C. Charignon

Table des matières

Première partie
Cours

1 Définition du type

Nous fixons deux ensembles C et V. L'ensemble C sera appelé l'ensemble des « clefs » et V l'ensemble des « valeurs ». Un dictionnaire permet d'associer à un élément de C un élément de V. Autrement dit, un dictionnaire est une fonction de C dans V. Il s'agit d'un type très versatile, qu'on utilise dans de nombreuses situations.

Selon le point de vue adopté ici, une clef peut ne pas avoir de valeur associée, autrement dit un dictionnaire n'est pas forcément une « application » de C vers V (pas forcément définie sur C entier).

Les opérations permises par le type des dictionnaires sont :

- Créer un dictionnaire vide.
- Ajouter une association : étant donné $c \in C$ et $v \in V$, décider que v sera associée à c. Si une valeur était déjà associée à c, la nouvelle écrase l'ancienne.
- Lire la valeur associée à une clef : étant donné $c \in C$, renvoyer la valeur associée à c. Si aucune valeur n'est associée à c, une erreur est déclenchée.

2 Dictionnaire persistant par liste d'association

Une liste d'association est une liste de couples de la forme `(clef, valeur)`.

On programme facilement la recherche et l'ajout d'une association dans une telle liste. La fonction de recherche est déja présente dans OCaml : c'est `List.assoc`.

Le défaut de cette stratégie est que la recherche est en $O(n)$, où n est le nombre d'éléments dans le dictionnaire. On la réservera donc aux situations où l'efficacité n'est un critère primordial.

3 Dictionnaire persistant par arbres binaires de recherche

Dans un ABR, l'opération de recherche est efficace : pour peu que l'ABR a soit équilibré, la fonction `appartient` a une complexité en $O(\log n)$, où n est le nombre de nœuds de a.

Par une petite modification, on peut obtenir une structure de dictionnaire (persistant) : il suffit de mettre à chaque nœud un couple `(cle,valeur)`.

C'est une méthode efficace pour obtenir un dictionnaire persistant.

```
1 type ('a, 'b) dictionnaire = ('a * 'b) arbre;;
```

Programmons alors les opérations de base :

- `assoc`
- `insere` Il y a une subtilité ici : on ne peut pas se contenter d'utiliser la fonction d'insertion des arbres binaires de recherche déjà programmée car dans le cas où la clef serait déjà présente, la novelle entrée serait insérée en dessous de l'ancienne. Nous allons donc reprogrammer cette fonction en prenant soin que dans le cas d'une clef déjà existante, la nouvelle entrée *remplace* l'ancienne.
  Remarque : Dans d'autres implémentations (listes d'association et tables de hachage à venir, une nouvelle entrée viendrait recouvrir l'ancienne, mais l'ancienne resterait en mémoire, plus loin dans la liste. Donc si ensuite on supprime cette nouvelle entrée, l'ancienne réapparaît. Par contre, cela gaspille un peu de mémoire, ralentit un peu les recherches.
- `est_une_clef` Pour savoir si il existe une entrée correspondant à une certaine clef, on pourrait lancer `assoc` et rattraper une éventuelle erreur. Mais il sera plus propre d'écrire cette fonction. Notons cependant qu'en pratique, de tester si une clef est présente puis de chercher la valeur associée prendrait deux parcourts de l'arbre : la méthode en rattrapant l'erreur serait deux fois plus rapide.

4 Dictionnaire mutable par table de hachage

4.1 Syntaxe Caml

Voici la syntaxe d'un dictionnaire mutable en Ocaml. Les fonctions se trouvent dans la bibliothèque `Hashtbl` (”hash table”, cf plus loin l'explication de ce nom).

- Nouveau dictionnaire de taille n : `Hashtbl.create n`.
  La taille est juste le nombre de cases à allouer lors de la création du dictionnaire, mais il sera automatiquement agrandi si besoin.
- Ajouter un élément *val* à la clé *cle* : `Hashtbl.add dico cle val`
- Chercher l'élément correspondant à la clé *cle* : `Hashtbl.find dico cle`. Ceci renvoie l'erreur `Not_found` si il n'y a pas d'entrée pour cette clé.
  Comme il n'y a pas de fonction pour tester si il y a une entrée pour une clé donnée, on est obligé de passer par cette erreur parfois...
- Supprimer l'élément correspondant à la clé *cle* : `Hashtbl.remove dico cle`.

4.2 Table de hachage

La table de hachage est la méthode la plus fréquente pour réaliser un dictionnaire mutable.

Pour l'utiliser, on a besoin d'une fonction, appelé « fonction de hachage » qui à n'importe quelle donnée associe un nombre entier. Ci-dessous, on supposera donnée une fonction de hachage f, qui fonctionne en temps $O(1)$, sans se préoccuper de la manière dont elle est définie...

La recherche d'une bonne fonction de hachage est en général une opération importante qui peut être difficile. Si on a pour but d'utiliser un dictionnaire, il peut être judicieux de penser à la future fonction de hachage au moment de créer le type qu'on va utiliser (« hash-consing »).

Sous OCaml, utiliser `Hashtbl.hash`, de type `'a -> int`.

Voyons maintenant le principe d'une table de hachage. On choisit un entier $k \in \mathbb{N}^*$. On crée un tableau redimensionnable de k cases. Chacune de ces k cases contiendra une liste d'association, et les éléments à enregistrer seront répartis parmi ces k listes d'association.

Soit c une clé, et d la donnée correspondante. Le couple (c, d) sera enregistré dans la liste d'association de la case `Hashtbl.hash c % k` du tableau.

Ainsi, une bonne fonction de hachage devra faire en sorte que les différentes données sont réparties de manière aussi uniforme que possible entre les différentes cases du tableau.

En notant n le nombre d'entrées de la table, il y aura en moyenne n/k éléments par case, ce nombre s'appelle le « facteur de compression » de la table de hachage (« load factor » en anglais).

Lorsque le facteur de compression devient trop grand, on agrandit le tableau. Typiquement, on double sa taille.
Remarque : En Python, les dictionnaires sont agrandis dès que le facteur de compression dépasse 2/3.

Complexité des opérations de base :
Dans un premier temps si on n'implémente pas l'agrandissement automatique du tableau :

- *Ajout :* $O(1)$
- *Recherche :* Le temps pour rechercher la donnée correspondant à une clé c est le temps de calculer `hash c`, d'aller à la case correspondante, puis de chercher dans la liste qui s'y trouve. Total : $O(1) + O($ longueur de la liste $)$.
  - ◇ *Complexité au pire :* Le pire est que toutes les données soient dans la même case ! Dans ce cas la complexité est $O(n)$.
  - ◇ En général, si la table est bien équilibrée (bonne fonction de hachage), le nombre de données par case est limité. S'il existe une constante α telle que chaque case contient au plus α données, alors la recherche s'effectue en $O(1)$.
  - ◇ *Complexité moyenne :* supposons les données réparties aléatoirement, de sorte que chaque donnée a la probabilité $\frac{1}{k}$ d'aller dans chaque case, et calculons l'espérance de la complexité.
    Fixons une case c. Pour tout $i \in [\![1, n]\!]$ notons X_i la variable aléatoire de Bernoulli qui vaut 1 lorsque la i-ème donnée est dans la case c. Le nombre de données dans la case c est donc $\sum_{i=1}^{n} X_i$, et son espérance est $\sum_{i=1}^{n} E(X_i) = \frac{k}{n}$.
    Si nous supposons les clé équiprobables, une recherche à la même probabilité d'aboutir dans chaque case. Et finalement, l'espérance de la longueur de la liste rencontrée pour la recherche est $\frac{k}{n}$. La complexité est donc $O(1) + O(frackn)$.

Maintenant, si on décide d'agrandir le tableau dès que le facteur de compression devient trop grand :

- ◇ *Ajout :* la plupart du temps en $O(1)$ mais lorsqu'il y a un agrandissement à faire en $O(n)$ puisqu'il faut alors recopier le tableau dans un tableau plus grand.

Calculons la "complexité amortie" de l'ajout de données, c'est-à-dire la complexité moyenne pour partir d'une table vide et y rajouter les uns après les autres n éléments.

Calcul déjà fait lors de l'étude des tableaux redimensionnables.

Fixons donc $n \in \mathbb{N}^*$ et ajoutons n éléments les uns après les autres dans une table initialement vide. Notons k le nombre d'agrandissement nécessaires. Ainsi k est le plus petit entier tel que $n \leqslant 2^k$, nous avons donc :

$$2^{k-1} < n \leqslant 2^k.$$

Remarque : En formule, $k = \left\lceil \log_2 n + 1 \right\rceil$, partie entière "supérieure" de $\log_2 n + 1$.

La complexité d'un agrandissement est linéaire en la longueur du tableau. Notons $\alpha \in \mathbb{R}^{+*}$ une constante telle que pour tout $i \in \mathbb{N}$ l'agrandissement d'un tableau de longueur i a une complexité inférieure à αi (la constante du $O(i)$ en gros).

Notons β la constante pour l'ajout d'un élément dans la table.

Notons C_n la complexité de l'ajout des n éléments. Alors :

$$\begin{aligned} C_n &\leqslant \sum_{i=1}^{k} \alpha 2^i + \beta n \\ &= \alpha(2^{k+1} - 1) + \beta n \\ &\leqslant (4\alpha + \beta) n. \end{aligned}$$

Ainsi, $C_n = O_{n\to\infty}(n)$. La complexité pour insérer successivement n éléments est $O(n)$. On peut donc dire que la complexité moyenne pour chaque élément est $O(1)$.

Remarque : Pour être plus fin :

- On peut quand même dire que l'ajout d'un élément est en moyenne 5 ou 6 fois plus lent que pour un simple tableau, puisque le coût pour un élément dans un tableau est de β alors qu'ici le coût moyen de chaque ajout est $4\alpha + \beta$.
- Par contre le calcul précédent correspond au pire des cas. En effet si on n'ajoute pas successivement n éléments mais qu'on en supprime entre temps, on aura moins d'agrandissements à faire.

◇ *Recherche :* en notant λ le maximum maintenu du facteur de compression (par exemple $\frac{2}{3}$ en Python), on a une complexité en $O(1) + O(\lambda) = O(1)$ comme vu précédemment.

4.3 Bonus : en python

Voici la syntaxe d'un dictionnaire en python (hors programme mais extrêmement pratique pour ceux qui feraient un TIPE avec Python par exemple). Elle est très proche de celle du type `list`.

- Créer un dictionnaire : `dico = { clé1 : val1 ; clé2:val2; ...}`
- Récupérer la valeur correspondant à une clé : `dico[cle]`
- Modifier / rajouter une valeur pour une clé : `dico[cle]=val`
- Liste des toutes les clés utilisées : `dico.keys()`
- D'où, pour tester si il y a une entrée correspondant à une clé : `cle in dico.keys()`

5 Ensemble

Dans plusieurs situations, on a juste besoin d'une structure d'ensemble. Les opérations de base étant :

- Rajouter un élément dans l'ensemble
- Tester si un élément est dans l'ensemble

Il suffit de prendre un dictionnaire et de n'utiliser que les clés. C'est pourquoi dans certains cas vous verrez qu'on utilise un dictionnaire sans jamais utiliser les valeurs enregistrées.

Ainsi, pour une structure d'ensemble persistant, un arbre binaire de recherche fait très bien l'affaire (à condition le type des éléments à rassembler soit muni d'une relation d'ordre).

Pour une structure d'ensemble mutable, prendre une table de hachage, et dans chaque case enregistrer une liste d'éléments de l'ensemble. Si on veut utiliser les tables de hachage de Caml, mettre un élément quelconque (par exemple 0) pour toutes les valeurs.

cf exercice : 2

Deuxième partie
Exercices

Exercices : dictionnaires

Exercice 1. ** Tri par dénombrement (sur tableaux)

Pour cet exercice, on utilisera le module `Hashtbl` de Caml.

1. Deux petites fonctions sur les dictionnaires :
   - `incr_dico` qui prend un dictionnaire d'entiers et une clef, et qui augmente de 1 la valeur associée à la clef s'il y en a, ou l'initialise à 1 sinon.
   - `nb_de` qui prend un dictionnaire d'entiers et une clef, et renvoie la valeur associée à la clef s'il y en a, ou 0 sinon

   Pour ces deux fonctions on pourra utiliser la structure `try ... with |Not_found -> ....`
2. Écrire une fonction `compte` prenant en entrée un tableaux `t` et renvoyant un dictionnaire associant à chaque élément x de `t` le nombre de fois que x apparaît dans `t`. On utilisera ici une table de hachage de Caml comme dictionnaire.
3. Modifier la fonction précédente pour qu'elle renvoie en outre le minimum et le maximum de `t`.
4. En déduire une implémentation du tri par dénombrement.
5. Quelle est la complexité de ce tri ? Dans quelles situations est-il judicieux de l'utiliser ?

Exercice 2. ** Dédoublonnage

Écrire une fonction qui prend en entrée une liste `l` et qui renvoie un dictionnaire dont les clefs sont les éléments de `l`. En déduire une fonction de dédoublonnage. Quelle est sa complexité ?

Exercice 3. * calcul d'un élément majoritaire

Dans une élection par scrutin uninominal, on récupère dans un tableau les noms inscrits sur les bulletins de vote. Par exemple `[| "Durand"; "Dupont"; "Durand"; "Moulin",...|]`.

Écrire une fonction pour déterminer s'il y a un gagnant au premier tour et si oui lequel. On ne s'autorisera qu'un seul parcours du tableau.

Exercice 4. **! Numérotation

1. Écrire une fonction `numerote` prenant en entrée une liste `l` et renvoyant un couple `(l', d)` tel que :
   - `l'` est la liste des éléments de `l` sans doublon ;
   - `d` est un dictionnaire associant à chaque élément de `l'` son « numéro » c'est-à-dire sa place dans `l'`.

   Cette fonction sera utile à plusieurs reprise en seconde année car elle permet d'associer à n'importe quelle liste de n'importe quel type d'éléments des numéros qui les identifient de manière unique. Par exemple nous pourrons numéroter les sommets d'un graphe afin de créer la matrice qui indique comment sont reliés ces sommets.

2. *Amélioration :* Le dictionnaire obtenu permet de retrouver facilement le numéro d'un élément. Mais il serait utile de pouvoir également retrouver rapidement l'élément correspondant à un numéro donné.
   (a) Si pour chercher l'élément de numéro i, on va chercher l'élément d'indice i dans `l'`, quelle est la complexité ?
   (b) Quelle structure de donnée permettrait une meilleure complexité ?
   (c) Modifier votre programme pour qu'il renvoie le dictionnaire `d` et la structure de donnée que vous avez proposée à la question précédente.
   (d) *Plus sophistiqué mais élégant :* Au lieu de renvoyer le dictionnaire et l'autre objet, renvoyer deux fonctions `numero_of_element` et `element_of_numero`, réciproques l'une de l'autre, dont j'espère le nom suffisamment explicite pour se passer de description.

   Commentaire : Pour définir le cardinal d'en ensemble E, on trouve un entier n et une bijection $\phi : [\![0, n[\![\to E$. On dit alors que E est de cardinal n, et ϕ s'appelle une « énumération » de E. C'est exactement ce que nous avons fait ici : l'énumération est `element_of_int`.

Exercice 5. ** Compression zip (très) simplifiée

1. Écrire une fonction récursive `liste_de_mots_of_string` de type `string -> string list` prenant en entrée une chaîne de caractères et renvoyant la liste de ses mots (les mots étant supposés séparés par des espaces). C'est donc l'analogue de la méthode `split(" ")` de Python.
2. Écrire une fonction `compresse` qui prendra en entrée une liste de mots `l` (telle que renvoyée par la fonction précédente) et qui renverra un couple formé de :

(a) un dictionnaire tel que celui obtenu dans l'exercice 4, qui permet de numéroter les éléments le `l` ;

(b) une liste d'entiers obtenue en remplaçant chaque élément de `l` par son numéro.

3. Écrire une fonction `decompresse` réciproque de la précédente.

4. *Mise en pratique :* À présent, on veut écrire et lire les données compressées dans un fichier.

(a) Écrire une procédure `liste_vers_fichier` permettant d'enregistrer le contenu d'une liste dans un fichier. On convient d'écrire un élément par ligne, en commençant par l'élément en tête de liste.

(b) Écrire une procédure analogue `dico_vers_fichier`. On convient d'utiliser une ligne par entrée du dictionnaire, chaque ligne étant de la forme `mot numéro` (on utilise un espace pour séparer le mot de son numéro, ce qui permettra d'utiliser `liste_de_mots_of_string` lors du décodage).

(c) Écrire une fonction `liste_de_mots_of_fichier` qui prend un fichier enregistré dans un fichier .txt et renvoie la liste de mots correspondants.

(d) Rassembler tous les morceaux pour obtenir une procédure prenant en entrée un fichier texte et créant un fichier contenant le texte compressé. Vérifier si le fichier obtenu est plus léger que le fichier source.

(e) Écrire la fonction de décodage correspondante.