université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP 7

Présentation

Le but du TP est de se familiariser avec les tables de hachage en générale et avec leur implémentation en OCaml. On retravaillera aussi les aspects impératifs d'OCaml. exercices chacun situé dans un sous-répertoire (`exo1` et `exo2`). Il est fortement conseillé de se placer dans le répertoire contenant ces deux sous-répertoire. Chacun des exercices à la même structure :

— un fichier `main.ml` et d'autres fichiers `.ml` contenant le code OCaml que vous devez compléter

— un fichier `dune` contenant les instructions de compilation

La commande `dune build` construit les deux exécutables `exo1/main.exe` et `exo2/main.exe`

1 Implémentation des tables de hachage

Le but est de se familiariser avec l'implémentation des tables de hachage. Il est demandé d'implémenter les fonctions permettant de rechercher puis d'ajouter dans une table de hachage.

Il est suggéré de ne *pas* aller voir le cours, mais plutôt d'essayer de ré-implémenter les fonctions en se souvenant du principe. En cas de blocage, évidemment, les supports de cours sont là pour aider [1]

1. Lire le code du module `HT`. Ce dernier commence par définir deux synonymes pour les fonctions prédéfinies `Hashtbl.hash` et `(=)`. On vous demande d'utiliser ces re-définitions dans votre code (pour pouvoir expérimenter avec une mauvaise fonction de hachage).
2. Compléter le code de `find : ('k, 'v) t -> 'k -> 'v`. Cette dernière prend en argument une table et une clé et renvoie la valeur associée. Si la valeur est absente, la fonction lève l'exception `Not_found`.
3. Compléter le code de `simple_add : ('k, 'v) t -> 'k -> 'v -> unit`. Cette fonction remplace la valeur associée à la clée donnée si elle est présente et ajoute la paire clé, valeur sinon.
4. Compléter le code de la fonction `iter : ('k -> 'v -> unit) -> ('k, 'v) t -> unit`, qui applique une fonction donnée à chaque clé et valeur de la table. Attention, la fonction ne prend pas directement une paire comme argument mais prend la clé et la valeur comme deux arguments séparés.
5. Compléter une fonction `resize: ('k,'v) t -> unit` qui redimensionne la table en doublant la taille du tableau sous-jacent. Attention, la fonction en renvoie pas une nouvelle table mais modifie la table passée en argument.

   Il pourrait être judicieux de créer une table temporaire et d'y ré-ajouter les valeurs au moyen de `iter` et `simple_add`.
6. Compléter la fonction `add : ('k, 'v) t -> 'k -> 'v -> unit` qui appelle `simple_add` puis qui contrôle que le nombre d'élément total n'est pas supérieur à deux fois la taille du tableau sous-jacent. Si c'est le cas, la fonction appelle resize pour agrandir ce tableau.
7. Écrire une fonction `debug : ('k, 'v) t -> ('k -> string) -> ('v -> string)` qui affiche le contenu de la table de la façon suivante. Chaque case du tableau sous-jacent est affiché sur une ligne. Le contenu de la case est affiché comme une liste de triplet, avec la clé, le hash de la clé et la valeur. Par exemple pour une table contenant les clé de 0 à 10, toutes associées à la valeur `"A"`, on souhaite afficher :

```
0 -> [(2, 648017920, A); (6, 899338544, A); (9, 531229256, A)]
1 -> []
2 -> [(0, 129913994, A)]
3 -> [(1, 883721435, A); (10, 875313035, A)]
4 -> [(8, 894170852, A)]
```

1. les typos dans le code des supports ont été corrigées.

```
5 -> [(3, 152507349, A); (7, 631987845, A)]
6 -> []
7 -> [(4, 127382775, A); (5, 378313623, A)]
```

Les deux fonctions prises en argument en plus de la table permettent de convertir les clés et les valeurs en chaînes.

8. À l'extérieur du module `HT` écrire une fonction de test qui
   — crée une table
   — y insère des clés entières de 0 à 32 toutes associées à la valeur `"A"`
   — après chaque insertion, efface le terminal au moyen de la fonction `clear_screen` fournie, affiche la table au moyen de la fonction `HT.debug`, appelle `flush stdout` pour forcer l'affichage de l'écran et enfin fait une pose de 1 seconde avec l'expression `Unix.sleep 1`.

   Constater que la table se redimensionne correctement et que les listes ne deviennent jamais trop profondes.

2 Collisions

On souhaite essayer de déterminer si deux chaînes de caractères « normales » ont des chances d'être en collisions, c'est à dire d'obtenir la même valeur par une fonction de hachage. On utilisera pour cela le fichier `french.txt`. Ce dernier contient une liste de 346200 mots français, utilisés pour la correction orthographique sous Linux.

1. écrire une fonction

   ```
   collect_collisions : (string -> int) -> string -> (int, string list) Hashtbl.t
   ```

   Cette dernière prend en argument une fonction de hachage, un nom de fichier supposé contenir un mot par ligne. Cette fonction renvoie une table de hachage (en utilisant le type OCaml et le type de l'exercice 1). La table doit associer, pour toute valeur de hash calculée la liste des mots ayant cette valeur de hash. Les fonctions utiles pour cette question sont :
   — `open_in : string -> in_channel` qui prend un nom de fichier et renvoie un descripteur ouvert en lecture
   — `In_channel.input_line : in_channel -> string option` qui renvoie soit la valeur `None` si on est en fin de fichier soit une valeur `Some s` où `s` est la ligne courante dans le fichier dont le descripteur est donné
   — `Hashtbl.create : int -> ('k, 'v) Hashtbl.t` crée une table de hachage. L'entier donné permet de donner une taille initiale au tableau sous-jacent (mais le tableau est de toute façon redimensionné si besoin).
   — `Hashtbl.find : ('k, 'v) Hashtbl.t -> 'k -> 'v` renvoie la valeur associée à la clé donnée ou lève l'exception `Not_found` si la clé n'est pas dans la table.
   — `Hashtbl.replace : ('k, 'v) Hashtbl.t -> 'k -> 'v -> unit` qui associe la clé et la valeur données dans la table [2]

   Indication vous pouvez réfléchir en groupe à l'algorithme à utiliser (qui n'est pas très dur) dans un premier temps, puis dans un second temps écrire cet algorithme en OCaml.

> **Réponse:** On souhaite trouver les mots ayant la même valeur de hachage. Le type de retour de la fonction demandée donne une indication sur la façon de procéder. On veut renvoyer une table associant des entiers (le hash) à une liste de chaînes (la liste des mots ayant ce hash). On procède donc comme suit :
> — déclarer une variable locale contenant une table (vide)
> — ouvrir le fichier
> — dans une boucle ou une fonction récursive, pour chaque ligne (= chaque mot) du fichier :
> — calculer son hash
> — rechercher si son hash est déjà dans la table de hachage et renvoyer la liste de mots l associée
> — si ce n'est pas le cas, renvoyer la liste vide pour l

2. La fonction `Hashtbl.add` a un comportement particulier qu'on ne souhaite pas avoir pour ce TP.

— si le mot n'est pas déjà dans l, alors l'ajouter et remettre l dans la table
— une fois arrivé en fin de fichier, renvoyer la table

On utilise ici des listes car on suppose le nombre de collisions faible par rapport au nombre total de mots. Rechercher si un mot est déjà dans la liste est linéaire. On aurait pu remplacer cette liste par un `Set` (vérification en temps logarithmique) où même par une table de hachage (vérification en temps constant amorti).

2. Écrire une fonction `display_collisions : (int, string list) Hashtbl.t -> unit` qui affiche dans la console, pour chaque valeur de hash ayant des collisions (i.e. plusieurs mots ont cette valeur de hash) la liste des mots ainsi que le nombre total de valeur de hash ayant des collisions et la taille de la collision maximale.
3. Donner une implémentation de la fonction de hachage des chaînes de Java. Cette dernière calcule :

$$\sum_{i=0}^{n-1} 31 c_i$$

où c_i est le code du $i^{\text{ème}}$ caractères de la chaîne. Afficher maintenant les statistiques de collisions.

Rappel en OCaml on peut calculer la longueur d'une chaîne avec `String.length`, accéder au `i`$^{\text{ème}}$ caractère avec `s.[i]` (avec point et crochets), et on peut obtenir le code d'un caractère avec `Char.code`