Exercice 3 (4 points)

Cet exercice porte sur la programmation orientée objet et sur les arbres. Il contient deux parties indépendantes.

Lors de la création d'un jeu vidéo, un développeur décide d'utiliser la programmation orientée objet. Au cours de ce jeu, différents personnages vont s'affronter tour à tour. Pour cela, le développeur décide de créer une classe `Personnage`.

Un personnage est caractérisé par les données suivantes :

- `clan` : chaine de caractères qui identifie le clan auquel le personnage appartient ;
- `vie` : nombre entier qui représente le nombre de points de vie du personnage. Un personnage n'est plus actif dès que le nombre de points de vie devient inférieur ou égal à zéro ;
- `force` : nombre entier qui représente le nombre de points de force.

Les personnages peuvent effectuer différentes actions : attaquer un ennemi, se défendre face à un ennemi, etc.

Partie A : programmation orientée objet

1. Recopier et compléter les lignes de code numérotées de 2 à 5 :

```
1  class Personnage:
2      def __init__(..., nom_clan, pts_vie, pts_force):
3          ...clan = nom_clan
4          ...vie = pts_vie
5          ...force = pts_force
6
7      def attaque(...):
8          ...
9
10     def defense(...):
11         ...
```

2. Donner les attributs ainsi que les méthodes de cette classe.

3. Écrire l'instruction qui permet d'instancier le personnage `luther` du clan `"Umbrella"` avec 100 points de vie et 15 points de force.

4. Écrire le code Python de la méthode `attaque` correspondant à la description suivante :

- elle s'applique sur un personnage (l'attaquant) et prend en paramètre `autre_perso` (le personnage attaqué) ;
- elle abaisse le nombre de points de vie du personnage attaqué d'une valeur égale au nombre de points de force du personnage qui attaque ;

- elle renvoie 1 s'il reste des points de vie au personnage attaqué, 0 sinon.

Exemple : si le personnage `luther` qui a 15 points de force attaque le personnage `rayleigh` qui a 120 points de vie alors l'appel `luther.attaque(rayleigh)` permet de mettre à jour le nombre de points de vie de `rayleigh`, soit 120 – 15 = 105 points de vie. La méthode renvoie 1 dans ce cas puisqu'il reste des points de vie à `rayleigh`.

5. Le développeur souhaite garder en mémoire toutes les actions réalisées par un personnage afin de pouvoir les annuler les unes après les autres et ainsi revenir à un état antérieur du personnage.

Indiquer sans la justifier la structure de données la plus pertinente à utiliser pour garder en mémoire les différentes actions du personnage.

Partie B : arbres binaires de recherche

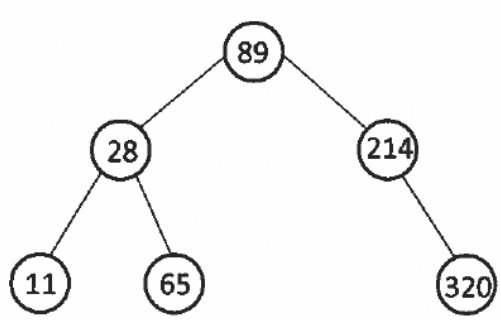
Dans cette partie :

- les arbres binaires de recherche ne peuvent pas contenir plusieurs fois la même clé ;
- un arbre binaire de recherche limité à un nœud a une hauteur égale à 1 ;
- la taille d'un arbre est le nombre de nœuds de cet arbre ;
- la valeur d'un nœud (aussi appelée clé) est :
  - strictement supérieure à celle des nœuds de son sous-arbre gauche,
  - strictement inférieure à celle des nœuds de son sous-arbre droit,
- la racine d'un arbre binaire est le seul nœud n'ayant pas de parent.

On dispose de l'interface suivante pour manipuler les arbres binaires de recherche :

`est_vide :`	arbre → booléen	renvoie vrai si l'arbre est vide, faux sinon
`gauche :`	arbre → arbre	renvoie le sous-arbre gauche
`droit :`	arbre → arbre	renvoie le sous-arbre droit
`valeur_racine :`	arbre → valeur	renvoie la valeur de la racine de l'arbre

6. On considère l'arbre binaire de recherche ci-dessous :



a) Donner sans justification la hauteur et la taille de l'arbre.

b) On insère dans l'arbre binaire de recherche ci-avant la clé dont la valeur est 46. Représenter l'arbre obtenu après cette insertion.

7. On propose la fonction `parcours` ci-dessous, en langage Python :

```
1  def parcours(arbre):
2      if not est_vide(arbre):
3          parcours(gauche(arbre))
4          print(valeur_racine(arbre))
5          parcours(droit(arbre))
```

a) Donner le nom de ce parcours.

b) En considérant l'arbre donné en exemple (sans l'ajout de la valeur 46), indiquer l'ordre dans lequel les valeurs des nœuds seront affichés.

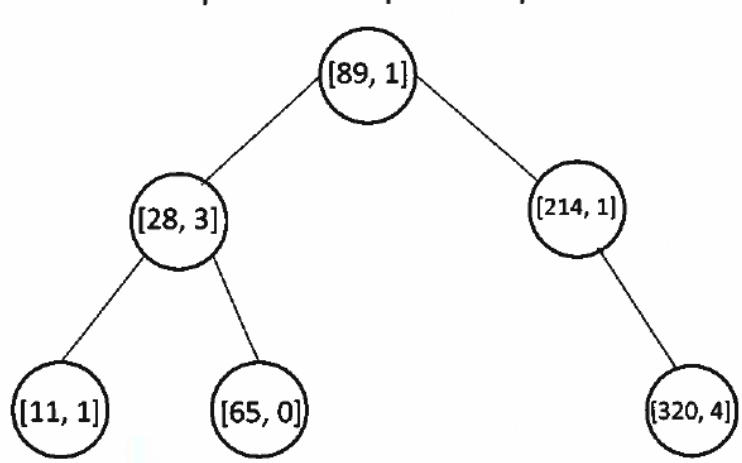
8. Pour le jeu vidéo défini dans la partie A, on souhaite collecter l'équipement d'un personnage (vêtements, armes, nourriture, ...) en utilisant un arbre binaire de recherche.
Chaque type de matériel est identifié par un nombre entier. D'autre part, un personnage peut disposer d'un même matériel en plusieurs exemplaires ; il peut par exemple posséder 12 flèches, toutes de même type (et possédant donc la même référence).

Dans cette question, les clés sont des tableaux non dynamiques contenant deux informations :

- un premier entier (la référence du matériel),
- un deuxième entier (la quantité correspondante pour ce matériel).

Le placement dans l'arbre binaire de recherche se fait en fonction de la référence du matériel (premier élément du tableau).

L'arbre donné précédemment peut devenir par exemple :



Si `cle` vaut `[28, 3]`, alors `cle[0]` vaut `28` et `cle[1]` vaut `3`.

La fonction `recherche_dichotomique` prend en paramètre un arbre binaire de recherche et un entier (référence de matériel) et renvoie la quantité associée au matériel si celui-ci est présent dans l'arbre, - 1 sinon.

Recopier et compléter la fonction `recherche_dichotomique` suivante :

```
1  def recherche_dichotomique(arbre, ref_materiel):
2      """
3      Entrée : un arbre binaire de recherche dont les nœuds
4               sont des tableaux ;
5               un entier (la référence du matériel recherché).
6      Sortie : un entier -1 si la référence du matériel
7               n'est pas présente dans l'arbre ou la quantité
8               si la référence est présente.
9      """
10     if est_vide(arbre):    #  référence absente de l'arbre
11         return ...
12     elif ref_materiel < valeur_racine(arbre)[...]:# recherche
13         return ...                  # dans le sous-arbre gauche
14     elif ref_materiel > valeur_racine(arbre)[...]:# recherche
15         return ...                  # dans le sous-arbre droit
16     else:
17         # référence présente, on renvoie la quantité
18         #correspondante
19         return valeur_racine(arbre)[...]
```