DS n°4 . correction . TP2I

Exercice 1 : (médiane des médianes)

Q1)
```
let rec longueur l = match l with
| [] -> 0
| e::ll -> 1 + longueur ll
```
(* on procède récursivement sur la structure de liste *)

Q2)
```
let rec insertion l a = match l with
| [] -> [a]
| e::ll -> if a<e then a::l else e::(insertion ll a)
```
(* récursivement, et on distingue les cas a<e, a≥e *)

Q3)
```
let rec tri_insertion l = match l with
| [] -> []
| e::ll -> let t_ll = tri_insertion ll in
           insertion t_ll e
```
(* récursivement ; on trie la liste ll avant d'y insérer e *)

Q4)
```
let rec selection_n l n = match l with
| [] -> failwith "indice invalide"
| e::ll -> if n=0 then e else selection_n ll (n-1)
```
(* récursivement : - si la liste est vide, c'est un échec
- si la liste est non-vide et n=0, c'est l'élément de tête ; si n>0 on cherche le (n-1)ème élément de ll *)

Q5)
```
let rec paquets_de_cinq l = match l with
| a::b::c::d::e::ll -> [a;b;c;d;e]::paquets_de_cinq ll
                       (* s'il y a au moins 5 élts *)
| [] -> []  (* cas particulier de la liste vide *)   NB : ≠ [[]]
| _ -> [l]  (* s'il y a entre 1 et 4 éléments *)
```

Q6)
```
let rec medians l = match l with                (* récursivement,
  | [] -> []                                     pour chaque liste, on
  | l1 :: ll -> let n = longueur l1 in           calcule la médiane :
      let t_l = tri_insertion l1 in              tri + sélection de l'élément
      let med = selection_n t_l (n/2) in         médian *)
      med :: (medians ll)
```

NB : on reconnait un List.map :
```
let medians l =
  let mediane l1 = selection_n (tri_insertion l1) (longueur l1 / 2) in
  List.map mediane l
```

Q7)
```
let rec partage p l =                  (* à nouveau récursivement ... *)
  match l with
  | [] -> [], [], 0, 0
  | e :: ll -> let (l1, l2, n1, n2) = partage p ll in
      if e <= p then (e :: l1, l2, n1+1, n2)     (* e ≤ p : on met e dans la première liste *)
      else (l1, e :: l2, n1, n2+1)               (* e > p : on met e dans la seconde liste *)
```

Q8)
```
let rec selection l k =                (* on implémente l'algorithme *)
  let n = longueur l in
  if n <= 5 then selection_n (tri_insertion l) k
  else let l5 = paquets_de_cinq l in
       let m = medians l5 in                     (* NB : / : division entière *)
       let p = selection m (((n+4)/5)/2) in
       let l1, l2, n1, n2 = partage p l in
       if k < n1 then selection l1 k else selection l2 (k - n1)
```

Q9) On procède par récurrence forte sur n.

(Pour initier l'inégalité, on doit avoir $n \geq 5$. On initialise donc pour $n \in [\![1, 54]\!]$.)

$P(n)$: " $T(n) \leq (200 + T(55)) \times n$ "

* Initialisation : $\forall n \in [\![1, 54]\!]$, $T(n) \leq T(55) \leq 200 + T(55)$

T croissante $\leq (200 + T(55)) \times n$.

$n \geq 1$

* Hérédité : Soit $n \geq 55$. Supposons $P(k)$ vraie $\forall k \in [\![1, n-1]\!]$,

démontrons $P(n)$. D'après l'inégalité :

$$T(n) \leq T\left(\underbrace{\left\lfloor \frac{n+4}{5} \right\rfloor}_{= a \in [\![1, n-1]\!]}\right) + T\left(\underbrace{\left\lfloor \frac{8n}{11} \right\rfloor}_{= b \in [\![1, n-1]\!]}\right) + 4n$$

D'où, d'après $P(a)$ et $P(b)$:

$$T(n) \leq (200 + T(55)) \times a + (200 + T(55)) \times b + 4n$$

$$\leq \underbrace{200 \times (a+b) + 4n}_{\leq 200n \ ?? \ (1)} + T(55) \times \underbrace{(a+b)}_{\leq n \ ?? \ (2)}$$

or $a + b \leq \frac{n+4}{5} + \frac{8n}{11} = \frac{11n + 44 + 40n}{55} = \frac{44 + 51n}{55} \leq \frac{52n}{55} \leq n$ (car $44 \leq n$) d'où (2).

et $200 \times (a+b) + 4n \leq 200n \iff (a+b) \leq \frac{196}{200} n \iff \frac{a+b}{n} \leq \frac{49}{50}$

Ce qui est vrai car $\frac{a+b}{n} \leq \frac{52}{55}$ (cf ci-dessus) et $\frac{52}{55} < \frac{49}{50}$

(1) d'où $T(n) \leq (200 + T(55))n$ $\left(\iff 1 - \frac{52}{55} > 1 - \frac{49}{50} \iff \frac{3}{55} > \frac{1}{50}\right.$

$\left. \iff 150 > 55 \text{ vrai}\right)$

* Conclusion : $\forall n \in \mathbb{N}^*$, $P(n)$ vraie

(et l'algo. proposé est de complexité linéaire !)

Exercice 2 : (Elément majoritaire)

Q1] Par l'absurde, supposons qu'il existe au moins deux éléments majoritaires x et y et n_x, n_y leur nombre d'occurrences ($x \neq y$) respectifs. On a d'— part $n_x + n_y \leq n$ (ce sont des éléments du tableau)

et $n_x > \lfloor \frac{n}{2} \rfloor$ et $n_y > \lfloor \frac{n}{2} \rfloor$

d'où $n_x \geq \lfloor \frac{n}{2} \rfloor + 1$ et $n_y \geq \lfloor \frac{n}{2} \rfloor + 1$

d'où $n_x + n_y \geq 2 \lfloor \frac{n}{2} \rfloor + 2$ or $2 \lfloor \frac{n}{2} \rfloor = n$ si n pair
$= n-1$ si n impair
$\geq n-1$

D'où $n_x + n_y \geq n+1$

donc $n_x + n_y > n$ contradiction ! (Si existe un élément majoritaire, celui-ci est unique.)

Q2]
```
int occurences(int t[], int taille, int x){
    int c = 0;
    for(int i=0; i<taille; i+=1){
        if (t[i] == x){
            c+=1;
        }
    }
}
```
// on parcourt le tableau en comptant le nombre d'occurrences de x.

Q3)
```
bool majoritaire(int t[], int taille){
    for(int i=0; i<taille; i+=1){
        if (occurences(t, taille, t[i]) > taille/2){   // NB : division entière
            return true;
        }
    }
    return false;
}
```
// pour chaque élément du tableau, on vérifie s'il est majoritaire. Si ce n'est jamais le cas, c'est qu'il n'y a pas d'élt majoritaire

Q4) occurrences est en $\Theta(n)$ (avec n : taille du tableau)
majoritaire réalise dans le pire des cas n tours de boucles,
avec un appel à occurrences et quelques opérations en $\Theta(1)$ à
chaque tour.

La complexité de la fonction majoritaire est donc en $\boxed{\Theta(n^2)}$

Q5) Soit $\mathcal{P}$: "toutes les balles de la corbeille sont de même couleur que celle du sommet du tube".

Initialisation : avant d'entrer dans la boucle : la corbeille est vide
donc $\mathcal{P}$ vraie. (NB : $\forall x \in E, \mathcal{P}(x)$ est vraie pour $E = \emptyset$
en effet $\exists$ une balle de la corbeille de couleur différente de celle du sommet est fausse !)

Conservation : supposons $\mathcal{P}$ vraie en début de tour de boucle.
Soit c la couleur du sommet du tube et des balles de la corbeille.
Soit c' la couleur de la balle prise dans le sceau

* si $c' = c$: on met la balle dans la corbeille : $\mathcal{P}$ reste vraie
* si $c' \neq c$: la couleur du sommet du tube devient c' ,
  - soit la corbeille est vide et $\mathcal{P}$ est vraie (voir initialisation)
  - soit la corbeille est non vide, on remet par dessus une balle de couleur c et toutes les balles de la corbeille (s'il en reste) sont de couleur c : $\mathcal{P}$ est vraie.

Dans tous les cas $\mathcal{P}$ est vrai en fin de tour de boucle.

$\mathcal{P}$ est un invariant de boucle.

Et comme on sort de la boucle <u>while</u> (le nombre de balles dans le sceau est un variant de boucle : entier positif strictement décroissant)

$\mathcal{P}$ est vraie en sortie de boucle while

Q6] Soit Q : "deux salles situées côte-à-côte dans le hutte sont de couleurs différentes".

Initialisation : avant d'entrer dans la boucle Q est vraie : il n'y a qu'une salle dans le hutte.

Conservation : Supposons Q vraie en début de tour de boucle.
On note c et c' de même qu'à la question précédente.

- si $c' = c$: on ne modifie pas le hutte : Q reste vraie.
- si $c' \neq c$: on empile c' (changement de couleur) et si la corbeille est non vide on empile c (nouveau changement de couleur)

Dans les deux cas Q reste vraie.

Donc Q est un invariant de boucle, et puisqu'on sort de la boucle urbile (voir question précédente) Q est vraie en sortie de boucle urbile

Q7] Soit c' une couleur différente de celle du sommet du hutte, $n_{c'}$ son nombre d'occurrences / salles. En fin d'algorithme, toutes les salles de couleur c' sont dans le hutte, et d'après l'alternance des couleurs et le fait qu'au sommet il y a une salle de couleur différente de c' $n_{c'} \leq \frac{n_t}{2}$ où n_t est le nombre de salles dans le hutte.

Or $n_t \leq n$ d'où $\boxed{n_{c'} \leq \frac{n}{2}}$

et $n_{c'}$ étant entier $n_{c'} \leq \left\lfloor \frac{n}{2} \right\rfloor$: c' n'est pas une couleur majoritaire

Q8) bool majoritaire-efficace(int t[], int taille){
assert(taille > 0); // ou if(taille == 0) return false
int sommet = t[0]; // il suffit de conserver la couleur du sommet
int cordille = 0; // il suffit de conserver le nombre de balles dans la cordille

for (int i = 1; i < taille; i += 1) {
if (t[i] == sommet) {
cordille += 1;
} else if (cordille > 0) { // cas t[i] ≠ sommet et il y a des balles dans la cordille
cordille -= 1;
} else { // cas t[i] == sommet et cordille vide
sommet = t[i];
}
}
// invariant : sommet est le seul candidat majoritaire

return occurences(t, taille, sommet) > taille/2;
}

Q9) L'application de l'algorithme est en $\Theta(n)$: $n-1$ tours de boucle avec quelques opérations en $\Theta(1)$ à chaque tour.

L'appel final à occurences est en $\Theta(n)$

D'où une complexité en $\Theta(n) + \Theta(n) = \boxed{\Theta(n)}$

(et même $\Theta(n)$)

Exercice 3 (listes à enjambement)

Q1)

```
maillon_t * init() {
    maillon_t* s1 = malloc(sizeof(maillon_t));
    s1 -> donnee = INT_MIN;
    s1 -> suivant = malloc(sizeof(maillon_t));
    s1 -> suivant -> donnee = INT_MAX;
    s1 -> suivant -> suivant = NULL;
    return s1;
}
```

// on alloue les deux maillons initiale dans le tas.

Q2)

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

// t -> suivant n'est jamais NULL grâce à la sentinelle finale

// on avance tant que la valeur du maillon suivant est < v.

(version récursive :

```
maillon* localise(maillon_t* t, int v) {
    if (t -> suivant -> donnee < v) {
        return localise(t -> suivant, v);
    } else {
        return t;
    }
}
```

)

Q3)

- insertion dans l'ensemble vide $[-\infty] - [+\infty] \rightsquigarrow [-\infty] - [v] - [+\infty]$ (true)
- insertion d'une valeur absente $[-\infty] - - - [+\infty] \rightsquigarrow [-\infty] - - - [v] - - - [+\infty]$ (true)
- insertion d'une valeur présente $[-\infty] - [v] - [+\infty] \rightsquigarrow$ inchangé (false)

Q4) `&t` est de type `maillon_t**` qui ne correspond pas au type attendu du premier paramètre de localise.

On corrige avec (7). `maillon_t* p = localise(t, v);`

Q5) on remarque que la fonction proposée ne renvoie jamais false... le test d'insertion d'une valeur déjà présente va échouer.

On corrige :

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

// localise renvoie le maillon précédent l'emplacement de v (pour pouvoir mettre à jour le pointeur)

Q6)

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

// on localise v, puis on met à jour les pointeurs (si la valeur était bien présente) et rend le maillon libéré au vas

Q7) localise est en $\mathcal{O}(n)$ où n est la longueur de la liste.

supprime et insère appellent localise puis font un nombre d'opérations borné. supprime et insère sont donc en $\mathcal{O}(n)$

Q8) Les 3 régions mémoire pour stocker les données sont :

- la zone "statique" pour les variables globales
- la pile (paramètres et variables locales)
- le tas (pour l'allocation dynamique de mémoire).

Ici : ligne 14 : 717 est écrit dans la zone statique.

ligne 17 : v est passé en paramètre d'un appel de fonction, donc 717 sera écrit dans la pile

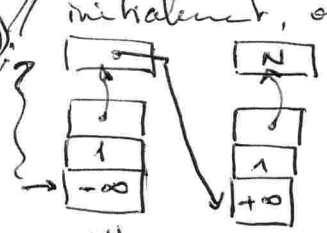
au cours de l'appel, 717 est écrit dans le maillon dynamiquement alloué, donc dans le tas

> 717 est écrit dans les 3 zones mémoire !

Q9)

```
chainon_t * cyl_init() {
    chainon_t* s1 = malloc(sizeof(chainon_t));
    s1->donnee = INT_MIN;
    s1->hauteur = 1;
    s1->suivants = malloc(1 * sizeof(chainon_t));
    chainon_t* s2 = malloc(sizeof(chainon_t));
    s2->donnee = INT_MAX;
    s2->hauteur = 1;
    s2->suivants = malloc(1 * sizeof(chainon_t));
    s1->suivants[0] = s2;
    s2->suivants[0] = NULL;
    return s1;
}
```

initialement, on a



NB : suivants est un tableau de pointeurs alloué dans le tas.

Q10)

```
bool est_contient(chainon_t* t, int v) {
    int i = t->hauteur - 1;   // on part du niveau maximal
    while (i >= 0) {
        while (t->suivants[i]->donnee < v) {   // on localise le
            t = t->suivants[i];                // maillon précédant
        }                                      // v
        if (t->suivants[i]->donnee == v) {
            return true;          // on a trouvé !
        } else {
            i -= 1;               // on descend d'un niveau
        }
    }
    return false;   // v n'a pas été trouvé sur le niveau 0
}                   // donc v est absent.
```

— FIN —