



DIU EIL – UE 5

Arbre

Nicolas Pronost

Définition d'un arbre

- Un arbre est une structure de donnée hiérarchique, composé de **nœuds** et de **relations de précedence** entre ces nœuds → les listes sont séquentielles
- 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
- Un arbre est donc une **structure réursive**, un successeur d'un nœud étant lui-même un arbre

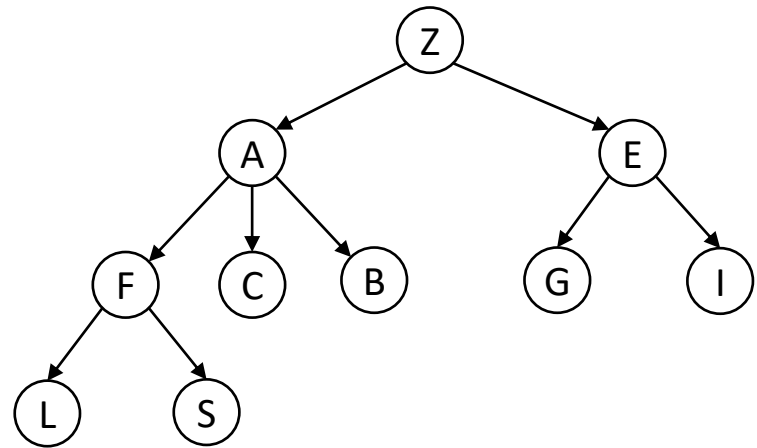
point d'entrée
de l'arbre

Vocabulaire

- La **racine** est le nœud sans prédécesseur (point d'accès au contenu de l'arbre entier)
- 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

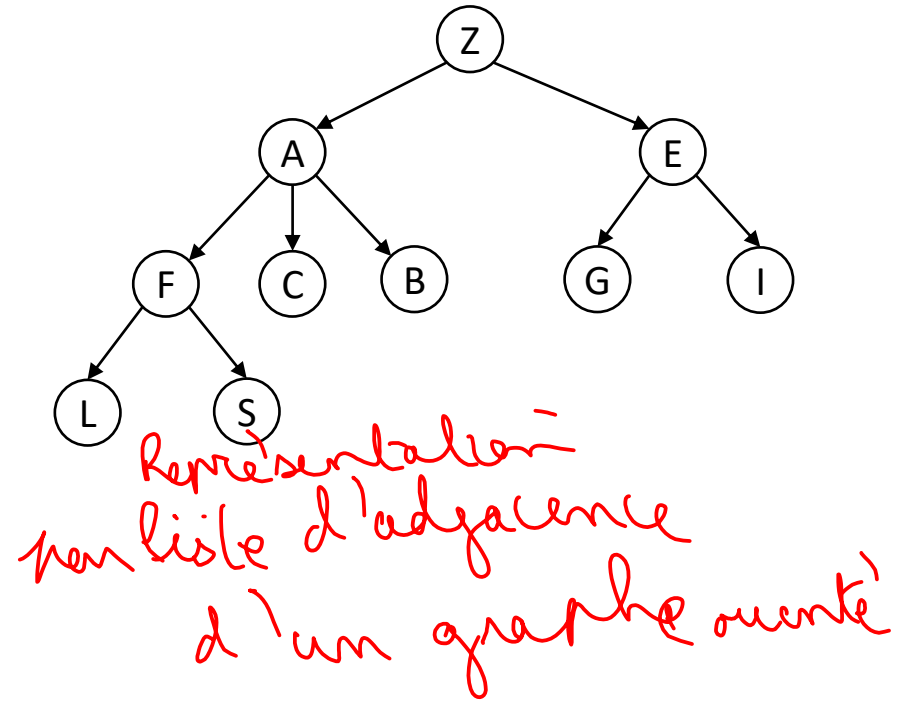
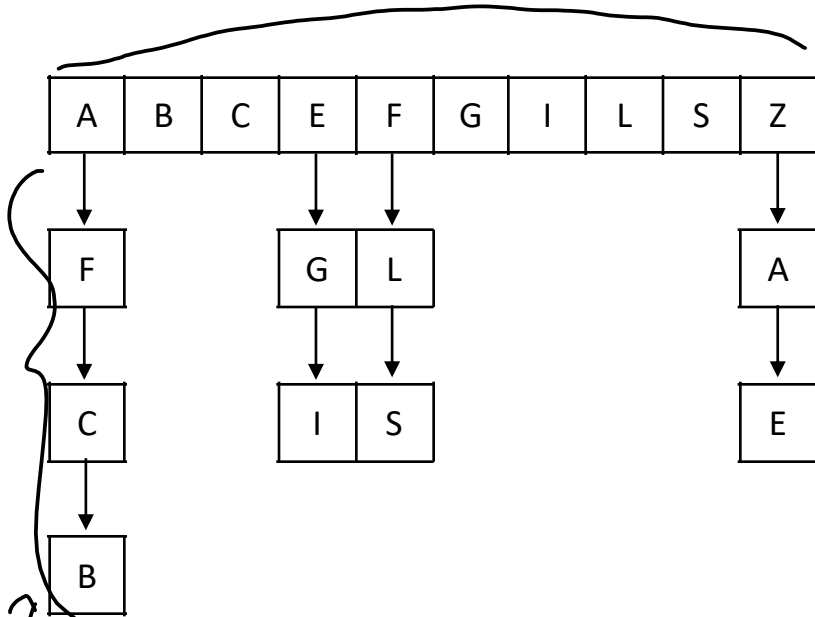
Exemple

- 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

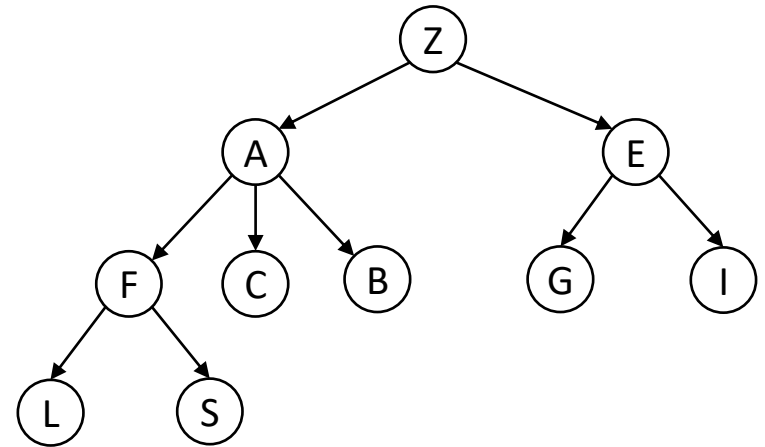
- Par liste d'adjacence *tous les nœuds de l'arbre*



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 par matrice d'adjacence d'un graphe orienté (matrice non symétrique)

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 : $O(n^2)$

→ 1 première liste avec tous les nœuds
et 1 deuxième dimension avec des listes
totalisant au plus $n-1$ fils

Représentation d'un arbre

- Comparaison entre liste d'adjacence et tableau 2D
 - Supprimer une feuille (*pas de fils*)
 - 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 tableau dynamique)
 - mettre à 1 le père en $O(1)$
 - $O(n^2) + O(1) = O(n^2)$

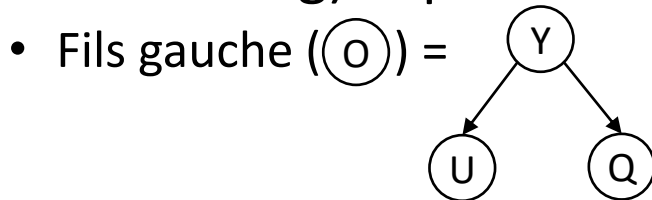
↳ coût doit recopier tout le tableau

↳ dans un meilleur cas avec un tableau dynamique, coût en $O(n)$

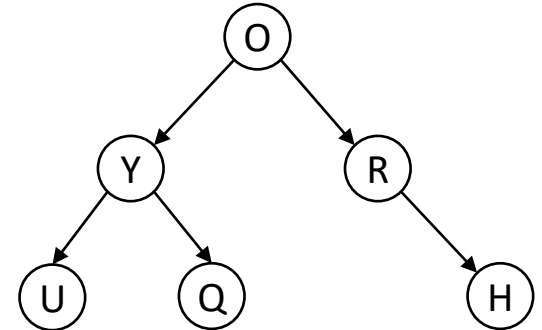
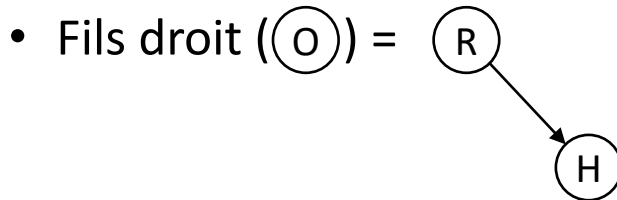
Au final l'implémentation
avec listes d'adjacences a une
meilleure complexité que celle avec
le tableau à deux dimensions.

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

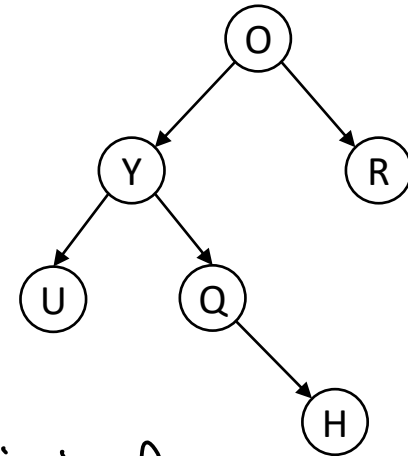
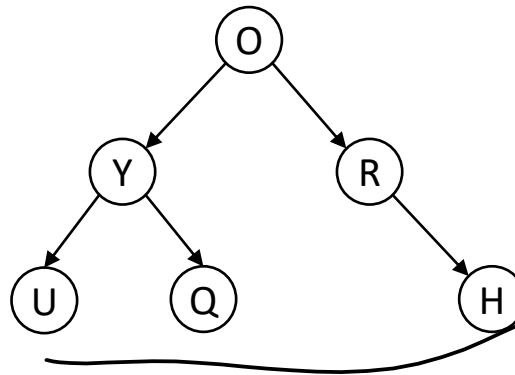
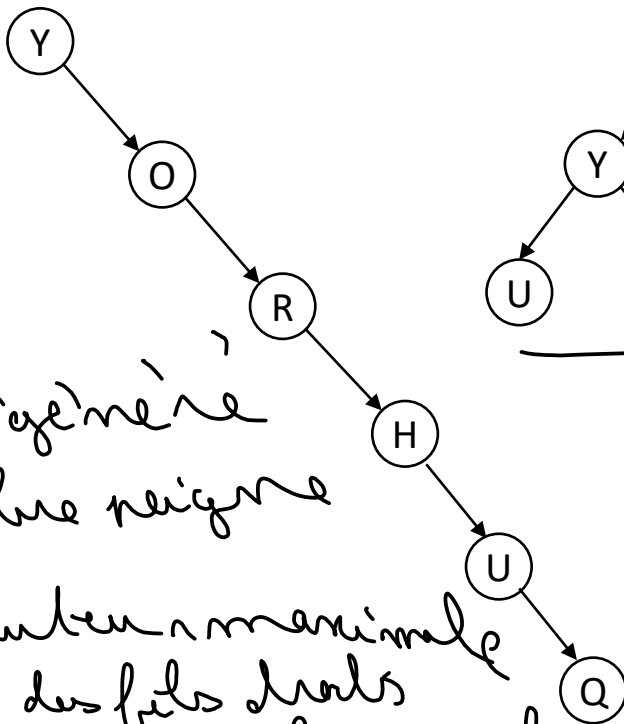


- On appelle **fils droit** (ou sous arbre droit ou sad) le deuxième successeur



Arbre binaire

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



dégénéré
arbre peigne

— hauteur maximale
que des fils droits
— que des fils gauches

arbre équilibré

— hauteur minimale
par rapport au nb
de nœuds

— on ne commence pas
une nouvelle profondeur
sans avoir rempli la précédente

Fonctionnalités de Arbre (binaire)

- **Constructeur** `Arbre()`
 - Postconditions : l'arbre est initialement vide
- **Destructeur** `~Arbre()`
 - Postconditions : libération de la mémoire utilisée, 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 à l'écran
- **Procédure** `insérerElement (e)`
 - 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
- **Procédure** `supprimerElement (e)`
 - Postcondition : l'élément e est recherché et supprimé de l'arbre, sans effet si le noeud n'est pas présent, les propriétés de l'arbre sont conservées
- **Fonction** `hauteurArbre () : entier`
 - Résultat : la hauteur de l'arbre (longueur de sa plus longue branche), ou -1 s'il est vide
- **Fonction** `rechercherElement(e) : (booléen,Arbre)`
 - Résultat : un tuple indiquant si l'élément e est dans l'arbre et le noeud-arbre de l'élément (None si absent)

Mise en œuvre d'un arbre en Python

- Un arbre vide est représenté par l'attribut info à None
- Les attributs fg et fd sont des arbres donc tous les nœuds sont des arbres, une feuille a deux attributs fg et fd à None


```
class Arbre :  
    def __init__(self, info=None, fg=None, fd=None) :  
        self.info = info  
        self.fg = fg  
        self.fd = fd  
  
    # ...
```

→ si info vaut None
alors l'arbre est vide

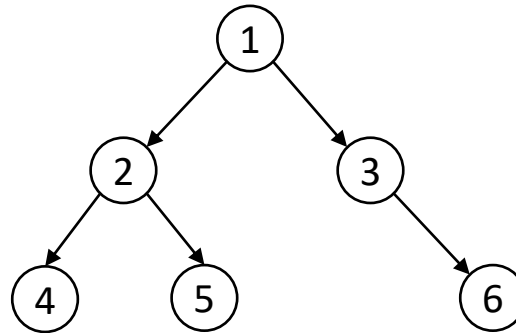
```
a1 = Arbre()      # arbre vide  
a2 = Arbre(2)     # arbre avec un nœud (la racine)
```

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**  à quel moment est traité le nœud
 - 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

on traite en
dernier ce
qu'on a vu
en premier

Algorithmes de parcours

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

```
def afficherParcoursInfixe(self):  
    if self.info:  
        if self.fg : self.fg.afficherParcoursInfixe()  
        print(self.info, end=' ') # on traite le noeud  
        if self.fd : self.fd.afficherParcoursInfixe()
```

On a choisi que si un
noeud existe si son champ
info est différent de None

→ Ainsi on ne peut pas faire le parcours
infixe d'un arbre vide. On pourrait
représenter des

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

\hookrightarrow en TD

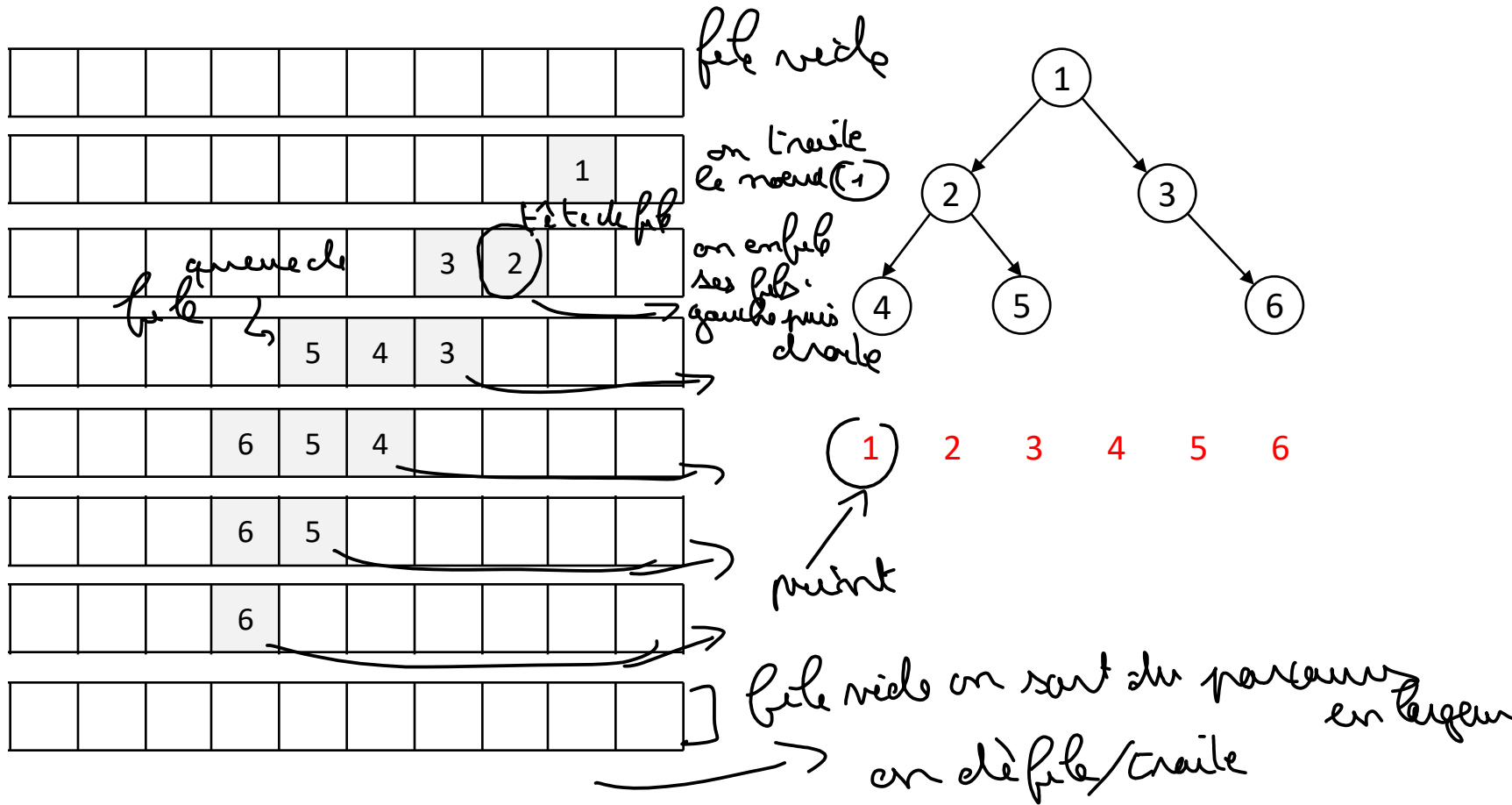
Algorithmes de parcours

- 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
- Algorithme difficile à écrire de manière récursive*

```
def afficherParcoursLargeur(self):  
    f = pf.File([]) → file vide  
    f.enfiler(self)  
    while not f.estVide():  
        n = f.traiter() → on défile la tête de file  
        if n.fg : f.enfiler(n.fg)  
        if n.fd : f.enfiler(n.fd)  
        print(n.info, end=' ') # on traite le noeud
```

Algorithmes de parcours

- Exemple de parcours en largeur



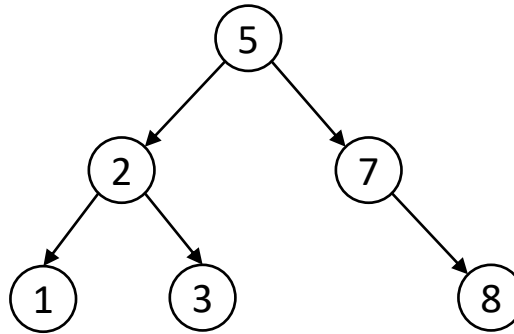
Arbre binaire de recherche

Arbre binaire sans type d'arbre et arbre binaire de recherche
sans type de arbre binaire

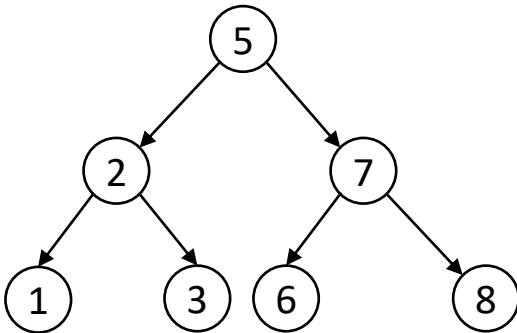
- Un arbre binaire de recherche (ABR) est une structure permettant de ranger des informations ordonnées
- C'est un arbre binaire où pour tout nœud n de l'arbre, les nœuds du fils gauche de n sont plus petits que n et les nœuds du fils droit sont plus grands que n
 - « $SAG < \text{nœud} < SAD$ »
- Les 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] plus compliqué

Insertion d'un élément dans un ABR

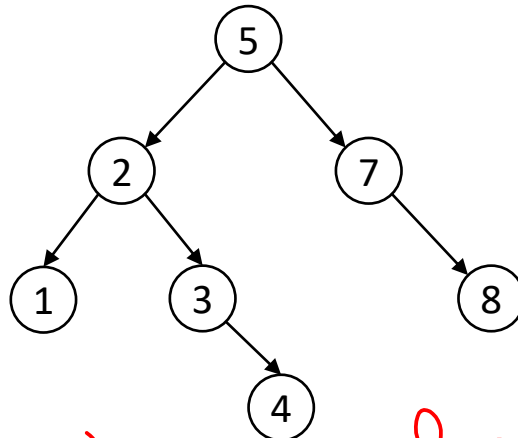
- Depuis l'arbre



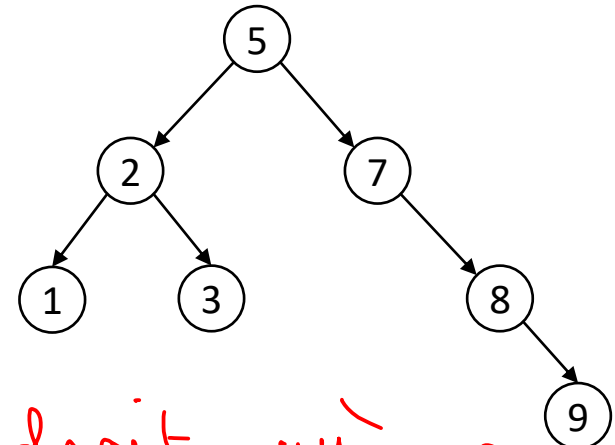
insertion de 6



insertion de 4



insertion de 9

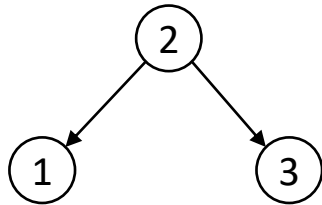


Il n'y a toujours qu'un seul endroit où insérer un élément dans un ABR (dans une feuille)

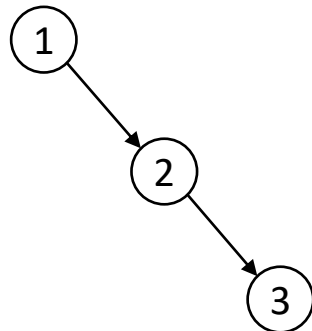
Insertion d'un élément dans un ABR

- 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

```
def insererElement(self, e):  
    if not self.info:  
        # le sous arbre est vide  
        self.info = e  
    else:  
        if e < self.info:  
            if self.fg : #le sous arbre existe déjà, on continue  
                self.fg.insererElement(e)  
            else: # le sous arbre n'existe pas, on le créer  
                self.fg = Arbre(e)  
        if e > self.info:  
            if self.fd :  
                self.fd.insererElement(e)  
            else:  
                self.fd = Arbre(e)  
        # rien à faire si égal, déjà présent, arrêt de la récursion
```

} cas d'insertion dans un arbre vide

↳ on ne fait rien si $e == self.info$ car on n'insère pas un élément déjà présent

```
a = Arbre()           # arbre vide  
a.insererElement(2)   # ajout d'un noeud valant 2 à la racine  
a.insererElement(3)   # ajout d'un noeud valant 3 en fd de la racine
```


Recherche d'un élément dans un ABR

```
def rechercherElement(self, e):  
    if not self.info: return (False, None)  
    if e == self.info: return (True, self)  
    if e < self.info:  
        if self.fg:  
            return self.fg.rechercherElement(e)  
        else:  
            return (False, None)  
    if e > self.info:  
        if self.fd:  
            return self.fd.rechercherElement(e)  
        else:  
            return (False, None)
```

} autre nœud

} si l'élément existait alors il aurait été là

```
(trouve, noeud) = a.rechercherElement(3)
```

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

4 cas

ordre croissant
de difficulté
de traitement

Suppression dans un ABR

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

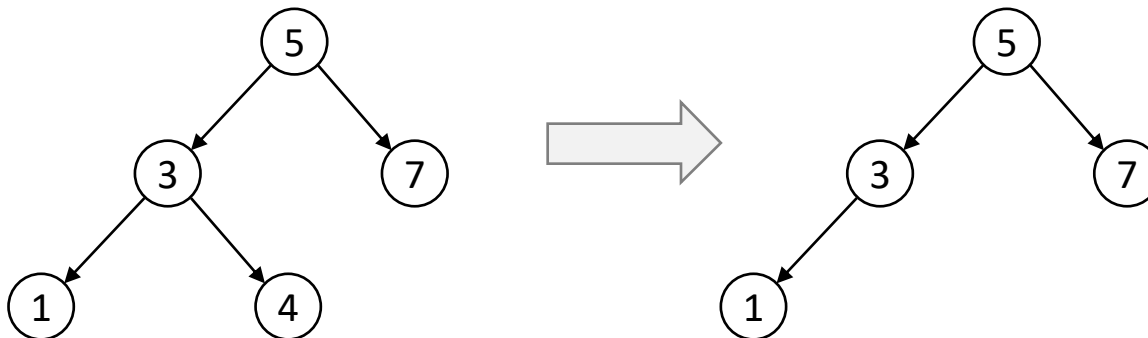
➤ Rien à faire

> Voir la discussion en page 8

- Le nœud à supprimer est une feuille

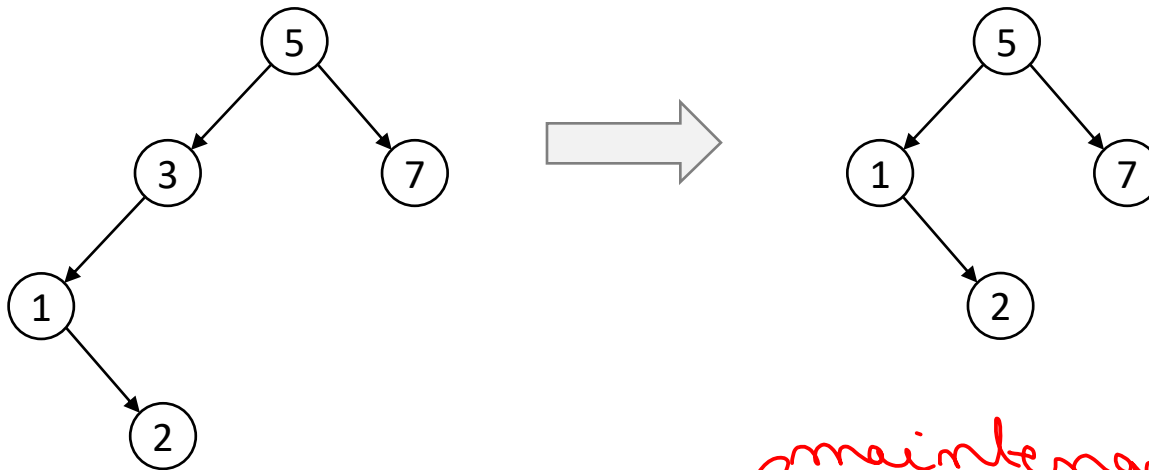
➤ On supprime la feuille et on met à jour le père

- 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



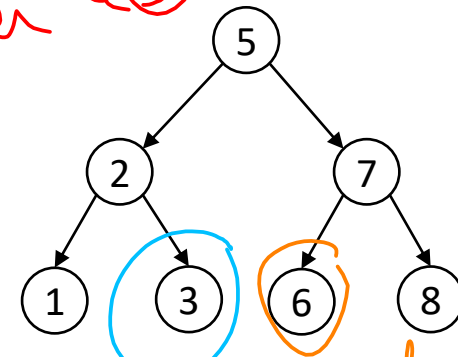
maintenant le
fils gauche de 5
est (1) et plus (3)
↳ supprimer le nœud (3)
et mettre à jour le père.

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

ici on veut supprimer 5

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

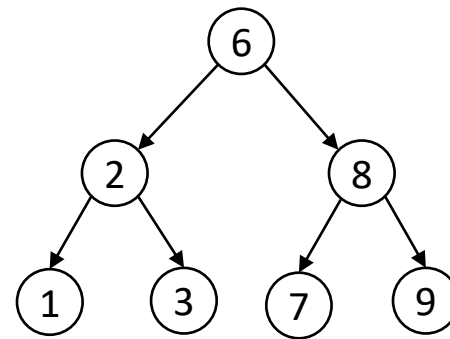
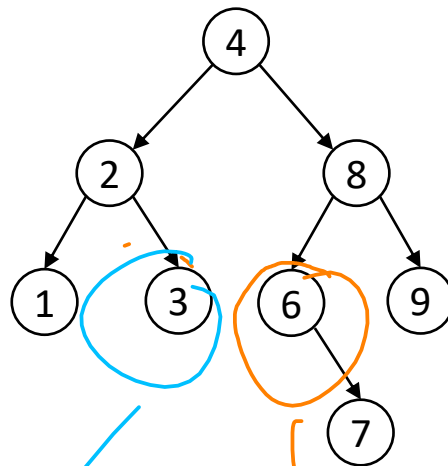


*plus proche
prédécesseur*

*plus
proche
successeur*

Suppression dans un ABR

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



on remplace
par 6(6)

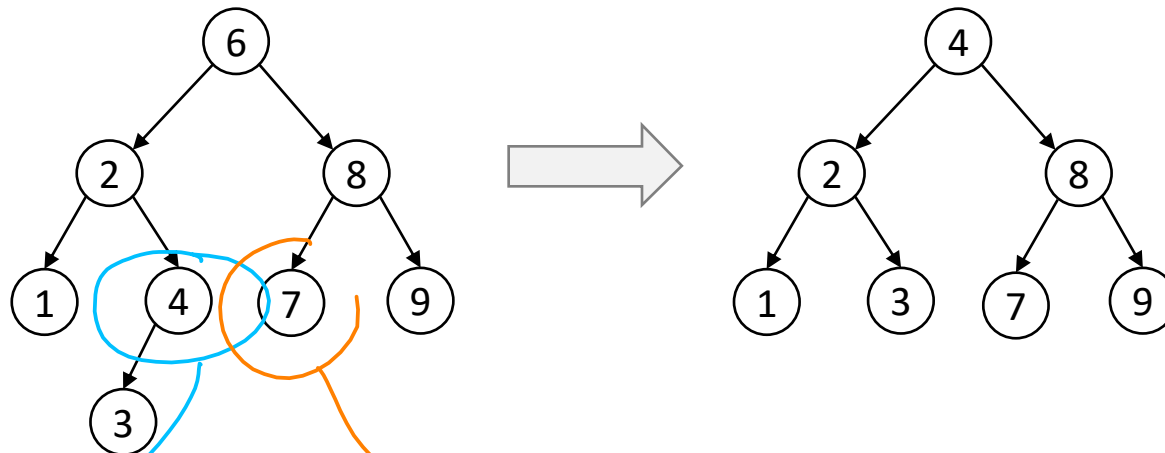
plus
proche
prédéceseur

plus proche successeur

on prend 6 sous
arbre droit de 6
et on le met comme
sous arbre gauche de 8

Suppression dans un ABR

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

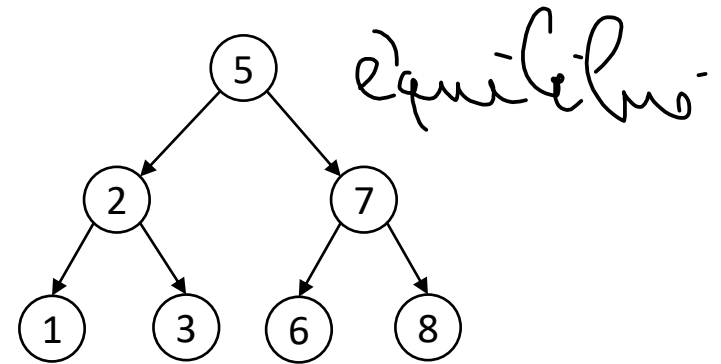
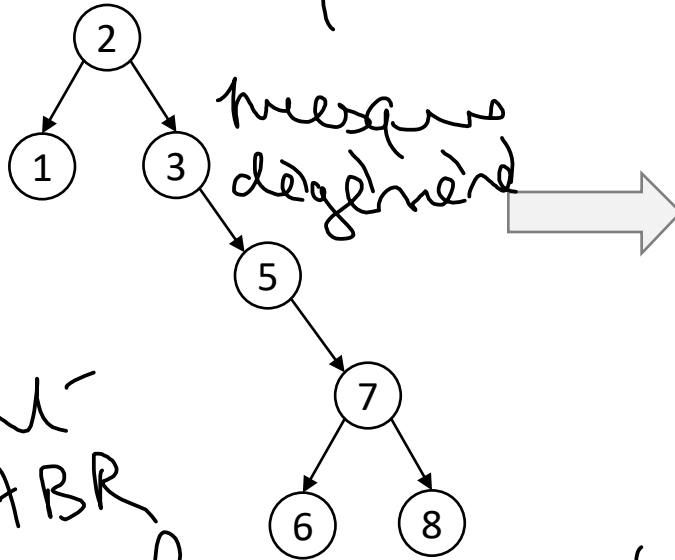


plus
proche
prédécesseur

plus proche
successeur

Déséquilibre d'un ABR

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



Ce sont
deux ABR
mais celui de droite
pour les recherches

est plus performant