Concours Blanc MPSI : Option Informatique (Durée : 4 heures)

Chaque programme devra être écrite dans le langage OCaml. On veillera à présenter et indenter de manière lisible chacun d'entre eux. Vous pouvez utiliser `(* *)` pour insérer des commentaires. Vous pouvez utiliser une fonction dans les fonctions suivantes même si celle-ci n'a pas été écrite. Quand on demande une complexité il s'agit de la complexité temporelle. Les calculatrices sont interdites.

Questions de cours/TP : Implémentation d'une pile

1. En représentant une pile par une liste, écrire des fonctions réalisant les opérations suivantes, et donner leurs coûts :

   - Tester si la pile est vide.

   - Ajout d'un élément au sommet de la pile (on renverra une nouvelle pile).

   - Enlever l'élément au sommet de la pile et récupérer sa valeur (on renverra le couple (élément, nouvelle pile)).

2. Reprendre la question précédente en implémentant la pile par un vecteur (sa taille sera donc bornée, on ne pourra pas ajouter d'élément si le vecteur est plein). On devra mémoriser l'indice du sommet de pile (par convention la pile vide a un sommet d'indice -1), on utilisera donc le type enregistrement :

   ```
   type 'a pile = {mutable sommet :  int ; stack :  'a array} ;;
   ```

   Cette fois les fonctions devront modifier la pile donnée en entrée.

3. Comparer les deux implémentations.

Exercice 1

Dans une classe de n élèves, les élèves sont numérotés de 0 à $n-1$. Un professeur souhaite faire l'appel, c'est à dire déterminer quels élèves sont absents.

Partie A

1. Écrire une fonction `mini :  int list` $\to$ `(int * int list)` qui prend en argument une liste non vide d'entiers distincts, et renvoie le plus petit élément de cette liste, ainsi que la liste de départ privée de cet élément (pas forcément dans l'ordre initial).

2. En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de comparaisons de la fonction précédente ?

3. Écrire une fonction `liste :  int` $\to$ `int` $\to$ `int list` telle que `liste i j` renvoie la liste constituée des entiers entre i et $j-1$ (on renverra la liste vide si $i \geqslant j$). Donner sa complexité en fonction de i et j.

4. En utilisant les fonctions `mini` et `liste`, écrire une fonction `absents : int list → int → int list` qui, étant donnés une liste non vide d'entiers distincts et un entier n, renvoie, dans un ordre quelconque, la liste des entiers de $[0; n-1]$ qui n'y sont pas. On pourra écrire une fonction auxiliaire récursive `aux l i j` qui renvoie la liste des entiers entre i et $j-1$ absents de l.

5. En notant k la longueur de la liste donnée en argument, quelle est la complexité dans le pire des cas en nombre de comparaisons (en fonction de n et k) de la fonction précédente ?

Partie B

Dans cette partie, une salle de classe pour n élèves est décrite par la donnée d'un tableau à n entrées. Si `tab` est un tel tableau et `i` un entier de $[0; n-1]$, alors `tab.(i)` donne le numéro de l'élève assis à la place i (ou -1 si cette place est vide).

1. Écrire une fonction `asseoir : int list → int → int array`, qui prend en argument une liste non vide d'entiers distincts et un entier n, et renvoie un tableau représentant une salle de classe pour n élèves où chaque élève de la liste à été assis à la place numérotée par son propre numéro. Les entiers supérieurs ou égaux à n seront ignorés.

2. En déduire une fonction `absent2 : int list → int → int list` qui étant donné une liste non vide d'entiers distincts et un entier n, renvoie la liste des entiers de $[0; n-1]$ qui n'y sont pas. Les entiers supérieurs ou égaux à n seront ignorés.

3. En notant k la longueur de la liste donnée en argument, quelle est la complexité en nombre de lectures et d'écritures dans un tableau (en fonction de n et k) de la fonction précédente ?

Partie C

Dans cette partie indépendante des précédentes, les élèves sont déjà assis en classe.

1. On considère la fonction `place : int array → int → unit` suivante :

```
let rec place tab i =
   if i <> -1 then
   begin
      let temp=tab.(i) in
      tab.(i) <- i;
      place tab temp
   end;;
```

   On note `classe` le tableau `[|-1;4;5;-1;3;0|]` (on suppose $n = 6$ dans cette question). Donner le tableau `classe` après l'exécution de `place classe 1`. On donnera l'état de la variable `classe` à chaque appel récursif de la fonction.

2. On considère la fonction `placement : int array → int → unit` ci-dessous :

```
let placement tab n =
   for i=0 to n-1 do
      if tab.(i) <> -1 && tab.(i) <>i then
      begin
         let temp=tab.(i) in
         tab.(i) <- -1;
         place tab temp
      end
   done;;
```

Si `classe` est un tableau d'entiers (les entiers positifs sont distincts) de taille n représentant une classe, que fait `placement classe n` ?

Exercice 2

On suppose défini le type arbre de la manière suivante :

```
type arbre =
   |Feuille of int
   |Noeud of arbre * arbre ;;
```

Remarquons qu'avec cette représentation, seules les feuilles portent une étiquette (qui est un entier) et les nœuds internes n'en ont pas. On dit qu'un arbre est un *peigne* si tous les noeuds à l'exception éventuelle de la racine ont au moins une feuille pour fils. On dit qu'un peigne est un peigne *strict* si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est *rangé* si le fils droit d'un noeud est toujours une feuille. La figure 1 donne l'exemple d'un peigne à 5 feuilles.

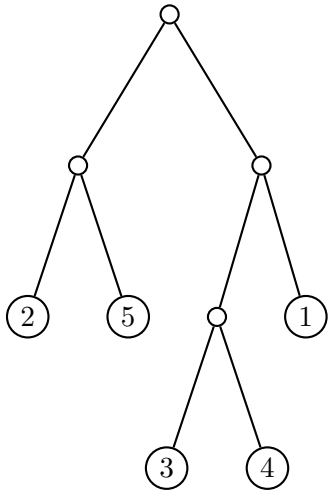

Figure 1: Un peigne à 5 feuilles

1. Représenter un peigne rangé à 5 feuilles.
2. La *hauteur* d'un arbre est le nombre de noeuds maximal que l'on rencontre pour aller de la racine à une feuille (la hauteur d'une feuille seule est 0). Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.
3. Écrire une fonction `est_range : arbre` $\rightarrow$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé (indication : raisonner récursivement !).
4. Écrire une fonction `est_peigne_strict : arbre` $\rightarrow$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict. En déduire une fonction `est_peigne : arbre` $\rightarrow$ `bool` qui renvoie `true` si l'arbre donné en argument est un peigne.

   Indication : On pourra remarquer qu'un arbre est un peigne si et seulement si c'est une feuille ou un arbre dont les deux fils sont des arbres peignes stricts.
5. On souhaite ranger un peigne donné. Supposons que le fils droit N de sa racine soit un noeud possédant au moins une feuille. Notons A_1 le sous-arbre gauche de la racine, f l'une des feuilles du noeud N et A_2 l'autre sous-arbre du noeud N. On va utiliser l'opération de *rotation* qui construit un nouveau peigne où

- le fils droit de la racine est le sous-arbre A_2 ;
- le fils gauche de la racine est un noeud de sous-arbre gauche A_1 et de sous-rabre droit la feuille f.

La figure 2 illustre le principe d'une rotation.

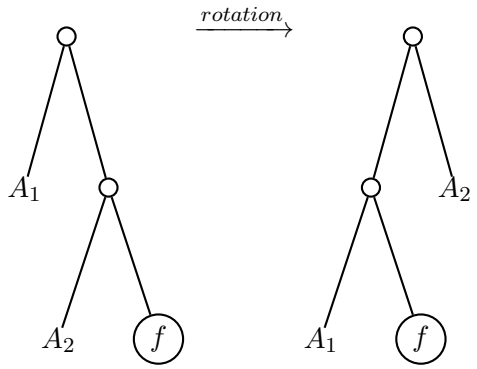

Figure 2: Schéma d'une rotation

(a) Donner le résultat d'une rotation sur l'arbre de la figure 1.

(b) Écrire une fonction `rotation : arbre → arbre` qui effectue l'opération décrite ci-dessus. La fonction renverra l'arbre initial si une rotation n'est pas possible.

(c) Décrire un algorithme permettant de ranger un peigne, c'est-à-dire le transformer en peigne rangé ayant les mêmes feuilles. Appliquer votre algorithme au peigne de la figure 1. Justifier sa terminaison.

(d) Écrire une fonction `rangement : arbre → arbre` qui réalise l'algorithme précédent et renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

Exercice 3

On rappelle la définition de la suite de Fibonacci:

$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \quad \text{pour} \quad n \geqslant 2$$

Partie A. Calcul des termes de la suite

1. On considère la fonction `fibo : int → int` suivante :

```
let rec fibo = function
|0->0
|1->1
|n->fibo (n-1) + fibo (n-2);;
```

   Pourquoi est-ce une mauvaise idée d'utiliser cette fonction pour calculer F_n ?

2. Écrire une fonction `fibo2 : int → int` qui, étant donné n, calcule F_n en effectuant un nombre linéaire en n d'additions.

Partie B. Décomposition de Zeckendorf

Pour $n \in \mathbb{N}$, on appelle décomposition de Zeckendorf de n une décomposition de n comme somme de termes distincts (d'indices supérieurs ou égal à 2) de la suite de Fibonacci, de telle manière qu'il n'y ait pas deux termes d'indices consécutifs dans la somme. Autrement dit, il existe des indices $c_1, c_2, \ldots, c_k$ tels que :

- $c_0 \geq 2$,
- pour tout $i < k$, $c_{i+1} > c_i + 1$ (pas d'indices consécutifs),
- $n = \sum_{i=0}^{k} F_{c_i}$.

Par exemple, $2 + 5 = F_3 + F_5$ est une décomposition de Zeckendorf de 7 (avec $c_0 = 3$ et $c_1 = 5$), alors que $3 + 5 = F_4 + F_5$ n'est pas une décomposition de Zeckendorf de 8 car F_4 et F_5 sont deux termes consécutifs de la suite de Fibonacci.

1. Déterminer une décomposition de Zeckendorf de 20, 21 et 22.
2. Montrer que tout entier strictement positif admet une décomposition de Zeckendorf (on admet qu'elle est unique).

Partie C. Codage de Fibonacci

On appelle codage de Fibonacci d'un entier positif k un tableau `tab` de 0 et de 1 indiquant par des 1 les indices des termes de la suite de Fibonacci utilisés dans la décomposition de Zeckendorf de k (`tab`.(0) indique alors si F_2 est utilisé dans la représentation). On remarque que par définition de la décomposition de Zeckendorf, `tab` ne peut contenir deux 1 consécutifs.

$[|0; 1; 0; 1|]$ est ainsi un codage de Fibonacci de 7

1. Écrire une fonction `decode : int array` $\rightarrow$ `int` qui traduit en entier une représentation de Fibonacci d'un entier.
2. (a) Décrire sans l'implémenter une fonction `plusun : int array` $\rightarrow$ `int array`, qui à partir d'une représentation de Fibonacci d'un entier k, renvoie une représentation de Fibonacci de l'entier $k + 1$.
   (b) Décrire sans l'implémenter une fonction `moinsun : int array` $\rightarrow$ `int array`, qui à partir d'une représentation de Fibonacci d'un entier k non nul, renvoie une représentation de Fibonacci de l'entier $k - 1$.