

SYNTHESE : Synthèse Arbres

■ Structures de données

L'écriture sur des exemples simples de plusieurs implémentations d'une même structure de données permet de faire émerger les notions d'interface et d'implémentation, ou encore de structure de données abstraite. Le paradigme de la programmation objet peut être utilisé pour réaliser des implémentations effectives des structures de données, même si ce n'est pas la seule façon de procéder.

Le lien est établi avec la notion de modularité qui figure dans la rubrique « langages et programmation » en mettant en évidence l'intérêt d'utiliser des bibliothèques ou des API (*Application Programming Interface*).

Contenus	Capacités attendues	Commentaires
Arbres : structures hiérarchiques. Arbres binaires : nœuds, racines, feuilles, sous-arbres gauches, sous-arbres droits.	Identifier des situations nécessitant une structure de données arborescente. Évaluer quelques mesures des arbres binaires (taille, encadrement de la hauteur, etc.).	On fait le lien avec la rubrique « algorithmique ».

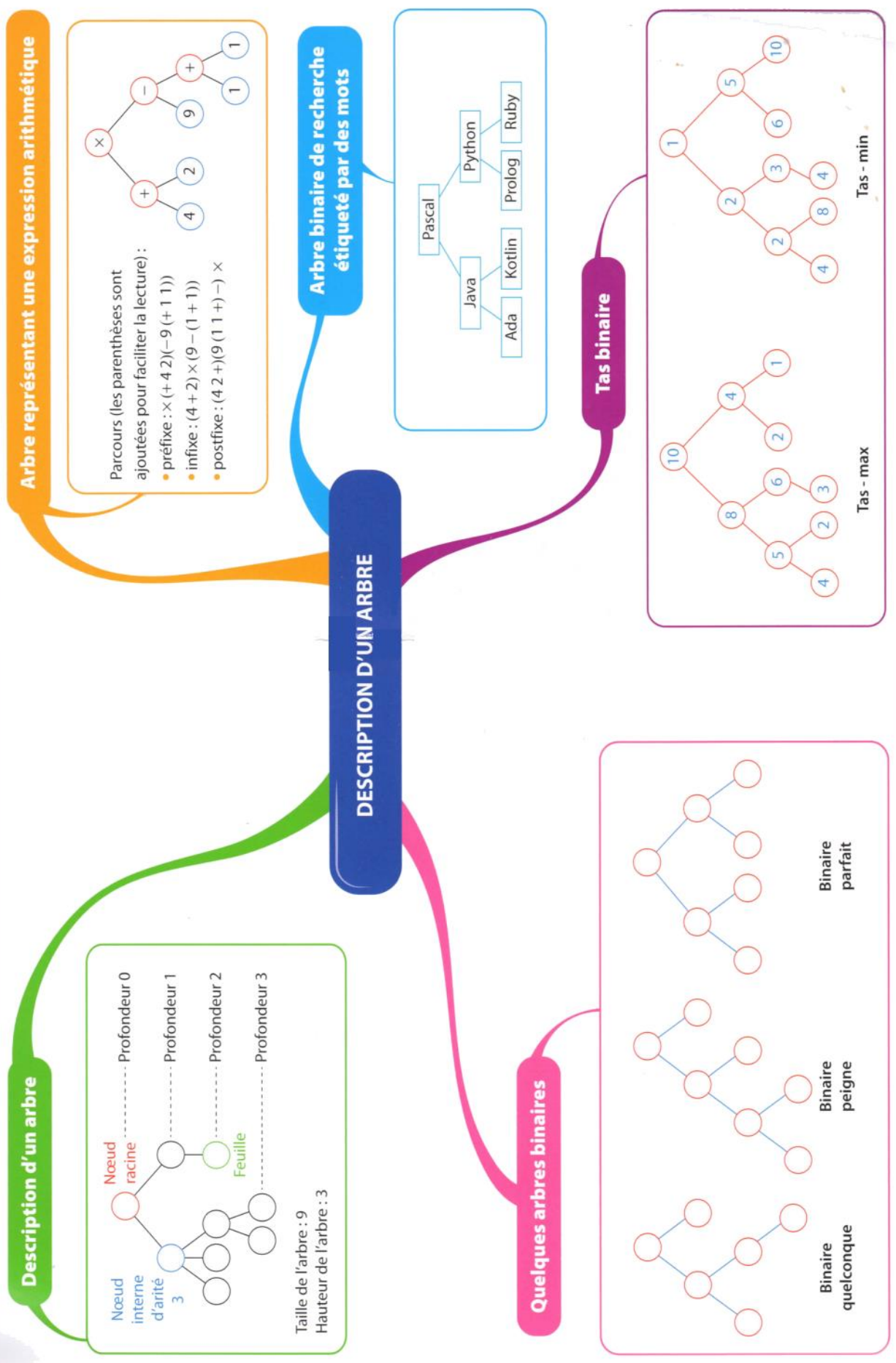
■ Algorithmique

Le travail de compréhension et de conception d'algorithmes se poursuit en terminale notamment via l'introduction des structures d'arbres et de graphes montrant tout l'intérêt d'une approche récursive dans la résolution algorithmique de problèmes.

On continue l'étude de la notion de coût d'exécution, en temps ou en mémoire et on montre l'intérêt du passage d'un coût quadratique en n^2 à $n \log_2 n$ ou de n à $\log_2 n$. Le logarithme en base 2 est ici manipulé comme simple outil de comptage (taille en bits d'un nombre entier).

Contenus	Capacités attendues	Commentaires
Algorithmes sur les arbres binaires et sur les arbres binaires de recherche.	Calculer la taille et la hauteur d'un arbre. Parcourir un arbre de différentes façons (ordres infixe, préfixe ou suffixe ; ordre en largeur d'abord). Rechercher une clé dans un arbre de recherche, insérer une clé.	Une structure de données récursive adaptée est utilisée. L'exemple des arbres permet d'illustrer la programmation par classe. La recherche dans un arbre de recherche équilibré est de coût logarithmique.

Structures de données arborescentes

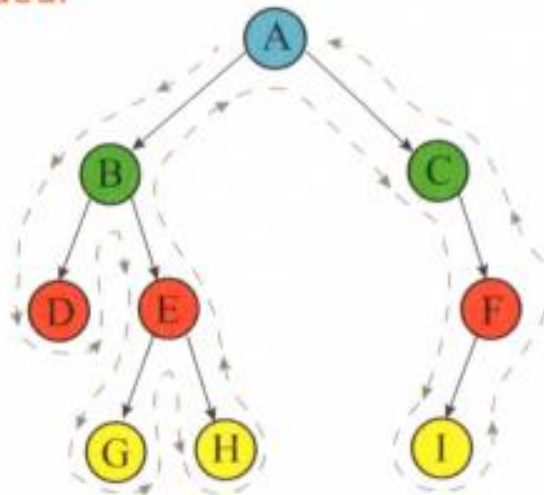


● Propriétés

Soit un arbre de hauteur h à f feuilles et n nœuds.

$n \leq 2^{h+1} - 1$	$\log_2(n + 1) - 1 \leq h \leq n$
$h \geq \log_2(f)$	$\log_2(f) \leq \log_2\left(\frac{n + 1}{2}\right)$

● Parcours en profondeur



Ordre	Description	Liste des nœuds
Préfixe	Ordre dans lequel on rencontre les nœuds pour la première fois.	A - B - D - E - G - H - C - F - I
Postfixe	Ordre dans lequel on rencontre les nœuds pour la dernière fois.	D - G - H - E - B - I - F - C - A
Infixe	Chaque nœud ayant un fils gauche est répertorié la seconde fois qu'on le rencontre, et chaque nœud n'ayant pas de fils gauche est répertorié la première fois qu'on le rencontre.	D - B - G - E - H - A - C - I - F

● Parcours en largeur

On liste les nœuds suivant leur niveau dans l'arbre :

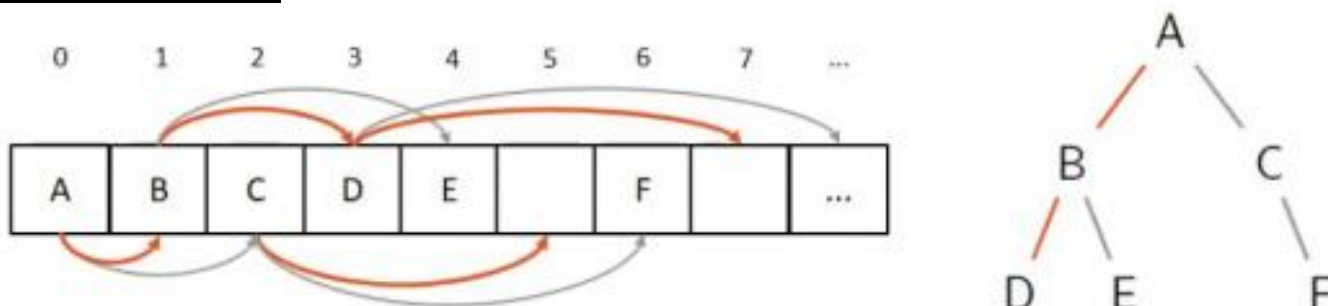
$$\underbrace{A}_{\text{niveau 0}} - \underbrace{B - C}_{\text{niveau 1}} - \underbrace{D - E - F}_{\text{niveau 2}} - \underbrace{G - H - I}_{\text{niveau 3}}$$

Algorithme, Préfixe [noeud] + appelRécursif sur sag + appelRécursif sur sad

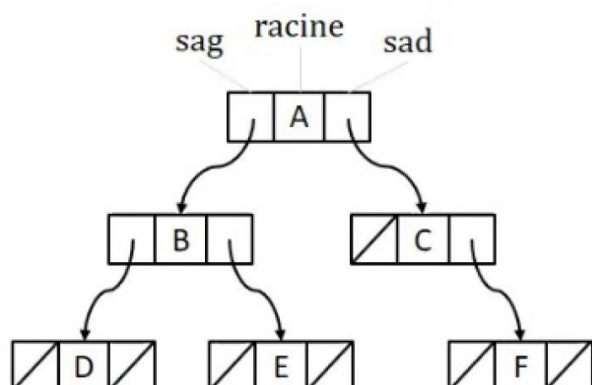
Algorithme, Postfixe appelRécursif sur sag + appelRécursif sur sad + [noeud]

Algorithme, Postfixe appelRécursif sur sag + appelRécursif sur sad + [noeud]

Représentation liste

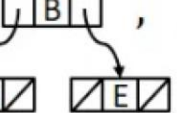
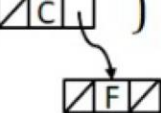


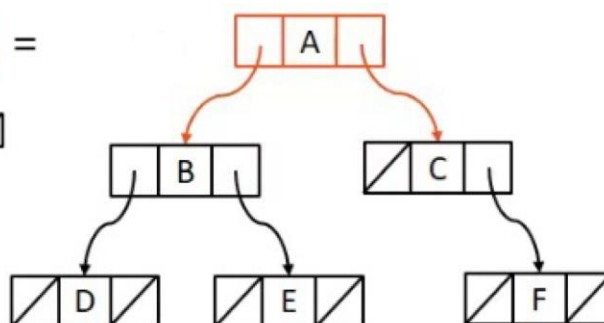
Représentation par une structure récursive : le nœud



```
class Noeud :
    # Constructeur
    def __init__(self, valeur) :
        self.donnée = valeur
        self.gauche = None
        self.droit = None

    # Affichage
    def __str__(self) :
        return str(self.donnée)
```

assembler(A,  , ) =



La recherche dans un ABR : algorithme

En moyenne, la complexité de la recherche dans un ABR est en $\Theta(\ln n)$.

Recherche(A,x) :

Si estVide(A) alors faux

Si x = racine(A) alors Vrai

Si x < racine(A) alors Recherche(sag(A),x)

Sinon Recherche(sad(A),x)