NFP136- STRUCTURES ARBORESCENTES

PLAN

- **Arbres binaires**
- **Arbres généraux**
- **Tas**

Arbres Binaires

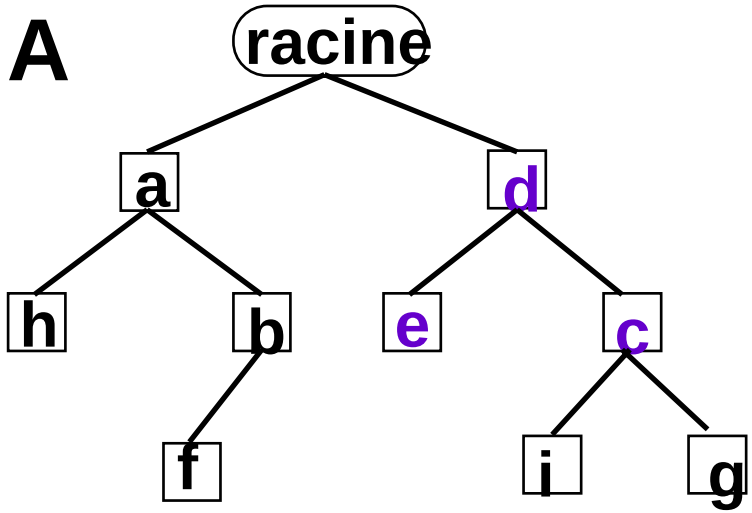
"arbre" en informatique =

"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)

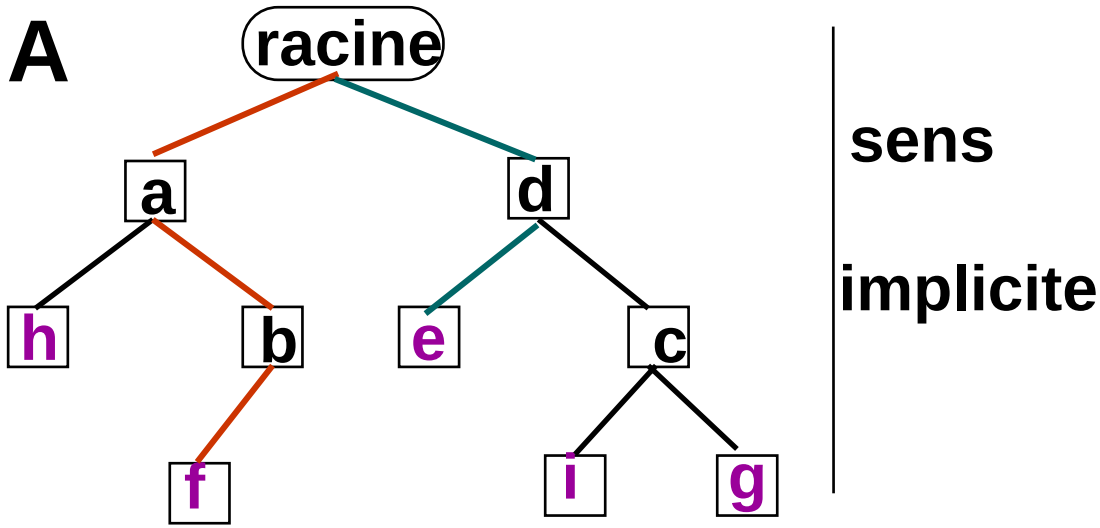
définition récursive d'un arbre binaire

ensemble vide

ou

ensemble formé

- **d'une racine**
- **d'un sous-arbre droit**
- **d'un sous arbre gauche**

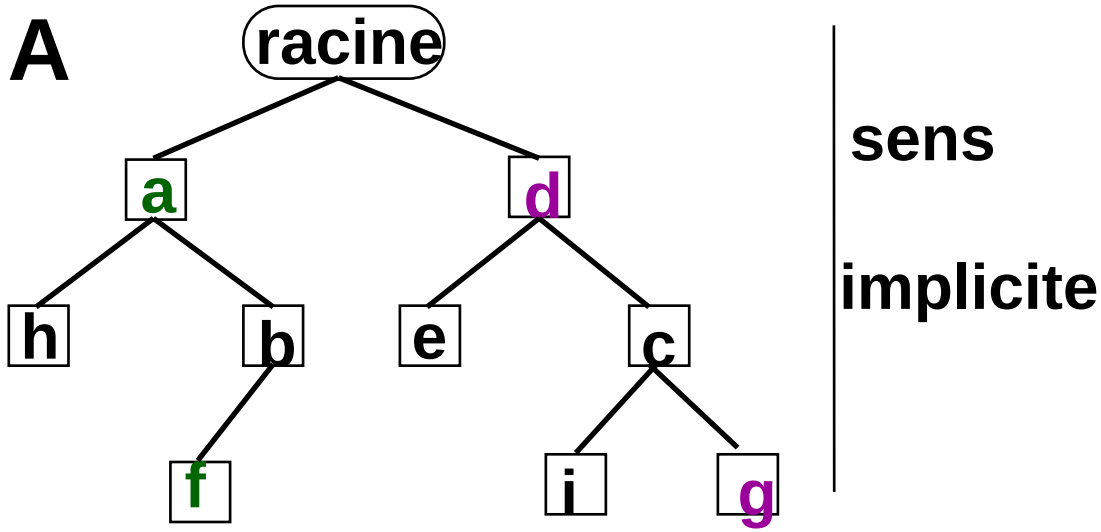


feuille: nœud sans fils (h,f,e,i,g)

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)*

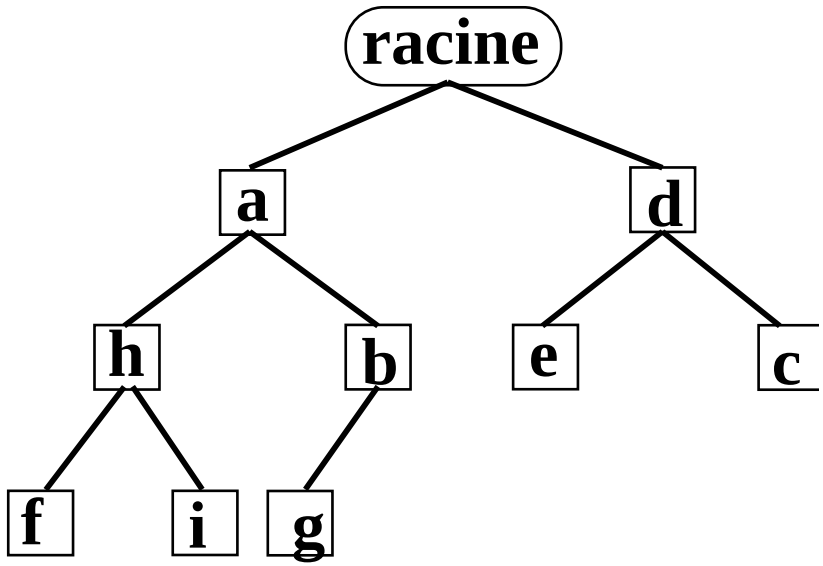


ascendant de x: nœud situé sur le chemin de r à x *(d ascendant de g)*

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

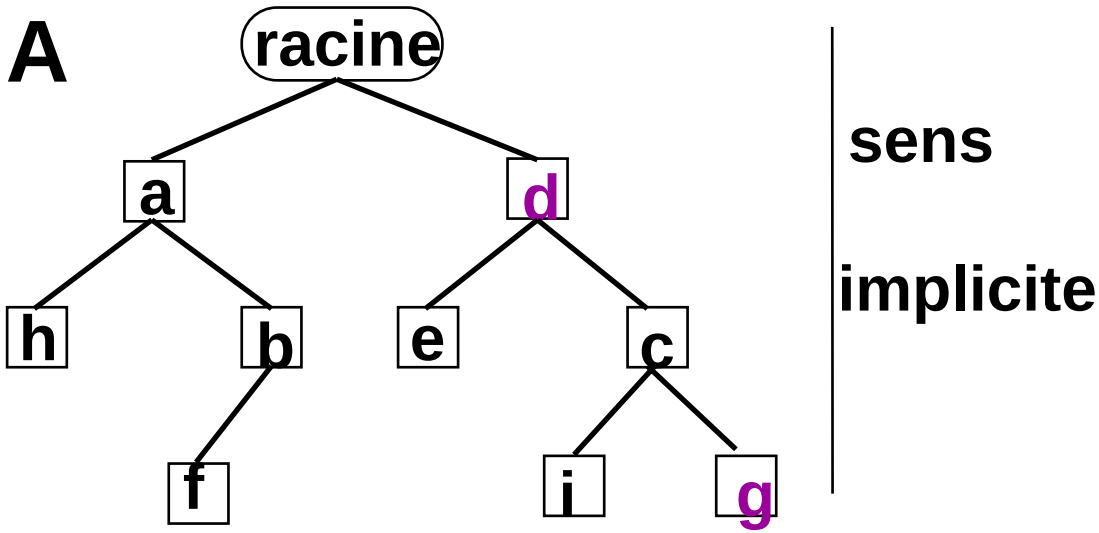
ARBRE BINAIRE PARFAIT

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



Représentation des Arbres

ARBRES BINAIRES



REPRÉSENTATION

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

REPRÉSENTATION

EXEMPLE

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
r	a	d	h	b	e	c	-	-	f	-	-	-	i	g

b est en 4

ses fils sont en (2x4+1=) 9 et (2x4+2=) 10

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

son père est en ((4-1)/2=)1*, c'est donc *a

REPRÉSENTATION

- **représentation par un tableau**

T_arbre: tableau [ ] de Elt

sommet i $\rightarrow$ **fils en 2i+1 et 2i+2**

$\rightarrow$ **père en (i-1)/2**

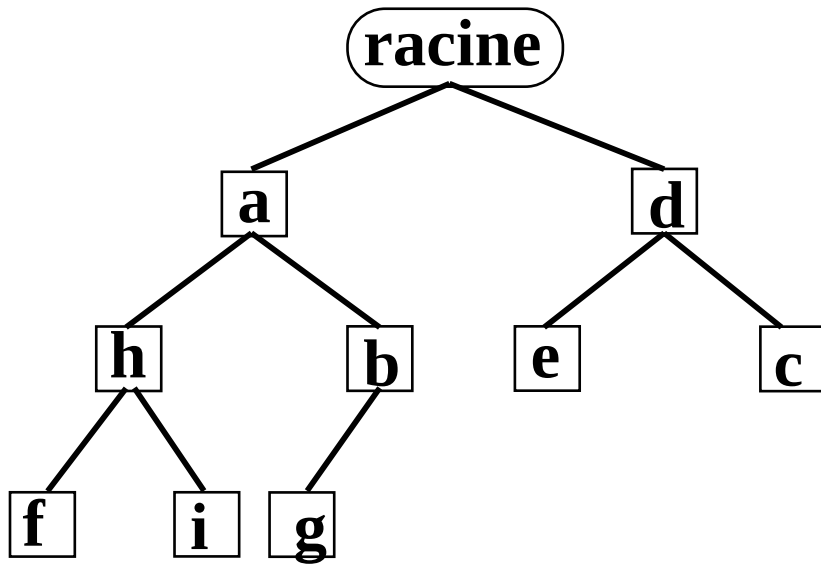
parcours facile de l'arbre

place mémoire perdue

ARBRE BINAIRE PARFAIT

→**pas de place**
mémoire perdue

EXEMPLE



0	1	2	3	4	5	6	7	8	9
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)**

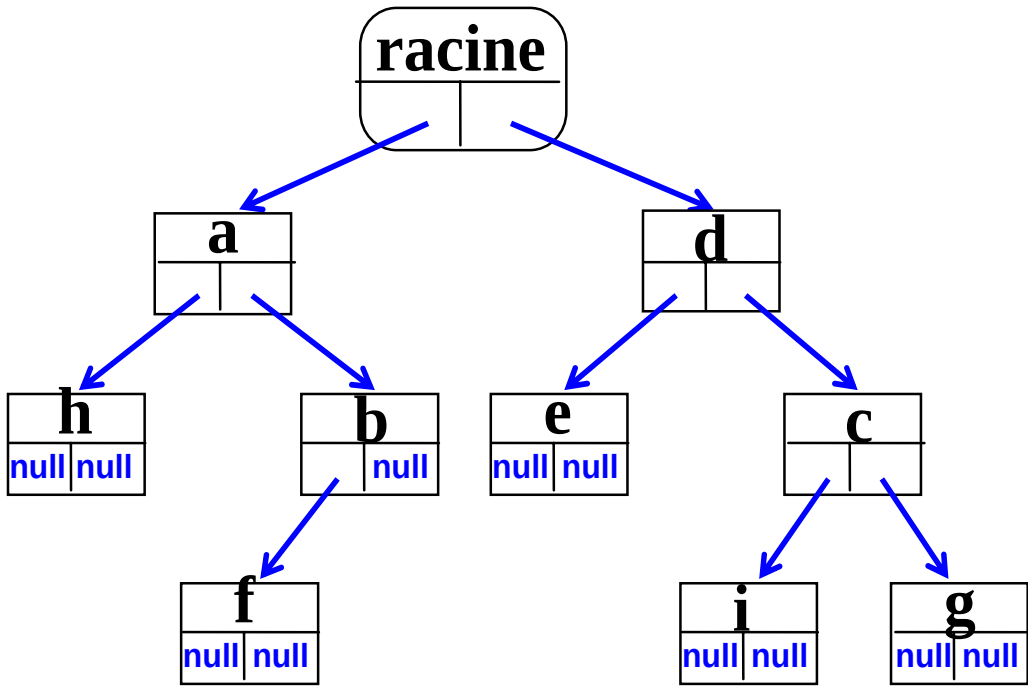
Remontée: haut pointeur vers le père

- **représentation par un chaînage**

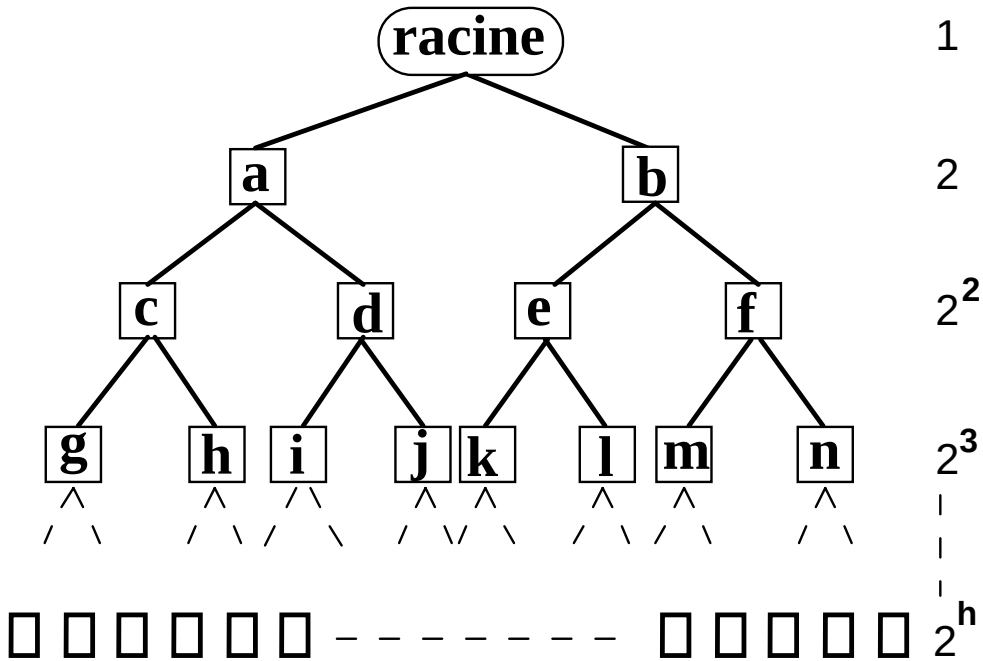
```
class Arbre {
   private int valeur;
   private Arbre filsG;
   private Arbre filsD;

public Arbre(int a, Arbre g, Arbre d) {
    valeur = a;
    filsG= g;
    filsD=d;
}
```

- **représentation par un chaînage**



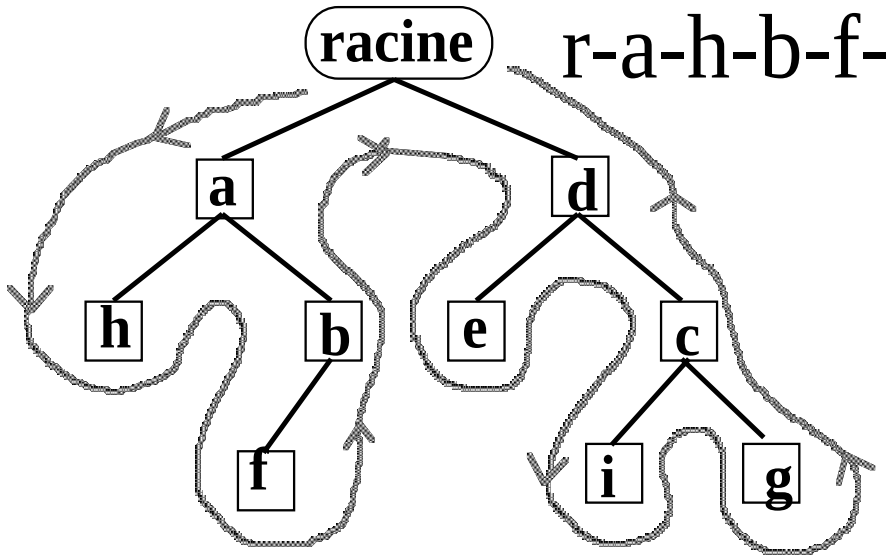
Hauteur d'un Arbre



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

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

PARCOURS D'ARBRES



Parcours en [profondeur préfixe] **(**back-track)

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

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

Méthode récursive:

```
public void profondeurPref (){
  System.out.print(valeur + " | ");
  if (filsG != null) filsG.profondeurPref();
  if (filsD != null) filsD.profondeurPref();
}
```

Ici pour un affichage mais cela peut être pour un traitement quelconque

En utilisant une pile :

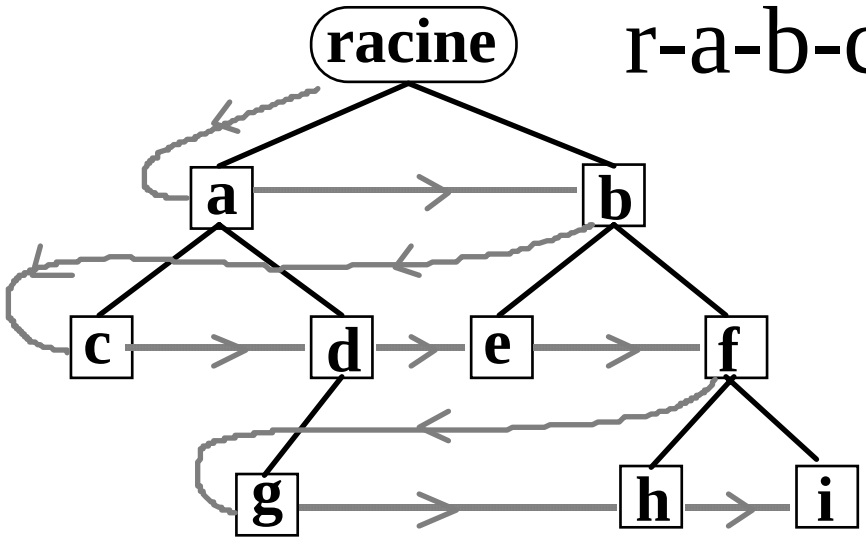
```
public void profondeurPrefPile (){
        PileArbre P=new PileArbre ();
        P.empiler(this);
        Arbre A;
        while (P != null){
                A=P.sommet(); if (A==null) return;
                P.depiler();
                System.out.print(A.valeur + " * ");
                if (A.filsD != null) P.empiler (A.filsD);
                if (A.filsG != null) P.empiler (A.filsG);
        }
}
```

Suppose l'existence d'une classe PileArbre

Mise en commun : Que fait le programme suivant ?

```
public static void main (String[] args) {
                Arbre A=new Arbre (5, null, null);
                A = new Arbre(4, A, null);
                A = new Arbre(10, new Arbre(3, null, null), A);
                Arbre Abis= new Arbre(7, new Arbre(8, null, null),
                                         new Arbre(9, null, null));
                Abis = new Arbre(2, new Arbre(60, null, null), Abis);
                A = new Arbre(15, A, Abis);
                A.profondeurPref();
                System.out.println("");
                System.out.println("parcours avec pile");
                A.profondeurPrefPile();
                System.out.println("");
}
```

parcours en largeur

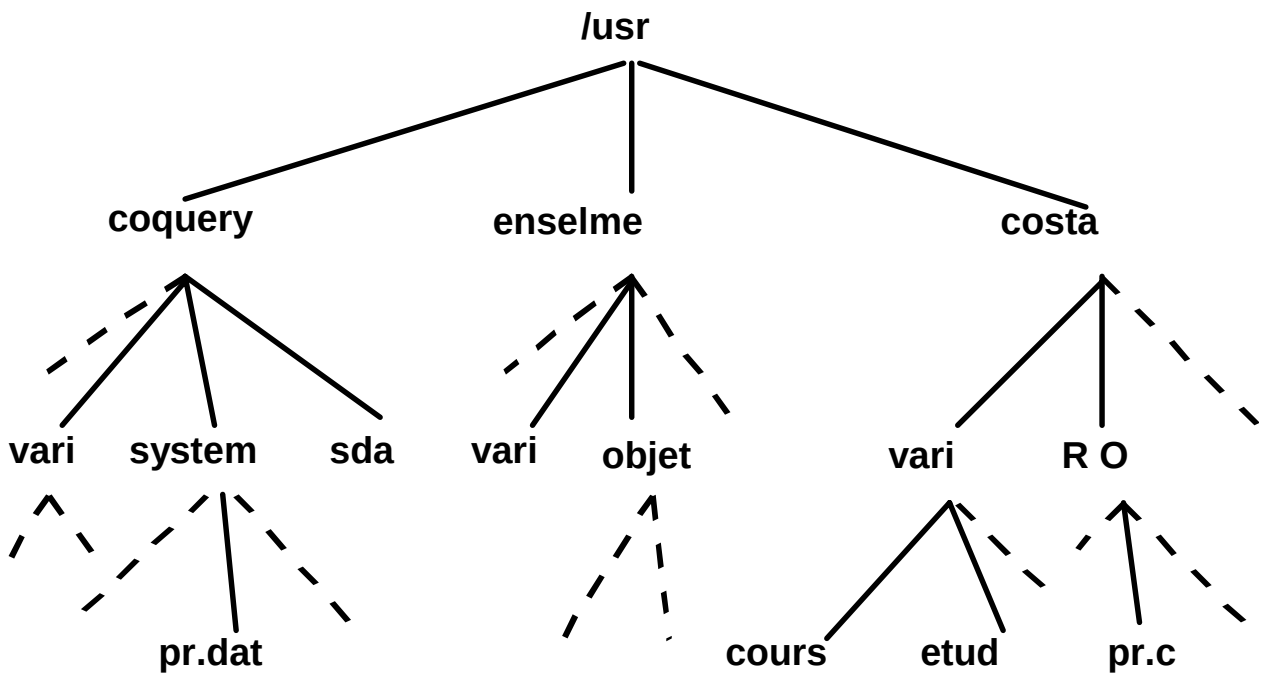


r-a-b-c-d-e-f-g-h-i

Arbres Généraux

EXEMPLE

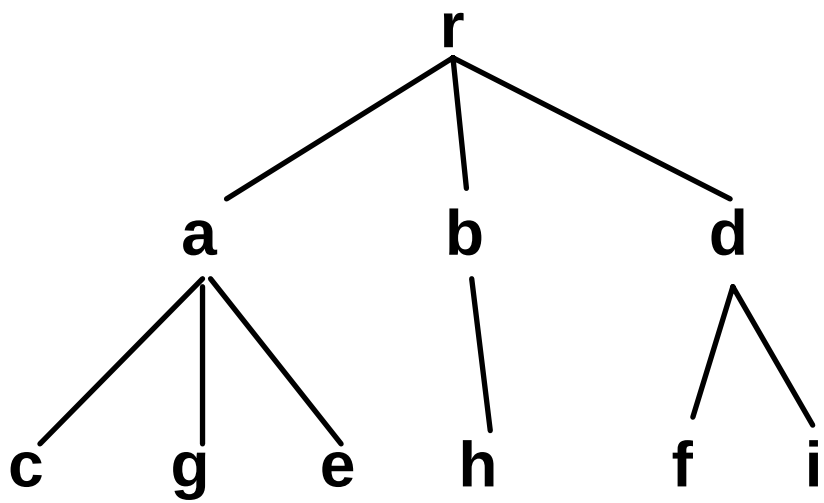
directory sous UNIX

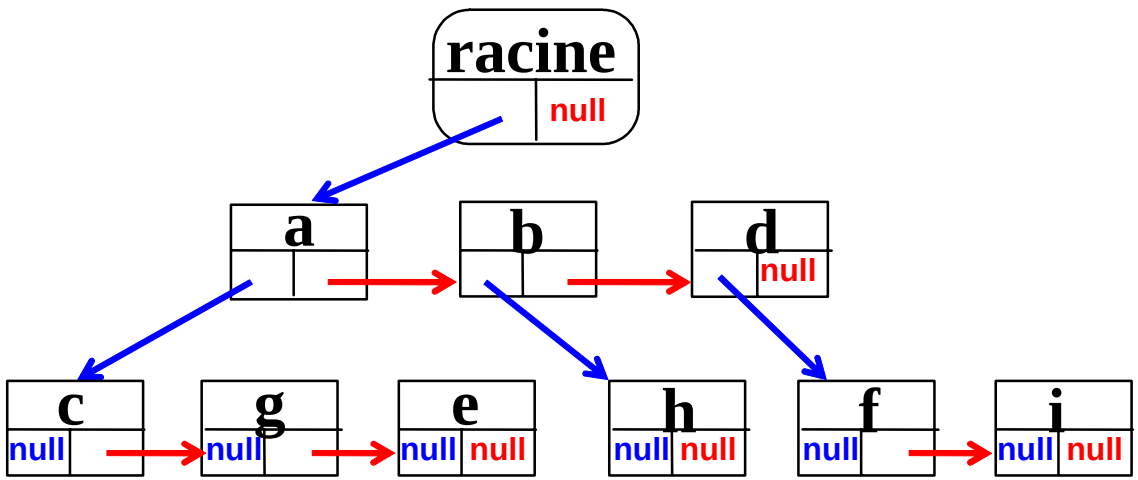


- **représentation par un chaînage**

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 frère 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

Les tas

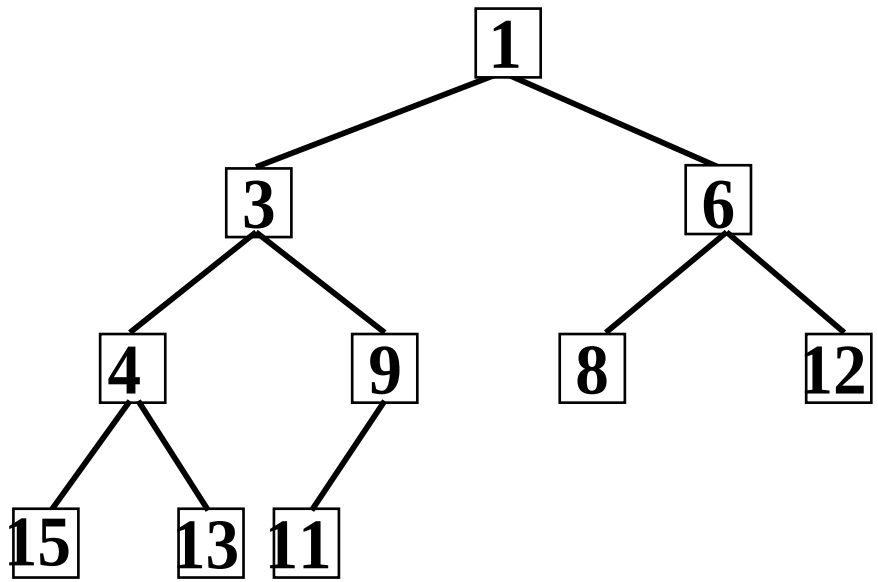
Définition d'un TAS

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

(tas min: ≤ tas max: ≥)

racine

= clé minimale



0	1	2	3	4	5	6	7	8	9	10	11
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 (classe T_clé).

- 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 Tas

Implémentation java d'un tas

```
public class Tas{
   private int[] tab;
   private int fin;

public Tas( int t ){
   tab = new int[t];
   fin = 0;
}

public boolean estVide(){return (fin==0);}

public boolean estPlein(){return(fin>=tab.length);}
```

méthodes	conditions
minimum( ) retourne t_clé	tas non vide
supprimer_min ( )	tas non vide
inserer (T_clé clé)	tas non plein

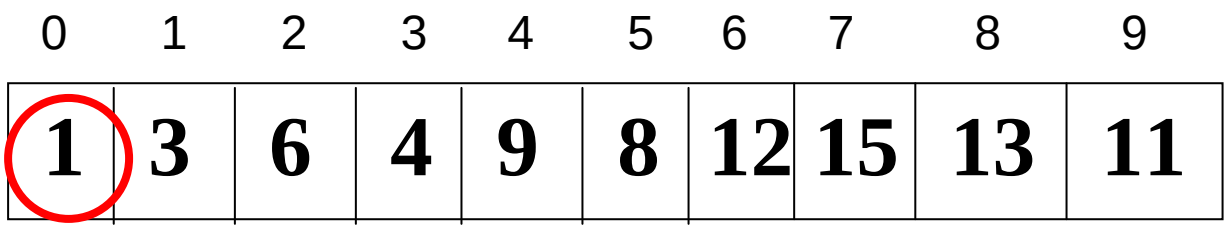
création d'un tas vide :

```
Tas T = new Tas(taille);
```

LES METHODES

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

```
public int minimum () {
   if (fin ==0)through new …;
   return (tab[0]);
}
```

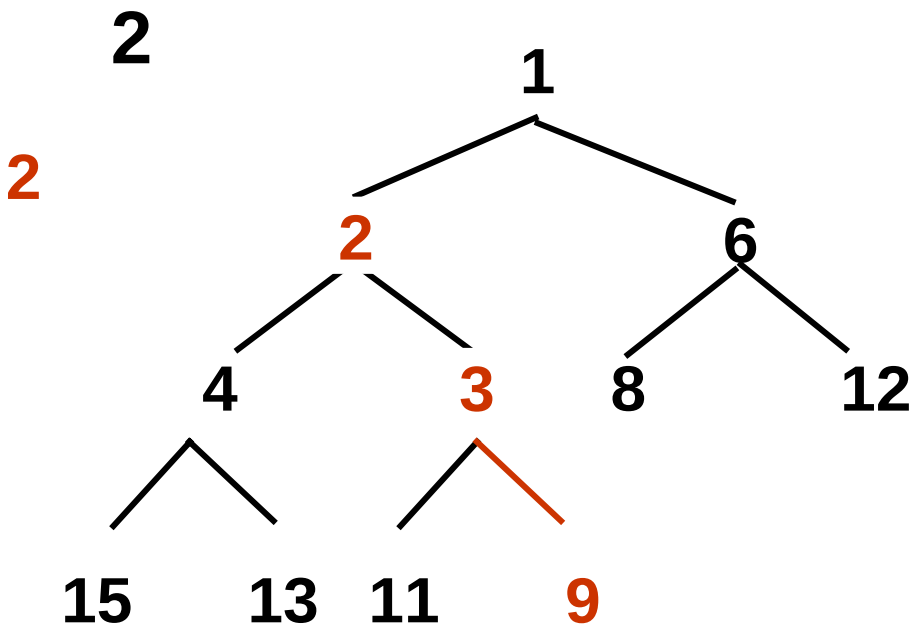


fonction en O(1)

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**

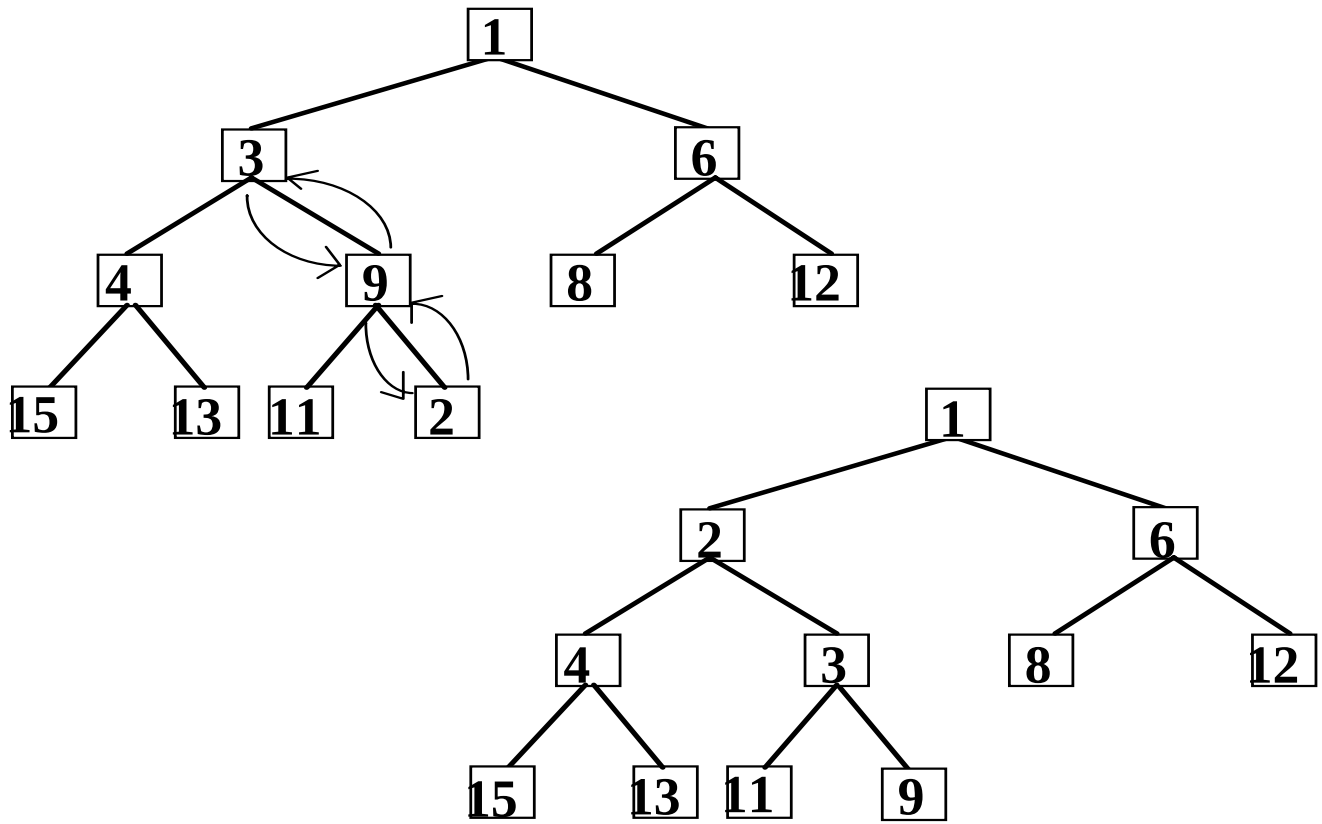


0	1	2	3	4	5	6	7	8	9	10
1	2	6	4	3	8	12	15	13	11	9

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**



```
public void inserer( int  x ){
  int fils=fin;
  int pere= (fils-1) / 2;
  while (fils >0 && tab[pere]> x){
     tab[fils]=tab[pere]; //on descend
     fils=pere;
     pere= (fils-1) / 2;
  }
  //soit fils==0 soit tab[pere]<=x
  tab[fils] = x;
  fin++;
}
```

```
public void inserer( int  x ){
   int fils=fin;
   int pere= (fils-1) / 2;
   while (fils >0 && tab[pere]> x){
       tab[fils]=tab[pere]; //on descend
       fils=pere;
       pere= (fils-1) / 2;
   }
   //soit fils==0 soit tab[pere]<=x
   tab[fils] = x;
   fin++;
}
```

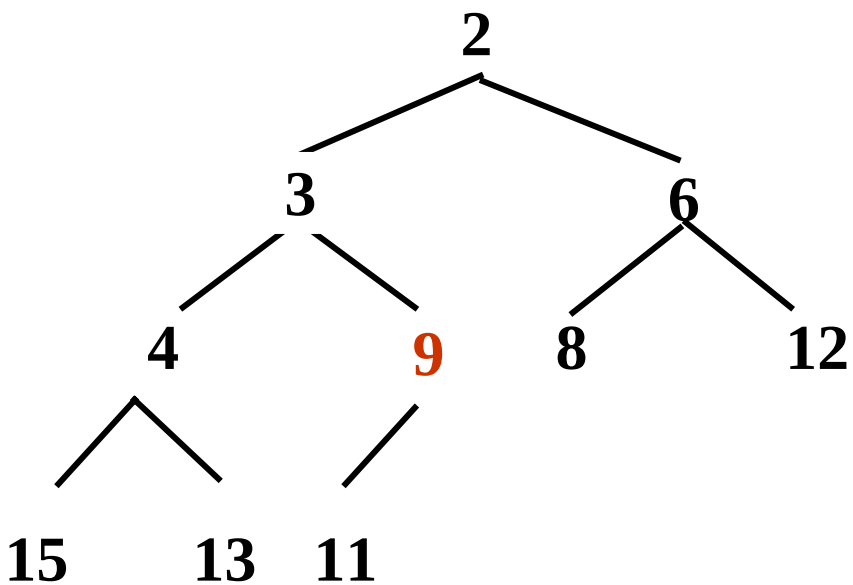
Au pire, nombre de passage dans la boucle= hauteur de l'arbre h

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***

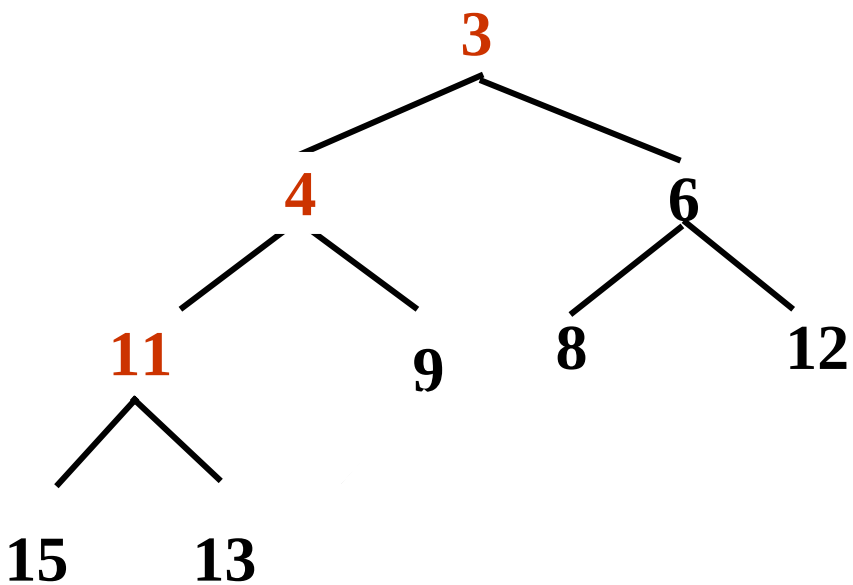


0	1	2	3	4	5	6	7	8	9	
2	**3**	**6**	**4**	**9**	**8**	**12**	**15**	**13**	**11**	

Suppression du minimum:

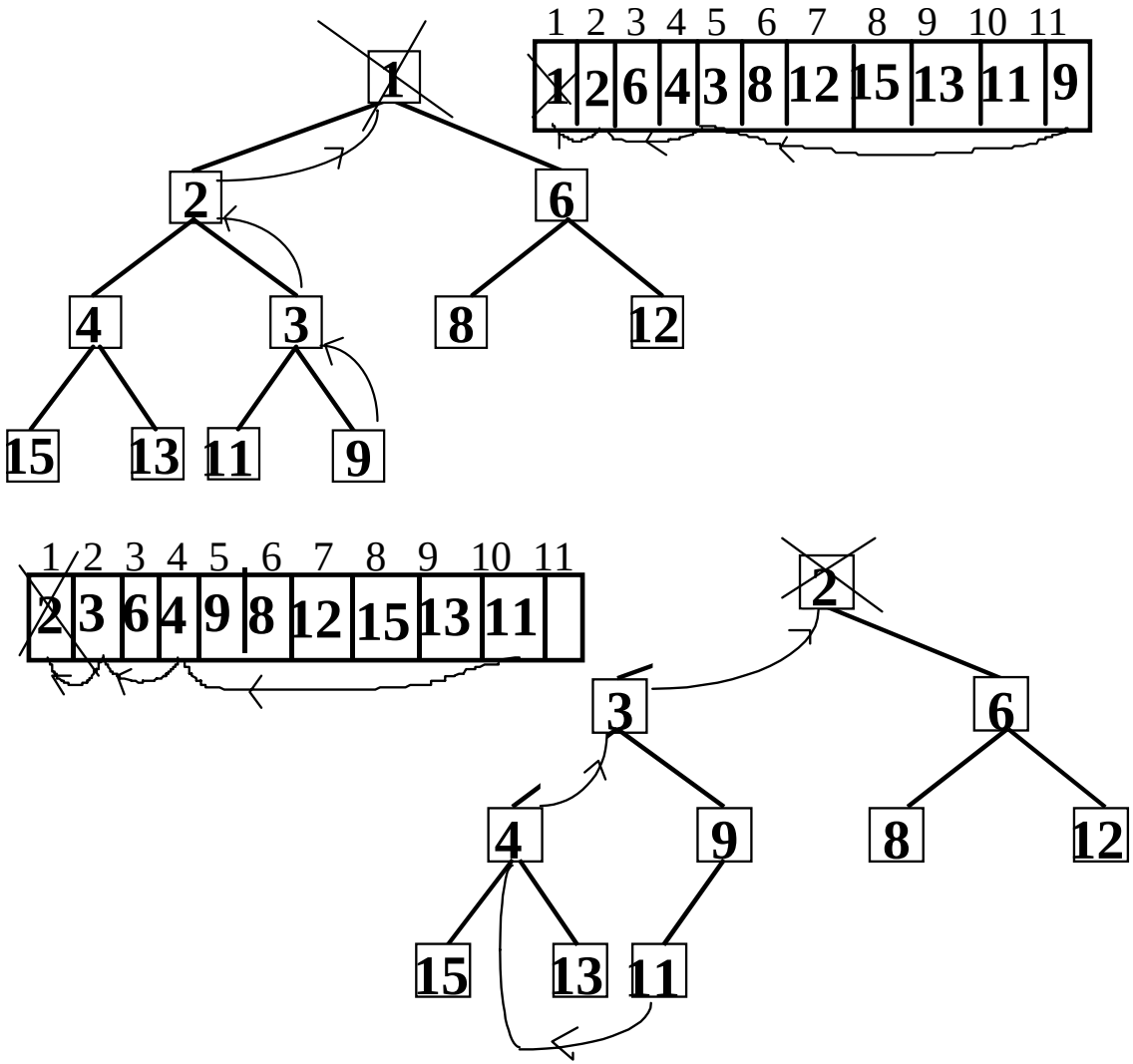
*EXEMPLE (suite) suppression de **2***

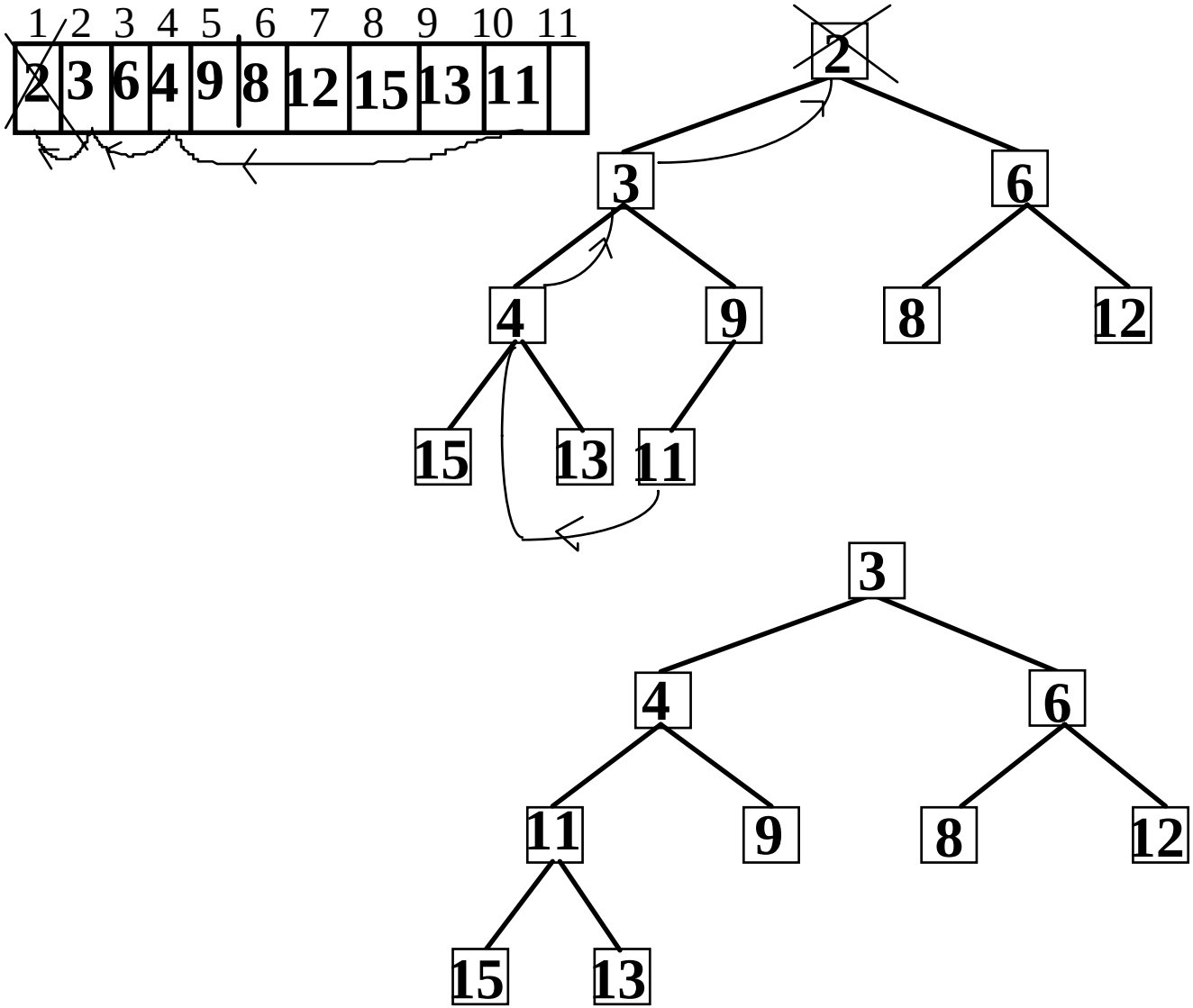
*derclé = **11***



0	1	2	3	4	5	6	7	8	9
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

```
public void supprimer_min(  ){

        fin=fin-1;
        int x=tab[fin]; // c'est x que nous devons reclasser
        int pere =0;
         int fils;
        while (true){
                //test si fils gauche existe
                if ((2*pere+1) < fin) fils = 2*pere+1;
                   else break;
                //test si fils droit existe et est plus petit
                if ((2*pere+2) < fin)
                        if (tab[2*pere+2] <= tab[2*pere+1] )
                                fils = 2*pere+2;

                if (x <= tab[fils]) break;
                tab[pere]=tab[fils]; //on remonte tab[fils]
                pere= fils; //on descend pere
        }
        // x <= tab[fils]
        tab[pere] = x;
}
```

Ici aussi, 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 dans le cours sur les tris)