Devoir surveillé - Terminale spécialité NSI

Durée : 2h

Calculatrice non autorisée – Aucun document autorisé

EXERCICE 1

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 sous-arbre 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.

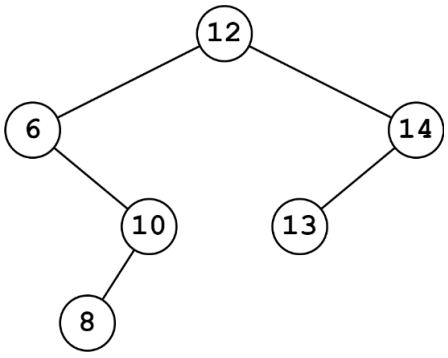

Figure 1 : Exemple d'un arbre binaire

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 :

```
1   class Noeud:
2           def __init__(self, tache, indice):
3                   self.tache = tache   #ce que doit accomplir le robot
4                   self.indice = indice #indice de priorité (int)
5                   self.gauche = ABR()   #sous-arbre gauche vide (ABR)
6                   self.droite = ABR()   #sous-arbre droit vide (ABR)
7
8       def get_tache(self):
9           return self.tache
10
11      def get_indice(self):
12          return self.indice
13
14      def get_gauche(self):
15          return self.gauche
16
17      def get_droite(self):
18          return self.droite
19
20
21  class ABR:
22          #arbre binaire de recherche initialement vide
23          def __init__(self):
24                  self.racine = None #arbre vide
25                  #Remarque : si l'arbre n'est pas vide, racine est #une instance de la classe Noeud
26
27          def est_vide(self):
28                  """renvoie True si l'arbre autoréférencé est vide, False sinon"""
29                  return self.racine == None
30
31          def insere(self, nouveau_noeud):
32                  """insere un nouveau noeud, instance de la classe Noeud, dans l'ABR"""
33                  if self.est_vide():
34                          self.racine = nouveau_noeud
35                  elif self.racine.get_indice() ..... nouveau_noeud.get_indice():
36                          self.racine.get_gauche().insere(nouveau_noeud)
37                  else:
38                          self.racine.get_droite().insere(nouveau_noeud)
39
```

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 36** de la méthode `insere` pour que l'arbre binaire de recherche réponde bien à la définition de l'encadré « **Rappel** » de l'exercice.

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 :

```
41  def est_present(self, indice_recherche) :
42          """renvoie True si l'indice de priorité indice_recherche (int) passé en paramètre est
43          déjà l'indice d'un nœud
44          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:

```
61  def tache_prioritaire(self):
62          """renvoie la tâche du noeud situé le plus à gauche de l'ABR
63          supposé non vide"""
64          if self.racine.....est_vide():#pas de nœud plus à gauche
65                  return self.racine.....
66          else:
67                  return self.racine.get_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

Cet exercice porte sur les structures de données (files et la programmation objet en langage python)

Un supermarché met en place un système de passage automatique en caisse.
Un client scanne les articles à l'aide d'un scanner de code-barres au fur et à mesure qu'il les ajoute dans son panier.
Les articles s'enregistrent alors dans une structure de données.

La structure de données utilisée est une file définie par la classe `Panier`, avec les primitives habituelles sur la structure de file.
Pour faciliter la lecture, le code de la classe `Panier` n'est pas écrit.

```
class Panier():
  def __init__(self):
     """Initialise la file comme une file vide."""

  def est_vide(self):
     """Renvoie True si la file est vide, False sinon."""

  def enfiler(self, e):
     """Ajoute l'élément e en dernière position de la file,
    ne renvoie rien."""

  def defiler(self):
     """Retire le premier élément de la file et le renvoie."""
```

Le panier d'un client sera représenté par une file contenant les articles scannés.
Les articles sont représentés par des tuples `(code_barre, designation, prix, horaire_scan)` où

- `code_barre` est un nombre entier identifiant l'article ;
- `designation` est une chaine de caractères qui pourra être affichée sur le ticket de caisse ;
- `prix` est un nombre décimal donnant le prix d'une unité de cet article ;
- `horaire_scan` est un nombre entier de secondes permettant de connaitre l'heure où l'article a été scanné.

1. On souhaite ajouter un article dont le tuple est le suivant `(31002, "café noir", 1.50, 50525).`
   Ecrire le code utilisant une des quatre méthodes ci-dessus permettant d'ajouter l'article à l'objet de classe `Panier` appelé `panier1`.

2. On souhaite définir une **méthode** `remplir(panier_temp)` dans la classe `Panier` permettant de remplir la file avec tout le contenu d'un autre panier `panier_temp` qui est un objet de type `Panier`.

Recopier et compléter le code de la méthode `remplir` en remplaçant chaque …… par la primitive de file qui convient.

```
def remplir (self, panier_temp) :
        while not  panier_temps. ……..……. :
             article = panier_temp. ……..……..
             self. ……..……..(article)
```

3. Pour que le client puisse connaitre à tout moment le montant de son panier, on souhaite ajouter une **méthode** `prix_total()` à la classe `Panier` qui renvoie la somme des prix de tous les articles présents dans le panier.
Ecrire le code de la méthode `prix_total`. **Attention, après l'appel de cette méthode, le panier devra toujours contenir ses articles**.

4. Le magasin souhaite connaitre pour chaque client la durée des achats. Cette durée sera obtenue en faisant la différence entre le champ `horaire_scan` du dernier article scanné et le champ `horaire_scan` du premier article scanné dans le panier du client. Un panier vide renverra une durée égale à zéro. On pourra accepter que le panier soit vide après l'appel de cette méthode.
Ecrire une **méthode** `duree_courses` de la classe `Panier` qui renvoie cette durée.

EXERCICE 3

Cet exercice porte sur le schéma relationnel de bases de données et les requêtes SQL.

Pour élaborer un jeu vidéo sur le thème des chevaliers de la table ronde, on crée une base de données **Chevalier**. Celle-ci permet de disposer de toutes les caractéristiques de tous les personnages du jeu. Deux relations sont créées avec différents attributs.

Personnage

Idperso	nom	frere_de	points	Idqualite
1	Merlin	NC	40	2
2	Galehaut	NC	40	5
3	Lancelot	Hector	50	1
4	Gareth	Gaheris	20	4
5	Galahad	NC	25	3
6	Gaheris	Gareth	30	1
7	Erek	NC	25	1
8	Lamorak	Perceval	30	1
9	Perceval	Lamorak	35	3
10	Hector	Lancelot	50	6
11	Keud	Arthur	80	7
12	Bedivere	Lucan	20	6
13	Lucan	Bedivere	25	7

Qualité

Idqualite	nom_qualite	vertu
1	apprenti chevalier	tempérance
2	magicien	piété
3	preux chevalier	courage
4	espion	justice
5	seigneur	piété
6	sage	sagesse
7	grand chevalier	justice

NC signifie : « Non communiqué »
Voici le schéma relationnel de cette base de données :

Personnage(Idperso : INT, nom : VARCHAR, frere_de : INT, points : INT, #Idqualite : INT)
Qualite(Idqualite : INT,nom_qualite : VARCHAR,vertu : VARCHAR)

l'attribut #Idqualite de la relation Personnage est une clée étrangère qui fait référence à l'attribut Idqualite de la relation Qualité

Dans la suite, toutes les requêtes demandées seront écrites en langage SQL.

1. Combien y a t-il d'attributs dans la relation **Personnage** ?
2. Pourquoi l'attribut Idqualite de la relation **Personnage** ne peut pas servir de clef primaire.
3. Écrire une requête permettant d'afficher le nom et les points de tous les personnages.
4. Écrire une requête permettant d'afficher le nombre de vertus différentes.
5. Les requêtes ci-dessous sont toutes refusées. Pour chacune d'elle, expliquer ce refus en donnant le nom de la contrainte d'intégrité qui n'a pas été respectée :

a.

```
INSERT INTO Personnage VALUES (13, "Pirlouit", "Johan", 1 , 1 );
```

b.

```
INSERT INTO Personnage VALUES (14, "Pirlouit", "Johan", 40,8);
```

c.

```
INSERT INTO Personnage VALUES (15,"Pirlouit","Johan", "40 points",1);
```

d.

```
DELETE FROM Qualité
WHERE Idqualite=1;
```

e.

```
UPDATE Personnage SET Idqualite=8
WHERE nom="Merlin";
```

f.

```
UPDATE Qualité SET Idqualite=1
WHERE vertu="courage";
```

g.

```
UPDATE Qualité SET nom_qualite=False
WHERE vertu="courage";
```

6. Une erreur a été identifiée dans la relation **Personnage**. Pour l'enregistrement qui a un Idperso égal à 11 : l'attribut "frere_de" n'est pas Arthur mais Antor. Écrire une requête permettant de faire cette modification.

7. On applique la requête suivante :

```
UPDATE Personnage SET points=points+10
WHERE points < 40;
```

Indiquer alors le nombre de points des personnages suivants : Lancelot, Perceval et Merlin.

8. Écrire une requête permettant d'afficher le nom et la vertu des personnages dont les points sont égaux à 40.

9. Écrire dans le bon ordre **les** requête**s** permettant de créer un quatorzième personnage en sa qualité de "*reine*" : la courageuse « Lady Lovelace » avec 100 points.

10. On souhaite supprimer la qualité de "*preux chevalier*". Expliciter une démarche permettant de réaliser cette suppression. (On ne demande pas forcément d'écrire les requêtes en SQL)