II. Les algorithmes sur les arbres

Cours

A. Les parcours

Parcourir un arbre binaire consiste à accéder à l'information contenue dans ses nœuds : à leur étiquette.

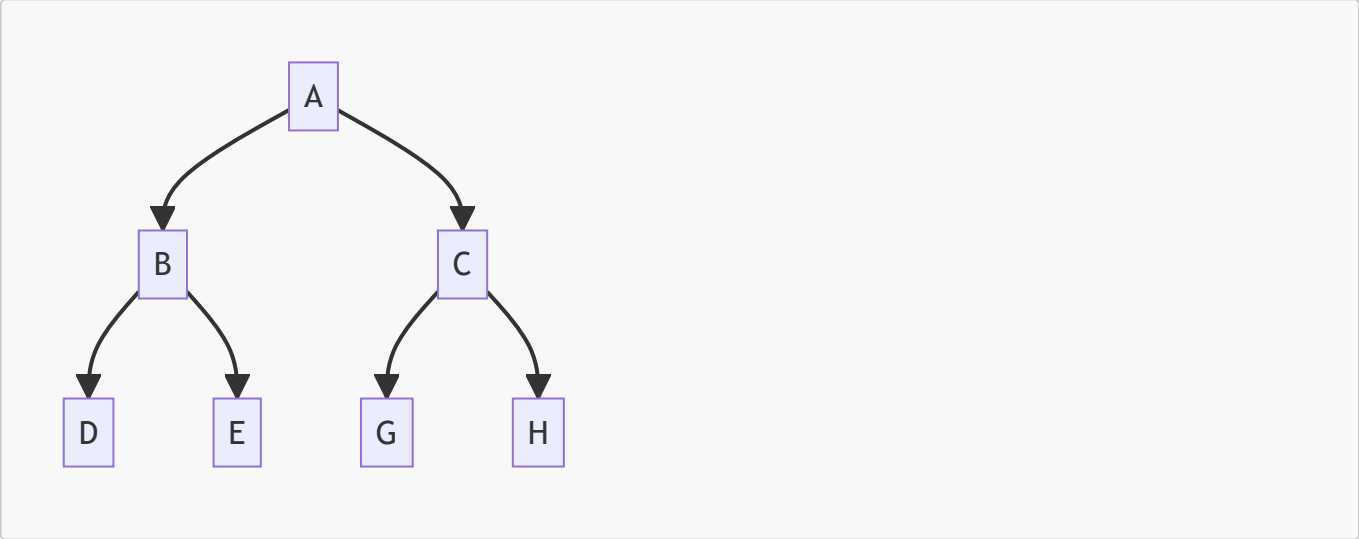
A.I. Parcours en profondeur

Un **parcours en profondeur** consiste à parcourir les nœuds en suivant les branches.

On considère que chaque nœud d'un arbre binaire est visité 3 fois, dans l'ordre suivant :

- une fois par la gauche,
- une fois par en-dessous,
- une fois par la droite.

Illustrons-le sur l'arbre suivant :



- **Parcours préfixe :** on liste chaque nœud la première fois qu'il est rencontré.

??? abstract "Exercice : quel est le parcours préfixe de l'arbre précédent ?" A, B, D, E, C, G, H.

- **Parcours infixe :** on liste chaque nœud la deuxième fois qu'il est rencontré.

??? abstract "Exercice : quel est le parcours infixe de l'arbre précédent ?" D, B, E, A, G, C, H

- **Parcours suffixe ou postfixe :** on liste chaque nœud la dernière fois qu'il est rencontré.

??? abstract "Exercice : quel est le parcours sufixe de l'arbre précédent ?" D, E, B, G, H, C, A

A.II. Parcours en largeur

Dans un **parcours en largeur**, on liste chaque nœud niveau par niveau (un niveau correspondant à une profondeur égale).

!!! abstract "Représenter ce parcours sur l'arbre ci-dessous :"

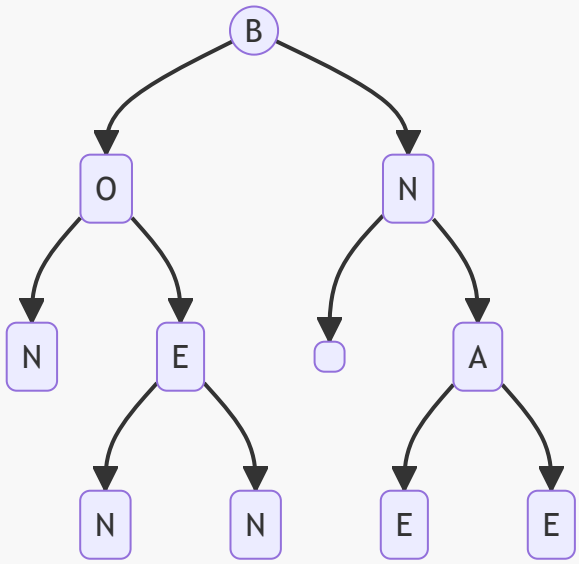
```
``` mermaid
graph TB
  A --> B
  A --> C
  B --> D
  B --> E
  C --> G
  C --> H
```
??? note "Solution"
    A, B, C, D, E, G, H
```

B. Sur les ABR

Dans un ABR, les nœuds sont ordonnés. Cela facilite donc les recherches, et l'insertion d'un nouveau nœud ne peut se faire au hasard.

B.I. Recherche

??? note "Quel algorithme appliquer pour faire une recherche dans un ABR ?"
On parcourt l'arbre en commençant par la racine : - si elle est égale à la valeur recherchée, la valeur est trouvée,
- sinon si celle-ci est plus petite que la valeur recherchée, on continue le parcours dans le **sous-arbre droit**,
- sinon, on continue dans le **sous-arbre gauche**. Dans les deux derniers cas, on réapplique l'algorithme en partant de la racine du sous-arbre en question, jusqu'à tomber sur un **sous-arbre vide**.

B.II. Insertion

??? note "Quel algorithme appliquer pour insérer un élément dans un ABR ?" On applique le même algorithme que pour rechercher un élément, sauf que lorsque l'on arrive sur un sous-arbre vide, on le remplace par un sous-arbre ayant la valeur à insérer comme racine.

!!! abstract "En appliquant ces algorithmes, rechercher 6 et 3, et insérer 10 et 1 dans l'ABR suivant :"

```
```mermaid
graph TB
  A[8] --> B[4]
  A --> C[12]
  B --> D[2]
  B --> E[6]
  E --> F[7]
  C --> G[9]
  C --> H[15]
  H --> J[20]
```
```

```
??? note "Solution"
    - 6<8 -> sous-arbre gauche -> 6>4 -> sous-arbre droit -> 6 trouvé
     - 3<8 -> sous-arbre gauche -> 3<4 -> sous-arbre gauche -> 3>2 -> sous-arbre
droit -> arbre vide : valeur non-trouvée.
    ```mermaid
    graph TB
      A[8] --> B[4]
      A --> C[12]
      B --> D[2]
      D --> K[1]
      B --> E[6]
      E --> F[7]
      C --> G[9]
      G --> I[10]
      C --> H[15]
      H --> J[20]
    ```
```

TD : Propriétés et algorithmique sur les arbres

Ce TD traite de deux exemples d'arbres, dont on extrait les propriétés et sur lesquels on applique les différents parcours.

A. L'arbre de la nouvelle année

On considère l'arbre suivant :



A.I. Propriétés

- Donner la **taille** et la **hauteur** de cet arbre.

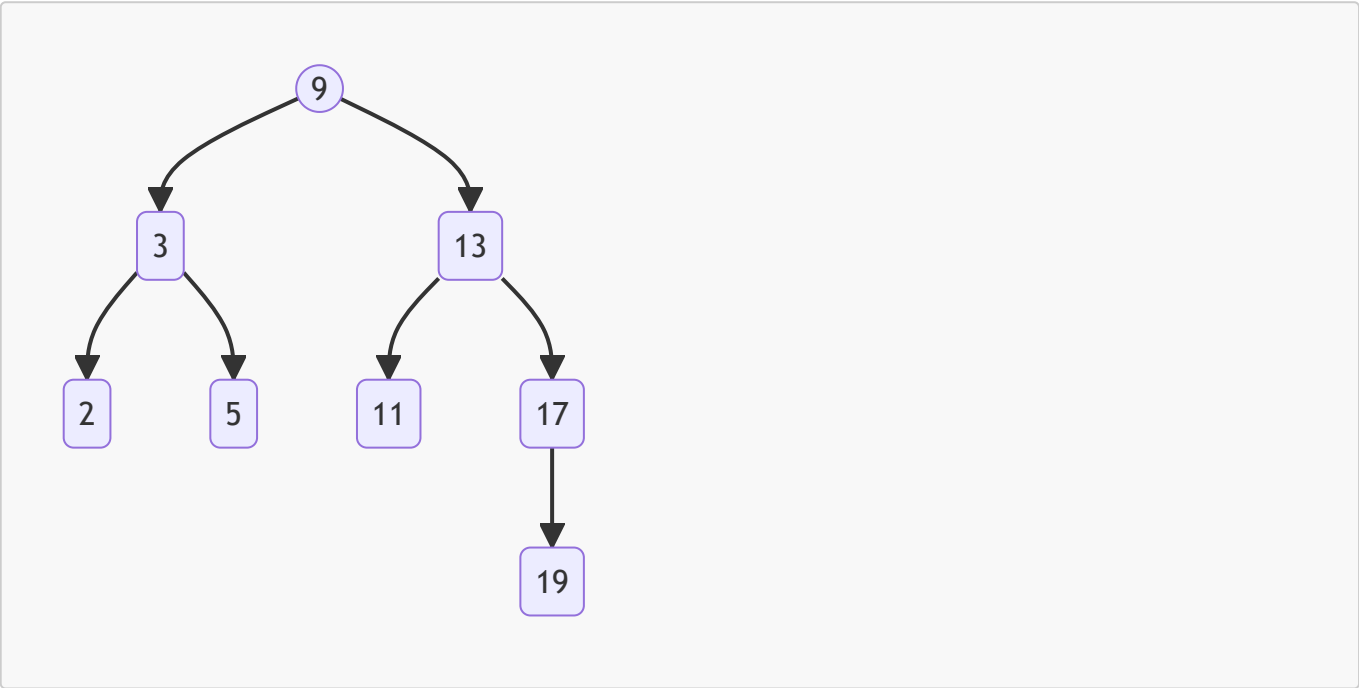
- Est-ce un **arbre binaire** ? Un **arbre binaire de recherche** ? Justifier.

A.II. Parcours

- Donner les **parcours en profondeur** préfixe, infixe et suffixe de cet arbre.
- Donner le **parcours en largeur** de cet arbre.

B. Arbre de recherche

On considère l'arbre suivant :



1. Justifier qu'il s'agit bien d'un **arbre binaire de recherche**.
2. Appliquer l'**algorithme d'insertion** pour y ajouter le chiffre 7.
3. Appliquer l'**algorithme de recherche** pour trouver les nombres 11 et 14.
4. Donner les **parcours en profondeur** préfixe, indixe et suffixe de cet arbre. Quel parcours permet d'obtenir la liste des nombres dans l'ordre croissant ?
5. Donner le **parcours en largeur** de cet arbre.

TP : Algorithmes sur les arbres binaires

!!! tip "Dans ce TP, nous utilisons la classe `ArbreBinaire` définie dans le TP sur l'implémentation des arbres binaires, et l'arbre d'exemple du même TP."

A. Calcul de mesures

B.I. La taille

1. On propose une description d'un algorithme **récursif** permettant de calculer la taille d'un arbre binaire. A partir de celle-ci, proposer un pseudo-code de l'algorithme.

!!! note "Description"
La taille de l'arbre correspond à la **somme de la taille de ses sous-arbres**. On parcourt donc les sous-arbres gauche et droit récursivement, et on incrémente la taille à chaque fois que l'on rencontre un nouveau nœud. L'algorithme s'arrête lorsque l'appel récursif se fait sur un arbre vide.

2. Tester cet algorithme en l'implémentant en Python et en utilisant l'arbre d'exemple implémenté précédemment avec la classe `ArbreBinaire`. N'oubliez pas que vous pouvez afficher le résultat avec la méthode `affiche`.

B.II. La hauteur

!!! note "Description" L'algorithme qui permet de calculer la **hauteur** d'un arbre binaire est similaire à celui utilisé pour la taille. La différence est qu'à chaque niveau de profondeur atteint (chaque nouvel appel récursif), on incrémente de 1 la hauteur, et on sélectionne le maximum des hauteurs entre celle du sous-arbre gauche et celle du sous-arbre droit.

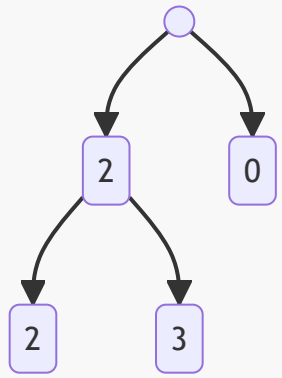
1. Proposer un pseudo-code de cet algorithme.
2. L'implémenter sur le même arbre qu'en I.A.

N.B.: on pourra utiliser la fonction `max` de Python pour sélectionner la plus grande des deux hauteurs.

B. II. Les parcours

II.A. Parcours en profondeur

1. Donner les parcours préfixe, infixe et postfixe de l'arbre d'exemple du TP précédent :



On propose une fonction qui effectue un de ces parcours en profondeur d'un arbre binaire et stocke le résultat dans une liste Python.

```
def parcours(arbre):
    if arbre == None:
        return []
    else:
```

```
        r = arbre.getRacine()
        ag = arbre.getABGauche()
        ad = arbre.getABDroit()
        return parcours(ag) + [r] + parcours(ad)
```

2. A quel parcours ce code-correspond-il ? Justifier.
3. Combien de fois accède-t-on à chaque nœud de l'arbre ? En déduire la complexité de cet algorithme en fonction de la taille de l'arbre n.
4. Modifier le code pour obtenir les deux autres parcours en profondeur.
5. Tester ces parcours sur l'arbre d'exemple et vérifier leur bon fonctionnement grâce aux résultats obtenus "à la main".
6. Proposer une autre version de chacun de ces parcours, qui ne renvoie pas une liste Python mais *affiche* au fur et à mesure les nœuds de l'arbre rencontrés.

II.B. Parcours en largeur

On fournit un code du parcours en largeur d'un arbre binaire.

```
def parcoursLargeur(arbre):
    f = File()
    parcours = []
    f.enfiler(arbre)

    while not(f.vide()):
        e = f.defiler()
        parcours.append(e.getRacine())
        if e.getABGauche() != None:
            f.enfiler(e.getABGauche())
        if e.getABDroit() != None:
            f.enfiler(e.getABDroit())

    return parcours
```

1. Identifier les types des entrées et sorties, ainsi que celui de la variable temporaire.
2. Indiquer le contenu des variables `e`, `f` et `parcours` à l'initialisation (code exécuté jusqu'à la ligne 4) et à la fin de chaque passage de boucle lorsque la fonction est appliquée à l'arbre d'exemple utilisé dans les parties précédentes. On représentera l'arbre schématiquement.
3. Quelle est l'utilité de `f` ?
4. Recopier cette fonction sur ordinateur en ajoutant sa spécification *(les types des variables d'entrée et sortie)*. On n'oubliera pas d'importer une implémentation de structure de file.
5. Tester le programme sur l'arbre d'exemple.

TP : Algorithmes sur les arbres binaires de recherche

A. Représentation

On propose d'utiliser une autre représentation pour manipuler des arbres binaires de recherche. On utilise deux classes : `Noeud` et `ABR`.

```
class ABR:
    def __init__(self, racine):
        self.racine = racine

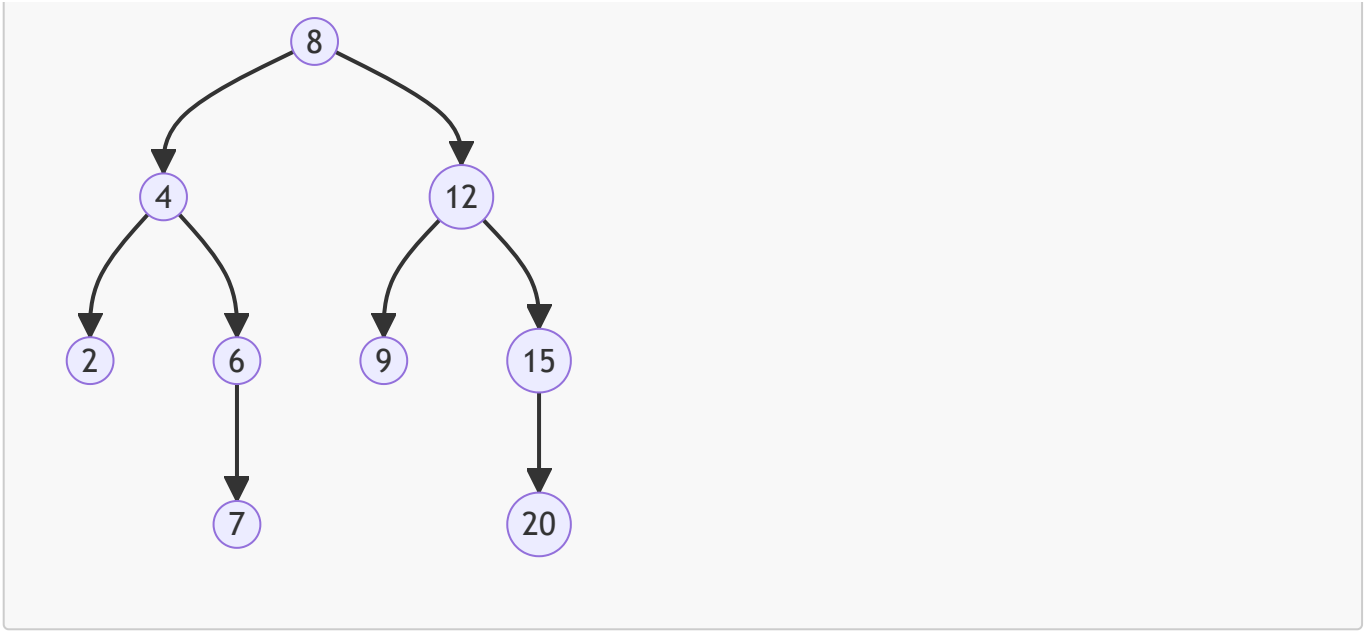
    def affiche_abr(self):
        if self is None:
            representation = "L'arbre est vide !"
        else:
            representation = self.racine.affiche()
        return representation

class Noeud:
    def __init__(self, valeur, fils_gauche, fils_droit):
        self.valeur = valeur
        self.fils_gauche = fils_gauche
        self.fils_droit = fils_droit

    def affiche(self, space = 0):
        spaces = "  "*space
        print(spaces, self.valeur)
        if self.fils_gauche:
            self.fils_gauche.affiche(space+1)
        if self.fils_droit:
            self.fils_droit.affiche(space+1)
```

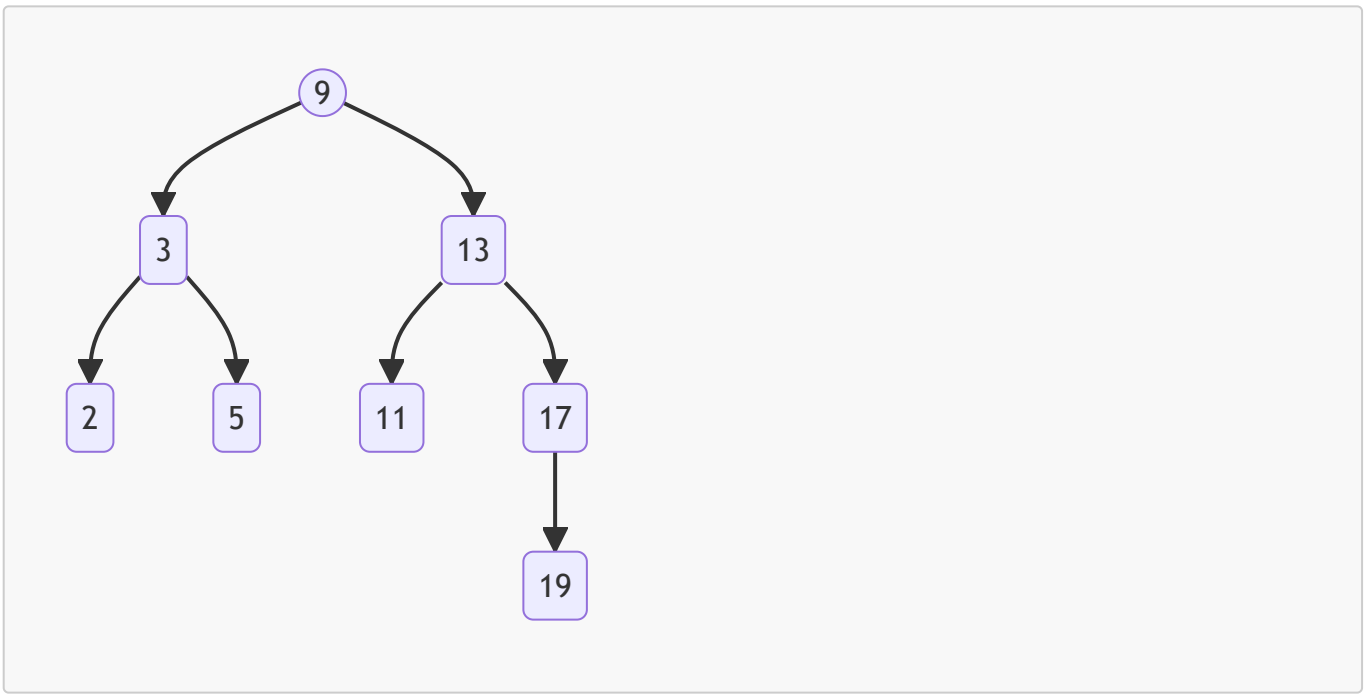
Exemple d'utilisation des classes : Le code suivant représente l'arbre ci-dessous :

```
n7 = Noeud (7, None, None)
n6 = Noeud (6, None, n7)
n2 = Noeud (2, None, None)
n4 = Noeud ( 4, n2, n6)
n20 = Noeud (20, None, None)
n15 = Noeud (15, None, n20)
n9 = Noeud (9, None, None)
n12 = Noeud (12, n9, n15)
n8 = Noeud (8, n4, n12 )
a = ABR(n8)
```



!!! tip "Récupérer le fichier suivant sur l'ENT."

1. Utiliser ces classes pour représenter l'ABR du TD précédent (ci-dessous), en vous aidant de l'exemple.



2. Afficher l'arbre obtenu en utilisant la méthode `affiche_abr`.

B. Algorithme de recherche

Nous allons cette fois-ci écrire des méthodes pour les deux classes, *en nous inspirant de ce qui a été fait pour la fonction d'affichage (implémentée dans les deux classes)*.

Le principe est d'effectuer la recherche dans la classe `Noeud` (en considérant qu'on peut la faire en partant de n'importe quel noeud de l'ABR) et donc de créer une méthode `rechercher` dans celle-ci, puis de créer une méthode `rechercher_abr` dans la classe `ABR`, en appelant la méthode `rechercher` de la classe `Noeud` sur la racine de l'ABR.

1. Implémenter ces méthodes pour rechercher une clé dans un ABR, à partir de l'algorithme décrit dans le cours.

2. Les tester sur l'ABR d'exemple utilisé dans les TP précédents.

C. Algorithme d'insertion

De la même manière que pour la recherche, implémenter une méthode `inserer` dans la classe `Noeud` et une dans la classe `ABR`, `inserer_abr`, qui permettent d'insérer une clé. Les tester sur le même ABR que l'algorithme de recherche.