Informatique

Devoir surveillé 4

1 février 2022

MP2I 2021/2022

CIV

Durée totale 3h.
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. Utilisez de préférence plusieurs couleurs. **Commentez** votre code dès que ce qu'il fait n'est pas évident.
Avant chaque fonction compliquée à écrire, ou quand vous n'y arrivez pas, indiquez avec des phrases ce que vous voulez faire. Cela permet de vous donner éventuellement des points si l'idée était bonne ou si vous y êtes presque.

1 A link to a past TP

On va reprendre le type des listes chaînées en C :

```c
struct maillon {
    int valeur;
    struct maillon *suivant;
};

typedef struct maillon maillon;

struct liste {
    maillon *debut;
    maillon *fin;
};

typedef struct liste liste;
```

- Dans un maillon, soit `suivant` pointe vers le maillon suivant de la chaîne ou vaut `NULL`.
- La liste vide aura la valeur `NULL` pour `debut` et `fin`.
- Dans une liste non vide, `debut` pointe sur le premier maillon et `fin` sur le dernier.

On suppose définies des fonctions

```c
// cree un maillon
maillon *creer_maillon(int valeur, maillon *suivant);
// supprime un maillon
void suppr_maillon(maillon *);
// cree une liste vide
liste *creer_liste(void);
// supprime une liste
void suppr_liste(liste *);
```

Q01 Écrire une fonction de prototype

```c
int longueur(liste *);
```

qui renvoie la longueur d'une liste.

Q02 Écrire une fonction de prototype

```c
int somme(liste *);
```

qui renvoie la somme des valeurs des éléments d'une liste.

Q03 Écrire une fonction de prototype

```c
void liste_ajout_en_tete(liste *l, int valeur);
```

qui ajoute un maillon en tête de la liste passée en paramètre en $O(1)$.
On fera attention au cas où la liste est vide.

Q04 Écrire une fonction de prototype

```c
maillon *liste_cherche_maillon(liste *l, int valeur);
```

qui renvoie un pointeur sur le premier maillon de valeur `valeur` ou renvoie `NULL` s'il n'y en a pas.

Q05 Écrire une fonction de prototype

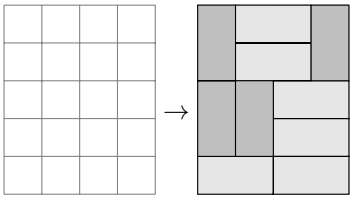
```
void liste_ajout_maillon_apres(liste *l, maillon *m, int valeur);
```

qui insère en $O(1)$ un maillon après le maillon m. On fera en sorte que l'insertion soit valide même si m est le dernier maillon de la liste.

Q06 Expliquer comment cette structure permet d'implémenter une pile et donner les différentes complexités des opérations de la structure liste.

2 Domino's Tiling

On cherche à générer des pavages d'un rectangle de taille $w \times h$ par des dominos, c'est-à-dire des tuiles de tailles 2×2 et sans recouvrement. Un exemple d'un tel pavage est donné ici :



Q07 Donner une condition nécessaire et suffisante sur le couple (w, h) pour qu'un tel pavage existe.
Attention : CNS veut dire qu'il faut donner une condition, montrer qu'elle est vérifiée quand il y a un pavage et montrer que si la condition est vérifiée, alors un pavage existe.

On suppose à partir de maintenant que (w, h) vérifie cette CNS et qu'ainsi il existe un pavage.
On va représenter un telle grille par des cases de trois types :

- `E` si la case est à remplir (`E` comme *empty*)
- `H` si la case appartient à un domino horizontal
- `V` si la case appartient à un domino vertical

```
type case = P | H | V
```

Ainsi, on a perdu la représentation en dominos en les *coupant en deux*. Si on considère le pavage de l'exemple précédent, il correspond à la matrice 'OCaml' :

```
[|
  [| V; H; H; V |];
  [| V; H; H; V |];
  [| V; V; H; H |];
  [| V; V; H; H |];
  [| H; H; H; H |];
|]
```

Cependant, si on considère la première valeur `V` elle ne peut venir que d'un seul domino vertical commençant ici, on peut ainsi procéder à l'envers en l'enlevant :

```
[|
  [| E; H; H; V |];
  [| E; H; H; V |];
  [| V; V; H; H |];
  [| V; V; H; H |];
  [| H; H; H; H |];
|]
```

Et on continue ensuite avec la première case différente de `E` en allant de haut en bas et de gauche à droite.
On identifie un domino à un couple de coordonnées $((x_1, y_1), (x_2, y_2))$ indiquant les cases occupées (l'ordre n'est pas important ici).

```
type domino = (int * int) * (int * int)
```

Q08 Écrire une fonction de signature

```
val extrait_pavage : case array array -> domino list
```

qui extrait la liste des dominos d'une matrice de pavage éventuellement partielle. On pourra modifier la matrice passée en entrée si c'est plus simple pour vous (indice : ce sera le cas).

Pour déterminer un pavage, on va utiliser un algorithme de backtracking. Les mouvements sont des dominos.

Q09 Écrire une fonction de signature

```
val mouvements : case array array -> domino list
```

qui étant donnée une grille renvoie tous les dominos qu'on peut placer.
Attention : même si on a le même type, le problème est différent de la fonction précédente.

Q10 Écrire des fonctions de signature

```
val place : case array array -> domino -> unit
val enleve : case array array -> domino -> unit
```

qui place et enlève respectivement un domino

Q11 Écrire une fonction de signature

```
val complet : case array array -> bool
```

qui indique si une grille est complètement pavée.

Q12 Écrire une fonction récursive de signature

```
val enumere : case array array -> unit
```

qui enumere les pavages de dominos de la grille passée en paramètre et lève l'exception `Solution l` quand la grille correspond à un pavage et `l` est la suite de domino correspondante telle qu'obtenue avec `extrait_pavage`.
L'exception est définie par :

```
exception Solution of domino list
```

Q13 Écrire une fonction de signature :

```
val pavage : int -> int -> domino list
```

telle que `pavage w h` renvoie une liste de domino correspondant à un pavage d'une grille de dimension $w \times h$.

Q14 Reprendre la fonction `enumere` pour en déduire une fonction de signature

```
val enumere_tous : case array array -> domino list list
```

qui renvoie la liste de tous les pavages depuis une grille donnée.
En déduire ainsi une fonction

```
val pavages : int -> int -> domino list list
```

qui renvoie tous les pavages pour une dimension donnée de grille.

3 Matoucheespaceestcassée

Et ma blême araignée, ogre illogique et las
Aimable, aime à régner, au gris logis qu'elle a.
Victor Hugo

On s'intéresse dans cette partie au problème du placement des espaces dans une phrase qui n'en a pas. Par exemple la phrase suivante en anglais

bothearthandsaturnspin

pourrait se décomposer en **both earth and saturn spin** ou **bot heart hands at urns pin**.
On va considérer des `char array` ici pour les chaînes, mais tout marche de la même manière avec des `string`.
On considère un dictionnaire, ou *wordlist*, définie globalement :

```
val dico : char array list
```

Notation : `t[i..j]` est le sous-tableau de `t` constitué des éléments d'indice `i` à `j`.

Q15 Écrire une fonction de signature :

```
val est_mot : char array -> int -> int -> bool
```

tel que l'appel à `est_mot t i j` renvoie `true` lorsque le `t[i..j]` est dans le dictionnaire.

On va résoudre le problème de l'ajout d'espaces par backtracking avec une version plus simple : pour chaque chaîne, on se contente de décider s'il est possible ou non de rajouter des espaces pour qu'elle ne soit composée que de mots dans le dictionnaire. On dit qu'un tableau est **séparable** quand on peut ajouter ces espaces.
On a donc `t` **séparable** s'il existe i tel que `t[0..i-1]` est un mot et `t[i..n-1]` est **séparable** où `|t| = n` est la longueur de t.

Q16 Écrire une fonction récursive de signature :

```
val : separable : char array -> int -> bool
```

telle que `separable t i` indique si `t[i..n-1]` est séparable où $n = |t|$.

Q17 On note T_p le nombre d'appels à `est_mot` dans l'exécution de `separable t i` quand $p = |t| - i$ est le nombre d'élément du sous-tableau considéré.

1. Montrer que $T_p \leq \sum_{i=0}^{p-1} T_i + O(p)$
2. Quitte à se placer dans un pire cas, on peut supposer qu'il existe une constante A telle que $T_p = \sum_{i=0}^{p-1} T_i + Ap$. Montrer que pour $p > 0$, $T_p = 2T_{p-1} + A$.
3. En déduire que $T_p = O(2^p)$.

