Programmation Avancée (INF441)
Contrôle de remplacement et de rattrapage

3 septembre 2015

Les parties 1 et 2 sont indépendantes l'une de l'autre. Elles peuvent être traitées dans l'ordre de votre choix.

1 Une file de priorité

Dans cette partie, on demande (autant que possible) d'écrire **du code Java correct**.

On souhaite compléter le squelette de la classe `VectorPriorityQueue`, donné dans la figure 1, pour implémenter une file de priorité. Les éléments contenus dans la file ont un type `E` arbitraire. Le constructeur a pour argument une borne P, et initialise une nouvelle file vide. On suppose ensuite que les priorités sont des nombres entiers compris entre 0 (inclus) et P (exclus). La méthode `insert` attend un élément `element` et sa priorité `p`. Elle ajoute cet élément à la file. La méthode `extract` retire de la file un élément de priorité minimale et le renvoie. Elle lance l'exception `NoSuchElementException` si la file est vide.

On appelle « vecteur » un objet de classe `Vector`, c'est-à-dire un tableau, redimensionnable si besoin. Les principales méthodes de la classe `Vector` sont données dans la figure 2. On rappelle qu'un vecteur a, à tout moment, une certaine taille n, donnée par la méthode `size`. Lors de sa construction, un vecteur a une taille 0. Les méthodes `get` et `set` exigent que l'indice `i` soit compris entre 0 (inclus) et n (exclus). La méthode `add` ajoute un élément à la fin du vecteur, donc incrémente n de 1. La méthode `remove` supprime l'élément d'indice `i` (et décale d'un cran vers la gauche les éléments qui suivent), donc décrémente n de 1. La méthode `setSize` modifie la valeur de n. Si la nouvelle valeur est supérieure à l'ancienne, les nouveaux éléments du vecteur sont initialisés à la valeur `null`.

On demande de représenter la file de priorité en mémoire à l'aide d'un vecteur de taille P dont la case d'indice p contient l'ensemble des éléments de priorité p. Cet ensemble pourra lui-même être représenté à l'aide d'un vecteur, dans lequel l'ordre des éléments n'a pas d'importance.

Question 1 Indiquez quel(s) doi(ven)t être le(s) champ(s) de la classe `VectorPriorityQueue`. Donnez ensuite l'implémentation du constructeur et des méthodes `insert` et `extract`. ◇

On admet que la complexité asymptotique en temps de la méthode `add` de la classe `Vector` est $O(1)$. Cette propriété est vraie au sens amorti, c'est-à-dire que le coût de n appels successifs à `add` est $O(n)$.

Question 2 Cette propriété de `add` étant admise, quelles sont les complexités asymptotiques en temps du constructeur et des méthodes `insert` et `extract` ? Exprimez-les en fonction des paramètres n et P, où n est le nombre actuel d'éléments de la file. Justifiez brièvement votre réponse. ◇

Afin de tester la classe `VectorPriorityQueue`, on souhaite écrire un petit morceau de code qui utilise cette classe. On choisit d'écrire une fonction qui trie en place un tableau de chaînes de caractères par longueur croissante. L'algorithme proposé est simple : on insère d'abord tous les éléments du tableau dans la file, puis on extrait tous les éléments de la file et on les stocke, dans l'ordre, dans le tableau.

Question 3 Écrivez une fonction `static void sort (String[] strings, int P)` en suivant cette suggestion. On suppose que les chaînes de caractères contenues dans le tableau `strings` ont une longueur strictement inférieure à P. ◇

Question 4 Quelle est la complexité asymptotique en temps de la fonction `sort`, exprimée en fonction des paramètres n et P, où n est la longueur du tableau `strings` ? Justifiez brièvement votre réponse. ◇

Pour la question suivante, on se place dans le cadre d'un scénario d'utilisation particulier de la file de priorité. On fait l'hypothèse que, à chaque fois que `insert(p, e)` est appelée, l'entier `p` est supérieur ou égal à la priorité des éléments qui ont été extraits de la file jusqu'ici.

La fonction `sort` ci-dessus respecte cette hypothèse, puisque toutes les insertions se font avant toutes les extractions. L'algorithme de Dijkstra, qui alterne insertions et extractions, la respecte également, puisque la priorité utilisée lors d'une insertion est toujours de la forme $p + d$, où p est la priorité du dernier élément extrait de la file et d est le poids (positif ou nul) d'une arête du graphe.

Sous cette hypothèse, on souhaite effectuer une série de $2n$ opérations, dont n insertions et n extractions, et on souhaite que la complexité asymptotique **totale** en temps de ces $2n$ opérations soit $O(n + P)$.

Question 5 Apportez des modifications (simples) à votre classe `VectorPriorityQueue` de la question 1 de façon à atteindre cet objectif. Justifiez brièvement pourquoi le coût total de n insertions et n extractions est bien $O(n + P)$. ◇

La question qui suit ne dépend que de la question 1.

Question 6 On souhaite que la file de priorité soit utilisable même lorsque la borne P n'est pas connue à l'avance. Apportez des modifications (simples) à votre classe `VectorPriorityQueue` de la question 1 de façon à atteindre cet objectif. ◇

2 Mémoïsation

Dans cette partie, on demande (autant que possible) d'écrire **du code OCaml correct**.

À titre d'exemple de calcul coûteux, nous utilisons la fonction C qui à un couple d'entiers (n, p) (où $0 \leq p \leq n$) associe le coefficient binômial C_n^p. Cette fonction mathématique peut être définie de façon inductive :

$$\begin{aligned} C_n^0 &= 1 \\ C_n^n &= 1 \\ C_{n+1}^{p+1} &= C_n^p + C_n^{p+1} \quad \text{si } 0 \leq p < n \end{aligned}$$

Question 7 En traduisant littéralement la définition mathématique ci-dessus, écrivez une fonction récursive `c` de type `int * int -> int` telle que, pour tous nombres entiers `n` et `p` tels que `0 <= p <= n`, l'appel `c (n, p)` calcule C_n^p. Vous ne vous inquiéterez pas d'efficacité ni des dépassements de capacité (*overflows*) liés à la précision limitée du type `int`. ◇

Question 8 Donnez une borne supérieure de la complexité asymptotique en temps, dans le cas le pire, de votre fonction `c`. Exprimez-la en fonction du paramètre n. Justifiez brièvement votre réponse. ◇

```
class VectorPriorityQueue<E> {
  public VectorPriorityQueue (int P) { ... }
  public void insert (int p, E element) { ... }
  public E extract () throws NoSuchElementException { ... }
}
```

FIGURE 1 – Squelette de la classe `VectorPriorityQueue`

```
class Vector<E> {
  Vector ();              // constructor
  E get (int i);          // get element at index i
  void set (int i, E e); // set element at index i
  void add (E e);         // append element at the end
  E remove (int i);       // remove element at index i
  int size ();            // get size of vector
  void setSize (int n);  // set size of vector
}
```

FIGURE 2 – Définitions (simplifiées) issues de la bibliothèque de Java

```
module Hashtbl : sig
  type ('a, 'b) t
  val create: int -> ('a, 'b) t
  val add: ('a, 'b) t -> 'a -> 'b -> unit
  val find: ('a, 'b) t -> 'a -> 'b
end
```

FIGURE 3 – Définitions issues de la bibliothèque d'OCaml

Dans une application où les coefficients binômiaux seraient très fréquemment utilisés, on aimerait éviter de calculer plusieurs fois la valeur d'un même `c (n, p)`. On aimerait donc définir une fonction `memo_c`, de même type que la fonction `c`, telle que, si on appelle `memo_c (n, p)` une première fois, alors `c (n, p)` est appelée et le calcul est effectué, mais si ensuite on appelle à nouveau `memo_c (n, p)` pour les mêmes entiers `n` et `p`, alors le résultat déjà calculé est renvoyé immédiatement. Cette technique est appelée mémoïsation.

Cette idée peut s'appliquer en fait non seulement à la fonction `c` de type `int * int -> int`, mais à n'importe quelle fonction `f` de type `'a -> 'b`.

Pour mémoriser les résultats des calculs déjà effectués, on propose d'employer une table de hachage, de type `('a, 'b) Hashtbl.t`. La figure 3 rappelle les principales fonctions proposées par la bibliothèque d'OCaml concernant les tables de hachage. `create n` crée une nouvelle table. Son argument entier `n` est une suggestion de taille de la table, mais n'a pas d'importance, car la table est auto-redimensionnable. `add t x y` ajoute à la table `t` une association de la valeur `y` à la clef `x`. Enfin, `find t x` recherche la valeur associée à la clef `x` dans la table `t`, et lance l'exception `Not_found` si cette clef n'apparaît pas dans la table.

Question 9 Écrivez une fonction `memo` de type `('a -> 'b) -> ('a -> 'b)` telle que, si `f` est une fonction de type `'a -> 'b`, alors l'appel `memo f` alloue une nouvelle table de hachage, l'utilise pour construire une version mémoïsante de la fonction `f`, et renvoie cette fonction. ◇

Question 10 Supposons donnés deux entiers `n` et `p`. On effectue les définitions suivantes, où `c` est la fonction définie lors de la question 7 :

```
let memo_c = memo c
let _ = memo_c (n, p)
let _ = memo_c (n, p)
```

Quelle est la complexité en temps de chacun de ces trois appels de fonction ? Exprimez-la en fonction du paramètre n. ◇

Question 11 Que pensez-vous des résultats obtenus lors des questions 9 et 10 ? La complexité des fonctions `c` ou `memo_c` est-elle satisfaisante ? Pourrait-on mieux exploiter la mémoïsation pour obtenir un calcul plus efficace ? Si oui, proposez une nouvelle implémentation de la fonction `memo_c`. Quelle est sa complexité asymptotique en temps ? ◇