VARI 3
STRUCTURES ARBORESCENTES

PLAN

- **Arbres binaires**
- **Arbres généraux**
- **Files de priorité et Tas**

3.1 ARBRES BINAIRES

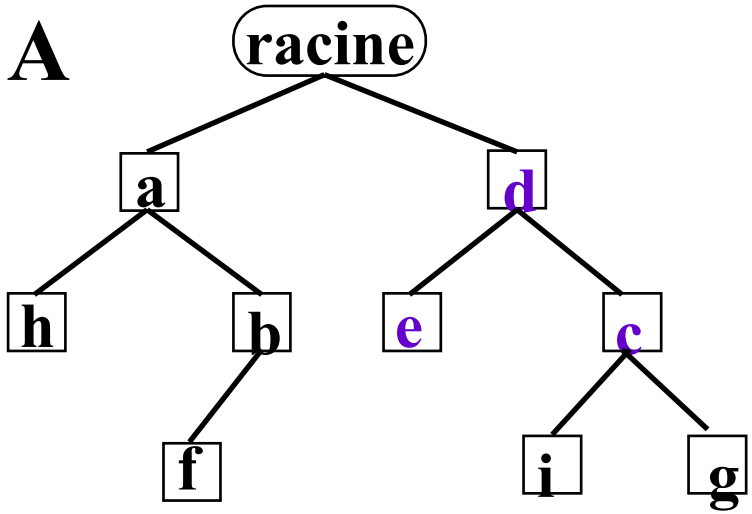
"arbre" en informatique =

"arbre enraciné" (rooted tree)

= arborescence en théorie des graphes

Ex: arbre généalogique, tournois, arbre des espèces animales, …

arborescence binaire: chaque sommet ou nœud a au plus 2 successeurs ou fils dont il est le père



sens

implicite

(d père de e et c)

définition récursive d'un arbre

ensemble vide

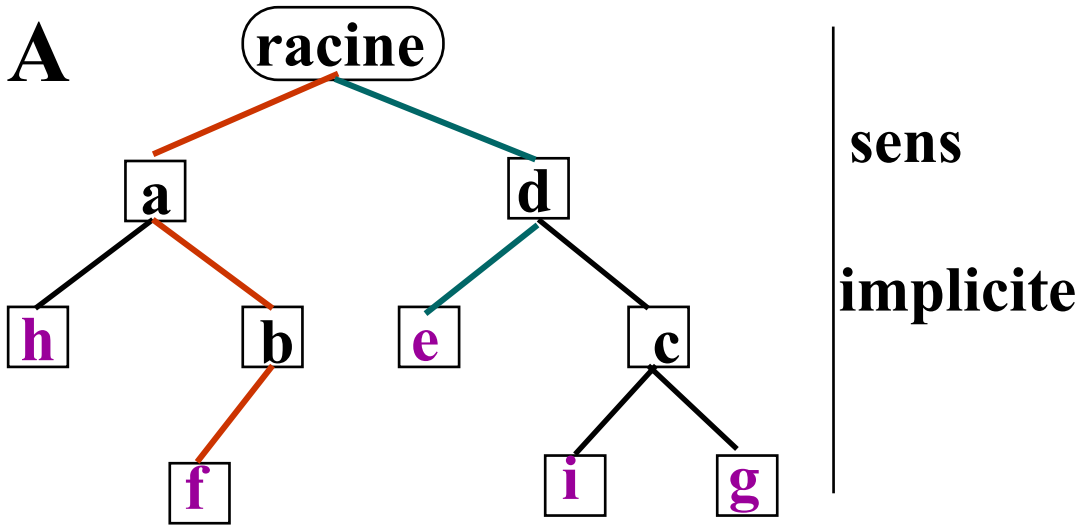
ou

ensemble formé

d'une racine et

d'un sous-arbre droit et

d'un sous arbre gauche

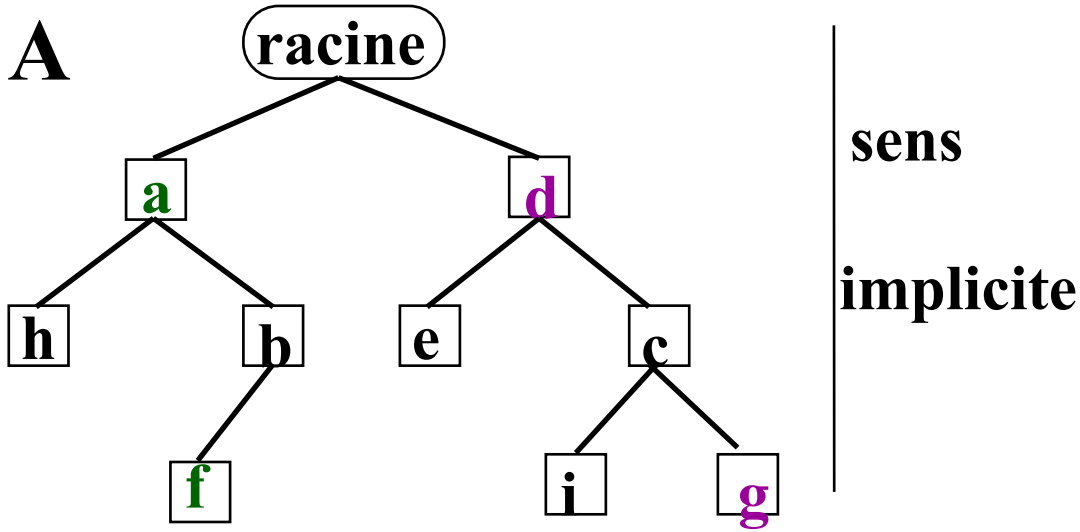


feuille: nœud sans fils (h,f,e,i,g)

branche: chemin de la racine à une feuille

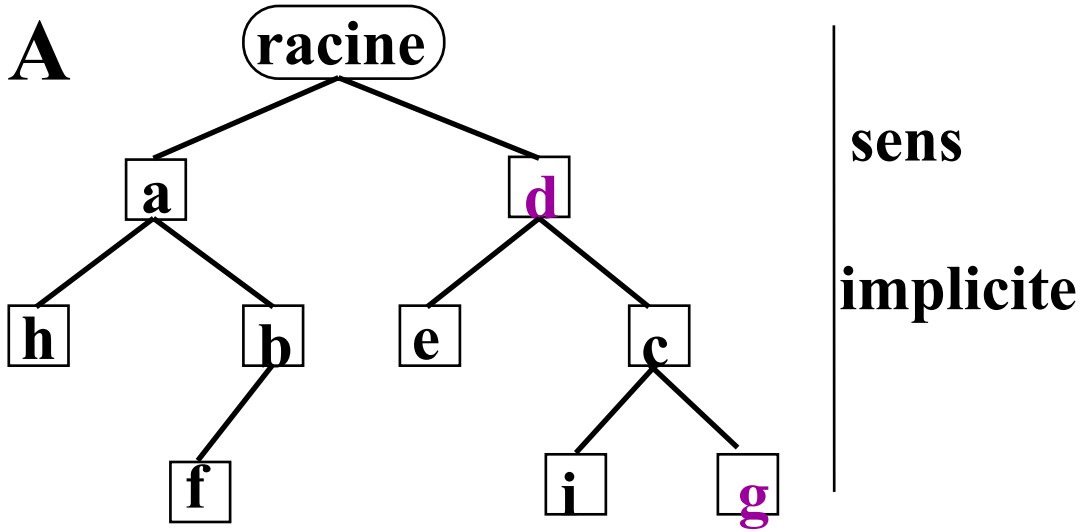
(r-d-e)

hauteur (ou profondeur): longueur de la plus longue branche *(ex: r-a-b-f donc h(A)=3)*



ascendant de x: nœud situé sur le chemin de r à x (*d ascendant de g*)

descendant de x: nœud t.q. ∃ un chemin de x à ce nœud (*f descendant de a*)



3.1.1 REPRÉSENTATION

r	a	d	h	b	e	c	-	-	f	-	-	-	i	g

Dans tout ce chapitre nous supposerons que les tableaux sont indicés de 1 à n (et non de 0 à n-1)

3.1.1 REPRÉSENTATION

EXEMPLE

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
r	a	d	h	b	e	c	-	-	f	-	-	-	i	g

***b** est en 5*

ses fils sont en (2x5=) 10 et (2x5+1=) 11

ce sont donc f et - , soit un seul fils (gauche): f

son père est en (5/2=)2, **c'est donc** ***a***

3.1.1 REPRÉSENTATION

- **représentation par un tableau**

T_arbre: tableau [ ] de Elt

sommet I → fils en 2.I et 2.I+1

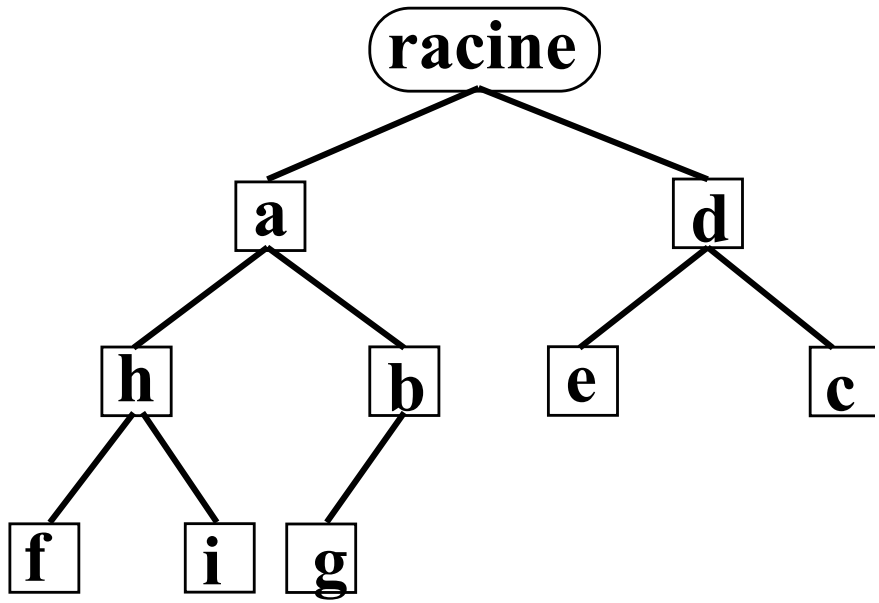
→ père en I/2

parcours facile de l'arbre

place mémoire perdue

ARBRE BINAIRE PARFAIT

toutes les feuilles sont situées sur 2 niveaux au plus, les feuilles du dernier niveau sont groupées sur la gauche



→pas de place

mémoire perdue

EXEMPLE

1	2	3	4	5	6	7	8	9	10
r	a	d	h	b	e	c	f	i	g

- **représentation par un chaînage**

Un "noeud" contient un élément et deux pointeurs

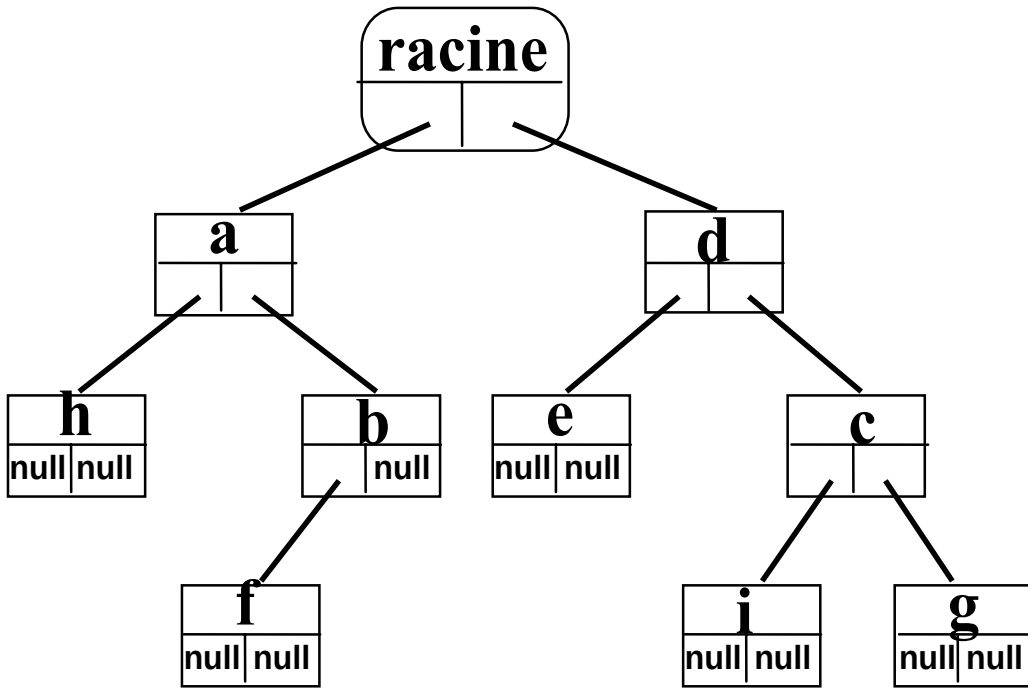
x	**élément de la classe Elt**
gauche	**pointeur vers le sous-arbre fils gauche (null si il n'existe pas)**
droit	**pointeur vers le sous-arbre fils droit (null si il n'existe pas)**

Remontée: haut pointeur vers le père

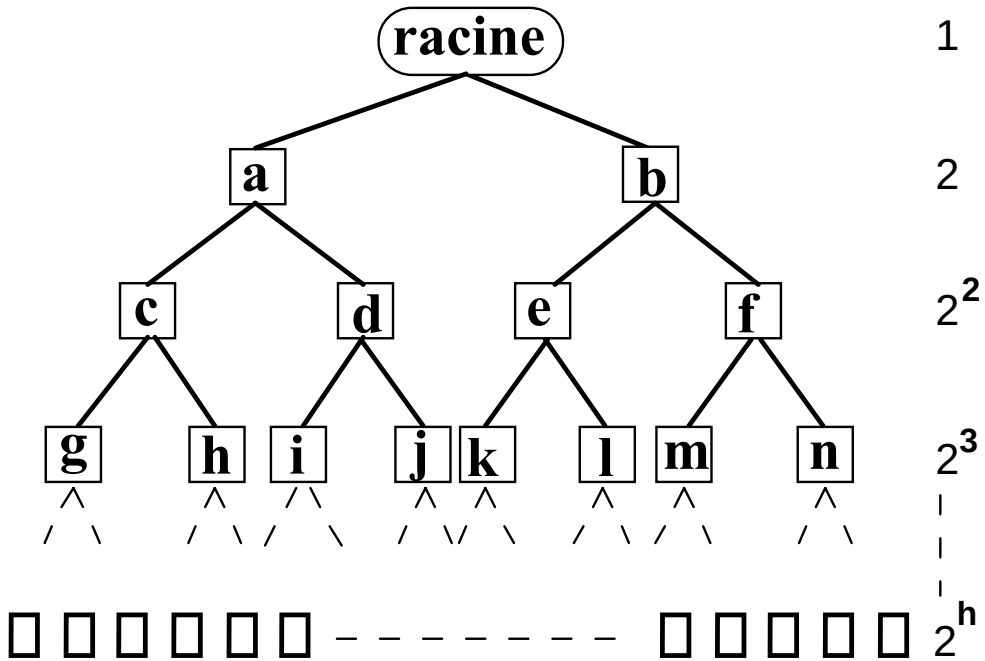
- **représentation par un chaînage**

```
classe Arbre {
Elt  valeur;
Arbre  filsgauche;
Arbre  filsdroit;

Arbre (Elt element, Arbre gauche, Arbre droit){
    this.valeur = element;
    this.filsgauche = gauche;
    this.filsdroit = droit;
    }
//les méthodes

}
```

- **représentation par un chaînage**



3.1.2 HAUTEUR

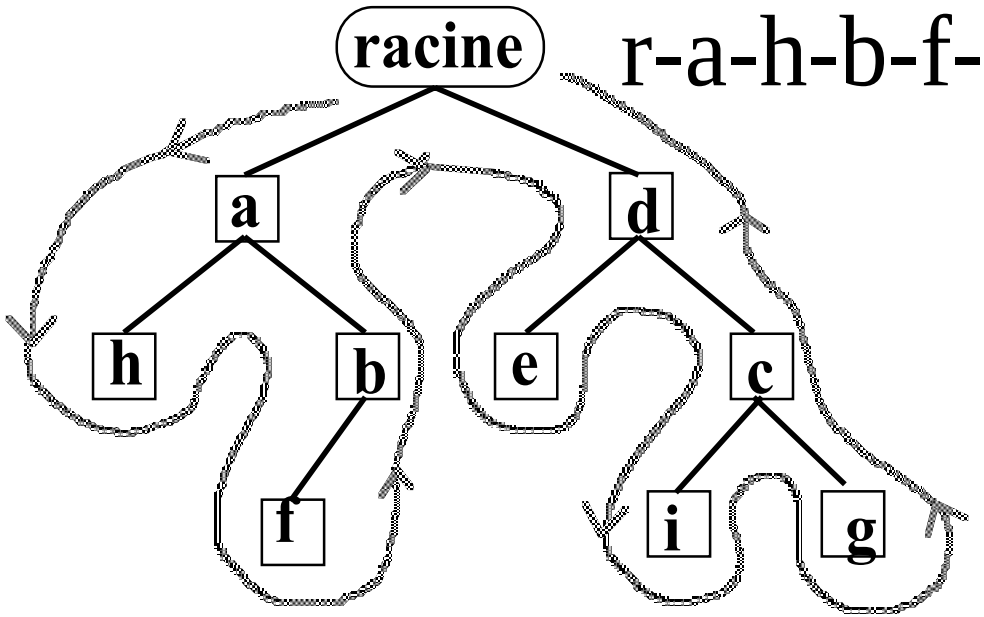


Nombre d'éléments = $N \leq 1 + 2 + 2^2 + 2^3 + \ldots + 2^h$

$N \leq 2^{h+1} - 1$ soit $h \geq \log_2 (N+1) - 1$

3.1.3 PARCOURS D'ARBRES

parcours en profondeur (back-track)



r-a-h-b-f-d-e-c-i-g

programmation de l'exploration facile à l'aide d'une pile ou de la récursivité

Soit un traitement à faire en tout nœud d'un arbre

programme récursif:

void **traitement** **(arbre)**

début

traiter_la_racine_de_arbre;

si **gauche** $\neq$ **null** alors

traitement(gauche);

finsi;

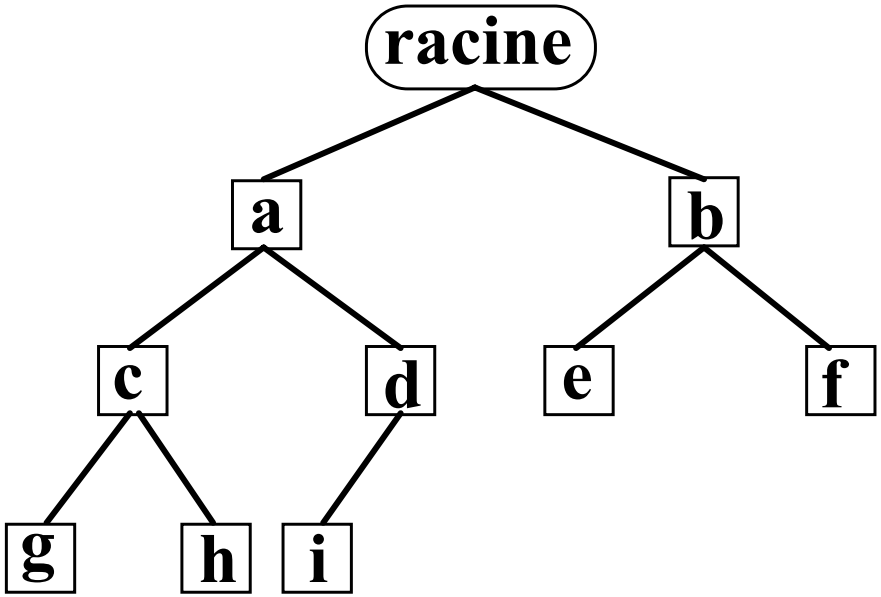
si **droit** $\neq$ **null** alors

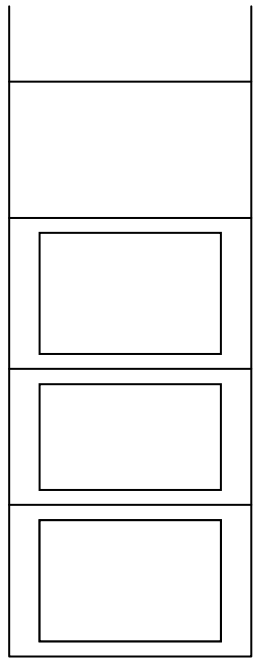
traitement(droit);

finsi;

fin

remarque: depuis un nœud on n'accède qu'à ses fils





Empiler b Traiter a

Empiler d Traiter c

Empiler h Traiter g

Sommet Dépiler Traiter h

Sommet Dépiler Traiter d

Traiter i

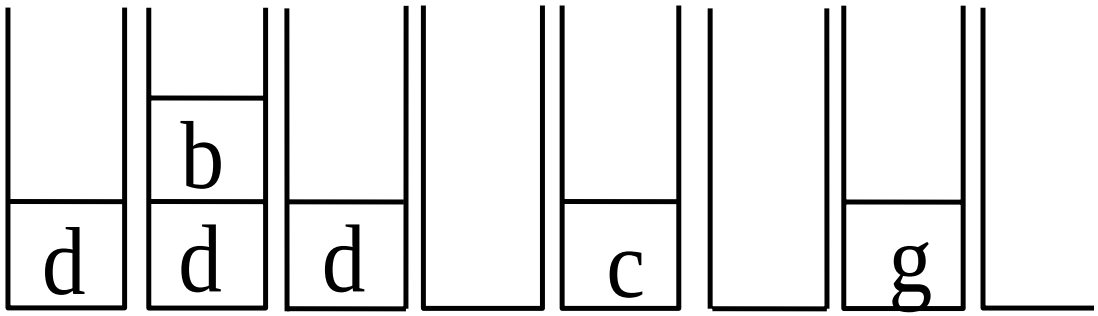
Sommet Dépiler traiter b

Empiler f Traiter e

Sommet Dépiler Traiter f

Pile vide: fin

EXEMPLE (suite) **impossible de "passer" de g à b**



évolution de la pile

traiter racine;

empiler (d);

traiter(a);

empiler (b);

traiter(h);

(--pas de fils,rien à empiler)

dépiler; (→b);

traiter(b);

(--pas de fils droit,rien à empiler)

traiter(f);

(--pas de fils,rien à empiler)

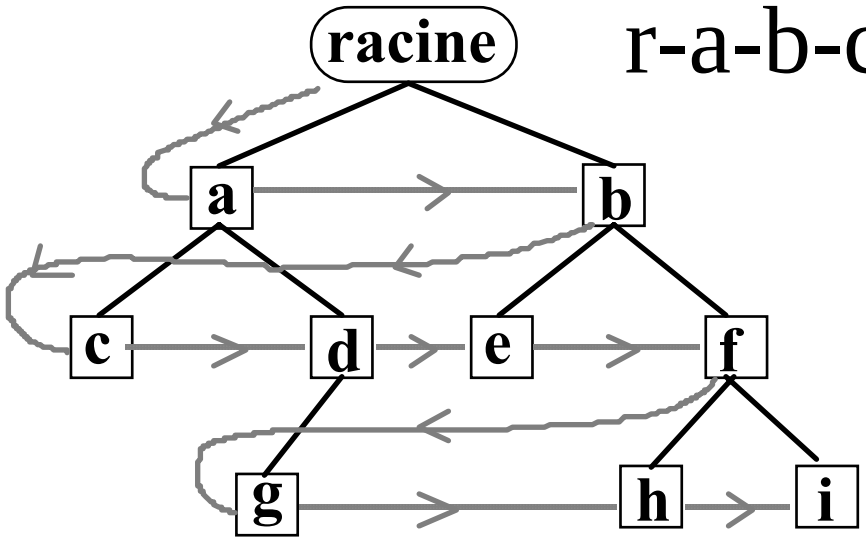
dépiler; (→d);

traiter(d);

empiler(c);

traiter(e); …

parcours en largeur



r-a-b-c-d-e-f-g-h-i

ou parcours quelconque

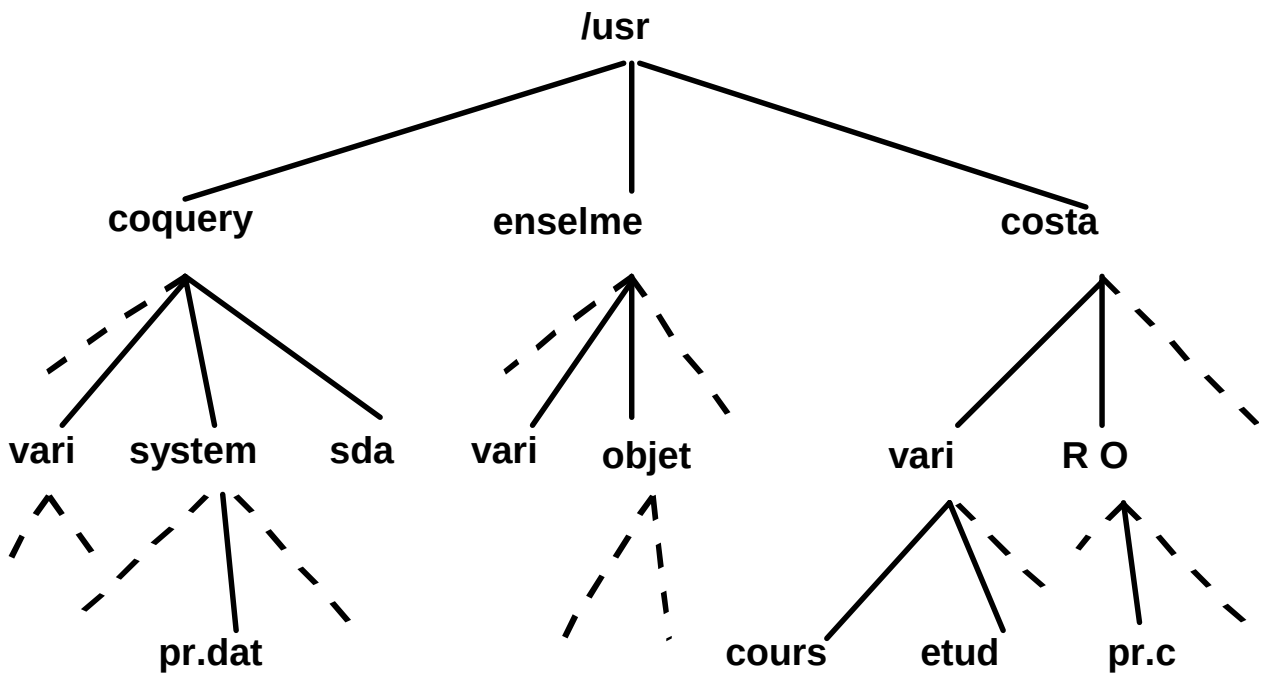
r-a-c-b-d-f-e-h-g-i

programmation plus complexe car

espace mémoire requis important

3.2 ARBRES GÉNÉRAUX

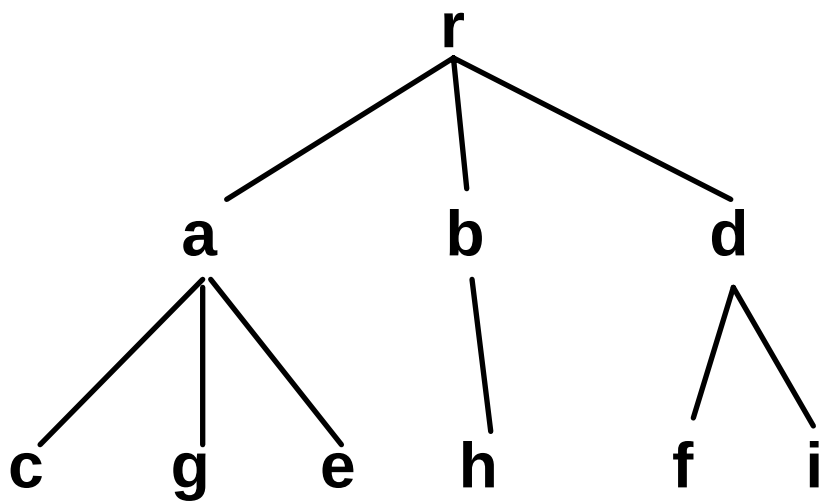
EXEMPLE **directory sous UNIX**

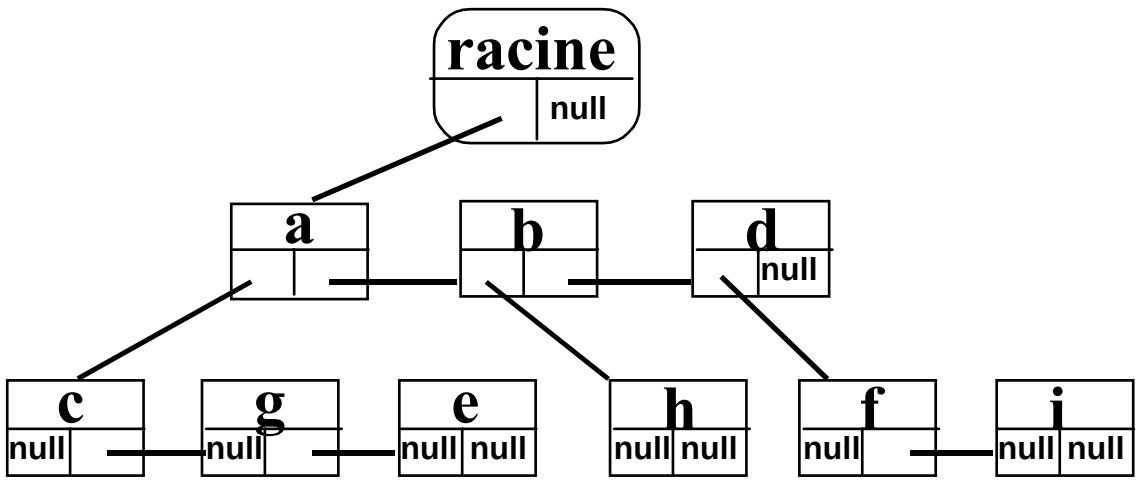


- **représentation par un chaînage**

Un "g_noeud" contient un élément et deux pointeurs

x	**élément de la classe Elt**
premier fils	**pointeur vers le fils gauche (null si il n'existe pas)**
frère droit	**pointeur vers le fils droit (null si il n'existe pas)**


r
a
b
d
c
g
e
h
f
i


racine
null
a
b
d
null
c
null
g
null
e
null
null
h
null
null
f
null
i
null
null

3.3 Les TAS

files de priorité

heap

3.3.1 Définition d'un TAS

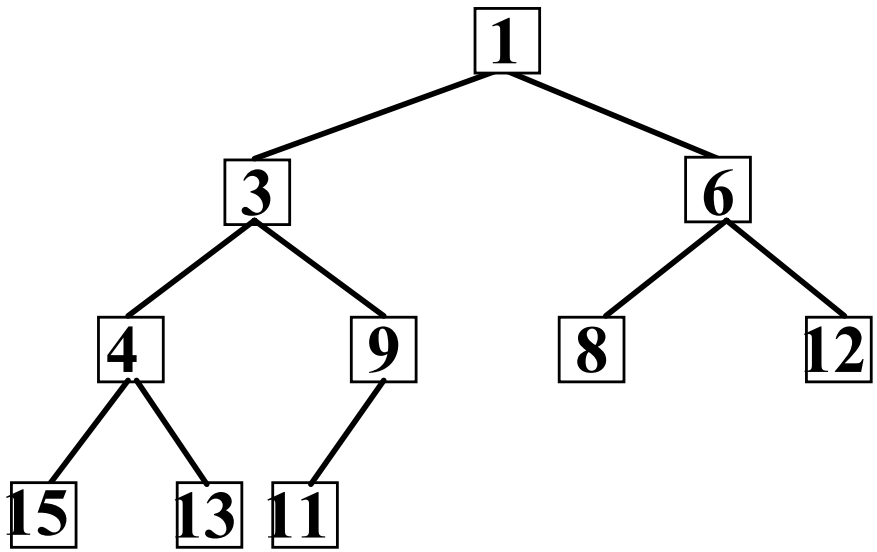
- **arbre binaire parfait**
- **clé d'un nœud ≤ clés de tous ses descendants**

(tas min: ≤ tas max: ≥)

racine

= clé

minimale



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

Pour simplifier la présentation des tas,
on supposera que le tas ne contient
que des entiers, ou des clés
(ensemble totalement ordonné)
représentées par des entiers (ou classe C_clé).

Si le tas contient des éléments de la classe Elt alors il est ordonné selon les clés des éléments.
Dans la suite, il faut alors écrire tas[I].clé pour tas[I] et e.clé pour e, …

- Un tas est un arbre binaire parfait dont les éléments sont ordonnés selon leurs clés; il est représenté par un tableau.
- On peut ajouter un élément si le tas n'est pas "plein" en conservant la structure d'arbre binaire parfait et l'ordre.
- On peut retirer l'élément le plus petit du tas, premier élément du tableau.

On va définir une classe "C_tas"

3.3.2 Implémentation d'un tas

```
classe C_tas{
    entier taille;
    C_clé[ ] tas;
    entier long=0;


C_tas( entier t){
    this.taille=t;
    this.tas=new C_clé[t];
    }
//les méthodes
}
```

méthodes	conditions
min_tas ( ) retourne C_clé	tas non vide
supprimer_min ( )	tas non vide
insérer (C_clé clé)	tas non plein
est_vide ( ) retourne booléen;	

création d'un tas vide :

```
C_tas    le_tas = new C_tas(taille);
```

booléen **estvide ( ) ;**

début

retourner (long==0);

fin

fonction en O(1)

Dans toute la présentation des tas, on omet "this"

C_clé min_tas ( );

//retourne la plus petite clé (celle de la racine)

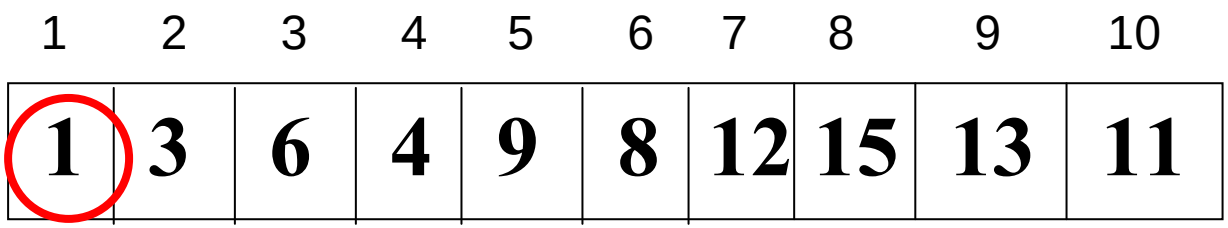
début

si **est_vide( )** alors "erreur";

sinon **retourner tas[1];**

finsi;

fin

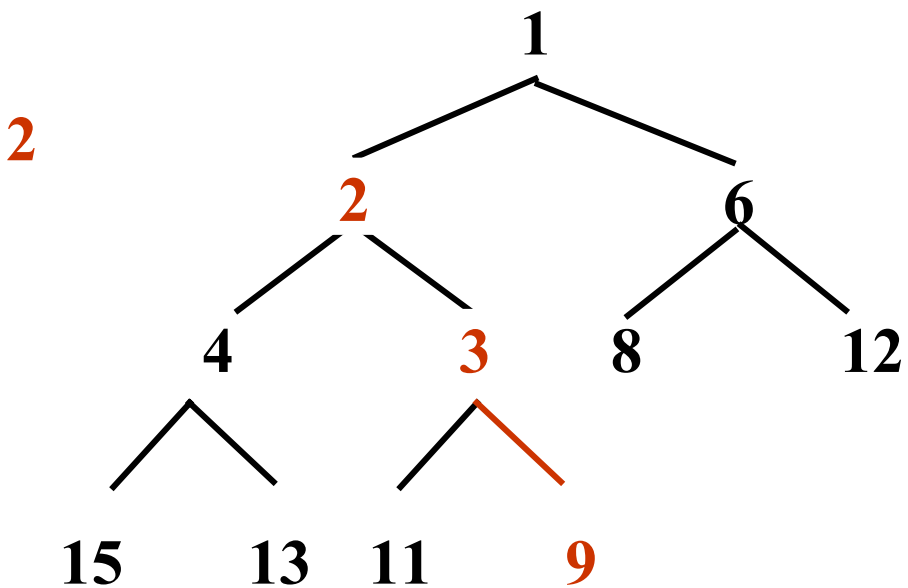


fonction en O(1)

insertion:

ajout d'une feuille à l'arbre; placement de la clé de façon à garder la structure de tas (elle "remonte")

EXEMPLE **min_tas = 1** **ajout de la clé 2**

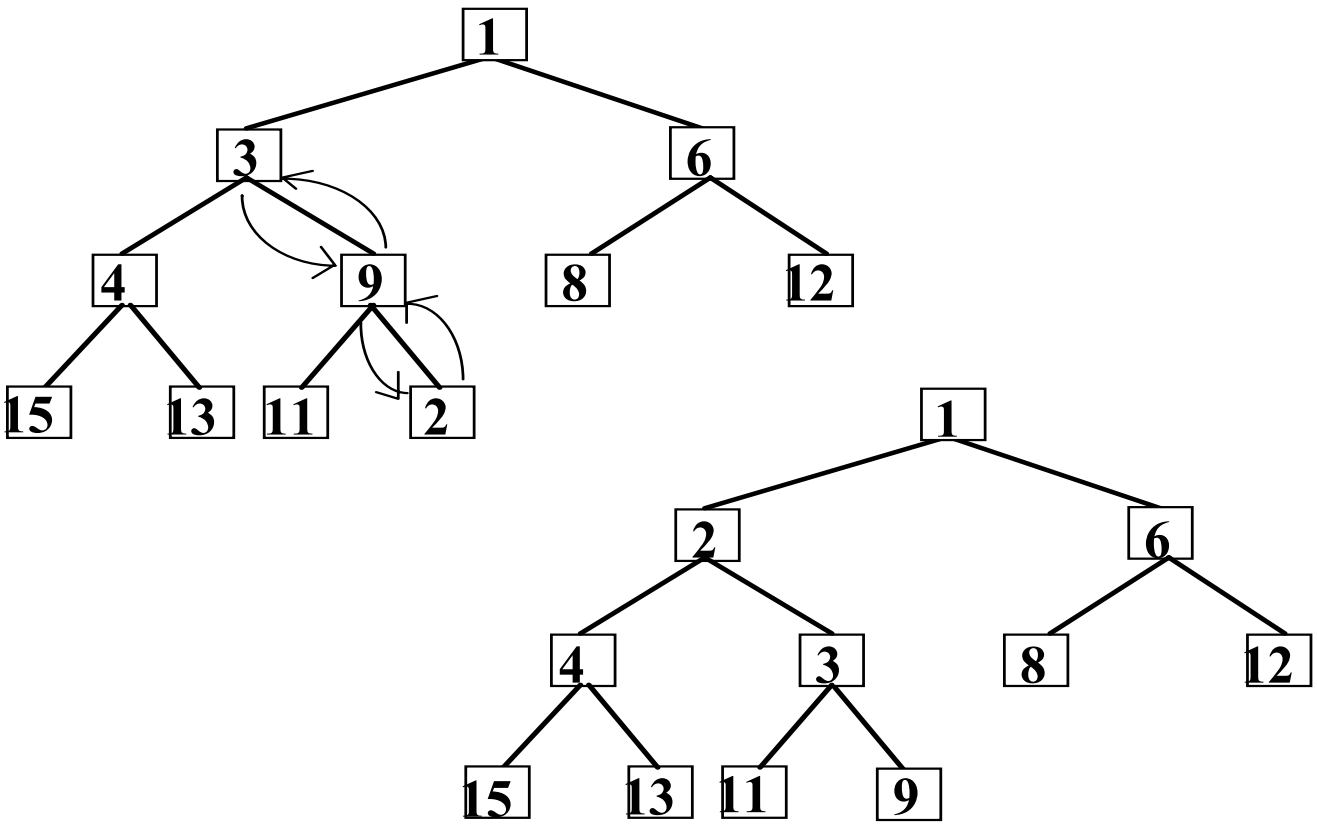


1	2	3	4	5	6	7	8	9	10
1	2	6	4	3	8	12	15	13	11	9

insertion:

ajout d'une feuille à l'arbre; placement de la clé de façon à garder la structure de tas (elle "remonte")

EXEMPLE **min_tas = 1** **ajout de la clé 2**



```
void  insérer (C_clé clé)
 Indice  fils, père;  booléen place_trouvée=faux;
début
        si long == taille   alors  "erreur"; //tas plein
        sinon
            long = long + 1;
            fils = long;                              // 1ère place de clé
            père = long;
            tant que   non place_trouvée et  père ≠ 1 faire
                  père = fils/2;
                  si tas[père] ≤ clé alors
                     place_trouvée = vrai;          //= fils
                  sinon  tas[fils] = tas[père];      //descendre la clé du père
                         fils = père;                //remonter d'un niveau
                  finsi;
            fait;
            tas[fils] = clé;                          //la clé est à sa place
         finsi;
fin                                                   //si fils=1, c'est la racine
```

```
void insérer (C_clé clé)
 Indice fils, père; booléen place_trouvée=faux;
début
        si long == taille   alors  "erreur"; //tas plein
        sinon
            long = long + 1;
            fils = long;                              // 1ère place de clé
            père = long;
            tant que   non place_trouvée et  père ≠ 1 faire
                    père = fils/2;
                    si tas[père] ≤ clé alors
                       place_trouvée = vrai;
                    sinon  tas[fils] = tas[père];
                           fils = père;
                    finsi;
            fait;
            tas[fils] = clé;
         finsi;
fin
```

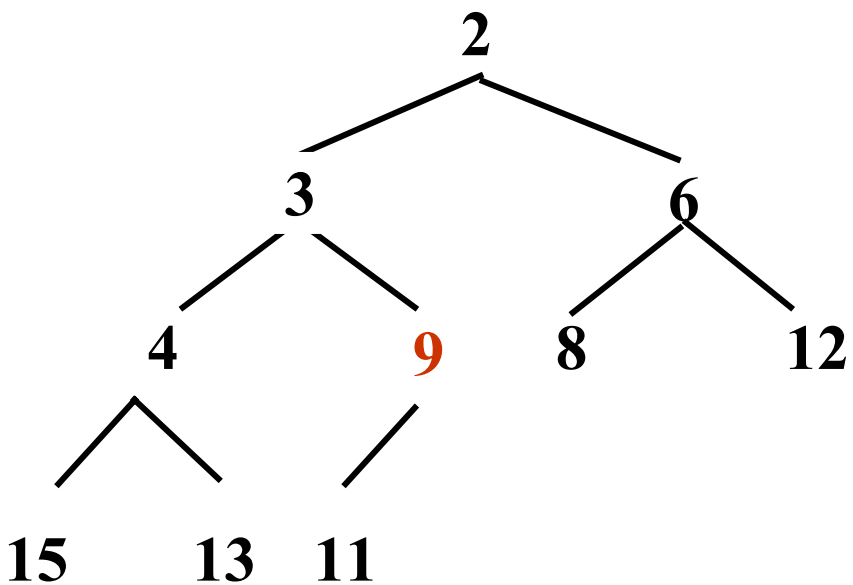
Au pire, nombre de passage dans la boucle= hauteur de l'arbre h

méthode en O(log n)

Suppression du minimum:

Disparition de la dernière feuille de l'arbre: "derclé" à replacer. Remontée des clés dans le tas: on remonte en chaque nœud le plus petit des 2 fils jusqu'à avoir trouvé la place de "derclé"

*EXEMPLE suppression de **1** derclé = **9***

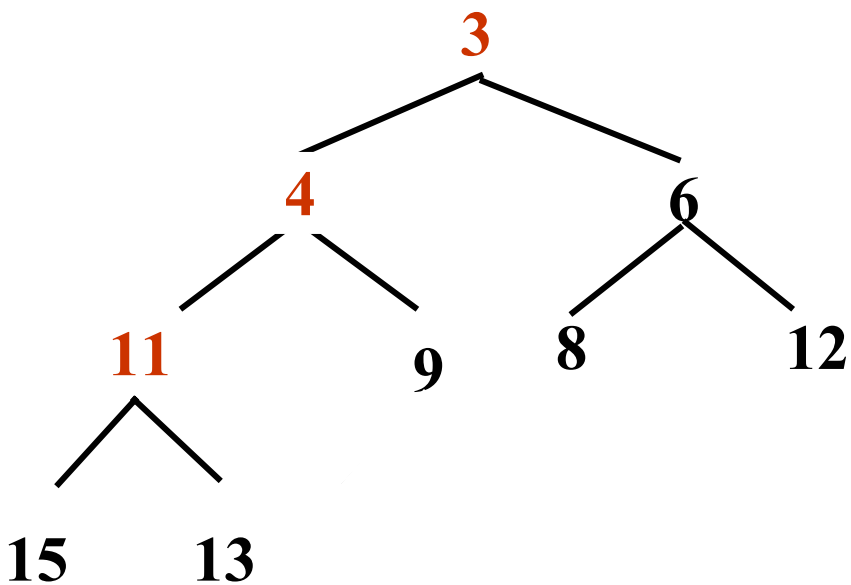


1	2	3	4	5	6	7	8	9	10	
2	**3**	**6**	**4**	**9**	**8**	**12**	**15**	**13**	**11**	

Suppression du minimum:

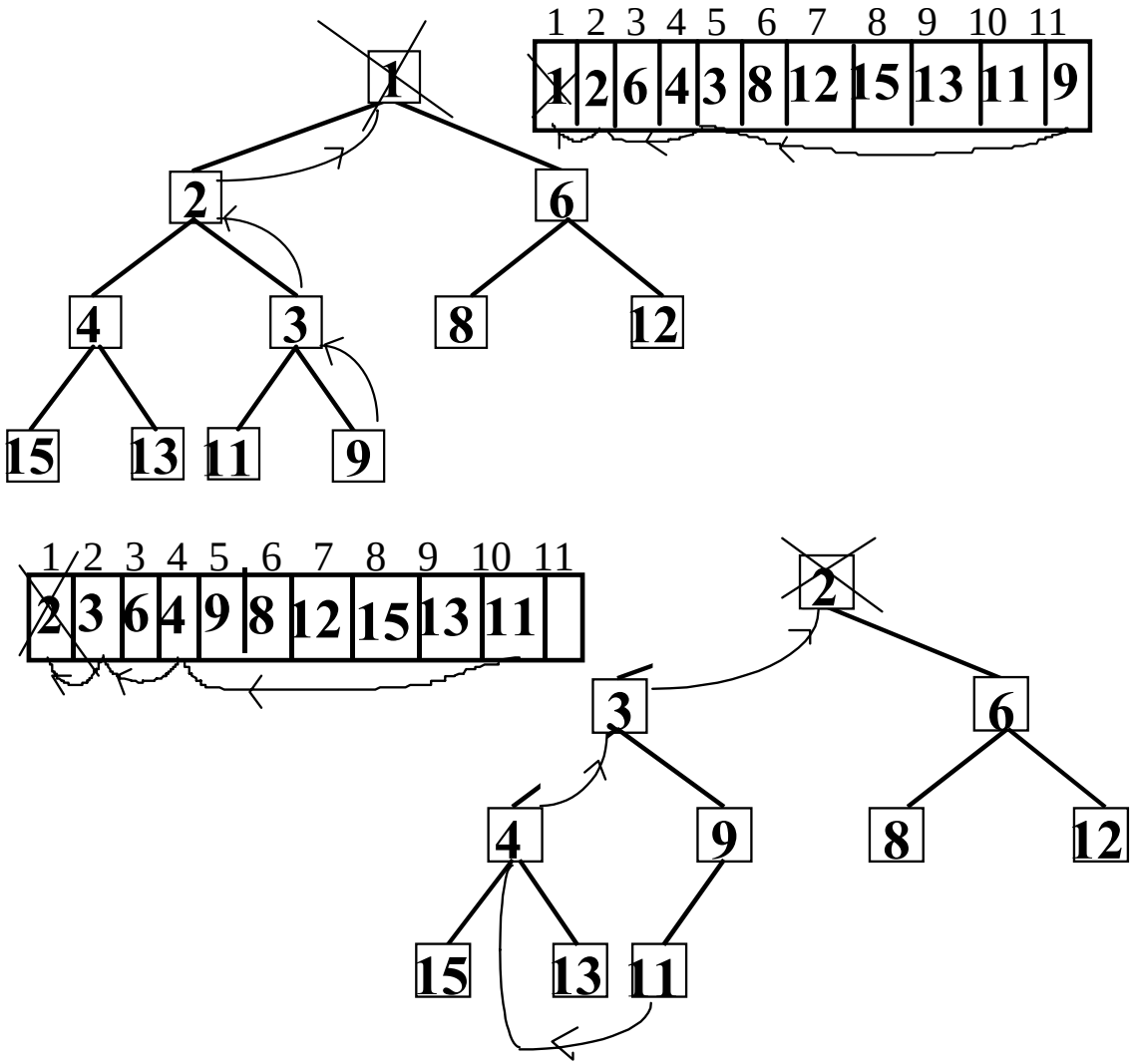
*EXEMPLE (suite) suppression de **2***

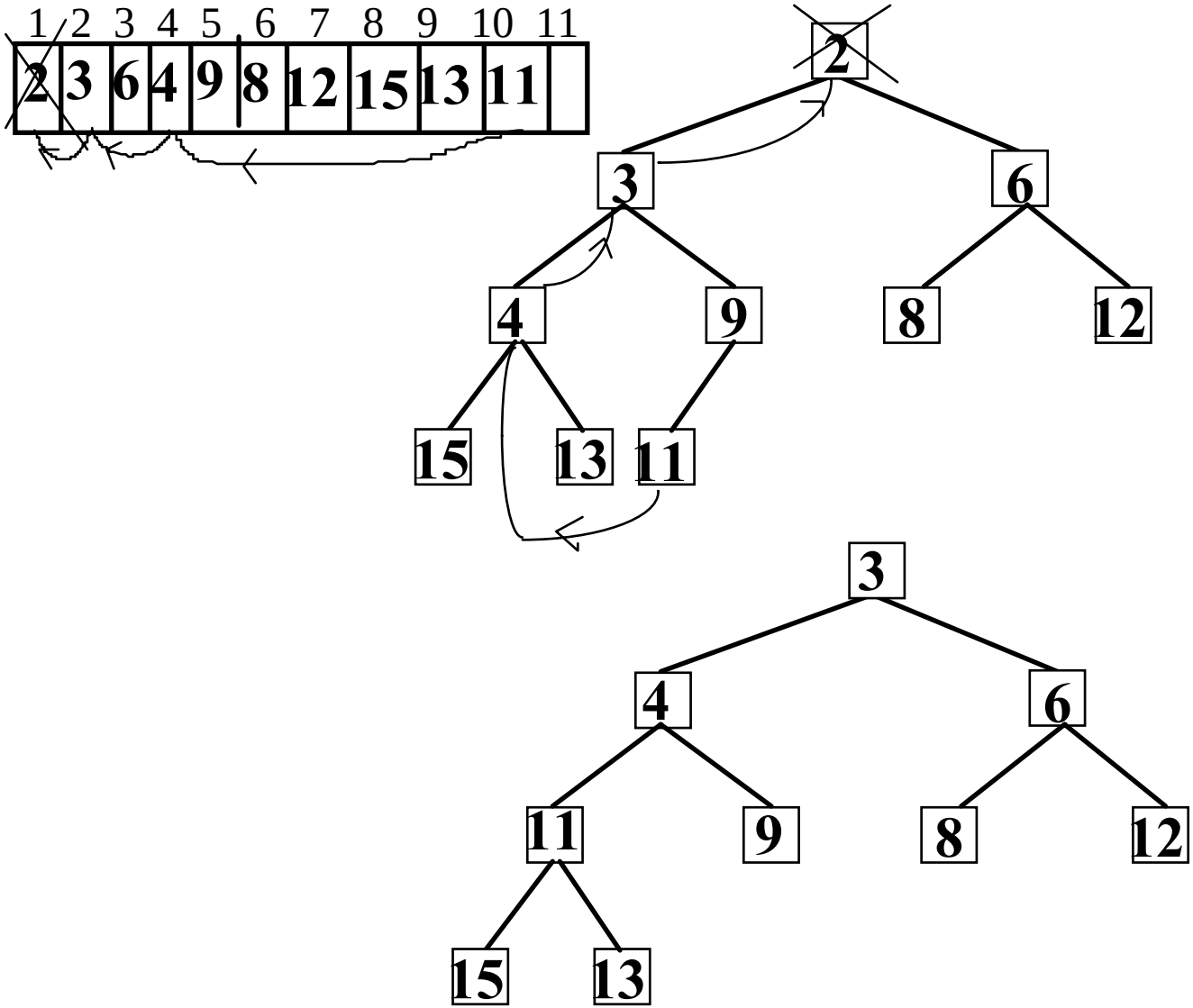
*derclé = **11***



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

EXEMPLE suppression du minimum




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

void **supprimer_min ( )**

//la dernière feuille de l'arbre va disparaître: en partant du haut, on remonte
//le plus petit des fils tant qu'il est inférieur à la dernière feuille; ensuite, on
//déplace la dernière feuille vers la place libre (place_trouvée)

Indice fils, père; **booléen place_trouvée= faux;**

C_clé derclé; *//clé de la dernière feuille*

début

si long = 0 alors "erreur"; *//tas vide* **sinon**

derclé = tas[long];

long = long - 1; *//on supprime la dernière feuille*

si long ≠ 0 alors

//sinon on a supprimé la racine et le tas est vide

père = 1; *//départ de la racine*

```
tant que 2.père ≤ long et non place_trouvée faire
  //tant qu'il reste un fils et qu'on n'a pas replacé derclé
  fils = 2.père;                //on passe au premier fils
  si fils < long alors          //il a un frère et on prend le plus petit des 2
    si tas[fils+1] ≤ tas[fils] alors fils = fils + 1; finsi;
  finsi;
  si  derclé ≤ tas[fils]  alors      //les clés des fils sont ≥ derclé
    place_trouvée = vrai;            //c'est père
  sinon
    tas[père] = tas[fils];           //on remonte le plus petit fils
    père = fils;                     //descendre d'un niveau dans l'arbre
  finsi;
fait;
tas[père] = derclé;                  //on replace derclé
finsi; finsi; fin
```

```
tant que 2.père ≤ long et non place_trouvée faire
  //tant qu'il reste un fils et qu'on n'a pas replacé derclé
  fils = 2.père;              //on passe au premier fils
  si fils < long alors        //il a un frère et on prend le plus petit des 2
    si tas[fils+1] ≤ tas[fils] alors fils = fils + 1; finsi;
  finsi;
  si  derclé ≤ tas[fils]  alors
    place_trouvée = vrai;
  sinon
    tas[père] = tas[fils];
    père = fils;
  finsi;
fait;
tas[père] = derclé;
finsi; finsi; fin
```

Au pire, nombre de passage dans la boucle= hauteur de l'arbre h

méthode en O(log n)

complexité

estvide et min_tas en O(1)

insérer et supprimer_min
en O(h)= O($\log_2$ n)

structure intéressante si
accès fréquent au minimum
(cf tri par tas: cours 4)