Dépendances. Ce sujet contient trois parties indépendantes qui doivent être traitées toutes les trois. On veillera à bien indiquer sur la copie les changements de partie.

Attendus. Il est attendu des candidates et des candidats des réponses construites. Ils seront aussi évalués sur la précision, le soin et la clarté de la rédaction.

Partie I. Robot de dépôt de solutions chimiques

Cette partie doit être traitée dans le langage C avec des appels de type POSIX. Un rappel de certaines fonctions est disponible en annexe. Aucun `#include` n'est demandé. Ce sujet s'intéresse principalement à la synchronisation de processus légers (appelés dans la suite threads) à l'aide de mutex et de sémaphores. Les trois sections sont indépendantes.

On s'intéresse à la partie contrôle d'un ou plusieurs robots d'injections de produits chimiques dans des éprouvettes. On considère avoir un bras injecteur contrôlable par un ou plusieurs threads sur un ordinateur connecté à ce bras. Dans la suite, on supposera avoir une bibliothèque de fonctions dont on donnera la spécification mais dont on supposera ne pas pouvoir modifier ni consulter le code.

On s'intéresse à résoudre les problèmes découlant du cas où plusieurs threads d'un programme interagissent avec le robot au travers de la bibliothèque de contrôle en même temps. On considère exister la structure `struct formule`.

Dans l'ensemble du sujet, on suppose avoir un seul processeur avec un seul cœur.

1 Une rangée d'éprouvettes

On suppose dans cette partie être dans le cas où le programme composé de plusieurs threads contrôle un seul bras pouvant se déplacer au dessus d'une rangée unique de `N>0` éprouvettes.

On suppose avoir trois fonctions appelée, `int libre()`, `void aller(int i)` et `void injecter(struct formule *f)` qui respectivement permettent d'obtenir l'indice d'une éprouvette vide, au bras d'aller à l'éprouvette `i` entre `0` inclus et `N` exclus, et enfin d'injecter une solution dont la formule est passée en argument dans l'éprouvette actuellement sous le bras que l'on considère vide. Dans cette section on considère avoir assez d'éprouvettes par rapport au nombre d'expériences prévues.

Le comportement d'une fonction utilisant le bras sera alors par exemple (sous l'hypothèse qu'il y a toujours au moins une éprouvette vide disponible) :

```
1 int déposer(struct formule *f) {
2     int i = libre();
3     aller(i);
4     injecter(f);
5     return i;
6 }
```

Question 1.1. On se place dans le cas où deux threads tentent « en même temps » d'utiliser la fonction `déposer` décrite ci-dessus. Explicitez un exemple montrant qu'il est possible que les deux threads déposent leur solution dans la même éprouvette.

Question 1.2. On tente de résoudre le problème avec des mutex

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int déposer(struct formule *f) {
4    pthread_mutex_lock(&mutex);
5    int i = libre();
6    pthread_mutex_unlock(&mutex);
7
8    aller(i);
9    injecter(f);
10   return i;
11 }
```

Quel est le problème de cette solution ?

Question 1.3. On tente de corriger le problème de la solution précédente comme suit :

```
1  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2
3  int déposer(struct formule *f) {
4    pthread_mutex_lock(&mutex);
5    int i = libre();
6    aller(i);
7    injecter(f);
8    return i;
9    pthread_mutex_unlock(&mutex);
10 }
```

Cette version fonctionne-t-elle ? Expliquez pourquoi et corrigez-la si elle ne fonctionne pas.

Question 1.4. On suppose, pour cette question **uniquement**, que la formule est à encoder pour le modèle spécifique de bras : la fonction `injecter` n'est pas capable de directement traiter la formule de type `struct formule`. La fonction `encoder` permet de traduire la formule pour le bras utilisé. Le code de base deviendrait:

```
1  int déposer(struct formule *f) {
2     int i = libre();
3     aller(i);
4     struct codage code = encoder(f, "bras␣modèle␣1");
5     injecter(&code);
6
7     return i;
8  }
```

On considère aussi que la fonction (sans effets de bords) `encoder` est très lente, beaucoup plus lente que le temps de déplacement du bras. Modifiez le code précédent pour qu'il soit possible par plusieurs threads de l'appeler en parallèle sans problème de synchronisation et que ces appels soient le plus efficace possible.

1.1 Ajout du vidage des éprouvettes

On considère maintenant avoir la fonction `struct resultat *analyser()` qui permet de vider et d'analyser l'éprouvette qui est sous le bras. Le code d'un thread complet serait alors (en considérant que le mutex `mutex` a été créé en tant que variable globale) :

```
1 ...
2 struct formule f = creer_formule(..);
3 int pos = déposer(&f);
4 ... // attente avec calcul du temps de déplacement du bras
5 pthread_mutex_lock(&mutex);
6 aller(pos)
7 struct resultat *res = analyser()
8 pthread_mutex_unlock(&mutex);
9 ...
```

Question 1.5. Dans cette question, on considère toujours avoir au moins une éprouvette vide lors de l'appel à la fonction `libre`. Est-il possible de garantir que l'attente est d'une heure (à la seconde près) entre l'appel à `injecter` (dans la fonction `déposer`) et l'appel à la fonction `analyser` dans le cas de deux threads ? On considère que le temps de déplacement du bras est déterministe (calculable en fonction de sa position actuelle et de sa destination) et est de l'ordre de la minute. On justifiera la réponse.

Question 1.6. Même question si on est dans le cas d'un seul thread toujours à la précision de la seconde ? Avec une précision d'un cycle de processeur ? On justifiera les deux réponses.

Question 1.7. On suppose avoir maintenant plus d'expériences à faire que d'éprouvettes. On propose la modification suivante de la fonction `déposer`. On considère toujours avoir un mutex `mutex` global initialisé. On considère aussi qu'après que la fonction `analyser` termine de vider une éprouvette, la fonction `libre` est capable de renvoyer sa position. Lorsqu'il n'y a pas d'éprouvette vide, la fonction `libre` renvoie la valeur `-1`. On transforme la fonction `déposer` :

```
1  int déposer(struct formule *f) {
2    int i = -1;
3    while (i == -1) {
4      pthread_mutex_lock(&mutex);
5      i = libre();
6
7      if (i != -1) {
8        aller(i);
9        injecter(f);
10     }
```


Tournez la page S.V.P.

```
11      pthread_mutex_unlock(&mutex);
12    }
13    return i;
14  }
```

Est-ce que cela fonctionne ? Justifier votre réponse

Question 1.8. En utilisant la fonction `déposer` de la question précédente, explicitez quel impact négatif a cette approche sur les autres threads et/ou sur les autres processus ?

Question 1.9. On suppose maintenant avoir accès aux sémaphores, et il n'y a plus accès qu'aux fonctions `aller`/`injecter`/`analyser` (i.e. plus à la fonction `libre`). Proposer une implémentation en C des fonctions `déposer` et `récupérer` en précisant les arguments et en justifiant ceux éventuellement ajoutés. Le but est d'autoriser l'utilisation suivante:

```
1  ???
2  struct formule f = creer_formule(...)
3  int pos = déposer(&f, ???)
4  /// attendre en fonction de la formule
5  struct resultat *res = récupérer(pos, ???)
6  ???
```

Précisez les variables partagées utilisées et l'initialisation du sémaphore dans la fonction `main` si besoin. On s'attachera à ce que l'implantation proposée résolve le problème soulevé dans la question précédente. On considérera avoir accès à une fonction `int trouver(int *tab, int n)` qui trouve et renvoie la valeur de l'indice dans le tableau `tab` de `n` entiers qui a pour valeur `0`. Si aucune valeur n'est `0` alors la fonction renvoie `-1`.

Question 1.10. L'implémentation proposée de la synchronisation est dans le cadre des threads, expliquer les types de modifications qu'il aurait fallu faire pour réaliser cette synchronisation dans un cadre multi-processus.

1.2 Passage à la 2D

Pour augmenter le nombre d'expériences possibles en même temps, on ajoute plusieurs rangées d'éprouvettes (toutes de la même taille), tout en gardant un seul bras. On ajoute un moteur permettant de changer de rangée d'éprouvettes. On a donc maintenant deux fonctions indépendantes `allerX` et `allerY`. On a respectivement maintenant `NY` rangées de `NX` éprouvettes. La fonction `libreXY` modifie ses arguments pour renvoyer les coordonnées d'un emplacement vide au moment de son appel. Dans cette partie, on suppose à nouveau avoir assez d'emplacement pour l'ensemble des expériences.

```
1 void déposer(struct formule *f) {
2   int x,y;
3   libreXY(&x, &y);
4   allerX(x);
5   allerY(y);
6   injecter(f);
7 }
```

Question 1.11. Combien de mutex au minimum sont nécessaires pour permettre l'utilisation de la fonction `déposer` par plusieurs threads du même programme en garantissant uniquement le bon fonctionnement du système (i.e. on ne s'intéressera pas à la problématique de performance dans cette question). Modifier le code précédent pour ajouter cette ou ces mutex.

Question 1.12. Les deux moteurs sont indépendants, plutôt que d'attendre que le déplacement sur X soit fini avant de faire le déplacement sur Y, il est possible de faire les deux en même temps. Proposez une implémentation de la fonction déposer réalisant cette amélioration.

1.3 Analyse non destructive dans le cas d'une seule rangée

On change légèrement l'expérience tout en gardant le même matériel. Les expérimentateurs mettent des produits dans chacune des éprouvettes et modifient légèrement le matériel pour que l'analyse ne vide plus les éprouvettes. Les expérimentateurs veulent étudier l'évolution du produit dans le temps et vont donc appeler la fonction `analyser` régulièrement. On revient dans le cas où l'on a une seule rangée et on considère que les éprouvettes sont déjà remplies au lancement du programme.

Une version séquentielle d'un thread de suivi d'une des expériences serait donc :

```
1 ...
2 while(true) {
3   aller(i)
4   struct resultat *res = analyser()
5   traiter(res, thread_data);
6   free(res);
7 }
```

La fonction `traiter` utilise uniquement des données internes à un thread (la variable `thread_data`) et donc il n'y a aucun problème à en exécuter plusieurs en même temps.

On se rend compte que le moteur est assez lent, mais est 10 fois plus rapide pour aller en valeurs croissantes que décroissantes. Par exemple, en étant en position 1, aller(2) est dix fois plus rapide que aller(0).

Question 1.13. Expliciter le déplacement de la tête dans le cas où les évènements suivants se produisent :

Tournez la page S.V.P.

- L'analyse de 5 est en cours et a commencé à T0 (qui finira à T0+6)
- Un autre thread lance à T0+1 l'analyse de 3 (qui prendra 1)
- Un autre thread lance à T0+2 l'analyse de 7 (qui prendra 4)
- Un autre thread lance à T0+8 l'analyse de 9 (qui prendra 1)

Toutes les durées sont en Unité de Temps (UT). On donnera l'ordre des actions ainsi que les UT correspondantes par rapport à T0 (donc T0+3 sera donné comme 3). On considère que déplacer le bras d'une position en valeur croissante prend une UT. On considère aussi que l'on a déjà réglé le problème de synchronisation entre les threads.

Question 1.14. Écrire une fonction `analyser_en(i)` qui garantit que si plusieurs analyses sont en attente, celle qui sera débloquée sera soit celle dont la position est la plus proche supérieurement si elle existe, soit celle dont la position est la plus petite sinon.

Annexe

```
pthread_create // Créer un nouveau thread
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);

pthread_join // Attendre la fin d'un autre thread
int pthread_join(pthread_t th, void **thread_return);


pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock
                  , pthread_mutex_destroy // Opérations sur les mutex
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

sem_init // Initialiser un sémaphore non nommé
int sem_init(sem_t *sem, int pshared, unsigned int value);

sem_destroy // Détruire un sémaphore non nommé
int sem_destroy(sem_t *sem);

sem_post // Déverrouiller un sémaphore
int sem_post(sem_t *sem);

sem_wait // Verrouiller un sémaphore
int sem_wait(sem_t *sem);
```

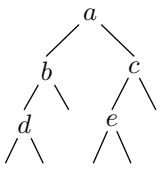
Partie II. Tableaux flexibles

Cette partie doit être traitée dans le langage OCaml.

Dans ce problème, on représente un tableau par un arbre binaire. Si le tableau est vide, il est représenté par l'arbre vide. Sinon, l'élément d'indice 0 du tableau est stocké à la racine de l'arbre, le sous-arbre gauche contient une représentation des éléments d'indices de la forme $2i+1$ et le sous-arbre droit contient une représentation des éléments d'indices de la forme $2i+2$. Ainsi, le tableau de taille 5

0	1	2	3	4
a	b	c	d	e

est représenté par l'arbre binaire



avec l'élément a d'indice 0 à la racine, le sous-tableau $[b, d]$ des éléments d'indices impairs récursivement organisé pour former le sous-arbre gauche et le sous-tableau $[c, e]$ des éléments d'indices pairs récursivement organisé pour former le sous-arbre droit.

Question 2.1. Dessiner l'arbre binaire correspondant à un tableau de taille 12.

Mise en œuvre en OCaml. On suppose que les éléments sont d'un type `elt` donné qui n'est pas précisé. Un tableau flexible est représenté par une valeur du type OCaml suivant :

```
type tree = E | N of tree * elt * tree
```

La valeur `E` représente l'arbre binaire vide. Une valeur $\texttt{N}(\ell, x, r)$ représente un arbre binaire non vide, avec la valeur x à la racine, un sous-arbre gauche ℓ et un sous-arbre droit r. La taille d'un arbre t, notée $|t|$, est son nombre de nœuds. On a donc :

$$\begin{aligned} |\texttt{E}| &= 0 \\ |\texttt{N}(\ell, x, r)| &= 1 + |\ell| + |r| \end{aligned}$$

Question 2.2. Écrire une fonction `get: int -> tree -> elt` qui reçoit en arguments un entier i et un tableau flexible t, avec $0 \leq i < |t|$, et qui renvoie l'élément d'indice i du tableau représenté par t. La complexité doit être proportionnelle à la hauteur de t. On ne demande pas de la justifier.

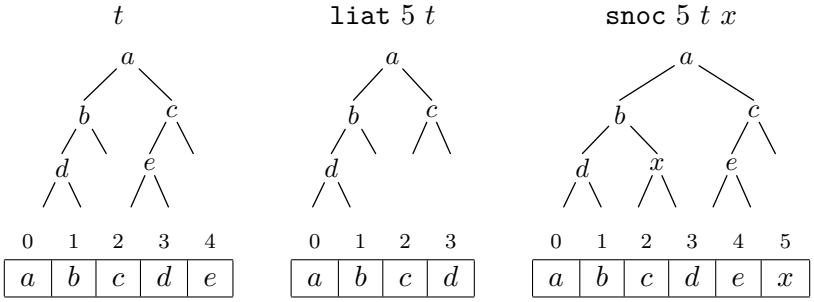
Tournez la page S.V.P.

Invariant structurel. On remarque que dans un tableau flexible, le sous-arbre gauche a toujours au moins autant de nœuds que le sous-arbre droit, car il y a au moins autant d'indices de la forme $2i+1$ que d'indices de la forme $2i+2$. Plus précisément, *pour tout sous-arbre* $\texttt{N}(\ell, x, r)$ d'un tableau flexible, on a

$$|r| \leq |\ell| \leq |r| + 1 \tag{1}$$

Question 2.3. Montrer que la hauteur d'un tableau flexible de taille n est en $\mathcal{O}(\log n)$.

Agrandir/rétrécir du côté droit. Notre objectif est de réaliser quatre opérations pour respectivement agrandir ou rétrécir le tableau d'une case sur l'une ou l'autre de ses extrémités. Commençons par le côté droit, c'est-à-dire celui des indices les plus grands. L'opération `liat` n t supprime le dernier élément d'un (arbre représentant le) tableau t de taille n. L'opération `snoc` n t x ajoute un élément x à la fin d'un (arbre représentant le) tableau t de taille n. Si on reprend l'exemple du tableau donné en introduction, ces deux opérations donnent respectivement les arbres suivants :



En dessous de chaque arbre, on a dessiné le tableau qu'il représente. Voici par exemple le code de la fonction `liat` :

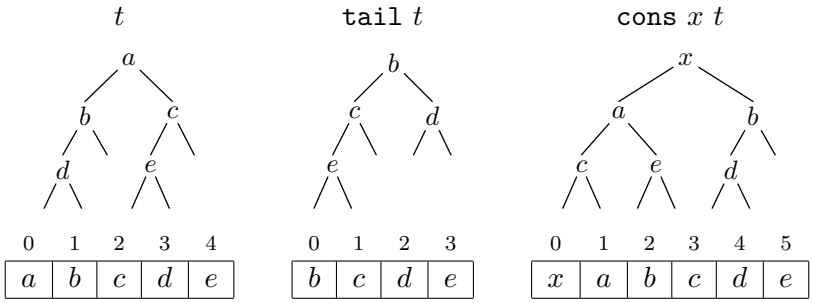
```
let rec liat n = function
  | N(E, _, E) -> E
  | N(l, x, r) -> if n mod 2 = 0 then N(liat (n / 2) l, x, r)
                                 else N(l, x, liat (n / 2) r)
  | E -> assert false
```

Question 2.4. Pour un tableau flexible t de taille n, donner la complexité de `liat` n t en fonction de n.

Question 2.5. Montrer la correction totale de la fonction `liat`, c'est-à-dire que, sous l'hypothèse que t est un tableau flexible de taille $n \geq 1$, alors `liat` n t termine et renvoie bien un tableau flexible de taille $n-1$ avec les éléments attendus.

Question 2.6. Donner un code pour la fonction `snoc: int -> tree -> elt -> tree`. Pour un tableau flexible t de taille n, et un élément x, l'appel `snoc` n t x renvoie un tableau flexible de taille $n+1$ avec les mêmes éléments que t aux indices $0 \leq i < n$ et l'élément x à l'indice n. La complexité doit être $\mathcal{O}(n)$. On ne demande pas de la justifier.

Agrandir/rétrécir du côté gauche. Agrandir ou rétrécir le tableau du côté gauche, c'est-à-dire du côté des indices les plus petits, est plus délicat à réaliser, car on décale alors les indices des éléments. L'opération `tail` t supprime le premier élément du tableau t. L'opération `cons` x t ajoute un élément x au début du tableau t. Si on reprend toujours le même exemple, alors ces deux opérations donnent respectivement les arbres suivants :



On prendra le temps de bien comprendre ce qui se passe ici. En particulier, on observera comment les éléments passent d'un côté à l'autre de l'arbre. En effet, les éléments situés à des indices pairs (resp. impairs) avant se retrouvent à des indices impairs (resp. pairs) après.

Question 2.7. Donner un code pour la fonction `tail: tree -> tree`. Pour un tableau flexible t de taille $n \geq 1$, l'appel `tail` t renvoie un tableau flexible de taille $n-1$ où le premier élément de t a été supprimé. La complexité doit être $\mathcal{O}(\log n)$ et on demande de la justifier.

Question 2.8. Donner un code pour la fonction `cons: elt -> tree -> tree`. Pour un tableau flexible t de taille n, l'appel `cons` x t renvoie un tableau flexible de taille $n+1$ où un premier élément x a été ajouté à gauche des éléments de t. La complexité doit être $\mathcal{O}(\log n)$. On ne demande pas de la justifier.

Construction à partir d'un vrai tableau. Si on se donne un (vrai) tableau `a`, de type `elt array` et de taille n, on peut chercher à construire un tableau flexible contenant les mêmes éléments que `a`. Si on le fait naïvement en appelant n fois la fonction `snoc` (ou n fois la fonction `cons`) alors la complexité sera $\mathcal{O}(n \log n)$ au total. Mais on peut faire mieux.

Question 2.9. Écrire une fonction `of_array: elt array -> tree` qui reçoit en argument un tableau `a` de taille n et construit un tableau flexible ayant les mêmes éléments que `a`, en temps $\mathcal{O}(n)$. Indication : on pourra introduire une fonction auxiliaire qui, pour deux entiers $m \geq 1$ et $k \geq 0$ donnés, construit le tableau flexible des éléments de `a` d'indices $mi + k$.

Tableau flexible constant. Pour cette dernière question, on cherche à construire un tableau flexible d'une certaine taille n dont tous les éléments sont identiques à un élément donné. Comme pour la question précédente, on pourrait le construire à partir des fonctions `cons` ou `snoc`, avec une complexité totale $\mathcal{O}(n \log n)$. Mais là encore, on peut faire mieux, et même beaucoup mieux !

Tournez la page S.V.P.

Question 2.10. Écrire une fonction `make: int -> elt -> tree` qui reçoit en arguments un entier $n \geq 0$ et un élément x, et renvoie un tableau flexible de taille n dont tous les éléments sont égaux à x. La complexité en temps doit être $\mathcal{O}(\log n)$. On demande de la justifier. Donner par ailleurs la complexité en espace.

Partie III. Gestion d'un concours de recrutement

Cette partie considère une version très simplifiée de la gestion d'un concours de recrutement comme celui de l'agrégation. La première section considère des bases de données pour la gestion de l'inscription. La deuxième section modifie ces bases de données avec les notes des candidats (sans considérer la correction des copies). La troisième section vise la gestion des épreuves orales.

1 Bases de données pour l'inscription

Tous les candidats et candidates doivent s'inscrire sur un site web. Ils doivent fournir un certain nombre d'informations dont leur identité et les concours où ils sont candidats. Ce formulaire web permet alors de remplir la base de données dont nous donnons une version très simplifiée en figure 1. Les noms des tables sont sur fond grisé, les clés primaires sont en gras et les clés étrangères en italique.

La table Académie liste les académies de métropole et d'outre-mer avec d'une part une clé primaire **aid** qui est un code interne et d'autre part le nom usuel de l'académie dans le champs « nom ». La table Candidat reprend les informations personnelles du candidat. Le formulaire web force le genre à être 'H' ou 'F'.

La table Aménagement liste les aménagements possibles lors des épreuves avec d'une part une clé primaire **amid** et une description textuelle comme 'Tiers-temps' ou 'Salle isolée'. Selon le ou les handicaps reconnus (qui n'apparaissent pas dans ces tables), un candidat peut avoir droit à un ou plusieurs aménagements. La table ListeAmén liste les aménagements de chaque candidat. Chaque candidat peut avoir plusieurs aménagements et chaque aménagement peut s'appliquer à plusieurs candidats. C'est donc la paire **{ *cid*, *amid* }** qui est la clé primaire de ListeAmén.

Pour l'instant, on ne considère que deux concours : le capes NSI et l'agrégation externe d'informatique. Pour chaque concours, nous avons une table avec les numéros des candidats qui souhaitent y concourir ainsi que l'académie où il passe ce concours (on pourrait envisager de passer les deux concours dans deux académies distinctes, mais chaque concours correspond à une unique académie).

Pour l'agrégation, le candidat a un choix sur la troisième épreuve, représenté par le champs *ep* qui vaut 'A' pour « Étude de cas informatique » et 'B' pour « Fondements de l'informatique ».

Les champs *aid* de AgrégExtInfo et CapesNSI sont des clés étrangères provenant de Académie. Les champs *cid* de ces deux tables sont des clés étrangères provenant de Candidat ainsi que la clé primaire de ces tables. Le champs *cid* de ListeAmén est également une clé étrangère provenant de Candidat. Le champs *amid* de ListeAmén est une clé étrangère provenant de Aménagement.

Tournez la page S.V.P.

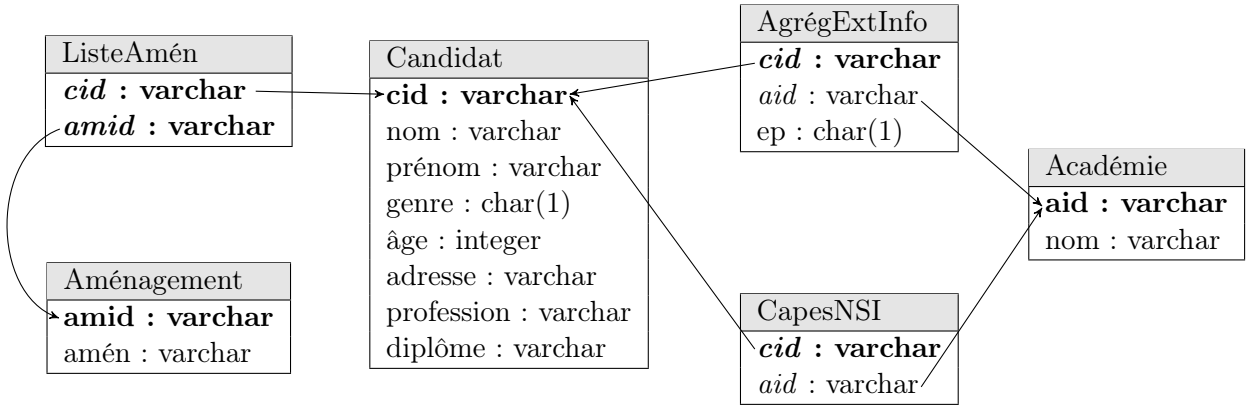


FIGURE 1 – Bases de données pour l'inscription

Question 3.1. Justifier brièvement l'intérêt d'avoir *cid* comme clé primaire de AgrégExtInfo.

Question 3.2. Écrire une requête SQL qui répond à : combien de candidats passent l'agrégation externe d'informatique dans chaque académie ? Le but est de savoir combien de sujets imprimer, on affichera en première colonne le nom de l'académie et en deuxième le nombre de candidats.

Question 3.3. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : pour les candidats qui passent les deux concours dans deux académies différentes, donner leur nom de famille et le nom de ces deux académies.

Question 3.4. Écrire une requête SQL qui répond à : pour les candidats à l'agrégation et pas au CAPES, donner le nombre de candidats par profession.

Question 3.5. Écrire une requête SQL qui répond à : donner la proportion de femmes qui choisissent l'épreuve spécifique « Étude de cas informatique » sur le nombre de candidats ayant choisi cette épreuve spécifique.

Question 3.6. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : donner les noms de famille des candidats ayant l'aménagement 'Salle isolée' mais pas l'aménagement 'Tiers-temps'.

Question 3.7. Écrire à la fois une requête SQL et une formule en algèbre relationnelle qui répondent à : donner les noms des académies où il n'y a pas de candidat ayant un aménagement 'Tiers-temps' pour l'agrégation.

Question 3.8. Écrire une requête SQL qui répond à : donner les noms des académies telles que cette académie a le nombre maximal d'aménagements 'Salle isolée' pour l'agrégation.

2 Bases de données pour les notes des épreuves écrites

On suppose que la correction se fait dans un autre outil (scan des copies, double correction de chaque copie, anonymat...) et que les notes sont ensuite basculées dans notre base de données initiale. On modifie alors AgrégExtInfo en

AgrégExtInfo
***cid* : varchar**
aid : varchar
ep : char(1)
note-ep1 : real
note-ep2 : real
note-ep3 : real

où note-epX est la note obtenue à l'épreuve X.

Question 3.9. Écrire une requête SQL qui répond à : faire afficher les noms des 42 candidats à l'agrégation ayant les meilleurs notes finales, triés par note finale décroissante. On rappelle que les trois épreuves ont le même coefficient, la note finale est donc la somme des notes des 3 épreuves écrites.

Question 3.10. Écrire une requête SQL qui répond à : calculer la moyenne de la note finale des candidats ayant au moins deux aménagements.

Question 3.11. Il est bien sûr simpliste de ne considérer que deux concours : il y a plus d'une centaine de concours, dont les CAPES, CAPET, agrégations externes, internes... Proposer une modélisation en tables qui permet d'avoir un nombre de concours non borné : vous indiquerez les tables et leur champs, ainsi que les clés primaires et étrangères.

Question 3.12. Écrire une requête SQL qui répond à : quel est le maximum du nombre de concours passé par une seule personne (dans votre modélisation) ?

3 Gestion des épreuves orales

Cette partie doit être traitée dans le langage Python.

Une fois les candidats admissibles, il faut organiser les épreuves orales et convoquer chaque admissible pour passer les trois épreuves orales. Une possibilité est d'avoir pour base un fichier `oral.csv` de ce type :

```
21/06, Leçon, 13h15, 13h30, 17h30, 18h30, A, 2845, Recoque, Alice
22/06, TP,    8h,    8h15,  13h15,  14h15,  C, 7743, Turing,  Alan
...
```


Tournez la page S.V.P.

Ainsi, Alice Recoque a pour numéro de candidate 2845 et elle passe l'épreuve de leçon le 21 juin. Elle est convoqué à 13h15 pour une préparation de 13h30 à 17h30 et une interrogation de 17h30 à 18h30 avec le jury A (chaque jury est représenté par une lettre). On rappelle que chaque candidat a 3 épreuves, leçon, modélisation et TP avec 4h ou 5h de préparation.

Le but est de vérifier que ce fichier csv ne comporte aucune erreur ou typo, car il est utilisé pour convoquer les candidats, organiser l'accueil des candidats et le travail du jury.

Pour un traitement en Python de ce fichier, on peut commencer par l'importer :

```
import csv
t_oral = []
with open('oral.csv', newline='') as csvfile:
    spamreader = csv.reader(csvfile, delimiter=',')
    for row in spamreader:
        t_oral.append(row)
```

qui donne alors le tableau de tableaux `t_oral` (temporaire) suivant :

```
[
  ['21/06', 'Leçon', '13h15', '13h30', '17h30', '18h30', 'A',
       '2845', 'Recoque', 'Alice']
  ['22/06', 'TP', '8h', '8h15', '13h15', '14h15', 'C', '7743',
       'Turing', 'Alan']
... ]
```

FIGURE 2 – Extrait de la valeur de la variable `t_oral` après appel au code Python précédent.

Le but est de programmer plusieurs fonctions de vérification. Chacune devra renvoyer un booléen indiquant si le fichier est correct ou non. Si non, elle devra écrire un message d'erreur explicatif.

Question 3.13. Écrire une fonction Python `verif_nblc(N,t_oral)` qui, à partir du nombre de candidat `N` et de `t_oral`, vérifie que le nombre de lignes de `t_oral` est correct, et pour chaque ligne, que son nombre de colonnes est correct.

Pour la suite des vérifications, une première étape consiste à « nettoyer » `t_oral` de façon à obtenir le tableau de tableaux `oral` suivant :

```
       0       1       2     3      4       5      6      7         8          9
 [
    [  21,  'Leçon',  795,  810,  1050,  1110,  'A',  2845,  'Recoque',  'Alice'  ]
    [  22,    'TP',   480,  495,   795,   855,  'C',  7743,  'Turing',   'Alan'   ]
    ...
 ]
```

FIGURE 3 – Extrait de la valeur de la variable `oral` après « nettoyage » de `t_oral`. On a indiqué à la première ligne en italique la numérotation des colonnes, qui n'apparaît pas dans `oral`.

Sachant que les oraux se déroulent en juin, la première colonne ne contient que le numéro (entier) du jour. Les colonnes d'horaires sont le nombres de minutes depuis 0h00 : ainsi '13h15' correspond à $13 \times 60 + 15 = 795$ minutes. Le numéro de candidat est transformé en un entier. Les colonnes restantes sont inchangées.

Question 3.14. Écrire une fonction Python `verif_temps(oral)` qui vérifie la cohérence des horaires, c'est-à-dire pour chaque ligne :

- il y a 15 minutes entre la convocation et le début de préparation,
- il y a 4h de préparation en modélisation et leçon et 5h en TP,
- il y a 1h d'interrogation.

Question 3.15. Écrire une fonction Python `verif_nbj(oral, nb_jury)` qui vérifie que pour chaque horaire de chaque jour, il y a moins de `nb_jury` candidats convoqués.

Question 3.16. Écrire une fonction Python `verif_ep(oral)` qui vérifie que chaque candidat a 3 épreuves qui sont 'TP', 'Leçon' et 'Modélisation' sur 3 jours différents.

Pour aider l'humain à lire le tableau, le fichier csv en entrée a en fait une première ligne explicative :

Jour,	**Ép**,	**Hconvoc**,	**Hdéb**,	**Hpass**,	**Hfin**,	**Jury**,	**cid**,	**Nom**,	**Prénom**
21/06,	Leçon,	13h15,	13h30,	17h30,	18h30,	A,	2845,	Recoque,	Alice
22/06,	TP,	8h,	8h15,	13h15,	14h15,	C,	7743,	Turing,	Alan
...									

On peut alors, en utilisant plutôt `csv.DictReader`, obtenir le tableau de dictionnaires `t_dict_oral` :

```
[
  {'Jour': '21/06', 'Ép': 'Leçon', 'Hconvoc': '13h15',
    'Hdéb': '13h30', 'Hpass': '17h30', 'Hfin': '18h30',
    'Jury': 'A', 'cid': '2845', 'Nom': 'Recoque',
    'Prénom': 'Alice'},
  {'Jour': '22/06', 'Ép': 'TP', 'Hconvoc': '8h',
    'Hdéb': '8h15', 'Hpass': '13h15', 'Hfin': '14h30',
    'Jury': 'C', 'cid': '7743', 'Nom': 'Turing',
    'Prénom': 'Alan'},
  ...
]
```

FIGURE 4 – Extrait de la valeur de la variable `t_dict_oral`.

On peut alors le « nettoyer » de façon similaire au passage de `t_oral` en figure 2 à `oral` en figure 3. On obtient le tableau de dictionnaires `dict_oral` suivant.

Tournez la page S.V.P.

```
[
  {'Jour': 21, 'Ép': 'Leçon', 'Hconvoc': 795, 'Hdéb':
    810, 'Hpass': 1050, 'Hfin': 1110,  'Jury': 'A',
    'cid': 2845, 'Nom': 'Recoque', 'Prénom': 'Alice'},

  {'Jour': 22, 'Ép': 'TP', 'Hconvoc': 480, 'Hdéb':
    495, 'Hpass': 795, 'Hfin': 855,  'Jury': 'C',
    'cid': 7743, 'Nom': 'Turing', 'Prénom': 'Alan'},
  ...
]
```

FIGURE 5 – Extrait de la valeur de la variable `dict_oral` après « nettoyage » de `t_dict_oral`.

Question 3.17. Écrire une fonction Python `dict_verif_nomno(dict_oral)`, qui vérifie que, dans la variable `dict_oral`, chaque numéro de candidat correspond bien à un seul nom de famille et prénom.

Question 3.18. Écrire une fonction Python `dict_verif_ep(dict_oral)` qui vérifie que chaque candidat a 3 épreuves qui sont 'TP', 'Leçon' et 'Modélisation' sur 3 jours différents.

Question 3.19. Donner trois avantages ou inconvénients de l'utilisation des dictionnaires par rapport à l'approche précédente.

Question 3.20. Donner trois avantages ou inconvénients de l'utilisation d'un fichier csv par rapport à une base de données pour la problématique de cette section.

Question 3.21. Écrire une fonction Python `clean_dict(t_dict_oral)` de « nettoyage » qui, à partir de la variable `t_dict_oral` de la figure 4, renvoie une valeur qui correspond à la figure 5. Elle doit également vérifier que la date et les horaires sont dans des intervalles possibles (entre 1 et 31 pour la date et entre 0h et 24h pour les horaires).

* *
*