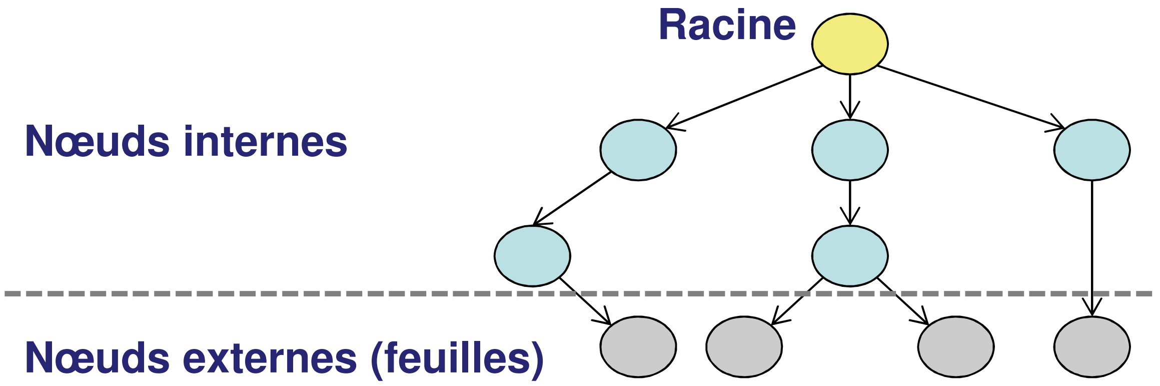
PLAN

Partie II. Structures de données et Algorithmes

II. 2. Structures de données arborescentes

2.1. Introduction

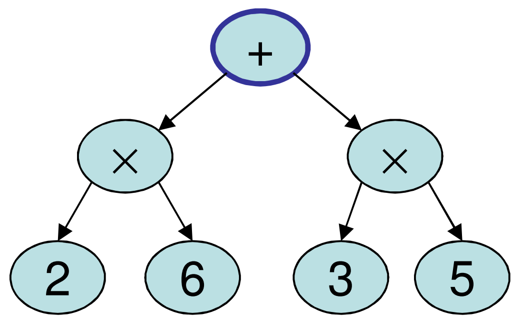
- **Arbre**
  - Type abstrait de données contenant des entités en relation
  - **Chaque entité peut avoir entre 0 et *n* successeurs**
  - **Chaque entité a au plus un prédécesseur**
  - Entité = nœud de l'arbre
  - Entités particulières
    - la racine (pas de prédécesseur)
    - les feuilles (pas de successeurs)
  - Un seul chemin de la racine vers tout autre sommet



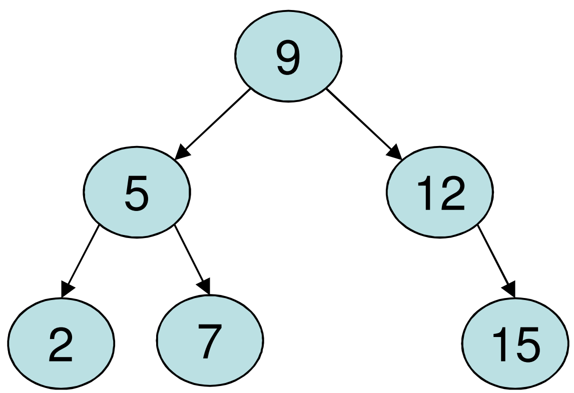
- **Opérations de base**
  - **Opérations d'accès**
    - **Accéder à un élément**
    - **Tester si la structure vide**
  - **Opérations de construction**
    - **Initialiser : obtenir une structure « vide »**
    - **Insérer un élément**
  - **Opérations de modification**
    - **Enlever un élément**

- **Différentes variantes d'arbres**
  - **binaires** (n =2)
  - ***n*-aires** (n quelconque)
  - **ordonnés**
  - **non ordonnés**

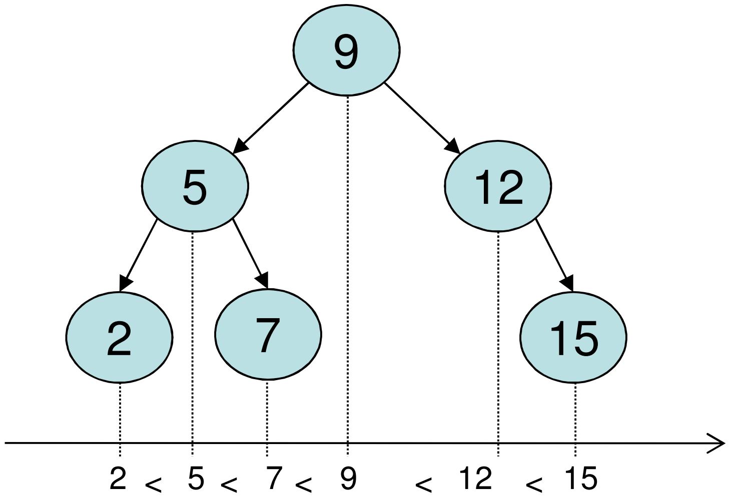
- **Arbre binaire**
  - Chaque nœud a 2 successeurs au maximum
    - **Exemple**

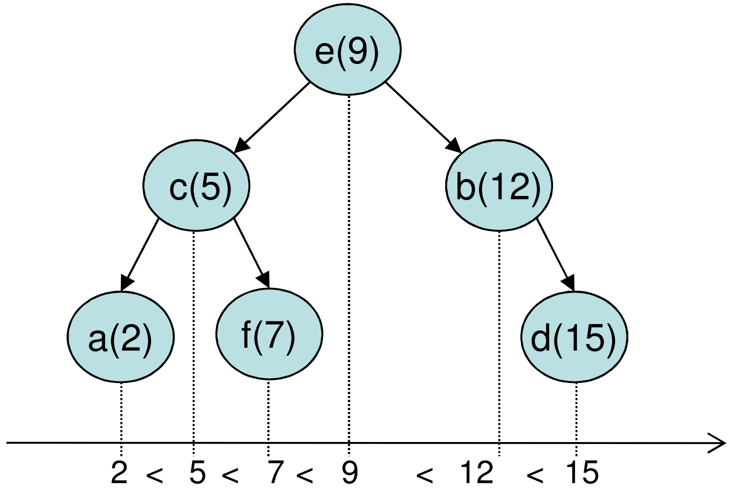


- **Arbre binaire de recherche**
  - **Arbre binaire**
  - **Ordonné**
    - Relation d'ordre entre les éléments contenus dans les sommets



- **Eléments et clés**
  - à chaque élément (pas forcément un nombre) ➔ une clé (la clé = nombre)
  - la clé peut être l'élément lui-même (si c'est un nombre)
  - ∃ relation d'ordre entre les clés



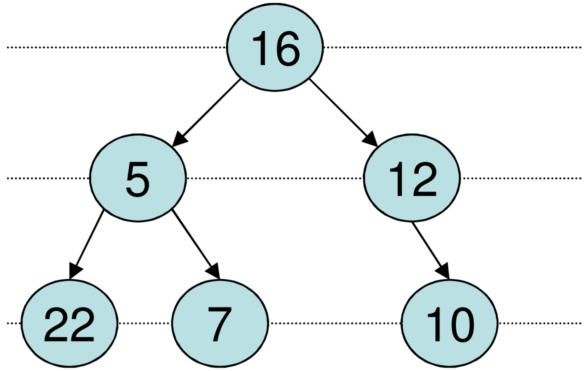


- **Propriété d'un ABR**
  - Tout sous-arbre (r, G, D) d'un ABR respecte les relations d'ordre suivantes:
    - Toutes les (clés des) éléments des nœuds de G sont inférieur(e)s à r
    - Toutes les (clés des) éléments des nœuds de D sont supérieur(e)s à r

2.2. Définitions et propriétés générales

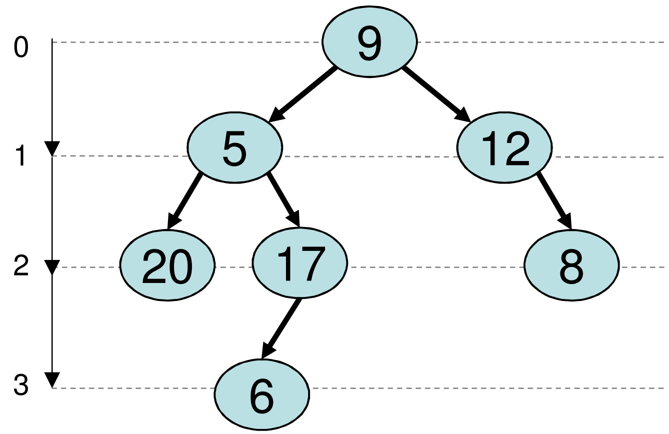
- **Définition récursive d'un arbre binaire**
  - **B est un Arbre Binaire** ssi :
    - c'est un arbre vide ***B* = ∅**

    ou bien
    - il est de la forme ***B* = (*r*, *G*, *D*)** avec
      - r : nœud racine de B
      - G et D sous arbres binaires disjoints issus de r (G : sous arbre gauche, D sous arbre droit)



- **Vocabulaire** :
  - Soit **B = (r, G, D) un arbre binaire**
    - une **feuille** = un nœuds sans fils (G et D sont vides)
    - une **branche** de B = un chemin allant du nœud racine r à une feuille
    - **nœuds internes** = nœuds de B possédant des successeurs
    - **nœuds externes** = nœuds de B sans successeurs

- **Hauteur d'un arbre binaire B = (r, G, D), notée h(B) :**
  - h(B) = distance maximale d'une feuille de B à la racine de B
  - **Exemple :**
    - 9 est à distance 0 de la racine
    - 5 et 12 sont à distance de 1
    - 20, 17 et 8 sont à distance de 2
    - 6 est à distance de **3**



⇒ h(B)= **3**

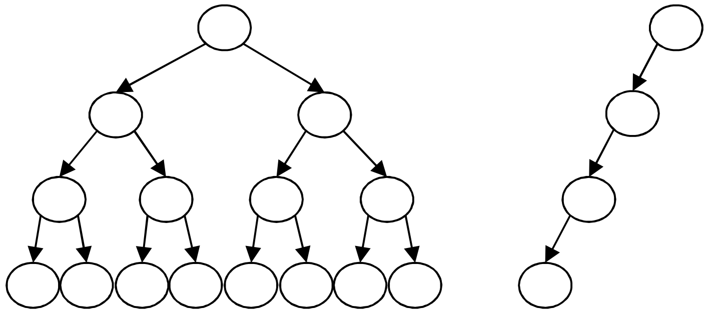
– **Définition récursive de la hauteur h(B) :**

Pour tout arbre B non vide

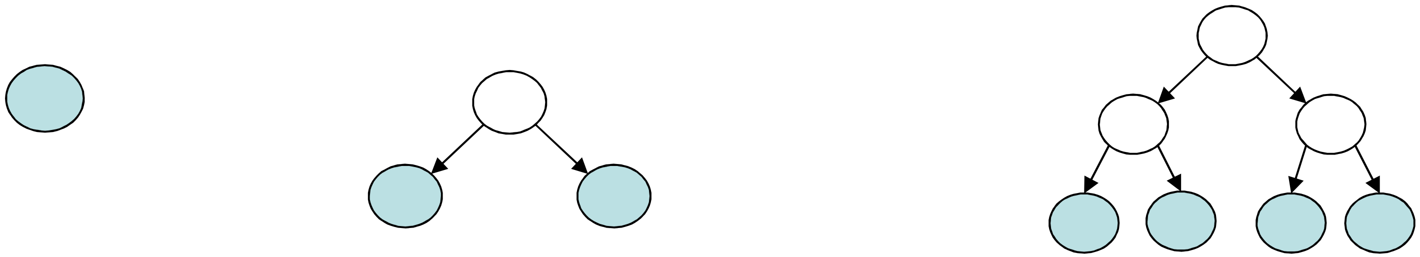
si B n'a aucun sous-arbre ……………….…….**h(B) = 0**

si B a 2 sous-arbres G et D non vides……… **h(B) = 1 + max(h(G), h(D))**

- Exemple d'arbres avec *n* minimal et *n* maximal pour *h* donnée



- Exemples d'arbres avec *n* maximal pour *h* donnée

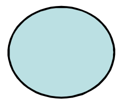


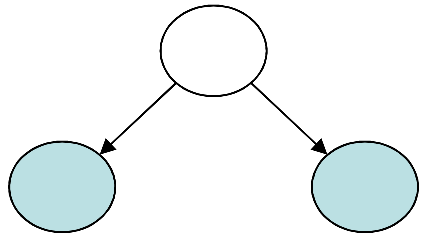
- **Relations entre hauteur *h* et taille *n* d'un arbre binaire**

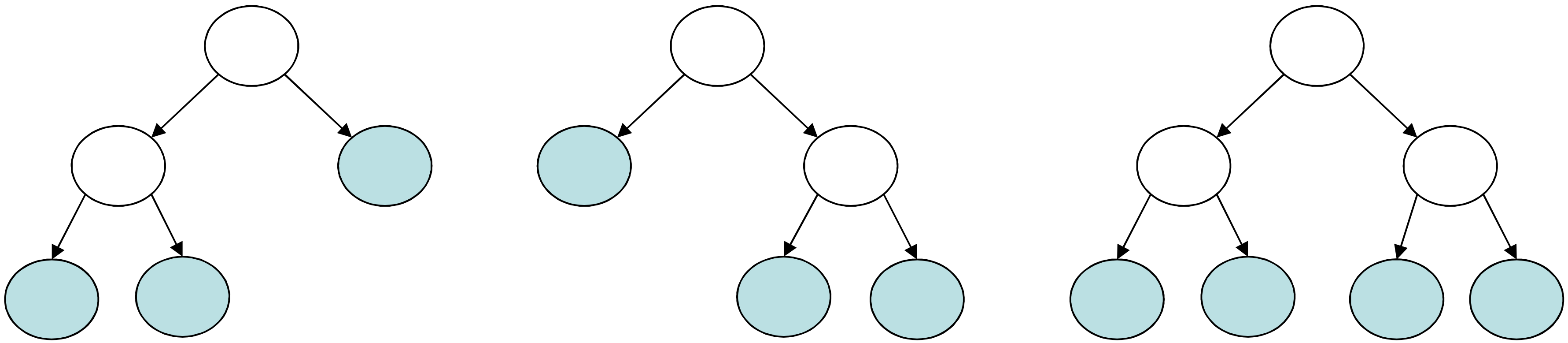
$h+1 \leq n \leq 2^{h+1} - 1$ $\log_2 n \leq h \leq n-1$

- **Arbre binaire complet**
  - B est un arbre binaire complet si :
    - il est non vide
    - chaque nœud possède 0 ou 2 fils
    - un arbre binaire complet ayant p nœuds externes (feuilles) a p-1 nœuds internes

Exemples d'arbres binaires complets
(en bleu = nœuds externes ou feuilles)







2.3.1 Opérations sur les Arbres Binaires

- **Types**
  - **`Arbre_Binaire`** et **`Element`** (information contenue en 1 nœud)

- **Opérations sur un arbre binaire**

  - **`Est_Arbre_Vide`**`(A : Arbre_Binaire)` ➔ `Booléen`
  - **`Racine`**`(A : Arbre_Binaire)` ➔ `Element`
  - **`Sous_Arbre_Gauche`**`(A : Arbre_Binaire)` ➔ `Arbre_Binaire`
  - **`Sous_Arbre_Droit`**`(A : Arbre_Binaire)` ➔ `Arbre_Binaire`
  - **`Appartient`**`(E : Element, A: Arbre_Binaire)` ➔ `Booléen`

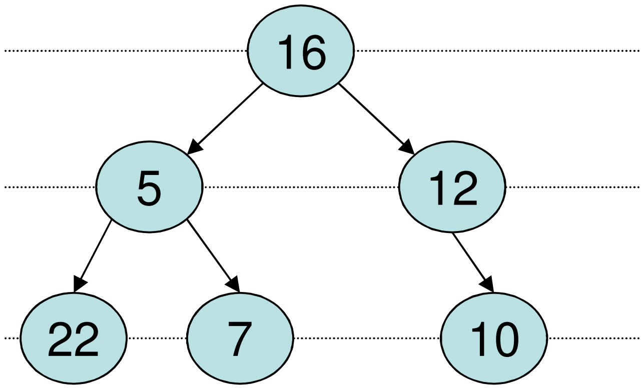
2.3.1 Opérations sur AB

- **Opérations de construction**
  - **`Creer_Arbre_Vide`** ➔ `Arbre_Binaire`
    - Construit un arbre binaire vide
  - **`Construire_Arbre`**`(E : Element; G, D : Arbre_Binaire)`
    ➔ `Arbre_Binaire`

    Renvoie un arbre construit à partir de E, G et D = arbre qui a :
    E comme racine, G en sous-arbre gauche et D en sous-arbre droit

- **Opérations de modification**
  - **`Supprimer_Racine`**`(A : Arbre_Binaire)` ➔ `Arbre_Binaire`
    - Retourne un arbre binaire privé de sa racine (∃ plusieurs solutions)

- **Exercice**
  - En utilisant les opérations définies ci-dessus, construire l’arbre suivant



  - Questions :
    - Quelles sont les autres possibilités de construction ?
    - Quels sont les types d’insertion possibles d’un élément dans un arbre ?

- **Autres opérations**
  - **Afficher**
    - Il faut passer en chaque nœud (= parcourir l'arbre)
    - Plusieurs parcours possibles => plusieurs types d'affichage
  - **Rechercher**
    - Nécessite une « Clé » (fonction Elément ➔ Clé)
    - Arbre binaire de recherche (en O*(h)* = O(log *n*))
  - **Insérer**
    - Ajouter un élément
    - à une position donnée : racine par exemple
    - dans un arbre binaire / un arbre binaire de recherche
  - **Supprimer**
    - Supprimer un élément dans un arbre binaire, dans un arbre binaire de recherche

2.3.2 Implémentation

- **Paquetage pour Arbres binaires (implémentation avec des pointeurs)**

```
generic
  type Element is private;

package Arbres_Binaires is
   type Arbre_Bin is private;


  -- spécifications des différentes opérations

  function Est_Arbre_Vide(A : Arbre_Bin) return boolean;
  function Racine(A : Arbre_Bin) return Element;
  function Sous_Arbre_Droite(A : Arbre_Bin) return Arbre_Bin;
  function Sous_Arbre_Gauche(A : Arbre_Bin) return Arbre_Bin;
  function Creer_Arbre return Arbre_Bin;
  function Construire_Arbre(E:Element; G,D:Arbre_Bin) return
                                                        Arbre_Bin;
  function Supprimer_Racine(A : Arbre_Bin) return Arbre_Bin;

   . . .
```

2.3.2 Implémentation

- **Implémentation avec des pointeurs(suite)**

```
 . . .

private

   type Lien is access Noeud;
   type Noeud is record
       Info : Element;
       Gauche, Droit : Lien;

   end record;

   type Arbre_Bin is record
       Racine : Lien;

   end record;

end Arbres_Binaires;
```

Option : hauteur, pere….

Option : nb_elements …

2.3.2 Implémentation

- **Implémentation d'un arbre binaire de recherche avec des pointeurs**

```
Hypothèse :    Cle = Element
(              ex : integer)
```

```
generic
   type Element is private;
   with function "<="(EG, ED : Element) return boolean;
package Arbres_Binaires_Recherche is
   type ABR is private;

   -- spécifications des différentes opérations
   -- idem package Arbres_Binaires
```

Opérations spécifiques à un ABR, ex : insérer, supprimer …

```
private
   type Lien is access Noeud;
   type Noeud is record
       Info : Element
       Gauche, Droit : Lien;
   end record;
end Arbres_Binaires_Recherche;
```

```
type ABR is record
   Racine : Lien;
end record;
```

2.3.2 Implémentation

- **Implémentation d'un arbre binaire de recherche avec des pointeurs**

Hypothèse : Cle ≠ Elemen

```
generic
   type Element is private;        -- (ex : Une_Personne)

   type Cle is private;            -- (ex : le n° INSEE)

   with function Cle_De(E : Element) return Cle;
   with function "<"(CleG, cleD : Cle) return boolean;

package Arbres_Binaires_Recherche is
  type ABR is private;

  -- spécifications des différentes opérations

private
  -- idem ABR précédent

end Arbres_Binaires_Recherche;
```

- **Implémentation d'un arbre binaire avec un tableau**

```
generic
    type Element is private;
package Arbres_Binaires is
   type Arbre_Bin is private;

   -- spécifications des différentes opérations
   -- idem précédent

private

end Arbres_Binaires;
```

Exercice : comment implémenter un arbre binaire avec un tableau ?

- **Opérations particulières**
  - Sur des arbres binaires
  - Sur des arbres binaires de recherche

  - **Parcourir**
  - **Rechercher**
  - **Insérer**
  - **Supprimer**

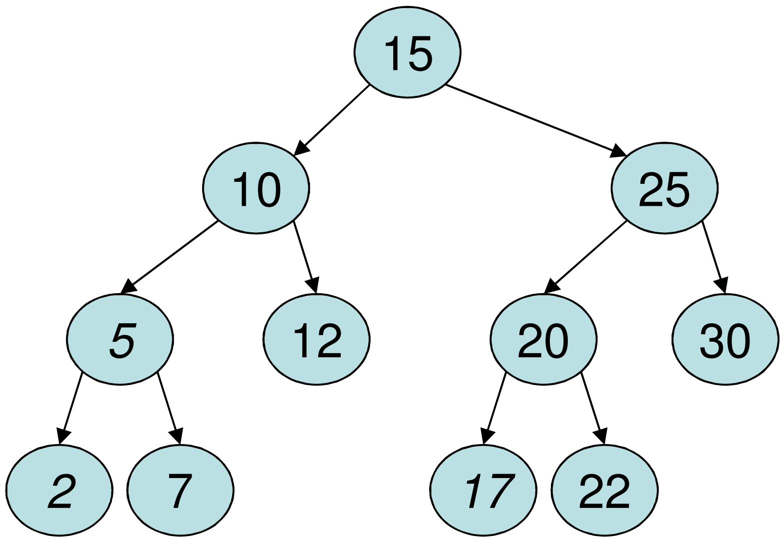
  - **Equilibrer**

Principes et Algorithmes

2.3.3. Parcours d'un arbre binaire

Visiter chaque nœud d'un arbre binaire pour appliquer un même traitement.

Exemple : afficher toutes les valeurs des nœuds d'un arbre binaire de recherche par ordre croissant




```
Afficher_Arbre(B)

si B n'est pas vide alors
      Afficher_Arbre(Sous_Arbre_Gauche(B))
      Afficher(Racine(B))
      Afficher_Arbre(Sous_Arbre_Droit(B))
finsi
```

```
Résultat de l'affichage :
2 5 7 10 12 15 17 20 22 25 30
```

Généralisation

Le parcours précédent est un cas particulier du **parcours générique** suivant :

```
Parcourir(B)
       --version récursive
si B n'est pas vide alors
     Traitement1(Racine(B))
     Parcourir(Sous_Arbre_Gauche(B))
     Traitement2(Racine(B))
     Parcourir(Sous_Arbre_Droit(B))
     Traitement3(Racine(B))
finsi
```

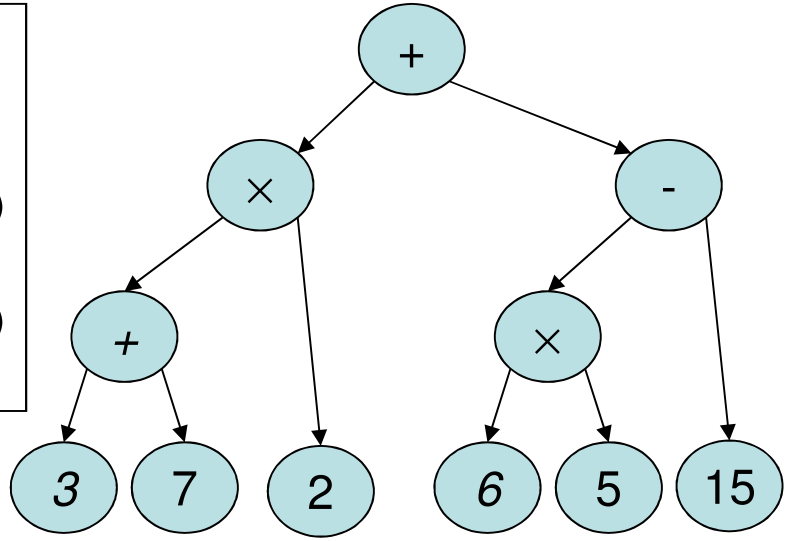
Dans le parcours pour **`Afficher_Arbre`** (cf transparent préced.) on a :

```
Traitement1(   ) = null
Traitement2(   ) = Afficher(Noeud)
Traitement3(   ) = null
```

`Afficher_Arbre` réalise **un parcours infixe** ou « **Gauche-Racine-Droite** »

Différents types de parcours : **Parcours infixe** ou « **Gauche-Racine-Droite** »

```
Parcourir(B)
      --version récursive
si B n'est pas vide alors
     Parcourir(Sous_Arbre_Gauche(B))
     Traitement2(Racine(B))
     Parcourir(Sous_Arbre_Droite(B))
finsi
```



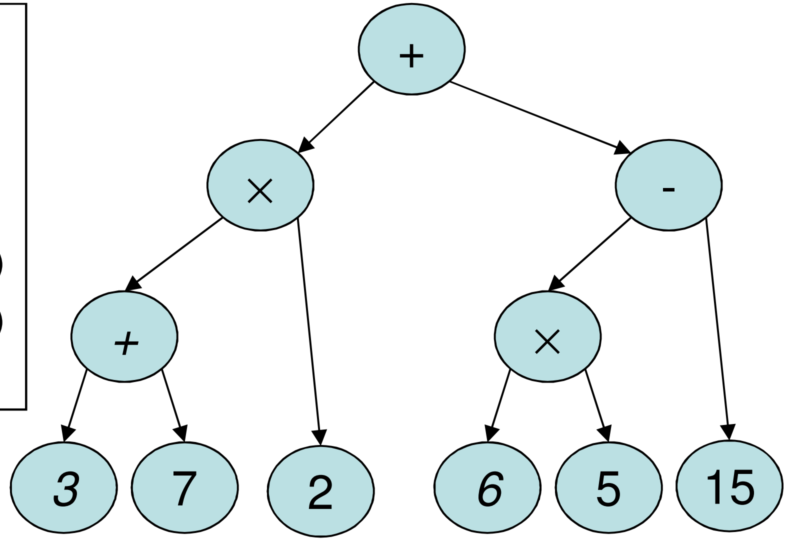

```
Résultat de l'affichage :
3 + 7 × 2 + 6 * 5 – 15
```

Exercice : **modifier les `Traitement` pour ajouter les parenthèses '(' et ')'** et obtenir : (((3+7)*2)+((6*5)-15))

2.3.3. Parcours

Différents types de parcours : **Parcours préfixe** ou « **Racine-Gauche-Droite** »

```
Parcourir(B)
      --version récursive
si B n'est pas vide alors
     Traitement1(Racine(B))
     Parcourir(Sous_Arbre_Gauche(B))
     Parcourir(Sous_Arbre_Droite(B))
finsi
```




```
Résultat de l'affichage :
+ ×+ 3 7 2 - * 6 5 15
```

Remarque :

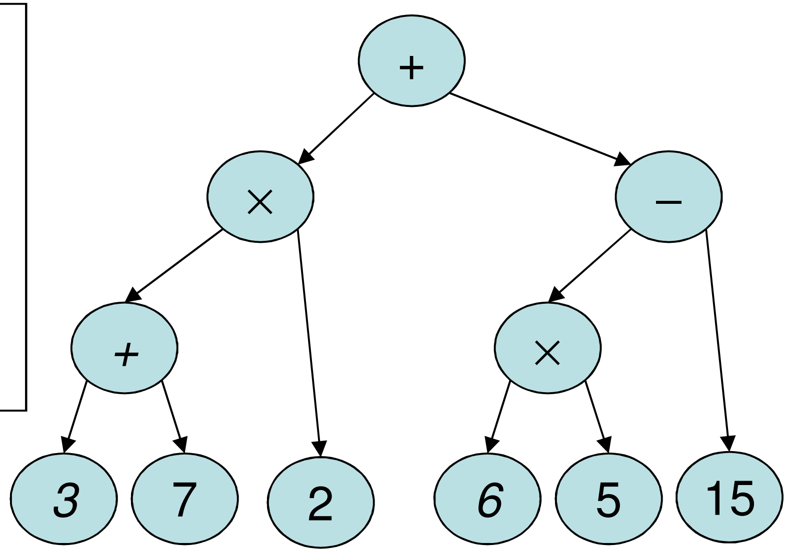
Les expressions obtenues sont proches de la *notation fonctionnelle* (le nom d'une fonction précède ses paramètres).

Exercice : **modifier les `Traitement` pour ajouter les parenthèses '(' et ')' et la virgule ',' aux expressions précédentes et obtenir l'expression** :

```
+(×(+(3,7),2), -(*(6,5),15))
```

Différents types de parcours : **Parcours postfixe** ou « **Gauche-Droite-Racine** »

```
Parcourir(B)
        --version récursive
si B n'est pas vide alors
     Parcourir(Sous_Arbre_Gauche(B))
     Parcourir(Sous_Arbre_Droite(B))
     Traitement3(Racine(B))
finsi
```



Résultat de l'affichage :
3 7 + 2 × 6 5 × 14 – +

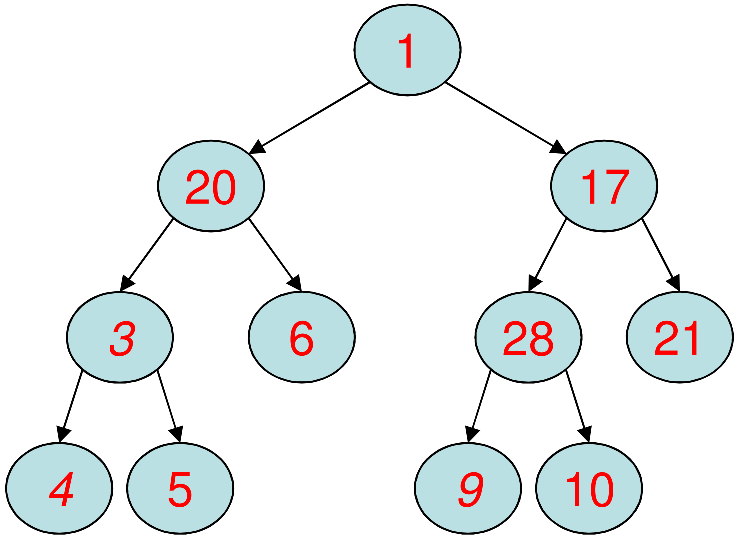
Remarque :

Les expressions obtenues sont proches dites en ***notation polonaise*** (l'opérateur succède aux opérandes).

Parcours en « profondeur d'abord » avec un algorithme itératif

(utilise une pile de **`Arbre_Bin`***)*

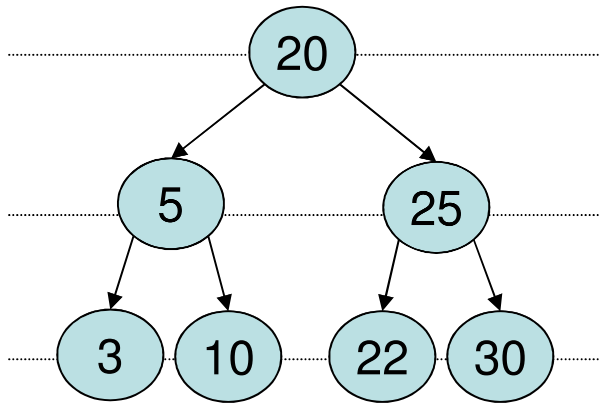
```
Parcours_Profondeur(B)
  -- version itérative

  Empiler(Pil, B)
  tantque not Est_Vide(Pil) faire
     s:= Sommet(Pil); Depiler (Pil);
     Traitement(s)
     G := Sous_Arbre_Gauche(s);
     D := Sous_Arbre_Droit(s);
     si not Est_Arbre_Vide(D) alors
        Empiler(Pil, D);
     finsi
     si not Est_Arbre_Vide(G) alors
        Empiler(Pil, G);
     finsi
   fintantque
```



Si Traitement = Afficher_Racine , on obtient :

1 20 3 4 5 6 17 28 9 10 21

Parcours par niveaux ou « en largeur d'abord » avec un algorithme itératif (utilise une file de **`Arbre_Bin`**)




```
Parcours_par_Niveaux(B)
      -- version itérative

  Ajouter(Fil, B)
  tantque not Est_Vide(Fil) faire
     s := Tete(Fil); Retirer(Fil);
     Traitement(s)
     G := Sous_Arbre_Gauche(s);
     D := Sous_Arbre_Droit(s);
     si not Est_Arbre_Vide(G) alors
        Ajouter(Fil, G);
     finsi
     si not Est_Arbre_Vide(D) alors
        Ajouter(Fil, D);
     finsi
   fintantque
```

Si Traitement = Afficher_Racine , on obtient :

20 5 25 3 10 22 30

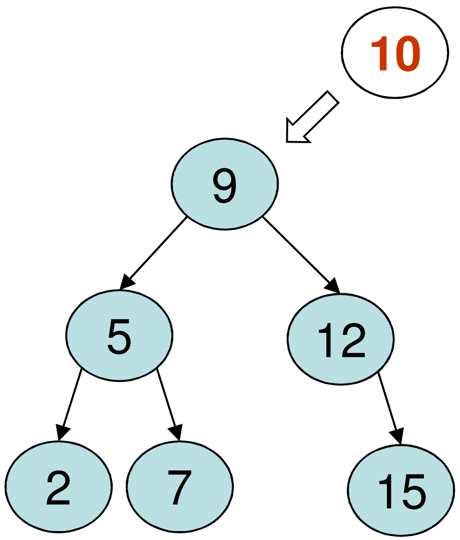
2.3.4. Recherche et Insertion dans un arbre binaire de recherche (ABR)

- **Soit A un ABR et soit un E un élément (ou une clé)**

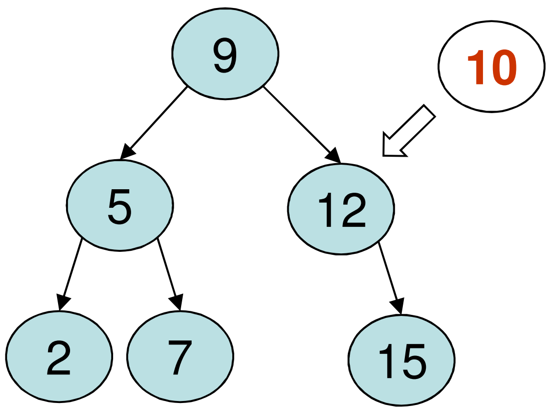
  Rechercher si E est présent dans A ➔ booléen

- **Principe**

  Comparer E au nœud courant et descendre éventuellement vers le sous-arbre G ou vers le sous-arbre D selon le résultat de la comparaison.



Comparaison 1 :
10 > 9 ?



Comparaison 2 :
10 > 12 ?

12 n'a pas de ss-arbre gauche

stop !

$\Rightarrow$ $10 \notin A$

 28

Programme Ada pour la recherche

ici : retourne un booléen ➔ Appartient

```
------------ version récursive -----------------------
function Rechercher(E: Element; A: ABR) return Boolean is

begin
  if Est_Vide(A) then
       return false;
  elsif E = Racine(A) then                    -- A.Debut.all.info
       return true;
  elsif E < Racine(A) then                    -- A.Debut.all.info
       return Rechercher(E, Sous_Arbre_Gauche(A));
  else -- E > Racine(A)                        -- A.Debut.all.info
       return Rechercher(E, Sous_Arbre_Droite(D));
  end if;
end Rechercher;
```

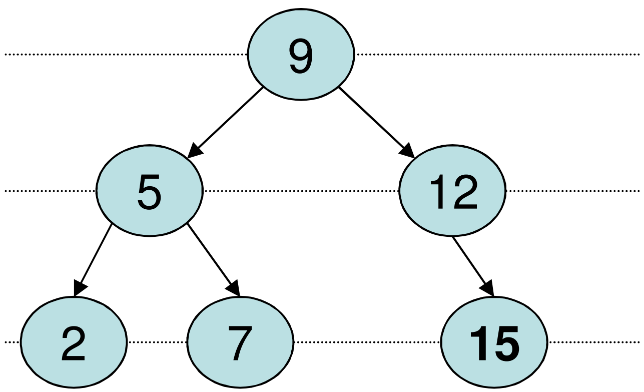
2.3.4. Recherche et Insertion

- **Complexité de Rechercher dans un ABR**
  - Proportionnel au nombre de nœuds de la branche parcourue

  ⇒ fonction de la hauteur de l'arbre

  - Soit *h* la hauteur de A et *n* la taille (nb d'éléments) de A
  - Complexité de **Rechercher** dans A = $\boldsymbol{O(h) = O(log(n))}$

Exemple avec un arbre de hauteur 3



Recherche du nœud 15 (succès)

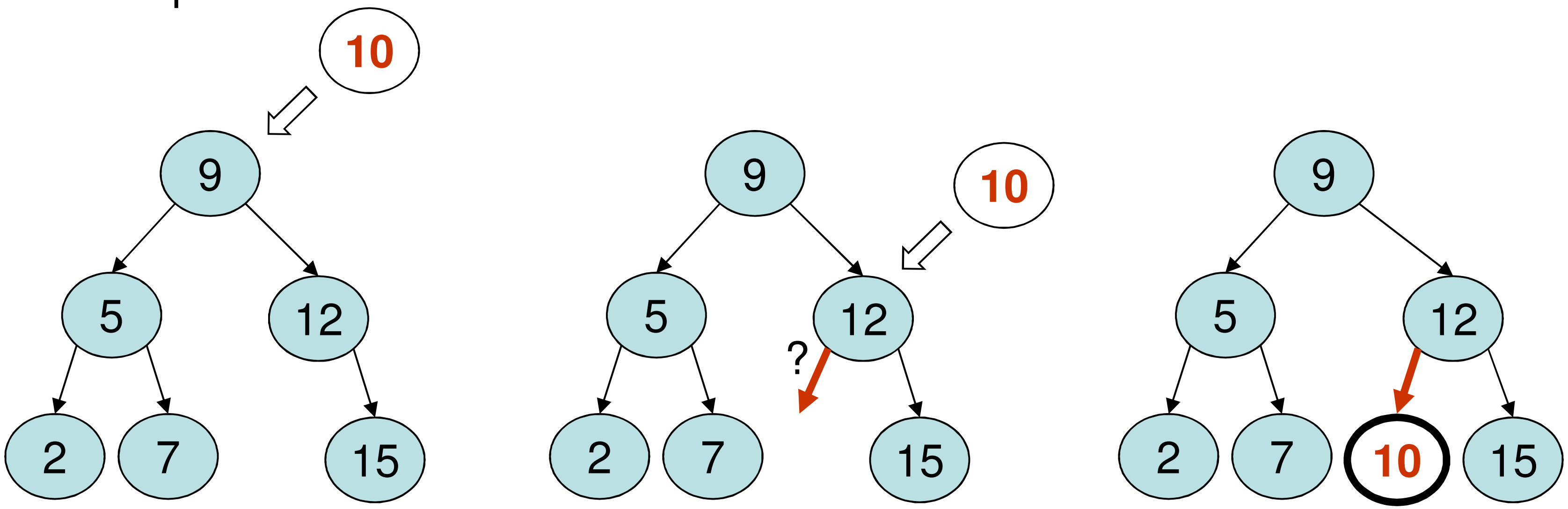
15 = 9 ? Non, chercher dans le sous-arbre droit (car 15 > 9)
15 = 12 ? Non, chercher dans le sous-arbre droit (car 15 > 12)
15 = 15 ? Oui, arrêt.
Rechercher 15 a nécessité **3 comparaisons**

Exemple: Recherche du nœud 10 (échec)

10 = 9 ? Non, chercher dans le sous-arbre droit (car 10 > 9)
10 =12 ? Non, chercher dans le sous-arbre gauche (car 10<12)
Stop (car 12 n'a pas de fils gauche)
Rechercher 8 a nécessité **2 comparaisons**

- **Principes pour l'insertion de E dans une feuille de l'ABR A:**
  - **insérer** E dans A s'il n'est pas déjà présent ; pour cela :
  - parcourir A jusqu'à atteindre un nœud R sans fils gauche tel que E<R ou un nœud R sans fils droit tel que E>R
  - puis créer une nouvelle feuille **de valeur** E à la place du fils manquant

Exemple



Comparaison 1 :
10 > 9 ?

Comparaison 2 :
10 > 12 ?

Insertion

 31

Programme Ada d'insertion en feuille

```
------------ Insertion en feuille (version récursive)-----

procedure Inserer(E : in Element; A : in out ABR) is

begin
  if Est_Vide(A) then                   -- creation d'une nouvelle feuille
       A := Construire_Arbre(E, Creer_Arbre_Vide, Creer_Arbre_Vide);
  elsif E = Racine(A) then                          -- E se trouve deja dans A
       null;
  elsif E < Racine(A) then                          -- insertion a gauche
       Inserer(E, Sous_Arbre_Gauche(A));
  else -- E > Contenu(Racine(A))                    -- test inutile
       Inserer(E, Sous_Arbre_Droit(A));             -- insertion a droite
  end if;

end Inserer;
```

Exercice à faire : écrire la version itérative (avec une boucle tantque)

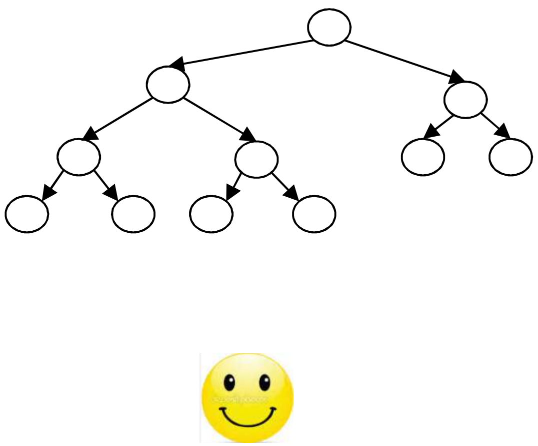
2.3.4. Recherche et Insertion

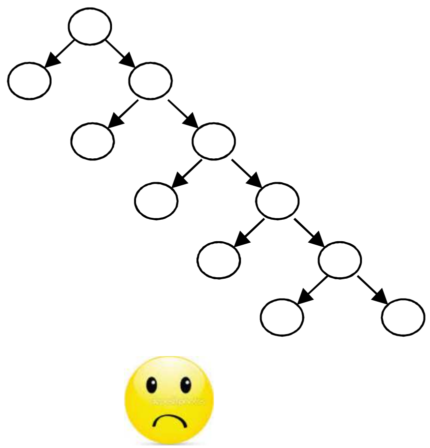
- **Complexité de Inserer dans un ABR**
  - Proportionnel au nombre de nœuds visités
  - Fonction de la hauteur de l'arbre

  - Soit *h* la hauteur de A et *n* la taille (nb d'éléments) de A

  Complexité de **Inserer** dans A = ***O*(*h*)**

  Si A arbre *« équilibré »* ***O*(*h*)** = ***O*(*log n*)**





2.3.5. Suppression dans un arbre binaire de recherche

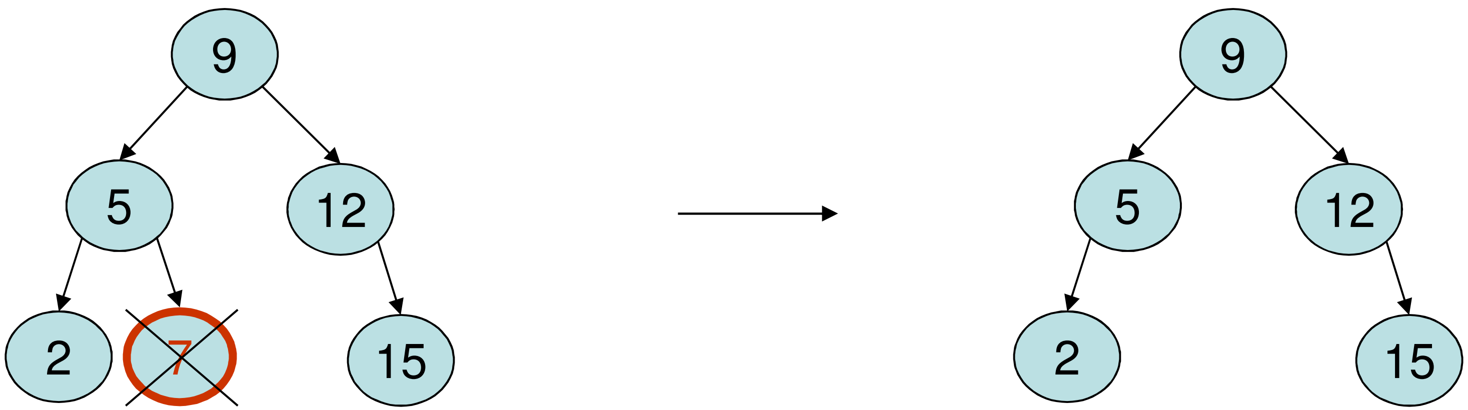
- **Soit A un ABR et soit un élément E**

  Supprimer le nœud *x* contenant l'élément E (on sait rechercher *x*)

  3 cas :

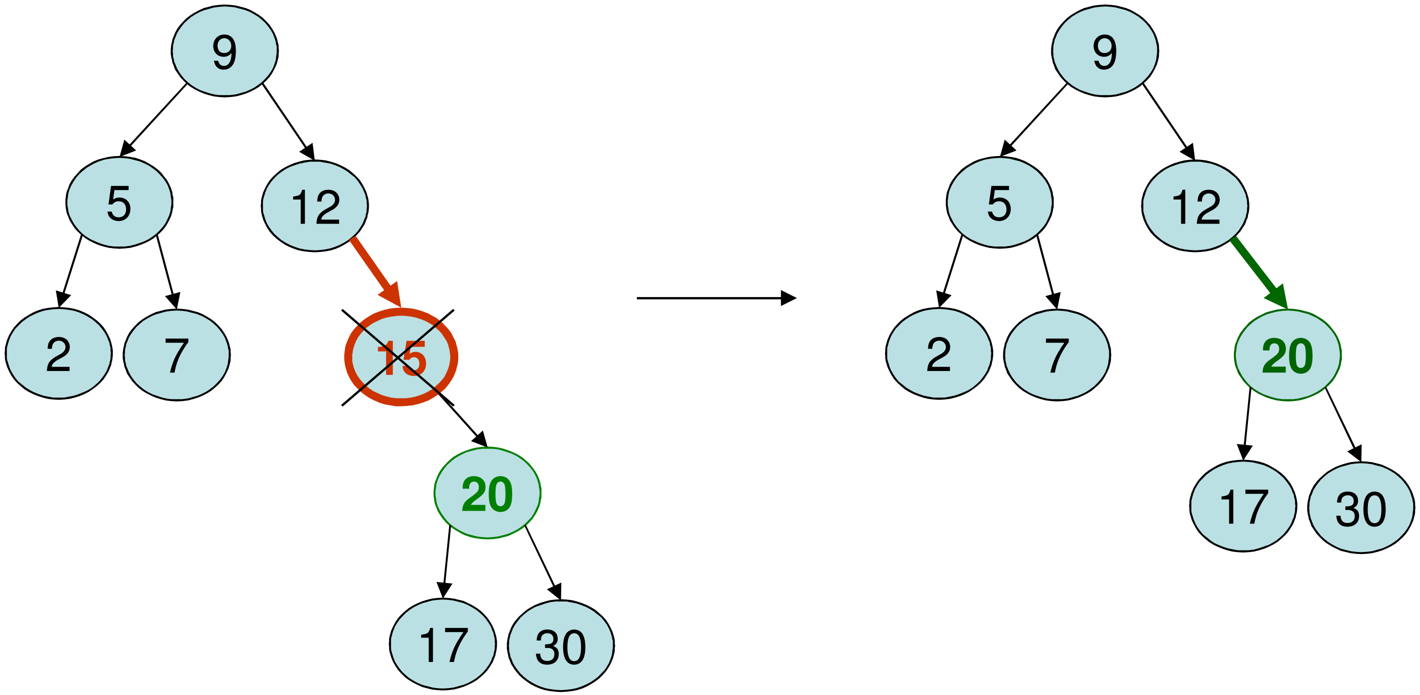
  - **Cas n° 1** (très facile) : *x* n'a aucun fils : on le supprime

Exemple : suppression du nœud contenant 7



- **Cas n° 2** (facile) : *x* a un fils unique (le gauche OU le droite) :
  son fils unique « remonte » et prend sa place

Exemple : suppression du nœud contenant15



2.3.5. Suppression

- **Cas n° 3** (difficile) : *x* a deux fils : il y a 2 solutions

    (1) chercher **le nœud de valeur immédiatement inférieure** à E dans le sous-arbre gauche de *x*

    ou

    (2) chercher **le nœud de valeur immédiatement supérieure à E** dans le sous-arbre droit de *x*

    Dans les 2 cas : supprimer le nœud trouvé et placer sa valeur dans le nœud *x*.

- **Solution (1)** (la solution (2) est symétrique)

    a) chercher l'étiquette la plus proche de E par valeur inférieure dans le sous-arbre gauche : MaxG

    (*Cheminer* ***le plus longtemps <u>à droite</u> dans le sous-arbre gauche de x*** *; le premier nœud* ***qui n'a pas de fils droit*** *est MaxG)*

    b) supprimer le nœud contenant MaxG

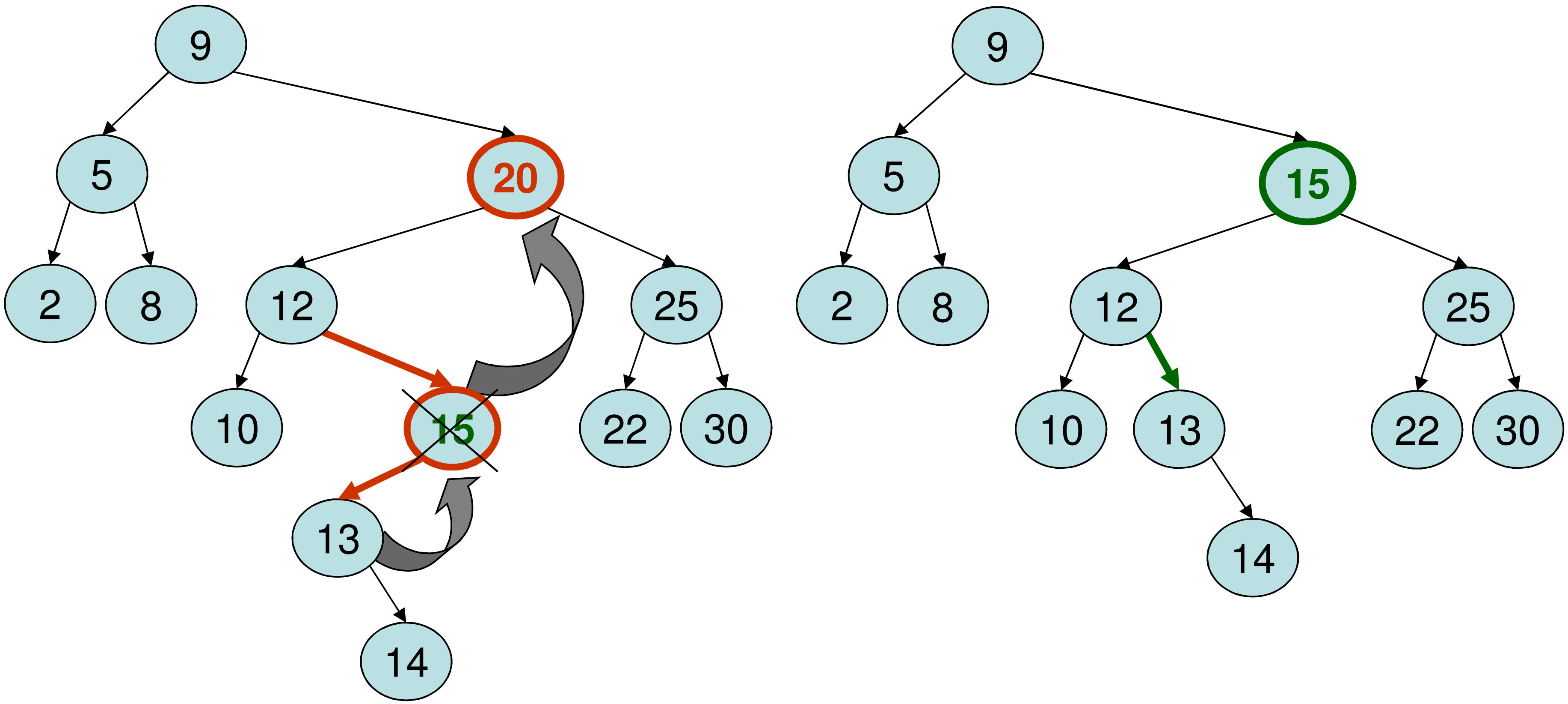
    c) remplacer la valeur E dans le nœud *x* par la valeur MaxG

36

2.3.5. Suppression

Exemple : suppression de l'étiquette **20**

- recherche du plus grand élément (MaxG) dans le sous-arbre gauche : 15
- suppression du nœud contenant **MaxG = 15**
- remplacement de la valeur 20 par la valeur 15



2.3.5. Suppression

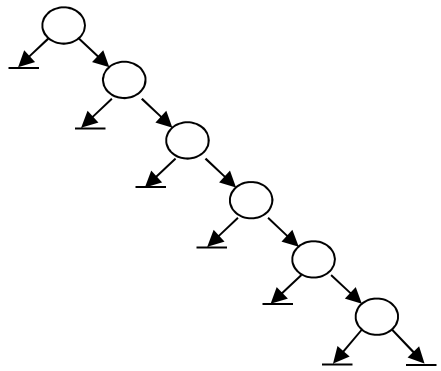
```
procedure  Supprimer(E:in Element; A:in out ABR) is
   ValMax : Element;

   procedure Enlever_Max(A : in out ABR; MaxG : out Element) is
   begin -- la procedure n'est jamais appelee avec A = null
     ............
   end Enlever_Max;

begin
  if Est_Vide(A) then null;               -- ou raise Element_Inexistant
  elsif E < Racine(A) then                -- Chercher E
        Supprimer(E, Sous_Arbre_Gauche(A));
  elsif E > Racine(A) then
        Supprimer(E, Sous_Arbre_Droit(A));
  else                                    --  E est la racine de A
        if Est_Vide(Sous_Arbre_Gauche(A)) then
            A := Sous_Arbre_Droit(A);
        elsif Est_Vide(Sous_Arbre_Droit(A)) then
            A := Sous_Arbre_Gauche(A);
        else  Enlever_Max(Sous_Arbre_Gauche(A), ValMax);
              Mise_A_Jour_Racine(ValMax, A);    ➔ OPERATION A DEFINIR
        end if;
  end if;
end Supprimer;
```

2.3.5. Suppression

```
with Unchecked_Deallocation;

procedure Free is new Unchecked_Deallocation(Noeud, ABR);


procedure Enlever_Max(A : in out ABR; MAxG : out Element) is
  Recup : ABR;   -- va pointer sur le noeud a desallouer

begin            -- la procedure n'est jamais appelee avec A = null
  if Est_Vide(Sous_Arbre_Droit(A)) then
     MaxG   := Racine(A);
     Recup := A;
     A := Sous_Arbre_Gauche(A); -- A pointe sur son fils gauche
     Free(Recup);                -- on recupere le noeud desalloue
  else
     Enlever_Max(Sous_Arbre_Droit(A), MaxG);
  end if;
end Enlever_Max;
```

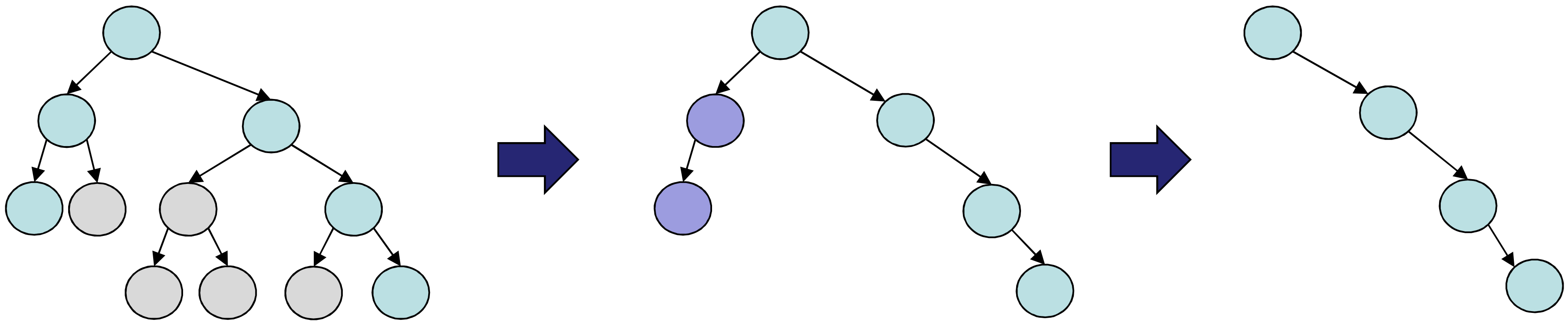
- **Complexité de Supprimer dans un ABR**
  - Proportionnel au nombre de nœuds visités
  - Fonction de la profondeur des nœuds ie. de la hauteur de l'arbre
  - $\Rightarrow$ même complexité que **`Rechercher`** et **`Inserer`**

- **Complexité Rechercher/Insérer/Supprimer dans un ABR = $O(h)$**
  - On a $\log_2 n \leq h \leq n\text{-}1$ où n est le nombre de nœuds de l'ABR
  - Eviter les cas défavorables où la $h \cong n$



2.3.6. Rotations et Equilibrage dans les arbres binaires de recherche

- **Exemple d'évolution d'un arbre binaire**



Arbre « quasi équilibré »

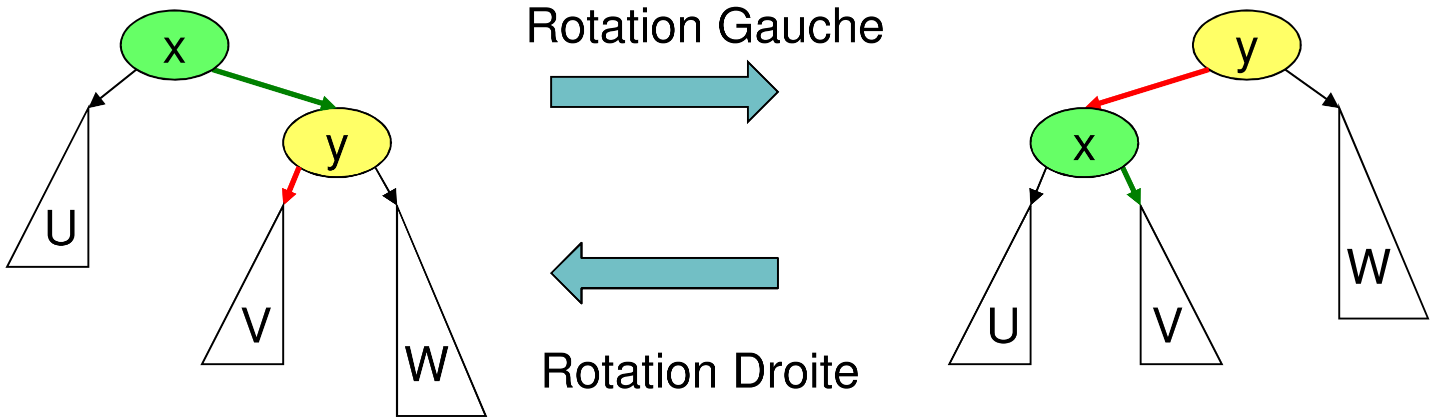
après 5 suppressions

après 7 suppressions l'arbre n'est plus du tout équilibré

Comment maintenir l'équilibrage ?

- **Rotations**
  - opérations générales sur les Arbres Binaires
  - permettent le rééquilibrage
  - soit A = (x, U, (y, V, W)) un ABR

Rotation Gauche G(A) = (y, (x, U, V), W)



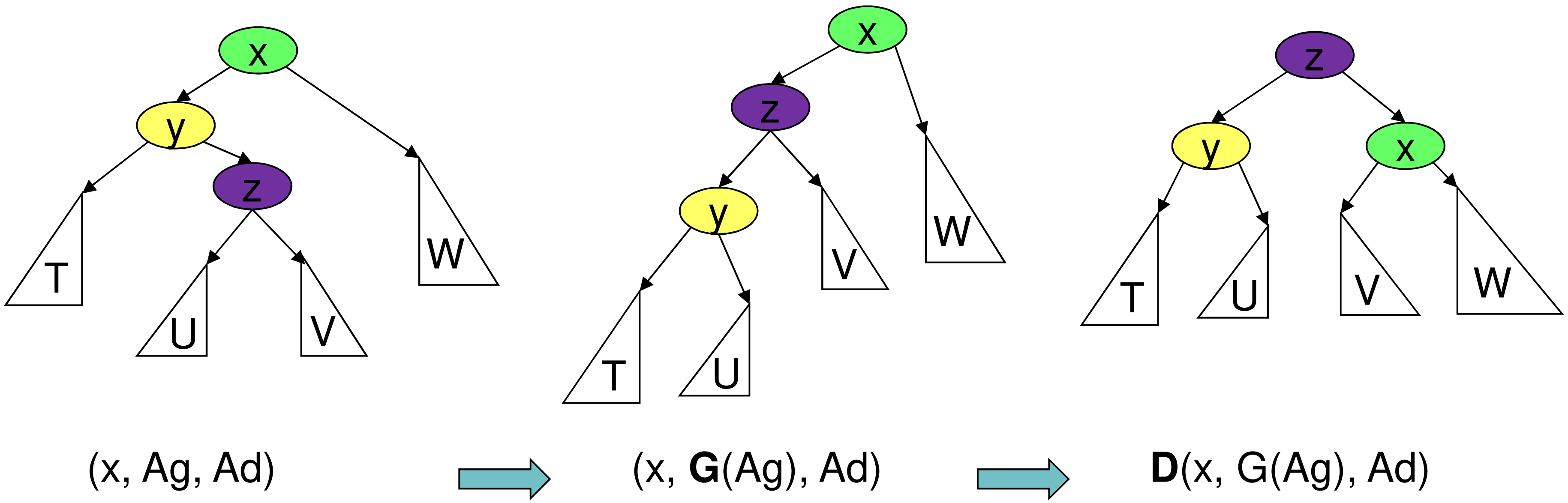
- soit A' = (y, (x, U, V), W) un ABR

Rotation Droite D(A') = (x, U, (y, V, W))

- **Double Rotations**
  - 2 sortes : Gauche – Droite et Droite – Gauche

  - soit A = (x, Ag, Ad) un ABR

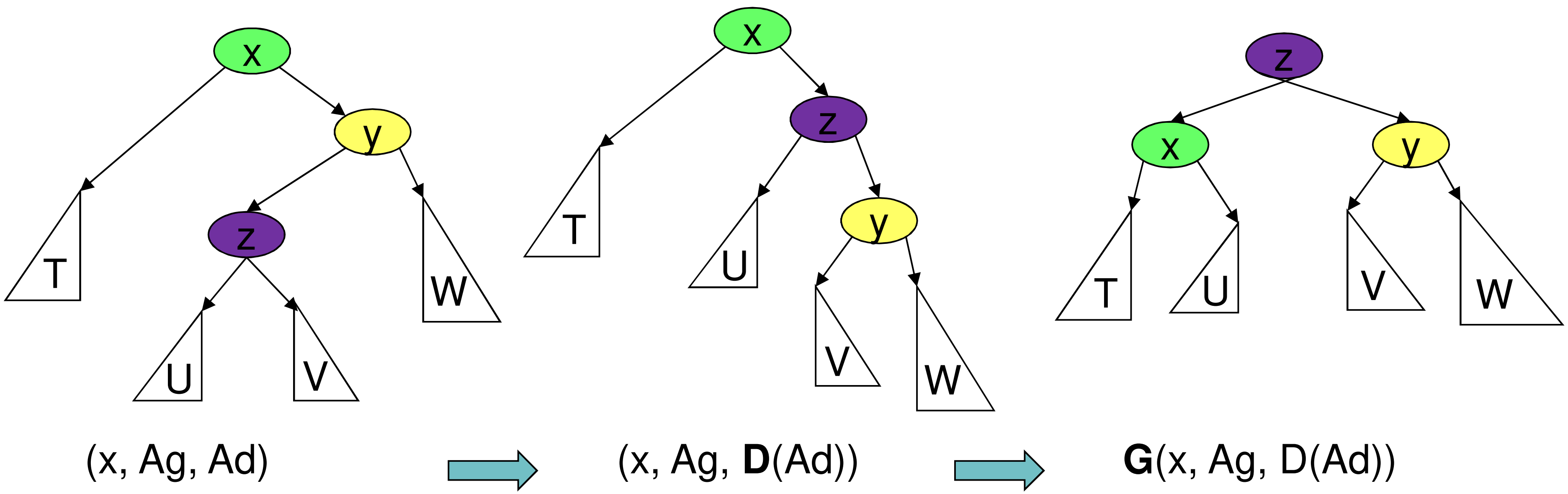
Rotation Gauche -Droite D(x, G(Ag) Ad)



2.3.6. Equilibrage et Rotation

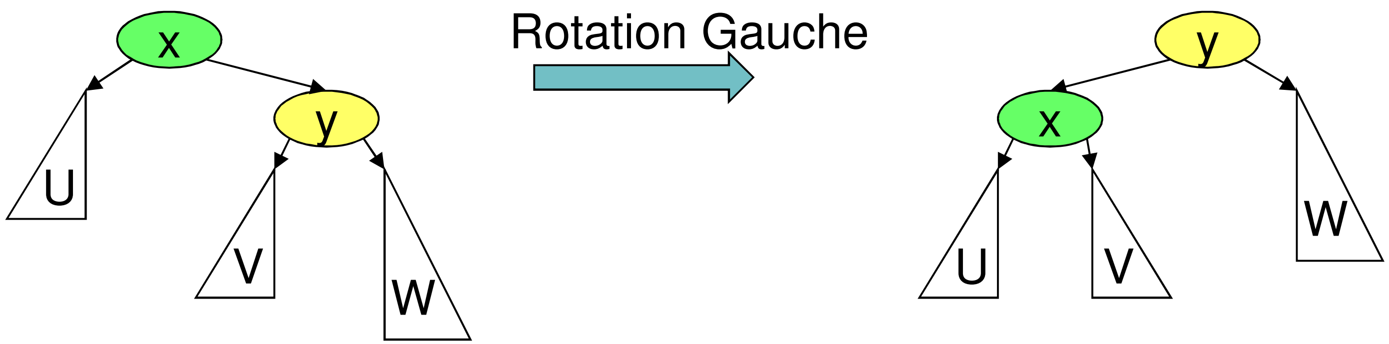
– soit A = (x, Ag, Ad) un ABR

Rotation Droite-Gauche G(x, Ag, D(Ad))



2.3.6. Equilibrage et Rotation

- **Algorithmes de Rotation**
  - Exemple pour rotation gauche




```
procedure Rotation_Gauche(A: in out Arbre_Bin) is
   Aux : Arbre_Bin; -- ou ABR
begin
   Aux              := A.Debut.Droit;
   A.Debut.Droit    := Aux.Debut.Gauche;
   Aux.Debut.Gauche := A;
   A                := Aux;
end Rotation_Gauche;
```

Complexité : temps constant (si implémentation avec pointeurs)

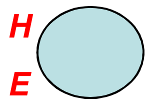
2.3.6. Equilibrage et Rotation

- **Equilibrage**
  - Rappel définition Hauteur d'un arbre binaire A, H(A)
    - Hauteur de chaque nœud = distance à la racine
    - Hauteur d'un arbre = Maximum des hauteurs de chaque feuille
  - Facteur d'équilibre E d'un arbre binaire A
    - Si A est vide alors **E(A) = 0**
    - Si A = (r, G, D) alors **E(A) = H(G) – H(D)**
- **Généralités sur les arbres équilibrés**
  - Objectif : sous-arbre gauche et sous-arbre droit de même hauteur
  - Vérifier ce principe à tous les nœuds de l'arbre
  - Maintenir un équilibre parfait à chaque insertion/suppression => coût élevé
  - Définir une condition plus faible d'équilibre donnant lieu à de bonnes performances pour le maintien de cette condition

    $\Rightarrow$ **arbres équilibrés dits « Arbres AVL »**

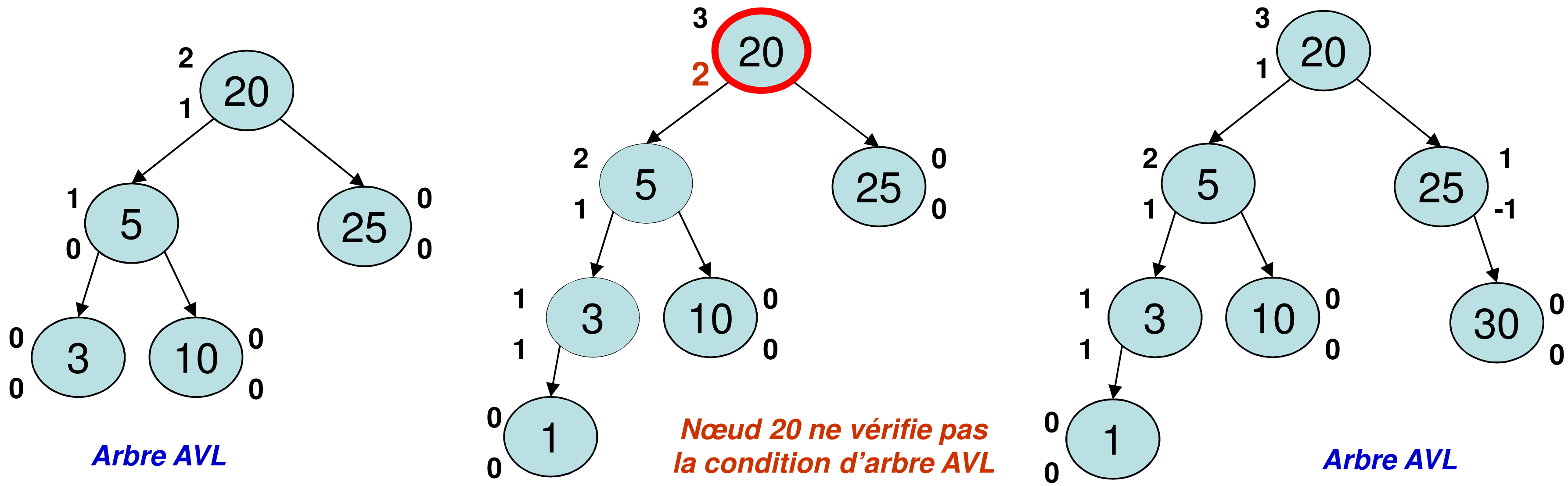
2.3.6. Equilibrage et Rotation

- **Arbres AVL (*Adelson Velskii Landis*)**
  - Un arbre binaire A est dit arbre AVL si tout sous-arbre *S* de *A* (*A* compris) vérifie : $|E(S)| \leq 1$
  - Exemple : notation



H = hauteur

E = facteur d'équilibrage



Arbre AVL

Nœud 20 ne vérifie pas la condition d'arbre AVL

Arbre AVL

m6 PB - Voir avec la définition de la hauteur

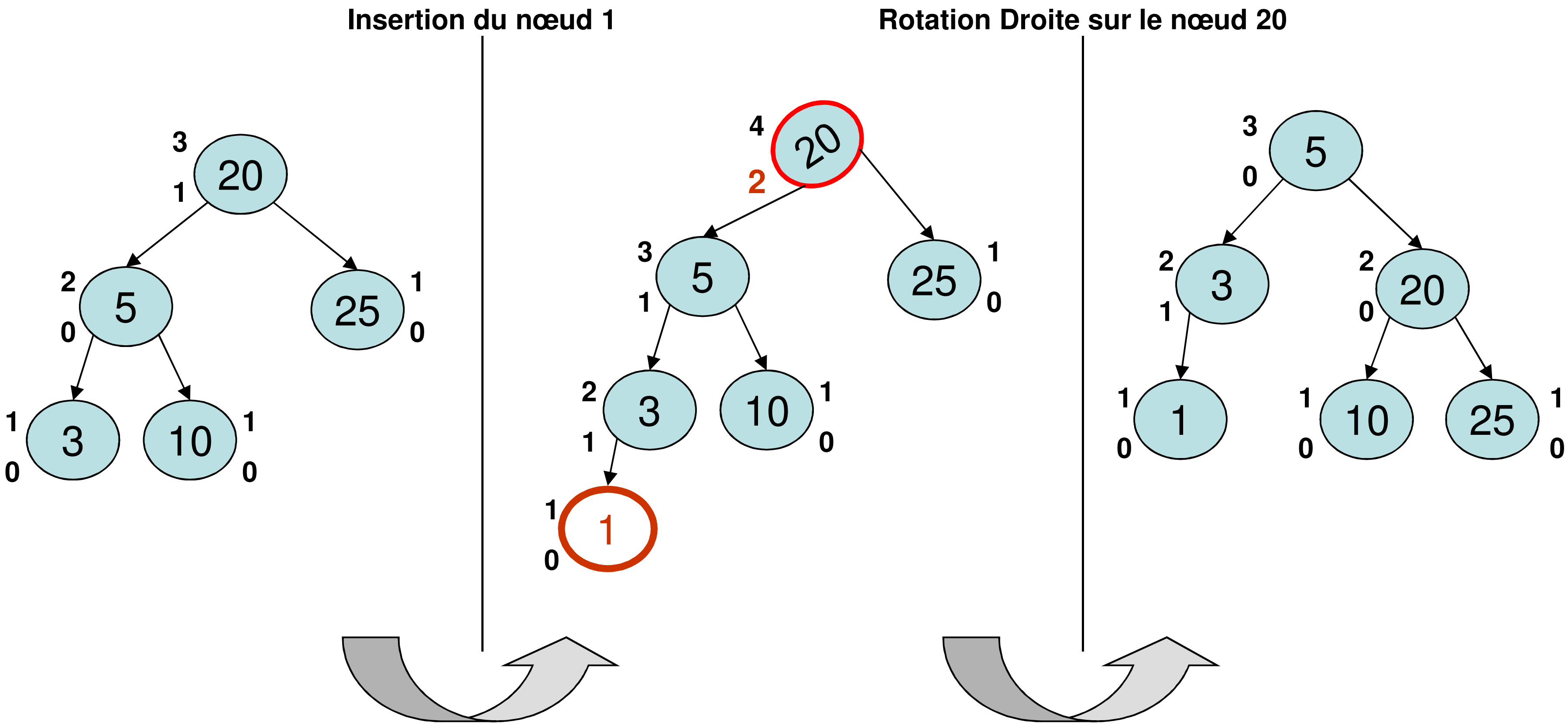
On dira que la hauteur d'un arbre vide vaut -1

mjhuguet; 12/01/2013

- **Insertion et Suppression dans un Arbre AVL**
  - Rétablir la condition d'équilibre après chaque insertion ou suppression

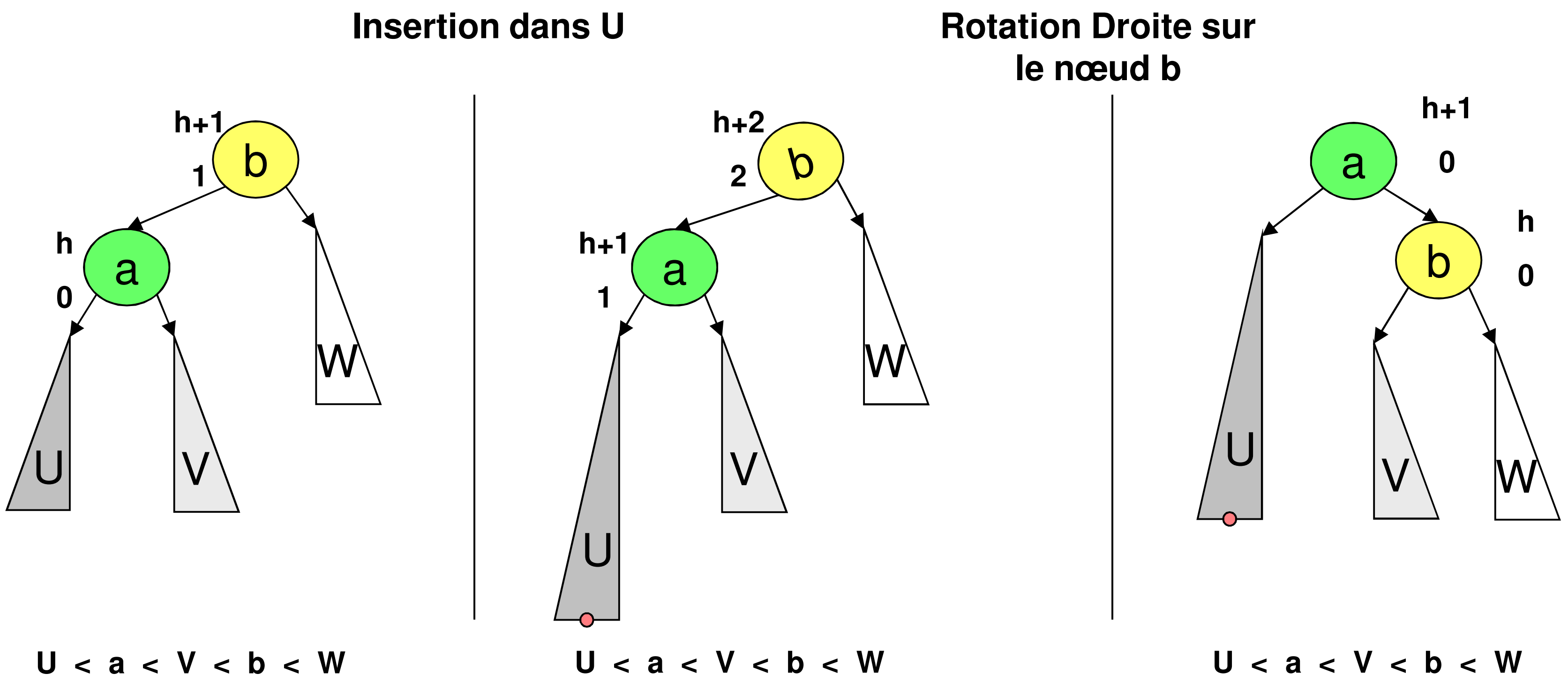
- **Insertion :**
  - Hypothèse : l'arbre vérifie la condition avant l'insertion
  - Insertion en feuille classique
  - Après insertion : déséquilibre possible pour les nœuds sur le chemin de la racine au nœud inséré
  - Augmentation maximum de la hauteur de 1
  - **Remonter du nœud inséré vers la racine en appliquant des rotations sur chaque sous-arbre déséquilibré**

- **Exemple d'Insertion dans un Arbre AVL**



- **Différentes situations**
  – Effectuer une rotation simple (Droite ou Gauche) pour rétablir l'équilibre
  – Effectuer une rotation double (Gauche-Droite ou Droite-Gauche)

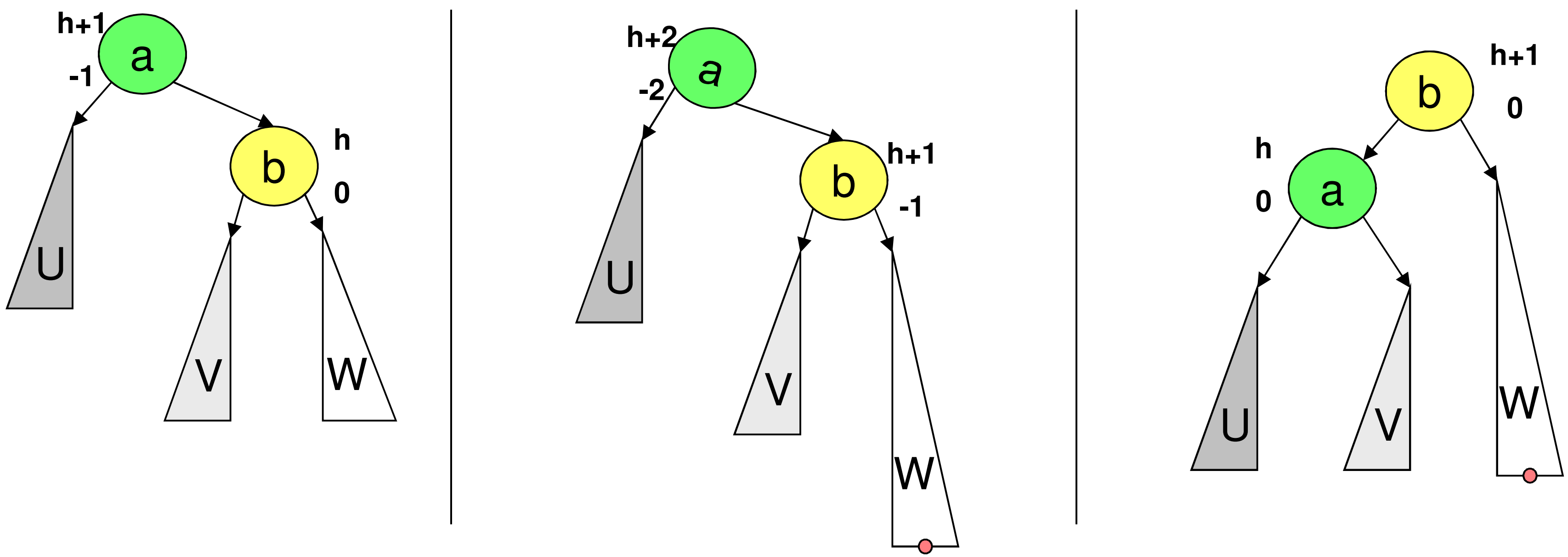
  – Cas nécessitant une **rotation DROITE**



2.3.6. Equilibrage et Rotation

– Cas nécessitant une **rotation GAUCHE**

Insertion dans W **Rotation Gauche sur le nœud b**



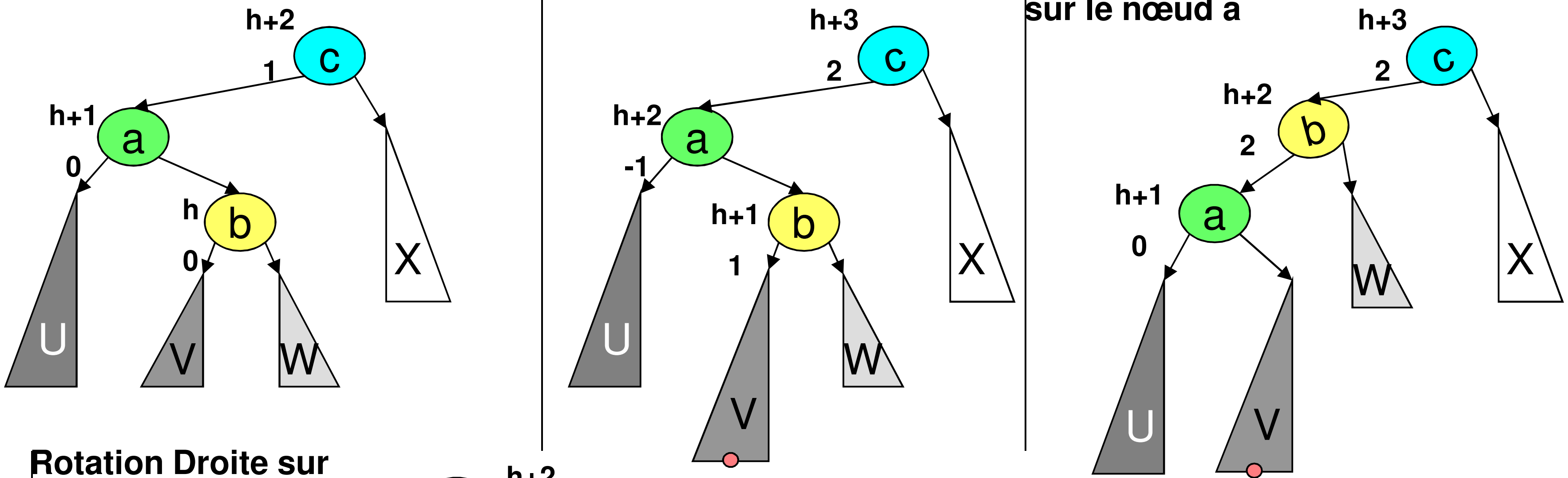
U < a < V < b < W **U < a < V < b < W** **U < a < V < b < W**

2.3.6. Equilibrage et Rotation

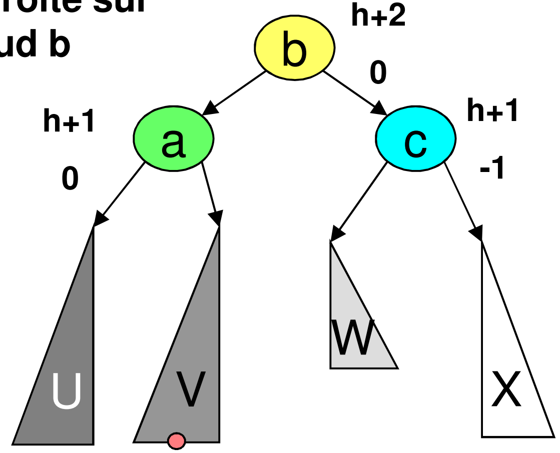
– Cas nécessitant une **double rotation GAUCHE-DROITE**

Insertion dans V

Rotation Gauche sur le nœud a



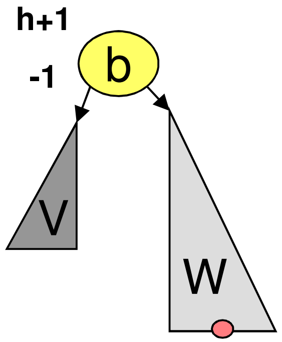
Rotation Droite sur le nœud b



À chaque étape, la relation d'ordre est maintenue :

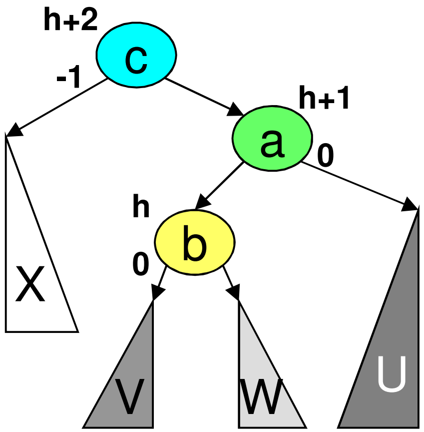
U < a < V < b < W < c < X

Même solution dans le cas insertion en W

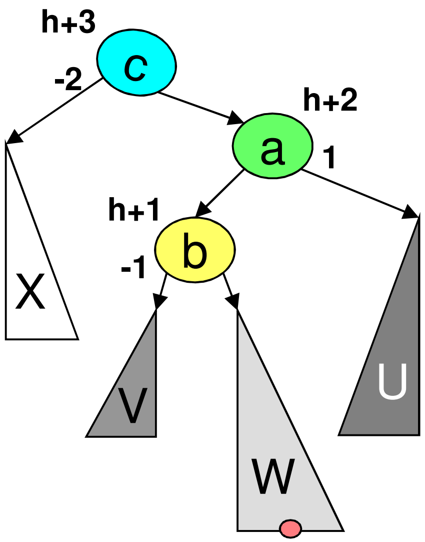


2.3.6. Equilibrage et Rotation

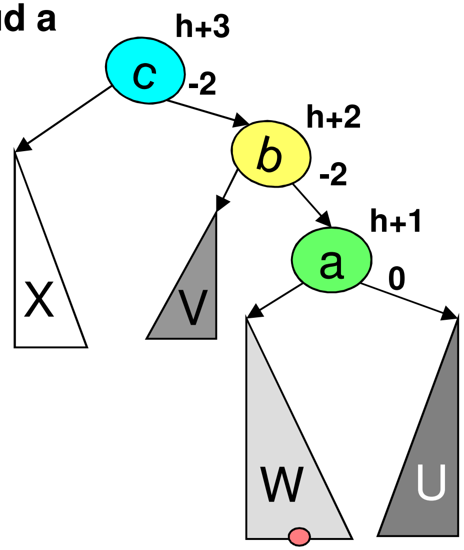
– Cas nécessitant une **double rotation DROITE-GAUCHE**



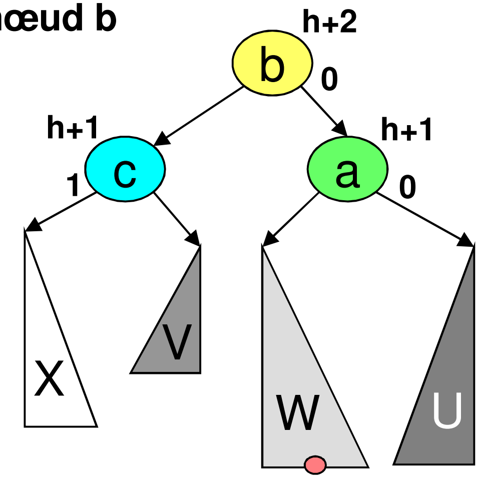
Insertion dans W



Rotation Droite sur le nœud a



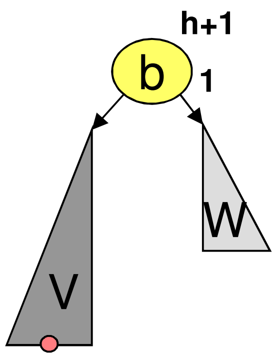
Rotation Gauche sur le nœud b



À chaque étape, la relation d'ordre est maintenue :

X < c < V < b < W < a < U

Même solution dans le cas insertion en V



- **Suppression dans un Arbre AVL**
  - Rotations successives du nœud à supprimer jusqu'à arriver à une feuille
  - Choisir les rotations pour maintenir la condition AVL
  - Supprimer ensuite la feuille

- **Propriété :**
  - Insertion et Suppression dans un AVL à n nœuds : O(log n)

- Bibliographie sur les AVL
    - Donald Knuth. "The Art of Computer Programming": Searching and Sorting Algorithms.
    - G.M. Adelson-Velskii and E.M. Landis."An algorithm for the organization of information", 1962
  - Liens internet
    - http://fr.wikipedia.org/wiki/Arbre_AVL
    - Nombreuses applet de simulation d'AVL

- **Autres familles d'arbres équilibrés**

  - **Arbres bicolores, dits Arbres « Rouge et Noir »** de Guibas et Sedgewick
    - couleur associée à chaque nœud (rouge ou noir)
    - (1) toutes les feuilles sont noires
    - (2) la racine est noire
    - (3) le père d'un nœud rouge est noir
    - (4) les chemins issus d'un même nœud et se terminant en une feuille ont le même nombre de nœuds noirs

  - **Arbres de Arne Anderson (AA) : variante des arbres bicolores**
    - condition supplémentaire : nœud fils gauche ne peut pas être rouge

  - Insertion et Suppression : rotations et changements de couleur

Exercices

- Arbres équilibrés (AVL)
  - Ecrire le programme Ada d'équilibrage après insertion dans ABR
    - Ajout d'un champ **Hauteur** ou d'un champ **Equilibre** en chaque noeud
    - Ajout d'une fonction d'accès à la hauteur

```
type Lien is access Noeud;
type Noeud is record
       Info : Element
       Gauche, Droit : Lien;
       Hauteur : natural;
 end record;
```

```
type ABR is record
   Debut : Lien
end record;
```

```
function H(A : in ABR) return natural is
begin
  if A = null then return 0;
  else return A.Hauteur;
  end if;
end H;
```

Exercices

- Insertion suivie d'un équilibrage (à développer)

```
procedure Inserer(E : in Element; A : in out ABR) is

begin
  if Est_Vide(A) then              -- creation d'une nouvelle feuille
        A := Construire_Arbre(E, Creer_Arbre, Creer_Arbre);
  elsif E = Racine(A) then         -- E se trouve deja dans A
        null;

  elsif E < Racine(A) then           -- insertion fils gauche
     Inserer(E, Sous_Arbre_Gauche(A));
     -- si desequilibre gauche, reéquilibrer par 1 ou 2 rotations

  else -- E > Contenu(Racine(A))   -- insertion fils droit
     Inserer(E, Sous_Arbre_Droit(A));
     -- si desequilibre droit, reéquilibrer par 1 ou 2 rotations

  end if;
end Inserer;
```

- Idem pour la suppression

2.4. Les Tas

1. Définition

Les tas permettent de réaliser le type abstrait **file de priorité**.
Une file de priorité est un ensemble dans lequel l'insertion et la suppression tiennent compte d'un **ordre de priorité**.

Un **Tas est un arbre binaire** vérifiant deux propriétés :

(a) Il existe une **relation d'ordre partiel** $P \leq N$ entre tout couple (P, N) de nœuds adjacents (P est le père de N)

En fait, $\exists$ deux types de tas possibles :

(I) les tas « *min* » : la racine est l'élément minimum ; chaque nœud non terminal est $\leq$ à ses fils (et donc à tous ses descendants).

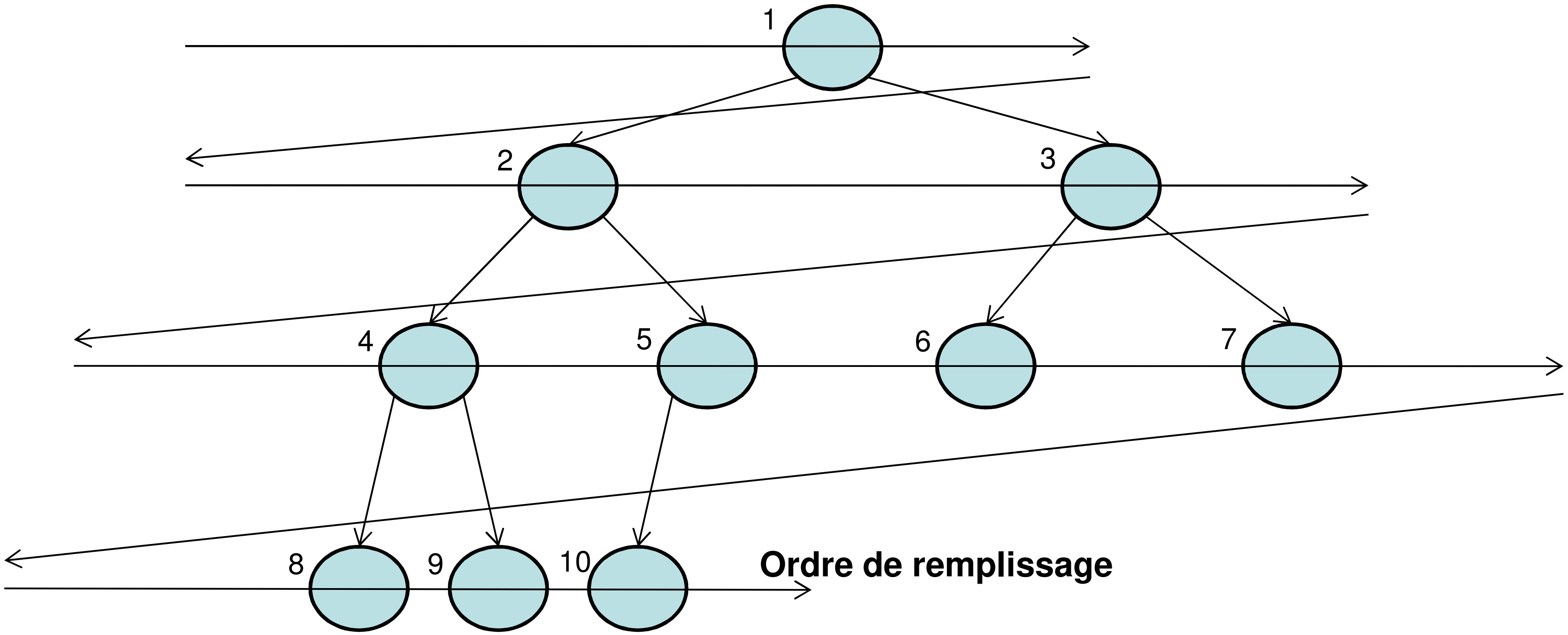
(II) **les tas « *max* »** : la racine est l'élément maximum ; chaque nœud non terminal est $\geq$ à ses fils (et donc à tous ses descendants).

Par la suite, ss perte de généralité, on ne considère que des tas « *min* ».

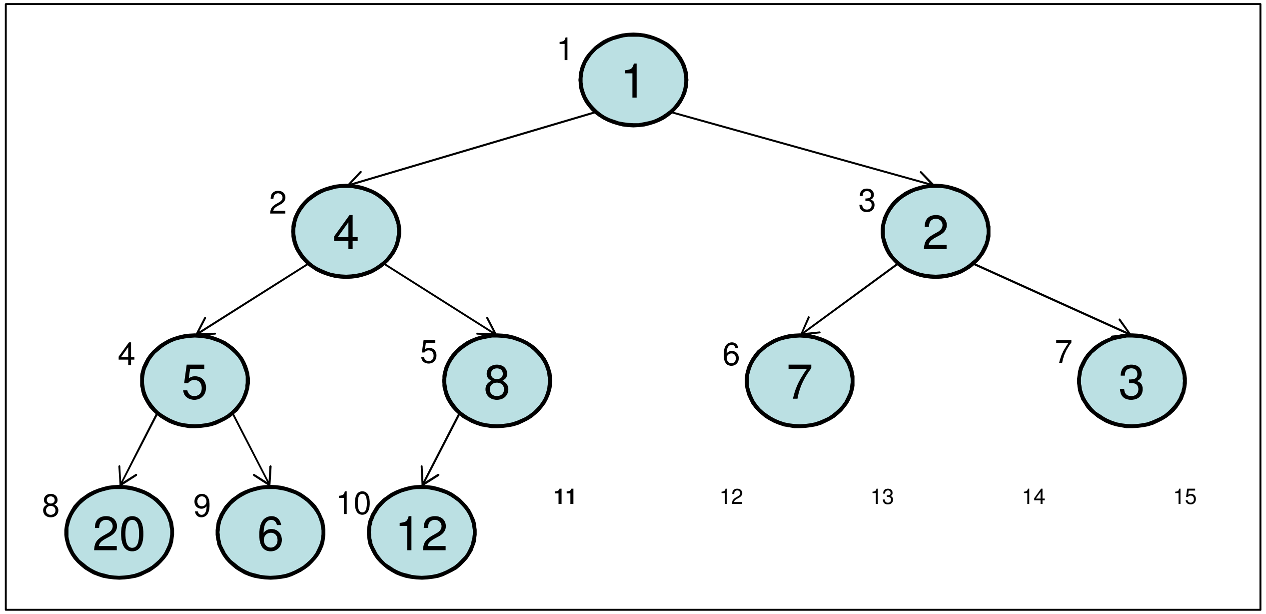
(b) Un tas T est un arbre binaire **quasi-complet** :

Si *h* est la hauteur de T, tous les niveaux 1, 2, … *h*-1 sont complets ; seul le niveau *h* est éventuellement incomplet.

De plus chaque nœud a une place ; lorsqu'on ajoute de nouveaux éléments, on doit remplir chaque niveau **de la gauche vers la droite**.



Exemple



(a) Vérification de la relation d'ordre :

1≤4, 1≤2, 4≤5, 4 ≤8, 2≤7, 2≤3, 5≤20, 5≤6, 8≤12

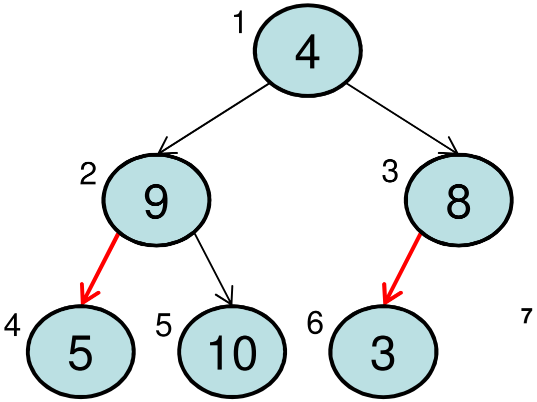
Remarque: les nœuds d'un même niveau ne sont pas forcément ordonnés.

(b) Vérification que l'arbre est presque complet :

- les niveaux 1,2,3 sont complets (on pourrait encore ajouter 5 places sur le niveau 4
- le tas a 10 éléments et la prochaine place libre est la n° 11

Contre-exemples

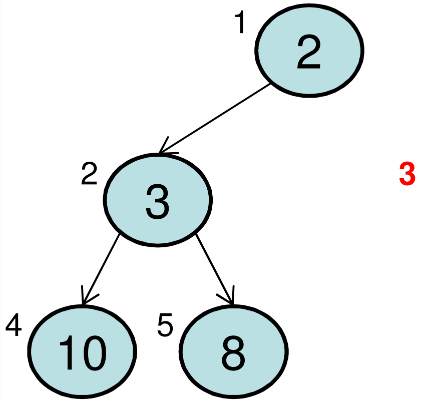
Les arbres binaires donnés ci-dessous ne sont pas des tas.



La relation d'ordre n'est pas respectée :

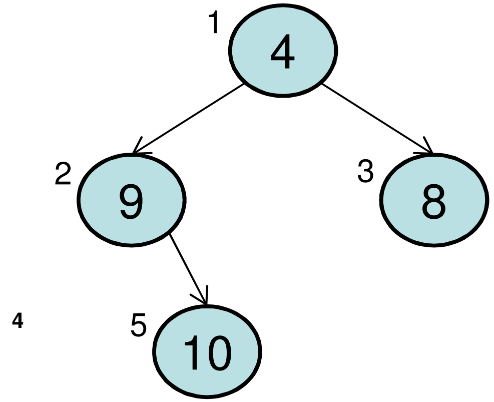
9>5,

8>3,



La relation d'ordre est respectée mais …

le niveau 2 n'est pas complet.



La relation d'ordre est respectée,

Le niveau 2 est complet mais …

le niveau 3 est mal rempli.

2. Relation entre hauteur et nombre d'éléments

Rappel

Hauteur d'un arbre binaire B : H(B) = plus grande profondeur d'une feuille

Si B vide alors H(B) = 0

Sinon (B=(r,G,D)) H(B) = 1 + max(H(G), H(D))

Un arbre binaire complet de hauteur 1 a exactement 1 nœud

Un arbre binaire complet de hauteur 2 a exactement 3 nœuds

…

Un arbre binaire complet de hauteur *h* a exactement $n = (2^h - 1)$ nœuds.

Hauteur d'un tas

Un tas est un arbre binaire presque complet (seul le dernier niveau n'est pas toujours complet), d'où :

Un tas de *n* éléments de hauteur *h* vérifie :

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

ou encore :

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

3. Opérations primitives sur un tas

Dans le cas général, les nœuds *x* du tas ont deux caractéristiques distinctes :

- une information : **info(x)**
- une clé : **clef(x)**

Les nœuds sont ordonnés par la valeur des clés :

si y ancêtre de x dans le tas, alors on a : cle(y) ≤ cle(x)

Par la suite, sans perte de généralité sur le plan des algorithmes, on considère des tas contenant uniquement des entiers (info = cle).

Opérations primitives

Créer_Tas : crée un tas vide
Insérer : insère un élément dans le tas
Supprimer_Minimum : retourne le minimum et l'enlève du tas

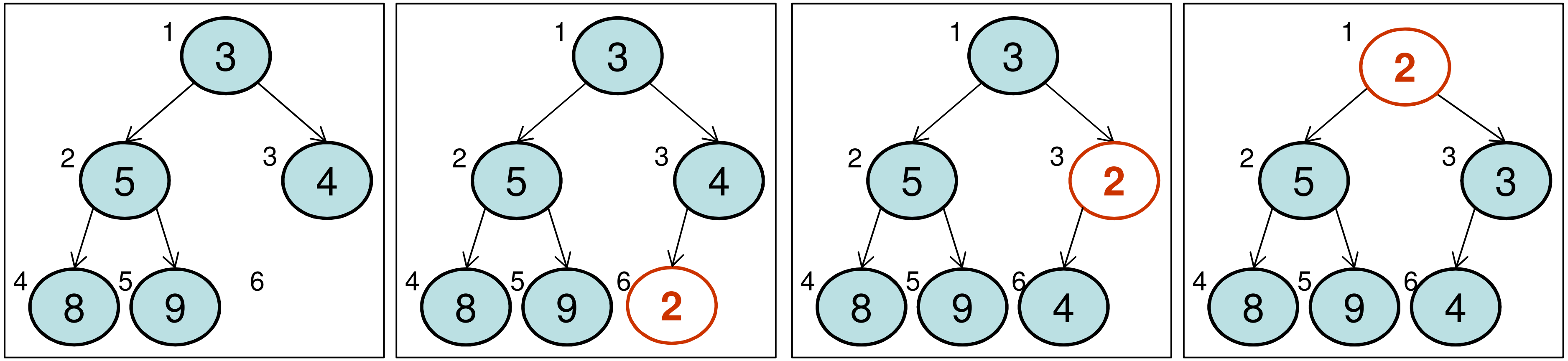
Extensions

Est_vide ? : indique si le tas est vide
Cardinal ? : indique le nombre d'éléments dans le tas
Minimum ? : retourne le plus petit élément (sans l'enlever)
Supprimer : enlève un élément du tas
Mettre_A_Jour : modifie la priorité d'un élément du tas

Insérer un nouvel élément dans le tas

L'insertion s'effectue en 2 phases :

a. Occupation de la prochaine place libre du tas
b. Remontée dans la branche qui va vers la racine



Le tas initial.

On veut insérer
Un nouvel
élément : **2**.

On insère **2** sur la 1ère place libre du tas.

Mais 2 est mal placé, car 4>2.

On fait remonter 2 en l'échangeant avec 4.

Mais 2 est encore mal placé, car 3>2.

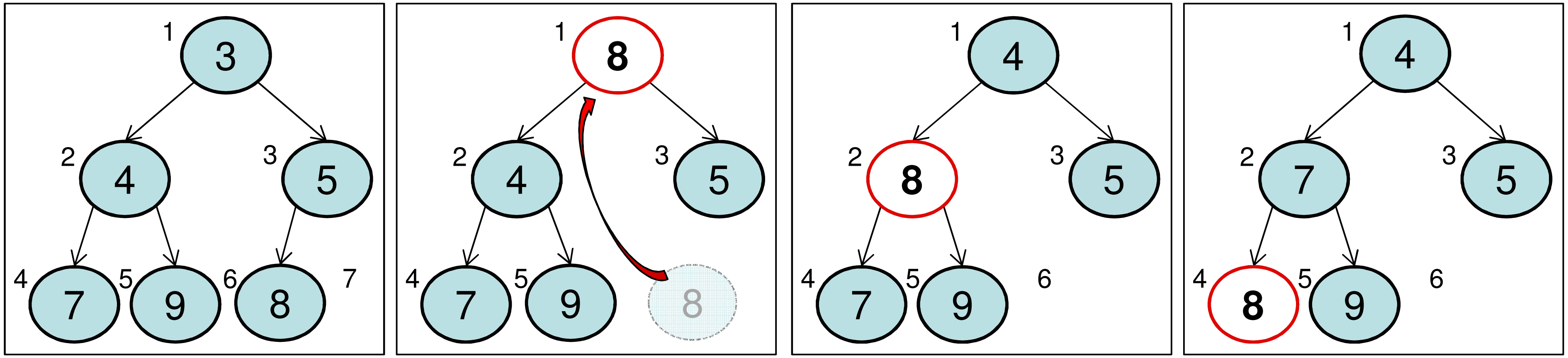
On fait remonter 2 en l'échangeant avec 3.

Le tas final est correctement arrangé.

Supprimer le minimum

Le minimum est à la racine. La suppression du minimum s'effectue en 2 phases :

a. remplacement par l'élément situé à la dernière place
b. descente du nouvel élément jusqu'à ce qu'il soit bien placé



Le tas initial.

On veut supprimer la racine : 3.

On déplace le dernier élément (c'est 8)

à la racine.

Mais 8 est mal placé, car 8>4 et 8>5.

8 descend : il est échangé avec le + petit de ses fils : 4.

Mais 8 est encore mal placé, car 8>7.

8 descend : il est échangé avec le + petit de ses fils : 7.

Le tas final est correctement arrangé.

Complexité théorique des opérations sur un tas

Dans les 2 cas (insertion d'un élément, suppression de la racine), la complexité des opérations dépend essentiellement :

- de la facilité à accéder à la fin du tas (prochaine place libre ou dernière place)

- du nombre d'opérations réalisées pour faire monter ou descendre un élément.

Or, on sait que dans le pire des cas, il faudra au plus h comparaisons (h = hauteur du tas) pour qu'un élément mal placé trouve sa place définitive ; or $h \cong log_2(n)$.

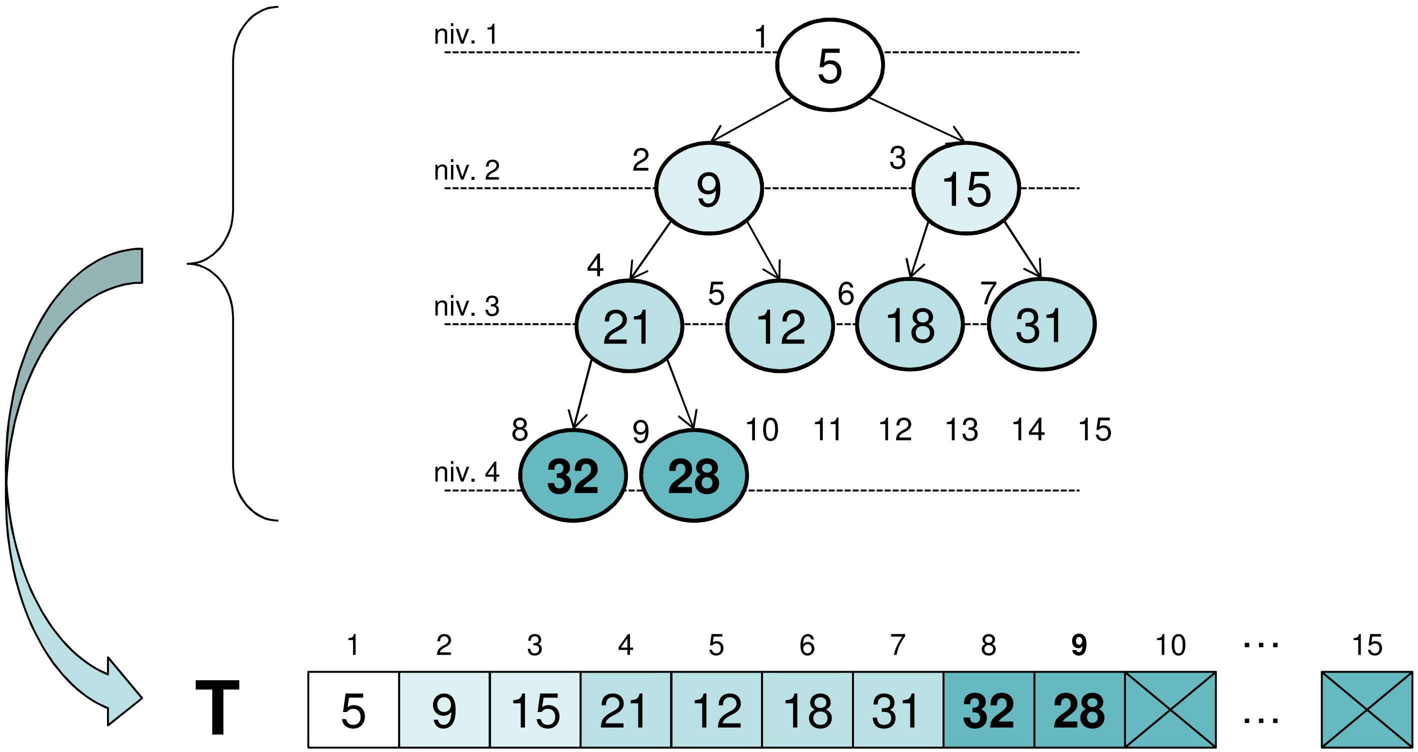
Une implémentation correcte de la structure de tas doit donc permettre de réaliser des opérations de complexité ***O*(*log* n).**

L'implémentation la plus connue et la plus efficace consiste à utiliser un **tableau**.

4. Opérations primitives sur un tas codé par un tableau

Principe

Un tas de hauteur *h* peut être codé par un tableau T de $2^h – 1$ cases.
ex : h=4 , on utilise un tableau T de 15 cases.



Propriétés

soit : ***taille*(T)** le nombre de cases du tableau T
p la place d'un élément *x* dans le tableau : ***x* = T(*p*)** et
n le cardinal du tas (nombre courant d'éléments)

- dernier élément de T = T(n) si n=*taille*(T) ⇔ le tableau est plein.
  sinon prochaine place libre = T(n+1)

- pere(x) n'est défini que si p>1 (tout nœud a 1 père, sauf la racine)
  pere(x) = T(*p*/2) '/' = division entière

- 2.p = n ⇔ x a un fils unique, le fils gauche
  fils_gauche(x) = T(2.*p*)

- 2.p +1 = n ⇔ x a exactement deux fils :
  fils_gauche(x) = T(2.*p*)
  fils_droite(x) = T(2.*p* + 1)

Passer d'un nœud à son père ou d'un nœud à ses fils est donc très simple.

Code avec une implémentation basée sur un tableau

```
inserer(E, T)
    -- si cardinal(T) = taille_max(T) alors        -- si T est plein, il faut allouer 1 niveau de plus
    --   augmenter_taille(T)                       -- ex : on lui alloue 2^h cases supplémentaires
    -- fsi

    pos ← cardinal(T) +1                           -- on commence par placer E dans le
    T(pos) ← E                                     -- prochain emplacement libre de T
     incrementer_cardinal(T)                       -- le nombre d'éléments augmente

    si pos>1 alors                                 -- si E n'est pas à la racine, alors on essaie
       move_up(pos,T)                              -- de le faire remonter jusqu'à ce qu'il soit
     finsi                                         -- bien placé
```

```
enlever_racine(T, R)
    si cardinal(T) = 0 alors                       -- si le tas est vide on ne peut
            lever_exception(Tas_Vide)              -- pas enlever sa racine
    sinon
            R ← T(1)                               -- R est le plus petit élément
            T(1) ← T(cardinal(T))                  -- on remplace la racine par le dernier
            decrementer_cardinal(T)                -- on diminue le nombre d'éléments

             si cardinal(T) > 1 alors              -- l'élément a la racine est peut-être mal placé
                    move_down(1,T)                 -- on doit essayer de le faire descendre
             finsi
    finsi
```

2.4. Les tas

```
move_up(i,T)                            -- pour essayer de faire remonter T(i)
    p ← i/2                             -- on détermine la position p du père de T(i)

    tantque i>1 et T(p)>T(i)            -- tantque T(i) est mal placé (< à son père)
        Auxi    ← T(i)
        T(i)    ← T(p)                  -- on le permute avec son père
        T(p)    ← Auxi

        i       ← p                     -- on se prépare à recommencer avec le père
        p       ← i div 2
    ftq
```

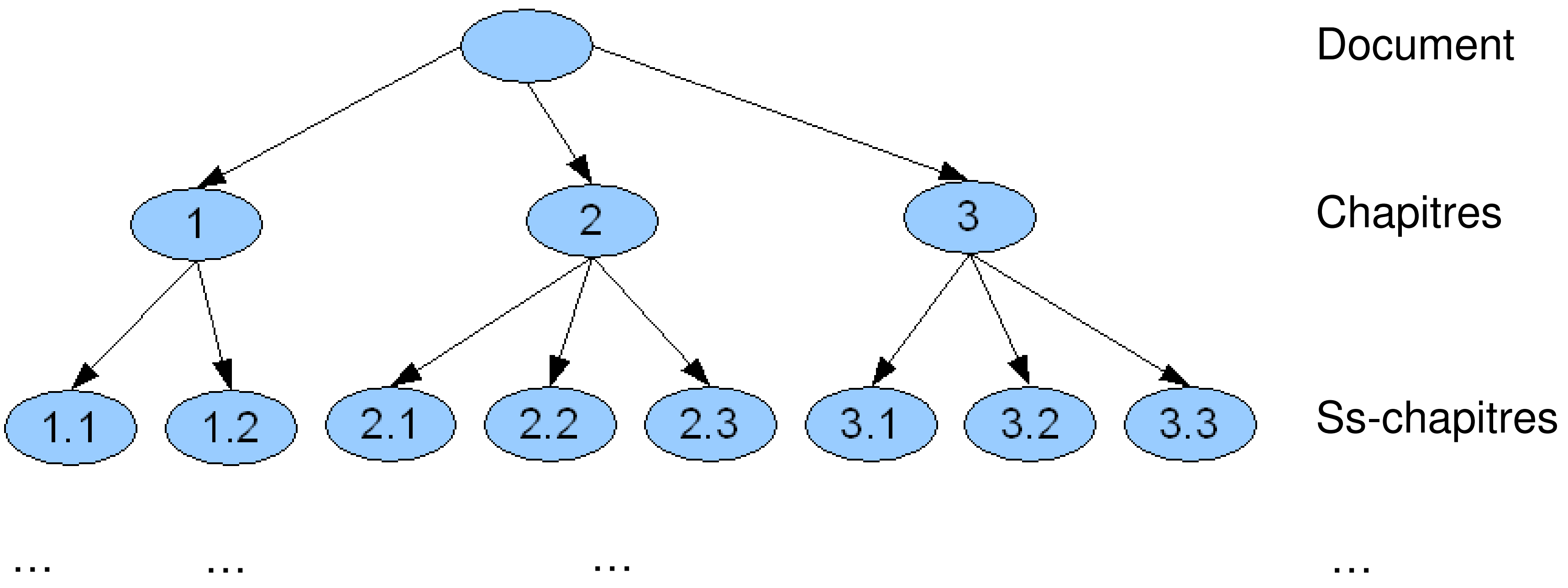
```
move_down(p,T)                      -- pour essayer de faire descendre l'élément placé en p
      n ← cardinal(T)
      fini ← faux
      tantque 2.p ≤ n et fini = faux    -- on arrête si on est au dernier niveau ou si on a fini
                                        -- (sinon le nœud en p a au moins un fils gauche en 2.p)

            Fils_Min ← 2.p              -- on cherche le plus petit fils : on suppose que c'est le gauche
            si 2.p +1≤n  et si T(2.p+1) < T(2.p) alors   -- s'il y a aussi un fils droit et s'il est
                    Fils_Min ← 2.p+1                      -- plus petit, on garde le droit
            finsi

            si T(Fils_Min) < T(p) alors      -- si le + petit des fils est + petit que son père
                  Auxi ← T(p)
                  T(p) ← T(Fils_Min)         -- on permute le pere et le + petit fils
                  T(Fils_Min) ←Auxi

                  p ← Fils_Min               -- on se prépare à faire descendre le nouveau fils
            sinon
                  fini ← vrai                -- le nœud actuel est bien placé : on a fini
            finsi
      ftq
```

• 2.5. Arbres *n*-aires

Un nœud a au plus 1 père mais peut avoir entre 0 et *n* fils

2.5.1 Exemples

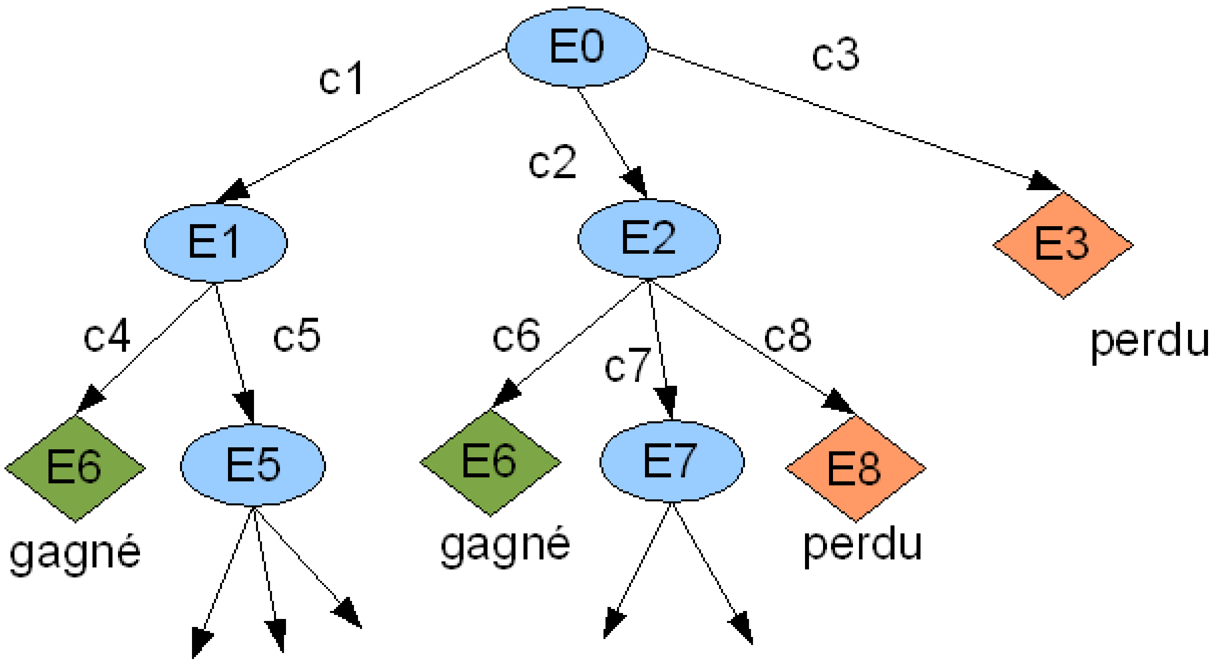
Exemple 1 : **arborescence d'un document**



Exemple 2 : **arbre de jeu**

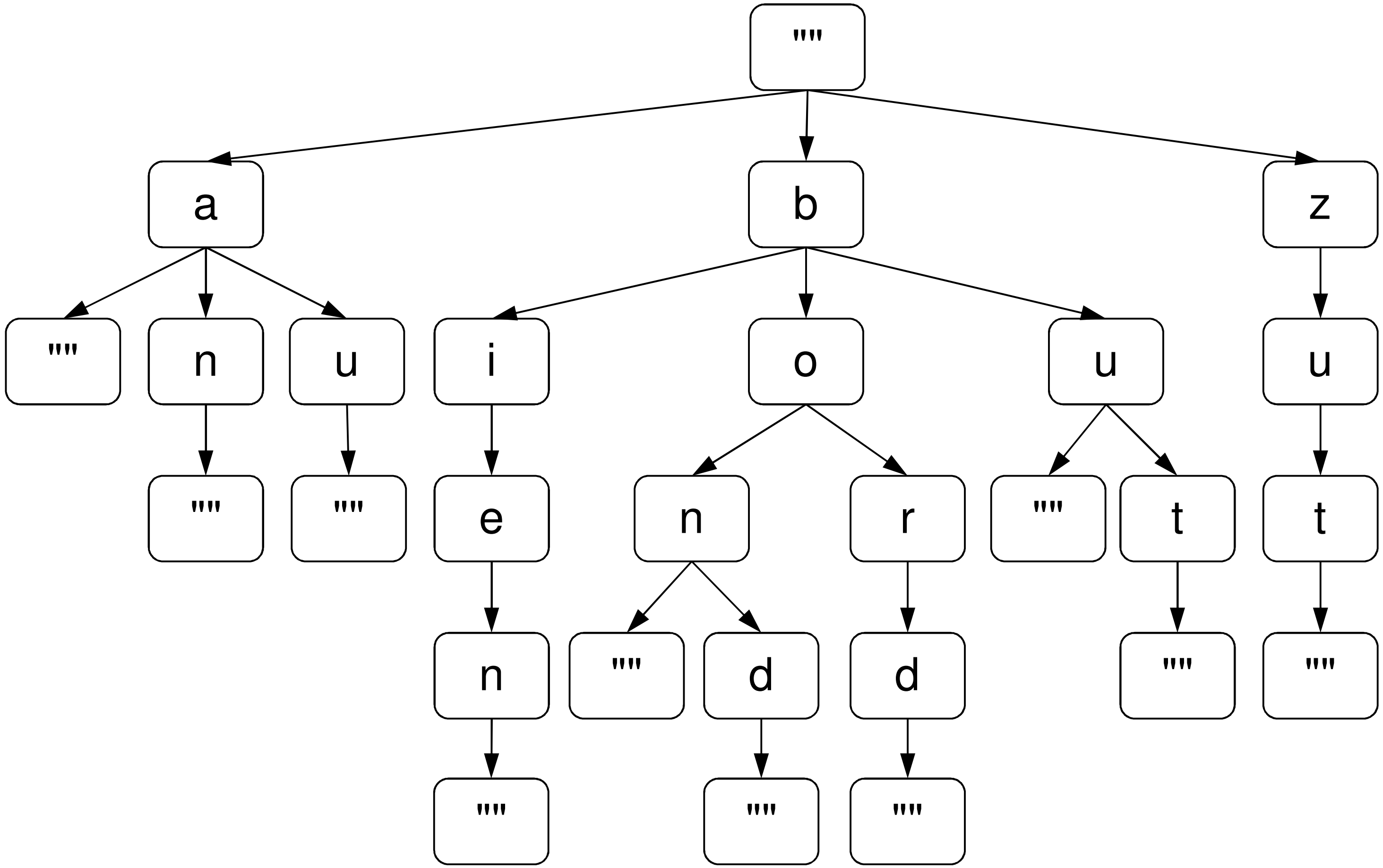
Développement d'un jeu *sur N coups maximum*

- Nœud = situation de jeu Racine = situation initiale
- Feuilles : situation non développables (N atteint ou fins de partie)
- Arc : coup jouable depuis une situation pour atteindre la situation suivante



Exemple 3 : représentation d'un **dictionnaire**

pour chaque mot ∃ un chemin unique de la racine jusqu'à une feuille



Dictionnaire = { a, an, au, bien, bon, bond, bord, bu, but, zut }

2.5.2 Exemples d’implémentations possibles

A chaque nœud, on associe une liste de nœuds fils

```
type Cellule;
type Liste is access Cellule;
type Nœud;
type Lien is access Nœud;
type Cellule is record
      Vers : Lien;
      Suiv : Liste;
end record;
type Noeud is record
      Info : Element;
      Fils : Liste;
end record;
```

– Exemple d'application

- Réseau informatique avec liste de connexion entre certains ordinateurs
- Déterminer si l'ordinateur x peut communiquer avec l'ordinateur y ?
- 1 ordinateur = 1 classe (= 1 arbre)
- Quand 2 ordinateurs ont une connexion ➔ **union** de 2 classes
- Si x et y sont dans la même classe (**find**) : ils peuvent communiquer

– Autre exemple (voir cours graphes 3MIC)

- Recherche d'arbres couvrants dans un graphe

2.6. Problème Union - Find

- **Nombreuses application en informatique**

  – On dispose d'une **partition** d'un ensemble d'éléments E, cad. ensemble des parties non vide de E telles que
    - sont 2 à 2 disjointes
    - la réunion des différentes parties (ou classes) de E = E

  – Maintien de cette partition ➔ opérations associées
    – Trouver la classe d'un élément (**Find**)
    – Faire l'union de 2 classes (**Union**)

  – **Structure de données pour représenter une partition ?**

  ➔ **représentation par une forêt**

- **Représentation d'une partition par un tableau**
  - Classe(x) : la classe d'un élément x de E

Exemple E = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

x	0	1	2	3	4	5	6	7	8	9
Classe(x)	2	3	1	4	4	1	2	4	1	4

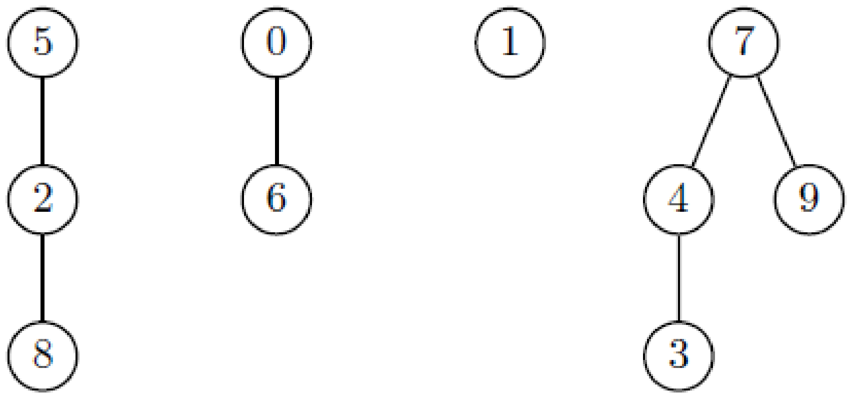
Partition : {{2, 5, 8}, {0, 6}, {1}, {3, 4, 7, 9}}

- Trouver la classe d'un élément : direct $O(1)$
- Faire l'union de 2 classes : $O(n)$

2.6. Union - Find

- **Représentation d'une partition par une forêt**
  - Pour chaque classe : 1 représentant (= un élément)
  - une classe = un arbre   une partition = une forêt
  - Racine de l'arbre = représentant de la classe

Exemple   E = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}



Partition : {{2, 5, 8}, {0, 6}, {1}, {3, 4, 7, 9}}

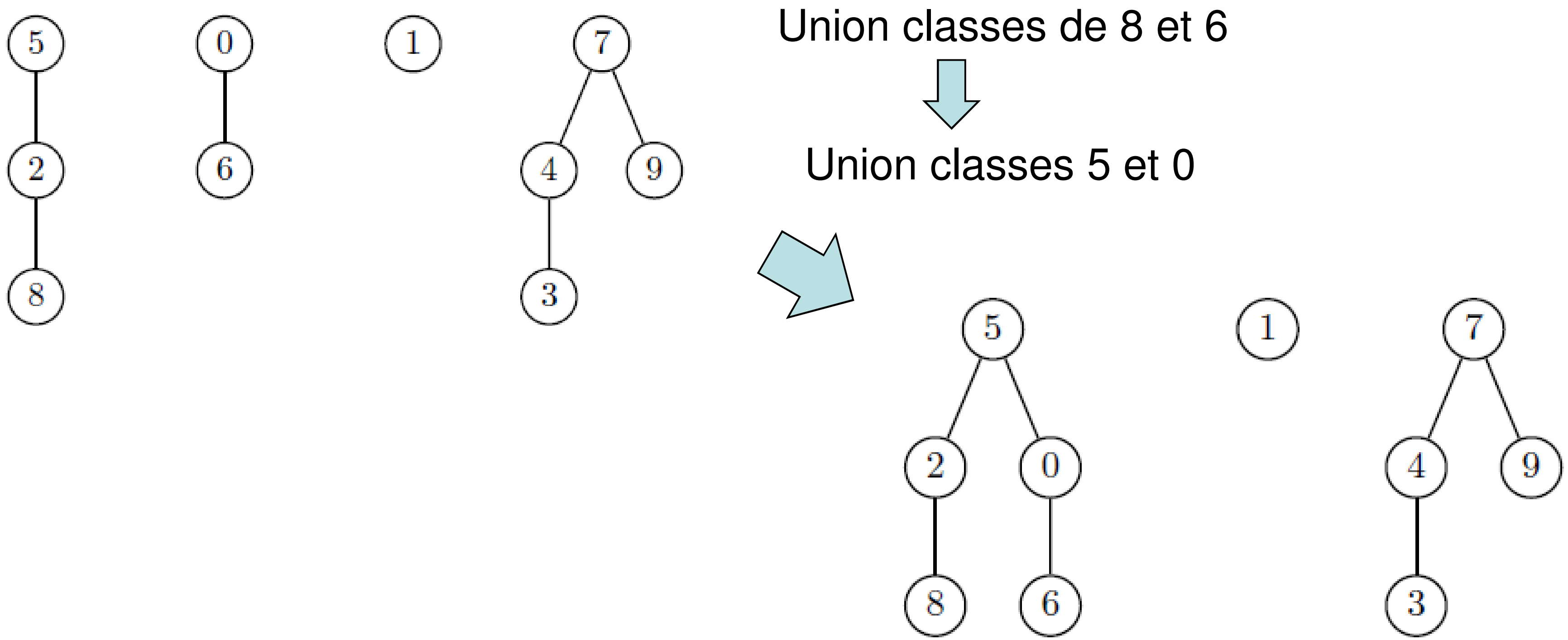
- **Représentation d'une forêt**
  - Tableau d'entiers donnant pour chaque élément son père dans l'arbre
  - Pour la racine r : père(r) = r   (par convention)

Exemple précédent

x	0	1	2	3	4	5	6	7	8	9
Pere(x)	0	1	5	4	7	5	0	7	2	7

  - Trouver la classe d'un élément x : trouver son représentant
    - Trouver son père, le père de son père … jusqu'à arrivée à la racine
    - Complexité : dépend de la hauteur de l'arbre ➔ O(h)

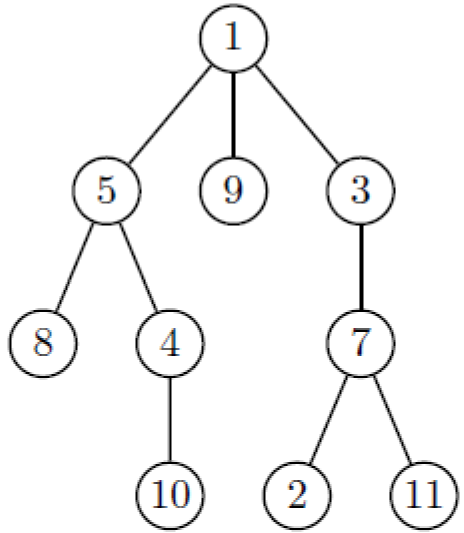
– Faire l’union de 2 classes
  - Ajouter la racine d’une classe comme fils de la racine de l’autre classe
  - **Règle** : la racine de l’arbre le plus petit devient le fils
  - Complexité : temps constant à partir des racines

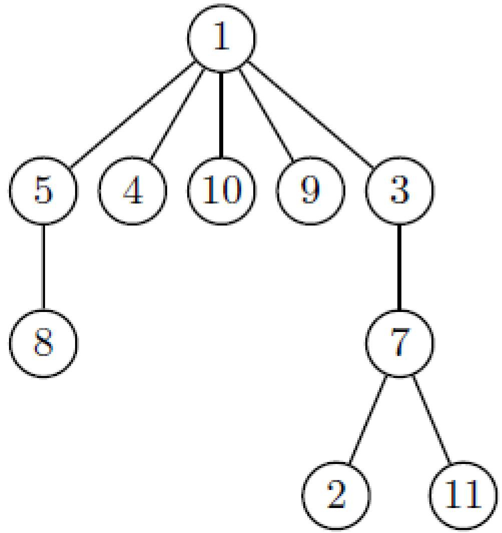


– Trouver la classe d'un élément
  - Amélioration lorsque plusieurs appels à l'opération **Trouver**:
    – faire remonter les éléments parcourus lors de la recherche comme fils de la racine
  - On parle de compression de chemins

  ➔ Arbre aplati

Exemple : trouver 10





2.5. Arbres quelconques

- Autre exemple : méthode de recherche arborescente
  - Exploration d'un ensemble de solutions
  - Arborescence implicite, développée par une méthode
  - Racine : aucune variable explorée / Nœud : choix d'une variable
  - Arcs : choix d'une valeur pour une variable
  - Feuille : solution