université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP 8

Présentation

Le but du TP est de se familiariser avec la structure de *trie* (« tri » ou « traille », comme vous voulez). Il convient d'avoir lu le fichier `trie.ml` disponible sur la page du cours. Le fichier en ligne contient des exemples et de nombreux commentaires qui ne sont pas dans le fichier du TP (pour ne pas alourdir trop ce dernier visuellement).

Cette feuille pourra potentiellement durer 2 séances (pour les parties 1, 2 et 3, la partie 4 étant juste du bonus).

1 Opérations sur les *tries*

1. Écrire la fonction d'ajout dans un *trie* :

   ```
   add : string -> 'a -> 'a trie -> 'a trie
   ```

   Cette fonction prend en argument une clé, une valeur et un *trie* et renvoie le *trie* dans lequel la valeur a été ajoutée pour la clé. Le principe est similaire à celui de la recherche. On parcours la clé caractère par caractère. On appelle nœud courant, la valeur `Node(o, l)` sur laquelle on est.

   — si on est en fin de chaîne, alors on peut créer un nœud `Node (Some v, l)`, où `v` est la valeur à ajouter (la fonction d'ajout remplace donc une éventuelle ancienne valeur s'il y en avait une).

   — sinon on est sur le $i^{\text{ème}}$ caractère `ci`, on parcourt la liste des couples `(c, t)` de la liste `l` :

   — si `c` est égal à `ci`, alors ajouter récursivement la valeur `v` dans le sous-arbre `t` en avançant au caractère $i+1$;

   — si `c` est plus petit que `ci`, alors ajouter `v` dans la suite de la liste, (on n'oubliera pas de remettre le couple `(c, t)` en début de liste)

   — si `c` est plus grand que `ci` ou que la liste est vide, alors il suffit d'ajouter la valeur `v` dans l'arbre `empty` en avançant au $(i+1)^{\text{ème}}$ caractère

2. Tester votre fonction en ajoutant les associations suivante dans un *trie* initialement vide :

$$\{\texttt{"a"} \mapsto 10, \texttt{"abc"} \mapsto 11, \texttt{"abd"} \mapsto 12\}$$

verifier que vous retrouvez bien ces valeurs avec `find`. Vérifier aussi que `find_trie` sur la clé `"ab"` vous renvoie bien un nœud non terminal.

3. Écrire la fonction `union : ('a * 'a -> 'a) -> 'a trie -> 'a trie -> 'a trie` qui fait l'union de deux *tries*. Cette fonction est telle que `union choose t1 t2` renvoie le trie contenant les clés de `t1` et celles de `t2`. Pour les valeurs associées à ces clés :
   — si une clé n'est présente que dans `t1` ou dans `t2`, la valeur associée est utilisée dans le résultat
   — si une même clé est présente dans les deux *tries*, alors les valeurs associées sont passées (sous forme d'un couple) à la fonction `choose` qui choisira l'une des deux valeurs. Par exemple, `union fst t1 t2` renvoie l'union des deux *tries* et choisi les valeurs de `t1` en cas de conflit.
4. Utiliser les deux listes `keys1` et `keys2` pour construire deux *tries*. Le premier *trie*, `t_add` est construit en ajoutant tour à tour les clés de `keys1` et `keys2` On choisira d'associer la même valeur pour toutes les clés (par exemple 1). Construire maintenant un *trie* `t_union` résultant de l'union d'un *trie* construit à partir de `keys1` et d'un autre construit à partir de `keys2`.
5. D'après vous est-ce que `t_add` et `t_union` auront la même forme ? Vous pouvez tester en regardant le résultat de l'expression OCaml `t_add = t_union` (on rappelle que `=` est l'égalité structurelle qui parcours récursivement deux valeurs du même types tester leur égalité). Est-ce une propriété que l'on peut avoir avec les tables de hachage ou les arbres binaires de recherche ? (justifier).

2 Complétion

On souhaite maintenant illustrer une application particulièrement importante des *trie* : la complétion d'une chaîne donnée par un suffixe.

On implémente pour cela l'algorithme du cours qui permet de renvoyer la liste des clés d'un *trie*. Dans cette partie, on ignore les valeurs et on se contente de tester si les nœuds sont terminaux (`Some _`) ou non-terminaux (`None`) (en d'autres termes on considère nos *trie* comme des ensembles).

1. Écrire une fonction `rev_implode : char list -> string` qui renverse une liste de caractères

et les assemble en une chaîne.

Les fonctions `List.rev_map`, `String.concat` et `String.make` peuvent être utilisées pour écrire cette fonction simplement. Ces dernières sont documentées ici :

— https://v2.ocaml.org/releases/4.14/api/List.html

— https://v2.ocaml.org/releases/4.14/api/String.html

2. Écrire maintenant une fonction `all_keys : 'a trie -> string list` qui renvoie toutes les clés d'un *trie* donné en argument, dans l'ordre croissant.
3. En déduire une fonction `complete : string -> 'a trie -> string list` qui étant donné un *trie* et une chaîne revoie toutes les suffixes de cette chaîne présent dans le *trie*.
4. Écrire un code de test qui :
   — charge le fichier `french.txt` fourni dans un *trie*
   — effectue ensuite une boucle qui demande à l'utilisateur une chaîne de caractère et affiche toutes les complétions possibles du dictionnaire français. On pourra utiliser la fonction `In_channel.input_line` pour lire le fichier (cf. TP7) et la fonction `read_line ()` pour lire les saisies de l'utilisateur.

3 Comparaisons

Écrire du code de test qui

— charge une liste de mots dans une table de hachage, dans une map (en utilisant un module `module StrMap = Map.Make(String)`) et dans un trie, et qui associe pour chaque mot du dictionnaire l'entier 1.

— effectue un grand nombre (par exemple 100000) recherches dans un mot que vous savez être dans le dictionnaire

(vous devez donc écrire 6 fonctions, une pour chaque type de test et chaque structure de données). Testez vos fonctions avec les mots du fichier `french.txt` et mesurer leur temps d'exécution avec la fonction ci-dessous,

```
1  let time f arg msg =
2    let () = Gc.full_major () in
3    let () = Gc.compact () in
4    let t0 = Unix.gettimeofday () in
5    let res = f arg in
6    let t1 = Unix.gettimeofday () in
7    Printf.printf "\%s: %fms\n" msg (1000. *. (t1-.t0));
```

```
8     res
```

L'appel `time f arg msg` calcule le temps passé dans le calcul de `f arg`, une fois ce calcul fait, il affiche le temps écoulé en ms puis renvoie la valeur calculée. La fonction force un appel au *garbage collector* pour éviter que ce dernier ne se déclenche de façon intempestive pendant l'exécution de `f`. Proposer des hypothèses pour ces temps d'exécution, elles seront discutées en cours.

4 Dessine moi un *trie*

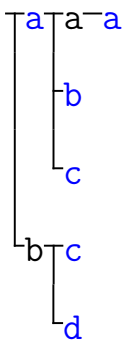
On se propose de dessiner un *trie* en ASCII-art, comme dans les supports de cours en utilisant :

— des séquences d'échappement ANSI pour mettre les nœuds terminaux en bleu

— des caractères Unicode de dessin de « boîtes » pour les liens dans l'arbre

Les caractères Unicode en question sont disponible sur la page :

https://en.wikipedia.org/wiki/Box_Drawing



Pour afficher un caractère dans le terminal, on peut utiliser le format `"%c"`. Attention cependant des « caractères » tels que `"┬"` ne sont représentable que par des séquences UTF-8 (le code de ces caractères fait plus que un octet). Ils ne sont donc pas représentables par le type `char` d'OCaml. On pourra donc écrire `Printf.printf "%s" "┬"`, mais pas `Printf.printf "%c" '┬'`

Un affichage en couleur dans le terminal peut être fait au moyen de la séquence :

`Printf.printf "\x1b[`n`m.....\x1b[0m"`

Ou n est le code de la couleur que l'on souhaite utiliser et `.....` le texte à afficher. Les codes couleurs sont documentés ici :

https://en.wikipedia.org/wiki/ANSI_escape_code