Exercice 1 (4 points).

Cet exercice porte sur les listes, les arbres binaires de recherche et la programmation orientée objet.

Lors d'une compétition de kayak, chaque concurrent doit descendre le même cours d'eau en passant dans des portes en un minimum de temps. Si le concurrent touche une porte, il se voit octroyé une pénalité en secondes. Son résultat final est le temps qu'il a mis pour descendre le cours d'eau auquel est ajouté l'ensemble des pénalités qu'il a subies.
Un gestionnaire de course de kayak développe un programme Python pour gérer les résultats lors d'une compétition.
Dans ce programme, pour modéliser les concurrents et leurs résultats, une classe `Concurrent` est définie avec les attributs suivants :

- `nom` de type `Str` qui représente le pseudonyme du compétiteur ;
- `temps` de type `Float` qui est le temps mis pour réaliser le parcours en secondes ;
- `penalites` de type `Int` qui est le nombre de secondes de pénalité cumulées octroyées au concurrent ;
- `temps_tot` de type `Float` qui correspond au temps total, c'est-à-dire au temps mis pour réaliser le parcours auquel on a ajouté les pénalités.

On suppose que tous les concurrents ont des temps différents dans cet exercice.

Le code Python incomplet de la classe `Concurrent` est donné ci-dessous.

```
class Concurrent:
    def __init__(self, pseudo, temps, penalite):
        self.nom = pseudo
        self.temps = temps
        self.penalite = ...
        self.temps_tot = ...
```

1. (a) Recopier et compléter le code du constructeur de la classe `Concurrent`.

   On exécute l'instruction suivante : `c1 = Concurrent("Mosquito", 87.67, 12)`

   (b) Donner la valeur de l'attribut `temps_tot` de `c1`.

   (c) Donner l'instruction permettant d'accéder à la valeur `temps_tot` de `c1`.

2. Pendant la course, des instances de la classe `Concurrent` sont créées au fur et à mesure des arrivées des concurrents.
   On définit une classe `Liste` pour les stocker au fur et à mesure. Cette classe implémente la structure de données abstraite liste dont l'interface est munie :

   — du constructeur qui ne prend pas de paramètre et qui crée une liste vide.
   Exemple : `L = Liste()`

   — de la méthode `est_vide` qui ne prend pas de paramètre et qui renvoie un booléen : `True` si la liste est vide ou `False` sinon.
   Exemple : On considère la liste `L = <c1, c2, c3>` ou `c1`, `c2` et `c3` sont des instances de `Concurrent`.
   L'appel `L.est_vide()` renvoie `False`.

— de la méthode `tete` qui ne prend pas de paramètre et qui renvoie un objet de type `Concurrent` ayant pour valeur le premier élément de la liste. Cet élément sera appelé tête de la liste dans la suite de l'exercice.

Cette méthode ne s'applique que sur des listes non vides.

Exemple : On considère la liste `L = <c1, c2, c3>` ou `c1`, `c2` et `c3` sont des instances de `Concurrent`.

`L.tete()` a pour valeur `c1`.

Remarque : Après exécution de `L.tete()`, la liste `L` reste inchangée et vaut toujours `<c1, c2, c3>`.

— de la méthode `queue` qui ne prend pas de paramètre. Cette méthode renvoie la liste sur laquelle elle s'applique privée de son premier élément.

Cette méthode ne s'applique que sur des listes non vides.

Exemple : On considère la liste `L = <c1, c2, c3>`

L'appel `L.queue()` renvoie la liste `<c2, c3>`

Remarque : Après exécution de `L.queue()`, la liste `L` reste inchangée et vaut toujours `<c1, c2, c3>`.

— de la méthode `ajout` qui prend en paramètre un concurrent `c` et qui modifie la liste sur laquelle elle s'applique, en ajoutant `c` en tête.

Exemple 1 :

Si `L` est la liste vide, `L.ajout(c)` modifie la liste `L` qui devient `<c>`

Exemple 2 :

Si `L` est la liste `<c1, c2, c3>`, `L.ajout(c)` modifie la liste `L` qui devient `<c, c1, c2, c3>`

On considère le script Python suivant :

```
1 c1= Concurrent("Mosquito",87.67,12)
2 c2= Concurrent("Python Fute",89.73,4)
3 c3= Concurrent("Piranha Vorace",90.54,0)
4 c4= Concurrent("Truite Agile",84.32,52)
5 c5= Concurrent("Tortue Rapide",92.12,2)
6 c6= Concurrent("Lievre Tranquille",93.45,0)
7
8 resultats=Liste()
9 resultats.ajout(c1)
10 resultats.ajout(c2)
11 resultats.ajout(c3)
12 resultats.ajout(c4)
13 resultats.ajout(c5)
14 resultats.ajout(c6)
```

Après exécution, ce script génère une liste `resultats` que l'on peut représenter par :

`<c6, c5, c4, c3, c2, c1>`

(a) On considère la liste `resultats` ci-dessus.
Donner la ou les instruction(s) qui permet(tent) d'accéder à `c4`.

(b) Donner la ou les instruction(s) qui permet(tent) d'accéder au temps total du concurrent stocké en tête de la liste `resultats`.

3. On souhaite créer une fonction `meilleur_concurrent` qui prend en paramètre une liste `L` de concurrents et qui renvoie l'objet `Concurrent` correspondant au concurrent le plus rapide. On suppose que la liste est non vide.

Recopier et compléter le code Python, donné ci-dessous, de la fonction `meilleur_concurrent`.

```
def meilleur_concurrent(L) :
        conc_mini = L.  ...
        mini = conc_mini.temps_tot
        Q = L.queue()
        while not(Q.est_vide()):
                elt = Q.tete()
                if elt.temps_tot ... mini :
                        conc_mini = elt
                        mini = elt.temps_tot
                Q = Q.  ...
        return ...
```

4. Pour simplifier le stockage des résultats, on décide de stocker les objets de la classe `Concurrent` dans un arbre binaire de recherche. Chaque nœud de cet arbre est donc un objet `Concurrent`.

Dans cet arbre binaire de recherche, en tout nœud :

— le concurrent enfant à gauche est plus rapide que le nœud ;

— le concurrent enfant à droite est moins rapide que le nœud.

Pour implémenter la structure d'arbre binaire de recherche, on dispose d'une classe `Arbre` munie, entre autres, d'une méthode `ajout` qui prend en paramètre un objet `c` de type `Concurrent` et qui modifie l'arbre binaire sur lequel elle s'applique, en y ajoutant le concurrent `c` tout en maintenant la propriété d'arbre binaire de recherche.

On ajoute dans un arbre vide successivement les concurrents de la liste `resultats` en partant de la tête de la liste (soit, dans le cas présent, `c6`, puis `c5`, puis `c4`,...).

Dessiner l'arbre binaire de recherche obtenu. On rappelle le temps total de chaque concurrent :

Concurrent	c6	c5	c4	c3	c2	c1
temps_tot	93,45	94,12	136,32	90,54	93,73	99,67