Option Informatique

Devoir Maison *n*°01

Pour le 22/04/2024.

Remarque : les fonctions que vous écrivez seront précédées d'une explication de votre code, succinte mais claire.

Exercice 1 : tri par dénombrement

Dans cet exercice, on souhaite trier des **listes d'entiers positifs**, donc des `int list` qui vérifient de plus que tous leurs éléments sont positifs. On propose la stratégie suivante :

- On détermine le max m des éléments de ℓ.
- On crée un tableau t de longueur $m+1$ rempli de 0.
- On modifie t de sorte que les $t.(i)$ contiennent le nombre de fois que i apparaît dans ℓ.
- On reconstitue une liste triée en parcourant t.

Par exemple, si on souhaite trier la liste `l = [5;2;1;2;5;5;2;2]`, on commence par déterminer que l'élément maximal de `l` est 5, puis on crée le tableau `t=[|0;0;0;0;0;0|]`, puis on le modifie en `t=[|0;1;4;0;0;3|]`, et enfin on parcourt `t` pour produire la liste triée `[1;2;2;2;2;5;5;5]`.

Q1. Écrire une fonction `maximum : int list → int` qui prend comme argument une liste d'entiers `l` et retourne comme résultat le plus grand élément de `l`. De plus, si la liste `l` n'est pas formée exclusivement d'entiers positifs, votre fonction devra provoquer une exception.

Q2. Écrire une fonction `decompte : int list → int array → unit` qui prend comme argument une liste `l` et un tableau `t`, et ne retourne rien mais a pour effet de bord d'incrémenter chaque case `t.(i)` du tableau à chaque apparition de `i` dans `l`.

Q3. Finalement, écrire une fonction `tri_tiroirs : int list → int list` qui trie son argument à l'aide de l'algorithme décrit au début de l'exercice.

Q4. On fait les hypothèses suivantes : on estimera qu'une comparaison d'entiers a une complexité constante, que la création d'un tableau de longueur k a une complexité en $O(k)$, et que les opérations d'accès à une case et de modification d'une case d'un tableau sont de complexité constante.

(a) Déterminer la complexité de la fonction `tri_tiroirs` en fonction de la longueur `n` de son argument et du plus grand élément `m` de son argument.

(b) Pour quel type de listes d'entiers positifs précisément ce tri est-il adapté ? On n'hésitera pas à le comparer à d'autres algorithmes de tri connus.

Exercice 2 : Chemins et promenades dans un arbre

On définit le type `'a arbre` comme suit :

```
type 'a arbre =
| Feuille of 'a
| Noeud of 'a * ('a arbre) * ('a arbre);;
```

Les feuilles seront considérées comme ces cas particulier de nœuds. Le premier nœud de l'arbre est appelé sa racine ; s'il ne s'agit pas d'une feuille, ses deux fils sont appelés le fils gauche et le fils droit de l'arbre.

On rappelle que la hauteur $h(a)$ d'un arbre a est définie comme étant égale à 0 si a est une feuille, et comme $1 + \max\big(h(fg), h(fd)\big)$ si a est un nœud de fils gauche fg et de fils droit fd.

Q5. Écrire une fonction `hauteur : 'a arbre → int` permettant de calculer la hauteur d'un arbre. Votre fonction devra visiter au plus une fois chaque nœud de l'arbre (cela signifie que, pour chaque nœud n de l'arbre, au plus un appel récursif de toute fonction utilisée dans le calcul devra se faire sur le sous-arbre dont la racine est n).

Définition : On appelle **chemin** dans l'arbre une liste non vide $[n_{k+1}; n_k; \ldots; n_1]$ de nœuds tels que, pour tout $i \in \{1, \ldots, k\}$, n_i est l'un des deux fils de n_{i+1}. L'entier k est alors appelé la **longueur** du chemin.

En voyant l'arbre comme un graphe, la longueur d'un chemin est le nombre d'arêtes empruntées par le chemin.

Q6. Montrer par induction structurelle que la hauteur de l'arbre est la plus grande longueur d'un chemin dans l'arbre (on pourra admettre qu'un chemin réalisant cette longueur a nécessairement pour premier nœud la racine de l'arbre et pour dernier nœud une feuille).

Définition : On appelle **promenade** dans l'arbre une liste non vide $[n_1; \ldots; n_p; s; m_q; \ldots; m_1]$ de nœuds tels que $n_p \neq m_q$, n_1 et m_1 sont des feuilles de l'arbre, et $[s; n_p; \ldots; n_1]$ et $[s; m_q; \ldots; m_1]$ soient deux chemins de l'arbre. L'entier $p + q$ est appelé la **longueur** de la promenade.

Autrement dit, une promenade est une liste de nœuds dont le premier est une feuille, dont les nœuds suivants s'obtiennent en prenant le père du nœud précédent jusqu'à arriver à un certain nœud s (pas nécessairement la racine), puis dont les nœuds suivants s'obtiennent en prenant un des deux fils du nœud précédent jusqu'à arriver à une feuille, la promenade ne revenant jamais sur ses pas. En voyant l'arbre comme un graphe, la longueur d'une promenade est le nombre d'arêtes empruntées pendant la promenade.

On souhaite déterminer la longueur maximale `lplp a` (pour "**l**ongueur d'une **p**lus **l**ongue **p**romenade") d'une promenade dans un arbre `a`. On va utiliser une stratégie de type *diviser pour régner*.

Q7. Si `a=(Feuille f)` est une feuille, donner sans démonstration `lplp a`. Si `a=(Noeud (n,fg,fd))` n'est pas une feuille, donner `lplp a` en fonction de `lplp fg`, `lplp fd`, `hauteur fg` et `hauteur fd`.

Q8. En utilisant votre fonction `hauteur` écrite précédemment, écrire une première version de la fonction `lplp : 'a arbre → int`. Justifier brièvement que votre fonction `lplp` ne visite **pas** une seule fois chaque nœud de son argument.

Q9. Adapter votre fonction pour qu'elle visite une seule fois chaque nœud de son argument.

On rappelle que le type `'a option` est défini par

```
type 'a option =
|None
|Some of 'a;;
```

Il s'agit donc du type des termes de type `'a`, auxquels on a rajouté le terme vide.

On souhaite maintenant coder les arbres à l'aide d'une structure impérative. Un `'a arbre`, disons `a`, de hauteur h sera codé par `t`, un `'a option array` de longueur $2^{h+1}-1$ de la façon suivante : `t.(0)` sera `Some x` où x est l'étiquette de la racine de l'arbre, puis, pour tout $k < 2^h - 1$, on aura trois cas :

- si `t.(k)` code l'étiquette d'une feuille de l'arbre, alors `t.(2k+1)` et `t.(2k+2)` devront être `None`.
- si `t.(k)` est `None` alors `t.(2k+1)` et `t.(2k+2)` devront être `None`.
- si `t.(k)` code l'étiquette d'un nœud de l'arbre qui n'est pas une feuille, alors `t.(2k+1)` et `t.(2k+2)` devront coder les étiquettes des deux fils de ce nœud.

Q10. Écrire une fonction `convertir : 'a arbre → 'a option array` permettant de réaliser cette conversion.

On souhaite maintenant expliciter une promenade de longueur maximale, ou plus précisément la liste des étiquettes des nœuds d'une telle promenade.

Q11. Écrire une fonction `plp : 'a arbre → 'a list` réalisant ceci.