Programming et Algorithme III Cours V: Arbre

Plan du cours

- Introduction d'arbre
- Parcours sur les arbres
- Arbre binaire (recherche)
- Tas

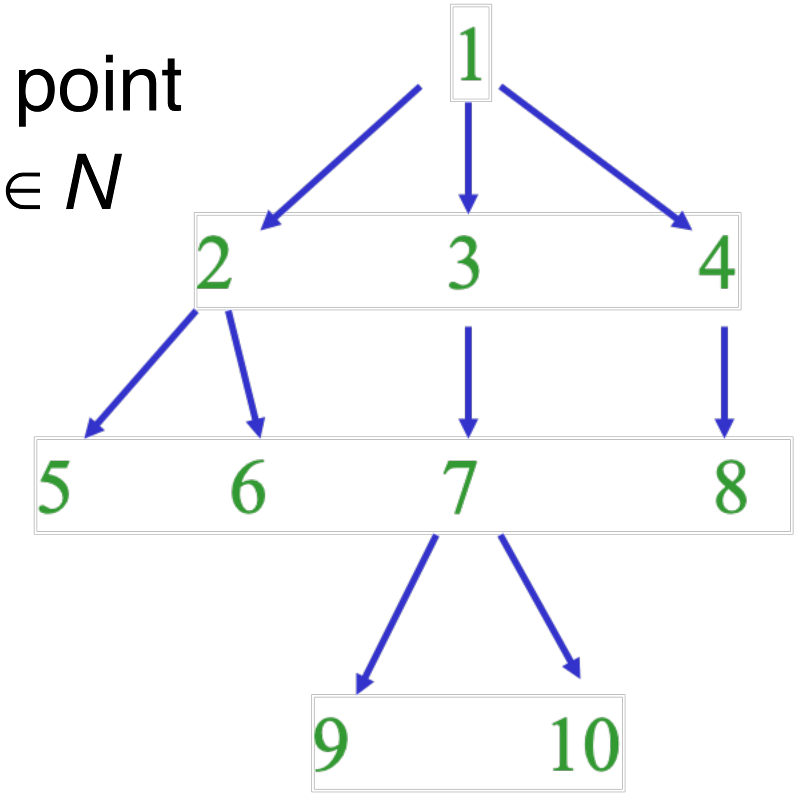
Arbre (Tree): vocabulaire

- Arbre ordinaire : $A = (N,P)$
  - N ensemble des nœuds (ou sommet, nodes)
  - Ces sommets sont reliés par des arcs (ou arêtes) orientés : parent (père) → enfant (fils), noté par une relation binaire *P* (« parent (père) de »)
  - Il existe dans tout arbre un nœud qui n'est le point d'arrivée d'aucun arc : c'est la racine (root) $r \in N$
  - une type de graphe
- $\forall x \in N$, $\exists$ un seul chemin de r vers x

$r = y_0 \;P\; y_1 \;P\; y_2 \;\cdots\; P\; y_n = x$

=> r n'a pas de parent

=> $\forall\, x \in N - \{ r \}$ $\;x$ a exactement un parent



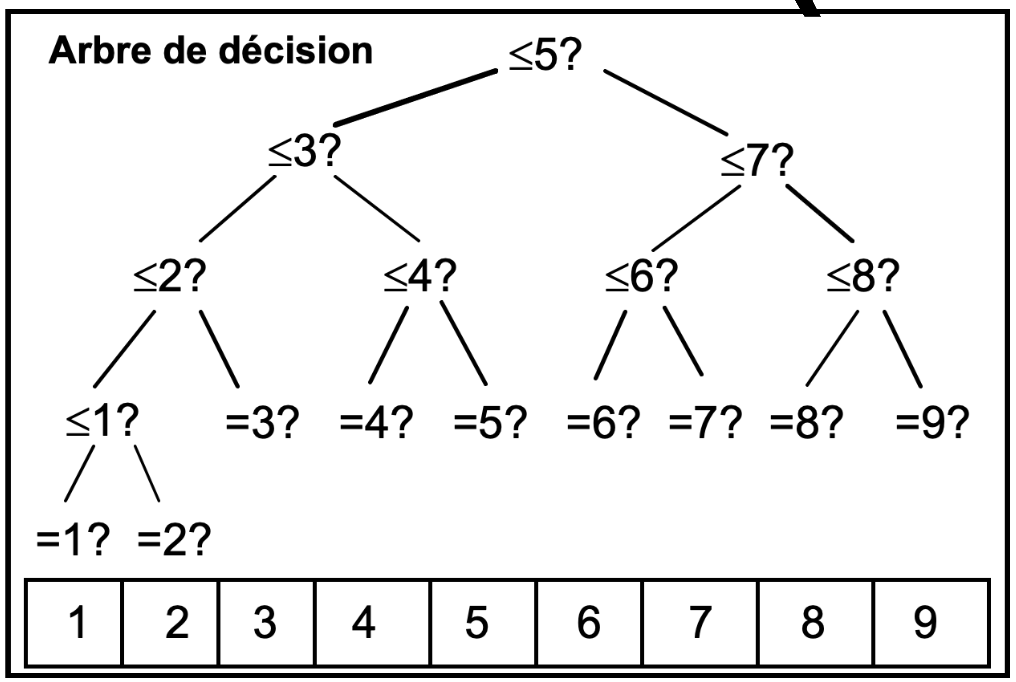
CERGY PARIS UNIVERSITÉ

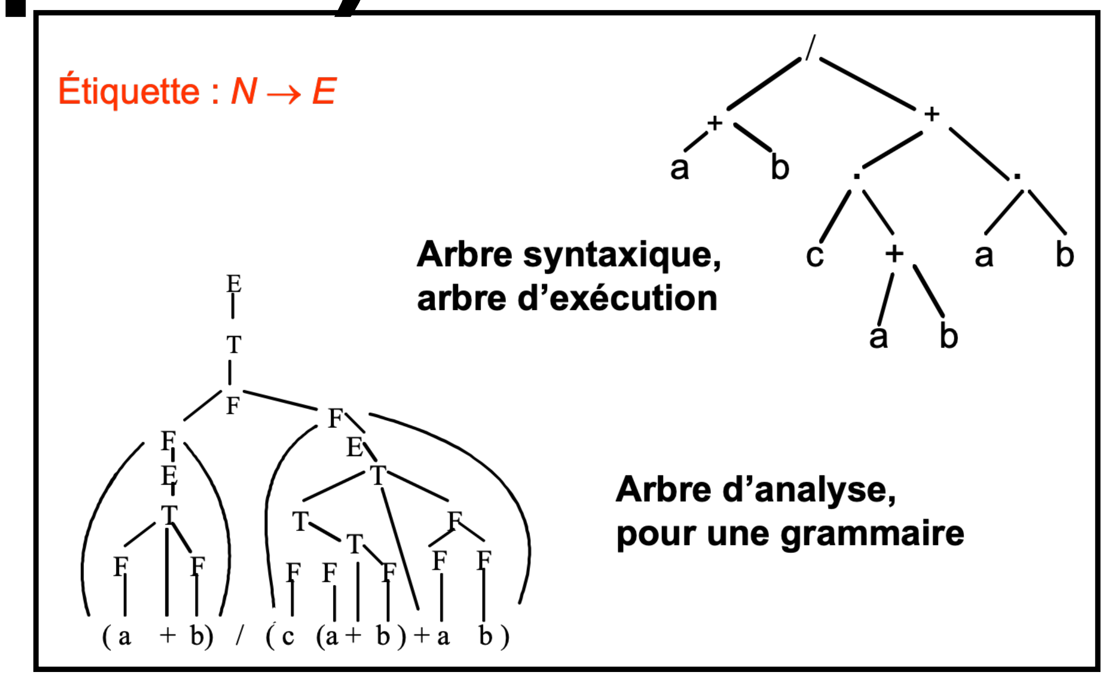
ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

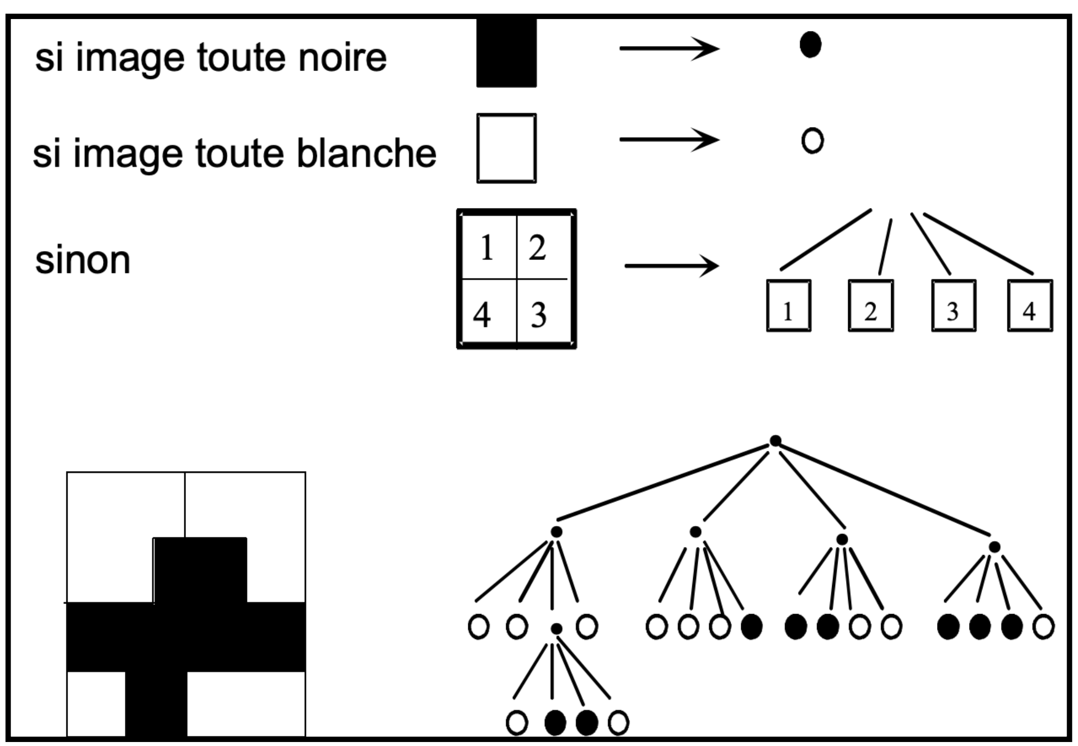
Utilisation d'arbres (exemples)

étiquette=clé ou key

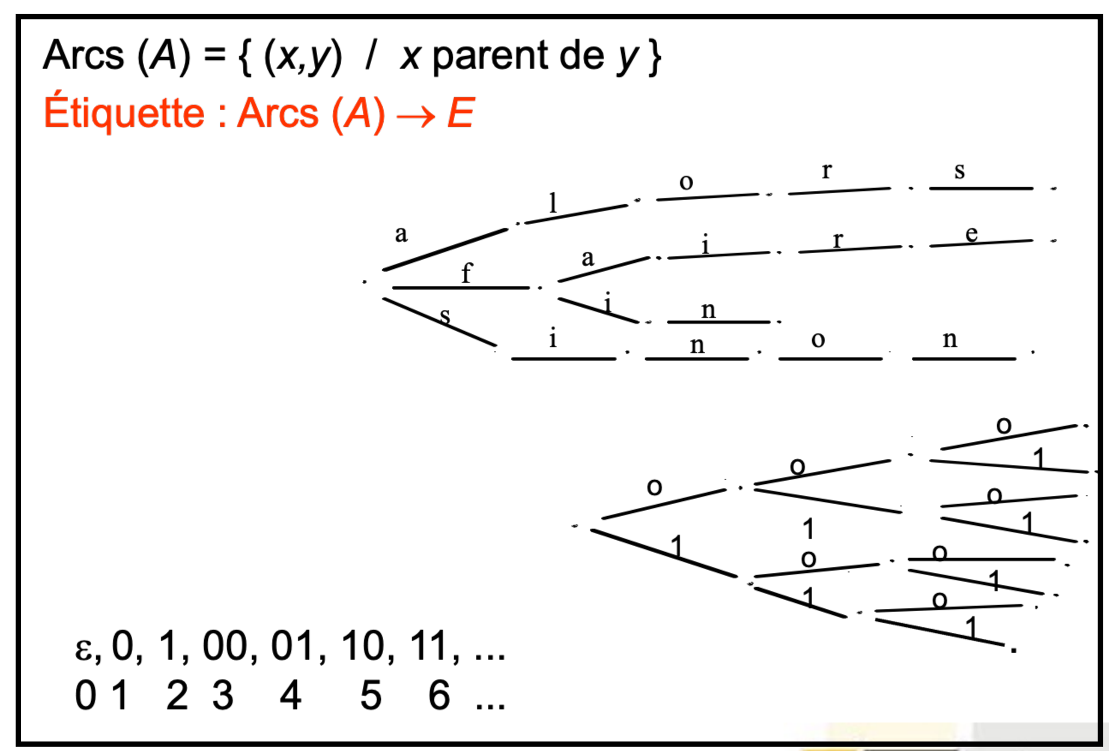
Arbre étiqueté







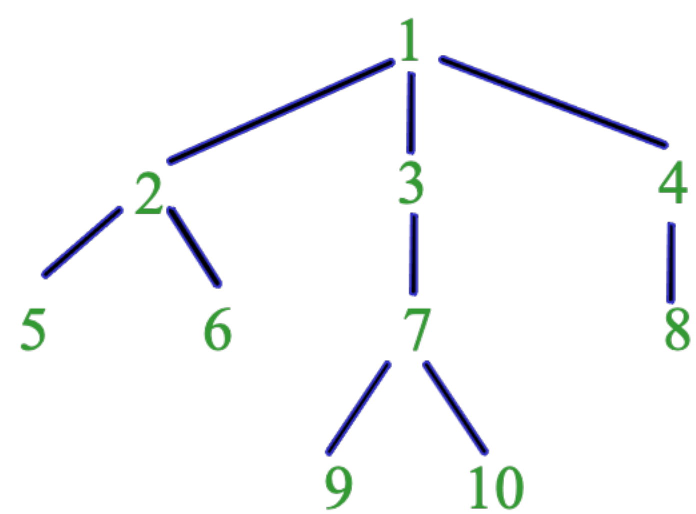
Arbre de quaternaire (Quad tree)



Arbres lexicographiques

Vocabulaire (1)

- La hauteur $h_A(x)$ d'un nœud x de arbre A est la longueur du plus long chemin de ce nœud aux feuilles qui en dépendent plus 1
  - C'est le nombre de nœuds du chemin
  - La hauteur d'un arbre h_A est la hauteur de sa racine
  - L'arbre vide a une hauteur 0



$h_A(8) = 0$

$h_A(7) = 1$

$h_A(3) = 2$

$h(A) = h_A(1) = 3$

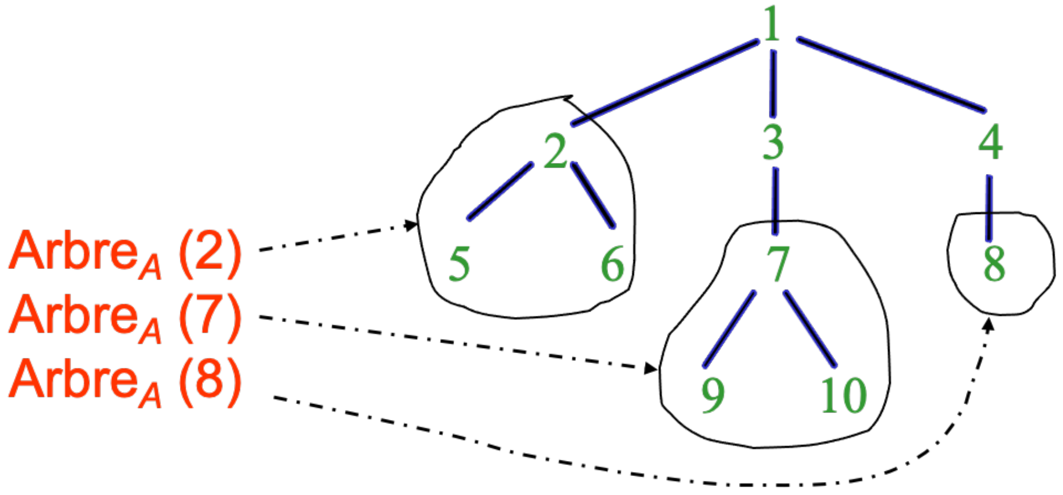
CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Vocabulaire (2)

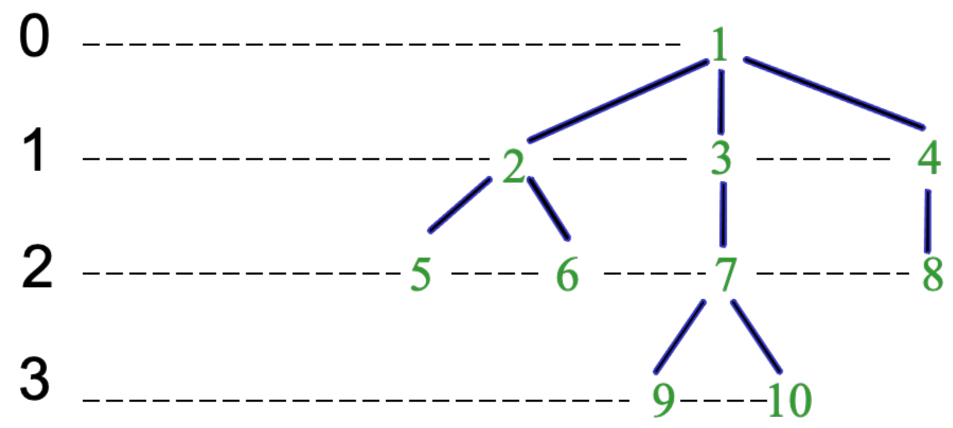
- Tout autre nœud est la racine d'un sous-arbre de l'arbre principal
- Un nœud qui n'est le point de départ d'aucun arc est appelé feuille (ou nœud externe)
- Pour les arbres binaires, on distinguera de façon visuelle le fils gauche du fils droit

A arbre *x* nœud de *A*
$Arbre_A(x)$ = sous-arbre de *A* qui a racine *x*
$h_A(x) = h(Arbre_A(x))$
$A = Arbre_A(racine(A))$

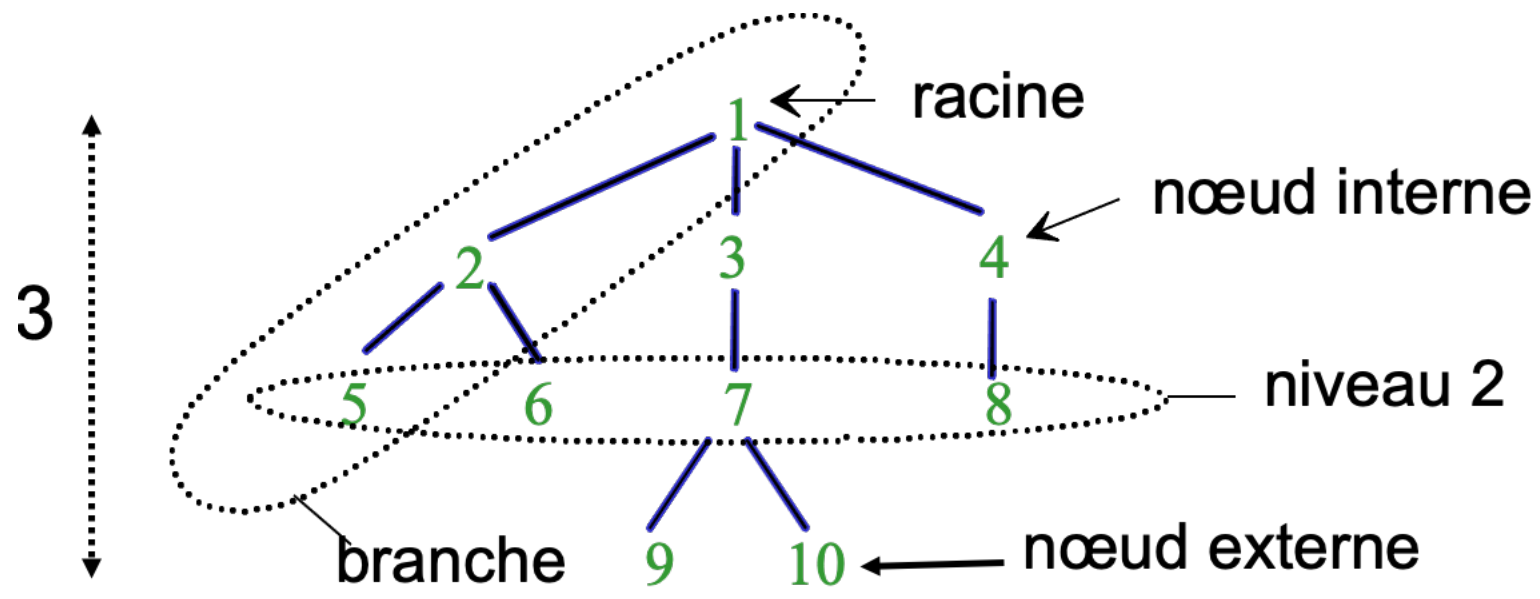


Vocabulaire (3)

- La profondeur (niveau) d’un nœud est le nombre de nœuds du chemin qui va de la racine à ce nœud
  - La racine d’un arbre est à une profondeur 0
  - La profondeur d’un nœud est égale à la profondeur de son père plus 1
  - Si un nœud est à une profondeur p, tous ses fils sont à une profondeur p+1
- Tous les nœuds d’un arbre de même profondeur sont au même niveau



Terminologie



2, 3, 4	enfants de 1
3, 4	frères de 2
1, 3, 7	ancêtres de 7
7, 9, 10	descendants de 7

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

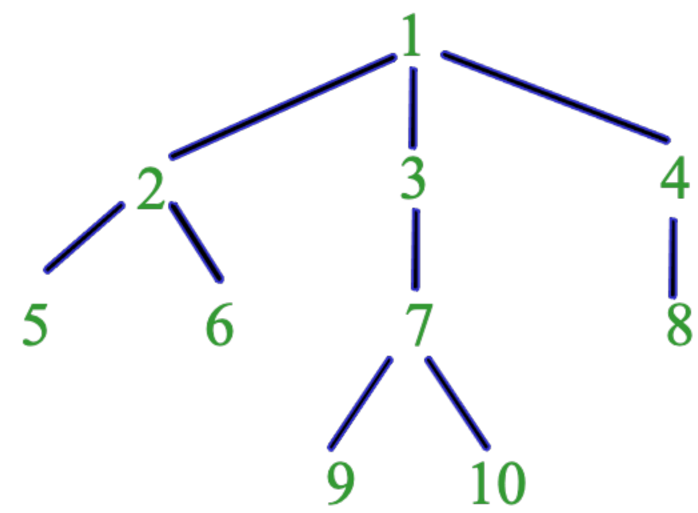
Parcours d'un arbre

- Fonction utile pour l 'exploration des arbres
- Parcours(A): arbre -> liste de ses nœuds (arbre vide -> liste vide)
- Deux types :

1. Parcours en profondeur (Depth-first search, DFS)
    - préfixe, postfixe (suffixe), infixe (symétrique)
    - parcours branche après branche
2. Parcours en largeur (ou hiérarchique) (Breadth-first search, BFS)
    - parcours niveau après niveau

Parcours en profondeur (DFS)

- Préfixe :
  - On affiche la racine, puis le sous-arbre gauche, puis le sous-arbre droit
- Suffixe:
  - On affiche le sous-arbre gauche, puis le sous-arbre droit, puis la racine
- symétrique:
  - On affiche le sous-arbre gauche, puis la racine, puis le sous-arbre droit



Arbre non vide $A = (r, A1, A2, \ldots, Ak)$

Parcours préfixe

$P(A) = (r).P(A1). \ldots .P(Ak)$

(1, 2, 5, 6, 3, 7, 9, 10, 4, 8)

Parcours suffixe

$S(A) = S(A1). \ldots .S(Ak).(r)$

(5, 6, 2, 9, 10, 7, 3, 8, 4, 1)

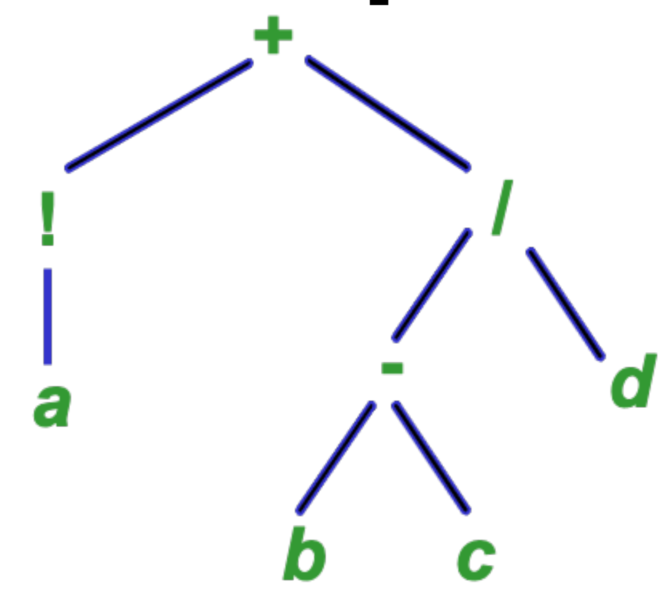
Parcours infixe

$I(A) = I(A1).(r).I(A2). \ldots .I(Ak)$

(5, 2, 6, 1, 9, 7, 10, 3, 8, 4)

Utilisation d'infixe: exemple des expressions arithmétiques

Arbre syntaxique de $a! + \frac{b-c}{d}$



Parcours préfixe (preorder en anglais)

+ ! a / - b c d

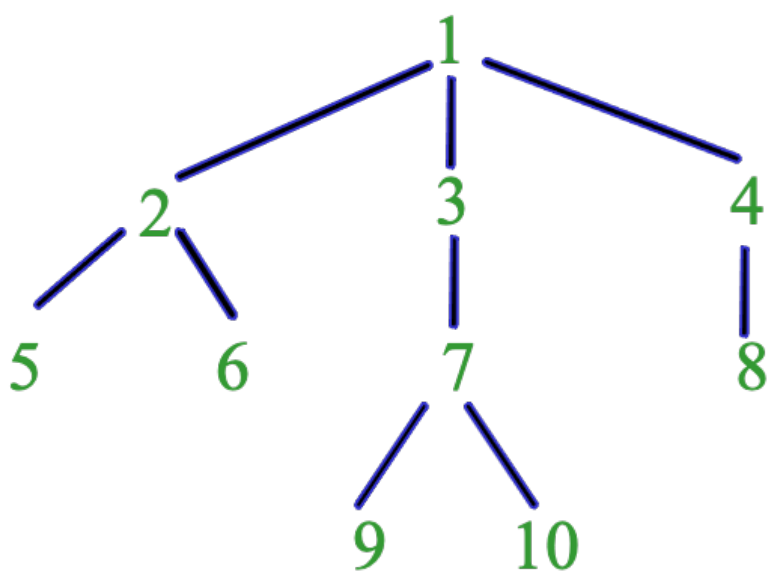
Parcours postfixe (postorder en anglais)

a ! b c - d / +

Parcours symétrique (priorités et parenthèses)

(a !) + ((b - c) / d)

Parcours en largeur (BFS)



Arbre non vide A = (r, A1, A2, …, Ak)

Parcours en largeur (hiérarchique)

$H(A) = (r, x_1, \ldots, x_i, x_{i+1}, \ldots x_j, x_{j+1}, \ldots, x_n)$

nœuds de niveau 0, 1, 2, ...

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

Type abstrait d'un arbre

- Ensemble

  Arbres plantaires de nœuds étiquetés

- Axiomes principaux

  Racine(A) et Enfants(A) définis ssi. A non vide

  Racine (Cons(r, L)) = r

  Enfants(Cons(r, L)) = L

- Opérations

  Arbre-vide : -> arbre

  Racine : arbre -> nœud

  Enfants : arbre -> liste d'arbres

  Cons : nœud x liste d'arbres -> arbre

  estVide : arbre -> booléen

  Elt : nœud -> élément

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Implantation (par tableau)

Représentation de la relation P

  tableau des parents

Avantages

  représentation simple

  parcours faciles vers la racine

  économique en mémoire

Inconvénients

  accès difficiles aux nœuds depuis la racine

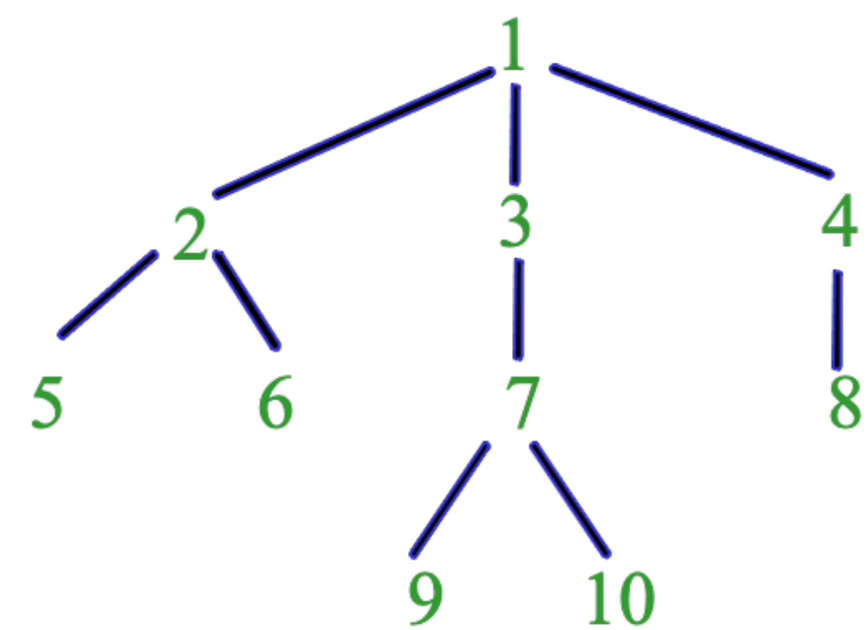


Tableau de parents P

-	1	1	1	2	2	3	4	7	7
1	2	3	4	5	6	7	8	9	10

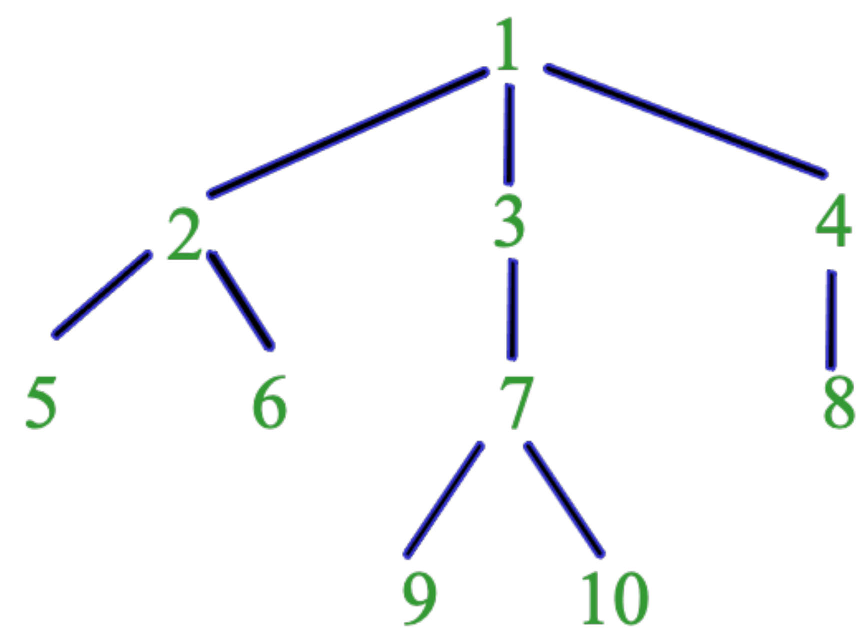
Implantation (par chaînage)

Représentation des listes de sous-arbres par chaînage

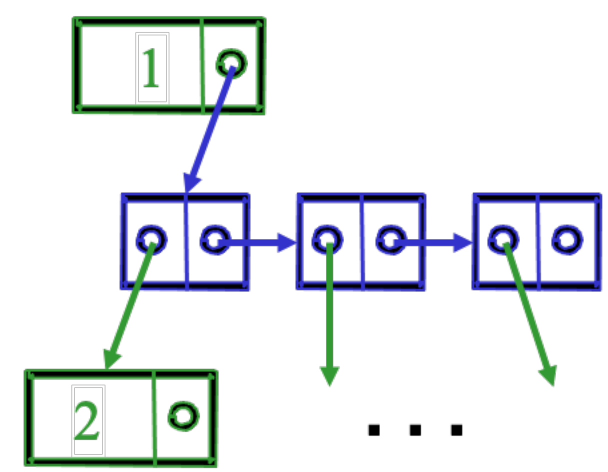
Deux types de pointeurs:

pointeur d'arcs et

pointeur de noeud



Arbre souvent identifié à l 'adresse de sa racine (comme pour un tableau en C)



Avantages

accès faciles depuis la racine

correspond à la définition récursive

Inconvénients

parcours difficiles vers la racine

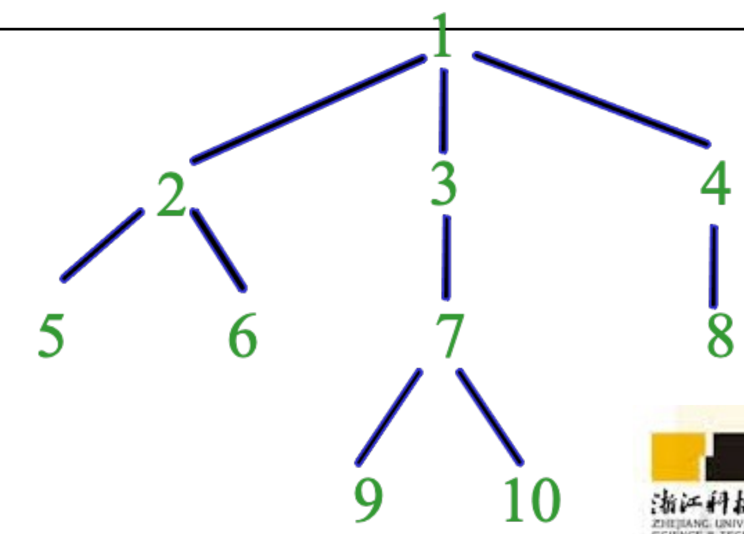
relativement gourmand en mémoire

Fonctions préfixe et suffixe (récursives)

```
fonction préfixe (A arbre) : liste de nœuds;
début
    si A = arbre vide alors
        retour (suite vide)
    sinon {
        L<-( Racine (A) );
        pour B <- premier au dernier élément
de Enfants(A) faire
            L <- L.Préfixe(B);
        retour (L);
    }
fin
```

```
fonction postfixe (A arbre) : liste de
nœuds;
début
    si A = arbre vide alors retour (suite
vide)
    sinon {
        L <- ( );
        pour B <- premier au dernier élément
de Enfants(A) faire
            L <- L.postfixe(B);
         L <- L.(racine(A) );
        retour (L);
    }
fin
```

Temps d 'exécution : *O(n)*
sur un arbre à n nœuds représenté par pointeurs



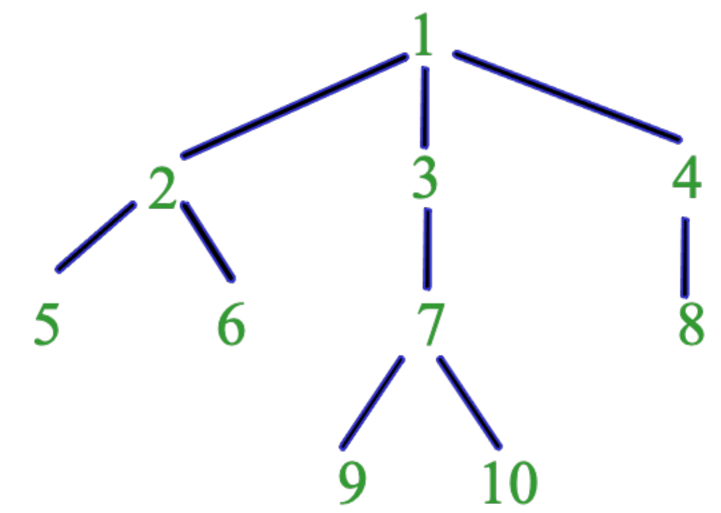
Fonction préfixe itérative

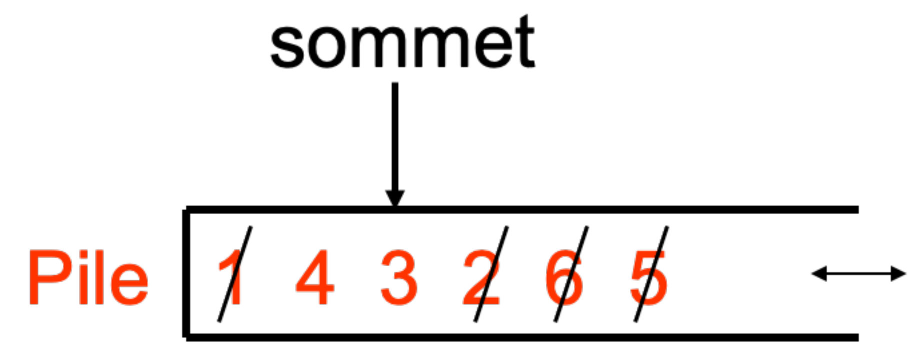
```
fonction Préfixe (A arbre) : liste de nœuds ;
début
    L <- ( ) ;
    Pile <- Empiler( Pile-vide, A ) ;
    tant que non Vide( Pile ) faire {
        A’ <- sommet ( Pile ) ;
        Pile <- Dépiler ( Pile ) ;
        si A’ non vide alors {
            L <- L.(racine(A’) ) ;
            pour B <- dernier au premier élément de Enfants(A’) faire
                Pile <- Empiler ( Pile, B ) ;
        }
    }
    retour ( L ) ;
fin
```

Temps d’exécution *O(n)* comme version récursive
si, en plus, bonne implémentation de la pile

Exemple (Préfixe, itérative)

```
fonction Préfixe (A arbre) : liste de nœuds ;
début
    L <- ( ) ;
    Pile <- Empiler( Pile-vide, A ) ;
    tant que non Vide( Pile ) faire {
        A’ <- sommet ( Pile ) ;
        Pile <- Dépiler ( Pile ) ;
        si A’ non vide alors {
            L <- L.(racine(A’) ) ;
            pour B <- dernier au premier élément de
Enfants(A’) faire
                 Pile <- Empiler ( Pile, B ) ;
        }
    }
    retour ( L ) ;
fin
```





Liste *L* = (1, 2, 5, 6, .

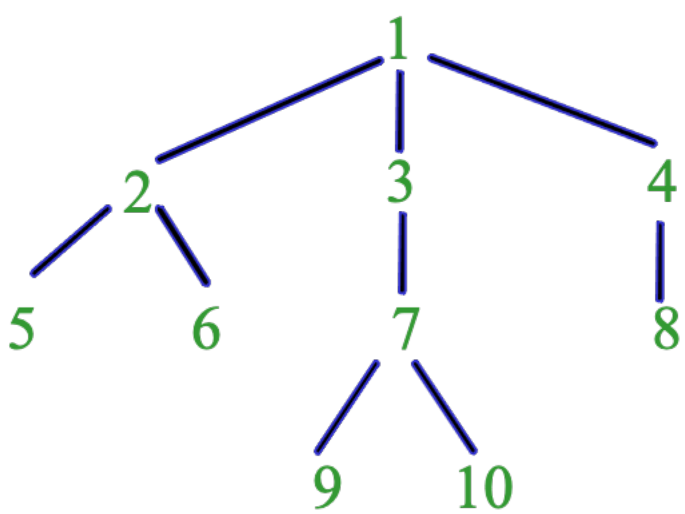
Parcours en largeur par itération

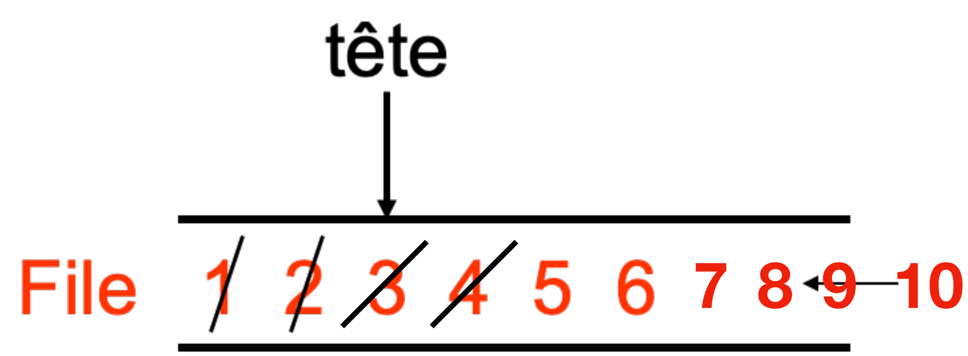
```
fonction Largeur (A arbre) : liste de nœuds ;
début
 L <- ( ) ;
 File <- Enfiler( File-vide, A ) ;
 tant que non Vide( File ) faire {
   A’ <- tête ( File ) ;
   File <- Enlever ( File ) ;
   si A’ non vide alors {
    L <- L.(racine(A’) ) ;
    pour B  <-  premier au dernier élément de Enfants(A’) faire
      File  <-  Ajouter ( File, B ) ;
       }
   }
   retour ( L ) ;
fin
```

Temps d’exécution $O(n)$ si bonne implémentation de la file
pas de version récursive

Exemple (en largeur, itérative)

```
fonction Largeur (A arbre) : liste de nœuds ;
début
  L <- ( ) ;
  File <- Enfiler( File-vide, A ) ;
  tant que non Vide( File ) faire {
    A’ <- tête ( File ) ;
    File <- Enlever ( File ) ;
    si A’ non vide alors {
      L <- L.(racine(A’) ) ;
      pour B  <-  premier au dernier élément de
      Enfants(A’) faire
        File  <-  Ajouter ( File, B ) ;
          }
     }
     retour ( L ) ;
fin
```





Liste *L* = (1, 2, .

Arbre k-aires (*k*-ary/k-way tree)

Arbre k-aire : tout nœud possède *k* sous-arbres (vides ou non), *k* fixé

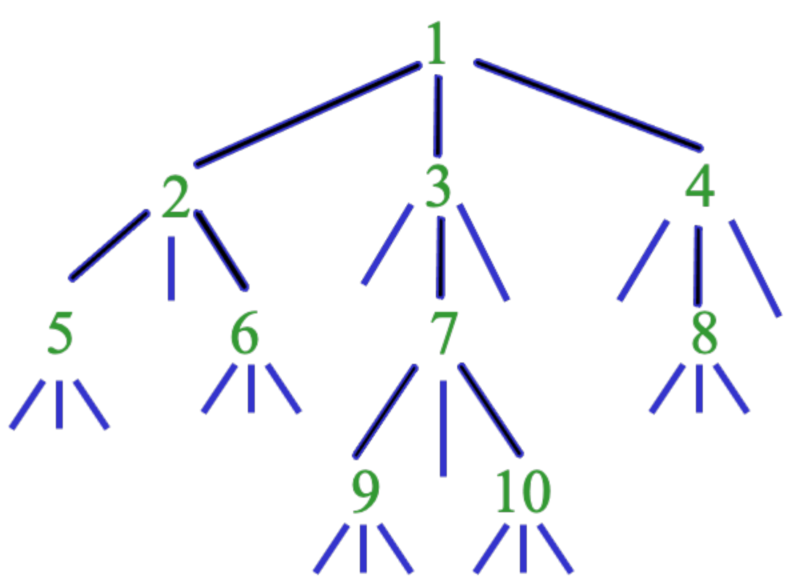
$A = \begin{cases} \Lambda & \text{arbre vide ou} \\ (r, A_1, \ldots, A_k) & \end{cases}$

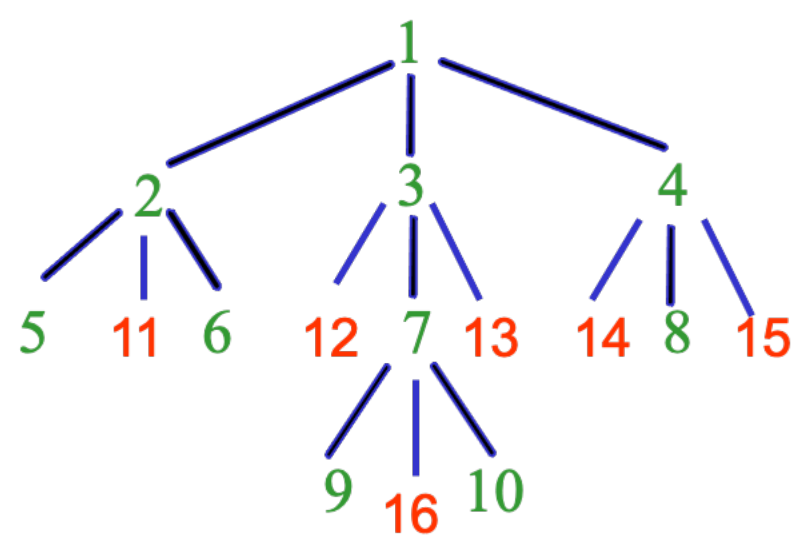
r élément, $A_1, \ldots, A_k$ arbres k-aires

Nœuds $(A) = \{r\} \cup (\cup$ Nœuds $(A_i))$

unions disjointes

Arbre k-aire complet : tout nœud interne possède *k* fils





Arbre k-aires n'est pas efficace, car plupart des nœuds sont vide en pratique.
Souvent, un arbre k-aires est converti vers un arbre binaire.

Type abstrait d'arbre binaire

- **Ensemble**

  Arbres binaires étiquetés

- **Opérations**

  Arbre-vide : -> arbre

  Racine : arbre -> nœud

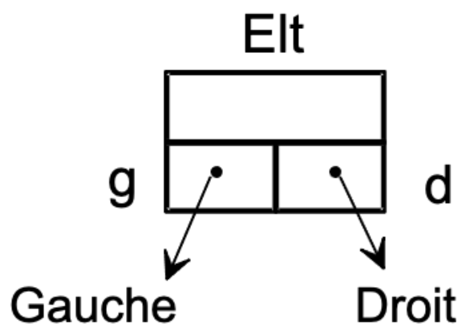
  Gauche, Droit : arbre -> arbre

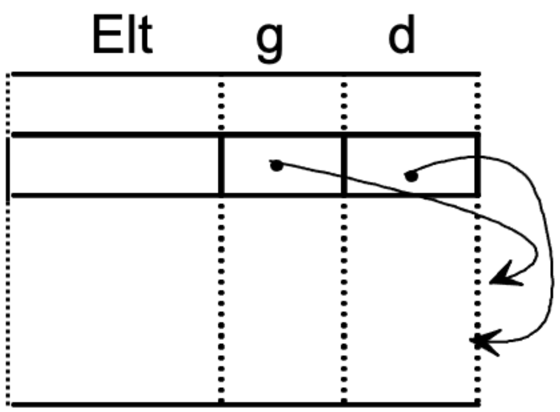
  Cons : nœud x arbre x arbre -> arbre

  Elt : nœud -> élément

  Vide : arbre -> booléen

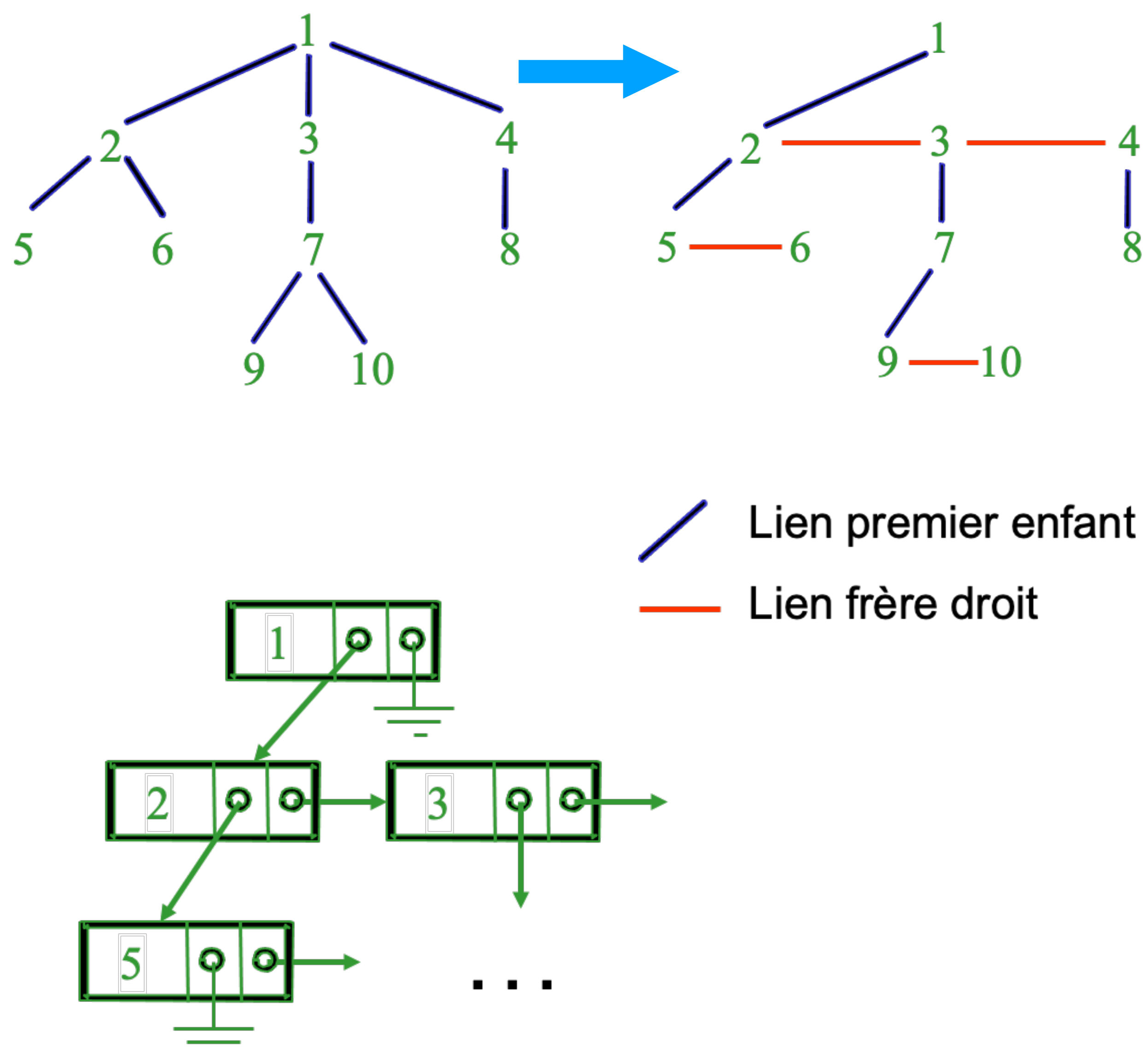
- Implantation par pointeurs ou curseurs



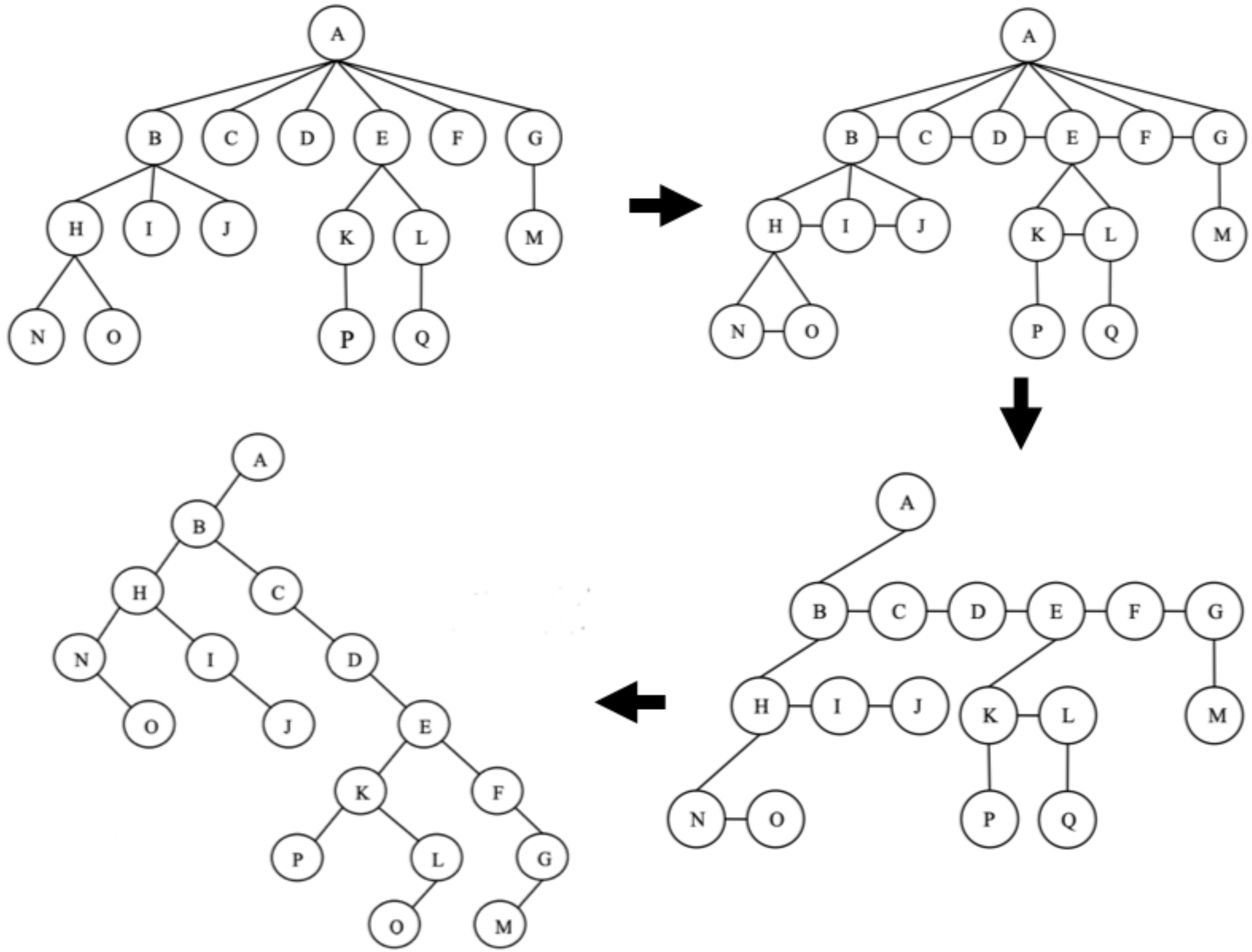


Binarisation

1. Relier tous les nœuds enfants immédiats d'un nœud parent donné afin de former une liste de liens.
2. Ensuite, conserver le lien entre le parent et le premier enfant (i.e. le plus à gauche) et supprimons tous les autres liens vers le reste des enfants.
3. Répéter sur tous les enfants jusqu'à ce que tous les nœuds soient traités.



Binarisation



Arbre binaire complet

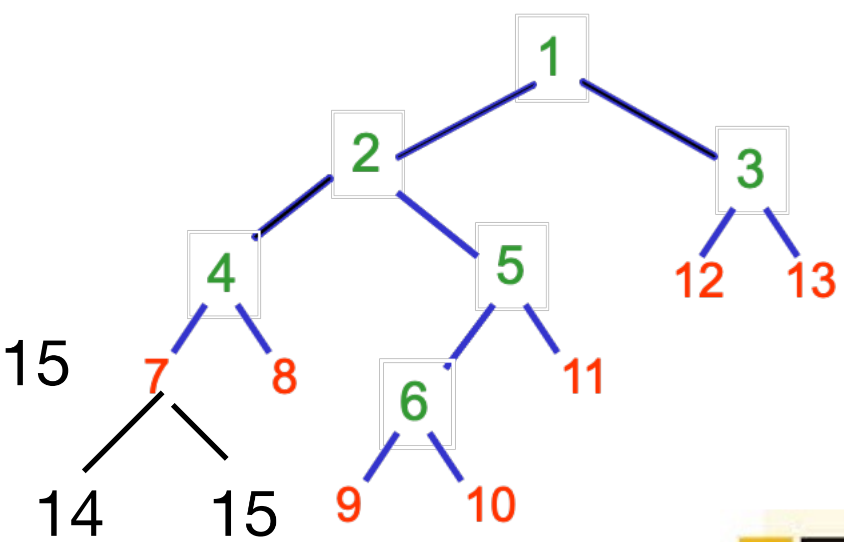
- Arbre binaire complet (feuillu) : deux types de nœuds, internes ou feuilles
  - tout nœud interne possède deux enfants ;
  - toute feuille est un nœud externe.

$$A = \begin{cases} (\, f\,) & f \text{ de type feuille} \\ (r, G, D\,) & r \text{ de type interne , } G, D \text{ arbre binaires feuillus} \end{cases}$$

Nœuds (A) = { r } ∪ Nœuds (G) ∪ Nœuds (D)   unions disjointes

Nœuds internes : 1, 2, 3, 4, 5, 6 **7**

Feuilles : ~~7~~, 8, 9, 10, 11, 12, 13 14, 15



Taille des arbres feuillus

- Arbre complet (feuillu) : nombre de feuilles = nombre de nœuds internes + 1
- Récurrence sur le nombre de nœuds de l 'arbre A :
  - si un seul nœud, c 'est une feuille ; propriété satisfaite.
  - sinon, il existe un nœud interne dont les enfants sont des feuilles, i.e. un sous-arbre *(x, g, d)* où *g, d* sont des feuilles. Soit *B* obtenu de *A* en remplaçant *(x, g, d)* par une feuille *f*. *B* est un arbre feuillu de plus petite taille ; par récurrence l'égalité est satisfaite sur *B*; donc aussi sur *A* qui possède un nœud interne et une feuille de plus. CQFD.
-

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

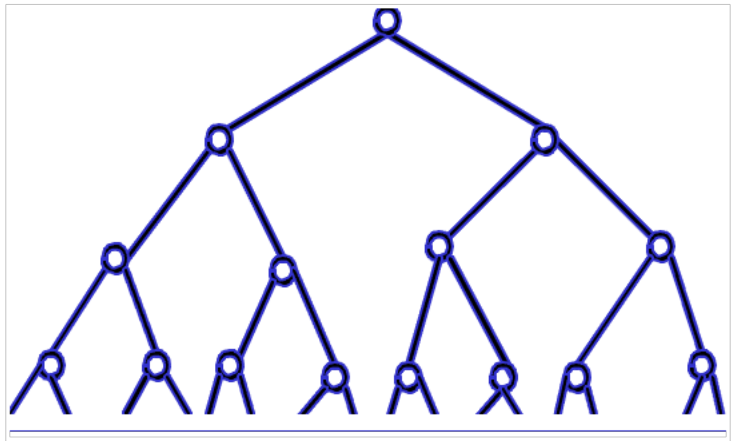
Mesures des arbres binaires

Arbre binaire à hauteur h et n nœuds

Arbre plein

2^i nœuds au niveau i

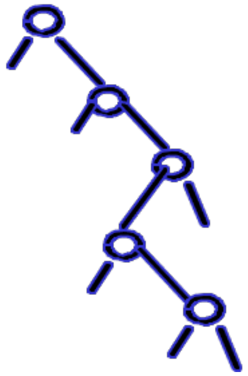
$n = 2^{h+1} - 1$



Arbre filiforme

1 nœud par niveau

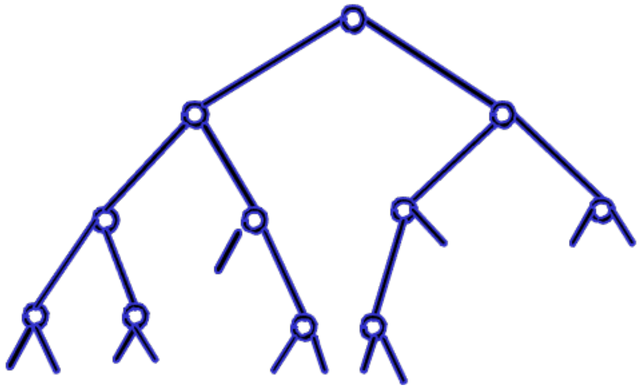
$n = h + 1$



Arbre binaire

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

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



CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

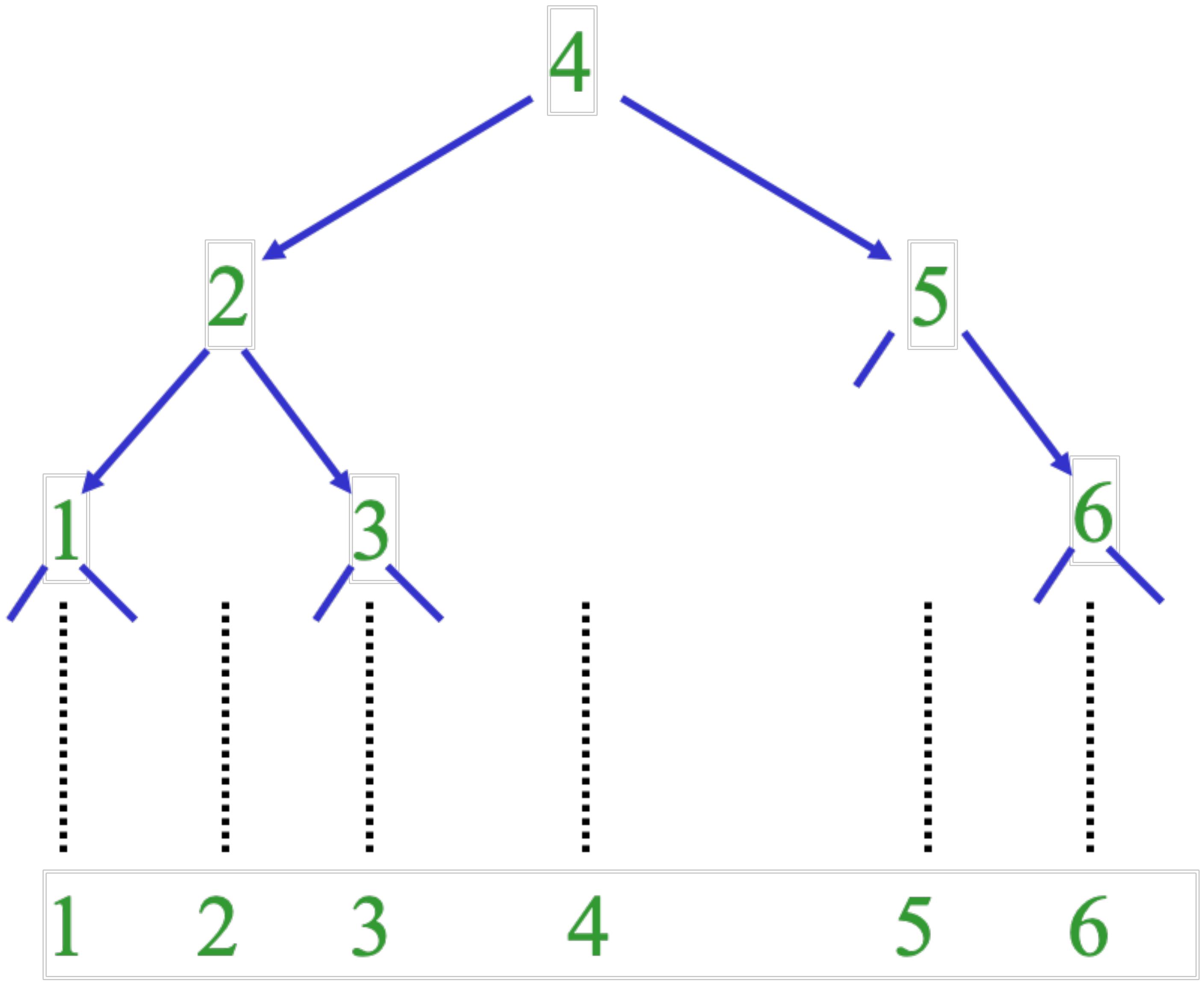
Arbre binaire de recherche (ABR)

- Soit A un arbre binaire avec les nœuds étiquetés par les éléments, A est un arbre binaire de recherche
  - ssi. l'ordre symétrique donne une liste croissante des éléments
  - ssi. en tout nœud p de A, $Elt(G(p)) < Elt(p) < Elt(D(p))$
  - ssi $A = Arbre\_vide$ ou $A = (r, G, D)$ avec G, D arbres binaires de recherche et $Elt(G) < Elt\ (r) < Elt(D)$

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Exemple

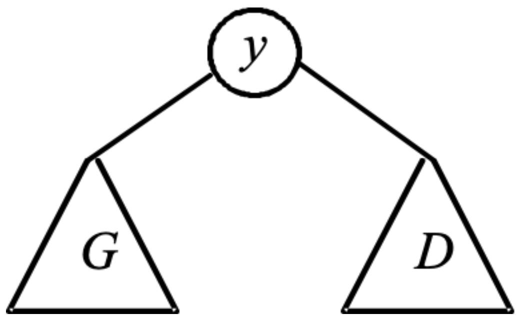


Recherche récursive

Recherche d'une valeur *x* à partir d'un nœud *A* d'un ABR :
Elément (A, x) = vrai ssi. *x* est étiquette d'un noeud de *A* (abr)

Élément (A, x) =

faux	si A vide
vrai	si x = y
Elément (G(A), x)	si x < y
Elément (D(A), x)	si x > y



Calcul en O(Hauteur(A))
donc $O(\log_2 n)$ en arbre équilibré

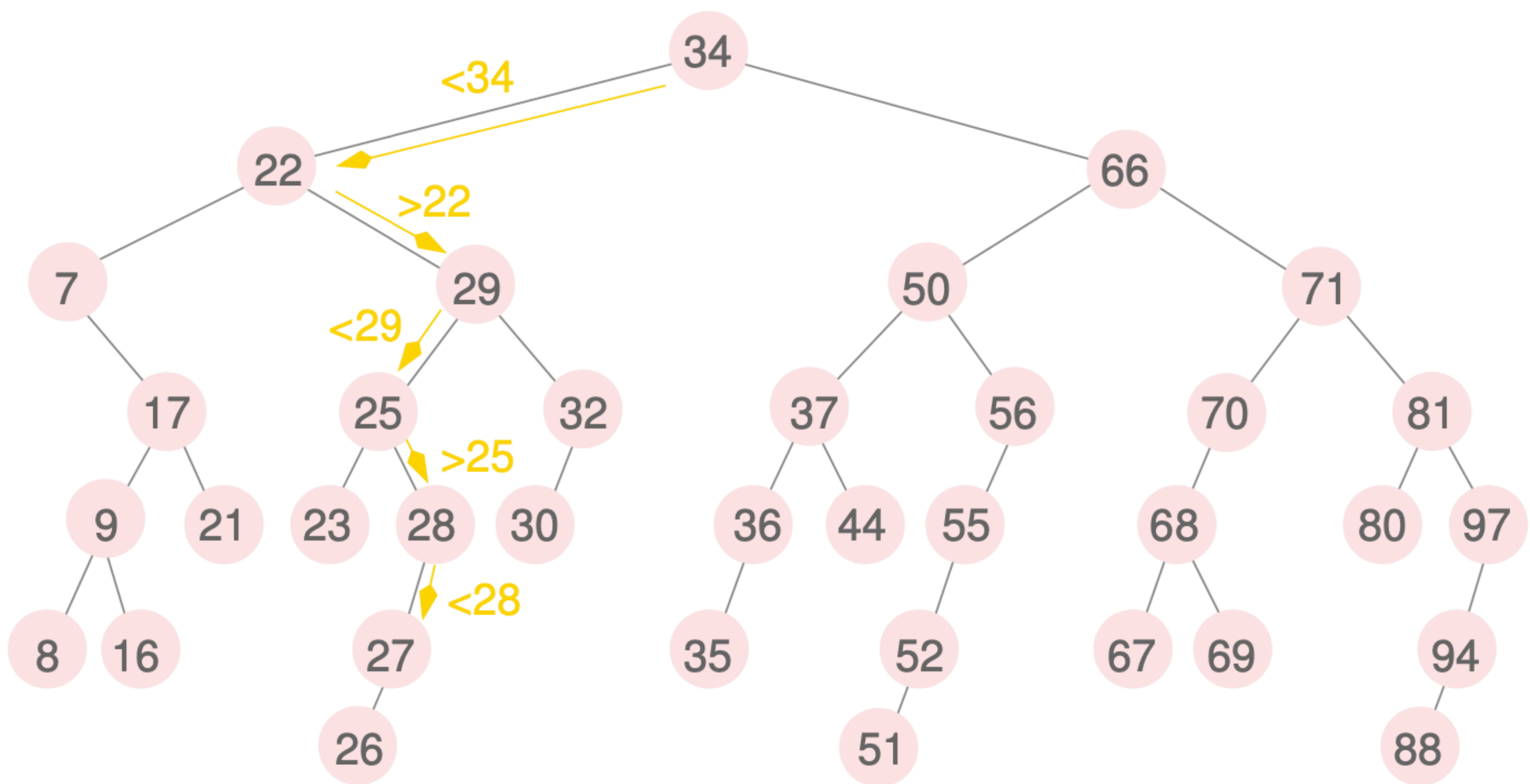
```
ABRRecherche(A abr,x élément)
Début
  Si A = Null Alors
    Retourner Faux
  Sinon Si x = Elt(A) Alors
    Retourner A
  Sinon Si x < Elt(A) Alors
    Retourner ABRRecherche(G(A),x)
  Sinon
    Retourner ABRRecherche(D(A),x)
  FinSi
Fin
```

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Exemple

recherche de la valeur 27

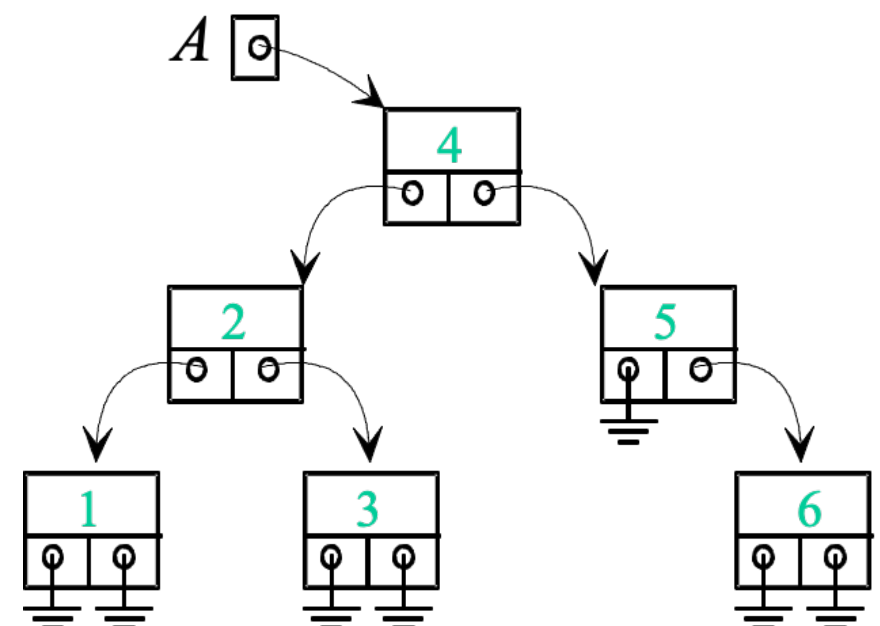


CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

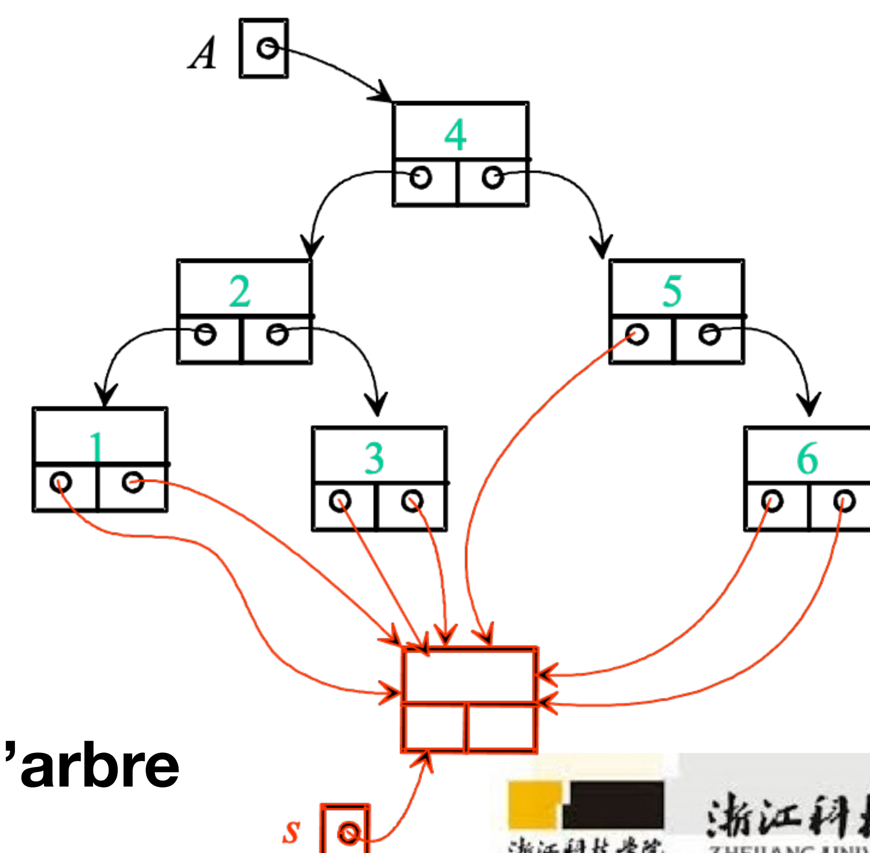
Recherche itérative

```
fonction ABRRecherche (A abr, x élément) : adresse
début
  p <- Racine (A) ;
  tant que  p ≠ nil et  x ≠ Elt (p)  faire
    si x < Elt (p)  alors  p <- p->g ;
    sinon p <- p->d ;
  retour (p) ;
fin
```



Avec sentinelle:

```
fonction Place (A abr, x élément) : adresse ;
/* version itérative, avec sentinelle s */
début
    p  <- Racine( A) ; s <- Elt <- x ;
    tant que  x ≠ Elt (p)  faire
        si x < Elt (p) alors p <-  p->g ;
        sinon p <- p->d ;
    si p = s alors  retour (nil)
    sinon  retour (p)
fin
```



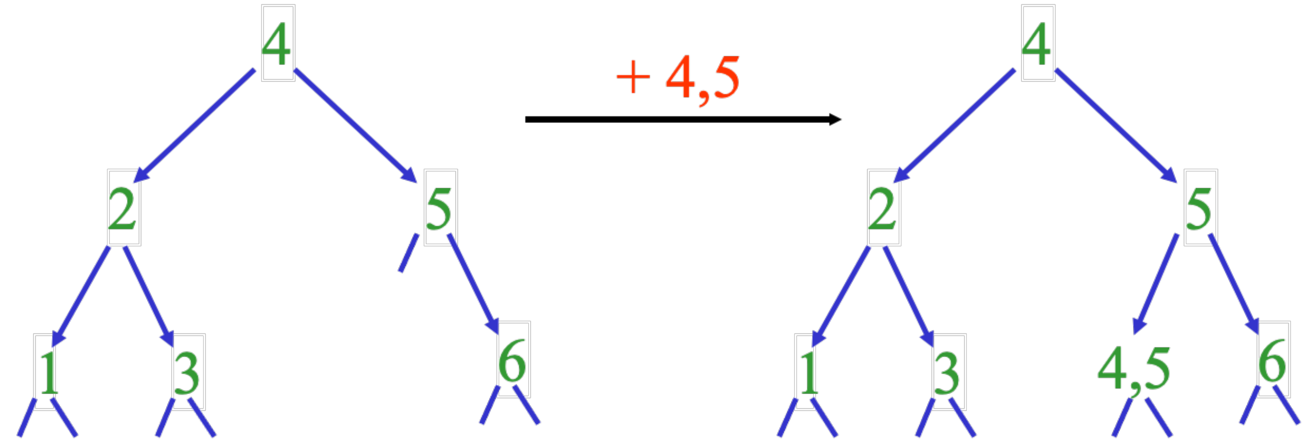
Avantage de sentinelle: éviter la manip directe sur l'arbre

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Ajout dans ABR

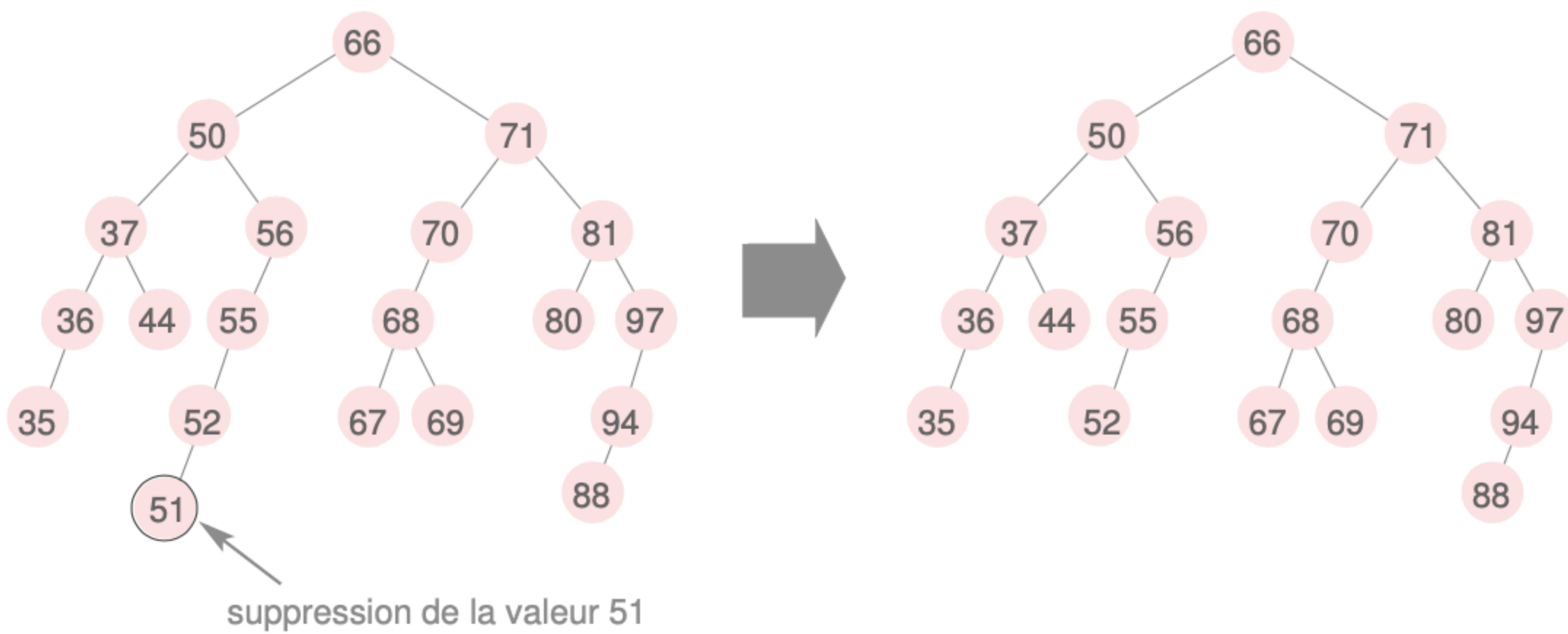
$$A + \{x\} = \begin{cases} (x, \wedge, \wedge) & \text{si } A = \wedge \text{ , arbre vide} \\ (r, G + \{x\}, D) & \text{si } x < \text{Elt}(r) \\ (r, G, D + \{x\}) & \text{si } x > \text{Elt}(r) \\ A & \text{sinon} \end{cases}$$



Le principe est le même que pour la recherche.
Un nouveau noeud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arrêtée.

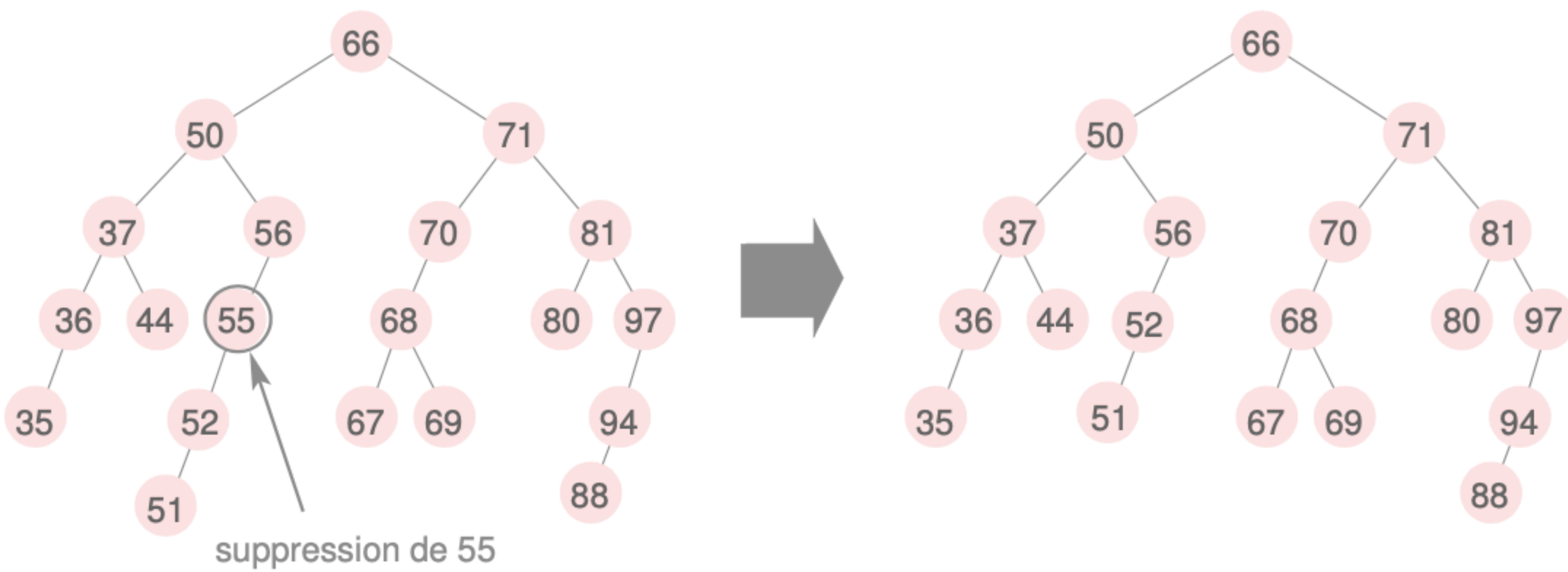
```
fonction Ajouter (A abr, x élément) : abr ;
début
  si  Vide(A)  alors
    retour (arbre à 1 seul nœud d'étiquette x) ;
  sinon si x < Elt (Racine (A))  alors
    retour (Racine (A), Ajouter(G(A), x), D(A)) ;
  sinon si  x >  Elt (Racine (A))  alors
    retour (Racine (A), G(A), Ajouter (D(A), x)) ;
  sinon  retour (A) ;     /* x =  Elt (Racine (A)), rien à faire */
fin
```

Suppression d'un élément dans un ABR (1)



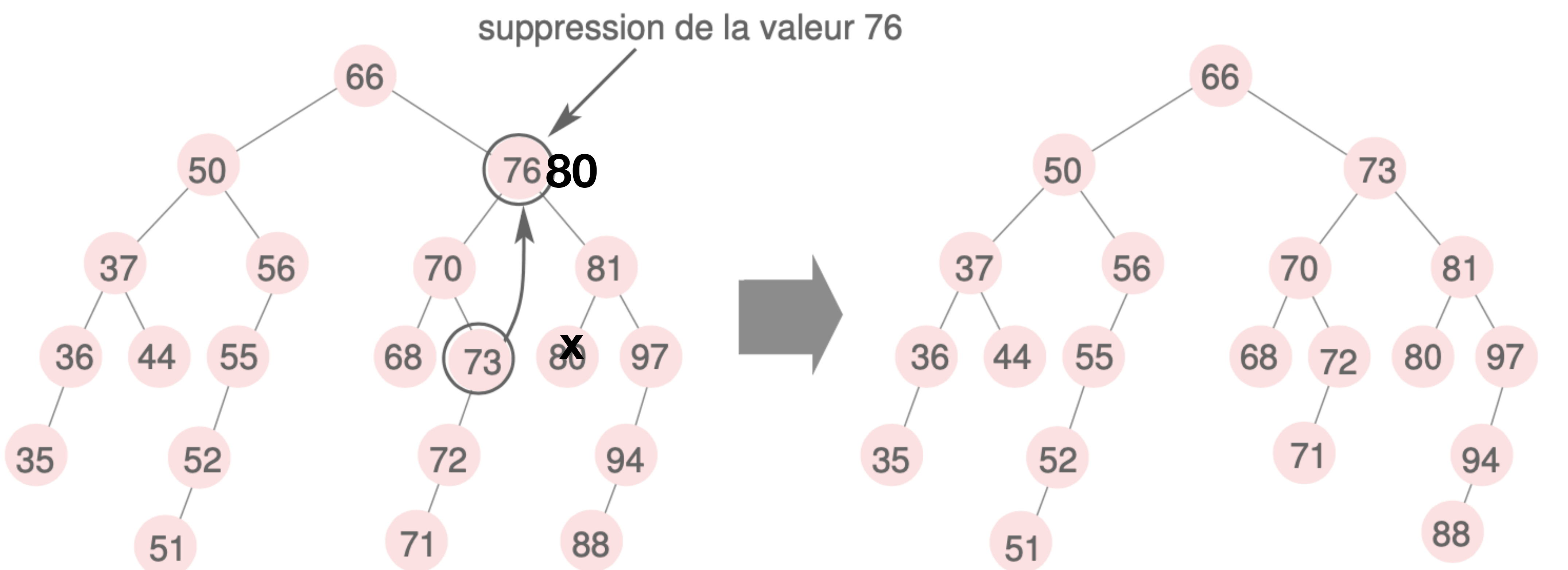
Cas 1 : le noeud à supprimer n'a pas de fils, c'est une feuille. Il suffit de décrocher le noeud de l'arbre, c'est-à-dire de l'enlever en modifiant le lien du père, si il existe, vers ce fils. Si le père n'existe pas l'arbre devient l'arbre vide.

Suppression d'un élément dans un ABR (2)



Cas 2 : le noeud à supprimer a un fils et un seul. Le noeud est décroché de l'arbre comme dans le cas 1. Il est remplacé par son fils unique dans le noeud père, si ce père existe. Sinon l'arbre est réduit au fils unique du noeud supprimé.

Suppression d'un élément dans un ABR (3)



Cas 3 : le noeud à supprimer *p* a deux fils. Soit *q* le noeud de son sous-arbre gauche qui a la valeur la plus grande (on peut prendre indifféremment le noeud de son sous-arbre droit de valeur la plus petite). Il suffit de recopier la valeur de *q* dans le noeud *p* et de décrocher le noeud *q*. Puisque le noeud *q* a la valeur la plus grande dans le fils gauche, il n'a donc pas de fils droit, et peut être décroché comme on l'a fait dans les cas 1 et 2.

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Suppression d'un élément dans un ABR (4)

$A = (r, G, D)$

$$A - \{x\} = \begin{cases} (r, G - \{x\}, D) & \text{si } x < \mathrm{Elt}(r) \\ (r, G, D - \{x\}) & \text{si } x > \mathrm{Elt}(r) \\ D & \text{si } x = \mathrm{Elt}(r) \text{ et } G \text{ vide} \\ G & \text{si } x = \mathrm{Elt}(r) \text{ et } D \text{ vide} \\ (r, G - \{\mathrm{MAX}(G)\}, D) & \text{sinon} \\ \text{avec } \mathrm{Elt}(r) = \mathrm{MAX}(G) & \end{cases}$$

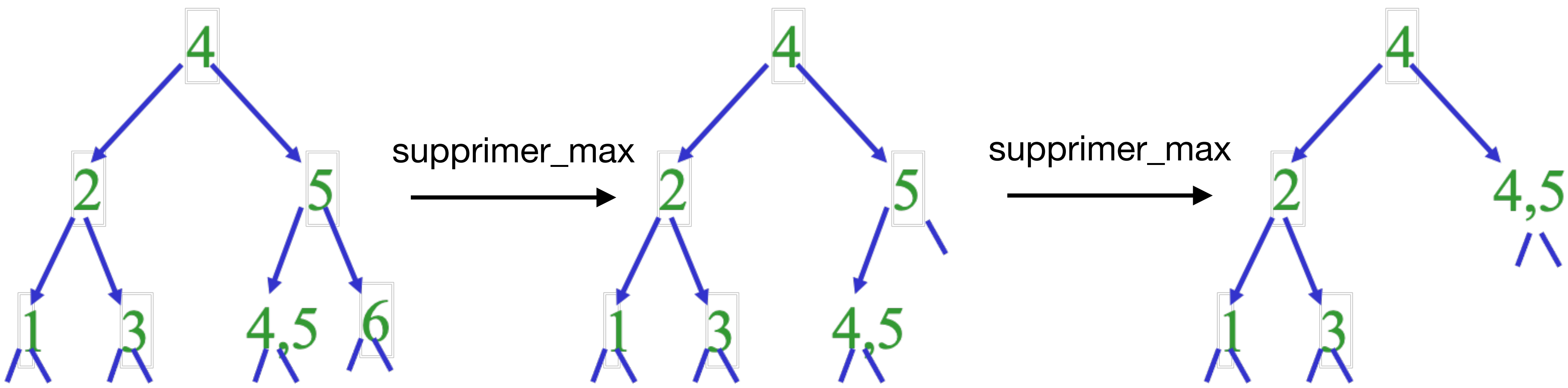
```
fonction  Enlever (A abr, x élément) : abr ;
début
  si Vide (A) alors
    retour (A) ;
  sinon si  x < Elt (Racine (A)) alors
    retour (Racine(A), Enlever (G(A), x), D(A))
  sinon si  x >  Elt (Racine (A) ) alors
    retour (Racine (A), G(A), Enlever (D(A),x) )
  sinon si  Vide (G(A)) alors
    retour (D(A))
  sinon si  Vide (D(A)) alors
    retour (G(A))
  sinon{
   Elt (Racine (A)) <- MAX (G(A))  ;
   retour ( Racine (A), supprimer_max ( G(A) ), D(A) ) ;}
  finsi
fin
```

CERGY PARIS UNIVERSITÉ

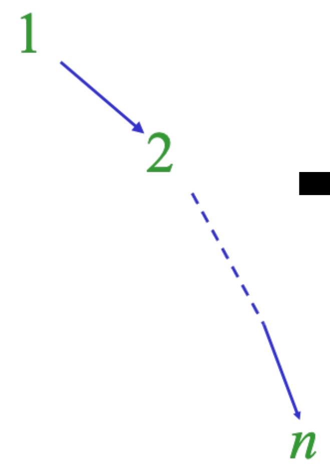
浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Suppression d'un élément dans un ABR (5)

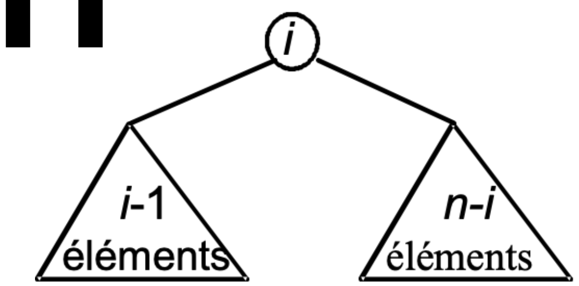
supprimer_max




```
fonction supprimer_max (A abr) : abr :
début
  si vide ( D (A) )  alors
    retour ( G (A) )
  sinon
    retour ( Racine(A), G (A), supprimer_max ( D (A)) ) ;
  finsi
fin
```



Temps d'insertion



Cas pire:
Insertions successives de 1, 2, ..., *n* dans l'arbre vide

Complexité d'insérer un nœud: n
Donc pour n nœud:

$0 + 1 + \ldots + n - 1 = \frac{n(n-1)}{2}$

Cas moyen:
Insertion d'un nœud : $\log_2 n$
Toutes permutations de (1, 2, ..., *n*) <u>équiprobables</u>.
P (*n*) = moyenne du nombre de noeuds par chemin
= moyenne des 1 + hauteur (*i*)

$$h_{arbre} = 1 + \max(h_{gauche}, h_{droit})$$

$$\mathrm{E}[h_n] = \frac{1}{n}\sum_{i=1}^{n}[1 + \max(h_{i-1}, h_{n-i})]$$

Définir, $Y_n = 2^{h_n}$ par la récurrence, on a $Y_n = 2 \times \max(Y_{i-1}, Y_{n-i})$

$$\mathrm{E}[Y_n] = \sum_{i=1}^{n}\frac{1}{n}\mathrm{E}[2 \times \max(Y_{i-1}, Y_{n-i})] = \frac{2}{n}\sum_{i=1}^{n}\mathrm{E}[\max(Y_{i-1}, Y_{n-i})]$$

Pour deux valeur non-négative, on a la relation :

$$\mathrm{E}[\max(X, Y)] \leq \mathrm{E}[\max(X, Y) + \min(X, Y)] = \mathrm{E}[X] + \mathrm{E}[Y]$$

$$\mathrm{E}[Y_n] \leq \frac{2}{n}\sum_{i=1}^{n}(\mathrm{E}[Y_{i-1}] + \mathrm{E}[Y_{n-i}]) = \frac{2}{n}\sum_{i=0}^{n-1}2\,\mathrm{E}[Y_i]$$

Par expérience:
$$\mathrm{E}[Y_n] \leq \frac{1}{4}\binom{n+3}{3}$$

$$2^{\mathrm{E}[X_n]} \leq \mathrm{E}[Y_n] \leq \frac{4}{n}\sum_{i=0}^{n-1}\mathrm{E}[Y_i] \leq \frac{1}{4}\binom{n+3}{3} = \frac{(n+3)(n+2)(n+1)}{24} \rightarrow \mathrm{E}[h_n] = O(\log n)$$

Tri par arbre

```
fonction TRI (L liste) : liste ;
début    A <- abr_vide ;
    pour  x <- premier au dernier élément de L  faire
        A   <-  Ajouter (A, x) ;
    retour (SYM (A)) ;
fin.

SYM: parcours infixe (symétrique) ("inorder" en anglais)
```

Tri par double changement de représentation

Temps :

maximal $O(|L| \log |L|)$ avec abr équilibré

moyen $O(|L| \log |L|)$ avec abr ordinaire

maximal $O(|L|^2)$ avec abr ordinaire

Arbre équilibré

AVL : arbre équilibrés en hauteur

- pour tout nœud *p*, $| h(D(p)) - h(G(p)) | \leq 1$

B-arbres

- toutes les feuilles sont au même niveau
- la racine est une feuille ou possède ³ 2 enfants
- tout autre nœud interne a entre a et b enfants
- et $2 \leq a < b \leq 2a - 1$

B-arbre ordinaire => $b = 2a - 1$

2.3-arbre

2.3.4-arbre ou arbre bicolore ou arbre rouge-noir

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Tas

Réalisé par arbres partiellement ordonnés (tas)

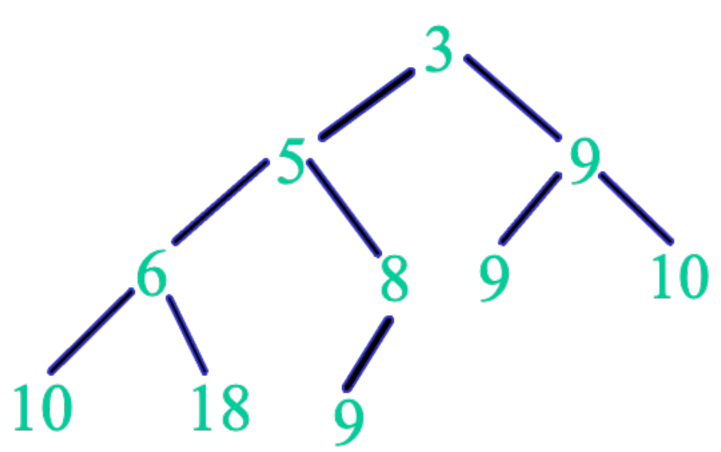
Qu’est-ce que c’est? (tas, ou arbre binaire tassé)

Un arbre binaire *A* est partiellement ordonné ssi. :

1. *A* est parfait (complet, sauf peut-être au dernier niveau) et
2. pour tout nœud p ≠ racine, Elt (p) ≥ Elt (Parent (p)) (ou inverse, pour l’ordre décroissant)

En anglais: heap (max heap, min heap)

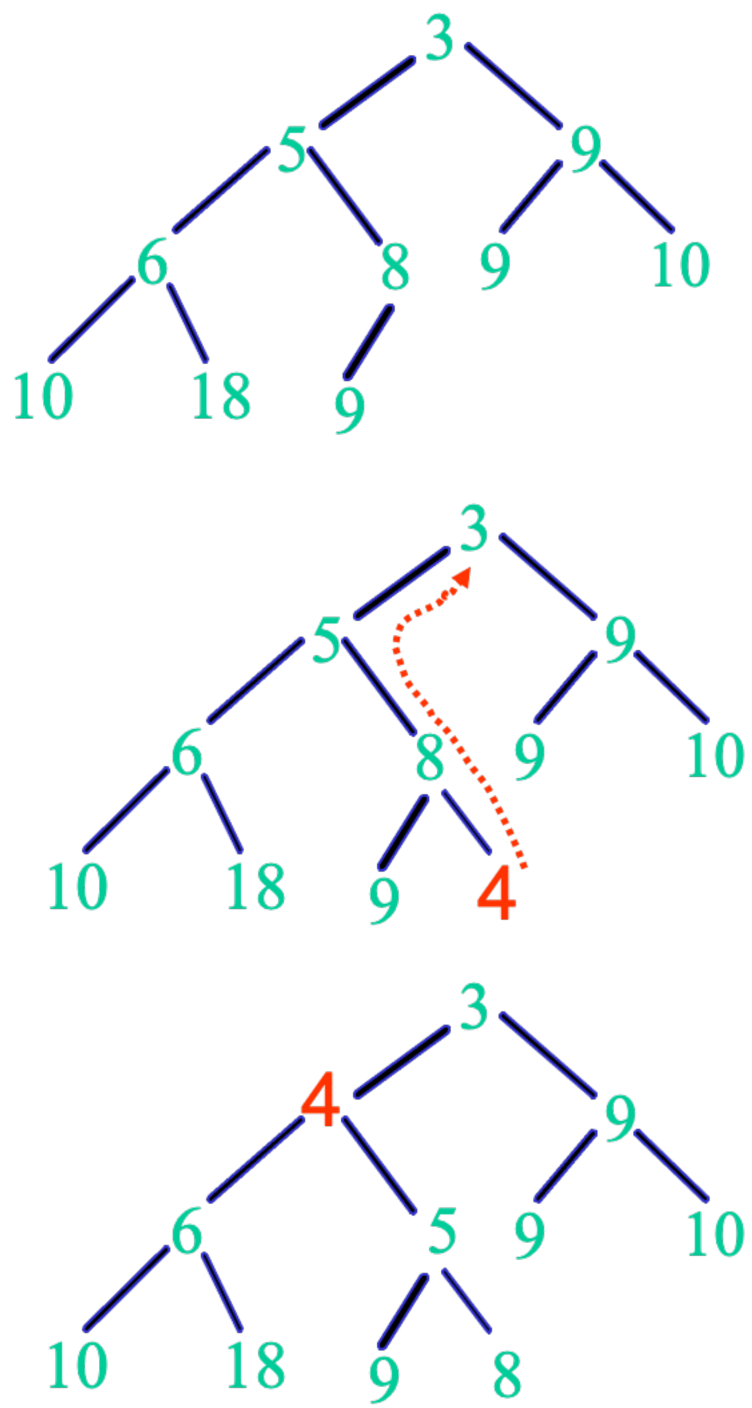
Hauteur(A) = $\lfloor \log_2 |A| \rfloor$



CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Ajout (insérer, insert)



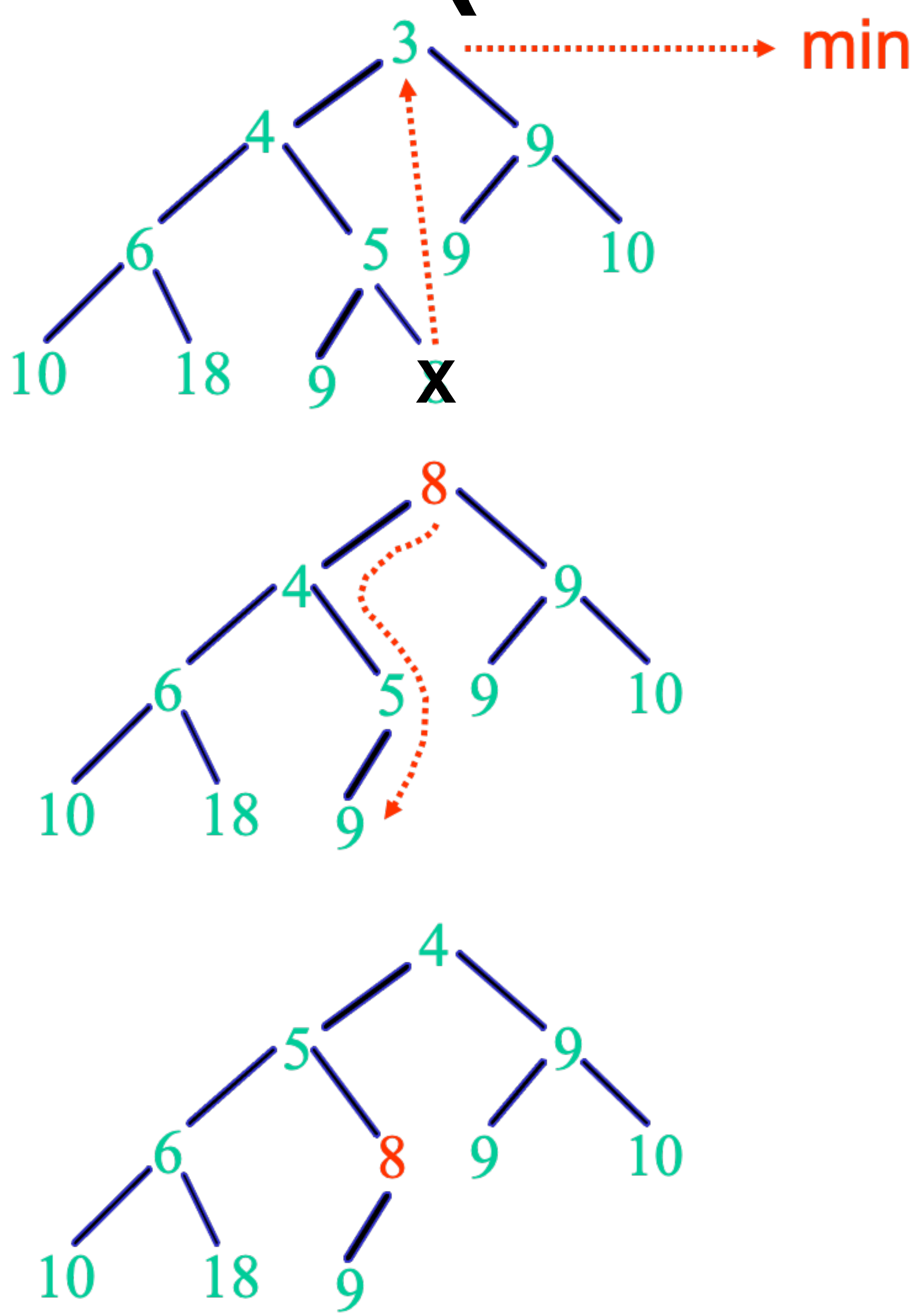
insert (A, 4)

- Placer le nouveau nœud dans la première position libre
- Permuter avec le parent jusqu'à l'obtention d'un tas
- Complexité: O(log n)

CERGY PARIS UNIVERSITÉ

浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

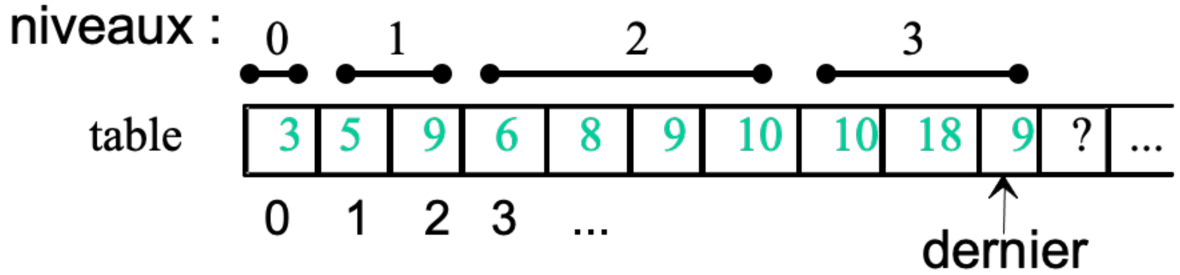
Suppression du minimum (removeMin)

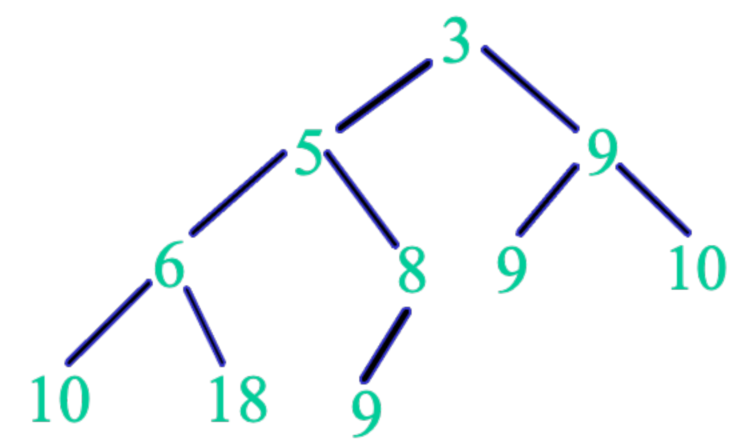


removeMIN (A)

- Donner à la racine la valeur du dernier nœud.
- Supprimer le dernier nœud.
- Echanger avec le plus petit fils jusqu'à l'obtention d'un tas.
- Complexité: O(log n)

Implantation en table





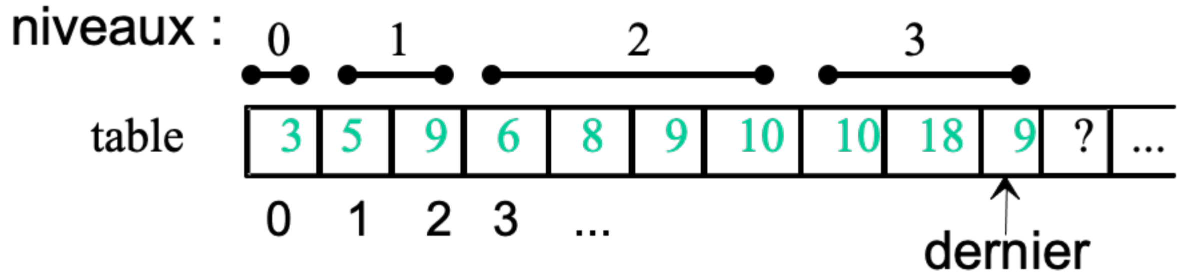
parent (i) = $\lfloor (i-1)/2 \rfloor$ si $i > 0$

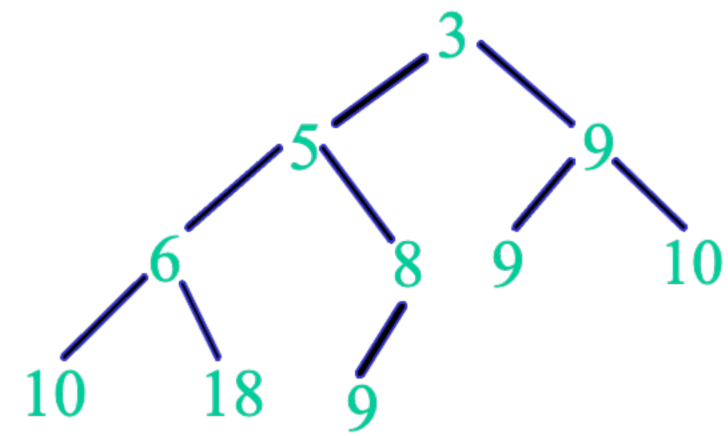
enfantgauche (i) = $2i+1$ si $2i+1 \leq$ dernier

enfantdroit (i) = $2i+2$ si $2i+2 \leq$ dernier

Pourquoi pas par nœud d'arbre?

```
typedef struct {
  element table[MAX];
  int dernier;
} Filedepriorité;
```






```
fonction insert (t tas, x élément) : tas ;
début
  t.dernier <- i <- t.dernier + 1 ;   t [i] <- x  ;
  tant que i > 0  et t.table [i] < t.table [⌊(i-1)/2⌋]  faire {
  échanger t.table [i] et t.table[⌊(i-1)/2⌋] ;  i <- ⌊(i-1)/2⌋ ;
  }
  retour (t) ;
fin


fonction removeMin (t tas non vide) : tas ;
début
  t.dernier  <- d <- t.dernier-1 ;   t.table [0]  <-  t.table [d+1] ;   i <- 0 ;
  répéter
  fin <- vrai ;
  si  2i+2 ≤ d alors {
  si t.table [2i+1] < t.table [2i+2] alors  k <- 2i+1  sinon  k <- 2i+2 ;
  si t[i] > t.table[k] alors {
                 échanger t.table [i] et t.table [k] ; i <- k ;  fin <- faux ;
  } sinon  si  2i+1= d  et  t.table [i] > t.table [d] alors {
  échanger t.table [i] et t.table [d] ;
  }
  jusqu'à fin ; retour (t) ;
fin
```

Tri par tas en ordre croissant (heap sort) avec tas max

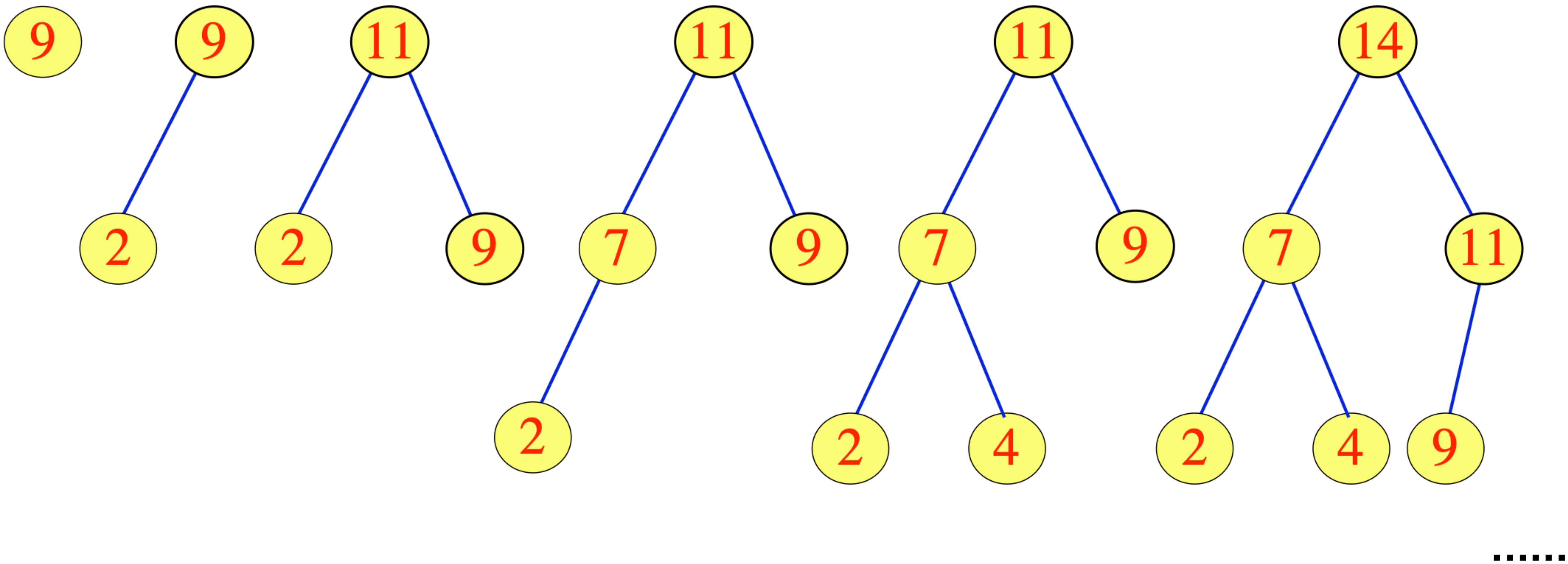
- On applique le tas max (i.e., pour tout nœud p ≠ racine, Elt (p) ≤ Elt (Parent (p))   )
- On part d'un tableau vide L'. On commence par construire un tas en ajoutant successivement au tas vide les éléments L'[0], L'[1], …
- On répète ensuite les opérations suivantes :
  - prendre le maximum,
  - le retirer du tas,
  - le mettre à droite du tas
- Complexité: O(n log n)

CERGY PARIS UNIVERSITÉ

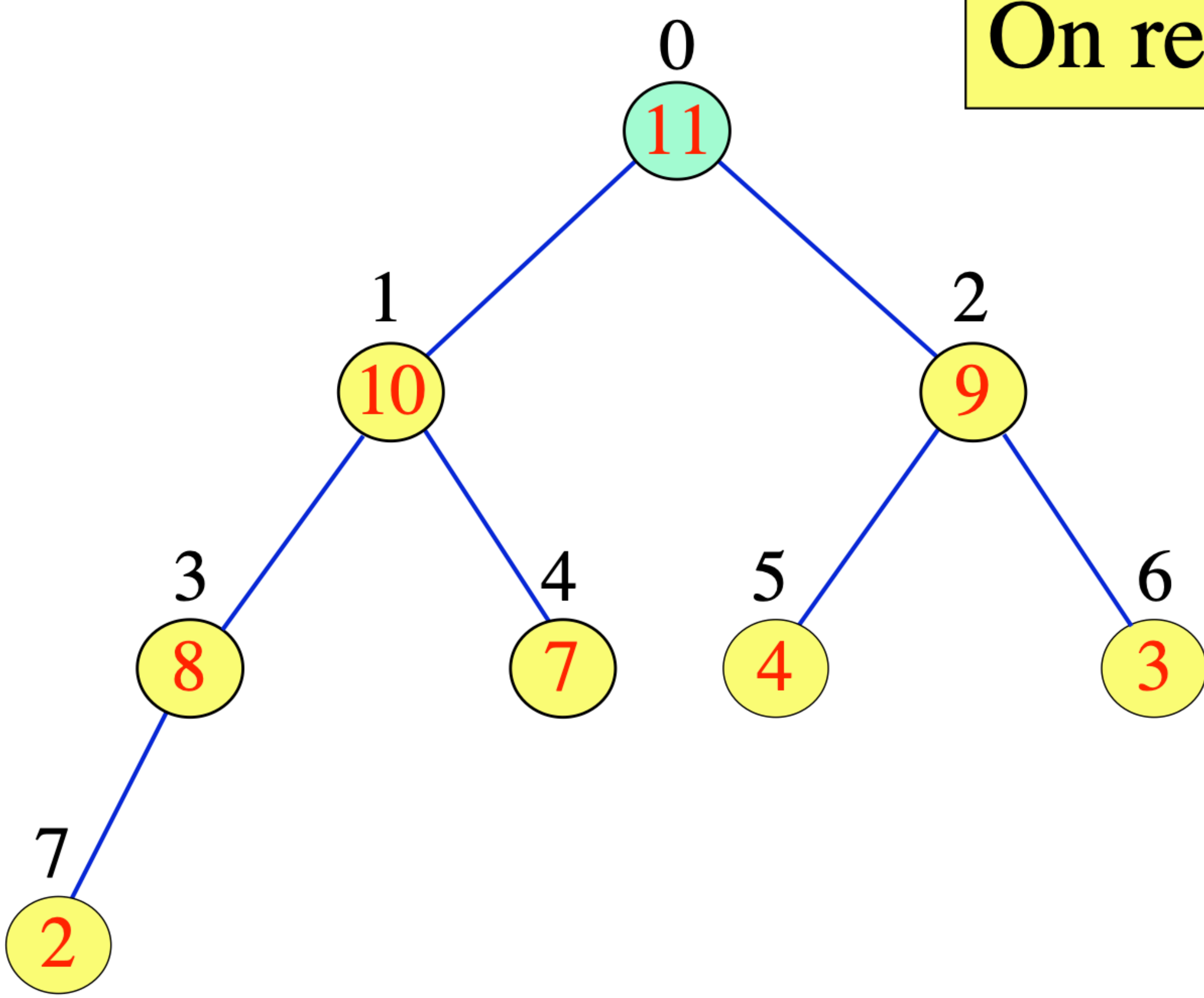
浙江科技學院 ZHEJIANG UNIVERSITY OF SCIENCE & TECHNOLOGY

Exemple

Trier (9, 2, 11, 7, 4, 14, 3, 16, 8, 10, 15)



On retire les éléments un à un.



…

0	1	2	3	4	5	6	7	8	9	10
11	10	9	8	7	4	3	2	14	15	16

Tri par tas en ordre descendant avec tas min

```
fonction Sort(L liste) : liste ;
début
A <- tas_vide ;
  pour x <- premier au dernier élément de L faire  O(L)
        A <- Insert (A, x) ;  O(log L)
  fin pour
  L' <- liste_vide ;
  tant que non Vide(A) faire
    L' <- L' U (MIN(A)) ;  O(L log L)
    A <- removeMin(A) ;
  fin tant que
  retour (L') ;
fin
```

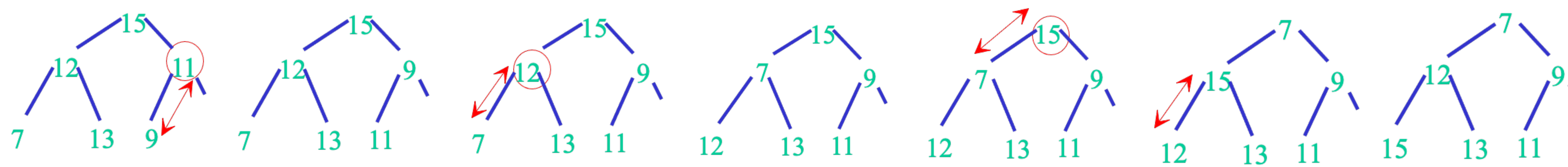
Temps maximum $O(|L| \log |L|)$

Temps du "pour" O(L log L+L log L) = O(L log L) O(n Log n)

$O(|L| \log |L|)$ avec $L = (n, n\text{-}1, \ldots, 3, 2, 1)$

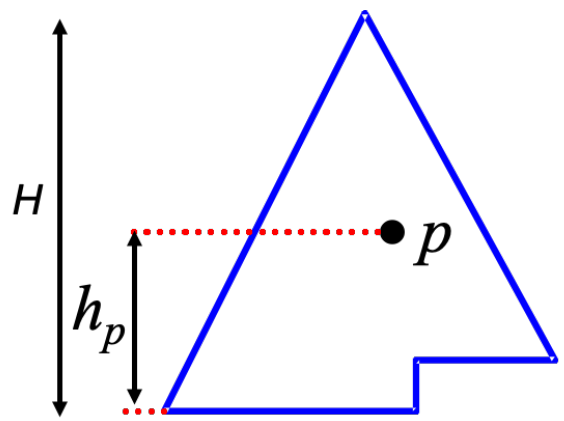
Mise en tas globale

Création d'un tas avec n éléments déjà dans la table
Transformation d 'un arbre binaire parfait en tas




```
fonction Mise_en_tas ( tab [0 .. n-1] table ) : tas ;
début
  pour i <-  ⌊(n-2)/2⌋  à 0  pas -1  faire
    Entas (tab, i) ; fin-pour
  retour (tab, n-1) ;
fin
procédure Entas (tab [0 .. n-1] table,  i indice) ;
/* les sous-arbres de racines k sont des tas, pour i < k ≤  n-1 */
début
  si  2i+2 = n  ou  tab [2i+1] ≤ tab [2i +2] alors  k <- 2i+1 sinon  k <- 2i+2
  si tab [i] > tab [k]  alors {
    échanger tab [i] et tab [k] ;
  si k £ ⌊(n-2)/2⌋    alors  Entas (tab, k) ;
  }
fin
```

Temps linéaire



Temps (Entas, *p*) $\leq$ a h_p
nombre d 'échanges $\leq h_p$
nombre de comparaisons $\leq 2h_p$
$2^H \leq n \leq 2^{H+1}-1 \quad H = \lfloor \log_2 n \rfloor$

L'algorithme peut passer beaucoup de temps sur les noeuds proches de la racine qui sont peu nombreux
mais il passe peu de temps sur les éléments profonds dans l'arbre qui sont très nombreux

Temps (Mise_en_tas, *n*)

$$\leq \alpha H.1 \;+\; \alpha\,(H-1).2 \;+\; \ldots \;+\; \alpha\,(H-i).2^i \;+\; \ldots \;+\; \alpha 1.2^{H-1}$$

$$\leq \alpha \sum_{i=0}^{H-1} (H-i)2^i = \alpha 2^H \sum_{i=0}^{H-1} \frac{H-i}{2^{H-i}} \leq \alpha n \sum_{i=0}^{H-1} \frac{H-i}{2^{H-i}} = \alpha n \sum_{k=1}^{H} \frac{k}{2^k} \leq 2\alpha n$$

$$\begin{array}{rcl}
\displaystyle\sum_{k=1}^{\infty} \frac{k}{2^k} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \ldots & = & 2 \\
 & \parallel & \\
= \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots & = & 1 \\
 & + & \\
+ \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \ldots & = & \frac{1}{2} \\
 & + & \\
+ \frac{1}{8} + \frac{1}{16} + \ldots & = & \frac{1}{4} \\
 & + & \\
+ \ldots & = & \ldots
\end{array}$$

Théorème : l'algorithme de mise en tas fonctionne en temps O(n)