Informatique
MP2I 2022/2023
DS6
CIV

Durée totale 4h.
Tout document interdit. Tout dispositif électronique interdit.
On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

1 Arbres naires en C

On va représenter ici des arbres naires en C.
On considère le type suivant :

```
struct noeud {
    int valeur;
    int arite;
    struct noeud *enfants;
};

typedef struct noeud arbre;
```

Ainsi, chaque nœud contiendra : sa valeur, c'est-à-dire l'étiquette du nœud, son arité, c'est-à-dire son nombre de nœuds enfants et le tableau de ses enfants.
Les nœuds d'arité nulle sont les feuilles.

Q.1

Quelle différence avec le type vu en OCaml pour la même structure ? Expliquer.

Démonstration. On ne peut pas connaitre la taille d'un tableau passé par pointeur comme dans cette structure. Il est donc nécessaire d'avoir également l'arité dans les nœuds.
On remarque qu'on aurait aussi pu faire une liste chaînée de nœuds enfants, mais cela rajoute beaucoup de complexité dans la gestion de la mémoire. □

Q.2

Expliquer comment on peut représenter l'arbre vide en considérant des éléments du type `arbre *`.

Démonstration. Il suffit de le représenter avec le vecteur `NULL`. Une subtilité ici, l'arbre vide n'est jamais un sous-arbre. On est plus proche d'un type optionnel comme en OCaml. □

Q.3

Écrire une fonction de prototype

```
int taille(arbre *a);
```

qui renvoie le nombre de nœud d'un arbre.

Démonstration.

```
int taille(arbre *a) {
    if (a == NULL)
        return 0;

    int t = 1;
    for(int i = 0; i < a->arite; i++) {
        t += taille(&a->enfants[i]);
    }
    return t;
}
```

□

Q.4

Écrire une fonction qui affiche tous les nœuds dans un parcours préfixe.

```
void affiche(arbre *a);
```

Démonstration.

```
void affiche(arbre *a) {
    if (a == NULL)
        return;

    printf("%d\n", a->valeur);
    for(int i = 0; i < a->arite; i++) {
        affiche(&a->enfants[i]);
    }
}
```

□

Q.5

Écrire une fonction de prototype

```
int somme(arbre *a);
```

qui renvoie la somme des étiquettes des nœuds dans l'arbre.

Démonstration.

```
int somme(arbre *a) {
    if (a == NULL)
        return 0;

    int s = a->valeur;
    for(int i = 0; i < a->arite; i++) {
        s += somme(&a->enfants[i]);
    }
    return s;
}
```

□

Q.6

On souhaite libérer la mémoire occupée par un arbre et Justin Topain, votre camarade, propose le code suivant :

```
void libere(arbre *a) {
    free(a);
    for(int i = 0; i < a->arite; i++) {
        free(&a->enfants[i]);
    }
}
```

Indiquer les erreurs logiques dans ce code en expliquant pourquoi elles en sont. Proposer une version correcte de cette fonction.

Démonstration. Un problème ici est de libérer `a` avant d'accèder à `a->enfants` ou `a->arite`. De plus, il faudrait gérer l'arbre vide.
On a également un problème avec le passage par pointeur, `a->enfants[i]` est un `arbre` et pas un `arbre *`. Notons que c'est le même problème dans les questions précédentes.
Autre problème, cette fonction doit être récursive pour ne pas libérer qu'un nœud et ses enfants.

```c
void libere(arbre *a) {
    if (a == NULL)
                return;
    for(int i = 0; i < a->arite; i++) {
        libere(&a->enfants[i]);
    }
    free(a);
}
```

□

2 Recherche trichotomique

On considère ici un tableau de valeurs entières triées dans l'ordre croissant. Pour rechercher si un élément x est dans un tel tableau de taille $n \geq 2$, on propose de prendre les deux éléments d'indice $k = \lfloor \frac{n}{3} \rfloor$ et $p = \lfloor \frac{2n}{3} \rfloor$:

- Si $x = t[k]$ on renvoie k
- Si $x = t[p]$ on renvoie p
- Si $x < t[k] \leq t[p]$, on continue la recherche dans le tableau des indices 0 à $k-1$
- Si $t[k] < x < t[p]$, on continue la recherche dans le tableau des indices $k+1$ à $p-1$.
- Si $x > t[p]$, on continue la recherche dans le tableau des indices $p+1$ à $n-1$.

Quand le tableau a 0 ou 1 élément, on se contente de répondre faux ou de vérifier que le seul élément est x.

Q.7

La manière dont on a rédigé cet algorithme considère qu'on va extraire des sous-tableaux pour faire des appels récursifs. Justifier brièvement pourquoi ce n'est pas une bonne idée ici.

Démonstration. Ici, on a un algorithme où la fusion est clairement en $O(1)$. Or, les extractions de sous-tableaux induisent une fusion linéaire, ce qui est forcément plus coûteux que l'algorithme lui-même. □

Q.8

Décrire, sans écrire de code mais avec précision, un algorithme récursif de recherche n'effectuant pas d'extraction de sous-tableaux.

Démonstration. **Petite imprécision du sujet qui parle de renvoyer un indice dans les premiers cas puis change d'avis en demandant juste le booléen.**
On considère une fonction récursive d'interface $Rech(t, i, j, x)$ qui va chercher x dans t aux indices entre i et j (inclus).

- Si $j < i$ on renvoie faux.
- Si $j = i$, on teste si $t[i] = x$.
- Sinon, on pose $k = \lfloor \frac{i+j}{3} \rfloor$ et $p = \left\lfloor \frac{2(i+j)}{3} \right\rfloor$.
  - Si $x = t[k]$ ou $x = t[p]$ on renvoie vrai.
  - Si $x < t[k] \leq t[p]$, on continue avec $Rech(t, i, k-1)$
  - Si $t[k] < x < t[p]$, on continue avec $Rech(t, k+1, p-1)$
  - Si $x > t[p]$, on continue avec $Rech(t, p+1, j)$

□

On note $T(n)$ le nombre d'opérations élémentaires effectuées pour un tableau de n valeurs.

Q.9

Montrer qu'on a, pour $n \geq 2$,

$$T(n) \leq T\left(\left\lceil \frac{n}{3} \right\rceil\right) + O(1)$$

et $T(0) = T(1) = 1$

Démonstration. Pour les cas de base, le 1 est un peu grossier, on remarque juste qu'on effectue un $O(1)$ opérations élémentaires.
Si $n \geq 2$, on va couper en trois. Les fragments sont de taille $\leq \lceil \frac{n}{3} \rceil$, on continue dans un seul, et le coût du découpage est en $O(1)$, on a donc la majoration de $T(n)$ demandée. □

Q.10

On pose $u_n = T(3^n)$.
Exprimer u_n sans récursivité.

Démonstration. **Imprécision encore, c'est une majoration qu'on exprime ici.**
$u_0 = T(1) = 1$ et $u_{n+1} = T(3^{n+1}) \leq u_n + O(1)$.
Ainsi $u_n \leq nO(1) = O(n)$. □

Q.11

En déduire la complexité de cette recherche sous la forme d'un $O(\cdots)$.

Démonstration. Soit p tel que $3^{p-1} < n \leq 3^p$, donc $p = \lceil log_3 n \rceil$.
On a $T(n) \leq T(3^p) \leq O(p) = O(\log_3 n)$. □

Q.12

Cette recherche est-elle meilleure que la recherche dichotomique ?

Démonstration. On a $O(\log_3 n) = O(\log_2 n)$ donc on obtient exactement la même complexité asymptotique. □

3 Table de hachage avec chainage

On va réaliser dans cette partie une table de hachage avec chainage pour gérer les collisions en OCaml.
On va utiliser ici la fonction `Hashtbl.hash` qui le prototype suivant :

```
val hash : 'a -> int
```

et qui à toute valeur `x` va associer un entier strictement positif `hash x`. On a la garantie que si `x = y` alors `hash x = hash y`.

3.1 Ensemble

On considère le problème plus simple vu en classe consistant à représenter un ensemble fini de valeur.

Q.13

Quelle relation entre ce problème est celui de la création d'un dictionnaire ?

Démonstration. Il suffit de rajouter des couples `(clé,valeur)` est de ne considérer que la `clé` pour le hachage. □

On va réaliser ici une table de hachage où les cases contiennent des listes de valeur pour gérer les collisions.
On considère donc le type suivant :

```
type 'a table = 'a list array
```

Q.14

Écrire une fonction

```
val contient : 'a table -> 'a -> bool
```

qui indique si un élément est dans la table.

Démonstration.

```
let contient t x = List.mem x t.(hash x)
```

□

Q.15

Écrire une fonction

```
val ajoute : 'a table -> 'a -> unit
```

qui rajoute un élément dans la table.

Démonstration.

```
let ajoute t x =
    let h = hash x in
    t.(h) <- x :: t.(h)
```

□

3.2 Dictionnaire

On considère ici le type précédent enrichi pour des dictionnaires associant des `'a` à des `'b`.

```
type ('a, 'b) dico = ('a * 'b) list array
```

On continue encore avec le même principe de chainage en cas de collision mais on ne le fait que sur la première composante du couple.

Q.16

Écrire une fonction

```
val assoc : ('a * 'b) list -> 'a -> 'b
```

qui prend une liste de couples `l` et une valeur `x` et renvoie le premier `y` tel que `(x,y)` soit dans `l`.
Si l'élément n'est pas dedans, on fera une erreur.

Démonstration.

```
let rec assoc l x =
    match l with
    | [] -> failwith "Non présent"
        | (y,v)::q ->
                if x = y
                then v
                else assoc q x
```

□

Q.17

En déduire une fonction

```
val dico_recherche : ('a, 'b) dico -> 'a -> 'b
```

qui prend un dico `d` et une valeur `x` et renvoie un `y` associé.

Démonstration.

```
let dico_recherche d x = assoc d.(hash x) x
```

□

Q.18

Quels sont les défauts de cette implémentation comparativement à l'adressage ouvert vu en TP ?

Démonstration. Le problème est la fragmentation mémoire induite par les listes chaînées. Sans mécanisme particulier, chaque collision risque d'engendrer des lenteurs d'accès mémoire qui sont évitées par un tableau qui peut tenir complètement dans le cache du processeur. □

4 Médians

Un *médian* d'un ensemble $X = \{e_1, e_2, \ldots, e_n\}$ de n nombres entiers distincts et un nombre e dans X tel que les nombres d'éléments strictement plus petits et strictement plus grands que e dans X diffèrent d'au plus de 1 ($n > 0$). Si n est impair, le médian est unique ; si n est pair, il y a deux médians possibles.
Le problème de la *sélection* consiste à trouver l'élément de rang k dans X, c'est-à-dire l'élément se trouvant en k-ième position quand X est trié en ordre croissant ($1 \leq k \leq n$).
Nous supposerons l'ensemble X représenté par la liste de ses élements, c'est-à-dire par le type ensemble défini par :

```
type ensemble = int list
```

On supposera que les `ensemble` sont sans répétitions mais pas nécessairement triés.

Q.19

Écrire une fonction `card` qui prend un ensemble et renvoie son nombre d'éléments.

```
val card : ensemble -> int
```

Quelle est la complexité de cette fonction par rapport à n ?

Démonstration. Vu la question, il fallait réécrire la fonction `List.length`.

```
let rec card e =
    match e with
    | [] -> 0
    | _::q -> 1 + card q
```

□

Soit p un entier quelconque, on sera amené à partitionner l'ensemble X en éléments plus grands, égaux ou plus petits que p.

Q.20

Écrire la fonction `nPetits` qui prend en arguments un entier p et l'ensemble X ; et qui renvoie le nombre d'éléments strictement inférieurs à p dans X.

```
val nPetits : int -> ensemble -> int
```

Quelle est la complexité de cette fonction par rapport à n ?

Démonstration.

```
let rec nPetits p x =
    match x with
    | [] -> 0
    | t::q -> (if t < p then 1 else 0) + nPetits q
```

Cette fonction est linéaire en n car elle effectue un parcours de la liste. □

Pour sélectionner le k-ième élément de X, on prend un nombre p bien choisi dans X, appelé *pivot* ($p \in X$), et on effectue une partition de X par rapport à p.

Q.21

Écrire la fonction `partitionP` qui prend en arguments un entier p et l'ensemble X ; et qui renvoie l'ensemble de tous les éléments de X strictement plus petits que p.

```
val partitionP : int -> ensemble -> ensemble
```

Modifier cette fonction pour obtenir la fonction `partitionG` qui retourne l'ensemble de tous les éléments de X strictement plus grands que p. Quelles sont les complexités en temps de ces fonctions par rapport à n ?

Démonstration.

```
let rec partitionP p x =
    match x with
    | [] -> 0
    | t::q -> let q' = partitionP p q in
        if t < p then t::q' else q'

let rec partitionG p x =
    match x with
    | [] -> 0
    | t::q -> let q' = partitionG p q in
        if t > p then t::q' else q'
```

Exactement comme la question précédente : linéaire en n. □

Q.22

Écrire la fonction récursive `elementDeRang` qui prend en arguments un nombre k entier naturel et l'ensemble X ; et qui renvoie l'élément de rang k dans X, en choisissant comme pivot le premier élément de la liste représentant X.

```
val elementDeRang : int -> ensemble -> int
```

Démonstration.

```
let rec elementDeRang k x =
    match x with
    | [] -> failwith "Vide"
        | p::q ->
                let petits, grands =
                                partitionP p q, partitionG p q in
                let m = card petits in
                if k <= m
                then elementDeRang k petits
                else if k = m+1
        then p
        else elementDeRang (k-m-1) grands
```

□

Le nombre d'opérations pris par la fonction précédente varie en fonction du choix du pivot.

Q.23

Donner un ordre de grandeur, par rapport à n, du nombre maximum $M(n)$ d'opérations pris par la fonction précédente.

Démonstration. En pire cas, on effectue un appel récursif avec un seul élément de moins, comme les opérations sont linéaires dans la longueur des listes, on retrouve une complexité quadratique.
Plus précisément : $M(n) \leq M(n-1) + O(n)$ donc $M(n) = O(n^2)$. □

En supposant équiprobable tous les rangs possibles pour le pivot choisi dans X, on peut démontrer que le temps moyen pris par `elementDeRang` est borné supérieurement par $T(n)$ où T est une fonction croissante, vérifiant $T(0) = 0$ et la formule de récurrence suivante :

$$T(n) \leq \ell n + \frac{1}{n} \sum_{i=1}^{n} \max\{T(i-1), T(n-i)\}$$

(ℓ est une constante indépendante de n et k).

Q.24

En déduire une constante c, qu'on déterminera en fonction de ℓ, telle que pour tout n entier naturel, on a $T(n) \leq cn$.

Démonstration. Un peu pénible vu la manière dont l'énoncé le présente. On peut couper en deux la somme pour résoudre le maximum par croissance de T :
Si $n = 2p + \epsilon$ avec $\epsilon \in \{0, 1\}$, on a

$$\sum_{i=1}^{n} \max(T(i-1), T(n-i)) = \sum_{i=1}^{p} T(n-i) + \epsilon T(p) + \sum_{i=p+1+\epsilon}^{n} T(i-1) = \epsilon T(p) + 2 \sum_{i=p+1+\epsilon}^{n} T(i-1)$$

Si jamais $T(i-1) \leq c(i-1)$, on aurait
Dans le cas où $n = 2p$:

$$\begin{array}{rcl} T(n) & \leq & \ell 2p + \frac{2}{2p} c \sum_{i=p+1}^{n} (i-1) \\ & \leq & \ell 2p + \frac{2}{2p} c \left(\frac{2p(2p-1)}{2} - \frac{p(p+1)}{2} \right) \\ & \leq & \ell 2p + \frac{c}{2}(2(2p-1) - (p+1)) \\ & \leq & \ell 2p + \frac{c}{2} 3(p-1) \\ & \leq & n(\ell + \frac{3c}{4}) \end{array}$$

Dans le cas où $n = 2p + 1$:

$$\begin{array}{rcl} T(n) & \leq & \ell(2p+1) + c\frac{p}{2p+1} + \frac{2c}{2p+1} \sum_{i=p+2}^{2p+1} (i-1) \\ & \leq & \ell(2p+1) + \frac{c}{2p+1}(p + 2p(2p+1) - p(p+1)) \\ & \leq & \ell(2p+1) + \frac{c}{2p+1} p(3p+2) \\ & \leq & n(\ell + \frac{3c}{4}) \end{array}$$

Et on a dans les deux cas, $T(n) \leq cn$ pour $c = 4\ell$. □

On s'intéresse à présent à optimiser le coût dans le pire cas $M(n)$. Considérons d'abord les sous-ensembles

$$X_i = \{e_{5i+1}, e_{5i+2}, e_{5i+3}, e_{5i+4}, e_{5i+5}\}$$

de 5 éléments consécutifs dans X pour $0 \leq 5i + 5 \leq n$.
Soit Y l'ensemble des médians des X_i.

Q.25

Écrire une fonction `medians` qui prend en arguments l'ensemble X ; et qui renvoie l'ensemble Y.

```
val medians : ensemble -> ensemble
```

Quelle est la complexité en temps pris par cette fonction par rapport à n ?

Démonstration. On suppose ici que x est découpable en paquets de 5, sinon on ne comprend pas bien ce qu'on est censé faire :

```
let rec medians x =
        match x with
        | [] -> []
        | x1::x2::x3::x4::x5::q ->
                elementDeRang 3 [x1;x2;x3;x4;x5] :: medians q
```

Comme chaque calcul d'une médiane de 5 peut être supposé en temps constant, on a une complexité linéaire en n. □

Pour améliorer la vitesse de la fonction de sélection, on prend à présent comme pivot le médian de l'ensemble Y.

Q.26

Écrire la fonction `elementDeRangBis` qui prend en entrée un entier naturel k et l'ensemble X ; et qui renvoie l'élément de rang k dans X, en prenant comme pivot le médian de l'ensemble Y.

```
val elementDeRangBis : int -> ensemble -> int
```

Q.27

Montrer que son temps maximum $M'(n)$ vérifie la formule

$$M'(n) \leq \ell' n + M'\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + M'\left(7 \left\lfloor \frac{n}{10} \right\rfloor + 4\right)$$

où $\lfloor x \rfloor$ est la partie entière de x, et ℓ' est une constante que l'on ne cherchera pas à expliciter. En déduire que pour n suffisamment grand, on a $M'(n) \leq c'n$ où c' est une constante à déterminer en fonction de ℓ'.

Q.28

Expliquer pourquoi l'ordre de grandeur du nombre maximal d'opérations pris par la fonction de sélection serait différent si on avait regroupé les éléments de X par groupes de 3, plutôt que par groupes de 5.