CORRIGÉ

Remarques

Remarques générales

- Micro-détail : il vaut mieux déclarer une fonction `int f(void)` plutôt que `int f()` (en C, la deuxième version déclare une fonction prenant un nombre quelconque d'arguments de type quelconque).
- Une question demandant d'écrire une fonction qui libère une liste chaînée serait sans doute bienvenue : c'est un point important de la programmation en C, et cela rajouterai une question technique et assez facile à un sujet qui en manque peut-être un peu. Il faut classer entre eux les élèves visant MinesTelecom, en particulier...
- Dans la définition d'une loi géométrique tronquée, il me semble que Y doit être à valeurs dans $\mathbb{N}^*$.
- Les questions de programmation semblent assez simples, surtout comparées aux questions théoriques, dont plusieurs me semblent vraiment délicates.
- Il serait peut-être souhaitable de détailler un peu comment se passe l'insertion d'un élément dans une *skip list* (en précisant qu'on fait en sorte que les invariants de liste triée restent vérifiés pour chaque niveau), et d'expliciter le fait que l'on laisse ce problème de côté.

Point par point

Question 3 Difficile de savoir ce qui est attendu : un test renvoyant `true` et un renvoyant `false` suffisent-ils à obtenir tous les points ?

Question 13 Je ne vois pas vraiment ce qui est attendu : j'imagine que ce n'est pas l'échec de la recherche de `NAN`... Si le problème est plus généralement qu'il ne « faut pas » faire de tests d'égalité entre flottants, il est lié à la notion même d'ensemble de flottants et pas spécifiquement à la structure considérée. La connaissance de `INFINITY` ne me semble pas exigible.

Question 16 Il est fort probable que j'ai fait n'importe quoi, mais je ne vois pas l'intérêt de considérer $3\log_2 n$. J'ai l'impression que $2\log_2 n$ suffirait pour obtenir le petit-o, et $\log_2 n$ pour obtenir le grand-O qui permet déjà de conclure.

Question 26 Le pseudo-code me semble peu approprié ici. Par ailleurs, il y a une petite ambiguïté : faut-il acquérir le verrou de `t->suivant` avant d'accéder au champ `t->suivant->donnee` ? J'ai considéré que oui, mais il semble que la question 29 considère que non.

Question 28 Ma réponse ne me satisfait absolument pas, mais je trouve ce genre de questions extrêmement difficile à rédiger, sans doute par manque d'expérience.

Question 29 Il me semble qu'il n'est pas nécessaire de conserver le verrou sur le successeur pendant la suppression (se référer au corrigé), et donc que la question n'est pas vraiment correcte. Honnêtement je n'en suis pas complètement sûr, alors que j'y ai passé du temps et que j'en ai discuté avec un ami qui est plus spécialiste que moi : il me semble que cela montre que ce genre d'analyse, même sur une structure de donnée simple, est presque toujours très délicate. Si ce sujet avait été posé au concours, la correction de cette question aurait été extrêmement difficile à mon avis (pour les rares candidats qui seraient arrivés jusque là).

Question 30 Petite erreur : il faut commencer la suppression des multiples à p^2, pas à $p(p+1)$. Par ailleurs, la question me laisse pour le moins perplexe : la parallélisation du crible, tout comme le crible lui-même, n'a d'intérêt que si l'on utilise un tableau, sauf si j'ai vraiment raté quelque chose. J'ai mis des commentaires plus détaillés dans le corrigé de la question.

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 :

```c
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` :

```c
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.

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 ;
- $[X = i] = A_i \cap A_{i+1}^c \cap \cdots \cap A_n^c$

On en déduit :

$$\mathbb{P}(X = i) = \frac{1}{2^{n+1-i}}$$

▶ **Question 12** La solution la plus simple est de choisir directement le numéro de l'élément à renvoyer uniformément dans $[1 \dots n]$, en considérant que la sentinelle initiale porte le numéro zéro.

```
int longueur(maillon_t *t){
   int n = 0;
   while (t->suivant->donnee != INT_MAX) {
      n++;
      t = t->suivant;
   }
   return n;
}
```

```
int random_elt(maillon_t *t){
    int n = longueur(t);
    if (n == 0) assert(false);
    int x = 1 + random()%n;
    for (int i = 0; i < x; i++){
        t = t->suivant;
    }
    return t->donnee;
}
```

▶ **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 :

```
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** 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 17** En dehors des tableaux `suivants`, l'espace utilisé est clairement proportionnel à n. 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

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

▶ **Question 18** On traduit l'algorithme de l'énoncé :

```
bool enjmb_contient(chainon_t *t, int v){
    int h = t->hauteur;
    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 19** 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 $\mathbb{P}(H_i = 0) = 1/2$ 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 $\mathbb{P}(H_i = 1 \mid H_i \geqslant 1) = \frac{1}{2}$, 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 $p = \lfloor 3\log_2 n \rfloor + 1$:

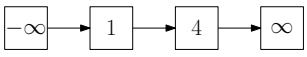
- $R' \cdot \mathbb{1}_{A_n^c} \leqslant R_0 + \cdots + R_p$, d'où $\mathbb{E}(R' \cdot \mathbb{1}_{A_n^c}) \leqslant p + 1 = O(\log n)$;
- $R' \cdot \mathbb{1}_{A_n} \leqslant n\mathbb{1}_{A_n}$, d'où $\mathbb{E}(R' \cdot \mathbb{1}_{A_n}) \leqslant n\mathbb{E}(\mathbb{1}_{A_n}) = n\mathbb{P}(\mathbb{1}_{A_n})$. Or $\mathbb{P}(\mathbb{1}_{A_n}) = \mathbb{P}(H_{\max} \geqslant p) \leqslant \frac{2(p+1)}{2^p} = O\left(\frac{\log n}{n^2}\right)$ en reprenant les calculs de la question 16, donc $\mathbb{E}(R' \cdot \mathbb{1}_{A_n}) = o(\log n)$.
- 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 20** 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*).

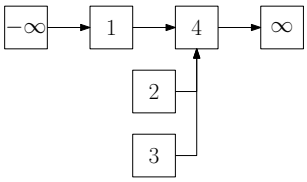
III Utilisation concurrente de listes chaînées

▸ **Question 21**

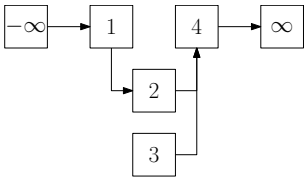
Considérons deux appels concurrents `insere(t, 2)` et `insere(t, 3)` dans cette liste :



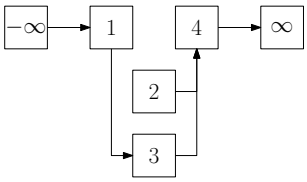
Les deux fils exécutent les lignes 7 à 11 (avec un entrelacement quelconque), on arrive alors dans cette situation :



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 22**

```
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 23** 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 24** 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 25** 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 26**

```
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 27** 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 28** 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 29** Le maillon à supprimer est le successeur `p` de celui (`c`) renvoyé par `grainfin_localise` : il est clair que si l'on effectue le `free` alors que l'autre fil est encore en train de manipuler `p` (par exemple, s'il est encore dans la fonction `grainfin_localise` avec la variable `t` égale à `p`), il y aura un problème (*use after free*, par exemple). De plus, si l'on essaie de supprimer de manière concurrente deux chaînons successifs, il est possible qu'un seul soit effectivement supprimé (si l'on commence par lire les deux champs `suivant` puis que l'on effectue les deux suppressions) : *a priori*, on aurait finalement un *double free* (deux appels à `free` sont exécutés, et dans ce cas c'est sur le même chaînon).

Remarque

Il me semble toutefois que cette situation ne peut pas se produire avec la fonction `grainfin_localise` écrite plus haut. En effet, pour renvoyer `c`, il a fallu lire la donnée de son successeur `p`, et pour ce faire acquérir le verrou sur `p`. Cela signifie que l'on possédait à un certain instant simultanément les verrous sur `p` et `c`, et donc que l'autre fil était soit strictement à droite de `p` (et l'est toujours), soit strictement à gauche de `c` (et l'est toujours, le verrou sur `c` n'ayant pas été libéré). Il y aurait en revanche un problème si l'autre fil pouvait revenir vers la gauche (s'il avait par exemple gardé un pointeur vers `p` et acquérait le verrou de ce chaînon une fois celui-ci libéré par le premier fil).

► **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.