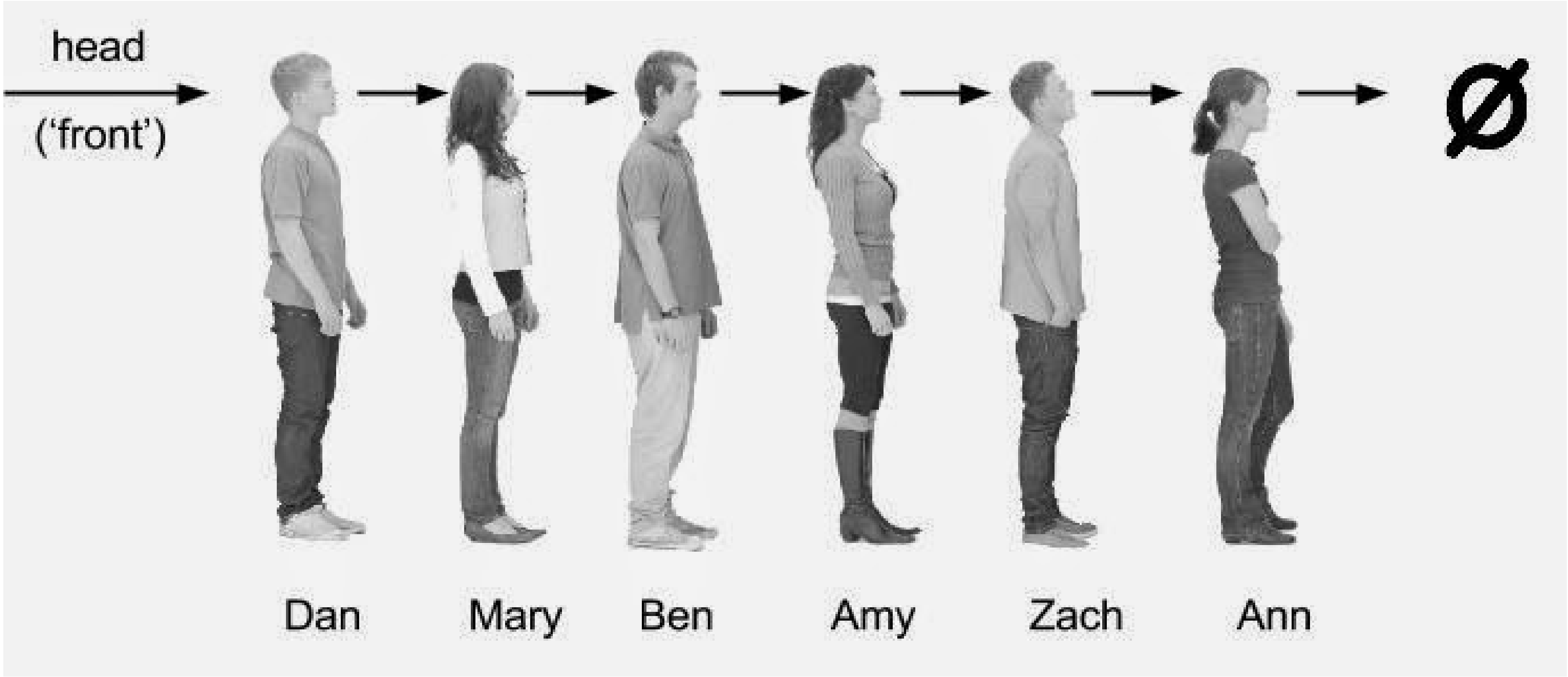


STRUCTURES DE DONNÉES ET ALGORITHMES FONDAMENTAUX

Initiation au développement (DEV)

RAPPELS DES ÉPISODES PRÉCÉDENTS

Tableaux

- stockage contigu d'un ensemble d'éléments
- Accès en O(1)
- Insertion/suppression en O(n)

Listes chaînées

- Liens entre des éléments « éparpillés » en mémoire
- Accès en O(n)
- Insertion/suppression en O(1)

Dans les deux cas

- Recherche en O(n) *(O(log(n)) si tableau trié)*

DEUX NOUVELLES STRUCTURES DE DONNÉES

Arbres binaires (de recherche!)

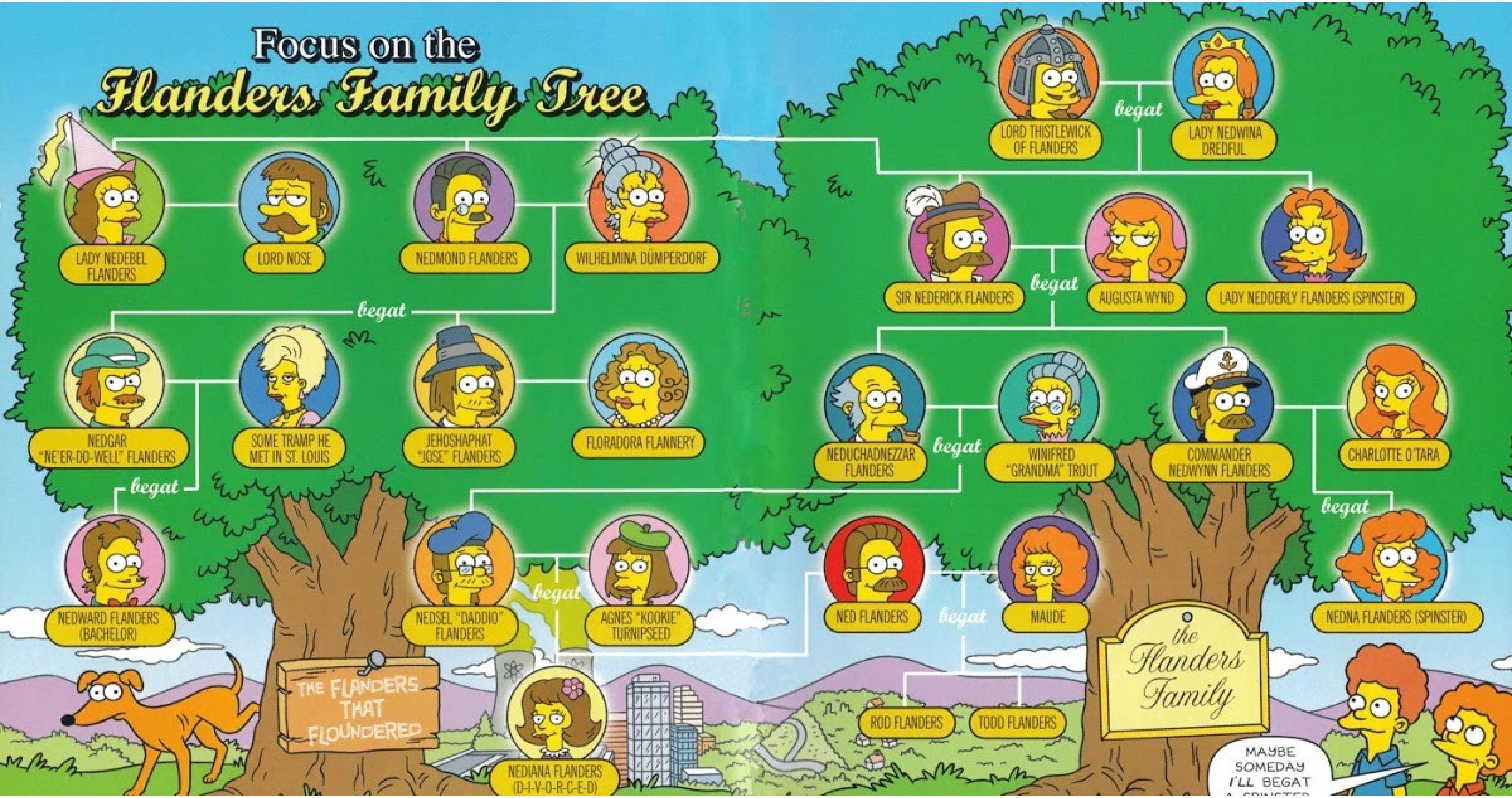
Extension de la liste chaînée, version simple de graphes

Tables de hachage

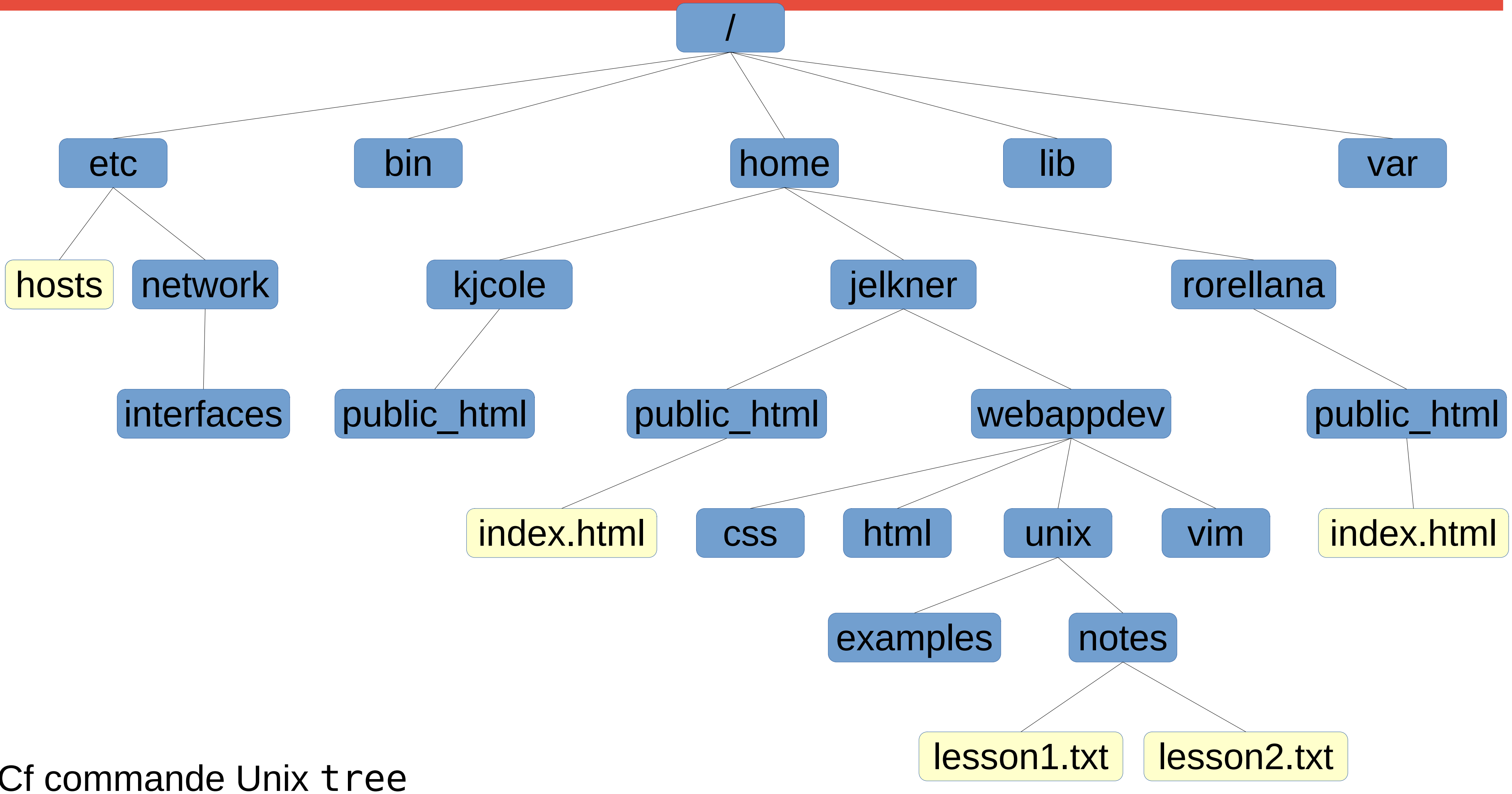
Permet en particulier de créer des tableaux associatifs

Objectif : Temps de recherche réduit

STRUCTURES ARBORESCENTES

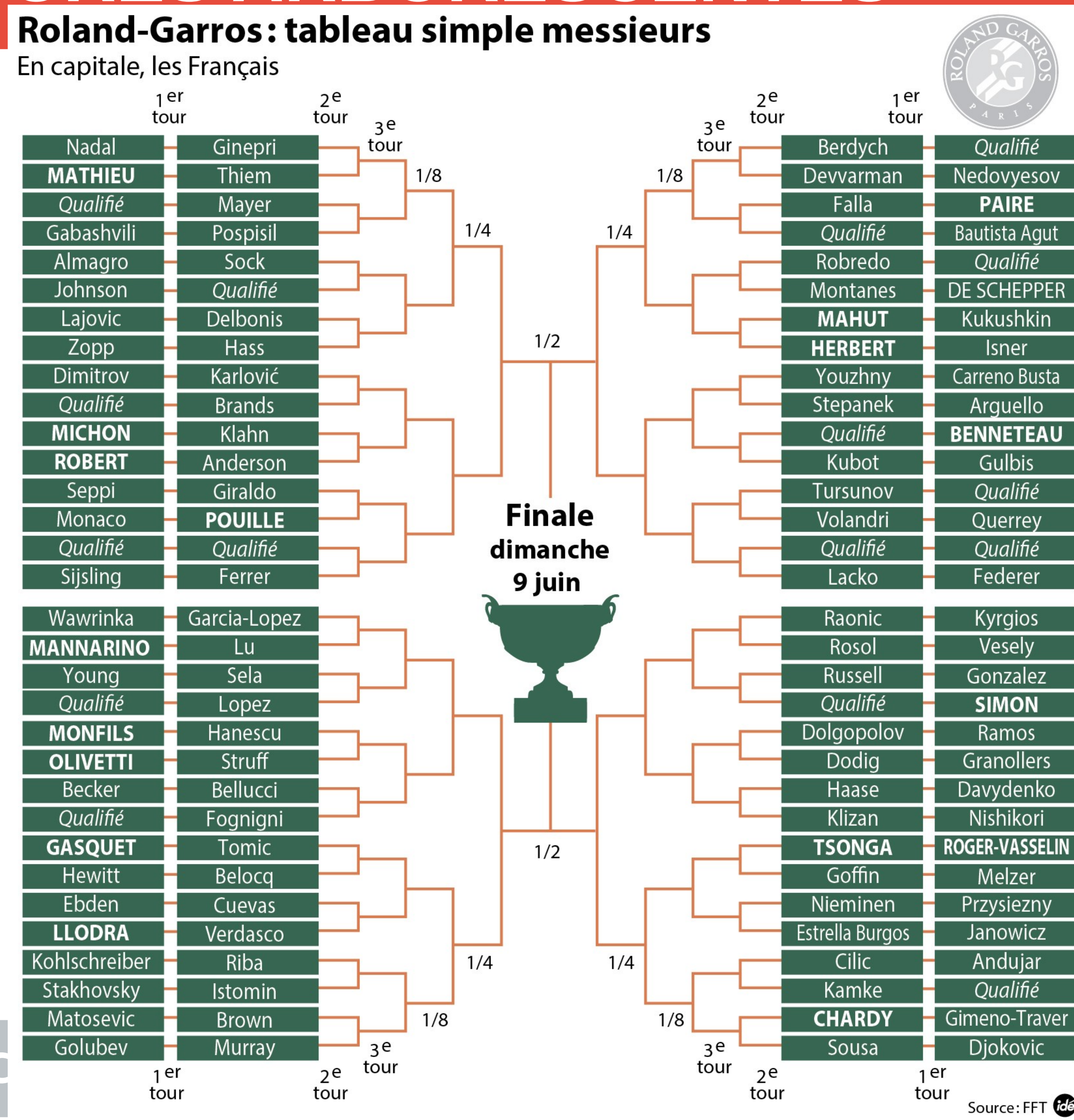


STRUCTURES ARBORESCENTES

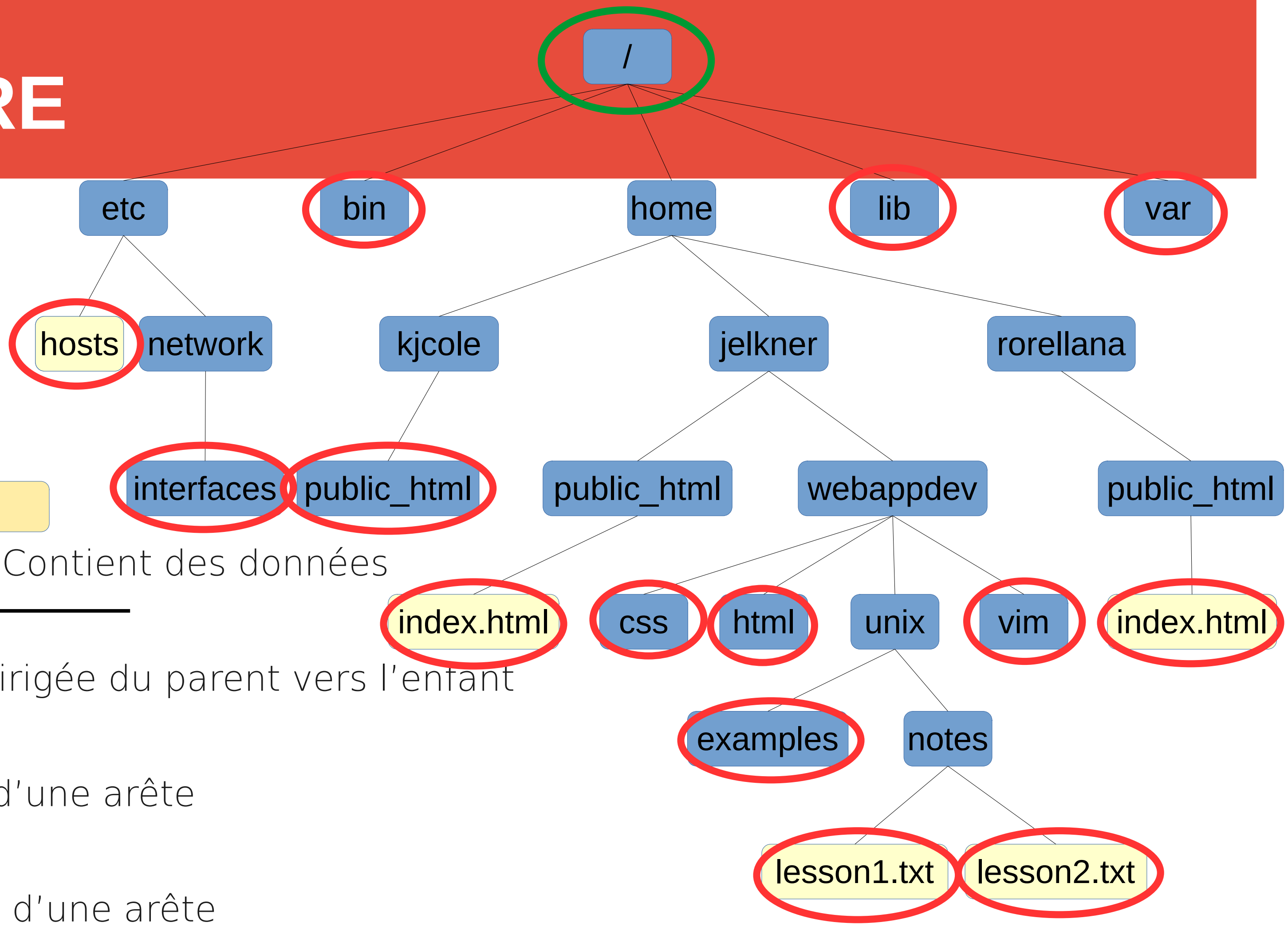


Cf commande Unix `tree`

STRUCTURES ARBORESCENTES



VOCABULAIRE



Nœud (*node*)

Cellule de base de l'arbre. Contient des données

Branche (*branch*)

Lien entre deux cellules. Dirigée du parent vers l'enfant

Enfant (*child*)

Nœud à l'extrémité finale d'une arête

Parent (*parent*)

Nœud à l'extrémité initiale d'une arête

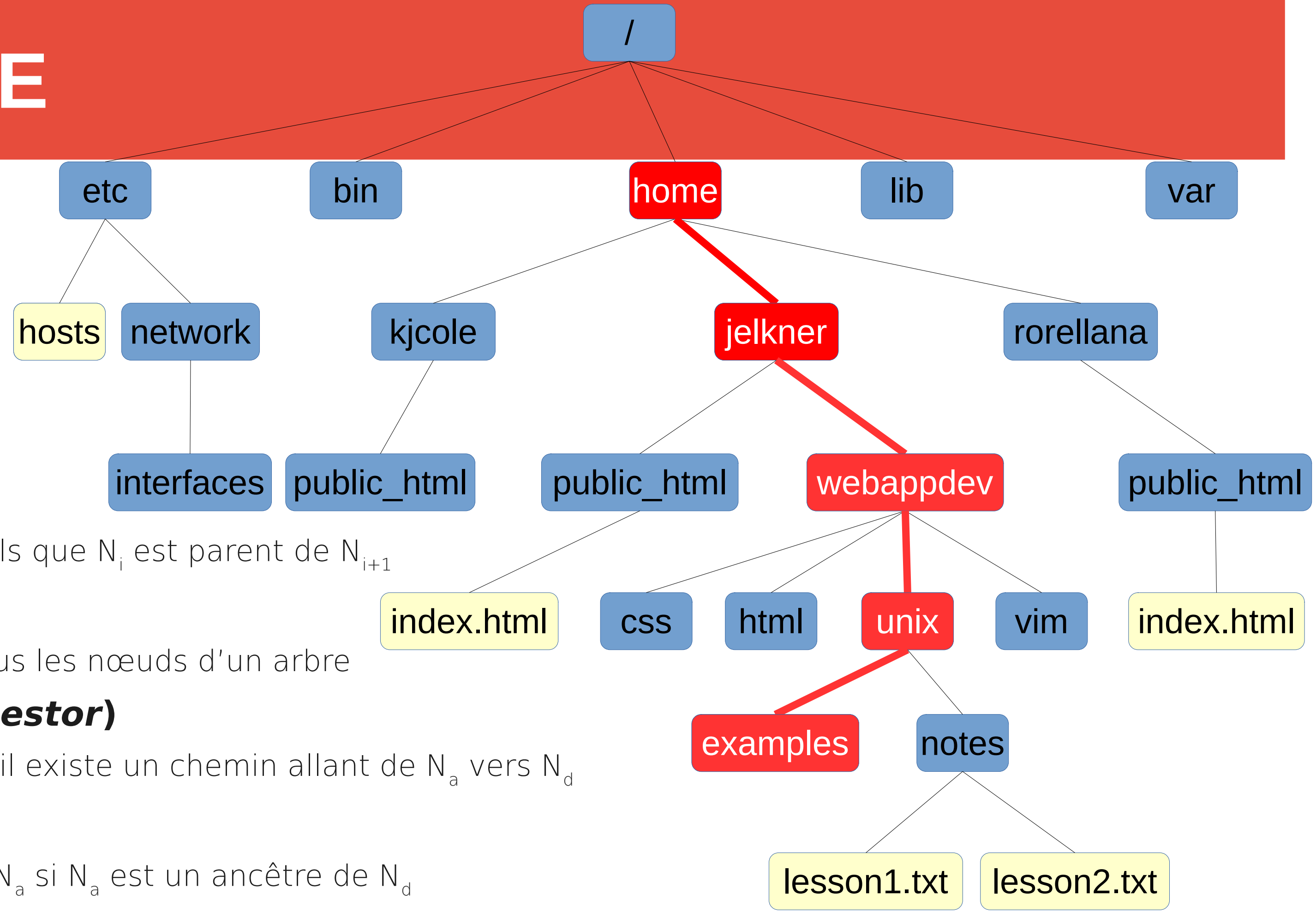
Racine (*root*)

Nœud sans parent

Feuille (*leaf*)

Nœud sans enfant

VOCABULAIRE



Chemin (*path*)

Séquence de nœuds $N_1, N_2, \ldots N_k$ tels que N_i est parent de N_{i+1}

Parcours (*traversal*)

Algorithme qui passe en revue tous les nœuds d'un arbre

Ancêtre (ou ascendant) (*ancestor*)

Nœud : N_a est un ancêtre de N_d s'il existe un chemin allant de N_a vers N_d

Descendant (*descendant*)

Nœud : N_d est un descendant de N_a si N_a est un ancêtre de N_d

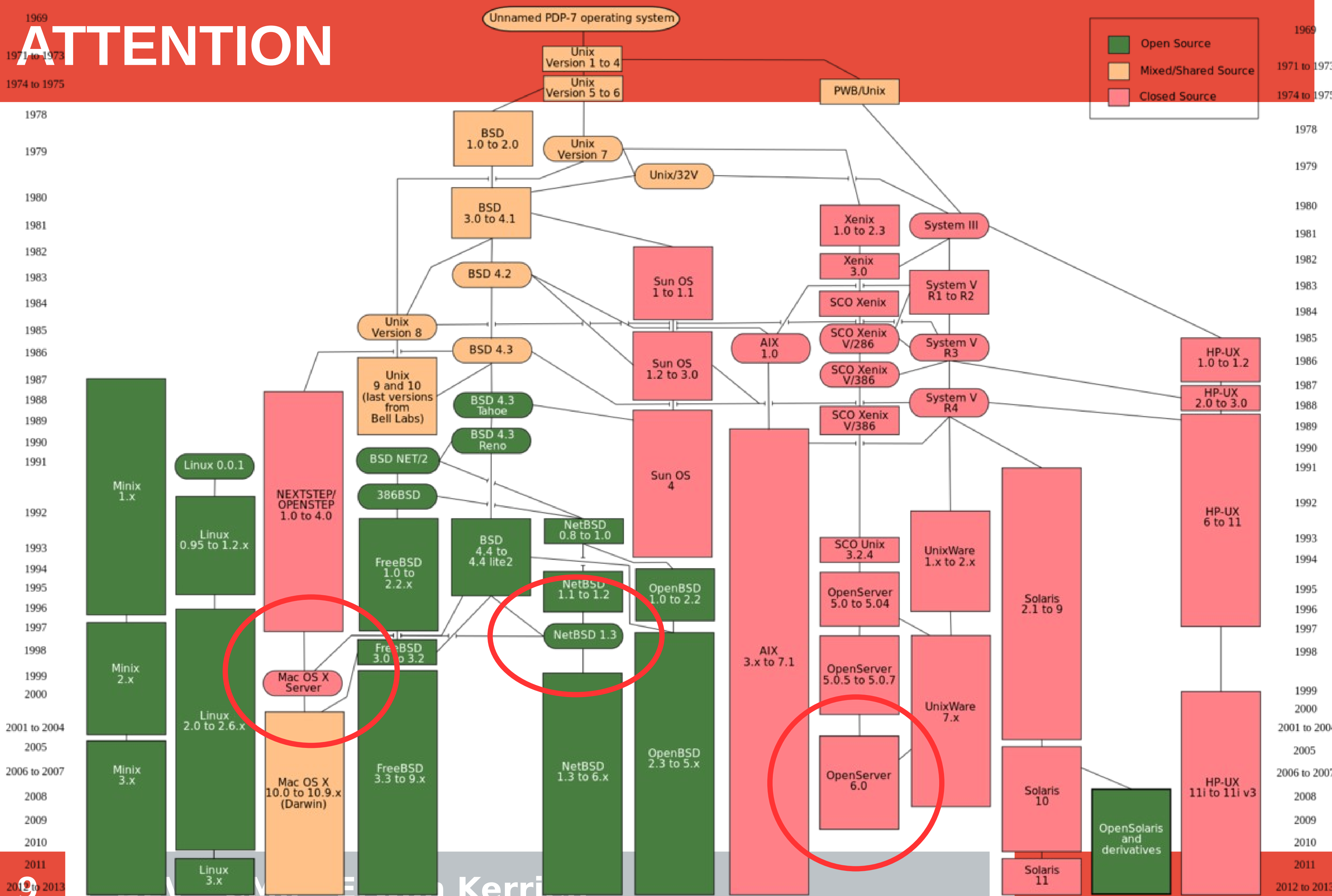
Arbre (*tree*)

A une racine et hormis la racine, tout nœud a un et un seul parent

Sous-arbre (*subtree*)

Arbre formé par tous les descendants d'un nœud

ATTENTION



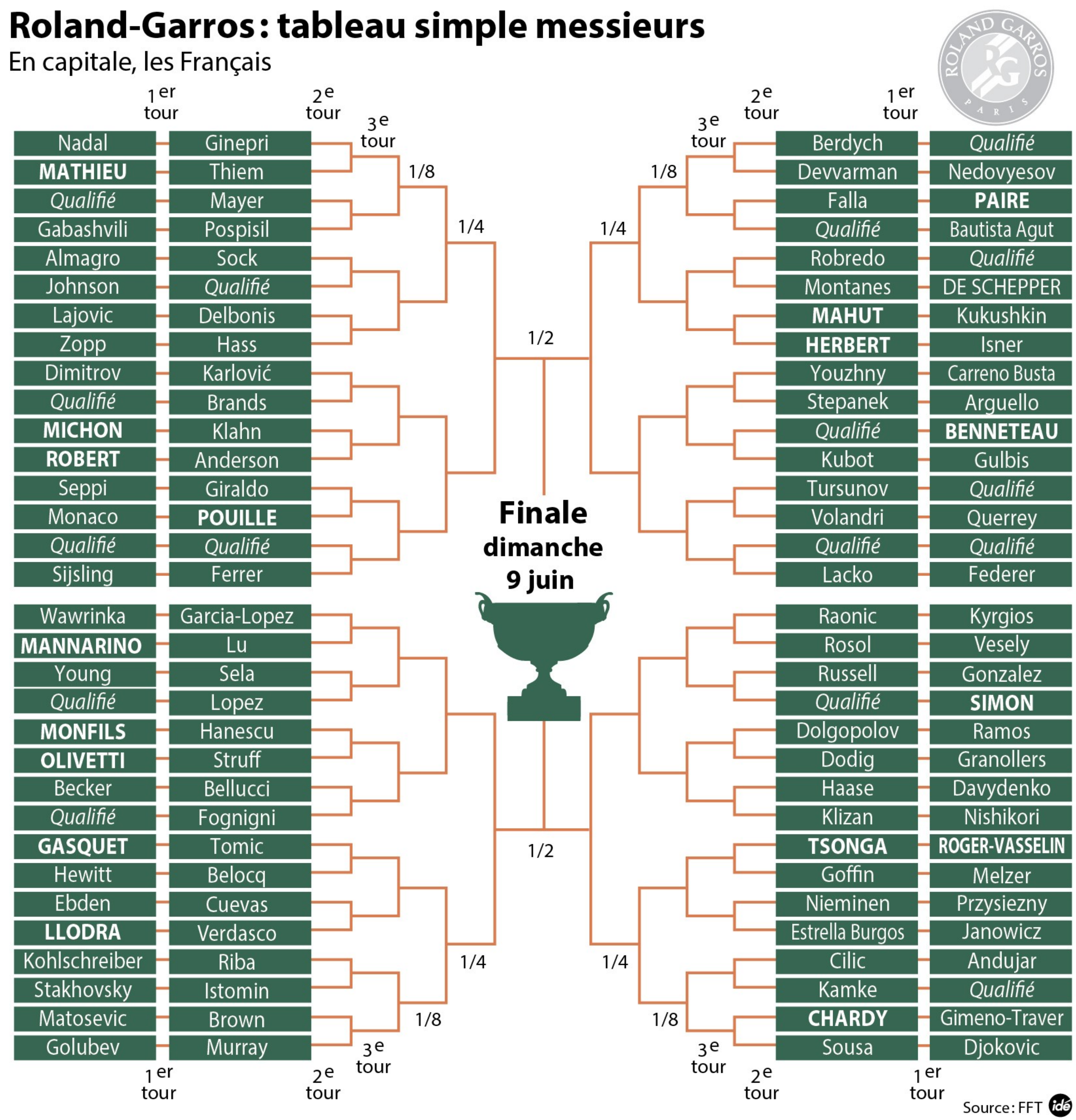
Ceci n’est pas un arbre !!!

ARBRE BINAIRE

B-Tree

Chaque nœud a au plus deux enfants

- Chaque enfant est soit vide, soit un arbre binaire
- Tous les sous-arbres sont disjoints
- Ex : tournoi

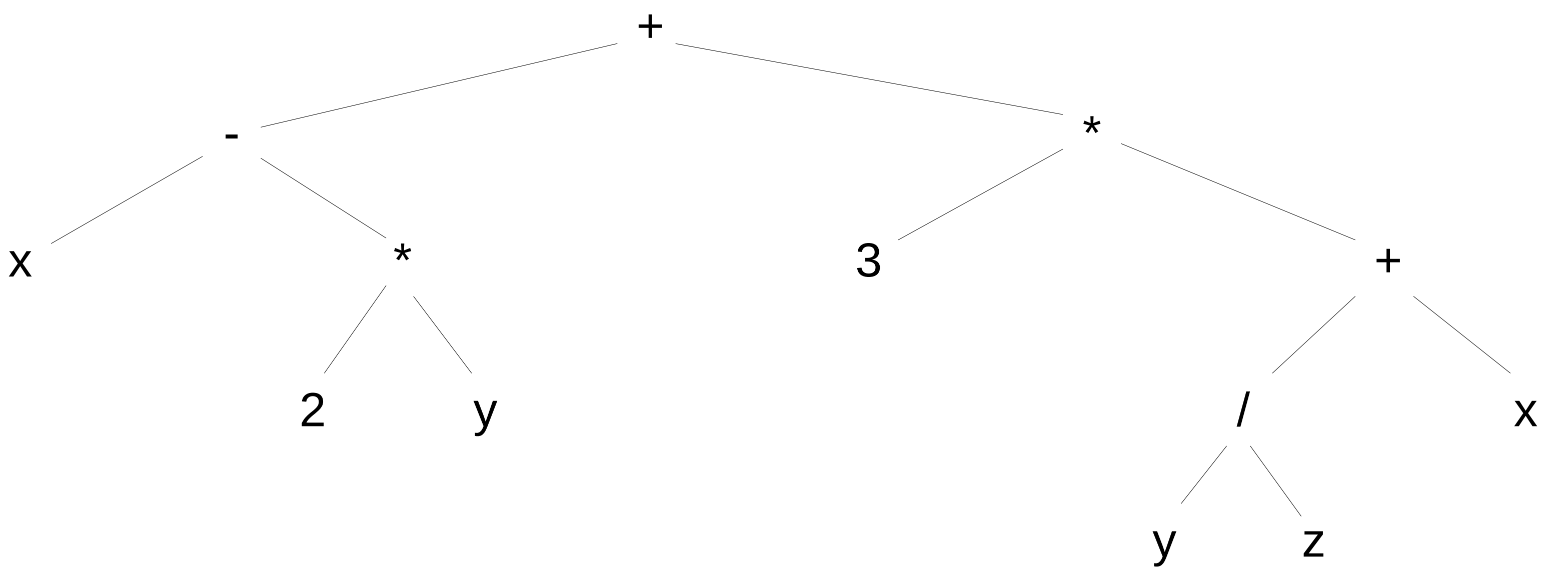


Tout arbre peut être représenté par un arbre binaire

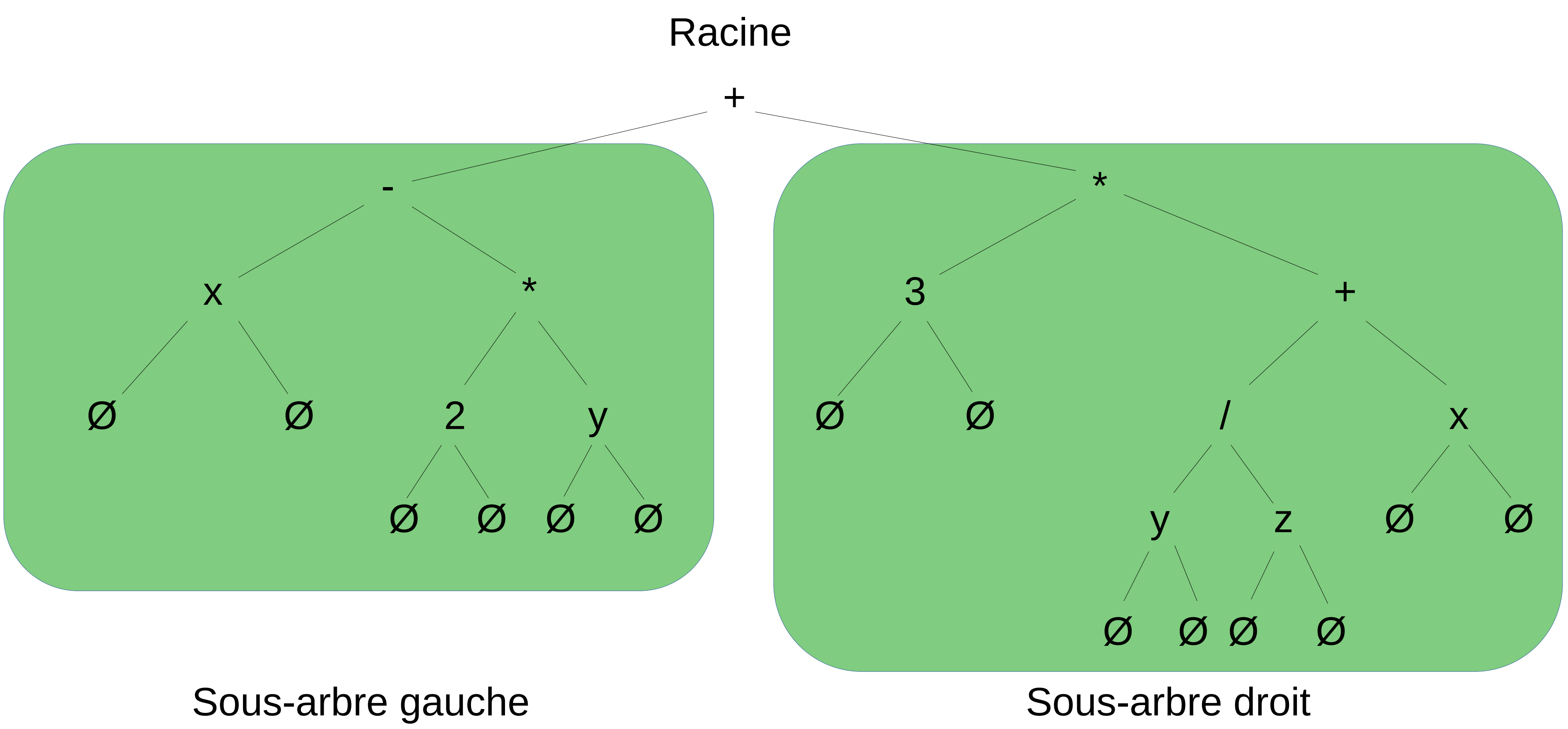
- Fils aîné – Frère droit
- Non vu dans ce cours

Arbre d’expression (*expression tree*)

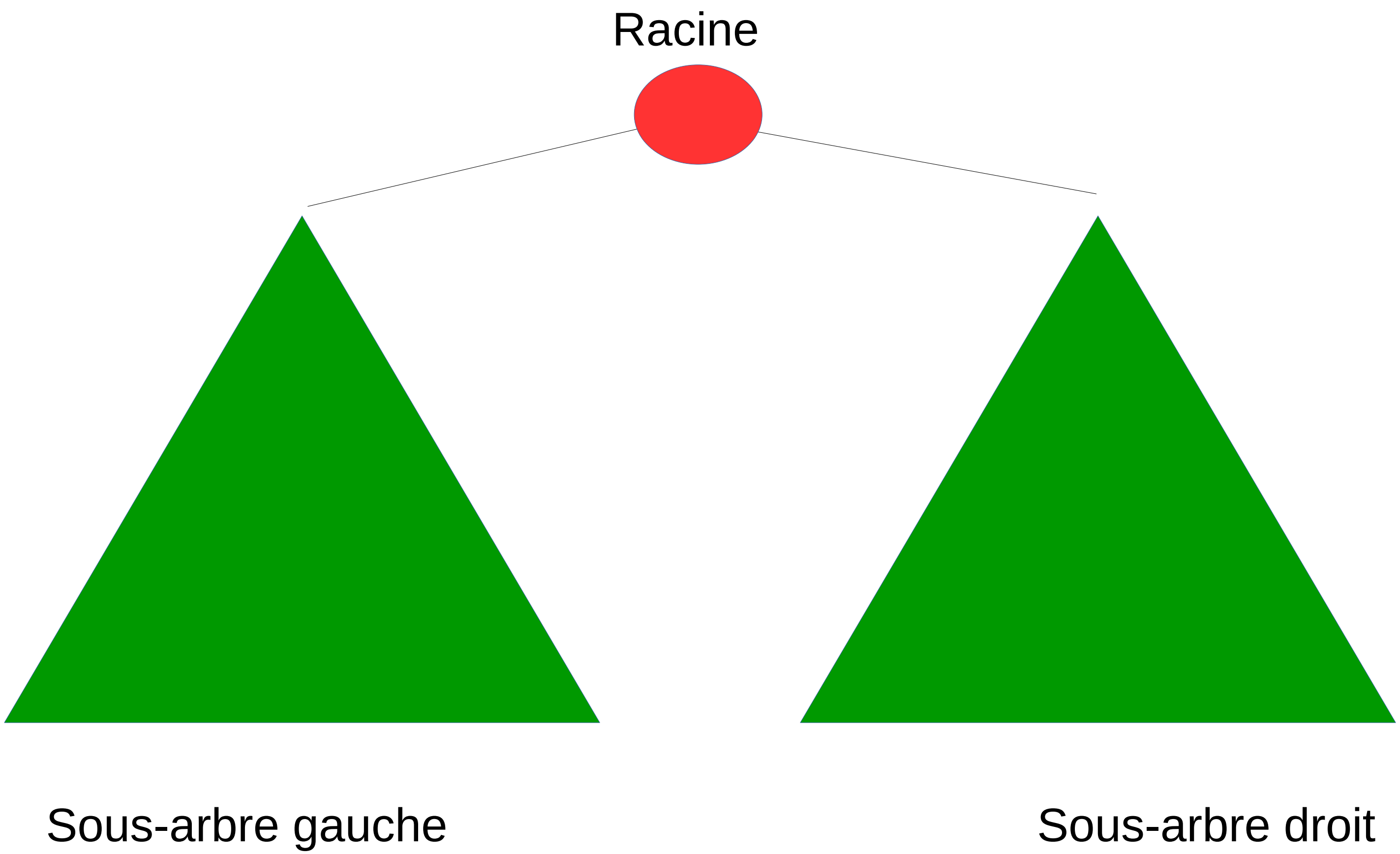
(x-2*y) + 3*(y/z+x)



DÉFINITION RÉCURSIVE



DÉFINITION RÉCURSIVE



TYPE ABSTRAIT DE DONNÉES

Signature

Sorte :

ABin, Nœud

Utilise :

Élément (élément_vide), Booléen

Opérations :

arbre_binaire_vide (Ø) : → Abin

estVide : ABin → Booléen

créer : Élément x ABin x ABin → ABin

Racine : ABin → Nœud

Contenu : Nœud → Élément

SAG : Nœud → ABin

SAD : Nœud → ABin

estDans : Nœud x ABin → Booléen

Propriétés/axiomes

Racine(A) défini ssi non estVide(A)

Contenu(Racine(Créer(E,G,D))) = E

SAG(Racine(Créer(E,G,D))) = G

SAD(Racine(Créer(E,G,D))) = D

Créer(E,G,D) est valide ssi G et D sont disjoints :

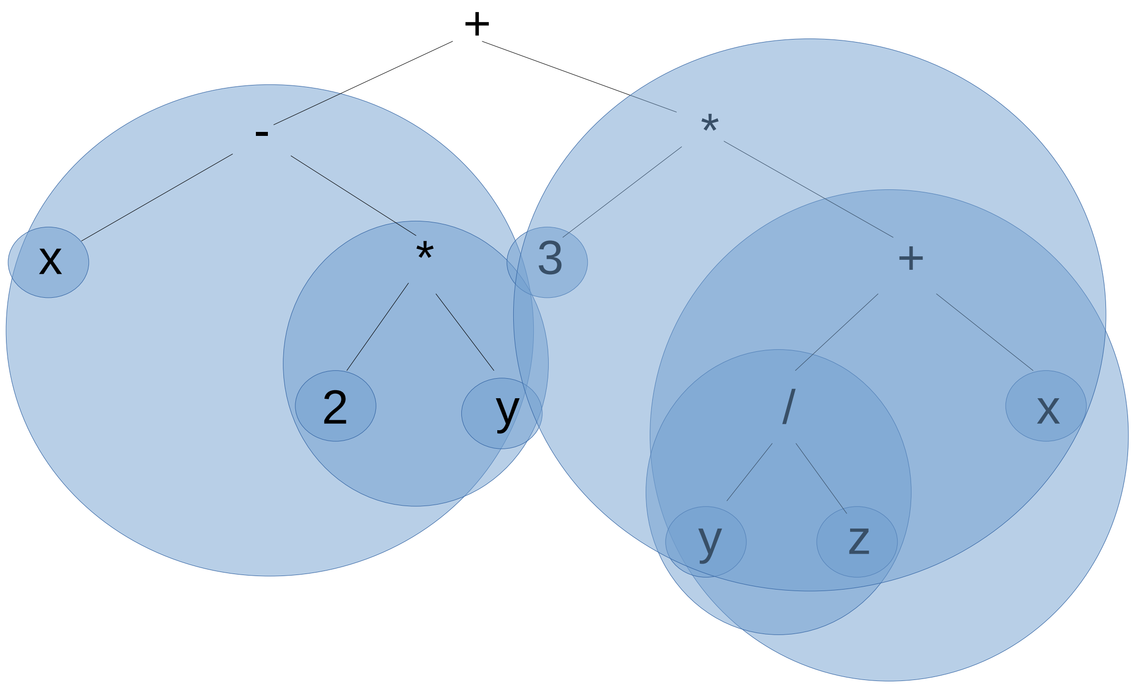
estDans(N,G) implique non estDans(N,D)

EXEMPLE DE CRÉATION

```
G1=Créer(2,Ø,Ø)
G2=Créer(y,Ø,Ø)
G3=Créer(*,G1,G2)
G4=Créer(x,Ø,Ø)
G5=Créer(-,G4,G3)

D1=Créer(y,Ø,Ø)
D2=Créer(z,Ø,Ø)
D3=Créer(/,D1,D2)
D4=Créer(x,Ø,Ø)
D5=Créer(+,D3,D4)
D6=Créer(3,Ø,Ø)
D7=Créer(*,D6,D5)

A=Créer(+,G5,D7)
```



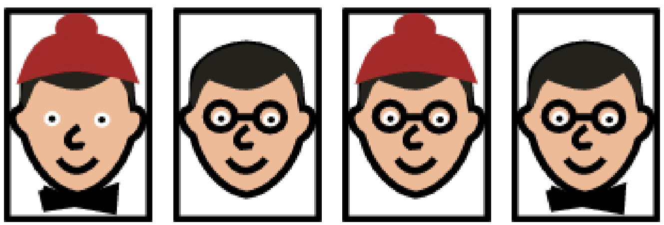
ARBRES VALUÉS ET NON VALUÉS

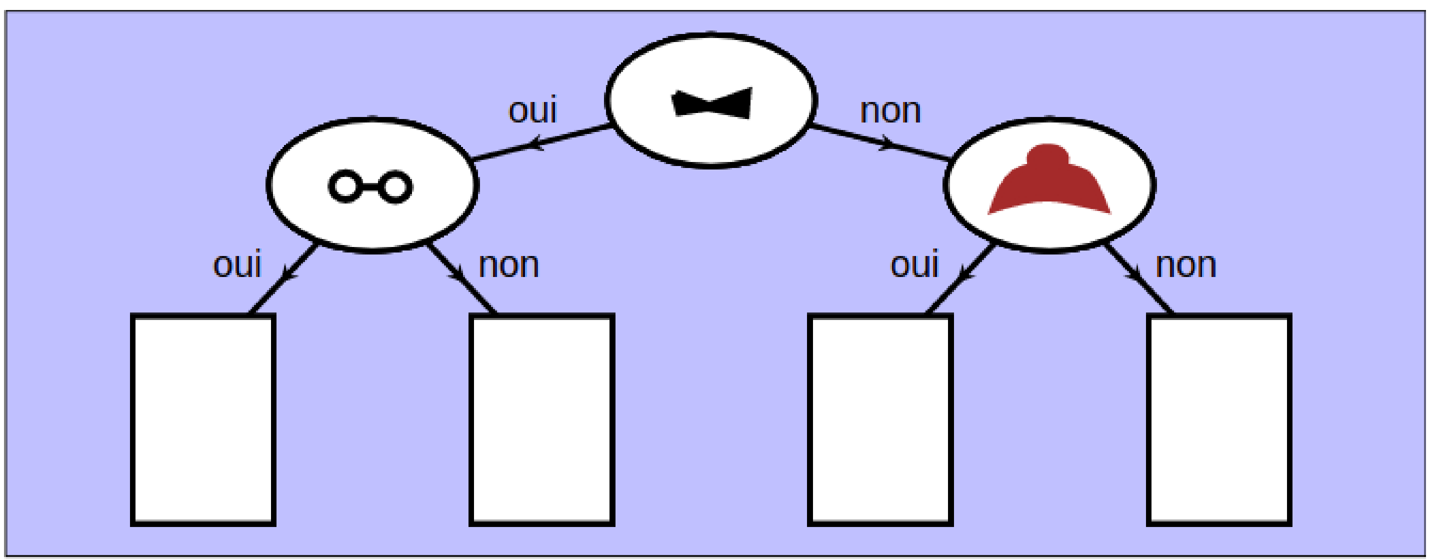
Arbre valué

Un arbre A est valué si Contenu(N)≠élément_vide pour tout N tel que estDans(N,A)=Vrai

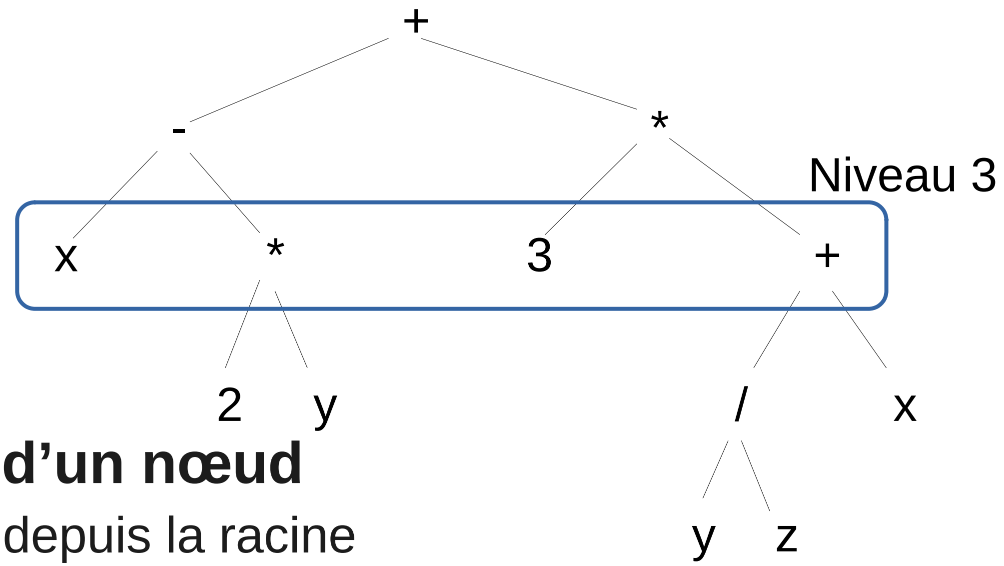
En général, un arbre non valué est tel que seules ses feuilles contiennent une valeur. Les autres nœuds sont des nœuds d'orientation

Exemple d'arbre non valué : Qui est-ce ?





QUELQUES MESURES SUR LES ARBRES



Taille (ex : 13)

- Nombre de nœuds
- Définition récursive :
  - Taille(Ø) = 0
  - Taille(Créer(E,G,D)) = 1 + Taille(G) + Taille(D)

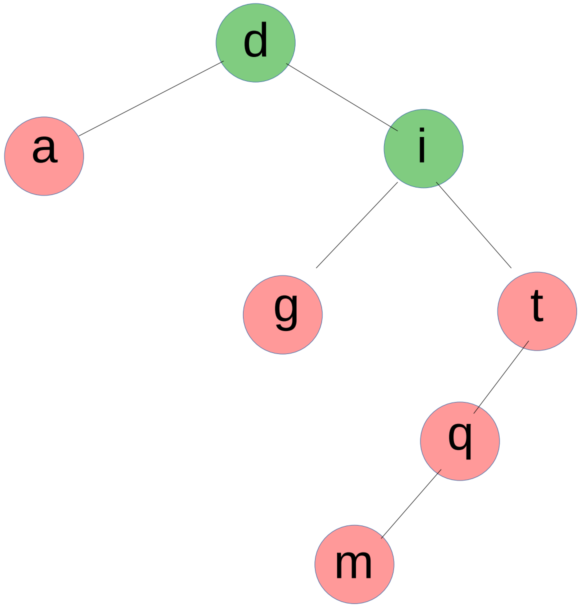
Niveau (ou hauteur, ou profondeur) d'un nœud

- Longueur du chemin (nombre de nœuds) depuis la racine
- Définition récursive
  - L(Racine(A)) = 1
  - $L(N_e) = 1+L(N_p)$ si N_p est parent de N_e

Hauteur (ou profondeur) d'un arbre (Ex : 5)

- Longueur du chemin maximal
- Deux définitions possibles (et équivalentes)
  - H(Ø) = 0 ; H(Créer(E,G,D)) = 1+max{H(G),H(D)}
  - H(A) = max{L(N) ; avec N nœud de A, A≠Ø}

NOEUDS EXTERNES ET INTERNES



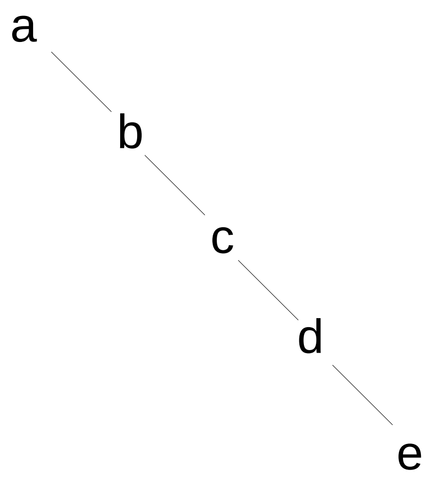
N est un nœud externe ssi SAG(N)=Ø ou SAD(N)=Ø

N est un nœud interne ssi SAG(N)≠Ø et SAD(N)≠Ø

(autrement dit ssi N n’est pas externe)

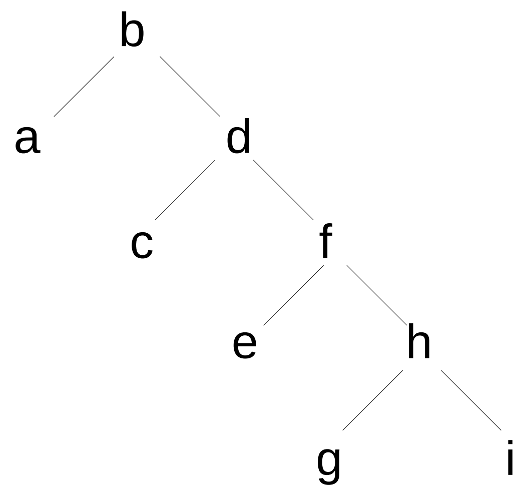
ARBRES BINAIRES PARTICULIERS

Arbre filiforme : tous ses nœuds sont externes



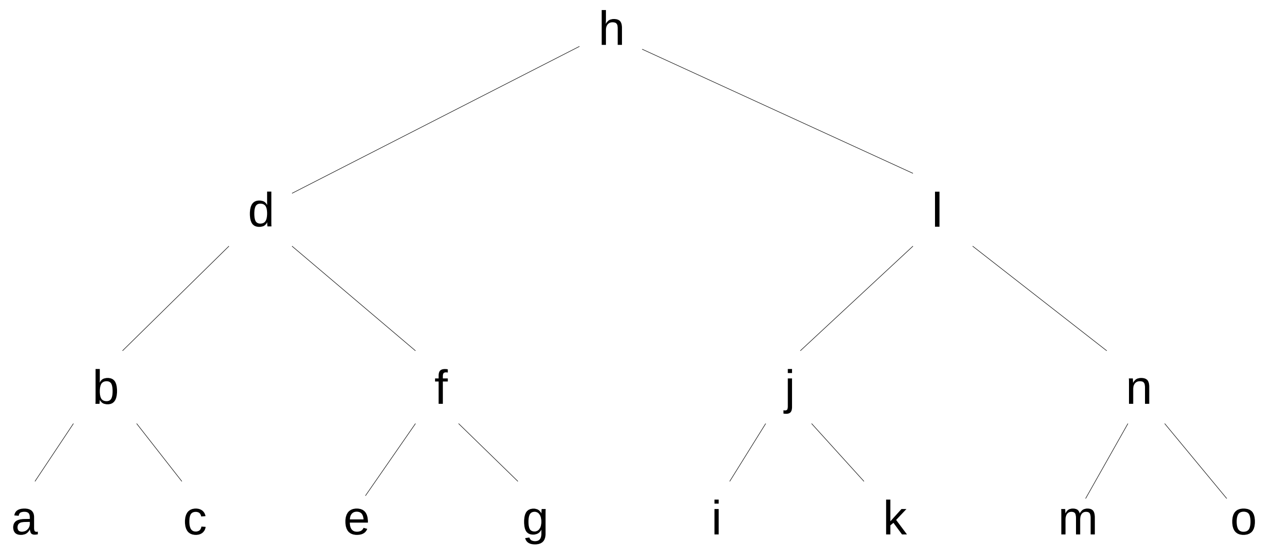
Liste chaînée...

Arbre peigne (droit) : tous les fils gauches sont des feuilles



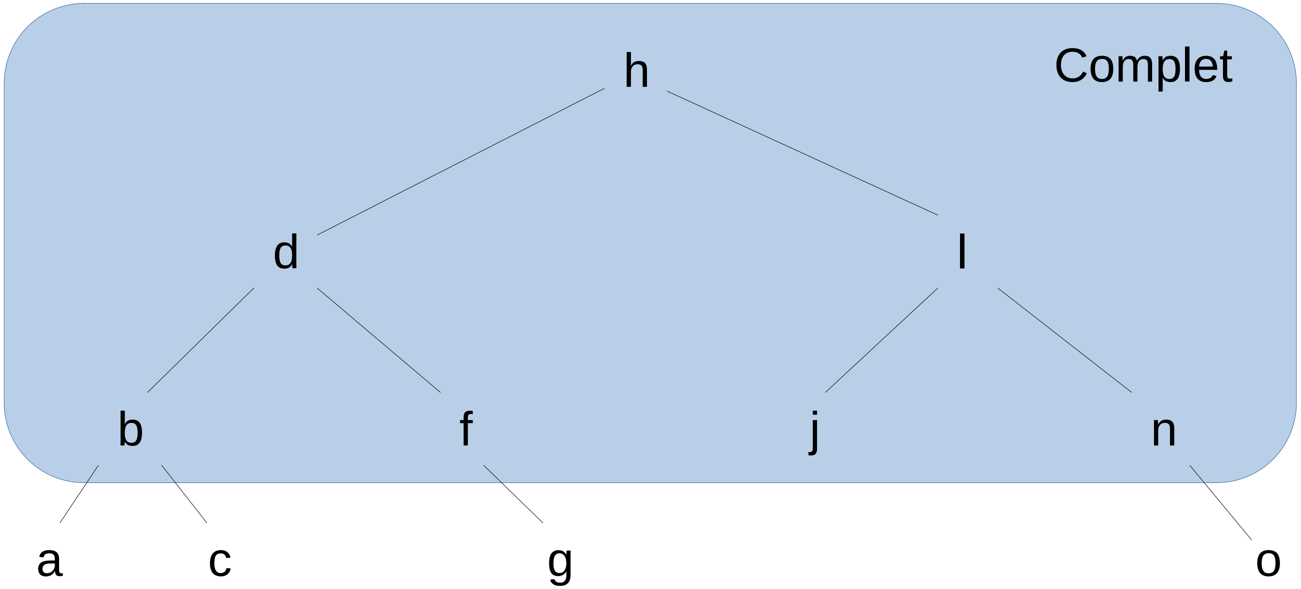
ARBRES BINAIRES PARTICULIERS

Arbre complet : tous ses nœuds externes ont même profondeur



ARBRES BINAIRES PARTICULIERS

Arbre parfait : tous les niveaux sont remplis, sauf éventuellement le dernier



PARCOURS D'UN ARBRE

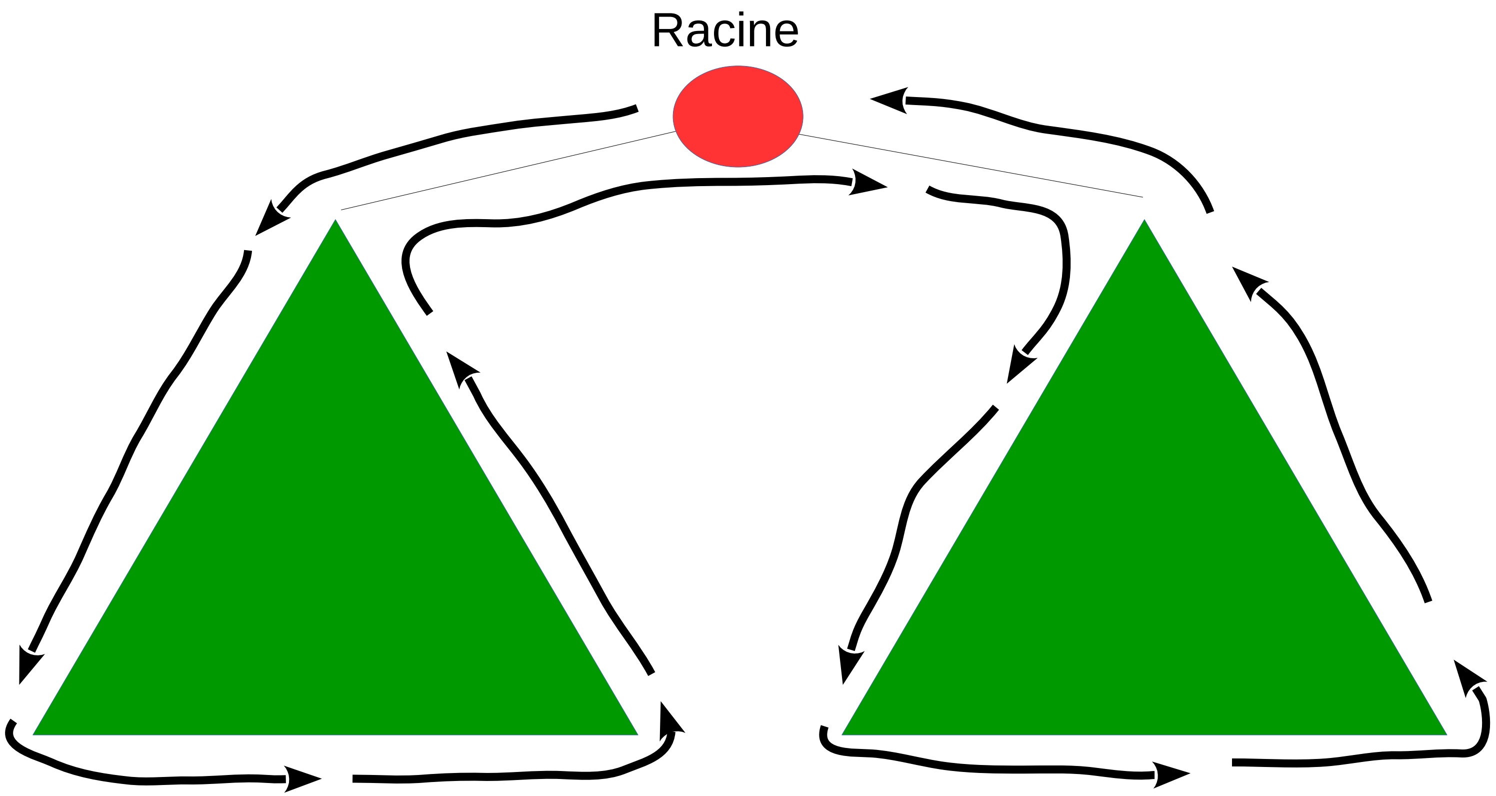
Parcours : visite de tous les nœuds

- En pratique, on applique un traitement à chaque nœud visité
- Exemple : TRAITEMENT(N) = Afficher(Contenu(N))

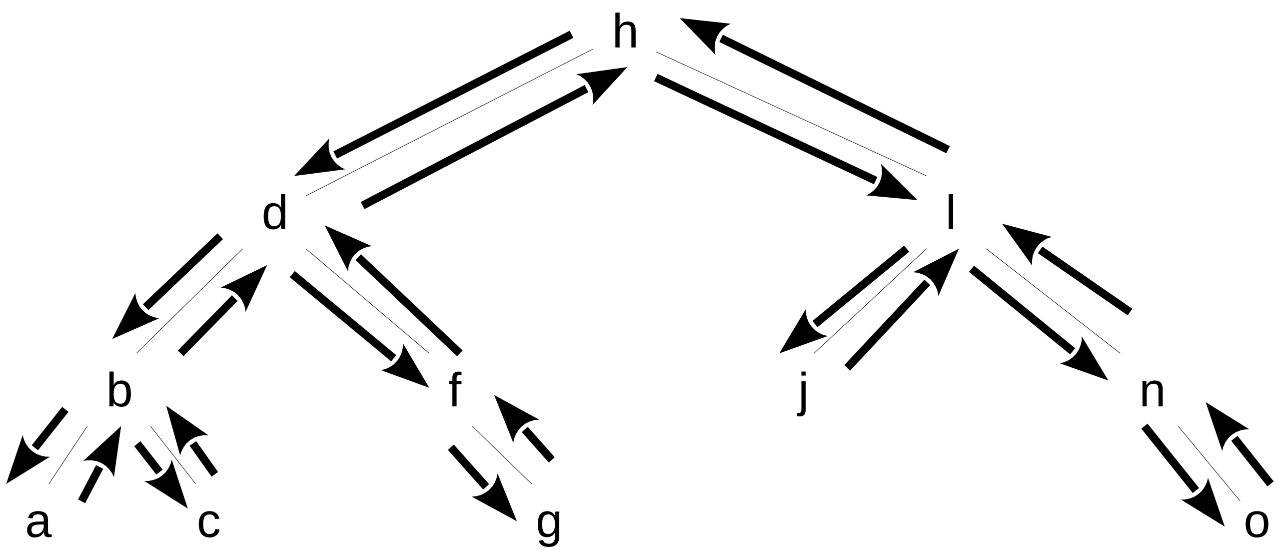
Deux types de parcours

- Parcours en **profondeur d'abord** à main gauche (resp. droite):
  - on visite le sous-arbre gauche (resp. droit) intégralement avant de visiter le sous-arbre droit (resp. gauche)
  - → Trois ordres de traitement : **préfixe, postfixe, infixe**
- Parcours en **largeur d'abord** (main gauche ou droite)
  - On visite tous les nœuds d'un niveau avant de passer au niveau suivant (de gauche à droite ou de droite à gauche)

PARCOURS EN PROFONDEUR D'ABORD (MAIN GAUCHE)



PARCOURS EN PROFONDEUR D'ABORD (MAIN GAUCHE)

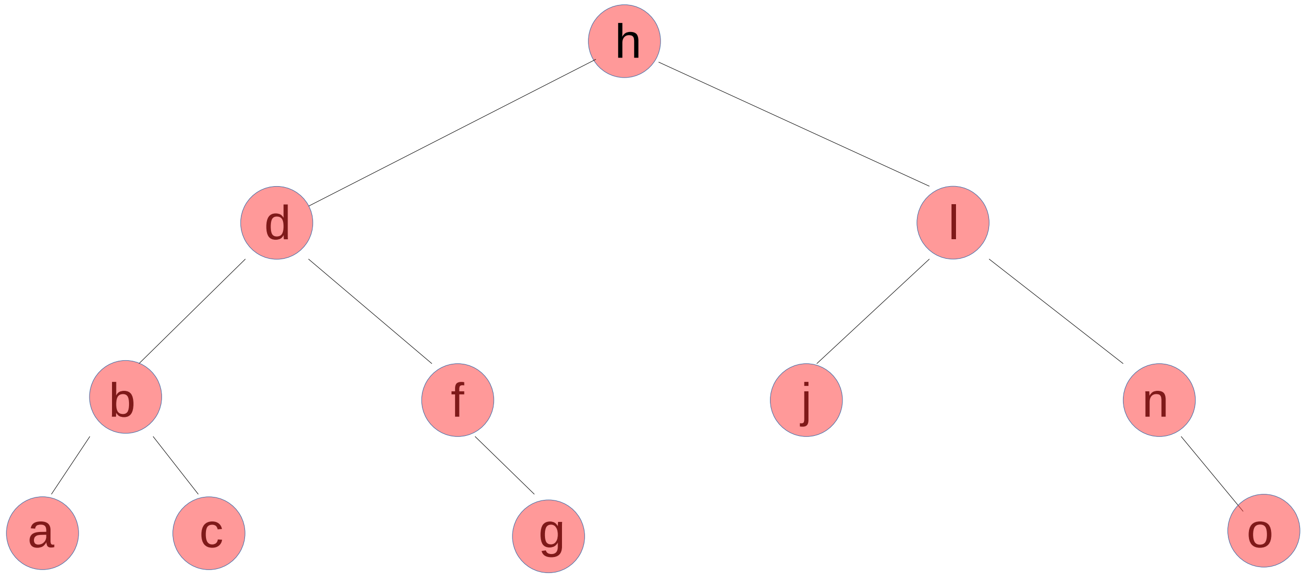


Algorithme récursif :

```
Procédure ParcoursProf(A:ABin)
    Début
        Si non estVide(A) alors
            ParcoursProf(SAG(Racine(A)))
            ParcoursProf(SAD(Racine(A)))
        FinSi
    Fin
```

Remarque :
où placer le traitement des nœuds ?

PARCOURS EN PROFONDEUR D'ABORD (MAIN GAUCHE) : ORDRE PRÉFIXE



Nœuds traités

h d b a c f g l j n o

Pile d'appel

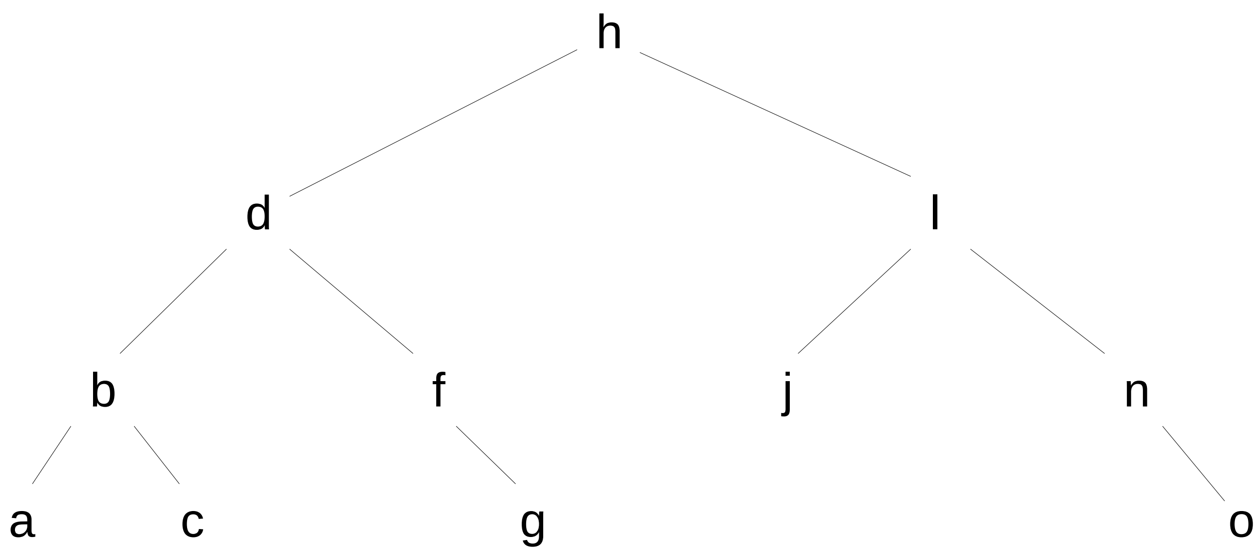
```
ParcoursProfPref(Ø)
ParcoursProfPref(g)
ParcoursProfPref(f)
ParcoursProfPref(d)
ParcoursProfPref(h)
```

Note : **<u>k</u>** *est l'arbre ayant le nœud contenant* k *pour racine*

Algorithme : Ordre préfixe

```
Procédure ParcoursProfPref(A:Abin)
    Début
        Si non estVide(A) alors
            TRAITEMENT(Racine(A))
            ParcoursProfPref(SAG(Racine(A)))
            ParcoursProfPref(SAD(Racine(A)))
        FinSi
    Fin
```

PARCOURS EN PROFONDEUR D'ABORD (MAIN GAUCHE) : ORDRE POSTFIXE



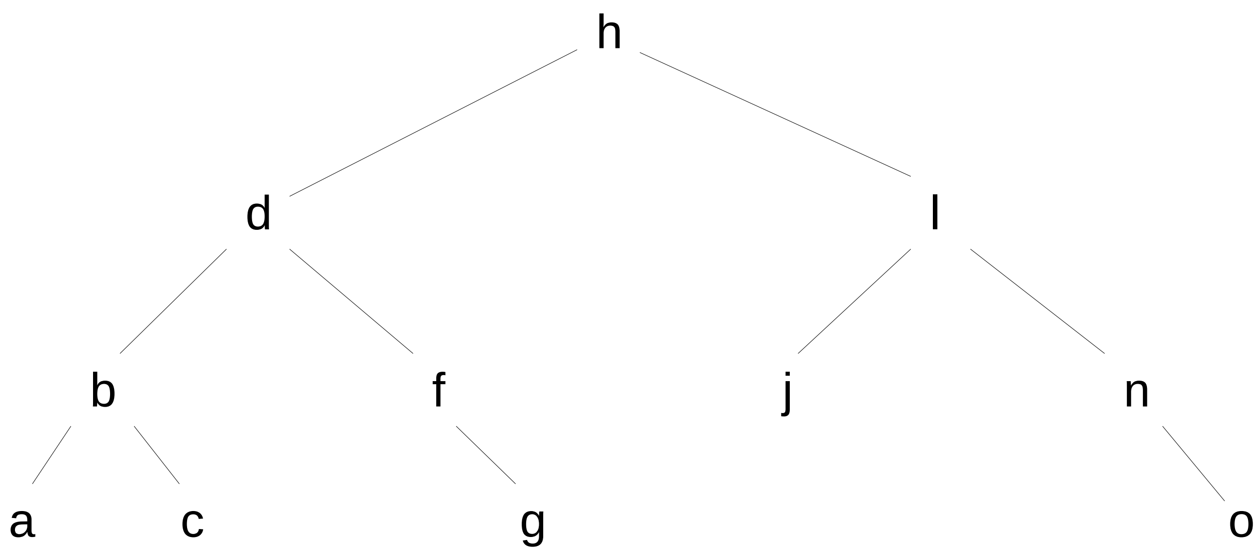
Nœuds traités

a c b g f d j o n l h

Algorithme : Ordre postfixe

```
Procédure ParcoursProfPost(A:ABin)
    Début
        Si non estVide(A) alors
            ParcoursProfPost(SAG(Racine(A)))
            ParcoursProfPost(SAD(Racine(A)))
            TRAITEMENT(Racine(A))
        FinSi
    Fin
```

PARCOURS EN PROFONDEUR D’ABORD (MAIN GAUCHE) : ORDRE INFIXE



Nœuds traités

a b c d f g h j l n o

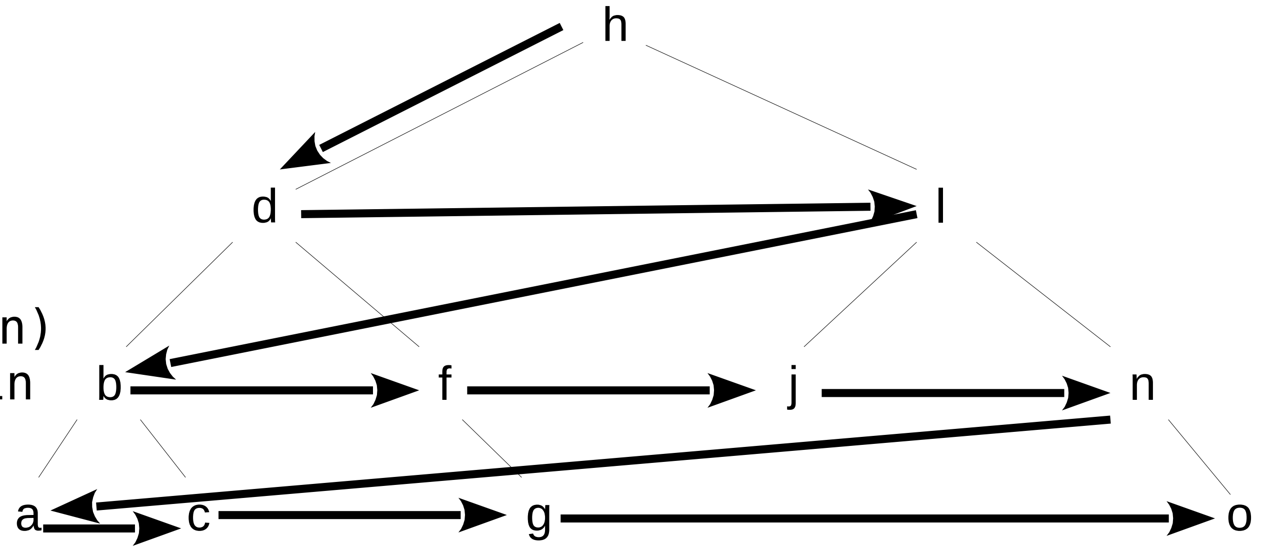
Ordre lexicographique !

Algorithme : Ordre infixe

```
Procédure ParcoursProfInf(A:ABin)
    Début
        Si non estVide(A) alors
            ParcoursProfInf(SAG(Racine(A)))
            TRAITEMENT(Racine(A))
            ParcoursProfInf(SAD(Racine(A)))
        FinSi
    Fin
```

PARCOURS EN LARGEUR D'ABORD

Remplacer la **pile** d'appel par une **file** de nœuds



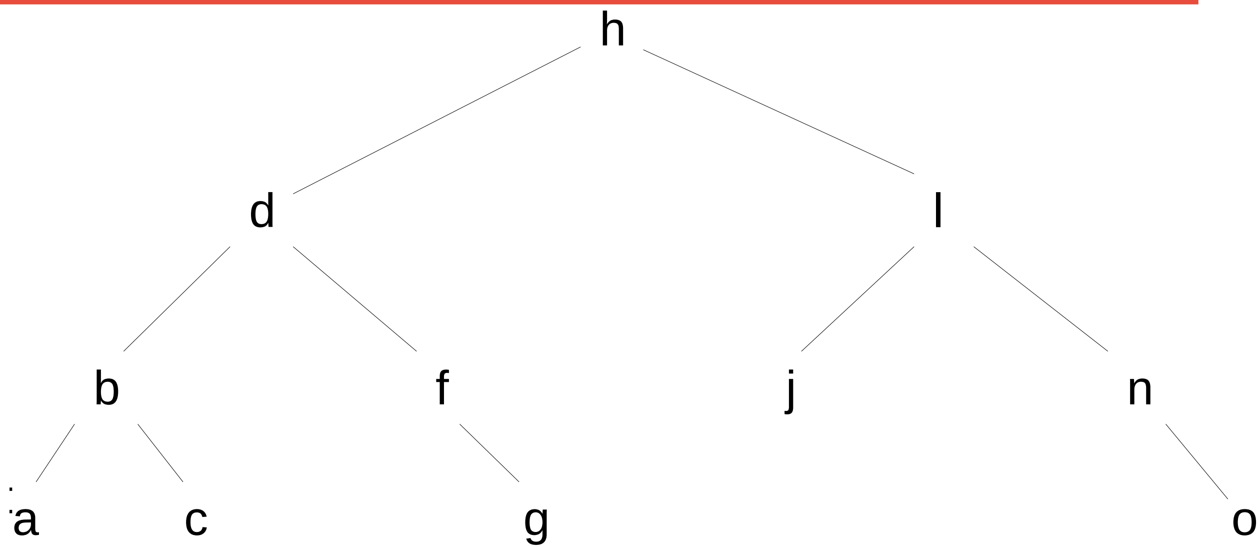
Algorithme :

```
Procédure ParcoursLarg(A:ABin)
    Variable : F:File, tA:ABin
    Début
        F ← file_vide
        F ← Ajoute(F,A)
        Tant que non estVide(F) faire
            tA ← Premier(F)
            F ← Retirer(F)
            Si non estVide(tA) alors
                TRAITEMENT(Racine(tA))
                F ← Ajoute(F,SAG(Racine(tA)))
                F ← Ajoute(F,SAD(Racine(tA)))
            Fin Si
        Fin Tant que
    Fin
```

Nœuds traités

h d l b f j n a c g o

ARBRES BINAIRES DE RECHERCHE (ABR)



Ordre local à chaque nœud :

- Contenu(Racine(SAG(N)) < Contenu(N) < Contenu(Racine(SAD(N))

Parcours infixe

- Ordre lexicographique

Longueur chemin max = hauteur (h = 4)

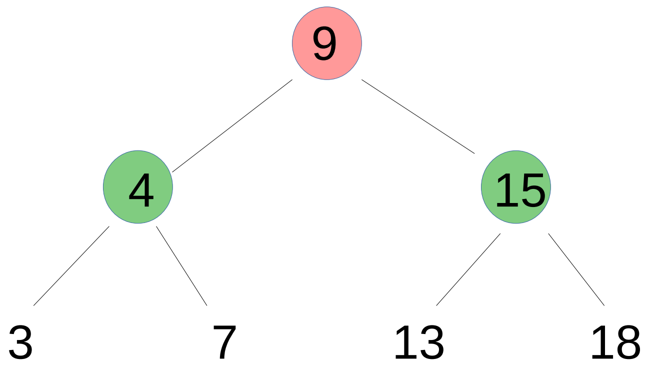
Nombre de nœuds maximaux (arbre complet) : 15 = $2^4 - 1 \rightarrow 2^h-1$

Recherche : pire cas = visite de h nœuds

- → complexité en O(log(taille(A))) si arbre complet (ou parfait)
- → complexité en O(taille(A)) si arbre filiforme

LIENS AVEC LA DICHOTOMIE

Tableau trié : [3 4 7 9 13 15 18]



Recherche en trois visites maximum (7 valeurs) : 3 ≈ log(7)

PROPRIÉTÉS D'UN ABR

Si ABR non vide

- Arbre valué
- Les SAG et SAD sont des ABR
- Les valeurs stockées dans le SAG sont strictement inférieures au contenu de la racine
- Les valeurs stockées dans le SAD sont strictement supérieures au contenu de la racine

Théorème

Un arbre binaire valué est un ABR ssi son parcours infixe produit des nœuds dont les valeurs sont strictement croissantes

OPÉRATIONS SUR LES ABR : RECHERCHE

Recherche d'un élément

estDans : Elément x ABR → Booléen

```
Fonction estDans(E:Elément, A:ABR):Booléen
    Début
        Si estVide(A) alors
            Renvoyer Faux
        Sinon Si E < Contenu(Racine(A)) alors
            Renvoyer(estDans(E,SAG(Racine(A)))
        Sinon Si E > Contenu(Racine(A)) alors
            Renvoyer(estDans(E,SAD(Racine(A)))
        Fin Si
        Renvoyer Vrai
    Fin
```

Complexité : pire cas, on atteint la feuille la plus basse → hauteur de l'arbre

O(taille) si ABR filiforme, O(log(taille)) si parfait

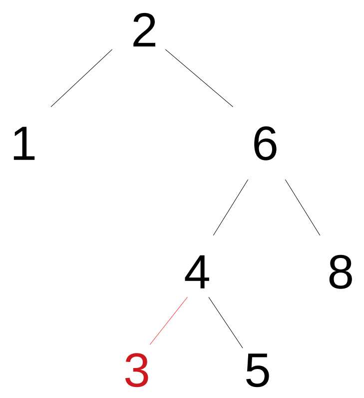
OPÉRATIONS SUR LES ABR : INSERTION

Insertion d'un élément : insertion dans une feuille

ajoute : Élément x ABR → ABR

```
Fonction ajoute(E:Elément, A:ABR):ABR
    Début
        Si estVide(A) alors
            A ← Créer(E,Ø,Ø)
        Sinon Si E < Contenu(Racine(A)) alors
            SAG(Racine(A)) ← ajoute(E,SAG(Racine(A))
        Sinon Si E > Contenu(Racine(A)) alors
            SAD(Racine(A)) ← ajoute(E,SAD(Racine(A))
        Fin Si
        Renvoyer A
    Fin
```

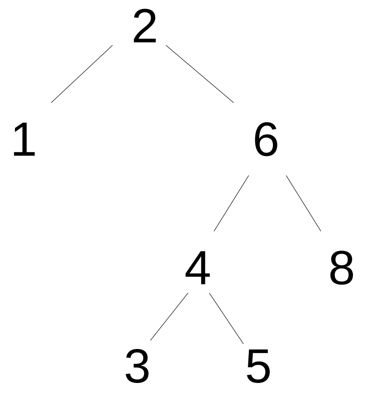
Ex : ajoute(3,A)



Complexité : pire cas, on ajoute comme enfant de la feuille la plus basse → hauteur de l'arbre

O(taille) si ABR filiforme, O(log(taille)) si parfait

OPÉRATIONS SUR LES ABR : SUPPRESSION

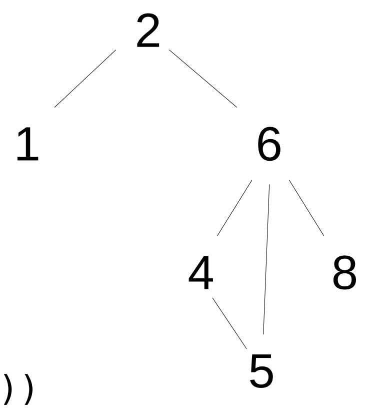


Suppression d'un élément : 3 cas
1) L'élément est dans une feuille (ex : 3)
→ on détruit cette feuille

```
Fonction supprimeFeuille(E:Élément, A:ABR):ABR
    Début
        Si non estVide(A) alors
            Si E < Contenu(Racine(A)) alors
                SAG (Racine(A)) ← supprimeFeuille(E,SAG(Racine(A))
            Sinon Si E > Contenu(Racine(A)) alors
                SAD(Racine(A)) ← supprimeFeuille(E,SAD(Racine(A))
            Sinon
                Désallouer(Racine(A))
                A ← Ø
            Fin Si
        Fin Si
        { Sinon, estVide(A) implique que l'élément n'est pas dans A
          → On ne fait rien }
        Renvoyer A
    Fin
```

OPÉRATIONS SUR LES ABR : SUPPRESSION

Suppression d'un élément : 3 cas
2) L'élément a un seul enfant (nœud externe, par ex : 4)
→ on remplace le nœud par son enfant




```
Fonction supprimeExterne(E:Élément, A:ABR):ABR
    Variable : tA:ABR
    Début
        Si non estVide(A) alors
            Si E < Contenu(Racine(A)) alors
                SAG  (Racine(A)) ← supprimeExterne(E,SAG(Racine(A))
            Sinon Si E > Contenu(Racine(A)) alors
                SAD(Racine(A)) ← supprimeExterne(E,SAD(Racine(A))
            Sinon
                tA ← Ø
                Si non estVide(SAG(Racine(A))) alors
                    tA ← SAG(Racine(A))
                Sinon
                    tA ← SAD(Racine(A))
                Fin Si
                Désallouer(Racine(A))
                A ← tA
            Fin Si
        Fin Si
        { Sinon, estVide(A) implique que l'élément n'est pas dans A
          → On ne fait rien }
        Renvoyer A
    Fin
```

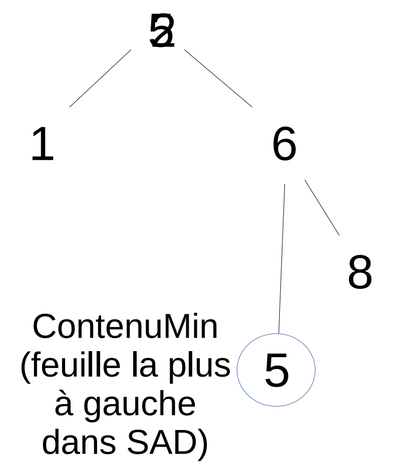
{ remplace
Désallouer(Racine(A)
A ←Ø
}

OPÉRATIONS SUR LES ABR : SUPPRESSION

Suppression d'un élément : 3 cas

3) L'élément a deux enfants (*nœud plein*, ex : 2)

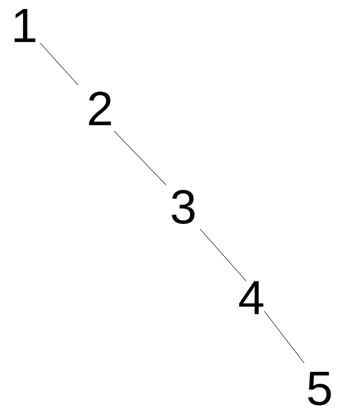
→ on remplace le nœud par le plus petit élément de SAD




```
Fonction supprime(E:Élément, A:ABR) :ABR
      Variable : tA:ABR, tE:Elément
      Début
            Si non estVide(A) alors
                  Si E < Contenu(Racine(A)) alors
                        SAG   (Racine(A)) ← supprime(E,SAG(Racine(A))
                  Sinon Si E > Contenu(Racine(A)) alors
                        SAD(Racine(A)) ← supprime(E,SAD(Racine(A))
                  Sinon
                        Si non estVide(SAG(Racine(A))) et non estVide(SAD(Racine(A))) alors
                              tE ← ContenuMin(SAD(Racine(A)))
                              Contenu(Racine(A)) ← tE
                              SAD(Racine(A)) ← supprime(tE,SAD(Racine(A)))
                        Sinon
                              tA ← Ø
                              Si non estVide(SAG(Racine(A))) alors
                                    tA ← SAG(Racine(A))
                              Sinon
                                    tA ← SAD(Racine(A))
                              Fin Si
                              Désallouer(Racine(A))
                              A ← tA
                        Fin Si
                  Fin Si
            Fin Si
            { Sinon, estVide(A) implique que l'élément n'est pas dans A
              → On ne fait rien }
            Renvoyer A
      Fin
```

ÉQUILIBRE D’UN ARBRE

Ex : création par insertions successives de [1,2,3,4,5]



Équivalent à une liste chaînée, donc recherche en O(taille) (taille=hauteur)
→ déséquilibre des SAG et SAD

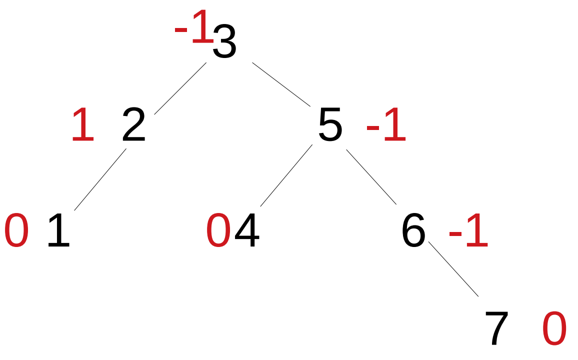
Définition : Équilibre d’un arbre A (H-équilibre)

0 si pour un arbre vide → E(Ø)=0

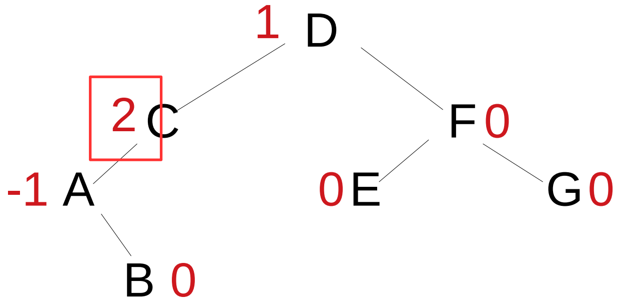
Différence des hauteurs des sous-arbres G et D → E(A)=H(SAG(Racine(A)))-H(SAD(Racine(A)))

Définition : Arbre équilibré

Un arbre binaire est équilibré si l’équilibre de tous ses sous-arbres est -1, 0, ou 1



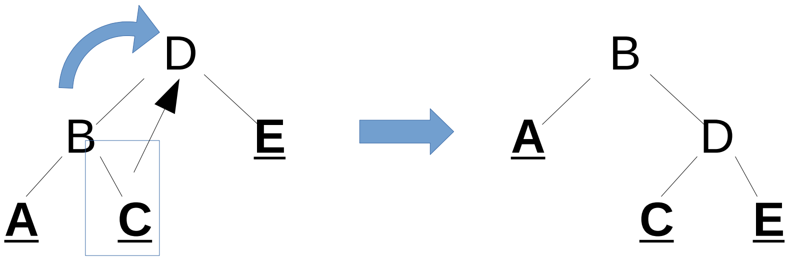
Équilibré



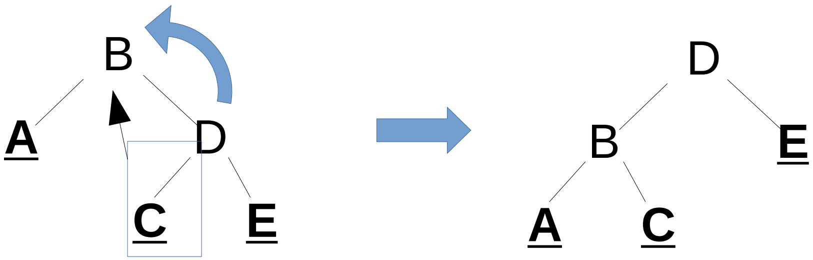
Déséquilibré

ROTATIONS

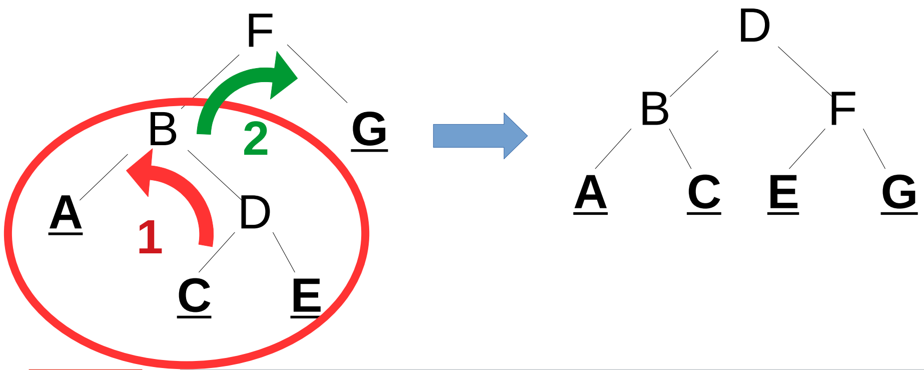
Droite



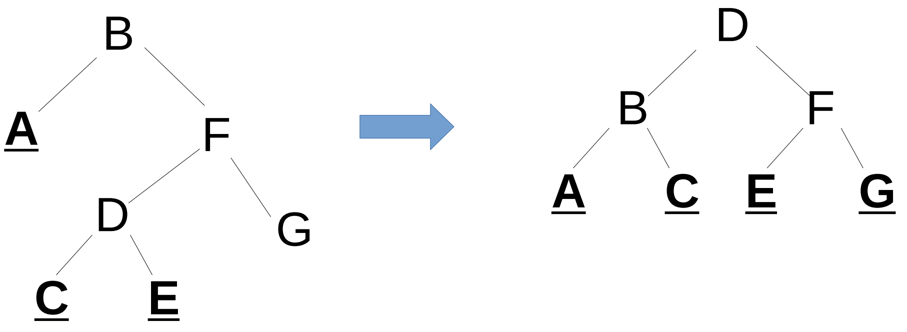
Gauche



Gauche-Droite



Droite-Gauche



ROTATIONS

Proposition

Après une insertion ou suppression 2 rotations suffisent au maximum pour ré-équilibrer un arbre.

Ces opérations se font en temps constant (O(1))

BILAN SUR LES ARBRES BINAIRES

Tout arbre peut se représenter par un arbre binaire (admis)

Arbre binaire de recherche

- Arbre valué
- ContenuMax(SAG) < Contenu < ContenuMin(SAD)
- Accès, insertion, suppression, recherche :**O(log(taille))**

TABLES DE HACHAGE

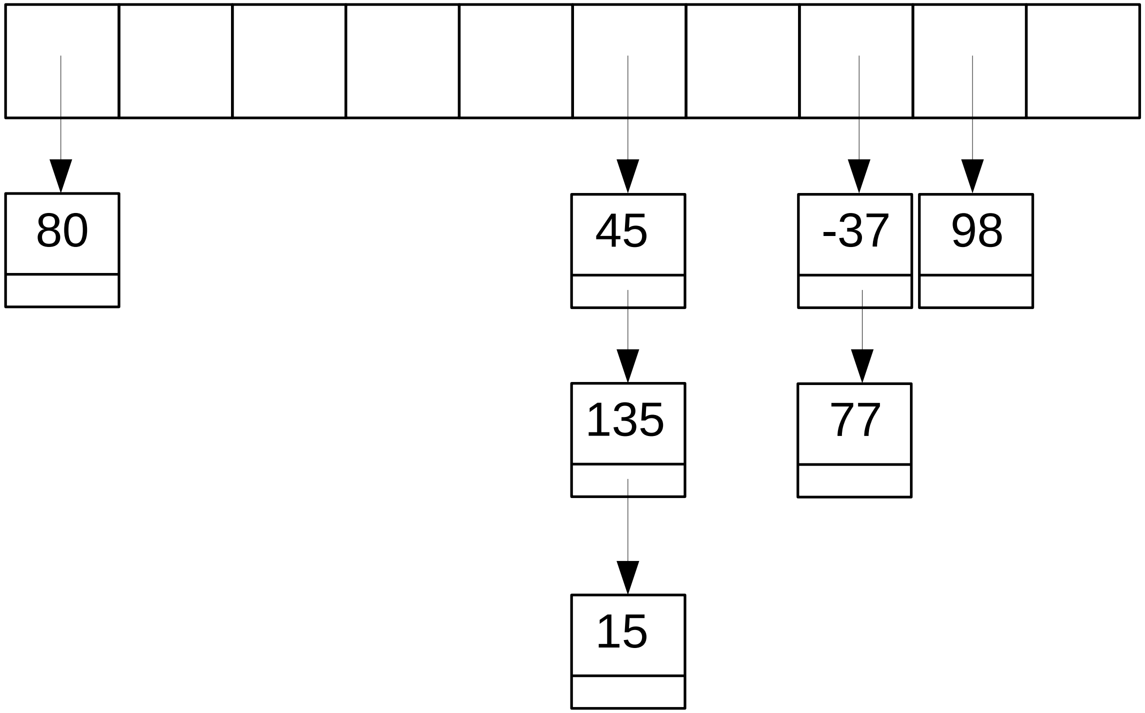
Structure de données : mélange tableau et liste chaînée (ou arbre si ordre)

- Fonction de hachage : h : Élément → Entier
  - Injective (unicité de l'entier retourné)
  - Non nécessairement bijective (plusieurs éléments peuvent correspondre au même entier)
- Principe
  - T : tableau de listes chaînées
  - E:Elément ; i ← h(E) ; ajouter E à la liste chaînée T[i]

EXEMPLE TRIVIAL

E:Entier ; h(E)=chiffre des unités

Ajouter 45,-37,135,80,15,77,98



HACHAGE ET TABLEAUX ASSOCIATIFS

Fonctions de hachage

- Autre fonction triviale : h(mot)=(somme des codes ASCII)%20
- Critère : répartition équilibrée des éléments
- Il existe des fonctions très efficaces
- Implémentées dans Python en tant que `set` ou `dict` (temps d'édition/recherche quasi constant)

Tableaux associatifs

- « indice »=mot (clé)
- Table de hachage : hachage de la clé, stockage du couple clé/valeur
- Implémentation aussi possible en arbre binaire de recherche (ordre sur clé)