info401 : Programmation fonctionnelle
TP 2 : arbre et tas, crible d'Eratosthene

Pierre Hyvernat
Laboratoire de mathématiques de l'université de Savoie
bâtiment Chablais, bureau 22, poste : 94 22
email : Pierre.Hyvernat@univ-savoie.fr
www : http://www.lama.univ-savoie.fr/~hyvernat/

Ce TP, comme les suivants, sera noté. Je ne demande aucun compte rendu à part, mais seulement un *unique* fichier Caml, commenté.

- Votre fichier doit contenir du code valide et ne doit pas provoquer d'erreur lorsqu'on l'évalue avec l'interprète Caml. (Testez avant de me l'envoyer : si votre fichier ne s'évalue pas correctement, vous perdez automatiquement 5 points sur votre note finale.)
- Votre fichier devra contenir un commentaire contenant votre nom, prénom et filière. (Idem, si ce n'est pas le cas, vous perdez 5 points sur votre note finale.)

Partie 1 : les tas (2h)

Un *tas* est une structure de donnée permettant d'optimiser la recherche du minimum dans un ensemble de données.

Un *tas* est simplement un arbre binaire qui vérifie la propriété inductive suivante :
- l'arbre vide est un tas,
- sinon, la racine est le plus petit élément,
- et les deux fils sont des tas.

Contrairement aux arbres binaires de recherche, cette structure ne permet pas d'optimiser la recherche d'un élément car on ne sait pas dans quel sous-arbre il faut chercher.

Question 1. Déclarez un type `'a arbre` pour les arbres binaires ; et programmez les fonctions suivantes :
- `taille : 'a arbre -> int`,
- `profondeur : 'a arbre -> int` qui calcule la longueur d'une plus grande branche.

Question 2. Programmez une fonction `equilibre : 'a arbre -> bool` qui vérifie si un arbre est équilibré (càd si la différence de longueurs entre la branche la plus longue et la branche la plus courte est au plus 1).

Question 3. Déclarez un type `'a tas` pour les tas et écrivez une fonction qui teste si un élément de ce type est effectivement un tas.

Question 4. Écrivez une fonction `tas_equilibre : 'a tas -> bool` qui vérifie si un arbre est un tas équilibré.

Question 5. Écrivez une fonction `min_tas : 'a tas -> 'a` qui permettra de calculer le minimum d'un tas.

Question 6. Écrivez une fonction `pop_tas : 'a tas -> 'a tas` permet de supprimer le le minimum d'un tas (tout en conservant la propriété d'être un tas).

Remarque : attention à gérer les cas problématiques correctement.

L'insertion dans un arbre de recherche équilibré est un peu compliquée (mais pas trop). Pour les tas, on peut utiliser le fait que les fils gauche et droit peuvent être permutés pour créer des tas équilibrés !

Question 7. Écrivez la fonction `insere : 'a -> 'a tas -> 'a tas` qui insère un élément dans un tas.

Cette fonction devra toujours insérer l'élément dans le fils droit, mais elle permutera le fils droit et le fils gauche. Cela permet de mélanger le tas et de le garder équilibré.

> *Remarque :* la propriété importante est qu'un tas créé uniquement grâce à la fonction `insere` est équilibré q. Il est par contre possible que `insere a t` ne soit pas équilibré même si `t` est équilibré.

Question 8. Programmez le *tri par tas* : pour trier une liste par ordre croissant, on met tous les éléments dans un tas, puis on vide le tas pour récupérer les éléments dans l'ordre.

Question 9. *(Bonus)* Comparez ce tri avec le tri fusion et le tri par insertion du TP1.

> *Remarque :* dans un langage impératif (C, Java, Pascal), les tas sont en général implémentés de manière très différente en utilisant des tableaux...

Partie 2 : Le crible d'Eratosthene (2h)

Pour trouver les nombres premiers (qui ne sont des multiples de personne, à part 1 et eux mêmes) plus petits que n, on peut procéder de la manière suivante :

- on commence avec tous les entiers strictement supérieurs à 1 (1 n'est pas premier), puis :
- on prend le premier élément (il est premier),
- on supprime tous les éléments qui sont divisible par celui-ci dans la suite de la liste,
- et on recommence jusqu'à épuisement de la liste.

Par exemple, pour `10` :

```
p = [ ]              l = [ 2; 3; 4; 5; 6; 7; 8; 9; 10 ]
p = [ 2 ]            l = [ 3; 5; 7; 9 ]
p = [ 2; 3 ]         l = [ 5 ; 7 ]
p = [ 2; 3; 5 ]      l = [ 7 ]
p = [ 2; 3; 5; 7 ]   l = [ ]
```

Les nombres premiers entre 1 et 10 sont effectivement 2, 3, 5 et 7.

Question 1. Écrivez une fonction `suppr_mult : int -> int list -> int list` qui permet de supprimer tous les multiples d'un nombre dans une liste.

Question 2. Écrivez une fonction `crible1 : int -> int list` qui calcule la liste des nombres premiers entre 0 et son premier argument (en utilisant l'algorithme ci-dessus).

Pour ceci, vous aurez probablement besoin d'une fonction locale `aux : int list -> int list -> int list`. Pour cette fonction, le premier argument `p` sera la liste des nombres premiers calculés jusqu'à présent (liste de gauche ci-dessus), et le second argument `l` sera la liste qu'il reste à traiter (liste de droite ci-dessus).

Jusqu'à quelle valeur de `n` pouvez-vous calculer `crible1 n` en un temps raisonnable ?

> *Consigne :* évitez les choses inutilement complexes comme les insertions d'un élément en fin de liste...

Question 3. On peut améliorer la complexité de cette fonction en constatant deux choses :

- on peut directement commencer avec la liste des nombres impairs, car la première étape supprime juste tous les multiples de 2 ;
- dés que la liste `l` ne contient que des nombres plus grands que $\sqrt{\texttt{n}}$, tous les nombres qu'elle contient sont premiers.

Écrivez la fonction `crible2 : int -> int list` qui avec ces améliorations.

Comparez les fonctions `crible1` et `crible2`.

On peut encore améliorer la fonction précédente en précalculant directement les 2 premières étapes, càd en supprimant directement les multiples de 2 et de 3. Pour ceci, on peut commencer à `5` et alternativement ajouter `2` puis `4` pour obtenir :

```
[ 5; 7; 11; 13; 17; 19; 23; 25; ...]
```

Si on veut commencer directement à l'étape suivante (en générant la liste des entiers non multiples de 2, 3 ou 5), il faut commencer avec `7` et ajouter successivement `4`, `2`, `4`, `2`, `4`, `6`, `2` et `6`.

Question 4. Écrivez une fonction `tourne : int list -> int -> int -> int list` qui permet de calculer une telle liste : pour "`tourne l i n`", on commence à `i`, on ajoute successivement les éléments de `l` et on s'arrête quand on dépasse `n`.

Question 5. Écrivez une fonction `crible3` correspondante, que vous comparerez avec les précédentes.

> *Remarque :* pour précalculer l'étape suivante, il faut commencer à `11` et utiliser la liste
>
> ```
> [ 2; 4; 2; 4; 6; 2; 6; 4; 2; 4; 6; 6; 2; 6; 4; 2; 6; 4; 6; 8;
>   4; 2; 4; 2; 4; 8; 6; 4; 6; 2; 4; 6; 2; 6; 6; 4; 2; 4; 6; 2;
>   6; 4; 2; 4; 2; 10; 2; 10 ]
> ```

Question 6. *Bonus, difficile* À votre avis, comment peut-on trouver les listes données précédemment.

Est-il efficace de précalculer de nombreuses étapes ?