BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

BAC BLANC

Sujet 1

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Le candidat traite les trois exercices proposés

Durée de l'épreuve : 3 heures 30

L'usage de la calculatrice n'est pas autorisé.

Dès que ce sujet vous est remis, assurez-vous qu'il est complet.

Ce sujet comporte 9 pages numérotées de 1/9 à 9/9

Le sujet est composé de 3 exercices indépendants.

EXERCICE 1 (10 points)

L'exercice porte sur les arbres binaires de recherche et la programmation objet.

Dans un entrepôt de e-commerce, un robot mobile autonome exécute successivement les tâches qu'il reçoit tout au long de la journée.

La mémorisation et la gestion de ces tâches sont assurées par une structure de données.

1. Dans l'hypothèse où les tâches devraient être extraites de cette structure (pour être exécutées) dans le même ordre qu'elles ont été mémorisées, préciser si ce fonctionnement traduit le comportement d'une file ou d'une pile. Justifier.

En réalité, selon l'urgence des tâches à effectuer, on associe à chacune d'elles, lors de la mémorisation, un indice de priorité (nombre entier) distinct : il n'y a pas de valeur en double.

Plus cet indice est faible, plus la tâche doit être traitée prioritairement.

La structure de données retenue est assimilée à un arbre binaire de recherche (ABR) dans lequel chaque nœud correspond à une tâche caractérisée par son indice de priorité.

Rappel : Dans un arbre binaire de recherche, chaque nœud est caractérisé par une valeur (ici l'indice de priorité), telle que chaque nœud du sous-arbre gauche a une valeur strictement inférieure à celle du nœud considéré, et que chaque nœud du sousarbre droit possède une valeur strictement supérieure à celle-ci. Cette structure de données présente l'avantage de mettre efficacement en œuvre l'insertion ou la suppression de nœuds, ainsi que la recherche d'une valeur.

Par exemple, le robot a reçu successivement, dans l'ordre, des tâches d'indice de priorité 12, 6, 10, 14, 8 et 13. En partant d'un arbre binaire de recherche vide, l'insertion des différentes priorités dans cet arbre donne la figure 1.

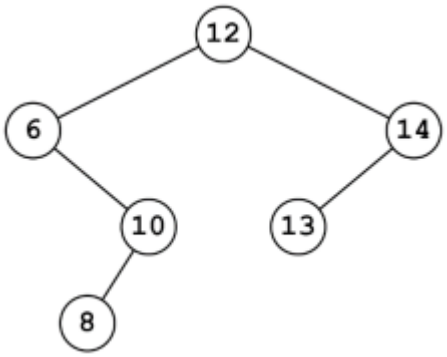

Figure 1 : Exemple d'un arbre binaire

2. En utilisant le vocabulaire couramment utilisé pour les arbres, préciser le terme qui correspond :

a. au nombre de tâches restant à effectuer, c'est-à-dire le nombre total de nœuds de l'arbre ;

b. au nœud représentant la tâche restant à effectuer la plus ancienne ;

c. au nœud représentant la dernière tâche mémorisée (la plus récente).

3. Lorsque le robot reçoit une nouvelle tâche, on déclare un nouvel objet, instance de la classe *Noeud*, puis on l'insère dans l'arbre binaire de recherche (instance de la classe *ABR*) du robot. Ces 2 classes sont définies comme suit :

```
class Noeud:
def __init__(self, tache, indice):
        self.tache = tache #ce que doit accomplir le robot
        self.indice = indice #indice de priorité (int)
        self.gauche = ABR() #sous-arbre gauche vide (ABR)
        self.droite = ABR() #sous-arbre droit vide (ABR)
class ABR:
    # arbre binaire de recherche initialement vide
    def __init__(self):
        self.racine = None #arbre vide
        #Remarque : si l'arbre n'est pas vide, racine est
        #une instance de la classe Nœud

    def est_vide(self):
        """renvoie True si l'arbre autoréférencé est vide, False sinon"""
        return self.racine == None

    def insere(self, nouveau_noeud):
        """insere un nouveau noeud, instance de la classe Noeud, dans l'ABR"""
        if self.est_vide():
            self.racine = nouveau_noeud
        elif self.racine.indice …… nouveau_noeud.indice
            self.racine.gauche.insere(nouveau_noeud)
        else:
            self.racine.droite.insere(nouveau_noeud)
```

a. Donner les noms des attributs de la classe Noeud.

b. Expliquer en quoi la méthode insere est dite récursive et justifier rapidement qu'elle se termine.

c. Indiquer le symbole de comparaison manquant dans le test à la ligne 26 de la méthode insere pour que l'arbre binaire de recherche réponde bien à la définition de l'encadré « Rappel » de la page 6

d. On considère le robot dont la liste des tâches est représentée par l'arbre de la figure 1. Ce robot reçoit, successivement et dans l'ordre, des tâches d'indice de priorité 11, 5, 16 et 7, sans avoir accompli la moindre tâche entretemps. Recopier et compléter la figure 1 après l'insertion de ces nouvelles tâches.

4. Avant d'insérer une nouvelle tâche dans l'arbre binaire de recherche, il faut s'assurer que son indice de priorité n'est pas déjà présent.

Écrire une méthode *est_present* de la classe *ABR* qui répond à la description :

```
def est_present(self, indice_recherche) :
    """renvoie True si l'indice de priorité indice_recherche (int) passé en paramètre est déjà
    l'indice d'un nœud de l'arbre, False sinon"""
```

5. Comme le robot doit toujours traiter la tâche dont l'indice de priorité est le plus petit, on envisage un parcours infixe de l'arbre binaire de recherche.

a. Donner l'ordre des indices de priorité obtenus à l'aide d'un parcours infixe de l'arbre binaire de recherche de la figure 1.

b. Expliquer comment exploiter ce parcours pour déterminer la tâche prioritaire.

6. Afin de ne pas parcourir tout l'arbre, il est plus efficace de rechercher la tâche du nœud situé le plus à gauche de l'arbre binaire de recherche : il correspond à la tâche prioritaire.

Recopier et compléter la méthode récursive *tache_prioritaire* de la classe *ABR*:

```
def tache_prioritaire(self):
    """renvoie la tache du noeud situé le plus à gauche de l'ABR supposé non vide"""
    if self.racine…….est_vide():#pas de nœud plus à gauche
        return self.racine……
    else:
        return self.racine.gauche……()
```

7. Une fois la tâche prioritaire effectuée, il est nécessaire de supprimer le nœud correspondant pour que le robot passe à la tâche suivante :

- Si le nœud correspondant à la tâche prioritaire est une feuille, alors il est simplement supprimé de l'arbre (cette feuille devient un arbre vide)

- Si le nœud correspondant à la tâche prioritaire a un sous-arbre droit non vide, alors ce sous-arbre droit remplace le nœud prioritaire qui est alors écrasé, même s'il s'agit de la racine.

Dessiner alors, pour chaque étape, l'arbre binaire de recherche (seuls les indices de priorités seront représentés) obtenu pour un robot, initialement sans tâche, et qui a, successivement dans l'ordre :

- étape 1 : reçu une tâche d'indice de priorité 14 à accomplir

- étape 2 : reçu une tâche d'indice de priorité 11 à accomplir

- étape 3 : reçu une tâche d'indice de priorité 8 à accomplir

- étape 4 : accompli sa tâche prioritaire

- étape 5 : reçu une tâche d'indice de priorité 12 à accomplir

- étape 6 : accompli sa tâche prioritaire

- étape 7 : accompli sa tâche prioritaire

- étape 8 : reçu une tâche d'indice de priorité 15 à accomplir

- étape 9 : reçu une tâche d'indice de priorité 19 à accomplir

- étape 10 : accompli sa tâche prioritaire

EXERCICE 2 (5 points)

Cet exercice est consacré à l'analyse et à l'écriture de programmes récursifs.

1.

a. Expliquer en quelques mots ce qu'est une fonction récursive.

b. On considère la fonction Python suivante :

Numéro de lignes	Fonction `compte_rebours`
1	`def compte_rebours(n):`
2	`    """ n est un entier positif ou nul """`
3	`    if n >= 0:`
4	`        print(n)`
5	`        compte_rebours(n - 1)`

L'appel *compte_rebours(3)* affiche successivement les nombres 3, 2, 1 et 0.

Expliquer pourquoi le programme s'arrête après l'affichage du nombre 0.

2. En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n. Par convention, la factorielle de 0 est 1. Par exemple :

- la factorielle de 1 est 1
- la factorielle de 2 est 2 x 1 =2
- la factorielle de 3 est 3 x 2 x 1 = 6
- la factorielle de 4 est 4 x 3 x 2 x 1 = 24…

Recopier et compléter sur votre copie le programme donné ci-dessous afin que la fonction récursive fact renvoie la factorielle de l'entier passé en paramètre de cette fonction.

Exemple : *fact(4)* renvoie 24.

Numéro de lignes	Fonction `fact`
1	`def fact(n):`
2	`    """ Renvoie le produit des nombres entiers`
3	`    strictement positifs inférieurs à n """`
4	`    if n == 0:`
5	`        return` ***`à compléter`***
6	`    else:`
7	`        return` ***`à compléter`***

3. La fonction somme_entiers_rec ci-dessous permet de calculer la somme des entiers, de 0 à l'entier naturel n passé en paramètre. Par exemple :

- Pour n = 0, la fonction renvoie la valeur 0.
- Pour n = 1, la fonction renvoie la valeur 0 + 1 = 1. …
- Pour n = 4, la fonction renvoie la valeur 0 + 1 + 2 + 3 + 4 = 10.

Numéro de lignes	Fonction `somme_entiers_rec`
1	`def somme_entiers_rec(n):`
2	`    """ Permet de calculer la somme des entiers,`
3	`    de 0 à l'entier naturel n """`
4	`    if n == 0:`
5	`        return 0`
6	`    else:`
7	`        print(n) #pour vérification`
8	`        return n + somme_entiers_rec(n - 1)`

L'instruction *print(n)* de la ligne 7 dans le code précédent a été insérée afin de mettre en évidence le mécanisme en œuvre au niveau des appels récursifs.

a. Écrire ce qui sera affiché dans la console après l'exécution de la ligne suivante :
res = somme_entiers_rec(3)

b. Quelle valeur sera alors affectée à la variable *res* ?

4. Écrire en Python une fonction *somme_entiers* non récursive : cette fonction devra prendre en argument un entier naturel n et renvoyer la somme des entiers de 0 à n compris. Elle devra donc renvoyer le même résultat que la fonction *somme_entiers_rec* définie à la question 3.

Exemple : *somme_entiers(4)* renvoie 10.

EXERCICE 3 (5 points)

Cet exercice porte sur la programmation en Python, la manipulation des chaînes de caractères, les arbres binaires de recherche et le parcours de liste.

1. On rappelle ici quelques notions sur la manipulation des chaînes de caractères en Python. Une chaîne de caractères se comporte comme un tableau de caractères que l'on ne peut pas modifier. Par exemple, on a le comportement suivant :

```
>>> une_chaine = 'Bonjour'
>>> une_chaine[3]
'j'
>>> une_chaine[3] = 'z'
TypeError: 'str' object does not support item assignment
```

On peut aussi utiliser l'opérateur de concaténation +.

```
>>> une_chaine = 'a' + 'b'
>>> une_chaine
'ab'
>>> une_chaine = une_chaine + 'c'
>>> une_chaine
'abc'
```

On définit la fonction *bonjour* par le code suivant :

```
1| def bonjour(nom)
2|         return 'Bonjour ' + nom + ' !'
```

a. Donner le résultat de l'exécution de bonjour('Alan').

On exécute le programme suivant :

```
une_chaine='Bonjour'
x = (une_chaine[2] == une_chaine[3])
y = (une_chaine[4] == une_chaine[1])
```

b. Donner le type et les valeurs des variables x et y.

c. Écrire une fonction occurrences_lettre(une_chaine,une_lettre) prenant en paramètres une chaîne une_chaine et une lettre une_lettre et renvoyant le nombre d'occurrences de une_lettre dans une_chaine

2. On rappelle qu'un arbre binaire de recherche est un arbre binaire pour lequel chaque nœud possède une étiquette dont la valeur est supérieure ou égale à toutes les étiquettes des nœuds de son fils gauche et strictement inférieure à celles des nœuds de son fils droit.

Sa taille est son nombre de nœuds ; sa hauteur est le nombre de niveaux qu'il contient.

On rappelle aussi que l'on peut comparer des chaînes de caractères en utilisant l'ordre alphabétique. On a par exemple :

```
>>> 'ab' < 'aa'
False
>>> 'abc' < 'acb'
True
```

On considère la liste de *mots animaux = ['python', 'chameau', 'pingouin', 'renard', 'gnou']*

a. Dessiner un arbre binaire de recherche contenant tous les mots de la liste animaux et de hauteur minimale.

b. Dessiner un arbre binaire de recherche contenant tous ces mots et de hauteur maximale.

3. On considère l'implémentation objet suivante d'un arbre binaire de recherche : On dispose d'une classe Abr contenant notamment les méthodes et attributs suivants :

- Si un_abr est une instance d'Abr alors un_abr.est_vide() renvoie True si l'arbre est vide et False sinon.
- Si un_abr est une instance d'Abr et si un_abr.est_vide() renvoie False, alors un_abr.valeur contient une chaîne de caractères représentant la valeur de la racine de l'arbre.
- Si un_abr est une instance d'Abr et si un_abr.est_vide() renvoie False, alors un_abr.sous_arbre_gauche et un_abr.sous_arbre_droit contiennent chacun une instance d'Abr.

On considère que la variable liste_mots_francais est une liste de 336531 mots en français et que la variable abr_mots_francais est une instance d'Abr contenant les mots de la liste.

On considère la fonction suivante :

```
def mystere(un_abr):
    if un_abr.est_vide():
        return 0
    else:
    return 1 + mystere(un_abr.sous_arbre_gauche) \
        + mystere(un_abr.sous_arbre_droit)
```

Remarque : on rappelle que le caractère \ en fin de ligne 5 indique que l'instruction se poursuit sur la ligne suivante.

a. Donner le résultat de mystere(abr_mots_francais), en justifiant le résultat. On veut calculer la hauteur de abr_mots_francais.

b. Donner le code d'une fonction hauteur(un_abr) permettant de faire ce calcul, en vous inspirant du code précédent.

4. Dans cette question, nous nous servirons uniquement de *liste_mots_francais* et plus de *abr_mots_francais*.

Pour aider à la résolution de mots croisés, on a décidé d'écrire une fonction *chercher_mots(liste_mots, longueur, lettre, position)* où *liste_mots* est une liste de mots français, *longueur* est la taille du mot recherché, *lettre* est une lettre du mot se trouvant à l'indice *position*.

Par exemple *chercher_mots(liste_mots_francais,3,'x',2)* renverra *['aux', 'box', 'dix', 'eux', 'fax', 'fox', 'lux', 'max', 'six']*.

a. Recopier et compléter la ligne 4 de la fonction ci-dessous :

```
def chercher_mots(liste_mots,longueur,lettre,position):
    res = []
    for i in range(len(liste_mots)):
        if …… and …… :
            res.append(liste_mots[i])
    return res
```

b. Expliquer ce que donne la commande suivante.

```
>>> chercher_mots(chercher_mots(liste_mots_francais,3,'x',2),3,'a',1)
```

On cherche un mot de 5 lettres dont on connait la fin *'ter'*.

c. Ecrire la commande permettant de chercher les mots candidats dans *liste_mots_francais*.