université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP Noté

Durée 1h30, tiers-temps aditionnel 30 minutes

Documents autorisés : documents sur le site du cours.

Lire en entier les consignes suivantes

Le TP comporte 3 exercices et est sur 20 points. Le barème est indicatif et reflète la difficulté, et donc le temps à passer sur chaque exercice. Il est aussi conseillé de laisser du code en commentaire si ce dernier ne compile pas ou qu'il donne un mauvais résultat, des points pourront tout de même être attribués.

La consultation de *documents de la page du cours* est autorisée. Les logiciels de messagerie et les IA génératives type ChatGPT, Copilot, ... ne sont pas des documents. De même les sites du style StackOverflow ou tout autre source d'information non présente sur la page du cours n'est pas un document de cours. Leur utilisation sera considérée comme une tentative de fraude.

Description de l'archive

Télécharger sur le site du cours l'archive du TP noté. Décompresser l'archive. Cette dernière contient les fichiers et répertoires suivants :

`infos.txt :` un simple fichier texte **à compléter** avec votre nom, votre prénom et votre adresse email

`init_tp.sh :` le script initialisant OCaml et VSCode dans la session de secours

`exo1/exo1.ml, exo2/exo2.ml, exo3/exo3.ml :` les trois fichiers de code à compléter

`rendu.sh :` un script générant une archive contenant tout vos fichiers, à déposer via le formulaire disponible sur la page du cours

Rendu du TP

Renseigner votre nom, prénom, adresse email (de préférence `@etu-upsaclay.fr` sinon adresse personnelle) dans le fichier `infos.txt`.

Se placer dans le répertoire contenant le script `rendu.sh` et l'exécuter :

```
./rendu.sh
```

Le script liste les fichiers ajoutés dans l'archive. L'archive se trouve dans le répertoire supérieur avec le nom `rendu-pfa-tp-note_l3_info_2024_`d`_`h`.tar.gz` où d et h sont la date et l'heure de création de l'archive. **Vous devez déposer l'archive créé via le formulaire sur la page du cours**, en saisissant le même email que celui présent dans le fichier `infos.txt` (une vérification est faite). Vous pouvez déposer autant de fois que vous voulez. Il est suggéré de soumettre fréquemment.

Initialisation, à faire une seule fois au début

Se placer dans le répertoire du TP noté et exécuter la commande

```
$  ./init_tp.sh
```

Il faut en suite **refermer et rouvrir le terminal**.

Code à compléter

Les trois exercices se trouvent chacun dans un sous-répertoire `exo1`, `exo2` et `exo3`. La commande `dune build` construit les trois exécutables `exo1/exo1.exe`, `exo2/exo2.exe` et `exo3/exo3.exe`. Pour chaque question, **la fonction à compléter ainsi que toutes vos fonctions auxiliaires non demandées** doivent se trouver **entre les commentaires** `(* Question X *)` et `(* Fin Question X *)`.

Exercice 1 (7 points)

On se donne un type de données permettant de modéliser des films :

```
type movie =
  { id : int; title : string; year : int; runtime : int; rank : int }
```

Un film est un enregistrement contenant un identifiant unique, un titre, une année de sortie, une durée en minutes et un classement dans la liste des meilleurs films (un entier, il peut y avoir 2 films avec le même numéro de classement). On se donne de plus un fichier texte contenant des films, dont le format est le suivant :

```
2065;The Treasure of the Sierra Madre;1948;126;66
2115;Million Dollar Baby;2004;132;116
2089;No Country for Old Men;2007;122;90
2009;Star Wars : Episode V - The Empire Strikes Back;1980;190;10
...
```

Le but de l'exercice est de charger ce fichier comme une liste de films puis d'écrire des fonctions permettant d'effectuer certaines requêtes sur une liste de films. **Du code vous permettant de tester vos fonction se trouve en fin de fichier.**

1. Compléter la fonction `parse_movie : string -> t` qui prend en argument une chaîne de caractères correspondant à une ligne du fichier et renvoie l'enregistrement correspondant. Si la ligne est mal formée, votre fonction peut lever une exception (par exemple avec `failwith "erreur"`). Le code fourni en fin de fichier utilise cette fonction pour lire le fichier `movies.txt` se trouvant dans le répertoire `exo1`.

   Rappel la fonction `String.split_on_char : char -> string -> string list` permet de découper une chaîne en une liste de champs selon un caractère de séparation. La fonction `int_of_string` permet de convertir une chaîne en entier.

2. Compléter la fonction `top_k : int -> (t -> bool) -> t list -> t list` telle que `top_k k pred movies` renvoie la liste des au plus `k` premières valeurs de `movies` pour lesquelles `pred` renvoie vrai (si la liste a moins d'éléments que `k` vérifiant la propriété, ils sont tous renvoyés).

3. Compléter la fonction `group_by : (t -> 'a) -> t list -> ('a * int) list` telle que `group_by f movies` renvoie la liste des paires `(k, nk)` où `nk` est le nombre de tous les films pour lesquels `f` renvoie `k`. Par exemple :

   ```
   group_by (fun m -> m.year) movies
   ```

   renvoie pour chaque année de film la liste des films sortis cette année là. On aura donc un résultat de la forme

   ```
   [ (2005, 7) (* si 7 films en 2005 *);
     (2000, 3) (* si 3 films en 2000 *);
     (2010, 8) (* si 8 films en 2010 *); ... ]
   ```

   Indications il y a deux méthodes possibles
   - On peut former la liste des `(f m, m)` pour chaque film, la trier selon la première composante en utilisant `compare` comme comparaison, puis la parcourir la liste triée et compter les occurrences successives.
   - On peut sinon créer une table de hachage générique (`Hashtbl.t`) et utilisant les `f m` comme clé et un compteur comme valeur, puis reconvertir cette table en liste (par exemple avec `Hashtbl.fold`).

Exercice 2 (6 points)

On se place dans le cadre des arbres binaires de recherche de type AVL. On modifie le type pour permettre de représenter des *multi-ensembles* c'est à dire qu'en plus de la valeur, on stocke dans chaque nœud le nombre d'occurrences. Le type ainsi que le code de ré-équilibrage est stocké dans le fichier `abr.ml`. **Ce dernier n'est pas à lire ni à modifier**. Le fichier `abr.mli` exporte uniquement la définition du type `tree` et une fonction de construction `merge`.

```
type 'a tree = Leaf | Node of (int * 'a tree * 'a * int * 'a tree)

val merge : 'a tree -> 'a tree -> 'a tree
(** [merge t1 t2] prend en argument deux arbres tels que
    toutes les valeurs de [t1] sont strictement inférieures aux
    valeurs de [t2] et renvoie l'arbre équilibrés contenant les
    éléments de [t1] et de [t2].
*)
```

Dans le type `'a tree`, pour le constructeur `Node`, le premier entier est la hauteur de l'arbre, suivi de l'arbre gauche, suivi de l'élément racine, suivi du nombre d'exemplaires de cet élément (un entier strictement positif), suivi de l'arbre droit.

La fonction `merge` prend en argument (dans cet ordre) l'arbre gauche et l'arbre droit (dont les valeurs sont supposées strictement plus grandes que celles de l'arbre gauche) et renvoie le nouvel arbre contenant les valeurs des deux arbres arguments Elle fait en sorte que le résultat soit un arbre équilibré.

Le fichier `exo2.ml` définit un foncteur `Make(E : Comparable)` comme vu en cours. Il utilise le module `Abr`. Le but de cet exercice est d'implémenter la fonction `add : E.t -> int -> t -> t` qui ajoute `n` copies d'un élément à un arbre. Du code de test est fourni en fin de fichier, en dehors du foncteur.

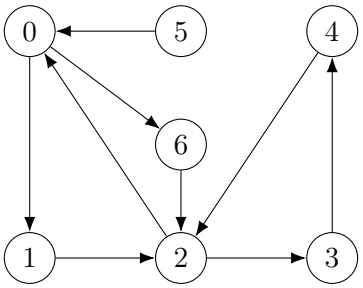
1. Compléter la fonction `singleton : E.t -> int -> t` qui telle que `singleton v n` renvoie le multi-ensemble contenant `n` copies de l'élément `v`. Si `n` est négatif ou nul, votre fonction renvoie l'arbre vide.
2. Compléter la fonction `join : t -> E.t -> int -> t -> t` telle que `join l v n r` prend en argument :
   — un arbre gauche `l`
   — une valeur `v` supposées strictement supérieure à toutes les valeurs de `l`
   — un entier `n` supposé strictement positif
   — un arbre droit `r` dont les valeurs sont strictement supérieures à `v`

   Les tailles de `l` et `r` sont quelconque. La fonction renvoie l'arbre équilibré construit à partir de ses arguments.

   Remarque : cette fonction est très simple et non récursive, elle fait juste un usage judicieux de `merge` et `singleton`.
3. Compléter la fonction `add : E.t -> int -> t -> t` telle que `add v n t` renvoie l'arbre `t` dans lequel `n` copies de `v` ont été ajoutées. On autorise `n` à **être négatif**. Par exemple, `add v (-4) t` retire `4` exemplaires de l'élément `v` de `t`.
   — s'il y avait `10` éléments `v`, il en reste `6` dans l'arbre résultat
   — s'il y avait `4` éléments `v` ou moins dans `t`, alors l'élément `v` n'apparaît plus dans l'arbre résultat.

Exercice 3 (7 points)

On s'intéresse à la résolution du problème de trouver les *chemins eulériens* d'un graphe. Un chemin est dit eulérien s'il passe par toutes les arrêtes du graphe. On donne ci-dessous un exemple de graphe, sa représentation en OCaml par une table de hachage et l'un de ses chemin eulériens.




```
(* le graphe de test *)
let graph = Hashtbl.create 16
let () =
  Hashtbl.add graph 0 [1;6];
  Hashtbl.add graph 1 [2];
  Hashtbl.add graph 2 [0;3];
  Hashtbl.add graph 3 [4];
  Hashtbl.add graph 4 [2];
  Hashtbl.add graph 5 [0];
  Hashtbl.add graph 6 [2]
```

```
[(5, 0); (0, 1); (1, 2); (2, 0); (0, 6); (6, 2); (2, 3); (3, 4); (4, 2)]
```

On commence par suivre l'arrête $(5, 0)$, puis $(0, 1)$, ..., et on termine par $(4, 2)$. Ce chemin contient bien les 9 arrêtes du graphe (par contre on remarque qu'on est passé plusieurs fois par les mêmes sommets, ce qui est autorisé). Le graphe est représenté par une table de hachage OCaml, dont les clés sont les sommets et les valeurs la liste des voisins. Les sommets sont les entiers consécutifs de 0 à $n-1$.

1. Compléter la fonction `get_edges : (int, int list) Hashtbl.t -> int -> (int*int) list` telle que `get_edges graph v` renvoie la liste des arrêtes dont l'origine est `v`. Pour le graphe d'exemple, `get_edges graph 2` renvoie `[(2,0); (2,3)]`.
2. Compléter la fonction `count_edges : graph -> int` qui renvoie le nombre total d'arrêtes dans le graphe. Pour le graphe d'exemple la fonction doit renvoyer `9`.
3. Compléter la fonction

   `eulerian : ((int*int) list -> unit) -> (int, int list) Hashtbl.t -> unit`

   telle que `eulerian f graph` appelle la fonction `f` sur tous les chemins eulériens du graphe `graph`. Le code est partiellement donné. La fonction interne `loop` prend en argument un sommet `v` et maintient dans `path` la liste des arrêtes visitées et dans `n` la taille de cette liste. Le cas de base (`TODO 1`) est atteint lorsque la taille de la liste est égale au nombre d'arrêtes du graphe. Dans le cas récursif (`TODO 2`), pour chaque arrête issue de `v` qui n'a pas déjà été visitée, on l'ajoute au chemin et on s'appelle récursivement sur le voisin accessible par cette arrête. Enfin, la fonction `loop` doit être appelée pour tous les sommets (`TODO 3`). Le code **en fin de fichier** appelle la fonction `eulerian` et affiche la liste des chemins trouvés pour le graphe d'exemple.