ÉCOLE NATIONALE SUPÉRIEURE D'ARTS ET MÉTIERS
UNIVERSITÉ HASSAN II DE CASABLANCA

Algorithmique Avancée

Tri par Fusion
&
Tri par Tas

(Merge sort & Heap Sort)

Animé par : Dr. ibrahim GUELZIM
Email : ib.guelzim@gmail.com

Sommaire

- Rappels
  - Introduction et notions générales
  - Analyse et conception d'algorithmes
  - Complexité d'algorithmes classiques : 3 Tris de tableaux, 2 recherches dans un tableau, Schéma de Hörner
  - Preuves d'algorithmes
- **Autres algorithmes de tri :**
  - **Tri par fusion**
  - **Tri par Tas**
- Complexité moyenne :
  - Application au Tri rapide
  - Structures de Données Probabilistes :
    - Notions sur les Tables de Hachage et Fonctions de Hachage,
    - Bloom Filter,
    - Count Min Sketch

– Programmation dynamique

- Traitements de chaines de Caractères :
  - Recherche de chaine de caractères
  - Compression de données

Partie 1 : Tri par Fusion

- Fusion de deux tableaux triés
- Algorithme du Tri par fusion
- Complexité

Fusion de 2 tableaux **triés**

- Tableaux : B et C

6	8	12	16

2	5	7	17	19

- Tableau Auxiliaire : Aux

- **Idée: Répéter**
  - Comparer les éléments "pointés" de **B** et de **C**
  - Mettre l'élément adéquat (**min** si tri **croissant**) à la position adéquate du tableau auxiliaire **Aux**
  - Mettre à jour les "pointeurs" (**indices**) des tableaux concernés.

Fusion de 2 sous tableaux triés

```
Fonction Fusionner(a[] : Entier, lo : Entier, mid : Entier, hi : Entier)
// Fusionner a[lo, mid) avec a[mid, hi) (déjà triés) dans aux[0, hi-lo)
Variable i, j, k, N, aux[] : Entier
Début
    i ← lo, j ← mid, N ← hi - lo
    Pour k allant de 0 à N-1 faire
        Si(i = mid) // vérification débordement tableau à gauche
         aux[k] = a[j], j ← j + 1
        Sinon si (j = hi) // vérification débordement tableau à droite
                aux[k] ← a[i], i ← i + 1
              Sinon si (a[i] > a[j])
                        aux[k] ← a[j], j ← j + 1
                    Sinon
                        aux[k] ← a[i], i ← i + 1
                    FinSi
              FinSi
        FinSi
    FinPour
    Pour k allant de 0 à N-1 faire // Recopier dans a[lo, hi)
         a[lo + k] ← aux[k]
    FinPour
Fin
```

Partie 2 : Tri par Fusion

- Fusion de deux tableaux triés
- Algorithme du Tri par fusion
- Complexité

Tri par Fusion (Merge-Sort)

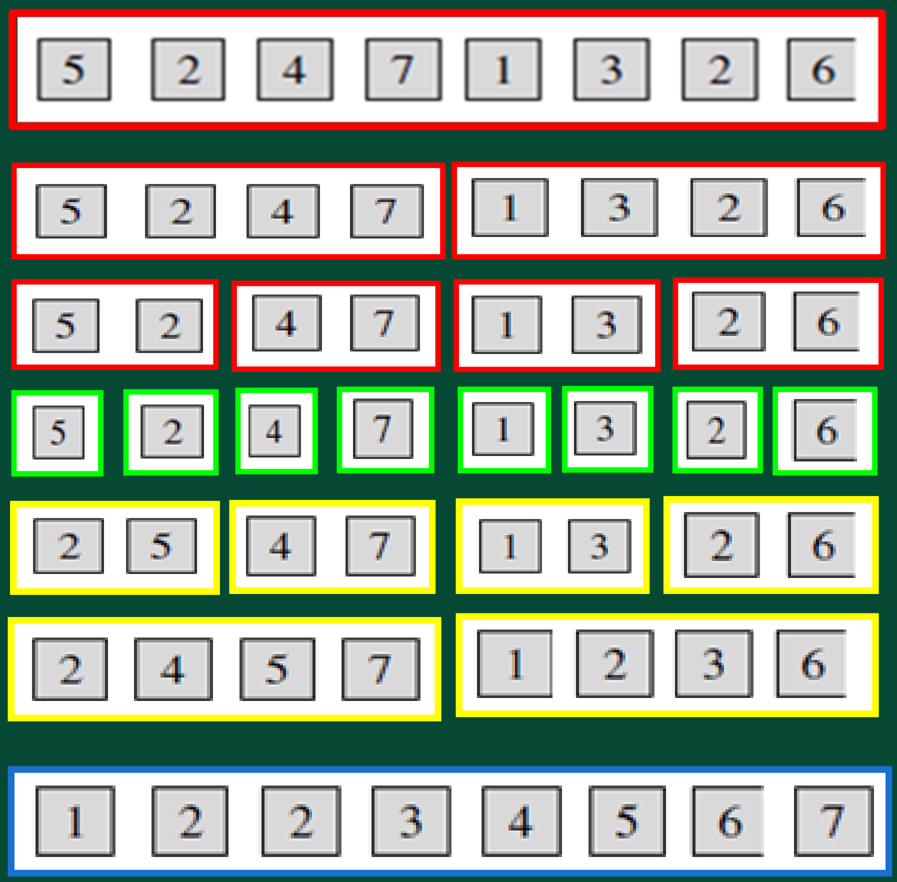
- Tri par Fusion :
  1. Si le tableau ne contient qu'un élément, alors il est déjà trié.
  2. Divisez le tableau en deux moitiés.
  3. Triez chaque moitié de manière **récursive**.
  4. Fusionnez les deux moitiés (**triées**) pour obtenir un ensemble trié.

- Utilise un espace auxiliaire (tableau ...)

- Proposé par John Von Neumann (1903-1957)
  - Pionnier de l'informatique.
  - Architecture de von Neumann utilisée dans la quasi-totalité des ordinateurs modernes,
  - Test pour voir comment sa machine serait à la hauteur sur d'autres tâches.

Tri par Fusion (Merge-Sort)

- Exemple : Trier le tableau 5, 2, 4, 7, 1, 3, 2, 6
- Séquence initiale:



- Tri par Fusion :
  1. Si le tableau ne contient qu'un élément, alors il est déjà trié.
  2. Divisez le tableau en deux moitiés.
  3. Triez chaque moitié de manière récursive.
  4. Fusionnez les deux moitiés (triées) pour obtenir un ensemble trié.

Tri par Fusion : Pseudo Code

```
Fonction Fusionner(a[] : Entier, lo : Entier, mid : Entier, hi : Entier)

Fonction Tri_Fusion(a[] : Entier, lo : Entier, hi : Entier)
Variable N, mid : Entier
Début
   // Tri du tableau a[lo, hi) : l'index hi est exclu
   N ← hi - lo
   Si (N <= 1)
       retourner 0 // tableau taille 1 considéré trié
   FinSi
   mid ← lo + N/2
   Tri_Fusion(a, lo, mid)
   Tri_Fusion(a, mid, hi)
   Fusionner(a, lo, mid, hi)
Fin
```

Tri par Fusion : Complexité

- $T(N) = T(N/2) + T(N/2) + 2N$ // **2N** va et vient de a vers aux
- Pour simplifier : $N = 2^n$
- Par construction $T(1) = 0$
- $T(2^n) = 2 \times T(2^{n-1}) + 2 \times 2^n$
- $T(2^n) / 2^n = T(2^{n-1}) / 2^{n-1} + 2 \times 1$
- $T(2^n) / 2^n = T(2^{n-2}) / 2^{n-2} + 2 \times 2$
- $T(2^n) / 2^n = T(2^{n-3}) / 2^{n-3} + 2 \times 3$
- $T(2^n) / 2^n = T(2^{n-n}) / 2^{n-n} + 2 \times n$
- $T(2^n) / 2^n = 2 \times n$ // puisque $T(1) = 0$
- $T(N) = 2 \times N \times n$ // or $n = \log_2(N)$
- $T(N) = 2 \times N \times \log_2(N) = O(N \log_2(N))$

- Tri par Fusion :
  1. Si le tableau ne contient qu'un élément, alors il est déjà trié.
  2. Divisez le tableau en deux moitiés.
  3. Triez chaque moitié de manière récursive.
  4. **Fusionnez** les deux moitiés (triées) pour obtenir un ensemble trié.

- Partie 2 : Tri par Tas
  - Arbres Binaires
  - Tas (Heaps)
  - Tris par Tas (Heap sort)

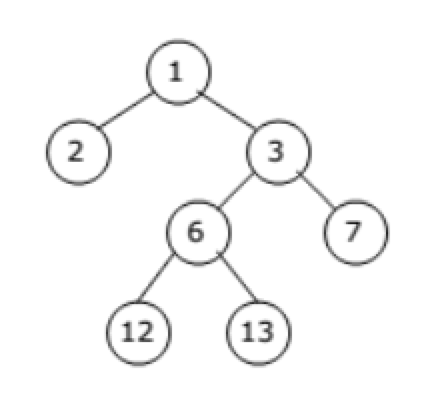
Arbres

- Un arbre est schématisé (modélisé) par des nœuds dont chacun possède éventuellement des descendants (fils).
- Chaque nœud possède au plus un père.
- Le nœud ne possédant pas de père s'appelle **racine de l'arbre**. (en haut de l'arbre)
- Un nœud ne possédant pas de fils est une **feuille**.
- **Arbre binaire**: arbre où chaque nœud possède au plus deux fils. On parle de fils gauche (FG) et fils droit (FD).
- Le fils gauche (resp. droit) est la racine du sous-arbre gauche (resp. droit).

Arbres

- La **profondeur** d'un nœud p dans un arbre est la longueur (nombre d'arêtes) du chemin à partir de la racine jusqu'au nœud p.

- Ex:
  - La profondeur du nœud valué par 7 est égale à 2
  - La profondeur du nœud valué par 6 est égale à 2
  - La profondeur du nœud valué par 13 est égale à 3
  - La profondeur du nœud valué par 2 est égale à 1
  - La profondeur du nœud valué par 1 est égale à 0

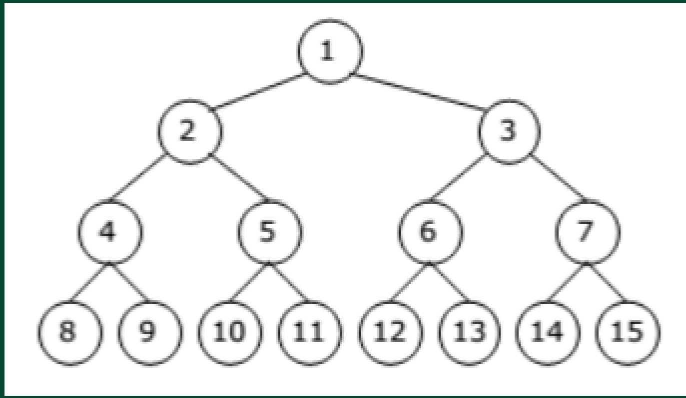


- La **hauteur** d'un arbre binaire est la profondeur maximale d'un nœud, ou -1 si l'arbre est vide.

- Ex: La hauteur de l'arbre ci-haut est égale à 3.

Arbres Binaires Complet (ABC)

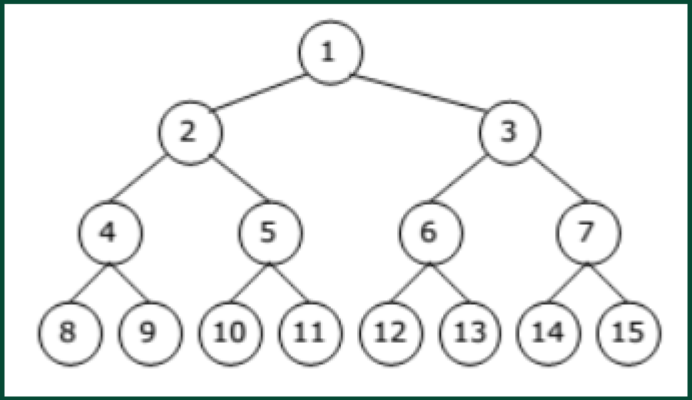
- Un **Arbre Binaire Complet** **(ABC)** est un arbre binaire (non vide) de hauteur h, possédant 2^h feuilles.
- Corollaires :
  - Pour un arbre binaire complet:
    - La profondeur de chaque **feuille** est égale à la **hauteur** h de l'arbre.
    - Le nombre de **nœuds** est $2^{h+1} - 1$,
- Exemple:



Arbres Binaires Complet (ABC) ____old

- Un **arbre binaire complet** est :
  - un arbre dont la profondeur de chaque **feuille** est égale à la **hauteur** h de l'arbre.
  - Le nombre de **feuilles** d'un arbre binaire complet est 2^h

- Exemple:



- Le nombre de **nœuds** d'un arbre binaire complet est $2^{h+1} - 1$,

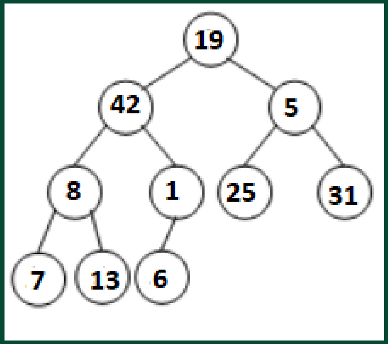
Arbres Binaires Presque-Complet (ABPC)

- Un **<u>arbre binaire presque-complet</u>** est un arbre de hauteur h tel que:
  1. Pour d : 0,…,h-1 il y a 2^d nœud(s) de profondeur d
  2. Les feuilles de profondeur h sont les feuilles les plus à gauche.

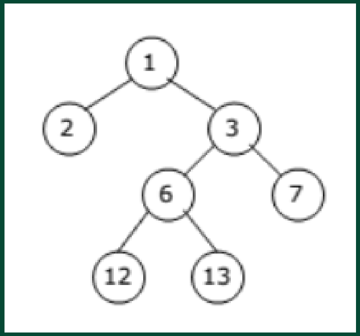
- La profondeur des feuilles d'un ABPC est égale à h ou h-1

- Nomenclature anglophone:
  - Nearly complete binary tree, or
  - Almost complete binary tree

\- Exemple d'AB Presque-complet:



\- Contre Exemple (AB non presque complet):



ABPC : Implémentations

- Par tableau A[]:
  - Utilisation des indices
  - Le fils gauche de A[i], s'il existe, est à la position 2*i + 1
  - Le fils droit de A[i], s'il existe, est à la position 2*i + 2
  - Le père de A[i], exceptée la racine, est à la position: (i-1)/2

ABPC : Parcours en Profondeur

- ParcoursPrefixe (Arbre binaire T de racine r)

  Visiter_racine [r]
  ParcoursPrefixe (Arbre de racine fils _gauche [r])
  ParcoursPrefixe (Arbre de racine fils_droit [r])

- ParcoursPostfixe (Arbre binaire T de racine r)

  ParcoursPostfixe (Arbre de racine fils _gauche [r])
  ParcoursPostfixe (Arbre de racine fils_droit [r])
  Visiter_racine [r]

- ParcoursInfixe (Arbre binaire T de racine r)

  ParcoursInfixe (Arbre de racine fils _gauche [r])
  Visiter_racine [r]
  ParcoursInfixe (Arbre de racine fils_droit [r])

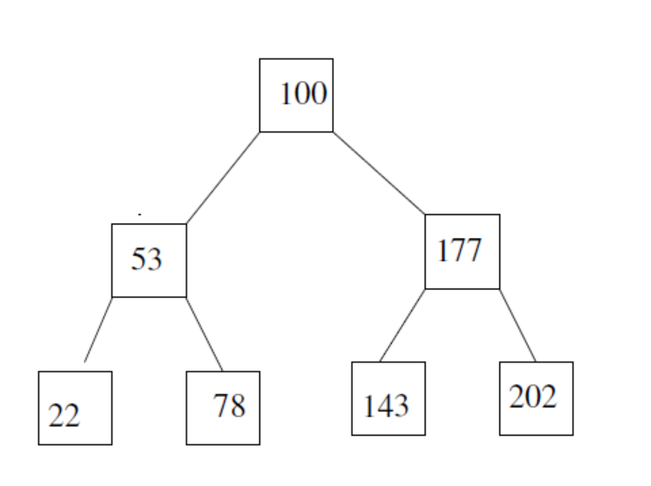
ABPC : Parcours

- En largeur – par niveau de hauteur (cf tas):
  - 100 – 53 – 177 – 22 – 78 – 143 - 202

- En profondeur:
  - Prefixe (RGD):
  - 100 – 53 – 22 – 78 – 177 – 143 - 202
  - (RGD ~ 1er passage)

  - Postfixe (GDR):
  - 22 – 78 – 53 – 143 – 202 – 177 - 100
  - (GDR ~ dernier passage)

  - Infixe (GRD):
  - 22 - 53 - 78 - 100 - 143 -177 - 202
  - (GRD ~ 2ème passage)



Arbre Binaire de Recherche (ABR)

- Un arbre binaire de recherche de racine A est un arbre binaire tel que:
  - chaque élément du sous-arbre gauche est inférieur ou égal à A,
  - chaque élément du sous-arbre droit est supérieur ou égal à A
  - selon la mise en œuvre de l'ABR, on pourra interdire ou non des éléments de valeur égale.
  - Les éléments que l'on ajoute deviennent des feuilles de l'arbre.

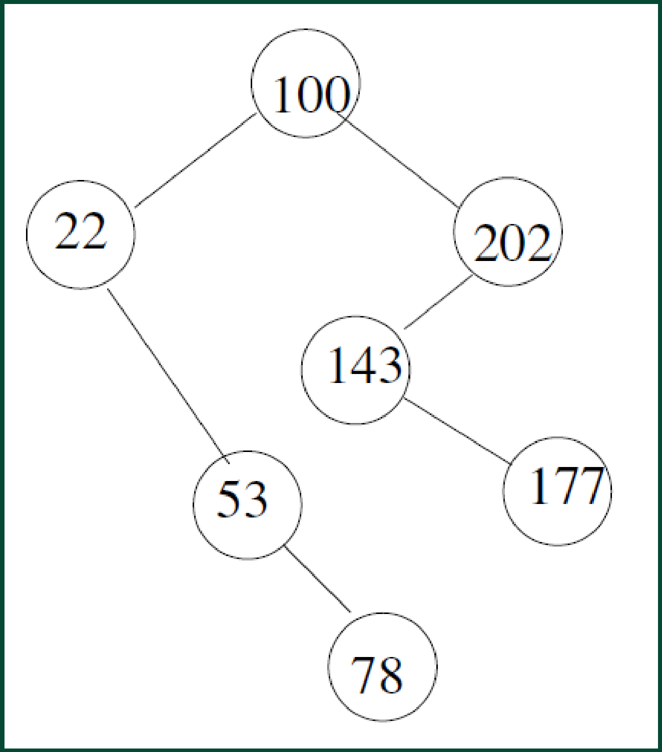
- Utilité: rapidité de
  - Recherche d'un élément (spécialement du Min ou du Max)
  - insérer un élément
  - supprimer un élément

ABR : Parcours

- En profondeur: (Attention aux nœuds qui n'ont pas de fils gauche)
  - prefixe (RGD):
  - 100 - 22 - 53 - 78 - 202 - 143 – 177
  - (RGD ~ 1er passage)

  - suffixe (GDR dernier pass):
  - 78 - 53 - 22 - 177 - 143- 202 – 100
  - (GDR ~ dernier passage)

  - Infixe (GRD):
  - 22 - 53 - 78 - 100 - 143 -177 - 202
  - (GRD ~ 2ème passage)
  - Remarque : suite triée



Tas

- Un tas binaire est schématisé par un arbre binaire presque-complet, où chaque nœud est **prioritaire** par rapport à ses fils,

- Implémenté par un tableau T tel que pour chaque élément T[i] :
  - Le fils gauche, s'il existe, est à la position 2*i + 1
  - Le fils droit, s'il existe, est à la position 2*i + 2
  - Le père, exceptée la racine, est à la position: (i-1)/2

- Si la priorité est représentée par une relation de **supériorité**, on parle de **tas max**, tel que:

$$T[\text{parent}(i)] \geq T[i]$$

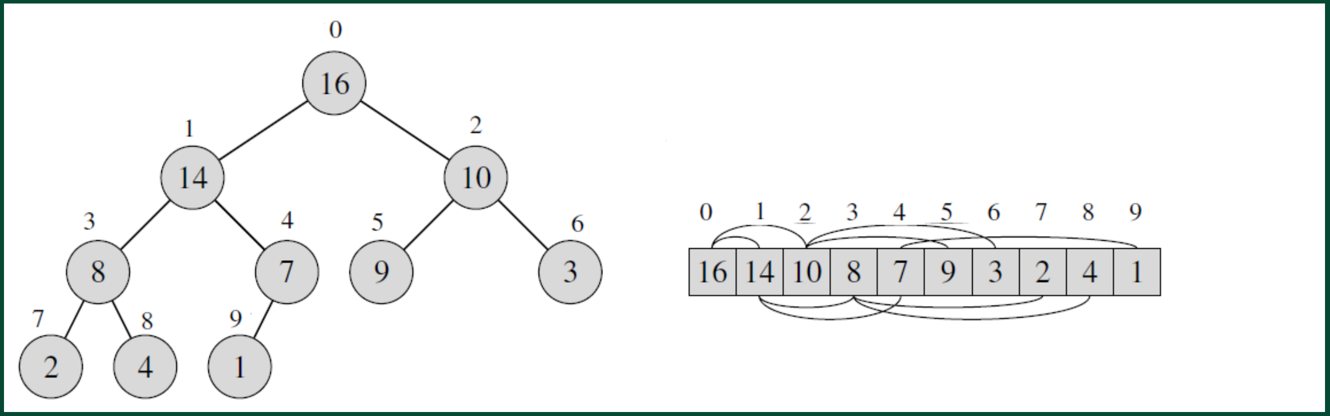
  Tq la fonction parent(i) retourne (i-1)/2

- Si la priorité est représentée par une relation d'**infériorité**, on parle de **tas min**, tel que:

$$T[\text{parent}(i)] \leq T[i]$$

Tas

- Exemple de tas max:



– Noter que pour chaque nœud, il n y a pas nécessairement une relation d'ordre entre le fils gauche et le fils droit:
  - FG(0) > FD(0)
  - FG(3) < FD(3)

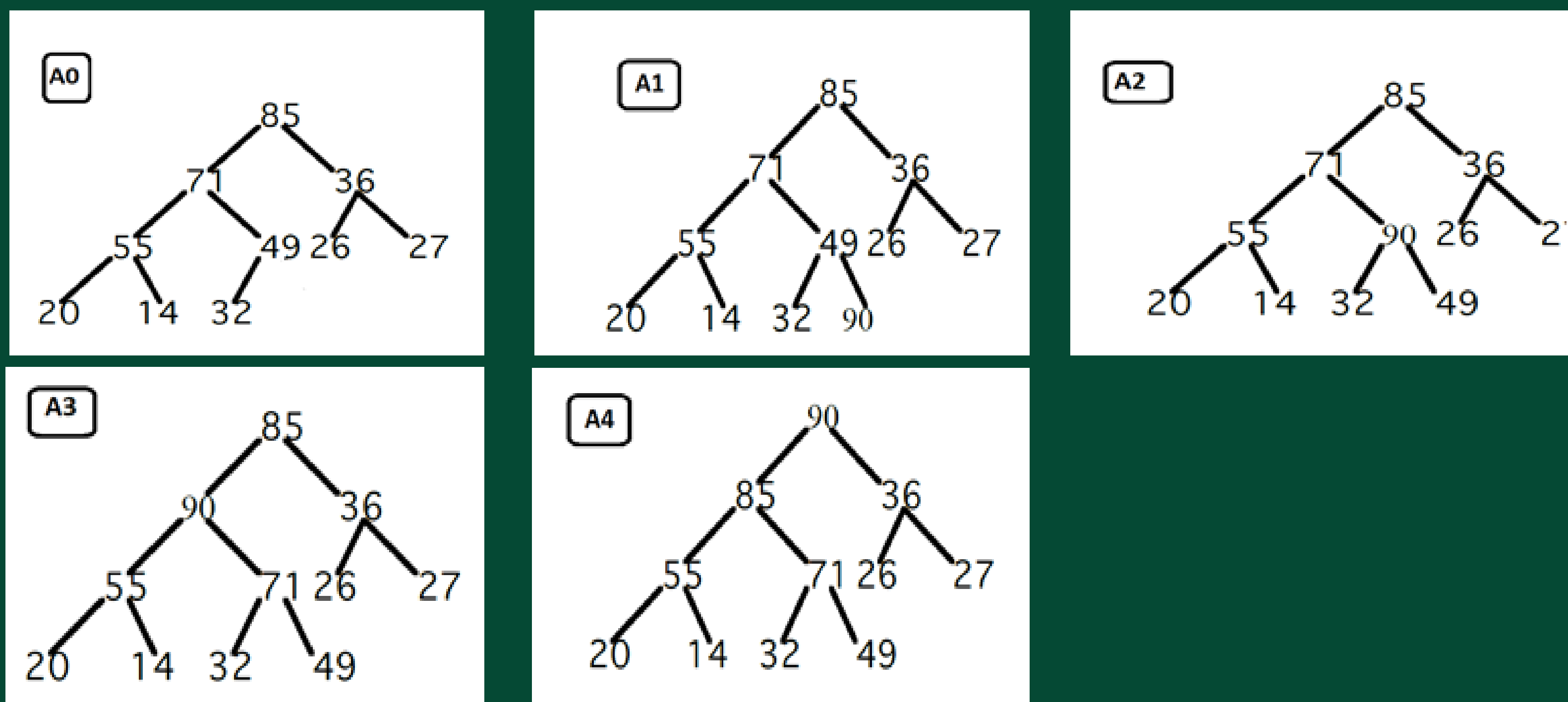
– Noter aussi qu'on peut remplir le tableau T à partir de l'arbre en faisant un parcours horizontal de chaque niveau de profondeur, en commençant par la racine (niveau 0, puis 1, …).

Tas

- Un tas (binaire) T a deux attributs :
  - longueur[T]: nombre d'éléments du tableau T,
  - taille[T]: nombre d'éléments du tas rangés dans le tableau T.
- cf application plus loin

Tas: Ajout d'un élément

- Ajout d'un élément <u>v</u> à un tas T de taille n (dernier élément du tas est à l'indice n-1)
  - Insérer le nouvel élément à l'indice n,
  - Tant qu'on n'a pas atteint la racine, faire pour le nœud <u>v</u> :
    - Si la propriété du tas max est vérifié avec le père, alors **<u>arrêt</u>**
    - Sinon Permuter le nœud en question avec son père.
- Exemple: Ajout de 90 au tas T: 85 71 36 55 49 26 27 20 14 32



Tas: Ajout d'un élément:: Pseudo-code

```
Fonction permuter(a: ^ Entier, b: ^Entier): vide
Variable tmp: Entier
Début
    tmp ← a^
    a^ ← b^
    b^ ← tmp
Fin

Fonction aj_Elm_Tas(T[]: Entier, taille: Entier, v: Entier): vide
Variables: indfils, indpere : Entier
Début
    T[taille] ← v
    indfils ← taille
    indpere ← (indfils -1)/2
    TantQue (T[indpere] < T[indfils] ET indpere ≥ 0) alors
        permuter(&T[indpere],&T[indfils])
        indfils ← indpere
        indpere ← (indfils -1)/2
    FinTantQue
Fin
```

Construction d'un Tas max

- Nous allons construire un tas max à partir d'un tableau T de taille n

- Idée:

  Pour chaque élément T[i], tq i ≥ 1
  - Supposer que le tableau T[0 … i-1] constitue déjà un tas max
  - Ajouter l'élément T[i] au tas T[0 … i-1] de taille i

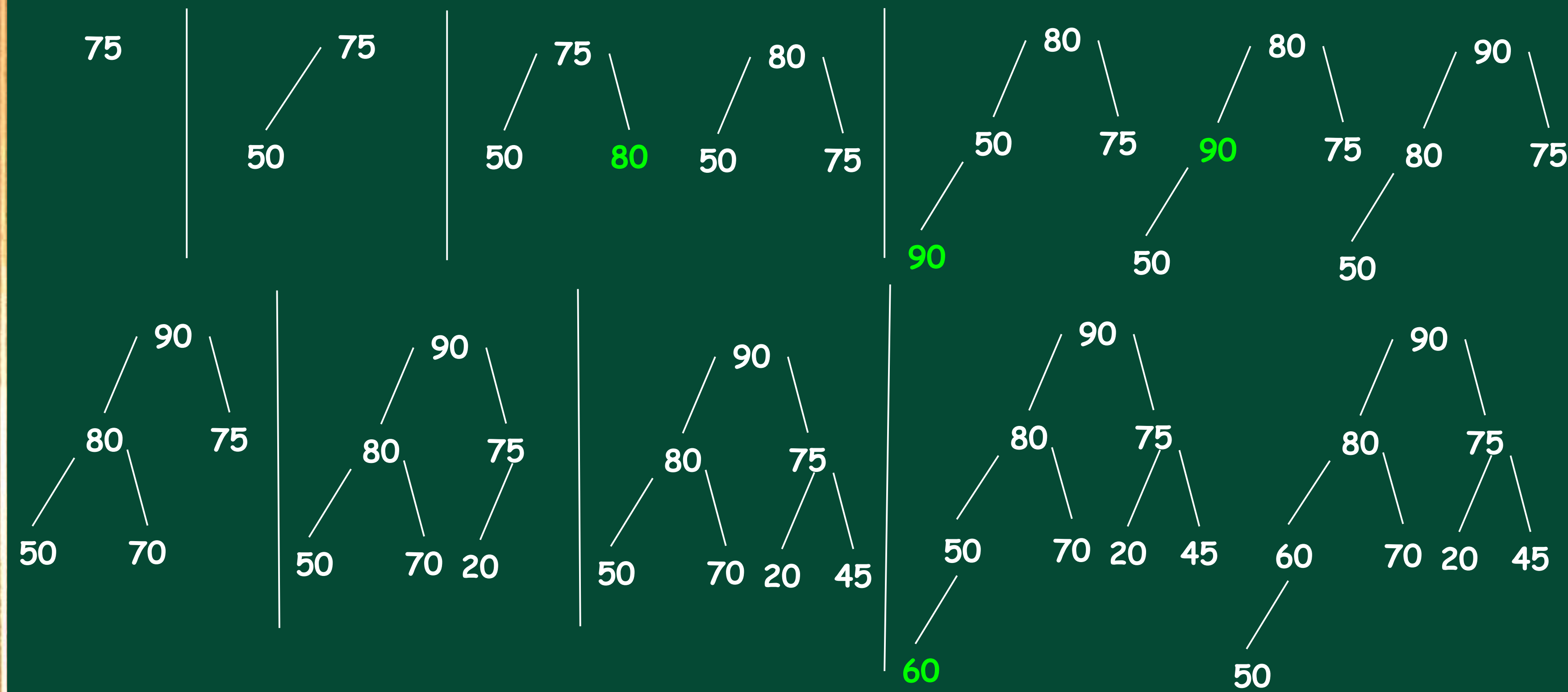
- Pseudo-code:

```
Fonction Const_Tas(T[] : Entier, n: Entier): vide
Variable i : Entier
Début
   Pour i allant de 0 à n-1 Faire
     aj_Elm_Tas(T, i, T[i])
   FinPour
Fin
```

- Ex: construire un tas max à partir du tableau T: { 75, 50, 80, 90, 70, 20, 45, 60, 30, 111 }

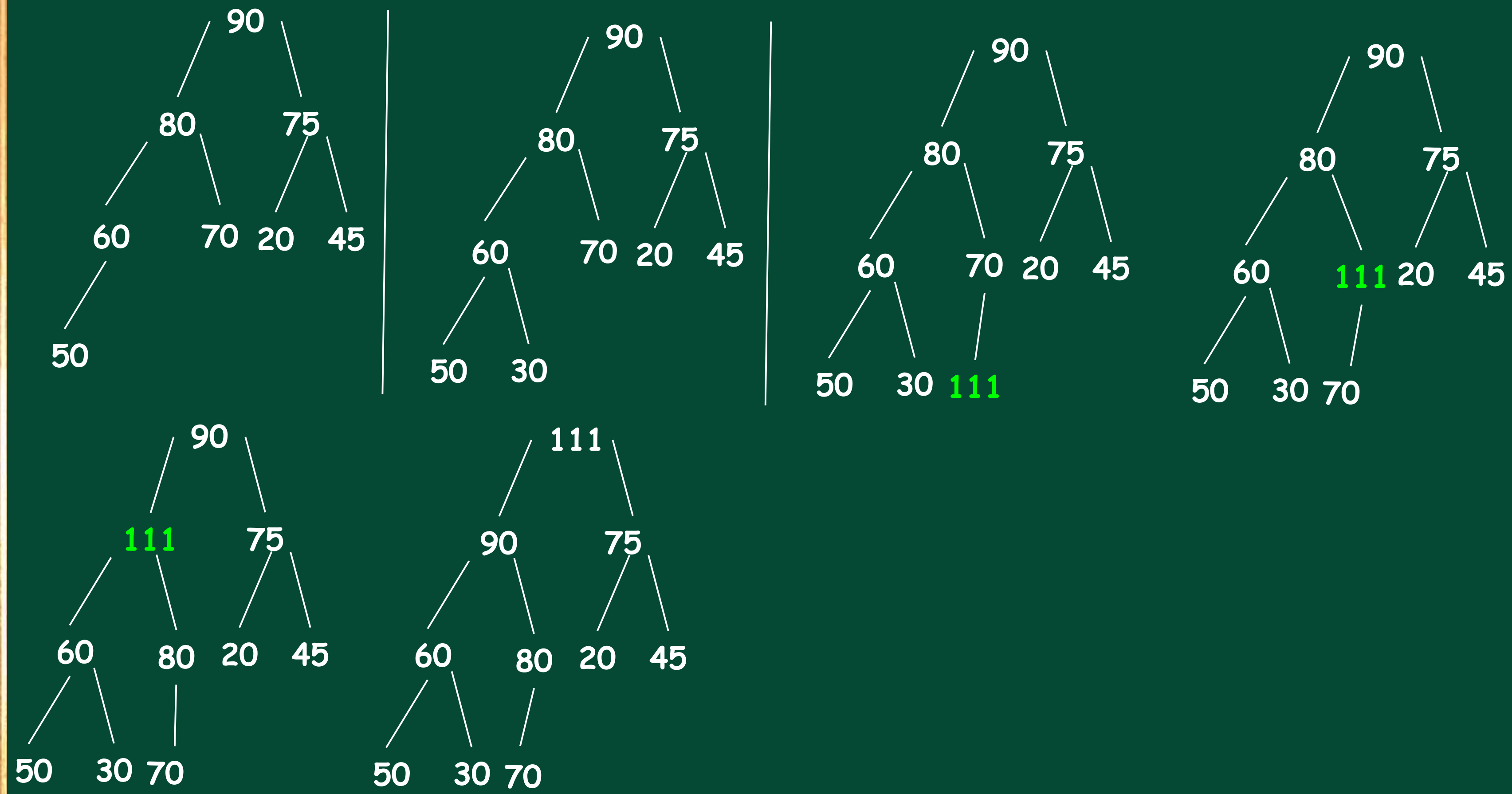
Construction d'un Tas max : Exemple

- Construire un tas max à partir du tableau T: { 75, 50, 80, 90, 70, 20, 45, 60, 30, 111 }



Construction d'un Tas max : Exemple

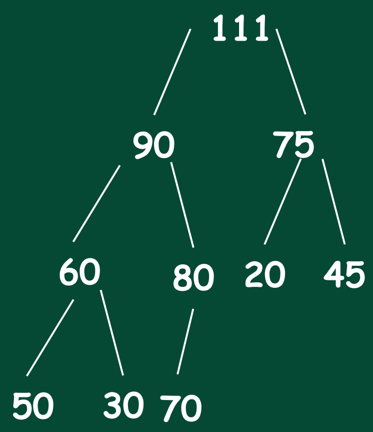
- Construire un tas max à partir du tableau T: { 75, 50, 80, 90, 70, 20, 45, 60, 30, 111 }



Construction d'un Tas max : Exemple

- Construire un tas max à partir du tableau T: { 75, 50, 80, 90, 70, 20, 45, 60, 30, 111 }

- Tas :



- Tableau

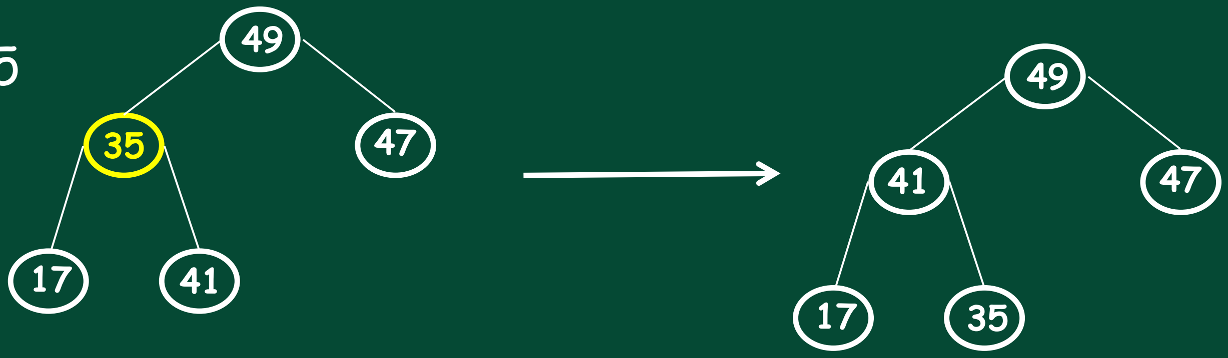
111	90	75	60	80	20	45	50	30	70

Tas max: Descente d'un élément

- Soit T un tas max de taille n avec une exception: "L'élément T[i] représente une **violation singulière** de la propriété du tas max "

- Solution: Faire descendre T[i] dans l'arborescence jusqu'à un nœud où la propriété du tas max soit respectée.
- Comment ?
  1. Tant qu'on n'a pas atteint une feuille:
  2. Echanger le nœud qui représente la violation avec le plus grand de ses fils, et vérifier:
  3. S'il n'y a plus de violation, alors **arrêt**
  4. Sinon, retour à 1

- Exemple: nœud 35



Tas max: Descente d'un élément :: Pseudo-code

```
Fonction Desc_Elm(T[]: Entier, i: Entier, taille: Entier): vide
Variables indFG, indFD, posTouve : Entier
indFG ← 2*i + 1 , indFD ← 2*i + 2 , posTrouve ← 0
TantQue (indFG < taille ET indFD < taille ET posTrouve = 0)
    Si (T[i] >= T[indFG] ET T[i] >= T[indFD])
        posTrouve ← 1
    Sinon si (T[indFG] < T[indFD])     // Permuter avec le plus grand des fils
          permuter(&T[i],&T[indFD])
          i ← 2*i + 2
          sinon
              permuter(&T[i],&T[indFG])
              i ← 2*i + 1
          FinSi
    FinSi
    indFG ← 2*i + 1
    indFD ← 2*i + 2
FinTantQue
Si (posTrouve = 0 ET indFG < taille ET T[i] < T[indFG] )
     permuter(&T[i],&T[indFG])
FinSi
Fin
```

Tri par Tas

- Soit T un tableau de longueur n, que l'on veut trier dans un ordre croissant, le tri par tas consiste à :
  - Transformer T en un tas max
  - Répéter n-1 fois (extraction de la racine):
    1. Décrémenter la taille du tas (et non la longueur du tableau)
    2. Permuter la racine T[0] avec T[taille] (la racine est le max du tas T)
    3. Descente de la racine (nouveau T[0])
- Pseudo-code

```
Fonction Tri_Tas (Tab[]: Entier, n: Entier): vide
Variable taille : Entier
Début
Const_Tas(Tab,n)
taille ← n
Pour i allant de 0 à n-2 Faire
   taille ← taille-1
   permuter(&Tab[0],&Tab[taille])
   Desc_Elm(Tab,0,taille)
FinPour
Fin
```

Tri par Tas : Application

- Tri par tas du tableau T: { 75, 50, 80, 90, 70, 20, 45, 60, 30, 111 }

  Transformer T en un tas max

T:	111	90	75	60	80	20	45	50	30	70
Iter 1:	90	80	75	60	70	20	45	50	30	**111**
Iter 2:	80	70	75	60	30	20	45	50	**90**	**111**
Iter 3:	75	70	50	60	30	20	45	**80**	**90**	**111**
Iter 4:	70	60	50	45	30	20	**75**	**80**	**90**	**111**
Iter 5:	60	45	50	20	30	**70**	**75**	**80**	**90**	**111**
Iter 6:	50	45	30	20	**60**	**70**	**75**	**80**	**90**	**111**
Iter 7:	45	20	30	**50**	**60**	**70**	**75**	**80**	**90**	**111**
Iter 8:	30	20	**45**	**50**	**60**	**70**	**75**	**80**	**90**	**111**
Iter 9:	**20**	**30**	**45**	**50**	**60**	**70**	**75**	**80**	**90**	**111**

Tri par Tas : complexité

- Trier T par tas :
  - Transformer T en un tas max :
    complexité : $O(n \log(n))$
    (peut atteindre $O(n)$)

  - Répéter n-1 fois (extraction de la racine):
    1. Décrémenter la taille du tas (et non la longueur du tableau)
    2. Permuter la racine T[0] avec T[taille] (la racine est le max du tas T)
    3. Descente de la racine (nouveau T[0]). Complexité : $O(\log(n))$

    Complexité (n fois $O(\log(n))$) = $O(n \log(n))$

- Complexité de l'algorithme du tri par tas : $O(n \log(n))$

references

- Introduction à l'algorithmique. Cours et exercices. Cormen et al. 2e édition.
- Computer Science, An Interdisciplinary Approach. Edgewick / Wayne. http://introcs.cs.princeton.edu
- http://examradar.com/binary-trees/
- https://en.wikipedia.org/wiki/Binary_tree
- http://tdinfo.phelma.grenoble-inp.fr/1Apet/td/td8b.pdf
- http://homepages.math.uic.edu/~leon/cs-mcs401-s08/handouts/nearly_complete.pdf