# Les arbres binaires de recherche

Numérique et Sciences Informatiques - Lycée P. Méchain

## Notion d'arbre binaire de recherche

Un **arbre binaire de recherche** (**ABR** ou **BST** en anglais, Binary Search Tree) est un arbre binaire dont chaque nœud possède une clé et tel que la clé de chaque nœud est à la fois

- supérieure (ou égale) à la clé de tous les nœuds enfants qui sont à sa gauche ;
- inférieure à toutes les clé des nœuds enfants qui sont à sa droite.

Les nœuds dans un **ABR** sont donc ordonnés de manière à ce que :

- les enfants à gauche d'un nœud ont des clés inférieures (ou égales) à lui ;
- les enfants à droite d'un nœud ont des clés supérieures à lui.

Cela doit être vrai pour chaque nœud de l'arbre.

**Exemple :** créer un ABR en insérant les nombres 5, 3, 8, 6, 2, 4


FIGURE 1 – Insersion de 5

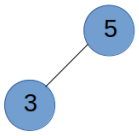


FIGURE 2 – Insersion de 3

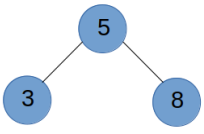


FIGURE 3 – Insersion de 8

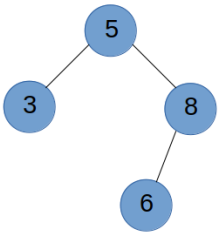


FIGURE 4 – Insersion de 6

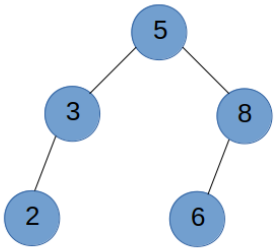


FIGURE 5 – Insersion de 2

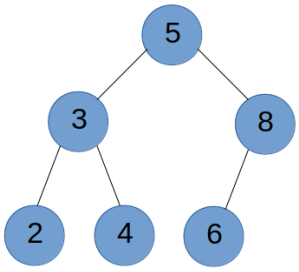


FIGURE 6 – Insersion de 4

### Les doublons

• Une même clé peut apparaître plusieurs fois ; elle est alors être insérée dans le sous-arbre gauche (comme proposé dans la définition d'un arbre binaire page précédente).

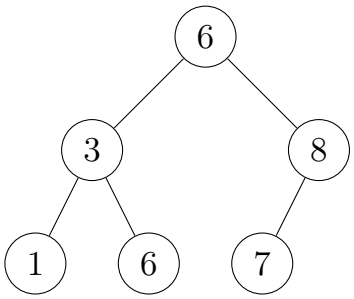


FIGURE 7 – Exemple d'ABR dans lequel une clé apparaît plusieurs fois.

• On peut également vouloir réaliser des ensembles où chaque clé apparaît une fois. C'est le choix que l'on fera dans la suite de ce document.

### Les arbres équilibrés

Un arbre binaire est complètement équilibré si tous ses niveaux sont complets, sauf éventuellement le dernier.

• Rechercher une clé dans un ABR équilibré est semblable à une recherche dichotomique dans un tableau trié : on divise la zone de recherche en deux à chaque comparaison. La complexité est donc en $O(log(n))$ ou $O(log_2(n))$ avec n la taille de l'arbre.
Rechercher une clé dans un ABR non équilibré est moins efficace, avec une complexité en O(n) dans le pire des cas (arbre dégénéré ou arbre peigne).

• Insérer une nouvelle clé dans un arbre équilibré est de complexité $O(log(n))$. Après l'insertion de nouveaux éléments, il se peut que l'arbre ne soit plus équilibré. Après insertion d'un nouveau nœud l'arbre est donc modifié afin de conserver l'arbre équilibré.

### Principe pour rechercher une clé

On considère ici que toutes les clés sont distinctes. On utilisera une méthode ou une fonction récursive.
La démarche est la suivante :

- si la racine n'est pas vide alors :
  - si la clé recherchée est égale à celle de la racine, fin de la recherche ;
  - si la clé recherchée est inférieure à celle du nœud, chercher la clé dans le sous-arbre gauche ;
  - si la clé recherchée est supérieure à celle du nœud, chercher la clé dans le sous-arbre droit ;

### Méthode pour insérer une clé

Pour insérer une clé dans l'arbre binaire on l'insérera au niveau des feuilles en se basant sur le même principe que la recherche d'une clé.

On ne se préoccupera pas de l'équilibrage de l'arbre (hors programme)

# Exercices

## Exercice 1 : parcours d'un ABR

1. **Donner** la représentation graphique d'un ABR en insérant les clés 6, 3, 1, 7, 5, 9, 8, 2, 4 (dans cet ordre)
2. **Donner** l'ordre des nœuds visités lors d'un parcours en largeur de l'arbre.
3. **Donner** l'ordre des nœuds visités pour les parcours préfixe, postfixe puis infixe de l'arbre.
4. **Donner** le parcours à utiliser (en largeur, en profondeur préfixe, en profondeur infixe, en profondeur suffixe) afin d'afficher les clés dans l'ordre croissant.

## Exercice 2 : implémenter un ABR avec une approche objet

On se propose d'implémenter un arbre binaire de recherche à partir de la classe ABR ci-dessous.

Attributs :

— **racine** : la clé de la racine (type int)
— **gauche** : sous-arbre binaire gauche (type ABR)
— **droit** : sous-arbre binaire gauche (type ABR)

Interface de la classe ABR (signature des méthodes publiques) :

— **est_vide() $->$** Bool : renvoie True si l'arbre est vide ;
— **inserer(cle : int) $->$** None : insertion d'une clé dans l'arbre ;
— **afficher() $->$** None : affiche les clés de l'ABR dans l'ordre croissant ;
— **rechercher(cle : int) $->$** Bool : renvoie *True* si la clé est dans l'arbre et *False* sinon.

1. **Compléter** le code Python de la classe ABR ci-dessous.

```
class ABR:
    '''Un ABR référence un noeud ou rien. On suppose qu'un ABR ne peut
       contenir qu'une seule occurrence de chaque valeur.'''
    def __init__(self, cle: int = None):
        self.racine = cle
        self.gauche = None
        self.droit = None

    def est_vide(self):
    '''Renvoie True si l'arbre est vide sinon False.'''
        return self.racine is None

    def inserer(self, cle: int):
        '''Insertion d'une clé dans l'arbre.'''

     def afficher(self):
        '''Affiche les clés de l'ABR dans l'ordre croissant'''

     def rechercher(self, cle: int):
        '''Renvoie True si la clé est dans l'arbre et False sinon'''
```

2. **Écrire** le code Python qui permet de créer un ABR à partir de nombres générés aléatoirement.
3. **Tester** les méthodes *rechercher* et *afficher*.

## Exercice 3 : implémenter un ABR avec une approche fonctionnelle

On utilise dans cet exercice un arbre binaire de recherche (ABR) pour stocker un ensemble ordonné de clés C de type *String* ainsi qu'un ensemble de valeurs V (de type Int).
L'arbre binaire sera étiqueté par un tuple (C, V).

On souhaite implémenter 2 opérations sur cet arbre :

- l'insertion d'un nouveau couple clé/valeur ;
- la recherche d'une valeur associée à une clé donnée.

Pour cela on utilisera une approche fonctionnelle dans laquelle les structures de données donc les instances de la classe Nœud (voir ci-dessous) sont immuables. Un objet crée peut être utilisé pour consulter son contenu ou pour créer un nouvel arbre mais ne peut pas être modifié.
Ainsi on ne va pas modifier l'arbre reçu en argument par la fonction qui permet d'insérer un nœud mais on va en renvoyer un nouveau.

### Implémentation d'un nœud en Python

On utilisera la classe Nœud ci-dessous :

```
class Nœud:
    """Un nœud d'un arbre binaire"""
    def __init__(self, g, e, d):
        self.gauche = g
        self.etiquette = e
        self.droit = d
```

1. **Écrire** le code Python de la fonction *inserer* dont la signature est :

   ```
   inserer(arbre: Nœud, etiquette: tuple) -> Nœud
   ```

2. **Définir** l'ordre dans lequel insérer à l'aide de la fonction *inserer* les étiquettes ci-dessous afin d'obtenir un ABR parfaitement équilibré. On utilisera le nom de la ville (sans accents) comme clé et le nombre d'habitants comme valeur. **Représenter** l'arbre obtenu.

   - ('saint-quentin', 55649)
   - ('soissons', 28410)
   - ('laon', 25358)
   - ('château-thierry', 14602)
   - ('tergnier', 13734)
   - ('chauny', 11831)
   - ('villers-cotterêts', 10951)
   - ('hirson', 9158)
   - ('bohain-en-vermandois', 5670)
   - ('gauchy', 5335)
   - ('guise', 4919)
   - ('delleu', 3780)
   - ('saint-michel', 3476)
   - ('fère-en-tardenois', 3143)
   - ('fresnoy-le-grand', 2956)

3. **Écrire** le code Python qui permet de créer l'ABR et d'insérer les étiquettes de manière à avoir un arbre parfaitement équilibré.
4. **Écrire** le code Python de la fonction $affficher$ qui permet d'afficher les villes et le nombre d'habitants (dans l'ordre croissant des villes). Signature de la fonction $affficher$ :

   ```
   afficher(arbre: Nœud)
   ```

5. **Écrire** le code Python de la fonction *rechercher_ville* dont la signature est :

   ```
   rechercher_ville(arbre: Nœud, ville: String)
   ```

   et qui renvoie le nombre d'habitants.