

Algorithmique Avancée & Programmation

*Un grand merci à
Christian Vercauter,
rédacteur des documents
dont ce poly s'inspire*

Séances 4 et 5: fil rouge

Test 4



- QCM moodle
 - On attend toutes les réponses justes, pas seulement la plus précise
- 20 minutes

On attend toutes les réponses justes, pas seulement la plus précise

Théorème maître : sélection de la règle :

On compare la croissance de $n^c = n^{\log_b(a)}$ à celle de $f(n)$

- Règle 1 : n^c croît plus vite que $f(n)$
 - Si $f(n) = O(n^{\log_b(a-\varepsilon)})$ avec $\varepsilon > 0$ $\equiv f(n) = O(n^d)$ avec $d < c$
 - Alors, $T(n) = \Theta(n^c)$
- Règle 2 : n^c et $f(n)$ ont des croissances équivalentes
 - Si $f(n) = \Theta(n^c \cdot \log^k(n))$ avec $k \geq 0$ (généralement, $k=0$)
 - Alors, $T(n) = \Theta(n^c \cdot \log^{k+1}(n))$
- Règle 3 : n^c croît moins vite que $f(n)$, et $f(n)$ satisfait une *condition de régularité*
 - Si $f(n) = \Omega(n^{\log_b(a+\varepsilon)})$ avec $\varepsilon > 0$ $\equiv f(n) = \Omega(n^d)$ avec $d > c$
 - Et $\exists k < 1$ tq. $a f(n/b) \leq k f(n)$ pour n suffisamment grand
 - Alors, $T(n) = \Theta(f(n))$

Cadrage séance 4 :

- Test papier : questions sur les tris, application du théorème général
 - Surtout théorique
- Présentation du fil rouge de l'année
- Structures de données, principaux algorithmes associés
- Constitution des groupes
 - 4 étudiants par groupe
- Mise en oeuvre la démarche :
 - Conception
 - Développement
 - Evaluation

TEA

(indicatif)

- Préparation du travail sous forme d'un document de conception (à définir) :
 - Architecture du code
 - Prototypes des fonctions
 - ...

Après cette séance, vous devez :

- Avoir compris le cahier des charges du fil rouge
- Avoir compris les algorithmes sous-jacents au sujet considéré
- Avoir sélectionné les solutions techniques que vous emploierez, et les avoir organisées dans une architecture bien documentée
- Vous être réparti les tâches de développement à réaliser

Cadrage séance 5 :

- 2h
- Pas de test en séance
 - Sauf rattrapages éventuels d'épreuves pour lesquelles l'étudiant était excusé
- Retours et conseils individuels sur leur travail de conception
- Séance de développement avec possibilité de demander des conseils à l'intervenant présent

TEA

(indicatif)

- Poursuite du travail sur le fil rouge, qui devra être rendu (code + rapport) et sera évalué
 - Utilisation de tests de similarité de code source pour identifier les plagiats éventuels
- Mini-challenge entre les groupes pour déterminer un classement
 - Le classement comptera pour partie dans la note du fil rouge

Fil Rouge 2022

Arbres partiellement ordonnés

- Arbres : définitions
- Arbres partiellement ordonnés : APO
- Tri pas tas
- Minimier, Minimier indirect
- Codage de Huffman
- Présentation du fil rouge
- Code Couleur

Fil Rouge 2020

Graphes

- Graphes : définition, représentation
- Algorithmes de recherche de plus courts chemins

Fil Rouge 2021 :

Arbres

- Arbres binaires : définitions, propriétés
- Arbres binaires de recherche : ABR
 - Implémentation
 - Parcours d'arbres et affichage graphique
- Arbres équilibrés : AVL
 - Rééquilibrages
 - Implémentation
 - Complexité

Fil Rouge 2023

Jeux combinatoires abstraits

- Minimax
- Élagage alpha-beta
- Tables de hachage



Arbres binaires : Définitions

Arbre informatique

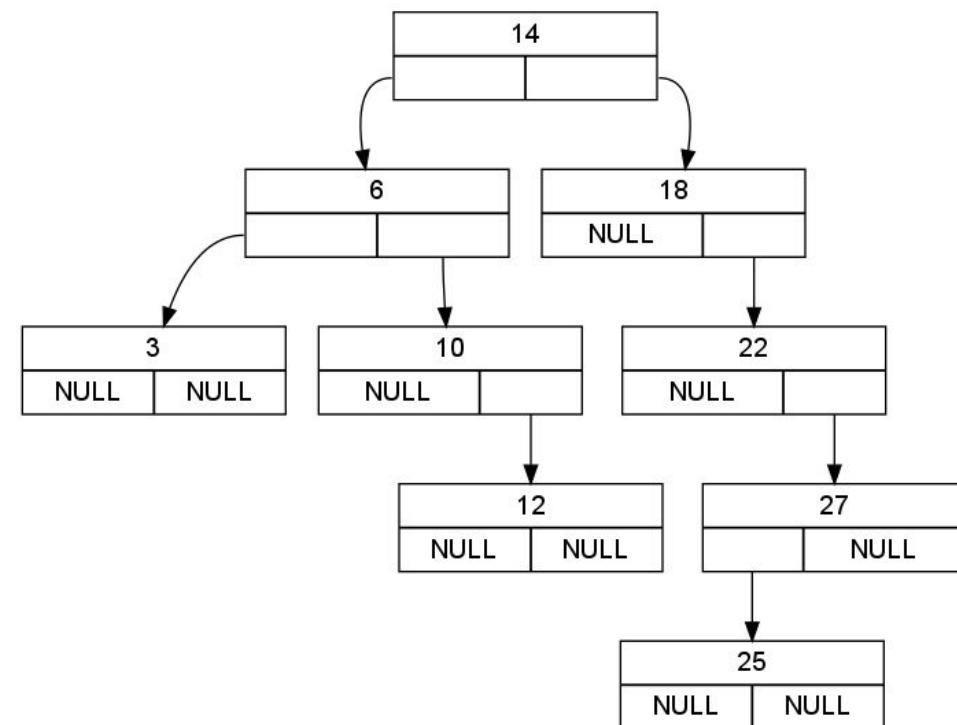
- En informatique, un **arbre** est une **structure de données** permettant d'organiser des données de manière hiérarchique :
 - Système de fichiers et de répertoires d'un système d'exploitation
 - Structure d'un document : volume, chapitre, sous-chapitre, paragraphes
 - Syntaxe d'un langage de programmation...
- Analogie avec les arbres généalogiques et arbres végétaux :
 - **Nœud père**, nœud **fil**s, nœud **frère**
 - **Racine**, **feuilles**, **branches**, ...

Arbre : définitions

- **Nœud** : cellule portant l'information et des liens vers d'autres cellules
- **Racine** : premier nœud de l'arbre, n'a pas de père
- **Feuille** : nœud n'ayant pas d'enfants
- **Nœud interne** : nœud ayant au moins un enfant
- **Branche** : suite de nœuds, du nœud **racine** jusqu'à une feuille

Arbre Binaire

- Un **arbre binaire** peut être :
 - Un arbre vide
 - Un arbre constitué d'une **racine** reliée à, **au plus, deux sous-arbres binaires**
- Définition récursive !



Types d'arbres binaires

- Complet

- Tous les niveaux sont remplis

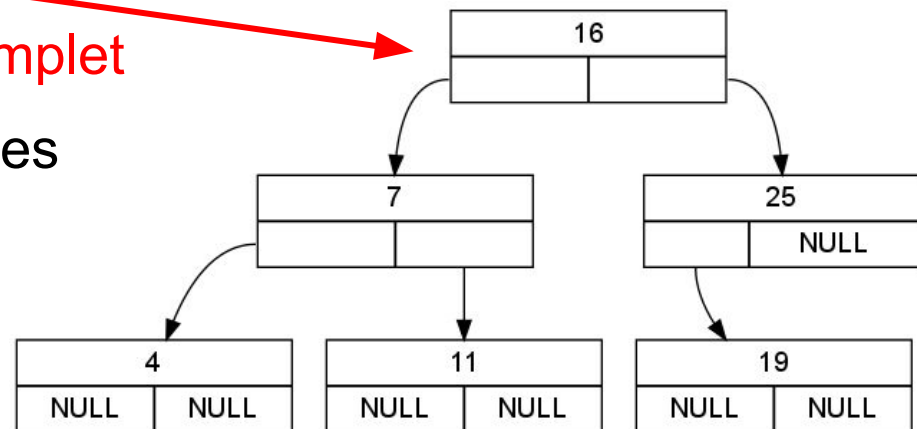
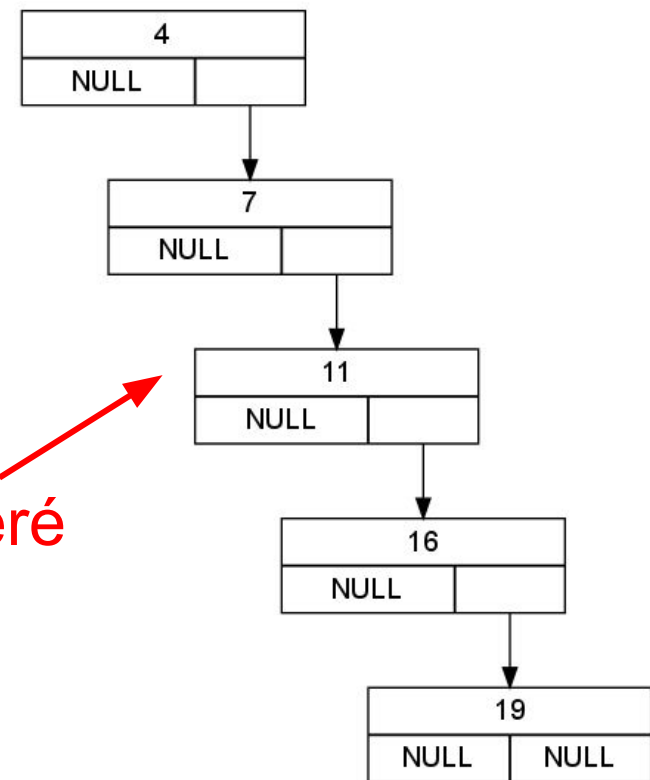
- Strict ou localement complet

- Tous les nœuds possèdent zéro ou deux fils

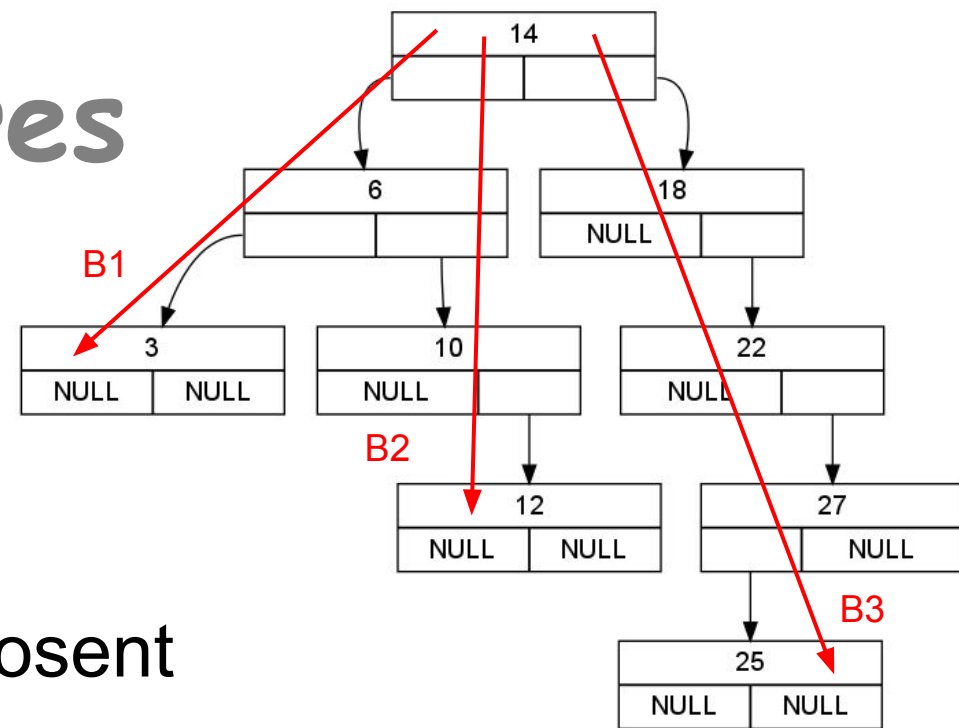
- Quasi-complet

- Seul le dernier niveau est incomplet
- Il peut manquer quelques feuilles

- Dégénéré



Propriétés des arbres (binaires ou non)



- **Longueur** d'une branche :
 - Nombre d'arcs qui la composent
- **Hauteur** d'un arbre :
 - Longueur de la plus longue branche
 - -1 s'il est vide, 0 si un seul nœud
- **Profondeur** d'un nœud :
 - Nombre d'arcs sur la branche qui mène à ce nœud
 - La racine est un nœud de profondeur 0

Exercice

Propriétés des arbres binaires

Longueur d'une branche : Nombre d'arcs qui la composent

Hauteur d'un arbre : Longueur de la plus longue branche (-1 s'il est vide)

Profondeur d'un nœud : Nombre d'arcs sur la branche qui mène à ce nœud

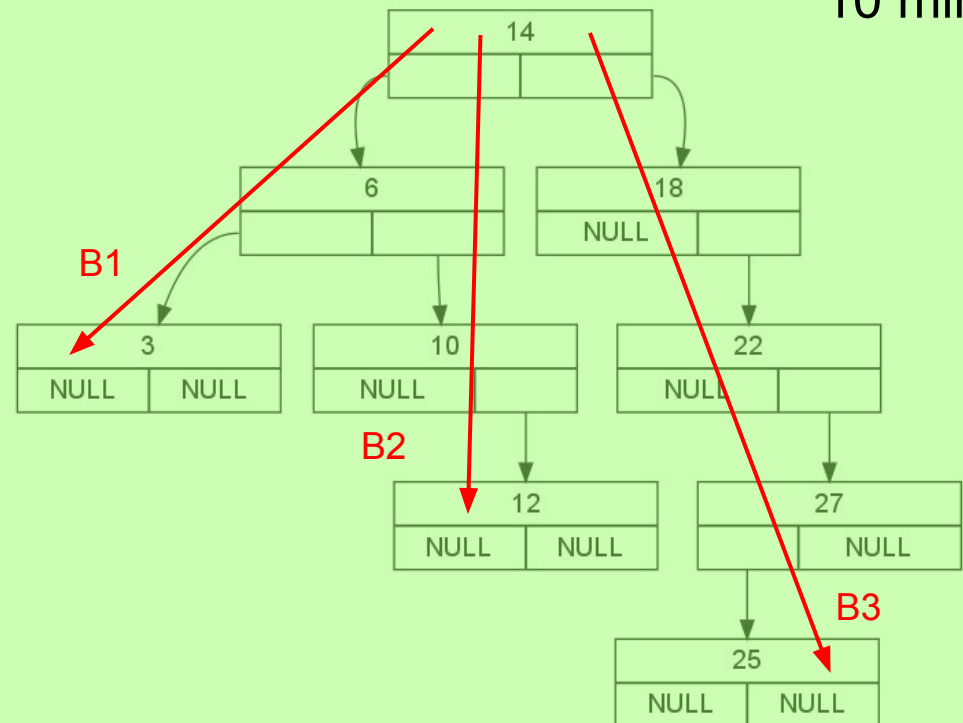
La racine est un nœud de profondeur 0

3h20



10 min

- Longueur de chaque branche ?
- Hauteur de l'arbre ?
- Profondeur de chaque nœud ?

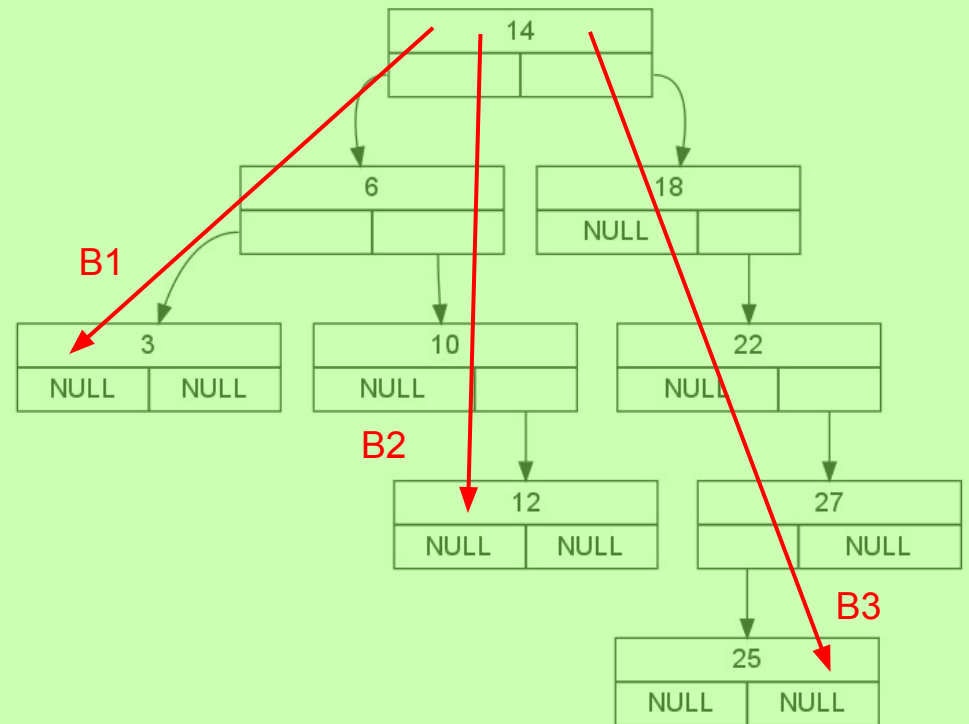


Longueur d'une branche : Nombre d'arcs qui la composent
Hauteur d'un arbre : Longueur de la plus longue branche (-1 s'il est vide)
Profondeur d'un nœud : Nombre d'arcs sur la branche qui mène à ce nœud
La racine est un nœud de profondeur 0

Exercice

Propriétés des arbres binaires

- $L(B1) = 2$
- $L(B2) = 3$
- $L(B3) = 4$
⇒ Hauteur de l'arbre = 4
- $P(14) = 0$
- $P(6) = P(8) = 1$
- $P(3) = P(10) = P(22) = 2$
- $P(12) = P(27) = 3$
- $P(25) = 4$



Exercice corrigé

Propriétés des arbres binaires

3h10



20 min

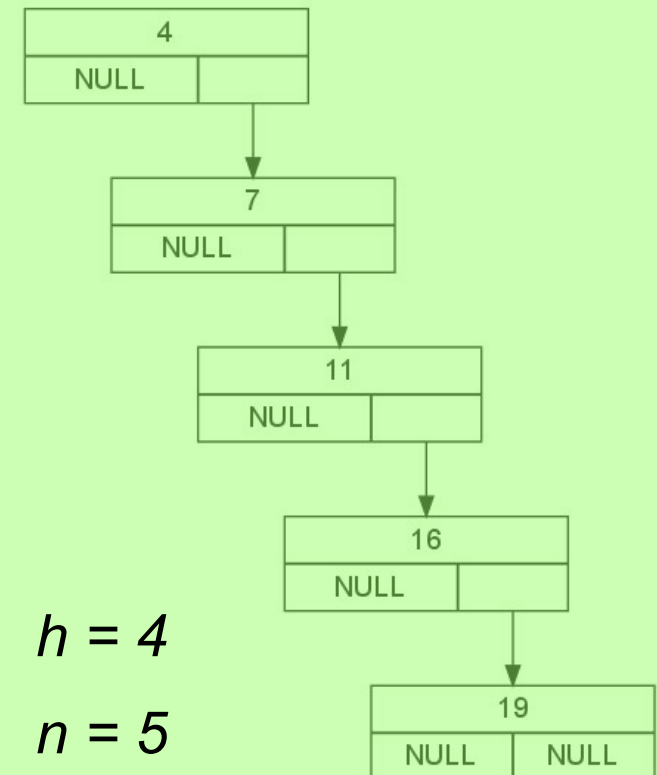
- Nombre **minimum** de noeuds d'un arbre binaire de hauteur h ?
- Nombre de nœuds d'un arbre binaire **complet** de hauteur h ?
- Nombre **maximum** de feuilles d'un arbre binaire de hauteur h ?
- Nombre de nœuds d'un arbre binaire **quasi-complet** de hauteur h ?
- Hauteur minimale d'un arbre binaire de n nœuds ?
- Hauteur maximale d'un arbre binaire de n nœuds ?

Exercice

Propriétés des arbres binaires

- Nombre minimum de nœuds d'un arbre binaire de hauteur h ?
 - Cas d'un arbre dégénéré

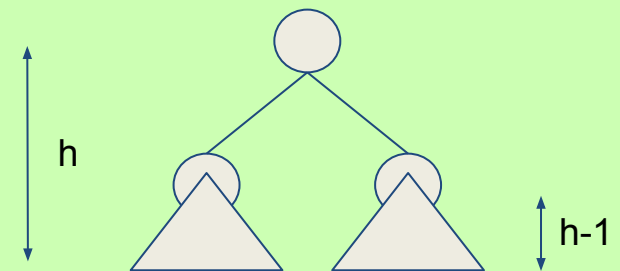
- $n_{\min} = h+1$



Exercice

Propriétés des arbres binaires

- Nombre de nœuds d'un arbre binaire **complet** de hauteur h ?
 - C'est le nombre **maximum** de nœuds que l'on peut implanter dans un arbre de hauteur h
 - $n_{\max}(h) = 2^{h+1} - 1$
- Par récurrence :
 - $n_{\max}(0) = 1$
 - $n_{\max}(1) = 3 \quad \dots$
 - $n_{\max}(h) = 1 + 2 * n_{\max}(h-1)$



Exercice

Propriétés des arbres binaires

- Nombre maximum de feuilles d'un arbre binaire de hauteur h ?
 - Construction d'un arbre complet
 - $f_{\max}(h) = 2^h$
- A chaque niveau supplémentaire, on ajoute 2 fois plus de feuilles qu'au niveau précédent
- $f_{\max}(0) = 1$
- $f_{\max}(1) = 2$
- $f_{\max}(h) = 2 * f_{\max}(h-1)$

Exercice

Propriétés des arbres binaires

- Nombre de nœuds d'un arbre binaire **quasi-complet** de hauteur h ?
- Seul le **dernier niveau** est incomplet
- Le sous-arbre racine sans ses feuilles est complet
 - Il est de hauteur $h-1$
- $n_{qc}(h) = n_{max}(h-1) + \text{nombre de noeuds du dernier niveau (feuilles)}$
- $n_{qc}(h) = 2^h - 1 + \text{nombre de feuilles d'un arbre de hauteur } h$
- Nombre maximum de feuilles d'un arbre de hauteur h : 2^h
- $2^h - 1 + 1 \leq n_{qc}(h) \leq 2^h - 1 + 2^h$
- $2^h \leq n_{qc}(h) \leq 2^{h+1} - 1$

Exercice

Propriétés des arbres binaires

- Hauteur minimale d'un arbre binaire de n nœuds ?
 - On remplit tous les niveaux au maximum pour réduire la hauteur de l'arbre
 \Rightarrow Arbre **quasi-complet** : $2^h \leq n_{qc}(h) \leq 2^{h+1} - 1$
- $2^{h_{min}} \leq n \leq 2^{h_{min}+1} - 1$
- $2^{h_{min}} \leq n$ et $n+1 \leq 2^{h_{min}+1}$
- $h_{min} \leq \log_2(n)$ et $\log_2(n+1) \leq h_{min} + 1$
- $\log_2(n+1) - 1 \leq h_{min} \leq \log_2(n)$ *NB : h_{min} est un entier !*
- $h_{min} = \lceil \log_2(n+1) \rceil - 1 = \lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2(n) \rfloor$

Cf. feuille de calcul :

[Hauteur minimale d'un arbre binaire de \$n\$ nœuds](#)

$\lceil n \rceil$: plus petit entier supérieur ou égal à n

$\lfloor n \rfloor$: plus grand entier inférieur ou égal à n

$$h_{\min} = \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2(n) \rfloor$$

- $\log_2(n+1) - 1 \leq h_{\min} \leq \log_2(n)$
- $\log_2(n) - \log_2(n+1) + 1 = 1 - \log_2(1+1/n) < 1$
 - Donc il n'y a qu'un seul entier au plus dans l'intervalle $[\log_2(n+1)-1, \log_2(n)]$
- Par les propriétés des parties entières et puisque h_{\min} est un entier :
- $\log_2(n+1) - 1 \leq \lceil \log_2(n+1) - 1 \rceil \leq h_{\min} \leq \lfloor \log_2(n) \rfloor \leq \log_2(n)$
 - Il y a trois entiers dans l'intervalle considéré, ils sont donc égaux...

Exercice

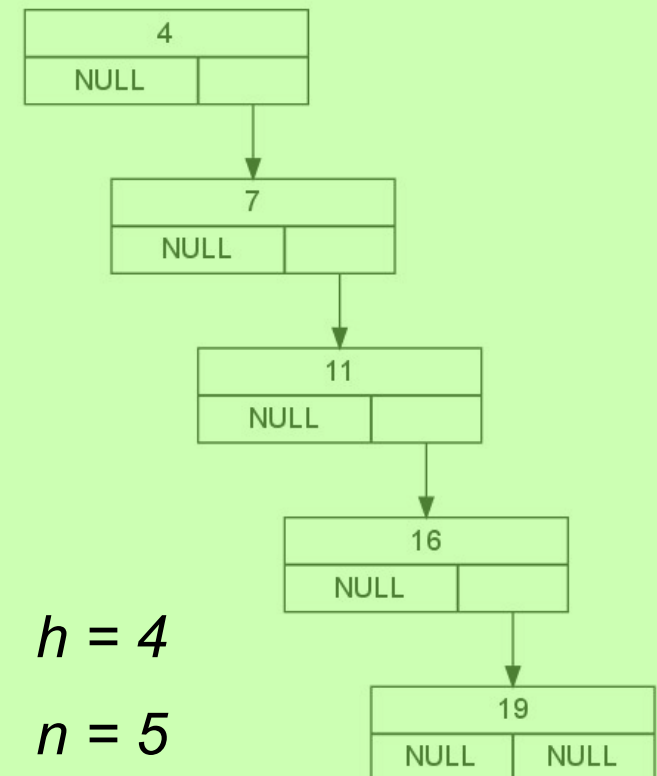
Propriétés des arbres binaires

- Hauteur maximale d'un arbre binaire de n nœuds ?
- Cas d'un arbre dégénéré

- $h_{\max}(n) = n-1$

- De façon générale :

$$\lfloor \log_2(n) \rfloor \leq h < n$$



Complexité dans les arbres binaires

- La plupart des opérations réalisées sur un arbre binaire ont un **coût proportionnel à la hauteur** de l'arbre binaire
- Coût en **$\Omega(\log(n))$** et **$O(n)$**
 - La complexité est en **$\Theta(\log(n))$** si l'arbre est équilibré
 - La complexité se dégrade en **$\Theta(n)$** si l'arbre est déséquilibré
- Rendre optimales les opérations sur un arbre binaire de recherche, c'est **réduire sa hauteur**
 - On applique des transformations pour rééquilibrer l'arbre binaire après insertion ou suppression : **Arbres AVL** (Cf. fil rouge 2021)

Files de priorité
Tas
APO
Minimiers/Maximiers

Définitions
Opérations sur un tas
Tri par tas : heapsort

File de priorité

"Priority Queues"

- Ensemble d'éléments à qui on attribue un **rang de priorité** avant qu'ils n'entrent dans la file et qui en **sortiront** précisément selon leur rang :
 - L'élément de rang **le plus élevé en premier**
- C'est un **type abstrait de données** opérant sur un ensemble ordonné et muni des opérations fondamentales suivantes :
 - Insérer un élément dans la file (selon son rang)
 - Accéder à l'élément de plus haut rang
 - Supprimer l'élément de plus haut rang
 - Augmenter le rang d'un élément

Cas file
de type "MAX"

Cas file
de type "MAX"

File de priorité (2)

Applications

- Planification de tâches dans un système d'exploitation
- Recherche du plus court chemin dans un graphe
 - Algorithme de Dijkstra
 - Algorithme A*
- Compression de documents
 - Codage de Huffman
- https://en.wikipedia.org/wiki/Priority_queue#Applications

File de priorité (3)

Implémentations

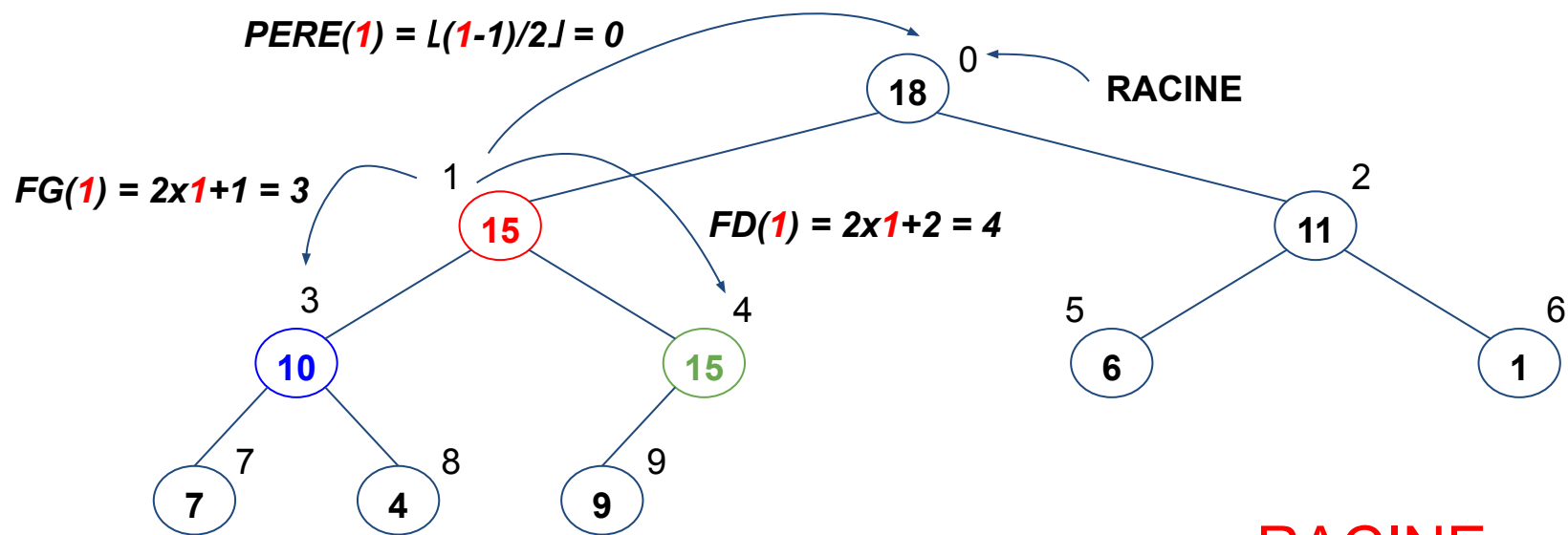
- À l'aide d'un **tableau ordonné** :
 - Recherche du max : $O(1)$
 - C'est le 1er élément du tableau
 - Insertion d'un élément, suppression du max : $O(n)$
 - Recherche de la position d'insertion : $O(\log n)$ comparaisons
 - Décalage d'au plus n éléments en cas d'insertion/suppression du max. : $O(n)$
- À l'aide d'une **structure de tas** :
 - Recherche du max : $O(1)$
 - Insertion d'un élément, suppression du max. : $O(\log n)$

Structure de tas

- Un **tas** (binaire) est un **arbre binaire quasi-complet** dans lequel tout nœud satisfait la **condition maximière** :
 - La valeur portée par **chaque nœud** est **supérieure ou égale à celles de ses descendants**
 - L'élément de plus grande valeur est donc dans la racine du tas
- Appellations équivalentes :
 - Maximier, minimier (permet d'indiquer le critère d'organisation utilisé)
 - Arbre partiellement ordonné (APO)
 - Binary Heap [https://en.wikipedia.org/wiki/Heap_\(data_structure\)](https://en.wikipedia.org/wiki/Heap_(data_structure))
- NB : ce tas n'a **rien à voir** avec le tas utilisé pour l'allocation dynamique de mémoire
- On le représente en mémoire par un **tableau**
 - L'élément de plus grande valeur est le 1^{er} élément du tableau

Représentation d'un tas

- Représentation sous forme d'**arbre binaire quasi-complet**



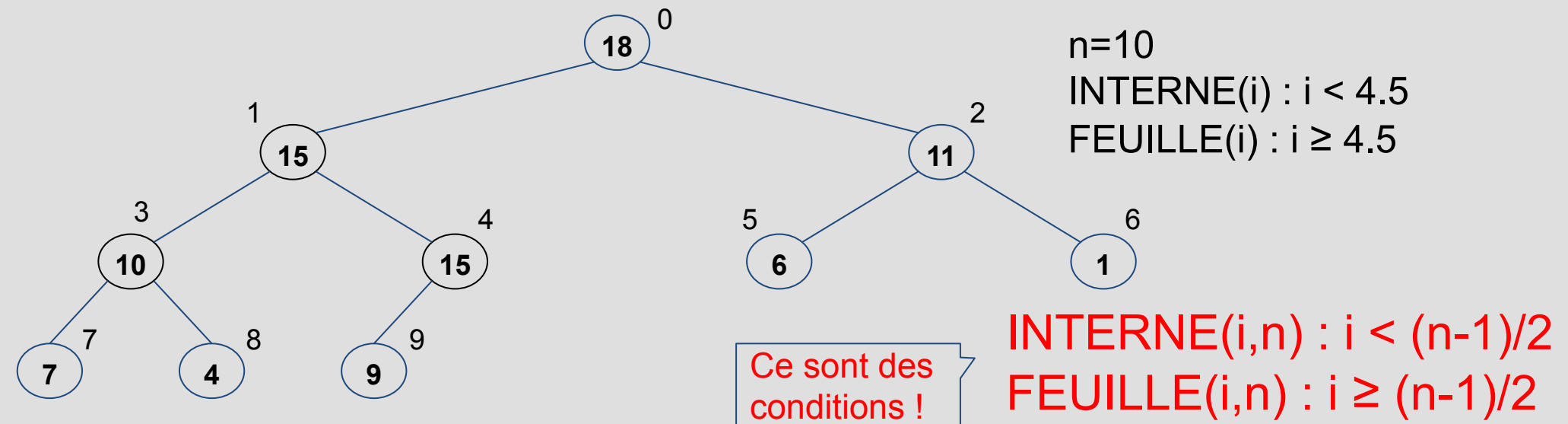
- Représentation sous forme de **tableau**

18	15	11	10	15	6	1	7	4	9
0	1	2	3	4	5	6	7	8	9

RACINE : nœud 0
 $PERE(i) : L(i-1)/2$
 $FG(i) : 2i + 1$
 $FD(i) : 2i + 2$

Représentation par un tableau : Où sont les feuilles ?

- Dans un tas de taille n , à quelle condition le nœud d'indice i est-il une feuille ? \Rightarrow S'il n'a pas de fils gauche !
- $\text{FEUILLE}(i) \Leftrightarrow \text{FG}(i) \geq n \Leftrightarrow 2i + 1 \geq n \Leftrightarrow i \geq (n-1)/2$
- $\text{INTERNE}(i) \Leftrightarrow \text{FG}(i) < n \Leftrightarrow 2i + 1 < n \Leftrightarrow i < (n-1)/2$



Représentation par un tableau : Combien y-a-t-il de nœuds internes ?

- Dans un tas de taille n , combien y-a-t-il de nœuds internes ?

- Indice du dernier nœud interne i_d tel que $i_d < (n-1)/2$

- Nombre de nœuds internes $nb_i = i_d + 1$

- Si n est impair, $(n-1)/2$ est entier, $i_d = (n-1)/2 - 1$

- $\Rightarrow nb_i = (n-1)/2 = n/2 - 1/2 = \lfloor n/2 \rfloor$

- Si n est pair, $i_d = \lfloor (n-1)/2 \rfloor = (n-2)/2 = n/2 - 1$

- $\Rightarrow nb_i = n/2 = \lfloor n/2 \rfloor$

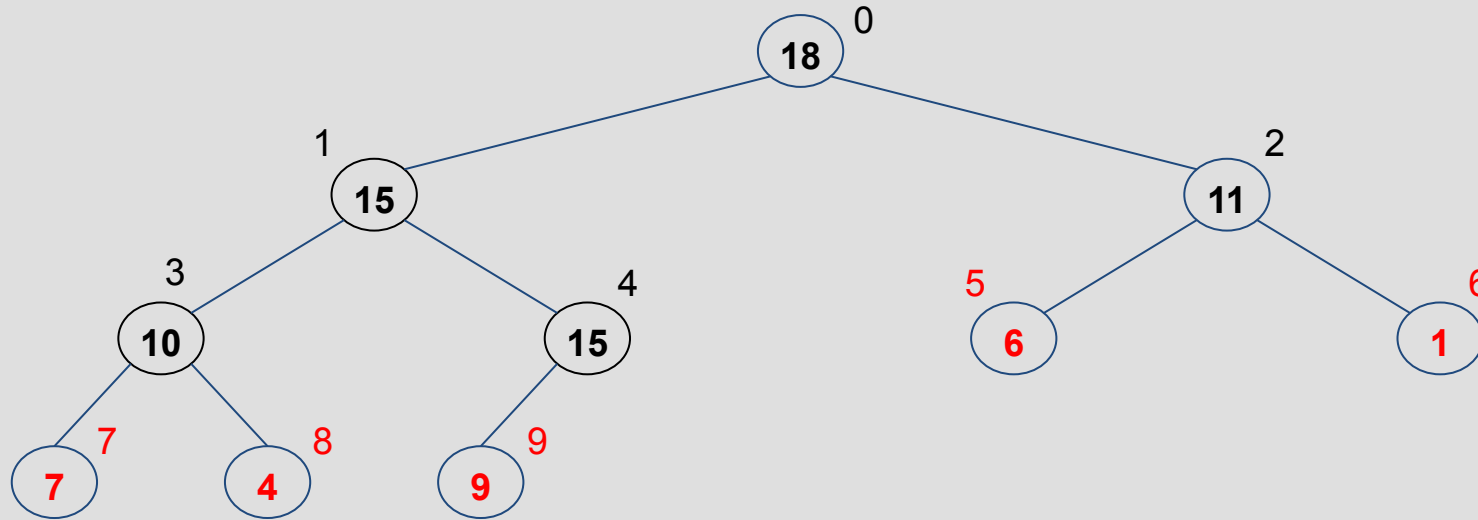
- $n = \lfloor n/2 \rfloor + \lceil n/2 \rceil$

- Nombre de feuilles : $\lceil n/2 \rceil$

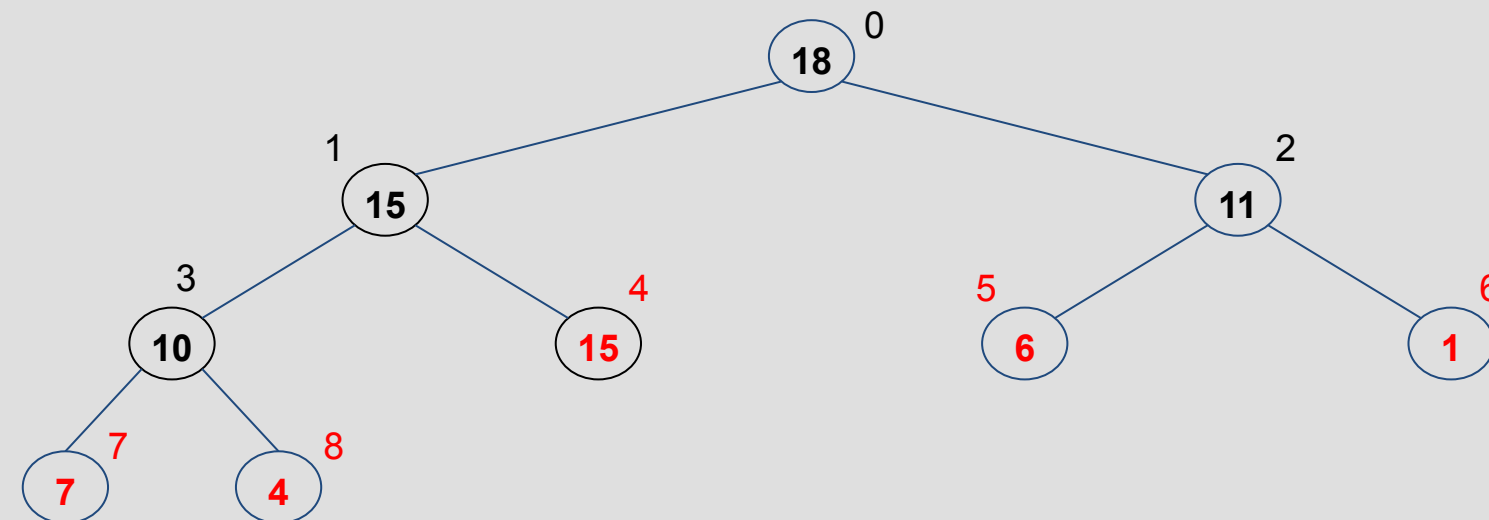
$\lfloor n/2 \rfloor$ nœuds internes
 $\lceil n/2 \rceil$ feuilles

INTERNE(i, n) : $i < (n-1)/2$ $\lfloor n/2 \rfloor$ nœuds internes
FEUILLE(i, n) : $i \geq (n-1)/2$ $\lceil n/2 \rceil$ feuilles

Vérifions...



$n=10$
INTERNE(i) : $i < 4.5$
 $\lfloor n/2 \rfloor = 5$ internes
FEUILLE(i) : $i \geq 4.5$
 $\lceil n/2 \rceil = 5$ feuilles



$n=9$
INTERNE(i) : $i < 4$
 $\lfloor n/2 \rfloor = 4$ internes
FEUILLE(i) : $i \geq 4$
 $\lceil n/2 \rceil = 5$ feuilles



Exercice 1

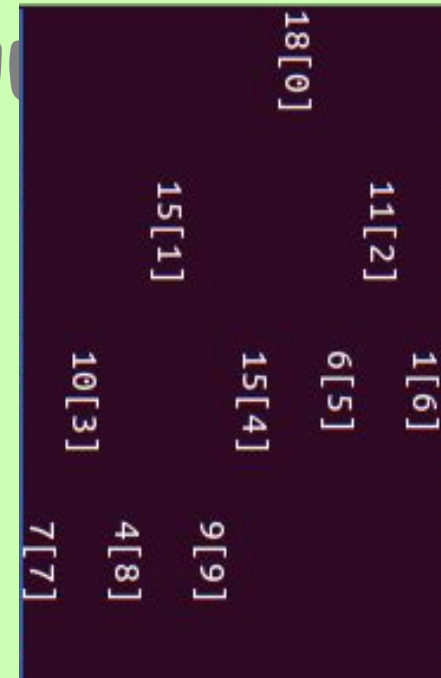
Affichage pseudo-graphique

```
#define isINTREE(i,n)      (i<n)
#define iPARENT(i)        (i-1)/2
#define iLCHILD(i)        (2*i)+1
#define iRCHILD(i)        (2*i)+2

void showPOT(int t[], int n);
void showPOT_rec(int t[], int n, int root, int indent);
```

- *APO* \Leftrightarrow *Arbre Partiellement Ordonné*
- *POT* \Leftrightarrow *Partially Ordered Tree*

```
int ex1[] = { 18, 15, 11, 10, 15, 6, 1, 7, 4, 9};
showPOT(ex1, 10);
```

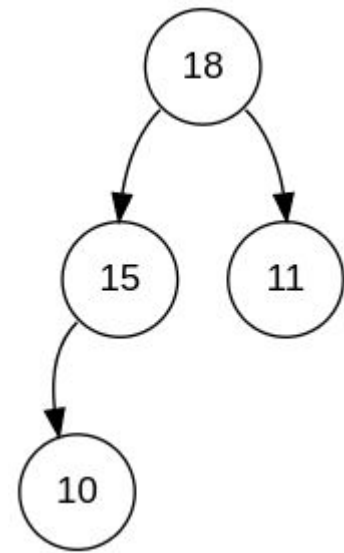


```
$ ./ex1/ex1.exe
Affichage d'un APO (nbElt : 10)
      1[6]
    11[2]  6[5]
18[0]      15[4]  9[9]
      15[1]      4[8]
              10[3]  7[7]
```

Affichage d'un APO

Format **dot**

NB : les noeuds doivent contenir des valeurs toutes distinctes : on utilise leurs indices



```
digraph POT_test {  
    node [fontname="Arial", shape="circle", width=0.5];  
    0 [label = "18"];  
    0:sw -> 1;  
    1 [label = "15"];  
    0:se -> 2;  
    2 [label = "11"];  
    1:sw -> 3;  
    3 [label = "10"];  
}
```

[apo.dot](#)

Exercice corrigé 1

Affichage d'un APO au format dot

2h30



10 min

- Tester sur votre machine Linux ou en ligne :
 - `dot -T png apo.dot -o apo.png`
 - <https://dreampuf.github.io/GraphvizOnline>
- `void genDotPOT_rec(int t[], int n, int root, FILE *fp)`
 - Écrit la structure dot correspondant à l'APO passé en paramètre dans le flux fp
- `void createDotPOT(int t [], int n, const char *basename)`
 - Produit le fichier dot correspondant à l'APO passé en paramètre
 - Appelle `genDot`

```

static void genDotPOT_rec(int t[], int n, int root, FILE *fp){
    // Attention : les fonction toString utilisent un buffer alloué comme une variable statique
    // => elles renvoient toujours la même adresse
    // => on ne peut pas faire deux appels à toString dans le même printf()

    // t : tas
    // n : taille du tas
    // root : indice de la racine du sous-arbre à produire
    // fp : flux correspondant à un fichier ouvert en écriture où écrire le sous-arbre

    /*
    0 [label = "18"];
    0:sw -> 1;
    1 [label = "15"];
    0:se -> 2;
    2 [label = "11"];
    1:sw -> 3;
    3 [label = "10"];

    */

    if (! isINTREE(root,n)) {
        return;
    }

    // cas général

    // sous-arbre droit
    genDotPOT_rec(t, n, iRCHILD(root), fp);

    // racine implantée à l'indice root dans le tas
    fprintf(fp,"%d [label=\"%d\\n\", root, t[root]);

    // arc menant à gauche ?
    if (isINTREE(iLCHILD(root),n)) {
        //0:sw -> 1;
        fprintf(fp,"%d:sw -> %d\\n",root,iLCHILD(root) );
    }

    // arc menant à droite ?
    if (isINTREE(iRCHILD(root),n)) {
        //0:se -> 2;
        fprintf(fp,"%d:se -> %d\\n",root, iRCHILD(root));
    }

    // sous-arbre gauche
    genDotPOT_rec(t, n, iLCHILD(root), fp);
}

```

Panorama des opérations sur un tas (de type maximier)

- Accès à l'élément maximum
- Insertion d'un nouvel élément
 - cf. remonter
- Transformer en maximier V1
 - cf. remonter
- Suppression de l'élément maximum
 - cf. descendre
- Transformer en maximier V2
 - cf. descendre

Opérations sur un tas de taille n

(1)

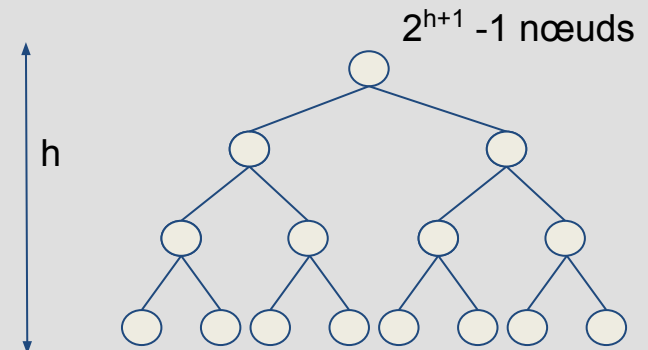
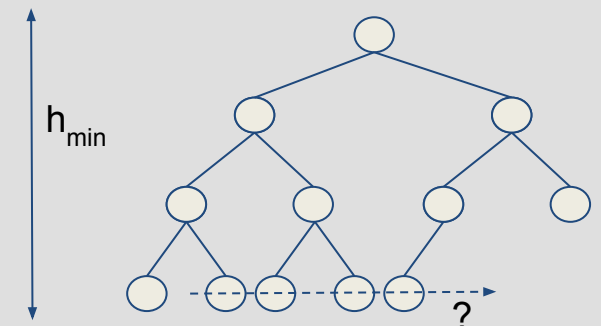
Accès, Insertion

- Accès au maximum : en $O(1)$: c'est l'élément racine
- Insertion d'un élément :
 - Création d'un nouveau nœud : $O(1)$
 - Accrochage au niveau le plus bas, le plus à gauche possible pour conserver la propriété quasi-complète : $O(1)$, le nombre d'éléments du tas étant connu
 - **Remonter** cet élément en procédant par échange avec le nœud père pour rétablir si nécessaire la condition maximière (quand l'élément inséré est grand) : $O(\log n)$
 - Nombre de comparaisons et d'échanges inférieur ou égal au nombre de nœuds présents sur la plus longue branche : $\lfloor \log_2(n) \rfloor + 1$

Nombre de nœuds présents sur la plus courte/longue branche d'un arbre binaire quasi-complet de n nœuds ?

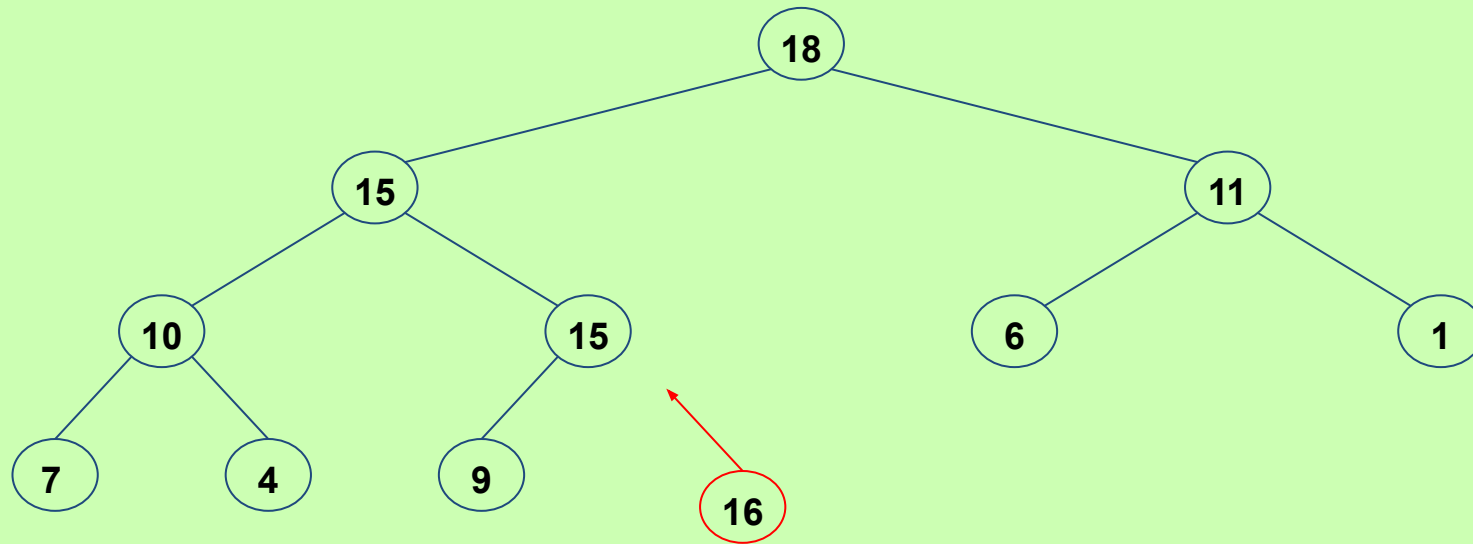
- hauteur = nombre d'**arcs** sur la plus longue branche = nombre de **nœuds** - 1
- Hauteur d'un arbre binaire quasi-complet de n nœuds ?
 - C'est la hauteur minimale d'un arbre binaire !
 - $h_{\min} = \lceil \log_2(n+1) \rceil - 1 = \lceil \log_2(n+1) \rceil - 1 = \lfloor \log_2(n) \rfloor$
- Nombre de nœuds sur la plus longue branche :
 - $n_{b+} = h_{\min} + 1 = \lceil \log_2(n+1) \rceil = \lfloor \log_2(n) \rfloor + 1$
- Nombre de nœuds sur la branche la plus courte :
 - $n_{b-} = \lfloor \log_2(n+1) \rfloor$
 - $\log_2(n+1)$ est entier seulement si l'arbre est complet

$$2^h \leq n \leq 2^{h+1} - 1$$



Exemple 1

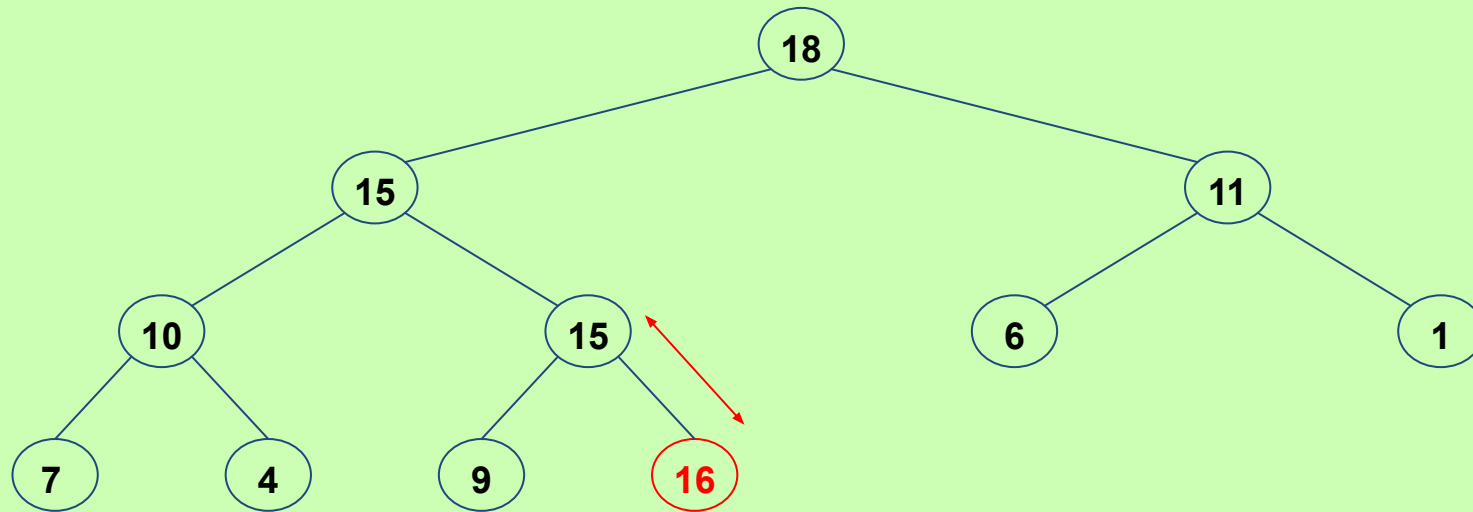
Insertion de 16 (1)



- Création d'un nouveau nœud contenant la valeur 16
- Accrochage du nouveau nœud au niveau le plus bas, le plus à gauche possible

Exemple 1

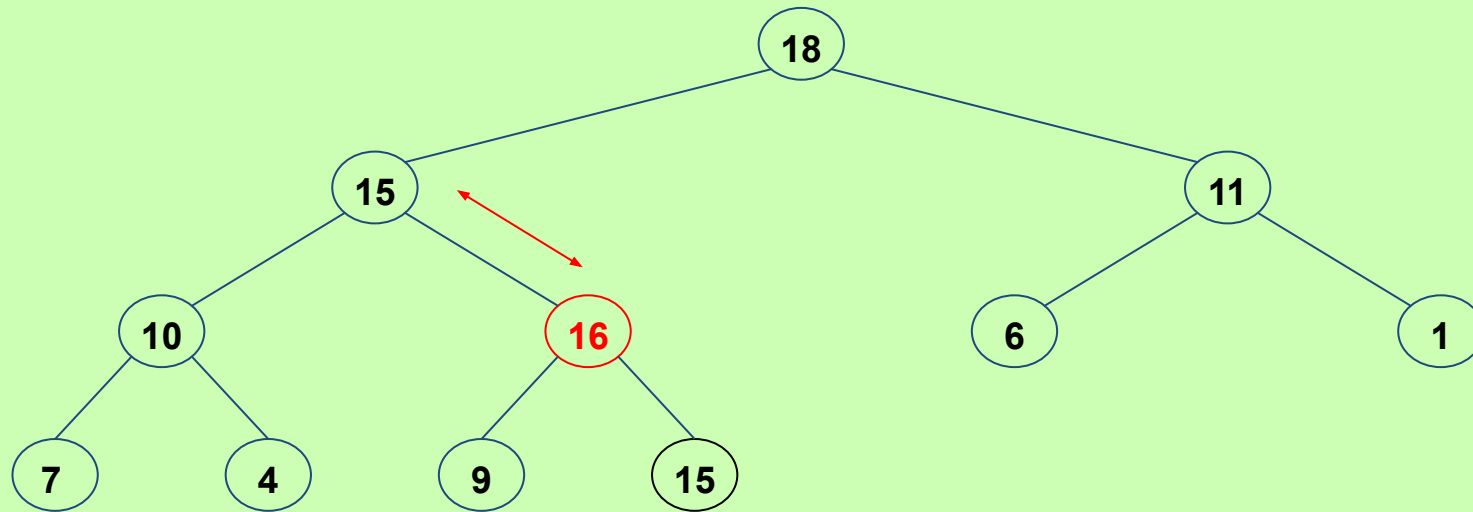
Insertion de 16 (2)



- Comparaison de 16 et de la valeur du nœud père (15)
- La condition maximière n'est pas valide, l'échange de ces deux nœuds est nécessaire

Exemple 1

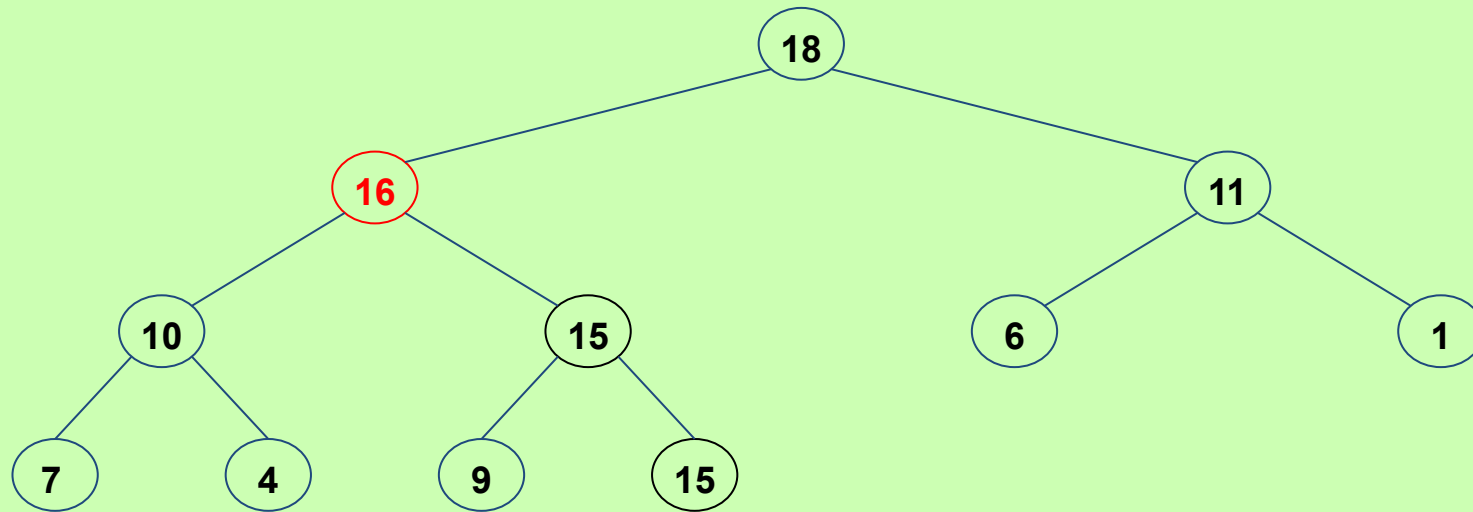
Insertion de 16 (3)



- Comparaison de 16 et de la valeur du nœud père (15)
- La condition maximière n'est pas valide, l'échange de ces deux nœuds est nécessaire

Exemple 1

Insertion de 16 (4)



- Comparaison de 16 et de la valeur du nœud père (18)
- La condition maximière est vérifiée, la procédure d'insertion est terminée

Procédure remonter(M, k)

- Remonter l'élément d'indice k, dans un maximier M dont les nœuds en sont les k – 1 premiers éléments, de façon à constituer un maximier de k nœuds :

remonter(M, k) :

```
Tant que      le nœud k de M a un père
ET            VAL(nœud k) supérieur à VAL(nœud Père(k))
|  Echanger nœud k et nœud Père(k)
|  k ← Père(k)
Fin Tant que
```

VAL(nœud k) \neq M[k]

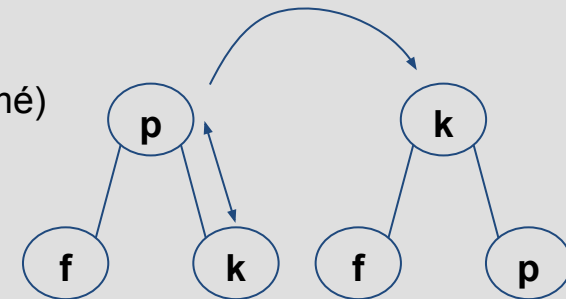
- Cas des *minimiers indirects*

- Coût $R(k)$ = nombre de comparaisons
 - \leq nombre de nœuds sur la plus **courte branche** d'un maximier de k-1 nœuds
- Nombre de noeuds sur la plus courte branche d'un maximier de k-1 nœuds : $\lfloor \log_2(k) \rfloor$
- $R(k) = \lfloor \log_2(k) \rfloor = O(\log k) \Rightarrow$ Coût de l'insertion en $O(\log n)$

Preuve de correction

```
remonter(M, k) :  
    Tant que      le nœud k de M a un père  
    ET            VAL(nœud k) supérieur à VAL(nœud Père(k))  
        Echanger nœud k et nœud Père(k)  
        k ← Père(k)  
    Fin Tant que
```

- Hypothèse d'appel : on remonte une valeur dans un maximier M, dont seul l'élément d'indice k ne vérifie pas la condition maximière
- Résultat à démontrer : après l'exécution de la fonction remonter, tous les nœuds vérifient la condition maximière
- Preuve :
 - Les seuls nœuds modifiés sont les nœuds présents sur la branche menant de l'élément k à la racine
 - Il faut montrer que pour tout nœud n sur le chemin d'insertion de k jusqu'à la racine 0, la condition maximière est respectée : $VAL(\text{nœud } n) \geq VAL(\text{nœud } FD(n))$ et $VAL(\text{nœud } n) \geq VAL(\text{nœud } FG(n))$
 - Il suffit de montrer que cette propriété est vraie après chaque échange (*Cf. invariant de boucle*)
 - Soit p l'indice du nœud père de k, f l'indice du nœud frère de k avant l'échange
 - Puisque M était un maximier, on avait $VAL(p) \geq VAL(f)$
 - L'échange a lieu si $VAL(k) > VAL(p)$ (sinon le maximier était bien déjà bien formé)
 - Après échange : $VAL(k) > VAL(FD(k))$ car $VAL(k) > VAL(p)$
 - $VAL(k) > VAL(f)$ car $VAL(k) > VAL(p)$ et $VAL(p) \geq VAL(f)$



Opérations sur un tas de taille n

(2)

Création d'un maximier V1

Cf. exemple dans la section tri par tas

- Réorganiser les n premiers éléments d'un tableau M pour former un maximier de n éléments :

transformerEnMaximierV1(M, n) :

```
| Pour k variant de la seconde position de M jusqu'à la dernière  
|   remonter(M, k)  
| Fin Pour
```

- Coût $C_1(n) = \sum_{k=2 \rightarrow n} R(k) = \sum_{k=2 \rightarrow n} \lfloor \log_2(k) \rfloor \leq \underbrace{\sum_{k=2 \rightarrow n} \log_2(k)}_{\log_2(n!)}$
- Formule de Stirling : $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right)$
- $C_1(n) = O(n \log(n))$

Opérations sur un tas de taille n

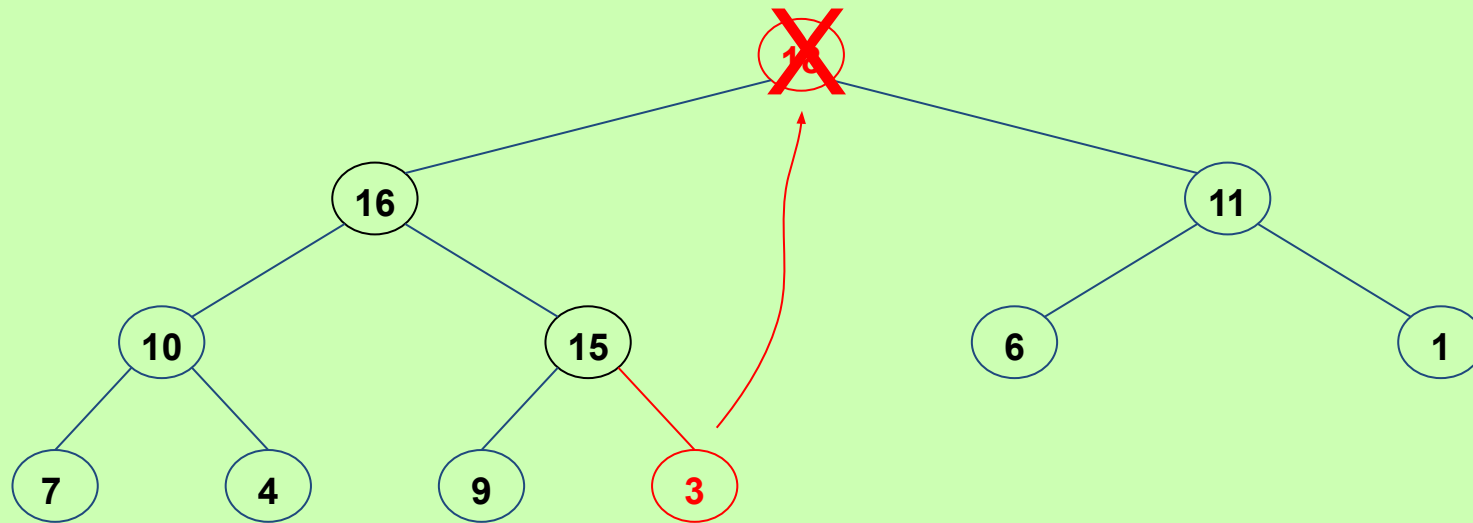
(3)

Suppression du maximum

- On substitue à l'élément maximum, l'élément le plus à droite du dernier niveau : $O(1)$
 - La propriété de quasi-complétude de l'arbre binaire est assurée
- On fait **descendre** l'élément placé à la racine jusqu'à ce que la condition maximière soit vérifiée
 - À chaque étape, lorsque la condition maximière n'est pas vérifiée pour le nœud courant, on le permute avec le plus grand de ses deux nœuds fils
- Le nombre de comparaisons et d'échanges est fonction de la hauteur de l'arbre : opération en $O(\log n)$

Exemple 2

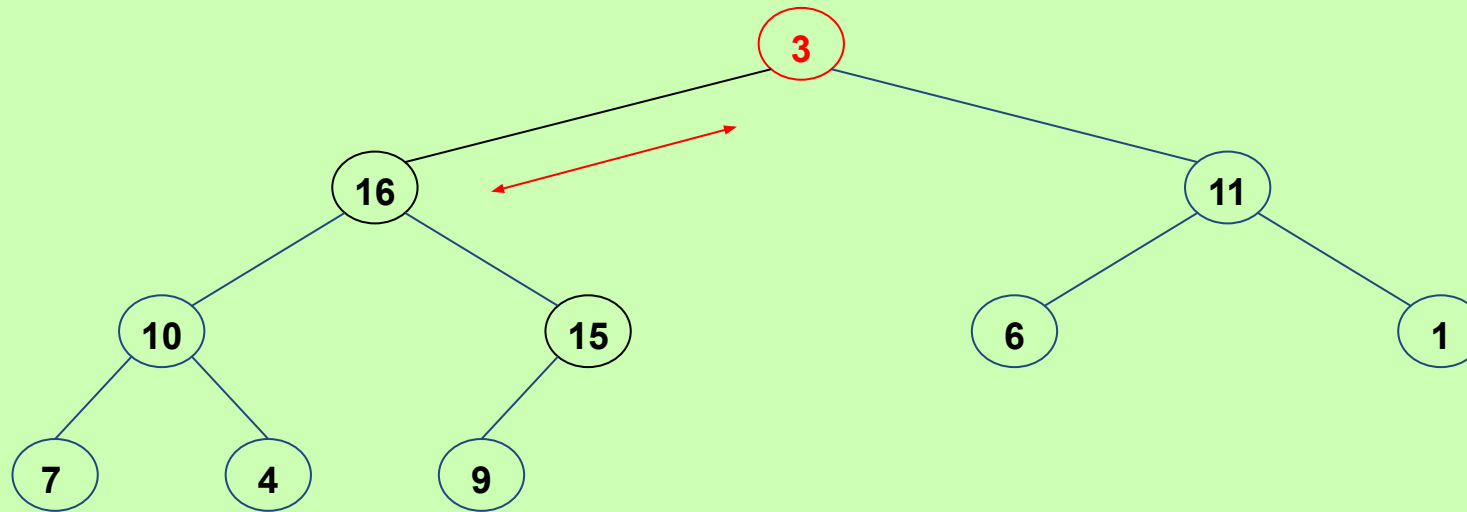
Suppression de 18 (1)



- Le nœud 18 est supprimé et remplacé par le nœud situé le plus à droite sur le dernier niveau

Exemple 2

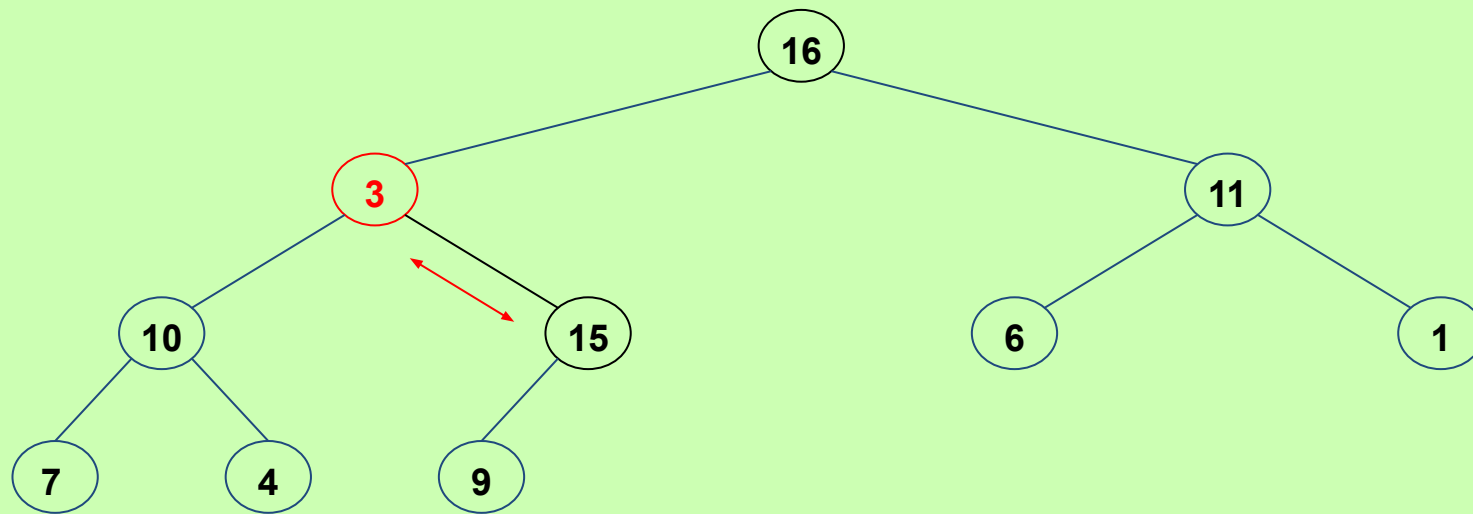
Suppression de 18 (2)



- La condition maximère n'est pas vérifiée pour le nœud courant (3), on l'échange avec le **plus grand** de ses nœuds fils (pour rétablir la condition maximère)
- Echange du nœud 16 et du nœud 3

Exemple 2

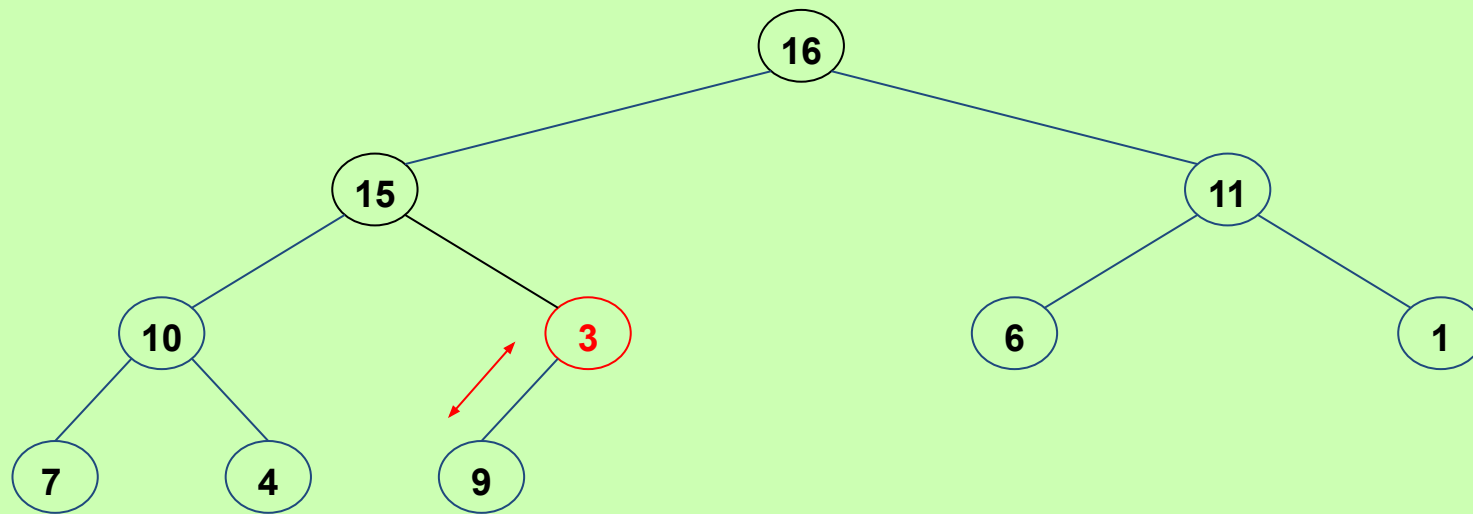
Suppression de 18 (3)



- La condition maximière n'est pas vérifiée pour le nœud courant (3), on l'échange avec le **plus grand** de ses nœuds fils
- Echange du nœud 3 avec le nœud 15

Exemple 2

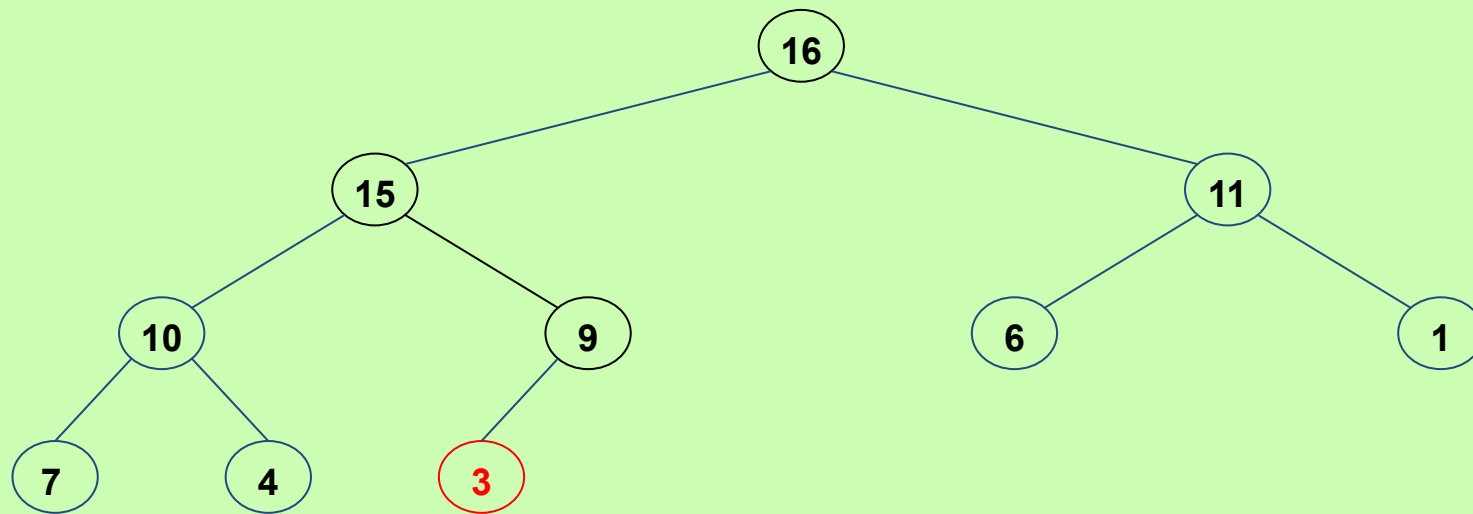
Suppression de 18 (4)



- La condition maximière n'est pas vérifiée pour le nœud courant (3), on l'échange avec son unique nœud fils
- Echange du nœud 3 avec le nœud 9

Exemple 2

Suppression de 18 (5)



- Le nœud courant n'a plus de fils, la condition maximière est désormais vérifiée pour le nœud courant (3)
- La procédure de suppression est terminée

Procédure **extraireMax**(M, n)

- Extraire l'élément maximum d'un maximier M de n éléments
 - Coût proportionnel à la hauteur de l'arbre **$O(\log(n))$**

extraireMax(M, n) :

```
Max ← VAL(premier nœud de M)
premier nœud de M ← dernier nœud de M    // MIEUX : échanger premier
supprimer le dernier                      // et dernier, et diminuer taille tas
descendre(M, indice du premier, n-1)
retourner Max
```

- Procédure **descendre**(M, k, n)
 - Faire descendre le nœud d'indice k dans un maximier M de n éléments dont seul l'élément racine est mal placé pour rétablir la condition maximière
 - Coût proportionnel à la hauteur du sous-arbre ancré en k

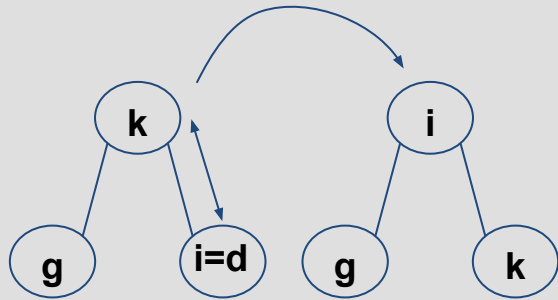
Procédure descendre(M, k, n)

Faire descendre le nœud d'indice k dans un maximier M de n éléments dont seul l'élément racine est mal placé pour rétablir la condition maximière

descendre(M, k, n)

```
Fin ← FEUILLE(k,n) // Fin si k est une feuille
Tant que pas Fin
  i ← FG(k) // FG existe obligatoirement !
  Si FD(k) existe ET si VAL(nœud FD(k)) > VAL(nœud FG(k))
    | i ← FD(k) // FD est plus grand !
  Fin si
  Si VAL(nœud k) ≥ VAL(nœud i)
    | Fin ← VRAI // Fin si k est à sa place
  Sinon
    | Echanger nœud i et nœud k // k descend vers
    | k ← i // son plus grand fils
    | Fin ← FEUILLE(k,n) // Fin si k devient une feuille
  Fin si
Fin tant que
```

Preuve de correction



descendre (M, k, n)

```

Fini ← FEUILLE(k,n) // Fini si k est une feuille
Tant que pas Fini
    i ← FG(k) // FG existe obligatoirement !
    Si FD(k) existe ET si VAL(nœud FD(k)) > VAL(nœud FG(k))
        i ← FD(k) // FD est plus grand !
    Fin si
    Si VAL(nœud k) ≥ VAL(nœud i)
        Fini ← VRAI // Fini si k est à sa place
    Sinon
        Echanger nœud i et nœud k // k descend vers
        k ← i // son plus grand fils
        Fini ← FEUILLE(k,n) // Fini si k devient une feuille
    Fin si
Fin tant que
    
```

- Hypothèse d'appel : on descend une valeur dans un maximier M, de n éléments, dont seul l'élément d'indice k ne vérifie pas la condition maximière
- Résultat à démontrer : après l'exécution de la fonction descendre, tous les nœuds vérifient la condition maximière
- Preuve :
 - Les seuls nœuds modifiés sont les nœuds présents sur la branche menant de l'élément k à une feuille
 - Il faut montrer que pour tout nœud n sur le chemin de descente de k jusqu'à une feuille, la condition maximière est respectée : $VAL(\text{nœud } n) \geq VAL(\text{nœud } FD(n))$ et $VAL(\text{nœud } n) \geq VAL(\text{nœud } FG(n))$
 - Il suffit de montrer que cette propriété est vraie après chaque échange
 - Le nœud i est le plus grand des deux fils
 - Si l'échange n'a pas lieu, c'est que le plus grand des deux fils est inférieur à k, donc que k vérifie déjà la condition maximière : $VAL(k) \geq VAL(i) \geq VAL(\text{second fils})$
 - L'échange a lieu si $VAL(i) > VAL(k)$. Comme i est le plus grand des deux fils, on a aussi $VAL(i) > VAL(\text{second fils})$. Après l'échange, on a donc bien $VAL(i) > VAL(g)$ et $VAL(i) > VAL(d)$

Opérations sur un tas de taille n

(4)

Création d'un maximier V2

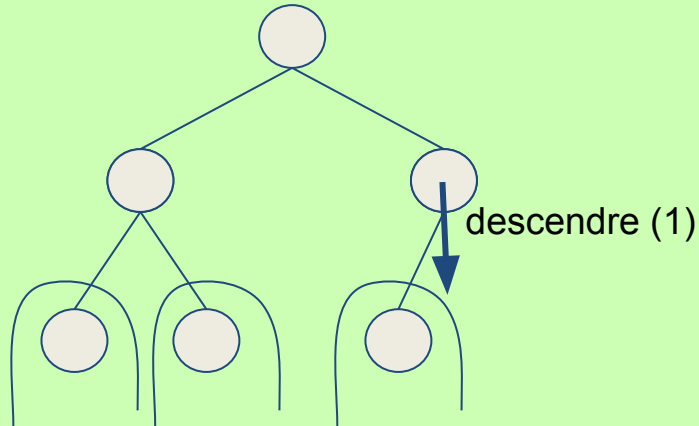
- La procédure descendre peut aussi être utilisée pour transformer un tableau quelconque de n éléments en maximier
 - On considère que les feuilles sont **déjà** des maximiers
 - Puisqu'elles n'ont pas de fils !
 - Il suffit de faire redescendre chaque nœud interne, en partant du dernier et en remontant jusqu'à la racine

transformerEnMaximierV2 (M, n) :

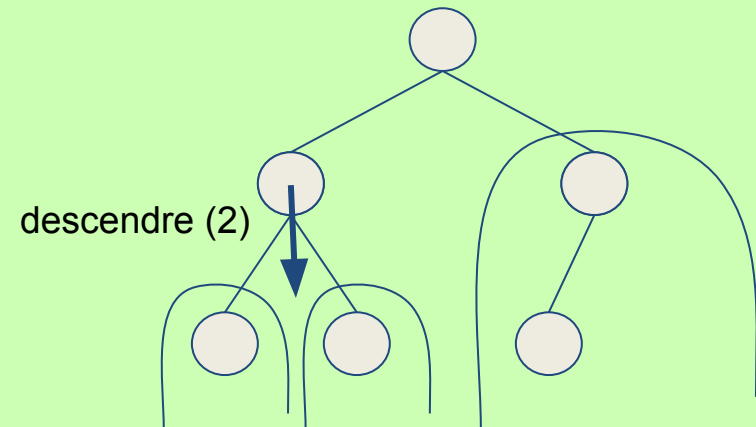
```
Pour k variant de l'indice du dernier nœud interne jusqu'à la racine
|   descendre (M, k, n)
Fin Pour
```

Illustration

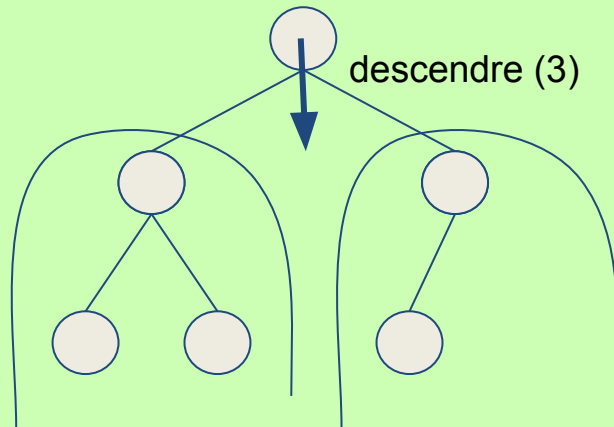
Transformer en maximier v2



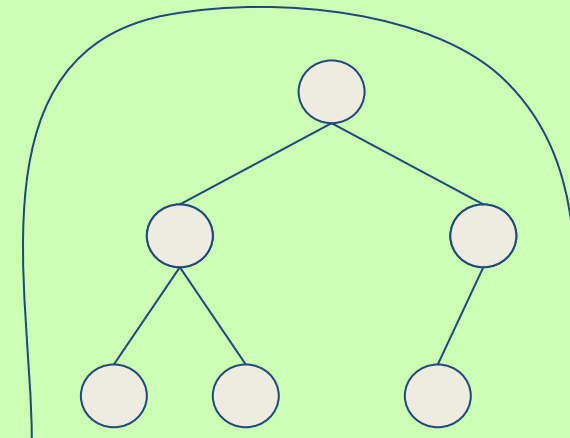
Les feuilles sont déjà des maximiers
On fait descendre le dernier nœud interne



Le sous-arbre correspondant est devenu un maximier
On fait descendre le prochain nœud interne



Le sous-arbre correspondant est devenu un maximier
On fait descendre le prochain nœud interne

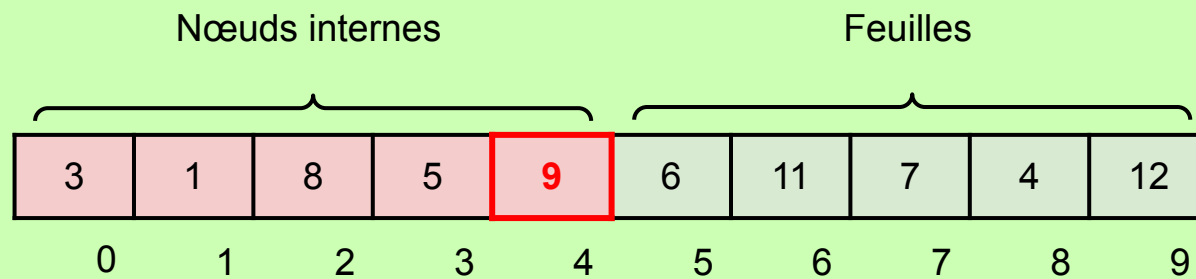
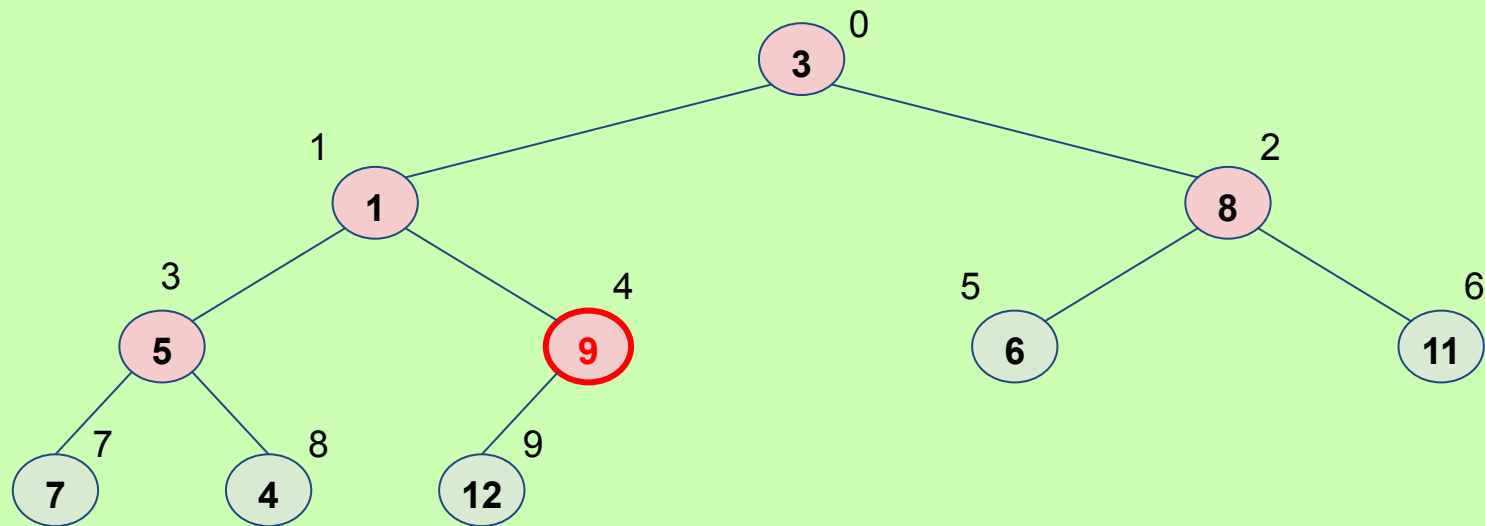


Tout l'arbre est devenu un maximier !

Exemple 3

Transformer en maximier v2

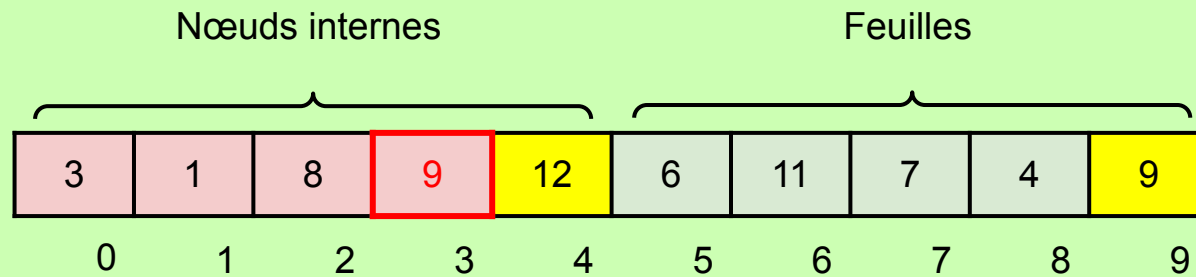
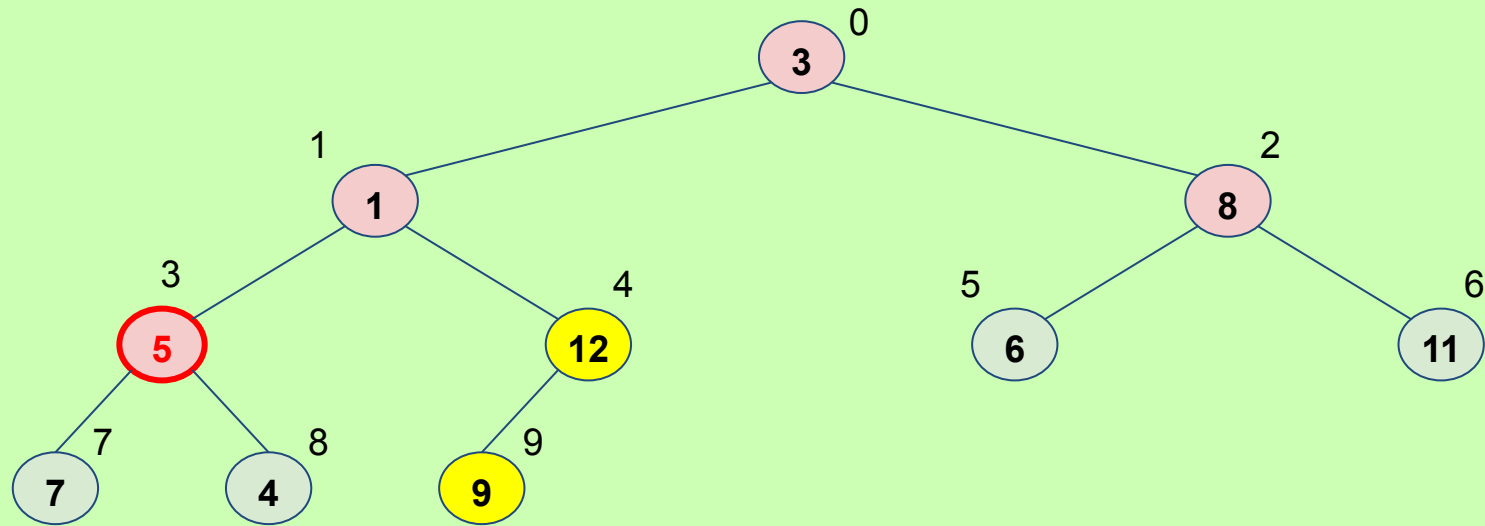
(1)



Exemple 3

Transformer en maximier v2

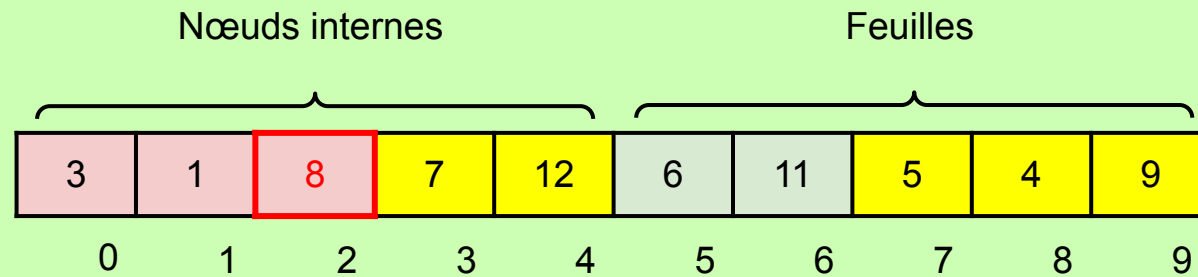
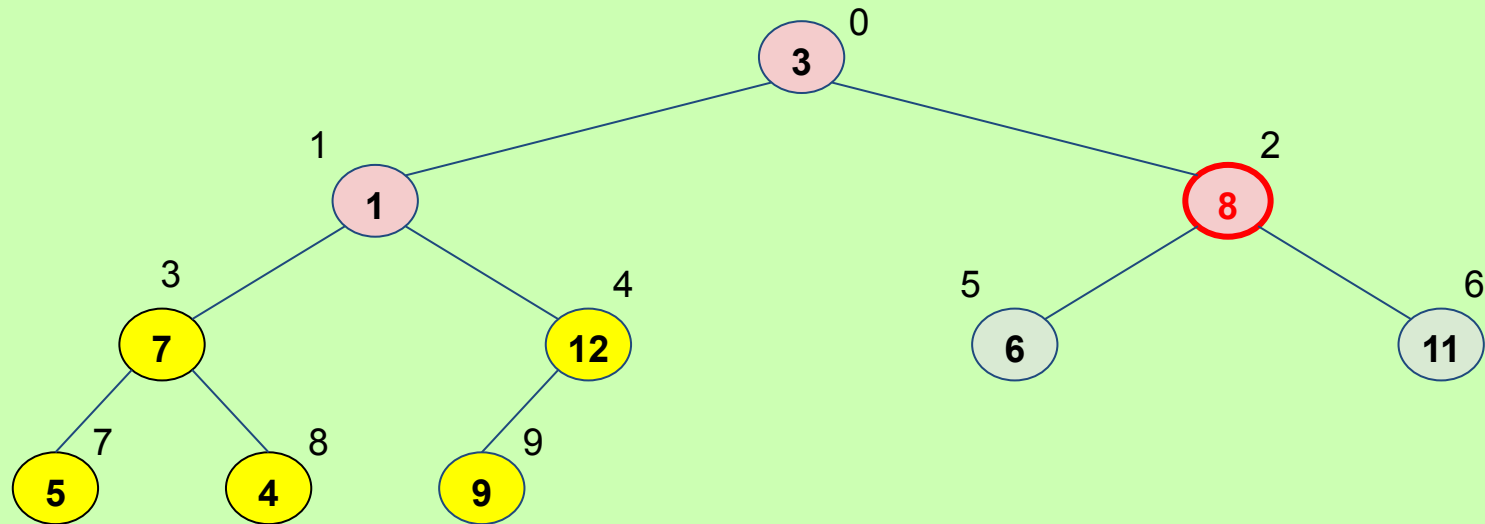
(2)



Exemple 3

Transformer en maximier v2

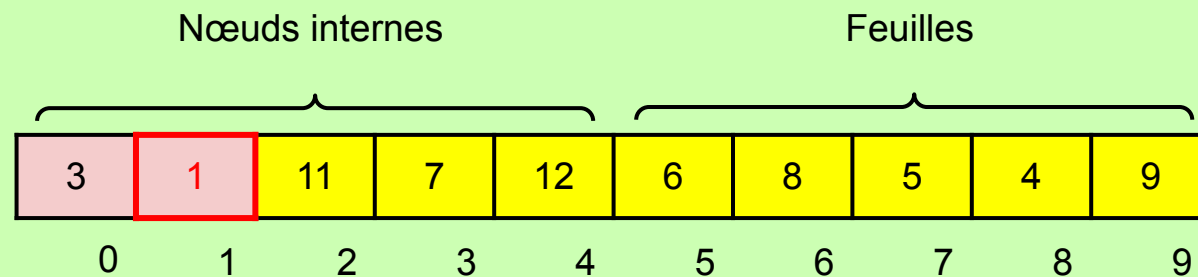
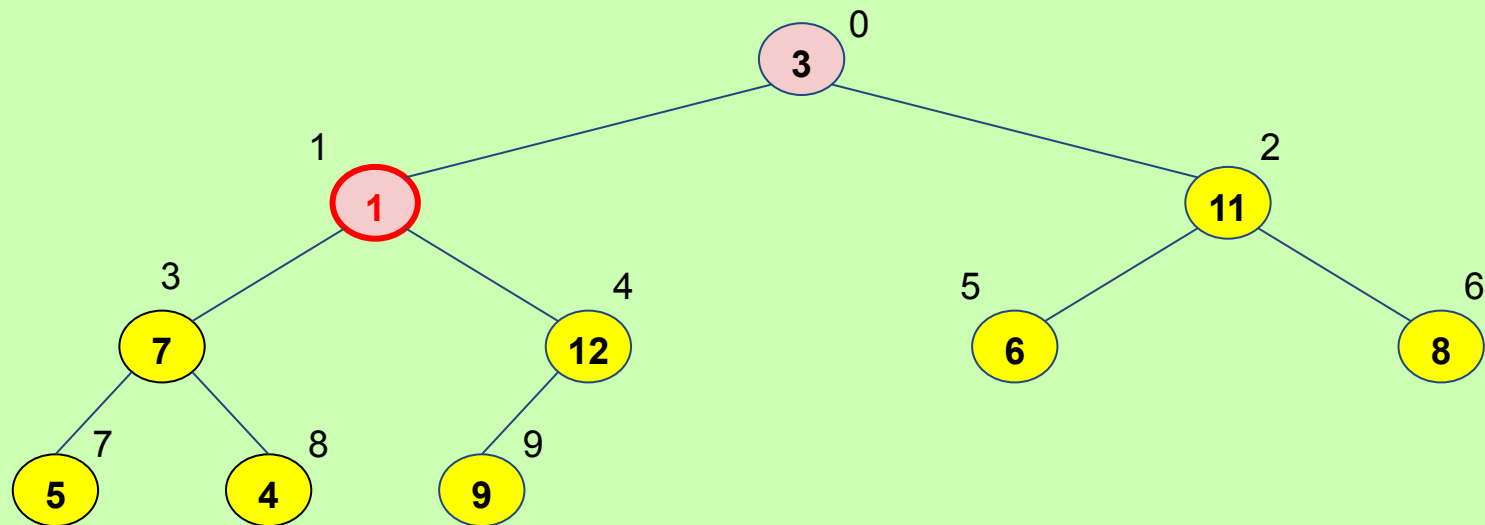
(3)



Exemple 3

Transformer en maximier v2

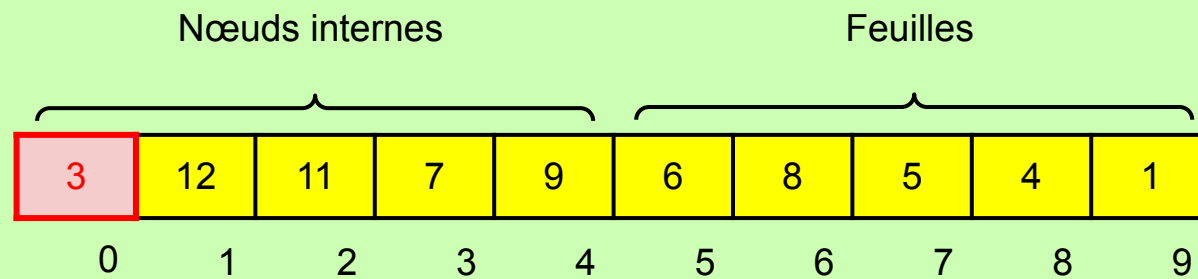
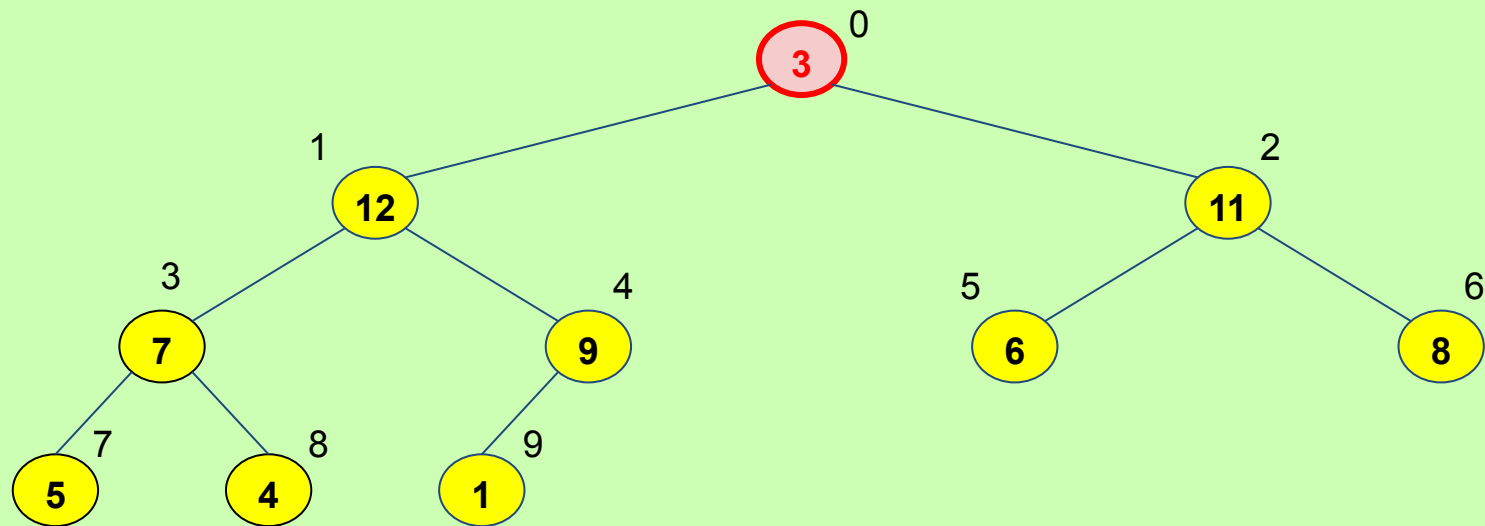
(4)



Exemple 3

Transformer en maximier v2

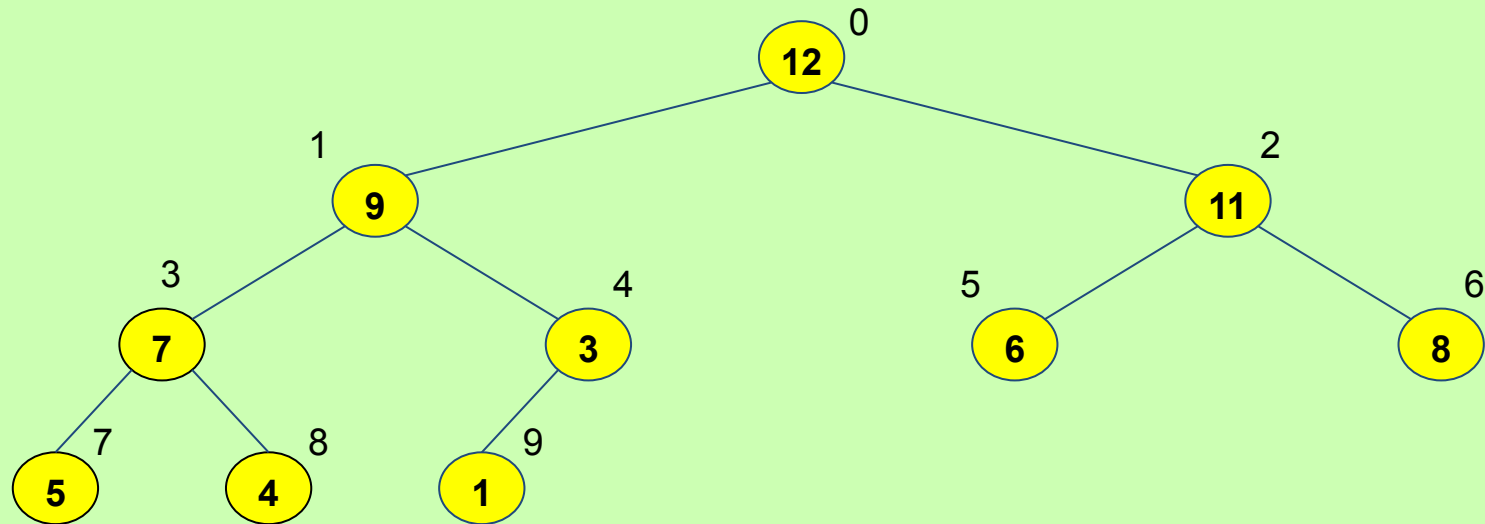
(5)



Exemple 3

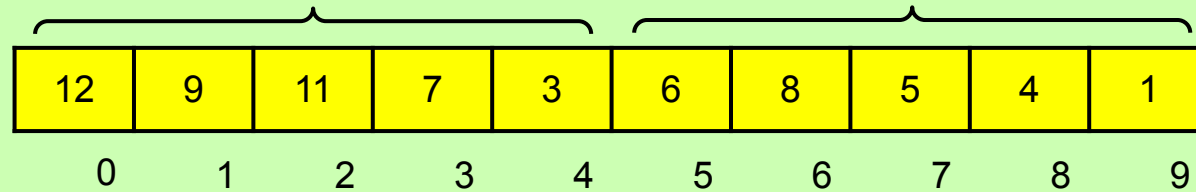
Transformer en maximier v2

(6)



Nœuds internes

Feuilles

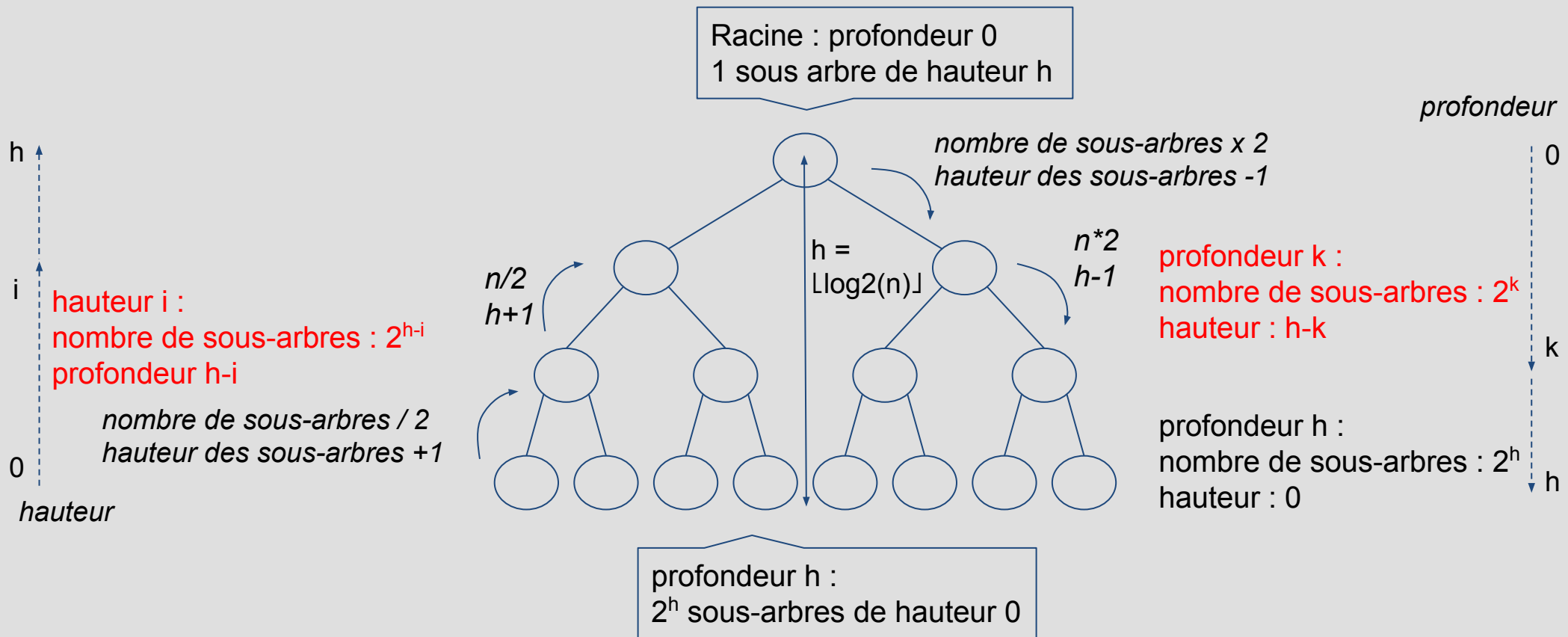


Création d'un maximier V2

Analyse fine du coût

- Il faut appliquer descendre dans tous les sous-arbres de hauteur non nulle
 - 2^{h-i} sous-arbres de hauteur i
 - Coût de descendre dans un sous-arbre de hauteur h : $O(h)$
- Coût $C_2(n) = \sum_{i=1 \rightarrow h} 2^{h-i} O(i) = O(\sum_{i=1 \rightarrow h} i 2^{h-i}) = O(2^h \sum_{i=1 \rightarrow h} i / 2^i) = O(2^h \sum_{i=1 \rightarrow h} i \frac{1}{2^i})$
- Hauteur d'un tas de n éléments : $h = \lfloor \log_2(n) \rfloor$
 $\Rightarrow C_2(n) = O(n \sum_{i=1 \rightarrow h} i \frac{1}{2^i})$
- $\sum_{h=0 \rightarrow k} x^h = x^{k+1} - 1 / x - 1 = \sum_{h=0 \rightarrow \infty} x^h = 1 / 1 - x$ si $|x| < 1$
- Par dérivation : $\sum_{h=0 \rightarrow \infty} h x^{h-1} = 1 / (1-x)^2 \Rightarrow \sum_{h=0 \rightarrow \infty} h x^h = x / (1-x)^2$
- Pour $x = 1/2$: $\sum_{h=0 \rightarrow \infty} h x^h = x / (1-x)^2 = 1/2 / 1/4$
- $C_2(n) = O(2n) = O(n)$: complexité linéaire

Nombre de sous-arbres dans un arbre binaire complet ?



Implémentation

Structure de tas

- Implémentation dynamique
- **nbElt** : nb effectif de nœuds
- **nbMaxElt** : capacité maximale du tas
- **tree** : table contenant les valeurs des nœuds organisés sous la forme d'un arbre partiellement ordonné

```
typedef struct {  
    unsigned int nbElt;  
    unsigned int nbMaxElt;  
    T_elt * tree;  
} T_heap;
```

Macro fonctions

```
#define iPARENT(i)          (i-1) / 2
#define iLCHILD(i)          (2*i) + 1
#define iRCHILD(i)          (2*i) + 2
#define iLASTINTERNAL(n)    n/2 - 1
#define isINTERNAL(i,n)     (2*i < (n-1))
#define isLEAF(i,n)         (2*i >= (n-1))
#define isINTREE(i,n)       (i < n)
#define isROOT(i)           (i == 0)
#define nbINTERNALS(n)      n/2
#define nbLEAVES(n)         (int) ceil((double)n/2)
```

- $PERE(i) : \lfloor (i-1)/2 \rfloor$
- $FG(i) : 2i + 1$
- $FD(i) : 2i + 2$
- $INTERNE(i,n) : i < (n-1)/2$
- $FEUILLE(i,n) : i \geq (n-1)/2$
- $\lfloor n/2 \rfloor$ nœuds internes
- $\lceil n/2 \rceil$ feuilles

```
ceil, floor : #include <math.h>
```



Exercice corrigé 2

```
T_heap * newHeap(unsigned int nbMaxElt);
```

```
void freeHeap(T_heap *p);
```

```
T_heap * initHeap(T_elt t[], int n) ;
```

```
    // Utilisation de memcpy
```

```
void showHeap(T_heap *p);
```

```
void showHeap_rec(T_heap *p, int root, int indent);
```



Exercice 2

descendre
alias “tamiser” ou “entasser”
⇔ sift en anglais

```
void swap(T_heap *p, int i, int j);  
void siftUp(T_heap *p, int k); // Tester ex1  
void addElt(T_heap *p, T_elt e);  
void buildHeapV1(T_heap * p); // Tester ex4  
  
void siftDown(T_heap *p, int k);  
T_elt getMax(const T_heap *p);  
T_elt removeMax(T_heap *p); // Tester ex2  
void buildHeapV2(T_heap * p); // Tester ex3
```



Tri par tas

Histoire

- Version initiale publiée en 1962 par Robert W. Floyd (1936-2001) sous le nom Treesort
- Version améliorée en juin 1964 par J. W. J. Williams sous le nom Heapsort
- Version à nouveau améliorée par Robert W. Floyd en décembre 1964
- Algorithme de tri en $O(n \log n)$: complexité optimale

Principe

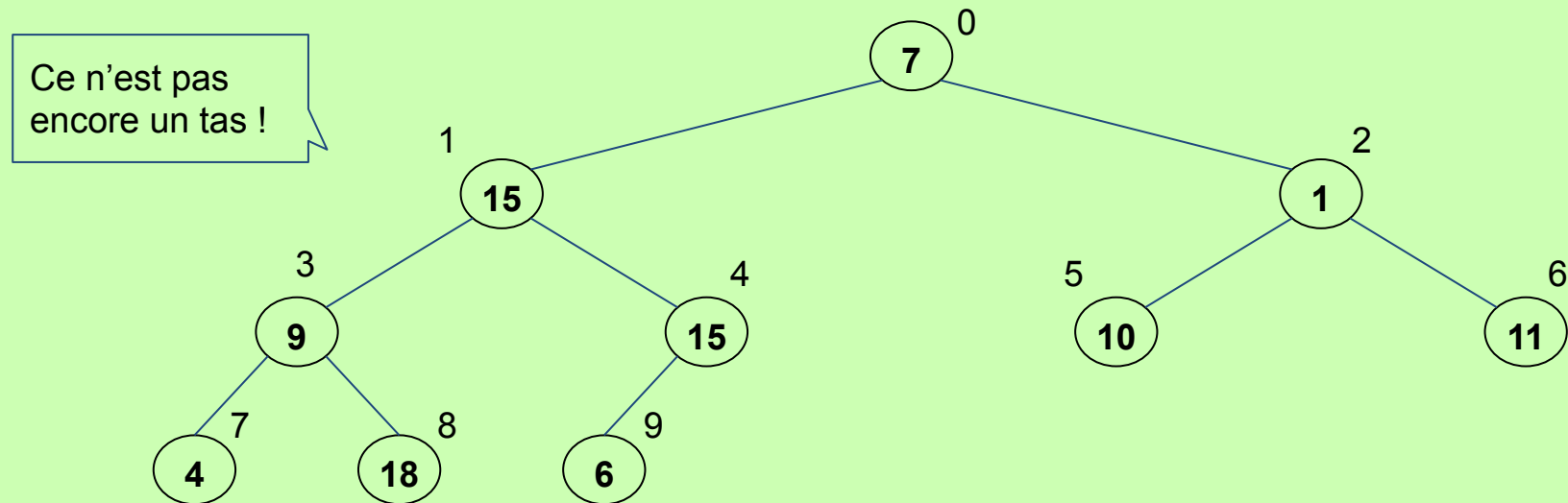
- Organiser l'ensemble à trier en tas
 - $O(n \log n)$ (V1) ou $O(n)$ (V2)
- Répéter $n-1$ fois :
 - extraire l'élément maximum et le placer en fin du tas
 - Là où se situait l'élément le plus à droite du dernier niveau, avant la suppression du maximum
 - $O(n \log n)$ pour la seconde phase de l'algorithme
- Le tableau dans lequel le tas avait été formé est maintenant ordonné en $O(n \log n)$

Exemple 4

- Soit le tableau à ordonner de 10 éléments :

7	15	1	9	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9

- Représentation sous forme d'arbre binaire quasi-complet :



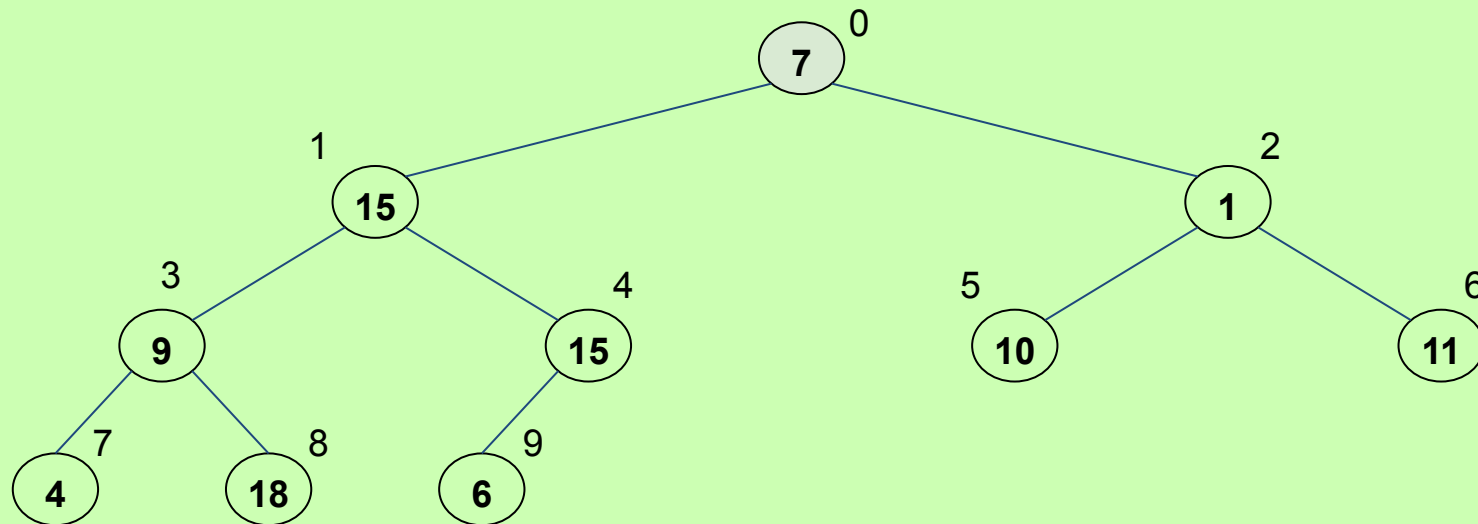
Ex4

Transformer en maximier v1 (1)

- Dans son état initial, seul le premier élément forme un tas

7	15	1	9	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9

- Nous allons insérer dans ce tas chacun des éléments suivants



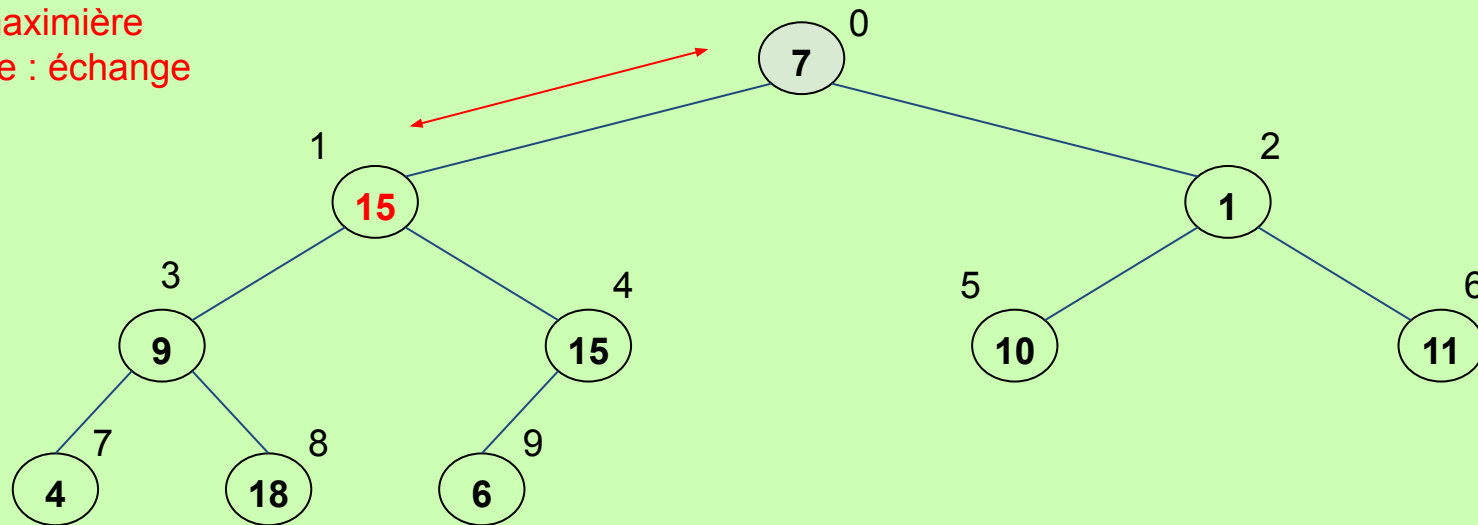
Ex4

Transformer en maximier v1 (2)

- Insertion de 15 et remontée :

7	15	1	9	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9

La condition maximière
n'est pas valide : échange
du 15 et du 7

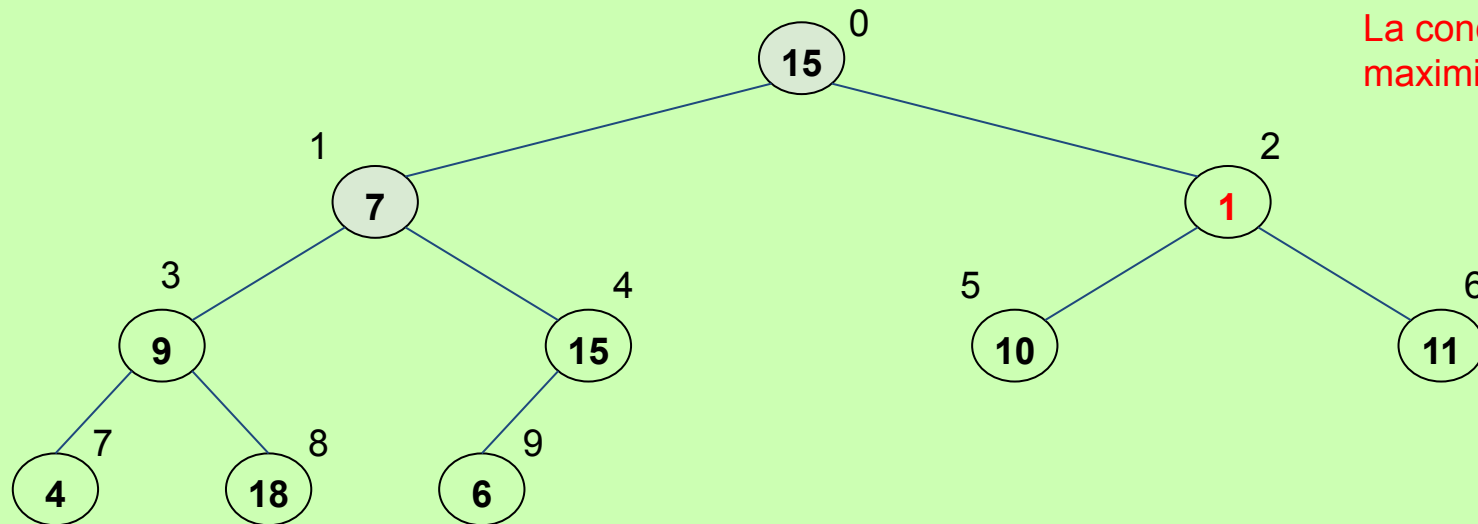


Ex4

Transformer en maximier v1 (3)

- Insertion de 1 et remontée :

15	7	1	9	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9



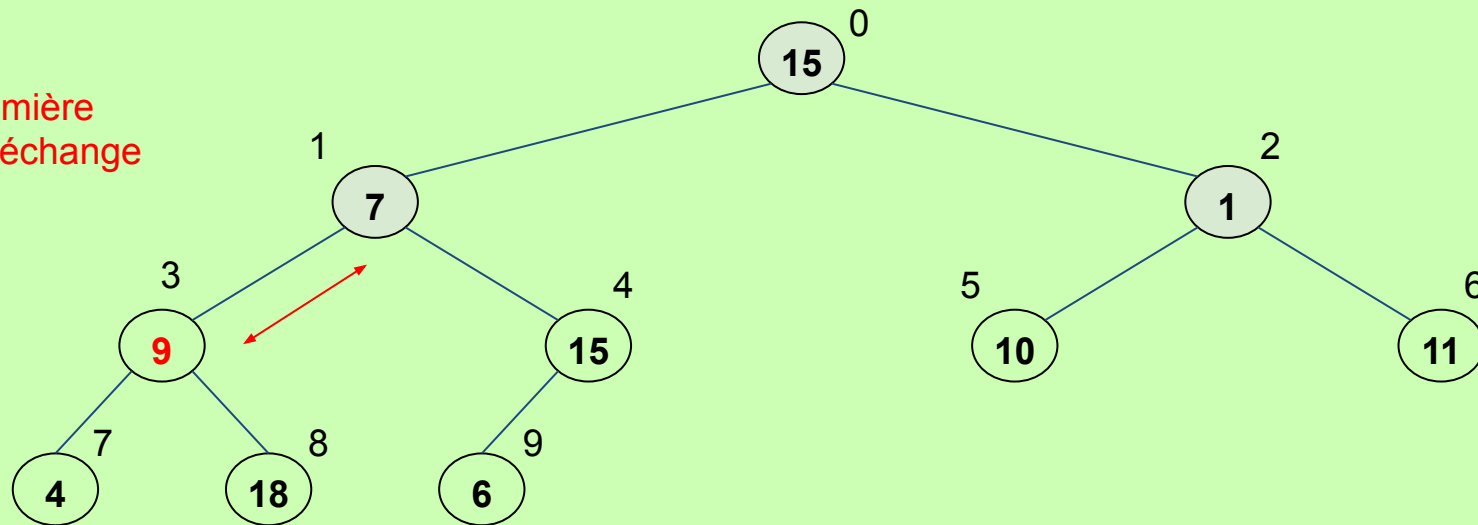
La condition
maximière est valide

Ex4

Transformer en maximier v1 (4)

- Insertion de 9 et remontée :

15	7	1	9	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9



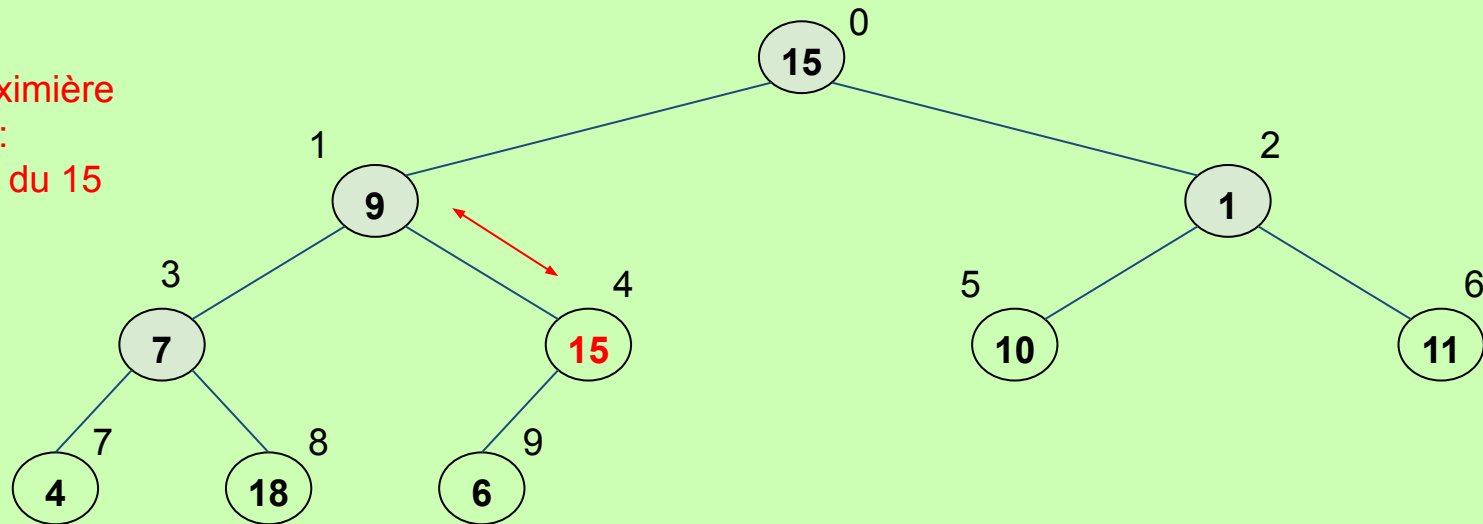
La condition maximère
n'est pas valide : échange
du 7 et du 9

Ex4

Transformer en maximier v1 (5)

- Insertion de 15 et remontée :

15	9	1	7	15	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9



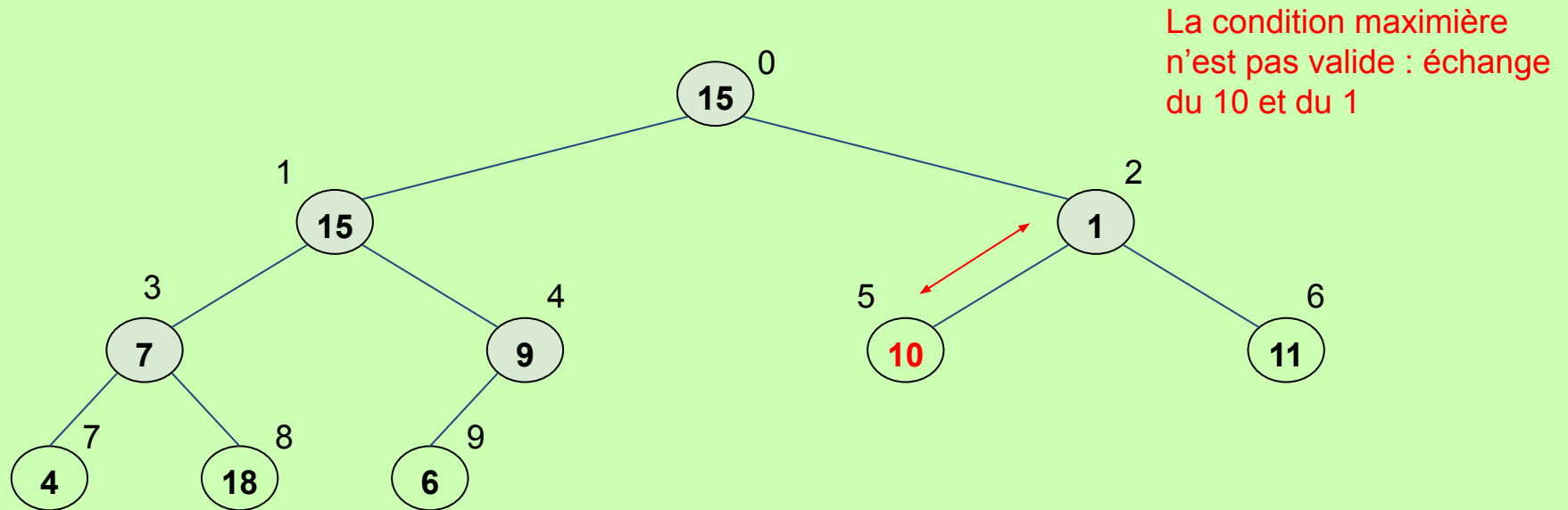
La condition maximière
n'est pas valide :
échange du 9 et du 15

Ex4

Transformer en maximier v1 (6)

- Insertion de 10 et remontée :

15	15	1	7	9	10	11	4	18	6
0	1	2	3	4	5	6	7	8	9

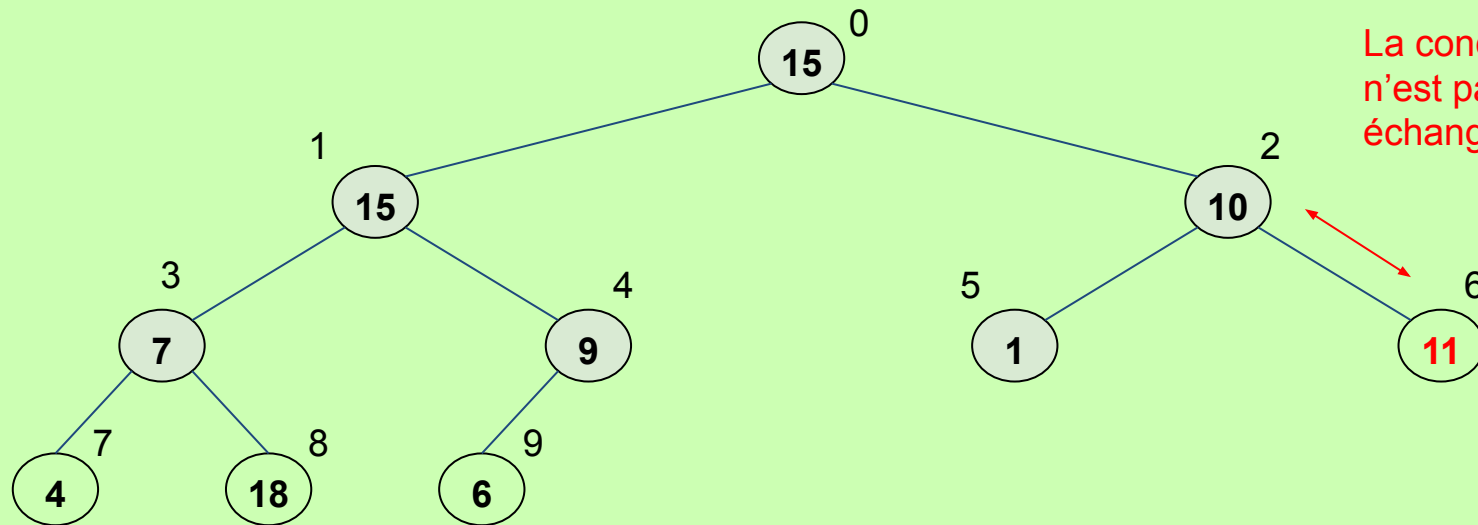


Ex4

Transformer en maximier v1 (7)

- Insertion de 11 et remontée :

15	15	10	7	9	1	11	4	18	6
0	1	2	3	4	5	6	7	8	9



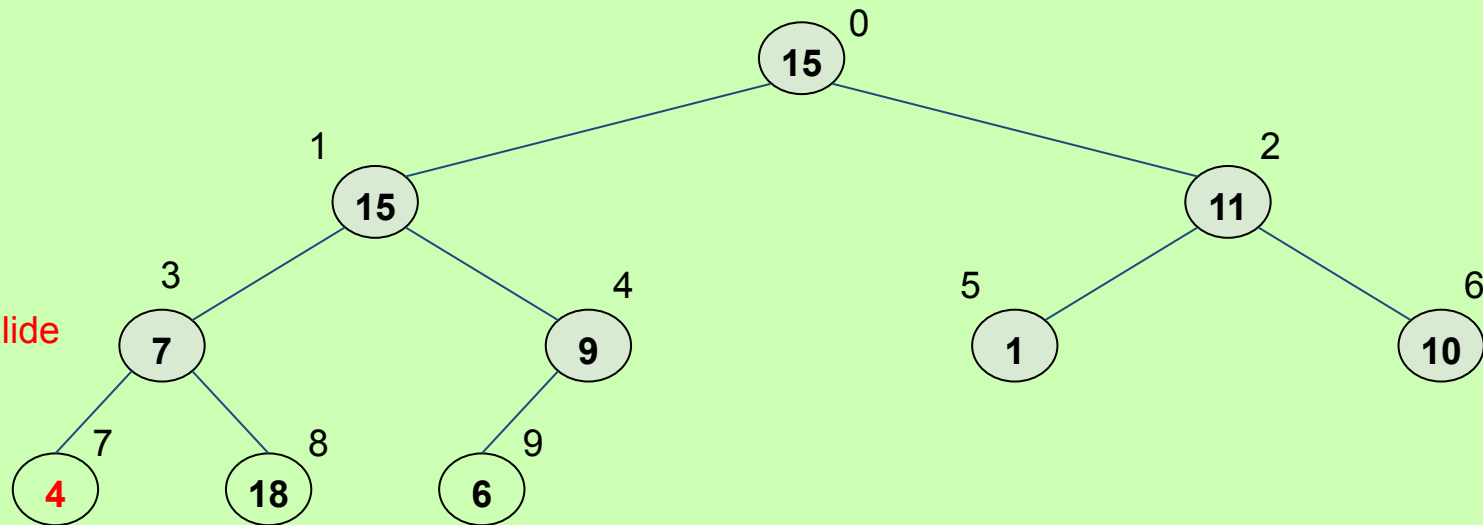
La condition maximière
n'est pas valide :
échange du 10 et du 11

Ex4

Transformer en maximier v1 (8)

- Insertion de 4 et remontée :

15	15	11	7	9	1	10	4	18	6
0	1	2	3	4	5	6	7	8	9



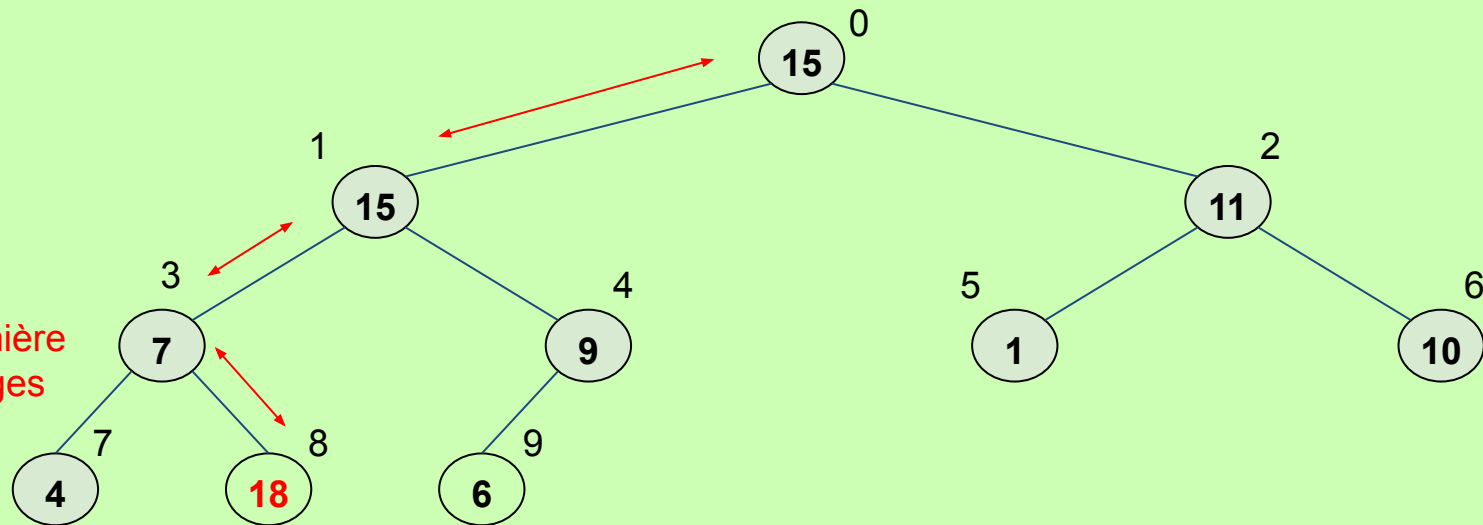
La condition
maximière est valide

Ex4

Transformer en maximier v1 (9)

- Insertion de 18 et remontée :

15	15	11	7	9	1	10	4	18	6
0	1	2	3	4	5	6	7	8	9



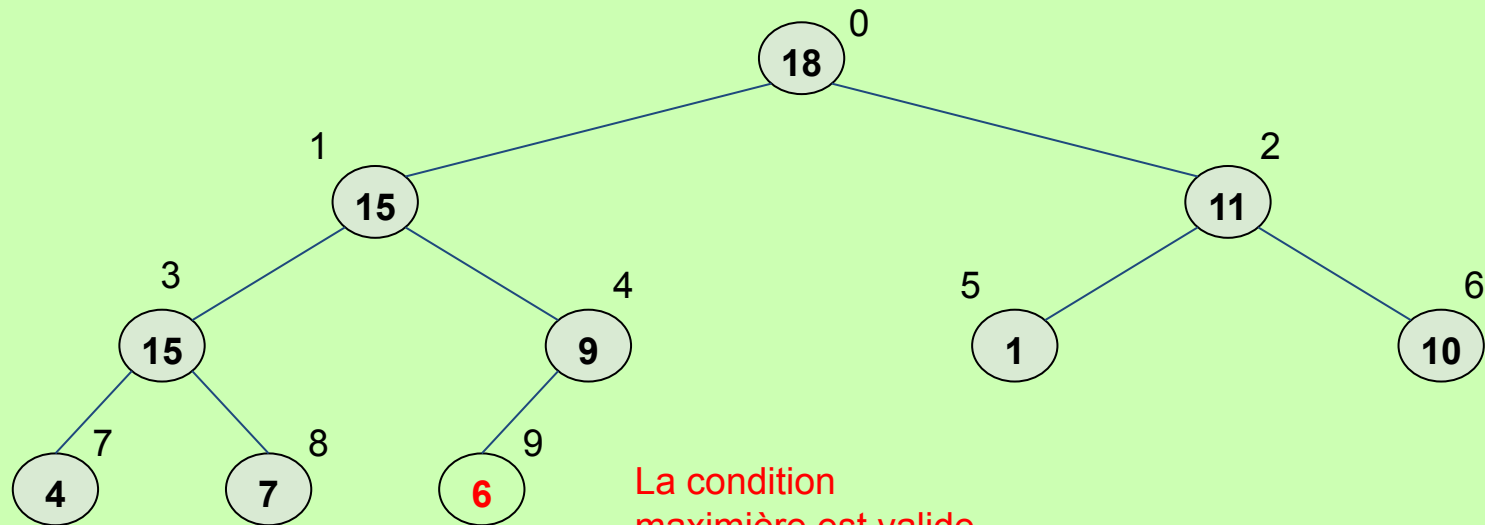
Condition maximière
invalide : échanges

Ex4

Transformer en maximier v1 (10)

- Insertion de 6 et remontée :

18	15	11	15	9	1	10	4	7	6
0	1	2	3	4	5	6	7	8	9

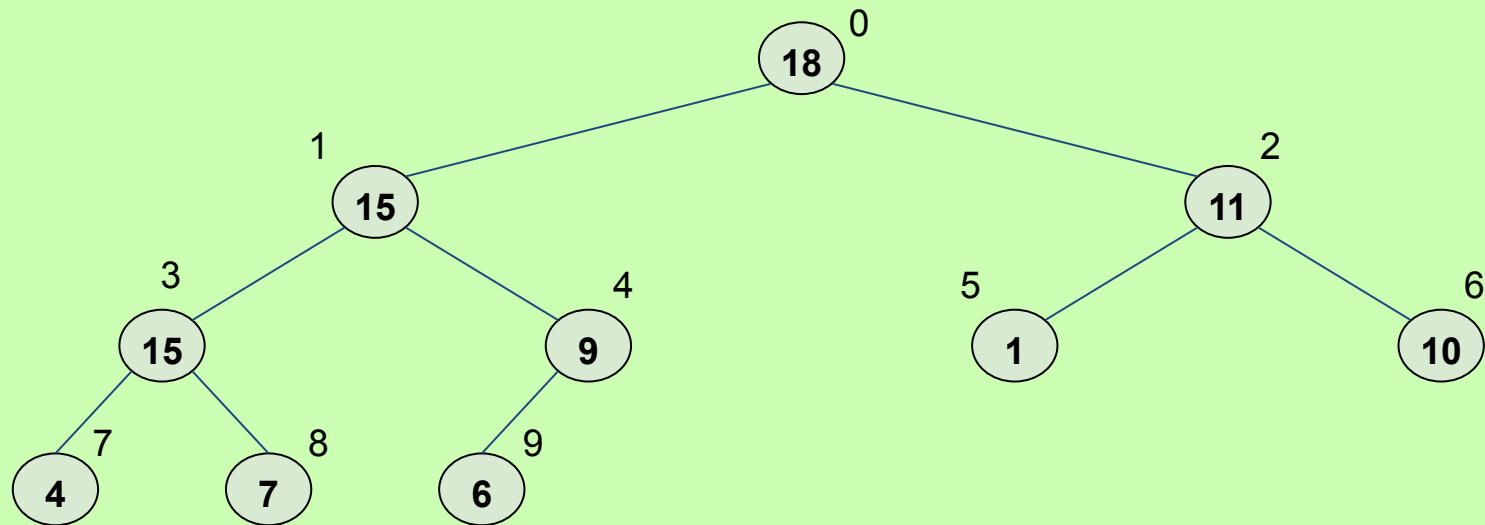


Ex4

Transformer en maximier v1 (11)

- La construction du tas est terminée

18	15	11	15	9	1	10	4	7	6
0	1	2	3	4	5	6	7	8	9



Deuxième phase du tri

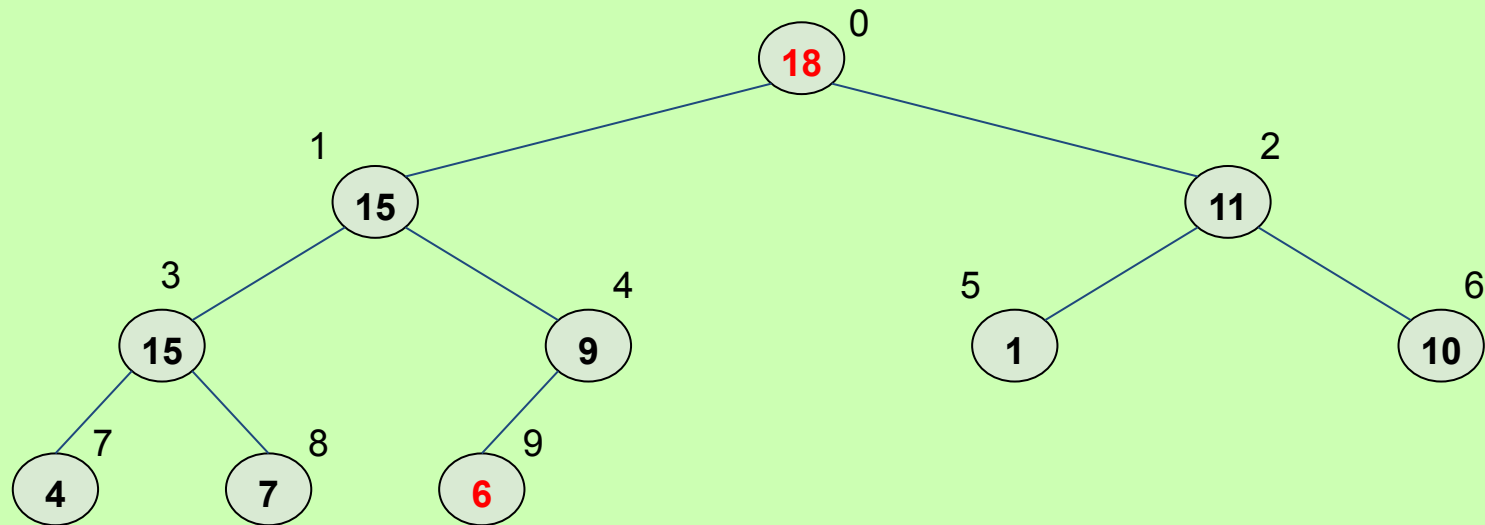
- Phase de traitement itératif dans laquelle à chaque étape :
 - On extrait le maximum du tas
 - On place le maximum extrait à l'endroit où était situé le dernier élément du tas (au niveau le plus bas, le plus à droite)
 - On réorganise le tas (comme dans l'opération de suppression du maximum)
- A chaque étape la taille du tas est réduite d'une unité et un élément a été rangé à sa place définitive
- Cette phase se termine lorsque le tas est réduit à seul élément

Ex4

Extraction du maximum (1)

- Extraction du max 18, échange avec le dernier (du tas actuel) et redescente de 6

18	15	11	15	9	1	10	4	7	6
0	1	2	3	4	5	6	7	8	9

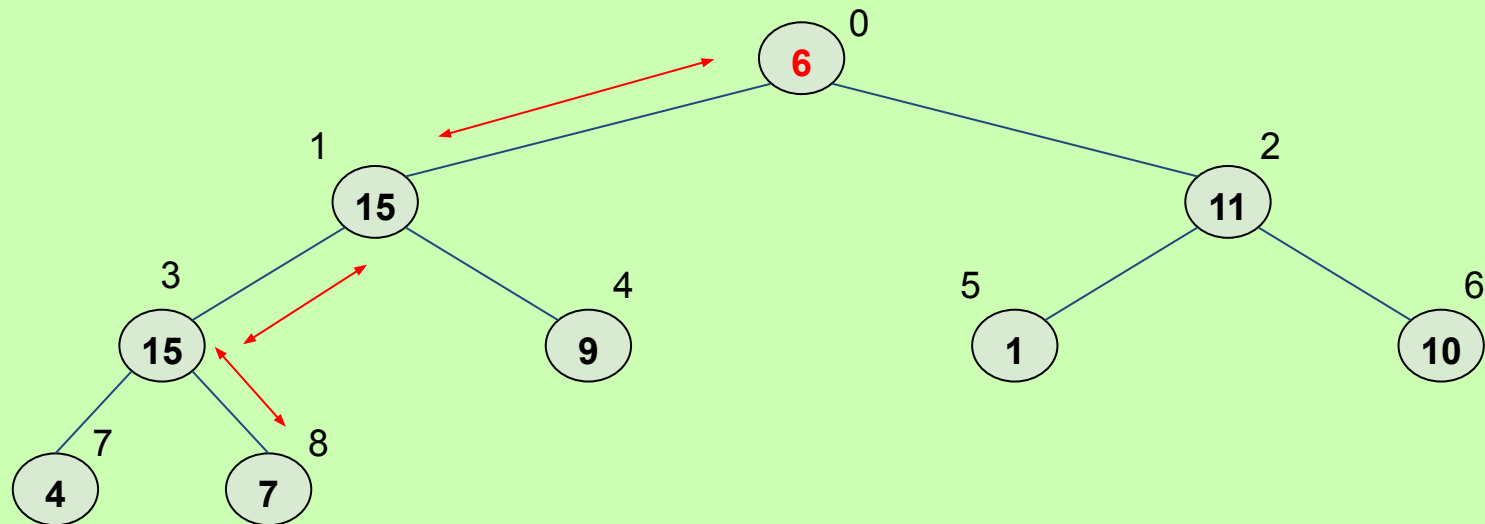


Ex4

Extraction du maximum (2)

- Extraction du max 18, échange avec le dernier (du tas actuel) et redescente de 6

6	15	11	15	9	1	10	4	7	18
0	1	2	3	4	5	6	7	8	9

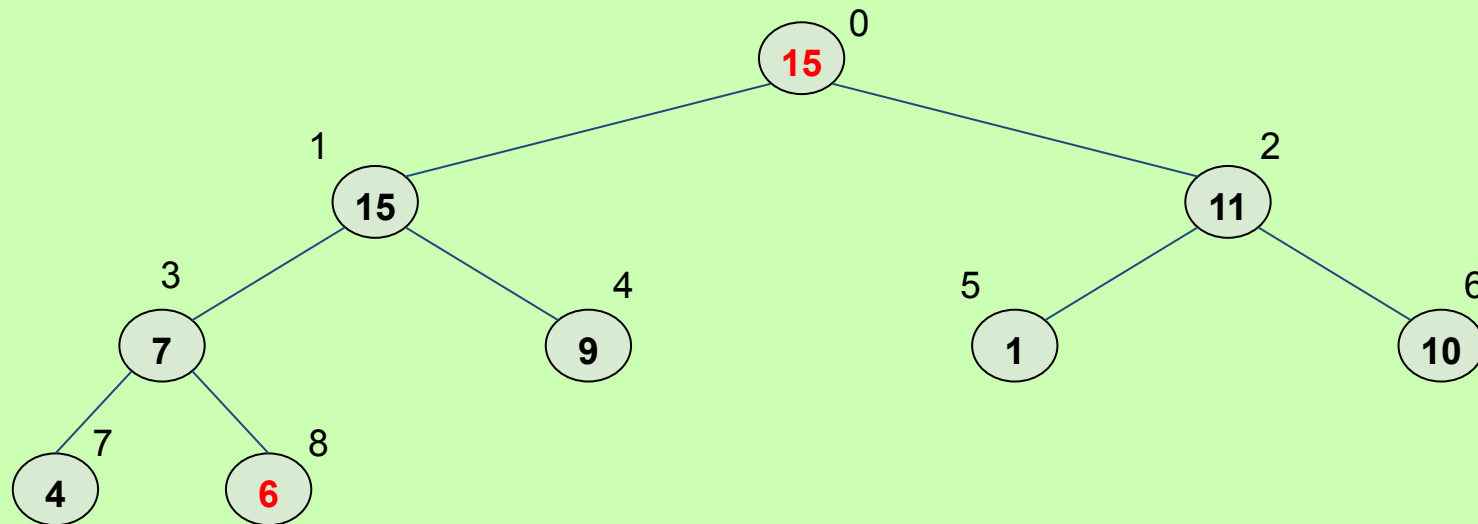


Ex4

Extraction du maximum (3)

- Extraction du max 15, échange avec le dernier (du tas actuel) et redescente de 6

15	15	11	7	9	1	10	4	6	18
0	1	2	3	4	5	6	7	8	9

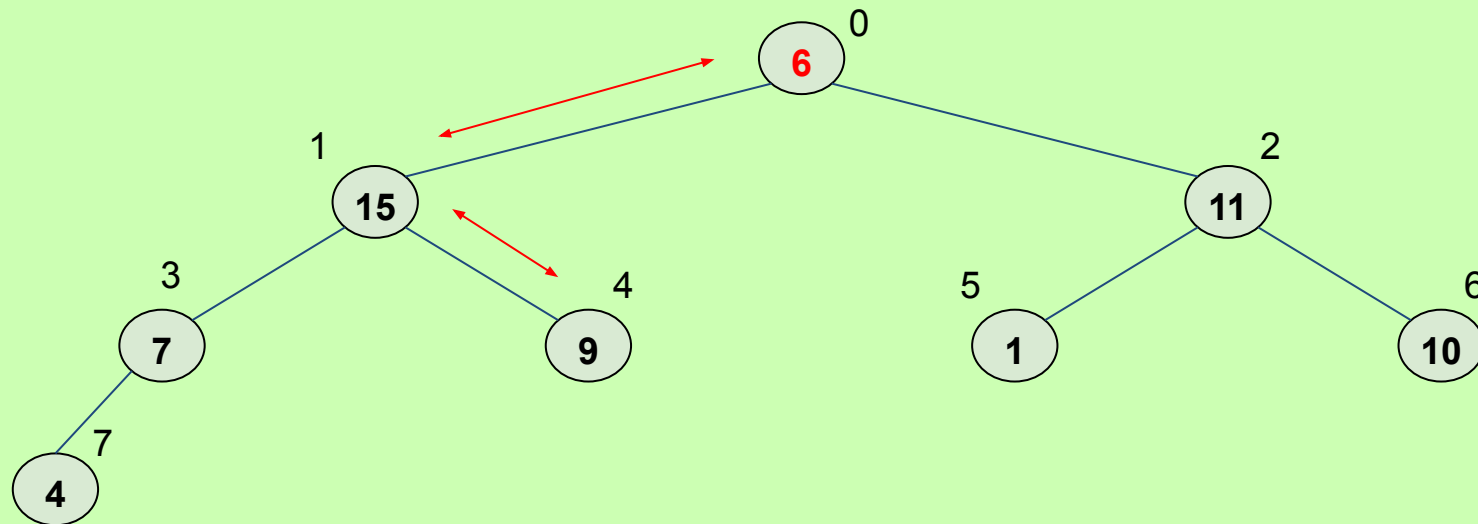


Ex4

Extraction du maximum (4)

- Extraction du max 15, échange avec le dernier (du tas actuel) et redescente de 6

6	15	11	7	9	1	10	4	15	18
0	1	2	3	4	5	6	7	8	9

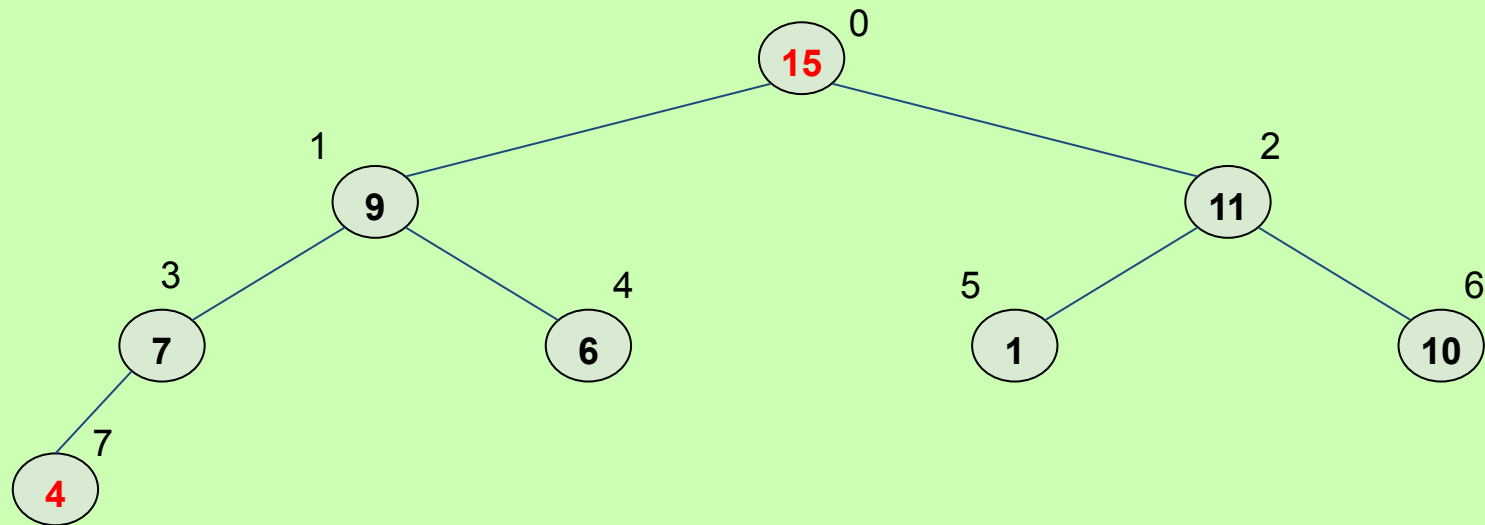


Ex4

Extraction du maximum (5)

- Extraction du max 15, échange avec le dernier (du tas actuel) et redescente de 4

15	9	11	7	6	1	10	4	15	18
0	1	2	3	4	5	6	7	8	9

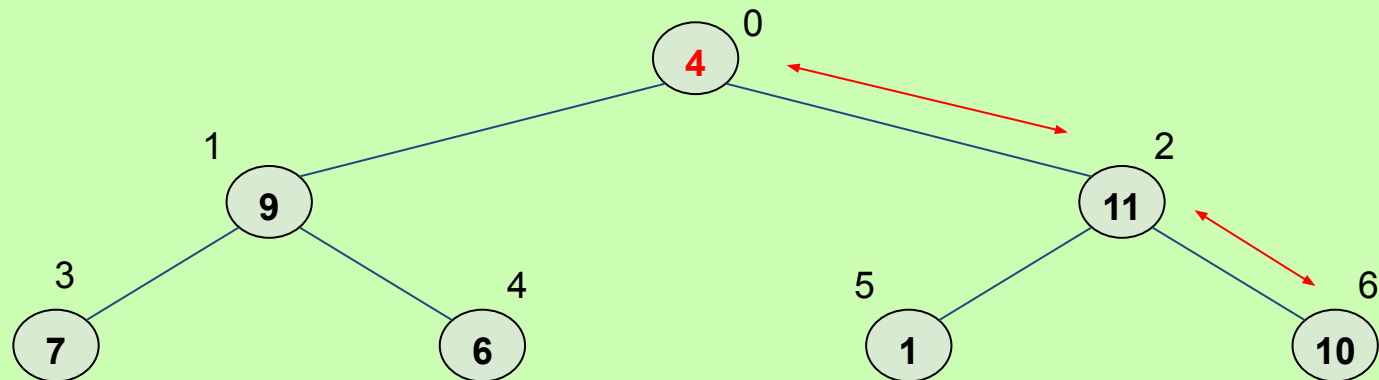


Ex4

Extraction du maximum (6)

- Extraction du max 15, échange avec le dernier (du tas actuel) et redescente de 4

4	9	11	7	6	1	10	15	15	18
0	1	2	3	4	5	6	7	8	9



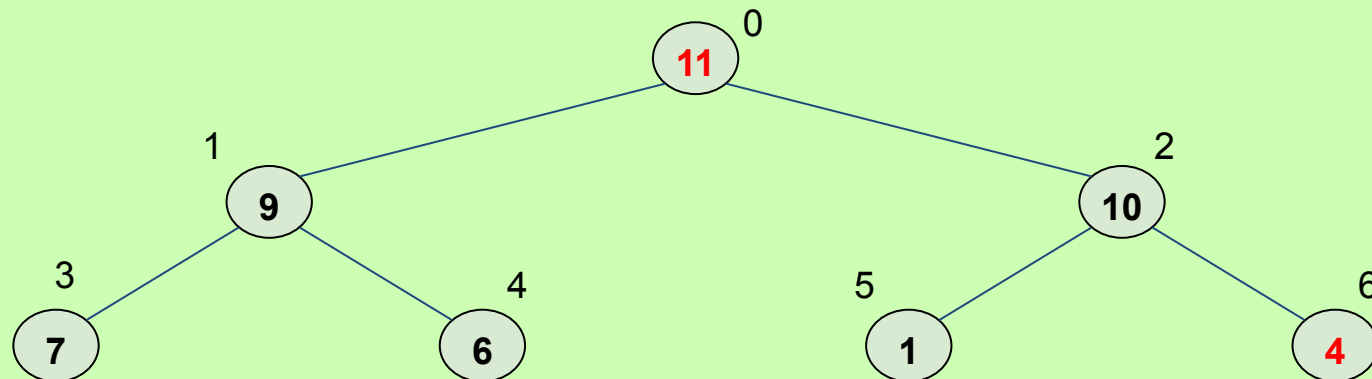
Ex4

Extraction du maximum

(7)

- Extraction du max 11, échange avec le dernier (du tas actuel) et redescente de 4

11	9	10	7	6	1	4	15	15	18
0	1	2	3	4	5	6	7	8	9

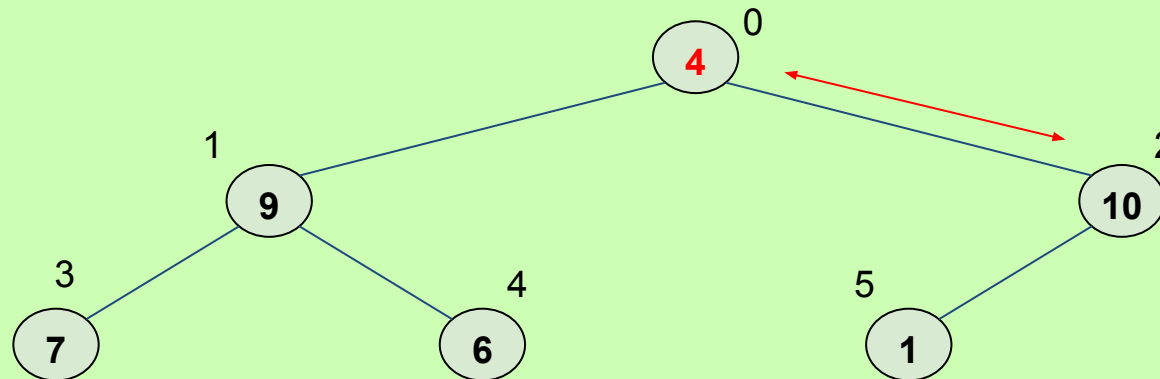


Ex4

Extraction du maximum (8)

- Extraction du max 11, échange avec le dernier (du tas actuel) et redescende de 4

4	9	10	7	6	1	11	15	15	18
0	1	2	3	4	5	6	7	8	9

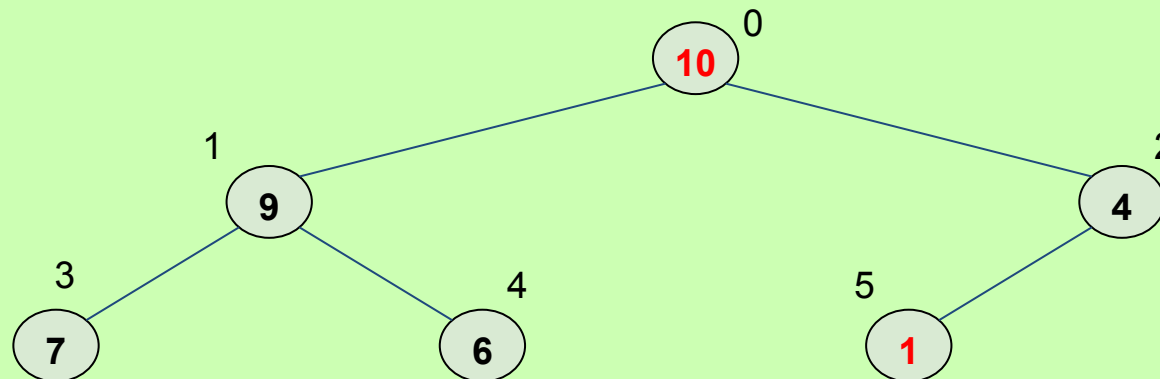


Ex4

Extraction du maximum (9)

- Extraction du max 10, échange avec le dernier (du tas actuel) et redescente de 1

10	9	4	7	6	1	11	15	15	18
0	1	2	3	4	5	6	7	8	9

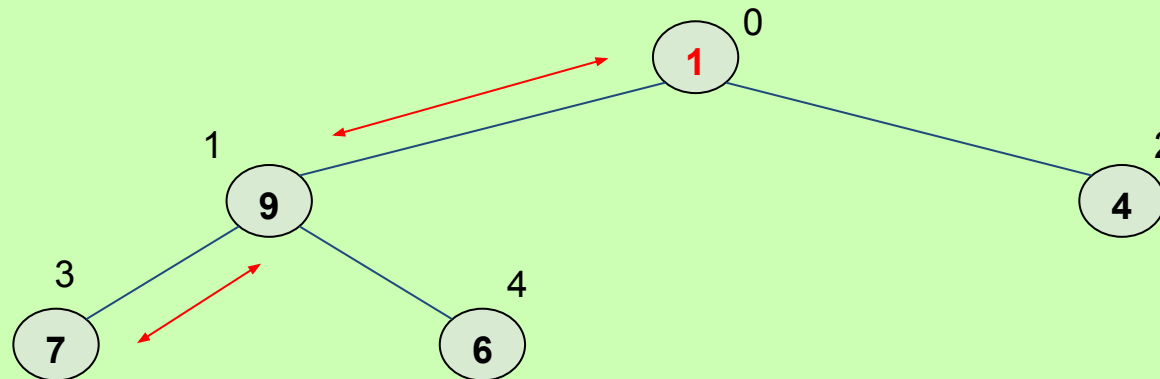


Ex4

Extraction du maximum (10)

- Extraction du max 10, échange avec le dernier (du tas actuel) et redescente de 1

1	9	4	7	6	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

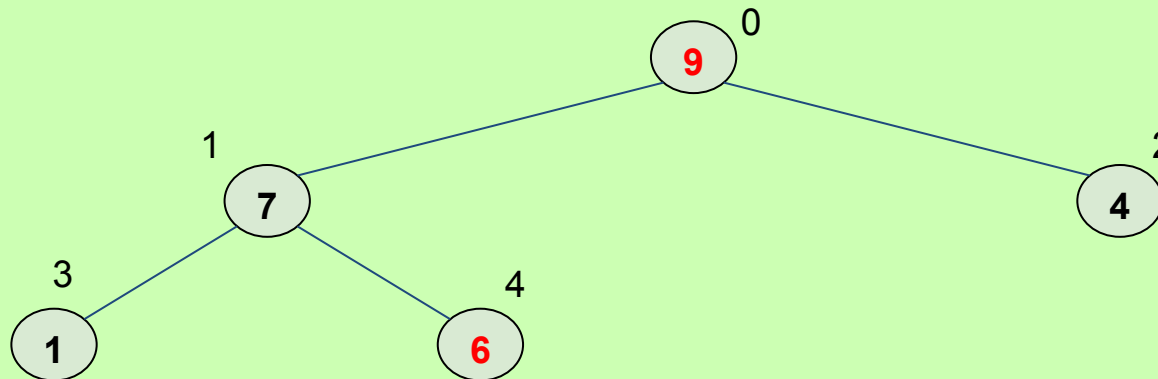


Ex4

Extraction du maximum (11)

- Extraction du max 9, échange avec le dernier (du tas actuel) et redescende de 6

9	7	4	1	6	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

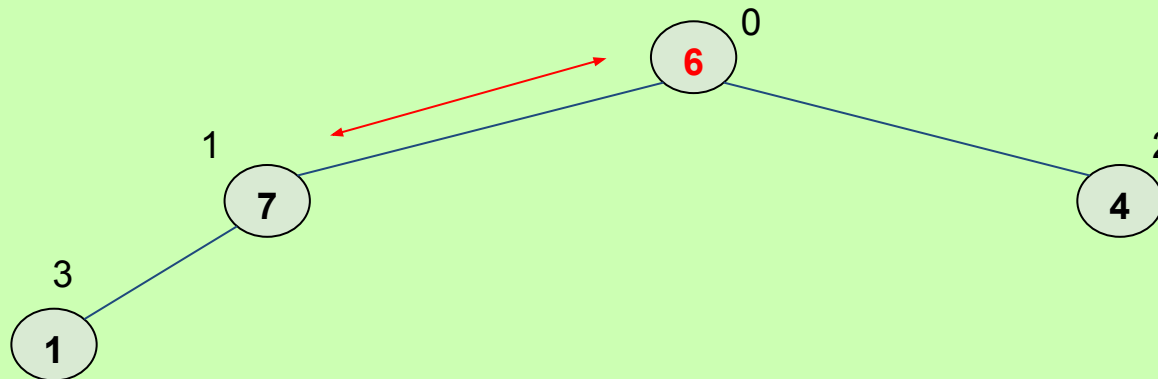


Ex4

Extraction du maximum (12)

- Extraction du max 9, échange avec le dernier (du tas actuel) et redescende de 6

6	7	4	1	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

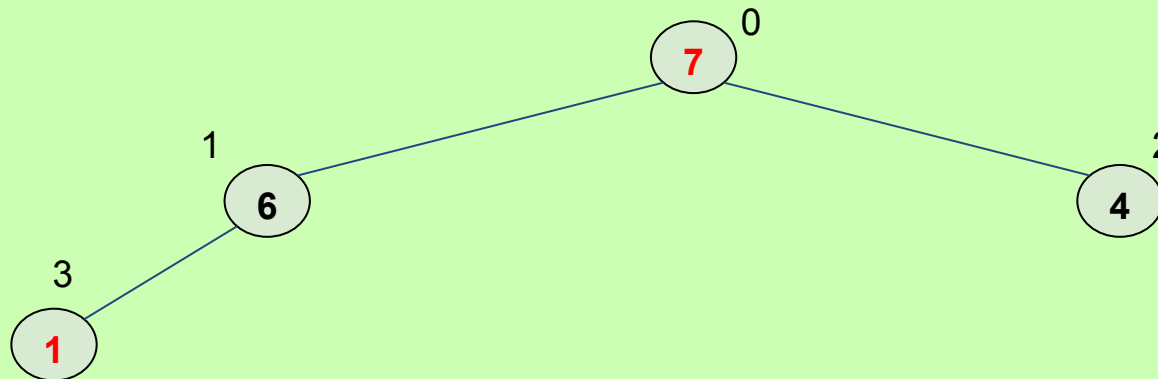


Ex4

Extraction du maximum (13)

- Extraction du max 7, échange avec le dernier (du tas actuel) et redescende de 1

7	6	4	1	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

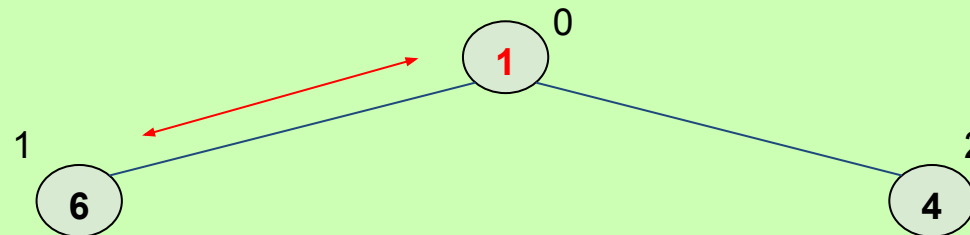


Ex4

Extraction du maximum (14)

- Extraction du max 7, échange avec le dernier (du tas actuel) et redescende de 1

1	6	4	7	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

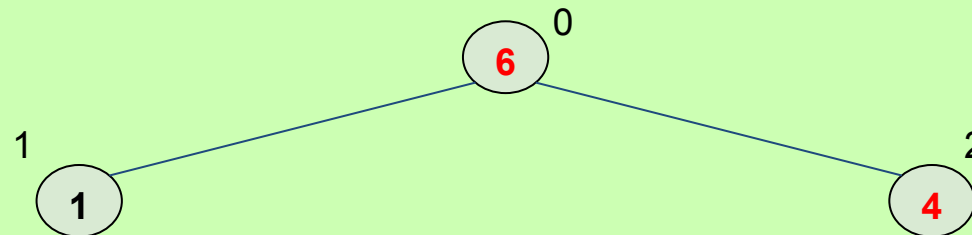


Ex4

Extraction du maximum (15)

- Extraction du max 6, échange avec le dernier (du tas actuel) et redescende de 4

6	1	4	7	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

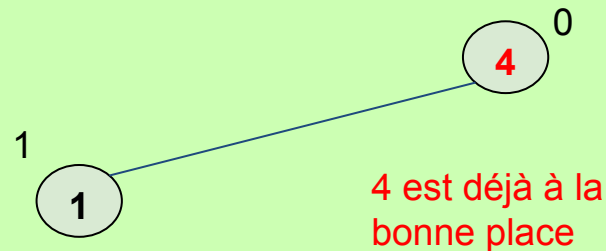


Ex4

Extraction du maximum (16)

- Extraction du max 6, échange avec le dernier (du tas actuel) et redescende de 4

4	1	6	7	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

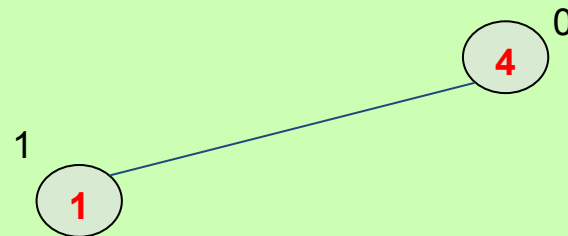


Ex4

Extraction du maximum (17)

- Extraction du max 4, échange avec le dernier (du tas actuel) et fin de la procédure

4	1	6	7	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9



Ex4

Extraction du maximum (18)

- Extraction du max 4, échange avec le dernier (du tas actuel) et fin de la procédure

1	4	6	7	9	10	11	15	15	18
0	1	2	3	4	5	6	7	8	9

Tri par tas

Algorithme

trierParTasV1(T, n) :

```
M = transformerEnMaximierV1(T, n)
Tant que n > 1
    Echanger nœud 0 et nœud n-1
    n ← n-1
    descendre(M, 0, n)
```

trierParTasV2(T, n) :

```
M = transformerEnMaximierV2(T, n)
Tant que n > 1
    extraireMax(M, n) // si extraireMax échange premier/dernier
    n ← n-1           // inutile si extraireMax modifie n
```



Exercices

```
void heapSortV1 (T_heap *p)
void heapSortV2 (T_heap *p)
T_data heapSortS3 (T_data d, int n)
```

Fil Rouge

- Ex3 : Implémenter de deux façons différentes
- Ex4 : Tester le tri de chaînes
- Comparer avec d'autres tris à l'aide des outils de la séance 3

Fil Rouge

Bilan

- Algorithme de tri efficace, présentant par rapport aux autres algorithmes de tri en $O(n \log n)$, certains avantages :
 - Algorithme de tri **en place**
 - Contrairement au tri par fusion qui a besoin d'un espace mémoire au moins égal à $n/2$ pour trier une table de n éléments
 - Algorithme en $O(n \log n)$ dans tous les cas
 - Contrairement au tri rapide pour lequel il peut y avoir dégénérescence en $O(n^2)$
- Mais :
 - Algorithme qui n'est **pas stable**
 - Algorithme deux fois moins rapide que le quicksort en moyenne

Heapsort : Complexité

```
trierParTas(T, n) :
```

```
    M = transformerEnMaximierV2(T, n) //  $C_1$ 
```

```
    Tant que n > 1 //  $C_2$ 
```

```
        Echanger nœud 0 et nœud n-1
```

```
        n ← n-1
```

```
    descendre(M, 0, n)
```

- Coût $C(n) = C_1(n) + C_2(n)$
- $C_1(n) = O(n)$
- $C_2(n) = \sum_{k=2 \rightarrow n} 1 + \lfloor \log_2(k) \rfloor \leq n-1 + \underbrace{\sum_{k=2 \rightarrow n} \log_2(k)}_{\log_2(n!)}$
- Formule de Stirling : $n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \theta\left(\frac{1}{n}\right)\right)$
- $C_2(n) = O(n \log(n))$
- $\Rightarrow C(n) = O(n \log(n))$

https://haltode.fr/algo/tri/tri_rapide.html#complexite

*“Il faut savoir que le **tri rapide peut s'exécuter deux fois plus vite que le tri par tas pour des raisons de mémoire cache**. Les deux algorithmes ont la même complexité en moyenne, mais le tri par tas compare en général des éléments du tableau qui sont assez éloignés contrairement au tri rapide. Or, quand vous accédez à un tableau, votre ordinateur place une certaine partie de ce tableau (ou la totalité) dans une mémoire cache pour que l'accès à ce dernier se fasse plus rapidement. Dans le cas de très grandes entrées, le tri par tas va obliger la mémoire à charger et décharger successivement des parties du tableau (trop grand pour être entièrement stocké dans la mémoire cache), ce qui ralentira l'exécution du programme.”*



Codage de Huffman

Présentation

- David Huffman, 1952
- Méthode de compression de document **sans perte** basée sur un codage de **longueur variable**
 - L'idée est d'associer aux caractères du document à compresser, une **séquence de bits d'autant plus courte que leur fréquence est élevée**
 - Principe similaire au code Morse (1838, Samuel Morse et Alfred Vail)
- Technique largement utilisée et très efficace pour la compression des données
 - Des économies de 20% à 90% sont courantes, selon les caractéristiques des données à compresser

Principe

- Calcul du **nombre d'occurrences** de chaque caractère du document à traiter
- Construction de l'**arbre de codage de Huffman** en partant des feuilles qui, associées aux caractères, portent comme information leur nombre d'occurrences
 - On associe les deux nœuds **de plus faible nombre d'occurrences** pour créer un nouveau **nœud père**, dont le nombre d'occurrences est la somme des valeurs de ses fils
 - On réitère ce processus avec les feuilles et les nœuds internes restants jusqu'à ne plus en avoir qu'un seul, la racine
- L'arbre de codage étant créé, le codage d'un caractère est déterminé par le **chemin depuis la racine**, permettant d'atteindre la feuille qui lui est associée
 - Bit 0 pour la branche de gauche, bit 1 pour celle de droite (par exemple)
 - Les caractères les **moins fréquents** sont plus bas dans l'arbre, et utilisent **davantage de bits** pour être codés

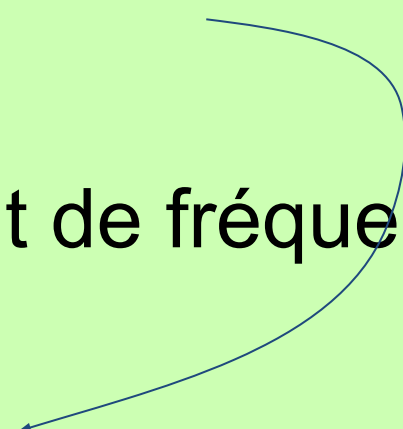
Exemple

- Encodage de la chaîne “ABBACADABRA”
- Nombre d'occurrences de chaque caractère :

A	B	C	D	R
5	3	1	1	1

- Tri des caractères par ordre croissant de fréquence :

C	D	R	B	A
1	1	1	3	5

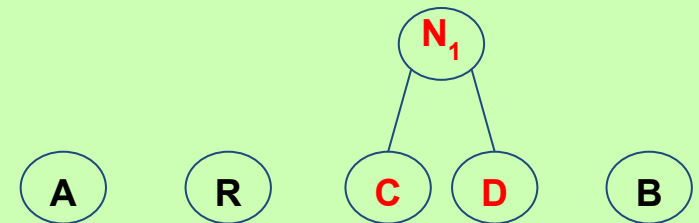


Construction de l'arbre de codage (1)

- On associe les deux nœuds de plus faible nombre d'occurrences pour créer un nouveau nœud père, dont le nombre d'occurrences est la somme des valeurs de ses fils
- Extraction de C (1) => fils gauche
- Extraction de D (1) => fils droit
- Création du père N_1 (1+1)

C	D	R	B	A
1	1	1	3	5

R	N_1	B	A
1	2	3	5

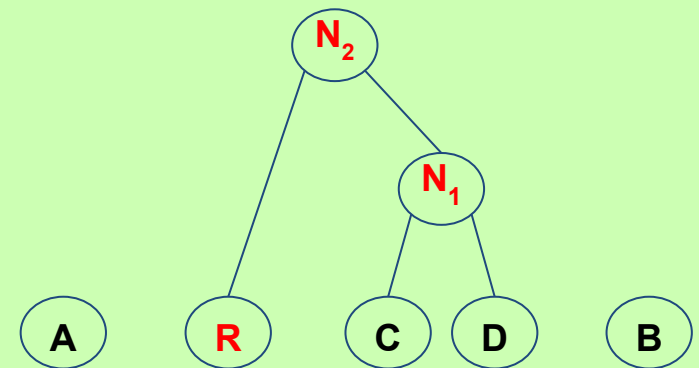
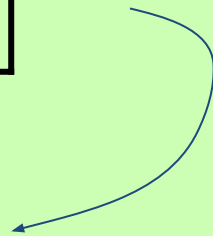


Construction de l'arbre de codage (2)

- On associe les deux nœuds de plus faible nombre d'occurrences pour créer un nouveau nœud père, dont le nombre d'occurrences est la somme des valeurs de ses fils
- Extraction de R (1) \Rightarrow fils gauche
- Extraction de N_1 (2) \Rightarrow fils droit
- Création du père N_2 (1+2)

R	N_1	B	A
1	2	3	5

N_2	B	A
3	3	5

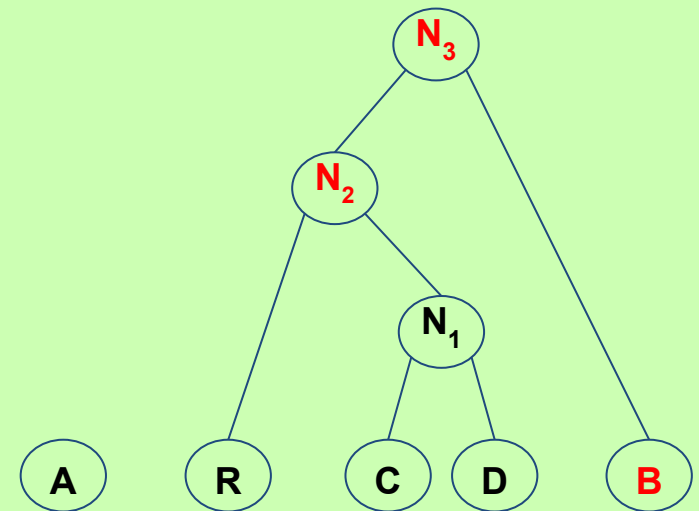
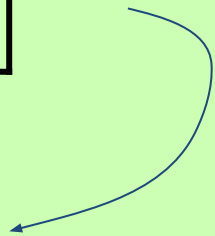


Construction de l'arbre de codage (3)

- On associe les deux nœuds de plus faible nombre d'occurrences pour créer un nouveau nœud père, dont le nombre d'occurrences est la somme des valeurs de ses fils
- Extraction de N_2 (3) \Rightarrow fils gauche
- Extraction de B (3) \Rightarrow fils droit
- Création du père N_3 (3+3)

N_2	B	A
3	3	5

A	N_3
5	6



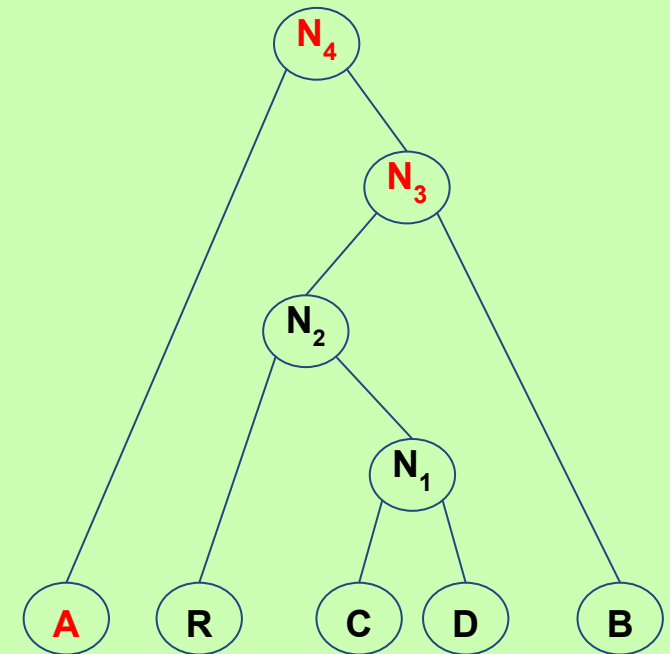
Construction de l'arbre de codage (4)

- On associe les deux nœuds de plus faible nombre d'occurrences pour créer un nouveau nœud père, dont le nombre d'occurrences est la somme des valeurs de ses fils
- Extraction de A (5) \Rightarrow fils gauche
- Extraction de N_3 (6) \Rightarrow fils droit
- Création du père N_4 (5+6)

A	N_3
5	6

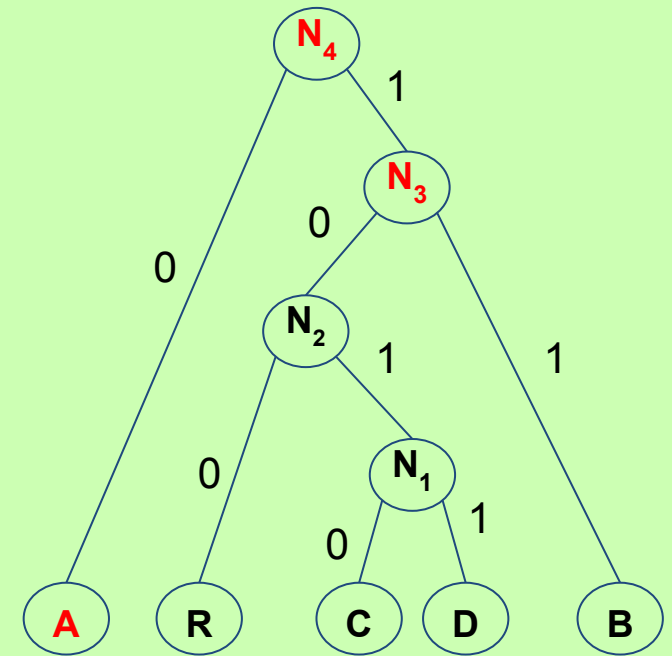
N_4
11

Processus de construction terminé



Codage des caractères (1)

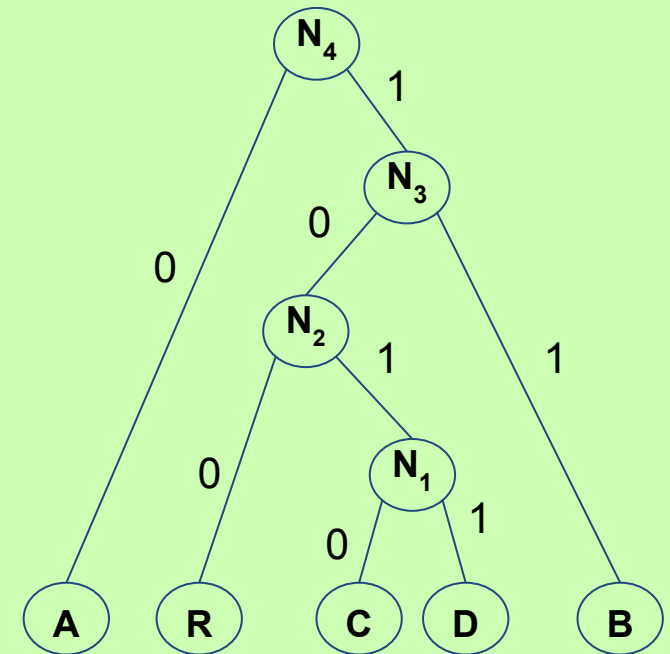
- L'arbre de codage étant créé, le codage d'un caractère est déterminé par le chemin depuis la racine, permettant d'atteindre la feuille qui lui est associée
 - Bit 0 pour la branche de gauche
 - Bit 1 pour celle de droite



Codage des caractères (2)

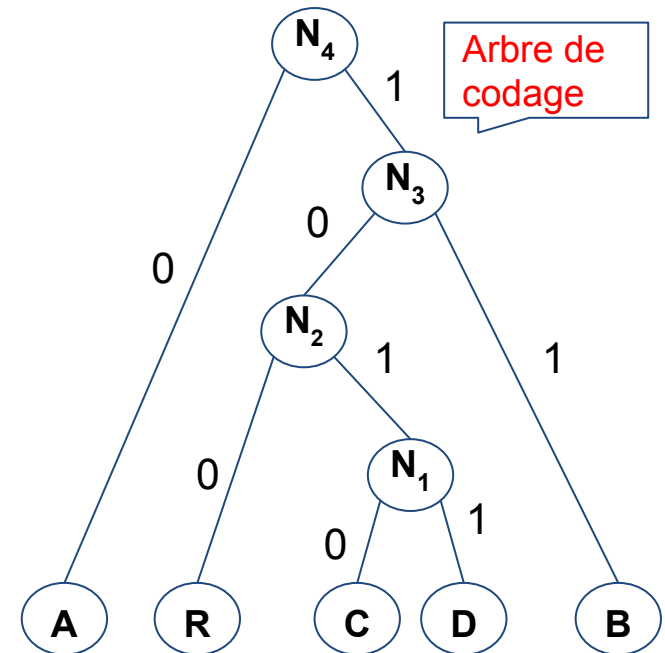
- L'arbre de codage étant créé, le codage d'un caractère est déterminé par le chemin depuis la racine, permettant d'atteindre la feuille qui lui est associée
 - Bit 0 pour la branche de gauche
 - Bit 1 pour celle de droite

Lettre	A	B	C	D	R
Occurrences	5	3	1	1	1
Code	0	11	1010	1011	100
Longueur du code	1	2	4	4	3



Propriétés du code de Huffman

- Par construction, les caractères les plus fréquents sont plus hauts, donc le nombre de symboles pour les coder est moindre
- L'arbre est **localement complet (strict)** : chaque nœud a deux enfants (nœuds internes) ou 0 (feuilles)
- **Code préfixe** (code instantané) :
 - Codage sans préfixes : le code d'un caractère n'est jamais le préfixe du code d'un autre : on ne peut les confondre
 - Contrairement au code Morse qui nécessite des caractères d'espacement entre les lettres (silences)



Lettre	A	B	C	D	R
Occurrences	5	3	1	1	1
Code	0	11	1010	1011	100
Longueur du code	1	2	4	4	3

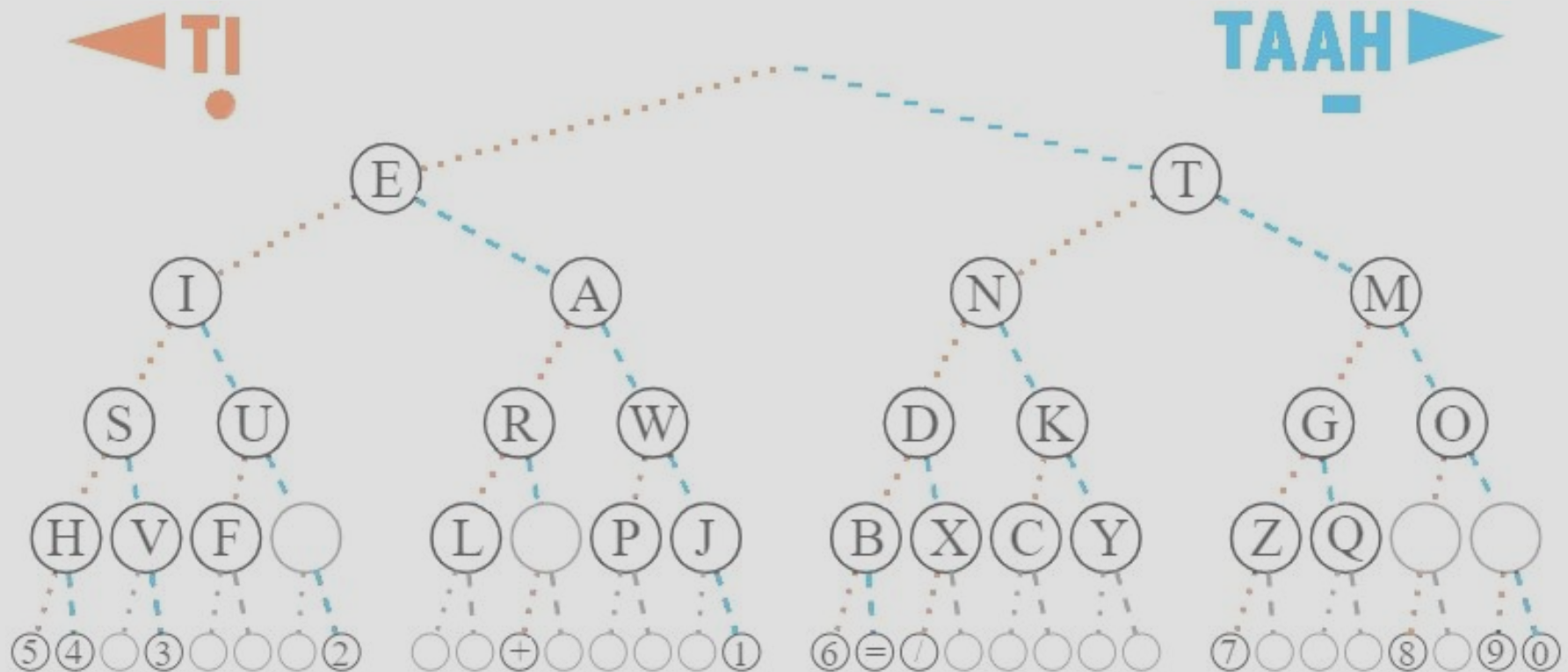
Table de codage

Propriétés du code de Huffman

- **Code préfixe optimal** : compression maximale permise par encodage à base de symboles et une distribution de probabilité d'apparition des caractères dans le document à encoder connue
 - <https://polaris.imag.fr/jean-marc.vincent/index.html/ALGO5/Codage-Shannon-Huffman.pdf>
 - **Algorithme glouton** : des choix localement optimaux produisent une solution globalement optimale
- **Codage non canonique** : il existe plusieurs codes de Huffman de qualité équivalente pour un même document
 - Pour le rendre canonique, on peut ordonner les symboles de même fréquence dans l'**ordre lexical**

Code Morse

- Code de longueur variable **non préfixe**



Implémentation

- Nombres d'étapes de l'algorithme : $n-1$
 - A chaque étape, on extrait deux nœuds, on en recrée un dans le tableau des occurrences
 - Si n caractères à encoder, il faut $n-1$ étapes pour qu'il ne reste plus que la racine : $n - \text{nbEtapes} = 1 \Leftrightarrow \text{nbEtapes} = n-1$
- Nombre de nœuds internes à créer : $n-1$
 - Un nœud interne est créé à chaque étape
- Comment extraire les caractères les moins fréquents de manière efficace ? Utilisation d'un minimier !

Implémentation : Minimier indirect

- Utilisation d'un **minimier indirect** pour extraire les caractères les moins fréquents de manière efficace
 - On ne connaît les caractères que par leur code ascii
 - Mais on ne les trie pas par codes ascii, on les trie par fréquence
 - ⇒ Le minimier indirect **trie les codes ascii** en fonction de leurs occurrences qui sont enregistrées dans un **tableau additionnel**
- Cf. codes ASCII 7 bits : 128 caractères à encoder
 - ⇒ prévoir assez d'espace dans le tableau additionnel pour stocker le nombre d'occurrences de 127 nœuds internes supplémentaires

Minimier indirect pour le codage de Huffman

- Implémentation statique
- **nbElt** : nb effectif de nœuds dans le tas
- **tree** : table décrivant l'organisation des nœuds dans le tas
- **data** : nombres d'occurrences des nœuds du tas

```
#define MAXCARS 128

typedef struct {
    unsigned int nbElt;
    unsigned char
        tree[MAXCARS];
    int data[2*MAXCARS-1];
} T_indirectHeap;
```

Implémentation :

Arbre de codage de Huffman

- Représentation de la structure de l'arbre sous forme d'un tableau de 255 cases :
 - 128 premières cases : caractères à coder
 - 127 cases suivantes : nœuds internes
- Stockage uniquement des **indices des noeuds pères** de chaque nœud
 - -indicePere pour le fils gauche
 - +indicePere pour le fils droit

```
#define MAXCARS 128
```

```
int huffmanTree [2*MAXCARS-1];
```

Exemple d'implémentation

Initialisation des structures

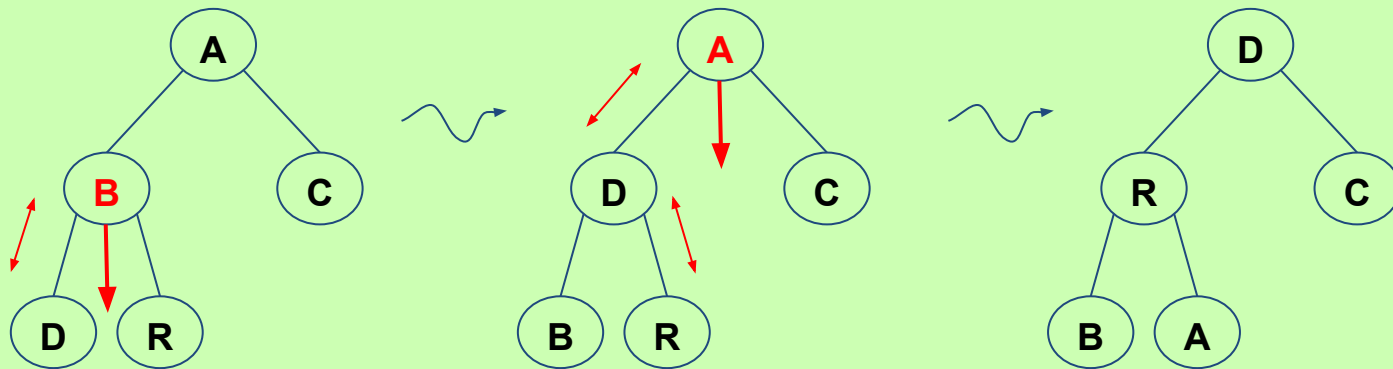
- Encodage de la chaîne “ABBACADABRA”

tree					nbElements : 5										
65 (A)	66 (B)	67 (C)	68 (D)	82 (R)											
data															
0	0	...	5	3	1	1	...	1	...	0	0	0	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255
huffmanTree															
-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

Exemple d'implémentation

Création du minimier indirect

- Transformer en minimier V2 : faire redescendre les nœuds internes...



tree

nbElements : 5

68 (D)	82 (R)	67 (C)	66 (B)	65 (A)
--------	--------	--------	--------	--------

data

0	0	...	5	3	1	1	...	1	...	0	0	0	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

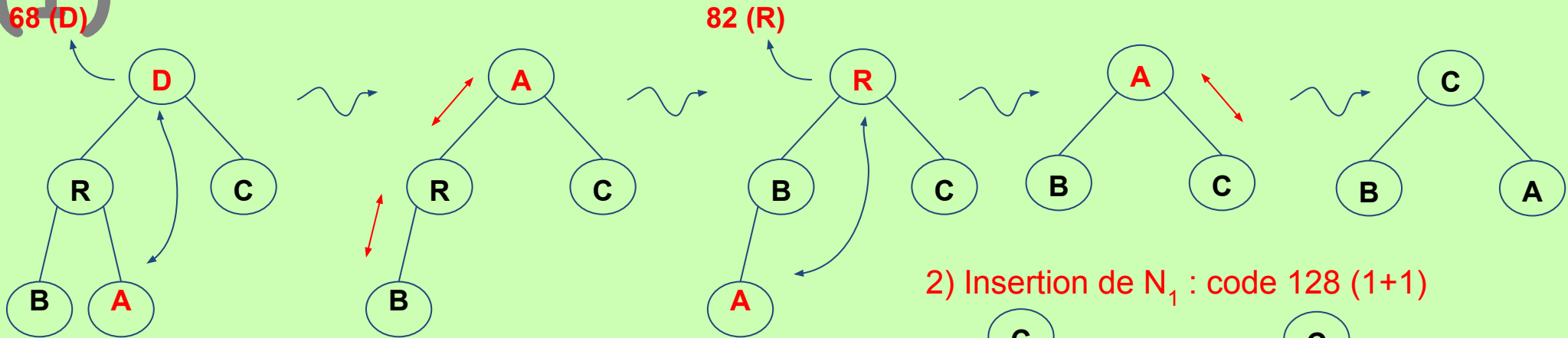
huffmanTree

-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

Exemple d'implémentation

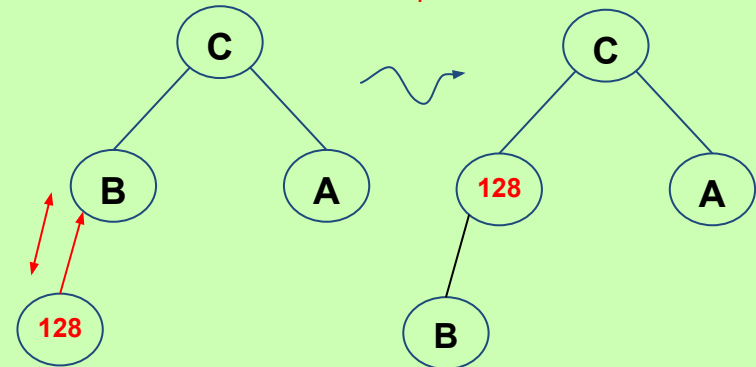
Construction de l'arbre de codage

(1)

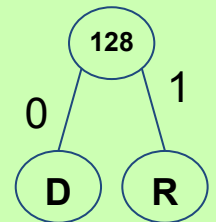


1) Extraction de D(1) puis R(1)

2) Insertion de N_1 : code 128 (1+1)



3) Ajout de 128 dans l'arbre de codage



tree				nbElements : 4			
67 (C)	128	65 (A)	66 (B)				

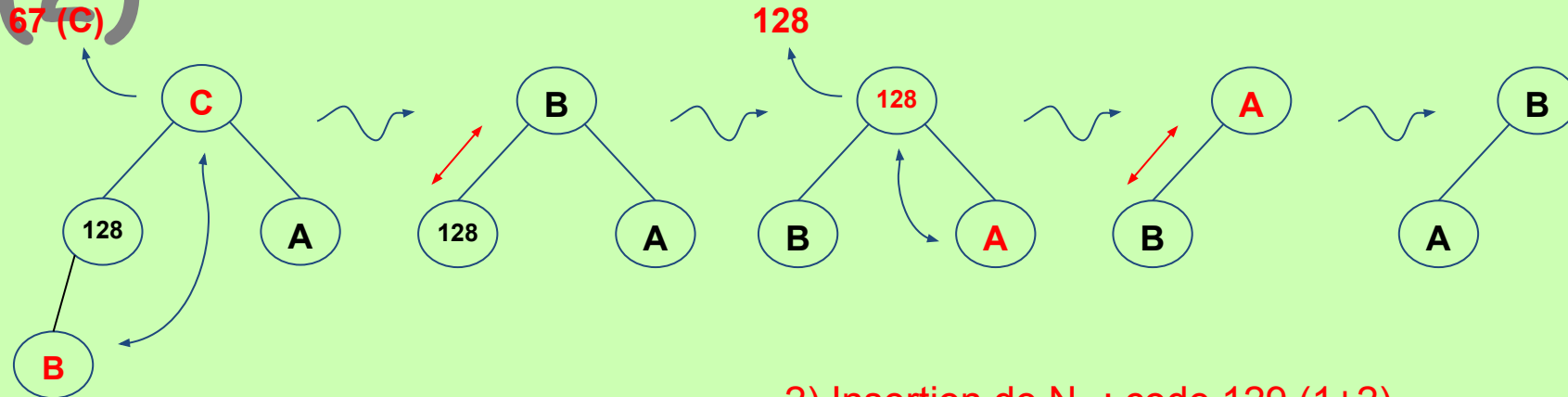
data															
0	0	...	5	3	1	1	...	1	...	2	0	0	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

huffmanTree															
-256	-256	-256	-256	-256	-256	-128	-256	+128	-256	-256	-256	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

Exemple d'implémentation

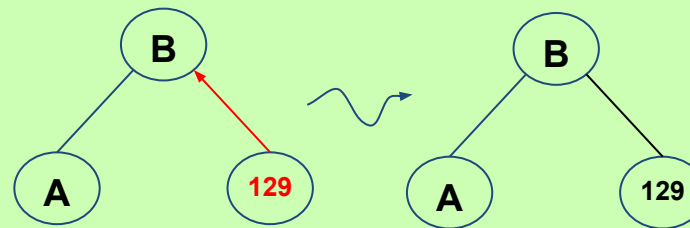
Construction de l'arbre de codage

(?)

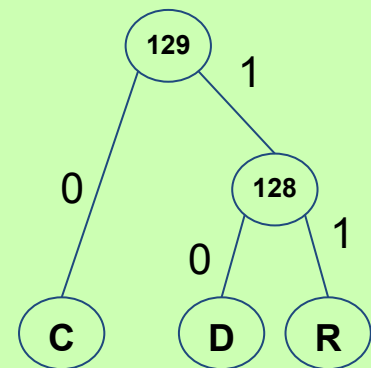


1) Extraction de C(1) puis 128(2)

2) Insertion de N_2 : code 129 (1+2)



3) Ajout de 129 dans l'arbre de codage



tree			nbElements : 3		
66 (B)	65 (A)	129			

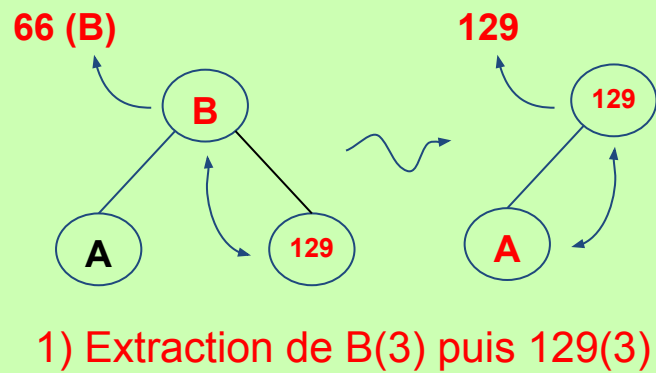
data															
0	0	...	5	3	1	1	...	1	...	2	3	0	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

huffmanTree															
-256	-256	-256	-256	-256	-129	-128	-256	+128	-256	+129	-256	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

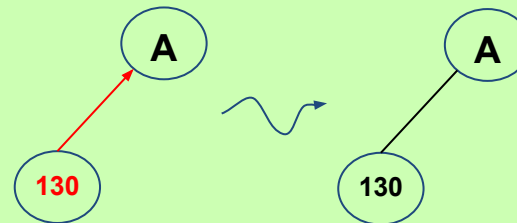
Exemple d'implémentation

Construction de l'arbre de codage

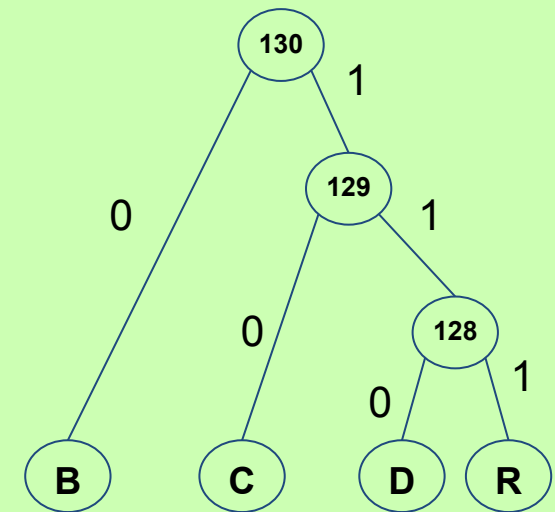
(3)



2) Insertion de de N_3 : code 130 (3+3)



3) Ajout de 130 dans l'arbre de codage



tree nbElts : 2

65 (A)	130
--------	-----

data

0	0	...	5	3	1	1	...	1	...	2	3	6	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

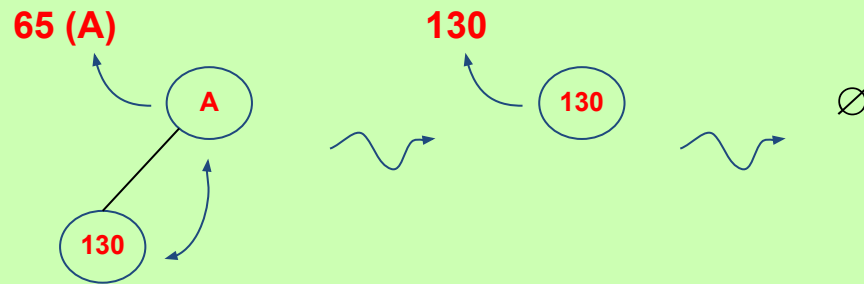
huffmanTree

-256	-256	-256	-256	-130	-129	-128	-256	+128	-256	+129	+130	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

Exemple d'implémentation

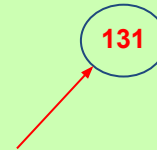
Construction de l'arbre de codage

(4)

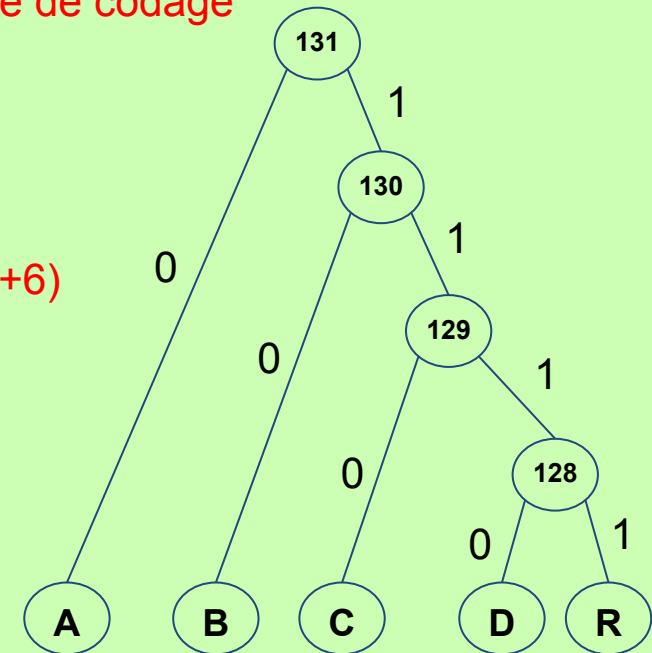


1) Extraction de A(5) puis 130(6)

2) insertion de N4 : code 131 (5+6)



3) Ajout de 131 dans l'arbre de codage



tree nbElts : 1

131

data

0	0	...	5	3	1	1	...	1	...	2	3	6	11	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

huffmanTree

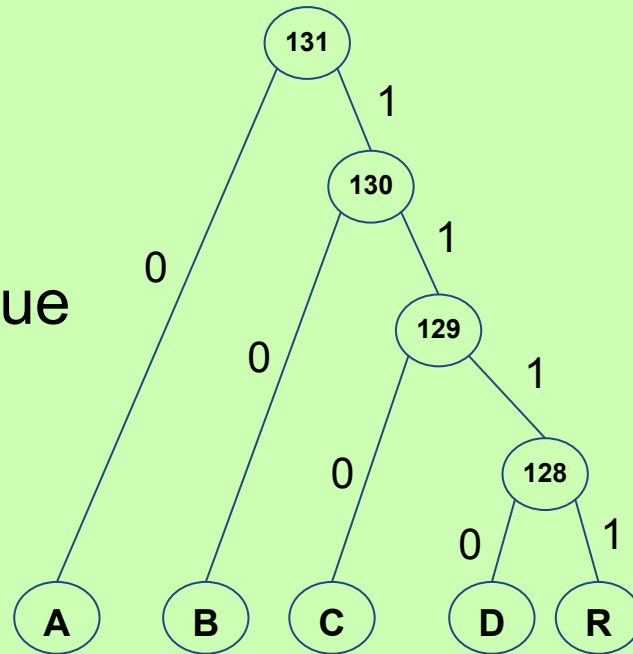
-256	-256	-256	-131	-130	-129	-128	-256	+128	-256	+129	+130	+131	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

Exemple d'implémentation

Bilan

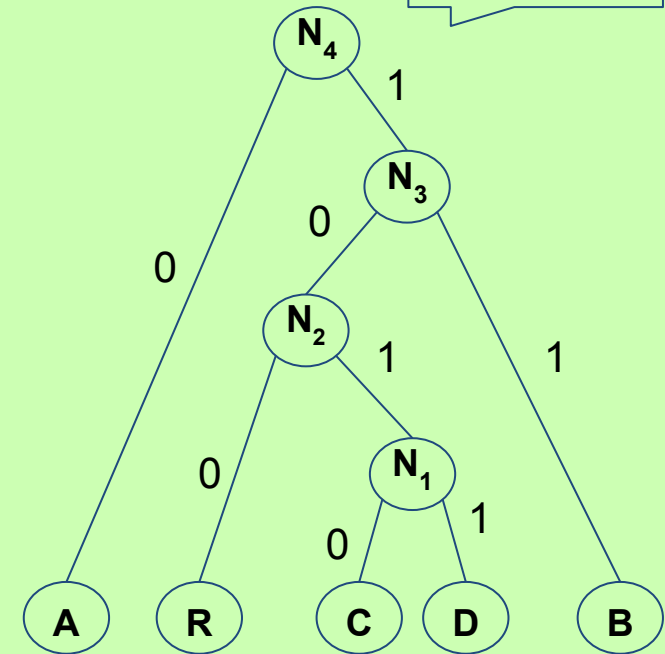
Arbres de codage

- Codage préfixe
 - Optimal
 - Non canonique



Lettre	A	B	C	D	R
Occurrences	5	3	1	1	1
Code	0	10	110	1110	1111
Longueur du code	1	2	3	4	4

Tables de codage



Lettre	A	B	C	D	R
Occurrences	5	3	1	1	1
Code	0	11	1010	1011	100
Longueur du code	1	2	4	4	3

Codage de Huffman

Algorithme

Mi : minimier indirect
Ht : arbre de codage de Huffman
 C_1, C_2 : caractères extraits
 N_i : noeud inséré

```
huffman(D)
```

```
  Mi = analyserDocument(D) // Comptage des occurrences
  Ht = initHuffmanTree() // Initialisation de l'arbre de codage
  transformerEnMinimierV2(Mi) // Réorganisation du minimier
  Pour i = 1 jusqu'à n-1
     $C_1$  = extraireMin(Mi) // Extraire et réorganiser
     $C_2$  = extraireMin(Mi) // Extraire et réorganiser
    AjouterNoeud(Ht,  $N_i$ ) // Ajout dans l'arbre de codage
    insérerMI(Mi,  $N_i$ , VAL( $C_1$ ) + VAL( $C_2$ )) // Insérer et réorganiser
```

Codage de Huffman

Complexité

Mi : minimier indirect
Ht : arbre de codage de Huffman
 C_1, C_2 : caractères extraits
 N_i : noeud inséré

huffman(D)

Mi = analyserDocument(D) // Comptage des occurrences	$O(n)$
Ht = initHuffmanTree() // Initialisation de l'arbre de codage	$O(n)$
transformerEnMinimierV2(Mi) // Réorganisation du minimier	$O(n \log(n))$
Pour i = 1 jusqu'à n-1	
C_1 = extraireMin(Mi) // Extraire et réorganiser	$O(\log(n-i))$
C_2 = extraireMin(Mi) // Extraire et réorganiser	$O(\log(n-i-1))$
AjouterNoeud(Ht, N_i) // Ajout dans l'arbre de codage	$O(1)$
insérerMI(Mi, N_i , VAL(C_1) + VAL(C_2)) // Insérer et réorganiser	$O(\log(n-i))$

↑
n-1 fois
↓

⇒ $C(n) = O(n \log(n))$

Entête de Huffman

Méthode naïve

- Il faut transmettre l'**entête de Huffman** avec le fichier codé pour que le fichier puisse être décodé !
- Méthode naïve : On insère au début du fichier un tableau contenant les **relations entre code ASCII et code de Huffman**
 - Ce n'est pas très efficace car cela prend de la place et qu'il ne sera pas facile de procéder au décodage en utilisant cette structure de données

Entête de Huffman

Première amélioration

- Première amélioration : on transmet une version simplifiée du document ayant servi au codage, en indiquant pour chaque caractère du fichier initial sa fréquence
 - Ce qui revient à **transmettre le minimier initial**
- Pour décoder, il suffira d'exécuter une nouvelle passe de l'algorithme de Huffman sur cet entête pour obtenir l'arbre de codage de Huffman

Entête de Huffman

Seconde amélioration

- On transmet directement l'arbre de codage de Huffman
- Pour cela, on parcourt l'arbre en profondeur (parcours préfixe) et on affiche à chaque nœud rencontré un 1 s'il s'agit d'une feuille, et 0 sinon
 - Cela permet de reconstituer de manière unique la « forme » de l'arbre
 - Cela nécessite d'enregistrer les relations descendantes (filsG, filsD) dans l'implémentation de l'arbre de codage...
- Il suffit ensuite de lister dans l'entête les caractères associés aux feuilles de l'arbre pour recréer l'arbre de codage de Huffman
- Cette méthode ne fonctionne bien entendu que pour les arbres binaires localement complets dans lesquels chaque nœud qui n'est pas une feuille a deux enfants

Entête de Huffman

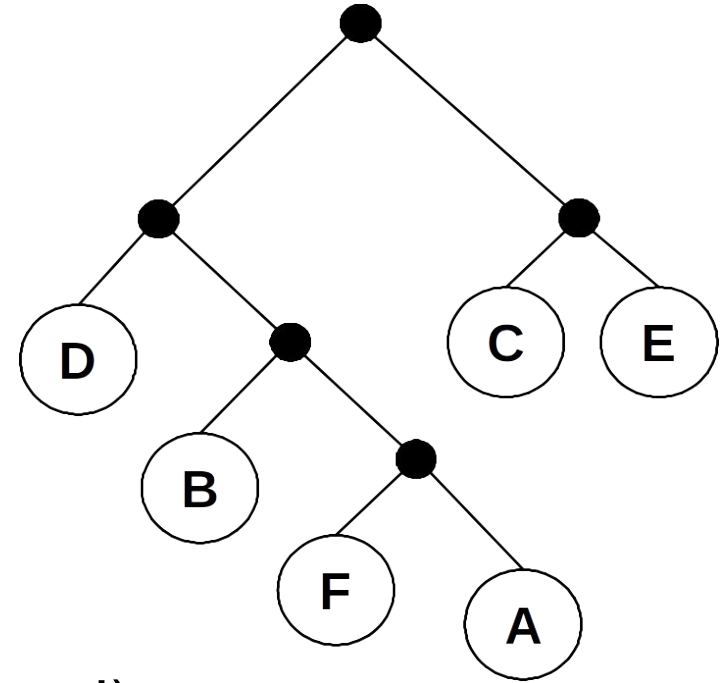
Seconde amélioration - Exemple

- L'entête de Huffman sera codé :

00101011011

DBFACE

- Explications :
 - 0 (racine est un noeud)
 - 0 (le fils gauche de la racine est un noeud)
 - 1 (le fils gauche du fils gauche de la racine est une feuille : **D**)
- Le fait d'arriver sur une feuille stoppe le parcours en profondeur, et avance d'un cran en largeur



Fréquence des caractères de la langue française

Lettre	Fréquence	Lettre	Fréquence
A	8.40 %	N	7.13 %
B	1.06 %	O	5.26 %
C	3.03 %	P	3.01 %
D	4.18 %	Q	0.99 %
E	17.26 %	R	6.55 %
F	1.12 %	S	8.08 %
G	1.27 %	T	7.07 %
H	0.92 %	U	5.74 %
I	7.34 %	V	1.32 %
J	0.31 %	W	0.04 %
K	0.05 %	X	0.45 %
L	6.01 %	Y	0.30 %
M	2.96 %	Z	0.12 %



Projet Fil Rouge 2022

**Cahier des charges
Organisation
Evaluation**

Cahier des charges

Programme 1 : tri par tas

- Permettre de sélectionner la nature du tas (minimier ou maximier) à l'aide d'une constante symbolique
- Développer un programme permettant d'adapter votre tri par tas à la structure de test de la séance 3 (complexité)
- Comparer expérimentalement les vitesses des tris de votre TEA3 avec celle du tri par tas

Cahier des charges

Programme 2 : codage de Huffman

- Produire un programme prenant sur son entrée standard les caractères d'un texte à compresser
- Le programme devra réaliser un codage de huffman et afficher la table de codage générée ainsi que le texte compressé
 - L'arbre partiellement ordonné et l'arbre de codage devront être générés dans un fichier sous forme graphique au fur et à mesure de l'exécution du programme
 - le code devra être canonique en respectant l'ordre lexicographique des caractères lors de leur extraction du minimier indirect

Cahier des charges

Programme 3

- Produire un programme prenant un ou deux paramètres :
- Lorsque le programme prend deux paramètres, il doit compresser un texte et produire un fichier contenant le texte compressé précédé de l'entête de huffman permettant de le décompresser
 - paramètre 1 : chemin du fichier à compresser
 - paramètre 2 : chemin du fichier à produire
- Lorsque le programme prend un seul paramètre, il s'agit du chemin d'un fichier à décompresser

Ressources

- Cahier des charges détaillé
- https://docs.google.com/document/d/1p1PfOS_ubiSPvHGVTYc3Qu23RgZKFcVjHy64hnpj6GF8/edit?usp=sharing

Cadrage séance 5 :

- Pas de test en séance
- Retours et conseils individuels sur le travail des groupes
- Séance de développement avec possibilité de demander des conseils à l'intervenant présent

Organisation / Evaluation

- Équipes de **4 étudiants maximum** du **même groupe de TD**
 - Objectif : au maximum 8 groupes d'étudiants par groupe TD
- Remise du travail :
 - Rendre code + CR **au plus tard 24h avant la dernière séance**
 - Attention aux critères de qualité (livraison, CR, gestion de projet...) énoncés précédemment ! (capsule 6)
- Evaluation :
 - Qualité du code, du CR, de la livraison, de la gestion de projet
 - Comparaison de l'efficacité des programmes pour des arbres de taille et complexité croissantes
 - Un classement sera établi sur toute la promo
 - La note finale du fil rouge dépendra en partie de ce classement

Dernière séance

- Soutenances de 30 minutes/équipe
 - 20 minutes de présentation, 10 minutes de questions
- En groupe TD
 - Pas plus de 8 équipes par groupe TD

Code Couleur

Légende des textes

- mot-clé important, variable, contenu d'un fichier, code source d'un programme
- chemin ou url, nom d'un paquet logiciel
- commande, raccourci
- commentaire, exercice, citation
- culturel, optionnel

Culturel / Approfondissement

- A ne pas connaître intégralement par coeur
 - Donc, le reste... est à maîtriser parfaitement !
- Pour anticiper les problématiques que vous rencontrerez en stage ou dans d'autres cours
- Pour avoir de la conversation à table ou en soirée...

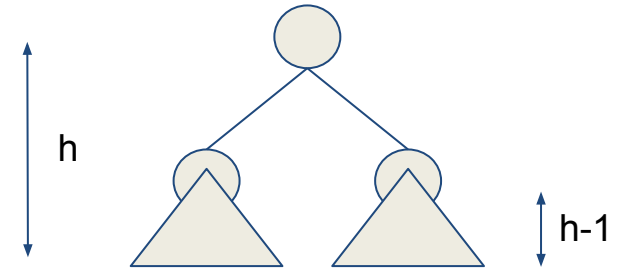
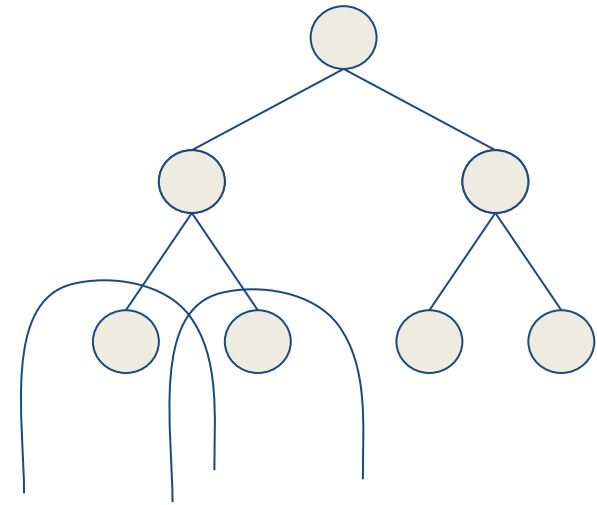
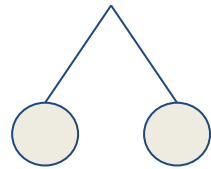
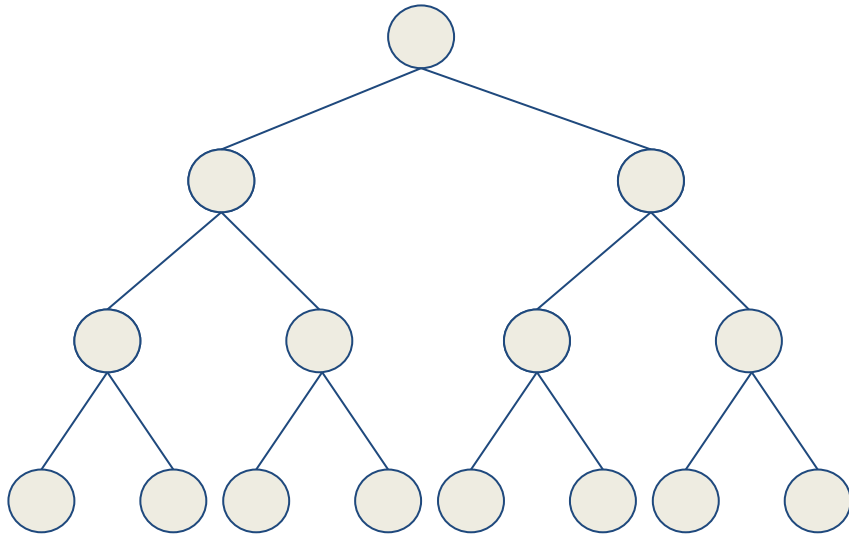
Exemples ou Exercices

- Brancher le cerveau
- Participer
- Expérimenter en prenant le temps...

Bonnes pratiques, prérequis

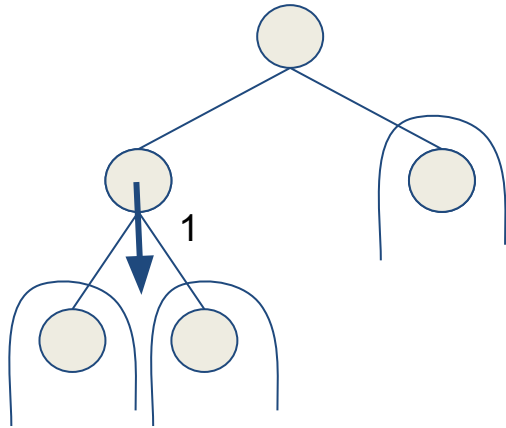
- Des éléments d'organisation indispensables pour un travail de qualité
- Des rappels de concepts déjà connus

Annexes

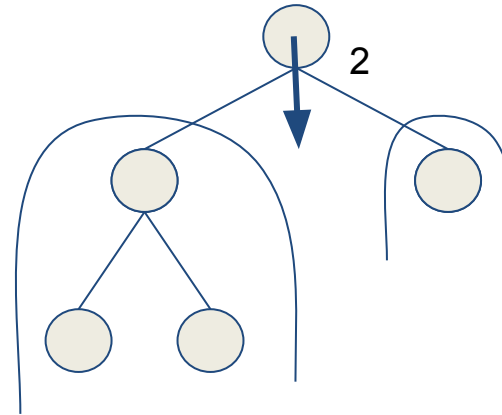


Illustration

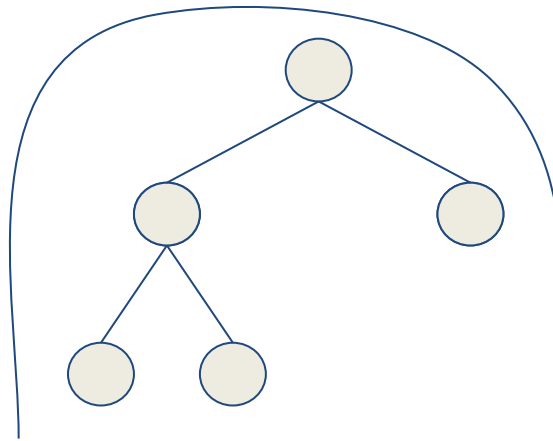
Transformer en maximier v2



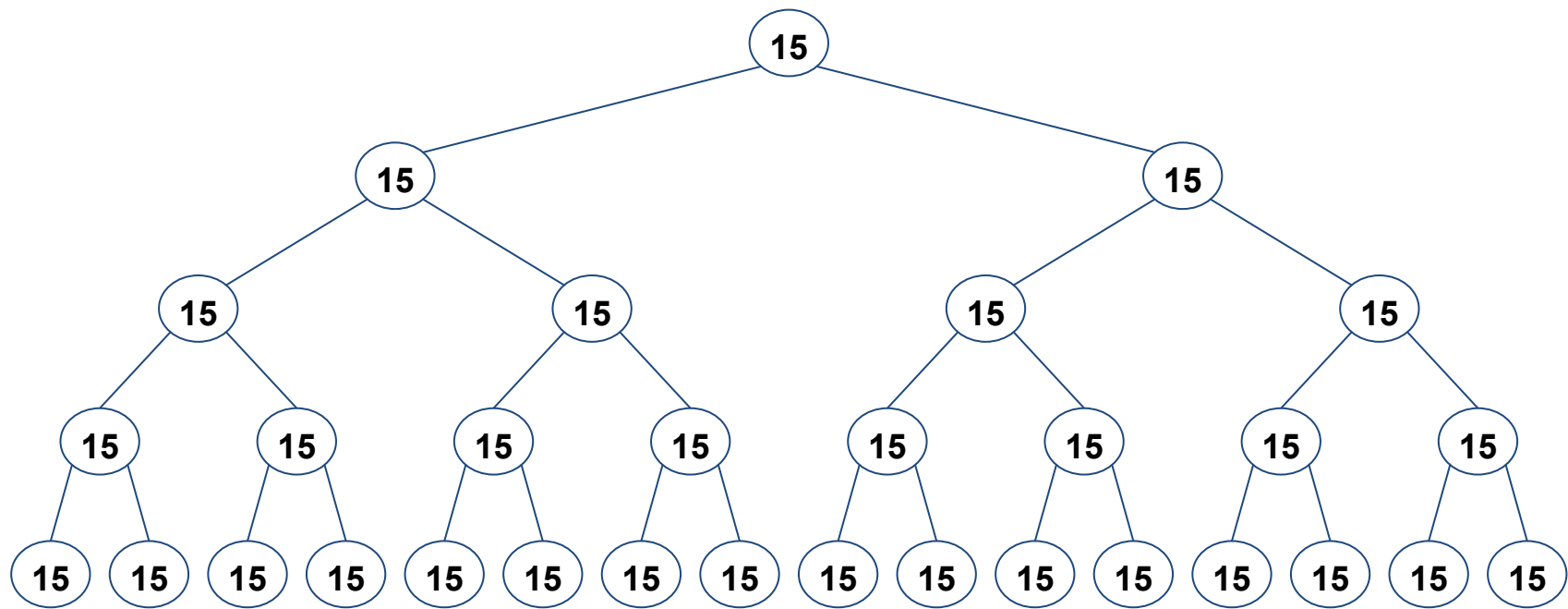
Les feuilles sont déjà des maximiers
On fait descendre le dernier nœud interne



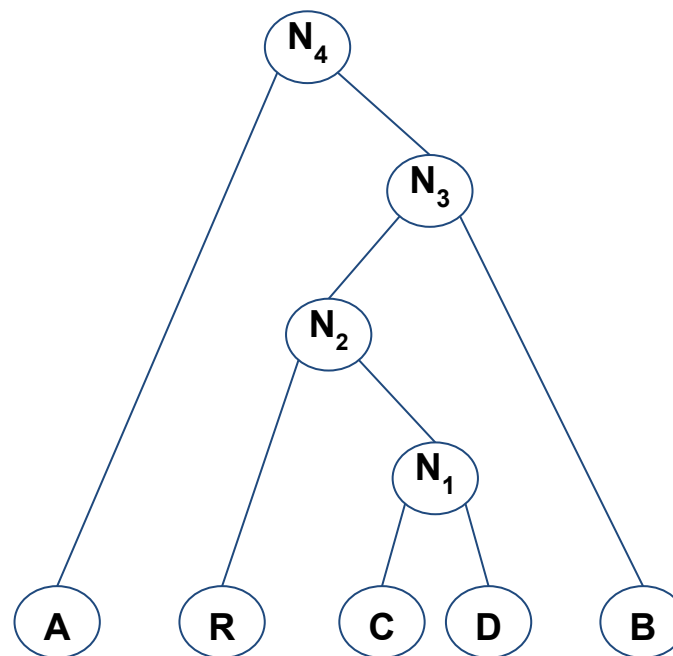
Le sous-arbre correspondant est devenu un maximier
On fait descendre le prochain nœud interne

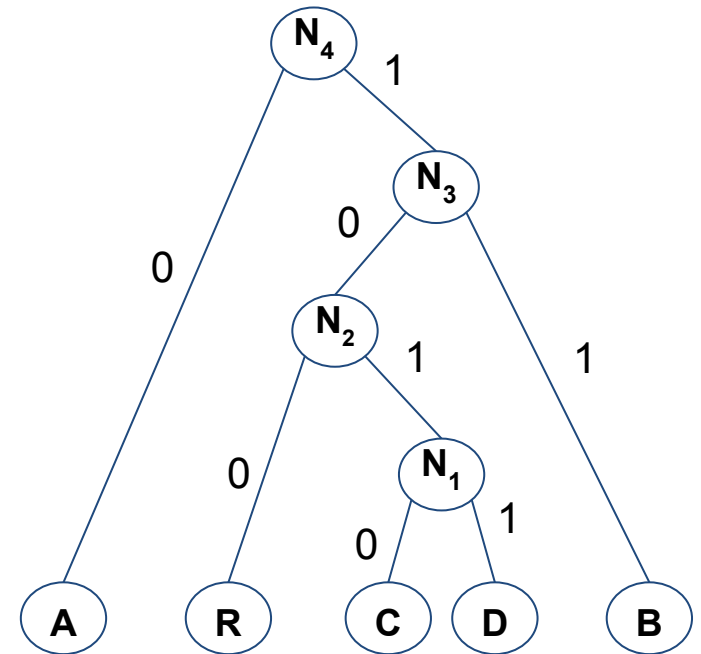
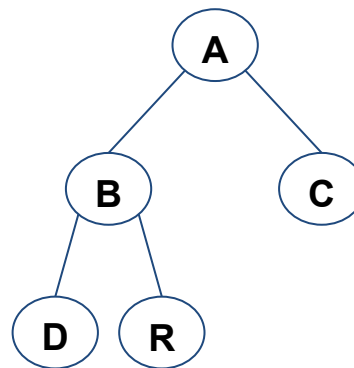
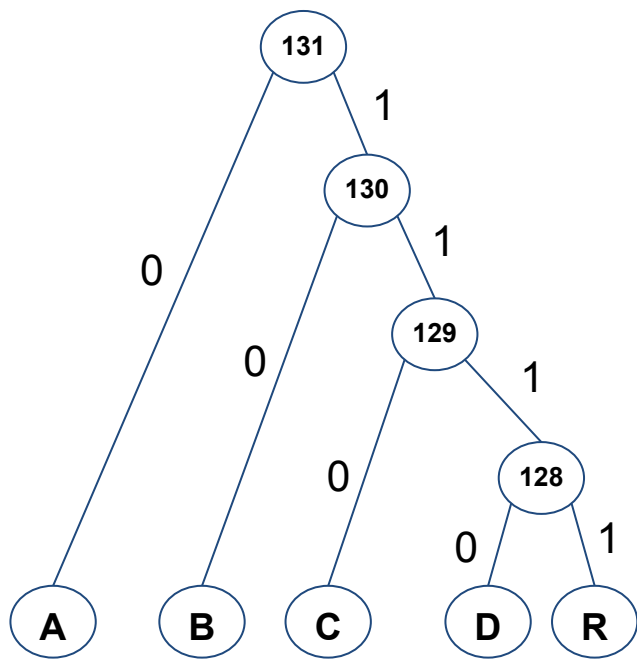


Tout l'arbre est devenu un maximier !



18	15	11	10	15	6	1	7	4	9
0	1	2	3	4	5	6	7	8	9





tree nbElements : 5

68 (D)	82 (R)	67 (C)	66 (B)	65 (A)
--------	--------	--------	--------	--------

data

0	0	...	5	3	1	1	...	1	...	0	0	0	0	...	0
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

huffmanTree

-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256	-256
0	1	...	65	66	67	68	...	82	...	128	129	130	131	...	255

TODO (en cours)

TODO : utiliser macro val pour traiter minimiers indirects
affichage pseudo graphique
memcpy
prouver complexité de heap sort et