TD Info 3 : Arbres et recherche

Michael Monerau

4 novembre 2010

1 Arbres binaires

On rappelle qu'un arbre binaire est défini comme étant soit une feuille, soit une paire d'arbres binaires.

Question 1.1. Définition
Définir un type Caml pour représenter un arbre binaire, dont l'étiquette des nœuds internes et des feuilles est de type générique `'a`.

Dans la suite, on pourra supposer que les étiquettes sont des entiers positifs ou nuls, et donc représenter l'arbre vide par une feuille d'étiquette -1.

Question 1.2. Expression arithmétique
On peut ainsi décrire une expression mathématique avec des opérateurs binaires. Comment utiliser cette structure de données pour représenter par exemple l'expression : $5 * 3 + 2 * 10/2$?

Écrire une fonction `eval_expr` qui prend en entrée un arbre d'expression comme vous l'aurez défini, et qui renvoie le résultat de l'expression (on supposera que les expressions s'écrivent ici simplement avec les opérateurs binaires `+`, `-`, `*`, `/` et les constantes entières).

Remarque : La première phase d'une compilation ou d'une analyse de code est toujours le passage à un tel *arbre de syntaxe abstraite* (analyse syntaxique). Ainsi, on a une représentation plus pratique du code puisqu'il est maintenant séparé en opération. Selon la sémantique des opérations, on peut alors ensuite interpréter le code à partir de cet arbre (analyse sémantique).

Le plus souvent, l'arbre est généré automatiquement à l'aide d'un "parser", généré lui-même automatiquement à partir d'une grammaire du langage analysé.

2 Arbres binaires de recherche

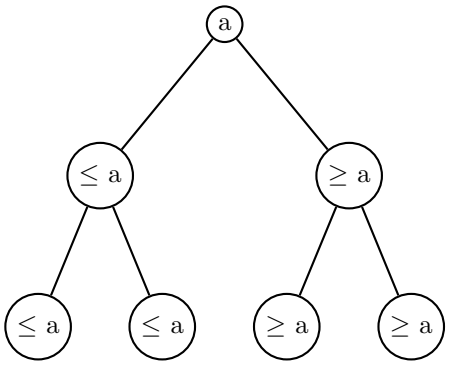
Les arbres binaires de recherche constituent un moyen de retrouver rapidement des éléments parmi un grand ensemble. En général, l'élément à trouver est un pointeur vers une structure de données associée à la clé recherchée (dictionnaire associatif).

Nous allons voir comment les manipuler, leurs avantages et inconvénients et comment améliorer leurs performances.

2.1 Définition

On suppose que les étiquettes qu'on utilise sont ordonnées. Ici, nous utiliserons des entiers comme étiquettes et utiliserons donc l'ordre habituel $\leq$.

Un BST (*Binary Search Tree*) est un arbre binaire avec une seule contrainte : pour tout nœud interne, les étiquettes de tous les éléments de son sous-arbre gauche lui sont inférieures ou égales, et supérieures ou égales pour son sous-arbre droit. Graphiquement :



Question 2.1. Check
Écrire une fonction `check_BST : 'a arbre -> bool` qui dit si l'arbre passé en argument est un BST.

Quelle est la complexité de cette fonction ?

Question 2.2. Recherche
Écrire une fonction `search : 'a -> 'a arbre -> bool` qui renvoie `true` si l'élément est trouvé dans l'arbre passé en argument, et `false` sinon.

Quelle est la complexité de cette fonction ?

Question 2.3. **Balance**

On définit la *balance* d'un nœud interne comme la différence entre la hauteur de son sous-arbre droit et gauche (et 0 pour une feuille).

Écrire une fonction `balance : 'a arbre -> int` qui renvoie la balance de la racine de l'arbre passé en argument.

Pour avoir une exécution de `search` la plus rapide possible en moyenne, préfère-t-on une balance faible ou importante ?

2.2 Opérations basiques

Avant d'explorer comment améliorer la balance d'un BST, voyons comment exécuter les opérations de base sur cette nouvelle structure de données.

Question 2.4. **Insertion**

Écrire une fonction `insert : 'a -> 'a arbre -> 'a arbre` qui insère un élément particulier dans un BST.

Quelle est la complexité ?

Question 2.5. **Création**

En déduire une fonction `create_BST : 'a list -> 'a arbre` qui transforme une liste en BST.

Quelle est la complexité ?

Question 2.6. **Tri**

Comment obtenir les éléments d'un BST dans l'ordre croissant ? Utiliser simplement `print_int` pour visualiser le résultat.

En déduire subtilement une méthode de tri de liste. Quelle en est sa complexité ?

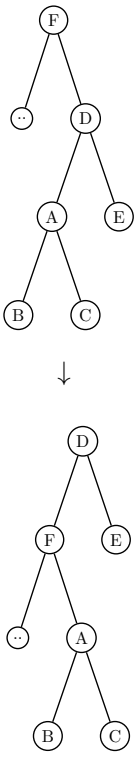
Question 2.7. **Sélection**

Comment obtenir le k-ième élément d'un BST ? En déduire toujours subtilement une méthode pour avoir le k-ième élément d'une liste. Quelle en est sa complexité ?

2.3 Rotation

Comme on l'a vu plus haut, il est plus intéressant d'avoir des BST qui ont une *faible* balance. Pour cela, un ingrédient important est la *rotation*.

Le plus simple pour l'expliquer est un exemple. La rotation à gauche de l'arbre suivant :



C'est-à-dire que la racine est devenue le fils gauche de son fils droit.

Inversement pour la rotation à droite.

Question 2.8. **Correct**

Vérifier que le procédé transforme bien un BST en un BST.

Question 2.9. **Rotation**

Écrire deux fonctions `rotR : 'a arbre -> 'a arbre` et `rotL : 'a arbre -> 'a arbre` qui effectuent les deux opérations de rotation. Complexité ?

Question 2.10. **Insertion à la racine**

En déduire une méthode d'insertion à la racine dans un BST. Quelle est la complexité ?

Question 2.11. **Partitionnement**

Écrire une fonction `partition : 'a arbre -> int -> 'a arbre` qui prend en entrée un BST et un entier k et qui retourne un BST avec les même éléments mais où le k-ième élément est en racine.

Question 2.12. **Join**

Écrire une fonction `join : 'a arbre -> 'a arbre -> 'a arbre` qui forme un BST contenant l'union des deux BST passés en arguments. Complexité ?

Question 2.13. **Suppression**

Écrire une fonction `delete : 'a -> 'a arbre -> 'a arbre` qui supprime un élément d'un BST. S'il est présent à plusieurs endroits, on se contente de supprimer la première occurence rencontrée.

Quelle est la complexité ?

2.4 Vers l'équilibre

Maintenant voyons quelques techniques pour essayer d'améliorer les performances des BST, selon plusieurs critères.

2.4.1 Équilibre parfait

Question 2.14. **Équilibre**

En utilisant la fonction de partitionnement, écrire une fonction `equilibre : 'a arbre -> 'a arbre` qui équilibre parfaitement un BST.

Est-ce que cette fonction serait utile pour créer des BST équilibrés à partir de tableaux ?

2.4.2 BST avec aléas

Jusque là, on insère un élément soit comme feuille (première méthode), soit comme racine (deuxième méthode). Si on crée un tableau avec chacune de ces méthodes indépendamment, le BST final n'a pas de raison d'être équilibré (et en a plutôt de ne pas l'être).

Pour remédier à cela, on alterne aléatoirement entre insertion à la racine et en feuille. On part de l'observation que lorsqu'on a un tableau de n éléments, chacun des éléments a une probabilité $1/n$ d'être racine.

Par conséquent, on choisit pour cet algorithme d'insertion que quand on insère un élément dans un BST contenant déjà n éléments, on l'insère en racine avec probabilité $\frac{1}{n+1}$, et sinon on l'insère récursivement (avec aléas) dans le bon sous-arbre (et donc la probabilité d'être racine du sous-arbre est $\frac{1}{n'+1}$ mais où $n' < n$).

Question 2.15. **BST aléatoire**

Implémenter cet algorithme en une fonction `RandBST : 'a vect -> 'a arbre`.

Indication 1 : vous pouvez conserver un champ `taille` dans vos sommets qui indique la taille du sous-arbre dont il est racine, mais sinon recalculez à chaque fois la valeur, ce n'est pas bien grave pour notre cas.

Indication 2 : utiliser la fonction `random_int : int -> int` (attention, 2 fois le caractère '_') pour avoir un tirage au sort aléatoire (`random_int i` renvoie un nombre aléatoire entre 0 et i inclus).

Ensuite, il y a également les autres algorithmes qui suivent, ie. suppression aléatoire, join aléatoire, mais nous n'allons pas les traiter ici.

Question 2.16. **Tests**

Générer des BST aléatoirement et regarder leur balance. Comparer avec les BST non aléatoires générés sinon.

2.4.3 Splay BST

Chez les Splay BST, le principe est de faire 2 rotations lors de l'insertion en racine (et de la recherche, le long du chemin de recherche). Après analyse, on voit que cela a pour effet de diviser par deux la distance des nœuds croisés à la racine. Ainsi, si on recherche un de ces éléments par la suite, il seront accessibles plus rapidement.

C'est donc une structure adaptative : on essaie d'amortir le coup. Les éléments cherchés souvent sont rapidement accessibles, mais ceux qui sont rarement utilisés sont plus longs à trouver.

Les détails techniques ne sont pas essentiels, on ne traitera pas les détails de l'algorithme.

On peut cependant remarquer qu'on a des garanties globales sur de tels arbres : la création d'un Splay BST de n éléments est garanti en $O(n \ln n)$ même si localement, on n'a pas de garantie de ne pas être linéaire lors d'une insertion (cas typique d'une complexité amortie).

2.4.4 Arbres 2-3-4

Les arbres 2-3-4 ne sont plus des BST car les nœuds peuvent avoir deux, trois ou quatre fils. Mais ils ont alors autant d'éléments sur leur étiquette (moins 1), et les liens vers les fils correspondent (pour un noeud (a, b) avec 3 fils, le premier fils doit vérifier $\leq a$, le deuxième $\in [a, b]$, le dernier $\geq b$).

C'est donc une généralisation des BST.

On dit qu'un tel arbre est équilibré lorsque toutes les feuilles sont à même distance de la racine (± 1).

Il y a un algorithme "simple" et efficace ($O(\ln n)$) pour conserver l'arbre parfaitement équilibré à l'insertion (voir explications au tableau). On ne peut donc pas espérer mieux.

Malheureusement, l'implémentation directe des arbres 2-3-4 est fastidieuse et pas si efficace que ça.

Cependant, on peut reformuler les arbres 2-3-4 en simples BST si on les abstrait comme il faut en des *arbres rouge-noir* (voir explications au tableau).

Il suffit alors de retranscrire les algorithmes faciles à comprendre des arbres 2-3-4 dans ce nouveau cadre des arbres rouge-noir. On obtient alors une structure de données très efficace : on y insère en $O(\ln n)$ et on recherche en $O(\ln n)$ également.