Arbres de priorité équilibrés

Présentation des APE

Un arbre de priorité équilibré (APE) est un arbre binaire étiqueté par des éléments d'un ensemble E totalement ordonné et vérifiant les trois propriétés suivantes :

1. *Si* A *est un APE non vide et* e *est l'étiquette de la racine de* A*, alors* e *est la plus petite étiquette parmi celles figurant dans l'arbre* A*.*
2. *Si* A *est un APE non vide alors les branches gauche et droite de* A *sont des APE.*
3. *Si* A *est un APE non vide de branche gauche* G *et de branche droite* D *alors :*

$$\text{taille}(G) \leqslant \text{taille}(D) \leqslant \text{taille}(G) + 1.$$

La propriété *1* signifie qu'on peut trouver la plus petite étiquette de A en temps constant, c'est ce qui fait l'intérêt premier des arbres de priorité. Les propriétés *2* et *3* garantissent qu'un APE est « équilibré » dans le sens où les deux branches issues de chaque nœud ont même taille à une unité près, et en cas de différence, c'est la branche droite qui comporte un nœud de plus. On peut démontrer, par récurrence sur n, que la hauteur d'un APE à n nœuds est majorée par $\log_2(n)$, donc les opérations d'insertion et suppression sur un APE décrites ci-après ont une complexité $O(\ln n)$ lorsque l'arbre considéré est de taille n. Les APE sont utilisés en informatique pour stocker des tâches à accomplir, chaque tâche ayant un certain niveau de priorité (les tâches prioritaires ont des niveaux de priorité inférieurs aux tâches moins prioritaires), de sorte qu'on puisse extraire efficacement l'une des tâches les plus prioritaires à accomplir, et qu'on puisse insérer efficacement de nouvelles tâches dans l'APE. L'objectif du TP est d'étudier une implémentation possible des APE et de programmer les opérations suivantes :

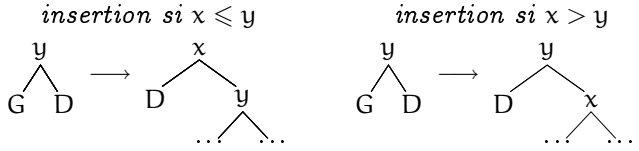
insertion : on donne un APE A et une étiquette x, former un nouvel APE B contenant toutes les étiquettes figurant dans A et l'étiquette x. Si A comporte des répétitions, B devra comporter les mêmes répétitions, donc $\text{taille}(B) = \text{taille}(A) + 1$.

union : on donne une étiquette x et deux APE G, D tels que $\text{taille}(G) \leqslant \text{taille}(D) \leqslant \text{taille}(G) + 1$, former un APE A contenant toutes les étiquettes de G, de D et l'étiquette x (donc $\text{taille}(A) = \text{taille}(G) + \text{taille}(D) + 1$).

extraction à droite : on donne un APE non vide A, retourner l'étiquette x la plus à droite dans A et un APE B contenant toutes les étiquettes de A sauf celle extraite (donc $\text{taille}(B) = \text{taille}(A) - 1$).

extraction de la racine : on donne un APE non vide A, retourner l'étiquette x de la racine et un APE B contenant toutes les étiquettes présentes dans A sauf celle extraite (donc $\text{taille}(B) = \text{taille}(A) - 1$).

Algorithme d'insertion : le cas où $A = \varnothing$ est trivial. Pour $A \neq \varnothing$, comparer l'étiquette x à insérer à celle de la racine de A : y. Placer la plus petite des deux étiquettes à la racine de B, insérer la plus grande dans la branche gauche de A et *permuter* la nouvelle branche droite et la branche gauche de A pour former les branches de B.



Le but de la permutation des branches droite et gauche entre A et B est de maintenir la propriété *3* des APE : si $\text{taille}(D) = \text{taille}(G)$ alors la branche droite de B comporte un nœud de plus que sa branche gauche, et si $\text{taille}(D) = \text{taille}(G) + 1$ alors les deux branches de B ont même taille.

Algorithmes d'extraction de la racine : le cas où l'une au moins des branches de A est vide est trivial. Lorsque ces deux branches sont non vides, on dispose des deux algorithmes suivants pour former B :

Méthode 1 : *soient* G *et* D *les branches gauche et droite de* A *; extraire de* D *l'étiquette la plus à droite, soit* x *cette étiquette et* D' *l'arbre formé des étiquettes de* D *restantes. Former l'arbre* $B = \text{union}(x, D', G)$.

Méthode 2 : *soient* G *et* D *les branches gauche et droite de* A, x *l'étiquette de la racine de* G *et* y *celle de* D. *Soient enfin* G_1, D_1 *les branches gauche et droite de* G. *Extraire la racine* y *de* D, *soit* D' *l'arbre restant. Si* $x \geqslant y$ *former l'arbre* B *de racine* y, *de branche gauche* D' *et de branche droite* G. *Si* $x < y$ *alors former l'arbre* B *de racine* x, *de branche gauche* D' *et de branche droite* $\text{union}(y, G_1, D_1)$.

Comme pour l'algorithme d'insertion, on permute les branches gauche et droite de A lors de la formation de B de façon à garantir la condition *3*.

Implémentation des APE

On se limitera dans le TP à des arbres à étiquettes entières ou réelles, la valeur d'une étiquette codant son niveau de priorité. Les arbres binaires ne sont pas prédéfinis en CAML, mais ils font l'objet d'une bibliothèque séparée. Pour utiliser cette bibliothèque, insérez au début de votre programme les instructions suivantes :

```
load_object "arbres.zo";;
#open "arbres";;
install_printer "print_int_b_arbre";;
```

l'instruction `load_object "arbres.zo"` a pour effet de charger dans l'interpréteur CAML le code compilé contenant (entre autres) la déclaration des arbres binaires telle qu'on l'a vue en cours :

```
type 'a b_arbre = B_vide | B_noeud of 'a * ('a b_arbre) * ('a b_arbre);;
```

Ne pas recopier cette déclaration, elle est incluse dans le fichier `arbres.zo`. L'instruction `#open "arbres"` permet d'utiliser les noms définis dans le module `arbres` sans avoir à les préfixer par `arbres__`. Enfin, l'instruction `install_printer "print_int_b_arbre"` signale à l'interpréteur CAML qu'il doit utiliser la fonction `print_int_b_arbre` (définie dans le module `arbres`) pour afficher les arbres binaires à étiquettes entières sous forme semi-graphique. On aura par exemple :

```
#let a = B_noeud(1, B_noeud(2,B_vide,B_vide),
                    B_noeud(3,B_noeud(4,B_vide,B_vide),
                              B_noeud(5,B_vide,B_vide)));;
a : int b_arbre =   1
                   / \
                  2   3
                      /\
                     4  5

#
```

Écrire les fonctions `insertion`, `union`, `extrait_droite`, `extrait_racine_1` et `extrait_racine_2` décrites dans la présentation. Les fonctions d'extraction retourneront un couple constitué de l'étiquette extraite et de l'arbre des étiquettes restantes.

Contrôle. Utiliser la fonction suivante pour créer un APE à n nœuds dont les étiquettes sont des entiers aléatoires compris entre a et b :

```
let rec random_ape n a b =
  if n = 0 then B_vide
  else let x = a + random__int(b-a+1)
       and y = random_ape (n-1) a b in insertion x y
;;
```

Écrire une fonction `verif` qui prend un arbre binaire en argument et retourne un couple de type `bool*int`, le booléen dit si l'arbre donné est bien un APE, et l'entier donne la taille de cet arbre.

Comparaison expérimentale des deux méthodes d'extraction

Les deux méthodes d'extraction de la racine d'un APE de n nœuds ont la même complexité théorique : $O(\ln n)$, et il n'est pas immédiat de dire si une des méthodes est plus rapide que l'autre. *Vous pouvez faire des paris (non financiers) avant de traiter cette partie du TP.* Dans un tel cas, on peut procéder à une *comparaison*

expérimentale : tirer des APE au hasard et appliquer les deux méthodes d'extraction en comptant pour chaque méthode le nombre d'opérations effectuées. On choisit ici de ne compter que les comparaisons entre étiquettes, les autres opérations (examen d'un arbre, construction d'un nœud à partir d'une étiquette racine d'une branche gauche et d'une branche droite, etc.) étant considérées comme ayant des durées négligeables.

Afin d'obtenir automatiquement le nombre total de comparaisons effectuées, on définit un compteur qui est augmenté d'une unité à chaque fois qu'une comparaison est effectuée :

```
let cmp = ref 0;;              (* compteur de comparaisons *)

let lt = prefix <  ;;          (* renomme les opérations de *)
let le = prefix <= ;;          (* comparaison usuelles      *)
let gt = prefix >  ;;
let ge = prefix >= ;;

let prefix <  a b = incr cmp; lt a b ;; (* et les redéfinit *)
let prefix <= a b = incr cmp; le a b ;;
let prefix >  a b = incr cmp; gt a b ;;
let prefix >= a b = incr cmp; ge a b ;;
```

Saisir ces déclarations, puis revalider les fonctions `extrait_racine_1` et `extrait_racine_2`, de sorte qu'elles utilisent les nouvelles versions des opérateurs <, <=, > et >=. Ceci fait, on obtient le nombre de comparaisons effectuées par les deux méthodes d'extraction par :

```
let a = random_ape 100 0 1000 in
cmp := 0;
let _ = extrait_racine_1 a in
let c1 = !cmp in
cmp := 0;
let _ = extrait_racine_2 a in
let c2 = !cmp in
(c1,c2);;
```

S'inspirer de ce code pour écrire une fonction `moyenne` qui prend deux entiers n et p en arguments, et qui retourne les nombres moyens de comparaisons effectuées par les deux algorithmes d'extraction, les moyennes étant calculées sur p APE aléatoires de taille n.

Une application des APE : la suite de Hamming

Un nombre entier $n \in \mathbf{N}^*$ est appelé *nombre de* Hamming si ses seuls diviseurs premiers éventuels sont $2, 3, 5$. La suite de Hamming est la suite des nombres de Hamming classée par ordre croissant. On remarque que si x est un nombre de Hamming alors $2x, 3x, 5x$ le sont aussi, et que tous les nombres de Hamming à part 1 s'obtiennent en multipliant par 2 ou 3 ou 5 un nombre de Hamming plus petit, ce qui donne l'algorithme suivant pour générer la liste des n premiers nombres de Hamming :

1. *Initialiser un APE avec le nombre 1 et initialiser la liste résultat à* $\varnothing$.
2. *Extraire la racine de l'APE,* x.
3. *Si* x *n'a pas déjà été vu : le placer dans la liste résultat et insérer* $2x$, $3x$, $5x$ *dans l'APE.*
4. *Recommencer les étapes 2 et 3 jusqu'à avoir obtenu* n *nombres.*

Programmer cet algorithme.

Amélioration : soit x un nombre de Hamming extrait de l'APE. Si x est divisible par 5 : $x = 5y$ alors $2x = 5 \times 2y$ a déjà été inséré dans l'APE au moment où l'on extrayait $2y$ (ce qui a eu lieu avant l'extraction de x car $2y < x$), de même $3x = 5 \times 3y$ est aussi présent dans l'APE, donc il est inutile d'insérer ces nombres, seul $5x$ est nouveau. Lorsque x n'est pas divisible par 5 mais l'est par 3 alors par un raisonnement analogue on constate qu'il est inutile d'insérer $2x$ dans l'APE. On se convaincra que si l'on évite ces insertions inutiles, alors chaque nombre de Hamming est inséré au plus une seule fois dans l'APE, donc le test « x non déjà vu » dans l'algorithme précédent devient inutile.

Calcul rapide des nombres de HAMMING

Supposons, pour un certain entier k, disposer de tous les nombres de HAMMING dans l'intervalle $[\![2^{k-1}, 2^k[\![$, rangés dans un APE A_k. Alors on peut former un APE A_{k+1} contenant tous les nombres de HAMMING de l'intervalle $[\![2^k, 2^{k+1}[\![$ de la manière suivante :

1. *Multiplier par 2 tous les nombres figurant dans* A_k.
2. *Insérer dans l'arbre obtenu les nombres de* HAMMING *impairs non encore vus et inférieurs à* 2^{k+1}.

L'étape *1* peut être effectuée en temps constant si l'on conserve dans une variable auxiliaire c_k le coefficient multiplicateur à appliquer à A_k. Il suffit alors de multiplier ce coefficient par 2, ce qui donne c_{k+1}, on n'a pas besoin de parcourir l'arbre A_k. L'étape *2* peut être effectuée en générant dans l'ordre les nombres de HAMMING divisibles uniquement par 3 et 5 à l'aide d'un APE auxiliaire B_k en adaptant la méthode vue à la section précédente (initialiser B à 1, extraire la racine x et insérer 5x si x est divisible par 5, insérer 3x et 5x sinon). Lors de l'insertion d'un nombre de HAMMING x dans A_{k+1}, il faut tenir compte du coefficient multiplicateur, c'est-à-dire insérer en fait le nombre x/c_{k+1}. Ceci impose d'utiliser pour A_k un arbre à étiquettes de type `float` et non `int` car la division ne tombe pas juste (x est un entier impair et $c_k = 2^k c_0$).

Écrire une fonction :

```
transforme :  float b_arbre * int b_arbre * int * int -> float b_arbre * int b_arbre * int * int
```

qui prend en argument un quadruplet constitué des APE A_k, B_k, du coefficient c_k et de la taille t_k de A_k et qui retourne en résultat le quadruplet $(A_{k+1}, B_{k+1}, c_{k+1}, t_{k+1})$. En déduire une fonction `hamming_rapide` qui retourne le n-ème nombre de HAMMING. Cette fonction calculera les quadruplets (A_k, B_k, c_k, t_k) successifs jusqu'à obtenir :

$$t_0 + t_1 + \ldots + t_{k-1} \leqslant n < t_0 + t_1 + \ldots + t_k.$$

Alors le n-ème nombre de HAMMING est celui de rang $n - t_0 - \ldots - t_{k-1}$ dans A_k. On pourra observer le gain en complexité (mesurée en nombre de comparaisons effectuées) apporté par cet algorithme par rapport à l'algorithme naïf consistant à générer les $n+1$ premiers entiers de HAMMING vu à la section précédente.