

École Polytechnique

CSC_41011

Les bases de la programmation et de l'algorithmique

Jean-Christophe Filliâtre

files de priorité

- conditions habituelles d'un TD
mais pas d'accès à Internet
- programme = blocs 1 à 6
- **polycopié autorisé**
- dictionnaire électronique autorisé

penser à amener une feuille et un crayon

quels sont les 100 titres de pages Wikipedia les plus longs ?

(ou encore : quelles sont les M plus grandes valeurs parmi N ?)

- tout lire
⇒ espace N
- chercher le plus long titre, puis le suivant, etc.
⇒ temps $N \times M$

- tout lire
⇒ espace N
- trier les N valeurs (cf la semaine prochaine)
⇒ temps $N \times \log N$

il y a en a trop

```
Exception in thread "main"  
java.lang.OutOfMemoryError: Java heap space
```

- on ne conserve que M valeurs en mémoire, dans un tableau
⇒ espace M
- chaque nouvelle valeur est comparée à la plus petite
 - si elle est plus petite, on l'ignore
 - sinon, elle prend sa place⇒ temps $N \times M$

- on ne conserve que M valeurs en mémoire, dans un **AVL** (en supposant les doublons autorisés)
 - ⇒ espace M
- chaque nouvelle valeur est comparée à la plus petite
 - si elle est plus petite, on l'ignore
 - sinon, elle prend sa place
 - ⇒ temps $N \times \log M$

on utilise les seules opérations

- ajouter un élément
- trouver le plus petit
- supprimer le plus petit

file de priorité

```
class PriorityQueue<E> {
    PriorityQueue<E>() // nouvelle file, vide
    boolean isEmpty() // vide ?
    int size() // nombre d'éléments
    void add(E x) // ajouter x
    E getMin() // plus petit élément
    E removeMin() // supprime et renvoie
}
```

si on préfère `getMax` et `removeMax`, il suffit d'**inverser** la relation d'ordre

on peut réaliser une file de priorité avec une solution

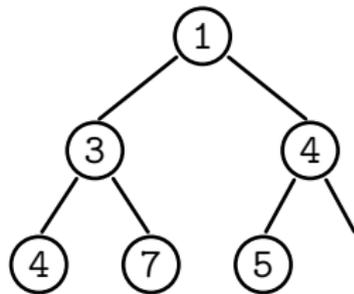
- plus simple
- plus compacte
- de même complexité

que celle consistant à utiliser un ABR équilibré

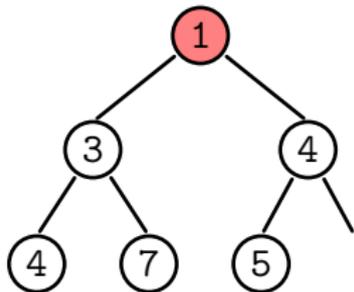
structure de tas

un **tas** (en anglais *heap*) est un arbre binaire où

tout nœud n'est pas plus grand
que les deux nœuds immédiatement
au-dessous

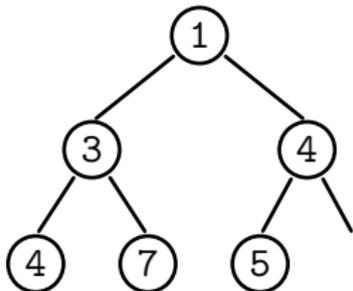


le plus petit élément d'un tas est situé **à la racine**



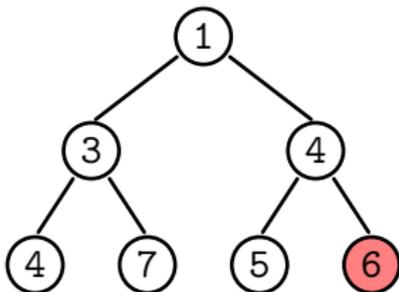
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 6



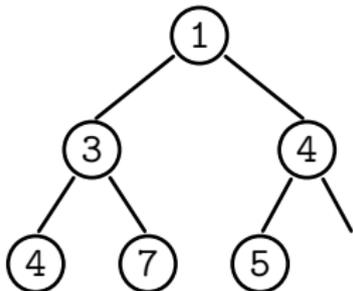
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 6



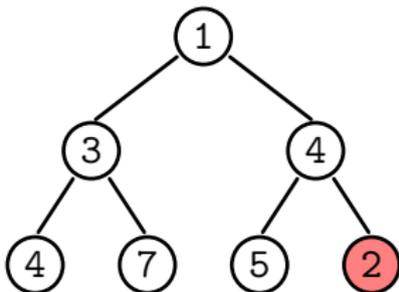
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 2



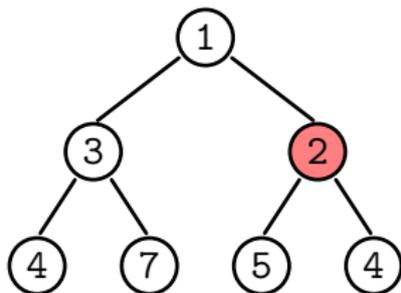
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 2



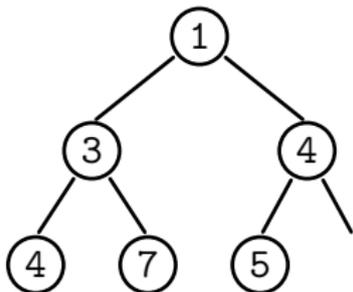
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 2



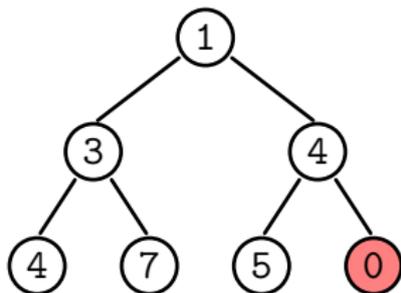
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 0



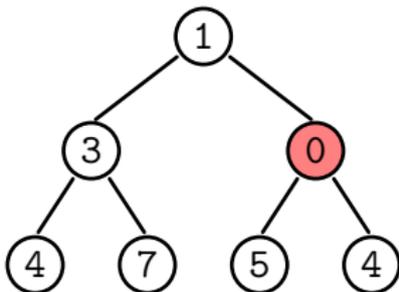
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 0



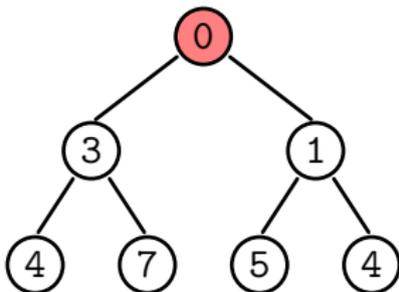
1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

exemple : on ajoute 0



1. on l'ajoute « en bas »
2. on le fait « **remonter** » autant que nécessaire

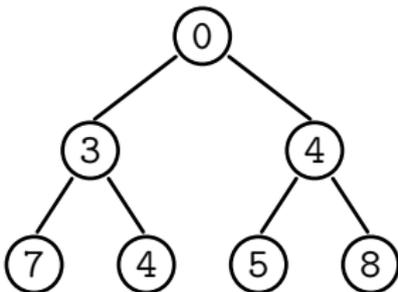
exemple : on ajoute 0



supprimer le plus petit élément

1. on remplace la racine par « l'élément en bas à droite »
2. on le fait « **descendre** » autant que nécessaire

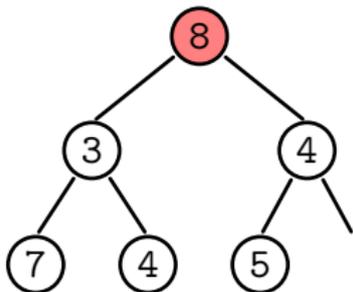
exemple :



supprimer le plus petit élément

1. on remplace la racine par « l'élément en bas à droite »
2. on le fait « **descendre** » autant que nécessaire

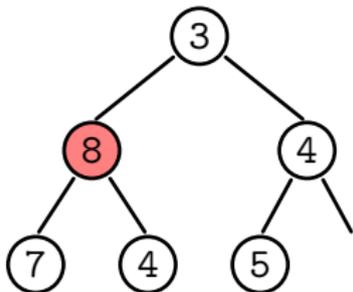
exemple :



supprimer le plus petit élément

1. on remplace la racine par « l'élément en bas à droite »
2. on le fait « **descendre** » autant que nécessaire

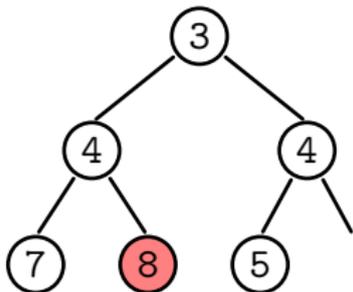
exemple :



supprimer le plus petit élément

1. on remplace la racine par « l'élément en bas à droite »
2. on le fait « **descendre** » autant que nécessaire

exemple :



on pourrait construire explicitement un arbre (comme pour les ABR)
mais c'est inutile

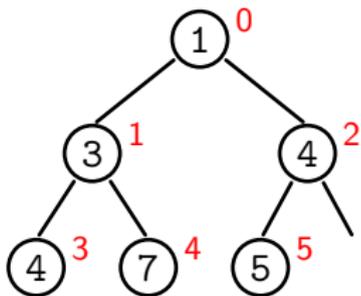
on conserve en permanence un **arbre binaire (presque) parfait** :
tous les niveaux sont complets, sauf peut-être le dernier

un tel arbre est très facilement stocké **dans un tableau**

a

1	3	4	4	7	5
---	---	---	---	---	---

 ...



$$\text{racine} = 0$$

$$\text{gauche}(i) = 2i + 1$$

$$\text{droit}(i) = 2i + 2$$

$$\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$$

en supposant que les éléments sont ici des entiers

```
class PriorityQueue {  
    private int[] elts;  
    private int size;
```

le constructeur définit la capacité maximale

```
PriorityQueue(int capacity) {  
    this.elts = new int[capacity];  
    this.size = 0;           // initialement vide  
}
```

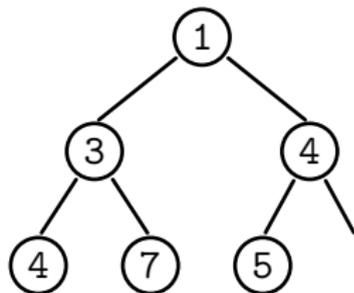
```
int getMin() {  
    if (isEmpty()) throw new NoSuchElementException();  
    return this.elts[0];  
}
```

complexité $O(1)$

ajout d'un élément

ajouter x à la position i et le faire monter

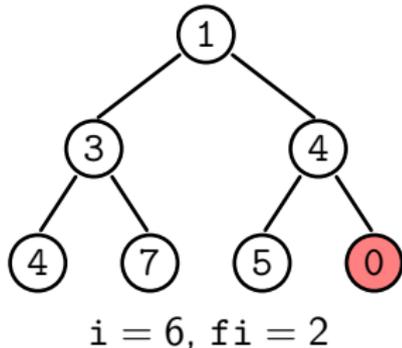
```
private void moveUp(int x, int i) {  
    while (i > 0) {  
        int fi = (i - 1) / 2;  
        int y = this.elts[fi];  
        if (y <= x) break;  
        this.elts[i] = y;  
        i = fi;  
    }  
    this.elts[i] = x;  
}
```



(version récursive dans le poly)

ajouter x à la position i et le faire monter

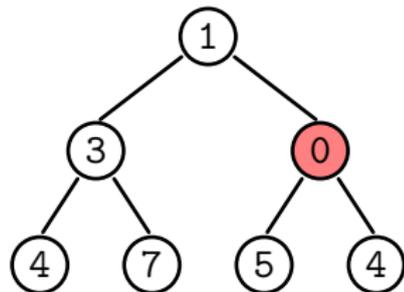
```
private void moveUp(int x, int i) {  
    while (i > 0) {  
        int fi = (i - 1) / 2;  
        int y = this.elts[fi];  
        if (y <= x) break;  
        this.elts[i] = y;  
        i = fi;  
    }  
    this.elts[i] = x;  
}
```



(version récursive dans le poly)

ajouter x à la position i et le faire monter

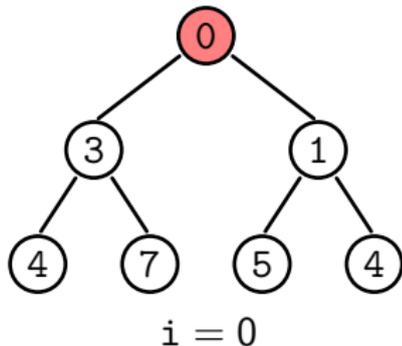
```
private void moveUp(int x, int i) {  
    while (i > 0) {  
        int fi = (i - 1) / 2;  
        int y = this.elts[fi];  
        if (y <= x) break;  
        this.elts[i] = y;  
        i = fi;  
    }  
    this.elts[i] = x;  
}
```



(version récursive dans le poly)

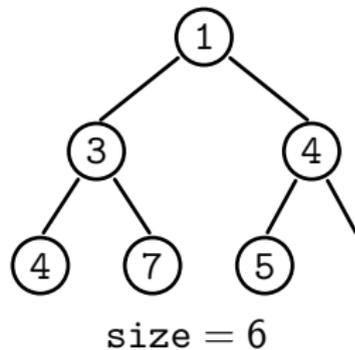
ajouter x à la position i et le faire monter

```
private void moveUp(int x, int i) {  
    while (i > 0) {  
        int fi = (i - 1) / 2;  
        int y = this.elts[fi];  
        if (y <= x) break;  
        this.elts[i] = y;  
        i = fi;  
    }  
    this.elts[i] = x;  
}
```

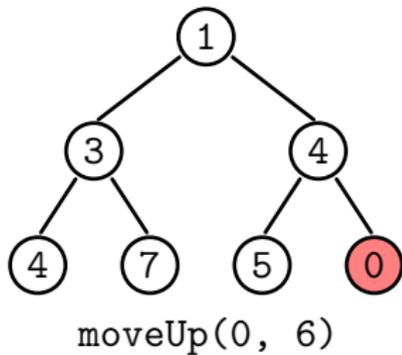


(version récursive dans le poly)

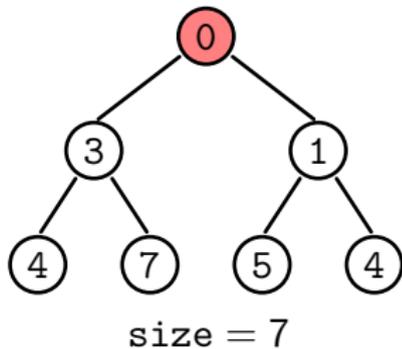
```
void add(int x) {  
    moveUp(x, this.size++);  
}
```



```
void add(int x) {  
    moveUp(x, this.size++);  
}
```



```
void add(int x) {  
    moveUp(x, this.size++);  
}
```



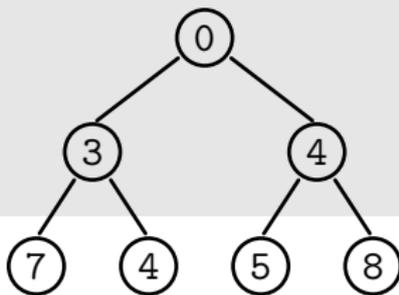
le coût n'excède pas la hauteur de l'arbre, c'est-à-dire $\log N$

en effet, l'arbre est toujours équilibré, par construction

supprimer le plus petit élément

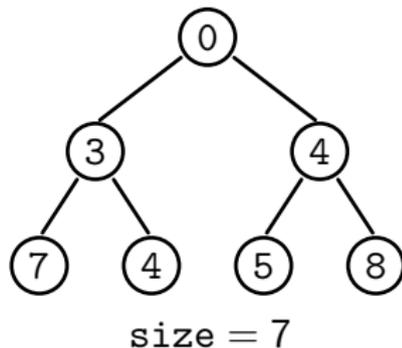
ajouter x à la position i et le faire descendre

```
private void moveDown(int x, int i) {  
    int n = this.size;  
    while (true) {  
        int j = 2 * i + 1;  
        if (j >= n) break;  
        if (j + 1 < n && this.elts[j] > this.elts[j + 1]) j++;  
        if (x <= this.elts[j]) break;  
        this.elts[i] = this.elts[j];  
        i = j;  
    }  
    this.elts[i] = x;  
}
```

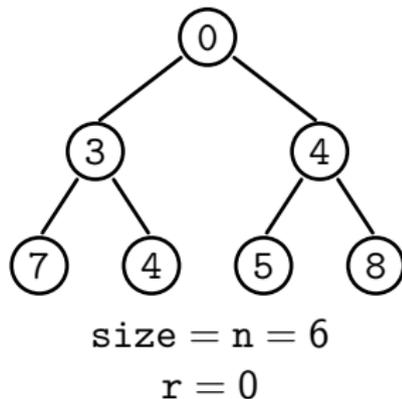


(version récursive dans le poly)

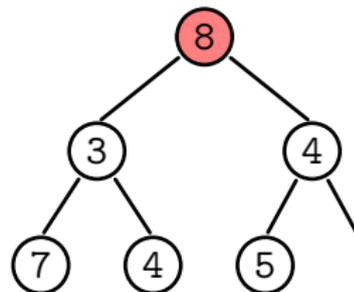
```
int removeMin() {  
    if (isEmpty()) throw new NoSuch...;  
    int n = --this.size;  
    int r = this.elts[0];  
    if (n > 0)  
        moveDown(this.elts[n], 0);  
    return r;  
}
```



```
int removeMin() {  
    if (isEmpty()) throw new NoSuch...;  
    int n = --this.size;  
    int r = this.elts[0];  
    if (n > 0)  
        moveDown(this.elts[n], 0);  
    return r;  
}
```



```
int removeMin() {  
    if (isEmpty()) throw new NoSuch...;  
    int n = --this.size;  
    int r = this.elts[0];  
    if (n > 0)  
        moveDown(this.elts[n], 0);  
    return r;  
}
```

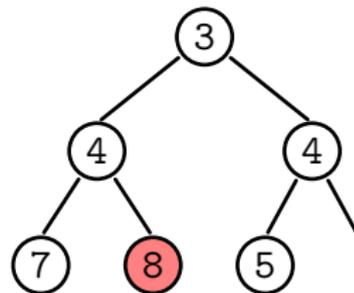


size = n = 6

r = 0

moveDown(8, 0)

```
int removeMin() {  
    if (isEmpty()) throw new NoSuch...;  
    int n = --this.size;  
    int r = this.elts[0];  
    if (n > 0)  
        moveDown(this.elts[n], 0);  
    return r;  
}
```



size = n = 6

r = 0

moveDown(8, 0)

le coût n'excède pas la hauteur de l'arbre, c'est-à-dire $\log N$

pour bien faire

on écrit un code **générique**

```
class PriorityQueue<E extends Comparable<E>> {  
    ...  
}
```

(comme pour les arbres binaires de recherche ; cf amphi 5)

on ne limite pas la capacité de la file de priorité
en utilisant un **tableau redimensionnable** (cf amphi 1)

```
class PriorityQueue<E ...> {  
    private Vector<E> elts;  
  
    PriorityQueue() {  
        this.elts = new Vector<E>();  
    }  
  
    ...  
}
```

on trouve

```
java.util.PriorityQueue<E>
```

soit la classe E implémente Comparable<E>

```
PriorityQueue()  
PriorityQueue(int initialCapacity)
```

soit on fournit un comparateur

```
PriorityQueue(int initialCapacity,  
              Comparator<E> comparator)
```

retour sur le problème initial

```
class Title implements Comparable<Title> {
    final String title;

    Title(String title) { this.title = title; }

    public int compareTo(Title that) {
        return this.title.length() - that.title.length();
    }

    public String toString() { return this.title; }
}
```

```
int M = 100;
Scanner sc = new Scanner(new File("fr-pages.txt"));
PriorityQueue<Title> q = new PriorityQueue<Title>();

while (sc.hasNext()) {
    String s = sc.next();
    q.add(new Title(s));
    if (q.size() > M)
        q.remove();
}

while (!q.isEmpty())
    System.out.println(q.remove().title);
```

autre application

on peut se servir d'une file de priorité **pour trier**

1. on met tous les éléments dans une file
2. on les ressort dans l'ordre, avec `removeMin`

exemple avec un tableau d'entiers

```
static void heapsort(int[] a) {  
    PriorityQueue<Integer> h = new PriorityQueue<>(a.length);  
    for (int x: a) h.add(x);  
    for (int i = 0; i < a.length; i++) a[i] = h.removeMin();  
}
```

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

4	0	0	0	0	0	0
---	---	---	---	---	---	---

④

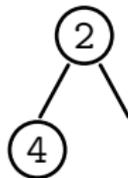
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

2	4	0	0	0	0	0
---	---	---	---	---	---	---



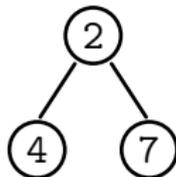
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

2	4	7	0	0	0	0
---	---	---	---	---	---	---



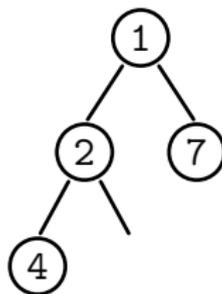
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

1	2	7	4	0	0	0
---	---	---	---	---	---	---



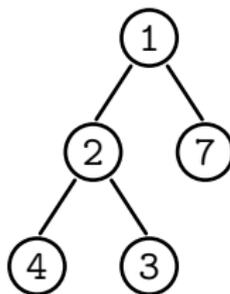
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

1	2	7	4	3	0	0
---	---	---	---	---	---	---



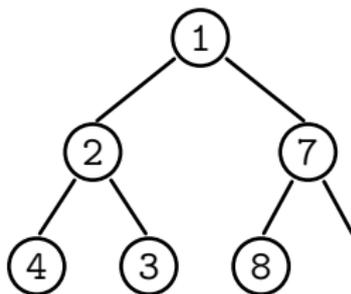
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

1	2	7	4	3	8	0
---	---	---	---	---	---	---



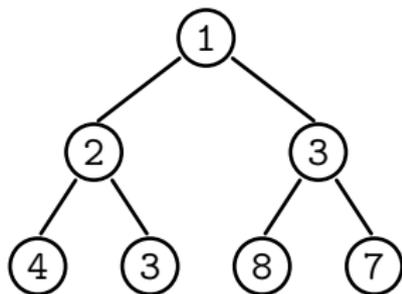
construction

a

4	2	7	1	3	8	3
---	---	---	---	---	---	---

h

1	2	3	4	3	8	7
---	---	---	---	---	---	---



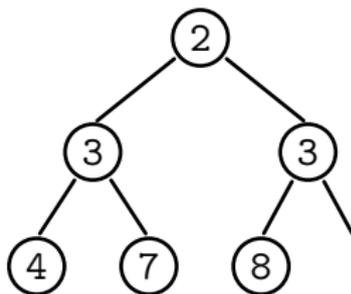
déconstruction

a

1	2	7	1	3	8	3
---	---	---	---	---	---	---

h

2	3	3	4	7	8	7
---	---	---	---	---	---	---



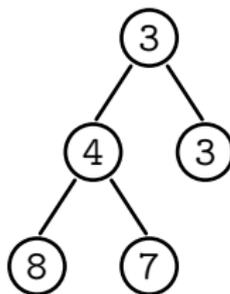
déconstruction

a

1	2	7	1	3	8	3
---	---	---	---	---	---	---

h

3	4	3	8	7	8	7
---	---	---	---	---	---	---



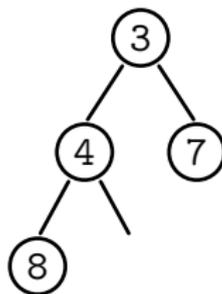
déconstruction

a

1	2	3	1	3	8	3
---	---	---	---	---	---	---

h

3	4	7	8	7	8	7
---	---	---	---	---	---	---



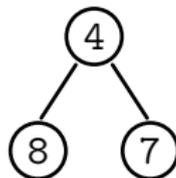
déconstruction

a

1	2	3	3	3	8	3
---	---	---	---	---	---	---

h

4	8	7	8	7	8	7
---	---	---	---	---	---	---



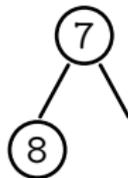
déconstruction

a

1	2	3	3	4	8	3
---	---	---	---	---	---	---

h

7	8	7	8	7	8	7
---	---	---	---	---	---	---



déconstruction

a

1	2	3	3	4	7	3
---	---	---	---	---	---	---

h

8	8	7	8	7	8	7
---	---	---	---	---	---	---

⑧

déconstruction

a

1	2	3	3	4	7	8
---	---	---	---	---	---	---

h

8	8	7	8	7	8	7
---	---	---	---	---	---	---

la première phase (construction du tas) coûte

$$\log 1 + \log 2 + \cdots + \log(N - 1) \sim N \log N$$

la seconde phase (déconstruction du tas) coûte

$$\log N + \log(N - 1) + \cdots + \log 1 \sim N \log N$$

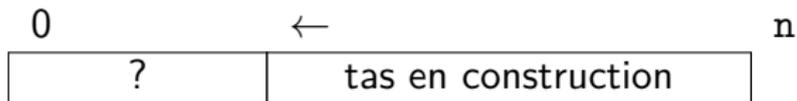
on obtient donc un algorithme de tri en $O(N \log N)$

nous verrons la semaine prochaine que c'est **optimal**

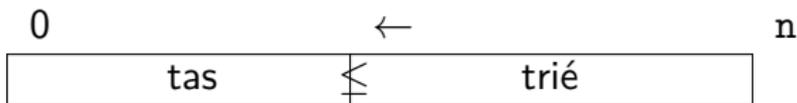
en **espace**, cependant, on utilise un tableau supplémentaire de taille N
peut-on faire mieux ?

on peut réaliser le tri par tas **en place**,
c'est-à-dire dans le tableau lui-même

1. construction



2. déconstruction



note : **plus grand** élément à la racine (l'ordre est inversé)

ajouter x à la position i et le faire descendre dans le tas $a[0..n[$

```
static void moveDown(int[] a, int i, int x, int n) {
    while (true) {
        int j = 2 * i + 1;
        if (j >= n) break;
        if (j + 1 < n && a[j] < a[j + 1]) j++;
        if (a[j] <= x) break;
        a[i] = a[j];
        i = j;
    }
    a[i] = x;
}
```

c'est la même méthode que précédemment (seul l'ordre est inversé)

```
static void heapsort(int[] a) {
    int n = a.length;

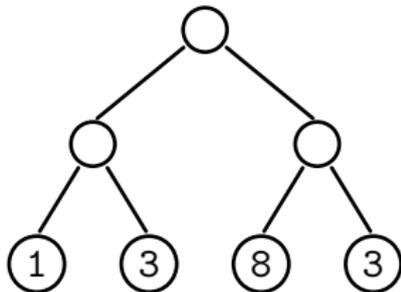
    // phase 1 : construction du tas
    for (int k = n / 2 - 1; k >= 0; k--)
        moveDown(a, k, a[k], n);

    // phase 2 : déconstruction du tas
    for (int k = n - 1; k >= 1; k--) {
        int v = a[k];
        a[k] = a[0];
        moveDown(a, 0, v, k);
    }
}
```

```
static void heapsort(int[] a) {  
    int n = a.length;  
  
    // phase 1 : construction du tas  
    for (int k = n / 2 - 1; k >= 0; k--)  
        moveDown(a, k, a[k], n);  
}
```

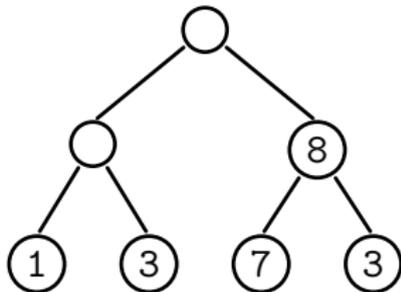
a

4	2	7	1	3	8	3
---	---	---	---	---	---	---



```
static void heapsort(int[] a) {  
    int n = a.length;  
  
    // phase 1 : construction du tas  
    for (int k = n / 2 - 1; k >= 0; k--)  
        moveDown(a, k, a[k], n);  
}
```

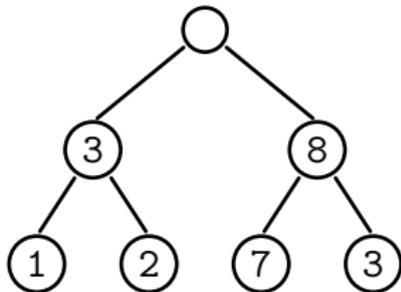
a [4 | 2 | 8 | 1 | 3 | 7 | 3]



```
static void heapsort(int[] a) {  
    int n = a.length;  
  
    // phase 1 : construction du tas  
    for (int k = n / 2 - 1; k >= 0; k--)  
        moveDown(a, k, a[k], n);  
}
```

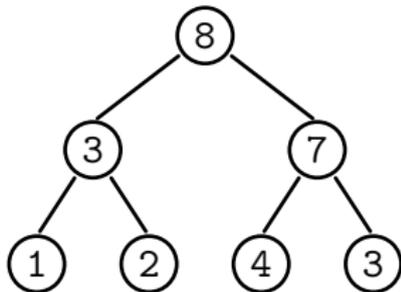
a

4	3	8	1	2	7	3
---	---	---	---	---	---	---



```
static void heapsort(int[] a) {  
    int n = a.length;  
  
    // phase 1 : construction du tas  
    for (int k = n / 2 - 1; k >= 0; k--)  
        moveDown(a, k, a[k], n);  
}
```

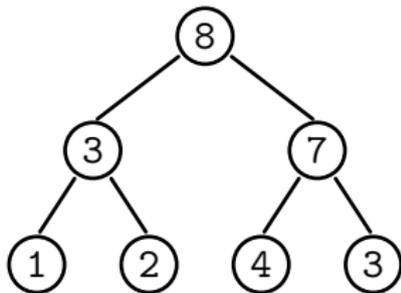
a 8 3 7 1 2 4 3



```
// phase 2 : déconstruction du tas  
for (int k = n - 1; k >= 1; k--) {  
    int v = a[k];  
    a[k] = a[0];  
    moveDown(a, 0, v, k);  
}
```

a

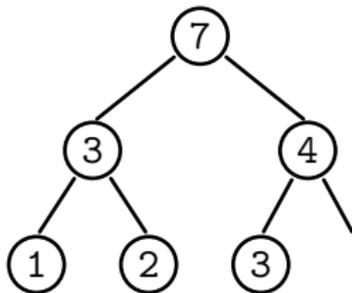
8	3	7	1	2	4	3
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas  
for (int k = n - 1; k >= 1; k--) {  
    int v = a[k];  
    a[k] = a[0];  
    moveDown(a, 0, v, k);  
}
```

a

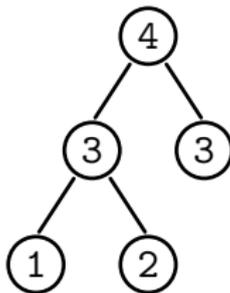
7	3	4	1	2	3	8
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas  
for (int k = n - 1; k >= 1; k--) {  
    int v = a[k];  
    a[k] = a[0];  
    moveDown(a, 0, v, k);  
}
```

a

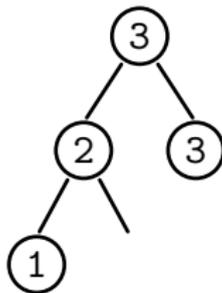
4	3	3	1	2	7	8
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas
for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
}
```

a

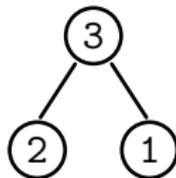
3	2	3	1	4	7	8
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas
for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
}
```

a

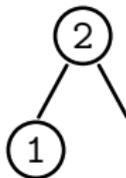
3	2	1	3	4	7	8
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas
for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
}
```

a

2	1	3	3	4	7	8
---	---	---	---	---	---	---



```
// phase 2 : déconstruction du tas
for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
}
```

a

1	2	3	3	4	7	8
---	---	---	---	---	---	---

 ①

```
// phase 2 : déconstruction du tas
for (int k = n - 1; k >= 1; k--) {
    int v = a[k];
    a[k] = a[0];
    moveDown(a, 0, v, k);
}
```

a

1	2	3	3	4	7	8
---	---	---	---	---	---	---

chaque appel à `moveDown` coûte au pire $O(\log N)$

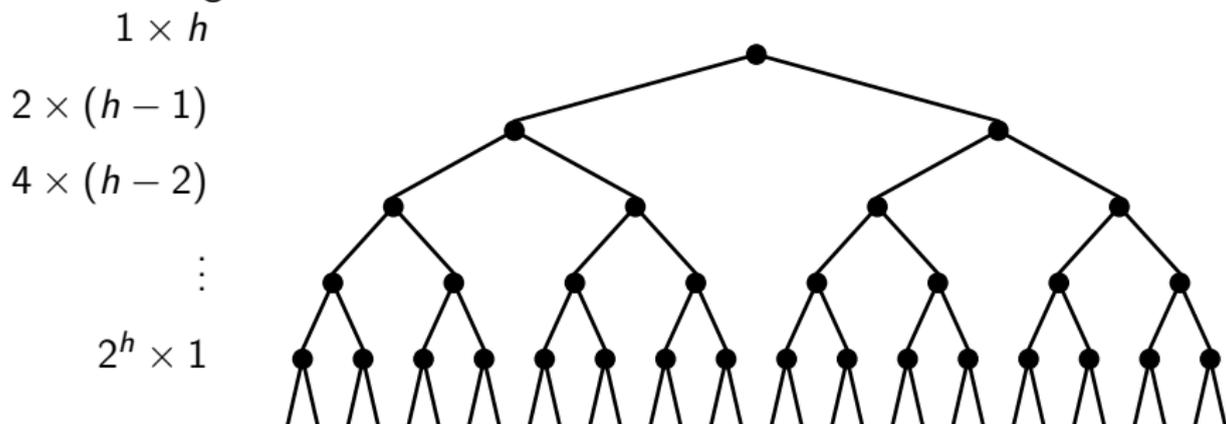
la phase 1 fait $N/2$ appels à `moveDown`

la phase 2 fait $N - 1$ appels à `moveDown`

d'où un coût total $O(N \log N)$, optimal

on peut être plus précis en ce qui concerne la phase 1

posons $h = \log N$



$$\begin{aligned} C &\leq \sum_{i=0}^h 2^{h-i} \times i \\ &= 2^h \sum_{i=0}^h \frac{i}{2^i} \\ &\leq 2^h \sum_{i=0}^{\infty} \frac{i}{2^i} \\ &= 2^h \frac{1/2}{(1 - 1/2)^2} \\ &= 2^h \times 2 \\ &= 2N \end{aligned}$$

la construction d'un tas à partir de N valeurs se fait en $O(N)$

```
class PriorityQueue<E> {  
    PriorityQueue<E>() // nouvelle file, vide  
    PriorityQueue<E>(Collection<E> c) // en temps  $O(N)$   
    ...  
}
```

file de priorité = structure offrant

- `getMin` en $O(1)$
- `add` et `removeMin` en $O(\log N)$

nombreuses applications, en particulier

- priorités dans un système multitâche
- tri optimal en $O(N \log N)$
- algorithme de Dijkstra (amphi 10)
- algorithme de Kruskal (dans le poly)
- algorithme de Huffman

algorithme de Huffman

transformer un texte en une suite de 0 et de 1

mississippi → 100011110111101011010

ici, on a choisi

i \rightarrow 0

s \rightarrow 11

m \rightarrow 100

p \rightarrow 101

c'est un **code préfixe** : aucune séquence n'est le préfixe d'une autre

si les caractères sont les feuilles d'un arbre binaire



on obtient un code préfixe avec 0 = gauche et 1 = droite

(c'est un arbre de préfixes ; cf amphi 6)

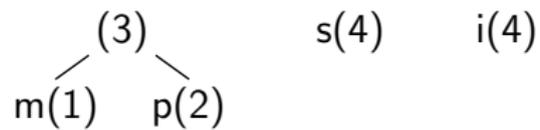
l'algorithme de Huffman construit un tel arbre,
étant donné un nombre d'occurrences pour chaque caractère

1. initialement, chaque caractère est une feuille avec sa fréquence
2. tant qu'il reste au moins deux arbres
 - sélectionner les deux arbres avec les plus petites fréquences
 - les réunir et ajouter les fréquences

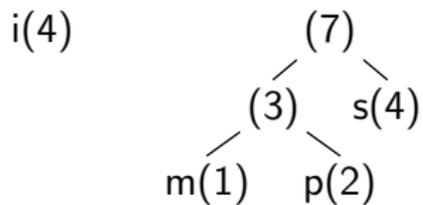
avec le texte mississippi

m(1) p(2) s(4) i(4)

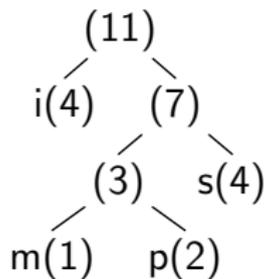
avec le texte mississippi



avec le texte mississippi



avec le texte mississippi



si chaque caractère c_i apparaît avec la fréquence f_i ,
l'arbre de Huffman minimise

$$S = \sum_i f_i \times d_i$$

où d_i est la profondeur du caractère c_i

c'est-à-dire la longueur du texte encodé
(preuve dans le poly, page 164)

on va se servir d'une **file de priorité contenant des arbres**, ordonnés selon la fréquence

```
class HuffmanTree implements Comparable<HuffmanTree> {  
    int freq;  
  
    public int compareTo(HuffmanTree that) {  
        return this.freq - that.freq;  
    }  
}
```

comme la semaine dernière, on peut

1. faire de la classe HuffmanTree une classe abstraite

```
abstract class HuffmanTree ...
```

2. avec deux sous-classes, une pour les feuilles

```
class Leaf extends HuffmanTree {  
    final char c;  
    ...  
}
```

et l'autre pour les nœuds

```
class Node extends HuffmanTree {  
    final HuffmanTree left, right;  
    ...  
}
```

le cœur de l'algorithme

```
PriorityQueue<HuffmanTree> pq = new PriorityQueue<>();  
for (Leaf l: alphabet)  
    pq.add(l);
```

```
while (pq.size() > 1) {  
    HuffmanTree left = pq.remove();  
    HuffmanTree right = pq.remove();  
    pq.add(new Node(left, right));  
}
```

voir le poly pour

- le calcul des fréquences (exercice 97)
- le codage du texte
- le décodage

- **lire le poly**
 - chapitre 8 Files de priorité
 - chapitre 14 Algorithme de Huffman

il y a des **exercices** dans le poly
suggestion : ex 97 p 169

- **bloc 8** : tri

- conditions habituelles d'un TD
mais pas d'accès à Internet
- programme = blocs 1 à 6
- **polycopié autorisé**
- dictionnaire électronique autorisé

penser à amener une feuille et un crayon