BACCALAURÉAT GÉNÉRAL

ÉPREUVE D'ENSEIGNEMENT DE SPÉCIALITÉ

SESSION 2024

NUMÉRIQUE ET SCIENCES INFORMATIQUES

Jeudi 18 janvier (8h – 11h30)

Durée de l'épreuve : **3 heures 30**

L'usage de la calculatrice et du dictionnaire n'est pas autorisé.

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

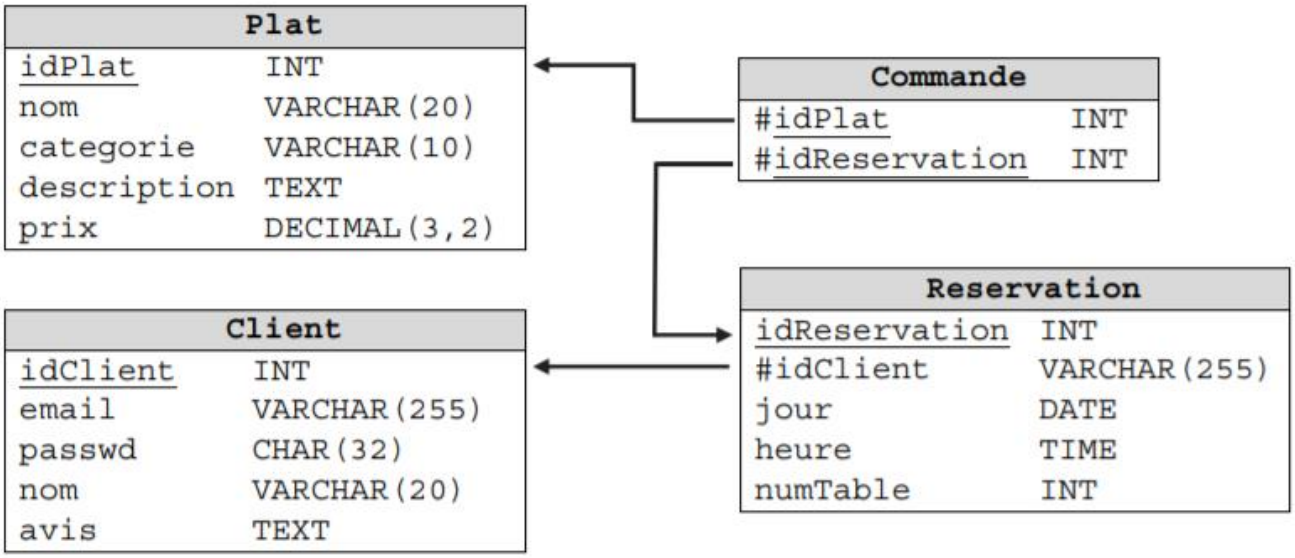
Ce sujet comporte 8 pages numérotées de 1/8 à 8/8.

Les 3 exercices proposés sont indépendants.

Exercice 1 (6 points)

Cet exercice porte sur les bases de données relationnelles.

Une restauratrice a mis en place un site Web pour gérer ses réservations en ligne. Chaque client peut s'inscrire en saisissant ses identifiants. Une fois connecté, il peut effectuer une réservation en renseignant le jour et l'heure. Il peut également commander son menu en ligne et écrire un avis sur le restaurant.
Le gestionnaire du site Web a créé une base de données associée au site nommée `restaurant`, contenant les quatre relations du schéma relationnel ci-dessous :



Dans le schéma relationnel précédent, un attribut souligné indique qu'il s'agit d'une clé primaire. Un attibut précédé du symbole # indique qu'il s'agit d'une clé étrangère et la flèche associée indique l'attribut référencé. Ainsi, par exemple, l'attribut `idPlat` de la relation `Commande` est une clé étrangère qui fait référence à l'attribut `idPlat` de la relation `Plat`.

Dans la suite, les mots clés suivants du langage SQL pourront être utilisés dans les requêtes :

`SELECT, FROM, WHERE, JOIN, ON, DELETE, UPDATE, SET, INSERT INTO, AND, OR.`

1. Parmi les trois requêtes suivantes, écrites dans le langage SQL, laquelle renvoie les valeurs de tous les attributs des plats de la catégorie `'entrée'` :

R1 :
```
SELECT nom, prix
FROM Plat
WHERE categorie  = 'entrée';
```

R2 :
```
SELECT *
FROM Plat
WHERE categorie  = 'entrée';
```

R3 :
```
UPDATE Plat
SET categorie  = 'entrée'
WHERE 1;
```

2. Écrire, dans le langage SQL, des requêtes d'interrogation sur la base de données `restaurant` permettant de réaliser les tâches suivantes :

   a. Afficher les noms et les avis des clients ayant effectué une réservation pour la date du `'2021-06-05'` à l'heure `'19:30:00'`.
   b. Afficher le nom des plats des catégories `'plat principal'` et `'dessert'`, correspondant aux commandes de la date `'2021-04-12'`.

3. Que réalise la requête SQL suivante ?

```
INSERT INTO Plat
VALUES(58,'Pêche Melba', 'dessert', 'Pêches et glace vanille', 6.5);
```

4. Écrire des requêtes SQL permettant de réaliser les tâches suivantes :

   a. Supprimer les commandes ayant comme `idReservation` la valeur `2047`.

   b. Augmenter de 5% tous les prix de la relation `plat` strictement inférieurs à `20.00`.

Exercice 2 (6 points)

Cet exercice porte sur les arbres binaires de recherche.

Dans cet exercice, les arbres binaires de recherche ne peuvent pas comporter plusieurs fois la même clé. De plus, un arbre binaire de recherche limité à un nœud a une hauteur de 1.
On considère l'arbre binaire de recherche représenté ci-dessous (figure 1), où **val** représente un entier :

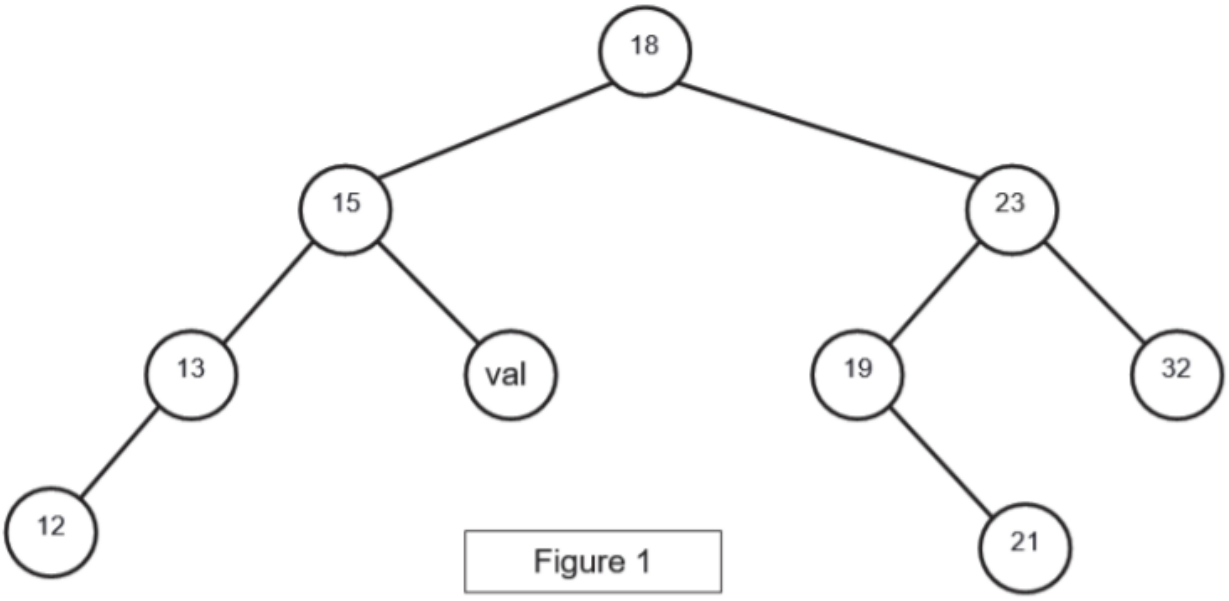


Figure 1

1.

- **a.** Donner le nombre de feuilles de cet arbre et préciser leur valeur (étiquette).
- **b.** Donner le sous arbre-gauche du nœud 23.
- **c.** Donner la hauteur et la taille de l'arbre.
- **d.** Donner les valeurs entières possibles de val pour cet arbre binaire de recherche.

On suppose, pour la suite de cet exercice, que **val** est égal à 16.

2. On rappelle qu'un parcours infixe depuis un nœud consiste, dans l'ordre, à faire un parcours infixe sur le sous arbre-gauche, afficher le nœud puis faire un parcours infixe sur le sous-arbre droit.
Dans le cas d'un parcours suffixe, on fait un parcours suffixe sur le sous-arbre gauche puis un parcours suffixe sur le sous-arbre droit, avant d'afficher le nœud.

- **a.** Donner les valeurs d'affichage des nœuds dans le cas du parcours infixe de l'arbre.
- **b.** Donner les valeurs d'affichage des nœuds dans le cas du parcours suffixe de l'arbre.

3. On considère la classe `Noeud` définie de la façon suivante en Python :

```
class Noeud():
    def __init__(self, v):
        self.ag = None
        self.ad = None
        self.v = v

    def insere(self, v):
        n = self
        est_insere = False
        while not est_insere :
            if v == n.v:
                est_insere = True                    } Bloc 1
            elif v < n.v:
                if n.ag != None:                     }
                    n = n.ag                         }
                else:                                } Bloc 2
                    n.ag = Noeud(v)                  }
                    est_insere = True                }
            else:
                if n.ad != None:                     }
                    n = n.ad                         }
                else:                                } Bloc 3
                    n.ad = Noeud(v)                  }
                    est_insere = True                }

    def insere_tout(self, vals):
        for v in vals:
            self.insere(v)
```

a. Représenter l'arbre construit suite à l'exécution de l'instruction suivante :

```
racine = Noeud(18)
racine.insere_tout([12, 13, 15, 16, 19, 21, 32, 23])
```

b. Ecrire les deux instructions permettant de construire l'arbre de la figure 1. On rappelle que le nombre **val** est égal à 16.

c. On considère l'arbre tel qu'il est présenté sur la figure 1. Déterminer l'ordre d'exécution des blocs (repérés de 1 à 3) suite à l'application de la méthode `insere(19)` au nœud racine de cet arbre.

4. Ecrire une méthode `recherche(self, v)` qui prend en argument un entier `v` et renvoie la valeur `True` si cet entier est une étiquette de l'arbre, `False` sinon.

Exercice 3 (8 points)

Cet exercice porte sur l'algorithmique et la programmation en Python. Il aborde les notions de tableaux de tableaux et d'algorithmes de parcours de tableaux.

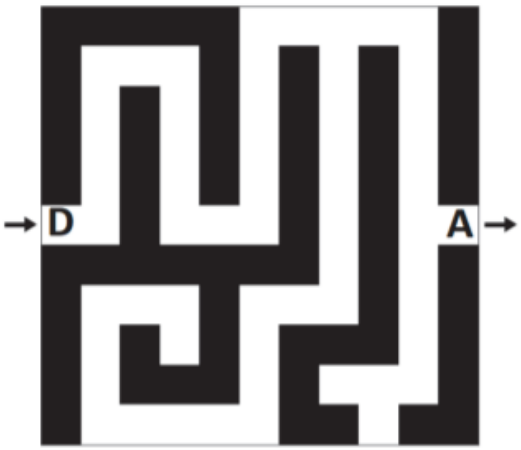
Partie A : Représentation d'un labyrinthe

On modélise un labyrinthe par un tableau à deux dimensions à n lignes et m colonnes avec n et m des entiers strictement positifs.
Les lignes sont numérotées de 0 à $n-1$ et les colonnes de 0 à $m-1$.
La case en haut à gauche est repérée par $(0,0)$ et la case en bas à droite par $(n-1, m-1)$.

Dans ce tableau :

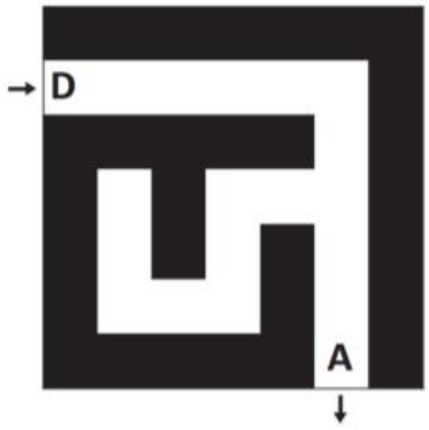
- 0 représente une case vide, hors case de départ et arrivée,
- 1 représente un mur,
- 2 représente le départ du labyrinthe,
- 3 représente l'arrivée du labyrinthe.

Ainsi, en Python, le labyrinthe ci-dessous est représentée par le tableau de tableaux `lab1`.




```
lab1 = [[1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1],
        [1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1],
        [2, 0, 1, 0, 0, 0, 1, 0, 1, 0, 3],
        [1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1],
        [1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 1]]
```

1. Le labyrinthe ci-dessous est censé être représenté par le tableau de tableaux `lab2`. Cependant, dans ce tableau, un mur se trouve à la place du départ du labyrinthe. Donner une instruction permettant de placer le départ au bon endroit dans `lab2`.




```
lab2 = [[1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 0, 1],
        [1, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 0, 1, 0, 1],
        [1, 0, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 3, 1]]
```

2. Écrire une fonction `est_valide(i, j, n, m)` qui renvoie `True` si le couple (i, j) correspond à des coordonnées valides pour un labyrinthe de taille (n, m), et `False` sinon. On donne ci-dessous des exemples d'appels.

```
>>> est_valide(5, 2, 10, 10)
True
>>> est_valide(-3, 4, 10, 10)
False
```

3. On suppose que le départ d'un labyrinthe est toujours indiqué, mais on ne fait aucune supposition sur son emplacement. Compléter la fonction `depart(lab)` ci-dessous de sorte qu'elle renvoie, sous la forme d'un tuple, les coordonnées du départ d'un labyrinthe (représenté par le paramètre `lab`). Par exemple, l'appel `depart(lab1)` doit renvoyer le tuple `(5, 0)`.

```
def depart(lab) :
    n = len(lab)
    m = len(lab[0])
    ...
```

4. Écrire une fonction `nb_cases_vides(lab)` qui renvoie le nombre de cases vides d'un labyrinthe (comprenant donc l'arrivée et le départ).
Par exemple, l'appel `nb_cases_vides(lab2)` doit renvoyer la valeur `19`.

Partie B : Recherche d'une solution dans un labyrinthe

On suppose dans cette partie que les labyrinthes possèdent un unique chemin allant du départ à l'arrivée sans repasser par la même case. Dans la suite, c'est ce chemin que l'on appellera solution du labyrinthe.

Pour déterminer la solution d'un labyrinthe, on parcourt les cases vides de proche en proche. Lors d'un tel parcours, afin d'éviter de tourner en rond, on choisit de marquer les cases visitées. Pour cela, on remplace la valeur d'une case visitée dans le tableau représentant le labyrinthe par la valeur 4.

1. On dit que deux cases d'un labyrinthe sont voisines si elles ont un côté commun.
On considère une fonction `voisines(i, j, lab)` qui prend en arguments deux entiers i et j représentant les coordonnées d'une case et un tableau `lab` qui représente un labyrinthe. Cette fonction renvoie la liste des coordonnées des cases voisines de la case de coordonnées (i, j) qui sont valides, non visitées et qui ne sont pas des murs. L'ordre des éléments de cette liste n'importe pas.

   Ainsi, l'appel `voisines(1, 1, [[1, 1, 1], [4, 0, 0], [1, 0, 1]])` renvoie la liste `[(2, 1), (1, 2)]`.

   Que renvoie l'appel `voisines(1, 2, [[1, 1, 4], [0, 0, 0], [1, 1, 0]])` ?

2. On souhaite stocker la solution dans une liste `chemin`. Cette liste contiendra les coordonnées des cases de la solution, dans l'ordre. Pour cela, on procède de la façon suivante.

- Initialement :
  - déterminer les coordonnées du départ : c'est la première case à visiter ;
  - ajouter les coordonnées de la case départ à la liste `chemin`.
- Tant que l'arrivée n'a pas été atteinte :
  - on marque la case visitée avec la valeur 4 ;
  - si la case visitée possède une case voisine libre, la première case de la liste renvoyée par la fonction `voisines` devient la prochaine case à visiter et on ajoute à la liste `chemin` ;
  - sinon, il s'agit d'une impasse. On supprime alors la dernière case dans la liste `chemin`. La prochaine case à visiter est celle qui est désormais en dernière position de la liste `chemin`.

a. Le tableau de tableaux `lab3` ci-dessous représente un labyrinthe.

```
lab3 = [[1, 1, 1, 1, 1, 1],
        [2, 0, 0, 0, 0, 3],
        [1, 0, 1, 0, 1, 1],
        [1, 1, 1, 0, 0, 1]]
```

La suite d'instructions ci-dessous simule le début des modifications subies par la liste `chemin` lorsque l'on applique la méthode présentée.

```
# entrée: (1, 0), sortie (1, 5)
chemin = [(1, 0)]
chemin.append((1, 1))
chemin.append((2, 1))
chemin.pop()
chemin.append((1, 2))
chemin.append((1, 3))
chemin.append((2, 3))
```

Compléter cette suite d'instructions jusqu'à ce que la liste `chemin` représente la solution. *Rappel : la méthode* `pop` *supprime le dernier élément d'une liste et renvoie cet élément.*

b. Recopier et compléter la fonction `solution(lab)` donnée ci-dessous de sorte qu'elle renvoie le chemin solution du labyrinthe représenté par le paramètre `lab`.
On pourra pour cela utiliser la fonction `voisines`.

```
def solution(lab):
    chemin = [depart(lab)]
    case = chemin[0]
    i = case[0]
    j = case[1]
    ...
```

Par exemple, l 'appel `solution(lab2)` doit renvoyer `[(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]`.