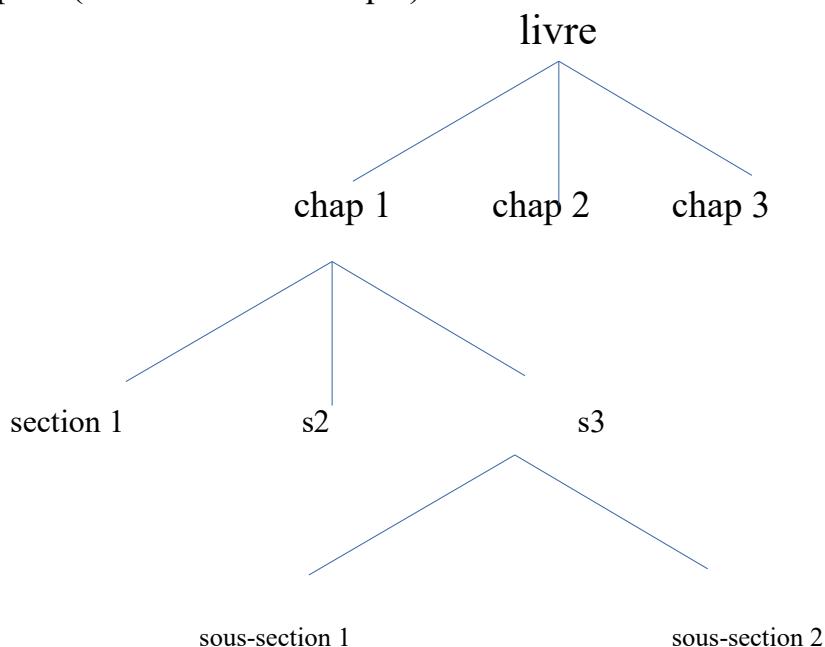


## ARBRES :

exemple : (structure hiérarchique)



livre est composé de 3 chapitres, chapitres constitués de 3 sections, etc..

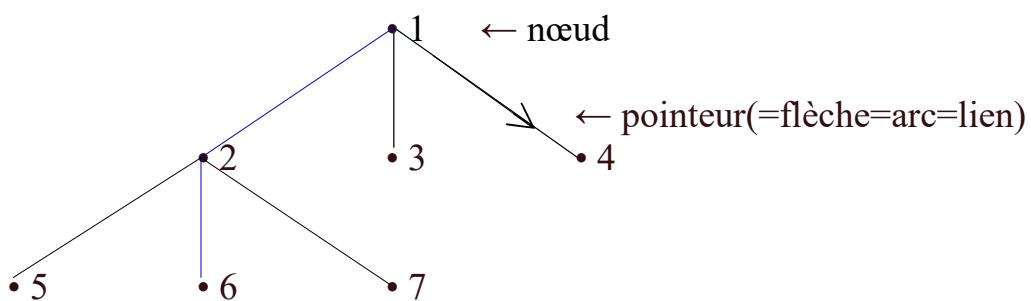
livre = **racine de l'arbre**

autre exemple : france a des régions, constituées de départements, constitués de communes....

Tout est hiérarchisé

## vocabulaire concernant les arbres :

exemple : arborescence A



un nœud (=sommet) est un élément de base de la structure de données

nombre de nœuds : 7

racine de l'arbre : 1

les fils de 1 : 2, 3, 4

3 n'a pas de fils : c'est une **feuille**

feuilles de A : 5, 6, 7, 3, 4

représentation de l'arbre A par le tableau de papa :

nœuds	1	2	3	4	5	6	7
papa	-	1	1	1	2	2	2

Il y a un chemin (path) qui va du nœud 1 vers celui 6  
 un nœud  $y$  est un descendant d'un nœud  $x$  s'il existe un chemin de  $x$  vers  $y$ .  
 dans ce cas,  $x$  est un ancêtre de  $y$   
 la racine est l'ancêtre de tous les autres nœuds  
 tous les nœuds sont descendants de la racine

exemple : les nœuds 3 et 6 ne sont pas en relation ancêtre-descendant car il n'y a pas de chemin entre eux

hauteur de l'arbre : longueur (nombre de flèches traversées) maximum d'un chemin depuis la racine vers une feuille  
 ici  $A = 2$

le niveau d'un nœud  $x$  est la longueur du chemin depuis la racine vers  $x$   
 racine = niveau 0  
 enfants de la racine = niveau 1  
 etc

$N_i$  = ensemble de nœuds du niveau  $i$   
 $\{x\}$  il existe un chemin de longueur  $i$  depuis la racine  
 exemple :  
 $N_2 = \{5, 6, 7\}$   
 $N_3 = \emptyset = N_4 = N_5 \dots$

## ARBRES

Un arbre est défini récursivement.

**Récursivité :** fonction définie en fonction de elle-même, directement ou indirectement

exemple : factorielle

$n! = 1$  si  $n=0$  (base)  
 $n*(n-1)!$  Si  $n>=1$  (récursion)

$n=4$

$4!=4*3!=4*3*2!=4*3*2*1!=4*3*2*1*0!=24$

Factorielle en C :

```
#include <stdio.h>
int fact(int n)
    {if (n==0)
```

```

        return 1;           //LA BASE
return n*fact(n-1);}    //APPEL RÉCURSIF

int main(void){
    int n;
    printf("n = ");
    scanf("%d", &n);
    printf("la factorielle %d est %d", n, fact(n));      //APPEL

```

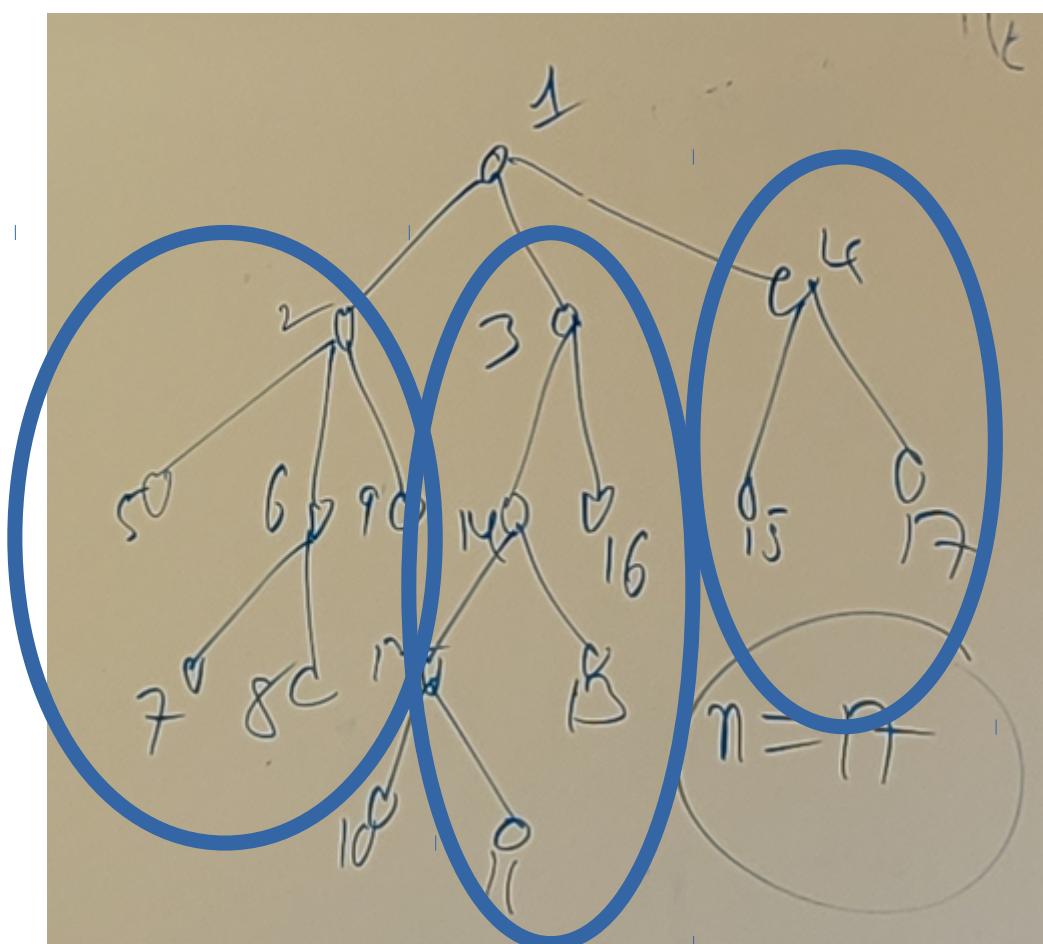
Définition (ARBRE) :

La BASE : un arbre peut être noté par NULL. = racine

Un arbre peut être un seul nœud n. Ce nœud est la racine de l'arbre.

RECURSION :

SI  $A_1, A_2, \dots, A_k$  sont k arbres avec les racines  $n_1, n_2, \dots, n_k$  respectivement, alors on construit un nouvel arbre A avec la racine n

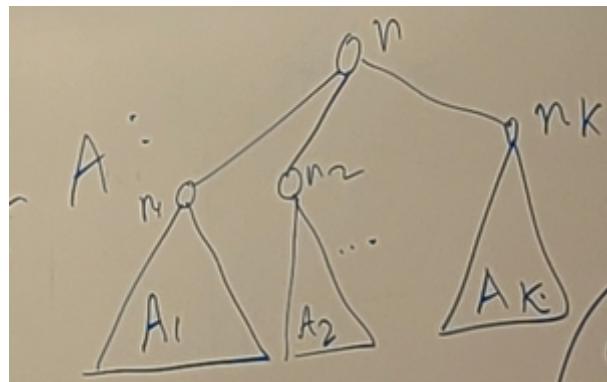


A1  
k=3

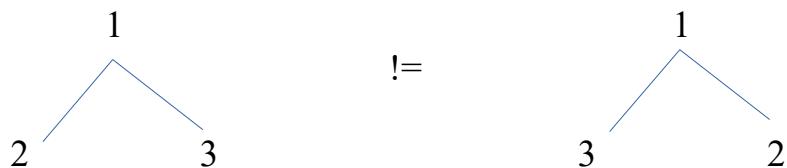
A2

A3

$n=1, n_1=2, n_2=3, n_3=4$



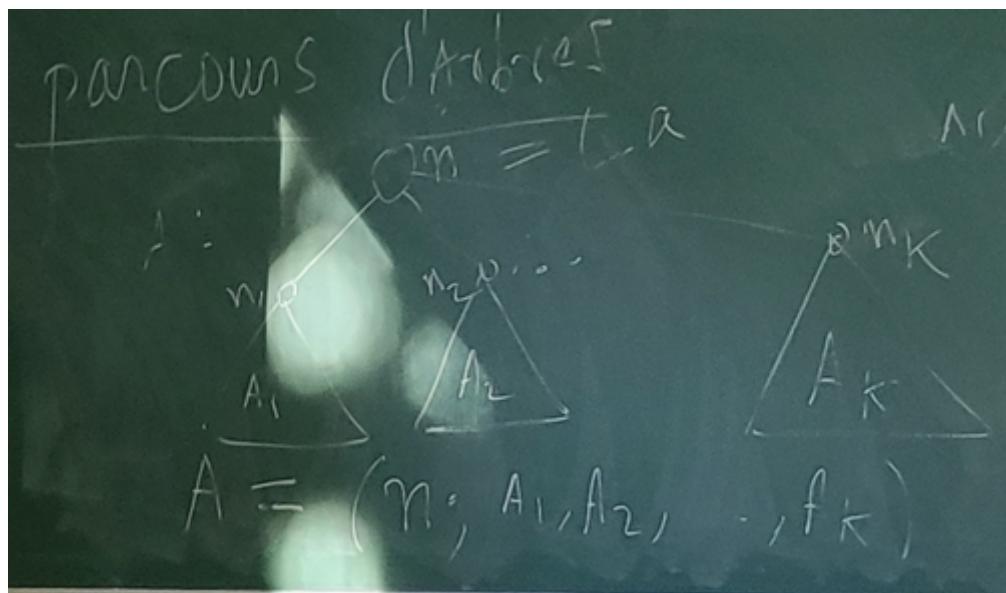
$n_1, n_2, \dots, n_k$  sont les fils/filles de nœud 1.  $A_1, A_2, \dots, A_k$  sont les sous-arbres de 1.



les arbres sont ordonnés

$$A = (1, A_1, A_2, A_3, \dots, A_k) \quad (1 = \text{la racine})$$

parcours d'arbre



$A_1, A_2, A_3, \dots, A_k$  sont les sous arbres de  $A$  ou  $n$

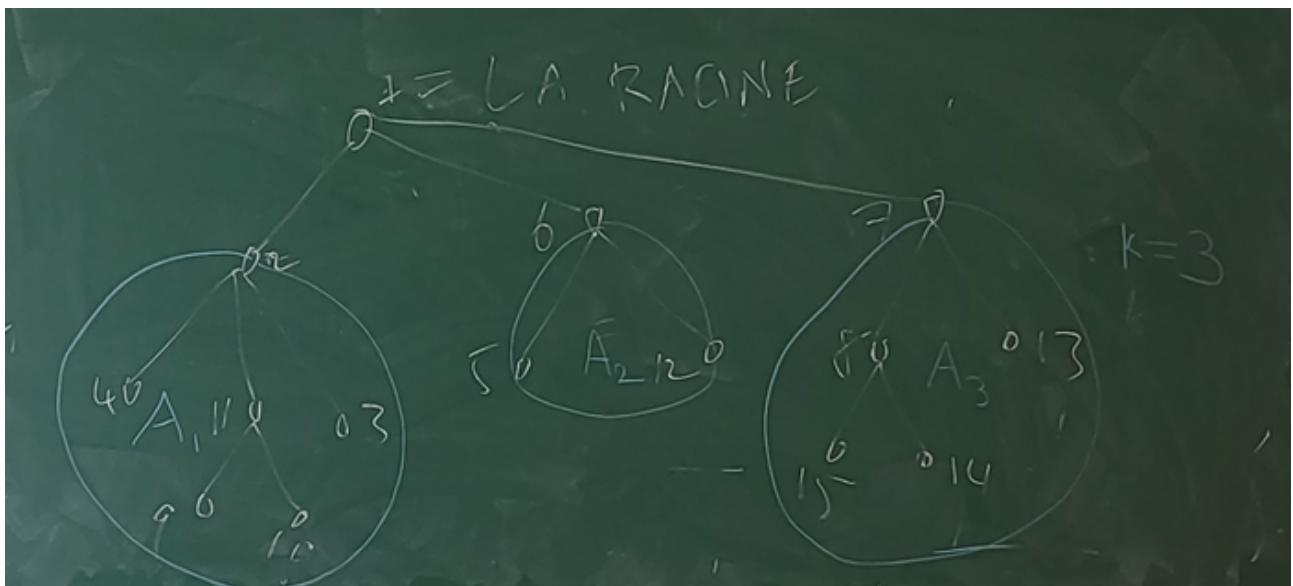
**parcours :** traiter systématiquement chaque nœud de l'arbre de l'arbre  
Il y a 4 façons de parcourir un arbre :

1. préfixe (récursivement)
  2. infixé (récursivement)
  3. postfixe (récursivement)
  4. par niveau
- (les 3 premiers parcours sont définis récursivement)

remarque qui n'a rien à voir : arbre peut être vide, de la même manière qu'un ensemble

**Parcours préfixe :** Soit  $A = (n; A_1, A_2, \dots, A_k)$  ( $n$  étant la racine)

1. BASE Si  $A = \text{NULL}$ , alors  $\text{préfixe}(A) = [] = \text{une liste vide}$
  - Si  $A = (n)$ ,  $A = \bullet n = \text{il n'y a pas de sous arbre}$   
alors  $\text{préfixe}(A) = [n]$
  - Récursion : Si  $A = (n; A_1, A_2, \dots, A_k)$   
alors  $\text{préfixe}(A) = [n, \text{préfixe}(A_1), \text{préfixe}(A_2), \dots, \text{préfixe}(A_k)]$
- Exemple (préfixe) : trouver la liste de nœuds dans l'ordre préfixe



$$\text{préfixe}(A) = (1, \text{préfixe}(A_1), \text{préfixe}(A_2), \text{préfixe}(A_3))$$

- on se concentre sur le sous-arbre  $A_1$  :

$$\text{préfixe}(A_1) = (2; \text{préfixe}(4), \text{préfixe}(\bullet 11, \bullet 9, \bullet 10), \text{préfixe}(3))$$

on développe  $\text{préfixe}(\bullet 11, \bullet 9, \bullet 10)$

$$\begin{aligned} \text{préfixe}(\bullet 11, \bullet 9, \bullet 10) &= (11; \text{préfixe}(9), \text{préfixe}(10)) \\ &= (2, 4, 11, 9, 10, 3) \end{aligned}$$

donc, après s'être concentré sur  $A_2$  et  $A_3$ , on a :

$$\text{préfixe}(A) = (1, 2, 4, 11, 9, 10, 3, 6, 5, 12, 7, 8, 15, 14, 13)$$

**Parcours infixé :**

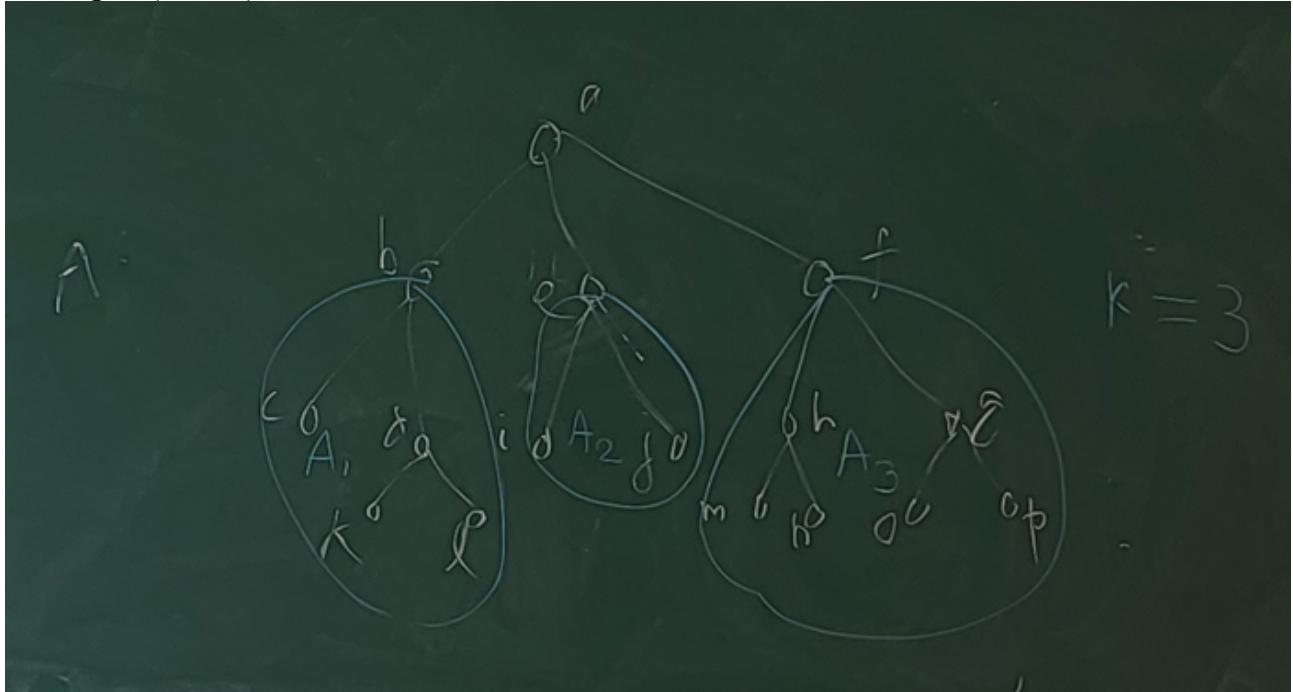
- BASE Si  $A = \text{NULL}$ , alors  $\text{infixe}(A) = [] = \text{une liste vide}$

Si  $A = (n)$ ,  $A = \bullet n =$  il n'y a pas de sous arbre  
alors  $\text{infixe}(A) = [n]$

Récursion :  $A = (n; A_1, A_2, \dots, A_k)$

alors  $\text{infixe}(A) = (\text{infixe}(A_1), n, \text{infixe}(A_2), \text{infixe}(A_3), \dots, \text{infixe}(A_k))$

Exemple (infixe) :



$$\text{infixe}(A) = (\text{infixe}(A_1), a, \text{infixe}(A_2), \text{infixe}(A_3))$$

$$\begin{aligned} \text{infixe}(A_1) &= (\text{infixe}(c), b, \text{infixe}(d, k, l)) \\ &= (c, b, k, d, l) \end{aligned}$$

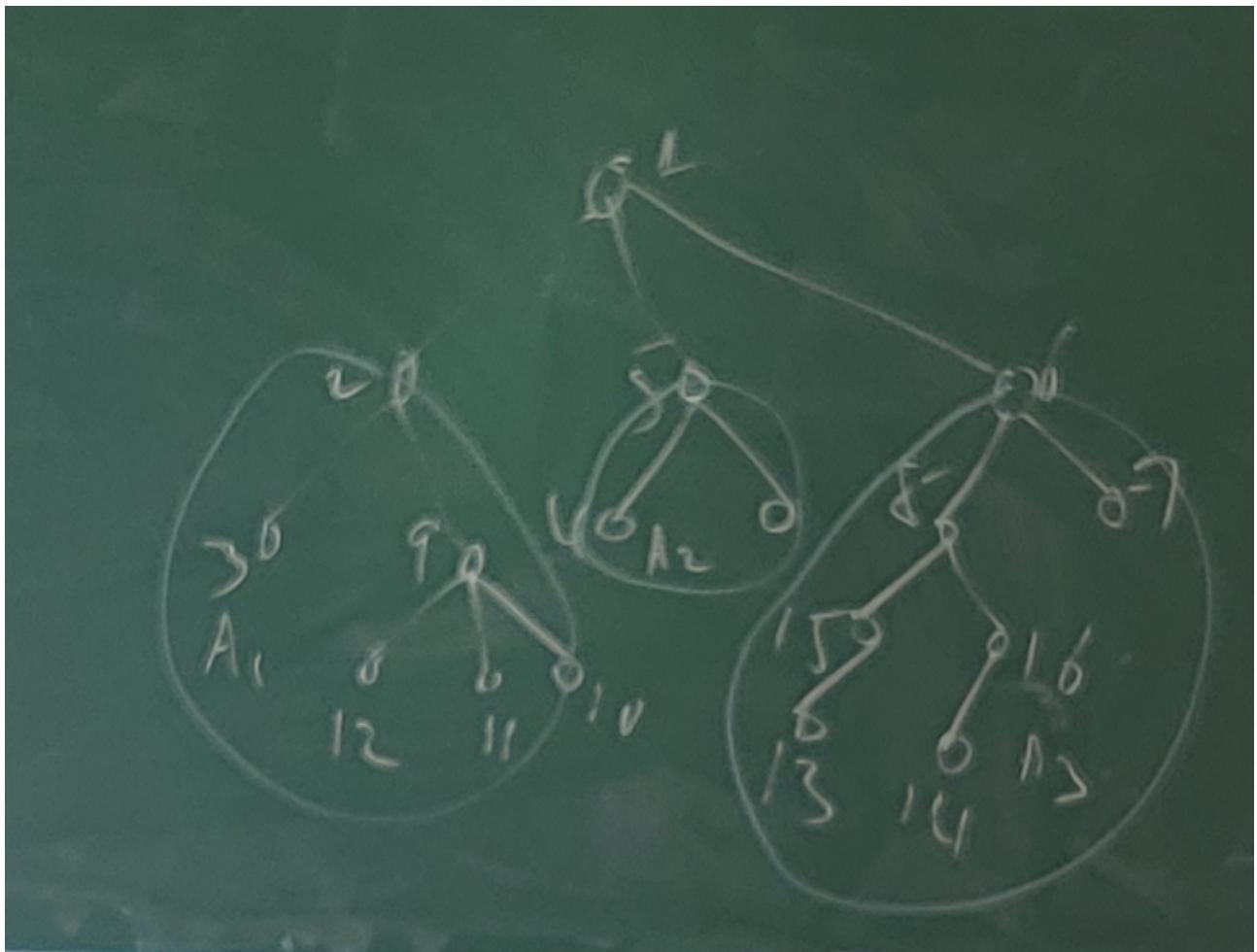
$$\text{infixe}(A) = (c, b, k, d, l, a, i, e, j, m, h, n, f, o, g, p)$$

### parcours postfixe :

BASE      Si  $A = \text{NULL}$ , alors  $\text{postfixe}(A) = [] =$  une liste vide  
 Si  $A = (n)$ ,  $A = \bullet n =$  il n'y a pas de sous arbre  
 alors  $\text{postfixe}(A) = [n]$

Récursion :  $A = (n; A_1, A_2, \dots, A_k)$

$\text{postfixe}(A) = (\text{postfixe}(A_1), \text{postfixe}(A_2), \dots, \text{postfixe}(A_k), n)$



$\text{postfixe}(A) = (\text{post}(A_1), \text{post}(A_2), \text{post}(A_3), 1)$

$\text{postfixe}(A_1) = (\text{post}(3), \text{post}(9, 12, 11, 10), 2)$   
 $= (3, 12, 11, 10, 9, 2)$

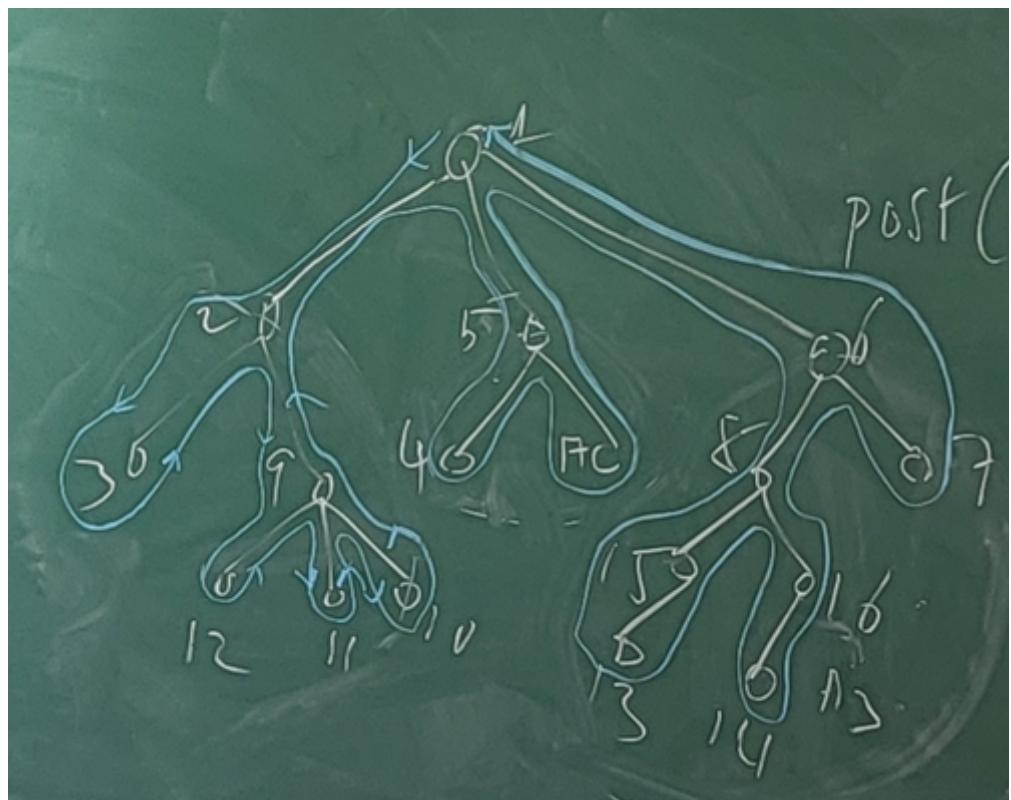
donc

$\text{postfixe}(A) = (3, 12, 11, 10, 9, 2, 4, 17, 5, 13, 15, 14, 16, 8, 7, 6, 1)$

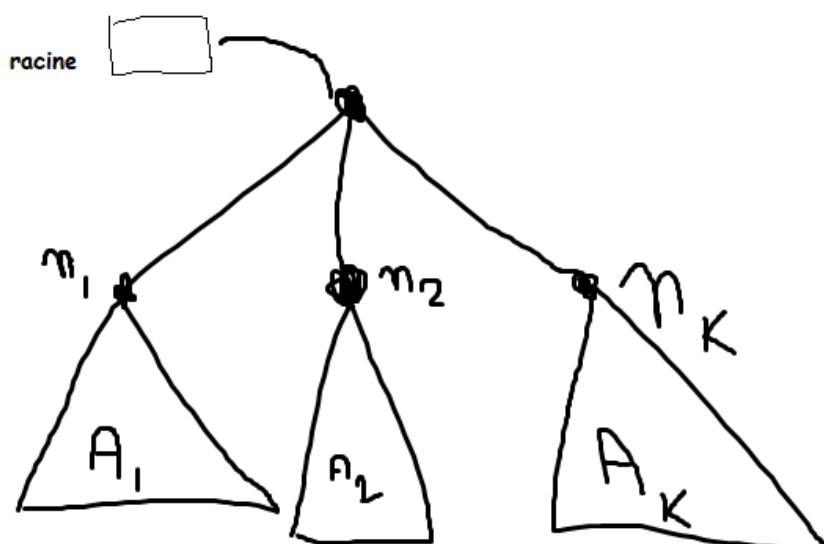
### ASTUCE POUR LES 3 PARCOURS

Imaginons une promenade autour de l'arbre dans le sens contraire des aiguilles d'une montre, comme indiqué dans la figure.

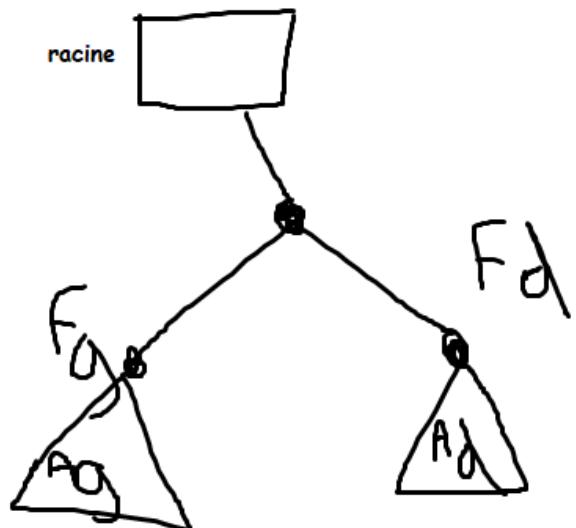
$\text{préfixe}(A) = \text{écrire les nœuds pendant la première rencontre}$   
 $= (1, 2, 3, 9, 12, 11, 10, 5, 4, 17, 6, 8, 15, 13, 16, 14, 7)$



un arbre ordonné :



Un arbre binaire :



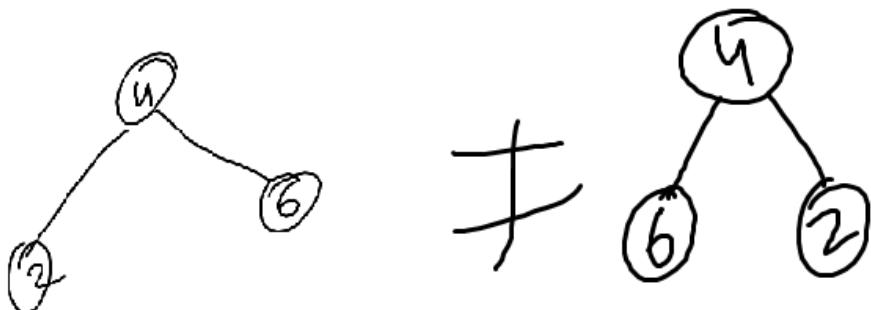
chaque nœud possède 2 enfants, fils gauche et fils droit

Ag : sous arbre gauche

Ad : sous arbre droit

Un arbre binaire peut-être vide noté par NULL.

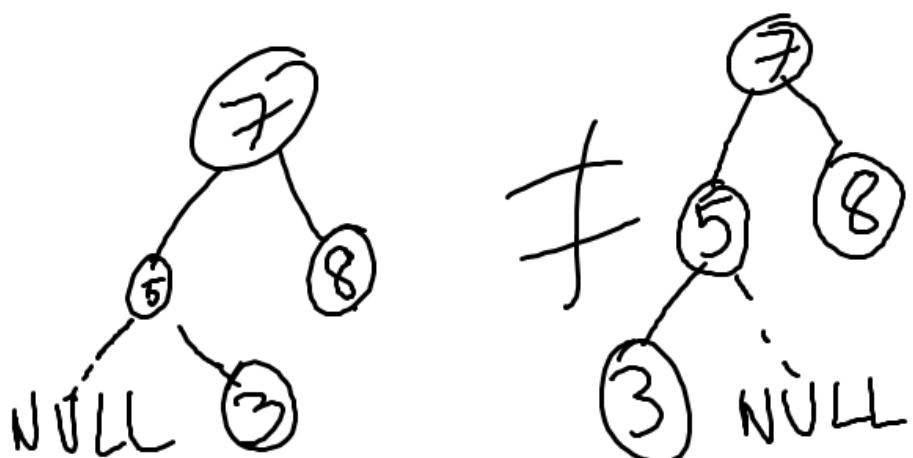
Exemple : arbre binaire



$$(2,6) \neq (6,2)$$

ABR : (arbre binaire recherche)

C'est un arbre binaire tel que  $fg < \text{papa}$  et  $fd > \text{papa}$   $\forall$  nœud de l'arbre



Arbre binaire mais pas ABR

ABR

méthode de construction d'un ABR :

la première valeur est la racine. Ensuite évoluer selon la méthode suivante :  
si plus petit on va à gauche  
si plus grand on va à droite

def: arbre binaire complètement équilibré

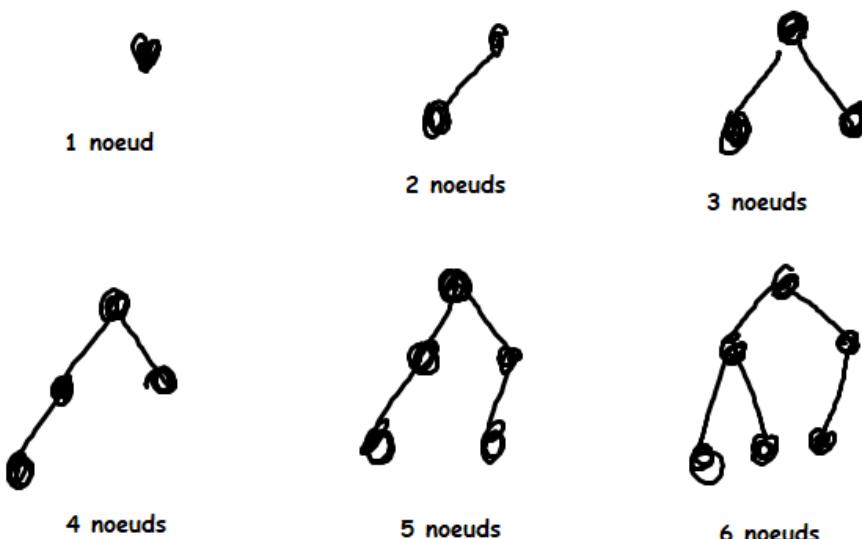
Un arbre binaire est complètement équilibré si

$$(Ag - Ad) \leq 1$$

pour tout nœud

arbre binaire complètement équilibré

ex :

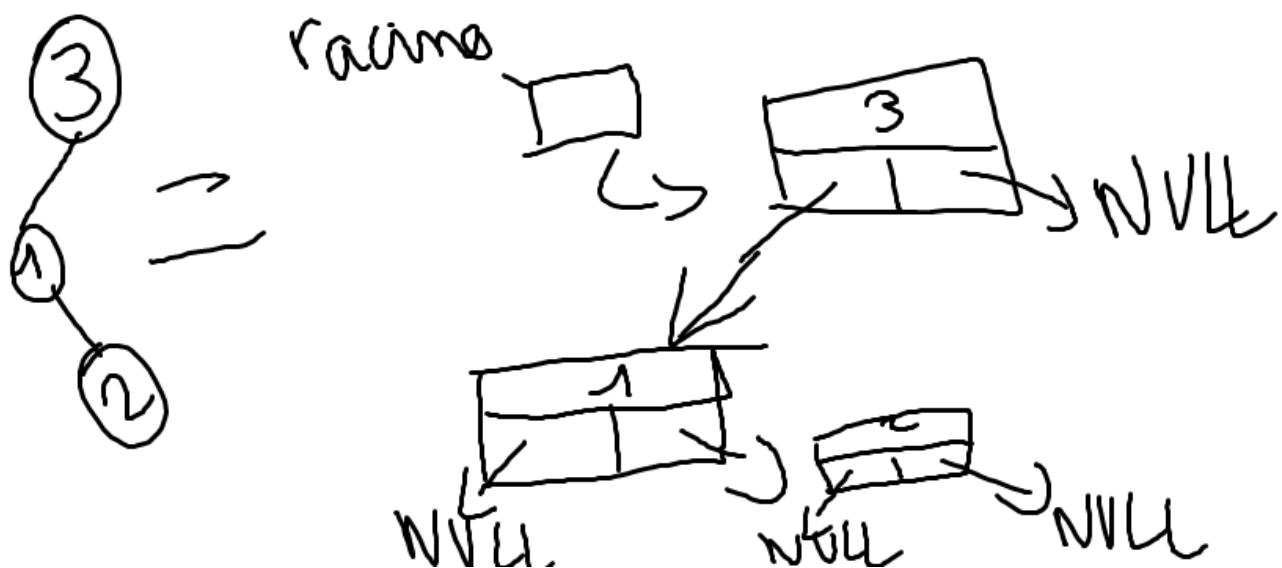


### un nœud en c :

```
struct noeud
{int info;
struct noeud *fg, *fd ;};
```

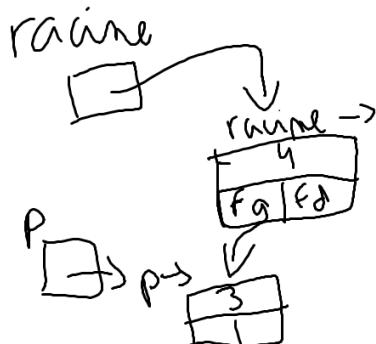
visualisation de nœud :

Info	
fg	fd



```

struct noeud {int info; struct noeud *fg, *fd;};
struct noeud *racine, *p;
racine = malloc(sizeof(*racine));
racine -> info = 4;
p=malloc(sizeof(*p));
p -> info = 3;
racine-fg=p;
  
```

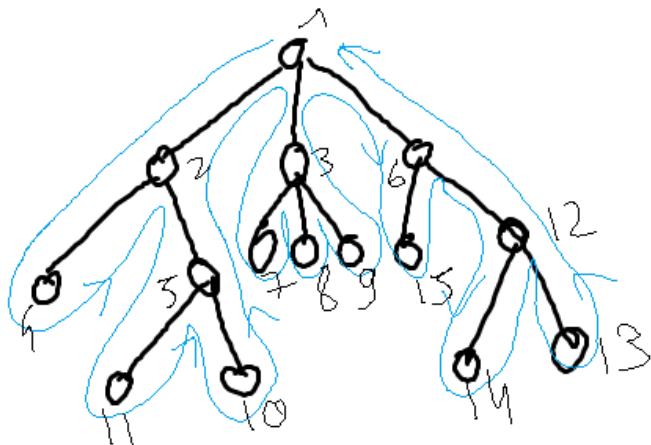


```

#include <stdio.h>
#include <stdlib.h>
struct noeud {
int info;
struct noeud *fg, *fd;};
typedef struct noeud * lien;           // (struct noeud * = lien)
lien racine, int n;                  // n = nombre de noeuds
//construction de l'arbre binaire parfaitement équilibré de n noeuds
arbre (int n){
int ng, nd, x;
lien p;
if (n==0) return NULL;
else {
}}}
  
```

parcours d'arbre SUITE

ex



**préfixe(A)** : écrire les nœuds dès la première rencontre :  
(1,2,4,5,11,10,3,7,8,9,6,15,12,14,13)

**postfixe(A)** : écrire les nœuds à la dernière rencontre :  
(4, 11, 10, 5, 2, 7, 8, 9, 3, 15, 14, 13, 12, 6, 1)  
(faut écrire les nœuds qu'on ne verra plus : adieu!!!)

**infixe(A)** : écrire les feuilles dès la 1ère rencontre, les autres nœuds (nœuds pas feuilles=noeuds intérieurs) pendant la deuxième rencontre (dans la BAY):  
(4, 2, 11, 5, 10, 1, 7, 3, 8, 9, 15, 6, 14, 12, 13)

écrivons un schéma d'algorithme pour préfixe, postfixe et infixe

```
void prefixe(noeud n){  
/*écrire les nœuds de l'arbre dans l'ordre préfixe*/  
ecrire(n);  
pour tout fils f de n dans l'ordre de gauche vers la droite faire prefixe(f)//appel  
récuratif  
}
```

```
void postfixe(noeud n){  
/*écrire les nœuds de l'arbre dans l'ordre postfixe*/  
pour tout fils f de n dans l'ordre de gauche vers la droite faire prefixe(f)//appel  
récuratif  
ecrire(n);  
}
```

(infixe(A)=(infixe(A<sub>1</sub>); n; infixe(A<sub>2</sub>), infixe(A<sub>3</sub>),...infixe(A<sub>k</sub>)))

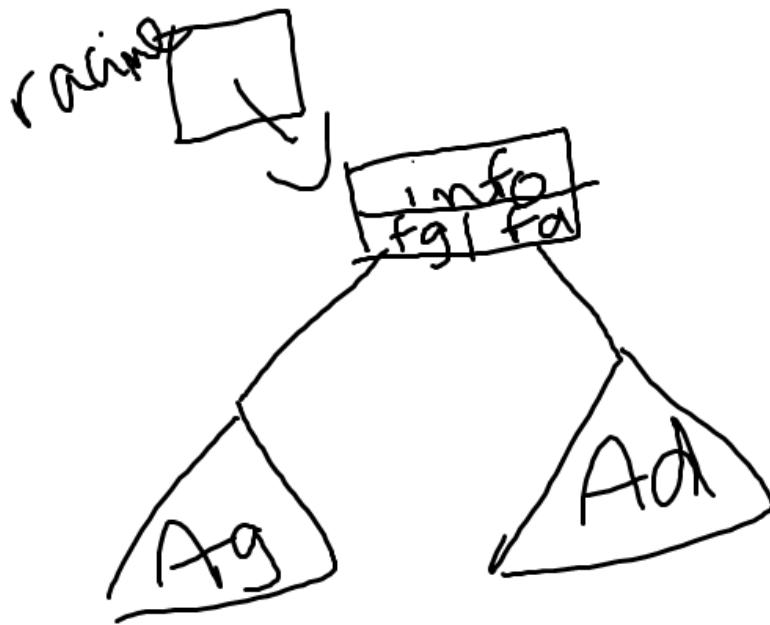
```
void infixe(noeud n){  
/*écrire les nœuds de A dans l'ordre infixe*/
```

```

si n est une feuille de A alors ecrire(n)
sinon {
infixe(le fils le plus à gauche de n)
ecrire(n);
pour tout fils f (sauf le plus à gauche) dans l'ordre de gauche à droite faire infixe(f)
//appel récursif}
}

```

un arbre binaire



préfixe, infixé, postfixe pour arbre binaire : (en c)

```

void prefixe(lien r){
if (r!=NULL){
    printf("%d", r->info);
    prefixe(r -> fg);           //APPEL RÉCURSIF
    prefixe(r -> fd);           //APPEL RÉCURSIF
}
}

```

```

void postfixe(lien r){
if (r!=NULL){
    postfixe(r -> fg);         //APPEL RÉCURSIF
    postfixe(r -> fd);         //APPEL RÉCURSIF
    printf("%d", r->info);
}
}

```

```

void infixe(lien r){
if (r!=NULL){
    infixe(r -> fg);           //APPEL RÉCURSIF
    printf("%d", r→info);
    infixe(r -> fd);          //APPEL RÉCURSIF
}
}

```

écrire un programme en c pour construire un arbre parfaitement équilibré de n noeuds

```

#include <stdio.h>
#include <stdlib.h>
struct noeud {
int v;
struct noeud *g, *d;
} ;
int n;
struct noeud *racine;
struct noeud *arbre(int n){
int x, ng, nd;
struct noeud *p;
if (n==0) return NULL;
else{
    ng=n/2;
    nd=n-ng-1;
    scanf("%d", &x);
    p=(struct noeud )*malloc(sizeof *p);
    p-> n=x;
    p-> ng=arbre(ng);           //APPEL RÉCURSIF
    p-> nd=arbre(nd);
}
return p;
}

//affichage de l'arbre
void printtree(struct noeud *t, int h){
//cette fonction imprime l'arbre t avec indentation
int i;
if (t!=NULL){
    printtree(t-> d, h+1);
    for(i=1; i<=h; i++) printf(" ");
    printf("%d\n", t→n);
    printtree(t->g, h+1);
}
}

```

```

int main(){
    scanf("%d", &n);
    racine=arbre(n);
    printtree(racine, 0);
    return 0;
}

```

construction d'un arbre binaire de recherche

```

struct noeud {
    int v, compteur;
    struct noeud *g, *d;};
}

void rech_insertion(struct noeud *p, int x){
if (p==NULL){
    //créer un nœud
    p=(struct noeud *)(malloc(sizeof *p));
    p->v=x; p->
    compteur=1;
    p->g=NULL;
    p->d=NULL;
}
else{
    if (x<p->v ) rech_insertion(p->g,x)
    else if(x>p->v) rech_insertion(p->d,x)
    else p->compteur++;
}
}

int main(){
    //initialisation de racine
    racine=NULL;
    while(scanf("%d", &k)!=EOF){
        rech_insertion(racine,k);}
    printtree(racine, 0);
}

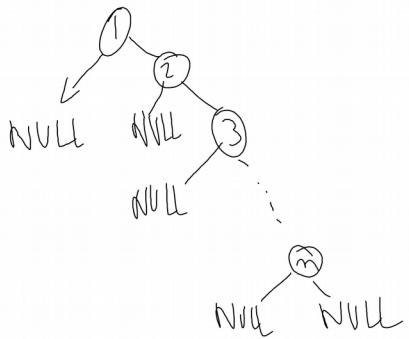
```

### Arbres binaires complets et quasi-complets

construisons un arbre ABR avec les entrées : 1, 2, 3, ... n.

Nous aurons un arbre binaire de recherche ABR qui est **dégénéré** (= une liste chaînée)

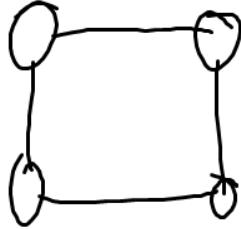
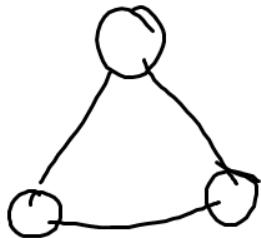
L'élément inséré sera automatiquement feuille de l'arbre (plus grand donc va toujours à droite)



La hauteur de cet arbre =  $h$  est le nombre maximum de flèche à traverser depuis la racine pour atteindre une feuille :  $h = n-1$ .

#### Remarque :

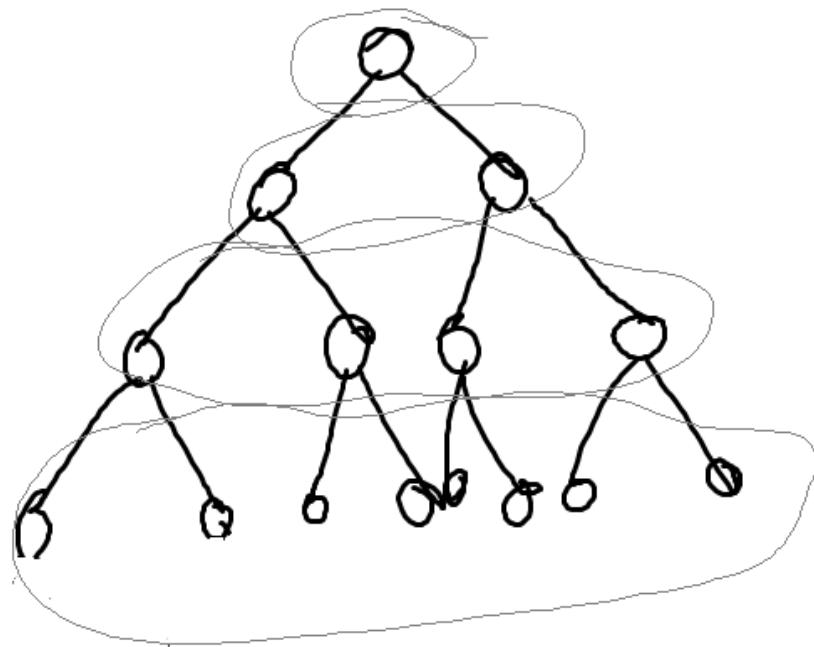
- Un arbre de  $n$  nœuds possède exactement  $n-1$  arcs (=flèches)
- Depuis la racine il n'y a qu'un seul chemin vers un autre nœud
- Un arbre n'a pas de cycles  
(ex: cycles : triangle)



etc

#### Arbre binaire complet :

exemple :



$$n_0 : 2^0 \text{ nœuds}$$

$$n_1 : 2^1 \text{ nœuds}$$

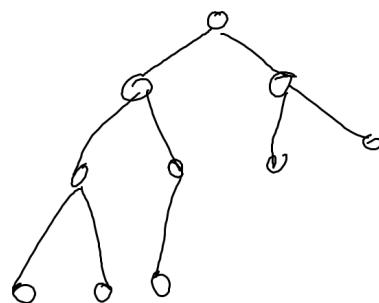
$$n_2 : 2^2 \text{ nœuds}$$

$$n_3 : 2^3 \text{ nœuds}$$

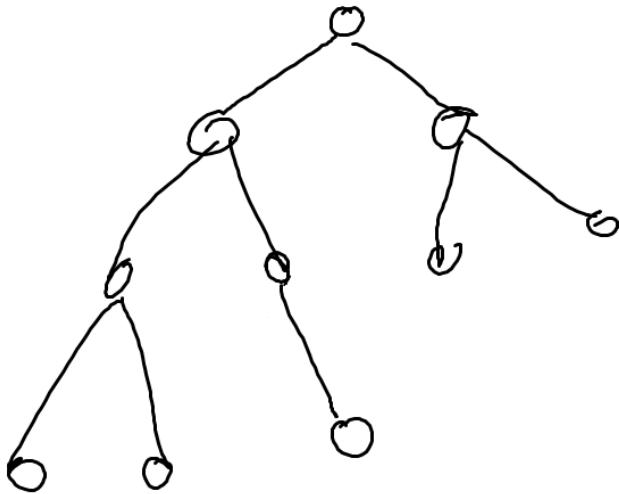
hauteur de cet arbre = 3

### Arbre binaire quasi complet :

exemple :



exemple d'arbre **pas** quasi complet :



Les feuilles de l'arbre binaire quasi complet sont toutes alignées vers la gauche.

Trouvons la relation entre la hauteur  $h$  et le nombre de nœuds d'un arbre binaire complet

$$n_0 : 2^0, n_1 : 2^1, n_3 : 2^3 \dots n_h : 2^h$$

$$\text{Donc, } n = \text{le nombre de nœuds} = n_0 + n_1 + n_2 + \dots + n_h \\ = 2^0 + 2^1 + 2^2 + \dots + 2^h$$

#### FORMULE :

$$1 + r + r^2 + \dots + r^{n-1} = \frac{r^n - 1}{r - 1}$$

= série géométrique où  $r$ =la raison et  $r!=1$

ici,  $r=2$  et  $n=h+1$ . donc

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^h = \frac{2^{h+1} - 1}{2} - 1 = 2^{h+1} - 1$$

$$2^{h+1} = n + 1$$

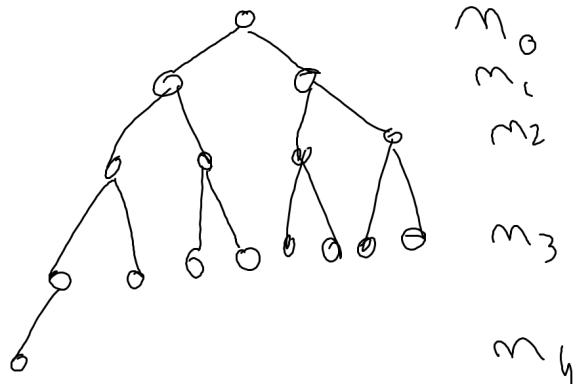
$$h+1 = \log_2(n+1)$$

$$h = \log_2(n+1) - 1 \approx \log n$$

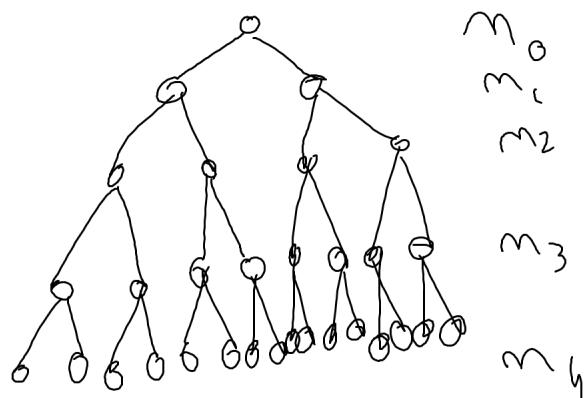
$n$	$\log_2 n$
10	$\approx 3 (2^3 \approx 10)$
100	$\approx 7 (2^7 \approx 100)$
1000	$\approx 10 (2^{10} \approx 1000)$
$10^6$	$\approx 20$
$10^9$	$\approx 30$

Trouvons la relation entre  $n$  (le nombre de nœuds) et la hauteur d'un arbre binaire quasi complet  
 (évidemment, un arbre binaire complet est un arbre binaire quasi complet mais la réciproque est fausse)

arbre quasi complet de hauteur  $h = 4$ :



arbre binaire quasi complet (car complet) de hauteur 4 :



$$2^0 + 2^1 + 2^2 + 2^3 + 1 \leq n \leq 2^0 + 2^1 + 2^2 + 2^3 + 2^4$$

plus généralement :

$$2^0 + 2^1 + 2^2 + \dots + 2^{h-1} + 1 \leq n \leq 2^0 + 2^1 + 2^2 + \dots + 2^h$$

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

ou

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

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

ou

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

$$\log_2(2^h) \leq \log_2(n) < \log_2(2^{h+1})$$

$$h \leq \log_2(n) < h+1$$

ou

$$h \approx \log_2 n$$

La suppression d'un nœud dans un ABR :

on distingue 3 cas :

cas 1 : (Fais néant)

L'élément cherché à éliminer n'est pas dans l'arbre

cas 2 :

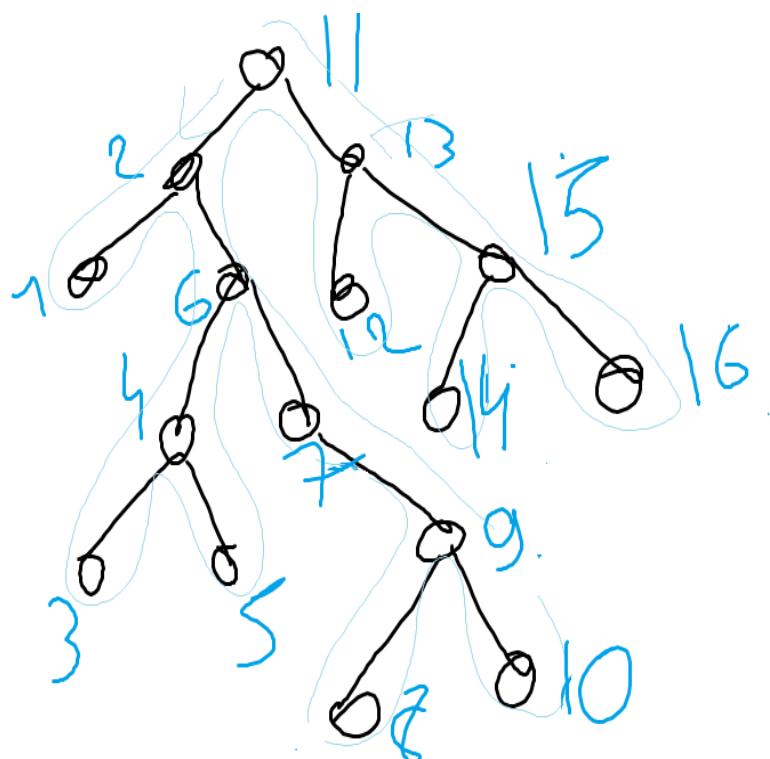
Le nœud à éliminer a au plus 1 enfant (donc 0 ou 1).

cas 3 : le nœud à éliminer a exactement 2 enfants (enfant gauche et enfant droit)

Illustrons cet algo par des exemples :

exemple 1 :

ABR :



Parcours infixé : (=tri)

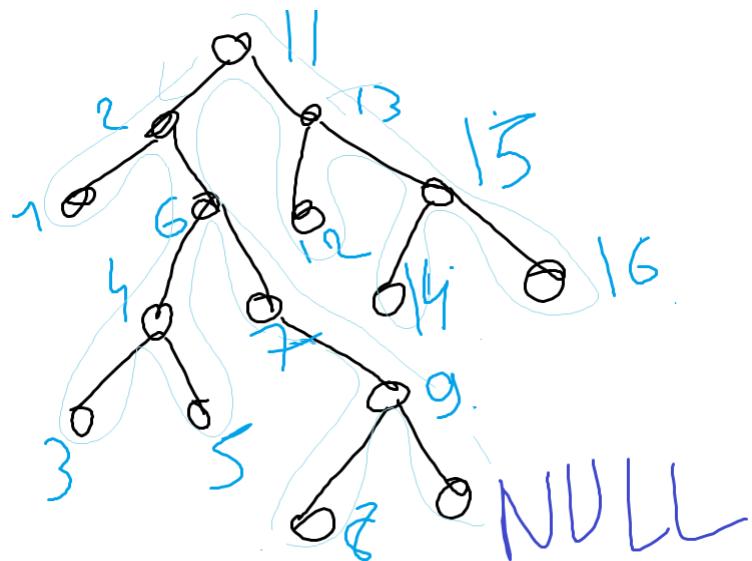
écrire les feuilles dès la 1ère rencontre. Les autres (les nœuds) intérieurs pendant la 2ème rencontre

**cas 1 :**

Éliminons 10

c'est une feuille qui n'a pas d'enfants

l'arbre devient



pointeur droit de 9 = NULL

Éliminons 3 : c'est une feuille

affectation : pointeur pointeur gauche de 4 = NULL

**cas 2 :**

éliminons 7, qui a exactement un enfant

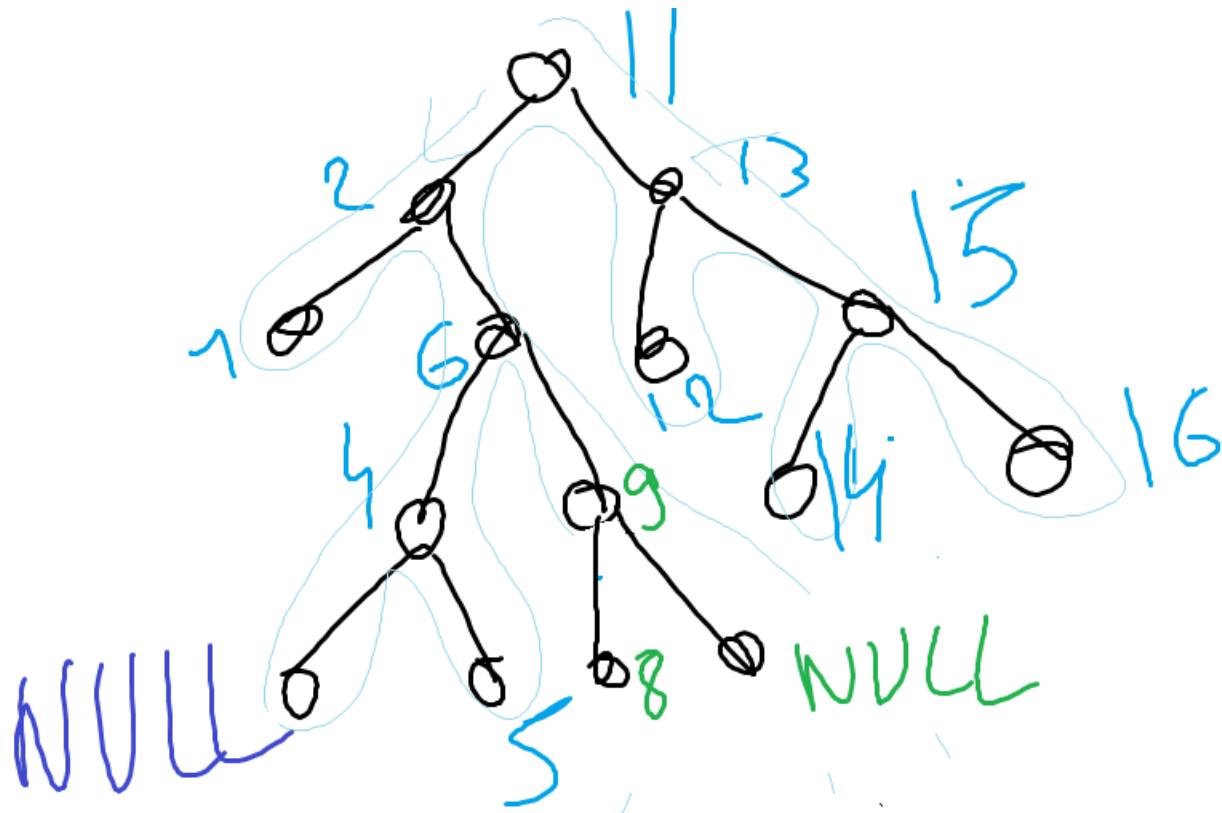
avant :



après



l'ABR devient donc



**cas 3 :** le nœud à éliminer a exactement 2 enfants

ex: éliminons 10

algo : descendre vers le fils gauche de 10 : 2

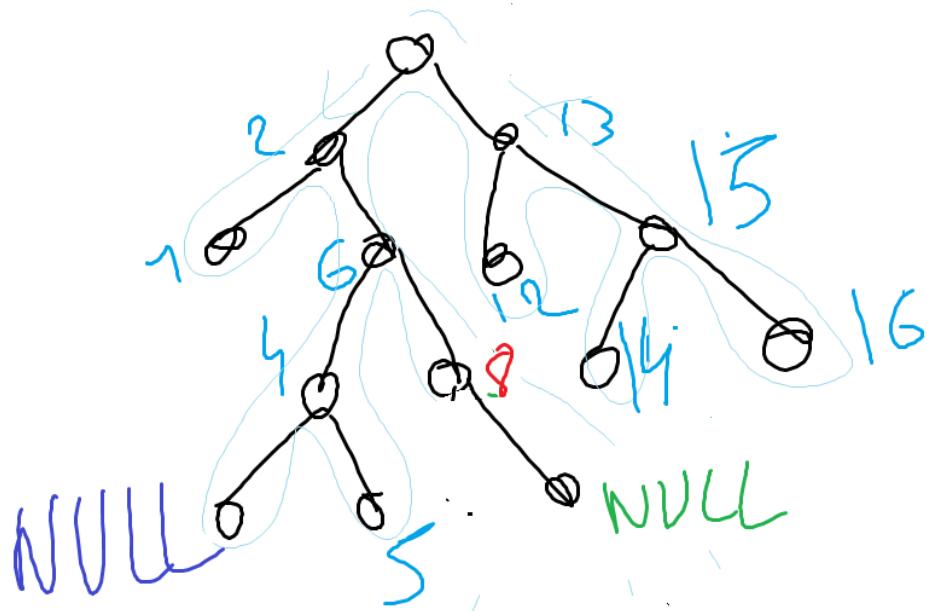
dans le sous-arbre gauche de 10, chercher un nœud le plus à droite : 9

notons que 1 est l'entier maximum dans  $A_{g10}$  (arbre gauche de 10)

2 est un nœud possédant au plus un enfant

pour éliminer 10 : on copie le nœud contenant 10 une fois copié, éliminons le nœud 9 qui a au plus 1 enfant

on tombe sur le premier cas



## Démonstration d'algorithmes (l'invariant de la boucle) terminaison d'algorithmes, et complexité d'algorithmes

système : une collection d'entités d'entités et les interactions entre ces entités



exemple de système : algo

**def :** L'invariant d'un système est une propriété/énoncé qui se conserve à travers les transformations subies par le système.

**Système :** un ensemble d'entités et les interactions entre ces entités

programme d'ordinateur : l'invariant d'un programme sont attachés aux certains pointeurs d'un programme. Ils sont les ponts entre le texte du programme (qui est statique) et son exécution (qui est dynamique).

l'invariant d'une boucle : c'est une propriété/une condition qui reste vraie à un point donné de la boucle (Ex: l'entrée de la boucle) quel que soit le nombre d'exécutions effectuées par la boucle auparavant.

Il y a 3 composants à démontrer l'invariant de la boucle

- 1. Initialisation
- 2. Conservation/Préservation
- 3. Terminaison

Ex: trouver l'invariant de la boucle

INPUT : deux entiers  $m > 0, n > 0$

OUTPUT : pgcd( $m, n$ ) (plus grand commun diviseur)

ALGO : (Euclide 3 siècles av. J-C)

l'algo même est l'invariant

$$(\text{pgcd}(30, 15)=15$$

$$\text{pgcd}(36, 18)=18 \text{ LA BASE}$$

$$\text{pgcd}(m, n) = \begin{cases} n & \text{si } m \% n == 0 \text{ (LA BASE)} \\ \text{pgcd}(n, m \% n) & \text{si } m \% n != 0 \end{cases}$$

( $m \% n$  = le reste de la division de  $m$  par  $n$  :  $r=m \% n$ )

fragment en c :

$$r=m \% n;$$

```

while (r!=0){
    m=n;
    n=r;
    r=m%n;
}
return n;

```

Démonstration :

Prouvons que  $\text{pgcd}(m,n)=\text{pgcd}(n,m \% n)$  si  $m \% n \neq 0$

LA BASE : soit  $r=m \% n$

si  $r=0$ , alors  $n$  divise  $m$  et  
évidemment,  $\text{pgcd}(m,n)=n$ .

Prouvons maintenant que si  $r \neq 0$  alors  $\text{pgcd}(m,n)=\text{pgcd}(n,r)$

démonstration :

soit  $d$  divise  $m$  et  $d$  divise  $n$

montrons que  $d$  divise  $n$  et  $r$

$d$  divise  $n$  par supposition. Il nous reste à montrer que  $d$  divise  $r$ . mais  $r =$  le reste de la division de  $m$  par  $n$ .

Donc, on peut écrire  $m=nq+r$ ,  $0 \leq r < n$

(ou  $r = m - nq$ )

(rappel : si  $x$  divise  $a$ ,  $x$  divise  $b$ ,

alors  $x$  divise  $a+b$ ,  $a-b$ ,  $pa+qb$ , etc)

$d$  divise  $m$  et  $d$  divise  $n$  (supposition)

donc  $d$  divise  $m-nq$

donc  $D(m, n)$  est inclus dans  $D(m, r)$

où  $D(m, n)=$  l'ensemble de tous les diviseurs positifs de  $m$  et  $n$

et  $D(n, r)=$  l'ensemble de tous les diviseurs positifs de  $n$  et  $r$

Montrons maintenant  $D(n, r)$  est inclus dans  $D(M,n)$

soit  $d \in D(n,r)$  alors  $d$  divise  $n$  et  $d$  divise  $r$ .

prouvons que  $d$  divise  $m$  et  $n$

(mais  $d$  divise  $n$  est dans l'hypothèse)

il suffit de montrer que  $d$  divise  $r$ .

MAIS  $m = nq+r$ ,  $0 \leq r < n$

$d$  divise  $r$  divise  $n$  (hypothèse)

donc  $d$  divise  $nq+r=m$

donc  $D(m,n)=D(n, r)$

(ex:  $D(18,12)=D(12,6)$

$\{1,2,3,6\}=\{1,2,3,6\}$

)

$\max D(m,n) = \max D(n,r)$

$\downarrow$                      $\downarrow$

$\text{pgcd}(m,n) = \text{pgcd}(n,r)$

convergence : une des propriétés requises d'un algorithme est que il se termine après un nombre fini d'opérations

méthode générale pour prouver la convergence ou terminaison d'une boucle while :

on pose une fonction N dépendant certaines variables de la boucle while avec la propriété suivante :

1. si b est vrai alors  $N > 0$
2. chaque exécution de la boucle diminue la valeur de la fonction N

Ex: montrons que l'algo euclide se termine

```
r = m%n;  
while (r!=0){  
    m=n;  
    n=r;  
    r=m%n;}  
return n;
```

Terminaison d'algorithmiques : (suite)

pgcd(m,n):

```
r=m%n;  
while(r!=0){  
    m=n;  
    n=r;  
    r=m%n;  
}  
return n;
```

Posons une fonction entière

$N=m \% n$

STEP 1 : si  $B(=r!=0)$  est vrai alors  $N>0$

$B=\text{vrai}$  veut dire  $r!=0$ , c à d  $m \% n!=0(>0)$

mais  $N=m \% n$

donc,  $N>0$

STEP 2 : chaque itération diminue la valeur de N

à l'entrée while :

$0 < r < n$  (=diviseur)

$m=n$ ;  $n=r$ ;

donc la valeur de n diminue à chaque itération

Une suite strictement décroissante positive se termine

problème (non résolu!)

la fonction suivante se termine

```

int puzzle(int N){
    if (N==1) return 1;
    if (N%2==0) puzzle(N/2);           //APPEL RÉCURSIF
    return puzzle(3*N+1);             //APPEL RÉCURSIF
puzzle(5) renvoie 1

```

ex :

input : m, n entiers  $> 0$

output : le pgcd(m,n)

algo :  $\text{pgcd}(m,n)=m$       si  $m=n$       (la base)

iteration (invariant):       $\text{pgcd}(m,n)=\text{pgcd}(m-n,n)$ , si  $m>n$   
 $\text{pgcd}(m,n)=\text{pgcd}(m, n(m))$  si  $n>m$

Ecrire une boucle while pour cet algo

prouver l'invariant et montrer la terminaison de while (la fonction entière  
 $N=\max(m,n)$ )

ex:

input : un tableau t de taille n trié  $\leq$  et un entier x

output :      oui si x appartient à t  
               non si x appartient pas à t

i=1;

j=n;

do

```

k=(i+j)/2;
if (x>t[k]) i=k+1;
else j=k-1;
while ((t[k]!=x)&&(i<=j))

```

trouver l'invariant de la boucle

### La complexité des algorithmes :

un problème est une question à répondre par "oui" ou "non"

(le problème de décision :

Un problème d'optimisation se transforme en une suite des problèmes de décisions)  
par rapport aux paramètres des problèmes

paramètres : les paramètres sont décrits sans préciser la valeur

### La taille du problème : noté par n

a chaque problème on associe un entier n appelé la taille du problème

ex : problème : trier n entiers dans l'ordre

la taille = n

pour un arbre de n nœuds la taille = n

pour le problème de pgcd(m,n)

la taille =  $\max(m,n)$       (une manière informelle)

de manière formelle :  $\log \max(m,n)$

La complexité d'un algo est exprimée en fonction de la taille  $n$  du problème, notée par  $T(n) = \text{nombre d'instructions élémentaires effectuées par l'algo}$   
 $(T \text{ pour temps})$

Notation  $O, \Omega, \theta$  (03A9),  $\Theta$  (03B8) :

on écrit  $f(n) = O(g(n))$

si  $|f(n)| \leq c * |g(n)|$  if  $n \geq n_0$  ( $c > 0$ , une constante)

montrer que  $1+2+3\dots+n=O(n^2)$

démonstration :

on sait que  $1+2+3\dots+n=n(n+1)/2$

montrons donc que  $n(n+1)/2=O(n^2)$

c'est à dire montrer que

$n(n+1)/2 = c*n^2$  pour tout  $n \geq n_0$

$n(n+1)/2 = n^2 + n/2 = n^2/2 + n/2$

$\leq n^2/2 + n^2/2$  pour tout  $n \geq 1$  ( $n_0$ )

$\leq 1n^2$  pour tout  $n \geq 1$

$(c=1, n_0=1)$

montrer que  $1^2+2^2+3^2+\dots+n^2=O(n^3)$

(rappel : notation  $O$

$T(n) = O(n^2)$

$T$  : temps

$n$  : la taille d'entrées ou quantité d'entrées

veut dire  $T(n) \leq cn^2$  pour tout  $n \geq n_0$  ( $c > 0$  une constante. Cette constante dépend de la structure de données, du compilateur, de la machine)

$(n \geq n_0$  veut dire à partir d'un certain rang)

il faut montrer que  $1^2+2^2+3^2+\dots+n^2 \leq cn^3$  à partir d'un certain rang

on sait que  $1^2+2^2+3^2+\dots+n^2 = \frac{n(n+1)(2n+1)}{6}$

( rappel :

$$1+2+3\dots+n=n(n+1)/2$$

$$1^2+2^2+3^2+\dots+n^2=\frac{n(n+1)(2n+1)}{6}$$

$$1^3+2^3+3^3+\dots+n^3=\left(\frac{n(n+1)}{2}\right)^2$$

)

Donc il faut prouver que

$$\frac{n(n+1)(2n+1)}{6} \leq cn^3 \text{ pour } n \geq n_0$$

(ou) il faut montrer que  $\frac{n(n+1)(2n+1)}{6n^3} \leq c$ , une constante pour tout  $n \geq n_0$

$$\begin{aligned}
\frac{n(n+1)(2n+1)}{6n^3} &= \frac{(n+1)(2n+1)}{6n^2} = \frac{1}{6} \left( \frac{n+1}{n} \right) \left( \frac{2n+1}{n} \right) \\
&= \frac{1}{6} \left( \frac{n}{n} + \frac{1}{n} \right) \left( \frac{2n}{n} + \frac{1}{n} \right) \\
&= \frac{1}{6} \left( 1 + \frac{1}{n} \right) \left( 2 + \frac{1}{n} \right) \\
&\leq \frac{1}{6} (1+1)(2+1) \\
&\leq \frac{1}{6} * 2 * 3 \\
&\leq 1 \quad (c=1, n \geq 1, n_0=1)
\end{aligned}$$

Exercice : montrer que  
 $1^3 + 2^3 + 3^3 + \dots + n^3 = O(n^4)$   
on sait que  $\left(\frac{n(n+1)}{2}\right)^2$

Exercice : considérons la fonction suivante  
int mystère (int m, int n){

```

    int t;
    if (m<n) t=m;
    else t=n;
    while ((m%t!=0)&&(n%t!=0))
        t=t-1;
    return t;
}
```

Effectuer l'appel de mystère(18,12)  
que fait la fonction ?

S'arrête après 6 itérations, retourne 6  
== pgcd

Considérons l'organigramme suivant :  
(pour trouver le pgcd(m,n))

a=m;

b=n;

a!=b?

Si non : pgcd(a ou b) → stop

si oui : a > b ?

    si oui : a=a-b

    si non : b=b-a

retour au début

==

```

a=m; b=n;
while (a!=b)
    if (a>b) a=a-b;
    else b=b-a;
return a;

```

questions trouver l'invariant (à quel point)

montrer que la boucle while se termine

l'invariant : a l'entrée de la boucle while

$\text{pgcd}(a,b) = \text{pgcd}(a-b,b)$  si  $a > b$

$\text{pgcd}(a,b) = \text{pgcd}(a,b-a)$  si  $b > a$

prouvons  $\text{pgcd}(a,b) = \text{pgcd}(a-b,b)$  si  $a > b$

(l'autre égalité est semblable)

notation :  $D(p,q) =$  l'ensemble de tous les diviseurs positifs de  $p$  et  $q$   
 $(D = \text{diviseur})$

démonstration : soit  $d$  appartient à  $D(a,b)$

alors  $d$  divise  $a$  et  $b$

évidemment,  $d$  divise  $a-b$  et ( $d$  divise  $b$  (hyp))

$d \in D(a-b, b)$

donc  $D(a,b)$  est inclus dans  $D(a-b, b)$  (**I.**)

inversement, soit  $d \in D(a-b,b)$

alors  $d$  divise  $a-b$  et  $b$

donc  $d$  divise la somme  $(a-b)+b=a$  (et divise  $b$  (hyp))

donc  $d \in D(a,b)$

donc  $D(a-b, b)$  est inclus dans  $D(a,b)$  (**II.**)

**I.** et **II.** Veut dire  $D(a,b)=D(a-b, b)$

en particulier,  $\max D(a,b) = \max D(a-b,b)$

$\text{pgcd}(a,b)=\text{pgcd}(a-b, b)$

ex:  $a=18, b=12$

$D(18,12) = \{1, 2, 3, 6\}$

$D(a-b, b)=D(6,12) = \{1, 2, 3, 6\}$

### Invariant, finitude, complexité

(suite)

exemple : oui ou non ? Justifier la réponse.

**1)**  $O(f(n)) - O(f(n)) = 0$

Réponse : non

justification :

soit  $f(n) = n^2$

alors  $2n^2 \leq 2xn^2$  pour tout  $n \geq 1$

$\wedge$

c

$$2n^2 = O(n^2) \quad (\text{I})$$

$$n^2 \leq \underset{\wedge}{(1)} n^2 \quad \text{pour tout } n \geq 1$$

$$\text{donc } n^2 = O(n^2) \quad (\text{II})$$

$$\begin{aligned} \text{donc, } O(n^2) - O(n^2) &= (\text{I}) - (\text{II}) \\ &= 2n^2 - n^2 = n^2 = O(f(n)) \end{aligned}$$

2)  $O(f(n)) - O(f(n)) = ?$

Réponse :  $O_1(f(n)) - O_2(f(n)) = O(f(n))$

Justification : soit  $T_1(n) = O_1(f(n))$   
 $T_2(n) = O_2(f(n))$

$$T_1(n) = O_1(f(n))$$

par définition,  $|T_1(n)| \leq c_1 |f(n)|$  pour tout  $n \geq n_1$  (I)

$$T_2(n) = O_2(f(n))$$

par définition,  $|T_2(n)| \leq c_2 |f(n)|$  pour tout  $n \geq n_2$  (II)

$$|T_1(n) \pm T_2(n)| \leq |T_1(n)| + |T_2(n)|$$

(inégalité du triangle  $|a \pm b| \leq |a| + |b|$ )

(I) et (II) sont vraies à la fois si  $n \geq \max(n_1, n_2)$

$$\begin{aligned} \text{donc pour tout } n \geq \max(n_1, n_2), \quad &|T_1(n)| + |T_2(n)| = |f(n)| (c_1 + c_2) \\ &\leq c |f(n)| + c |f(n)| = 2c |f(n)| \quad (2c = \text{const} > 0) \end{aligned}$$

soit  $c = \max(c_1, c_2)$

donc  $|T_1(n) \pm T_2(n)| \leq 2c |f(n)|$

par définition,  $T_1(n) \pm T_2(n) = O(f(n))$

$\rightarrow O(f(n)) \pm f(n) = O(f(n))$

Règle de la somme :

si  $T_1(n) = O(f(n))$  et

$T_2(n) = O(g(n))$

alors  $T_1(n) + T_2(n) = O(\max(|f(n)|, |g(n)|))$

(costaud mange les petits (plot twist!!!))

ex :  $O(n^2) + O(n)$

$O(\max(n^2, n)) = O(n^2)$

ex :  $O(n) + O(\log n) = O(\max(n, \log n))$   
 $= O(\log n)$

### Règle du produit :

si  $T_1(n) = O(f(n))$  et

$T_2(n) = O(g(n))$

alors  $T_1(n) T_2(n) = O(|f(n)| |g(n)|)$

ex :  $O(n^2) * O(n)$

$$= O(n^2 * n) = O(n^3)$$

ex :  $O(n) * O(\log n) = O(n \log n)$

### Théorème :

SI  $P(n) =$  un polynôme de degré  $k$

$$= a_0 + a_1 n + a_2 n^2 + \dots + a_k n^k$$

alors  $P(n) = O(n^k)$

où  $n =$  la taille d'entrée

$$(a_k \neq 0)$$

démonstration : montrons que, par définition,  $|P(n)| \leq c * n^k$  pour tout  $n \geq n_0$

$$P(n) = a_0 + a_1 n + a_2 n^2 + \dots + a_k n^k$$

$$|P(n)| = |a_0 + a_1 n + a_2 n^2 + \dots + a_k n^k|$$

$$\leq |a_0| + |a_1 n| + |a_2 n^2| + \dots + |a_k n^k|$$

$$\leq |a_0| + |a_1| |n| + |a_2| |n^2| + \dots + |a_k| |n^k|$$

$$\leq n^k \left( n^k * \left( \frac{|a_0|}{n^k} + \frac{|a_1|}{n^{k-1}} + \frac{|a_2|}{n^{k-2}} + \dots + |a_k| \right) \right)$$

$$\leq n^k (|a_0| + |a_1| + \dots + |a_k|)/c \quad \text{pour tout } n \geq 1 (= n_0)$$

donc  $P(n) = O(n^k)$  par définition

ex : montrer que

$$1^3 + 2^3 + 3^3 + \dots + n^3 = O(n^4)$$

on sait que

$$1^3 + 2^3 + 3^3 + \dots + n^3 = \frac{n(n+1)^2}{2} = \frac{n^2(n+1)^2}{4}$$

= un polynôme de degré 4

$$= O(n^4)$$

### Notation $\Omega$ :

la notation  $O$  est pour la borne supérieure  $(\leq)$  (pire des cas)

la notation  $\Omega$  est pour la borne inférieure  $(\geq)$  (meilleur des cas)

la notation  $\theta$  :  $\dots \leq \dots \leq \dots$

def ( $\Omega$ ) :

on écrit  $f(n) = \Omega(g(n))$

si  $|f(n)| \geq c * |g(n)|$

pour tout  $n \geq n_0$

( $c > 0$ , une constante)

montrer que

$$1+2+3+4+\dots+n = \Omega(n^2)$$

on sait que  $1+2+\dots+n = \frac{n(n+1)}{2}$

il faut montrer que  $\frac{n(n+1)}{2} = \Omega(n^2)$

ou  $\frac{n(n+1)}{2} \geq cn^2$  pour tout  $n \geq n_0$

ou  $\frac{n(n+1)}{2n^2} \geq c$

$$\frac{n(n+1)}{2n^2} = 1/2 \left( (n+1)/n \right)$$

$$= 1/2 \left( 1 + 1/n \right)$$

$$\geq 1/2 \left( 1 + 0 \right) \quad (\text{car } 1/n \geq 0 \text{ pour tout } n \geq 1)$$

$$\geq 1/2 c$$

Définition ( $\theta$ ) :

On écrit  $g(n) = \theta(f(n))$  si  $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$  pour  $n \geq n_0$   
( $c_1 > 0, c_2 > 0$ , des constantes)

Remarque :

$g(n) = \theta(f(n))$  si et seulement si

$$- g(n) = O(f(n))$$

$$\text{et} \quad - g(n) = \Omega(f(n))$$

Ex :  $1+2+3+\dots+n = \theta(n^2)$

déjà fait plus haut car  $1+2+3+\dots+n = O(n^2)$

et  $1+2+3+\dots+n = \Omega(n^2)$

Considérons la fonction suivante :

```
void bizarre(int n){  
    int x=0, y=0, i, j;  
    for (i=1 ; i<=n ; I++){  
        if (i%2==1){  
            for (j=i ; j<=n ; j++)  
                x = x+1;  
            for (j=i ; j<=i ; j++)  
                y = y+1;  
        }  
    }  
}
```

Soit A(n) le nombre d'affectations effectuées à la variable x ou y  
Alors trouver A(n) en fonction de n.

Exprimer A(n) en fonction de O,  $\Omega$ , et si possible en  $\theta$  :

la deuxième boucle est effectuée  $n-i+1$  fois

la troisième boucle est effectuée  $i-1+1=i$  fois

la somme :  $(n-i+1)+i = n+1$  affectations pour x ou y

le if veut dire que i est impair : combien d'entiers impairs entre 1 et n ?

$$[\frac{n+1}{2}] \quad (=partie entière)$$

remarque :  $[x] \leq x$

$$\text{donc } A(n) = [\frac{n+1}{2}](n+1)$$

mais  $[x] \leq x$

$$\text{donc } [\frac{n+1}{2}] \leq \frac{n+1}{2}$$

$$\text{donc } A(n) = [\frac{n+1}{2}](n+1)$$

$$\leq (\frac{n+1}{2})(n+1) = \frac{(n+1)^2}{2}$$

= un polynôme de degré 2

$$= O(n^2)$$

Exprimons A(n) avec  $\Omega$

$$\text{donc } A(n) = [\frac{n+1}{2}](n+1)$$

n	$[\frac{n+1}{2}] \geq$	?
1	1	0.5
2	1	1
3	2	1.5
4	2	2
5	3	2.5
6	3	3

$$\geq (n/2)(n+1) = \frac{n(n+1)}{2} = \Omega(n^2) \quad (\text{déjà fait})$$

$$A(n) = O(n^2) \text{ et } A(n) = \Omega(n^2)$$

$$\text{donc } A(n) = \theta(n^2)$$

Écrire une fonction non récursive pour le parcours préfixe d'un AB

$$\text{prefixe}(A) = (1, 2, 3, 6, 4, 8, 5, 7, 9, 10, 13, 12, 11)$$

Les noeuds sont écrits dès la première rencontre

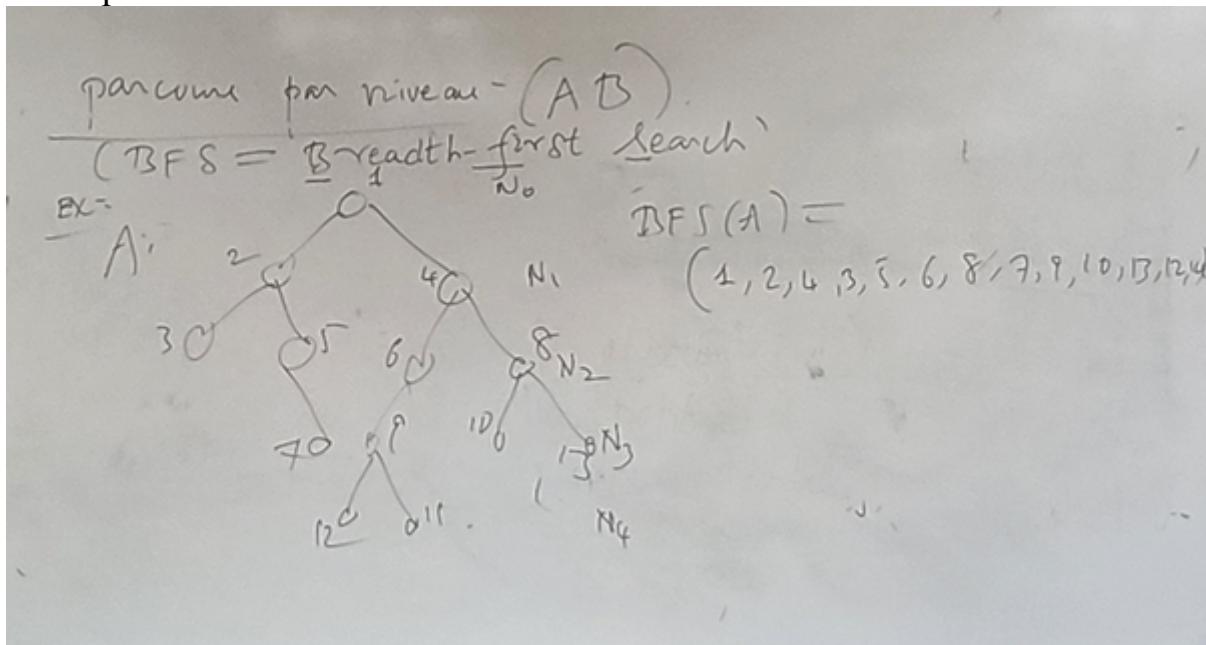
$\text{infixe}(A) = (6, 3, 2, 8, 1, 4?7, 5, 13, 10, 12, 9, 11)$   
 $\text{postfixe}(A) : (6, 3, 8, 4, 3, 7, 13, 12, 10, 11, 9, 5, 1)$

```

//réursive :
void prefixe(lien t){
    if (t != NULL){
        printf(" %d", t->v);
        prefixe(t->g);
        prefixe(t->d);
    }
}

```

parcours par niveau : BFS



### Tableau de complexité :

n=la taille du programme

Algorithmes	complexité
Recherche dichotomique ( <i>divide and conquer</i> )	$O(\log n)$
Recherche séquentielle linéaire	$O(n)$
Quick sort	$O(n \log n)$ (moyenne)
Tri insertion / sélection bulle	$O(n^2)$
Multiplication matricielle $(a_{ij})(b_{ij})$	$O(n^3)$

Tri shell	$\theta(n^{1.5}) = O(\sqrt{n})$
TSP (traveling salesman problem)	Exponentielle $O(2^n)$ (turing prize)

Ex : que fait la fn : trouver sa complexité :

```
int mystere (int t[], int g, int d, int v){
    //t[g] ≤ t[g+1] ≤ ... ≤ t[d]
    while (g<=d){
        int m=(g+d)/2;                      //DIVIDE
        if (t[m]==v) return m;                //v trouvé à l'indice m
        if (t[m]<v)  d=m-1;
        else g=m+1;
    }
    return -1                                //v n'est pas dans t[g...d]
```

}

la fonction cherche v dans  $t[g] \leq t[g+1] \leq \dots t[d]$   
par divide and conquer

l'appel mystère( $t, 0, n-1, v$ ) chercher v dans  $t[0] \leq t[1] \leq \dots t[n-1]$   
complexité : la taille de Pb=n

$$T(n) = \begin{cases} 1 & \text{si } n=1 \quad (\text{la base}) \\ T\left(\frac{n}{2}\right) + 1 & \text{si } n>1 \end{cases} \quad (+1 = \text{coût de division})$$

$$\text{Résolvons } T(n) = T\left(\frac{n}{2}\right) + 1$$

pour faciliter la tâche, soit n une puissance de 2, soit  $n=2^p$ , posons  $n=2^p$  ( $p \geq 1$ )

$$T(2^p) = T(2^{p-1}) + 1 \quad (\text{I})$$

$$\begin{aligned} p &\leftarrow p-1; \\ T(2^{p-1}) &= T(2^{p-2}) + 1 \quad (\text{II}) \end{aligned}$$

En reportant (II) dans (I),

$$\begin{aligned} T(2^p) &= (T(2^{p-1}) + 1) + 1 \\ &= T(2^{p-2}) + 2 \\ &= T(2^{p-3}) + 3 \\ &= T(2^{p-p}) + p = T(1) + p = 1 + p \end{aligned}$$

$$\text{Donc, } T(n) = 1 + P \text{ mais } 2^p = n \iff p = \log_2 n = 1 + \log_2 n$$

$$\begin{aligned} 2^3 = 8 &= O(\log_2 n) \\ \iff 3 &= \log_2 8 \end{aligned}$$

Résoudre la récurrence  $T(0) = 0$  si  $n=0$       (la base)  
 $T(n) = 2T(n-1) + 1$     si  $n \geq 1$

$$\begin{aligned} T(n)+1 &= 2T(n-1)+1+1 \\ T(n)+1 &= 2(T(n-1)+1) \end{aligned} \quad (\text{I})$$

posons  $u(n)=T(n)+1$   
 $u(n-1)=T(n-1)+1$   
devient  $u(n)=2u(n-1)$  (II)  
 $n \leq n-1,$   
 $u(n-1)=2u(n-2)$  (III)

en reportant III dans II :

$$\begin{aligned} u(n) &= 2 * 2 * u(n-2) = 2^2 u(n-2) \\ u(n) &= 2^2 u(n-2) \\ u(n) &= 2^3 u(n-3) \text{ de même} \\ &= 2^4 u(n-4) \\ &= 2^n u(n-n) = 2^n u(0) = 2^n * 1 = 2^n \end{aligned}$$

$$\begin{aligned} u(n) &= 2^n \\ T(n)+1 &= 2^n \\ T(n) &= 2^n - 1 \end{aligned} \quad (\text{tour d'hanoi})$$

Résoudre : la récurrence

(Qs le plus favorable)

$$T(n) = \begin{cases} 1 & \text{si } n=1 \\ 2T(n/2)+n & \text{si } n>1 \end{cases} \quad (\text{la base})$$

$n$  = la taille du problème

pour faciliter, posons  $n=2^p$

$$T(2^p) = 2T(2^{p-1}) + 2^p$$

divisons par  $2^p$

$$T(2^p)/2^p = (T(2^{p-1})/2^{p-1}) + 1$$

$$\text{posons } u_p = T(2^p)/2^p$$

$$u_p = u_{p-1} + 1$$

$$p \leftarrow p=1,$$

$$u_{p-1} = u_{p-2} + 1 \quad (\text{II})$$

en reportant II dans I :

$$u_p = (u_{p-2} + ?) + ? = u_{p-2} + 2$$

de même

$$u_p = u_{p-3} + 3$$

$$u_p = u_{p-p} + p = u_0 + p$$

$$\text{mais } u_0 = (T(2^0)/2^0) = T(1)/1 = 1$$

$$\text{donc } u_p = 1 + p$$

$$\text{mais } 2^p = n$$

$$\text{donc } p = \log_2 n$$

$$u(p) = (T(2^p)/2^p) = 1 + \log_2 n \text{ c à d } (T(n)/n) = 1 + \log_2 n,$$

$$T(n) = n(1 + \log_2 n)$$

$$= n + n \log_2 n$$

$$= O(n \log_2 n)$$

Résoudre :

$$T(n) = \begin{cases} 1 & \text{si } n=1 \\ 2T(n/2) + 1 & \text{si } n>1 \end{cases}$$

(la base)  
(la récursion)

posons  $n=2^p$

$$T(2^p) = 2T(2^{p-1}) + 1$$

$$T(2^p) = 2T(2^{p-1}) + 1 \quad (I)$$

$p \leftarrow p-1$  (on remplace  $p$  par  $p-1$ )

$$T(2^{p-1}) = 2T(2^{p-2}) + 1 \quad (II)$$

en reportant II dans I

$$\begin{aligned} T(2^p) &= 2(2T(2^{p-2}) + 1) + 1 \\ &= 2^2T(2^{p-2}) + 2 + 1 \\ &= 2^3T(2^{p-3}) + 2^2 + 2 + 1 \end{aligned} \quad (III)$$

...

$$= 2^pT(2^{p-p}) + (2^{p-1} + 2^{p-2} + \dots + 2 + 1) \quad \text{serie géométrique}$$

$$= 2^pT(1) + 2^p - 1$$

$$= 2^p * 1 + 2^p - 1$$

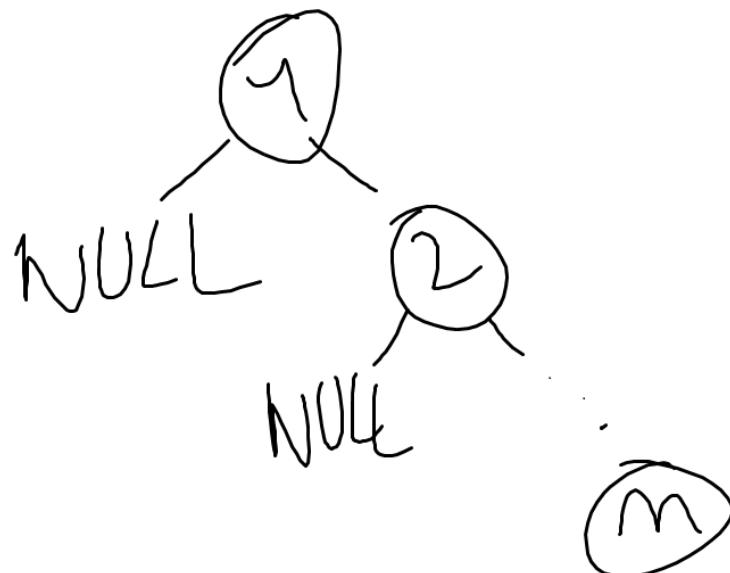
$$= n + n - 1 = 2n - 1 = O(n)$$

(formule :  $1+r+r^2+\dots+r^{n-1}=r^n-1/r-1$ )

### La complexité moyenne de la recherche et l'insertion dans un ABR

Nous allons prouver que la complexité moyenne est de  $O(\log_2 n)$  où  $n$  = taille du problème = le nombre de noeuds de l'arbre

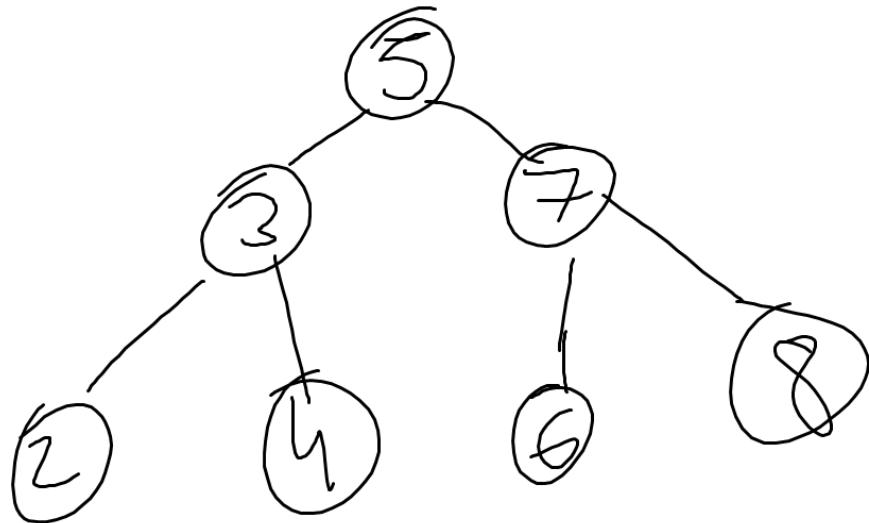
On lit les entrées 1,2,3..., n dans un ordre "au hasard". c'est à dire toutes les  $n!$  permutations peuvent être lues comme entrée (la complexité dans le pire des cas)



La hauteur de cet arbre :  $n = O(n)$

(Le meilleur des cas :

n=8



Un arbre Binaire complet (quasi-complet)

Nous avons déjà montré que la hauteur d'un ABR complet ou quasi-complet =  $O(\log n)$

)

revenons à la complexité moyenne

Soit  $a_n$  la complexité moyenne de la recherche et de l'insertion d'un ABR

Rappel :

$x_1$	$x_2$	$x_3$	...	...	$x_n$
$p_1$	$p_2$	$p_3$	...	...	$p_n$

$$\sum p_i = 1$$

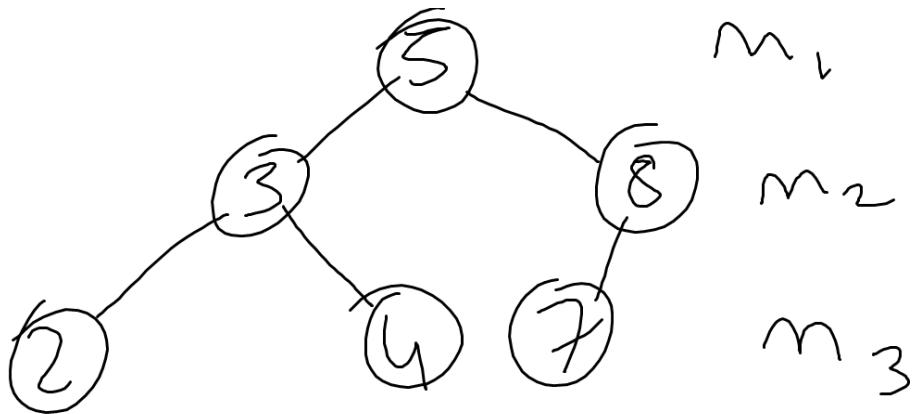
$$E(X) = \text{espérance mathématiques de var } A = \sum x_i p_i$$

$a_n$  = la hauteur moyenne de ABR

= la somme des produits de niveau d'un nœud et la probabilité d'accès à ce nœud

$$= 1/n \sum p_i \text{ où } p_i = \text{la longueur d'un chemin de la racine au nœud } i$$

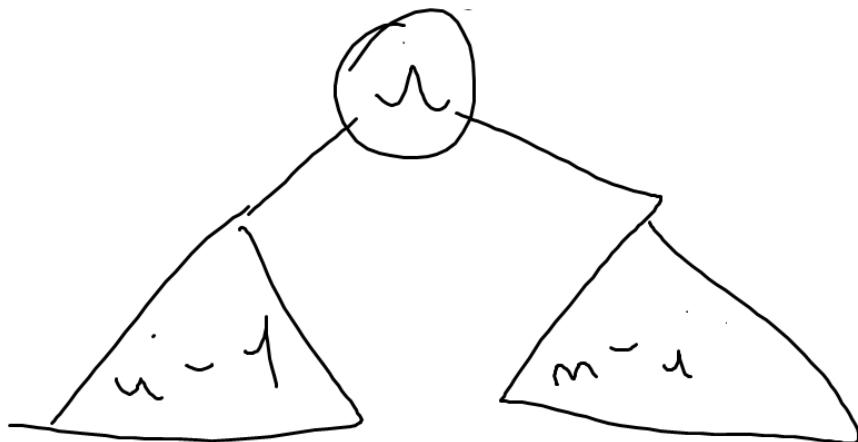
La probabilité que le premier entier lu est  $i$  ( $1 \leq i \leq n$ ) =  $1/n$  (équiprobable)  
c'est la racine de l'ABR



$$n = 6$$

$$1 * 1/6 + 2 * 1/6 + 2 * 1/6 + 3 * 1/6 + 3 * 1/6 + 3 * 1/6$$

équiprobable



$$1 + (i-1) + (n-i) = n$$

(i fixe comme racine)

$$\begin{aligned} a_n &= \text{la hauteur moyenne de l'ABR en fixant } i \text{ pour la racine} \\ &= 1 * 1/n + (a_{i-1} + 1)((i-1)/n) + (a_{n-i} + 1)((n-i)/n) \end{aligned}$$

En faisant varier la racine i,

( $\Sigma$  = somme de  $i=1$  à  $n$ )

$$\begin{aligned} a_n &= 1/n \sum a_n^i \\ &= 1/n \sum (1 * 1/n + (a_{i-1} + 1)((i-1)/n) + (a_{n-i} + 1)((n-i)/n)) \\ &= 1/n^2 \sum (1 + (a_{i-1} + 1)(i-1) + (a_{n-i} + 1)(n-i)) \\ &= 1/n^2 \sum (1 + (i-1)a_{i-1} + (i-1) + (n-i)a_{n-i} + n-i) \\ &= 1/n^2 \sum (n + (i-1)a_{i-1} + (n-i)a_{n-i}) \\ &= 1/n^2 \sum n (= n^2) + 1/n^2 \sum ((i-1)a_{i-1} + (n-i)a_{n-i}) \\ &= 1/n^2 * n^2 + 1/n^2 (0*a_0 + 1*a_1 + 2*a_2 + \dots + (n-1)a_{n-1} + (n-1)a_{n-1} + (n-2)a_{n-2} + 1*a_1 + 0*a_0) \\ &= i + 2/n^2 \sum (\text{somme de } i=1 \text{ à } n-1) ia_i \end{aligned}$$

La complexité moyenne de la recherche et insertion dans un arbre binaire de recherche (ABR) avec  $n$  nœuds  $1 \leq i \leq n$   
c'est l'ABR avec  $i$  pour racine

notation :

$a_n$  = la complexité moyenne  
(average=moyenne,  $n$  = nombre de nœuds )

$a_n$  = la somme de produits de niveau de chaque nœud et la probabilité de son accès  
(on suppose équiprobabilité)

$$= 1/n \sum_{i=1}^n p_i$$

où  $p_i$  = path = la longueur de chemin depuis la racine vers le nœud  $i$

L'arbre avec la racine  $i$  fixée

$$a_n^{(i)} = 1/n(1*1/n + (a_{i-1}+1)((i-1)/n)+(a_{n-i}+1)((n-i)/n))$$

Faisant varier

$$\begin{aligned} a_n &= 1/n \sum a_n^{(i)} = 1/n \sum ((1*1/n + (a_{i-1}+1)((i-1)/n)+(a_{n-i}+1)((n-i)/n))) \\ &= 1+2/n^2 \sum_{i=1}^{n-1} i * a_i \end{aligned} \quad (I)$$

(I) exprime  $a_n$  en fonction de  $a_1, a_2, \dots, a_{n-1}$

Nous voulons exprimer  $a_n$  en fonction de  $a_{n-1}$

$$(I) a_n = 1 + (2/n^2) * (n-1) * a_{n-1} + 2/n^2 \sum_{i=1}^{n-2} i * a_i \quad (II)$$

$n \leftarrow n-1$  dans (I)

$$a_{n-1} = 1 + 2/(n-1)^2 \sum_{i=1}^{n-2} i * a_i$$

$$a_{n-1} - 1 = 2/(n-1)^2 \sum_{i=1}^{n-2} i * a_i \quad \text{multiplions par } (n-1)^2/n^2$$

$$(a_{n-1}-1) * ((n-1)^2/n^2) = 2/n^2 \sum_{i=1}^{n-2} i * a_i$$

en reportant  $2/n^2 \sum_{i=1}^{n-2} i * a_i = (a_{n-1}-1) * ((n-1)^2/n^2)$  dans (II), on obtient

$$a_n = 1 + 2/n^2(n-1)a_{n-1} + (a_{n-1}-1)((n-1)^2/n^2)$$

$$a_n = 1/n^2 (n^2 + 2(n-1)a_{n-1} + (a_{n-1}-1)(n-1)^2)$$

$$= 1/n^2 (n^2 + 2(n-1)a_{n-1} + a_{n-1}(n-1)^2 - (n-1)^2)$$

$$= 1/n^2 (a_{n-1}(2(n-1) + (n-1)^2) + n^2 - (n-1)^2)$$

$$= 1/n^2 ((n^2 - 1)a_{n-1} + 2n - 1) = a_n$$

nous avons exprimé  $a_n$  en fonction de  $a_{n-1}$

$$\begin{aligned} (\text{ex : } n! &= n * (n-1)! \\ &\quad n * (n-1) * (n-2) * \dots * 2 * 1) \end{aligned}$$

exprimons  $a_n$  uniquement en fonction de  $n$

$$a_n = 2 * ((n+1)/n) * h_n - 3$$

où  $h_n$  = le nombre harmonique

$$= 1 + 1/2 + 1/3 \dots + 1/n = (\text{somme de } x=1 \text{ à } n) 1/x \sim \text{intégrale de } 1/x \text{ from 1 to } n$$

$$=[\ln]_1^n = \ln n - \ln 1 = \ln n = h_n$$

$$a_n \sim 2*(n+1)/n * \ln n - 3$$

$$\sim 2(1 + 1/n) \ln n - 3$$

$$\sim 2 \ln n + 1/n * \ln n - 3 = O(\ln n) = O(\log n)$$

### Arbre AVL (Adelson-Velski)

Un ABR est AVL si la différence des hauteurs de l'arbre gauche et droit diffère de au plus 1 pour chaque nœud.

Un arbre parfaitement équilibré est AVL, l'inverse est faux.

### Arbre Fibonacci

(rappel :  $f_h = f_{h-1} + f_{h-2}$

définition : 1. arbre vide est un arbre Fibonacci de hauteur 0.

2. un seul nœud est un arbre Fibonacci de hauteur 1

3. (Récursion)

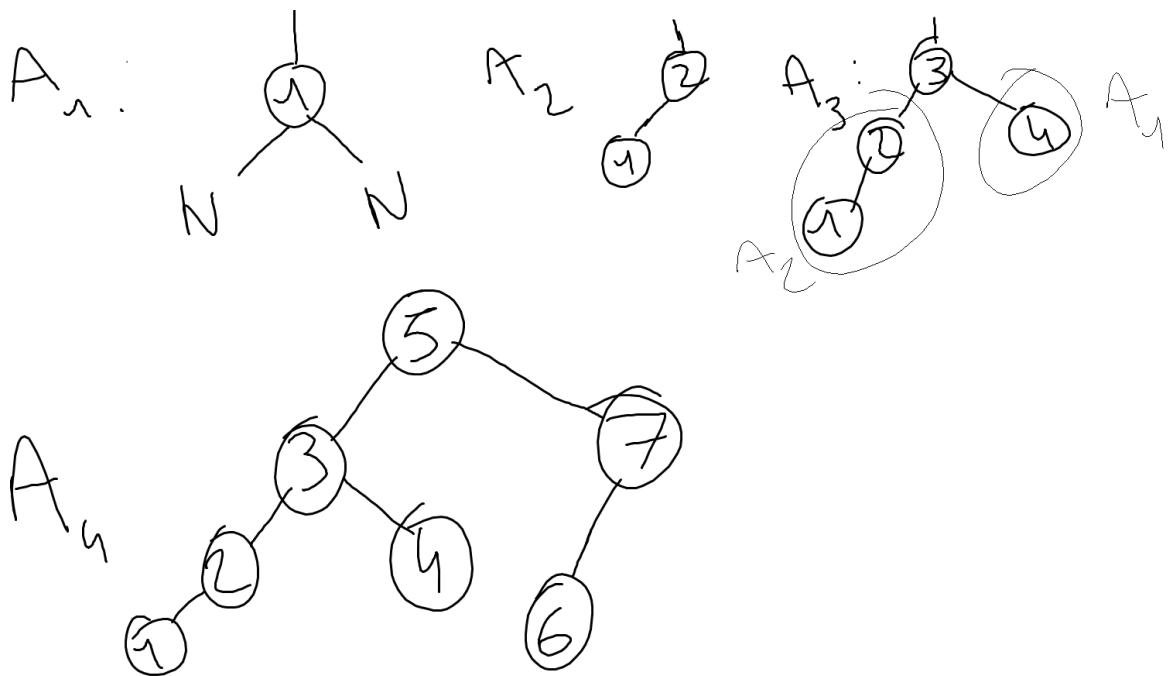
si  $A_{h-1}$  et  $A_{h-2}$  sont les arbres Fibonacci de hauteur  $h-1$  et  $h-2$  respectivement, alors :  $A_h = (x; A_{h-1}, A_{h-2})$  est un arbre Fibonacci de hauteur  $h$ .



4. Aucun autre arbre est un arbre Fibonacci

ex :  $A_0 = \text{NULL}$ ,

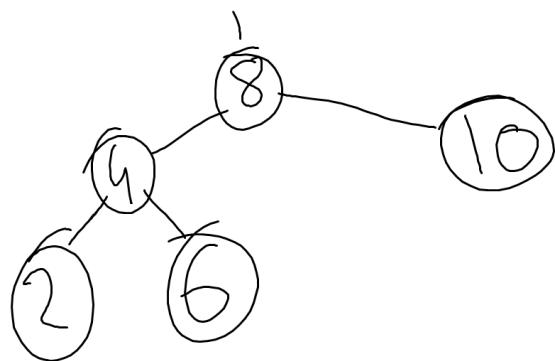
(on ne fait pas attention ici au contenu : on veut juste un ABR, c'est la hauteur qui est fibo)



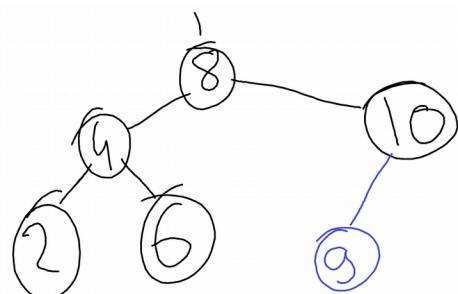
Notons que par définition, un arbre Fibonacci est un arbre AVL.

### Arbre AVL (Insertion-Suppression)

Ex : un AVL

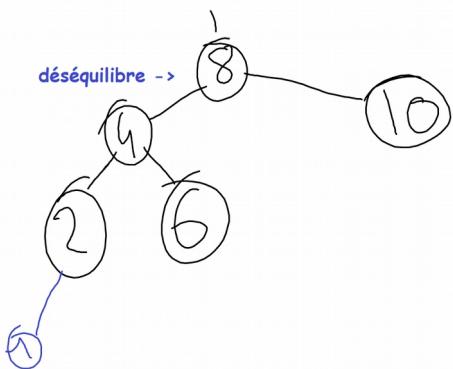


ajoutons 9



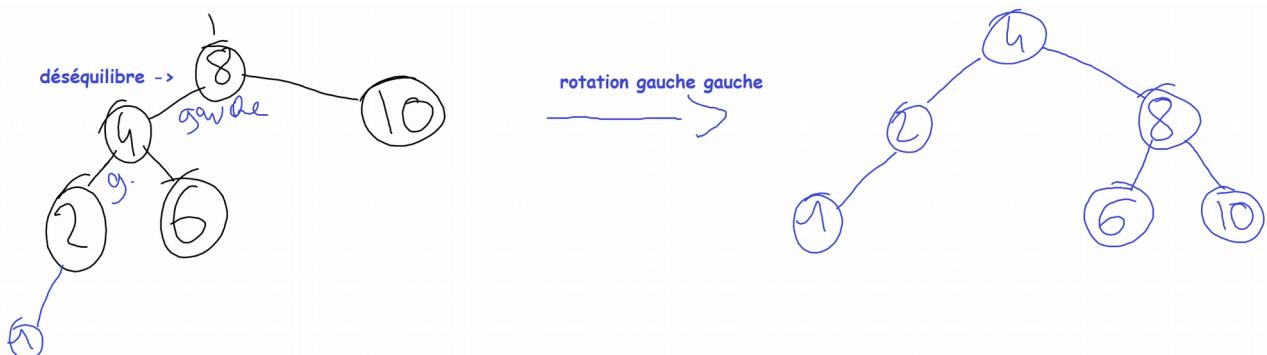
l'arbre reste AVL.

Reprendons l'arbre de départ et ajoutons 1.



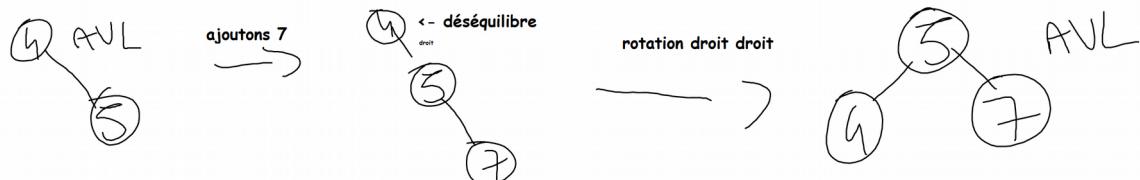
Restructurons l'arbre pour qu'il soit AVL.

Pour cela, cherchons le 1<sup>er</sup> nœud déséquilibré en remontant depuis le nœud inséré. Ici, c'est le nœud 8

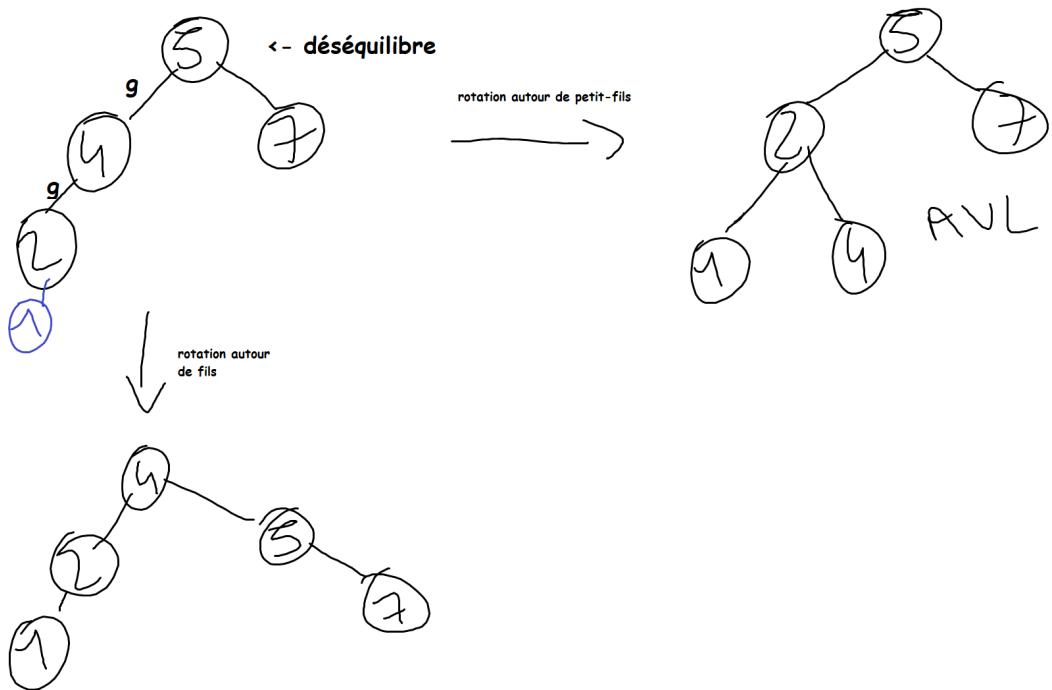


l'arbre obtenu est un AVL

Exemple :

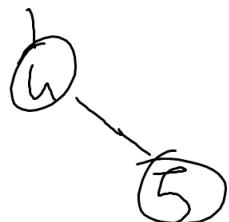


Exemple :



Double rotation : gauche droit ou droit gauche  $\rightarrow$  rotation autour de **petit fils** ( $\varepsilon^2$ )  
 simple rotation : gauche gauche ou droit droit  $\rightarrow$  rotation autour de **fils** ( $\varepsilon$ )

Arbre AVL :



$Ag = \text{NULL}$ ,  $Ad = 5$

c'est un arbre AVL.

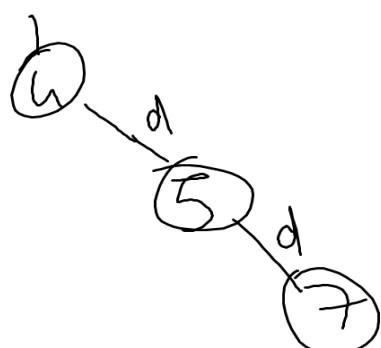
(Arbre vide : hauteur = 0)

Arbre d'un seul nœud : hauteur = 1)

$h(Ag) = 0$ ,  $h(Ad) = 1$

La différence :  $1 - 0 \leq 1$

Insérons 7



Non AVL. Déséquilibre à 4

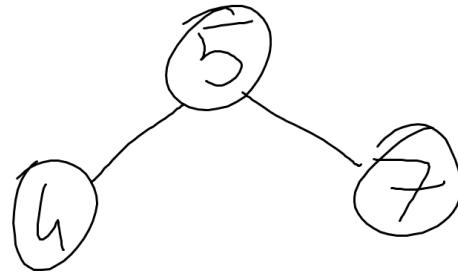
Ag=NULL :  $h(Ag)=0$

Ad = 5, 7 :  $h(Ad)=2$

Différence 2>1

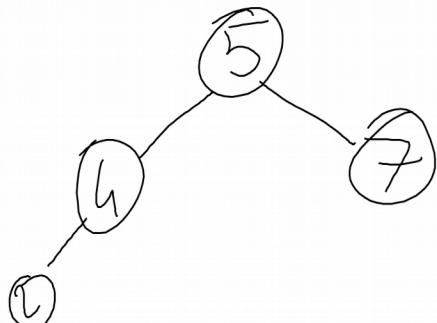
droit droit donc rotation autour de fils (5)

pour restructurer



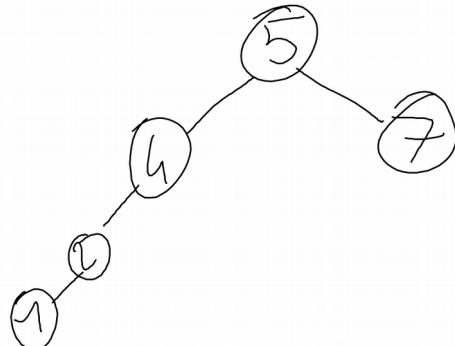
AVL

Insérons 2



AVL

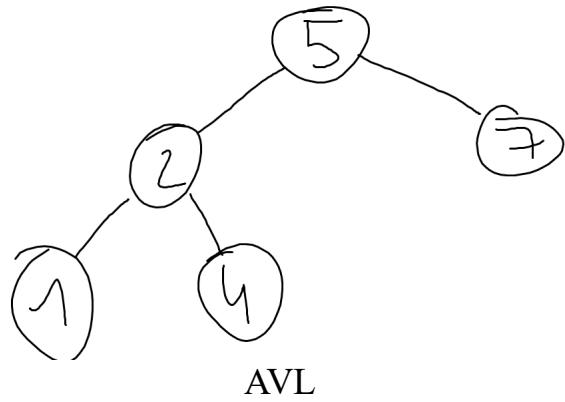
Insérons 1



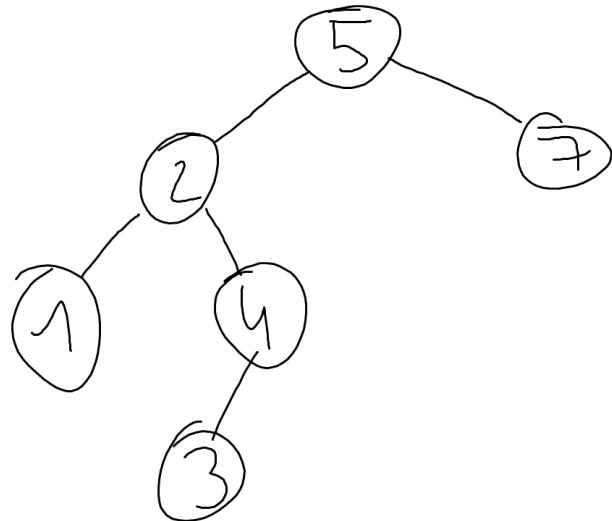
non AVL

déséquilibre à 4 (from en bas to en haut, le premier n'est pas équilibré)

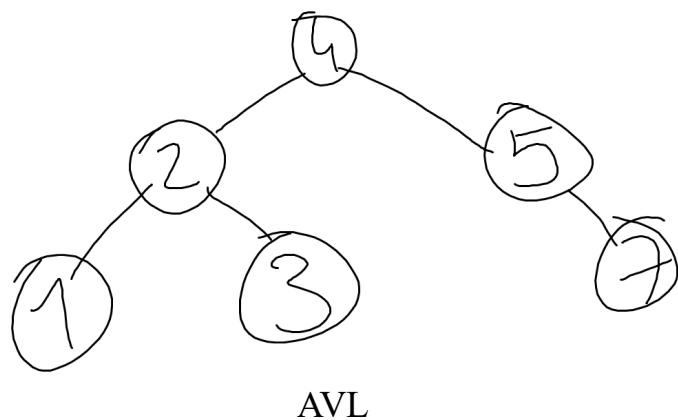
gauche gauche donc rotation autour du fils 2



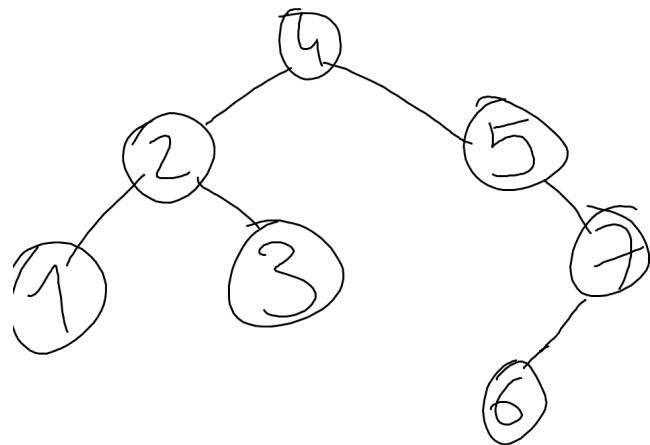
insérons 3



gauche droit donc rotation autour de petit fils



Insérons 6



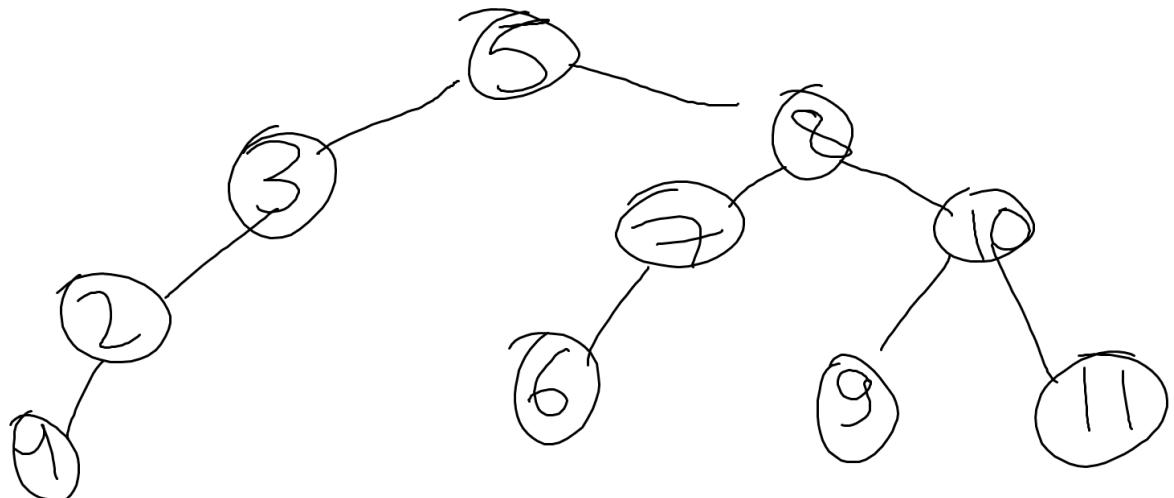
$$h(Ag(5)) = h(\text{NULL}) = 0$$

$$h(Ad(5)) = h(7, 6) = 2$$

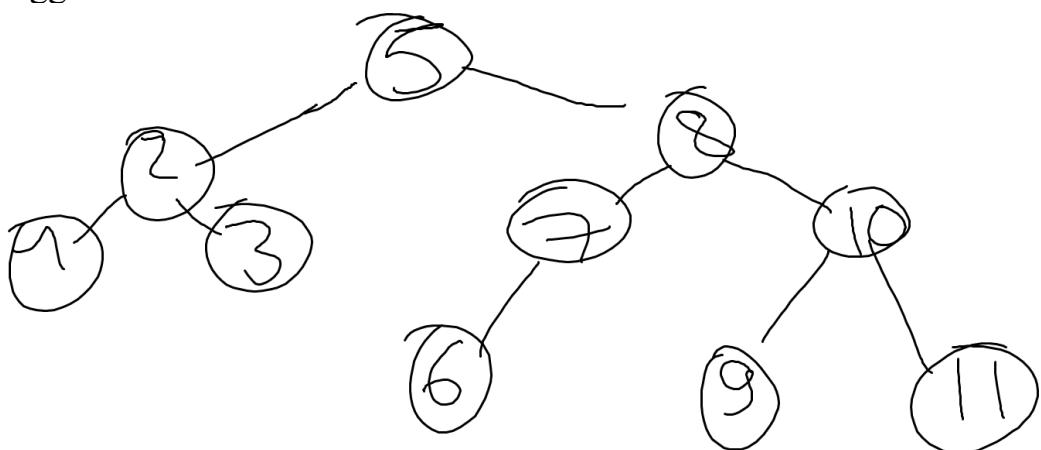
non AVL

droit gauche donc rotation autour de petit fils qui devient la racine

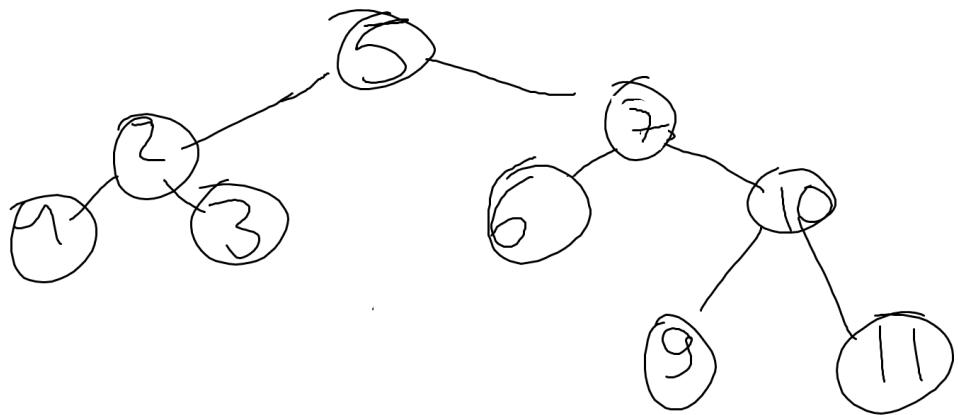
SUPPRESSION :



rotation gg autour de 2

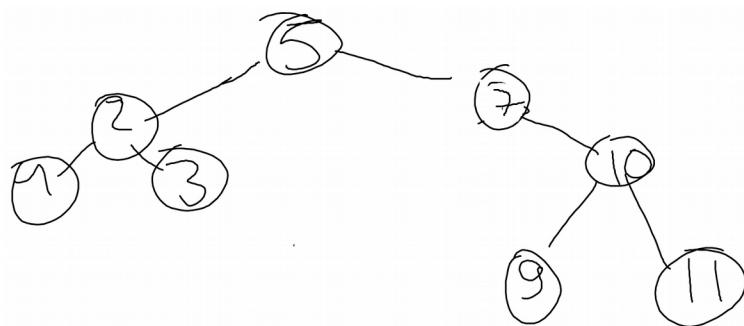


on suppr 8



AVL

supprimons 6



non avl, déséquilibre à 7

rotation dd autour de fils 10 de 7

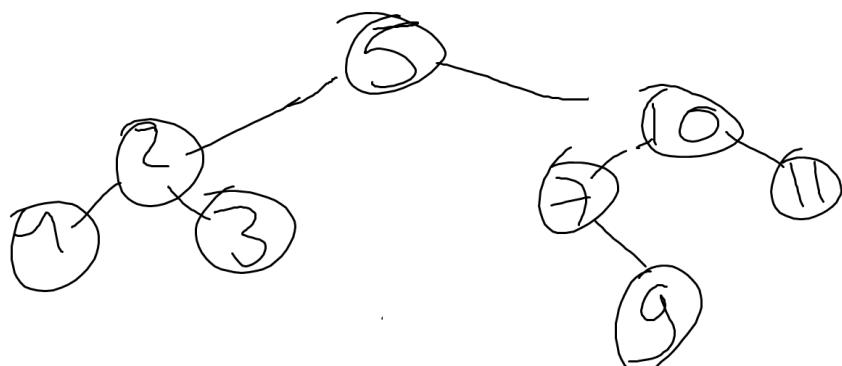
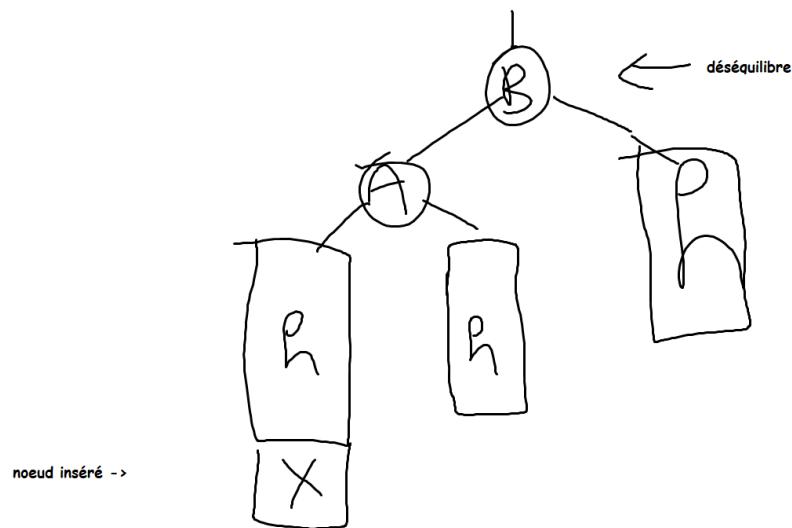


Schéma général pour rotation gauche gauche :



sous arbres sont noté par des rectangles, le nœud inséré par croix dans rectangle simple rotation gauche gauche autour de fils (A)

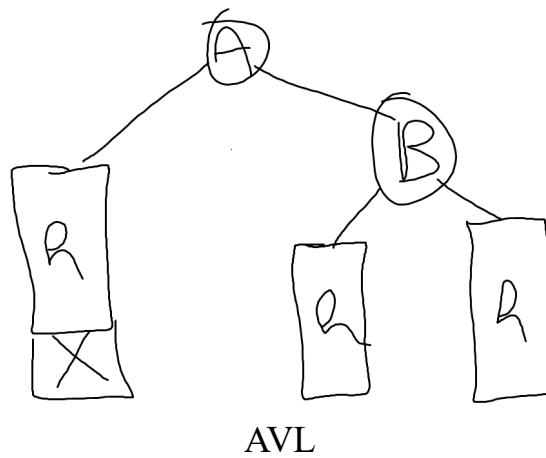
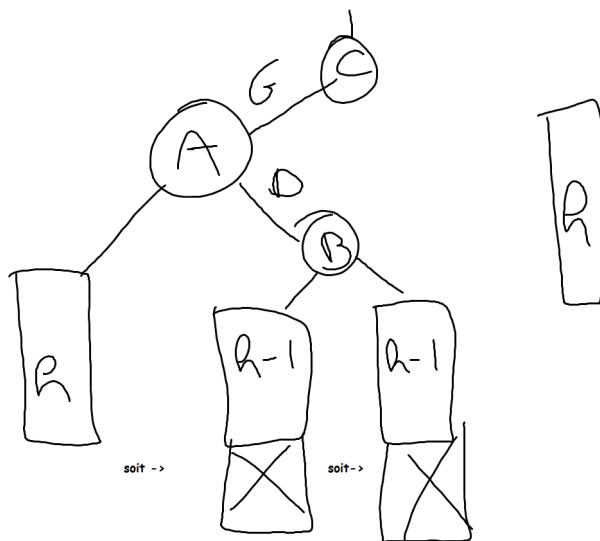
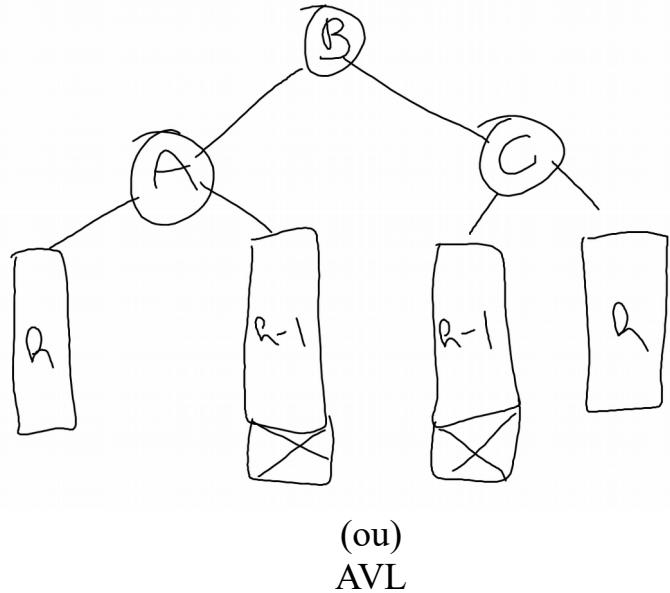


Schéma double rotation GD  
A < B < C



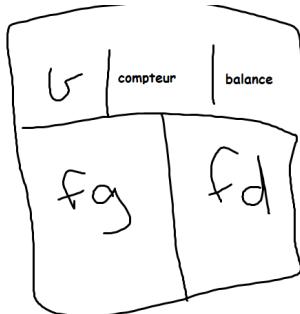
rotation autour de petit fils (B)



Remarque :

Notons que les seuls mouvements des sous\_arbres permis sont des mouvements verticaux tandis que les positions horizontales relatives des sous-arbres et les nœuds A, B, C doivent être conservés.

On enrichit la déclaration de nœud :

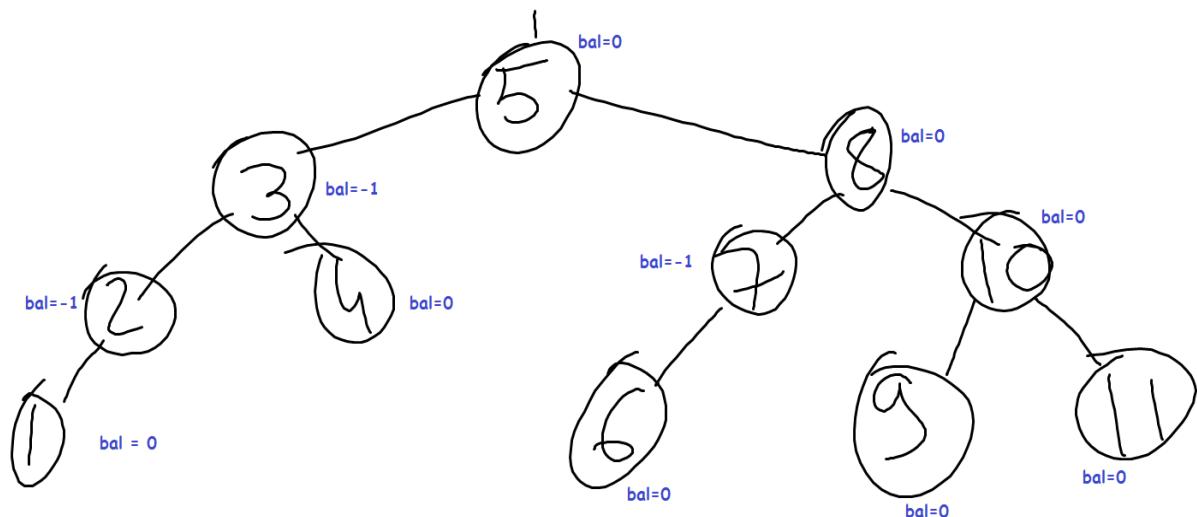


balance = différence des hauteurs de l'arbre droit-arbre gauche.

Appartient à {-1, 0, +1}

```
struct noeud{
    int v;
    int compteur, bal;
    struct noeud *fg, *fd;
};
```

Exemple :



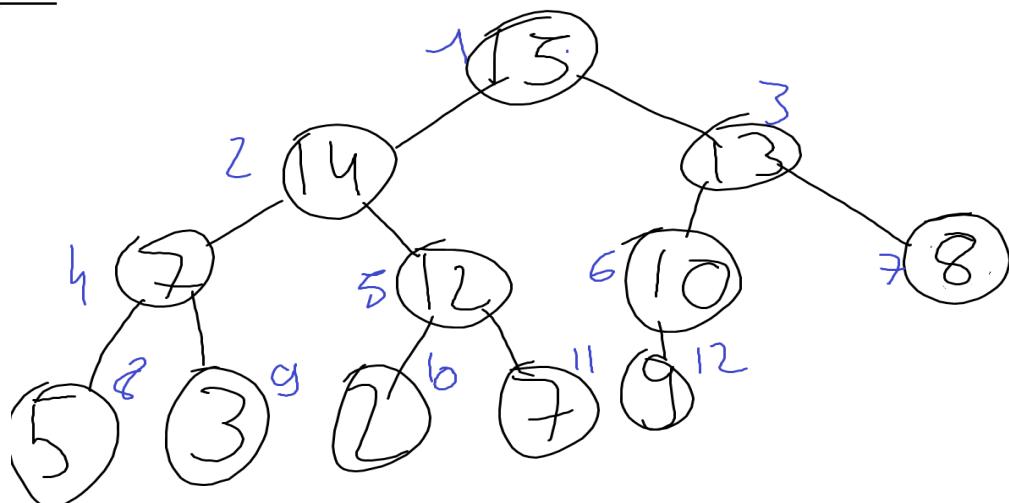
un arbre est AVL si et seulement si la balance de chaque nœud appartient à {-1, 0, +1}

Pour restructurer un arbre non AVL en un arbre AVL, il faut non seulement effectuer les rotations (GG/DD/GD/DG), mais aussi corriger le facteur de balance en remontant dans l'arbre à partir du nœud inséré.

### Tri par tas :

Un tas (*heap*) est un arbre binaire **quasi-complet** tel que la racine contient la valeur **maximum**.

Exemple :

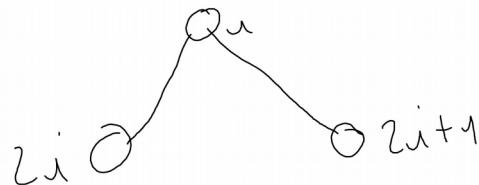


i	1	2	3	4	5	6	7	8	9	10	11	12
t	15	14	13	7	12	10	8	5	3	2	7	9

Un tas est un tableau  $t[1 \dots n]$  tel que  
 $t[i] \geq t[2i]$   
 $t[i] \geq t[2i+1]$  pour tout  $i$  défini  
 $(2i+1 \leq n)$

Notons que le parent du nœud avec l'indice  $i$  = partie entière de  $\frac{i}{2}$

Autrement dit, les enfants du nœud avec l'indice  $i$  (s'ils existent) sont  $2i$  ou  $2i+1$ .



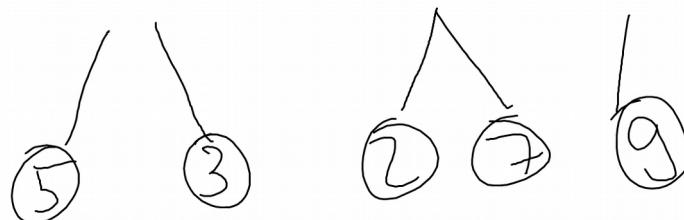
### Tri par tas :

Exemple 1 :

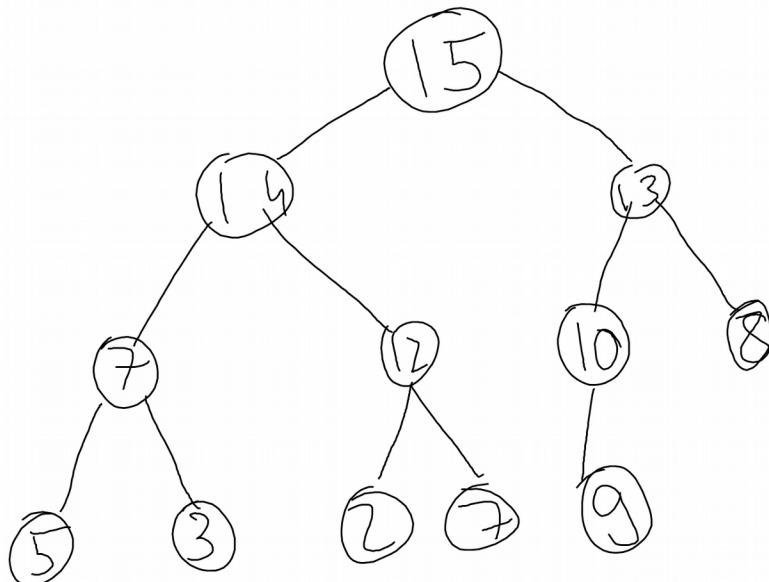
Trier le tableau  $t=(15, 14, 13, 7, 12, 10, 8, 5, 3, 2, 7, 9)$

$n=12$  entiers

partie entière de  $n/2 = [n/2] = 6$



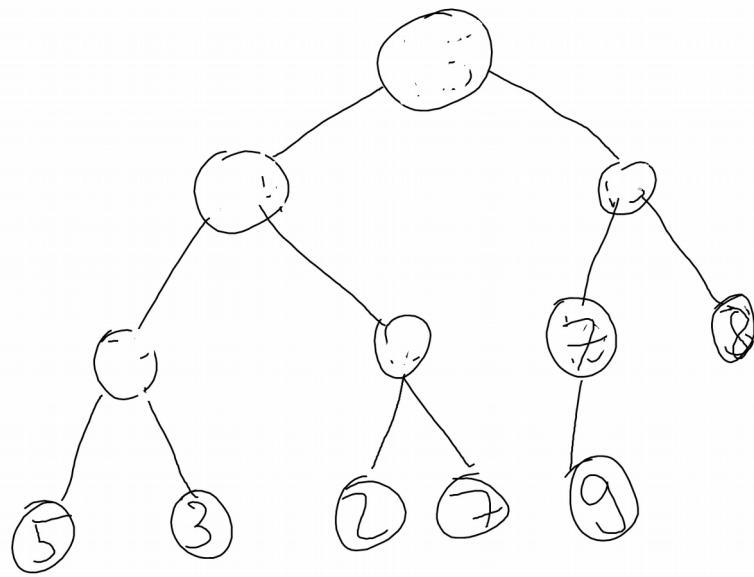
notons que  $t[\frac{n}{2} + 1 \dots]$  forme déjà un tas. On a placé  $\{8, 5, 3, 2, 7, 9\}$



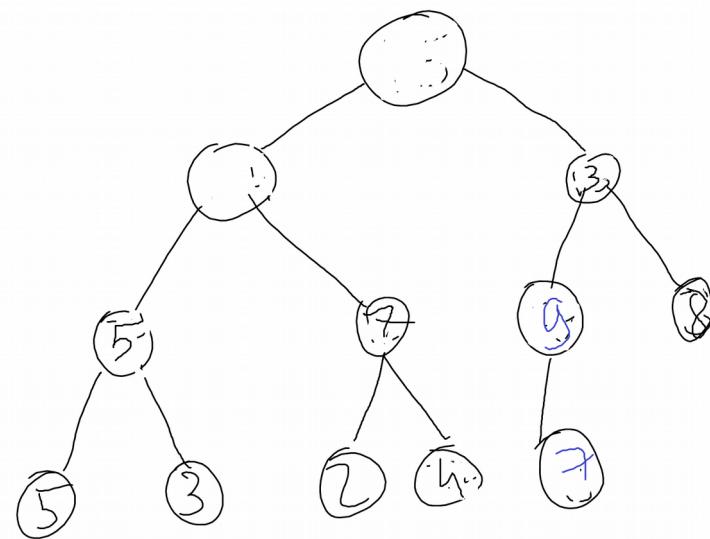
Exemple 2 :

Trier le tableau  $t=(2, 1, 3, 5, 8, 5, 3, 2, 7, 9)$

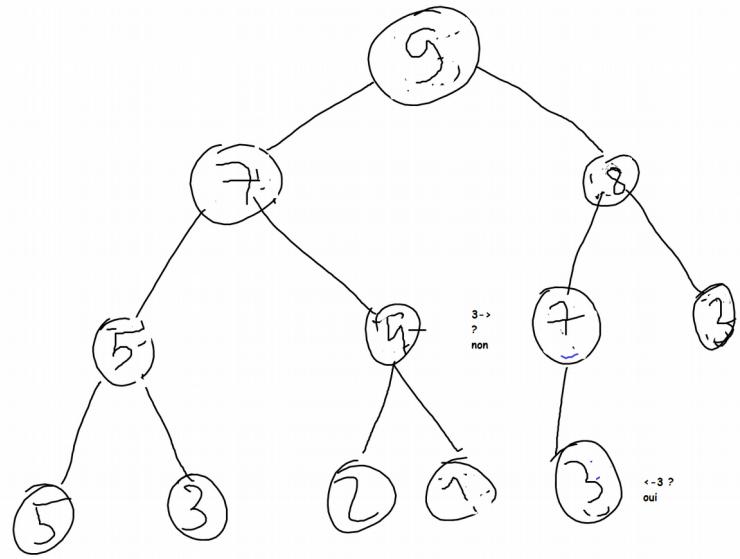
Phase de construction :



pas un tas. Donc restructurons



pareil. Le 3 n'est pas à sa place



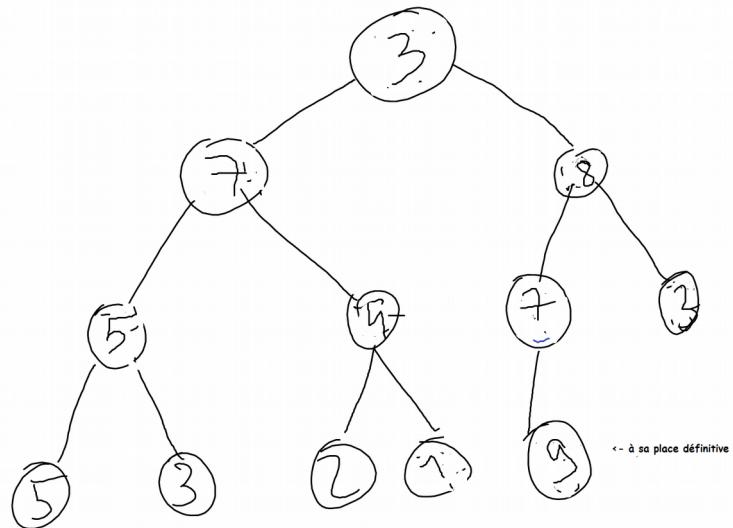
$t$  est devenu  $t(9, 7, 8, 5, 4, 7, 2, 5, 3, 2, 1, 3)$

donc, pour construire un tas :

1. on place les derniers éléments en feuilles
2. on place le reste (en partant de la fin) en allant vers le haut, si ça marche pas on échange. Si 1 fils on échange avec celui là, si 2 fils on échange avec le plus gros

Phase de tri :

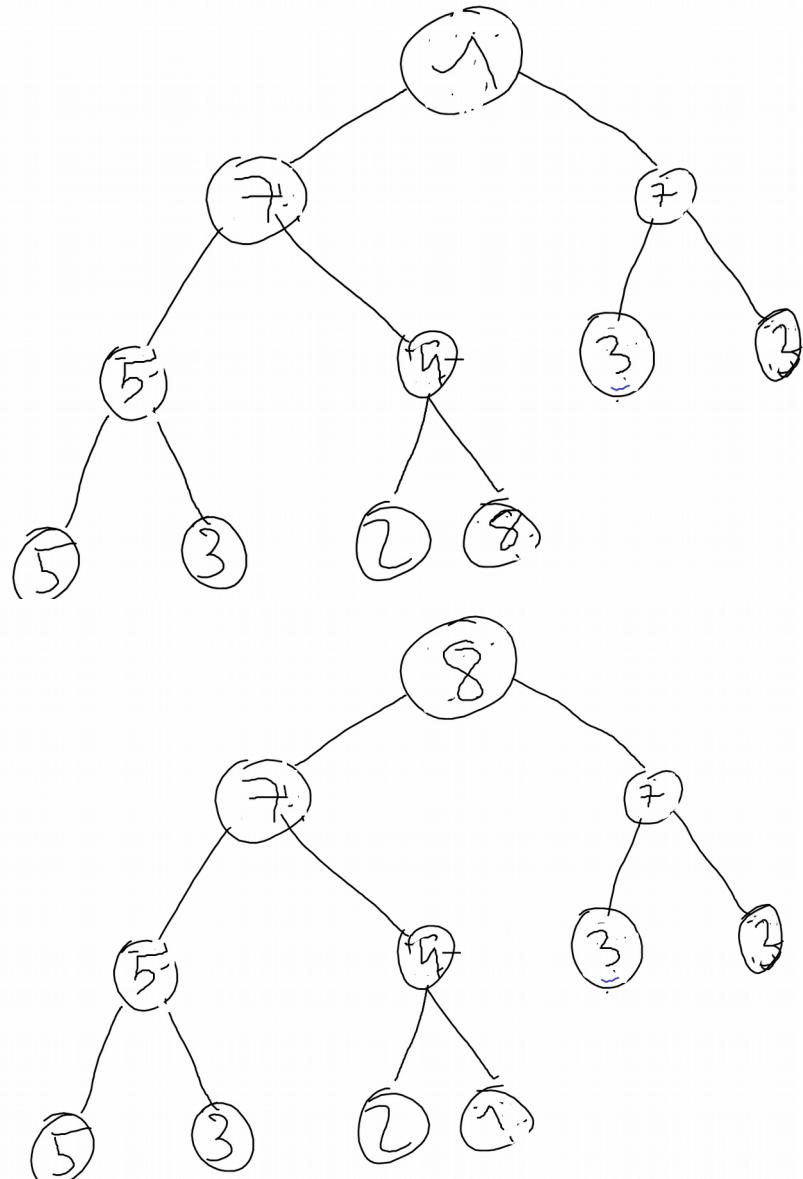
Echanger  $t[1]$  et  $t[12]$



9 est à sa place, donc on l'enlève :  $t(3, 7, 8, 5, 4, 7, 2, 5, 3, 2, 1, 9)$

ce n'est plus un tas, donc on échange

On échange  $t[1]$  et  $t[11]$



8 est à sa place, on l'enlève (ou déconnecte, enlève le pointeur. À l'examen mettre une croix sur pointeur).

Ce n'est plus un tas, donc on échange...

On fait ceci jusqu'à arriver à un seul élément.

#### La complexité de tri par tas :

La hauteur d'un arbre binaire quasi-complet de  $n$  nœuds =  $\log_2 n$

La construction d'un tas demande  $\frac{n}{2} \log_2 n = O(n \log_2 n)$

La phase de tri demande  $O(n \log_2 n)$  (on échange  $n$  fois, et hauteur un  $\log_2 n$ )

$$T(n) = O(n \log n) + O(n \log n)$$

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

Effectuer tri par tas  $t=(2, 1, 7, 3, 5, 6, 2, 1)$