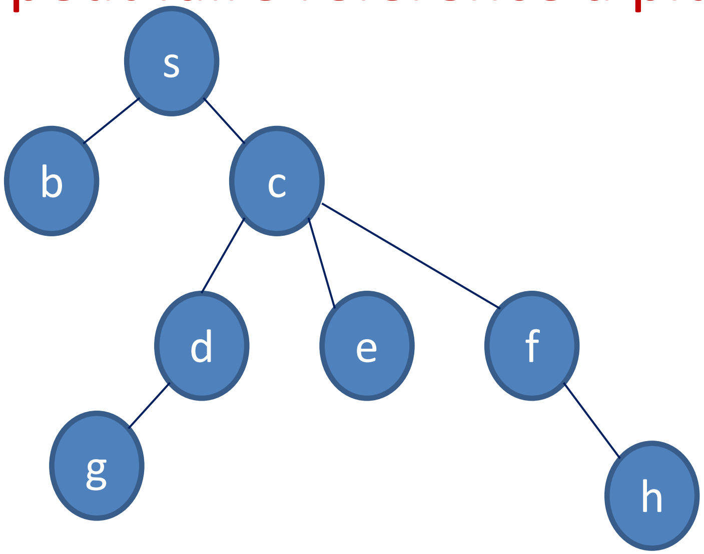
Structure « arbre »

L' arbre « quelconque »

- Un arbre de sommet s, *comme par exemple l'arbre généalogique*, est une structure chainée dans laquelle :
  - chaque élément (différent de « s ») est référencié une seule fois.
  - chaque élément peut faire référence à plusieurs autres éléments.

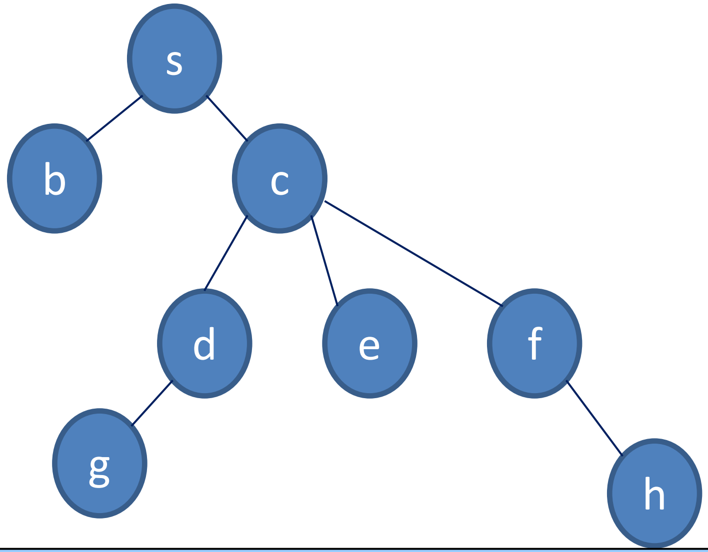
Exemple :



Définition d'un arbre quelconque

- Définition : Soit A un ensemble et « s » un élément de A
  - A est un arbre de sommet « s » si A est muni d'une relation binaire antisymétrique R (a R b ssi b est fils de a) telle que :
    - Pour tout a # s de existe une unique suite $b_0.,.b_n$ de A tel que $b_0 = s$ $b_n = a$ telle que b_{i-1} R b_i
    - A il « s » n'est fils d'aucun élément de A

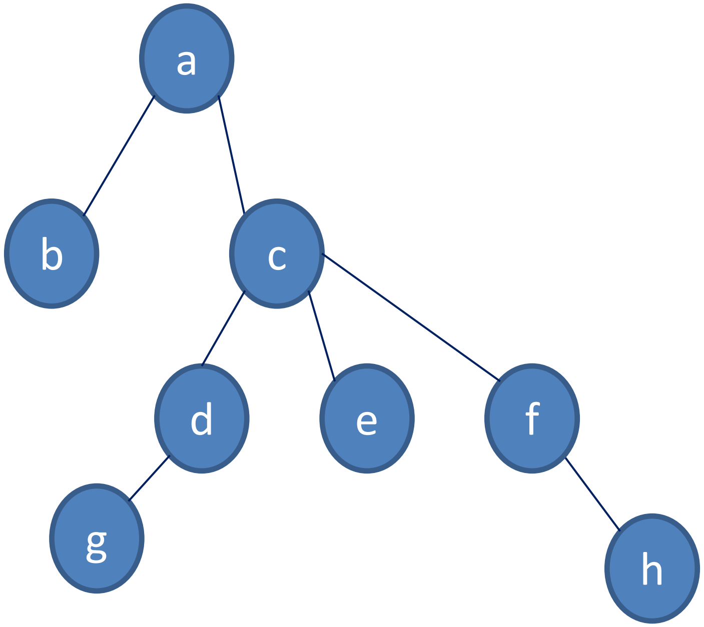
Exemple :



Terminologie usuelle

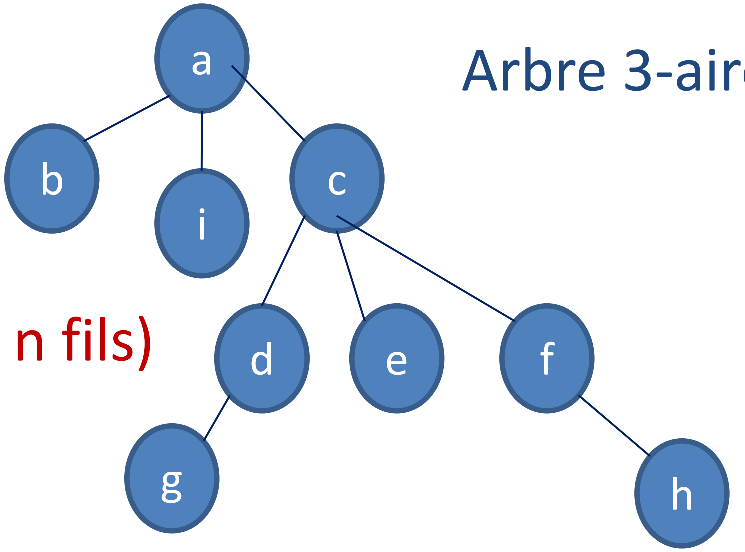
- Les éléments d'un arbre sont appelés **nœuds**.
- Le nœud d'entrée dans l'arbre qui permet d'accéder à tous les autres s'appelle **sommet** ou encore **racine** de la structure.
- Lorsqu'il y a un lien entre deux éléments, on parle de façon naturelle de **nœud père** et **nœud fils** .
- Un nœud qui n'a pas de fils est dit :

  nœud terminal ou **feuille** de l'arbre



Les arbres n-aires

- Un arbre n-aires est une structure chainée dans laquelle :
  - ➢ chaque élément est référencié une seule fois.
  - ➢ chaque élément peut faire référence à **n éléments** au maximum**.**

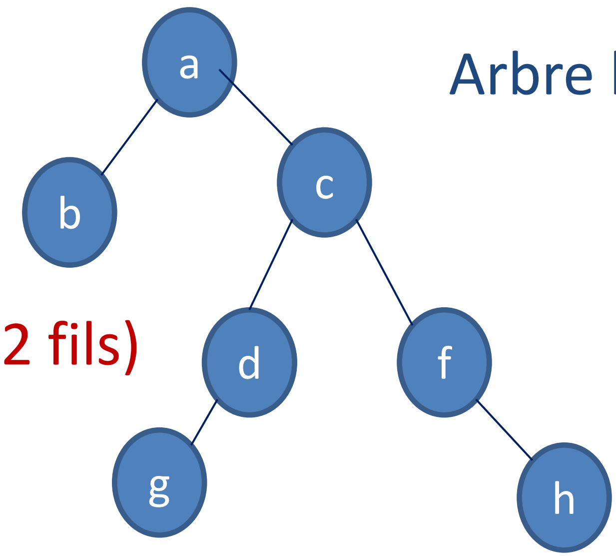


Arbre 3-aire

(chaque nœud a au maximum n fils)

Les arbres binaires

- Un arbre binaire est une structure chainée dans laquelle :
  - chaque élément est référencié une seule fois.
  - chaque élément peut faire référence au maximum à **2 éléments.**



Arbre binaire

(chaque nœud a au maximum 2 fils)

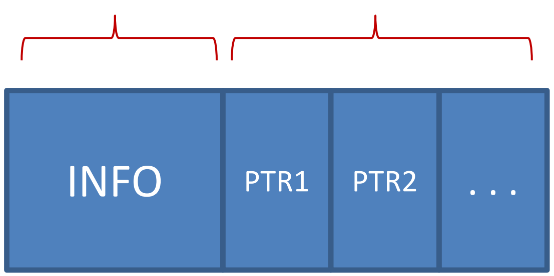
Réalisation de la structure arbre

- Évolutivité : → Variable dynamique

(Le nombre des nœuds varie)

- composition
  - Une partie information qui caractérise le nœud lui-même.
  - Une partie qui représente ses liens avec d'autres nœuds

    cette partie contient des pointeurs



Réalisation d’un arbre «quelconque»

- A la manière du livret de l’état civil:
  - le père permet d’accéder au fils ainé et,
  - chaque fils permet d’accéder à son frère cadet

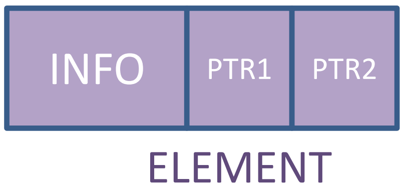
INFO : renseigne sur la partie information du nœud

PTR1 : indique l’adresse du fils ainé (ou NULL)

PTR2 : Indique l’adresse du frère cadet (ou NULL)

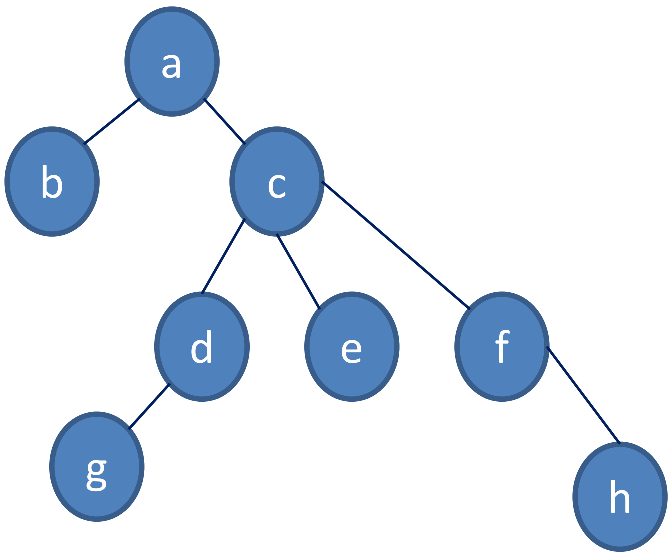
Réalisation d’un arbre «quelconque»

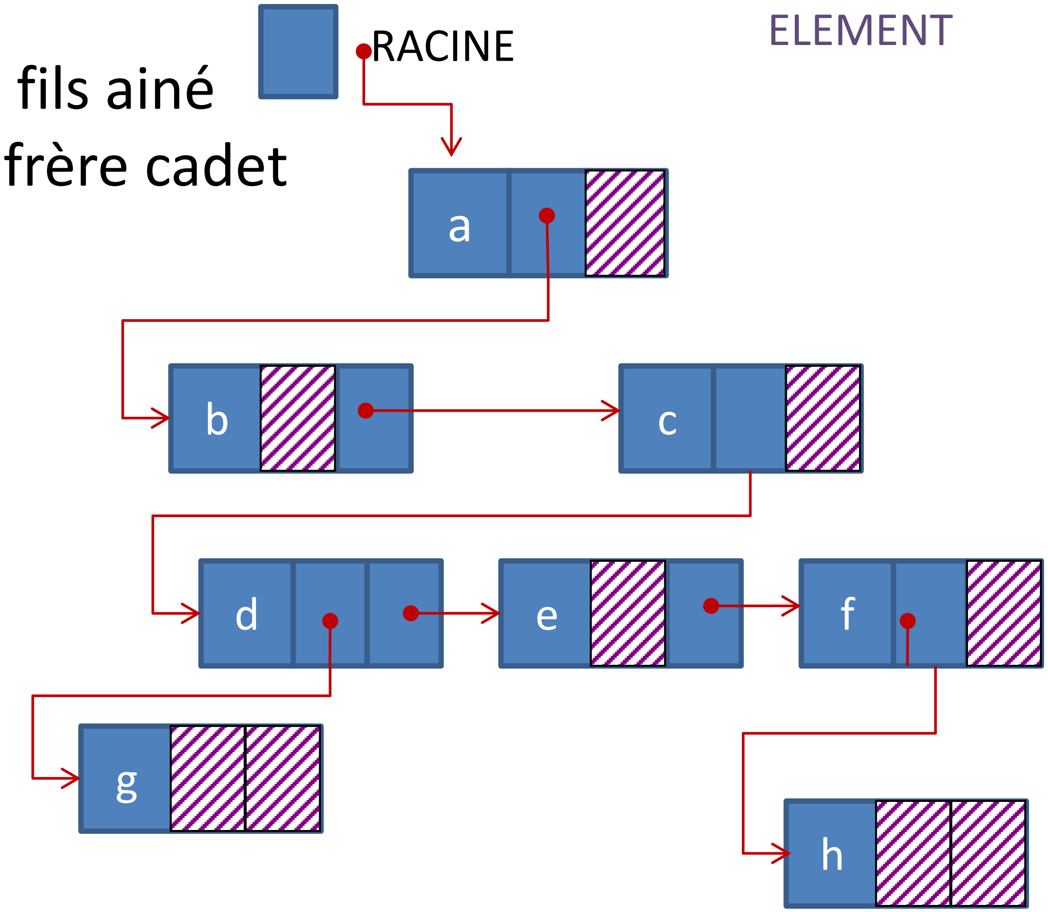
A chaque NŒUD, nous associons une var. dyn. de type



ELEMENT :

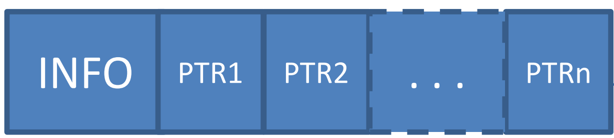
- PTR1 : indique l’adresse du fils ainé
- PTR2 : Indique l’adresse du frère cadet





Réalisation de l’arbre « n-aire »

- Chaque nœud à au maximum n fils :
  - le père permet d’accéder à tous ses fils.



INFO : renseigne sur la partie information du nœud

PTR1 : indique l’adresse du premier fils (ou NULL)

PTR2 : Indique l’adresse du deuxième fils (ou NULL)

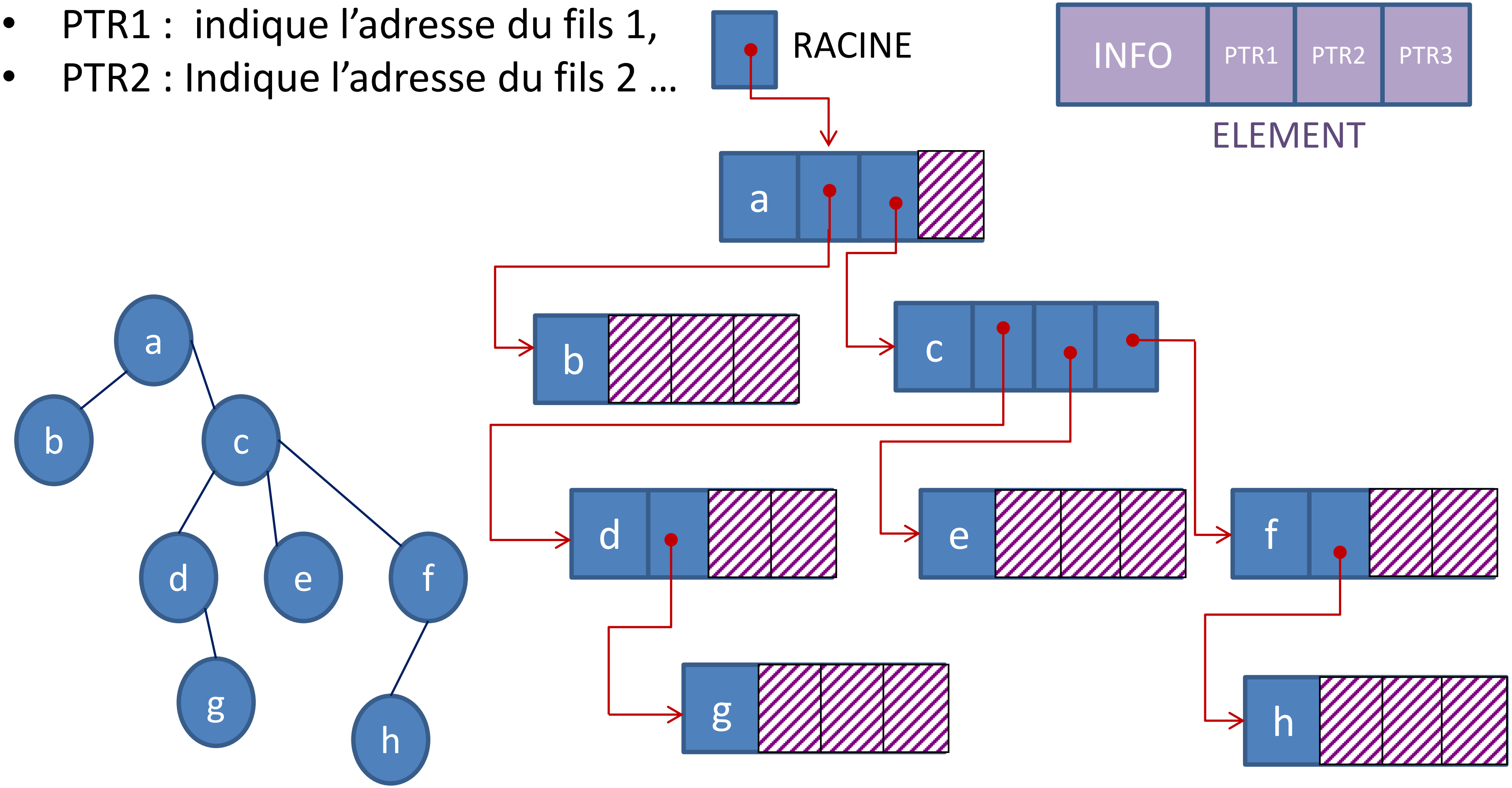
. . .

PTRn : Indique l’adresse du n-ème fils (ou NULL)

Réalisation de l'arbre n-aire (n=3) sans ordre sur les fils

A chaque NŒUD nous associons une variable dynamique de type ELEMENT :

- PTR1 : indique l'adresse du fils 1,
- PTR2 : Indique l'adresse du fils 2 …

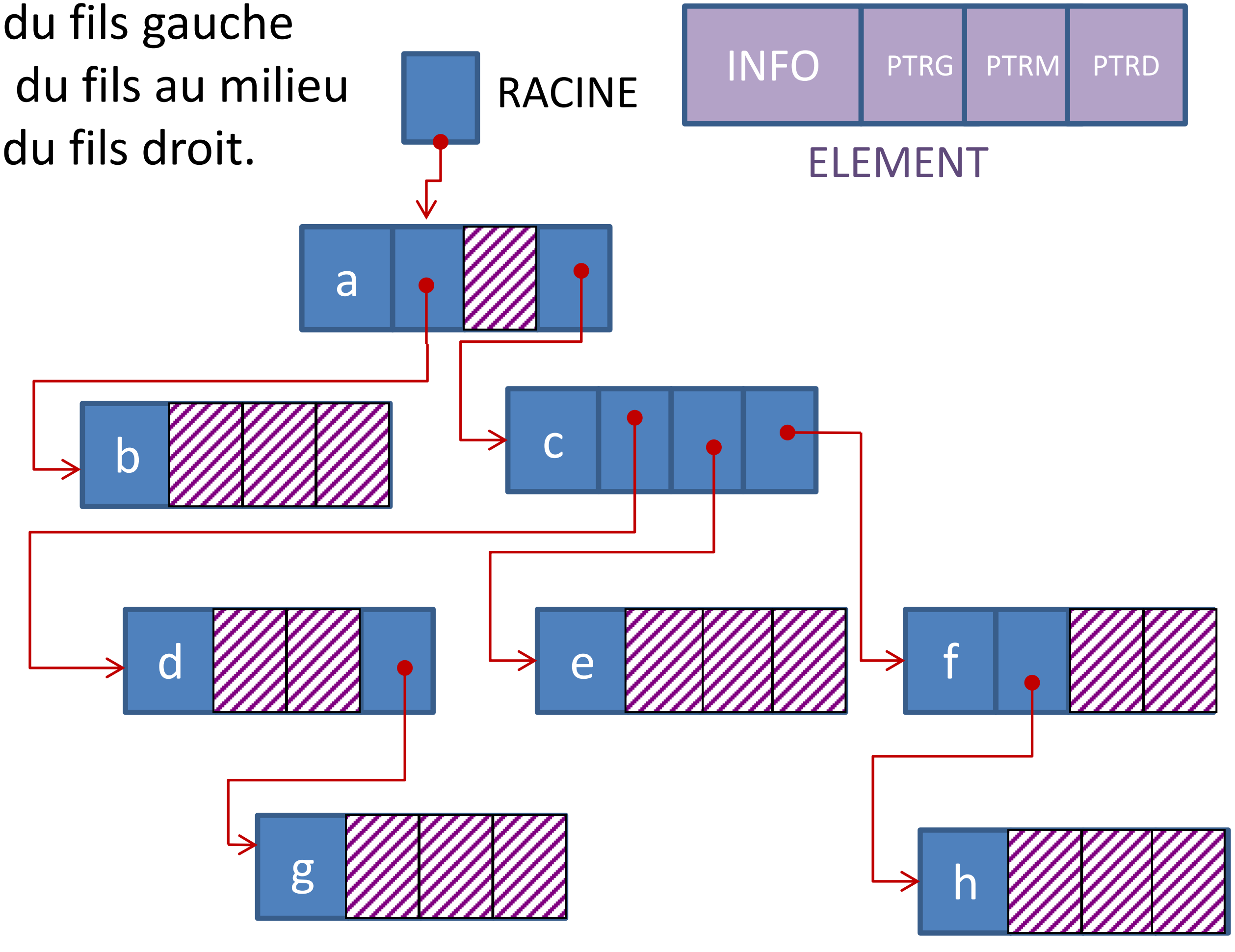


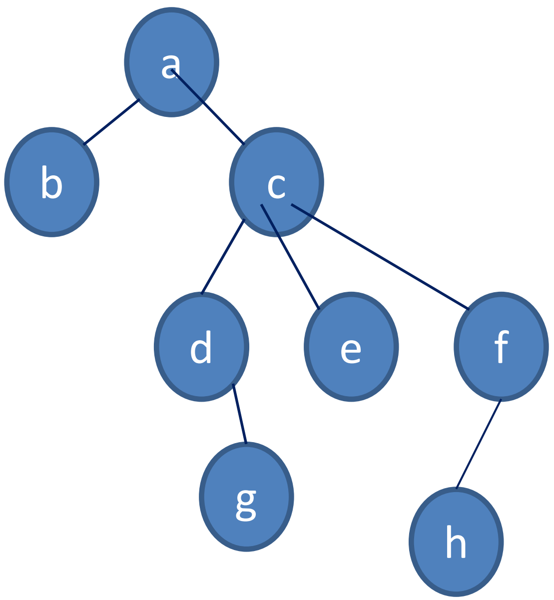
Réalisation de l’arbre n-aire (n=3)

Avec ordre : (Gauche, Milieu, Droit) sur les fils

A chaque NŒUD nous associons une variable dynamique de type ELEMENT :

- PTRG indique l’adresse du fils gauche
- PTRM indique l’adresse du fils au milieu
- PTRD indique l’adresse du fils droit.





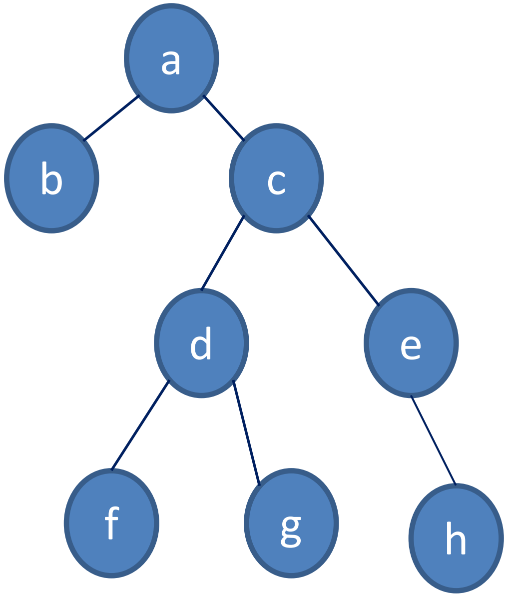
Réalisation de l'arbre binaire (n=2)

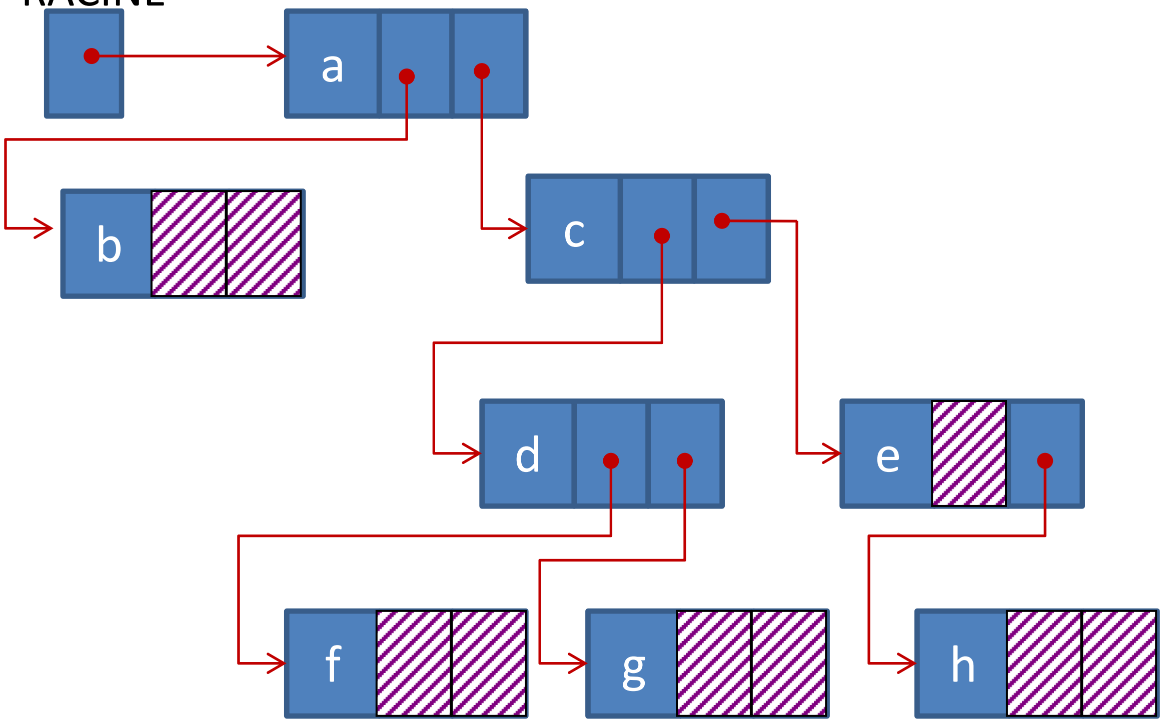
Avec ordre : (Gauche - Droit) sur les fils

A chaque NŒUD nous associons une variable dynamique de type ELEMENT :

- PTRG indique l'adresse du fils gauche
- PTRD indique l'adresse du fils droit

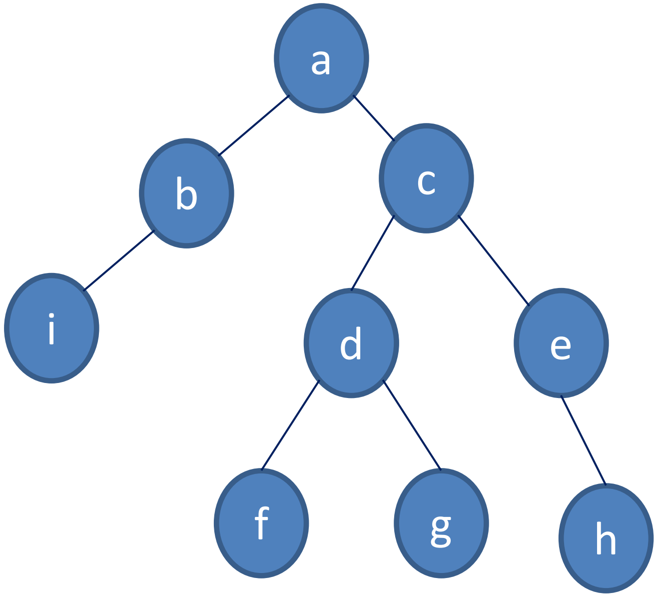
ELEMENT





Algorithme de parcours arbre binaire

- Un algorithme de parcours est un algorithme qui permet de passer en revue tous les éléments d'une structure; une seule fois chacun.



Idée :

```
Algorithme Parcours_arbre (racine ='a')
Début
    Afficher la sommet ;
    Parcourir le sous arbre droit ;
    Parcourir le sous arbre gauche ;
fin
```

Première ébauche :

```
Algorithme Parcours_arbre (racine ='a')
début
   Afficher Le sommet;
   Parcours_arbre (racine ='b') ;
Parcours_arbre (racine ='c') ;
fin
```

La récursivité : exemple

- Algorithme qui calcule n! (n factoriel)
- **Algo Factor**(n)

```
début
    p=1 ;
    Pour i=2,n
        p=p*i ;
    fin-pour
fin
```

- En utilisant la relation de récurrence : $n! = n * (n-1)!$
  - Pour calculer n! on calcule d'abord (n-1)!
  - Puis on multiplie le résultat par n.
  - . . . pour calculer (n-1)! On peut réutiliser la même relation de récurrence …

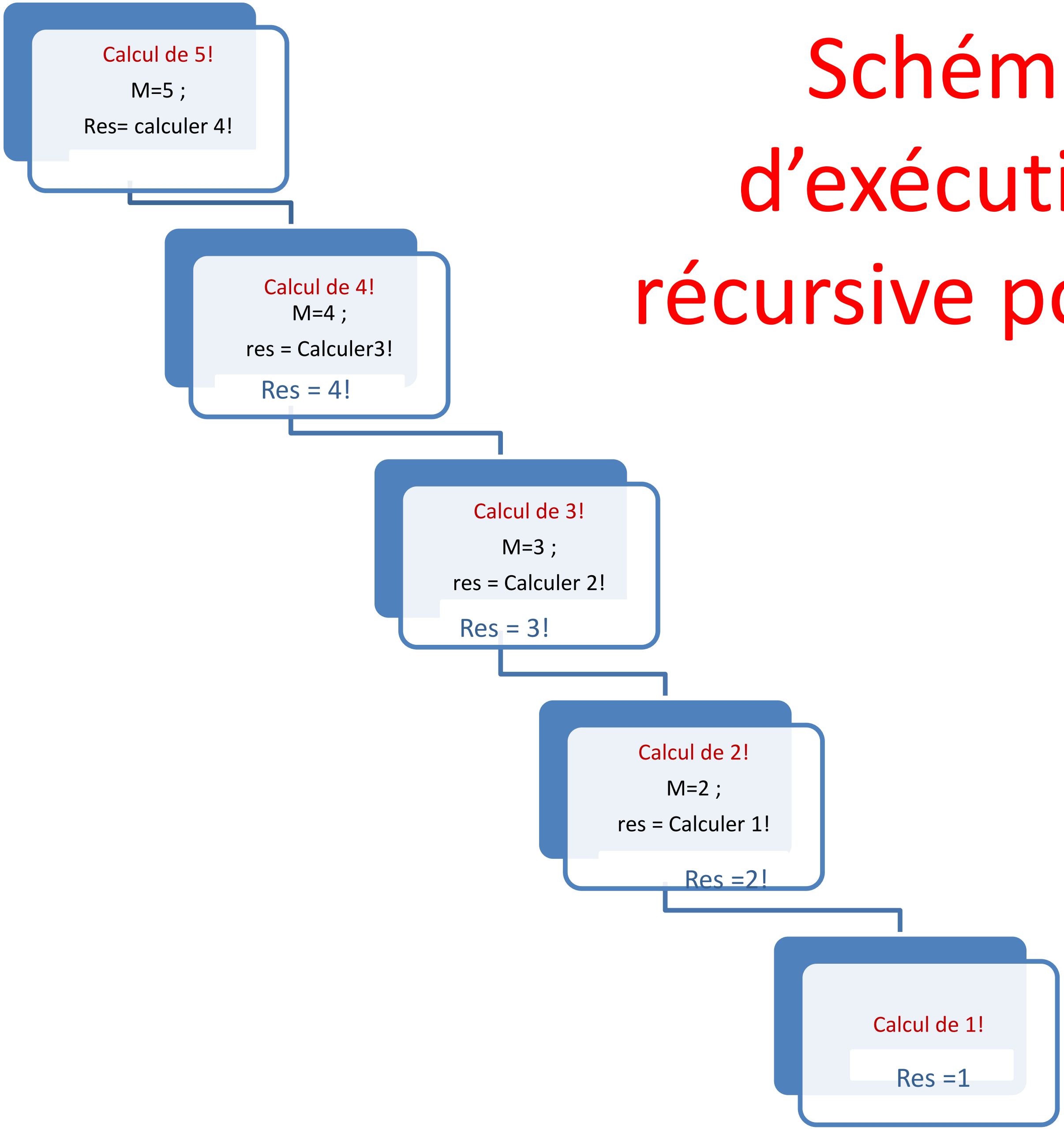
Une autre optique:

- Si ma tache est de calculer 5!
- Je met en mémoire la valeur 5
- Je donne la tache calculer 4! à un autre;
  - Sa tache est de calculer 4!
  - Il met en mémoire la valeur 4
  - Il donne la tache calculer 3! à un autre;
    - Sa tache est de calculer 3!
    - Il met en mémoire la valeur 3
    - Il donne la tache calculer 2! à un autre;
      - Sa tache est de calculer 2!
      - Il met en mémoire la valeur 2
      - Il donne la tache calculer 1! à un autre
        - » Comme 1! = 1 ce dernier transmet le résultat 1 à son appelant.

Rédaction de l’algorithme

```
Factorecursif(n) de type entier
Début
    Si (n=1)
      Retourner ( 1 )
    Sinon
      Retourner ( n*Factorecursif(n-1) )
fin
```

Schéma d’exécution récursive pour 5!



Rappel (langage C) . . . Le passage de paramètre par valeur

Exécution d'une fonction

```
float Afonc (int I, float X)
{
int j;
int Afonc=X;
for (j=1;j<I;j++)
     Afonc = Afonc *X;
return Afonc;   }
```

Lors de l'appel de la fonction les variables I, X prennent les valeurs que fixe la fonction appelante

Exemple *de fonction appelante*:

```
float z, y = 7;
int K= 3;
z = Afonc(K, y)
```

I prend la valeur de K (i.e. 3)

X prend la valeur de y (i.e. 7)

Cette fonction utilise :

- un entier I paramètre de passage
- un réel X paramètre de passage
- un entier j variable locale interne

Elle renvoie un réel :

Afonc paramètre de passage

Remarque :

les paramètres de passage sont des variables internes initialisables lors de l'appel de la fonction Afonc

A la fin de l'exécution de Afonc résultat réel Afonc est copié dans z

Rappel (langage C) . . . Ce qui se passe lors d'1 appel de fonction

Une fonction A appel une autre fonction B

Système (compilateur & S.E.)

- Donne la mémoire pour les variables locales de B
- Initialise toutes les variables de passage de B à l'aide des valeurs fixées par l'appelant

Système (compilateur & S.E.)

- Exécute les séquences de programmation de B
- Affecte le résultat à la variable typée paramètre de passage côté appelant

Rappel sur la récursivité
2- Le principe

- Etant donné P_n un problème d'ordre n
- La récursivité, lorsque elle est possible, a pour principe:
  - Ecrire P_n en $\mathcal{F}$onction de P_{n-1}: $P_n = \mathcal{F}(P_{n-1})$ où P_{n-1} présente les mêmes aspects que P_n mais à l'ordre n-1
  - A l'aide de $\mathcal{F}$, écrire P_{n-1}, puis P_{n-2} et ainsi de suite
  - Comme n est fini on est sûr de converger vers une situation triviale (calcul de P_1 ou P_0)
  - Une fois P_1 calculé, on remonte à l'aide de $\mathcal{F}$ à P_2, P_3 ,…jusqu'à P_n

3- Quelques critères pour réussir une récursivité

- Pour espérer converger la taille du problème doit diminuer suite à (k) appels récursifs.
- L'algorithme récursif doit toujours contenir un teste:
  - Ce teste caractérise le cas trivial.
  - Il met fin aux appels récursifs et évite la boucle infinie.
- La structure Sn sur laquelle porte P doit admettre une définition récursive (liste, arbre,..)

4- Quelques définitions récursives

- Une liste chainée non vide est constituée
  - soit d'un élément (l'élément de tète)
  - soit d'un élément suivit d'un liste chainée.
- Un arbre binaire non vide est
  - soit un élément (sommet) ,
  - soit un élément suivit de deux arbres binaires.

Comment « SYSTÈME » gère t-il la récursivité ?

```
. . .
int factr( int n)
{if   (n==1)  return ( 1 );
else return ( n*factr(n-1) );}
. . .
```

```
. . .
Main()
. . .
A= factr(3)
```

Exécute factr(3)
- attribue mémoire pour n et factr
- initialise n à 3
- test n=? 1
- Exécute factr(2)
- Retour 3*factr(2)

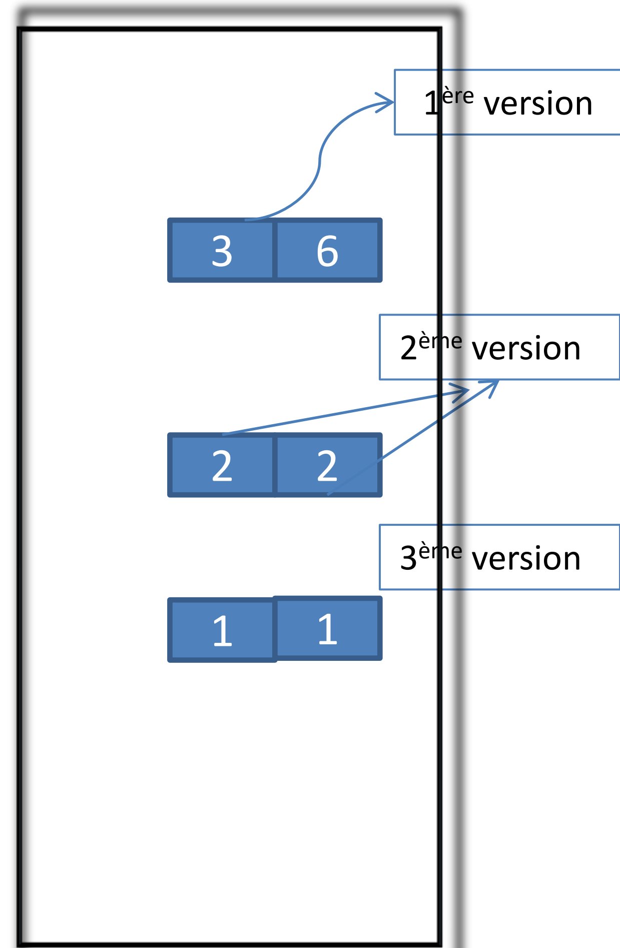
1ère version

Exécute factr(2)
- attribue mémoire pour n et factr
- initialise n à 2
- test n=? 1
- Exécute factr(1)
- Retour 2*factr(1)

2ème version

Exécute factr(1)
- attribue mémoire pour n et factr
- initialise n à 1
- test n=? 1
- Retour factr=1

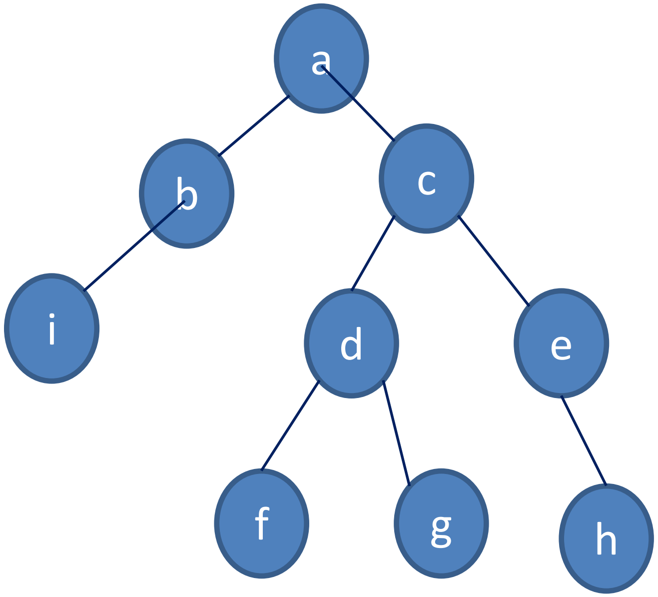
3ème version



Mémoire centrale

Algorithme récursif de parcours

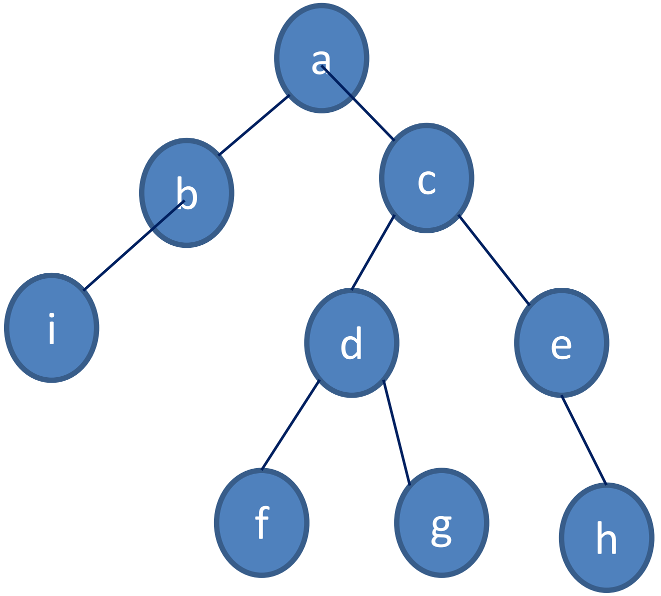
Algorithme **ParcourArbre** (RACINE)
Début
si (RACINE # NULL)
Afficher (RACINE->INFO) ;
si (RACINE > PTRD #NULL)
ParcouraArbre (RACINE -> PTRD) ;
si (RACINE > PTRG # NULL)
ParcourArbre (RACINE -> PTRG) ;
fin



Affichage résultant : a , c , e , h , d , g , f , b , i

Algorithmes de parcours Préfixe

```
Algorithme prefixe (RACINE )
Début
        si (RACINE # NULL)
                Afficher (RACINE-> INFO) ;
        si (RACINE ->PTRD #NULL)
                Prefixe (RACINE -> PTRD) ;
        si (RACINE ->PTRG # NULL)
                Prefixe (RACINE -> PTRG) ;
fin
```

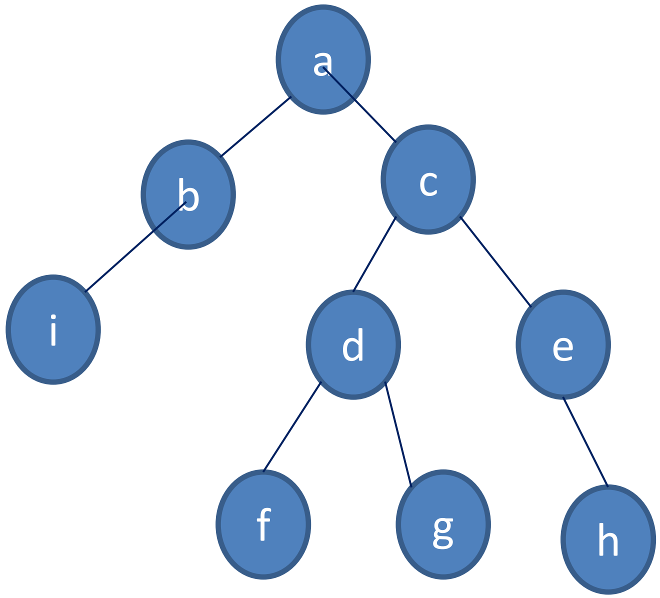


Affichage résultant : a , c , e , h , d , g , f , b , i

Algorithmes de parcours Postfixe

```
Algorithme postfixe (RACINE )
Début
        si (RACINE ->PTRD # NULL)
                Postfixe (RACINE -> PTRD) ;
        si (RACINE -> PTRG # NULL)
                Postfixe (RACINE ->PTRG) ;
        si (RACINE # NULL)
                Afficher (RACINE-> INFO) ;

fin
```

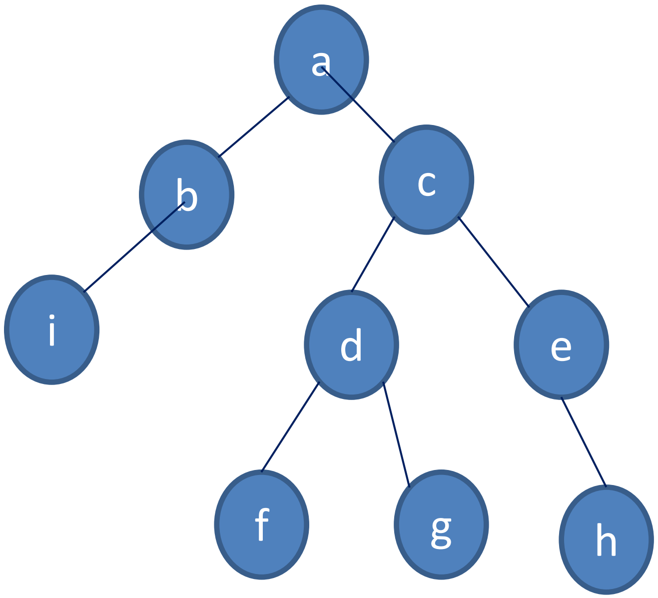


Affichage résultant : h , e , g , f , d , c , i , b , a

Algorithmes de parcours infixe

```
Algorithme infixe (RACINE )
Début
        si (RACINE -> PTRD # NULL)
                infixe (RACINE ->  PTRD) ;
        si (RACINE # NULL)
                Afficher (RACINE-> INFO) ;
        si (RACINE -> PTRG #NULL)
                infixe (RACINE ->PTRG) ;

fin
```

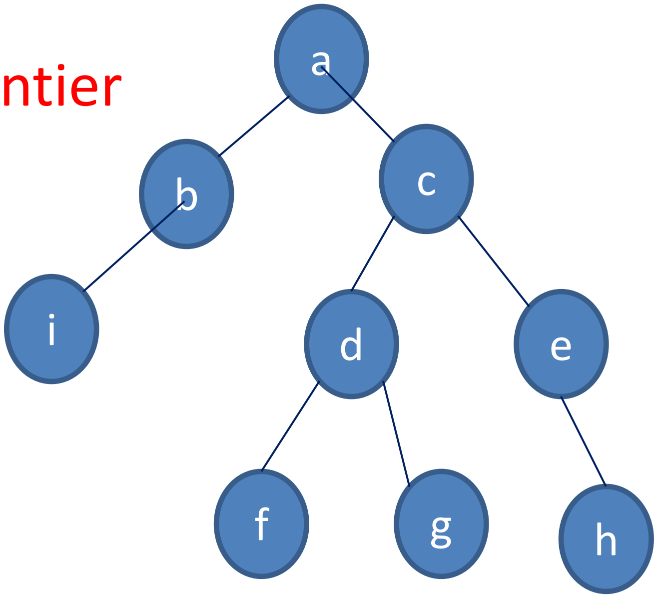


Affichage résultant : h , e , c , g, d , f, a, b , i

Application des algorithmes de parcours

On utilisera les algorithmes de parcours comme « squelette » pour d'autres algorithmes : nombre de nœuds de la structure

Algorithme **Precompte** (RACINE) de type entier
Début
    si (RACINE #NULL)
        retourner (1 **+**
        Precompte (RACINE ->PTRD) **+**
        Precompte (RACINE ->PTRG) **)** ;
    sinon
        retourner (0)
fin



Application des algorithmes de parcours

On utilisera les algorithmes de parcours comme « squelette » pour d'autres algorithmes : plus grand élément de la structure

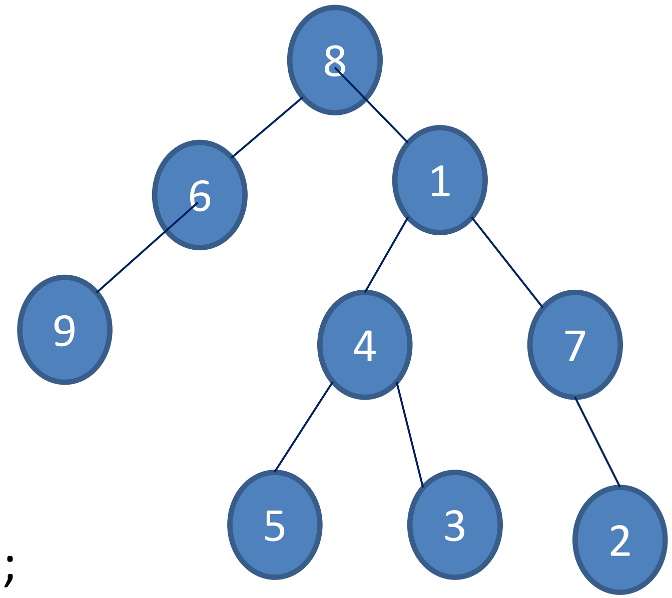
Rem: on doit s'assurer au préalable que la structure n'est pas vide

Algorithme **Postmax** (RACINE) de type entier
Début
    msg , msd de type entier ;
    msd <- RACINE->INFO ;
    msg <- RACINE->INFO ;

    si (RACINE ->PTRD # NULL)
      msd <- **Postmax** (RACINE ->PTRD)

    si (RACINE ->PTRG #NULL
      msg <- **Postmax** (RACINE ->PTRG)

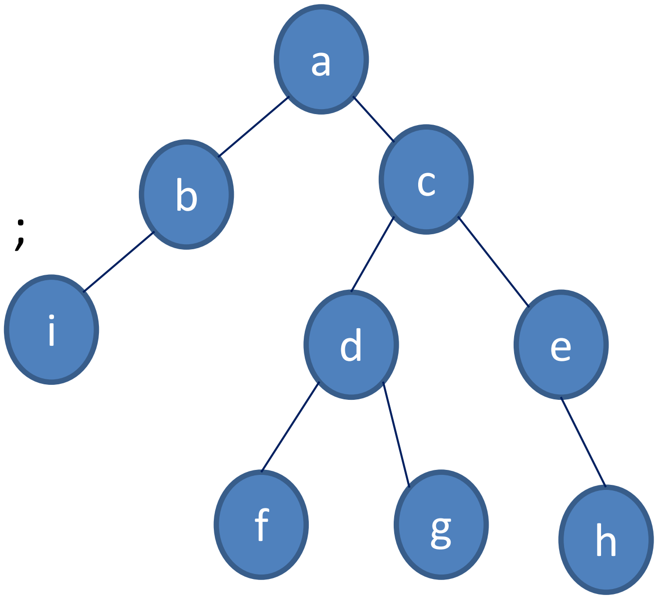
    retourner max (RACINE-> INFO ,**msd** , **msg**) ;
fin



Algorithmes d'affichage et de niveau

Rem: premier appel
prefixe&niv (RACINE,0)

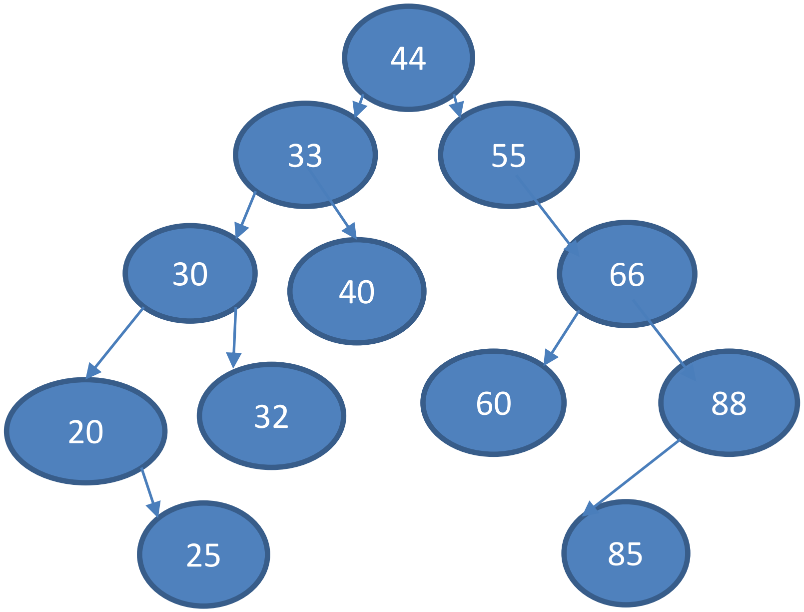
Algorithme **prefixe&niv** (RACINE,niv)
Début
    si (RACINE #NULL)
        Afficher (RACINE->INFO, niv) ;
    si (RACINE ->PTRD # NULL)
        Prefix&niv (RACINE -> PTRD,niv+1) ;
    si (RACINE -> PTRG #NULL)
        Prefixe&niv (RACINE -> PTRG,niv+1) ;
fin



Affichage résultant : a 1 , c 2 , e 3, h 4 , d 3, g 4 , f 4, b 2 , i 3

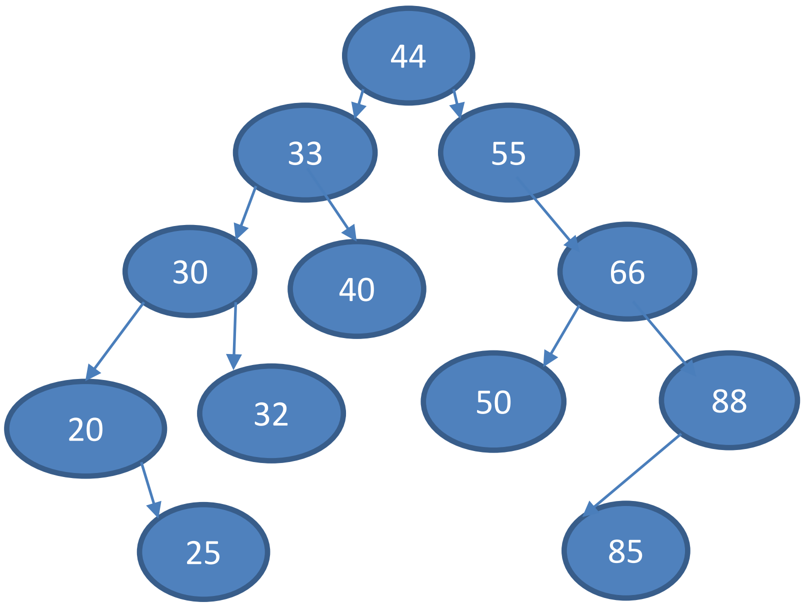
Arbre binaire de recherche

Arbre binaire de recherche ou arbre ordonne



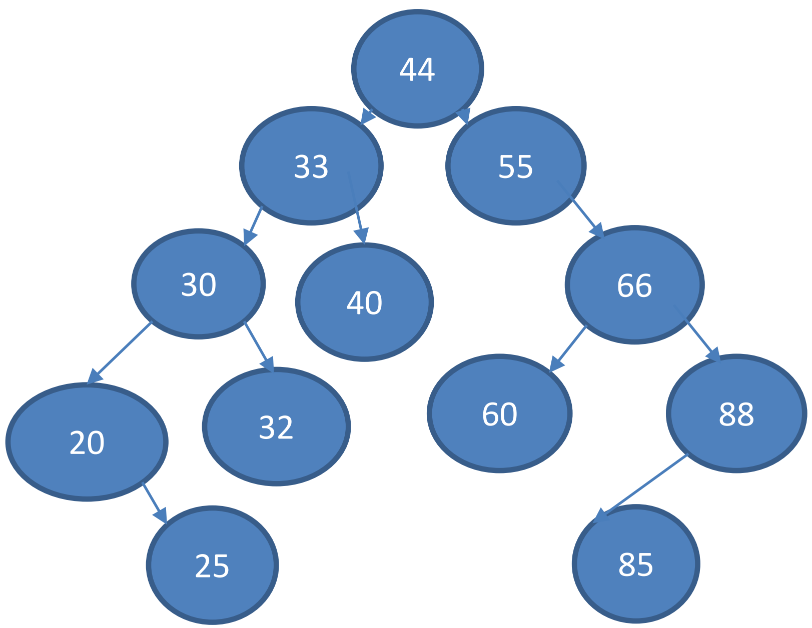
Arbre binaire dans lequel **pour chaque nœud,** tous les éléments du sous arbre droit sont pus grands que le nœud
et tous les nœuds du sous arbre gauche sont plus petit que le nœud

ATTENTION l'ordre est relatif aux sous arbres et non pas uniquement aux fils



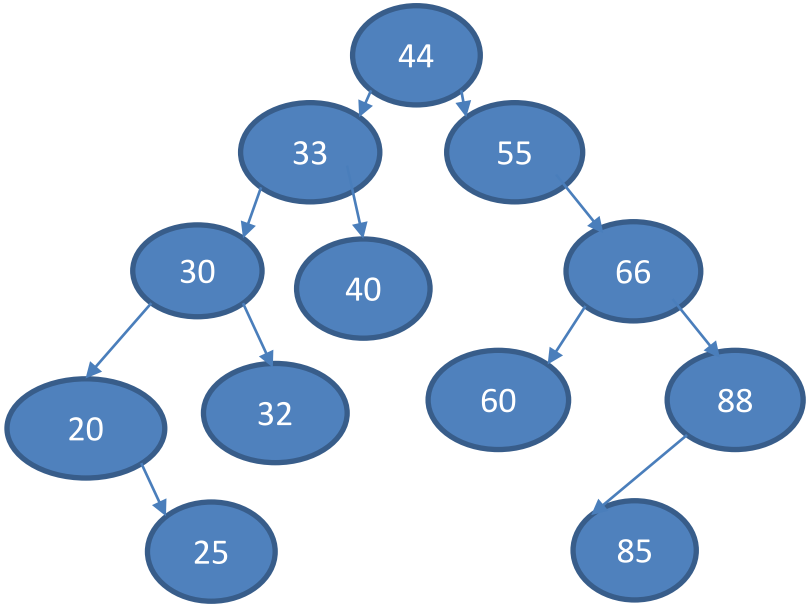
CECI n'est pas un arbre de recherche le nœud 50 est situé dans le sous arbre droit de 55

Algorithme de recherche



Comment savoir si 14 se trouve ou non dans l'arbre?
Comparons 14 à la racine 44
Nous déduisons qu'il faut chercher dans le sous arbre gauche:
Puisque 14<44 Alors 14 < tout nœud du sous arbre droit
Nous réappliquons la même méthode avec 33 jusqu'au nœud :
20 (on a 14 < 20 ET ce nœud n'a pas de fils gauche)
Donc nous pouvons conclure que 14 n'est pas dans cet arbre

Algorithme de recherche Avantage :



Lors de l'exemple précédant, le premier test avec la racine a permis d'éliminer tous les éléments du sous arbre droit (car, par définition ils sont tous plus grands que 44)

Rappelons que dans les tableaux, lors de la recherche par dichotomie une comparaison avec l'élément du milieu permettait aussi de disqualifier la moitié des éléments du domaine de recherche

Nous porions donc prétendre à une complexité algorithmique identique à condition de bien organiser l'arbre binaire : il faudrait pour cela que, pour chaque nœud, le nombre d'élément de son SRD soit à peu-près égale au nombre d'élément du SAG