

# Structures de données et algorithmes – TP1 1

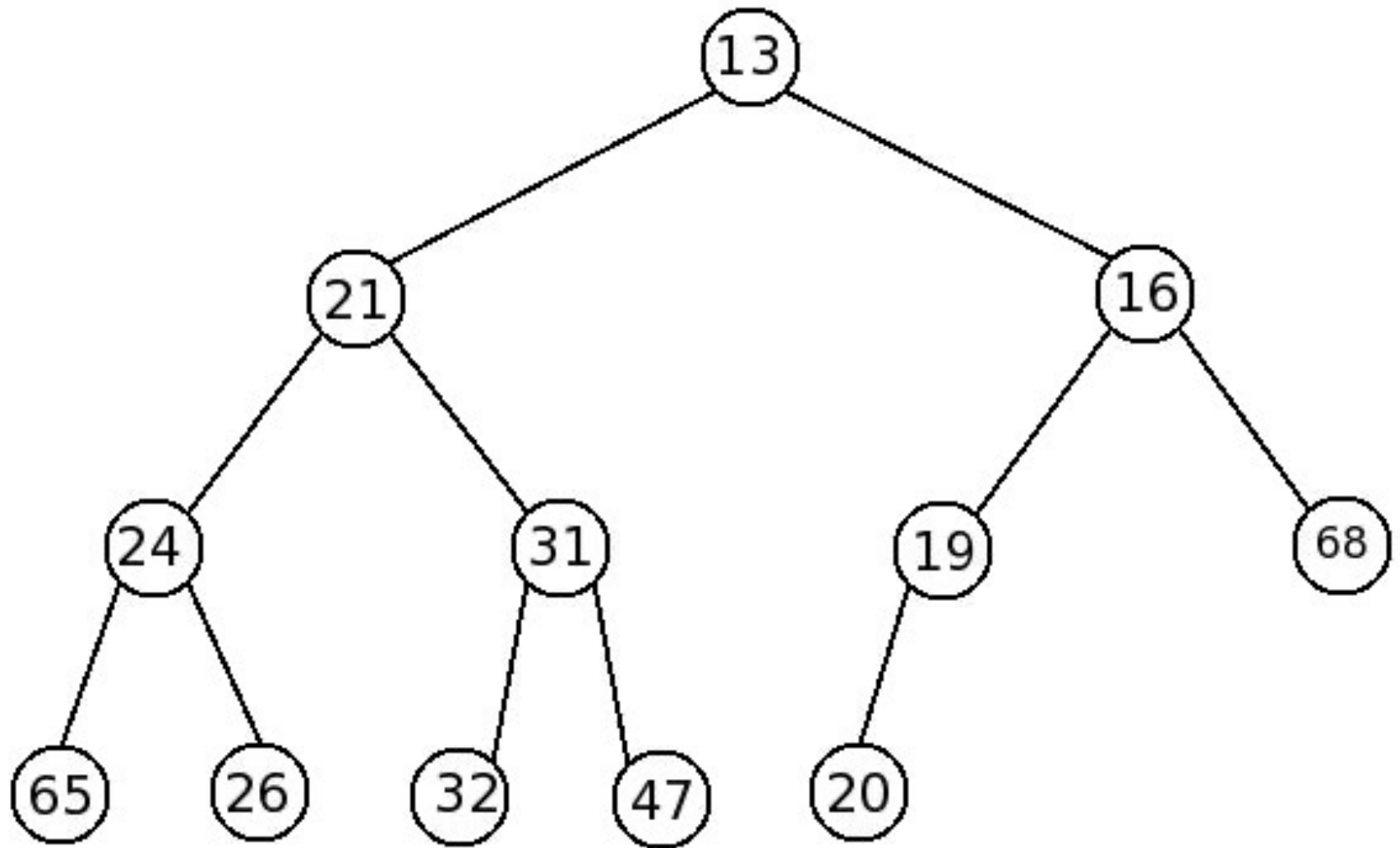
Iulia-Cristina Stanica

# Objectifs pour aujourd'hui

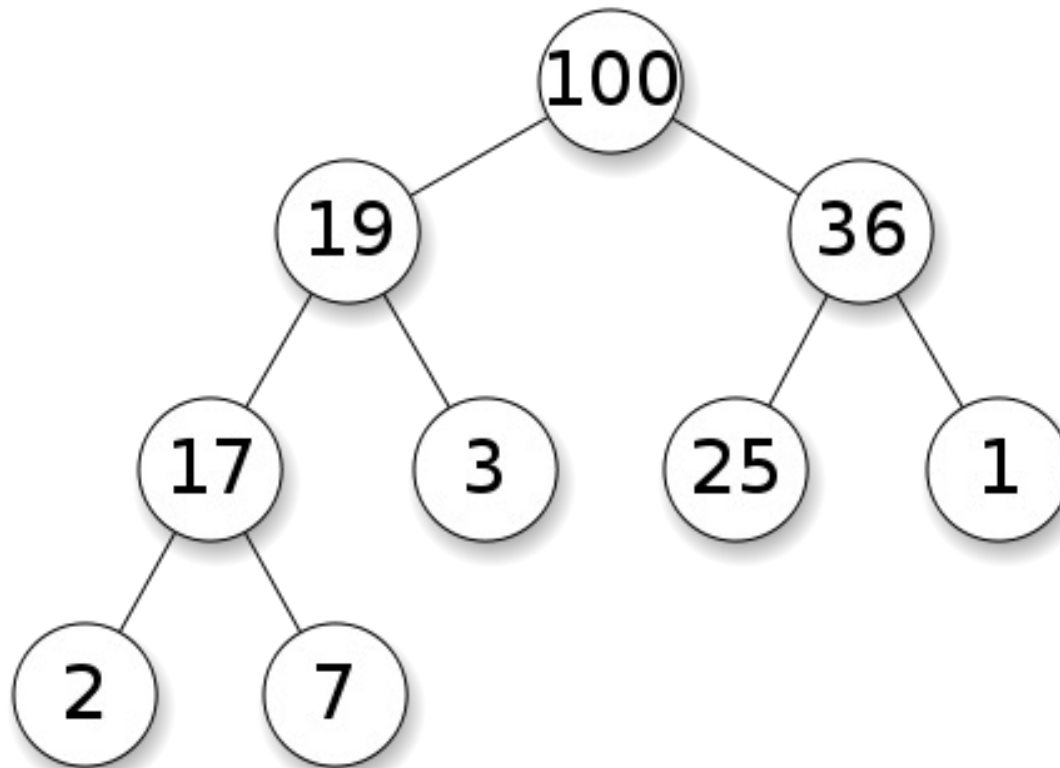
## Tas (Heap)

- ▶ Tas (Heap)
- ▶ Tri par tas (Heap sort)
- ▶ Implémentation en C++

# Min heap: lowest at top



# MAX Heap



# Exercice 1

- ▶ Insérer dans un min-tas vide les numéros suivants: 25, 17, 36, 2, 3, 100, 1, 17, 19
- ▶ Supprimer 2 éléments du tas.
- ▶ (exercice sur papier)

# Ex 2

- ▶ Tester l'implémentation du Heap en insérant les mêmes éléments de l'exercice 1: 25, 17, 36, 2, 3, 100, 1, 17, 19. Vérifier les résultats de l'effacement.
- ▶ Ajoutez à la classe Heap les fonctions de liaisons entre le parent et ses enfants (formules diapo 19). Vérifiez les valeurs pour l'exercice 1.

Chaque fonction a la structure:

```
int Parent( int CI)
{
    //opérations en fonction de CI
}
```

## Ex 3

- ▶ Ajoutez une fonction pour afficher le tas par niveaux. Verifier le resultat pour le tas de l'exercice 1.

# Tri par tas (max tas) – Heap Sort

- Le tri par tas est un algorithme de tri par comparaisons d'un tableau.
- L'idée qui sous-tend cet algorithme consiste à voir le tableau comme un arbre binaire. Le premier élément est la racine, le deuxième et le troisième sont les deux descendants du premier élément, etc.
- Dans l'algorithme, on cherche à construire un tas (vérifier toutes ses propriétés)
- Effacer la racine plusieurs fois et mettez-la sur la dernière place libre du tableau. Ajustez le tas.
- Pour un max tas on va obtenir un tri en ordre croissant.

10	4	8	5	12	2	6	11	3	9	7	1
----	---	---	---	----	---	---	----	---	---	---	---

[https://commons.wikimedia.org/wiki/  
File:Heap\\_sort\\_example.gif](https://commons.wikimedia.org/wiki/File:Heap_sort_example.gif)



# Ex 4

- ▶ Considérer le tableau suivant:  
25, 17, 36, 2, 3, 100, 1, 17, 19.
- ▶ Trier le tableau à l'aide de tri par tas (ordre décroissante).

## ► Hint – Etapes (possible solution)

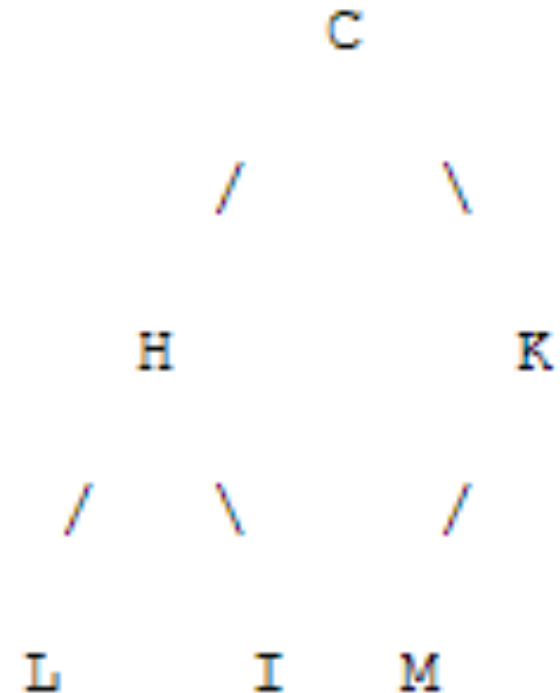
1. Dans un nouveau constructeur (avec les paramètres: dimension et tableau), on va convertir le tableau d'éléments dans un tas.
  - 1.1. On a premièrement un arbre binaire. D'abord aller à l'indice du dernier nœud qui peut être un parent (Notons « ind ») – **Lequel?**
  - 1.2. Appliquer routine FilterDown à chaque nœud entre cet indice ind et 0 (hint: **modifier le filterDown pour recevoir i comme parametre**).
2. Créer une fonction HeapSort dans laquelle vous trie le tableau en utilisant le tas (Par ex: extraire la racine plusieurs fois et la mettre sur la premiere position libre dans le tableau – a la fin!). Le tableau sera le parametre.
3. Créer un tableau dans le main et lisez ses valeurs du clavier. Créer un tas et tester le tri du tableau.

# Support theorique

# Arbre binaire presque complet

► **Def:** Un arbre binaire presque complet est un arbre binaire dans lequel les 3 conditions suivantes sont réunies:

- toutes les feuilles sont sur le dernier niveau (ou les derniers 2 niveaux) de l'arbre
- toutes les feuilles sont dans les positions les plus gauches possibles
- tous les niveaux sont complètement remplis par des nœuds (sauf éventuellement le dernier niveau)



# Tas (Heap)

- ▶ **Def:** Un *tas* minimale (décroissant) est un arbre binaire presque complet dans lequel la valeur du chaque nœud parent est inférieure ou égale aux valeurs de ses nœuds enfants.
- ▶ **Observations:**
  - La valeur minimale se trouve dans le nœud racine.
  - Un chemin à partir d'une feuille à la racine passe à travers les données dans l'ordre décroissant.

# Operations

- ▶ Insertion
  - ▶ Effacement
  - ▶ Obtenir la racine
- 

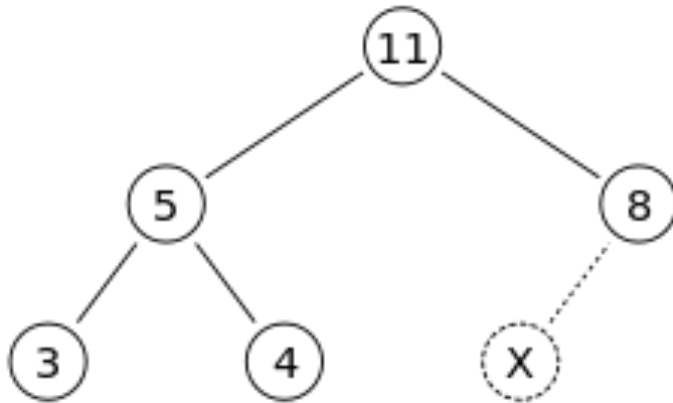
# 1. Insertion

## ► Etapes:

- 1. Ajouter l'élément sur la dernière place disponible du tas (**dernier niveau, le plus gauche possible**)
- 2. Comparer l'élément ajouté avec son parent; si l'ordre est correcte, arrêter
- 3. Sinon, changer l'élément et son parent et retourner au pas précédent

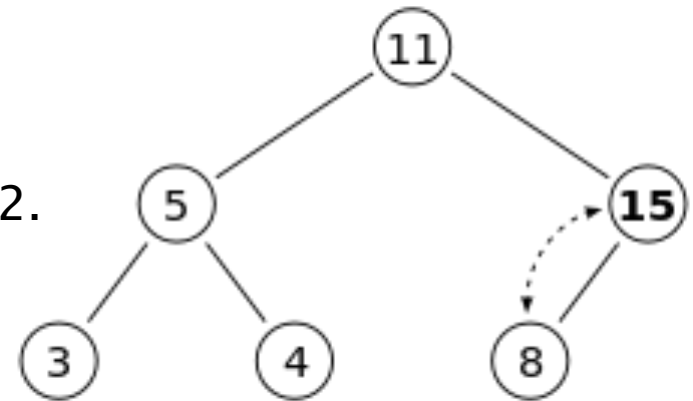
# Exemple (max heap):

1.



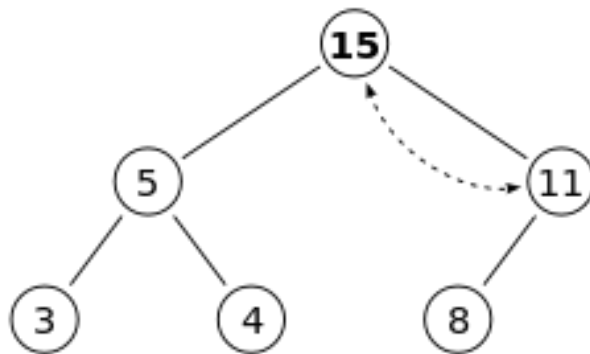
On va ajouter 15 (a la place de X)

2.



On compare 15 avec son parent et on fait l'interchangement

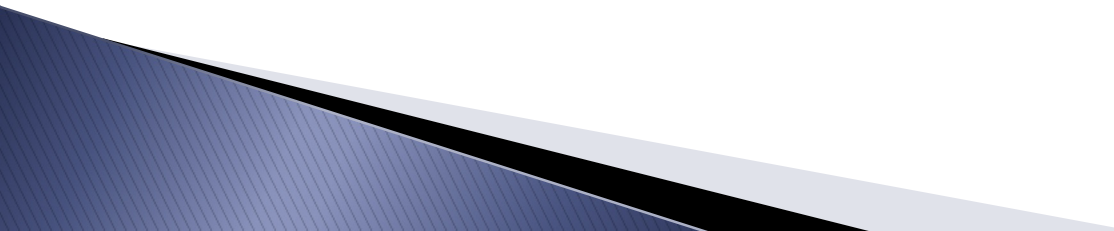
3.



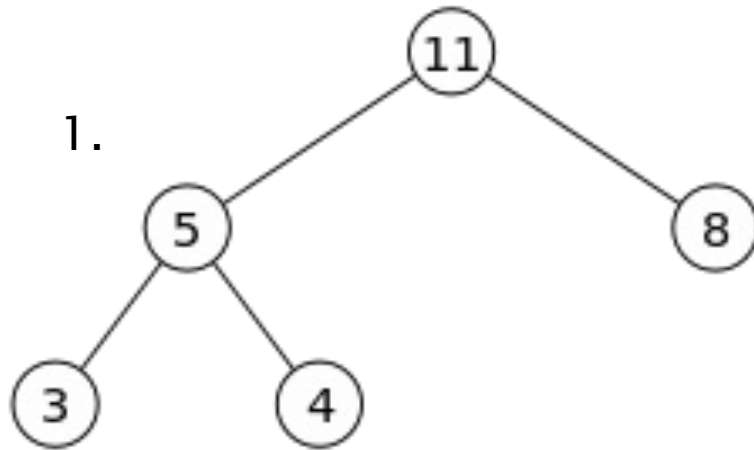
Meme operation entre 15 et la racine.



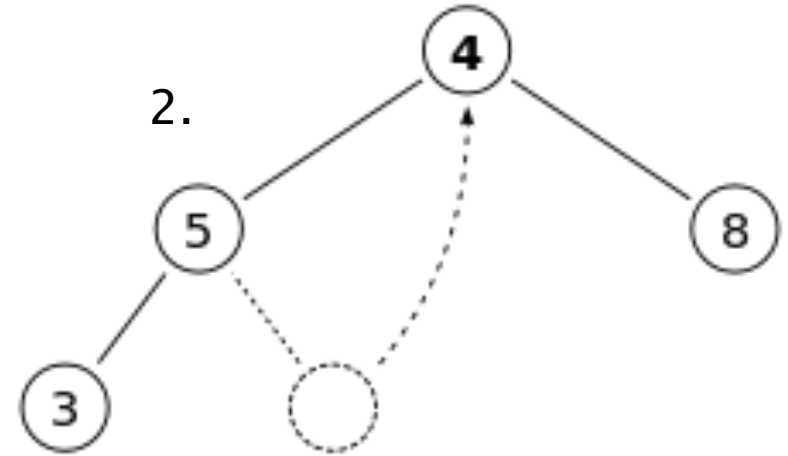
## 2. Effacement

- ▶ On efface toujours la racine du tas.
  - ▶ Etapes:
    - 1. Remplacer temporairement la racine avec le **dernier élément du dernier niveau**
    - 2. Comparer la nouvelle racine avec ses enfants; si l'ordre est correcte, arrêter
    - 3. Sinon, changer l'élément avec son (plus grand) enfant et retourner au pas précédent
- 

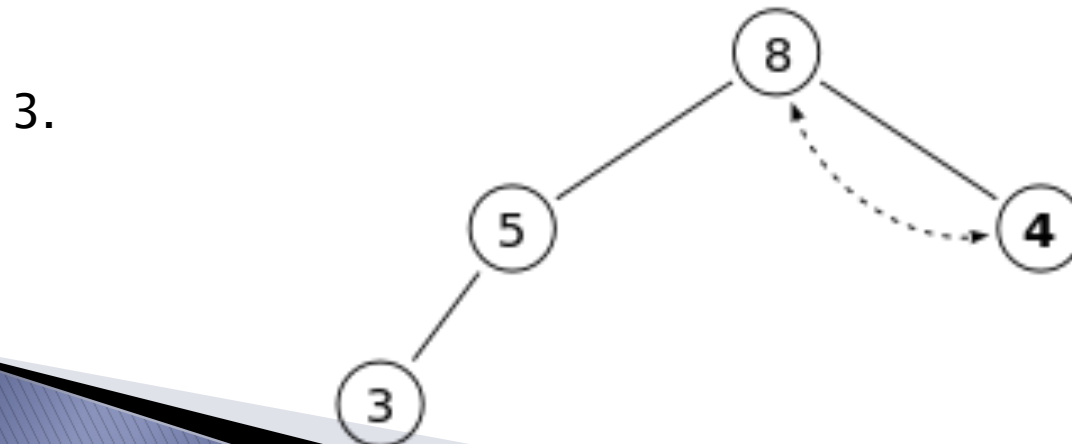
# Exemple (max heap):



On efface la racine (11)



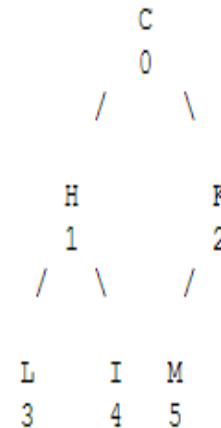
On met le dernier enfant a sa place



On met le plus grand enfant de la racine a sa place.

# Mise en œuvre des données de type Heap/Tas

- ▶ comme un arbre binaire
- ▶ comme un tableau
  - définir les numéros des nœuds en fonction du niveau du haut vers le bas, de gauche à droite
  - stocker les données dans un tableau
  - si CI est l'indice actuel:
    - $\text{Parent}(\text{CI}) = (\text{CI} - 1) / 2$
    - $\text{RightChild}(\text{CI}) = 2 * (\text{CI} + 1)$
    - $\text{LeftChild}(\text{CI}) = 2 * \text{CI} + 1$



C	H	K	L	I	M
0	1	2	3	4	5

CI pour H = 1 => RightChild pour H (CI) =  $2 * (1 + 1) = 4$  => I

# Exo extra

- ▶ Mettre en œuvre un algorithme pour trouver le plus grand element dans un min-tas.
  - La propriété du min heap nécessite que le nœud parent soit inférieur à son ou ses nœuds enfants. De ce fait, nous pouvons en conclure qu'un nœud non-feuille ne peut pas être l'élément maximal car son nœud enfant a une valeur supérieure.

