CAPES NSI M2 – session 2026

Épreuve disciplinaire

Capes NSI

Épreuve disciplinaire

Remarques générales sur le sujet

Il est attendu des candidates et candidats une rédaction précise et soignée. Toute tentative pertinente sera valorisée lors de la correction.

Ce sujet est composé de 2 parties entièrement indépendantes. Chaque partie est décomposée en 4 sous-parties relativement indépendantes, qui peuvent être traitées dans l'ordre souhaité. Il est cependant fortement conseillé de réaliser la partie 1.2 avant la partie 1.3, ainsi que la partie 2.3 avant la partie 2.4.

Il est conseillé de bien lire l'entièreté des consignes avant de répondre aux questions, même si certaines questions préalables ne sont pas réalisées. Pour traiter une question de codage en Python, il est admis que les fonctions des questions précédentes sont correctement implémentées.

Voici la liste des seules fonctions et méthodes prédéfinies en Python qu'il est possible d'utiliser de manière totalement libre dans ce sujet :

- Les fonctions `range`, `len` et les fonctions de conversion de types.
- Les méthodes `append`, `pop` et `copy` sur les listes.
- Les méthodes `keys`, `values` et `items` sur les dictionnaires.

Si d'autres fonctions ou méthodes prédéfinies sont nécessaires ou autorisées, cela sera précisé dans les questions correspondantes.
Les questions demandant d'écrire du code Python utilisent les annotations de type usuelles du langage. Il est possible, mais non obligatoire, d'annoter de la même façon les réponses.

En ce qui concerne le langage SQL, il n'y a aucune restriction sur les instructions utilisées.

La première partie traite de l'ordonnancement de processus. Après quelques questions préliminaires (partie 1.1), deux premiers algorithmes d'ordonnancement classiques sont étudiés (partie 1.2), puis deux autres plus complexes prenant en compte les priorités entre processus (partie 1.3). Enfin, il est proposé une implémentation de files de priorité à l'aide de tas binaires (partie 1.4).

La seconde partie traite de la gestion des index des bases de données à l'aide de tables de hachage. Après une introduction sur le niveau logique des bases de données avec notamment le langage SQL (partie 2.1), une présentation du hachage est effectuée (partie 2.2). Enfin, le hachage dynamique linéaire est étudié à travers d'abord sa modélisation (partie 2.3) puis son implémentation (partie 2.4) en Python.

Partie 1. Ordonnancement de processus

Dans cette partie, nous allons étudier l'ordonnancement des processus d'un système d'exploitation interactif classique tels que ceux des distributions Linux. Nous considérerons travailler dans un système monoprocesseur.

Partie 1.1 — Questions préliminaires

Question 1. Définir les termes `thread` et `processus` et donner leur(s) différence(s).

On rappelle dans le tableau 1 les différents états possibles d'un processus.

État	En anglais	Signification
Exécution	Running (R)	Le processus est en cours de fonctionnement, il effectue un travail actif.
Sommeil	Sleeping (S)	Le processus est en attente d'un événement extérieur. Il se met en sommeil. Ce sommeil est dit « interruptible » car un signal peut le réveiller.
Sommeil non interruptible	Down (D)	Le processus est en attente exclusif d'un événement extérieur. Il se met en sommeil. Aucun signal ne peut le réveiller en dehors de l'événement dont il est en attente.
Arrêt	Stopped (T)	Le processus a été temporairement arrêtée par un signal. Il ne se réveillera qu'à la réception d'un signal spécifique.
Zombie	Zombie (Z)	Le processus est terminée, mais son statut de terminaison n'a pas encore été lu. La lecture de ce statut le supprimera définitivement de la liste des processus.

Tableau 1 – États possibles d'un processus

Question 2.

a) Donner deux commandes Linux permettant de connaître la liste des processus actifs sur le système ainsi que leurs états respectifs.

b) Si on lance l'une de ces commandes, donner l'état affiché pour le processus déclenché par cette commande, et l'état majoritairement affiché pour les autres processus.

Question 3. Expliquer, au travers d'un exemple pertinent, la notion d'interblocage, ainsi qu'une technique employée permettant de résoudre une situation d'interblocage.

Partie 1.2 — Premiers algorithmes d'ordonnancement

Dans cette sous-partie, nous considérons disposer en Python de deux classes `Pile` et `File` manipulant des objets quelconques, dont l'interface est donnée dans le tableau 2 ci-dessous.

Nom de la méthode	**Commentaires**
Constructeur	Création d'une structure vide.
`est_vide() -> bool`	Renvoie `True` si et seulement si la structure est vide..
`ajouter(e: object) -> None`	Ajout d'un élément dans la structure, donné en entrée de la méthode. L'élément est ajouté à la bonne place selon les caractéristiques de la classe associée.
`retirer() -> object`	Suppression d'un élément de la structure, renvoyé en sortie de la méthode. L'élément est retiré depuis la bonne place selon les caractéristiques de la classe associée.

Tableau 2 – Interface des classes `Pile` et `File`

Dans cette sous-partie, nous allons étudier 2 algorithmes d'ordonnancement. Les processus étudiés seront ceux définis dans le tableau 3 ci-dessous.

N° du processus	**Instant d'arrivée (après x ms)**	**Durée d'exécution (en ms)**
1	0	7
2	4	3
3	4	1
4	2	9

Tableau 3 – Processus étudiés dans cette sous-partie

Afin de simuler l'exécution de processus, nous allons définir la classe `Processus` possédant les attributs `pid` et `duree` définissant respectivement le numéro du processus et la durée d'exécution restante du processus.

Question 4. Définir en Python le constructeur de la classe `Processus`, et deux méthodes :

- `executer_durant(self, temps: int) -> int`
  Cette méthode prend en entrée le temps alloué pour l'exécution du processus, l'exécute durant ce temps autant que possible, sans que cela ne dépasse la durée restante du processus, et renvoie le temps réellement exécuté par ce processus.
- `est_termine(self) -> bool`
  Cette méthode renvoie un booléen permettant de savoir si l'exécution du processus est terminée.

On définit le dictionnaire Python `d_processus` (ligne 3 du code ci-dessous) permettant de stocker toutes les informations de la table :

```python
p1 = Processus(1, 7)
p2 = Processus(2, 3)
d_processus = {0: [p1], 4: [p2]}
p3 = Processus(3, 1)
p4 = Processus(4, 9)
a_ajouter = [(4, p3), (2, p4)]
d_processus[a_ajouter[0][0]] +=  a_ajouter[0][1]
d_processus[a_ajouter[1][0]] +=  a_ajouter[1][1]
```

Dans le dictionnaire `d_processus`, les clés sont les instants d'arrivée et les valeurs sont les listes des processus arrivant aux instants correspondants.

Question 5. Les deux dernières lignes du code ci-dessus permettent d'ajouter « à la main » les processus `p3` et `p4` dans le dictionnaire `d_processus`, mais il y a une erreur.

a) Corriger l'erreur et donner le contenu du dictionnaire après l'exécution de ce code corrigé.

b) Modifier ces deux dernières lignes afin d'ajouter dans le dictionnaire `d_processus` l'ensemble des processus se trouvant dans la liste `a_ajouter` sans connaître le contenu de cette liste.

Premier arrivé, premier servi

Le premier algorithme d'ordonnancement étudié sera celui basé sur la politique du « Premier arrivé, premier servi » qui consiste à exécuter les processus dans leur ordre d'arrivée. Cette politique est non préemptive, c'est-à-dire que le processus s'exécutera tant qu'il ne s'interrompt pas lui-même. Nous considérerons ici que l'auto-interruption d'un processus n'arrive jamais.

Question 6.

a) Donner une structure de données permettant d'implémenter cette politique. Discuter de la pertinence d'utiliser cette politique dans le cadre de l'ordonnancement des processus.

b) En suivant cette politique, représenter graphiquement le fil d'exécution des différents processus définis dans le tableau 3, sur une échelle de temps dont l'unité est la milliseconde.

c) Calculer la moyenne des temps totaux de chaque processus depuis leur arrivée jusqu'à leur terminaison.

Question 7. Définir en Python la fonction suivante implémentant cette politique :
`ordonnancement_paps(d_processus: dict[int, list[Processus]]) -> list[int]`
Cette fonction :

- prend en entrée le dictionnaire des processus défini à la question 5.
- renvoie une liste dont les indices sont les nombres de millisecondes écoulées depuis le début, et chaque valeur est le numéro du processus qui a été exécuté à l'instant considéré.

On considère que la clé de valeur 0 existe nécessairement dans le dictionnaire et que nous n'arriverons jamais dans une situation où un processus arrivera après la fin de l'exécution de tous les processus précédents. Il est possible de ne pas utiliser la structure trouvée à la question 6a pour implémenter cette fonction.

À la fin de l'exécution de cette fonction, tous les processus doivent avoir la valeur 0 pour leur attribut `duree`.
Exemple d'exécution :
`ordonnancement_paps({1: [Processus(1, 2)], 0: [Processus(2, 2)]})`
renvoie [2, 2, 1, 1]
On donne ci-dessous un extrait de la documentation de la fonction `sorted` prédéfinie en Python, et qui pourrait vous aider :

```
sorted(iterable, /, *, key=None, reverse=False)
    Renvoie une nouvelle liste triée depuis les éléments d'iterable.
```

Tourniquet

Le deuxième algorithme d'ordonnancement étudié est un algorithme plus courant : le tourniquet ou *round-robin*. Dans cet algorithme, un quantum est d'abord défini, et permet de savoir combien de temps processeur est alloué à chaque processus lorsqu'il s'exécute. Si un processus n'est pas terminé lorsque le quantum est terminé, on met à jour les données de ce processus. Cet algorithme utilise une file.

Question 8.

a) Dans cette question, nous considérons utiliser la mémoire cache du processeur pour l'ordonnancement. Dans le cas où un nouveau processus arrive à la fin d'un quantum et que le processus venant de s'exécuter n'est pas terminé, justifier qu'il est préférable de placer en queue de file d'abord le processus venant de s'interrompre plutôt que celui venant d'arriver.

b) On suppose que la stratégie de placement décrite à la question 8a est mise en œuvre. En suivant l'algorithme du tourniquet pour un quantum de 1ms, représenter graphiquement le fil d'exécution des différents processus définis dans le tableau 3, sur une échelle de temps dont l'unité est la milliseconde.

c) Calculer la moyenne des temps totaux de chaque processus depuis leur arrivée jusqu'à leur terminaison.

Question 9. En vous aidant des questions précédentes, justifier celui que vous utiliseriez entre l'algorithme « Premier arrivé, premier servi » et l'algorithme du tourniquet.

Le code incomplet ci-après définit en Python la fonction `ordonnancement_tourniquet` implémentant l'algorithme du tourniquet. Cette fonction a les mêmes entrées/sorties que la fonction d'ordonnancement de la question 7 :

- elle prend en entrée le dictionnaire des processus défini à la question 5.
- elle renvoie une liste dont les indices sont les nombres de millisecondes écoulées depuis le début, et chaque valeur est le numéro du processus qui a été exécuté à l'instant considéré.

On considère que la clé de valeur 0 existe nécessairement dans le dictionnaire et que nous n'arriverons jamais dans une situation où un processus arrivera après la fin de l'exécution de tous les processus précédents.
À la fin de l'exécution de cette fonction, tous les processus doivent avoir la valeur 0 pour leur attribut `duree`. On considère que le quantum est fixé à 1ms.

```python
def ordonnancement_tourniquet(d_processus: dict[int, list[Processus]])
                                                              -> list[int]:
    ordonnancement = []
    file = File()
    for processus in ...:
        file.ajouter(processus)
    t = 0
    while ...:
        processus = file.retirer()
        ordonnancement += [...] * processus.executer_durant(1)
        if not ...:
            file.ajouter(processus)
        t += 1
        if t in d_processus.keys():
            for processus in ...:
                file.ajouter(processus)
    return ordonnancement
```

Question 10.

a) Compléter les instructions manquantes de cette fonction. Vous pouvez ne recopier que les 5 lignes à compléter, à condition de reporter leur numéro.

b) Donner sans la justifier la complexité de la fonction complétée à la question a, puis démontrer sa correction totale.

c) Améliorer la fonction en y ajoutant un paramètre entier `quantum` définissant le quantum à appliquer à chaque ajout de processus dans la liste finale. Attention à bien prendre en compte le cas où un nouveau processus arrive pendant l'exécution d'un autre processus. Recopier entièrement la fonction.

Question 11.

a) L'exécution des fonctions des questions 7 et 10 engendre une modification des durées d'exécution des processus. Sans l'implémenter, proposer un attribut ou une méthode à ajouter dans la classe `Processus` afin de conserver les durées initiales des processus au sein du dictionnaire.

b) Définir en Python la fonction :

```python
moyenne_temps_processus(d_processus: dict[int, list[Processus]],
                        ordonnancement: list[int]) -> float
```

Cette fonction :

- prend en entrées :
  — le dictionnaire des processus défini à la question 5.
  — une liste telle que celles renvoyées par les questions 7 et 10.
- renvoie la moyenne des temps d'exécution de chaque processus.

Cette fonction doit donc renvoyer les valeurs correspondantes aux réponses des questions 6c et 8c lorsqu'on l'applique sur les données de l'exercice.

Partie 1.3 — Prise en compte des priorités entre les processus

L'algorithme du tourniquet évoqué précédemment part du principe que tous les processus ont une priorité égale. Or dans un système interactif, on aimerait par exemple exécuter prioritairement les processus que l'utilisateur a lancé, quitte à augmenter le temps d'exécution des processus en arrière-plan. De même, un processus ayant besoin d'accéder régulièrement à une ressource pourrait être exécuté en premier afin d'exécuter d'autres processus pendant que celui-ci attend une réponse de cette ressource.

Question 12. En suivant cette logique, justifier l'intérêt de rendre prioritaire un programme Python venant d'être lancé par l'utilisateur plutôt que son application de mail lui envoyant une notification à la réception d'un nouveau mail.

On ajoute donc un attribut entier `priorite` à la classe `Processus`, et on ajoute le paramètre correspondant au constructeur de cette classe.

Dans cette sous-partie, les processus étudiés sont définis dans le tableau 4 et on considère que la priorité de valeur 1 est la plus faible.

N° du processus	Arrivée après x ms	Durée d'exécution (en ms)	Priorité
1	3	2	4
2	0	3	1
3	4	6	3
4	4	3	2
5	7	1	5
6	1	5	2

Tableau 4 – Processus étudiés dans cette sous-partie

Priorités exclusives

Dans un premier temps, nous partons du principe qu'il n'y a préemption qu'à l'arrivée d'un processus plus prioritaire que l'actuel. En d'autres termes, un processus s'exécute tant qu'aucun processus de priorité strictement supérieure n'arrive, et l'arrivée d'un processus de priorité supérieure arrête temporairement le processus actuel.

Question 13.

a) En suivant cette politique, représenter graphiquement le fil d'exécution des différents processus définis dans le tableau 4, sur une échelle de temps dont l'unité est la milliseconde.

b) Que remarquez-vous pour le processus n°2 ? Est-ce raisonnable ?

Question 14.

a) Définir en Python la fonction :
`ordonnancement_termine(l_processus: list[Processus]) -> bool`
Cette fonction :

- prend en entrée une liste d'objets de la classe `Processus`.
- renvoie `False` s'il reste au moins un processus à exécuter, `True` sinon.

b) Définir en Python la fonction :
`priorite_processus(processus: Processus) -> int`
Cette fonction :

- prend en entrée un objet de la classe `Processus`.
- renvoie la valeur de son attribut `priorite`.

Le code ci-dessous définit en Python la fonction `ordonnancement_priorite` implémentant l'algorithme de l'ordonnancement par priorités exclusives. Cette fonction a les mêmes entrées/-sorties que les fonctions d'ordonnancement des questions 7 et 10 :

- elle prend en entrée le dictionnaire des processus défini à la question 5.
- elle renvoie une liste dont les indices sont les nombres de millisecondes écoulées depuis le début, et chaque valeur est le numéro du processus qui a été exécuté à l'instant considéré.

Pour simplifier, on fixe le quantum à 1ms, et donc il n'a pas besoin d'être mis en entrée de la fonction.

```
1  def ordonnancement_priorite(d_processus: dict[int, list[Processus]])
2                                                           -> list[int]:
3      ordonnancement = []
4      l_processus = d_processus[0].copy()
5      t = 0
6      while ordonnancement_termine(l_processus):
7          l_processus.sort(key=None, reverse=True)
8          processus = l_processus[0]
9          processus.executer_durant(1)
10         ordonnancement.append(processus.pid)
11         if processus.est_termine():
12             l_processus.pop()
13         t += 1
14         if t in d_processus.keys():
15             l_processus += d_processus[t]
16     return ordonnancement
```

On donne ci-dessous un extrait de la documentation de la méthode `sort` sur les listes :

```
sort(*, key=None, reverse=False)
    Cette méthode trie la liste sur place, en utilisant uniquement des
    comparaisons < entre les éléments. [...]
    sort() accepte deux arguments qui ne peuvent être fournis que nommés :
        - key spécifie une fonction d'un argument utilisée pour extraire une
        clé de comparaison de chaque élément de la liste. [...]
        - reverse, une valeur booléenne. Si elle est True, la liste d'éléments
        est triée comme si toutes les comparaisons étaient inversées.
```

Question 15. La fonction `ordonnancement_priorite` définie précédemment est actuellement incorrecte. Corriger en précisant leurs numéros, les lignes incorrectes afin que la fonction implémente l'algorithme de l'ordonnancement par priorités exclusives.

Ordonnancement par tirage au sort

Une des manières de résoudre les problèmes engendrés par les priorités exclusives est d'effectuer un ordonnancement par tirage au sort : on effectue un tirage au sort inéquitable en donnant plus de chances d'être tiré à un processus ayant une priorité plus grande. Pour cela, à chaque instant, on calcule la somme des priorités des processus présents, notée n, et on donne à chaque processus `priorite` chances sur n d'être tiré au sort.

Question 16. Définir en Python la fonction :
`somme_priorites(l_processus: list[Processus]) -> int`
Cette fonction :

- prend en entrée une liste d'objets de la classe `Processus`.
- renvoie la somme des priorités des processus de cette liste.

Le code incomplet ci-dessous définit en Python la fonction `ordonnancement_au_sort` implémentant l'algorithme de l'ordonnancement par tirage au sort. Cette fonction a les mêmes entrées/sorties que les fonctions d'ordonnancement des questions 7 et 10 :

- elle prend en entrée le dictionnaire des processus défini à la question 5.
- elle renvoie une liste dont les indices sont les nombres de millisecondes écoulées depuis le début, et chaque valeur est le numéro du processus qui a été exécuté à l'instant considéré.

Pour simplifier on fixe le quantum à 1ms : il n'a pas besoin d'être mis en entrée de la fonction.

```
1  from random import randint
2  def ordonnancement_au_sort(d_processus: dict[int, list[Processus]])
3                                                              -> list[int]:
4      ordonnancement = []
5      l_processus = d_processus[0].copy()
6      t = 0
7      while not ordonnancement_termine(l_processus):
8          rand = randint(...)
9          cumul_priorites = 0
10         i = 0
11         processus = None
12         while processus is None:
13             cumul_priorites += priorite_processus(...)
14             if rand <= cumul_priorites:
15                 ...
16             i += 1
17         processus.executer_durant(1)
18         ordonnancement.append(processus.pid)
19         if processus.est_termine():
20             l_processus.pop(...)
21         t += 1
22         if t in d_processus.keys():
23             l_processus += d_processus[t]
24     return ordonnancement
```

On donne ci-dessous la documentation de la fonction `randint` de la bibliothèque `random` :

```
random.randint(a, b)
    Renvoie un entier aléatoire N tel que a <= N <= b.
```

Question 17. Compléter les instructions manquantes de cette fonction. Vous pouvez ne recopier que les 4 lignes à compléter, à condition de reporter leur numéro.

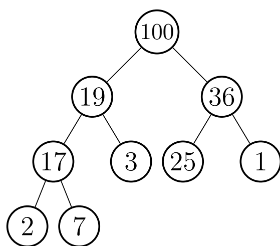
Partie 1.4 — Les tas binaires

On choisit dans cette sous-partie de prendre en compte la notion de priorité entre processus au travers d'une file de priorité, implémentée à l'aide d'un tas binaire.

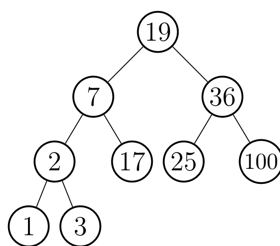
Un tas binaire est :

- Un arbre binaire complet : tous les niveaux de l'arbre binaire sauf le dernier doivent être totalement remplis et si le dernier ne l'est pas totalement, alors il doit être rempli de gauche à droite.
- Un tas : la clé de chaque nœud doit être supérieure ou égale aux clés de chacun de ses fils.

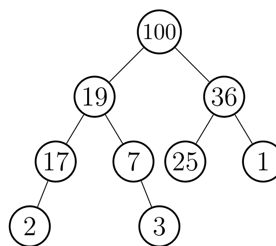
Question 18. En suivant la définition ci-dessus, donner en justifiant pour ceux qui ne le sont pas, parmi les 4 arbres ci-dessous, celui qui est un tas binaire.



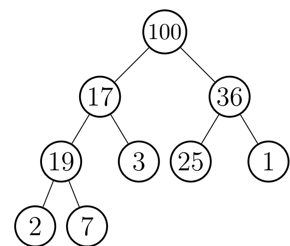
Arbre 1



Arbre 2



Arbre 3



Arbre 4

Question 19. En supposant que la priorité d'un processus est constante au cours du temps, expliquer quelles pourraient être les valeurs de chaque nœud du tas binaire dans le contexte du sujet.

Dans la suite, nous simplifierons en considérant ne stocker que des entiers dans le tas binaire.

Un tas binaire étant un arbre binaire complet, nous décidons de l'implémenter de manière compacte avec une liste Python :

- La racine se situe à l'indice 0
- Étant donné un nœud à l'indice i, son fils gauche est à l'indice $2i + 1$ et son fils droit à l'indice $2i + 2$
- Étant donné un nœud à l'indice $i > 0$, son père est à l'indice $\frac{i-1}{2}$ (arrondi à l'entier inférieur).

La figure 1 représente graphiquement un tas binaire sous forme de liste.

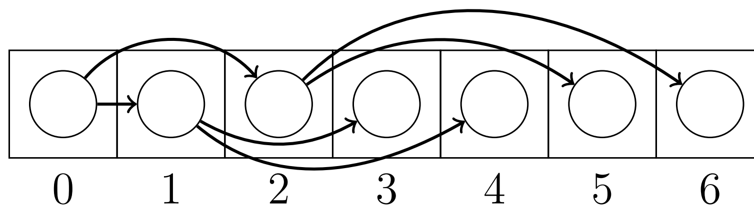


Figure 1 – Liste implémentant un tas binaire de 7 nœuds

Question 20.

a) En suivant cette implémentation, représenter la liste du tas binaire de la question 18.

b) À quel parcours de l'arbre correspond l'ordre des valeurs de la liste ?

Dans la suite, on se propose d'étudier 2 opérations sur les tas binaires : l'ajout d'un nouveau nœud et la suppression de la racine.

Ajout d'un nouveau nœud

L'opération d'insertion d'un nœud de valeur x consiste à suivre l'algorithme suivant :

- On insère x à la prochaine position libre, soit la position libre la plus à gauche possible sur le dernier niveau ;
- Puis, tant que x n'est pas la racine de l'arbre et que x est strictement supérieur à son père, on échange les positions de x et de son père.

Question 21.

a) En dessinant l'arbre à chaque étape de l'algorithme, l'appliquer en insérant la valeur 50 dans le tas binaire de la question 18.

b) De même, donner l'état de la liste à chaque étape de l'algorithme lors de l'insertion de la valeur 50.

Question 22.

a) Définir en Python la fonction récursive :
`percolation_haut(tas: list[int], i: int) -> None`
Cette fonction :
- prend en entrées :
  — un tas binaire sous forme de liste Python.
  — l'indice de la valeur à remonter dans l'arbre.
- effectue la deuxième partie de l'algorithme d'insertion dans un tas binaire, c'est-à-dire les échanges successifs de x et de son père.

b) Définir en Python la fonction :
`insertion_tas_binaire(tas: list[int], valeur: int) -> None`
Cette fonction :
- prend en entrées :
  — un tas binaire sous forme de liste Python.
  — une valeur entière.
- insère la valeur au bon endroit de la liste en appliquant l'algorithme d'insertion dans un tas binaire.

Cette fonction doit utiliser la fonction définie en a.

Question 23.

a) Donner en justifiant la complexité de la fonction `insertion_tas_binaire`.

b) Démontrer la correction totale de la fonction `percolation_haut`, puis en déduire celle de `insertion_tas_binaire`.

Suppression de la racine

L'opération de suppression de la racine d'un tas binaire, assez similaire à l'insertion, consiste à suivre l'algorithme suivant :
- On déplace à la place de la racine le nœud qui était en dernière position de l'arbre, que l'on note x ;
- Puis, tant que x n'est pas une feuille de l'arbre et que x est strictement inférieur à son fils de plus grande valeur, on échange les positions de x et de son fils de plus grande valeur.

Question 24.

a) En dessinant l'arbre à chaque étape de l'algorithme, l'appliquer en supprimant la racine du tas binaire de la question 18.

b) De même, donner l'état de la liste à chaque étape de l'algorithme lors de la suppression de la racine.

Question 25.

a) Définir en Python la fonction récursive :
`percolation_bas(tas: list[int], i: int) -> None`
Cette fonction :
- prend en entrées :
  — un tas binaire sous forme de liste Python qui vérifie la propriété de tas max sauf éventuellement en un nœud dont la valeur est inférieure à celle de son éventuel père, mais n'est pas nécessairement supérieure à celle de ces éventuels enfants.
  — l'indice i du nœud x éventuellement problématique qu'il faut alors descendre.
- effectue la deuxième partie de l'algorithme de suppression de la racine dans un tas binaire, c'est-à-dire les échanges successifs de x et de son fils de plus grande valeur.

b) Définir en Python la fonction :
`sup_racine_tas_binaire(tas: list[int]) -> int|None`
Cette fonction :
- prend en entrée un tas binaire sous forme de liste Python.
- supprime la racine de la liste en appliquant l'algorithme de suppression de la racine dans un tas binaire.
- renvoie la racine ainsi supprimée, ou `None` si le tas est vide.

Cette fonction doit utiliser la fonction définie en a.

Question 26.

a) Donner en justifiant la complexité de la fonction `sup_racine_tas_binaire`.

b) Démontrer la correction totale de la fonction `percolation_bas`, puis en déduire celle de la fonction `sup_racine_tas_binaire`.

Partie 2. Gestion des index des bases de données

Dans cette partie, nous allons aborder les niveaux logiques et physiques des bases de données.

Partie 2.1 — Niveau logique : utilisation d'un SGBD

Question 27. Donner la définition de l'acronyme SGBD et indiquer les fonctionnalités qu'il assure.

Les SGBD relationnels vérifient pour leurs transactions les propriétés d'Atomicité, d'Isolation, de Cohérence et de Durabilité (acronyme ACID).

Question 28. Définir chacune de ces propriétés.

Soit le schéma de base de donnée suivant dans lequel le nom des relations précède les parenthèses, les attributs et les domaines sont écrits dans les parenthèses, les attributs soulignés sont des clés primaires, et les attributs précédés du symbole # sont des clés étrangères.

- *Film(<u>id : `int`</u>, titre : `text`, année : `int`, durée : `int`, genre : `text`, pays : `text`)*
- *Personne(<u>id : `int`</u>, nom : `text`, prénom : `text`)*
- *Réalise(<u>#film_id : `int`, #personne_id : `int`</u>)*

Question 29. Écrire en SQL les requêtes permettant de :

a) créer la relation *Réalise*. On suppose que les autres relations sont créées. Votre réponse doit inclure la définition de la clé primaire et des clés étrangères ;

b) ajouter à la relation *Film* le film de genre *documentaire* réalisé en *France* en 2025, d'une durée de 130 minutes et d'identifiant 128 dont le titre est *"Le sujet"* ;

c) remplacer dans la relation *Personne* le prénom de la réalisatrice 64 par *Alyssa* ;

d) afficher les titres et les durées des films produits entre les années 2000 et 2020 (incluses) réalisés par la personne d'identifiant 32 ;

e) afficher la durée moyenne des films réalisés par *Quentin Tarantino*.

Partie 2.2 — Hachage de données

Index et tables de hachage

Pour optimiser les recherches dans les tables d'une base de données, il est courant d'utiliser des **index**. L'indexation suivant un attribut ou un n-uplet d'attributs crée une structure de donnée dédiée qui permet une recherche souvent bien plus efficace qu'un parcours séquentiel des enregistrements.

Parmi les structures existantes, il est possible d'utiliser les **tables de hachages**. De façon générale, une table de hachage permet de stocker des associations entre des *clés* et des *valeurs*.

Dans le contexte des SGBD et des index des bases de données, la clé d'un enregistrement est la valeur de l'attribut (ou du n-uplet d'attributs) défini sur un domaine D sur lequel est construit l'index et la valeur est l'adresse physique du bloc de données du disque dur qui contient l'enregistrement et ses données.

Avant d'évoquer plus concrètement le hachage des index en bases de données, cette sous-partie propose quelques rappels et exemples plus génériques sur le hachage.

Fonction de hachage

Tout hachage de données repose sur une fonction prenant en entrée un objet et renvoyant un entier dans un intervalle prédéfini. Cette fonction doit posséder certaines propriétés mathématiques comme l'assurance d'une répartition homogène vers les différentes valeurs entières (même si les objets sont proches) et certaines propriétés informatiques comme le fait que deux objets égaux doivent avoir la même valeur de hachage.

En Python, une telle fonction est prédéfinie : il s'agit de `hash(object) -> int` qui prend en paramètre un objet immuable et renvoie un nombre entier.

Question 30. Expliquer ce qu'est un objet immuable en donnant un exemple et un contre-exemple en Python.

Question 31.

a) À partir du code de la fonction `h` ci-dessous, indiquer le type de la variable `N` et les caractéristiques de la valeur renvoyée par h.

```
1 def h(x, N):
2     return hash(x) % N
```

b) Indiquer l'intérêt d'une telle fonction dans le contexte des tables de hachage.

Hachage de chaînes de caractères

Parmi les fonctions de hachage simples classiques sur les chaînes de caractères, la plupart s'appuie sur la représentation binaire de chaque caractère d'une chaîne, en additionnant chaque valeur binaire correspondante.

Question 32. Donner le nom de deux normes de représentation binaire des caractères en informatique, en précisant leurs différences et/ou similarités.

On donne dans le tableau 5 la représentation décimale des lettres minuscules de la langue française selon une norme répandue.

Lettre	a	b	c	d	e	f	g	h	i	j	k	l	m
Valeur	97	98	99	100	101	102	103	104	105	106	107	108	109
Lettre	n	o	p	q	r	s	t	u	v	w	x	y	z
Valeur	110	111	112	113	114	115	116	117	118	119	120	121	122

Tableau 5 – Représentation décimale des lettres minuscules de la langue française

Question 33.

a) Donner le résultat de la fonction de hachage d'addition des valeurs correspondantes à chaque caractère pour les mots suivants :
   - `"isn"`
   - `"nids"`
   - `"nsi"`

b) Que remarquez-vous ? Pourquoi ?

Une manière de régler le problème soulevé par la question 33 est de prendre en compte la position de chaque caractère dans le calcul de la somme, en pondérant chaque représentation de caractère par sa position dans la chaîne, en débutant les positions à 1. Le tableau 6 illustre les positions considérées pour la chaîne `"informatique"`.

Caractère	i	n	f	o	r	m	a	t	i	q	u	e
Position	1	2	3	4	5	6	7	8	9	10	11	12

TABLEAU 6 – Positions considérées pour la chaîne `"informatique"`

Question 34.

a) Appliquer ce correctif sur les chaînes de caractères de la question 33 en donnant leur résultat avec la nouvelle fonction de hachage présentée ci-dessus.

b) Définir en Python la fonction :
   `somme_caracteres(chaine: str) -> int`
   Cette fonction :
   - prend en entrée une chaîne de caractères.
   - renvoie la somme des représentations décimales de chaque caractère de la chaîne, pondérée par les positions non nulles de chaque caractère.

   Cette fonction doit donc renvoyer, pour les mots de la question 33, les valeurs de la question 34a. On donne ci-dessous un extrait de la documentation des fonctions `ord` et `chr` prédéfinies en Python, et qui pourraient vous aider :

```
ord(c)
    Renvoie le nombre entier représentant le code [...] du caractère
    représenté par la chaîne donnée. Par exemple, ord('a') renvoie le
    nombre entier 97.

chr(i)
    Renvoie la chaîne représentant un caractère dont le code de caractère
    [...] est le nombre entier i. Par exemple, chr(97) renvoie la chaîne
    de caractères 'a'.
```

Partie 2.3 — Modélisation du hachage des index en Python

Dans le contexte des index de bases de données, une table de hachage est concrètement munie d'un tableau (appelé *répertoire*) de n cases (appelées *alvéoles*), et d'une fonction $h : D \to [\![0, n-1]\!]$ appelée *fonction de hachage*, qui à une clé $c \in D$, associe l'indice i du *répertoire* dans lequel la $i^{\text{ème}}$ case contient l'adresse du bloc de données auquel appartient l'enregistrement de clé c.

La fonction de hachage n'étant pas nécessairement injective, deux clés distinctes peuvent être hachées vers le même indice i, on parle alors de *collision*. Les collisions sont gérés par des structures de listes chaînées.

Lorsque la taille du répertoire ne change pas, on parle de *hachage statique* qui n'est pas efficace pour les collections de taille variable. Le **hachage dynamique linéaire**, quant à lui, voit la taille de son répertoire évoluer en fonction des ajouts, et permet donc de s'adapter à la taille de la collection indexée. Il est ainsi bien plus efficace.

Dans ce problème, nous nous intéresserons au hachage dynamique linéaire qui associe la clé d'indexation d'un enregistrement à la tête d'une liste chaînée de blocs du disque dur dont l'un de ces blocs contient les données de l'enregistrement. Des fonctions de hachages sont utilisées de façon intermédiaire pour associer la clé de l'enregistrement à l'indice de la case du répertoire (appelée *alvéole*) pointant vers la tête de la liste chaînée.

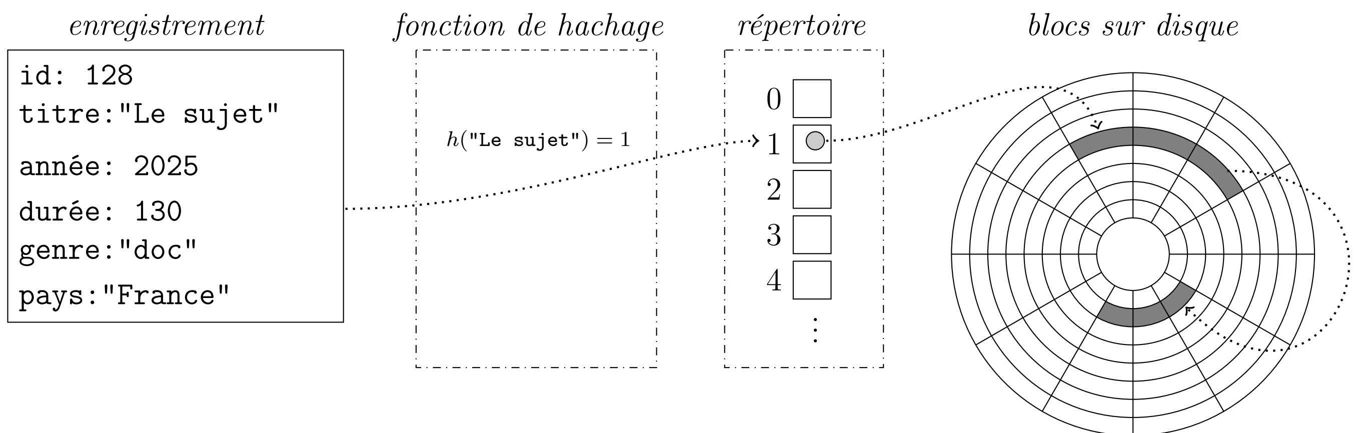


FIGURE 2 – Exemple de structure de table de hachage linéaire indexant l'attribut `titre`

Dans la suite et pour simplifier les exemples, les enregistrements écrits dans les blocs ne seront constitués que du hash binaire sur 4 bits de leurs clés d'indexation. Par exemple si la clé d'indexation est le titre des films et que le hash du texte `"Le sujet"` vaut 1, on écrira dans le bloc de données concerné uniquement la valeur `'0001'`.

Le schéma de la figure 3 ci-dessous représente un répertoire de 3 alvéoles (numérotées $00, 01$ et 10 en binaire) pointant vers des listes de blocs. Chaque bloc a une *capacité* égale à 2, c'est-à-dire qu'il contient au maximum 2 enregistrements. Ce schéma illustre une situation où :

- l'alvéole 00 pointe vers une liste chaînée contenant un unique bloc et l'enregistrement `0000` ;
- l'alvéole 01 pointe vers une liste chaînée contenant deux blocs et les enregistrements `1011`, `0101` et `0001` ;
- l'alvéole 10 pointe vers une liste chaînée contenant un unique bloc et les enregistrements `1010` et `0110`.

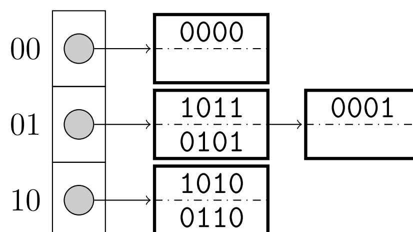


Figure 3 – Exemple de répertoire pointant vers des blocs chaînés

Pour la modélisation des blocs, nous définirons comme attributs (en supposant que le type `Enregistrement` existe) :

- `capacite: int` : nombre maximum d'enregistrements que le bloc peut contenir.
- `enregistrements: list[Enregistrement]` : séquence d'enregistrements. Initialement la séquence est vide et vaudra `[]`.
- `suivant: None|Bloc` : lien vers le bloc suivant auquel il est éventuellement chaîné. Lors de sa création, ce lien est vide et vaudra `None`.

Le tableau 7 ci-dessous présente l'interface de la classe `Bloc`.

Méthode d'instances	**Fonctionnalité**
Constructeur`(capacite: int)`	Création d'un bloc vide dont la capacité vaut `capacite` et qui n'est lié à aucun autre bloc.
`taille() -> int`	Renvoie le nombre total d'enregistrements que possède la liste chaînée de blocs dont la tête est l'instance courante.
`ajouter(e: Enregistrement) -> None`	Si l'attribut `enregistrements` de l'objet a un nombre d'élément inférieur à la capacité du bloc, alors ajoute l'enregistrement `e` dans `enregistrements`. Sinon, crée un nouveau bloc dans `suivant` s'il n'existe pas déjà puis ajoute récursivement `e` dans le bloc `suivant`.
`vider() -> list[Enregistrement]`	Déplace tous les enregistrements de la liste chaînée de blocs dont la tête est l'instance courante dans un tableau qui est renvoyé par la méthode. Après l'exécution de cette méthode, l'instance est donc un bloc qui n'est relié à aucun autre bloc et qui est vidé de tous ses enregistrements.

Tableau 7 – Interface de la classe `Bloc`

Question 35. En supposant que la classe `Bloc` a été implémentée, recopier et compléter les lignes 8 à 12 du code suivant afin que l'exécution ne lève aucune exception.

```
CAPACITE_MAX = 2
blocs = [Bloc(CAPACITE_MAX) for _ in range(3)]
blocs[0].suivant = Bloc(CAPACITE_MAX)
blocs[0].ajouter('110')
blocs[0].ajouter('010')
blocs[0].ajouter('000')
blocs[1].ajouter('001')
assert blocs[2].capacite               == ...
assert blocs[0].enregistrements[1]     == ...
assert blocs[1].suivant                == ...
assert len(blocs[0].enregistrements)   == ...
assert len(blocs)                      == ...
```

Question 36. Définir en Python selon l'interface définie dans le tableau 7 :

a) Le constructeur de la classe `Bloc`.

b) La méthode **récursive** `taille`.

c) La méthode **récursive** `ajouter`.

Question 37.

a) Justifier la correction partielle de la méthode `taille`.

b) Justifier la terminaison de la méthode `ajouter`.

Partie 2.4 — Tables de hachage linéaires

La structure de table de hachage linéaire voit son répertoire s'agrandir au fur et à mesure que des enregistrements y sont insérés. Ce changement de taille est dynamique et est déterminé en fonction d'un **taux de remplissage**. À chaque insertion, ce taux de remplissage est comparé à une valeur de seuil qu'il ne faut pas dépasser.

La fonction ci-dessous présente une implémentation en Python du calcul d'un taux de remplissage permettant de limiter la longueur des listes chaînées. Elle prend en argument une structure de table de hachage linéaire (séquence de listes chaînées de blocs) et renvoie un nombre décimal. Dans cette implémentation, `CAPACITE_MAX` est une constante égale à la capacité maximale des blocs.

```
def taux(repertoire: list[Bloc]) -> float:
    n_enregistrements = 0
    for i in range(len(repertoire)):
        n_enregistrements += repertoire[i].taille()
    n_total = len(repertoire) * CAPACITE_MAX
    return n_enregistrements/n_total
```

Question 38. En prenant comme argument la structure définie par l'exemple de la figure 3, déterminer la valeur renvoyée par la fonction `taux`.

Les tables de hachage linéaires sont des structures dont la taille évolue dynamiquement à mesure que le nombre d'enregistrements référencés augmente. On appelle **éclatement** le fait d'ajouter une alvéole à une structure.

Pour illustrer ce principe, étudions l'évolution d'une structure de table de hachage :

- qui ne contient initialement aucun enregistrement et qui se remplit ;
- qui manipule des blocs de capacité maximale 2 ;
- qui a pour constante $N = 2$ (nombre initial d'alvéoles) et pour variable $p = 00$ (marqueur qui identifie la première alvéole et qui va évoluer au fur et à mesure du remplissage afin de marquer successivement les alvéoles).

Le schéma de la figure 4 illustre 5 états de la structure à différents moments durant son remplissage. Les états sont numérotés de (a) à (e). Le marqueur p est la valeur grisée sur la figure.

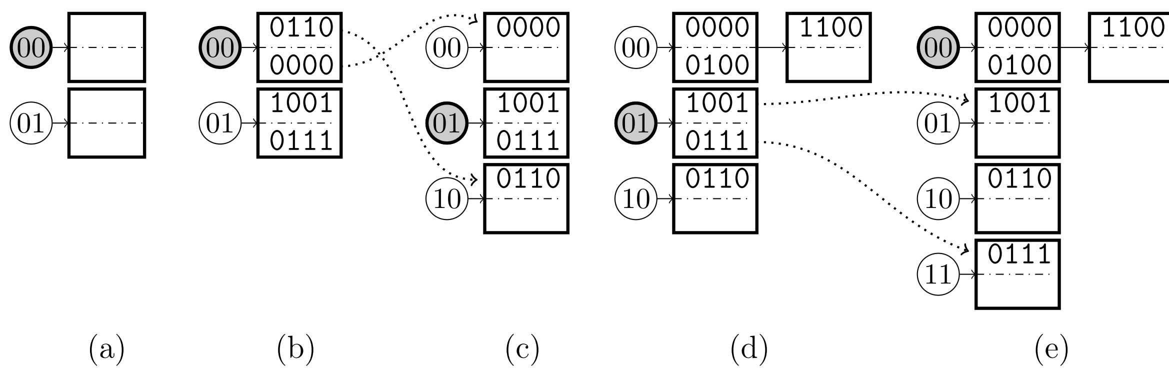


FIGURE 4 – Exemple de hachage linéaire pour $N = 2$

<u>**1ère série d'éclatement :**</u>

Étape (a) $N = 2$ et $p = 00$. La structure est donc une séquence de 2 blocs vides numérotés 00 et 01. C'est l'alvéole 00 qui est marquée pour le prochain *éclatement*.

Étape (b) Pour arriver dans cet état, il y a ajout de 3 enregistrements **sans** éclatement puis d'un 4ème qui va déclencher l'**éclatement** de l'alvéole marquée par $p = 00$, et faire passer la structure de l'étape (b) à l'étape (c).

Étape (c) L'alvéole 10 a été ajoutée. Les enregistrements de l'alvéole éclatée 00 se sont répartis entre elle (00) et la nouvelle (10). C'est désormais l'alvéole suivante qui est marquée par la variable $p = 01$.

Étape (d) 2 enregistrements ont été ajoutés. Il y a trop d'enregistrements. Il va y avoir **éclatement** de l'alvéole marquée par $p = 01$. Comme cette alvéole était la dernière alvéole du début de cette série (étape (a)), cet éclatement termine cette série.

<u>**Fin de la 1ère série d'éclatement**</u>

Étape (e) L'alvéole 11 a été ajoutée et les enregistrements de l'alvéole éclatée 01 se sont répartis entre cette dernière et la nouvelle. Il y a désormais 4 alvéoles dans la structure. Ce nombre a doublé par rapport à la première étape (a).

La prochaine série d'éclatement commence par une réinitialisation du marqueur à $p = 00$. L'étape (e) constitue ainsi l'état de la structure au début de la 2ème série d'éclatement. À partir de maintenant et jusqu'au prochain doublement de la taille, ce seront successivement les 4 alvéoles qui seront éclatées et réparties jusqu'à l'éclatement de la 7ème alvéole. Et ainsi de suite.

Les étapes (a) à (d) de la figure 4 montrent donc la 1ère série d'éclatement d'une structure de table de hachage linéaire de taille initiale $N = 2$.

Question 39. En partant d'un répertoire de taille initiale $N > 0$, déterminer combien il y a eu de séries d'éclatements complètes lorsque le répertoire possède k alvéoles avec $k \in \mathbb{N}$ et $k > N$.

Les fonctions de hachages

Plaçons nous dans le cas d'une $i^{\text{ème}}$ série d'éclatements (pour $i > 0$) avec un répertoire de taille initiale $\underline{N = 2}$. Lors de cette série, la table de hachage est munie de deux fonctions de hachages :

$$\begin{cases} hash(x) & = & \text{fonction de hachage initiale} \\ h_i(x) & = & \text{hash}(x) \bmod 2^i \qquad\qquad \text{pour } i > 0 \end{cases}$$

qui utilisent la fonction *modulo*, notée mod , qui calcule le reste de la division euclidienne. Dans ce cas particulier et avec $a \in \mathbb{N}$ et $n \geq 0$, la fonction modulo peut aussi être définie par :

$$a \bmod 2^n = \text{ nombre formé par les } n \text{ bits de poids faibles de } a$$

Lors de l'éclatement de l'alvéole $k < 2^i$, la nouvelle alvéole $(k + 2^i)$ est ajoutée au répertoire. On souhaite alors que les enregistrements se répartissent entre l'alvéole éclatée et la nouvelle. Pour cela, l'astuce est d'utiliser la fonction de hachage h_{i+1}.

Question 40. Vérifier la propriété précédente en montrant que pendant la $i^{\text{ème}}$ série d'éclatement, lorsque l'alvéole $k < 2^i$ éclate, la valeur de hachage des clés de ses enregistrements avec la fonction h_{i+1} vaut k ou $k + 2^i$.

Insertion avec un hachage linéaire

L'objectif est d'implémenter une table de hachage linéaire. Pour cela nous allons confondre la table de hachage avec son répertoire qui est une séquence de blocs de type `list[Bloc]` et implémenter :

- la fonction de hachage `h_i`.
- la fonction `eclater` qui modifie la structure en éclatant un bloc donné et en répartissant les enregistrements dans les bonnes alvéoles.
- et pour finir la fonction `inserer` qui insère dans la structure un nouvel enregistrement.

Voici ci-dessous les spécifications des fonctions nécessaires :

Python	Fonctionnalité
`CAPACITE_MAX: int`	Constante contenant le nombre d'enregistrements que peut contenir au maximum un bloc (*capacité*).
`TAUX_REMPLISSAGE: float`	Constante contenant un nombre compris entre `0.0` et `1.0` qui indique le taux de remplissage maximal acceptable pour la structure.

Python	Fonctionnalité
`h_i(i: int, x) -> int`	Fonction de hachage qui prend en argument l'indice $i > 0$ de la série courante d'éclatement et x la valeur de la clé à hacher. La fonction renvoie la valeur de hachage $h_i(x)$.
`eclater(table: list[Bloc], i, p) -> None`	Prend en argument la *table* à modifier, l'indice i de la série courante d'éclatement et p le marqueur indiquant l'alvéole à éclater. Cette fonction vide la chaîne de blocs pointée par l'alvéole p en mémorisant temporairement les enregistrements. Une nouvelle alvéole est ajoutée à *table* pointant vers un nouveau bloc de capacité égale à `CAPACITE_MAX`. Enfin, ajoute chaque enregistrement x mémorisé dans l'alvéole d'indice $h_{i+1}(x)$.
`inserer(x, table, i, p) -> tuple[int, int]`	Insère l'enregistrement x dans *table*. Prend aussi en argument i l'indice de la série d'éclatement courante et p le marqueur indiquant l'alvéole à *éventuellement* éclater. Cette fonction renvoie le couple d'entiers i et p qui ont pu voir leurs valeurs changer (en cas d'éclatement avec ou sans réinitialisation de la série d'éclatements).

Tableau 8 – Spécification des constantes et des fonctions

Question 41. En respectant les spécifications du tableau 8, implémenter les fonctions :

a) `h_i(i, x)` : en utilisant la fonction `hash` de la bibliothèque standard et en confondant pour simplifier l'enregistrement x et sa clé ;

b) `eclater(table, i, p)` : en utilisant la classe `Bloc` définie dans la sous-partie 2.3 et dont l'interface est disponible dans le tableau 7.

Insérer un nouvel enregistrement x dans la table de hachage se déroule en plusieurs étapes :

1. L'indice de l'alvéole associée à la clé de l'enregistrement est déterminé par la fonction de hachage. Pour cela l'indice est calculé par $h_i(x)$ et si l'indice obtenu est inférieur au marqueur p alors il est recalculé par la fonction $h_{i+1}(x)$.
2. L'enregistrement est ajouté à la structure.
3. Si le taux de remplissage de la table est supérieur ou égal au seuil, alors l'alvéole p éclate et p est mis à jour.
4. Suite à l'éventuel éclatement, si p marque l'alvéole 2^i, c'est que la taille du répertoire a doublé durant cette série d'éclatement. C'est donc la fin de la série i. La variable i est incrémentée et le marqueur p est réinitialisé à l'indice de la première alvéole.
5. Enfin, pour transmettre l'état de la structure les valeurs i et p sont renvoyées.

Question 42. En vous appuyant sur l'interface de la classe `Bloc` et les fonctions vues précédemment, implémenter la fonction `inserer`.