DS n°4 - Informatique - 3h

Les trois exercices sont indépendants. Vous pouvez les traiter dans l'ordre que vous voulez, mais pour chacun d'eux les questions doivent être traitées dans l'ordre.

Il est conseillé de consacrer environ 1h à chaque exercice.

Si vous repérez ce qui vous semble être une erreur d'énoncé, vous en faites mention dans votre copie et poursuivez votre composition.

Pour le code C, on suppose inclus les fichiers d'entête : `<stdio.h>`, `<stdlib.h>`, `<stdbool.h>`, `<assert.h>`

Le code écrit doit être clair, expliqué brièvement et commenté si besoin ; l'indentation et la syntaxe doivent être respectées.

La présentation, la rédaction et l'orthographe seront pris en compte dans la note finale.

Exercice 1 - *Mediane des médianes - en OCaml*

La sélection du $(k+1)$ème plus petit élément d'une liste d'entiers L, non nécessairement triée, consiste à trouver le $(k+1)$ème élément de la liste obtenue en triant L dans l'ordre croissant.

Par exemple, si $L = [9;5;2;7;1;8]$ le 3ème plus petit élément de L est 5. On pourra remarquer que si la liste L est triée dans l'ordre croissant, le $(k+1)$ème plus petit élément est l'élément de rang k (ou d'indice k) dans L. Ainsi pour $k = 0$, on cherche le plus petit élément de la liste ; pour $k = 1$, on cherche le 2ème plus petit élément de la liste.

L'algorithme naïf qui consiste à trier la liste puis en renvoyer l'élément de rang k est en $\mathcal{O}(n \log n)$ (si on utilise un tri efficace).

On présente ici un algorithme permettant de résoudre ce problème de sélection avec une complexité temporelle linéaire dans le pire cas. Celui-ci est basé sur le principe de "diviser pour régner" et sur le choix d'un bon pivot pour partager la liste en deux sous-listes.

Les fonctions demandées sont à écrire en OCaml et ne doivent faire intervenir aucun trait impératif du langage (références, tableaux ou autres champs mutables ou exception par exemple).

Étant donné un réel a, on note $\lfloor a \rfloor$ le plus grand entier inférieur ou égal à a.

Dans un premier temps, on écrit des fonctions auxiliaires qui seront utiles pour la fonction principale.

Q1. Écrire une fonction récursive de signature `longueur:'a list->int` telle que `longueur l` est la longueur de la liste `l`.

Q2. Écrire une fonction récursive de signature `insertion:'a list->'a->'a list` telle que `insertion l a` est la liste triée dans l'ordre croissant obtenue en ajoutant l'élé ment `a` dans la liste croissante `l`.

Q3. En déduire une fonction récursive de signature `tri_insertion:'a list->'a list` telle que `tri_insertion l` est la liste obtenue en triant `l` dans l'ordre croissant.

Q4. Écrire une fonction récursive de signature `selection_n:'a list->int->'a` telle que `selection_n l n` est l'élément de rang `n` de la liste `l`. Par exemple, `selection_n [4;2;6;7;1;15] 3` est égal à 7.

Q5. Écrire une fonction récursive de signature `paquets_de_cinq:'a list->'a list list` telle que `paquets_de_cinq l` est une liste de listes obtenue en regroupant les éléments de la liste `l` par paquets de cinq sauf éventuellement le dernier paquet qui est non vide et qui contient au plus cinq éléments. Par exemple :

- `paquets_de_cinq []` est égal à `[]`,
- `paquets_de_cinq [2;1;2;1;3]` est égal à `[[2;1;2;1;3]]`,
- `paquets_de_cinq [3;4;2;1;5;6;3]` est égal à `[[3;4;2;1;5];[6;3]]`

Q6. Écrire une fonction récursive de signature `medians:'a list list->'a list` telle que `medians l` est la liste `m` obtenue en prenant l'élément médian de chaque liste apparaissant dans la liste de listes l.
On convient que pour une liste A dont les éléments sont exactement $a_0 \leqslant a_1 \leqslant ... \leqslant a_{n-1}$, l'élément médian est $a_{\lfloor n/2 \rfloor}$.
Dans le cas où la liste L n'est pas triée, l'élément médian désigne l'élément médian de la liste obtenue en triant L par ordre croissant.
Par exemple : `medians [[5;1;5;3;2];[4;3;1];[1;3];[5;1;2;4]]` est égal à `[3;3;1;2]`.

Q7. Écrire une fonction de signature `partage:'a->'a list->'a list*'a list*int*int` telle que `partage p l` est un quadruplet `l1,l2,n1,n2` où `l1` est la liste des éléments de `l` plus petit que `p`, `l2` est la liste des éléments de `l` strictement plus grand que `p`, `n1` et `n2` sont respectivement les longueurs de `l1` et `l2`.

On donne maintenant la fonction de sélection :

Algorithme 1 - Sélection du $(k+1)^e$ plus petit élément

```
1  SELECTION L k :
   /* L est une liste, k est un entier positif                          */
2  début
3      n ← LONGUEUR L
4      si n ≤ 5 alors
5          M ← (TRI_INSERTION L)
6          retourner l'élément de rang k de M
7      fin
8      sinon
9          L_Cinq ← PAQUETS_DE_CINQ L
10         M ← MEDIANS L_Cinq
11         pivot ← SELECTION M  ((n+4)//5)//2
           /* L'opérateur // désigne le quotient d'entiers. Le rang ((n+4)//5)//2
              correspond au rang du médian de la liste M                 */
12         L1, L2, n1, n2 ← PARTAGE pivot L
13         si k < n1 alors
14             retourner SELECTION L1 k
15         fin
16         sinon
17             retourner SELECTION L2 (k − n1)
18         fin
19     fin
20 fin
```

Q8. Écrire une fonction récursive de signature `selection:'a list->int->'a` telle que `selection l k` est le $(k+1)$ème plus petit élément de la liste `l`.
L'écriture de la fonction sera une traduction en OCaml de l'algorithme ci-dessus.

On cherche à déterminer la complexité en nombre de comparaisons de la fonction `selection`. Pour tout $n \in \mathbb{N}$, on note $T(n)$ le nombre maximum de comparaisons entre éléments lors d'une sélection d'un élément quelconque dans des listes L sans répétition de taille n.

En analysant l'algorithme ci-dessus, il est possible de démontrer que :

$$\forall n \geqslant 55,\ T(n) \leqslant T\left(\left\lfloor \frac{n+4}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{8n}{11} \right\rfloor\right) + 4n$$

Q9. En admettant la proposition ci-dessus, montrer que pour tout entier n supérieur à 1, on a :

$$T(n) \leqslant (200 + T(55))n$$

Pour l'initialisation, on pourra remarquer que T est une fonction croissante.

Exercice 2 - *Elément majoritaire d'un tableau - en C*

On s'intéresse au problème suivant : savoir si un tableau contient un élément majoritaire. Si un tableau a une taille $n > 0$, un élément de ce tableau est dit majoritaire s'il apparaît strictement plus que $\lfloor n/2 \rfloor$ fois (on rappelle que $\lfloor . \rfloor$ dénote la partie entière). Par exemple, un tableau de taille 5 contenant comme éléments 2, 7, 2, 2, 9 possède 2 comme élément majoritaire, alors qu'un tableau contenant 2, 7, 2, 2, 9, 8 n'a pas d'élément majoritaire.

Q1. Montrer que si un tableau admet un élément majoritaire, alors celui-ci est unique.

On commence par une résolution naïve du problème.

Q2. Ecrire une fonction `int occurrences(int t[], int taille, int x)` prenant en entrée un tableau de taille `taille`, un élément `x`, et renvoyant le nombre de fois que `x` apparaît dans le tableau.

Q3. Ecrire une fonction `bool majoritaire(int t[], int taille)` prenant en entrée un tableau de taille `taille` et renvoyant un booléen indiquant s'il possède un élément majoritaire.

Q4. Donner la complexité de votre fonction en fonction de la taille notée n du tableau.

On propose une résolution plus efficace. Cette approche consiste à trouver d'abord un unique candidat majoritaire, puis à vérifier si celui-ci est effectivement majoritaire.

La première partie de l'approche est détaillée dans l'algorithme ci-dessous. Pour que ce soit plus visuel, on propose l'analogie suivante : le tableau d'entiers est un seau de balles colorées (deux balles de même couleur correspondent à deux éléments identiques dans le tableau). On dispose d'un tube se comportant comme une pile, et d'une corbeille.

Algorithme 1 :

Entrée : Le seau de balles
Sortie : une couleur de balle
Créer un tube vide, une corbeille vide;
Placer une balle du seau dans le tube ;
tant que *il reste des balles dans le seau* **faire**
  prendre une balle dans le seau;
  si *la balle est de la même couleur que celle au sommet du tube* **alors**
    mettre la balle à la corbeille
  sinon
    empiler la balle dans le tube ;
    si *la corbeille est non vide* **alors**
      prendre une balle de la corbeille et la mettre dans le tube
retourner *la couleur de la balle au sommet du tube*

Q5. Montrer que "toutes les balles éventuellement présentes dans la corbeille ont la même couleur que celle au sommet du tube" est un invariant de boucle.

Q6. Montre que "deux balles situées côté à côte (ou plutôt, l'une en dessous de l'autre) dans le tube ont des couleurs différentes" est un invariant de boucle.

Q7. Considérons à la fin de l'algorithme une couleur qui n'est pas celle de la balle au sommet du tube. Montrer à l'aide des invariants précédents qu'elle apparaissait moins de $n/2$ fois dans le seau, avec n le nombre initial de balles.

On en déduit que, pour voir si un tableau admet un élément majoritaire, il suffit de tester si l'élément renvoyé est majoritaire.

Q8. Ecrire une fonction d'en tête `bool majoritaire_efficace(int t[], int taille)` de même spécification qu'à la question 3, mais avec l'approche décrite ci-dessus. A la place du tube, vous utiliserez un entier (indiquer ce qu'il représente). A la place de la corbeille, vous utiliserez également un entier (indiquer également ce qu'il représente).

Q9. Quelle est la complexité de votre fonction ?

Exercice 3 - *Listes à enjambement - en C*

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 représentable par le type **`int`** en machine et sont représentées par $-\infty$ et ∞ 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).

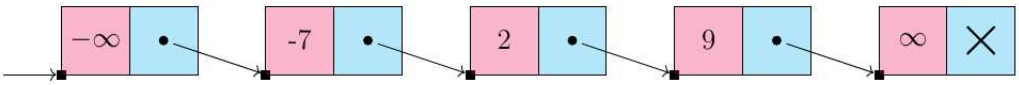

Figure 1 – Exemple d'ensemble d'entier représenté par une liste chaînée

☐ 1 – Écrire en C une fonction `maillon_t *init()` 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 l'état 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. Nous exécutons le programme suivant.

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

Dire dans laquelle ou dans lesquelles de ces régions la valeur entière 717 est inscrite.

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*.

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.

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()` 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.

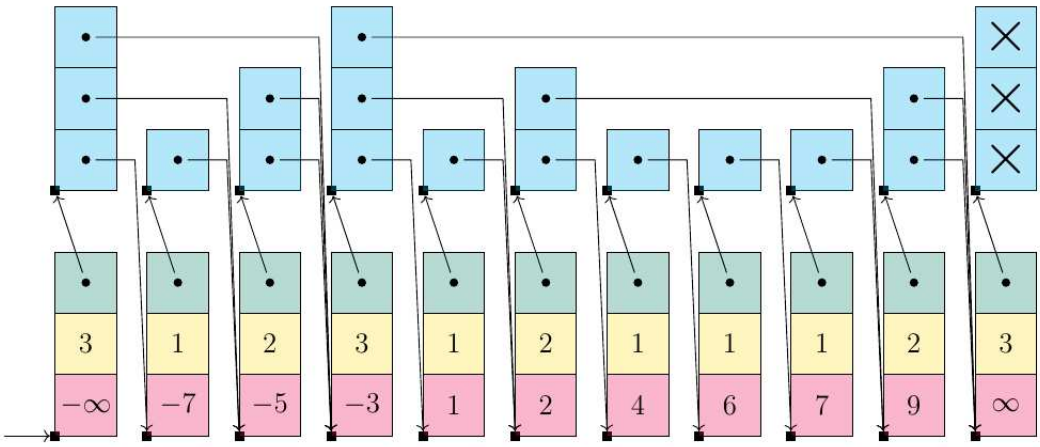

FIGURE 2 – Ensemble $\{-7, -5, -3, 1, 2, 4, 6, 7, 9\}$ représenté par une liste à enjambements (vue complète des chaînons)

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.

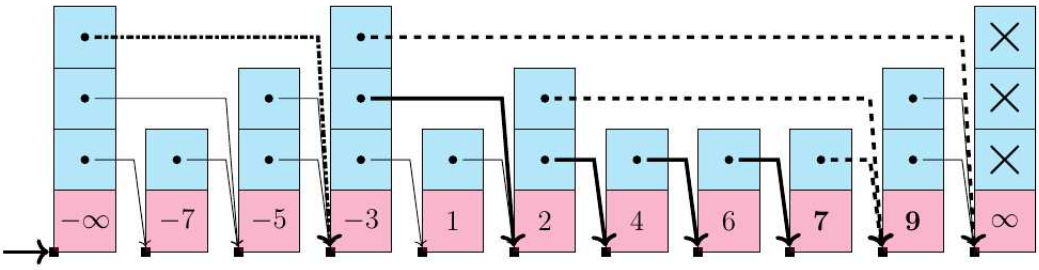

FIGURE 3 – Recherche de l'entier 8 dans une liste à enjambements (vue compacte des chaînons)

☐ 18 – 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.

FIN DU SUJET