Université Claude Bernard Lyon 1

# LIFAPSD – Algorithmique, Programmation et Structures de données

Nicolas Pronost

LIRIS

# Chapitre 8
# Arbre

# Définition d'un arbre

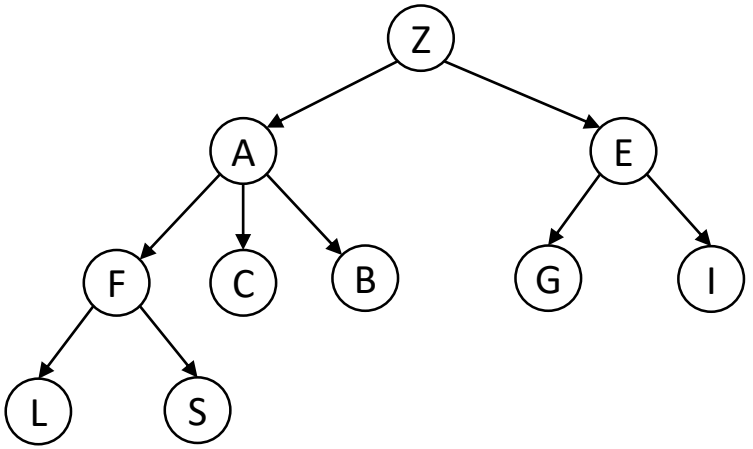
- Un arbre est une structure de donnée hiérarchique, composé de nœuds et de relations de précédence entre ces nœuds
- Chaque nœud possède
  - 0, 1, 2, ..., n successeur(s)
  - un et un seul prédécesseur (sauf la racine qui en a aucun)
- Un nœud ne peut pas être à la fois prédécesseur et successeur d'un autre nœud

# Vocabulaire

- La racine est le nœud sans prédécesseur (point d'accès au contenu de l'arbre)
- Une feuille est un nœud sans successeur
- Une branche est la suite des nœuds liant la racine à une feuille
- Un fils est un successeur d'un nœud
- Le père est le prédécesseur d'un nœud
- Le degré d'un nœud est le nombre de fils de ce nœud
- La profondeur d'un nœud est le nombre de prédécesseur entre ce nœud et la racine
- La hauteur d'un arbre est la profondeur maximale de tous les nœuds

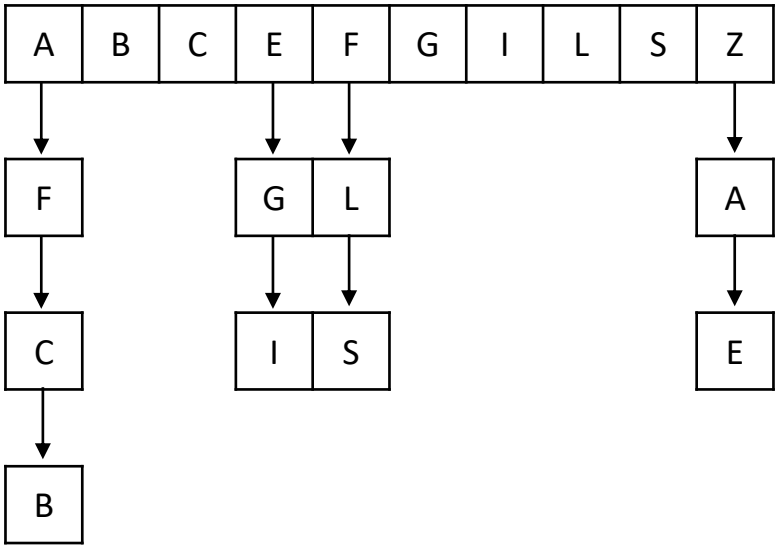
# Exemple d'un arbre n-aire

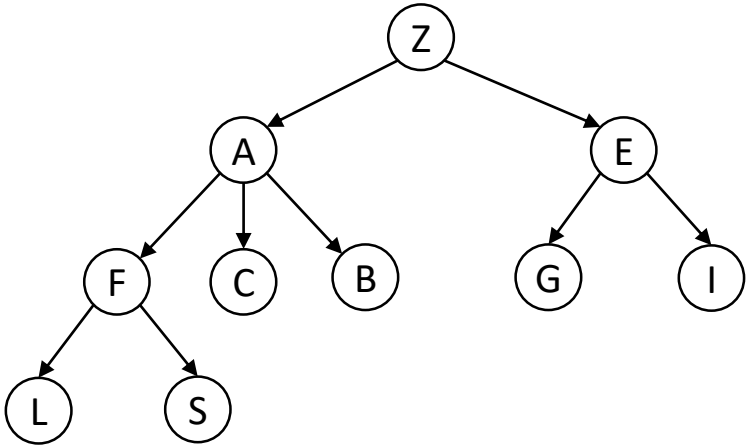
- Racine (arbre) = Z
- Feuilles (arbre) = {L S C B G I}
- Branche (S) = {S F A Z}
- Fils (A) = {F C B}
- Père (F) = A
- Degré (A) = 3
- Profondeur (C) = 2
- Hauteur (arbre) = 3



# Représentation d'un arbre

- Par liste d'adjacence

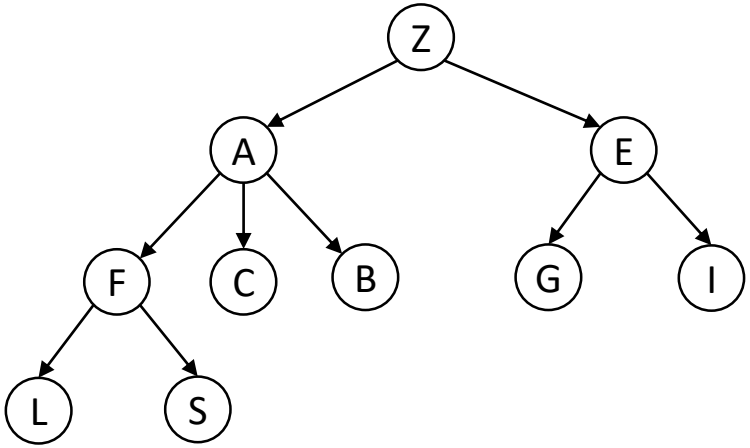


# Représentation d'un arbre

- Par tableau 2D

| | A | B | C | E | F | G | I | L | S | Z |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| B | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| E | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| G | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| L | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| S | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Z | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |




Le nœud de la ligne a comme fils le nœud de la colonne

# Représentation d'un arbre

- Comparaison entre liste d'adjacence et tableau 2D
  - Espace mémoire
    - Dans tous les cas, la liste prend en mémoire deux fois le nombre de nœuds moins 1 (la racine n'a pas de prédécesseur) : $O(n)$
    - Dans tous les cas, le tableau prend en mémoire : $O(n^2)$
  - Rechercher toutes les relations père-fils
    - Dans tous les cas, il faut parcourir tous les liens de la liste pour trouver toutes les relations : $O(n)$
    - Dans tous les cas, il faut parcourir tout le tableau pour trouver toutes les relations : $O(n^2)$

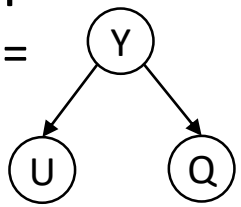
# Représentation d'un arbre

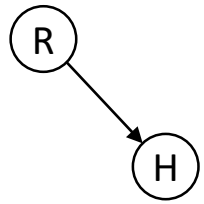
- Comparaison entre liste d'adjacence et tableau 2D
  - Supprimer une feuille
    - Il faut trouver la feuille à supprimer et mettre à jour la liste
      - recherche de la feuille à supprimer et de son père en $O(n)$ + suppression de la feuille dans la liste d'adjacence en $O(1) = O(n) + O(1) = O(n)$
    - Il faut trouver et supprimer la ligne et la colonne (même indice) de la feuille à supprimer et recopier les lignes/colonnes suivantes
      - recherche de la feuille à supprimer en $O(n)$ + suppression avec recopie en $O(n^2)$ dans le pire des cas (première ligne/colonne à supprimer)
      - $O(n) + O(n^2) = O(n^2)$

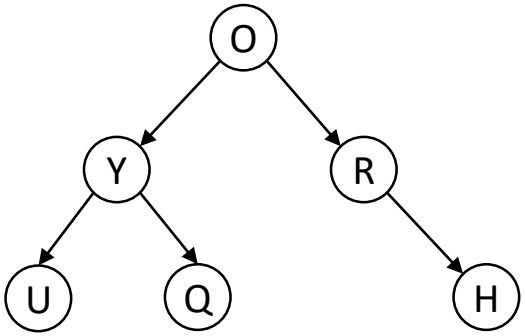
# Représentation d'un arbre

- Comparaison entre liste d'adjacence et tableau 2D
  - Ajouter un fils à un nœud donné
    - Il faut ajouter une nouvelle cellule à la liste des nœuds et une nouvelle cellule à la liste du nœud donné
      - ajout de la nouvelle cellule en tête de liste en $O(1)$
      - ajout de la nouvelle cellule en tête de liste des fils du nœud donné en $O(1)$
      - $O(1) + O(1) = O(1)$
    - Il faut ajouter une ligne et une colonne au tableau, mettre les éléments à 0 sauf pour le père qui est à 1
      - ajout d'une ligne et d'une colonne à 0 en $O(n^2)$ avec un tableau statique (en $O(n)$ amorti avec un TableauDynamique)
      - mettre à 1 le père en $O(1)$
      - $O(n^2) + O(1) = O(n^2)$

# Arbre binaire

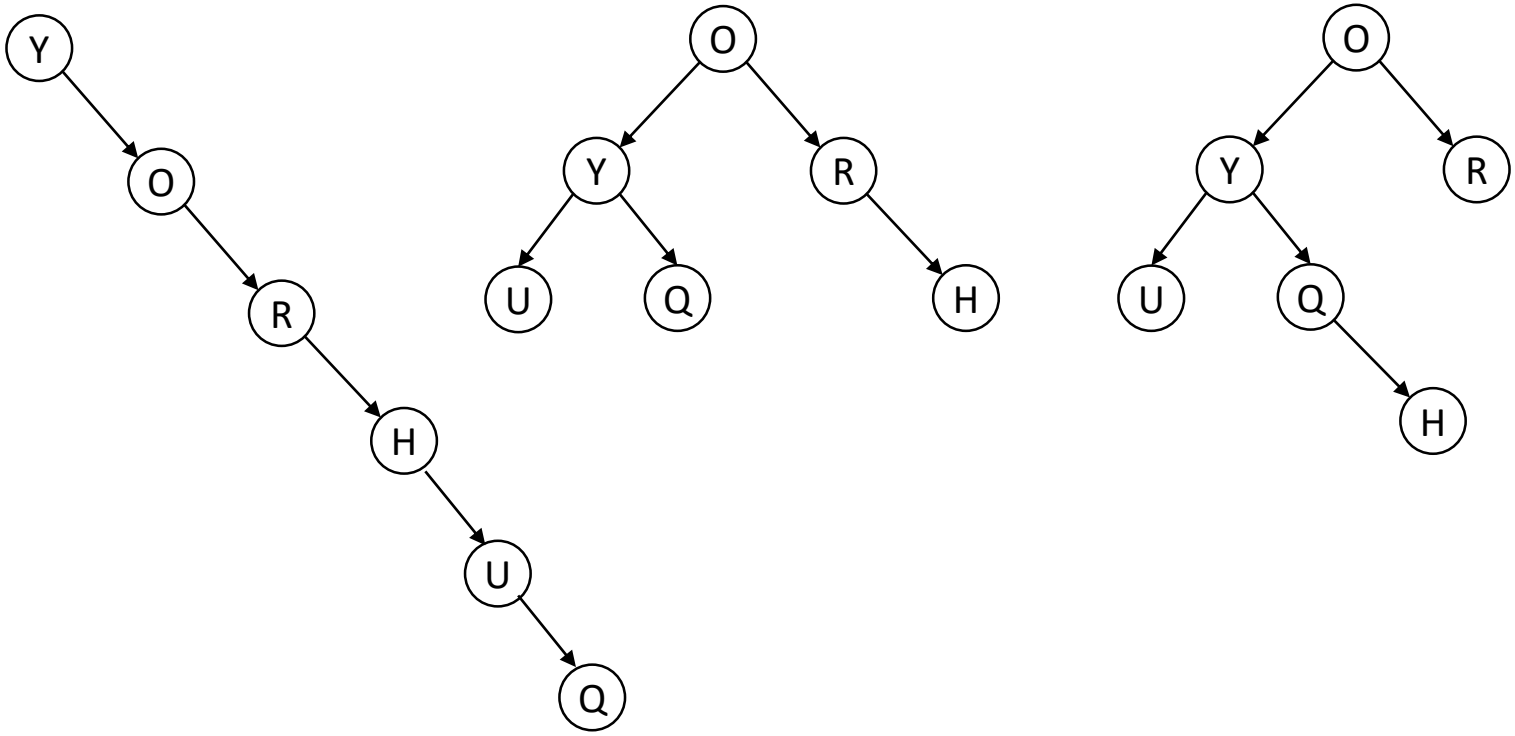
- Un arbre binaire est un arbre qui a au plus 2 fils (i.e. 0, 1 ou 2)
  - Le degré maximal d'un nœud est 2
- On appelle fils gauche (ou sous arbre gauche ou sag) le premier successeur
  - Fils gauche (O) = 

- On appelle fils droit (ou sous arbre droit ou sad) le deuxième successeur
  - Fils droit (O) = 

# Arbre binaire

- Un arbre binaire peut être dégénéré ou équilibré ou aucun des deux



# Module Arbre (binaire)

**Module** Arbre

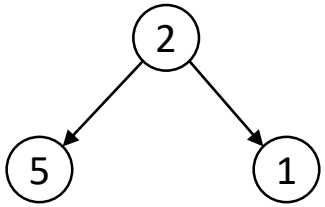
- **Importer:**
  - Module ElementA
- **Exporter:**
  - Type Noeud
  - Type Arbre
    - Constructeur Arbre()
      - Postconditions : l'arbre est initialement vide
    - Destructeur ~Arbre()
      - Postconditions : libération de la mémoire utilisée sur le tas, l'arbre est vide
    - Procédure vider ()
      - Postcondition : l'arbre ne contient plus aucun élément
    - Fonction estVide () : booléen
      - Résultat : retourne vrai si l'arbre est vide, faux sinon
    - Procédure afficher ()
      - Postcondition : l'arbre est affiché sur la sortie standard
    - Procédure insererElement (e : ElementA)
      - Postcondition : si e n'existe pas déjà dans l'arbre, alors un nouveau noeud contenant e est inséré, si e existe déjà dans l'arbre, alors l'arbre est inchangé
    - Fonction hauteurArbre () : entier
      - Résultat : la hauteur de l'arbre (longueur de sa plus longue branche), ou -1 s'il est vide

# Mise en œuvre d'un arbre en C++

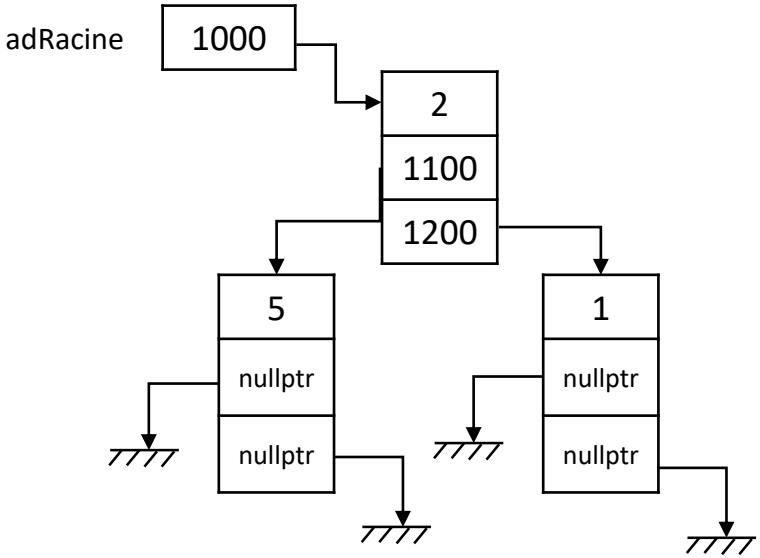
Arbre.h

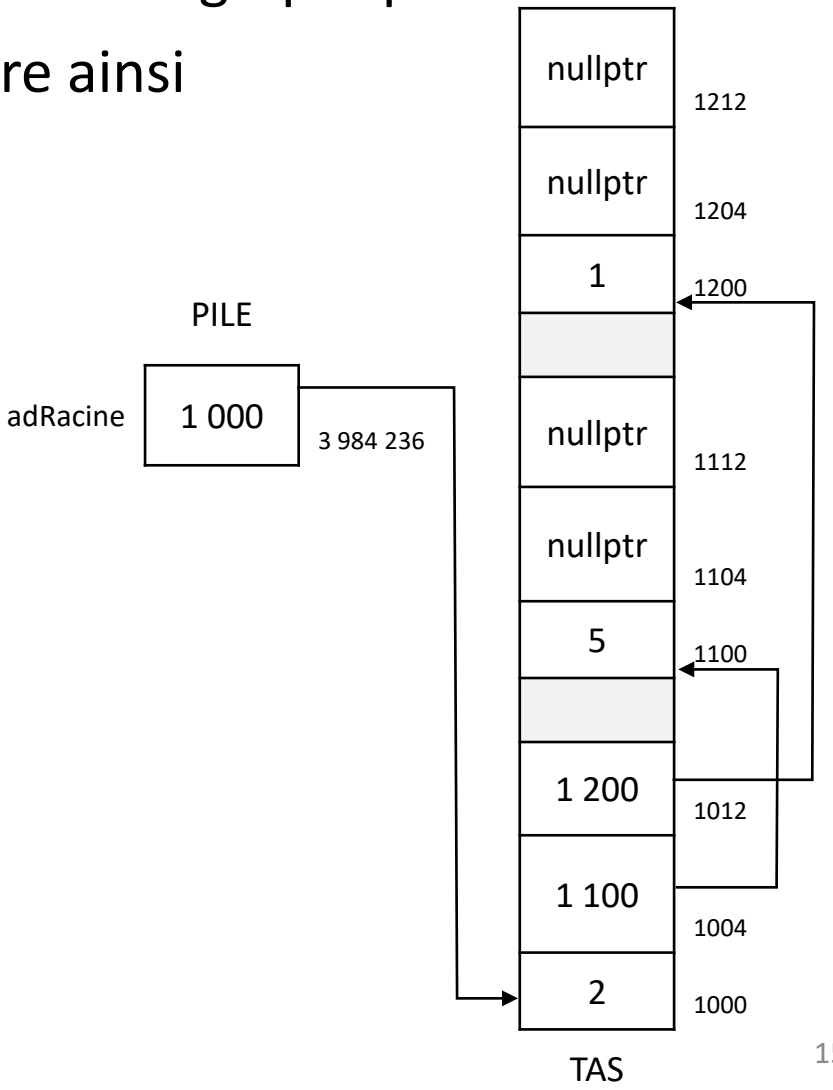
```
struct Noeud {
    ElementA info;
    Noeud * fg;
    Noeud * fd;
};

class Arbre {
public:
    Noeud * adRacine;
    // ...
}
```

# Arbre en mémoire

- L'arbre 

est représenté graphiquement et en mémoire ainsi

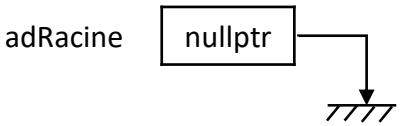


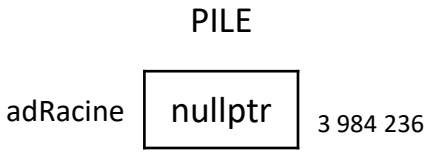



# Création d'un arbre vide

- Constructeur de la classe Arbre

```
Arbre::Arbre () {
   adRacine = nullptr;
}
```

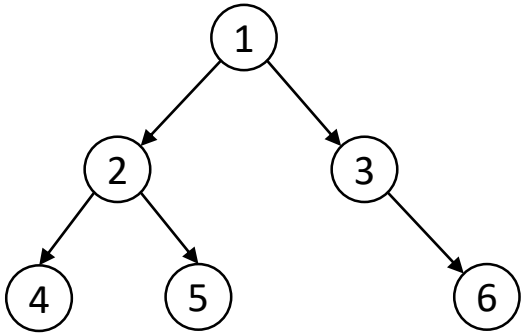


# Parcours d'un arbre

- On a plusieurs façons de visiter tous les nœuds d'un arbre, donnant des ordres de visite différents
- Parcours en profondeur
  - Parcours en ordre (infixe) : fils gauche, nœud, fils droit
  - Parcours en pré-ordre (préfixe) : nœud, fils gauche, fils droit
  - Parcours en post-ordre (postfixe) : fils gauche, fils droit, nœud
- Parcours en largeur
  - Parcours niveau après niveau (i.e. profondeur par profondeur)

# Parcours d’un arbre




- Parcours en ordre (infixe) : 4 2 5 1 3 6
- Parcours en pré-ordre (préfixe) : 1 2 4 5 3 6
- Parcours en post-ordre (postfixe) : 4 5 2 6 3 1
- Parcours en largeur : 1 2 3 4 5 6

# Algorithmes de parcours

- Les parcours en profondeur s'écrivent très facilement avec une procédure récursive

```
Procédure parcoursArbre ()
Postcondition : parcours du contenu de l'arbre en mode infixe
Début
     parcoursApartirDeNoeud(adRacine)
Fin

Procédure parcoursApartirDeNoeud (n : lien sur Noeud)
Début
   si n ≠ nullptr alors
      parcoursApartirDeNoeud(n->fg)
      traitementDuNoeud(n)
      parcoursApartirDeNoeud(n->fd)
   fin si
Fin
```

# Algorithmes de parcours

- Que faut-il modifier à l'algorithme précédent pour effectuer un parcours préfixe et un parcours post-fixe?

- Ces algorithmes peuvent également être écrits de manière itérative en utilisant une pile ou une file (cf. TD)

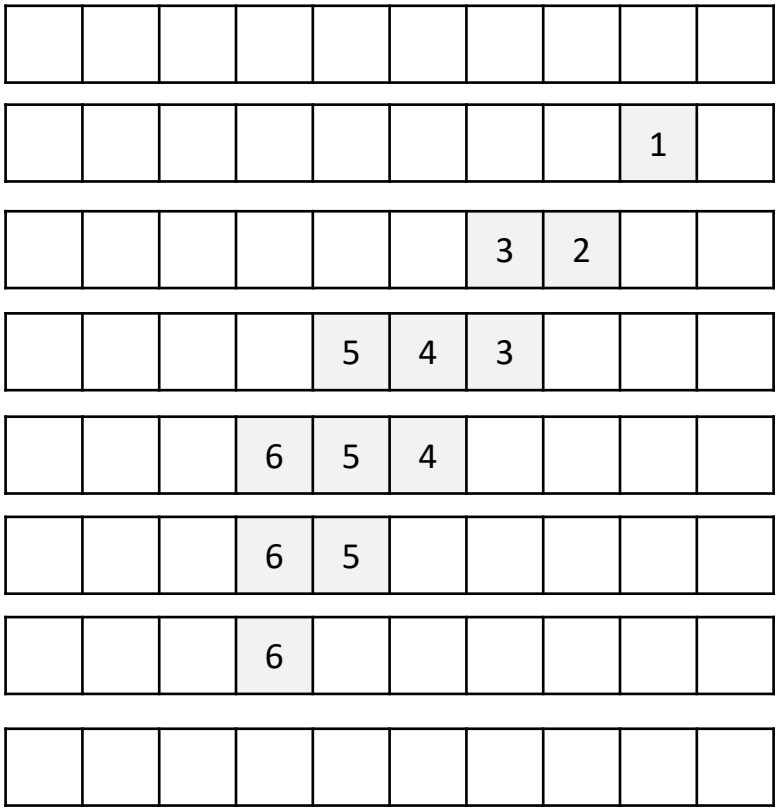
# Algorithmes de parcours

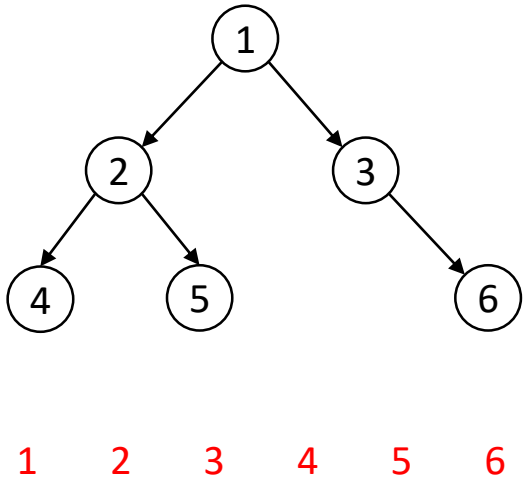
- Le parcours en largeur visite les nœuds par profondeur (en partant de la racine)
- Le plus facile est d'utiliser une file pour stocker les nœuds à visiter

```
Variables locales : f : File, n : lien sur Noeud
Début
   f.enfiler(adRacine)
   Tant que non f.estVide() faire
      n ← f.premierDeLaFile()
      f.defiler()
      si n->fg ≠ nullptr alors f.enfiler(n->fg) fin si
      si n->fd ≠ nullptr alors f.enfiler(n->fd) fin si
      traitementDuNoeud(n)
    fin Tant que
Fin
```

# Algorithmes de parcours

- Exemple de parcours en largeur



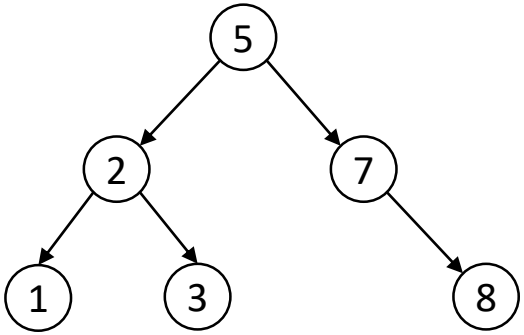

1 2 3 4 5 6

# Arbre binaire de recherche

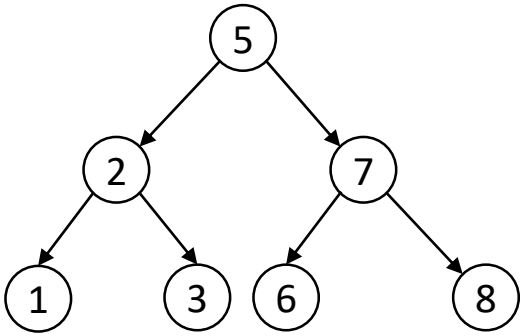
- Un arbre binaire de recherche (ABR) est une structure permettant de ranger des informations ordonnées
- C'est un arbre binaire où pour tout nœud n de l'arbre, les nœuds du fils gauche de n sont plus petits que n et les nœuds du fils droit sont plus grands que n
  - « SAG < nœud < SAD »
- Les procédures d'insertion et de suppression doivent faire respecter cette règle
  - Il faut donc trouver la bonne place où ajouter un nœud
  - On peut donc avoir besoin de réorganiser l'arbre après suppression d'un nœud

# Insertion d’un élément dans un ABR

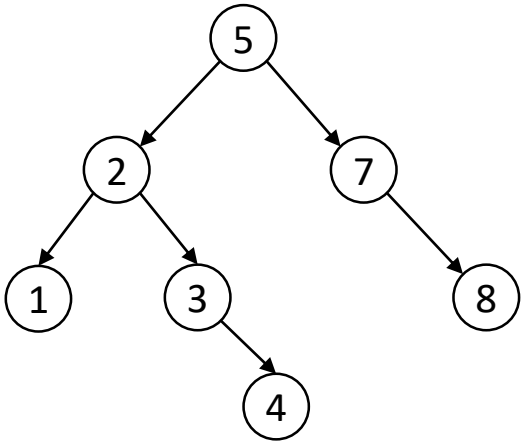
- Depuis l’arbre



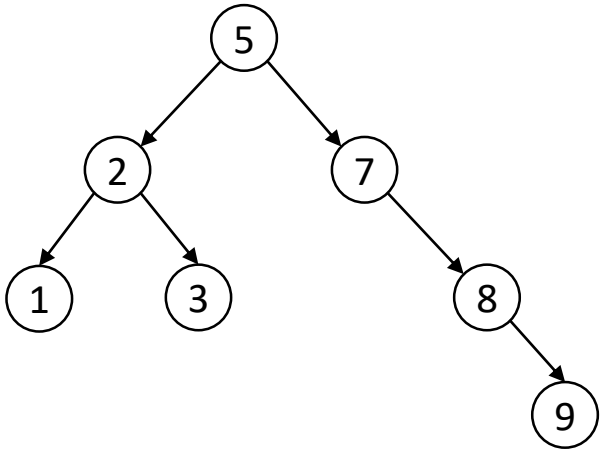

insertion de 6




insertion de 4

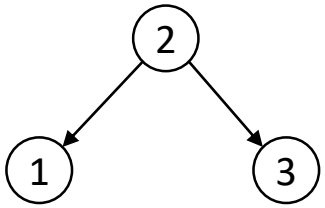



insertion de 9

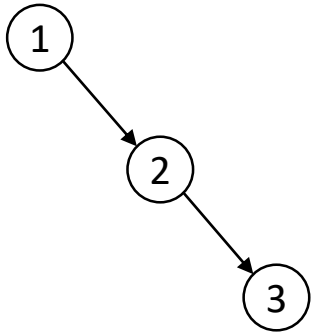


# Insertion d'un élément dans un ABR

- On insère toujours dans une feuille de l'arbre, on recherche juste la bonne place
- Deux arbres « identiques » (i.e. avec les mêmes éléments) mais dont l'ordre d'insertion des éléments est différent donne deux arbres différents
  - Insertion dans l'ordre 2 1 3

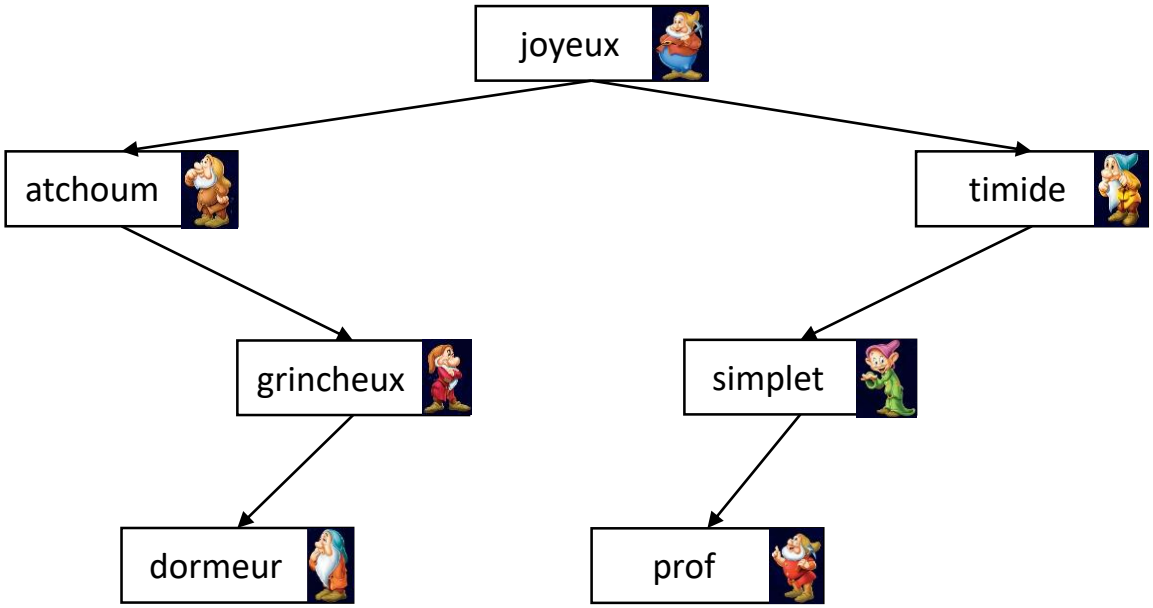



  - Insertion dans l'ordre 1 2 3



# Insertion d'un élément dans un ABR

- Exemple avec les 7 nains insérés dans l'ordre suivant {joyeux, atchoum, grincheux, timide, simplet, prof, dormeur} où la relation d'ordre est l'ordre alphabétique



# Insertion : méthode récursive

```
Procédure insererElement (e : ElementA)
Postcondition : après insertion de l'élément e on a toujours bien SAG < noeud < SAD
Début
   insererElementAPartirDuNoeud(adRacine,e)
Fin

Procédure insererElementAPartirDuNoeud (n: lien sur Noeud, e: ElementA)
Paramètre en mode donnée-résultat : n
Début
   si n = nullptr alors
      n ← réserve Noeud, n->fg ← nullptr, n->fd ← nullptr, n->info ← e
   sinon
      si e ≠ n->info alors
         si e < n->info alors
            insererElementAPartirDuNoeud(n->fg,e)
         sinon
            insererElementAPartirDuNoeud(n->fd,e)
         fin si
   fin si
Fin
```

# Insertion : méthode itérative

```
Procédure insererElement (e : ElementA)
Postcondition : après insertion de l'élément e on a toujours bien SAG < noeud < SAD
Variable locale : n : lien sur Noeud, fin : booléen
Début
   si adRacine = nullptr alors
      n ← réserve Noeud, n->fg ← NULL, n->fd ← nullptr, n->info ← e, adRacine ← n
   sinon
      fin ← faux
      n ← adRacine
      Tant que non fin faire
         si e ≠ n->info alors
            si e < n->info alors
               si n->fg = nullptr alors
                  n->fg ← réserve Noeud, (n->fg)->fg ← nullptr, (n->fg)->fd ← nullptr, (n->fg)->info ← e, fin ← vrai
               sinon
                  n ← n->fg
               fin si
            sinon
               // idem pour le fils droit
            fin si
         sinon
            fin ← vrai
         fin si
      fin Tant que
   fin si
Fin
```

# Recherche : méthode récursive

```
Fonction rechercherElement (e : ElementA) : lien sur Noeud
Résultat : retourne le lien sur le Noeud si e existe, nullptr sinon
Début
   retourne rechercherElementAPartirDuNoeud(adRacine,e)
Fin


Fonction rechercherElementAPartirDuNoeud (n: lien sur Noeud, e: ElementA) : lien sur Noeud
Début
   si n ≠ nullptr alors
      si e = n->info alors retourne n
      sinon
         si e < n->info alors retourner rechercherElementAPartirDuNoeud(n->fg,e)
         sinon retourner rechercherElementAPartirDuNoeud(n->fd,e)
      fin si
   fin si
   retourne nullptr
Fin
```

# Suppression dans un ABR

- Il y a quatre cas à distinguer
  - le nœud à supprimer n'est pas dans l'arbre
  - le nœud à supprimer est une feuille
  - le nœud à supprimer a un seul fils
  - le nœud à supprimer a deux fils

# Suppression dans un ABR

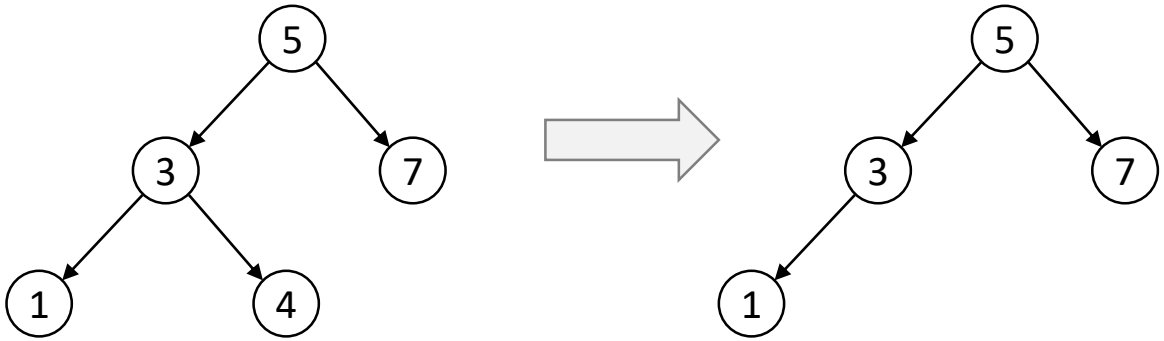
- Le nœud à supprimer n'est pas dans l'arbre

➢ Rien à faire

- Le nœud à supprimer est une feuille

➢ On supprime la feuille et on met à jour le père

  - suppression du nœud 4

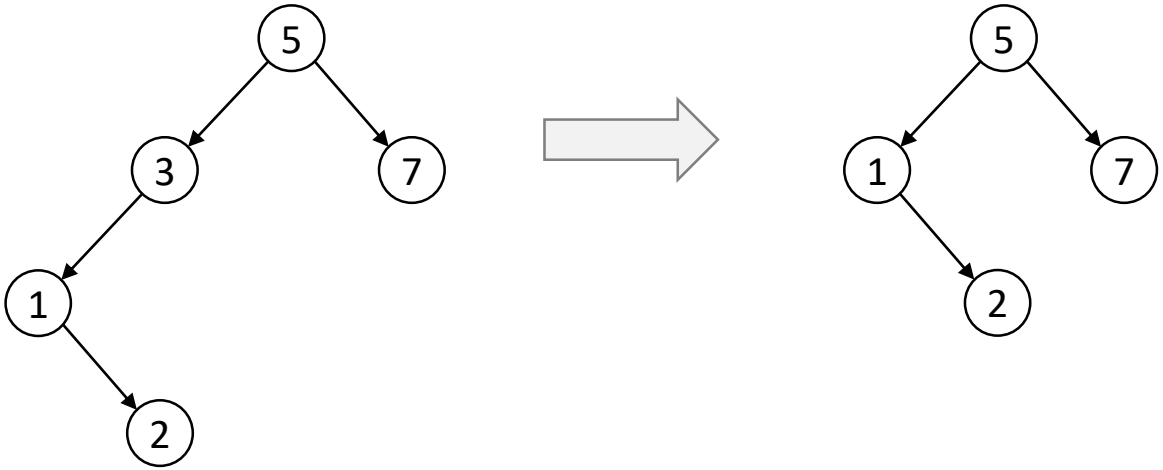


# Suppression dans un ABR

- Le nœud à supprimer a un seul fils

➢ Il suffit de court-circuiter le nœud à supprimer (i.e. le père pointe sur le fils non nul)

  - suppression du nœud 3



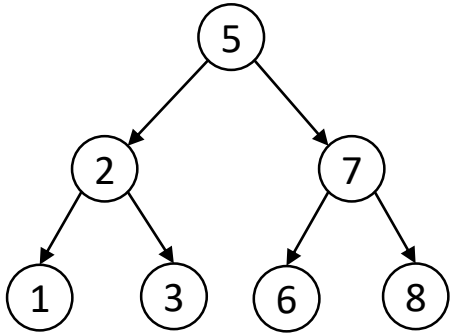
# Suppression dans un ABR

- Le nœud à supprimer a deux fils

➢ Pour conserver la propriété d'un ABR, il faut remplacer le nœud par son plus proche successeur ou plus proche prédécesseur

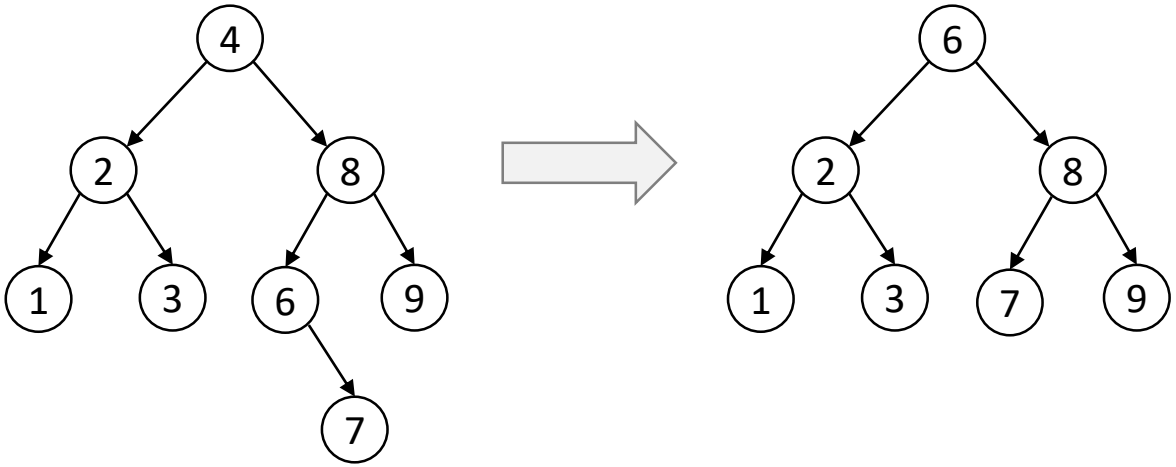
  - Plus proche successeur = le nœud le plus à gauche du sous arbre droit
  - Plus proche prédécesseur = le nœud le plus à droite du sous arbre gauche

  - Plus proche successeur de 5 = 6
  - Plus proche prédécesseur de 5 = 3



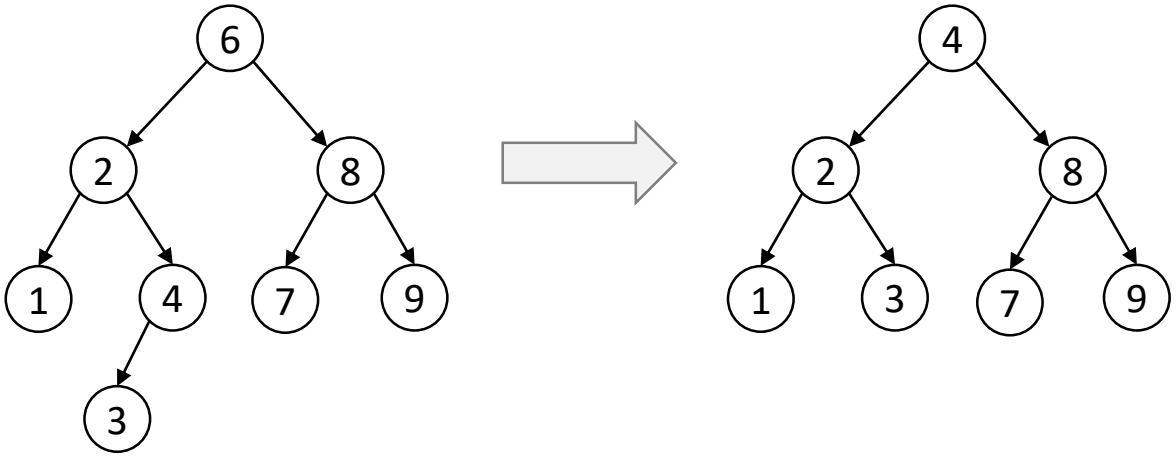
# Suppression dans un ABR

- Suppression du nœud 4 par remplacement avec le plus proche successeur (i.e. 6)



# Suppression dans un ABR

- Suppression du nœud 6 par remplacement avec le plus proche prédécesseur (i.e. 4)



# Déséquilibre d'un ABR

- Les ajouts et suppressions peuvent rendre un arbre déséquilibré, jusqu'à dégénéré
- Pour pouvoir faire des recherches efficaces d'éléments (en $O(\log n)$) il faut rééquilibrer l'arbre
  - à chaque ajout/suppression ou moins souvent