Algorithmique, Structures de Données et Graphe

Licence 2 – IMA

2023 -2024

Programme

- La structure de données Arbre Binaire.
- Introduction aux graphes.
  Représentation informatique des graphes.
- Coloration de graphes.
- Exploration d'un graphe.
- Chemin du cout minimum d'un graphe.

Objectifs

- Savoir écrire un algorithme de qualité et le traduire en un langage de programmation (Java)
- Savoir structurer (organiser) les données d'un problème
- Savoir écrire des algorithmes de base sur les graphes et les utiliser dans différentes applications
- Appliquer au projet informatique de l'année

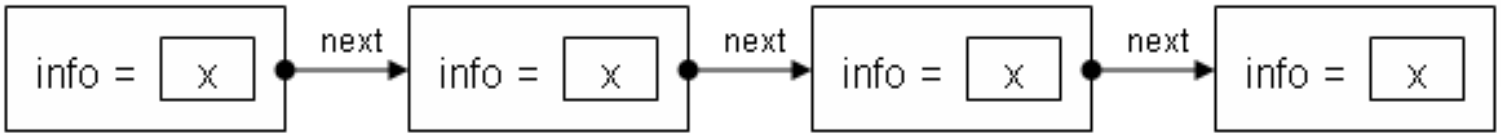
Structure de Données

- Ensemble organisé d'informations pouvant être décrit, créé et modifié par des algorithmes

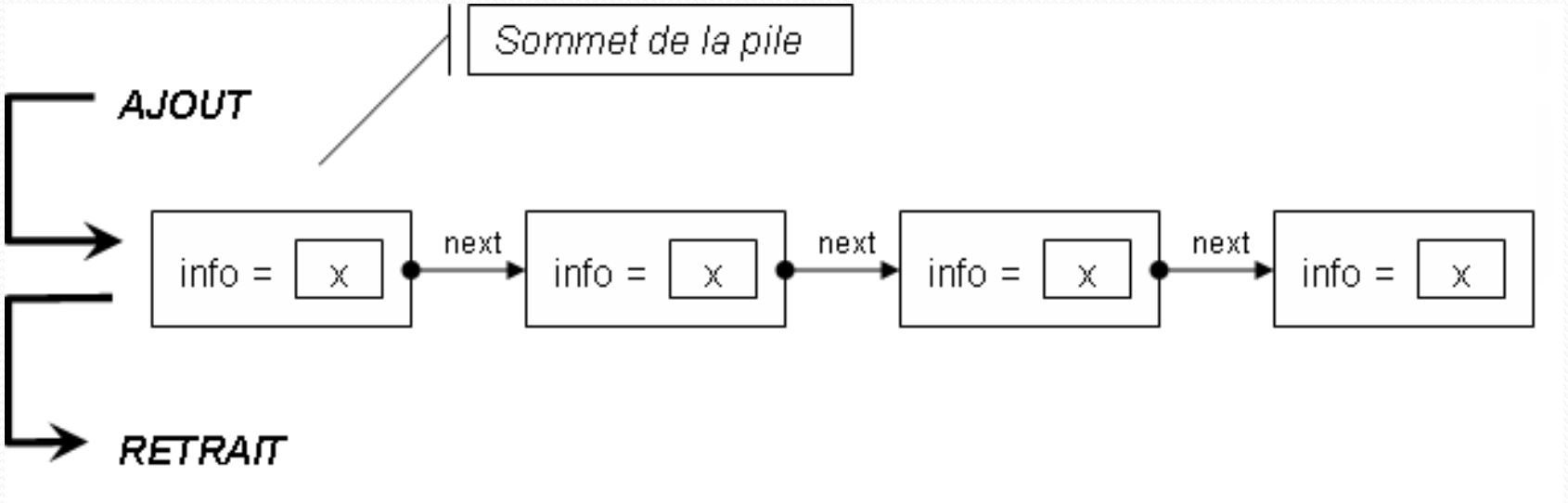
- Organisation interne :
  - structuration adoptée pour stocker les informations
  - implémentation des primitives

- Deux formes de liens entre informations :
  - les structures de données linéaires (piles, files et listes)
  - les structures de données ramifiées (arbres, tas, graphes)

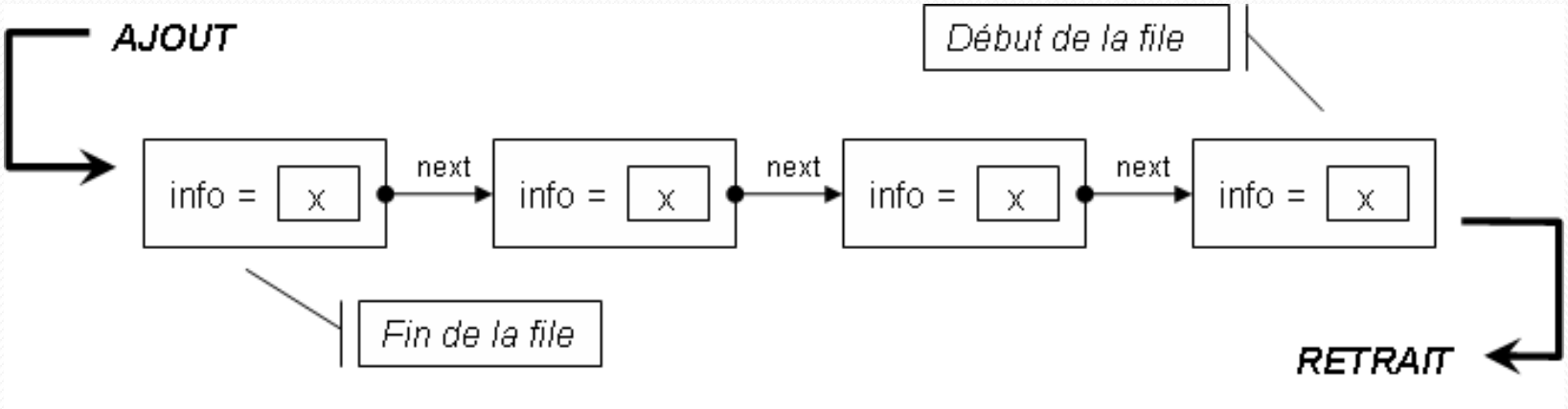
- Liste : possibilité d'ajout et de suppression en tout point



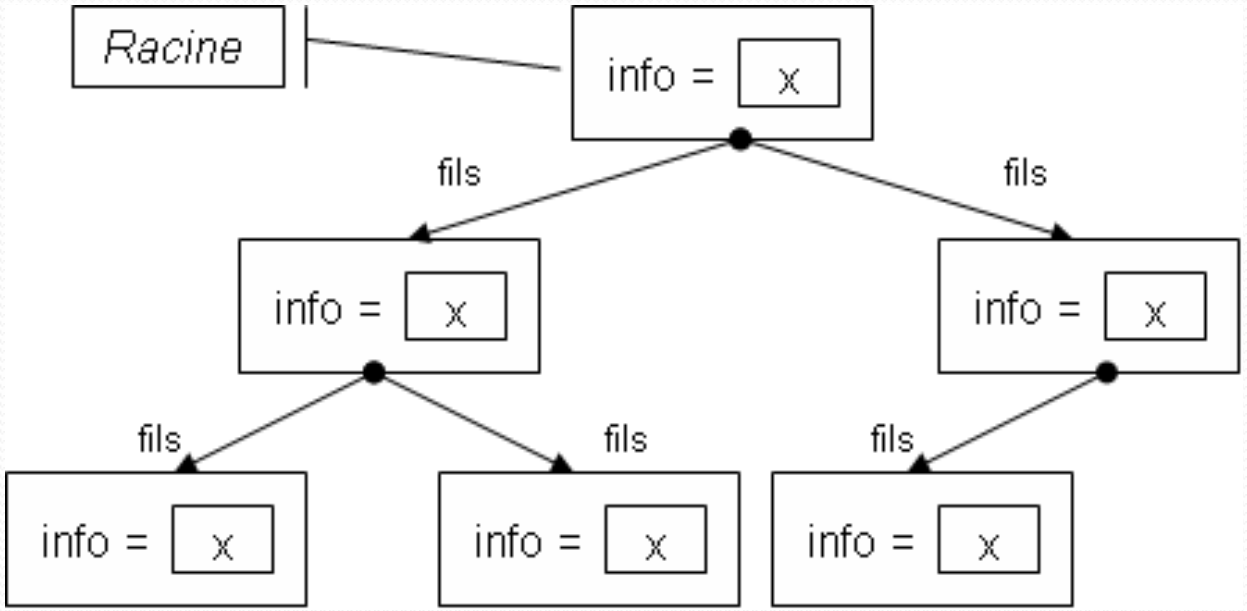
- Pile : « Dernier Entré, Premier Sorti »



- File : « Premier Entré, Premier Servi »

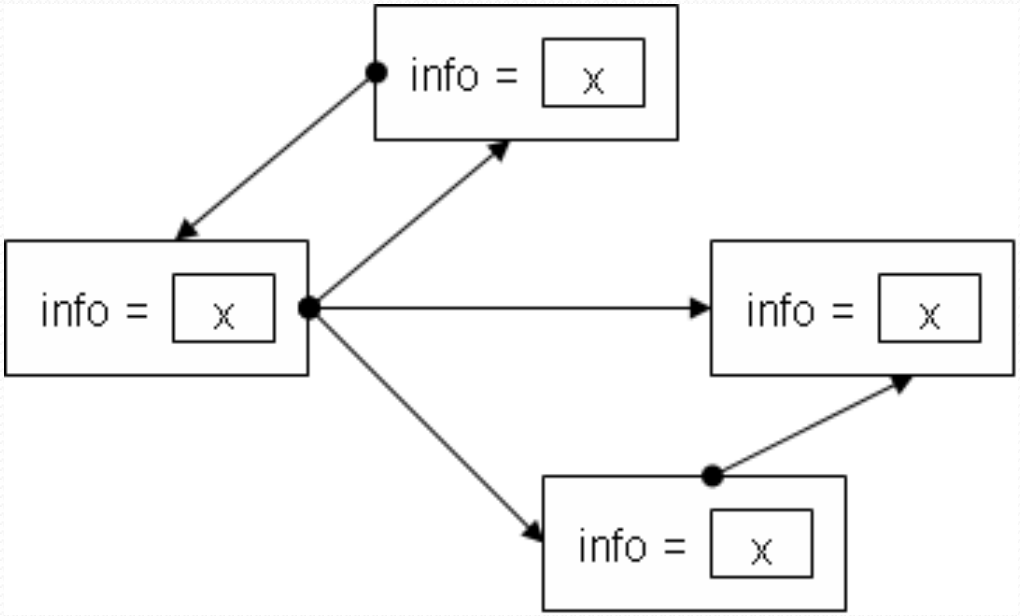


- Arbre : Structure hiérarchique « père-fils »



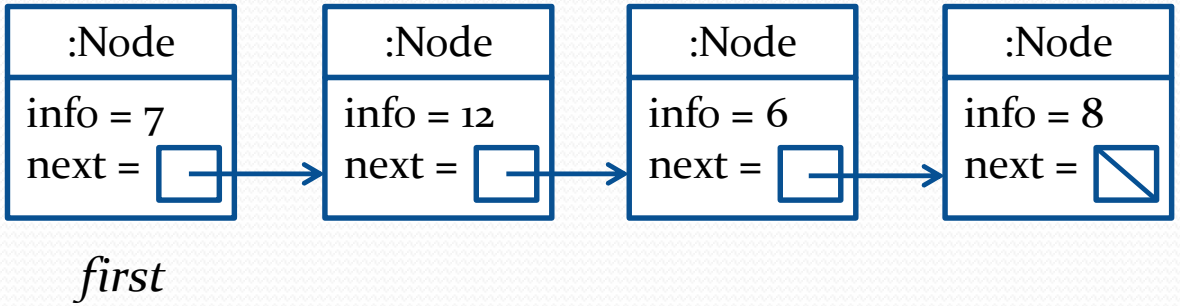
- Tas : Arbre où l'info du nœud parent est plus grande (plus petite) que celles de ses enfants

- Graphe : tous liens possibles



Liste chainée

- Structure de données linéaire où chaque élément a un successeur (sauf le dernier)



- Chaque élément est rangé dans un objet (***nœud***) contenant :
  - la valeur de l'élément
  - et le ***lien*** vers le nœud de l'élément suivant

- Une liste est définie par le lien vers le premier nœud

Node
– info : Entier – next : Node
+ Node(Entier) + getInfo() : Entier + getNext() : Node + setNext(Node)

ListByRef
– first : Node
+ ListByRef() – addFirst(x : Entier) – addAfter(x : Entier, p : Node) + add(i : Entier, x : Entier) + remove(x : Entier) + get(i : Entier) : Entier + contains(x : Entier) : booléen + print()

- Il y a différentes variantes de listes chaînées

```
public class Node {
    private int info;
    private Node next;

    public Node(int x) {
        info = x;
        next = null;
    }
    ....
}

public class ListByRef {
    private Node first;

    public ListByRef() {
        first = null;
    }
    ....
}
```

Opérations sur les listes chaînées

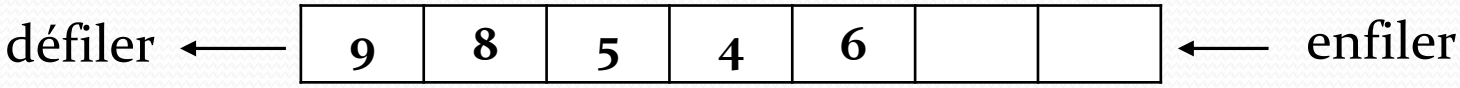
- Parcours des éléments : A l'aide d'une variable qui représente le nœud courant

- Ajout d'un élément dans une liste :
  - Créer un nouveau nœud contenant l'élément à ajouter.
  - Insérer ce nœud à l'endroit souhaité en modifiant le lien entre les nœuds.

- Suppression d'un élément d'une liste :
  - Rechercher le nœud contenant l'élément à supprimer et le nœud qui le précède.
  - Supprimer le nœud contenant l'élément en modifiant les liens entre ses voisins.

SD File (*Queue*)

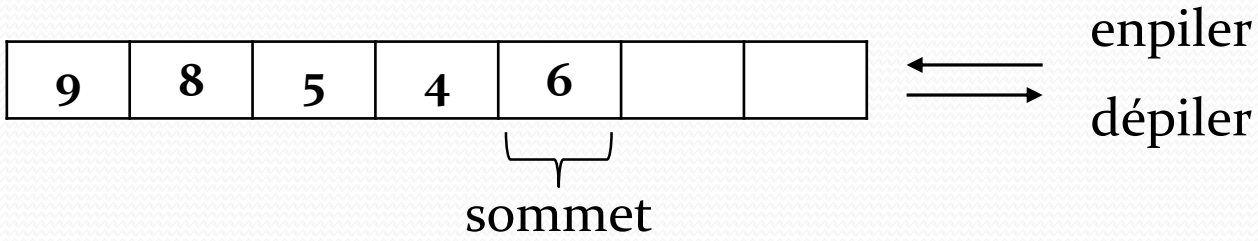
- Liste qui fonctionne d'après le principe « *Premier Entré, Premier Sorti* » (*FIFO*)



- Applications :
  - Parcours de graphes : parcours en largeur
  - File d'attente : guichet, ordonnancement des tâches, liste d'impressions

SD Pile (*Stack*)

- Liste qui fonctionne d'après le principe « *Dernier Entré, Premier Sorti* » (*LIFO*)



- Applications :
  - Parcours de graphes : parcours en profondeur
  - Recherche de chemin dans un labyrinthe
  - Transformation des expressions : infixée en post-fixée
    Evaluation d'une expression post-fixée

Algorithmes récursifs

- Un algorithme récursif est un algorithme qui s'appelle lui-même.

- Un algorithme récursif doit comporter :
  - Un ou plusieurs cas d'arrêt où l'exécution s'arrête (*)
  - L'expression du problème à résoudre par lui-même avec une donnée de taille plus petite (ou plus grande) sous forme des appels récursifs (*relation de récurrence*) (**)

Exemple : Calcul de la factorielle

- Définition :
  - Version 1 :

    $n! = 1.2.3...n$

  - Version 2 :

    $$n! = \begin{cases} 1 & \text{si } n = 0 \\ n.(n-1)! & \text{si } n > 0 \end{cases}$$

- Algorithme :
  - Version 1 : Algorithme itératif

Algorithme *fact*(n : Entier)
Variables
  i, p : Entiers
Début
  p ← 1
  Pour i de 1 à n **Faire**
    p ← p * i;
  FinPour
  Renvoyer p
Fin

- Version 2 : Algorithme récursif

Algorithme *fact*(n : Entier)
Variables
  p : Entiers
Début
  Si n = 0 **Alors**
    p ← 1 (*)
  Sinon
    p ← n * *fact*(n – 1) (**)
  FinSi
  Renvoyer p
Fin

Définitions récursives pour listes

- Une liste *L* est :
  - soit vide
  - soit composée :
    - du premier élément (*L.premier*)
    - et de la liste des éléments restant (*L.reste*)

- Une liste est une structure de données récursive. Les traitements sur une liste peuvent être définis récursivement.

- Vérifier si *x* appartient à la liste *L* :
  - Si *L* est vide : Faux
  - Sinon :
    - Si *x* = *L.premier* : Vrai
    - Sinon : Vérifier si *x* appartient à la liste *L.reste*

- Calculer la somme des éléments de la liste *L* de nombres
  - Si *L* est vide : 0
  - Sinon, calculer la somme des éléments de la liste *L.reste* puis y ajouter *L.premier*

```
Algorithme rechercher(x : Entier, l : Liste d'Entiers)
Variables
   r : Booléen
Début
   Si l est vide Alors
      r ← Faux                                   (*)
   Sinon
      Si l.premier = x Alors
         r ← vrai
      Sinon
         r ← rechercher(x, l.reste)              (**)
      FinSi
   FinSi
   Renvoyer r
Fin
```

- Exemple : Définir dans la classe *ListByRef* une méthode récursive qui calcule la somme de ses éléments.

- Remarques :
  - La méthode doit avoir un paramètre représentant la liste et qui peut être décomposé en le premier élément et du reste.

  - Si la liste est réprésentée par son premier nœud *p* de type *Node*,
    p.getInfo() donne le premier élément de la liste
    p.getNext() donne le premier nœud de la liste de éléments restant

- Définir une méthode récursive qui calcule la somme des éléments d'une liste dont le premier nœud est *p* :

```
public int sum(Node p) {
  if (p == null) {
    return 0;
  } else {
    return p.getInfo() + sum(p.getNext());
  }
}
```

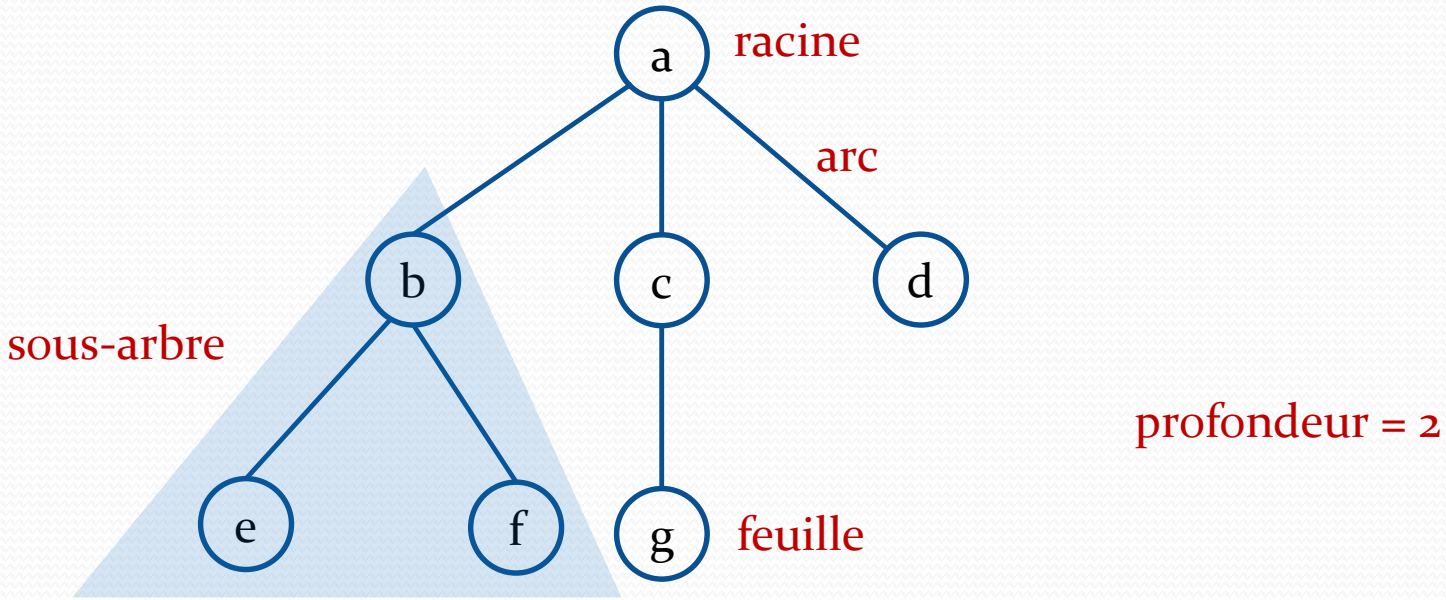
- La méthode qui calcule la somme des éléments de la liste dans *ListByRef* sera :

```
public int sum() {
   return sum(first);
}
```

Arbres Binaires

Définitions

- Structure de données Arbre :
  - Chaque élément peut avoir plusieurs successeurs mais un seul prédécesseur.
  - Un élément n'a pas de prédécesseur et est appelé ***racine***.
  - Il y a un chemin de la racine à tout nœud.



- La structure de donnée Arbre est une structure de donnée *récursive*.

  Un arbre A est :
  - soit vide
  - soit composé d'une racine connectée à des arbres A_1, …, A_n appelés ***sous-arbres*** de A

- ***Racine***, ***sous-arbre***, ***nœud***, ***arc***
  feuille : nœud sans successeur
  nœud interne : nœud avec successeur

- ***Degré*** d'un nœud : nombre de ses successeurs

- ***Niveau*** d'un nœud : sa distance à la racine en nombre d'arcs

- ***Profondeur/hauteur*** : le plus grand niveau des feuilles (longueur de la plus longue branche)

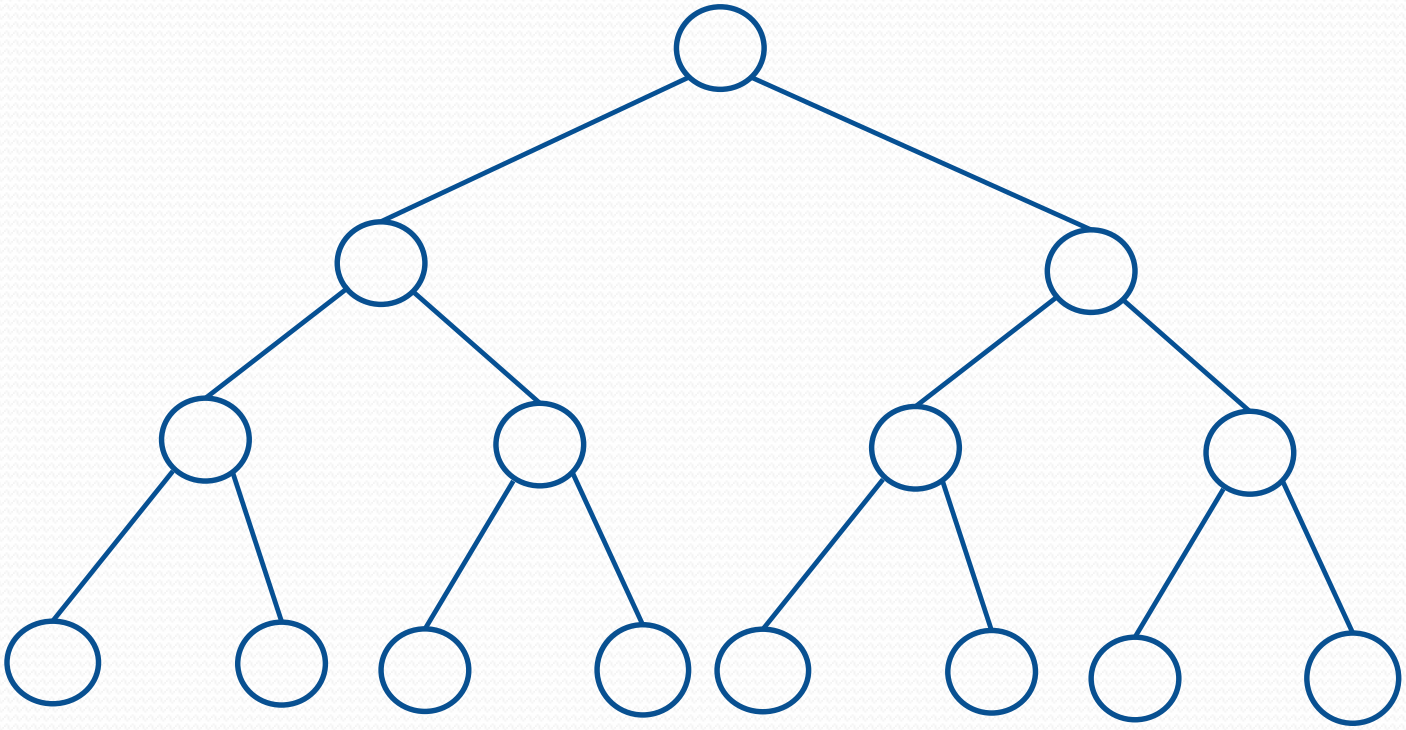
- Arbre ***n-aire*** : *n* est le degré maximal des nœuds

- Arbre n-aire ***complet*** :

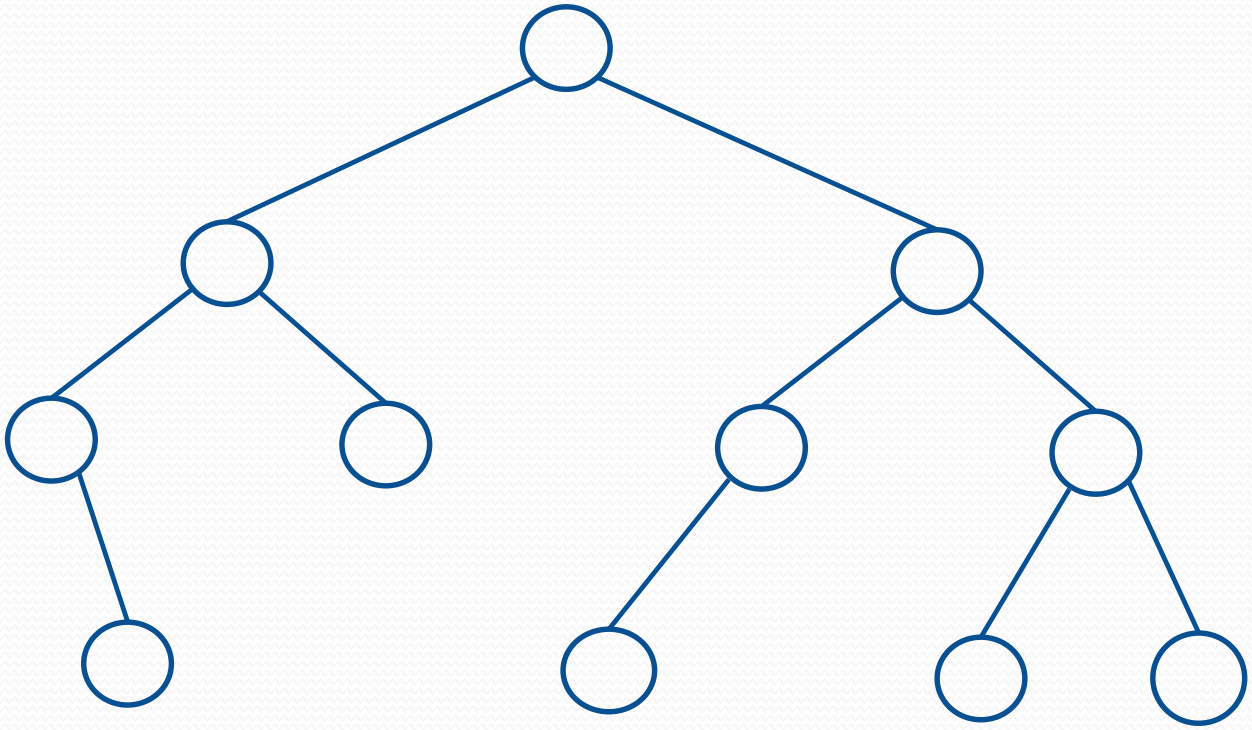
  Tout nœud interne a le degré *n*,
  toutes les feuilles sont du même niveau

- Arbre n-aire ***complet à gauche*** :

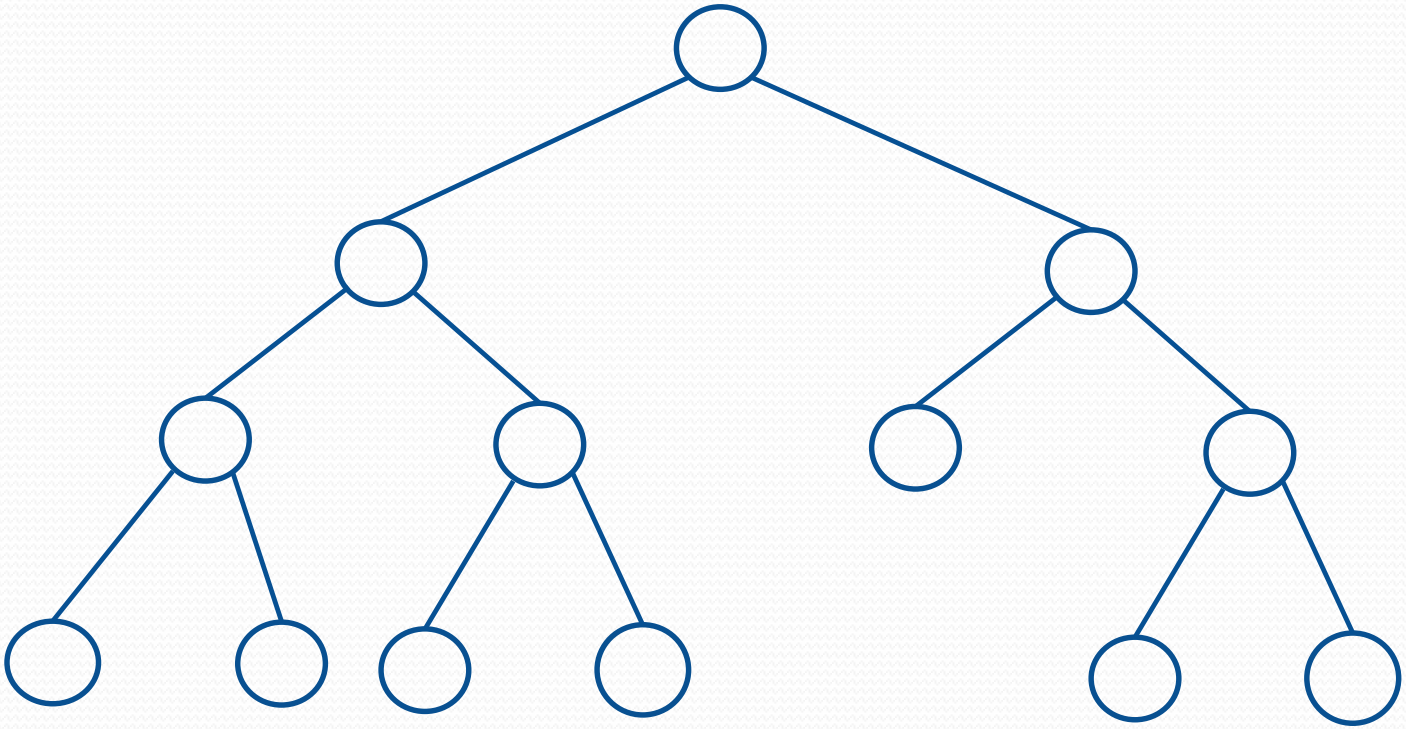
  complet jusqu'à l'avant dernier niveau,
  toutes les feuilles sont cadrées à gauche



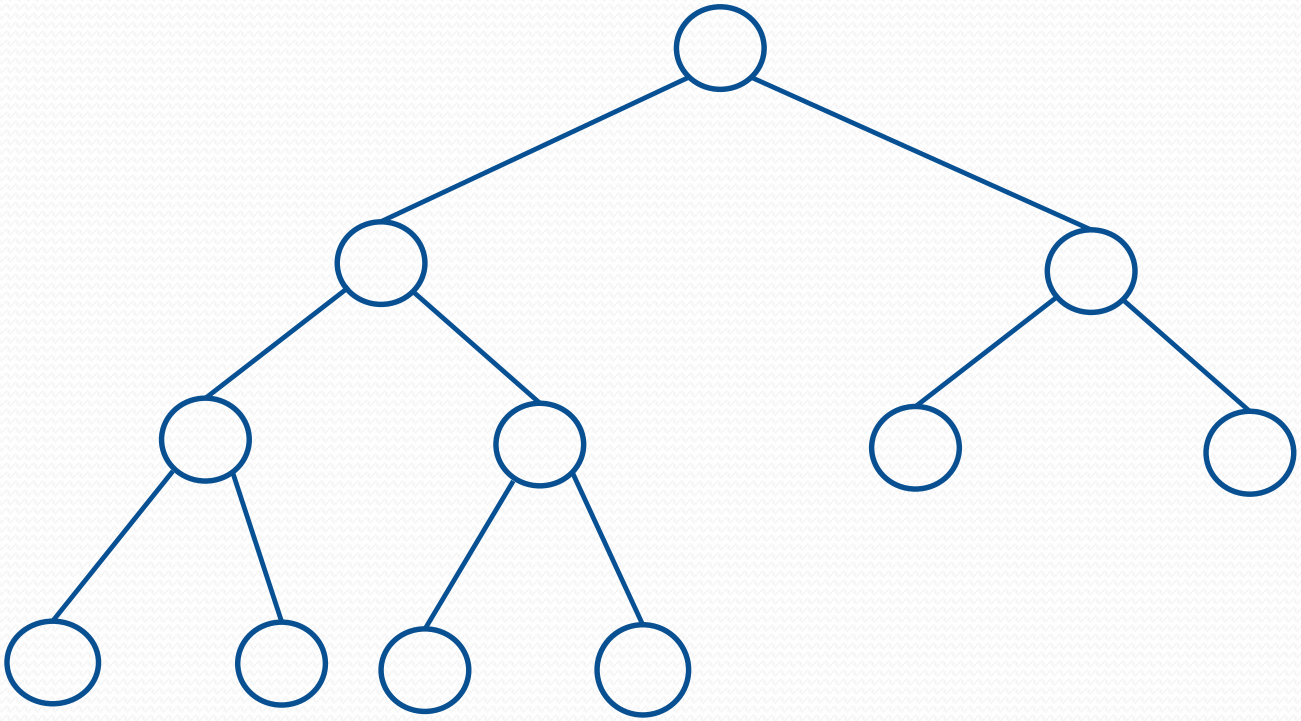
arbre complet



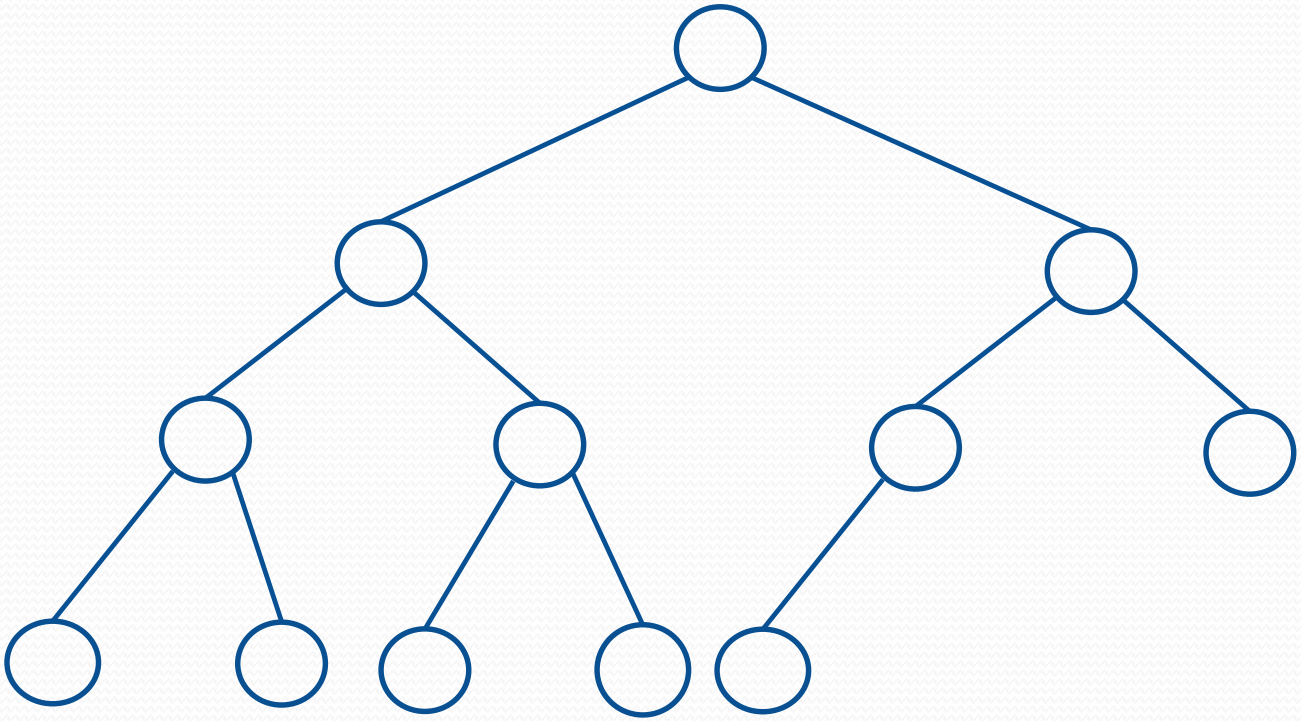
arbre complet à gauche ?



arbre complet à gauche ?

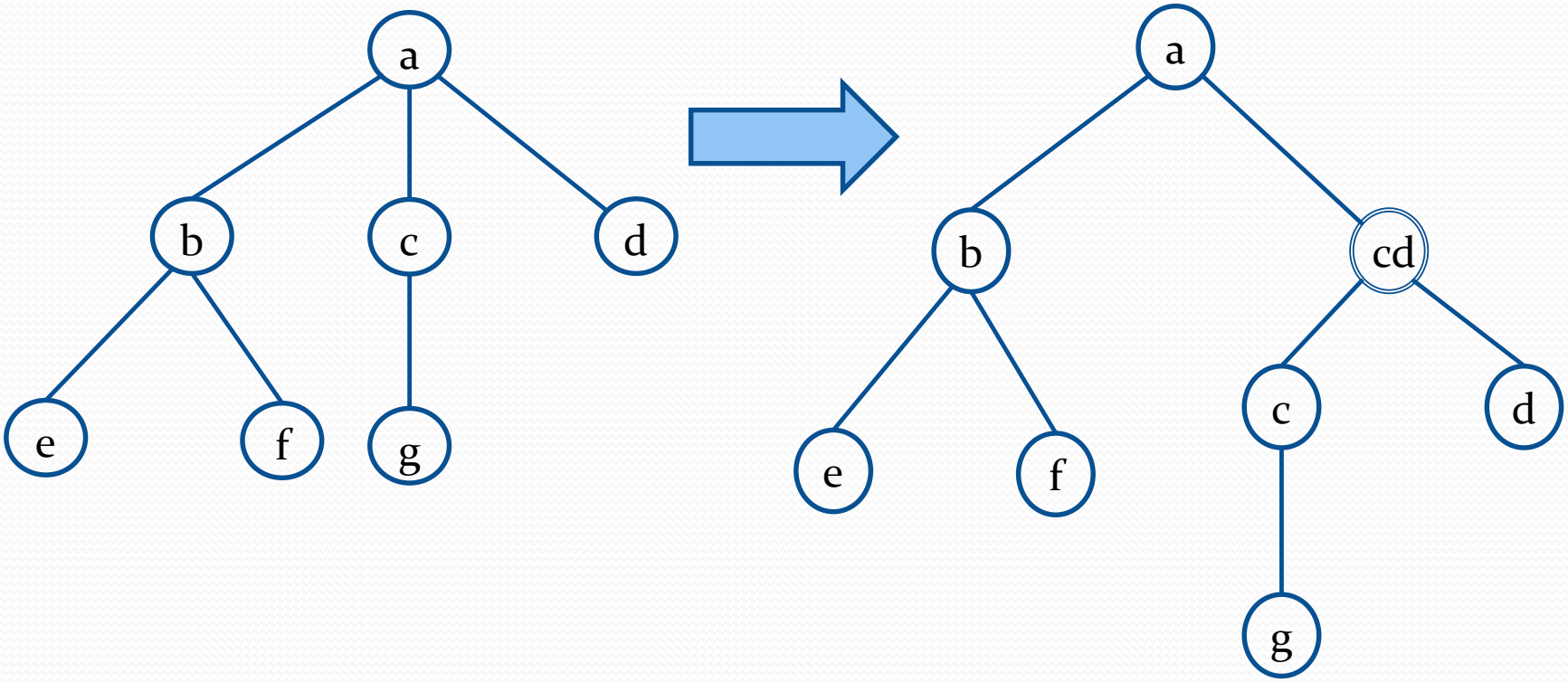


arbre complet à gauche ?



arbre complet à gauche ?

- Tout arbre n-aire, n > 2, peut être restructuré en un arbre binaire en ajoutant des nœuds artificiels



Arbre binaire

- Tout nœud a au plus deux successeurs

- Un arbre binaire est :
  - soit vide
  - soit composé d'une racine, d'un sous-arbre gauche et d'un sous-arbre droit.

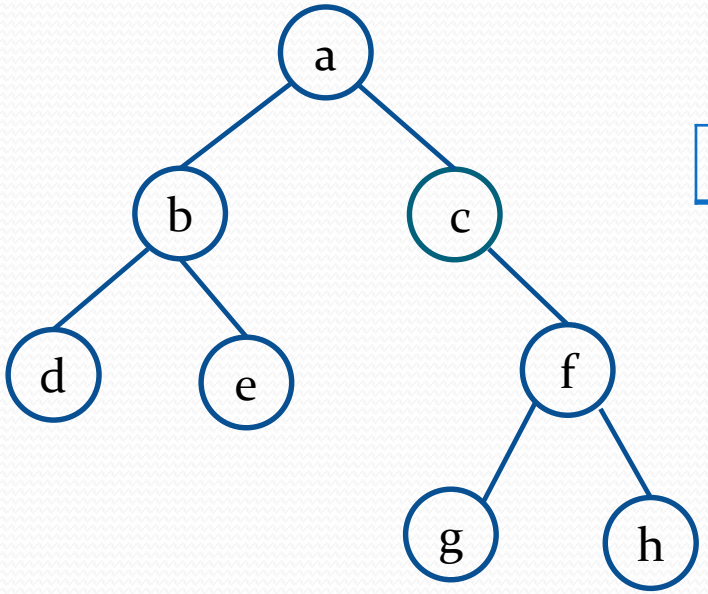
- Organisation interne :
  - liens implicites par un tableau
  - liens explicites par références

Liens implicites

- Par un tableau *t* :

  Chaque élément est rangé dans *t*[i], i = 1, 2, …

  Ses successeurs sont rangés dans *t*[2*i] et *t*[2*i+1]

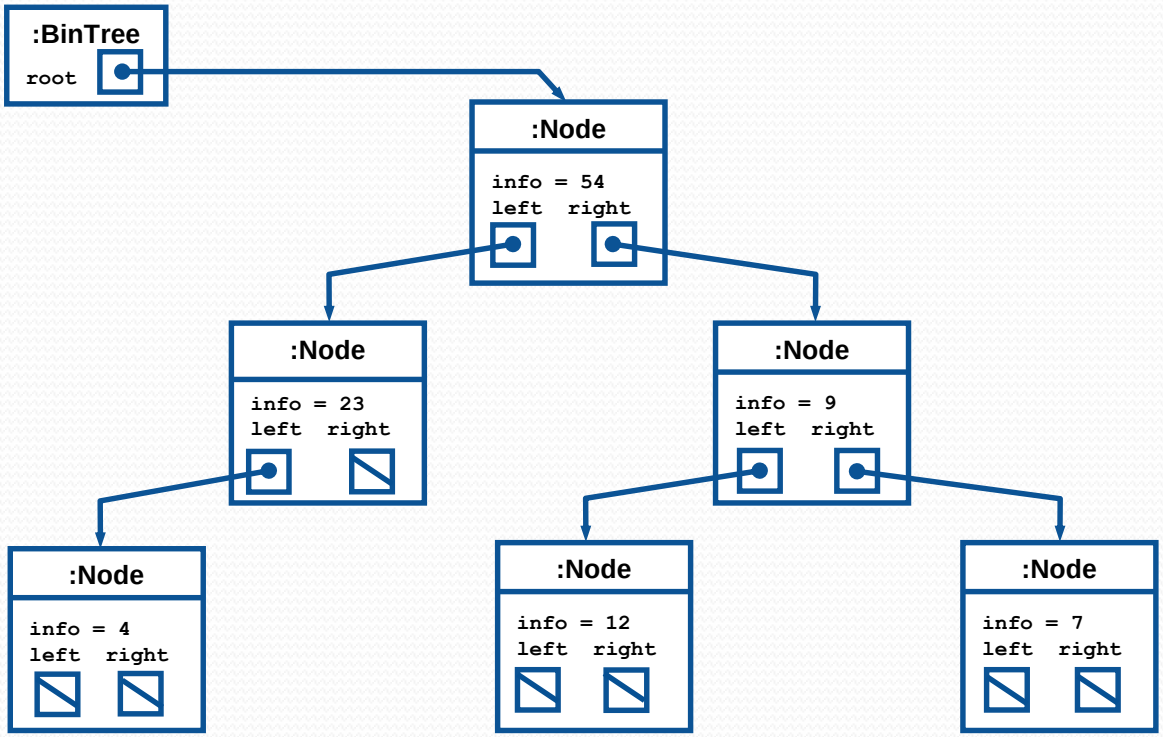


1	2	3	4	5	6	7	8	…	14	15
a	b	c	d	e	-	f	-	…	g	h

Liens explicites par référence

- Chaque élément est rangé dans un nœud contenant :
  - sa valeur
  - un lien vers la racine du sous-arbre gauche
  - un lien vers la racine du sous-arbre droit

- Exemple :


:BinTree
root
:Node
info = 54
left right
:Node
info = 23
left right
:Node
info = 9
left right
:Node
info = 4
left right
:Node
info = 12
left right
:Node
info = 7
left right

- Définir en Java les classes permettant de représenter un arbre binaire contenant des nombres entiers.

Node
– info : Entier – left : Node – right : Node
+ Node(Entier) + getInfo() : Entier + getLeft() : Node + setLeft(Node) + getRight() : Node + setRight(Node)

BinTree
– root : Node
+ BinTree() + print() + ...

```
public class Node {
   private int info;
   private Node left, right;
   ....
}

public class BinTree {
   private Node root;

   public BinTree() {
      root = null;
   }
   ....
}
```

Parcours d'un arbre binaire

- Différents parcours suivant l'ordre de visite de la racine (*R*), du sous-arbre gauche(*SAG*) et du sous-arbre droit(*SAD*) :
  - *RGD*, *GRD*, *GDR*
  - *RDG*, *DRG*, *DGR*

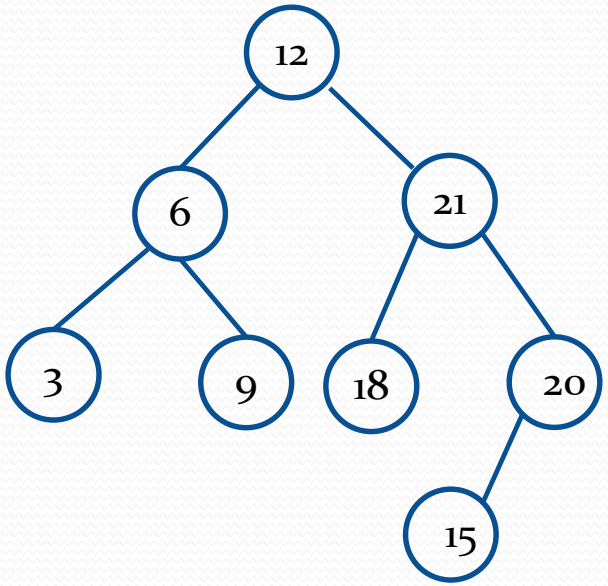
- Algorithme récursif :

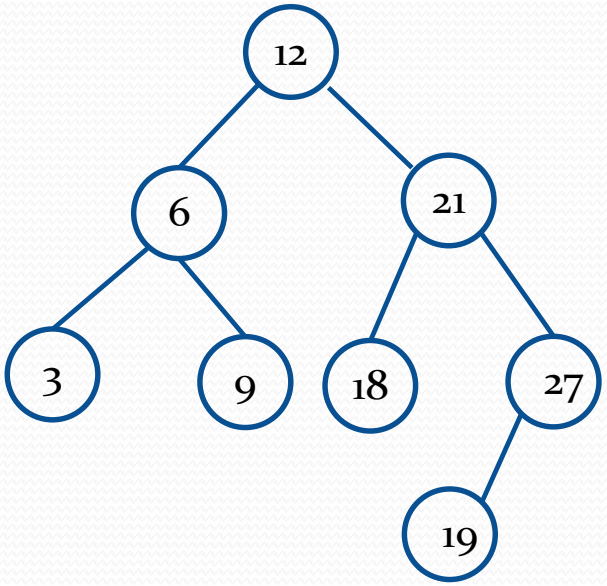
```
Algorithme parcoursGRD(r : Node)
Début
   Si r ≠ null Alors
      parcoursGRD(r.left)
      Traiter r.info
      parcoursGRD(r.right)
   FinSi
FinPour
```

Arbre binaire de recherche

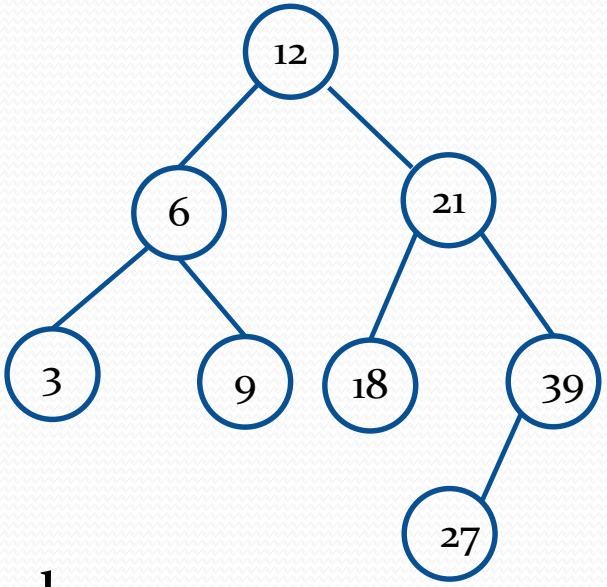
- L'information dans tout nœud est :
  supérieure à celle de tout nœud du SAG
  et inférieure à celle de tout nœud du SAD

- Si l'on parcourt par GRD un arbre binaire de recherche, on obtient la liste de tous ses éléments triée par ordre croissant

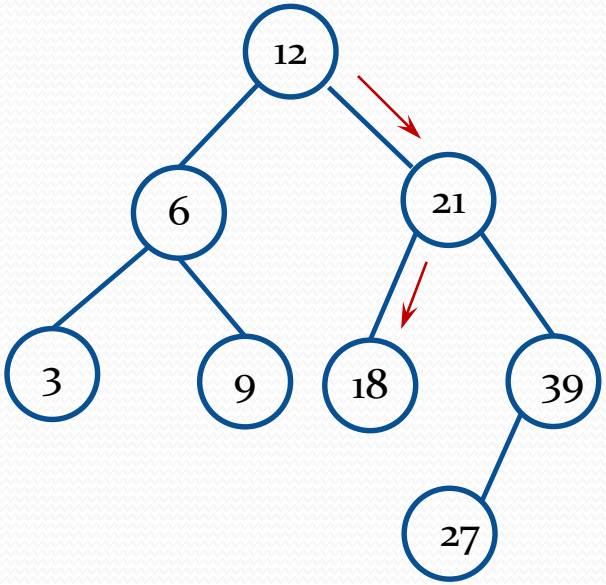

12
6
21
3
9
18
20
15


12
6
21
3
9
18
27
19

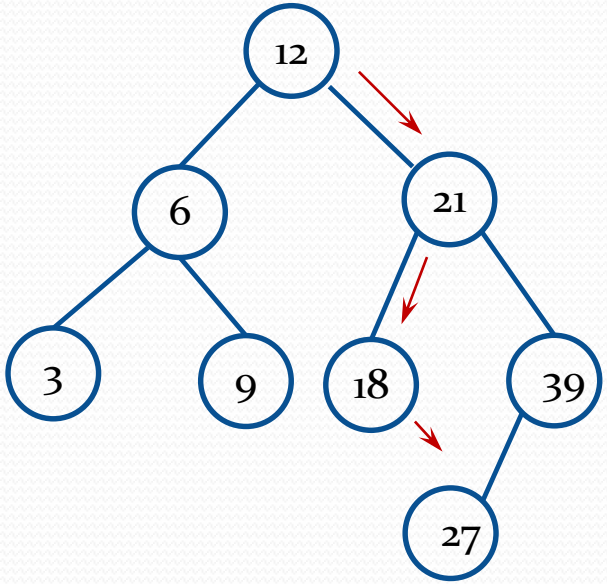
- Recherche d'un élément dans un ABR :
  - Parcourir la branche pouvant contenir l'élément cherché, suivant sa valeur.



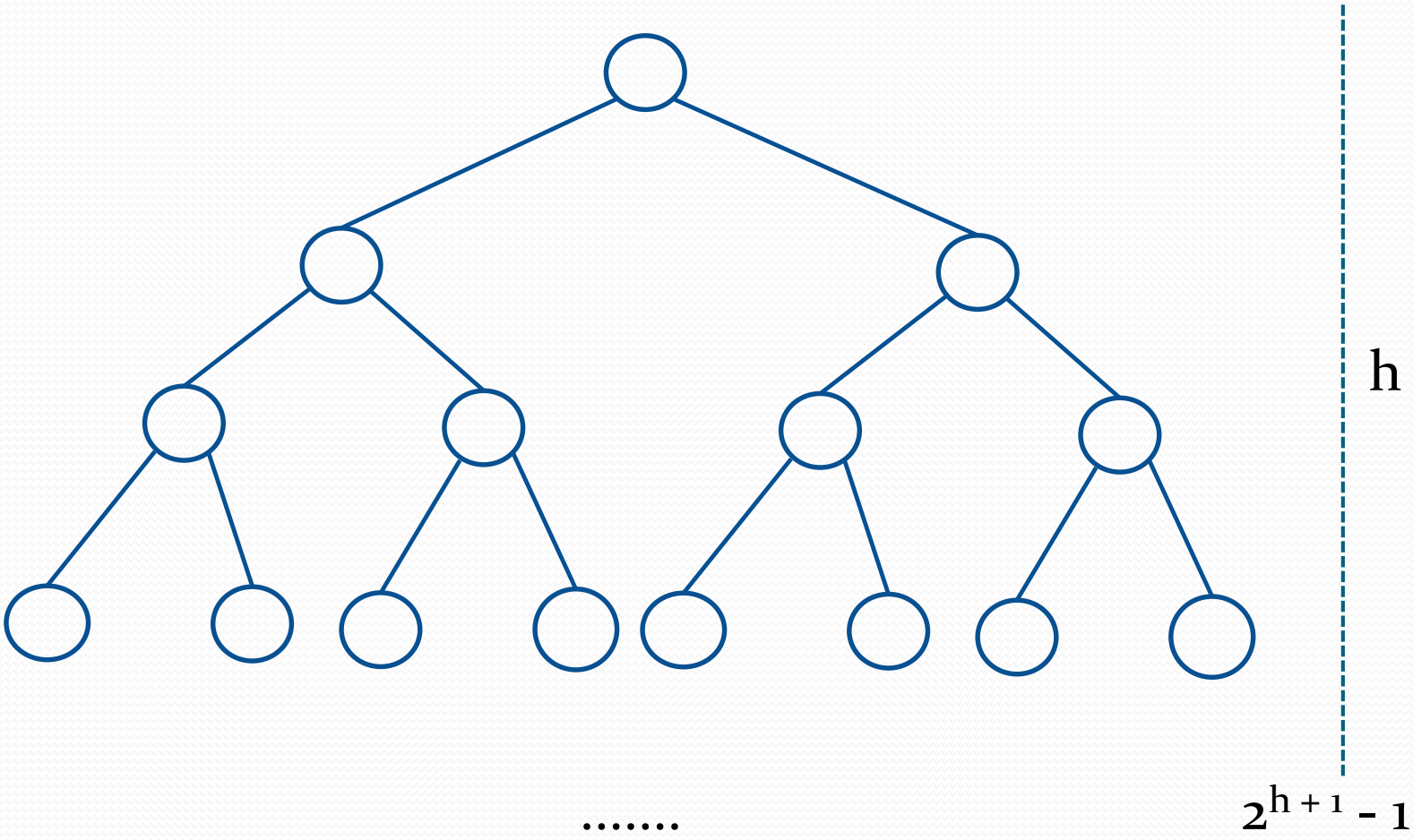
  - Remarque :
    Si l'arbre est équilibré et a n nœuds,
    la longueur du parcours est $O(log(n))$



Chercher 18



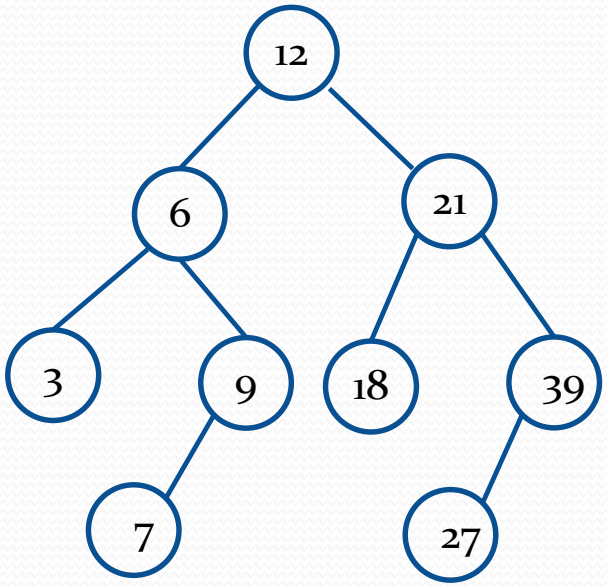
Chercher 20


h
.....
$2^{h+1} - 1$

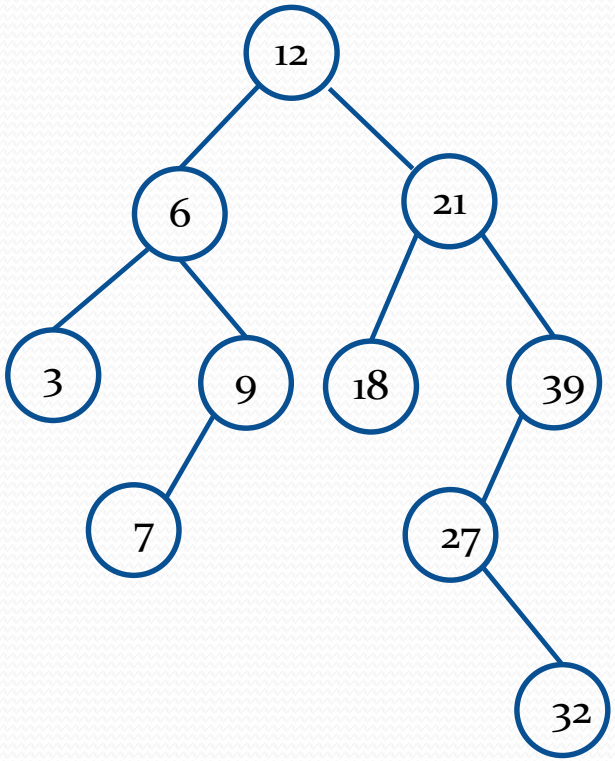
- Chercher x :

current ← root
Tant que current ≠ *null* et current.info ≠ x **Faire**
    Si x < current.info **Alors**
        current ← current.left
    Sinon
        current ← current.right
    FinSi
FinTantque
Si current ≠ *null* **Alors**
    Renvoyer *Vrai*
Sinon
    Renvoyer *Faux*

- Ajout d'un élément à un ABR :
  - Ajouter une feuille contenant le nouvel élément :
    - parcourir jusqu'à la fin de la branche pouvant avoir la nouvelle feuille
    - créer le nouveau nœud et l'attacher au dernier nœud de la branche à l'endroit approprié
  - Ajouter le nœud contenant le nouvel élément à la racine



Ajouter 7



Ajouter 32

- Suppression d'un élément d'un ABR :

1. L'élément se trouve dans une feuille :

   Supprimer la feuille en modifiant le lien de son parent

2. L'élément se trouve dans un nœud avec un seul fils

   Supprimer le nœud en rattachant son fils à son parent

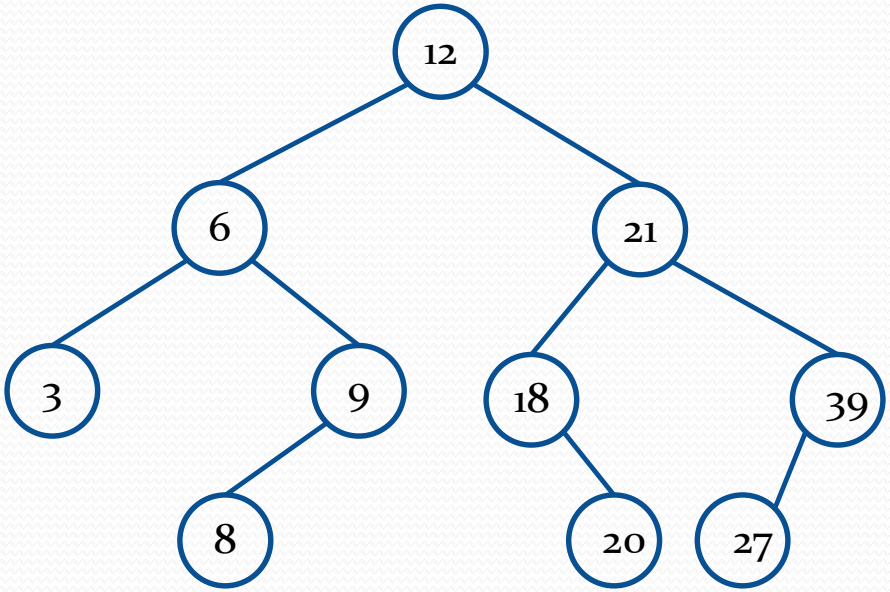
3. L'élément se trouve dans un nœud avec deux fils

   Mettre la plus petite valeur de son SAD dans ce nœud
   (qui écrase la valeur à supprimer)

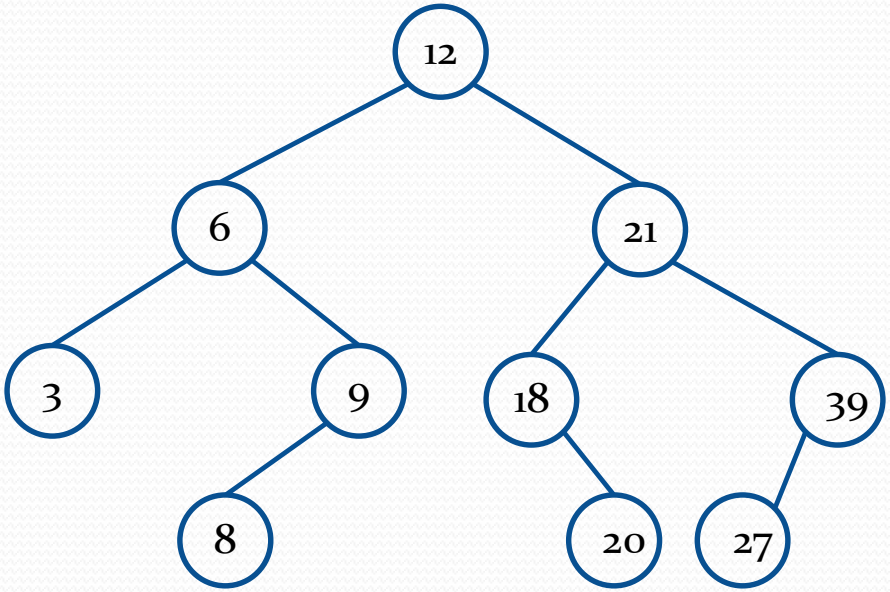
   Supprimer le nœud contenant cette plus petite valeur
   (comme à 2.)

   Remarque : On peut faire d'une façon similaire avec le SAG

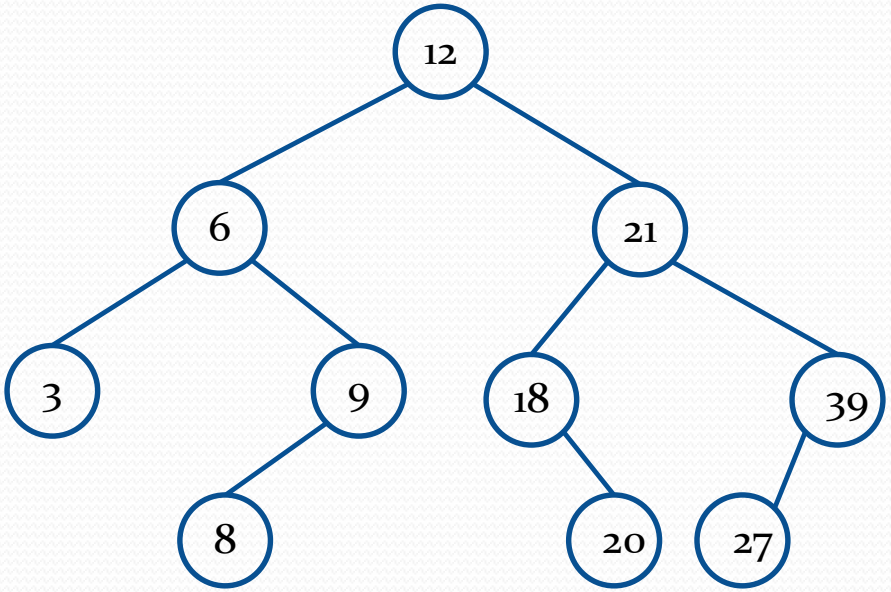
Supprimer 3



Supprimer 9



Supprimer 12



- Algorithme :

  x : nœud contenant l'élément à supprimer
  y : nœud à supprimer
  z : nœud fils de *y*
  parent : nœud parent de *y*

  Chercher *x* et son *parent*
  Si *x* a 0 ou 1 fils :
    y ← *x*
  Si *x* a 2 fils :
    y ← nœud contenant le min du SAD de *x*
    parent ← *parent* de *y*
  z ← nœud fils de *y*
  Mettre l'info de *y* dans *x*
  Attacher *z* à *parent*