Épreuve disciplinaire

Préliminaires

Notes de programmation Python

Il est demandé d'accompagner les programmes Python d'explications de conception rédigées.
Vous disposez, pour répondre aux questions de ce sujet, des fonctions de listes suivantes :

— création d'une liste `li` de taille n dont tous les éléments ont pour valeur v : `li = [v]*n`.
— renvoi de la taille d'une liste `li` : `len(li)`.
— insertion (avec décalage à droite) à un indice i dans une liste `li` : `li.insert(i,v)`.
— ajout de la valeur v en fin de liste : `li.append(v)`.
— position du premier élément de la liste égal à v (en comptant à partir de 0) : `li.index(v)`.
— énumération des éléments d'une liste, dans l'ordre : `for x in li`.
— construction de listes par *compréhension* : `[x**2 for x in li]` est une nouvelle liste construite en mettant au carré chaque élément de la liste `li`.
— les fonctions permettant de sélectionner des tranches (*slices*) : `li[start:end]` sélectionne la sous-liste d'indices de `start` à `end-1`.

Il est autorisé d'utiliser l'évaluation gauche droite paresseuse des expressions booléennes, par exemple, si l'expression `expr_bool1` s'évalue à `False`, alors l'expression `expr_bool1 and f(42)` est directement évaluée à False (sans appel de la fonction `f`).

Enfin, on utilise la possibilité en Python des versions supérieures à 3.10 de décrire les types des paramètres et de retour de fonction, par exemple : `foo(a:int, b:int)->bool` est une fonction nommée `foo`, à deux paramètres entiers, retournant un Booléen.

Notes de complexité

La complexité d'un algorithme est le nombre d'opérations élémentaires réalisées par celui-ci et calculée en fonction des données d'entrées, et selon le contexte (à rappeler), exprimée dans le meilleur cas, le pire cas, ou en moyenne. Cette complexité sera dans tout ce sujet exprimée à l'aide de l'opérateur O, dont on rappelle la définition ici : on dit que $f = O(g)$ s'il existe une constante K et une constante M telle que pour tout $n > M$, $f(n) \leqslant Kg(n)$.

On rappelle que les *listes* Python sont en fait des tableaux redimensionnables. En conséquence, le calcul de la taille, l'accès à un élément et l'ajout en fin de liste sont considérés comme des opérations élémentaires, avec une complexité $O(1)$. L'insertion et le parcours sont des opérations linéaires en la taille de la liste.

Vocabulaire d'arbres

Dans ce sujet :

— les arbres sont des graphes acycliques connectés, enracinés et étiquetés ;
— un nœud sans enfant est dénommé feuille ;
— on appelle la longueur d'un chemin le nombre de nœuds de ce chemin, extrémités inclues ;
— on appelle arité d'un nœud le nombre d'enfants de ce nœud ; par extension on appelle arité d'un arbre le maximum de l'arité de ses nœuds. Un arbre binaire est d'arité 2.
— on appelle distance entre deux nœuds A et B la valeur $l - 1$, où l désigne la longueur du chemin (A, B) ;
— on appelle niveau d'un nœud d'un arbre la distance de ce nœud à la racine de l'arbre (la racine d'un arbre est de niveau 0) ;
— la hauteur d'un arbre est la longueur du plus grand chemin de la racine à une feuille (en nombre de nœuds) ;
— un arbre est équilibré si tous les chemins de la racine aux feuilles ont la même longueur ;

Rappel mathématique

$$\text{Pour } x \neq 1 \ : \ \sum_{i=0}^{i=n-1} x^i = \frac{1-x^n}{1-x}$$

1 Arbres Binaires de Recherche

Un arbre binaire de recherche est un *arbre binaire* (i.e. chaque nœud possède 0, 1 ou 2 enfants), dans lequel les nœuds sont étiquetés par des *entiers naturels* appelés *clefs*. De plus, si l'on considère un nœud de clef k, alors chaque nœud de son sous-arbre gauche (resp. droit) a une clef inférieure ou égale à k (resp. supérieure ou égale). Un exemple de tel arbre est donné à la Figure 1.

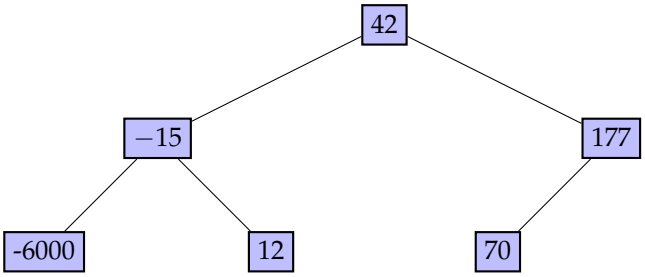


FIGURE 1 – Un arbre binaire de recherche

Nous proposons la définition suivante pour les arbres binaires en Python : tout d'abord, nous définissons un type `Noeud` sous forme d'une classe (un nœud a forcément deux enfants, qui sont soit des nœuds, soit `None`) :

```
class Noeud:
    """
    Noeud d'un arbre
    """
    def __init__(self, clef:int, gauche= None, droite= None):
        """ Initialisation, noeud non vide"""
        self.gauche = gauche
        self.droite = droite
        self.clef = clef
```

Par la suite, un arbre binaire est soit l'arbre vide (`None`) soit enraciné sur un nœud (`Noeud`).
L'arbre de la Figure 1 peut donc être construit de la façon suivante :

```
mon_arbre = Noeud(42, Noeud(-15, Noeud(-6000), Noeud(12)), Noeud(177, Noeud(70), None));
```

À ce stade, rien ne permet de s'assurer que l'arbre construit est un arbre binaire de recherche.

Question #1

Écrire une fonction récursive en Python de recherche d'une clef dans un arbre binaire de recherche (contenant au moins un nœud). Cette fonction aura la signature suivante :
`recherche(root : Noeud, k : int)->bool`

Question #2

Donner la complexité algorithmique de votre fonction en fonction du nombre de nœuds de l'arbre binaire de recherche. On distinguera des cas selon la forme des arbres en entrée.

Question #3

En utilisant un parcours récursif, écrire une fonction `collecte` qui collecte et ajoute dans une liste Python les valeurs des clefs des nœuds d'un arbre binaire de recherche par ordre croissant, et renvoie cette liste. On pourra utiliser la concaténation de listes et l'ajout en fin de liste (avec `append`) sans se préoccuper de la complexité.

Question #4

Nous proposons l'algorithme suivant pour déterminer si un arbre binaire est un arbre binaire de recherche :

```
1      def est_ABR_1(self):
2      """
3      renvoie True ssi l'arbre est un ABR
4      """
5      temp = True
6      val = self.clef
7      if self.gauche is not None:
8          temp = self.gauche.clef < val and self.gauche.est_ABR_1()
9      if self.droite is not None:
10         temp = temp and self.droite.clef > val and self.droite.est_ABR_1()
11      return temp
```

En exhibant un arbre (non feuille) bien choisi, montrer que cet algorithme est incorrect.

Nous désirons maintenant proposer un algorithme correct : celui-ci est basé sur l'invariant structurel suivant : lorsque l'on observe un sous-arbre (d'un arbre binaire de recherche) enraciné en un nœud de clef k :

— Dans le sous-arbre de gauche, le maximum des clefs est inférieur ou égal à k.

— Dans le sous-arbre de droite, le minimum des clefs est supérieur ou égal à k.

Pour simplifier, il est supposé que l'on connaît une borne inférieure (`inf=INF`) et une borne supérieure (`sup=SUP`) des valeurs des clefs de l'arbre. L'algorithme consiste alors à propager un intervalle (initialement égal à `(inf,sup)`) de la racine vers les feuilles à chaque appel récursif du parcours.

Question #5

Recopier sur votre copie l'arbre de la Figure 1 en indiquant à côté de chaque nœud les intervalles propagés par l'algorithme décrit ci-dessus.

Question #6

Sur votre contre-exemple de la question 4, comment cet algorithme va-t-il décider que ce n'est pas un arbre binaire de recherche ?

Question #7

Implémenter la fonction `est_ABR_2_aux(self, inf:int, sup:int)->bool` qui propage les intervalles de haut en bas et décide si l'arbre binaire courant est un arbre binaire de recherche.

2 B-trees

Pour une première définition, nous nous reportons à Wikipédia :

> « En informatique, un arbre B (appelé aussi B-arbre par analogie au terme anglais « B-tree ») est une structure de données en arbre équilibré. Les arbres B sont principalement mis en œuvre dans les mécanismes de gestion de bases de données et de systèmes de fichiers. Ils stockent les données sous une forme triée et permettent une exécution des opérations d'insertion et de suppression en temps toujours logarithmique.
>
> Le principe est de permettre aux nœuds parents de posséder plus de deux nœuds enfants : c'est une généralisation de la notion d'arbre binaire de recherche. Ce principe minimise la taille de l'arbre et réduit le nombre d'opérations d'équilibrage. De plus un B-tree grandit à partir de la racine, contrairement à un arbre binaire de recherche qui croît à partir des feuilles. »

Dans cette partie nous allons tout d'abord implémenter ces B-trees et en caractériser certaines opérations.

2.1 Définition, algorithmes élémentaires

Contrairement à la section précédente, les B-trees ne seront pas binaires, mais d'arité quelconque. De plus, les nœuds ne contiennent pas une clef unique, mais un ensemble de clefs.

Définition
Un B-tree d'ordre t (cf. Figure 2) est un arbre qui satisfait les propriétés suivantes :

- (S1) Tous les chemins de la racine à une feuille ont la même longueur (le même nombre de nœuds), appelée hauteur de l'arbre et dénotée par h.
- (S2) La racine est une feuille ou bien a au moins 2 enfants.
- (S3) Chaque nœud qui n'est ni racine ni feuille possède au moins $t+1$ enfants.
- (S4) Chaque nœud a au plus $2t+1$ enfants.
- (C1) Les nœuds contiennent des clefs : la racine contient de 1 à $2t$ clefs ; et les autres nœuds, entre t et $2t$ clefs.
- (C2) Dans chaque nœud, les éléments sont stockés par ordre croissant.
- (C3) Chaque nœud qui contient ℓ clefs (et qui n'est pas une feuille) a $\ell+1$ enfants.
- (C4) Considérons un nœud P contenant ℓ clefs $x_0 \dots x_{\ell-1}$. Soient $P_0, \dots P_\ell$ ses enfants, et $K(P_i)_{0 \leqslant i \leqslant \ell}$ l'ensemble des clefs du sous-arbre dont la racine est P_i. Alors :
  - $\forall y \in K(P_0), y \leqslant x_0$;
  - $\forall 1 \leqslant i \leqslant \ell - 1, \forall y \in K(P_i),\ x_{i-1} \leqslant y \leqslant x_i$;
  - $\forall y \in K(P_\ell), x_{\ell-1} \leqslant y$;

Les propriétés sont de deux ordres : quatre propriétés **structurelles**, préfixées par « S », et des propriétés de contenu/de clefs, préfixées par « C ». Remarquons que la propriété C4 est une généralisation de la propriété fondamentale des arbres binaires de recherche.

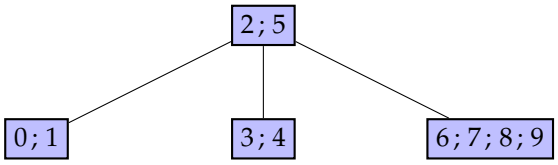
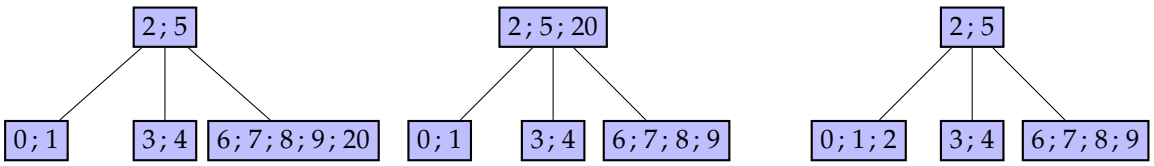


FIGURE 2 – Un exemple de B-tree d'ordre 2. Les nœuds contiennent un ensemble de clefs, triées par ordre croissant.

Question #8
Les arbres de la Figure 3 sont-ils des B-trees d'ordre 2 ?



(a) Arbre 1 (b) Arbre 2 (c) Arbre 3

FIGURE 3 – Quelques arbres.

L'implémentation du type B-tree en Python est fournie ci-dessous :

```
class BTreeNode:
  def __init__(self, t:int, f=False):
    self.feuille = f # True si le noeud est une feuille
    self.clefs = []   # Les valeurs des clefs du noeud
    self.enfants = []  # les enfants du noeud (qui sont des BTreeNode)
    self.ordre = t       # l'ordre du noeud
    self.n = 0       # nombre de clefs du noeud
```

```
class BTree:
  def __init__(self, t):
    self.root = BTreeNode(t,True)
    self.ordre = t  # l'ordre du btree
```

Ainsi l'appel `BTree(2)` renvoie une instance de la classe BTree d'ordre 2 représentant une feuille sans clef ni enfant. Remarquons en particulier que :

— Rien ne garantit les invariants de la définition d'un B-tree. Il faut donc bien faire attention à les maintenir.

— La numérotation des enfants et celle des clefs commencent à 0.

Par commodité, toutes les fonctions écrites seront des méthodes de la classe BTree.

Question #9

Écrire une fonction récursive d'impression d'un B-tree, `def print_node(self, niveau=0)`, qui affiche chacun des nœuds du `BTreeNode` courant, à partir du nœud x . Chaque nœud sera affiché sur une nouvelle ligne, avec le nombre de clefs qu'il contient, ainsi que les valeurs de ces clefs. Ainsi, sur l'exemple de la Figure 2, l'appel `B.root.print_node()` doit donner l'impression suivante :

```
Niveau  0   2:2 5
Niveau  1   2:0 1
Niveau  1   2:3 4
Niveau  1   4:6 7 8 9
```

2.2 Recherche dans un B-tree

La recherche d'une clef dans un B-tree va suivre le même motif que pour un arbre binaire de recherche : en effet, elle va combiner deux algorithmes : la recherche dans une liste triée (`keys`) et la recherche parmi les enfants (`child`) d'un nœud donné.

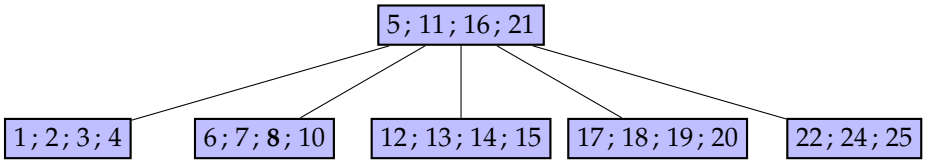


FIGURE 4 – Exemple de B-tree (d'ordre 2) pour la recherche

Question #10

Dans le B-tree de la figure 4, expliquer les étapes de recherche de la clef dont la valeur est 8. Utiliser les caractéristiques du B-tree pour ne pas réaliser un parcours entier de l'arbre.

Question #11

En déduire, pour un nœud fixé, une condition d'arrêt de la recherche dans ce nœud ainsi que le numéro du sous-arbre dans lequel rechercher récursivement.

Question #12

Implémenter en Python `recherche_clef(self : BTreeNode, k : int)-> bool` cherchant la clef k dans l'arbre courant à partir du nœud courant.

Étudions maintenant la complexité algorithmique de la recherche. Considérons des B-trees non vides, d'ordre $t > 1$, de hauteur $h \geqslant 1$. Dans un premier temps, nous allons compter le nombre de nœuds minimum N_{min} et maximum N_{max} d'un tel B-tree. Il est recommandé de procéder en comptant les nœuds par niveau.

Question #13

Montrer que $N_{min} = 1 + \frac{2}{t}\left((t+1)^{h-1} - 1\right)$ pour $h \geqslant 2$.

Question #14

Montrer que $N_{max} = \frac{1}{2t}\left((2t+1)^h - 1\right)$ pour $h \geqslant 1$.

Question #15

En déduire un encadrement du nombre de clefs contenues dans un B-tree d'ordre t et de hauteur h.

Question #16

Expliquer rapidement comment optimiser le coût de la recherche dans un nœud. Quelle est la complexité de cet algorithme ?

Question #17

Conclure sur la complexité algorithmique de la recherche dans un B-tree en fonction du nombre de nœuds n.

2.3 Insertion dans un B-tree

L'algorithme d'insertion d'une clef dans un B-tree réalise des étapes similaires à celui de l'insertion dans un arbre binaire de recherche : l'arbre est parcouru récursivement à partir de la racine, jusqu'à trouver une feuille dans laquelle insérer l'élément. L'insertion se réalise donc **dans une feuille**. Les B-trees ont néanmoins une contrainte supplémentaire : un nombre minimum et un nombre maximum de clefs pour les nœuds. Avant d'insérer une clef dans un nœud, il faudra s'assurer qu'un nœud « n'est pas complet » (au sens où le nombre maximum de clefs par nœud n'est pas atteint).

Question #18

Expliquer comment insérer la clef 23 dans le B-tree de la Figure 4.

Question #19

Implémenter une fonction `insertion_position(self : BTreeNode, k : int) -> int` qui, appelée sur un BTreeNode, calcule l'indice auquel il faudrait insérer la clé k dans la liste `self.clefs` (en respectant le critère C4 et en supposant que la liste `self.clefs` ne soit pas déjà trop pleine).

Question #20

En utilisant la fonction précédente, écrire la fonction `insertion_non_complet(self : BTreeNode, k:int)` qui réalise l'insertion d'une clef k dans un `BTreeNode`, dans le cas où celui-ci n'est pas complet.

Question #21

Donner la condition en Python qui permet de déterminer qu'un nœud x est complet.

Dans le cas où l'endroit d'insertion est complet, le nœud va être « scindé en deux » (*split*), afin de récupérer deux nœuds de tailles plus petites. La Figure 5 explique le processus. On considère un B-tree d'ordre $t = 2$ et un sous-arbre enraciné en x, dont l'un des enfants y, doit être scindé, car il contient trop de clefs (ici 5 clefs). Le nœud y est donc scindé en deux nœuds y et z de taille égale à 2 et les sous-arbres associés sont « distribués » en conséquence. L'élément « en trop », ici 8, est alors **éjecté**, pour être inséré par la suite dans l'ensemble des clefs du nœud parent, x (qui lui même peut être complet, comme nous le verrons par la suite).

Question #22

Écrire la méthode `def split(self : BTreeNode)-> (int, BTreeNode)` d'un `BTreeNode` dont le nombre de clefs est supposé (temporairement) égal à $2t + 1$. La méthode actualise le nœud y et renvoie la valeur « éjectée » (8 dans la Figure 5), ainsi que le nouveau nœud créé (z dans la Figure 5). *Cette procédure, non récursive, ne réalise pas l'insertion dans le nœud parent*. Il est fortement recommandé d'utiliser les *slices* (« tranches ») Python.

La procédure d'insertion récursive peut réaliser de multiples scindages, si la valeur éjectée d'un nœud doit être insérée dans un nœud déjà complet. Le résultat final de la procédure d'insertion est donc possiblement un B-tree très différent du B-tree initial. Ainsi, l'insertion de la clé 9 dans le B-tree de la Figure 4 a pour résultat le B-tree de la Figure 6.

Question #23

Que se passe-t-il lors de l'insertion si la racine est complète ?

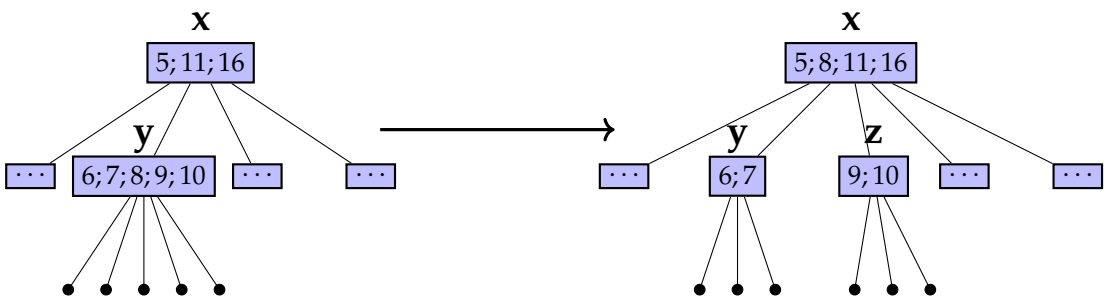


FIGURE 5 – Illustration (partielle) de l'insertion de 8 dans le nœud **y** avec « split » : lors de l'insertion, le nœud **y** possède temporairement 5 clefs, ce qui n'est pas permis pour un B-tree d'ordre 2. La procédure de découpage ("split") va donc créer le nouveau nœud **z** et éjecter la valeur 8. Cette valeur sera retournée au parent à la fin de l'appel récursif d'insertion.

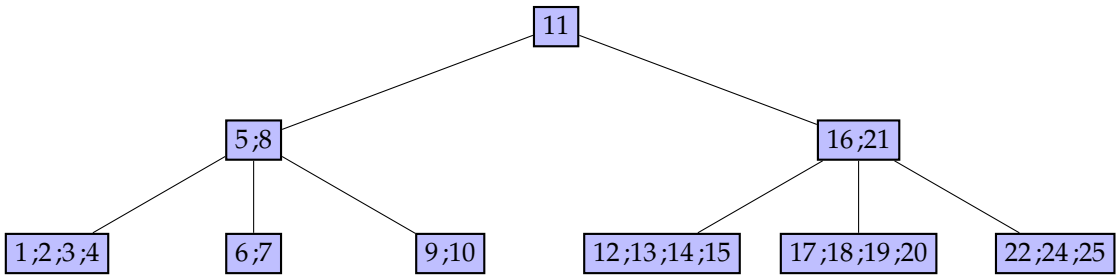


FIGURE 6 – B-tree résultant de l'insertion de la clef 9 dans le B-tree de la Figure 4

La méthode d'insertion au niveau `BTree` est fournie ci-dessous.

```
def insert(self, k): # Version bottom-up de l'insertion dans un Btree
    val, newnode = self.root.insert(k)
    if newnode is not None:
        self.root = self.root.make_new_root(val, newnode)
```

Question #24

Implémenter la méthode `make_new_root` qui traite le cas de la racine complète.

Question #25

Terminer l'implémentation de l'insertion en implémentant la méthode `insert` de la classe `BTreeNode`.

3 Application : Indexation dans les Bases de Données

Dans cette partie, nous allons étudier une application des B-trees, le stockage *persistant* des tables dans un système de gestion de bases de données (SGBD). En effet, les données des tables sont stockées dans des fichiers, pour pouvoir traiter leur potentiel grand volume et les sauvegarder dans le temps. Ces fichiers sont supposés stockés sur un disque dur local.

Une grande partie de la littérature des bases de données est consacrée à l'étude et à la conception de structures de données performantes pour le stockage et le requêtage efficace de ces données. Le but principal est de limiter la quantité de données lues sur le support persistant.

Dans cette partie, nous considérons à titre d'exemple, une base de données de taille moyenne, contenant des informations sur l'oeuvre de Shakespeare (https://www.opensourceshakespeare.org).

3.1 Schéma relationnel et requêtes

La Figure 7 donne un aperçu de deux des relations de la base de données précitée, avec des exemples d'*enregistrements*.

Characters (Personnages)

CharID	CharName	Abbrev	WorkID	SpeechCount
'1citizen'	'First Citizen'	'FC'	'romeojuliet'	3
'1musician-oth'	'First Musician'	'FM'	'othello'	5
'1musician-rj'	'First Musician'	'FM'	'romeojuliet'	10
'2gentleman-oth'	'Second Gentleman'	'SG'	'othello'	5
'2musician-rj'	'Second Musician'	'SM'	'romeojuliet'	3

Works (Œuvres)

WorkID	Title ,	LongTitle	Date
'othello'	'Othello'	'The Tragedy of Othello, Moor of Venice'	1604
'hamlet'	'Hamlet'	'The Tragedy of Hamlet, Prince of Denmark'	1600
'romeojuliet'	'Romeo and Juliet'	'The Tragedy of Romeo and Juliet'	1594

FIGURE 7 – Extrait de la Base de Données Open Shakespeare (simplifiée en deux tables).

Question #26

Dans la relation `Characters`, que signifie le fait que `CharID` est une clef primaire ? `WorkID` est-il une clef étrangère pour la relation `Characters` ?

On souhaite effectuer plusieurs recherches dans la base de données. Les recherches sont décrites comme suit, et sont désignées par une étiquette R1, R2, R3, R4.

- R1 : Rechercher toutes les informations décrites dans la table 'Characters', concernant le personnage désigné par l'identifiant '2musician-rj'.
- R2 : Rechercher le nombre de mots prononcés (attribut `SpeechCount`) par le personnage désigné par l'identifiant '2musician-rj'.
- R3 : Lister tous les titres longs des oeuvres de Shakespeare écrites entre 1589 et 1600.
- R4 : Lister l'ensemble des titres longs des oeuvres où apparaissent des musiciennes ou des musiciens (dont l'attribut `CharName` contient *Musician*).

Dans la suite, les recherches seront désignées par leur étiquette.

Question #27

Exprimer chacune des recherches (R1, R2, R3 et R4) sous forme d'une requête SQL distincte ; chaque requête est indépendante des autres.

3.2 Stockage des enregistrements

Nous considérons par la suite que le type 'char' est stocké sur 8 bits (1 octet), et le type 'mediumint' est stocké sur 3 octets. Chaque enregistrement est stocké comme la suite des informations des attributs de cet enregistrement.

Le stockage des données sur disque dur se fait par *blocs* de 4096 octets. Chaque bloc comprend une entête de 200 octets. Les enregistrements de la relation `Characters` sont stockés dans un fichier, dont la taille est un multiple de 4096 octets (*blocs*).

La relation `Characters` comporte 1264 enregistrements et la table correspondante est créée en SQL par la commande suivante :

```
CREATE TABLE IF NOT EXISTS `Characters` (
  `CharID` varchar(50) NOT NULL DEFAULT '',
  `CharName` varchar(75) NOT NULL DEFAULT '',
  `Abbrev` varchar(10) NOT NULL DEFAULT '',
  `WorkID` varchar(50) NOT NULL DEFAULT '',
  `SpeechCount` mediumint unsigned NOT NULL DEFAULT 0,
  PRIMARY KEY (`CharID`)
) DEFAULT CHARSET=latin1;
```

Question #28

Dans un enregistrement, les champs décrivant des chaînes de caractères sont munis d'un premier octet donnant le nombre effectif de caractères de la chaine, sans sentinelle de fin de chaîne. Donner la taille, en octets, de l'enregistrement `('1musician-rj', 'First Musician', 'FM', 'romeojuliet', 10)`.

Question #29

Donner la taille minimale $SIZE_{min}$ et maximale $SIZE_{max}$ d'un enregistrement de la relation `Characters`.

Question #30

Calculer le nombre maximal d'enregistrements de taille $SIZE_{max}$ par bloc. En déduire la taille maximale (en octets) du fichier stockant la relation `Characters`.

3.3 Recherche dans une table

Nous nous intéressons maintenant à la complexité d'une opération de sélection simple sur la table `Characters`, qui serait par exemple réalisée lors de l'exécution de la commande SQL associée à la recherche `R1`.

Nous supposons que la base de données est implantée sur un disque dur magnétique et que le temps moyen d'accès à un bloc sur disque est 15 ms ; une fois le bloc chargé en mémoire, la recherche à l'intérieur du bloc est réalisée séquentiellement, et le temps d'accès à l'enregistrement au sein d'un bloc est jugé négligeable.

Question #31

En l'absence de structure supplémentaire, l'accès aux blocs du fichier contenant la table `Characters` est séquentiel. Donner une estimation du temps maximal nécessaire à la recherche d'un enregistrement particulier dans la table `Characters`.

En général, l'accès à la base de données est réalisée au travers d'un SGBD (Système de Gestion de Base de Données). Le SGBD réalise les opérations de création / modification / sauvegarde / requêtage sur des éléments de la base de données. En particulier, il maintient un *système d'indexation* qui permet d'améliorer les temps d'accès aux enregistrements des tables de la base.

L'indexation consiste à maintenir, dans une structure dédiée, l'information minimale permettant de localiser chaque enregistrement de façon efficace. Chaque enregistrement est décrit par un couple : (identifiant unique de l'enregistrement, numéro de bloc dans lequel l'enregistrement est créé). Cette structure est maintenue cohérente par le SGBD lors de la création ou de la destruction d'un enregistrement dans une table.

Cette structure d'indexation peut être réalisée au travers de *tables d'index*. Une table d'index associée à une relation stocke les couples (identifiant unique, numéro de bloc) de la table décrivant la relation ; les couples sont rangés par identifiant croissant. On suppose qu'un numéro de bloc est stocké sur 8 octets.

Question #32

On considère la relation `Characters`. Indiquer le champ à utiliser pour représenter l'identifiant unique de l'enregistrement dans la table d'index, et la relation d'ordre adaptée. Déterminer une évaluation de la taille maximale de la table d'index associée à la table `Characters`, (en nombre d'octets et en nombre de blocs).

Question #33

Donner les avantages et inconvénients d'utiliser un index sous forme de table lors des opérations d'insertion, de suppression ou de recherche d'un enregistrement d'une table représentant une relation de la base de données.

Une autre implémentation de la structure d'indexation est basée sur les B-trees vus précédemment, avec la variante suivante :

— Seules les feuilles contiennent des couples (clef/ numéro de bloc) : les clefs seront les identifiants

— Les nœuds qui ne sont pas des feuilles ne contiennent que des clefs (et un lien vers leurs enfants). Ces clefs sont des recopies de certaines clefs apparaissant dans les feuilles ; elles servent de balises lors des parcours du B-tree, afin d'orienter la recherche vers le sous-arbre adéquat.

Question #34

Montrer qu'un B-tree d'ordre 3 et de hauteur 4 suffit à permettre la construction d'un index pour la clé `charID` pour la table `Characters`. En considérant qu'une adresse (de bloc) est stockée sur 8 octets, et en ignorant la place prise par le chaînage parent-enfant, donner une évaluation de la place en mémoire prise par ce B-tree. Expliquer l'intérêt de la structure d'index sous forme de B-tree par rapport à la structure en table.

Question #35

Quelles sont les étapes à effectuer pour répondre à la requête associée à la recherche `R2` ?

On souhaite réaliser un parcours efficace des index et tables lors de la recherche `R3`. Pour ce faire, on cherche à construire un index relatif à la table `Works`, permettant un parcours séquentiel des enregistrements de cette table, selon un index combinant différents attributs de la relation `Works`.

Question #36

Quel couple d'attributs de la relation `Works` faut-il intégrer dans l'index et quelle est la relation d' ordre associée à ces couples pour traiter efficacement la requête `R3` ?

Question #37

Quelle modification simple apporteriez-vous au B-tree correspondant à l'indexation des enregistrements de la relation `'Works'` afin de parcourir séquentiellement les feuilles du B-tree, en respectant l'ordre associé ?

On suppose que des structures d'index en B-tree ont été construites, pour les deux tables `'Characters'` et `'Works'`, en intégrant la modification de la question précédente.

Question #38

Quelles sont les étapes à effectuer pour réaliser la requête `R3` ?

Question #39

Quel index pourrait être proposé pour réaliser efficacement les parcours d'index et de table associés à la requête R4 ?

Question #40

Quels sont les critères à privilégier pour associer un attribut de table à un index ? Pourquoi faut-il éviter de maintenir trop d'index différents pour une table donnée ?

On suppose maintenant que l'on travaille sur une base de données bien plus grande. Elle comprend une table `T` de 4,2 millions d'enregistrements, stockés sur 300 000 blocs.

Question #41

Déterminer dans ce cas, le temps maximal pour la recherche d'un enregistrement particulier dans `T` lorsqu'il n'y a pas de mécanisme d'indexation.

On cherche à construire une structure d'index en B-tree pour optimiser les accès à `T`. Différents arbres sont possibles :

1. Un B-tree d'ordre 3 et de taille 8 et
2. Un B-tree d'ordre 6 et de hauteur 6 permettent de stocker l'index de T.

Question #42

On suppose que chacun des deux arbres tient en mémoire RAM. Parmi ces deux arbres, quelle est la structure la plus adaptée et pourquoi ?