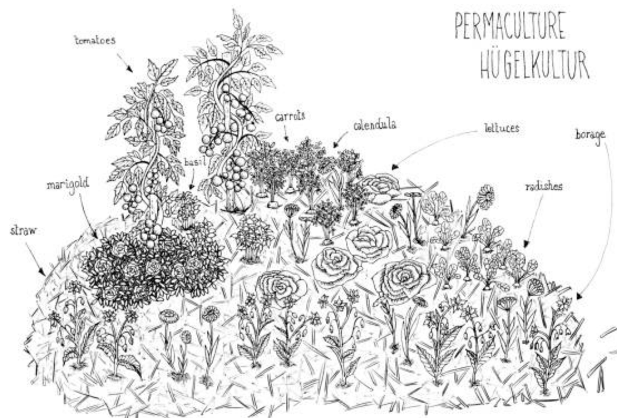
# DS nº 07 — 4h

Le sujet est composé de deux parties indépendantes : un premier exercice portant sur les bases de données, puis le problème du sujet 0 MPI de la banque Mines-Ponts.

## 0. Au potager d'Aster




Le potager d'Aster est un (fictif mais plausible) maraîchage solidaire en permaculture et en agriculture biologique installé depuis plusieurs années sur quelques parcelles d'une commune des balcons de Belledonne.

Pour tenir compte des associations favorables et défavorables entre les légumes et autres plantes installées au potager, une base de données a été créée récemment. Le modèle relationnel utilisé est donné ci-dessous :

- `Plantes(id, nom, variete, famille, description)`
  $\{(i, n, v, f, d) \in$ `Plantes` $\iff$ le légume d'identifiant $i$ a pour nom $n$, sa variété est $v$, sa famille est $f$ et sa description est $d$ $\}$.
- `Favorable(#id1, #id2)`
  $\{(i_1, i_2) \in$ `Favorable` $\iff i_2$ est favorable à $i_1\}$. Ainsi le voisinage de $i_2$ est favorable à $i_1$. Cela n'implique pas nécessairement que $i_1$ soit favorable à $i_2$.
- `Defavorable(#id1, #id2)`
  $\{(i_1, i_2) \in$ `Defavorable` $\iff i_2$ est défavorable à $i_1\}$. Ainsi le voisinage de $i_2$ est défavorable à $i_1$. À nouveau, cela n'implique pas que $i_1$ est ou non défavorable pour $i_2$.

Les attributs `id1` et `id2` des relations `Favorable` et `Defavorable` sont des clés étrangères vers la clé primaire `id` de la relation `Plantes`.

*Comme pour les programmes en C ou en OCAML, une requête non triviale doit être accompagnée d'une phrase simple permettant de mettre en valeur l'idée principale et toute astuce utilisée. Une requête qui ne m'apparaît pas limpide instantanément et qui n'est pas succinctement expliquée ne sera pas lue.*

❏ $\alpha$) Écrire une requête en langage `SQL` qui permet d'obtenir le nom, la variété et la description de toutes les plantes de la famille `'Astéracées'`.

❏ $\beta$) Écrire une requête en langage `SQL` qui permet de connaître le nombre de plantes de chaque famille.

❏ $\gamma$) Écrire une requête en langage `SQL` qui permet d'obtenir le nom, la variété et la famille de toutes les plantes dont le voisinage est favorable au légume dont le nom est `'Chou'` et la variété `'Rouge'`. *Remarque : un légume est une plante.*

Cela arrive qu'une plante $f_1$ favorable pour la plante $p$ qui nous intéresse soit défavorable pour une plante $f_2$ qui est pourtant elle-même favorable pour $p$. Dans ce cas, il y aurait au moins l'entrée $(f_2, f_1)$ dans la relation `Defavorable` et les entrées $(p, f_1), (p, f_2)$ dans la relation `Favorable`.

❏ $\delta$) On suppose que l'identifiant du Chou Rouge dans cette base de données est 26. Écrire une requête en langage `SQL` qui permet de déterminer les identifiants des plantes qui sont défavorables à au moins une plante qui est favorable au Chou Rouge. *On veillera à ne pas effectuer de jointures inutiles.*

❏ $\varepsilon$) En utilisant une jointure à gauche, écrire une requête en langage `SQL` permettant d'obtenir les identifiants des plantes qui sont favorables au Chou Rouge sans être défavorables à une des plantes favorables au Chou Rouge. On pourra utiliser sans modification la requête écrite à la question précédente, que l'on pourra désigner par $\mathcal{R}$ avec un alias `R`. *Cette requête doit être écrite avec une jointure à gauche et sans opérateurs ensemblistes.*

Vous voilà ravie ou ravi ! Il semble que salsifis, betteraves, concombres, haricots et autres cornichons pourront être installés dans le voisinage du Chou Rouge. C'est une excellente nouvelle... seulement où et quand ces légumes devront-ils ou pourront-ils être semés ?

Au potager d'Aster, les maraîchères et maraîchers disposent de plusieurs parcelles qui sont identifiées par leur numéro de parcelle sur le cadastre de la commune, numéro représenté par une chaîne de caractères, par exemple `0E0575`. Pour chaque parcelle, sa surface, son altitude, sa latitude, sa longitude et une description sont connues. Les parcelles cultivées sont arrangées autour d'une mare et se situent pour la plupart à proximité d'une parcelle de forêt. Chaque parcelle est découpée en plusieurs planches, nom donné à une bande de terre qui héberge effectivement des cultures volontaires. C'est l'unité de terrain sur laquelle sont placées les plantes d'une même association ou une plante en particulier en cas de monoculture ce qui est assez rare au potager d'Aster. Pour chaque planche, la longueur et la largeur sont connues. De plus, à chaque planche a été attribué un numéro entier unique parmi toutes les planches quelle que soit leur parcelle d'implantation.

L'objectif des maraîchers est de faire évoluer leur base de données afin de pouvoir conserver les informations relatives aux cultures réalisées d'une année sur l'autre, de suivre l'occupation des planches, de prévoir les rotations des familles de plantes installées et petit à petit de pouvoir repérer ce qui fonctionne le mieux. La culture d'une plante peut être réalisée sur une même planche plusieurs semaines consécutives d'une même année. Dans chaque cas cela donne, si tout se passe bien, une récolte lors d'une semaine et d'une année. Les récoltes obtenues sont pesées et la masse peut être conservée.

❏ $\zeta$) En sélectionnant les informations importantes du texte précédent, proposer un schéma entité/association, avec cardinalités des associations, susceptible de répondre aux nouvelles problématiques soulevées. Sur ce schéma, on ne demande pas de représenter les éventuels attributs des entités. Le schéma doit être accompagné d'explications. *Remarque : il convient de loucher sur la suite du sujet ici et de toujours lire quelques questions en avance. Le schéma relationnel obtenu à la question suivante, qui découle du modèle entité/association de cette question, doit permettre de pouvoir répondre à la toute dernière question.*

❏ $\eta$) En déduire un modèle relationnel complet, sur lequel les clés primaires seront soulignées. S'il y a lieu, les clés étrangères seront visualisées avec un caractère # et une phrase précisera le lien représenté. Les domaines des attributs ne seront pas précisés. Les relations `Plantes`, `Favorable` et `Defavorable` ne seront pas modifiées et ne doivent pas être ré-écrites.

❏ $\theta$) Écrire une requête en langage `SQL` qui permet de déterminer le nom des plantes qui étaient présentes sur la planche `5750042` en semaine 23 de l'année 2023 et la masse récoltée pour chaque plante à l'issue de cette culture.

# Préliminaires

## Présentation du sujet

L'épreuve est composée d'un problème unique comportant 30 questions. Dans ce problème nous étudions différentes variantes d'implémentation du type de données abstrait EnsembleEntiers, qui permet de stocker une collection d'entiers. Ce type abstrait est muni de trois primitives : l'insertion, la suppression et le test d'appartenance.

Le problème est divisé en trois sections. Dans la première section (page 2), nous implémentons EnsembleEntiers à l'aide de listes chaînées. Dans la deuxième section (page 5), indépendante de la première, nous améliorons le fonctionnement de la structure en passant à des listes à enjambements. Dans la troisième section (page 8), qui prolonge la première, nous étudions le comportement de listes chaînées dans des situations de concurrence.

Dans tout l'énoncé, un même identificateur écrit dans deux polices de caractère différentes désignera la même entité, mais du point de vue mathématique avec la police en italique (par exemple $n$ ou $n'$) et du point de vue informatique avec celle en romain avec espacement fixe (par exemple `n` ou `nprime`).

## Travail attendu

Pour répondre à une question, il est permis de réutiliser le résultat d'une question antérieure, même sans avoir réussi à établir ce résultat. Des rappels de mathématique et de programmation sont faits en annexe et peuvent être utilisés directement.

Selon les questions, il faudra coder des fonctions à l'aide du langage de programmation C exclusivement, en reprenant le prototype de fonction fourni par le sujet, ou en pseudo-code (c-à-d. dans une syntaxe souple mais conforme aux possibilités offertes par le langage C).

Quand l'énoncé demande de coder une fonction, sauf demande explicite de l'énoncé, il n'est pas nécessaire de justifier que celle-ci est correcte ou de tester que des préconditions sont satisfaites.

Le barème tient compte de la clarté des programmes : nous recommandons de choisir des noms de variables intelligibles ou encore de structurer de longs codes par des blocs ou par des fonctions auxiliaires dont on décrit le rôle. Lorsqu'une réponse en pseudo-code est permise, seule la logique de programmation est évaluée, même dans le cas où un code en C a été fourni en guise de réponse.

# 1. Listes triées simplement chaînées

## 1.1. Autour des opérations de base

Nous supposons définies deux constantes entières `INT_MIN` et `INT_MAX` qui désignent respectivement le plus petit entier et le plus grand entier représentables par le type **`int`** en machine. Elles sont représentées par $-\infty$ et $\infty$ dans les schémas.

**Indication C :** Nous introduisons une structure de maillon constituée de deux champs par la déclaration suivante.

```
1.  struct maillon {
2.      int donnee;
3.      struct maillon *suivant;
4.  };
5.  typedef struct maillon maillon_t;
```

**Définition :** Dans l'ensemble de cette partie, nous réalisons le type abstrait EnsembleEntiers à l'aide d'une liste de maillons simplement chainés. Nous supposons que tous les entiers insérés, supprimés ou recherchés sont strictement compris entre `INT_MIN` et `INT_MAX`. Nous maintenons les trois invariants suivants :

✠ 1. Pour tous maillons $m$ et $m'$ consécutifs dans la liste chaînée, de champs `donnee` respectifs $u$ et $u'$, on a l'inégalité $u < u'$ (autrement dit, la liste est triée et ne contient pas de doublons).

✠ 2. La liste est encadrée par deux maillons sentinelles ayant `INT_MIN` comme champ `donnee` en tête de liste et `INT_MAX` en fin de liste.

✠ 3. Pour toute valeur entière $u$ contenue dans l'ensemble, il existe un maillon accessible depuis le maillon sentinelle de tête ayant $u$ comme champ `donnee`.

Par exemple, l'ensemble $\{2, -7, 9\}$ est représenté par la liste chaînée dessinée en figure 1 (où la croix représente le pointeur nul).

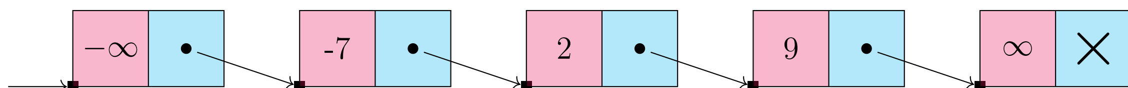



Figure 1 – Exemple d'ensemble d'entier représenté par une liste chaînée

☐ 1 – Écrire en C une fonction `maillon_t *init(void)` dont la spécification suit :
*Effet :* crée une copie de l'ensemble vide par l'instanciation de deux nouveaux maillons sentinelles chaînés entre eux.
*Valeur de retour :* pointeur vers le maillon de tête.

☐ 2 – Écrire en C une fonction `maillon_t *localise(maillon_t *t, int v)` dont la spécification suit :
*Précondition :* Le pointeur $t$ désigne le maillon sentinelle de tête d'une liste chaînée.
*Postcondition :* En notant $u$ le champ `donnee` du maillon désigné par la valeur de retour et $u'$ celui du maillon successeur, on a les inégalités $u < v \leqslant u'$.

Nous souhaitons écrire une fonction `bool insere(maillon_t *t, int v)` ainsi spécifiée :
*Précondition :* Le pointeur $t$ désigne le maillon sentinelle de tête d'une liste chaînée.
*Postcondition :* La liste désignée par le pointeur $t$ contient la valeur entière $v$ ainsi que les autres valeurs précédemment contenues.
*Valeur de retour :* Booléen `true` si la liste contient un élément de plus et `false` sinon.

☐ 3 – Présenter sous forme de croquis un jeu de données de test de la fonction `insere`, qui couvre notamment l'ensemble des valeurs de retour possibles. Dans chaque cas, on dessinera les états initial et final de la liste à la manière de la figure 1 et on donnera la valeur de retour.

Nous proposons le code erroné suivant.

```
6.  bool insere_errone(maillon_t *t, int v) {
7.      maillon_t *p = localise(&t, v);
8.      maillon_t *n = malloc(sizeof(maillon_t));
9.      n->suivant = p->suivant;
10.     n->donnee = v;
11.     p->suivant = n;
12.     return true;
13. }
```

☐ 4 – Le compilateur produit le message d'erreur : `incompatible pointer types passing 'maillon_t **' to parameter of type 'maillon_t *'`. Expliquer ce message et proposer une première correction.

☐ 5 – Discerner le ou les tests de la question 3 manqués par la fonction `insere_errone`. Corriger en conséquence la fonction `insere_errone`.

Dans ce qui suit, nous supposons que la fonction `bool insere(maillon_t *t, int v)` a été codée correctement.

☐ 6 – Écrire en C une fonction `bool supprime(maillon_t *t, int v)` dont la spécification suit :
*Précondition :* Le pointeur $t$ désigne le maillon sentinelle de tête d'une liste chaînée.
*Postcondition :* La liste désignée par le pointeur $t$ ne contient pas la valeur entière $v$ mais contient les autres valeurs précédemment contenues.
*Valeur de retour :* Booléen `true` si la liste contient un élément de moins et `false` sinon.

☐ 7 – Calculer la complexité en temps des fonctions `insere` et `supprime`.

☐ 8 – Un programme C peut stocker ses données dans différentes régions de la mémoire. Citer ces régions. Dire dans laquelle ou dans lesquelles de ces régions la valeur entière 717 est inscrite lorsque nous exécutons le programme suivant.

```
14. int v = 717;
15. int main(void) {
16.     maillon_t *t = init();
17.     insere(t, v);
18.     return 0;
19. }
```

## 1.2. Extensions des opérations de base

Nous disposons d'une implémentation alternative du type abstrait EnsembleEntiers par une structure de données d'arbre binaire de recherche.

☐ 9 – Définir le terme *arbre binaire de recherche*. Citer une propriété désirable afin que la complexité en temps des trois primitives (insertion, suppression et test d'appartenance) soit logarithmique. Nommer un exemple d'arbre binaire de recherche qui jouit de cette propriété.

☐ 10 – Décrire, en langue française ou par du pseudo-code, un algorithme aussi efficace que possible qui transforme un ensemble représenté sous forme d'arbre binaire de recherche en un ensemble représenté sous forme d'une liste chainée avec sentinelles. Donner sa complexité en temps et en espace.

Nous souhaitons compléter l'implémentation du type EnsembleEntiers de la section 1.1 par une primitive supplémentaire qui renvoie un élément aléatoirement choisi dans un ensemble non vide. Nous considérons que l'expression `random()%z` engendre un entier aléatoire choisi uniformément entre 0 et $z-1$ et évacuons toute préoccupation relative à la validité de $z$.

Dans une première ébauche, nous proposons le code suivant.

```
20.  int random_elt(maillon_t *t) {
21.      maillon_t *c = t->suivant;
22.      int ret = c->donnee;
23.      if (c->donnee == INT_MAX) {
24.          assert(false);
25.      }
26.      int z = 2;
27.      while ((c->suivant)->donnee != INT_MAX) {
28.          c = c->suivant;
29.          if (random()%z) {
30.              ret = c->donnee;
31.          }
32.      }
33.      return ret;
34.  }
```

☐ 11 – Décrire avec quelle probabilité l'expression `random_elt(t)` renvoie un élément $u$ lorsque le pointeur $t$ désigne un ensemble à $n$ éléments dont $u$ est le $i$-ième élément.

☐ 12 – Modifier, si besoin, la fonction `random_elt` afin qu'elle renvoie un élément distribué selon une loi de probabilité uniforme (i.e. chaque élément de l'ensemble est équiprobable).

Alternativement à ce qui précède et uniquement dans la question 13, nous envisageons de représenter des ensembles de flottants.

☐ 13 – Dire quelles difficultés supplémentaires il y aurait à manipuler des listes dont les données sont de type **`double`** plutôt que de type **`int`**.

# 2. Listes à enjambement

Afin d'accélérer l'opération de test d'appartenance, nous nous proposons d'analyser une implémentation alternative du type abstrait EnsembleEntiers par des *listes à enjambement*. Une liste à enjambement est une structure de données probabiliste, qui généralise les listes chaînées.

☐ 14 – À titre préliminaire, nommer une ville casino par laquelle est qualifié un algorithme dont le résultat est toujours correct et dont la complexité en temps est aléatoire.

**Définition :** Informellement, une liste à enjambement est une liste chaînée dont on a enrichi certains maillons par des pointeurs supplémentaires qui permettent d'avancer plus rapidement vers la fin de la liste lors de la recherche d'un élément (*cf.* exemple en figure 2). Mathématiquement, une liste à enjambement est une suite finie de listes chaînées $K = (K_h)_{0 \leqslant h < H}$ telle que pour tout indice $h$ compris entre 0 et $H-2$, la liste chaînée $K_{h+1}$ est une sous-chaîne de la liste chaînée $K_h$. La liste chaînée $K_h$ s'appelle la *couche de niveau h* de la liste à enjambement $K$. Informatiquement, nous ne stockons qu'une seule fois les éléments qui apparaissent dans plusieurs couches. Nous représentons une liste à enjambement à l'aide de *chaînons* qui comportent une valeur entière et un tableau, de longueur non nulle et variable en fonction du chaînon, contenant des pointeurs vers des chaînons de plus en plus éloignés.

**Indication C :** Nous adoptons la déclaration de type suivante

```
35.  struct chainon {
36.      int donnee;
37.      int hauteur;
38.      struct chainon **suivants;
39.  };
40.  typedef struct chainon chainon_t;
```

Le champ `hauteur` est utilisé pour noter la longueur du tableau de pointeur désigné par le champ `suivants`. Nous maintenons la présence de deux chaînons sentinelles, de valeur `INT_MIN` et `INT_MAX`, afin d'encadrer les éléments de la liste à enjambement. Les hauteurs des deux chaînons sentinelles sont égales à la plus grande hauteur des chaînons encadrés. Les chaînons sont triés par donnée croissante et sans doublon.

Pour déterminer la hauteur d'un chaînon non sentinelle, nous jouons à sa création plusieurs fois à pile ou face avec une pièce équilibrée. À chaque succès, nous augmentons la hauteur du chaînon d'une unité ; au premier échec, nous cessons notre jeu. Il s'ensuit que la hauteur d'un chaînon non sentinelle suit une loi géométrique de paramètre $\frac{1}{2}$ et de support $\mathbb{N}^\star$ (*cf.* rappels en annexe).

La figure 2 contient un exemple de représentation de l'ensemble $\{-7, -5, -3, 1, 2, 4, 6, 7, 9\}$. Dans cet exemple, nous avons tiré des chaînons de hauteur 1, 2 ou 3. La couche de niveau 0 est formée de tout l'ensemble ; la couche de niveau 1 est formée des entiers $-5$, $-3$, 2 et 9 ; la couche de niveau 2 est formée du seul entier $-3$.

☐ 15 – Écrire en C une fonction `chainon_t *enjmb_init(void)` dont la spécification suit :
*Effet :* crée une copie de l'ensemble vide par l'instanciation de deux nouveaux chaînons sentinelles de hauteur 1 chaînés entre eux.
*Valeur de retour :* pointeur vers le chaînon de tête.

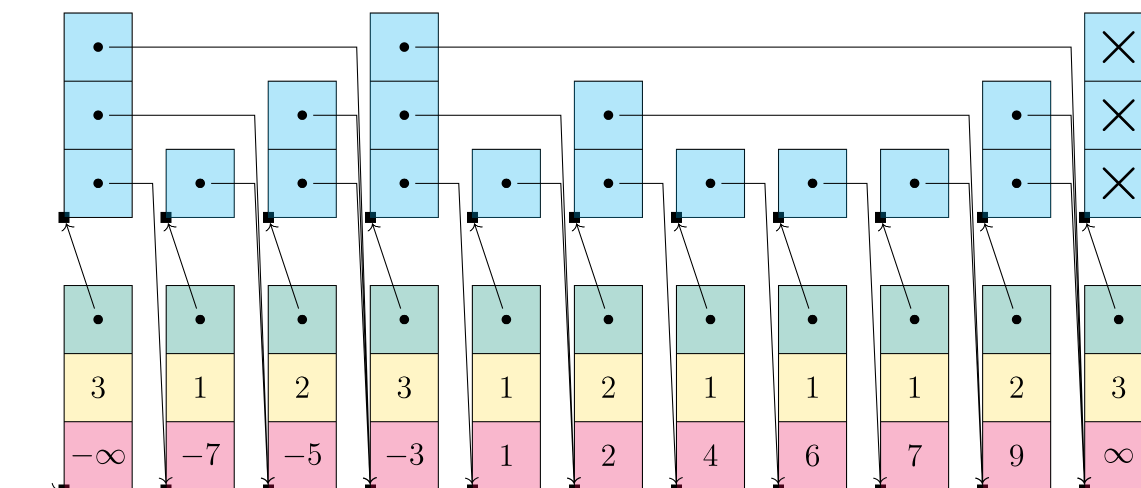



FIGURE 2 – Ensemble $\{-7, -5, -3, 1, 2, 4, 6, 7, 9\}$ représenté par une liste à enjambements (vue complète des chaînons)

☐ 16 – Écrire une fonction en langage C `void enjmb_detruit(chainon_t *t)` dont la spécification suit :
*Précondition :* Le pointeur $t$ désigne le chaînon sentinelle de tête d'une liste à enjambement.
*Effet :* Tous les chaînons de la liste désignée par $t$ sont supprimés, sentinelles comprises.

Soit $K$ une liste à enjambement contenant $n$ éléments distincts fixés à l'avance : $u_1 < u_2 < \cdots < u_n$. Pour chaque entier $i$ compris entre 1 et $n$, nous notons $H_i$ la hauteur du chaînon contenant $u_i$ : les variables $(H_i)_{1 \leqslant i \leqslant n}$ sont des variables aléatoires indépendantes qui suivent une loi géométrique de paramètre $\frac{1}{2}$ et de support $\mathbb{N}^\star$. Nous notons encore $H_{\max} = \max_{1 \leqslant i \leqslant n} H_i$ la hauteur des deux chaînons sentinelles. L'inégalité de Boole permet d'affirmer que, pour tout entier $h$, nous avons $\mathbb{P}(H_{\max} = h) \leqslant \sum_{i=1}^{n} \mathbb{P}(H_i = h)$. Nous rappelons la formule : pour tout entier naturel $a$, nous avons $\sum_{h=a}^{+\infty} \frac{h}{2^h} = \frac{2(a+1)}{2^a}$.

Nous notons $A_n$ l'événement « il existe un chaînon de hauteur strictement supérieure à $3 \log_2 n$ », $A_n^c$ son complémentaire et $\mathbb{1}_{A_n}$ sa fonction indicatrice. La notation $\mathbb{E}(X) = \sum_{x=0}^{+\infty} x \mathbb{P}(X = x)$ désigne l'espérance d'une variable aléatoire $X$ à valeur entière.

☐ 17 – Etablir l'estimation $\mathbb{E}\left(H_{\max} \cdot \mathbb{1}_{A_n}\right) = o(\log_2 n)$ afin de montrer que

$$\mathbb{E}\left(H_{\max}\right) = O\left(\log_2 n\right).$$

☐ 18 – Soit $S$ la complexité en espace de la liste à enjambement $K$. Exprimer la variable aléatoire $S$ en fonction de $n$ et des variables $(H_i)_{1 \leqslant i \leqslant n}$ et $H_{\max}$. Montrer que l'espérance $\mathbb{E}(S)$ de la variable aléatoire $S$ est linéaire en $n$.

Pour rechercher un élément dans une liste à enjambement, on progresse successivement dans les listes chaînées des différents niveaux, en partant du niveau le plus élevé pour finalement

terminer la recherche dans le niveau le plus bas. À chaque dépassement de la valeur ciblée, on descend d'un niveau.

Par exemple, pour rechercher l'élément 8 comme illustré dans la figure 3 : on démarre du chaînon sentinelle ; on progresse jusqu'au chaînon −3 grâce à la couche supérieure de niveau 2 ; ensuite, on progresse jusqu'au chaînon 2 grâce à la couche intermédiaire de niveau 1 ; finalement, on progresse jusqu'au chaînon 7 grâce à la couche inférieure de niveau 0. La recherche échoue enfin.

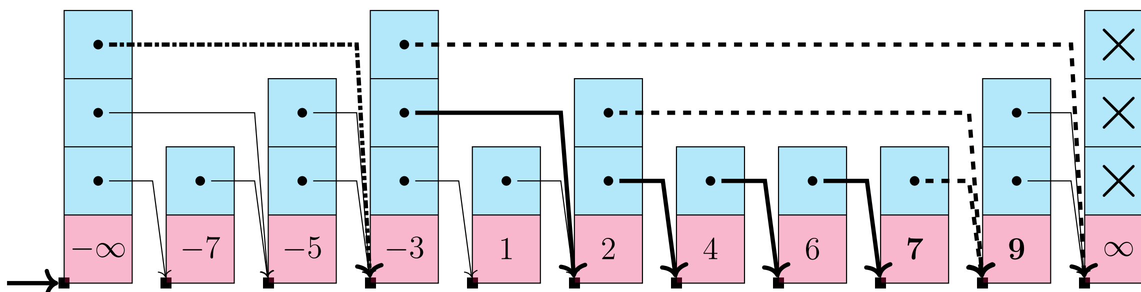



Figure 3 – Recherche de l'entier 8 dans une liste à enjambements (vue compacte des chaînons)

☐ 19 – Écrire une fonction en langage C `bool enjmb_contient(chainon_t *t, int v)` dont la spécification suit :
*Précondition :* Le pointeur $t$ désigne le chaînon sentinelle de tête d'une liste à enjambement.
*Valeur de retour :* Booléen `true` si l'entier $v$ appartient à la liste et `false` sinon.

Soit $v$ un entier fixé à l'avance. Nous notons $R$ le nombre de pointeurs vers un chaînon sondés lors de la recherche de l'élément $v$ dans la liste à enjambement $K$ fixée plus haut. Nous observons que $R$ est une variable aléatoire qui satisfait l'inégalité

$$R \leqslant H_{\max} + 1 + R_{H_{\max}-1} + \cdots + R_1 + R_0,$$

où le terme $H_{\max}$ compte le nombre de pointeurs sondés mais non utilisés pour avancer, où le terme 1 compte le pointeur par lequel on quitte le chaînon sentinelle et où la variable aléatoire $R_h$ représente le nombre de pointeurs employés pour avancer entre deux chaînons internes de la couche de niveau $h$. On considère que $R_h$ vaut 0 si le niveau $h$ est absent. Par exemple, pour la recherche de l'élément 8 illustrée dans la figure 3, on a $H_{\max} = 3$, $R_2 = 0$, $R_1 = 1$ et $R_0 = 3$.

☐ 20 – Indiquer sommairement, d'une part, pourquoi pour tout entier $h$, la variable aléatoire $R_h$ suit une loi géométrique tronquée de paramètre $\frac{1}{2}$ et de support inclus dans $\mathbb{N}$ et observer, d'autre part, que $R' = R_{H_{\max}-1} + \cdots + R_1 + R_0 \leqslant n$ afin de démontrer, en écrivant que $R' = R' \cdot \mathbb{1}_{A_n^c} + R' \cdot \mathbb{1}_{A_n}$, que l'espérance $\mathbb{E}(R)$ de la variable aléatoire $R$ satisfait

$$\mathbb{E}(R) = O(\log_2 n).$$

☐ 21 – Expliquer comment se compare la complexité en temps du test d'appartenance dans une liste à enjambement avec la complexité de la même opération dans un arbre binaire de recherche.

# 3. Utilisation concurrente de listes chaînées

Dans toute cette partie, on raisonne sur les algorithmes en supposant que les fils d'exécution peuvent s'entrelacer mais que les instructions d'un même fil s'exécutent dans l'ordre du programme.

L'entête `#include <pthread.h>` a été déclarée ; la syntaxe de certaines fonctions s'y rattachant est rappelée en annexe.

☐ 22 – Deux fils d'exécution distincts exécutent la fonction `insere` (définie page 3) sur la même liste. Montrer qu'une course critique advient de leur exécution concurrente et que la cohérence de la liste chaînée n'est pas garantie car certains des invariants ✠ 1., ✠ 2., ✠ 3. peuvent être violés à l'issue des exécutions.

## 3.1. Exclusion mutuelle à gros grains

Dans les questions 23 à 26, nous considérons que la totalité du code des fonction d'insertion, de suppression et de test d'appartenance forme une section critique, que nous cherchons à protéger par exclusion mutuelle avec un unique verrou rattaché à la liste. Nous déclarons :

```
41.  struct liste_grosgrain {
42.      maillon_t *tete;
43.      pthread_mutex_t verrou;
44.  };
45.  typedef struct liste_grosgrain liste_grosgrain_t;
```

Nous nous fixons comme règle de toujours verrouiller le verrou avant toute opération sur la liste.

☐ 23 – Écrire une fonction `bool grosgrain_insere(liste_grosgrain_t *l, int v)` en C ou en pseudo-code dont la spécification suit :
*Précondition :* Le pointeur $l$ désigne une liste chaînée.
*Postcondition :* La liste désignée par le pointeur $l$ contient la valeur entière $v$ ainsi que les autres valeurs précédemment contenues.
*Valeur de retour :* Booléen `true` si la liste contient un élément de plus et `false` sinon..

☐ 24 – Expliquer pourquoi l'exclusion mutuelle ne serait pas effective avec une fonction de prototype alternatif `bool grosgrain_insere_bis(liste_grosgrain_t l, int v)`.

Nous supposons qu'une fonction `liste_grosgrain_t *grosgrain_init(void)` et une fonction `bool grosgrain_supprime(grosgrain_liste_t *l, int v)` ont été codées à l'image des questions 1 et 6.

Soit $n$ un entier. Dans le premier fragment de code suivant, une liste vide est créée. Pour tout entier $i$ compris entre 0 et $n-1$, un fil d'exécution y ajoute l'entier $i$ et retire l'entier $i+1$ mod $n$.

```
46.  const int n = 8;
47.  liste_grosgrain_t *l_partage;
48.
49.  void *foo(void *args_ptr) {
```

```
50.       int *args = (int *) args_ptr;
51.       grosgrain_insere(l_partage, args[0]);
52.       grosgrain_supprime(l_partage, args[1]);
53.       return NULL;
54.   }
55.
56.   int main(void) {
57.      l_partage = grosgrain_init();
58.      pthread_t t_id[n];
59.      int args[n][2];
60.      for (int i = 0; i<n; i++) {
61.          args[i][0] = i;
62.          args[i][1] = (i+1)%n;
63.          pthread_create(&t_id[i], NULL, &foo, &args[i]);
64.      }
65.      for (int i = 0; i<n; i++) {
66.          pthread_join(t_id[i], NULL);
67.      }
68.      return 0;
69.   }
```

☐ 25 – Dire si le code ci-dessus se termine pour toute exécution. Le cas échéant, pour chaque entier $\ell$ compris entre 0 et $n$, exhiber une trace d'exécution qui montre que la liste à gros grains `l_partage` peut contenir $\ell$ éléments à l'issue de l'exécution ou démontrer que ce n'est pas possible.

Soit $n$ un entier. Dans le second fragment de code suivant, une liste contenant les entiers 0, 1, …, $n-1$ est initialisée. Pour tout entier $i$ compris entre 0 et $n-1$, un fil d'exécution répète la suppression de l'entier $i$ jusqu'à le supprimer réellement, fait de même avec l'entier $(i+1) \mod n$ puis ajoute à nouveaux ces deux entiers.

```
70.   const int n = 8;
71.   liste_grosgrain_t *l_partage_bis;
72.
73.   void *bar(void *args_ptr) {
74.       int *args = (int *) args_ptr;
75.       while (!grosgrain_supprime(l_partage_bis, args[0])) {}
76.       while (!grosgrain_supprime(l_partage_bis, args[1])) {}
77.       grosgrain_insere(l_partage_bis, args[0]);
78.       grosgrain_insere(l_partage_bis, args[1]);
79.       return NULL;
80.   }
81.
82.   int main(void) {
83.      l_partage_bis = grosgrain_init();
84.      for (int i = 0; i<n; i++) {
85.          grosgrain_insere(l_partage_bis, i);
86.      }
87.      pthread_t t_id[n];
88.      int args[n][2];
```

```
89.     for (int i = 0; i<n; i++) {
90.         args[i][0] = i;
91.         args[i][1] = (i+1)%n;
92.         pthread_create(&t_id[i], NULL, &bar, &args[i]);
93.     }
94.     for (int i = 0; i<n; i++) {
95.         pthread_join(t_id[i], NULL);
96.     }
97.     return 0;
98. }
```

☐ 26 – Dire si le code ci-dessus se termine pour toute exécution. Le cas échéant, pour chaque entier $\ell$ compris entre 0 et $n$, exhiber une trace d'exécution qui montre que la liste à gros grains `l_partage_bis` peut contenir $\ell$ éléments à l'issue de l'exécution ou démontrer que ce n'est pas possible.

## 3.2. Exclusion mutuelle à grains fins

L'implémentation de listes protégées à gros grains demeure grossière. Nous souhaitons affiner la protection contre les courses critiques en dotant chaque maillon de son propre verrou.

**Indication C :** Nous adoptons la nouvelle déclaration :

```
99.  struct maillon_protege {
100.     int donnee;
101.     struct maillon_protege *suivant;
102.     pthread_mutex_t verrou;
103. };
104. typedef struct maillon_protege maillon_protege_t;
```

Nous nous fixons comme règle de verrouiller tout maillon avant de l'utiliser et de toujours parcourir la liste en progressant en verrouillant tout maillon successeur pendant que son prédécesseur est encore verrouillé. Ainsi, un ou deux maillons consécutifs sont toujours verrouillés. Par exemple, lors du parcours de la liste chaînée représentée à la figure 1, l'ordre des verrouillages et des déverrouillages est le suivant :

— le maillon contenant `INT_MIN` est verrouillé
— le maillon contenant $-7$ est verrouillé, puis le maillon contenant `INT_MIN` est déverrouillé
— le maillon contenant 2 est verrouillé, puis le maillon contenant $-7$ est déverrouillé
— le maillon contenant 9 est verrouillé, etc

☐ 27 – Écrire en langage C ou bien en pseudo-code une fonction `maillon_protege_t *grainfin_localise(maillon_protege_t *t, int v)` ayant la même spécification qu'à la question 2 et présidée, de surcroît, par le principe de progression exposé dans le paragraphe qui précède. À l'issue de la fonction, le maillon désigné par la valeur de retour doit être verrouillé.

☐ 28 – Démontrer que l'exécution concurrente de deux appels à la fonction `grainfin_localise` ne conduit pas à des interblocages.

Nous réécrivons la fonction `insere` pour la rendre responsable de déverrouiller le maillon verrouillé par l'appel à la fonction `grainfin_localise`.

```
105.  bool grainfin_insere(maillon_protege_t *t, int v) {
106.      maillon_protege_t *p = grainfin_localise(t, v);
107.      if ((p->suivant)->donnee == v) {
108.          pthread_mutex_unlock(&p->verrou);
109.          return false;
110.      }
111.      else {
112.          maillon_protege_t *n = malloc(sizeof(maillon_protege_t));
113.          n->suivant = p->suivant;
114.          n->donnee = v;
115.          p->suivant = n;
116.          pthread_mutex_unlock(&p->verrou);
117.          return true;
118.      }
119.  }
```

☐ 29 – Deux fils d'exécution distincts exécutent la fonction `grainfin_insere`. Montrer que le ou les invariants identifiés comme enfreints à la question 22 sont maintenant préservés par ces exécutions.

Le *crible d'Eratosthène* est une méthode naïve de construction de la liste des nombres premiers compris entre 2 et un entier $n$ fixé à l'avance. Informellement, il consiste à dresser la liste ordonnée des entiers compris entre 2 et $n$, puis, pour tout entier $p$ encore présent, supprimer de cette liste les multiples $p^2$, $(p+1)p$, $(p+2)p$, $(p+3)p$, $\ldots$ encore présents.

☐ 30 – Nous nous proposons d'implémenter le crible d'Eratosthène par une méthode concurrente : un fil d'exécution principal construit la liste des entiers entre 2 et $n$, puis la parcourt ; chaque suppression des multiples des entiers $p$ rencontrés dans le parcours est confiée à un nouveau fil d'exécution. Discuter les avantages et inconvénients de l'emploi de listes protégées à gros grains ou à grains fins dans ce cadre. Peut-on synchroniser les fils d'exécution de sorte que le fil principal ne rencontre que des nombres premiers ? Est-on garanti, dans ce cas, que les suppressions de nombres composés soient réparties équitablement entre les différents fils d'exécution ?

# A. Annexe : rappels de probabilité

**Définition :** On dit qu'une variable aléatoire $G$ suit une *loi géométrique* de paramètre $\frac{1}{2}$ et de support $\mathbb{N}^\star = \{1, 2, 3, \dots\}$ s'il existe une suite de variables de Bernoulli $(X_i)_{i\in\mathbb{N}^\star}$ indépendantes de probabilité de succès $\frac{1}{2}$ telles que $G$ désigne le rang du premier échec dans la suite $(X_i)_{i\in\mathbb{N}^\star}$. Pour tout entier non nul $g$, on a

$$\mathbb{P}(G = g) = \frac{1}{2^g} \qquad \text{et} \qquad \mathbb{E}(G) = \sum_{g\in\mathbb{N}^\star} \frac{g}{2^g} = 2.$$

**Définition :** On dit qu'une variable aléatoire $G'$ suit une *loi géométrique tronquée* de paramètre $\frac{1}{2}$ et de support inclus dans $\mathbb{N} = \{0, 1, 2, \dots\}$ s'il existe une variable aléatoire $G$ suivant une loi géométrique de paramètre $\frac{1}{2}$ et de support $\mathbb{N} = \{0, 1, 2, \dots\}$ et une variable aléatoire $Y$ à valeur dans $\mathbb{N}^\star$ telles que

$$G' = \min(G, Y - 1).$$

Autrement dit, une loi géométrique tronquée désigne le rang du premier échec dans les $Y$ premiers termes d'une suite de variables de Bernoulli $(X_i)_{i\in\mathbb{N}}$ et vaut $Y - 1$ s'il n'y a pas d'échecs. On a

$$\mathbb{E}(G') \leqslant \mathbb{E}(G) = \sum_{g\in\mathbb{N}} \frac{g}{2^{g+1}} = 1.$$

# B. Annexe : rappels de programmation

Le type `pthread_t` désigne des fils d'exécution.

L'instruction `pthread_create(pthread_t *th_id, NULL, &ma_fonction, void *args)` crée un nouveau fil d'exécution qui appelle la fonction `ma_fonction` sur le ou les arguments désignés par `args` et qui s'exécute simultanément avec le fil d'exécution appelant.

L'instruction `pthread_join(th_id, NULL)` suspend l'exécution du fil d'exécution appelant jusqu'à ce que le fil d'exécution identifié par `th_id` achève son exécution.

Le type `pthread_mutex_t` désigne des verrous.

L'instruction `pthread_mutex_lock(&v)` verrouille le verrou `v`.

L'instruction `pthread_mutex_unlock(&v)` déverrouille le verrou `v`.

Fin de l'épreuve