DS n° 07 — corrigé

0. Au potager d'Aster

❑ α) Pas de difficulté.

SQL
```sql
SELECT nom, variete, description FROM Plantes WHERE famille = 'Astéracées'
```

❑ β) Pour connaître le nombre de plantes de chaque famille, il faut aussi implicitement renvoyer la famille associée, sinon on obtient la liste des nombres de plantes des familles, sans savoir à quelle famille cela se rattache.

SQL
```sql
SELECT famille, COUNT(id) FROM Plantes GROUP BY famille
```

❑ γ) Pour avoir les plantes dont le voisinage est favorable au Chou Rouge, il faut les plantes dont l'identifiant apparaît comme `id2` dans la relation `Favorable`, avec `id1` celui du Chou Rouge ;

SQL
```sql
SELECT P2.nom, P2.variete, P2.famille
FROM Plantes P2 JOIN Favorables F ON P2.id = F.id2
                JOIN Plantes P1 ON F.id1 = P1.id
WHERE P1.nom = 'Chou' AND P1.variete = 'Rouge'
```

❑ δ) On a uniquement besoin de travailler avec les identifiants. Il n'est pas nécessaire (il ne faut pas) refaire de jointure avec Plantes tant qu'on ne demande pas d'informations qui ne sont que dans Plantes. On cherche les identifiants qui sont défavorable à un identifiant favorable à l'identifiant 26.

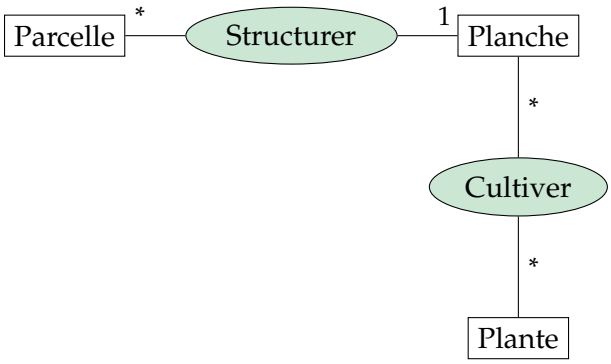
SQL
```sql
SELECT D.id2 AS id
FROM Defavorable D JOIN Favorable F ON D.id1 = F.id2
WHERE F.id1 = 36
```

❑ ε) On considère les plantes favorables au Chou Rouge et on fait une jointure à gauche avec la relation $\mathcal{R}$ de la question précédente. Il ne faut conserver que les plantes que l'on n'a pas réussi à associer lors de cette jointure, c'est-à-dire celle dont le champ `R.id` est **`NULL`**.

SQL
```sql
SELECT F.id2
FROM Favorable F LEFT JOIN R ON F.id2 = R.id
WHERE F.id1 = 36 AND R.id IS NULL
```

❑ ζ) La difficulté ici est de sélectionner ce qui va être important. On peut se faire une idée en lisant la suite des questions, afin d'être certaine ou certain que notre modèle conviendra pour écrire la requête de la dernière question. On reste simple et on propose un modèle minimal. Comme demandé, on ne perd pas de temps à indiquer les attributs des entités et associations. On n'a pas représenté ci-dessous les associations `Favorable` et `Defavorable` qui relient `Plante` à elle-même avec des cardinalités $\star$–$\star$.

- Une planche est un des morceaux qui structurent une parcelle.
- Une parcelle peut contenir plusieurs planches.
- Une plante peut être cultivée (semée) plusieurs fois. Une planche peut être occupée par la culture de plusieurs plantes.



❑ η) Ici il faut donner les clés primaires, tous les attributs, etc. Il n'était pas attendu une description aussi poussée que celle du début du sujet. Les attributs pour `Parcelles` et `Planches` sont directement obtenus à partir de la description donnée dans le sujet. Pour la relation `Cultures` on suppose que le semis et la récolte sont effectuées la même année. Les attributs `annee` et `semaine` de la relation `Cultures` doivent apparaître dans la clé primaire pour permettre de semer de nouveau la même plante sur la même planche.

```
Parcelles(numCadastre, surface, longitude, latitude, altitude, description)
Planches(num, #numParcelle, longueur, largeur)
Cultures(#idPlante, #numPlanche, annee, semaineSemis, semaineRecolte, masseRecolte)
```

- L'attribut `num` de la relation `Planches` est une clé étrangère qui référence l'attribut `numCadastre` de la relation `Parcelles`.
- L'attribut `numPlanche` de la relation `Cultures` est une clé étrangère qui référence l'attribut `num` de la relation `Planches`.
- L'attribut `idPlante` de la relation `Cultures` est une clé étrangère qui référence l'attribut `id` de la relation `Plantes`.

❑ θ) Il faut prendre garde à prendre en compte les plantes présentes sur la planche en semaine 23, donc semées mais pas encore récoltées.

SQL

```sql
SELECT nom, masseRecolte
FROM Plantes JOIN Culuture ON id = idPlante
WHERE numPlanche = 5750042 AND annee = 2023
      AND semaineSemis <= 23 AND semaineRecolte >= 23
```

CORRIGÉ

I Listes triées simplement chaînées

I.1 Autour des opérations de base

▸ **Question 1** On crée une fonction `cree_maillon` qui nous sera utile à de multiples reprises :

```
maillon_t *cree_maillon(int donnee, maillon_t *suivant) {
    maillon_t *maillon = malloc(sizeof(maillon_t));
    maillon->donnee = donnee;
    maillon->suivant = suivant;
    return maillon;
}
```

La fonction `init` s'écrit ensuite :

```
maillon_t *init(void) {
    maillon_t *premier = cree_maillon(INT_MIN, NULL);
    maillon_t *dernier = cree_maillon(INT_MAX, NULL);
    premier->suivant = dernier;
    return premier;
}
```

▸ **Question 2** La présence de sentinelles évite tous les cas particuliers :

```
maillon_t *localise(maillon_t *t, int v) {
    while (t->suivant->donnee < v) {
        t = t->suivant;
    }
    return t;
}
```

▸ **Question 3** On aimerait tester au moins les cas suivants :

Insertion dans une liste vide
- $v = 0$, liste initiale $\boxed{-\infty} \to \boxed{\infty}$
- valeur de retour `true`, liste $\boxed{-\infty} \to \boxed{0} \to \boxed{\infty}$

Nouvelle valeur en tête
- $v = 12$, liste initiale $\boxed{-\infty} \to \boxed{30} \to \boxed{40} \to \boxed{\infty}$
- valeur de retour `true`, liste $\boxed{-\infty} \to \boxed{12} \to \boxed{30} \to \boxed{40} \to \boxed{\infty}$

Nouvelle valeur au milieu
- $v = 35$, liste initiale $\boxed{-\infty} \to \boxed{30} \to \boxed{40} \to \boxed{\infty}$
- valeur de retour `true`, liste $\boxed{-\infty} \to \boxed{30} \to \boxed{35} \to \boxed{40} \to \boxed{\infty}$

Nouvelle valeur en queue
- $v = 50$, liste initiale $\boxed{-\infty} \to \boxed{30} \to \boxed{40} \to \boxed{\infty}$
- valeur de retour `true`, liste $\boxed{-\infty} \to \boxed{30} \to \boxed{40} \to \boxed{50} \to \boxed{\infty}$

Valeur déjà présente
- liste initiale $\boxed{-\infty} \to \boxed{30} \to \boxed{40} \to \boxed{50} \to \boxed{\infty}$, valeurs 30, 40, 50 (trois tests distincts)
- valeur de retour `false`, liste inchangée (pour chacun des trois tests).

▸ **Question 4** L'argument `t` est de type `maillon_t*`, donc `&t` est de type `maillon_t**`, or la fonction `localise` attend un `maillon_t*`. Il faut donc remplacer la ligne 7 par :

```
maillon_t *p = localise(t, v);
```

▸ **Question 5** La fonction proposée échoue aux trois derniers tests (ceux pour une valeur déjà présente). On peut la corriger ainsi :

```
bool insere(maillon_t *t, int v) {
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee == v) {return false;}
    maillon_t *n = malloc(sizeof(maillon_t));
    n->suivant = p->suivant;
    n->donnee = v;
    // ou simplement maillon_t *n = cree_maillon(v, p->suivant);
    p->suivant = n;
    return true;
}
```

▸ **Question 6** C'est très similaire à `insere` :

```
bool supprime(maillon_t *t, int v) {
    maillon_t *p = localise(t, v);
    if (p->suivant->donnee != v) {return false;}
    maillon_t *n = p->suivant;
    p->suivant = n->suivant;
    free(n);
    return true;
}
```

▸ **Question 7** La fonction `localise` prend un temps proportionnel au nombre de maillons parcourus, donc en $O(n)$, où n est la longueur de la liste. Les autres opérations effectuées par `insere` et `supprime` sont en temps constant, donc elles ont toutes deux une complexité en $O(n)$.

▸ **Question 8** Les trois régions possibles sont la pile, le tas, et la zone statique. La zone statique contient les variables globales : c'est donc là qu'est stockée la valeur 717 associée au `v` global. Au moment de l'appel `insere(t, v)`, la valeur 717 sera copiée sur la pile. Enfin, cette valeur sera recopiée dans le champ `n->donnee` d'un maillon alloué sur le tas et donc aura également une occurrence dans le tas.

I.2 Extensions des opérations de base

▸ **Question 9** Un arbre binaire de recherche est un arbre binaire dont les étiquettes sont choisies dans un ensemble totalement ordonné $(E, \leqslant)$ et tel que chaque nœud (g, x, d) vérifie $\max g < x < \min d$ (en convenant que $\max \emptyset < x < \min \emptyset$ pour tout x de E).
L'arbre doit être *équilibré* si l'on veut que les opérations élémentaires soient en temps logarithmique ; les arbres rouge-noir sont un exemple d'arbres auto-équilibrants.

▸ **Question 10** Si l'on souhaite être efficace, il faut insérer les éléments par ordre décroissant dans une liste initialement vide (puisque le coût de l'insertion est proportionnel au nombre de maillons à parcourir). Une manière de procéder (Insere et Init correspondent aux fonctions déjà écrites) :

Algorithme 1 Conversion ABR vers liste triée

fonction AjouteArbre(arbre, liste)
  si arbre $\neq \perp$ **alors**
    AjouteArbre(arbre.droite, liste)
    Insere(arbre.racine, liste)
    AjouteArbre(arbre.gauche, liste)

fonction ConvertitArbre(arbre)
  liste $\leftarrow$ Init()
  AjouteArbre(arbre, liste)
  renvoyer liste

Chaque insertion se fait en tête de la liste vu l'ordre du parcours, on effectue donc un simple parcours d'arbre avec travail constant à chaque nœud : la complexité est en $O(n)$ en temps. En mémoire, on a une complexité en $O(h)$ (hauteur de l'arbre) sur la pile à cause de la profondeur d'appel, et $O(1)$ sur le tas si l'on ne compte pas la taille de la liste renvoyée ($O(n)$ sinon).

▸ **Question 11** Notons A_i l'événement : « le i-ème tirage donne 1 » et X la variable aléatoire donnant le numéro de l'élément choisi dans la liste (on numérote les éléments de 1 à n). On a :

- $\mathbb{P}(A_i) = 1/2$, et les A_i sont mutuellement indépendants ;
- $[X = 1] = A_1^c \cap \cdots \cap A_{n-1}^c$;
- $[X = i] = A_{i-1} \cap A_i^c \cap \cdots \cap A_{n-1}^c$ si $i \geqslant 2$

On en déduit :

$$\mathbb{P}(X = i) = \begin{cases} \frac{1}{2^{n+1-i}} & \text{si } i \geqslant 2 \\ \frac{1}{2^{n-1}} & \text{si } i = 1 \end{cases}$$

▸ **Question 12** Il suffit d'incrémenter z à chaque tour de boucle, et de modifier la condition pour remplacer l'ancien élément par le i-ème avec probabilité $\frac{1}{i}$.

```
int z = 2;
while (c->suivant->donnee != INT_MAX) {
  c = c->suivant;
  if (random() % z == 0) {
    ret = c->donnee;
  }
  z++;
}
```

L'analyse suivante n'est pas demandée, mais montrons que cela assure effectivement que chaque élément de l'ensemble est équiprobable. Notons T_i la valeur obtenue au i-ème tirage et X_i la variable aléatoire donnant le numéro de l'élément choisi dans la liste après i tirages. Les variables aléatoires T_i sont mutuellement indépendantes et $\mathbb{P}(T_i = 0) = \frac{1}{i+1}$. Montrons que l'on a alors l'invariant « après le i-ème tirage, pour tout $1 \leqslant k \leqslant i+1$, $\mathbb{P}(X_i = k) = \frac{1}{i+1}$ ». En effet, avant le premier tirage, donc après 0 tirages, on a bien $\mathbb{P}(X_0 = 1) = 1$. Pour $1 \leqslant i \leqslant n-1$, supposons l'invariant établi pour $i-1$ tirages et montrons sa conservation après i tirages. On a $[X_i = i+1] = [T_i = 0]$ et donc $\mathbb{P}(X_i = i+1) = \frac{1}{i+1}$. Pour $1 \leqslant k \leqslant i$, $[X_i = k] = [X_{i-1} = k] \cap [T_i \neq 0]$ et ces deux événements sont indépendants. Ainsi $\mathbb{P}(X_i = k) = \frac{1}{i} \times \frac{i}{i+1} = \frac{1}{i+1}$ et l'invariant est conservé. La variable aléatoire X_n, qui donne le numéro de l'élément renvoyé à la toute fin, suit donc bien une loi uniforme.

▸ **Question 13** En dehors du problème de la pertinence des tests d'égalité entre flottants (inévitable si l'on souhaite représenter des ensembles de flottants), il n'y a pas de difficulté particulière. On peut remplacer `INT_MIN` et `INT_MAX` par `-INFINITY` et `INFINITY`.

II Listes à enjambement

▸ **Question 14** Las Vegas.

▸ **Question 15** On commence par écrire une fonction créant un chaînon de donnée et hauteur fixées, avec tous ses pointeurs égaux à `NULL` :

```
chainon_t *cree_chainon(int v, int hauteur) {
    chainon_t *c = malloc(sizeof(chainon_t));
    c->donnee = v;
    c->hauteur = hauteur;
    c->suivants = malloc(hauteur * sizeof(chainon_t*));
    for (int i = 0; i < hauteur; i++) {
        c->suivants[i] = NULL;
    }
    return c;
}
```

On obtient ensuite :

```c
chainon_t *enjmb_init(void) {
    chainon_t *deb = cree_chainon(INT_MIN, 1);
    chainon_t *fin = cree_chainon(INT_MAX, 1);

    deb->suivants[0] = fin;

    return deb;
}
```

▸ **Question 16** On suit les pointeurs de la couche de niveau 0 en libérant les maillons, ainsi que le tableau `suivant`, au passage.

```c
void enjmb_detruit(chainon_t *t) {
  while (t != NULL) {
    chainon_t *s = t->suivant[0];
    free(t->suivants);
    free(t);
    t = s;
  }
}
```

▸ **Question 17** L'existence des espérances ne pose pas de problème. On remarque d'abord qu'on a :

$$\begin{aligned}\mathbb{P}(H_{max} = x) &\leqslant \sum_{i=1}^{n} \mathbb{P}(H_i = x)\\ &= \sum_{i=1}^{n} \frac{1}{2^x}\\ &= \frac{n}{2^x}\end{aligned}$$

En posant $p = \lfloor 3\log_2 n \rfloor + 1$, on a donc :

$$\begin{aligned}\mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n}) &= \sum_{x \geqslant p} x\mathbb{P}(H_{max} = x)\\ &\leqslant \sum_{x \geqslant p} x\frac{n}{2^x}\\ &= n\sum_{x \geqslant p} \frac{x}{2^x}\\ &= \frac{2n(p+1)}{2^p}\end{aligned}$$

Or $2n(p+1) \sim 6n\log_2 n$ et $2^p \geqslant 2^{3\log_2 n} = n^3$, donc $\mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n}) = O\left(\frac{\log n}{n^2}\right) = o(\log n)$.
D'autre part, on a $H_{max} \cdot \mathbb{1}_{A_n^c} \leqslant 3\log_2 n$, donc $\mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n^c}) \leqslant 3\log_2 n$. Finalement :

$$\begin{aligned}\mathbb{E}(H_{max}) &= \mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n} + H_{max} \cdot \mathbb{1}_{A_n^c})\\ &= \mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n}) + \mathbb{E}(H_{max} \cdot \mathbb{1}_{A_n^c})\\ &= o(\log n) + O(\log n)\\ &= O(\log n)\end{aligned}$$

▸ **Question 18** En dehors des tableaux `suivants`, l'espace utilisé est clairement proportionnel à n, disons $3n$. Pour ces tableaux :

- les deux sentinelles contribuent à hauteur de H_{max} chacune ;
- les autres maillons contribuent $\sum_{i=1}^{n} H_i$ au total.

En négligeant les facteurs de proportionnalité, on a donc

$$\begin{aligned}\mathbb{E}(S) &= \mathbb{E}\left(3n + 2H_{\max} + \sum_{i=1}^{n} H_i\right)\\ &= n + O(\log n) + \sum_{i=1}^{n} \underbrace{\mathbb{E}(H_i)}_{2}\\ &= 5n + O(\log n)\\ &= O(n)\end{aligned}$$

▸ **Question 19** On traduit l'algorithme de l'énoncé :

```
bool enjmb_contient(chainon_t *t, int v) {
    int h = t->hauteur - 1;
    while (t->donnee != INT_MAX) {
        if (t->donnee == v) {return true;}
        while (h >= 0 && t->suivants[h]->donnee > v) {
            h--;
        }
        if (h < 0) {return false};
        t = t->suivants[h];
    }
    return false;
}
```

▸ **Question 20** Considérons le chemin suivi pour arriver au maillon m cherché, à l'envers.

- Tant que m est de hauteur 1, on suit un lien de niveau 0 et l'on remplace m par son voisin de gauche : comme $\mathbb{P}(H_i = 1) = 1/2$ et que les H_i sont indépendantes, on a bien une loi géométrique tronquée, avec Y la position de m dans la liste.
- Ensuite, on ne suivra plus de liens de niveau 0 : on élimine donc tous les maillons de niveau 0. Tant que le maillon m actuel est de hauteur 2, on suit un lien de niveau 1 et l'on remplace m par son voisin de gauche (dans la liste sans chaînon de niveau 0). Comme $\mathbb{P}(H_i = 2 \mid H_i \geqslant 1) = \frac{1}{2}$, on a à nouveau une loi géométrique tronquée.
- On continue, en commençant par éliminer les maillons de hauteur 2 et en se déplaçant vers la gauche tant qu'on ne peut pas monter...

Les R_h suivent donc des lois géométriques tronquées, et leurs espérances sont toutes inférieures ou égales à 1. Ensuite, en notant encore $p = \lfloor 3\log_2 n \rfloor + 1$:

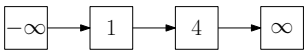
- $R' \cdot \mathbb{1}_{A_n^c} \leqslant R_0 + \cdots + R_{p-2}$, car le niveau maximal est $p-1$, d'où $\mathbb{E}(R' \cdot \mathbb{1}_{A_n^c}) \leqslant p - 1 = O(\log n)$;
- $R' \cdot \mathbb{1}_{A_n} \leqslant n\mathbb{1}_{A_n}$, d'où $\mathbb{E}(R' \cdot \mathbb{1}_{A_n}) \leqslant n\mathbb{E}(\mathbb{1}_{A_n}) = n\mathbb{P}(\mathbb{1}_{A_n})$. Or $\mathbb{P}(\mathbb{1}_{A_n}) = \mathbb{P}(H_{\max} \geqslant p) \leqslant \frac{2n}{2^p} = O\left(\frac{1}{n^2}\right)$ en sommant les majorations de $\mathbb{P}(H_{\max} = x)$ de la question 17, donc $\mathbb{E}(R' \cdot \mathbb{1}_{A_n}) = O(\frac{1}{n})$.
- On a donc $\mathbb{E}(R) = 1 + \mathbb{E}(H_{\max}) + \mathbb{E}(R' \cdot \mathbb{1}_{A_n^c}) + \mathbb{E}(R' \cdot \mathbb{1}_{A_n}) = O(\log n)$.

▸ **Question 21** La complexité du test d'appartenance est proportionnelle à R, donc d'espérance $O(\log n)$. Le point important est que cette moyenne ne dépend que des résultats des tirages aléatoires réalisés, et pas des données insérées.
Pour un ABR, on a du $O(\log n)$ dans le pire cas si l'arbre est auto-équilibrant (rouge-noir, par exemple). Dans le cas d'un ABR « basique », en revanche, l'insertion successive de $1, \ldots, n$ dans un arbre initialement vide produira un arbre dont la profondeur moyenne des nœuds sera de l'ordre de n : il en ira de même de la complexité en moyenne de la recherche.
Les garanties de complexité sont donc meilleures ici que pour un ABR classique, mais moins bonnes que pour un ABR auto-équilibrant.
Remarque
Si l'on insère les entiers de 1 à n *dans un ordre aléatoire*, on peut montrer que l'espérance de la hauteur de l'ABR obtenu est en $O(\log n)$. Cependant, cela n'a d'intérêt que si les données sont aléatoires, ce qui n'a aucune raison d'être le cas (et qui n'a pas besoin d'être supposé pour une *skip list*).

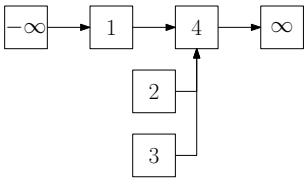
III Utilisation concurrente de listes chaînées

▸ **Question 22**

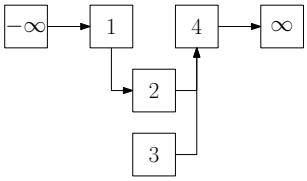
Considérons deux appels concurrents `insere(t, 2)` et `insere(t, 3)` dans cette liste :



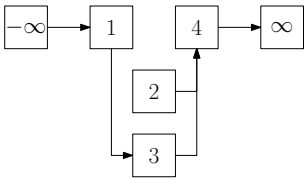
Les deux fils exécutent les lignes 7 à 11 (avec un entrelacement quelconque), on arrive alors dans cette situation :



Le premier appel exécute la ligne 11 (puis termine) :



Le deuxième appel en fait de même :



L'invariant numéro 3 est clairement violé : le maillon de valeur 2 n'est pas accessible (on aura d'ailleurs une fuite de mémoire).

III.1 Exclusion mutuelle à gros grains

▸ **Question 23**

```c
bool grosgrain_insere(liste_grosgrain_t *t, int v) {
    pthread_mutex_lock(&t->verrou);
    bool b = insere(t->tete, v);
    pthread_mutex_unlock(&t->verrou);
    return b;
}
```

▸ **Question 24** Avec le prototype alternatif, la structure est copiée pour être passée. On effectue donc en particulier une copie du verrou, et chaque appel à `grosgrain_insere` travaille en fait avec son propre verrou : cela équivaut à une absence de verrou.

▸ **Question 25** Il n'y a qu'un seul verrou, donc pas d'interblocage possible (et toutes les boucles sont bornées) : le code termine systématiquement.
Notons $+i$ l'appel (effectué une unique fois) `grosgrain_insere(l_partage, i)` et $-i$ l'appel (également unique) `grosgrain_supprime(l_partage, i)`. La seule contrainte sur les traces d'exécution est que $+i$ précède $-(i+1)\%n$.

- Pour $0 \leqslant l < n$, on considère la trace suivante (les fils 0 à $l-1$ s'exécutent successivement et entièrement, puis les $n-l$ autres insertions, puis les $n-l$ autres suppressions) :

$$\underbrace{+0,-1}_{T_0},\underbrace{+1,-2}_{T_1},\dots,\underbrace{+l-1,-l}_{T_{l-1}},\underbrace{+l}_{T_l},\dots,\underbrace{+(n-2)}_{T_{n-2}},\underbrace{+(n-1)}_{T_{n-1}},\underbrace{-(l+1)}_{T_l},\dots,\underbrace{-(n-1)}_{T_{n-2}},\underbrace{-0}_{T_{n-1}}$$

  Cette trace est bien valide et les éléments présents dans l'ensemble à la fin sont exactement $1,\dots,l$: il y en a l (correct y compris quand $l=0$).
- Il n'est pas possible d'obtenir n éléments dans l'ensemble final : le dernier appel est nécessairement une suppression, et l'élément ainsi supprimé ne peut faire partie du résultat.

On peut donc obtenir entre 0 et $n-1$ éléments dans l'ensemble, mais pas n.

▶ **Question 26** Notons $T_i : s(j)$ pour « le fil i exécute (avec succès) l'appel `grosgrain_supprime(j)`. Un début de trace possible est :

- $T_0 : s(0)$
- $T_1 : s(1)$
- ...
- $T_{n-1} : s(n-1)$

La liste est désormais vide, et tous les fils tournent en boucle en tentant de supprimer un élément inexistant. Le code ne termine donc pas nécessairement.
Dans le cas où l'exécution termine, on remarque que chaque suppression d'un élément est nécessairement suivie (dans le même fil) de la réinsertion de ce même élément : l'ensemble final contiendra donc tous les entiers de 0 à $n-1$.

III.2 Exclusion mutuelle à grains fins

▶ **Question 27**

```
maillon_protege_t *grainfin_localise(maillon_protege_t *t, int v) {
    pthread_mutex_lock(&t->verrou);
    while (true) {
        pthread_mutex_lock(&t->suivant->verrou);
        if (t->suivant->donnee >= v) {
            pthread_mutex_unlock(&t->suivant->verrou);
            return t;
        }
        pthread_mutex_unlock(&t->verrou);
        t = t->suivant;
    }
}
```

▶ **Question 28** Considérons le fil possédant le verrou le plus à droite : s'il cherche à acquérir un verrou, il s'agit nécessairement du suivant (dans l'ordre de la liste), qui est nécessairement libre. Il ne peut donc être bloqué : il n'y a pas d'interblocage.

▶ **Question 29** Soit m_i le maillon localisé par le fil T_1 pour lequel l'appel `grainfin_localise` se termine en premier. Ce maillon a été verrouillé par t_1 et ne sera déverrouillé qu'à la fin de l'exécution de T_1. Le fil T_2 :

- soit est contraint à rester strictement à gauche de m_i jusqu'à la fin de l'exécution de T_1, et ne peut même pas localiser m_{i-1} (il faudrait temporairement acquérir le verrou de m_i). Dans ce cas, soit T_2 effectuera une insertion avant m_{i-1}, complètement indépendante de celle de T_1, soit il sera bloqué jusqu'à la fin de l'exécution de T_1 et insérera dans la nouvelle liste ;
- soit est déjà strictement à droite de m_i, et y reste. Dans ce cas, l'insertion est indépendante de celle de T_1 : T_1 insère entre m_i et m_{i+1}, T_2 entre m_j et m_{j+1} avec $j \geqslant i+1$ (et même $j \geqslant i+2$ en fait).

Dans tous les cas, les invariants sont respectés.

▶ **Question 30** Commençons par remarquer qu'il semble impossible d'écrire un crible d'Erathosthène, parallèle ou pas, en utilisant des listes chaînées, tout au moins si l'on considère que l'une des caractéristiques fondamentales de cet algorithme est qu'il n'effectue aucune division.
Une version avec listes à gros grains ne présente visiblement aucun intérêt : un seul fil pouvant s'exécuter à la fois, autant écrire du code *singlethread*.
Dans la version à petits grains, il est assez facile de s'assurer que le fil principal ne rencontre que des nombres premiers. En effet, il est impossible qu'un fil en « dépasse » un autre lors du parcours (si l'on respecte la règle énoncée en début de partie). Il suffit donc de s'assurer que le fil T_p de suppression des multiples de p verrouille p^2 avant que le fil principal n'y arrive. Pour cela, le fil principal peut commencer par verrouiller le successeur de p avant de lancer T_p, et charger ce dernier de libérer le successeur de p après avoir verrouillé p^2.
Les fils ne vont absolument pas se répartir les nombre composés de manière équitable : le fil T_2 éliminera la moitié des nombres, T_{17} en éliminera (nettement) moins de 1/17. On pourrait croire que T_{17} peut arriver sur un multiple de 34 avant T_2, puisqu'il est censé commencer à 17×17 au moment où le fil principal voit 17 (ce qui lui permettrait de dépasser le fil T_2). C'est en fait impossible, le fil principal ne pouvant lancer T_{17} que sur un

nœud qui lui est immédiatement accessible (sur le successeur de 17, donc). Accessoirement, on va créer environ $n/\log n$ fils, ce qui est démesuré et inefficace.
Enfin, on peut remarquer que dans un « vrai » crible, l'élimination des multiples de p parcourt environ n/p cases ($(n-p^2)/p$, plus précisément). Ici, le fil T_p parcourra tous les chaînons situés à droite de p et non encore éliminés : il y en a nettement plus (en particulier, tous les nombres premiers entre p et n seront nécessairement parcourus). La possibilité de « sauter » des cases est fondamentale dans le crible. Comme, de plus, le coût du parcours d'un chaînon va ici être dominé par celui de l'acquisition et de la libération du verrou associé, on peut s'attendre à des performances catastrophiques.