Option Informatique MP – DS02 v5

Durée 4h.

Le devoir se compose de deux exercices et deux problèmes, totalement indépendants.

Vous devez traiter les questions avec soin et clarté.

Les programmes doivent être commentés suffisamment pour être lisibles sans effort.

Les exercices peuvent être traités dans un ordre quelconque, mais il ne faut pas les enchevêtrer, prenez des copies séparées si nécessaire.

Il n’est pas du tout nécessaire de tout traiter pour réussir.

Exercice 1 [E3A]

On suppose défini le type arbre de la manière suivante :

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

On dit qu'un arbre est un *peigne* si tous les nœuds à l'exception éventuelle de la racine ont au moins une feuille pour fils.
On dit qu'un peigne est *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 nœud est toujours une feuille.
Un arbre réduit à une feuille est un peigne rangé.

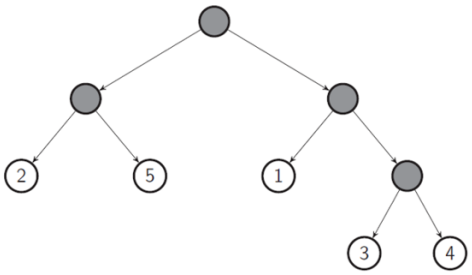


Figure 1 – Un peigne à cinq feuilles.

1. Représenter un peigne rangé à 5 feuilles.

2. La *hauteur* d'un arbre est le nombre de nœuds 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 ($n \geq 1$) ? On justifiera la réponse.

3. Écrire une fonction `est_range : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.

4. Écrire une fonction `est_range_strict : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict.
En déduire une fonction `est_peigne : arbre -> bool` qui renvoie `true` si l'arbre donné en argument est un peigne.

5. On souhaite ranger un peigne donné. Supposons que le fils droit N de sa racine ne soit pas une feuille. Notons A_1 le sous-arbre gauche de la racine, f l'une des feuilles du nœud N et A_2 l'autre sous-arbre du nœud N. On va utiliser l'opérateur 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 nœud de sous-arbre gauche A_1, et de sous-arbre droit la feuille f.

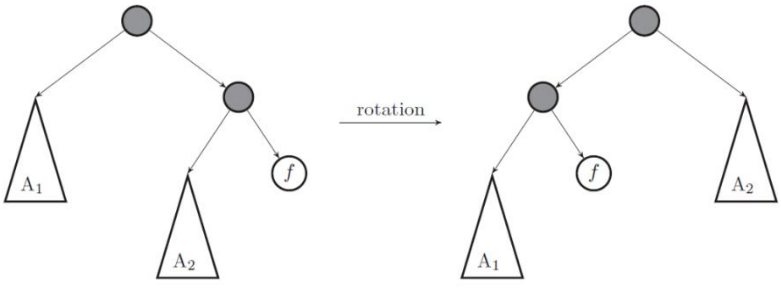


Figure 2 – Une rotation.

(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) Écrire une fonction `rangement : arbre -> arbre` qui range un peigne donné en argument, c'est-à-dire qu'il 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.

(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) Écrire une fonction `rangement : arbre -> arbre` qui range un peigne donné en argument, c'est à dire qu'il 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 2

Nous souhaitons classer les fromages français dans un arbre : les fromages les plus proches seront sur des branches "voisines" de l'arbre. Dans ce but est défini un type d'arbre étiqueté : `type arbre = Fromage of int | N of arbre * arbre * float;;`

Pour chaque fromage nous disposons d'un ensemble de mesures : apport énergétique, teneur en calcium, en protéines, ...

Nom	Apport énergétique (en kJ/100g)	Calcium (en mg/100g)	...
Camembert	1150	333	...
Maroilles	348	335	...
Roquefort	374	601	...
⋮	⋮	⋮	⋮

Le tableau précédent est représenté en Caml par une matrice globale `fromages` de flottants. Par exemple, `fromages.(0).(1)` donne `333.0`, la teneur en calcium (colonne numéro 1) du camembert (fromage numéro 0). Chaque fromage f est alors représenté par une ligne $[|f_0;\dots;f_q|]$ de la matrice `fromages`.

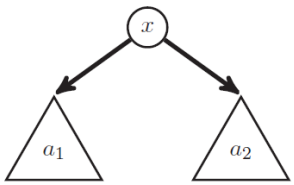
Étant donné un tableau global `ponderation`$= [|\alpha_0;\dots;\alpha_q|]$ de flottants strictement positifs on définit la distance $d(a,b)$ entre deux fromages $a = [|x_0;\dots;x_q|]$ et $b = [|y_0;\dots;y_q|]$ par $d(a,b) = \sum_{k=0}^{q} \alpha_k |x_k - y_k|$, puis on définit la distance $d(A,B)$ entre deux ensembles de fromages A et B par

$$d(A,B) = \min_{a\in A \text{ et } b\in B} d(a,b)$$

Enfin, on définit la distance entre deux arbres a_1 et a_2 comme la distance entre l'ensemble des fromages présents dans a_1 et l'ensemble des fromages présents dans a_2.

1. Écrire une fonction `distance : int -> int -> float`.
   `distance i j` calcule la distance entre les fromages de numéros `i` et `j`.
2. Définir en Caml une matrice globale `dist`, telle que, pour tous fromages `i` et `j`, `dist.(i).(j)` est égal à la distance entre les fromages `i` et `j`.
3. Écrire une fonction `dist_arbr : arbre -> arbre -> float` qui étant donné deux arbres de fromages, calcule la distance entre ces deux arbres.

4. La fusion de deux arbres a_1 et a_2 est un arbre de la forme ci-après où la racine est étiquetée par x, la distance entre a_1 et a_2.



On appelle *forêt* une liste d'arbres. Un dendrogramme est un arbre construit de la manière suivante.

(a) On crée une forêt contenant un arbre de la forme `Fromage(i)` par fromage.
(b) Tant qu'il y a plusieurs arbres dans la forêt, on cherche deux arbres séparés par la plus petite distance possible, puis on les fusionne.
(c) Lorsqu'il ne reste plus qu'un seul arbre, on renvoie celui-là.

Écrire une fonction `dendrogramme : unit -> arbre` qui crée le dendrogramme de tous les fromages référencés dans la matrice globale `fromages`.

Problème 1

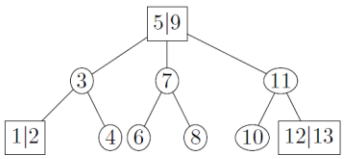
Un B-arbre 2-3 est un arbre qui est soit vide, soit composé

- de feuilles ayant une ou deux étiquettes
- de noeuds ayant une étiquette et deux fils
- de noeuds ayant deux étiquettes et trois fils.

Si a est un B-arbre, on note de façon classique $\mathcal{E}(a)$ l'ensemble des étiquettes de a.
Ils sont organisés comme les Arbres Binaires de Recherche, c'est-à-dire:

- quand une feuille a deux étiquettes, la plus petite est à gauche.
- quand un noeud a une étiquette e et deux fils f_1 et f_2 :
  $\forall x \in \mathcal{E}(f_1)\, \forall y \in \mathcal{E}(f_2)\ : x \leqslant e \leqslant y$
- quand un noeud a deux étiquettes e_1 ,e_2 et trois fils f_1,f_2 et f_3 :
  $\forall x \in \mathcal{E}(f_1)\, \forall y \in \mathcal{E}(f_2) \forall z \in \mathcal{E}(f_3)\ : x \leqslant e_1 \leqslant y \leqslant e_2 \leqslant z$
- de plus dans un b-arbre toutes les feuilles sont à la même profondeur.

Exemple de B-arbre



On utilisera le type suivant:

```
type barbre =
  | Nil
  | Feuille1 of int
  | Feuille2 of (int*int)
  | Noeud1 of (barbre * int * barbre)
  | Noeud2 of (barbre * int * barbre * int * barbre) ;;
```

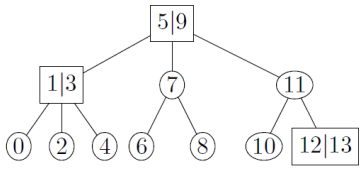
1. Définir en Caml l'arbre précédent en utilisant le type ci-dessus, on l'appellera `a`.

La recherche se fait comme dans un arbre binaire de recherche.

L'insertion se fait dans les feuilles de l'arbre suivant le procédé suivant:

- On cherche une feuille où insérer l'élément.
- Si on trouve une feuille à une étiquette, on la transforme en feuille à deux étiquettes.
- Si on trouve une feuille à deux étiquettes on fait un nouvel arbre que l'on essaie de «greffer» au père.
- Si le père n'a qu'une seule étiquette cela ne pose pas de problème, par contre si le père a déjà deux étiquettes, on refait un nouvel arbre ayant l'élément médian comme racine que l'on essaie de «greffer» plus haut.

Par exemple si on ajoute 0 à l'arbre précédent on trouve:



Quand on essaie de «greffer» un arbre à une racine ayant déjà deux étiquettes, on obtient un nouvel arbre avec un étage de plus.

Pour réaliser l'insertion on écrira une fonction auxiliaire récursive:

`insere : int -> barbre -> barbre*int*barbre`.

Quand le résultat de la fonction auxiliaire `insere` est sous la forme `(a,e,Nil)` cela veut dire qu'il n'y a pas d'étiquette qui remonte, par contre si le résultat est de la forme ; `(f1,e,f2)` alors il faut «greffer» l'arbre d'étiquette `e` et de fils `f1` et `f2`.

Écrire les fonctions

2. de calcul de la hauteur ; la tester sur l'arbre `a` précédent ;
3. de calcul du nombre de noeuds (remarque : les feuilles sont aussi comptées comme noeuds, et un noeud portant 2 étiquettes compte pour 1, par exemple l'arbre `a` possède 10 feuilles) ; la tester sur l'arbre `a` ;
4. de calcul du nombre total d'éléments présents (une feuille ou un noeud à 2 étiquettes compte pour 2, par exemple l'arbre `a` possède 13 éléments au total) ; la tester sur l'arbre `a` ;
5. de recherche dans un B-arbre ; effectuer les recherches de 8 et 14 dans l'arbre `a` ;
6. de parcours infixe du B-arbre ; la tester sur l'arbre `a` ;
7. d'insertion dans un B-arbre ; effectuer l'insertion de 0 dans l'arbre `a`.

Mise à plat

La mise à plat d'un arbre est sa transformation réversible en une liste, par exemple pour pouvoir le mémoriser dans un fichier.

Une feuille sera codée ainsi: la liste de ses étiquettes dans l'ordre suivi d'un symbole de Fin,

Un noeud intérieur sera codé ainsi: la liste de ses étiquettes dans l'ordre suivi d'un symbole de Séparation, suivi des listes codant les fils.

On utilisera pour les éléments de la liste le typage suivant:

```
type arbre_plat = Entier of int | Fin | Sep;;
```

Pour l'exemple, la mise à plat de `a` donne:

```
[Entier 5; Entier 9; Sep; Entier 3; Sep; Entier 1; Entier 2; Fin; Entier 4; Fin;
Entier 7; Sep; Entier 6; Fin; Entier 8; Fin; Entier 11; Sep; Entier 10; Fin; Entier
12; Entier 13; Fin]
```

8. Écrire la fonction de mise à plat d'un B-arbre. La tester sur l'arbre `a`, en nommant `l` la liste obtenue.
9. Écrire la fonction de décodage (c'est-à-dire qui reconstruit l'arbre à partir de la mise à plat). On pourra utiliser une fonction auxiliaire récursive qui construit une première feuille et la renvoie avec le reste de la liste. La tester sur la liste `l` et vérifier que l'on retrouve bien l'arbre `a`.

Problème 2 [E3A 2015 – exercice 3]

On souhaite analyser les résultats de sondages concernant une élection.
Les candidats à l'élection sont numérotés de 0 à $k-1$. Les résultats de chaque sondage sont stockés dans un tableau. Si le sondage a recueilli N réponses, le tableau comporte N cases, une pour chaque réponse ; la $i^{\text{ème}}$ case du tableau contient le numéro du candidat proposé par la $i^{\text{ème}}$ personne sondée. On a ainsi un tableau T de longueur N qui contient des entiers naturels entre 0 et $k-1$.
Le sondage donne le candidat numéroté i élu si le nombre i est dans strictement plus de $N/2$ cases de T. Par exemple, un sondage correspondant au tableau $[2,4,5,0,4,4,4]$ donne le candidat 4 élu. Mais un tableau ne donne pas toujours un élu. Par exemple, le tableau $[1,2,3,4,6,2,3,3]$.

On suppose qu'un entier k, $k \geq 1$, est défini.

1. (a) Écrire une fonction `nb` de type `int array -> int -> int` telle que si a est un entier naturel et `tab` un tableau de taille N issu d'un tel sondage, `nb tab a` est le nombre de cases du tableau `tab` qui contiennent a.
(b) Évaluer la complexité d'un appel à `nb` en fonction de N et de k.
(c) En déduire une fonction `elu1` de type `int array -> int` telle que `elu1 tab` est l'entier donné élu par tableau `tab` si celui-ci existe et -1 sinon.
(d) Évaluer la complexité d'un appel à `elu1` en fonction de N et de k.

2. On se propose d'utiliser la stratégie diviser pour régner pour déterminer l'éventuel élu donné par un tableau. On dispose de deux fonctions, qu'on ne cherchera pas à décrire :

   - `miGauche : int vect -> int vect` qui prend en argument un tableau `tab` de longueur $N \geqslant 2$ et retourne le tableau de longueur $\lfloor N/2 \rfloor$ formé par les $\lfloor N/2 \rfloor$ premières cases de `tab`,
   - `miDroite : int vect -> int vect` qui prend en argument un tableau `tab` de longueur $N \geqslant 2$ et retourne le tableau de longueur $N - \lfloor N/2 \rfloor$ formé par les $N - \lfloor N/2 \rfloor$ dernières cases de `tab`.

   (a) Soit `tab` un tableau de longueur $N \geqslant 2$. Démontrer que si `tab` donne a comme élu alors celui-ci est aussi donné élu par le tableau `miGauche tab` ou par le tableau `miDroite tab`.
   (b) Proposer une fonction `elu2`, utilisant la stratégie diviser pour régner, de type `int vect -> int * int` ; si `tab` est un tableau, `elu2 tab` est le couple `(a,n)` si l'entier `a` est donné élu par `tab` et apparait dans `n` cases exactement de `tab`, et `(-1,0)` si `tab` ne donne pas d'élu. On pourra utiliser la fonction `nb` définie en 1(a).
   (c) Evaluer la complexité de cette fonction en fonction de N.

3. Soit T un tableau de longueur N. On dit que le nombre entier a est un *postulant* pour la valeur n du tableau T si : n est un entier strictement supérieur à $N/2$ tel que a apparait au plus (au sens large) n fois dans T et tout entier b distinct de a, apparait au plus (au sens large) $N-n$ fois dans T. Par exemple, 3 est un postulant pour $n=5$ du tableau $[1,2,3,4,3,2,3,3]$.

   On dit que le nombre entier a est un postulant du tableau T s'il existe un nombre entier $n > N/2$ tel que a est un postulant pour la valeur n du tableau T

   (a) Démontrer que si le tableau T donne a élu alors a est un postulant de T.
   (b) Démontrer que si a est un postulant de T, alors aucun autre élément de T ne pourrait être donné comme élu.
   (c) Donner un exemple de tableau qui contient un postulant mais ne donne aucun élu et un exemple de tableau n'ayant aucun postulant.
   (d) Soit T un tableau de longueur un entier pair N. On note TG le tableau de longueur $N/2$ formé par les $N/2$ premières cases de T et TD le tableau de longueur $N/2$ formé par les $N/2$ dernières cases de T.

   i. On suppose que le tableau TD ne donne pas d'élu. Soit a un postulant pour la valeur l du tableau TG. Démontrer que a est un postulant de T. On exprimera la valeur d'un entier n tel que $n > N/2$ et a est un postulant pour la valeur n du tableau T en fonction de l et N.

ii. Soient a un postulant pour la valeur l du tableau TG et b un postulant pour la valeur m du tableau TD.

A. On suppose que $a = b$. Démontrer que a est un postulant de T. On exprimera la valeur d'un entier n tel que $n > N/2$ et a est un postulant pour la valeur n du tableau T en fonction de l et m.

B. On suppose que $a \neq b$ et $m > l$. Démontrer que b est un postulant pour la valeur $N/2 + m - l$ de T.

C. On suppose que $a \neq b$ et $m = l$. Démontrer que T ne donne pas d'élu.

(e) Ecrire une fonction `postulant : int vect -> int * int` telle que, si `tab` est un tableau d'entiers naturels de longueur N, `postulant tab` est un couple `(a,n)` tel que

- lorsque `postulant tab` renvoie le couple `(-1,0)`, le tableau `tab` n'a pas d'élu,
- lorsque `postulant tab` renvoie le couple `(a,n)` avec `n` $> N/2$, `a` est un postulant pour la valeur `n` du tableau `tab`. On supposera pour simplifier que la taille du tableau est une puissance de 2. La procédure utilisera la stratégie diviser pour régner et aura une complexité linéaire, ce qu'on justifiera précisément.

(f) En déduire une fonction `elu3`, de type `int vect -> int` qui prend en argument un tableau `tab` et retourne l'entier élu de `tab` si celui-ci existe et `-1` sinon, de complexité linéaire.