Notes de cours Option Info MP/MP* : Arbres binaires

Définition

Un *arbre* est un graphe connexe acyclique enraciné : tous les sommets ont un unique parent (sauf la racine).

Si une arête mène du sommet i au sommet j, on dit que i est le *père* de j et que j est le *fils* de i. En général on représente l'arbre de sorte que le père soit toujours au-dessus de ses fils, ce qui fait que dans ce cas ce n'est pas la peine d'orienter les arêtes. On parlera alors plutôt de *nœud* que de sommet. Un nœud qui n'a pas de fils est appelé *feuille* (ou *nœud externe*), et *nœud interne* sinon.

Notons que chaque nœud est la racine d'un arbre constitué de lui-même et de ses descendants, qu'on appelle *sous-arbre* de l'arbre initial.

On appelle *arité* d'un nœud le nombre de ses fils. Un *arbre binaire* est un arbre tels que tous les nœuds ont une arité inférieure ou égale à deux, c'est-à-dire ont au plus deux fils. Parmi eux, on distingue un arbre binaire qui est l'*arbre vide*, qui ne possède aucun nœud. Cela nous permet de définir formellement par induction les arbres binaires de la manière suivante:

Un ensemble E étant donné, l'ensemble des arbres binaires étiquetés par E est défini comme suit:

-`nil` est un arbre binaire appelé arbre vide

-Si $x \in E$ et F_g et F_d sont deux arbres binaires, alors $A = (F_g, x, F_d)$ est un arbre binaire.

x est l'étiquette de la racine de A, et F_g et F_d sont appelés respectivement *sous-arbre gauche* et *sous-arbre droit* de A.

En général, on ne fait pas figurer l'arbre vide dans les représentations graphiques.

Donner des applications des arbres.

Définition du type `'a arbre` en Caml:

La définition inductive peut se résumer à:

$$\text{Arbre} = \text{Nil} + \text{Arbre} \times \text{Noeud} \times \text{Arbre}$$

ce qui permet d'adopter la définition suivante en Caml:

```
type 'a arbre = Nil | Noeud of ('a arbre * 'a * 'a arbre) ;;
```

Fonctions inductives

La plupart des résultats que nous allons voir se prouvent par induction structurelle, c'est-à-dire suivant le principe suivant (similaire à la récurrence pour les entiers): soit $\mathcal{R}$ une propriété sur les arbres binaires. On suppose que:

-$\mathcal{R}$(`nil`) est vraie

-Pour tout $x \in E$ et tous arbres binaires F_g et F_d, si $\mathcal{R}(F_g)$ et $\mathcal{R}(F_d)$ sont vraies, alors $\mathcal{R}((F_g, x, F_d))$ est vraie.

De même, on peut définir une fonction f sur l'ensemble des arbres binaires en définissant $f(\texttt{nil})$ puis $f((F_g, x, F_d))$ connaissant $f(F_g)$ et $f(F_d)$.

Exemples: On définit par induction structurelle la taille $|A|$ d'un arbre binaire A comme suit (à compléter):
-$|\texttt{nil}| = ...$
-Si $A = (F_g, x, F_d)$, $|A| = ...$
De même on définit la hauteur $h(A)$ par:
-$h(\texttt{nil}) = ...$
-Si $A = (F_g, x, F_d)$, $h(A) = ...$

Dit autrement, $|A|$ est le nombre de nœuds de A et $h(A)$ est la distance maximale d'un nœud à la racine.

Exercice: Écrire en Caml des fonctions calculant la taille, la hauteur, le nombre de feuilles et le nombre de nœuds internes d'un arbre binaire.

Théorème. Soit A un arbre binaire, alors:

$$h(A) + 1 \leqslant |A| \leqslant 2^{h(A)+1} - 1.$$

Preuve. Par induction structurelle:
-$h(\texttt{nil}) = -1$ et $|\texttt{nil}| = 0$, on a bien les deux inégalités.
-Si le résultat est vrai pour F_g et F_d, et si $A = (F_g, x, F_d)$, on a:

$$|A| = 1 + |F_g| + |F_d| \geqslant 1 + h(F_g) + 1 + h(F_d) + 1 \geqslant 1 + (1 + \max(h(F_g), h(F_d))) = 1 + h(A),$$

$$|A| = 1 + |F_g| + |F_d| \leqslant 1 + 2^{h(F_g)+1} - 1 + 2^{h(F_d)+1} - 1 \leqslant 2 \times 2^{\max(h(F_g),h(F_d))+1} - 1 = 2^{h(A)+1} - 1.$$

Remarque: Le résultat précédent se réécrit : $\log_2(|A| + 1) \leqslant h(A) \leqslant |A| - 1$. Les deux cas extrémaux correspondent respectivement aux arbres complets et aux peignes (faire les dessins). On a tout intérêt (notamment dans le cas d'un arbre binaire de recherche) à ce que la hauteur soit plutôt logarithmique en $|A|$ plutôt que linéaire. On parle dans le premier cas d'arbre équilibré.

Arbres binaires de recherche

Les arbres binaires de recherche (ou ABR) sont particulièrement adaptés à la recherche d'un élément, et permettent donc d'implémenter des structures de données telles que les dictionnaires. Ils sont en général étiquetés par des couples (clé, valeur) où la clé est entier, cependant pour plus de clarté nous les représenterons uniquement par leurs clés, ils seront donc étiquetés par des entiers (deux à deux distincts).

Définition. Un arbre binaire de recherche est soit vide soit de la forme (F_g, x, F_d) où:
-F_g et F_d sont des ABR.
-Tout nœud de F_g est inférieur à x.
-Tout nœud de F_d est supérieur à x.

Parcours

Il existe plusieurs méthodes de parcours, voici les trois principales:

- Parcours infixe, dans l'ordre: Fils gauche-étiquette-Fils droit
- Parcours préfixe, dans l'ordre: étiquette-Fils gauche-Fils droit

- Parcours postfixe, dans l'ordre: Fils gauche-Fils droit-étiquette

Complexité: (compléter) Quel que soit le parcours, tous les nœuds sont visités une et une seule et le coût est ..... en le nombre de nœuds, seul l'ordre de visite change.

Dans le cas des ABR, le parcours infixe est particulièrement intéressant:

Proposition. Lors du parcours infixe d'un ABR, les nœuds sont visités dans l'ordre .....

Preuve (exercice): Par induction structurelle.

Exercice : écrire les fonctions associées (le parcours infixe sera vu en TP)

Recherche d'un élément

L'opération la plus courante dans un ABR est bien sûr la recherche d'une valeur associée à une clé, la structure d'ABR permettant d'améliorer le coût par rapport à la recherche naïve qui parcourt l'intégralité de l'arbre.

L'algorithme est le suivant, si $A = (F_g, x, F_d)$ et que l'on recherche l'élément c:
-Si $c = x$, renvoyer la valeur associée à x.
-Si $c < x$, rechercher c dans .....
-Si $c > x$, rechercher c dans .....

À faire en TP : écrire la fonction associée

Complexité: Elle est ..... Dans le cas d'un arbre équilibré, cela donne un coût .....

Recherche de la clé minimale/maximale

On peut également chercher la clé minimale ou maximale dans un ABR. Pour le minimum, la recherche se fait en suivant le fils .....tant qu'il n'est pas vide, pour le maximum, on suit le fils ..... Le coût est .....

Exercice : écrire la fonction associée

Recherche du prédécesseur/successeur

Enfin, étant donnée une clé c, on peut se demander quel est son prédécesseur, c'est-à-dire la plus grande clé strictement inférieure à c dans l'arbre, ou son successeur (plus petite clé strictement supérieure). Pour le prédécesseur, l'idée est la suivante : tant que la racine est $\geqslant c$ on cherche dans le fils ....., dès qu'on trouve un élément $x < c$ on a un prédécesseur potentiel, mais il faut chercher dans son fils .... qu'il n'y a pas un autre élément intermédiaire. Il faut pour cela introduire une exception, c'est-à-dire un message d'erreur renvoyé quand une condition est atteinte (ici : on a atteint la fin d'une branche sans avoir trouvé un élément intermédiaire) , la différence avec `failwith` étant qu'une exception peut être "rattrapée" (ici : si on ne trouve pas d'élément intermédiaire entre x et c, on renvoie x) alors que `failwith` stoppe l'exécution. Une exception se déclare comme suit:

`exception NotFound` (une exception commence toujours par une majuscule)

Pour lever l'exception au cours du programme: on utilise `raise (nom de l'exception)` .

Pour la rattraper, on commence par écrire `try` avant le programme, puis `with (nom de l'exception) -> .....`

Fonction à écrire ensemble.

Insertion dans un ABR

Il existe deux manière d'insérer un élément dans un ABR: aux feuilles ou à la racine. Pour insérer aux feuilles, on parcourt l'arbre comme dans l'algorithme de recherche jusqu'à tomber sur une feuille. *fonction à écrire en TP*

Pour insérer à la racine, on introduit une fonction auxiliaire qui partitionne l'arbre pour séparer les éléments inférieurs et supérieurs à la nouvelle racine. (voir exemple au tableau)

Complexité: .....

Suppression dans un ABR

La suppression d'un élément est plus compliquée. Si l'on veut supprimer c dans l'arbre $A = (F_g, x, F_d)$ on procède ainsi:

-Si $c < x$: on supprime c dans ....

-Si $c > x$: on supprime c dans ....

-Si $c = x$: si F_g ou F_d est vide on renvoie l'autre fils, sinon on doit chercher un élément minimal dans .... et le mettre à la racine à la place de x (on pourrait aussi chercher le maximum dans .....).

À faire en TP : écrire la fonction associée (écrire d'abord la fonction auxiliaire qui supprime le minimum d'un ABR et renvoie le couple (min, nouvel arbre))

Complexité: .....

Remarque. Les fonctions d'insertion et de suppression peuvent faire apparaître des déséquilibres dans l'arbre et faire que la hauteur n'est plus logarithmique en le nombre de sommets (exemple d'un peigne quand on insère les dans l'ordre croissant). Toutefois on peut montrer qu'il existe une constante c telle que si à partir de l'arbre vide on insère n nœuds de manière aléatoire (les $n!$ ordres possibles sont équiprobables), la hauteur moyenne des $n!$ ABR obtenus est équivalente à $c\log(n)$ (le cas moyen est plus proche du cas favorable). Il existe dans le cas des arbres équilibrés (arbres AVL, arbres rouge-noir...) des fonctions de rééquilibrage, cela est hors-programme mais peut tomber dans un sujet de concours.

Application 1 : tri avec un ABR

Expliquer comment trier une liste ou un tableau à l'aide d'un ABR. Donner les complexités dans le pire et meilleur des cas et comparer avec les autres tris connus. L'appliquer en TP.

Application 2 : Faire un dictionnaire avec un ABR

Rappeler la définition d'un dictionnaire vue en 1ère année et les opérations élémentaires dessus.

Rappeler les implémentations des dictionnaires vues en 1ère année et les complexités des opérations élémentaires associées.

La structure d'arbre binaire permet l'implémentation d'une structure ..... de dictionnaire (une fois que l'arbre est créé, on ne peut pas modifier un élément sans recréer un nouvel arbre).

Donner les complexités des fonctions de recherche/ajout/suppression d'une clé dans le pire et le meilleur des cas, et comparer avec les implémentations précédentes.

On peut aussi implémenter un dictionnaire avec une table de hachage (voir TP bonus). Dans ce cas le coût des algorithmes est en $O(n)$ dans le pire des cas mais en $O(1)$ en moyenne. De plus cette structure est impérative (on peut modifier les éléments après création).