DS 3

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou dans le titre du problème, en fonction de la situation.

Questions de cours ou d'application directe du cours

Question de cours 1 : Définir la notion de structure de données abstraite, exemple à l'appui.

Question de cours 2 : Donner une différence importante entre une pile et une file.

Question de cours 3 : Donner une différence importante entre une pile et une liste simplement chaînée.

Question de cours 4 : Donner au moins deux façons différentes d'implémenter un tableau associatif.

Question de cours 5 : Définir rigoureusement l'ordre lexicographique basé sur deux ordres.

Question de cours 6 : Écrire un exemple de type somme en OCaml ainsi qu'une valeur de ce type.

Question de cours 7 : Écrire un exemple de type enregistrement en OCaml ainsi qu'une valeur de ce type.

Exercices issus des TD et TP

Exercice T1 : Écrire une fonction qui prend en entrée une main (tableau de cartes, celles-ci étant du type défini ci-après) et qui détermine le nombre de points dans cette main, au sens de la fonction ci-après.

```
type valeur = Sept | Huit | Neuf | Dix | Valet | Dame | Roi | As;;
type couleur = Trefle | Pique | Coeur | Carreau;;

type carte = { va : valeur; coul : couleur };;

let points_carte carte = match carte.va with
| As -> 11 | Dix -> 10 | Roi -> 4 | Dame -> 3 | Valet -> 2 | _ -> 0;;
```

Exercice T2 : Implémenter en OCaml la structure de file bornée, dont l'interface est donnée ci-après.

```
type 'a fileb = { mutable taille : int; mutable tete : int; mutable queue : int; donnees : 'a array };;
val creer_fileb : int -> 'a -> 'a fileb;;
val enfiler_fileb : 'a fileb -> 'a -> unit;;
val defiler_fileb : 'a fileb -> 'a;;
val est_vide_fileb : 'a fileb -> bool;;
```

Exercice T3 : Écrire en OCaml la fonction `map : ('a -> 'b) -> 'a list -> 'b list`, qui applique son premier argument aux éléments de son deuxième argument dans l'ordre pour former une liste.

Exercice T4 : Implémenter le tri par dénombrement, expliqué ci-après, en C. On supposera que le tableau en entrée contiendra des valeurs entre `0` et `k`, pour un `k` en argument.

Le tri par dénombrement permet de trier une structure dont les éléments proviennent d'un ensemble fini. Par exemple, on voudrait trier un million de caractères, ce qui s'assimile à trier un million de nombres entre 0 et 127. Pour ce faire, on crée un tableau de taille 128 dont les éléments sont le nombre d'occurrences de chacun des nombres de 0 à 127, que l'on va compter en parcourant la structure. Une fois le parcours terminé, on constitue directement la version triée de la structure.

Exercices

Exercice 1 : On considère une séquence de couples formés par une clé et une valeur flottante. Il s'agit de construire la séquence de même type ayant les mêmes clés mais en un seul exemplaire, et la valeur associée à chaque clé doit être la somme des valeurs associées à cette clé dans la séquence de départ. Écrire trois fonctions faisant ce travail, l'une en OCaml lorsque les séquences sont des listes **sans utiliser une autre structure de données directement ou indirectement**, l'une en OCaml lorsque les séquences sont déjà des dictionnaires de type `Hashtbl.t`, la dernière en C lorsque les séquences sont des tableaux et les couples remplacés par des structures où la clé est un entier et la valeur un flottant.

Pour les exercices suivants, **à faire en OCaml**, on signale que la fonction `int` du module `Random` permet d'obtenir un entier pseudo-aléatoire entre zéro inclus et l'argument de la fonction exclu. On suppose par la suite (par abus) que tous les entiers ont la même probabilité d'être obtenus par cette fonction.

Exercice 2 : Écrire une fonction `randint` prenant en argument deux entiers et renvoyant un entier pseudo-aléatoire entre son premier argument inclus et son deuxième argument inclus (on supposera sans le vérifier le premier entier inférieur ou égal au deuxième), avec la même propriété d'équiprobabilité. Écrire aussi une fonction `choice` prenant en argument une liste et renvoyant un élément de la liste au hasard avec équiprobabilité. Écrire finalement une fonction `sample` prenant en argument une liste et un entier et renvoyant une liste obtenue en tirant successivement et sans remise autant d'éléments de la liste en premier argument que la valeur du deuxième argument (si cette valeur est supérieure à la taille de la liste, on déclenchera une erreur). Il n'y a pas d'objectif de complexité mais la terminaison doit être garantie, et pas seulement avec probabilité un.

Exercice 3 : Donner la complexité de la fonction `sample` écrite (dans le pire des cas, sans tenir compte de l'aléa).

Exercice 4 : Écrire une fonction `ri` simulant la fonction `int` du module `Random` en se servant d'une fonction fictive sans argument, supposée disponible sans passer par un module, appelée `randbool` supposée renvoyer `true` ou `false` de manière équiprobable. L'équiprobabilité de `ri` peut être approximative suivant la méthode choisie.

Problème 1 : Implémentation de la structure de file [intégralement en C]

Dans le TP sur les structures de données, la structure de file n'est implémentée qu'en OCaml. Il s'agit ici de faire essentiellement le même travail, mais en C. **Les files ne contiendront ici que des entiers.**

La première implémentation consiste à faire une file non seulement bornée, mais dont le nombre d'enfilements lui aussi est limité à un nombre fixé et connu à l'avance. Le principe est alors de créer une structure contenant un tableau support, un entier précisant le taille du tableau support, un entier indiquant l'indice où l'on écrira le prochain élément et un entier indiquant l'indice où on lira le prochain élément. Une fois défilé, un élément reste dans le tableau, et la zone effective de la file est la plage d'indices entre les deux entiers supplémentaires (de l'indice de lecture inclus jusqu'à l'indice d'écriture exclu).

Question P1.1 : Réaliser cette implémentation, avec la création du type et les fonctions de création (ayant comme argument la capacité du tableau support), d'enfilement, de défilement et le test de vacuité. En cas d'enfilement au-delà de la taille possible ou de défilement alors que la file est vide, on utilisera l'instruction `exit(1);`.

Question P1.2 : Application de cette version restreinte des files : mutation d'une pile par renversement de son contenu. Cette opération pourrait se faire avec deux piles supplémentaires (transférer successivement le contenu de la pile dans la première pile supplémentaire, puis dans la deuxième pile supplémentaire, puis dans la pile de départ), mais on propose de le faire avec une file, dont la taille maximale sera la taille de la pile à renverser. Écrire une fonction réalisant cette opération, en supposant l'existance d'un type `intpile` avec les opérations élémentaires ci-après et une fonction `taillepile` (en temps constant pour que cela ait de l'intérêt, prévu dans l'implémentation).

```
intpile creerpile();
void empiler(intpile p, int e);
int depiler(intpile p);
bool estvide(intpile p);
```

La deuxième implémentation consiste à faire une file bornée dans un « tableau circulaire ». La structure contient toujours un tableau support, sa taille et les deux indices en question, ainsi qu'un troisième entier indiquant la taille effective. Quand l'indice d'écriture dépasse la capacité du tableau, il revient à zéro. L'erreur n'est alors déclenchée lors d'un enfilement que si la taille est déjà égale à la capacité.

Question P1.3 : Réaliser cette nouvelle implémentation. On ajoutera une fonction déterminant si une file est pleine.

Question P1.4 : Application de cette version des files : caisses dans un supermarché. On considère un modèle ultra simplifié de supermarché avec `n` caisses et des clients dont le temps de passage en caisse est renseigné (les clients sont assimilés à des entiers représentant ce temps). On suppose que chaque client se déplace instantanément à la caisse libre du plus petit indice dès qu'une caisse se libère, et qu'avant leur passage ils font partie d'une unique file d'attente représentée par une file dont on borne la taille à un certain `k`. Écrire une fonction prenant en argument le tableau des clients et sa taille ainsi que les entiers `n` et `k` et qui gère les enfilements (si la file ne sature pas), les défilements (si une caisse se libère) en utilisant toutes les variables nécessaires pour le bon fonctionnement du programme, afin de renvoyer le temps total mis pour que tous les clients aient quitté la caisse où ils seront allés.

La troisième implémentation consiste à utiliser des maillons de manière similaire au cas des listes chaînées mais en limitant les opérations élémentaires disponibles à ce que les files permettent. Il y aura donc la structure de maillon (un attribut entier qui est l'information et un pointeur vers le maillon suivant dans l'ordre d'enfilement) et la structure de file qui a deux attributs : un pointeur vers le premier maillon (où se font les défilements) et un pointeur vers le dernier maillon (où se font les enfilements). Ici, les enfilements ne poseront jamais de problème (on considère que l'allocation de mémoire réussit toujours), et il y a bien entendu toujours la vérification de non-vacuité avant de défiler.

Question P1.5 : Réaliser cette dernière implémentation.

Question P1.6 : Application de cette version complète des files : arithmétique binaire. On assimile un entier à une file de `0` et de `1` telle que le premier élément enfilé soit le bit de poids fort. Écrire une fonction prenant en argument deux files représentant chacune un entier et renvoyant une file représentant leur somme. On pourra muter les files.

Problème 2 : Arithmétique inductive [intégralement en OCaml]

Parmi les manières inductives de définir l'arithmétique sur les entiers naturels, la plus connue est sans doute la version de Peano, et la plus troublante celle de Von Neumann (correspondant également à la construction de Zermelo). Nous allons commencer par cette dernière.

On peut définir les entiers naturels ainsi : le zéro est représenté par l'ensemble vide, et le successeur d'un entier n est représenté par la réunion de l'ensemble représentant n et de l'ensemble dont le seul élément est l'ensemble représentant n.

En clair et en assimilant un entier à l'ensemble qui le représente, on a $0 = \emptyset$ et $n + 1 = n \cup \{n\}$.

Question P2.1 : Représenter selon cette méthode les entiers de 1 à 4. Par ailleurs, quelle est la taille de l'ensemble représentant l'entier n, en fonction de n ?

Question P2.2 : Écrire une fonction `vn` prenant en argument un entier et renvoyant sa représentation selon la méthode de Von Neumann. On réfléchira pour commencer au type à employer.

L'axiomatique de Peano revient à utiliser la constante 0 et la fonction successeur, notée mathématiquement S, pour décrire n'importe quel entier naturel.

Par exemple, le nombre 5 sera $S(S(S(S(S(0)))))$.

Question P2.3 : Créer un type somme `entier_peano` reprenant ce principe.

Question P2.4 : Écrire une fonction de conversion du type entier vers le type `entier_peano` et vice-versa.

Les quatre fonctions à suivre doivent prendre en argument deux entiers du type `entier_peano` et renvoyer un (ou deux pour la dernière fonction) entiers de ce type. **Il est totalement interdit de convertir les entiers.** (Les fonctions de conversion ont en fait été écrites pour tester la correction et sont demandées dans un but pédagogique.)

Question P2.5 : Écrire une fonction d'addition.

Question P2.6 : Écrire une fonction de soustraction (on déclenchera une erreur si l'opération mathématique correspondante produit un entier strictement négatif).

Question P2.7 : Écrire une fonction de multiplication.

Question P2.8 : Écrire une fonction de division euclidienne. La fonction renverra un couple formé du quotient et du reste dans la division euclidienne, dans cet ordre.

Annexe

On rappelle à toute fin utile des fonctions du module `Hashtbl` avec des informations sur leur spécification :

- `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.clear th` vide la table de hachage en argument. La fonction `Hashtbl.reset` a le même effet mais la table de hachage revient en plus à son nombre initial de places.
- `Hashtbl.copy th` crée une copie de la table de hachage en argument.
- `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.length th` renvoie le nombre de clés (doublons compris) 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 pour éviter de lever une exception si la clé est absente ;
- `Hashtbl.iter f th` appelle la fonction (sans valeur de retour, donc effectuant juste un effet secondaire) 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).
- `Hashtbl.fold f th acc` appelle la fonction fournie, prenant des clés et des valeurs ainsi qu'un accumulateur (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). La fonction renverra la valeur de l'accumulateur à la suite de tous ces appels successifs.