Licence MIASHS

INF F5 — Arbres

UGA UFR SHS Université Grenoble Alpes

Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Généralités
- Arbre binaire
- Arbre binaire ordonné
- Arbre binaire équilibré
- Arbre n-aire

Généralités

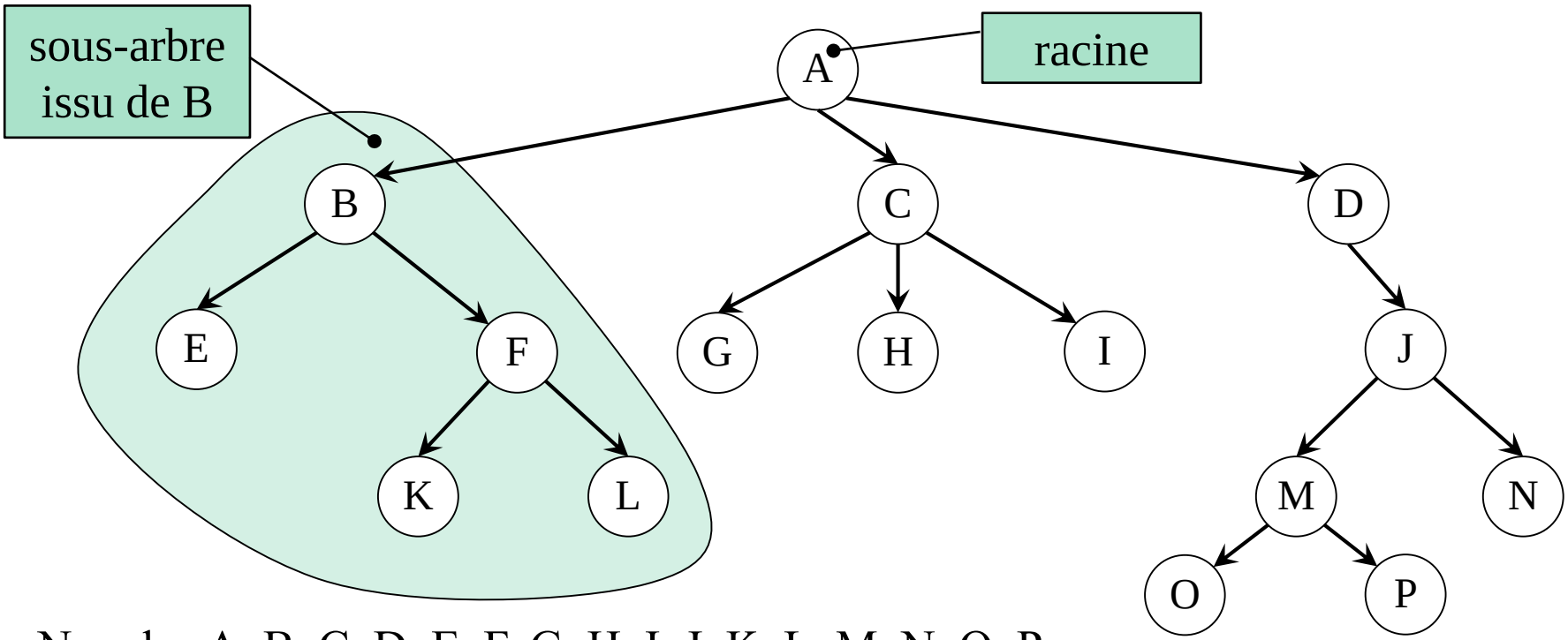
Arbre

- On appelle généralement « **arbre** » en informatique ce que l'on appelle « **arborescence** » en théorie des graphes.
- Une arborescence est graphe orienté acyclique possédant une racine unique
  - il existe un chemin unique de la racine vers n'importe quel sommet de l'arborescence.
- Les structures d'arbre sont très largement utilisées en informatique (structures de données, bases de données, système de fichiers, arbre de décision, …).

Arbre

- Les sommets d'un arbre sont appelés « **nœuds** ».
- La « **racine** » d'un arbre est l'unique nœud n'ayant pas de parent (ou prédécesseur).
- Les « **feuilles** » de l'arbre sont les nœuds qui n'ont pas de descendants (ou successeurs).
- La « **hauteur** » (ou profondeur) d'un arbre est la longueur du plus long chemin menant de la racine à l'une de ses feuilles.
- Chaque « **fils** » (ou successeur) d'un nœud non feuille engendre un « **sous-arbre** » ayant pour racine ce fils.

Exemple d'arbre



Nœuds : A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P

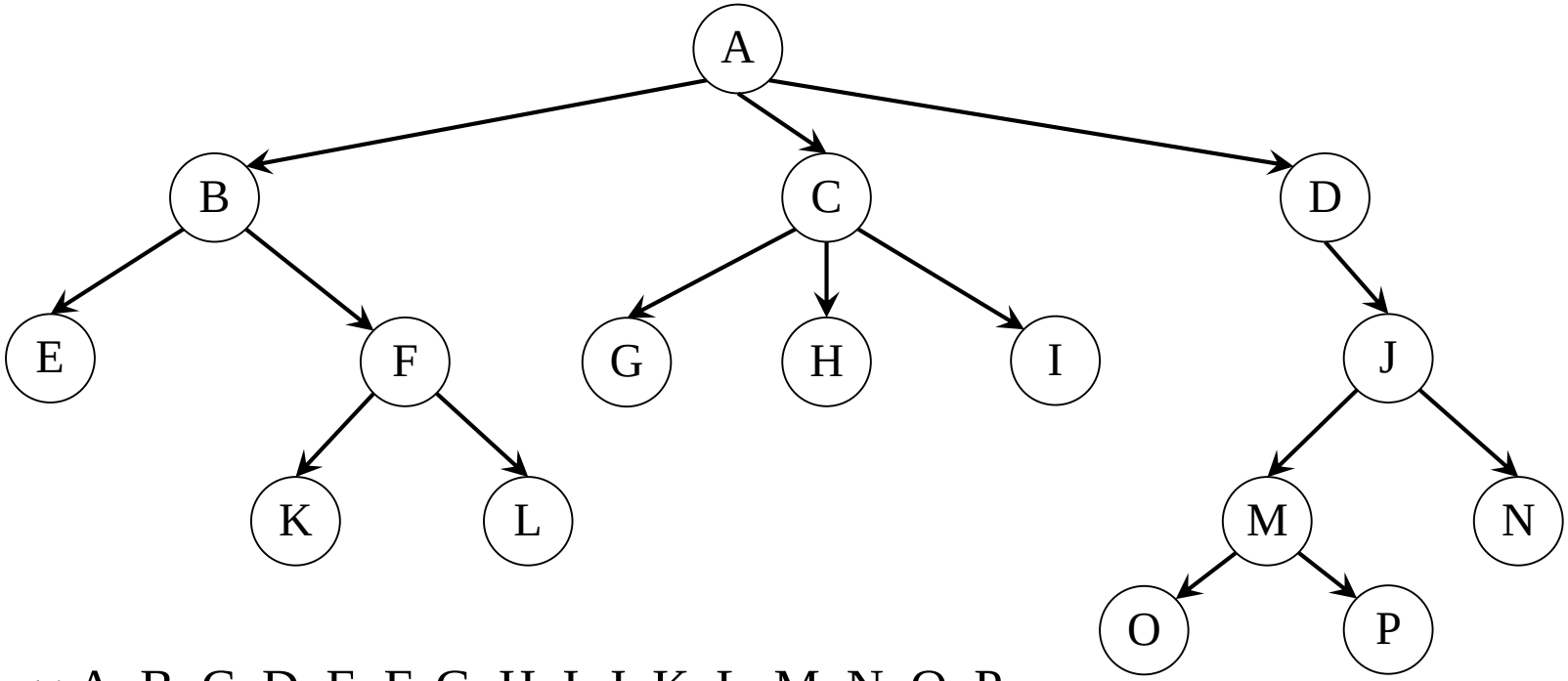
Feuilles : E, G, H, I, K, L, N, O, P

Hauteur : 5

Parcours d'arbre

- Parcours en largeur (par niveaux de profondeur)
  - on considère la racine,
  - puis les fils de la racine,
  - puis les fils des fils de la racine, …
- Parcours en profondeur
  - préfixé : un nœud est considéré chacun de ses fils (qui est considéré avant chacun de ses fils qui …).
  - postfixé : un nœud est considéré après chacun de ses fils (qui est considéré après chacun de ses fils qui …).

Exemple de parcours



Largeur : A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P

Préfixé : A, B, E, F, K, L, C, G, H, I, D, J, M, O, P, N

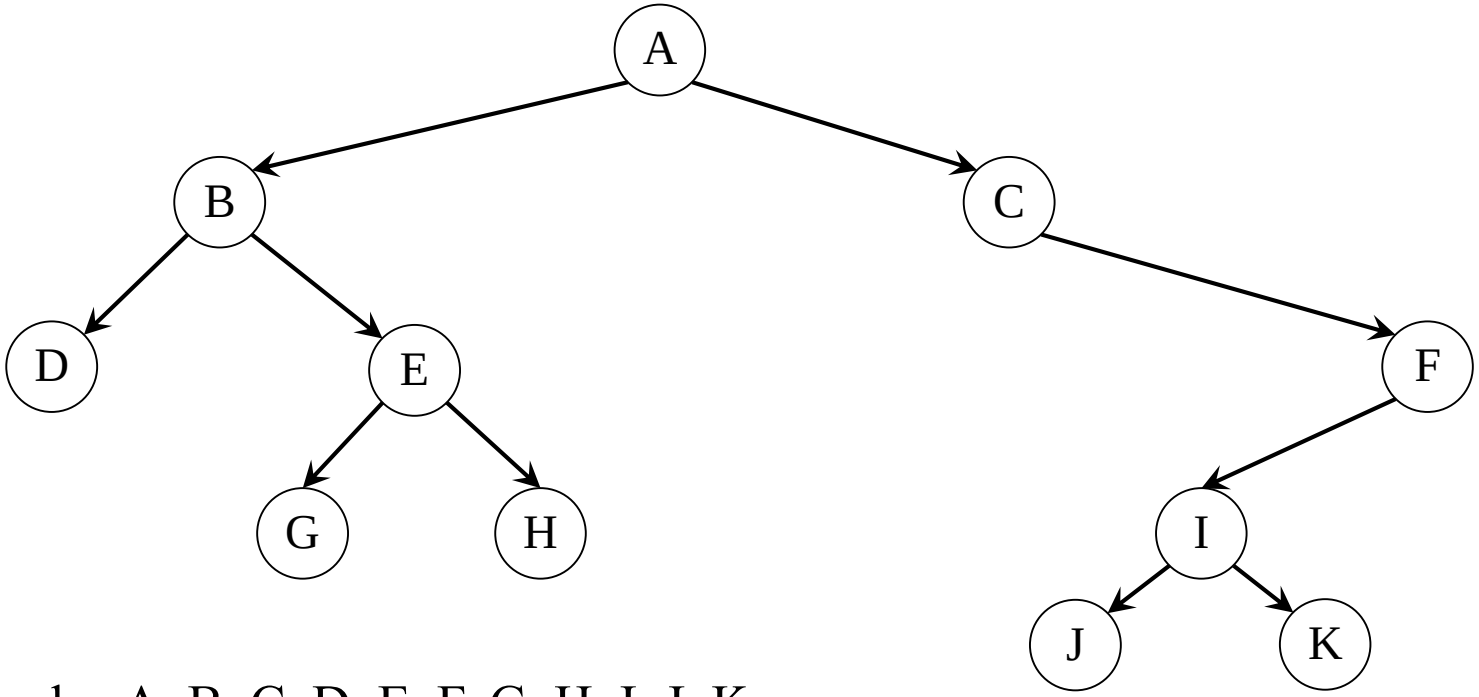
Postfixé : E, K, L, F, B, G, H, I, C, O, P, M, N, J, D, A

Arbre binaire

Arbre binaire

- Un **arbre binaire** est un arbre dans lequel les nœuds peuvent avoir au maximum 2 fils.
- Un arbre binaire de n nœuds a au plus $(n + 1) / 2$ feuilles.
- Un arbre binaire de n nœuds a une hauteur h telle que $\log_2(n + 1) \leq h \leq n$.

Exemple d'arbre binaire



12 nœuds : A, B, C, D, E, F, G, H, I, J, K

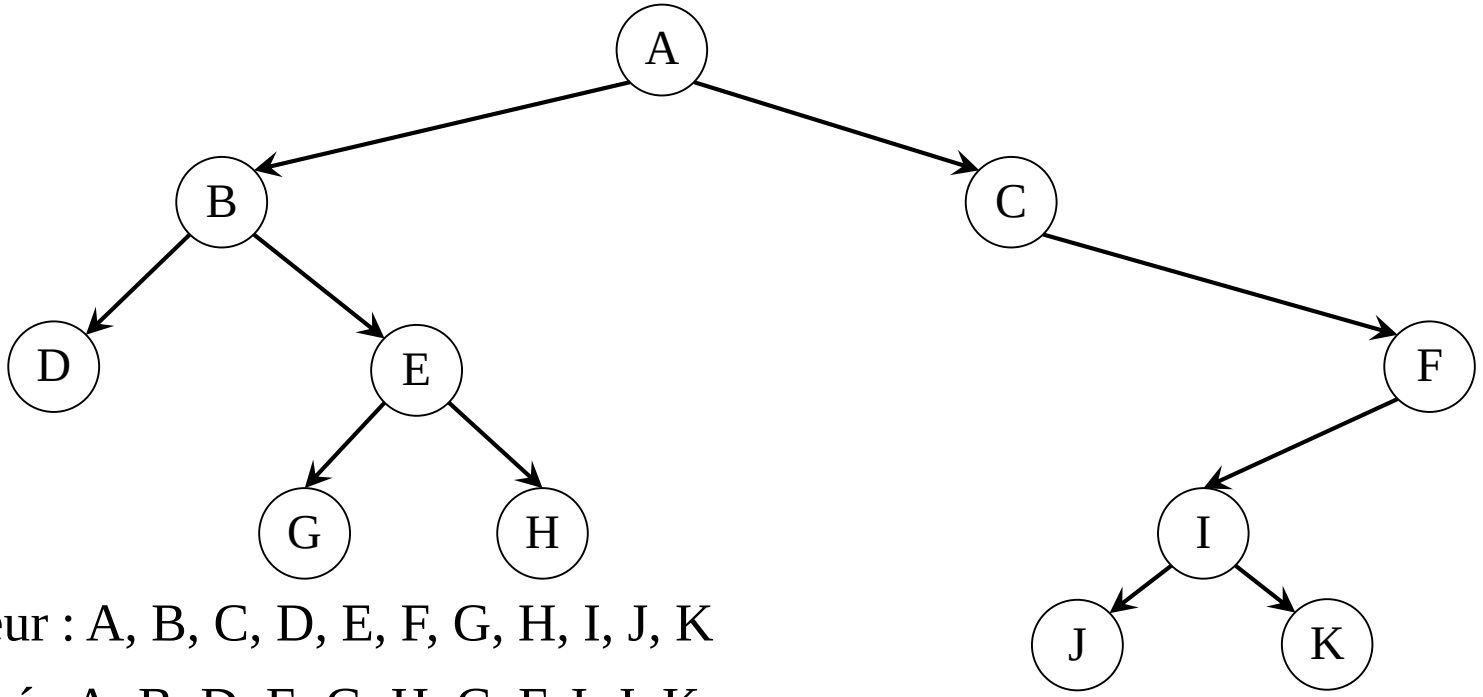
5 feuilles $\leq (12 + 1) / 2 = 6$

Hauteur : 5, $\log_2(12 + 1) = 3 \leq 4 \leq 12$

Parcours d'arbre binaire

- Parcours en largeur (par niveaux de profondeur)
- Aux 2 parcours en profondeur définis pour tous les arbres s'ajoute un nouveau parcours
  - préfixé,
  - postfixé,
  - infixé : on considère son sous-arbre gauche avant un nœud puis son sous-arbre droit.

Parcours d'arbre binaire



Largeur : A, B, C, D, E, F, G, H, I, J, K

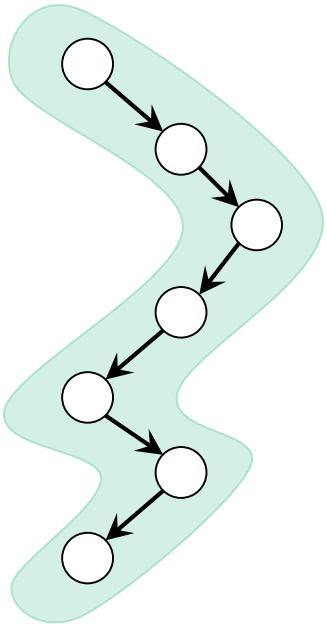
Préfixé : A, B, D, E, G, H, C, F, I, J, K

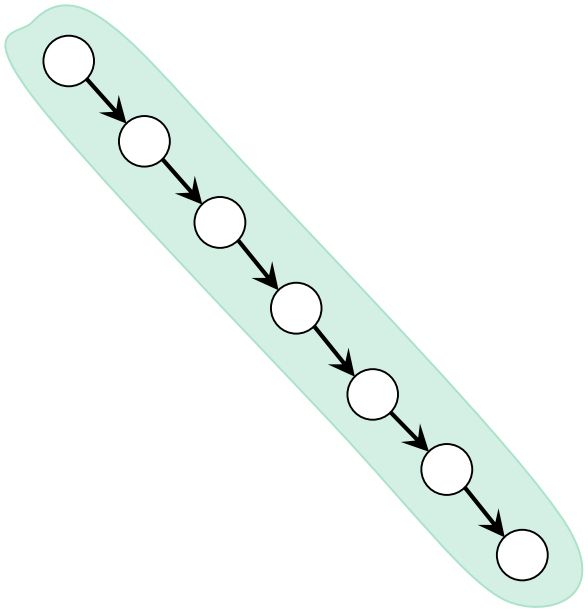
Postfixé : D, G, H, E, B, J, K, I, F, C, A

Infixé : D, B, G, E, H, A, C, J, I, K, F

Arbres binaires remarquables

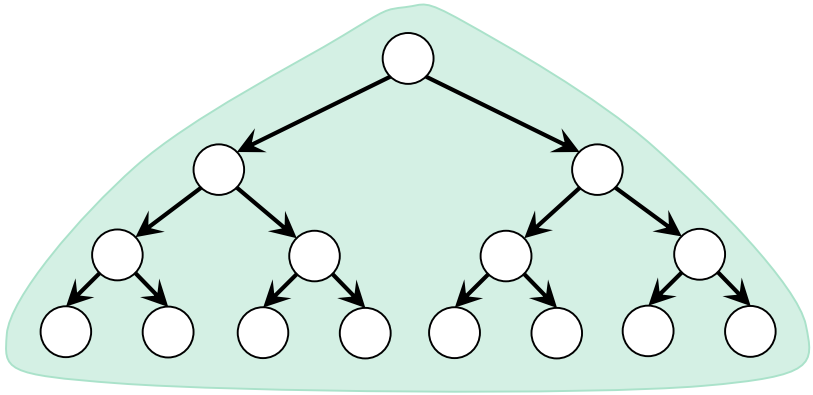
- **Arbre binaire dégénéré** : tous les nœuds sauf son unique feuille ont un seul fils. Il est entièrement constitué par un chemin allant de la racine vers son unique feuille. La hauteur est maximale.





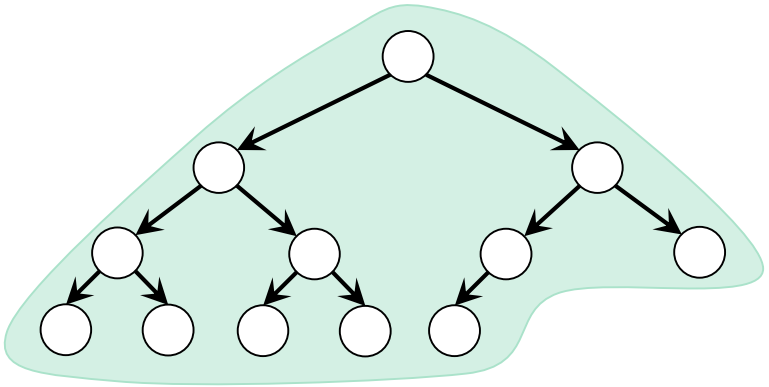
Arbres binaires remarquables

- **Arbre binaire complet** : il a exactement 2^h-1 nœuds, dont 2^{h-1} feuilles. Tous les niveaux sont remplis, ses nœuds non feuilles ont tous exactement 2 fils. La hauteur de l'arbre est minimale.



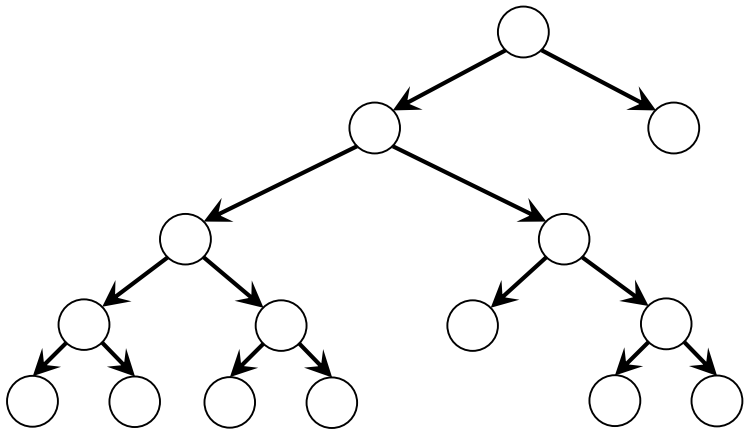
Arbres binaires remarquables

- **Arbre binaire parfait** : tous les niveaux sauf éventuellement le dernier sont remplis, les feuilles du dernier niveau sont groupées à gauche.



Arbres binaires remarquables

- **Arbre binaire homogène** (ou localement complet) : tous ses nœuds non feuilles ont exactement 2 fils.

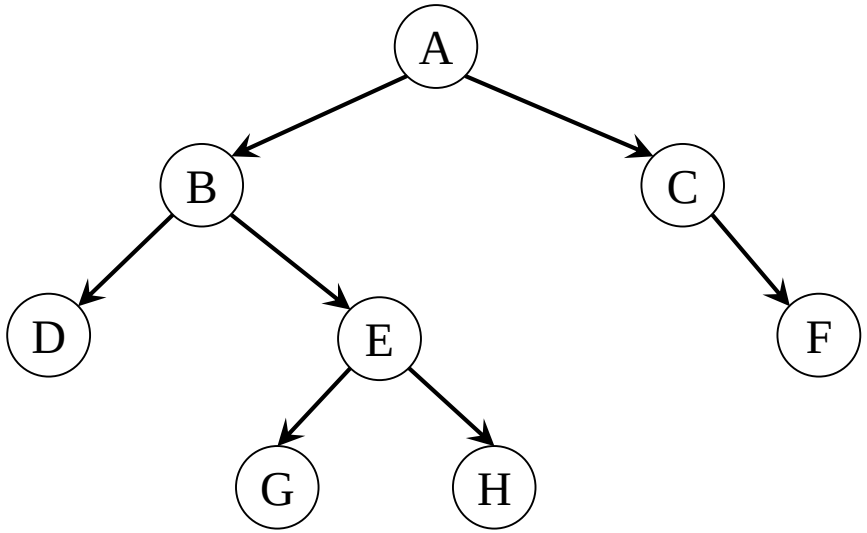


Programmation

- Les arbres binaires peuvent être représentés de différentes façons. Les deux les plus courantes sont :
  - la représentation contigüe, bien adaptée aux arbres complets ou parfaits.
  - la représentation chaînée adaptée à tous les types d'arbres.

Représentation contigüe

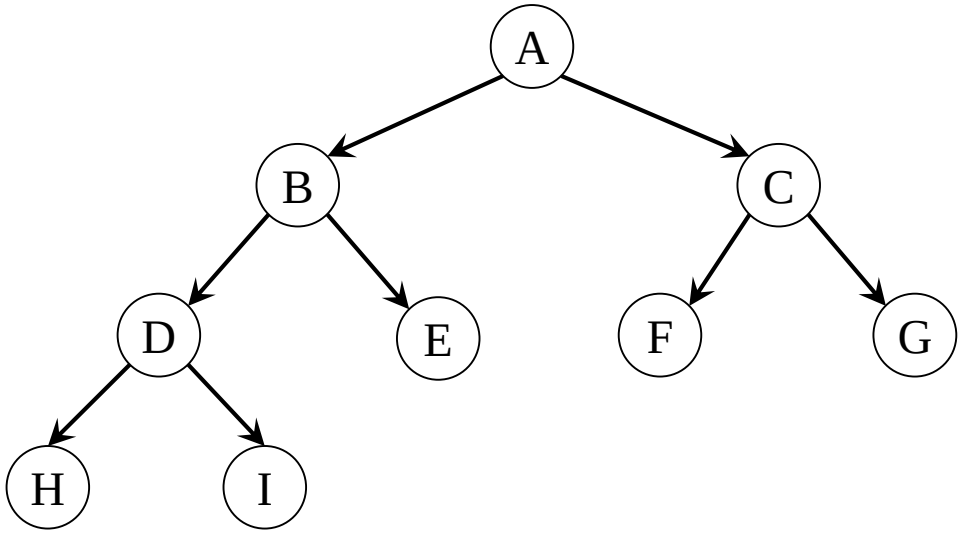
- Les nœuds sont stockés dans une structure indicée (tableau ou liste). La racine est stockée à l'indice 0. Le fils gauche et le fils droit du nœud stocké à l'indice *i* se trouvent respectivement à l'indice 2 * *i* + 1 et 2 * *i* + 2 s'ils existent.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	null	F	null	null	G	H	null	null	null	null	null

Représentation contigüe

- Pour un arbre binaire parfait, il n'y a pas de « trous » dans la structure indicée.



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	F	G	H	I	null	null	null	null	null	null	null

Représentation chaînée

- Chaque nœud est représenté par au moins 3 attributs : 1 pour la valeur (« étiquette ») et 2 pour les fils gauche et droit.
- ```
  public class Arbre<T> {
      private Nœud<T> racine;

      // ...
  }
  ```
- ```
  public class Noeud<T> {
      private T valeur;
      private Noeud<T> gauche;
      private Noeud<T> droit;

      // ...
  }
  ```

Parcours en largeur

```
public class Arbre<T> {
    // ...

    public void largeur() {
        if (racine == null)
            return;
        List<Noeud<T>> l = new ArrayList<>();
        l.add(racine);
        for (int i = 0; i <l.size(); i ++) {
            Noeud<T> fGauche = l.get(i).getGauche();
            Noeud<T> fDroit = l.get(i).getDroit();
            if (fGauche != null)
                l.add(fGauche);
            if (fDroit != null)
                l.add(fDroit);
        }
        for (Noeud<T> n : l)
            System.out.print(n.getValeur + " ");
}
```

Parcours préfixé

- ```
  public class Arbre<T> {
      // ...

      public void prefixe() {
          if (racine == null)
              return;
          racine.prefixe();
      }
  }
  ```

- ```
  public class Noeud<T> {
      //...

      public void prefixe() {
          System.out.print(valeur + " ");
          if (gauche != null)
              gauche.prefixe();
          if (droit != null)
              droit.prefixe();
      }
  }
  ```

Parcours infixé

- ```
  public class Arbre<T> {
      // ...

      public void infixe() {
          if (racine == null)
              return;
          racine.infixe();
      }
  }
  ```

- ```
  public class Noeud<T> {
      //...

      public void infixe() {
          if (gauche != null)
              gauche.infixe();
          System.out.print(valeur + " ");
          if (droit != null)
              droit.infixe();
      }
  }
  ```

Parcours postfixé

```
public class Arbre<T> {
    // ...

    public void postfixe() {
        if (racine == null)
            return;
        racine.postfixe();
    }
}
```

```
public class Noeud<T> {
    //...

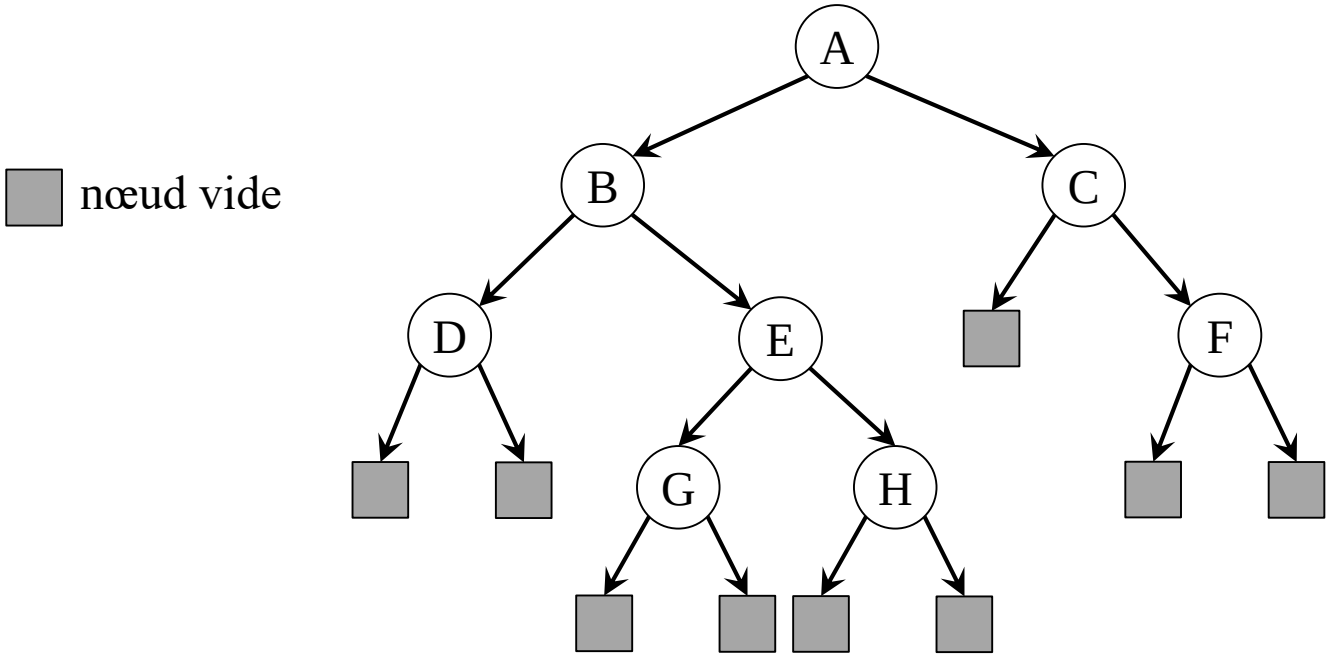
    public void postfixe() {
        if (gauche != null)
            gauche.postfixe();
        if (droit != null)
            droit.postfixe();
        System.out.print(valeur + " ");
    }
}
```

Cas des arbres homogènes

- Si l'arbre représenté est un arbre homogène, on peut définir une classe pour distinguer les feuilles.
- ```
  public class Arbre<T> {
      private NoeudAbstrait<T> racine;
      // ...
  }
  ```
- ```
  public abstract class NoeudAbstrait<T> {
      private T valeur;
      // ... comportement abstrait
  }
  ```
- ```
  public class Noeud<T> extends NoeudAbstrait<T> {
      private NoeudAbstrait<T> gauche;
      private NoeudAbstrait<T> droit;
      // ... comportement d'un nœud
  }
  ```
- ```
  public class Feuille<T> extends NoeudAbstrait<T> {
      // ... comportement d'une feuille
  }
  ```

Arbre binaire étendu

- Pour la représentation chaînée d'un arbre binaire, on peut considérer un type de nœud spécial correspondant au nœud vide.



Représentation chaînée avec des nœuds vides

- L'utilisation d'un type NoeudVide permet de s'affranchir de nombreux tests sur **null** : on s'appuie sur le polymorphisme.

```
public class Arbre<T> {
    NoeudAbstrait<T> racine;
    // ...
}
```

```
public abstract class NoeudAbstrait<T> {
    // ... comportement abstrait
}
```

```
public class Noeud<T> extends NoeudAbstrait<T> {
    private T valeur;
    private NoeudAbstrait<T> gauche;
    private NoeudAbstrait<T> droit;
    // ... comportement d'un nœud
}
```

```
public class NoeudVide<T> extends NoeudAbstrait<T> {
    // ... comportement d'une nœud vide « ne rien faire »
}
```

Simplification du parcours infixé avec des nœuds vides

```
public class Arbre<T> {
    // ...
    public void infixe() {
        racine.infixe();
    }
}
```

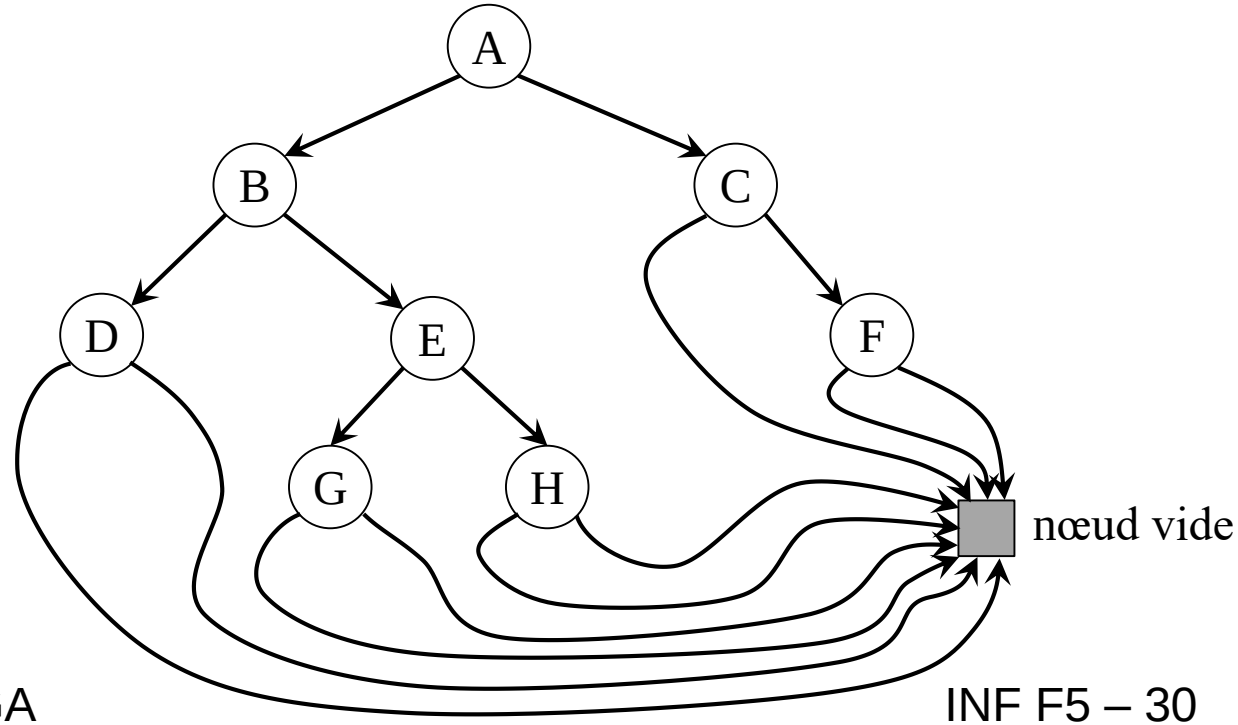
```
public abstract class NoeudAbstrait<T> {
    // ...
    public abstract void infixe();
}
```

```
public class Noeud<T> extends NoeudAbstrait<T> {
    // ...
    public void infixe() {
        gauche.infixe();
        System.out.print(valeur + " ");
        droit.infixe();
    }
}
```

```
public class NoeudVide<T> extends NoeudAbstrait<T> {
    // ...
    public void infixe() {
    }
}
```

Nœud vide unique

- Étant donné que le nœud vide ne porte aucune information spécifique, il peut être instance unique d'une classe « Singleton » afin d'optimiser l'utilisation de la mémoire.



NoeudVide singleton

- Pour s'assurer que NoeudVide a une instance unique, on rend son constructeur **private** et on prévoit un accesseur **static** vers l'instance unique.
-
```
public class NoeudVide<T> extends NoeudAbstrait<T> {
    private static NoeudVide instance = new NoeudVide();

    private NoeudVide() {
    }

    public static NoeudVide getInstance() {
        return instance;
    }

    // … comportement d'une nœud vide « ne rien faire »
}
```

NoeudVide singleton

- Dans les autres classes, on utilise l'appel à l'accesseur **static** plutôt qu'un appel au constructeur de NoeudVide.
-
```
public class Arbre<T> {
    private NoeudAbstrait<T> racine;

    public Arbre() {
        racine = NoeudVide.getInstance();
    }

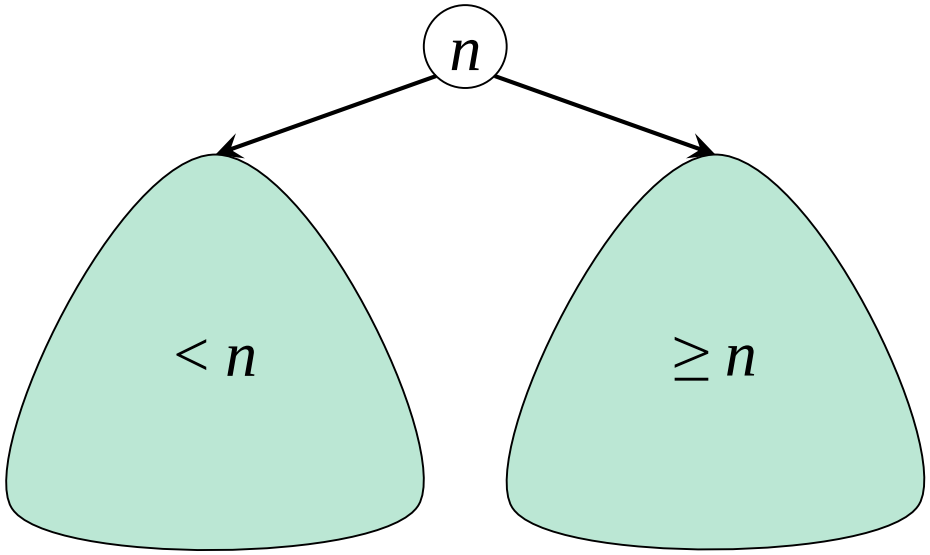
    public boolean isEmpty() {
        return racine instanceof NoeudVide;
    }

    // ...
}
```

Arbre binaire ordonné

Arbre binaire ordonné

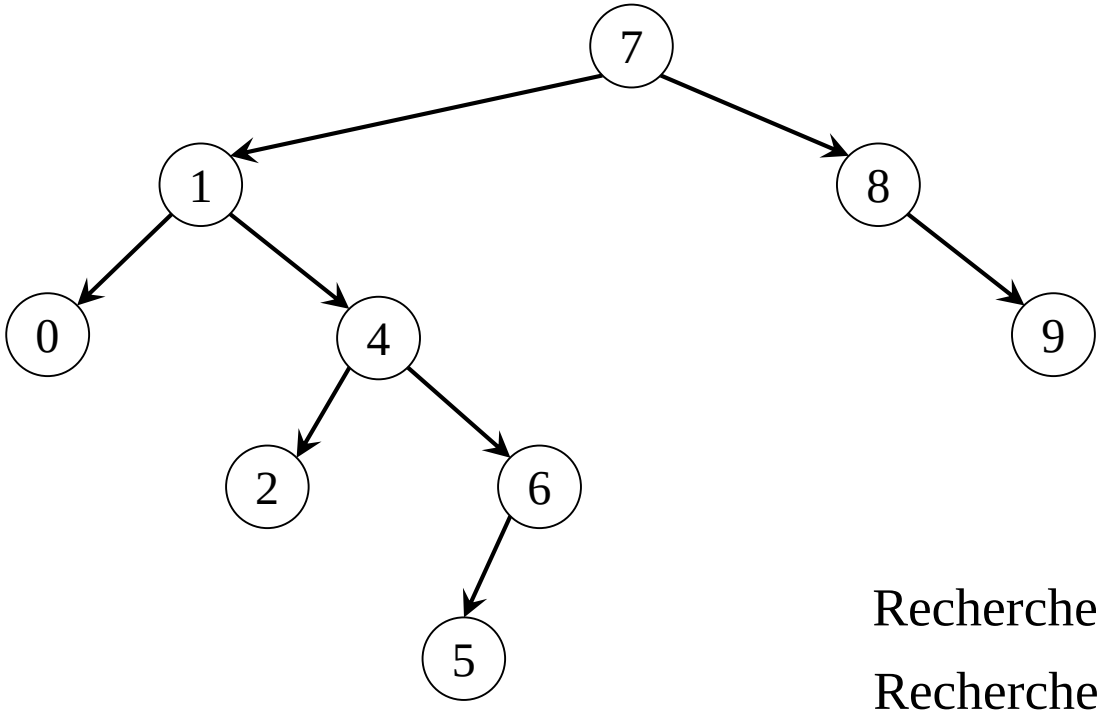
- S'il existe un ordre sur les valeurs de nœud, on peut construire un arbre binaire ordonné.
- Dans un **arbre binaire ordonné** (ou arbre binaire de recherche), tout nœud est tel que sa valeur est supérieure aux valeurs portés dans son sous-arbre gauche et inférieur aux valeurs portés par son sous-arbre droit.



Arbre binaire ordonné

- Un arbre binaire ordonné est une structure optimisée pour la recherche de valeur :
  - on compare la valeur cherchée à celle de la racine et on détermine si l'on doit explorer dans le sous-arbre gauche ou le sous-arbre droit en fonction du résultat de la comparaison,
  - on répète cette opération pour tous les nœuds rencontrés jusqu'à trouver la valeur recherchée ou arriver en bas de l'arbre.
- La recherche d'une valeur dépend donc uniquement de la hauteur de l'arbre : $\log_2(n + 1) \leq h \leq n$.

Recherche dans un arbre binaire ordonné



Recherche de 4 → trouvé

Recherche de 5 → trouvé

Recherche de 3 → non trouvé

Recherche dans un arbre binaire ordonné

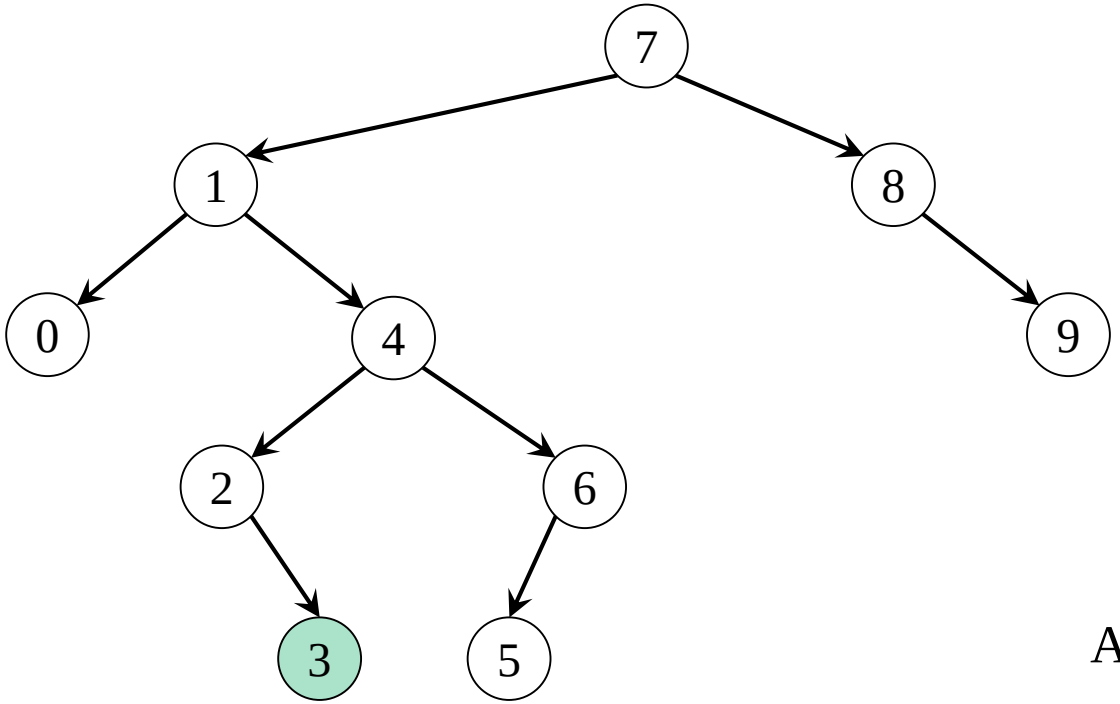
```
public class ArbreBinaireOrdonne<T extends Comparable<T>> {
    // ...
    public boolean contient(T v) {
        racine.contient(v);
    }
}
```

```
public abstract class NoeudAbstrait<T extends Comparable<T>> {
    // ...
    public abstract boolean contient(T v);
}
```

```
public class Noeud<T extends Comparable<T>> extends NoeudAbstrait<T> {
    // ...
    public boolean contient(T v) {
        int comp = v.compareTo(valeur);
        if (comp == 0)
            return true;
        return (comp < 0) ? gauche.contient(v) : droit.contient(v);
    }
}
```

```
public class NoeudVide<T extends Comparable<T>> extends NoeudAbstrait<T> {
    // ...
    public boolean contient(T v) {
        return false;
    }
}
```

Ajout dans un arbre binaire ordonné



Ajout de 3

Ajout dans un arbre binaire ordonné

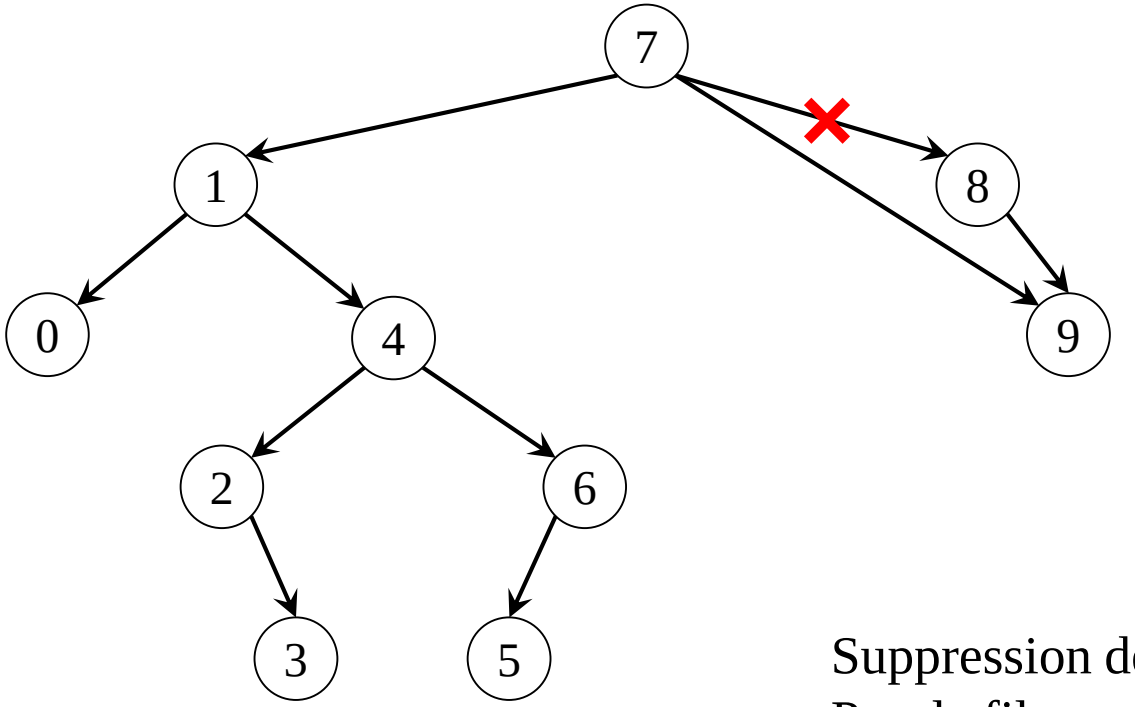
```
public class ArbreBinaireOrdonne<T extends Comparable<T>> {
    // ...
    public void ajout(T v) {
        racine = racine.ajout(v);
    }
}
```

```
public abstract class NoeudAbstrait<T extends Comparable<T>> {
    // ...
    public abstract NoeudAbstrait<T> ajout(T v);
}
```

```
public class Noeud<T extends Comparable<T>> extends NoeudAbstrait<T> {
    // ...
    public NoeudAbstrait<T> ajout(T v) {
        if (v.compareTo(valeur) < 0)
            gauche = gauche.ajout(v);
        droit = droit.ajout(v);
    }
}
```

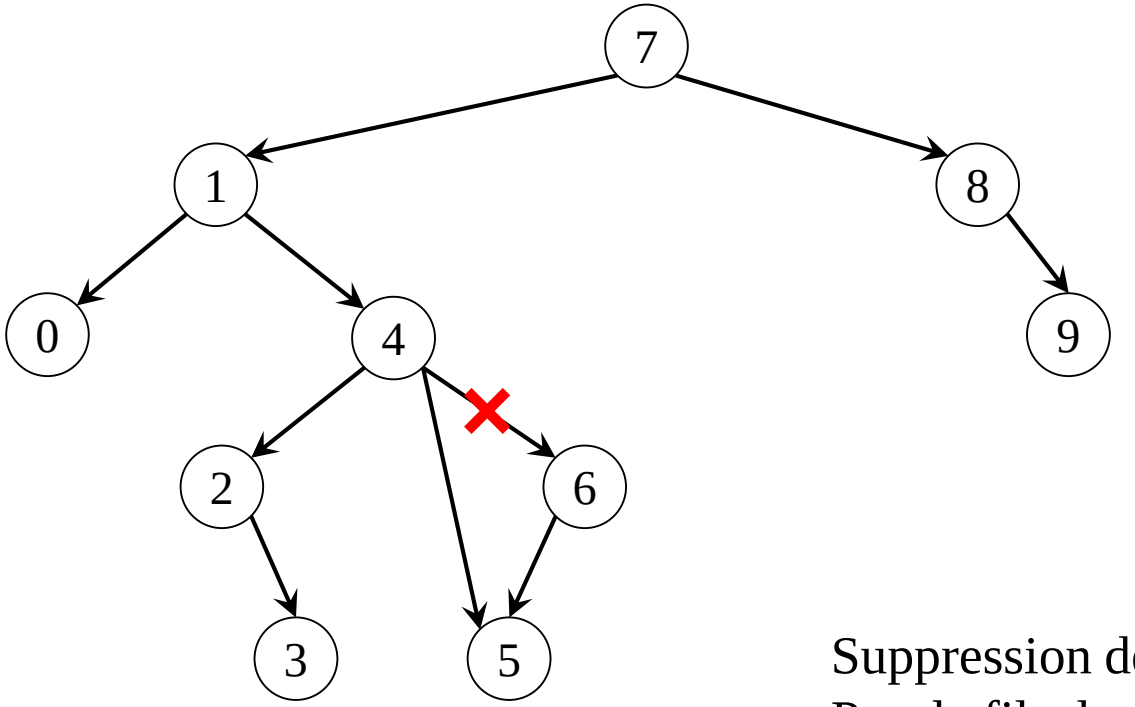
```
public class NoeudVide<T extends Comparable<T>> extends NoeudAbstrait<T> {
    // ...
    public NoeudAbstrait<T> ajout(T v) {
        return new Noeud(v, this, this);
    }
}
```

Suppression dans un arbre binaire ordonné



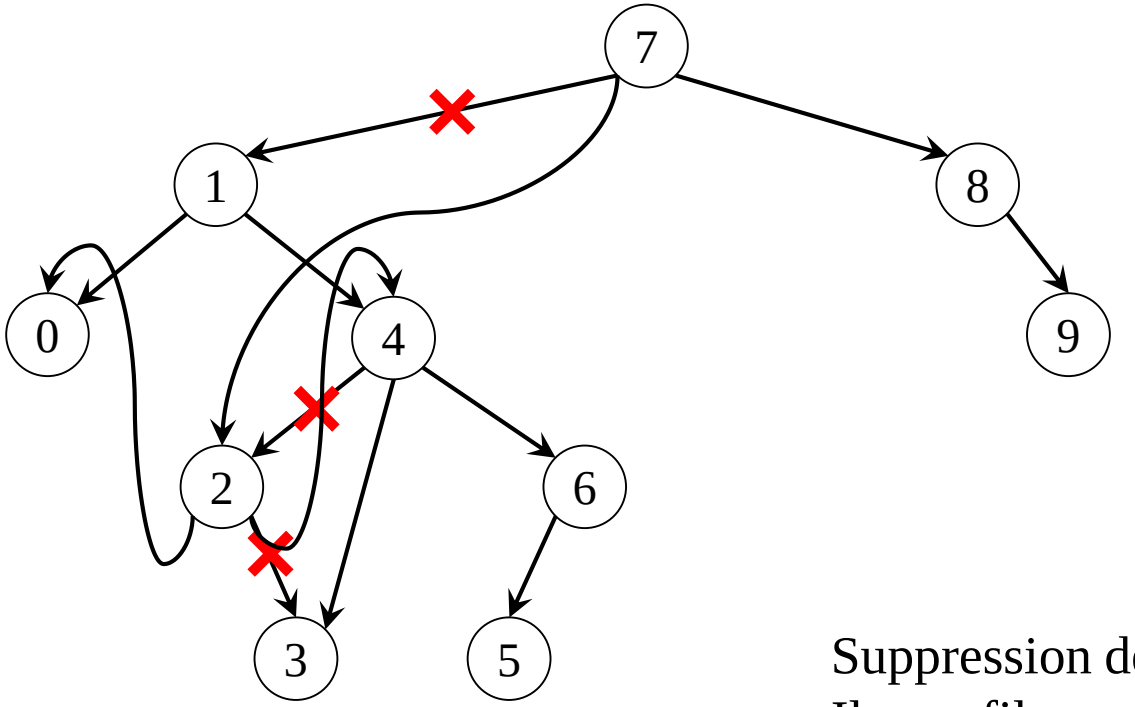
Suppression de 8 :
Pas de fils gauche, on le remplace par son fils droit

Suppression dans un arbre binaire ordonné



Suppression de 6 :
Pas de fils droit, on le remplace par son fils gauche

Suppression dans un arbre binaire ordonné



Suppression de 1 :
Il a un fils gauche et fils droit,
on le remplace par le nœud le plus
à gauche dans son sous-arbre droit

Suppression dans un arbre binaire ordonné

- ```
  public class ArbreBinaireOrdonne<T extends Comparable<T>> {
      // ...
      public void suppression(T v) {
          racine = racine.suppression(v);
      }
  }
  ```
- ```
  public abstract class NoeudAbstrait<T extends Comparable<T>>
  {
      // ...
      public abstract NoeudAbstrait<T> suppression(T v);
  }
  ```
- ```
  public class NoeudVide<T extends Comparable<T>> extends
                                                  NoeudAbstrait<T> {
      // ...
      public NoeudAbstrait<T> suppression(T v) {
          return this; // v non trouvée
      }
  }
  ```
  ```

Suppression dans un arbre binaire ordonné

```
public class Noeud<T extends Comparable<T>> extends NoeudAbstrait<T> {
    // ...
    public NoeudAbstrait<T> suppression(T v) {
        int comp = v.compareTo(valeur);
        if (comp < 0) {
            gauche = gauche.suppression(v);
            return this;
        }
        if (comp > 0) {
            droit = droit.ajout(v);
            return this;
        }
        if (gauche instanceof NoeudVide)
            return droit;
        if (droit instanceof NoeudVide)
            return gauche;
        // v est portée par this et a 2 fils
        // ...
    }
}
```

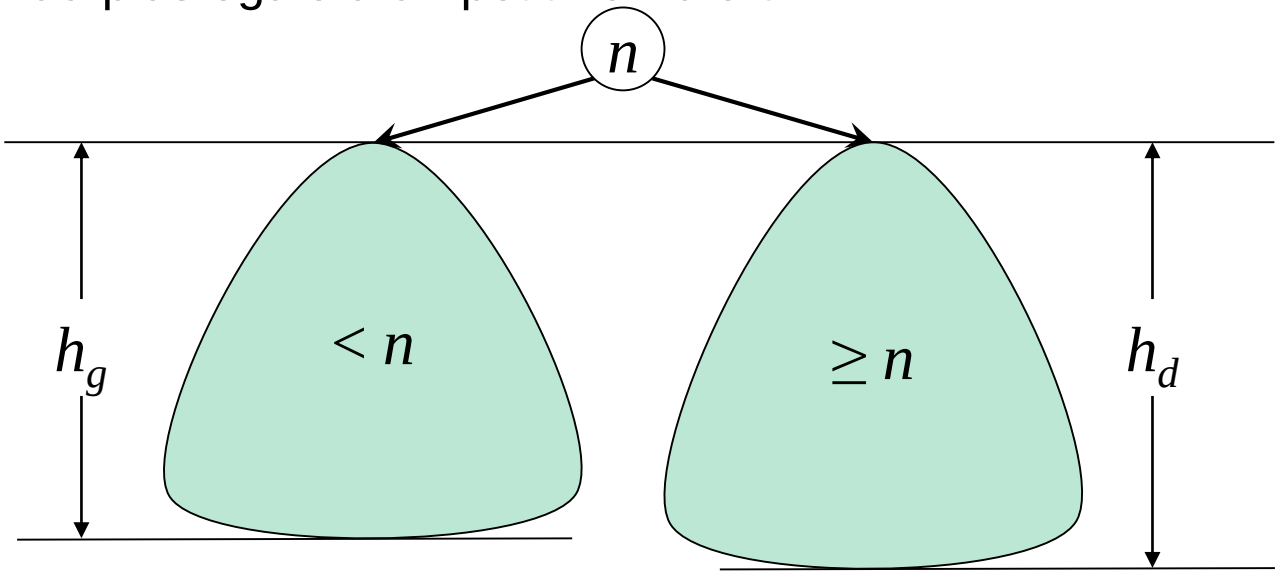
Suppression dans un arbre binaire ordonné

```
public class Noeud<T extends Comparable<T>> extends
                                                NoeudAbstrait<T> {
    // ...
    public NoeudAbstrait<T> suppression(T v) {
        // ...
        // v est portée par this qui a 2 fils, on le remplace par
        // le nœud le plus à gauche dans son sous-arbre droit
        Noeud<T> n  = (Noeud<T>) droit, pere = this;
        while (!(n.gauche instanceof NoeudVide)) {
            pere = n;
            n = (Noeud<T>) n.gauche;
        }
        if (pere != this) {
            pere.gauche = n.droit;
            n.droit = droit;
        }
        n.gauche = gauche;
        return n;
    }
}
```

Arbre binaire équilibré

Arbre binaire équilibré

- Le temps de recherche dans un arbre binaire ordonné est proportionnel à la hauteur de l'arbre, mais dans le pire des cas (celui d'un arbre dégénéré), cette hauteur est égale au nombre de nœuds.
- Un **arbre binaire équilibré** est un arbre binaire ordonné dans lequel tout nœud admet pour ses 2 sous-arbres engendrés une différence de hauteur au plus égale à un petit nombre ε.



$|h_g - h_d| \leq \varepsilon$

Arbre binaire équilibré

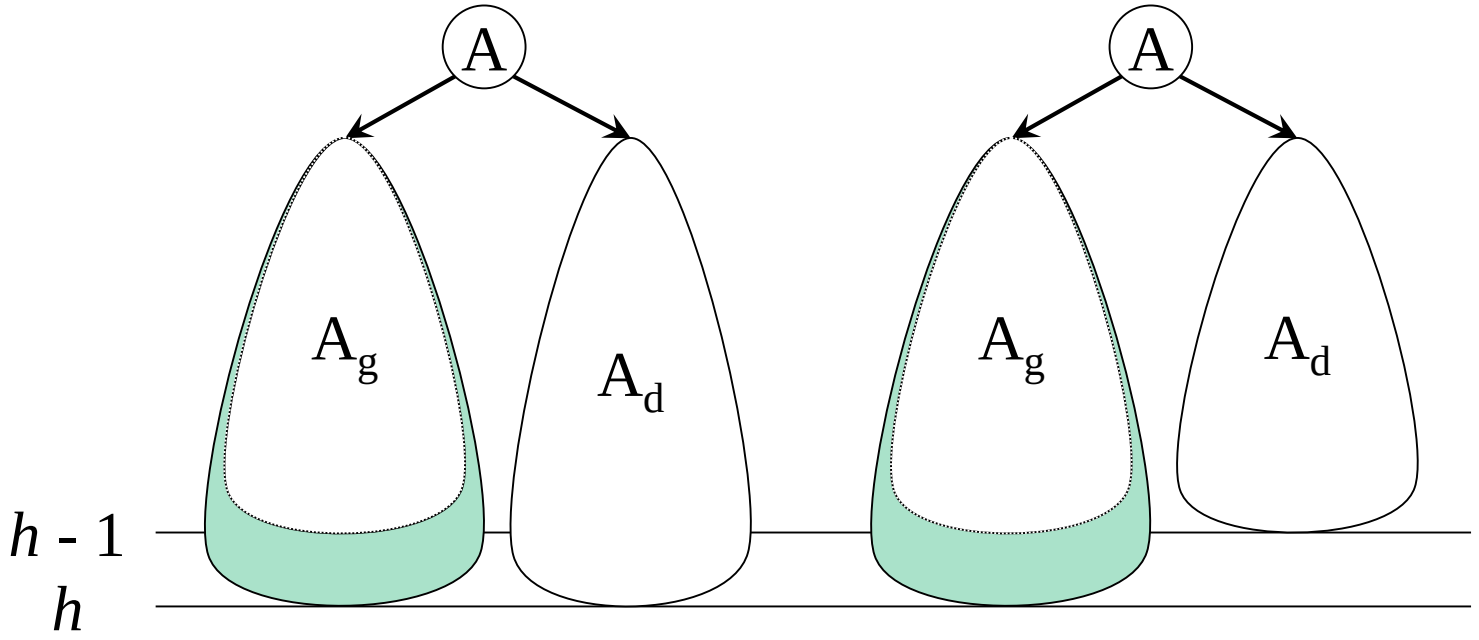
- On effectue une recherche dans un arbre équilibré de la même manière que dans un arbre binaire ordonné, mais la hauteur de l'arbre étant optimisée, elle est plus efficace.
- L'ajout ou la suppression de nœuds est également effectuée de façon similaire mais sont parfois suivis de ré-équilibrages.
- Il existe plusieurs techniques de ré-équilibrage :
  - arbres AVL (Adelson, Velskij, et Landis),
  - arbres rouge-noir (ou arbres bicolores).

Ré-équilibrage AVL après ajout à gauche

- Lorsqu'un ajout a eu lieu dans le sous-arbre gauche (A_g) d'un nœud A (précédemment équilibré), plusieurs cas peuvent se poser :
  1. La hauteur de A_g n'augmente pas, A reste équilibré.
  2. La hauteur de A_g augmente :
     a. La hauteur de A_g était 1 de moins que celle du sous-arbre droit (A_d), A est encore mieux équilibré.
     b. La hauteur de A_g était égale à celle de A_d, A reste équilibré.
     c. La hauteur de A_g était 1 de plus que celle de A_d, un ré-équilibrage est nécessaire.

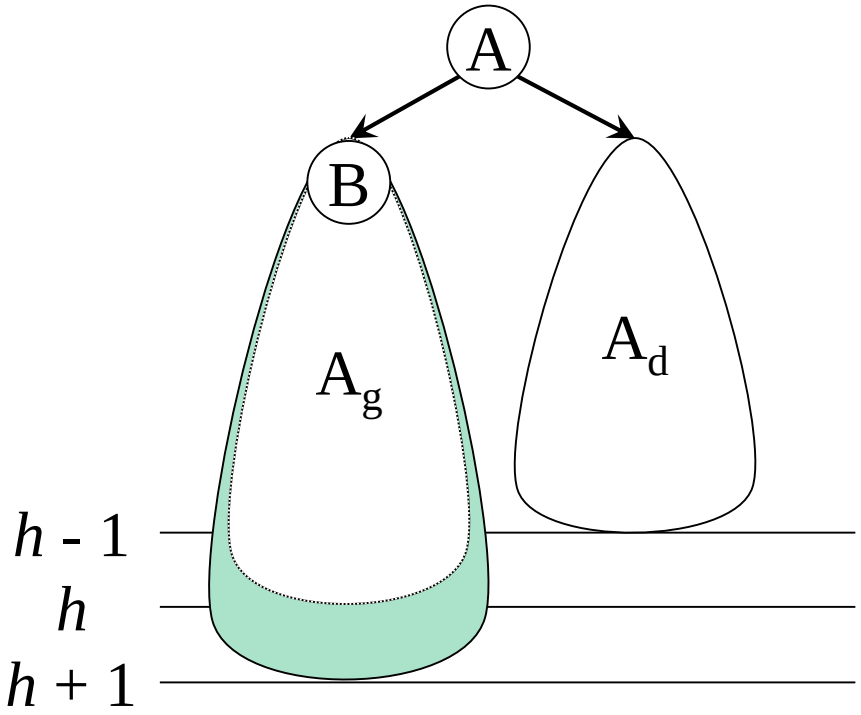
Ré-équilibrage AVL après ajout à gauche

- Cas 2-a et 2-b où aucun ré-équililbrage n'est nécessaire.



Ré-équilibrage AVL après ajout à gauche

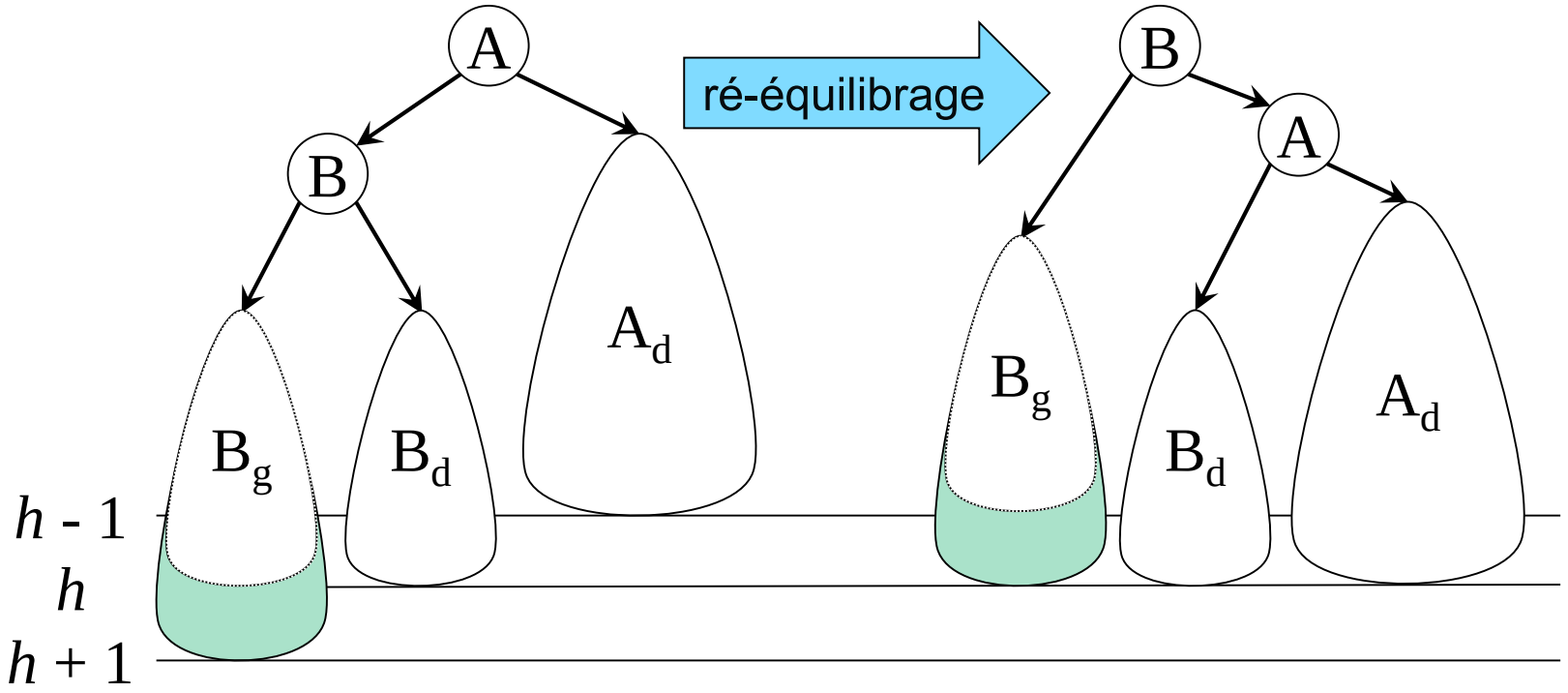
- Cas 2-c, un ré-équililbrage est nécessaire. On doit considérer B la racine de A_g. 2 nouveaux sous-cas à considérer.



Cas 2-c

Ré-équilibrage AVL après ajout à gauche

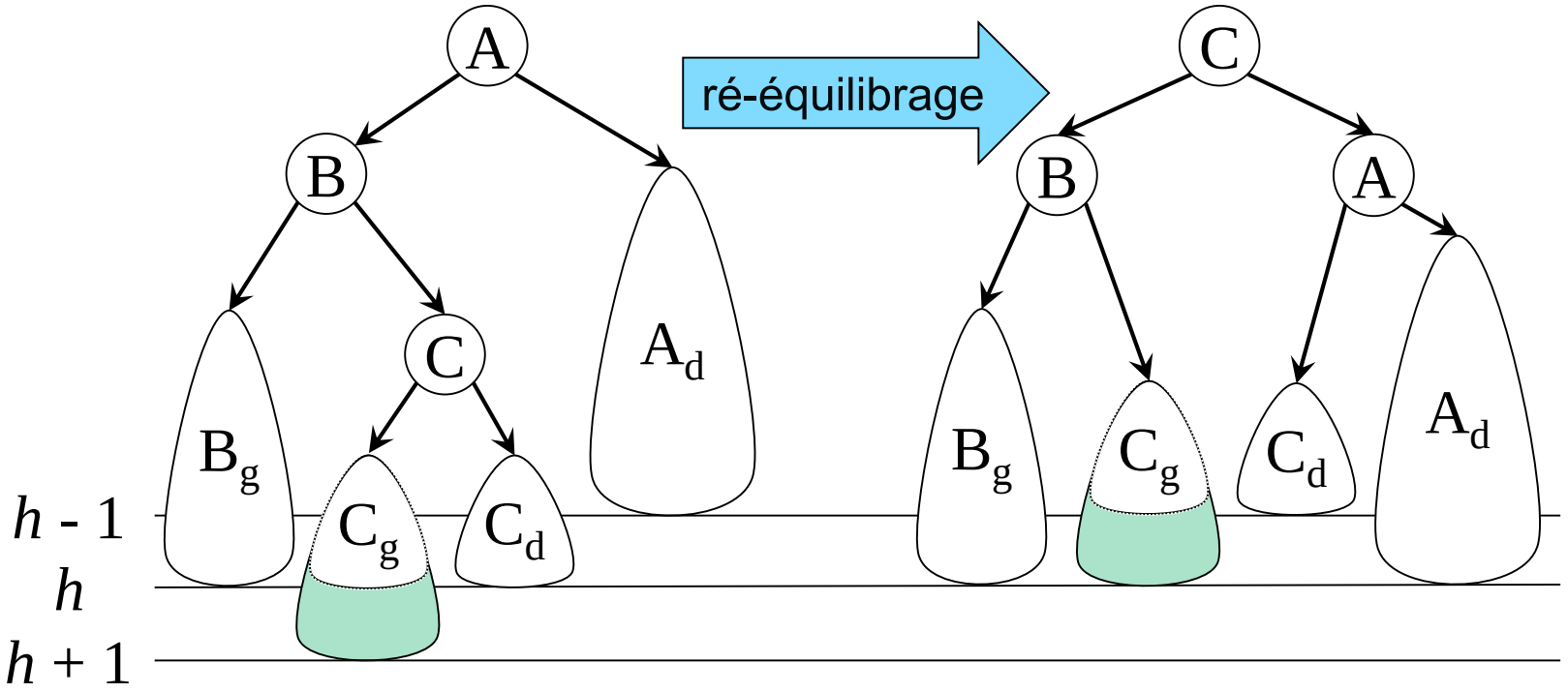
- Cas 2-c-1, B_g est plus haut que B_d après ajout.



Cas 2-c-1

Ré-équilibrage AVL après ajout à gauche

- Cas 2-c-2, B_g est moins haut que B_d après ajout.



Cas 2-c-2

Arbre AVL

- Les ré-équilibrages après ajout à droite sont considérés de manière symétrique aux ré-équilibrages après ajout à gauche.
- Des ré-équilibrages peuvent être aussi nécessaires après suppression.
- Les opérations d'ajout et de suppression consistent essentiellement à déterminer si la modification intervient à gauche ou à droite d'un nœud, puis à invoquer les méthodes de ré-équilibrages correspondantes.
- On peut ajouter à la représentation d'un nœud un attribut correspondant à la différence (soit -1, 0, ou 1) entre la hauteur de son sous-arbre gauche et celle de son sous-arbre droit pour plus d'efficacité.

Arbre n-aire

Arbre n-aire

- Un arbre *n*-aire est un arbre dans lequel les nœuds peuvent avoir jusqu'à *n* fils.
- On le représente généralement en utilisant l'une des deux techniques suivantes
  - les fils d'un nœud sont stockés dans une structure séquentielle (tableau, liste, …).
  - l'arbre n-aire n'est pas directement représenté, mais un arbre binaire équivalent est représenté.
- Optionnellement, on peut aussi référencer le nœud père d'un nœud (qu'il s'agisse d'un arbre binaire ou d'un arbre n-aire).

Nœuds avec une liste de fils

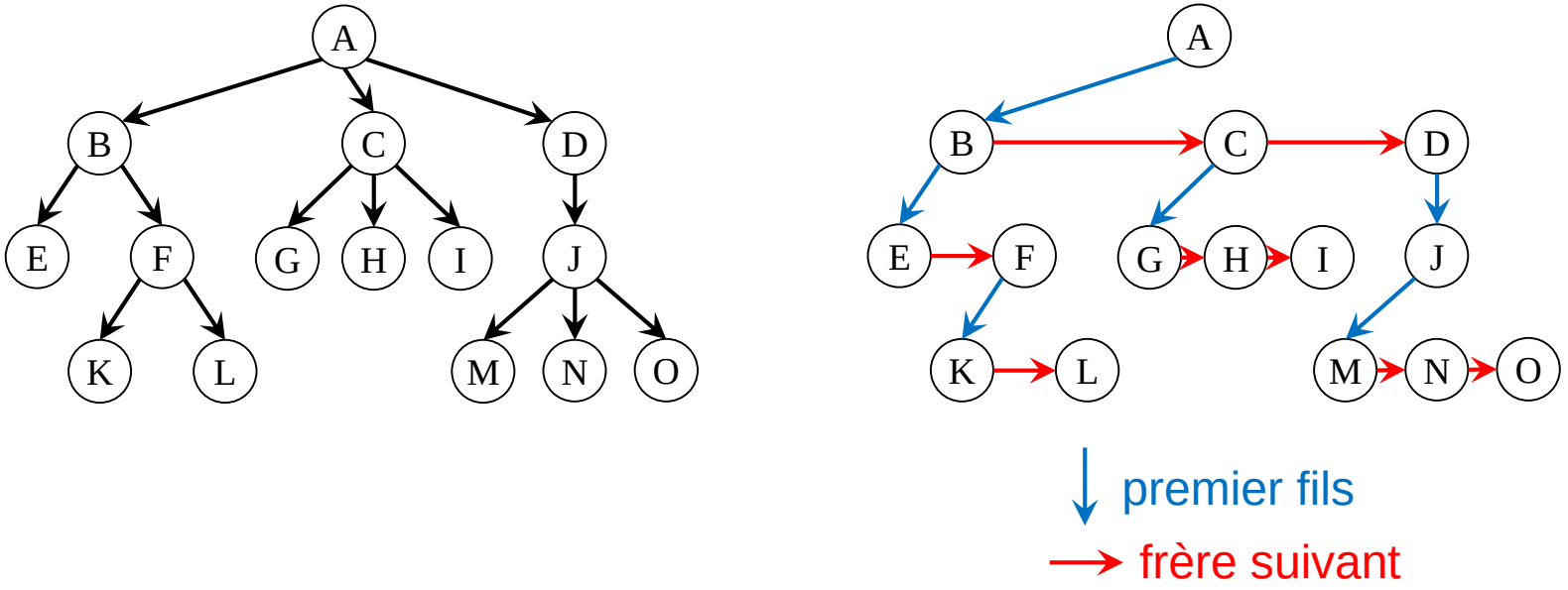
- Chaque nœud est représenté par au moins 2 attributs : 1 pour la valeur (« étiquette ») et 1 pour la liste de ses fils.
- ```
  public class Arbre<T> {
      private Nœud<T> racine;

      // ...
  }
  ```
- ```
  public class Noeud<T> {
      private T valeur;
      private List<Noeud<T>> enfants;
      private Noeud<T> père; // optionnel

      // ...
  }
  ```

Représentation par un arbre binaire équivalent

- On interprète l'arbre n-aire comme un arbre binaire : deux liens de types différents pour chaque nœud.



Représentation par un arbre binaire équivalent

```
public class Arbre<T> {
    private Nœud<T> racine;

    // …
}
```

```
public class Noeud<T> {
    private T valeur;
    private Noeud<T> premierFils;
    private Noeud<T> frereSuivant;
    private Noeud<T> père; // optionnel

    // …
}
```