NFP136 – ARBRES BINAIRES ET TAS

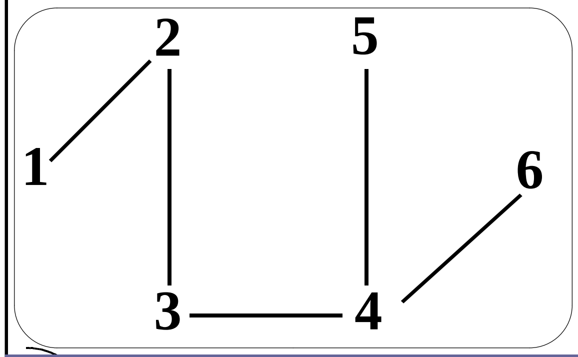
PLAN

- **Définitions**
- **Représentation des arbres (généraux)**
- **Représentation des arbres binaires**
- **Tas**
- **ABR et AVL**

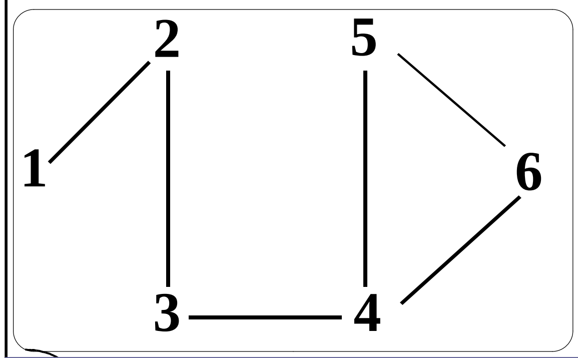
Arbres : définitions

Arbre :

= structure qui contient des éléments (nœuds) reliés entre eux par des « traits », de telle façon qu'entre toute paire de nœuds il existe une et une seule façon d'aller d'un nœud à l'autre en suivant les traits.



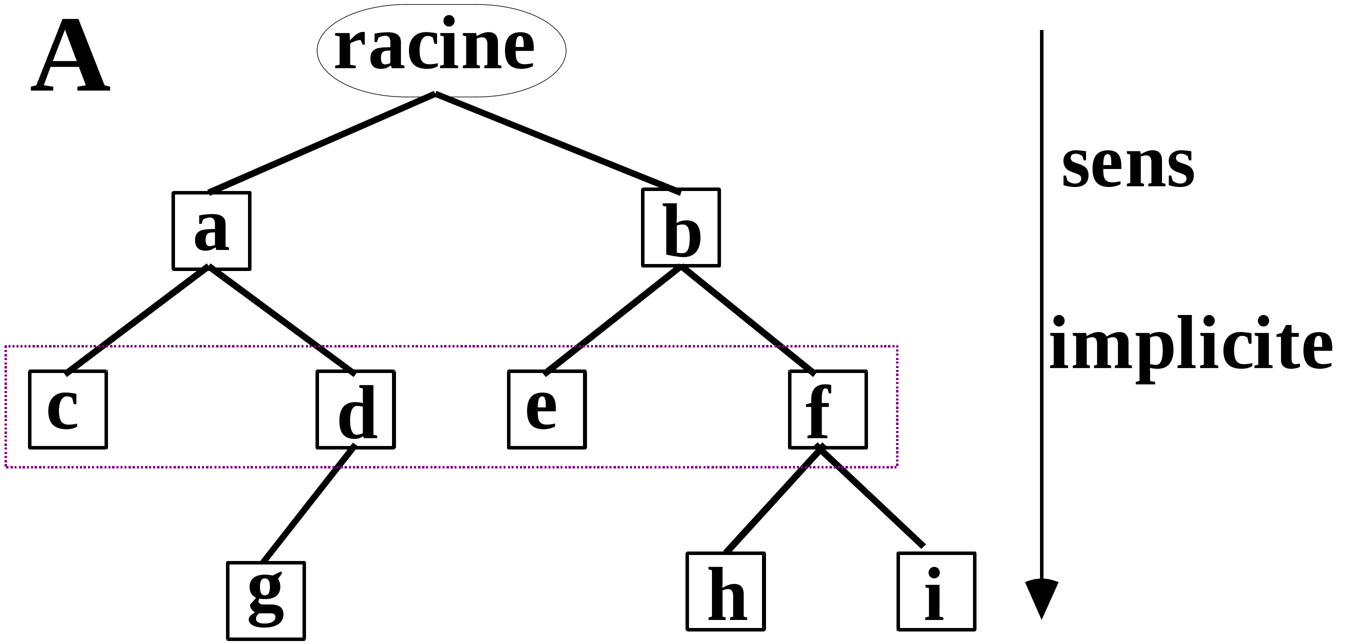
Un arbre à 6 nœuds



Pas un arbre !

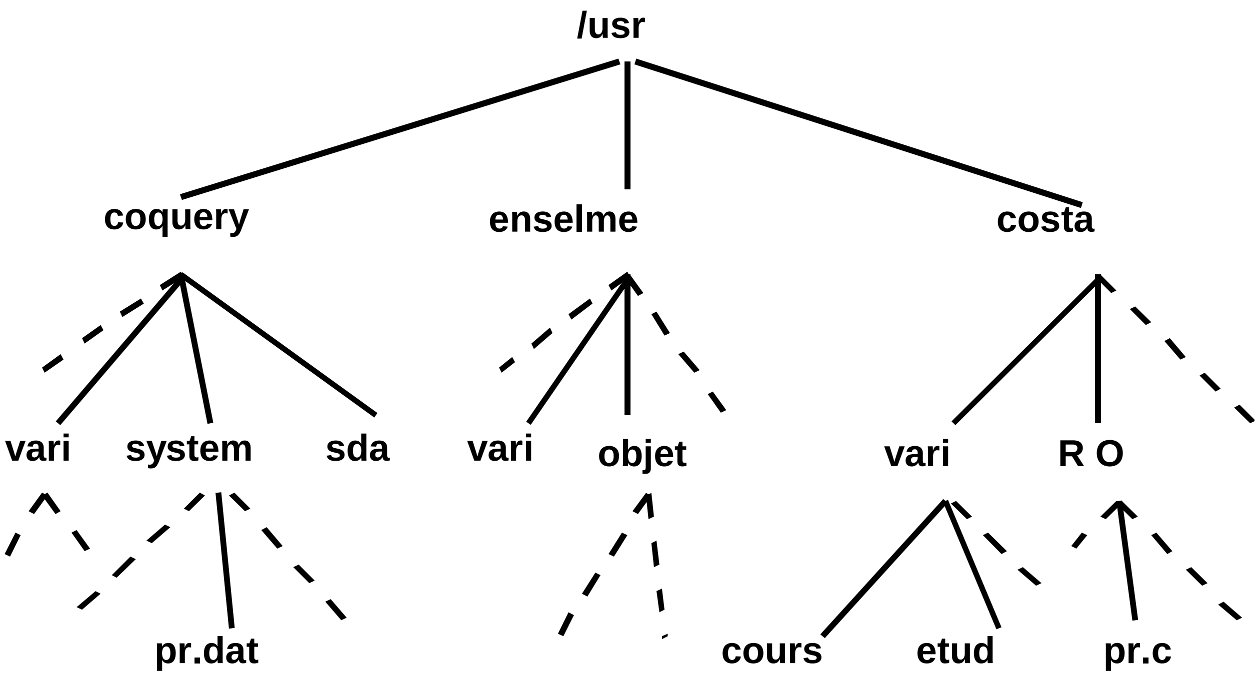
En informatique, ce qu'on appelle un arbre est souvent considéré comme « enraciné » en un nœud *racine* (en haut), et tous les traits sont implicitement orientés de haut en bas : on parle alors d'arborescence (*rooted tree*, en anglais).

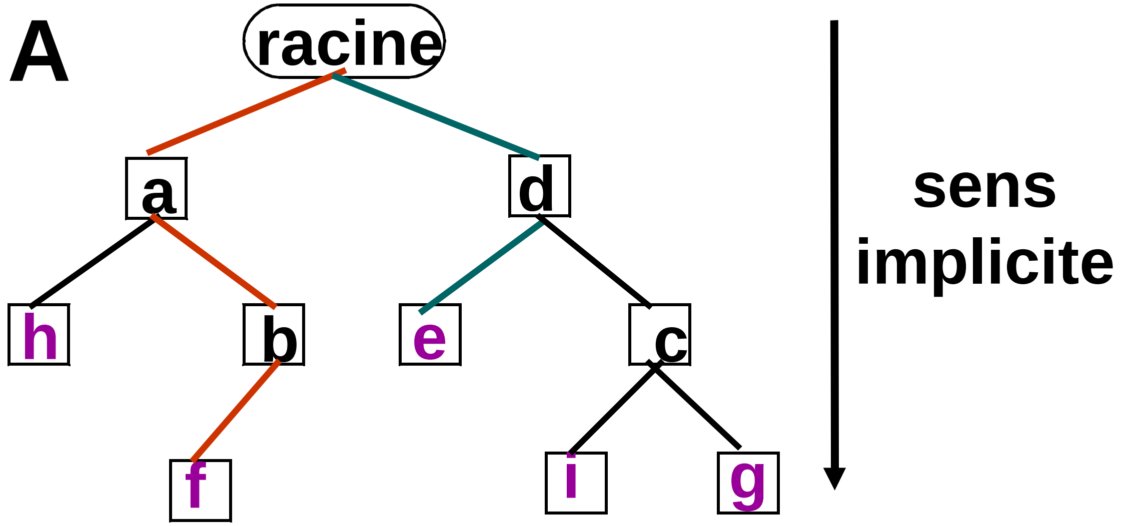
Niveau (ensemble de nœuds séparés de la racine par le même nombre de traits)



Ex. : arbre généalogique, arbre phylogénétique, etc.

EXEMPLE : **répertoires sous UNIX/LINUX**

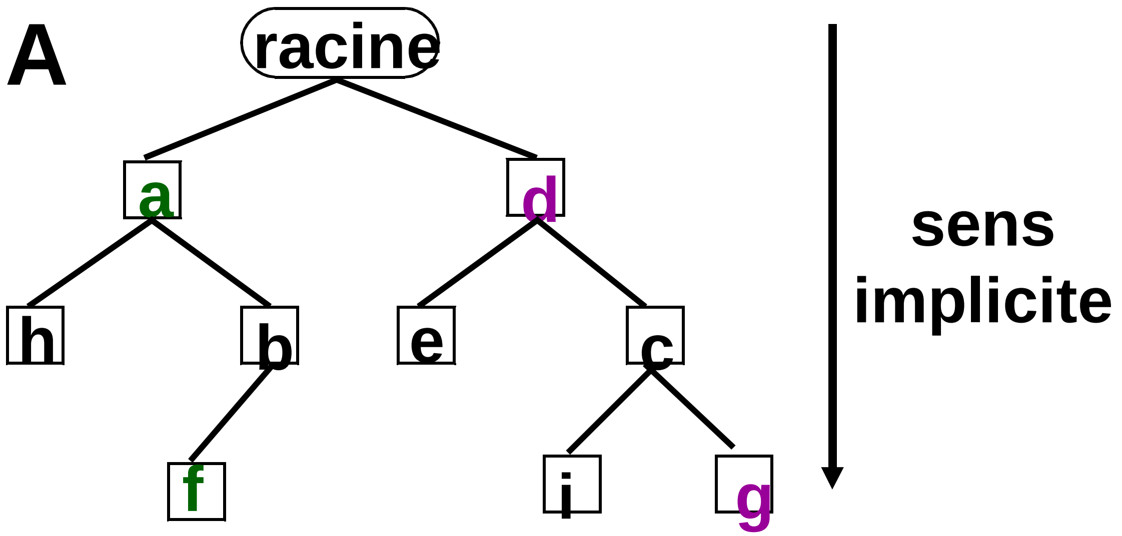




Feuille : **nœud sans nœud en-dessous** (h,f,e,i,g)

Branche : **ensemble des traits reliant la racine à une des feuilles** (*par exemple* : racine-d-e)

Hauteur **(ou profondeur), notée *h* : -1 + nombre de niveaux, donc longueur (en nombre de traits) de la plus longue branche** *(ex. : r-a-b-f ==> h(A)=3)*

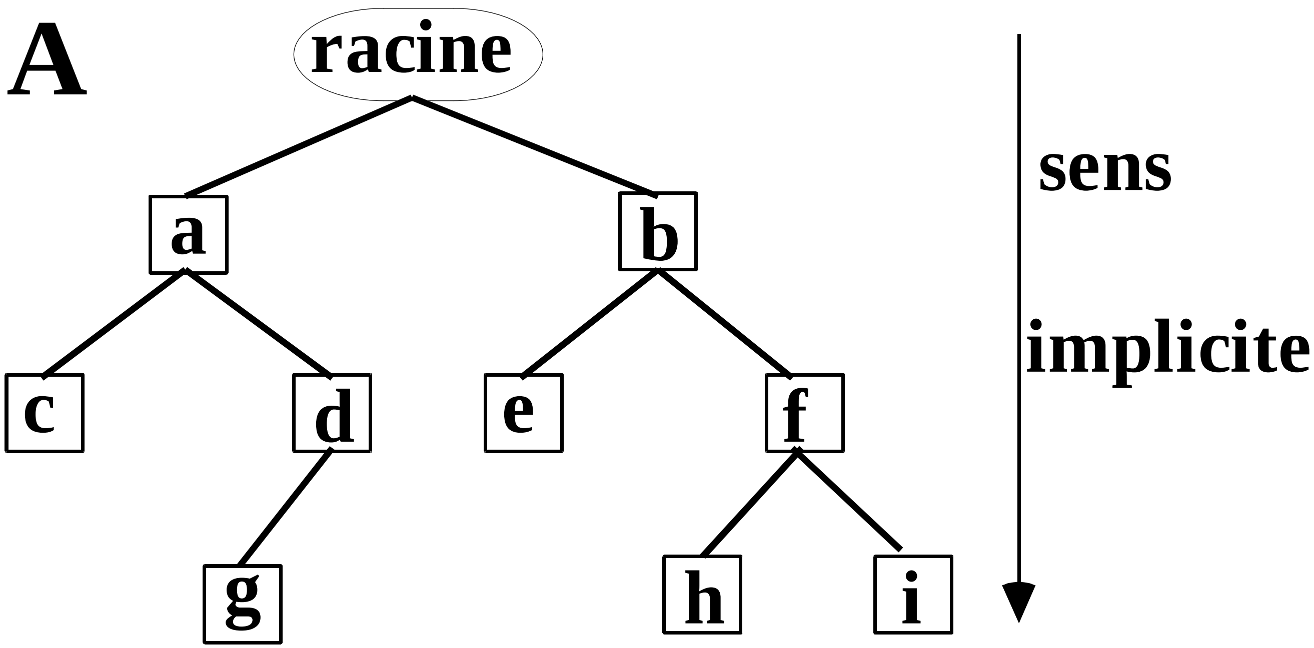


<u>Ascendant de x</u> : nœud au-dessus de x dans l'arborescence *(d ascendant de g)*

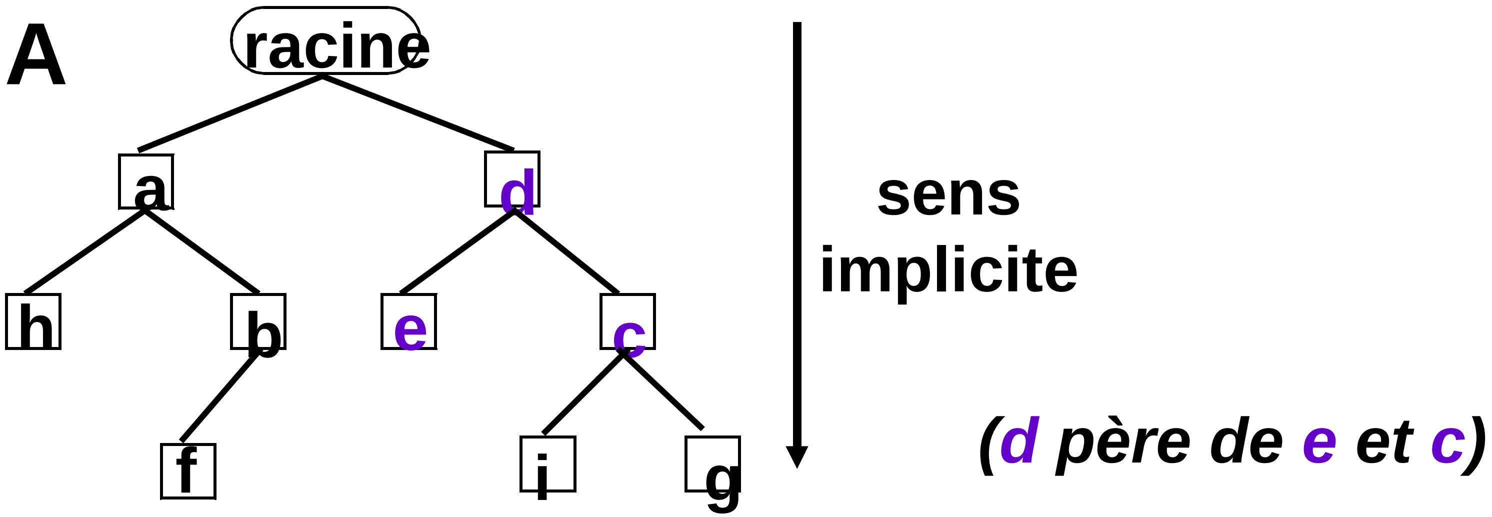
<u>Descendant de x</u> : nœud sous x dans l'arborescence *(f descendant de a)*

Qu'en est-il de e et i ?

Arborescence (arbre) *binaire* : chaque nœud a au plus deux nœuds sous lui qui lui sont reliés (appelés successeurs, ou fils), et dont il est le père.



(Par exemple, c et d sont des fils de a, mais pas g ni b)



Définition récursive d'un arbre binaire

ensemble vide

ou

ensemble formé

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

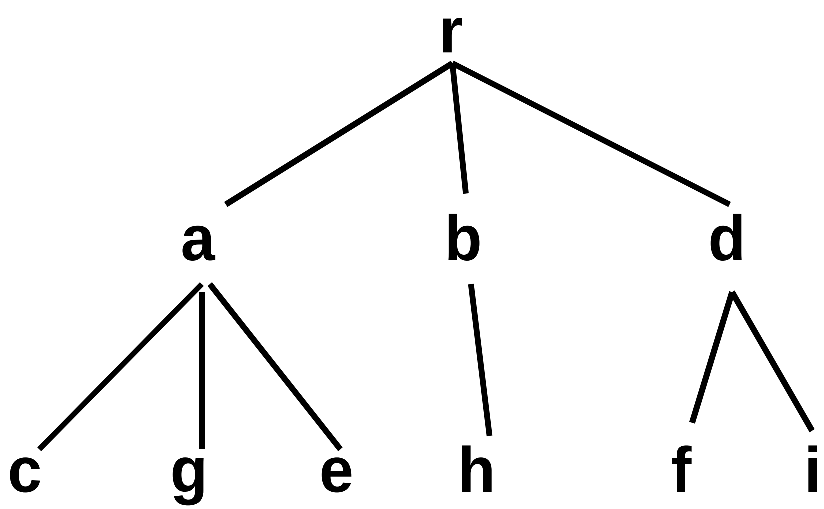
Représentation des arbres

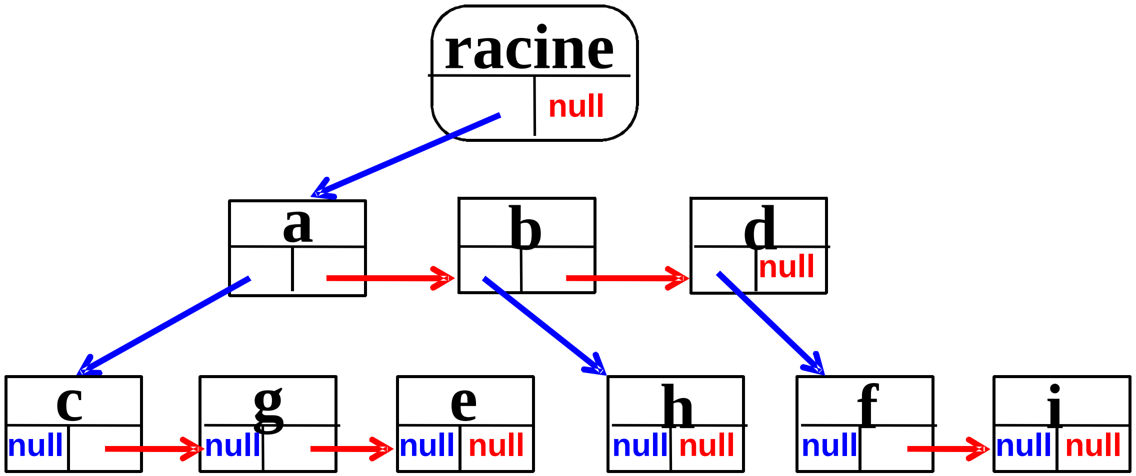
- **Représentation par un chaînage**

Un « nœud » d'un arbre peut être représenté par un objet qui contient 3 (ou 4) informations : une clé (identifiant), 2 adresses (références), et parfois une donnée (par exemple, un entier), qui peut être la clé.

c :	**clé (entier, chaîne)**
d :	**donnée (entier, chaîne, tableau, etc.)**
premier fils :	**adresse du fils gauche (null si n'existe pas)**
frère droit :	**adresse du frère droit (null si n'existe pas)**

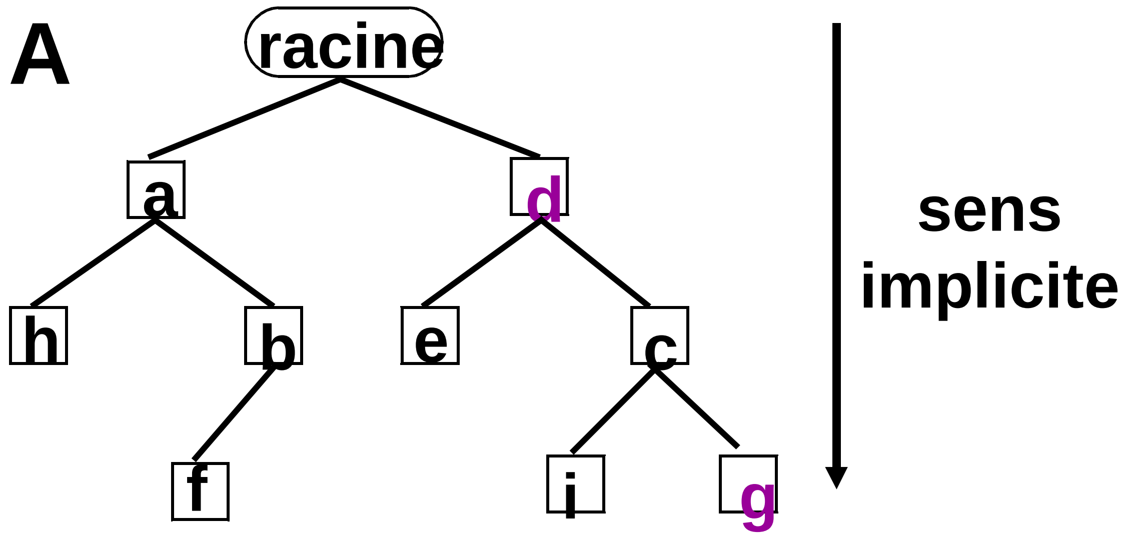
<u>clé=donnée</u>





Représentation et propriétés des arbres binaires

clé=donnée



REPRÉSENTATION (NIVEAU PAR NIVEAU, DE GAUCHE A DROITE)

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

REPRÉSENTATION ARBRES BINAIRES

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 et :**

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 (floor((4-1)/2)=)1*, c'est donc *a

REPRÉSENTATION DES ARBRES BINAIRES : QUEL BILAN ?

- **Représentation par un <u>tableau</u> de données :**

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

$\rightarrow$ **père en (i-1)/2 (arrondir inf.)**

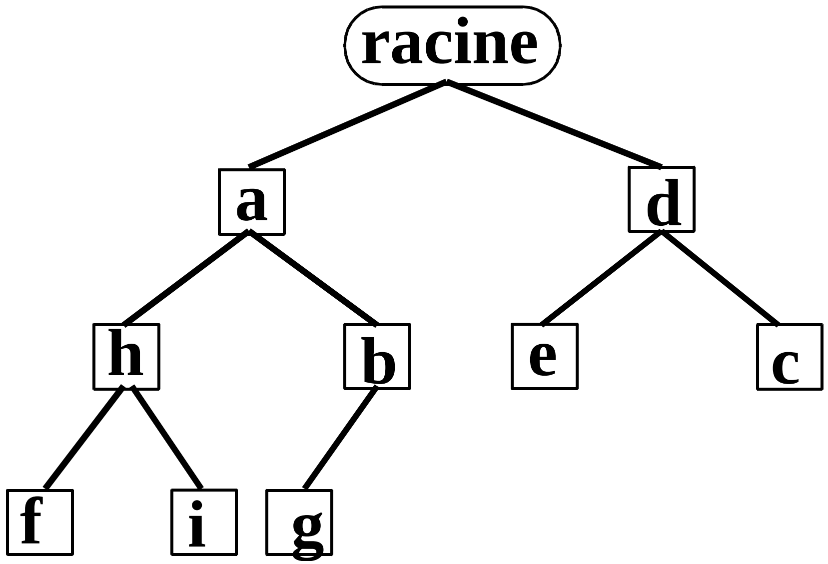
(Preuve par récurrence sur i.)

==> Exploration facile de l'arbre (par indices)

MAIS place mémoire potentiellement perdue

ARBRE BINAIRE PARFAIT

Un arbre binaire complet sur les h premiers niveaux (si h>0) + un dernier niveau où tous les sommets sont groupés sur la gauche.



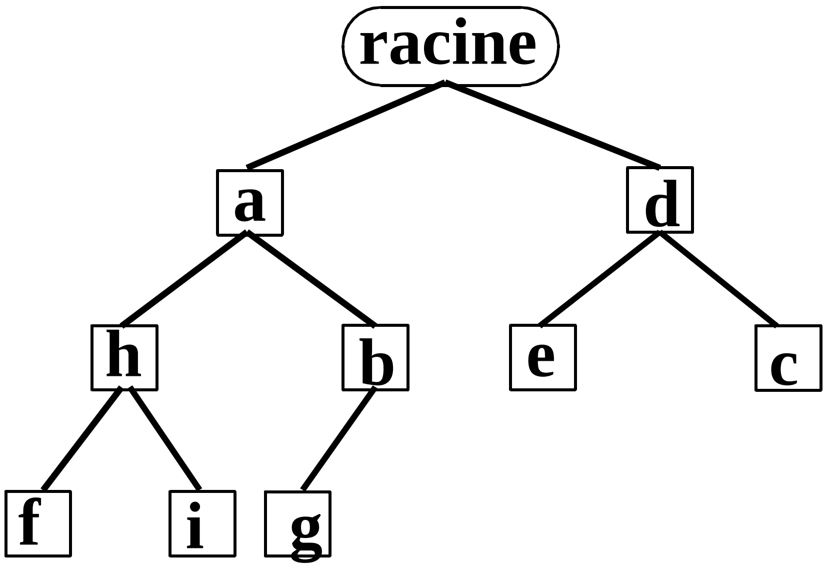
Où un arbre binaire **complet** est un arbre binaire où tout sommet a 2 fils, sauf ceux du dernier niveau (les feuilles).

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 « nœud » peut être représenté par un objet qui contient 3 (ou 4) informations : une clé (id.), 2 adresses (références), et parfois une donnée (par exempe, un entier), qui peut être la clé.

c : **clé (entier, chaîne)**

d : **donnée (entier, chaîne, tableau, etc.)**

gauche : **adresse du sous-arbre fils gauche (null s'il n'existe pas)**

droit : **adresse du sous-arbre fils droit (null s'il n'existe pas)**

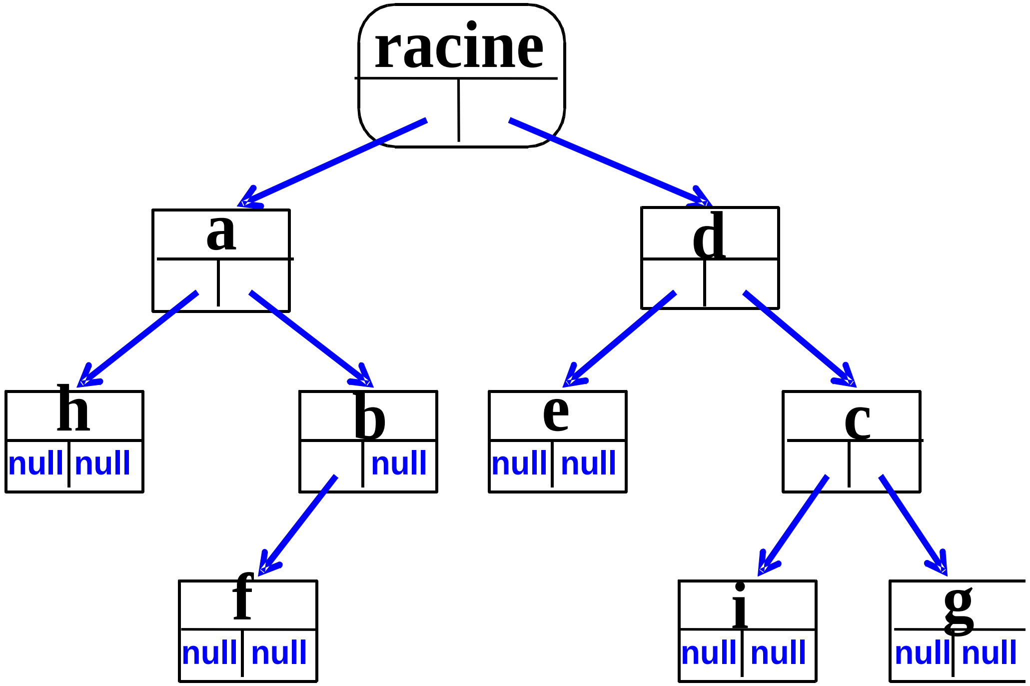
(*facultatif, pour remontée* : haut (adresse du père))

• Représentation par un chaînage en JAVA

```
public class Arbre { //on considère ici des arbres d'entiers
   private int cle;
   private int donnee; //cela pourrait être un char, ou autre
   private Arbre filsG;
   private Arbre filsD;

   public Arbre(int c, int d, Arbre g, Arbre d) {
    cle = c;
    donnee = d;
    filsG = g;
    filsD = d;
   }
}
```

- **Représentation par un <u>chaînage</u> : illustration**

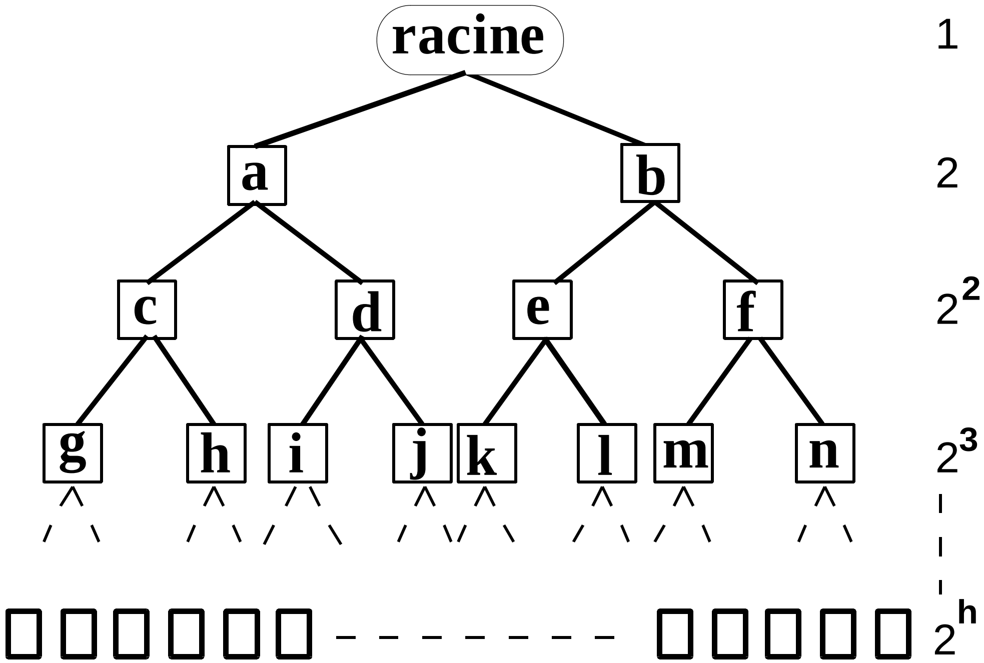


Exemple d'utilisation :

```
public static void main (String[] args) {
      Arbre A=new Arbre(1, 5, null, null);
      A = new Arbre(2, 4, A, null);
      A = new Arbre(3, 10, new Arbre(4, 3, null, null), A);
      Arbre Abis=new Arbre(5, 7, new Arbre(6, 8, null, null),
                                  new Arbre(7, 9, null, null));
      Abis=new Arbre(8, 2, new Arbre(9, 60, null, null), Abis);
      A = new Arbre(10, 15, A, Abis);
}
```

Quel arbre A obtient-on à la fin de ce programme ?

Hauteur d'un arbre binaire



Nombre de nœuds = N $\leq 1 + 2 + 2^2 + 2^3 + \ldots + 2^h$

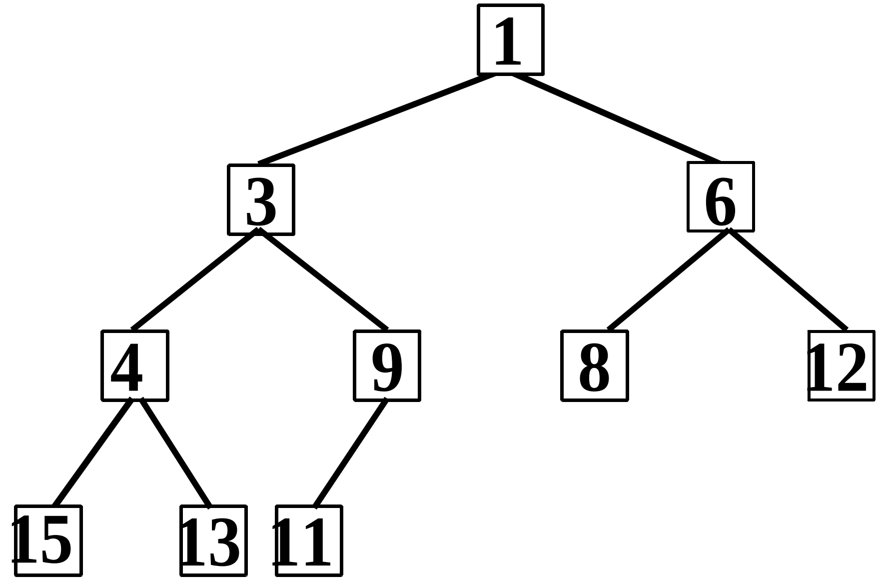
==> N $\leq 2^{h+1} - 1$, soit h $\geq \log_2(N+1) - 1$

(égalité si arbre binaire complet)

Les Tas

Définition d'un TAS

- **arbre binaire parfait (avec clé = donnée)**
- **+ valeur d'un nœud ≤ (≤ si tas min, ≥ si tas max) aux valeurs de tous ses descendants**



==> clé minimale ?

= valeur de la racine

= valeur en case 0 !

(si tas min)

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

Implémentation JAVA d'un Tas

```
public class Tas {
  private int[] tab;
  private int nbVal=0;
  //nb de valeurs, donc indice de la 1ère case libre

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

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

  public boolean estPlein() {return(nbVal==tab.length);}

  …
}
```

méthodes	**conditions**
int minimum()	tas non vide
supprimerMin()	tas non vide
inserer(int valeur)	tas non plein

Création d'un tas vide :

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

LES METHODES

```
public int minimum() {
/* retourne la plus petite valeur (celle de
  la racine) */
  return(tab[0]);
}
```

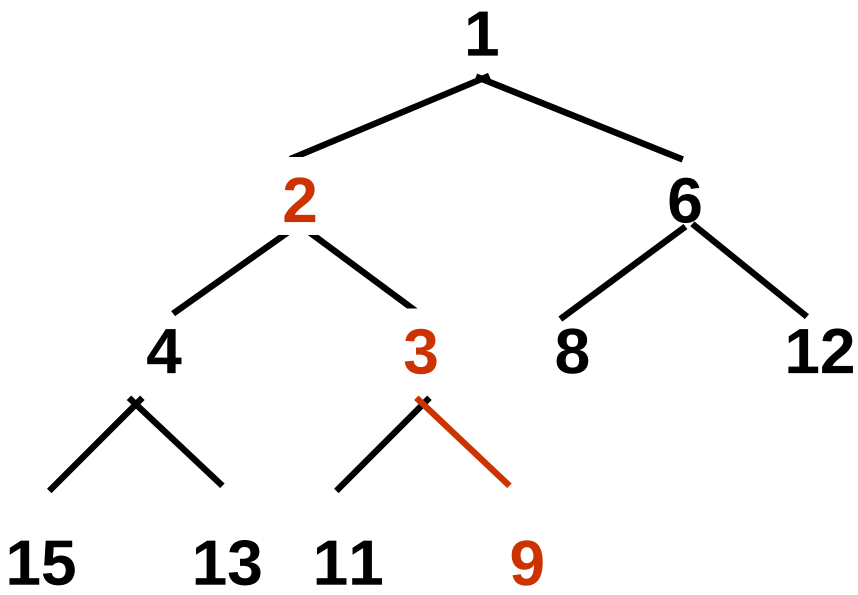
0	1	2	3	4	5	6	7	8	9
1	**3**	**6**	**4**	**9**	**8**	**12**	**15**	**13**	**11**

fonction en O(1)

Insertion :

Ajout d'une feuille à l'arbre, puis placement de la clé de façon à garder la structure de tas (la clé « remonte »)

EXEMPLE : **min tas = 1 ==> ajout de la clé 2**

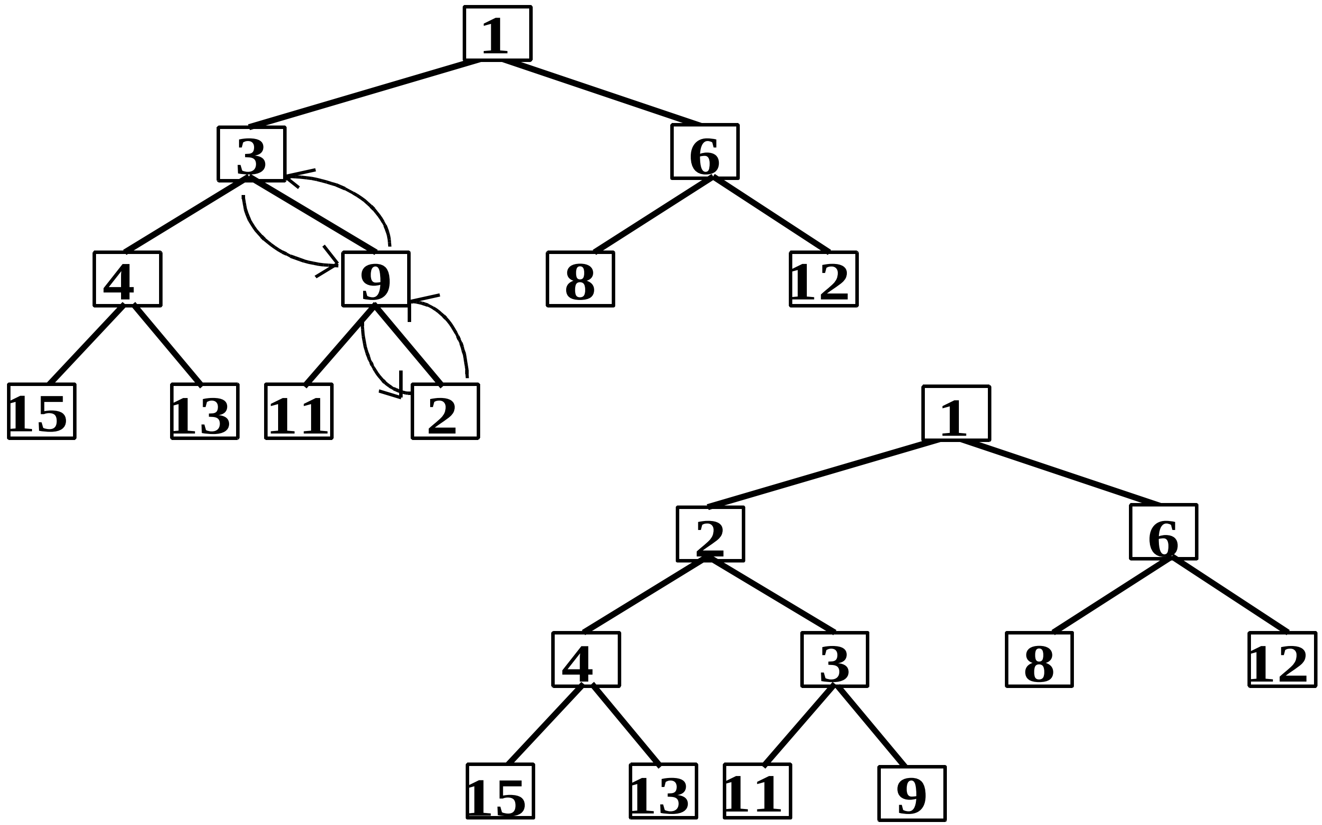


0	1	2	3	4	5	6	7	8	9	10
1	**2**	**6**	**4**	**3**	**8**	**12**	**15**	**13**	**11**	**9**

Insertion :

Ajout d'une feuille à l'arbre, puis placement de la clé de façon à garder la structure de tas (la clé « remonte »)

EXEMPLE : **min tas = 1 ==> ajout de la clé 2**



```
public void inserer(int val) {
  int fils=nbVal, pere=(fils-1)/2;
  tab[fils]=val;
  while(pere>=0 && val<tab[pere]) {
     tab[fils]=tab[pere];//père descend
     tab[pere]=val;//val remonte
     fils=pere;
     pere=(fils-1)/2;
  }
  nbVal++;
}
```

```
public void inserer(int val) {
  int fils=nbVal, pere=(fils-1)/2;
  tab[fils]=val;
  while(pere>=0 && val<tab[pere]) {
      tab[fils]=tab[pere];//père descend
      tab[pere]=val;//val remonte
      fils=pere;
      pere=(fils-1)/2;
  }
  nbVal++;
}
```

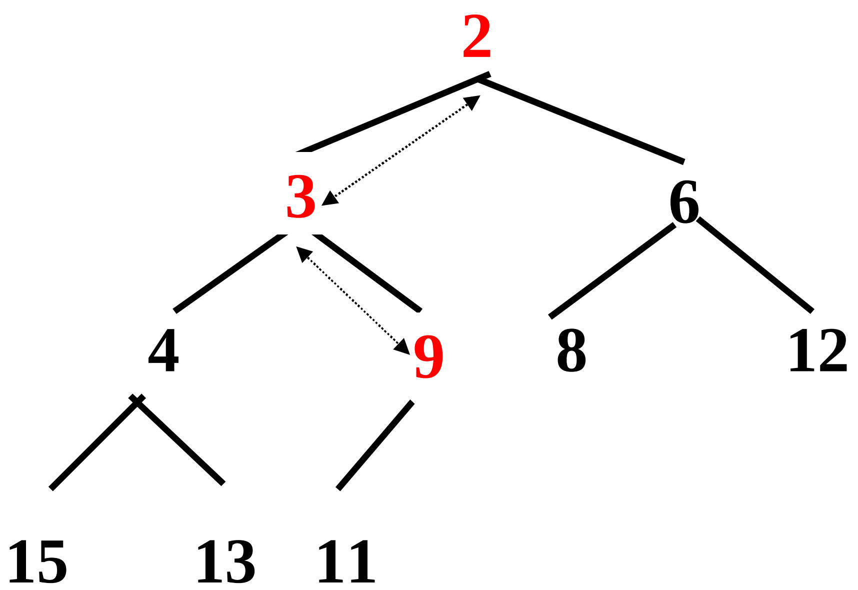
Au pire, nombre de passage dans la boucle=hauteur h de l'arbre

==> méthode en O(h)

Suppression du minimum :

Disparition de la dernière feuille (« dernière clé ») de l'arbre, mise à la place de la racine, et 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 « dernière clé ».

*EXEMPLE : suppression de **1** ; on a alors dernière clé = **9***

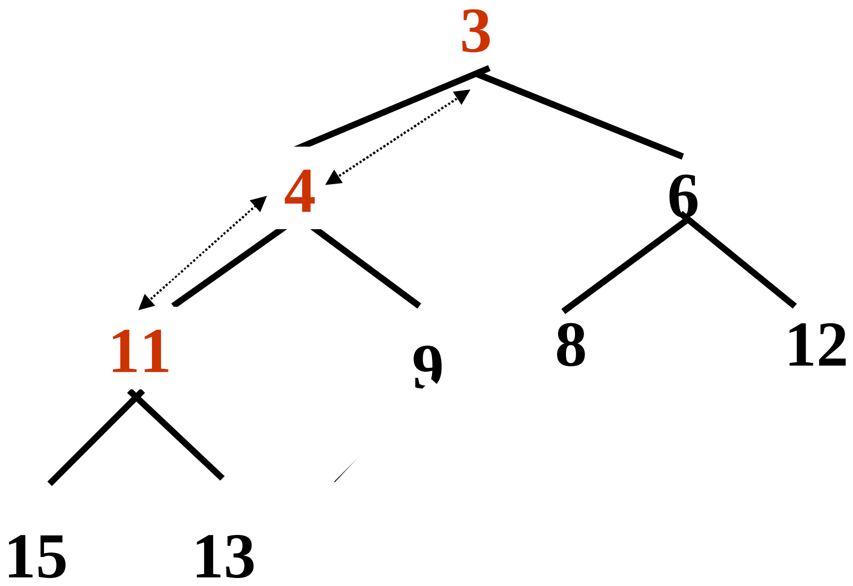


0	1	2	3	4	5	6	7	8	9	
2	**3**	**6**	**4**	**9**	**8**	**12**	**15**	**13**	**11**	

Suppression du minimum :

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

On a alors dernière clé = ***11***



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

```
public void supprimerMin() {

   nbVal--; //on diminue de 1 le nombre d'éléments
  int fils, pere=0, val=tab[nbVal]; //val = « dernière clé »
   tab[0]=val; //cette valeur est stockée à la racine
  do {
     fils=-1; //au début, pas de fils identifié de valeur < val
     if((2*pere+1)<nbVal) //si fils gauche existe
       fils=2*pere+1; //le meilleur fils courant est le gauche
     if(fils!=-1 && (2*pere+2)<nbVal && tab[2*pere+2]<tab[fils])
     //si fils droit existe et a 1 valeur + petite que le gauche
       fils++; //le fils ayant la valeur min est le droit
     if(fils!=-1 && val>tab[fils]) { //si val > min des 2 fils
       tab[pere]=tab[fils]; //le min des 2 fils remonte
       tab[fils]=val; //val descend
       pere=fils; //le sommet contenant val est maintenant fils
     }
     else
       fils=-1; //aucun fils, ou aucun de valeur < val
 } while(fils!=-1); //stop si feuille ou bien val <= aux 2 fils
}
```

Ici aussi, au pire, nombre de passage dans la boucle = hauteur de l'arbre h ==> méthode en O(h)

MAIS, dans un tas, les h premiers niveaux forment un arbre binaire complet, donc $N \geq 2^h - 1$, et on en déduit : $\log_2(n+1) - 1 \leq h \leq \log_2(n+1)$

Insérer un élément dans un tas ou supprimer son minimum se fait donc en O(h) = O(log n) opérations, ce qui en fait une structure très intéressante *en cas d'accès fréquent au minimum*

Un exemple d'application : le tri par tas

Principe :

- **Transformer le tableau à trier en tas,**
- **Extraire un à un les éléments min (racines) en conservant la structure de tas, et les stocker dans cet ordre dans le tableau trié.**

void **triParTas (tableau d'entiers tabATrier)**

entier n=tabATrier.length; **Tas unTas = new Tas(n)**;

début

pour i **= 0** à **n-1** faire *//construction du tas associé à tabATrier*

unTas.inserer(tabATrier[i]); *//O(log n) pour chaque i*

fait;

pour i**=0** à **n-1** faire

//on sélectionne un par un les minimums successifs du tas, qui sont mis à leur place dans le tableau : bonne version du tri par sélection

tabATrier[i]=unTas.minimum(); *//O(1) pour chaque i*

//suppression du minimum en gardant la structure de tas

unTas.supprimerMin(); *//O(log n) pour chaque i*

fait;

fin

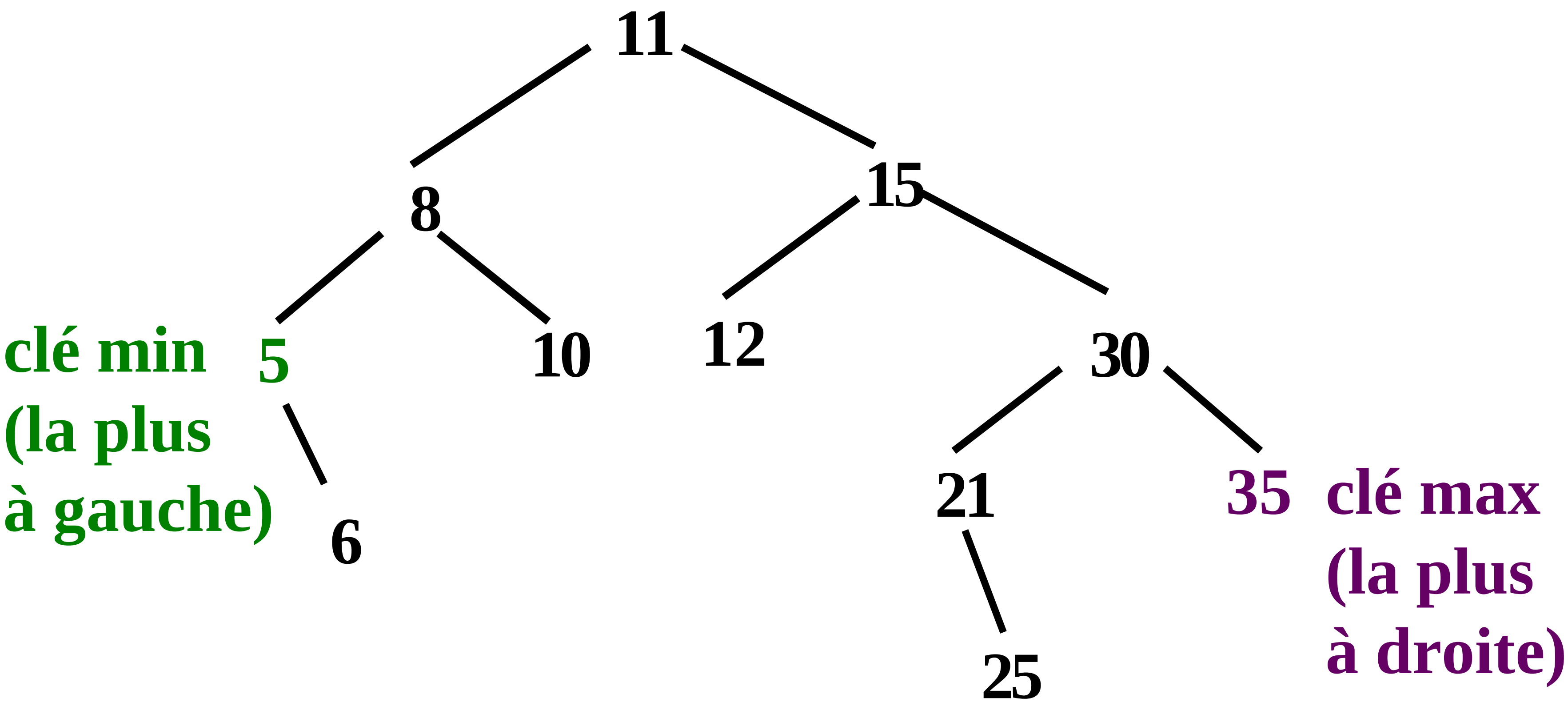
==> Complexité du tri par tas = O(n log n)

ARBRE BINAIRE DE RECHERCHE

Définition

Arbre binaire tel qu'en tout nœud la clé (valeur) du nœud est supérieure à celle de tous ses descendants de gauche et inférieure à celle de tous ses descendants de droite.

EXEMPLE D'ABR (arbre binaire de recherche) :



PRESENTATION DES ABR

ABR : **structure de données pour stocker de gros volumes d'informations, éventuellement de taille variable (structure dynamique)**

Arbre binaire A

A.g = sous-arbre de gauche ayant A pour racine

A.d = sous-arbre de droite ayant A pour racine

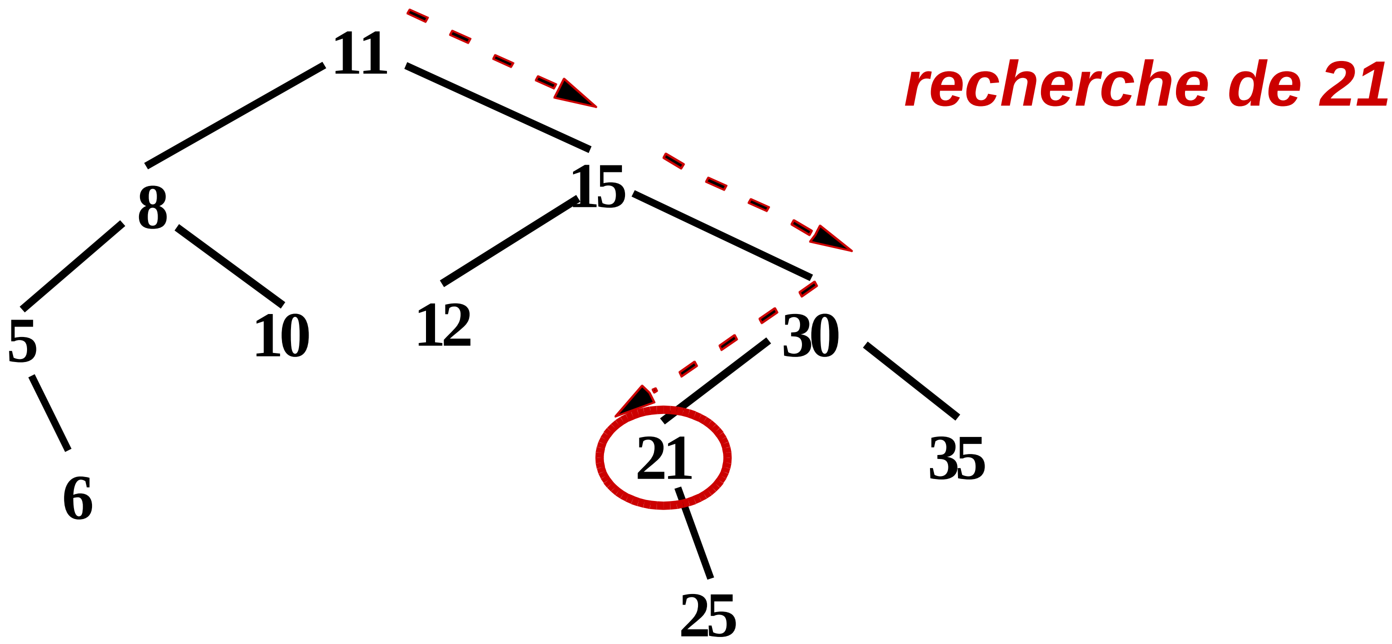
Clés **une donnée $\rightarrow$ une clé (par exemple, un entier)**

{clés} : ensemble totalement ordonné

Recherche :

Descente (récursive) dans l'arborescence, avec comparaison en chaque nœud

EXEMPLE :



Principe de la recherche de la donnée d (de type String) de clé c dans l'arbre A

précondition : c$\in$A

recherche(c, A) renvoie String

si **c == A.cle** alors

    retourner (A.donnee) *//donnée de la racine de A*

sinon

    si **c < A.cle** alors

        retourner (recherche(c, A.gauche))

    sinon **retourner (recherche(c, A.droit))**

    finsi

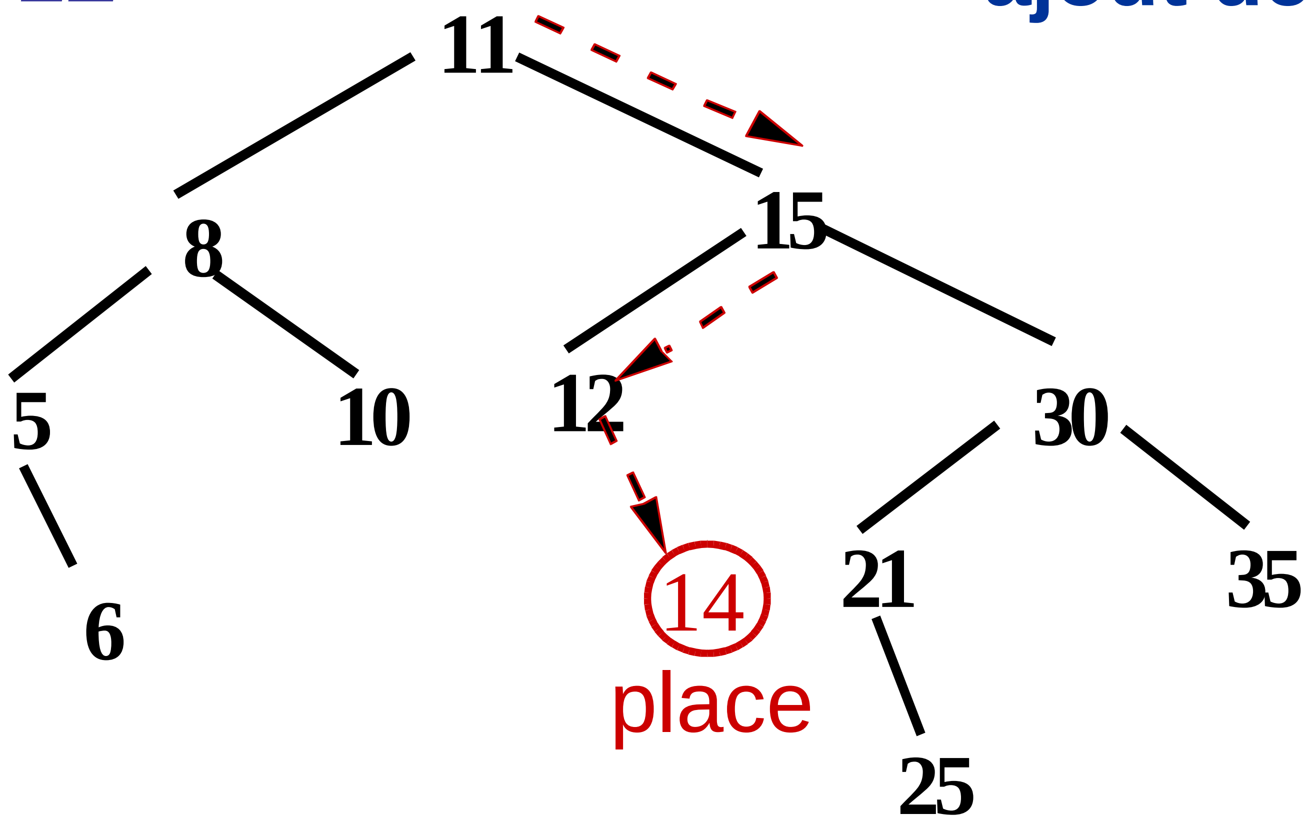
finsi

==> Complexité = O(h(A))

AJOUT D'UN ELEMENT

EXEMPLE

ajout de 14



==> Complexité = O(h(A))

Bilan sur les arbres binaires de recherche

- **Recherche,**
  ajout,
  ET <u>suppression</u> (non détaillée) :

  complexité = O(h(A))

  avec $\log_2 n \leq h(A) \leq n-1$

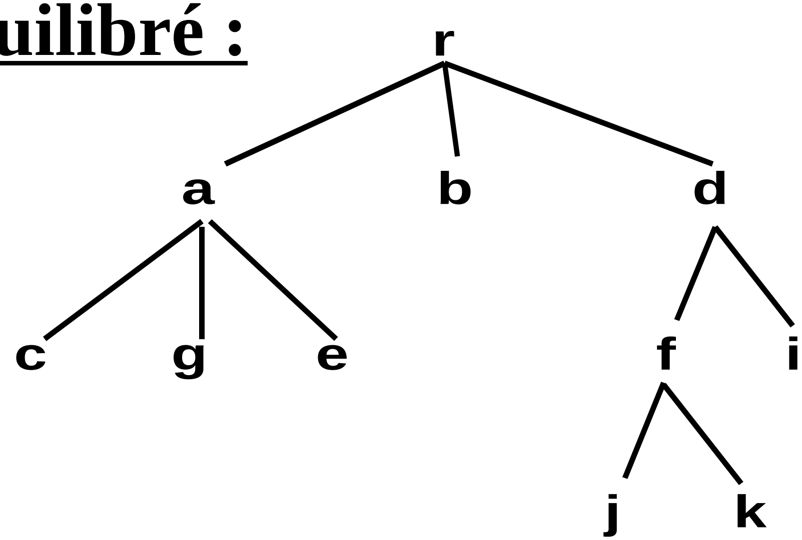
  (complexité en moyenne = O(log n))

- **Problème : éviter les cas défavorables**

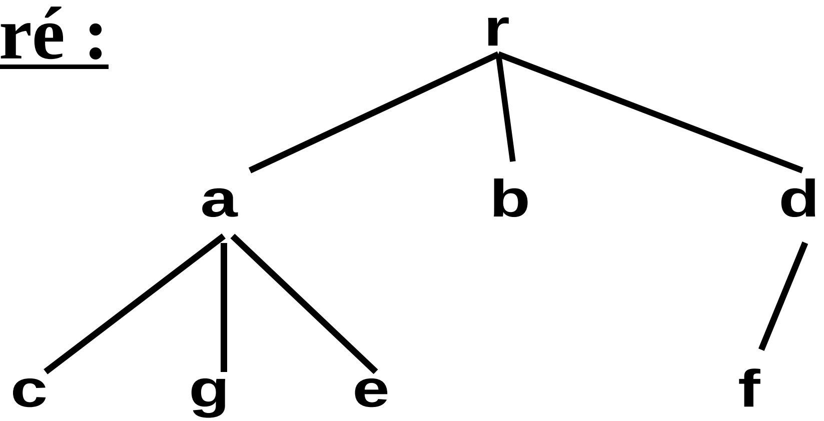
  $\rightarrow$ arbres « équilibrés »

Arbres équilibrés ou non

arbre non équilibré :



arbre équilibré :

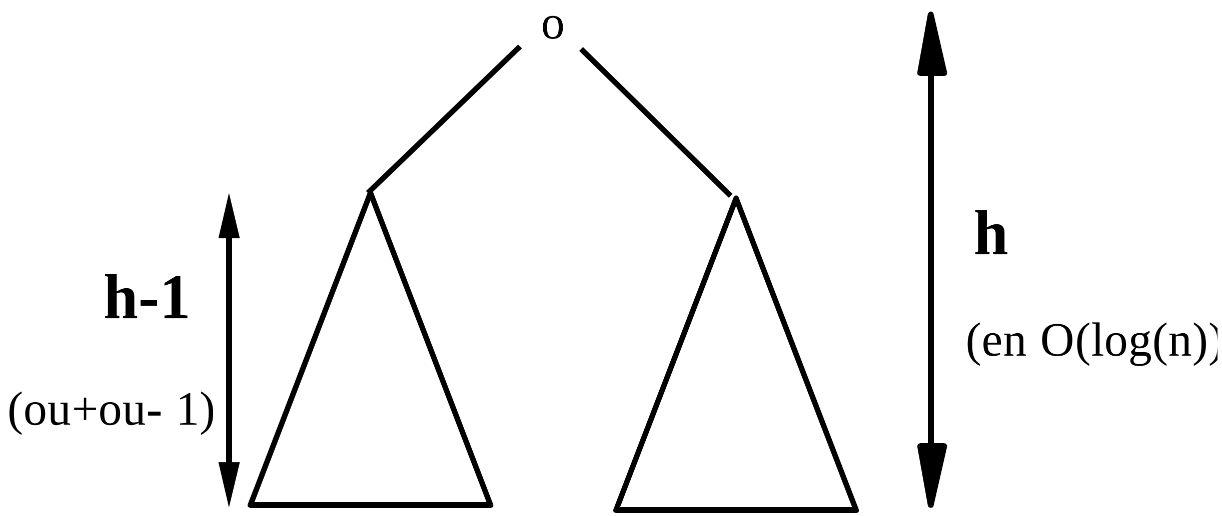


==> les feuilles sont sur au plus 2 niveaux

Arbre AVL (du nom de ses concepteurs !) :

ABR tel qu'en tout nœud la différence de hauteur entre les sous-arbres gauche et droit est au plus de 1

$\Rightarrow$ ***h = O(log n) dans un arbre AVL, donc la recherche dans ces arbres est en O(log n) !***



Problème : maintenir l'équilibre !

REEQUILIBER UN AVL

Rotation : opération permettant de rééquiliber un AVL.

Principe des opérations dans un AVL :

- **Opérations recherche, ajout et suppression faites comme dans un arbre binaire de recherche,**
- **Puis, si besoin, rééquilibrage de l'arbre (par des rotations) pour obtenir un AVL en O(log n).**

Bilan :

Recherche, ajout, suppression en O(log n) dans un AVL !

CONCLUSION : TAS ET AVL

TAS :

+ rechercher le minimum (ou maximum, selon le tas)

+ supprimer le minimum (ou maximum, selon le tas)

+ insérer un élément

- rechercher un élément quelconque

- supprimer un élément quelconque

AVL :

+ rechercher, insérer ou supprimer un élément quelconque

- rechercher le minimum