大の要素を取り出す
 - ▶ INCREASE-KEY キュー内の要素の優先度を上げる

精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ 優先度付きキュー (priority queue)

優先度付きキュー (priority queue)

- ▶ 優先度付きキュー (priority queue) は、各要素に優先度がついている要素の集合。
- ▶ max 優先度付きキューは、この集合から常に優先度が最大のものを取り出す操作が得意されているもの。
- ▶ 次の操作が得意されている：
 - ▶ INSERT 締入
 - ▶ MAXIMUM 優先度が最大の要素を返す
 - ▶ EXTRACT-MAXIMUM 優先度が最大の要素を取り出す
 - ▶ INCREASE-KEY キュー内の要素の優先度を上げる

- max 優先度付きキューに対して min 優先度付きキューもある。
- タスクがいっぱいあるときに、一番重要なものを先に実行するようなイメージ。
- 配列でも実装できるが、実行時間が長いです。整列させて、順番に取り出したらいい

実装の比較

実装	初期化	取り出し	挿入	優先度変更
整列された配列 ヒープ	$O(n \lg n)$ $O(n \lg n)$	$O(1)$	$O(n)$	$O(n)$

MAXIMUM と EXTRACT-MAXIMUM

- ▶ 優先度が最大の要素を返す操作は $O(1)$ 。

MAXIMUM と EXTRACT-MAXIMUM

- ▶ 優先度が最大の要素を返す操作は $O(1)$ 。
- ▶ 優先度が最大の要素を取り出す操作は：その要素をヒープから除外して、集合をまたヒープにする。 $O(\lg n)$ 。

MAXIMUM と EXTRACT-MAXIMUM

- ▶ 優先度が最大の要素を返す操作は $O(1)$ 。
- ▶ 優先度が最大の要素を取り出す操作は：その要素をヒープから除外して、集合をまたヒープにする。 $O(\lg n)$ 。

MAX-HEAP-MAXIMUM(A)

- 1 **if** $A.\text{heap-size} < 1$
- 2 **error** “heap underflow”
- 3 **return** $A[1]$

MAX-HEAP-EXTRACT-MAX(A)

- 1 $max = \text{MAX-HEAP-MAXIMUM}(A)$
- 2 $A[1] = A[A.\text{heap-size}]$
- 3 $A.\text{heap-size} = A.\text{heap-size} - 1$
- 4 $\text{MAX-HEAPIFY}(A, 1)$
- 5 **return** max

精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ MAXIMUM と EXTRACT-MAXIMUM

MAXIMUM と EXTRACT-MAXIMUM

- 優先度が最大の要素を返す操作は $O(1)$ 。
- 優先度が最大の要素を取り出す操作は：その要素をヒープから除外して、集合をまたヒープにする。 $O(\lg n)$ 。

MAX-HEAP-MAXIMUM(A)

```
1 if  $A.heap\_size < 1$ 
2 error "heap underflow"
3 return  $A[1]$ 
```

MAX-HEAP-EXTRACT-MAX(A)

```
1 root = MAX-HEAP-MAXIMUM( $A$ )
2  $A[1] = A[A.heap\_size]$ 
3  $A.heap\_size = A.heap\_size - 1$ 
4 MAX-HEAPPY( $A, 1$ )
5 return root
```

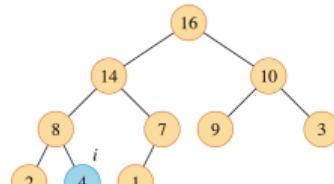
- 返すは根っこを参照するだけで十分。
- 取り出す操作は、参照して、集合から除外して、残りをまらヒープにする。ヒープソートのループ内と同じ操作。
- ここまでヒープソートと同じ。

実装の比較

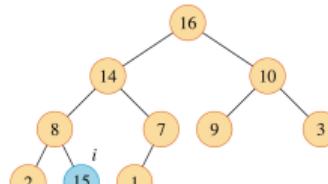
実装	初期化	取り出し	挿入	優先度変更
整列された配列 ヒープ	$O(n \lg n)$ $O(n \lg n)$	$O(1)$ $O(\lg n)$	$O(n)$	$O(n)$

INCREASE-KEY と INSERT

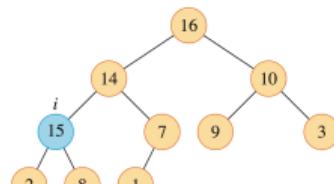
- ▶ キュー内の要素の優先度を上げる。
- ▶ 優先度を上げても、その要素を根っことした部分木はまだヒープ。
- ▶ ただし、その親以上の部分木はヒープと限らない。
- ▶ 親より優先度が大きかったら交換していく。



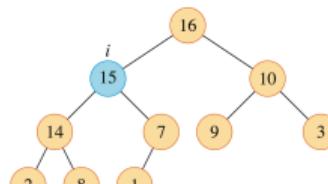
(a)



(b)



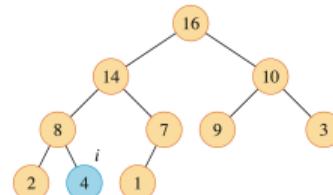
(c)



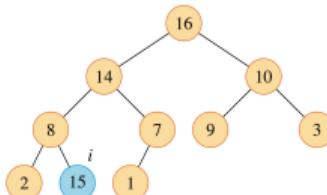
(d)

INCREASE-KEY と INSERT

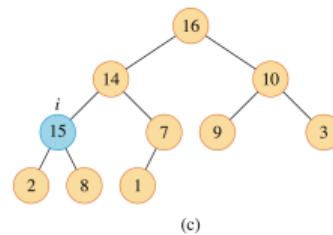
- ▶ キュー内の要素の優先度を上げる。
- ▶ 優先度を上げても、その要素を根っことした部分木はまだヒープ。
- ▶ ただし、その親以上の部分木はヒープと限らない。
- ▶ 親より優先度が大きかったら交換していく。



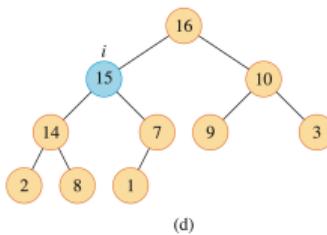
(a)



(b)



(c)



(d)

挿入も、とりあえず葉っぱにくっつけて浮上させる。

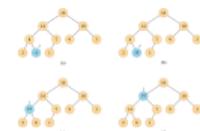
精読『アルゴリズムイントロダクション 第4版』

└ ヒープソート

└ INCREASE-KEY と INSERT

INCREASE-KEY と INSERT

- ▶ キュー内の要素の優先度を上げる。
- ▶ 優先度を上げても、その要素を根っことした部分木はまだヒープ。
- ▶ ただし、その親以上の部分木はヒープと限らない。
- ▶ 親より優先度が大きかったら交換していく。



挿入も、とりあえず重っぽくくっつけて浮上させる。

- その要素を適当なところまで浮上させる。
- MAX-HEAPIFY と逆の操作。

INCREASE-KEY & INSERT

MAX-HEAP-INCREASE-KEY(A, x, k)

- 1 **if** $k < x.key$
- 2 **error** “new key is smaller than current key”
- 3 $x.key = k$
- 4 find the index i in array A where object x occurs
- 5 **while** $i > 1$ and $A[\text{PARENT}(i)].key < A[i].key$
- 6 exchange $A[i]$ with $A[\text{PARENT}(i)]$, updating the information that maps priority queue objects to array indices
- 7 $i = \text{PARENT}(i)$

MAX-HEAP-INSERT(A, x, n)

- 1 **if** $A.\text{heap-size} == n$
- 2 **error** “heap overflow”
- 3 $A.\text{heap-size} = A.\text{heap-size} + 1$
- 4 $k = x.key$
- 5 $x.key = -\infty$
- 6 $A[A.\text{heap-size}] = x$
- 7 map x to index heap-size in the array
- 8 MAX-HEAP-INCREASE-KEY(A, x, k)

実装の比較

実装	初期化	取り出し	挿入	優先度変更
整列された配列	$O(n \lg n)$	$O(1)$	$O(n)$	$O(n)$
ヒープ	$O(n \lg n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$