Examen de programmation avancée

ENSIIE, semestre 2

mercredi 1er avril 2015

Durée : 1h45.
Tout document personnel autorisé (pas de prêt entre voisins). L'usage de la calculatrice ou de tout autre appareil électronique n'est pas autorisé (sauf traducteur pour les étudiants étrangers).
Ce sujet comporte 3 exercices indépendants, qui peuvent être traités dans l'ordre voulu. Il contient 5 pages.
Le barème est donné à titre indicatif, il est susceptible d'être modifié. Le total est sur 20 points.

Exercice 1 : Modularité et compilation séparée (7 points)

On souhaite écrire un programme qui prend une chaîne de caractères contenant des mots et des parenthèses, qui transforme celle-ci en arbre et qui affiche l'arbre en notation préfixe. Il n'est pas nécessaire pour cet exercice de savoir comment la suite de parenthèses et de mots est transformée en arbre.

On procède comme suit : on définit un type `token` dont les éléments sont soit une parenthèse ouvrante `lpar`, soit une parenthèse fermante `rpar`, soit un mot `word(`w`)`. Une fonction `split` sépare la chaîne de caractères en suite de `token`. Une fonction `parse` transforme cette suite de `token` en arbre (type `tree`). Enfin l'arbre est affiché en utilisant une fonction `print_tree`.

On organise le projet suivant ce découpage :

Module `Parser`

- un type abstrait `token`,
- deux fonctions `lpar` et `rpar` sans argument qui retourne un `token`,
- une fonctions `word` qui prend en argument une chaîne de caractère et qui retourne un `token`,
- une fonction `parse` qui prend en argument une chaîne de caractère, une fonction qui prend une chaîne de caractère et qui retourne un `token`, et qui retourne un `tree`.

Module `Lexer`

- une fonction `split` qui prend en argument une chaîne de caractère et qui retourne un `token`.

Module `Tree`

- un type abstrait `tree`,
- une fonction `empty` sans argument qui retourne un `tree`,
- une fonction `node` qui prend en argument une chaîne de caractère et deux `tree` et qui retourne un `tree`,
- une procédure `print_tree` qui prend en argument un `tree` et qui l'affiche (type de retour `[unit|void]`).

Module `Main`

- une fonction principale qui applique la fonction `parse` sur le premier paramètre de la ligne de commande, en utilisant comme fonction de découpage `split`, puis qui affiche le résultat avec `print_tree`.

1. Quelles sont les relations de dépendances entre les modules ? On donnera la réponse sous la forme d'une liste de phrases de la forme « Le module X dépend de l'interface du module Y. »
2. Quelle commande faut-il écrire manuellement pour compiler le module `Lexer` ? En OCaml, on supposera que les interfaces des modules sont déjà compilées.
3. En supposant que tous les modules sont compilés, quelle commande faut-il taper manuellement pour faire l'édition de liens ? (Attention à l'ordre des modules en OCaml !)
4. On modifie l'interface de `Tree`. Quel(s) module(s) faut-il recompiler ?
5. Écrire le `Makefile` correspondant au projet. On n'oubliera pas d'écrire une cible pour produire l'exécutable final qu'on appellera `prog`.
6. Écrire les interfaces des modules `Parser`, `Lexer` et `Tree`.

   Rappel : en C, pour donner le prototype d'une fonction qui prend en argument une fonction d'un type `a` vers un type `b`, on met comme type pour l'argument `b (*) (a)`.
7. Écrire l'implémentation du module `Main`.

   Rappel : pour accéder aux arguments de la ligne de commande en C, on définit la fonction `main` avec deux arguments `int argc, char **argv`. `argc` est le nombre d'arguments plus 1, `argv` est le tableau des arguments en tant que chaînes de caractères (le premier des éléments du tableau étant le nom du programme).

   On OCaml, on peut utiliser le tableau `Sys.argv`.

   Rappel : en C, pour passer une fonction comme argument d'une autre, il suffit de passer son nom.

Exercice 2 : Bûcheron (ou comment hacher des arbres) (4 points)

On considère les arbres binaires d'entiers définis en OCaml par

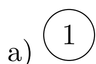
```
type ab = Vide | Noeud of ab * int * ab
```

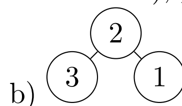
et en C par

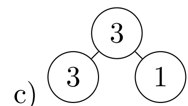
```
typedef struct ab_base* ab;
struct ab_base { ab fg; int val; ab fd; };
```

On définit une fonction de hachage sur les arbres binaires d'entier de la façon suivante : on fait la somme des entiers présents dans l'arbre, puis on utilise la méthode de la division sur cette somme, c'est-à-dire qu'on prend le reste de la division euclidienne par la taille de la table.

1. Quelle est la valeur de hachage associée aux arbres suivants (dans lesquels on a omis de représenter explicitement les sous-arbres vides), pour une table de taille 5 ?

a) 

b) 

c) 

2. Écrire une fonction `tree_hash` qui prend en argument un arbre binaire et un entier (représentant la taille de la table) et qui renvoie la valeur de hachage de l'arbre calculée comme indiqué ci-dessus.
3. Quelle est la complexité de cette fonction (en fonction de la taille de l'arbre) ?
4. On considère une table de hachage de taille 5, dont les clefs sont des arbres binaires avec la fonction de hachage ci-dessus, et dont les valeurs sont des entiers. Pour chaque arbre de la question 1 on insère dans cette table une association entre l'arbre et le produit de ses nœuds. Représenter l'état de la mémoire.

Exercice 3 : Tri par tas (9 points)

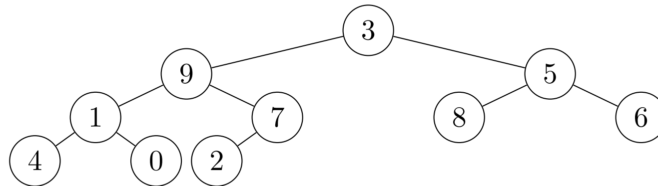
On considère les arbres binaires d'entier comme définis dans l'exercice 2 (type `ab`).
Comme vu en cours, il est également possible de représenter les arbres par un tableau :

– La racine est dans la première case (indice 0).

– Les sous-noeuds d'un noeud en case i sont en case $2i+1$ et $2i+2$.

– Le parent d'un noeud i est en case $\lfloor \frac{n-1}{2} \rfloor$.

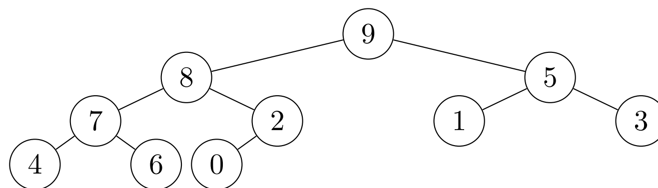
Ainsi, l'arbre



est représenté par le tableau

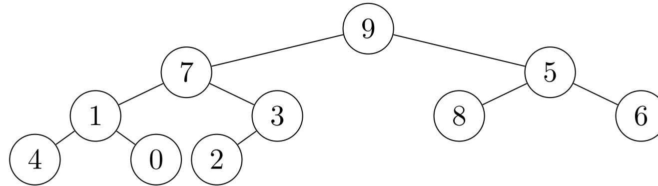
3	9	5	1	7	8	6	4	0	2

Un arbre est dit *ordonné en tas* si la valeur de chaque nœud est supérieure à celle de ses fils éventuels. Par exemple, le premier arbre dessiné ci-dessus n'est pas ordonné en tas, tandis que celui-ci l'est :



1. Montrer par induction que la racine d'un arbre ordonné en tas contient la valeur maximale contenue dans l'arbre.
2. Écrire une fonction `est_ordonne_tas` qui teste si un arbre (type `ab`) est ordonné en tas.
3. Donner la complexité de cette fonction en fonction du nombre de nœud dans l'arbre.
4. Écrire une fonction `est_tas` qui prend en argument un tableau d'entier représentant un arbre, un entier représentant la taille de ce tableau, et qui teste si l'arbre représenté par le tableau est ordonné en tas. On pourra écrire une fonction auxiliaire qui prend troisième argument entier correspondant à l'indice de la case correspondant au sous-arbre considéré.

Pour ordonner un arbre en tas, on s'appuie sur une opération de base appelée *tamisage* qui consiste à faire descendre la valeur d'un nœud si celle-ci est plus petite que celle d'un de ces fils. Concrètement, dans ce cas, on échange la valeur du nœud avec la plus grande de celles des racines des sous-arbres, puis on procède récursivement jusqu'à atteindre une feuille. Sur le premier exemple d'arbre, si on tamise 3, on va l'échanger avec 9, puis avec 7, et on s'arrête car 2 est plus petit. On obtient donc l'arbre :



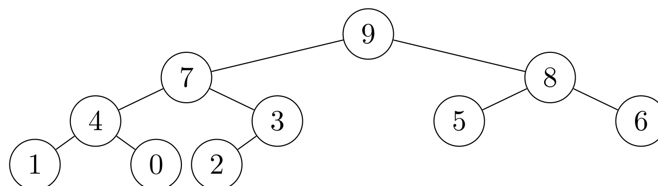
5. Écrire une procédure `tamiser` de prototype OCaml `int array -> int -> int -> unit` et C `void tamiser(int*,int,int);` dont le premier argument est un tableau représentant un arbre, le deuxième est la taille du tableau et le troisième est l'indice du nœud que l'on veut faire descendre.

   On supposera qu'on dispose d'un fonction `echange` qui prend en argument un tableau et deux indices et qui échange le contenu des cases du tableau aux indices donnés.

   Indication : il faudra distinguer les cas où le nœud est une feuille, a uniquement un fils gauche ou a deux fils.

6. Quelle est la complexité dans le pire des cas de la fonction `tamiser` en fonction de la longueur du tableau ?

Pour ordonner un arbre en tas, il suffit de tamiser tous ses nœuds en commençant par ceux du bas. Bien évidement, il ne sert à rien de tamiser les feuilles. Par conséquent, pour un arbre représenté par un tableau de taille n, on peut commencer par le nœud d'indice $\lfloor \frac{n}{2} \rfloor - 1$ puis diminuer d'indice jusqu'à arriver à la racine. Sur le premier exemple ci-dessus, on commence par s'intéresser au nœud contenant 7, qui n'a pas besoin d'être tamisé ; puis on tamise 1 en l'échangeant avec 4 ; on tamise 5 en l'échangeant avec 8 ; 9 n'a pas besoin d'être tamisé ; on tamise 3 en l'échangeant avec 9 puis avec 7. Au final on obtient



7. Justifier que le dernier nœud concerné est à l'indice $\lfloor \frac{n}{2} \rfloor - 1$.

8. Écrire une procédure `ordonner_en_tas` de prototype OCaml `int array -> int -> unit` et C `void ordonner_en_tas(int*,int);` dont le premier argument est un tableau représentant un arbre et le deuxième est la taille du tableau.

   On pourra s'aider d'un fonction auxiliaire avec un argument supplémentaire qui est l'indice de la dernière case à tamiser.

9. Quelle est la complexité dans le pire des cas de la fonction `ordonner_en_tas` en fonction de la longueur du tableau ?

On cherche à trier un tableau d'entiers dans l'ordre croissant. Pour cela on va procéder de la manière suivante :

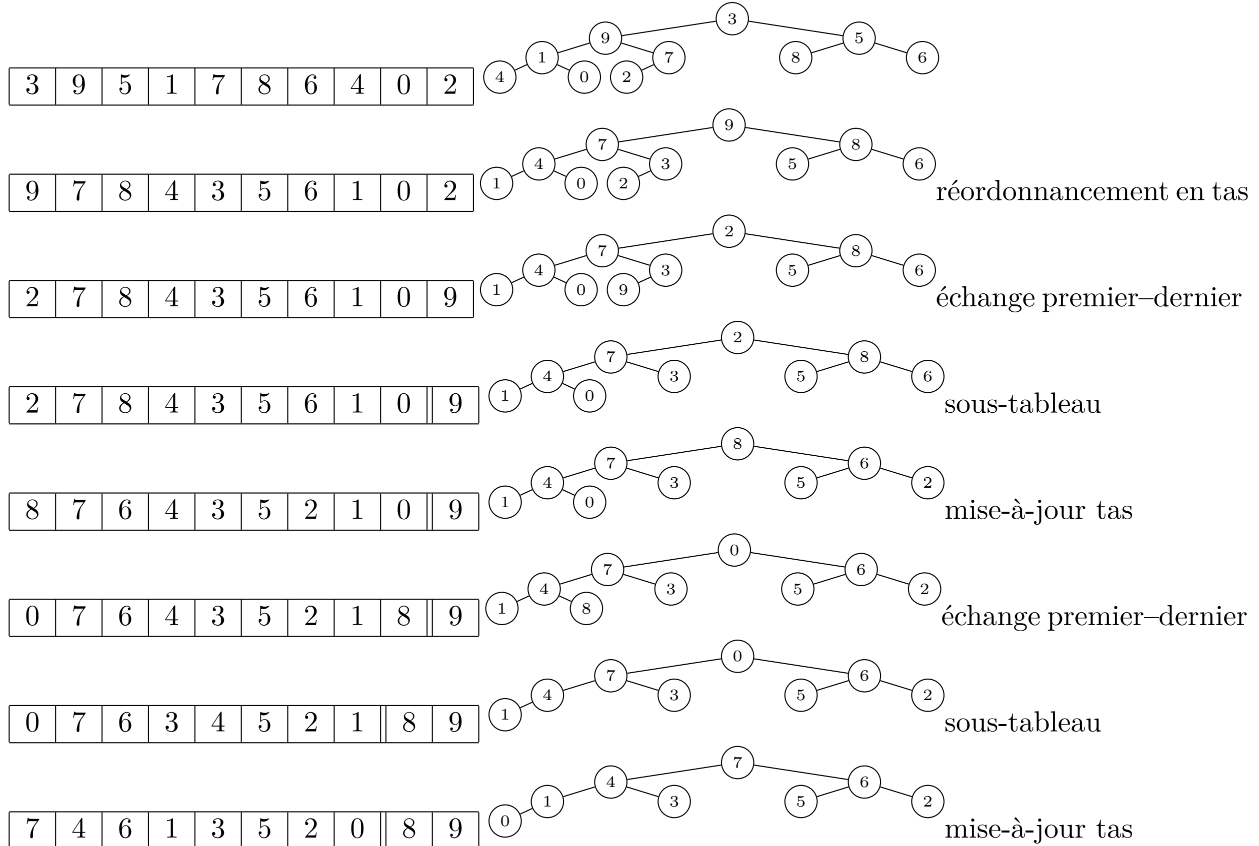
– On considère le tableau comme la représentation d'un arbre.

– On réordonne les nœuds pour que l'arbre soit ordonné en tas.

– L'élément maximal est donc à la racine. On l'échange avec le dernier élément.

– On recommence sur le tableau sans la dernière case.

Par exemple, on a l'évolution suivante (on donne le tableau et l'arbre associé) :



etc. jusqu'à ce que le tableau soit trié.

Pour ordonner le tableau en tas, on distingue deux cas :

– À la première itération, tous les éléments sont potentiellement mal placés, on applique `ordonner_en_tas`.

– Aux itérations suivantes, après avoir échangé l'élément maximal avec le dernier, seule la racine est mal placée, il suffit de tamiser celle-ci.

10. Écrire la fonction `tri_par_tas` de prototype OCaml `int array -> int -> unit` et C `void tri_par_tas(int*,int);` qui trie le tableau avec le tri par tas, le deuxième argument étant la taille du (sous-)tableau considéré.

    On pourra écrire une fonction auxiliaire qui suppose que le tableau passé en argument représente un arbre ordonné en tas.

11. Donner la complexité de `tri_par_tas` en fonction de la taille du tableau.