Numérique et Sciences Informatiques

Épreuve type BAC

20 février 2023

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 9 pages numérotées de 1/9 à 9/9.

Nom : _____

Classe : _____

Exercice 1 : Base de données 8 points

Dans cet exercice, on pourra utiliser les mots clés suivants du langage SQL : SELECT, FROM, WHERE, JOIN, ON, INSERT INTO, UPDATE, SET, VALUES, COUNT, OR, AND, LIKE.

Les articles d'un journal sont recensés dans une base de données Journal. Cette base contient des informations sur chaque article (titre, auteur et date de parution). Chaque auteur est caractérisé par son nom et son prénom. On associe de plus à certains articles des thèmes.

Le schéma relationnel de la base de données est donné ci-dessous :

- Articles (idArticle, titre, #auteur, dateParution)
- Auteurs (idAuteur , nom, prenom)
- Themes (idTheme , theme)
- Traitements (idTraitement , #article, #theme)

Dans ce schéma, les clés primaires sont soulignées et les clés étrangères sont précédées du symbole #.

L'attribut auteur de la relation Articles est une clé étrangère faisant référence à l'attribut idAuteur de la relation Auteurs.

L'attribut article de la relation Traitements est une clé étrangère faisant référence à l'attribut idArticle de la relation Articles.

L'attribut theme de la relation Traitements est une clé étrangère faisant référence à l'attribut idTheme de la relation Themes.

Tous les attributs dont le nom est préfixé par id sont des nombres entiers ainsi que l'attribut dateParution (les dates sont notées au format aaaaMMjj, par exemple 20220103 pour le 3 janvier 2022). Les autres attributs sont des chaînes de caractères.

Les entrées de la relation Traitements correspondent aux thèmes abordés par des articles. Un article peut aborder 0, 1 ou plusieurs thèmes.

1. Expliquer pourquoi il n'est pas possible d'insérer une entrée dans la relation Articles si les relations Auteurs et Themes sont vides.

On fournit les extraits suivants de chacune des relations.

Articles

idArticle	titre	auteur	dateParution
1	Présidence française de l'UE	2	20220112
2	Les bleues en demies !	3	20220110

Auteurs

idAuteur	nom	prenom
1	Londres	Alfred
2	Jeraus	Jean
3	Zola	Étienne

Themes

idTheme	themes
1	Politique
2	Sport
3	Europe
4	Handball

Traitements

idTraitement	article	theme
1	1	1
2	1	3
3	2	2
4	2	3

2. L'article nommé « *Les bleues en demies !* » portait sur la coupe d'Europe de handball. Écrire une requête SQL permettant d'ajouter le thème `handball` à cet article dans la table (ou relation) `Traitements`.

3. Le journaliste « Jean Jèraus » tient à ce que son nom soit écrit avec l'accent grave. Ce n'est pas le cas pour l'instant dans la base de données. Écrire la requête SQL permettant de mettre à jour les informations afin de satisfaire sa demande.

4. Pour chacun des items suivants, écrire une requête SQL permettant d'extraire les informations demandées.

a. Le titre des articles parus après le 1^{er} janvier 2022 inclus.

b. Le titre des articles écrits par l'auteur Étienne Zola (on sait que son `idAuteur` est le 3).

c. Le nombre d'articles écrits par l'auteur Jacques Pulitzer (présent dans la table `Auteurs` mais on ne connaît pas son `idAuteur`).

d. Les dates de parution des articles traitant du thème « Sport ».

Exercice 2 : Arbres binaires équilibrés 12 points

On s'intéresse dans cet exercice aux arbres binaires **équilibrés**.

On dit qu'un arbre binaire est équilibré si, quel que soit le nœud considéré, les hauteurs des sous-arbres issus de ses fils gauche et droit ne diffèrent au maximum que de 1. Dans la figure 1, l'arbre de gauche est équilibré alors que celui de droite ne l'est pas.

On considère que la hauteur d'un arbre réduit à sa racine vaut 1. La hauteur d'un arbre vide vaut donc 0.

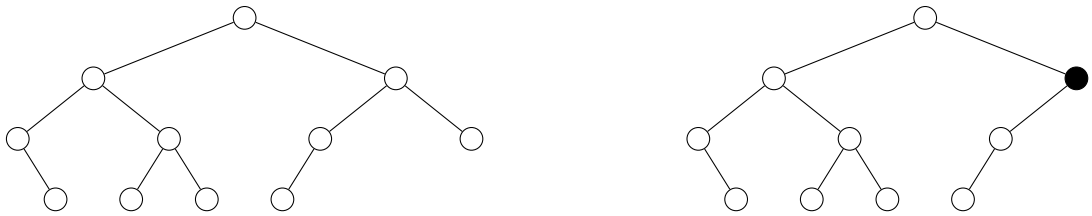


FIGURE 1 – Un arbre binaire équilibré (à gauche) et un non-équilibré (à droite)

Partie A

Afin de déterminer si un arbre est équilibré ou non, on calcule la **balance** de chaque nœud. Cette grandeur est égale à la différence des hauteurs de son fils droit et de son fils gauche. Dans la figure 1, le nœud colorié en noir a une balance de -2. En effet, il n'a pas de fils droit et son fils gauche a une hauteur de 2 : sa balance vaut donc $0 - 2 = -2$.

Un arbre binaire est équilibré si la balance de chacun de ses nœuds vaut -1, 0 ou 1.

1. a. On considère l'arbre de la figure 2. Compléter la figure sur cet énoncé en indiquant à côté de chaque nœud sa balance.

b. Cet arbre est-il équilibré ?

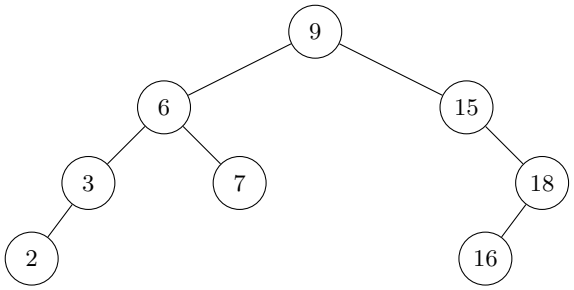
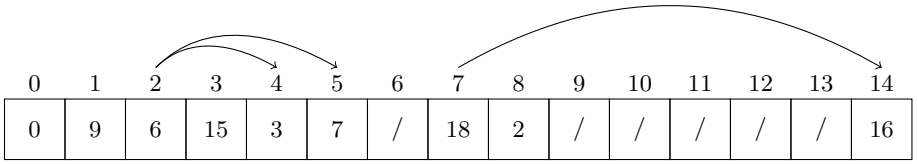


FIGURE 2

On implémente les arbres binaires à l'aide de listes `python`. Dans ce cadre :

- On place la valeur `0` dans la cellule d'indice 0. **Cette valeur sera ignorée dans les différents traitements**
- la valeur de la racine de l'arbre est dans la cellule d'indice 1
- si l'on considère un nœud dont la valeur est stockée dans la cellule n :
  - la valeur de son fils gauche est stockée dans la cellule $2n$
  - la valeur de son fils droit est stockée dans la cellule $2n + 1$
- si un nœud est vide, on stocke une valeur absurde dans le tableau (`None` en `python`)
- le tableau se termine à la dernière valeur non-vide




```
[0, 9, 6, 15, 3, 7, None, 18, 2, None, None, None, None, None, 16]
```

FIGURE 3 – Représentation de l'arbre de la figure 2 dans un tableau et une liste `python`

2. a. Donner la liste `python` représentant l'arbre de la figure 4.

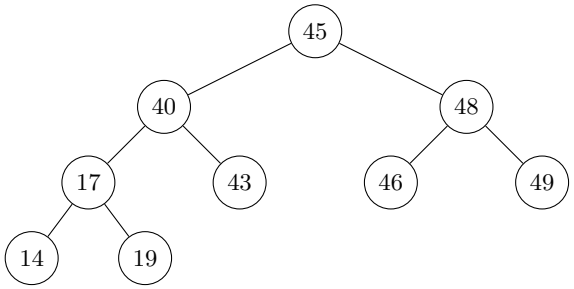


FIGURE 4

b. Dessiner l'arbre représenté par la liste suivante `[0, 35, 30, 12, 20, 23, 10, 5, 2]`.

On fournit la fonction `f` prenant en argument un arbre donné sous la forme d'une liste `python` ainsi que l'indice du nœud racine d'un de ses sous-arbres et renvoyant un nombre entier.

```python
def f(arbre: list, i: int) -> int:
    if i >= len(arbre) or arbre[i] is None:
        return 0
    else:
        g = f(arbre, 2*i)
        d = f(arbre, 2*i + 1)
        return 1 + max(g, d)
```

3. a. On considère la variable `arbre` égale à `[0, 30, 20, 40, 18, 25, None, 47]`. Que renvoie l'appel `f(arbre, 1)` ? Justifier.

b. Que permet de calculer la fonction `f` ?

4. Compléter sur cet énoncé le code de la fonction suivante indiquant si un arbre est équilibré ou non. On rappelle que par définition un arbre vide est équilibré. `arbre` est la liste `python` contenant un arbre et `i` l'indice d'un nœud, racine du sous-arbre étudié.

```python
def estEquilibre(arbre: list, i : int) -> bool:
   if i >= len(arbre) or arbre[i] is None:
        return ......
    else:
        balance = f(arbre, .....) - f(arbre, .....)
        reponse = balance in [..., ..., ...]
        return reponse and estEquilibre(arbre,.....) and .........(.........,.....)
```

Partie B :

On s'intéresse dans cette sous-partie aux arbres binaires de recherche. On rappelle que dans un tel arbre, quel que soit le nœud considéré, son fils gauche, s'il existe, a une valeur qui lui est inférieure ou égale et son fils droit, s'il existe, une valeur qui lui est strictement supérieure. Les différents arbres dessinés plus haut sont tous des arbres binaires de recherche.

1. Donner les parcours *préfixe*, *infixe* et *suffixe* de l'arbre de la figure 4.

Dans le cas où un arbre binaire de recherche n'est pas équilibré, la méthode ci-dessous permet de construire un arbre binaire de recherche équilibré contenant les mêmes valeurs :

(1) Déterminer l'ordre de parcours *infixe* de l'arbre. Le stocker dans une liste `ordre`

(2) Créer un arbre vide nommé `nouveau` (une liste contenant la valeur 0 selon l'implémentation choisie ici). La liste `nouveau` sera réutilisée dans la suite de l'algorithme sans avoir besoin d'être passée en argument de fonctions

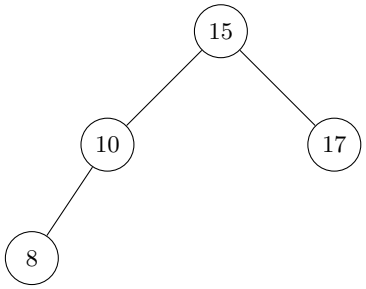
(3) La variable `i` prend la valeur 1 (indice de la racine de `nouveau`)

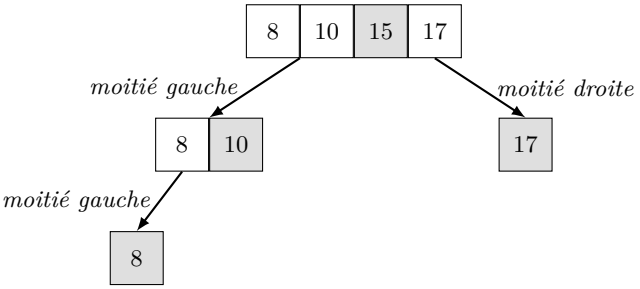
(4) Appeler la fonction récursive `construireABR(i, ordre)` qui :

- (a) Ajoute la valeur `None` à la liste `nouveau` jusqu'à ce qu'elle soit de longueur `i+1`
- (b) Donne la valeur du « milieu » de `ordre` à `nouveau[i]`
- (c) Détermine la `moitié gauche de ordre`
- (d) Si celle-ci est non-vide, appelle `construireABR(2*i, moitié gauche de ordre)`
- (e) Détermine la `moitié droite de ordre`
- (f) Si celle-ci est non-vide, appelle `construireABR(2*i+1, moitié droite de ordre)`

La `moitié gauche de ordre` (resp. `moitié droite de ordre`) est la liste des valeurs de `ordre` d'indice strictement inférieur (resp. supérieur) à celui du milieu. On la crée facilement en faisant `ordre[:len(ordre)//2]` (resp. `ordre[(len(ordre)//2+1):]`)

Cette méthode, appliquée à l'ordre de parcours `[8, 10, 15, 17]` permet de construire l'arbre stocké dans la liste `[0, 15, 10, 17, 8]`.





2. Dessiner l'arbre construit par cette méthode lorsque l'ordre proposé est `[20, 21, 22, 23, 24, 25]`.

On donne ci-dessous le pseudo-code d'une fonction renvoyant l'ordre de parcours infixe d'un arbre fourni sous forme d'une liste `python`. Cette fonction utilise une pile.

```
Fonction infixe(arbre) :
    p est une Pile vide
    visites est une liste vide
    n est égal à 1 # l'indice du nœud étudié dans arbre
    repetition est égal à Vrai
    Tant que repetition est Vrai :
        Tant que n < longueur(arbre) et arbre[n] n'est pas None :
            Ajouter n à la pile
            n est égal à l'indice du fils gauche du nœud d'indice n
        Si la longueur de la pile est nulle :
            repetition est égal à Faux
        Sinon :
            Dépiler une valeur de la pile dans n
            Ajouter arbre[n] à la fin de visites
            n est égal à l'indice du fils droit du nœud d'indice n
    Renvoyer visites
```

3. Transcrire cette fonction en `python`. On pourra utiliser une liste `python` afin d'implémenter la structure de pile.

4. Écrire en `python` le code de la fonction `construireABR` décrite plus haut (point (4) de l'encadré).

Exercice 3 : Bin-Packing 10 points

On considère le problème suivant :

- on dispose de N objets de poids donnés. Les poids sont des nombres entiers strictement positifs
- on souhaite ranger ces objets dans des boîtes ayant toutes la même capacité maximale : chacune d'entre elle peut recevoir un poids maximal de $P_{boîte}$. Le poids maximal des objets est inférieur ou égal au poids maximal des boîtes
- on souhaite ranger ces objets en utilisant un minimum de boîtes

Par exemple, si les poids des objets sont 15, 12, 19, 13, 18 et que le poids maximal des boîtes vaut 30, il est possible de n'utiliser que 3 boîtes : [15, 13], [19] et [18, 12].

1. Proposer une répartition optimale dans le cas suivant :

- les poids des objets sont 15, 5, 4, 17, 26, 11, 13

- le poids maximal des boîtes vaut 30

Afin d'implémenter ce problème en `python`, on donne les poids des objets dans une liste nommée `objets`. Répondre au problème revient à construire une liste de listes `python` dans laquelle chaque sous-liste contient les poids des objets contenus dans une boîte. On nomme cette liste `repartition`.

Si l'on reprend l'exemple donné plus haut, on a :

- `objets = [15, 12, 19, 13, 18]`
- `repartition = [[15, 13], [19], [18, 12]]`

On voit bien que l'objet de poids `12` est dans la troisième boîte (`boites[2]` vaut `[18, 12]`).

2. On considère le problème résolu et la variable `repartition` correctement construite. Quel appel `python` permet de connaître le nombre de boîtes utilisées ?

3. Écrire en `python` la fonction `poidsBoite` :

- prenant en argument le contenu d'une boîte sous forme d'une liste nommée `boite` contenant les poids des objets
- renvoyant le poids total de cette boîte

En utilisant l'exemple cité plus haut, l'appel `poidsBoite([15, 13])` renverra `28`.

On peut montrer (mais pas ici) que ce problème est NP-difficile : cela signifie qu'il n'existe pas d'algorithme « efficace » (en temps polynomial) permettant de le résoudre. On se rabat donc sur des algorithmes fournissant des solutions approchées.

Un premier algorithme consiste à prendre les objets dans l'ordre dans lequel ils sont fournis et à placer chacun d'entre eux dans la **première** boîte pouvant les accueillir. Appliquer cet algorithme à l'exemple fourni la répartition suivante : `[[15, 12], [19], [13], [18]]`.

On appelle cet algorithme **méthode de la première position**.

4. On considère le problème suivant :

- les poids des objets sont 8, 3, 9, 2, 1, 7
- le poids maximal des boîtes vaut 10

a. Appliquer la méthode de la première position.

b. Cet algorithme est-il optimal ? Justifier.

5. Compléter sur cet énoncé le code de la fonction suivante implémentant la méthode de la première position. On réutilise la fonction `poidsBoite` définie à la question **3**. `objets` est la liste contenant les poids des objets à placer et `pMaxi` le poids maximal que peut contenir une boîte.

```python
def premierePosition(objets : list, pMaxi : int) -> list:
    # La répartiton
    repartition = []
    # On ajoute une boîte vide
    repartition.append(...)

    for objet in ..... : # parcours des objets
        ajout = False # permet de savoir si l'objet a été ajouté
        for boite in repartition :
            if poidsBoite(.....) + objet <= ..... :
                # l'objet tient dans cette boite
                boite.append(.....) # on l'ajoute
                ajout = True
                break
        if not ajout : # l'objet ne tient dans aucune des premières boîtes...
            repartition.append(.....) # on l'ajoute dans une nouvelle boîte

    return repartition
```

Un second algorithme consiste à prendre les objets dans l'ordre dans lequel ils sont fournis et à les placer dans la **meilleure** boîte pouvant les accueillir. Par « meilleure » on entend la boîte dans laquelle il reste le moins de place possible après l'ajout.

Dans le cas d'un poids maximal de 10 et d'objets pesant 3, 8 et 2 (fournis dans cet ordre), cette méthode renvoie les boîtes [3] et [8, 2].

6. Proposer un exemple prouvant que cet algorithme n'est pas optimal.

Afin de trouver efficacement la meilleure boîte, on trie les boîtes dans l'ordre décroissant des poids qu'elles contiennent après chaque ajout. On utilise ici une méthode identique à celle du tri par insertion et consistant à permuter l'ordre des boîtes jusqu'à ce que la boîte dans laquelle on vient d'ajouter un objet soit à la bonne position.

7. On considère que l'on vient d'insérer l'objet de poids `p` dans la boîte d'indice `i`. Compléter sur cet énoncé la portion de code permettant positionner correctement cette boîte au sein de la liste `repartition`. On utilise une démarche analogue à celle du tri par insertion.

```python
# On "remonte" cette boîte à sa position triée
while i > .. and poidsBoite(.....) > ........(repartition[...]) :
    ..........................................................................
    i = ............
```

On améliore ces deux méthodes toutefois en triant la liste des objets dans l'ordre des poids décroissants.

8. Entre le « *tri par sélection* » et le « *tri fusion* », lequel permet de trier le plus efficacement une grande liste de nombres distribués aléatoirement ?

On considère les objets [1, 2, 1, 3, 3, 10, 10, 1, 1, 6] et un poids maximal des boîtes de 15.

9. a. Indiquer dans le tableau ci-dessous la répartition obtenue en appliquant chaque méthode.

Méthode	Répartition
Première position **sans** tri	
Meilleure position **sans** tri	
Première position **avec** tri	
Meilleure position **avec** tri	

b. Quelle(s) méthode(s) semble(nt) la(les) plus efficace(s) ? Justifier.