université PARIS-SACLAY | FACULTÉ DES SCIENCES D'ORSAY

PFA

L3 Info, L3 Mag Info

TP 3

Présentation

Le but du TP est de se familiariser avec les arbres binaires de recherche et en particulier les AVLs tels qu'ils ont été présentés en cours. Le TP est basé sur une archive à télécharger sur la page du cours. Dans cette archive on trouve les fichiers :

`tree.ml` : ce dernier contient la définition du type des arbres binaires de recherche et toutes les fonctions associées. **Il faut compléter ce fichier**.

`main.ml` : ce fichier contient le programme principal, c'est dans celui-ci qu'il faudra écrire vos tests. **Il faut compléter ce fichier**.

`draw.ml` : ce fichier contient une fonction permettant de représenter visuellement un arbre binaire de recherche sous forme d'une image au format SVG. Son utilisation sera décrite dans l'énoncé. **Il ne faut pas modifier ce fichier**.

`dune-project, dune, .ocamlformat` : fichiers de configuration du projet. **Il ne faut pas modifier ces fichiers**.

L'exécutable étant constitué de plusieurs fichiers sources, on le construira avec la commande :

```
$ dune build
```

Cette dernière re-compile tous les fichiers modifiés depuis la dernière compilation et crée un exécutable `main.exe` dans le répertoire courant. Tous les fichiers temporaires créés par la compilation se trouvent dans un répertoire `_build`. Il est possible de nettoyer le répertoire avec la commande :

```
$ dune clean
```

On ne présente pas plus en détail ces commandes qui feront l'objet d'un cours plus général sur la programmation modulaire en OCaml.

> **Attention** on suppose dans la suite que le TP est fait sous Linux, de préférence sur les machines du PUIO. Les étudiants qui ne sont pas dans cette situation sont invités à lire les instructions en fin d'énoncé pour pouvoir répliquer le TP sur leur machine.

1 Arbres AVL

Comme expliqué en cours, le code des arbres AVLs est séparés en deux parties. Une partie générique sur les arbres binaires de recherches (début du fichier `tree.ml`) c'est à dire sur le type :

```
1 type ('a, 'info) tree =
2   | Leaf
3   | Node of ('info * ('a, 'info) tree * 'a * ('a, 'info) tree)
```

On rappelle que la variable de type `'a` représente le type des éléments de l'arbre et que la variable de type `'info` représente le type d'information à stocker dans les nœuds pour rééquilibrer l'arbre. Dans le cadre des AVLs, manipuler cette information est simplement la hauteur de l'arbre. À la suite de ce type (lignes 6-106) sont définies les opérations de construction générique (`add_gen`, `union_gen`, ...). Ces fonctions sont celles présentées en cours. Elles prennent comme argument optionnel la fonction `join` qui permet de construire des arbres équilibrés. Dans la seconde partie du fichier (ligne 110 et suivantes) on définit le type des AVLs ainsi

qu'une fonction `join_avl` permettant de faire le rééquilibrage. Les fonctions `add`, `remove`, ...sont spécialisées avec `join_avl`. On étudie cette fonction de ré-équilibrage dans la seconde partie du TP.

1.1 Opérations de bases

1. Prendre le temps de lire le fichier `tree.ml`. La plupart des fonctions seront à rajouter à la fin, sauf les fonctions de rééquilibrage qui seront à ajouter ligne 144 et suivantes.
2. Ouvrir le fichier `main.ml` et le lire. Ce dernier contient du code de test ainsi que la définition d'une fonction `dump : string -> ('a, 'info) tree -> unit`. Un appel `dump f t` écrit dans le fichier dont le nom est donné par `f` une représentation graphique de l'arbre binaire de recherche `t` (ce code utilise une fonction auxiliaire écrite pour vous dans le module `Draw`). La fonction `test1` montre un exemple d'utilisation de ces fonctions. Compiler le programme (`dune build`) et l'exécuter (`./main.exe`). Constater que le répertoire `img/` contient maintenant les deux fichiers `.svg` créés par la fonction `test1`. Vous pouvez visualiser ces fichiers au moyen de la commande :

   ```
   eog img/test1_*.svg
   ```

3. Écrire un autre test `test2` permettant de tester les fonctions de constructions (au choix union, intersection , différence, ...). Remplacer l'appel à `test1` par `test2` dans le fichier `main.ml` puis relancer le programme.
4. Écrire les fonctions suivantes à la fin du fichier `tree.ml` :
   — `is_empty : ('a, 'info) tree -> bool` qui renvoie vrai si l'arbre est vide
   — `mem : 'a -> ('a, 'info) tree -> bool` qui renvoie vrai si l'élément rechercher est dans l'arbre
   — `iter : ('a -> unit) -> ('a, 'info) tree -> unit` qui applique la fonction passée en argument à tous les éléments de l'arbre par ordre croissant.
   — `fold : ('a -> 'b -> 'a) -> ('a, 'info) tree -> 'b -> 'a` telle que `fold f t acc` soit équivalent à `(f vn ...(f v2 (f v1 acc)))` où les `vi` sont les éléments de l'arbre `t` par ordre croissant.

   Pour toutes les comparaisons on utilisera la fonction `compare : 'a -> 'a -> int` de la bibliothèque standard d'OCaml, comme vu en cours.

   Remarque ces trois fonctions ont été données en cours, il est important de savoir les réécrire sans regarder les supports. Écrire dans `main.ml` des tests pour ces fonctions. Attention, il n'est pas obligatoire de faire des tests graphiques, les fonctions ne construisant pas d'arbre peuvent être testées en affichant les résultats des tests dans le terminal.
5. Indiquer comment exprimer `iter` en fonction de `fold`
6. Écrire une fonction
   `partition : ('a -> bool) -> ('a, 'info) tree -> (('a, 'info) tree * ('a, 'info) tree)`
   qui prend en argument un prédicat et un arbre et renvoie l'arbre des valeurs pour lesquelles le prédicat est vrai et et celui pour lesquelles le prédicat est faux.
7. Proposer une façon d'implémenter `subset : ('a, 'info) tree -> ('a, 'info) tree -> bool` telle que `subset t1 t2` renvoie vrai si `t1` est un sous-ensemble de `t2`.
8. Écrire une fonction `elements: ('a, 'info) tree -> 'a list` qui renvoie la liste des valeurs de l'arbre par ordre croissant. On écrira directement cette fonction (sans utiliser `fold`) pour construire directement la liste dans le bon ordre.

1.2 Équilibrage

On se concentre maintenant sur la partie la plus complexe, le code de `join_avl`. Comme vu en cours, le code de `join_avl` utilise deux fonctions auxiliaires :

```
let rec join_avl_right l v r =
  match l with
  | Leaf -> failwith "impossible"
  | Node (_, ll, vl, rl) ->
      if height rl <= height r + 1 then
        let new_r = node rl v r in
        if height new_r <= height ll + 1 then node ll vl new_r
        else rotate_left (node ll vl (rotate_right new_r))
      else
        let new_r = join_avl_right rl v r in
        let new_t = node ll vl new_r in
        if height new_r <= height ll + 1 then new_t else rotate_left new_t


let join_avl_left l v r = (* symétrique *)

let join_avl l v r =
  if height l > height r + 1 then join_avl_right l v r
  else if height r > height l + 1 then join_avl_left l v r
  else node l v r
```

La fonction `join_avl_right l v r` construit l'AVL équilibré en faisant l'hypothèse que :
- l'arbre `l` a une hauteur supérieure à celle de `r` de 2 ou plus
- les valeurs de l'arbre `l` sont inférieures à `v`
- les valeurs de l'arbre `r` sont supérieures à `v`
- les arbres `l` et `r` sont équilibrés

On souhaite constater empiriquement que la fonction `join_avl` fonctionne bien pour fusionner deux arbres dont les hauteurs diffèrent d'au plus 2, et donne des arbres déséquilibrés sinon.

9. Écrire (dans `main.ml`) une fonction
   `tree_of_height: int -> int -> bool -> (int, 'a) tree`
   telle que `tree_of_height h n b` renvoie un arbre de hauteur `h`, contenant des valeurs supérieures (resp. inférieures) à `n` si `b` vaut vrai (resp. si `b` vaut false).
10. Se servir de la fonction pour créer deux arbres de hauteurs très différentes (par exemple 4 et 8) les fusionner avec join (et une valeur intermédiaire) et constater que le résultat est un arbre équilibré, c'est à dire que les tailles de tous les chemins diffèrent d'au plus 1.
11. Proposer une façon de construire deux arbres équilibrés contenant les même valeurs mais des structures différentes.
12. On souhaite maintenant écrire la fonction `same_fringe t1 t2` prenant deux ABR en argument et répondant vrai si et seulement s'ils contiennent les même valeurs. De plus, on souhaite que cette fonction s'arrête le plus tôt possible. En particulier on ne souhaite *pas* convertir les deux arbres en listes triées puis comparer les listes (car si les listes diffèrent par leur plus petit élément, on a créé les deux listes pour rien).

    On propose l'algorithme suivant. Écrire une fonction
    `next : ('a, 'info) tree list -> ('a * ('a, 'info) tree) option`.

    Cette prend en argument une liste d'arbres renvoie le plus petit élément dans cette liste et le retire. On pourra procéder comme ceci pour une liste s :
    - si s est vide, renvoyer `None`
    - sinon s est une liste $t::ss$.
      - si t est vide se rappeler sur ss
      - sinon si t est une feuille de valeur v, renvoyer `Some(`v`, `ss`)`
      - sinon si t = (l, v, r), se rappeler sur l`::(Leaf,`v`,Leaf)::`r`::`ss

    Utiliser ensuite la fonction `next` pour parcourir les deux arbres en parallèle par valeur croissantes et s'arrêter dès que deux valeurs diffèrent.

2 Travail en dehors du PUIO

Il faut que les paquets `gg` et `vg` soient installés :

```
opam install gg vg
```

Il faut ensuite utiliser le visualiseur d'image de votre système à la place de la commande `eog`. Le cas échéant, les images de types `.svg` peuvent être ouvertes avec un navigateur Web tel que Firefox ou Chrome.