DS 4

Informatique MP2I

Julien Reichert

Questions de cours

Question de cours A1 : Soit la structure d'arbre binaire définie comme étant soit un arbre vide soit un nœud avec deux enfants qui sont des arbres binaires. Montrer que le nombre d'arbres vides dans un arbre est un de plus que le nombre de nœuds.

Question de cours A2 : Donner une définition acceptable d'un peigne droit.

Question de cours A3 : Donner en la démontrant la relation entre la hauteur et la taille d'un arbre binaire complet non vide avec la convention de vocabulaire donnée en cours.

Questions de cours en OCaml

Question de cours B1 : Donner un type possible pour les arbres d'arité quelconque.

Question de cours B2 : Rappeler dans l'ordre avec quel opérateur on réalise respectivement un test d'égalité (arithmétique, pas physique), un test de différence (idem), la modification d'une référence et la modification de l'élément à un certain indice du tableau.

Question de cours B3 : Définir un type somme indirectement récursif et un type enregistrement directement récursif. Dans les deux cas, il faut pouvoir créer des instances finies. Créer une instance finie et une instance infinie de chaque type.

Questions de cours en C

Question de cours C1 : Quelle option de compilation permet de détecter entre autres les fuites de mémoire ?

Question de cours C2 : Écrire un type possible pour un arbre binaire d'entiers, avec ou sans l'encapsulation (et donc l'utilisation de deux types) qu'on a observée pour les listes chaînées pour permettre d'avoir une structure vide.

Question de cours C3 : Définir une structure indirectement récursive (la méthode est la même que pour définir des fonctions mutuellement récursives). Il faut pouvoir créer des instances finies. Créer une instance finie et une instance infinie.

Exercices

Le langage est au choix **parmi OCaml et C** pour tous les exercices demandant d'écrire un programme **sauf les deux premiers**. Cependant, **si un exercice est traité dans les deux langages, aucune des deux versions ne sera corrigée**. Pour changer d'avis alors qu'une partie sur un des langages a déjà été commencée au propre, cette partie sera à barrer de manière bien visible et sera ignorée.

Consigne : tout exercice prenant en argument ou renvoyant une « séquence » prendra en argument / renverra un tableau en C (ainsi que sa taille, comme d'habitude), mais en OCaml le choix est laissé entre tableau et liste (si c'est pertinent, le choix peut être différent entre argument et valeur de retour, mais on évitera de mélanger des types différents au niveau des arguments). Pour cette raison, **toute fonction écrite en OCaml sera accompagnée de sa signature, dont l'exactitude et la cohérence avec l'énoncé feront partie de la notation** (les prototypes en C sont de toute manière systématiques aussi). On rappelle à tout hasard qu'en OCaml un type comme `list` et `array` est toujours paramétré par un autre type...

On rappelle la documentation du module `Hashtbl` :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option, donc si la clé est absente aucune exception n'est levée, c'est simplement le cas où `None` est renvoyé ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).

Exercice 1 [en OCaml obligatoirement] : Créer un type pour représenter la structure d'arbre binaire sans étiquettes. Écrire une fonction qui prend en argument un entier et qui renvoie un arbre binaire complet de hauteur cet entier. Écrire aussi une fonction qui prend en argument un arbre binaire et qui détermine s'il est complet. Calculer la complexité en temps et en espace de ces deux fonctions.

Exercice 2 [en C obligatoirement] : Créer une structure de liste chaînée d'entiers au choix (la structure de maillon avec ou sans structure supplémentaire pointant sur le premier maillon) en écrivant le code permettant de définir la structure (l'instruction introduite par `typedef` n'est pas obligatoire). Préciser à quel endroit du code une allocation de mémoire est nécessaire. Écrire pour cette structure une fonction de libération d'une liste chaînée. Aucune opération élémentaire n'a besoin d'être écrite sauf celles qui seraient éventuellement utilisées.

Exercice 3 [Suggestion : OCaml] : Écrire une fonction prenant en argument une séquence (d'entiers si c'est en C) et renvoyant l'élément le plus fréquent de cette séquence. En cas d'égalité, on renverra le premier élément à avoir eu toutes ses apparitions.

Exercice 4 : Écrire une fonction prenant en argument une séquence d'entiers représentant des valeurs de pièces ainsi qu'un entier représentant une somme et renvoyant une séquence d'entiers permettant de déduire une façon de rendre la somme en utilisant le moins de pièces possibles avec le détail des valeurs à utiliser. Il est aussi possible de produire par exemple des couples renseignant pour toutes les pièces (utilisées ou non) le nombre d'exemplaires. On ne supposera pas que le système monétaire associé ait la moindre propriété.

Exercice 5 : Écrire une fonction prenant en argument une séquence non vide (pas à vérifier) d'entiers et renvoyant le plus petit entier supérieur ou égal au minimum de la séquence et n'y figurant pas.

Exercice 6 : Soit k un entier naturel non nul. On considère un tableau (d'entiers en C / de type quelconque en OCaml) dont on veut envoyer (en mutant le tableau, évidemment) les k premiers éléments à la fin, comme si on échangeait le bloc qui les contient avec le bloc formé par le reste du tableau. Quelle serait la complexité d'un algorithme naïf **en espace constant** (algorithme qu'on n'écrira pas mais dont on donnera l'idée) ? Proposer l'idée d'un algorithme dont **la complexité en espace reste constante** et la complexité en temps ne dépend pas de k mais seulement de la taille du tableau et écrire l'algorithme en question.

Exercice 7 : Créer une structure de « table de comptage » ressemblant à un dictionnaire à ceci près que les opérations élémentaires sont limitées à la création d'une table, l'incrémentation de la valeur associée à une clé (si elle n'y figure pas encore, on insère la clé associée à la valeur 1), la consultation de la valeur associée à une clé (comportement au choix si elle n'y figure pas) et la récupération de la liste des clés (le choix de ces opérations est une improvisation de ma part). On pourra s'appuyer sur toute séquence de base existante (les tables de hachage sont exclues, mais il n'est pas interdit de créer une table de comptage comme on a créé les tables de hachage en TP). Toutes les complexités seront à calculer, leur optimisation n'ayant pas de poids sur la notation. Si l'exercice est fait en C, les clés seront des chaînes de caractères.

Exercice 8 : Déterminer ce que fait la fonction suivante (après la création de type nécessaire), et calculer sa complexité. Commenter.

```
type t = E | C of int * t * t;;

let rec f r = match r with
| E -> max_int
| C(a, b, c) when f b < f c -> if a < f b then a else f b
| C(a, b, c) -> if a < f c then a else f c;;
```

Problème 1

Ce problème est à traiter intégralement en OCaml.

On souhaite réaliser une structure de données appelée dèque, dont le nom est le calque du raccourci anglais *deque*, signifiant *double-ended queue*, c'est-à-dire file à double entrée.

Contrairement à la file (et à la pile), une dèque supporte des opérations élémentaires d'ajout et de retrait (renvoyant l'élément retiré) sur ses deux extrémités, et la structure est en pratique munie de pointeurs vers le premier et vers le dernier élément.

Le type sera défini ainsi :

```
type 'a elem_deque = { valeur : 'a; mutable suivant : 'a elem_deque option;
mutable precedent : 'a elem_deque option };;

type 'a deque = { mutable vide : bool; mutable debut : 'a elem_deque option;
mutable fin : 'a elem_deque option };;
```

Exercice du problème 1 : Écrire une fonction de création de dèque et les quatre opérations élémentaires mentionnées.

Problème 2

Ce problème est à traiter intégralement en C.

Nous allons réaliser ici un interpréteur pour le langage Brainfuck. Ce langage minimaliste permet de réaliser tous les calculs que ferait un ordinateur (on parle de *Turing-complétude*) grâce à huit instructions élémentaires.

On considère un tableau d'octets supposé illimité à gauche et à droite (indexé par $\mathbb{Z}$, en quelque sorte), tous les octets étant initialement à la valeur zéro. Chaque octet correspond en pratique à un caractère de la table ASCII. En ceci, cela ressemble à la gestion des caractères par le langage C.

Un pointeur figure sur ce qu'on pourrait appeler l'indice zéro, même si l'invariance par translation fait que la seule chose qui compte est la position relative par rapport au point de départ. Ce pointeur sera manipulé par le programme au fur et à mesure de l'exécution.

On considère également un flux d'entrée, qui sera assimilé pour nous à une chaîne de caractères. À tout moment, le programme peut récupérer le caractère suivant, la lecture ne pouvant pas revenir en arrière, avec un comportement indéterminé si on tente de lire au-delà du zéro terminal de la chaîne (on considère qu'une erreur sera déclenchée, pour ce problème).

Pour simplifier, on supposera que les octets seront représentés par des entiers et on fera les calculs modulo 256.

Les huit instructions disponibles sont les suivantes (tout autre caractère est ignoré) :
- `+` : Augmente d'un la valeur de l'octet à la position où se situe le pointeur.
- `-` : Diminue d'un la valeur de l'octet à la position où se situe le pointeur.
- `>` : Déplace le pointeur d'un cran vers la droite.
- `<` : Déplace le pointeur d'un cran vers la gauche.
- `,` : Remplace l'octet à la position où se situe le pointeur par le caractère suivant du flux d'entrée.
- `.` : Imprime l'octet à la position du pointeur.
- `[` : Passe à l'instruction suivante si l'octet à la position où se situe le pointeur est non nul, passe après l'instruction `]` correspondante sinon.
- `]` : Instruction sans effet qui sert à marquer la fin d'une boucle conditionnelle.

Le résultat est imprimé dans la console, donc inutile de se poser la question du zéro terminal.

Les fonctions à écrire prendront un argument de type chaîne de caractères et appelé « le programme Brainfuck ». Le flux d'entrée sera aussi mis en argument de la fonction qui interprète le programme Brainfuck.

Question P1 : Quelle est le rapport entre la taille du tableau que l'on utilisera et la taille qu'on aurait pu avoir ?

Question P2 : À quoi faut-il faire particulièrement attention en manipulant le type `int` lorsque l'on fait des calculs modulo 256 ?

Question P3 : En admettant qu'entre le début et la fin de toute boucle le pointeur soit à la même position, écrire une fonction déterminant la position minimale et la position maximale possibles (difficile de savoir si une boucle est ignorée...) du pointeur au cours d'une exécution, par rapport à sa position de départ. L'argument sera le programme Brainfuck, le flux d'entrée n'est pas censé avoir d'influence sous notre hypothèse.

Question P4 : Peut-on anticiper le nombre de caractères lus simplement à la lecture du programme et donc savoir s'il y a un risque de déclenchement d'erreur ? Selon le même principe, peut-on anticiper les positions extrémales juste à partir du programme Brainfuck ?

Question P5 : En supposant la fonction de prototype `void interpreteur(char* prog, char* flux)` déjà écrite, écrire une fonction `main` permettant de tester l'interpréteur sur un programme Brainfuck qui est à trouver par ailleurs (en tant que chaîne de caractères) dont le principe est que le flux d'entrée est recopié en imprimant deux fois de suite chaque caractère. Il faut que la compilation produise un exécutable qui prend en argument une chaîne de caractères qui constituera le flux d'entrée.

Question P6 : Le seul risque qu'un programme en Brainfuck déclenche une erreur de syntaxe est un problème d'appariement des délimiteurs de boucle. Écrire une fonction qui prend en argument un programme Brainfuck et qui détermine si la syntaxe est correcte.

Question P7 : Savoir qu'il n'y a pas d'erreurs de syntaxe n'est pas suffisant pour l'exécution du programme, il faut aussi connaître les appariements des crochets, si possible à l'avance pour des raisons de complexité. Écrire une fonction qui prend en argument un programme Brainfuck et qui renvoie les couples de crochets appariés, selon une forme au choix exploitable (et exploitée à l'avant-dernière question). En particulier, il faudra expliquer comment la valeur de retour représente les couples de crochets.

Question P8 : Pour revenir à la question P3, inutile de réfléchir aux positions extrémales et de créer un tableau de la taille adéquate avec la position de départ au bon endroit si on se sert d'une structure de données pertinente pour contenir les octets nécessaires. Quelle pourrait être cette structure, potentiellement un peu différente de celles vues en cours ? Expliquer comment elle fonctionnerait et l'implémenter.

On peut supposer une structure adéquate réalisée même sans avoir fait la question P8 pour la fin du problème. On donnera simplement l'interface.

Question P9 : Écrire une fonction déterminant en temps au plus logarithmique en le nombre de couples de crochets l'indice du crochet ouvrant correspondant à un crochet fermant donné dans un programme Brainfuck. Un des arguments doit être l'indice du crochet fermant à apparier, l'autre argument peut être le programme, la valeur de retour de la fonction de la question P7 ou même n'importe quelle structure qui s'en déduit.

Pour avoir les points sur cette question, il est nécessaire d'utiliser des arguments correctement construits quelque part sur la copie.

Question P10 : Écrire la fonction de prototype `void interpreteur(char* prog, char* flux)` qui exécute un programme Brainfuck sur le flux d'entrée, en se servant des fonctions intermédiaires écrites ou supposées écrites.