Révisions : dictionnaires, arbres binaires de recherche et équilibrage

1 Introduction

Dans ce sujet, on souhaite créer un dictionnaire, implémenté par un arbre binaire de recherche. Pour ce faire, on définit le type suivant :

```
type ('a, 'b) dict =
  Node of ('a, 'b) dict  (* fils gauche *)
        * 'a             (* clé *)
        * 'b             (* valeur *)
        * int            (* taille *)
        * ('a, 'b) dict  (* fils droit *)
  | Empty
```

On suppose qu'il est possible de trier les clés avec les opérateurs de comparaison usuels, et que l'arbre manipulé est un arbre binaire de recherche pour les clés (toutes les clés dans le sous-arbre gauche sont inférieures ou égales à la clé de la racine, toutes celles du sous-arbre droit sont supérieures ou égales à la clé de la racine). On impose par ailleurs les invariants suivants :

- toutes les clés sont distinctes ;
- le quatrième élément d'un objet `Node` contient toujours un entier représentant la taille du sous-arbre enraciné qu'il représente (le nombre de couples clé-valeurs).

Pour les tests vous trouverez à l'adresse `cdn.sci-phy.org/mp/dictionnaires.ml` un fichier contenant la déclaration du type et un dictionnaire contenant onze couples, dont les clés sont les chaînes de caractères contenant les mots « zéro », « un », ... « dix » et les valeurs sont les entiers correspondant auxdits mots.

2 Implémentation élémentaire

1. Proposer une fonction `taille` de signature `('a, 'b) dict -> int` prenant en argument un dictionnaire et renvoyant le nombre de couples clé-valeur qu'il contient. On attend une complexité constante.

2. Proposer une fonction `mini_cle` de signature `('a, 'b) dict -> 'a` retournant la plus petite clé. Quelle est sa complexité ? On testera la fonction sur le dictionnaire fourni.

3. On suppose (pour cette question uniquement) les valeurs du dictionnaires comparables avec l'opérateur de comparaison usuel. Proposer une fonction `mini_val` de signature `('a, 'b) dict -> 'b` retournant la plus petite valeur. Quelle est sa complexité ? On testera la fonction sur le dictionnaire fourni.

4. Proposer une fonction `membre` de signature `('a, 'b) dict -> 'a -> bool` prenant en argument une clé et renvoyant un booléen indiquant si la clé est présente dans le dictionnaire. On attend une complexité en la hauteur de l'arbre représentant le dictionnaire. Tester la fonction avec quelques clés.

5. Proposer une fonction `valeur` de signature `('a, 'b) dict -> 'a -> 'b` prenant en argument une clé et renvoyant la valeur associée si la clé est présente, et levant une erreur sinon. On attend une complexité en la hauteur de l'arbre représentant le dictionnaire. Tester la fonction avec quelques clés.

6. Proposer une fonction `ajoute` de signature `('a, 'b) dict -> 'a -> 'b -> ('a, 'b) dict` prenant un dictionnaire, une clé, une valeur et renvoyant un dictionnaire dans lequel le couple a été ajouté (si la clé n'était pas présente) ou la valeur a été mise à jour (si la clé était déjà présente). On attend une complexité en la hauteur de l'arbre, et on prendra garde à assurer les invariants attendus. Ajouter les couples clés-valeur pour « onze » et « douze », et vérifier que les fonctions `taille` et `valeur` donnent bien les résultats attenus.

7. Proposer une fonction `construit` de signature `('a * 'b) list -> ('a, 'b) dict` prenant en argument une liste de couples et construisant et renvoyant un dictionnaire représentant cette liste. Quelle est sa complexité dans le pire des cas ? En moyenne ? Tester la fonction avec une liste de couples de votre choix.

8. On suppose à présent que la liste contient uniquement des clés distinctes, et qu'elles sont rangées par ordre croissant. Proposer une fonction `construit_eq` qui effectue la même tâche que `construit`, mais garantit que la hauteur de l'arbre construit est inférieure ou égale à $\log(n)$ où n est la longueur de la liste. Quelle est sa complexité ? Tester la fonction ainsi écrite.

9. Proposer une fonction `vers_liste` de signature `('a, 'b) dict -> ('a * 'b) list` prenant en argument un arbre représentant un dictionnaire et retournant une liste de couples clé-valeur. On s'efforcera d'écrire une fonction de complexité linéaire en le nombre n de couples. Tester cette fonction avec le dictionnaire fourni et avec celui que vous avez construit vous-même.

3 Équilibrage

Pour garantir que les opérations sur le dictionnaire ont une complexité en $O(n \log n)$ où n représente le nombre de couples présents dans le dictionnaire, il nous faut nous assurer que les arbres binaires de recherche restent suffisamment « équilibrés ». Il existe différentes solutions pour y parvenir (arbres AVL, arbres rouge-noir...) La solution que nous allons mettre en œuvre aujourd'hui est celle des arbres « bouc-émissaires ».

Pour ce faire, il nous faut choisir un paramètre d'équilibrage $\alpha \in]0.5, 1[$. On prendra $\alpha = 2/3$.

Un nœud d'un arbre est dit α−équilibré en taille si la taille de ses sous-arbres gauche et droit sont tous deux strictement inférieurs à α fois la taille du sous-arbre enraciné en ce nœud.

10. Proposer une fonction `est_equil_t` de signature `('a, 'b) dict -> bool` prenant en argument arbre représentant un dictionnaire non-vide et retournant un booléen indiquant si sa racine est α−équilibrée en temps constant.

Un arbre est dit α−équilibré en taille s'il est α−équilibré pour tous ses nœuds. Un arbre bouc-émissaire n'est pas nécessairement toujours α−équilibré mais on garantit en revanche que sa hauteur ne sera jamais « trop grande » en s'assurant qu'elle est toujours inférieure où égale à

$$\lfloor \log_{1/\alpha} n \rfloor + 1$$

où n est la taille de l'arbre (le nombre de couples mémorisés dans le dictionnaire), ce qui suffit à garantir [1] que la recherche d'une clé est bien de complexité logarithmique en le nombre de couples mémorisés.

La recherche d'une clé, qui ne touche pas à l'arbre, n'est pas modifiée. En revanche, on va modifier l'opération consistant à ajouter une clé pour garantir que cette propriété reste vraie, de la façon suivante :

- L'insertion se fait au niveau des feuilles de l'arbre.
- Lors de l'insertion, on détermine la hauteur du nouvel arbre. Si après cette insertion la condition n'est plus vérifiée, alors on cherche, en remontant, un nœud, parmi les ancêtres du nouveau nœud, qui ne soit pas α−équilibré [2], et on remplace l'ensemble du sous-arbre enraciné en ce nœud par un nouveau sous-arbre bien équilibré, en extrayant une liste tous les couples clés-valeurs par ordre croissant et en reconstruisant un arbre équilibré à partir de cette liste (on utilisera les fonctions précédentes).

11. Proposer une nouvelle fonction `ajoute` qui implémente cette stratégie, et vérifier son bon fonctionnement avec quelques tests.

On peut aisément montrer que, après cette opération, la hauteur de l'arbre respecte à nouveau la condition (la hauteur de l'arbre a nécessairement été réduit) et que, si la complexité de `ajoute` est linéaire en la taille de l'arbre dans le pire des cas, elle est logarithmique *en moyenne*.

1. Le but de cette séance de TP est de s'entraîner à la programmation OCaml, donc on l'admettra, mais vous êtes invités à réfléchir à cette preuve.
2. On peut montrer qu'il existe, ce n'est pas le but de cette séance de TP, mais c'est un bon entraînement!