



ARBRES

Arbres binaires

Représentation des arbres

Fonctions primitives sur les arbres

Parcours d'arbres

Arbres ordonnés

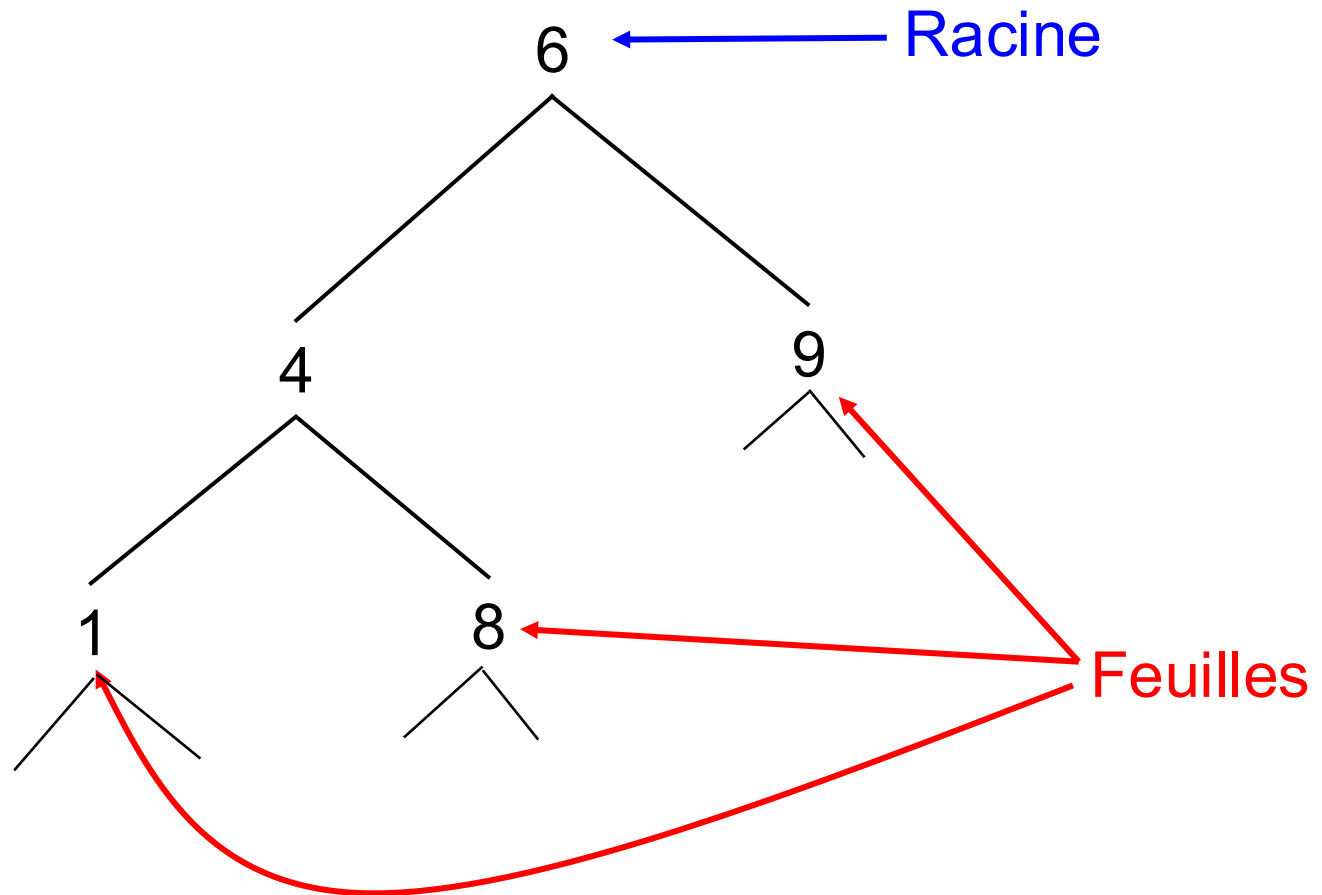
À QUOI SERVENT LES ARBRES ?

- Les arbres, comme les listes, permettent de représenter un nombre variable de données
- Le principal avantage des arbres par rapport aux listes est qu'ils permettent de ranger les données de telle sorte que les recherches soient plus efficaces

DÉFINITION

- Un arbre est soit un **nœud**, soit un **arbre vide**
- Un nœud a des **fils** qui sont eux aussi des arbres
- Si tous les fils d'un nœud sont vides, alors le nœud est qualifié de **feuille**
- Les nœuds portent des **valeurs**, ce sont les données que l'on veut stocker
- Si tous les nœuds de l'arbre ont n fils, alors l'arbre est dit **n -aire**

EXEMPLE D'ARBRE



ARBRES BINAIRES

- Un arbre binaire est :
 - soit l'arbre vide
 - soit un nœud qui a exactement deux fils (éventuellement vides)
- Pour manipuler les arbres binaires, on a besoin de primitives
 - d'accès
 - de test
 - de construction

PRIMITIVES SUR LES ARBRES BINAIRES (1)

○ Primitives d'accès

- **valeur** : retourne la valeur d'un arbre non vide
- **fils-g** : retourne le fils de gauche d'un arbre non vide
- **fils-d** : retourne le fils de droite d'un arbre non vide

○ Primitives de test

- **arbre?** : retourne vrai si un élément donné est un arbre
- **arbre-vide?** : retourne vrai si un arbre donné est vide
- **feuille?** : retourne vrai si un arbre donné est une feuille
- **arbre=?** : retourne vrai si deux arbres donnés sont égaux

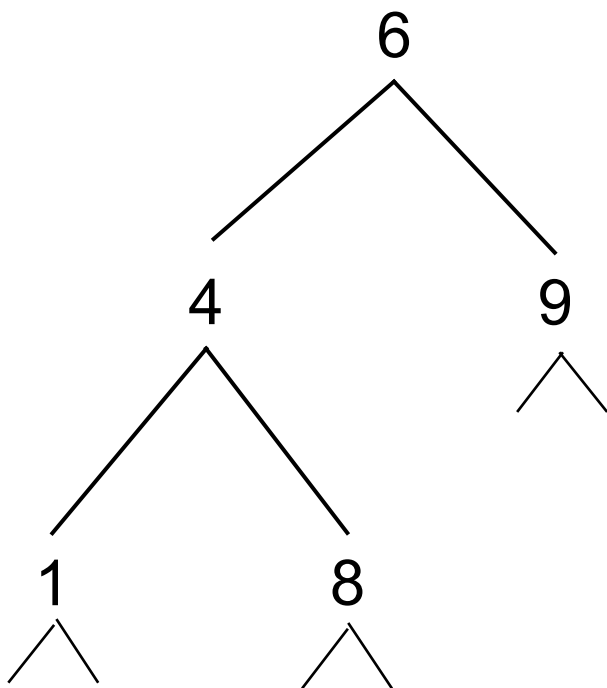
PRIMITIVES SUR LES ARBRES BINAIRES (2)

○ Primitives de construction

- **arbre-vidé** : crée et retourne un arbre vide
- **cons-binaire** : crée et retourne un arbre avec une valeur donnée et deux arbres donnés qui seront ses deux uniques fils

EXEMPLES D'UTILISATION DES PRIMITIVES

Soit l'arbre a :



(valeur a) \rightarrow 6

(valeur (fils-g a)) \rightarrow 4

(valeur (fils-d (fils-g a))) \rightarrow 8

(arbre? a) \rightarrow #t

(arbre-vide? a) \rightarrow #f

(arbre-vide? (fils-d a)) \rightarrow #f

(arbre-vide? (fils-g (fils-d a))) \rightarrow #t

(feuille? (fils-d a)) \rightarrow #t

(feuille? (fils-g a)) \rightarrow #f

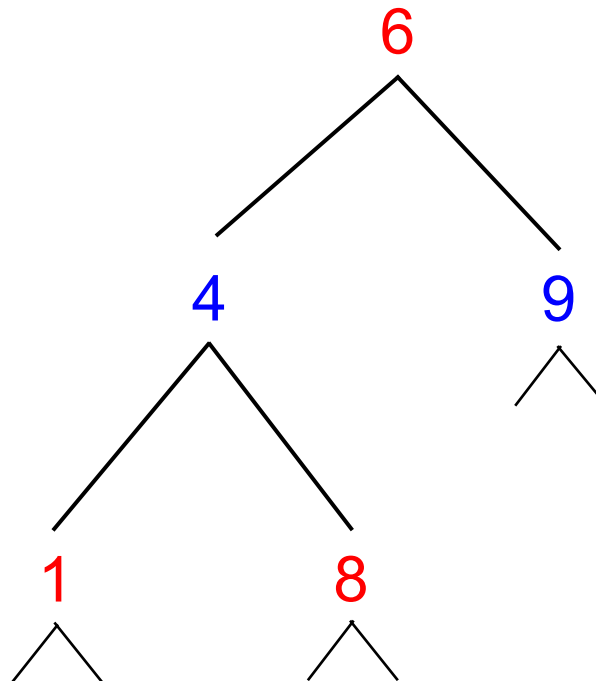
REPRÉSENTATION DES ARBRES BINAIRES

- Nous choisissons d'utiliser les **listes** pour représenter les arbres
- Un **arbre vide** sera représenté par la liste vide '()
- Un **noeud** sera une liste de 3 éléments
 - le **car** est sa **valeur**
 - le **cadr** son **fils gauche**
 - le **caddr** son **fils droit**

EXEMPLE DE REPRÉSENTATION D'UN ARBRE BINAIRE

(define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))

↓ ↓ ↓
valeur **fils-g** **fils-d**



DÉFINITIONS DES PRIMITIVES (1)

- **valeur** : retourne la valeur d'un arbre non vide
(define **valeur** ; → atome
 (lambda (arbre) ; arbre non vide
 (car arbre)))
- **fils-g** : retourne le fils de gauche d'un arbre non vide
(define **fils-g** ; → arbre
 (lambda (arbre) ; arbre non vide
 (cadr arbre)))

DÉFINITIONS DES PRIMITIVES (2)

- **fils-d** : retourne le fils de droite d'un arbre non vide
(define **fils-d** ; → arbre
 (lambda (arbre) ; arbre non vide
 (caddr arbre)))
- **arbre-vide?** : retourne vrai si un arbre donné est vide
(define **arbre-vide?** ; → booléen
 (lambda (arbre) ; arbre
 (null? arbre)))

DÉFINITIONS DES PRIMITIVES (3)

- **arbre?** : retourne vrai si une liste donnée est un arbre

(define **arbre?** ; → booléen

(lambda (l) ; liste

(or (null? l) ; l'arbre vide

(and (= 3 (length l))

(not (list? (car l)))

(list? (cadr l))

(arbre? (cadr l))

(list? (caddr l))

(arbre? (caddr l))))))

DÉFINITIONS DES PRIMITIVES (4)

- **feuille?** : retourne vrai si l'arbre donné est une feuille

```
(define feuille? ; → booléen  
  (lambda (arbre) ; arbre  
    (and (not (arbre-vidé? arbre))  
          (arbre-vidé? (fils-g arbre))  
          (arbre-vidé? (fils-d arbre))))))
```

- **arbre=?** : retourne vrai si deux arbres donnés sont égaux

```
(define arbre=? ; → booléen  
  (lambda (arbre1 arbre2) ; arbres  
    (equal? arbre1 arbre2)))
```

DÉFINITIONS DES PRIMITIVES (5)

- **arbre-vide** : retourne un arbre vide

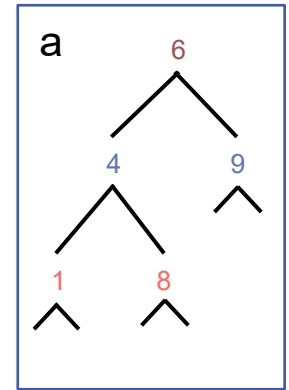
```
(define arbre-vide ; → arbre  
  (lambda ()  
    '() ))
```

- **cons-binaire** : crée un arbre avec une valeur donnée et deux arbres donnés qui seront ses deux uniques fils

```
(define cons-binaire ; → arbre  
  (lambda (val fg fd) ; val atome, fg et fd arbres  
    (list val fg fd)))
```

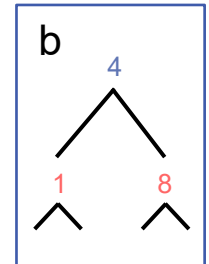
EXEMPLES

- (define a '(6 (4 (1 () ()) (8 () ())) (9 () ())))



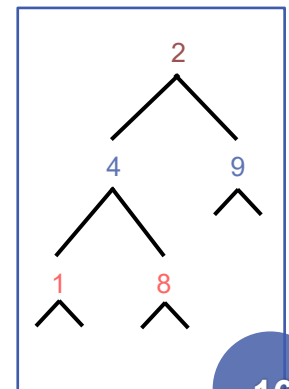
- (define b (fils-g a))

- b → (4 (1 () ()) (8 () ()))



- (cons-binaire 2 b (fils-d a))

→ (2 (4 (1 () ()) (8 () ())) (9 () ()))



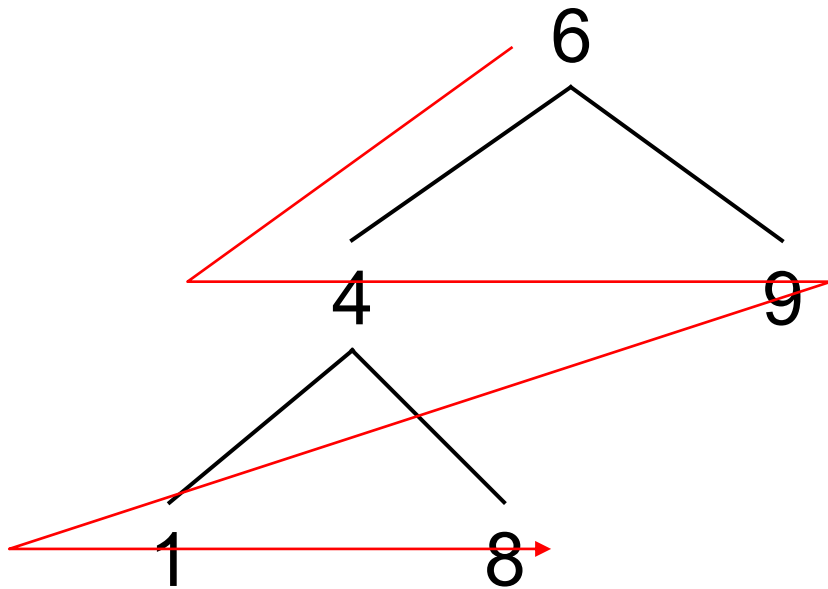
POURQUOI UTILISER DES PRIMITIVES ?

- Pourquoi utiliser (arbre-**vide**) au lieu de '()' et **fils-g** au lieu de **cadr** ?
- Si on décide de changer la représentation des arbres :
 - sans primitives, il faut réécrire toutes les fonctions sur les arbres
 - avec primitives, il suffit de modifier les primitives

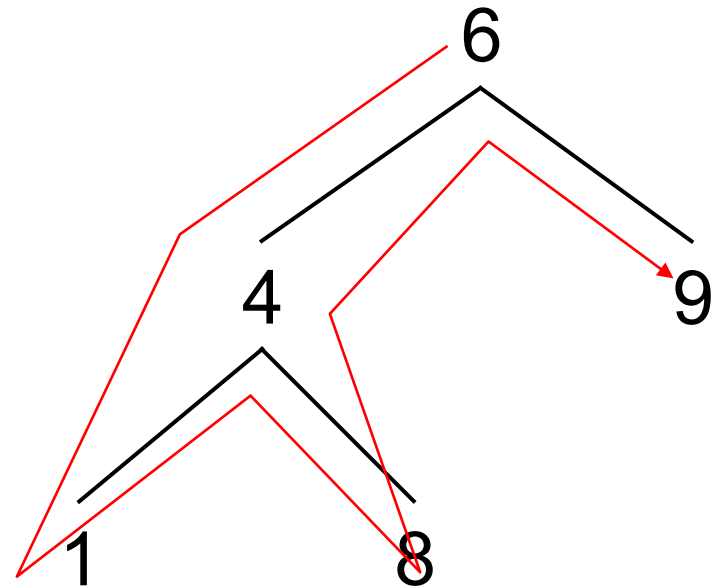
PARCOURS D'ARBRES

- Un arbre contient un ensemble de données
- Pour utiliser ces données, il faut **parcourir** l'arbre
 - en profondeur
 - ou en largeur

PARCOURS EN LARGEUR / PROFONDEUR



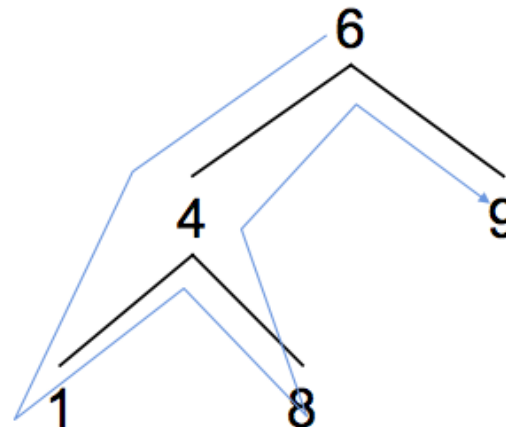
Parcours en largeur



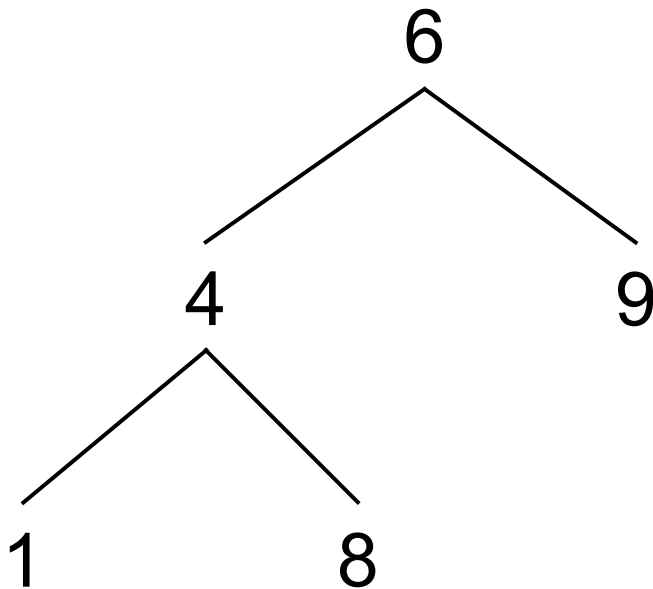
Parcours en profondeur

PARCOURS EN PROFONDEUR : PRINCIPE

- Parcourir un arbre en profondeur consiste à **passer ses nœuds en revue**, en commençant toujours par le même fils, et en **descendant le plus profondément possible** dans l'arbre
- Lorsque l'on arrive sur un arbre vide, on **remonte** jusqu'au nœud supérieur et on **redescend** dans le fils encore inexploré



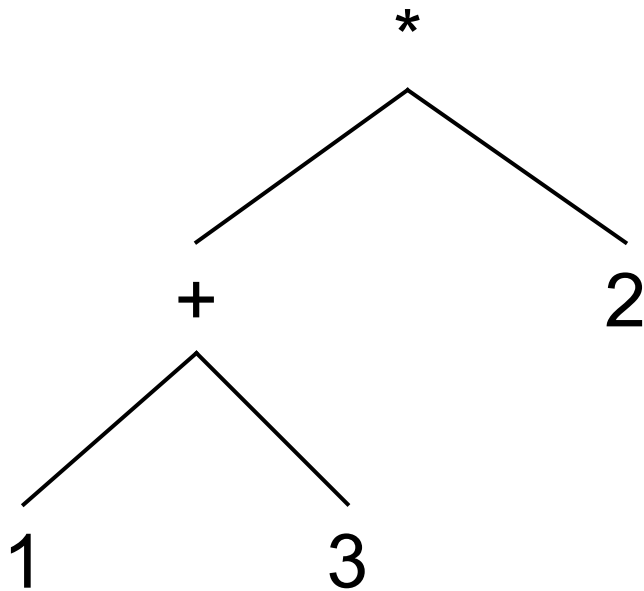
3 PARCOURS EN PROFONDEUR



- Parcours **infixe** :
 - Fils-g **Valeur** Fils-d
 - □ 1 4 8 6 9
- Parcours **préfixe** :
 - **Valeur** Fils-g Fils-d
 - □ 6 4 1 8 9
- Parcours **postfixe** :
 - Fils-g Fils-d **Valeur**
 - □ 1 8 4 9 6

EXEMPLE

En ajoutant des parenthèses pour chaque fils



- Parcours **infixe** :
 - □ (((1) + (3)) * (2))
 - notation classique math
- Parcours **préfixe** :
 - □ (* (+ (1) (3)) (2))
 - notation préfixe Scheme
- Parcours **postfixe** :
 - □ (((1) (3) +) (2) *)
 - parenthèses inutiles

POUR ÉCRIRE UNE FONCTION QUI EFFECTUE UN PARCOURS EN PROFONDEUR

- Pour écrire une fonction f , sur un arbre A
 - Si A est vide, on retourne une valeur constante, généralement l'élément neutre de f
 - Si A n'est pas vide :
 - on rappelle f sur les deux fils de A , ce qui retourne deux résultats : R_g et R_d
 - puis on retourne un résultat qui ne dépend que de R_g , R_d et de la valeur de la racine de A

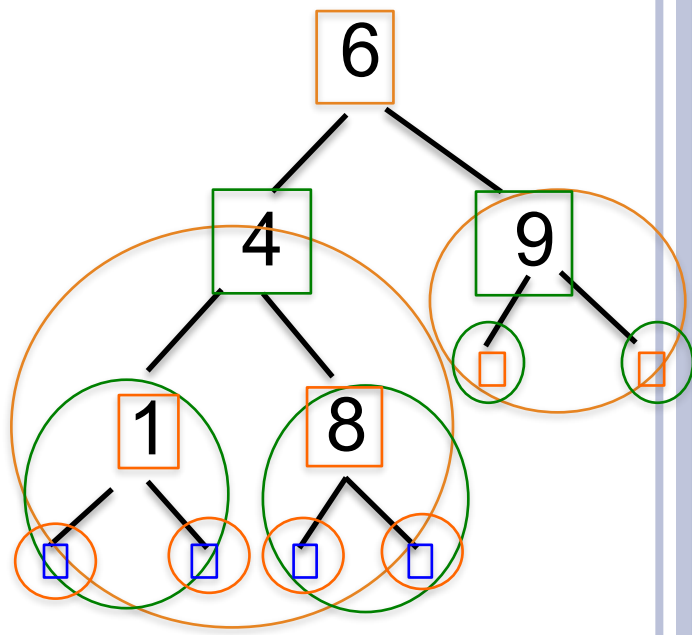
EXEMPLE : SOMME DES VALEURS D'UN ARBRE

```
(define somme ; → nombre  
  (lambda (A) ; A arbre de nombres  
    (if (arbre-vide? A)  
        0  
        (+ (somme (fils-g A))  
           (somme (fils-d A))  
           (valeur A)))))
```

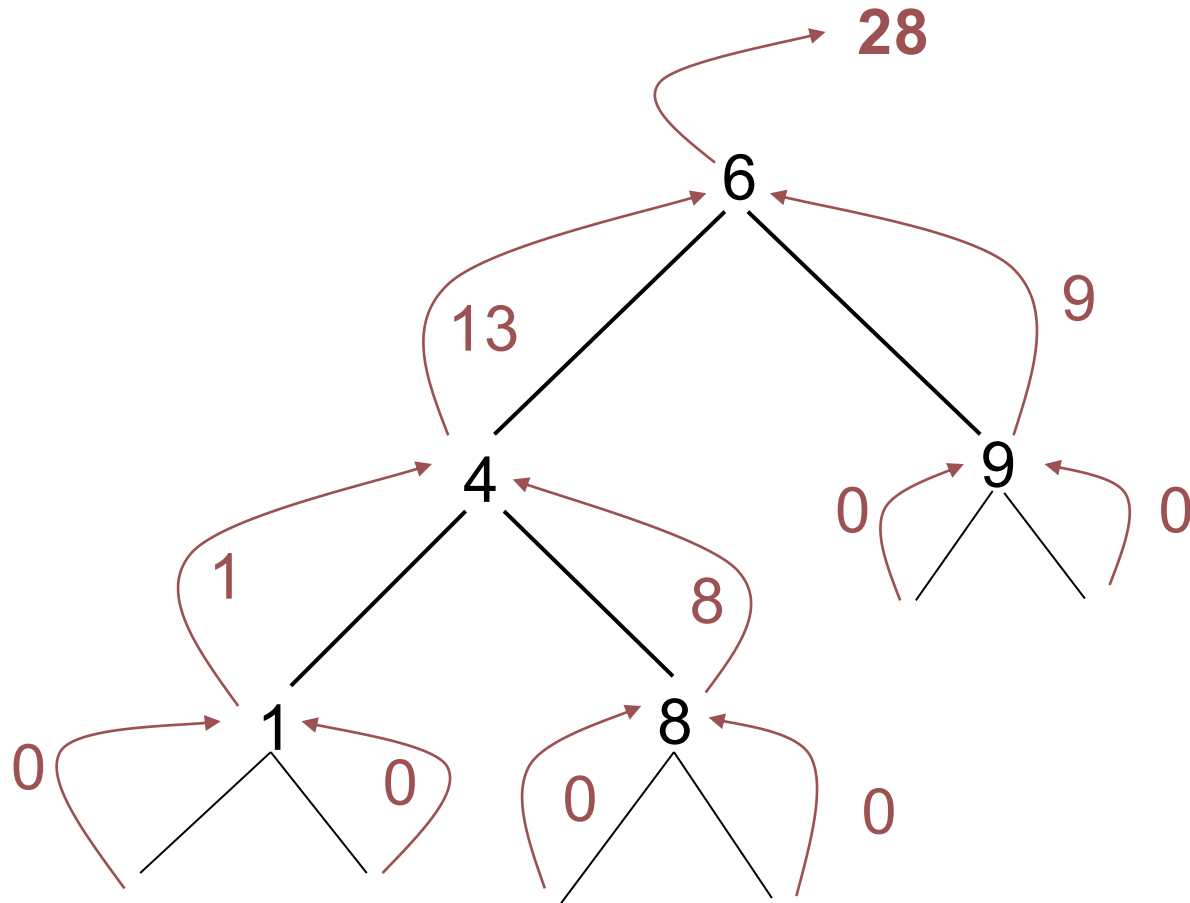

(somme A)

(+ (somme (fils-g A)
 (+ (somme (fils-g A)
 (+ (somme (fils-g A)
 (somme (fils-d A)
 0
 0
 1)
 (somme (fils-d A)
 (+ (somme (fils-g A)
 (somme (fils-d A)
 0
 8)
 0
 4)
 (somme (fils-d A)
 (+ (somme (fils-g A)
 0
 (somme (fils-d A)
 0
 0
 9)
 0
 6)

ILLUSTRATION



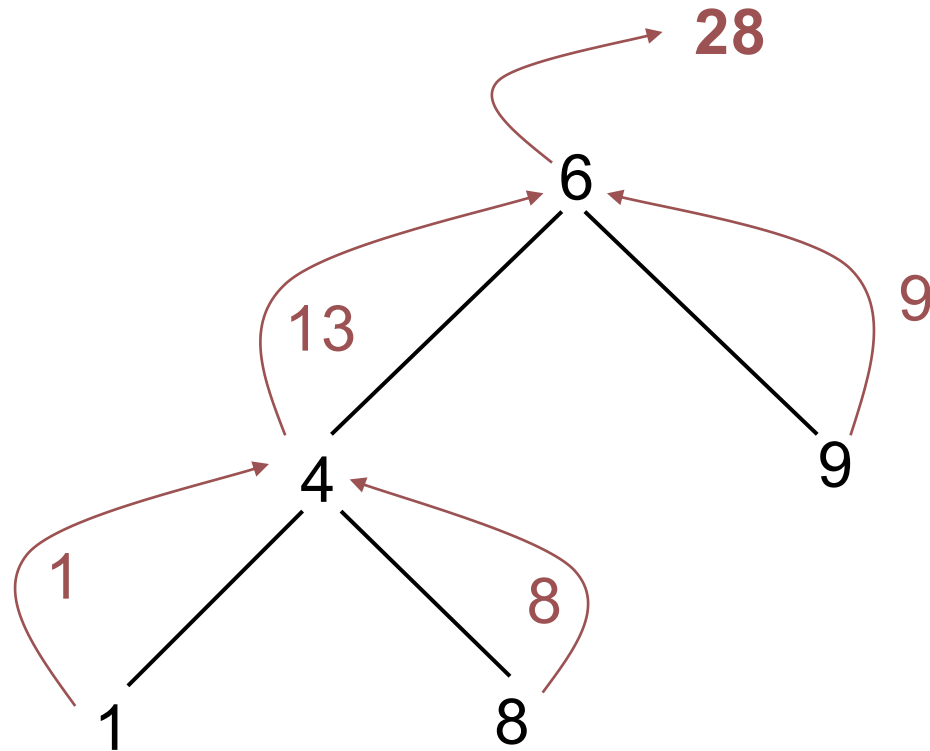
FONCTIONNEMENT SUR UN EXEMPLE



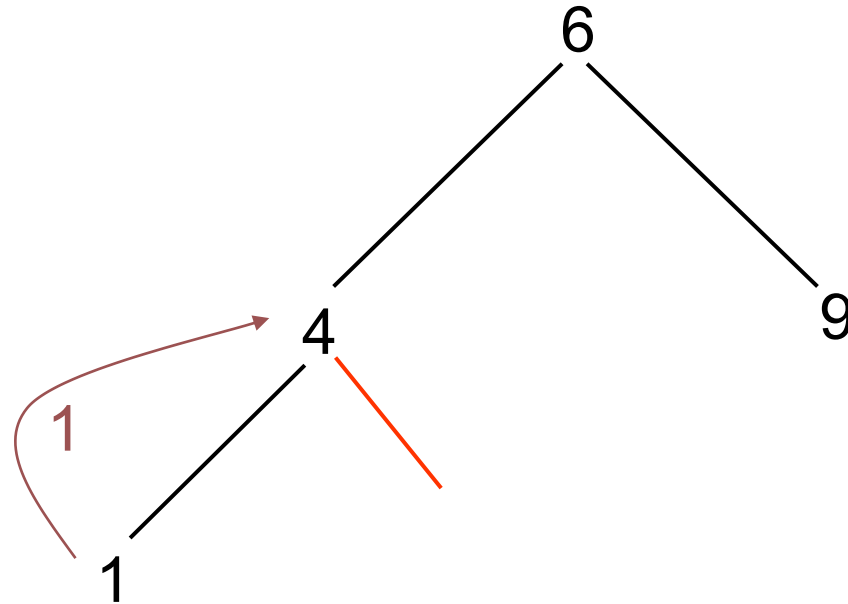
MODIFICATION DU CAS D'ARRÊT

```
(define somme ; → nombre  
  (lambda (A) ; A arbre de nombres  
    (if (feuille? A)  
        (valeur A)  
        (+ (somme (fils-g A))  
           (somme (fils-d A))  
           (valeur A))))))
```

PREMIER TEST



DEUXIÈME TEST



Il manque le cas de l'arbre vide

MORALITÉ

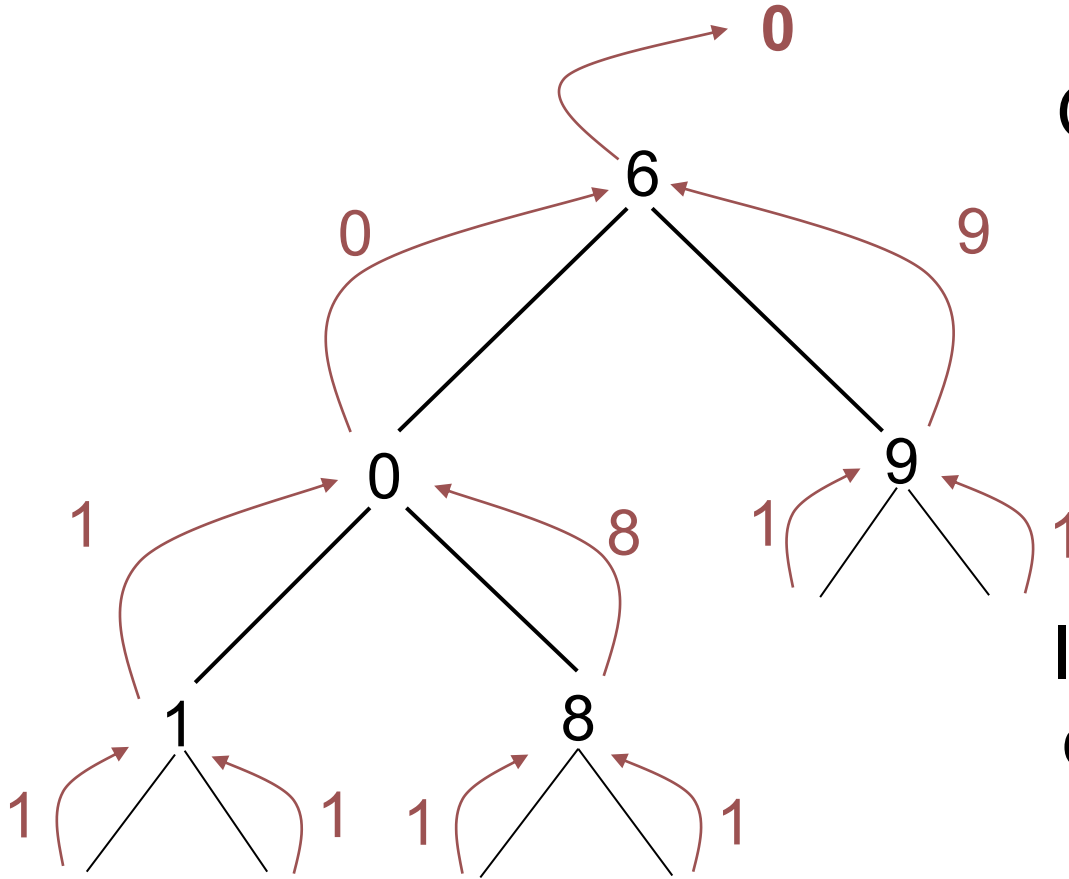
- Il faut toujours tester les fonctions sur un arbre dont un nœud n'a qu'un seul fils
- Quand la fonction peut retourner une valeur pour l'arbre vide, il faut toujours prévoir ce cas d'arrêt
 - Quand ce n'est pas possible, il y a de nombreux cas à prévoir (voir fonction minimum, TDTP7) :
 - feuille
 - fils gauche vide mais pas le fils droit
 - fils droit vide mais pas le fils gauche
 - deux fils non vides

PARCOURS PARTIELS

- Il est parfois souhaitable d'arrêter le parcours même si tous les nœuds n'ont pas été passés en revue
- Exemple : produit des valeurs d'un arbre

```
(define produit ; → nombre  
  (lambda (A) ; A arbre de nombres  
    (if (arbre-vide? A)  
        1  
        (* (produit (fils-g A))  
           (produit (fils-d A))  
           (valeur A))))))
```

PREMIER TEST



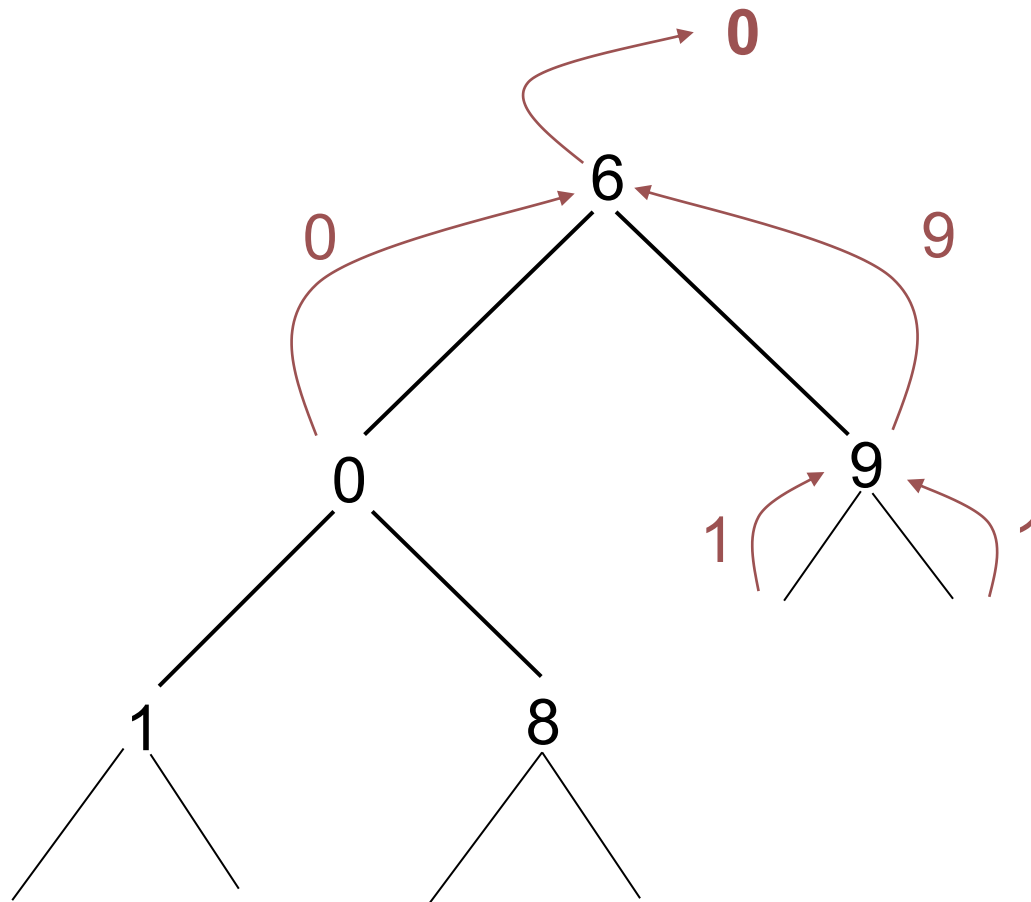
On fait des calculs inutiles

Il faut s'arrêter dès
qu'on rencontre la
valeur 0

MODIFICATION DE LA FONCTION

```
(define produit ; → nombre
  (lambda (A) ; A arbre de nombres
    (cond ((arbre-vide? A) 1)
          ((= 0 (valeur A)) 0)
          (else (* (produit (fils-g A))
                    (produit (fils-d A))
                    (valeur A))))))
```

DEUXIÈME TEST



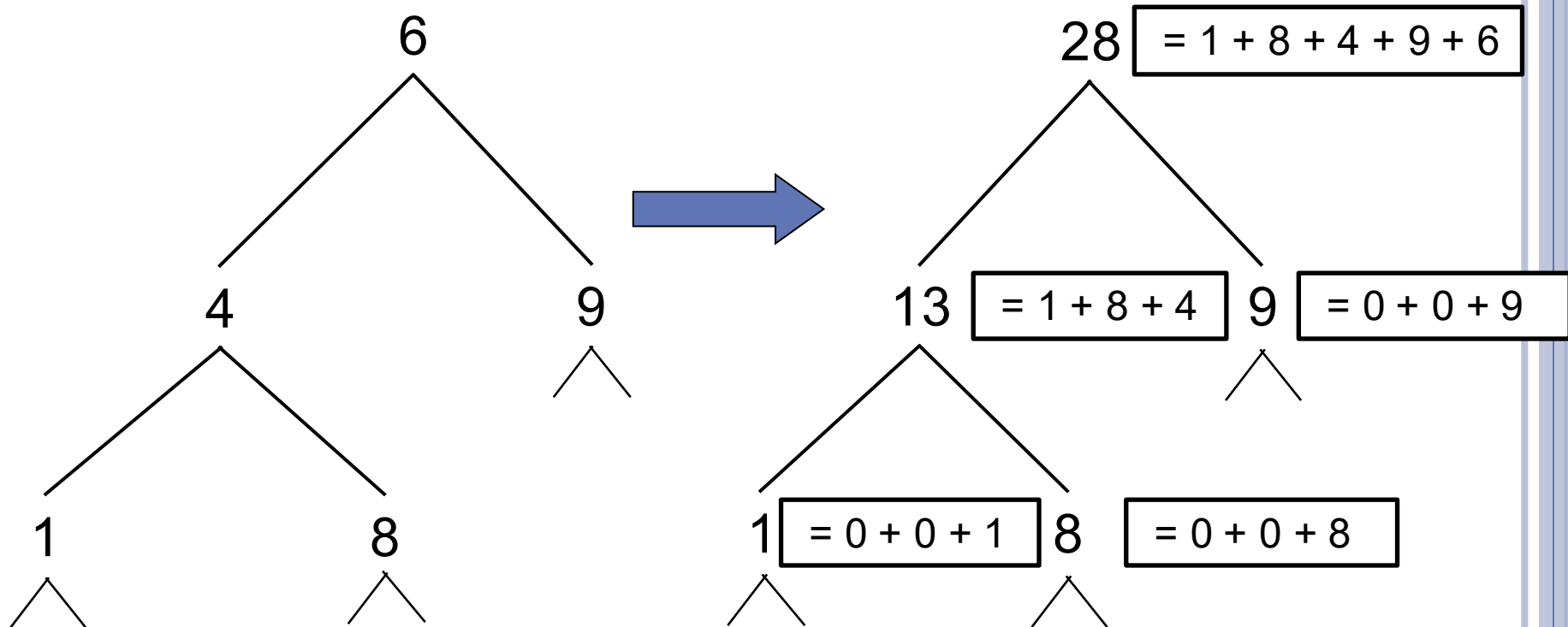
MODIFICATION ET CRÉATION D'ARBRES

- Exemple : écrire une fonction qui ajoute 1 à tous les nœuds d'un arbre qui contient des nombres
- Il ne s'agit pas d'une modification (ajouter 1), mais d'une création :
écrire une fonction qui retourne un arbre identique à celui passé en argument, mais dans lequel on a ajouté 1 à tous les nœuds

FONCTION AJOUTE1

```
(define ajoute1 ; → arbre  
  (lambda (A) ; A arbre de nombres  
    (if (arbre-vide? A)  
        A  
        (cons-binaire (+ 1 (valeur A))  
                       (ajoute1 (fils-g A))  
                       (ajoute1 (fils-d A))))))
```

SOMME DES VALEURS DES FILS



PREMIÈRE SOLUTION :

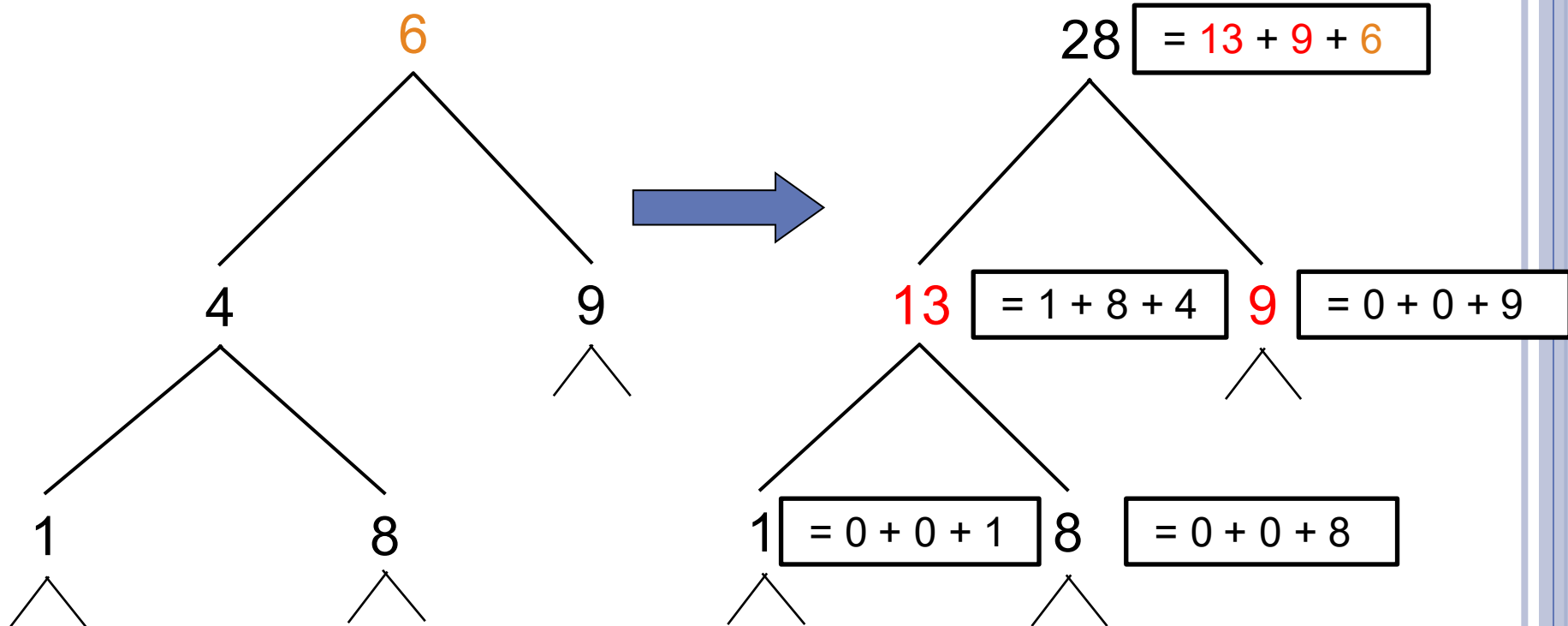
UTILISER LA FONCTION SOMME

```
(define somme-fils ; → arbre  
  (lambda (a) ; a arbre de nombres  
    (if (arbre-vide? a)  
        (arbre-vide)  
        (cons-binaire  
          (somme a)  
          (somme-fils (fils-g a))  
          (somme-fils (fils-d a))))))
```

RÉFLEXION SUR CETTE FONCTION

- La complexité de cette fonction est beaucoup trop grande
- Il faut utiliser la valeur de la racine du résultat de l'appel récursif sur les fils :
 - ils contiennent déjà la somme des valeurs de tous les nœuds de chacun des fils

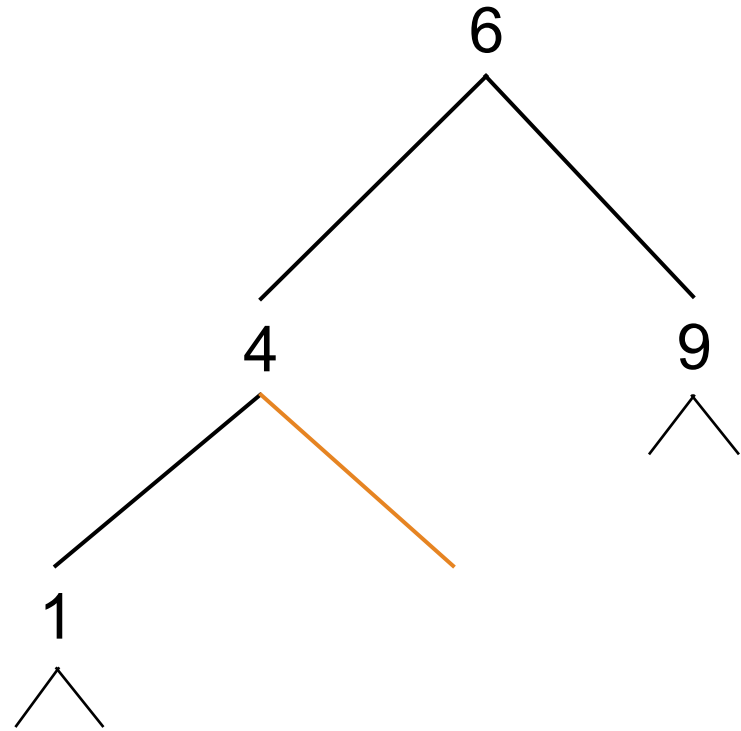
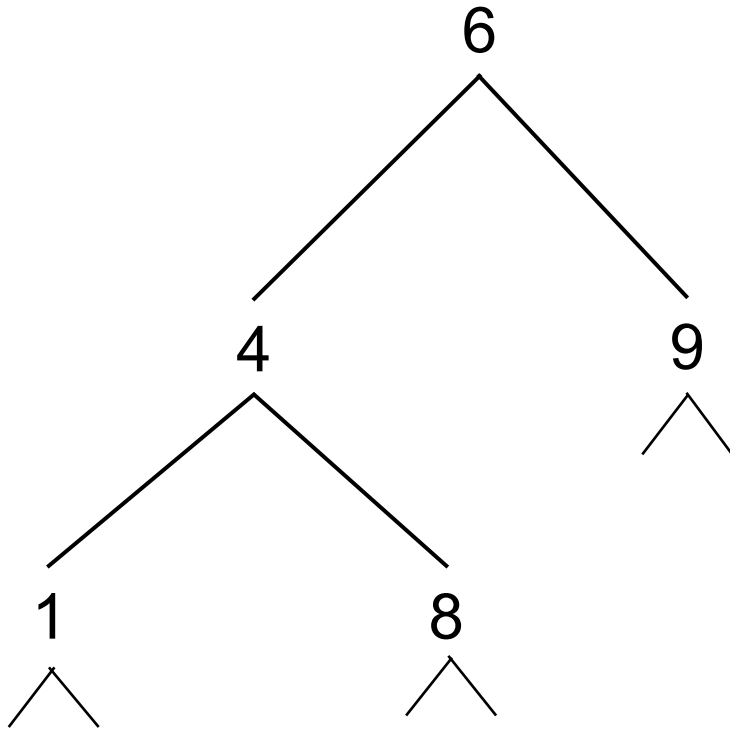
SOMME DES VALEURS DES FILS



MODIFICATION DE LA FONCTION

```
(define somme-fils ; → arbre
  (lambda (A) ; A arbre de nombres
    (if (arbre-vide? A)
        (arbre-vide)
        (let ((g (somme-fils (fils-g A)))
              (d (somme-fils (fils-d A))))
          (cons-binaire
            (+ (valeur A) (valeur g) (valeur d))
            g
            d))))))
```

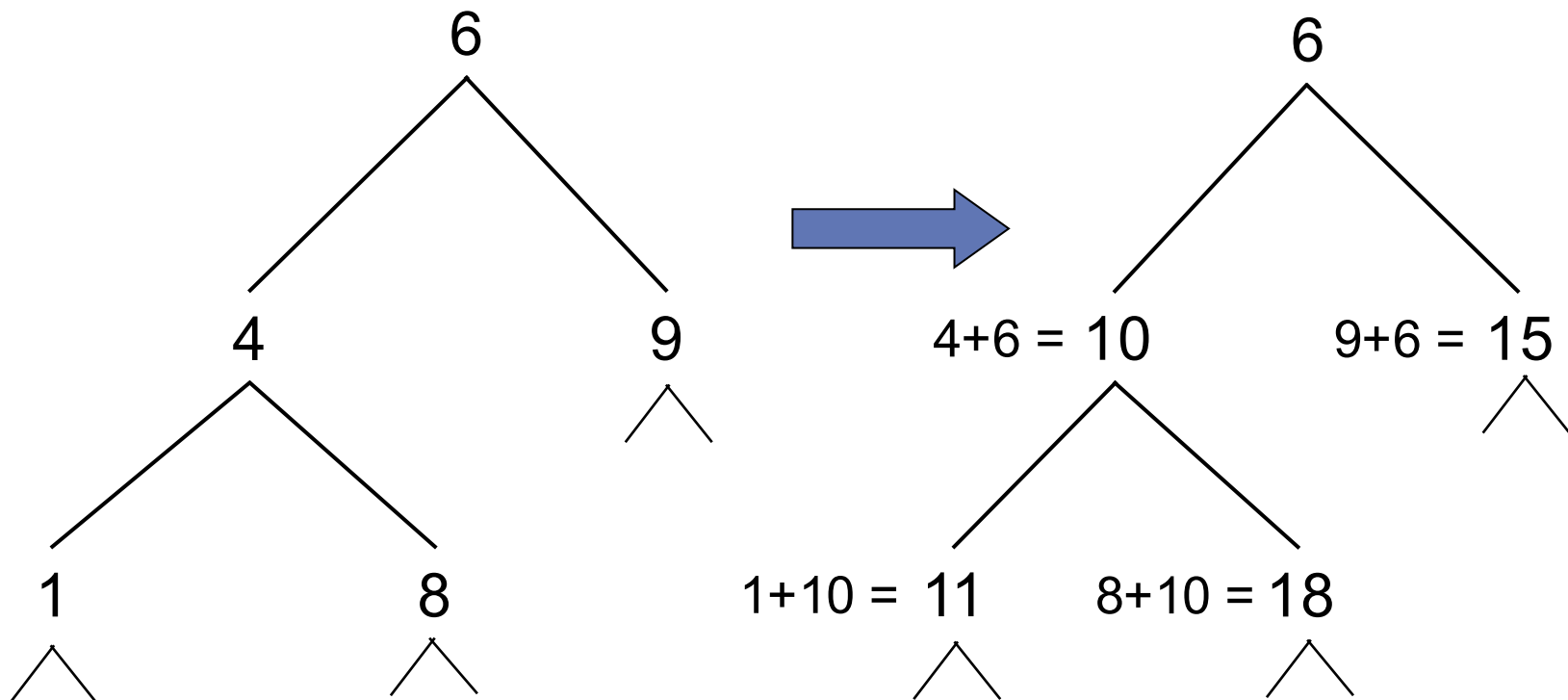
TEST DE LA FONCTION



CORRECTION DE LA FONCTION

```
(define somme-fils ; → arbre
  (lambda (A) ; A arbre
    (if (arbre-vide? A)
        (arbre-vide)
        (let ((g (somme-fils (fils-g A)))
              (d (somme-fils (fils-d A))))
          (cons-binaire
            (+ (valeur A)
              (if (arbre-vide? g) 0 (valeur g))
              (if (arbre-vide? d) 0 (valeur d)))
            g
            d))))))
```

SOMME DES VALEURS DES PÈRES



LE PÈRE DOIT PARLER À SES FILS

- Dans toutes les fonctions précédemment écrites, le résultat dépendait des **fils**
- Ici, le résultat dépend du **père**
 - ⇒ il doit transmettre une information à ses fils au moment des appels récurifs
 - ⇒ **paramètre supplémentaire** pour passer le résultat du père au fils
- NB : la fonction construit quand même le résultat en remontant

FONCTION SOMME-PERE

- (define **somme-pere** ; → arbre
 (lambda (A) ; A arbre de nombres
 (**somme-pere2** A 0)))
- (define **somme-pere2** ; → arbre
 (lambda (A n) ; A arbre de nb, n nombre
 (if (arbre-vide? A)
 (arbre-vide)
 (let ((v (+ (valeur A) n)))
 (cons-binaire
 v
 (**somme-pere2** (fils-g A) v)
 (**somme-pere2** (fils-d A) v)))))))

(Somme-pere2 A 0)

$$v = 0 + 6$$

(cons-binaire 6

(somme-pere2 (fils-g A) 6)

$$v = 6 + 4 = 10$$

(cons-binaire 10

(somme-pere2 (fils-g A) 10)

$$v = 10 + 1 = 11$$

(cons-binaire 11

(somme-pere2 (fils-g A) 11) → (arbre-vide)

(somme-pere2 (fils-d A) 11) → (arbre-vide)

(somme-pere2 (fils-d A) 10)

$$v = 10 + 8 = 18$$

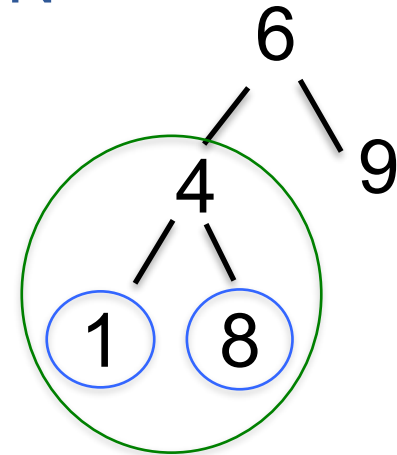
(cons-binaire 18

(somme-pere2 (fils-g A) 18) → (arbre-vide)

(somme-pere2 (fils-d A) 18) → (arbre-vide)

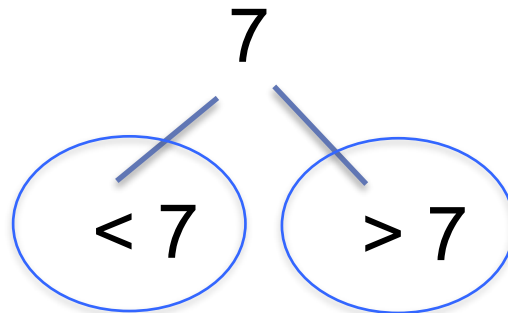
(somme-pere2 (fils-d A) 6)

ILLUSTRATION



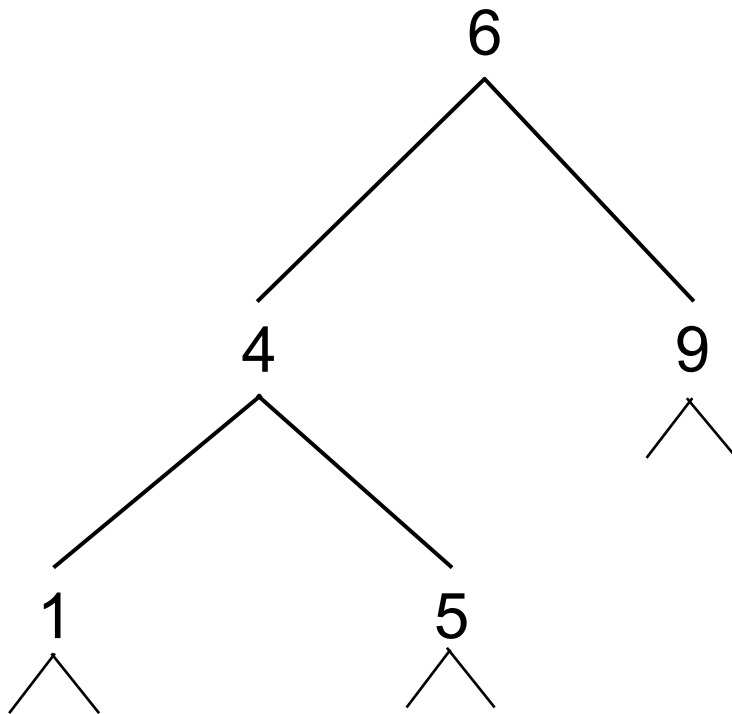
ARBRES BINAIRES DE RECHERCHE (OU ORDONNÉS)

- Les valeurs des nœuds doivent pouvoir être ordonnées
- En chaque nœud de l'arbre, la valeur du nœud est :
 - supérieure à toutes celles de son fils gauche
 - inférieure à toutes celles de son fils droit

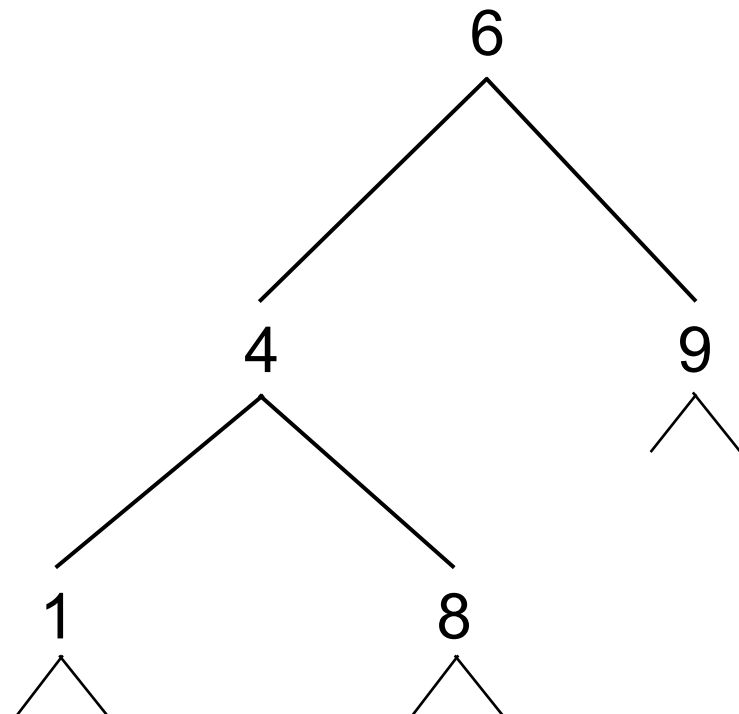


- On suppose qu'il n'y a pas deux fois la même valeur dans un ABR

EXEMPLES



Arbre ordonné



Arbre non ordonné

RECHERCHE D'UN ÉLÉMENT DANS UN ARBRE BINAIRE QUELCONQUE (1)

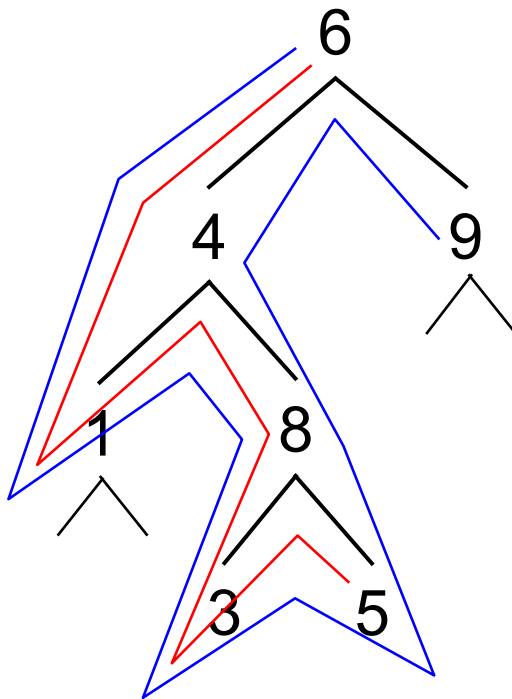
- On souhaite écrire une fonction qui teste l'appartenance d'une valeur V à un arbre A
- Principe : tant qu'on n'a pas trouvé la valeur V , il faut comparer V avec toutes les valeurs de l'arbre A

RECHERCHE D'UN ÉLÉMENT DANS UN ARBRE BINAIRE QUELCONQUE (2)

- Algorithme :

- Cas d'arrêt :
 - Si A est vide Alors Retourne Faux
 - Si $\text{valeur}(A)=V$ Alors Retourne Vrai
- Appels récurifs :
 - Chercher V dans $\text{fils-gauche}(A)$
 - Puis si on n'a toujours pas trouvé V, chercher V dans $\text{fils-droit}(A)$

EXEMPLE



Recherche fructueuse :
Chercher 5

Cas le pire
Recherche infructueuse :
Chercher 7

Complexité au pire :
nombre de nœuds de
l'arbre

RECHERCHE D'UN ÉLÉMENT DANS UN ARBRE BINAIRE ORDONNÉ (1)

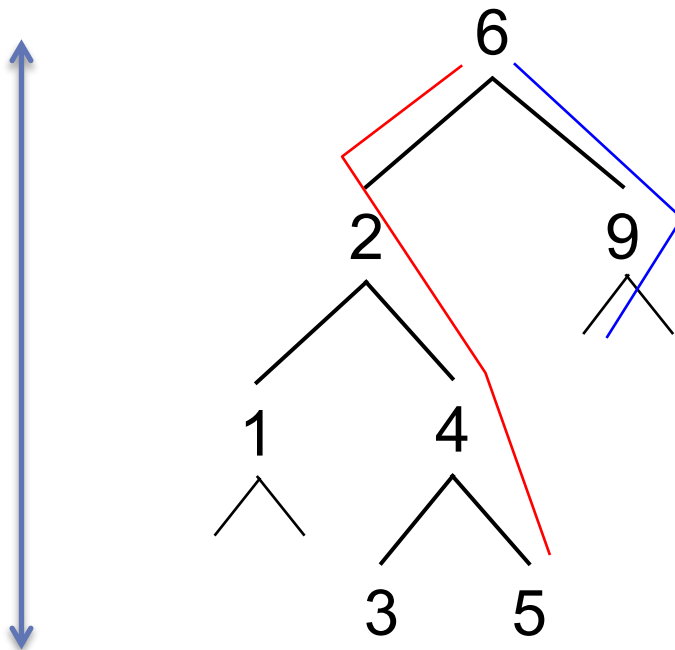
- Principe : utiliser le fait que l'arbre est ordonné pour choisir dans quelle branche de l'arbre chercher

RECHERCHE D'UN ÉLÉMENT DANS UN ARBRE BINAIRE ORDONNÉ (2)

- Algorithme :

- Cas d'arrêt :
 - Si A est vide Alors Retourne Faux
 - Si $\text{valeur}(A) = V$ Alors Retourne Vrai
- Appels récurifs :
 - Si $V > \text{valeur}(A)$ Alors chercher V dans $\text{fils-droit}(A)$
 - Si $V < \text{valeur}(A)$ Alors chercher V dans $\text{fils-gauche}(A)$

EXEMPLE



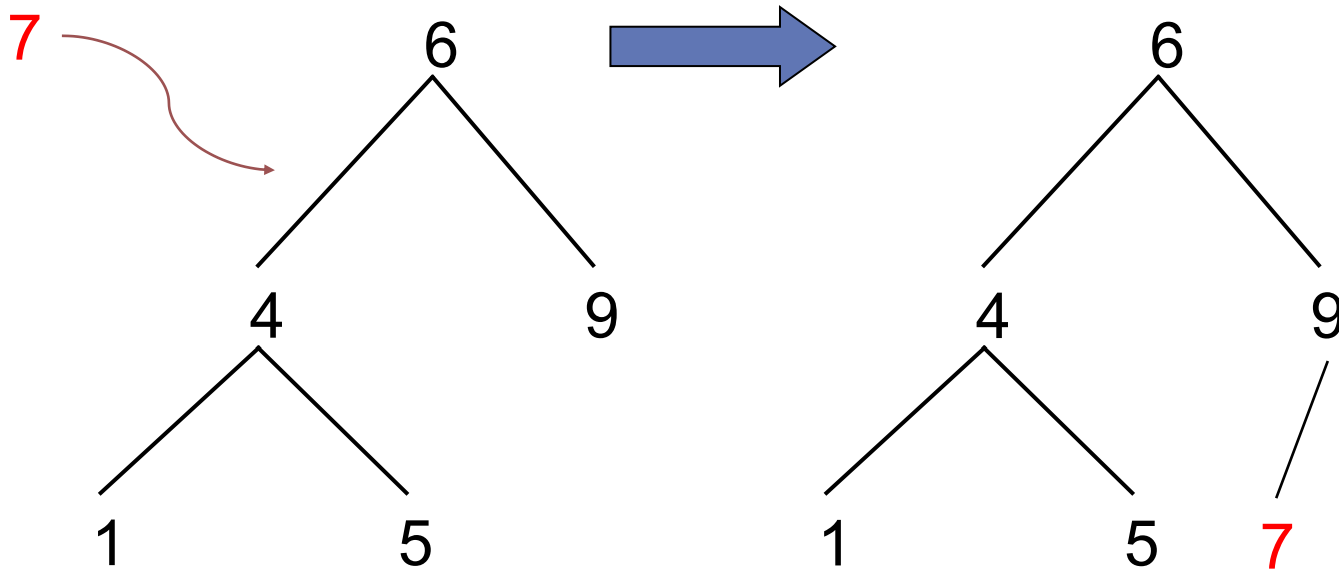
Recherche fructueuse :
Chercher 5

Recherche infructueuse :
Chercher 7

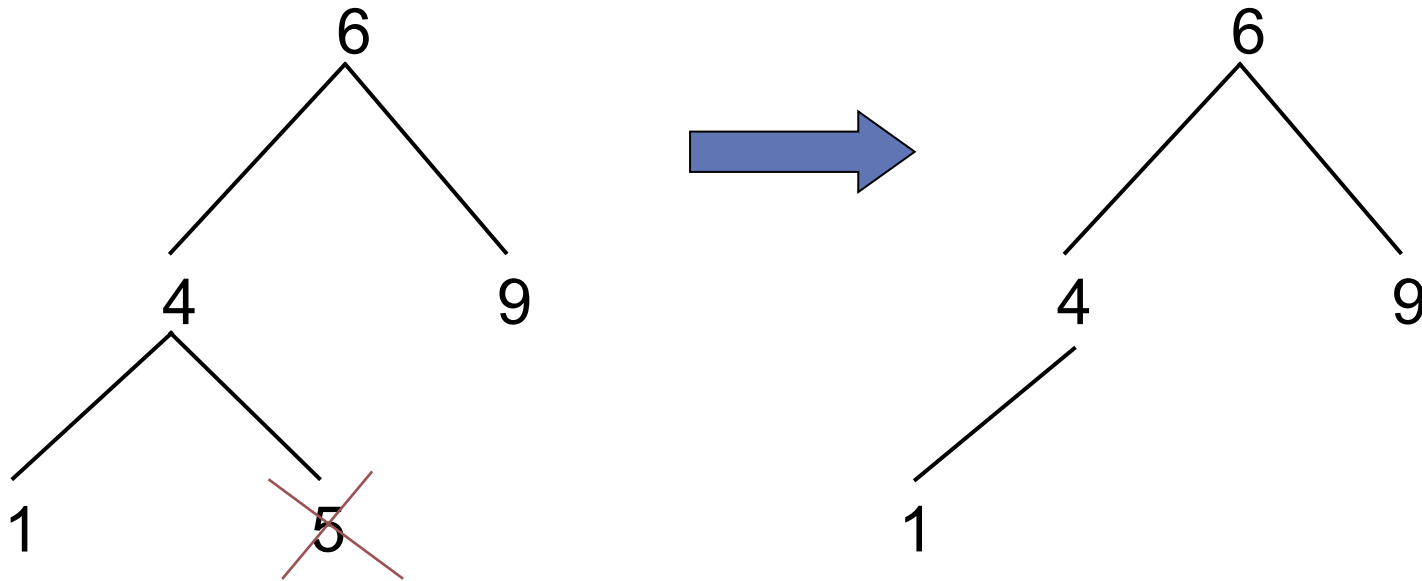
Complexité au pire :
hauteur de l'arbre

INSERTION DANS UN ABR

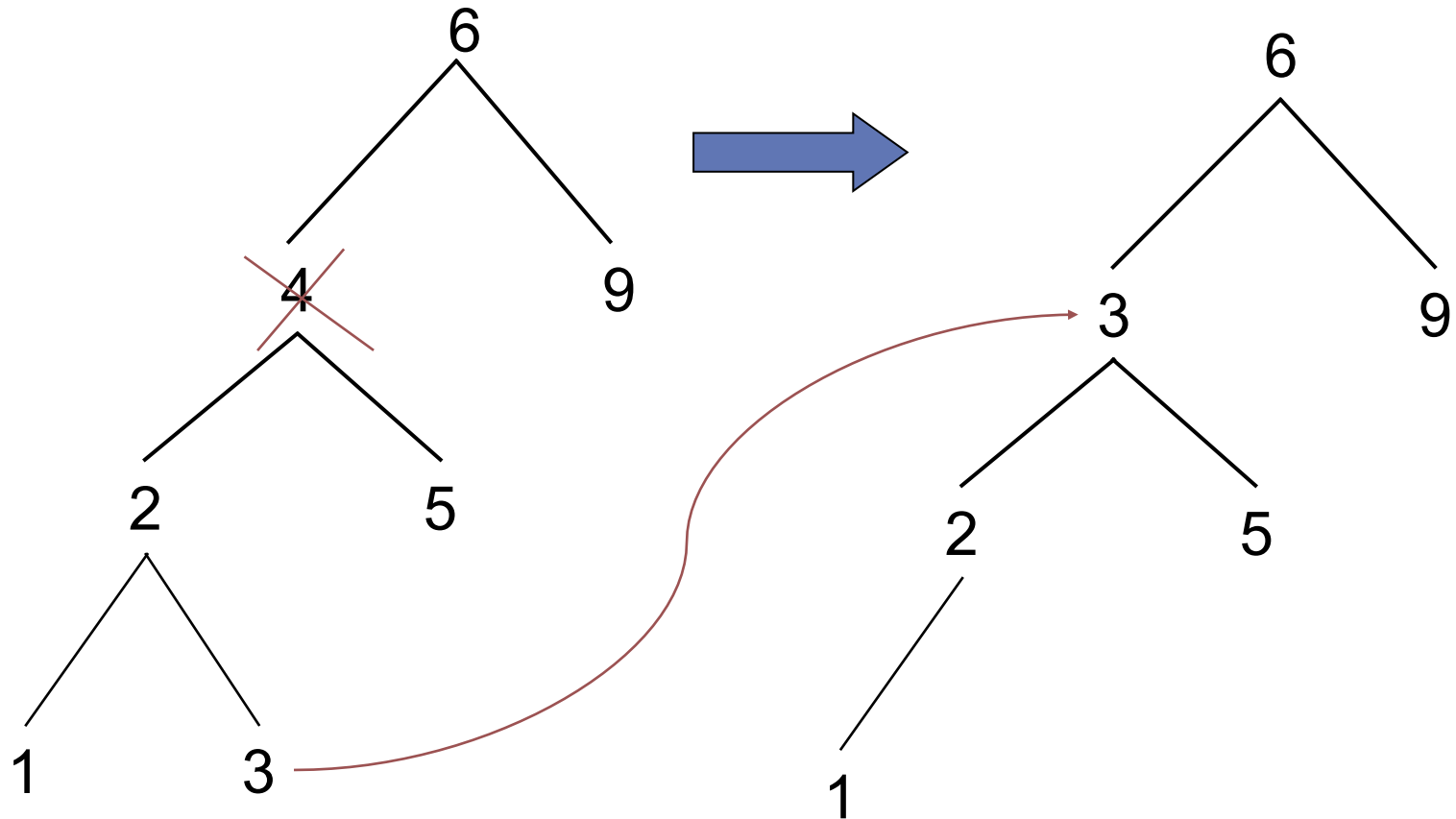
- Principe : on insère aux feuilles



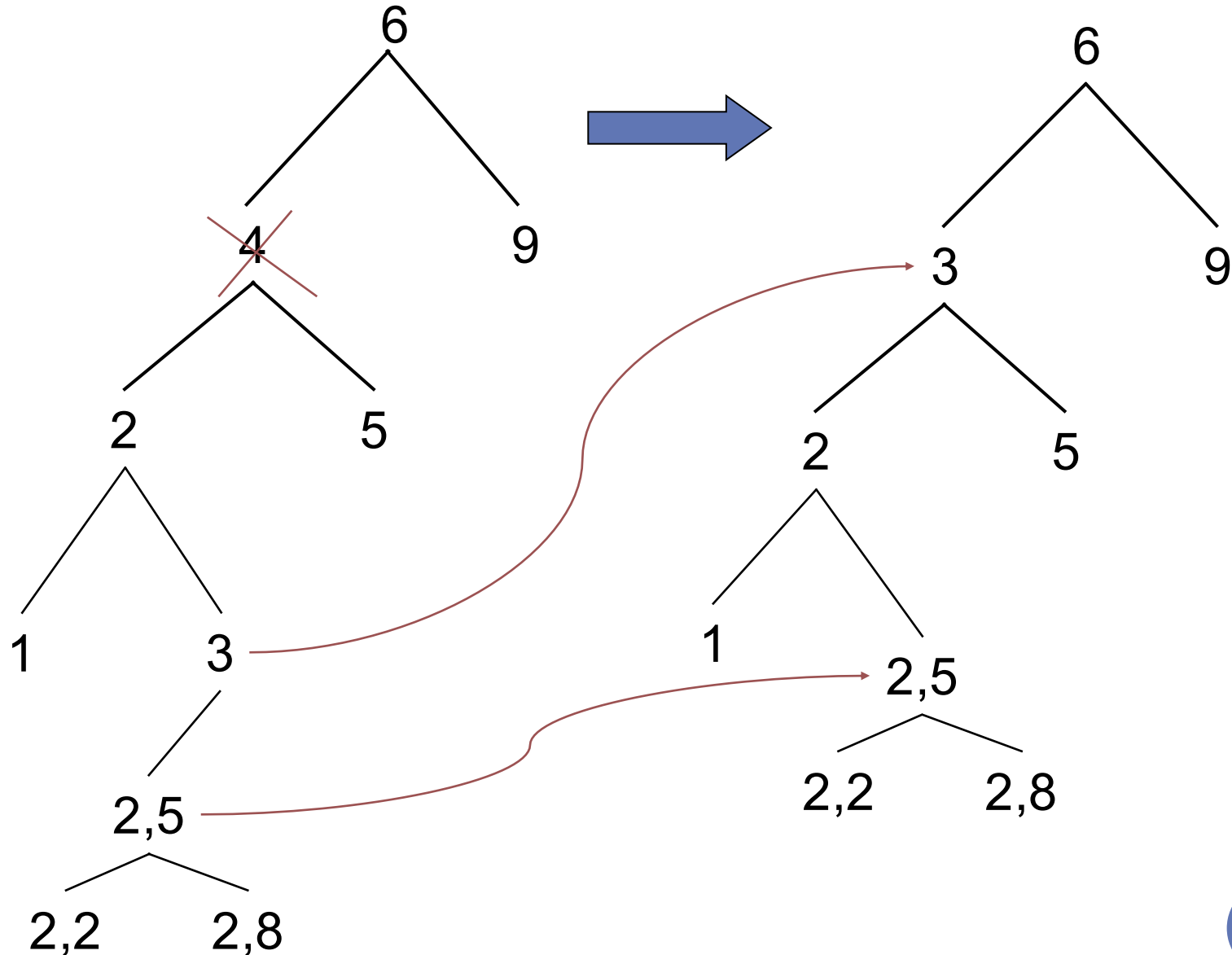
SUPPRESSION DANS UN ABR



SUPPRESSION DANS UN ABR



SUPPRESSION DANS UN ABR



SUPPRESSION DANS UN ABR

- Pour supprimer la valeur V dans un ABR
 - Si V est une feuille, alors on supprime la feuille
 - Sinon on remplace la valeur V par la valeur V' qui lui est immédiatement inférieure (ou immédiatement supérieure), de manière à respecter l'ordre, puis on supprime V' qui est le plus grand élément du fils gauche de V (resp. le plus petit élément de son fils droit)
 - V' est une feuille ou un élément qui n'a pas de fils droit (resp. pas de fils gauche), et peut donc être supprimée facilement